Java 中间件技术及其应用开发

Java Middleware Development

李伟斌 等编著

中国水利水电出版社
www.waterpub.com.cn

内 容 提 要

本书使用丰富的案例介绍了使用 Java 技术进行中间件编程的方法及技巧，包括 JSP、Java Servlet、JDBC 数据库开发、使用 JavaMail 开发邮件应用程序、RMI（远程方法调用）、Enterprise JavaBeans 开发、使用 CORBA 以及 Java IDL 进行开发、JNDI 编程、JMS 应用开发、Java XML 编程以及 Java 开发 Web 服务等。

本书还介绍了最新的中间件技术，包括数据集成中间件、门户中间件、网格中间件、工作流中间件、RFID 中间件、企业应用集成中间件、数字电视中间件。

本书的内容均为目前的热点和读者所关注的问题，也包括对很多人来说悬而未决的难题。书中的许多案例甚至可以不加修改就应用到开发实践中。

本书适合专业的 Java 程序员阅读，也可以作为正在进行 Java 开发的各类程序员的必备参考书。即使是不擅长这一领域的开发人员，通过详细的实例讲述，也可以循序渐进地掌握本书的内容。

图书在版编目（CIP）数据

Java 中间件技术及其应用开发 / 李华飚等编著. —北京：
中国水利水电出版社，2007
ISBN 978-7-5084-4914-2

Ⅰ. J… Ⅱ. 李… Ⅲ. JAVA 语言—程序设计 Ⅳ. TP312

中国版本图书馆 CIP 数据核字（2007）第 131418 号

书　　名	Java 中间件技术及其应用开发
作　　者	李华飚　等编著
出版　发行	中国水利水电出版社（北京市三里河路 6 号　100044） 网址：www.waterpub.com.cn E-mail：mchannel@263.net（万水） sales@waterpub.com.cn 电话：（010）63202266（总机）、68331835（营销中心）、82562819（万水）
经　　售	全国各地新华书店和相关出版物销售网点
排　　版	北京万水电子信息有限公司
印　　刷	北京市天竺颖华印刷厂
规　　格	787mm×1092mm　16 开本　32.5 印张　788 千字
版　　次	2007 年 9 月第 1 版　2007 年 9 月第 1 次印刷
印　　数	0001—5000 册
定　　价	58.00 元

序

互联网技术的蓬勃发展使得越来越多的企业应用从传统的 C/S（客户/服务器）结构向三层架构转化。中间件的最早定义为：介于应用程序和操作系统之间的软件，它代表一类产品。IT 环境的复杂性使得企业在信息化建设的过程中，需要构建一个能够集门户、集成、运行和管理、安全、开发、部署等众多功能于一体的应用基础平台，中间件在完成这一任务方面发挥了巨大的作用。而从用户角度看，企业所构建的平台越复杂，就越需要一个具有统一性、简便性和扩展性的方案。

一般说来，中间件产品主要应用了以下几类技术：

- **数据访问技术** 提供直接访问不同数据源的手段，而不必改变应用层的程序，比如大家很熟悉的 ODBC、JDBC、JDO、ADO 这些数据标准接口。
- **基于消息的中间件技术（MOM）** MOM 提供一个异步消息传送机制。MOM 中间件在所有需要集成的应用系统内，都要安装 MOM 的 API 调用程序，以把消息递交给中间件进行处理。显然，MOM 的优点是：发送方和接收方不必在线等待（松耦合）。缺点是：会发生过载情况。当然随着产品的不断改进，在负载平衡方面会得到进一步改善。提供 MOM 产品的主要有 IBM MQSeries、微软的 MSMQ。随着 JMS（Java Message Service）的兴起，MOM 产品的应用也将变得更为广泛。相对于下面讲到的应用服务器，它比较“单纯”——专为整合服务。
- **远程过程调用（Remote Procedure Call，RPC）** 远程过程调用也同 MOM 一样利用客户/服务器模式，不同点是，RPC 是同步在线的，即要等待接收方的回复才继续自己的工作。这减少了出错概率，当然由于等待使速度变慢。
- **交易处理监控器（Transaction Processing Monitors）** TP 监控器作为一个中间件，是现代应用服务器的雏形。它主要用于监控和管理客户端及多个后台应用之间的交易状况，考虑负载平衡，将客户的请求映射到各个应用。
- **对象请求代理（Object Request Brokers，ORB）** ORB 是专门替应用转交功能组件的“中介”，是业务功能整合层次的整合中间件，基于 COM、CORBA 及 Java RMI 等标准组件。
- **应用服务器** 应用服务器是一个软件开发平台，除可以开发独立的应用系统外，服务器也集成了各种整合技术。此外，很多厂商还专门开发了许多特制的适配器或连接器，因此，利用应用服务器做整合平台，也是目前很多企业考虑的选项之一。
- **Portal Server** Portal Server 往往是提供一种集成的门户服务。通过可扩展的门户框架集成了对企业信息的标准访问途径，可以将内部和合作伙伴与他们所需要的信息连接起来。

SUN 公司的 Sun ONE（Open Net Environment，开放网络环境）是主流的 Web 技术体系之一。SUN ONE 以 Java 技术为核心，包括 J2SE/J2EE/J2ME，并基于一系列开放和流行标准、技术及协议。SUN ONE 的开放性、可移植性、跨平台和扩展性在进行中间件编程方面得天独

厚，具备无可比拟的优势。

本书介绍了基于 Java 技术的中间件编程，包括如下方面：JSP、Java Servlet、JDBC 数据库开发、使用 JavaMail 开发邮件应用程序、RMI（远程方法调用）、Enterprise JavaBeans 开发、使用 CORBA 以及 Java IDL 进行开发、JNDI 编程、Java XML 编程以及 Java 开发 Web 服务等。

本书加深了对中间件理论体系的介绍，并且增加了热门中间件技术的介绍，包括消息中间件、数据集成中间件、门户中间件、网格中间件、工作流中间件、RFID 中间件、企业应用集成中间件、数字电视中间件，使读者可以更深刻地了解中间件技术原理。

本书详细介绍了如何搭建开发和运行环境，在讲解理论的同时提供了精彩案例，读者按照本书的实例，可以举一反三，融会贯通，理解中间件编程的精髓。

本书的内容均是目前的热点和读者所关注的问题，也包括对很多人来说悬而未决的难题。书中的许多案例甚至可以不加修改就应用到开发实践中。

本书适合专业的 Java 程序员阅读，也可以作为正在进行 Java 开发的各类程序员的必备参考书。即使是不擅长这一领域的开发人员，透过详细的实例讲述，也可以循序渐进地掌握本书的内容。有关本书的技术讨论，可以发邮件到 washing_lhb@sina.com。

本书由李华飚和郭英奎编写。在编写过程中，李旭伟、徐志强、柳振良、唐洪福、彭锦艳、彭传明、史进才、李艳丽、王恒、顾有松、毕宗睿等参与了部分工作。IBM 中国有限公司的褚洪峰提供了许多技术上的支持。顾有松教授和中国科学院周剑平博士对本书的结构提供了许多宝贵的意见。最后，特别感谢北京万水电子信息有限公司孙春亮总编的大力鼓励和支持。

编 者

2007 年 4 月

目　　录

第 1 章　中间件技术导论

计算机技术迅速发展，从硬件技术看，CPU 速度越来越快，处理能力越来越强；从软件技术看，应用程序的规模不断扩大，特别是 Internet 及 WWW 的出现，使计算机的应用范围更为广阔，许多应用程序需在网络环境的异构平台上运行。这一切都对新一代的软件开发提出了新的需求。在这种分布异构环境中，通常存在多种硬件系统平台（如 PC、工作站、小型机等），在这些硬件平台上又存在各种各样的系统软件（如不同的操作系统、数据库、语言编译器等），以及多种风格各异的用户界面，这些硬件系统平台还可能采用不同的网络协议和网络体系结构连接。如何把这些系统集成起来并开发新的应用是一个非常现实而困难的问题。

为解决分布异构问题，人们提出了中间件（middleware）的概念。

1.1　中间件的定义

中间件（middleware）是基础软件的一大类，属于可复用软件的范畴。顾名思义，中间件处于操作系统软件与用户应用软件的中间。中间件在操作系统、网络和数据库之上，应用软件之下，总的作用是为处于自己上层的应用软件提供运行与开发的环境，帮助用户灵活、高效地开发和集成复杂的应用软件。

在众多关于中间件的定义中，人们比较普遍接受的是 IDC 的定义：中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源，中间件位于客户机服务器的操作系统之上，管理计算资源和网络通信。

IDC 对中间件的定义表明，中间件是一类软件，而非一种软件；中间件不仅仅实现互连，还要实现应用之间的互操作；中间件是基于分布式处理的软件，最突出的特点是其网络通信功能。

有人根据这种定义提出一个简单的公式：

中间件=平台+通信

也可以认为，中间件是位于平台（硬件和操作系统）和应用之间的通用服务，如图 1-1 所示，这些服务具有标准的程序接口和协议。针对不同的操作系统和硬件平台，它们可以有符合接口和协议规范的多种实现。

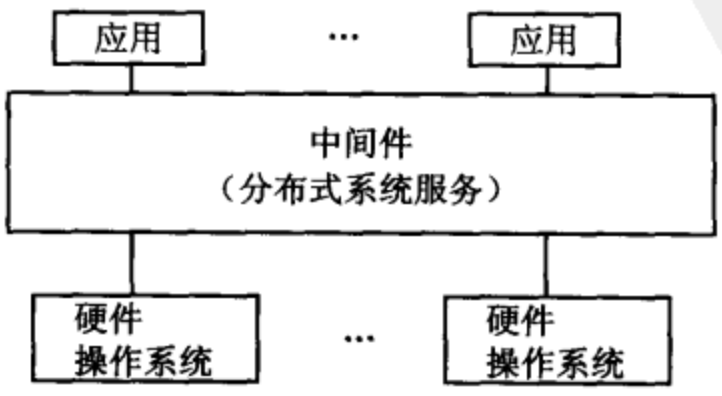


图 1-1　中间件

最早具有中间件技术思想及功能的软件是 IBM 的 CICS，但由于 CICS 不是分布式环境的产物，因此人们一般把 Tuxedo 作为第一个严格意义上的中间件产品。Tuxedo 是 1984 年在当时属于 AT&T 的贝尔实验室开发完成的，但由于当时分布式处理并没有在商业应用上获得像今天这样的成功，Tuxedo 在很长一段时期里只是实验室产品，后来被 Novell 收购，在经过 Novell 并不成功的商业推广之后，1995 年被现在的 BEA 公司收购。尽管中间件的概念很早就已经产生，但中间件技术的广泛运用却是在最近 10 年之中。BEA 公司 1995 年成立后收购 Tuxedo 才成为一个真正的中间件厂商，IBM 的中间件 MQSeries 也是 20 世纪 90 年代的产品，其他许多中间件产品也都是最近几年才成熟起来。国内在中间件领域的起步阶段正是整个世界范围内中间件的初创时期。可以说，在中间件领域，国内的起步时间并不比国外晚多少。

1.2 中间件的分类

按照 IDC 的分类方法，中间件分为六类：

（1）终端仿真/屏幕转换。

（2）数据访问中间件（UDA）。

（3）远程过程调用（RPC）中间件。

（4）消息中间件（MOM）。

（5）交易中间件（TPM）。

（6）对象中间件。

根据 2003 年前后的发展状况，又可以把中间件分为两大类：一类是底层中间件，用于支撑单个应用系统或解决一类问题，包括交易中间件、应用服务器（WAS）、消息中间件、数据访问中间件等；另一类是高层中间件，更多的用于系统整合，包括企业应用集成中间件（EAI Suites）、工作流中间件（Workflow）、门户中间件（Portal）等，它们通常会与多个应用系统打交道，在系统中的层次较高，并大多基于前一类的底层中间件运行。

下面分别介绍各种类型的中间件。

1.2.1 终端仿真/屏幕转换

这种类型的中间件用以实现客户机图形用户接口与已有的字符接口方式的服务器应用程序之间的互操作。它们应用于早期的大型机系统，主要功能是将终端机的字符界面转换为图形界面，目前此类中间件在国内已没有应用市场。

1.2.2 数据访问中间件

数据访问中间件是为了建立数据应用资源互操作的模式，对异构环境下的数据库实现联接或文件系统实现联接的中间件。

数据访问中间件适用于应用程序与数据源之间的互操作模型，客户端使用面向数据库的 API，以直接访问和更新基于服务器的数据源，数据源可以是关系型、非关系型和对象型。这类中间件大都基于 SQL 语句，采用同步通信方式。此类中间件使应用开发更加简单，但如果是透过广域网使用，会带来严重的效率问题，因为在低速网上来回交互 SQL 语句会使通信流量过大，同时对数据压缩、加密带来不便。

1.2.3　远程过程调用中间件

通过远程过程调用机制，开发人员编写客户方的应用，需要时可以调用位于远端服务器上的过程。

远程过程调用（RPC）是一种广泛使用的分布式应用程序处理方法。一个应用程序使用 RPC 来“远程”执行一个位于不同地址空间里的过程，并且从效果上看和执行本地调用相同。事实上，一个 RPC 应用分为两个部分：server 和 client。server 提供一个或多个远程过程；client 向 server 发出远程调用。server 和 client 可以位于同一台计算机，也可以位于不同的计算机，甚至运行在不同的操作系统之上。它们通过网络进行通信。相应的 stub 提供数据转换和通信服务，从而屏蔽不同的操作系统和网络协议。在这里 RPC 通信是同步的。采用线程可以进行异步调用。

在 RPC 模型中，client 和 server 只要具备了相应的 RPC 接口，并且具有 RPC 运行支持，就可以完成相应的互操作，而不必限制于特定的 server。因此，RPC 为 client/server 分布式计算提供了有力的支持。同时，远程过程调用 RPC 所提供的是基于过程的服务访问，client 与 server 进行直接连接，没有中间机构来处理请求，因此也具有一定的局限性。比如，RPC 通常需要一些网络细节以定位 server；在 client 发出请求的同时，要求 server 必须是活动的等等。

1.2.4　交易中间件

交易中间件是专门针对联机交易处理系统而设计的。联机交易处理系统需要处理大量并发进程，处理并发涉及到操作系统、文件系统、编程语言、数据通信、数据库系统、系统管理、应用软件等，是一项相当艰巨的任务，但是工作的难度可以通过采用一个交易中间件来简化。交易中间件就是一组程序模块，用以大大减少开发一个联机交易处理系统所需的编程量。X/OPEN 组织专门定义了分布式交易处理的标准及参考模型，把一个联机交易系统划分成资源管理（RM）、交易管理（TM）和应用（AP）三部分，定义了应用程序、交易管理器以及多个资源管理器之间如何协同工作。资源管理器是指数据库和文件系统，交易管理器可归入交易中间件。交易中间件管理由应用声明和提交的交易，并通过两阶段提交协议等方式保证分布式交易的完整性、控制并发、实现交易路由和均衡负载。

交易中间件理论上相对成熟，功能和性能界定清晰，但基本上只适用于联机交易系统，如银行业务系统、订票系统等。尽管交易信息也是消息，交易中间件也是基于消息的传输，也可支持同步和异步方式，但与消息中间件的定位差距较大，属于一种较专用的中间件。

1.2.5　消息中间件

消息中间件（Message-Oriented Middleware，MOM）指的是利用高效可靠的消息传递机制进行平台无关的数据交流，并基于数据通信来进行分布式系统的集成。通过提供消息传递和消息排队模型，它可以在分布式环境下扩展进程间的通信，并支持多通信协议、语言、应用程序、硬件和软件平台。目前流行的 MOM 中间件产品有 IBM 的 MQSeries、BEA 的 MessageQ 等。消息传递和排队技术有以下三个主要特点：

（1）通信程序可在不同的时间运行。程序不在网络上直接相互通话，而是间接地将消息放入消息队列，因为程序间没有直接的联系，所以它们不必同时运行。消息放入适当的队列时，

目标程序甚至根本不需要正在运行；即使目标程序在运行，也不意味着要立即处理该消息。

（2）对应用程序的结构没有约束。在复杂的应用场合中，通信程序之间不仅可以是一对一的关系，还可以是一对多和多对一方式，甚至是上述多种方式的组合。多种通信方式的结构并没有增加应用程序的复杂性。

（3）程序与网络复杂性相隔离。程序将消息放入消息队列或从消息队列中取出消息来进行通信，以及与此关联的全部活动，比如维护消息队列、维护程序和队列之间的关系、处理网络的重新启动和在网络中移动消息等是 MOM 的任务，程序不直接与其他程序通话，并且它们不涉及网络通信的复杂性。

消息中间件可以既支持同步方式，又支持异步方式。中间件领域目前最热门的技术是异步的消息中间件，异步中间件技术比同步中间件技术具有更强的容错性，在系统故障时可以保证消息的正常传输，因而在过去的两年里增长迅速。异步中间件技术可以分为两类：广播方式和发布/订阅（Publish/ Subscribe）方式。广播方式把消息分发给系统的所有用户。发布/订阅方式可以指定哪种类型的用户可以接收哪种类型的消息。发布/订阅方式由于更加智能有效，事实上已成为异步中间件的非正式标准。

1.2.6 对象中间件

面向对象的技术一直是软件界努力追求的目标，传统的对象技术通过封装、继承及多态提供了良好的代码重用功能。但是这些对象只存在于一个程序中，外面的世界并不知道它们的存在，也无法访问它们。面向对象的中间件就是要解决这些问题，它提供一个标准的构件框架，能使不同厂家的软件通过不同的地址空间、网络和操作系统实现交互访问。该构件的具体实现、位置及所依附的操作系统对客户来说都是透明的。例如，用户通过简单的组装或扩展已有的构件就可以建立一个客户机/服务器结构的信息系统。面向对象的中间件技术的目标就是为软件用户及开发者提供一种应用级的即插即用的互操作性，就像现在使用集成块和扩展板一样。

有关对象中间件的标准相继问世，像 OMG 组织的 CORBA、Microsoft 的 COM 以及 IBM 公司的 SOM 等，这些标准都极大地促进了对象中间件技术的发展，随着面向对象的应用系统的逐渐增长，对象中间件的需求也在逐年加大。对象技术的优势和对象中间件的标准化，促使对象中间件的功能将最终涵盖其他几类中间件的功能而成为中间件产品的主流。

1.2.7 应用服务器

Web 应用服务器是当前被集成商和客户广为接受的中间件种类。这里所说的 Web 应用服务器主要是指基于 Java 和 J2EE/EJB 的应用服务器软件。

国内市场上应用的 Web 应用服务器主要有三类：国外、国内厂家的商业化产品及一些 Open Source（开源）软件。Open Source 软件目前主要在一些小的系统中使用，通常是由有实力的小厂商带进客户的应用中去的。这些软件总的来看还不错，但实际运行中表现的可靠性稍低，而应用开发商通常没有能力去维护。尽管 Web 应用服务器正面临“商品化”的趋势，计算机系统厂商正在把国外的 Web 应用服务器产品捆绑在系统上供客户在一定时间内免费试用，给国内的 Web 应用服务器厂商造成了一定的压力，但国内的 Web 应用服务器产品还是有生存的空间。稳定的系统加上适当的价格（在系统规模大时价格优势更加明显）和良好的服务是国

内厂商的竞争优势。

Web 应用服务器在技术方面的更新主要是对标准的支持、新的开发工具和对 Web 服务的支持等方面。多数情况下，用户在实际开发或应用中并没有感受到这些厂家所津津乐道的差异。大致从 2002 年的下半年开始，国内软件开发商才开始较多地使用 EJB（Enterprise JavaBeans）来构造应用，并在金融、电信等重要行业用 Web 应用服务器来集成已有的关键任务应用。目前国内的中小软件开发商和系统集成商已经开始了解和使用 Web 应用服务器产品，基于 Web 应用服务器开发应用。

1.2.8　企业应用集成中间件

企业应用集成（Enterprise Application Integration）中间件通常指的是企业应用集成服务器。企业应用集成服务器不是一个新的概念，实际上，作为“消息中介”或“消息中介”支撑软件，EAI 服务器在国内早就有应用。

消息中间件、事务处理中间件、Web 应用服务器、门户服务器等中间件产品都有支持集成企业信息系统（EIS）的能力，并在实际的企业信息系统集成中实实在在地发挥着作用。与这些产品不同，EAI 服务器强调的是应用系统之间相互访问与集成的需求与能力，访问是双向的。集成的中心是作为枢纽的 EAI 服务器，枢纽向外的辐射（通过消息中间件等技术）把各个应用系统连接集成在一起。枢纽执行应用之间格式的转换、消息传递的路由选择与控制及传输任务。

EAI 服务器通常是一个有着重量级“价格”的软件系统。软件通常运行在消息中间件之上，可以和 Web 应用服务器等中间件集成。

1.2.9　工作流中间件

许多人都知道以文档为中心的支持文档流转的“工作流软件”。今天被看好的工作流软件定位于支持业务流程的自动化，即能够方便地处理集成。这些工作流软件或者以消息中间件，或者以 Web 应用服务器为底层支撑。建立在消息中间件之上的工作流软件一般都有 Windows 或 UNIX 上的客户端，支持与 Web 应用服务器的集成，并提供使用浏览器获得工作列表、执行流程实例和监控管理工作流的能力。

由于看好工作流产品市场，2002 年，国内不少软件厂商或系统集成商准备或已经投入到了工作流软件产品的开发行列中。从现有的产品来看，这些软件多数是基于 Web 应用服务器的，工作流引擎运行在 Web 应用服务器上，以浏览器作为工作流程中参与人员的操作界面，具备可视化的流程定义工具等必要的功能模块。产品也简单易用，适合于构建一个单一组织内的工作流应用。也有的厂商在开发针对特定行业应用需要的工作流产品。

1.2.10　门户中间件

门户中间件指的是门户服务器（Portal Server）。门户服务器是 Web 应用服务器上的“应用”，是中间件市场上值得关注的新品。尽管有关门户服务器技术的介绍很早就有了，但大牌 IT 厂商的产品基本上都是 2002 年年初或上半年推出的。

基于门户服务器建设企业门户的基本好处是开发商可以利用门户服务器提供的构筑门户应用的基础组件工具 portlet 小程序。应用开发商可以开发很多这样的“门户组件”同时集成

别人开发的“门户组件”来构建企业门户。使用门户的用户可以主动地选择可选的“门户组件”进行个性化的选择，构造自己的“门户”。“门户”可以是对企业后端应用的访问，也可以是自己或别人的网站的一部分。由于用户在向门户描述自己时可能给出了自己的兴趣或爱好，门户网站也可以根据这些信息主动地“推”信息给使用者。

门户服务器在支持个性化和多渠道接入方面特点突出。门户服务器的重要意义还在于它作为 Web 服务的“客户端”的作用。有人这样看待门户技术，称它在 Web 服务时代的重要性就如同 Client/Server 时代的 Windows，因特网时代的浏览器。

1.2.11 安全中间件

安全中间件的产生有着深刻的用户需求。随着网络拓扑结构的日益复杂，许多应用安全软件难以适应复杂多变的网络环境，尽管许多用户花费大量精力为各项应用系统开发安全软件，但仍无法确保安全策略的一致性。面对移动办公的兴起，越来越多的公司开始关注系统的伸缩性和可扩展性。而对于军事、政府等对安全有着特殊要求的用户来说，产生不安全的因素大多是由众多安全产品及其操作系统引起的。在一些用户看来，传统的“防火墙+入侵检测+防病毒产品”等安全产品集合而成的安全模式开始受到挑战，安全产品实质上都是对网络层进行基本防护，而来自企业内部的攻击者根本不受防火墙的监控，入侵者只要躲过入侵检测的过滤，就可以合法访问服务器端口，并在无须认证和授权的情况下浏览服务器的内容和应用。解决以上问题的需求就成为安全中间件发展的动力。

安全中间件是以公钥基础设施（PKI）为核心、建立在一系列相关国际安全标准之上的一个开放式应用开发平台，并对 PKI 基本功能如对称加密与解密、非对称加密与解密、消息摘要、单向散列、数字签名、签名验证、证书认证，以及密钥生成、存储、销毁等进一步扩充，进而形成系统安全服务接口、应用安全服务接口、储存安全服务接口和通信安全服务接口。安全中间件可以跨平台操作，为不同操作系统上的应用软件集成提供方便，满足用户对系统伸缩性和可扩展性的要求。在频繁变化的企业计算机环境中，安全中间件能够将不同的应用程序无缝地融合在一起，使用户业务不会因计算环境的改变而遭受损失。同时，安全中间件屏蔽了安全技术的复杂性，使设计开发人员无须具备专业的安全知识背景就能够构造高安全性的应用。

安全中间件是近年来才逐渐发展起来的，目前涉足该技术的厂商包括 IBM、CA 等。随着电子商务、电子政务的日渐兴起，安全中间件将迎来更为广阔的发展空间。

1.3 中间件、基础件和平台

和中间件相关的 IT 概念还包括基础件、平台等。本节比较这些概念的异同。

1.3.1 中间件和基础件

2002 年 2 月，BEA 公司突破中间件的框架，提出了“基础件”的新理念。BEA 认为，基础件是对中间件的延伸与超越，它以应用服务器为核心，通过合并同类项的方法，把企业中新的需求提炼出来，从而构成了一个集门户、集成、运行和管理、安全、开发、部署等众多技术功能于一体的应用基础平台。如果说中间件满足了企业系统高效、快速响应市场变化的

需求，其特点集中在某一个应用方面，那么，基础件解决的则是应用与应用之间的集成，它的目标是提高整个企业的生产力，降低成本结构和增强响应能力。

基础件理念的提出表明了这样一个思想：打破过去割裂混乱的局面，重新为用户打造和提供应用服务的软件平台。

1.3.2　平台和中间件

“平台”的概念早已有之，是软件系统分层结构思想的具体体现。今天的“平台”具有了新的含义。大体上，可以用下面的公式来描述：

平台=中间件+业务组件

即“平台”中包含中间件，中间件是构造平台的基础；在中间件之上有一层和应用有关联的“业务组件”，以进一步简化应用系统的开发。“平台”是中间件思想的深化，是能够进一步简化应用开发并能让行业专家理解的“高层”的中间件。之所以不把“平台”简单地归纳进“中间件”的门类，有两个主要的原因：其一，这种软件与通用的中间件有很大的区别，可能限定于特定的行业或领域；其二，这种软件基于“通用”的中间件开发，“通用”的中间件是其基础。上面的公式中，“业务组件”可能是跨行业的，也可能限定于特定行业或特定行业的特定类应用。业务组件可能表现为框架性的东西，如跨领域的通用框架（应用框架）、限定于特定领域的领域框架。领域框架采用的是 OMG 的概念。OMG 把领域划分为金融、制造、电信、保健、电子商务、运输等大的方面，而每一方面又细分成几个小的方面；应用框架是跨行业的框架，范围比 OMG 的水平 CORBA 设施还要大。

“平台”的概念早已被国内企业所接受，今天许多软件开发商、系统集成商在开发应用的过程中都或多或少地朝着这个方向努力，以求得最大的可复用性，提高开发效率、降低成本、提高应用软件的可靠性。“平台”基于中间件构建，如果“平台”具有跨同类中间件的能力，则“平台”具有最大的可移植性。对于国内的应用软件开发商和系统集成商而言，“平台”是提高竞争力、提高应用系统的技术含量的重要手段。

1.4　中间件特点及优势

通常意义下，中间件应具有以下的一些特点：满足大量应用的需要；运行于多种硬件和操作系统（OS）平台；支持分布式计算，提供跨网络、硬件和操作系统（OS）平台的透明性的应用或服务的交互功能；支持标准的协议；支持标准的接口。

开发人员通过调用中间件提供的大量 API，实现异构环境的通信，从而屏蔽异构系统中复杂的操作系统和网络协议。

中间件提供客户机与服务器之间的连接服务，这些服务具有标准的程序接口和协议。针对不同的操作系统和硬件平台，它们可以有符合接口和协议规范的多种实现。

由于标准接口对于可移植性和标准协议对于互操作性的重要性，中间件已成为许多标准化工作的主要部分。对于应用软件开发，中间件远比操作系统和网络服务更为重要，中间件提供的程序接口定义了一个相对稳定的高层应用环境，不管底层的计算机硬件和系统软件怎样更新换代，只要将中间件升级更新，并保持中间件对外的接口定义不变，应用软件几乎不需任何修改，从而保护了企业在应用软件开发和维护中的重大投资。

中间件是一种独立的系统软件或服务程序，分布式应用软件借助这种软件在不同的技术之间共享资源。中间件软件管理着客户端程序和数据库或者早期应用软件之间的通信。

中间件在分布式的客户和服务之间扮演着承上启下的角色，如事务管理（也称作交易管理）、负载均衡以及基于 Web 的计算等。

利用这些技术有助于减轻应用软件开发者的负担，使他们利用现有的硬件设备、操作系统、网络、数据库管理系统以及对象模型创建分布式应用软件时更加得心应手。由于中间件能够保护企业的投资，保证应用软件的相对稳定，实现应用软件的功能扩展；同时中间件产品在很大程度上简化了一个由不同硬件构成的分布式处理环境的复杂性，所以它的出现正日益引起用户的关注。

世界著名的咨询机构 Standish Group 在一份研究报告中归纳了中间件的十大优越性。

（1）应用开发：Standish Group 分析了 100 个关键应用系统中业务逻辑程序、应用逻辑程序及基础程序所占的比例；业务逻辑程序和应用逻辑程序仅占总程序量的 30%，而基础程序占了 70%，使用传统意义上的中间件一项就可以节省 25%～60%的应用开发费用。如果以新一代的中间件系列产品来组合应用，同时配合以可复用的商务对象构件，则应用开发费用可节省至 80%。

（2）系统运行：没有使用中间件的应用系统，其初期的资金及运行费用的投入要比同规模的使用中间件的应用系统多一倍。

（3）开发周期：基础软件的开发是一件耗时的工作，若使用标准商业中间件则可缩短开发周期 50%～75%。

（4）减少项目开发风险：研究表明，没有使用标准商业中间件的关键应用系统开发项目的失败率高于 90%。企业自己开发内置的基础（中间件）软件是得不偿失的，项目总的开支至少要翻一倍，甚至会多十几倍。

（5）合理运用资金：借助标准的商业中间件，企业可以很容易地在现有或遗留系统之上或之外增加新的功能模块，并将它们与原有系统无缝集成。依靠标准的中间件，可以将老的系统改头换面成新潮的 Internet/Intranet 应用系统。

（6）应用集成：依靠标准的中间件可以将现有的应用、新的应用和购买的商务构件融合在一起进行应用集成。

（7）系统维护：需要一提的是，基础（中间件）软件的自我开发是要付出很高代价的，此外，每年维护自我开发的基础（中间件）软件的开支则需要最初开发费用的 15%～25%，每年应用程序的维护开支也还需要当初项目总费用的 10%～20%左右。而在一般情况下，购买标准商业中间件每年只需付出产品价格的 15%～20%的维护费，当然，中间件产品的具体价格要依据产品购买数量及哪一家厂商而定。

（8）质量：基于企业自我建造的基础（中间件）软件平台上的应用系统，每增加一个新的模块，就要相应地在基础（中间件）软件之上进行改动。而标准的中间件在接口方面都是清晰和规范的。标准中间件的规范化模块可以有效地保证应用系统质量及减少新旧系统维护开支。

（9）技术革新：企业对自我建造的基础（中间件）软件平台的频繁更新是极不容易实现的（不实际的）。而购买标准的商业中间件，则对技术的发展与变化可以放心，中间件厂商会责无旁贷地把握技术方向和进行技术革新。

（10）增加产品吸引力：不同的商业中间件提供不同的功能模型，合理使用，可以让你的应用更容易增添新的表现形式与新的服务项目。从另一个角度看，可靠的商业中间件也使得企业的应用系统更完善，更出众。

具体地说，中间件屏蔽了底层操作系统的复杂性，使程序开发人员面对一个简单而统一的开发环境，减少程序设计的复杂性，将注意力集中在自己的业务上，不必再为程序在不同系统软件上的移植而重复工作，从而大大减少了技术上的负担。

中间件带给应用系统的不只是开发的简单、开发周期的缩短，也减少了系统的维护、运行和管理的工作量，还减少了计算机总体费用的投入。Standish Group 的调查报告显示，由于采用了中间件技术，应用系统的总建设费用可以减少 50%左右。在网络经济大发展、电子商务大发展的今天，从中间件获益的不只是 IT 厂商，IT 用户同样是赢家，并且是更有把握的赢家。

中间件作为新层次的基础软件，其重要作用是将不同时期、在不同操作系统上开发的应用软件集成起来，彼此之间像一个天衣无缝的整体协调工作，这是操作系统、数据库管理系统本身做不到的。中间件的这一作用，在技术不断发展之后，使以往在应用软件上的劳动成果仍然物有所用，节约了大量的人力、财力投入。

1.5　小结

本章介绍了中间件的定义以及中间件的分类。后面章节将介绍针对几种主要中间件类型的开发。

第 2 章　应用服务器概述

应用服务器并非传统意义上的软件，而是一个可以提供通过 Internet 来实施电子商务和企业计算的平台，所以有人称之为“Internet 上的操作系统”。本章介绍多层应用体系结构，以及应用服务器的功能。

2.1　传统的应用体系结构

传统的应用体系结构主要有两种：C/S（Client/Server）模式与 B/S（Browser/Server）模式。

2.1.1　C/S（客户端/服务器）模式

C/S（Client/Server）结构，即大家熟知的客户机和服务器结构，如图 2-1 所示。它是软件系统体系结构，通过它可以充分利用两端硬件环境的优势，将任务合理分配到 Client 端和 Server 端来实现，降低了系统的通信开销。目前许多应用软件系统都是 Client/Server 形式的两层结构，由于现在的软件应用系统正在向分布式的 Web 应用发展，Web 和 Client/Server 应用都可以进行同样的业务处理，应用不同的模块共享逻辑组件；因此，内部的和外部的用户都可以访问新的和现有的应用系统，通过现有应用系统中的逻辑可以扩展出新的应用系统。这也正是目前应用系统的发展方向。

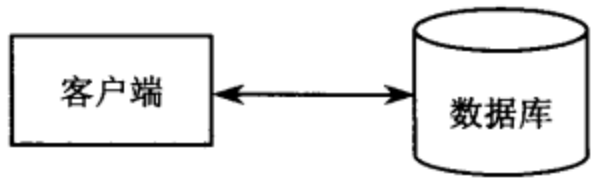


图 2-1　C/S 体系结构

传统的 C/S 体系结构虽然采用的是开放模式，但这只是系统开发一级的开放性，在特定的应用中无论是 Client 端还是 Server 端都还需要特定的软件支持。由于没能提供用户真正期望的开放环境，C/S 结构的软件需要针对不同的操作系统开发不同版本的软件， 加之产品的更新换代十分快，已经很难适应百台电脑以上局域网用户同时使用。

C/S 体系结构软件的优势与劣势如下：

（1）服务器端运行负荷较轻。最简单的 C/S 体系结构的数据库应用由两部分组成，即客户应用程序和数据库服务器程序。二者可分别称为前台程序与后台程序。运行数据库服务器程序的机器，也称为服务器端。一旦启动服务器程序，就随时等待响应客户程序发来的请求；客户应用程序运行在用户自己的电脑上，对应于数据库服务器，可称为客户端，当需要对数据库中的数据进行任何操作时，客户程序就自动地寻找服务器程序，并向其发出请求，服务器程序根据预定的规则做出应答，送回结果，服务器端运行负荷较轻。

（2）数据的存储管理功能较为透明。在典型的 C/S 数据库应用中，数据的存储管理功能是由服务器程序独立进行的。并且通常把那些不同的（不管是已知还是未知的）前台应用所不能违反的规则，在服务器程序中集中实现，例如访问者的权限、编号不准重复这样的规则。所有这些，对于工作在前台程序上的最终用户，是“透明”的，他们无须过问（通常也无法干涉）背后的过程，就可以完成自己的一切工作。在客户/服务器架构的应用中，前台程序不

是“瘦客户端”，麻烦的事情都交给了服务器和网络。在 C/S 体系下，数据库不能真正成为公共、专业化的仓库，它受到独立的专门管理。

（3）C/S 体系结构的劣势是高昂的维护成本且投资大。首先，采用 C/S 体系结构，要选择适当的数据库平台来实现数据库数据的真正“统一”，使分布于两地的数据同步完全交由数据库系统去管理，但逻辑上两地的操作者要直接访问同一个数据库才能有效实现。有这样一些问题，如果需要建立“实时”的数据同步，就必须在两地间建立实时的通信连接，保持两地的数据库服务器在线运行，网络管理工作人员既要对服务器维护管理，又要对客户端维护和管理，这需要高昂的投资和复杂的技术支持，维护成本很高，维护任务量大。

其次，传统的 C/S 体系结构的软件需要针对不同的操作系统开发不同版本的软件，由于产品的更新换代十分快，代价高和低效率已经不适应工作需要。在 JAVA 这样的跨平台语言出现之后，B/S 体系结构更是猛烈冲击 C/S，并对其形成威胁和挑战。

2.1.2　B/S（浏览器/服务器）模式

B/S（Browser/Server）结构即浏览器和服务器结构。在 B/S 体系结构系统中，用户通过浏览器向分布在网络上的许多服务器发出请求，服务器对浏览器的请求进行处理，将用户所需信息返回到浏览器。B/S 结构简化了客户机的工作，客户机上只需配置少量的客户端软件。服务器将担负更多的工作，对数据库的访问和应用程序的执行将在服务器上完成。浏览器发出请求，而其余如数据请求、加工、结果返回以及动态网页生成等工作全部由 Web Server 完成。实际上 B/S 体系结构是把二层 C/S 结构的事务处理逻辑模块从客户机的任务中分离出来，由 Web 服务器单独组成一层来负担其任务，这样客户机的压力减轻了，把负荷分配给了 Web 服务器。这种三层体系结构如图 2-2 所示。

图 2-2　B/S 体系结构

这种结构不仅把客户机从沉重的负担和不断对其提高的性能的要求中解放出来，也把技术维护人员从繁重的维护升级工作中解脱出来。由于客户机把业务逻辑处理工作分给了 Web 服务器，使客户机一下子“苗条”了许多，不再负责处理复杂计算和数据访问等关键事务，只负责显示部分，所以维护人员不再为程序的维护工作奔波于每个客户机之间，而把主要精力放在 Web 服务器上程序的更新上。这种三层结构在层与层之间相互独立，任何一层的改变不会影响其他层的功能。

经过近一两年的应用，B/S 体系结构也暴露出了许多不足之处，具体表现在以下几个方面：

（1）由于浏览器只是为了进行 Web 浏览而设计的，当其应用于 Web 应用系统时，许多功能不能实现或实现起来比较困难。比如通过浏览器进行大量的数据输入，或进行报表的应答都是比较困难和不便的。

（2）复杂的应用构造困难。虽然可以用 ActiveX、Java 等技术开发较为复杂的应用，但是相对于发展已非常成熟的 C/S 的一系列应用工具来说，这些技术的开发复杂，并且没有完

全成熟的技术可供使用。

（3）HTTP 可靠性低有可能造成应用故障，特别是对于管理者来说，采用浏览器方式进行系统的维护是非常不安全与不方便的。

（4）Web 服务器成为对数据库的惟一的客户端，所有对数据库的连接都通过该服务器实现。Web 服务器同时要处理客户请求以及与数据库的连接，当访问量大时，服务器端负载过重。

（5）由于业务逻辑和数据访问程序一般由 JavaScript、VBScript 等嵌入式小程序实现，分散在各个页面里，难以实现共享，给升级和维护也带来了不便。同时由于源代码的开放性，使得商业规则很容易暴露，而商业规则对应用程序来说则是非常重要的。

2.2 多层应用体系结构

随着中间件与 Web 技术的发展，三层或多层分布式应用体系越来越流行。在这种体系结构中，客户端只存放表示层软件，应用逻辑（包括事务处理、监控、信息排队、Web 服务等）采用专门的中间件服务器，后台是数据库。在多层分布式体系结构中，系统资源被统一管理和使用，用户可以通过网络门户（portal）透明地使用整个网络资源。

在多层体系结构中，各层次按照以下方式进行划分，实现明确分工。

- 瘦客户：提供简洁的人机交互界面，完成数据的输入/输出。
- 业务服务（中间层）：完成业务逻辑，实现客户与数据库对话的桥梁。同时，在这一层中，还应实现分布式管理、负载均衡、Fail/Recover、安全隔离等。
- 数据服务：提供数据的存储服务。一般就是数据库系统。

多层分布式体系结构主要有如下特点：

（1）安全性：中间层隔离了客户直接对数据库服务器的访问，保护了数据库的安全。

（2）稳定性：对于要求全天候工作的业务系统，多层分布式体系结构提供了更可靠的稳定性。中间层缓冲了客户端与数据库的实际连接，使数据库的实际连接数量远小于客户端应用数量。当然，连接数越少，我们的数据库系统就越稳定。Fail/Recover 机制能够在一台服务器当机的情况下，透明地把客户端工作转移到其他具有同样业务功能的服务器上。

（3）易维护：由于业务逻辑在中间服务器，当业务规则变化后，客户端程序基本不做改动。

（4）快速响应：通过负载均衡以及中间层缓存数据能力，可以提高对客户端的响应速度。

（5）系统扩展灵活：基于多层分布式体系结构，当业务增大时，可以在中间层部署更多的应用服务器，提高对客户端的响应，而所有变化对客户端透明。

分布式多层体系结构的开发主要考虑三方面的技术，首先是开发环境，开发人员需要一种创建新组件、并将已有组件加以集成的开发环境；其次是应用程序的集成，开发人员需要集成各种应用程序，以创建出更强大的应用；第三是应用程序的配置，分布式多层体系结构的开发需要配置平台的支持，以便在用户剧增时能有效地扩展，并保持系统的稳定。

目前多层分布应用的开发，比较重要的有两种规范，即 COM+和 CORBA。其中 COM+主要用于 Windows 平台，CORBA 则提供跨平台的能力。同时，随着分布式应用的发展，旧的硬件/软件平台的不断更新，跨硬件平台、网络环境、操作系统以及跨不同数据库的应用系统不断出现，使传统的开发工具越来越陷入尴尬境地，因此中间件应运而生。

2.3　J2EE 应用体系结构

目前，Java 2 平台有 3 个版本，分别是适用于小型设备和智能卡的 Java 2 平台 Micro 版（Java 2 Platform Micro Edition，J2ME）、适用于桌面系统的 Java 2 平台标准版（Java 2 Platform Standard Edition，J2SE）、适用于创建服务器应用程序和服务的 Java 2 平台企业版（Java 2 Platform Enterprise Edition，J2EE）。

J2EE 是一种利用 Java 2 平台来简化企业解决方案的开发、部署和管理相关的复杂问题的体系结构。J2EE 技术的基础就是核心 Java 平台或 Java 2 平台的标准版，J2EE 不仅巩固了标准版中的许多优点，例如“编写一次、随处运行”的特性、方便访问数据库的 JDBC API、CORBA 技术以及能够在 Internet 应用中保护数据的安全模式等，同时还提供了对 EJB（Enterprise JavaBeans）、Java Servlets API、JSP（Java Server Pages）以及 XML 技术的全面支持。其最终目的就是成为一个能够使企业开发者大幅缩短投放市场时间的体系结构。

J2EE 体系结构提供中间层集成框架来满足无须太多费用而又需要高可用性、高可靠性以及可扩展性的应用的需求。通过提供统一的开发平台，J2EE 降低了开发多层应用的费用和复杂性，同时提供对现有应用程序集成强有力的支持，完全支持 Enterprise JavaBeans，有良好的向导支持打包和部署应用，添加目录支持，增强了安全机制，提高了性能。

J2EE 平台使用了一个多层的分布式的应用程序模型。应用程序的逻辑根据其实现的不同功能被封装到组件中，组成 J2EE 应用程序的大量应用程序组件根据在其所属的多层的 J2EE 环境中所处的层被安装到不同的机器中。图 2-3 表示两个多层的 J2EE 应用程序根据下面的描述被分为不同的层，包含了以下四个部分：

（1）运行在客户端机器的客户层组件。
（2）运行在 J2EE 服务器中的 Web 层组件。
（3）运行在 J2EE 服务器中的商业层组件。
（4）运行在 EIS 服务器中的企业信息系统（EIS）层软件。

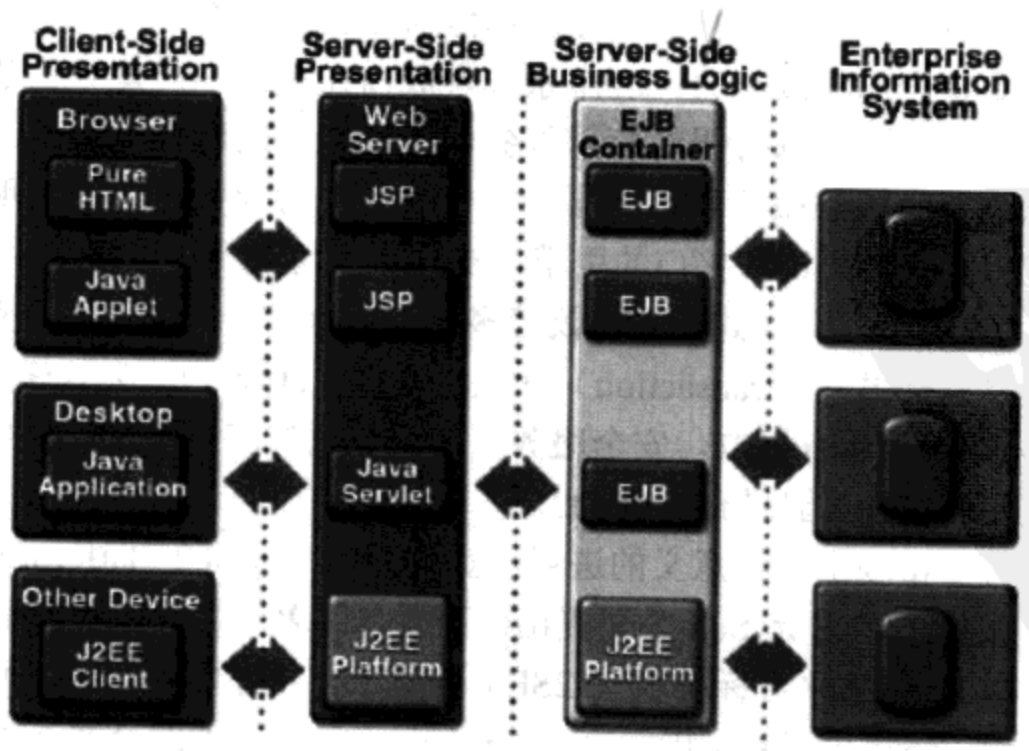


图 2-3　J2EE 体系结构

J2EE 应用程序由组件组成。一个 J2EE 组件就是一个自带功能的软件单元，它随同相关的类和文件被装配到 J2EE 应用程序中，并实现与其他组件的通信。J2EE 规范是这样定义 J2EE 组件的：

（1）客户端应用程序和 applet 是运行在客户端的组件。

（2）Java Servlet 和 Java Server Pages（JSP）是运行在服务器端的 Web 组件。

（3）Enterprise JavaBean（EJB）组件（enterprise bean）是运行在服务器端的商业软件。

J2EE 组件由 Java 编程语言写成，并和用该语言写成的其他程序一样进行编译。J2EE 组件和“标准的”Java 类的不同点在于：它被装配在一个 J2EE 应用程序中，具有固定的格式并遵守 J2EE 规范；它被部署在产品中，由 J2EE 服务器对其进行管理。

2.4 应用服务器

目前在 Internet/Intranet/Extranet 环境中，企业级应用系统大多采用三层或多层应用模式。为了方便开发、部署、运行和管理基于多层结构的应用，需要以网络和分布式计算的底层技术为基础，构建一个完整的应用框架，提供相应的支撑平台作为多层应用的基础设施，这一支撑平台的关键就是位于中间层的应用服务器。应用服务器是一个创建、部署、运行、集成和维护多层分布式企业级应用的平台。如果应用服务器与 Web 服务器相结合，或者包含了 Web 服务器的功能，则称之为 Web 应用服务器。

应用服务器并非传统意义上的软件，而是一个可以提供通过 Internet 来实施电子商务的平台，所以有人称之为“Internet 上的操作系统”。

在企业应用中，应用服务器可以提供如下好处：提高企业应用开发的有效性，保障业务逻辑和组件的重用性；提高企业应用的性能，如高运行性能和响应时间、可伸缩性、可靠性等；使企业应用更易于监控和管理，降低系统维护和升级成本。由于应用服务器的重要作用和关键地位，它已经成为当今业界的一个热点。

作为企业级应用的解决方案，应用服务器应当提供：①加快开发过程、确保开发质量，促使应用快速进入市场的开发能力；②促使应用能够以灵活而有效方式运行的部署能力；③与各种后端系统有效整合的集成能力。应用系统集成商在帮助客户建立、部署和运行企业应用时所需要的灵活性和功能，都源于对应用服务器各种能力的组合与运用。具体来说，应用服务器的功能可以划分为核心服务和扩展服务。核心服务为业务逻辑的实现提供支持；对应用服务器的管理能力；提供协议和接口的引擎，支持通信协议（如 HTTP、IIOP）、数据库互联标准（如 JDBC、ODBC）、分布式计算协议等多种工业标准。扩展服务为支持高端应用而应当提供的功能，主要包括事务（transaction）处理，集群（cluster），失效恢复（fail over），负载均衡（load balancing），缓存机制，安全服务，与企业已有应用系统的集成能力，开发有效性（与应用程序开发环境和工具的结合能力）。

近年在应用服务器市场上最具意义的进展，就是 J2EE（Java 2 Platform Enterprise Edition）的出现。前面已经提到，J2EE 是 Sun 公司提出的开发、部署、运行和管理基于 Java 分布式应用的标准平台。它以 Java 2 平台标准版（J2SE）为基础，继承了标准版的许多优点（如“编写一次，随处运行”），还提供了对 EJB、Java Servlet、JSP 等技术的全面支持。J2EE 使用 EJB Server 作为商业组件的部署环境，在 EJB Server 中提供了分布式计算环境中组件需要的服务，

例如组件生命周期的管理、数据库连接的管理、分布式事务的支持、组件的命名服务等。

J2EE 用于实现应用服务器有其优势，它可以利用 Java 语言自身具有的跨平台性、可移植性、对象特性、内存管理等方面的性能，为应用服务器的实现提供一个完整的底层框架。J2EE 中定义的各种服务，包括 JSP 和 Servlet 容器、EJB 容器、JDBC、JNDI（名字目录服务）、JTS/JTA（事务服务）、JMS（消息服务）等，也分别为应用服务器提供了各种支持。实现商业逻辑的 EJB 组件可以更加高效地运行在应用服务器中，用户可以通过 Java Servlet 或者 JSP 调用运行在 EJB Server 中的 EJB，也可以通过 IIOP 直接访问运行在 EJB Server 中的组件。除了应用服务器的基本特性以外，J2EE 应用服务器还应实现：支持 Java 编程的工业标准，包括 EJB、JDBC、JNDI、RMI-IIOP、JCA、JTS/JTA 等；能够与业界主要的 IDE（如 Borland JBuilder、VisualCafe 等）集成；与标准的 Java 操作平台兼容，如 Sun、IBM 等系统平台；使用完全的 Java 语言编码实现，保证良好的可移植性和支持 Java 的语言特性。

2.5　小结

本章主要介绍了 J2EE 多层应用体系结构以及应用服务器。后面章节将介绍应用服务器的安装配置，以及应用服务器编程。

第 3 章　准备上手

本章介绍在进行应用服务器编程之前要做的准备工作，包括开发环境的搭建、运行环境的搭建。

3.1　开发环境的搭建

要开发一个 JSP 程序，就必须选择好 JSP 的开发工具。自从 JSP 的技术规范发布以后，市场上出现了不少的 JSP 开发工具及支持 JSP 的应用程序服务器。下面将介绍一些实践中比较常用的开发工具。

现在的开发工具很多，开发 Java 程序的如 Borland 公司的 JBuilder，其他的有 Web 服务器厂商开发的编程工具，像 BEA 的 WebLogic Workshop、Allaire Jrun Studio，IBM 的 IBM WebSphere Studio 和 VisualAge For Java 等。对于初学者来说，使用这些工具并没有什么很大的好处，而且使用工具生成的一些代码有时更让我们费解，因此建议还是使用一些文本编辑器。现在流行的能识别 Java、JSP 语法的文本编辑器有 EditPlus 和 UltraEdit。Web 的开发离不开界面的设计，而 HTML 的开发工具大概就只有 Dreamweaver 一枝独秀了。

Dreamweaver MX 的界面如图 3-1 所示，它的下载地址为 http://www.macromedia.com/。而 EditPlus 2.0 的界面如图 3-2 所示，它的下载地址为 http://www.editplus.com。

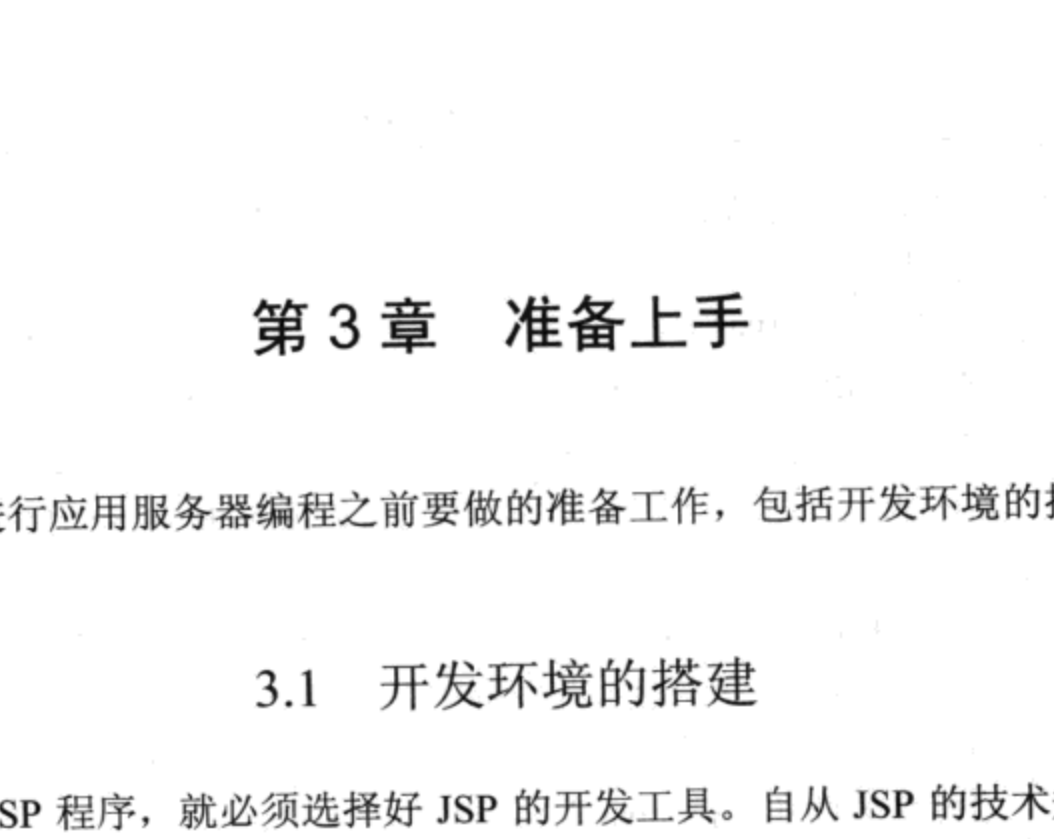


图 3-1　Dreamweaver MX 的界面

Dreamweaver 和 EditPlus 都是 Windows 下的应用程序。直接执行它们的安装程序，按照安装向导的指引就可以顺利地完成安装。

图 3-2　EditPlus 的界面

JBuilder（目前常用版本是 9.0，界面如图 3-3 所示）是一种纯 Java 的产品，其中一个最大的优点是提供了一个开发 Java 程序的集成环境，而且功能非常强大。JBuilder 对 EJB 具有良好的支持，而且可以很方便地和主流的 J2EE 服务器相结合，这些后边的章节会详细介绍。但是，由于这个工具是用纯 Java 语言编写的，所以运行起来特别耗内存，并不是一般的机器都能流畅运行的。JBuilder 对 JSP 的支持不是很理想，如果你只想做 JSP、Servlet 的开发，建议还是使用文本编辑器比较简便。

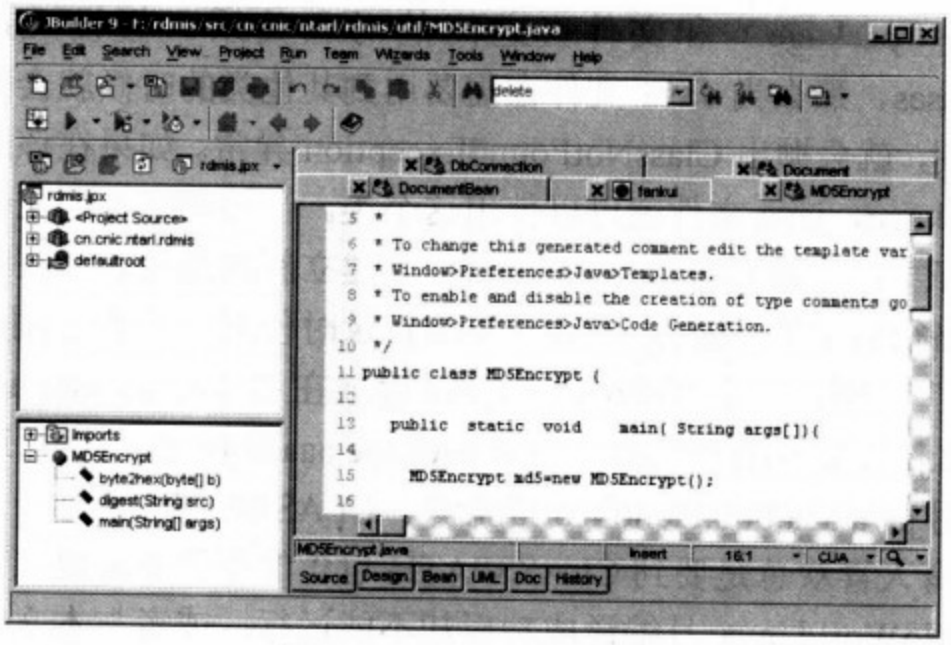


图 3-3　JBuilder 9 的界面

如果使用 JBuilder 或 VisualAge for Java 等集成开发环境（IDE）进行开发的话，直接安装该工具即可。因为编辑和编译运行都是集成在工具内部的，因而使用起来会更加方便。究竟使用哪种方式，读者可以根据自己的习惯进行选择。至此，基本的 Java 程序的编辑工具环境就搭建完成了。

3.2　运行环境的搭建

3.2.1　Java 运行环境的搭建

要想开发一个 JSP 程序，首先必须配置好 JSP 的运行环境。在众多的操作系统中，Windows

在桌面应用的统治地位至今还没有被打破，大部分用户都是在 Windows 环境下开发程序的。即便是不运行在 Windows 平台的软件，好多也是在 Windows 下开发，然后再上传到 Linux、UNIX 服务器上的。下面就针对在 Windows、Linux 操作系统下配置 JSP 运行环境的问题做详细说明。

要编译和调试运行 Java 程序，首先需要安装 JDK，可以从 Sun 公司的站点 java.sun.com 下载 JDK，最新稳定版本是 1.4.1。当然需要根据使用的平台的不同来下载不同的版本。在 Windows 平台下，下载 j2sdk-1_4_1-windows-i586.exe 后运行安装程序，选择安装路径后将会自动完成安装。

然后是环境变量的配置。在不同的操作系统下配置环境变量的方法有些不同，但本质都是一样。它的作用是让应用程序可以方便快捷地找到所需要的路径。

配置 Java 运行环境主要用到的环境变量有 3 个。

- JAVA_HOME ：就是 Java 的安装路径。
- PATH：PATH 是 Windows 固有的，要加上 JAVA_HOME\bin 目录，这样在运行 javac、java 等命令时就不用输入很长的路径了。
- CLASSPATH：CLASSPATH 是运行 Java 非常重要的一个环境变量，Java 在编译和运行应用程序时都要通过它去找到需要的类文件。放在 CLASSPATH 里的一般是路径或 jar 文件，如果是路径的话就是说这个路径下的类可以使用。还要考虑到 Java 文件的包（package）和文件夹之间的对应关系。假设 CLASSPATH 中只有 C:\java\classes，那么当 Java 文件在需要用到其他类的时候就会到这个路径下去找。如果找不到，就会抛出 ClassNotFoundException 异常。如果在该路径下有需要的类而且没有 package，java 文件就可以使用这个类。当然把许多的文件放在一个路径下并不是一个好主意。需要在 C:\java\classes 下建立新的路径时，在新路径下的类文件就必须以该路径名作为 package。建立多级目录时使用“.”来分割包的路径，而不是用“/”或“\”。假设一个 HelloWorld.java 存放在 C:\java\classes\aaa\bbb\下，那么这个 HelloWorld.java 文件的开头必须包含 package aaa.bbb; 然后运行这个 HelloWorld 程序就必须使用 java aaa.bbb.HelloWorld 命令。CLASSPATH 还有一个很重要的元素就是“.”。没有人喜欢总是要到 CLASSPATH 包含的路径下去建立文件，当你建立一个正确的 HelloWorld.java 且编译成功后却不能运行，或者与本文件在同一路径下编译程序时报告找不到该类的错误，就是因为你的 CLASSPATH 里没有包含“.”。说起来很简单，事实证明很多初学 Java 的人都遇到过这种错误，并且迷惑不解。还有就是 jar 文件，其实 Java 为 jar 做的就是把一些文件和文件夹进行了压缩。当 Java 程序在用到其中的类时，直接到压缩文件里去找，而不是路径，也可以理解为 Java 在使用它时自己进行了解压缩。你可以用 Java 自带的 jar 程序生成或解压 jar 文件，也可以直接使用 Winzip 应用程序。例如前面提到过的 HelloWorld.java，我们到 C:\java\classes 路径下把 aaa 文件夹用 Winzip 压缩成 aaa.zip(这个文件名可以随便起，不像 package 似的要求必须和路径名相同），只要把 aaa.zip 放到 CLASSPATH 下就可以在任何路径下运行 java aaa.bbb.HelloWorld 命令了。

1. 在 Windows 2000 中设置环境变量

鼠标右击 My computer→Advanced→Environment Variables...，如图 3-4 所示。出现如图 3-5

所示的对话框，单击 New 按钮。

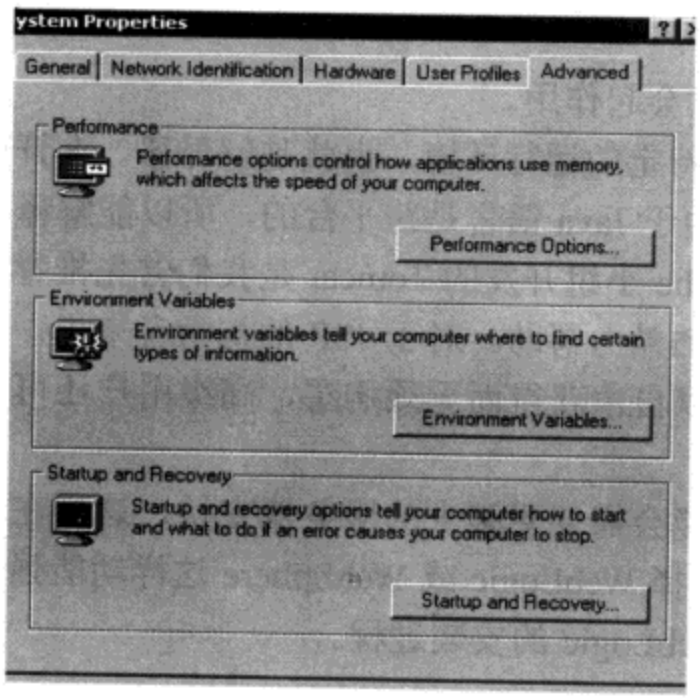


图 3-4　设置环境变量

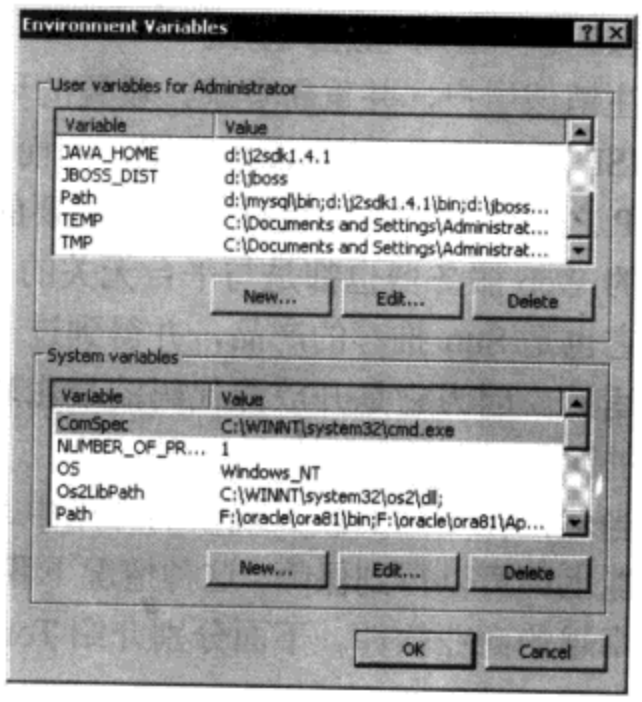


图 3-5　添加环境变量

在如图 3-6 所示的对话框中分别输入如下环境变量：

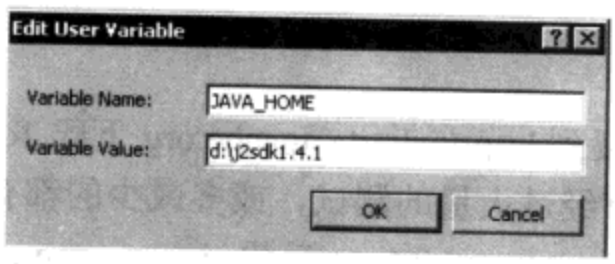


图 3-6　添加环境变量

```
JAVA_HOME       C:\j2sdk1.4.1
Path          C:\j2sdk1.4.1\bin
```

设置完成后要注销该用户，并重新登录，环境变量才会起作用。

2. 在 Windows 98 中设置环境变量

使用文本编辑器打开 C:\autoexec.bat 文件，并在后边添加：

```
SET JAVA_HOME=c:\j2sdk1.4.1
SET PATH=%PATH%;C:\j2sdk1.4.1\bin
```

然后，重新启动 Windows 98。

3. 在 Linux 下搭建 Java 运行环境

把下载的 j2sdk-1_4_1-linux-i586.bin 文件放到/usr/local 目录下。使用以下命令来改变文件属性，使它可以被执行：

```
chmod 0755 j2sdk-1_4_1-linux-i586.bin
```

然后运行该程序：

```
./ j2sdk-1_4_1-linux-i586.bin
```

该命令运行的结果将生成 j2sdk-1_4_1-linux-i586.tar.gz 文件。使用以下命令解压缩该文件：

```
tar -xzvf j2sdk-1_4_1-linux-i586.tar.gz
```

该文件解压缩后生成目录 j2sdk-1_4_1。

4. 在 Linux 下设置环境变量

输入命令 vi /etc/profile，并在该文件后面添加以下内容：

```
JAVA_HOME=/usr/local/j2sdk-1_4_1
PATH=$PATH: /usr/local/j2sdk-1_4_1/bin
export JAVA_HOME PATH
```

注销该用户，并重新登录，设置的环境变量才会起作用。

JSP 是基于 Web 的 Java 应用，因而它需要有特定的运行环境，也就是解释器。在开始学习 JSP 之前，首先构造运行 JSP 所需要的环境。由于 Java 语言是跨平台的，所以能解释 Java 语言的 Web 服务器也都是与平台无关的。由 Apache 小组开发的 Tomcat 是我们首先推荐的。同时它也是 Sun 推荐的产品，并得到过 Sun 公司的鼎力帮助。对于 JSP 的初学者来说，它是最合适的。因为它是开放源代码的自由软件，可以自由获得而无须购买，高级用户还可以通过它的源代码进行更深入的学习。

当然，在实际的较大规模的应用中，我们可能会需要更强大的服务器支持，这样在功能和性能上都可以得到保障。这种情况下我们应该选择 WebLogic 或 WebSphere 这样功能强大和性能卓越的商业软件。下面分别介绍 Tomcat 和 WebLogic 的安装过程。

3.2.2 Tomcat 的安装

可以说 Tomcat 的安装非常简单，下面分别介绍使用最广泛的 Windows 和 Linux 平台上 Tomcat 的安装。

1. Windows 平台

（1）得到 Tomcat。首先要到 http://jakarta.apache.org 下载 Tomcat。Tomcat 5.0 已经有试用版了，但是新版本一般都没有经过大量的测试，或多或少的都有一些问题。建议还是使用稳定的版本，这里使用的是 tomcat-4.0.6。下载 jakarta-tomcat-4.0.6.zip，然后解压缩到 C:\java\jakarta-tomcat-4.0.6 文件夹中。

（2）设置环境变量。环境变量的设置方法前面已经介绍过，这里就不详细说明了。从 Tomcat 4.0 开始，Tomcat 已经不依赖环境变量中的 CLASSPATH 了。这里的设置只是为了我们编写的 Java 程序可以顺利编译。

```
set CATALINA_HOME= C:\java\jakarta-tomcat-4.0.6
set CLASSPATH=.; C:\java\jakarta-tomcat-4.0.6\common\servlet.jar
```

设置 Tomcat 的 CLASSPATH，需要编辑 CATALINA_HOME\bin\setclasspath.bat 文件。

（3）运行 Tomcat。在 Windows 98 环境下需要做如下操作：右击 startup.bat，将内存的初始环境由自动替换改为 2816，如图 3-7 所示。

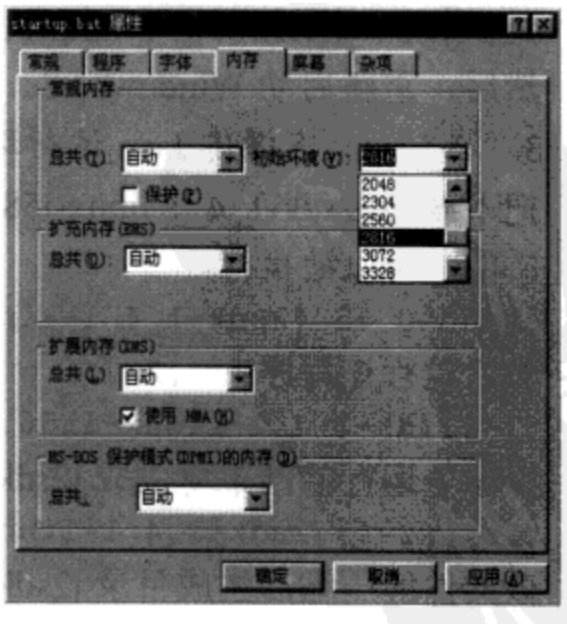


图 3-7 startup.bat 内存的设置

进入 C:\java\jakarta-tomcat-4.0.6\bin 文件夹，双击 startup.bat 或进入命令行方式。或者在 DOS 状态下进入 C:\java\jakarta-tomcat-4.0.6\bin 目录，键入 startup 并回车。Tomcat 启动后如图 3-8 所示。

打开 IE 并在地址栏中输入 http://localhost:8080，出现如图 3-9 所示的页面，这表明 Java、Jsp 和 Servlet 环境已经配置好了。

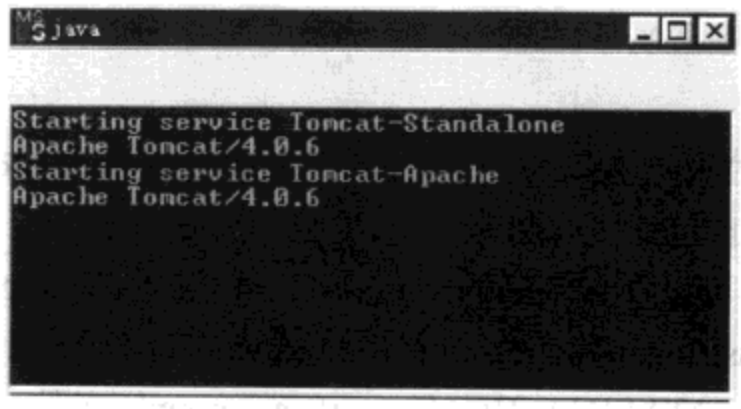


图 3-8　在 Windows 中启动 Tomcat

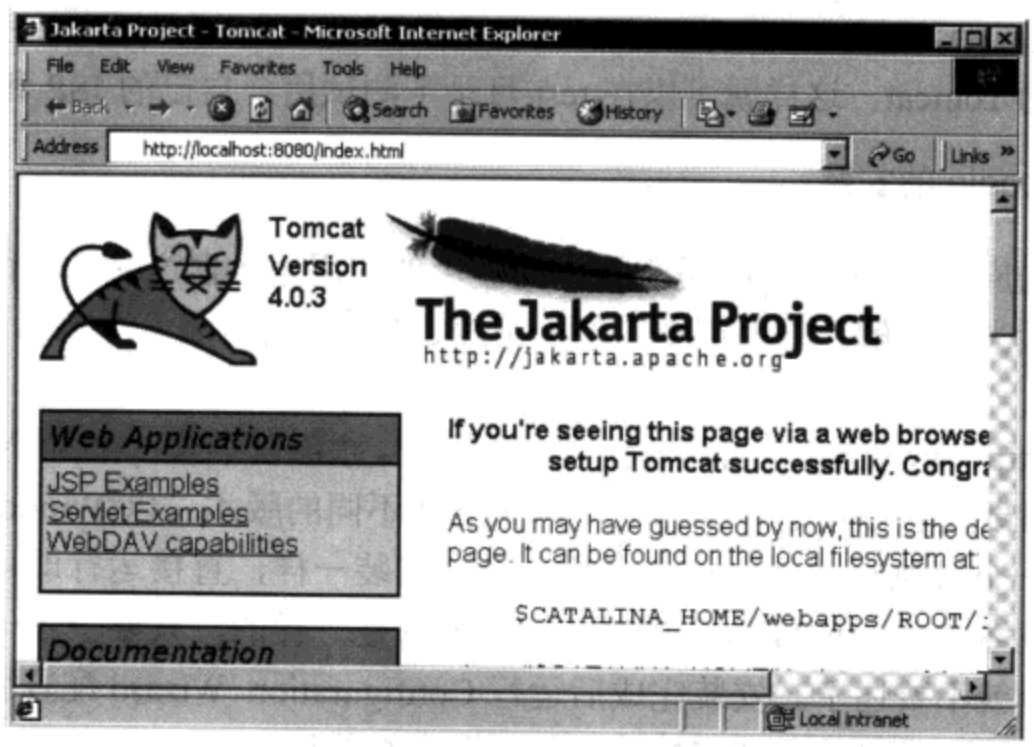


图 3-9　Tomcat 的默认页面

2. Linux 平台

（1）安装 Tomcat。首先要到 http://jakarta.apache.org 站点下载 Tomcat，即下载 jakarta-tomcat-4.0.6.tar.gz 文件，然后使用以下命令将该文件解压缩到 /usr/local/jakarta-tomcat-4.0.6 目录。

```
tar -xzvf jakarta-tomcat-4.0.6.tar.gz
```

（2）设置环境变量。

注意：在 Windows 环境下，字符的大小写并不会影响到程序执行的结果，但是在 Linux 中不行。

输入命令 vi /etc/profile 并在该文件中添加以下代码行：

```
CATALINA_HOME /usr/local/jakarta-tomcat-4.0.6
CLASSPATH .: /usr/local/jakarta-tomcat-4.0.6/lib/servlet.jar
export CATALINA_HOME CLASSPATH
```

（3）注销该用户，并重新登录，以让配置生效。

（4）使用以下代码行来运行 Tomcat：

```
/usr/local/jakarta-tomcat-4.0.6/bin/startup.sh
```

打开 IE 浏览器，并在地址栏中输入 http://localhost:8080。如果浏览器中出现和图 3-9 一样的页面，则说明安装成功。

注意：如果安装失败，请仔细检查环境变量的设置是否正确，尤其在 Linux 下注意严格区

分字符的大小写。

3. 建立自己的目录

Tomcat 4.x 的默认文档根目录为$CATALINA_HOME\webapps\ROOT，测试程序可以放在这个目录里，也可以建立自己的工作目录。

对于 Windows 来说，可以在 C:\java\jakarta-tomcat-4.0.6\webapps 目录下建立目录 test。对于 Linux 来说，可以在 /usr/local/jakarta-tomcat-4.0.6/webapps 目录下建立目录 test。

用文本编辑器打开$CATALINA_HOME/conf/server.xml 文件。在<!-- Tomcat Examples Context -->前边加上以下代码行：

```
<Context path="/test" docBase="test" debug="0" reloadable="true"/>
```

然后重新启动 Tomcat，这样就可以在 test 目录下测试用户自己的 JSP 和 Servlet 文件了。

3.2.3 WebLogic 的安装

作为商业级的应用，WebLogic 是现在广泛使用的应用服务器产品，因而读者对它的安装和配置也应该有适当的了解。下面简要介绍一下 WebLogic 的安装。

1. 下载

首先需要到 http://www.bea.com 站点注册一个用户，才可以下载一个 30 天期限的 WebLogic 试用版。在下载时要注意根据运行平台的不同来下载不同的版本。以 Windows 平台为例，下载后的安装包是一个可执行文件，与大多数软件的安装一样，直接运行即可。接下来的安装步骤很简单，而且各种平台的版本安装运行基本上都是一样的，只要一步一步根据安装向导执行就可以，这里就不赘述了。安装完成后运行 Configuration Wizard 配置服务器，同样是根据向导一步一步进行，这里的安装和配置向导都有详细的说明。

2. 启动服务器

可以用两种方式来启动 WebLogic 服务器。

如果在安装过程中如图 3-10 所示的界面中选择了 Yes，就可以在 Windows 的开始菜单中去启动 WebLogic。

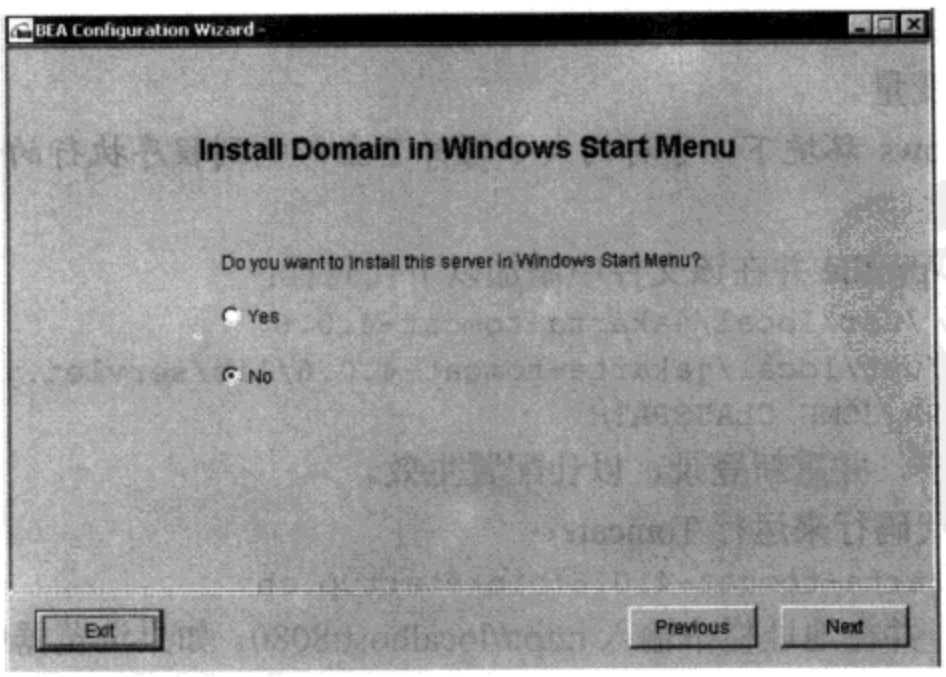


图 3-10 WebLogic 安装界面

另外，还可以直接进入服务器的配置路径下，该默认路径是 \bea\user_projects\mydomain，

然后运行命令 startExampleServer.cmd。因为 WebLogic 程序相对较大，可能需要较长的时间才能完成启动过程。

为了验证 WebLogic 是否已经启动，在 IE 浏览器的地址栏中输入 http://localhost:7001/。如果看到如图 3-11 所示的页面，则说明 WebLogic 启动成功。

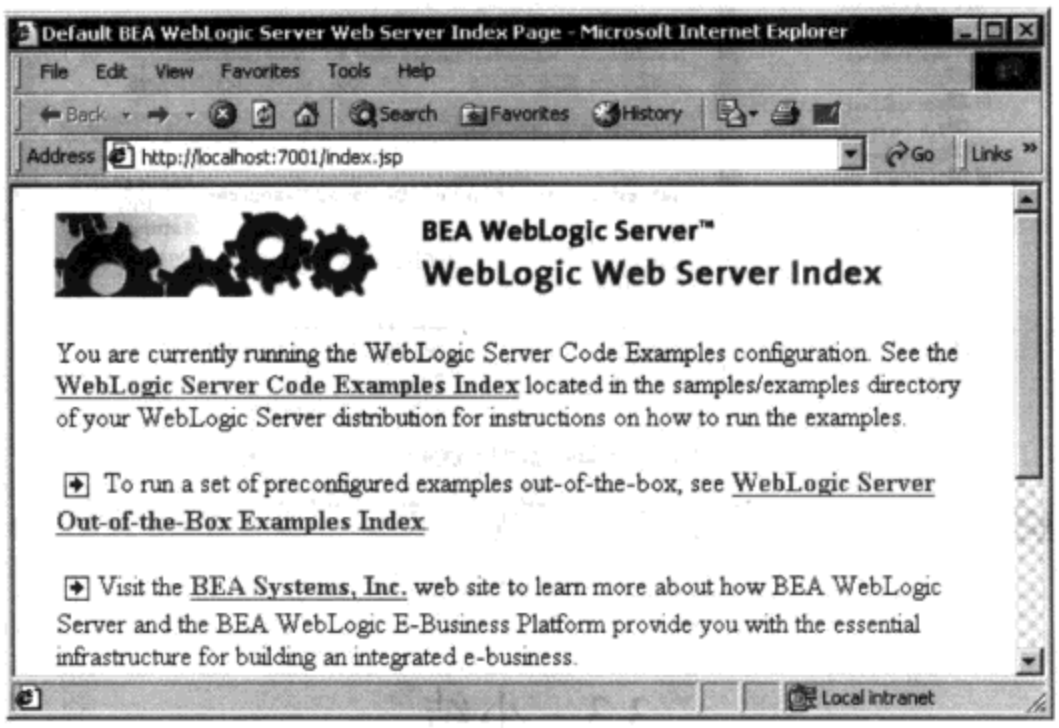


图 3-11　WebLogic 的默认页面

3. 管理和配置 WebLogic

在启动 WebLogic 服务器之后，可以通过 Web 方式对它进行配置和管理。首先登录到 WebLogic 的控制台页面，在浏览器地址栏中输入 http://localhost:7001/console 后，显示如图 3-12 所示的页面。这里要求输入用户名和密码。

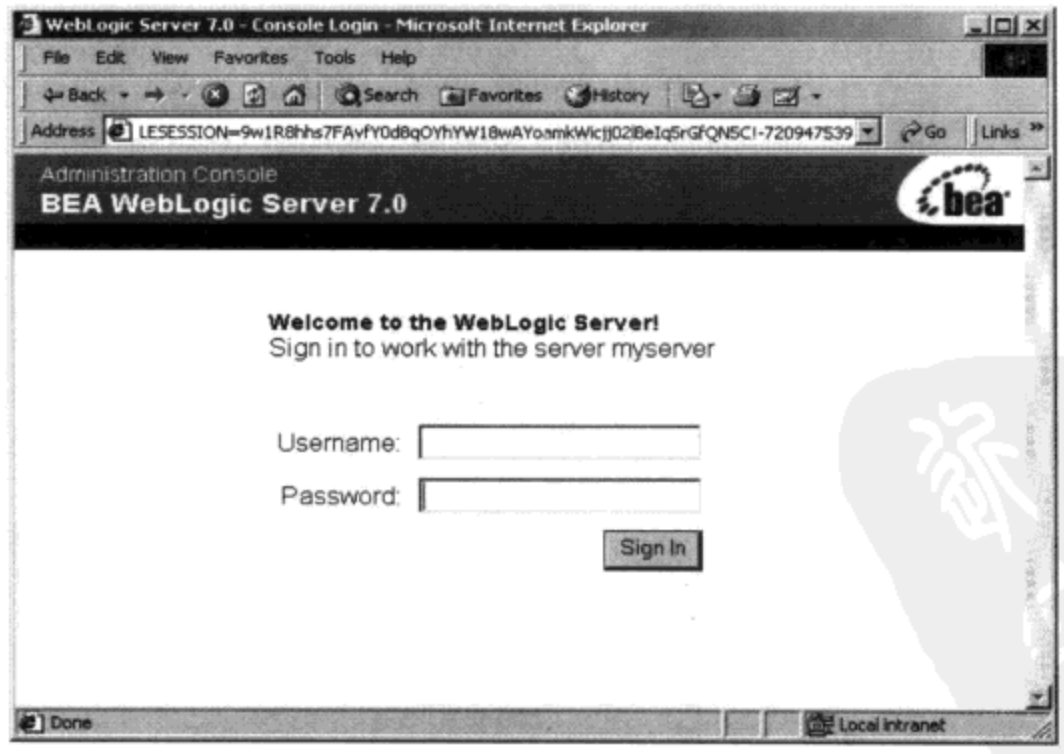


图 3-12　WebLogic 控制台登录页面

在输入用户名和密码后可以修改配置你的服务，该管理页面如图 3-13 所示。

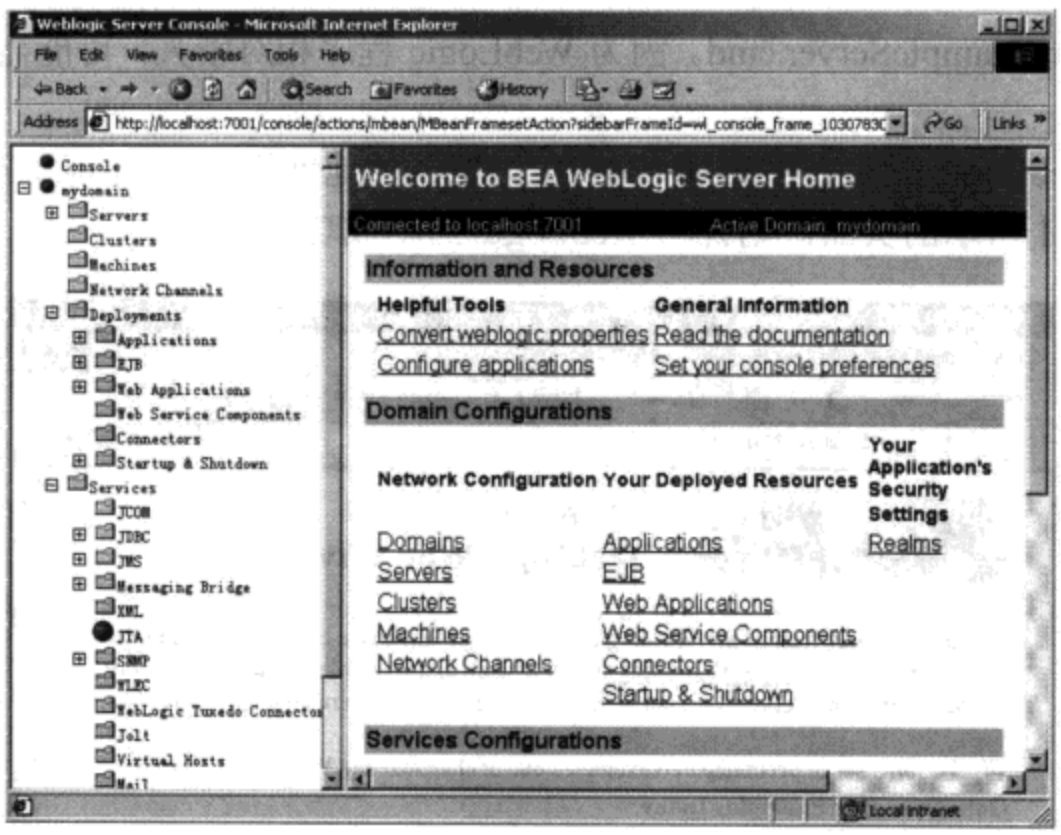


图 3-13　WebLogic 控制台页面

3.3　小结

“工欲善其事，必先利其器”，本章主要介绍了 Java 编程环境的构建，其中包括开发工具如 EditPlus、Dreamweaver 等，运行 Java 的解释器 JDK、JSP，Servlet 的解释器 Tomcat 和 WebLogic。有了这些工具，就可以开始编写、运行和测试 Java 程序了。

第 4 章　JSP 编程范例

本章将了解 JSP 基本的体系结构。通过 JSP 的实例来学习 JSP 文件的元素以及用于表示它们的标记符。学习 JSP 对 Cookie 和 Session 的控制，了解 Cookie 和 Session 的相同点和不同点。学习使用 JSP 来建立一个到数据库的连接。了解用 JavaBean 组件对逻辑进行封装并把它们同 JSP 结合起来。

JSP（Java Server Pages）是由 Sun 公司在 Servlet 的基础上开发出来的一种动态网页制作技术，使用它可以将网页中的动态部分和静态 HTML 相分离。开发人员可以使用平常得心应手的工具并按照平常的方式来书写 HTML 语句。JSP 技术的设计目的是使得构造基于 Web 的应用程序更加容易和快捷，而这些应用程序又能够与各种 Web 服务器、应用服务器、浏览器和开发工具共同工作。JSP 规范是 Web 服务器、应用服务器、交易系统，以及开发工具供应商间广泛合作的结果。在传统的网页 HTML 文件或 XML 文件中加入 Java 程序片段（Scriptlet）和 JSP 标记（tag），就构成了 JSP 网页（*.jsp）。Web 服务器在第一次遇到访问 JSP 网页的请求时，Web 服务器的 JSP 引擎自动把 JSP 文件编译成 Servlet，然后执行 Servlet，同时将执行结果以 HTML 或 XML 格式返回给客户。如果以后再有对该 JSP 文件的请求，JSP 引擎检查该 JSP 文件的最后更新时间。如果发现 JSP 文件的最后更新时间比该 JSP 文件对应的 Servlet 新，则重新编译该 JSP 文件，否则直接执行 Servlet 而不去访问 JSP 文件。正因为这个原因，在调试 JSP 时会觉得它的效率很低。也正是因为有了这个牺牲，JSP 在正式执行时才会有更高的效率。

4.1　简单的 JSP 范例：显示一句话

4.1.1　实例说明

这里用一个简单的例子来说明 JSP 的基本结构和语法。这个例子跟很多其他编程语言入门的例子一样，使用 JSP 简单地输出一句“hello world”。通过这个例子，可以了解到 JSP 的基本语法。

用文本编辑器（如前边介绍的 EditPlus）新建一个文本文件，名称为 helloworld.jsp 并在该文件中添加下面 4 行代码：

```
<%@ page contentType="text/HTML;charset=GB2312"%>
<%  //简单显示一句话的示例
    out.print("Hello world!");
%>
```

我们以这个例子来说明 JSP 文件的基本语法。JSP 文件以“.jsp”为扩展名，并将它放置到$CATALINA_HOME\webapps\test 目录下。

注意：$CATALINA_HOME 表示你的 Tomcat 的安装目录。如果你是按照第 3 章的方法安

装的，则$CATALINA_HOME 就等于 C:\java\tomcat-4.0.6，也就是说你应该把 helloworld.jsp 文件放到 C:\java\tomcat-4.0.6\webapps\test 目录下。以后我们还会用这种方法来表示路径，请注意按照自己的实际情况去寻找相应的路径。

4.1.2 代码分析

这里，我们以这个简单的 JSP 文件为例说明 JSP 的语法基本元素。

1. JSP 页面伪指令（JSP Page Directive）

JSP 页面伪指令为页面提供全局信息，如导入语句、错误处理页面或该页面是否为会话的一部分等，其基本格式为：

```
<%@ page att="val" %> 或<jsp:directive.page att="val"\>
```

在本例中 page 有两个属性，contentType="text/HTML"定义内容类型为 HTML 文本类型，而 charset=GB2312 定义本页使用 GB2312 为该页内码。虽然本例中的字符编码并没有实际作用，但如果页面中有中文时，这个标记是不可省略的。

除了 contentType 和 charset 属性以外，page 指令还可以包含以下属性：

- language="java"：声明该脚本语言的种类，暂时只能用"java" 。
- extends="package.class"：标明 JSP 编译时需要加入的 Java 类的全名，但是要慎重地使用它，因为它会限制 JSP 的编译能力。
- import="{package.class | package.* }, ..."：需要导入的 Java 包的列表，这些包就作用于程序段、表达式以及声明。
- session="{true|false}"：表明会话数据是否可供页面使用，默认为 true。
- buffer="{none|8kb|sizekb}"：确定输出流是否可以缓冲，默认设置为 8kb，并且与下边的 autoFlush 一起使用。
- autoFlush="{true|false}"：如果设置为 true，当缓冲区满时，刷新输出缓冲，而不是引发一个异常事件。默认设置为 true。
- isThreadSafe="{true|false}"：默认设置为 true，它会通知 JSP 引擎能够立刻处理多个用户的请求。同步状态是开发者的工作，以确保该页面是线程安全的。如果把 isThreadSafe 设为 false，那么只能使用控制客户访问网页的单线程模式。
- info="text"：可以通过页面上的 Servlet.getServletInfo()方法进行访问的网页上的信息。
- isErrorPage="{true|false}"：标记该页面是否为错误处理页面，和下边的 errorPage 一起使用。
- errorPage="path to error page"：给出处理异常事件的 JSP 页面的路径。如果设置此项，应保证该路径的有效性，并把该 JSP 页面的 isErrorPage 设为 true。

2. JSP 表达式（JSP Expression）

JSP 表达式的基本格式为：

```
<% 表达式 %>或者<jsp:expression>表达式</jsp:expression>
```

JSP 表达式用于计算并输出，它等价于<%out.print("表达式"); %>。其中的表达式元素表示的是一个在脚本语言中被定义的表达式，它在运行后自动转化为字符串，然后插入到这个表达式在 JSP 文件的位置显示。因为这个表达式的值已经被转化为字符串，所以可以在一行文本中插入这个表达式（形式和 ASP 完全一样）。

3. JSP 代码段（JSP Scriptlet）

JSP 代码段的基本格式为：

```
<% 代码 %>或<jsp:scriptlet>代码</jsp:scriptlet>
```

在 JSP 代码段中插入用于服务的代码。在这里代码和平常的 Java 代码完全一样，每条语句需要以分号结束。使用代码段可以像在普通 Java 程序中一样声明将要用到的变量或者调用方法。本例中就是通过一段代码段来完成“Hello world”的输出。这里的 out 对象是 JSP 的一个内置对象，它是 javax.jsp.JspWriter 的一个实例，并提供了几个方法使用于向浏览器回送输出结果。除了 out 以外，还有下边几个隐含对象可供使用。

- request：表示客户请求，是 HttpServletRequest 的子类。通常一个用户有请求的话，它将会包含有参数列表。
- response：表示 JSP 页面的响应，是 HttpServletResponse 的子类。
- pageContext：表示需要通过一个统一的 API 可以访问的页面属性和隐含对象。
- session：表示与请求相联系的 HTTP 会话对象。
- application：表示通过调用 getServletConfig().getContext()返回 Servlet 的环境。
- config：表示页面的 ServletConfig 对象。
- page：表示页面引用它自身的方法（相当于 Java 代码中的 this）。
- exception：表示传递到错误页面 URL 的没有捕获到的 Throwable 的子类。

4.1.3　运行结果

确保已经启动了 Tomcat，打开浏览器，并在地址栏中输入 http://localhost:8080/test/helloworld.jsp，就会看到页面上显示“Hello world!”，效果如图 4-1 所示。其中 8080 是端口号，浏览器默认访问 80 端口，而 Tomcat 的默认端口是 8080。要想修改这个端口号，在$CATALINA_HOME\conf\server.xml 中会看到有下边一条语句：

```
<Connector className="org.apache.catalina.connector.http.HttpConnector"
           port="8080" minProcessors="5" maxProcessors="75"
           enableLookups="true" redirectPort="8443"
acceptCount="10" debug="0" connectionTimeout="60000"/>
```

图 4-1　helloworld 显示页面

把 port="8080"中的 8080 改为 80。请确保你没有其他的程序占用 80 端口（如 IIS、Apache 等都是默认占用 80 端口），这样重新启动 Tomcat 之后，地址栏中就可以不输入端口号了。

4.2 使用表单实例

4.2.1 实例说明

通过上面的例子，我们了解了 JSP 的基本语法和构成。下面的例子演示如何通过表单向 JSP 传送数据，以及 JSP 如何获取并处理这些表单数据。表单是 HTML 中的一种重要元素，用于收集用户的输入信息，它可以包含输入区、选择框、单选/复选钮等元素，也就是通常说的 DHTML。所谓动态的 HTML 就是用户与服务器之间交互使用信息，其基本方法是用户通过表单提交信息，服务器处理完表单后把处理结果返回给用户。

通过下面的例子，读者将掌握如何把表单提交给 JSP 处理，以及 JSP 如何通过 request 对象的 getParameter 方法获得表单数据。

把数据提交给 JSP 程序的方法有两种：GET 方法和 POST 方法。在 HTML 中表单有一个属性是 method，它的值可以是 get 也可以是 post，这两种方法都可以达到提交数据的目的。它们之间的主要区别是：

- 使用 get 方法提交的数据会在 URL 地址栏中显示出来，而使用 post 方法提交的数据是不会显示出来的。
- post 方法没有数据类型和数据量的限制，而 get 方法只能提交文本类型的数据。其中数据量的限制也就是 URL 长度的限制，一般为 2048 字节。
- 由于 get 方法提交的数据会出现在 URL 地址栏中，而一般的浏览器都有自动记录 URL 地址的功能，所以使用 get 方法提交数据存在不安全性。而与安全相关的数据必须使用 post 方法来提交，当然仅使用 post 方法也是远远不够的。

4.2.2 代码分析

下面的例子由一个 HTML 页面和一个 JSP 页面共同完成。其中表单（form）是包含在 HTML 文件中的，而动作（action）指向 formexample.jsp 文件，由 JSP 来处理表单数据。它的全部代码如下：

```
<html>
<head>
<title>表单示例</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body>
<form name="form1" method="post" action="formexample.jsp">
  <table width="75%" height="136" border="0" align="center">
    <tr>
      <td align="right">您的用户名：</td>
      <td><input name="username" type="text" id="username"></td>
    </tr>
    <tr>
      <td align="right">您的密码：</td>
      <td><input name="password" type="password" id="password"></td>
    </tr>
```

```
    <tr>
      <td align="right"><input type="submit" name="Submit" value="提交"></td>
      <td><input type="reset" name="Submit2" value="重新填写"></td>
    </tr>
  </table>
</form>
</body>
  </html>
```

JSP 从一个请求中获取表单数据，通过 request 对象的 getParameter 方法取出表单（form）中的数据。它的代码如下：

```
<%@ page contentType="text/HTML;charset=GB2312"%>
<%//接收表单数据
String username=request.getParameter("username");
String password=request.getParameter("password");
//把用户名转化为汉字编码
username=new String(username.getBytes("8859_1"),"gb2312");
//把接收到的数据显示到页面上
out.print("您的用户名是：  "+username+"<br>");
out.print("  您的密码是：  "+password);
%>
```

这个 JSP 例子示范了如何接收一个表单请求并获取表单字段的方法。它的主体代码仍然是由代码段来完成的。首先通过 request 对象的 getParameter 方法来获取表单中的请求字段。request 对象也是 JSP 内置对象之一，表示提交给 JSP 页面的一个请求。它有一些方法来获取该请求的属性或者执行关于该请求的操作。getParameter 方法是获取请求中的字段，其中的参数是字段名，并返回该字段的值（字符串）。前一个页面中的用户名和密码字段分别是 username 和 password，所以分别用这两个参数来调用 getParameter 方法，以取得前一个表单中填写的这些值。最后的两条语句将取出的用户名和密码值输出到页面上来。

4.2.3　运行结果

该 HTML 页面包含一个表单和两个输入域，其中一个是用户名，而另一个是密码。该页面在浏览器中的显示如图 4-2 所示。

JSP 在接收完表单数据后，将把接收到的数据显示到该页面上，结果如图 4-3 所示。

图 4-2　提交表单页面

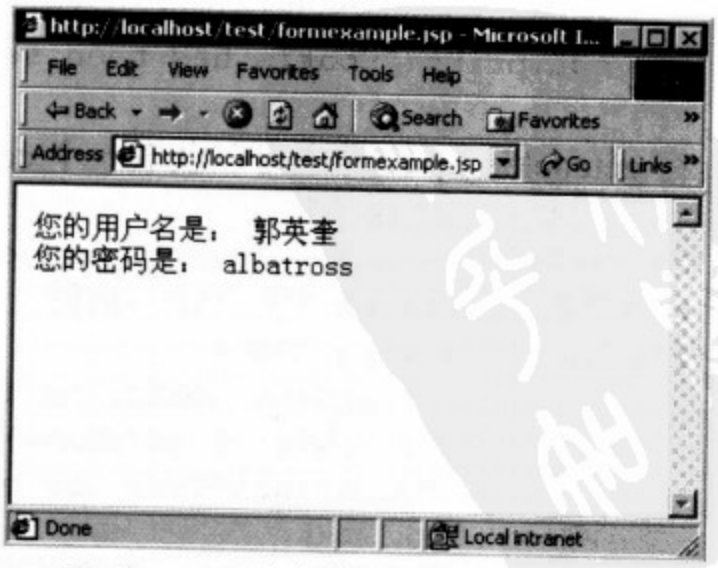


图 4-3　JSP 从表单中获取的表单数据

4.3 JSP 处理 cookie

4.3.1 实例说明

这个例子演示的是在 JSP 中如何使用 cookie 保存客户状态。出于安全考虑，Web 程序不能随意读写用户本地文件，而这又给 Web 编程造成了极大的不便。所以采取的折衷方案是 Web 程序可以通过浏览器在用户的硬盘上写一些限定长度、限定文件个数的文本内容，以用来保存用户信息，这就是 cookie。用户可以在本机上查看以往的 cookie。为了方便对 cookie 有直观的认识，以 IE 浏览器为例，它保存 cookie 的路径如下：

```
Windows 98   C:\windows\Temporary Internet Files\
Windows 2000   C:\Documents And Settings\Administrator\Local Settings
\Temporary Internet Files\
```

上边的路径是按照默认安装的情况下，Windows 2000 中假设是 Administrator 用户，如果不是这样则按照实际情况去查找。在 Windows 98 和 Windows 2000 中都有一个 cookie 的文件夹，在里边可以看到用户的 cookie。但这不是真正的 cookie，只是一个映像，即便你删掉它们，你的 cookie 依然有效。如果 cookie 没有设定过期时间，也就是当浏览器关闭时 cookie 就立刻失效，这样的 cookie 是不会出现在上边提到的文件夹中的。

4.3.2 代码分析

在 setcookie.jsp 文件中创建一个 cookie，如下面的代码所示：

```
<%
//设定一个名称为"username"的 Cookie，值为"albatross"
    Cookie mycookie=new Cookie("username","albatross");
//设置 cookie 的生命期
//保存时间以秒计算，下例是保存一年
    mycookie.setMaxAge(365*24*60*60);
//设置 cookie 的访问路径
//可访问的目录是/test/，就是说只有在/test/下的文件才能访问到这个 cookie
    mycookie.setPath("/test");
//将 cookie 添加到 response 中，注意在此之前不能有任何输出
    response.addCookie(mycookie);
    out.print("Cookie has been setted.");
%>
```

在 getcookie.jsp 文件中可以获取一个 cookie，如下面的代码所示：

```
<%  //取得 cookie 资料
Cookie[] thecookie=request.getCookies();
//取得名叫"username"的 cookie 的值
if(thecookie.length>0){
    for(int i=0;i<thecookie.length;i++){
        if(thecookie[i].getName().equals("username"))
            out.print("Your username is: "+thecookie[i].getValue());
    }
}else{//如果没有任何 cookie，则输出下面的内容
    out.print("Sorry there isn't a cookie named 'username'");
```

```
}
%>
```

在 setcookie.jsp 文件中，首先创建一个名称为"username"、值为"albatross"的 cookie 对象，并将这个 cookie 的最大生命期设为 1 年。最后将该 cookie 放入 response 对象中进行输出，客户端的浏览器取得该 cookie 并保存。

在 getcookie.jsp 文件中，从一个请求中取出所包含的全部 cookie，并找出名字为 username 的 cookie，获取它的值并显示。

注意读取 cookie 时必须用数组。由于这里只有一个 cookie，可以直接用 thecookie[0]读取，但实际应用中还是应该按照例子中的方法来进行读取。

cookie 还有一个重要的属性是 Domain，由于很少使用，上边的例子并没有提到。设置 Domain 的方法是 mycookie.setDomain("your.domain");，例如在一个以 mydomain.com 为域名的站点中有几个子域，如 www1.mydomain.com 和 www2.mydomain.com。这时按照默认的情况，在 www1.mydomain.com 中设置的 cookie 不会被 www2.mydomain.com 中的程序访问到。如果在设置 cookie 时加上 mycookie.setDomain(“mydomain.com”);，这个 cookie 就会被 mydomain.com 中所有的子域访问到了。如果要测试这个例子的话，还会涉及到虚拟主机和域名解析的问题，有兴趣的读者可以参照有关书籍，这里就不介绍了。

4.3.3　运行结果

setcookie.jsp 程序的运行结果如图 4-4 所示。

getcookie.jsp 程序的运行结果如图 4-5 所示。

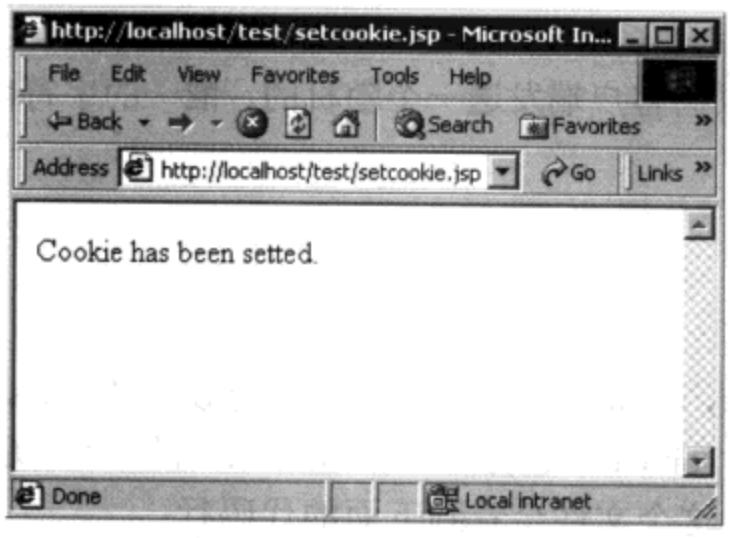


图 4-4　设置一个 cookie

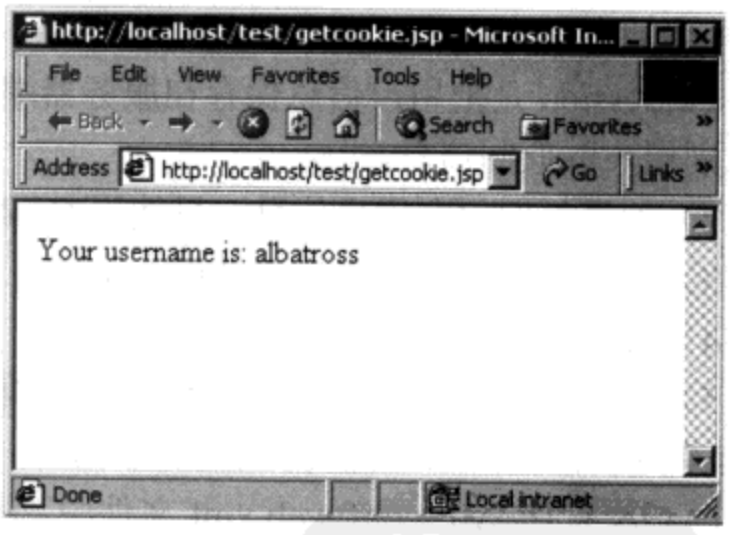


图 4-5　获取一个 cookie

4.4　Session 的管理范例：购物车

4.4.1　实例说明

下面这个例子主要介绍如何在 JSP 中使用 Session 来保存客户的状态。首先解释 Session 以及会话的概念。会话是用户在一段时间内与服务器之间的一系列交互的活动。会话就如同两个人说话，说话的双方都知道在与谁说。而对于普通的网页来讲，如用户到网站上看新闻，

每个人看到的都是一样的。用户知道是对谁讲，而服务器并不知道对谁讲，不管是谁来访问服务器都是那些相同的内容。但是有些情况下服务器端必须知道用户是谁，例如网上购物。用户在网上购物而服务器端竟然不知道是谁买的，这是不可想象的。

Web 编程不同于普通的应用程序。例如我们打开一个 Word 文件，对文件进行编辑，在这个过程中我们写过的内容是不会消失的，因为打开文件的时候，操作系统就给这个请求分配了一个进程。只要我们不关闭 Word 应用程序，这个进程就不会结束。而这个进程不结束，我们就可以很方便地保存想要保存的东西。再如我们平时玩的网络游戏，客户端和服务器端通过 Socket 相互连接。只要用户不退出游戏，这个连接就是保持的，服务器端的进程也是保持的。这样用户的信息就能够被保存。而 Web 编程就不同了，Web 编程使用的是 HTTP 协议，HTTP 协议是一个无状态连接协议，就是说当用户给 Web 服务器发送请求，Web 服务器接受该请求后做出响应，并分配一个进程。当进程运行完毕后把结果返回给浏览器，这个进程就结束了，连接也相应地结束。当用户再有新的请求时，重新建立一个连接，服务器重新为用户的请求生成一个新的进程。也就是说当用户向服务器发送一个请求时，服务器无法判定这个用户的前一个请求是什么。那么服务器端怎样保存用户的信息呢？传统的做法是通过 Cookie 来实现的，Cookie 是保存在客户端的文本片断，它记录着需要的属性和值。因为 Cookie 是保存在客户端的，因而存在安全性方面的问题，同时客户也会为了避免过多的本地信息被 Web 服务器获得而关闭浏览器的 Cookie 功能。Session 和 Cookie 相比的相同和不同之处如下。

（1）相同点：

1）都是保存用户的信息。

2）都是服务器端与浏览器交互信息。

（2）不同点：

1）Session 是把用户信息保存在服务器端，只向客户端发送一个随机的、惟一的字符串用来识别用户并取回信息。

2）如果长期保存用户信息，只能用 Cookie。

3）如果保存大量信息，只能用 Session。

4）Session 比 Cookie 安全。

5）Session 和 Cookie 都有一个有效期的概念，Cookie 的有效期是生成时设定的，Session 的有效期是由 Web 服务器设定的。如 Tomcat 服务器的 Session 的设定是在 $CATALINA_HOME\conf\web.xml 文件中，打开这个文件会看到下面的代码行：

```
<session-config>
    <session-timeout>30</session-timeout>
</session-config>
```

Tomcat 服务器默认的 Session 过期时间为 30 分钟，这个时间是从用户的最后一次请求开始计算的，所以如果用户在有效期内没有向服务器提交任何请求，用户 Session 内的内容就会过期。

在 JSP 中可以使用 Session 对象来实现会话的保持。Session 对象代表的就是当前的会话，可以将与会话有关的属性值保存在 Session 对象中，或者从 Session 中取出某些属性值。需要记录状态的一个典型例子就是购物车，用户在网上购物时，通常会选择若干件不同的商品，在选购过程中需要将已经选好的商品暂时存放在购物车中，继续去选择其他商品。购物车就

需要记录当前客户已经选好的商品的信息并保持到当前会话结束，例如客户选择完成并提交了订单。下面的例子就简单说明如何用 Session 实现购物车的功能。

4.4.2　编程思路

该程序要求记录用户购物车的信息。一种办法是记录用户的 ID，并且把该用户加入购物车的商品写入数据库，然后根据用户的 ID 就可以查出用户购物车中的商品。另一种办法就是把用户购物车中的商品存放在一个 Session 对象中，这种办法比较容易实现，而且更少地占用服务器资源。

为简单起见，我们把所有的商品放到一个数组里，把商品在数组里的标号作为商品的 ID，程序只需要把这个 ID 记录进 Session 就可以查出用户所选中的商品了。

保存在购物车中的商品可以生成多个 Session，也就是说用户每添加一个商品，程序就生成一个 Session 对象。这样 Session 的管理容易混乱，所以还是在一个 Session 里保存多个商品信息比较好。所以就需要一个 Vector 类，首先把商品放入这个 Vector 中，再把这个 Vector 放进 Session 就可以了。

4.4.3　代码分析

```
<%@ page language="java" import="java.util.*" %>
<%@ page contentType="text/HTML;charset=GB2312"%>
<%
//该数组存放所有的商品
String[] products={"土豆","黄瓜","菠菜","西红柿","茄子","海带"};
//显示所有可购买商品的字符串
StringBuffer showProductList=new StringBuffer();
//显示所有购物车中的商品
StringBuffer inCartProducts=new StringBuffer();
int i;
//Vector 是 Java 中非常重要的一个类，这里我们用来存放购物车中的商品编码，并把
//它存放到 Session 中
Vector inCart=new Vector();
if(session.getValue("cart")!=null)  //判断购物车中是否已经有商品
       inCart=(Vector)session.getValue("cart");          //从 session 中取出用户
                                                         //购物车中商品的信息

//如果用户单击了"加入购物车"按钮
if(request.getParameter("add")!=null){
    try{
        //把用户请求中的商品编码转换成 int 型
        i=Integer.parseInt(request.getParameter("item"));
        inCart.addElement(new Integer(i));  //把商品编码存入 Vector
    }catch(NumberFormatException e){
//虽然我们能保证不会出现 NumberFormatException，但这一语句也必不可少
    }
}

//如果用户单击了"从购物车中删除"按钮
```

```
if(request.getParameter("remove")!=null){
    try{
        //把用户请求中的商品编码转换成 int 型
        i=Integer.parseInt(request.getParameter("item"));
inCart.removeElement(new Integer(i));//把商品编码从 Vector 中删除
    }catch(NumberFormatException e){}
}

session.putValue("cart",inCart);                    //把 Vector 存入 Session，以保证购
                                                    //物车信息不会丢失

Integer[] s = new Integer[inCart.size()];   //构建一个 Integer 数组
inCart.copyInto(s); //把 Vector 中的信息转到数组中，以便于显示到页面上
for(i=0;i<s.length;i++){    //把购物车信息存放进 inCartProducts 变量中，以显示到页面上
    inCartProducts.append((i+1)+": "+products[s[i].intValue()]+"<br>\n");
}
//把所有商品信息存放进 showProductList 变量中，以显示到页面上
for(i=0;i<products.length;i++){
    showProductList.append("<option value="+i+">"+products[i]+"\n");
}
%>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=gb2312">
   <title>购物车示例</title>
</head>
<body>
<form method=POST action=cart.jsp>
<BR>
请选择物品加入购物车或从购物车中删除：
<br><br>
<SELECT NAME="item">
    <%=showProductList%>
</SELECT>
<INPUT TYPE=submit name="add" value="加入购物车">
<INPUT TYPE=submit name="remove" value="从购物车中删除">
</form>
<hr>
你的购物车中有下列商品：<br>
<%=inCartProducts%>
</body>
</html>
```

4.4.4 运行结果

在用户第一次访问时，购物车中没有任何商品，只列出商品列表供用户选择。当用户选择了商品并加入购物车后，在下边显示购物车中的商品。当用户选择从购物车中删除商品时，如果用户的购物车中有该项商品，则删除这个商品。在这个程序中，删除商品的顺序是先添加的先删除。当然也可以自己定义一个 removeElement 方法来改变这个顺序。该

页面如图 4-6 所示。

图 4-6　购物车页面

4.5　连接数据库并分页显示记录范例

4.5.1　实例说明

在实际的编程过程中经常遇到要显示数据库中的数据，而数据库中的数据较多，全部显示的话会显得很乱而且效率低，这时我们就会考虑把数据分页显示给用户。下边就是一个简单的分页显示的例子。

通过这个例子可以掌握在 JSP 中使用 JDBC 访问数据库的方法，以及对数据库记录进行分页显示的方法。在这个例子中主要是对 ResultSet 对象的操作。

4.5.2　准备工作

1. 安装 Mysql

Mysql 数据库是一个免费的、开放源代码的数据库，使用方便且性能也很好，尤其是在 Linux 系统下。前边我们介绍过三种流行的 Web 开发语言：JSP、ASP 和 PHP。虽然它们都支持当前所有的主流数据库，但在实际使用过程中默认最佳的结合是 ASP+SQL Server、PHP + Mysql 以及 Java + Oracle。虽然并没有谁规定必须这么用，但厂商之间的默契使得这种组合是非常完美的。我们会在以后的章节介绍使用 Oracle 数据库，这里数据库的选择并不是重点，但从简便易用考虑 Mysql 还是最好的。

Mysql 数据库只适合中小型应用，在实际应用过程中很少有用户在 Windows 系统下使用 Mysql 数据库。如果存放大量的数据，Mysql 是很容易崩溃的，但对于测试来讲已经足够我们使用了。

到 http://www.mysql.com 站点下载 mysql-3.23.52-win.zip，解压缩该文件后进行安装。安装过程很简单，只要连续单击 Next 按钮就可以了。

2. 安装 JDBC 驱动

可以到 http://www.mysql.com 站点下载 JDBC 驱动程序 mysql-connector-java-2.0.14-bin.jar，

可以把这个文件放到$CATALINA_HOME\webapps\test\WEB-INF\lib 目录下，还可以修改$CATALINA_HOME\bin\setclasspath.bat 文件，并把 mysql-connector-java-2.0.14-bin.jar 的路径加入 CLASSPATH 中，如图 4-7 所示。

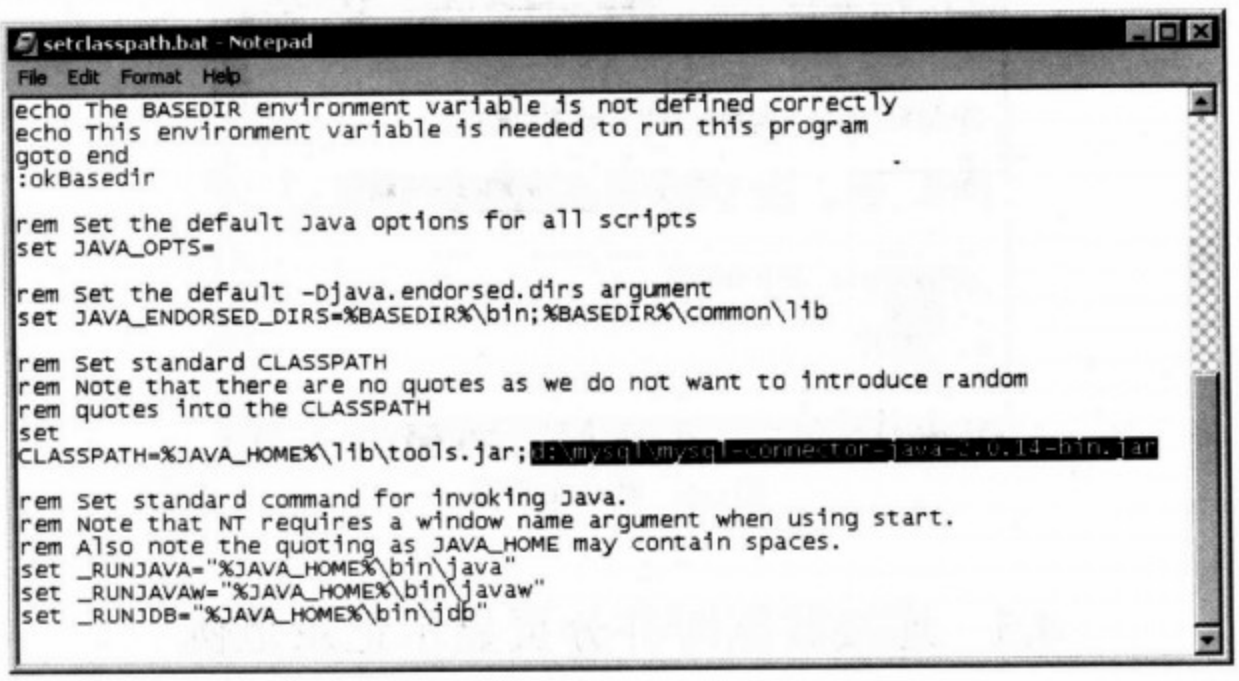


图 4-7　在 Tomcat 中设置 CLASSPATH

3. 设置用户密码

Mysql 数据库安装好后，如果还没有启动 Mysql，只需在命令行状态下运行 mysql 就可以启动 Mysql 数据库了。使用 Mysql 不需要用户名和密码，只要在命令行方式下输入 mysql 就可以登录了。

登录 Mysql 以后就可以输入 SQL 语句来执行数据库操作了，最基本的 SQL 语句有下列几个：

```
Show databases;     //显示数据库里所有的库
Use 库名;          //使用一个库
Show tables;        //显示这个库里所有的表
Desc 表名;          //显示表的结构
Select * from 表名; //显示表里所有的数据
```

注意每个命令后边都要有分号，这样 Mysql 才会执行这条语句。Mysql 添加用户的方法是在库 mysql 的表 user 中添加，还可以用下边的 SQL 语句：

```
Grant 权限 on 库.表 to 用户名@地点 identified by '密码';
```

这里我们用 grant all on * to root@localhost identified by ‘password’;命令，以把所有的库中的表的所有操作权限交付给在本地登录密码是 password 的 root 用户，如图 4-8 所示。如果数据库和 Web 服务器不在同一台机器上，添加用户时就需要把 localhost 改为 Web 服务器的 IP 地址或域名。Mysql 的使用方面的细节这里就不详细介绍了，如果要了解这方面的信息，请参考其他书籍或电子文档。

4. 建表并添加数据

这里使用一种比较简单的方法，先建立一个文本文件 user.sql，然后直接导入数据库。user.sql 文件的内容如下所示：

```
#使用 test 数据库
use test;

#建立 userinfo 表
```

```
    create table userinfo(
        id int primary key auto_increment,
        username varchar(20) not null,
        password varchar(20) not null,
        phone    varchar(20),
        email    varchar(40)
    );
    #加入数据
    INSERT  INTO  userinfo  VALUES  ('',' 林 冲 ','linchong','13664248347',
'linchong@liangshan.com');
    INSERT  INTO  userinfo  VALUES  ('',' 鲁 志 深 ','luda','13664244523',
'luda@liangshan.com');
    INSERT  INTO  userinfo  VALUES  ('',' 宋 江 ','songjiang','13638974623',
'songjiang@liangshan.com');
    INSERT INTO userinfo VALUES ('','扈三娘', 'husanniang' , '13698706578' ,
'husanniang@liangshan.com');
    INSERT INTO userinfo VALUES ('' , '李逵' , 'likui' , '13664893846' ,
'likui@liangshan.com');
    INSERT INTO userinfo VALUES ('' , '史进' , 'shijin' , '13676539946' ,
'shijin@liangshan.com');
    INSERT INTO userinfo VALUES ('' , '阮小二' , 'ruanxiaoer' , '13665438764' ,
'ruanxiaoer@liangshan.com');
    INSERT INTO userinfo VALUES ('' , '卢俊义' , 'lujunyi' , '13664248343' ,
'lujunyi@liangshan.com');
    INSERT INTO userinfo VALUES ('' , '武松' , 'wusong' , '13678323849' ,
'wusong@liangshan.com');
    INSERT INTO userinfo VALUES ('' , '阮小五' , 'ruanxiaowu' , '13643638765' ,
'ruanxiaowu@liangshan.com');
    INSERT INTO userinfo VALUES ('' , '燕青' , 'yanqing' , '13643635745',
'yanqing@liangshan.com');
    INSERT INTO userinfo VALUES ('' , '孙二娘' , 'sunerniang' , '13643645634',
'sunerniang@liangshan.com');
```

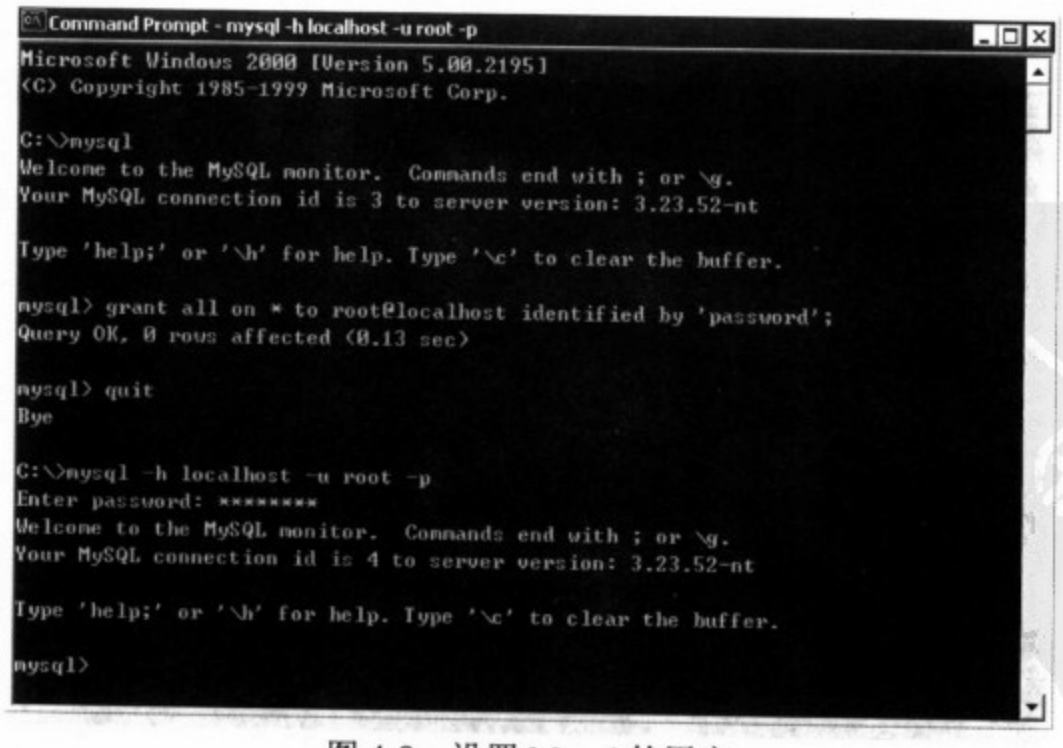


图 4-8　设置 Mysql 的用户

导入的方法为 mysql <user.sql –uroot –ppassword，如图 4-9 所示。

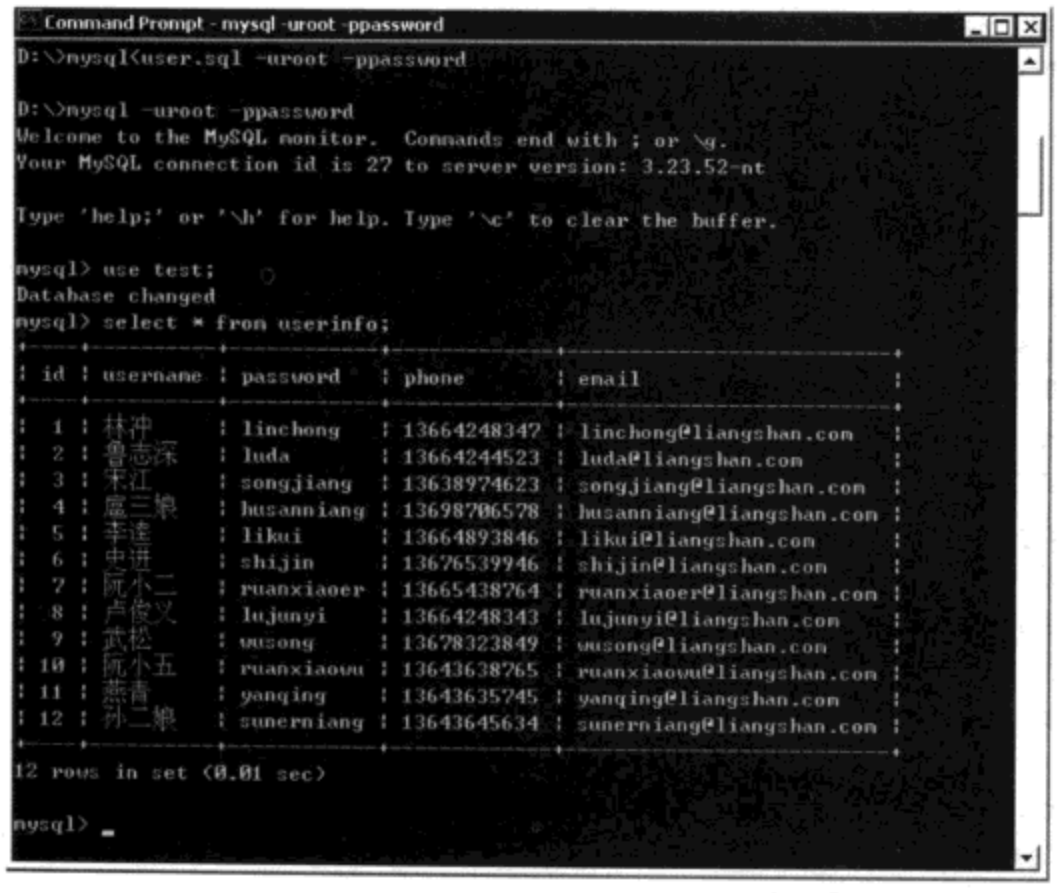


图 4-9　在 Mysql 数据库中建表并导入数据

4.5.3　编程思路

分页显示主要有两种方法。一种方法是在查询数据库时给查询语句一个条件，只查出指定个数的数据。这样做的效率会比较高，可是现在还没有一个标准的 SQL 语句能完成这个操作。流行的数据库各有各的方法，并且有些数据库还不支持这种查询。

另一种分页显示的方法是 Mysql 数据库可以用“limit 起始条目,显示条目数”命令来实现。如使用上边的表 userinfo，在命令行输入 select * from userinfo limit 4,5;命令，结果如图 4-10 所示。

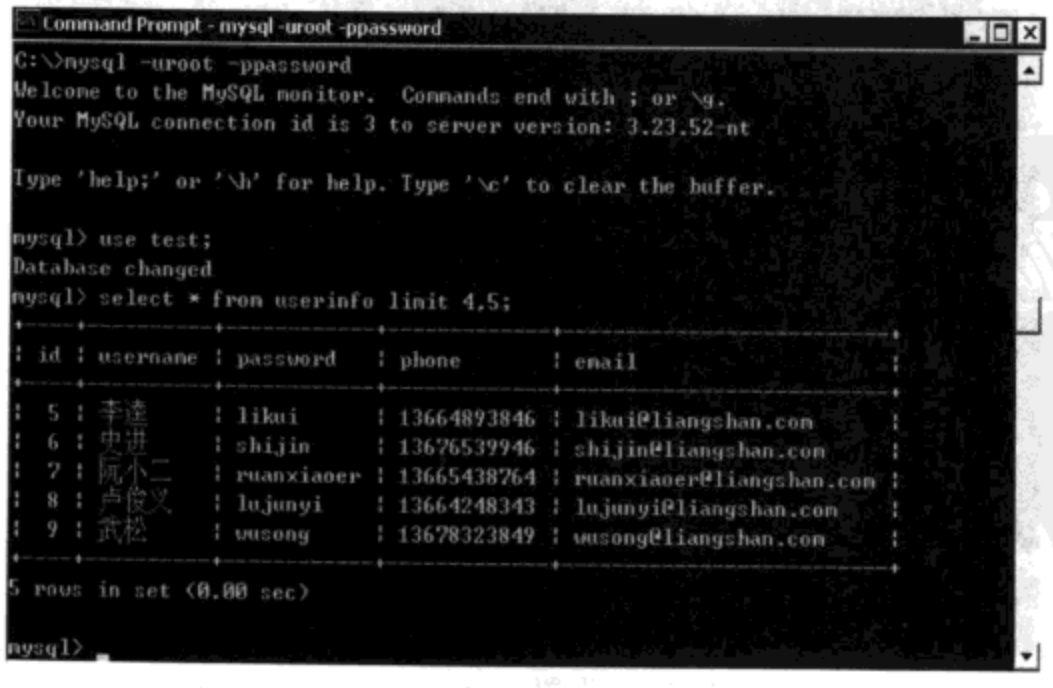


图 4-10　Mysql 数据库中限制结果集的方法

使用上边这种方法，只要在程序中控制起始条目和显示条目数就可以实现分页显示的效果，这样做的缺点是如果要得到总页数还要再进行一次查询才能得到，并且如果程序要移植到其他的数据库就要进行大的修改。

在下边的例子中我们采用另外一种方法，把所有的记录查询出来，并且通过控制 ResultSet 对象来实现分页显示。

4.5.4　代码分析

下面的例子演示在 JSP 中通过 JDBC 访问数据库的方法，以及如何将数据库记录分页显示在页面上。JDBC 是 Java 数据库连接（Java Database Connectivity）的缩写，它由一组用 Java 编程语言编写的类和接口组成。JDBC 为工具/数据库开发人员提供了一个标准的 API，使他们能够用纯 Java API 来编写数据库应用程序。有了 JDBC，向各种关系数据库发送 SQL 语句就是一件很容易的事。有了 JDBC API，就不必为访问 Sybase 数据库专门编写一个程序，为访问 Oracle 数据库又专门编写一个程序，为访问 Informix 数据库又编写另一个程序了。只需用 JDBC API 编写一个程序就够了，它可向相应数据库发送 SQL 语句。同时，使用 Java 编程语言编写的应用程序与平台无关，就无须去忧虑要为不同的平台编写不同的应用程序了。将 Java 和 JDBC 结合起来，将使程序员只需编写一遍程序就可让它在任何平台上运行。

为了简便明了起见，我们把数据库连接放到一个 JSP 文件里。而在正式使用时一般都是把数据库连接封装起来，而只提供一些方法来返回 ResultSet 或自己定义的 Collection。这样一旦数据库发生了修改，只要改动一个文件就可以了。

```
<%@ page language="java" import="java.sql.*,java.io.*,java.util.*" %>
<%@ page contentType="text/HTML;charset=GB2312"%>
<%
//把经常需要修改的数据放在前边，以方便修改
String username="root";   //数据库用户名
String password="password"; //数据库用户密码
int pagesize=5;           //每页显示的数据数量

//JSP 程序中半数以上的 Exception 是 NonePointerException，在声明变量时赋予一个初始值
//能缩短调试时间
ResultSet rs = null;     //数据库查询结果集
Connection conn=null;
Statement stmt=null;
//下边两个变量是把动态生成的部分集中在一起放入页面
//这样做的好处是当修改页面时就可以直接用 Dreamweaver 来编辑了
StringBuffer userInfo=new StringBuffer();
StringBuffer pageInfo=new StringBuffer();
try {//注册驱动程序
    Class.forName("com.mysql.jdbc.Driver");
}catch(java.lang.ClassNotFoundException e) {
    System.err.println("Driver Error " + e.getMessage());
}
//连接数据库并创建 Statement 对象
String sConnStr = "jdbc:mysql://localhost/test?user="+username+"&password=
"+password+"&useUnicode=true&characterEncoding=8859_1";
```

```
try {
    conn = DriverManager.getConnection(sConnStr);
    conn.setAutoCommit(true);
    //建立 Statement 对象并设置指针可前后移动
    stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE ,
     ResultSet.CONCUR_READ_ONLY);
}catch(Exception e) {
    System.err.println("数据库连接错误: " + e.getMessage());
}
//通过 Statement 执行 SQL 语句来获得查询结果
try{//从表 userInfo 中取出数据
    rs=stmt.executeQuery("select * from userInfo");
}catch(SQLException ex){
    System.err.println("数据库查询错误: "+ex);
}
//从用户的请求中获取当前页码
//获得需要显示的页，如果用户请求没有页码参数，则默认是第 1 页
String paramPage=request.getParameter("page");
int currentPage=1;
try{
    currentPage=Integer.parseInt(paramPage);
}catch(Exception e){
    currentPage=1;
}
//将要显示的当前页的数据记录放入 userinfo 中
rs.last();          //把指针置底
int totaluser=rs.getRow();          //获得结果数量
//计算出总页数
int pagecount=(int)Math.ceil((float)totaluser/(float)pagesize);
int i=1;
rs.absolute((currentPage-1) * pagesize + 1);      //把指针放到要显示的第一个数据
//把结果放进变量 userInfo
while(i<=pagesize && !rs.isAfterLast()){
    userInfo.append("<tr bgcolor=#ffffff>\n");
    userInfo.append("<td align=center>"+((currentPage-1)*pagesize+i)+"</td>");
    userInfo.append("<td align=center>"+rs.getString("username")+"</td>");
    userInfo.append("<td>"+rs.getString("phone")+"</td>");
    userInfo.append("<td>"+rs.getString("email")+"</td>");
    userInfo.append("</tr>");
    if(!rs.next())          //当到达最后一个记录时退出循环
        break;
    i++;
}
//关闭数据库连接
try{
    rs.close();          //关闭结果集对象
    stmt.close();        //关闭 Statement 对象
    conn.close();        //关闭数据库连接对象
}catch(SQLException e){
    System.err.println(e);
```

```
    }
    //将要显示页码信息放入 pageInfo
    //把显示翻页的字段放进变量 pageInfo
    pageInfo.append("第"+currentPage+"页 共"+pagecount+"页,共"+totaluser+"个用户 ");
    if(currentPage>1)          //在当前页大于 1 时有向前翻页的连接，否则没有
      pageInfo.append(" <a href='userlist.jsp?page="+(currentPage-1)+"'>&lt;&lt;
上一页</a>");
    else
        pageInfo.append("&lt;&lt; 上一页");
    if(currentPage<pagecount)      //在当前页小于总页数时有向前翻页的连接，否则没有
        pageInfo.append(" <a href='userlist.jsp?page="+(currentPage+1)+"'>下一页
&gt;&gt;</a>");
    else
        pageInfo.append("下一页 &gt;&gt;");
    //显示结果
    %>
    <html>
    <head>
    </head>
    <body>
    <table width="80%" border="0" align="center" cellpadding=5 cellspacing=1
bgcolor=#000000>
      <tr align="center" bgcolor=#cccccc>
        <td width="10%">编号</td>
        <td width="20%">真实姓名</td>
        <td width="35%">电话号码</td>
        <td width="35%">邮件地址</td>
      </tr>
      <%=userInfo%>
    </table>
    <table width="80%" border="0" align="center">
        <tr>
            <td align=right><%=pageInfo%>
            </td>
    </tr>
    </table>
    </body>
    </html>
```

这个例子主要演示了在 JSP 中使用 JDBC 来访问数据库的方法。JDBC 通过统一的 API 来进行数据库操作，对于实际的应用程序来说，只做很少的改动就可以移植到其他数据库平台上。对不同数据库的访问是通过各个数据库自身的 JDBC 驱动程序来提供的。在这里用到的是 Mysql 的 JDBC 驱动程序 mysql-connector-java-2.0.14-bin.jar。

上例中下面的语句打开一个数据库连接并创建 Statement 对象，将要通过该 Statement 对象来执行 SQL 语句。

```
    String sConnStr="jdbc:mysql://localhost/test?user="+username+"&password= " +
password + " &useUnicode=true&characterEncoding=8859_1";

    try {
```

```
        conn = DriverManager.getConnection(sConnStr);
        conn.setAutoCommit(true);
        //建立 Statement 对象并设置指针可前后移动
        stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE ,
    ResultSet.CONCUR_READ_ONLY);
    }catch(SQLException e) {
    System.err.println("数据库连接错误: " + e.getMessage());
    }
```

在有了 Statement 对象后，可以通过它来执行 SQL 语句。JDBC 支持标准的 SQL 语法，有数据库开发经验的用户可以很容易地掌握。

```
    try{//从表 userInfo 中取出数据
        rs=stmt.executeQuery("select * from userInfo");
    }catch(SQLException ex){
        System.err.println("数据库查询错误: "+ex);
    }
```

取得的结果集在 rs 对象中，将当前页面需要显示的记录以 HTML 的格式拷贝到 userinfo 对象中，以便显示使用。

4.5.5 运行结果

该程序运行结果的页面如图 4-11 所示。

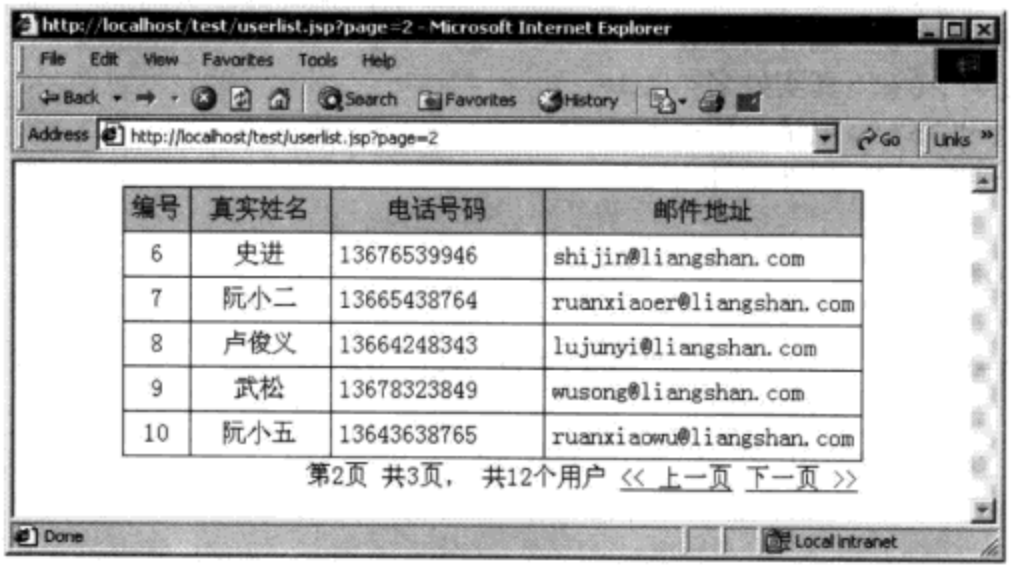


图 4-11 分页显示

4.6 JSP 实现文件上传范例

4.6.1 实例说明

这个例子主要示范的是在 JSP 中使用 JavaBeans 的方法。JavaBeans 就是用 Java 语言遵守一定的规范而编写的组件的模型。JavaBeans 最初是把重点放在开发工具中的可视化操作组件上，当然它同样适用于非可视化编程。这本书的后边还要详细介绍企业级的 JavaBeans（Enterprise JavaBeans），也就是 EJB。EJB 是 Sun 公司提出的一种侧重于服务器端的可以合理部署 Java 组件的服务器框架模型。

这里先介绍 JavaBeans 在 JSP 编程中的应用。JavaBeans 是一种满足下列准则的 Java 类：

- 公有类。
- 不带参数的公有构造函数。
- 用于模拟属性的公有 set 和 get 方法。

这里通过一个文件上传的实例来说明在 JSP 中使用 JavaBeans 的方法。

4.6.2　编程思路

前边介绍 form 表单时，接收表单提交来的信息用 request.getParameter("***")的方法。这样之所以行得通是因为 Servlet 已经内置了对通用表单的解析方法，而对于文件上传必须用 enctype="multipart/form-data" 的表单。Servlet 和 JSP 都没有帮我们解析这种表单，所以就必须自己解析。要解析这个表单，就必须知道浏览器传来的内容是什么，这里就需要编写一个程序接收浏览器送来的信息，只有知道了这个信息才可能解析出表单中的值。

先编写一个 Java 文件建立一个 Socket 以监听一个端口，这样让浏览器把信息提交到这个端口，然后把接收到的信息打印出来。

首先建立一个 HTML 文件：

```
<html>
<body>
<center><b>文件上传示例</b></center>
<br>
<table>
  <form     method="post"     enctype="multipart/form-data"     action="http:
//localhost:8888">
        <tr>
            <td> <input type=file size=20 name="fname"> </td>
            <td> <input type=Submit value="上传"> </td>
        </tr>
      </form>
    </table>
    </body>
    </html>
```

注意这个文件中的 form 的 action。

Server.java 的代码如下，由于不在本书的范围之内，这里就不详细介绍了，只需知道它的运行结果就可以了。

```
import java.net.*;
import java.io.*;
public class Server{
    public static void main(String[] arg){
        int port=8888;      //监听的端口号，注意和 HTML 代码中的 action 相对应
        try {
            ServerSocket s=new ServerSocket(port);
            System.out.println("Waiting on port "+port);
            while(true){
                Socket socket=s.accept();
                InputStream is=socket.getInputStream();
                InputStreamReader isr= new InputStreamReader(is);
```

```
                BufferedReader br=new BufferedReader(isr);
                while(true){    //按行读取，按行输出
                    String str=br.readLine();
                    System.out.println(str);
                }
            }
        }catch(Exception e){
            System.err.println(e);
        }
    }
}
```

编译 Server.java 文件并运行，然后打开上边的 HTML 文件并选中一个文件，单击“提交”按钮，结果如图 4-12 所示。

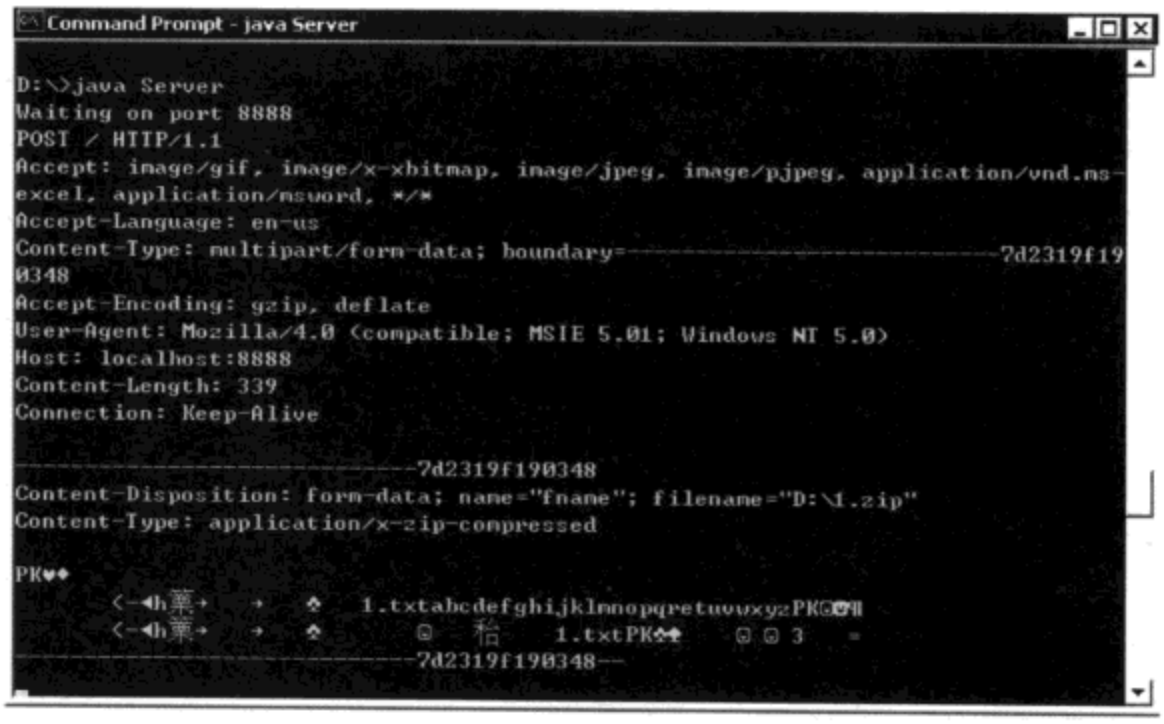


图 4-12　获得浏览器提交的信息

其中 boundary=-----------------------------7d2319f190348 是说明限界符的值，下边的两个限界符之间的东西才是我们需要的，上边的是浏览器的信息，这里就不介绍了。Content-Disposition: form-data;对我们没有用处，只是说明这是通过表单传来的数据。name="fname";是表单中文件输入框的名称。filename="D:\1.zip" 是文件在用户磁盘的路径，程序通过它来解析文件名。Content-Type: application/x-zip-compressed 是文件类型。然后跟着是一个空行，后边直到下一个 boundary 就是用户上传来的文件。这个示例的作用主要就是把这段内容写进文件。

通过上边的操作，我们已经知道了浏览器提交信息的格式，接下来把文件解析出来也就水到渠成了。

4.6.3　代码分析

这个例子的页面代码很简单。一个 HTML 页面用来显示表单、接受用户输入需要上传的文件名，它的代码如下：

```
<html>
<body>
<center><b>文件上传示例</b></center>
```

```
<br>
<table>
  <form method="post" enctype="multipart/form-data" action="uploadfile.jsp">
    <tr>
        <td> <input type=file size=20 name="fname"> </td>
        <td> <input type=Submit value="上传"> </td>
    </tr>
  </form>
</table>
</body>
</html>
```

注意 form 中的一个属性是 enctype="multipart/form-data" ，要实现文件的上传这个属性是必需的，还有 form 的 method 只能是 post，而不能是 get。

该表单的提交对象是下面的 JSP 页面，即 uploadfile.jsp。JSP 的代码非常简单，因为程序的主要工作都交给 JavaBeans 去完成了，使用 JavaBeans 的好处在这里已可见一斑了。

```
<%@ page language="java"%>
<%@ page contentType="text/HTML;charset=GB2312"%>
<jsp:useBean id="fub" scope="page" class="mybean.FileUploadBean" />

<%
String Dir = "D:\\upload";  //保存文件的路径，请确保目录存在，或改到其他目录
//通过调用 JavaBeans 的方法完成上传操作
fub.setUploadDirectory(Dir);
fub.uploadFile(request);
out.print("<center><font color=red>成功上载文件至" + Dir + "</font></center>");
%>
```

通过上边的例子可以看到 JavaBeans 的最一般的使用方法。使用 JavaBeans 首先要声明<jsp:useBean>。id 属性是用来识别 JavaBeans 的，也就是这个 bean 的实例。class=""是标明一个类，可以简单地认为这就等于 Java 程序中的 mybean.FileUploadBean fub = new mybean.FileUploadBean();，其中的 scope 属性标明这个 bean 的作用范围，也就是这个 bean 的生命周期。在这里 scope="page"表示这个 bean 只作用于本页，也是 JavaBeans 的默认属性。scope 的值还可能是"request"、"session"和"application"。

程序中的第 2 行是 JSP 页面的编译器指令，它指定页面的编码和内容类型等属性。这里需要强调的是 JSP 中使用 JavaBeans 的方法，即第 3 行。这里的 useBean 关键字表示要创建并使用 JavaBeans 对象，id 属性指定该对象的名称，后面对这个 JavaBean 的引用就通过这个 id。scope 属性定义这个对象的作用域，作用域可以有以下 4 种取值：

- Page：表示作用域仅限当前页面中。
- Request：bean 实例维持当前的客户请求。
- Session：bean 持续的时间和会话一样长。
- Application：bean 实例随应用一起创建，并且持续到应用程序结束。

接下来首先调用 FileUploadBean 的 setUploadDirectory(Dir)方法，其中的 Dir 参数为上传文件存放的目录，这里将它设为服务器本地的一个文件夹。然后调用 FileUploadBean 的 uploadFile 方法，该方法的参数为 request 对象，需要上传的文件的路径名和文件名都在该 request 中，由 JavaBeans 读取并实现上传。具体的文件上传功能是由 FileUploadBean 来实现的，

下面给出该文件的代码。不过这里的重点是在 JSP 中使用 JavaBeans 的方法。

下面是实现上传文件的 JavaBeans 的 FileUploadBean.java 代码：

```
package mybean;

import java.io.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.ServletInputStream;
import javax.servlet.ServletException;

public class FileUploadBean{
    private static String newline = "\r\n";        //设定换行符
    private String uploadDirectory = ".";          //默认的保存位置
    private String ContentType = "";               //文件类型
    private String CharacterEncoding = "";         //编码格式

    //设定文件要保存在服务器中的位置
    public void setUploadDirectory(String s){
        uploadDirectory = s;
    }
    //提取文件名，本方法是把用户上传的文件按照原名保存
    private String getFileName(String s){
        int i = s.lastIndexOf("\\");
        if(i < 0 || i >= s.length() - 1){
            i = s.lastIndexOf("/");
            if(i < 0 || i >= s.length() - 1)
                return s;
        }
        return s.substring(i + 1);
    }

    //得到 content-type
    public void setContentType(String s){
        ContentType = s;
        int j;
        if((j = ContentType.indexOf("boundary=")) != -1){
            ContentType = ContentType.substring(j + 9);
            ContentType = "--" + ContentType;
        }
    }
    //得到文件编码格式
    public void setCharacterEncoding(String s){
        CharacterEncoding = s;
    }

    public void uploadFile( HttpServletRequest req) throws ServletException,
IOException{
        setCharacterEncoding(req.getCharacterEncoding());
        setContentType(req.getContentType());
        uploadFile(req.getInputStream());
    }
```

```
        public void uploadFile( ServletInputStream servletinputstream) throws
ServletException, IOException{

            String s5 = null;
            String filename = null;
            byte Linebyte[] = new byte[4096];
            byte outLinebyte[] = new byte[4096];
            int ai[] = new int[1];
            int ai1[] = new int[1];

            String line;
            //得到文件名
            while((line    =    readLine(Linebyte,    ai,    servletinputstream,
CharacterEncoding)) != null){
                int i = line.indexOf("filename=");
                if(i >= 0){
                    line = line.substring(i + 10);
                    if((i = line.indexOf("\"")) > 0)
                        line = line.substring(0, i);
                    break;
                }
            }

            filename = line;
            if(filename != null && !filename.equals("\"")){
                filename = getFileName(filename);

                String sContentType = readLine(Linebyte, ai, servletinputstream,
CharacterEncoding);
                if(sContentType.indexOf("Content-Type") >= 0)
                    readLine(Linebyte, ai, servletinputstream, CharacterEncoding);

                //建立新文件的文件句柄
                File file = new File(uploadDirectory, filename);

                //建立生成新文件的输出流
                FileOutputStream fileoutputstream = new FileOutputStream(file);

                while((sContentType = readLine(Linebyte, ai, servletinputstream,
CharacterEncoding)) != null){
                    if(sContentType.indexOf(ContentType) == 0 && Linebyte[0] == 45)
                        break;

                    if(s5 != null){
                        //写入新文件
                        fileoutputstream.write(outLinebyte, 0, ai1[0]);
                        fileoutputstream.flush();
                    }
```

```
                    s5    =    readLine(outLinebyte,    ai1,    servletinputstream,
CharacterEncoding);
                    if(s5 == null || s5.indexOf(ContentType) == 0 && outLinebyte[0]
== 45)
                    break;
                    fileoutputstream.write(Linebyte, 0, ai[0]);
                    fileoutputstream.flush();
                }

                byte byte0;
                if(newline.length() == 1)
                byte0 = 2;
                else
                byte0 = 1;
                if(s5 != null && outLinebyte[0] != 45 && ai1[0] > newline.length()
* byte0)
                fileoutputstream.write(outLinebyte, 0, ai1[0] - newline.length()
* byte0);
                if(sContentType  !=  null  &&  Linebyte[0]  !=  45  &&  ai[0]  >
newline.length() * byte0)
                fileoutputstream.write(Linebyte, 0, ai[0] - newline.length() *
byte0);
                fileoutputstream.close();
            }
        }
        //readLine 函数把表单提交上来的数据按行读取
        private String readLine(byte Linebyte[], int ai[],ServletInputStream
servletinputstream,String CharacterEncoding){
            try{ //读取一行
                ai[0] = servletinputstream.readLine(Linebyte, 0, Linebyte.length);
                if(ai[0] == -1)
                    return null;
            }catch(IOException ex){
                return null;
            }
            try{
                if(CharacterEncoding == null){
                    //用默认的编码方式把给定的 byte 数组转换为字符串
                    return new String(Linebyte, 0, ai[0]);
                }else{
                    //用给定的编码方式把给定的 byte 数组转换为字符串
                    return new String(Linebyte, 0, ai[0], CharacterEncoding);
                }
            }catch(Exception ex){
                return null;
            }
        }
    }
```

4.6.4　运行结果

该 HTML 文件的显示效果如图 4-13 所示。

上传成功后的显示页面如图 4-14 所示。可以自己检查一下 D:\upload 目录下是否有该文件。

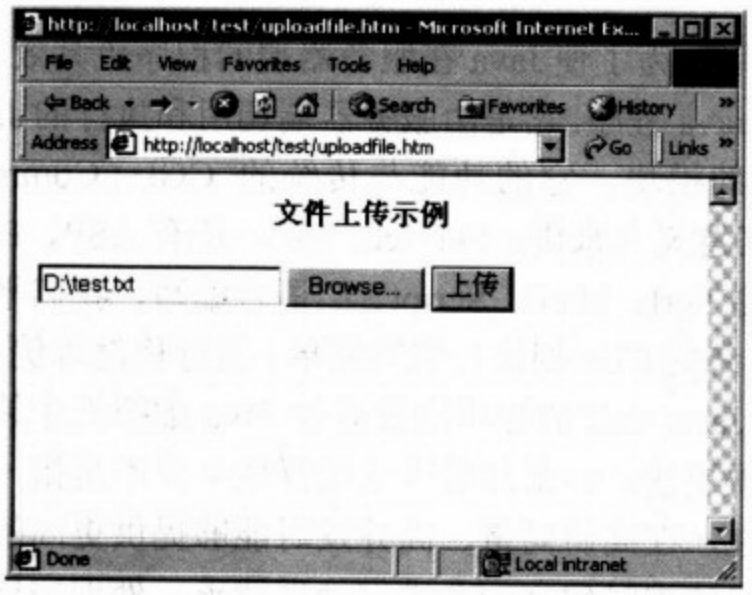


图 4-13　输入上传文件路径及文件名

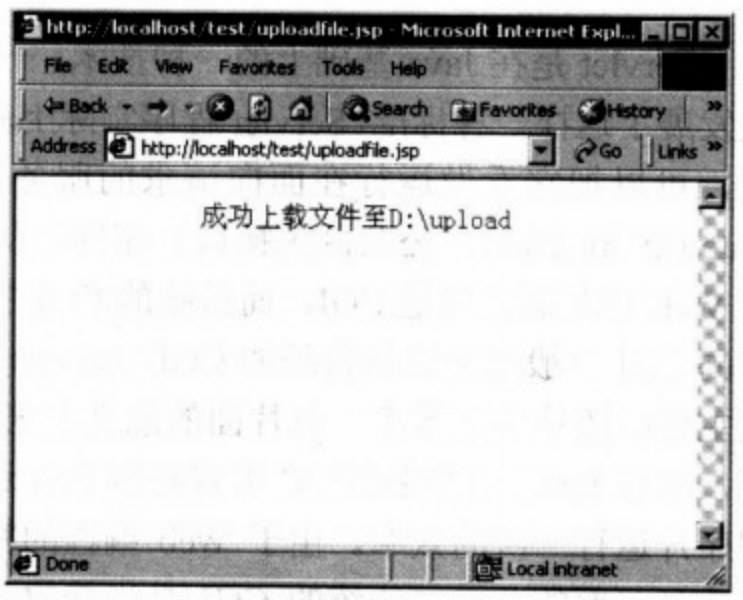


图 4-14　上传成功

4.7　小结

本章主要介绍了 JSP 的基本语法、内部对象的使用以及 JavaBeans 的使用方法和一般规则。通过本章的学习，可以掌握一般的 JSP 代码的编写。

本章的重点：

- 使用 JSP 处理表单数据。
- Cookie 和 Session 的异同及各自的使用方法。
- 基本的连接数据库的操作，以及分页显示的处理。
- 使用 JavaBeans。

第 5 章　Java Servlet 编程范例

Servlet 是在 Java 基础上的一种技术、一种标准。为了使 Java 在服务器端的程序更具效率，就发展了这样一种标准。Servlet 程序实际上就是 Java 程序，它是由服务器端调用和执行的 Java 类，可以把它看做运行在面向请求的服务器上的模块，它的功能与传统的 CGI（Common Gateway Interface，公共网关接口）相同。从严格意义上来讲，Servlet、JSP，还有 ASP、PHP 等 Web 开发语言都是 CGI，而传统的 CGI 主要由 Perl、Shell、Script、C 语言编写，我们平时讲的 CGI 一般是指这些传统的 CGI。Servlet 和传统的 CGI 相比有编写简单、运行稳健等优点，最主要的还是运行效率。从片面的意义上来看，Java 编写的程序需要通过 Java 虚拟机才能运行，所以 Java 程序很难比 C 语言甚至 Perl 运行得更快，但是这里讲的运行效率并不是指一个程序片运行一次的效率。由于 Web 编程的特殊性，在高访问量、高并发时能够提供更高的效率才是问题的关键。传统的 CGI 是当接收到客户请求时建立一个进程处理请求，处理完毕后进程结束。而 Servlet 被客户端发出的第一次请求激活，处理完请求后继续在后台运行以等待新的请求，每个请求将生成一个新的线程，而不是一个完整的进程。Servlet 只有在 Web 服务器关闭时才会卸载，这就保证了 Servlet 在高访问量、高并发时比传统的 CGI 有更高的执行效率。也正是这个原因，如果 Servlet 已经被激活，在修改 Servlet 又重新编译后需要重新启动 Web 服务器，这样对 Servlet 的修改才会生效。传统的 CGI 现在也有如 modperl、fastCGI 等方法使程序驻留在内存，也会使程序有非常高的运行效率，但由于其编写复杂、维护困难等原因被新技术替代已经成为必然。

Servlet 是 JSP 的基础，按照常理应该先讲 Servlet 再讲 JSP，可是 JSP 比 Servlet 更加通俗易懂，先知其然再知其所以然也不失为一个好办法。先有了感性的认识再逐步地探究其原理往往更能事半功倍，所以我们把 Servlet 放到 JSP 的后边。

在这一章我们要学习使用 Servlet 进行服务器端编程。Servlet 是 Java 的 Web 应用程序开发的最基本的技术，JSP 也是首先被编译转换成 Servlet 然后执行的。Servlet 为服务器端的代码和基于 Web 的客户端之间提供了一种简便的通讯形式，因此成为 Java 服务器端编程的核心。为了避免 Servlet 程序员考虑过多的网络连接、获取请求、产生特定格式的响应等细节问题，这些问题都由称作 Servlet Container（容器）或 Servlet Engine（引擎）的部分来完成，而 Servlet 开发人员只需要通过 Servlet API 实现业务层次的逻辑即可。Servlet Engine 和 Servlet API 都包含在 Java Servlet 开发人员工具包（Java Servlet Development Kit，JSDK）中，它可以从 http://java.sun.com/products/servlet 下载。本章将主要介绍使用 Servlet API 开发服务器端应用程序的方法。

5.1　简单的 Servlet 例子

5.1.1　实例说明

这个例子首先演示 Servlet 接收一个表单请求并将该请求中的数据显示出来，它的目的在

于让读者了解 Servlet 的基本结构和方法。同时，通过此例来解释 Servlet 中的一些基本概念，以及熟悉 Servlet 涉及的一些内嵌对象。

Java Servlet是以.java为后缀的源文件，通过javac编译成.class文件。为了运行这些Servlet实例，需要把编译好的 Servlet 的 class 文件放进 WEB-INF/classes 中。注意 Tomcat 默认要在使用 Servlet 时在浏览器地址栏加入/servlet/，例如我们要把下面的例子 servletExample.java 编译成为 servletExample.class，请确保把 servlet.jar 加入了 CLASSPATH 中，否则编译器将报告找不到 javax.servlet 的错误。在 Tomcat-4.x 中，Servlet.jar 在$CATALINA_HOME/common/lib 目录下，Servlet API 就包含在该包里。要使用 Servlet，只需要把编译好的 Servlet 类文件放在 $CATALINA_HOME/webapps/test/WEB-INF/classes/ 下，访问这个 Servlet 就要访问 http://localhost/test/servlet/servlet 类名。

5.1.2　代码分析

servletExample.java 的代码如下所示：

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
//继承 HttpServlet 来实现自己的 Servlet
public class servletExample extends HttpServlet {
//重载 HttpServlet 的 doGet 方法。首先设置内容类型和页面编码为
//text/html;charset=gb2312。然后输出 HTML 文件的头部信息
public void doGet(HttpServletRequest request,
HttpServletResponse response) throws IOException, ServletException{
        response.setContentType("text/html;charset=gb2312");
//使用 response 的 out 对象输出 HTML 文件头
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<body>");
        out.println("<head>");
        out.println("<title>Servlet 示例</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<h3>Servlet 示例</h3>");
//接收客户端请求中传来的数据
        String username = request.getParameter("username");
        String password = request.getParameter("password");
         if (username!=null){//转换为中文字符集
             username=new String(username.getBytes("8859_1"),"gb2312");
         }
         out.println("<br>");
//将取到的数据输出到页面中
        if (username != null || password != null) {
           out.println("你的姓名是：");
           out.println(username + "<br>");
           out.println("你的密码是：");
```

```
            out.println(password);
        }
//再在输出页面中产生一个表单，接受用户再输入姓名和密码信息
        out.print("<form action=\");
        out.print("servletExample\");
        out.println("method=POST>");
        out.println("姓名");
        out.println("<input type=text size=20 name=username>");
        out.println("<br>");
        out.println("密码");
        out.println("<input type=password size=20 name=password>");
        out.println("<br>");
        out.println("<input type=submit value='提交'>");
        out.println("</form>");
        out.println("</body>");
        out.println("</html>");
}
//doPost 方法的处理与 doGet 一致，即对 post 请求的处理与 get 请求一致
public void doPost(HttpServletRequest request,
HttpServletResponse response)throws IOException, ServletException{
        doGet(request, response);
}
}
```

这个 Servlet 继承了 HttpServlet 类。在 Servlet API 中定义了 GenericServlet 基类，它提供了 Servlet 接口的基本实现部分。HttpServlet 扩展了 GenericServlet 类并且支持 HTTP 协议的实现。HttpServlet 将是学习的重点，因为虽然 Servlet 可以支持多种协议，HTTP 仍然是 Servlet 使用最广泛的协议。我们的 Servlet 继承了 HttpServlet 类，它就获得了执行 HTTP 协议的能力。

方法 doGet 是这个 Servlet 的实现体，该方法是 Servlet 实际处理 HTTP 请求的方法。HTTP 协议中的请求类型有 GET、POST、HEAD 等，我们通常需要处理的是 GET 和 POST 类型。当收到 GET 类型的请求时，HttpServlet 中的实现会将请求分派给 doGet 方法。因此，只要将处理 GET 请求的代码放在 doGet 方法中，它就可用来处理 GET 请求。

doGet 方法有两个参数，分别是 HttpServletRequest 和 HttpServletResponse 类型。这两种类型都是 Servlet 规范中定义的接口，分别代表请求对象和应答对象。在实现 doGet 方法时，从 request 对象中获取请求的详细信息，并将处理结果输出到 response 中。结果的输出通过从 response 对象获得的 PrintWriter 对象完成。在 out 对象中写入 HTML 格式的文本，这个 HTML 就是实际的输出结果。也就是说，请求该 Servlet 返回的结果与下面的 HTML 页面的效果是一样的。

```
<html>
<head>
    <title>Servlet 示例</title>
</head>
<body bgcolor="white">
        <h3>Servlet 示例</h3>
<form action=\servletExample\ method=POST>
    姓名 <input type=text size=20 name=username>
    <br>
```

```
    密码 <input type=password size=20 name=password>
    <br>
    <input type=submit value='提交'>
</form>
</body>
</html>
```

5.1.3　运行结果

第一次访问时的页面如图 5-1 所示，在输入框中输入相应的信息后，单击“提交“按钮。该表单提交的结果页面如图 5-2 所示。

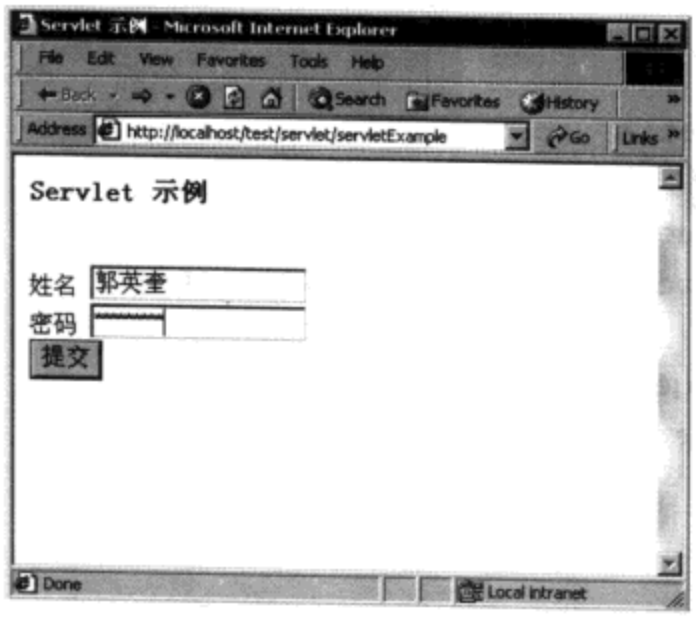


图 5-1　请求页面

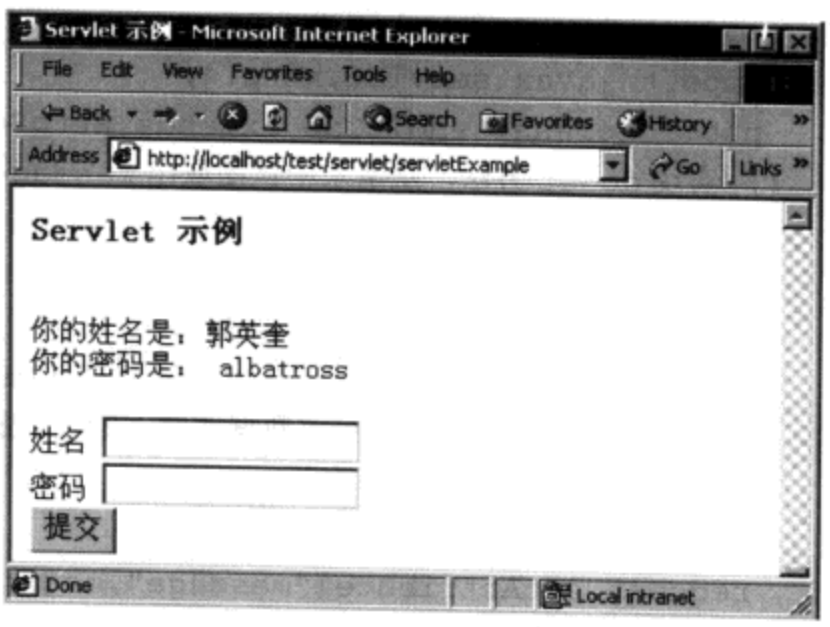


图 5-2　Servlet 输出页面

5.2　Servlet 与 JSP 之间的通信

5.2.1　实例说明

由于 JSP 是先被转换成 Servlet 然后编译的，所以在服务器端直接使用 Servlet 的效率要稍高于使用 JSP 的效率。但 JSP 是将 Servlet 以类似于 HTML 的 tag 方式表现出来，在易读性和输出页面方面要比 Servlet 有更大的方便性。基于它们各自的特点，Servlet 通常用来处理业务逻辑，而 JSP 用来处理页面的表现部分。因此它们之间的通讯是很必需的，下面通过一个例子来演示 Servlet 如何将结果返回给 JSP。

5.2.2　编程思路

程序之间的值的互相传递是编程时经常要用到的，其中最简单的办法是类似浏览器的 GET 方法，即直接使用 URL 传递值。这种方法简便直接，需要注意的是进行 URL 编码后才能保证值会被正确地传递。而且 URL 的长度是有限制的，如果需要传递的数据太大，即 URL 太长，就不能保证值会被正确地传递了。当然还可以通过 Session 来传递值，这种方法在第 4 章已经讲过。在 Servlet 中使用 Session 与在 JSP 中基本相同，这里就不介绍了。但是这种方法浪费系统资源而且效率不高。在 JSP 中还可以使用类似浏览器端的 POST 方法，这是在 ASP

和 PHP 中很难实现的。

```
<jsp:forward page="your url">
    <jsp:param name="ParameterName " value="ParameterValue" >
<jsp:param name="ParameterName 2" value="ParameterValue2" >
</jsp:forward>
```

Servlet 也为我们准备了一种方法，下边通过实例讲述这种方法的实现。

5.2.3 代码分析

sendToJsp.java 的代码如下所示：

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class sendToJsp extends HttpServlet {
//实现 doGet 方法，从请求中试图取出 message 数据
public void doGet(HttpServletRequest request,
HttpServletResponse response)throws IOException, ServletException{
        //接收客户端传来的数据
String msg = request.getParameter("message");
//检查是否有该信息，如果该数据非空，则将其放入 request 对象，然后
//将请求分派给一个 JSP 页面
        if(msg!=null){
request.setAttribute("message",msg);
getServletConfig().getServletContext().getRequestDispatcher
("/receivemsg.jsp").forward(request,response);
        }else{
//如果该数据为空，生成一个包含 form 的 HTML 页面请求，并输入要发送给 JSP 的数据
            response.setContentType("text/html;charset=gb2312");
            PrintWriter out = response.getWriter();
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet 返回值给 Jsp 示例</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h3>Servlet 返回值给 Jsp</h3>");
            out.println("<br>");
            out.print("<form action=sendToJsp method=POST>");
            out.println("你要传递的信息：");
            out.println("<input type=text size=20 name=message>");
            out.println("<input type=submit value='发送'>");
            out.println("</form>");
            out.println("</body>");
            out.println("</html>");
        }
    }
//doPost 方法的处理与 doGet 一致，即对 POST 请求的处理与 GET 请求一致
```

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

这个例子中 Servlet 将请求转发给 JSP 是通过下面的语句实现的：

```
    getServletConfig().getServletContext().getRequestDispatcher("/receivemsg.j
sp").forward(request, response);
```

其中的 receivemsg.jsp 就是要转发请求的 JSP 页面，而 request 和 response 是 Servlet 的 doGet 方法的传入参数。这个 JSP 只是简单地从请求中取出 message 数据并进行显示，它的源代码如下所示：

```
    <%@ page contentType="text/HTML;charset=GB2312"%>
    <%
        String msg=(String)request.getAttribute("message");
        out.print("你的信息是：  "+msg);
    %>
```

5.2.4　运行结果

这个 Servlet 首先尝试从 request 中取出 message 数据。如果它不存在，则说明是第一次请求该 Servlet，这时会显示一个页面，要求用户输送要发送的 message，如图 5-3 所示。

图 5-3　接受输入消息

数据提交后的运行结果如图 5-4 所示。

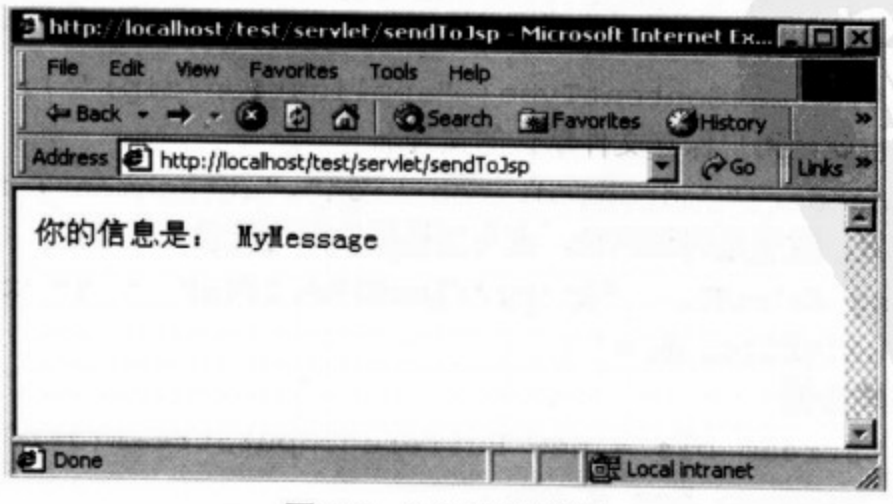


图 5-4　JSP 收到消息

上边的实例演示了如何实现从 Servlet 向 JSP 程序传递值。从 JSP 向 Servelt 传递值的方法同样可以使用 URL，还可以使用上边提到的 jsp:forward 的方法实现。

5.3 控制输出流

5.3.1 实例说明

Servlet 发送给客户端浏览器的内容大部分是 HTML 文件，但在有些情况下还需要 Servelt 显示图片、JavaScript 文件以及样式表文件等。下面这个例子演示如何通过 Servlet 实现下载文件。有时候我们不愿意让用户随意下载我们网站上的东西，需要做一下身份验证，就可以用下边的办法来掩饰文件的真实路径，甚至该文件不在我们的服务器上也可以记录每个文件的下载次数。

5.3.2 编程思路

浏览器接收到服务器传来的信息首先要解析的是 content-type，然后是处理数据。在前边的例子中用到过 response.setContentType("text/html;charset=gb2312")，浏览器接收到这样的 content-type 就会解释显示这个文件。由于这里要用户下载这个文件，就要把 content-type 设置为其他类型，一般通用的压缩文件的 content-type 是"application/zip"。浏览器接到这个 content-type 就认为它是二进制的压缩文件，就会在窗口提示下载。如果你装有 Flashget 或 NetAnts，这些程序就会被调用启动以接收下载的信息。当然这个文件的 content-type 并不一定必须是 Zip 压缩文件，它可以是任何形式的文件。浏览器只解析服务器端告诉它的 content-type，而不去解析这个文件以检查服务器是不是在骗它，它对服务器是持百分之百的信任态度。这也是由 HTTP 协议的特性决定的，浏览器不可能接收完文件并解析出文件的 content-type 后再做出反应，它必须要提前做出反应。

5.3.3 代码分析

download.java 的代码如下所示：

```
public class download extends HttpServlet {
    public void doPost(HttpServletRequest request,
HttpServletResponse response)throws IOException, ServletException{
//设置 Content-type 为 application/zip，浏览器会弹出下载窗口
        response.setContentType( "application/zip;" );
//指定文件名，也就是用户保存文件默认的文件名
response.setHeader("Content-disposition","attachment; filename=test.zip" );
//指定文件的位置，这里是网络路径，也可以根据实际情况设为本地路径
        String fileURL = "http://localhost/test/test.zip";
URL url=new URL(fileURL);
//打开输入流和输出流
BufferedInputStream bis = new BufferedInputStream(url.openStream());
BufferedOutputStream bos = new
BufferedOutputStream( response.getOutputStream());
```

```
//从输入流中读出数据，并写入输出流
        byte[] buff = new byte[2048];
        int bytesRead;
        while (-1!=(bytesRead=(bis.read(buff,0,buff.length)))){
         bos.write(buff,0,bytesRead);
        }
        if (bis!=null){
        bis.close();
        }
//关闭输入流和输出流
        if (bos!=null){
        bos.close();
        }
    }
//doGet 方法与 doPost 方法一致
    public void doGet(HttpServletRequest request,
HttpServletResponse response) throws IOException, ServletException{
        doPost(request,response);
    }
}
```

这个示例的文件源的 URL 是 http://localhost/test/test.zip。如果要测试这个程序，必须保证 CATALINA_HOME/webapps/test/下有 test.zip 这个文件。当然在实际的应用程序中这样做不如直接指定本地文件效率高，在这里只是做一个测试。如果文件不在本地，才应该使用这种方法。

5.3.4　运行结果

该示例的运行结果如图 5-5 所示。

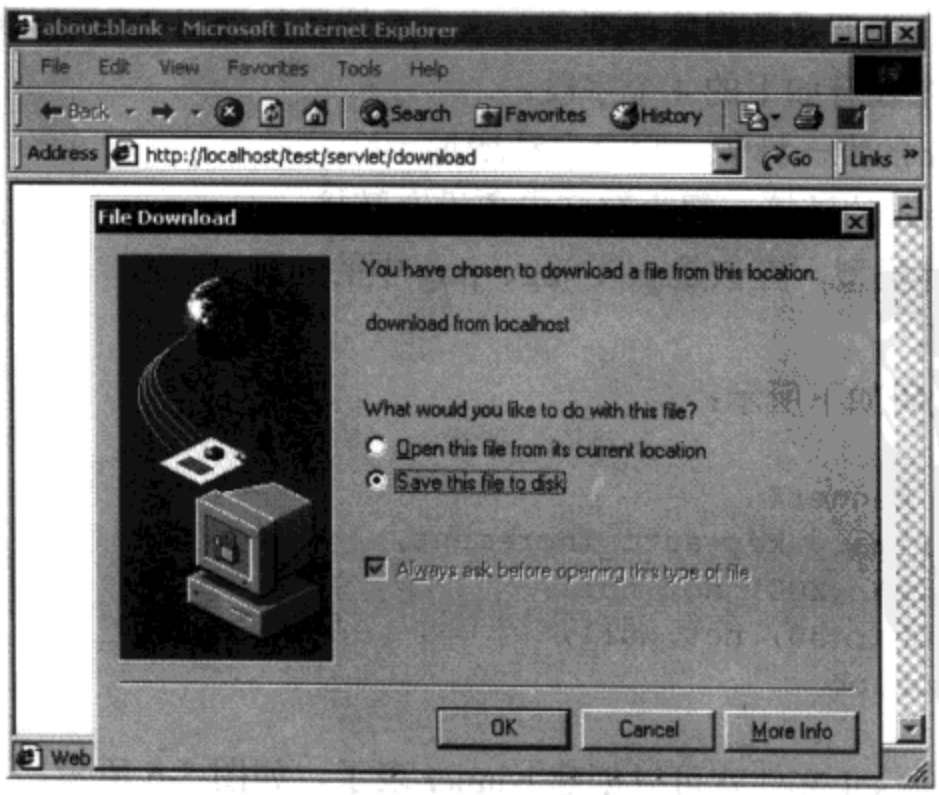


图 5-5　文件下载页面

5.4 用 Servlet 管理广告条显示

5.4.1 实例说明

广告已经成为我们生活中不可或缺的部分，我们几乎无时无刻不在主动或被动地受到广告的影响。网络编程中当然也离不开广告，下边这个例子的作用就是轮流显示广告。因为广告客户已经为广告付钱，所以程序必须使所有的广告都能得到显示，而且机会均等。当然有些情况比如根据付款的多少区分优先级等，这里就不做考虑了。下边这个示例的作用是管理页面的广告显示。

5.4.2 编程思路

管理广告显示首先就是要能方便地添加广告条。广告的最基本的数据是图片和链接指向。保存数据最基本的有两种方法，一种是把图片保存在固定路径下，把图片的路径和链接保存在数据库中。这种方法涉及到两个问题，如果允许上传图片并保存图片，就必须开放保存图片位置的写权限。这样做的缺点首先是程序交给客户使用时，你必须告诉他要开放哪个位置的写权限，如果客户是个电脑盲，真的要花好大的力气才能使他明白。还有更重要的是安全问题，开放写权限对安全有着巨大的损害。另一种方法是把图片和链接都保存在数据库中，这种方法的缺点就是增加了服务器的开销。下边的实例就是采用这种方法。

5.4.3 准备工作

这里的准备工作主要是数据库的准备，这个示例还是使用 Mysql 数据库。首先创建一个 banner 表，并且这个表有如下的 4 个字段。

- id：banner 的惟一标识，它会自动增加。
- link：banner 的链接，很少有广告条没有链接。
- type：图片类型，应该是 gif、jpg、png 中的一种。
- file：图片。

banner.sql 的代码如下所示：

```
use test;
create table banner(
    id int primary key auto_increment,
    link varchar(200) not null,
    type varchar(50) not null,
    file longblob
);
```

直接导入 banner.sql 文件就可以创建 banner 表了，如图 5-6 所示。

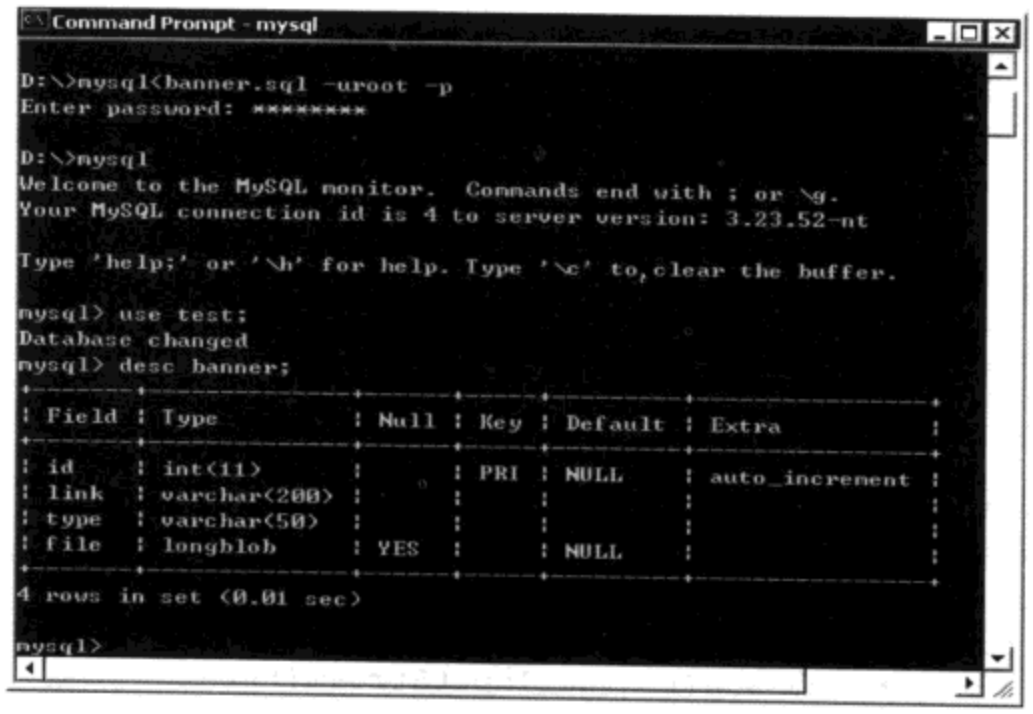


图 5-6　创建一个 banner 表

5.4.4　代码分析

formData.java 的代码如下所示：

```
package mail;
//自定义数据类型
import java.util.*;
public class formData{
    public String name;      //表单项名称
    public String value;     //表单项的值，如果是文件，这个值是文件名
    public String contentType;     //如果是文件，这个值是文件的类型，否则为 null
    public byte[] file;      //把文件保存为字节数组
}
```

前边已经讲过关于文件上传并解析表单数据，但只是解析一个文件。这个例子稍微要复杂一些，表单中可以有任意多的条目（text、button、textarea 和 checkbox 等），也可以有任意多的文件，但原理还是一样的。

这个文件是解析表单后，返回一个 formData 的 Vector，并将文件保存在 byte 数组中。

parseRequest.java 的代码如下所示：

```
import java.util.*;
import java.io.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.ServletInputStream;
import javax.servlet.ServletException;

public class ParseRequest{
    private String ContentType = "";          //表单类型
    private String CharacterEncoding = "";    //编码格式

    //设定表单类型
    public void setContentType(String s){
        ContentType = s;
        int j;
```

```
            if((j = ContentType.indexOf("boundary=")) != -1){
                ContentType = ContentType.substring(j + 9);
                ContentType = "--" + ContentType;
            }
        }
        //设置文件编码格式
        public void setCharacterEncoding(String s){
            CharacterEncoding = s;
        }

        public Vector getFormData( HttpServletRequest req)
    throws ServletException, IOException{
            setCharacterEncoding(req.getCharacterEncoding());
            setContentType(req.getContentType());
            return getFormData(req.getInputStream());
        }
        //解析表单数据并保存到 Vector 中
        public Vector getFormData( ServletInputStream servletinputstream)
    throws ServletException, IOException{
            Vector formdatas=new Vector();
            formData fd=null;
            int size=2048000;//上传的文件的最大限制为 2M
            byte[] buffer = new byte[size];
            int bufferLength=0;
            byte temp[];
            byte Linebyte[] = new byte[4096]; //设定缓冲区大小为 4096
            //由于 Java 中不可以用指针，在函数中改变外部的值用数组方法
            int ai[] = new int[1];
            int ai1[] = new int[1];

            String line;
            String boundary="--";
            String itemname="";
            String filename="";
            //textarea 的值可能有回车，用 StringBuffer 累加
            StringBuffer sb=new StringBuffer("");
            boolean nameflag=false;
            boolean fileflag=false;
            int skipline=0;
            int i;
            /*从 ServletInputStream 读取一行，注意这里不是严格的读取一行，如果在小于缓
            * 冲区大小的时候出现换行符，则读取一行，否则读满缓冲区
            * Linebyte：缓冲区
            * ai：读到的位置，也就是换行符的位置
            * servletinputstream：浏览器请求
            * CharacterEncoding：数据类型
        */
            while((line    =    readLine(Linebyte,    ai,    servletinputstream,
    CharacterEncoding)) != null){
                //得到 boundary
```

```
if((i=line.indexOf("boundary=--"))>=0){
                boundary=line.substring(i+10);
                continue;
            }
            //这里使用 nameflag 和 fileflag 控制读取
            if(!nameflag&&!fileflag){
                //得到表单项的名称
                itemname=getName(line);

                if(itemname!=null&&itemname!=""){
                    fd=new formData();
                    fd.name=itemname;
                    nameflag=true;
                    //得到文件名
                    filename=getFileName(line);

                    if(filename!=null && filename!=""){
                        fd.value=filename;
                        fileflag=true;
                    }else{
                        fileflag=false;
                    }
                    skipline=0;
                    continue;
                }
                //如果表单项的类型不是 file
                if(nameflag&&!fileflag){
                    if(skipline<1){
                    //跳过一行
                        skipline++;
                        continue;
                    }
                    else{
                        //得到表单项的值
                        if(line.indexOf(boundary)>=0){
                            nameflag=false;
                            fd.value=trim(sb.toString());
                            formdatas.addElement(fd);
                            sb=new StringBuffer("");
                        }else{
                                sb.append(line);
                        }
                    }
                    continue;
                }
                //如果表单项的类型是文件
                if(nameflag&&fileflag){
```

```
if(skipline<1){
    //得到文件类型
    if(line.indexOf("Content-Type:")>=0)
        fd.contentType=trim(line.substring(14));
    else{
        //跳过一行
        skipline++;
    }
    continue;
}else{
        //把文件内容保存到 byte 数组
        if(line.indexOf(boundary)>=0){

            if(bufferLength>=2){
                //去掉最后的换行符
                bufferLength-=2;
                temp=new byte[bufferLength];
                for(i=0;i<bufferLength;i++)
                    temp[i]=buffer[i];
                fd.file=temp;
                buffer=new byte[size];
                bufferLength=0;
            }

            formdatas.addElement(fd);
                nameflag=false;
                fileflag=false;
            }else{
                for(i=0;i<ai[0];i++,bufferLength++)
                    buffer[bufferLength]=Linebyte[i];

            }

        }

    }
    return formdatas;
}
//得到表单项的名称
public String getName(String s){
    int i=s.indexOf(" name=\"");
    if(i>0){
        s=s.substring(i+7);
        i=s.indexOf("\"");
        return s.substring(0,i);
    }else
        return "";
}
```

```
    //提取文件名
    private String getFileName(String s){
        try{
            int i = s.indexOf("filename=");
            if(i >= 0){
                s = s.substring(i + 10);
                if((i = s.indexOf("\"")) > 0)
                    s = s.substring(0, i);
                i = s.lastIndexOf("\\");
                if(i < 0 || i >= s.length() - 1){
                    i = s.lastIndexOf("/");
                    if(i < 0 || i >= s.length() - 1)
                        return s;
                }
                return s.substring(i + 1);
            }else
                return "";
        }catch(Exception e){
            return "";
        }
    }

    //使用 readLine 函数，以把表单提交上来的数据按行读取
    private String readLine(byte Linebyte[], int ai[],ServletInputStream
servletinputstream,String CharacterEncoding){
        try{ //读取一行
            ai[0] = servletinputstream.readLine(Linebyte, 0, Linebyte.length);
            if(ai[0] == -1)
                return null;
        }catch(IOException ex){
            return null;
        }
        try{
            if(CharacterEncoding == null){
                //用默认的编码方式把给定的 byte 数组转换为字符串
                return new String(Linebyte, 0, ai[0]);
            }else{
                //用给定的编码方式把给定的 byte 数组转换为字符串
                return new String(Linebyte, 0, ai[0], CharacterEncoding);
            }
        }catch(Exception ex){
            return null;
        }
    }
    //这个 trim 函数用来去掉回车换行符，而不是 String 中的 trim 函数用表去掉空格
    public String trim(String s){
        if(s.charAt(s.length()-1)=='\n'){
            if(s.length()<=2)
                return "";
            else
```

```
                s=s.substring(0,s.length()-2);
            return trim(s);
        }
        if(s.indexOf("\r\n")==0){
            s=s.substring(2);
            return trim(s);
        }
        return s;
    }
}
```

addBanner.java 的代码如下所示：

```
//这个文件的作用是上传图片并保存到数据库中
import java.io.*;
import java.sql.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class addBanner extends HttpServlet {
public void doGet(HttpServletRequest request,HttpServletResponse response)
throws IOException, ServletException{
       response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        try{
            ParseRequest pf=new ParseRequest();
            //解析表单数据
            Vector v= pf.getFormData(request);

            formData fd;
            String link="";
            String contentType="";
            byte[] image=null;

            Enumeration e=v.elements();
            while(e.hasMoreElements()){
                fd=(formData)e.nextElement();
                if(fd.name.equals("link")){
                //得到广告的链接
                    link=fd.value;
                    continue;
                }else if(fd.name.equals("image")){
                //得到图片
                    contentType=fd.contentType;
                    image=fd.file;
                }
            }
            if(!link.equals("")&&!contentType.equals("")&&image!=null){
                //注册驱动程序
Class.forName("com.mysql.jdbc.Driver");
```

```
//连接数据库的字符串
String sConnStr = "jdbc:mysql://localhost/test?user=root&password"
                    +"=password&useUnicode=true&characterEncoding=8859_1";
                //建立一个数据库连接
                Connection conn = DriverManager.getConnection(sConnStr);
                //添加广告信息的 SQL 语句
String sql="insert into banner (link,type,file) values(?,?,?)";
                //建立准备状态
                PreparedStatement pstmt = conn.prepareStatement(sql);
                //加入数据
                pstmt.setString(1,link);
                pstmt.setString(2,contentType);
                pstmt.setBytes(3,image);
                //执行操作
                pstmt.executeQuery();
                //关闭数据库连接
                pstmt.close();
                conn.close();
                out.println("连接和图片已经记入数据库");
            }else{
                out.println("数据不完整，请重新输入。");
            }
        }catch(NullPointerException npe)
        }catch(Exception e){
            System.err.println(e);
            e.printStackTrace();
        }

    out.println("<html>");
    out.println("<body>");
    out.println("<head>");
    out.println("</head>");
    out.println("<body bgcolor=\"white\">");
    out.println("<h3>上传图片</h3>");
     out.print("<form action=\"");
    out.print("addBanner\" ");
    out.println("method=POST  enctype='multipart/form-data'>");
    out.println("连接");
    out.println("<input type=text size=20 name=link>");
    out.println("<br>");
    out.println("图片");
    out.println("<input type=file size=20 name=image>");
    out.println("<br>");
    out.println("<input type=submit value='提交'>");
    out.println("</form>");
    out.println("</body>");
    out.println("</html>");
}
 //本文件无区别地对待 post 和 get 方法递交来的数据
public    void    doPost(HttpServletRequest    request,HttpServletResponse
```

```
response)throws IOException, ServletException{
            doGet(request, response);
        }
    }
```

showBanner.java 的代码如下所示：

```
//这个文件的作用是随机显示 banner
import java.io.*;
import java.sql.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class showBanner extends HttpServlet {
public void doGet(HttpServletRequest request,HttpServletResponse response)
throws IOException, ServletException{
        try{
            Class.forName("com.mysql.jdbc.Driver");
            String sConnStr = "jdbc:mysql://localhost/test?user=root&password"
+"=password&useUnicode=true&characterEncoding=8859_1";
            Connection conn = DriverManager.getConnection(sConnStr);
            String sql="select id,link from banner where link!='' and
            type!='' and file!=''";
            Statement stmt = conn.createStatement();
ResultSet rs=stmt.executeQuery(sql);
rs.last();              //把指针置底以便得到广告的数量
Random r=new Random();
//生成小于广告条数量的随机数
int selectedbanner=Math.abs((r.nextInt())%(rs.getRow()));
int i=0;
int id=0;
String link="";

rs.absolute(1);         //把指针返回到顶部
while(true){
    if(selectedbanner==i++){ //当 i 等于随机数时得到该 banner 的 id 和链接
        id=rs.getInt("id");
        link=rs.getString("link");
        break;
    }
    rs.next();
}
//关闭数据库连接
rs.close();
stmt.close();
conn.close();
```

```
            response.setContentType("text/html;charset=gb2312");
            PrintWriter out = response.getWriter();
            //输出显示随机广告条的 HTML 代码，图片用 Servlet 显示
            out.print("<a href='"+link+"'><img border=0
src='showImage?id="+id+"'></a>");

        } catch(Exception e){
            e.printStackTrace();
        }

    }
    //本文件无区别地对待 post 和 get 方法递交来的数据
public void doPost(HttpServletRequest request,HttpServletResponse response)
throws IOException, ServletException{
       doGet(request, response);
    }
}
```

showImage.java 的代码如下所示：

```
//用来显示图片的 Servlet
import java.io.*;
import java.sql.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class showImage extends HttpServlet {
public void doGet(HttpServletRequest request,HttpServletResponse response)
throws IOException, ServletException{
        try{
            //得到要显示的图片的 ID
            String id=request.getParameter("id");
            //注册驱动程序
            Class.forName("com.mysql.jdbc.Driver");
            //初始化数据库连接字符串
            String sConnStr = "jdbc:mysql://localhost/test?user=root&password"
+"=password&useUnicode=true&characterEncoding=8859_1";
            //连接数据库
Connection conn = DriverManager.getConnection(sConnStr);
            //获取图片数据的 SQL 语句
String sql="select type,file from banner where id="+id;
            //建立一个 Statement 对象
Statement stmt = conn.createStatement();
//执行查询
            ResultSet rs=stmt.executeQuery(sql);
            InputStream in=null;
```

```
            //得到图片的类型和图片数据
            if (rs.next())
                in = rs.getBinaryStream("file");
            response.setContentType(rs.getString("type"));
            //建立输出对象
            ServletOutputStream sout = response.getOutputStream();
            byte b[] = new byte[1024];
            //输出图片
            while(in.read(b)!=-1) {
                sout.write(b);
            }
            sout.flush();
            sout.close();
            //关闭数据库连接
            rs.close();
            stmt.close();
            conn.close();
        } catch(Exception e){
            e.printStackTrace();
        }
    }
public void doPost(HttpServletRequest request,HttpServletResponse response)
throws IOException, ServletException{
        doGet(request, response);
    }
}
```

5.4.5 运行结果

运行上述代码，可以添加广告条，如图 5-7 所示。
广告条添加成功后的页面如图 5-8 所示。

图 5-7 添加广告条的页面

图 5-8 广告条添加成功后的页面

连续添加几个广告条后，运行 showBanner 程序，在连续刷新时添加的广告条会随机地出现在页面上，如图 5-9 所示。

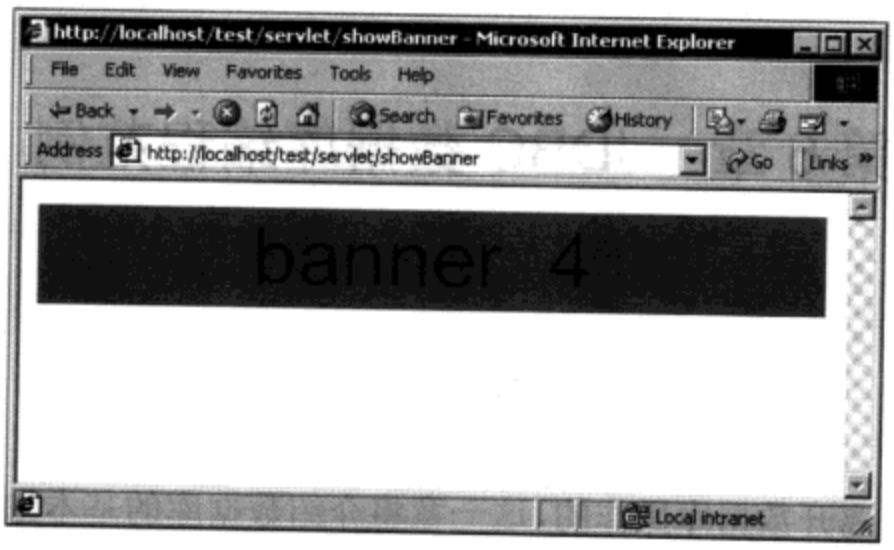


图 5-9　广告条的随机显示

5.5　小结

本章介绍了 Servlet 程序的编写方法，Servlet 是 JSP 的基础，通过本章的学习不但可以掌握 Servlet 的编写方法，而且可以更深入地了解 JSP 的运行原理。

本章的重点如下：

- 使用 Servlet 处理表单数据。
- Servlet 和 JSP 程序之间的通信。
- Servlet 处理文件输入输出流。
- 使用 Servlet 管理广告条的显示。

第 6 章　JDBC 数据库编程范例

6.1　JDBC 简介

服务器编程很少有离开数据库操作的，这一章将介绍数据库的基本知识以及用 JDBC 技术访问数据库的原理。JDBC 经过这几年的发展已经成为 Java 体系中不可缺少的部分。

SQL（Structure Query Language）即结构化查询语言，它的主要功能就是和各种数据库建立联系以及进行沟通。它是按照 ANSI（美国国家标准协会）的规定，作为关系型数据库管理体系的标准语言，目前绝大多数的关系型数据库系统都支持 SQL 语言标准。如 Oracle、Microsoft SQL Server、Sybase，以及我们前边提到的 Mysql 都采用的是 SQL 语言标准，像 Create、Insert、Update、Delete 等 SQL 语句几乎可以在所有的数据库中进行操作。值得注意的是，很多数据库产品对 SQL 语句进行了扩展，使得其功能更强大，具有更高执行效率。但是这些非标准 SQL 语句一般不能在其他的数据库中使用，如果在编程时无法确定最终会使用哪种数据库，或考虑将来有可能采用其他的数据库，也就是说要建立与数据库无关的程序，注意一定要在程序中全部采用标准 SQL 语句。

JDBC（Java 数据库连接，Java Database Connectivity）是一种基于 X/Open 的 SQL 命令级接口，它由一组用 Java 语言编写的类和接口组成，使得程序开发人员可以建立一个与数据库无关、与平台无关的编程接口来建立数据库应用程序。

6.1.1　获得连接

获得数据库连接的标准方法是 DriverManager.getConnection()。这个方法主要包括三个参数，即连接数据库的 URL、使用数据库的用户名和密码。有时也会把用户名和密码放到一个 URL 里，如前边用到过的连接代码：

```
String ConnURL = "jdbc:mysql://localhost/test?"
    +"user="+username+"&password="+password
    +"&useUnicode=true&characterEncoding=8859_1";
Connection conn = DriverManager.getConnection(ConnURL);
```

JDBC 的 URL 提供了一种标识数据库的方法，使相应的驱动程序能识别该数据库并与之建立一个连接。

6.1.2　发送 SQL 语句

一旦与数据库建立好了连接，就可以向该数据库发送 SQL 语句并执行操作。JDBC 对发送的 SQL 语句没有任何限制，所以就要求用户确保 SQL 语句能在数据库中正确执行，否则程序就会抛出异常 SQLException。

JDBC 提供了 3 个向数据库发送 SQL 语句的方法。

- Statement：该对象由 createStatement()创建，用于发送简单的 SQL 语句，如：

```
Statement stmt = conn.createStatement();
```

- PreparedStatement：PreparedStatement 对象用于发送带有一个或多个输入参数的 SQL 语句，它继承并扩展了 Statement 接口。由于使用 PreparedStatement 时 SQL 语句已经经过预编译处理，所以用 PreparedStatement 会比用 Statement 执行效率更高。
- CallableStatement：CallableStatement 由方法 prepareCall()创建，用于执行 SQL 存储过程。存储过程是在数据库中建立并保存，可以像调用函数一样来使用的 SQL 语句。CallableStatement 扩展并继承了 PreparedStatement，同时加入了处理 OUT 参数和 INOUT 参数的方法。

使用 Statement 对象执行 SQL 语句主要有 3 种方法：execute()、executeQuery()、executeUpdate()。

- executeQuery：executeQuery()用于处理返回单个结果集，如执行 Select 语句。
- executeUpdate：executeUpdate()用于执行 insert、update 和 delete 等语句，返回值是一个整数，以标识影响行数。
- execute：execute()方法用于执行返回多个结果集、多个计数或两者的结合。

6.1.3　返回结果

ResultSet 对象是包含符合 SQL 语句中条件的所有行的集合，它通过一套 get 方法提供对这些数据的访问。ResultSet 中有一个标志其当前行数的指针，最初它位于第一行的前边，每调用一次 ResultSet.next()，指针向下移动一行，所以在调用 next()方法之前 ResultSet 的当前行是无意义的。可以通过 ResultSet.getCursorName()方法获得当前指针的名字。

获得当前行某一列的值用 ResultSet 提供的 getXXX 的方法，如：getInt()、getFloat()、getString()等，它的参数可以是 int 型，也可以是 String 型。如果是字符串型，则参数表示要获取的列的列名；如果是整数型，则表示结果集里列的编号。需要注意的是，列的编号从 1 开始，而不是大家习惯的 0。

6.2　Java 数据库连接范例

6.2.1　实例说明

这是一个简单的用 JSP 使用 JDBC 连接数据库的例子，通过这个实例可以看到连接数据库的操作步骤。这个例子和前边的 JSP 部分的分页显示的例子基本相同，只不过前边主要讲解如何传递 page 参数以及实现分页显示，而这个例子主要讲解连接数据库的方法。

前边使用过 Mysql 数据库，Mysql 数据库简单实用，而且是免费的开放源代码的产品，不过毕竟过于简单。这一章的例子主要以使用 Oracle 数据库为主。不过可以看到其实并没有很大的区别，这也是 JDBC 的优点之一。

6.2.2　准备工作

（1）安装 Oracle。Oracle 数据库的安装这里就不详细介绍了，根据提示一步一步地安装

就可以了。

（2）启动 Oracle。根据 Oracle 不同的版本和各自的定制，Oracle 数据库不一定会在操作系统启动时自动启动，有时需要手动启动。启动 Oracle 数据库可以先运行 sqlplus，在需要输入用户名的时候输入：

```
sys/oracle as sysdba
```

登录后输入 startup，如果 Oracle 数据库还没有启动，那么它就会启动了。只不过要稍微等一会儿，Oracle 数据库的启动需要一些时间。打开 Oracle 的监听器，在命令行状态下输入 lsnrctl start 回车。打开 SQLPlus 用 System 用户登录，默认的密码是 manager。Oracle 就会自动启动了。

（3）创建用户 root，密码为 password：

```
create user root identified by password
default tablespace users
temporary tablespace temp
quota unlimited on user;
```

系统会提示用户已经创建好了。下面可以赋予 root 用户权限：

```
grant create session,create table create any index to root;
```

（4）创建一个表。Oralce 和 Mysql 等简单数据库不同，所有的操作都要提交后才能真正得到执行。为了简便起见，可以把 SQLPlus 的 autocommit 打开。单击 SQLPlus 菜单中的选项→环境，如图 6-1 所示。

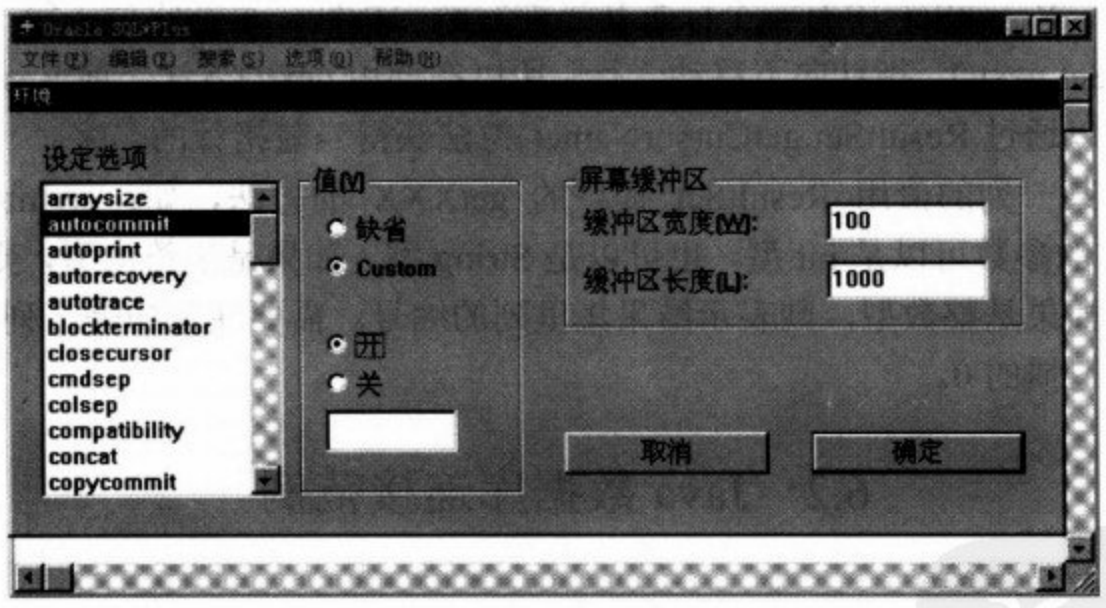


图 6-1 “环境”窗口

然后执行 orauserinfo.sql 程序，方法是在 SQLPlus 中输入：@文件的保存路径，如下所示：

```
SQL>@d:\orauserinfo.sql;
```

orauserinfo.sql 的代码如下所示：

```
CREATE SEQUENCE id start with 1 increment by 1;

CREATE TABLE userinfo (
  id int NOT NULL,
  username varchar(20) NOT NULL,
  password varchar(20) NOT NULL,
  phone varchar(20),
  email varchar(40)
```

```
);

INSERT INTO userinfo VALUES (id.nextval,
'林冲' , 'linchong' , '13664248347' , 'linchong@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'鲁志深' , 'luda' , '13664244523' , 'luda@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'宋江' , 'songjiang' , '13638974623' , 'songjiang@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'扈三娘' , 'husanniang' , '13698706578' , 'husanniang@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'李逵' , 'likui' , '13664893846' , 'likui@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'史进' , 'shijin' , '13676539946' , 'shijin@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'阮小二' , 'ruanxiaoer' , '13665438764' , 'ruanxiaoer@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'卢俊义' , 'lujunyi' , '13664248343' , 'lujunyi@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'武松' , 'wusong' , '13678323849' , 'wusong@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'阮小五' , 'ruanxiaowu' , '13643638765' , 'ruanxiaowu@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'燕青' , 'yanqing' , '13643635745' , 'yanqing@liangshan.com');
INSERT INTO userinfo VALUES (id.nextval,
'孙二娘' , 'sunerniang' , '13643645634' , 'sunerniang@liangshan.com');
```

这样一个 Oracle 数据库就创建好了。

6.2.3　代码分析

userlist.jsp 的代码如下所示：

```
<%@ page language="java" import="java.sql.*,java.util.*" %>
<%@ page contentType="text/HTML;charset=GB2312"%>
<%
//把经常需要修改的数据放在前边，以方便修改
String user="root";      //数据库用户名
String password="password"; //数据库密码
String url=" jdbc:oracle:oci8:@oralin";      //数据库连接字符串
int pagesize=5;          //每页显示的数据量

Connection conn=null;       //数据库连接对象
Statement stmt=null;        //数据库连接状态对象
ResultSet rs = null;        //数据库查询结果集

//下边两个变量是把动态生成的部分集中在一起放入页面
//这样做的好处是当修改页面时就可以直接用 DreamWeaver 编辑了
StringBuffer userInfo=new StringBuffer();
StringBuffer pageInfo=new StringBuffer();
try{
//注册驱动程序
```

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
//获得数据库连接
conn = DriverManager.getConnection (url, user, password);
//由于只处理一个事件，所以设置为自动提交
conn.setAutoCommit(true);
//建立 statement 对象并设置指针可前后移动
stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
}catch(Exception e) {
    System.err.println("数据库连接错误: " + e.getMessage());
}
try{//从表 userInfo 中取出数据
    rs=stmt.executeQuery("select * from userInfo");
}catch(SQLException ex){
    System.err.println("数据库查询错误: "+ex);
}

//获得需要显示的页，如果接收不到参数，默认是第 1 页
String paramPage=request.getParameter("page");
int currentPage=1;
try{
    currentPage=Integer.parseInt(paramPage);
}catch(Exception e){
    currentPage=1;
}

rs.last();                          //把指针置底
int totaluser=rs.getRow();  //获得结果数量
//计算出总页数
int pagecount=(int)Math.ceil((float)totaluser/(float)pagesize);
int i=1;
rs.absolute((currentPage-1) * pagesize + 1);    //把指针放到要显示的第一个数据
//把结果放进变量 userInfo
while(i<=pagesize && !rs.isAfterLast()){
    userInfo.append("<tr bgcolor=#ffffff>\n");
    userInfo.append("<td align=center>"+((currentPage-1)*pagesize+i)+"</td>");
    userInfo.append("<td align=center>"+rs.getString("username")+"</td>");
    userInfo.append("<td>"+rs.getString("phone")+"</td>");
    userInfo.append("<td>"+rs.getString("email")+"</td>");
    userInfo.append("</tr>");
    if(!rs.next())                  //当到达最后一个记录时退出循环
        break;
    i++;
}

try{
    rs.close();             //关闭结果集对象
    stmt.close();           //关闭 Statement 对象
    conn.close();           //关闭数据库连接对象
}catch(Exception e){
    System.err.println(e);
}
```

```
    //把显示翻页的字段放进变量 pageInfo
    pageInfo.append("第"+currentPage+"页 共"+pagecount+"页，共"+totaluser+"个用户 ");
    if(currentPage>1)//在当前页大于 1 时有向前翻页的连接，否则没有
        pageInfo.append(" <a href='userlist.jsp?page="+(currentPage-1)+"'>&lt;
&lt; 上一页</a>");
    else
        pageInfo.append("&lt;&lt; 上一页");
    if(currentPage<pagecount)     //在当前页小于总页数时有向后翻页的连接，否则没有
        pageInfo.append(" <a href='userlist.jsp?page="+(currentPage+1)+"'>下一页
&gt;&gt;</a>");
    else
        pageInfo.append("下一页 &gt;&gt;");
    %>
    <html>
    <head>
    </head>
    <body>
    <table width="80%" border="0" align="center" cellpadding=5 cellspacing=1
bgcolor=#000000>
      <tr align="center" bgcolor=#cccccc>
        <td width="12%">编号</td>
        <td width="23%">真实姓名</td>
        <td width="25%">电话号码</td>
        <td width="40%">邮件地址</td> </tr>
      <%=userInfo%>
    </table>
<table width="80%" border="0" align="center">
    <tr>
        <td align=right><%=pageInfo%></td>
    </tr>
</table>
</body>
</html>
```

6.2.4　运行结果

在 Web 服务器中运行该程序，结果如图 6-2 所示。

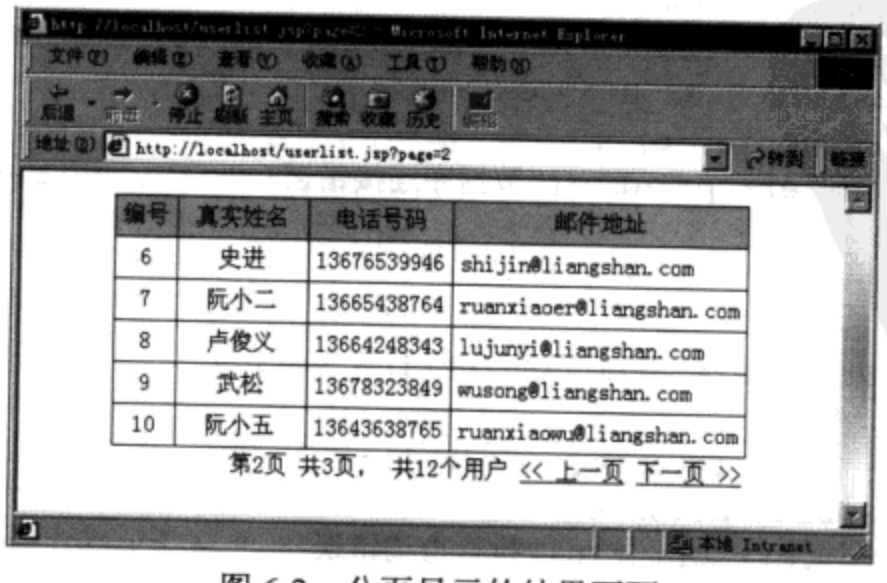


图 6-2　分页显示的结果页面

6.3 JavaBean 封装数据库操作范例

6.3.1 实例说明

通过上边的例子可以看到如何连接数据库并使用数据库，但在实际的应用程序中会有很多程序要访问数据库。如果数据库要修改，哪怕仅仅是修改一下密码，也要修改所有的程序，而且这还涉及到编程中的一个重要的概念“代码复用”。所以在实际的应用程序中一般是把连接数据库的操作放进一个程序中，这样在需要数据库连接时只要调用这个文件就可以了。一般的做法是把数据库连接做成一个 JavaBean 来封装数据库操作。

这个例子就是来演示如何把数据库操作封装到一个 JavaBean 中，并在 JSP 中使用这个 JavaBean。这里还是使用前边分页显示用户列表的例子。

6.3.2 代码分析

dbConnect.java 的代码如下所示：

```
package mybean;
import java.sql.*;
public class dbConnect {
    String user="root";      //数据库用户名
    String password="password"; //数据库密码
    String url=" jdbc:oracle:oci8:@oralin";     //数据库连接字符串
Connection conn = null;
Statement stmt = null;
ResultSet rs = null;
public dbConnect() {
    try {
        //注册驱动程序
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        //获得数据库连接
        conn = DriverManager.getConnection (url, user, password);
        //由于只处理一个事件，设置为自动提交
        conn.setAutoCommit(true);
        //建立 Statement 对象
        stmt= onn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
     ResultSet.CONCUR_READ_ONLY);
    }catch(Exception e) {
        System.err.println("数据库连接错误: " + e.getMessage());
    }
}

public ResultSet executeQuery(String sql) {
    try {
        rs = stmt.executeQuery(sql);
    }catch(SQLException ex) {
        System.err.println("执行查询错误 : " + ex.getMessage());
    }
```

```
        return rs;
    }
    public int executeUpdate(String sql) {
        int affectRows=-1;
        try {
            affectRows = stmt.executeUpdate(sql);
        }catch(SQLException ex) {
            System.err.println("执行更新错误 : " + ex.getMessage());
        }
        return affectRows;
    }
}
```

usebeanuserlist.jsp 的代码如下所示：

```
<%@ page language="java" import="java.sql.*,java.util.*" %>
<%@ page contentType="text/HTML;charset=GB2312"%>
<jsp:useBean id="dbBean" scope="page" class="mybean.dbConnect" />
<%
int pagesize=5;          //每页显示的数据量
//下边两个变量是把动态生成的部分集中在一起放入页面
//这样做的好处是当修改页面时就可以直接用 DreamWeaver 编辑了
StringBuffer userInfo=new StringBuffer();
StringBuffer pageInfo=new StringBuffer();
//使用 JavaBean 执行数据库操作
ResultSet rs=dbBean.executeQuery("select * from userInfo");
//获得需要显示的页，如果接收不到参数，默认是第 1 页
String paramPage=request.getParameter("page");
int currentPage=1;
try{
    currentPage=Integer.parseInt(paramPage);
}catch(Exception e){
    currentPage=1;
}

rs.last();                    //把指针置底
int totaluser=rs.getRow();  //获得结果数量
//计算出总页数
int pagecount=(int)Math.ceil((float)totaluser/(float)pagesize);
int i=1;
rs.absolute((currentPage-1) * pagesize + 1);     //把指针放到要显示的第一个数据
//把结果放进变量 userInfo
while(i<=pagesize && !rs.isAfterLast()){
    userInfo.append("<tr bgcolor=#ffffff>\n");
    userInfo.append("<td align=center>"+((currentPage-1)*pagesize+i)+"</td>");
    userInfo.append("<td align=center>"+rs.getString("username")+"</td>");
    userInfo.append("<td>"+rs.getString("phone")+"</td>");
    userInfo.append("<td>"+rs.getString("email")+"</td>");
    userInfo.append("</tr>");
    if(!rs.next())              //当到达最后一个记录时，退出循环
        break;
```

```
        i++;
    }
    //把显示翻页的字段放进变量 pageInfo
    pageInfo.append("第"+currentPage+"页 共"+pagecount+"页，共"+totaluser+"个用户 ");
    if(currentPage>1)              //在当前页大于 1 时有向前翻页的连接，否则没有
        pageInfo.append(" <a href='usebeanuserlist.jsp?page="+(currentPage-1)+
"'>&lt;&lt; 上一页</a>");
    else
        pageInfo.append("&lt;&lt; 上一页");
    if(currentPage<pagecount)     //在当前页小于总页数时有向后翻页的连接，否则没有
        pageInfo.append("                                                    <a
href='usebeanuserlist.jsp?page="+(currentPage+1)+"'>下一页 &gt;&gt;</a>");
    else
        pageInfo.append("下一页 &gt;&gt;");
    %>
    <html>
    <head>
    </head>
    <body>
    <table width="80%" border="0" align="center" cellpadding=5 cellspacing=1
bgcolor=#000000>
      <tr align="center" bgcolor=#cccccc>
        <td width="12%">编号</td>
        <td width="23%">真实姓名</td>
        <td width="25%">电话号码</a></td>
        <td width="40%">邮件地址</td> </tr>
      <%=userInfo%>
    </table>
    <table width="80%" border="0" align="center">
        <tr>
            <td align=right><%=pageInfo%></td>
        </tr>
    </table>
    </body>
    </html>
```

6.3.3 运行结果

把 dbConnect.java 文件放到$CATALINA_HOME\webapps\test\web-inf\classes\mybean 目录下并使用 javac 编译成 class 文件，把 usebeanuserlist.jsp 放到$CATALINA_HOME\webapps\test\目录下。这样使用浏览器访问 usebeanuserlist.jsp 时会得到与图 6-2 一样的结果。

6.4 数据库连接池

6.4.1 实例说明

在 JDBC 中，一个 Connection 对象表示一个对数据库的连接。建立一个数据库的连接非常耗费时间，因为数据库引擎不但要分配通信和内存资源，而且还要验证用户和设立安全环

境。当然花费的时间是不确定的，长达 2 秒钟的连接时间并不是很少见，所以在所有的客户之间共享一组已经打开的连接会节约很多时间。

池化数据库连接资源对于访问数据库的大多数服务器都是有益的，所以当你的程序能符合下列条件时，就尽量地使用数据库连接池。

- 所有用户通过一个通用的数据库账户访问数据库，而不是每个用户使用一个特定的数据库账户访问数据库。
- 数据库连接只用于单个请求的持续时间，而不是用于来自相同客户的多个请求的持续时间。

一个对于数据库连接池的应用程序应该做的工作是：

（1）获得管理一个或多个连接池的对象的引用。

（2）从连接池中获得一个连接。

（3）使用获得的连接。

（4）将该连接返回给连接池。

一般的连接池对象包括以下几个成员：

```
int MinimumConnections;       //最小连接数
int MaximumConnections;       //最大连接数
Connection [] connections;  //保存连接，不一定是数组，可以是 Stack、Vector 等
Init();                       //初始化连接池
Public Connection getConnection();       //获得一个连接
Public void returnConnection(Connection conn); //归还一个连接
```

最小连接数可以没有，默认为零。如果管理多个连接池还需要有连接池的名字，获得连接和归还连接时需要连接池名字的参数。

6.4.2　编程思路

使用连接池首先要获得对连接池对象的引用，这里的对象应该是惟一的。也就是说无论有多少个进程访问连接池，连接池对象只有一个实例，也就是在整个程序运行中连接池对象要常驻内存。对于一个需要使用数据库连接的进程来讲，绝大部分只需要使用一个连接对象。如果有使用多个连接对象的情况，在设计连接池时必须保证连接池的最大连接数不小于单个进程需要的最大连接数。

6.4.3　代码分析

ConnManager.java 的代码如下所示：

```
package mybean;

import java.io.*;
import java.sql.*;
import java.util.*;
/**
*数据库连接池
*/
public class ConnManager {
    int minConn=2;                                    //连接池最小连接数
```

```
    int maxConn=5;                                  //连接池最大连接数
    String user="root";                             //数据库用户名
    String password="password";                     //数据库密码
    String url="jdbc:oracle:oci8:@oralin";          //数据库连接字符串
    String logFile="d:\\dbpool.log";                //日志文件路径
    PrintWriter loger=null;                         //记录 log 的对象

    int connAmount=0;                               //现有连接的个数
    Stack connStack = new Stack();                  //使用 Stack 保存数据库连接

private static ConnManager instance;
    /**
    * 返回惟一实例。如果是第一次调用此方法，则创建该实例
    */
    public static synchronized ConnManager getInstance(){
        if (instance == null) {
            instance = new ConnManager();
        }
        return instance;
    }


    /**
    *  构造函数实现类的初始化
    *  功能：打开 log 文件，注册驱动程序，根据最小连接数生成连接
    */
    private ConnManager()
    {
        try {
            loger = new PrintWriter(new FileWriter(logFile, true), true);
        }catch (IOException ioe) {
            System.err.println("无法打开日志文件: " + logFile);
            loger = new PrintWriter(System.err);
        }
        //注册驱动程序
        try{
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            log("成功注册驱动程序");
        }catch(Exception e){
            log("无法注册驱动程序");
        }
        //根据最少连接数生成连接
        for(int i=0;i<minConn;i++){
            connStack.push(newConnection());
        }
    }
    /**
    * 将文本信息写入日志文件
    */
    private void log(String msg) {
```

```
        loger.println(new Date() + ": " + msg);
    }
    /**
    * 将不再使用的连接返回给连接池
    */
    public synchronized void freeConnection(Connection con) {
        connStack.push(con);          //将指定连接入栈
        notifyAll();                  //唤醒正在等待连接的进程
        log("归还一个连接到连接池");
    }

    /**
    * 从连接池获得一个可用连接。如没有空闲的连接且当前连接数小于最大连接
    * 数限制，则创建新连接。
    */
    public synchronized Connection getConnection() {
        Connection con = null;
        log("从连接池申请一个连接");
        log("现在可用的连接总数为: "+connStack.size());
        if (!connStack.empty()) {
        // 获取一个可用连接
            con = (Connection)connStack.pop();
        }else if (connAmount < maxConn) {
            con = newConnection();
        }
        else{
            try{
                log("等待连接");
                wait(100000);
                return getConnection();
            }catch(InterruptedException ie){}
        }
        return con;
    }

    /**
    * 创建新的连接
    */
    private Connection newConnection() {
        Connection con = null;
        try {
            con = DriverManager.getConnection(url, user, password);
            connAmount++;
            log("连接池创建一个新的连接");
        }catch (SQLException e) {
            log("无法创建下列 url 的连接: " + url);
            return null;
        }
        return con;
    }
```

```
}
```

要使用这个连接池，可以通过下面的方法：

```
<%@ page language="java" import="mybean.ConnManager"%>
//得到连接池的惟一实例
ConnManager pool=ConnManager.getInstance();
//得到连接
Connection conn=pool.getConnection();
//使用连接
//……

//归还连接
pool.freeConnection(conn);
```

注意连接池文件中的保证连接池只生成一个实例的方法，如下所示：

```
private static ConnManager instance;
public static synchronized ConnManager getInstance(){
    if (instance == null) {
        instance = new ConnManager();
    }
    return instance;
}
```

为了保证不会被错误地使用，构造函数 ConnManager()是私有的。

6.4.4 运行结果

从下面的 log 文件可以看出连接池的运行结果，如下所示：

```
Fri Nov 08 16:34:47 PST 2002: 成功注册驱动程序
Fri Nov 08 16:34:59 PST 2002: 连接池创建一个新的连接
Fri Nov 08 16:34:59 PST 2002: 连接池创建一个新的连接
Fri Nov 08 16:34:59 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:34:59 PST 2002: 现在可用的连接总数为：2
Fri Nov 08 16:34:59 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:34:59 PST 2002: 现在可用的连接总数为：1
Fri Nov 08 16:34:59 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:34:59 PST 2002: 现在可用的连接总数为：0
Fri Nov 08 16:35:00 PST 2002: 连接池创建一个新的连接
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：0
Fri Nov 08 16:35:00 PST 2002: 连接池创建一个新的连接
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：0
Fri Nov 08 16:35:00 PST 2002: 连接池创建一个新的连接
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：0
Fri Nov 08 16:35:00 PST 2002: 等待连接
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：0
Fri Nov 08 16:35:00 PST 2002: 等待连接
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：0
```

```
Fri Nov 08 16:35:00 PST 2002: 等待连接
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：0
Fri Nov 08 16:35:00 PST 2002: 等待连接
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：1
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：2
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：3
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：4
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：4
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：3
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：2
Fri Nov 08 16:35:00 PST 2002: 从连接池申请一个连接
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：1
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：1
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：2
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：3
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：4
Fri Nov 08 16:35:00 PST 2002: 归还一个连接到连接池
Fri Nov 08 16:35:00 PST 2002: 现在可用的连接总数为：5
```

6.5　Java 开发存储过程

6.5.1　实例说明

存储过程是存储在数据库中的一段程序。当创建存储过程时，系统会对其进行编译，并将执行代码存储到数据库中。

（1）设计存储过程的方针。

1）在定义存储过程时，要使用其完成单一并相对集中的任务。

2）在定义存储过程时，不要定义已经由其他特征提供功能的过程。

（2）存储过程的优点。

1）运行速度加快。在网站数据库服务器中，一般交互式的 SQL 命令，每次执行前数据库服务器都要为其建立预编译的过程。而存储过程在第一次执行之后，经过了优化和编译的过程，储存在高速缓存之中。在接下来的运行中可以直接从高速缓存中执行，省去了以后执行的优化和编译阶段，节省了执行过程的大量时间，从而加快了执行速度。

2）网络负荷减少。当客户端发出执行存储过程的请求时，只有执行存储过程的命令在内部网络上传送，当它们到达数据库服务器时，运行存储过程，客户端在网上只接收返回结果或状态信息，所以使得客户机与服务器的通信量降至最小，大大减少了网络负荷。缩短了用户请求的响应时间，避免了用户枯燥的等待。

3）团队开发方便。网站程序编制过程中，JSP、Servlet 等调用存储过程能够减少在程序开发中构造复杂 SQL 语句的难度。由于存储过程的可重用、可共享性，使得其可被多处重复使用，也可以被多个用户共享。在开发中反复使用存储过程，给网站的团队开发带来了极大的方便，而且使站点更易于维护和更新。

4）安全机制放心。存储过程本身有很强的安全机制，只有具有相应的系统权限才能够调用相应的存储过程，或者只访问存储过程而不能够访问其中涉及的表或视图，只通过存储过程中所给出的功能来间接操作数据库。在存储过程的代码中可以包含对信息和数据的合法性检查、对业务规则要求的各种完整性检查等，这无疑给那些安全性较差的网站平台带来了福音。

5）服务用户满意。存储过程可以充分利用数据视点集中的原则，使用户把注意力集中在所关心的数据上、简化用户的数据查询操作、使不同的用户能够多角度“看待”同一数据，能够用存储过程建立非常复杂的查询，以非常复杂的方式更新（update、delete、select、insert）数据库。同时存储过程能够自动对复杂或敏感的事务进行处理，对某些表进行处理，可以保证这些表的数据完整性。

在以前的 Oracle 版本中，开发存储过程是通过 PL/SQL 来完成的。而从 Oracle8i 版本开始，不仅可以使用 PL/SQL 开发存储过程，而且还可以使用 Java 语言来开发存储过程。

6.5.2 代码分析

manipulate_userinfo.java 的代码如下所示：

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;

public class manipulate_userinfo {

  public static void addUser(String name,String pass,String phone,String
email)throws SQLException {
    // 建立到数据库的默认连接
    Connection conn = new OracleDriver().defaultConnection();
    // 构造动态 SQL 语句
     String sql = "INSERT INTO userinfo (id,username,password,phone,email) "
  +"VALUES (id.nextval,?,?,?,?)";
    try {
      // 准备动态 SQL 语句
      PreparedStatement pstmt = conn.prepareStatement(sql);
      // 设置动态 SQL 参数值
      pstmt.setString(1,name);
       pstmt.setString(2,pass);
       pstmt.setString(3,phone);
       pstmt.setString(4,email);
      //执行动态 SQL 语句
```

```
        pstmt.executeUpdate();
        //关闭动态 SQL 语句
        pstmt.close();
      } catch (SQLException e) {}
    }

    public static void deleteUser(int id)throws SQLException {
      //建立到数据库的默认连接
      Connection conn = new OracleDriver().defaultConnection();
      //构造动态 SQL 语句
      String sql = "DELETE FROM userinfo  WHERE id = ?";
      try {
        //准备动态 SQL 语句
        PreparedStatement pstmt = conn.prepareStatement(sql);
        //设置动态 SQL 参数值
        pstmt.setInt(1, id);
        //执行动态 SQL 语句
        pstmt.executeUpdate();
        //关闭动态 SQL 语句
        pstmt.close();
      } catch (SQLException e) { }
    }

    public static void updateUser (int id,String name,String pass,String
phone,String email)throws SQLException{
        //建立到数据库的默认连接
        Connection conn = new OracleDriver().defaultConnection();
        //构造动态 SQL 语句
        String sql = "UPDATE userinfo SET username = ? ,password = ?,phone = ?,email
= ? WHERE id = ?";
        try {
          //准备动态 SQL 语句
          PreparedStatement pstmt = conn.prepareStatement(sql);
          //设置动态 SQL 参数值
          pstmt.setString(1, name);
           pstmt.setString(2, pass);
           pstmt.setString(3, phone);
           pstmt.setString(4, email);
          pstmt.setInt(5, id);
          //执行动态 SQL 语句
          pstmt.executeUpdate();
          //关闭动态 SQL 语句
          pstmt.close();
        } catch (SQLException e) { }
      }
    }
```

6.5.3　生成调用

当开发 Java 存储过程时，首先应该编写 Java 源代码。只有 public static 方法可以作为 Java

存储过程。编写了 Java 源代码之后，接下来应该将 Java 代码及相应的 Java 类装载到 Oracle8i 数据库中。装载 Java 代码及类到 Oracle 数据库可以使用 Oracle 的 loadjava 工具，通过该工具可以快速装载 Java 源代码（.java）、Java 二进制代码（.class）以及 Java 打包文件（.jar），如图 6-3 所示。

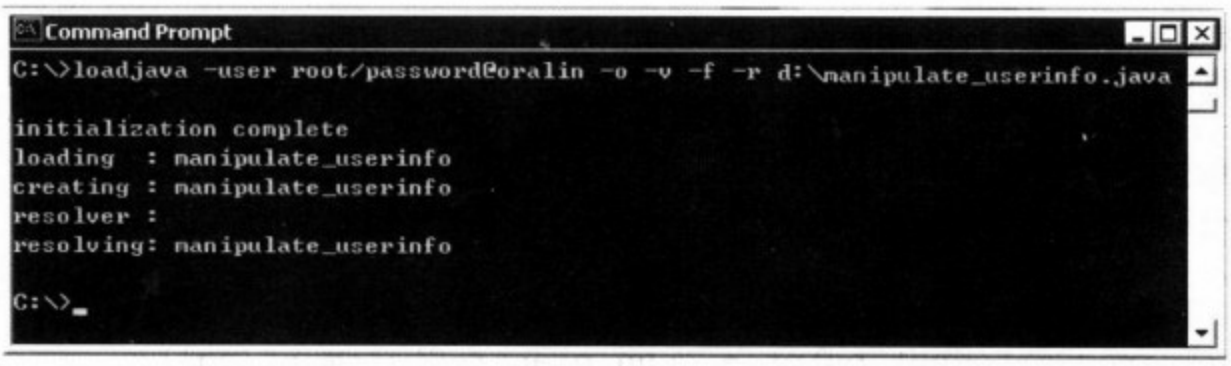


图 6-3 使用 loadjava 工具装载 Java 源代码

存储过程所对应的 Java 方法返回值必须为空（void）。在装载了 Java 类之后，接下来应该生成对 public static 方法的调用说明，最终完成 Java 存储过程的开发工作，如图 6-4 所示。

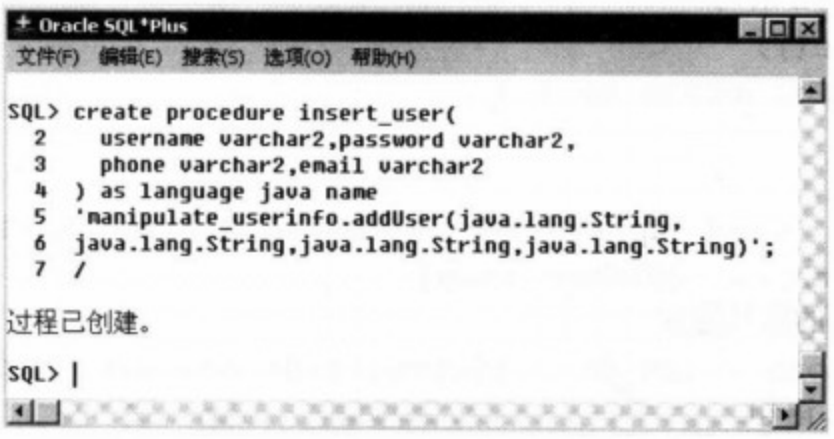


图 6-4 创建添加用户的存储过程

图 6-4 创建了一个添加用户的存储过程，修改用户信息以及删除用户和这个例子差不多，而且比这个还简单，这里就不介绍了。验证这个存储过程是否真的好用，如图 6-5 所示。

图 6-5 验证存储过程

这样，就完成用 Java 开发 Oracle 存储过程。使用这个过程非常简单，只需用 Call 的方法调用存储过程，在程序中可以使用 SQL 语句调用这个存储过程。例如：

```
String sql="call delete_user(3) ";
stmt.excuteQuery(sql);
```

6.6　使用 Java 开发触发器

在大型数据库设计中，会经常用到触发器。它的特点是：一旦被定义，就存在于后台数据库系统（Server，服务器方）中，并会在相应条件下自动地隐式执行，从而使得它的设计既与前台（Client，客户机方）的平台无关，又免除了前台相关的数据操作设计。

数据库触发器（database triggers）是响应插入、更新或删除等数据库事件而执行的过程。它定义了当一些数据库相关事件发生时应采取的动作，可用于管理复杂的完整性约束，或监控对表的修改，或通知其他程序表已发生修改。它的类型有：语句级触发器以及行级触发器。前者可以在语句执行前或执行后被触发。后者在每个触发语句影响的行触发一次。还有 before 和 after 触发的命令，在 insert、update 和 delete 之前或之后执行，引用新旧值进行处理。如果需通过触发器设定插入行中的某列值，则为了访问“新”值，需使用一个触发器 before insert，使用 after insert 则不行。instead of 触发器命令，使用它告诉 oracle 应执行什么操作。

下边通过一个实例讲述一下使用 Java 创建 Oracle 数据库的触发器的过程，这个例子的主要任务是当用户资料（userinfo 表的数据）发生变化时，通过另一个表存储这个变化。为了简便起见，在这里只记录电话号码的修改情况。

因为要保存数据的变化，首先需要建立一个表来保存数据。如下面的代码行所示：

```
CREATE TABLE phonenumbackup(
    User_id int,
    Old_num varchar(20),
    New_num carchar(20)
);
```

6.6.1　编写代码

Trigger.java 的代码如下所示：

```
import java.sql.*;
import java.io.*;
import oracle.jdbc.driver.*;
public class Trigger {
  public static void backup_phone_num(int id,String old_num,String new_num)
  throws SQLException {
    //建立到数据库的默认连接
    Connection conn = new OracleDriver().defaultConnection();
    String sql = "INSERT INTO phonenumbackup VALUES(?,?,?)";
    //使用 try ... catch 语句获取并抛出异常
    try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setInt(1, id);
      pstmt.setString(2, old_num);
      pstmt.setString(3, new_num);
      pstmt.executeUpdate();
      pstmt.close();
```

```
    } catch (SQLException e) {}
  }
}
```

可以看出这段代码和建立存储过程的代码并没有什么区别。触发器的关键是触发，而触发是由数据库控制的，Java 代码只能告诉数据库触发以后干什么。

6.6.2 生成调用

在这里首先要做的还是装载 Java 代码，如图 6-6 所示。

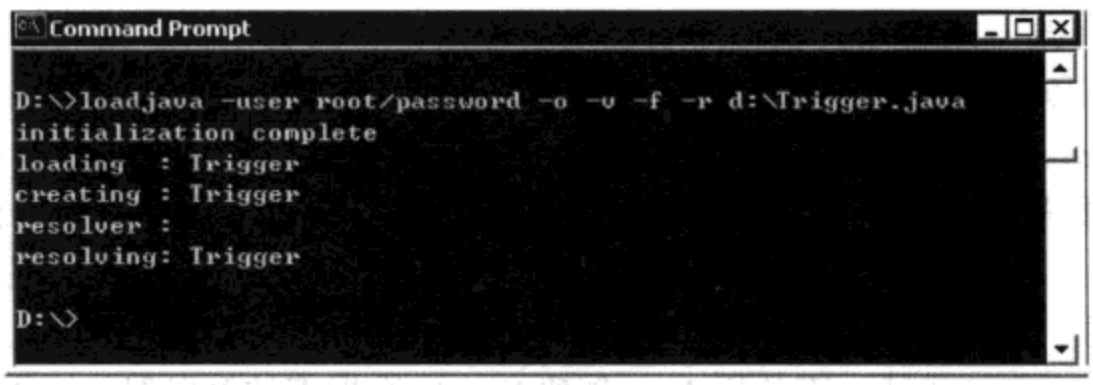


图 6-6 装载 Java 代码

装载了 Java 代码以后，要建立数据库的调用，数据库收到触发信息后调用 Java 代码。首先还是要建立过程，这样数据库收到触发信息后才会调用这个过程。建立过程的方法和建立存储过程的方法也没有什么不同，如图 6-7 所示。

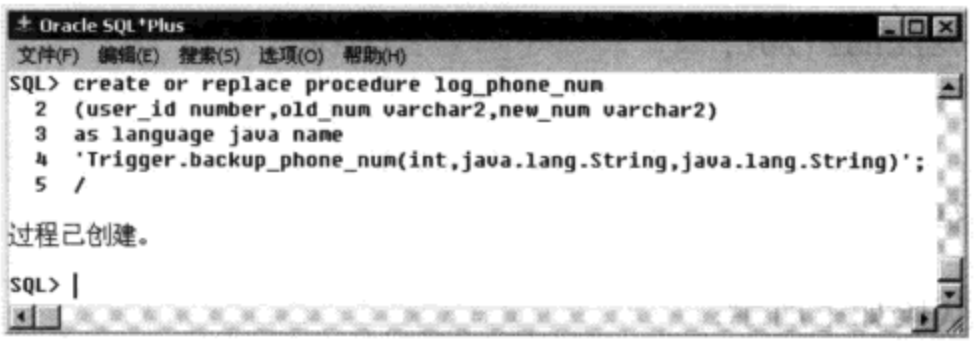


图 6-7 为该触发器建立一个过程

然后才是建立数据库的触发器，当发生触发事件后调用下面的过程，如图 6-8 所示。

正如 SQLPlus 告诉我们的“触发器已创建”，那么如何检验触发器是否正常工作呢？非常简单，只要输入 SQL 语句，更新数据就可以了。可以参考下边的测试，如图 6-9 所示。

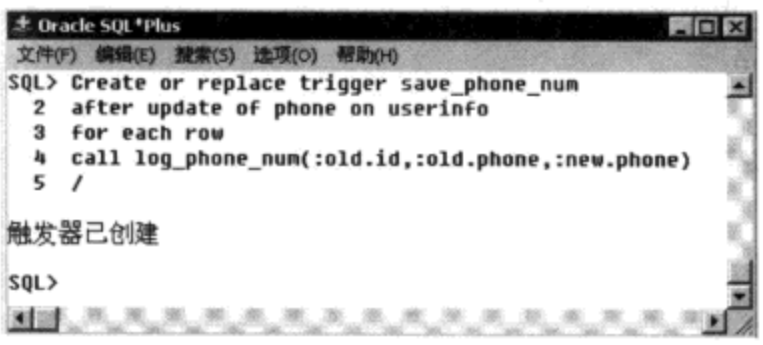


图 6-8 创建一个触发器

图 6-9 触发器的使用

6.7　使用 SQLJ

今天的企业级应用程序通常运行在多数据库平台上，例如 Oracle、DB2、Sybase、SQL Server、Informix 等。在这种环境里，代码的可移殖性和可维护性正变得越来越重要。从一个数据库平台移植代码到另一个数据库平台是非常复杂的并且要花费大量时间，因为数据库供应厂商都使用他们自己专有的过程语言（例如 Oracle 使用 PL/SQL，而 Sybase 和 SQL Server 使用 Transact - SQL）。

近几年来，Oracle 开发人员一直在使用 PL/SQL（一种提供了到关系数据库语言 SQL 的过程扩展部分的语言）来构建管理大量的数据阵列的复杂系统。但是，用 PL/SQL 编写的存储过程只能在 Oracle 数据库运行。SQL 开发人员有了一种编写代码的强有力的工具，可以很容易地移植到其他的数据库，这就是 Java，因其在跨平台开发和国际互联网络上的强大功能使它成为流行的开发语言。Java 承诺的统一的、可移植的应用软件开发解决办法可以在简单的、低成本的 IT 基础设施上执行，所以主要的开发工具供应厂商和设备供应者都支持 Java。居于领导地位的软件供应厂商，例如 Oracle 和 IBM，都在他们的数据库和其他的应用程序平台上整合了 Java 虚拟机（JVM）。Oracle 数据库管理系统公司在 Oracle8i 中引入了对 Java 的扩展支持功能。在 Oracle 中，有两种使用 Java 的基本方法：其中之一的 JDBC 在前边的章节已经介绍过了，第二种方法就是 SQLJ，在这一节中就简要地介绍一下 SQLJ。可以说，它提供了比 JDBC 更高的程序员劳动生产率。

6.7.1　什么是 SQLJ

SQLJ 是一个与 Java 编程语言紧密集成的嵌入式 SQL 版本，这里的嵌入式 SQL 是用来在其宿主通用编程语言（如 C、C++、Java、Ada 和 COBOL）中调用 SQL 语句。在一个嵌入式 SQL 程序中，SQL 语句可以像其宿主语言中的程序构造一样得到支持。当然，C 和 C++也可以通过宿主语言函数调用，即通过开放数据库连接接口（ODBC）调用 SQL 语句。与此相似，Java 程序可以通过 Java 数据库连接（JDBC）方法调用来调用 SQL 语句。但是，这一类的函数调用接口较之嵌入式 SQL 接口是一个水平相当低的程序接口。因为这时的 SQL 语句是作为字符串参数传送给函数，而不是直接在宿主语言中嵌入代码的。

在 Oracle8i 中，特别强调了 Java 编程语言和 Internet/Intranet 数据库应用程序的开发。Oracle8i 的重要特性之一就是全面和高效地支持 SQLJ。与 Oracle8 一样，Oracle8i 不仅提供对关系型数据库处理的强有力的支持，同时还支持一些对象—关系型结构，如集合类型、用户定义类型和对象类型。

SQLJ 由一系列定义了 SQL 与 Java 之间相互作用的子句和程序扩充组成。SQLJ 是在 Java 编程语言中静态嵌入 SQL。换句话说，一个 SQLJ 程序是一个包含静态嵌入式 SQL 语句的 Java 程序。请注意，在静态嵌入式 SQL 中，所有嵌入到程序中的 SQL 语句在编译时都是已知的。而在动态嵌入式 SQL 中，至少有一些 SQL 语句是等到运行时才被确定的。SQLJ 通过这种静态嵌入式 SQL 模型完善了 JDBC 动态嵌入式 SQL 模型，这是因为 JDBC 只提供给 Java 一个动态嵌入式 SQL 接口，而 SQLJ 提供了一个静态嵌入式 SQL 接口。从此，通过使用 SQLJ，Java 程序员在 Java 和 SQL 之间有了两种不同的程序接口：JDBC 和 SQLJ。而其他编程语言，诸如

C、C++、FORTRAN、COBOL 和 Ada 基本上使用同一种嵌入式 SQL，而 SQLJ 作为一个与 ANSI 标准多少有些不同的嵌入式 SQL 一直为 Java 所专用。这就引出了一个问题：为什么惟独 Java 有其自己的嵌入式 SQL，而其他的编程语言共用一个嵌入式 SQL？原因之一是 SQLJ 与 Java 的紧密耦合，特别是 Java 类可在 SQL 表中作为字段类型使用。另外，SQLJ 提供了游标结构的一种强类型版本，又称为迭代器。这种迭代器结构巧妙地集成到 Java 语言中，每一个迭代器就是一个 Java 类。同时要注意的是，与 SQLJ 不同，其他编程语言的嵌入式 SQL 同时包括静态和动态 SQL 构造。

SQLJ 作为一个新的标准的出现是多个厂商共同努力以求在 Java 程序中提供对静态嵌入式 SQL 支持的结果。一个“纯”SQLJ 程序是静态的，因为所有的 SQL 语句都必须在编译前确定。但是，在一个 SQLJ 语句之内包含 JDBC 调用会使该程序成为动态的，因为程序可将 SQL 语句组织成一个字符串，然后将该字符串传送给 JDBC 方法去执行。SQLJ 可用来在 JServer 环境下执行存储过程、触发器和类，也可用于 Enterprise JavaBeans 和 CORBA。

SQLJ 由一系列子句组成，这些子句扩展了 Java 程序。语言的标准说明是一个联合说明，它由包括 IBM、Compaq/Tandem、JavaSoft、Oracle、Sybase 和 Informix 在内的主要数据库工具厂商和数据库厂商联合支持。SQLJ 提供了在访问数据库的客户端和中间件以及在利用 Java 的数据服务器端开发应用程序的方法。SQLJ 应用程序是可移植的，并且可以使用标准 JDBC 驱动程序与多数厂商的数据库进行通信。

当编写一个 SQLJ 程序（源代码）时，所编写的是一个 Java 程序并按一定的语法规则嵌入 SQL 语句，这套语法规则说明了怎样将 SQL 语句嵌入到 Java 源代码中。然后运行一个 SQLJ 翻译器将 SQLJ 程序转换为一个标准 Java 程序。一个 Oracle SQLJ 翻译器在原理上与其他 Oracle 嵌入式 SQL 预编译器类似。SQLJ 翻译器完成下列工作：

- 对嵌入式 SQL 构造进行语法检查。
- Java 和 SQL 数据类型检查。

在翻译源代码时，SQLJ 翻译器用 SQLJ 运行时库中的调用来替代嵌入式 SQLJ 语句，该运行时库真正实现 SQL 操作。这样翻译的结果是得到一个可使用任何 Java 翻译器进行编译的 Java 源程序。一旦 Java 源程序被编译，Java 执行程序就可在任何数据库上运行。

SQLJ 运行环境由纯 Java 实现的小 SQLJ 运行时库（这里的小，意指其中包括少量的代码）组成，该运行时库转而调用相应数据库的 JDBC 驱动程序。

一个 SQLJ 程序可以在多种环境下运行。所编写的 SQLJ 程序可以在客户端进行，不管是在瘦客户端如 Web 浏览器和网络计算机上，还是在一个工作站或 PC 上。由于这种位置的透明性，使得用户可以很方便地将 SQLJ 程序从一个地方移植到另一个地方或从一个系统移植到另一个系统。

SQLJ 的优点如下：

- 紧凑的代码。SQLJ 代码比 JDBC 代码更加紧凑并且无差错，在编译时对语法和语义进行检查。SQLJ 翻译器提供了类型检查和模式对象检查来找出在 SQL 语句中的语法错误或遗漏、拼错这样的错误，这是在编译过程中而不是在运行过程中进行的。因此，使用 SQLJ 编写的程序比使用 JDBC 编写的程序更加健壮。
- 多厂商互用性。SQLJ 语法是由主要的软件供应厂商开发和支持的。因为 SQLJ 程序使用运行时 JDBC 调用访问数据库，所以 SQLJ 可以访问任何 JDBC 驱动程序可以实

现的数据库服务器。

- 灵活的部署。因为 SQLJ 运行时程序库是基于 Java 的程序，所以 SQLJ 应用程序可以在任何 JDBC 配置环境中进行配置，例如瘦客户端、中间层或是数据库服务器等。
- 供应厂商具体定制。SQLJ 通过后续的 Java 字节码的定制支持供应厂商具体产品的特色和扩展。它可以用来改善 SQL 查询语言的执行性能，使用具体供应厂商提供的性能或功能上的扩展，而不用考虑 SQLJ 程序的变化、调试和运行记录等情况。

SQLJ 访问数据库需要如下几个步骤：

（1）导入 SQLJ 类（sqlj.runtime.*；sqlj.runtime.ref.*）。

（2）导入 Oracle SQLJ 定制器（sqlj.runtime.*）。

（3）导入 JDBC 类（java.sql.*）。

（4）建立默认或指定数据库的连接（DefaultContext 对象、ConnectionContext 对象）。

（5）执行上下文（ExecutionContext 对象），并处理执行结果。

（6）关闭数据库连接。

6.7.2　准备工作

在使用 SQLJ 之前，必须把下边的压缩文件加入到 CLASSPATH 才能保证文件能正确编译执行。在 JSP 中使用 SQLJ，还要注意加入到 Tomcat 等 Web 服务器的 CLASSPATH 中。

```
$ORACLE_HOME/sqlj/lib/translator.zip
$ORACLE_HOME/sqlj/lib/runtime.zip
```

既然要操作数据库，创建表是必不可少的一步。使用 SQLPlus 直接导入此文件，或者拷贝粘贴也可以执行。bookstore.sql 的代码如下所示：

```
create table bookstore (
id int,
bookName varchar(200),
price number(6,2)
);
```

这个表很简单，用来保存书的信息，在这里只有标识号、书名和价格 3 个字段。

6.7.3　代码分析

SqljExample.sqlj 的代码如下所示：

```
import sqlj.runtime.ref.*;
import java.sql.*;
import oracle.sql.* ;
import oracle.sqlj.runtime.Oracle;
public class SqljExample
{
    String db_url="jdbc:oracle:oci8:@oralin";
    String db_user="root";
    String db_password="password";
    boolean db_auto_commit=true;
    /* 方法 dbConnect()用于建立数据库连接 */
    public static void dbConnect() throws SQLException
    {
```

```
    Oracle.connect(db_url,db_user,db_password,db_auto_commit);
}
//方法 insert-book 用于将数据插入到 bookstore 表中
public static void insert_book(int ID,String BookName,float Price){
    try{
        NUMBER id=new NUMBER(ID);
        CHAR name=new CHAR(BookName,CharacterSet.make(870));
        NUMBER price=new NUMBER(Price);
        insert_book(id,name,price);
    }catch(Exception e){
        e.printStackTrace();
    }
}
// 方法 insert_book()用于将数据插入到表中
public static void insert_book(NUMBER id,CHAR name,NUMBER price)
throws SQLException
{
        #sql{insert into bookstore values (:id,:name,:price)};
}
//方法 update_price()用于更新价格信息
public static void update_price(int id,float price) throws SQLException
{
    #sql{update bookstore set price=:price where id=:id};
}
//方法 delete_book()用于删除书的信息
public static void delete_book(int id) throws SQLException
    {
    #sql{delete from bookstore where id=:id};
    }
}
```

看了上边的文件就会发现，原来 SQLJ 这么简单。SQLJ 设计的初衷就是让不熟悉 Java 的人也能使用 Java，但还是有许多问题难以达到。从上边的代码可以看到，有两个 insert_book 函数，也就是面向对象编程中的术语“过载”，为什么要过载这个函数呢？从代码中可以看出是为了类型转换，SQL 中的数据类型不一定等同于 Java 中的数据类型。这个例子中的数据类型转换还有一个明显的好处，由于 Oracle 数据库在输入字符串时，字符串中不可以含有单引号（大多数据库都如此），所以在进行数据库操作时当有字符串操作时，程序必须要检查字符串中有没有非法字符。一般情况下使用 replace 函数把单引号转换为“\'”（在 SQL Server、Access 中要换成两个单引号）就可以了。使用上边的函数，Oracle 的 CHAR 就自动转换了。

另外，在进行类型转换时需要注意如下几点：

- Java 数值类型（如 int、float、double）不是“亲数据库”的，它们在值为空时不是 null 而是“0”，这会引起数据库的误解。
- 使用 Java 的对象类型（如 Interger、Float、Double）或 JDBC SQL 类型（java.sql.Type.*）可以避免上边提到的问题。
- SQL 数据类型和 Java 的数据类型精度不同，转换时可能出现精度丢失。
- 在程序中声明 Java 结构时总是使用 oracle.sql.* 中的数据类型是一个不错的办法，这

样就能解决上边所有的问题。

6.7.4　运行结果

首先编译 SqljExample.sqlj 文件，不是用 javac 而是使用 Oracle 的 sqlj 命令，如下所示：

```
sqlj -user root/password SqljExample.sqlj
```

如果正确地配置了 CLASSPATH 就会发现编译结果，文件夹中生成了几个文件：SqljExample.java、SqljExample.class、SqljExample_SJProfile0.ser 和 SqljExample_SJProfileKeys.class。使用类似下边的文件就可以使用这个 SQLJ 了。BookStoreManage.java 的代码如下所示：

```
public class BookStoreManage
{
    public static void main(String[] args){
        try{
            //生成 SqljExample 实例
            SqljExample se=new SqljExample();
            //连接数据库
            se.dbConnect();
            se.insert_book(1,"红楼梦",65.50f);
            se.insert_book(2,"水浒传",38.00f);
            se.update_price(1,34.00f);
            se.delete_book(2);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

编译运行 BookStoreManage.java 文件，然后查看数据库中的结果，就可以证明 SQLJ 确实能工作了。在 Java 程序或 JSP 文件中使用 SQLJ 只要按照上边的方法就可以了。

SQLJ 的功能还有很多，上边提到的使用 Java 编写 Oracle 的存储过程也可以使用 SQLJ 来实现，这里就不详细介绍了，有兴趣的话可以参考有关 Oracle 数据库的书籍。

6.7.5　SQLJ 的要点

- ConnectionContext 和 DefaultContext。ConnectionContext 对象保留了底层的 JDBC 连接对象的实例，Java 程序可以使用特定的 ConnectionConnext 对象执行数据库操作。DefaultContext 对象是它的一个子类，如果程序中只使用一个数据库连接对象，可以使用 DefaultContext 简化程序。
- ExecutionContext。使用 ConnectionContext 或 DefaultContext 执行的所有数据库操作都要使用 ExecutionContext 对象，ExecutionContext 保持最近数据库操作的状态。
- 嵌入的 SQL 语句和数据库操作。SQLJ 是一个域编译器，使用 SQLJ 可以在 Java 类定义中嵌入完整的 SQL 语句和其他数据库操作。包含内嵌的数据库操作的程序通常被称为“宿主程序”，内嵌的语句被称为 SQLJ 的可执行语句。
- 宿主表达式的基本格式。

（1）以冒号开始。

（2）最简单的情况下，宿主表达式的说明是宿主变量的名称，否则宿主表达式必须用大

括号分开。

（3）如果宿主表达式的模式为 OUT 或 INOUT，该变量必须是已分配的。

- SQLJ 源文件。SQLJ 的源文件必须以 sqlj 为扩展名，SQLJ 源文件可以声明为一个公有类，且文件名必须和这个类的名称保持一致。
- Oracle 的 SQLJ 翻译器运行的基本步骤如下：

（1）翻译 SQLJ 的源文件。

（2）进行 SQL 语法检查。

（3）连接数据库并进行 SQL 语义检查。

（4）生成配置表。

（5）生成 Java 源文件和类文件。

6.8 小结

本章主要介绍了使用 Java 语言开发数据库应用程序，并以 Oracle 数据库为主介绍了使用 Java 语言开发数据库的存储过程和触发器。通过本章的学习并不能使你成为使用 Oracle 数据库的 Java 高手。正如本书的目的一样，它能让你对这种技术有一个概念上的了解。如果有兴趣学习更多的数据库服务器的功能和 Java 语言相结合的技术，可以参阅其他的专门书籍。

本章主要讲述了以下几个概念：

- 使用 JDBC 访问数据库。
- 使用数据库连接池。
- 使用 Java 创建 Oracle 数据库的存储过程及触发器。
- 使用 SQLJ 访问数据库。

本章后边几节有些偏重于 Oracle 数据库而不是 Java 了，但毕竟 Oracle 数据库提供了如此多的 Java 语言支持，了解一些也是有必要的。

第 7 章　使用 Java 进行 XML 编程

7.1　XML 简介

XML（eXtensible Markup Language，可扩展标记语言）是由 W3C（World Wide Web Consortium，互联网联合组织）于 1998 年 2 月发布的一种标准，同 HTML 一样是 SGML（Standard Generalized Markup Language，标准通用标记语言）的一个简化子集。由于它将 SGML 的丰富功能与 HTML 的易用性结合到了 Web 的应用中，自推出以来迅速得到软件开发商的支持和程序开发人员的喜爱，显示出了强大的生命力。

7.1.1　XML 与 HTML 的比较

Internet 提供了全球范围的网络互连与通信功能，Web 技术的发展更是一日千里，其丰富的信息资源给人们的学习和生活带来了极大的便利。特别是应运而生的 HTML（超文本标记语言），其简单易学、灵活通用的特性，使人们发布、检索、交流信息都变得非常简单，从而使 Web 成了最大的环球信息资源库。然而，电子商务、电子出版、远程教育等基于 Web 的新兴领域的全面兴起使得传统的 Web 资源更加复杂化、多样化，数据量的日趋庞大对网络的传输能力也提出更高的要求。同时，人们对 Web 服务功能的需求也有了更高的标准，比如用户需要对 Web 进行智能化的语义搜索和对数据按照不同的需求进行多样化显示等个性化服务；公司和企业要为客户创建和分发大量有价值的文档信息，以降低生产成本，以及对不同平台、不同格式的数据源进行数据集成和数据转化等，这些需求越来越广泛和迫切。

传统的 HTML 由于自身特点的限制，不能有效地解决上述问题。作为一种简单的表示性语言，HTML 只能显示内容而无法表达数据内容。而这一点恰恰是电子商务、智能搜索引擎所必需的。另外，HTML 语言不能描述矢量图形、数学公式、化学符号等特殊对象，在数据显示方面的描述能力也不尽如人意。最重要的是 HTML 只是 SGML 的一个实例化的子集，它的可扩展性差，用户根本不能自定义有意义的标记供他人使用。这一切都成为 Web 技术进一步发展的障碍。

SGML 是一种通用的文档结构描述标记语言，为语法标记提供了异常强大的工具，同时具有极好的扩展性，因此在数据分类和索引中非常有用。但 SGML 复杂度太高，不适合网络的日常应用，加上开发成本高、不被主流浏览器所支持等原因，使得 SGML 在 Web 上的推广受到阻碍。在这种情况下，开发一种兼具 SGML 的强大功能、可扩展性以及 HTML 的简单性的语言势在必行，由此诞生了 XML 语言。

7.1.2　XML 的优缺点

XML 的优势之一是它允许不同组织、个人建立适合自己需要的标记集合，并且这些标记可以迅速地投入使用。这一特征使得 XML 可以在电子商务、政府文档、司法、出版、CAD/CAM、

保险机构、厂商和中介组织信息交换等领域中一展身手。针对不同的系统、厂商提供各具特色的独立解决方案。

XML 的最大优点在于它的数据存储格式不受显示格式的制约。一般来说，一篇文档包括 3 个要素：数据、结构和显示方式。对于 HTML 来说，显示方式内嵌在数据中，这样在创建文本时，不必时时考虑输出格式。如果因为需求不同而需要对同样的内容进行不同风格的显示时，要从头创建一个全新的文档，重复工作量很大。此外 HTML 缺乏对数据结构的描述，对于应用程序理解文档内容、抽取语义信息都有诸多不便。

XML 把文档的三要素独立开来分别处理。首先把显示格式从数据内容中独立出来，保存在样式表文件（StyleSheet）中，这样如果需要改变文档的显示方式，只要修改样式表文件就行了。XML 的自我描述性质能够很好地表现许多复杂的数据关系，使得基于 XML 的应用程序可以在 XML 文件中准确高效地搜索相关的数据内容，而忽略其他不相关部分。XML 还有其他许多优点，比如它有利于不同系统之间的信息交流，完全可以充当网际语言，并有希望成为数据和文档交换的标准机制。

当然，XML 作为一个新建立的标准，还有许多不足之处。它在强调了数据结构的同时，在语义表达能力方面略显不足。例如定义了<地址>这样一个标记，如果不是在文档中实际定义内容，我们就无法知道是要表达家庭住址还是 E-mail 地址。另外，XML 的有些技术尚未形成统一的标准，充分支持 XML 的应用处理程序很少，甚至浏览器对 XML 的支持也是有限的。所以，XML 还不能完全取代 HTML，毕竟 HTML 是最为方便、快捷的网上信息发布方式。况且 HTML 是描述数据显示的语言，而 XML 是描述数据及其结构的语言，二者在功能上也是截然不同的。HTML 有大约一百个不同的标记，大多数标记又有相当多的标签，每个标签还有几种、十几种不同的取值。乍看起来，XML 比 HTML 更加强大，所以你会觉得它有更多的标记。其实不然，XML 预定义的标记数目近乎为 0，只是描述了一个用来定义标记集的方法。当我们用这个方法规定好一个标记集，并根据这些规定填入文本内容后，这些标记就和纯文本一起构成了一个 XML 文件。

下边是一个 XML 的例子。bookStore.xml 的代码如下所示：

```
<?xml version="1.0" encoding="GBK" standalone="yes"?>
<bookList>
    <book>
        <name>Java 编程入门</name>
        <author>张三</author>
        <publishDate>2003-1-5</publishDate>
        <price>35.0</price>
    </book>
    <book hot="true">
        <name>XML 在 Java 中的应用</name>
        <author>李四</author>
        <publishDate>2002-12-16</publishDate>
        <price>92.0</price>
    </book>
</bookList>
```

尽管 XML 允许你“随心所欲”地建立自己的标记集，但一旦这个标记集建立起来，就不能再那么“随心所欲”地编写你的 XML 文件了，你必须严格遵守 XML 的语法和自己的标记

集的规定。当把一个 XML 文件提交给一个 XML 处理程序时，为了保证处理程序能够很好地理解它，XML 必须遵守 XML 的语法标准。最起码，这个 XML 文件应该是“格式良好的”（well-formed）。否则，处理程序将会不知所措，无法再进一步进行下去，这时你能得到的全部结果，也只不过是一个“致命错误”的抱怨而已。

在 XML 中，“格式良好”有着明确的标准，即要遵守 XML 规范中的语法规则。无论是从物理结构上讲，还是从逻辑结构上讲，XML 都必须符合规范才能被正确解释处理。

当你开始着手写一个 XML 文件时，最好以一个“XML 声明”作为开始。之所以说“最好”，是因为 XML 声明在文件中是可选内容，可加可不加，但 W3C 推荐加入这一行声明。因此，作为一个良好的习惯，我们通常把 XML 声明作为 XML 文件的第一行。它的作用就是告诉 XML 处理程序：“下面这个文件是按照 XML 文件的标准对数据进行标记的”。

一个最简单的 XML 声明是这样的：

```
<?xml version=1.0 ?>
```

在该声明中还有两个可选属性，分别是 standalone 和 encoding。因此，一个完整的 XML 声明是这样的：

```
<?xml version="1.0" standalone ="no" encoding="GB2312"?>
```

- version：在一个 XML 的处理指示中必须包括 version 属性，指明所采用的 XML 的版本号，而且，它必须在属性列表中排在第一位。由于当前的 XML 最新版本是 1.0，所以我们看到的无一例外都是 version="1.0"。在 2002 年 10 月 15 日以候选推荐标准（Candidate Recommendation）形式，发表了 XML1.1 版本。
- standalone：这个属性表明该 XML 文件是否和一个独立的标记声明文件配套使用。因此，如果该属性置为 yes，说明没有另外一个配套的 DTD 文件来进行标记声明。相反，如果这个属性置为 no，则有可能有这样一个文件。
- encoding：所有的 XML 语法分析器都要支持 8 位和 16 位的编码标准。不过，XML 可能支持一个更庞大的编码集合。在 XML 规范的 4.3.3 节中，列出了一大堆编码类型。但一般我们用不到这么多编码，最常用的就是简体中文码：GB2312，也叫“国标码”，采用哪种编码取决于你文件中用到的字符集。如果标签是用中文来写的，这时就必须要在声明中加上 encoding = "GB2312"的属性。

元素是 XML 文件内容的基本单元。从语法上讲，一个元素包含一个起始标记、一个结束标记以及标记之间的数据内容。其形式是：<标记>数据内容</标记>。元素中还可以再嵌套别的元素。上边例子的元素"bookList"中就嵌套了元素"book"，而元素"book"中又嵌套了"name"、"author"等元素。其中"bookList"元素包含了文件中所有的数据信息，称之为根元素。"<book>"、"<name>"、"<author>"等称为 XML 的标记，XML 对于标记的语法规定要比 HTML 严格得多。

- 标记必不可少。任何一个格式良好的 XML 文件中至少要有一个元素。也就是说，标记在 XML 文件中是必不可少的。
- 大小写有所区分。在标记中必须注意区分大小写。在 HTML 中，标记<table>和<TABLE>是一回事，但在 XML 中，它们是两个截然不同的标记。
- 要有正确的结束标记。结束标记除了要和开始标记在拼写和大小写上完全相同之外，还必须在前面加上一个斜杠“/”。因此，如果开始标记是<TABLE>，则结束标记应该写作</TABLE>。

- XML 严格要求标记配对。因此，HTML 中的
、<HR>的元素形式在 XML 中是不合法的。不过，为了简便起见，当一对标记之间没有任何文本内容时，可以不写结束标记，而在开始标记的最后冠以斜杠“/”来确认，这样的标记称为“空标记”。例如，HTML 中的标记<HR>在 XML 中的使用方式应该是：<HR/>。
- 标记要正确嵌套。在一个 XML 元素中允许包含其他 XML 元素，但这些元素之间必须满足嵌套性。因此，下面这么写是错误的：

```
<BOOK>
<NAME>JAVA 编程入门</BOOK>
</NAME>
```

- 标记命名要合法。标记应该以字母、下划线“_”或冒号“：”开头，后面跟字母、数字、句号“.”、冒号、下划线或连字符“-”，但是中间不能有空格，而且任何标记不能以“xml”起始。另外，最好不要在标记的开头使用冒号，尽管它是合法的，但可能会带来混淆。
- 有效使用属性。标记中可以包含任意多个属性。在标记中，属性以名称/取值对出现。属性名不能重复，名称与取值之间用等号“=”分隔，且取值用引号引起来。

7.1.3 XML 的注释

在 XML 中，注释的方法和 HTML 完全相同，不过以下几点还是要注意：

- 在注释文本中不能出现字符“-”或字符串“--”，XML 处理器可能把它们和注释结尾标志“-->”相混淆。
- 不要把注释文本放在标记之中。
- 注释不能嵌套。在使用一对注释符号表示注释文本时，要保证其中不再包含另一对注释符号。

7.2 DTD 和 Schema

7.2.1 DTD 简介

前边我们详细介绍了一个“格式良好的”XML 文件应该满足哪些要求。“格式良好”是对 XML 文件的基本要求，它使得 XML 文件结构清晰、完整，便于处理程序进行解析，进一步可以简化处理程序的编写工作，并加快浏览速度、减少浏览所需占用的内存空间。

然而，即便你已经可以保证写的是一个“格式良好的”XML 文件了，它仍然未必能够体现 XML 的精髓。XML 的精髓是什么呢？对，就是我们前面讲到的基于信息描述的、能够体现数据信息之间逻辑关系的、可以确保文件的易读性和易搜索性的自定义标记！一个完全意义上的 XML 文件不仅应该是“格式良好的”，而且还应该是使用了这些自定义标记的“有效”的 XML 文件。

一个“有效的”文件首先应该是“格式良好”的。但这还远远不够，它还要往前更进一步。一个 XML 文件必须遵守文件类型描述 DTD（文档类型定义，Document Type Definition）中定义的种种规定。DTD 实际上是“元标记”的产物，它描述了一个标记语言的语法和词汇

表，也就是定义了文件的整体结构以及文件的语法。简而言之，DTD 规定了一个语法分析器为了解释一个“有效的”XML 文件所需要知道的所有规则的细节。它能实现指定可行标签、元素、属性、每一元素的应用次数以及元素在指定 XML 文件中的排序等功能。DTD 可以提供数据类型强化的限定类型，从另一种意义上来说，也就是可以自行确定一个元素是否含有其他元素、其他数据或者为空。

当打开 DTD 关联 XML 文件时，就是要使 DTD 在该 XML 文件中生效，如果违反了 DTD 规则，那么就会出错。XML 文件必须满足两个正确的标准，第一个就是格式要正确，也就是说要满足基本 XML 语法规则，文档必须能够被读取才说明格式是正确的，否则就存在格式错误问题，也就不能被读取。另一个标准称为“生效”，它是可选的，意思就是说文件应该使 DTD 或 XML 生效，XML 解析器不能读取 XML 文件或者不能纠正 XML 文件的话，也就不能继续下去。

DTD 可以与它所定义的元素保存在同一文档中，也可以通过外部的 URL 连接到该文档。这种外部的 DTD 可以由不同的文档和 Web 共享。DTD 为应用程序、组织和利益组进行信息交换提供了结构规范。下边是一个 DTD 的例子，以给前面的 bookStore.xml 加上 DTD 文件。

bookStore1.xml（使用内部的 DTD）的代码如下所示：

```
<?xml version="1.0" encoding="gb2312" standalone="yes"?>
<!DOCTYPE bookList [
    <!ELEMENT bookList (book)*>
    <!ELEMENT book (name,author,publishDate,price)>
    <!ELEMENT name (#PCDATA)>
    <!ELEMENT author (#PCDATA)>
    <!ELEMENT publishDate (#PCDATA)>
    <!ELEMENT price (#PCDATA)>
    <!ATTLIST book hot (true|false) "false">
]>
<bookList>
    <book>
        <name>Java 编程入门</name>
        <author>张三</author>
        <publishDate>2003-1-5</publishDate>
        <price>35.0</price>
    </book>
    <book hot="true">
        <name>XML 在 Java 中的应用</name>
        <author>李四</author>
        <publishDate>2002-12-16</publishDate>
        <price>92.0</price>
    </book>
</bookList>
```

bookStore2.xml（使用外部的 DTD）的代码如下所示：

```
<?xml version="1.0" encoding="gb2312" standalone="yes"?>
<!DOCTYPE bookList SYSTEM "bookStore.dtd">
<bookList>
    <book>
        <name>Java 编程入门</name>
```

```
        <author>张三</author>
        <publishDate>2003-1-5</publishDate>
        <price>35.0</price>
    </book>
    <book hot="true">
        <name>XML 在 Java 中的应用</name>
        <author>李四</author>
        <publishDate>2002-12-16</publishDate>
        <price>92.0</price>
    </book>
</bookList>
```

bookStore.dtd 的代码如下所示：

```
<?xml encoding="GBK"?>
<!ELEMENT bookList (book) >
<!ELEMENT book (name,author,publishDate,price)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT author (#PCDATA)>
<!ELEMENT publishDate (#PCDATA)>
<!ELEMENT price (#PCDATA)>
<!ATTLIST book hot (true|false) "false">
```

从上边的例子可以看出，DTD 声明的一般过程是：

```
<!ELEMENT elementname (elementtype modifiers)>
```

DTD 元素修改器定义了可用内容及元素的应用次数，其规则如表 7-1 所示，如果你了解 PERL 的正则表达式（现在许多语言都支持正则表达式了，包括 Java），就会觉得该表很容易理解并记住。

表 7-1 DTD 的规则

修饰符	例子	含义
没有修饰符	Element (A)	A 仅可以出现一次
?	Element (A) ?	A 可以不出现或出现一次
*	Element (A) *	A 可以不出现也可出现任意多次
+	Element (A) +	A 可以出现多次但至少出现一次
|	Element (A|B)	或者出现 A 或者出现 B
EMPTY	Element EMPTY	元素不能包含任何数据
#PCDATA	Element (#PCDATA)	可以是任何非 XML 元素的数据
CDATA	Element CDATA	字符数据类型，不包括（<、>、&、"）
ID	Element ID	用于确认文档中的元素，不可相同
IDREF	Element IDREF	引用文档中 ID 类属性的元素
IDREFS	Element IDREFS	引用多个 ID 类属性的元素，用空格分开 ID
ENTITY	Element ENTITY	代表从外部文件中获得的二进制数据
ENTITIES	Element ENTITIES	ENTITY 的复数形势，用空格分开 ENTITY
NMTOKEN	Element NMTOKEN	一个有效的 XML 名称
NOTATION	Element NOTATION	标记符号
ANY	Element ANY	任何数据

注意这里 CDATA 是字符数据类型，但不能包括“<”、“>”、“&”、“"”。如果使用这些符号，就需要使用编码符号，如表 7-2 所示。

表 7-2　DTD 的编码符号

符号	编码符号
<	<
>	>
&	&
“	"

NMTOKEN 类型属性的值被限定为“有效的 XML 名称”，有效的 XML 名称请参照前边小节中的内容。ANY 类型属性的值可以是任何数据类型，如果贪图方便而过多地使用也不是不可以，只是这样的 DTD 就没有什么意义了，一般只在根元素使用或根本不使用。

前边讲述了如何定义一个元素以及它的内容，如何描述父元素与子元素之间错综复杂的关系，下边介绍一下如何定义元素的属性。

前边的例子中包括一行<book hot="true">，hot 在这里称为 book 的属性，可以看到 DTD 中有这么一行<!ATTLIST book hot (true|false) "false">，这就是描述属性的基本用法。

根据 XML 文件是否必须为一个属性提供取值，属性的默认值又可以分为以下 4 类。

- 必须赋值的属性（#REQUIRED）：关键字 REQUIRED 说明 XML 文件中必须为这个属性给出一个属性值，而且不用提供默认值。
- 属性值可有可无的属性（#IMPLIED）：当使用关键字时，语法解释器不再强行要求在 XML 文件中给该属性赋值，而且也无须在 DTD 中为该属性提供默认值。可以说，这是对属性值有无的最低要求，现实中经常用到。
- 固定取值的属性（#FIXED）：还有一种特殊情况，你需要为一个特定的属性提供一个默认值，并且不希望 XML 文件的编写者把你的默认值替代掉。这时候，就应该使用 FIXED 关键字，同时为该属性提供一个默认值。
- 定义默认值的属性：如果不使用上面任何一种关键字的话，该种属性就是属于这种类型。对于这种属性，需要在 DTD 中为它提供一个默认值。而在 XML 文件中可以为该属性给出新的属性值来覆盖事先定义的默认值。也可以不另外给出属性值，后一种情况下它就默认为采用 DTD 中给出的默认值。

7.2.2　Schema 简介

使用 DTD 虽然在指定许可的元素、需要的元素以及给定 XML 文档中如何组织元素等方面给我们以较大的方便，但是一旦你想针对特定元素施加数据类型就会遇到麻烦了。DTD 规范严格地定义了结构，但只支持相对功能较弱的内容类型规范，而对强制性结构化却无计可施。比如名为 Date 的数据，如何规定它必须包含有效值呢？

这就要指望 XML Schema 了，XML Schema 作为建议已经于 2001 年提交给了 W3C，这意味着它最终将成为一般用途的建议标准。假如你对此感兴趣，不妨到 W3C 网站找些官方文档和内容简介之类的材料来看看。注意，其他 Schema 定义也是有的，包括日本的标准 RELAX 和

微软公司的 XDR。可是，XML Schema 是惟一受到 W3C 承认的建议标准。

和 DTD 相比，Schema 有如下优点：

- Schema 使用的是 XML 语法。
- Schema 可以用 XML 解析器来解析。
- Schema 允许全局性元素（在整个 XML 文档中元素用相同方式来使用）和局部性元素（元素在特定的上下文中有不同的含义）。
- Schema 提供丰富的数据类型（如整型、布尔型、日期类型等），而且一个元素中的数据类型可以进行规定，甚至可以根据需要自定义数据类型 XDR。

一个使用 Schema 的 XML 文件 bookStore3.xml 的代码如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<x:bookList xmlns:x="urn:book">
    <book>
        <name>Java 编程入门</name>
        <author>张三</author>
        <publishDate>2003-1-5</publishDate>
        <price>35.0</price>
    </book>
    <book hot="true">
        <name>XML 在 Java 中的应用</name>
        <author>李四</author>
        <publishDate>2002-12-16</publishDate>
        <price>92.0</price>
    </book>
</x:bookList>
```

Schema 文件 book.xsd 必须位于 bookStore3.xml 的相同路径，它的代码如下所示：

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="bookList" type="bookListData"/>
    <xsd:complexType name="bookListData">
        <xsd:sequence>
            <xsd:element name="book" type="bookdata"
 minOccurs="0" maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
    <xsd:complexType name="bookdata">
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element name="author" type="xsd:string"/>
            <xsd:element name="publishDate" type="xsd:date"/>
            <xsd:element name="price" type="xsd:float"/>
        </xsd:sequence>
        <xsd:attribute name="hot" type="xsd:string"/>
    </xsd:complexType>
</xsd:schema>
```

7.2.3 Schema 的文件结构

在讲述元素的定义方法之前，先来讲一下 Schema 文件的结构。其实，Schema 文件和其

他 XML 文件的样子非常相似，也是由一组元素构成的，其根元素是"Schema"。"Schema"元素是 XML Schema 中第一个出现的元素，用于表明该 XML 文档是一个 Schema 文档。相应的，"Schema"的结束标记一般在文档的末尾。这样，一个 Schema 的结构如下：

```
<Schema name="schema-name" xmlns="namespace" >
.....
</Schema>
```

Schema 具有两个属性：name 指定该 Schema 的名称，而 xmlns 则指定该 Schema 包含的名称空间。注意，一个 XML Schema 文档中可以包含多个名称空间。一个 Schema 文件包含以下基本元素：

- ElementType。声明 XML 文件中会出现的元素，ElementType 的语法表达如下：

```
<ElementType    name="元素名"
    content="{ empty | textOnly | eltOnly | mixed }"
    type="元素类型"    order="{ one | seq | many }"
    model="{ open | closed }" >
```

在 ElementType 的几个属性中，name 的含义不言而喻，是所声明元素的名称，它是不可缺少的。content 是 ElementType 的一个重要属性，它指明 ElementType 所声明的元素是否为空、是否包含文本、是否包含子元素，还是既包含文本又包含子元素。type 指定该元素的数据类型，我们会在后面详细讨论。order 指定该元素的子元素的排列顺序规则。最后，model 指定该元素是否可以包含未在本 Schema 中定义的元素和属性，它主要用于其他 Schema 的引入，也就是其他名称空间的引入。open 表明该元素可以包含其他未在 XML Schema 中定义的元素和属性，而 close 表明该元素只能包含在本 XML Schema 中定义过的元素和属性。默认状态下，XML Schema 的 model 取值“open”，也就是说，该元素可以包含其他未在 XML Schema 中定义的元素和属性。但是，这并不意味着任何元素和属性都可在 Schema 中出现，允许出现的前提是这些“异类”元素和属性必须在单独的 XML Schema 中加以定义，并且必须在引用它们的元素中以名称空间形式指定其出处。

- Element。ElementType 只是起到声明元素的作用，至于元素的内容究竟是什么，则要靠它的子元素 element 来说明。element 的语法表达如下：

```
<element
    type="元素类型"
    [minOccurs="{ 0 | 1 }"]
    [maxOccurs="{ 1 | * }"]
>
```

element 实际上是对该 Schema 中 ElementType 声明的引用，而具体引用什么元素类型，就要靠 type 属性指定了。type 属性不可缺少，并且为了保证 type 指定的是已经声明过的元素，要求它的取值必须同某个 ElementType 中的 name 属性严格一致。至于其他两个属性倒是可有可无。minOccurs 指定该元素在其父元素中出现的最小次数，默认值为 1，表明该元素至少出现一次；也可以取值为 0，表明该元素是可选的，可以不出现。maxOccurs 则指定了该元素出现的最大次数，默认值同样为 1，表明该元素至多出现一次；也可取值为“*”，表明该元素在 XML 实例文档中出现次数不受限制。除了数字之外，还可以使用 ZEROORMORE、ONEORMORE、OPTIONAL、REQUIRED 等英文修饰。

- Group。和 DTD 一样，Schema 也有组的概念，它的语法表示如下：

```
<group
```

```
 order="{one | seq | many}"
[minOccurs="{ 0 | 1 }"]
[maxOccurs="{ 1 | * }"]
>
```

和 DTD 的规定相同，组里的内容可以是元素，也可以是另一个子组。属性 order 指定该组中的元素或子组的顺序。seq 是 sequence 的缩写。

- AttributeType。AttributeType 元素也是 Schema 中的重要元素之一，用于定义该 Schema 文档中出现的属性类型。AttributeType 的语法表达如下：

```
<AttributeType
  name="属性名"
  type="属性类型"
  values="枚举值列表"
  default="默认值"
  required="{yes | no}"
>
```

- Attribute。AttributeType 和 attribute 的关系与 ElementType 和 element 的关系相同，AttributeType 只是起到声明属性的作用，而真正指明一个元素具有哪些属性还需依靠 attribute 元素。attribute 的语法表达如下：

```
<attribute type="attribute-type" default="default-value" [required="{yes | no}"] >
```

前边提到过，Schema 提供了比 DTD 丰富得多的数据类型，也就是 type 的取值范围，它包括基本类型和派生类型。其中的基本类型和 DTD 的数据类型基本相同，包括 ID、IDREF、IDREFS、ENTITY、ENTITIES、NMTOKEN、NMTOKENS、NOTATION，代表的意思和 DTD 也基本一样，就不详细介绍了，可以参考前边的 DTD 类型。Schema 的派生类型主要有：

- string：字符串。
- boolean：布尔值。
- number：数值型。
- dataTime：日期时间类型。
- binary：二进制数据块。
- uri（Universal Resource Identifier）：统一资源标识符。
- integer：整数型，由 number 类型派生。
- decimal：小数型，由 number 类型派生。
- real：实数型，由 number 类型派生。
- date：日期，由 dateTime 类型派生。
- time：时间，由 dateTime 类型派生。
- timePeriod：时间段，由 dateTime 类型派生。

除了上边介绍的这些基本类型和派生类型外，Schema 还允许用户自己定义类型。请看下边的例子：

```
<datatype id="phoneNum">
    <basetype id="integer"/>
    <lexicalRepresentation>
        <lexical>99999999</lexical>
        <lexical>999-99999999</lexical>
```

```
        </lexicalRepresentation>
    </datatype>
```

上边定义的是一种新的数据类型 phoneNum，这种数据类型的基本类型是整数型，这种类型的数据可以是 8 个数字，也可以是 3 个数字加上横线，后边再跟着 8 个数字。

7.2.4　名域和 Schema 的结合

XML 在检查 XML 文档合法性的时候，会使用名域机制来引用外部的 Schema。名域也称为名称空间。如果某个元素属于一个名域，那么这个元素必须遵守这个名域所包含的 Schema。如果这个元素的一个属性（属性 A）属于另一个名域（名域 B），那么属性 A 就必须遵守名域 B 中的 Schema。通过下边这个例子可以看得更明白。

```
<Document xmlns="http://www.aaa.com/aaa.xsd"
          xmlns:x="http://www.bbb.com/bbb.xsd"
          xmlns:y="http://www.ccc.com/ccc.xsd">
    <element1>
        <name1/>
        <x:name2 y:someattr="123"/>
    </element1>
</Document>
```

上面代码的开头部分声明了 3 个名域，它们分别是：xmlns="http://www.aaa.com/aaa.xsd"、xmlns:x="http://www.bbb.com/bbb.xsd"、xmlns:y=http://www.ccc.com/ccc.xsd。其中，第一个是文档的默认名域，凡是文档中没有指定名域的元素都属于该名域，例如<element1>元素。第二个和第三个不是默认名域，要使用这两个名域，必须在元素或属性的名称前加上名域。例如<x:name2 y:someattr="123"/>，其中 name2 元素的名域是 http://www.bbb.com/bbb.xsd，而 someattr 属性的名域是 http://www.ccc.com/ccc.xsd。

7.2.5　使用 Java 解析 XML 文件

XML 是一种文本文件，我们完全可以自己去解析 XML 方法，但这毕竟太复杂了，幸运的是已经有人替我们做了这些工作。W3C 提出了两种主要的关于 XML 的应用编程接口（API）：基于树（tree-based）的 API 和基于事件（event-based）的 API。基于树的 API DOM（文档对象模型，Document Object Model），首先汇总一个 XML 文档进内部树结构，然后允许一个应用程序去浏览该树。基于事件的 API SAX（XML 的简单 API，Simple API for XML），则通过把分析事件（如一个元素的开始和结束）直接报告给应用程序的调用返回函数，而通常并不建立一棵内部树。应用程序的实时处理器处理不同的事件。

由于 XML 毕竟是新生事物，新生的东西总是面临着统一标准的问题，这个问题至今还没有很好地解决。世界上很多大公司都纷纷开发出了自己的解决方案，Sun 提供了 Java XML Pack，IBM 提供了 IBM XML4J，而 Apache 则推出了 Xerces 和 Xalan 等，这就留给我们一个难题，到底使用哪一家的产品？回答这个问题也很简单，那就是使用最简单的。

JDOM 是两位著名的 Java 开发人员兼作者 Brett Mclaughlin 和 Jason Hunter 的创作成果。2000 年初在类似于 Apache 协议的许可下，JDOM 作为一个开放源代码项目正式开始研发了。如今它已成长为包含来自广泛的 Java 开发人员的投稿、集中反馈及错误修复的系统，并致力于建立一个完整的基于 Java 平台的解决方案，并且通过 Java 代码来访问、操作、输出 XML 数据。

7.3 XML 的样式表

看了上边的例子，一定觉得比较奇怪，难道这就是 XML 的功能？当然不是，上边的例子只是简单介绍一下怎样使用 Java 语言解析、操作 XML，而根本没有用到 XML 的强大之处。另外从 XML 诞生的那一天起，就有人呼吁用 XML 来替代 HTML。但几年过去了，你会发现 HTML 仍然占据着主导地位，这里面的原因很多。首先是 HTML 的功能已经能满足大部分用户的需要，尽管它的功能受一些限制，但它的通用性、易用性总是促使程序员不忍放弃 HTML。另外就是浏览器的问题，各种浏览器之间的不兼容，致使生成一个通用的协议非常困难。再有就是 XML 自己的问题，大家在使用浏览器浏览网页时可能注意到，尤其是低速网的用户更容易注意到，浏览器在显示网页时不是一下子把网页显示出来，而是一部分一部分地显示。如果你熟悉 TCP/IP 协议就会知道，在数据的传输中，也就是说从服务器向浏览器发送数据时是把数据分成包，一个包一个包地发送。由于 HTML 是不严谨的，浏览器在没有全部收到数据包时就已经在解释显示了。而对于 XML 来说，这就不可能实现了，因为 XML 使用的是严谨的数据结构，只有在浏览器收到所有的数据包，并检测到这个 XML 文件确实是正确的格式后才会开始解释、显示。例如当浏览器收到的数据中包括<html>标志，如果是 HTML 文件，浏览器就不管后边是不是有</html>标志，就已经开始解释了。这也是程序员敢于写不标准的 HTML 的原因，也是造成现在把 HTML 转换成合法的 XML 格式非常困难的罪魁祸首。现在还有一种叫 XHTML 的，它就是主张把现在的不规范的 HTML 转换成符合 XML 标准的文件，如<hr>这种单一的标志是不允许的，必须要转换成<hr/>或<hr></hr>。即便是这样，如果浏览器必须下载完所有的数据后才能解释、显示对于大多数的低速网用户仍是难以忍受的，即便是宽带网也无法保证网络不会阻塞。由于上面的一些关系，XML 要想完全取代 HMTL 看起来还有一段路要走。但是也应该看到，XML 取代 HTML 是一种必然的趋势，上述的问题会逐步地得到解决，只不过比预期的要慢罢了。

7.3.1 使用 CSS 样式表

CSS 是一种样式描述规则，目前 W3C 有两个推荐标准：CSS1 和 CSS2。CSS1 于 1996 年 12 月通过，CSS2 则于 1998 年 5 月通过。CSS2 是在 CSS1 的基础上制定的，基本上涵盖了 CSS1，并在 CSS1 的基础上增加了媒体类型、特性选择符、声音样式等功能，并对 CSS1 原有的一些功能进行了扩充。

样式表是用来定义网页格局的模板，通过使用样式表，就可以像 Microsoft Word 一样来格式化网页中的各种格式块。在 Web 页面中使用样式表有如下好处：

- 定义页面中的标头、页边、缩进、点的大小以及各种背景颜色。
- 只需要修改样式表文件就可以修改整个 Web 网站的设计风格。
- 在同类因子中可以通过样式表设计出不同的样式。

样式表是样式定义的集合。样式的定义由一个标记的名称和定义这个标记的显示方式的属性列表组成。属性包括属性名和属性值，其间使用冒号分开，同时各种不同的属性以分号分开。下边是一个 HTML 文件使用样式表的简单例子，test.htm 的代码如下所示：

```
<html>
```

```
<head>
<style>
h1{
    font-size: 16pt "行楷,宋体,黑体";
    font-weight: bold;
    color: red
}
td{
    background:yellow;
}
</style>
</head>
<body>
<table border=1>
    <tr>
        <td><h1>这是使用 h1 标记，粗体，红字</h1></td>
    </tr>
    <tr>
        <td>默认 td 中的样式</td>
    </tr>
</table>
<body>
</html>
```

使用 IE 浏览这个网页时，显示的结果如图 7-1 所示。

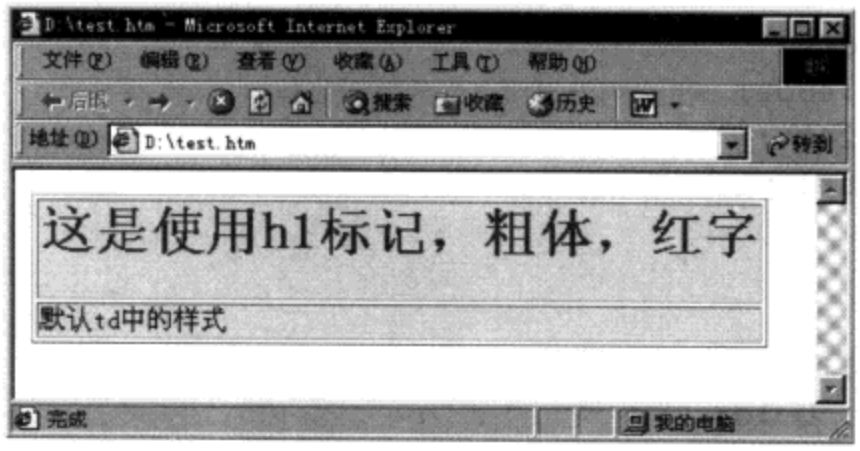


图 7-1　test.htm 页面的显示

CSS 制定之初就是供 HTML 使用的，在 HTML 中使用样式表就不多讲了。下边看一下在 XML 中使用 CSS 样式表。考虑下边的 XML 文件，books.xml 的代码如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<books>
    <book>
        <Title>红楼梦</Title>
        <Author>曹雪芹</Author>
        <Class>古典小说</Class>
    </book>
    <book>
        <Title>射雕英雄传</Title>
        <Author>金庸</Author>
        <Class>武侠小说</Class>
```

```
    </book>
</books>
```

在 IE 中浏览 books.xml 的效果如图 7-2 所示。

图 7-2 books.xml 页面的效果

虽然在页面中可以通过左边的加减符号展开或者折叠标记的显示，但是这种显示效果显然难以让我们满意。幸好可以使用 CSS 样式表来定义各种元素的显示格式，这里我们为它编写了一个样式表文件 style.css，源代码如下所示：

```
book{
    display: block;
    margin-left:40pt;
    margin-top:24pt;
    margin-right:40pt;
}
Title
{
    display: inline;
    font-weight: bold;
    font-size: 16pt;
    width:100pt;
}
Author
{
    display: inline;
    color: red;
    width:50pt;
}
Class
{
    display: inline;
    background-color:yellow;
    width:50pt;
}
```

然后修改原来的文件，在第一行的下边加上下边的语句，以引用这个样式表文件。

```
<?xml-stylesheet type="text/css" href="style.css"?>
```

图 7-3 所示的显示效果虽然还不是很精致，至少还说得过去吧。如果你有美工基础，肯定能做得比这漂亮多了。

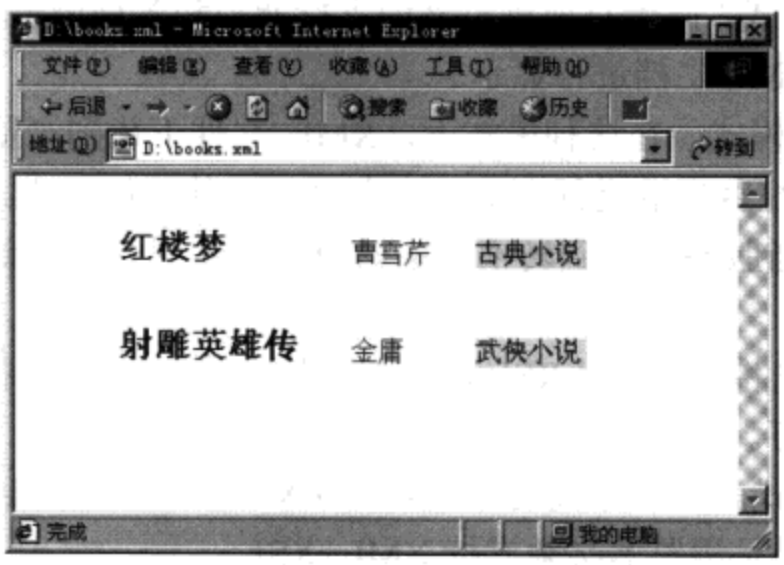


图 7-3　使用样式表文件的 books.xml

7.3.2　使用 XSL 样式表

前边已经讲述了层叠样式表 CSS，本节要介绍的是另一种可扩展样式表语言 XSL（eXtensible Stylesheet Languge），它也是由 W3C 制定的。XSL 样式语言自提出以来就争议颇多，前后经过了几次大的修改。XSL 最近的一个草案于 2000 年 3 月提出，仍然有待进一步修改完善，因此还不能作为正式的设计依据。

CSS 是一种静态的样式描述格式，其本身不遵从 XML 的语法规范。而 XSL 则不同，它是通过 XML 进行定义的。它遵守 XML 的语法规则，是 XML 的一种具体应用，提供了比 CSS 强大得多的功能。也就是说，XSL 本身就是一个 XML 文档，系统可以使用同一个 XML 解释器对 XML 文档及其相关的 XSL 文档进行解释处理。

XSL 由两大部分组成：第一部分描述了如何将一个 XML 文档转换为可浏览或可输出的格式；第二部分则定义了格式对象 FO（fomatted object）。在输出时，首先根据 XML 文档构造源树，然后根据给定的 XSL 将这个源树转换为可以显示的结果树，这个过程称作树转换。最后再按照 FO 解释结果树，产生一个可以在屏幕、纸、语音设备或其他媒体中输出的结果，这个过程称作格式化。

到目前为止，W3C 还未能出台一个得到多方认可的 FO，但是描述树转换的这一部分协议却日趋成熟。已从 XSL 中分离出来，另取名为 XSLT（XSL 转换，XSL Transformations），其正式推荐标准于 1999 年 11 月 16 日问世，现在一般所说的 XSL 大都指的是 XSLT。与 XSLT 一同推出的还有其配套标准 XPath，这个标准用来描述如何识别、选择、匹配 XML 文档中的各个构成元件，其中包括元素、属性、文字内容等。

XSLT 主要的功能就是转换，它将一个没有形式表现的 XML 内容文档作为一个源树，将其转换为一个有样式信息的结果树。在 XSLT 文档中定义了与 XML 文档中各个逻辑成分相匹配的模板，以及相应的匹配转换方式。值得一提的是，尽管制定 XSLT 规范的初衷只是利用它来进行 XML 文档与可格式化对象之间的转换，但它的巨大潜力却表现出它可以很好地描述 XML 文档向任何一个其他格式的文档作转换的方法，例如转换为另一个逻辑结构的 XML 文

档、HTML 文档、XHTML 文档等。

使用 XSL 定义 XML 文档显示方式的基本思想是：通过定义转换模板，将 XML 源文档转换为带样式信息的可浏览文档。最终的可浏览文档可以是 HTML 格式、FO 格式或者其他面向显示方式描述的 XML 格式。限于目前浏览器的支持能力和各种浏览器之间的兼容性，多数情况下还是转换为一个 HTML 文档进行显示。

（1）服务器端转换模式。在这种模式下，XML 文件下载到浏览器前先转换成 HTML，然后再将 HTML 文件送往客户端进行浏览。它有以下两种工作方式。

- 动态方式：即当服务器接到转换请求时再进行实时转换，这种方式无疑对服务器要求较高。
- 批量方式：首先将 XML 用 XSL 转换好一批 HTML 文件，接到请求后调用转换好的 HTML 文件即可。

（2）客户端转换模式。这种方式是将 XML 和 XSL 文件都传送到客户端，然后由浏览器实时转换。它的工作前提是浏览器必须支持 XML+XSL。

7.3.3 使用 XSL 的例子

下边是一个使用 XSL 的简单例子，它用来显示股票代码。Stocks.xml 的代码如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<?xml-stylesheet type="text/xsl" href="stock.xsl"?>
<stocks>
    <stock>
        <name>上海医药</name>
        <code>600849</code>
        <price>11.840</price>
    </stock>
    <stock>
        <name>科大创新</name>
        <code>600551</code>
        <price>18.680</price>
    </stock>
    <stock>
        <name>金山股份</name>
        <code>600396</code>
        <price>15.180</price>
    </stock>
</stocks>
```

上边的文件和前边讲过的 XML 文件没有什么区别，值得注意的只有一条语句：

```
<?xml-stylesheet type="text/xsl" href="stock.xsl"?>
```

它的作用就是调用 XSL 样式表，这和调用 CSS 样式表的方法基本一致。下边就看这个 XSL 样式表 stock.xsl 的代码：

```
<?xml version="1.0" encoding="gb2312"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
<html>
    <body>
        <table border="1" align="center" cellpadding="5">
            <tr bgcolor="#999999">
```

```
                <td>股票名称</td>
                <td>股票代码</td>
                <td>今日价格</td>
            </tr>
            <xsl:for-each select="stocks/stock">
                <tr>
                    <td><xsl:value-of select="name"/></td>
                    <td><xsl:value-of select="code"/></td>
                    <td><xsl:value-of select="price"/></td>
                </tr>
            </xsl:for-each>
        </table>
    </body>
</html>
</xsl:template>
</xsl:stylesheet>
```

为了能看懂上边的代码，这里先介绍几个常用的 XSL 语句，如表 7-3 所示。

表 7-3　常用的 XSL 语句

主要语句	含义
xsl:stylesheet	声明语句
xsl:template	相当于编程中函数的概念
xsl:template match = ""	相当于函数调用，去匹配引号中指定的节点
xsl:for-each select = ""	循环语句，遍历与引号中的属性值相同的节点
xsl:value-of select = ""	赋值语句，取出引号中指定的属性值

使用 IE 浏览器打开 stocks.xml 的效果如图 7-4 所示。

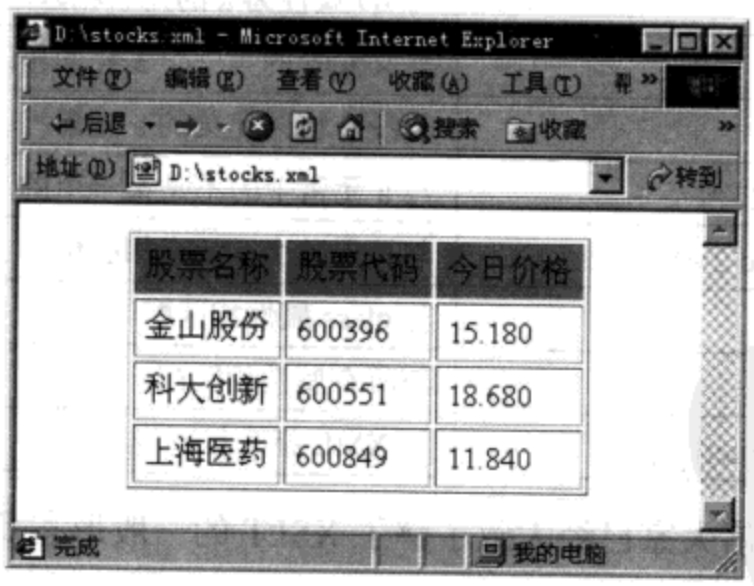


图 7-4　stocks.xml 的效果图

1．XSL 的工作方式

XSL 是表达样式表的语言，每一个样式表描述了呈现一类 XML 源文档的规则，这个呈现的过程包括两步。

（1）由源树建立结果树并构造结果树，就是将模式与模板相结合实现。模式与源树中的元素相匹配，模板被实例化产生部分结果树，结果树和源树是分离的。源树可以被过滤和重

新排序，还可以增加任意结构。

（2）结果树被解释并格式化输出，就是用该 XSL 文档中规定的格式化词汇表实现结果树的构造。正规地来讲，这个词汇表是一个 XML 的名域。词汇表中的每一个元素类型对应一个格式化对象类。一种格式化对象类表达一种特定的格式化表现方式。

样式表包含了一套模板的规则集合。一个模板分为两部分：匹配源树终结点的模式和实例化后组成部分结果树的模板。当一个模板被实例化时，XSL 处理器将执行模板中的每一条指令，并把 XML 元素置换为其产生的结果树片段。元素只有在被执行的指令中选中才会被处理。

2. 模板规则

XSL 在执行模板规则的指定动作之前，必须找到与该模板规则的模式相匹配的 XML 元素。如前边例子中的<xsl:for-each select="stocks/stock">，它的模式就是"stocks/stock"，XSL 处理器只对<stocks>中的<stock>元素执行动作。我们还可以使用正则表达式编写出更复杂的模式，如表 7-4 所示。

表 7-4 模板的正则表达式

举例	含义
select="a"	与 a 元素相匹配
select="a/b"	与 a 元素中的 b 子元素相匹配
select="*"	与任何元素相匹配
select="a|b"	与 a 元素或 b 元素相匹配
select="a//b"	与 a 元素中的 b 孙子元素相匹配
select="/"	与根元素相匹配
select="text()"	与文本节点元素相匹配
select="id(a) "	与 ID 属性为 a 的元素相匹配
select="a[1] "	与父元素的第一个 a 元素相匹配
select="a[last()=1] "	与没有相同兄弟的 a 元素相匹配
select="a/b[position()>1] "	与 a 元素中的不是第一个的 b 元素相匹配
select="a[position() mod 2=0] "	与第偶数个 a 元素相匹配
select="@class"	与 class 属性相匹配
select="@*"	与所有属性相匹配
select="processing-instruction()"	与 XML 指令相匹配

从前面的例子中，相信大家已经大致了解了 XSLT 的一般形态及功能。下边，我们再来综合地论述一下。

3. 文档结构

前面说过，XSLT 文档本身就是一个 XML 文档，因此 XSLT 文档的第一条语句自然就是：

```
<?xml version="1.0" ?>
```

接下来是样式表部分：

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
…
```

```
</xsl:stylesheet>
```

在这里也可以写为：

```
<xsl:transform version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
…
</xsl:transform>
```

xsl:transform 与 xsl:stylesheet 具有相同的含义，都表示元素所包含的内容为样式表。xsl:stylesheet 元素必须包含有"version"属性，用以指示该 XSL 文档遵从哪一个版本的 XSL 标准。另外，xmlns:xsl 指示了 XSL 的名称空间。在 XSLT 标准中，定义了 XSLT 的名称空间为 http://www.w3.org/1999/XSL/Transform。

XSLT 在进行转换时，首先遍历 XML 源文档树，以找到要处理的节点，然后将定义好的模板信息施加到该节点中。

4. 模板及应用

xsl:template 是一个模板元素，通常每个 xsl:template 有一个节点匹配属性，由"match="来指定。在对模板进行匹配时使用"xsl:apply-templates"，选择要匹配的模板，这相当于一个调用的过程。对节点的匹配规则遵照 XPath。

5. 计算节点值

在使用 XSLT 进行转换时，常常需要获取节点值。使用 xsl:value-of 元素可达到这个目的，如下面的这条语句：

```
<xsl:value-of select="code"/>
```

这条语句得到的结果是股票代码的值，select 属性指定要获取的是哪一个节点的节点值。

6. 循环处理

使用 xsl:for-each 可对所选节点依次进行处理，如下面的例子在生成表格处理中，就是利用这个循环将各个学生的信息取出放入表格中的，它的写法是：

```
<xsl:for-each select="student" order-by="name">
 …
</xsl:for-each>
```

7. 排序

对于用 xsl:for-each 或 xsl:apply-templates 匹配的节点，可使用 xsl:sort 将所选节点的内容进行排序，如表 7-5 所示。

表 7-5　使用 xsl:sort 语句进行排序

排序方式	举例	含义
按大小写排序	<xsl:sort case-order="upper-first" select="@id"/>	以 id 为关键字按大写优先排序
	<xsl:sort case-order="lower-first" select="@id"/>	以 id 为关键字按小写优先排序
按字母顺序排序	<xsl:sort order="ascending" select="@id "/>	以 id 为关键字按字母升序排序
	<xsl:sort order="descending" select="@id "/>	以 id 为关键字按字母降序排序

续表

排序方式	举例	含义
按数据类型排序	<xsl:sort data-type="text" select="@id"/>	以 id 为关键字按文本类型排序，如对于一组 id 数据 101、2、44、305 来说，排序结果是 101、2、305、44
	<xsl:sort data-type="number" select="@id"/>	以 id 为关键字按数据类型排序，上面一组数据的排序结果是 2、44、101、305

另外，还有一种指定排序的方法，就是在前面股票行情的例子中所使用的 order-by：

```
<xsl:for-each select="stocks/stock" order-by="code">
```

也可使得输出股票的结果时按照股票代码进行排序。

8. 元素与属性创建

XSLT 是一个动态的样式单，在处理过程中可产生新的元素或元素属性，方法如表 7-6 所示。

表 7-6 元素与属性的创建

内容	元素	举例	转换结果
创建元素	xsl:element	<xsl:element name="TITLE"> 股票行情 </xsl:element>	<TITLE>股票行情</TITLE>
创建属性	xsl: attribute	<h1> <xsl:attribute name="style"> color:red</xsl:attribute> 股票行情 </h1 >	股票行情
创建文本	xsl:text	<xsl:text>股票行情 </xsl:text >	输出文字： 股票行情
创建处理指令	xsl:processing-instruction	<xsl:processing-instruction name="xml-stylesheet"> href="book.css" type="text/css" </xsl:processing-instruction>	<?xml-stylesheet href="book.css" type="text/css"?>
创建注释	xsl:comment	<xsl:comment> 此信息只供参考</xsl:comment>	<!--此信息只供参考-->

9. 输出格式与编码问题

XSLT 是一个转换语言，它的目的是将 XML 源文档转换为另一种格式文档。它的输出结果可以是 HTML 文档，也可以是带 CSS 的 XML 文档，具体的输出格式由 xsl:output 指定。如果要输出为一个 HTML 文档，则可以编写为：

```
<xsl:output method="html"/>
```

同样，要输出一个 XML 文档，则可以编写为：

```
<xsl:output method="xml"/>
```

如果文档中没有出现 xsl:output，将默认输出为 XML 文档。如果在匹配模板时使用了<HTML>标记，则输出为 HTML 文档。输出为 HTML 文档时系统会自动加上下面的语句：

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

此外，还可以利用 xsl:output 指定编码方式，如 UTF-8、UTF-16 和 GB2312 等。例如：

```
<xsl:output method="html" encoding="GB2312"/>
```

它指定了该 XSLT 的输出结果是 HTML 格式以及编码方式为 GB2312。

前边介绍了两种 XML 的样式表：CSS 和 XSL。CSS 和 XSL 均属于样式表的一种，都可以用来设定文档的外观。比较起来，它们主要有以下几个大的不同：

- 用途不同。CSS 最早是针对于 HTML 提出的，后来又将其应用于 XML 之中。它既可以为 HTML 文档中的各个成分设定样式，又可以为 XML 中的成分设定样式。XSL 是专门针对 XML 提出的，它不能处理 HTML 文档。但它有一个 CSS 无法达到的功能，即用一个命令行来将一个 XML 文档转换为另一个文档并存盘。
- 处理结果不同。XSL 采用的是一种转换的思想，它将一种不含显示信息的 XML 文档转换为另一种可以用某种浏览器浏览的文档。转换后的输出码或者存为一个新的文档，或者暂存于内存中，但都不修改源代码。而 CSS 则没有任何转换动作，只是针对结构文档中的各个成分，依照样式规定一一设定外观式样，再由浏览器依据这些式样显示文档，在整个过程中没有任何新码产生。
- 表现能力不同。在 XSL 中定义的 90%的样式规定，实际上在 CSS 中都有定义。但仍然有一些效果是 CSS 无法描述的，必须使用 XSL 才可以。这些功能包括文本的置换，例如将一个美国的时间表示格式转换为一个中国的时间表示格式；根据文本内容决定显示方式，例如将 60 分以上的分数用黑色显示，60 分以下的分数用红色显示；将文档中的成分按照某一个子成分的值进行排序，例如将商品按售价进行排序。此外，还有对于超链接的支持、对于 FRAME 的支持、对于某些语种文字从上到下、行从右到左的排列格式的支持等，这些都是 XSL 所独有的。
- 语法不同。XSL 是根据 XML 的语法进行定义的，实际上它又是 XML 的一种应用。而 CSS 的语法自成体系，且比较简单、易学易用。

7.4　DOM 和 SAX

前边都是在孤立地讲述 XML，并讨论和 XML 相关的协议和工具，教会大家如何直接写一个 XML 文件，如何利用浏览器分析、浏览它。但实际上同 HTML 一样，XML 有时是动态生成的，需要我们编写一段代码一个脚本，作为一个“局外人”间接地去创建、访问和操作一个 XML 文件。还有些时候，我们所开发的应用程序需要能够读懂别人写的 XML 文件，从中提取所需要的信息。在以上这些情况下，我们都需要一个 XML 接口。这个接口应是界面友好的，通过它将应用程序与 XML 文档结合在一起。在这一节里，介绍由 W3C 和 XML_DEV 邮件列表成员分别提出的两个标准应用程序接口：DOM 和 SAX。

实际上，XML 文档就是一个文本文件，因此在需要访问文档中的内容时，必须首先书写一个能够识别 XML 文档信息的文本文件阅读器，也就是通常所说的 XML 语法分析器，由它来解释 XML 文档并提取其中的内容。这就要求每个应用 XML 的人都要自己去处理 XML 的语法细节，显然这是一项非常耗时的工作。更糟糕的是，如果需要在不同的应用程序或开发环境中访问 XML 文档中的数据，这样的分析器代码就要被重写多次。

数据库有标准的 ODBC/JDBC 这样的接口规范，我们编写数据库应用程序的时候只要针对于特定的接口即可，可以不管后台的数据库系统究竟是 ORACLE 还是 SYBASE，是 DB2

还是 SQL Server，这给数据库应用程序的开发带来了很大的便利。同样的道理，在做 XML 的应用开发时，一个统一的 XML 数据接口也是必需的。

W3C 意识到了上述问题的存在，于是制定了一套书写 XML 分析器的标准接口规范，即 DOM。除此之外，XML_DEV 邮件列表中的成员根据应用的需求也自发地定义了一套对 XML 文档进行操作的接口规范，即 SAX。这两种接口规范各有侧重，互有长短，应用都比较广泛。

DOM 的全称是 Document Object Model，也即文档对象模型。在应用程序中，基于 DOM 的 XML 分析器将一个 XML 文档转换成一个对象模型的集合（通常称 DOM 树），应用程序正是通过对这个对象模型的操作，来实现对 XML 文档数据的操作。通过 DOM 接口，应用程序可以在任何时候访问 XML 文档中的任何一部分数据，因此，这种利用 DOM 接口的机制也被称作随机访问机制。

DOM 接口提供了一种通过分层对象模型来访问 XML 文档信息的方式，这些分层对象模型依据 XML 的文档结构形成一棵节点树。无论 XML 文档中所描述的是什么类型的信息，即便是制表数据、项目列表或一个文档，利用 DOM 所生成的模型都是节点树的形式。也就是说，DOM 强制使用树模型来访问 XML 文档中的信息。由于 XML 本质上就是一种分层结构，所以这种描述方法是相当有效的。

SAX 的全称是 Simple APIs for XML，也即 XML 简单应用程序接口。与 DOM 不同，SAX 提供的访问模式是一种顺序模式，这是一种快速读写 XML 数据的方式。当使用 SAX 分析器对 XML 文档进行分析时，会触发一系列事件，并激活相应的事件处理函数。应用程序通过这些事件处理函数实现对 XML 文档的访问，因而 SAX 接口也被称作事件驱动接口。

DOM 树所提供的随机访问方式给应用程序的开发带来了很大的灵活性，它可以任意地控制整个 XML 文档中的内容。然而，由于 DOM 分析器把整个 XML 文档转化成 DOM 树并存放在了内存中，因此，当文档比较大或者结构比较复杂时，对内存的需求就比较高。而且，对于结构复杂的树的遍历也是一项耗时的操作。所以，DOM 分析器对机器性能的要求比较高，实现效率不十分理想。不过，由于 DOM 分析器所采用的树结构的思想与 XML 文档的结构相吻合，同时鉴于随机访问所带来的方便，因此 DOM 分析器还是有很广泛的使用价值的。

SAX 分析器在对 XML 文档进行分析时，触发了一系列的事件。由于事件触发本身是有时序性的，因此 SAX 提供的是一种顺序访问机制，对于已经分析过的部分，不能再倒回去重新处理。SAX 之所以被叫做“简单”应用程序接口，是因为 SAX 分析器只做了一些简单的工作，大部分工作还要由应用程序自己去做。也就是说，SAX 分析器在实现时，只是顺序地检查 XML 文档中的字节流，判断当前字节是 XML 语法中的哪一部分、是否符合 XML 语法，然后再触发相应的事件，而事件处理函数则要由应用程序自己来实现。同 DOM 分析器相比，SAX 分析器缺乏灵活性。然而，由于 SAX 分析器实现简单，对内存要求比较低，因此它的实现效率比较高。对于那些只需要访问 XML 文档中的数据而不对文档进行更改的应用程序来说，SAX 分析器更为合适。

综上所述，无论是 DOM 接口还是 SAX 接口，都有其优缺点。也正因如此，它们将长期并存下去，并在不同的应用程序中发挥不同的作用。

7.4.1 使用 DOM

DOM 的基本对象有 5 个：Document、Node、NodeList、Element 和 Attr。

Document 对象代表了整个 XML 的文档，所有其他的 Node 都以一定的顺序包含在 Document 对象之内，排列成一个树形的结构。程序员可以通过遍历这棵树来得到 XML 文档的所有内容，这也是对 XML 文档操作的起点。我们总是先通过解析 XML 源文件而得到一个 Document 对象，然后再来执行后续的操作。此外，Document 还包含了创建其他节点的方法，比如 createAttribute()用来创建一个 Attr 对象。它所包含的主要的方法如下。

- createAttribute(String)：用给定的属性名创建一个 Attr 对象，并可在其后使用 setAttributeNode 方法来放置在某一个 Element 对象上面。
- createElement(String)：用给定的标签名创建一个 Element 对象，代表 XML 文档中的一个标签，然后就可以在这个 Element 对象上添加属性或进行其他的操作。
- createTextNode(String)：用给定的字符串创建一个 Text 对象，Text 对象代表了标签或者属性中所包含的纯文本字符串。如果在一个标签内没有其他的标签，那么标签内的文本所代表的 Text 对象是这个 Element 对象的惟一子对象。
- getElementsByTagName(String)：返回一个 NodeList 对象，它包含了所有给定标签名字的标签。
- getDocumentElement()：返回一个代表这个 DOM 树的根节点的 Element 对象，也就是代表 XML 文档根元素的那个对象。

Node 对象是 DOM 结构中最为基本的对象，它代表了文档树中的一个抽象的节点。在实际使用的时候，很少会真正用到 Node 这个对象，而是用诸如 Element、Attr、Text 等 Node 对象的子对象来操作文档。Node 对象为这些对象提供了一个抽象的、公共的根。虽然在 Node 对象中定义了对其子节点进行存取的方法，但是有一些 Node 子对象（比如 Text 对象），并不存在子节点，这一点是要注意的。Node 对象所包含的主要方法有：

- appendChild(org.w3c.dom.Node)：为这个节点添加一个子节点，并放在所有子节点的最后。如果这个子节点已经存在，则先把它删掉再添加进去。
- getFirstChild()：如果节点存在子节点，则返回第一个子节点。与此相对等的，还有 getLastChild()方法返回最后一个子节点。
- getNextSibling()：返回在 DOM 树中这个节点的下一个兄弟节点。与此相对等的，还有一个 getPreviousSibling()方法返回其前一个兄弟节点。
- getNodeName()：根据节点的类型返回节点的名称。
- getNodeType()：返回节点的类型。
- getNodeValue()：返回节点的值。
- hasChildNodes()：判断是不是存在子节点。
- hasAttributes()：判断这个节点是否存在属性。
- getOwnerDocument()：返回节点所处的 Document 对象。
- insertBefore(org.w3c.dom.Node new，org.w3c.dom.Node ref)：在给定的一个子对象前再插入一个子对象。
- removeChild(org.w3c.dom.Node)：删除给定的子节点对象。
- replaceChild(org.w3c.dom.Node new，org.w3c.dom.Node old)：用一个新的 Node 对象代替给定的子节点对象。

NodeList 对象，顾名思义，就是代表一个包含了一个或者多个 Node 的列表。可以简单地

把它看成一个 Node 的数组，可以通过以下方法来获得列表中的元素。

- GetLength()：返回列表的长度。
- Item(int)：返回指定位置的 Node 对象。

Element 对象代表的是 XML 文档中的标签元素，它继承于 Node，也是 Node 的最主要的子对象。在标签中可以包含属性，因而 Element 对象中有存取其属性的方法。而任何 Node 中定义的方法，也可以用在 Element 对象上面。

- getElementsByTagName(String)：返回一个 NodeList 对象，它包含在这个标签中其下的子孙节点中具有给定标签名字的标签。
- getTagName()：返回一个代表这个标签名字的字符串。
- getAttribute(String)：返回标签中给定属性名称的属性值。在这里需要注意的是，因为 XML 文档中允许有实体属性出现，而这个方法对这些实体属性并不适用。这时需要用 getAttributeNodes()方法得到一个 Attr 对象来进行进一步的操作。
- getAttributeNode(String)：返回一个代表给定属性名称的 Attr 对象。

Attr 对象代表了某个标签中的属性。Attr 继承于 Node，但是因为 Attr 实际上是包含在 Element 中的，它并不能被看作是 Element 的子对象，因而在 DOM 中 Attr 并不是 DOM 树的一部分，所以 Node 中的 getparentNode()、getpreviousSibling()和 getnextSibling()返回的都将是 null。也就是说，Attr 其实是被看作包含它的 Element 对象的一部分，它并不作为 DOM 树中单独的一个节点出现。这一点在使用的时候要同其他的 Node 子对象相区别。

前边提到的 DOM 的基础类和方法不局限语言和工具，当然也包括 Java 语言。由于本书讲的是 Java 语言，所以就必须在 Java 语言的基础上使用 DOM。

DOM 规范提供了 API 的规范，目前 Sun 公司推出的 jdk1.4 的 Java API 遵循了 DOM level 2 Core 推荐接口的语义说明，提供了相应的 Java 语言的实现。

在 org.w3c.dom 中，jdk1.4 提供了 Document、DocumentType、Node、NodeList、Element、Text 等接口，这些接口均是访问 DOM 文档所必需的。可以利用这些接口创建、遍历、修改 DOM 文档。

在 javax.xml.parsers 中，jdk1.4 提供的 DocumentBuilder 和 DocumentBuilderFactory 组合可以对 XML 文件进行解析，并转换成 DOM 文档。

在 javax.xml.transform.dom 和 javax.xml.transform.stream 中，jdk1.4 提供了 DOMSource 类和 StreamSource 类，可以用来将更新后的 DOM 文档写入生成的 XML 文件中。

使用 XML 文件最常用的功能是浏览 XML 文档内的层次结构和数据信息，以及修改这些数据信息。先看一下下边的例子，这个例子的功能是打印出 XML 文档内的数据，重点就是遍历各个节点的过程。

先看一下目标文件 publication.xml 的代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<publication>
    <book>
        <Title>The Mythical Man-month</Title>
        <Writer>Frederick P. Brooks Jr.</Writer>
        <PublishDate>1975-03-12</PublishDate>
    </book>
```

```
    <book>
        <Title>Think In Java</Title>
        <Writer>Bruce Eckel</Writer>
        <PublishDate>1999-04-01</PublishDate>
    </book>
</publication>
```

useDomPrintElement.java 的代码如下所示：

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
public class useDomPrintElements{
    public static void main(String[] args){
        try{
            //获得一个 XML 文件的解析器
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            //解析 XML 文件生成 DOM 文档的接口类，以便访问 DOM
            DocumentBuilder builder = factory.newDocumentBuilder();
            //Document 接口描述了对应于整个 XML 文件的文档树
            Document document = builder.parse( new File("publication.xml") );
            //获得根元素的子节点列表
            Element element = document.getDocumentElement();
            NodeList nodelist = element.getChildNodes();
            //如果预知根元素的名称，也可以使用下边的方法获取根元素的子节点列表
            //NodeList nodelist = document.getElementsByTagName("publication");
            //用递归方法实现 DOM 文档的遍历
            GetElement(nodelist);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
    public static void GetElement(NodeList nodelist){
        Node cnode;
        String str;
        if(nodelist.getLength() == 0){
            // 该节点没有子节点
            return;
        }
        for(int i=0;i<nodelist.getLength();i++){
            cnode = nodelist.item(i);
            if(cnode.getNodeType() == Node.ELEMENT_NODE){
                //打印节点名称
                System.out.println(cnode.getNodeName());
                //递归调用 GetElement
                GetElement(cnode.getChildNodes());
            }
            else if(cnode.getNodeType() == Node.TEXT_NODE){
                str = cnode.getNodeValue().trim();
                // 当该对象不为空时显示信息
                if(str.length()>0)
                    System.out.println("       "+str);
```

```
            }
        }
    }
}
```

下面再看一下如何使用 DOM 来修改 XML 文档，这个例子的功能是在 publication 中添加一本新书。修改原有数据的实现和上边的例子差不多，也是要先遍历到要修改的数据，修改数据的办法可以参照下边的文件，useDomEditElement.java 的代码如下所示：

```
import java.io.*;
import javax.xml.parsers.*;
import org.w3c.dom.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

public class useDomEditElements{
    public static void main(String[] args){
        Text textMsg;
        try{
            //获得一个 XML 文件的解析器
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            //解析 XML 文件生成 DOM 文档的接口类，以便访问 DOM
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document document = builder.parse( new File("publication.xml") );
            // 获得根对象
            Element root = document.getDocumentElement();
            // 在 DOM 文档中增加一个 Element 节点
            Element book = document.createElement("book");
            //在 DOM 文档中增加一个 Element 节点
            Element title = document.createElement("Title");
            //在 DOM 文档中增加一个 Text 节点
            textMsg=document.createTextNode("Applied Cryptography");
            title.appendChild(textMsg);
            //把 title 节点转化为 book 的子节点
            book.appendChild(title);
            Element author = document.createElement("Author");
            textMsg=document.createTextNode("Bruce Schneier");
            author.appendChild(textMsg);
            //把 author 节点转化为 book 的子节点
            book.appendChild(author);
            Element date = document.createElement("publishDate");
            textMsg=document.createTextNode("1994-02-23");
            date.appendChild(textMsg);
            //把 date 节点转化为 book 的子节点
            book.appendChild(date);
            //把 book 节点转化为 root 的子节点
            root.appendChild(book);

            //类 TransformerFactory 实现获得将 DOM 文档转化为 XML 文件的转换器
            TransformerFactory tfactory = TransformerFactory.newInstance();
```

```
            //类 Transformer 实现转化 API
            Transformer transformer = tfactory.newTransformer();
            // 将 DOM 对象转化为 DOMSource 类对象，该对象表现为转化
            //成别的表达形式的信息容器
            DOMSource source = new DOMSource(document);
            // 获得一个 StreamResult 类对象，该对象是 DOM 文档转化成的其他形式的
            //文档的容器
            StreamResult result = new StreamResult(new File("publication.xml"));
            // 调用 API，将 DOM 文档转化成 XML 文件
            transformer.transform(source,result);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

创建节点的过程有些千篇一律，但需要注意的是，对 Element 中所包含的 text（在 DOM 中，这些 text 也是代表了一个 Node 的，必须为它们创建相应的 node），不能直接用 Element 对象的 setNodeValue()方法来设置这些 text 的内容，而需要用创建的 Text 对象的 setNodeValue() 方法来设置文本，这样才能够把创建的 Element 及其文本内容添加到 DOM 树中。

TransformerFactory 类也同样应用了工厂模式，使得具体的代码同具体的转换器无关，实现的方法和 DocumentBuilderFactory 相同。Transformer 类的 transfrom 方法接受两个参数，即一个数据源 Source 和一个输出目标 Result。这里分别使用的是 DOMSource 和 StreamResult，这样就能够把 DOM 的内容输出到一个输出流中。当这个输出流是一个文件的时候，DOM 的内容就被写入到文件中去了。

7.4.2　使用 SAX

SAX 并不是由 W3C 官方提出的标准，可以说是“民间”的事实标准。实际上，它是一种社区性质的讨论产物。虽然如此，在 XML 中对 SAX 的应用丝毫不比 DOM 少，几乎所有的 XML 解析器都会支持它。

与 DOM 比较，SAX 是一种轻量型的方法。通过前边的方法已经看到了，在处理 DOM 的时候，我们需要读入整个的 XML 文档，然后在内存中创建 DOM 树，以生成 DOM 树上的每个 Node 对象。当文档比较小的时候，这不会造成什么问题，但是一旦文档大起来，处理 DOM 就会变得相当费时费力。特别是其对于内存的需求，也将是成倍的增长，以致于在某些应用中使用 DOM 是一件很不划算的事。这时候，一个较好的替代解决方法就是 SAX。

SAX 在概念上与 DOM 完全不同。首先，不同于 DOM 的文档驱动，它是事件驱动的，也就是说，它并不需要读入整个文档，而文档的读入过程也就是 SAX 的解析过程。所谓事件驱动，是指一种基于回调（callback）机制的程序运行方法。

在 XMLReader 接受 XML 文档时，在读入 XML 文档的过程中就进行解析，也就是说读入文档的过程和解析的过程是同时进行的，这和 DOM 区别很大。解析开始之前，需要向 XMLReader 注册一个 ContentHandler，也就是相当于一个事件监听器。在 ContentHandler 中定义了很多方法，比如 startDocument()，它定制了当在解析过程中，遇到文档开始时应该处理的事情。当 XMLReader 读到合适的内容，就会抛出相应的事件，并把这个事件的处理权代理给

ContentHandler，并调用其相应的方法进行响应。

ContentHandler 实际上是一个接口，当处理特定的 XML 文件的时候，就需要为其创建一个实现了 ContentHandler 的类来处理特定的事件。可以说，这个实际上就是 SAX 处理 XML 文件的核心。下面看看定义在其中的一些方法：

```
void characters(char[] ch, int start, int length);
```

这个方法用来处理在 XML 文件中读取字符串，它的参数是一个字符数组以及读取的字符串在这个数组中的起始位置和长度。可以很容易地使用 String 类的一个构造方法来获得这个字符串的 String 类：

```
String charEncontered=new String(ch,start,length);
void startDocument();
```

当遇到文档的开头的时候将调用这个方法，可以在其中做一些预处理的工作。

```
void endDocument();
```

和上面的方法相对应，当文档结束的时候将调用这个方法，可以在其中做一些善后的工作。

```
void startElement(String namespaceURI, String localName, String qName,
Attributes atts);
```

当读到一个开始标签的时候，会触发这个方法。在 SAX1.0 版本中并不支持名域，而在新的 2.0 版本中提供了对名域的支持，这里参数中的 namespaceURI 就是名域，localName 是标签名，qName 是标签的修饰前缀。当没有使用名域的时候，这两个参数都为 null。而 atts 是这个标签所包含的属性列表，通过 atts 就可以得到所有的属性名和相应的值。要注意的是，SAX 中一个重要的特点就是它的流式处理，在遇到一个标签的时候，它并不会记录以前所碰到的标签。也就是说，在 startElement()方法中，所有你知道的信息，就是标签的名字和属性。至于标签的嵌套结构、上层标签的名字以及是否有子元素等其他与结构相关的信息，都是不得而知的，都需要程序来完成。这使得 SAX 在编程处理上没有 DOM 来得那么方便。

```
void endElement(jString namespaceURI,String localName, String qName)
```

这个方法和上面的方法相对应，在遇到结束标签的时候将调用这个方法。

因为 ContentHandler 是一个接口，在使用的时候可能会有些不方便，因而 SAX 还为其制定了一个 Helper 类 DefaultHandler。它实现了这个接口，但是其所有的方法体都为空。在实现的时候，只需要继承这个类，然后重载相应的方法即可。

下边看一个使用 SAX 遍历 XML 文档的例子，这个例子相对使用 DOM 来讲有些复杂了，注意体会前边介绍的几个方法。UseSAXParseXML.java 的代码如下所示：

```
import java.util.*;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import org.xml.sax.Attributes;
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.SAXException;

public class useSAXParseXML{

    public static void main(String[] args) throws Exception {

        String filename="../publication.xml";
        //将我们的解析器对象化
        ConfigParser handler = new ConfigParser();
```

```
            //获取 SAX 工厂对象
            SAXParserFactory factory = SAXParserFactory.newInstance();
            factory.setNamespaceAware(false);
            factory.setValidating(false);
            //获取 SAX 解析
            SAXParser parser = factory.newSAXParser();
            try{
                //将解析器和解析对象联系起来并开始解析
                parser.parse(filename, handler);
            }catch(Exception e){
                e.printStackTrace();
            }
        }
    }
    class ConfigParser extends DefaultHandler{
        private String temp;
        private Stack element=new Stack();
        private StringBuffer currentValue = new StringBuffer();

        //定义开始解析元素的方法，这里是将<xxx>中的名称 xxx 提取出来
        public void startElement(String uri, String localName, String qName,
Attributes attributes) throws SAXException {
            //清空 currentValue 的值
            currentValue.delete(0, currentValue.length());
            element.push(qName);
        }
        //在遇到</xxx>结束后，将之前的名称和值一一打印出来
        public void endElement(String uri, String localName, String qName) throws
SAXException {
            temp=(String)element.pop();
            if(temp.equals("Title"))
                System.out.println("Title:"+currentValue);
            else if(temp.equals("Writer"))
                System.out.println("Writer:"+currentValue);
            else if(temp.equals("PublishDate"))
                System.out.println("PublishDate:"+currentValue);
        }
        //这里是将<xxx></xxx>之间的值加入到 currentValue
        public void characters(char[] ch, int start, int length) throws SAXException {
            currentValue.append(ch, start, length);
        }
    }
```

7.5　在 JSP 中使用 XML

7.5.1　实例说明

这个例子的作用是使用 XML 来保存数据，你不禁要问：保存数据用数据库不就可以了

吗？当然，把数据都保存在数据库里是非常好的办法，但是并一定总是适用。比如数据量很小的时候，我们还要去维护一个数据库就显得有点可笑了，尤其是现在的大部分虚拟主机上使用数据库都是要另外付费的。在没有 XML 以前，我们的选择只有读写文件。读写文件最大的缺点就是对格式的把握，我们必须花费很多的时间去考虑怎样定义一个合理的保存格式，哪怕是一个空格也要考虑到，而且在读这个文件时我们也没有办法保证这个文件的格式是可以被正确读出的。现在 XML 可以轻松解决这个问题了。

下边是一个保存书籍信息的例子，把书籍的详细信息保存为 XML 文件，再用 JSP 去读取这个文件，把它翻译成 HTML 并显示到页面上，而且还可以通过 Web 方式添加新的书籍到 XML 文档里。通常编写程序时都要涉及到配置文件的问题，如果你还在使用类似 ini 文件的方法，可能就有人笑你已经过时了。使用 XML 会安全得多，下边这个例子如果转为配置文件的读取和修改也是很容易的。

7.5.2 编程思路

编程思路很简单，就是先创建一个 XML 文档，用它保存书籍的信息，然后用 JSP 解析这个 XML 文档。在这里，还使用前边使用过的 XML 文件。

前边已经讲过，Java 解析 XML 文档的方法很多，标准的用法当然是使用 DOM 或 SAX，但是这两种办法都有各自的缺点。还有其他公司开发的方法，比如 Apache 小组的 Xerces 等。比较各种方法，最为简单和直观的是 Jdom。

7.5.3 代码分析

booklist.jsp 的代码如下所示：

```
<%@ page language="java"
import="java.util.*,org.jdom.*,org.jdom.output.*,org.jdom.input.*,java.io.
*" %>
<%@ page contentType="text/HTML;charset=GB2312"%>
<%
//定义 XML 文档的保存位置
String filepath="D:\\book.xml";
Element root,book;
List books;
//使用 StringBuffer 类保存书的 HTML 格式，显得代码清晰，易于修改
StringBuffer bookInfo=new StringBuffer();
try{
    //实例化一个 SAXBuilder 类，来操作 XML 文档
    SAXBuilder sb = new SAXBuilder();
    //从文件构造一个 Document，因为 XML 文件中已经指定了编码，所以这里不必了
    Document doc = sb.build(new FileInputStream(filepath));
    root = doc.getRootElement();          //得到根元素
    books = root.getChildren();           //得到根元素所有子元素的集合
    for(int i=0;i<books.size();i++){
        book=(Element)books.get(i);       //得到一本书的信息
        bookInfo.append("<tr>");
        bookInfo.append("<td>"+book.getChildText("name")+"</td>");
        bookInfo.append("<td>"+book.getChildText("author")+"</td>");
```

```
            bookInfo.append("<td>"+book.getChildText("publishDate")+"</td>");
            bookInfo.append("<td>"+book.getChildText("price")+"</td>");
            bookInfo.append("</tr>");
        }
    }catch(Exception e){
        out.println(e);
    }
    %>
    <html>
    <head>
    </head>
    <body>
    <br>
    <br>
    <table align=center width=480 cellpadding=5 border=1>
        <tr align=center>
            <td colspan=4><b>图书列表</b></td>
        </tr>
        <tr align=center bgcolor=#cccccc>
            <td>书名</td>
            <td>作者</td>
            <td>出版日期</td>
            <td>价格</td>
        </tr>
    <%=bookInfo%>
    </table>
    </body>
    </html>
```

addbook.jsp 的代码如下所示：

```
    <%@pagelanguage="java"import="java.util.*,org.jdom.*,org.jdom.output.*,org
.jdom.input.*,java.io.*"  %>
    <%@ page contentType="text/HTML;charset=GB2312"%>
    <%
    String msg="";  //用一个 msg 变量保存提示信息，这样做的好处很多
    String filepath="D:\\book.xml";          //XML 文档的保存位置
    Element root,book;
    List books;

    try{
        //接收上传的参数
        String bookname=request.getParameter("bookname");
        String author=request.getParameter("author");
        String year=request.getParameter("year");
        String month=request.getParameter("month");
        String day=request.getParameter("day");
        String price=request.getParameter("price");
        String publishDate=year+"-"+month+"-"+day;
        //把得到的书名、作者转换为汉字编码
        bookname=new String(bookname.getBytes("8859_1"),"gb2312");
        author=new String(author.getBytes("8859_1"),"gb2312");
```

```
//实例化一个 SAXBuilder 类，来操作 XML 文档
SAXBuilder sb = new SAXBuilder();
//从文件构造一个 Document，因为 XML 文件中已经指定了编码，所以这里不必了
Document doc = sb.build(new FileInputStream(filepath));
root = doc.getRootElement();          //得到根元素
books = root.getChildren();           //得到根元素所有子元素的集合
Element newbook=new Element("book");
//加入一本新书
Element el=new Element("name");
el.setText(bookname);
newbook.addContent(el);
el=new Element("author");
el.setText(author);
newbook.addContent(el);
el=new Element("publishDate");
el.setText(publishDate);
newbook.addContent(el);
el=new Element("price");
el.setText(price);
newbook.addContent(el);
books.add(newbook);

//把修改的信息记入 XML 文档
String indent = "    ";       //定义生成的 XML 文档的格式
boolean newLines = true;
XMLOutputter outp = new XMLOutputter(indent,newLines,"GBK");
        outp.output(doc, new FileOutputStream(filepath));

        msg="已经成功的添加了\""+bookname+"\"";
    }catch(NullPointerException npe){
    /*如果用户第一次访问这个文件，没有数据上传，则会抛出 NullPointerException。
*这样做有些投机，实际应用中不建议这样用，可以使用
* if(request.getParameter("Submit")!=null){}进行判断有无数据提交*/
    }catch(Exception e){
        out.println(e);
    }
    %>
    <html>
    <head>
    </head>
    <body>
    <form name="form1" method="post" action="addbook.jsp">
      <table width="75%" border="1" align="center">
        <tr align="center">
          <td colspan="2"><b>添加新书</b></td>
        </tr>
    <%if(!msg.equals("")){%>
        <tr align="center">
          <td colspan="2"><font color=red><%=msg%></font></td>
        </tr>
```

```
<%}%>
    <tr>
      <td align="right">书名：</td>
      <td>
        <input type="text" name="bookname">
      </td>
    </tr>
    <tr>
      <td align="right">作者：</td>
      <td>
        <input type="text" name="author">
      </td>
    </tr>
    <tr>
      <td align="right">出版日期：</td>
      <td>
        <select name="year">
<%  Calendar cal = Calendar.getInstance();
    cal.setTime(new Date());
    for(int i=cal.get(Calendar.YEAR);i>1980;i--)
        out.println("<option value="+i+">"+i+"</option>");
%>
        </select> 年
        <select name="month">
<%for(int i=1;i<=12;i++)
    out.println("<option value="+i+">"+i+"</option>");
%>
        </select>月
        <select name="day">
<%for(int i=1;i<=31;i++)
    out.println("<option value="+i+">"+i+"</option>");
%>
        </select>日</td>
    </tr>
    <tr>
      <td align="right">价格</td>
      <td>
        <input type="text" name="price" size="5" maxlength="6">
        元</td>
    </tr>
    <tr>
      <td colspan="2" align="center">
        <input type="submit" name="Submit" value="提交">
      </td>
    </tr>
  </table>
</form>
</body>
</html>
```

7.5.4 运行结果

第一次运行 booklist.jsp 的结果如图 7-5 所示，它显示出两本书的信息。

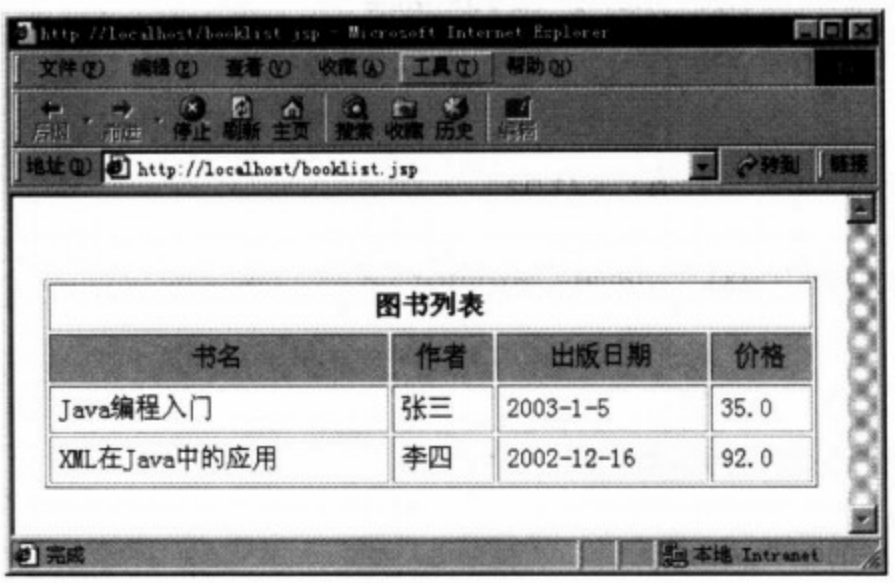


图 7-5 booklist.jsp 的效果图

转到 addbook.jsp 页面，添加一本新书，如图 7-6 所示。然后单击“提交”按钮。在提交数据后，再回到 booklist.jsp，会发现新添加的书已经加入到书籍列表中了，如图 7-7 所示。

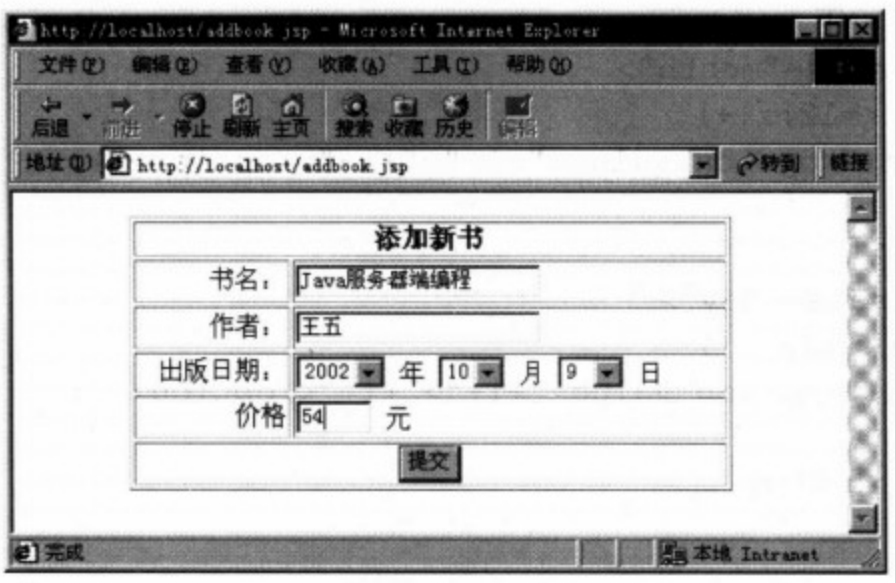


图 7-6 addbook.jsp 的效果图

图 7-7 加入新书籍后的 booklist.jsp 页面

7.6　小结

这一章简要介绍了 XML 以及 XML 的使用，并介绍如何使用 Java 语言解析 XML 文档。XML 的强大功能不是通过这么几节内容就能讲述清楚的，看完了这一章你能对 XML 有个清晰的概念就达到目的了。

同时，由于 XML 相关的技术和标准在不断发展和更新，甚至可以说是日新月异，因此书中的许多内容也许很快就会过时。我们既希望这一天来得快些，又希望这一天来得慢些。不过没关系，万变不离其宗，至少 XML 的核心、XML 的精髓不会变，因为它才是 XML 制胜的武器。

第 8 章　分布式对象概述

8.1　分布式计算介绍

随着近十几年来计算机硬件及网络技术，特别是 Internet 的飞速发展，基于分布式计算及组件技术的软件开发在企业应用开发中逐渐占据了主导地位。

8.1.1　分布式计算的概念

简单地说，分布式计算就是多个软件互相共享信息以完成统一的计算功能。分散的计算资源通过分布式计算可以形成强大的计算能力。这些软件既可以在同一台机器上运行，也可以在通过网络连起来的几台不同的机器上运行。

看一下下面的简单算式在分布式系统中的实现过程，如图 8-1 所示。

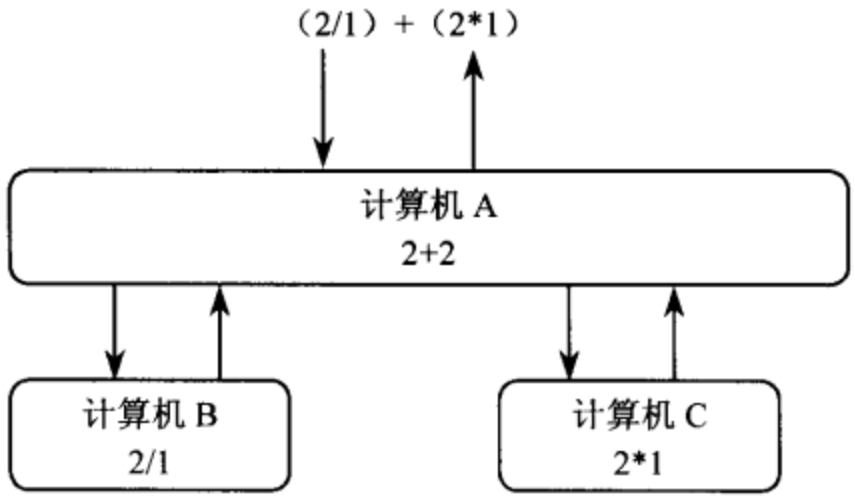


图 8-1　分布式计算的实现过程

假设有三台计算机，A 负责加法，B 负责除法，C 负责乘法。

计算机 A 接到上述算式，首先将算式的除法部分和乘法部分转移到计算机 B 和计算机 C 进行除法和乘法计算，然后将两机器计算的结果进行相加并返回相应的结果。

有人也许会问，为什么这么简单的计算不在一台机器上运行呢？上面的例子只是一种最简单的情况，而企业计算不仅复杂得多，而且运算量也大得惊人，这时单靠计算机硬件速度的提升已经很难满足快速增长的企业计算的需求。如今的企业计算往往需要几台计算机共同协作才能实现某一功能。

从图 8-1 中可以看到，如果 A、B、C 三台计算机之间的网络速度快的可以忽略不计，而且调度管理的开销也不太大，通过多台 PC 机的互联，完全可以达到中型计算机以上的计算能力，这时企业也许才会真正体会到 SUN MICROSYSTEMS 公司的“网络就是计算机”这句名言的魅力。

8.1.2　构成分布式计算的一些基本要素

下面介绍构成分布式计算的一些基本要素。

- 网络通信机制。所有分布式计算环境的基础是计算机之间的通信。虽然这个过程是最基本的过程，但也是必需的，它主要是指网络七层结构中的传输层。
- 同步和异步传输。同步模式的操作就是发送者必须接收到接收者的反馈后才能继续往下工作；而不需要接收者反馈信息的工作模式，或者至少不需要接收者立即反馈的，就叫做异步模式。这两种模式的区别通常决定了某种协议是否适合某一特定任务。比如：用手机给朋友打电话就是同步方式，发短信息就是异步方式。
- 远程过程调用（RPC）。支持 RPC 的概念十分简单，使进程产生的调用看起来像是一个普通的调用，而真正的执行是在其他进程中——也许是一个远程系统中的进程。各种 RPC 执行协议都朝着一个共同的目标在发展，那就是用隐藏执行细节来简化进程间通信的复杂性。
- 网络 API。通信功能的核心部分通常是由操作系统和网络相关的 API 提供。这两种程序调用大量的通信函数来完成实际的系统间数据的传输及接收。总的来说，这些底层组件为底层的通信模块提供了一定层次的抽象，同时也将更高层次的地址标识和数据转换等功能留给高一层的服务模块。
- 通信的角色（客户端，服务器端，对等端）。一个可以称作“服务器端”的线程，通常的任务是打开通信信道，并等待其他线程来与其联系；而主动去联系“服务器端”线程来开始进行通信的线程，一般来讲就是“客户端”；至于“对等端”，它既可以充当客户程序，也可以充当服务程序。
- 系统调度和负载均衡。负载均衡是分布式计算中的一个重要内容，它的主要目标在于均衡所有结点上的负载，以使得所有结点上的负载基本相等，这种相等并非简单的任务数目相等，而是依据这些异构结点的性能分派的加权相等。
- 名称的查找机制。如上例中的计算机 A 要调用计算机 B 上的运算，首先要在网络找到计算机 B 的名字，同时还要保证系统中不能有另一台计算机与计算机重名。
- 安全机制。对于分布式计算，由于可能有大量数据在网络上传播，建立一种安全机制来保证数据在传播中的安全也是十分重要的。
- 事务管理。事务处理是解决分布式环境下局部失败、并发访问等问题的重要技术。

8.2　分布式对象

20 世纪 90 年代以来，计算技术逐步进入以网络为中心的新时期，用户迫切希望在网络上建立更为丰富的分布式客户/服务器应用；不仅实现数据共享，而且支持知识共享和各类计算资源的共享，并能实现包括整个企业在内的各个层次的协同工作。

为适应上述要求，分布对象技术成为分布式计算环境发展的主流方向。其技术特点为：

（1）主要针对异构环境下的应用互操作问题。

（2）系统核心的对象管理将客户/服务器模型与面向对象技术结合在一起。

（3）提供面向对象的 API。

（4）已经成为建立集成框架和软件部件标准的核心技术。

应该说，分布对象技术是伴随网络而发展起来的一种面向对象的技术。以前的计算机系统多是单机系统，多个用户是通过联机终端来访问的，没有网络的概念。网络出现后，产生了 Client/Server 的计算服务模式，多个客户端可以共享数据库服务器和打印服务器等。随着网络的更进一步发展，许多软件需要在不同厂家的网络产品、硬件平台、网络协议异构环境下运行，应用的规模也从局域网发展到广域网。在这种情况下，Client/Server 模式的局限性也就暴露出来了，于是中间件应运而生。中间件是位于操作系统和应用软件之间的通用服务，它的主要作用是屏蔽网络硬件平台的差异性和操作系统与网络协议的异构性，使应用软件能够比较平滑地运行于不同平台上。同时中间件在负载平衡、连接管理和调度方面起了很大的作用，使企业级应用的性能得到大幅提升，满足了关键业务的需求。但是在这个阶段，客户端是请求服务的，服务器端是提供服务的，它们的关系是不对称的。随着面向对象技术的进一步发展，出现了分布式对象技术。可以这么说，分布式对象技术是随着网络和面向对象技术的发展而不断完善起来的。20 世纪 90 年代初 CORBA 1.0 标准的颁布，揭开了分布式对象计算的序幕。

分布式对象计算中，通常参与计算的计算体（分布式对象）是对称的。分布式对象往往又被称为组件（Component），组件是一些独立的代码的封装体，在分布式计算的环境下可以是一个简单的对象，但大多数情况下是一组相关的对象复合体，提供一定的服务。分布环境下，组件是一些灵敏的软件模块，它们可以位置透明、语言独立和平台独立地互相发送消息，实现请求服务。

目前主要的分布式互操作标准有 OMG 组织的 CORBA 标准、Sun 公司的 Java RMI 标准和 Microsoft 的 OLE/COM/DCOM 标准。

8.2.1 CORBA 体系结构

CORBA 的体系结构如图 8-2 所示。在该体系结构中主要描述了以下内容：

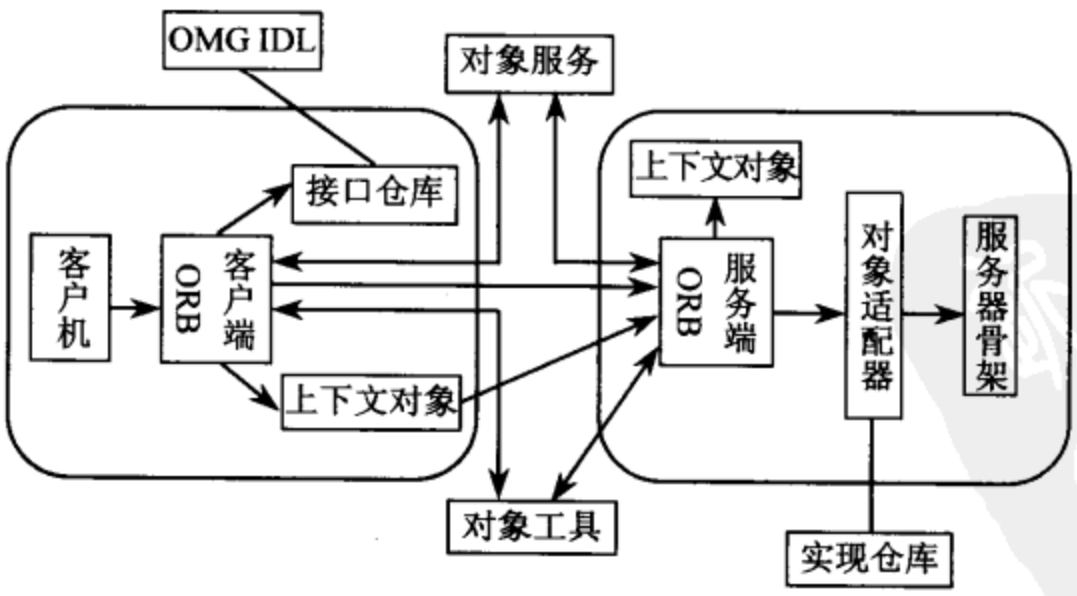


图 8-2 CORBA 体系结构

1. 接口与 IDL 语言

接口是 CORBA 系统中一个非常重要的概念，因为它代表了对象的服务能力，为客户提供

了操作对象的惟一方法。简单地说，接口就是一组相关函数的集合，接口中每一个函数都给出了详细的说明，包括函数名、参数个数、参数类型、返回类型以及可能抛出的异常。必须指出的是，接口只定义了函数的原型，并没有给出具体的实现，这就留给开发者足够的灵活性来提供他们自己的函数实现。接口是通过一种中性的接口描述语言（IDL，Interface Description Language）来定义的。虽然 IDL 语言只提供了被 ORB 操作的对象的概念框架，但是 ORB 在运行时并不需要 IDL 的源代码。只要桩程序或运行状态下接口库中的等价信息是可用的，ORB 就能通过特定的方式完成其功能。

2. 桩和骨架

桩（Stub）可以看成是实际对象在客户进程中的映像，其中的接口必须是预先定义好的，因此它为客户提供了一种静态的调用方式。与桩相对应的骨架（Skeleton，也有人翻译为框架）为服务器提供了一种静态的实现方式。IDL 编译器翻译描述对象接口的 IDL 文件，生成对应具体编程语言的 IDL 桩和 IDL 骨架。桩负责将客户请求进行编码，发送到对象实现端，并对收到的结果进行解释，然后把结果或异常返回给客户。与此相反，骨架对客户的请求进行解码，定位目标对象和请求执行的对象方法，激活该方法，然后把结果或异常信息编码返回给客户端。

3. 动态调用接口和动态骨架接口

与桩和骨架不同，动态调用接口（DII，Dynamic Invocation Interface）和动态骨架接口（DSI，Dynamic Skeleton Interface）提供了动态调用的方式和动态实现的方式，它们使得用户可以在事先不知道对象接口信息的情况下通过查询接口库或采取其他手段动态地获得对象接口信息，然后使用 ORB 核心接口中的 DII 动态调用方法构造客户请求并发送到对象实现。在对象实现方可以使用 DSI 的动态分发机制处理客户方的请求。与静态方式相比，动态方式提供了很大的灵活性，但是它的工作效率没有静态方式高。另外，实现系统的接口大多可以预先确定，所以大部分情况下都是采用静态方式。

客户方与服务器方所采用的方式并不一定要一一对应，也就是说，客户方支持静态和动态两种调用方式，服务器方也支持静态和动态两种实现方式，经过组合得到的 4 种方式都可能出现，例如，客户方使用静态调用方式，而服务器方使用动态构架接口是允许的，反之亦然。

4. ORB 核心和对象适配器

与 UNIX 的实现机制相类似，CORBA 规范将那些相对固定的、单一的功能提取出来交由 ORB 内核实现，以保证它的可靠性、高效性与可重用性。作为整个 CORBA 系统的基础，ORB 内核通过屏蔽诸如服务器位置、实现方式、通信协议等具体细节，为客户方与服务器方之间的通信提供了透明的传输机制。当前的应用系统存在着各种不同的对象实现方式，例如可执行程序、面向对象的数据库等，他们有不同的定位对象、解释对象、激活对象的方法。因此，如果将现存的所有对象实现的解决方案都包含在 ORB 内核中，它必将非常庞大、冗余并且难以移植。为了确保 ORB 内核的高效与可移植性，一个称为对象适配器（OMA）的中间层加入到 ORB 与对象实现之间。定位目标对象的任务从 ORB 内核中分离出来，交给对象适配器来完成。

8.2.2　COM/DCOM

COM（Common Object Model）有时被称为公共对象模型，微软官方则称之为组件对象模

型（Component Object Model）。DCOM 用于分布式计算，是微软开发设计的，作为对 COM 的一个扩展。

COM/DCOM 的前身 OLE（Object Linking and Embedding，对象链接和嵌入）是用于在微软的 WIN 3.1 操作系统中链接文档。开发 COM 是为了在一个单一的地址空间中，动态地集成组件。COM 为在一个单一的应用程序中复杂客户二元组件的动态使用提供支持。组件交互是基于 OLE2 接口和协议的。虽然 COM 使用 OLE2 接口和协议，但我们也必须知道 COM 不是 OLE。DCOM 是 COM 的扩展，支持基于网络的交互，允许通过网络进行进程处理。

COM 允许客户调用服务，服务是由 COM 兼容的组件通过定义一个二元兼容规范和实现过程来提供的。COM 兼容组件（COM 对象）提供了一系列的界面，允许客户通过这些界面来调用相关的对象。

COM 定义了客户和对象之间的二元结构，并且作为用不同程序语言书写的组件之间的相互操作的基础，只要该语言的编译器支持微软的二元结构。

COM 对象可以具有复杂的接口，但是每一个类必须具有它自己惟一的类标识符（CLSID），并且它的接口必须具有全球惟一的标识符（GUID），以避免名字冲突。对象和界面是通过使用微软的 IDL（接口定义语言）来定义的。COM 体系结构不允许轻易地对接口作修改，这种方法有助于防止潜在的版本不兼容性。COM 开发者为了给对象提供新的功能，必须努力为对象创建新的接口。COM 对象是在服务器内运行的，服务器为客户访问 COM 对象提供了三种方法。

（1）在服务器中处理（In-process Server）：客户和服务器在相同的内存处理进程中运行，并且通过使用功能调用的方法彼此通信。

（2）本地对象代理（Local Object Proxy）：允许客户使用内部进程通信方法访问服务器，而服务器运行于同一物理机器的一个不同的进程中。这种内部进程通信方法也称为远程过程调用。

（3）远程代理对象（Remote Proxy Object）：允许客户访问在另一台机器上运行的远程服务器。客户和服务器的通信使用分布式计算环境 RPC。远程对象支持这种方法，被称为 DCOM 服务器。

COM 的两个主要的扩展 MTS 和 MSMQ 是由 DCOM 提供的。DCOM 服务器对象支持多线程。MTS（Microsoft Transaction Server）运行于 Windows NT Server，提供事务导向的进程，并且是基于 DTC（Distributed Transaction Coordinator，分布式事务协调）的。MTS 作为一个实时中间环境包含于 DCOM 中，提供对 MSMQ 的异步应用操作，以及针对 RPC（Remote Procedure Call）的同步操作。DCOM 中还内建了远程垃圾收集的功能，使之成为一个健壮的系统。

8.2.3 Java RMI

Java Remote Method Invocation（RMI，Java 远程方法调用）允许开发人员使用 Java 编写分布式对象。下文将介绍 RMI 及其优点。

RMI 为采用 Java 对象的分布式计算提供了简单而直接的途径。这些对象可以是新的 Java 对象，也可以是围绕现有 API 的简单的 Java 包装程序。Java 体现了“编写一次，到处运行”的模式。而 RMI 可将 Java 模式进行扩展，使之可在任何地方运行。

因为 RMI 是以 Java 为核心的，所以，它将 Java 的安全性和可移植性等强大功能带给了分布式计算。开发人员可将代理和业务逻辑等属性移动到网络中最合适的地方。如果要扩展 Java 在系统中的使用，RMI 将使开发人员充分利用其强大功能。

RMI 可利用标准 Java 本地方法接口（JNI）与现有的和原有的系统相连接。RMI 还可利用标准 JDBC 包与现有的关系数据库连接。RMI/JNI 和 RMI/JDBC 相结合，可帮助开发人员利用 RMI 与目前使用非 Java 语言的现有服务器进行通信，而且在需要时可扩展 Java 在这些服务器上的使用。RMI 可帮助开发人员在扩展使用时充分利用 Java 的强大功能。

从最基本的角度看，RMI 是 Java 的远程过程调用（RPC）机制。与传统的 RPC 系统相比，RMI 具有若干优点，因为它是 Java 面向对象方法的一部分。传统的 RPC 系统采用中性语言，所以是最普通的系统，不能提供所有可能的目标平台所具有的功能。

RMI 以 Java 为核心，可以采用本地方法与现有系统相连接。这就是说，RMI 可采用自然、直接和功能全面的方式为开发人员提供分布式计算技术，而这种技术可帮助开发人员以不断递增和无缝的方式为整个系统添加 Java 功能。

RMI 的主要优点如下：

（1）面向对象。RMI 可将完整的对象作为参数和返回值进行传递，而不仅仅是预定义的数据类型。也就是说，开发人员可以将类似 Java 哈希表这样的复杂类型作为一个参数进行传递。而在目前的 RPC 系统中，只能依靠客户机将此类对象分解成基本数据类型，然后传递这些数据类型，最后在服务器端重新创建哈希表。RMI 则不需额外的客户程序代码（将对象分解成基本数据类型），直接跨网络传递对象。

（2）可移动属性。RMI 可将属性（类实现程序）从客户机移动到服务器，或者从服务器移到客户机。例如，可以定义一个检查雇员开支报告的接口，以便查看雇员是否遵守了公司目前实行的政策。在开支报告创建后，客户机就会从服务器端获得实现该接口的对象。如果政策发生变化，服务器端就会开始返回使用了新政策的该接口的另一个实现程序。不必在用户系统上安装任何新的软件就能在客户端检查限制条件，从而向用户提供了快速的反馈，并降低服务器的工作量。这样就能具备最大的灵活性，因为政策改变时只需要编写一个新的 Java 类，并将其在服务器主机上安装一次即可。

（3）设计方式。对象传递功能使开发人员可以在分布式计算中充分利用面向对象技术的强大功能，如二层和三层结构系统。如果能够传递属性，那么开发人员就可以在他的解决方案中使用面向对象的设计方式。所有面向对象的设计方式无不依靠不同的属性来发挥功能，如果不能传递完整的对象，包括实现和类型，就会失去设计方式上所提供的优点。

（4）安全。RMI 使用 Java 内置的安全机制保证下载执行程序时用户系统的安全。RMI 使用专门为保护系统免遭恶意小应用程序侵害而设计的安全管理程序，可保护系统和网络免遭潜在的恶意下载程序的破坏。在情况严重时，服务器可拒绝下载任何执行程序。

（5）便于编写和使用。RMI 使得 Java 远程服务程序和访问这些服务程序的 Java 客户程序的编写工作变得轻松、简单。远程接口实际上就是 Java 接口。服务程序大约用三行指令宣布本身是服务程序，其他方面则与任何其他 Java 对象类似。这种简单方法便于快速编写完整的分布式对象系统的服务程序，并快速地开发软件的原型和早期版本，以便于进行测试和评估。因为 RMI 程序编写简单，所以维护也简单。

（6）可连接现有/原有的系统。RMI 可通过 Java 的本地方法接口 JNI 与现有系统进行交

互。利用 RMI 和 JNI，开发人员就能用 Java 语言编写客户端程序，还能使用现有的服务器端程序。在使用 RMI/JNI 与现有服务器连接时，可以有选择地用 Java 重新编写服务程序的任何部分，并使新的程序充分发挥 Java 的功能。类似地，RMI 可利用 JDBC、在不修改使用数据库的现有非 Java 源代码的前提下与现有关系数据库进行交互。

（7）编写一次，到处运行。RMI 是 Java“编写一次，到处运行”方法的一部分。任何基于 RMI 的系统均可 100%地移植到任何 Java 虚拟机上，RMI/JDBC 系统也不例外。如果使用 RMI/JNI 与现有系统进行交互工作，则采用 JNI 编写的代码可与任何 Java 虚拟机进行编译、运行。

（8）分布式垃圾收集。RMI 采用其分布式垃圾收集功能收集不再被网络中任何客户程序所引用的远程服务对象。与 Java 虚拟机内部的垃圾收集类似，分布式垃圾收集功能允许 用户根据自己的需要定义服务器对象，并且明确这些对象在不再被客户机引用时会被删除。

（9）并行计算。RMI 采用多线程处理方法，可使用户的服务器利用这些 Java 线程更好地并行处理客户端的请求。

（10）Java 分布式计算解决方案。RMI 从 JDK 1.1 开始就是 Java 平台的核心部分，因此，它存在于任何一台 Java 虚拟机中。所有 RMI 系统均采用相同的公开协议，所以，所有 Java 系统均可直接相互对话，而不必事先对协议进行转换。

8.2.4 CORBA、DCOM 和 RMI 的比较

RMI 直接把分布式对象模型嵌入到 Java 语言内部，使得 Java 程序员可以自然地编写分布式程序，不必离开 Java 环境，或者涉及 CORBA IDL 以及 Java 到 CORBA 的类型转换。然而 RMI 不遵守 CORBA 标准，基本上是 Java-to-Java 技术，它需要客户方程序和服务方程序都用 Java 编写，难以实现与其他语言编写的对象之间的互操作。

DCOM 是从 COM 改造过来的。Microsoft 把 DCOM 作为开发 Internet 和组件的基础，已经搭载到 Windows NT 4.0 以上版本和 Windows 98 中。COM 这一技术部分是作为规范，它定义对象实现的二进制标准，用于单机上应用之间的通信，对象实现与使用的语言无关。DCOM 是 COM 的分布式扩展，在 DCE RPC 之上构造对象的远程过程调用层支持对远程对象的访问。一个 DCOM 对象（又称为 ActiveX 对象）是支持一个或多个接口的组件，DCOM 接口指预先规定的一组相关函数。DCOM 类实现一个或多个接口，由一个又一个 128 位类 ID 惟一标识。客户程序通过获得指向 DCOM 对象的接口的指针与该对象交互，通过指针调用其操作。客户程序从不直接访问对象。DCOM 对象不支持对象 ID，因此，客户程序不能与某个特定的对象发生联系。

8.3 基于面向对象技术的应用软件体系结构

软件体系结构是一个程序或系统的构件的组织结构、它们之间的关联关系以及支配系统设计和演变的原则和方针。一般地，一个系统的软件体系结构描述了该系统中的所有计算构件，构件之间的交互、连接件以及如何将构件和连接件结合在一起的约束。

研究软件体系结构的首要问题是如何表示软件体系结构，即如何对软件体系结构建模。针对规模日益庞大、结构日益复杂的应用软件，系统模型的设计目标是提高实际应用系统的

开放性和集成性，同时兼顾效率。

软件系统的开放性包括数据的开放性、功能的开放性和系统的可扩充性，是否具备良好的开放性基本取决于系统模型。软件系统的集成性是指通过一致的信息描述手段和处理机制将各功能子系统统一到同一个集成环境，集成性的好坏也基本取决于系统模型。软件系统的效率通常包括系统运行的效率和应用开发的效率。运行效率是系统运行时的时空复杂度，而应用开发的效率指开发的难易程度和执行效率。效率大部分取决于系统模型，也与系统的具体实现有关。

开放的系统模型使子功能部件的集成易于实现，同时也必然提高应用开发的效率；集成和高效反过来又有利于更好地达到开放的目的。这三者相辅相成，其中又以开放性作为集成和效率的基础，只有开放才有集成，只有开放才有效率。

针对应用软件系统的开放性，曾先后出现了许多类型的系统模型，代表了应用软件技术与产业发展的不同阶段。各阶段中有代表性的系统模型依次为：以数据为中心的系统模型、以执行为中心的系统模型、面向对象的系统模型和基于总线的系统模型。

以数据为中心的系统模型如图 8-3 所示。这类模型将数据库放在系统的核心层次共享，各功能部件采用统一的数据描述，各子系统的开发过程完全独立；子系统间有统一的数据交换接口；整体的可扩充性好，可任意增加符合数据交换标准的应用程序。但是，这种模型整体结构松散，集成性不够良好；只能做到数据复用，不能做到功能复用，造成大量的代码冗余；由于应用相关数据的存在，难于定义符合所有应用需求的数据接口标准，因此会出现数据语义失真。从开放性的角度来讲，这类系统只具有数据开放性，不具有功能开放性，但其可扩充性很好。

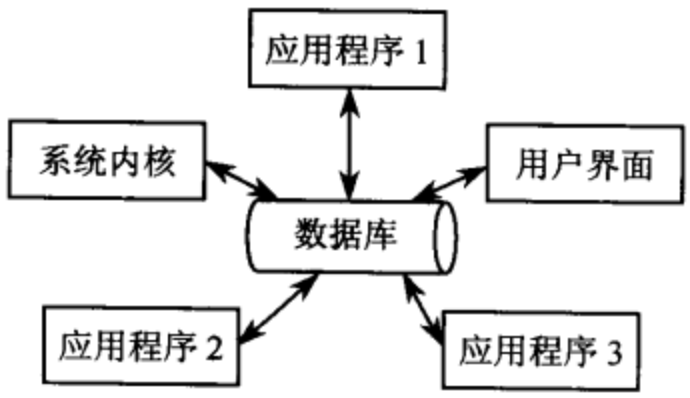


图 8-3　以数据为中心的系统模型

以执行为中心的系统模型着眼于将不同的应用系统通过统一的执行中心来实现数据和用户界面的共享和一致，如图 8-4 所示。这类模型将共有的计算和执行功能从应用程序中分离出来，放在执行中心，避免了代码冗余；用户与系统的交互和应用程序分离，便于实现统一风格的用户界面；和数据库的任何数据交换都要通过执行中心进行，有利于数据的严格管理，保证了数据的一致性。这类模型解决了以数据为中心的系统模型的代码冗余及界面风格不统一等问题，但仍存在一些缺陷：执行中心的功能设计复杂，很难确切定义符合所有应用要求的功能集合，而且实现起来也相当困难；执行中心同时与用户界面和所有应用程序保持通信，又管理着数据，负担过重，容易形成瓶颈。总之，这类模型既具有数据的开放性，又具有功能的开放性，可扩充性也好，整体上优于以数据为中心的系统模型。

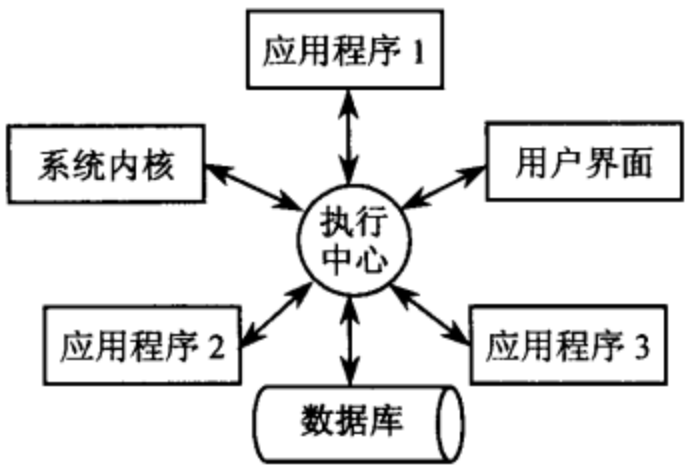


图 8-4　以执行为中心的系统模型

随着面向对象技术的成熟，出现了更为简练的面向对象的系统模型，与前两类模型的设计思想有较大差别，如图 8-5 所示。其中，系统内核对象中封装的是能为用户界面对象和所有应用对象所共享的数据及相应的操作；用户界面对象中封装的是用户界面数据及相应的操作；应用对象中封装的是应用数据及相应的操作。所有这些对象通过相互间的通信协调来完成指定的功能。从系统构成的角度来说，这类模型的结构是无中心的，系统由各对象实体构成，各对象实体具有平等的地位，这与以数据为中心和以执行为中心的模型不同。面向对象的系统模型的主要优点在于，数据和功能的合理封装降低了由于数据和功能的集中管理所带来的通信上的开销和操作上的复杂性；另外，系统的无中心结构也使系统的构成变得更加灵活。从整体上看，面向对象的系统模型无论其开放性还是有效性都要优于前两类模型。

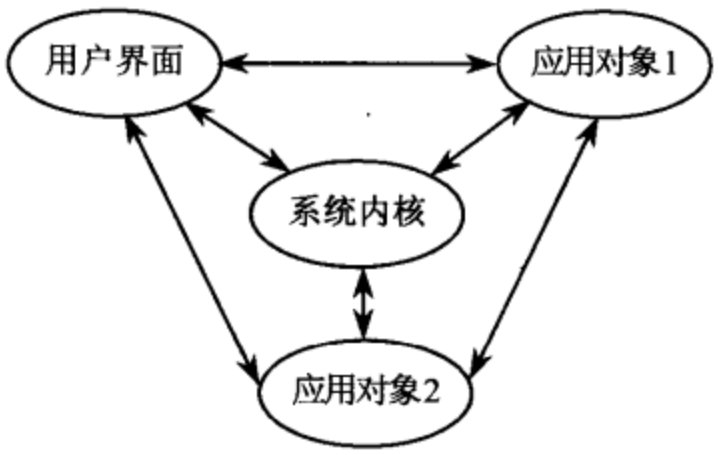


图 8-5　面向对象的系统模型

面向对象模型比以往的模型有了很大的进步，但仍有不足：对象之间的联系是一种点对点的直接联系，当系统中对象数目增加时，通信链接数将以平方级激增；同时，为支持通信，每个对象实体都要维护一个包含所有对象实体功能服务信息的功能服务信息库，这部分信息不但重复，而且还要保证其一致性。这些开销都损害了系统的效率。更大的问题还在于：对象的接口没有一致的标准，造成向系统中扩充对象时的随意与不规范，不利于系统的维护以及对象的复用。

以面向对象为基础的组件和中间件技术的逐渐成熟又为应用软件系统的建模引入了新的思想，产生了基于总线的系统模型，如图 8-6 所示。组件、中间件是继面向对象技术之后发展起来的新的软件工程技术，是面向对象技术的延伸。基于总线的系统模型仍然是一种面向对

象的结构，但系统中的对象是按照规范设计的模块，这些定义良好的软件模块（组件）在系统中共存，并且充分地相互作用。按照这种结构，可以将若干组件组合起来，以建立更大和更复杂的系统。

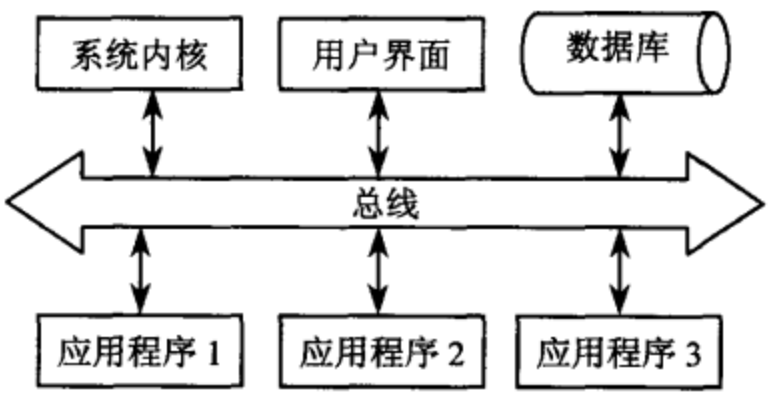


图 8-6　基于总线的系统模型

这种模型的关键在于一种高效的总线结构，使组件之间能以一个公共的接口互相连接，做到组件的即插即用、无缝集成。这种模型的系统中，组件间的通信链接数是线性的，并且由于各组件接口规范的一致性，通信的复杂度大大下降，也提高了组件的互操作性。

根据组件在系统中地位的差异，应用软件系统中的组件可以分为两个层次：核心组件和应用组件。相对于核心组件来说，应用组件所要求的系统服务要少得多，请求服务的频度也较低。相应地，总线也可以分为核心总线和应用总线，从而形成双总线结构。在这种总线结构中，核心组件和应用组件之间仍然保持良好的互操作性，但应用总线屏蔽了应用组件的一部分服务请求，减少了核心总线上的流量，从而提高了应用软件系统核心的效率。

核心组件与核心总线构成了应用软件系统的原型和框架，在此基础上与各应用组件集成。

通常在分布式环境中，应用组件不是直接连接到应用总线（也称 Broker）上，而是通过一个软件代理（Agent）间接地连在总线上，如图 8-7 所示。Agent 的作用在于，一方面代理应用组件的复杂通信过程，使应用组件更专注于功能的实现；另一方面将适应不同应用需求的组件内部的异构数据转换成同构数据，以保证 Broker 上通信语言的统一。由此可见，在基于总线的系统模型中，Broker 与 Agent 构成了介于应用组件（客户）与核心系统（服务器）之间的中间件。

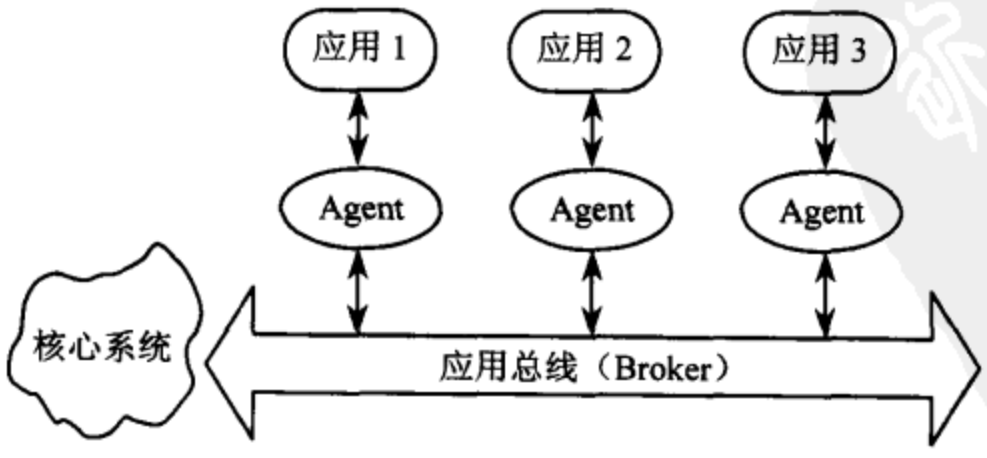


图 8-7　应用系统与核心组件的集成

8.4 Web Service 和 SOA

8.4.1 介绍 SOA

SOA（Service Oriented Architect，面向服务的架构）是指为了解决在 Internet 环境下业务集成的需要，通过连接能完成特定任务的独立功能实体实现的一种软件系统架构。从这个定义中表达的前提有下面两点。

（1）软件系统架构：SOA 不是一种语言，也不是一种具体的技术，而是一种软件系统架构，它尝试给出在特定环境下推荐采用的一种架构，从这个角度上来说，它更像一种模式（Pattern）。因此它与很多已有的软件技术（比如面向对象技术），是互补的而非互斥的。它们分别面向不同的应用场景，来满足不同的特定需求。

（2）SOA 的使用范围：需求决定同时也限制功能。SOA 并不是包治百病的万灵丹，它最主要的应用场合在于解决 Internet 环境下的不同商业应用之间的业务集成问题。在下面会详细讨论 Internet 的各种特点如何决定了 SOA 的特点，这里只需要先简单回顾一下 Internet 环境区别于 Intranet 环境的几个特点：①大量异构系统并存，计算机硬件工作方式不同，操作系统不同，编程语言也不同；②大量、频繁的数据传输仍然速度缓慢并且不稳定；③版本升级无法完成，我们根本就无法知道互联网上有哪些机器直接或者间接地使用某个服务。

基于上面的前提，下面就介绍 SOA 的三大基本特征。

1. 独立的功能实体

在 Internet 这样松散的使用环境中，任何访问请求都有可能出错，因此任何企图通过 Internet 进行控制的结构都会面临严重的稳定性问题。SOA 非常强调架构中提供服务的功能实体的完全独立自主的能力。传统的组件技术，如.NET Remoting、 EJB、COM 或者 CORBA，都需要有一个宿主（Host 或者 Server）来存放和管理这些功能实体；当这些宿主运行结束时，这些组件的寿命也随之结束。这样当宿主本身或者其他功能部分出现问题的时候，在该宿主上运行的其他应用服务就会受到影响。

SOA 架构中非常强调实体自我管理和恢复能力。常见的用来进行自我恢复的技术，比如事务处理（Transaction）、消息队列（Message Queue）、冗余部署（Redundant Deployment）和集群系统（Cluster）在 SOA 中都起着至关重要的作用。

2. 大数据量低频率访问

对于.NET Remoting，EJB 或者 XML-RPC 这些传统的分布式计算模型而言，他们的服务提供都是通过函数调用的方式进行的，一个功能的完成往往需要通过客户端和服务器来回很多次，函数调用才能完成。在 Intranet 的环境下，这些调用给系统的响应速度和稳定性带来的影响都可以忽略不计，但是在 Internet 环境下这些因素往往是决定整个系统是否能正常工作的一个关键决定因素。因此 SOA 系统推荐采用大数据量的方式一次性进行信息交换。

3. 基于文本的消息传递

由于 Internet 中大量异构系统的存在决定了 SOA 系统必须采用基于文本而非二进制的消息传递方式。在 COM、CORBA 这些传统的组件模型中，从服务器端传往客户端的是一个二进制编码的对象，在客户端通过调用这个对象的方法来完成某些功能；但是在 Internet 环境下，

不同语言，不同平台对数据，甚至是一些基本数据类型的定义不同，给不同的服务之间传递对象带来很大的困难。由于基于文本的消息本身是不包含任何处理逻辑和数据类型的，因此服务间只传递文本，对数据的处理依赖于接收端的方式可以帮忙绕过兼容性这个大泥坑。

此外，对于一个服务来说，Internet 与局域网最大的一个区别就是 Internet 上的版本管理极其困难，传统软件采用的升级方式在这种松散的分布式环境中几乎无法进行。采用基于文本的消息传递方式，数据处理端可以只选择性地处理自己理解的那部分数据，而忽略其他的数据，从而得到非常理想的兼容性。

8.4.2　Web Service 与 SOA

就现在而言，Web Service 最适合实现 SOA 的一些技术的集合，事实上最近 SOA 的火爆在很大程度上归功于 Web Service 标准的成熟和应用的普及为广泛地实现 SOA 架构提供了基础。下面让我们看看 Web Service 中的各种协议是如何互相工作来满足 SOA 所需的特点的。

（1）独立的功能实体。通过 UDDI 的目录查找，可以动态改变一个服务的提供方而无须影响客户端的应用程序配置。所有的访问都通过 SOAP 访问进行，只要 WSDL 接口封装良好，外界客户端是根本没有办法直接访问服务器端的数据的。

（2）大数据量低频率访问。通过使用 WSDL 和基于文本（Literal）的 SOAP 请求，可以实现能一次性接收大量数据的接口。这里需要着重指出的是，SOAP 请求分文本方式和远程调用（RPC）两种方式，正如上文已经提到的，采用远程调用方式的 SOAP 请求并不符合这点要求。但是令人遗憾的是，现有的大多数 SOAP 请求采用的仍然是远程调用（RPC）方式，在某些平台上，例如 IBM WebSphere 的早期版本，甚至没有提供文本方式的 SOAP 支持。

（3）基于文本的消息传递。Web Service 所有的通信是通过 SOAP 进行的，而 SOAP 是基于 XML 的，不同版本之间可以使用不同的 DTD 或者 XML Schema 加以辨别和区分。因此只需要为不同的版本提供不同的处理就可以轻松实现版本控制的目标。

无论开发人员现在的系统是否牵涉到基于 Internet 的业务集成，采用 SOA 推荐的架构都对提高开发人员系统的扩展性有很大帮助，下面是在系统中引入 SOA 后需要在软件架构方面做出的改变。

（1）使用基于文本方式的 SOAP 调用，摆脱远程调用中出现的函数参数类型等与数据无关的信息，保证所有 SOAP 传递的都是有意义的商业数据。依赖于 Schema，而不是类定义对这些数据进行解释。

（2）传统的三层 Web 应用将可能变成四层结构。传统意义上的商业逻辑层将被进一步划分为存放每个会话（Session）信息的客户逻辑层和与状态无关（Stateless）的 SOA 层。

8.5　小结

本章介绍了分布式计算以及几种分布式对象 CORBA、Java RMI、DCOM。并且，介绍了发展迅速的 Web Service 技术。后面章节将通过范例讲述这些技术。

第 9 章　RMI 编程范例

在目前的程序开发中，分布式计算是解决大型应用的一种重要手段。它指的是一种应用程序的设计模式，其中的程序、所处理的数据和计算能力都分布于网络中。这样可以很好地平衡各个计算机的处理能力，另外这也是具体应用发展的需要。现代大型应用程序都倾向于从单一的一个应用程序向由位于不同地方的一些小应用程序协同工作所组成。

远程方法调用（Remote Method Invocation，RMI）是 Sun 公司规定的允许在不同的 JAVA 虚拟机（Java Virtual Machine，JVM）之间进行对象间通信的一种规范。在 RMI 中，JVM 可以位于一个或多个计算机上，其中一个 JVM 可以调用存储在另一个 JVM 中的对象方法。这就使得应用程序可以远程调用其他对象的方法，从而达到分布式计算的目的，以共享各个系统的资源和处理能力。

在具体介绍 RMI 结构之前，先比较一下 RMI 和其他一些实现不同 JAVA 虚拟机上的应用程序之间通信技术的区别。除了 RMI 外，基于 JAVA 的实现不同 JAVA 虚拟机上的应用程序之间通信的技术主要有两种：套接字和 JAVA 消息服务（Java Messaging Service，JMS）。

使用套接字是实现程序间通信的最为灵活和强大的方式。但是它必须通过应用级协议进行通信，要求应用程序之间使用同样的协议，并且要求设计通信过程中的错误判断等。

JMS 与 RMI 的区别在于，采用 JMS 服务，对象是在物理上被异步地从网络的某个 JVM 上直接移动到另一个 JVM 上。而 RMI 对象是绑定在本地 JVM 中，只有函数参数和返回值是通过网络传送的。

9.1　介绍 RMI

RMI 的规范定义："……在 JAVA 分布式对象模型中，远程对象是指它的方法可以从另外一个位于不同主机上的 JAVA 虚拟机来调用的对象。这种类型的对象采用一个或多个远程接口进行描述，这些接口声明了远程对象的方法。远程方法调用（RMI）就是对一个远程对象的远程接口中的方法进行调用……"。

RMI 的目的就是要使运行在不同的计算机中的对象之间的调用表现得像本地调用一样。RMI 应用程序通常包括两个独立的程序：服务器程序和客户机程序。需要注意的是，服务器程序和客户机程序不是绝对的而是相对的，"服务器程序"是指具有可被远程调用的方法的单个远程对象；"客户机程序"是指要调用远程对象上的远程方法的对象。因此，一个对象可以同时是客户机程序和服务器程序。

典型的服务器应用程序将创建多个远程对象，使这些远程对象能够被引用，然后等待客户机调用那些远程对象上的方法。而典型的客户机程序则从服务器中得到一个或多个远程对象的引用，然后调用远程对象的方法。RMI 为服务器和客户机进行通信和信息传递提供了一种机制，这样的应用程序就被称为分布式对象应用程序。得到一个远程对象的引用显然和得到一个本地对象的引用有所不同，但是只要得到了对象的一个引用，就可以像使用本地对象

一样使用该远程对象。RMI 的结构机制自动解释对象方法的调用，以寻找远程对象或处理远程请求，甚至可以实现远程垃圾回收。

需要注意的是，远程对象总是通过引用来访问的，客户机程序只有在得到了远程对象接口的引用之后才能调用远程对象上的方法。

9.2　一个 RMI 会话系统

本节将以一个简单的例子说明如何使用 RMI 结构进行远程方法调用。采用 RMI 开发客户机/服务器应用程序一般包括下面 6 个基本步骤：

（1）定义远程接口。

（2）实现这个远程接口。

（3）生成 stub（客户代理）和 skeleton（服务器实体）。

（4）编写使用远程对象的客户程序。

（5）启动注册表并登记远程对象。

（6）运行服务器和客户程序。

如前所述，要实现远程调用需要一个客户程序和一个服务器程序。下面就分别描述如何构建客户程序和服务器程序。

9.2.1　构建服务器程序

RMI 对象对接口有着强烈的依赖。在需要创建一个远程对象的时候，通过传递一个接口来隐藏基层的实施细节。所以客户得到远程对象的一个句柄正好同一些本地的根代码连接，由后者负责通过网络通信。但我们并不关心这些事情，只关心通过自己的接口句柄发送和接收消息。

创建一个远程接口时，必须遵守下列规则：

- 远程接口必须为 public 属性（不能有“包访问”；也就是说，它不能是“友好的”）。否则，一旦客户试图装载一个实现了远程接口的远程对象，就会得到一个错误。
- 远程接口必须扩展接口 java.rmi.Remote。
- 除与应用程序本身有关的异常外，远程接口中的每个方法都必须在自己的 throws 从句中声明 java.rmi.RemoteException。
- 作为参数或返回值传递的一个远程对象（不管是直接，还是本地对象中嵌入），必须声明为一个远程接口，不可声明为实例化的一个类。

下面是远程接口 RMITest1I.java 的代码，如下所示：

```
//rmiTest1 远程接口
package rmiTest;
import java.rmi.*;
public interface RMITest1I extends Remote {
long getPerfectTime() throws RemoteException;
}
```

它表面上与其他的接口类似，只是对 Remote 进行了扩展。同时，所有的方法都会“抛”出 RemoteException。该文件中的接口和方法都是 Public 的。

使用以下命令编译 RMITest1I.java，以生成 RMITest1I.class（rmiTest 是一个包，编译时注意路径）：C:\RMI>javac rmiTest\RMITest1I.java。

服务器程序必须包含一个扩展了 UnicastRemoteObject 类并实现该远程接口的类。这个类也可以包含其他附加的方法，但客户只能使用远程接口中的方法。因为客户是指向远程接口的一个句柄，而不是它的哪个类。

必须为远程对象定义构件函数，即使只准备定义一个默认构件函数，以用它调用基础类的构件函数。必须把它明确地编写出来，因为它必须“抛”出 RemoteException 异常。

下面列出远程接口实现类 RMITest1 的代码：（它代表精确计时服务）

```
//RMITest1I 远程接口的实现类 RMITest1
package rmiTest;
import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
public class RMITest1 extends UnicastRemoteObject implements RMITest1I
{
//默认构件函数，它也要“抛”出 RemoteException 异常
public RMITest1() throws RemoteException {
super();
}

public long getPerfectTime() throws RemoteException {
return System.currentTimeMillis();
}

public static void main(String[] args) {
/*创建和安装一个安全管理器，以让其支持 RMI。作为 Java 开发包的一部分，适用于 RMI 的惟一一个是 RMISecurityManager*/
if (System.getSecurityManager() == null) {
System.setSecurityManager(new RMISecurityManager());
}

try {
/*创建远程对象的一个或多个实例，下面是 RMITest1 对象*/

RMITest1 rt = new RMITest1();

/*向 RMI 远程对象注册表中注册至少一个远程对象。一个远程对象拥有的方法即可生成指向其他远程对象的句柄，这样客户到注册表里访问一次，得到第一个远程对象即可*/
Naming. rebind("//localhost/RMITest1", rt);
System.out.println("Bind OK.");
} catch (Exception e) {
e.printStackTrace();
}
}
}
```

使用以下命令编译 RMITest1.java，以生成 RMITest1.class（rmiTest 是一个包，编译时注

意路径）：C:\RMI>javac rmiTest\RMITest1.java。

在有了远程接口及其实现以后，就需要在注册表中对它进行绑定，以使客户能获得该对象。客户可以在主机上通过字符名称来寻找对象。为了注册一个对象，首先需要生成一个 stub 和 skeleton（需要的话）。从注册表传回给客户的是代表一个远程对象的 stub。

下面说明如何创建一个远程对象的 stub 和 skeleton。要完成这个工作可使用 rmic 编译器，rmic 编译器生成远程对象的本地代理和服务器实体。存根（Stub）是远程对象在客户端的一个代理，它将 RMI 调用传递给服务器端的服务器实体（Skeleton），后者负责将该调用传递给实际的远程方法。输入如下命令：

```
C:\RMI>rmic -d C:\RMI rmiTest.RMITest1
```

在 Java2 中可以使用“-v1.2”参数来抑制服务器实体的产生，Java2 中可以只使用本地代理。

执行这个命令，若 rmic 成功运行，rmiTest 目录里就会多出两个新类：RMITest1_Stub.class 和 RMITest1_Skel.class，它们分别对应的是本地代理（stub）和服务器实体（skeleton）。

9.2.2　构建客户程序

RMI 全部的宗旨就是尽可能地简化远程接口对象的使用。客户程序中要做的惟一一件额外的事情是查找主机上的注册表，并从服务器得到远程对象的一个引用。注意到把这个引用转化为远程接口类型是很关键的。需要记住的是，在 RMI 中，客户程序总是和接口进行交互，而从不直接与对象实现进行交互。

下面就是客户机 DisplayPerfectTime.java 程序的代码，如下所示：

```
//使用远程对象 PerfectTime
package rmiTest;

import java.rmi.*;
import java.rmi.registry.*;

public class DisplayPerfectTime {
/*** DisplayPerfectTime 构造子注解*/
public DisplayPerfectTime() {
super();
}

public static void main(String[] args) {
if (System.getSecurityManager() == null) {
System.setSecurityManager(new RMISecurityManager());
}
try {
RMITest1I t = (RMITest1I) Naming.lookup("//localhost/RMITest1");
for (int i = 0; i < 10; i++) {
System.out.println("PerfectTime:" + t.getPerfectTime());
}
} catch (Exception e) {
e.printStackTrace();
}
}
```

```
}
```

使用以下命令来编译 DisplayPerfectTime.java 文件：C:\RMI>javac rmiTest\DisplayPerfectTime.java。

9.2.3 注册对象并启动服务器和客户程序

在运行 RMITest1 类和 DisplayPerfectTime 类之前，用户首先必须在宿主 RMITest1 的计算机上启动 RMI 注册(Registry)程序。即使将要运行 RMITest1 的计算机与运行 DisplayPerfectTime 的是同一台机器，这一步也是必需的。注册表服务器的名字是 rmiregistry。在 32 位 Windows 环境中，可使用 start rmiregistry 命令让它在后台运行。然后分别打开两个不同的进程运行服务器端和客户机端。

除非已经安装了允许 RMI 操作的默认安全策略，否则用下列命令运行 RMITest1 时会出现安全异常：

```
C:\RMI>java rmiTest.RMITest1
```

要生成 JAVA 安全策略文件，重载 RMI 安全管理器附带的一些安全限制。下面的代码显示了仅供测试的安全策略文件（rmiTest.policy）。

```
grant {
    //完全放开限制
    permission java.security.AllPermission;
};
```

注意：真正的运行系统中一定不要使用该策略文件，因为它将关闭 RMI 安全管理器的所有安全检查，使系统安全性降低。

为绑定 RMITest1 到注册表，运行服务端程序，可以用 java.security.policy 系统属性指定其他的安全策略。在 Windows 下，输入下列命令以在后台启动 RMITest1 程序：

```
C:\RMI>java -Djava.security.policy= rmiTest.policy rmiTest.RMITest1
```

然后运行客户端程序，如下所示：

```
C:\RMI>java rmiTest.DisplayPerfectTime

PerfectTime: 1034688545197
PerfectTime: 1034688545197
PerfectTime: 1034688545197
PerfectTime: 1034688545197
PerfectTime: 1034688545197
PerfectTime: 1034688545207
PerfectTime: 1034688545207
PerfectTime: 1034688545207
PerfectTime: 1034688545207
PerfectTime: 1034688545207
```

9.3 带有回调的 RMI 会话

前面已经讨论了客户程序是如何进行远程调用的。事实上，在很多 RMI 应用程序中，客户机不仅要远程调用服务器程序，而且服务器经常要回调客户程序，例如进度反馈或管理通知。一个更恰当的例子是交谈程序，每个客户程序同时也是服务器程序。图 9-1 显示了一个回

调过程。

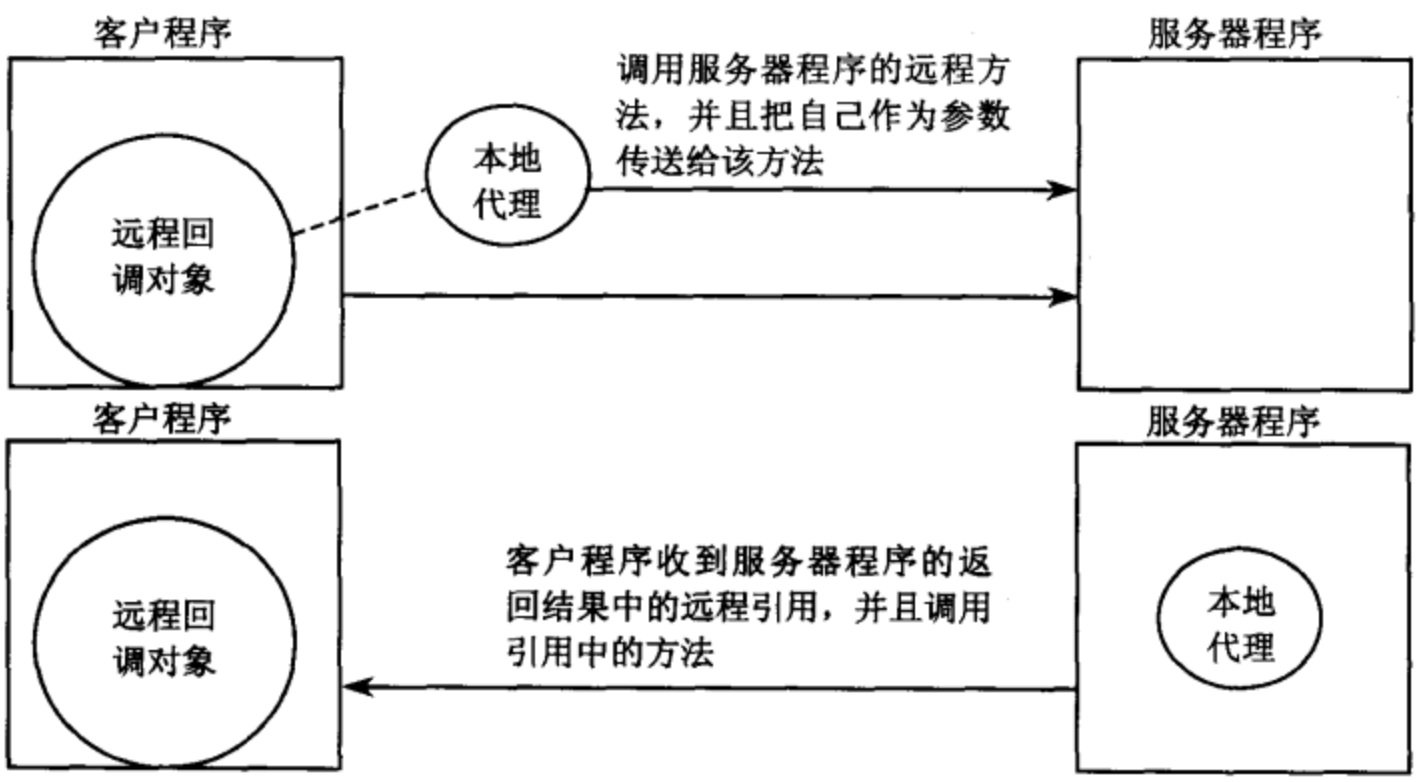


图 9-1　带有回调的 RMI 会话

实现远程回调的具体过程和一般的远程调用程序的步骤基本相同，主要的区别在于客户程序和服务器程序都是远程对象。

9.3.1　服务器程序

下面是服务器端远程对象接口 RMITest2I.java 的代码：

```
//RMITest2 远程接口
package rmiTest;

import java.rmi.*;

public interface RMITest2I extends Remote {
    /* RMITest2ClientI 是客户程序应该实现的远程接口*/
    public void registerClient(RMITest2ClientI client) throws RemoteException;
    public void unregisterClient(RMITest2Client client) throws RemoteException;
}
```

下面列出远程接口实现类 RMITest2 的代码：

```
//RMITest2I 远程接口的实现类 RMITest2
package rmiTest;

import java.net.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.util.*;

public class RMITest2 extends UnicastRemoteObject implements RMITest2I,
Runnable{
```

```
    protected HashSet clients;
    //默认构件函数，它也要抛出 RemoteException 异常
    public RMITest2() throws RemoteException {
        super();
        clients = new HashSet();
}
public void run(){
        for (;;) {
            Iterator iter = clients.iterator();
            while(iter.hasNext()) {
                RMITest2ClientI rt = (RMITest2ClientI)iter.next();
            try {
                    rt.getPerfectTime();
                }catch(Exception exc) {
                    System.out.println("移除无效对象。");
                    iter.remove();
                }
            }
            try{
                Thread.sleep(1000);
            }catch(Exception ignore){}
        }
    }
    public void registerClient(RMITest2ClientI client) throws RemoteException{
        System.out.println("加入客户对象。");
        clients.add(client);
    }

public void unregisterClient(RMITest2ClientI client) throws RemoteException{
    System.out.println("移除客户对象。");
    clients.remove(client);
}
public static void main(String[] args) {
    try {
        /*创建和安装一个安全管理器，以让其支持 RMI。作为 Java 开发包的一部分，
        *适用于 RMI 的惟一一个是 RMISecurityManager。*/
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        /*创建远程对象的一个或多个实例，下面是 RMITest2 对象*/
        RMITest2 rt = new RMITest2();
        /*向 RMI 远程对象注册表中注册至少一个远程对象。一个远程对象拥有的
        *方法即可生成指向其他远程对象的句柄，这样客户到注册表里访问一次，
        *得到第一个远程对象即可。*/
        Naming. rebind("//localhost/RMITest2", rt);
        System.out.println("Bind OK.");
        (new Thread(rt)).start();
    } catch (Exception e) {
        e.printStackTrace();
    }
```

```
    }
}
```

9.3.2　客户程序

下面是客户程序需要实现的远程接口 RMITest2ClientI.java 的代码：

```
//RMITest2ClientI 远程接口
package rmiTest;
import java.rmi.*;
public interface RMITest2ClientI extends Remote
{
    public void getPerfectTime() throws RemoteException;
}
```

下面是实现 RMITest2ClientI 远程接口的客户程序 RMITest 2Client.java 的代码：

```
RMITest2Client.java
//RMITest2ClientI 远程接口的实现
package rmiTest;
import java.rmi.*;
import java.rmi.server.*;
public class RMITest2Client extends UnicastRemoteObject implements RMITest2ClientI
{
    public RMITest2Client() throws RemoteException
    {
    }
    public void getPerfectTime() throws RemoteException
    {
        System.out.println("PerfectTime: " + System.currentTimeMillis());
    }
    public static void main(String[] args)
    {
        try{
            if (System.getSecurityManager() == null)    {
                System.setSecurityManager(new RMISecurityManager());
            }
            RMITest2ClientI client = new RMITest2Client();
            RMITest2I server = (RMITest2I)Naming.lookup("//localhost/RMITest2");
            server.registerClient(client);
        }catch(Exception exc){
            exc.printStackTrace();
        }
    }
}
```

需要注意的是，对于客户机和服务器程序来说，都需要生成一个本地代理和服务器实体。

9.4　远程对象激活

使用 RMI 时的一个问题是需要手工启动服务器，并且客户机要在服务器程序将服务器对象注册到 RMI 注册表之后才能使用服务器对象。为此，RMI 激活框架提供了另一种访问服务

器对象的方法，即利用特殊安装程序，可以使 RMI 激活框架启动服务器对象。

我们所讨论的远程对象基本上都是 java.rmi.UnicastRemoteObject 类的实例。远程对象在任何时候都是可以访问的，即使没有客户程序正在执行也是如此。考虑这样一种情况，在服务器上当远程对象的数目或远程对象使用的资源数量很大时，将成为系统的一个主要性能瓶颈。因此，针对这种情况提出了对象激活的概念。这样一来，就不需要保证每个服务器都运行，只要保证运行激活服务器 rmid 即可。

对象激活就是允许远程对象根据需要被执行。例如，当访问一个“可被激活的”远程对象（通过方法调用）时，如果那时远程对象没有运行，系统就会在适当的 JVM 中初始化这个对象的执行。RMI 则采用了滞后激活，就是把对象激活操作推迟，直到客户第一次使用该对象，即第一次进行方法调用。

从客户的观点来看，一个可激活的对象与通常的远程对象之间没有任何不同。对于客户来说，整个激活机制是完全透明的，客户程序永远都不会意识到这个激活过程背后发生的事情。

为了实现激活功能，在激活框架中有几个相互协作的实体：

- 远程对象本身。
- 包装程序，用来注册对象。通常情况下，包装程序向激活系统进行几个方法调用，以提供应当如何激活对象的有关细节。
- 第三个实体是激活驻留程序，它记录像注册表之类的信息，如关于什么时候与对象有何关系等（驻留程序是 rmid）。

这时可以关联这些实体。一个远程对象被激活的步骤如下：

- 远程对象应当继承 java.rmi.activation.Activatable 类，而不是 java.rmi.UnicastRemoteObject 类。
- 远程对象应当包含一个采用两个参数的特殊的构造函数。这两个参数中，第一个是 ActivationID 类型的激活标识符；第二个为可选的激活数据，即一个 java.rmi.MarshalledObject 对象。这不像非激活的远程对象那样包含一个没有参数的构造函数。当要激活对象时，RMI 系统就会调用这个特殊的构造函数。
- 创建一个激活描述器（java.rmi.activation.ActivationDesc），并用 Activator（rmid）来注册它。

而对于远程对象的远程接口则不必采取什么改变或修改，毕竟要改变的是对象的实例的实现，而不是改变如何从外部看待对象。

下面将以 9.2 节的程序为例，将其修改为可激活的服务器程序。

9.4.1 创建远程接口

下面是远程对象接口代码，它和前面的远程对象接口没有什么不同。

RMI_Active_Interface.java 的代码如下所示：

```
package demo.rmi.rmi_active;

import java.rmi.*;

public interface RMI_Active_Interface extends Remote
{
    long getPerfectTime() throws RemoteException;
```

```
}
```

9.4.2　实现远程接口

下面是远程对象接口的实现，注意构造函数的不同。

RMI_Active.java 的代码如下所示：

```
package demo.rmi.rmi_active;

import java.rmi.activation.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.util.*;

public class RMI_Active extends Activatable implements RMI_Active_Interface
{
    //特殊的构造函数，它也要抛出 RemoteException 异常
    public  RMI_Active(ActivationID  id,  MarshalledObject  data)  throws
RemoteException
    {
        super(id, 0);
    }

    //远程接口中函数的实现
    public long getPerfectTime() throws RemoteException
    {
        return System.currentTimeMillis();
    }
}
```

这里的构造函数只是简单地调用了 super(id, 0)，也可以通过 data 参数来传递数据。

9.4.3　注册激活对象程序

下面是用于注册激活对象的程序 RMI_Active_Setup.java 的代码：

```
package demo.rmi.rmi_active;

import java.rmi.activation.*;
import java.rmi.*;
import java.rmi.registry.*;
import java.rmi.server.*;
import java.util.*;

public class RMI_Active_Setup
{
    public static void main(String[] args)
    {
        /*创建和安装一个安全管理器，以让其支持 RMI。作为 Java 开发包的一部分，
*适用于 RMI 的惟一一个是 RMISecurityManager。*/
        if (System.getSecurityManager() == null)
        {
```

```
            System.setSecurityManager(new RMISecurityManager());
        }

    try
    {
     /*设置程序的策略文件*/
     Properties secProperties = new Properties();
     secProperties.put("java.security.policy", "c:\\demo\\rmi\\policy");

     /*使用默认环境参数*/
     ActivationGroupDesc.CommandEnvironment env = null;

     /*创建 ActivationGroupDesc 对象*/
     ActivationGroupDesc group =
new ActivationGroupDesc(secProperties, env);
     /*生成一个 ActivationGroupID*/
     ActivationGroupID groupID =
ActivationGroup.getSystem().registerGroup(group);
     /*注册 group*/
     ActivationGroup.createGroup(groupID, group, 0);
     String codeLocation = "file:///C:/";
     /*创建 ActivationDesc 对象，并以远程对象类名和加载类的路径为参数。
*还可以传递激活对象的初始数据，这里是 null。*/
ActivationDesc desc = new
ActivationDesc("demo.rmi.rmi_active.RMI_Active", codeLocation, null);
      /*使用激活服务注册对象*/
      RMI_Active_Interface rt = (RMI_Active_Interface)Activatable.register(desc);

/*向 RMI 远程对象注册表中注册远程对象。一个远程对象拥有的方法即可生成指向
*其他远程对象的句柄，这样客户到注册表里访问一次，得到第一个远程对象即可。*/
      Naming. rebind("//localhost/RMI_Active", rt);

      System.out.println("Bind OK.");
      System.exit(0);
    }catch (Exception e){
      e.printStackTrace();
    }
     }
}
```

9.4.4 客户程序

客户程序和一般的 RMI 客户程序没有什么不同。下面是客户程序 RMI_Active_Client.java 的代码：

```
package demo.rmi.rmi_active;

import java.rmi.*;
import java.rmi.registry.*;
```

```
public class RMI_Active_Client
{
    public static void main(String[] args)
    {
        /*创建和安装一个安全管理器，以让其支持 RMI。作为 Java 开发包的一部分，
*适用于 RMI 的惟一一个是 RMISecurityManager*/
        if (System.getSecurityManager() == null)    {
            System.setSecurityManager(new RMISecurityManager());
        }
        try{
            RMI_Active_Interface t = (RMI_Active_Interface)
            Naming.lookup("//localhost/RMI_Active");
            for (int i = 0; i < 10; i++){
                System.out.println("PerfectTime:" + t.getPerfectTime());
            }
        } catch (Exception e){
            e.printStackTrace();
        }
    }
}
```

9.4.5　运行程序

运行程序时要按下面几步执行：

（1）生成远程对象的本地代理和服务器实体。

（2）启动 RMI 注册表：

```
rmiregistry -Djava.security.policy=c:\demo\rmi\policy
```

（3）运行 rmid 并指定策略文件：

```
rmid -Djava.security.policy= c:\demo\rmi\policy
```

（4）运行用于注册激活对象的程序，也需要指定策略文件：

```
java -Djava.security.policy= c:\demo\rmi\policy
demo.rmi.rmi_active.RMI_Active_Setup
```

（5）运行客户程序，也需要指定策略文件：

```
java -Djava.security.policy= c:\demo\rmi\policy
demo.rmi.rmi_active.RMI_Active_Client
```

9.5　在 IIOP 上运行 RMI

基于 Internet 对象请求中介协议（Internet Inter-ORB Protocol，IIOP）上的 RMI 在 Java 中直接集成了公共对象请求代理协议（Common Object Request Broker Architecture，CORBA）的分布式计算环境。尽管 CORBA 和 RMI 是相互竞争的技术，但在 JAVA 中它们可以一起使用，这要归功于 RMI-IIOP 将 CORBA 与 RMI 结合在一起。

CORBA 1.0 最大的问题之一是没有定义对象通信的标准协议，每个 ORB 实现方法各不相同。CORBA 2.0 引入了标准协议，使不同的 ORB 可以相互通信。其中的基本协议是通用 ORB 间协议（General Inter-ORB Protocol，GIOP）。GIOP 定义 ORB 在各种请求中要发送的数据项目，但并没有提出实际网络协议。CORBA 2.0 还定义了基于 TCP/IP 的 Internet ORB 间协议

（IIOP）。不管它是否要支持其他协议，每个 ORB 都要实现 IIOP。

RMI-IIOP 是由 IBM 和 SUN 共同开发的，它是一个针对 IIOP 的 RMI 新版本。它结合了 RMI 的易编程特点和 CORBA 的互操作性特征。从某种程度上说，RMI-IIOP 是 RMI 与 CORBA 结合的结果，因为远程接口可以用 Java 来编写，并且用 JAVA RMI API 来实现。这些接口也可以用其他任何一种 OMG 映射所支持的语言来实现，这些语言也被厂商提供的 ORB 所支持。类似地，客户程序也可以用其他语言来编写，在其中使用 Java 远程接口派生而来的 IDL。

9.5.1 服务器端程序

远程接口的编写和一般 RMI 程序没有什么不同。下面是远程接口 RMI_IIOP_Interface.java 的代码：

```
package demo.rmi.rmi_iiop;

import java.rmi.*;

public interface RMI_IIOP_Interface extends Remote
{
    public long getPerfectTime() throws RemoteException;
}
```

远程接口的实现和一般 RMI 程序有所不同，主要表现在它需要继承的是 PortableRemoteObject 类，而不是 UnicastRemoteObject 类。下面是远程接口实现的 RMI_IIOP.java 的代码：

```
import javax.rmi.PortableRemoteObject;
import javax.naming.*;
import java.rmi.*;
import java.util.*;

public class RMI_IIOP extends PortableRemoteObject implements RMI_IIOP_Interface
{
    public RMI_IIOP() throws RemoteException
    {
        super();
    }

    //远程接口中函数的实现
    public long getPerfectTime() throws RemoteException
    {
        return System.currentTimeMillis();
    }
}
```

注册远程对象的程序和注册过程与一般的 RMI 程序不太一样。首先使用 rmic 带上-iiop 选项为远程对象产生本地代理和服务器实体的同时，为 IIOP 产生一个 tie 类。然后需要运行 tnameserv.exe 代替 rmiregistry 作为名字服务器。这个服务器提供了 IIOP CosNaming 服务，客户通过访问它来查询对象。最后在启动注册远程对象程序时，需要设定 JNDI 上下文的两个重要的环境参数：java.naming.factory.initial 和 java.naming.provider.url。下面是注册远程对象的 RMI_IIOP_Setup.java 的代码：

```
package demo.rmi.rmi_iiop;

import java.rmi.*;
import java.util.*;
import javax.naming.*;

public class RMI_IIOP_Setup
{
    public static void main(String[] args)
    {
        /*创建和安装一个安全管理器，以让其支持 RMI。作为 Java 开发包的一部分，
*适用于 RMI 的惟一一个是 RMISecurityManager*/
        if (System.getSecurityManager() == null)    {
            System.setSecurityManager(new RMISecurityManager());
        }
   try
   {
    RMI_IIOP rmi_iiop = new RMI_IIOP();
    Context ctx = new InitialContext();
    ctx.rebind("/RMI_IIOP", rmi_iiop);
    System.out.println("Bind OK!");
   }catch (Exception e){
     e.printStackTrace();
   }
    }
}
```

9.5.2　客户端程序

客户端程序也需要作相应的改动，它使用 JNDI 上下文来获得一个对象的引用，而不是使用注册表。InitialContext 对象的 lookup()方法返回一个 Object 对象，它必须用 PortableRemoteObject 的 narrow()方法来进行类型转换。下面是客户端程序 RMI_IIOP_Client.java 的代码：

```
package demo.rmi.rmi_iiop;

import java.rmi.*;
import javax.rmi.PortableRemoteObject;
import javax.naming.*;

public class RMI_IIOP_Client
{

    public static void main(String[] args)
    {
        /*创建和安装一个安全管理器，以让其支持 RMI。作为 Java 开发包的一部分，
*适用于 RMI 的惟一一个是 RMISecurityManager*/
        if (System.getSecurityManager() == null) {
            System.setSecurityManager(new RMISecurityManager());
        }
        try
        {
```

```
    System.setProperty("java.naming.factory.initial",  "com.sun.jndi.cosnaming.
CNCtxFactory");
            Context ctx = new InitialContext();

            //获取 JNDI 名为 RMI_IIOP 的对象
                Object obj = ctx.lookup("/RMI_IIOP");

                //进行类型转换
        RMI_IIOP_Interface t = (RMI_IIOP_Interface)PortableRemoteObject.narrow(obj,
RMI_IIOP_Interface.class);
                for (int i = 0; i < 10; i++){
                    System.out.println("PerfectTime:" + t.getPerfectTime());
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

9.5.3 运行程序

运行程序分为以下几步：

（1）生成远程对象的本地代理、服务器实体和 tie 类：

```
rmic -iiop demo.rmi.rmi_iiop.RMI_IIOP
```

（2）运行名字服务器（默认端口为 900）：

```
tnameserv
```

（3）运行注册远程对象的程序：

```
java -Djava.naming.factory.initial=com.sun.jndi.cosnaming.CNCtxFactory
-Djava.naming.provider.url=iiop://localhost:900
-Djava.security.policy=c:\RMI\rmiTest.policy
demo.rmi.rmi_iiop.RMI_IIOP_Setup
```

（4）运行客户端程序：

```
java  -Djava.security.policy=c:\RMI\rmiTest.policy  demo.rmi.rmi_iiop.RMI_
IIOP_Client
```

9.6 小结

本章介绍了关于 RMI 程序的基本框架和概念，并以简单的例子说明了在实际中的应用方式。RMI 是许多 Java 技术的基础，事实上，从 Java 还没有分布式模型的开始阶段到现在，RMI 已经出现有一段时间了。现在在 J2EE 中 RMI 是 Java 分布式模型的核心内容，许多其他技术都是基于 RMI 的模型和结构，如 JINI 和 JavaSpace 等，EJB 更是与 RMI-IIOP 紧密结合，它们在 JDK 核心中集成，以作为企业中间件的基本技术。如果读者还想对 RMI 有更深的了解，可以参考 Sun 公司的 RMI 规范。

第 10 章　EJB 编程范例

10.1　了解 EJB

Enterprise Bean 是实现了 Enterprise JavaBean（EJB）技术的 J2EE 组件。Enterprise Bean 在 EJB 容器这个 J2EE 服务器的运行环境中运行，如图 10-1 所示。

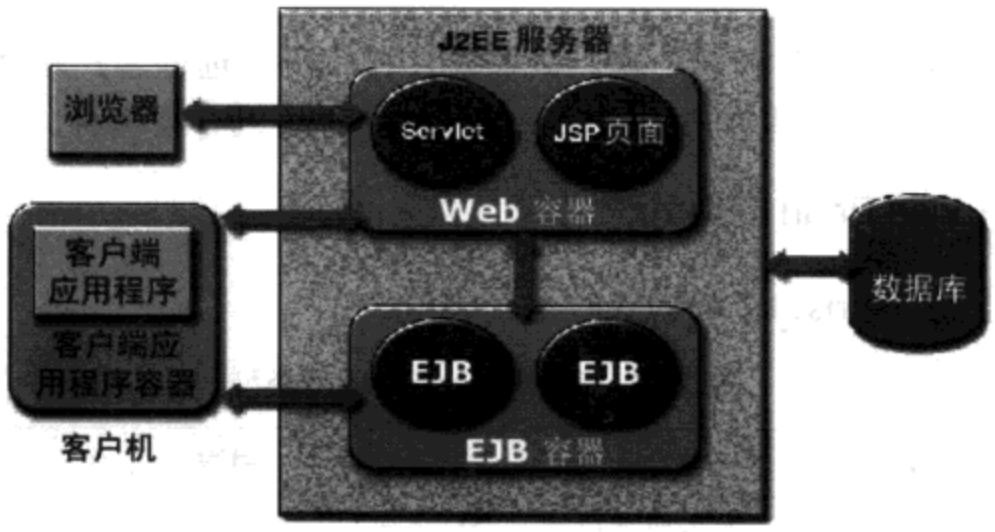


图 10-1　Enterprise Bean 的运行环境

EJB 容器提供了诸如 Enterprise Bean 的事务处理这样的系统级服务，不过这对于应用程序开发者来说是透明的。这些服务的存在使得用户可以快速地构建和部署 Enterprise Bean，实际上 Enterprise Bean 构成了事务型 J2EE 应用程序的核心。

10.1.1　Enterprise Bean 是什么

一个 Enterprise Bean 就是一个用 Java 编程语言编写的服务器端的组件，其中包含了应用程序的商业逻辑。所谓商业逻辑，是指实现应用程序用途的代码。例如，在一个存货管理的应用程序中，Enterprise Bean 可能会在名为 checkInventoryLevel 和 orderProduct 的方法中实现该程序的商业逻辑。客户端通过调用这些方法来访问应用程序提供的存货管理服务。

10.1.2　Enterprise Bean 的优点

Enterprise Bean 的使用大大减轻了分布式应用程序开发的工作量。

首先，因为 EJB 容器为 Enterprise Bean 提供了系统级的服务，bean 的开发者可以集中精力解决商业逻辑问题。EJB 容器而不是 bean 的开发者，来负责诸如事务管理和安全论证这样的系统级服务。

其次，因为是 bean 而不是客户端包含了应用程序的商业逻辑，客户端的开发者可以集中精力于客户端的外观。客户端的开发者不再需要编写执行商业规则或访问数据库的程序。这样做的结果是客户端成为真正意义上的瘦客户端，另外还有一个好处就是降低了对客户端的

硬件要求。

第三，因为 Enterprise Bean 是一种可移植的组件，应用程序的部署人员可以使用已有的 bean 来构建新的应用程序。同时，这些应用程序可以运行在任何合适的 J2EE 服务器上。

10.1.3 使用 Enterprise Bean 的时机

如果应用程序有以下几个需求之一时，就应该考虑使用 Enterprise Bean：

- 应用程序必须具有可伸缩性。为了适应不断增加的用户，你可能需要发布适应许多机器的应用程序组件。而应用程序的 Enterprise Bean 不仅可以运行在不同的机器上，而且它们的位置对于客户端来说也是完全透明的。
- 事务要求确保数据完整性。Enterprise Bean 支持事务，可以管理对共享对象的并行访问。
- 应用程序可能会有多种客户端。只需要几行代码，远程客户端就可以很方便地定位 Enterprise Bean。这些客户端可以只有几种，也可以有很多种。

10.1.4 Enterprise Bean 的类型

表 10-1 概括了 Enterprise Bean 的 3 种不同类型。

表 10-1 Enterprise Bean 的类型

Enterprise Bean 类型	用途
Session Bean	为客户端执行一个任务
Entity Bean	表现一个持久存储的商业实体对象
Message-Driven Bean	充当 Java 消息服务 API 中监听者的角色，以处理异步的消息（EJB2.0 中才有）

本章将详细介绍如何在企业应用程序中使用这 3 种 EJB。在这之前先了解一下 EJB 规范及相关技术。

一个 Enterprise Bean 类包含对组件的实现细节。Session Bean 的实现不同于 Entity Bean 的实现，一个 Session Bean 针对单一的客户完成一次连接或会话，其生存直到客户完成连接与会话或系统意外中止。当一个新的客户从 EJB 服务器访问到一个 Session Bean 时，EJB 容器创建一个新的 Session Bean 实例，其运行直到会话结束。Session Bean 实现从 Enterprise Bean 接口派生出的 javax.ejb.Session Bean 接口。Entity Bean 实现接口 javax.ejb.EntityBean，其描述了特定数据源中的数据。它能长时间存在于 EJB 容器中，不会随系统的意外终止而消失，并且可以让多个客户同时访问。

EJB 规范定义了许多 bean 类能够实现的标准接口，并且它定义了所有的 bean 类必须有的方法。然后容器调用这些方法来管理 bean。所有 bean 类（无论是会话 bean，还是实体 bean）必须实现的最基本的接口是 javax.ejb.EnterpriseBean。javax.ejb.EnterpriseBean 接口是所有 EJB 的基础接口。

10.1.5 EJB 对象

当客户想使用 Enterprise Bean 类的一个实例时，客户不必直接在实际的实例上调用方法。

该调用过程被 EJB 容器截取，bean 实例用容器中的对象代表。

Enterprise Bean 类不能通过网络直接被调用。我们知道 EJB 容器可以操纵网络，因此它通过容器将 bean 包装成可在网络上使用的对象。通过截取请求，容器可以自动执行许多必要的管理工作。EJB 容器可以跟踪哪个方法被调用，在系统管理者的用户接口上显示其用法等。因此，EJB 容器可以看作间接的存在于客户代码和 bean 之间的层。这个间接的层使用单独的网络对象来表示自己，这个对象称为 EJB 对象。EJB 对象作为容器物理的部分；所有的 EJB 对象都有针对容器特殊要求的代码。因此，容器提供商提供专门工具，来自动为 EJB 对象产生类文件。

10.1.6　RMI 和 EJB 对象

应当注意到 java.ejb.EJBObject 继承了 Java.rmi.Remote 类。Java.rmi.Remote 接口是 Java 远程方法调用（RMI）的一部分，任何一个实现 java.rmi.Remote 的对象都是 remote 对象，它可以被另外的 Java 虚拟机所调用。容器提供的 EJB 对象实现了远程接口，同时也间接地实现了 java.rmi.Remote 类，这样也就意味着你的 EJB 对象是完全符合网络需要的，可以被网络上的其他 Java 虚拟机调用。当然，EJB 接口也必须遵守 EJB 规范。EJB 远程接口必须遵守 Java 的 RMI 远程接口规范。例如：对于错误处理，二者的处理方式相同。远程接口同样也必须遵守 Java RMI 参数传递规范。不是什么都可以通过远程方法调用来在网络上传递，传递的参数必须符合 RMI 类型。

EJB 也继承了 RMI 的优点。对于 RMI，你正在调用的远程对象的物理地址是不可见的。这个特点同样也适用于 EJB，客户代码不必关心正使用的 EJB 对象是在邻近的计算机上还是从 Internet 传递来的。这样，EJB 对象可以和客户端处在同一个 JVM 中。

EJB 保证了本地分布式组件的透明度，这种透明对于多层配置来说是非常必要的。客户端代码是非常容易移植的，不受限于特殊的多层配置。EJB 容器可以以最佳化方式在本地执行。

10.1.7　远程接口

前面了解到，bean 客户调用 EJB 对象上的方法来代替调用 bean。为了执行它，EJB 对象必须复制 bean 类中的每个业务方法。但是，怎样才能使自动产生的 EJB 对象知道复制了哪个方法呢？这就用到了 bean 提供者编写的一个特殊的接口，这个接口复制所有与 bean 类相关联的业务逻辑方法。这个接口被称为远程接口。

远程接口必须遵循 EJB 规范的定义，所有的远程接口必须从 Sun 公司提供的通用接口继承而来，即 javax.ejb.EJBObject。

下面是一个 EJB 对象：

```
public interface javax.ejb.EJBObject
extends java.rmi.Remote
{
public abstract javax.ejb.EJBHome getEJBHome() throws
          java.rmi.RemoteException;
public abstract java.lang.Object getPrimaryKey() throws
       java.rmi.RemoteException;
public abstract void remove() throws java.rmi.RemoteException,
       javax.ejb.RemoveException;
```

```
public abstract javax.ejb.Handle getHandle() throws
       java.rmi.RemoteException;
public abstract boolean isIdentical(javax.ejb.EJBObject) throws
       java.rmi.RemoteException;
}
```

以上是对于所有 EJB 对象都必须拥有的方法。你不需要实现这些方法，这些方法的实现在生成 EJB 对象时由容器自动生成。客户端代码通过调用 javax.ejb.EJBObject 的方法来和 bean 协同工作。

10.1.8 Home 接口

Home 接口简单地定义了建立、删除和寻找 EJB 对象的方法。容器的 home 对象实现了 home 接口。通常，EJB 定义了所有 home 接口必须支持的许多方法，这些必需的方法定义在 javax.ejb.EJBHome 接口上，home 接口必须继承 Java.ejb.EJBHome 接口。

下面的代码是 javax.ejb.EJBHome 接口：

```
public interface javax.ejb.EJBHome extends java.rmi.Remote
{
public abstract EJBMetaData getEJBMetaData() throws
       java.rmi.RemoteException;
public abstract void remove(Handle handle) throws java.rmi.RemoteException,
       javax.ejb.RemoveException;
public abstract void remove(Object primaryKey) throws
       java.rmi.RemoteException,javax.ejb.RemoveException;
}
```

注意 javax.ejb.EJBHome 继承了 java.rmi.Remote 类，这意味着 home 接口同样也支持 RMI 远程对象，它所传递的参数也和 RMI 相同。

10.1.9 Home 对象

客户端代码处理 EJB 对象，而从不直接操作 bean。那么客户端如何得到 EJB 对象的一个引用呢？

客户端不直接将 EJB 对象实例化，因为 EJB 对象可以存在于不同的机器中。同样地，EJB 使本地透明化，因此客户端不知道它的确切所在。客户端代码通过 EJB 对象工厂得到 EJB 对象的一个引用。EJB 规范里称这种工厂为 home 对象，它主要起以下作用：

- 建立 EJB 对象。
- 找到已经存在的 EJB 对象。
- 删除 EJB 对象。

另外在一些细节方面，EJB 对象工厂同 EJB 对象的特征相同。

10.2 Session Bean 概述

10.2.1 Session Bean 是什么

一个 Session Bean 表现了在 J2EE 服务器中一个单一的客户端。客户端调用 Session Bean

的方法以访问部署在服务器上的应用程序。Session Bean 为它的客户端执行工作，通过在服务器中执行商业任务以简化客户端工作的复杂性。

正如它的名字所暗示的那样，Session Bean 和交互式的 Session 很类似。Session Bean 不可以是共享的，它可能仅仅只有一个客户端。同样地，交互式的 Session 可能也只有一个用户。和交互式的 Session 一样，Session Bean 也不是持久稳固的（也就是说，它的数据没有被存储到数据库中）。当客户端终止时，它的 Session Bean 也会终止并且不再与客户端关联。

在企业级应用系统内，Session Bean 是一种代表客户程序执行操作的 EJB。对于 EJB 客户程序来说，Session Bean 常常起着入口点或前端 EJB 的作用。EJB 客户程序通过与 Session Bean 的交互，从企业应用系统获取它们想要利用的功能或服务。

Session Bean 有两种类型：有状态 Session Bean（Stateful Session Bean）和无状态 Session Bean（Stateless Session Bean）。

10.2.2　有状态 Session Bean

一个对象的状态由实例变量的值来描述。对于有状态的 Session Bean 来说，实例变量描述了客户程序与 Bean 会话的状态。鉴于客户程序与 Bean 的交互关系，Bean 的状态信息通常称为会话状态。有状态 Session Bean 在 EJB 之内保留的状态信息与 EJB 客户程序有着明确的关系。有状态 Session Bean 的状态信息是指保存在 Bean 实例的域里面的数据，以及 Bean 实例持有的各种对象里面的数据。当一个 EJB 客户程序在某一时刻访问一个有状态 Session Bean，且改变了该 Bean 实例的状态，则状态信息将被保留。下一次 Bean 再次被访问时，Bean 的实例将使用原先保存的状态信息。

在客户程序与 Bean 交互期间，状态信息将一直有效。如果客户程序运行结束或拆除了 Bean，则会话结束，状态信息也不再保留。然而，状态信息的这种临时性并不成为问题，因为当客户程序与 Bean 之间的会话终止时，状态信息也就没有必要再保存了。

对于有状态 Session Bean 来说，容器承担着更多的 Bean 管理方面的责任。实际上，客户程序创建或拆除有状态 Session Bean，将直接关系到服务器端 Bean 实例的创建和拆除。此外，当资源紧张时，容器可能决定把一个或者多个有状态 Session Bean 串行化到持久性存储设备。一旦资源重新空闲，或出现了客户程序的请求，被串行化的 Bean 必须激活并转入活动内存。因此，在设计有状态 Session Bean 时，开发者必须考虑更多的问题。

10.2.3　无状态 Session Bean

一个无状态 Session Bean 不为一个具体的客户端维持一个会话状态。当一个客户端调用一个无状态 Session Bean 的方法时，bean 的实例变量可能会包含一个状态，但这个状态仅仅存在于调用的过程之中。当方法结束时，这个状态将不再保留。无状态会话 Bean 不在 EJB 之内保留面向特定客户程序的状态信息，但这并不意味着这类 EJB 不在本身的域或关联的对象里面保留任何状态数据。其真实含义是，这类 Bean 保持的状态信息不是为特定 EJB 客户程序下一次访问或使用而保留的。除了在方法调用的过程中以外，一个无状态 Session Bean 的所有实例是完全等价的，这就使得 EJB 容器可以将一个实例分配给任何客户端。

这种特点使得 EJB 容器能够更高效、更灵活地管理无状态 Session Bean。在任意时刻，任意一个客户程序可以使用容器创建任意一个无状态 Session Bean 的实例。因此，容器可以为这

类实例构造一个缓冲池，根据客户程序的需求从缓冲池分配 Bean 的实例，而无须顾虑哪一个实例属于哪一个客户程序。此外，必要时容器能够方便地创建或拆除 Bean 的实例，并根据应用规模和资源情况作出调整。虽然无状态 Session Bean 可能拥有状态信息，但在对 Bean 实例的连续两次调用之间，开发者不能假定这些状态信息的合法性。

因为无状态 Session Bean 可以支持许多客户端，它们可以为应用程序提供更好的可伸缩性。典型的情况是，对于相同数量的客户端，应用程序所需要的无状态 Session Bean 会比有状态 Session Bean 少得多。

有时，EJB 容器可能会将有状态 Session Bean 写入到次级存储器中。然而，无状态 Session Bean 从不需要写入到次级存储器中。因此，无状态 bean 与有状态 bean 相比可以提供更好的性能。

10.2.4 Session Bean 接口

编写一个 Session Bean 的类时，必须实现 javax.ejb.Session Bean 接口：

```
public interface javax.ejb.Session Bean extends javax.ejb.EnterpriseBean
{
public abstract void setSessionContext(SessionContext ctx)
throws java.rmi.RemoteException;
public abstract void ejbPassivate() throws java.rmi.RemoteException;
public abstract void ejbActivate() throws java.rmi.RemoteException;
public abstract void ejbRemove() throws java.rmi.RemoteException;
}
```

会话 bean 和实体 bean 都继承了 javax.ejb.EnterpriseBean 接口，让我们详细看看该接口中的各种方法。

1. setSessionContext(SessionContext ctx)

EJB 容器调用这个方法来通过会话上下文与 bean 连接。Bean 可以通过会话上下文向容器查询当前事物的状态和当前的安全状态等。

```
import javax.ejb.*;
public class MyBean implements Session Bean {
private SessionContext ctx;
public void setSessionContext(SessionContext ctx) {
this.ctx = ctx;
}
...
}
```

2. ejbCreate(...)

该方法用来初始化你的 Session Bean，可以定义多个不同参数的 ejbCreate()方法来用不同的方法初始化 bean。

```
import javax.ejb.*;
public class MyBean implements Session Bean {
    private int memberVariable;
    public void ejbCreate(int initialValue) {
        this.memberVariable = initialValue;
    }
    ...
}
```

ejbCreate()方法是容器可以调用的一个回调方法。客户端代码不能调用它，因为客户端不能直接处理 bean 对象，它们必须通过容器。客户端必须采用某种方法向 ejbCreate 方法传递参数，以提供初始化参数。Home 接口是客户端用来初始化调用的接口工厂。必须在 home 接口中复制每一个 ejbCreate()方法，例如，如果在 bean 类有下面的 ejbCreate 方法：

```
public void ejbCreate(int i) throws ...
```

那么必须在 home 接口里有下面的 create()方法 ：

```
public void create(int i) throws ...
```

客户端调用 home 接口中的 create()方法，将参数传递给 ejbCreate()。

3. ejbPassivate()

如果出现太多的实例 bean，EJB 容器可以将它们中的一些串行化，写到临时的存储空间，例如数据库或文件系统。同时，容器将释放它们所申请的空间。

```
import javax.ejb.*;
public class MyBean implements Session Bean {
public void ejbPassivate() {
<例如关闭 socket 等...>
}
...
}
```

4. ejbActivate()

当客户需要使用被串行化的 bean 时，容器将被串行化的 bean 重新导入内存，并激活它们。这时 bean 又被导入内存，需要重新得到 bean 所需要的资源。

```
import javax.ejb.*;
public class MyBean implements Session Bean {
public void ejbActivate() {
<例如打开 socket 等...>
}
...
}
```

5. ejbRemove()

当容器将会话 bean 的一个实例删掉时，将调用此方法。所有的 bean 都有这种方法，它没有任何参数，并且将释放所有已分配的资源。

```
import javax.ejb.*;
public class MyBean implements Session Bean {
public void ejbRemove() {
<处理销毁对象等操作>
}
...
}
```

容器可以在任何时候调用 ejbRemove()方法，但如果遇到异常，则有可能禁止容器调用此方法。

10.3　无状态 Session Bean 开发示例

为了使 bean 可以在任一容器中工作，bean 必须被附在一个接口中。在 EJB 中，在 Enterprise

Bean 类中提供了企业级 bean 组件的实现，这是个简单的遵循接口的 Java 类。从客户端来看，会话对象的生命周期如图 10-2 所示。

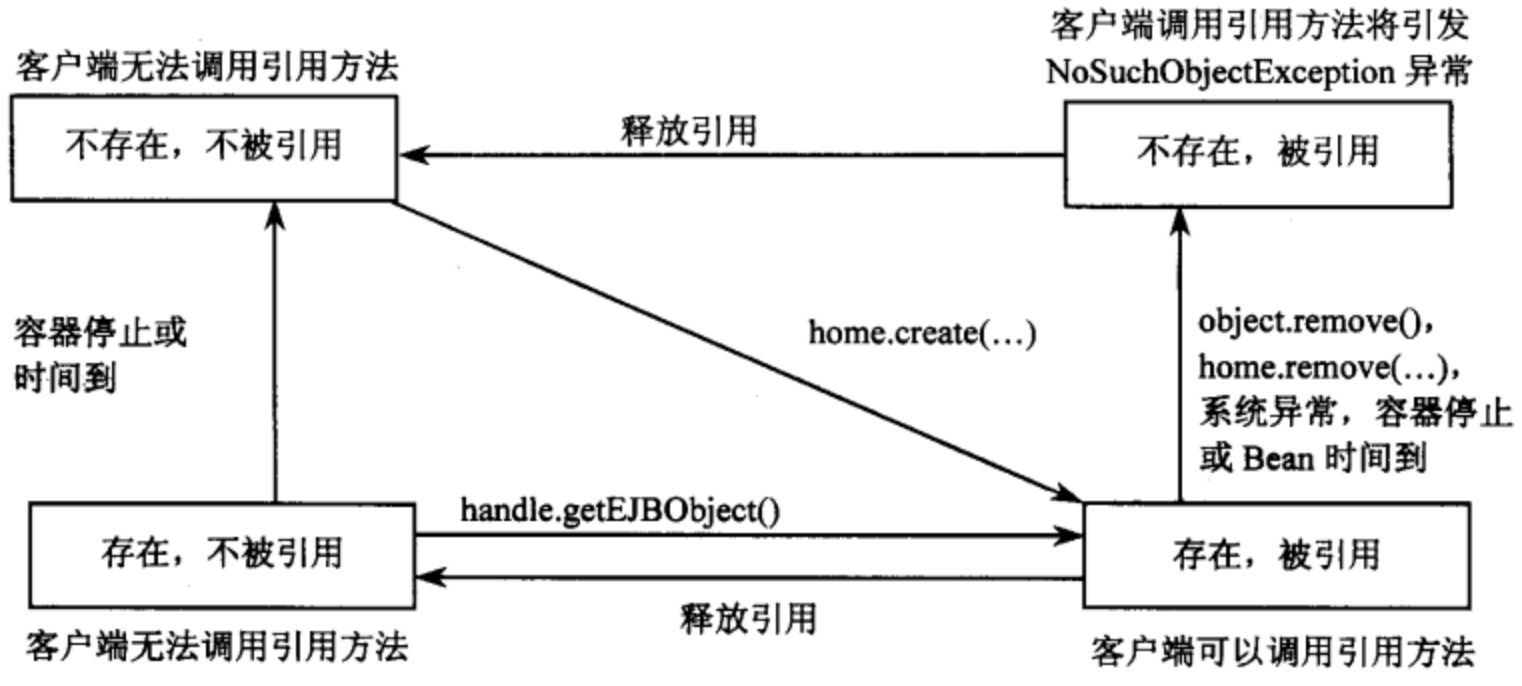


图 10-2　会话对象的生命周期

本节提供的例子演示了一个简化的股票交易操作，通过这个实例读者可以了解：

- 如何定义无状态 Session EJB 的主接口、远程接口和实现类。
- 如何使用对象作为远程方法的返回值。
- 如何在 ejb-jar.xml 中定义 EJB 属性和在无状态 Session EJB 中通过会话上下文使用环境参数。

对于一个无状态 Session EJB 应用程序来说，至少需要两个接口和两个类，它们是主接口和远程接口以及 Bean 实现类和客户端类。本例中开发一个简单的只打印欢迎信息的 Bean。根据前面的介绍，这个 Bean 应该包含 3 个部分。

- 主接口：StatelessHelloWorldHome。
- 远程接口：StatelessHelloWorld。
- Bean 实现类：StatelessHelloWorldImp。

本章后面还会比较无状态会话 Bean 和有状态会话 Bean 的生存周期的差别。我们将把注意力集中在怎样实现一个 Bean，而不是介绍怎样实现业务方法。下面将一步一步介绍无状态 Session Bean 的开发。

10.3.1　主接口（StatelessHelloWorldHome）

由于无状态 Session EJB 不需要保存自身的状态，所以在设计上比较简单，只需要定义一个不带参数的 create 方法。主接口的代码如下所示：

```
package demo.ejb.stateless;

import javax.ejb.EJBHome;
import java.rmi.*;

public interface StatelessHelloWorldHome extends EJBHome
{
```

```
    //无状态 Session Bean，只有一个不带任何参数的 create 方法
    public StatelessHelloWorld create() throws RemoteException;
}
```

这里定义的 create()方法没有任何参数。create()方法带参数时要求在生成过程中初始化 Bean 的状态，但是这里是一个无状态会话 Bean，所以不需要任何参数。create()方法返回远程对象的一个引用。对于运行时错误，这个方法将抛出异常或向客户生成异常。

10.3.2　远程接口（StatelessHelloWorld）

StatelessHelloWorld 远程接口只声明了一个方法 sayHello，这个方法将在 Bean 实现类中实现。整个接口代码如下所示：

```
package demo.ejb.stateless;

import javax.ejb.*;
import java.rmi.*;

public interface StatelessHelloWorld extends EJBObject
{
    public String sayHello(String strGreet) throws RemoteException;
}
```

10.3.3　Bean 实现类（StatelessHelloWorldImp）

StatelessHelloWorldImp 实现类实现了一个 Session Bean 接口。正如前面介绍的，这个接口有几个需要实现的回调方法。整个实现类代码如下所示：

```
package demo.ejb.stateless;

import java.rmi.*;
import javax.ejb.*;

public class StatelessHelloWorldImp implements Session Bean
{
    private SessionContext ctx;

public void setSessionContext(SessionContext arg0)
        throws EJBException, RemoteException
    {
        ctx = arg0;
    }

    public void ejbRemove() throws EJBException, RemoteException{
    }

    public void ejbActivate() throws EJBException, RemoteException{
    }

    public void ejbPassivate() throws EJBException, RemoteException{
    }
    //对应 Home 接口中的 create 方法
```

```
    public void ejbCreate() throws CreateException{
    }

    public String sayHello(String strGreet){
        return "Say " + strGreet;
    }
}
```

Bean 容器会在 Bean 生命周期的适当时间调用这些回调方法。在这里大部分都只给了一个空的方法实现。注意远程方法实现时并不抛出 RemoteException，但远程接口要求声明这个异常，EJB 容器将处理这个异常。这样，如果 Bean 实现类中的业务方法产生了异常，容器将向客户抛出 RemoteException 异常。

10.3.4 部署 EJB

EJB 的另一个很不容易适应的方面是不用去运行 EJB，而是需要部署 EJB。可以用某种封装工具生成包含 EJB 类的 JAR 文件、一些 XML 部署描述文件和一些针对所用容器的帮助类。每个 EJB 都需要一个部署描述文件，最简单的 Bean 也不例外。每个 EJB 厂家都提供自己的定制部署工具。本章介绍如何使用 Sun 公司的 J2EE SDK 进行部署。J2EE SDK 的部署工具是 Deployment Tool，它在 SDK 的 bin 目录中。在启动 Deployment Tool 后，出现如图 10-3 所示的界面。

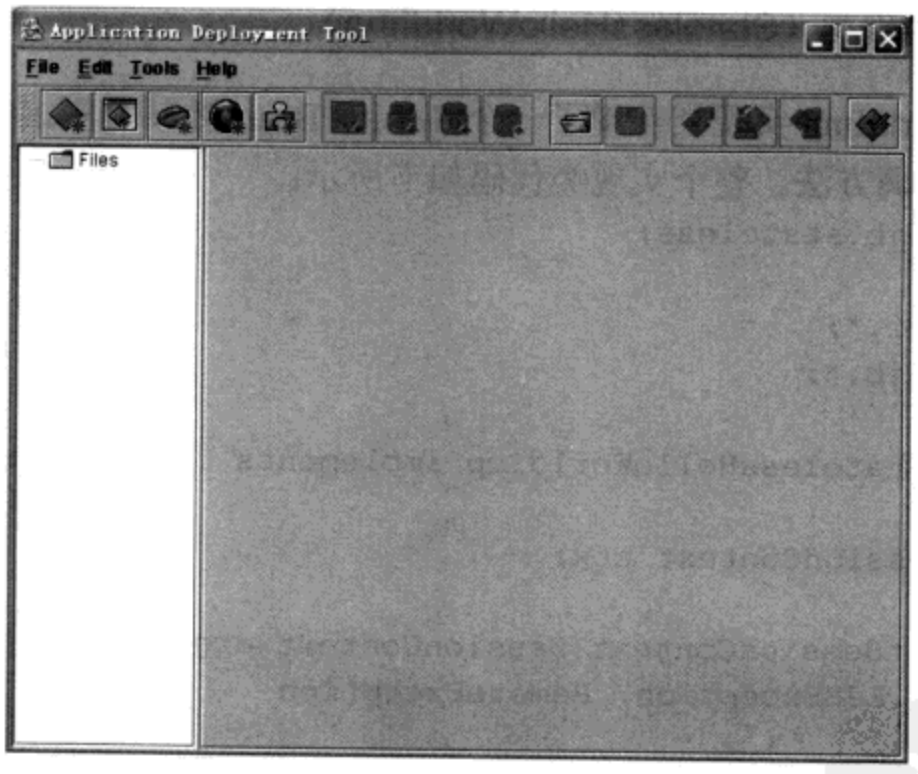


图 10-3 Deployment Tool 的界面

从 File 菜单中选择 New→Application 选项，在 New Application 对话框中输入要生成的 EAR（Enterprise Archive）文件名。EAR 文件就是 J2EE 应用程序的 JAR 文件。图 10-4 显示了该对话框。

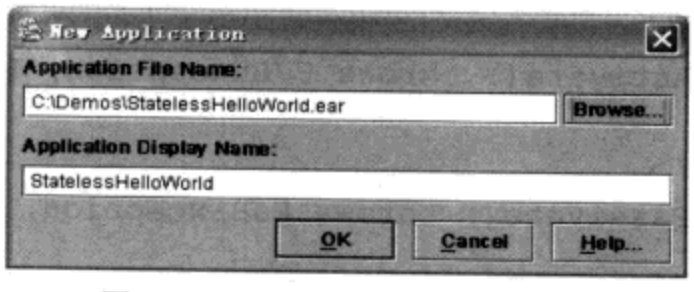


图 10-4 New Application 对话框

单击 OK 按钮后，结果如图 10-5 所示。

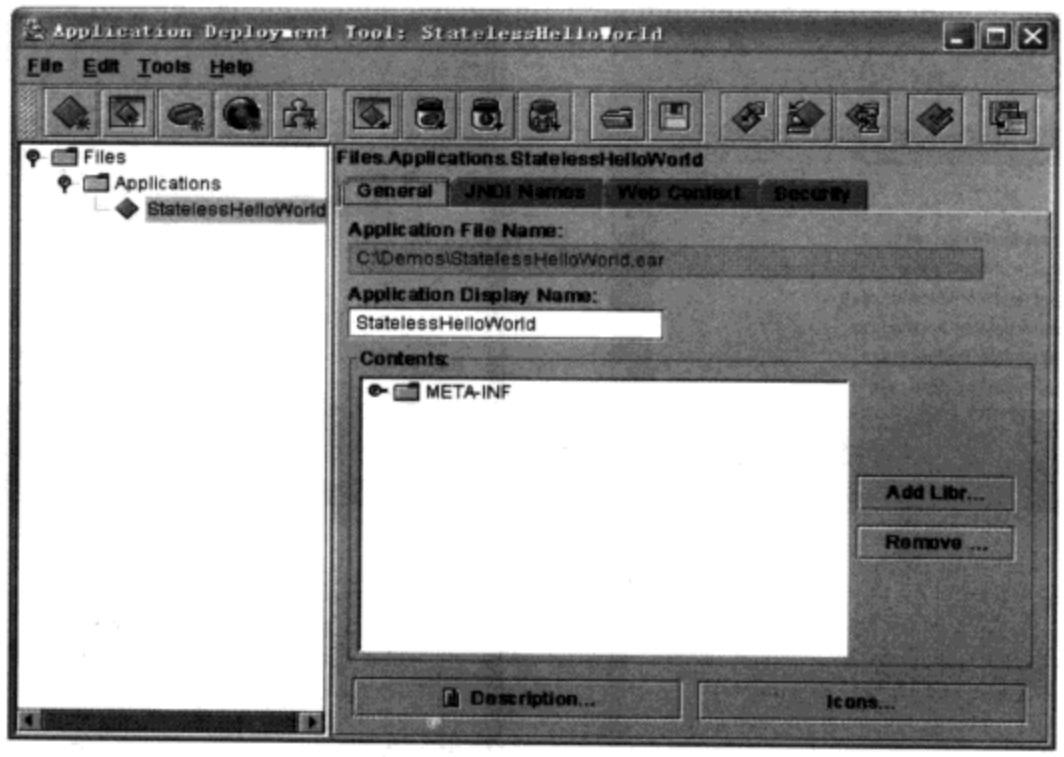


图 10-5　选择 New Application 后的界面

从 File 菜单中选择 New→Enterprise Bean，这时将显示 New Enterprise Bean 向导，可以改变 JAR 文件的显示名称。我们把 JAR 文件的显示名称改为 HelloWorld，如图 10-6 所示。

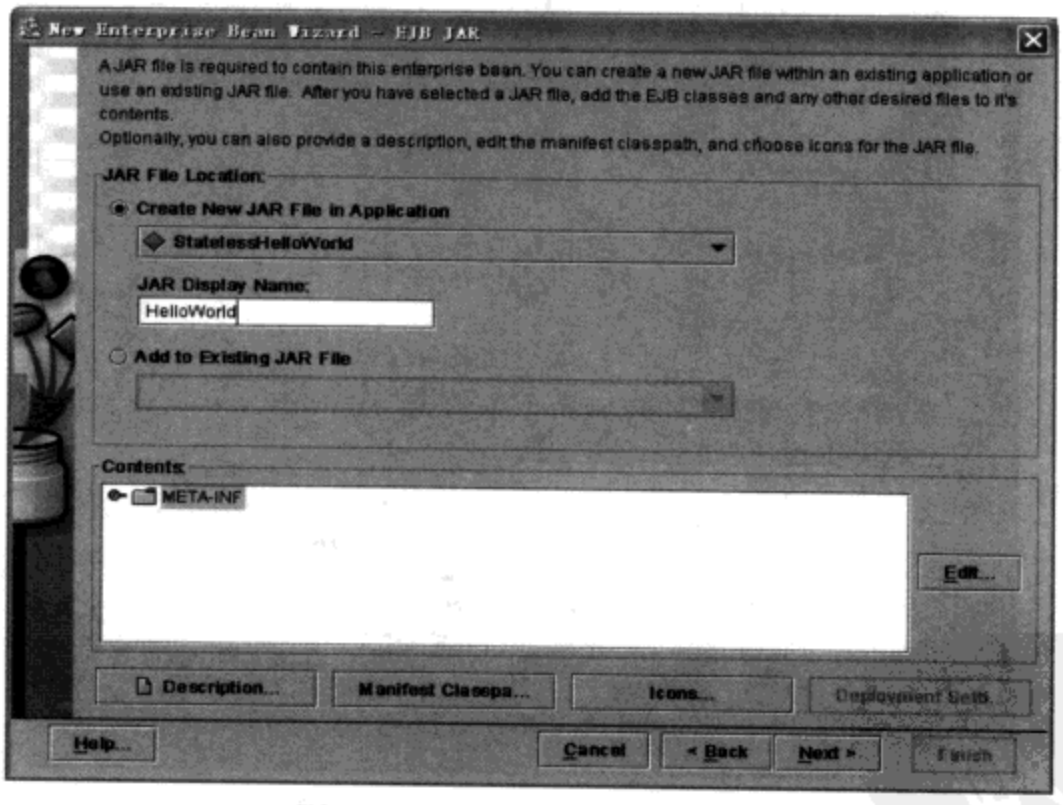


图 10-6　设置 JAR 的显示名称

单击右下角 Contents 区旁边的 Edit 按钮，将出现如图 10-7 所示的对话框。

通过这个对话框可以选择构成 EJB 的类文件，这里选择 StatelessHelloWorld.class、StatelessHelloWorldHome.class 和 StatelessHelloWorldImp.class 文件。单击 Add 按钮，如图 10-8 所示。

下面要确定 Enterprise Bean 类、Home 接口和 Remote 接口所使用的类。可以对 Bean 指定显示名称，并且只在 Deployment Tool 中使用，还有确定是 Session Bean 还是实体 Bean，以及

Session Bean 是有状态的还是无状态的。图 10-9 显示了如何配置这些项目。

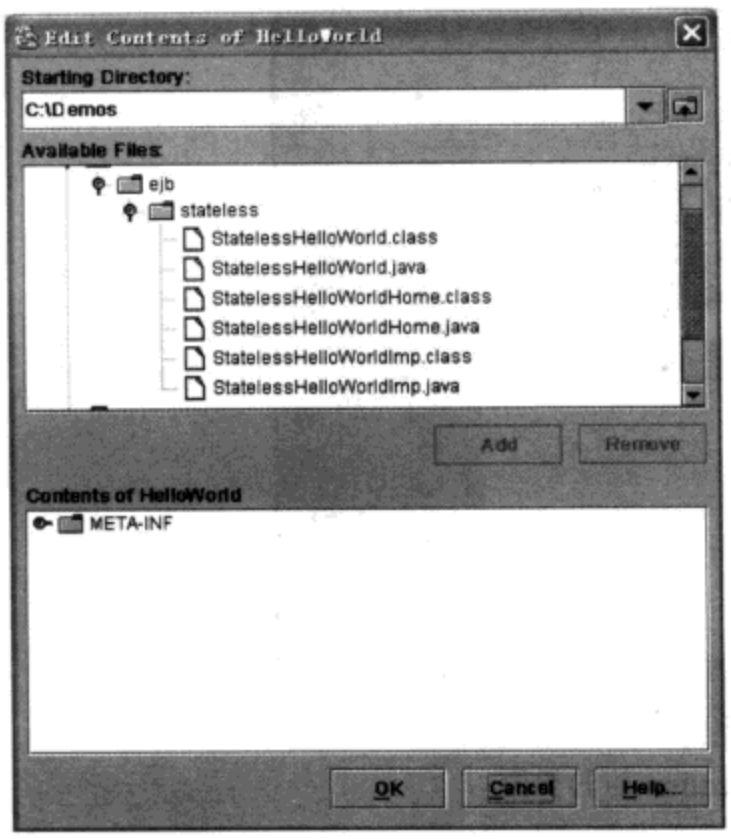


图 10-7 编辑 JAR 的内容

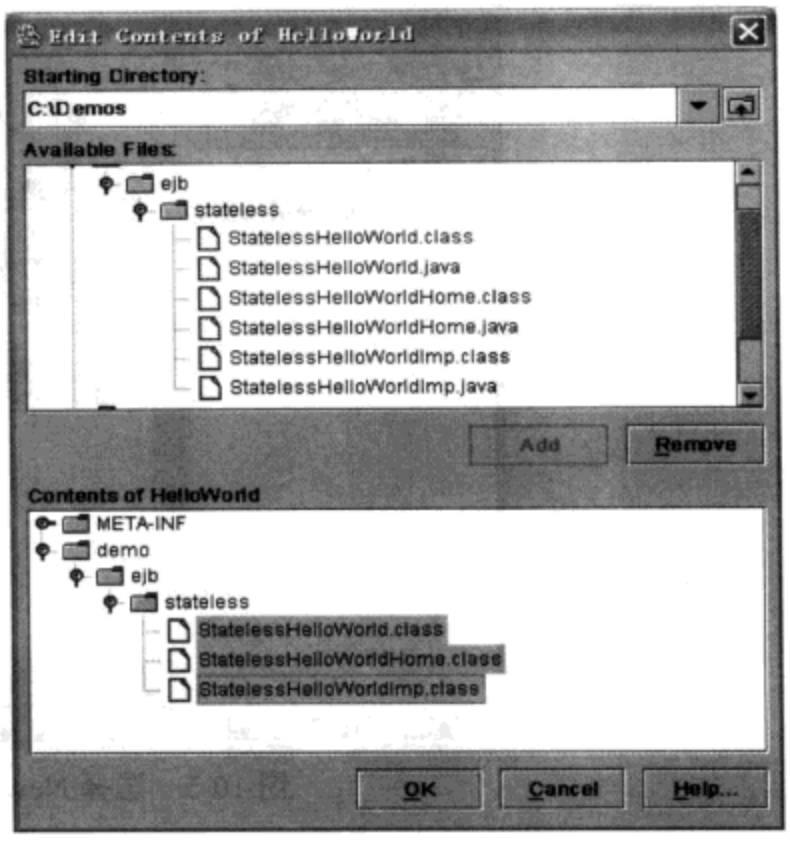


图 10-8 选择构成 EJB 的类文件

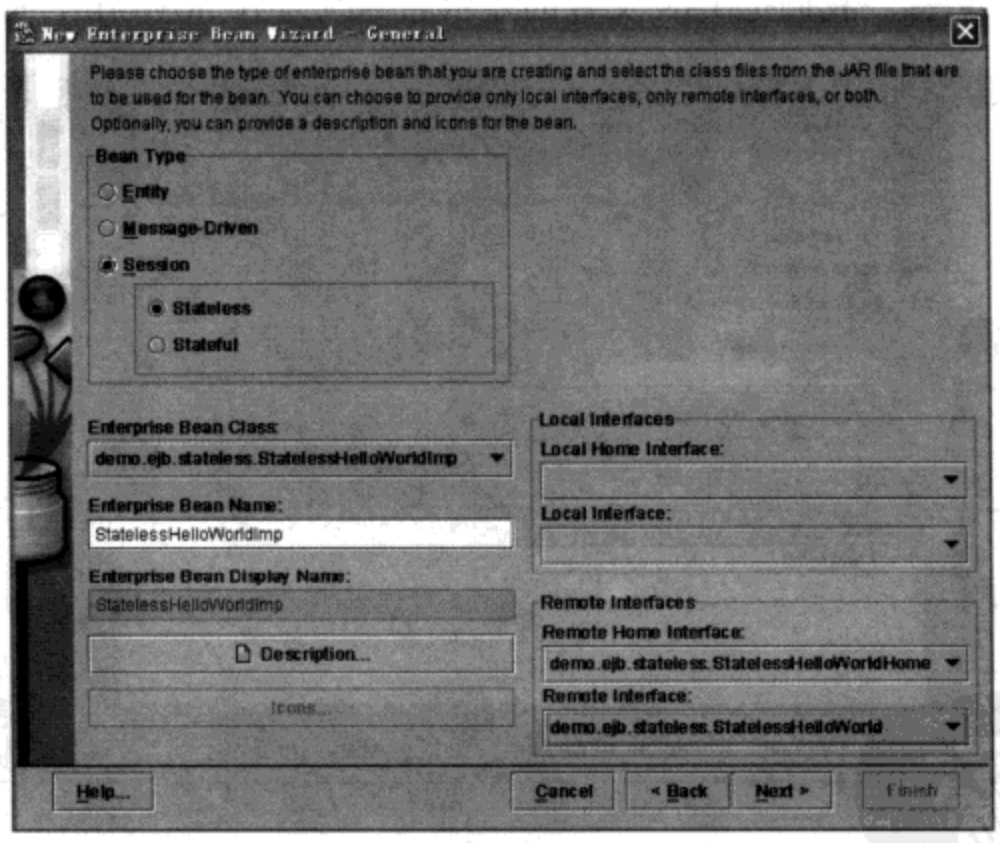


图 10-9 Enterprise Bean 的设置

图 10-10 用来设置 Bean 所需的事务支持。事务可以由 Bean 本身管理，也可以请求容器管理。我们的 Bean 不需要任何事务代码，所以这里选择由 Bean 本身管理。

接下来的页面对于 HelloWorld Bean 没有必要设置，所以直接单击 Finish 按钮。这样就实现了对 Bean 的部署。

接下来就要正式向 J2EE 服务器发布 Bean 了。要确保 EJB 服务器运行之后再部署。如果在 Deployment Tool 的窗口左侧树中 Servers 节点下没有出现 J2EE 服务器，则单击 File→Add Server 菜单项，以添加将要发布 Bean 的 J2EE 服务器，如图 10-11 所示。

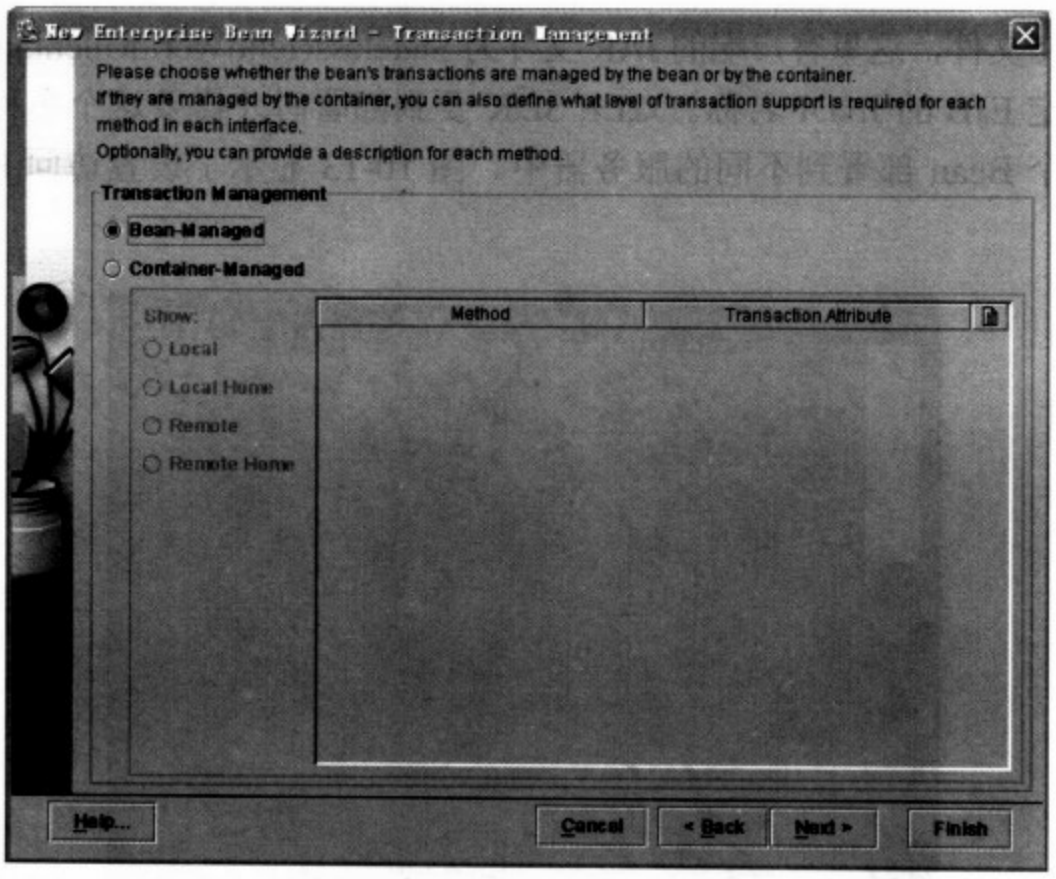


图 10-10　事务管理的设置

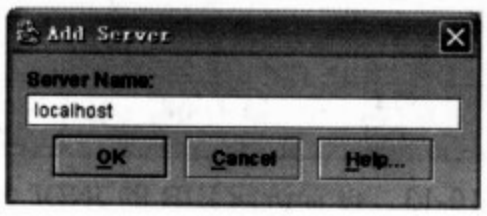


图 10-11　添加将要发布 Bean 的 J2EE 服务器

在服务器运行之后，选择 Tool→Deploy 菜单，出现如图 10-12 所示的对话框。

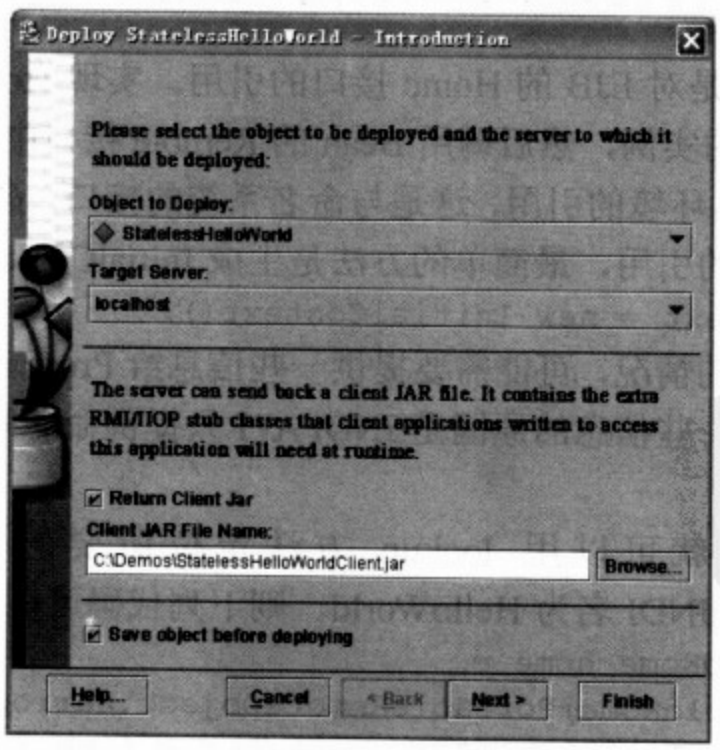


图 10-12　配置 StatelessHelloWorld

在此对话框中选择 Return Client Jar 选项。部署 EAR 文件或 JAR 文件时，通常又会生成许多类，服务器用其处理对 Bean 的网络访问。许多 EJB 服务器还生成客户端的 JAR 文件，其中包含客户机访问 EJB 时所需的类。J2EE SDK 需要客户端的 JAR 文件，所以一定要通过 Deployment Tool

生成客户端的 JAR 文件，这里客户端的 JAR 文件为 StatelessHelloWorldClient.jar。

最后，要确定 EJB 的 JNDI 名称。J2EE SDK 要到部署时才需要这个名称，因此可以用不同的名称将同一个 Bean 部署到不同的服务器中。图 10-13 显示了设置访问 EJB 的 JNDI 名称的对话框。

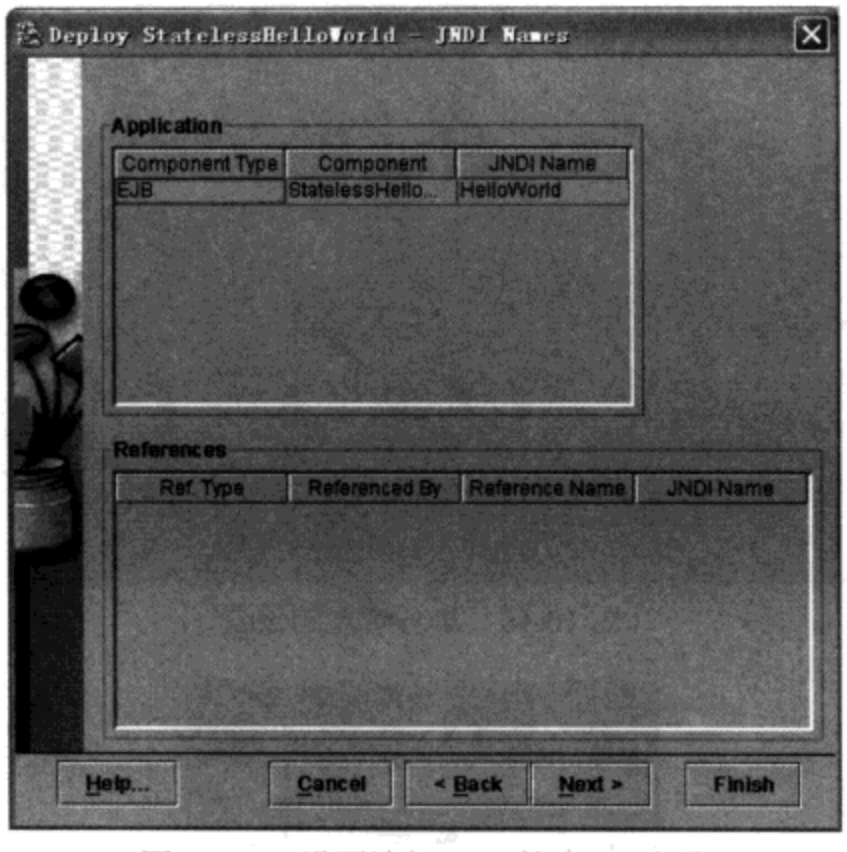


图 10-13　设置访问 EJB 的 JNDI 名称

单击 Finish 按钮以完成 Bean 的部署。

10.3.5　生成访问无状态 Session Bean 的客户端程序

使用 EJB 的惟一难点是对 EJB 的 Home 接口的引用。实现 Home 接口的引用后，就可以用 create()方法生成 Bean 的实例，然后调用 Bean 的 Remote 接口中的方法。

首先要实现 JNDI 命名环境的引用，这是与命名系统的接口，命名系统用于定位 EJB 和其他对象。要实现命名环境的引用，最简单的方法是生成 InitialContext 对象，例如：

```
Context namingContext = new InitialContext();
```

根据应用程序服务器的情况，可能需要提供一些信息给 Properties 对象或在运行客户机时定义系统属性。提供其他一些信息的原因是 JNDI API 只提供命名系统的接口，而不提供命名服务本身。

有了命名环境后，就可以用 lookup 方法定位所要的 EJB。例如，如果开发 StatelessHelloWorld Bean，JNDI 名为 HelloWorld，则下列代码可以定位 Bean 的 Home 接口：

```
StatelessHelloWorldHome home =
 (StatelessHelloWorldHome)PortableRemoteObject.narrow
 ( context..lookup("HelloWorld" ), StatelessHelloWorldHome.class );
```

注意在使用 EJB 时，不能用标准 Java 的强制类型转换运算来转换一个远程引用，而要用 PortableRemoteObject.narrow 方法。另外，不要定位 EJB 的 Remote 接口，而只需要定位 Home 接口，然后用 Home 接口中的 create 与 find 方法定位 Remote 接口。

下面是完整的客户端程序 StatelessHelloWorldClient.java 的代码：

```
package demo.ejb.stateless;

import java.util.*;
import javax.naming.*;
import javax.rmi.*;

public class StatelessHelloWorldClient
{
    public static void main(String[] args)
    {
        try
        {
            //实现 JNDI 命名环境的引用
            Context context = new InitialContext();

            //用 lookup 方法定位 EJB
            StatelessHelloWorldHome home =
                (StatelessHelloWorldHome)PortableRemoteObject.narrow
                (context.lookup("HelloWorld"),
                StatelessHelloWorldHome.class );
            //调用 home 接口的 create 方法得到 Session Bean 的引用
            StatelessHelloWorld session = (StatelessHelloWorld)home.create();
            //调用 Session Bean 的引用方法
            System.out.println(session.sayHello("Hello World!"));
            //销毁 Session Bean 的引用
            session.remove();
        }catch (Exception exc){
            exc.printStackTrace();
        }
    }
}
```

10.4　有状态 Session Bean 开发示例

本节我们要分析的是一个有状态 Session Bean 应用的例子。这是一个很简单的资金管理程序，它包含一个名为 StatefulFundManagerEJB 的有状态 Session EJB。该应用程序支持资金存入和提取操作，但数据仅在会话期间有效，即不能持久保存。该应用程序包含如下文件：

- Home 接口：StatefulFundManagerHome.java。
- 远程接口：StatefulFundManager.java。
- 会话 Bean 类：StatefulFundManagerEJB.java。

此外，本例还包含一个 InsufficientBalanceException.java 文件中定义的异常处理。

本例的 Session Bean 名称为 StatefulFundManagerEJB。和所有其他 Session Bean 一样，StatefulFundManagerEJB 满足以下要求：

- 实现 Session Bean 接口。
- 类定义成 public。
- 类不能定义成 abstract 或 final。

- 实现一个或多个 ejbCreate()方法。
- 实现业务方法。
- 包含一个公用的、不带参数的构造函数。
- 不能定义 finalize()方法。

一个 Session Bean 接口定义了 ejbRemove()、ejbActivate()、ejbPassivate()和 setSessionContext()方法。StatefulFundManagerEJB 不需要用到这些方法，但由于 Session Bean 接口声明了它们，因此必须要实现它们。在 StatefulFundManagerEJB 类中，这些方法都是空的。

10.4.1 StatefulFundManagerEJB 的主接口

和无状态 Session Bean 中一样，有状态 Session Bean 的主接口是从 EJBHome 接口扩展而来的。这里的主接口只声明了一个 create()方法，它不再是不带任何参数的了（当然也可以声明不带参数的 create()方法）。下面是 StatefulFundManagerHome.java 的主接口的代码：

```
package demo.ejb.stateful;

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

public interface StatefulFundManagerHome extends EJBHome
{
StatefulFundManager create(double amount) throws RemoteException, CreateException;
}
```

注意，由于这里的 create()方法带有一个参数，因此需要在实现类中编写 ejbCreate()方法的代码，这将在介绍实现类时讲述。

10.4.2 StatefulFundManagerEJB 的远程接口

和前面无状态 Session Bean 的例子一样，有状态 Session Bean 的远程接口也是从 EJBObject 接口扩展而来的，该远程接口定义了实现类需要实现的业务方法的声明。下面是远程接口 StatefulFundManager.java 的代码：

```
package demo.ejb.stateful;

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

public interface StatefulFundManager extends EJBObject
{
public void addFunds(double amount) throws RemoteException;

public void withdrawFunds(double amount) throws InsufficientBalanceException,
RemoteException;

public double getBalance() throws RemoteException;
}
```

10.4.3　StatefulFundManagerEJB 的实现类

注意一下 double 类型的变量。作为相对无状态 Session Bean 的改变，在这里使用这个变量是可行的。因为有状态 Session Bean 可以保存会话的状态，这里表现为变量的值得到了保持。这个变量保存客户账户的结余，其值（状态）由业务方法 addFunds()和 withdrawFunds()来修改，而 getBalance()业务方法返回变量的值。和无状态 Session Bean 一样的是，需要实现在 Session Bean 接口中声明的方法，只不过在这里这些方法都只是空实现。下面是实现类 StatefulFundManagerImp.java 的代码：

```
package demo.ejb.stateful;

import java.rmi.RemoteException;

import javax.ejb.EJBException;
import javax.ejb.Session Bean;
import javax.ejb.SessionContext;
import javax.ejb.CreateException;

public class StatefulFundManagerImp implements Session Bean
{
    double amount;

    public void setSessionContext(SessionContext arg0)
        throws EJBException, RemoteException
{
}

public void ejbRemove() throws EJBException, RemoteException
{
}

public void ejbActivate() throws EJBException, RemoteException
{
}

public void ejbPassivate() throws EJBException, RemoteException
{
}

public void ejbCreate(double amount) throws CreateException
{
    if (amount < 0)
    {
        throw (new CreateException("Invalid amount"));
    }
    else
    {
        this.amount = amount;
    }
```

```
}

public void addFunds(double amount)
{
    if (amount > 0)
    {
        this.amount += amount;
    }
}

public void withdrawFunds(double amount) throws InsufficientBalanceException
{
    if (this.amount < amount)
    {
        throw (new InsufficientBalanceException());
    }
    else
    {
        this.amount -= amount;
    }
}

public double getBalance()
{
    return amount;
}
```

前面主接口中声明了一个带有参数的 create()方法，在该实现类中有一个与之相对应的 ejbCreate()方法的实现。在 Bean 的生成过程中，容器将调用这个方法。在这里用它来初始化 amount 变量。

10.4.4 部署 EJB

部署 EJB 的过程和前面介绍的无状态 Session Bean 基本一样，只是在选择 Bean 类型时需要选择有状态 Session Bean，在此不再赘述。

10.4.5 生成访问有状态 Session Bean 的客户端程序

客户端程序 StatefulFundManagerTestClient.java 的代码如下所示：

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.text.*;

public class StatefulFundManagerTestClient extends JFrame implements
ActionListener
```

```
{
  JTextField amount = new JTextField(10);
  JButton addFunds = new JButton("Add Funds");
  JButton withdrawFunds = new JButton("Withdraw Funds");
  String msg = "Current account balance: ";
  JLabel status;
  StatefulFundManager manager;
  NumberFormat currencyFormatter;

  public StatefulFundManagerTestClient()
{
    super("Fund Manager");
  }

  public static void main(String[] args)
{
    new StatefulFundManagerTestClient().init();
  }

  public void init()
{
    buildGUI();

    addWindowListener(new WindowAdapter());
{
      public void windowClosing(WindowEvent evt)
      {
        System.exit(0);
      }
}

    addFunds.addActionListener(this);
    withdrawFunds.addActionListener(this);

    // 创建一个资金管理人
    createFundManager();
    // 将起始的余额设置为 0
    try
    {
      currencyFormatter = NumberFormat.getCurrencyInstance();
      String currencyOut = currencyFormatter.format(manager.getBalance());
      status.setText(msg + currencyOut);
    }catch (Exception re){
      re.printStackTrace();
    }
    pack();
    show();
  }

  public void buildGUI()
```

```
{

    GridBagLayout gl = new GridBagLayout();
    GridBagConstraints gc = new GridBagConstraints();
    Container container = getContentPane();
    container.setLayout(gl);

    gc.fill = GridBagConstraints.BOTH;
    JLabel label = new JLabel("Enter Amount");
    gl.setConstraints(label, gc);
    container.add(label);

    gc.gridwidth = GridBagConstraints.REMAINDER;
    gl.setConstraints(amount, gc);
    container.add(amount);

    gl.setConstraints(addFunds, gc);
    container.add(addFunds);
    gl.setConstraints(withdrawFunds, gc);
    container.add(withdrawFunds);

    status = new JLabel(msg);
    gl.setConstraints(status, gc);
    container.add(status);
  }

  public void actionPerformed(ActionEvent e)
{

    String str = amount.getText();

    if (str.equals(""))
    {
      return;
    }

    try
    {
      if (e.getSource() == addFunds)
    {

        // 调用 addFunds 方法
        manager.addFunds(Double.parseDouble(amount.getText()));

        // 调用 getBalance 方法
        String currencyOut = currencyFormatter.format(manager.getBalance());
        status.setText(msg + currencyOut);
      }

      if (e.getSource() == withdrawFunds)
```

```
    {

        // 调用 withdrawFunds 方法
        manager.withdrawFunds(Double.parseDouble(amount.getText()));

        // 调用 getBalance 方法
        String currencyOut = currencyFormatter.format(manager.getBalance());
        status.setText(msg + currencyOut);
      }

    }catch (Exception ex)
{
      ex.printStackTrace();
    }
  }

  public void createFundManager()
{
    try
    {
      Context initial = new InitialContext();
          Object objref = initial.lookup("MyStatefulFundManager");

      StatefulFundManagerHome home =
          (StatefulFundManagerHome) PortableRemoteObject.narrow(objref,
              StatefulFundManagerHome.class);

      manager = home.create(2000);

    }catch (Exception ex)
    {
      System.err.println("Caught an unexpected exception!");
      ex.printStackTrace();
    }
  }
}
```

这是一个图形化的应用程序，buildGUI 函数用来建立 GUI。程序结构还是比较简单的，读者需要注意的是如何调用 EJB 方法，并且注意有状态 Session Bean 保存了两次调用之间的状态。不过需要知道的是虽然有状态 Session Bean 保存了两次调用之间的状态，但是如果客户长时间不用 Bean 而 Bean 仍然处于激活状态，则会浪费许多资源，所以一般来说 EJB 容器在客户长时间不用 Bean 或服务器准备释放一些资源的时候将把 Bean 进行序列化，以使 Bean 进入序列化状态。序列化 Bean 的方法由具体的 EJB 容器厂商决定。一般来说，先序列化最早以前使用的 Bean。在序列化过程中，根据 EJB 规范容器负责调用 Bean 类的 ejbPassivate()方法。这样就使 Bean 编程者有机会在序列化 Bean 之前释放前面已分配的任何资源。当客户再次请求 Bean 时，容器负责激活序列化的 Bean。

10.5 Entity Bean 开发示例

Entity Bean 是两种主要 EJB（实体和会话）中的一种。Entity Bean 用来代表商业过程中处理的永久性的数据，例如：银行出纳员组件完成储蓄等商业过程，其中涉及的数据是银行账户数据。Entity Bean 用来代表底层的对象。最常用的是用 Entity Bean 代表关系库中的数据，它向 JDBC 或其他一些后端 API 经常访问的数据提供了一个面向对象的接口。一个简单的 Entity Bean 可以定义成代表数据库中表的一个记录，也就是每一个实例代表一个特殊的记录。更复杂的 Entity Bean 可以代表数据库中各表之间的关联视图。在 Entity Bean 中还可以考虑包含厂商的增强功能，如对象与关系之间映射的集成。通常用 Entity Bean 类代表一个数据库中的表比代表多个相关联的表更简单且更有效。反过来可以轻易地向 Entity Bean 类的定义中增加关联，这样可以最大地复用缓存并减小旧数据的表现。不仅如此，Entity Bean 提供了一个组件模型，可以让 EJB 开发人员将精力集中在 EJB 的商业逻辑上，而容器负责管理持续、事务和访问控制。

有两种基本的 Entity Bean：容器管理持久性的（CMP）和 Bean 管理持久性的（BMP）。容器使用 CMP 管理实体 Bean 的持久性。供应商工具用它来将实体字段映射到数据库，并且绝对没有数据库访问代码写入 EJB 类中。使用 BMP，Entity Bean 包含了数据库访问代码（通常是 JDBC 语句），负责读取其自身状态并将此状态写入数据库。BMP 实体对此有很大帮助，因为容器将提醒 Bean 何时需要更新状态或从数据库读取状态。容器还可以处理任何锁定或事务，因此数据库可以保持完整性。

10.5.1 Entity Bean 和 Session Bean 的比较

看起来 Session Bean 好像没什么用处，尤其对于数据驱动的应用程序。当然事实并不是这样。因为 Entity Bean（譬如说）代表底层数据库的一行，则 Entity Bean 实例和数据库记录间就是一对一的关系。因为多个客户端必须访问底层记录，这意味着不同于 Session Bean，客户端必须共享 Entity Bean。因为是共享的，所以 Entity Bean 不允许保存每个客户端的信息。而 Session Bean 允许保存客户端的状态信息，客户端和 Session Bean 实例间是一对一的关系。而 Entity Bean 允许保存记录的信息，Entity Bean 实例和记录间是一对一的关系。一个理想的情况是客户端通过 Session Bean 连接到服务器，然后 Session Bean 通过 Entity Bean 来访问数据库。这使得既可以保存客户端的信息，又可以保存数据库记录的信息。同时 Session Bean 也不能提供在相同或不同的 EJB 类调用间进行全局的事务控制。如果没有 Session Bean，应用程序开发者（客户端开发者）就必须理解 EJB 类的事务要求，并使用客户端的事务划分来提供事务控制。EJB 的主要好处就是应用程序开发者不需知道 EJB 类的事务需求。一个 Session Bean 可以代表一个商业操作并进行事务控制，而不需要客户端进行事务划分。

10.5.2 容器管理的持久性

对于 Bean 开发人员来说，容器管理的持久性 Bean 是最易于创建的，但对于 EJB 服务器来说却是最难以支持的。这是因为容器要自动处理所有用于使 Bean 的状态与数据库同步的逻辑。这就意味着 Bean 开发人员不需要编写任何数据访问逻辑，而 EJB 服务器应该自动负责所

有持续要求，这对于任何供应商都是一个很高的要求。大多数 EJB 供应商都支持关系数据库的自动持续，但支持级别却各不相同。某些供应商提供非常复杂的对象到关系映射，而某些供应商却非常有限。

Enterprise Bean 是一个完整的组件，它由至少两个接口和一个 Bean 实现类组成。下面先以一个 CMP 实体 Bean 的开发为例，来讲述 Entity Bean 开发的一般过程。

10.5.3　本地接口

要创建 CMP 实体 Bean 的新实例以便将数据插入到数据库中，必须调用 Bean 的本地接口上的 create()方法。本例中 Customer Bean 的本地接口由 CustomerHome 接口定义。CustomerHome 接口的定义如下所示：

```
public interface CustomerHome extends javax.ejb.EJBHome
{
public Customer create(Integer customerNumber)
                throws RemoteException,CreateException;

   public Customer create(Integer customerNumber, Name name)
            throws RemoteException,CreateException;

   public Customer findByPrimaryKey(Integer customerNumber)
                throws RemoteException, FinderException;

   public Enumeration findByZipCode(int zipCode)
                  throws RemoteException, FinderException;
}
```

以下是应用程序客户机如何使用本地接口创建一个新客户的示例：

```
InitialContext ctx = new InitialContext();
Object objHome = ctx.lookup("ejb/itso/Customer");

CustomerHome home = (CustomerHome) javax.rmi.PortableRemoteObject.narrow(
    objHome, CustomerHome.class); // 获取 CustomerHome 对象的一个引用

Name name = new Name("John", "W", "Smith");

Customer customer = home.create(new Integer(33), name);
```

Bean 的本地接口可以声明零个或多个 create()方法。在 Bean 类中，每个方法必须有相应的 ejbCreate()和 ejbPostCreate()方法。在运行时将链接这些 create 方法，因此在本地接口上调用 create() 方法时，容器会调用 Bean 类上的相应 ejbCreate()和 ejbPostCreate()方法。

当调用本地接口上的 create()方法时，容器会将 create()方法调用委托给 Bean 实例中匹配的 ejbCreate()方法。ejbCreate()方法用于在将记录插入数据库之前初始化实例状态。在本例中，它们初始化 customerID 和 Name 字段。ejbCreate()方法结束时（在 CMP 中，它们返回 null），容器将读取容器管理字段，并将一条新记录插入主键指向的 CUSTOMER 表。在本例中，customerID 映射成 CUSTOMER.ID 栏。

在 EJB 中，从技术上来说在将实体 Bean 的数据插入数据库（在 ejbCreate()方法期间发生）之前，该实体 Bean 并不存在。一旦插入了数据，实体 Bean 才真正存在，并且可以访问它自

己的主键和远程引用，而这在 ejbCreate()方法完成且数据进入数据库之前是不可能实现的。如果在被创建之后但在使用任何商业方法之前，Bean 需要访问它自己的主键或远程引用，可以在 ejbPostCreate()方法中执行此操作。ejbPostCreate()允许 Bean 在开始响应客户机请求之前，执行任何创建后处理。对于每一个 ejbCreate()方法，必须有一个匹配（匹配自变量）的 ejbPostCreate()方法。

本地接口中以 find 开头的方法都称作查找方法。这些方法用于根据传递的方法和自变量的名称，向 EJB 服务器查询特定实体 Bean。不幸的是，没有为查找方法定义标准查询语言，因此每个供应商都单独实现这些查找方法。在 CMP 实体 Bean 中，并没有用 Bean 类中的匹配方法实现查找方法；容器会在以供应商特定方式部署 Bean 时实现这些查找方法。部署人员使用供应商特有的工具来告诉容器某个特定的查找方法如何操作。某些供应商使用对象关系映射工具来定义查找方法的行为，而其他供应商只要求部署人员输入适当的 SQL 命令来完成这些操作。

EJB 有两种基本的查找方法：单一实体和多实体。单一实体查找方法返回匹配查找请求的一个特定实体 Bean 的远程引用。如果没有找到匹配的实体 Bean，该方法会抛出异常 ObjectNotFoundException。每个实体 Bean 必须定义方法名称为 findByPrimaryKey()的单一实体查找方法，此方法使用 Bean 的主键类型作为自变量（在以上示例中使用了 Integer 类型，它将 id 字段的 int 类型封装到 Bean 类中）。而多实体查找方法返回匹配查找请求的实体的一个集合（Enumeration 或 Collection 类型）。如果没有找到匹配的实体，多实体查找方法将返回一个空集（请注意，空集与空引用不同）。

10.5.4 远程接口

除了本地接口外，每个实体 Bean 必须定义一个远程接口。远程接口定义了实体 Bean 的商业方法逻辑。以下是 Customer Bean 的远程接口 Customer.java 的定义：

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Customer extends EJBObject
{

    public Name getName()
              throws RemoteException;

    public void setName(Name name)
              throws RemoteException;

    public Address getAddress()
              throws RemoteException;

    public void setAddress(Address address)
              throws RemoteException;

    public CreditCard getCreditCard()
              throws RemoteException;

    public void setCreditCard(CreditCard card)
              throws RemoteException;
```

```
}
```

以下是客户机应用程序如何使用远程接口与一个 Bean 进行交互的示例：

```
Customer customer = home.create(new Integer(33), name);  //获取该Bean的一个远程引用

// 获取该顾客的地址
Address addr = customer.getAddress();

// 改变邮政编码
addr.zip = "56777";

// 更新该顾客的地址
customer.setAddress(addr);
```

该远程接口中的商业方法被委托给 Bean 实例中相匹配的商业方法。在 Customer Bean 中，商业方法都是简单的读方法和写方法，但它们可以更复杂一些。换句话说，实体的商业方法并不仅限于读写数据，它们还可以执行其他任务和复杂的计算。

例如，如果客户有一个忠诚度程序，以用于奖赏老客户，那么该程序中可能会有根据驻留的累计数来升级会员资格的方法。请参见以下代码：

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Customer extends EJBObject
{
    public MembershipLevel addNights(int nights_stayed)
                        throws RemoteException;

    public MembershipLevel upgradeMembership()
                        throws RemoteException;

    public MembershipLevel getMembershipLevel()
                        throws RemoteException;

    ··· //从这里开始是一些简单的访问商业逻辑方法
}
```

其中的 addNights()和 upgradeMembership()方法比简单的读方法更为复杂，因为它们使用商业规则来更改会员资格的级别，这当然要比读写数据更为高级。

10.5.5　Bean 实现类

下面看看 Bean 的实现类，下面是实现类 CustomeBean.java 的代码：

```
import javax.ejb.EntityBean;

public class CustomerBean implements EntityBean
{
    int          customerID;
    Address      myAddress;
```

```
    Name          myName;
    CreditCard    myCreditCard;

    public Integer ejbCreate(Integer id)
{
        customerID = id.intValue();
        return null;
    }

    public void ejbPostCreate(Integer id)
{
    }

    public Customer ejbCreate(Integer id, Name name)
{
        myName = name;
        return ejbCreate(id);
    }

    public void ejbPostCreate(Integer id, Name name)
{
    }

    public Name getName()
{
        return myName;
    }

    public void setName(Name name)
{
        myName = name;
    }

    public Address getAddress()
{
        return myAddress;
    }

    public void setAddress(Address address)
{
        myAddress = address;
    }

    public CreditCard getCreditCard()
{
        return myCreditCard;
    }

    public void setCreditCard(CreditCard card)
{
```

```
        myCreditCard = card;
    }

    // 回调方法
    public void setEntityContext(EntityContext cntx)
{
    }

    public void unsetEntityContext()
{
    }

    public void ejbLoad()
{
    }

    public void ejbStore()
{
    }

    public void ejbActivate()
{
    }

    public void ejbPassivate()
{
    }

    public void ejbRemove()
{
    }
}
```

这是一个很好的示例，它说明了一个非常简单的 CMP 实体 Bean。请注意，在这个 Bean 中没有数据库访问逻辑，这是因为 EJB 供应商提供了将 CustomerBean 中的字段映射到数据库的工具。例如，CustomerBean 类可以映射到任何数据库，只要该数据库包含类似于 Bean 中字段的数据即可。在此情况下，Bean 的实例字段包括原语 int 和简单的一些从属对象（Name、Address 和 CreditCard），以及它们自己的属性。以下是这些从属对象的定义。

Name.java 的定义如下所示：

```
// Name 类
public class Name implements Serializable
{

    public String lastName, firstName, middleName;

    public Name(String lastName, String firstName,String middleName)
{
        this.lastName   = lastName;
        this.firstName  = firstName;
```

```
        this.middleName = middleName;
    }

    public Name()
{
}
}
```

Address.java 的代码如下所示：

```
// Address类
public class Address implements Serializable
{

    public String street, city, state, zip;

    public Address(String street, String city,String state, String zip)
{
        this.street = street;
        this.city   = city;
        this.state  = state;
        this.zip    = zip;
    }

    public Address()
{
}
}
```

CreditCard.java 的代码如下所示：

```
// CreditCard类
public class CreditCard implements Serializable
{

    public String number, type, name;
    public Date expDate;

    public CreditCard(String number, String type,String name, Date expDate)
{
        this.number  = number;
        this.type    = type;
        this.name    = name;
        this.expDate = expDate;
    }

    public CreditCard()
{
}
}
```

这些字段称作容器管理的字段，这是因为容器负责将它们的状态与数据库进行同步，即容器管理这些字段。容器管理的字段可以是任何原语数据类型或可串行化的类型。本例中使用原语 int（customerID）和可串行化对象（Address、Name、CreditCard）。要将从属对象映射

到一个数据库中，需要使用非常复杂的映射工具。在 Bean 中，并不是所有字段都是容器自动管理的字段；某些可能只是 Bean 过渡使用的纯实例字段。Bean 开发人员在部署描述信息中指出哪些字段是容器管理的，以便区别容器管理字段和纯实例字段。

容器管理的字段在数据库中必须有相应类型（RDBMS 中的列），它们之间的关系可以是直接映射或通过对象关系进行映射。例如，Customer Bean 可以映射到数据库中的 CUSTOMER 表，该表的定义如下：

```
CREATE TABLE CUSTOMER(
    id                INTEGER PRIMARY KEY,
    last_name         CHAR(30),
    first_name        CHAR(20),
    middle_name       CHAR(20),
    street            CHAR(50),
    city              CHAR(20),
    state             CHAR(2),
    zip               CHAR(9),
    credit_number     CHAR(20),
    credit_date       DATE,
    credit_name       CHAR(20),
    credit_type       CHAR(10)
);
```

使用容器管理的持续性，供应商必须使用一些专用工具以将 Bean 的容器管理字段映射到一个特定表中它们相应的列，在本例中是 CUSTOMER 表。一旦将 Bean 的字段映射到数据库，并且部署了 Customer Bean 之后，容器将负责在 CUSTOMER 表中创建记录、装入记录、更新记录和删除记录，以响应在 Customer Bean 的远程和本地接口上调用的方法。

容器管理的字段的子集（一个或多个）还将被标识成 Bean 的主键。主键可以是一个索引或指针，它指向数据库中组成 Bean 状态的惟一记录。在 Customer Bean 的情况中，id 字段是主键字段，它用于定位数据库中 Bean 的数据。原语单一字段主键表示成其相应的对象封装。例如，Customer Bean 的主键是 Bean 类中的原语 int，但对于 Bean 的客户机，它将自己表示成 java.lang.Integer 类型。由几个字段组成的主键叫做复合主键，它由 Bean 开发人员定义的特定类表示。主键在概念上类似于关系数据库中的主键（实际上当关系数据库用于持续时，它们通常是相同的）。

10.5.6　回调方法

Bean 实现类定义了与本地接口中的方法匹配的创建方法，以及与远程接口中的方法相匹配的商业方法。Bean 类还实现了一组回调方法，它们允许容器在 Bean 的生命周期中通知它要发生的事件。回调方法在 javax.ejb.EntityBean 接口中定义，此接口由所有实体 Bean 来实现，当然包括 CustomerBean 类。一个 EntityBean 接口有以下定义（请注意，Bean 类实现了这些方法）：

```
public interface javax.ejb.EntityBean
{
    public void setEntityContext();
    public void unsetEntityContext();
    public void ejbLoad();
```

```
    public void ejbStore();
    public void ejbActivate();
    public void ejbPassivate();
    public void ejbRemove();
}
```

其中的 setEntityContext()方法向 Bean 提供了与 EntityContext 容器的接口。EntityContext 接口包含了用于获取关于 Bean 在其中进行操作的上下文信息。EntityContext 接口用于访问有关调用程序的安全性信息，可用于确定当前事务的状态或强制事务的取消，或者用于获取 Bean 自身、其本地接口或其主键的引用。EntityContext 在实体 Bean 实例的生命周期中只设置一次，因此如果以后需要它的引用，则该引用应该放在 Bean 实例的一个字段中。

以上代码中的 Customer Bean 并没有使用 EntityContext，但它可以使用。例如，它可以使用 EntityContext 来确认调用者的会员资格是否是某个特定安全性角色。以下代码就是这样的一个示例，其中 EntityContext 用于确认调用者是否是 Manager，这是惟一可以将客户的信用卡类型设置成 WorldWide 卡（超级富翁的无限制卡）的角色（它会自动将持有该卡的客户标记成享有特别服务）。

```
import javax.ejb.EntityBean;
public class CustomerBean implements EntityBean
{

    int             customerID;
    Address         myAddress;
    Name            myName;
    CreditCard      myCreditCard;
    EntityContext   ejbContext;

    public void setEntityContext(EntityContext cntx)
    {
        ejbContext = cntx;
    }

    public void unsetEntityContext()
    {
        ejbContext = null;
    }

    public void setCreditCard(CreditCard card)
    {
        if (card.type.equals("WorldWide"))
            if (ejbContext.isCallerInRole("Manager"))
                myCreditCard = card;
            else
                throw new SecurityException();
        else
            myCreditCard = card;
    }

    public CreditCard getCreditCard()
```

```
    {
        return myCreditCard;
    }
    ...
}
```

其中的 unsetEntityContext()方法在 Bean 的生命周期结束时使用（在实例被清除出内存之前），以用于取消引用 EntityContext，并执行所有最终的清除工作。

在将实体 Bean 的状态与数据库进行同步时，会调用 CMP 实体中的 ejbLoad() 和 ejbStore()方法。容器用数据库中的状态刷新了 Bean 的容器管理字段之后，会立即调用 ejbLoad()方法。容器在将 Bean 的容器管理字段写到数据库之前，会调用 ejbStore()方法。在同步数据时，这些方法用于修改数据。当存储在数据库中的数据与 Bean 字段中使用的数据不同时，这些方法会经常用来进行同步数据。例如，这些方法可以用于在存储数据之前压缩数据，以及在从数据库检索数据时解压缩数据。

如果 EJB 容器不够完善以致于不能将从属对象映射成 CUSTOMER 表时，可以使用 Customer Bean 中的 ejbLoad()和 ejbStore()方法以将从属对象（Name、Address、CreditCard）转换成简单的 String 对象和原语类型。以下是如何修改 Bean 的一个示例：

```
import javax.ejb.EntityBean;

public class CustomerBean implements EntityBean
{

    //容器管理的字段
    int      customerID;
    String   lastName;
    String   firstName;
    String   middleName;
    ...

    // 非容器管理的字段
    Name         myName;
    Address      myAddress;
    CreditCard   myCreditCard;


    //商业方法
    public Name getName()
    {
        return myName;
    }
    public void setName(Name name)
    {
        myName = name;
    }
    ...

    public void ejbLoad()
    {
```

```
        if (myName == null)
            myName = new Name();

        myName.lastName = lastName;
        myName.firstName = firstName;
        myName.middleName = middleName;
        ...
    }

    public void ejbStore()
    {

        lastName   = myName.lastName;
        firstName  = myName.firstName;
        middleName = myName.middleName;
        ...
    }
}
```

在串行化 Bean 之后以及激活 Bean 之前，容器会分别在 Bean 上调用 ejbPassivate()和 ejbActivate()方法。实体 Bean 中的串行化表示解除 Bean 实例与其远程引用的关联，于是容器可以从内存中释放这个 Bean 或者重用它。它是容器使用的资源储备测量，以用于减少内存中的实例数量。如果某个 Bean 已经使用了一段时间，或者它是由容器执行的、用于使资源重用次数最大化的普通操作，那么可以串行化它。某些容器会从内存中释放 Bean，而其他容器则会为其他活动的远程引用重用实例。当要串行化（解除与远程引用的关联）或激活（与远程引用关联）Bean 时，ejbPassivate() 和 ejbActivate() 方法会向这个 Bean 发出通知。

10.5.7 Bean 管理的持久性

Bean 管理的持久性（BMP）负责将其状态与数据库同步，就像容器管理的持久性一样。Bean 使用数据库 API（通常是 JDBC）来读取其字段并将字段写入数据库，但容器会告诉它何时执行每个同步操作，并会自动管理 Bean 的事务。Bean 管理的持续性也可以让 Bean 开发人员灵活地执行对于容器来说太过于复杂的持续操作，或者使用容器不支持的数据源。例如，定制或旧的不支持的数据库。

在本节中，将把 Customer Bean 类修改成 BMP Bean，而不是 CMP Bean。这个修改根本不会影响远程或本地接口。实际上，我们不会直接修改原始 Customer Bean 类，而是通过扩展 Bean 并重载其中适当的方法来将它更改成 Bean 管理的持续。以下是一个类的定义，这个类将扩展 Customer Bean 类，以使它成为 BMP 实体。大多数情况下，不必扩展 Bean 以使它成为 BMP，只要直接将 Bean 实现成 BMP 即可。在这里执行这种策略（扩展 CMP Bean）是出于两个原因：它允许 Bean 可以是 CMP 或 BMP；它可以很方便地减少显示所需要的全部代码。以下是将要添加的 BMP 类的定义：

```
public class CustomerBean_BMP extends CustomerBean
{
    public void ejbLoad()
    {
```

```
        //重载的实现代码
    }
    public void ejbStore()
    {
        // 重载的实现代码
    }
    public void ejbCreate()
    {
        // 重载的实现代码
    }
    public void ejbRemove()
    {
        // 重载的实现代码
    }
    private Connection getConnection()
    {
        //新的帮助方法
    }
}
```

对于 BMP Bean 来说，它与 CMP Bean 的差别在于，容器和 Bean 使用 ejbLoad()和 ejbStore()方法的方式不同。在 BMP 中，ejbLoad()和 ejbStore()方法分别包含了用于从数据库读取 Bean 的数据和将更改写入数据库的代码。当 EJB 服务器决定要读写数据时，会自动在 Bean 上调用这些方法。CustomerBean_BMP Bean 管理它自己的持续。换句话说，ejbLoad()和 ejbStore()方法必须包括数据库的访问逻辑。只有这样，当 EJB 告诉 Bean 要装入和存储数据时，它才能完成任务。而容器会在适当的时候自动执行 ejbLoad()和 ejbStore()方法。

通常在事务开始时，也就是在容器将商业方法委托给 Bean 之前，容器会调用 ejbLoad()方法。以下代码显示了如何使用 JDBC 来实现 ejbLoad()方法：

```
import java.sql.Connection;

public class CustomerBean_BMP extends CustomerBean
{

public void ejbLoad()
{
    Connection con;
    try
    {
            Integer primaryKey = (Integer)ejbContext.getPrimaryKey();
        con = this.getConnection();
           Statement sqlStmt = con.createStatement("SELECT * FROM Customer
           " +" WHERE customerID = " +primaryKey.intValue());
          ResultSet results = sqlStmt.executeQuery();
          if (results.next())
          {
          // 从这个 Customer 表中获取名称的信息
             myName = new Name();
             myName.first = results.getString("FIRST_NAME");
             myName.last = results.getString("LAST_NAME");
```

```
                myName.middle = results.getString("MIDDLE_NAME");
                //从这个 Customer 表中获取地址信息
                myAddress = new Address();
                myAddress.street = results.getString("STREET");
                myAddress.city = results.getString("CITY");
                myAddress.state = results.getString("STATE");
                myAddress.zip = results.getInt("ZIP");
                // 从这个 Customer 表中获取信用卡信息
                myCreditCard = new CreditCard();
                myCreditCard.number = results.getString("CREDIT_NUMBER");
                myCreditCard.expDate = results.getString("CREDIT_DATE");
                myCreditCard.type = results.getString("CREDIT_TYPE");
                myAddress.name = results.getInt("CREDIT_NAME");
            }
        } catch (SQLException sqle)
{
        throw new EJBException(sqle);
        }
        finally
{
        if (con!=null)
          con.close();
        }
    }
}
```

在 ejbLoad()方法中，使用对 Bean 的 EntityContext 的 ejbContext()引用来获取实例的主键，这种方法要确保对数据库使用了正确的索引。显然，CustomerBean_BMP 需要使用前面所述的继承的 setEntityContext()和 unsetEntityContext()方法。

在事务结束时，也就是在容器试图将所有更改提交给数据库之前，容器会对 Bean 调用 ejbStore()方法。

```
import java.sql.Connection;

public class CustomerBean_BMP extends CustomerBean
{

  public void ejbLoad()
{
    ... //从数据库中读取数据
  }

  public void ejbStore()
{
    Connection con;
    try
    {
      Integer primaryKey = (Integer)ejbContext.getPrimaryKey();
      con = this.getConnection();
      PreparedStatement sqlPrep = con.prepareStatement(
        "UPDATE customer set " +
```

```
      "last_name = ?, first_name = ?, middle_name = ?, " +
      "street = ?, city = ?, state = ?, zip = ?, " +
      "card_number = ?, card_date = ?, " +
      "card_name = ?, card_name = ?, " +
      "WHERE id = ?"
    );
    sqlPrep.setString(1,myName.last);
    sqlPrep.setString(2,myName.first);
    sqlPrep.setString(3,myName.middle);
    sqlPrep.setString(4,myAddress.street);
    sqlPrep.setString(5,myAddress.city);
    sqlPrep.setString(6,myAddress.state);
    sqlPrep.setString(7,myAddress.zip);
    sqlPrep.setInt(8, myCreditCard.number);
    sqlPrep.setString(9, myCreditCard.expDate);
    sqlPrep.setString(10, myCreditCard.type);
    sqlPrep.setString(11, myCreditCard.name);
    sqlPrep.setInt(12,primaryKey.intValue());
    sqlPrep.executeUpdate();
  }
  catch (SQLException sqle)
  {
    throw new EJBException(sqle);
  }
  finally
  {
      if (con!=null)
        con.close();
    }
  }
}
```

在 ejbLoad()和 ejbStore()方法中，Bean 都会使用 JDBC 将其自己的状态与数据库进行同步。如果仔细研究过代码，也许会发现 Bean 从神秘的 this.getConnection()方法调用中获取它的一个数据库连接，而此方法还尚未实现。getConnection()方法并不是标准的 EJB 方法，它只是一个私有的辅助方法，在这里实现它是为了隐藏获取数据库连接的方法。以下代码是 getConnection() 方法的定义：

```
import java.sql.Connection;

public class CustomerBean_BMP extends CustomerBean
{

  public void ejbLoad()
  {
    ... //从数据库中读到数据
  }

  public void ejbStore()
  {
    ... //向数据库中写数据
```

```
    }

    private Connection getConnection() throws SQLException
  {

      InitialContext jndiContext = new InitialContext();
      DataSource source= (DataSource)jndiContext.lookup
      ("java:comp/env/jdbc/myDatabase");
      return source.getConnection();
    }
  }
```

在这里通过使用默认 JNDI 的上下文（称作 JNDI 环境命名上下文，即 ENC），可以从容器中获取数据库的连接。ENC 通过标准连接库的 javax.sql.DataSource 类型，以提供对事务和用 JDBC 连接的访问权。

在 BMP 中，容器调用 ejbLoad()和 ejbStore()方法使 Bean 实例与数据库中的数据进行同步。为了将实体插入数据库或除去数据库中的实体，可以使用类似的数据库访问来实现 ejbCreate()和 ejbRemove()方法。ejbCreate()方法将一个新记录插入到数据库中，而 ejbRemove()方法从数据库中删除实体的数据。容器会调用 BMP 实体的 ejbCreate()方法和 ejbRemove()方法，以响应对本地和远程接口中它们相应方法的调用。这些方法的实现代码如下所示：

```
public void ejbCreate(Integer id)
{
    this.customerID = id.intValue();
    Connection con;
    try
    {
      con = this.getConnection();
      Statement sqlStmt = con.createStatement("INSERT INTO customer id VALUES
("  + customerID + ")");
      sqlStmt.executeUpdate();
      return id;
    } catch(SQLException sqle)
    {
      throw new EJBException(sqle);
    }finally
    {
      if (con!=null)
        con.close();
    }
  }

  public void ejbRemove()
{
    Integer primaryKey = (Integer)ejbContext.getPrimaryKey();
    Connection con;
    try
    {
      con = this.getConnection();
      Statement sqlStmt = con.createStatement("DELETE FROM customer WHERE id
```

```
= "primaryKey.intValue());
          sqlStmt.executeUpdate();
        }
        catch(SQLException sqle)
        {
          throw new EJBException(sqle);
        }
        finally
        {
          if (con!=null)
            con.close();
        }
     }
```

在 BMP 中，Bean 类负责实现在本地接口中定义的查找方法。对于本地接口中定义的每个查找方法，在 Bean 类中必须有相应的 ejbFind()方法。ejbFind()方法用于定位数据库中相应的 Bean 记录，并将它们的主键返回给容器。然后容器将主键转换成 Bean 的一个引用，并将它们返回给客户机。以下是 CustomerBean_BMP 类中 ejbFindByPrimaryKey()方法实现的示例，它对应于 CustomerHome 接口中定义的 findByPrimaryKey()方法，代码如下所示：

```
    public Integer ejbFindByPrimaryKey(Integer primaryKey)
                throws ObjectNotFoundException
    {
        Connection con;
        try
        {
            con = this.getConnection();
            Statement sqlStmt = con.createStatement("SELECT * FROM Customer " +
                            " WHERE customerID = " +
                            primaryKey.intValue());

            ResultSet results = sqlStmt.executeQuery();
            if (results.next())
                return primaryKey;
            else
                throw ObjectNotFoundException();
        }
        catch (SQLException sqle)
        {
            throw new EJBException(sqle);
        }
        finally
        {
            if (con!=null)
            con.close();
        }
}
```

以上单一实体查找方法会返回一个主键，如果没有找到匹配的记录，则会抛出一个异常 ObjectNotFoundException。而多实体查找方法会返回一个主键的集合（java.util.Enumeration 或 java.util.Collection），容器将这个主键集合转换成远程引用的集合，然后将此集合返回给客户机。以下是如何在 CustomerBean_BMP 类中实现多实体 ejbFindByZipCode()方法的示例，该方法对应于 CustomerHome 接口中定义的 findByZipCode()方法，代码如下所示：

```
public Enumeration ejbFindByZipCode(int zipCode)
{
    Connection con;
    try
    {
        con = this.getConnection();
        Statement sqlStmt =con.createStatement("SELECT id FROM Customer " +
                        " WHERE zip = " +zipCode);

        ResultSet results = sqlStmt.executeQuery();
        Vector keys = new Vector();
        while(results.next()){
            int id = results.getInt("id");
            keys.addElement(new Integer(id));
        }
        return keys.elements();
    }
    catch (SQLException sqle)
    {
        throw new EJBException(sqle);
    }
    finally
    {
        if (con!=null)
        con.close();
    }
}
```

如果没有找到相匹配的 Bean 记录，则该方法会将一个空集合返回给容器，然后容器将空集合返回给客户机。

实现了所有这些方法并对 Bean 的部署描述信息进行了一些较小更改之后，就可以将 CustomerBean_BMP 类部署成一个 BMP 实体了。

10.6 Java 消息服务和消息驱动 Bean

10.6.1 消息概述

当两个软件组件进行通信时，通常是一个对象调用另一个对象的方法。但是也可以把这种方法调用看成是一个对象向另一个对象发送消息。例如，调用 Person 对象的 getName 方法

时，相当于向 Person 对象发送一个消息。一般说来，系统设计师喜欢考虑对象之间的消息，而方法调用只不过是实现消息的一种方式而已。

这种方法调用适合小规模的情况，对分布式系统也同样适用。当处理大系统时，方法调用就开始出现问题了。通常需要将两个具有完全不同时间概念的组件连接起来。例如，一个系统可能是交互式 GUI 应用程序，它需要有立即响应；而另一个系统可能是大型批处理系统，并不能立即响应。当 GUI 应用程序向批处理系统发送数据时，不可能一直都在等待响应。这时可以通过消息将数据发往批处理系统，然后 GUI 应用程序干自己的事。等批处理程序处理完毕后同样发送消息给 GUI 应用程序，告知数据已经处理完毕并将结果返回。

消息是连接系统组件的最常用方法，由此出现了许多面向消息的中间件（MOM，Message-Oriented Middleware）。消息的最大魅力在于提供了客户机与服务器的松散耦合。而通过方法调用交互的组件显然耦合更紧，对时间也更为敏感。

10.6.2　消息驱动 Bean

Enterprise JavaBeans（EJB）1.1 版本中定义了两种 Bean 类型：Session Bean 和 Entity Bean。客户端对象可以同步调用 EJB 1.1 的这两种组件的方法。然而，为了继承面向对象的中间件（Message Oriented Middleware，MOM）和 Java 消息服务（Java Message Service，JMS）的优点的需要，EJB 框架中也相应地加入异步的消息通信机制。所以，在 EJB 2.0 中就定义了第三种组件类型——Message Driven Bean（消息驱动组件）。

Message Driven Bean 兼备 EJB 和 JMS 的功能。当然，如果想要使用消息操作技术，那么只使用 JMS 就行了。但是新的消息驱动组件 Message Driven Bean 提供了消息通信的新的可能性。那么，这些组件如何整合到一个应用程序服务器框架中？它们的功能又如何扩展了过去 JMS 服务器的使用范围？请看下文！

10.6.3　EJB 和 JMS

前面已经说过了，EJB 1.1 为开发人员定义了两种企业级组件类型：Session Bean 和 Entity Bean。Session Bean 通常实现一些商业逻辑并且不能在多客户端共用。Entity Bean 描述一个实体的面向对象的概念，而这个实体往往存在于像数据库那样固定的存储容器中。在这两种组件模型中，使用本地的或远程的接口来简化客户端的交互作用。按照它们的定义，这种交互作用是严格同步的。举例来说，通过一个方法调用把一个请求发送给组件，然后服务器对象返回一个响应。

然而，在企业版应用程序的范围中，也经常需要异步的消息传递。比方说，一个客户可能想发给服务器一条信息，但是并不需要或者不想要服务器做出应答，这时客户端就没有必要等待服务器对象来处理请求。对于客户端应用程序来说，在确保消息最终能够到达服务器并被正常处理的前提下，提交一条消息然后继续处理本身的事务，将会在很大程度上提高通信的效率。

消息服务是一种在分布式应用程序之间提供消息传递服务的软件，它具有可靠、异步、宽松结合、语言中立、平台中立的特点，而且通常是可配置的。实现原理是：对发送者和接收者之间传递的消息进行封装，并在分布式消息客户程序结合的位置加上一个软件处理层。消息服务为消息的客户程序提供了一个接口，这个接口隔离了底层的消息服务，使得各种不

同的客户程序能够通过一个友好的编程接口方便地进行通信。

Java 消息服务（Java Message Service，JMS）是一个 Java API，它定义了消息的客户程序如何以一种标准化的形式与底层的消息服务提供者进行交互。JMS 提供了一种接口，底层消息服务提供者通过该接口向客户程序提供 JMS 消息服务。JMS 提供了点对点（Point-to-Point）消息模式和发布-订阅（Publish-Subscribe）消息模式。点对点消息模式通过一个消息队列实现，消息的生产者向队列写入消息，而消息的消费者从队列提取消息。发布-订阅消息模式通过一个话题（Topic）节点构成的层次结构实现，消息的生产者向这个层次结构发布消息，而消息的消费者向这个结构订阅消息。

点对点消息模式具有如下特点：

- 每一个消息只有一个消费者。
- 消息的接收者和发送者之间不存在时间上的依赖关系。不论发送者发送消息时接收者是否在运行，接收者都可以提取信息。
- 接收者对于成功处理的消息给出回执。

而发布-订阅消息模式具有如下特点：

- 每一个消息可以有多个消费者。
- 向某个话题订阅的客户程序只能收到那些在它订阅之后发布的消息。为了接收到消息，订阅者必须保持活动状态。因此，发布者和订阅者之间存在时间上的依赖关系。

JMS API 在一定程度上放宽了对这种依赖关系的要求，它允许创建持久性订阅（Durable Subscription）。有了持久性订阅，当订阅者不活动时发送的消息也能接收到。

EJB 2.0 规范定义了一种新的 EJB 类型，即消息驱动的 EJB（Message-Driven EJB，简称 MDB），它能够以 EJB 的形式实现 JMS 消息的接收者。消息驱动的 EJB 实现一组新的接口，这组接口使得 EJB 能够异步地接收和处理 JMS 消息生产者发送到队列或话题的消息。EJB 客户程序的构造方式与普通 JMS 消息生产者的构造方式完全一样，也就是说，JMS 消息生产者不必知道消息的消费者是一个 EJB。

相对于会话 Bean 和实体 Bean 而言，消息驱动的 Bean 最大的特点是客户程序不通过接口访问 Bean，而且消息驱动的 Bean 只有一个 Bean 类。从某些方面看，消息驱动的 Bean 类似于无状态 Session Bean：

- 消息驱动的 Bean 不为特定的客户保留数据或对话状态。
- 一个消息驱动 Bean 的所有实例都是等价的，这使得容器能够把消息指派给任意一个消息驱动 Bean 的实例。容器能够建立消息驱动 Bean 的缓冲池，以实现消息的并发处理。
- 一个消息驱动的 Bean 能够处理来自多个客户程序的消息。

消息驱动 Bean 的实例变量可以在处理客户消息期间包含一些状态信息，例如 JMS 连接、打开的数据库连接、对 EJB 对象的引用。当接收到一个消息时，容器调用消息驱动 Bean 的 onMessage()方法来处理消息。onMessage()方法通常把消息强制转换成为五种 JMS 消息类型之一，然后按照应用的业务逻辑的要求处理消息。

传递给消息驱动 Bean 的消息可能处于一个事务之内，这时 onMessage()方法内的所有操作都属于该事务的一部分。如果消息处理结果被拒绝，则系统将再次投递该消息。哪些时候应

该使用消息驱动的 Bean 呢？会话 Bean 和实体 Bean 能够发送 JMS 消息，并且能够同步接收消息，但不能异步接收。一些时候，为防止过多地占用服务器资源，在服务器端的组件中，我们要尽量避免阻塞，这时我们可以用消息驱动的 Bean 来异步接收消息。

10.6.4　MDB 体系结构

图 10-14 描述了消息驱动的 Bean 组件的基本体系结构。

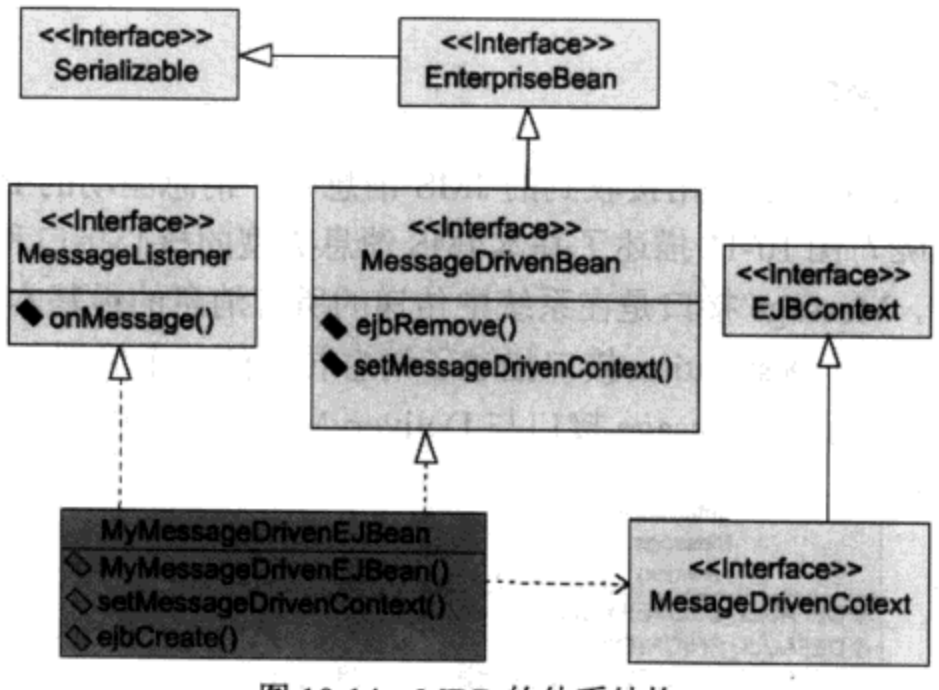


图 10-14　MDB 的体系结构

在图中，位于顶端的是 javax.ejb.EnterpriseBean 接口，它是所有 EJB 的基础接口。EnterpriseBean 接口派生出了 javax.ejb.MessageDrivenBean 接口，所有消息驱动的 EJB 类必须实现 javax.ejb.MessageDrivenBean 接口。此外，消息驱动的 Bean 还必须实现的接口是 javax.jms.MessageListener。一个公用的、非最终的、非抽象的消息驱动的 EJB，如图 10-14 显示的 MyMessageDrivenEJBean，必须同时实现 MessageListener 接口和 MessageDrivenBean 接口。消息驱动的 EJB 与其他类型的 EJB 不同，它们不把业务方法导出给客户程序，它们关心的只是遵从 EJB 容器的接口要求。由于这个原因，消息驱动的 Bean 必须有一个不需要参数的公用构造方法（ejbCreate()方法），而且不应该实现 finalize()方法。

在消息驱动的 Bean 中，setMessageDrivenContext()方法用来把一个 MessageDrivenContext 的对象实例传递给 EJB，它是 MessageDrivenBean 接口定义中容器调用的第一个方法。MessageDrivenContext 对象封装了一个 EJB 消息驱动容器上下文的接口，它支持消息驱动的 EJB 实例访问容器提供的运行时消息驱动上下文。

对于消息驱动的 EJB 来说，关键之一是要实现一个没有参数的 ejbCreate()方法。当 EJB 容器准备创建消息驱动 EJB 的实例时，它将调用这个方法。容器之所以决定创建某个 EJB 的实例，可能是因为它要构造一个 Bean 实例的缓冲池，也可能是因为它接收到了客户的请求。这个 ejbCreate()方法和其他 Bean 上的 EJB 构造方法类似，属于 EJB 实现的一种特殊的构造函数或初始化方法。

当 EJB 容器准备不让 Bean 实例继续处理客户程序的请求时，它就会调用消息驱动 Bean 的 ejbRemove()方法。何时在消息驱动的 Bean 上调用 ejbRemove()方法由 EJB 容器单独决定，而不受 EJB 客户程序的任何约束。应当注意的是，容器并不保证一定调用 ejbRemove()方法。

在正常操作时，容器会调用 ejbRemove()方法。但是，当消息驱动的 Bean 向容器抛出了系统异常时，就不能保证 ejbRemove()方法一定会被调用。由于这个原因，Bean 开发者必须按时检查和清除 Bean 分配的所有资源。

对于 Bean 开发者来说，最重要的任务也许就是实现 onMessage()方法。当一个异步消息必须由 Bean 实例处理时，容器将调用 onMessage()方法。onMessage()方法的参数是一个普通的 javax.jms.Message 的实例，消息驱动的 EJB 实例从这个 Message 的实例中提取待处理的数据以完成消息处理。

10.6.5 JMS 消息接口

那么，在 onMessage()方法调用接收到的 JMS 消息中，消息驱动的 Bean 如何提取信息以及可以提取哪些信息呢？图 10-15 描述了基本 JMS 消息类型的核心接口和概念。在一个以 JMS 为基础的消息系统中，Message 接口是在系统中传递的所有消息的最基本的接口（或称之为根接口，Root Interface），而 Destination 接口描述了消息传递的一个终端。类似地，消息有一个传递模式，图 10-15 还显示了 Message 接口与 DeliveryMode 接口的概念上的关系。

图 10-15 JMS 消息结果

JMS 消息的头信息可以通过一组标准的方法来设置或提取，这组标准方法的名字为 getJMSXXX()或 setJMSXXX()形式（下面分别称之为 get 方法和 set 方法）。其中 XXX 是消息头信息中的属性名字，例如 getJMSDeliveryMode()方法。在 Message 接口中，通过 get 方法和 set 方法操作的标准头信息的属性包括惟一的消息 ID、时间戳（Timestamp）、答复和目标地址、

消息传递模式、消息类型以及消息的优先级。

在 JMS 消息中，JMS 容器提供者特有的属性可以通过 getXXXProperty()方法提取，或通过 setXXXProperty()方法设置。其中 XXX 表示属性的类型，例如 Byte 就使用方法 getByteProperty(java.lang.String name)。每一个属性有一个通过 String 对象指定的名字和相应的值。其中名字以 JMSX 前缀开头的属性作为标准 JMS 属性保留。

与消息正文数据（或称之为消息体，与消息头相对而言）的 5 种类型对应，5 种消息类型扩展了 Message 接口，如图 10-16 所示。Byte 数据由 BytesMessage 封装，Serializable 对象由 ObjectMessage 封装，String 消息由 TextMessage 封装，键-值对由 MapMessage 封装，I/O 流由 StreamMessage 封装。

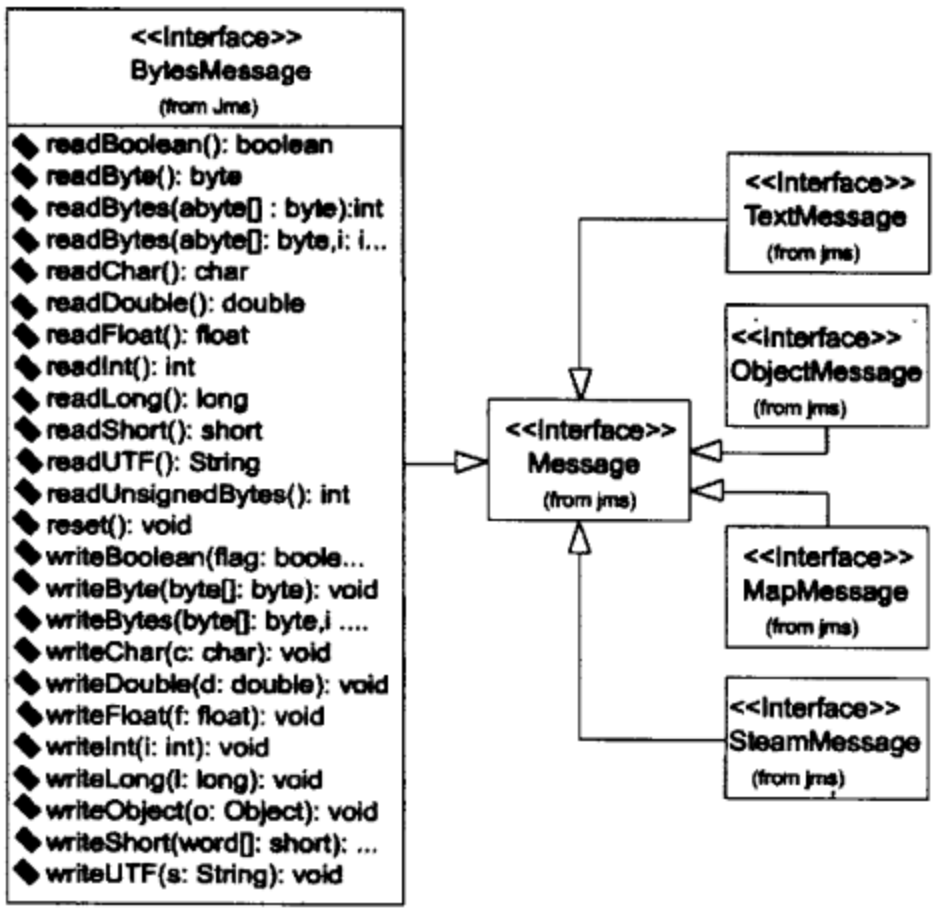


图 10-16　Message 接口的 5 种扩展类型

这些派生消息类型上的方法为特定类型的消息正文定义了 get 和 set 操作，而在 Message 基础接口内，有一个通用的 clearBody()方法。这个方法清除消息的正文，并把它置入只写模式。clearBody()方法只清除消息正文，不清除消息头或属性。如果消息正文是只读的，在调用该消息后，消息正文的状态将和新消息的空白正文状态一样。

10.6.6　MDB 客户程序接口

消息驱动 Bean 的客户程序并不知道接收端实际处理消息的将是一个 EJB。事实上，消息驱动 Bean 的客户程序的构造方法与普通 JMS 客户程序的构造方法完全一样。

JMS 的核心体系如图 10-17 所示。从图中可以看出，JMS 的 ConnectionFactory（连接工厂）的初始上下文通过 Java 命名和目录接口（Java Naming and Directory Interface，JNDI）创建。随后，连接工厂将用来创建与 JMS 服务提供者的连接。有了 JMS 连接，就可以获得消息生产者和消息消费者之间的会话（Session）。实际上，消息驱动 Bean 的客户程序就是消息的生产

者，它发送的消息将由消息驱动的 Bean（即消息消费者）来接收。

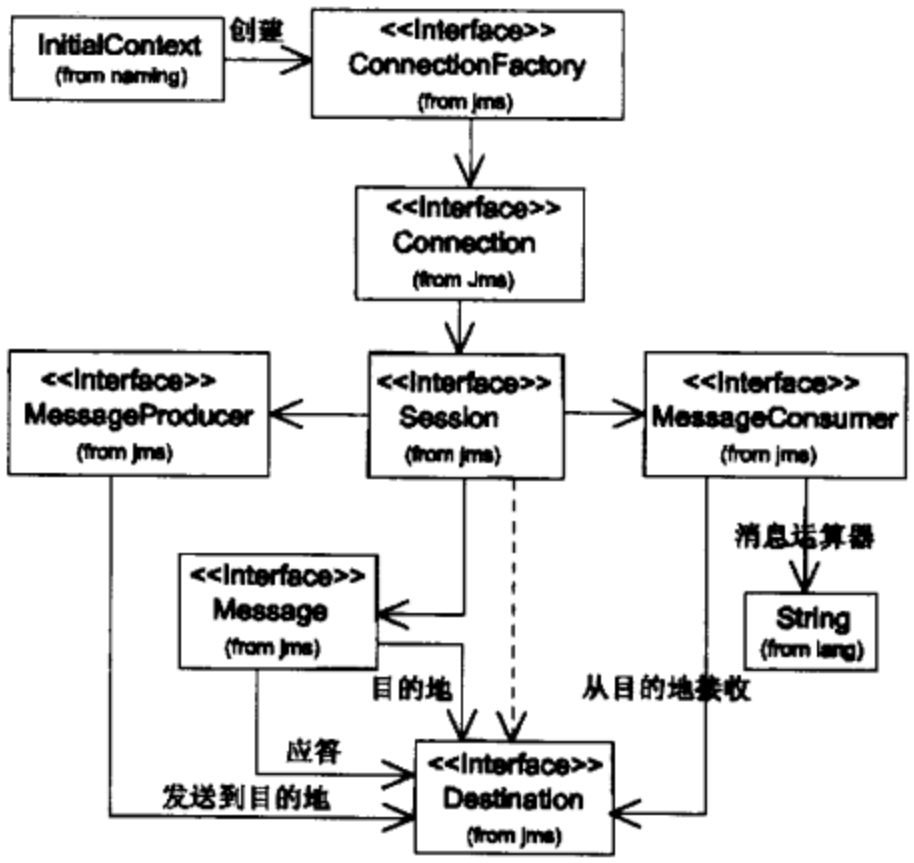


图 10-17　JMS 的核心体系

10.6.7　点对点消息队列模式

消息队列体系实际上是核心 JMS 体系的一个扩展，特别是它加入了对消息队列功能的支持。连接工厂、连接、会话、消息生产者、消息消费者等都用点对点消息队列形式对接口进行了扩展。

JMS 客户程序利用 JNDI 获得一个 QueueConnectionFactory 对象的引用。随后，用 QueueConnectionFactory.createQueueConnection()方法来创建一个 QueueConnection 对象的实例。调用 createQueueConnection()方法时可以指定一个用户名字和密码，或者使用该方法不带参数的版本，此时假定使用默认用户身份。

QueueConnection 接口是 Connection 接口的一种子类型，它代表着一个与 JMS 点对点消息队列服务的连接。JMS 客户程序调用 createQueueSession()方法创建 QueueSession 的实例，createQueueSession()方法调用中一个 boolean 类型的参数指定了 QueueSession 对象是否要提供事务支持。另外，回执的模式也在 createQueueSession()调用中通过参数指定，这个参数的值可以是 3 个静态的标识符之一：AUTO_ACKNOWLEDGE、CLIENT_ACKNOWLEDGE 和 DUPS_OK_ACKNOWLEDGE。

QueueSession.createQueue()方法返回一个 Queue 对象的实例，调用 Queue.getQueueName()方法可以返回队列的名字。

QueueSession.createSender()方法创建一个 QueueSender 消息生产者，利用 QueueSender 可以把消息发送到 Queue。消息可以通过各种不同的 QueueSender.send()方法发送到 Queue，这些不同的 send()方法能够把消息发送给 QueueSender 对象关联的 Queue 对象，或者发送给 send()方法调用中指定的 Queue 对象。消息递送模式、优先级、消息的有效时间都可以在调用

QueueSender.send()方法时指定。另外，发送给 Queue 的消息可以用 Session 接口中定义的各种消息构造方法创建。

10.6.8　发布—订阅消息模式

发布—订阅消息机制也是核心 JMS 机制的一种扩展，它增加了一些适合发布—订阅消息模式的功能。连接工厂、连接、会话、消息生产者、消息消费者等都用发布—订阅形式的接口进行了扩展。

JMS 客户程序通过 JNDI 获得一个 TopicConnectionFactory 对象的引用。TopicConnectionFactory.createTopicConnection()方法用来创建一个 TopicConnection 对象的实例。调用 createTopicConnection()方法时可以指定一个用户名字和密码，或者使用该方法不带参数的版本，此时假定使用默认用户身份。

TopicConnection 接口是 Connection 接口的一种子类型，它代表着一个与 JMS 发布—订阅消息服务的连接。JMS 客户程序调用 TopicConnection.createTopicSession()方法创建 TopicSession 的实例。会话的事务支持和回执模式也在创建 TopicSession 时指定。

TopicSession.createTopic()方法返回一个 Topic 对象的实例。Topic 接口封装了一个话题目的地，发布者将向该目的地发送消息，而订阅者从该目的地接收消息。不同的服务提供者按照不同的方式实现话题名称的层次结构，调用 Topic.getTopicName()方法可以获得话题的 String 形式的描述。

TopicSession.createPublisher()方法创建一个 TopicPublisher 消息生产者，它用来把消息发布到 Topic。消息可以通过各种不同的 TopicPublisher.publish()方法发布到 Topic，这些不同的 publish()方法能够把消息发送给 TopicPublisher 对象关联的 Topic 对象，或者发送给 publish()方法调用中指定的 Topic 对象。消息递送模式、优先级、消息的有效时间都可以在调用 TopicPublisher.publish()方法时指定。另外，发送给 Topic 的消息可以用 Session 接口中定义的各种消息构造方法创建。

10.6.9　消息驱动 Bean 应用实例

本实例是一个消息驱动 Bean 应用的简单例子。该客户端应用程序把消息发送到队列，这个队列由管理员通过 j2eeadmin 命令创建。JMS 提供者（这里是 J2EE 服务器）把消息传递给消息驱动 Bean 的实例，并且由 Bean 的实例处理消息。该程序由以下两部分构成。

- SimpleMessageClient：J2EE 应用客户程序，用来向队列发送消息。
- SimpleMessageEJB：一个消息驱动的 Bean，用来异步地接收和处理由客户程序发送到队列的消息。

1. 客户端

下面是客户端程序 SimpleMessageClient.java 的代码：

```
import javax.jms.*;
import javax.naming.*;

public class SimpleMessageClient
{
```

```
    public static void main(String[] args)
    {

        Context                jndiContext = null;
        QueueConnectionFactory  queueConnectionFactory = null;
        QueueConnection         queueConnection = null;
        QueueSession           queueSession = null;
        Queue                  queue = null;
        QueueSender            queueSender = null;
        TextMessage             message = null;
        final int                 NUM_MSGS = 3;

        try
        {
            jndiContext = new InitialContext();
        }catch (NamingException e) {
            System.out.println("不能创建 JNDI 上下文:"+ e.toString());
            System.exit(1);
        }

        try
        {
            queueConnectionFactory=(QueueConnectionFactory)jndiContext.lookup
("java:comp/env/jms/MyQueueConnectionFactory");
            queue=(Queue)jndiContext.lookup("java:comp/env/jms/QueueName");
        } catch (NamingException e){
            System.out.println("JNDI lookup failed: " +e.toString());
            System.exit(1);
        }

        try
        {
            queueConnection=queueConnectionFactory.createQueueConnection();
            queueSession=queueConnection.createQueueSession(false,
                   Session.AUTO_ACKNOWLEDGE);
            queueSender = queueSession.createSender(queue);
            message = queueSession.createTextMessage();

            String msgArray[]={"老大孙悟空","老二猪八戒","老三沙和尚"};

            for (int i = 0; i < NUM_MSGS; i++)
            {
                message.setText("我是" + msgArray[i] );
                System.out.println("Sending message: " +
                    message.getText());
                queueSender.send(message);
            }

        }catch (JMSException e) {
            System.out.println("异常:" + e.toString());
```

```
        }
        finally
        {
            if (queueConnection != null)
        {
                try
        {
                    queueConnection.close();
                } catch (JMSException e)
{}
            }
            System.exit(0);
        }
    }
}
```

下面看一下程序是如何工作的：SimpleMessageClient 把消息发送到 SimpleMessageBean 所监听的队列。该客户程序首先确定连接工厂和队列：

```
queueConnectionFactory =    (QueueConnectionFactory) jndiContext.lookup
        ("java:comp/env/jms/MyQueueConnectionFactory");
queue = (Queue)jndiContext.lookup("java:comp/env/jms/QueueName");
```

接下来，该客户程序创建队列连接、会话和一个消息发送器：

```
queueConnection = queueConnectionFactory.createQueueConnection();
queueSession = queueConnection.createQueueSession(false,
        Session.AUTO_ACKNOWLEDGE);
queueSender = queueSession.createSender(queue);
```

最后，客户程序把几个消息发送到队列：

```
message = queueSession.createTextMessage();
for (int i = 0; i < NUM_MSGS; i++)
{
    message.setText("我是" + msgArray[i] );
    System.out.println("Sending message: " + message.getText());
    queueSender.send(message);
}
```

2. MDB 组件

SimpleMessageEJB 类阐明了编写消息驱动 Bean 类的基本要求：

- 实现 MessageDrivenBean 接口和 MessageListener 接口。
- 类定义为 public 类型。
- 类不能定义成 abstract 或 final。
- 实现一个 onMessage()方法。
- 实现一个 ejbCreate()方法和一个 ejbRemove()方法。
- 包含一个 public 类型的不需要参数的构造方法。
- 不能定义 finalize()方法。

下面是消息驱动 Bean 的实例 SimpleMessageBean.java 的代码：

```
import java.io.Serializable;
import java.rmi.RemoteException;
import javax.ejb.EJBException;
```

```
import javax.ejb.MessageDrivenBean;
import javax.ejb.MessageDrivenContext;
import javax.ejb.CreateException;
import javax.naming.*;
import javax.jms.*;

public class SimpleMessageBean implements MessageDrivenBean, MessageListener
{
    private transient MessageDrivenContext mdc = null;
    private Context context;

    public SimpleMessageBean()
    {
        System.out.println("正在执行
        SimpleMessageBean.SimpleMessageBean()");
    }

    public void setMessageDrivenContext(MessageDrivenContext mdc)
    {
        System.out.println("正在执行"
            + "SimpleMessageBean.setMessageDrivenContext()");
        this.mdc = mdc;
    }

    public void ejbCreate()
    {
         System.out.println("正在执行 SimpleMessageBean.ejbCreate()");
    }

    public void onMessage(Message inMessage)
{
        TextMessage msg = null;

        try
        {
            if (inMessage instanceof TextMessage)
            {
                msg = (TextMessage) inMessage;
                System.out.println("MESSAGE BEAN:收到消息:  "  + msg.getText());
            }
            else
            {
                System.out.println("消息类型错误:  "
                    + inMessage.getClass().getName());
            }
        }
```

```
        catch (JMSException e)
        {
            e.printStackTrace();
            mdc.setRollbackOnly();
        }
        catch (Throwable te)
        {
            te.printStackTrace();
        }
    }
    public void ejbRemove()
{
        System.out.println("正在执行 SimpleMessageBean.remove()");
    }
}
```

与会话 Bean 和实体 Bean 不同，消息驱动的 Bean 没有定义客户程序访问的接口。客户程序并不是先定位消息驱动的 Bean，再调用这些 Bean 上的方法。虽然消息驱动的 Bean 没有业务方法，但它们可以包含由 onMessasge()方法内部调用的辅助方法。

当队列接收到一个消息，EJB 容器将调用消息驱动 Bean 的 onMessage()方法。在 SimpleMessageBean 类中，onMessage()方法把接收到的消息强制转化成 TextMessage 类型，然后显示出文本信息。

```
public void onMessage(Message inMessage)
{
TextMessage msg = null;

    try
    {
        if (inMessage instanceof TextMessage)
        {
            msg = (TextMessage) inMessage; System.out.println("MESSAGE BEAN:
        收到消息: "+ msg.getText());
        }
        else
        {
                  System.out.println("    消    息    类    型    错    误    :
    " + inMessage.getClass().getName());
        }
    }
    catch (JMSException e)
    {
        e.printStackTrace();
        mdc.setRollbackOnly();
    }
    catch (Throwable te)
    {
```

```
        te.printStackTrace();
        }
}
```

消息驱动 Bean 的 ejbCreate()方法和 ejbRemove()方法必须符合以下要求：

- 访问控制修饰符必须是 public。
- 返回值类型必须是 void。
- 不能有 static 和 final 修饰符。
- throws 子句不能定义任何应用自定义的异常。
- 不能带有参数。
- 在 SimpleMessageBean 类中，ejbCreate()方法和 ejbRemove()方法都是空的，不执行任何有实际意义的操作。

3. 部署和运行

要部署和运行该应用程序，首先运行 J2EE 服务器。为便于查看消息驱动 Bean 的输出，必须以-verbose 模式启动该服务器：

```
j2ee -verbose
```

然后，用下面的 j2eeadmin 命令创建队列：

```
j2eeadmin -addJmsDestination jms/MyQueue queue
```

验证队列是否已经创建成功：

```
j2eeadmin -listJmsDestination
```

如图 10-18 所示，说明创建成功。

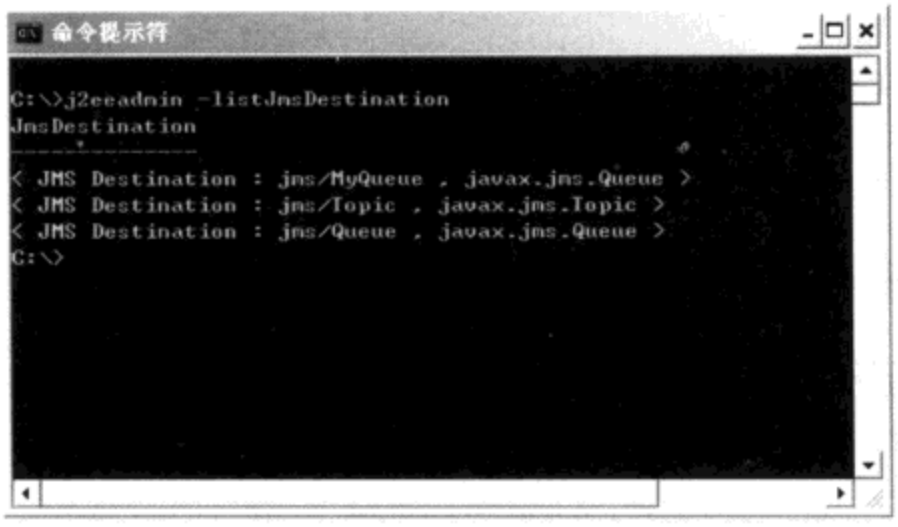


图 10-18　验证队列的创建

启动 Deployment Tool，选择菜单 File→Open，打开 SimpleMessageApp.ear 文件。接着，选择菜单 Tools→Deploy，以部署该应用。在出现部署提示时，选中 Return Client JAR 检查框。

启动一个命令窗口，进入 EAR 文件（SimpleMessageAppClient.jar 文件）所在目录，把环境变量 APPCPATH 设置为 SimpleMessageAppClient.jar。然后，执行下面的命令：

```
runclient -client SimpleMessageApp.ear -name SimpleMessageClient -textauth
```

在登录提示中，输入用户名 j2ee，输入密码 j2ee。此时，客户程序将输出以下内容：

```
Sending message: 我是老大孙悟空
Sending message: 我是老二猪八戒
Sending message: 我是老三沙和尚
```

在启动 j2ee 服务器的命令窗口，可以看到如图 10-19 所示的输出。

图 10-19　J2EE 服务器的输出

10.7　基于 Web 的 EJB 应用程序示例

在设计基于 Web 的 EJB 应用程序时，通常不知该从何下手。到底是从 Web 方开始，还是从 EJB 方开始呢？让我们看看典型的三层应用程序模型，在这里表示层取决于业务层的接口，业务层取决于数据层的接口。就某一点来说，需要先定义数据层再定义业务层，然后定义业务层再定义表述层。但是，通常我们最初总是在层与层之间跳来跳去，在每个层都定义一点点，直到有了好的设计构思，然后再开始深入设计每一层。并且从数据层开始，直到表示层。在收集应用程序的需求时，通常集中考虑表示层。这是因为在需求收集期间，我们关心的是系统如何与用户交互。只有知道系统的行为之后，才能正确地设计系统。

10.7.1　收集需求

我们经常会听到开发人员抱怨系统的需求不明确。需求不会是现成的，用户通常要看到东西之后才知道具体要什么，所以要尽可能地向用户显示可能提供的东西。

开始收集需求时，用户可能会说：“我要一个 Web 站点，让别人联机定购产品。”根据 Web 应用程序的经验，假设需要生成购物车应用程序。联机订单系统有三大部分：显示可用项目的产品目录、让客户选择项目和提交订单。因此，首先要确定如何完成这些工作。

刚开始，和用户一起坐下来，描绘可能的 Web 站点，显示项目的产品目录，并且可以让人单击项目将其加进购物车中。下一步，是要确定如何显示产品目录。如果是一小组项目，则可以全部在一页中显示；如果是一大组项目，则可能要增加搜索功能，或将产品目录组织成一系列页面。在收集需求期间，可以在纸上画出这些页面，或在计算机上生成草图，以描绘出用户完成订单要经过的每一个步骤。由于需要知道可能发生的各种情况，因此可以和用户一起过一遍，看看系统如何响应。例如，如何将项目加进购物车？如何从购物车中减去项目？如何下订单？

建立购物车应用程序时需要考虑的一个问题是项目应在购物车中保留多长时间。具体的说，浏览器关闭时怎么办？有些应用程序将项目与浏览器会话相关联。如果浏览器关闭，则会失去购物车中的项目。而有些站点将项目保留在数据库中。将项目保留在数据库中更方便，但是可能比使用会话慢一些。

是否需要保存用户选项？如果要，则需要跟踪用户，可能要用登录机制。用户首先要不要登录，或只在用户进行特定操作时才需要登录？下订单时，是保留信用卡信息还是要求用户重新输入信用卡信息？如果保留信用卡信息，则要采取保护信用卡信息的步骤，这是一个很大的责任。还应该考虑如何防止黑客进入数据库窃取信用卡信息。在建立系统需求时，事先应该考虑所有这些问题。可以参考其他站点是怎么实现的。

现在举一个例子，假设系统要把整个产品目录放在主页中。为了订购项目，单击“加入购物车”链接。主页会出现购物车的缩略图，用户还可以在另一页中浏览整个购物车。系统将购物车项目保存在数据库中，因此可以关闭浏览器，几天后返回时，购物车中仍然有相同的项目。还可以提供 E-mail 地址和口令，以使系统能记住用户。系统将订单存放在数据库中，这样可以轻松获取原先的订购清单。下订单完成时，输入发货地址和信用卡信息，并且下订单完成后，购物推车应该清空。

10.7.2 层的设计

在设计应用程序的不同组件时，最好从数据层开始，向外一直到表述层。首先要画出所处理对象的草图。从前面的需求看，需要跟踪一个客户，因此需要客户对象。还要有客户订购产品，因此需要产品对象和订单对象。订单中的各个项目表示为订购的项目，因此需要订单项目对象。图 10-20 显示了该应用程序的简单对象模型。

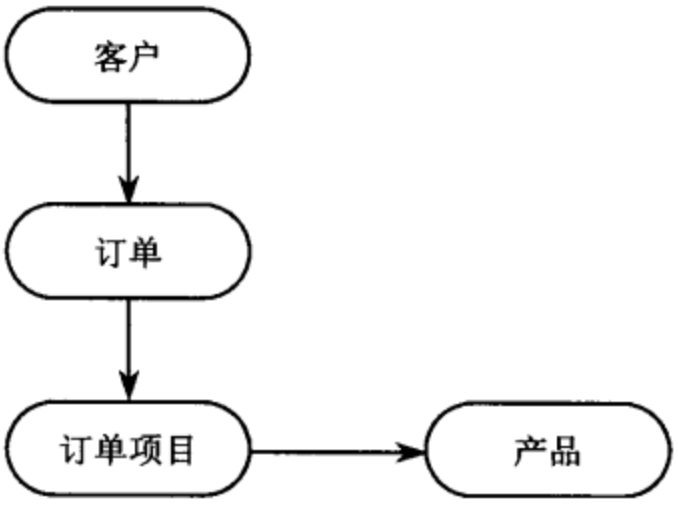


图 10-20 应用程序的简单对象模型

这些项目表示应用程序中的核心数据项目，即实体 Bean。这时已经知道 Bean 中可能要放些什么东西，因此可以事先画出属于这些 Bean 中的项目。

一个订单应该有订单项目、发货地址和记账信息。发货地址通常包含客户姓名、街道、城市、邮政编码和国家等。根据应用程序的情况，可能要增加其他信息。记账信息需要付款方式、信用卡号和各个价格汇总。

订单项目对象包含订购的产品、数量和总价。

客户至少要有某种标识名（用户名或 E-mail 地址），还可能包含口令，以便防止一个客户访问另一个客户的订单信息。可能要跟踪客户姓名和地址，以便用客户当前地址初始化新订单的发货信息。

产品有某种标识符、名称和价格。在典型的 Web 应用程序中，要将不同事物与产品相关

联，如图形、各种说明、发货信息、客户意见等。

在确定实体 Bean 的外观以后，就可以设计 Session Bean 以细化模型，并指出在哪里生成 Java 代码。

对于基本订单功能，要能够浏览项目清单、将项目加入订单、从订单中删除项目和完成订单。还要浏览旧订单清单，以便检查所下订单的状态。由于要跟踪客户数据，因此需要让客户注册到站点、登录和更新任何客户数据。

在标识 Session Bean 中的基本方法之后，开始布置 Web 站点。仔细研究完整的订单过程，看看 Web 页面如何与 Session Bean 交互。例如，要显示产品，Session Bean 中就要有显示产品的方法。如果要搜索呢？可能要在 Session Bean 中增加搜索方法。客户如何登录站点？如果是新客户，Session Bean 中能否保存新客户信息？

确保 Web 站点中输入的每个数据都存放成实体 Bean（如果需要）。完成这个过程之后，就知道 Session Bean 需要进行的一切、实体 Bean 需要进行的一切和 JSP 与小服务需要进行的一切了。

还要与用户一起仔细研究完整的订单过程，以保证站点符合用户的需求。对于更复杂的应用程序，还要考虑生成用户界面原型。研究建模过程时，很容易冒冒失失地开始编写代码。对于小型应用程序，通常可以在很短的时间内建模。但对于大项目，最好将项目分解成各个小部分。建模一个部分并实施，然后建模下一个部分再实施。

许多公司不愿花很多时间对应用程序建模，而是编写代码和不断修改代码，直到符合客户的需求。这是由于代码是最终结果，而建模过程则没有成就感。但是，他们并没有认识到，建模过程中很容易修改的东西，在代码中要修改就很费劲了。

建模和设计过程中还要记住一点，表示模型的符号并不是最重要的，模型本身才是最重要的。不要因为不懂 UML 或其他建模语言中的各种名词而难过，只要按自己理解的方式表示模型就可以了。有空的时候再去学习一些常用符号，看看是否能更好地表示模型。使用造型符号就像使用编程语言，使用 Java 之前也不需要学习 Java 的全部知识。

10.7.3　生成实体 Bean

定义实体 Bean 的第一步是生成 Remote 接口。如果设计阶段工作彻底，就已经知道 Remote 接口中需要的所有方法了，否则可能要多次改变接口才能得到正确的结果。不过说实话，即使系统模型很好，也还可能要改变接口，但不会改得那么多。下面显示了 Customer Bean 的 Remote 接口和 Home 接口。

Customer.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import javax.ejb.*;
import java.util.Collection;
import java.util.Date;

public interface Customer extends EJBObject
{
    public String getCustomerId() throws RemoteException;
```

```
    public String getFirstName() throws RemoteException;
    public void setFirstName(String aFirstName) throws RemoteException;

    public String getMiddleName() throws RemoteException;
    public void setMiddleName(String aMiddleName) throws RemoteException;

    public String getLastName() throws RemoteException;
    public void setLastName(String aLastName) throws RemoteException;

    public String getAddress1() throws RemoteException;
    public void setAddress1(String anAddress1) throws RemoteException;

    public String getAddress2() throws RemoteException;
    public void setAddress2(String anAddress2) throws RemoteException;

    public String getCity() throws RemoteException;
    public void setCity(String aCity) throws RemoteException;

    public String getState() throws RemoteException;
    public void setState(String aState) throws RemoteException;

    public String getZip() throws RemoteException;
    public void setZip(String aZip) throws RemoteException;

    public String getCountry() throws RemoteException;
    public void setCountry(String aCountry) throws RemoteException;

    public String getEmail() throws RemoteException;
    public void setEmail(String anEmail) throws RemoteException;

    public String getPassword() throws RemoteException;
    public void setPassword(String aPassword) throws RemoteException;

    public Date getLastLogin() throws RemoteException;
    public void setLastLogin(Date lastLogin) throws RemoteException;

    public Collection getOrders() throws RemoteException;
    public void setOrders(Collection orders) throws RemoteException;

    public Order getCurrentOrder() throws RemoteException;
    public void setCurrentOrder(Order aCurrentOrder) throws RemoteException;

    public CustomerView getView() throws RemoteException;
    public void setFromView(CustomerView aView) throws RemoteException;

    public void register(String email, String password,
                         String firstName, String middleName, String lastName)
throws RemoteException;
}
```

CustomerHome.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import javax.ejb.*;
import java.util.Collection;

public interface CustomerHome extends EJBHome
{
    public Customer create()
        throws RemoteException, CreateException;
    public Customer findByPrimaryKey(String primaryKey)
        throws RemoteException, FinderException;
    public Customer findByEmail(String email)
        throws RemoteException, FinderException;
}
```

注意除了基本数据类型之外，Bean 可以将自己以视图对象的形式返回。下面是 CustomerView 类的代码：

```
package usingj2ee.shopping;

import java.util.Date;

public class CustomerView implements java.io.Serializable
{
    public String email;
    public String firstName;
    public String middleName;
    public String lastName;
    public String address1;
    public String address2;
    public String city;
    public String state;
    public String zip;
    public String country;

    public CustomerView()
    {
    }

    public String getFirstName() { return firstName; }
    public void setFirstName(String aFirstName) { firstName = aFirstName; }

    public String getMiddleName() { return middleName; }
    public void setMiddleName(String aMiddleName) { middleName = aMiddleName; }

    public String getLastName() { return lastName; }
    public void setLastName(String aLastName) { lastName = aLastName; }

    public String getAddress1() { return address1; }
```

```
    public void setAddress1(String anAddress1) { address1 = anAddress1; }

    public String getAddress2() { return address2; }
    public void setAddress2(String anAddress2) { address2 = anAddress2; }

    public String getCity() { return city; }
    public void setCity(String aCity) { city = aCity; }

    public String getState() { return state; }
    public void setState(String aState) { state = aState; }

    public String getZip() { return zip; }
    public void setZip(String aZip) { zip = aZip; }

    public String getCountry() { return country; }
    public void setCountry(String aCountry) { country = aCountry; }
}
```

Customer 接口中的大多数方法是处理客户姓名和地址。由于客户可能有多个旧订单，因此可以请求 Customer 对象提供旧订单清单。此外，客户还有“当前订单”，也就是客户的购物车。

下面是 Order Bean 的 Remote 接口和 Home 接口的代码。

Order.java 的代码如下所示：

```
package usingj2ee.shopping;
import java.rmi.*;
import javax.ejb.*;
import java.util.Collection;
import java.util.Date;

public interface Order extends EJBObject
{
    public String getOrderId() throws RemoteException;

    public Customer getCustomer() throws RemoteException;
    public void setCustomer(Customer aCustomer) throws RemoteException;

    public String getAddress1() throws RemoteException;
    public void setAddress1(String anAddress1) throws RemoteException;

    public String getAddress2() throws RemoteException;
    public void setAddress2(String anAddress2) throws RemoteException;

    public String getCity() throws RemoteException;
    public void setCity(String aCity) throws RemoteException;

    public String getState() throws RemoteException;
    public void setState(String aState) throws RemoteException;

    public String getZip() throws RemoteException;
```

```
public void setZip(String aZip) throws RemoteException;

public String getCountry() throws RemoteException;
public void setCountry(String aCountry) throws RemoteException;

public String getCreditCard() throws RemoteException;
public void setCreditCard(String aCreditCard) throws RemoteException;

public Date getTimePlaced() throws RemoteException;

public double getSubtotal() throws RemoteException;
public double getShipping() throws RemoteException;
public double getTax() throws RemoteException;
public double getTotalPrice() throws RemoteException;

public boolean isEmpty() throws RemoteException;

public int getNextItemNumber() throws RemoteException;
public void setNextItemNumber(int nextItemNumber) throws RemoteException;

public String getStatus() throws RemoteException;
public void setStatus(String aStatus) throws RemoteException;

public OrderView getView() throws RemoteException;
public OrderHistoryView getHistoryView() throws RemoteException;
public void setFromData(OrderData data) throws RemoteException;

    public void addItem(OrderLineItem item) throws RemoteException;
    public void removeItem(OrderLineItem item) throws RemoteException;
    public void removeItem(String productId, int quantity)
        throws RemoteException;

    public String place() throws EJBException, RemoteException;
}
```

OrderHome.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import javax.ejb.*;
import java.util.Collection;

public interface OrderHome extends EJBHome
{
    public Order create(Customer aCustomer)
        throws RemoteException, CreateException;
    public Order findByPrimaryKey(String primaryKey)
        throws RemoteException, FinderException;
}
```

和 Customer Bean 的 Remote 接口一样，Order 对象也可以将自己以视图对象形式返回，可

以复制到视图对象和从视图对象初始化。下面是 OrderView 类和 OrderHistoryView 类的代码。

OrderView.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.util.Date;

public class OrderView implements java.io.Serializable
{
    protected String orderId;

    public String address1;
    public String address2;
    public String city;
    public String state;
    public String zip;
    public String country;
    public String creditCard;
    public Date timePlaced;
    public double subtotal;
    public double shipping;
    public double tax;
    public double totalPrice;
    public String status;

    public OrderLineItemView[] lineItems;

    public OrderView()
    {
    }

    public String getOrderId() { return orderId; }
    public String getAddress1() { return address1; }
    public String getAddress2() { return address2; }
    public String getCity() { return city; }
    public String getState() { return state; }
    public String getZip() { return zip; }
    public String getCountry() { return country; }
    public String getCreditCard() { return creditCard; }
    public Date getTimePlaced() { return timePlaced; }
    public double getSubtotal() { return subtotal; }
    public double getShipping() { return shipping; }
    public double getTax() { return tax; }
    public double getTotalPrice() { return totalPrice; }
    public String getStatus() { return status; }

    public OrderLineItemView[] getLineItems() { return lineItems; }
}
```

OrderHistoryView.java 的代码如下所示：

```
package usingj2ee.shopping;
```

```
import java.util.Date;

public class OrderHistoryView implements java.io.Serializable
{
    protected String orderId;

    public Date timePlaced;
    public double totalPrice;
    public String status;

    public OrderHistoryView()
    {
    }

    public String getOrderId() { return orderId; }
    public Date getTimePlaced() { return timePlaced; }
    public double getTotalPrice() { return totalPrice; }
    public String getStatus() { return status; }
}
```

Order 对象中的 place 方法负责下订单。从购物车角度看，这表示该订单不再是客户的当前订单。如果客户再往购物车中加项目，则进入新订单。

下面是 OrderLineItem Bean 的 Remote 接口和 Home 接口。

OrderLineItem.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import javax.ejb.*;
import java.util.Collection;

public interface OrderLineItem extends EJBObject
{
    public String getOrderId() throws RemoteException;

    public int getItemNumber() throws RemoteException;

    public int getQuantity() throws RemoteException;
    public void setQuantity(int aQuantity) throws RemoteException;

    public double getPrice() throws RemoteException;
    public void setPrice(double aPrice) throws RemoteException;

    public OrderLineItemView getView() throws RemoteException;

    public Order getOrder() throws RemoteException;

    public Product getProduct() throws RemoteException;
    public void setProduct(Product aProduct) throws RemoteException;
}
```

OrderLineItemHome.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import javax.ejb.*;
import java.util.Collection;

public interface OrderLineItemHome extends EJBHome
{
    public OrderLineItem create(Order parentOrder)
        throws RemoteException, CreateException;
    public OrderLineItem findByPrimaryKey(OrderLineItemPK primaryKey)
        throws RemoteException, FinderException;
}
```

下面是 Product Bean 的 Remote 接口和 Home 接口。

Product.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import javax.ejb.*;
import java.util.Collection;

/** 定义可以在 Customer 对象中调用的方法*/

public interface Product extends EJBObject
{
    public String getProductId() throws RemoteException;
    public void setProductId(String aProductId) throws RemoteException;

    public String getName() throws RemoteException;
    public void setName(String aName) throws RemoteException;

    public String getDescription() throws RemoteException;
    public void setDescription(String aDescription) throws RemoteException;

    public double getPrice() throws RemoteException;
    public void setPrice(double aPrice) throws RemoteException;

    public String getCategory() throws RemoteException;
    public void setCategory(String aCategory) throws RemoteException;

    public String getImageURL() throws RemoteException;
    public void setImageURL(String aURL) throws RemoteException;

    public ProductView getView() throws RemoteException;
}
```

ProductHome.java 的代码如下所示：

```
package usingj2ee.shopping;
```

```
import java.rmi.*;
import javax.ejb.*;
import java.util.Collection;

public interface ProductHome extends EJBHome
{
    public Product create(String productId)
        throws RemoteException, CreateException;

    public Product findByPrimaryKey(String primaryKey)
        throws RemoteException, FinderException;

    public Collection findAll()
        throws RemoteException, FinderException;

    public Collection findByCategory(String category)
        throws RemoteException, FinderException;

    public String[] getCategories()
        throws RemoteException;
}
```

在实现一个实体 Bean 时，最好用容器管理持久性（CMP），因为它能自动处理 Bean 之间的关系。

Customer Bean 和 Order Bean 都要求自动生成关键字。通常，可以用客户的 E-mail 地址作为 Customer Bean 的关键字，但由于购物车可以先建立订单而后登录，因此要在知道 E-mail 地址之前生成一个 Customer。下面显示了 Customer Bean 的实现类。

CustomerImpl.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import java.util.*;
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import javax.naming.*;
import java.util.Date;

public abstract class CustomerImpl implements EntityBean
{
    private EntityContext context;
    private Connection conn;
    public CustomerImpl()
    {
    }
    public void setEntityContext(EntityContext aContext)
    {
        context = aContext;
    }
```

```
public void unsetEntityContext()
{
    context = null;
}
public String ejbCreate()
    throws CreateException
{
    try
    {
        setCustomerId(getNextId());
        return null;
    }
    catch (SQLException exc)
    {
        throw new CreateException(
            "Unable to access database: "+exc.toString());
    }
}
public void ejbPostCreate()
    throws CreateException
{
}
public void ejbActivate()
    throws EJBException
{
}

public void ejbPassivate()
    throws EJBException
{
}

public void ejbLoad()
    throws EJBException
{
}
public void ejbStore()
    throws EJBException
{
}
public void ejbRemove()
    throws EJBException
{
}

public CustomerView getView()
{
    CustomerView view = new CustomerView();

    view.firstName = getFirstName();
```

```
        view.middleName = getMiddleName();
        view.lastName = getLastName();
        view.address1 = getAddress1();
        view.address2 = getAddress2();
        view.city = getCity();
        view.state = getState();
        view.zip = getZip();
        view.country = getCountry();
        view.email = getEmail();

        return view;
    }

    public void setFromView(CustomerView view)
    {
        setFirstName(view.firstName);
        setMiddleName(view.middleName);
        setLastName(view.lastName);
        setAddress1(view.address1);
        setAddress2(view.address2);
        setCity(view.city);
        setState(view.state);
        setZip(view.zip);
        setCountry(view.country);
    }

    protected Connection getConnection()
        throws SQLException, NamingException
    {
        InitialContext context = new InitialContext();

        DataSource ds = (DataSource) context.lookup(
            "java:comp/env/jdbc/OrderDB");

        return ds.getConnection();
    }

    protected String getNextId()
        throws SQLException
    {
        Connection conn = null;

        try
        {
            conn = getConnection();

            PreparedStatement ps = conn.prepareStatement(
                "update customer_id_seq set next_customer_id = "+
                    "next_customer_id + 1");
```

```
            if (ps.executeUpdate() != 1)
            {
                throw new SQLException("Unable to generate customer id");
            }

            ps.close();

            ps = conn.prepareStatement(
                "select next_customer_id from customer_id_seq");

            ResultSet rs = ps.executeQuery();

            if (rs.next())
            {
                return ""+rs.getInt("next_customer_id");
            }
            else
            {
                throw new SQLException("Unable to generate customer id");
            }
        }
        catch (NamingException exc)
        {
            throw new SQLException("Unable to generate customer id: "+
                exc.toString());
        }
        finally
        {
            try
            {
                conn.close();
            }
            catch (Exception ignore) {}
        }
    }

    public void register(String email, String password,
        String firstName, String middleName, String lastName)
    {
        setEmail(email);
        setPassword(password);
        setFirstName(firstName);
        setMiddleName(middleName);
        setLastName(lastName);
    }

    public abstract String getCustomerId();
    public abstract void setCustomerId(String customerId);

    public abstract String getFirstName();
```

```
public abstract void setFirstName(String aFirstName);

public abstract String getMiddleName();
public abstract void setMiddleName(String aMiddleName);

public abstract String getLastName();
public abstract void setLastName(String aLastName);

public abstract String getAddress1();
public abstract void setAddress1(String anAddress1);

public abstract String getAddress2();
public abstract void setAddress2(String anAddress2);

public abstract String getCity();
public abstract void setCity(String aCity);

public abstract String getState();
public abstract void setState(String aState);

public abstract String getZip();
public abstract void setZip(String aZip);

public abstract String getCountry();
public abstract void setCountry(String aCountry);

    public abstract String getEmail();
    public abstract void setEmail(String anEmail);

    public abstract String getPassword();
    public abstract void setPassword(String aPassword);

    public abstract Date getLastLogin();
    public abstract void setLastLogin(Date lastLogin);

    public abstract Collection getOrders();
    public abstract void setOrders(Collection orders);

    public abstract Order getCurrentOrder();
    public abstract void setCurrentOrder(Order aCurrentOrder);
}
```

10.7.4　生成会话 Bean

关于客户端应用程序是否能访问实体 Bean，人们的观点不一。显然，从技术角度看，客户机可以访问实体 Bean，这只是设计问题。在客户机直接使用实体 Bean 时，公用的业务逻辑可能移植到客户端应用程序中，使应用程序难以维护。作为一般原则，客户机最好通过会话 Bean 来访问实体 Bean。

既然客户机只用会话 Bean，有些开发人员就怀疑还有实体 Bean 存在的必要吗？尽管有些

开发人员喜欢编写烦琐的 SQL 代码，但大多数人还是喜欢集中考虑真正的业务问题，因为实体 Bean（特别是使用 CMP 的实体 Bean）可以减少会话 Bean 要进行的数据库操作。

这里的 ShoppingCartSession Bean 使客户端应用程序能够进行任何必要的操作，而不需要直接访问实体 Bean。客户机要访问实体 Bean 的内容时，ShoppingCartSession Bean 返回数据视图，而不是实体 Bean 本身的引用。

购物应用程序的实现方法相当复杂，很难处理过去没有注册的用户。因此这里要将数据与未注册用户相关联，然后在用户注册时保持旧数据。真正的平衡发生在用户界面与业务逻辑要分开时。业务逻辑是否需要知道未注册用户？如果需要，那么如何跟踪？前端要做多少工作以跟踪未注册用户？

实际上，会话 Bean 和前端都要做一些工作以处理未注册用户（和注册用户）。首先，会话 Bean 要允许寻找原先客户的数据（不管注册与否）。对于未注册用户，要用简单客户 ID 号。对于注册用户，则用 E-mail 地址和口令。

另外，会话 Bean 还要允许前端访问产品目录、生成订单和浏览原有订单。对于这个应用程序，会话 Bean 只要返回产品目录清单和目录中的产品清单（或所有产品的清单）。对于有几百个产品的复杂应用程序来说，可能还要增加其他方法，以让前端在产品清单中可以滚动。

要管理订单，就要能在订单中增加项目、删除项目、显示订单和下订单。许多联机购物应用程序要等下订单之前才询问发货信息，ShoppingCartSession Bean 就采用这个方法。

下面是 ShoppingCartSession Bean 的 Remote 接口和 Home 接口。

ShoppingCartSession.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import javax.ejb.*;

public interface ShoppingCartSession extends EJBObject
{
    public ProductView[] getAllProducts()
        throws EJBException, RemoteException;

    public ProductView[] getProductsByCategory(String category)
        throws EJBException, RemoteException;

    public String[] getProductCategories()
        throws EJBException, RemoteException;

    public void addToShoppingCart(String productId, int quantity)
        throws EJBException, RemoteException;

    public void removeFromShoppingCart(String productId, int quantity)
        throws EJBException, RemoteException;

    public String getCustomerId()
        throws EJBException, RemoteException;

    public String login(String email, String password)
```

```
        throws EJBException, RemoteException;

    public String createCustomer()
        throws EJBException, RemoteException;

    public String createCustomer(String email, String password,
        String firstName, String middleName, String lastName)
        throws EJBException, RemoteException;

    public String register(String email, String password,
        String firstName, String middleName, String lastName)
        throws EJBException, RemoteException;

    public boolean customerExists(String email) throws RemoteException;

    public void changePassword(String oldPassword, String newPassword)
        throws EJBException, RemoteException;

    public CustomerView getCustomerData()
        throws EJBException, RemoteException;

    public void updateCustomerData(CustomerView data)
        throws EJBException, RemoteException;

    public OrderView getCurrentOrder()
        throws EJBException, RemoteException;

    public void setOrderData(OrderData data)
        throws EJBException, RemoteException;

    public String saveOrder()
        throws EJBException, RemoteException;

    public OrderHistoryView[] getOrderHistory()
        throws EJBException, RemoteException;

    public OrderView getOrder(String orderId)
        throws EJBException, RemoteException;

    public boolean isEmpty()
        throws RemoteException;
}
```

ShoppingCartSessionHome.java 的代码如下所示：

```
package usingj2ee.shopping;

import java.rmi.*;
import javax.ejb.*;

public interface ShoppingCartSessionHome extends EJBHome
{
```

```
    public ShoppingCartSession create()
        throws RemoteException, CreateException;

    public ShoppingCartSession create(String customerId)
        throws RemoteException, CreateException;
}
```

ShoppingCartSession Bean 的实现类是整个应用程序的核心，它包含许多有用的思想，下面就是 ShoppingCartSessionImpl.java 的代码：

```
package usingj2ee.shopping;

import java.io.*;
import java.rmi.*;
import javax.rmi.*;
import javax.ejb.*;
import javax.naming.*;
import javax.sql.*;
import java.util.*;

public class ShoppingCartSessionImpl implements Session Bean
{
    protected Customer customer;
    protected ProductHome productHome;
    protected CustomerHome customerHome;
    protected OrderHome orderHome;
    protected OrderLineItemHome orderLineItemHome;

    private SessionContext context;

    public ShoppingCartSessionImpl()
    {
    }

    public void ejbCreate()
        throws CreateException
    {
        try
        {
            customer = null;

            Context ctx = new InitialContext();

            productHome= (ProductHome) PortableRemoteObject.narrow(
                        ctx.lookup("java:comp/env/ejb/ProductHome"),
                        ProductHome.class);

            customerHome = (CustomerHome) PortableRemoteObject.narrow(
                           ctx.lookup("java:comp/env/ejb/CustomerHome"),
                           CustomerHome.class);

            orderHome = (OrderHome) PortableRemoteObject.narrow(
```

```
                    ctx.lookup("java:comp/env/ejb/OrderHome"),
                    OrderHome.class);

        orderLineItemHome = (OrderLineItemHome) PortableRemoteObject.
                            narrow(ctx.lookup("java:comp/env/ejb/Ord
                            erLineItemHome"),
                            OrderLineItemHome.class);
    }
    catch (NamingException exc)
    {
        throw new CreateException("Couldn't create shopping cart: "+
            exc.toString());
    }
    catch (EJBException exc)
    {
        throw new CreateException("Couldn't create shopping cart: "+
            exc.toString());
    }
}

public void ejbCreate(String customerId)
    throws CreateException
{
    ejbCreate();

    try
    {
        customer = customerHome.findByPrimaryKey(customerId);
    }
    catch (FinderException exc)
    {
    }
    catch (RemoteException exc)
    {
        throw new CreateException("Couldn't create shopping cart: "+
            exc.toString());
    }
}

public void setSessionContext(SessionContext aContext)
{
    context = aContext;
}

public void ejbActivate()
{
}

public void ejbPassivate()
{
```

```
}

public void ejbRemove()
{
}

public ProductView[] getAllProducts()
    throws RemoteException, EJBException
{
    try
    {
        Collection products = productHome.findAll();
        return getProductViews(products);
    }
    catch (FinderException exc)
    {
        throw new EJBException("Error retrieving products: "+
            exc.toString());
    }
}

public ProductView[] getProductViews(Collection products)
    throws RemoteException, EJBException
{
    ArrayList views = new ArrayList();
    Iterator iter = products.iterator();
    while (iter.hasNext())
    {
        Product prod = (Product) PortableRemoteObject.narrow(
            iter.next(), Product.class);

        views.add(prod.getView());
    }

    return (ProductView[]) views.toArray(
        new ProductView[views.size()]);
}

public ProductView[] getProductsByCategory(String category)
    throws RemoteException, EJBException
{
    try
    {
        Collection products = productHome.findByCategory(category);

        return getProductViews(products);
    }
    catch (FinderException exc)
    {
        throw new EJBException("Error retrieving products: "+
```

```
            exc.toString());
        }
    }

    /*ProductHome 类中的 Home 方法*/
    public String[] getProductCategories()
        throws RemoteException, EJBException
    {
        return productHome.getCategories();
    }

    public void addToShoppingCart(String productId, int quantity)
        throws RemoteException, EJBException
    {
        try
        {
            Product product = (Product) PortableRemoteObject.narrow(
                productHome.findByPrimaryKey(productId), Product.class);

            if (customer == null)
            {
                customer = customerHome.create();
            }

            Order currOrder = customer.getCurrentOrder();

            if (currOrder == null)
            {
                currOrder = orderHome.create(customer);
                currOrder.setCustomer(customer);

                customer.setCurrentOrder(currOrder);
            }

            OrderLineItem lineItem = orderLineItemHome.create(currOrder);

            lineItem.setQuantity(quantity);
            lineItem.setProduct(product);
            lineItem.setPrice(product.getPrice() * quantity);

            currOrder.addItem(lineItem);
        }
        catch (CreateException exc)
        {
            exc.printStackTrace();
            throw new EJBException("Unable to add item to cart: "+
                exc.toString());
        }
        catch (FinderException exc)
        {
```

```
            exc.printStackTrace();
            throw new EJBException("Unable to add item to cart: "+
                exc.toString());
        }
    }

    public void removeFromShoppingCart(String productId, int quantity)
        throws RemoteException, EJBException
    {
        if (customer == null) return;

        Order currOrder = customer.getCurrentOrder();
        if (currOrder == null)
        {
            return;
        }
        currOrder.removeItem(productId, quantity);
    }

    public String getCustomerId()
        throws RemoteException, EJBException
    {
        if (customer == null)
        {
            return null;
        }
        else
        {
            return customer.getCustomerId();
        }
    }

    public String login(String email, String password)
        throws RemoteException, EJBException
    {
        try
        {
            Customer newCustomer = customerHome.findByEmail(email);

            if (newCustomer.getPassword().equals(password))
            {
                customer = newCustomer;

                return customer.getCustomerId();
            }
            else
            {
                return null;
            }
        }
```

```
        catch (FinderException exc)
        {
            return null;
        }
    }

    public String createCustomer()
        throws RemoteException, EJBException
    {
        try
        {
            customer = customerHome.create();

            return customer.getCustomerId();
        }
        catch (CreateException exc)
        {
            exc.printStackTrace();
            throw new EJBException("Error creating customer: "
                + exc.toString());
        }
    }

    public String createCustomer(String email, String password,
        String firstName, String middleName, String lastName)
        throws RemoteException, EJBException
    {
        try
        {
            try
            {
                Customer cust = customerHome.findByEmail(email);

                if (cust != null)
                {
                    throw new EJBException("Customer already exists");
                }
            }
            catch (FinderException ignore)
            {
            }

            customer = customerHome.create();
            customer.setEmail(email);
            customer.setPassword(password);
            customer.setFirstName(firstName);
            customer.setMiddleName(middleName);
            customer.setLastName(lastName);

            return customer.getCustomerId();
```

```
        }
        catch (CreateException exc)
        {
            exc.printStackTrace();
            throw new EJBException("Error creating customer: "
                + exc.toString());
        }
    }
    public String register(String email, String password,
        String firstName, String middleName, String lastName)
        throws RemoteException, EJBException
    {
        try
        {
            try
            {
                Customer cust = customerHome.findByEmail(email);

                if (cust != null)
                {
                    throw new EJBException("Customer already exists");
                }
            }
            catch (FinderException ignore)
            {
            }

            if (customer == null)
            {
                customer = customerHome.create();
            }

            customer.register(email, password, firstName, middleName, lastName);

            return customer.getCustomerId();
        }
        catch (CreateException exc)
        {
            exc.printStackTrace();
            throw new EJBException("Error creating customer: "
                + exc.toString());
        }
    }

    public boolean customerExists(String email)
    {
        try
        {
            Customer cust = customerHome.findByEmail(email);
```

```
            if (cust != null)
            {
                return true;
            }

            return false;
        }
        catch (Exception exc)
        {
            return false;
        }
    }

    public void changePassword(String oldPassword, String newPassword)
        throws RemoteException, EJBException
    {
        if (customer == null)
        {
            throw new EJBException(
                "Can't change the password - there's no customer!");
        }

        if (!customer.getPassword().equals(oldPassword))
        {
            throw new EJBException(
                "Old password is invalid");
        }

        customer.setPassword(newPassword);
    }

    public CustomerView getCustomerData()
        throws RemoteException, EJBException
    {
        if (customer == null)
        {
            return null;
        }
        else
        {
            return customer.getView();
        }
    }

    public void updateCustomerData(CustomerView data)
        throws RemoteException, EJBException
    {
        if (customer == null)
        {
            return;
```

```
        }
        else
        {
            customer.setFromView(data);
        }
    }

    public OrderView getCurrentOrder()
        throws RemoteException
    {
        if (customer == null)
        {
            return null;
        }

        Order order = customer.getCurrentOrder();

        if (order == null)
        {
            return null;
        }

        return order.getView();
    }

    public void setOrderData(OrderData data)
        throws RemoteException, EJBException
    {
        try
        {
            if (customer == null)
            {
                customer = customerHome.create();
            }

            Order order = customer.getCurrentOrder();

            if (order == null)
            {
                order = orderHome.create(customer);
                order.setCustomer(customer);

                customer.setCurrentOrder(order);
            }

            order.setFromData(data);
        }
        catch (CreateException exc)
        {
            exc.printStackTrace();
```

```
            throw new EJBException("Error updating order: "+exc.toString());
        }
    }
    public String saveOrder()
        throws RemoteException, EJBException
    {
        if (customer == null)
        {
            throw new EJBException("No order to save - no customer!");
        }

        Order order = customer.getCurrentOrder();

        if (order == null)
        {
            throw new EJBException("No order to save");
        }

        return order.place();
    }

    public OrderHistoryView[] getOrderHistory()
        throws RemoteException
    {
        if (customer == null)
        {
            return null;
        }

        Collection orders = customer.getOrders();

        ArrayList viewList = new ArrayList();

        Iterator iter = orders.iterator();

        while (iter.hasNext())
        {
            Order order = (Order) PortableRemoteObject.narrow(
                iter.next(), Order.class);

            if (order.getStatus() == null) continue;

            viewList.add(order.getHistoryView());
        }

        OrderHistoryView[] views = new OrderHistoryView[viewList.size()];

        return (OrderHistoryView[]) viewList.toArray(views);
    }
```

```
public OrderView getOrder(String orderId)
    throws RemoteException
{
    try
    {
        if (customer == null) return null;

        Order order = orderHome.findByPrimaryKey(orderId);

        if (order.getCustomer().getCustomerId().equals(
            customer.getCustomerId()))
        {
            return order.getView();
        }

        return null;
    }
    catch (FinderException exc)
    {
        return null;
    }
}

public boolean isEmpty()
        throws RemoteException
    {
        if (customer == null) return true;

        if (customer.getCurrentOrder() == null)
        {
            return true;
        }

        return customer.getCurrentOrder().isEmpty();
    }
}
```

10.7.5 生成 Web 接口

在有了实体 Bean 和会话 Bean 后，就可以生成应用程序的 Web 接口了。这个 Web 应用程序使用小服务与 JSP 的组合，用简单的模型—视图—控制器的设置。这里的小服务作为控制器，将浏览器请求转换成对象模型的改变，然后小服务将请求转换到 JSP 中绘制视图。

尽管这个应用程序不用定制标记，但生成了绘制应用程序特定部分的 JSP。例如，主页面显示用户、产品目录和购物车略图的信息，这 3 个数据视图分别用 JSP 页面实现。从下面的主 Shopping.jsp 页面中可以看出它并没有做太多的工作。

Shopping.jsp 的代码如下所示：

```
<html>
<body bgcolor="#ffffff">
```

```
<img src="logo.jpg">
<table border="0">
<tr>
<td><jsp:include page="CartDisplay.jsp"/></td>
<td>
<h1>Welcome to The Store</h1>
<p>
<jsp:include page="UserInfo.jsp"/>
<p>
<jsp:include page="Catalog.jsp"/>
</td>
</tr>
</table>
</body>
</html>
```

下面的 CartDisplay.jsp 清单显示了购物车的简化版本。这个页面假设总是放在另一个页面中，因此不包含<html>或<body>标记。

CartDisplay.jsp 的代码如下所示：

```
<%@ page import="usingj2ee.shopping.*" %>
<jsp:useBean id="cart" class="usingj2ee.shopping.CartBean" scope="session"/>
<%
    if (cart.getCart(request).isEmpty())
    {
%>
<p>Your shopping cart is empty.</p>
<%
    }
    else
    {
        OrderView order = cart.getCart(request).getCurrentOrder();

        out.println("<p>Your shopping cart contains:</p>");
        out.println("<table border=\"0\">");

        for (int i=0; i < order.lineItems.length; i++)
        {
            out.print("<tr><td>"+order.lineItems[i].name+"</td></tr>");
        }
        out.println("</table>");
        out.println("<a href=\"ShoppingCart.jsp\">View Shopping Cart</a>");
    }
%>
```

下面的 CartBean.java 清单显示了 CartBean 类，它实现了 HttpSessionBindingListener 接口，并探测何时从当前会话删除。EJB 与 Web 应用程序的另一个问题是客户端的本地代理不知道 HttpSession 类（也不应知道）。在会话消失时，会话 Bean 可能继续存在一段时间之后，EJB 容器才能知道客户已经消失。CartBean 类负责解决这个问题。

CartBean.java 的代码如下所示：

```
package usingj2ee.shopping;
```

```
import java.rmi.*;
import javax.ejb.*;
import javax.rmi.*;
import javax.naming.*;
import javax.servlet.http.*;

public class CartBean implements java.io.Serializable,
    HttpSessionBindingListener
{
    protected ShoppingCartSession cart;

    public CartBean()
    {
    }

    public ShoppingCartSession getCart(HttpServletRequest request)
    {
        if (cart == null)
        {
            synchronized(this)
            {
                if (cart == null)
                {
                    try
                    {
                    Context ctx = new InitialContext();

                    ShoppingCartSessionHome home =
                        (ShoppingCartSessionHome) PortableRemoteObject.
                        narrow(ctx.lookup("ShoppingCartSessionHome"),
                        ShoppingCartSessionHome.class);

                    String customerId = null;

                    Cookie cookies[] = request.getCookies();
                    for (int i=0; i < cookies.length; i++)
                    {
                        if (cookies[i].getName().equals(
                            "StoreCustomerId"))
                        {
                            customerId = cookies[i].getValue();
                            break;
                        }
                    }
                    if (customerId == null)
                    {
                        cart = home.create();
                    }
                    else
```

```
                {
                    cart = home.create(customerId);
                }
            }
            catch (NamingException exc)
            {
                exc.printStackTrace();
            }
            catch (CreateException exc)
            {
                exc.printStackTrace();
            }
            catch (RemoteException exc)
            {
                exc.printStackTrace();
            }
        }
    }

    return cart;
}

public void valueBound(HttpSessionBindingEvent evt)
{
}

public void valueUnbound(HttpSessionBindingEvent evt)
{
    if (cart != null)
    {
        try
        {
            cart.remove();
        }
        catch (Exception exc)
        {
            exc.printStackTrace();
        }
    }
}
```

由于小服务也要访问购物车会话，因此要使用与 JSP 页面相同的 CartBean 实例。下面的 AddToCartServlet.java 清单显示了将产品加进购物车的小服务。

AddToCartServlet.java 的代码如下所示：

```
package usingj2ee.shopping;

import usingj2ee.shopping.*;
import java.io.*;
```

```
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class AddToCartServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException
    {
        processRequest(request, response);
    }

    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException
    {
        processRequest(request, response);
    }

    public void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException
    {
        HttpSession session = request.getSession();
        CartBean cart = (CartBean) session.getAttribute("cart");

        if (cart == null)
        {
            cart = new CartBean();
            session.setAttribute("cart", cart);
        }

        String productId = request.getParameter("productId");

        ShoppingCartSession cartSession = cart.getCart(request);

        cartSession.addToShoppingCart(productId, 1);

        Cookie cookie = new Cookie("StoreCustomerId",
            cartSession.getCustomerId());
        cookie.setPath(request.getContextPath());
        cookie.setMaxAge(7776000);

        response.addCookie(cookie);

        response.sendRedirect(request.getContextPath()+"/Shopping.jsp");
    }
}
```

Web 应用程序中使用模型—视图—控制器机制的一个难点是，控制器不一定总是与视图分开。例如，AddToCartServlet 类在进行操作之后要显示 Shopping.jsp 视图页面，但控制器并不知道要显示哪个视图。因此，需要在不同情形中不经改变而复用控制器。

购物应用程序处理登录和注册时会遇到这种情形。显示订单历史或下订单时，需要登录。如果还没有帐号，则要注册。假设用户要结账以完成当前订单时，如果用户还没有登录，则要将登录窗体发给用户。如果在登录窗体中，用户指定自己是新用户，则要将注册窗体发给用户。登录和注册完成之后，让用户返回结账页面。

为了解决这个问题，购物应用程序将改向 URL 传递给登录和注册小服务，告诉它操作完成后将用户返回什么地方。这样，就不必在每次增加需要登录的新特性时改写登录和注册小服务。

下面的 LoginServlet.java 清单显示了 LoginServlet 类，可以看出，它自动探测将用户返回什么地方，但没有提供其他 URL 时，默认到 Shopping.jsp。

LoginServlet.java 的代码如下所示：

```
package usingj2ee.shopping;

import usingj2ee.shopping.*;
import java.io.*;
import java.net.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class LoginServlet extends HttpServlet
{
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException
    {
        processRequest(request, response);
    }

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

    public void processRequest(HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException
    {
        String hasAccount = request.getParameter("account");

        if ((hasAccount != null) && (hasAccount.equals("no")))
        {
            getServletContext().getRequestDispatcher(
```

```
            getContextPath()+"/Register.jsp").
            forward(request, response);
        return;
    }

    String email = request.getParameter("email");
    String password = request.getParameter("password");

    HttpSession session = request.getSession();

    CartBean cart = (CartBean) session.getAttribute("cart");

    if (cart == null)
    {
        cart = new CartBean();
        session.setAttribute("cart", cart);
    }

    String customerId = cart.getCart(request).login(email, password);

    if (customerId == null)
    {
        getServletContext().getRequestDispatcher(
            getContextPath()+"/LoginFailed.html").
            forward(request, response);
        return;
        }

        session.setAttribute("loggedIn", "true");

        Cookie cookie = new Cookie("StoreCustomerId",
            cart.getCart(request).getCustomerId());
        cookie.setPath(request.getContextPath());
        cookie.setMaxAge(7776000);

        response.addCookie(cookie);

        String redirectURL = request.getParameter("redirectURL");

        if (redirectURL != null)
        {
            response.sendRedirect(request.getContextPath()+redirectURL);
        }
        else
        {
            response.sendRedirect(request.getContextPath()+"/Shopping.jsp");
        }
    }
}
```

10.7.6　部署应用程序

这个购物应用程序需要使用 EJB 2.0。EJB 部署比较简单，Bean 没有任何特殊的安全要求。但 Web 应用程序有一个小小的要求：JSP 和小服务应部署为 Web 应用程序。具体地说，JSP 假设小服务是应用程序的一部分，不用常见的/servlet/前缀引用小服务。这个设计的好处是小服务不需要指定 JSP 的完整路径，可以假设它们共享相同的环境路径。

下面的清单显示了这个应用程序的 web.xml 文件。注意文件中不需要包括 JSP，但要包括使用的所有小服务。

```
<?xml version="1.0" ?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
1.2//EN" "http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
<web-app>
    <display-name>cart</display-name>
    <servlet>
        <servlet-name>AddToCartServlet</servlet-name>
        <servlet-class>usingj2ee.shopping.AddToCartServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>LoginServlet</servlet-name>
        <servlet-class>usingj2ee.shopping.LoginServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>RegisterServlet</servlet-name>
        <servlet-class>usingj2ee.shopping.RegisterServlet</servlet-class>
    </servlet>
    <servlet>
        <servlet-name>SaveOrderServlet</servlet-name>
        <servlet-class>usingj2ee.shopping.SaveOrderServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>AddToCartServlet</servlet-name>
        <url-pattern>AddToCartServlet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>LoginServlet</servlet-name>
        <url-pattern>LoginServlet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>RegisterServlet</servlet-name>
        <url-pattern>RegisterServlet</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>SaveOrderServlet</servlet-name>
        <url-pattern>SaveOrderServlet</url-pattern>
    </servlet-mapping>
</web-app>
```

10.8 小结

本章全面讨论了 J2EE 体系结构中的 3 种 EJB，并在最后给出了一个综合的例子。

对于会话 Bean 有以下要点。

- 会话 Bean 代表一个过程、任务或工作流。
- 存在两种会话 Bean：有状态会话 Bean 和无状态会话 Bean。
- 有状态会话 Bean 可能以伸缩性为代价，以简化应用程序开发。
- 无状态会话 Bean 可以向客户端或其他 Bean 提供服务。
- 环境变量和外部资源可以在部署描述器中声明，并且通过 JNDI 名字空间访问。

对于实体 Bean 有以下要点。

- 在应用程序中，实体 Bean 表示共享的、事务性数据。
- 实体 Bean 的持久化可以由 Bean 开发者或者 EJB 容器管理。
- EJB 容器管理的持久化可以提升效率，但是可能不具备满足开发者需要的功能。
- Bean 管理的持久化强迫 Bean 开发人员承担从对象至数据存储的状态转换负担。

对于消息驱动 Bean 有以下要点。

- 消息驱动 Bean 不包含本地或者远程接口。其处理过程在 onMessage()方法中完成。
- 消息驱动 Bean 可能使用由 Bean 管理的持久化或者由 EJB 容器管理的持久化。
- 消息驱动 Bean 不包含客户端安全性上下文。

第 11 章　CORBA 以及 Java IDL 编程范例

CORBA（Common Object Request Broker Architecture，公共对象请求代理体系结构）是由 OMG（Object Management Group，对象管理组织）提出的应用软件体系结构和对象技术规范，其核心是一套标准的语言、接口和协议，以支持异构分布应用程序间的互操作性及独立于平台和编程语言的对象重用。

近年来，随着互联网技术的日益成熟，公众及商业企业正享受着高速、低价网络信息传输所带来的高品质数字生活。但是，由于网络规模的不断扩大以及计算机软硬件技术水平的飞速提高，给传统的应用软件系统的实现方式带来了巨大挑战。

首先，在企业级应用中，硬件系统集成商基于性能、价格、服务等方面的考虑，通常在同一系统中集成来自不同厂商的硬件设备、操作系统、数据库平台和网络协议等，由此带来的异构性给应用软件的互操作性、兼容性以及平滑升级能力带来了严重问题。

另外，随着基于网络的业务不断增多，传统的客户机/服务器（C/S）模式的分布式应用方式越来越显示出在运行效率、系统网络安全性和系统升级能力等方面的局限性。

为了解决分布式计算环境（DCE，Distributed Computing Environment）中不同硬件设备和软件系统的互联、增强网络间软件的互操作性以及解决传统分布式计算模式中的不足等问题，对象管理组织（OMG）提出了公共对象请求代理体系结构（CORBA），以增强软件系统间的互操作能力，并且使构造灵活的分布式应用系统成为可能。

正是基于面向对象技术的发展和成熟、客户/服务器软件系统模式的普遍应用以及集成已有系统等方面的需求，推动了 CORBA 技术的成熟与发展。作为面向对象系统的对象通信的核心，CORBA 为当今网络计算环境带来了真正意义上的互联。

11.1　介绍 CORBA

11.1.1　对象管理组织（OMG）简介

OMG 成立于 1989 年，作为一个非营利性组织，它致力于开发在技术上具有先进性、在商业上具有可行性并且独立于厂商的软件互联规范，推广面向对象模型技术，增强软件的可移植性（Portability）、可重用性（Reusability）和互操作性（Interoperability）。

11.1.2　CORBA 主要版本的发展历程

1990 年 11 月，OMG 发表《对象管理体系指南》，初步阐明了 CORBA 的思想。

1991 年 10 月，OMG 推出 1.0 版，其中定义了接口定义语言（IDL）、对象管理模型以及基于动态请求的 API 和接口仓库等内容；1991 年 12 月，OMG 推出了 CORBA 1.1 版，在澄清了 1.0 版中存在的二义性的基础上，引入了对象适配器的概念；1996 年 8 月，OMG 基于以前的升级版本，完成了 2.0 版的开发，该版本中重要的内容是对象请求代理间协议（IIOP，

Internet Inter-ORB Protocol）的引入，实现了不同厂商的 ORB 真正意义上的互通；1998 年 9 月，OMG 发表了 CORBA 2.3 版，增加了支持 CORBA 对象的异步实时传输、服务质量规范等内容。

目前，宣布支持 CORBA 2.3 规范的中间件厂商包括 Inprise（Borland）、Iona、BEA System 等著名的 CORBA 产品生产商。

11.1.3 CORBA 体系结构概述

CORBA 规范充分利用了现今软件技术发展的最新成果，在基于网络的分布式应用环境下实现应用软件的集成，使得面向对象的软件在分布、异构环境下实现可重用、可移植和互操作。其特点可以总结为如下几个方面：

- 引入中间件（MiddleWare）作为事务代理，以完成客户机（Client）向服务对象方（Server）提出的业务请求（引入中间件概念后，分布计算模式如图 11-1 所示）。

图 11-1 引入中间件的分布计算模式

- 实现客户与服务对象的完全分开，客户不需要了解服务对象的实现过程以及具体位置（参见图 11-2 所示的 CORBA 系统体系结构）。

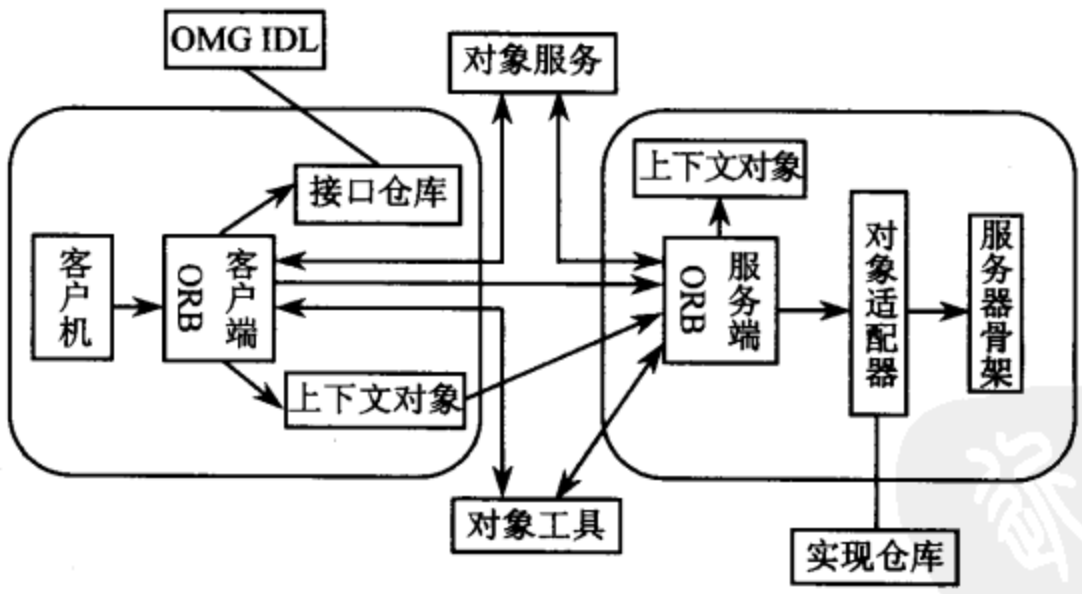


图 11-2 CORBA 系统体系结构

- 提供软总线机制，使得在任何环境下、采用任何语言开发的软件只要符合接口规范的定义，均能够集成到分布式系统中。
- CORBA 规范软件系统采用面向对象的软件实现方法开发应用系统，实现对象内部细节的完整封装，保留对象方法的对外接口定义。

在以上特点中，最突出的是中间件的引入，在 CORBA 系统中称为对象请求代理（ORB，

Object Request Broker）和采用面向对象的开发模式。

对象模型是应用开发人员对客观事物属性和功能的具体抽象。由于 CORBA 使用了对象模型，将 CORBA 系统中所有的应用看成是对象及相关操作的集合，因此通过对象请求代理（ORB），使 CORBA 系统中分布在网络中应用对象的获取只取决于网络的畅通性和服务对象特征获取的准确程度，而与对象的位置以及对象所处的设备环境无关。

11.1.4　CORBA 的主要应用方向及中间件产品介绍

CORBA 规范的推出，重新调整了客户机与服务器之间的关系。客户机可以向服务器提出事务请求，同时也可以为下一个请求充当服务器角色。

由于 CORBA 系统引入了中间件的概念即事务代理，由中间件完成客户机与服务器之间的通信，使得服务器对于客户机的位置相对透明，取消了原有分布式计算模型中客户机、服务器之间的一一对应关系。CORBA 客户机可以在运行时动态获得服务对象的位置，并且可以对多个服务对象提交事务请求，因此，它极大推动了分布计算的发展。

分布式计算是指网络中两个或两个以上的软件相互共享信息资源。这些软件可以位于同一台计算机中，也可以部署在网络节点的任意位置。基于分布式模型的软件系统具有均衡运行系统负载、共享网络资源的技术优势。

另外，CORBA 规范约束采用面向对象的分布式软件的构造方法，以接口定义语言的形式实现对象内部细节的完整封装，从而降低了软件系统的复杂程度，增加了软件功能的可重用性。CORBA 提供到 C/C++、Java、SmallTalk 等高级语言的映射，很大程度上减小了对程序设计语言的依赖性，使软件开发人员可以在较大范围内共享已有成果。

正是以上特点推动了分布式多层软件体系结构的发展。目前，CORBA 技术在银行、电信、保险、电力和电子商务领域都有广泛的应用。

软件市场中能够见到的 CORBA 中间件产品很多，但基于不同公司的产品战略以及研发方向，各个产品在服务性能、对高级语言的支持和所依赖的系统平台方面有很大区别。读者应该根据具体情况选择。

11.2　介绍 IDL

11.2.1　OMG IDL 文件概述

从本质上讲，OMG IDL 接口定义语言不是作为程序设计语言体现在 CORBA 体系结构中的，而是用来描述产生对象调用请求的客户对象和服务对象之间的接口的语言。OMG IDL 文件描述数据类型和方法框架，而服务对象则为一个指定的对象实现提供上述数据和方法。

OMG IDL 文件描述了服务器提供的服务功能，客户机可以根据该接口文件描述的方法向服务器提出业务请求。在大多数 CORBA 产品中都提供 IDL 到相关编程语言的编译器。程序设计人员只需将定义的接口文件输入编译器，设定编译选项后，就可以得到与程序设计语言相关的接口框架文件和辅助文件。IDL 文件应用过程如图 11-3 所示。

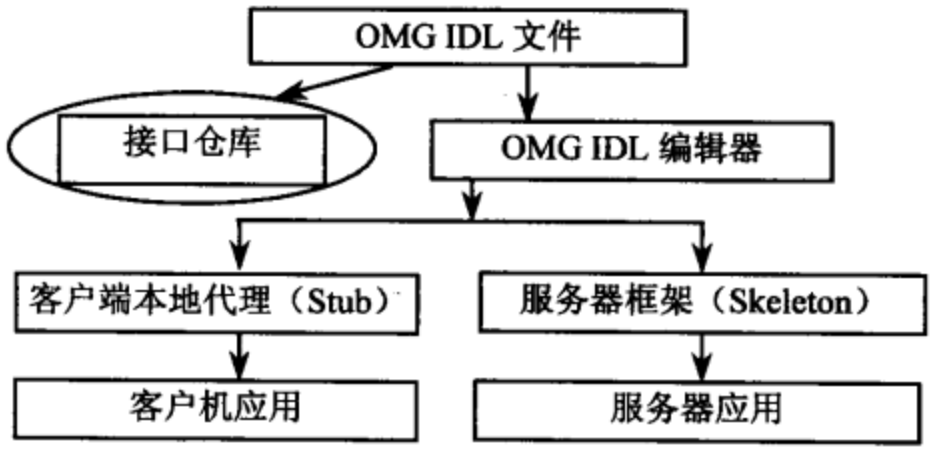


图 11-3　IDL 文件的应用过程

11.2.2　Java IDL 介绍

Java IDL 是 Java 2 开发平台中的 CORBA 功能扩展。在 Java 2 中引入 Java IDL，使得利用 OMG IDL 能够定义服务对象的基本功能，并且将 IDL 根据 CORBA 规范的要求，映射到 Java 语言，并以此开发出标准的具有互操作性和可连接性的分布式应用。Java IDL 使分布式、支持 Web 的 Java 应用可以基于 IIOP 协议透明地调用远程服务。

Java IDL 运行时（Runtime）组件包括一个全兼容的对象请求代理——Java ORB，它可用于基于 IIOP 协议实现分布式对象之间的通信。该 ORB 支持瞬态 CORBA 对象和瞬态名称服务器，并且 ORB 生存期受运行 ORB 进程生存期的限制。

11.3　开发 CORBA 应用的服务器程序

CORBA 对象服务的实现方式分为两种：对象的命名引用方式和字符串化对象引用方式。不论采用何种高级语言，创建 CORBA 应用程序的过程大体如下：

（1）进行系统分析，确定服务对象需要提供的功能。

（2）根据分析结果，编写 IDL 接口说明文件。

（3）编译接口说明文件，产生服务对象的骨架与客户对象的本地代理（可选）。

（4）基于服务器框架，编写服务对象实现程序。

（5）基于客户端本地代理，编写客户对象调用程序。

（6）分别编译客户对象和服务对象程序。

（7）启动服务对象程序。

（8）启动客户对象程序。

下面通过一个实例，描述如何通过 Java 创建 CORBA 应用程序。本实例将在服务对象端对一个字符串对象赋值，客户端通过调用该服务对象方法获取该字符串的值。本节将讲述服务器端编程，而客户端编程将在下一节讲述。

（1）服务对象接口定义。用 IDL 编写出服务对象方法 getMessage.idl 的描述程序。

```
module getMessage
{
    interface getIt
    {
```

```
        string returnString(in string str);
    };
};
```

（2）编译 getMessage.idl。Java JDK 提供了对 CORBA 的支持，Java IDL，即 idlj 编译器（jdk1.3.0_01 以上版本）就是一个 ORB，可用来在 Java 语言中定义、实现和访问 CORBA 对象。Java IDL 支持的是一个瞬间的 CORBA 对象，即在对象服务器处理过程中有效。实际上，Java IDL 的 ORB 只是一个类库而已，并不是一个完整的平台软件，但它对 Java IDL 应用系统和其他 CORBA 应用系统之间提供了很好的底层通信支持，并且实现了 OMG 定义的 ORB 基本功能。

可以执行下面命令为接口定义文件生成客户端本地代理和服务器框架：

```
idlj -f all getMessage.idl
```

-f 参数指明生成内容，all 包括客户端本地代理和服务器框架，这样在当前目录下会生成一个 getMessage 目录。其目录下将包括下列文件：_getItStub.java、getIt.java、getItHelper.java、getItHolder.java、getItOperations.java 和 getItPOA.java 或_getItImplBase.java（根据 JDK 使用的版本不同而不同）。

（3）编写服务对象程序。GetItImpl.Java 的代码如下所示：

```
// 引入相关类库
import GetMessage.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

class GetItImpl extends GetItPOA
{
    private ORB orb;

    public void setORB(ORB orb_val)
    {
        orb = orb_val;
    }

    public String returnString(String str)
    {
        return "\nHello " + str;
    }
}
```

server.java 的代码如下所示：

```
public class server
{
    public static void main(String args[])
    {
        try
        {
            // 创建和初始化 ORB
```

```
            ORB orb = ORB.init(args, null);

            // 取得对 rootpoa 对象的引用并激活 POAManager
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references
("RootPOA"));
            rootpoa.the_POAManager().activate();

            // 创建服务对象并将其向 ORB 注册
            GetItImpl getItImpl = new GetItImpl();
            getItImpl.setORB(orb);

            // 取得对象引用
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(getItImpl);
            GetIt href = GetItHelper.narrow(ref);

            // 获取根命名上下文
            // NameService 调用名称服务
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");

            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);

            // 绑定命名中的对象引用
            String name = "GetIt";
            NameComponent path[] = ncRef.to_name( name );
            ncRef.rebind(path, href);

                System.out.println("GetItServer ready and waiting ...");

            // 等待来自客户机的调用
            orb.run();
        }
        catch (Exception e)
        {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
        System.out.println("GetItServer Exiting ...");
    }
}
```

11.4 开发 CORBA 应用的客户机程序

下面是客户机程序 client.java 的代码：

```
//引入相关类库
import GetMessage.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
```

```
    //客户端对象方法
    public class client
    {
        static GetIt getItImpl;

        public static void main(String args[])
        {
            try
            {
                // 创建和初始化 ORB
                ORB orb = ORB.init(args, null);
                // 获取根命名服务上下文对象
                org.omg.CORBA.Object  objRef  =  orb.resolve_initial_references
("NameService");

                NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
                //解析命名中的对象引用，注意这里的名称和服务器中的相同
                String name = "GetIt";
                getItImpl = GetItHelper.narrow(ncRef.resolve_str(name));

                System.out.println("Obtained a handle on server object: ");
                System.out.println(getItImpl.returnString(args[0]));
            }
            catch (Exception e)
            {
                System.out.println("ERROR : " + e) ;
                e.printStackTrace(System.out);
            }
        }
    }
```

按照以下步骤来运行该程序：

（1）启动命名服务，其中 1050 为通信端口号。

```
start orbd -ORBInitialPort 1050
```

（2）运行服务端程序。

```
start java server -ORBInitialPort 1050 -ORBInitialHost localhost
```

（3）运行客户端程序。

```
java client arg0 -ORBInitialPort 1050 -ORBInitialHost localhost
```

11.5　编写 CORBA 客户机 applet

现在，让研究将 CORBA 客户机编写为一个 applet 而不是独立应用程序时的差异。这里是一些基本的差异：

- 通常在 applet 的 init 方法而不是 main 中编码 CORBA 初始化。
- 对于 applet，CORBA.init 方法有一点不同。第一个参数是指向 applet 本身的 this 指针。

尽管存在这些差异，applet 客户机中与 CORBA 相关的代码和应用程序中的基本相同。这

里将仍然使用 CORBA 命名服务来查找远程引用，并将返回的引用限制成正确类型等。

applet 和应用程序之间的另一个差异是要从浏览器内使用 applet，因此需要编写引用 applet 的 HTML 页面。并且事实证明，CORBA applet 的 HTML 页面显得很麻烦，我们将在后面看到。

以下的 MeetingClientApplet.java 代码显示了编写 CORBA 客户机 applet 的模式：

```
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class MeetingClientApplet extends java.applet.Applet
{
  public void init ()
  {
    try
    {
      //1.初始化 ORB 和名称服务
      //2. 在名称服务中查找远程对象
      //3. 调用远程方法（如果保存了引用，也可以从其他方法来调用）
    }
    catch ( Exception e )
    {
    }
  }
}
```

下面是用于 applet 的 HTML 页面：

```
<HTML>
<title>Meeting Client Applet</title>
<center><h1>Meeting Client</h1></center>
The message from the MeetingServer is:
<p>
<OBJECT classid-"clsid:ZAD9C240-044E-1101-B3E9-00Z05F499D93"
WIDTH - 500 HEIGHT - 120
codebase-"http://java.sun.com/productions/plugin/1.1.1/
  jinstall-111-win32.eaVersion-1,1,1,0">

<PARAM NAME-"type" VALUE-"application/x-java-applet;version-1.1">
<COMMENT>
<EMBED type="application/x-java-applet;version-1.1"
java_CODE - "MeetingClientApplet" WIDTH - 500 HEIGHT - 120
pluginapage-"http://java.sun.com/products/plugin/1.1.1/plugin-install.html">
<NOEMBED></COMMENT>
</NOEMBED></EMBED>
</OBJECT>
</HTML>
```

有的浏览器可能并不支持 JDK 1.2，而 JDK 1.2 是使用 Java IDL 必需的。为了解决这个问题，Sun 创建了一个浏览器插件应用程序，来替代用作浏览器的普通 JVM 的 JDK 1.2 虚拟机。因此，在运行 CORBA applet 之前，需要从 javasoft 网站下载该插件并配置浏览器以使用它。然后可以运行前面提到的转换器来创建 HTML，这样就激活了插件。

这样做也许太麻烦了！好消息是不必编写这一切（除非真的喜欢做这种事情）。相反，可以编写引用 CORBA 客户机 applet 的“普通”HTML 页面，然后运行转换器程序，Java 2 SDK 的标准版（J2SE）1.3 中包含了这个程序。

11.6　使用 CORBA 范例：Java 和 C++混合编程

通过前面对 CORBA 的学习，大家都已经了解到 CORBA 是一个完全中间性的语言，可以使用接口定义语言（IDL）定义开发时使用接口的 Client 和实现接口的 Server 所需要的信息。Client 和 Server 的具体实现代码并不在 IDL 定义中编写，而是使用某种目标语言的 IDL 编译器生成所需代码的本地代理及 helper 类，Client 和 Server 再使用真正的编程语言进行具体实现。

为了保证在不同的 CORBA 产品基础之上构建的分布式对象可以相互通信，Client 和 Server 通过 ORB（对象请求代理）进行通信。一般的运行流程是 Client 把请求发送给 ORB，ORB 再把请求发送给 Server；然后 Server 把返回结果发送 ORB，ORB 再把返回结果发送给 Client。ORB 可以说是 Client 和 Server 之间的翻译者。即使 Client 和 Server 使用不同的编程语言编写，只要是符合相同的 IDL 定义，ORB 也可以完成相互的通信。

本节例子中的服务器用 Java 编写，客户机分别用 Java 和 C++（VC6+omniORB）编写。Java 和 C++混合编程的步骤和前述的编写步骤大致相同，总体的编写过程如下：

（1）用 IDL 定义一个接口文件，以描绘要实现的功能，也可以说是定义一个要实现功能的模板。

（2）使用 IDL to Java 编译器（这里是 idlj）将 IDL 文件转化为 Java 编程语言中编写的接口定义，生成所需的代码的本地代理及 helper 类。

（3）使用 Java 语言编写客户机和服务器的实现程序。

（4）使用 IDL to C++编译器（这里是 omniidl）将 IDL 文件转化为 C++编程语言中编写的接口定义，生成所需代码的本地代理及 helper 类。

（5）使用 C++语言编写客户机实现程序（当然也可编写服务器程序）。

（6）启动命名服务 tnameserv。

（7）启动 Java 编写的服务程序。

（8）用 Java 和 C++编写的客户机分别调用相应的服务。

下面一步一步地讲述实现过程。

11.6.1　编写 SysProp.idl

该 IDL 文件描述了一个方法，功能是返回系统属性。SysProp.idl 文件的内容如下所示：

```
interface SysProp
{
    string getProperty(in string name);
};
```

11.6.2　编写 Java 的服务器程序

首先需要把 IDL 文件转化为 Java 编程语言代码的本地代理类及 helper 类。需要执行如下

命令：

```
idlj -fall SysProp.idl
```

此时，在该目录内将生成文件：SysProp.java、SysPropHelper.java、SysPropHolder.java、SysPropOperations.java、SysPropPOA.java、_SysPropStub.java。

然后编写服务器程序 SysPropServer.java，下面是它的代码：

```
import SysProp.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;
import org.omg.PortableServer.*;
import org.omg.PortableServer.POA;

class SysPropImpl extends SysPropPOA
{
    private ORB orb;

    public void setORB(ORB orb_val)
    {
        orb = orb_val;
    }

    public String getProperty(String key)
    {
        System.out.println("调用"+key);
        String S;
        S=System.getProperty(key);
        if (S==null)
        {
            S="null";
        }
        System.out.println(key+"="+S);
        return S;
    }
}
public class SysPropServer
{
    public static void main(String args[])
    {
        try
        {
            // 创建和初始化 ORB
            ORB orb = ORB.init(args, null);

            // 取得对 rootpoa 对象的引用并激活 POAManager
            POA rootpoa = POAHelper.narrow(orb.resolve_initial_references
("RootPOA"));
            rootpoa.the_POAManager().activate();
            // 创建服务对象并将其向 ORB 注册
            SysPropImpl sysPropImpl = new SysPropImpl();
```

```
            sysPropImpl.setORB(orb);
            // 取得对象引用
            org.omg.CORBA.Object ref = rootpoa.servant_to_reference(sysPropImpl);
            SysProp href = SysPropHelper.narrow(ref);
            // 获取根命名上下文
            // NameService 调用名称服务
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContextExt ncRef = NamingContextExtHelper.narrow(objRef);
            // 绑定命名中的对象引用
            String name = "SysProp";
            NameComponent path[] = ncRef.to_name( name );
            ncRef.rebind(path, href);
            System.out.println("SysPropServer ready and waiting ...");
            // 等待来自客户机的调用
            orb.run();
        }

        catch (Exception e)
        {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
        System.out.println("GetItServer Exiting ...");
    }
}
```

11.6.3　编写 Java 的客户机

接下来先编写一个 Java 的客户机。下面是客户端 SysPropClient.java 的代码：

```
import SysProp.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class SysPropClient
{
    public static void main(String[] args)
    {
        try
        {
            String SetInfo,ReturnInfo,ref;
            org.omg.CORBA.Object objRef;
            SysProp syspropRef;
            ORB orb = ORB.init(args, null);
            //使用 IOR 的方法开始
            if (args.length>=1)
            {
                ref=args[0];
            }
            else
```

```
{
    System.out.println("SysPropClient <IOR 字符串>  [环境变量]");
    return;
}
objRef = orb.string_to_object(ref);
syspropRef = SysPropHelper.narrow(objRef);
//使用 IOR 的方法结束

/*
//使用 ORB 的方法的开始
objRef = orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
// 进行服务定位
NameComponent nc = new NameComponent("SysProp", "");
NameComponent path[] = {nc};
syspropRef = SysPropHelper.narrow(ncRef.resolve(path));
//使用 ORB 的方法的结束
*/
if (args.length>1)
{
    SetInfo=args[1];
}
else
{
    SetInfo="java.home";
}
            System.out.println("开始调用");
            ReturnInfo = syspropRef.getProperty(SetInfo);
            System.out.println(SetInfo+"="+ReturnInfo);
        }
        catch (Exception e)
        {
            System.out.println("ERROR : " + e) ;
        }
    }
}
```

注意在代码内有一段注释掉的代码，以“//使用 ORB 的方法的开始”开始，以“//使用 ORB 的方法的结束”结束。这段代码是使用 ORB 方法的代码，如果在代码中“//使用 IOR 的方法开始”前一行添加“/*”，在“//使用 IOR 的方法结束”后一行添加“*/”，而把“//使用 ORB 的方法的开始”前面的“/*”去掉，并把“//使用 ORB 的方法的结束”后面的“*/”去掉，就是使用 ORB 方法的代码，程序运行时就是“SysPropClient [环境变量]”的方式。

11.6.4 编写 C++的 IOR 客户机

从实践上来讲，编写 C++的客户机程序同 Java 并没有多大的区别，只不过 Java 是用 idlj 生成代码的本地代理类及 helper 类，而 omni 是用 omniidl 生成代码的本地代理类及 helper 类，而它们的编程思想及编码过程非常相似。需要先把 IDL 文件转化为 C++编程语言的代码的本地代理类及 helper 类。可以执行如下命令：

```
omniidl -bcxx SysProp.idl
```

在正常的情况下目录内将生成 C++编程语言的代码的本地代理类及 helper 类 SysProp.hh 和 SysPropSK.cc。否则，请检查执行程序及文件。

接下来需要编写客户机程序，下面是 IOR 客户机程序 SysPropC.cc 的代码：

```
#include <iostream.h>
#include <SysProp.hh>
int main(int argc, char** argv)
{
    try
    {
        CORBA::ORB_var orb = CORBA::ORB_init(argc, argv, "omniORB3");
        if( argc < 2 || argc > 3 )
        {
            cout << "usage:  SysPropC <IOR 字符串> [环境变量名]" << endl;
            return 1;
        }
        CORBA::Object_var obj = orb->string_to_object(argv[1]);
        SysProp_ptr echoref = SysProp::_narrow(obj);
        if( CORBA::is_nil(echoref) )
        {
            cerr << "没有对象" << endl;
            return 1;
        }
        const char* message;
        if (argc==3)
        {
            message=argv[2];
        }
        else
        {
            message="java.home";
        }
        CORBA::String_var dest = echoref->getProperty(message);
        cout <<  (char*)message << "=" <<(char*)dest << endl;
        orb->destroy();
    }
    catch(...)
    {
        cerr << "Caught unknown exception." << endl;
    }
    return 0;
}
```

下面来编写 make 文件 dirc.mak，如下面所示：

```
TOP = C:\omni
OMNI_DYNAMIC_LIB = msvcstub.lib -NODEFAULTLIB:libcmt.lib -NODEFAULTLIB:libcmtd.lib
CORBA_CPPFLAGS = -D__WIN32__ -D__x86__ -D__NT__ -D__OSVERSION__=4
CORBA_LIB      = omniORB304_rt.lib omnithread2_rt.lib \
                 $(OMNI_DYNAMIC_LIB) \
                 wsock32.lib advapi32.lib \
```

```
                -libpath:$(TOP)\lib\x86_win32
CXXFLAGS       = -O2 -MD -GX $(CORBA_CPPFLAGS) $(DIR_CPPFLAGS)
CXXLINKOPTIONS =
.SUFFIXES: .cc
.cc.obj:
  cl /nologo /c $(CXXFLAGS) /Tp$<
all:: SysPropC.exe
SysPropC.exe: SysPropSK.obj SysPropC.obj
  link -nologo $(CXXLINKOPTIONS) -out:$@ $** $(CORBA_LIB)
clean::
  -del *.obj
  -del *.exe
veryclean::
  -del *.obj
  -del echoSK.* echo.hh
  -del *.exe
SysProp.hh SysPropSK.cc: SysProp.idl
    $(TOP)\bin\x86_win32\omniidl   -T   -bcxx   -Wbh=.hh   -Wbs=SK.cc   -Wbtp
SysProp.idl
```

11.6.5 运行程序

为了方便，将设置环境参数写入一个批处理文件，该批处理文件内容如下：

```
set TOP=c:\omni
set path=%TOP%\bin\x86_win32;%path%
set LIB=%TOP%\bin\x86_win32;%LIB%
set INCLUDE=%TOP%\include;%INCLUDE%
set VcOsDir=
set VSCommonDir=
```

设置完环境参数后，然后编译源程序：

```
nmake -f dirc.mak
```

打开一个命令行窗口，执行以下命令：

```
tnameserv
```

再打开另一个命令行窗口，执行服务器程序：

```
java SysPropServer
```

再在另一个命令行窗口执行客户机程序：

```
SysPropC IOR:XXX JAVA.HOME
```

11.7 利用动态调用方式实现分布式应用

通常，基于客户端本地代理程序的情况下，需要预先知道被调用方法的名称、参数、返回值类型等信息。但是，往往实际应用中在设计客户端应用程序时，并不知道服务对象实现的具体细节，甚至服务对象实例还没有被创建。根据这种情况，CORBA 定义了另外一种客户对象调用服务对象实现方法的方式：客户端程序以动态调用接口（DII）方式实现，而服务对象以动态骨架接口（DSI）形式创建。这样，在分布式应用程序设计时，很大程度上减少了客户端应用程序创建时对服务对象的依赖程度，从而大大地提高了分布式应用程序设计的灵活性。

11.7.1　客户端动态调用接口（DII）

在动态调用方式下创建客户端和服务对象实现应用程序时，不再依靠本地代理程序和骨架程序对服务对象的引用。因此，在 IDL 到 Java 语言映射过程中需要指定映射工具的动态调用映射选项-dynamic_marshal，在 VisiBroker 中映射的方式为：

```
idl2java -dynamic_marshal filename.idl
```

在创建基于动态调用的分布式应用程序的客户端时，在客户端应首先创建请求（Request）对象，并在请求对象中指定服务对象中的方法名称以及应用的参数等。在创建请求对象后，以同步形式或异步形式查询服务对象的运行情况，并在服务对象运行结束后解析返回结果，从而结束动态调用请求。

在动态调用方式下，客户端必须向服务对象提出服务请求（Request）。Request 对象代表客户对象向服务对象提出调用请求。创建 Request 对象的方式有两种：一种是通过对调用对象的引用，用被引用对象的_request 方法创建 Request 对象，然后调用 Request 对象的 add_value 方法来添加方法调用需要的参数；另一种方式是调用被引用对象的_create_request 方法来创建对服务对象的调用请求。利用_request 方法创建请求对象的例子代码为：

```
org.omg.CORBA.Request request = serverObject._request("getName")
```

在上面的代码中创建的请求对象用于调用服务对象的名称为 getName 的方法。

服务请求由对服务对象的引用、要调用的服务对象方法名称和参数列表构成。请求的参数以元素列表的形式实现，其中的元素是 NamedValue 结构类型的实例，该结构定义如下：

```
typedef unsigned long Flags;
struct NamedValue
{
    Identifier name;
    any argument;
    long len;
    Flags arg_modes;
}
```

其中参数的意义分别为加入列表中的元素的名称、变量值、变量长度和参数的模式标识。

CORBA 为动态调用接口定义了很多方法，用于在客户端创建调用请求、查询返回结果和对返回结果进行解析等。下面介绍常用的客户端实现动态调用的方法。

- void create_request(in Context ctx,

in Identifier operation,

in NVList arg_list,

inout NamedValue result,

out Request request,

in Flags req_flags)

其中参数分别为调用请求应用的上下文对象、调用的方法名称、调用方法的参数列表、调用返回结果、新的返回请求和请求标志等。

- boolean poll_response(in Request req)

其中的参数 req 为用户创建的服务请求。该方法用于查询以异步形式向服务对象发送的服务请求的完成情况。返回 TRUE 说明请求已经完成，否则说明未完成。

- get_response() raises (WrongTransaction)

该方法用于查询以同步方式向服务对象发送的服务请求的完成情况。方法返回调用请求的结果。如果服务对象端的被调用方法没有完成，则 get_response 方法一直等待，直至请求完成。类似的方法还有 get_next_response，用于在客户端发送多个调用请求的情况下查询下一个请求的执行情况。

11.7.2 服务对象动态骨架接口（DSI）

服务对象动态骨架接口（DSI）位于服务对象端，用于动态处理客户端提出的服务请求。动态骨架接口通过对象适配器接收客户端服务请求，并将该请求传递给服务对象实现。

在处理客户请求时，服务对象实现不去分析客户是使用本地代理方法还是动态调用接口来提出对象服务请求。同时，发出调用的客户程序也不去分辨服务对象接收服务请求是不是通过服务对象骨架。

服务对象端 CORBA 对象结构如图 11-4 所示。

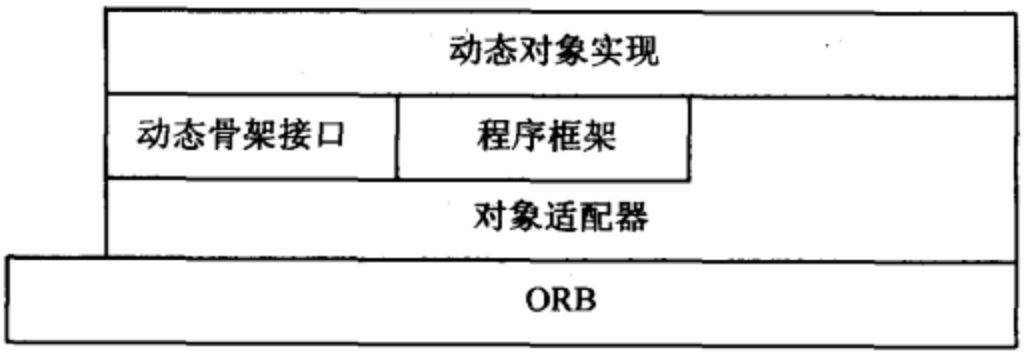


图 11-4 服务对象端 CORBA 对象结构

基于动态骨架接口的服务对象实现程序的编写步骤为：

（1）以 DynamicImplementation 接口为父对象创建服务对象实现程序。

（2）实现 DynamicImplementation 接口的 invoke 方法，应用程序接收客户对象提出的服务对象请求。

（3）通过 narrow 方法获得对根 POA 的引用。

（4）通过根 POA 的 create_POA 方法创建服务 POA（POA Servant），在服务 POA 中指定服务对象方法的实现。

在创建服务对象实现程序时，必须实现抽象类 DynamicImplementation 中的 invoke 等方法。该类的定义为：

```
package org.omg.CORBA;
public abstract class DynamicImplementation
extends org.omg.CORBA.portable.ObjectImpl
{
    public abstract void invoke(org.omg.CORBA.ServerRequest request);
}
```

可以看出 DynamicImplementation 抽象类中定义了抽象方法 invoke。该方法用于在服务对象实现程序中处理客户对象提出的服务请求。在该方法中，首先分析 request 实例中的调用方法名称，分析客户提出的方法在服务对象实现端是否有定义。如果没有定义，则抛出 BAD_OPERATION 异常。其代码为：

```
if (!request.operation().equals("getName"))
{
    throw new org.omg.CORBA.BAD_OPERATION();
}
```

上面的代码中，通过 request 对象的 operation 方法，分析需要调用的方法名称。因此，可以在程序中判断该方法是否在服务对象端有实现。在 request 请求的方法名称正确的情况下，服务对象实现端应用程序调用适合的方法来处理用户请求。

11.7.3　程序举例

（1）对象功能描述和系统简要设计。本例模拟电信资费管理业务流程，在服务对象端注册一个电话用户。用户在远程客户端查询该用户本月发生的电话费用。根据对象功能的说明，用 UML 描述出服务对象需要实现的功能。

```
getFee() : float
openAccout( string name ) : Agent
```

（2）服务对象接口定义。根据系统分析的结果，用 IDL 编写出服务对象方法描述程序 TeleComm.idl。

```
module TeleComm
{
    interface Agent
    {
    float getFee();
    };
    interface AgentManager
    {
    Agent openAccount(in string name);
    };
};
```

运行 idlj 生成类文件：

```
idlj -dynamic_marshal TeleComm.idl
```

（3）接口的实现。Agent 接口的实现如下：

```
    public class AgentImpl extends DynamicImplementation
    {
        //构造 AgentImpl 类
        //get 方法在 invoke 动态调用方法中按名字匹配调用
        public synchronized org.omg.CORBA.Object get(String name)
        {
            //申请账户金额，生成账户对象
        }
        public void invoke(org.omg.CORBA.ServerRequest request)
        {
              //申请可移植对象适配器
              //填写调用参数，提出申请账户请求
        }
    }
//AgentManager 接口实现
    public class AgentManagerImpl extends DynamicImplementation
    {
```

```
        //构造 AgentManager 类
        public void invoke(org.omg.CORBA.ServerRequest request)
        {
        //填写调用参数表，向 Agent 对象提出业务请求，获取 Agent 对象
        //……
        }
    }
//服务器端程序设计
    public class Server
    {
        public static void main(String[] args)
        {
        //初始化 ORB、申请 POA 和 POA 管理器
        //申请默认服务对象，激活 POA 管理器
        //等待调用请求
        //……
        }
    }
//客户端程序设计
    public class Client
    {
        public static void main(String[] args)
        {
        //初始化 ORB，定位 Agentmanager 对象
        //发出调用请求
        org.omg.CORBA.Request request=manager._request("getFee");
        request.invoke();
        //查询返回结果，取得返回值
        request.get_response();
        //异常处理
        //……
        }
    }
```

（4）程序设计及运行环境说明。上述基于动态调用接口和动态骨架接口的程序是基于 VisiBroker 4.5.1 中间件环境开发的。其他环境下程序的实现方式和运行可能会有所差别。在 VisiBroker 环境下，按如下方式运行程序：

- 启动 VisiBroker Smart Agent 事务代理。
- 启动服务对象程序：vbj Server。
- 启动客户对象程序：vbj Client。

其中 vbj 是 VisiBroker for Java 中的 Java 代码解释执行程序。读者也可以用 Java 解释工具来代替，但需要指定 VisiBroker 中的 vbjorb 等库。

11.8 小结

本章在详细讲解分布式应用体系结构、对象请求代理（ORB）、对象适配器、动态调用接口、动态骨架接口、IDL 语法、CORBA 服务等 CORBA 中的主要内容的基础上，分别基

于 Java IDL 和 VisiBroker 等 CORBA 中间件开发工具和产品演示了分布式客户和服务器程序的设计方法。

对于初学者来讲，CORBA 中的内容比较枯燥难懂。笔者建议在阅读基本理论的基础上，参考中间件产品的联机说明和附带的例子，并且实际编写分布式应用程序，以加深对 CORBA 理论的理解和掌握。但是，正是因为其难以掌握，而又在当今的软件研发中应用得如此广泛，才使得真正掌握了这门技术的人弥足珍贵。

第 12 章　JNDI 编程范例

企业级应用程序要使用大量不同的目录服务，而目录服务就是寻找特定名称的相关资源的查找服务。Java 命名和目录接口（JNDI）就是用来简化对目录基础结构的访问，而该目录基础结构是用在高级网络应用开发中的。目录是一种特殊类型的数据库，它提供了对数据存储的快速访问方式。在传统上，数据库被看成是关系型数据存储模型，例如在 Oracle 和 Microsoft 的 SQL Server 中。相反地，目录数据库按照层次结构来存储信息。

以前需要使用不同的 API 来访问不同的目录服务，如轻量级的目录访问协议（LightWeight Directory Access Protocol，LDAP）或 Sun 的网络信息服务（Network Information Service，NIS）。然而 JNDI 提供了标准的 API 来访问任何类型的目录。另外，JNDI 还允许在网络上存储和检索 Java 对象。

本章将介绍如下内容：

- 什么是目录和命名服务。
- 什么是 LDAP 及如何使 LDAP 与目录服务一起工作。
- 举例说明如何用 JNDI 来管理目录信息。
- 通过 JNDI 查找 Enterprise JavaBeans 范例。

12.1　介绍 JNDI

尽管大多数人将“目录服务”与“命名服务”交替使用，但两者的含义却有所不同。命名服务是将一个名称与特定资源相关联，而目录服务则是将名称与一组属性和资源相关联。在搜索命名服务时，只能搜索特定名称；而在搜索目录时，可以搜索符合特定属性组的所有项目。

命名服务和目录服务的一个有趣之处在于它们通常都是完成相同的工作，即将名称与一些属性或对象组对应。当然，并不是所有的目录服务都是一样的。有些用平面名字空间，而有些则提供树状结构；有些可以存储特定类型的对象，而有些则可以存储任何类型的对象。

Java 命名与目录接口（Java Naming and Directory Interface，JNDI）规定了命名服务与目录服务之间的差别。命名服务将一个名称映射到一个对象。RMI Registry 和 CORBA Naming Service 都是命名服务。只能把 RMI 对象存放在 RMI Registry 中，把 CORBA 对象存放在 CORBA Naming Service 中，而不能把它们存放到其他对象中。目录服务也存放对象，但目录服务识别这些对象的相关属性。可以用项目属性来搜索目录，例如，可以从 LDAP 目录中搜索特定部门的每个人或姓名为张三的某个人。

在 20 世纪 90 年代早期，轻量级的目录访问协议（LightWeight Directory Access Protocol，LDAP）被作为一种标准的目录协议而开发出来。因为 LDAP 现在已经是最流行的目录协议，所以 JNDI 能够访问 LDAP。LDAP 定义了客户如何访问服务器上的数据；它并没有定义服务器应该如何存储数据。更普遍的是，需要与为建立 LDAP 而建立的服务器打交道，如 Open

LDAP 或 iPlanet 目录服务器。但是，LDAP 可以成为任何数据存储类型的接口。所以许多流行的目录服务现在已经有了某种类型的 LDAP 接口，包括 NIS、NDS、Active Directory，甚至 Windows NT 域。

虽然 LDAP 在流行和使用方面正在发展，但是要被普遍采用仍然有一段较长的路要走。其他的一些目录服务，如 NIS 仍然被广泛使用。另外一个 Java 开发者需要面对的问题是，要使 Java 在作为企业级开发语言方面能够获得成功，必须支持当前的分布式计算标准。如公共对象请求代理结构（CORBA），它在大的企业中被广泛应用，这些企业中有不同类型的应用程序要相互操作。CORBA 为定义对象的位置而采用了一种命名服务。

对于 Java 应用开发者来说，事情就变得简单多了，只要通过创建标准的 API 与命名和目录服务交互就可以，类似于 Java 应用开发者为在 JDBC 中访问数据库所做的工作那样。对于大量用 Java 开发来说，这种 API 很重要，特别是在 EJB 方面。

有关 EJB 的一个关键内容是能够在网络上存储和检索 Java 对象。目录服务（最可能是 LDAP）将会是 Java 对象的主要存储手段，特别是在 Java 对象相当稳定的时候（也就是说，除了存储在网络上之外，更经常的情况是从网络上检索 Java 对象）。即当从网络中加载对象时（如人员的记录，或者向串行化的 Java 对象那样的二进制数据），目录服务可以使得查找和检索数据更快。

图 12-1 显示了一个客户与多种目录服务之间的关系。每个目录服务要求采用它们自己的 API，从而导致在客户程序中增加了复杂性和代码的重复性。

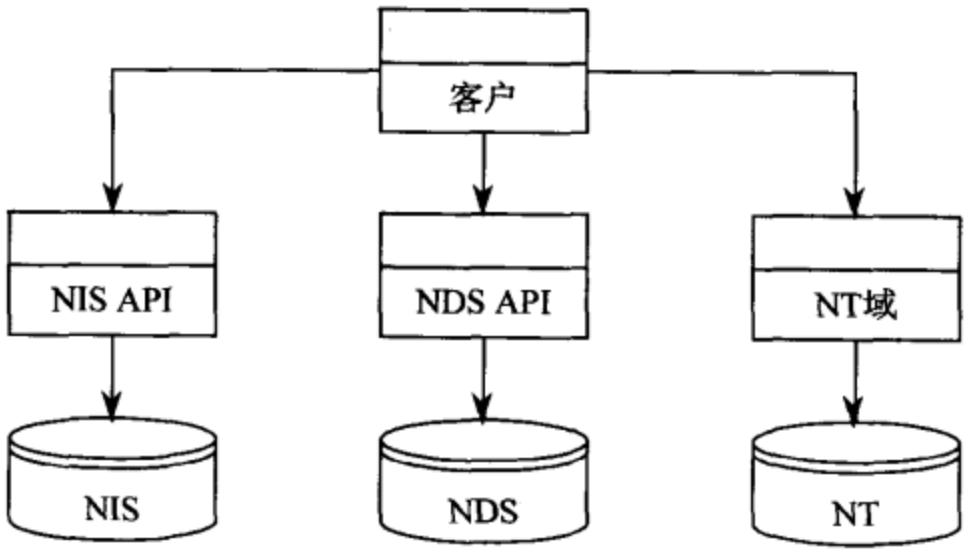


图 12-1　一个客户多种目录服务

下一个例子是如何用 JNDI 来为应用开发人员简化这部分内容。采用 JNDI 仍然要用到多个服务器和多种 API，但对于应用开发人员来说，从效果上看就好像在使用单个 API 一样，如图 12-2 所示。

为设计一个目录服务应用程序，使用单个标准的 API 会有以下优点：

- 因为把它当作了 Java 的标准部分，所以可以把它集成到许多方面的 Java 平台中。
- 更容易改变服务提供者（可以把 JNDI 服务提供者比喻成一个目录服务驱动器），就像改变使用 JDBC 驱动器那样。

但是，它也存在以下缺点：

- 创建的所有的服务提供者并不都相同，就像所有的 JDBC 数据库驱动器不都相同一样。

在协议层之上进行程序设计，所以有时不能使用已经习惯了的协议。

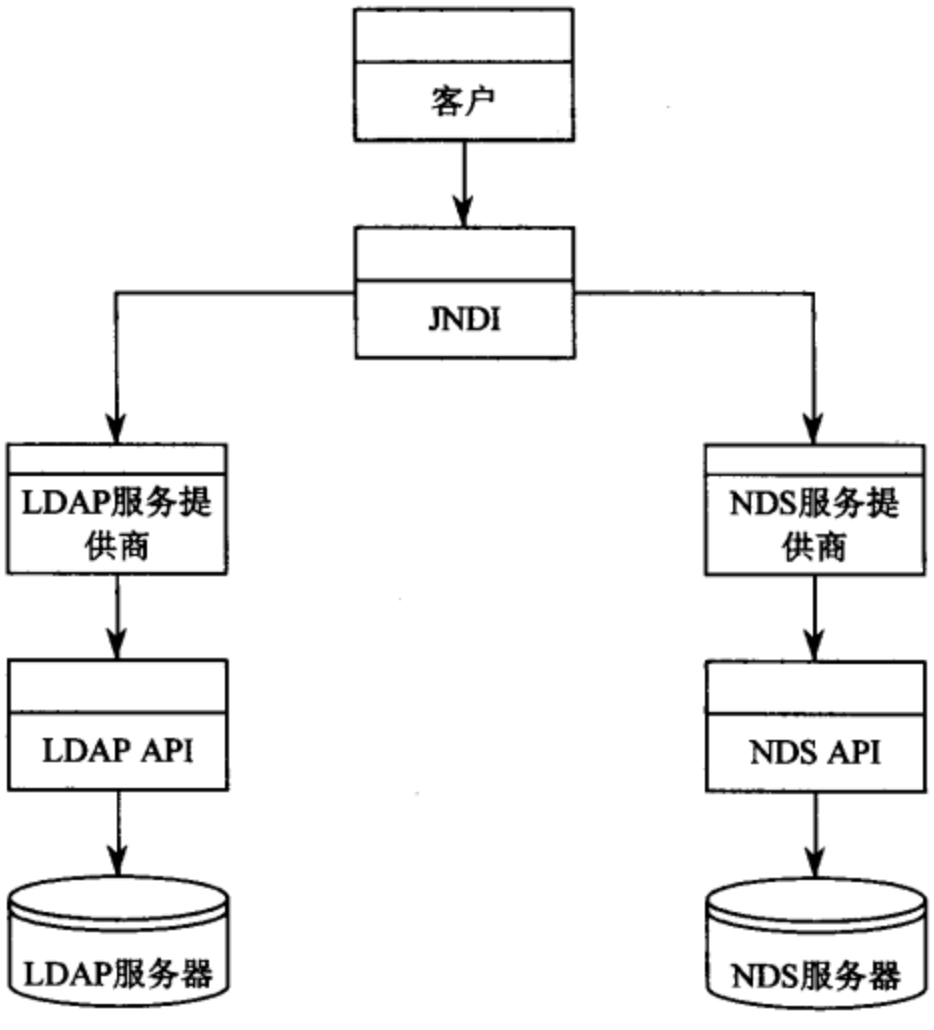


图 12-2　使用 JNDI 的目录服务

因为每个目录服务协议不同，所以即使采用了相同的 API 和适当的服务提供者，仍然要在应用程序中引入这些不同。

因此，我们更偏向于使用像 LDAP 这样的标准协议，从而可以挑选最好的服务提供者，而不必担心目录服务协议中的不同。下面是一个采用 JNDI 和 LDAP 结合的例子，是一个更优雅的解决方式，如图 12-3 所示。

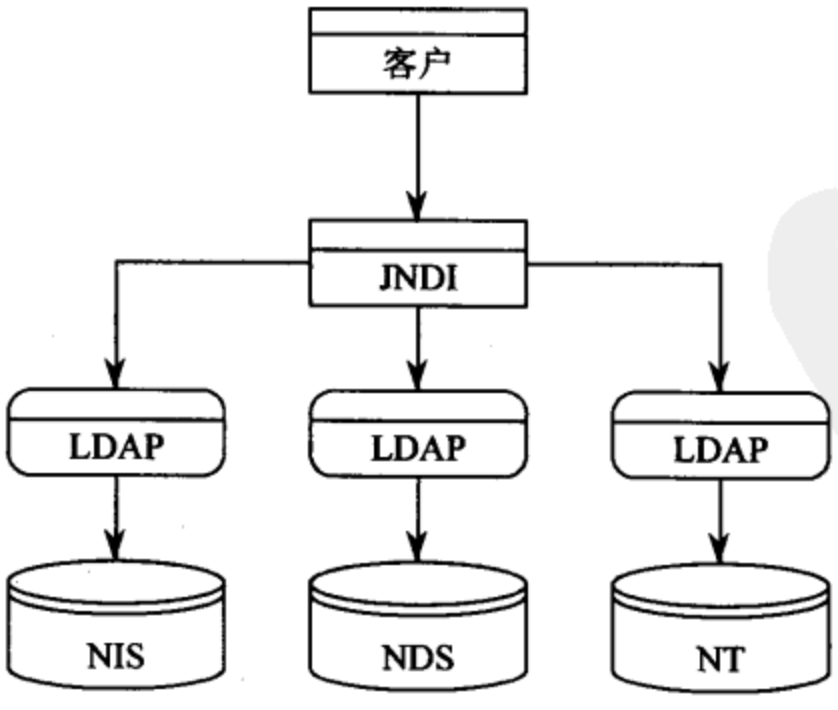


图 12-3　JNDI 与 LDAP 的结合使用

在这个结构中，采用 JNDI 与一个 LDAP 服务器通信。开发者只需要关心一个特定的协议（即 LDAP）和一个特定的 API（即 JNDI）。在这个例子中，要依赖由供应厂商提供相应的 LDAP 接口。而对于每一种流行的目录服务，都已经有产品允许通过 LDAP 与目录服务通信。

12.2　JNDI 架构

Java 命名和目录接口（Java Naming and Directory Interface，JNDI）是一套提供命名和目录功能的 API，Java 应用程序开发者通过使用 JNDI，在命名和目录方面的应用就有了共同的准则。

JNDI 包含一组 API 和一组 SPI（服务提供商接口，Service Provider Interface）。Java 程序通过 JNDI API 存取各种命名和目录服务，JNDI SPI 则使得各种命名和目录服务透明化，以允许 Java 程序通过 JNDI API 存取这些服务，如图 12-4 所示。

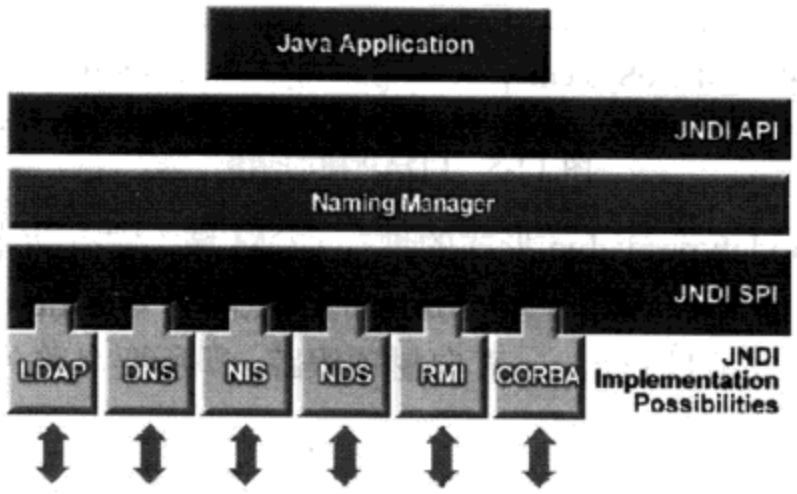


图 12-4　JNDI 的结构

JNDI 分为 3 个包。javax.naming 包含有命名服务的类别（classes）和访问接口（interfaces for accessing）。其中 Context API 让用户可以定义对象在命名空间中的“相对位置”。命令服务以上下文为接口，提供查看绑定、对象更名（renaming objects）等功能。InitialContext API 提供命名或目录服务的一个起始位置。因为在 JNDI 的世界中没有绝对的 root 观念，所有的动作都建立在上下文上；有了起始位置，用户才能借着它对其上下文的对象进行访问。NamingException API 是为 JNDI 定义的一组类别，负责检测所有发生在命名或目录服务里的异常情况。

javax.naming.directory 包由 javax.naming 扩展而来，提供访问目录服务的功能。它建立在命名服务上，增加对目录中的对象检索其属性和通过指定属性为条件来搜索等功能。DirContext API 提供对象在目录内上下文的接口，与 Context API 的运行方式类似，但更进一步定义了查询和更新目录中对象属性的方法。

javax.naming.spi 让系统开发人员为特定的命名或目录系统来编写使用 JNDI 的应用程序，例如在即插即用、Java 对象支持及多个命名系统等方面的应用。

12.3　利用 JNDI 在网络上搜索资源

在 JDK1.3 支持的目录服务中，LDAP 特别灵活。可以在 LDAP 目录中存放各种项目，LDAP

用层次（树）结构存储数据。要引用树中的项目，就需要列出树中的节点名。从所要项目开始，一直向树顶跟踪。图 12-5 显示了一个 LDAP 树。

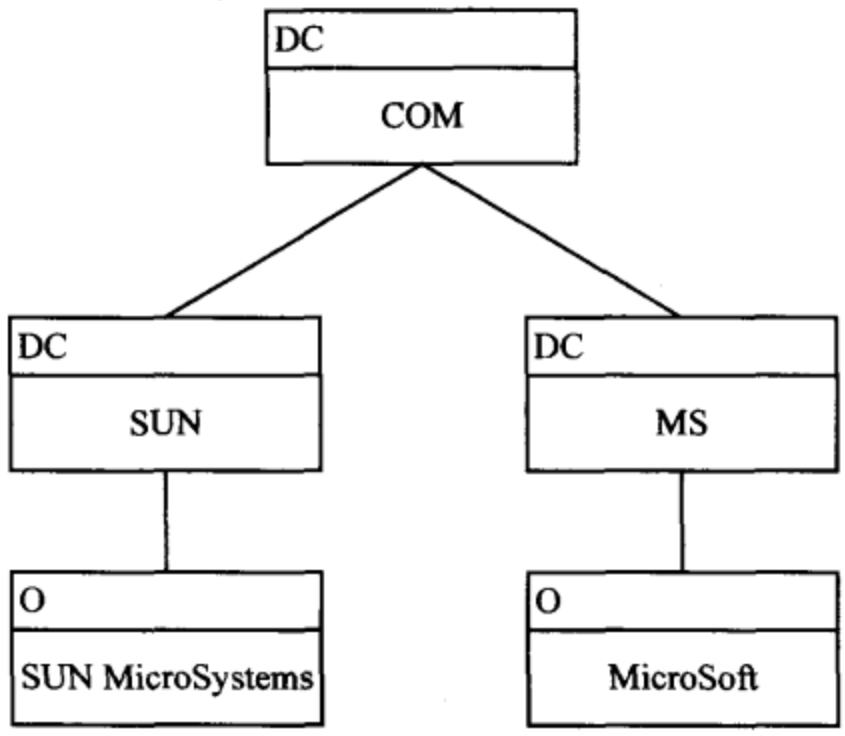


图 12-5 LDAP 树的示例

树中每个节点都有 nodetype=value 形式的惟一 COM 域，它有两个域组件。域组件的节点类型为 DC，叶子节点类型为 O，因此 Microsoft 的节点名为 O=Microsoft。如果用 JNDI 访问 Microsoft 节点，则要从所要项目开始，一直向树顶跟踪，即所要名称为 O=Microsoft、DC=MS、DC=COM。

尽管 LDAP 项目只是属性的集合，但 LDAP 也有类的概念。每个 LDAP 项目有 objectClass 属性，它列出对象的类层次。objectClass 不仅包含对象的类，还包含直到根类的所有父类的清单。类的嵌套一般不会太深，所以 objectClass 清单通常不会太长。

另外需要注意的是：类层次没有确定的目录树结构。目录树中的节点可以将其中一个父类作为子节点。表 12-1 列出了一些常见的 LDAP 类。这些类的完整清单在标准 RFC2256 中定义，可以在 http://www.ietf.org/rfc/rfc2256.txt 中找到。

表 12-1 一些常见的 LDAP 类

类名	父类	所要属性
top	None	objectClass
country	top	c
locality	top	None
organization	top	o
organizationalUnit	top	ou
person	top	sn, cn
organizationalPerson	top	None

LDAP 规范还列出了常见属性名。表 12-2 列出了常见属性名及其含义。

表 12-2　LDAP 的常见属性

属性名	含义
objectClass	对象及其父类的类名
dc	域环境的域名部分
cn	常用名，通常是对象名
sn	姓
c	标准两字符国家代码
l	地区（城市、国家或其他地区）
st	州或省
o	组织
ou	组织单元
title	人的职务
personalTitle	人的非职务身份
description	对象描述
mail	人的 E-mail 地址

另一个要知道的概念是，环境实际上是一组名称。可以调用 DirContext 对象的 createSubcontext 方法生成名字子环境。实际上，子环境就是从目录树中特定节点开始的名称子集。

12.4　用 JNDI 查找实例

用 JNDI 搜索 LDAP 目录很简单，只需要调用 DirContext 对象的 search 方法。搜索时主要有两种方法：指定匹配的属性集或 LDAP 过滤字符串。

下面是使用 JNDI 搜索 LDAP 目录的程序 NameSearch.java 的代码：

```
package usingj2ee.naming;

import javax.naming.*;
import javax.naming.directory.*;

public class NameSearch
{
    public static void main(String[] args)
    {
        try
        {
          // 得到上下文
            InitialDirContext ctx = new InitialDirContext();
          // 创建查找属性对象
            BasicAttributes searchAttrs = new BasicAttributes();
            searchAttrs.put("sn", "Tippin");
          // 从查找树顶端开始查找指定属性的节点
```

```
NamingEnumeration objs = ctx.search(
    "ldap://ldap.wutka.com/o=Wutka Consulting, dc=wutka, dc=com",
    searchAttrs);
// 循环查找返回结果
while (objs.hasMoreElements())
{
// 每个返回项目是一个 SearchResult 对象
    SearchResult match = (SearchResult) objs.nextElement();
// 打印节点名称
    System.out.println("Found "+match.getName()+":");
//得到节点属性
    Attributes attrs = match.getAttributes();
      NamingEnumeration e = attrs.getAll();
// 循环属性内容
    while (e.hasMoreElements())
    {
// 得到下一个属性
        Attribute attr = (Attribute) e.nextElement();
// 打印属性值
        System.out.print(attr.getID()+" = ");
        for (int i=0; i < attr.size(); i++)
        {
            if (i > 0)
                System.out.print(", ");
                System.out.print(attr.get(i));
        }
                System.out.println();
            }
            System.out.println("----------------------------------");
        }
    }
    catch (Exception exc)
    {
        exc.printStackTrace();
    }
  }
}
```

用过滤器搜索比较复杂，任何 LDAP 过滤字符串都要放在括号内。要匹配目录中的所有对象，可以使用过滤字符串，例如（objectClass=*）。

可以用=、>=、<=和~=（约等于）进行比较，例如（age>=19）。与、或、非的语法比较奇怪。如果要测试 age>=19 与 sn=Smith，则表达式为 ((age>=19)&(sn=Smith))。&表示与，|表示或，!表示非。对于&和|，可以在它们后面列出多个表达式。而 ！后面只能有一个表达式。

由于只可以进行大于或等于比较，因此可以用小于或等于结合 ！运算来达到进行大于比较的效果。例如，如果年龄必须大于 19 岁，则可以用 (!(age<=19))。如果需要合并 & 和 | 操作符，则要用括号分开表达式。参考下面的表达式：

```
((age>=19)|((age>=13)&(parentalPermission = true)))
```

下面的程序代码进行了简单的过滤器搜索，以转储整个目录的内容。AllSearch.java 的代

码如下所示：

```
package usingj2ee.naming;

import javax.naming.*;
import javax.naming.directory.*;

public class AllSearch
{
    public static void main(String[] args)
    {
        try
        {
            // 获得上下文
            InitialDirContext ctx = new InitialDirContext();
              //定义查询范围
            SearchControls searchControls = new SearchControls();
            searchControls.setSearchScope(SearchControls.SUBTREE_SCOPE);
            //从查找树顶端开始查找指定属性的节点
            NamingEnumeration objs = ctx.search(
                "ldap://ldap.wutka.com/o=Wutka Consulting, dc=wutka, dc=com",
"(objectClass=*)", searchControls);
            //循环查找返回结果
            while (objs.hasMoreElements())
            {
            //每个返回项目是一个 SearchResult 对象
                SearchResult match = (SearchResult) objs.nextElement();
                //打印节点名称
             System.out.println("Found "+match.getName()+":");
            //得到节点属性
             Attributes attrs = match.getAttributes();
          NamingEnumeration e = attrs.getAll();
            //循环属性内容
             while (e.hasMoreElements())
             {
            //得到下一个属性
                Attribute attr = (Attribute) e.nextElement();
            //打印属性值
                System.out.print(attr.getID()+" = ");
                for (int i=0; i < attr.size(); i++)
                {
                    if (i > 0)
                          System.out.print(", ");
                          System.out.print(attr.get(i));
                }
                System.out.println();
             }
             System.out.println("----------------------------------------");
        }
}
catch (Exception exc)
{
```

```
            exc.printStackTrace();
        }
    }
}
```

这个查询通过使用实现了 DirContext 接口（如这里使用的 InitialDirContext 类）对象的 search()方法来执行一个查询。这个查询至少需要提供查找基点和筛选器，还有其他的参数用来帮助管理这个结果。这个查询成功后，就返回一个 NamingEnumeration 对象。

在得到了初始上下文（ctx 对象）后，下一步就是要定义查询范围。它在 SearchControls 对象中设置，是一个 DirContext 对象的 search()方法的可选参数。如果没有定义一个查询范围，JNDI 就会认为范围为 subtree，从而接下来的一行代码就不需要。但是在这里，定义了查询范围。

```
//定义查询范围
    SearchControls searchControls = new SearchControls();
    searchControls.setSearchScope(SearchControls.SUBTREE_SCOPE);
```

在定义了查询范围之后，就可以执行实际的查询了：

```
NamingEnumeration objs = ctx.search(
            "ldap://ldap.wutka.com/o=Wutka Consulting, dc=wutka, dc=com",
            "(objectClass=*)", searchControls);
```

NamingEnumeration 对象中的每个元素都包含了 SearchResult 对象，它可以像下面这样进行检索：

```
SearchResult match = (SearchResult) objs.nextElement();
```

可以像下面这样获得一个入口的名称：

```
String dn = match.getName();
```

要得到一个入口的属性值，要用 SearchResult 类的 getAttributes()方法：

```
Attributes attrs = match.getAttributes();
```

这个方法会返回一个实现了 Attribute 接口的具体对象（InitialDirContext 类返回一个 BasicAttributes 对象）。

在得到了 Attributes 对象（记住它是一个集合类对象）后，可以用 NamingEnumeration 对象在它的元素之间遍历。

```
NamingEnumeration e = attrs.getAll();

    //循环属性内容
    while (e.hasMoreElements())
    {
    //得到下一个属性
    Attribute attr = (Attribute) e.nextElement();

    //打印属性值
    System.out.print(attr.getID()+" = ");
    for (int i=0; i < attr.size(); i++)
    {
        if (i > 0)
              System.out.print(", ");
             System.out.print(attr.get(i));
    }
    System.out.println();
    }
```

为在查询中返回的每个属性之间遍历，NamingEnumeration 类提供了一些方法。NamingEnumeration 对象中的每个元素都包含了一个 Attribute 对象，它代表了一个属性和属性值。Attribute 接口的 getID()方法返回属性的名称，而 getAll()方法会返回一个标准的 Java 枚举对象，可以访问它来获取单个的属性值。

12.5　小结

本章主要介绍了目录和命名服务，什么是 LDAP 以及如何使 LDAP 与目录服务一起工作，并且通过例子说明了如何用 JNDI 来管理目录信息和通过 JNDI 查找 Enterprise JavaBeans。

第 13 章　Java 开发 Web Service

13.1　什么是 Web Service

一时间，好像所有的计算机期刊、书籍和网站都在提 Web Service。然而，当前大多数对 Web Service 的介绍都没能清楚地说明 Web Service 到底是什么。他们只是鼓吹 Web Service 是多么多么的好，简直就像是在做广告。本章就是要讲清楚 Web Service 到底是什么，在什么情况下应该使用 Web Service。

看一下当前的应用程序开发，你会发现一个绝对的倾向：人们开始偏爱基于浏览器的瘦客户应用程序。这当然不是因为瘦客户能够提供更好的用户界面，而是因为它能够避免花在桌面应用程序发布上的高成本。发布桌面应用程序成本很高，一半是因为应用程序安装和配置的问题，另一半是因为客户和服务器之间通信的问题。

关于客户端与服务器的通信问题，如果大家都使用 Java 来开发应用程序，那就天下太平了。然而，事实上大多数商业数据仍然在大型主机上以非关系文件（VSAM）的形式存放，并由 COBOL 语言编写的大型机程序访问。同时，目前还有很多商用程序在继续使用 Java、C++、Visual Basic 和其他各种各样的语言编写。现在，除了最简单的程序之外，所有的应用程序都需要与运行在其他异构平台上的应用程序集成并进行数据交换。这样的任务通常都是由特殊的方法，如文件传输和分析、消息队列，还有仅适用于某些情况的 API（如 IBM 的高级程序到程序交流（APPC））等来完成的。现在，使用 Web Service，客户端和服务器就能够自由地用 HTTP 进行通信，而不管两个程序的平台和编程语言是什么。

Web Service 是建立可互操作的分布式应用程序的新平台。作为一个程序员，你可能已经用 COM 或 DCOM 建立过基于组件的分布式应用程序，或者使用 CORBA、RMI 等手段来实现远程的调用。Web Service 平台也是一套标准，只不过它做的更好。它定义了应用程序如何在 Web 上实现互操作性。你可以用任何你喜欢的语言，在任何你喜欢的平台上写 Web Service，只要可以通过 Web Service 标准对这些服务进行查询和访问。

Web Service 平台需要一套协议来实现分布式应用程序的创建。任何平台都有它的数据表示方法和类型系统。要实现互操作性，Web Service 平台必须提供一套标准的类型系统，用于沟通不同平台、编程语言和组件模型中的不同类型系统。在传统的分布式系统中，基于界面的平台提供了一些方法来描述界面、方法和参数（注：如 COM 和 CORBA 中的 IDL 语言）。同样的，Web Service 平台也必须提供一种标准来描述 Web Service，让客户可以得到足够的信息来调用这个 Web Service。最后，还必须有一种方法来对这个 Web Service 进行远程调用。这种方法实际是一种远程过程调用协议（RPC）。为了达到互操作性，这种 RPC 协议还必须与平台和编程语言无关。下面会简要介绍组成 Web Service 平台的这三个技术。

可扩展的标记语言（XML）是 Web Service 平台中表示数据的基本格式。除了易于建立和易于分析外，XML 主要的优点在于，它既是平台无关的、又是厂商无关的。这种无关性是比

技术优越性更重要的，因为软件厂商是不会选择一个由竞争对手所发明的技术的。

XML 解决了数据表示的问题，但它没有定义一套标准的数据类型，更没有说怎么去扩展这套数据类型。例如，整型数到底代表什么？16 位，32 位，还是 64 位？这些细节对实现互操作性都是很重要的。W3C 制定的 XML Schema（XSD）就是专门解决这个问题的一套标准。它定义了一套标准的数据类型，并给出了一种语言来扩展这套数据类型。Web Service 平台就是用 XSD 来作为其数据类型系统的。当用某种语言（如 Java、C++或 C#）来构造一个 Web Service 时，为了符合 Web Service 标准，使用的所有数据类型都必须转换为 XSD 类型。你用的工具可能已经自动帮你完成了这个转换，但你很可能会根据需要修改一下转换过程。

Web Service 建好以后，你或者其他人就会去调用它。简单对象访问协议（SOAP）提供了标准的 RPC 方法来调用 Web Service。实际上，SOAP 在这里有点用词不当，它意味着下面的 Web Service 是以对象的方式表示的。但事实并不一定如此，你完全可以把你的 Web Service 写成一系列的 C 函数，并仍然使用 SOAP 进行调用。SOAP 规范定义了 SOAP 消息的格式，以及怎样通过 HTTP 协议来使用 SOAP。SOAP 也是基于 XML 和 XSD 的，XML 是 SOAP 的数据编码方式。

你会怎样向别人介绍你的 Web Service 有什么功能，以及每个函数调用时的参数呢？你可能会自己写一套文档，你甚至可能会口头上告诉需要使用你的 Web Service 的人。这些非正式的方法至少都有一个严重的问题：当程序员坐到电脑前，想要使用你的 Web Service 的时候，他们的工具无法给他们提供任何帮助，因为这些工具根本就不了解你的 Web Service。解决方法是：用机器能阅读的方式提供一个正式的描述文档。Web Service 描述语言（WSDL）就是这样一个基于 XML 的语言，用于描述 Web Service 及其函数、参数和返回值。因为是基于 XML 的，所以 WSDL 既是机器可阅读的，又是人可阅读的，这将是一个很大的好处。一些最新的开发工具既能根据你的 Web Service 生成 WSDL 文档，又能导入 WSDL 文档并生成调用相应 Web Service 的代码。

13.2　一个简单的 SOAP 程序

13.2.1　实例说明

前边讲了 Web Service 的一些最基本的概念，如果你以前从来没有接触过 Web Service 可能已经觉得有点头大了，下边这个例子就试图将你从糊涂中解救出来。

HelloWorld 好像到哪里都很实用，因此，这里还是一个 HelloWorld 程序，它是基于 Tomcat 和 Apache SOAP 的一个简单应用程序。

13.2.2　准备工作

Apache SOAP，即 Apache 软件基金会对 SOAP 规范的实现。和所有其他 Apache 工程类似，Apache SOAP 工程是源代码开放的，并且按照 Apache 许可协议来发行。这是当前最好的 SOAP 实现之一。它最初是由 IBM 开发的，当时还叫 IBM-SOAP，后来被主动赠送给了 Apache 软件基金会的 Apache XML。Apache SOAP 的另一版本 AXIS 已经于 2002 年 10 月发布了 1.0 版本，AXIS 是全部重写的源程序，在架构上主要是使用 SAX 解析器代替了 Apache SOAP 使

用的 DOM。通过前边对 SAX 和 DOM 的介绍比较，这种优势已经很清楚了，AXIS 降低了内存开销，对于大信息量数据有更高的执行效率。对于一般的应用，Apache SOAP 已经能够满足需要了。

Apache SOAP 也是可以免费下载的。当前最新稳定版本为 2.3.1 版本，下载地址为 http://xml.apache.org/dist/soap/version-2.3.1/soap-bin-2.3.1.zip。前边提到 SOAP 是基于 XML 的，所以 Apache SOAP 还依赖一个 XML 解析器，Apache 的 XML 解析器是 Xerces，下载地址为 http://xml.apache.org/dist/xerces-j/Xerces-J-bin.1.4.4.zip。它还需要 mail.jar 和 activation.jar。

安装 Apache SOAP 非常简单，共包含如下 3 个简单的步骤：

（1）解开下载所得文件的 ZIP 压缩。解压缩之后就得到了一个 soap-2.3.1 子目录。为了便于管理，最好把 soap-2.3.1\webapps 下的 soap 文件夹拷贝到$CATALINA_HOME\webapps 下。

（2）配置 Web 环境。编辑$CATALINA_HOME\conf\server.xml 配置文件，在该文件中增加一个新的<Context>标记，如下所示：

```
<Context path="/ soap" docBase="soap"  debug="1" reloadable="true"/>
```

如果你没有按照第一步把 soap 文件夹拷贝到 Tomcat 的 webapps 目录，docBase 就必须指定到这个文件夹的绝对路径。

（3）设置 Web 服务器 classpath。从 Tomcat 4.0 开始，Tomcat 不再使用系统的 CLASSPATH，所以必须设定两次 CLASSPATH。设定系统的 CLASSPATH 使得程序能编译，设定 Tomcat 的 CLASSPATH 使得程序能运行。当然也有一个巧妙的办法是编辑 $CATALINA_HOME\bin\setclasspath.bat 文件，把 CLASSPATH 设定为%CLASSPATH%就可以了。这样只需要修改系统的 CLASSPATH 就可以了，不过 Apache 的程序员这样做自然有他的道理，CLASSPATH 的清晰对于程序正确的编译运行起着至关重要的作用，尤其是一旦出错，这种错误是很难查出来的。如果你已经按照本书的顺序把前边的练习都做过了，你会发现你的 CLASSPATH 是非常长的，而这么长的文字差一个标点都不行。在 Linux、UNIX 环境下字母的大小写也非常关键，所以最好还是把 Tomcat 的 CLASSPATH 和系统的 CLASSPATH 分开设定比较好。

同时，把 soap.jar 和 xerces.jar 加入 CLASSPATH 中。如果你有不支持名称空间的 XML 解析器，则支持名称空间的 XML 解析器必须放在前边，否则 Apache SOAP 将不能正确工作。

13.2.3 编写代码

HelloServer.java 的代码如下所示：

```
package hello;
public class HelloServer
{
    public String sayHello(String name)
    {
        System.out.println("sayHello("+name+")");
        return "Hello " + name + ", How are you?";
    }
}
```

13.2.4　部署服务

Apache SOAP 的服务部署最基本的办法有两个，这里先介绍第一种方法，就是使用 Apache SOAP 的管理工具部署。

（1）启动 Tomcat，打开浏览器，在地址栏输入 http://localhost/soap/admin/（如果你的 Tomcat 服务端口不是 80，在 localhost 后边加上“：端口号”，默认是 8080），会出现如图 13-1 所示的窗口。

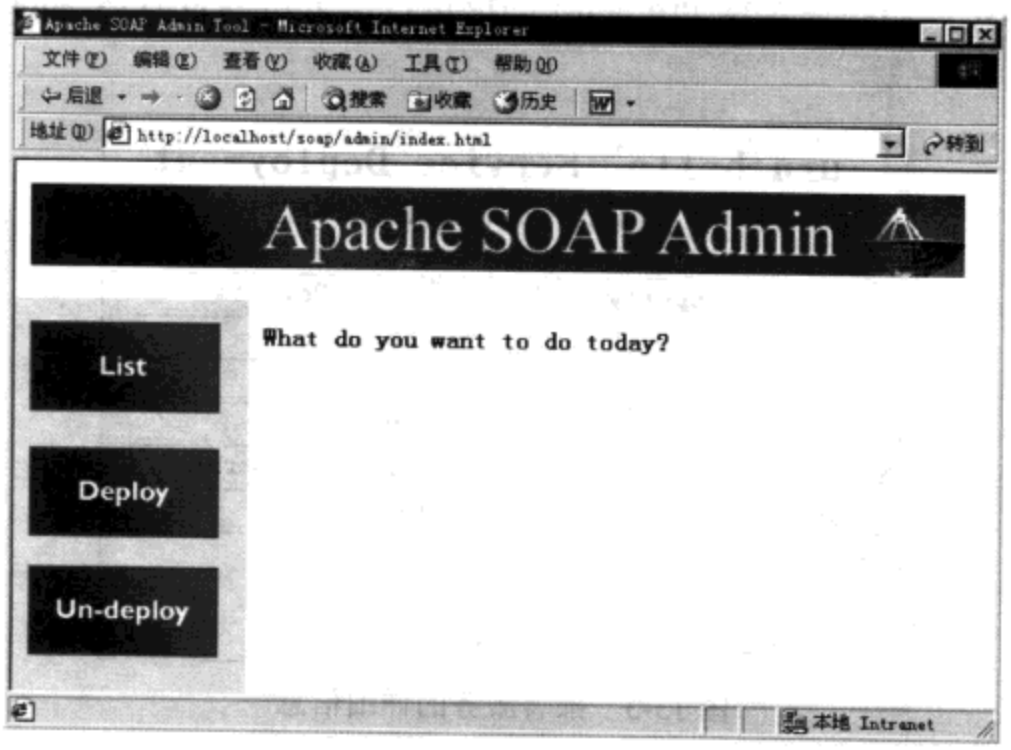


图 13-1　Apache SOAP 的管理页面

（2）单击左边的 Deploy 按钮，将出现一个部署服务的表单，如图 13-2 所示。如果程序报错，请仔细检查设置，尤其是 CLASSPATH 的设置。然后在表单中填入相应的数据，如表 13-1 所示。

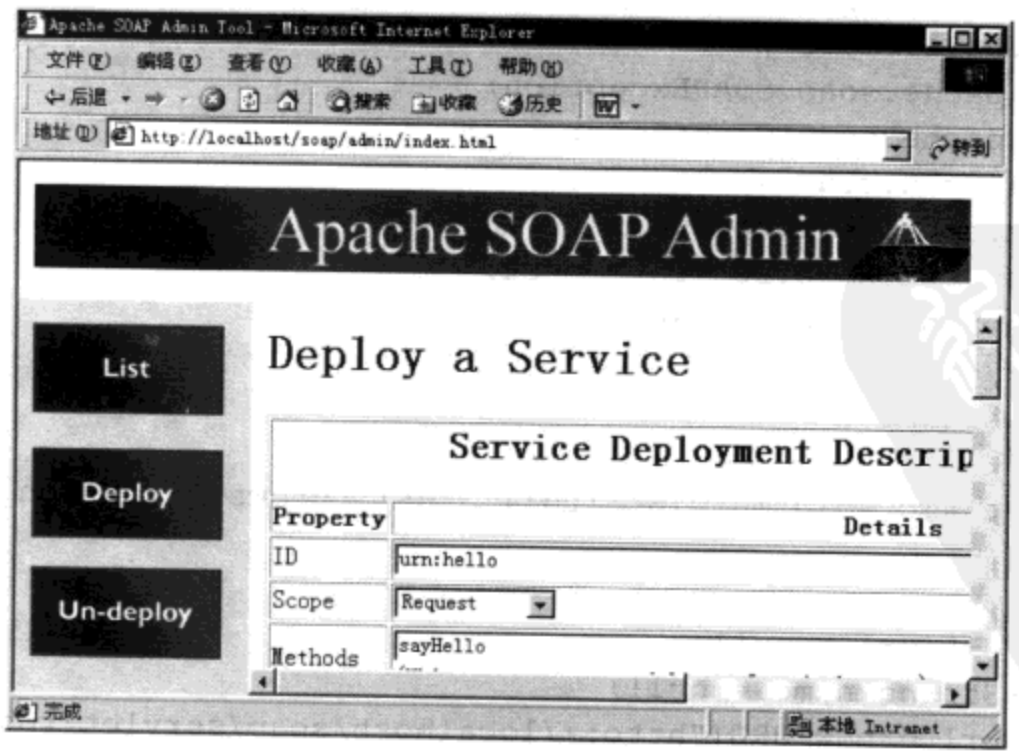


图 13-2　部署服务的表单

表 13-1 部署服务的设置

项目	相应数据
ID	urn:hello
Method	SayHello
Java Provider(Provider Class)	hello.HelloServer

其他的值均保持默认属性，提交表单以完成部署。

单击左边的 List 按钮，会发现 urn:hello 的链接，进去可以查看这次部署的详细信息，如图 13-3 所示。

'urn:hello' Service Deployment Descriptor

Property	Details
ID	urn:hello
Scope	Request
Provider Type	java
Provider Class	hello.HelloServer
Use Static Class	false
Methods	sayHello
Type Mappings	
Default Mapping Registry Class	

图 13-3 部署服务的详细信息

13.2.5 程序调用

使用 Apache SOAP 的 Web Service 已经部署好了，下面使用一个 Java 程序来调用这个服务。Client.java 的代码如下所示：

```
import java.net.URL;
import java.util.Vector;
import org.apache.soap.SOAPException;
import org.apache.soap.Constants;
import org.apache.soap.Fault;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;

public class Client
{
    public static void main(String[] args) throws Exception
    {
        try{
            URL url = null;
            String name = null;
            url = new URL("http://localhost/soap/servlet/rpcrouter");
            if (args.length==0)
                name="friend";
```

```
            else
                name = args[0];

            // 构造 Call 对象
            Call call = new Call();
            call.setTargetObjectURI("urn:hello");
            call.setMethodName("sayHello");
            call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
            //添加参数
            Vector params = new Vector();
            params.addElement(new Parameter("name", String.class, name, null));
            call.setParams(params);

            Response resp = null;
            try{
                resp = call.invoke(url, "");// 发出调用
            }
            catch( SOAPException e )
            {
System.err.println("Caught SOAPException (" + e.getFaultCode() + "): " +
e.getMessage());
                System.exit(-1);
            }

            // 检查应答
            if( !resp.generatedFault() ){//如果没有返回错误信息
                Parameter ret = resp.getReturnValue();
                Object value = ret.getValue();
                System.out.println(value);
            }else{//如果返回错误信息则打印出错误
                Fault fault = resp.getFault();
                System.err.println("Generated fault: ");
                System.out.println (" Fault Code = " + fault.getFaultCode());
                System.out.println    ("    Fault    String    =    "    +
fault.getFaultString());
            }
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

编译并运行这个程序，运行结果如图 13-4 所示。

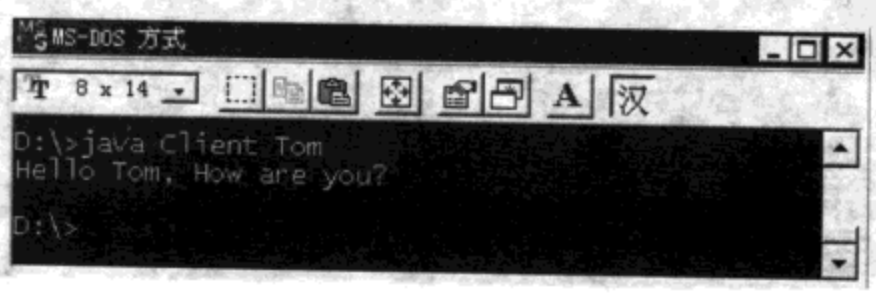


图 13-4　Client.java 的运行结果

正如所期望的那样，这个运行结果是正确的。但是它们之间到底进行了什么操作呢？可以通过下边的方法得到。说得简单点就是客户端程序构建了信息，发送到了服务器端。服务器端在收到客户端信息后，解析信息并处理信息，然后生成新的信息返回给客户端。我们要做的就是看看这些信息到底是什么。

先看看 Client.class 到底发出了什么信息。还是使用前边介绍过的方法，稍加修改，Server.java 的代码如下所示：

```
import java.net.*;
import java.io.*;
public class Server{
    public static void main(String[] arg){
        int port=80;          //监听的端口号
        try {
            ServerSocket s=new ServerSocket(port);
            System.out.println("Waiting on port "+port+"\r\n");
            while(true){
                Socket socket=s.accept();
                InputStream is=socket.getInputStream();
                InputStreamReader isr= new InputStreamReader(is);
                BufferedReader br=new BufferedReader(isr);
                FileOutputStream fos=new FileOutputStream("soapdata.txt");
                DataOutputStream dos=new DataOutputStream(fos);
                while(true){    //按行读取，按行输出
                    String str=br.readLine();
                    dos.writeBytes(str+"\r\n");     //把读取的信息记入文件
                    System.out.println(str);        //把读取的信息显示出来
                }
            }
        }catch(Exception e){
            System.err.println(e);
        }
    }
}
```

编译 Server.java 程序，并关闭占用 80 端口的 Web 服务器，然后运行该程序，结果如图 13-5 所示。

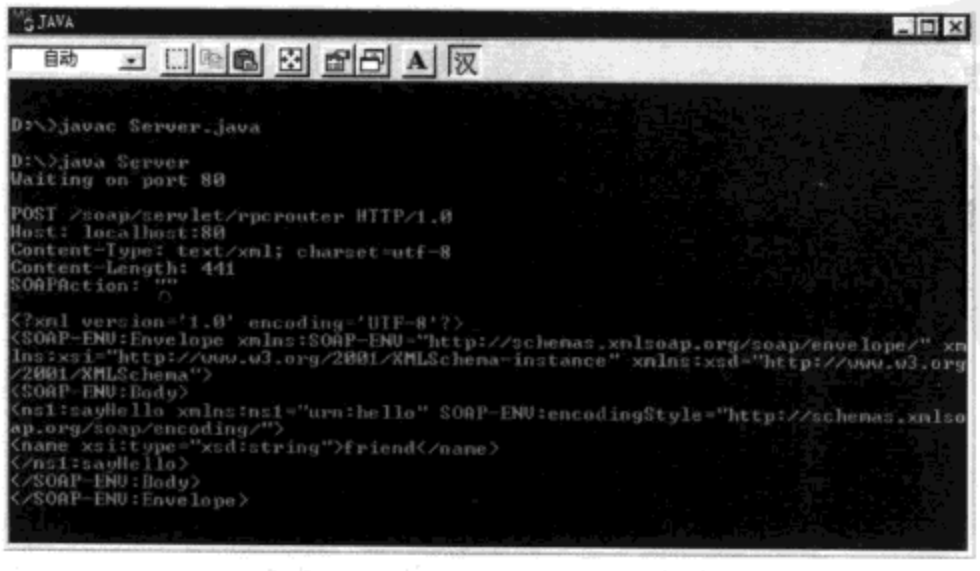


图 13-5　Server.java 的运行结果

上边显示的就是 Client 传送到 Web Service 服务器的信息，如何来得到服务器返回的信息呢？可以修改一下这个文件，让它接受返回值就可以了。还有更简单的办法，就是先把刚才得到的文件 soapdata.txt 的内容拷贝一下，然后 telnet 到本地 80 端口，选择粘贴即能得到 Web Service 服务器返回的信息了。可以使用 Windows 自带的 telnet 工具，还可以使用其他的 telnet 工具。现在最好用的工具应该是 CRT 和 NetTerm，这些工具开始都要填写服务器地址和端口，这里填写 localhost 和 80 就可以了。如图 13-6 所示，使用的是 SecureCRT（如果没有返回结果，加两个回车即可）。

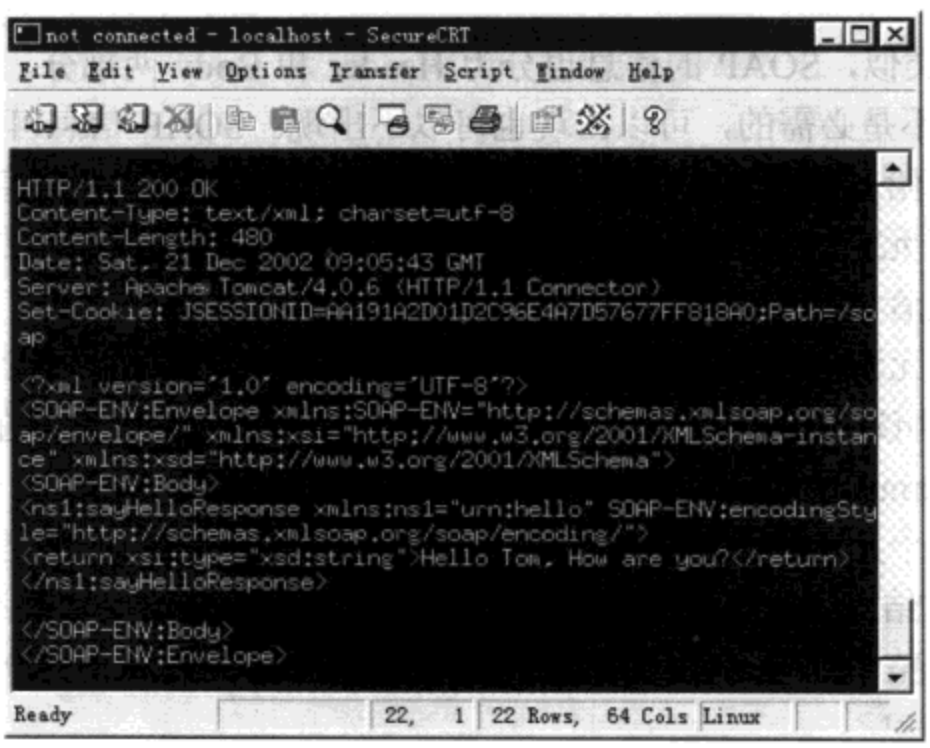


图 13-6　使用 telnet 下载数据

现在看明白了，SOAP 其实很简单，它就是定义了一系列的格式（基于 xml 的格式），使得客户端可以和服务器端相互通信。学习使用 SOAP 就是学习这些定制好的格式，并构建符合这些格式的信息。

13.3　SOAP 的信息结构

13.3.1　SOAP 封套（Envelope）

封套是 SOAP 信息中最外层的包容元素，它必须遵守以下的语法规则：

- 这个元素的名字永远是 Envelope。
- 每条 SOAP 信息必须包含这个元素并把它作为最外层的包含元素。
- 这个元素可以有名称空间声明并附带其他属性，包容在此元素中的所有的属性必须有名称空间方面的限定。Envelope 元素还可以包含子元素，类似于其附加属性。Envelope 的子元素也必须有名称空间的限制，而且它们后边必须紧跟着 SOAP Body 元素。Envelope 元素里，可以通过使用 encodingType 属性来指定编码风格，并且 SOAP 信息里所有的其他元素也有类似属性。

出于兼容性方面的考虑，SOAP 有一个版本核查机制。版本核查机制的传统用法是在应用

程序的范畴内向后兼容的问题。这种机制中大多数都采用主版本号加辅版本号这样一种模型。如 SOAP 的当前版本是 1.1，其中的主版本号为 1，辅版本号也为 1。可是 SOAP 不是通过数字来核查版本信息，而是通过使用一个与 http://schemas.xmlsoap.org/soap/envelop 名称空间相关联的 Envelope 元素。如果某个信息为其封套指定了其他的名字，它就必须被当作一个“版本不匹配”的情况来处理。如果出现了这种情况，SOAP 应用程序就会抛出 VersionMismatch 的错误信息。

13.3.2 SOAP 信息头（Header）

和 HTML 非常类似，SOAP 的信息也分为 Header 和 Body 两部分。SOAP 的信息头还是和 HTML 一样，并不是必需的，可以出现也可以不出现。SOAP 信息可以通过它把关于这条信息本身的信息传递给其他信息。如果 SOAP 的信息头出现在 SOAP 信息中，它必须是 SOAP Envelope 的第一个子元素。什么时候需要使用信息头呢？最常用的方法是使用 Header 对调用方进行身份验证，或者使用它对通信过程进行管理。

Header 的一级子元素必须使用限定的元素名，这个元素名必须由一个名称空间和一个本地名字组成。根据 XML 协议，应用程序会认为包含在 Header 中的一切元素都在一个名称空间中。

- SOAP:ENV:mustUnderstand：这个属性的取值可以是 1（true）或 0（false），如果取值为 0 表示这个元素没有意义，可以跳过继续处理后续元素。如果是 1 则表明，该 Header 条目的接收结点要么必须遵循语义并正确地处理这些语义，要么必须宣称处理消息失败，停止整个处理过程，并返回出错信息。该系统默认值为 0，比如下边的例子。

```
<SOAP-ENV:Header>
    <s:user xmlns:s="An URI" SOAP-ENV:mustUnderstand="0">
        albatross@sohu.com
    </s:user>
</SOAP-ENV:Header>
```

如果服务器对用户的操作活动进行登记，理解这个信息的服务器就会收集到使用此服务器的用户的邮件地址。如果信息头没有这个元素，服务器可以把这个信息记录为 unknown。假如把这个 SOAP-ENV:mustUnderstand 设为 1，服务器端就必须要得到正确的用户信息。

- SOAP-ENV:actor：SOAP 不一定会直接到达目的地，有可能经历好几个中间环节后才能最后到达目的地。这些“中间环节”指的是信息在传递过程中途经的 SOAP 应用程序，它们能够接收和转发 SOAP 信息。它们接收到 SOAP 信息后会先检查信息头，看看哪些是发给自己的，还要转发到哪里。信息的最终目的地都必须用 URI 标识出来，SOAP-ENV:actor 就是处理这一任务的。下边的例子根据信息头对客户的订单进行处理，这个订单要经过几次处理才能正确执行。下面沿着这个流程看一下，其中最初的客户请求如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://scheamas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
        <c:fillInID xmlns:c="http://www.china-publication.com/customer"
            SOAP-ENV:mustUnderstand="1"
            SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next">
            <c:bodyID>customer</c:bodyID>
```

```
    </c:fillInID>
    <p:placeOrder xmlns:p="http://www.china-publication.com/order"
        SOAP-ENV:mustUnderstand="1"
        SOAP-ENV:actor="http://www.china-publication.com/placeOrder">
        <p:bodyID>products</p:bodyID>
    </p:placeOrder>
    <s:shipOrder xmlns:s="http://www.china-publication.com/order"
        SOAP-ENV:mustUnderstand="1"
        SOAP-ENV:actor="http://www.china-publication.com/shipOrder">
        <s:bodyID>shipper</s:bodyID>
    </s:shipOrder>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
    <customer ID="customer">
        <name ID="customerName" name="张三"/>
        <address id="customerAddress">
            <city>北京</city>
            <details>知春路太月圆五号楼二单元 1202</details>
            <zipcode>100036</zipcode>
            <phone>62083412</phone>
        </address>
    </customer>
    <order ID="products">
        <dvd>神奇燕尾服</dvd>
        <dvd>英雄</dvd>
    </order>
    <express>
        <conveyer>神龙速递公司</conveyer>
        <destination>
            <name href="#customerName"/>
            <address href="#customerAddress"/>
        </destination>
    </express>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

上边的信息传递到第一个 SOAP 应用程序时，程序检查到它只需要处理第一个信息头，因为后边两个信息头的 actor 不是 http://schemas.xmlsoap.org/soap/actor/next，所以第一个 SOAP 处理程序将忽略它们。第一个信息头被处理后，把信息转向第二个信息头的 actor，即 http://www.china-publication.com/placeOrder。这时，再转发的时候信息头和信息体都会发生变化。转发后的信息如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>
        <p:placeOrder xmlns:p="http://www.china-publication.com/order"
            SOAP-ENV:mustUnderstand="1"
            SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next">
            <p:bodyID>products</p:bodyID>
        </p:placeOrder>
        <s:shipOrder xmlns:s="http://www.china-publication.com/order"
            SOAP-ENV:mustUnderstand="1"
```

```
        SOAP-ENV:actor="http://www.china-publication.com/shipOrder">
        <s:bodyID>shipper</s:bodyID>
    </s:shipOrder>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
    <customerID ID="2387432"/>
    <customer ID="customer">
        <name ID="customerName" name="张三"/>
        <address id="customerAddress">
            <city>北京</city>
            <details>知春路太月圆五号楼二单元 1202</details>
            <zipcode>100036</zipcode>
            <phone>62083412</phone>
        </address>
    </customer>
    <order ID="products">
        <dvd>神奇燕尾服</dvd>
        <dvd>英雄</dvd>
    </order>
    <express>
        <conveyer>神龙速递公司</conveyer>
        <destination>
            <name href="#customerName"/>
            <address href="#customerAddress"/>
        </destination>
    </express>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

可以看到，原来的第一个信息头被去掉了，这是必需的。因为每一条被转发的信息都相当于发送方和接收方重新建立的通话。第二个信息头变为第一个以后，它的 URI 的值就发生了变化，它的 actor 变成 http://schemas.xmlsoap.org/soap/actor/next，表明它将是必须处理的。同样，它还要把信息处理后再次进行转发。第二次转发后的信息如下所示：

```
<?xml version="1.0" encoding="gb2312"?>
<SOAP-ENV:Envelope xmlns:SOAP- ENV="http://schemas. xmlsoap.org/soap
/envelope/">
    <SOAP-ENV:Header>
        <s:shipOrder xmlns:s="http://www.china-publication.com/order"
            SOAP-ENV:mustUnderstand="1"
            SOAP-ENV:actor="http://schemas.xmlsoap.org/soap/actor/next">
            <s:bodyID>shipper</s:bodyID>
        </s:shipOrder>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <customerID ID="2387432"/>
        <customer ID="customer">
            <name ID="customerName" name="张三"/>
            <address id="customerAddress">
                <city>北京</city>
                <details>知春路太月园五号楼二单元 2202</details>
                <zipcode>100036</zipcode>
                <phone>62083412</phone>
```

```
            </address>
        </customer>
        <order ID="products">
            <dvd>神奇燕尾服</dvd>
            <dvd>英雄</dvd>
        </order>
        <express>
            <conveyer>神龙速递公司</conveyer>
            <destination>
                <name href="#customerName"/>
                <address href="#customerAddress"/>
            </destination>
        </express>
    </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

信息最后被送往最终目的地，以完成这个订单的执行过程。actor 属性提供了一种遍历某个文档并在不同的步骤里对不同的元素进行处理的手段。

13.3.3　SOAP 信息体

SOAP 的信息体（Body）是整个 SOAP 的内容核心，它包含着发给接收方的具体内容。它的功能非常强大，在后边还要进行详细介绍，这里就不讲了。

13.3.4　SOAP 错误

有很多原因会使得 SOAP 信息失败，SOAP 对常见的错误原因做出了规定。当一个调用失败的时候，SOAP 解释程序会把错误的原因返回给调用者。例如，HTTP 定义的错误代码，最常见的就是 404（找不到该文件）、403（没有权限，服务器拒绝访问）和 500（服务器执行错误）等。SOAP 没有使用整数作为错误符号，而是使用了 XML 限定的名字。在出错代码里，错误的类别和具体错误名字之间使用“.”来分隔。例如访问者没有权限访问它想要访问的资源，SOAP 程序会返回 Client.Authentication 的错误代码。

在 SOAP 技术标准中得到定义的出错代码主要有以下几类：

- VersionMismatch（版本不匹配）。SOAP 的版本信息是通过 URI 给出的，如前边的 http://schemas.xmlsoap.org/soap/envelop/。
- MustUnderstand（必须理解）。当一个 MustUnderstand 属性的值设定为 1 时，就是告诉接收方必须理解这个元素，如果 SOAP 无法理解这个元素。则返回该错误信息。
- Client（客户）。此类错误是指原始的 SOAP 信息里有错误，可能是信息项不足，或信息格式错误。如果收到这类出错码，首先应该检查的是发送信息方应用程序。
- Server（服务器）。这个类别的错误是指处理信息时出现了错误。出错原因出现在服务器方，可能是上游的处理程序没有得到正确的响应，或者是服务器没有可用磁盘空间等错误。

SOAP 出错时的返回信息的格式应该包含下列几项内容。

- 错误代码：这一项是必不可少的，编码格式前边已经讲过。
- 错误原因：错误原因应该是一个字符串，用易于理解的语言报告到底出现了什么错误。

- 错误报告者：像前边的例子，信息在传输过程中经过几个 SOAP 应用程序，所以在哪里出错必须说明。如果信息只传递到一个 SOAP 应用程序，或者出错的程序是信息的最终用户，这条信息可以省略。
- 错误详细资料：这一项也不是必需的，如果 SOAP 应用程序不能对信息体进行处理，就必须包含这个元素说明原因。如果出错的是信息头，就不会有这条信息了。

13.4 WSDL 语言和 UDDI

自从 SOAP 出现以后，有好多相关的技术标准纷纷被制定出来，而且这些标准还在不断地增加和完善。其中两个重要的标准就是 WSDL（Web 服务定义语言，Web Service Definition Language）和 UDDI（Universal Description Discover and Integration，通用性描述、发现和集成）。

WSDL 是由 IBM、Microsoft 等几家公司联合开发的，并且为了使它成为标准而提交给了 W3C 组织。当前版本是 WSDL1.1。WSDL 就是连接到 Web 服务的接口，它和前边提到过的 IDL（接口定义语言，Interface Definition Language）类似，只不过 WSDL 是连接 Web 服务的，而 IDL 是连接到 CORBA 领域的。除了描述提供的接口外，WSDL 文档还包括服务的位置，也就是规范中所说的终端。服务可以注册在一个可预先指定的大家都知道的地方，允许客户机用一种可靠的方法与这个地方进行连接。所以当需要改变原来配置时，就没有必要分别通知各个客户机。

为了具有更多的灵活性，WSDL 定义了服务中的这种组件，然后可以使用这种组件来定义不同的服务。组成服务的组件包括如下部分：

- 数据类型（type）包括 int、string、object 等。
- 服务输入和输出参数（message）。
- 输入参数与输出参数之间的关系和方法特征，也称为操作（operation）。
- 端口类型（portType）。
- 绑定（binding）。
- 服务（service）。

每一个 WSDL 都是一个 XML 文档，在该文档中都定义了名称空间。WSDL 默认的名称空间是 http://schemas.xmlsoap.org/wsdl/。表 13-2 中列出了常用的名称空间。

表 13-2 WSDL 的常用名称空间

推荐前缀	名称空间
Wsdl	http://schemas.xmlsoap.org/wsdl/
Soap	http://schemas.xmlsoap.org/wsdl/soap/
http	http://schemas.xmlsoap.org/wsdl/http/
Mime	http://schemas.xmlsoap.org/wsdl/mime/
Soapenc	http://schemas.xmlsoap.org/soap/encoding/
Soapenv	http://schemas.xmlsoap.org/soap/envelop
Xsi	http://www.w3.org/2001/XMLSchema-instance
Xsd	http://www.w3.org/2001/XMLSchema

一个 WSDL 文档具有以下元素：

- <definitions>元素。<definitions>元素是每个 WSDL 文档的根元素。在典型的情况下，它包含许多定义 WSDL 文档中使用的命名空间的属性。
- <import>元素。WSDL 文档不需要必须包含在一个文件中，它可以分成多个文件，这样就可以重复使用某些具有共同属性的文件。不同的文件就是通过<import>连接在一起的。
- <types>元素。如果使用 Java 开发 Web Service，就要使用 Java 对象和简单类型（int,float 等）在应用程序的不同部分之间作数据交换，也就是把对象串行化为一个 XML 文档。如果要提供 Web 服务，必须告诉用户期望它们提供何种类型的数据，以及它们希望返回何种类型的数据。下边就是在 WSDL 中<types>元素应用的一个例子。

```
<types>
    <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
        <xsd:element name="bookInfo">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="title" type="xsd:string"/>
                    <xsd:element name="author" type="xsd:string"/>
                    <xsd:element name="price" type="xsd:float"/>
                </xsd:sequence>
            </xsd:complexType>
        </xsd:element>
    </xsd:schema>
</types>
```

<types>元素包含服务要处理的数据类型的定义。几乎在所有的情况下，使用的都是 XML Schema 定义。如果没有特别的说明，就意味着这个服务只使用基本的数据类型，如整数型和字符串等。

- <message>元素。<message>就指的是交换的数据了。一条消息可以由多个参数组成，而这多个参数都是用一条消息代表，为此每条消息包含一个或多个<part>元素，<part>元素引用的必须是在<types>里已定义了的类型。
- <operation>元素和<portType>元素。一个操作<operation>是由一个或多个消息代表的，它有 4 种不同类型的操作。
  - 单向（One Way）。这种操作是指客户机把一条消息传到服务器，而服务器不产生任何回应。
  - 请求响应（Request-Response）。这是最常见的操作类型，客户机把请求发送到服务器，要求接收作为请求结果的响应。
  - 征求响应（Solicit-Response）。该操作是由服务器方发送一条消息到客户方，导致客户机返回给服务器方一条消息。
  - 通知（Notification）。服务器发送一条消息到客户方，而不要求有任何响应。

<operation>元素总是包含在<portType>元素之中，例如下边的这段代码：

```
<portType name="bookInfo">
    <operation name="getBookInfo">
    <input message="bookID"/>
```

```
        <input message="bookInfomation"/>
      </operation>
    </portType>
```

portType 定义的是一组<operation>。每个 operation 最多可以由 3 个元素构成：wsdl:input、wsdl:output 和 wsdl:fault。

- <binding>元素。<binding>元素描述了服务器与客户机通信时使用的机制，这个元素比前边的元素更复杂一些，因为可以放到里边的内容相当多。绑定的作用是为一个给定的 portType 所定义的 operation 和 message 定义信息格式和协议细节。这个元素只能指定一种协议，而不能出现任何地址信息。一个 portType 可以有一个或多个绑定定义。WSDL 作为一种独立的语言，与访问服务的具体机制是没有关系的。到目前为止，SOAP 还是这种访问最主要的选择，还可以使用 HTTP 和 MIME 机制。
- <port>元素。<port>元素是为了绑定到一个地址。WSDL 中的“端口”与 TCP/IP 协议中基于套接字（Socket）进行网络设计师所说的端口的含义是不同的。Socket 中的端口使用正整数代表一个确定某一网络地址的编号。
- <service>元素。<service>的作用是把彼此相关的多个端口编为一组。

前边主要介绍了 WSDL 语言，下面就讲一下如何用 Java 来实现 WSDL。

WSDL 描述了 Web 的接口，怎样把 WSDL 映射到 Java 中呢？请看表 13-3。

表 13-3 WSDL 与 Java 间的映射

WSDL 项	Java 项
<types>	Java 类
<message>	函数的参数或返回值
<operation>	函数
<portType>	Java 类

UDDI 是一个公共的注册表，旨在以一种结构化的方式来保存有关各公司及其服务的信息。通过 UDDI，人们可以发布和发现有关某个公司及其 Web 服务的信息。这些数据使用标准的分类法进行分类，因此可以按分类来查询信息。最重要的是，UDDI 包含有关公司服务的技术接口的信息。通过一套基于 SOAP 的 XML API 调用，用户可以在设计时和运行时与 UDDI 进行交互以发现技术数据，从而调用和使用这些服务。通过这种方法，UDDI 可以用作基于 Web 服务的软件系统的基础结构。

为何使用 UDDI？为何需要这种注册表？当我们面对具有数千甚至数百万个 Web 服务的软件系统时，将面临以下的严峻挑战：

- 如何发现 Web 服务。
- 如何按照某种合理的方式分类信息。
- 对本地化有什么影响。
- 对专用技术有什么影响。如何保障发现机制的互操作性。
- 当应用依赖于某项 Web 服务时，如何在运行时与该发现机制进行交互。

UDDI 的出现正是为了应对这些挑战。为了解决这些问题，许多公司（包括 Microsoft、IBM、Sun、Oracle、Compaq、HP、Intel 等）共同制定了一种基于开放式标准和非专用技术

的规范。该规范的 Beta 版于 2000 年 12 月发布，正式产品于 2001 年 5 月推出。它是一个全球业务注册表，建立在多个运营商节点上，用户可以通过这些节点免费搜索和发布信息。

通过 Web 服务的这种基础结构，现在就能够以一种通用的、与供应商完全无关的方式找到有关 Web 服务的数据，而且数据一致并且可靠。通过使用可扩展的分类系统和标识，用户可以进行精确的分类查询。运行时 UDDI 集成可以被合并到应用程序中去，因而大大地丰富了 Web 服务软件环境。

UDDI 数据存放在运营商（即承诺运营一个公共节点的公司）节点上。这种公共节点遵循 UDDI.org 组织管理的规范。目前已经建立了三个遵循 UDDI 规范版本 1.0 的公共节点，有 IBM 的 http://www-3.ibm.com/services/uddi/、Microsoft 的 http://uddi.mircosoft.com/default.aspx 以及 HP 的 http://uddi.hp.com/uddi/index.jsp 等。数据寄存运营商之间必须能通过安全通道复制数据，从而为整个 UDDI 提供数据冗余。将数据发布到一个节点上后，通过复制就可以在另一个节点上发现这些数据。目前，每隔 24 小时就进行一次复制。在将来，由于有更多的应用程序要依赖 UDDI 数据，复制的时间间隔还将缩短。

值得一提的是，对于数据寄存运营商实现其节点的方式，不存在一些专用的要求，只要其节点遵循 UDDI 规范即可，同时它们都要遵循相同的一套基于 SOAP 的 XML API 调用。客户端工具可以和这些节点进行无缝的交互操作。因此，UDDI 公共团体是一个最佳方案，它展示了 XML Web 服务模型如何跨异类环境进行工作。

为了更进一步了解 UDDI，下一步看看 UDDI 中存储的数据及其存储结构。UDDI 相对来说是轻量的，它被设计为“注册表”，而不是“储备库”。两者之间的差别很微妙，但却很重要。UDDI 的行为与之类似。与 Windows 注册表一样，它依靠全局惟一标识符（GUID）来搜索并定位资源。UDDI 查询最终指向一个接口（.WSDL、.XSD 和.DTD 文件等），或指向其他服务器上的实现。UDDI 因此可以回答以下问题：

- 已经发布了哪些基于 WSDL 并是为指定行业建立的 Web 服务接口。
- 哪些公司已经为其中一个接口写好了实现。
- 目前提供的 Web 服务（以某种方式分类）有哪些。
- 某个公司提供了哪些 Web 服务。
- 如果要使用某个公司的 Web 服务，需要与谁联系。
- 某个 Web 服务的实现细节是什么。

WSDL 已成为 Web 服务协议堆栈的重要组成部分。因此，有必要掌握 UDDI 和 WSDL 如何协同工作，以及每个协议如何解决接口和实现这两个相对的概念。WSDL 和 UDDI 都是为清楚说明抽象的元数据和具体实现之间的关系而设计的，了解为什么要这么划分是理解 WSDL 和 UDDI 的基础。

例如，WSDL 明确区分消息和端口，消息（Web 服务所需的语法和语义）始终是抽象的，而端口（调用 Web 服务的网络地址）始终是具体的。在 WSDL 文件中不需要提供端口信息。WSDL 可以只包含抽象的接口信息，而不提供任何具体的实现数据。这样的 WSDL 文件被认为是有效的。这样，WSDL 文件便从实现中分离出来。其重要意义之一在于：一个 WSDL 接口可以有多个实现。这种设计允许不同的系统为同一接口编写自己的实现，从而保证系统之间能进行对话。如果三个不同的公司实现了相同的 WSDL 文件，一个客户端软件根据这个 WSDL 接口创建了代理/存根代码，这个客户端软件就可以使用相同的代码与所有这三个实现

进行通信，只要更改访问点即可。

UDDI 通过 tModel 的概念描绘了抽象和实现之间的这种区别。tModel 结构（“技术模型”的简称）代表了技术指纹、接口和元数据的抽象类型。使用 tModel 的必然结果是绑定模板，它是一个或多个 tModel 的具体实现。在绑定模板内，要为 tModel 的特定实现注册访问点。如同 WSDL 架构允许分离接口和实现一样，UDDI 也提供了相似的机制，因为 tModel 可以独立于引用它的绑定模板而单独发布。例如，某标准化组织或行业组织可能为特定行业发布规范接口，然后多个公司可以为该接口编写实现。因此，各个公司的实现都需要引用同一个 tModel。WSDL 文件是 UDDI tModel 的完美示例。

发布到 UDDI 是一个比较直接的过程。第一步是确定在 UDDI 上为自己的服务建立模型所需的基本信息，之后便可以进行实际注册。这可通过基于 Web 的用户界面或编程两种方法完成。最后测试您的注册条目以确保注册正确，并且在不同类型的搜索和工具中都能按要求显示。考虑上述数据模型，在建立 UDDI 条目之前应准备好以下几个关键数据：

- 确定 Web 服务实现所需使用的 tModel（WSDL 文件）。开发 Web 服务时可以使用现有的接口，也可以使用自己设计的接口。如果 Web 服务基于现有的 WSDL，则需要确定该 WSDL 文件是否已经在 UDDI 上注册。如果已经注册，就需要记录其名称和 tModel Key，这就是注册 WSDL 文件时 UDDI 所生成的 GUID。另一方面，如果 Web 服务所基于的 WSDL 文件尚未在 UDDI 上注册，就需要准备创建一个新的 tModel 来代表这个接口。该 tModel 应具有一个统一资源标识符（URI）格式的名称，并指向 WSDL 文件所在的位置。

如果需要，可以用多种语言确定自己的名称和简介，以及公司 Web 服务的主要联系方法。UDDI 支持 xml:lang 名称空间，它允许公司用多种语言提供公司简介。另外，UDDI 还允许列出联系方式，包括电子邮件、电话和地址信息。联系列表用于列出公司内与 Web 服务相关的资源。例如，如果有人想要使用您的 Web 服务，并需要联系相应的业务关系经理，应该联系谁？使用公司的 Web 服务时，有关技术问题和谁联系？该联系人也应该列出。

- 确定适当的分类和标识。分类如北美行业分类系统（NAICS）、通用标准产品和服务代码（UNSPSC）、ISO 3166、标准行业分类（SIC）和 GeoWeb 地理分类。请选择一种最适于您的分类。
- 确定公司通过 UDDI 提供的 Web 服务。下一步，确定公司要在公共 UDDI 节点上注册的 Web 服务。这项服务有多个访问点吗？是否要给使用此 Web 服务的客户提供其他必需的参数和信息？

  注意，在 UDDI 上注册 Web 服务并不意味着每个人都有访问权。可以为 UDDI 注册表条目依次设置安全、授权和身份验证。仅知道 Web 服务的存在并不意味着就可以实际调用该服务。在授权访问 Web 服务之前，公司之间通常需要进行一些额外的交流。
- 为服务确定适当的分类。如同可以将公司分类一样，也可以将 Web 服务分类。因此，公司可能按商业级别被分类为 NAICS: Software Publisher (51121)，而旅馆预约 Web 服务的服务级别可能被分类为 NAICS: Hotels and Motels (72111)。

在完成建模之后，下一步就是注册服务了。您需要获取一个可访问 UDDI 注册表的账号。这不能通过编程来完成，因为必须要同意“使用规定”声明。可以通过使用 SOAP API 调用节点自身以编程进行注册。如果不希望更改注册表条目，或条目相对简单，则使用 Web 用户界

面就足够了。如果希望频繁更新，或条目很复杂，就要制作注册过程的脚本。

先介绍一下在 UDDI 注册中心使用 WSDL 的几种不同的方法。WSDL 语言将 Web 服务描述为一组对消息进行操作的网络端点。一个 WSDL 服务描述包含对一组操作和消息的一个抽象定义，绑定到这些操作和消息的一个具体协议，以及这个绑定的一个网络端点规范。

UDDI 提供的是一种发布和查找服务描述的方法。UDDI 数据实体提供对定义业务和服务信息的支持。WSDL 中定义的服务描述信息是 UDDI 注册中心信息的补充。UDDI 提供对许多不同类型的服务描述的支持。因此，UDDI 没有对 WSDL 的直接支持，也没有对任何其他服务描述机制的直接支持。在 UDDI 注册中心有 4 种主要的数据类型：businessEntity、businessService、bindingTemplate 和 tModel。

businessEntity 提供关于商家的信息，可以包含一个或多个 businessService。Web 服务的技术和业务描述在 businessService 和 bindingTemplate 中被定义。每个 bindingTemplate 包含一个对一个或多个 tModel 的引用。tModel 用于定义服务的技术规范。

为帮助在 UDDI 注册中心发布和查找 WSDL 服务描述，WSDL 文档被分为两种类型：服务接口（service interface）和服务实现（service implementations）。一个完整的 WSDL 服务描述是由一个服务接口和一个服务实现文档组成的。由于服务接口表示服务的可重用定义，它在 UDDI 注册中心被作为 tModel 发布。服务实现描述一个服务的实例，每个实例都是使用一个 WSDL service 元素定义的。服务实现文档中的每个 service 元素都被用于发布 UDDI businessService。

当发布一个 WSDL 服务描述时，在服务实现被作为 businessService 发布之前，必须将一个服务接口作为一个 tModel 发布。

在 UDDI 注册中心，服务接口被作为 tModel 发布。tModel 由服务接口提供者发布。tModel 中的一些元素是使用来自 WSDL 服务接口描述中的信息构建的。

13.4.1　WSDL 服务接口实例

```
<?xml version="1.0"?>
<definitions name="StockQuoteService-interface"
  targetNamespace="http://www.getquote.com/StockQuoteService-interface"
  xmlns:tns="http://www.getquote.com/StockQuoteService-interface"
  xmlns:xsd=" http://www.w3.org/2001/XMLSchema  "
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <documentation>
    Standard WSDL service interface definition for a stock quote service.
  </documentation>

  <message name="SingleSymbolRequest">
    <part name="symbol" type="xsd:string"/>
  </message>

  <message name="SingleSymbolQuoteResponse">
    <part name="quote" type="xsd:string"/>
  </message>
```

```
<portType name="SingleSymbolStockQuoteService">
  <operation name="getQuote">
    <input message="tns:SingleSymbolRequest"/>
    <output message="tns:SingleSymbolQuoteResponse"/>
  </operation>
</portType>

<binding name="SingleSymbolBinding"
        type="tns:SingleSymbolStockQuoteService">
  <soap:binding style="rpc"
              transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getQuote">
    <soap:operation soapAction="http://www.getquote.com/GetQuote"/>
    <input>
      <soap:body use="encoded"
          namespace="urn:single-symbol-stock-quotes"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </input>
    <output>
      <soap:body use="encoded"
          namespace="urn:single-symbol-stock-quotes"
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"/>
    </output>
  </operation>
  </binding>
</definitions>
```

其中 definitions 的 targetNamespace 用作 tModel 的名称，documentation 元素的内容用于描述 tModel，binding 绑定名用于限定 overviewURL。

13.4.2 根据 WSDL 服务接口创建的 UDDI tModel

```
<?xml version="1.0"?>
<tModel tModelKey="">
  <name>http://www.getquote.com/StockQuoteService-interface</name>
  <description xml:lang="en">
    Standard service interface definition for a stock quote service.
  </description>
  <overviewDoc>
    <description xml:lang="en">
      WSDL Service Interface Document
    </description>
    <overviewURL>
      http://www.getquote.com/services/
SQS-interface.wsdl#SingleSymbolBinding
    </overviewURL>
  </overviewDoc>
  <categoryBag>
    <keyedReference tModelKey="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4"
                    keyName="uddi-org:types" keyValue="wsdlSpec"/>
```

```
    <keyedReference tModelKey="UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"
                    keyName="Stock market trading services"
                    keyValue="84121801"/>
  </categoryBag>
</tModel>
```

其中 tModel 的名称根据 targetNamespace 设置。描述是根据与 definitions 元素相关联的 documentation 元素设置的。overviewURL 被设置为 WSDL 服务接口文档可通过网络访问到的位置，即 http://www.getquote.com/services/ SQS-interface.wsdl。它还包含对服务接口文档中名为 SingleSymbolBinding 的绑定的直接引用。既然这是服务接口文档中惟一的绑定，对绑定的这个引用就不是必需的。categoryBag 包含 wsdlSpec 条目以及其他的所有 keyedReference，而 keyedReference 指出这个服务接口描述的商业用途。

服务实现在 UDDI 注册中心是作为带有一个或多个 bindingTemplate 的 businessService 发布的，而 businessService 由服务提供者发布。

UDDI businessService 将为服务实现文档中定义的每个 service 元素创建一个新的 businessService。下面的内容包含 businessService 元素列表，这些 businessService 元素可根据 WSDL 服务实现文档的内容创建。

（1）UDDI businessService 描述。

1）name（必须有）。businessService 的 name 元素根据服务实现文档中的 service 元素的 name 属性设置。

2）description（可选项）。description 元素根据 service 元素中的 documentation 元素的内容设置。description 元素的英文值根据与 service 元素关联的 documentation 元素中的前 256 个字符设置。如果 documentation 元素不存在，那么 businessService 的 description 元素就没有被设置。

（2）UDDI bindingTemplate。

新的 bindingTemplate 元素是在 businessService 中为 service 元素中定义的每个 port 元素而定义的。

1）description。如果 port 元素包含一个 documentation 元素，那么就有一个 description 元素是根据 documentation 元素的前 256 个字符设置的。

2）accessPoint。对于一个 SOAP 或 HTTP 绑定，accessPoint 是根据与 port 元素相关联的扩展元素的 location 属性设置的。这个元素将包含 URL，且 URLType 属性是根据此 URL 中的协议设置的。对于不使用 URL 规范的协议绑定，应该使用 URLType 属性指出协议绑定的类型，并且 accessPoint 元素应该包含一个可用于定位使用指定协议的 Web 服务的值。

3）tModelInstanceInfo。bindingTemplate 将包含自己引用的每个 tModel 的一个 tModelInstanceInfo 元素，至少将有一个 tModelInstanceInfo 元素包含对表示服务接口文档的 tModel 的直接引用。

4）overviewURL。overviewURL 元素可能包含对服务实现文档的一个直接引用。对这个文档的引用仅用于提供对人类可读的文档的访问。这个文档中的其他所有信息都应该能够通过 UDDI 数据实体访问。通过维持对原始 WSDL 文档的直接引用，就可以确保被发布的文档就是查找操作返回的那个文档。如果这个文档包含多个端口，那么这个元素应该包含对端口名的直接引用。由于可能会有多个端口引用同一个绑定，只使用 tModel 中的直接引用是不够的。端口名被指定为 overviewURL 上的片段标识符。片段标识符是 URL 的一个扩展，使用“#”

字符作为一个分隔符。

13.4.3 WSDL 服务实现示例

```
<?xml version="1.0"?>
<definitions name="StockQuoteService"
  targetNamespace="http://www.getquote.com/StockQuoteService"
  xmlns:interface="http://www.getquote.com/StockQuoteService-interface"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema "
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns="http://schemas.xmlsoap.org/wsdl/">

  <documentation>
    This service provides an implementation of a standard stock quote service.
    The Web service uses the live stock quote service provided by XMLtoday.com.
    The XMLtoday.com stock quote service uses an HTTP GET interface to request
    a quote, and returns an XML string as a response.

    For additional information on how this service obtains stock quotes, go to
the XMLtoday.com web site: http://www.xmltoday.com/examples/soap/stock.psp.
  </documentation>

  <import namespace="http://www.getquote.com/StockQuoteService-interface"
     location="http://www.getquote.com/wsdl/SQS-interface.wsdl"/>

  <service name="StockQuoteService">
    <documentation>Stock Quote Service</documentation>

    <port name="SingleSymbolServicePort"
       binding="interface:SingleSymbolBinding">
      <documentation>Single Symbol Stock Quote Service</documentation>
      <soap:address location="http://www.getquote.com/StockQuoteService" />
    </port>
  </service>
</definitions>
```

其中 service 元素的 name 属性用作 businessService 的名称。service 元素中的 documentation 元素用于描述 businessService。import 端口名附加到 overviewURL，此 overviewURL 包含对服务实现文档的引用。服务的位置用于设置 bindingTemplate 中的 accessPoint。

13.4.4 根据 WSDL 服务实现创建的 UDDI 商业服务

```
<businessService businessKey="..." serviceKey="...">
  <name>StockQuoteService</name>

  <description xml:lang="en">
   Stock Quote Service
  </description>

  <bindingTemplates>
    <bindingTemplate bindingKey="..." serviceKey="...">
```

```
    <description>
      Single Symbol Stock Quote Service
    </description>
    <accesssPoint URLType="http">
     http://www.getquote.com/singlestockquote
    </accessPoint>
    <tModelInstanceDetails>
      <tModelInstanceInfo tModelKey="[tModel Key for Service Interface]">
        <instanceDetails>
          <overviewURL>
            http://www.getquote.com/services/ SQS.wsdl
          </overviewURL>
        </instanceDetails>
      </tModelInstanceInfo>
    </tModelInstanceDetails>
  </bindingTemplate>
</bindingTemplates>

<categoryBag>
  <keyedReference tModelKey="UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"
                  keyName="Stock market trading services"
                  keyValue="84121801"/>
</categoryBag>
</businessService>
```

其中，businessService 的 name 根据 WSDL 服务实现文档中的 service 的 name 设置。description 根据 service 元素中的 documentation 元素设置。bindingTemplate 的 description 是根据 port 元素中的 documentation 元素设置。accessPoint 根据 soap:address 扩展元素设置。tModelKey 被设置到与服务接口文档关联的 tModel 的 UUID。可使用 import 元素的 namespace 属性对 tModel 进行定位。overviewURL 是服务实现文档的位置。由于它是这个文档中惟一的引用，它不包含对 SingleServiceQuote 端口的引用。

13.4.5　查找 WSDL 服务接口

```
<?xml version="1.0"?>
<find_tModel generic="1.0" xmlns="urn:uddi-org:api">
  <categoryBag>
    <keyedReference tModelKey="UUID:C1ACF26D-9672-4404-9D70-39B756E62AB4"
                    keyName="uddi-org:types" keyValue="wsdlSpec"/>
    <keyedReference tModelKey="UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"
                    keyName="Stock market trading services"
                    keyValue="84121801"/>
  </categoryBag>
</find_tModel>
```

其中，find_tModel 消息将返回一列 tModel 关键字。使用 get_tModelDetail 消息可检索特定的服务接口描述的一个 tModel。

13.4.6　查找 WSDL 服务实现描述

```
<?xml version="1.0"?>
```

```
<find_service businessKey="..." generic="1.0" xmlns="urn:uddi-org:api">
  <categoryBag>
    <keyedReference tModelKey="UUID:DB77450D-9FA8-45D4-A7BC-04411D14E384"
                    keyName="Stock market trading services"
                    keyValue="84121801"/>
  </categoryBag>
</find_service>
```

其中，find_service 消息也可用于定位 businessService，该 businessService 是某个特定服务接口的实现。其中的消息包含 find_service 消息的一个示例，它将查找 businessEntity 中的所有 businessService，这个 businessEntity 是实现了股票报价服务的服务接口。由于服务接口是用 tModel 来表示，tModelBag 就被用于指定 WSDL 服务接口与股票报价服务相关联的 tModel 关键字。

13.4.7 查找 UDDI bindingTemplate

```
<?xml version="1.0"?>
<find_binding serviceKey="..." generic="1.0" xmlns="urn:uddi-org:api">
  <tModelBag>
    <tModelKey>[tModelKey for WSDL service interface]<tModelKey/>
  </tModelBag>
</find_service>
```

这条消息将返回一个或多个 bindingTemplate。访问过 bindingTemplate 之后，accessPoint 中列出了 Web 服务的端点。如果 bindingTemplate 是根据现有的 WSDL 服务实现文档创建的，那么 overviewURL 可能包含一个对这个文档的引用。可访问这个文档获取额外的、人类可读的关于 Web 服务的信息，这些信息是在 UDDI 注册中心找不到的。

13.4.8 UDDI bindingTemplate 示例

```
<?xml version="1.0"?>
<bindingTemplate bindingKey="" serviceKey="">
  <accesssPoint URLType="http">
    http://www.getquote.com/singlestockquote
  </accessPoint>
  <tModelInstanceDetails>
    <tModelInstanceInfo tModelKey="[tModel Key for Service Interface]">
      <instanceDetails>
        <overviewURL>
          http://www.getquote.com/services/ SQS.wsdl
        </overviewURL>
      </instanceDetails>
    </tModelInstanceInfo>
  </tModelInstanceDetails>
</bindingTemplate>
```

13.5 JSP 调用 Web Service 范例

前边第一个使用 SOAP 的 HelloWorld 范例，已经示范了 Web Service 的工作原理，但是它

仅仅是一个字符串的往返，还远远满足不了需要，这里的例子就是演示一个比较复杂的应用，而且可以通过 JSP 来调用它。

服务器端仍然使用 Apache SOAP。Apache SOAP 的安装前边已经讲了，这里就不多加叙述了。使用 JSP 调用 Web Service 和服务器之间或客户端程序调用 Web Service 并没有什么不同，如果在一台电脑上就没有使用 Web Service 的必要了。为了能测试这个程序，要开启至少两个服务，一个服务运行 Web 程序，也就是 JSP 程序，而另一个服务上运行 Apache SOAP。

可以这样使用 Tomcat 来运行这两个服务：首先在$CATALINA_HOME 下建立一个目录，叫做 jws，即 Java Web Service 的简写形式，很明显要在这个目录下运行 Web Service，并把上边讲过的 soap 整个目录移过来。然后修改配置文件$CATALIN_HOME\conf\server.xml，先删掉前边加进去的<Context path="/ soap" docBase="soap" debug="1" reloadable="true"/>。找到最后边的</Service>，并在这个标记的后边加上下边这些语句：

```
<Service name="Tomcat-Standalone2">
  <Connector className="org.apache.catalina.connector.http.HttpConnector"
             port="8080" minProcessors="5" maxProcessors="75"
             enableLookups="true" redirectPort="8444"
             acceptCount="10" debug="0" connectionTimeout="60000"/>
  <Connector className="org.apache.ajp.tomcat4.Ajp13Connector"
             port="8010" minProcessors="5" maxProcessors="75"
             acceptCount="10" debug="0"/>
  <Engine name="Standalone2" defaultHost="localhost" debug="0">
    <Logger className="org.apache.catalina.logger.FileLogger"
           prefix="catalina_log." suffix=".txt"
           timestamp="true"/>
    <Realm className="org.apache.catalina.realm.MemoryRealm" />
    <Host name="localhost" debug="0" appBase="jws" unpackWARs="true">
      <Valve className="org.apache.catalina.valves.AccessLogValve"
             directory="logs"  prefix="localhost_access_log." suffix=".txt"
             pattern="common"/>
      <Logger className="org.apache.catalina.logger.FileLogger"
             directory="logs"  prefix="localhost_log." suffix=".txt"
        timestamp="true"/>
     <Context path="/soap" docBase="soap" debug="0" reloadable="true"/>
  </Host>
  </Engine>
</Service>
```

注意其中端口的设置，由于前边的主服务开的是 80 端口，而新开的服务使用 8080 端口。如果你的主服务在 8080 端口，可以换用其他的端口。如果开启 Tomcat 时没有报错，说明设置的端口就是可以使用的了。

13.5.1　实例说明

这个例子的主要功能是远程调用数据库操作。如果有不同的站点需要公用数据库数据，可以使用数据库的远程调用手段。一旦数据库配置需要改变，比如为了保证安全定期修改密码，或数据库服务器的 IP 地址发生了变化，还要分别修改源程序。使用 Web Service 就可以避免这些麻烦，而且有利于数据库的安全。同时如果开发客户端软件，进行用户管理也可以通

过使用 Web Service，从而大大地降低开发难度和开发周期。

这个例子还是使用原来的数据库表进行用户管理的操作，使用 Mysql 的 userinfo 数据表。数据库的结构如下：

```
create table userinfo(
    id int primary key auto_increment,
    username varchar(20) not null,
    password varchar(20) not null,
    phone    varchar(20),
    email    varchar(40)
);
```

由于进行信息交换时，发送者与接收者之间只能使用 XML 传送，所以 Java 类型的数据必须要转换为 XML 数据，XML 数据也必须可以转换为 Java 类型数据，也就是可串行化的。这种串行化的操作当然可以通过编程自己解决，但是 Apache SOAP 已经做了这项工作。Apache SOAP 是通过类型映射实现串行化的，默认的类型映射的 Java 类是 org.apache.soap.encoding.SOAPMappingRegistry。这个类包括对原始 Java 类型（如 int、float 等）的映射、对原始 Java 类型的包装类（如 Integer、Float 等）和一些其他有用的类型的映射。每个类型映射都要包括下列信息：

- 描述编码方式的 URI。
- XML 元素的有效名称（Qname）。
- 实现类型的 Java 类。
- 对串行化到类型和从类型并行化产生相应的类。

Apache SOAP 除了实现前边讲过的默认类型的串行化器外，还提供了对于 JavaBean 的串行化器，它通过 org.apache.soap.encoding.soapenc.BeanSerializer 类实现。当使用的类与 JavaBean 规范定义的规则子集相兼容时，BeanSerializer 类就可以处理这个类的串行化。但并不是包括所有的 JavaBean 类型，这些类型必须符合下列规范：

- 有一个无参数的构建函数。
- 对全部属性都有（setXxx/getXxx）方法。
- 与对属性的调用顺序无关。

下边就通过代码实例详细讲解一下 Apache SOAP 实现这些功能的具体过程。

13.5.2 代码分析

UserInfo.java 是一个符合 Apache SOAP 所要求规范的 JavaBean，它的代码如下所示：

```
package user;
public class UserInfo extends Object
{
    private String userId;
    private String username;
    private String password;
    private String phone;
    private String email;
    public UserInfo(String userId,String username,
String password,String phone,String email){
        this.userId = userId;
```

```
        this.username = username;
        this.password = password;
        this.phone = phone;
        this.email = email;
    }
    public UserInfo(){
    }
    public void setUserId(String userId){
        this.userId=userId;
    }
    public void setUsername(String username){
        this.username=username;
    }
    public void setPassword(String password){
        this.password=password;
    }
    public void setPhone(String phone){
        this.phone=phone;
    }
    public void setEmail(String email){
        this.email=email;
    }
    public String getUserId(){
        return userId;
    }
    public String getUsername(){
        return username;
    }
    public String getPassword(){
        return password;
    }
    public String getPhone(){
        return phone;
    }
    public String getEmail(){
        return email;
    }
}
```

执行数据操作的类 UserManage.java 的代码如下所示：

```
package user;
import java.sql.*;

public class UserManage
{
    //通过用户的 ID 号得到用户的详细信息
    //注意返回值是一个 UserInfo 类型，BeanSerializer 要对它串行化
    public UserInfo getUserInfo(String userId){
        UserInfo user=new UserInfo();
        //得到数据连接
        Connection conn=getConnection();
```

```
if(conn!=null){
    try{
        //执行数据库操作
        String sql="select * from userinfo where id="+userId;
        Statement stmt = conn.createStatement();
        ResultSet rs=stmt.executeQuery(sql);
        if(rs!=null&&rs.next()){
            user.setUserId(userId);
            user.setUsername(rs.getString("username"));
            user.setPassword(rs.getString("password"));
            user.setPhone(rs.getString("phone"));
            user.setEmail(rs.getString("email"));
        }else{
            System.err.println("访问数据库错误");
        }
    }catch(Exception e){
        System.err.println(e);
    }finally{
        try{    //关闭数据库连接
            conn.close();
        }catch(Exception sqle){
        }
    }
}else{
    System.err.print("连接数据库错误");
}
        return user;
    }
    //修改用户信息，参数为 UserInfo 类型
    public String updateUserInfo(UserInfo user){
        //获得数据库连接
        Connection conn=getConnection();
        if(conn!=null){
            try{
                //执行数据库操作，修改用户信息
                String sql="update userinfo set username=?, password=?, phone=?,"
+"email=? where id="+user.getUserId();
                PreparedStatement pstmt = conn.prepareStatement(sql);
                pstmt.setString(1,user.getUsername());
                pstmt.setString(2,user.getPassword());
                pstmt.setString(3,user.getPhone());
                pstmt.setString(4,user.getEmail());
                pstmt.executeQuery();
                pstmt.close();
                return "1";
            }catch(Exception e){
                System.err.println("修改用户信息失败"+e);
                return "0";
            }finally{
                try{    //关闭数据库连接
```

```
                    conn.close();
                }catch(Exception sqle){
                }
            }
        }else
            System.err.println("数据库连接失败");
        return "0";
    }
    //创建新用户
    public  String  CreateUser(String  username,String  password,String
phone,String email){
        //获得数据库连接
        Connection conn=getConnection();
        if(conn!=null){
            try{
                //检测用户名是否已经存在，如果存在则返回添加新用户失败
                String sql="select count(id) as count from userinfo where
    username='"+username+"'";
                Statement stmt = conn.createStatement();
                ResultSet rs=stmt.executeQuery(sql);
                if(rs!=null&&rs.next()&&rs.getInt("count")==0){
                    //执行数据库操作，添加新用户
                   sql="insert into userinfo (username,password,phone,email)
values(?,?,?,?)";
                    PreparedStatement pstmt = conn.prepareStatement(sql);
                    pstmt.setString(1,username);
                    pstmt.setString(2,password);
                    pstmt.setString(3,phone);
                    pstmt.setString(4,email);
                    pstmt.executeQuery();
                    pstmt.close();
                    //得到新添加用户的 ID，由于用户名是一个 Unique 的数据类型
                    //可以通过这样的操作获得用户 ID
                    sql="select id from userinfo where username='"+username+"'";
                    rs=stmt.executeQuery(sql);
                    if(rs!=null&&rs.next()){
                        int id=rs.getInt("id");
                        return new Integer(id).toString();
                    }else{
                        return "服务器内部错误";
                    }
                }else{
                    return "用户名已经存在";
                }
            }catch(Exception e){
                System.err.println(e);
                return "连接数据库错误";
            }finally{
                try{    //关闭数据库连接
                    conn.close();
```

```
                }catch(Exception sqle){
                }
            }
        }else
            return "连接数据库错误";
    }
    //获得数据库连接
    public Connection getConnection(){
        try{
            Class.forName("com.mysql.jdbc.Driver");
            String sConnStr = "jdbc:mysql://localhost/test?user=root
&password=password&useUnicode=true&characterEncoding=gb2312";
            //返回数据连接
            return DriverManager.getConnection(sConnStr);
        }catch(Exception e){
            System.err.println("连接数据库错误: "+e);
            return null;
        }
    }
}
```

把这两个文件放到$CATALINA_HOME\jws\soap\WEB-INF\classes 中，然后就需要部署到 Web 服务中了。可以通过下边的部署文件部署服务，dd.xml 的代码如下所示：

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
                id="urn:userInfomation">
    <isd:provider type="java" scope="Application"
    methods="getUserInfo updateUserInfo createUser">
        <isd:java class="user.UserManage" static="false"/>
    </isd:provider>
    <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultL
istener>
    <isd:mappings>
        <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          xmlns:x="urn:manageuser" qname="x:user"
          javaType="user.UserInfo"
          java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
          xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
    </isd:mappings>
</isd:service>
```

由于 UserInfo 类必须通过 Apache SOAP 客户机和服务器间进行交换，所以必须要有一个映射元素。该映射元素包含着一个或多个定义实际类型映射的<map>元素，encodingStyle 属性用于指定映射需要的编码方式。xmlns 和 qname 属性的作用是指定被定义类型的元素名称和名称空间。用 java2XMLClassName 和 xml2JavaClassName 来指定该类型的串行化器（serializier）和并行化器（deserializer）。

使用以下命令来部署该服务：

```
java org.apache.soap.server.ServiceManagerClient
http://localhost:8080/soap/servlet/rpcrouter deploy dd.xml
```

服务部署的情况可以通过访问 http://localhost:8080/soap/admin/来查看。

然后就是调用该服务了，这里还是使用 JavaBean 来调用该服务，getUserInfo.java 的代码如下所示：

```
package user;
// 引用需要的 Apache SOAP API
import org.apache.soap.encoding.SOAPMappingRegistry;
import org.apache.soap.encoding.soapenc.BeanSerializer;
import org.apache.soap.Constants;
import org.apache.soap.SOAPException;
import org.apache.soap.Fault;
import org.apache.soap.util.xml.QName;
import org.apache.soap.rpc.Call;
import org.apache.soap.rpc.Parameter;
import org.apache.soap.rpc.Response;
import java.net.URL;
import java.util.Vector;

public class getUserInfo
{
    //传递给服务器端一个 UserID 的值，然后接收返回的 UserInfo 类
   public UserInfo getUserInfomation(String uid)
    {
        UserInfo value=null;
        //提供服务的 URL
        String RPCROUTER = "http://localhost:8080/soap/servlet/rpcrouter";
      //初始化串行化器
 SOAPMappingRegistry smr = new SOAPMappingRegistry();
BeanSerializer beanSer = new BeanSerializer();
 // 生成类的映射
 smr.mapTypes(Constants.NS_URI_SOAP_ENC,
     new QName("urn:manageuser", "user"),
          UserInfo.class, beanSer, beanSer);
// 初始化调用服务的对象
Call call = new Call();
call.setSOAPMappingRegistry(smr);
 //服务名称
call.setTargetObjectURI("urn:userInfomation");
 //需要使用的函数名称
call.setMethodName("getUserInfo");
call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
// Add the parameters
Vector params = new Vector();
 //加入参数
params.addElement(new Parameter("uid", String.class,uid, null));
call.setParams(params);
// 生成调用
 Response resp = null;
try{
    URL url = new URL(RPCROUTER);
    resp = call.invoke(url, "");
}
```

```
    catch (SOAPException e)
     {    //捕捉 SOAP 错误
         System.err.println("Caught SOAPException (" +
                            e.getFaultCode() + "): " +
                            e.getMessage());
         return value;
     }catch(Exception ex){
         ex.printStackTrace();
     }
    // 接收返回值
    if (!resp.generatedFault()) //Successful response
    {
       Parameter ret = resp.getReturnValue();
         //getValue ()方法的返回值为 Object 类型
       value = (UserInfo) ret.getValue();
         return value;
     }
    else      //返回失败信息
    {
       Fault fault = resp.getFault();
       System.err.println("Generated fault: ");
       System.err.println ("  Fault Code   = " + fault.getFaultCode());
       System.err.println ("  Fault String = " + fault.getFaultString());
    }
     return value;
}
}
```

下边再看一下使用这个 Bean 的 JSP 代码，这就很简单了，Userinfo.jsp 的代码如下所示：

```
<%@ page language="java" import="user.UserInfo" %>
<%@ page contentType="text/HTML;charset=GB2312"%>
<jsp:useBean id="user" scope="page" class="user.getUserInfo" />
<%
    String uid=request.getParameter("id");
    if(uid==null&&uid.equals(""))
        uid="5";

    UserInfo ui=user.getUserInfomation(uid);
    if(ui==null)
        out.println("error");
    else{
%>
<html>
<body>
姓名：<%=ui.getUsername()%><br>
电话：<%=ui.getPhone()%><br>
邮件地址：<%=ui.getEmail()%><br>
</body>
</html>
<%}%>
```

13.5.3　运行结果

userinfo.jsp 页面的显示结果如图 13-7 所示。

图 13-7　userinfo.jsp 的显示结果

13.6　小结

本章主要介绍了 Web Service 的基本原理，包括 SOAP 的信息结构，WSDL 语言和 UDDI 注册，还介绍了如何使用 Apache SOAP 以及用 Java 语言开发 SOAP 服务。通过本章的学习，读者应该可以了解 Web Service 的原理，并且可以开发简单的 Web 服务了。

第 14 章　消息中间件概述

14.1　消息传递概述

对于一个企业信息系统来说，在分布式系统的应用程序之间传递消息是很普遍的。当 CORBA、RMI 和 DCOM 等执行分布式对象调用时也要进行通信。消息传递服务和那些面向过程的远程通信模型之间的区别在于，它在通信模型之上增加了一个抽象层，从而组合了分布式消息发送者和接收者之间的连接。此外，在消息生产者和消息消费者之间存在一个中间服务（即中介服务），它负责直接将消息从生产者传送给消费者。另外，生产者并不在它们传送的消息之上阻塞。

消息传递服务是一个软件，它以一种可靠的、异步的、松散耦合的、语言无关的方式在分布式应用程序之间传递消息，通常还是可以配置的。消息传递服务对发送者和接收者之间的消息进行封装，通过在分布式消息传递客户之间提供一个软件等途径来完成这一任务。消息传递服务也为消息传递客户提供了一个接口，用以隔离底层的消息服务实现，这样异构客户之间可以通过对编程人员友好的接口进行通信。这样的架构也可以看作一种事件通知类型服务，其中消息是事件，在传递消息的客户之间传送这些消息就充当一种事件通知机制。与用来处理更为普遍的消息传递而设计的消息中间件相比，事件通知类型的消息传递系统一般设计为处理更简单的消息。

14.1.1　消息传递服务实现模型

图 14-1 展示了消息传递服务的一种实现模型，其中的中间件实现了消息服务的功能，它异步地从消息生产者接收消息，并且将消息路由到消息消费者。要传递消息的客户使用消息中间件接口，以一种透明的方式调用消息传递服务。

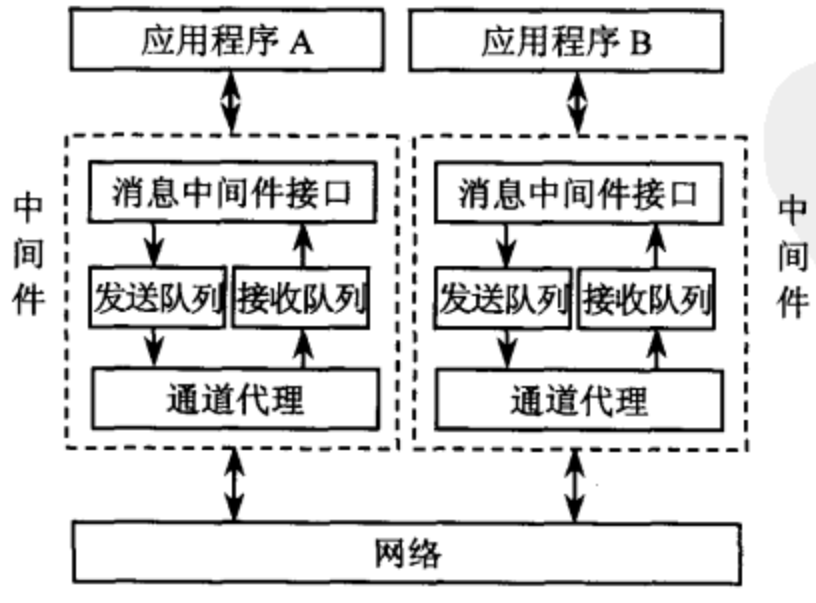


图 14-1　消息传递中间件

消息传递中间件（简称消息中间件）采用集中管理的持久机制和冗余机制，增强了消息传递服务的可靠性和可用性。消息传递中间件允许在消息传递客户和消息传递中间件之间建立连接，这样就减轻了客户管理与多个消息传递服务端点位置之间的连接任务。

14.1.2　点到点的消息传递

图 14-2 显示了一种类型的消息传递服务，它用于消息生产者和消息消费者之间点到点的通信。消息生产者（Client 1）将消息发送到由某个名字标识的特定消费者（Client 2）。这个名字实际上对应于消息服务中的一个队列（Queue），在消息传送给消费者之前它被存储在这个队列中。队列可以是持久的，以保证在消息服务出现故障时仍然能够传递消息。也就是说，即使主消息服务出现故障，备份消息服务实例仍然可以从持久队列中读取消息，并且将消息传递给消费者。

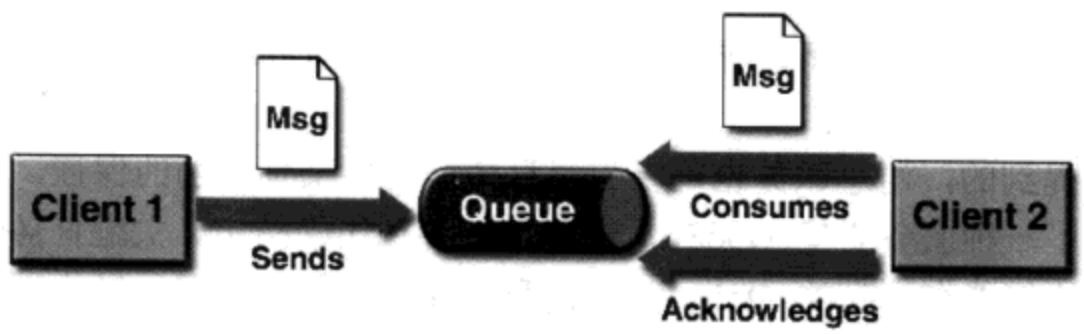


图 14-2　点到点消息队列服务

14.1.3　发布—订阅消息传递

图 14-3 显示了另外一种流行的消息传递模型，即发布—订阅消息传递服务。利用发布—订阅模型，消息发布者（Client 1）可以将消息发布到特定的主题（Topic）。然后按层的方式组织主题，并且在一个特定的主题上下文内，进一步允许发布和订阅以传递消息。

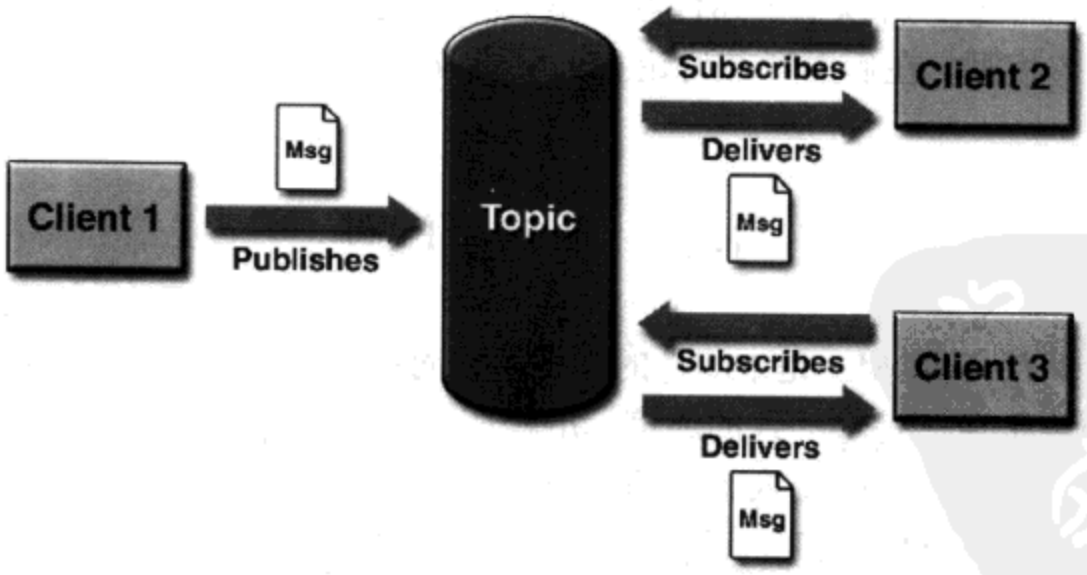


图 14-3　发布—订阅消息传递服务

14.1.4　“推”消息传递模型和“拉”消息传递模型

消息传递方式也可以根据消息“推”或“拉”模型进行区分。“推”消息模型是最典型的

一种，在这种模型中的消息生产者将一个消息发送给消息传递服务，消息传递服务又将消息推给消息消费者。

“拉”消息模型涉及另一种情况：消费者请求消息服务接收一个消息，消息服务从消息生产者拉该消息。

14.1.5 消息过滤、同步和质量

当一个消息消费者希望基于一些消息属性只接收某些消息时，可以利用消息过滤模式。消息过滤需要给予一些过滤规则判别哪些消息应当传递给消费者，哪些消息不应当传递给消费者。过滤规则以某种语言描述并且可以引用消息中的属性和值。

虽然存在各种各样的消息传递模型，所有的消息服务共享一个公共的异步属性。消息是从消息生产者发送给消息消费者的，且并不要求消息生产者阻塞处理直到消息被收到。但是典型的远程激发协议（如 CORBA 和 RMI）默认情况下的确实现了一些同步调用，即分布式客户的确阻塞直到调用完成。所以异步调用就为消息生产者和消息消费者提供了某种级别的时间无关性。

正如前面所提到的，消息服务也可以为消息传递提供某种级别的保证。消息传送的可靠性和可用性提供了一种服务质量（QoS），它可以在各种级别指定。关键任务消息的传送与常规消息的传送相比可能有更高的 QoS。许多消息传递服务都提供了一种方法可以在连接、消息类型和单独的消息级别之上指定 QoS。

14.1.6 电子邮件消息传递

在已经存在的各种类型的消息传递服务之中，传送和接收电子邮件消息可能是读者最熟悉和最为广泛使用的消息传递类型之一。电子邮件服务器将发送给特定电子邮件地址的消息排队，这样电子邮件用户以后才能检索这些消息。电子邮件客户软件包含一种传输机制，将邮件发送给邮件服务器。邮件服务器将电子邮件放在一个消息存储器中，这样以后才可以被电子邮件客户软件下载。所以电子邮件客户是一种典型的点到点类型的消息传递服务，它采用消息传递“拉”模型。在后面章节讨论 JavaMail 服务时，将会讨论到电子邮件消息传递服务标准。

14.2 消息中间件

消息传输中间件（MOM）是中间件的一种。它简化了应用之间数据的传输，屏蔽底层异构操作系统和网络平台，提供一致的通信标准和应用开发，确保分布式计算网络环境下可靠的、跨平台的消息传输和数据交换。它基于消息队列的存储—转发机制，并提供特有的异步传输机制，能够基于消息传输和异步事务处理实现应用整合与数据交换。

14.3 WebSphere MQ 概述

IBM 消息中间件 MQ 以其独特的安全机制、简便快速的编程风格、卓越不凡的稳定性、可扩展性和跨平台性，以及强大的事务处理能力和消息通信能力，成为业界市场占有率最高

的消息中间件产品。

MQ 具有强大的跨平台性，支持的平台数多达 35 种。它支持各种主流 UNIX 操作系统平台，如：HP-UX、AIX、SUN Solaris、Digital UNIX、Open VMX、SUNOS、NCR UNIX；支持各种主机平台，如：OS/390、MVS/ESA、VSE/ESA；同样支持 Windows NT 服务器。在 PC 平台上支持 Windows 9X/Windows NT/Windows 2000 和 UNIX（UnixWare、Solaris）以及主要的 Linux 版本（Redhat、TurboLinux 等）。此外，MQ 还支持其他各种操作系统平台，如：OS/2、AS/400、Sequent DYNIX、SCO OpenServer、SCO UnixWare、Tandem 等。

14.3.1　MQ 的基本概念

MQ 包括如下基本概念：

1. 队列管理器

队列管理器是 MQ 系统中最上层的一个概念，由它提供基于队列的消息服务。

2. 消息

在 MQ 中，把应用程序交由 MQ 传输的数据定义为消息，可以定义消息的内容并对消息进行广义的理解，比如：用户的各种类型的数据文件，某个应用向其他应用发出的处理请求等都可以作为消息。消息由两部分组成。

（1）消息描述符（Message Description 或 Message Header），描述消息的特征，如：消息的优先级、生命周期、消息 ID 等。

（2）消息体（Message Body），即用户数据部分。在 MQ 中，消息分为两种类型，非持久性（non-persistent）消息和持久性（persistent）消息。非持久性消息是存储在内存中的，是为了提高性能而设计的，当系统掉电或 MQ 队列管理器重新启动时，将不可恢复。当用户对消息的可靠性要求不高，而侧重系统的性能表现时，可以采用该种类型的消息，如：当发布股票消息时，由于股票消息是不断更新的，我们可能每若干秒就会发布一次，新的消息会不断覆盖旧的消息。持久性消息是存储在硬盘上，并且记录数据日志的，它具有高可靠性，在网络和系统发生故障等情况下都能确保消息不丢失、不重复。

此外，在 MQ 中，还有逻辑消息和物理消息的概念。利用逻辑消息和物理消息，可以将大消息进行分段处理，也可以将若干个本身完整的消息在应用逻辑上归为一组进行处理。

3. 队列

队列是消息的安全存放地，队列存储消息直到它被应用程序处理。

消息队列以下述方式工作：

（1）程序 A 形成对消息队列系统的调用，此调用告知消息队列系统，消息准备好了投向程序 B。

（2）消息队列系统发送此消息到程序 B 驻留处的系统，并将它放到程序 B 的队列中。

（3）适当时间后，程序 B 从它的队列中读此消息，并处理此消息。

由于采用了先进的程序设计思想以及内部工作机制，MQ 能够在各种网络条件下保证消息的可靠传递，可以克服网络线路质量差或不稳定的现状。在传输过程中，如果通信线路出现故障或远端的主机发生故障，本地的应用程序都不会受到影响，可以继续发送数据，而无须等待网络故障恢复或远端主机正常后再重新运行。

在 MQ 中，队列分为很多种类型，其中包括本地队列、远程队列、模板队列、动态队列、

别名队列等。

本地队列又分为普通本地队列和传输队列，普通本地队列是应用程序通过 API 对其进行读写操作的队列；传输队列可以理解为存储—转发队列，比如：将某个消息交给 MQ 系统发送到远程主机，而此时网络发生故障，MQ 将把消息放在传输队列中暂存，当网络恢复时，再发往远端目的地。

远程队列是目的队列在本地的定义，它类似一个地址指针，指向远程主机上的某个目的队列，它仅仅是个定义，不真正占用磁盘存储空间。

模板队列和动态队列是 MQ 的一个特色，它的一个典型用途是用作系统的可扩展性考虑。可以创建一个模板队列，当今后需要新增队列时，每打开一个模板队列，MQ 便会自动生成一个动态队列，还可以指定该动态队列为临时队列或者是永久队列，若为临时队列可以在关闭它的同时将它删除，相反，若为永久队列，可以将它永久保留，为我所用。

4. 通道

通道是 MQ 系统中队列管理器之间传递消息的管道，它是建立在物理的网络连接之上的一个逻辑概念，也是 MQ 产品的精华。

在 MQ 中，主要有三大类通道类型，即消息通道、MQI 通道和 Cluster 通道。消息通道是用于在 MQ 的服务器和服务器之间传输消息的，需要强调指出的是，该通道是单向的，它又有发送（sender）、接收（receive）、请求者（requestor）、服务者（server）等不同类型，供用户在不同情况下使用。MQI 通道是 MQ Client 和 MQ Server 之间通信和传输消息用的，与消息通道不同，它的传输是双向的。群集（Cluster）通道是位于同一个 MQ 群集内部的队列管理器之间通信使用的。

14.3.2 MQ 的工作原理

MQ 的工作原理如图 14-4 所示。

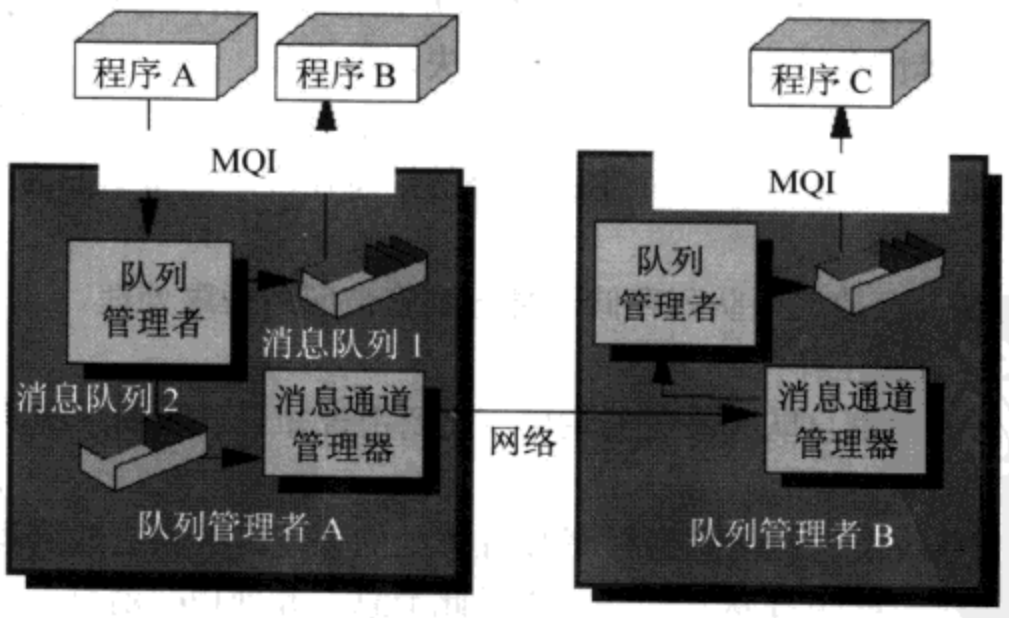


图 14-4 MQ 的工作原理

首先来看本地通信的情况，应用程序 A 和应用程序 B 运行于同一系统 A，它们之间可以借助消息队列技术进行彼此的通信：应用程序 A 向队列 1 发送一条消息，而当应用程序 B 需要时就可以得到该消息。

其次是远程通信的情况，如果消息传输的目标改为在系统 B 上的应用程序 C，这种变化不

会对应用程序 A 产生影响，应用程序 A 向队列 2 发送一条消息，系统 A 的 MQ 发现 Q2 所指向的目的队列实际上位于系统 B，它将消息放到本地的一个特殊队列——传输队列（Transmission Queue）上。我们建立一条从系统 A 到系统 B 的消息通道，消息通道代理将从传输队列中读取消息，并传递这条消息到系统 B，然后等待确认。只有 MQ 接到系统 B 成功收到消息的确认之后，它才从传输队列中真正将该消息删除。如果通信线路不通，或系统 B 不在运行，消息会留在传输队列中，直到被成功地传送到目的地。这是 MQ 最基本且最重要的技术——确保消息传输，并且是一次且仅一次（once-and-only-once）的传递。

MQ 提供了用于应用集成的松耦合的连接方法，因为共享消息的应用不需要知道彼此的物理位置（网络地址）；不需要知道彼此间怎样建立通信；不需要同时处于运行状态；不需要在同样的操作系统或网络环境下运行。

14.3.3　MQ 的基本配置举例

在图 14-4 中，要实现网络上两台主机间的通信，若采用点对点的通信方式，至少要建立如下 MQ 的对象。

在发送方 A：

（1）建立队列管理器 QMA：

crtmqm -q QMA

（2）定义本地传输队列：

define qlocal (QMB) usage (xmitq) defpsist(yes)

（3）创建远程队列：

define qremote (QR.TOB) rname (LQB) rqmname (QMB) xmitq (QMB)

（4）定义发送通道：

define channel (A.TO.B) chltype (sdr) conname ('IP of B') xmitq (QMB) + trptype (tcp)

在接收方 B：

（1）建立队列管理器 QMB：

crtmqm -q QMB

（2）定义本地队列 QLB：

define qlocal (LQB)

（3）创建接收通道：

define channel (A.TO.B) chltype (rcvr) trptype (tcp)

经过上述配置，就可以实现从主机 A 到主机 B 的单向通信，若要实现二者之间的双向通信，可参考此例创建所需要的 MQ 对象。

14.3.4　MQ 的通信模式

WebSphere MQ 包括如下通信模式：

（1）点对点通信。点对点方式是最为传统和常见的通信方式，它支持一对一、一对多、多对多、多对一等多种配置方式，支持树状、网状等多种拓扑结构。

（2）多点广播。MQ 适用于不同类型的应用。其中重要的，也是正在发展中的是“多点广播”应用，即能够将消息发送到多个目标站点（Destination List）。可以使用一条 MQ

指令将单一消息发送到多个目标站点，并确保为每一站点可靠地提供消息。MQ 不仅提供了多点广播的功能，而且还拥有智能消息分发功能，在将一条消息发送到同一系统上的多个用户时，MQ 将消息的一个复制版本和该系统上接收者的名单发送到目标 MQ 系统。目标 MQ 系统在本地复制这些消息，并将它们发送到名单上的队列，从而尽可能减少网络的传输量。

（3）发布/订阅（Publish/Subscribe）模式。发布/订阅功能使消息的分发可以突破目的队列地理指向的限制，使消息按照特定的主题甚至内容进行分发，用户或应用程序可以根据主题或内容接收到所需要的消息。发布/订阅功能使得发送者和接收者之间的耦合关系变得更为松散，发送者不必关心接收者的目的地址，而接收者也不必关心消息的发送地址，而只是根据消息的主题进行消息的收发。在 MQ 家族产品中，MQ Event Broker 是专门使用发布/订阅技术进行数据通信的产品，它支持基于队列和直接基于 TCP/IP 两种方式的发布和订阅。

（4）群集（Cluster）。为了简化点对点通信模式中的系统配置，MQ 提供 Cluster（群集）的解决方案。群集类似于一个域（Domain），群集内部的队列管理器之间通信时，不需要两两之间建立消息通道，而是采用群集（Cluster）通道与其他成员通信，从而大大简化了系统配置。此外，群集中的队列管理器之间能够自动进行负载均衡，当某一队列管理器出现故障时，其他队列管理器可以接管它的工作，从而大大提高系统的可靠性。

14.3.5 MQ Server 和 MQ Client

MQ 产品分为 Server 和 Client 两种版本，在 MQ 服务器的运行环境下，有队列管理器、队列、消息通道等对象，它提供全面的消息服务；MQ Client 提供了一个 MQ 应用程序的开发和运行环境，它是 MQ API 的 Client 实现。在客户端环境下，没有队列管理器、队列等对象，它通过 MQI 通道与服务器之间建立通信，并将消息从客户端发往服务器端的队列，或从 Server 端的队列中取得消息，它比较适合于网络条件较好或实时通信的情况。同时要指出的是，采用 MQ Client 并不会导致数据的丢失或不完整性。MQ Client 提供下列好处：适合同步处理的工作模式；减少系统负担；减少系统管理开销；减少磁盘空间要求等。

14.3.6 MQ 的 API

MQ 支持多种编程语言，其中包括 C、C++、Java、Visual Basic、COBOL、PL/1、RPG 等，同时也支持多种流行的开发工具，如 WebSphere Studio Application Developer、PowerBuiler、Microsoft Visual C++、Visual Basic、Delphi 等。并且，MQ 在不同平台上提供统一的编程接口，仅需重新编译就可完成不同平台间程序的移植。MQ 的 API 接口十分简单易学，用户仅需利用 MQ 的 13 个常用而又功能强大的函数调用，便可以以最快的速度，写出各种复杂的应用程序。用户可以将主要精力集中于应用业务逻辑的实现，而不是底层通信、异常处理等方面。

以 C 语言为例，一个 MQ 应用的开发流程如下：

```
MQCONN()/*建立与队列管理器的连接*/
MQOPEN()  /*打开要进行读写操作的队列*/
MQPUT()  /*将消息放入队列*/
```

```
MQGET()  /*从队列中读取消息*/
MQINQ()  /*查询队列的属性*/
MQSET()  /*设置队列的属性*/
MQCLOSE()  /*在读写等操作进行完之后，将队列关闭*/
MQDISC()  /*断开与队列管理器的连接，释放相关的资源*/
```

14.4　小结

这一章介绍了消息中间件的功能，常用的消息传递模型，并且介绍了 IBM 消息中间件 WebSphere MQ。后面章节将介绍 Java 消息服务（JMS）和 JavaMail。

第 15 章　JMS 应用开发

15.1　JMS 概述

15.1.1　什么是消息

消息是一种软件组件或应用之间的通信方法。消息系统是一种对等（peer-to-peer）的系统，消息客户可以向其他客户发送消息，也可以接收来自其他客户的消息。每一个客户和一个消息代理相连，由消息代理提供创建、发送、接收、读取消息的服务。

消息实现了松散耦合的分布式通信。组件发送消息到目的地（destination），消息接收者从该目的地提取消息。但是，消息的发送和接收却不是同时进行的。实际上，发送者不必去了解接收者，同样接收者也不必了解发送者；它们只需要知道消息格式和消息目的。因此，消息不同于远程方法调用（RMI）这种紧密耦合的通信方法，因为 RMI 要求知道远程应用的方法。

消息也不同于电子邮件（e-Mail），电子邮件是人与人或应用程序与人之间的通信方法。消息却只应用于应用程序和组件之间的通信。

15.1.2　什么是 JMS API

Java 消息服务（Java Message Service，JMS）是一组 Java 应用程序接口(Java API)，它提供创建、发送、接收、读取消息的服务。由 Sun 公司和合作伙伴设计的 JMS API 定义了一组公共的应用程序接口和相应语法，使得 Java 程序能够和其他消息组件进行通信。

使用 JMS API，减少了程序员需要学习的概念，但却提供了足够的功能，支持复杂的消息应用。在同一个 JMS 消息域中，它提高了消息应用程序在不同 JMS 提供者之间的可移植性。

使用 JMS API 可以实现松散耦合的通信，而且它还有如下特点：

（1）异步。JMS 提供者将到达的消息发送给客户，客户不用发送请求消息。

（2）可靠。JMS API 确保消息传送一次而且只传送一次。可靠性差的应用程序可能会丢失消息或者重复接收消息。

15.1.3　什么时候应用 JMS API

企业级的应用供应商在以下情况下可能会选择消息 API,而不是选择紧密耦合的 API（例如 RPC）：

（1）供应商希望组件不依赖其他组件的接口信息，这样，组件变得容易替换。

（2）供应商希望不论所有的组件是否都同时正常运行，也能运行应用。

（3）应用的商业逻辑要求组件发送消息给其他组件，并且在没有及时回复的情况下也能进行操作。

例如，一个汽车制造商的企业应用组件可以在以下情况下使用 JMS API：

（1）当库存水平低于正常水平时，库存（Inventory）组件将消息发送给制造（Factory）组件。

（2）制造（Factory）组件发送消息给零件（Parts）组件，使得它可以收集所需的零件。

（3）零件（Parts）组件将消息发送给库存（Inventory）组件和订购（Parts Order）组件，要求更新库存，从供应部订购新零件。

（4）制造（Factory）和零件（Parts）组件都能发送消息给会计（Accounting）组件，要求更新它们的预算。

（5）商务部发送产品目录给销售部。

使用消息完成以上任务，允许不同组件之间高效联系，用不着安排上级部门管理控制和使用其他资源。

图 15-1 显示了这个简单的例子如何工作。

图 15-1　企业应用中的消息

制造业只是企业如何应用 JMS API 的一个例子。像零售业、金融服务业、公共医疗卫生服务等其他行业也有消息应用。

15.1.4　JMS API 如何同 J2EE 平台工作

1998 年，JMS API 第一次发布时，它的最重要的目标是使 Java 应用程序能够访问现有的面向消息中间件，如 IBM 的 MQSeries。从那时起，许多厂家开始选择实现 JMS API,因此，JMS 产品才能够为企业提供更完整的消息性能。

在 J2EE 1.2 中，要求基于 J2EE 的服务供应商支持 JMS API，但不要求完全实现它。现在，在 J2EE 1.3 中，JMS API 是平台的一部分。应用程序开发者可以使用 J2EE 组件的消息功能。

J2EE 1.3 中的 JMS API 具有以下特征：

（1）应用客户端。EJB 组件、Web 组件可以发送或同步接收 JMS 消息。应用客户端也可以异步接收 JMS 消息（Applets 不支持 JMS API）。

（2）新引入的消息 bean，能够异步处理消息。JMS 提供者(JMS provider)可以使用消息 bean，实现消息的并发处理。

（3）可以在分布式事务中发送、接收消息。

JMS API 的引入，简化了商业逻辑的开发，允许松散耦合、可靠的、异步通信的 J2EE 组件和老式的消息系统之间进行通信，增强了 J2EE 平台的功能。开发者很容易把新的特性加入现有的 J2EE 应用当中。

J2EE 平台的 EJB 容器，支持分布式事务并且能够处理并发的消息，增强了 JMS API 的功能。

J2EE 1.3 还包括另一种新引入的技术——J2EE 连接器，它将 J2EE 应用和现有的企业信息系统（EIS,Enterprise Information System）高度集合起来。而 JMS API 则相反，它使得 J2EE 应用和现有的企业信息系统变得松散耦合。

15.1.5 JMS Provider（JMS 提供者）

对于一个消息应用来说，JMS 提供者是实现 JMS 最核心的部分。理想状态下，JMS 提供者使用纯 Java 编写后，以至于它在 applet 中运行都没问题，而且安装简单，可以跨平台使用。JMS 一个重要的目标就是减少 JMS 提供者所需要做的工作。

15.1.6 JMS Messages（JMS 消息）

JMS 定义了一套消息接口。客户端所使用的消息都是 JMS 提供者提供的。JMS 的一个主要目标就是客户端不依赖于 JMS 提供者，使用自己的 API 来创建和使用消息。

15.1.7 Portability（移植性）

在新的 JMS 规范中一个最主要的移植性就是，应用可以方便地在相同的消息域中跨产品使用。

15.1.8 JMS 不提供的功能

JMS 不提供如下功能：

（1）负载平衡/容错（Load Balancing/Fault Tolerance）。

（2）错误通知（Error/Advisory Notification）。

（3）管理（Administration）。

（4）安全（Security）。

（5）线路协议（Wire Protocol）。

（6）消息类型仓库（Message Type Repository）。

15.1.9 与其他 Java API 的关系

本节讲述 JMS 与其他 API 的关系，如下：

（1）与 JDBC API 的关系。JMS 客户端可以使用 JDBC API。在大多数情况中，它们可以被 EJB（Enterprise JavaBeans）使用，同时也可以被 JTA（Java Transaction API）使用。

（2）与 JavaBeans 控件的关系。JavaBeans 控件可以使用 JMS 会话来发送/接收消息。但 JMS 自己所定义的接口不能直接使用 JavaBeans 控件。

（3）与 EJB 的关系。JMS API 是 EJB 开发人员最重要的资源。它可以和其他的资源（如 JDBC）很好地连接起来。EJB 2.0 规范定义了一种消息驱动 bean（Message-driven Bean），它表示当一个 JMS 客户端发送消息时，bean 可以被异步调用。

（4）与 Java Transaction API（JTA）的关系。一个 JMS 客户端可以使用 JTA 来定义分布式事务；这只是客户端运行所在的事务环境的一个功能，它不是 JMS 的特性。JMS 提供者通过 JTA 可以有选择性地支持分布式事务。

（5）与 Java Transaction Service（JTS）的关系。JMS 可以和 JTS 联合使用在分布式事务中。

（6）与 Java Naming and Directory Interface（JNDI）API 的关系。JMS 客户端通过 JNDI

查找 JMS 对象。JMS 管理员使用提供者指定的工具来创建和配置这些对象。

15.2　JMS 体系结构

本节讨论基于消息的应用程序以及 JMS 所起的作用。

15.2.1　JMS 应用的组成

一个 JMS 应用程序包括如下部分：

（1）JMS 客户端：用来发送/接收消息的 Java 程序。

（2）非 JMS 客户端：用消息系统本地 API 代替 JMS 的客户端。

（3）消息：为了方便客户端间交换数据，每个应用所定义的消息集合。

（4）JMS 提供者：除了消息产品所需要的其他管理和控制功能外，实现了 JMS 规范的消息系统。

（5）管理对象：管理员创建的预配置 JMS 对象，可供客户端使用。

15.2.2　管理

JMS 规范中提供两种管理对象。

（1）ConnectionFactory：客户端使用 ConnectionFactory 对象，创建与 JMS 提供者之间的连接。

（2）Destination：客户端使用 Destination 对象，用来指定消息发送的目的地或者得到所接收消息的发送源头。

JMS 管理对象需要由管理员放入 JNDI 命名空间中。

图 15-2 表示 JMS 管理对象是如何工作的。

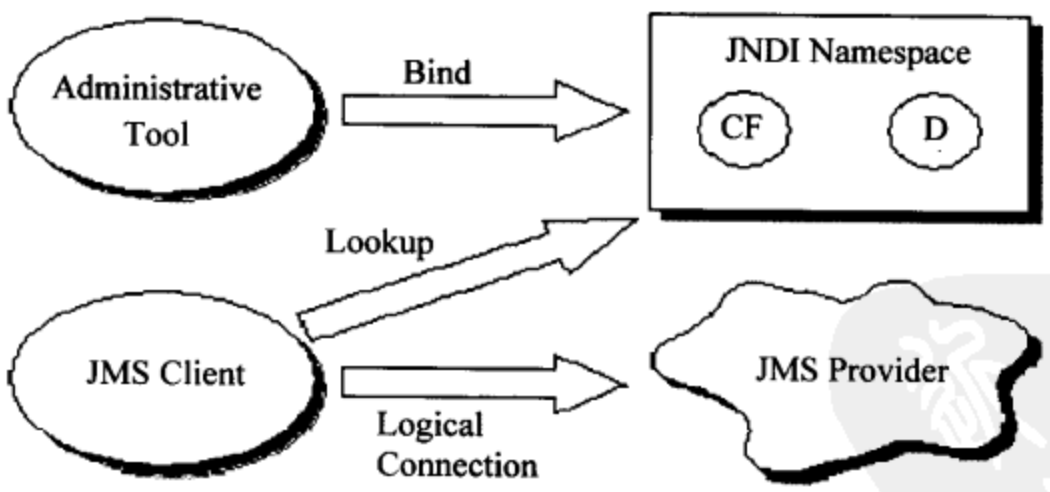


图 15-2　JMS 管理对象

15.2.3　两种消息模式

JMS 规范提供两种最普遍的消息模式：发布/订阅（Publish-Subscribe，Pub/Sub）和点对点（Point-To-Point，PTP）。

发布/订阅模式是一种一对多的发布模式。此模式中，客户端应用向 topics（主题）发布消

息，反过来，对此主题感兴趣的客户订阅 topics。所有订阅客户将收到每一份消息（遵从一定的服务质量、连接和选择）。

点对点模式提供传统的排队机制。此模式中，客户端应用通过一个队列发送消息到一个顺序获得消息的接收客户端。一个 JMS 消息队列是表示消息发送者目标及消息接收者数据源的管理对象。

JMS 规范允许客户端应用使用两种模式的混合方式，发布/订阅和点对点消息均使用 JMS 消息的相同格式。

15.2.4 JMS 接口

JMS 在基于通用消息概念的基础上，为两种消息模式都定义了相应的接口，如表 15-1 所示。

表 15-1 PTP 与 Pub/Sub 接口的对比

JMS 公共接口	PTP 接口	Pub/Sub 接口
ConnectionFactory	QueueConnectionFactory	TopicConnectionFactory
Connection	QueueConnection	TopicConnection
Destination	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver、QueueBrowser	TopicSubscriber

ConnectionFactory：客户端用来创建连接的管理工具。

Connection：表示到 JMS 提供者的连接。

Destination：用来封装消息目的地标识符的管理对象。

Session：发送/接收消息中的单线程上下文。

MessageProducer：由 Session 创建用来发送消息到目的地的对象。

MessageConsumer：由 Session 创建用来从消息目的地接收消息的对象。

JMS 对象关系如图 15-3 所示。

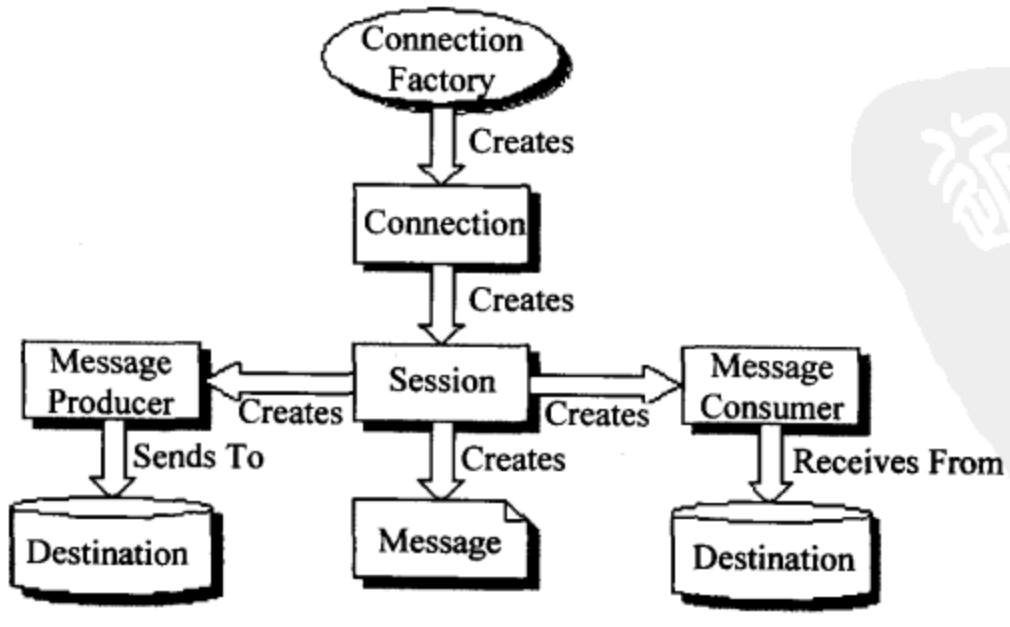


图 15-3 JMS 对象关系图

15.2.5　开发 JMS 应用

广义上讲，JMS 应用程序就是一个或多个进行消息交换的 JMS 客户端。

开发 JMS 客户端通常有如下步骤：

（1）通过 JNDI 查找 ConnectionFactory 对象。

（2）通过 JNDI 查找一个或多个 Destination 对象。

（3）ConnectionFactory 根据需要传递的消息来创建一个 JMS Connection 对象。

（4）使用 JMS Connection 对象来创建一个或多个 JMS Session 对象。

（5）使用 JMS Connection 对象和 Destination 对象来创建所需的 MessageProducer 对象和 MessageConsumer 对象。

（6）告诉 JMS Connection 对象可以开始传递消息。

15.2.6　安全性

JMS 规范不提供控制或配置消息完整性或消息机密性的功能。

15.2.7　多线程

JMS 可以要求它所有的对象都支持并发访问，虽然支持并发访问将明显增加系统消耗和复杂程度。

表 15-2 可以指明 JMS 对象是否支持并发使用。

表 15-2　JMS 对象对并发使用的支持

JMS 对象	是否支持并发使用
Destination	是
ConnectionFactory	是
Connection	是
Session	否
MessageProducer	否
MessageConsumer	否

有两个限制并发访问 Session 的原因。第一，Session 是一种支持事务的 JMS 实体，它实现多线程事务是非常困难的。第二，Session 支持异步消息消费。JMS 不要求用来处理异步消息消费的客户端代码有处理多条并发消息的能力，这是一点重要的因素。另外，如果 Session 被设置成多条异步消费者，那么，客户端将不会强制处理那些单独消费者并发执行的情况。这些约束将使 JMS 能更简单地用于典型客户端。

15.2.8　客户端触发机制

在基于消息的应用中，一些客户端常常被设计成周期性唤醒并等待消息处理结束。一个基于消息的应用触发机制通常是使用这种客户端触发机制。

JMS 不提供触发客户端执行这种机制。一些提供者可以通过使用管理工具提供这种触发

机制。

15.2.9 请求/答复（Request/Reply）

JMS 提供 JMSReplyTo 头域（header field）来指定将要回复的消息的 Destination 属性。回复消息中的 JMSCorrelationID 头域用来引用相关原始请求（Request）。另外，JMS 提供了一个工具，为每个可能用来作为回复的惟一目的地创建临时队列和主题。

企业消息应用支持多种 Request/Reply 风格，包括“一个消息请求产生一个消息回复”到“一个消息请求产生一个消息回复流”。

为了方便，JMS 为 Pub/Sub 和 PTP 定义了 request/reply 帮助类来实现一个简单的 request/reply 表单。JMS 提供者和客户端都有专门的实现。

15.3 JMS 消息模型

企业消息产品把消息看作由消息头和消息体组成的轻量级实体。消息头包含消息的路由和认证信息；消息体包含可以发送的应用程序数据。

15.3.1 目标

JMS 消息模型实现了如下目标：

（1）提供一个单一的、统一的消息 API。

（2）提供一套 API 用来匹配已存在的、非 JMS 程序所创建的消息。

（3）支持跨操作系统、不同体系结构、跨计算机语言的不同种类的应用开发。

（4）支持包含 Java 对象的消息。

（5）支持包含 XML 的消息。

15.3.2 JMS 消息组成

JMS 消息由两部分构成：header（消息头）和 body（消息体）。header 包含消息的识别信息和路由信息，body 包含消息的实际数据。

（1）Header：所有的消息都支持统一的 header 域。header 包含消息的识别信息和路由信息。

（2）Properties：标准 header 域的补充，消息提供一种内建的工具，可以用来为消息添加可选的 JMS 消息头。

- 应用程序指定属性：它提供一种为消息添加应用程序指定属性的机制；
- 标准属性：JMS 在有效的、可选的头域中定义了一些标准属性；
- 提供者指定属性：把提供者本地客户端所需要的提供者指定属性结合到 JMS 客户端中。

（3）Body：JMS 定义了一些消息类型主体，用来覆盖当前所使用的主要消息风格。

15.3.3 消息头（header fields）

一个完整的消息头将会传送给所有接收消息的 JMS 客户端，但不会传送给非 JMS 客户端。下面讲述各个 JMS 消息头域。

（1）JMSDestination：JMSDestination 头域包含消息将要发送到的目的地。在投递消息的

时候，该域被忽略；当投递完成后，投递方法使用它来指定目的地。

（2）JMSDeliveryMode：在投递消息的时候，JMSDeliveryMode 头域包含消息的指定投递模式。在投递消息的时候，该域被忽略；当投递完成后，投递方法使用它来指定投递模式。

（3）JMSMessageID：JMSMessageID 头域为每一个通过提供者投递的消息保留一个惟一识别码。在投递消息的时候，该域被忽略；当投递完成后，它包含提供者指定的值。

（4）JMSTimestamp：JMSTimestamp 头域为每个消息包含提供者发送的时间。事实上，它并不是一个确切时间，而是有一点的延迟。在投递消息的时候，该域被忽略；当投递完成后，它包含发送方法从调用到返回的时间差。

（5）JMSCorrelationID：客户端可以利用 JMSCorrelationID 头域将消息关联起来。一个典型应用就是将一个答复消息和它的请求消息关联起来。JMSCorrelationID 可以包含特定提供者的消息 ID（provider-specific message ID）、特定应用程序的字符串（application-specific String）以及本地提供者字节值（provider-native byte[] value）。每一个被提供者发送的消息都会赋予一个消息 ID 号，以便于消息间通过该 ID 相互联系。所有的消息 ID 号都必须用 ID 做前缀。

（6）JMSReplyTo：在发送消息的时候，JMSReplyTo 包含一个由客户端提供的 Destination 对象。

（7）JMSRedelivered：在通常情况下，提供者必须为每个再次投递的消息设置 JMSRedelivered 头域。如果把它设置成真，则表示该消息在过去已经被投递，应用程序要更加注意，防止重复消息投递。

（8）JMSType：在发送消息的时候，JMSType 包含由客户端提供的消息类型标志。

（9）JMSExpiration：当消息被发送的时候，JMSExpiration 头域包含由发送方法与当前 GMT 时间所计算出来的失效时间。

（10）JMSPriority：JMSPriority 包含消息的优先级。JMS 规范定义了 10 个优先级，从 0 到 9，0 为最低，9 为最高。

表 15-3 说明 JMS 头域如何设置。

表 15-3　JMS 消息头的设置

消息头	由谁设置
JMSDestination	send 或 publish 方法
JMSDeliveryMode	send 或 publish 方法
JMSExpiration	send 或 publish 方法
JMSPriority	send 或 publish 方法
JMSMessageID	send 或 publish 方法
JMSTimestamp	send 或 publish 方法
JMSCorrelationID	客户端
JMSReplyTo	客户端
JMSType	客户端
JMSRedelivered	提供者（Provider）

15.3.4 消息属性（Message Properties）

可以将它看为附加的消息头，用来支持 JMS 消息内建工具需要的属性值。它为消息提供了一种附加、可选的消息头机制。

属性取值用于消息选择机制，同时附带应用或其他消息基础结构所需的任意数据。属性名必须遵循消息选择标识符规则。属性值可以是 boolean、byte、short、int、long、float、double、String 或 Object。消息取值设置先于消息发送。当一个客户端收到一个消息，其属性为只读模式。JMS 规范属性名前缀是“JMSX”。其属性支持是可选的——并非所有 JMS 提供者均应用其属性的使用。

表 15-4 显示 JMS 规范中定义的属性。

表 15-4 JMS 规范中定义的属性

属性	类型	由谁设置	Use
JMSXUserID	String	发送方提供者	发送消息的 User ID
JMSXAppID	String	发送方提供者	发送消息的应用程序 ID
JMSXDeliveryCount	int	接收方提供者	试图传递的消息数量
JMSXGroupID	String	客户端	消息组 ID
JMSXGroupSeq	String	客户端	消息组序号
JMSXProducerTXID	String	发送方提供者	生成消息的事务 ID
JMSXConsumerTXID	String	接收方提供者	消费消息的事务 ID
JMSXRcvTimestamp	long	接收方提供者	JMS 传递消息给消费者的时间戳
JMSXState	int	提供者	消息状态

15.3.5 消息选择

很多消息应用系统都支持消息过滤和分类功能。

（1）消息选择器。JMS 消息选择器允许客户端通过指定头域的方式来选择想要的消息，只有在头域和属性都满足的条件下，消息才被投递。

（2）消息选择器语法。消息选择器语法遵从 SQL 92 标准。

（3）访问已发送消息。在发送了一个消息后，客户端可以在不影响已发送消息的前提下，保留或修改它。

（4）修改接收的消息。当消息被接收后，它的消息头可以修改，但是它的属性值和消息体是只读的。

15.3.6 JMS 消息体

JMS 规范定义了 5 种消息类型，均来自 Message 接口，此接口也定义了消息头及所有 JMS 消息使用的 acknowledge()方法。

- ByteMessage：未解析字节流。
- MapMessage：名称/取值对。这里名称为一字符串，取值为 Java 基本类型。可通过名

称或顺序访问其中的每一个名称/取值对。

- TextMessage：包含一个 java.util.StringBuffer 的消息。
- StreamMessage：Java 基本类型流。可顺序读取以及写入。
- ObjectMessage：包含一个实现了 Serializable 接口的 Java 对象的消息。

（1）清除消息体。Message 接口的 clearBody()方法用来将消息体的值重置为空。

（2）只读消息体。当消息被接收后，它的消息体是只读的。如果试图修改该消息体，就抛出 MessageNotWritableException 异常。

15.4　JMS 消息工具

PTP 和 Pub/Sub 消息模式共享一套 JMS 消息工具。

15.4.1　管理对象（Administered Objects）

JMS 管理对象是一种由 JMS 管理员创建的对象，该对象包含了 JMS 配置信息，可以被客户端使用，并用于管理企业的 JMS 应用程序。

虽然，JMS 管理对象接口（interface for administered objects）不是明确依赖于 JNDI，但是当客户端在命名空间中查找该接口的时候需要使用 JNDI。

管理员可以把管理对象放置于命名空间中任意位置。JMS 规范没有定义命名规范。

JMS 规范定义了两个管理对象：Destination 和 ConnectionFactory。JMS 规范要求管理对象实现 javax.naming.Referenceable 和 java.io.Serializable 接口，以便管理对象可以保存在 JNDI 命名上下文中（JNDI naming context）。

（1）Destination。JMS 规范没有定义标准的地址格式，它定义了 Destination 对象用来保存特定提供者的地址。

（2）ConnectionFactory。ConnectionFactory 对象封装了一系列由管理员定义的连接配置参数。客户端使用它来创建一个到 JMS 提供者的 Connection 对象。ConnectionFactory 对象支持并发访问。

15.4.2　Connection

Connection 对象表示客户端连接到 JMS 提供者的活动会话。它在 JVM 外部分配提供者资源。Connection 对象支持并发访问。

（1）认证。当创建一个连接的时候，客户端可以使用 name/password 的格式来认证。如果不指定，将使用当前会话的认证连接。

（2）客户端标识符。为客户端分配标识符的首选办法是，在特定客户端的 ConnectionFactory 对象中配置，并显式地赋给它所创建的连接。还有一种办法，使用特定提供者的值来设置连接的客户端标识符。

（3）Connection 安装（Connection Setup）。一个 JMS 客户端可以创建一个 Connection，一个或多个 Session，以及大量的 MessageProducer（消息生产者）和 MessageConsumer（消息消费者）。当 Connection 被创建的时候，它是 stop 状态，没有任何消息可以发送给它，这个状态要维持到安装（setup）结束。这个时候调用 Connection 的 start()方法，同时，消息抵达

Connection 的消费者。Connection 可以马上启动，它的安装可以后来再完成。

在 Connection 停止的时候，MessageProducer 可以发送消息。

值得注意的一点是，当 Connection 没有安装好的时候，客户端是不能发送消息的。

（4）暂停消息的投递。可以使用 stop()方法暂停投递一个连接的进入消息；如果需要恢复，可以使用 start()方法。当连接停止后，该连接的所有 MessageProducer 的投递将会暂停，同步接收将会阻塞，消息将不会发送到 MessageListener 处。

停止一个连接，将不会影响到它发送消息的能力。停止一个已经停止了的连接或者开始一个已经开始了的连接，对该连接来说是无影响的。Stop()方法必须在消息投递暂停后才能返回。如果 MessageListener 正在运行，调用 stop()方法，那么 stop()方法必须等待 MessageListener 返回后才能返回。

（5）关闭连接。由于在连接的过程中，提供者通常要分配 JVM 外部的重要资源，所以，客户端必须在不使用的时候关闭这些资源。值得注意的是，当消息消费者在处理它最后一条消息的过程中，试图使用一个关闭了连接的工具时，很有可能会得到异常错误。开发人员在写消息消费者的时候，必须把“最后一条消息”（last message）的情况考虑进去。

如果在一个或多个连接会话的消息监听器在处理消息的同时，要求关闭连接，那么所有该连接的工具和会话都必须保持可用，直到消息监听器把控制权交还给 JMS 提供者。

只有当消息按照一定顺序处理完毕后，才能关闭连接。这意味着，所有正在运行的消息监听者已经返回，所有没有完成的接收已经返回。

关闭一个连接必须在交易会话中回滚正在进行的事务。关闭一个连接不能强迫客户端接收会话进行接收。在一个已关闭连接的会话中调用 acknowledge()方法将会抛出 IllegalStateException 异常。一旦连接已经关闭，调用该连接，或者是连接中的会话，或者是消息消费者，或者是消息消费者，都会抛出 IllegalStateException 异常，调用 close()方法除外。

（6）会话（Sessions）。Connection 是 Session 的 factory（工厂），这些 Session 使用 JMS 提供者的连接以生产和消费消息。

（7）ConnectionMetaData。Connection 提供 ConnectionMetaData 对象。它提供 JMS 的最新版本，以及提供者的产品名称和版本。

（8）ExceptionListener。如果 JMS 提供者在连接中发现了异常，它会通知该连接中已注册的 ExceptionListener。JMS 提供者可以使用 getExceptionListener()方法来得到当前连接已注册的 ExceptionListener，如果当前没有任何注册的 ExceptionListener，将返回 null。

15.4.3 会话（Session）

JMS 会话（Session）是生产和消费消息的单线程上下文。虽然它在 JVM 外部分配资源，但还认为它是 JMS 的轻量级对象。

会话（Session）对象主要提供以下服务：

- MessageProducer 和 MessageConsumer 的 factory（工厂）。
- TemporaryTopic 和 TemporaryQueues 的 factory（工厂）。
- 提供了创建 Queue 或 Topic 对象的途径。
- 提供了提供者优化的消息工厂。
- 支持将跨会话生产者和消费者的操作组合为原子性事务。

- 定义了消息的顺序。
- 保留消费的消息，直到这些消息确认已经接收。
- 有顺序地执行 MessageListener 的注册。
- QueueBrowser 的 factory（工厂）。

1. 关闭会话（Closing a Session）

因为提供者会给会话分配 JVM 外部的资源，客户端必须在不使用会话（Session）的时候关闭它。垃圾收集器可能不会及时回收资源。这一点，对于 MessageProducer 和 MessageConsumer 来说是一样的。

在关闭会话的时候，它会等到消息处理按照一定次序结束后才返回。这意味着，当前已没有任何消息监听器在运行了，如果还存在未接收的消息，它将返回 null 或一条消息。

当关闭会话的时候，没有必要关闭它的消息生产者和消息消费者，会话的关闭只是给 JMS 提供者的一个信号，告诉提供者可以释放该会话的资源。

关闭一个会话必须在交易会话中回滚正在进行的事务。关闭一个连接不能强迫客户端接收会话进行接收。一旦关闭会话，调用任何该会话或该会话的消息消费者和消息生产者都会抛出 IllegalStateException 异常，调用 close()方法除外。

2. 创建临时 Destination

虽然会话可用来创建临时 Destination，但这只是为了方便。会话的实际活动范围应该是整个连接。

临时 Destination（包括 TemporaryQueue、TemporaryTopic 对象）是系统特别为它们所在的连接创建的目的地。只有作为连接的所有者才能为它们创建消息消费者。

一个典型的应用是把它作为 JMSReplyTo 的目的地址。

每一个 TemporaryQueue 或 TemporaryTopic 都是惟一的，不能复制。

虽然临时 Destination 可以在 JVM 外部分配资源，但是如果它们不再需要将会被删除。当它们被垃圾收集或所在连接被关闭后，它们将被自动删除。

3. 创建 Destination 对象

对于大多数使用 Destination 对象的客户端来说，最方便的方法是通过 JNDI 查找。

对于一些特殊客户端，需要根据提供者指定地址来动态创建 Destination 对象， Session 提供了特定 JMS 提供者的方法来完成该需要。

4. 优化消息实现

一个会话可以使用提供者优化实现的方法来创建消息，这样提供者可以减小处理消息的开销。

5. 使用会话的约定

会话被设计成一个线程在一段时间内串行使用。

一个最典型的用法就是在同步 MessageConsumer 中使用线程堵塞，直到消息的到来。该线程可以使用一个或多个会话的 MessageProducers。

在同一个会话中，当前已经有一个控制线程在等待接收消息，如果一个客户端试图使用控制线程同步接收消息，这是错误的。

另一个典型的应用是使用一个单一线程来组成会话，由会话来创建一个生产者和一个或多个异步消费者。在这里，消息生产者将被消费者的消息监听器单独使用。虽然会话串行执

行消费者的 MessageListeners，但是它们可以安全地共享该会话的资源。

6. 事务（Transactions）

会话可以被指定为 transacted。每一个设置为 transacted 的会话都支持事务。每一个事务都把一系列产生的消息和一系列消费的消息放入一个原子工作单元中。事务组织一个会话的输入消息流和输出消息流到一系列原子单元中。当一个事务被提交，它的原子输入单元被确认，它相关的原子输出单元被发送。当一个事务回滚，那么它生产的消息被取消，它消费的消息被自动恢复。

当一个事务所在的会话使用 commit()或者 rollback()方法的时候，就表示该事务完成了。当前会话的事务完成后将自动开始下一个事务。这就意味着，当前交易会话始终都有一个事务。

7. 分布式事务（Distributed Transactions）

JMS 不要求提供者支持分布式事务，如果需要支持的时候，将通过 JTA XAResource API 来提供。

虽然 JMS 客户端可以直接处理分布式事务，但是 JMS 规范建议尽量避免这么做。

8. 多会话（Multiple Sessions）

客户端可以创建多个会话，每个会话对于消息的生产者和消费者都是独立的。

对于 Pub/Sub 模式来说，如果会话的两个 TopicSubscriber 都订阅了同样的 Topic，每一个订阅者都将得到每一条消息，投递给一个订阅者将不会影响到后面的订阅者。

对于 PTP 模式来说，JMS 规范没有定义并发 QueueReceivers 使用相同 Queue 的具体语法，但是，JMS 规范并不阻止 JMS 提供者对此的支持，也就是说，消息能否投递到多个 QueueReceivers 上，关键在于 JMS 提供者是如何实现的。

9. 消息序列

JMS 客户端不必理解什么时候可以依赖于消息序列，什么时候不可以。

（1）消息接收顺序。消息按照会话定义的次序进行消费。这个次序对于消息的确认是相当重要的。JMS 规范定义了会话发送消息的目的地按照消息发出的次序接收消息。这对会话的输入消息流进行了一定次序的约束。JMS 规范没有定义跨目的地或从多个会话发送消息到目的地这种情况下的消息接收顺序。

（2）消息发送顺序。

- 优先级高的消息先于优先级低的消息发送。
- 如果 JMS 提供者失败，客户端将不能接收 NON_PERSISTENT 模式的消息。
- 如果 NON_PERSISTENT 和 PERSISTENT 模式的消息被发送到一个客户端，发送次序将由投递模式来保证。也就是说，NON_PERSISTENT 消息可能会比 PERSISTENT 消息早到。在同样优先级的情况下，PERSISTENT 消息不可能比 NON_PERSISTENT 消息早到。

10. 消息确认

JMS 支持消息接收的规范确认，与低层通信协议中的握手类似，当一个客户端成功接收到一个消息时，确认将通知消息代理。

确认模式由会话层控制，支持以下各种确认模式。

- AUTO_ACKNOWLEDGE：会话自动确认每个消息的接收，在同步模式中，这表明调用 receive()成功返回。

- CLIENT_ACKNOWLEDGE：允许客户端应用指明其成功接收了消息，但可能要延迟应答。应用必须对每一成功接收的消息调用 acknowledge()方法。
- DUPS_OK_ACKNOWLEDGE：一个 AUTO_ACKNOWLEDGE 的变体，提供一种“懒惰”机制，导致失败状态时发送重复消息。此模式只适用于不敏感数据，它可以提高消息系统的效率。

11. 重复消息投递

JMS 提供者不能重复投递一个已确认的消息的副本。

12. 重复消息生产

JMS 提供者不能生产重复的消息。这意味着，生产消息的客户端可以依靠 JMS 提供者来保证消息的消费者只能进行一次接收。

13. 并发消息投递

客户端希望使用多会话来并发投递消息。事实上，每一个会话的监听器线程都是并发运行的，也就是说，当一个会话的监听器在运行的时候，另一个会话的监听器也同时在运行。

15.4.4 MessageConsumer

客户端使用 MessageConsumer 从目的地接收消息。把 Queue 或 Topic 传递给一个会话的 createConsumer()方法，可以创建一个 MessageConsumer 对象。

（1）同步投递。客户端可以请求来自同一个 receive 方法的 MessageConsumer 的消息。

（2）异步投递。以异步方式接收消息，需要创建一个消息监听器，然后注册一个或多个使用 MessageConsumer 的 JMS MessageListener 接口。会话（主题或队列）负责产生某些消息，这些消息被传送给使用 onMessage 方法的监听器。

15.4.5 MessageProducer

客户端使用 MessageProducer 来发送到消息目的地。把 Queue 或 Topic 传递给一个会话的 createProducer()方法，可以创建 MessageProducer。

15.4.6 消息发送模式

JMS 支持两种消息发送模式。

（1）NON_PERSISTENT。这是最有效的发送模式，因为它不需要将消息注册到持久化存储。JMS 规范指出一个 JMS 提供者必须以一次性方式发送一个 NON_PERSISTENT 消息，意即代理可以丢失消息，但绝对不能两次发送同一个消息。

（2）PERSISTENT。此模式通知代理将消息放入一个数据源，作为客户端发送操作的扩展，这样确保消息在其他系统失败的情况下仍可存活。JMS 规范指出，一个 JMS 提供者必须以一次并且只有一次的方式发送一个 PERSISTENT 消息，不可以丢失，也不可以发送两次。

15.4.7 消息生存时间

生存时间参数是在确认所有预订者接收到消息之前、消息代理保存消息的时间。如果在初始发送之后，任何订阅者均未确认发送，此消息将维持到生存时间期限，直到订阅者连接至消息代理并接收发送。

如果生存时间为 0，消息将永不终止。当一个消息的生存时间到后，代理会丢弃它，JMS 规范并未定义任何形式的消息终止通知。客户端也不应该接收已终止的消息。然而，JMS 规范并未确保消息代理不允许此类事情的发生。通常，一个设置为永远存活的消息一旦被发送到所有当前预订者后，就会被丢弃。

15.4.8 异常

JMSException 是所有 JMS 异常的父类。

15.5 JMS Point-to-Point 模型

JMS 支持两种基本的消息传递机制。第一种机制是点到点的消息传递（point-to-point messaging），在这种机制下，消息由一个发布者（发送方）发送，由订阅者（接收方）接收。另一种机制是发布—订阅式的消息传递（publish-subscribe messaging），在这种机制下，消息由一个或多个发布者发送，由一个或多个订阅者接收。下面介绍 JMS Point-to-Point 模式。

（1）Queue 管理。JMS 规范没有定义这样一种机制，用来创建、管理以及删除长时间存在的消息队列。

（2）Queue。Queue 对象封装了提供者指定队列名，是客户端用来向 JMS 方法指明队列标识的途径。

（3）TemporaryQueue。TemporaryQueue 对象是为持续期间的 Connection 或 QueueConnection 对象创建的惟一 Queue 对象。它是一种只能由创建它的 Connection 或 QueueConnection 对象来消费的系统定义队列。

（4）QueueConnectionFactory。客户端使用 QueueConnectionFactory 来创建 QueueConnection，连接到 JMS PTP 提供者。

（5）QueueConnection。一个 QueueConnection 就是一个连接到当前 JMS PTP 提供者的活动连接。客户端使用 QueueConnection 对象创建一个或多个 QueueSession 来产生或消费消息。

（6）QueueSession。QueueSession 对象提供了创建 QueueReceiver、QueueSender、QueueBrowser 和 TemporaryQueue 对象的方法。当一个 QueueSession 终止时正好有消息被接收但是没有被确认，那么该消息将会保留，直到消息消费者下一次访问该队列时将被重新发送。

（7）QueueReceiver。客户端使用 QueueReceiver 从已经发送的消息队列中接收消息。

对于同一个队列，QueueReceiver 可以有两个会话，但是 JMS 规范并没有定义如何在 QueueReceiver 之间发送消息。

如果 QueueReceiver 指定了消息选择器，那么没有选择的消息仍将保留在队列中。消息选择器允许 QueueReceiver 跳过消息，这也就意味着，那些跳过的消息最后才能读取到，读取顺序将不会按照原先消息生产者所定义的顺序；如果 QueueReceiver 没有消息选择器，那么消息的读取顺序将按照消息生产者所定义的顺序。

（8）QueueBrowser。客户端使用 QueueBrowser 从队列中浏览消息，而不用删除消息。QueueBrowser 可以由 Session 或 QueueSession 来创建。

浏览方法将会在扫描队列消息后返回一个 java.util.Enumeration 数据类型。它可能是队列

整个内容的集合，也可能只包含符合消息选择器条件的消息。

（9）QueueRequestor。JMS 提供了 QueueRequestor 帮助类来简化服务请求工作。

QueueRequestor 构造器需要一个 QueueSession 对象和一个目的地队列。它为响应创建了一个 TemporaryQueue 对象，并提供了一个请求方法来发送请求消息，并等待回复。

（10）可靠性（Reliability）。一个队列由管理员创建，并长时间存在。无论消费消息的客户端是否激活，它都可以有效地保持消息并发送它。鉴于这个因素，客户端不用做特别的预防措施来保证不丢失消息。

15.6　JMS Publish/Subscribe 模型

JMS Publish/Subscribe 模型定义了如何向一个内容节点发布和订阅消息，这些节点称作主题（topic）。主题可以看作消息的传输中介，发布者（Publisher）发布消息到主题，订阅者（Subscriber）从主题订阅消息。使用主题作为消息传输中介，消息订阅者和消息发布者可以保持互相独立，不需要接触即可保证消息的传送。

1. Pub/Sub 延迟（Pub/Sub Latency）

虽然在所有的 Pub/Sub 系统中都存在延迟，但是消息的延迟可能会依赖于 JMS 提供者向已存在的订阅者发送消息的速度，还包括消息在传输过程中由 JMS 提供者保留的时间。

2. 持久性订阅（Durable Subscription）

订阅者（subscriber）对象的生命周期就是非持久性订阅的持续时间。这就意味着，客户端只能看见那些当它的订阅者是活动状态时所发布到某个主题的消息。如果订阅者为非活动状态，那么它将得不到发布到相应主题上的消息。

如果把订阅设置成持久性的，那么将花费很高的系统资源。持久性订阅者可以注册持久性订阅，并得到一个惟一标识符。拥有相同标识符的后来的订阅对象将恢复停留在先前订阅对象状态中的订阅。

如果当前没有活动的订阅者在进行持久性订阅，那么 JMS 将保留订阅的消息直到消息被接收或者消息失效为止。

所有的 JMS 提供者都必须能够运行 JMS 应用程序，动态创建或删除持久性订阅。另外，有些 JMS 提供者可以提供用来管理配置持久性订阅的工具。

3. 主题管理（Topic Management）

一些产品需要能够静态地定义主题，以及相关的授权控制列表，但 JMS 规范并没有定义用来创建、管理或删除主题的工具。

TemporaryTopic 是一种特殊类型的主题对象，用来为每个 TopicConnection 创建一个惟一的 Topic。

4. Topic

一个 Topic 对象封装了提供者指定的主题名。它是客户端用来为 JMS 方法指定主题标识符的一种途径。

许多 Pub/Sub 提供者允许以分层方式定义主题，以便主题可以嵌入其他主题层次下。JMS 规范没有限制 Topic 对象的表现形式，它可以是主题层次中的一层，也可以是主题层次中一个较大的部分。

主题层次的组织和订阅的粒度是 Pub/Sub 应用程序架构的一个重要部分，但是 JMS 规范并没有定义怎么去做。

5. TemporaryTopic

一个 TemporaryTopic 对象是为 Connection 或 TopicConnection 运行时创建的一个惟一 Topic 对象。它是一个能且只能由创建它的 Connection 或 TopicConnection 对象来消费的系统定义 Topic 对象。

就像它的定义一样，对一个临时主题创建持久性订阅是没有意义的，这样做，JMS 提供者有可能检测出这个程序错误，也有可能检测不出来。

6. TopicConnectionFactory

客户端使用 TopicConnectionFactory 来创建 TopicConnection，以连接到 JMS Pub/Sub 提供者。

7. TopicConnection

一个 TopicConnection 表示一个连接到 JMS Pub/Sub 提供者的活动连接。客户端使用 TopicConnection 创建一个或多个 TopicSession 来生产和消费消息。

8. TopicSession

TopicSession 提供了用来创建 TopicPublishers、TopicSubscribers 和 TemporaryTopics 的方法。它也提供了 unsubscribe()方法用来删除客户端的持久性订阅。

当一个 TopicSession 终止时，如果有些消息已经接收但是没有确认，那么持久性的 TopicSubscriber 将保留它们并重新发送；而非持久性的 TopicSubscriber 将不需要这么做。

9. TopicPublisher

客户端使用 TopicPublisher 来向一个主题发布消息。TopicPublisher 是 JMS MessageProducer 在 Pub/Sub 模式下的一个变体。也可以使用 MessageProducer 向一个主题发布消息。

10. TopicSubscriber

客户端使用 TopicSubscriber 来接收已经发布到某个主题的消息。TopicSubscriber 是 JMS MessageConsumer 在 Pub/Sub 模式下的一个变体。

通常情况下，TopicSubscriber 是非持久性的。它们仅仅接收那些在它们是活动状态时发布的消息。

在有些情况下，一个连接可以向一个主题订阅和发布消息，但是订阅者的 NoLocal 属性将禁止传递该连接发布的消息。

一个 TopicSession 允许为每个目的地创建多个 TopicSubscriber，它将把消息投递到每个具有接收消息条件的 TopicSubscriber 中去。每一个消息的拷贝将被认为是完全独立的消息。对其中的一个做处理将不会影响到其他的拷贝。比如说，当其他的消息正在等待它们的消费者处理的时候，另一条消息可以马上投递。

11. 持久性的 TopicSubscriber（Durable TopicSubscriber）

如果一个客户端需要接收到所有的消息，包括它的 TopicSubscriber 对象在非活动状态时发布到某一个主题的消息，就要用到持久性的 TopicSubscriber。持久性的 TopicSubscriber 可以由 Session 或 TopicSession 创建。JMS 保存了所有持久性订阅的列表，并保证所有来自 Topic 发布者的消息都保留下来，直到它们被确认或者过期。

持久性订阅者的会话都必须提供相同的客户端标识符。另外，每一个客户端都必须为它的每一个持久性订阅指定一个惟一的标识。在同一段时间内，只能有一个会话能够使用

TopicSubscriber 来进行特殊的持久性订阅。

客户端可以创建一个具有相同名字和新主题和（或）消息选择器，或 NoLocal 属性的 TopicSubscriber，来改变已存在的持久性订阅。改变一个持久性订阅相当于删除或者重建它。

Session 和 TopicSession 提供 unsubscribe()方法来删除由客户端创建的持久性订阅。如果客户端删除一个正拥有活动的 TopicSubscriber 的持久性订阅，或者消息的接收正好是当前事务的一部分，或者该消息在该会话中还没有被确认的时候，将出现错误。

12. 恢复和重新投递

一个非持久性订阅的非确认的消息可以在该非持久性订阅的生命周期内被恢复，当一个非持久性订阅被终止的时候，无论消息是否被确认它都将被抛弃。

只有持久性订阅才能可靠地恢复没有被确认的消息。

使用 PERSISTENT 投递模式投递一个消息到一个主题，将不会改变它的恢复和重新投递模型。为了确保投递，一个 TopicSubscriber 将建立一个持久性订阅。

13. 管理订阅

理想状态下，发布者和订阅者在它们被创建的时候，由提供者动态注册。从客户端的角度来看，这很正常；但从管理者的角度来看，这意味着其他的任务都需要支持发布者和订阅者的创建。

JMS 规范没有定义，该分配多少资源用于消息的存储，资源少于多少会出现溢出。

14. TopicRequestor

JMS 提供了 TopicRequestor 帮助类来简化操作服务请求。

TopicRequestor 的构造器需要一个 TopSession 参数和一个 Topic 参数。它为每一个响应创建一个 TemporaryTopic，并且提供 request()方法用来发送请求消息，并等待它的答复。

15. 可靠性

当所有针对某一个主题的消息必须被接收的时候，将使用一个持久性订阅者。JMS 确保在持久性订阅者为非活动时发布的消息都保存在 JMS 中，并且在以后持久性订阅者变成活动状态时投递这些消息。

非持久性订阅将被使用在可以容忍消息丢失的场合。表 15-5 中列出了不同发布方式的可靠性。

表 15-5　发布/订阅可靠性

发布方式	非持久性订阅	持久性订阅
NON_PERSISTENT	最多一次 （在不活动状态下将丢失）	最多一次
PERSISTENT	有且仅有一次 （在不活动状态下将丢失）	有且仅有一次

15.7　JMS 异常

本节介绍 JMS 异常的种类以及处理方法。

1. JMSException

JMS 定义的 JMSException 是所有 JMS 方法抛出异常的根类。JMSException 是可预期的异常，能够被 catch 捕获，按照处理其他相关异常的方式处理它。JMSException 提供了以下信息：

- 一个提供者指定字符串，用来描述错误信息。这个字符串是标准的 Java 异常消息，可以通过 getMessage() 方法得到。
- 一个提供者指定错误代码。
- 其他异常的相关信息。通常情况下，JMS 异常都是底层问题导致的，在有些情况下，这些底层异常将会连接到 JMS 异常上。

在 JMS 方法的信号中只包含 JMSException。JMS 方法能够抛出属于任何 JMS 提供者指定异常的 JMS 标准异常。

2. 标准异常

除了 JMSException 以外，JMS 规范还定义了一些附加的异常来标准化基本错误情况的描述。只有在 JMS 要求的情况，指定的 JMS 异常才会被抛出。

JMS 规范定义了如下标准异常：

- IllegalStateException：当在不正确的时间请求一个方法，或者提供者在不适当的状态下被请求响应，抛出该异常。
- JMSSecurityException：当提供者拒绝由客户端提供的 username/password 的时候，抛出该异常。如果由于安全限制而阻止了一些方法的完成，也将抛出该异常。
- InvalidClientIDException：当一个客户端试图把一个连接的客户端标识符设置成提供者拒绝的值，会抛出该异常。
- InvalidDestinationException：当一个目的地地址不能被提供者找到，或者不再存在，会抛出该异常。
- InvalidSelectorException：当一个客户端试图给消息选择器赋予错误的值，会抛出该异常。
- MessageEOFException：当一个 StreamMessage 或 BytesMessage 正在被读取的时候，一个非预期的流结束到达，会抛出该异常。
- MessageFormatException：当一个 JMS 客户端试图使用一种不支持的数据类型或者试图使用一种错误的数据类型来从消息中读取数据，会抛出该异常。
- MessageNotReadableException：当一个 JMS 客户端试图读取一个只写的消息时，会抛出该异常。
- MessageNotWritableException：当一个 JMS 客户端试图写一个只读的消息时，会抛出该异常。
- ResourceAllocationException：当一个提供者不能分配给一个方法所请求的资源时，会抛出该异常。
- TransactionInProgressException：由于一个事务正在进行而导致一个操作无效时，会抛出该异常。
- TransactionRolledBackException：当调用 Session.commit 导致当前事务回滚的时候，会抛出该异常。

15.8　JMS 应用程序服务器工具

这部分主要讲述用来订阅消息的并发处理的 JMS 工具，也讲述了 JMS 提供者如何支持 JTS 会话。这些工具主要是由 JMS 提供者使用的。

15.8.1　订阅消息的并发处理

JMS 提供了一个工具，专门用来创建可以并发消费消息的 MessageConsumer。

这个工具把工作分为三个角色：

- JMS 提供者（JMS provider）：它的任务是投递消息。
- 应用程序服务器（Application Server）：它的任务是创建消费者和管理使用并发 MessageListener 对象的线程。
- 应用（Application）：它的任务是定义一个订阅，这个订阅有一个目的地地址、可选的消息选择器和提供一个单线程的 MessageListener 类来消费消息。应用程序服务器将构造多个应用来并发消费消息。

1. Session

Session 为应用程序服务器提供三种可使用的方法。

- setMessageListener()和 getMessageListener()：一个会话的 MessageListener 通过一个 ConnectionConsumer 来消费已经分配给当前会话的消息。
- run()：由于消息是通过 ConnectionConsumer 分配给它的会话，它将被会话的 MessageListener 连续处理，当监听器处理完最后一条消息后返回，run()方法也返回。

一个应用程序服务器将提供一个 MessageListener 类，该类包含了由应用开发人员编写的用来处理消息的单线程代码。它也将由指定消息的监听器告诉目的地和消息选择器。

一个应用服务器在创建 JMS Connection、ConnectionConsumer 以及 Session 来处理消息时要特别小心，因为，它们将按需创建很多 MessageListener 实例，并注册给每个它拥有的会话。

虽然在会话服务中将会使用很多的监听器，但是这些监听器很有可能是它所在的会话将它作为构造器参数使用的。

2. ServerSession

ServerSession 是由应用程序服务器实现的对象。它总是被应用服务器用来和 JMS 会话打交道。

ServerSession 实现了两个方法。

- getSession()：返回当前 ServerSession 的 JMS Session。
- start()：开始 ServerSession 线程的执行，并导致相关 JMS Session 的 run()方法的执行。

3. ServerSessionPool

ServerSessionPool 是由应用程序服务器实现的对象，该对象用来为处理消息的 ConnectionConsumer 创建一个 ServerSession 池（pool）。

它只有一个 getServerSession()方法，这个方法将会从池中删除一个 ServerSession，并且将 ServerSession 传递给 ConnectionConsumer 去消费一条或多条消息。

JMS 规范没有定义如何去实现一个池。它有可能是一个 ServerSession 的静态池，也有可能是使用算法按需动态创建的。

如果 ServerSessionPool 不能再提供更多的 ServerSession，那么 getServerSession()方法将被阻塞。如果一个 ConnectionConsumer 被阻塞，将不能再投递消息，直到有一个 ServerSession 返回。

4. ConnectionConsumer

对于应用服务器，一个连接为创建一个 ConnectionConsumer 提供了专门的工具。将给待消费的消息指定一个目的地和一个消息选择器。另外，给 ConnectionConsumer 指定一个 ServerSessionPool 来处理消息。属性 maxMessages 用来限制一个 ConnectionConsumer 一次可以同时装载消息到一个 ServerSessionPool 的最大数量。

5. ConnectionConsumer 如何使用 ServerSession

由 JMS 提供者实现的 ConnectionConsumer 使用 ServerSession 来处理一个或多个到达的消息。它需要如下几步：

（1）从 ServerSessionPool 中得到一个 ServerSession。

（2）得到 ServerSession 的 Session。

（3）配合一个或多个消息一起装载 Session。

（4）启动 ServerSession 来处理消息。

一个针对 QueueConnection 的 ConnectionConsumer 将把它的消息装载到一个 QueueSession 中；一个针对 TopicConnection 的 ConnectionConsumer 将把它的消息装载到一个 TopicSession 中。

值得注意的是，JMS 规范没有定义如何实现第三步，虽然 ConnectionConsumer 和 Session 都是由相同 JMS 提供者实现的，但是实现装载是采用各自的私有机制。

6. 应用程序服务器如何实现 ServerSession

JMS 规范没有定义一个 ServerSession 是如何实现的，不过这里有一个典型的实现。

（1）应用程序服务器创建一个 ServerSession 线程，并注册 ServerSession 的 runObject，对于应用程序服务器来说，runObject 的实现是私有的。

（2）ServerSession 的 start()方法是调用它线程的 start()方法。就像所有的 Java 线程一样，调用 start()方法来开始线程的执行。

（3）runObject 首先会做一些内部管理，然后再调用 Session 的 run()方法，在返回的时候，runObject 会把它拥有的 ServerSession 放回 ServerSessionPool 然后返回，这将终止 ServerSession 线程的执行，并且可以循环使用。

7. 结果

JMS 定义了一种灵活的机制，会根据每个参与并发消息消费的参与者所具有的权限来合理分工。

JMS 提供者在消息被投递到 MessageListener 前会保留对消息的控制权。这将确保消息的确认被直接控制。

应用程序服务器不但控制着 ConnectionConsumer 的配置，还管理着所有用来执行 MessageListener 的线程。

图 15-4 说明三个角色和它们所实现的对象之间的关系。

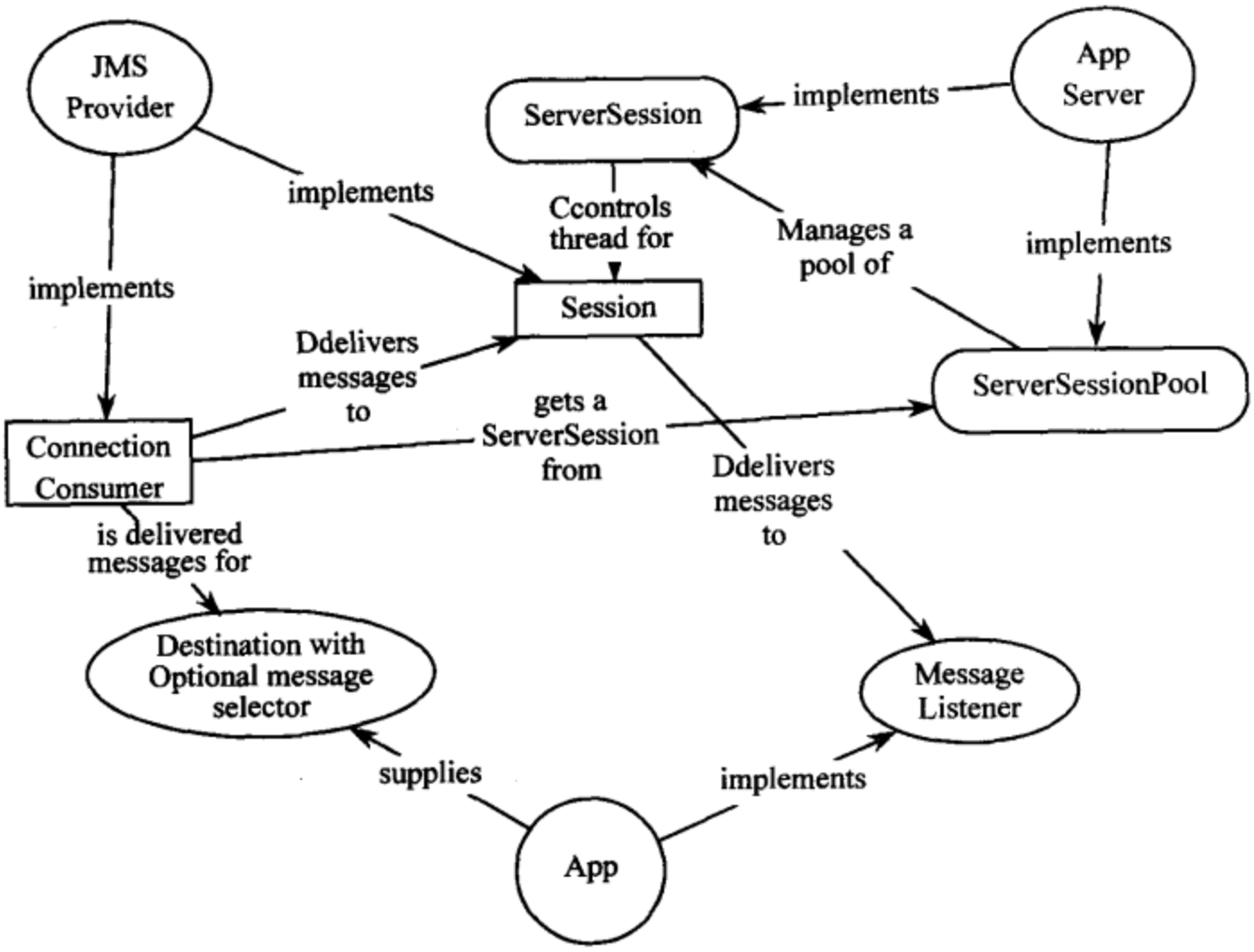


图 15-4　三个角色和它们所实现的对象之间的关系

图 15-5 描述了 ConnectionConsumer 如何把消息投递到 MessageListener。

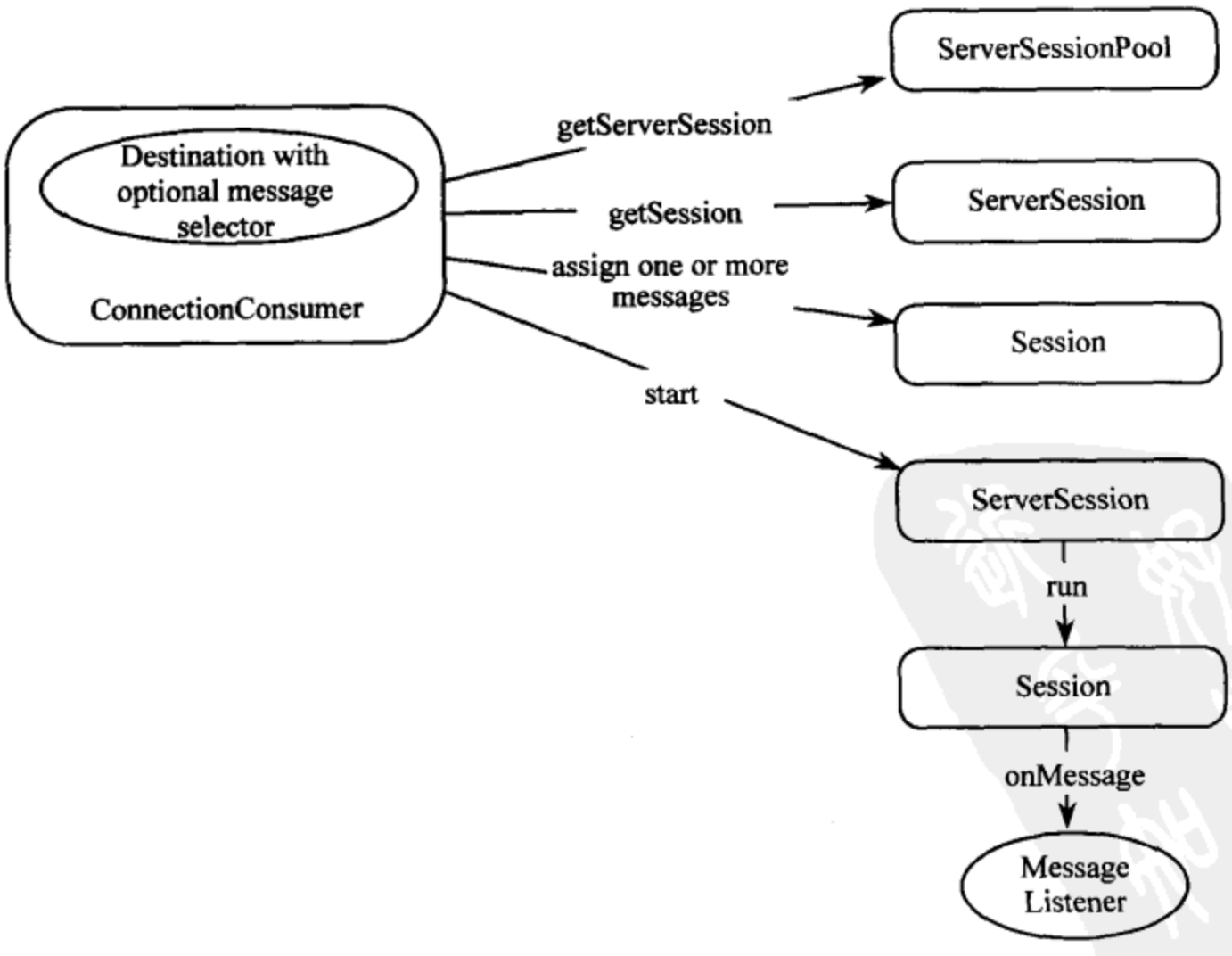


图 15-5　ConnectionConsumer 把消息投递到 MessageListener

15.8.2 XAConnectionFactory

一些应用程序服务器提供在分布式事务中对分配资源的支持。为了把 JMS 事务包含在分布式事务中，应用程序服务器需要让 JMS 提供者使用 Java Transaction API（JTA）。JMS 提供者通过使用 XAConnectionFactory 来支持 JTA。

XAConnectionFactory 提供和 ConnectionFactory 一样的认证选项。XAConnectionFactory 和 ConnectionFactory 一样，都属于 JMS 管理对象，都可以通过 JNDI 来查找它们。

15.8.3 XAConnection

XAConnection 继承了 Connection 的功能，提供创建 XASession 的能力。

15.8.4 XASession

使用 XASession 可以访问普通 Session 对象和由会话事务上下文控制的 javax.transaction.xa.XAResource 对象。XAResource 从功能上很类似那些由 X/Open XA Resource 定义的接口。

应用程序服务器控制着由 XAResource 获得的 XASession 的事务的分配。它使用 XAResource 将会话分配到分布式事务中，准备事务以及提交事务等。

值得注意的是，分布式事务上下文不能跟踪消息，是由于生产消息和接收消息不在同一个事务内。

15.8.5 JMS 应用程序服务器接口

PTP 和 Pub/Sub 模型都提供了相应版本的 JTS，如表 15-6 所示。

表 15-6 PTP 和 Pub/Sub 接口的关系

JMS 通用接口	PTP 接口	Pub/Sub 接口
ServerSessionPool		
ServerSession		
ConnectionConsumer		
XAConnectionFactory	XAQueueConnectionFactory	XATopicConnectionFactory
XAConnection	XAQueueConnection	XATopicConnection
XASession	XAQueueSession	XATopicSession

15.9 JMS 队列生产者/消费者范例

下面通过一个例子来演示发送消息和使用消息。不过，在运行示例代码之前，需要配置一下服务器。

15.9.1　发送消息

J2EE 参考实现预先配有一个队列连接工厂（名为 QueueConnectionFactory）和一个队列（名为 jms/Queue）。如果使用的是 JMS 服务器，而不是参考实现，或者如果你想试着更改队列连接工厂和/或队列的名称，请参考配置服务器一节。

下面是通过一个 JMS 队列发送消息的步骤。从示例程序 TestQueue 中抽出的代码片段也穿插在这些步骤中。

（1）通过按名字在 JNDI 中进行查找，获得一个指向 QueueConnectionFactory 的引用：

```
protected static String qfactoryName ="jms/queue/TechTipsQueueConnectionFactory";
    ...
    try {
    // 获得 JNDI 上下文
    InitialContext ctx = new InitialContext();
    // 获得连接工厂
    QueueConnectionFactory qcf =
    (QueueConnectionFactory)ctx.lookup(qfactoryName);
```

（2）从 QueueConnectionFactory 获得一个 QueueConnection，再从这个连接获得一个 QueueSession，按名字在 JNDI 中查找。

```
// 获得一个到队列的连接
qc = qcf.createQueueConnection();
//从该连接获得一个会话
QueueSession qs = qc.createQueueSession(
      false, Session.AUTO_ACKNOWLEDGE);
// 获得一个队列
Queue q = (Queue)ctx.lookup(queueName);
```

（3）使用 QueueSession 创建一个 QueueSender，将该队列作为一个参数来传递。（QueueSender 是 QueueSession 与某个特定队列之间的一个关联。）

```
// 使用这个会话创建一个 QueueSender
// 和一个 TextMessage
QueueSender qsnd = qs.createSender(q);
```

现在可以用 QueueSender 来发送消息到队列。

（4）创建一个消息对象（Message 的子类），然后使用 QueueSender 的发送方法将它们发送至目的地。示例程序从 Session 中获得一个 TextMessage 对象。接着示例程序将每个程序参数打包到 TextMessage 中，然后使用 QueueSender 将其发送至队列。注意，同一个 TextMessage 可以使用多次。

```
TextMessage tm = qs.createTextMessage();

// 为第一个参数之后的每个参数进行一次循环
// 以文本消息的形式发送参数字符串
for (int i = 2; i < args.length; i++) {
     tm.setText(args[i]);
     qsnd.send;
}
```

（5）关闭 QueueConnection。在 try/finally 程序块的最后一条语句中关闭连接是一个好习惯。这一步很重要：忘记关闭 QueueConnections 将可能导致服务器上的资源泄漏。

```
    } finally {
          if (qc != null) {
             qc.close();

          }
       }
```

要发送消息到一个消息队列，可以使用 TestQueue 程序（在默认的包中），加上一个参数 send，例如：

```
$ java TestQueue send jms/queue/MyTestQueue a b c d
Java Message Service 1.0.2 Reference
Implementation (build b14)
Sent: 'a'
Sent: 'b'
Sent: 'c'
Sent: 'd'
```

15.9.2 接收消息

TestQueue 程序按照以下步骤接收消息：

（1）从一个消息队列接收消息的开始两步与发送消息是一样的：先是查找连接工厂，再获得一个 QueueConnection，然后查找 Queue。

（2）从 QueueSession 获得一个 QueueReceiver：

```
QueueReceiver qrcv = qs.createReceiver(q);
```

（3）从 Queue 接收消息。示例程序进入一个循环，在这个循环中从队列获取消息并将它们打印到标准输出设备。

```
    qc.start();
    Message m = qrcv.receive(10000);
    while (m != null) {

          if (m instanceof TextMessage) {
             TextMessage tm = (TextMessage)m;
           System.out.println("Received text: '" +
                              tm.getText() + "'");
          } else {
             System.out.println("Received a " +
                                m.getClass().getName());
          }
          m = qrcv.receive(100);
       } finally {
          if (qc != null) {
             qc.close();
          }
       }
```

对 QueueConnection 的启动方法的调用将告诉连接开始接收消息。QueueReceiver 接收方法带有一个参数，该参数表明了等待一条消息的毫秒数。如果消息没有在规定时间内到达，该方法将返回 null 值。直到消息队列在 100 毫秒的时间内都保持为空，程序才开始读取和打印收到的消息。在 try/finally 程序块的结尾有一个 finally 子句，该子句像前面例子中一样关闭

QueueConnection。

要接收消息队列中的消息，可以使用 TestQueue 程序，再带上一个参数 recv，例如：

```
$ java TestQueue recv jms/queue/MyTestQueue
Java Message Service 1.0.2 Reference
Implementation (build b14)
Received text: 'a'
Received text: 'b'
Received text: 'c'
Received text: 'd'
```

15.9.3　配置服务器

如果你使用的不是参考引用，或者想更改队列和/或队列连接工厂的名字，就需要通过使用平台的管理工具配置 JMS 提供者。创建队列或者连接工厂之后，便留在服务器中，直到服务器重新启动。有些平台实现可能会为客户端提供扩展 API，以便程序化地创建目的地和连接工厂。

用来在 J2EE SDK 下创建受管理的对象的工具是应用服务器。要配置服务器，需要启动应用服务器。创建一个消息队列，并为它取一个 JNDI 名（只做一次）。要用 J2EE SDK 创建一个新的消息队列，首先需要为消息队列决定一个名字。为了便于管理，应该选择一个可以表明该队列是 JMS 使用的队列的名字，例如 jms/MyTestQueue。然后执行命令：

```
j2eeadmin -addJmsDestination <queuename> queue
```

用消息队列的实际名字替换<queuename>。

由于连接工厂名 QueueConnectionFactory 被硬性地放在示例程序中，因此要告诉程序使用一个不同的连接工厂名，就必须修改源代码，并且重新编译。例如，在 TestQueue.java 中：

```
// 更改变量的值，以更改 Connection factory 的名称
protected static String qfactoryName = "QueueConnectionFactory"
```

重新编译之后，使用 j2eeadmin（或者你自己的服务器的工具）创建一个 connection factory，给它取个 JNDI 名，如：

```
j2eeadmin -addJmsFactory jms/MyQueueConnectionFactory
```

要列出连接工厂，使用：

```
j2eeadmin -listJmsFactory
```

要列出目的地，使用：

```
j2eeadmin -listJmsDestination
```

对于不是 J2EE SDK 的平台，可以在该平台上使用 J2EE 产品的部署工具来创建所需的消息队列和连接工厂。

15.9.4　运行实例代码

可以使用命令 jar xvf ttmar2003.jar 将示例 jar 中的内容解压缩到工作目录下。在运行实例程序前，要确保 J2EE 服务器正在运行。

多次运行实例程序。试着发送一些成批的消息，但是不接收这些消息，检查一下输出的顺序。

这个实例程序为读者试着使用 JMS 队列提供了切入点。探索一下 JMS 接口使用的 API。例如，可以更改一下程序，试着使用三种不同的接收方式。探索一下事务性的消息发送，

或者试着发送各种类型的消息对象。JMS 有许多特性，熟练使用它可以提高读者的应用设计水平。

15.10 小结

本章介绍了 JMS 的基本概念，以及 JMS API 的主要内容。然后通过一个实例演示了如何发送消息、接收消息。

第 16 章　JavaMail 应用开发

尽管 WWW（World Wide Web）在互联网通信中非常流行，但迄今为止 Email 在互联网上的应用仍然是最广泛的，所以 Email 通信在 Java 服务器编程中也占有非常重要的地位。对常规的电子邮件使用者来说，Email 只是非常简单地执行一些常用的邮件传输，但是对编程人员来讲，Email 的打包、传输、发送和接收是很复杂的，这要涉及到许多协议。JavaMail 是 Sun 发布的用来处理这些问题的统一的 API。使用 JavaMail，编程人员就不必自己去处理那些复杂的协议，这些协议中主要有用于发送电子邮件的 SMTP（Simple Message Transfer Protocol，简单邮件传输协议）、用来接收电子邮件的 POP3 和 IMAP（Internet 消息访问协议，Internet Message Access Protocol）以及多用途网络邮件扩充协议（MIME，Multipurpose Internet Mail Extension protocol）。

使用电子邮件主要是发送和接收邮件，这里又涉及到一个服务器和客户端的问题。要使用电子邮件服务就必须要有一个电子邮件服务器。电子邮件服务起源于 UNIX，电子邮件服务器也以运行于 UNIX 服务器较为优秀，就连 Microsoft 的 Hotmail 以前都是使用的 UNIX 服务器。在这些邮件服务器中最著名的当然非 Sendmail 莫属了，但由于 Sendmail 的配置过于复杂，效率不是很高，当前又出现了许多免费的优秀的电子邮件服务器软件，如 qmail、xmail 等，还有 Apache 小组开发的纯 Java 的 James 软件。JavaMail API 主要是封装了电子邮件的基本协议，它可以用在客户端，也可以用在服务器端。但是对于大多数人来讲，自己开发电子邮件服务器没有多大的意义，毕竟已经有很多优秀的电子邮件服务器可供使用，而且还有免费的。这一章是以邮件客户端为主，虽然用 JSP、Servlet 开发的是服务器程序，这主要是针对 Web 服务器来讲的。但对于邮件服务器来讲，它又是邮件客户端。

16.1　用 JavaMail 发送简单邮件

16.1.1　实例说明

这个实例是用 JSP 和 JavaBeans 来发送一个有标题和正文的简单邮件，通过这个实例来初步认识一下 JavaMail 和 SMTP 协议。

SMTP（Simple Mail Transfer Protocol）即简单邮件传输协议，它定义了发送电子邮件的机制。在 JavaMail API 环境中，基于 JavaMail 的程序将和您的公司或因特网服务供应商（Internet Service Provider，ISP）的 SMTP 服务器通信。发送方的 SMTP 服务器可将消息中转至接收方的 SMTP 服务器，以便最终让用户经由 POP 或 IMAP 获得该电子邮件。

要使用 JavaMail 首先要到 http://java.sun.com 下载两个压缩包，一个是 JavaMail，它的最新版本是 JavaMail-1.3；另一个是 JAF（JavaBeans Activation Framework），它的最新版本是 1.0.2。由于 JavaMail 消息依赖这个包，下载后解压并把 JavaMail 的 Mail.jar 和 JAF 的 activation.jar 加入 CLASSPATH 中，这样程序才能正确编译。在 Tomcat 中使用时，前边已经提到如何设置 Tomcat 使用的 CLASSPATH，编辑 CATALINA_HOME\bin 中的 setclasspath.bat 或者把 jar 文件放到 WEB-INF\lib 目录里。

16.1.2 准备工作

由于在互联网上发送垃圾邮件的人太多，SMTP 服务器已逐渐地需要认证了，现在找一个可以匿名发邮件的 SMTP 服务器还真是很难。这个实例使用的是 localhost 的 SMTP，这就需要安装一个邮件服务器。邮件服务器的配置是相当麻烦的，但如果只是使用 SMTP 就很简单了。现在免费的邮件服务器很多，尤其是 Apache 小组还开发了纯 Java 的邮件服务器 James。虽然还没有正式稳定版本出台，但 SMTP 已经是成熟的了，而且 James 是开放源代码的，有兴趣的读者还可以参考 James 的源代码。

安装 James 是非常简单的，只要到 www.apache.org 下载一个压缩包，最新版本是 james-2.0a3.zip。解压缩后直接到解压的目录中运行 James-2.0a3\lib\run.bat 就可以了，注意要确保你的机器上没有其他的邮件服务器。如果有的话要先关闭其他邮件服务器，以免引起端口冲突，并造成启动失败。

启动 James 的结果如图 16-1 所示。

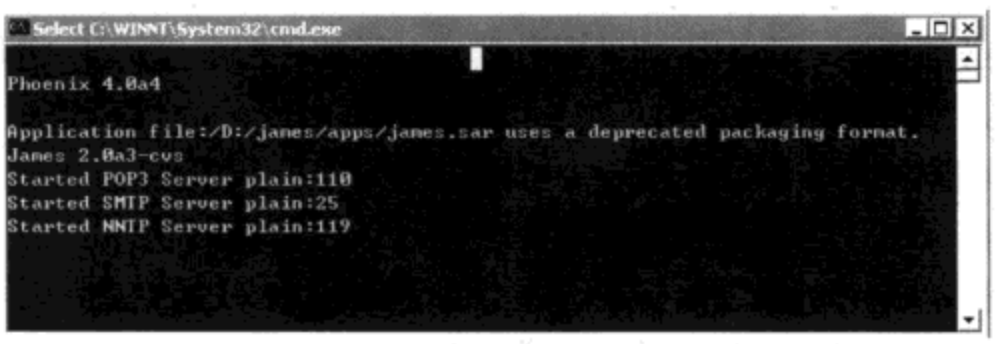


图 16-1 James 邮件服务器的启动

启动 James 以后就需要添加用户。打开 MS DOS 窗口，输入 telnet localhost 4555。输入以下命令（命令不显示在窗口）：

```
help
adduser elba 1234
setpassword elba 1234
listusers
```

结果如图 16-2 所示。

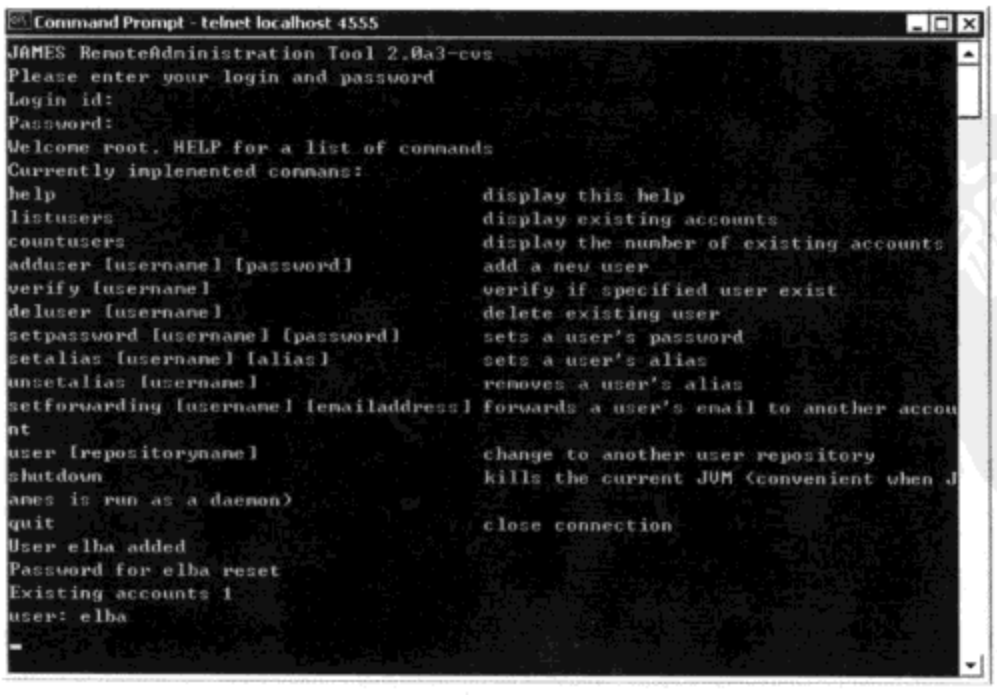


图 16-2 邮件服务器的命令窗口

从中可以看到已经成功地添加了用户 elba，密码为 1234。这样就可以使用 James 服务器来处理 Email 了。

16.1.3　编程思路

这个实例很简单，只需要输入接收者的邮件地址、邮件标题和邮件正文，发送邮件的程序代码封装到 Javabeans 里，JSP 文件代码相对就非常简单了。重点是使用 JavaMail 的 API，这些可以在 JavaMail 附带的文档里找到，或者到 Sun 公司的网站找到。

16.1.4　代码分析

首先是发送 Email 的 HTML 文件。通过这个文件来收集数据，把数据提交到 JSP 程序处理，并发送 Email。

simplemail.htm 的代码如下所示：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body bgcolor="#FFFFFF" text="#000000">
<br>
<p align="center"><b>简单的发送邮件实例</b></p>
<form name="form1" method="post" action="simplemail.jsp">
  <table    width="75%"    border="0"    cellspacing="2"    cellpadding="5"
align="center">
    <tr>
      <td align="right">收件人：</td>
      <td>
        <input type="text" name="to">
      </td>
    </tr>
    <tr>
      <td align="right">邮件标题：</td>
      <td>
        <input type="text" name="subject">
      </td>
    </tr>
    <tr>
      <td align="right">邮件正文：</td>
      <td>
        <textarea name="body" cols="36" rows="4"></textarea>
      </td>
    </tr>
    <tr>
      <td align="right"> </td>
      <td>
        <input type="submit" name="Submit" value="发送邮件">
      </td>
    </tr>
  </table>
</form>
</body>
```

```
</html>
```

这个 JSP 文件非常简单，只包括一个 Javabeans 的调用，以判断发送邮件成功或失败就可以了。simplemail.jsp 的代码如下所示：

```
<%@ page contentType="text/HTML;charset=GB2312" %>
<jsp:useBean id="mailBean" scope="page" class="mail.SimpleMail"/>
<jsp:setProperty name="mailBean" property="*"/>
<%
try{
    //调用 send 函数发送邮件
    if(mailBean.send())
        out.print("Send mail success. ");
    else
        out.print("Send mail failed.");
}catch(NullPointerException npe){
    //如果没有数据提交直接调用此文件，程序会抛出 NullPointerException
}catch(Exception e){
    out.println(e);
}
%>
```

下边是发送邮件的 JavaBeans 文件。SimpleMail.java 的代码如下所示：

```
package mail;

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;

public class SimpleMail{
    String to="";           //邮件接收者
    String subject="";      //邮件标题
    String body="";         //邮件正文
    //设置邮件接收者
    public void setTo(String to){
        this.to=to;
    }
    //设置邮件标题
    public void setSubject(String subject){
        try{//处理汉字编码
            this.subject = new String(subject.getBytes("iso-8859-1"),"gb2312");
        }catch(Exception e){
        }
    }
    //设置邮件正文
    public void setBody(String body){
        try{//处理汉字编码
            this.subject = new String(body.getBytes("iso-8859-1"),"gb2312");
        }catch(Exception e){
        }
    }
    //发送邮件
    public boolean send(){
        //SMTP 服务器是 localhost
```

```
        String smtp="localhost";
        try{
            //得到系统属性
            Properties props=System.getProperties();
            //设置 SMTP 服务器
            props.put("mail.smtp.host", smtp);
            //得到一个会话
            Session session=Session.getInstance(props, null);
            //生成一个 Message 实例
            Message msg = new MimeMessage(session);
            //设置发件人
            msg.setFrom(new InternetAddress("yourmail@yourdomain.com"));
//设置收件人
msg.setRecipients(Message.RecipientType.TO,InternetAddress.parse(to,false));
//设置邮件标题
msg.setSubject(subject);
//设置邮件正文
            msg.setText(body);
            //设置发送时间为系统当前时间
            msg.setSentDate(new Date());
            //发送邮件
            Transport.send(msg);
            return true;
        }catch(Exception e){
            e.pringStackTrace();
            return false;
        }
    }
}
```

16.1.5　运行结果

在 simplemail.htm 页面中输入内容，然后单击“发送邮件”按钮，如图 16-3 所示。

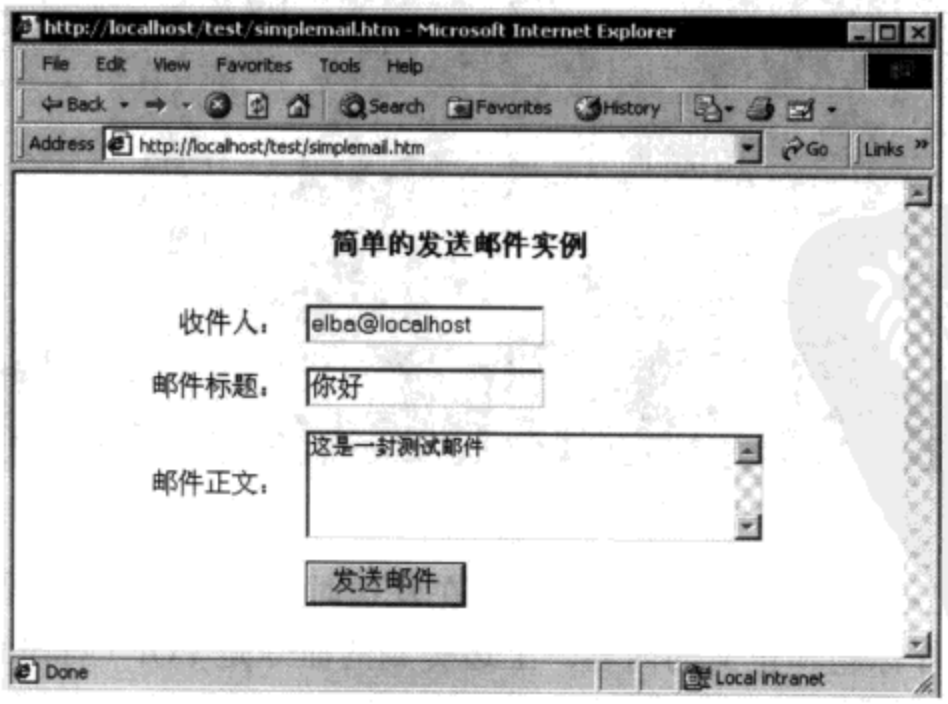


图 16-3　Simplemail.htm 的页面

发送成功后的页面如图 16-4 所示。

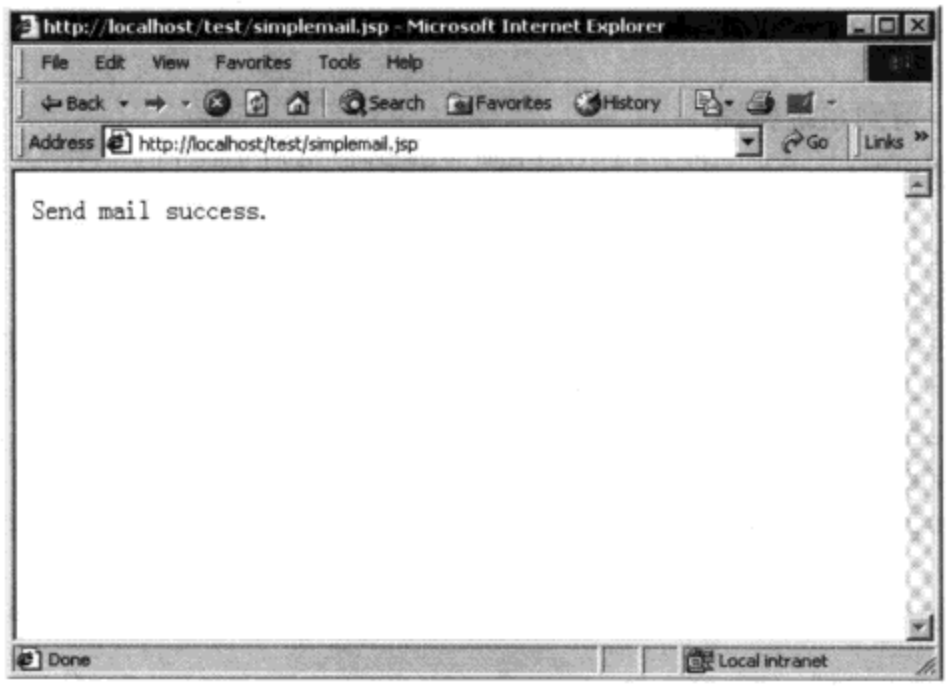


图 16-4　邮件发送成功的页面

这时就可以接收邮件了，以验证邮件是否被成功发送。一种方法是用邮件客户端软件，如 Outlook Express 或 Foxmail 等。另一种是以命令行方式登录 POP3 服务器，并在其中输入以下命令（命令不可见）：

```
telnet localhost 110
user  elba
pass 1234
list
retr 1
```

结果如图 16-5 所示。

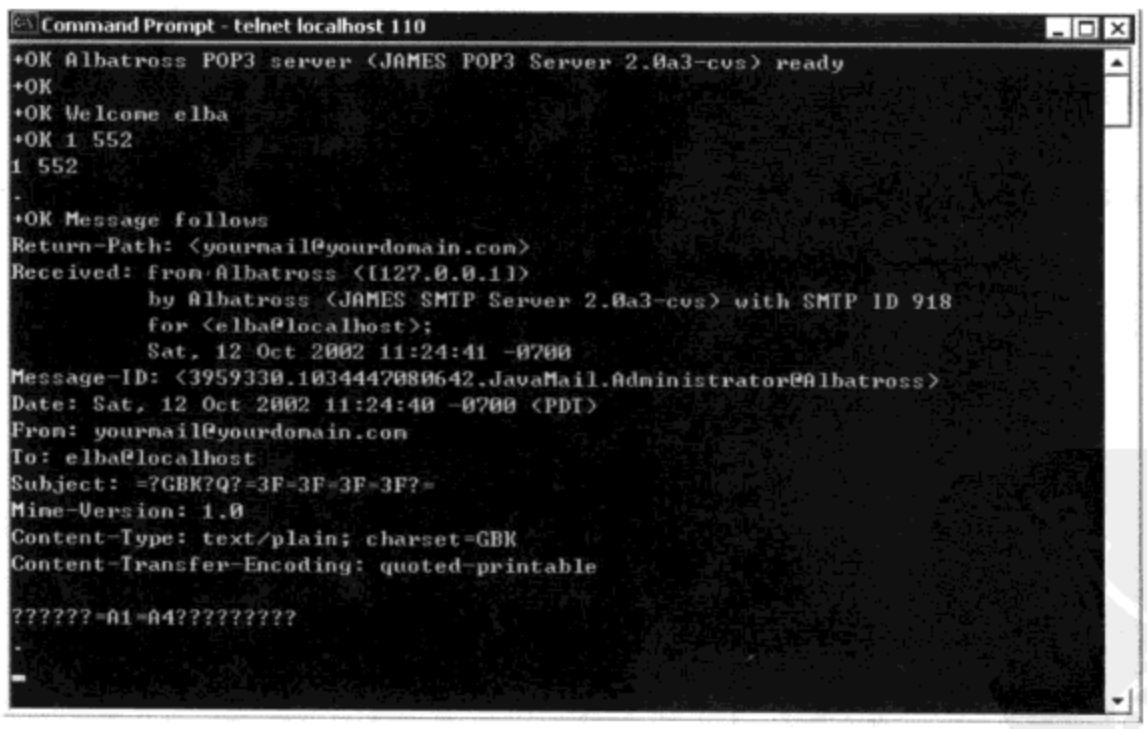


图 16-5　使用 POP3 方式接收电子邮件

可以看到邮件已经在邮箱里了。由于邮件在发送时已经进行了编码处理（这个以后会讲到），所以看不出邮件标题和正文的内容。上边提到的命令是 POP3 标准命令，不能通过 help 方式得到，有兴趣的读者可以参考 rfc 文档第 1939 篇。

如果使用外部邮箱，即便程序运行成功也不能保证你会准确地收到邮件。现在有很多邮

件服务器拒绝接收 localhost 发出的邮件，会将这种邮件认为是垃圾邮件。

16.2　用 JavaMail 发送 HTML 邮件

16.2.1　编程思路

HTML 文件主要由两部分构成：HTML 文本和图片。在 HTML 中加入图片有两种方法，第一种是直接把图片链接到网上的可见图片，这种方法简单实用，而且不会使生成的邮件过大，但是图片必须是网上可见的。还有一种办法是直接把图片加入邮件，实际上就是把图片作为邮件的附件，把图片链接指向这个附件。

（1）打开 Outlook Express 并创建一个新邮件，如图 16-6 所示。

（2）选择以后发送，该邮件就会出现在发件箱中。单击邮件以查看邮件属性，如图 16-7 所示。

图 16-6　创建一个新邮件

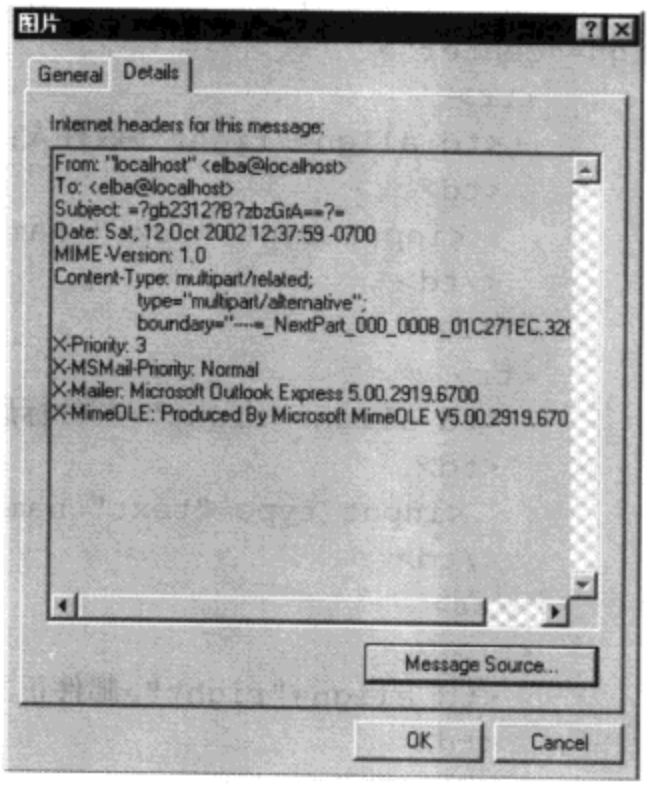


图 16-7　邮件的属性

（3）选择邮件详细资料后，以查看邮件源代码，如图 16-8 所示。

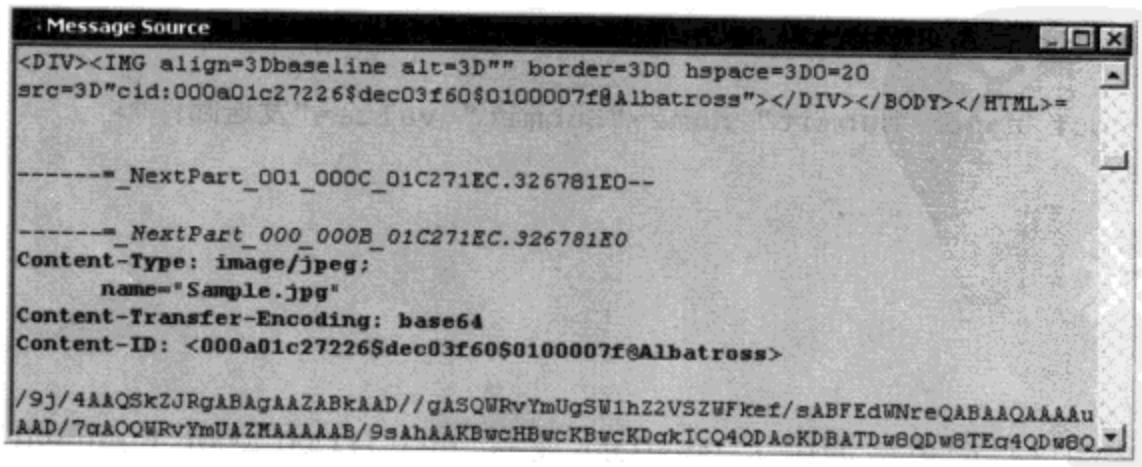


图 16-8　邮件的源代码

从图 16-8 中可以看到图片的 src 指向与下边图片文件的 Content-ID 相同。

由于加入图片的第二种方法通过 Web 方式处理比较复杂，这里就不举例说明了，如果有兴趣可以自己研究一下。

16.2.2 代码分析

下边这个例子是简单地发送 HTML 邮件的 HTML 代码。

htmlmail.htm 的代码如下所示：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body bgcolor="#FFFFFF" text="#000000">
<br>
<p align="center"><b>发送 HTML 邮件实例</b></p>
<form name="form1" method="post" action="htmlmail.jsp">
  <table width="75%" border="0" cellspacing="2" cellpadding="5"
align="center">
    <tr>
      <td align="right">收件人：</td>
      <td>
        <input type="text" name="to">
      </td>
    </tr>
    <tr>
      <td align="right">邮件标题：</td>
      <td>
        <input type="text" name="subject">
      </td>
    </tr>
    <tr>
      <td align="right">邮件正文：</td>
      <td>
        <textarea name="body" cols="36" rows="4"></textarea>
      </td>
    </tr>
    <tr>
      <td align="right"> </td>
      <td>
        <input type="submit" name="Submit" value="发送邮件">
      </td>
    </tr>
  </table>
</form>
</body>
</html>
```

这个 JSP 文件只是简单地调用了 JavaBean 文件 mail.HTMLMail。htmlmail.jsp 的代码如下所示：

```
<%@ page contentType="text/HTML;charset=GB2312" %>
<jsp:useBean id="mailBean" scope="page" class="mail.HTMLMail"/>
```

```
<jsp:setProperty name="mailBean" property="*"/>
<%
try{
    //调用 send 函数发送 Email
    if(mailBean.send())
        out.print("Send mail success. ");
    else
        out.print("Send mail failed.");
}catch(NullPointerException npe){
    //如果没有数据提交，只是调用本页，程序会抛出 NullPointerExcetpion
}catch(Exception e){
    out.println(e);
}
%>
```

JavaBeans 文件源代码 HTMLMail.java 的代码如下所示：

```
import javax.activation.*;

public class HTMLMail{
    String to="";              //收件人
    String subject="";         //邮件标题
    String body="";            //邮件正文
    //设置收件人
    public void setTo(String to){
        this.to=to;
    }
//设置邮件标题
public void setSubject(String subject){
    try{
        this.subject = new String(subject.getBytes("iso-8859-1"),"gb2312");
    }catch(Exception e){
    }
}
//设置邮件正文
public void setBody(String body){
    try{
        this.body = new String(body.getBytes("iso-8859-1"),"gb2312");
    }catch(Exception e){
    }
}
//发送邮件函数
public boolean send(){
    //SMTP 服务器为 localhost
    String smtp="localhost";
    try{
        //获得系统属性
        Properties props=System.getProperties();
        //设置 SMTP 服务器
        props.put("mail.smtp.host", smtp);
        //得到会话
        Session session=Session.getInstance(props, null);
```

```
            //生成邮件实例
            Message msg = new MimeMessage(session);
            //设置发件人
            msg.setFrom(new InternetAddress("yourname@yourdomain.com"));
            //设置收件人
msg.setRecipients(Message.RecipientType.TO,InternetAddress.parse(to,false));
            //设置邮件标题
            msg.setSubject(subject);
            //设置邮件正文
            msg.setDataHandler(new DataHandler(new
            ByteArrayDataSource(body,"text/html")));
            //设置发送时间为系统时间
            msg.setSentDate(new Date());
            //发送邮件
            Transport.send(msg);
            return true;
        }catch(Exception e){
            e.printStackTrace();
            return false;
        }
    }
}
```

上边的 HTMLMail.java 依赖于 ByteArrayDataSource.java 文件，该文件的作用是把字符串转化为输入字节流。这个文件在 JavaMail\demo 中。ByteArrayDataSource.java 的代码如下所示：

```
package mail;

import java.io.*;
import javax.activation.*;

public class ByteArrayDataSource implements DataSource {
    private byte[] data; // 字节数组
    private String type; // 数据类型

    // 由输入流构建 DataSource
    public ByteArrayDataSource(InputStream is, String type) {
        this.type = type;
        try {
            ByteArrayOutputStream os = new ByteArrayOutputStream();
            int ch;

             while ((ch = is.read()) != -1)
              //这里可以通过缓冲处理以提高效率
             os.write(ch);
             data = os.toByteArray();
        } catch (IOException ioex) { }
    }

    //由字节数组构建 DataSource
    public ByteArrayDataSource(byte[] data, String type) {
        this.data = data;
```

```
        this.type = type;
    }

    //由字符串构建 DataSource，这是 HTMLMail.java 用到的方法
    public ByteArrayDataSource(String data, String type) {
     try {
        this.data = data.getBytes("gb2312");
     } catch (UnsupportedEncodingException uex) { }
     this.type = type;
    }

    //返回 InputStream 的函数
    public InputStream getInputStream() throws IOException {
     if (data == null)
        throw new IOException("no data");
     return new ByteArrayInputStream(data);
    }
     //返回 OutputStream 的函数
    public OutputStream getOutputStream() throws IOException {
         throw new IOException("cannot do this");
    }
     //返回数据类型的函数
    public String getContentType() {
       return type;
    }
}
```

16.2.3　运行结果

在页面中添入数据，如图 16-9 所示。

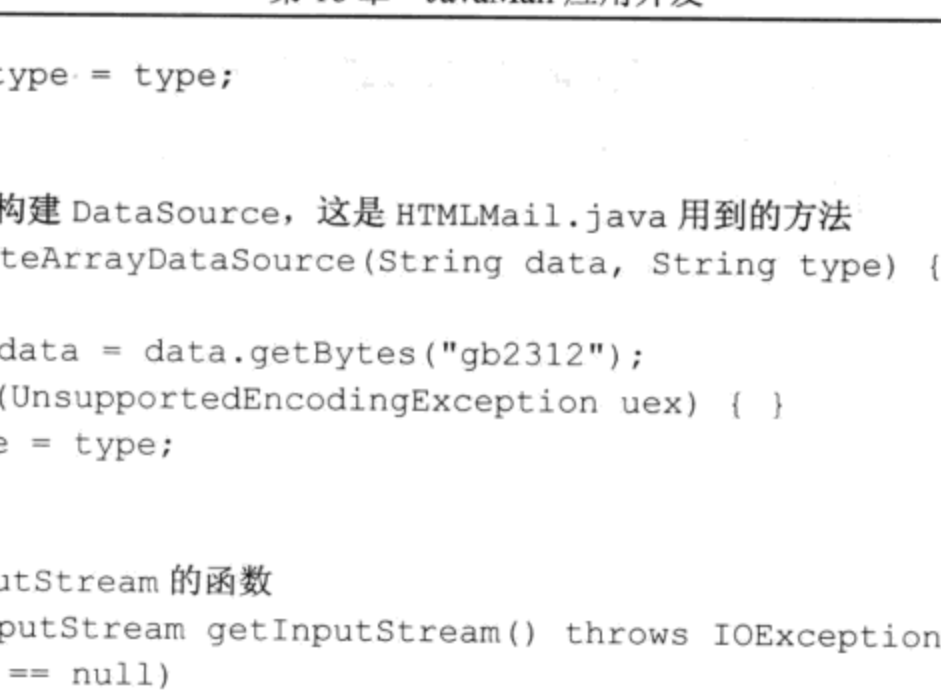


图 16-9　htmlmail.htm 的页面

邮件发送成功后的页面如图 16-10 所示。

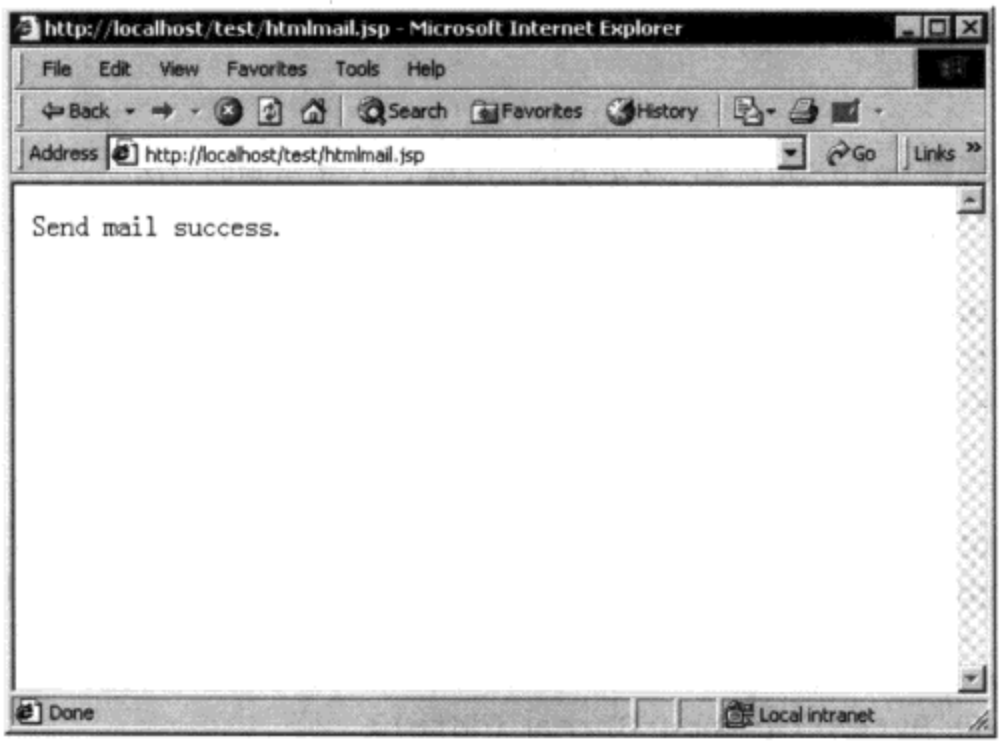


图 16-10　邮件发送成功后的页面

用 Outlook Express 接收此邮件，结果如图 16-11 所示。

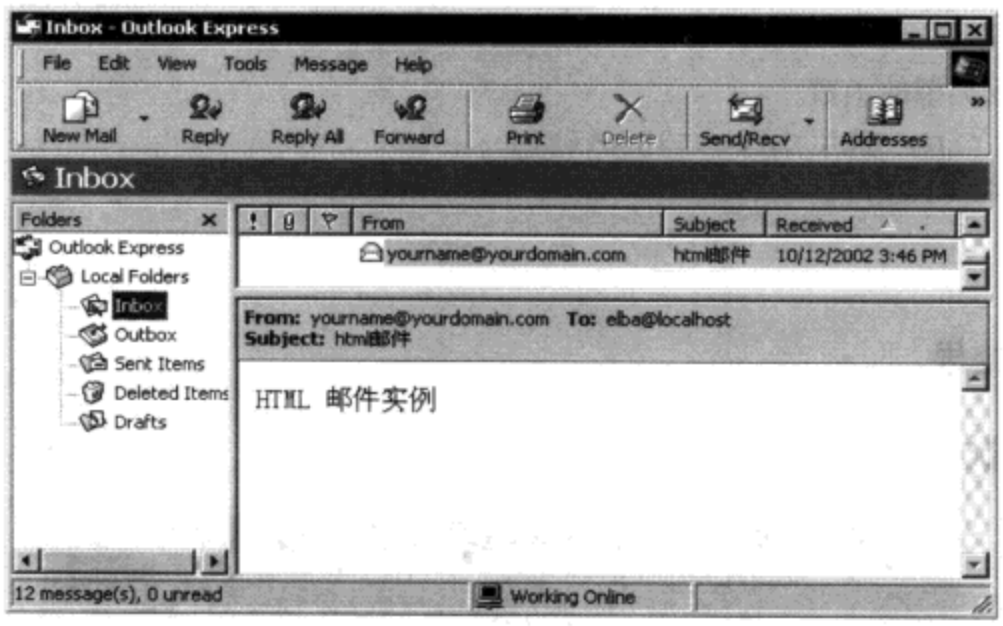


图 16-11　使用 Outlook Express 接收邮件

16.3　用 JavaMail 发送需要 SMTP 认证的邮件

16.3.1　实例说明

前边已经提到，由于大量的垃圾邮件的影响，现在绝大多数公用的 SMTP 服务器都需要认证才能发送邮件。Email 提供商也会告诉用户如何设置服务器，如 Outlook Express 一般进行如下设置，在菜单栏上单击“工具”→“账号”→“属性”→“服务器设置”，这时会出现如图 16-12 所示的窗口。在下边的 Incoming Mail Server（发送邮件服务器）中有一栏“Log on using Secure Password Authentication（我的邮件服务器需要身份认证）”，在这个前边打勾，这样用户才能成功地发送邮件。

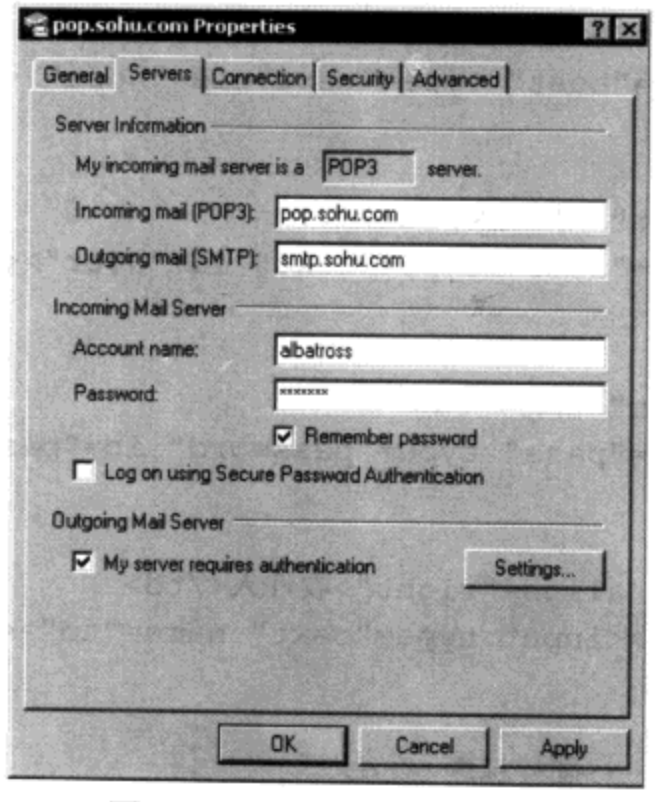


图 16-12　邮件服务器的设置

而对于发送邮件的程序来讲就不像打个勾这么简单了，当然也不是很复杂，因为 JavaMail 已经为我们考虑到了，只要继承一个 javax.mail.Authenticator 的类就可以了。

下边这个实例就是把前边的发送简单的邮件修改了一下，同时加上了服务器的验证。你会发现实现服务器身份验证真的很简单。

16.3.2　编程思路

实现 SMTP 服务器身份认证，只需要继承一个 javax.mail.Authenticator 类就可以了。这里把这部分独立出去会显得代码更清晰一些。

16.3.3　代码分析

下面是一个 HTML 文件，它需要用户输入发信人地址、SMTP 服务器、用户名和密码等信息。smtpauth.htm 的代码如下所示：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body>
<p>  </p>
<form name="form1" method="post" action="smtpauth.jsp">
  <table width="85%" border="0" align="center">
    <tr align="center">
      <td colspan="2">发送邮件示例</td>
    </tr>
    <tr>
      <td align="right">发件人</td>
      <td><input name="from" type="text" id="from"></td>
    </tr>
    <tr>
```

```
  <td align="right">SMTP 服务器</td>
  <td><input name="host" type="text" id="host"></td>
</tr>
<tr>
  <td align="right">用户名</td>
  <td><input name="user" type="text" id="user"></td>
</tr>
<tr>
  <td align="right">密码</td>
  <td><input name="pass" type="password" id="pass"></td>
</tr>
<tr>
  <td width="29%" align="right">收件人</td>
  <td width="71%"><input type="text" name="to"></td>
</tr>
<tr>
  <td align="right">邮件标题</td>
  <td><input name="subject" type="text" id="subject"></td>
</tr>
<tr valign="top">
  <td height="56" align="right">邮件正文</td>
  <td><textarea name="body" cols="45" rows="5" id="body"></textarea></td>
</tr>
<tr valign="top">
  <td height="32" align="right"> </td>
  <td><input type="submit" name="Submit" value="·¢ËÍÓÊ¼þ"></td>
</tr>
</table>
</form>
</body>
</html>
```

这里 JSP 文件调用了 SmtpAuth 类。smtpauth.jsp 的代码如下所示：

```
<%@ page contentType="text/HTML;charset=GB2312"%>
<jsp:useBean id="mailBean" scope="page" class="mail.SmtpAuth"/>
<jsp:setProperty name="mailBean" property="*"/>
<%
try{
    //调用 send 函数发送邮件
    if(mailBean.send())
        out.print("Send mail success. ");
    else
        out.print("Send mail failed.");
}catch(NullPointerException npe){
}catch(Exception e){
    out.println(e);
}
%>
```

下面是发送邮件的 JavaBeans，SmtpAuth.java 的代码如下所示：

```
package mail;
```

```
import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;

public class SmtpAuth{
    String from="";      //发件人地址
    String smtp="";      //SMTP 服务器
    String user="";      //用户名
    String pass="";      //密码
    String to="";        //收件人地址
    String subject="";   //邮件标题
    String body="";      //邮件正文
    //设置发件人地址
    public void setFrom(String from){
        this.from=from;
    }
    //设置 SMTP 服务器
    public void setSmtp(String smtp){
        this.smtp=smtp;
    }
    //设置用户名
    public void setUser(String user){
        this.user=user;
    }
    //设置密码
    public void setPass(String pass){
        this.pass=pass;
    }
    //设置收件人地址
    public void setTo(String to){
        this.to=to;
    }
    //设置邮件标题
    public void setSubject(String subject){
        try{
            this.subject=new String(subject.getBytes("iso-8859-1"),"gb2312");
        }catch(Exception uee){
            //捕捉 UnsupportedEncodingException
        }
    }
    //设置邮件正文
    public void setBody(String body){
        try{
            this.body=new String(body.getBytes("iso-8859-1"),"gb2312");
        }catch(Exception uee){
            //捕捉 UnsupportedEncodingException
        }
    }
    //发送邮件函数
    public boolean send(){
```

```
try{
    //获得系统属性
    Properties props=System.getProperties();
    //实例 SMTP 认证对象
    SmtpAuthenticator sa=new SmtpAuthenticator(user,pass);
    //设置 SMTP 服务器
    props.put("mail.smtp.host", smtp);
    //设置需要身份认证为真
    props.put("mail.smtp.auth","true");
    //获得一个会话
    Session session=Session.getInstance(props,sa);
    //生成邮件对象
    Message msg = new MimeMessage(session);
    //设置发件人
    msg.setFrom(new InternetAddress(from));
    //设置收件人地址
    msg.setRecipients(Message.RecipientType.TO,
                          InternetAddress.parse(to,false));
    //设置邮件标题
    msg.setSubject(subject);
    //设置邮件正文
    msg.setText(body);
    //设置发信时间为系统当前时间
    msg.setSentDate(new Date());
    //发送邮件
    Transport.send(msg);
        return true;
    }catch(Exception e){
        e.printStackTrace();
        return false;
    }
  }
}
```

SmtpAuth.java 依赖 SMTP 服务器认证的类。SmtpAuthenticator.java 的代码如下所示：

```
package mail;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.mail.*;
import javax.mail.internet.*;
public class SmtpAuthenticator extends javax.mail.Authenticator{
    String username;          //用户名
    String password;          //密码
    //实例化这个类时设置用户名和密码
    public SmtpAuthenticator(String username,String password){
        this.username=username;
        this.password=password;
    }
    //重载 getPasswordAuthentication 函数
    protected PasswordAuthentication getPasswordAuthentication(){
```

```
            return new PasswordAuthentication(username,password);
        }
}
```

16.3.4　运行结果

在 HTML 文件 smtpauth.htm 中输入数据，如图 16-13 所示。

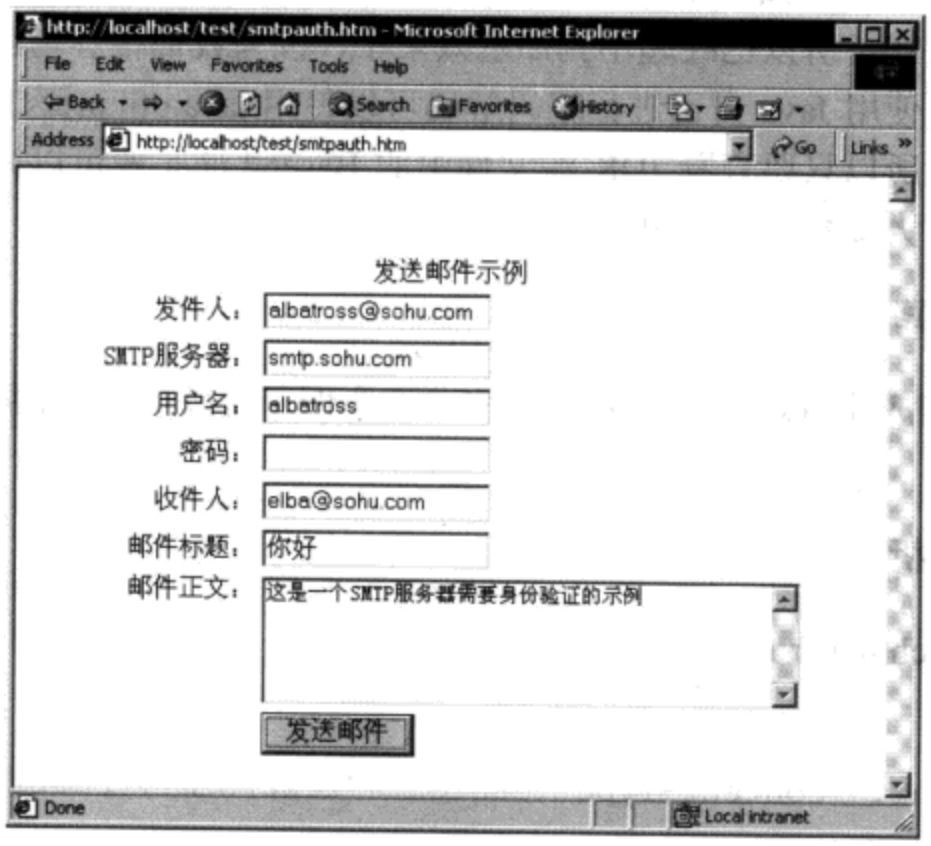


图 16-13　smtpauth.htm 的页面

单击“发送邮件”按钮后的页面如图 16-14 所示。

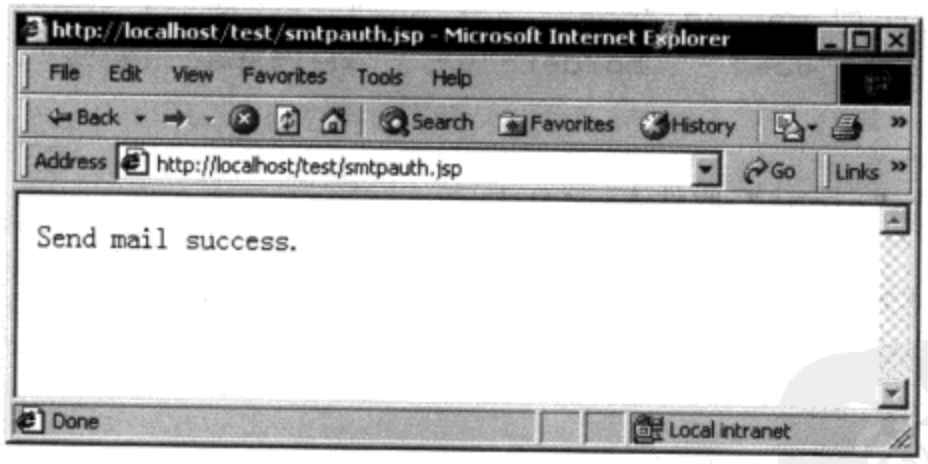


图 16-14　邮件发送成功的页面

16.4　用 JavaMail 发送带附件的邮件

16.4.1　实例说明

这个实例是演示如何在发送的邮件中加入附件。邮件中夹带附件的应用还是比较广的，但由于附件也是病毒的主要载体，所以对于来路不明的邮件如果携带附件最好直接删除。

如果要了解邮件是怎样携带附件的，可以按照前边讲的用 Outlook Express 发送图片的例子中的方法，先构建一个带有附件的邮件，然后查看它的源代码就可以看出邮件是怎样携带附件了。在邮件源码中，可以看到正文和附件内容都是由一些字母、数字和加号、斜杠等少数符号构成的，这是因为 Outlook Express 已经对邮件进行了编码。在邮件源码中，如果我们看到“Content-Transfer-Encoding: base64”这样的文字，就说明邮件的编码方式是 base64。邮件的编码方式很多，其中应用最广的就是 base64 了，JavaMail 默认也是使用 base64 编码。base64 是一种开放的编码方式，所以已经起不到加密效果了，因为我们很容易得到 base64 的编码方式或编码解码函数。使用 Java 自己进行编码解码可以使用 java.util.prefs.Base64 类，因为 JDK 也是开放源代码的，所以只要查看 JDK 源文件中的 Base64.java 就可以详细地了解 Base64 的编码规律了，当然对大多数用户来说没有这个必要。

16.4.2 编程思路

带附件的邮件构成比普通邮件要复杂一些，这个例子是使用本地硬盘上的文件作为附件发送的。

16.4.3 代码分析

HTML 文件 attachmail.htm 的代码如下：

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body bgcolor="#FFFFFF" text="#000000">
<br>
<p align="center"><b>发送带附件的邮件实例</b></p>
<form name="form1" method="post" action="attachmail.jsp">
  <table width="85%" border="0" cellspacing="2" cellpadding="5"
align="center">
    <tr>
      <td align="right">收件人：</td>
      <td>
        <input type="text" name="to">
      </td>
    </tr>
    <tr>
      <td align="right">邮件标题：</td>
      <td>
        <input type="text" name="subject">
      </td>
    </tr>
    <tr>
      <td align="right">邮件正文：</td>
      <td>
        <textarea name="body" cols="36" rows="4"></textarea>
      </td>
    </tr>
```

```
    <tr>
      <td align="right">附件：</td>
      <td>
        <input type=file name="file">
      </td>
    </tr>
     <tr>
      <td align="right"> </td>
      <td>
        <input type="submit" name="Submit" value="发送邮件">
      </td>
    </tr>
  </table>
</form>
</body>
</html>
```

这个 JSP 文件调用 AttachMail 类来发送邮件。Attachmail.jsp 的源代码如下：

```
<%@ page contentType="text/HTML;charset=GB2312"%>
<jsp:useBean id="mailBean" scope="page" class="mail.AttachMail"/>
<jsp:setProperty name="mailBean" property="*"/>
<%
try{
    //调用 send 函数发送邮件
    if(mailBean.send())
        out.print("发送邮件成功");
    else
        out.print("发送邮件失败");
}catch(NullPointerException npe){
}catch(Exception e){
    out.println(e);
}
%>
```

AttachMail.java 的代码如下：

```
package mail;

import java.util.*;
import javax.mail.*;
import javax.mail.internet.*;
import javax.activation.*;

public class AttachMail{
    String to="";              //收件人
    String subject="";         //邮件标题
    String body="";            //邮件正文
    String file="";            //文件名
    //设置收件人
    public void setTo(String to){
        this.to=to;
    }
    //设置邮件标题
```

```
public void setSubject(String subject){
    try{
        this.subject=new String(subject.getBytes("iso-8859-1"),"gb2312");
    }catch(Exception uee){
    }
}
//设置邮件正文
public void setBody(String body){
    try{
        this.body=new String(body.getBytes("iso-8859-1"),"gb2312");
    }catch(Exception uee){
    }
}
//设置文件
public void setFile(String file){
    this.file=file;
}
//发送邮件函数
public boolean send(){
    String smtp="localhost";
    try{
        //得到系统属性
        Properties props=System.getProperties();
        //设置 SMTP 服务器
        props.put("mail.smtp.host", smtp);
        //得到会话
Session session=Session.getInstance(props, null);
//生成 Message 实例
Message msg = new MimeMessage(session);
//设置发件人
msg.setFrom(new InternetAddress("yourmail@yourdomain.com"));
//设置收件人
msg.setRecipients(Message.RecipientType.TO,
                  InternetAddress.parse(to,false));
//设置邮件标题
msg.setSubject(subject);

//生成一个 Multipart 对象，由邮件正文和附件构成邮件体
Multipart multipart = new MimeMultipart();
MimeBodyPart mbp;
mbp = new MimeBodyPart();
mbp.setText(body);
//加入正文
multipart.addBodyPart(mbp);
mbp = new MimeBodyPart();
//加入文件
DataSource source = new FileDataSource(file);
mbp.setDataHandler(new DataHandler(source));
//解析出文件名
mbp.setFileName(getFileName(file));
```

```
                multipart.addBodyPart(mbp);

                // 把邮件体加入邮件中
                msg.setContent(multipart);
                //设置发送邮件时间为系统当前时间
                msg.setSentDate(new Date());
                //发送邮件
                Transport.send(msg);
                return true;
            }catch(Exception e){
                e.printStackTrace();
                return false;
            }
        }
        //解析文件名的函数
        public String getFileName(String str){
            int i=str.lastIndexOf("\\");
            return str.substring(i+1);
        }
    }
```

这个示例有点欺骗的嫌疑，添加附件时并不是真正地把文件传递上去了，而是程序只接收了文件的路径，程序通过这个路径找到文件。由于是在本地测试，当然不会出问题，但这样的程序千万不能放到互联网上使用。

这个示例的主要目的是演示如何把文件作为附件加入到邮件体中，如果不在本机运行程序会找不到附件。可以考虑使用第 4 章讲的文件上传的例子，先把文件上传到服务器上再调用文件，这样就不会有问题了。接下来的例子将演示真正地上传文件而且不保存文件，这样省去了读写文件，效率会更高一些，而且可以同时上传多个附件。

16.4.4　运行结果

在 HTML 文件 attachmail.htm 中输入数据，如图 16-15 所示。

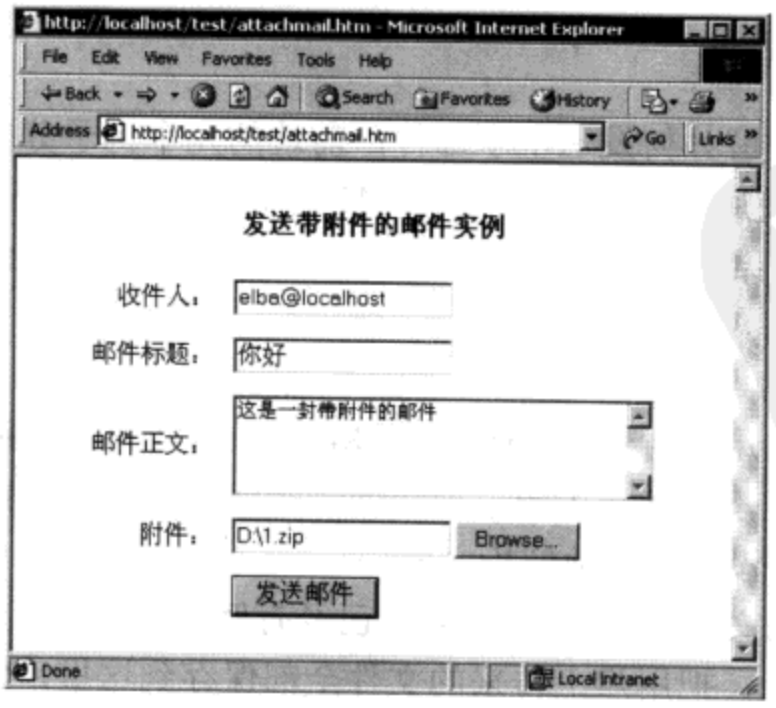


图 16-15　attachmail.htm 的页面

单击“发送邮件”按钮，以提交数据。结果页面如图 16-16 所示。

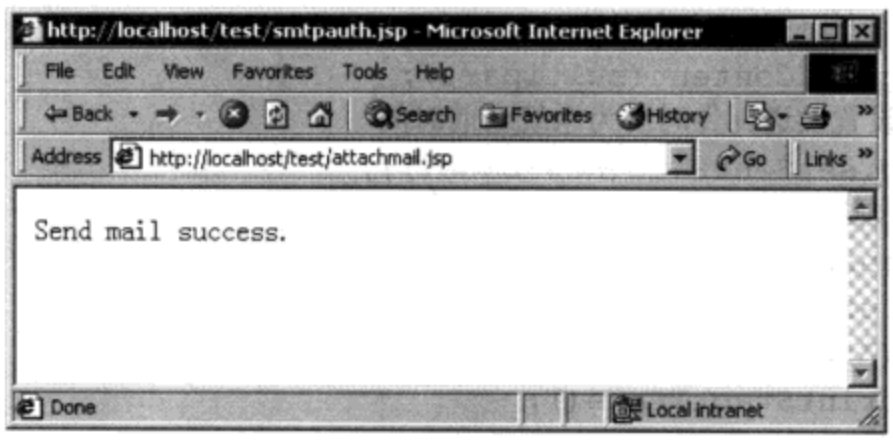


图 16-16　邮件发送成功的页面

用 Outlook Express 接收该邮件，可以看到该邮件带有附件，如图 16-17 所示。

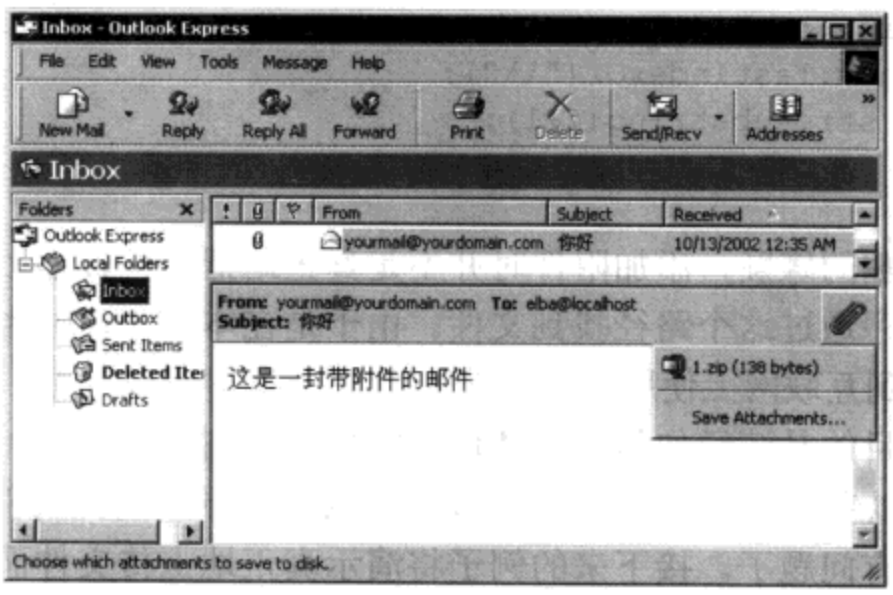


图 16-17　使用 Outlook Express 接收带有附件的邮件

16.5　发送电子邮件综合演练

16.5.1　实例说明

为了使大家更容易理解如何发送电子邮件，前边的几个例子分别示范了如何发送简单邮件、发送 HTML 邮件、使用需要身份认证的 SMTP 服务器、发送带有附件的邮件。下边这个例子是把前边所有的例子结合到一起的一个综合的发送电子邮件的示例。

16.5.2　编程思路

由于已经有了前边的几个例子，如何进行 SMTP 服务器的身份验证、如何发送 HTML 邮件和带有附件的邮件已经没有什么困难。前边的例子演示发送附件的时候只是简单地使用了上传来的文件路径，而并没有真正地解析文件。

如何处理用户上传的文件，因为在第 4 章已经有例子演示如何上传文件，所以可以创建一个保存临时文件的路径，把用户上传来的文件全部保存到这里，然后再作为附件发出，最后删除临时文件。这当然可行，但是它有两个缺点。一个是安全问题，前边已经提到过开放

可写的路径是巨大的安全隐患。第二个问题就是运行效率问题，这个临时文件需要经历写、读、删除三个阶段，每一步操作都很费时间，这就造成了执行效率的低下。还有一种办法是解析出文件以后不保存成文件，而是以字节流的形式保存在内存中，这样就省去了文件的操作，从而极大地提高了执行效率。这种方法也有一个缺点就是占用内存较多，这个问题可以通过限制用户上传的文件大小来改善。而事实上邮件服务器对邮件的大小都是有限制的，至少邮箱的大小是有限制的，把一个 5MB 的邮件发到一个只有 2MB 大小的邮箱当然是没有意义的。所以限制用户上传的文件的大小是没有什么问题的。

16.5.3　代码分析

sendmail.jsp 的代码如下所示：

```
<%@  page  contentType="text/HTML;charset=GB2312"  import="mail.formData,
java.util.*"%>
<jsp:useBean id="formBean" scope="page" class="mail.ParseForm"/>
<jsp:useBean id="mailBean" scope="page" class="mail.SendMail"/>
<%
try{
    formData fd;                    //自定义的数据类型
    //调用 ParseForm 解析浏览器数据，并返回给 Vector
    Vector vfd=formBean.parse(request);
    //调用 SendMail 发送邮件
    if(mailBean.sendmail(vfd)){
        out.print("发送邮件成功。");
    }else
        out.print("发送邮件失败。");
}catch(NullPointerException npe){
    //如果没有数据提交，会抛出 NullPointerException 异常
}catch(Exception e){
    out.println(e);
}
%>
<html>
<head>
<style>
td { font:9pt 宋体;}
input{ font:9pt 宋体;}
</style>
<script language=javascript>
<!-
//如果服务器不需要身份验证，则 username 和 password 可以不填写
function checkauth(arg1,arg2){
    if(arg2){
        if(arg1=="0"){
            document.form1.username.disabled=true;
            document.form1.password.disabled=true;
        }else{
            document.form1.username.disabled=false;
            document.form1.password.disabled=false;
```

```
            }
        }else{
            if(arg1=="1"){
                document.form1.username.disabled=true;
                document.form1.password.disabled=true;
            }else{
                document.form1.username.disabled=false;
                document.form1.password.disabled=false;
            }
        }
}
//-- >
</script>
</head>
<body bgcolor="#FFFFFF">
<form name="form1" method="post" enctype="multipart/form-data" action="">
  <table width="95%" border="0" cellspacing="0" cellpadding="0">
    <tr height=40>
     <td align="center" colspan=2><h4>发送邮件示例</h4></td>
    </tr>
    <tr >
     <td align="right" width="25%">发信人：</td>
     <td width="75%">
       <input type="text" name="from">
     </td>
    </tr>
    <tr>
     <td align="right" width="25%">
       <p>Smtp 服务器：</p>
     </td>
     <td width="75%">
       <input type="text" name="smtp">
       <input type="radio" name="authority" value="1" checked onclick="checkauth
(this.value,this.checked)">
       需要身份验证
       <input type="radio" name="authority" value="0" onclick="checkauth
(this.value,this.checked)">
       不需要身份验证 </td>
    </tr>
    <tr >
     <td align="right" width="25%">端口号：</td>
     <td width="75%">
       <input type="text" name="port" value=25>
     </td>
    </tr>
    <tr >
     <td align="right" width="25%">用户名：</td>
     <td width="75%">
       <input type="text" name="username">
     </td>
```

```
</tr>
<tr >
  <td align="right" width="25%">密码：</td>
  <td width="75%">
    <input type="password" name="password">
  </td>
</tr>
<tr>
  <td align="right" width=25%>收信人：</td>
  <td width="75%">
    <input type="text" name="to">
  </td>
</tr>
 <tr>
  <td align="right" width=25%>抄送：</td>
  <td width="75%">
    <input type="text" name="cc">
  </td>
</tr>
 <tr>
  <td align="right" width=25%>暗送：</td>
  <td width="75%">
    <input type="text" name="bcc">
  </td>
</tr>
<tr>
          <td align="right" width="25%">标题：</td>
          <td width="75%">
            <input type="text" name="subject">    <input type=checkbox
name=html value=1>正文以 HTML 格式发送
          </td>
        </tr>
        <tr valign="top">
          <td align="right" width="25%">正文：</td>
          <td width="75%">
            <textarea name="body" rows="4" cols="40"></textarea>
          </td>
        </tr>
        <tr valign=top>
          <td align="right" width="25%">
            <input type="button" name="button" value="添加附件" onclick="addattach()">
               </td>
          <td width="75%">
            <script language=javascript>
                 //预置 20 个文件并隐藏，当用户添加附件的时候显示
                 for(i=1;i<21;i++)
                     document.write("<div id=att"+i+" style='display:none'><table
border=0   ><tr><td><input   name=attach"+i+"   type=file   disabled=true><input
type=button value='删除附件' onclick=delatt('"+i+"')></td></tr></table></div>");
                 var attnum=0;
```

```
            //添加附件的时候显示一个文件
            function addattach(){
                attnum++;
                nf=eval("document.form1.attach"+attnum);
                nf.disabled=false;
                ni=eval("document.all.att"+attnum);
                ni.style.display='inline';
            }
            //删除附件
            function delatt(i){
                nf=eval("document.form1.attach"+i);
                nf.value="";
                nf.disabled=true;
                ni=eval("document.all.att"+i);
                ni.style.display='none';
            }
        </script>
      </td>
    </tr>
    <tr>
      <td align="center" colspan=2>
        <input type="submit" value="发送邮件">
      </td>
    </tr>
  </table>
</form>
</body>
</html>
```

formData.java 的代码如下所示：

```
package mail;
//自定义数据类型
import java.util.*;
public class formData{
    public String name; //表单项名称
    public String value;     //表单项的值。如果是文件，这个值就是文件名
    public String contentType;  //如果是文件，这个值就是文件的类型，否则为 null
    public byte[] file;      //把文件保存为字节数组
}
```

SmtpAuthenticator.java 的代码如下所示：

```
//用于 SMTP 服务器的身份验证
package mail;

import javax.activation.DataHandler;
import javax.activation.FileDataSource;
import javax.mail.*;
import javax.mail.internet.*;
public class SmtpAuthenticator extends javax.mail.Authenticator{
    String username;          //用户名
    String password;          //密码
    //实例化这个类时设置用户名和密码
```

```
    public SmtpAuthenticator(String username,String password){
        this.username=username;
        this.password=password;
    }
    //重载 getPasswordAuthentication 函数
    protected PasswordAuthentication getPasswordAuthentication(){
        return new PasswordAuthentication(username,password);
    }
}
```

parseRequest.java 的代码如下所示：

```
/*前边已经讲过关于文件上传并解析表单数据，但只是解析一个文件。这个例子稍微
*复杂一些，表单中可以有任意的条目（text、button、textarea 和 checkbox 等），也可
*以有任意多的文件，但原理还是一样的。
*这个文件是在解析表单后，返回一个 formData 的 Vector，并且文件保存在 byte 数组中。
*注意：这个文件只在 Windows 环境下成功运行，在其他操作系统可能因为换行符的
*问题运行失败，请注意做出相应的修改。
*/
import java.util.*;
import java.io.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.ServletInputStream;
import javax.servlet.ServletException;

public class ParseRequest{
    private String ContentType = "";              //表单类型
    private String CharacterEncoding = "";        //编码格式

    //设定表单类型
    public void setContentType(String s){
        ContentType = s;
        int j;
        if((j = ContentType.indexOf("boundary=")) != -1){
            ContentType = ContentType.substring(j + 9);
            ContentType = "--" + ContentType;
        }
    }
    //设置文件编码格式
    public void setCharacterEncoding(String s){
        CharacterEncoding = s;
    }

    public Vector getFormData( HttpServletRequest req)
throws ServletException, IOException{
        setCharacterEncoding(req.getCharacterEncoding());
        setContentType(req.getContentType());
        return getFormData(req.getInputStream());
    }
    //解析表单数据并保存到 Vector 中
    public Vector getFormData( ServletInputStream servletinputstream) throws
ServletException, IOException{
```

```
        Vector formdatas=new Vector();
        formData fd=null;
        int size=2048000;          //上传的文件的最大限制为 2MB
        byte[] buffer = new byte[size];
        int bufferLength=0;
        byte temp[];
        byte Linebyte[] = new byte[4096];              //设定缓冲区大小为 4096
        //由于 Java 中不可以用指针，在函数中改变外部的值使用数组方法
        int ai[] = new int[1];
        int ai1[] = new int[1];

        String line;
        String boundary="--";
        String itemname="";
        String filename="";
        //textarea 的值可能有回车，用 StringBuffer 累加
        StringBuffer sb=new StringBuffer("");
        boolean nameflag=false;
        boolean fileflag=false;
        int skipline=0;
        int i;
/* 从 ServletInputStream 读取一行。注意这里不是严格地读取一行，如果在小于
*  缓冲大小的时候出现换行符，则读取一行，否则读满缓冲区
*  Linebyte：缓冲区
*  ai：读到的位置，也就是换行符的位置
*  servletinputstream：浏览器请求
*  CharacterEncoding：数据类型
*/
    while((line = readLine(Linebyte, ai, servletinputstream,
CharacterEncoding)) != null){
        //得到 boundary
if((i=line.indexOf("boundary=--"))>=0){
            boundary=line.substring(i+10);
            continue;
        }
        //这里使用 nameflag 和 fileflag 控制读取
        if(!nameflag&&!fileflag){
            //得到表单项的名称
            itemname=getName(line);

            if(itemname!=null&&itemname!=""){
                fd=new formData();
                fd.name=itemname;
                nameflag=true;
                //得到文件名
                filename=getFileName(line);

                if(filename!=null && filename!=""){
                    fd.value=filename;
                    fileflag=true;
```

```
            }else{
                fileflag=false;
            }
            skipline=0;
            continue;
        }
        continue;
    }
    //如果表单项的类型不是文件
    if(nameflag&&!fileflag){
        if(skipline<1){
        //跳过一行
            skipline++;
            continue;
        }
        else{
            //得到表单项的值
            if(line.indexOf(boundary)>=0){
                nameflag=false;
                fd.value=trim(sb.toString());
                formdatas.addElement(fd);
                sb=new StringBuffer("");
            }else{
                    sb.append(line);
            }
        }
        continue;
    }
    //如果表单项的类型是文件
    if(nameflag&&fileflag){

            if(skipline<1){
                //得到文件类型
                if(line.indexOf("Content-Type:")>=0)
                    fd.contentType=trim(line.substring(14));
                else{
                    //跳过一行
                    skipline++;
                }
                continue;
            }else{
                    //把文件内容保存到 byte 数组中
                    if(line.indexOf(boundary)>=0){

                        if(bufferLength>=2){
                            //去掉最后的换行符
                            bufferLength-=2;
                            temp=new byte[bufferLength];
                            for(i=0;i<bufferLength;i++)
                                temp[i]=buffer[i];
```

```
                                fd.file=temp;
                                buffer=new byte[size];
                                bufferLength=0;
                            }

                            formdatas.addElement(fd);
                            nameflag=false;
                            fileflag=false;
                        }else{
                            for(i=0;i<ai[0];i++,bufferLength++)
                                buffer[bufferLength]=Linebyte[i];

                        }
                    }
                }
            }
            return formdatas;
        }
        //得到表单项的名称
        public String getName(String s){
            int i=s.indexOf(" name=\"");
            if(i>0){
                s=s.substring(i+7);
                i=s.indexOf("\"");
                return s.substring(0,i);
            }else
                return "";
        }

        //获取文件名
        private String getFileName(String s){
            try{
                int i = s.indexOf("filename=");
                if(i >= 0){
                    s = s.substring(i + 10);
                    if((i = s.indexOf("\"")) > 0)
                        s = s.substring(0, i);
                    i = s.lastIndexOf("\\");
                    if(i < 0 || i >= s.length() - 1){
                        i = s.lastIndexOf("/");
                        if(i < 0 || i >= s.length() - 1)
                            return s;
                    }
                    return s.substring(i + 1);
                }else
                    return "";
            }catch(Exception e){
                return "";
            }
        }
```

```
    //readLine 函数，用于把表单提交上来的数据按行读取
    private String readLine(byte Linebyte[], int ai[],ServletInputStream
servletinputstream,String CharacterEncoding){
        try{ //读取一行
            ai[0] = servletinputstream.readLine(Linebyte, 0, Linebyte.length);
            if(ai[0] == -1)
                return null;
        }catch(IOException ex){
            return null;
        }
        try{
            if(CharacterEncoding == null){
                //用默认的编码方式把给定的 byte 数组转换为字符串
                return new String(Linebyte, 0, ai[0]);
            }else{
                //用给定的编码方式把给定的 byte 数组转换为字符串
                return new String(Linebyte, 0, ai[0], CharacterEncoding);
            }
        }catch(Exception ex){
            return null;
        }
    }
    //这个 trim 函数是用来去掉回车换行符的，而不是用 String 中的 trim 函数去掉空格
    public String trim(String s){
        if(s.charAt(s.length()-1)=='\n'){
            if(s.length()<=2)
                return "";
            else
                s=s.substring(0,s.length()-2);
            return trim(s);
        }
        if(s.indexOf("\r\n")==0){
            s=s.substring(2);
            return trim(s);
        }
        return s;
    }
}
```

ByteArrayDataSource.java 的代码如下所示：

```
package mail;

import java.io.*;
import javax.activation.*;

public class ByteArrayDataSource implements DataSource {
    private byte[] data; // 字节数组
    private String type; // 数据类型

    // 由输入流构建 DataSource
```

```
public ByteArrayDataSource(InputStream is, String type) {
    this.type = type;
    try {
       ByteArrayOutputStream os = new ByteArrayOutputStream();
     int ch;

     while ((ch = is.read()) != -1)
          //这里可以通过缓冲处理以提高效率
        os.write(ch);
     data = os.toByteArray();
    } catch (IOException ioex) { }
}

//由字节数组构建 DataSource
public ByteArrayDataSource(byte[] data, String type) {
    this.data = data;
    this.type = type;
}

//由字符串构建 DataSource
public ByteArrayDataSource(String data, String type) {
 try {
     this.data = data.getBytes("iso-8859-1");
 } catch (UnsupportedEncodingException uex) { }
 this.type = type;
}

    //返回 InputStream 的函数
    public InputStream getInputStream() throws IOException {
     if (data == null)
         throw new IOException("no data");
         return new ByteArrayInputStream(data);
    }
     //返回 OutputStream 的函数
    public OutputStream getOutputStream() throws IOException {
         throw new IOException("cannot do this");
    }
     //返回数据类型的函数
    public String getContentType() {
       return type;
    }
}
```

SendMail.java 的代码如下所示：

```
package mail;

import java.io.IOException;
import java.io.OutputStream;
import java.util.Date;
import java.util.Hashtable;
import javax.activation.DataHandler;
```

```
import javax.activation.FileDataSource;
import javax.mail.*;
import javax.mail.internet.*;
import java.util.*;
import java.io.File;

public class SendMail{
    //发送一个邮件
    public boolean sendmail(Vector vfd){
        String smtp="";      //SMTP 服务器
        String port="25";    //SMTP 服务器端口，默认为 25
        String authority="";//是否需要身份验证
        String username=""; //用户名
        String password=""; //密码
        String from="";      //发件人
        String subject="";   //标题
        String body="";      //正文
        String html="";      //是否使用 HTML 格式发送邮件
        String to="";        //收件人
        String cc="";        //抄送
        String bcc="";       //暗送
        String contentType="text/html";  //默认的数据类型

        formData fd;  //自定义数据类型
        //获取 Vector 中的数据
        Enumeration e=vfd.elements();
        while(e.hasMoreElements()){
            fd=(formData)e.nextElement();
            if(fd.name.equals("smtp")){
                smtp=fd.value;
                continue;
            }else if(fd.name.equals("port")){
                port=fd.value;
                continue;
            }else if(fd.name.equals("authority")){
                authority=fd.value;
                continue;
            }else if(fd.name.equals("username")){
                username=fd.value;
                continue;
            }else if(fd.name.equals("password")){
                password=fd.value;
                continue;
            }else if(fd.name.equals("from")){
                from=fd.value;
                continue;
            }else if(fd.name.equals("subject")){
                subject=fd.value;
                continue;
            }else if(fd.name.equals("body")){
```

```
            body=fd.value;
            continue;
        }else if(fd.name.equals("html")){
            html=fd.value;
            continue;
        }else if(fd.name.equals("to")){
            to=fd.value;
            continue;
        }else if(fd.name.equals("cc")){
            cc=fd.value;
            continue;
        }else if(fd.name.equals("bcc")){
            bcc=fd.value;
        }
    }
    MimeBodyPart mbp=null;
    Multipart mp=null;
    //得到系统属性
    Properties props=System.getProperties();
    Session session=null;
    Message msg=null;
    //如果服务器需要身份验证
    if(authority.equals("1")&&username!=null&&password!=null){
        //实例化服务器身份验证对象
        SmtpAuthenticator sa=new SmtpAuthenticator(username,password);
        //设置 SMTP 服务器
            props.put("mail.smtp.host", smtp);
            //设置身份验证的属性为真
            props.put("mail.smtp.auth","true");
            //设置端口
            props.put("mail.smtp.port", port);
            //得到会话
            session=Session.getInstance(props,sa);
        }else{  //如果不需要身份验证
            //设置 SMTP 服务器
            props.put("mail.smtp.host", smtp);
            //设置端口
            props.put("mail.smtp.port", port);
            //得到会话
            session=Session.getInstance(props, null);
        }
    try{
        //生成邮件实例
        msg = new MimeMessage(session);
        //设置发件人
        msg.setFrom(new InternetAddress(from));
        //设置收件人
        if(!to.equals(""))
            msg.setRecipients(Message.RecipientType.TO,InternetAddress.parse
(to,false));
```

```
            //设置抄送收件人
            if (!cc.equals(""))
                msg.setRecipients(Message.RecipientType.CC,InternetAddress.parse
(cc,false));
            //设置暗送收件人
    if (!bcc.equals(""))
                msg.setRecipients(Message.RecipientType.BCC,InternetAddress.parse
(bcc,false));
            //设置邮件标题
            msg.setSubject(subject);

            mbp = new MimeBodyPart();
            //设置邮件正文
            mbp.setText(body);

            mp = new MimeMultipart();
            //加入到邮件正文
            mp.addBodyPart(mbp);
            MimeBodyPart mbpatt;
            //得到一个文件，并加入到附件中
            e=vfd.elements();
            while(e.hasMoreElements()){
                fd=(formData)e.nextElement();
                if(fd.name.indexOf("attach")==0&&fd.file!=null&&fd.file.length>0){
                    mbpatt=new MimeBodyPart();
                    mbpatt.setDataHandler(new DataHandler(new
ByteArrayDataSource(fd.file,fd.contentType)));
                    mbpatt.setFileName(fd.value);
                    mp.addBodyPart(mbpatt);
                }
            }
            //把正文和附件加入邮件体
            msg.setContent(mp);
            //设置发信日期为系统当前日期
            msg.setSentDate(new Date());
            //发送邮件
            Transport.send(msg);
        }catch(Exception ex){
            System.out.println(ex);
            return false;
        }
        return true;
        }
}
```

16.5.4　运行结果

运行 sendmail.jsp 并添入如下数据，结果如图 16-18 所示。

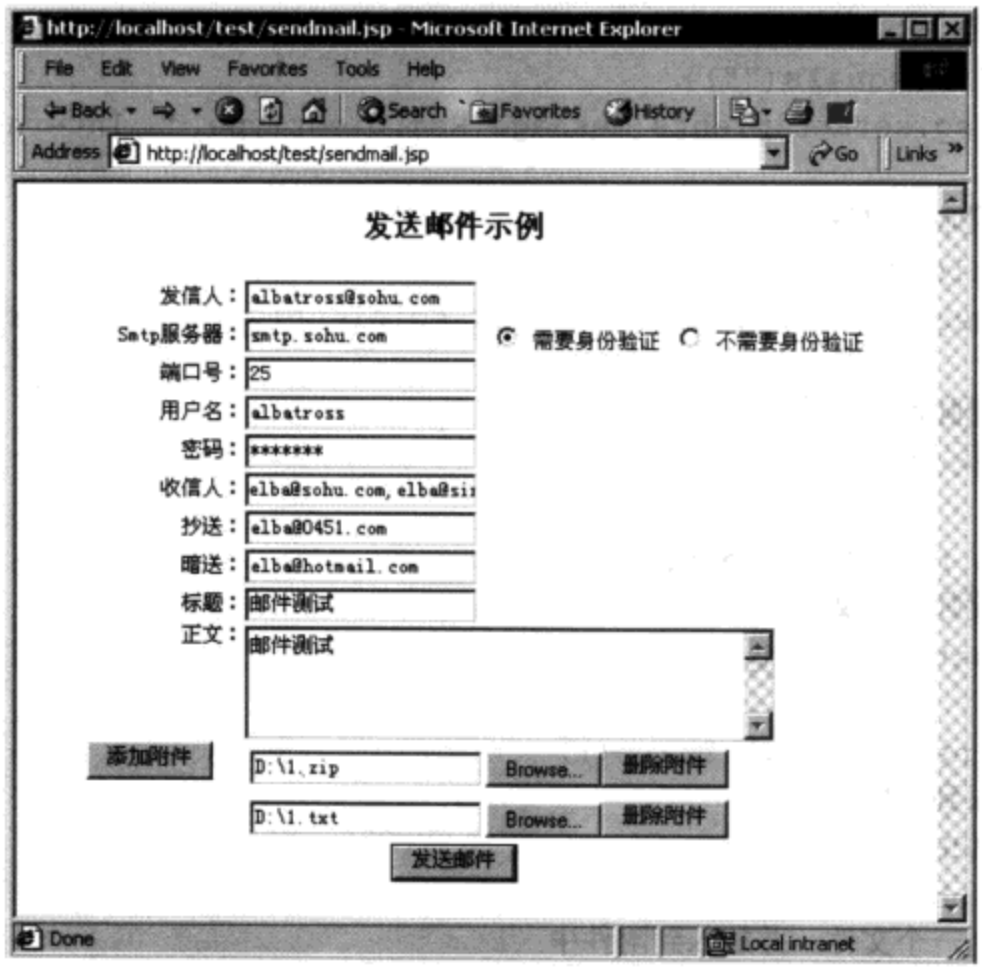


图 16-18　sendmail.jsp 的页面

单击“发送邮件”按钮，发送邮件后提示发送成功。

16.6　用 JavaMail 接收邮件列表

16.6.1　实例说明

这个例子演示如何使用 JavaMail API 接收邮件。接收邮件的协议主要有两个：一个是 POP3 协议，另一个是 IMAP4 协议。

POP（Post Office Protocol）指邮局协议。由于目前用的是版本 3，所以人们通常将它称为 POP3，RFC 1939 定义了这个协议。POP 和 SMTP 一样，也是一种机制，Internet 上大多数人通过它得到邮件。该协议规定每个用户只能有一个邮箱支持，这就是它所能做的，而这也造成了许多混淆。使用 POP 时，用户熟悉的许多性能并不是由 POP 协议支持的，如查看有几封新邮件消息这一性能。这些性能内建于如 Microsoft Outlook 之类的程序中，它们能记住一些事，诸如最近一次收到的邮件、还能计算出有多少是新的邮件。所以当使用 JavaMail API 时，如果读者想要知道这类信息，就只能由自己来计算了。

IMAP（Internet Message Access Protocol）即 Internet 消息访问协议，是一种更高级的用于接收消息的协议，在 RFC 2060 中有它的定义。目前使用的 IMAP 版本为 4，人们习惯将它称为 IMAP4。在用到 IMAP 时，邮件服务器必须支持这个协议。不能仅仅把使用 POP 的程序用于 IMAP，并指望它支持 IMAP 所有性能。假设邮件服务器支持 IMAP，基于 JavaMail 的程序可以利用这种情况。用户在服务器上可以有多个文件夹（folder），并且这些文件夹可以被多个用户共享。

因为具有这一更高级的性能，您也许会认为所有用户都会使用 IMAP，事实并不是这样。IMAP 要求服务器接收新消息，在用户请求时将这些新消息发送到用户手中，同时还要在每个用户的多个文件夹中维护消息。这样虽然能将消息集中备份，但随着用户长期的使用，文件夹越来越大，到磁盘空间耗尽时，每个用户都会受到损失。而使用 POP 协议，就能卸载邮件服务器上保存的消息了。这也是大多数的邮件服务器不支持 IMAP 的原因。

16.6.2　编程思路

由于 IMAP 的使用范围远不如其设计时想像的那样，这个例子先介绍如何使用 POP3 协议接收邮件。其要实现的功能是查看邮件列表，所以只需要解析邮件头就可以了。邮件在邮件服务器上有一个惟一标识，在 POP3 协议中可以使用 UIDL 命令得到这个标识，同时可以将它使用在 JavaMail API 中。

16.6.3　代码分析

maillist.htm 的代码如下所示：

```
<html>
<head>
<title>邮件列表</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body>
<p> </p>
<p align="center"> <strong>查看邮件</strong></p>
<form name="form1" method="post" action="maillist.jsp">
  <table width="75%" border="0" align="center">
    <tr>
      <td align="right">POP3 服务器：</td>
      <td><input name="host" type="text" id="host"></td>
    </tr>
    <tr>
      <td align="right">服务器端口：</td>
      <td><input name="port" type="text" id="port" value="110"></td>
    </tr>
    <tr>
      <td align="right">用户名：</td>
      <td><input name="username" type="text" id="username"></td>
    </tr>
    <tr>
      <td align="right">密码：</td>
      <td><input name="password" type="password" id="password"></td>
    </tr>
    <tr valign="bottom">
      <td height="46" align="right"> </td>
      <td><input type="submit" name="Submit" value="查看邮件">
      </td>
    </tr>
  </table>
```

```
</form>
</body>
</html>
```

maillist.jsp 的代码如下所示：

```
<%@pagecontentType="text/HTML;charset=GB2312"
import="java.net.*,java.io.*,java.util.*"%>
<jsp:useBean id="mailBean" scope="page" class="mail.MailList"/>
<%
String host="";          //POP3 服务器
int port;                //服务器端口
String username="";      //用户名
String password="";      //密码
//获取用户提交的数据
host=request.getParameter("host");
username=request.getParameter("username");
password=request.getParameter("password");
//把端口转化为整数型
try{
    port=Integer.parseInt(request.getParameter("host"));
}catch(Exception e){
    port=110;
}
//把用户提交的数据保存到 Session 中，以便于读取邮件详细内容时使用
session.putValue("host",host);
session.putValue("port",new Integer(port));
session.putValue("user",username);
session.putValue("pass",password);

try{
    //调用 list 函数得到邮件信息，并保存在二维数组中
    String[][] msgs=mailBean.list(host,port,username,password);
    //得到邮箱内的邮件总数
    out.print("<br>您的邮箱中共有 "+mailBean.getTotalMessages()
+"  封邮件<br>");
    if(mailBean.getTotalMessages()>0){
%>
<html>
<head>
<style>
td {font:9pt}
</style>
</head>
<body>
<table align=center width="95%" border="0" cellspacing="1" cellpadding="3"
bgcolor="#000000">
  <tr bgcolor="#999999" align="center">
    <td width="6%" align="center" >编号</td>
    <td width="24%" align="center">发件人</td>
    <td width="30%" align="center">标题</td>
    <td width="40%" align="center">发送日期</td>
```

```
    </tr>
    <%
        //显示邮件信息，因为邮件的 uid 用 get 方式传递，所以要进行 URL 编码
        for(int i=0;i<msgs.length;i++){
    %>
    <tr bgcolor="#FFFFFF">
        <td align=center><%=i+1%></td>
        <td><%=msgs[i][1]%></td>
        <td><a href='viewmail.jsp?uid=<%=URLEncoder.encode(msgs[i][0],"iso-8859-1")
%>'><%=msgs[i][2]%></a></td>
        <td><%=msgs[i][3]%></td>
    </tr>
    <%
            }
        }
    }catch(Exception e){
        out.println(e);
    }
    %>
    </table>
    </body>
    </html>
```

MailList.java 的代码如下所示：

```
package mail;
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.event.*;
import javax.mail.internet.*;
import javax.activation.*;

public class MailList {
    String protocol="pop3"; //使用 POP3 协议
    String mbox = "INBOX";  //读取收件箱的邮件（POP3 只能读取收件箱的邮件）
    int totalMessages=0;          //邮件总量
    //得到邮件的总数
    public int getTotalMessages(){
        return totalMessages;
    }
   public String[][] list(String host,int port,String user,String pass) {
        try{
            // 获得系统属性对象
            Properties props = System.getProperties();

            //获得会话
            Session session = Session.getDefaultInstance(props, null);
            //获得邮箱对象
            Store store = session.getStore(protocol);
            // 连接 POP3 邮箱
            store.connect(host, port, user, pass);
```

```
// 打开收件箱
Folder folder = store.getDefaultFolder();
folder = folder.getFolder(mbox);

//以只读方式打开邮件夹
folder.open(Folder.READ_ONLY);
//得到总邮件数
totalMessages = folder.getMessageCount();
//初始化二维数组，以便保存邮件信息
String[][] messages=new String[totalMessages][4];
//得到所有邮件
Message[] msgs = folder.getMessages();

//使用一个合适的获取邮件的方法
FetchProfile fp = new FetchProfile();
fp.add(FetchProfile.Item.ENVELOPE);
folder.fetch(msgs, fp);
//把文件夹转型为 POP3Folder，以便读取邮件的 UID
com.sun.mail.pop3.POP3Folder pf =(com.sun.mail.pop3.POP3Folder)folder;
Address[] a ;
InternetAddress ia;

byte[] bytes;
//读取邮件信息
for (int i = 0; i < msgs.length; i++) {
    //得到邮件的惟一标识
    String uid = pf.getUID(msgs[i]);
    messages[i][0]=uid;
    //得到邮件的发件人
    if ((a= msgs[i].getFrom()) != null) {
        ia=(InternetAddress)a[0];
        messages[i][1]=ia.getAddress();
    }
    //得到邮件的标题
    messages[i][2]=msgs[i].getSubject();
    if(messages[i][2]==null||messages[i][2].equals(""))
        messages[i][2]="没有邮件标题";
    bytes=messages[i][2].getBytes("iso-8859-1");
    for(int j=0;j<bytes.length;j++){
        if(bytes[j]<0){
            messages[i][2]=new String(bytes,"gb2312");
            break;
        }
    }
    //得到邮件的发送日期
    messages[i][3]=msgs[i].getSentDate().toString();
}
//关闭收件箱
folder.close(false);
//关闭邮箱
store.close();
return messages;
```

```
            } catch (Exception ex) {
                System.out.println("Oops, got exception! " + ex.getMessage());
                ex.printStackTrace();
                return new String[0][0];
            }
        }
}
```

16.6.4　运行结果

在 maillist.htm 中填写数据，以收取本地邮箱中的信件，如图 16-19 所示。

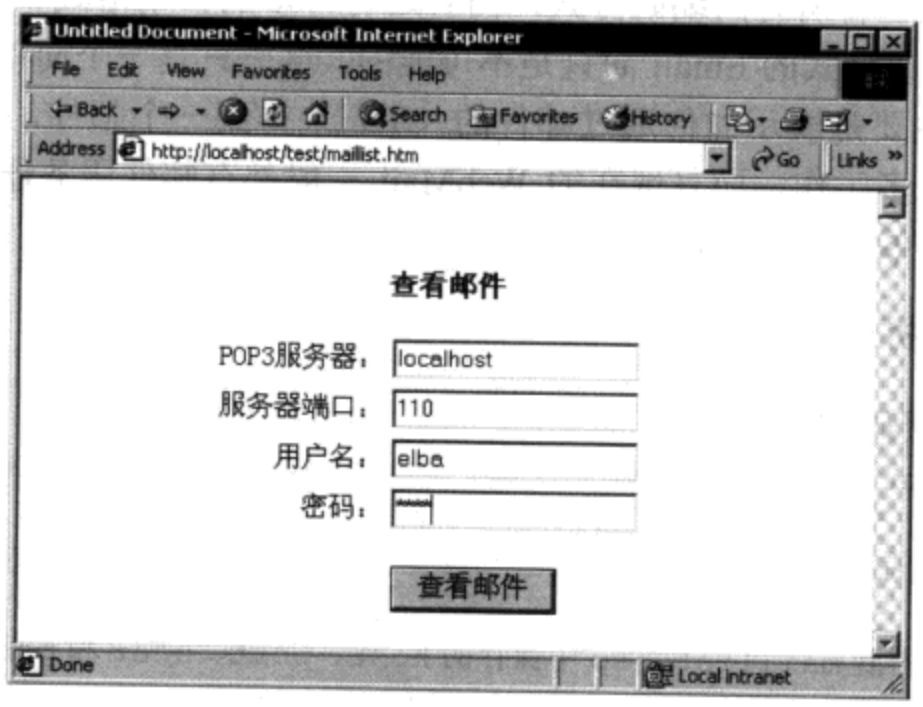


图 16-19　maillist.htm 的页面

单击“查看邮件”按钮后，由 maillist.jsp 显示邮件列表，如图 16-20 所示。

图 16-20　显示邮件列表的页面

16.7　用 JavaMail 接收单封邮件

16.7.1　实例说明

这个例子是前边接收邮件列表的延续。前边的例子只解析邮件头，可以得到邮件的发送

者、邮件标题和发送时间。这个例子就是要解析单封邮件，前边的例子中已经为这里做了铺垫。如把 POP3 服务器域名、端口、用户名、密码放到一个 session 里，这里就可以直接使用 session 里的值。前边的例子还得到了邮件的惟一标识 UID，这个例子就是通过邮件的 UID 来查收邮件。

16.7.2 编程思路

编写邮件程序最难的就是解析邮件，解析邮件为什么难呢？因为邮件格式不统一。前边已经提到过邮件编码的问题，如果要发送邮件，只要选择一种编码方式就可以了。但是如果要接收邮件，必须要包含所有的编码方式，而且还有许多的日期格式、boundary 格式等。总之，要做到能解析所有格式的 Email 简直是不可能的。这里只是一个简单的例子，可以解析纯文本格式的邮件，主要是演示如何通过邮件的 UID 来得到一个邮件，如果要真正编写一个 WebMail 还需要做很多工作，而且现在的 WebMail 一般都有邮件服务器的支持，这样做编写 WebMail 程序就会简单许多而且效率要高。

16.7.3 代码分析

viewmail.jsp 的代码如下所示：

```
<%@ page contentType="text/HTML;charset=GB2312" import="java.net.*,java.io.*,
java.util.*"%>
<jsp:useBean id="mailBean" scope="page" class="mail.ViewMail"/>
<%
//从 session 中得到 maillist.jsp 中保存的 host、port、user 和 pass 的值
String host=(String)session.getValue("host");
int port=((Integer)session.getValue("port")).intValue();
String user=(String)session.getValue("user");
String pass=(String)session.getValue("pass");
String uid=request.getParameter("uid");

String[] msgs={"","","",""};
try{
    //调用 getMail 函数得到邮件的信息
    msgs=mailBean.getMail(host,port,user,pass,uid);
}catch(Exception e){
    e.printStackTrace();
}
%>
<table align=center width="85%" border="0" cellspacing="1" cellpadding="3"
bgcolor="#000000">
    <tr bgcolor=#ffffff>
        <td align=right>发件人：</td>
        <td><%=msgs[0]%></td>
    </tr>
    <tr bgcolor=#ffffff>
        <td align=right>收件人：</td>
        <td><%=msgs[1]%></td>
    </tr>
```

```
    <tr bgcolor=#ffffff>
        <td align=right>邮件标题：</td>
        <td><%=msgs[2]%></td>
    </tr>
    <tr bgcolor=#ffffff>
        <td align=right valign=top>邮件正文：</td>
        <td>
        <pre>
        <%=msgs[3]%>
        <pre></td>
    </tr>
</table>
```

ViewMail.java 的代码如下所示：

```
package mail;
import java.util.*;
import java.io.*;
import javax.mail.*;
import javax.mail.event.*;
import javax.mail.internet.*;
import javax.activation.*;

public class ViewMail {

    String protocol="pop3";
    String mbox = "INBOX";
        int totalMessages=0;
        int newMessages=0;
       public   String[]   getMail(String   host,int   port,String   user,String
pass,String uid){
            try{
                // 得到系统属性
                Properties props = System.getProperties();
                //得到会话
                Session session = Session.getDefaultInstance(props, null);
                //获得一个邮箱
                Store store = session.getStore(protocol);
                //建立一个到邮箱的连接
                store.connect(host, port, user, pass);
                //打开收件夹
                Folder folder = store.getDefaultFolder();
                folder = folder.getFolder(mbox);
                //以只读方式打开邮件夹
                folder.open(Folder.READ_ONLY);
                String[] messages=new String[4];
                //得到邮件
                Message[] msgs = folder.getMessages();

                //设置获得邮件的方式
                FetchProfile fp = new FetchProfile();
                fp.add(FetchProfile.Item.ENVELOPE);
```

```
//获取邮件
folder.fetch(msgs, fp);
//把文件夹转型为 POP3Folder，以便读取邮件的 UID
com.sun.mail.pop3.POP3Folder pf =(com.sun.mail.pop3.POP3Folder)folder;
Address[] a ;

byte[] bytes;
StringBuffer sendto=new StringBuffer();
for (int i = 0; i < msgs.length; i++) {
    //比较邮件的 UID，以得到需要的邮件
    if(pf.getUID(msgs[i]).equals(uid)){
        if ((a= msgs[i].getFrom()) != null) {
            messages[0]=a[0].toString();
        }
        //如果有多个接收者，则以分号隔开
        if ((a = msgs[i].getRecipients(Message.RecipientType.TO)) !=
        null) {
            for (int j = 0; j < a.length; j++){
                    sendto.append((j==0?"":"; ")+a[j].toString());
            }
        }
        messages[1]=getGB(sendto.toString());
        messages[2]=getGB(msgs[i].getSubject());
        //只解析纯文本邮件
        if (msgs[i].isMimeType("text/plain") ||
msgs[i].isMimeType("text/html")) {
                    messages[3]=getGB(msgs[i].getContent().toString());
                }else{
                    messages[3]="不能解析邮件";
                }
            }
        }
        //关闭邮件夹
        folder.close(false);
        //关闭邮箱
        store.close();
        return messages;
    } catch (Exception ex) {
        ex.printStackTrace();
        return new String[0];
    }
}
 //转换为国标码的函数
 public static String getGB(String stri){
    try{
        byte[] bytes=stri.getBytes("iso-8859-1");
        for(int i=0;i<bytes.length;i++)
            if(bytes[i]<0)
                return new String(bytes,"gb2312");
        return stri;
```

```
            }catch(Exception e){
                return stri;
            }
        }
}
```

16.7.4　运行结果

图 16-21 是查看一封邮件的结果。

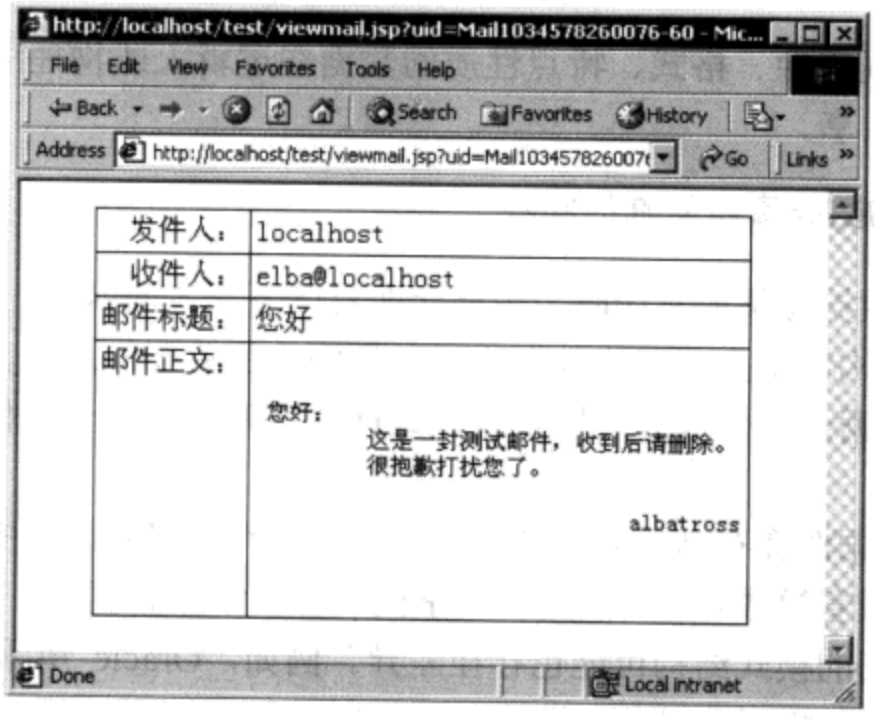


图 16-21　查看邮件的结果页面

16.8　小结

本章主要介绍了如何使用 JavaMail 来处理电子邮件。电子邮件协议主要包括 SMTP 协议、POP3 协议、IMAP4 协议和 MIME 协议，详细了解这些协议的内容请参阅相关的 RFC 文档。

本章重点：

- 使用 SMTP 协议发送邮件。
- 使用 SMTP 认证服务。
- 发送 HTML 邮件和带附件的邮件。
- 使用 POP3 协议接收邮件。
- 简单邮件体的编码解码。

第 17 章　数据集成中间件

17.1　数据集成概述

数据集成是把不同来源、格式、特点性质的数据在逻辑上或物理上有机地集中，从而为企业提供全面的数据共享。

17.1.1　集成异构数据库所面临的问题

在构建企业异构数据源集成系统时，主要会面对以下几方面问题：

（1）异构性。异构性是企业异构数据集成必须面临的首要问题，其主要表现在两方面：

1）系统异构，数据源所依赖的应用系统、数据库管理系统乃至操作系统之间的不同构成了系统异构。

2）模式异构，数据源在存储模式上的不同。一般的存储模式包括关系模式、对象模式、对象关系模式和文档嵌套模式等几种，其中关系模式为主流存储模式。需要注意的是，即便是同一类存储模式，它们的模式结构可能也存在差异。例如，Oracle 所采用的数据类型与 SQL Server 所采用的数据类型并不是完全一致的。

（2）完整性。异构数据源数据集成的目的是为应用提供统一的访问支持。为了满足各种应用处理（包括发布）数据的条件，集成后的数据必须保证一定的完整性，包括数据完整性和约束完整性两方面。

数据完整性是指完整提取数据本身，一般来说，这一点比较容易达到。

约束完整性，约束是指数据与数据之间的关联关系，是惟一表征数据间逻辑的特征。保证约束的完整性是良好的数据发布和交换的前提，可以方便数据处理过程，提高效率。

（3）性能。网络时代的应用对传统数据集成方法提出了挑战，提出了更高的标准。一般说来，当前负责集成的应用必须满足轻量快速部署，就是系统可以快速适应数据源改变和低投入的特性。

（4）语义冲突。信息资源之间存在着语义上的区别。这些语义上的不同可能引起各种矛盾，从简单的名字语义冲突（不同的名字代表相同的概念）到复杂的结构语义冲突（不同的模型表达同样的信息）。语义冲突会带来数据集成结果的冗余，干扰数据处理、发布和交换，所以如何尽量减少语义冲突也是数据集成的一个研究热点。

（5）权限瓶颈。由于数据库资源可能归属不同的单位，所以如何在访问异构数据源数据基础上保障原有数据库的权限不被侵犯，实现对原有数据源访问权限的隔离和控制，就成为连接异构数据资源库必须解决的问题。本书将该问题定义为权限瓶颈问题。

（6）附加约束。集成两个或多个数据源的时候，数据源的数据之间可能存在着某种联系，例如，保存在不同资源库中的关于同一套工装的信息之间存在着一定的逻辑联系，那么，把这种逻辑联系附加到集成结果中的过程就称为附加约束。

（7）集成内容限定。多个数据源之间的数据集成，并不是要将所有的数据进行集成，那么如何定义要集成的范围，就构成了集成内容的限定问题。

上面列举了在构建企业异构数据源集成系统时所必须面对的几个主要问题，其中，异构性、完整性、性能、语义冲突问题为异构数据集成中的共性问题，权限瓶颈、附加约束和集成内容的限定则属于企业异构数据集成的特性问题。值得指出的是，这些问题是相互联系、相互制约的，不应该简单地孤立对待。

17.1.2　数据集成模型分类

在企业数据集成领域，已经有了很多成熟的框架可以利用。目前通常采用联邦式、基于中间件模型和数据仓库等方法来构造集成的系统，这些技术在不同的着重点和应用上解决数据共享和为企业提供决策支持。下面将对这几种数据集成模型做一个基本的分析：

（1）联邦式数据库系统（Federated Distributed Database System），这种分布式数据库的特点是结点自治和没有全局数据模式，每个结点所看到的数据模式仅仅限于此结点所用到的数据。它一般由两部分组成：一个是本结点的数据模式，另一个是供本结点共享的其他结点上有关的数据模式。结点间的数据共享由双边协商确定。如果一个新的结点要加入系统，它开始时可以先用本结点的数据，然后与有关结点协商，共享其他结点的有关数据；本结点的数据也可以供其他结点共享。这种扩展是渐进的，不会影响原有系统的运行。由于每个结点所看到的数据模式不一样，好像系统中有多个逻辑数据库，这种分布式数据库因而在逻辑上也是分布的。由于没有全局数据模式，一个结点的数据模型的修改甚至一个结点的加入或撤离，仅仅影响有关结点。一个结点在给数据对象命名时，只要在本结点的数据模式内惟一就可以，不必考虑与其他无关数据对象的重名问题。每个结点好像拥有一个满足自己需要的集中式数据库一样，而不受制于全局数据模式，甚至不必有“全局概念”。结点的自治性很高。

（2）中间件模式通过统一的全局数据模型来访问异构的数据库、遗留系统、Web 资源等。中间件位于异构数据源系统（数据层）和应用程序（应用层）之间，向下协调各数据源系统，向上为访问集成数据的应用提供统一数据模式和数据访问的通用接口。各数据源的应用仍然完成它们的任务，中间件系统则主要集中为异构数据源提供一个高层次检索服务。

（3）数据仓库是在企业管理和决策中面向主题的、集成的、与时间相关的和不可修改的数据集合。其中，数据被归类为广义的、功能上独立的、没有重叠的主题。

这几种方法在一定程度上解决了应用之间的数据共享和互通的问题，但也存在以下的异同：

- 联邦数据库系统主要面向多个数据库系统的集成，其中数据源有可能要映射到每一个数据模式，当集成的系统很大时，对实际开发将带来巨大的困难。
- 中间件模式是目前比较流行的数据集成方法，它通过在中间层提供一个统一的数据逻辑视图来隐藏底层的数据细节，使得用户可以把集成数据源看为一个统一的整体。这种模型下的关键问题是如何构造这个逻辑视图并使得不同数据源之间能映射到这个中间层。
- 数据仓库技术则在另外一个层面上表达数据之间的共享，它主要是为了针对企业某个应用领域提出的一种数据集成方法，也就是前面所提到的面向主题并为企业提供数据挖掘和决策支持的系统。

17.2 基于 Web 服务的数据集成中间件设计与实现

下面通过具体的实例介绍一个基于 Web 服务的数据集成中间件设计与实现。通过提供所有异构数据源的虚拟视图来集成它们，这里的数据源可以是数据库、遗留系统、Web 数据源等。该系统提供给用户一个全局模式（也叫 Mediated 模式），用户提交的查询是针对该模式的，所以用户不必知道数据源的位置、模式及访问方法。

这种体系结构与联邦式数据库系统有如下区别：

- 可以集成非数据库数据源。
- 数据源可以不支持 SQL 查询。
- 数据源是完全自治的，这就意味着很容易向系统中添加/删除数据源。
- 由于系统中的数据源是自治的，所以对系统中数据源的访问通常是只读的，而联邦式数据库系统支持读/写访问。

该系统的体系结构如图 17-1 所示，该系统的主要部分是中介器（Mediator）和针对每个数据源的包装器（Wrapper）。这里中介器的功能是接收针对全局模式生成的查询，根据数据源描述信息及映射规则将接收的查询分解成每个数据源的子查询，再将子查询发送到每个数据源的包装器。包装器将这些子查询翻译成符合每个数据源模型和模式的查询，并把查询结果返回给中介器。中介器将接收的所有数据源的结果合并成一个结果返回给用户，如图 17-2 所示。

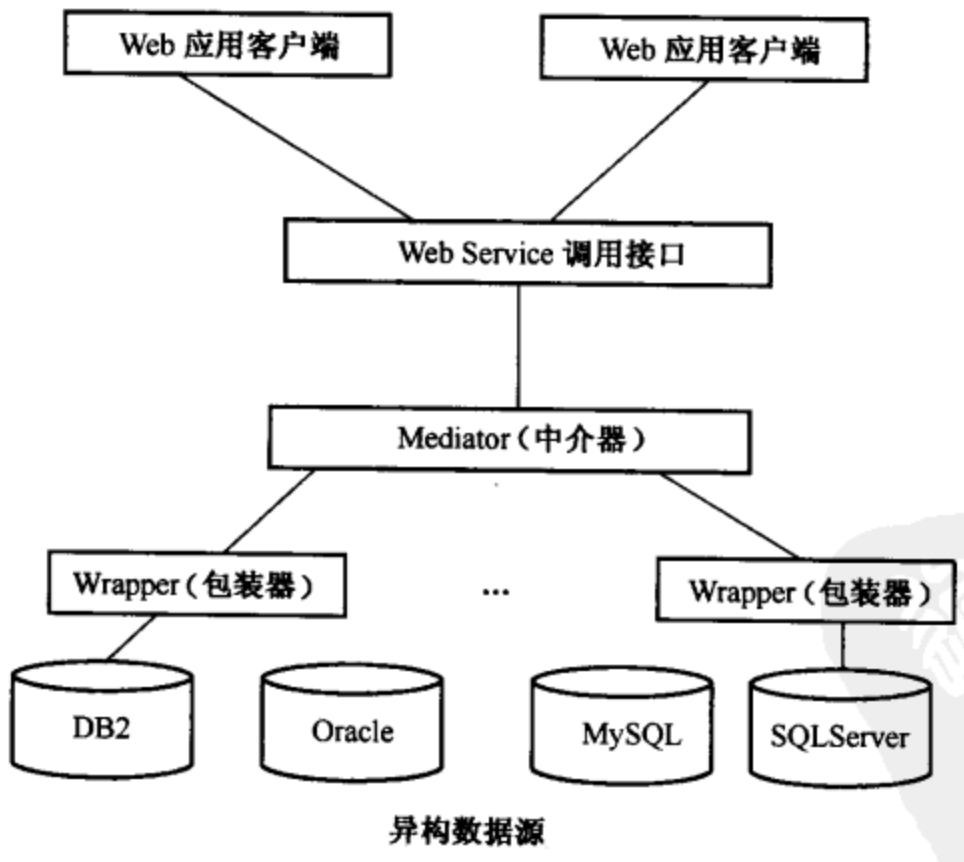


图 17-1 系统体系结构图

在 Web Services 的框架下使用一组 Web Services 协议构建信息集成系统。为中间件的数据访问接口创建一个 Web Service，然后使用 WSDL 向服务中心注册。客户端首先向注册中心发送查找请求，然后通过 SOAP 协议调用数据访问接口，从这些数据源获取数据。这种方法

具有完好封装、松散耦合、规范协议和高度的集成能力等特性。

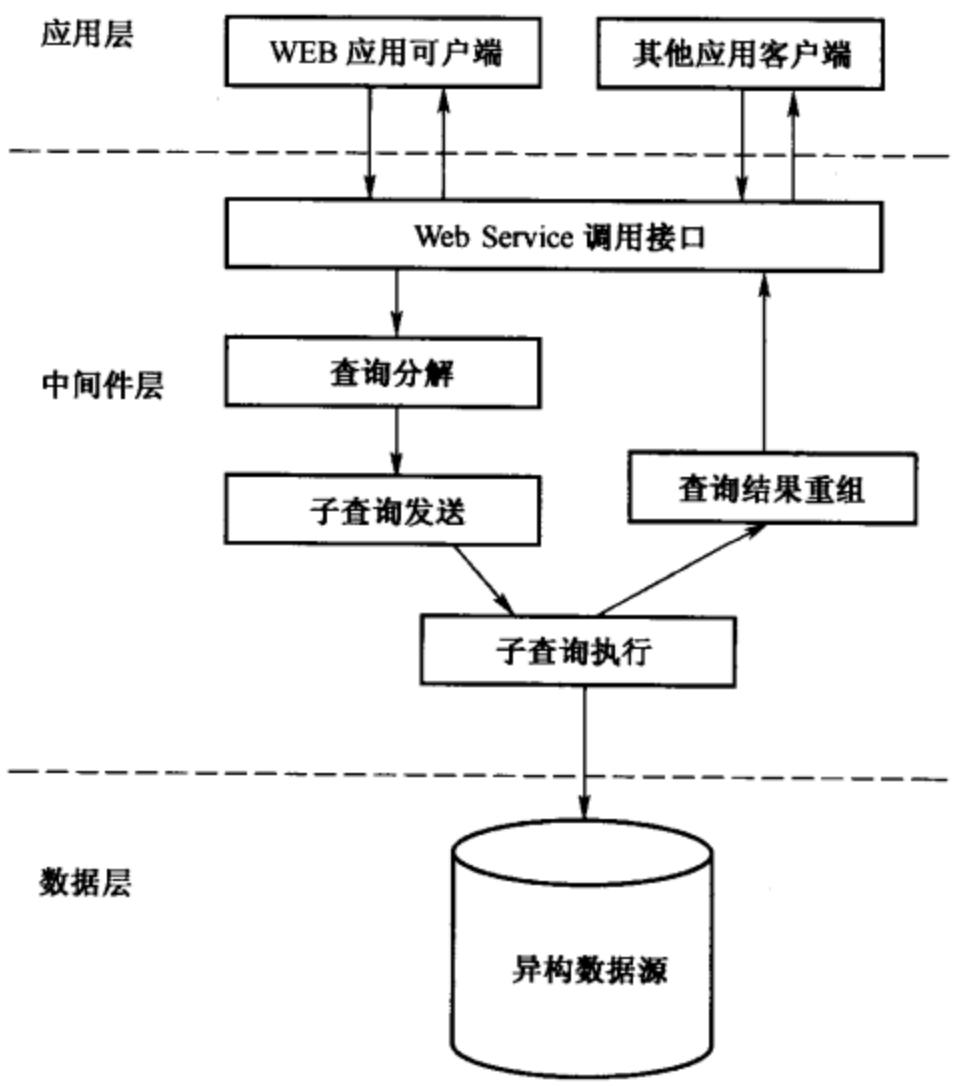


图 17-2　系统流程

该系统有五个核心模块：

（1）中间件配置模块：配置中间件和各个异构数据库的连接信息，包括数据库地址、登录用户名、密码、数据库名。

（2）查询分解模块：根据数据源描述信息及映射规则将接收的查询分解成每个数据源的子查询。

（3）子查询发送模块：分析用户提交的查询语句，从中找出该查询需要的数据源，然后到中间件的配置信息中查找相应的包装器，把连接信息和子查询发送到对应的包装器。

（4）子查询执行模块：子查询模块根据不同的数据库类型，加载对应的 JDBC 驱动程序，实现对多个不同数据库的查询操作。

（5）查询结果重组模块：查询结果重组模块收集子查询的结果，这些结果是以 DataSet 的形式存在的，然后合并为一个大的 DataSet 作为最终的结果。在返回结果给客户端时，根据 DataSet 生成一个结果 XML 文件给客户端，完成整个查询流程。

17.2.1　查询分解

为实现查询的高效分解，为中间件定义了新的查询语法如下：

（1）用关键字 DEFINE，以数据表为单位定义查询源数据单元，并以[]作为数据单元的分割符。

例：DEFINE [wrapper1.db1.table1=a] [wrapper2.db2.table2=b]

解释：包装器 wrapper1 上的数据库 db1 中的数据表 table1 为变量 a；包装器 wrapper2 上的数据库 db2 中的数据表 table2 为变量 b。

（2）用关键字 SELECT 指定结果集单元，以[]作为单元分割符。

例：SELECT[a.*][b.id]

（3）用关键字 FROM 指定结果集来自的数据单元，并以[]作为数据单元的分割符。

例：FROM [a][b]

（4）用关键字 WHERE AND OR 指定查询条件，并以[]作为条件单元的分割符。

例：WHERE [a.id=b.id]AND[a.age>b.age]

（5）用关键字 ADDITION 指定数据单元使用的属性/数据库函数/存储过程，并以[]作为单元分割符。

例：ADDITION [a.id=GROUP BY][b.age=ORDER BY DESC][a.age=DISTINCT]

[a.age=MAX]

含义：查询过程中以 a.id 进行分组；以 b.age 进行降序排列结果；结果中 a.age 惟一对 a.age 求和等。

用户提交查询请求后，查询分解模块依照上面的规则对查询请求的有效性、语法进行检查。检查成功则开始分解。查询分解按照以下原则进行：

（1）独立查询分解：以所需连接包装器为单位，将查询分解为一组子查询，每一组子查询都对应一个单独的数据源。

（2）依照相关性进行多库查询分解：两个包装器数据需要进行匹配的查询首先分解为对其中单一包装器的独立子查询，取出该独立子查询的结果数据生成匹配另一个包装器数据的子查询，从而实现两个包装器所连接数据需要匹配的查询操作。

分解实例：

```
DEFINE[server1.db1.table1=a][server2.db2.table2=b][server3.db3.table3=c]
SELECT [a.*][b.*]
FROM[a][b]
WHERE[a.begintime>'2004-9-8'][a.id>c.id][a.id=b.id] ADDITION[a.di=DISTINCT]
```

依据分解原则分解过程如下：

（1）分解出独立子查询，并记为新的查询单元变量：

```
DEFINE[SELECT[a.*]FROM[a]WHERE[a.begintime>'2004-9-8']
ADDITION[a.di=DISTINCT] = d]
```

（2）在独立子查询的基础上进行多库查询分解，每次查询记为新的查询单元变量：

```
DEFINE[SELECT[d.*]FROM[d]WHERE[d.id>c.id] = e]
DEFINE[SELECT[e.*][b.*]FROM[e][b]WHERE[e.id=b.id] = f]
```

查询结束，f 记录查询结果。

17.2.2 跨数据库的查询

假设有一家全国性银行在江西、北京和湖北都有分支机构，每个分支都有自己的数据库，利用这个基于 Web 服务的异构数据库集成中间件，综合该银行各分支的数据信息提供查询服务。描述如下：

（1）江西分行、湖北分行和北京分行分别安装包装器，进行本地配置，分别连接到自身的综合业务信息数据库，同时配置和中介器的连接，从而构成一个银行综合业务信息共享网络。

（2）客户通过中间件提供的 Web 服务接口，提交查询，查询经过分解，路由到相应的包装器，执行，最终返回结果给客户端。

查询实例：用户查询 2006-9-7 从江西和湖北汇往北京的汇款，且从江西汇出的汇款比从湖北汇出的汇款到达时间早。

客户端提交的查询如下：

```
Define [wrapper1.db1.info=a][ wrapper2.db2.infomation=b]
Select [a.*][b.*] from [a][b]
Where [a.destination=北京] And [a.begintime=2006-9-7]
And [b.destination=北京] And [b.begintime=2006-9-7]
And [a.endtime<B.ENDTIME]< p>
```

说明：wrapper1.db1.info=a　江西分行包装器所连接的数据

wrapper2.db2.infomation=b　湖北分行包装器所连接的数据

该查询被分解为：

（1）

```
Define [ Select a.* from [wrapper1.db1.info=a]
Where [a.destination=北京] And [a.begintime=2006-9-7]
=c]
```

（2）

```
Define [Select b.* from [wrapper2.db2.info=b]
Where [b.destination=北京] And [b.begintime=2006-9-7]
=d]
```

（3）取 1 中的数据和 2 中的数据进行匹配子查询：

```
Select [c.*][d.*] from[c][d] where [c.endtime]
```

整个查询结束。

17.3　OGSA-DAI 介绍

OGSA-DAI，即开放网格服务架构数据访问和集成（Open Grid Services Architecture-Data Access and Integration），此项目是由 DTI 资助的 e-Science 核心项目，合作伙伴有英国 e-Science 中心、IBM、苏格兰的 EPCC（爱丁堡并行计算中心）、Oracle 等。本节介绍 OGSA-DAI 的内部机制，解释了这些机制之间以及与用户应用程序之间的交互方式。本节使用的是来自 Globus Project 的 OGSA 参考实现，即 Globus Toolkit 3（GT3）。GT3 是基于 Java 的网格应用程序框架，它提供了一种环境，供网格应用程序注册其服务，维护其状态，并与其他应用程序进行通信。

在很多项目中，人们都需要一个中间件层来访问大量的并且基本上是静态的数据库，这就催生了 OGSA-DAI。OGSA-DAI 的结构是一种工具箱，它具有一些扩展点，可供开发人员扩展其功能，以适应自身的特定需求。数据库管理系统便是一个很好的例子。目前支持的有关系数据库（通过 JDBC）和 XML 数据库（通过 XMLDB）。应用程序的开发人员也可以开发他们自己的数据库支持。

OGSA-DAI 是一种中间件，其设计目标是提供一种简便的方法，在网格环境中实现数据的访问和集成。OGSA-DAI 符合基于 OGSA 的网格标准，并在 Globus Toolkit 3.0 上进行开发。支持 DB2、Oracle、Xindice、MySql 等数据库管理系统。

网格数据库是对现有数据库的网格化，基于开放网格服务体系结构提供网格数据库服务，使网格用户或其他网格服务可通过网格数据库服务访问网格中的各种异构数据库，从而达到数据资源的高度共享和协同处理，对数据资源的访问更加透明、高效、可靠，网格数据处理的能力更强，更好地满足更广泛虚拟组织的数据处理需求。

OGSA-DAI 项目致力于建造通过网格访问和集成来自不同的孤立数据源的中间件。这个项目是由 UK Database Task Force 提出构想，并紧密地和全球网格论坛数据访问和集成服务工作组（GGF DAIS-WG）以及 Globus 团队一起工作。总体上，OGSA-DAI 与 DAIS 相符合，它也努力成为 DAIS 网格数据库服务推荐标准的第一个参考实现。

OGSA-DAI 的目标是通过网格进行数据访问和集成提供统一的服务接口。通过 OGSA-DAI 的接口，不同的、异构的数据源和资源被视为逻辑上的单一资源。更重要的是，它还允许这些资源在 OGSA 的框架内进行集成。OGSA-DAI 网格服务提供基本的操作来完成复杂的操作，比如数据联邦、在虚拟组织进行分布式查询，但是它隐藏了如数据库驱动、数据格式和从客户端的传输机制等技术细节。

OGSA-DAI 项目的第一阶段已于 2003 年 7 月结束，第二阶段的研究开发始于 2003 年 10 月，并且已经发布了 OGSA-DAI WSRF 2.2 和 OGSA-DAI WSI 2.2 版本。

DAIS 工作组于 2002 年 2 月的 GGF4 会议上成立，它寻求促进与 OGSA 相适应的网格数据库服务的标准，初衷是提供对现有的、自主管理的数据库的一致访问，而不寻求开发一个新的数据存储系统，更准确地说，是要使这些系统在网格框架内更易于个别地或共同地使用。DAIS 将于 GGF9 上提出一个网格数据库服务的初始版，今后也将支持更广定义的数据库的访问和集成，例如：文件系统以及来自仪器和设备的数据流。

目前使用 OGSA-DAI 的项目有：AstroGrid、Biogrid、BioSimGrid、Bridges、FirstDIG、GeneGrid、ODD-Genes 和 OGSA-WebDB。

17.3.1 OGSA-DAI 的体系结构

OGSA-DAI 的体系结构如图 17-3 所示。

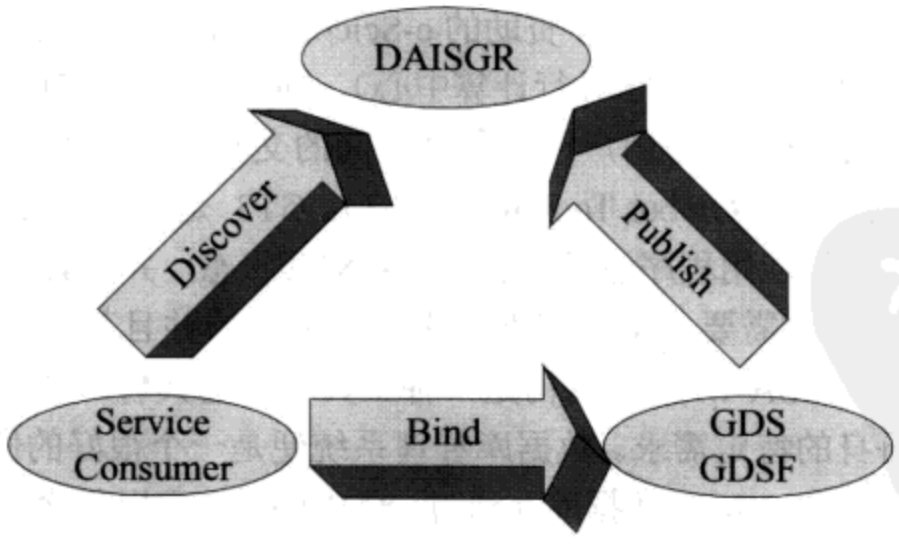


图 17-3 OGSA-DAI 的体系结构

从该图可以看出，其体系结构与 Web Service 很相似，都是 Discover（发现）、Bind（绑定）和 Publish（发布）机制。

下面对 OGSA-DAI 中使用的术语进行简单介绍。

网格数据服务（Grid Data Service，GDS）：通过这项服务可以访问某个数据资源（关系数据库或 XML 数据库，甚至是存储在普通文件中的数据）。

网格数据服务工厂（Grid Data Service　Factory，GDSF）：这项服务用于创建一个 GDS 实例来访问特定的数据资源。

服务组注册器（Service Group Registry，DAISGR）：这项服务用于找到所需要的 GDS，也可以通过它找到用于创建所需 GDS 的工厂。

执行文档（Perform Document）：一种 XML 格式的文档，用于定义要在 GDS 上执行的活动，如一条 SQL 查询，然后再定义如何将查询的结果传送给第三方。

响应文档（Response Document）：一种 XML 格式的文档，是 GDS 处理执行文档后返回的结果。

活动（Activity 或 Activities）：实现程序功能的核心模块。

它们之间的交互关系如图 17-4 所示。

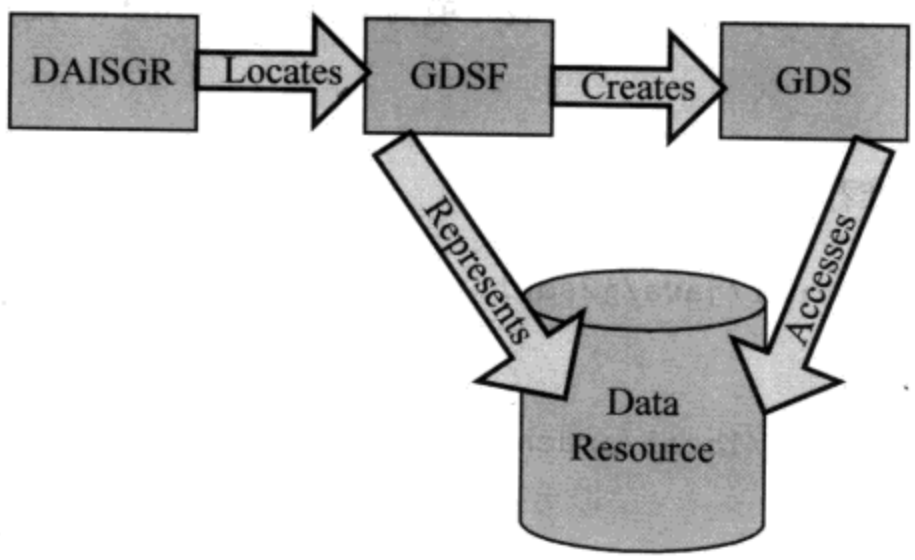


图 17-4　元素间的交互关系

整个交互过程可以分为以下八个步骤，如图 17-5 所示。

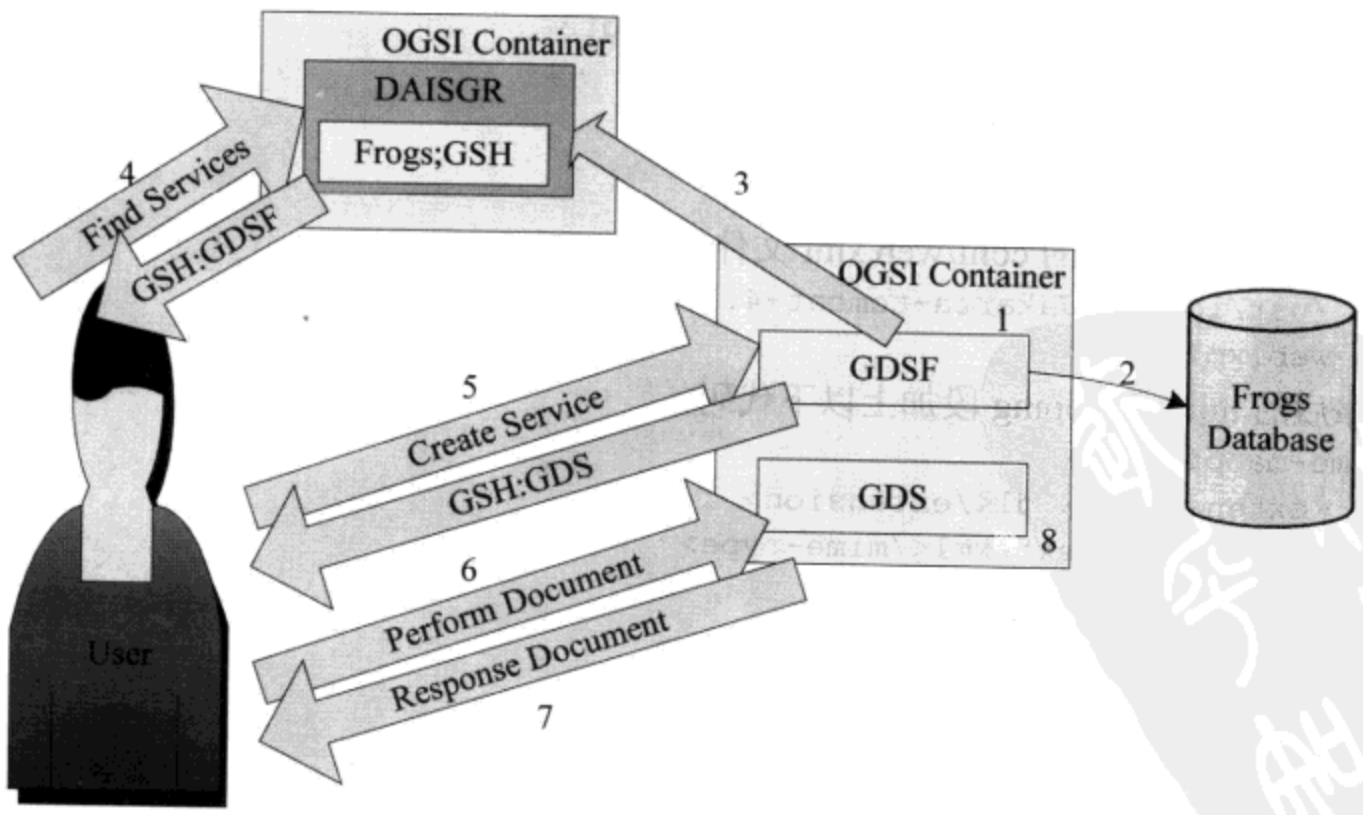


图 17-5　整个交互流程

（1）运行 OGSI 容器（OGSI container）为永久性服务。

（2）在此时 GDSF 代表 database：Frogs Database。

（3）GDSF 在 DAISGR 上注册。

（4）如果用户想了解数据库，可以直接查询 GDSF，也可以通过 DAISGR 定位合乎需要的 GDSF。

（5）用户请求创建一个 GDS。

（6）用户发送执行文档（Perform Document）和 GDS 通信进行交互。

（7）GDS 返回一个响应文档（Response Document）。

（8）用户销毁 GDS 或者让其自动消亡。

17.3.2 安装配置

本节介绍 OGSA-DAI 的安装配置。安装环境为 RedHat 9 Linux；建立用户 ogsadai，用此身份完成安装和部署。

安装步骤为：

（1）安装 JDK，推荐 1.3 以上版本。

```
$vi /etc/profile
$export JAVA_HOME=/usr/java/j2sdk1.4.1
```

（2）安装 ANT。

```
$vi /etc/profile
$export ANT_HOME=/usr/local/apache-ant-1.5.2
```

（3）安装 Tomcat。

```
$vi /etc/profile
$export CATALINA_HOME=/usr/local/Jakarta-tomcat-4.1.27
$export PATH=$PATH:$JAVA_HOME/bin:$ANT_HOME/bin
```

（4）安装 GT3 Core。

```
$tar -xvzf ogsa-3.2.tar.gz /usr/local/gt3
$ant setup
```

出现 Build Successful，即表示成功。

（5）把 GT3 部署到 Tomcat。

首先编辑 tomcat 下的 conf/web.xml 文件：

```
$cd /usr/local/ Jakarta-tomcat-4.1.27/conf
$vi web.xml
```

在靠后的 mime-mapping 段加上以下代码：

```
<mime-mapping>
    <extensioin>wsdl</extension>
    <mime-type>text/xml</mime-type>
</mime-mapping>
<mime-mapping>
    <extensioin>xsdl</extension>
    <mime-type>text/xml</mime-type>
</mime-mapping>
<mime-mapping>
    <extensioin>gwsdl</extension>
    <mime-type>text/xml</mime-type>
```

```
</mime-mapping>
```

保存退出后，开始部署：

```
$cd /usr/local/gt3
$ant -Dtomcat.dir=$CATALINA_HOME deployTomcat
```

出现 Build Successful，即表示成功。

（6）测试 GT3 部署。

```
$cd /usr/local/ Jakarta-tomcat-4.1.27
$./startup.sh
```

使用上述命令启动 tomcat，在浏览器中访问：http://localhost:8080/ogsa/services。如果能看到一个服务列表，说明部署成功。

（7）安装 OGSA-DAI。

```
$cd /usr/local/ogsadai
$tar -xvzf ogsadai-3.1-src.tar.gz
```

把所需要的类库文件复制到/usr/local/ogsadai/lib 下，然后运行：

```
$cd /usr/local/ogsadai
$ant deploy
```

（8）测试 DAI 是否部署成功。

```
http://localhost:8080/ogsa/services
```

服务列表中应该包括：

```
A Grid Data Service Factory   : ogsadai/GridDataServiceFactory
A DAI Service Group Registry : ogsadai/DAIServiceGroupRegistry
```

访问 http://localhost:8080/ogsa/service/ogsadai/GridDataServiceFactory?wsdl 应该看到一个 wsdl 文件，描述 GridDataServiceFactory。

访问 http://localhost:8080/ogsa/service/ogsadai/DAIServiceGroupRegistry?wsdl 应该看到一个 wsdl 文件，描述 DAIServiceGroupRegistry。

必须了解在设置工厂时进行的主要配置步骤，这里将工厂配置为实现下面的功能：

1）用特定的网格数据服务组注册器对其自身进行注册。

2）将自身连接到底层的数据资源上，其中包括下面的内容：

- 可用的活动、模式及实现类。
- 资源的元数据，可以有两种形式：静态（写在配置文件中）或通过实现类 MetaData-Extractor 获取。
- 驱动信息，其中包括数据资源的实现类，JDBC 驱动（或与之等价的对象）以及资源的 URI。

一旦 OGSA-DAI 启动，工厂就随之启动；然后，用注册器将其注册，并能通过配置文件中的静态信息和配置文件提供的 MetaDataExtractor 类访问到服务数据。工厂还将根据数据资源创建实现对象。然后，客户机确定在注册器中列出的众多工厂中应该使用哪一个。一旦选定合适的工厂，客户机就请求工厂创建一个 GDS 实例，以访问特定的数据资源。现在，GDS 已经准备好了，可以接收执行文档，运行数据库查询，传输查询结果和传送数据。

17.3.3　引擎架构

设定引擎（又叫做引擎）是 OGSA-DAI 的核心。它将执行文档发送到 GDS 上时

OGSA-DAI 中发生的每一件事情都编排在一起。首先介绍这个引擎做了哪些工作，然后再深入探讨其工作原理。

当 GDS 创建出来时，系统根据一些上下文信息和 GDS 的配置（最初是从工厂的配置中来的）将引擎实例化。这个上下文信息包括：

（1）活动及其对应的 XML 模式与实现类的清单。

（2）角色映射清单和角色映射的详细信息。

（3）数据资源实现与数据资源有关的细节信息。

当 GDS 接收到一份执行文件时，它将这份文档与一些上下文信息（如从用户凭证中获取的 DN）一起传送到引擎中。然后，引擎对执行文档依次进行如下处理：

（1）解析并验证执行文档。

（2）识别执行文档中指定的活动。

（3）创建这些活动的实现实例。

（4）创建这些活动之间管道的实例。

（5）创建活动处理器的实例并启动。

（6）用处理器处理这些活动，将产生的数据通过管道传递。

（7）将输出信息结合在一起，形成一份响应文档。

（8）将这份响应文档返回给 GDS，由后者返回客户机。

（9）在执行文档运行过程当中，可以随时通过 OGSI 的服务数据获取活动的状态。

这里介绍一下特异名称（Distinguished name，DN）。特异名称（DN）是一种符号，用于在 X509 凭证中惟一标识某个特定用户。典型的 DN 由用户名、组织名和国名组成，还可能包含一些附加信息。如/c=UK/o=IBM/ou=HS&T/l=Winchester/cn=Neil Hardman。

其中：

c=UK：国名；

o=IBM：组织名；

ou=HS&T：组织单位；

l=Winchester：位置；

cn=Neil Hardman：一般的姓名；

这个引擎可以一次处理一份执行文档。如果在一份执行文档运行期间又提交了另一份，那么就会返回一个 UserException。这并不会妨碍从一个工厂创建多个 GDS 实例，其中的每一个 GDS 实例都具有它自己的引擎，因此可以同时运行多份执行文档。

如果画一张图来表示，那么得到的就是如图 17-6 所示的一些类。

后面研究执行文档的详细运行过程。

17.3.4 设置

当执行文档发送到 GDS 上之后，GDS 将其传给引擎，引擎再用引擎上下文中的活动模式对执行文档中的内容进行验证。然后，用执行文档中的相关数据和上下文中的信息创建活动实例（例如，sqlQueryStatement 活动需要角色映射器以及数据资源的实现信息来连接数据库，并执行查询）。引擎负责计算出每一个活动的输入和输出信息应该如何链接在一起，然后在这些活动之间设置默认的管道。

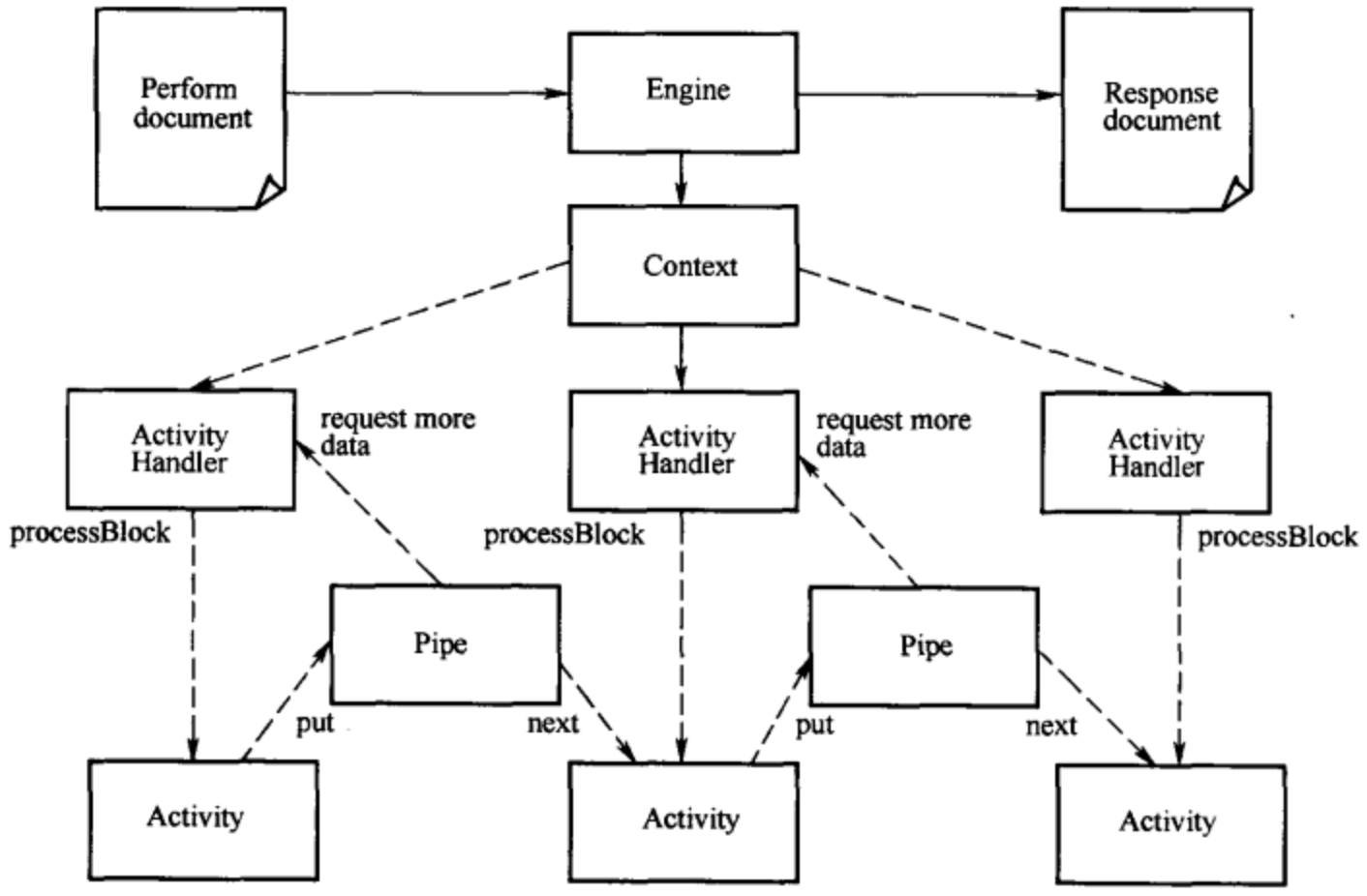


图 17-6　引擎架构

一旦出现未链接的活动输出，引擎就创建 deliverToResponse 活动，这样，默认的方式就是由活动所产生的任何数据都同时返回到响应文档中。下一步，引擎创建活动处理器，负责管理活动的调用方式，最终，上下文信息被提供给活动，这样就可以选择非默认的活动处理器或管道（如 inputStream 活动使用的是 SynchronizedGrowablePipe ）。

17.3.5　同步或异步

所有的设置完成之后，数据就可以开始流动了。如果引擎没有使用任何一个 deliverToResponse 活动（例如，如果执行文档定义所有的数据都通过 deliverToURL 活动传递到第三方，如某台 FTP 服务器），那么引擎就会启动一个新的线程，异步执行这些活动，然后将响应文档返回客户机。在这种情况下，响应文档仅仅包含一些简单的信息，用来指示执行文档就要开始执行了；否则，执行过程会立即开始，而且直到所有的活动都将其状态设置为 Complete 或出错时，才会返回一个响应。

在大多数情况下，与 OGSA-DAI 一起提供的几乎所有活动都用 String 对象作为其输出类型。

17.3.6　数据流动

数据在系统中的流动是通过一系列活动处理器进行协调的，每一个活动处理器管理数据链中的一个活动。这些活动本身通过管道链接在一起，可以提供活动内的数据通路。除了第一个和最后一个之外的每一个活动，都具有一个输入管道和一个输出管道。输出传递活动（如默认的 deliverToResponse，以及显式使用的 deliverToURL 活动，或是其他类似活动）负责通过使用一个特殊的处理器（即 RunAheadHandler ）创建数据流的实例。下面是数据流的情况：

（1）输出活动处理器调用其活动的 processBlock 方法。

（2）活动调用附加在其上的输入管道上的 next 方法，提供一个供处理的数据块。

（3）如果在输入管道中没有数据块，那么该管道将调用给它提供数据的那个活动的处理器。

（4）处理器再次调用它所管理的活动的 processBlock 方法。对于活动链中的每一个活动，都重复步骤 2、3、4。

（5）这个过程一直持续到提供最初数据源的那个活动上，这个活动可能是数据库查询的结果，也可能是要从另一个系统传送数据过来。

（6）当活动已经处理过，或是获取了一定的数据，它就会将这些数据块用 put 方法放置在其输出管道中。

最终，所有的数据都流过了整个系统，每一个活动（由最初的数据源开始启动）都将调用其附加的输出管道上的 close 方法。当后续活动试图从该管道调用数据时，将抛出 NoSuchElementException 异常，这就意味着已经没有数据了。

这里对“数据块”做一个介绍。在 OGSA-DAI 中，基本的数据单元就是块（block）。块是一些单独的 Java 对象（或是单独的对象或基本类型构成的数组）。也就是说，活动可以将任何 Java 对象放置在其输出中，供后续活动消费，根据数据流中前面活动的情况，也可以接收一系列的输入块。在活动之间不存在类型检查，这就意味着活动需要对它们从输入中接收到的数据进行检查，看是否是预期的类型；否则，将抛出 ClassCastException。

17.3.7 错误处理

如果在这个过程中产生一些错误，那么失败活动的状态就被置为 Error（既可以在活动中显式设置，也可以通过处理器中实现的异常处理机制进行设置）。引擎会监视所有活动的状态，如果它发现了出错状态，就会中断处理过程，然后将状态和所有的错误消息都返回到响应文档中。如果活动是异步执行的，那么过程将停止，客户机可以通过查询 GDS 的引擎状态（已经发布为 OGSI 服务数据）来发现这个错误。

17.3.8 清除

如果出现了错误，或是所有的活动状态都设置为 Complete，这时引擎就知道处理过程已经结束，可以将其清除了。如果系统使用了 deliverToResponse 活动，那么就生成响应文档，发送回客户机。然后，系统将所有的数据丢弃，这样引擎就可以处理到达的下一个执行文档了。

17.3.9 代码实例

代码 1 中的实例处理一份发送到 GDS 的执行文档，这份执行文档用 sqlQueryStatement 活动对数据库进行查询。查询的结果将用 WebRowSet XML 格式表示。接下来，对那个 WebRowSet XML 文档应用一个 XSLT 样式表，用 xslTransform 活动将查询结果转换成一组由逗号分割开的变量。最后，转换的结果通过默认的 deliverToResponse 活动传递回客户机，因为 xslTransform 活动的输出（本例中的名字叫做 transformedData）没有链接到任何其他活动上。

代码 1. 执行文档实例。

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<gridDataServicePerform xmlns="....">
  <!-- sqlQueryStatement Activity performs the query -->
  <<span class="boldcode">sqlQueryStatement</span> name="statement">
    <expression>
      select * from littleblackbook where id=100
    </expression>
    <webRowSetStream name="<span class="boldcode">statementOutput</span>"/>
  </sqlQueryStatement>
  <!-- xslTransform Activity transforms the results of the query using the In-Line
       XSLT -->
  <<span class="boldcode">xslTransform</span> name="transform">
    <inputXSLT>
      <!-- This is an XSL-Transform that transforms the results
           of an SQL query into comma separated variables -->
      <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
        <xsl:output method="text" indent="yes"/>
        <xsl:template match="/">
          <xsl:for-each select="RowSet/data/row/col">
            <xsl:value-of select="."/> ,
          </xsl:for-each>
        </xsl:template>
      </xsl:stylesheet>
    </inputXSLT>
    <inputXML from="<span class="boldcode">statementOutput</span>"/>
    <output name="<span class="boldcode">transformedData</span>"/>
  </xslTransform>
</gridDataServicePerform>
```

当引擎接收到这份文档，并验证完毕之后，它将两个经过命名的活动及与之关联的处理器实例化。然后在活动之间创建管道。这时，引擎发现 sqlQueryStatement 活动的输出链接到 xslTransform 活动（名为 statementOutput）的输入。引擎还意识到 xslTransform 活动具有悬空的输出（名为 transformedData），因此就实例化一个 deliverToResponse 活动及其处理器和管道。

因为 deliverToResponse 是一个输出活动，因此与之相关的处理器设置为 RunAheadHandler。其他处理器关联的是默认的 CallThroughPipe 和 SimpleHandler。RunAheadHandler 负责启动整个过程，数据流从 sqlQueryStatement 活动所查询的数据库经过 xslTransform 活动，进入 deliverToResponse 活动，在这里，数据块被合并到响应文档中，返回客户机。如代码 2 所示。

代码 2. 响应文档实例。

```
<gridDataServiceResponse xmlns="http://ogsadai.org.uk/namespaces/2003/07/
      gds/types">
  <result name="statement" status="COMPLETE"/>
  <result name="transform" status="COMPLETE"/>
  <result name="transformedData" status="COMPLETE">
    <![CDATA[101,
           Andrew Borley ,
           754 Palansuriya Crescent,
           Edinburgh ,
```

```
            0915084840 , ]]>
  </result>
</gridDataServiceResponse>
```

为了重用前面画的那幅图，现在的架构看起来应该如图 17-7 所示。

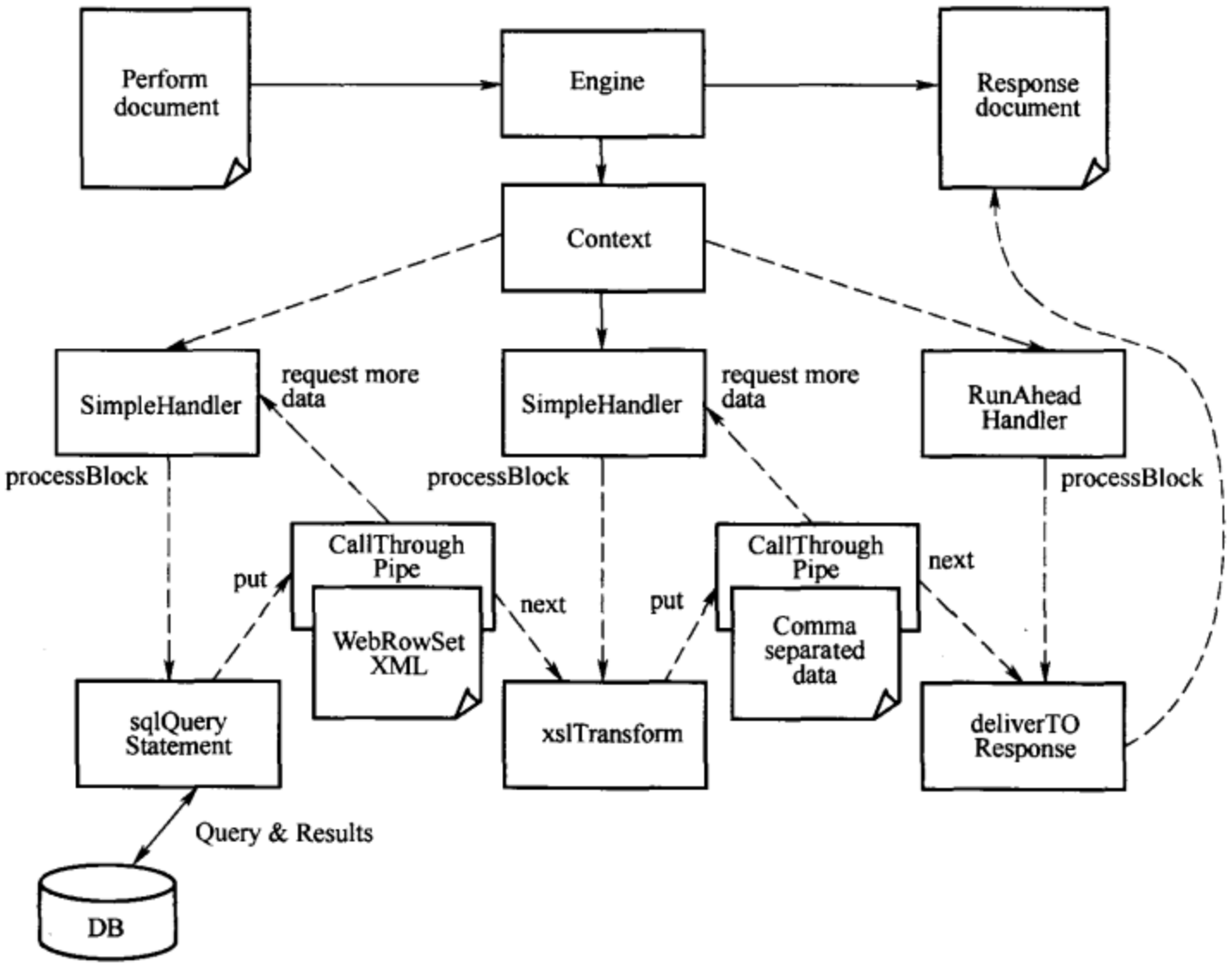


图 17-7　上例中的引擎架构

17.3.10　数据库访问

无论从哪个角度看，OGSA-DAI 的访问方式都应该和在网格环境之外访问数据库的方式尽可能相似。在这样的方式下，可以看到将数据库加入网格所带来的所有好处，也几乎没有花费什么代价来重新编写代码，更没有重新构造整个解决方案的架构。

现在已经有了一个执行文档，并将其发送到引擎中，引擎构造了一串活动，并用管道将这些活动连接起来。那么，这些活动该如何与 DBMS 交互来访问数据库中的数据呢？

无论使用的查询语言是 XPath、Xupdate，还是 SQL，所使用的类集合以及这些类之间的关系都比较相似。在这里只考虑通过 JDBC 连接 DBMS 执行 SQL 的情况，但是也会指出扩展点。通过学习这里的例子，也可以编写出在 XMLDB 上用 XPath、XUpdate 访问 XML DBMS 的代码，并知道如何编写自己的活动，以便与后端的数据资源进行交互。

这里要讨论的类如下所示：

（1）包 uk.org.ogsadai.porttype.gds.activity.sql

- SQLActivity

- RelationalResourceManagementActivity
- SQLBulkLoadRowSetActivity
- SQLQueryStatementActivity
- SQLStoredProcedureActivity
- SQLUpdateStatementActivity

（2）包 uk.org.ogsadai.porttype.gds.dataresource

- DatabaseMetaDataExtractor
- DataResourceImplementation
- JDBCDataResource
- MetaDataExtractor
- SimpleJDBCDataResourceImplementation
- SimpleJDBCMetaDataExtractor

GDS 是专门为访问特定的数据资源设计的，这种数据资源从物理上讲存在相互独立的任意 GDS。因此这就意味着代表数据资源的对象应该与工厂紧密相关。在与特定数据交互的过程中，会存在一个对象，用来在每个工厂的基础上管理或缓存连接。GDS 并不是孤立地管理它与数据库的交互，与此相反，每一个 GDS 都使用工厂类提供的 DataResourceImplementation 来管理连接。同一个工厂生成的多个访问同一数据资源的 GDS 都用同一个 DataResourceImplementation 对象来处理连接。

所有这一切，意味上面那些类可以划分为三组：

（1）DataResourceImplementation 类，负责处理 JDBC 连接、驱动程序、角色映射以及访问 DBMS。

（2）Activity 类，负责接受 SQL，并用 JDBC 连接将这个 SQL 传递给 DBMS，然后对返回的结果进行格式化。

（3）MetaDataExtractor 类，发布与 DataResource 有关的服务数据，以便在注册器或工厂服务中选择正确的数据资源。

17.3.11　DataResourceImplementation 类

这一部分介绍 DataResourceImplementation 类，如图 17-8 所示。

从图 17-8 中可以看到 JDBCDataResource 类的 getDriverClass 方法。在开发过程中发现，各种不同的数据库驱动中都存在一些很有意思的属性，于是创建了一个方法，称为 getDriverClass，用于找到驱动的类，并允许正确切换驱动的各种行为。

DataResourceImplementation 类掌管连接的处理和角色映射。前面提到，它们在配置文件中与数据资源相匹配，并为希望自己编写 DataResourceImplementation 的程序员提供扩展点。活动类应该仅仅依赖相关接口中的方法。

如果想要创建 MyPoolingJDBCDataResourceImplementation，那么有两种方法可供选择：

（1）可以从已经写好的 SimpleJDBCDataResourceImplementation 类继承，并覆盖 getConnection 和 returnConnection 方法。

（2）可以从抽象的 DataResourceImplementation 类继承，通过实现其中的 JDBCDataResource 接口来创建自己的类。重要的是 JDBCDataResource 接口，SQL 活动正是用这

个接口获取并返回数据库连接的。

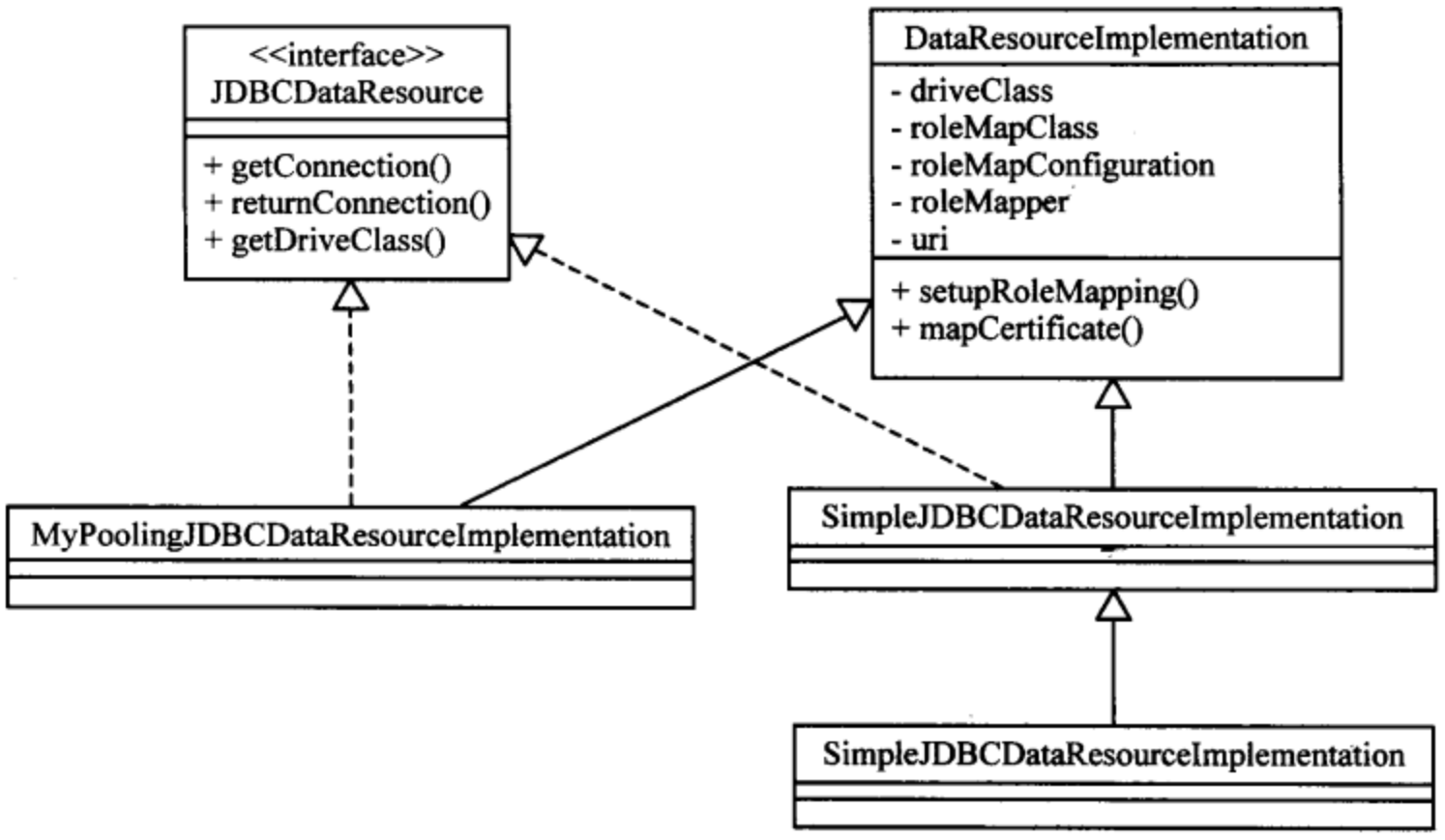


图 17-8 展示 DataResourceImplementations 的 UML 模型

一旦编写好这个新类，剩下的惟一问题就是替换 OGSA-DAI 配置文件，让某个数据资源使用这个 MyPoolingJDBCDataResourceImplementation。

对关系型访问而言，SQLActivity（下面会谈到）将通过 JDBCDataResource 的 getConnection 方法获取一个 JDBC 连接。实现的方法是将凭证作为参数传递。DataResourceImplementation 类提供的 mapCertificate 方法可以用于将这些凭证映射为数据库角色（根据角色映射器获取用户名和口令）。现在已经用 mapCertificate 方法获得了用户 ID 和口令，也从配置文件中获得了 URI，必要的条件都已经具备，可以创建 JDBC 连接了。

目前，编写自己的 XMLDB DataResourceImplementation 只有一种方法，就是对 SimpleXMLDataResourceImplementation 进行扩展，因为还没有定义 XMLDBResource 接口。

17.3.12 Activity 类

所有其他的 SQL 活动都要扩展抽象的 SQLActivity 类，如图 17-9 所示。这个抽象类中包含受保护的方法，可以用这些方法从活动中获取输入和输出信息。在这样的环境中，输入可以是值（由执行文档提供），也可以是引用（从其他活动中流入），目前还可以作为 SQL 参数、SQL 表达式以及存储过程名称等的响应。输出当然是 ResultSet，根据活动的类型不同，也可能是被更新的记录数目。

允许从流输入数据，特别是允许从同一个流获得多个输入，这种情况可能导致无法预计的结果。流中的值出现的顺序变得很重要，因为活动读取流中数据的顺序要与输入数据在活动元素中的声明顺序保持一致。

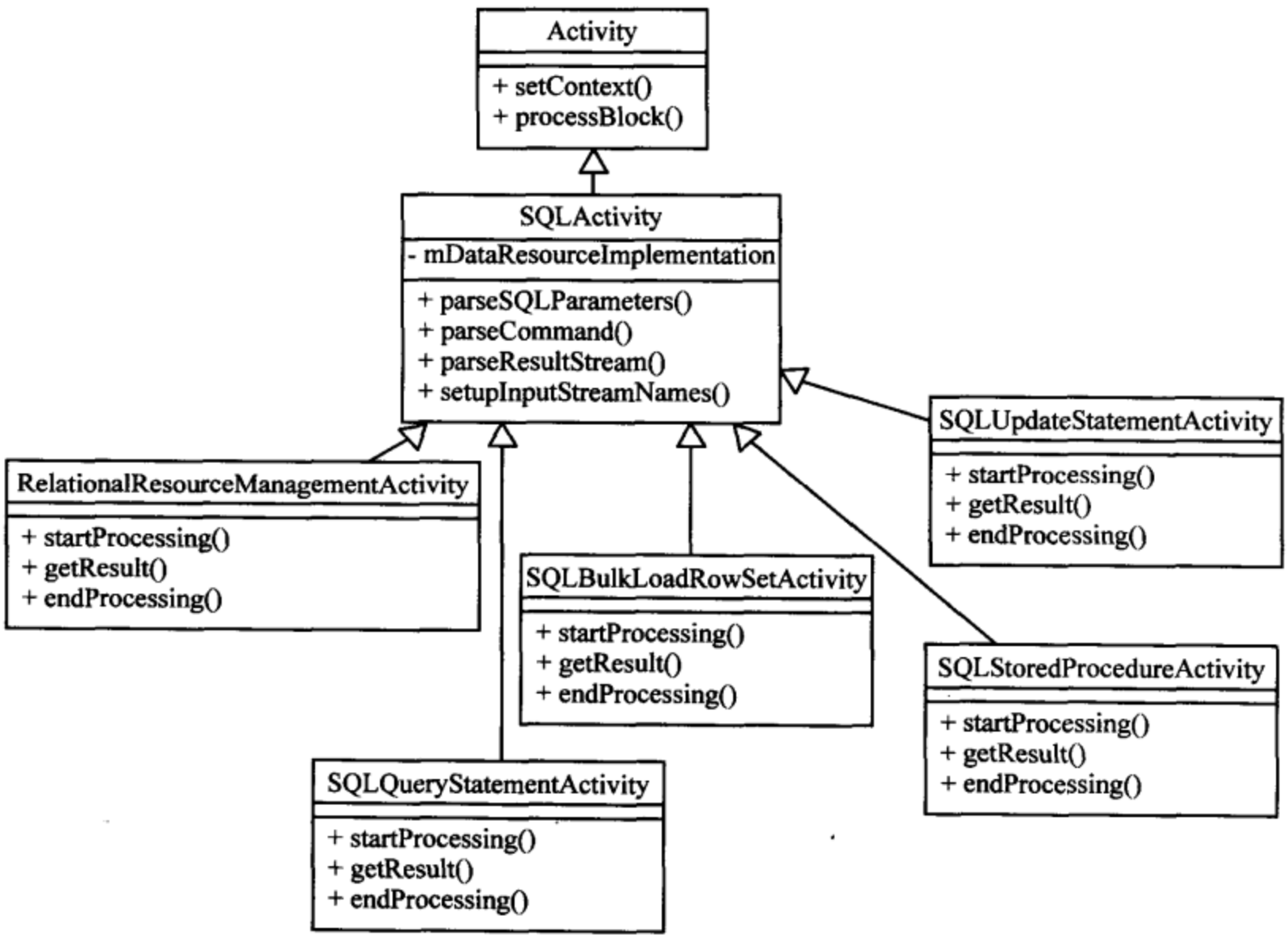


图 17-9　展示 SQL 活动的 UML 模型

在元素顺序的问题上有一种例外情况，即命令（SQL 表达式或存储过程名）。命令必须总是流中的第一个项。提出这样要求的原因有两点：

（1）SQL 活动元素只允许在 SQLParameters 之后和结果流之前声明命令。

（2）直到发现了命令之后，SQLParameters 才是有意义的，也才是有用的。

这些到底是什么意思呢？图 17-10 给出了解释。

结果就是，不必仅仅为了在 SQL 活动使用数据项之前改变它们在流中出现的顺序而设置一个转换程序。这样，如果使用的数据项在流中出现的顺序是参数 1、参数 3、参数 2、参数 1、参数 3、参数 2……，那么，只要保证在 SQL 活动中用相同的顺序声明 SQLParameters（即 1、3、2），就可以直接将这个流传递给 SQL 活动了。

设置好活动之后，就知道需要什么样的输入信息，也知道输出信息将去往何处。在整个处理过程中，都将实施跟踪，看连接是否保持打开状态。这种跟踪机制使用户能够在操作完成的时候将连接返回，并在适当的场合重新利用这个连接（如在 prepared statement 中）。

SQL 活动中的处理可以分为三个阶段：

（1）startProcessing。获得连接，设置下一条语句。根据活动的不同，连接可能会设置为只读。

（2）getResult。从输出流中获得下一个数据块。由于执行语句的结果可能是 ResultSet，这样就对应多个数据块，因此调用这个方法并不总会执行一条语句（第一次调用时肯定会执行

语句的）。

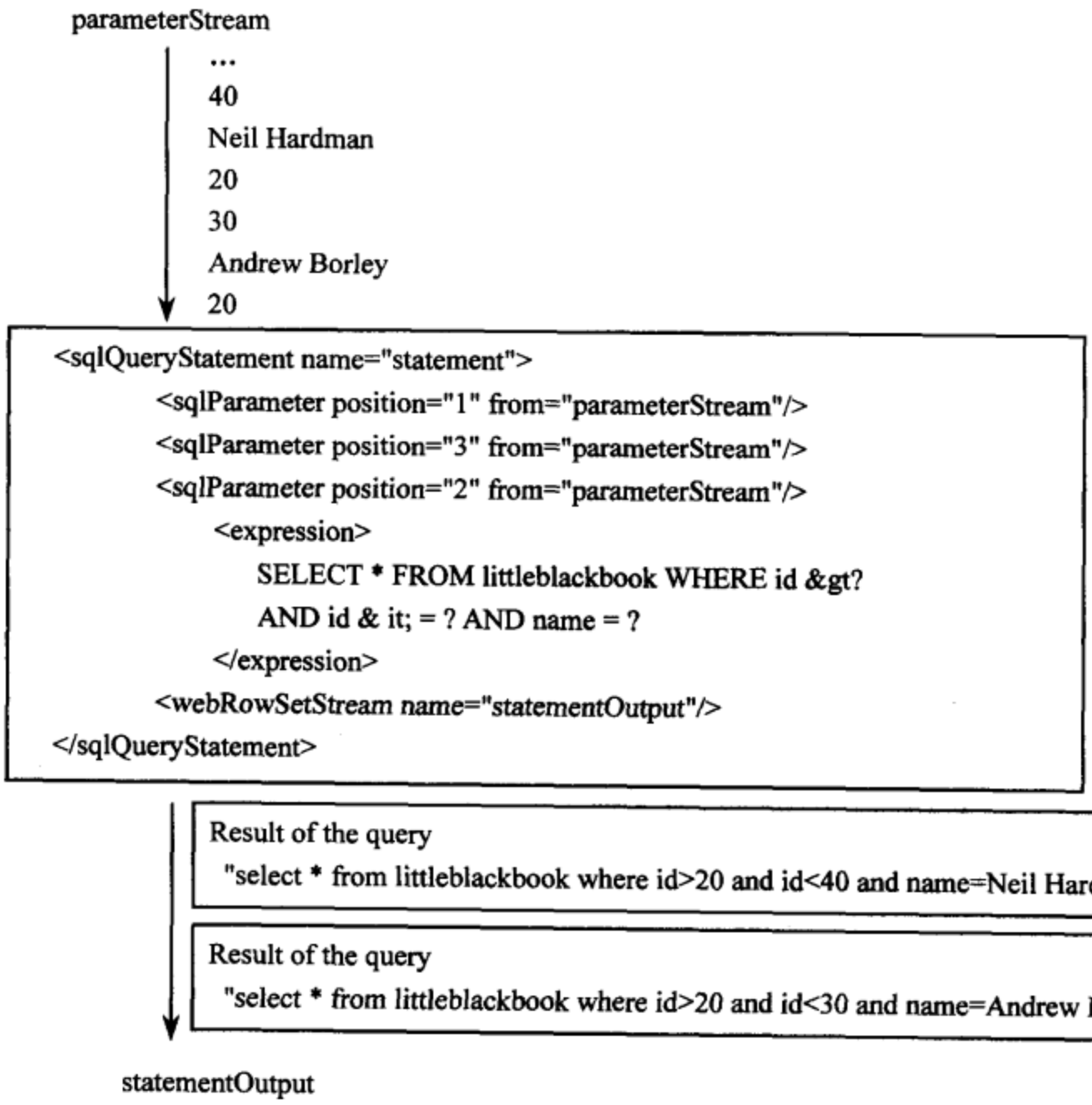


图 17-10　从单一流获得多项输入

（3）endProcessing。结束输出流并返回连接。但是，如果活动对应于 prepared statement，将选择保持连接，并允许重用该活动。

XMLDB 活动几乎与此相同，它们是通过扩展 XMLDBactivity 类以及一个扩展包（uk.org.ogsadai.porttype.gds.activity.xmldb.commands ）实现的。这个包中包含的代码可以使抽象的 XMLDBCommandActivity 变为可扩展的，并提供不同的 XMLDB 命令。xmlCollectionManagement 和 xmlResourceManagement 活动是从 XMLDBCommandActivity 继承得来。

用户可以开发新的活动来满足 Oracle 之类特定 DBMS 的需求。

17.3.13　MetaDataExtractor 类

用户该如何发现 GDS 的元数据呢？例如，用户该如何决定其中存在哪个表或字段呢？

假设理解了网格服务和 Globus Toolkit 3，因此，答案当然就是与 GDS 相关的服务数据。我们不想在 DBMS 之外保存与该数据库有关的字段的服务数据。因此，OGSA-DAI 使用了

GT3 的 ServiceDataValueCallBack 接口，这样就能根据需要通过 MetaDataExtractor 类来发现这些数据，如图 17-11 所示。

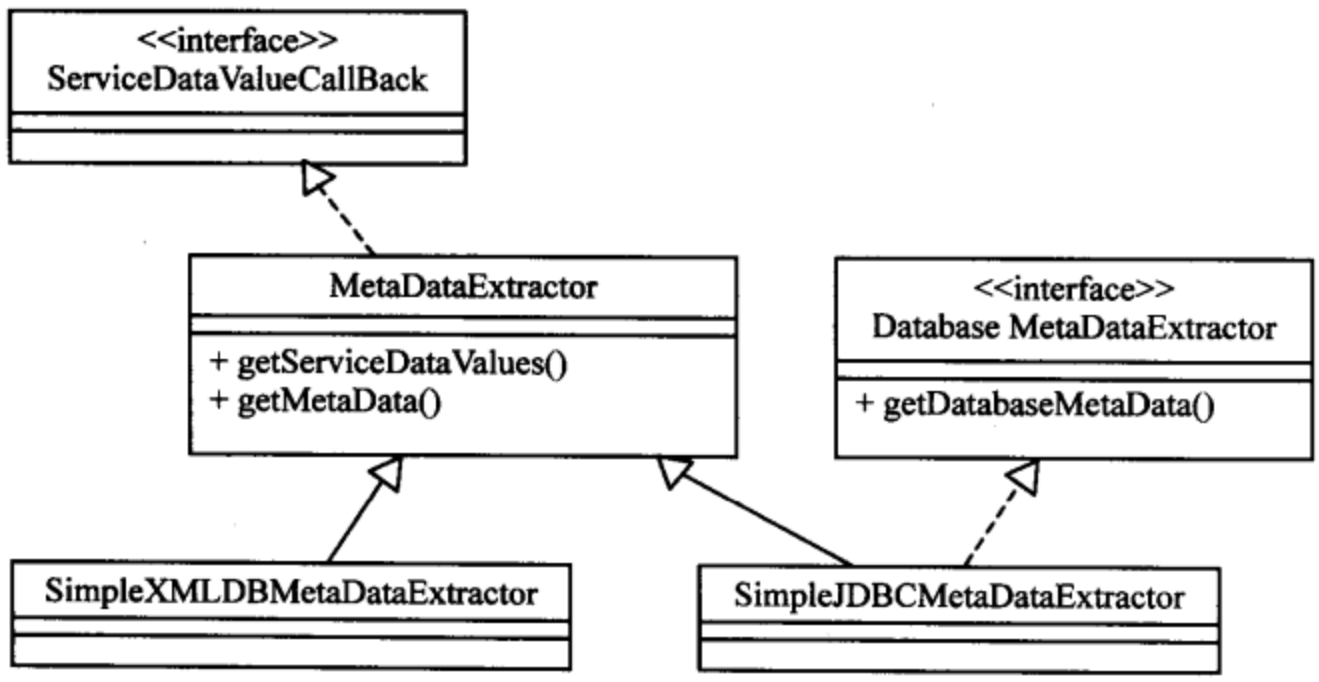


图 17-11　展示 MetaDataExtractor 的 UML 模型

17.3.14　开发实例

前面已经提到 Activity 在 OGSA-DAI 框架里实现核心的功能，扮演着重要的角色，包括三个方面：①Query Activities 直接面向数据源，从数据集里提取数据；②Transformation Activities 以某种方式转换数据，如 XSLT transformation activity、GZIP compression Activity、base64 encoding Activity 等；③Delivery Activities 把数据导入数据源或从数据源导出数据。

OGSA-DAI 提供了一个很好的体系框架，在这个框架下可以快速完成开发和部署。不必从头开始，只要专注于我们的业务逻辑，因此可以有效地缩短开发周期，提高开发效率。Activity 是这个框架中最基本也是最核心的组成单位。下面用一个例子来实现一个类似于 Java 类库中的 StringTokenizer 分解字符串的功能，以此来详细说明整个开发和部署流程。

1. 编写一个 Activity

只需要实现两个文件，一个是 Schema 文件，即 string_tokenizer.xsd；另外一个是 Java 类，即 StringTokenizerActivity.java。

修改一下 <CONTAINER>/webapps/Root/schema/ogsadai/activities 目录下的 grid_data_service_types.xsd 文件即可。

找到元素 MyActivityType，改为 StringTokenizerType，myActivity 改为 StringTokenizer。把 input 的名字改为 sourceString，把 output 的名字改为 tokens。

元素 configuration 为：

```
<xsd:element name="seperators" minOccurs="0" maxOccors="1">
    <xsd:complexType>
        <xsd:attribute name="value" use="required">
            <xsd:restriction base="xsd:string" />
        </xsd:attribute>
    </xsd:complexType>
</xsd:element>
```

同样，StringTokenizerActivity 类也是从 template 类模板生成的，只要修改其中的一些值，实现其中的一些方法即可。

把 activity 名改为 StringTokenizerActivity，把输入和输出改为与 Schema 相对应的值：

```
Stringinputname=((Element)element.getElementByTagName("sourceString").item
(0)).getAtrribute("from");
String  inputName=((Element)element.getElementByTagName("tokens").item(0)).
getAtrribute("name");
```

增加变量 mSeparators 存放用户指定的分隔符，变量 mTokenizer 存放 StringTokenizer。

```
NodeList separators=element.getElementByTagName("separators");
If(separators.getLength()==0)
{
    mSeparators=" ";
}
else
{
    Element separator=(Element)separators.item(0);
    mSeparators=separator.getAttribute("value");
}
```

实现最重要的 processBlock()方法：

```
while(mTokenizer==null || !mTokenizer.hasMoreTokens())
{
    String sourceString=(String)mInput.next();
    mTokenizer=new StringTokenizer(sourceString,mSeparators);
}
block=mTokenizer.nextToken();
mOutput.put(block);
```

最后实现 finalize，以防止执行的请求输出的结果还是以前请求所产生的：

```
mTokenizer=null;
```

2. 设置和部署

运行 javac - cp <ogsa>/lib/ogsadai.jar;<ogsa>/lib/xmlParserAPIs.jar StringTokenizer-Activity.java 编译 StringTokenizerActivity.java。

把 StringTokenizerActivity.class 放到<CONTAINER>/webapps/ogsa/WEB-INF/classes 目录下或者把这个类加到系统的 CLASSPATH 变量中。

把 string_tokenizer.xsd 放到<CONTAINER>/webapps/ogsa/schema/ogsadai/xsd/activities 目录下，最后在 GDSF 数据源配置文件中加上：

```
<activityMap name="StringTokenizer"
implementation="StringTokenizerActivity"
schemaFileName="http://<host>:<port>/schema/ogsadai/xsd/activities/string_
        tokenizer.xsd"/>
```

如果 Web 服务器是 Tomcat，则默认 host 为 localhost，port 为 8080。

17.4 小结

本章介绍了数据集成中间件的技术原理和开发实例，包括如下内容：

（1）介绍了数据集成的三种模型：联邦式数据库系统模型、中间件模型和数据仓库模型。

（2）介绍了一种基于 Web 服务的数据集成中间件的设计与实现思路。

（3）重点介绍了 OGSA-DAI 项目。OGSA-DAI 即开放网格服务架构数据访问和集成（Open Grid Services Architecture-Data Access and Integration）。OGSA-DAI 是一种中间件，其设计目标是提供一种简便的方法，在网格环境中实现数据的访问和集成。本章介绍了 OGSA-DAI 体系结构，以及如何通过 OGSA-DAI 访问和集成异构数据资源。

第 18 章　门户（Portal）中间件

18.1　门户系统概述

本节介绍门户系统的定义、功能和分类。请注意区分门户系统的概念和 Portal 技术。门户系统是应用层面的概念，Portal 技术是技术层面的概念。

18.1.1　门户系统定义

对于门户这个概念，目前有很多种说法，如门户系统、门户网站。门户是一个相对较新的领域，更多的是一种企业的应用，没有一个统一的学术的定义。但是根据目前的应用，可以从广义和狭义两个方面给出如下定义：

从广义上说，门户系统是一个应用框架，它将各种应用系统、数据资源和互联网资源集成到一个信息管理平台之上，并以统一的用户界面提供给用户。多用于企业，对于一个企业来说，门户可以使企业快速地建立企业对客户、企业对内部员工和企业对企业的信息通道，使企业能够发布存储在企业内部和外部的各种信息。

从狭义上来说，门户网站，是指通向某类综合性互联网信息资源并提供有关信息服务的应用系统。门户网站最初提供搜索引擎和网络接入服务，后来由于市场竞争日益激烈，门户网站不得不快速地拓展各种新的业务类型，希望通过门类众多的业务来吸引和留住互联网用户，以至于目前门户网站的业务包罗万象，成为网络世界的“百货商场”或“网络超市”。从现在的情况来看，门户网站主要提供新闻、搜索引擎、网络接入、聊天室、电子公告牌、免费邮箱、影音资讯、电子商务、网络社区、网络游戏、免费网页空间等。在我国，典型的门户网站有新浪网、网易和搜狐网。

18.1.2　门户系统功能

门户站点可以是水平的，具有许多不同类型的功能（“六个 C”），也可以垂直集中于一些领域，如某一类型的商务（如销售书籍的 Amazon. com）。它们可以是一个通道、一个目的站点，或者两者兼而有之。

“六个 C”概括了作为一个门户站点所应具备的基本功能，虽然不同的门户站点在各自的侧重点上有所不同，但最成功的门户站点努力体现所有这六个方面：

（1）连接（Connectivity）。越来越多的门户站点开始脱离以前只关注提供 Internet 接入或连接的业务模型而向前发展。目前，两种不同的业务模型正在出现：

- 高速接入：美国的 Excite@Home 和 RoadRunner 提供高速 Internet 接入。亚洲在这方面的例子包括 Pacific Convergence 公司，该公司主要从事泛亚洲的互联网宽带接入的网络建设。香港的 Wharf Cable 公司正计划利用现有的电缆基础设施来提供 Internet 接入。

- 低速 ISP：这类服务提供商由于它们有先行一步的领先优势和巨大的订户基础，因而在绝大多数市场占统治地位。这类例子包括英国的 Freeserve 公司。在亚洲，最广为人知的这类公司之一是 Pacific Internet 公司。

（2）导航（Context）：包括信息的搜索，将搜索到的数据安排到目录之中并提供导航工具。Yahoo 是靠提供这类服务起家的。在亚洲，香港地区的 Netvigator 公司、澳大利亚的 eCorp 公司（ninemsn.com）以及台湾地区的 Kimo 公司都提供上下文特性作为其门户站点的组成部分。在 Internet 不断发展以及每天都有海量的新信息出现在网络上的情况下，能够快速地在 Internet 上导航、绘制出 Internet 的路线图变得越来越重要。

（3）内容（Content）：这种功能是使门户站点本身也成为目的站点。在美国，这类站点的例子有 Sportline 和 Disney 公司的 Go 站点。已经开始提供内容的许多门户站点现在正加大提供内容服务的力度，例如，在澳大利亚，ninemsn.com（eCorp 与 Microsoft 各占一半资本的合资公司）使用由 eCorp 的父公司 Publishing and Broadcasting 有限公司（PBL）提供的内容，主要是使用 PBL 的杂志和电视台的内容。像 ninemsn 这类门户站点提供将用户留在自己站点的免费服务（黏性），如电子邮件和社区聊天板，然后再通过电子商务渠道向这些用户推销产品。

（4）商务（Commerce）：Amazon.com 和 eBay.com 是最著名的两家向客户提供电子商务服务的门户站点。电子商务为亚洲较为发达的市场（如澳大利亚、新加坡、韩国和中国香港地区，不包括日本）提供了巨大的商机。在另一些不太发达的市场中，首先必须解决结算和交付的问题。亚洲电子商务的一个真正的机会是生产厂商与购买厂商之间的商务（称为企业到企业 B2B 的商务）以 Asian Sources 为例，该公司为不同亚洲生产厂商生产的大宗产品寻找美国的买主；亚洲电子商务的另一个机会是供应链管理。在供应链管理中，信息系统的应用可以大大提高经商的效率（正像 Net-Cel.com 这类公司所追求的那样）。

（5）通信（Communications）：它是电子邮件以及 Internet 电话这类服务的前提。分别为 AOL 和 Microsoft 所拥有的 ICQ 和 Hotmail 提供电子邮件服务。值得注意的是，Internet 电话可能将是通信部分的下一个引起人们兴趣的应用。Netscape 公司最近宣布它将在产品中包括一个 IDT 公司的 Net2Phone 服务的图标，这种服务使美国用户可以从自己的 PC 打低价长途电话。该图标将出现在 Netscape 的浏览器和其 Netcenter 门户站点中。

（6）社区（Community）：社区包括像聊天室和兴趣小组这类设施。由于社区有助于将用户保持在门户站点，而不只是很快地经过，所以它正变得越来越重要。亚洲地区的门户站点都意识到其业务成功的关键将是发展“黏性”（stickiness），即一旦用户访问了 Web 站点，他（她）将在此 Web 站点呆上相当长一段时间，然后再离开。GeoCities 和 eBay 是两家从社区空间起家的著名公司。在中国市场中，Sina 和 Sohu 正在发展这种业务模型；在印度，IndiaWorld 和 Rediff 将注意力集中在迎合印度居民的需要；RenRen 则在香港地区努力实现这种模型。

随着作为商机的 Internet 的成熟，AOL 董事长 Steve Case 认为出现了六个新的从属性的“C”。这六个“C”没有对门户站点重新定义，但它们说明了原来的“六个 C”将来的发展方向。这六个“C”为：

（1）方便性（Convenience）：Internet 将可以被各种设备访问，这些设备包括电视、PC、移动电话和手持 PC。

（2）聚合（Convergence）：电视和 PC 仍将作为不同的设备，但它们将被 Internet 连接在一起。Internet 企业需要认识到聚合的界限在哪里。

（3）整合（Consolidation）：当 10 年前 PC 软件发展之时，许多公司都在销售单独的产品（如字处理程序或电子报表），Microsoft Windows 出现了并且将许多产品做在一起。Case 认为 Internet 也将发生这种情况。以 Microsoft 为例，它将其众多 Internet 特性整合在一个服务中，Microsoft 提供通信（Hotmail 的电子邮件服务、MSN Connect 和 WebTV 的接入）、搜索与导航（MSN.com）、商务站点（Expedia 的旅行、CarPoint 的汽车以及 Homeadviser 上的家庭购物）和内容（Encarta 上的百科全书、Gaming Zone 上的游戏、MSNBC 上的新闻以及 Slate 上的社论）。在亚洲，整合将需要时间，不过我们已经在 eCorp 的 ninemsn 门户站点上看到了一些这样的服务，该站点已提供 Hotmail，并在建立 Carpoint 和 Boatpoint 频道。

（4）竞争（Competition）：开拓 Internet 商业化的公司将面临来自内容提供商（如 News 公司和 Disney）和电信公司（如 AT&T）的更加激烈的竞争。

（5）合作（Cooperation）：Case 认为，Internet 企业的合作正在增加，这点在 Bertelsmann 与 Barnes & Noble 在网上图书零售中的合作得到了印证。其次，他认为在今后 5 年里，对 Internet 的规模和增长影响最大的是公共政策而不是技术，像隐私、儿童安全和税收这类问题将变得越来越重要。Internet 企业必须合作才能与政府一起讨论这些问题。

（6）客户关系（Customer Relationships）：通过与客户建立牢固的关系，各公司将可以在日益增长的竞争中更容易地留住客户。一个有趣的问题是门户站点是否能像报纸或电视网那样长期存在，它们是否只是 Internet 发展过程中的一个过渡阶段。随着消费者利用智能代理发展出更好的搜索能力，取代门户站点中介作用的力量可能会得到发展。总的来说，我们认为更强壮的门户站点，如 AOL、Yahoo 以及 Amazon.com 将长期存在，相信它们将迅速发展并包容新的技术。

18.1.3 门户站点的分类

（1）信息门户：信息门户的基本作用是为人们提供信息，它强调对结构化与非结构化数据的收集、访问、管理和无缝集成。这类门户必须提供数据查询、分析、报告等基本功能，用户可以通过企业信息门户非常方便地获取自己所需的信息。

（2）知识门户：知识门户还应该具有信息搜集、整理、提炼的功能，可以对已有的知识进行分类，建立知识库并随时更新知识库的内容。

（3）应用门户：应用门户实际上是对工作流程的集成。它以工作流程和服务应用为核心，把工作流程中功能不同的应用模块通过门户技术集成在一起。从某种意义上说，可以把应用门户看成是信息系统的集成界面，用户可以通过应用门户访问相应的服务和应用系统。

18.1.4 门户站点必须要解决的几个主要问题

虽然目前使用门户站点的多为企业，各种层出不穷的门户系统都是针对企业应用进行开发，但是通过分析企业门户所面对的问题，对读者在其他平台上应用门户系统也同样具有良好的借鉴作用。

1. SSO 单点登录

门户系统只有惟一入口，用户登录时采用单点登录 SSO（Single Sign-On）。传统登录方式下，系统管理员需要给每台机器上的系统，甚至是每台机器上的每个应用准备一套用户管理系统和用户授权策略，终端用户需要使用其中的任何应用时都需要做一次身份认证。

SSO 的机制是“单点登录、全网漫游”，用户访问系统作一次身份认证，随后就可以对所有被授权的网络资源进行无缝访问，而不需要多次输入认证信息。SSO 登录方式，减少了在不同系统中登录耗费的时间；避免了处理和保存多套系统用户的认证信息；减少了系统管理员管理用户权限的时间；增加了管理的便利性，可以通过直接禁止和删除用户来取消该用户对所有系统资源的访问权限；大大增加了系统的安全性。

2. PKI 与 CA 认证

传统方式下，安全方案是通过用户名/口令设置访问权限，但这只能提供单向用户认证，且口令容易被破译。对于那些需要进行双向身份认证的业务，这种用户名/口令的方案就无能为力了，这就需要 PKI（Public Key Infrastructure 公开密钥基础设施），它支持身份标识和认证、保密或隐私、数据完整性与不可否认等安全保障，是目前普遍认为最具实用性的方案。

CA 是 PKI 中最基本的元素，网络中的电子身份、电子交易行为、数据文件等安全操作都主要通过证书来实现。PKI 的部件还包括证书登记机构（RA）、存储和发布这些证书的电子目录、证书策略、证书路径、证书的使用者等。在门户系统中，身份验证、权限控制、电子交易等过程，CA 起着最关键的作用。

3. 业务流程（企业）

根据延续时间及复杂性，业务流程可分为三类：①流程到流程；②流程到人；③人到人。流程到流程的延续时间短且不复杂，主要用于从一个应用到另一个应用的数据转换，称为业务过程流（Process flow）；流程到人是以交易为中心，部分需要人工干预，部分可以自动完成；人到人的流程需要人紧密合作协同工作，后二者称为工作流（Work flow）。

4. 可伸缩性、扩展性

不同的单位对门户有不同的应用需求，可能是侧重于信息门户、知识门户或应用门户，也可能根据业务的发展需求会发生变化，门户系统的套件或解决方案应该具有良好的伸缩性，能够满足单位的不同需求。

换个角度说，门户产品套件应该是基于组件化开发的，具有良好的开放性，企业在某种程度上可以采用“热插拔”式应用，可以根据需求进行定制，能够方便地集成和运行各种应用系统。其 API 接口应该能处理业务对象和修改业务逻辑，可以配置用户界面，并易于降低开发和维护成本。硬件系统根据客户业务需要进行合理配置，随时扩展硬件，保护硬件投资。

5. 个性化的配置

门户系统的重要特性之一在于其个性化，针对不同的对象，定义不同的业务流程，提供不同的服务模式和服务内容。个性化应该包括以下内容：基于界面的个性化，不同的人员有不同的主界面，可定义个性化页面风格、样式、内容及使用方式，还可以定义自己的风格模板；基于工作的个性化，不同人员有不同的工作任务、工作资源、工作流程；基于规则的个性化，系统能够动态地制定角色权限和商务规则，以实现界面、内容、业务流程的个性化。

6. 与应用系统的集成

由于不同的单位在使用门户系统时会有不同的期望值和应用需求。下面我们以一个企业为例来说明在门户系统中如何集成应用系统。

EAI（企业应用集成）是企业门户的灵魂。能否实现对 EAI 的良好整合可以说是企业门户成功的关键。虽然说是“不破不立”，但建立企业门户并不意味着就一定要打破企业信息化原有的坛坛罐罐。在如今企业 IT 投资非常有限的情况下，如何能够充分利用企业原有系统是一

个非常重要的问题，这也是众多 CIO 们评估企业门户产品供应商的一个关键指标。

从 OA、CRM、SCM 到 ERP，从 Foxbase、FoxPro、Access 到 SQL Server 乃至 Oracle，企业内部原有的林林总总的应用和非结构化的数据，是企业门户必然要面对的。当然，企业内部各自为政的单元系统的应用所形成的信息孤岛，大多是历史遗留问题，是在企业信息化伊始就注定会存在的，因为信息化的整体规划总是滞后于信息化的单元应用。随着 Web 技术的日益发展和 B/S 模式的深入应用，这些问题终将一一得到解决。

7. 商务智能

门户系统应该具有多维的分类统计功能，通过观察跟踪，例如访问频率最高的页面和浏览量最多的内容、访问者的个人信息等，进行数据分析，得出访问者的偏好和客户的需求比率，同时通过协作过滤机制，推荐给具有同样兴趣的客户或处于同一产品链上的其他用户。另外，门户还应该通过对各类信息、数据和业务应用的综合统计、查询、分析，为决策提供快速的智能支持。

18.2 门户中间件

这一节介绍门户中间件。请读者注意学习下面几个基本概念：门户中间件、门户服务器（Portal Server）、Portal、Portlet 容器（Portlet Container）、Portlet。

18.2.1 门户中间件的定义

门户中间件指的是门户服务器（Portal Server）。门户服务器是 Web 应用服务器上的“应用”，是中间件市场上值得关注的新品。尽管有关门户服务器技术的介绍很早就有了，但大牌 IT 厂商的产品基本上都是 2002 年上半年推出的。

基于门户服务器建设企业门户的基本好处是开发商可以利用门户服务器提供的构筑门户应用的基础组件工具 portlet 小程序。应用开发商可以开发很多这样的“门户组件”，同时集成别人开发的“门户组件”来构建企业门户。使用门户的用户可以主动选择可选的“门户组件”进行个性化的选择，构造自己的“门户”。“门户”可以是对企业后端应用的访问，也可以是自己或别人网站的一部分。由于用户在向门户描述自己时可能给出自己的兴趣或爱好，门户网站也可以根据这些信息主动地“推”信息给使用者。

门户服务器在支持个性化和多渠道接入方面特点突出。门户服务器的重要意义还在于它作为 Web 服务的“客户端”的作用。有人这样看待门户技术，称它在 Web 服务时代的重要性就如同 Client/Server 时代的 Windows，因特网时代的浏览器。

18.2.2 Portal 介绍

这里讲到的 Portal 不同于前面一节介绍的门户系统，Portal 是一种 Web 应用，通常用来提供个性化、单次登录、聚集各个信息源的内容，并作为信息系统表现层的宿主。聚集是指将来自各个信息源的内容集成到一个 Web 页面里的活动。

完整的 Portal 通常由 Portal 服务器、Portlet 容器、Portlet 构成。Portal 服务器是容纳 Portlet 容器，支持 Portlet 呈现的普通或特殊 Web 服务器。Portal 服务器通常会提供个性化设置、单点登录、内容聚合、信息发布、权限管理等功能，支持各种信息数据来源，并将这些数据信息

放在网页中组合而成，提供个性化的内容定制，不同权限的浏览者能够浏览不同的信息内容。

Portal 的功能可以分为三个主要方面：

（1）Portlet 容器：Portlet 容器与 servlet 容器非常类似，所有的 Portlet 都部署在 Portlet 容器里，Portlet 容器控制 Portlet 的生命周期并为其提供必要的资源和环境信息。Portlet 容器负责初始化和销毁 Portlet，向 Portlet 传送用户请求并合成响应。

（2）内容聚集：Portlet 规范中规定 Portal 的主要工作之一是聚集由各种 Portlet 应用生成的内容。

（3）公共服务：Portlet 服务器的一个强项是它所提供的一套公共服务。这些服务并不是 Portlet 规范所要求的，但 Portal 的商业实现版本提供了丰富的公共服务，以有别于它们的竞争者。在大部分实现中都有望找到的几个公共服务有：

单一登录：只需登录 Portal 服务器一次就可以访问所有其他的应用，这意味着用户无需再分别登录每一个应用。例如一旦用户登录了他的 Intranet 网站，就能访问 mail 应用、IM 消息应用和其他 Intranet 应用，不必再分别登录这些应用。

Portal 服务器会为用户分配一个通行证库。只需要在 mail 应用里设定一次用户名和密码，这些信息将以加密的方式存储在通行证库中。在已登录到 Intranet 网站并要访问 mail 应用时，Portal 服务器会从通行证库中读取通行证替用户登录到 mail 服务器上。用户对其他应用的访问也将照此处理。

个性化：个性化服务的基本实现使用户能从两方面个性化其页面：第一，用户可以根据其自身的喜好决定标题条的颜色和控制图标；第二，用户可以决定在其页面上有哪些 Portlet。例如，如果是个体育迷，可能会用一个能提供钟爱球队最新信息的 Portlet 来取代股票和新闻 Portlet。

一些在个性化服务方面领先的商业实现版本允许建立为用户显示什么样的应用所依据的标准（如收入和兴趣）。在这种情况下，可以设定一些像“对任何收入为 X 的用户显示馈赠商品的 Portlet”和“对任何收入为 X 的用户显示打折商品的 Portlet”这样的商业规则。

此外还有一些公共服务，比如机器翻译，是由 Portal 服务器将 Portlet 生成的内容翻译为用户要求的语言。大部分的商业 Portal 服务器都支持手持设备访问，并具有针对不同的浏览终端生成不同内容的能力。

上面提到了 Portal 的主要功能，然而，在现实场景中，不论是开源的 Portal 框架实现，还是商业 Portal 产品都在 JSR168 规范标准的基础上作了扩展。总的说来，一般 Portal 可能会包含以下功能，如表 18-1 所示。

表 18-1　Portal 的功能

功能	描述
内容聚合	能够把各种不同应用的内容聚合到一个统一的页面呈现给用户
基于角色的视图定制	能够基于组织机构中不同的用户角色生成不同的视图内容。例如，人力资源总监和财务经理登录后所看到的页面也是不同的
个性化	用户能够根据个人喜好定制符合自己风格的页面和内容。例如，小王喜欢淡蓝色的格调，并且投资股票，则他可以选择一个淡蓝色风格的主题，并且使用一个已经定制好的股票 Portlet，允许小王设定此 Portlet 的自动刷新时间和自选股等

续表

功能	描述
单点登录	只需登录 Portal 服务器一次就可以访问所有其他的应用，这意味着用户无需再分别登录每一个应用
协作功能	一些 Portal 框架可能会提供复杂的 Portlets 用于聊天、应用程序共享、白板、在线会议、论坛等
国际化	根据 locale 的不同呈现不同国家的文字
工作流	这里主要指支持跨越不同数据源和应用的工作流
支持不同的客户端	包括主流 Web 浏览器、PDA 等

18.2.3 门户 JSR168 规范和 WSRP 标准

在实际的应用场景中，经常需要把不同的应用程序集成在一个统一的用户界面上。Portal 技术可以很好地解决这一问题，但需要开发一定数量的 Portlet。但有时需要把同一个 Portlet 部署在不同的门户中，此时可能要费很大的力气重新为每个门户重新开发相同的 Portlet。有很多通用的 Portlet 在升级应用服务器时不能很好地做到重用。此时怎么办？

只有遵循业界公认的标准，可以减轻移植和互操作带来的痛苦。随着门户标准 JSR168 和用于远程 Web 服务 Portlet（WSRP）的广泛采用，将能够很容易做到重用和维护。

总的来讲，这些新标准正在给门户部署的本质带来根本性的变化，伴随而来的是门户开发商市场的剧变。越来越多的客户根据门户的总体体系结构是否符合该企业来选择门户技术，而不是根据门户的一组专用的特性，如根据 Portlet 的数目来选择。

1. JSR168

JSR168 规范为创建 Portlet 建立了标准的 API。它是为实现 Portlet、基于 Java 的门户服务器和其他 Web 应用程序之间的互操作性而设计的。现在大多数开源的和商用的 Portal 产品都支持 JSR168。因此开发符合 JSR168 规范的 Portlet 的客户可以很容易地将 Portlet 从某一开发商的门户移到另一个开发商的门户中。此外，客户现在还可以使用迅速增长的开箱即用、符合标准的 Portlet。按照 Java 标准化组织（Java Community Process）所述，JSR168 Portlet 拥有一个适用于所有门户客户端的简单的标准的 API，支持多种类型的客户端（多设备、多浏览器），支持本地化和国际化，允许门户应用程序的热部署和重新部署，并包含声明性安全（与 servlet 和 EJB 规范中使用的机制相同）。

2. WSRP

由 JSR168 标准获取的益处被 WSRP 进一步得到增强。WSRP 是由 OASIS（一个由开发电子商务标准的行业专家所组成的非赢利性社团）创建，它使得开发的 Portlet 可以被远程的门户展现出来。WSRP 使原来极难实现的功能成为可能。例如，部署一次 Portlet，可以把它们传递到任何符合标准的门户中去；将第三方提供的 Portlet 整合进自己的门户中，增强来自不同开发商的门户之间的互操作性。

18.2.4 Portal 页面的元素

图 18-1 显示了 Portal 页面的各种元素。

Portal 页面由两部分组成：一部分是由页面中的 Portlet 生成的内容，另一部分是由 Portal

服务器生成的内容。

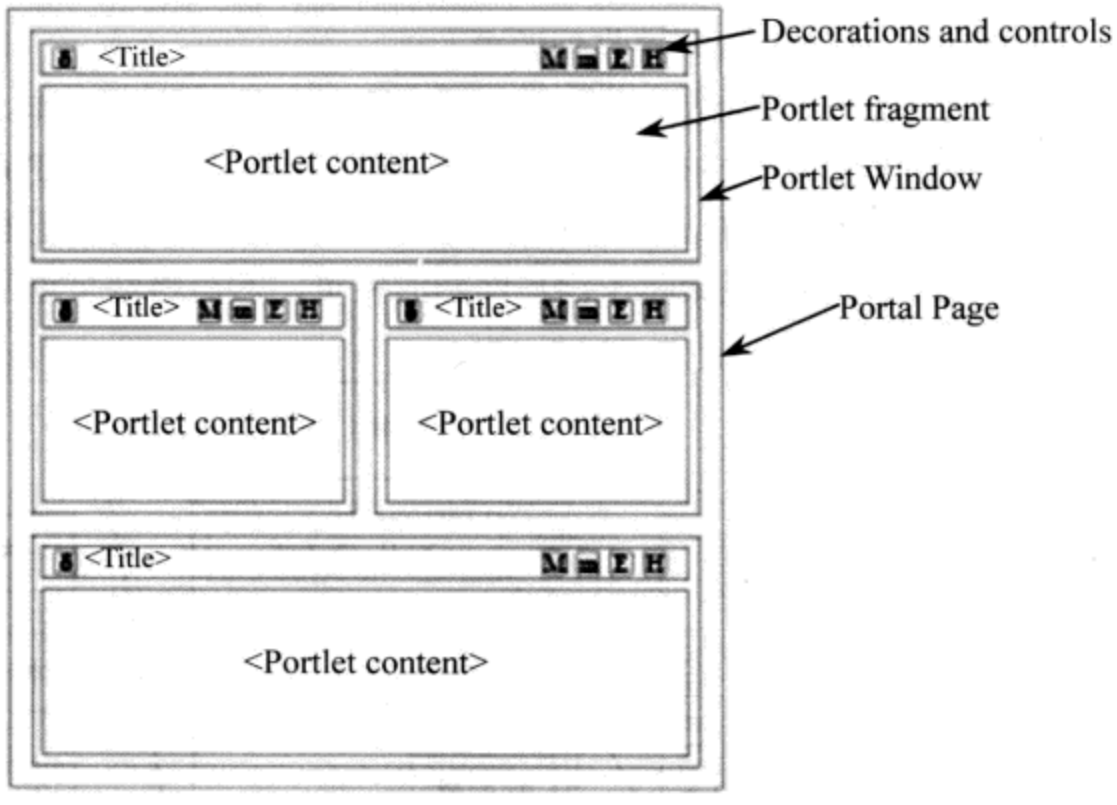


图 18-1　portal 页面的元素

每个 Portal 页面可以包括一个或多个 Portlet 窗口，每个 Portlet 窗口又分为两部分：一个是外观，它决定了 Portlet 窗口的标题条、控制和边界的样式；另一个是 Portlet 段，它由 Portlet 应用填充。

Portal 服务器决定了 Portal 页面的整体观感，像标识、标题条颜色、控制图标等。通过修改几个 JSP 和 css 模板文件就可以改变 Portal 的整个外观。

Portlet 运行在 Portlet 容器内。Portlet 容器接收 Portlet 外产生的内容。Portlet 容器把 Portlet 内容传到 Portal；Portal server 根据 Portlet 产生的内容创建 Portal 页面，并传送给客户端，由客户端显示，如图 18-2 所示。

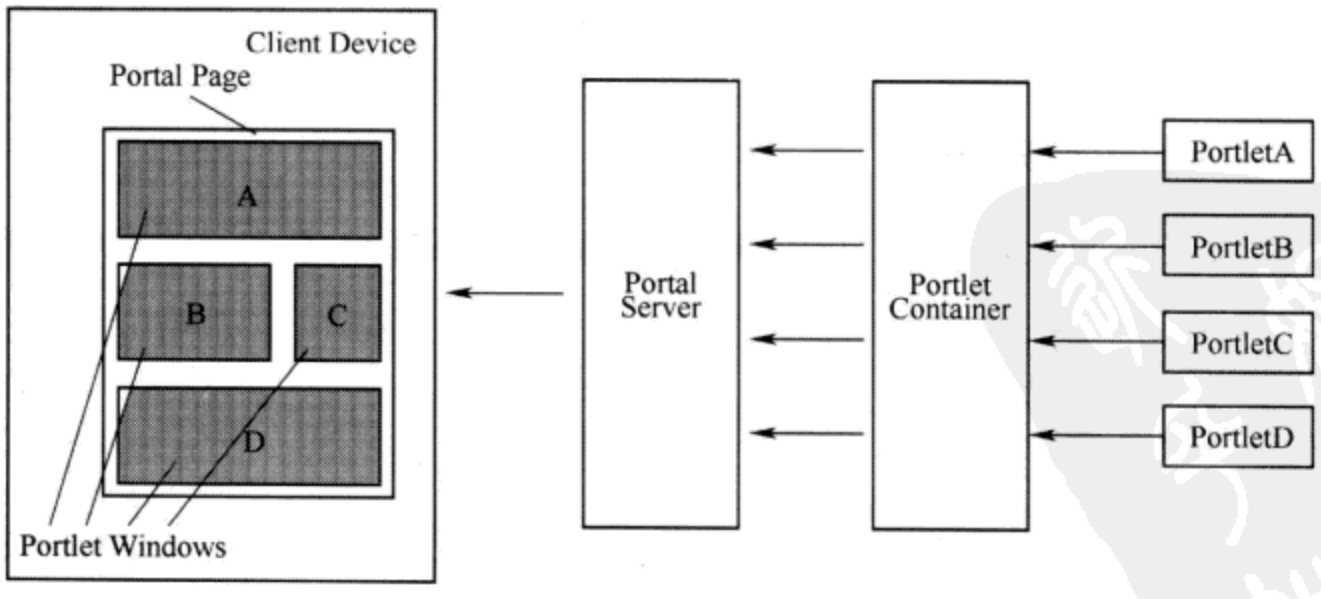


图 18-2　Portal Server、Portlet 容器与 Portlet 的关系

客户使用客户终端访问 Portal。在收到请求后，Portal 判断满足请求的 Portlet 列表。Portal 通过 Portlet 容器调用 Portlet。Portal 用 Portlet 产生的片断创建 Portal 页面，同时把页面返回给用户。

综上所述，Portal 的处理流程分为六个步骤（如图 18-3 所示）：

（1）一个客户端（如一个 Web 浏览器）在被验证之后向 Portal 发出 HTTP 请求。

（2）Portal（或称为 Portal Server）接收到请求。

（3）Portal 判断请求是否包含与组成门户网站网页的 Portlet 有关的动作。

（4）如果存在与某个 Portlet 相关的动作，Portal 请求 Portlet 容器调用 Portlet 处理动作。

（5）Portal 通过 Portlet 容器调用 Portlet，获得被包含在产生的门户网站网页中的内容片段。

（6）Portal 将 Portlet 产生的结果聚集于门户网站的网页，然后将网页返回至客户端。

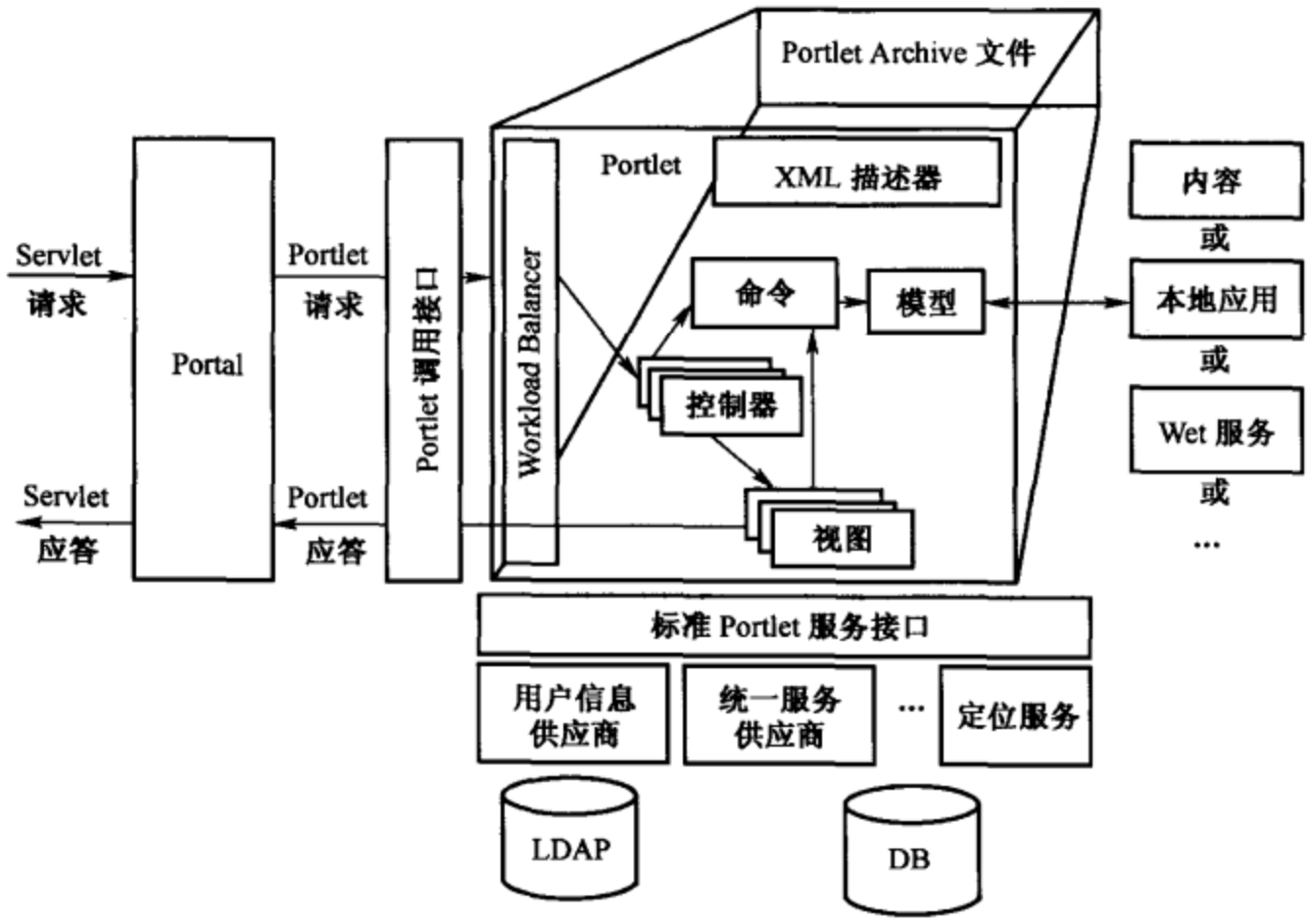


图 18-3 Portal 处理流程

需要注意的是，Portal 服务器是建立在 HTTP 服务器的基础上的，Portal 服务器不可独立运行。

18.2.5 Portal 的实现原理

下面介绍 Portal 的实现原理。读者可以在下面三个网站上注册自己的用户，体会 Portal 的功能：

http://my.msn.com

http://my.yahoo.com

http://my.liferay.com

My MSN 的功能最灵活强大，用户可以任意拖放操作栏目（column）和内容板块（content）的位置和个数。My Liferay 只能选择固定的栏目（column）布局，但可以在本栏目（column）内移动内容板块（content）的位置。My Yahoo 只能选择固定的栏目（column）布局，而且不能移动内容板块（content）的位置。My Liferay 的界面如图 18-4 所示。

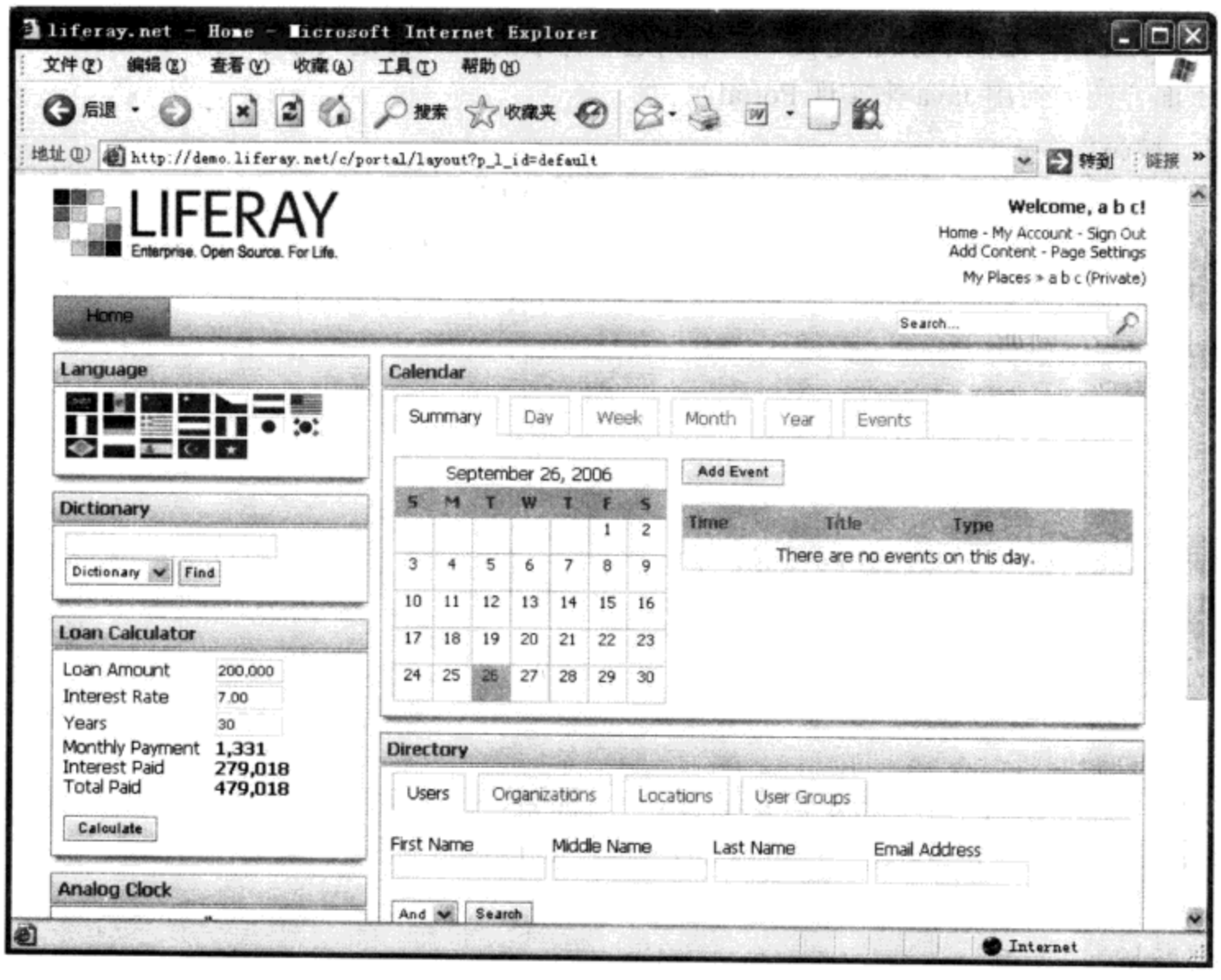


图 18-4　Liferay 门户的界面

Portal 的结构分为三层：

- Page
- Column（或称为 Pane）
- Content（或称为 Portlet）

Portal 的整个操作流程如下：

（1）每个 Column 的下方都有一个 Add Content 按钮，让用户选择加入自己喜欢的内容。从这里可以知道，Portal 系统里面有一个公用的 Common Portlet Repository，供用户选用。JSR168 Portlet 规范里面定义了 Portlet Deployment Discriptor。Common Portlet Repository 以这个 Portlet Deployment Discriptor 的格式存放。开源项目 JetSpeed 的 XReg 文件用来存放 Common Portlet Repository 的定义。

（2）加入 Content 之后，用户的 Page 和 Column 里面就多了这个 Content。下次用户登录时，就会看到自己定制的 Portal 版面。从这里可以看出，Portal 系统会记录用户的个人 Portal 配置信息 - User Portal Config。开源项目 JetSpeed 的 PSML 文件用来存放 User Portal Config 的定义。

综上所述，Add Content 的整个流程为：

Common Portlet Repository → Add Content → Personal Portal Config

Display Portal 的整个流程为：

从 Personal Portal Config 读取用户配置的 Portlet ID → 根据 Portlet ID 从 Common Portlet

Repository 查找详细的 Portlet 定义 → 根据这个详细的 Portlet 定义显示这个 Portlet。

下面介绍如何用 Java 来实现 Portal。

1. 动态包含

首先，采用最简单的思路，用 100 个 JSP 文件（1.jsp，2.jsp，3.jsp，… 100.jsp 等）代表 100 个 Portlet。用户页面 MyPage.jsp 包含用户选定的多个 Portlet。假设用户选取的 Portlet 为 1.jsp, 3.jsp 和 7.jsp 三个 Portlet，那么如何在 MyPage.jsp 中显示这些 Portlet？最直观的做法是，用 jsp:include。例如：

```
<table>
   <tr><td>
      <jsp:include page="1.jsp" />
   </td></tr>
   <tr><td>
   <jsp:include page="3.jsp" />
   </td></tr>
   <tr><td>
   <jsp:include page="7.jsp" />
   </td></tr>
</table>
```

由于<jsp:include>只能指定固定的 JSP 文件名，不能动态指定 JSP 文件名，需要把<jsp:include>翻译为 Java code - RequestDispatcher.include()。下面换成这种写法。

Java 代码

```
<table>
   <tr><td>
      <% request.getRequestDispatcher("1.jsp").include(request, response); />
   </td></tr>
   <tr><td>
      <% request.getRequestDispatcher("3.jsp").include(request, response); />
   </td></tr>
   <tr><td>
      <% request.getRequestDispatcher("7.jsp").include(request, response); />
   </td></tr>
</table>
```

进一步改进 MyPage.jsp 如下：

```
<% String[] fileNames = {"1.jsp", "3.jsp", "7.jsp"}; %>
<table>
<% for(int i = 0; i < fileNames.length; i++) {
  String fileName = fileName s[i]; %>
   <tr><td>
      <% request.getRequestDispatcher(fileName).include(request, response); />
   </td></tr>
<% } // end for %>
</table>
```

其中的 fileNames 的内容可以各种各样，只要 RequestDispatcher 能够处理。

比如 Velocity，fileNames = {"1.vm", "3.vm", "7.vm"};

比如 URL，fileNames = {"/portlet1.do", "/portlet3.do", "/portlet4.do"};

可以看到，如果从用户配置中读取 fileNames 的内容，这就是一个简单的 Portal 实现。

Java 代码

```
<% String[] fileNames = (String[])session.getAttribute("portlets.config"); %>
<table>
<% for(int i = 0; i < fileNames.length; i++) {
  String fileName = fileNames[i]; %>
    <tr><td>
        <% request.getRequestDispatcher(fileName).include(request, response); />
    </td></tr>
<% } // end for %>
</table>
```

2. Portlet 接口

下面扩展这个例子。假设每个 Portlet 都规定实现一个 Portlet 接口。

```
interface Portlet {
  void render(request, response);
};
```

MyPage.jsp 如下：

```
<% String[] portletClassNames = (String[])session.getAttribute("portlets.
      config"); %>
<table>
<% for(int i = 0; i < portletClassNames.length; i++) {
  String className = portletClassNames[i];
  Portlet portlet = (Portlet)Class.forName(className).newInstance(); %>
    <tr><td>
        <% portlet. render (request, response); />
    </td></tr>
<% } // end for %>
</table>
```

Portlet 类的示例代码如下：

```
public class Portlet7{
  public void render(request, response){
    request.getRequestDispatcher("7.jsp").include(request, response);
  }
};
```

上述代码是 Portal 显示 Portlet 核心流程的一个简化版本。JSR168 Portlet 规范里定义了真正的 Portlet 接口定义。

3. Portlet 操作

Portlet 的操作包括：最大化、最小化、恢复、关闭、编辑、帮助、上下移动等。这些操作都有对应的 Action 类。开源项目 JetSpeed 的 module/actions/controls 目录下面包含 Maximize、Minimize、Close 等 Action 类。开源项目 Liferay 的 portal/action 目录下面包含 Maximize、Minimize、Close 等 Action 类。Portal 的操作不仅包括上述 Portlet 的操作，而且包括其他更高级别的操作，如 Add/Move Page、Add/Move Column、换 Layout、换 Skin 等。

4. Portlet Cache

操作 Portlet 时，往往只操作某个特定的 Portlet，或者只是变化 Portlet 的位置。这时页面中大多数 Porlet 的内容是不变的，只有一小块 Portlet 变化。需要把 Portlet 的内容缓存起来。Portlet 接口有一个 render（request，response）方法，我们可以定制 response 类截获 Portlet 的

输出，保存到 Portal 系统的内容 Cache 当中。比如，前面提到 Liferay 开源项目，其 StringServletResponse 类把 Portlet 的输出保存到一个 String 当中。

18.2.6 几种开源 Portal 的简单介绍分析

下面介绍四种主要的开源 Portal，包括 Pluto、Liferay、eXo 和 Jetspeed，有助于对具体项目选择合适的开发平台。

（1）Pluto。2003 年 10 月 JSR168 规范 1.0 正式公布后，Jakarta Apache 就开始实施 Pluto 计划（冥王星计划），最终开发出该规范的一个参考实现（Reference Implementation），即 Pluto。Pluto 的 1.0.1-rc2 版于 2004 年 12 月发布。

Pluto 实现基于 JSR168 的一个 Portlet Container，相当于为开发者提供了一个运行 Portlet 的工作平台。Pluto 本身也提供了一个简单的 Portal 模块，该模块仅仅是为了满足 Portlet 容器和 JSR 168 的需要而写的，因而显得非常简单，提供的实用的 Portlet 也非常少。从某种意义上说，Pluto 更像是一个 Portlet Container，作为一个实用的 Portal 开发框架尚需要更强大的支持。但新版本的 Pluto 仍没有推出。对于 Pluto 的应用开发，Apache 更推荐使用 Jetspeed 项目框架。

尽管 Pluto 作为一个完整的 Portal 应用还非常欠缺，但不少有影响力的 Portal 项目使用 Pluto 作为 Portlet Container。这些项目包括 Jetspeed、Cocoon uPortal、Jahia 等。由此可见 Pluto 的重要性。从开发者和学习者的角度看，Pluto 的意义还在于为开发者和学习者提供一个深入了解 Portlet Container 的简洁的参考实例。

（2）Liferay。Liferay（支持 JSR168）代表了完整的 J2EE 应用，最高版本是 2005 年 1 月推出的 Professional 3.2.0，其主要优点有：

1）使用第三方的开源项目，如 Hibernate 等。特别是前台界面部分使用了 Struts 技术。

2）支持包括中文在内的多种语言。

3）支持较多的先进技术，如 Web Services、EJB、JMS、SOAP、XML 等。

Liferay 的缺点是它缺乏一个简单清晰可拓展的架构设计，整个架构比较复杂且庞大；Struts1.1 本身并不支持 JSR168，所以 Liferay 在实现诸如上下文共享等问题上显得十分笨重且没有从根本上解决这些问题；Portlet 设计也显得比较凌乱。此外，如果读者的门户系统准备应用于商业用途，那么需要购买 License，并且基于它进行二次开发比较困难。

（3）eXo。eXo（支持 JSR168）基于 JSF 的 Portal 实现。最新版本是 2004 年 10 月发布的 1.0RC1 版。

主要优点包括：

1）由 AOP（AspectJ）实现的内容管理系统，极大地提高了内容管理性能。

2）基于 Pico Container 的 Portlet Container，Pico 是一个著名的 IoC3 轻量级容器。同时也实现了上下文共享，二次开发的流程比较清晰。

3）使用 Struts 框架技术。

4）提供工作流技术服务（Workflow Service）。

5）提供了很多交流工具，通过 XML 可以为结构化的信息轻易地创建视图。

由上可见 eXo 采用了诸多先进技术，但存在不少缺点。主要缺点是：由于 Portal Server 本身的数据是使用 xmldb 来处理，保存到数据库的数据都是乱码而且它所有默认的平台字符

集都是 ISO-8859_1；缺乏中文的充分支持，对于中文门户的开发并没有优势；由于 JSF 是重量级的表现层框架，使得 eXo 的二次开发工作量比较大；对于商业 Portal 应用开发需要购买 License；总体开发难度较大。

（4）JetSpeed。JetSpeed 是 Apache 组织的开源 Portal 项目。JetSpeed 目前有两个版本可供选择：1.X 和 2.0 版本。JetSpeed1.X 出现得比较早，但第一个较为成熟的版本发布是 1.4。此后 1.X 不断推出新版本。但 JetSpeed1.X 都不支持 JSR168。当 JSR168 在 2003 年发布后，Apche 开始开发 JetSpeed2，提供对 JSR168 的全面支持。

18.2.7　Portlet 介绍

Portlet 是一种基于 Web 组件的 Java 技术，由 Portlet Container（即 Portlet 容器）进行管理，处理请求并动态返回页面，可以作为 Portal 的可即插即用的界面组件。

Portlet Container 用来管理 Portlet 的生命周期，提供其运行所需要的必要环境，并且给 Portlet Preferences 提供持久性（Persistent）存取服务，但是其不支持 Portlet 的 aggregation（内容聚合）。内容聚合由 Portal 组件提供，这一点需要弄清楚。注意，Portlet Preferences 是 Portlet 的一个新特性，提供类似数据库的功能，但不能用来取代数据库，只能用来存取简单的 Portlet 配置参数。

与 servlet 类似，Portlet 是部署在容器内用来生成动态内容的 Web 组件。从技术角度讲，Portlet 是一个实现了 javax.portlet.Portlet 接口的类，它被打包成 war 文件格式部署到 Portlet 容器里。

Portlet 在以下方面与 servlet 相似：

- Portlet 由特定的容器管理。
- Portlets 生成动态内容。
- Portlet 的生命周期由容器管理。
- Portlet 通过请求/响应模式与 Web 客户端交互。

Portlet 在以下方面与 Servlet 相异：

- Portlet 只能生成标记段，而不是整个文档。
- Portlet 没有可供直接访问的 URL 地址。不过还是能够让别人通过 URL 访问到 Portlet，可以把包含该 Portlet 页面的 URL 发给他。
- Portlet 不能随意地生成内容，这是因为 Portlet 生成的内容最终要成为 Portal 页面的一部分。如果 Portal 服务器要求的是 html/text 类型，那么所有的 Portlet 都应生成 html/text 类型的内容。再比方说，如果 Portal 服务器要求的是 WML 类型，那么所有的 Portlet 都应生成 WML 类型的内容。

Portlet 还提供了一些附加的功能：

（1）设置参数的持久化存储：Portlet 提供了一个 PortletPreferences 对象，用来保存用户的设置参数。这些参数被存入一个持久化数据库，这样服务器重启后数据依然有效。开发者不必关心这些数据存储的具体实现机制。

（2）请求处理：Portlet 提供了更为细粒度的请求处理。对于用户在 Portlet 上动作时向该 Portlet 发出的请求（一种称为活跃期的状态），或者因用户在其他 Portlet 上动作而引发的刷新页面请求，Portal 服务器提供了两种不同的回调方法来处理。

（3）Portlet 模式：Portlet 用模式的概念来表示用户在做什么。在使用 mail 应用时，用户

可能会用它来读信、写信或检查信件——这些都是 mail 应用的预定功能，Portlet 通常以 VIEW 模式提供这些功能；但还有一些活动，像指定刷新时间或（重新）设置用户名和密码，这些活动允许用户定制应用的行为，因此它们用的是 EDIT 模式；Mail 应用的帮助功能用的是 HELP 模式。

如果仔细想想，其实这里面并没有什么新东西，它们反而大部分都是普通的业务需求。Portlet 规范的作用在于它提供了一个抽象层，这才是它对所有与之相关的人（最终用户、开发者和管理员）的价值所在。

作为一个开发者，可以将所有与 VIEW 模式有关的业务逻辑放入 doView()方法，将与应用配置有关的业务逻辑放入 doEdit()方法，将与帮助有关的逻辑放入 doHelp()方法。这就简化了管理员对 Portlet 应用的访问控制管理，因为只需改变 Portlet 的访问权限就能决定用户能做什么。例如，如果 mail 应用的一个用户能够在 EDIT 模式下设定用户名和密码，那么就可以断定他具有 EDIT 模式访问权限。

不妨考虑这样一种情形：我是一个 Intranet 网站的管理员，我的公司买了一个能显示新闻信息的第三方 Portlet 应用，该应用允许用户指定跟踪新闻更新的 URL 地址，我想借助它为用户显示公司的内部新闻。另一个需求是我不想让用户通过该应用来跟踪任何其他的新闻信息来源。作为管理员，我可以为所有的用户指定一个用于内部新闻更新的 URL 地址，同时通过改变 Portlet 应用的部署描述符来取消其他人修改该地址的权限。

由于所有的 Portlet 应用都具有相似的 UI 界面，因此采用 Portlet 可使网站对最终用户更具吸引力。如果用户想阅读任何一个应用的帮助信息，可以单击“帮助”按钮；用户也知道单击“编辑”按钮能让他进入应用的配置屏。标准化的用户界面使得 Portlet 应用更吸引人。

（4）窗口状态：窗口状态决定了 Portal 页面上留给 Portlet 生成内容的空间。如果单击最大化按钮，Portlet 将占据整个屏幕，成为用户惟一可用的 Portlet；而在最小化状态，Portlet 只显示为标题条。作为开发者，应当根据可用空间的大小来定做内容。

用户可以根据需要通过增删 Portlet 的方式配置他们的页面。在 Windows 或 XWindows 窗口程序中，窗口有最大化和最小化的窗口状态，Portlet 也有窗口状态。窗口状态是 Portal 页面空间数量的指示器，它被赋值给 Portlet 产生的内容中。当调用 Portlet 时，容器提供当前窗口状态给 Portlet。Portlet 使用窗口状态决定多少信息需要 render。Portlet 能在处理 action 请求时编程改变他们的窗口状态。Portlet 规范定义了 3 种窗口状态：normal、maximized 和 minimized。

1）Normal 窗口状态。表示 Portlet 要和其他的 Portlet 共享页面。它也表示目标设备限制了显示能力。因此，Portlet 限制输出的尺寸。

2）Maximized 窗口状态。表示 Portlet 是 Portal 页面惟一的 Portlet，或与其他的 Portlet 相比有更多的空间。当窗口状态是 Maximized 时，Portlet 产生更多的内容。

3）Minimized 窗口状态。当 Portlet 是 minimized 状态，它将是最小限度的输出，或不输出。

（5）用户信息：通常 Portlet 向发出请求的用户提供个性化的内容，为了能更加行之有效，Portlet 需要访问用户的属性信息，如姓名、email、电话等。Portlet API 为此提供了用户属性的概念，开发者能够用标准的方式访问这些属性，并由管理员负责在这些属性与真实的用户信息数据库（通常是 LDAP 服务器）之间建立映射关系。

18.2.8　Portlet 的四种模式

前面说到了 Portlet 的模式，Portlet 有四种模式：View，Edit，Help 和 Config。

可用的模式根据用户的角色受到限制。例如，匿名用户只能使用 View 和 Help 模式，而已验证用户可以使用 Edit 模式。

举例来说，一个用户可以定制的股票信息 Portlet，View 模式允许用户查看股票列表，Help 模式提供给用户帮助手册，Edit 模式允许用户定制自己关注的股票列表，而 Config 模式允许管理员改变股票服务的一些配置。

（1）View 模式。View 模式期待的功能是产生标记，反映当前的状态给 Portlet。例如，View 模式的 Portlet 包含 1 个或多个框，用户可以操作或交互，或组成不需要任何用户交互的静态内容。

Portlet 开发者需要通过覆盖 doView 方法实现 view Portlet 模式的功能。

（2）Edit 模式。通过 Edit 模式，Portlet 提供内容和逻辑让用户用户化 Portlet 的行为。Edit 模式包含一个或多个框，用户可以定位并输入他们的用户化数据。

典型地，Edit 模式将设置或修改 Portlet 的参数。Portlet 开发者需要通过覆盖 doEdit 方法实现 edit Portlet 模式的功能。

（3）Help 模式。当在 Help 模式，Portlet 提供 Portlet 的 help 信息。这个 help 信息将会是简单的框，用连贯的正文或上下文敏感的帮助阐明了整个 Portlet。Portlet 开发者需要通过覆盖 doHelp 方法实现 Help Portlet 模式的功能

（4）Config 模式。当在 Config 模式时，管理员可以进行相应的操作。Portlet 开发者需要通过覆盖 doConfig 方法实现 Config Portlet 模式的功能。

18.2.9　Portlet 的生命周期

一个 Portlet 有着良好的生命周期管理，定义了怎样装载、实例化和初始化，怎样响应来自客户端的请求及怎样送出服务。

为了成功地创建 Portlet，必须遵照 Portlet 生命周期。javax.portlet.Portlet 接口中的方法定义了该生命周期，这些生命周期方法是 init()、render()、processAction()和 destroy()。

（1）初始化。当启动一个 Portlet 应用或在加载条件下，Portal 容器需要某个 Portlet 执行来自客户端的请求时，需要执行 init()方法。它用于获得所需的任何昂贵资源（如数据库连接），并执行其他一次性活动。在 Portlet 初始化时，经常需要用到 PortletConfig 对象获取初始化参数和各种资源。

（2）处理请求。通过调用 processAction()方法处理不同种类的动作和 render()方法呈现内容。通常是通过 Portlet 创建的 URL 提交请求。Portlet URL 包括 action URL 和 render URL 两种类型。一般情况下，一个 aciton URL 对应一个 action 请求和当前 Portle 页面上所有的 Portlet 的 render 请求，一个 render URL 对应当前 portle 页面上所有的 Portlet 的 render 请求。但是如果 Portlet 使用了缓存技术，则 portal/portlet 容器可能不会调用 render()方法，而直接从缓存中取出展现标记段。

- 动作处理

调用 processAction()方法，在此方法中可以获得 ActionRequest 和 ActionResponse 两个参

数。通过 ActionRequest 参数，可以获取 action 请求的参数、窗口状态、porlet 模式、portal 上下文对象、portlet 会话对象和 Portlet preference 数据。在处理 action 请求时，可以转发到一个指定的 URL。通过 ActionResponse 对象，Portlet 可以改变它的模式和窗口状态。

- 呈现内容

调用 render()方法，可以获得 RenderRequest 参数和 RenderResponse 两个参数。通过 RenderRequest 参数，可以获取 action 请求的参数、窗口状态、porlet 模式、portal 上下文对象、portlet 会话对象和 portlet preference 数据。通过 RenderResponse 对象可以生成呈现内容，或委托给 servlet 或 JSP 生成呈现内容。但也有一些限制：第一，生成 HTML 标记段不得使用下列标签：base、body、iframe、frame、frameset、head、html 和 title；第二，生成 XHTML 和基于 XHTML 标记段不得使用下列标签：base、body、iframe、head、html 和 title。

javax.portlet.GenericPortlet 类提供了呈现内容的一些默认功能。我们创建的大多数 portlet 将会扩展 javax.portlet.GenericPortle 类，而不是直接实现 javax.portlet.Portlet 接口。GenericPortlet 类实现了 render()方法。如果 portlet 的窗口状态被最小化，那么 render()方法不能做任何事情。如果 portlet 的窗口状态不是最小化，那么 render()方法设置在 portlet.xml 文件中指定的标题，并调用 doDispatch()方法。根据 Portlet 模式，doDispatch()方法适当地调用 doView()、doEdit()和 doHelp()方法。这样，由于 GenericPortlet 类帮助实现 render()方法，并且提供 doView()、doEdit()和 doHelp()方法来覆盖，因此 GenericPortlet 类比 Portlet 接口更便于扩展。

（3）完成。当 Portlet 的实例被撤销部署时，使用 destroy()方法来释放这些资源。

18.2.10 GenericPortlet 基类

GenericPortlet 像 Servlet 一样，编写的 Portlet 也必须直接或者间接地扩展基类 GenericPortlet，这个是由 JCP 针对 Portal 提出的 JSR168 规范定义的。只要扩展自规范的 GenericPortlet，所有的 Portlet 都可以在支持 JSR168 规范的 Portal 服务器上运行。

GenericPortlet 统一定义了可供 Portal 容器识别和调用的方法，包括：

public init()：初始化。

public init(PortletConfig)：初始化。

public getInitParameter(String)：取得在 Portlet.xml 中定义的初始化参数。

public getInitParameterNames()：取得在 Portlet.xml 中定义的全部初始化参数。

public getPortletConfig()：取得包含初始化参数的配置对象 PortletConfig 实例。

public getPortletContext()：取得 Portlet 上下文。

public getPortletName()：取得在 Portlet.xml 中定义的 Portlet 名称。

public getResourceBundle(Locale)：取得 Portlet 国际化的 Resource Bundle。

protected getTitle(RenderRequest)：取得 Portlet 的标题。

protected doView(RenderRequest，RenderResponse)：Portlet 浏览模式的处理方法。

protected doEdit(RenderRequest，RenderResponse)：Portlet 编辑模式的处理方法。

protected doHelp(RenderRequest，RenderResponse)：Portlet 帮助模式的处理方法。

protected doDispatch(RenderRequest，RenderResponse)：Portlet 行为分发。

protected processAction(RenderRequest，RenderResponse)：Portlet 处理 Action Request 的方法。

protected render(RenderRequest，RenderResponse)：Portal 处理 Render Request 的方法。

public destroy()：Portlet 销毁，终止其生命周期。

在 Portlet Portal 运行时，doView、doEdit 和 doHelp 三个方法分别被调用，用以生成 Portlet 标记。同样也可以调用 Servlet 生成 Portlet 标记，或者不调用 JSP 或 Servlet，直接在方法中得到 PrintWriter，然后用最简单的 pw.println()打印出内容。这个过程类似 Servlet。

18.2.11　Portlet 标签库

使用 portlet 标签库允许用户在 JSP 中获取 Portlet 特定的元素，如 RenderRequest 对象和 RenderResponse 对象；还可以在 JSP 中生成 Portlet URL。在 JSP 页面中必须有类似下面的声明：

```
<%@ taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
```

18.3　JetSpeed 项目介绍

Jetspeed 是 Apache Jakarta 项目中的开放源码门户系统，使用 Java 和 XML。它是企业级信息门户的原型实现，是 portlet 的运行环境，它包含了许多 Portlet 实例，可以管理众多的 Portlet，还提供了 Portlet API，可方便地用于开发新的 Portlet。它使得最终用户可以通过手机、浏览器、PDA 等多种设备来使用各种各样的网络资源（如应用程序、数据以及这之外的任何网络资源）。在这里，Jetspeed 扮演了一个处于信息和用户之间的 hub 的角色。

Jetspeed 的实现需要依赖 Apache Jakarta 的其他几个项目：Turbine、Velocity 和 ECS，如图 18-5 所示。

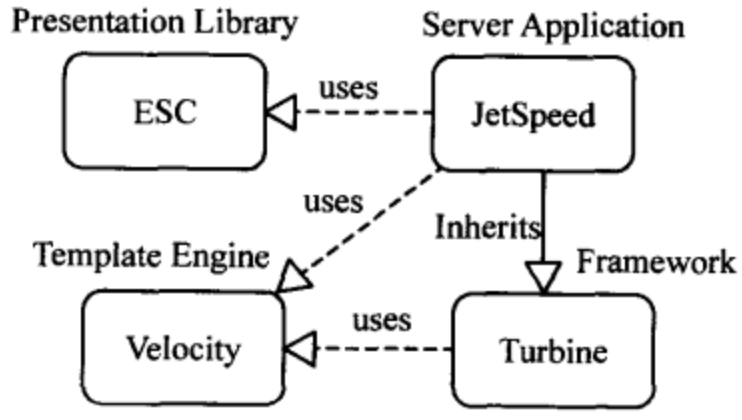


图 18-5　Jetspeed 与 Apache Jakarta 其他项目之间的关系

由图 18-5 可以看出：

（1）Jetspeed 是一个基于 Turbine 网络应用框架（Framework）的 Web 应用引擎。Turbine 是构建 Web 应用程序的 MVC 框架，任何支持 Servlet 标准的服务器都可以平稳地运行 Turbine（如 Tomcat、Resin 等）。Servlet 充当了 controller 的作用，它处理浏览器请求和启动用户需要访问的后端资源对象数据库、网格资源等，相当于一个 Java Bean，或者生成将在浏览器中显示的包含结果数据的"View" Bean。Jetspeed 在 turbine 的基础上提出了 Portlet 的概念，提高了网站的可定制性和可重用性。

（2）Velocity 是一种基于 Java 的模板引擎，它允许网页设计者能够使用在 Java 代码中定义的方法，velocity 将 Java 代码与网页设计分开了，使网页设计者能够与 Java 程序员并行工作来开发网站。Velocity 能用于生成网页、SQL 和其他模板所能产生的输出。Velocity 能够为

Turbine 应用框架提供模板服务。Velocity＋Turbine 能让 Web 应用程序真正地按照 MVC 模式来开发。Jetspeed 和 Turbine 都使用了 Velocity 这个模板框架来开发 portal 页面。

（3）ECS 的全名是 Element Construction Set，它是一个 Java API，为各种标记语言（包括 HTML4.0 和 XML）生成元素（element）。Jetspeed 是一种基于 XML 的应用引擎，它的注册表 xreg 文件和门户视图的门户结构标记语言 psml 文件都是基于 XML 的。

在 Jetspeed 中有三种最重要的文件：

（1）xreg 文件：xreg 相当于 Jetspeed 的注册表，用于登记 portlet、control、controller、skin、mediatype 等原始资源的定义，Jetspeed 中默认地把它实现为文件形式，各种原始资源类型都有自己的注册表文件。

（2）Psml 文件：Psml 是门户结构标记语言的简称。用于组织 xreg 中的原始资源形成一个对门户视图的定义，当用户使用桌面浏览器访问 Jetspeed 系统时，这个系统根据用户的 URL 定位一个 Psml 文档，接着解释这个文档形成 HTML 代码流返回给浏览器，浏览器展现这个代码流从而形成视窗化的门户视图。以上两种文件是实时地影响 Jetspeed 运行的参数，系统管理员可以在系统运行时时改变它们。

（3）properties 文件：Properties 定义了 Jetspeed 的重要服务及其参数，是 Jetspeed 启动时需要读的参数，系统运行时不会改变它，如果改变了这些参数，Jetspeed 必须重新启动。

18.4 JetSpeed 的安装配置

18.4.1 JetSpeed 安装的需求

安装 JetSpeed 时必需准备如下组件：

（1）一个 Java Servlet 引擎/容器，必须支持 Servlet 2.2 或 Servlet 2.3 API。Jetspeed 是在 Tomcat 上开发的。

（2）JDK 1.3 或更高版本的 Java 虚拟机，Jetspeed 已经在大多数 SUN 和 IBM 的虚拟机上进行了测试。

18.4.2 安装 JetSpeed

这个安装过程包含了将 Jetspeed 作为一个 J2EE Web 应用程序进行安装。安装一个 Web 应用程序很简单。对于 Web 应用程序来说，这个应用程序都可以包含在一个文件中，然后将这个文件放入一个特定的目录。

下面是下载 Jetspeed，编译和安装它的步骤：

（1）下载或者从 CVS 获取 Jetspeed，参考前面的内容。

下载 JetSpeet 的网址是 http://portals.apache.org/。

（2）编译 Jetspeed。如果下载的是 WAR 包，那么跳过这一步。

```
cd <jetspeedRoot>/build
[Win32/DOS] .\build war
[Win32/cygnus] bash build.sh war
[unix] ./build.sh war
```

（3）下载 Tomcat 或者其他支持 Servlet 2.2 或 2.3 API 的容器。这里使用 Jakarta Tomcat 作为例子，下载网址为 http://tomcat.apache.org/。

（4）安装 Tomcat，请参考 Tomcat 站点的文档。

（5）把 Jetspeed WAR 文件拷贝到 Tomcat 的 Web 应用程序目录中：

```
[Win32] copy <jetspeedRoot>\bin\jetspeed.war <tomcat_home>\webapps\
[Unix] cp <jetspeedRoot>/bin/jetspeed.war <tomcat_home>/webapps/
```

（6）启动 Tomcat：

```
[Win32] <tomcat_home>\bin\startup.bat
[Unix] <tomcat_home>/bin/startup.sh
```

（7）连接到 Jetspeed，使用链接 http://localhost:8080/jetspeed/，已经创建了两个默认的账号：

用户名: turbine 密码: turbine。

用户名: admin 密码: jetspeed。

在注册一个新用户时，注册过程包含一个向用户发送电子邮件的步骤。如果要在 Windows NT/2000 上使用这个功能，需要一个运行在本地的 SMTP 服务器，可以使用 Apaches James Mail 企业版服务器。邮件主机的设定将需要做一些修改，修改的文件是<tomcat_home>/webapps/jetspeed/WEB-INF/conf/TurbineResources.properties。

18.4.3　数据库配置

默认的 Jetspeed 部署使用它自己的安全模型来对用户进行授权和保留最少的用户信息，Turbine 安全模型可以和任何 JDBC 2.0 兼容的驱动程序工作。下面的数据库是经过测试的：

- Hypersonic SQL
- MySQL
- Postgres
- DB2
- Oracle
- Sybase
- SQL Server

Jetspeed 中包含一个流行的 Hypersonic-SQL 数据库，这个数据库是系统默认的数据库，配置选项在 Torque.properties 文件中。如果要让 Jetspeed 使用不同的数据库，需要修改 Torque.properties 文件。下面是针对 MySQL 数据库的例子：

```
    torque.database.default.adapter=mysql
    torque.dsfactory.default.connection.driver = org.gjt.mm.mysql.Driver
    torque.dsfactory.default.connection.url = jdbc:mysql://localhost:3306/
jetspeed
    torque.dsfactory.default.connection.user = root
    torque.dsfactory.default.connection.password = <password>
```

下面的是针对 Oracle 的例子：

```
    torque.database.default.adapter=oracle
    torque.dsfactory.default.connection.driver=oracle.jdbc.driver.OracleDriver
    torque.dsfactory.default.connection.url = jdbc:oracle:thin:@<host>:1521:<sid>
    torque.dsfactory.default.connection.user = <username>
```

```
torque.dsfactory.default.connection.password = <password>
```

对于 Oracle 来说，需要替换的是：

<host>：Oracle 实例的主机名，如 localhost。

<sid>：主机上的 Oracle 数据库 sid。

<username>：Oracle 数据库用户名。

Torque.properties 文件位于/webapp/WEB-INF/conf 目录中。建立数据库的脚本包含在源代码发行版中的 src/sql/external 目录下。只需运行相应的脚本，例如：使用 DB2 时就运行 turbine-db2.sql 来创建数据库表。对于使用 Hypersonic SQL 来说，这一步是没有必要的。如果要移植数据库，系统提供了/src/sql/external/populate*.sql 脚本，只要运行针对你的数据库的脚本即可。

该脚本将创建两个数据库用户：

用户名: turbine 密码: turbine。

用户名: admin 密码: jetspeed。

18.5 JetSpeed 开发实例

本节以 Hello World 为例，介绍如何使用 JSR 168 API 编写 portlet，打包成 portlet 应用，部署 portlet 应用到 portlet 容器上。

18.5.1 创建项目的目录结构

portlet 项目的最基本的几个目录是：

HelloWorld\JavaSource 放置 Java 源代码；

HelloWorld\WebContent\Web-INF\classes：放置 Java Class 文件；

HelloWorld\WebContent\Web-INF\lib：放置 jar 文件，如 jstl.jar 、standard.jar （JSTL - JSP Standard Tag Library 及 Apache 的 JSTL 的实现）；

HelloWorld\WebContent\Web-INF\tld：放置 taglib 定义文件，如 portlet.tld 或 portlet.tld（portlet JSP tag）。这些目录下面的 jar 文件和 tld 文件可以从安装好的 JetSpeed 中找到。

18.5.2 创建 Portlet Java 代码

下面是 HelloWorldPortlet.java 的代码。注意：

（1）import 语句，这里使用的全部是 java 或 javax 标准类库，说明这个 portlet 代码应该是可以运行在支持相应标准的服务器上面。

（2）对于一些常量，使用了 public static final 修饰符，有助于提供 Java 代码的性能。

（3）processAction 方法是 Portlet 的核心方法之一，例子代码在这里处理 JSP 中 FORM 表单提交的数据，并把得到的数据放到一个 Java Bean 中，该 Java Bean 又被放到 PortletSession 中供 JSP 文件调用。

proccessAction 处理完毕后，portlet 引擎会运行 portlet 的 doView 方法。doView 方法根据逻辑、输入数据或者配置，调用不同的 JSP 文件进行数据展示。

代码如下：

```
package com.ibm.spc;

import java.io.*;

import javax.portlet.*;

/**
 *
 * A sample portlet based on GenericPortlet
 *
 */
public class HelloWorldPortlet extends GenericPortlet {

 public static final String JSP_FOLDER = "/com_ibm_spc/jsp/";  // JSP folder name
 public static final String VIEW_JSP  = "HelloWorldPortletView";  // JSP file
        //name to be rendered on the view mode
 public static final String VIEW_BEAN  = "HelloWorldPortletBean";  // Bean name
        //for the view mode request
 public static final String SAY_HELLO_ACTION = "Say_Hellow_Action";  // Action
        //name for submit form
 public static final String YOUR_NAME  = "YourName";  // Parameter name for the
        //text input

 /**
  * Serve up the <code>view</code> mode.
  *
  * @see javax.portlet.GenericPortlet#doView(javax.portlet.RenderRequest,
  * javax.portlet.RenderResponse)
  */
 public void doView(RenderRequest request, RenderResponse response) throws
        PortletException, IOException {
  // Set the MIME type for the render response
  response.setContentType(request.getResponseContentType());

  // Invoke the JSP to render
  PortletRequestDispatcher rd = getPortletContext().getRequestDispatcher
         (getJspFilePath(request, VIEW_JSP));
  rd.include(request,response);
 }

 /**
  * Process an action request.
  *
  * @see javax.portlet.Portlet#processAction(javax.portlet.ActionRequest,
  * javax.portlet.ActionResponse)
  */
 public void processAction(ActionRequest request, ActionResponse response)
        throws PortletException, java.io.IOException {
  if( request.getParameter(SAY_HELLO_ACTION) != null ) {
   // Make a session bean
```

```
    PortletSession session = request.getPortletSession();
    HelloWorldPortletBean viewBean = new HelloWorldPortletBean();
    session.setAttribute(VIEW_BEAN, viewBean);

    System.out.println("debug HelloWorld " + request.getParameter(YOUR_NAME));

    // Set form text in the view bean
    viewBean.setFormText(request.getParameter(YOUR_NAME));
   }
  }

  /**
   * Returns JSP file path.
   *
   * @param request Render request
   * @param jspFile JSP file name
   * @return JSP file path
   */
  private static String getJspFilePath(RenderRequest request, String jspFile) {
   String markup = request.getProperty("wps.markup");
   if( markup == null )
    markup = getMarkup(request.getResponseContentType());
   return JSP_FOLDER+markup+"/"+jspFile+"."+getJspExtension(markup);
  }

  /**
   * Convert MIME type to markup name.
   *
   * @param contentType MIME type
   * @return Markup name
   */
  private static String getMarkup(String contentType) {
   if( "text/vnd.wap.wml".equals(contentType) )
    return "wml";
   return "html";
  }

  /**
   * Returns the file extension for the JSP file
   *
   * @param markupName Markup name
   * @return JSP extension
   */
  private static String getJspExtension(String markupName) {
   return "jsp";
  }

}
```

18.5.3　创建 JSP

JSP 文件中首先声明它不需要创建新的 HTTP Session，返回页面的内容是 HTML 页面；然后 import 声明需要引用标准 Java 类库 java.util、javax.portlet 以及我们自己的类库 com.ibm.spc（因为我们的 HelloWorldPortlet 类文件是在 com.ibm.spc 包中，读者也可以根据自己的喜好选择 HelloWorldPortlet 类所放置的包）；接着声明使用 portlet 标记库，<portlet:defineObjects/>使用 portlet 标记库的标记 defineObjects，定义了 JSP 中要使用的 3 个变量：

```
RenderRequest renderRequest
RenderResponse renderResponse
PortletConfig portletConfig
```

JSP 的代码如下：

```
<%@ page session="false" import="java.util.*,javax.portlet.*,com.ibm.spc.*" %>
<%@taglib uri="http://java.sun.com/portlet" prefix="portlet" %>
<portlet:defineObjects/>
```

接下来，从<portlet:defineObjects/>语句定义的变量 renderRequest 当中获取 PortletSession，进而得到 session 当中保存的数据并显示在 JSP 页面上。

```
 <%
 PortletSession session = renderRequest.getPortletSession();
 HelloWorldPortletBean bean = (HelloWorldPortletBean)session.getAttribute
        (HelloWorldPortlet.VIEW_BEAN);
%>
 if (bean != null) {
  String formText = bean.getFormText();
  if( formText.length()>0 ) {
%>
  Hello <%=formText%>.
 <%
  }
 }
 %>
```

最后部分是使用 portlet 标记库的另一个标记 actionURL 产生一个 URL 指向当前页面中的这个 portlet，生成的 URL 能够触发当前 portlet 的 action 请求，或者说这个 URL 能够触发当前 portlet 的 processAction 方法。

```
      <FORM method="POST" action="<portlet:actionURL/>">
      <LABEL  for="<%=HelloWorldPortlet.YOUR_NAME%>">Please input your name here,
</LABEL><BR>
      <INPUT name="<%=HelloWorldPortlet.YOUR_NAME%>" type="text"/>
      <INPUT    name="<%=HelloWorldPortlet.SAY_HELLO_ACTION%>"    type="submit"
value="Submit"/>
     </FORM>
```

18.5.4　编译 portlet

编写好 portlet 的 Java 代码后就可以把它编译成二进制 class 文件了。下面的脚本中使用 JAVA_HOME 环境变量指向 WebSphere Application Server 5.0.2 中的 IBM JDK 1.3.1（读者可以根据自己的系统进行设置，设置为本系统的 JDK 安装路径）。脚本中使用 CP 变量指向 Tomcat

4.1 中带的 Servlet 2.3 类库，以及 Pluto 的 JSR 168 portlet 类库。脚本最后的动作是编译 HelloWorld portlet，并把编译好的 class 文件放到 WebContent\WEB-INF\classes 目录。

```
set JAVA_HOME=C:\WebSphere\AppServer\java
set PATH=%JAVA_HOME%\bin
set tomcat.home.pluto=e:\ApacheSoftwareFoundation\Tomcat4.1
set CP=.
rem Servlet 2.3 API jar file
set CP=%CP%;%tomcat.home.pluto%\common\lib\servlet.jar
rem JSR 168 API jar file
set CP=%CP%;%tomcat.home.pluto%\shared\lib\portlet-api.jar
rem Specify where to place generated class files
set target_path=..\WebContent\WEB-INF\classes
cd JavaSource
javac -classpath %CP% -d %target_path% com\ibm\spc\*.java
```

18.5.5 创建 Web 应用的部署描述文件

Portlet 应用也是一个 J2EE Web 应用，拥有一个 Web 应用部署描述文件 web.xml。web.xml 文件中 taglib 标记部分是关于 Portlet Tag Library 的定义，在 Portlet 应用的 JSP 文件中可以使用这种 Tag Lib。

下面的代码片断声明使用 URI 是 http://java.sun.com/portlet 的 tag lib，tag lib 的前缀是 portlet。关于 Portlet Tag Library 请参考 Java Portlet Specification。

注意：2004 年 2 月的 pluto 中 portlet 部署程序中不能分析处理 web.xml 文件中 welcome-file 的标记，相信 Apache 会在后继的版本中修正这个问题。解决办法是，从 web.xml 文件中去除有关的 tag；或者修改 pluto 代码，为 servletdefinitionmapping.xml 文件添加 welcome-file 标记，为 org.apache.pluto.portalImpl.om.servlet.impl. WebApplicationDefinitionImpl java 类添加一个字段来解决这个问题。

```
<?xml version="1.0" encoding="UTF-8"?>
<!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 id="WebApp">
 <display-name>HelloWorld Web Application</display-name>
 <taglib id="PortletTLD">
  <taglib-uri>http://java.sun.com/portlet</taglib-uri>
  <taglib-location>/WEB-INF/tld/portlet.tld</taglib-location>
 </taglib>
</web-app>
```

18.5.6 创建 Portlet 部署描述文件

每个 Portlet 应用除了 Web 应用部署描述文件 web.xml 外，还有一个 Portlet 部署描述文件 portlet.xml。该文件中包括该 Portlet Application 中一个或多个 portlet 的定义。

下面的 portlet.xml 文件中首先是<portlet-app/>，其中引用了 SUN 公司的关于 portlet 描述文件的名字空间的定义文件 portlet-app_1_0.xsd；然后是各个<portlet/>定义，包括名字和描述信息、国际化的名字和描述信息、portlet 的 class 类名、portlet 的初始化参数、国际化用户界面中使用的资源文件。HelloWorld Portlet 有一个初始化参数 wps.markup。在我们的 portlet 代

码中使用 renderRequest.getProperty("wps.markup")获得这个初始化参数的值。

```
<?xml version="1.0" encoding="UTF-8"?>
<portlet-app xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
        version="1.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schema- Location="http://java.sun.com/xml/ns/portlet/portlet-
        app_1_0.xsd http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd">
    <portlet>
        <description>A simple HelloWorld portlet</description>
        <description xml:lang="en">A simple HelloWorld portlet</description>
        <portlet-name>HelloWorldPortlet</portlet-name>
        <display-name>HelloWorld portlet</display-name>
        <display-name xml:lang="en">HelloWorld portlet</display-name>
        <portlet-class>com.ibm.spc.HelloWorldPortlet</portlet-class>
        <init-param>
            <name>wps.markup</name>
            <value>html</value>
        </init-param>
        <expiration-cache>0</expiration-cache>
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
        </supports>
        <supported-locale>en</supported-locale>
        <resource-bundle>com.ibm.spc.nl.HelloWorldPortletResource
             </resource-bundle>
        <portlet-info>
            <title>HelloWorld portlet</title>
        </portlet-info>
    </portlet>
</portlet-app>
```

18.5.7　创建 war 文件

使用 JDK 的命令 jar 把 class 文件、JSP 文件、jar 包、JSP 标记库、Web 部署描述文件 web.xml、portlet 部署描述文件 portlet.xml 等打包成 web archive 文件。

```
set JAVA_HOME=C:\WebSphere\AppServer\java
set PATH=%JAVA_HOME%\bin
cd WebContent
jar cf ..\build\HelloWorld.war .
```

18.5.8　发布和调用

将打包生成的 HelloWorld.war 文件拷贝到/jetspeed/WEB-INF/deploy 目录下即可。系统会自动部署到 Portal 中。

在页面编辑状态下，点击新增 Portlet 会出现“Portlet 选择器”，里面会新增一个“HelloWorld portlet”应用，选中后确定就可将其添加到页面中。

除此之外还有个更简单的办法可以显示 portlet。在/jetspeed/WEB-INF/pages 目录下的 default-page.psml 文件中增加一个 fragment 定义。

```
<fragment id="dp-19" type="portlet" name="HelloWorld::HelloWorldPortlet">
    <property name="row" value="6"/>
    <property name="column" value="0"/>
</fragment>
```

其中：

id 为任意项；

name 为${portlet.application.id}和${portlet.name}的组合；

${portlet.application.id}是实际的 war 文件名；

${portlet.name}为在 portlet.xml 中定义的<portlet-name>；

row，column 为页面中的位置。

18.5.9 错误及解决方法

将应用部署到页面后发现不能正常显示，报 Portlet is Not Available 错误。经过对日志分析，原来 jetspeed2 不支持 PropertyManager 容器服务。在代码里使用了 renderRequest.getProperty ("wps.markup")获得这个初始化参数的值。

修改程序通过初始化来获得参数的值：

```
private static String defaultMarkupSource = "";

    /**
    * Initialize portlet configuration.
     */
    public void init(PortletConfig config) throws PortletException
    {
      super.init(config);

      defaultMarkupSource = config.getInitParameter("wps.markup");
    }

private static String getJspFilePath(RenderRequest request, String jspFile)
    {
      //String markup = request.getProperty("wps.markup");
      String markup = defaultMarkupSource;

      if (markup == null)
        {
          markup = getMarkup(request.getResponseContentType());
        }

      return JSP_FOLDER + markup + "/" + jspFile + "." +
      getJspExtension(markup);
    }
```

重新编译部署，应用即可正常显示。

18.5.10 运行结果

启动 Tomcat 就可以看到 JSR 168 portlet 了，如图 18-6 所示。

在页面中输入 Li Ping，将看到运行结果如图 18-7 所示。

图 18-6　HelloWorld Portlet

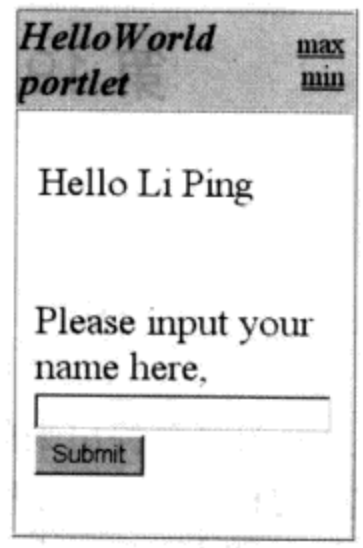


图 18-7　Portlet 运行结果

18.6　小结

本章介绍了门户中间件的基本原理和开发实例，包括如下内容：

（1）介绍了门户系统的定义、功能和分类。

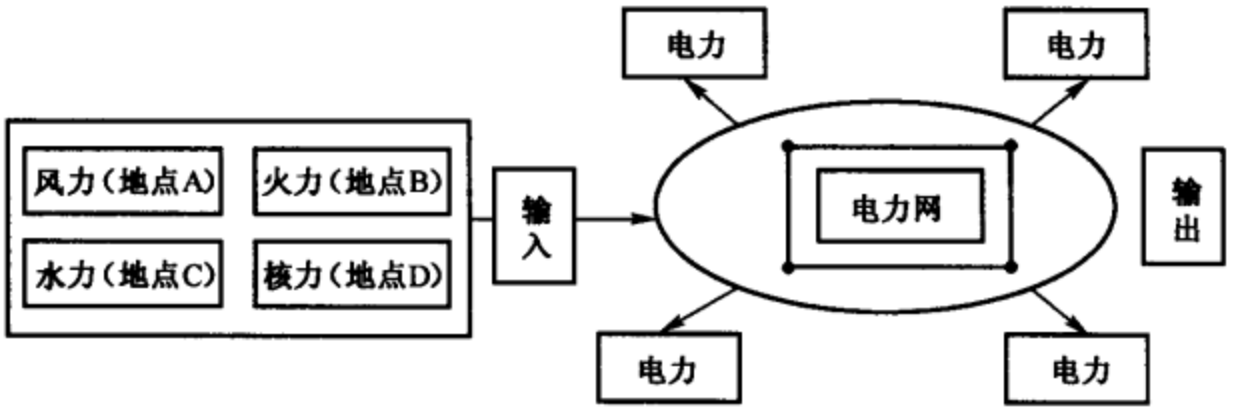
（2）介绍了门户中间件和相应的概念：门户中间件、门户服务器（Portal Server）、Portal、Portlet 容器（Portlet Container）和 Portlet。介绍了 Portal 的实现原理和 Portlet 的开发方法。

（3）介绍了 JetSpeed 项目，JetSpeed 的安装配置。然后，开发了一个 Hello World Portlet 程序，并部署到 JetSpeed 平台上。

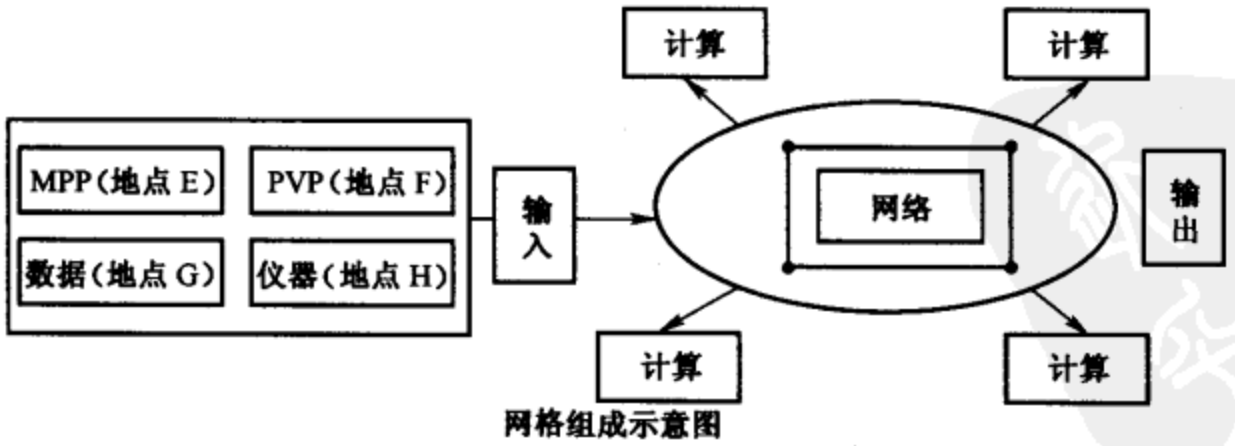
第 19 章　网格中间件

19.1　网格计算的定义

网格是借鉴电力网的概念提出来的新概念，代表了一种先进的技术和基础设施，是继 Internet 后又一次重大的科技进步。而所谓的网格指的是一个集成的计算与资源环境，或者说是一个计算资源池，它能够充分吸纳各种计算资源，并将它们转化成一种随处可得的、可靠的、标准的同时还是经济的计算能力。这里的计算资源还包括了网络通信能力、数据资源、仪器设备、甚至是人等各种相关资源。网格的最终目的是希望用户在使用网格计算能力时，就如同现在使用电力一样方便，不需要知道它是从哪个地点的发电站输送出来的，也不需要知道该电力是通过什么样的发电机产生的，而是一种统一形式的“电能”。网格也希望给最终的使用者提供与地理位置无关、与具体的计算设施无关的通用的计算能力。可以通过图 19-1 简单对比两者的关系。



电力网构成示意图



网格组成示意图

图 19-1　电力网与网格的对比关系

那么，什么又是网格计算（Grid Computing）呢？从广义概念上讲，网格计算即基于网格的问题求解。而从狭义概念上来讲，网格计算就是指将分布的计算机组织起来协同解决复杂的科学和工程计算问题。而对于狭义的网格概念而言，网格一般被称为计算网格，

即网格是主要用来解决科学与工程计算方面的问题的网格。但网格并不是只有计算网格这一种，依据求解问题的特点，人们又提出了科学网格、核心数据网格、地球系统网格、军事网格等行业网格。

对于网格的认识普遍存在这样两种观点，一种观点认为网格就是仅仅通过网络把计算机、人、仪器、数据等连起来，而没有将它们作为一个有机的统一整体来看待，过分强调物理的网络和离散的网格资源是一种过时的观点；第二种观点就是把网格看做中间件系统，这种观点也是不全面的，中间件在网格中占有重要的地位，但是将网格就看做中间件系统也是不全面的，这两种观点都具有一定的片面性，第一种观点过分强调了网格物理上的资源组成，而第二种观点过分强调网格逻辑上的功能，需要把两者结合起来才是完整的网格体系，物理资源本身和对物理资源的管理与逻辑上的抽象都是十分重要的，而且两者也是密不可分的，它们是网格环境中的两大核心要素。它们之间的关系如图 19-2 所示。

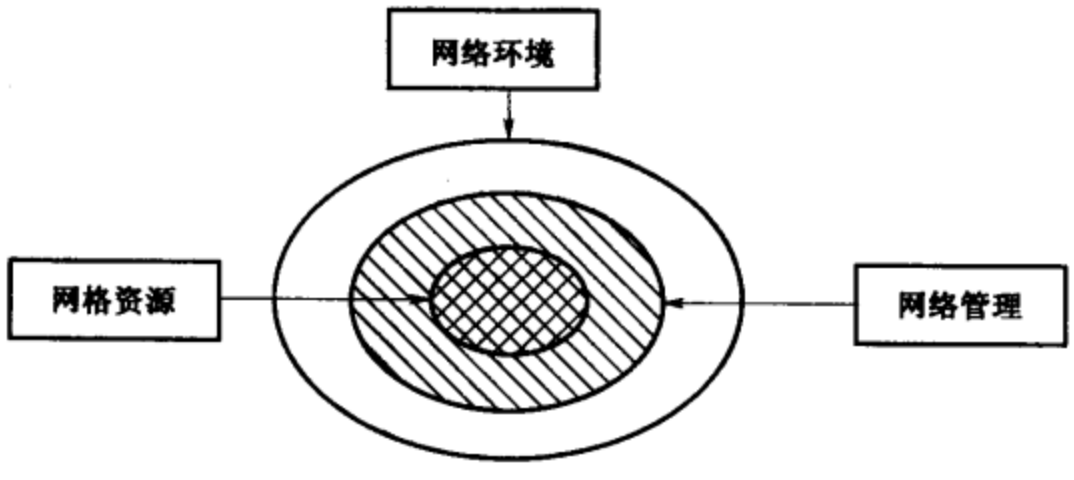


图 19-2　网格环境、资源与管理之间的关系

对于网格计算概念的理解，我们可以拿它和传统意义上的一些概念进行类比来做进一步的阐述。如果说我们把因特网看做通信网络的话，那么网格计算可以被认为是计算网络，是用于合用资产之间协调资源共享和问题解决的工具和协议，这些合用资产（一般又称合用资产为虚拟组织）可以分布在世界各地，它们是异构的（有些是 PC，有些是服务器，也许有些是大型机和超级计算机），在某种程度上是自主的（网格可能访问不同组织的资源），而且是临时的。

19.2　网格中间件

19.2.1　网格组成与网格中间件的作用

网格环境的构建层次如图 19-3 所示，主要由资源、中间件、工具软件和应用程序等几部分组成。其中资源由分布在 Internet 上的各类资源组成，包括各类主机、工作站甚至 PC 机，也可以是上述机型的机群系统、大型存储设备、数据库或其他设备。中间件是网格计算的核心，负责提供远程管理、资源分配、存储访问、登录和认证、安全性和服务质量（QoS）等。工具软件和应用程序提供用户二次开发利用的环境、工具、语言及接口等，以便更好地利用网格资源。

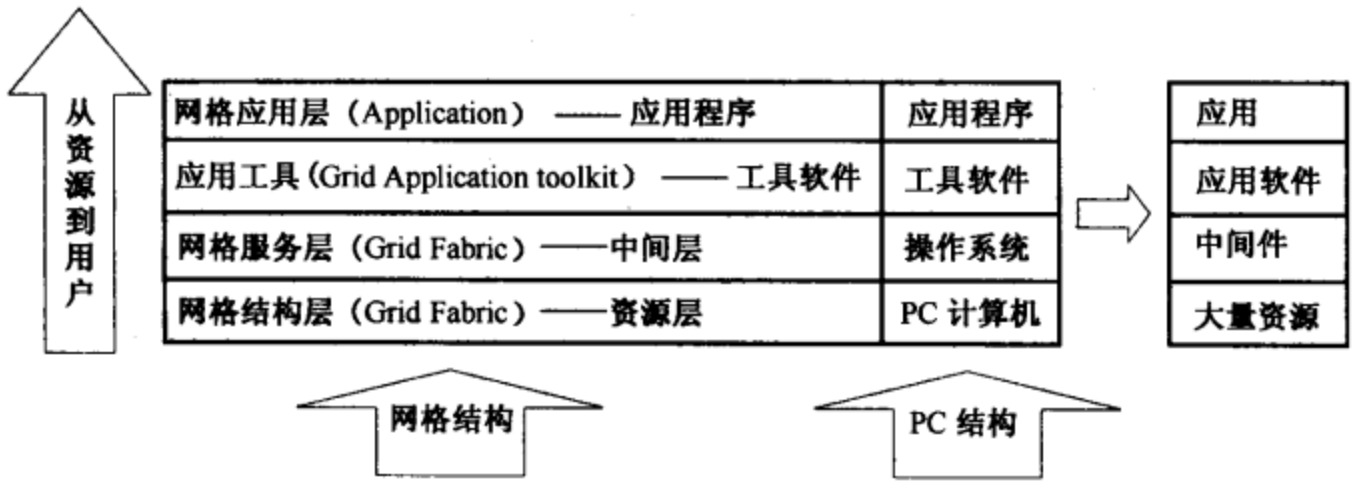


图 19-3 网格组成

19.2.2 开放网格服务结构 OGSA

目前，比较重要的网格体系结构有两个：一个是伊安·福斯特（Ian Foster）等在早些时候提出的五层沙漏结构；另一个是以 IBM 为代表的工业界的影响下，在考虑到 Web 技术的发展与影响后，伊安·福斯特（Ian Foster）等提出的开放网格服务结构 OGSA（Open Grid Services Architecture）。本书介绍开放网络服务结构 OGSA。

开放式的网格服务体系 OGSA 是一个由节点和连线构成的框架。该框架的节点是网格服务，而网格服务之间的连线是网格服务相互交流时所用的语言。网格服务是特殊的网络服务，专供用来维持和管理网格体系。

OGSA 具有下列目标：

（1）跨分布式异构平台管理资源。

（2）交付无缝的服务质量（Quality of Service，QoS）。网格的拓扑结构通常十分复杂，而且网格资源的交互往往是动态的。有一点很重要，即网格可以提供健壮的后台服务，如授权、访问控制和委托。

（3）为自治管理解决方案提供公共基础。网格可以包含许多资源，还有大量的配置组合、交互以及状态与故障模式的改变。对于这些资源来说，一些智能自动调节与自治管理方式是必不可少的。

（4）定义开放的、已公布的接口。OGSA 是一种由 GGF 标准团体进行管理的开放式标准。为了不同资源的互操作性，网格必须构建在标准接口及协议之上。

（5）利用行业标准的集成技术。OGSA 的创始者很有远见地利用了现有解决方案。OGSA 的基础是 Web 服务。

OGSA 架构由四个主要的层构成，如图 19-4 所示。

从下到上依次为：

（1）资源：物理资源和逻辑资源。资源的概念是 OGSA 以及通常意义上的网格计算的中心部分。构成网格能力的资源并不仅限于处理器，物理资源包括服务器、存储器和网络。

物理资源之上是逻辑资源。它们通过虚拟化和聚合物理层的资源来提供额外的功能。通用的中间件，如文件系统、数据库管理员、目录和工作流管理人员，在物理网格之上提供这些抽象服务。

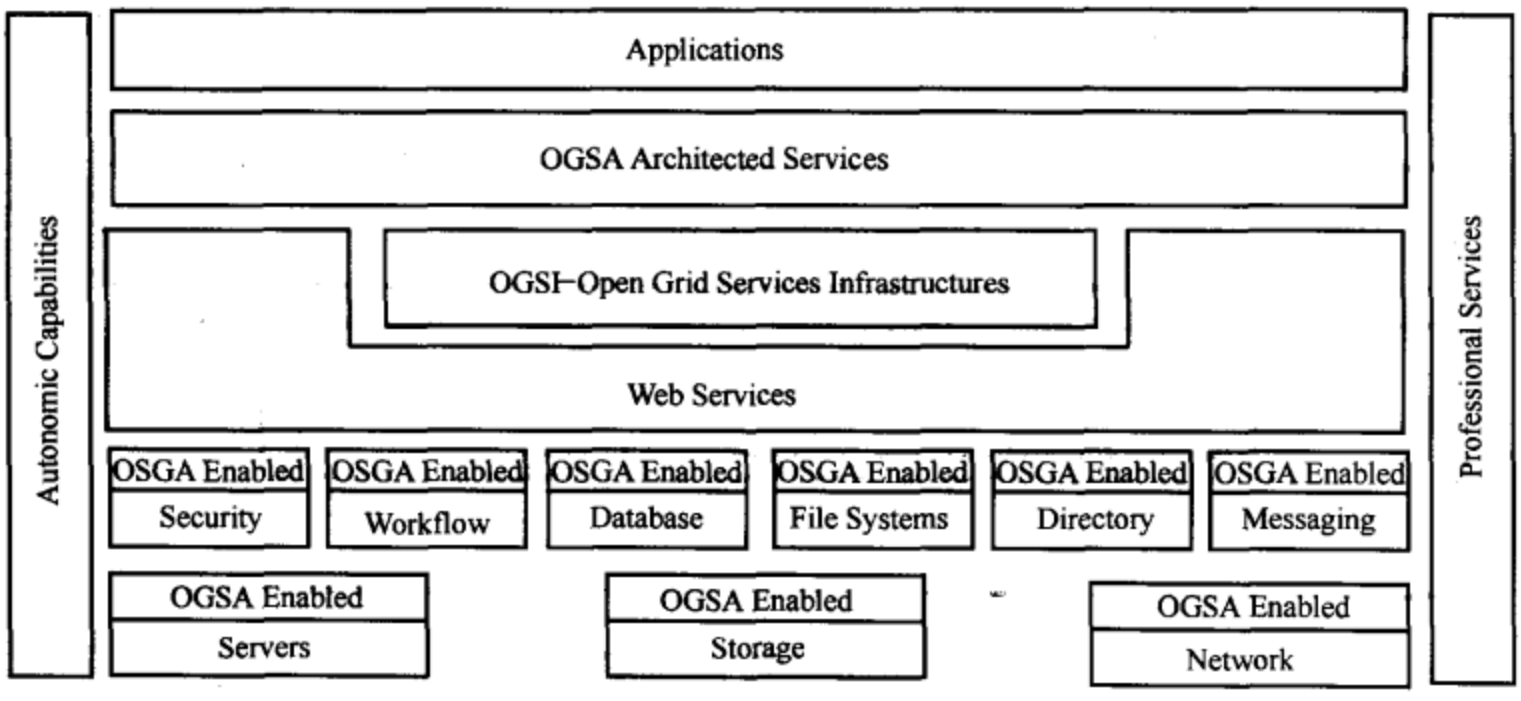


图 19-4　OGSA 的主要架构

（2）Web 服务以及定义网格服务的 OGSI（Open Grid Services Infrastructure，开放网格服务基础设施）扩展。OGSA 架构中的第二层是 Web 服务。 这里有一条重要的 OGSA 原则：所有网格资源（逻辑的与物理的）都被建模为服务。OGSI 规范定义了网格服务并建立在标准 Web 服务技术之上。OGSI 利用诸如 XML 与 Web 服务描述语言（Web Services Description Language，WSDL）这样的 Web 服务机制，为所有网格资源指定标准的接口、行为与交互。OGSI 进一步扩展了 Web 服务的定义，提供了动态的、有状态的和可管理的 Web 服务的能力，这在对网格资源进行建模时都是必需的。

（3）基于 OGSA 架构的服务。Web 服务层及其 OGSI 扩展为下一层提供了基础设施：基于架构的网格服务。GGF（Global Grid Forum，全球网格论坛）目前正在致力于在诸如程序执行、数据服务和核心服务等领域中定义基于网格架构的服务。随着这些新架构的服务开始出现，OGSA 将变成更加有用的面向服务的架构（SOA）。

（4）网格应用程序。随着时间的推移，一组丰富的基于网格架构的服务不断被开发出来，使用一个或多个基于网格架构的服务的新网格应用程序也将出现。这些应用程序构成了 OGSA 架构的第四个主要的层。

OGSA 结构较五层沙漏结构有着以下特点：

（1）以服务为中心的模型。如果说五层沙漏结构是以协议为中心的“协议结构”，其试图实现的是对资源的共享，那么 OGSA 就是以服务为中心的“服务结构”，其实现的是对服务的共享。OGSA 将一切看做服务，并定义了“网格服务”，该服务提供了一组接口，这些接口遵守特定的惯例，解决服务发现、动态服务创建、生命周期管理、通知等问题。因此，网格是可扩展的网格服务的集合。简单地说，网格服务=接口/行为+服务数据。

（2）统一的 Web Service 框架。Web Service 描述了一种新出现的、重要的分布式计算范式，定义了一种技术，用于描述被访问的软件组件、访问组件的方法以及找到相关服务提供者的发现方法，解决了发现和激发永久服务的问题。OGSA 是符合标准的 Web Service 框架的。但是在网格中，大量的是临时服务，因此 OGSA 对 Web Service 进行了扩展，提出的是网格服务（Grid Service）的模仿，使得它可以支持临时服务实例，并且能够支持创建和删除。

下面来看 OGSA 的两个主要逻辑组件：Web 服务加 OGSI 层和基于 OGSA 架构的服务层，如图 19-5 所示。

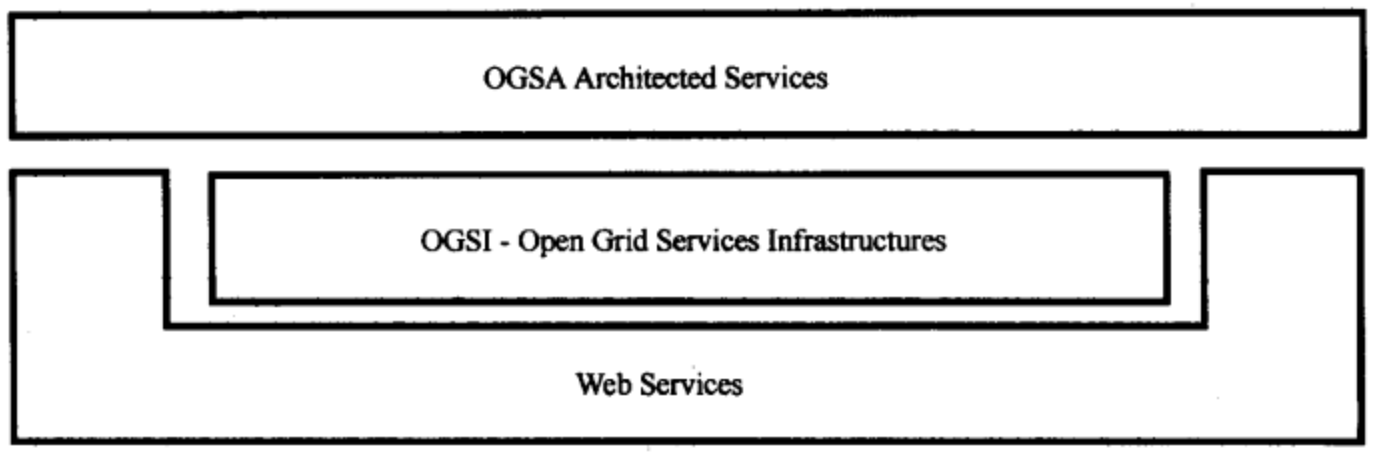


图 19-5　OGSA 的两个主要逻辑组件

为什么它们看起来像是相互独立的呢？因为 GGF OGSA 工作小组相信，通过增加核心 Web 服务功能来满足网格服务需求是非常必要的。OGSI 通过在以下两个领域引入接口和约定来扩展 Web 服务：

第一，网格中服务具有动态及可能瞬变的特性。在网格中，特殊的服务实例会随着工作的分派、资源的配置与供给以及系统状态的变化而不断地产生和销毁。因此，网格服务需要接口来管理它们的创建、销毁以及生命周期管理。

第二，就是状态。网格服务可以拥有与自身相关的属性和数据。这在概念上类似于面向对象编程中对象的传统结构。对象有其行为和数据。同样地，Web 服务需要得到扩展，从而支持与网格服务相关的状态数据。

19.2.3　Web 服务的交互模型

OGSI 引入了一种网格服务的交互模型。通过提供发现、生命周期、状态管理、创建与销毁、事件通知以及引用管理的接口，OGSI 为软件开发人员提供了一种统一的建模和与网格服务进行交互的方式。图 19-6 描述了这些接口。不论开发人员正在开发的是网格服务还是应用程序，OGSI 编程模型都会为网格软件提供一种一致的交互方式。

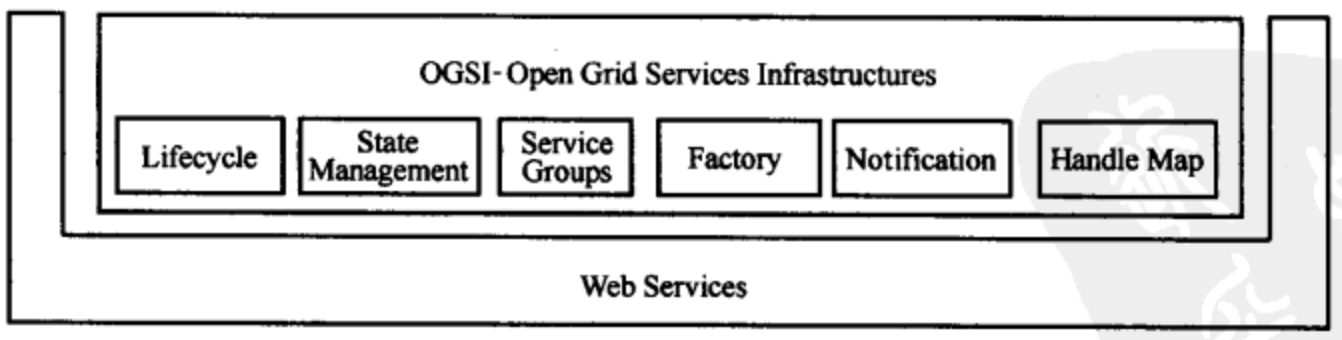


图 19-6　OGSI 组件

下面介绍 OGSI 引入的接口和约定。

（1）工厂。实现此接口的网格服务提供了一种创建新网格服务的方式。工厂可以创建有限功能的临时实例，比如创建服务以表示执行特殊任务的调度程序，或者它们可能创建生存时间更长的服务，比如一个常用数据集的本地拷贝。并不是所有网格服务都是动态创建的，其中

有些可能是作为网格中物理资源的实例被创建的，如处理器、存储器或网络设备。

（2）生命周期。因为网格服务可能是瞬变的，所以网格服务实例是使用指定的生命周期来创建的。用户可以根据依赖或管理该服务的组件的需要，确定和扩展任何特殊服务实例的生命周期。生命周期机制是以这样一种方式构架的，即在不需要大规模分布式垃圾收集清理程序的情况下，防止网格服务无限地消耗资源。

（3）状态管理。网格服务可以具有状态。OGSI 规定了一个用于表示这种状态的名为 Service Data 的框架，以及一个用于检查或修改该状态的名为 Find/SetServiceData 的机制。此外，OGSI 要求每个网格服务都必须支持的 Service Data Elements 中要有最低数量的状态，并要求所有服务都要实现 Find/SetServiceData portType。

（4）服务组。服务组是网格服务的集合，它们使用 Service Data 来建立索引以用于特定目的。例如，用户可能使用它们来收集所有这样的服务，它们在网格内特定群集节点中代表资源。

（5）通知。被建模用于网格服务的状态信息（Service Data）会随着系统的运行而变化。网格服务之间的许多交互要求动态地监控状态变化。通知把一种传统的发布和订阅范式应用于这种监控。网格服务支持一个接口（NotificationSource），以允许其他网格服务（NotificationSink）订阅进行变更。

（6）HandleMap。当工厂创建网格服务的一个新实例时，工厂会返回新实例化的服务标识。这个标识由两部分组成：一个网格服务句柄（Grid Service Handle，GSH）和一个网格服务引用（Grid Service Reference，GSR）。GSH 保证无限期地引用该网格服务，而 GSR 可以在该网格服务的生命周期内发生改变。HandleMap 接口提供一种在给定 GSH 的情况下获得 GSR 的途径。

19.3　网格中间件项目 Globus 简介

Globus 项目是目前国际上最有影响的网格计算相关的项目之一。它发起于 20 世纪 90 年代中期，其最初的目的是希望把美国境内的各个高性能计算机中心通过高性能网络连接起来，方便美国的大学和研究机构使用，提高高性能计算机的使用效率。当时在美国建立了一个试验环境—I-WAY，它把位于美国防大 17 个不同地点的 60 多个组织的超级计算机和资源通过高性能网络联系起来，进行大规模科学模拟、协同工作、并行计算等科学研究，这实际上是 Globus 项目的前身。随着对 Globus 项目的深入研究，针对它的目标也进一步扩展，希望通过 Globus 项目可方便地对地理上分布的研究人员建立虚拟组织，进行跨学科的虚拟合作。目前，Globus 项目把在商业计算领域中 Web Service 技术融合在一起，希望不仅仅局限于科学计算领域，而且能够对各种商业应用进行广泛的、基础性的网格环境支持，实现更方便的信息共享和互操作，从而对商业模式、人的工作方式和生活方式产生深远的影响。

Globus 项目是美国 Argonne 国家实验室的研发项目，在初始阶段，全美有十多所大学和研究机构参与了该项目的研究工作。Globus 对信息安全、资源管理、信息服务、数据管理以及应用开发环境等网格计算的关键理论和技术进行了广泛的研究，开发出能在多种平台下运行的网格计算软件工具包（Globus Toolkit），能够用来帮助规划和组建大型的网格试验和应用平台，开发适合大型网格系统运行的大型应用程序。Globus 工具包是 Globus 最重要的实践成果，Globus Toolkit 工具包的第二版（GT2）成为了网格计算的事实标准。它着重于可用性和互操

作性能力，定义和实现了一些协议、API 和服务。

2002 年 2 月，在加拿大多伦多市召开的全球网格论坛 GGF 会议上，Globus 项目组和 IBM 共同倡议了一个全新的网格标准 OGSA。OGSA 是开放网格服务体系，它把 Globus 标准与以商用为主的 Web Services 的标准结合起来，网格服务统一以 Services 的方式对外界提供。

2003 年符合 OGSA 规范的 Globus Toolkit 3.0（GT3）发布，这标志着 OGSA 已经从一种理念、一种体系结构，走到付诸实践的阶段了。GT3 提供了一个完整的开放网格服务基础设施（OGSI）实现，它的许多功能重构成与 OGSI 兼容的服务。GT3 将服务发现、程序执行作业的提交、监控和可靠的文件传输，定义成了与 OGSI 兼容的服务。其他如数据传递、副本定位和授权等服务也尽量构建成与 OGSI 相兼容。并且 GT3 定义了一组关于使用 Web 服务描述语言（WSDL）和扩展标识语言（XML）模式的约定与扩展，以便启用有状态服务。

2004 年 1 月，美国 Akamai Technologies、The Globus Alliance、惠普、IBM、Sonic Software 和 TIBCO Software 六家公司公布了统一网格计算和 Web 服务的新标准 WS-Notification 和 WS-Resource Framework。Web 服务资源框架（WSRF）是 OGSI 的重构和发展，利用新的 Web 服务标准。WSRF 基本保留了 OGSI 中的所有功能，同时更改了一些语法，还在其表示中采用了不同的技术。Web 服务通知（WSN）为 Web 服务提供基于消息的发布和预定能力。WSRF 和 WSN 都是建立在已存在的 Web 服务定义和技术基础上的，帮助实现了网格计算、系统管理和 Web 服务的统一。

2005 年 1 月 31 日发布的 Globus Toolkit 4（GT4），实现了 WSRF 和 WSN 标准。GT4 提供 API 来构建有状态的 Web 服务，其目标是建立分布式异构计算环境。所有知名的 GT3 协议都被重新设计为可以使用 WSRF，并且 GT4 也在其中增添了一些新的 Web 服务的组件。

Globus Toolkit 的体系结构如图 19-7 所示，由 7 个模块组成，它提供一种“打包服务”（bag of services），也就是说，这些模块可以单独或联合使用来开发网格应用和编程工具。

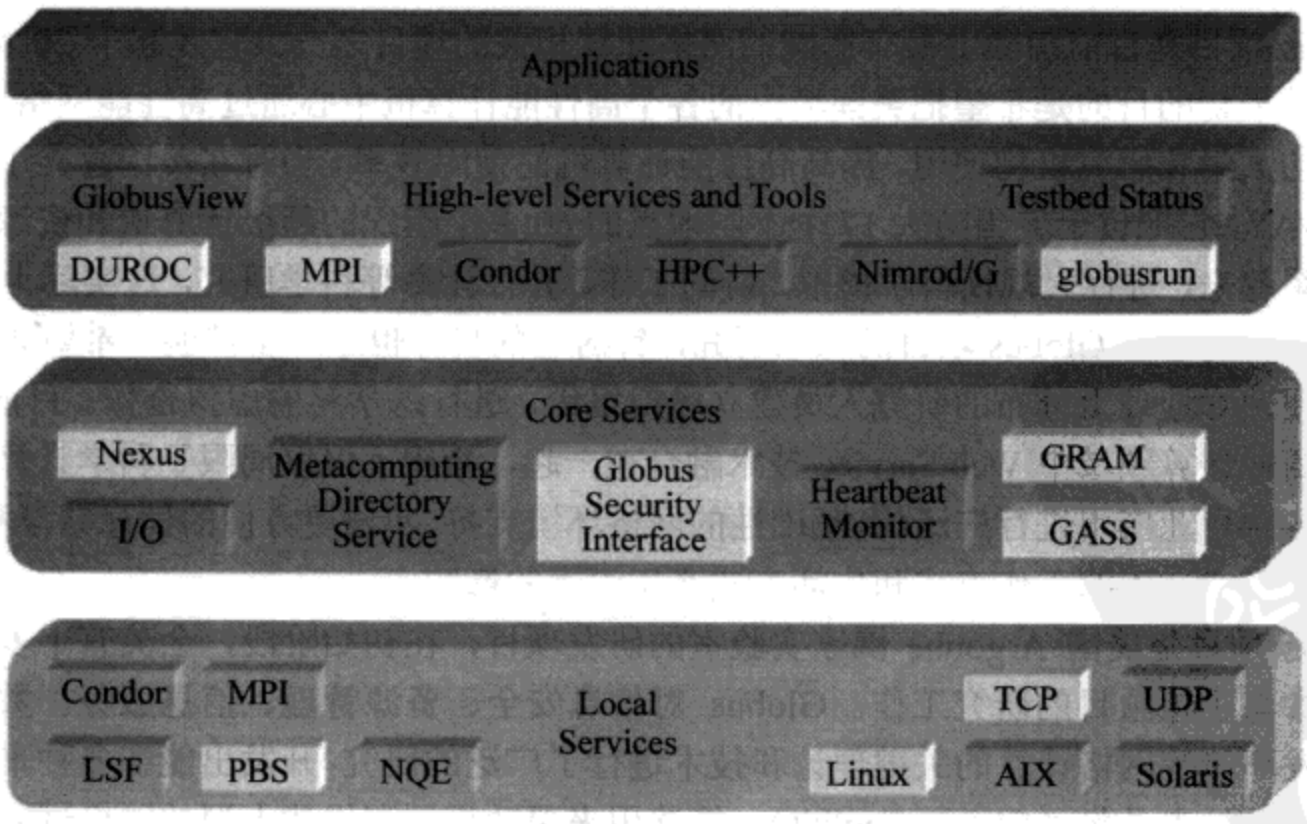


图 19-7 Globus Toolkit 体系结构

组成 Globus Toolkit 的 7 个模块如下：

（1）网格资源分配管理器（Globus Resource Allocation Manager，GRAM）

（2）网格安全架构（Grid Security Infrastructure，GSI）

（3）元计算目录服务（Metacomputing Directory Services，MDS）

（4）远程数据访问（Global Access to Secondary Storage，GASS）

（5）Globus Toolkit I/O

（6）Nexus

（7）Heartbeat Monitor（HBM）

下面逐一介绍这 7 个模块。

19.3.1　网格资源分配管理器 GRAM

如图 19-8 所示，网格资源分配管理器处于 Globus 资源管理体系结构的最底层，提供资源分配、创建进程、监控以及管理服务。GRAM 的下层是一些本地资源管理工具，目前 Globus 支持 6 个本地资源管理工具：NQE、EASY_LL、LSF、Load Leveler、Condor 和简单的 fork 语句。GRAM 的上层是一些特殊的资源代理和资源协同分配机制。GRAM 允许用户运行远程作业，并提供了一套 API 来递交、取消、监控以及终止作业。GRAM 提供给用户的界面是 Gatekeeper。当用户递交一个作业时，请求将被发送到远程计算机的 Gatekeeper 上。Gatekeeper 处理这个请求并为这个作业创建一个作业管理器（job manager），作业管理器启动并监控远程作业，并将运行状态信息发回至本地机器上的用户，当远程作业正常或非正常结束时，作业管理器相应地也会终止。

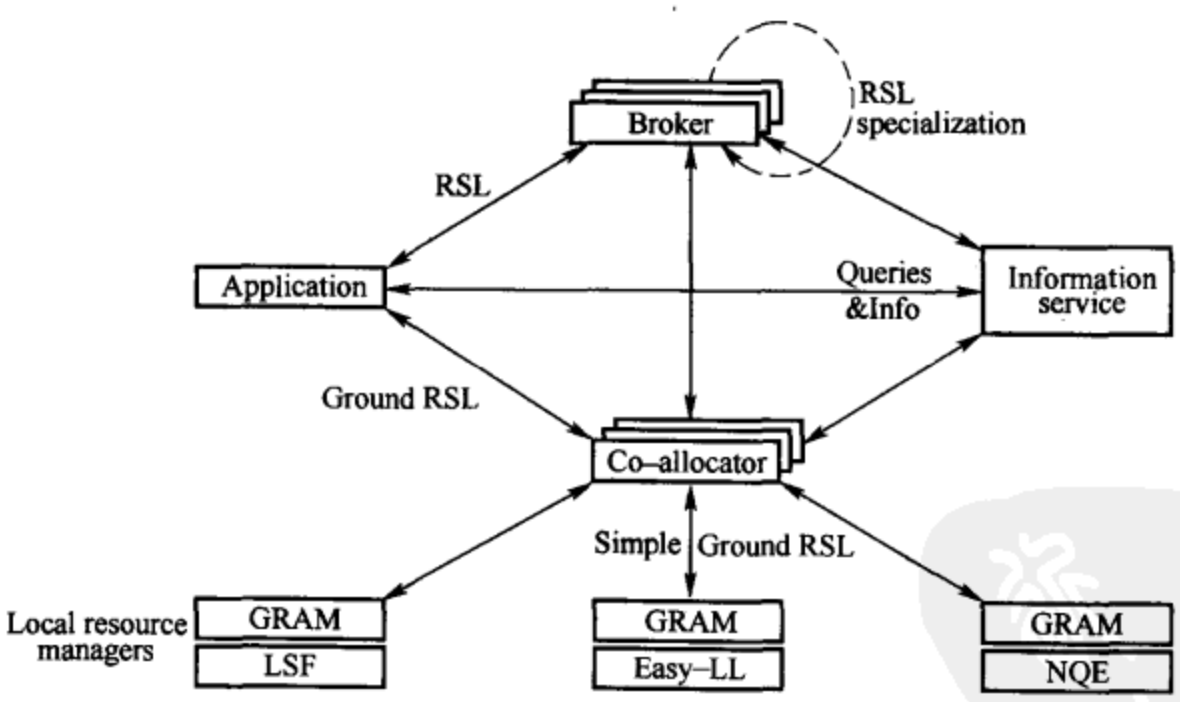


图 19-8　Globus 资源管理体系结构

GRAM 主要负责：

（1）解析并执行以资源规范语言（Resource Specification Language）描述的作业请求，作业请求包括资源选择、创建作业进程和作业控制。

（2）确保远程监控和管理作业已经建立。

（3）更新 MDS。

19.3.2 网格安全架构 GSI

由于网格要对多个组织所拥有和管理的高性能计算机、网络、数据库以及高性能仪器进行综合、协同使用，因此网格结构中的安全问题非常关键，关系到一个网格系统的成败。网格安全架构（Grid Security Infrastructure，GSI）是 Globus Toolkit 中负责安全的模块，它建立在公钥加密技术、X.509 证书以及安全套接字层通信协议（Secure Socket Layer Communication Protocol）之上，并在这些协议的基础之上增加了支持单一签名（Single Sign On）的功能。GSI 主要解决了认证问题，提供了认证用户和资源身份的认证机制。GSI 提供单一签名认证服务，并支持本地对访问的控制，将全局用户身份映射为本地用户身份。

19.3.3 元计算目录服务 MDS

元计算目录服务（Metacomputing Directory Services，MDS）是一个分布在由 Globus Toolkit 管理的资源上的综合信息服务，它提供了一个管理网格动态及静态信息的框架。MDS 主要的功能是对资源进行定位与测定。定位是指查找到某些资源，这些资源满足要执行的作业对操作系统版本、处理器类型、内存大小、网络带宽、负载程度以及所装软件等要求；测定是对资源的物理特性、连接性以及性能等方面进行确认。

MDS 主要建立在轻量目录协议（Lightweight Directory Access Protocol v3, LDAP）之上，MDS 的目录结构、数据描述以及 API 都是以 LDAP 定义的。

如图 19-9 所示，MDS 的体系结构主要由两个基本部分构成：高度分散的信息提供者和特定聚合目录服务（Specialized Aggregate Directory Services）。信息提供者构成了一个可以访问网格元素的详细及动态信息的公共架构。例如，一个计算资源的信息提供者可以提供该计算资源有多少个计算节点以及有多少个计算节点空闲、内存的总量以及可用的内存数，操作系统的版本以及平均负载量；一个正在运行的应用程序的信息提供者可以提供该应用程序的配置以及当前运行状态。特定聚合目录服务则主要提供从特定虚拟组织的观点来考察综合资源（Federated Resources）及服务。例如：目录 A 列出了某个虚拟组织拥有的、并装有某种操作系统的计算资源，而目录 B 则列出了对正在运行的应用程序进行监控的信息。

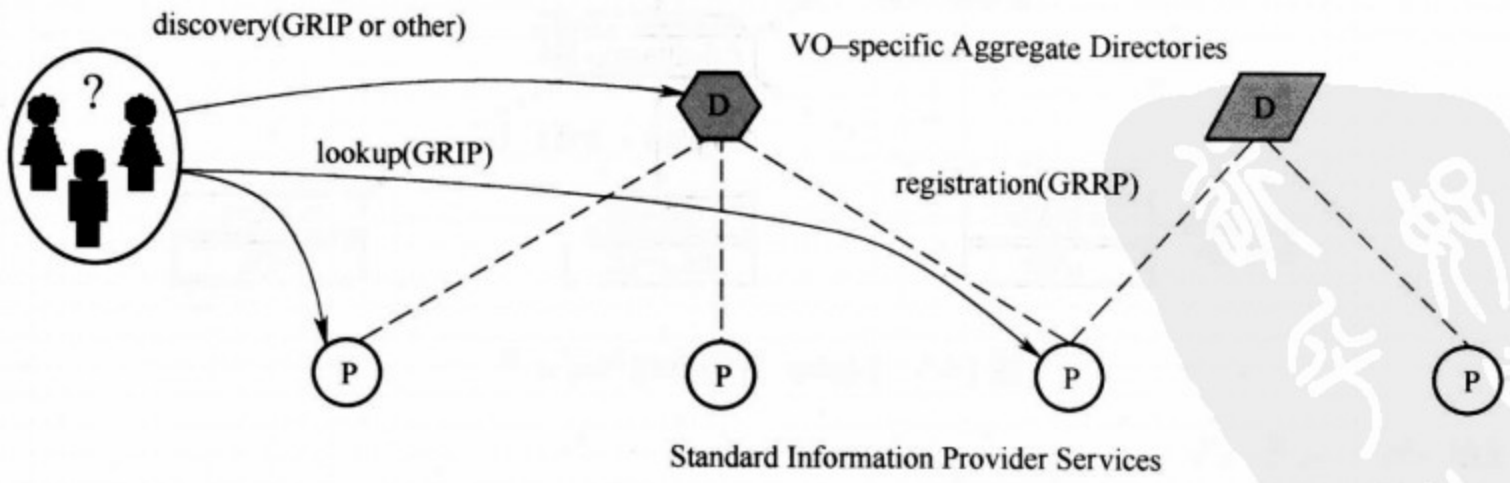


图 19-9 MDS 体系结构

目前在 Globus 中的 MDS 中，信息提供者是网格资源信息服务（Grid Resource Information Service，GRIS），特定聚合目录服务是网格检索信息服务（Grid Index Information Service，GIIS）。

GRIS 是一个可配置的提供信息的模块。GRIS 提供了一个统一方法来查询网格资源（如计算节点、存储系统、科学设备、网络、数据库等）的当前配置、性能及状态。GIIS 是一个可配置的聚合目录模块。GIIS 将随意的 GRIS 服务组织起来，从而提供一个一致的、连贯的网格系统视图，这样就方便了网格应用程序。用户可以使用 GIIS 来检索自己感兴趣的资源。

19.3.4　远程数据访问 GASS

在许多领域的计算中，程序经常与运行程序所需的数据不在同一节点，甚至相距很远，这样就需要一种既能对远程请求做有限的功能性限制又不影响程序执行的高性能性的数据访问机制。远程数据访问（Global Access to Secondary Storage，GASS）就是这样一种数据移动及访问服务。它通过统一资源定位器（Uniform Resource Locator）定义了一个全局名字空间，这样就允许应用程序通过标准的 I/O 接口访问远程文件。通过加入默认数据移动策略以及支持程序员对数据移动地管理，实现了高性能的程序计算。GASS 与 Globus 其他模块结合起来进行服务，如：GASS 使用了 Globus 的安全及通信模块，而 MDS 则使用了 GASS 进行远程实时监控。

19.3.5　Globus Toolkit I/O

Glbus Toolkit I/O 对 TCP、UDP 以及文件 I/O 提供了一个易用的接口。它支持多线程、同步和异步接口，并与 GSI 结合起来。

19.3.6　Nexus

Globus 通信模块建立在 Nexus 通信库基础之上。Nexus 定义了五个概念：节点（node）、上下文（context）、线程（thread）、通信连接（communication link）和远程服务请求（remote service request）。实现并操纵这五个概念的 Nexus 函数集就构成了 Globus 通信接口，这个通信接口为 Globus 其他模块广泛使用，并可以用于建立各种各样的高层服务。Nexus 在广域通信方式下实现了远程服务请求（Remote Service Request）、同步过程调用、消息传递、分布式共享内存等多种通信机制，并支持对服务质量、可靠性、加密与否等参数的选择。

Nexus 支持多种通信模式。Nexus 通过在通信连接中由起点（startpoint）与终点（endpoints）构成，Nexus 通信方式可以是多个起点对一个终点，也可以是一个起点对多个终点（广播模式），可以通过自动或人工来选择通信方式。

19.3.7　Heartbeat Monitor（HBM）

HBM 提供了一个简单的、高可靠性的机制来监视作业的运行状态。HBM 的首要目标是可靠性与健壮性，为此，HBM 的实现不依赖于其他 Globus 模块（如 MDS）和任何特定的容错模块。

HBM 由以下三个部分构成：HBM 客户库（HBM Client Library，HBMCL）、HBM 本地监视器（HBM Local Monitor，HBMLM）和 HBM 数据收集器（HBM Data Collector，HBMDC）。

19.4　Globus Toolkit 4.0 开发实例

本节首先部署 Globus Toolkit 4.0 开发环境，然后开发网格服务，并使用 GT4 工具进行服

务的部署。

19.4.1 部署 GT4 开发环境

使用 GT4（Globus Toolkit 4.0）进行网格程序的开发，需要部署以下软件：

（1）Sun Java SDK，提供 Java 环境，可从 http://java.sun.com 进行下载安装，本章使用版本 1.4.2，下载地址为：http://java.sun.com/j2se/1.4.2/download.html。

（2）Eclipse IDE，由 IBM 提供的进行 Java 开发的 IDE 环境，提高开发效率，可从链接 http://www.eclipse.org/downloads/index.php 下载此软件。

（3）Apache Jakarta Tomcat，基于 Java 的 Web 服务器软件，其下载链接地址为：http://jakarta.apache.org/tomcat/。

（4）Sysdeo Eclipse Tomcat Launcher Plug-in，一个 Eclipse 插件，用于将 Tomcat 集成到 Eclipse 环境中，其下载链接地址为：http://www.sysdeo.com/eclipse/tomcatPlugin.html。

（5）GT4 WS Core、WSRF 和 Web Service Notification（WSN）系列标准的 Globus Toolkit 实现。它提供了一些 API 和工具用来构建有状态的 Web 服务（WS-Resources），其下载链接地址为：http://www-unix.globus.org/toolkit/downloads/development/。

（6）globus-build-service，一个进行 GT 服务部署及编译的小工具，其下载链接地址为：http://gsbt.sourceforge.net/content/view/14/31/。

Eclipse 的安装比较简单，解开安装文件即可，这里假设把它安装在 C:\dev\eclipse 路径下，安装完毕后更改 JRE 的位置，使其使用新安装的 JDK，而不是默认的 C:\Program Files\Java 目录中的 JRE，其方法为：转到 Window→Preferences 菜单中，并打开 Eclipse Preferences 对话框，选择 Java→Installed JREs，单击 Add 将 JDK 添加到这个列表中，如图 19-10 所示。

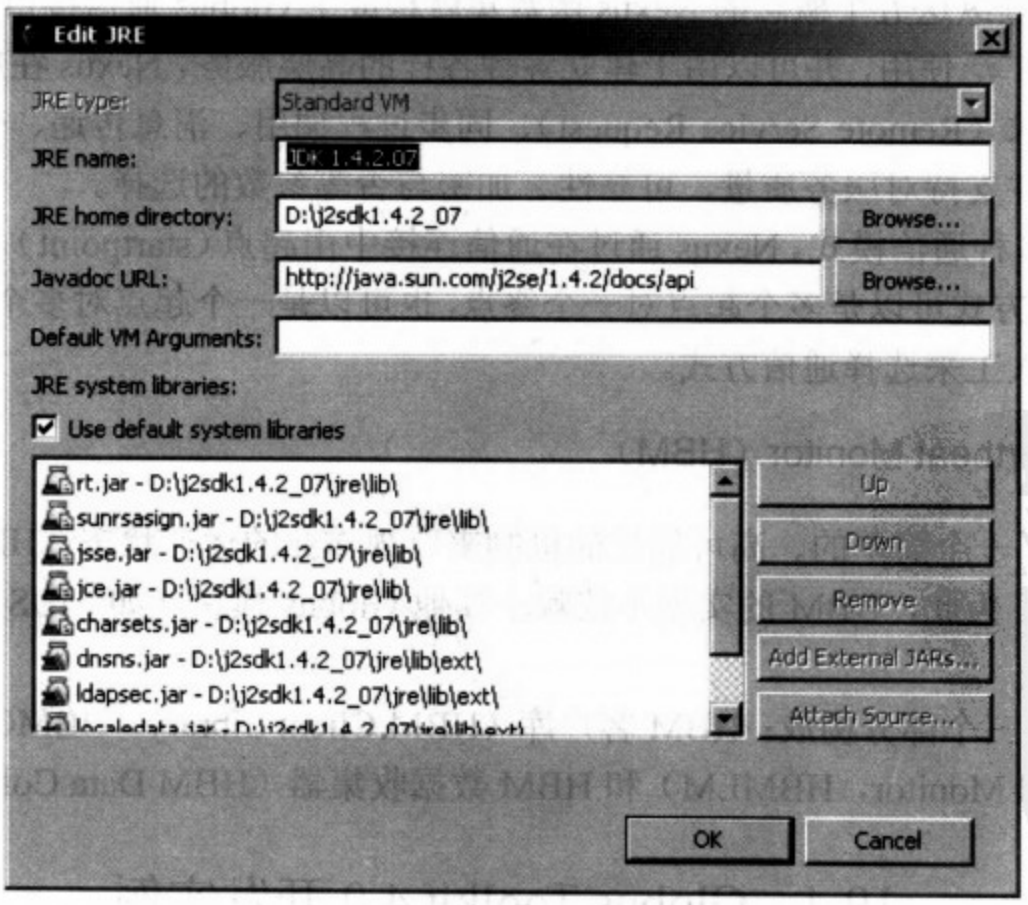


图 19-10 将 JDK 添加到已安装的 JRE 中

选中它并将其设置为默认值，如图 19-11 所示，单击 OK 按钮完成配置过程。

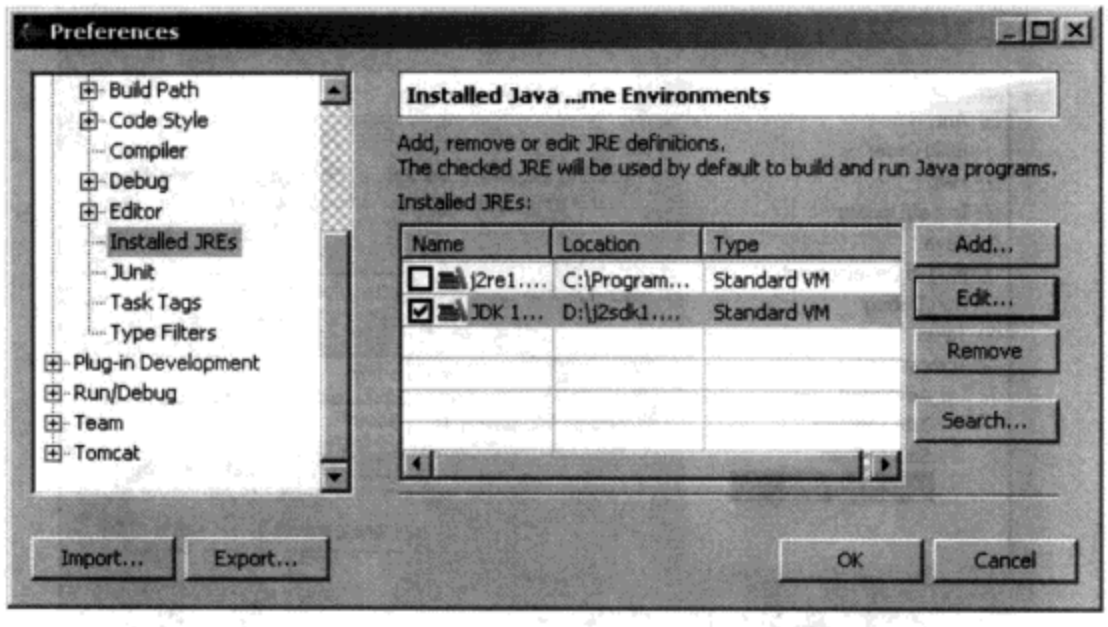


图 19-11　设置默认的 JDK

这里将 Tomcat 也安装在 C:\dev 目录里，同 Eclipse 一样，解开即可，即 C：\dev\jakarta-tomcat-5.0.28，为其增加一个管理用户 admin，在文件 C：\dev\jakarta-tomcat-5.0.28\conf\tomcat-users.xml 中添加如下行即可：

```
<user username="admin" password="admin" roles="admin,manager"/>
```

接着使用 Sysdeo Tomcat Launcher 插件将 Tomcat 与 Eclipse 整合起来，首先将下载下来的 Sysdeo Tomcat Launcher 插件放到 Eclipse 的 plug-ins 中，重启 Eclipse，并转到 Window→Preferences 菜单中，打开 Eclipse Preferences 对话框，选择 Tomcat 页面，然后选择 Version 5.x 按钮，填充 Tomcat home 文本框，如图 19-12 所示。

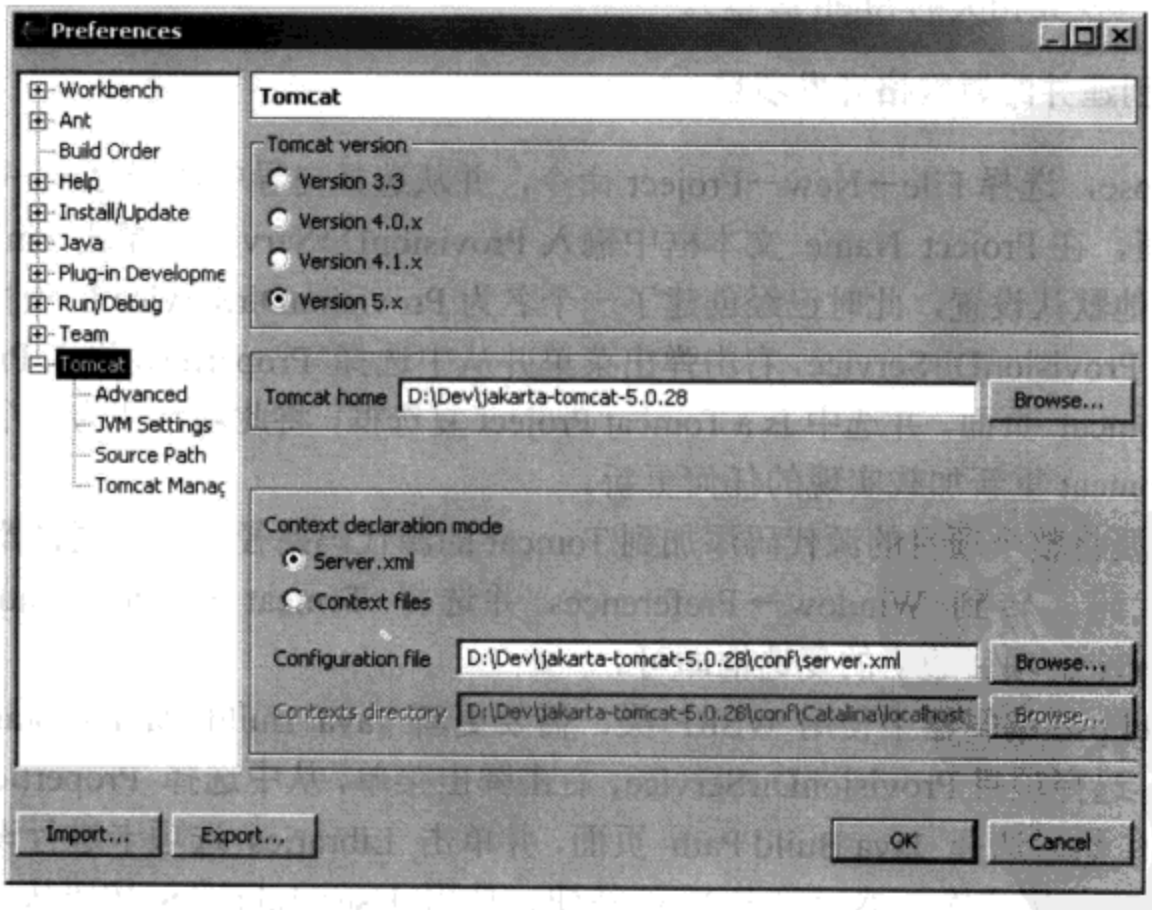


图 19-12　设置 Tomcat 版本

另外，还需保证 JVM Settings 页面中选择使用了刚才设置的使用 JDK 的 JRE 环境，最后，在 Tomcat Manager App 页面中输入管理者的用户名和密码，如图 19-13 所示。

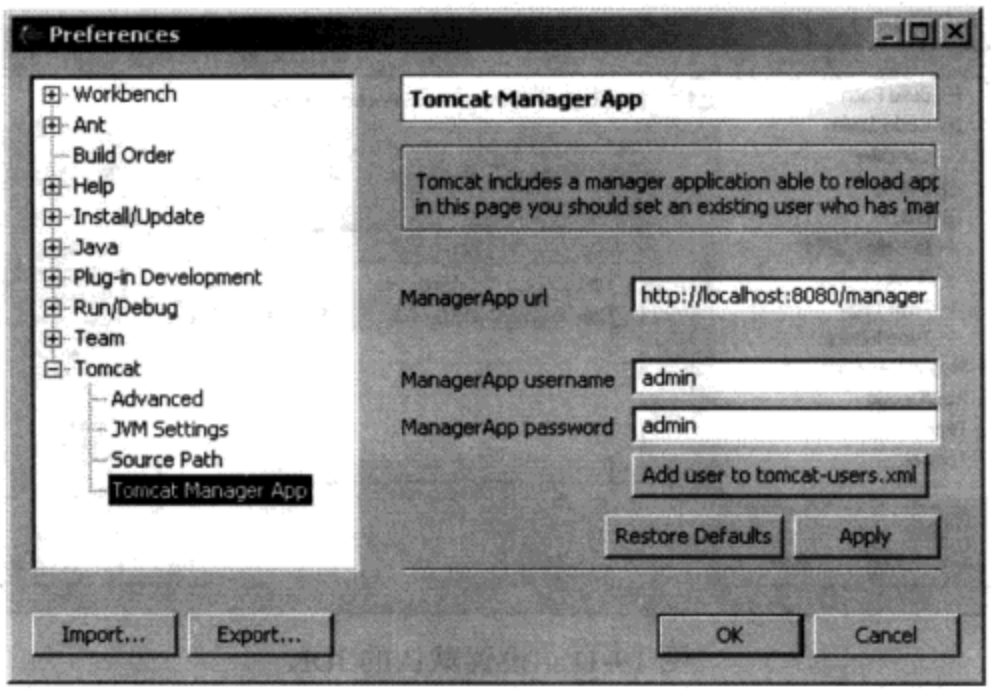


图 19-13 Tomcat Manager App 页面

将下载的 GT4 WS Core 包解压后放到 C:\dev\GTK 目录中，并在此路径中设置 GLOBUS_LOCATION 环境变量的值，另外还需从链接 http://ant.apache.org/bindownload.cgi 下载 Ant 编译 WS-Core。在 MS DOS 命令行窗口中运行以下命令，编译 WS-Core 并将其安装在 C:\dev\GTK 中。

```
C:\>cd %GLOBUS_LOCATION%\ws-core-3.9.5
C:\Dev\GTK\ws-core-3.9.5>ant all
```

最后将 globus-build-service 放置在 C:\dev\GTK\etc 中，它主要用于帮助编译 GAR 文件的 Ant 编译文件（及其相关的 Shell 脚本）。

19.4.2 创建并配置网格开发项目

打开 Eclipse，选择 File→New→Project 命令，并从选择向导中选择 Java→Java Project；单击 Next 按钮，在 Project Name 文本框中输入 ProvisionDirService；单击 Finish 按钮，接受新建项目的其他默认设置，此时已经创建了一个名为 ProvisionDirService 的项目。

选择项目 ProvisionDirService，右击弹出菜单并从中选择 Properties，进入 Project properties 页面，选择 Tomcat 页面，并选中 Is a Tomcat Project 复选框，将此项目变为一个 Tomcat 项目。这样可以使 Tomcat 重新加载实现的任何更新。

另外还需要将整个项目的源代码添加到 Tomcat 的源代码路径中，使调试器在进行调试时使用最新的代码。转到 Window→Preferences 并选择 Tomcat→Source Path 页面，选中 ProvisionDirService 项目边上的复选框即可。

为了在 Eclipse 编辑器中使用 WSRF 类，需要更新 Java Build Path Libraries，使其包括 WS-Core JAR，选择项目 ProvisionDirService，右击弹出菜单，从中选择 Properties，进入 Project properties 页面，然后选择 Java Build Path 页面，并单击 Libraries 选项卡进行设置，使用 Add Library 按钮来添加用户库，然后单击 Next 按钮，使用 User Libraries 按钮从 GT4 库目录中直接创建一个用户库。在 User Libraries 对话框中单击 New 按钮，并创建一个名为 GT4 Library 的库，如图 19-14 所示。

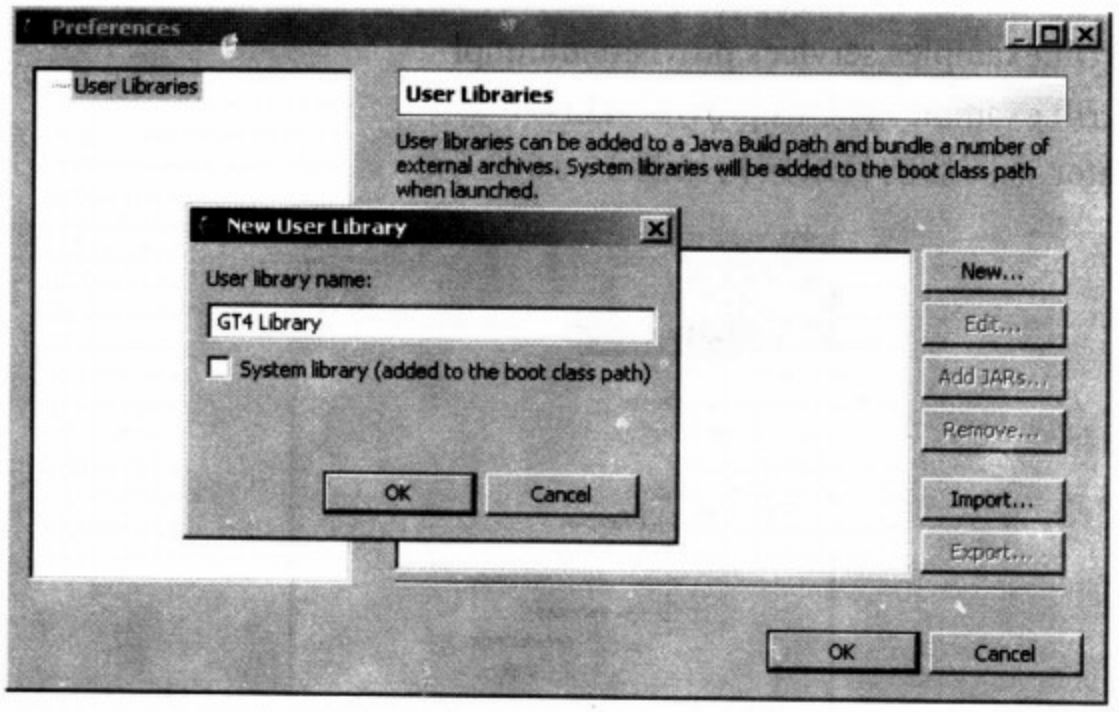


图 19-14　新建一个用户库

单击 Add JARs，并选择 %GLOBUS_LOCATION%\lib 中的所有 JAR 文件，将其添加到库中。单击 OK 和 Finish 按钮，Libraries 选项卡就已经配置好了。更新后的 Libraries 选项卡如图 19-15 所示。

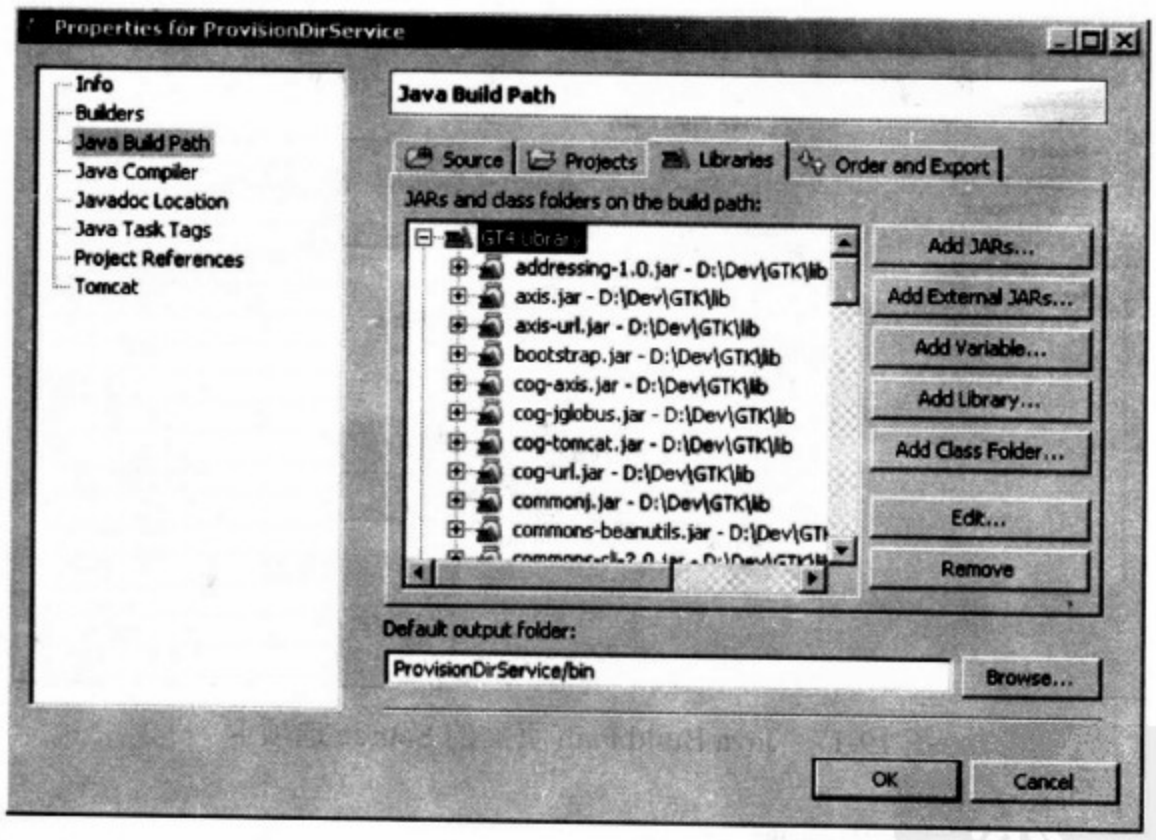


图 19-15　更新后的 Libraries 选项卡

接着创建用于存放这些代码的文件夹和包结构，在菜单中选择 Window→Show View→Navigator 命令，添加 Navigator 视图，并创建用于存放描述服务接口的 WSDL 文档的文件夹位置。其方法为：选择 New→Package 命令，将其命名为 schema.examples.ProvisionDirService，这样就会在项目的根目录中出现一个 schema/examples/ProvisionDirService/ 路径。依此法继续添加以下四个包：

- etc
- build.classes

- org.merrill.examples.services.provisiondir.impl
- org.merrill.examples.clients.provisiondir

更新 Navigator 视图后的目录结构如图 19-16 所示。

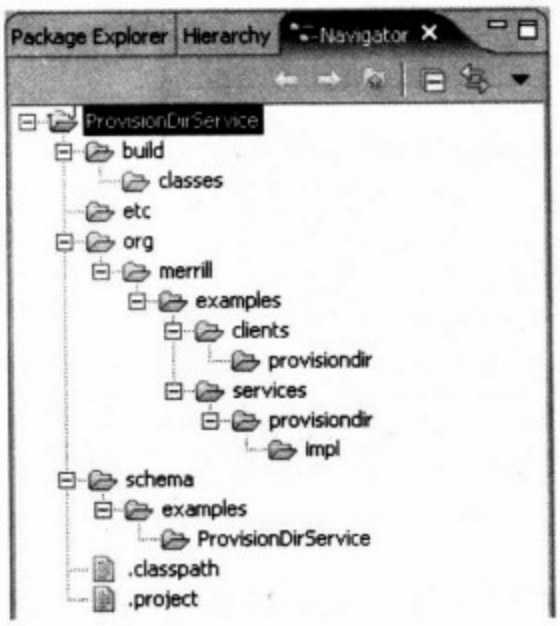


图 19-16　更新后的 Navigator 视图

最后设置编译源码后的输出目的地，如图 19-17 所示。

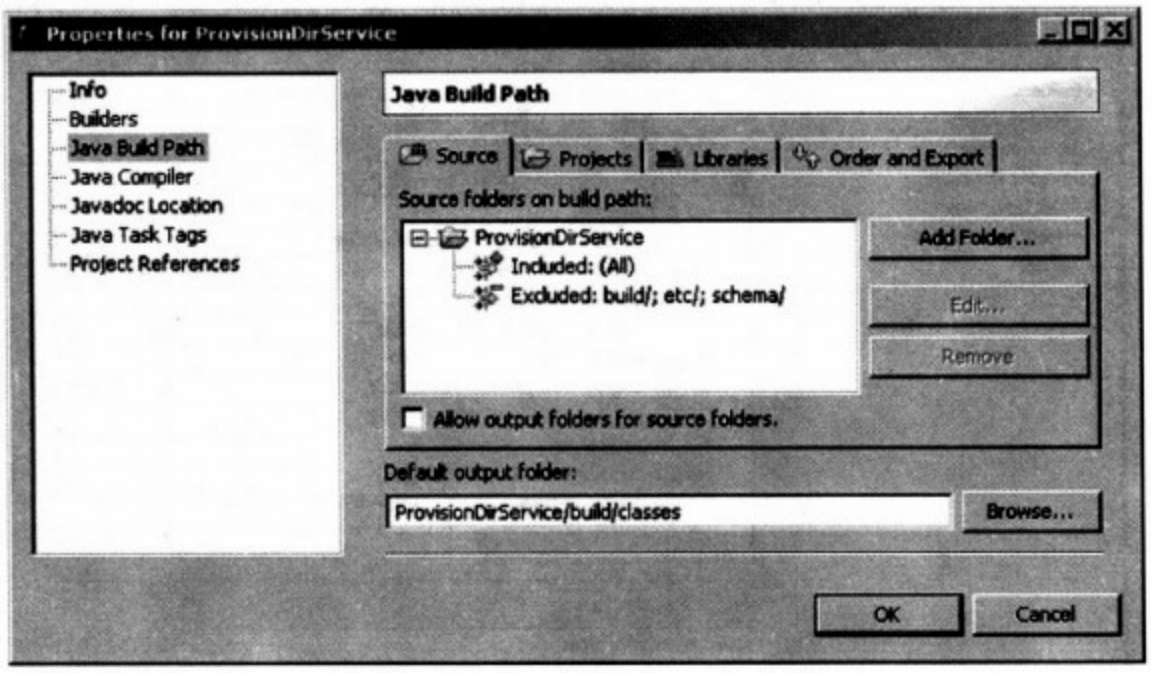


图 19-17　Java Build Path 页面的 Source 选项卡

19.4.3　GT4 网格服务开发步骤

使用 GT4 开发及部署一个基于 WSRF 的 Web 服务大体上分为 5 个步骤：

（1）使用 WSDL 定义服务接口。

（2）使用 Java 语言实现服务。

（3）使用 WSDD 及 JNDI 定义部署的相关参数。

（4）使用 Ant 编译生成 GAR 包。

（5）使用 GT4 工具进行服务的部署。

下面以一个简单的 MathService 网格服务为例来说明以上五步，在此例中主要实现一些简

单的加/减法运算。

19.4.4　使用 WSDL 定义服务接口

首先需要定义一些服务接口，以指定将要为外部世界所提供的服务接口，而暂时不用考虑算法、交互等内部工作机制，仅仅需要知道为用户所提供的一些操作。在 GT4 中，使用了一种特定的 XML 语言来描述服务接口，用来描述 Web 服务所提供的一些服务，通常将这种语言简称为 WSDL，即 Web Service Description Language。

因此，首先创建 WSDL 文件来描述将要提供的服务接口，文件存储在项目文件夹 schema/examples/ProvisionDirService/ProvisionDir.wsdl 中，它是一个描述服务接口的 XML 文档。这个服务接口描述了外部世界如何与我们的服务进行交互，尤其是可以使用哪些选项。其内容如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="ProvisionDirService"
    targetNamespace="http://examples.merrill.org/provisiondir/
          ProvisionDirService"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:tns="http://examples.merrill.org/provisiondir/ProvisionDirService"
    xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/03/addressing"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
    xmlns:wsrlw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-Resource
          Lifetime-1.2-draft-01.wsdl"
    xmlns:wsrp="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-Resource
          Properties-1.2-draft-01.xsd"
    xmlns:wsrpw="http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-Resource
          Properties-1.2-draft-01.wsdl"
    xmlns:wsdlpp="http://www.globus.org/namespaces/2004/10/WSDLPreprocessor"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<wsdl:import
    namespace=
    "http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProp
          erties-1.2-draft-01.wsdl"
    location="../../wsrf/properties/WS-ResourceProperties.wsdl" />

<types>
<xsd:schema targetNamespace="http://examples.merrill.org/provisiondir/Provision
DirService"
    xmlns:tns="http://examples.merrill.org/provisiondir/ProvisionDirService"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:import
        namespace="http://schemas.xmlsoap.org/ws/2004/03/addressing"
        schemaLocation="../../ws/addressing/WS-Addressing.xsd" />

    <!-- Requests and responses -->

    <xsd:element name="listDir">
```

```
    <xsd:complexType/>
</xsd:element>
<xsd:element name="listDirArrayResponse">
    <xsd:complexType>
    <xsd:sequence>
        <xsd:element name="a" type="xsd:string" minOccurs="0" maxOccurs="un
            bounded"/>
    </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="changeCwd" type="xsd:string"/>
<xsd:element name="changeCwdResponse">
    <xsd:complexType/>
</xsd:element>

<!-- Resource properties -->

<xsd:element name="Cwd" type="xsd:string"/>

<xsd:element name="ProvisionDirResourceProperties">
<xsd:complexType>
    <xsd:sequence>
        <xsd:element ref="tns:Cwd" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
</xsd:complexType>
    </xsd:element>

</xsd:schema>
</types>

<message name="listDirInputMessage">
    <part name="parameters" element="tns:listDir"/>
</message>
<message name="listDirOutputMessage">
    <part name="parameters" element="tns:listDirArrayResponse"/>
</message>

<message name="changeCwdInputMessage">
    <part name="parameters" element="tns:changeCwd"/>
</message>
<message name="changeCwdOutputMessage">
    <part name="parameters" element="tns:changeCwdResponse"/>
</message>

<portType name="ProvisionDirPortType"
    wsdlpp:extends="wsrpw:GetResourceProperty"
    wsrp:ResourceProperties="tns:ProvisionDirResourceProperties">

    <operation name="listDir">
```

```
        <input message="tns:listDirInputMessage"/>
        <output message="tns:listDirOutputMessage"/>
    </operation>

    <operation name="changeCwd">
        <input message="tns:changeCwdInputMessage"/>
        <output message="tns:changeCwdOutputMessage"/>
    </operation>

</portType>

</definitions>
```

虽然 WSDL 文档通常是用户看不到的，但是有几个地方应该注意：

（1）<portType>元素描述了两个操作：listDir 和 changeCwd。注意，这里是通过从 Resource Properties WSDL 派生端口类型来支持资源属性的（还要注意，尽管这可以与软件一起使用，但是 WSRF 通常都是使用扩展端口类型，而不使用这种旧的风格；从技术上来讲，应该包含全部的定义）。这两个操作都由一个输入和一个输出消息组成，这两个消息使用上面的 <message> 标签进行描述。

（2）由于这里是使用文档/文本风格的消息进行交换，因此这些消息必须有一个消息部分（即使逻辑方法不使用参数或者使用多个参数，不返回值或者返回多个值，都是如此。）

（3）这些“包装”消息部分是在 WSDL 的<!-- Requests and responses -->部分进行描述的，WSDL 描述了请求和响应消息的模式格式。listDir 元素没有子域，因为逻辑 listDir()方法不需要参数。listDirArrayResponse 元素充当返回值的包装；它是一个字符串数组，指出了当前工作目录中的条目。changeCwd 元素是一个字符串类型，changeCwdResponse 元素没有子域，因为这个方法没有返回值。

（4）资源属性（当前工作目录）是在<!-- Requests and responses -->部分中进行描述的。

19.4.5　使用 Java 语言实现服务

接着使用 Java 来实现网格服务，在项目中添加相关 Java 文件，主要包括以下几个部分：

（1）org/merrill/examples/services/provisiondir/impl/ProvisionDirQName.java 文件，定义了一个非常方便的接口类，其中包含了与这个网格服务有关的 QName URI/名称空间常量，通过服务（和客户机）类对这个接口的实现，就可以直接引用这些常量，而不用通过项目再复制它们。其内容如下：

```
package org.merrill.examples.services.provisiondir.impl;

import javax.xml.namespace.QName;

public interface ProvisionDirQNames {
   public static final String NS = "http://examples.merrill.org/provisiondir/Pro
                                   visionDirService";

    public static final QName RESOURCE_PROPERTIES = new QName(NS,
            "ProvisionDirResourceProperties");
```

```
        public static final QName RESOURCE_REFERENCE = new QName(NS,
                "ProvisionDirResourceReference");

        /* Insert ResourceProperty Qnames here. */

        public static final QName RP_CWD = new QName(NS, "Cwd");
    }
```

（2）org/merrill/examples/services/provisiondir/impl/ProvisionDirService.java 文件，主要用来为公开本地目录信息提供核心功能的服务实现。其内容如下：

```
    package org.merrill.examples.services.provisiondir.impl;

    import java.rmi.RemoteException;

    import org.globus.wsrf.ResourceContext;
    import org.globus.wsrf.Resource;
    import org.globus.wsrf.ResourceProperties;
    import org.globus.wsrf.ResourceProperty;
    import org.globus.wsrf.ResourcePropertySet;
    import org.globus.wsrf.impl.ReflectionResourceProperty;
    import org.globus.wsrf.impl.SimpleResourcePropertySet;

    import org.merrill.examples.provisiondir.ProvisionDirService.
ListDirArrayResponse;
    import org.merrill.examples.provisiondir.ProvisionDirService.
ChangeCwdResponse;

public class ProvisionDirService implements Resource, ResourceProperties {

    /* Resource Property set */
    private ResourcePropertySet propSet;

    /* Insert resource properties here. */
    private String cwd;

    /* Constructor. Initializes RPs */
    public ProvisionDirService()
            throws RemoteException {

        this.propSet = new SimpleResourcePropertySet(
                ProvisionDirQNames.RESOURCE_PROPERTIES);

        try {
            /* Initialize Resource Properties here.*/
            ResourceProperty cwdRP = new ReflectionResourceProperty(
                    ProvisionDirQNames.RP_CWD,
                    "Cwd",
                    this);
            this.propSet.add(cwdRP);
            setCwd("/");
```

```
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage());
        }

    }

    public ListDirArrayResponse listDir() throws RemoteException {

        java.io.File currentDir = new java.io.File(cwd);
        return new ListDirArrayResponse(currentDir.list());
    }

    public ChangeCwdResponse changeCwd(String newCwd) throws RemoteException {

        setCwd(newCwd);
        return new ChangeCwdResponse();
    }

    /* Insert get/setters for Resource Properties here.*/

    public String getCwd() {
        return cwd;
    }

    public void setCwd(String cwd) {
        this.cwd = cwd;
    }


        /* Required by interface ResourceProperties */
        public ResourcePropertySet getResourcePropertySet() {
            return this.propSet;
        }

    }
```

这个类中有几个地方需要注意：

（1）资源属性（如我们的当前工作目录）是作为类的私有域进行维护的。

（2）需要在构造函数中对它们进行初始化，并将其插入资源属性集中。

（3）需要为那些遵循 get<field name> 和 set<field name> 模式命名方法的资源属性创建简单的取值/赋值方法。

（4）这里的操作方法（在本例中是 listDir() 和 changeCwd()方法）通常都只有一个返回类型；它们在 WSDL 中都是在输入和返回类型之后定义的。不要担心编辑器是否会显示红色的编译错误——因为尚未生成表示这些类型的存根类。

19.4.6　使用 WSDD 及 JNDI 定义部署的相关参数

创建 JNDI 部署文件，它可以让 GT4 WSRF 实现找到该服务的资源主目录，将其存放在项目路径 org/merrill/examples/services/provisiondir 下，文件名为 deploy-jndi-config.xml，其源

代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<jndiConfig xmlns="http://wsrf.globus.org/jndi/config">

<service name="examples/ProvisionDirService">
    <resource name="home" type="org.globus.wsrf.impl.ServiceResourceHome">
    <resourceParams>

        <parameter>
            <name>factory</name>
            <value>org.globus.wsrf.jndi.BeanFactory</value>
        </parameter>

    </resourceParams>

    </resource>
</service>
</jndiConfig>
```

这个文件目前基本上是空的，它需要另外导入两个文件才能正常工作。典型版本的 WS-ResourceProperties.wsdl 和 WS-Addressing.xsd 文件会引用用户可能尚未在机器上创建的目录，这个可直接通过搜索从网上下载即可。

另外还需要创建文件 org/merrill/examples/services/provisiondir/ deploy-server.wsdd，它是一个 WSDD 配置文件，用于告诉 Web Service 容器（Tomcat）如何发布 Web 服务，其源代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment name="defaultServerConfig"
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <service name="examples/ProvisionDirService" provider="Handler" use="literal"
            style="document">
        <parameter name="className" value="org.merrill.examples.services.
            provisiondir.impl.ProvisionDirService"/>
        <wsdlFile>share/schema/examples/ProvisionDirService/ProvisionDir_ser
            vice.wsdl</wsdlFile>
        <parameter name="allowedMethods" value="*"/>
        <parameter name="handlerClass" value="org.globus.axis.providers.RPCProv
            ider"/>
        <parameter name="scope" value="Application"/>
        <parameter name="providers" value="GetRPProvider DestroyProvider"/>
        <parameter name="loadOnStartup" value="true"/>
    </service>
</deployment>
```

目前，整个项目的文档结构如图 19-18 所示。

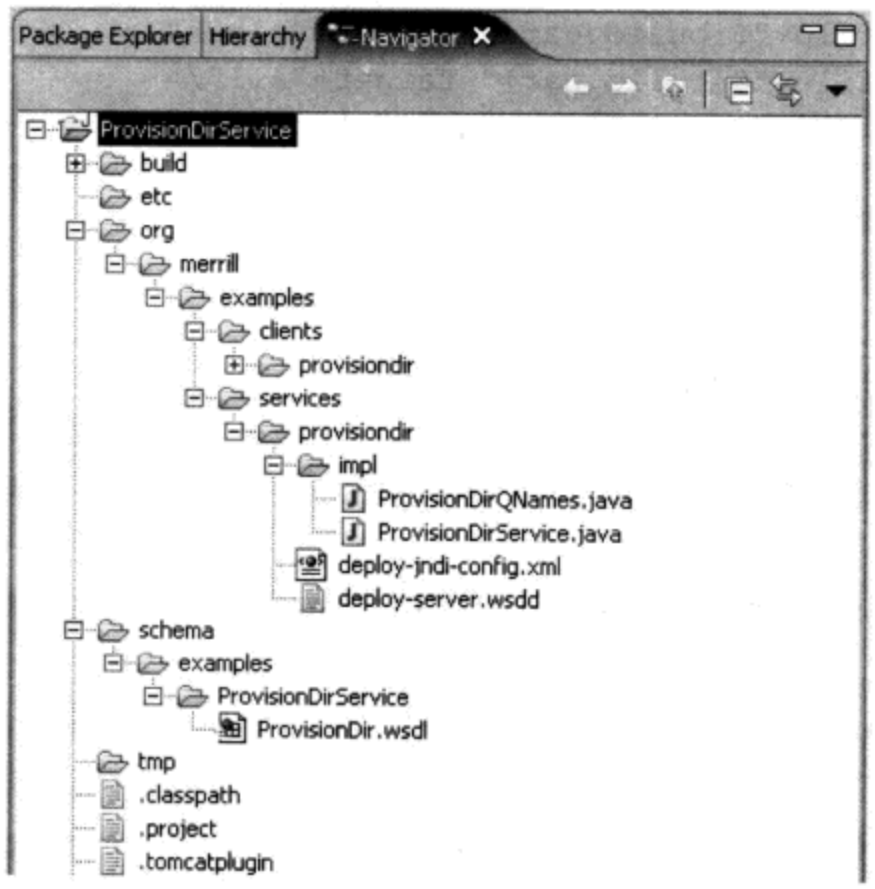


图 19-18　项目浏览

19.4.7　使用 Ant 编译生成 GAR 包

在此来创建启动配置，用于编译和部署已经开发好的网格服务。启动配置是 Eclipse 为执行一个或多个 Ant 编译文件目标而提供的一种机制。对于 ProvisionDirService 项目来说，希望创建一个启动配置，通过单击鼠标，它就可以自动执行以下过程：

（1）生成服务存根和参数数据结构类。

（2）对服务 WSDL 执行一些补偿调整（tweaking）操作。

（3）汇编服务 GAR，它包含了 Web Service 容器部署服务所需要的所有文件和信息。

（4）将 GAR 部署到 GT4 目录树中。

（5）将 GT4 WSRF 部署到 Tomcat 中。

globus-build-service 和 WS-Core 发行版包含了几个 Ant 编译文件，它们可以累积完成所有这些任务，但是需要创建一个“主”Ant 编译文件在一个启动配置中对它们进行协调，这个主编译文件就是 buildservice.xml，它负责对 globus-build-service 和 WS-Core 编译文件中的操作进行协调，将其放在项目的根目录下，源代码内容为：

```
<?xml version="1.0"?>
<!--
-->
<project default="all" name="Grid Service Buildfile" basedir=".">
    <description>
        Grid Service Buildfile
    </description>

    <property environment="env"/>
    <target name="all">
```

```
        <ant antfile="${build.gar}" target="clean"/>
        <ant antfile="${build.gar}" target="all"/>
        <ant antfile="${build.packages}" target="deployGar"/>
        <ant antfile="${build.tomcat}" target="deploySecureTomcat" dir="/Dev/
            GTK/share/globus_wsrf_common/tomcat" />
    </target>
</project>
```

这个“主”编译文件会调用以下外部编译文件：

（1）Globus-Build-Service build.xml Ant 编译文件（它位于解压后的 d:/dev/GTK/etc/目录中）；

（2）WS-Core build-packages.xml Ant 编译文件（C:/dev/GTK/share/globus_wsrf_common/build-packages.xml）；

（3）WS-Core tomcat.xml Ant 编译文件（C:/dev/GTK/share/globus_wsrf_common/tomcat/tomcat.xml）。

在开始为这个 buildservice.xml 编译文件创建一个启动配置之前，需要先创建一个 buildservice.properties 文件，它包含了这个服务所特有的 name=value 属性，以及指导各种编译任务通过.GAR 的创建和部署过程所需的属性。这个 buildservice.properties 编译文件应该被添加到该项目的根目录中。这个文件的源代码如下：

```
package=com.merrill.examples.services.provisiondir
interface.name=ProvisionDir
package.dir=org/merrill/examples/services/provisiondir
schema.path=examples/ProvisionDirService
service.name=ProvisionDirService
gar.filename=org_merrill_examples_provisiondir

build.gar=C:/dev/GTK/etc/build.xml
build.packages=C:/dev/GTK/share/globus_wsrf_common/build-packages.xml
build.tomcat=C:/dev/GTK/share/globus_wsrf_common/tomcat/tomcat.xml
gar.name=C:/dev/eclipse/workspace/ProvisionDirService/org_merrill_examples
_provisiondir
tomcat.dir=C:/dev/jakarta-tomcat-5.0.28
```

使用这个现在已经链接到项目上的 Ant 编译文件和所创建的一个对应的 .properties 文件，我们就可以创建一个启动配置，它会为这个服务创建.GAR 文件（并填充其他必要的文件）。要实现这一点，请用鼠标右击 buildservice.xml 文件，并选择 Run→External Tools 命令，如图 19-19 所示。

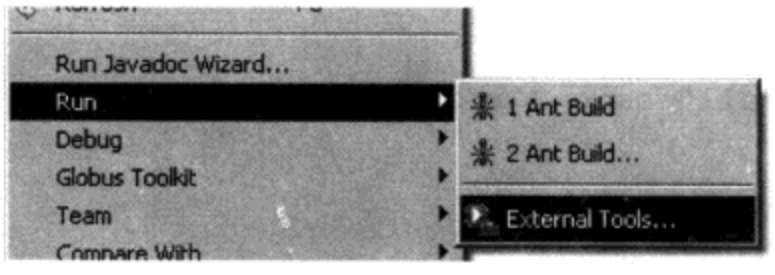


图 19-19 Run→External Tools 选项

如果需要，请从 Configurations 面板中选择 Ant Build，并单击 New 按钮，然后选择

ProvisionDirService/buildservice.xml，并将其重命名为 Build and Deploy ProvisionDir，并在 Main 选项卡中的 Base Directory 域中输入该项目的根目录 ${workspace_loc:/ProvisionDirService}，如图 19-20 所示。

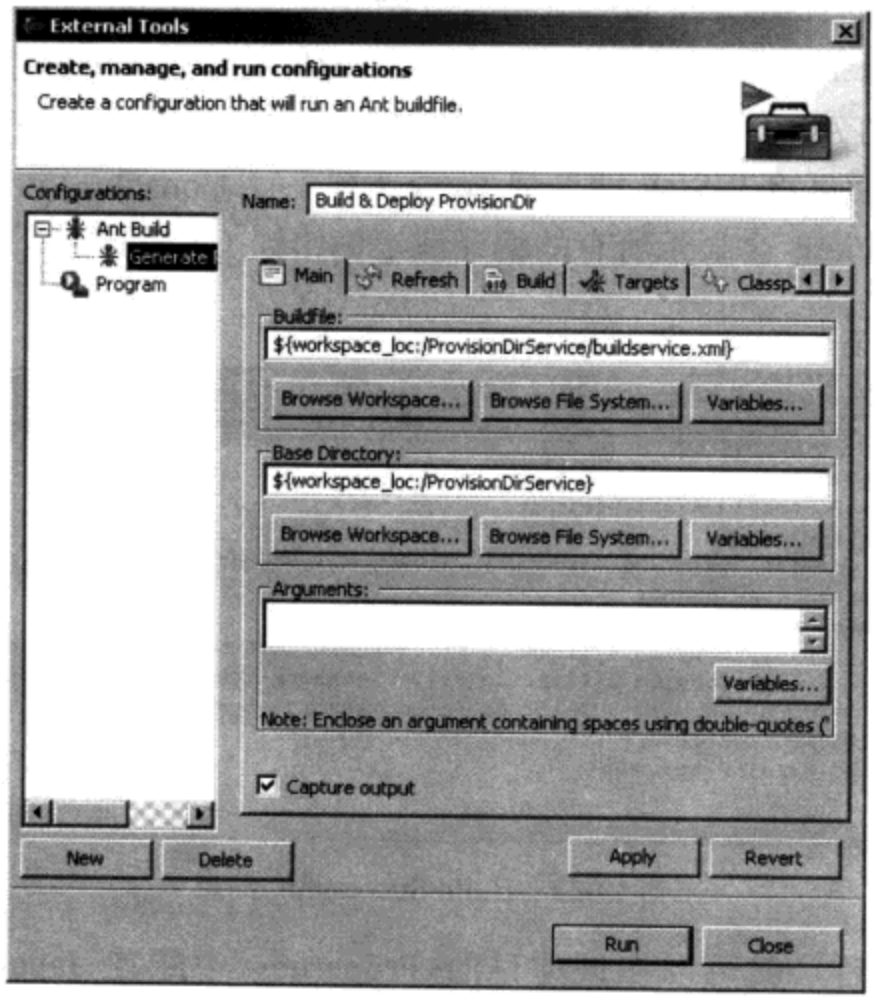


图 19-20　External Tools 对话框

在 Properties 选项卡中取消选中 Use global properties 复选框，并单击 Add→Files 按钮，将 buildservice.properties 文件从该项目的根目录中添加进来，如图 19-21 所示。

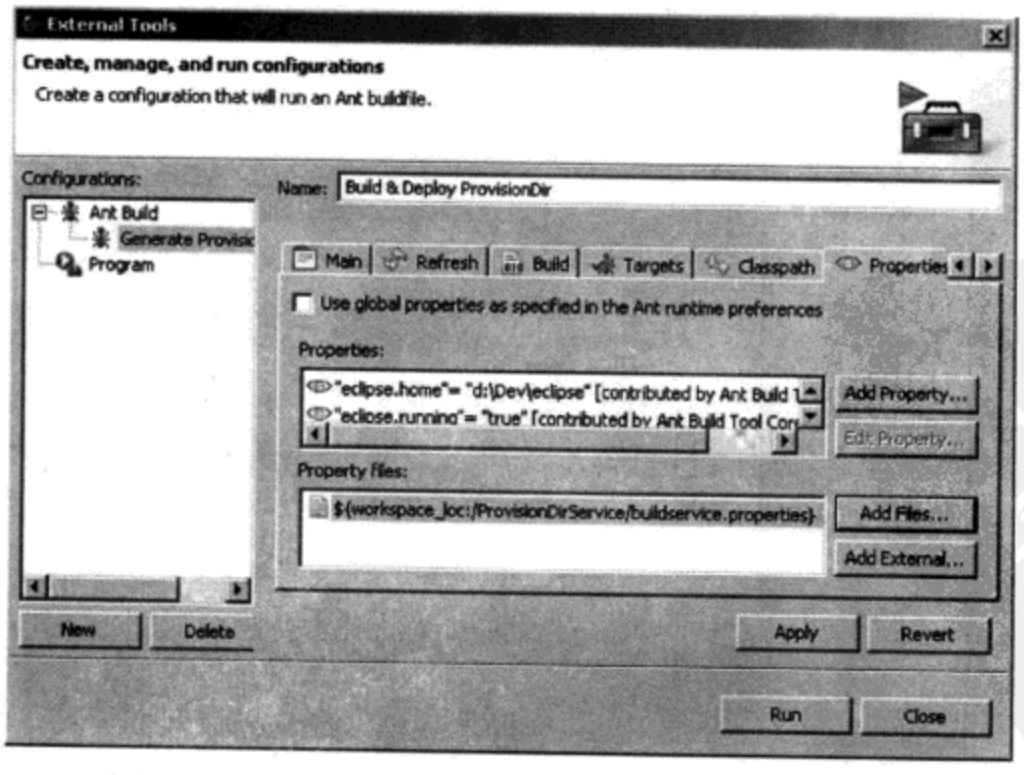


图 19-21　External Tools 对话框的 Properties 选项卡

最后，在 Common 选项卡中选中 External Tools 复选框，这样在单击工具条上的 External

Tools 图标时，就可以显示下拉菜单中的配置。

19.4.8 使用 GT4 工具进行服务的部署

现在已经完成了项目的设置，接下来可以开始反复进行编译/运行/调试/编辑了。可以通过在主工具条上单击 External Tools icon→Build and Deploy ProvisionDir 启动配置来编译并部署网格服务。这可以启动 Ant 任务，并生成其他所有的文件，创建服务的 GAR，并将这个 GAR 部署到 WSRF 中，然后再将 WSRF 部署到 Tomcat 中。在 Console View 中应该可以看到编译过程的输出。Console View 会显示所有的编译或汇编错误（如果存在）或一条 Build Successful 消息，如图 19-22 所示。

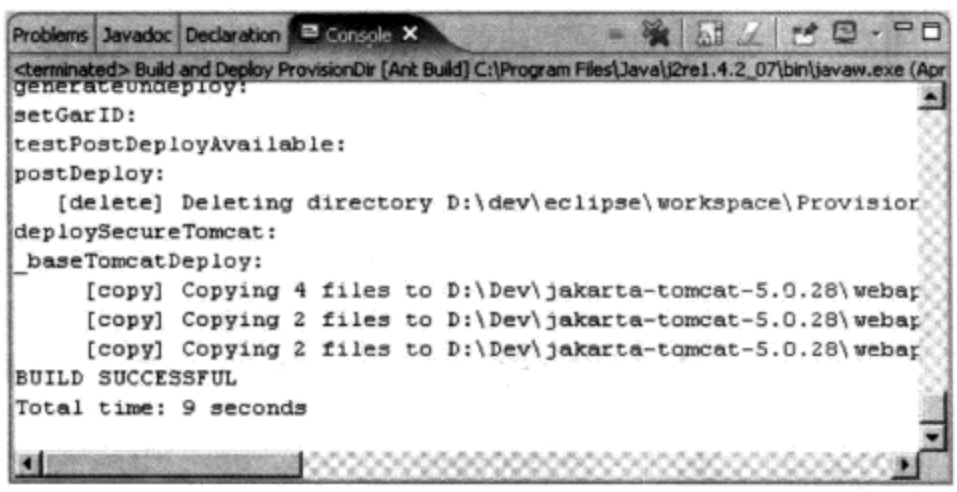


图 19-22 Build Successful 消息

在生成存根类之后，就可以在该项目的 Properties 中配置 Java Build Path 页面中的 Libraries 选项卡，使其在 Build Path 中包含存根类文件夹。这样做将会禁止显示 Editor View 的红色编译错误。切换到该项目的 Properties 中，选择 Java Build Path 页面，并单击 Add Class Folder 按钮，如图 19-23 所示，并单击 OK 按钮关闭对话框。

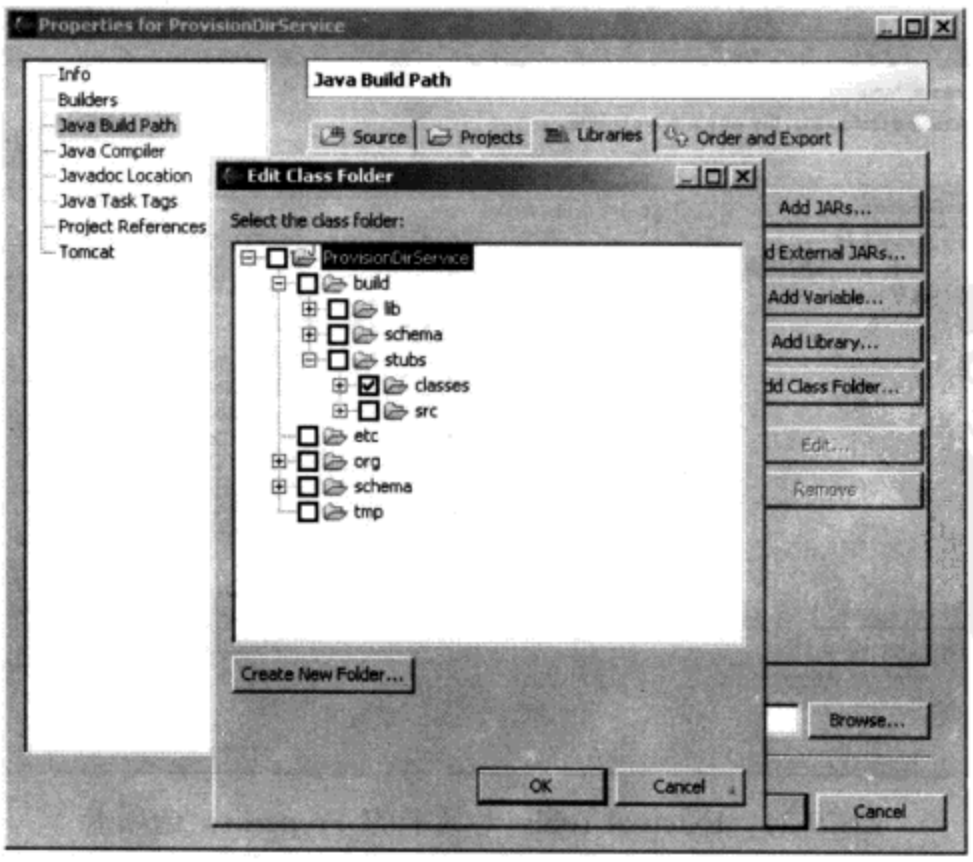


图 19-23 Edit Class Folder 对话框

现在可以将源代码附加到存根类上了：右击类文件夹的 Source Attachment，并指定/ProvisionDir/build/stubs/src 文件夹，单击 OK 按钮完成配置过程。

运行（和停止）网格服务与启动 Tomcat 容器一样简单，可以通过单击 Start/Stop/Restart Tomcat 工具条按钮实现。

最后使用文件 org/merrill/examples/clients/provisiondir/Client.java 所实现的客户机来测试完成的网格服务，当然可以使用其他的语言（如 Perl）来实现测试客户机。在此处的源代码如下：

```
package org.merrill.examples.clients.provisiondir;

import org.apache.axis.message.addressing.Address;
import org.apache.axis.message.addressing.EndpointReferenceType;
import org.globus.axis.util.Util;

import org.merrill.examples.provisiondir.ProvisionDirService.ListDir;
import org.merrill.examples.provisiondir.ProvisionDirService.ListDirArrayResponse;
import org.merrill.examples.provisiondir.ProvisionDirService.ChangeCwdResponse;
import org.merrill.examples.provisiondir.ProvisionDirService.ProvisionDirPortType;
import org.merrill.examples.provisiondir.ProvisionDirService.service.ProvisionDir
ServiceAddressingLocator;
import org.merrill.examples.services.provisiondir.impl.*;

import org.oasis.wsrf.properties.GetResourcePropertyResponse;

public class Client implements ProvisionDirQNames  {
    static {
        Util.registerTransport();
    }

    public static void main(String[] args) {
        ProvisionDirServiceAddressingLocator locator =
            new ProvisionDirServiceAddressingLocator();

        try {
            String serviceURI=args[0];

            // Create endpoint reference to singleton service
            EndpointReferenceType endpoint = new EndpointReferenceType();
            endpoint.setAddress(new Address(serviceURI));
            ProvisionDirPortType provisionDir =
                locator.getProvisionDirPortTypePort(endpoint);

            // Get PortType
            provisionDir = locator.getProvisionDirPortTypePort(endpoint);

            // Access resource properties (get CWD)
            GetResourcePropertyResponse cwd = provisionDir.getResourcePro
                perty(RP_CWD);
            System.out.println("\nCurrent Working Directory (Cwd RP): " +
```

```
            cwd.get_any()[0].getValue());

        // Perform a listing
        ListDirArrayResponse listings = provisionDir.listDir(new ListDir());
        String[] array = listings.getA();
        System.out.println("\nListing:");
        for (int i = 0; i < array.length; i++) {
            System.out.println("\t" + array[i]);
        }

        // change the current working directory
        provisionDir.changeCwd("/temp");

        // Access resource properties (get CWD)
        cwd = provisionDir.getResourceProperty(RP_CWD);
        System.out.println("\nCurrent Working Directory (Cwd RP): " +
            cwd.get_any()[0].getValue());

        // Perform a listing
        listings = provisionDir.listDir(new ListDir());
        array = listings.getA();
        System.out.println("\nListing:");
        for (int i = 0; i < array.length; i++) {
            System.out.println("\t" + array[i]);
        }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

    }
```

客户机的操作非常简单，步骤如下：

（1）使用所生成的 locator 类，客户机会为远程的 ProvisionDirService 创建一个代理存根。

（2）客户机检索并显示 Cwd 资源属性。

（3）客户机调用 listDir() 操作，并检索包装数据结构，其中包含了描述当前工作路径中内容的字符串列表。

（4）客户机调用 changeCwd() 操作，以切换资源的当前目录。

（5）客户机检索新的 Cwd 资源属性，并再次显示它。

（6）客户机再次调用 listDir()，并显示新当前工作目录中的内容。

要测试客户机，简单地在 Client.java 文件上右击，并从弹出菜单中选择 Run→Run。在所显示的 Run 对话框中，选择 Arguments 选项卡，并在 Program Arguments 文本框中输入 http://127.0.0.1:8080/wsrf/services/examples/ProvisionDirService，单击 Close 保存这个文件的运行配置。

完成这个步骤之后，我们就可以通过简单地用鼠标右击 Client.java 文件并选择 Run→Java Application 命令来重复地运行客户机程序了，其测试结果如图 19-24 所示。

图 19-24　Console 视图

19.5　小结

本章介绍了网格计算的概念，网格中间件的关键技术和 Globus 网格项目，包括如下内容：

- 网格计算的概念。
- 网格环境的构建层次和网格中间件的作用。
- 开放网格服务结构 OGSA。
- 网格中间件项目 Globus。
- 基于 Globus Toolkit 4.0 的网格服务开发，包括部署开发环境、代码编写、编译，以及使用 GT4 工具进行服务的部署。

第 20 章　工作流中间件

20.1　工作流技术概述

20.1.1　工作流的定义

十几年来，不同的研究者对工作流分别提出了不同的定义，他们从不同的角度对工作流的概念进行了描述。

工作流管理联盟（WfMC，Workflow Management Coalition）的定义是：工作流是一类能够全部或部分自动执行的经营过程，它能够根据一系列过程规则、文档、信息或任务在不同的执行者之间进行传递与执行。

PeopleSoft 公司给出的定义是：工作流是一个用来实施经营过程实践的机制。

IBM Almaden 研究中心给出的定义是：工作流是经营过程的一种计算机化的表示模型，定义了完成整个过程所需用到的各种参数。

范玉顺（清华大学国家 CIMS 中心）认为，工作流是一种反映业务流程的计算机化模型，是为了在先进计算机环境下实现经营过程集成与经营过程自动化而建立的可由工作流管理系统执行的业务模型。

以上这些对工作流的定义，是用非形式化的语言对工作流所进行的描述，虽然各不相同，但基本上都达到了这样一个共识：工作流是经营过程的一个计算机实现。这些工作流定义分别反映了经营过程如下几方面的问题，即经营过程是什么、怎么做、由谁来做、做得怎么样。

同时，这些定义也说明了工作流与一般的工作流程之间的区别。前者需要借助计算机软件来完成，并完全在软件系统的控制之下；而后者则没有这种约定。它当中的某些步骤可能也需要用到计算机，但这只不过是局部的计算机应用，整个过程并不是在计算机控制之下。

同样，这些定义中也可以看出工作流系统与群件系统之间的区别。后者强调的是群组工作情况下，不同成员之间的通信、协作与协调的问题；而前者是完成组织业务流程的自动化执行与监控，将通过建模得到的组织业务流程模型进行实例化并投入运行，对运行的过程模型中的活动执行情况进行调度、管理和控制。

20.1.2　工作流管理系统 WfMS

为了实现对业务过程的工作流管理，需要有相应的软件系统的支撑。此种软件系统就称为工作流管理系统（Workflow Management System，WfMS），根据 WfMC 的定义，工作流管理系统是“一种在工作流形式化表示的驱动下，通过软件的执行而完成工作流定义、管理及执行的系统”，其主要目标是对业务过程中各步骤（或称活动、环节）发生的先后次序及同各步骤相关的相应人力或信息资源的调用等进行管理而实现业务过程的自动化，当然此种管理可能会在不同的信息及通信环境下实现，所涉及的范围可以小至一个几人的工作组，也可以大到企

业（机构）与企业（机构）之间。一般而言，所有的 WfMS 都将包含如下三大功能模块，即建立阶段功能、运行阶段的控制功能和普通用户及应用程序的交互功能（也就是运行阶段的人机交互功能）。关于这三大功能的划分及相互之间的关系，可以用图 20-1 表示。

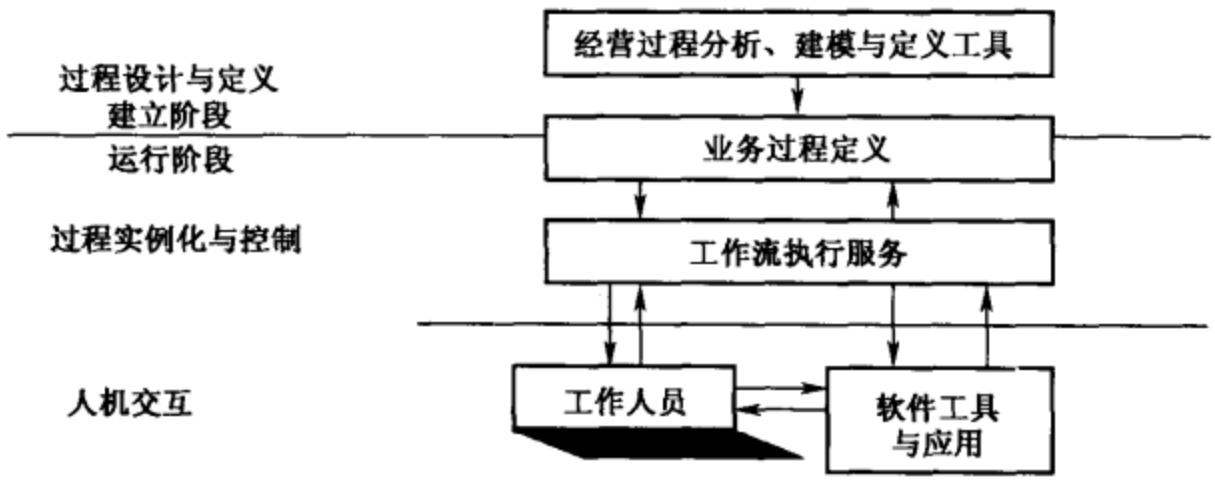


图 20-1　工作流管理系统功能模块划分

（1）建立阶段功能。主要考虑工作流过程和相关活动的定义和建模功能，完成业务过程转化成某种能够被计算机处理的形式化表示，这种形式化表示，就称为过程定义（也称过程模型、过程模板、过程元数据等）。过程定义中一般都包含对业务过程中的各个活动步骤的描述、同这些活动相关的各种计算机或人工操作，以及在各活动之间进行切换的各种控制规则。过程定义所定义的，实际上就是控制工作流过程执行的各种参数。

（2）运行阶段的控制功能。在一定的运行环境下，执行工作流过程，并完成每个过程中活动的排序和调度功能。主要进行在某个计算机环境中工作流过程的管理及各活动步骤之间的状态转换。运行时控制系统将解释过程定义，完成过程的可操作实例的创建及控制，调度过程中各活动步骤，为所涉及的用户生成待其处理的任务，并在合适的时机调用有关的应用程序资源等。这些服务我们一般就称之为“工作流执行服务”，这些系统可能是通过一个工作流引擎提供这些服务，而在另外一些系统则可能会用到多个协作的工作流引擎。工作流引擎是工作流运行时控制系统的核心部分，各种过程实例的执行都是由它负责完成的。

（3）运行阶段的人机交互功能。实现各种活动执行过程中用户与应用系统之间的交互。其中一个核心概念是“工作项”。当工作流执行服务发现某个活动的完成需要用户的参与时，它将会生成一个“工作项”。工作项描述的是处理相应的任务所需的全部信息，如待处理的数据、可能会用到的应用程序等。WfMS 将维护一个由所有待处理的工作项构成的“工作项池”。对每一个用户而言，系统都将会为之维护一个待他完成的工作项的清单，这个清单就称为“工作项列表”。不同的系统将采用不同的方式来维护工作项列表。

工作流应用所生成并处理的数据被称为“工作流应用数据”，这种数据将只被工作流应用所处理，但工作流执行服务可能需要将它们在不同的活动所调用的应用程序之间进行传递。工作流应用所生成或修改的一部分数据可能会被用到活动之间切换条件的计算以及其他的一些控制操作，这部分数据被称为“工作流相关数据”。

这里应该说明的是，工作流管理是一个同具体应用关系极为紧密的领域，它的一些基本思想和做法在许多不同的应用场合及应用系统中都可能会反映出来，这也是导致不同的人对工作流管理有不同理解的重要原因。

20.1.3 工作流管理系统分类

目前已有上百种声称具有工作流管理功能的商品化软件或原型系统。为了对这些系统的功能、特点等有一个具体清晰的认识，可以根据工作流过程本身的特点、系统建模的方式、所使用的底层支撑技术，以及工作流过程的执行方式等的不同而对它们进行相应的分类。

1. 结构化的与即时的

结构化工作流指的是在实际工作过程中会反复重复、严格按照某个固定的步骤进行的业务过程。定义此种工作流所需要的各种类型的信息可以通过对业务过程进行详细的分析而得到，从而得到完整的过程定义，并在以后的应用过程中反复使用。大量的办公程序，如公文处理、审批等都属此类。

即时工作流则是针对那些重复性不是很强或没有重复性的工作流程的，关于这类流程执行所需的有关参数（如参加者等）事先无法确定，而必须推迟到过程实例运行时才能确定，同时在执行过程中还可能会发生一些意外的情况。这种动态多变的特点在提供更高灵活性的同时，也为过程的建模与执行带来了更多的复杂性。

2. 面向文档的与面向过程的

前者的侧着点在于将电子形式的文档、图像等在有关的人员之间进行分发，以便能够得到不同人的处理与审阅。现有的文档管理与映像管理系统均属此类。

在面向过程的 WfMS 中，工作流被描述成一序列执行环节。与各环节相应都有待处理的数据对象。各环节的数据对象可以按不同的方式分发到其他环节中去，如可以将数据对象的值作为控制条件，或者依此数据对象组装成其他的数据对象等。高端的 WfMS 一般都属此类系统。

3. 基于邮件的和基于数据库的

前者使用电子邮件来完成过程实例执行过程中消息的传递、数据的分发与事件的通知。低端的系统所使用的经常就是此种方法，它可以充分发挥电子邮件系统在广域环境下的数据分发功能，但整个系统将运行于一种松散耦合的模式下。

在基于数据库的 WfMS 中，所有的数据都保存在某种类型的 DBMS 中，过程的执行实际上就是对这些数据的查询与处理。高端的大规模系统所使用的一般都是此种方法。

4. 任务推动的与目标拉动的

前者指的是从过程的开始逐步地一个环节一个环节地执行，当某个活动实例被处理完之后，后续的有关活动将被创建并被激活，由此直至整个工作流程的完成。这是目前大多数面向过程的 WfMS 所使用的执行方式。

在目标拉动的 WfMS 中，一个业务流程被看成是一个目标。过程实例执行时，该目标将被分解得到多个相互之间按一定约束条件的关联起来的可执行的多个环节，其中各环节还可以当成是子目标而进一步进行分解。在各环节均执行完毕之后，整个过程也就完成了。目标拉动是一种全新的执行方式，下一代的 WfMS 将具有此种特征。应该说明的是，上述分类只是从不同的角度入手的。一般来说，后面那些特点将给 WfMS 带来更好的灵活性，同时也将成为那些能够支持跨机构的大规模复杂工作流管理、面向关键任务的 WfMS 不可缺少的特征。

20.1.4　工作流系统参考模型

不论是从用户的角度来讲，还是从开发者的角度来讲，都需要一个大家都共同遵守的标准，按照此种标准来决定一个工作流管理系统应该包含哪些组成部分，各部分应提供哪些标准的服务。按照此种标准开发出来的 WfMS 将能够满足上面所提出的要求。在这种背景下，WfMC 关于 WfMS 的参考模型就应运而生了。

图 20-2 给出了 WfMC 提出的工作流参考模型。

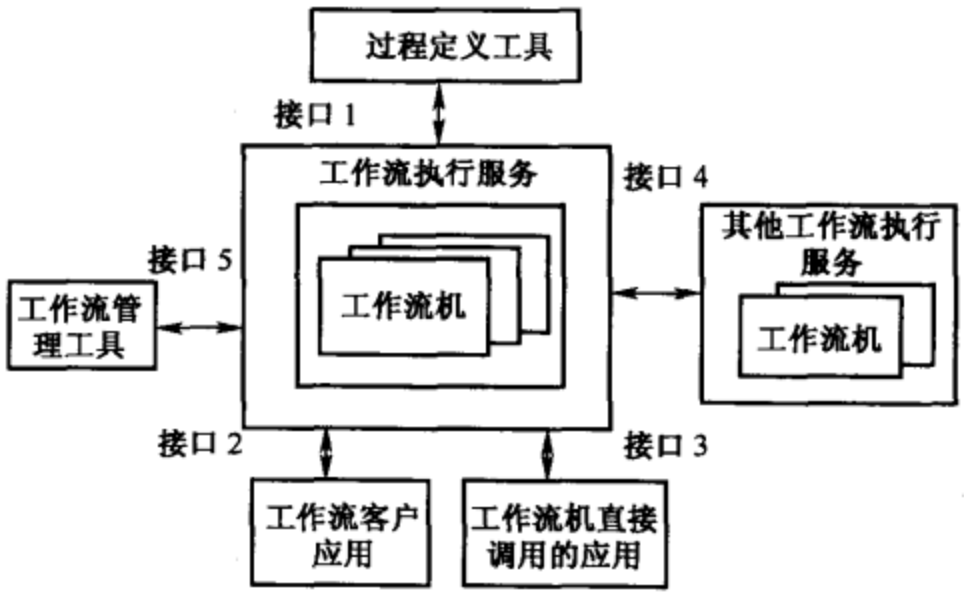


图 20-2　工作流参考模型

系统中各组件的描述如下：

（1）过程定义工具。主要功能是给用户提供一种对实际业务过程进行分析、建模的手段，并生成业务过程的可被计算机处理的形式化描述（过程定义）。这也就是建立阶段功能的主要内容。过程定义工具与工作流执行服务之间的交互是通过接口 1（工作流定义读/写接口）完成的，它为工作流过程定义信息的交换提供了标准的交换格式及 API 调用。

（2）工作流执行服务。它借助于一个或多个工作流引擎来激活并解释过程定义的全部或部分，并同外部的应用程序进行交互来完成工作流过程实例的创建、执行与管理，如过程定义的解释、过程实例的控制（创建、激活、暂停、终止等），在过程各活动之间的游历（控制条件的计算与数据的传递等），并生成有关的工作项通知用户进行处理等，为工作流程的进行提供一个运行时环境。工作流执行服务一般是由一个工作流引擎提供的。在大型 WfMS 中，工作流的运行时控制可能需要多个工作流引擎共同完成，例如某个大的工作流过程中可能会包含多个子流程，这些子流程就可以由另外的工作流引擎来提供运行时控制环境，甚至这些子流程可能需要其他异质的工作流执行服务来完成。这一点涉及到 WfMS 系统之间的互连。为实现有限的互连，需要定义互连模型、互连一致性级别及操作元素集，这些将构成接口 4 的内容。

（3）客户应用程序。它的作用是给用户提供一种手段，以处理过程实例运行过程中需要人工干预的任务。每一个这样的任务就被称作是一个工作项，它包括处理上的一些要求（如处理时间的限制）及待处理的数据对象等。WfMS 将为每一个用户维护一个工作项列表，它表示当前需要该用户处理的所有任务。客户与工作流执行服务之间的接口为接口 2（客户应用程序

API）。WfMS 的各种服务，如会话连接、过程控制、活动控制、过程状态、活动状态、工作项列表的处理以及过程实例的管理等，都可以通过此接口而得到。

（4）被调应用程序。指工作流执行服务在过程实例的运行过程中调用的、用以对应用数据进行处理的应用程序。在过程定义中包含这种应用程序的详细信息，如类型、地址等。目前已有的几种方式包括应用代理（它通过一个标准的接口同执行服务进行交互）、某种标准的互换机制、本地过程调用、远程执行调用等。接口 3 的目标就是提供一些标准的服务供应用代理使用。基于这些服务也可以开发出一些专门的应用直接同工作流执行服务交互。关于这些服务的语义及语法细节还有待更深入的研究。目前初步确定的服务大致可分成会话建立、活动管理（双向的）以及数据处理等几类。

（5）管理及监控工具。其功能是对 WfMS 中过程实例的状态进行监控与管理，如用户管理、角色管理、审计管理、资源控制（包括过程管理及过程状态控制等）。它与工作流执行服务之间的交互是通过接口 5（管理及监控接口）完成的。该接口规范详细描述了需要从过程执行过程所发生的各种事件上捕获和记录的各种信息，如过程实例信息、活动实例信息、工作项信息及远程操作信息等。

上述五个接口被统称为 Workflow API（WAPI）。这些标准的制定对于实现不同厂家的产品之间的互操作（如用一个厂家的管理与监控工具去管理另外一个厂家的工作流执行服务）及基于工作流执行服务开发新的应用具有重要意义。

20.1.5 工作流中间件定义

许多人都知道以文档为中心的支持文档流转的“工作流软件”。今天被看好的工作流软件定位于支持业务流程的自动化，即能够方便地进行处理集成。这些工作流软件以消息中间件或 Web 应用服务器为底层支撑。建立在消息中间件之上的工作流软件一般都有 Windows 或 UNIX 上的客户端，支持与 Web 应用服务器的集成，并提供使用浏览器获得工作列表、执行流程实例和监控管理工作流的能力。

综上所述，本书认为工作流中间件就是用于支持业务流程自动化的中间件，通过工作流中间件，可以实现多个企业应用系统之间的业务流程集成。

由于看好工作流产品市场，2002 年，国内不少软件厂商或系统集成商准备或已经投入到了工作流软件产品的开发行列中。从现有的产品来看，这些软件多数是基于 Web 应用服务器的，工作流引擎运行在 Web 应用服务器上，以浏览器作为工作流程中参与人员的操作界面，具备可视化的流程定义工具等必要的功能模块。产品也简单易用，适合于构建一个单一组织内的工作流应用。也有的厂商在开发针对特定行业应用需要的工作流产品。

20.1.6 工作流中间件与应用系统的结合方式

本节介绍工作流中间件与应用系统的结合方式。

1. 工作流系统在应用中的两种运行模式

应用系统需要在工作流管理系统之上开发实现自己的系统。一般而言，工作流执行服务有以下两种运行方式：

（1）嵌入运行模式（Build-in）。工作流引擎以一个工具包的形式向应用系统提供服务接口，供应用程序在同一个 Java 虚拟机下调用。在这种模式下，工作流引擎可作为业务流程建

模、管理和控制功能的基础构件存在。

（2）独立运行模式（Independence）。在这种模式下，应用程序与工作流引擎分别运行在不同的服务器下，工作流执行服务独立运行，对外提供流程控制服务。应用系统需要开发自己的在其系统内运行的任务表处理器（即工作项列表处理器）。应用程序与工作流引擎之间通过 JMS、RMI、SOAP 等协议进行通信。工作流系统应为应用程序提供基于这些通信协议的接口。现在，越来越多的轻量级异构协议实现（如 ActiveMQ、Hessian、Burlap 等）都提供了基于这些通信协议的接口。

2. 与应用事务的结合

在工作流系统之上开发的业务系统应对应用的事务与工作流引擎的事务进行统一管理。如某个提交任务的过程包括必要的业务处理和调用工作流 API 的流程处理，这两种逻辑应被作为一种原子事务操作统一处理。

当工作流引擎作为构件形式运行于应用系统的环境中（即本节所述模式一），对于事务的管理方式有以下三种：

（1）无事务管理。小型项目，当并发量不大或业务过程简单时，可以采取无事务的管理模式。

（2）独立管理模式。应用系统与工作流管理分别管理自己的事务，二者不进行统一控制。这种模式也仅适用于某些小型项目的开发。

（3）应用统一管理模式。由应用对应用的业务执行过程及流程控制过程进行统一的开始、提交、回滚的事务控制。这是最为安全可靠的应用模式，可以严格地保证业务数据与流程数据的状态一致性。

当采用工作流执行服务独立运行的应用模式时，对事务的管理方式有以下三种：

（1）无事务管理。小型项目，当并发量不大或业务过程简单时，可以采取无事务的管理模式。

（2）独立管理模式。应用系统与工作流管理分别管理自己的事务，二者不进行统一控制。这种模式也仅适用于某些小型项目的开发。

（3）分布式事务管理模式。工作流引擎与应用系统使用符合 JTS 规范的事务管理器进行事务管理。事务的开始与结束由应用系统统一利用 JTS 事务管理规范进行管理。

3. 与应用调用方式的结合

在 WFMC 组织提供的工作流参考模型中，接口 2、3 只是简要定义了客户端应用程序和工作项列表处理程序之间的交互和调用，并没有针对引擎的实现提供具体的标准。大部分工作流产品或多或少引入工具代理（ToolAgent），在业务系统实现时，作为业务组件而存在的应用程序，其被调用的方式一般有以下四种形式：

（1）由工作流引擎自动调用应用程序。在这种情况下，应用程序所属的活动为由引擎自动执行的活动。引擎创建出这个活动后，立即启动活动，并由工作流引擎通过 ToolAgent 自动调用应用程序。

（2）应用程序何时被执行由应用系统决定，应用系统首先从引擎获取需要执行的应用程序，通过调用工作流引擎所提供的接口，由工作流引擎根据应用程序所指定的工具代理激活应用程序。

（3）应用程序何时被执行由应用系统决定，应用系统首先从引擎获取需要执行的应用程

序，应用程序对象记录了它所需要的工具代理，然后通过工具代理激活应用程序的执行。

（4）应用程序何时被执行由应用系统决定，应用系统首先从引擎获取需要执行的应用程序，并直接执行之。

在瘦客户端（B/S 形式）下，应用可不单独开发工具代理实现，即第四种调用方式。

4. 与应用事件的结合

流程实例对象、活动实例对象和工作项实例对象等可以支持事件接口，允许应用实现以构件化形式存在的事件插件，并将此实现注册到工作流定义对象中。业务应用系统在执行工作流 API 控制工作流对象时，由工作流系统自动触发事件实现，这样使工作流开发的应用系统的各个部分不是 build into，而是 plug-in into，即达到配置化。

应用可以将某业务逻辑作为创建流程的事件进行定义，则工作流应用系统在创建流程时将会自动执行此业务逻辑。工作项也可以定义一组任务提醒事件，那么在流程运行阶段，当某执行人有新的工作项产生时，除了在该执行人的标准任务处理器中增加任务记录外，还可以按照定义以一种消息的方式将此传递给执行人。可能的任务消息提醒方式为邮件提醒、即时消息提醒等。如果应用系统需要实现其他的个性化任务提醒方式（如手机提醒），也可通过这种可扩展的事件机制与工作流系统集成。

工作流与应用结合的方面还有很多，如在 WfMC 规范中，只是定义了参与者及参与者类型，并未提供完整的组织模型结构。现在，国内外著名厂商纷纷推出了自己的工作流产品，如 IBM 的 MQWorkflow、BEA 的 WebLogic Workshop、Oracle 工作流等。国内的产品主要有中创软件商用中间件公司的 InfoFlow、西安协同软件的 SynchroFLOW、东方通科技的 TongFlow 等。

20.2 jBPM 工作流引擎

20.2.1 jBPM 项目简介及其特点

本节介绍一个开源的工作流中间件 jBPM。jBPM 全称是 Java Business Process Management，是一种基于 J2EE、参考 WfMC 模型规范定义实现的轻量级工作流管理系统。jBPM 是公开源代码项目，它的使用要遵循 LPGL。jBPM 在 2004 年 10 月 18 日发布了 2.0 版本，并在同一天加入了 JBOSS，成为 JBOSS 企业中间件平台的一个组成部分，它的名称也改成了 JBOSS jBPM。

jBPM 最大的特色就是它的商务逻辑定义部分没有完全采用目前的一些规范，如 WfMC's XPDL，而是采用了它自己定义的 JBoss jBPM Process Definition Language（jpdl）。JPDL 认为一个商务流程可以被看成是一个 UML 状态图，JPDL 就是详细定义了这个状态图的每个部分，如起始、结束状态，状态之间的转换等，同时 jBPM 也加入了 Petri 网的概念来满足流程的定义，比较明显的就是在其规范中应用了令牌（token）。

jBPM 的另一个特点是它使用 Hibernate 来管理它的数据库。Hibernate 是目前 Java 领域应用较广的一种数据持久层解决方案。通过 Hibernate，jBPM 将数据的管理职能分离出去，自己专注于商务逻辑的处理。

最后，jBPM 对流程节点采用预定义事件来触发业务处理逻辑，对在流程中集成程序逻辑

提供了良好的支持，这使以往工作流操作业务数据的弱项有了比较好的解决方式。

20.2.2　软件体系结构及其优点

1. 软件体系结构

整个 jBPM 工作流引擎基本实现了 WMFC 规范的四个接口（如图 20-3 所示），并在 XPDL 规范的基础上做了若干变更和扩展，其结合了 UML 的建模方法，使得工作流的建模更好地融入了整个软件开发过程。

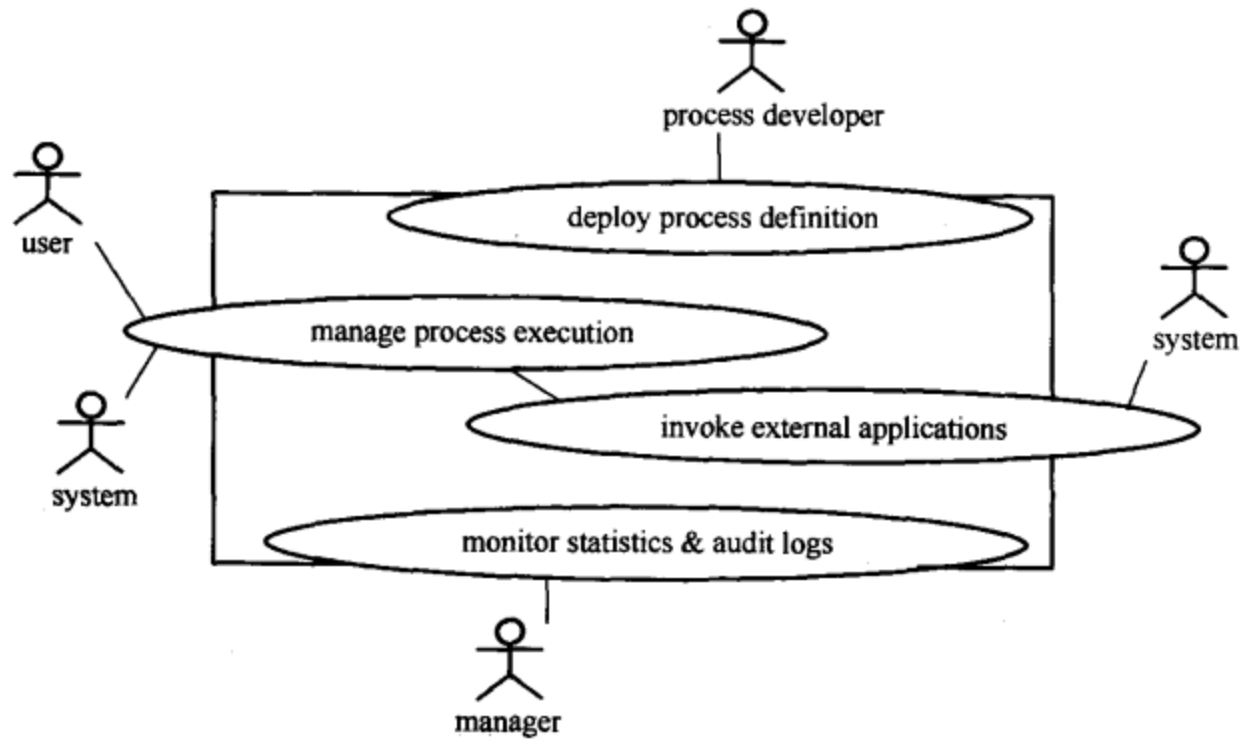


图 20-3　jBPM 所实现 WMFC 模型规范的接口

jBPM 实现的 WFMC 的主要四个接口如下：

（1）定义（Definition）。工作流系统的定义接口使流程开发人员能够部署流程定义。注意，这里的“流程开发人员”可以是业务分析师和软件开发人员的组合。

（2）执行（Execution）。执行接口使用户和系统可以操作流程实例。流程实例是流程定义的执行。流程定义的控制流通过状态机描述。执行接口的两个主要方法是启动一个流程实例通知工作流系统一个状态结束了。

（3）应用（Application）。应用接口代表了工作流系统发起的工作流系统和外部系统之间的交互。当一个用户或系统操作一个流程实例的运行时，会生成一些事件（如一个迁移的执行）。流程定义中可以指定一段响应一个事件的可执行代码逻辑，这段代码和组织内外部的系统打交道。

（4）监控（Audit）。管理人员通过监控接口获得流程运行的确切数据，有时运行日志也可用于审计。

2. jBPM 流程定义的层次

jBPM 作为一个比较完善的工作流引擎，流程定义的内容可以分为四个不同的层次：状态（state）、上下文（context）、程序逻辑（programming logic）和用户界面（UI）。

（1）状态。所有状态和控制流的描述都属于业务流程的状态层。控制流定义了必须被执行的指令的顺序，控制流由我们书写的命令、if 语句、循环语句等确定。在业务流程中的控制

流基本与此一致。但在业务流程中不是使用命令而是使用状态作为基本元素。

流程定义中的状态也指定了执行依赖于哪个参与者。在活动图中，泳道（swimlanes）的标注代表这些参与者的名字。工作流系统使用这些信息构建任务列表，这是一般工作流系统都有的功能。如前所述，参与者可以是人也可以是系统。对于需要人参与的状态，工作流系统必须在运行时计算出具体的个人。这样的计算使工作流系统必须依赖于组织结构信息。

流程定义的控制流包含一组状态和它们之间的关系。状态之间的逻辑关系描述了哪些执行路径可以同时执行，哪些不可以。同步执行路径用分叉（forks）和联合（joins）建模，异步执行路径用判断（decisions）和合并（merges）建模。注意，在大多数模型中，在每个状态之前都有一个隐式合并。

jBPM 的业务流程建模参考了 UML 活动图和 Petri 图。作为一种直观和通用的表达，活动图在图形表述上有一个主要问题，就是没有区分状态和动作，它们都用活动来表示。缺少这种区分（导致状态概念的缺失）是学术界对 UML 活动图的主要批评；UML 活动图的第二个问题是在 UML2.0 版中引入的。当多个变迁（transitions）到达一个活动时，以前的版本规定这是一个默认合并（merge），在 2.0 版中规定这是一个需要同步的默认联合（join）。jBPM 对以上两条构建语义作如下的变化：

1）在用图形描述业务流程时，只建模状态层（状态和控制流），不要包括动作。这意味着图形中的矩形都是状态而不是活动。

2）jBPM 同时引入了 Petri 图的令牌（Token）概念：如果多个变迁到达一个状态，默认定义为不需要同步的合并（merge）。 在流程运行过程中，工作流系统用一个令牌（token）作为指针跟踪流程的状态。这相当于 Von Neuman 体系中的程序计数器。当令牌到达一个状态时，它被分配给工作流系统等待的外部参与者。外部参与者可以是个人、组织或计算机系统。我们定义流程运行的执行人或系统为“参与者”（actor）。只有在工作流系统将令牌分配给一个参与者时，才需要访问组织结构信息。工作流系统通过分配令牌构建任务列表。

（2）流程上下文变量。简称变量，是与流程实例相关的变量。流程开发人员可以使用流程变量存储跨越流程实例整个生命周期的数据。一些工作流管理系统有固定数目的数据类型，另外一些是开发人员可以自己定义的数据类型。

注意，变量也可以用来存放引用（references）。一个变量可以引用数据库中的记录、网络上的文件等。什么时候使用引用，取决于使用引用数据的其他应用。

和流程变量相关的另一个令人感兴趣的方面是：工作流系统如何将数据转化为信息。工作流是用于组织内部跨越各种异构系统实现任务和数据协同的。对于业务流程中人工执行的任务，工作流系统负责从其他相关系统，如 SAP、数据库、CRM 系统、文档管理系统收集数据。在业务流程的每一个人工步骤，只有相关联的数据项被从异构系统中收集和计算。通过这种方式，从不同系统来的数据被转换并展现为信息。

（3）程序逻辑层。如前所述，动作是在流程运行过程中工作流系统响应指定的事件（event）执行的一段程序逻辑（programming logic）。程序逻辑可以是二进制或源代码形式的、用任何语言或脚本编写的软件。程序逻辑层是所有这些软件片断和关于在什么事件发生时调用它们的信息的组合。程序逻辑的例子包括发 E-mail、通过消息代理发消息、从 ERP 系统中获取数据和更新数据库。

（4）用户界面层。一个参与者通过向流程变量中填充数据的事件来触发结束一个状态。

比如，在请假的例子中，老板提供“同意”或“不同意”数据到流程中。某些工作流系统允许指定哪些数据可以填充到流程中，以及它们如何在流程变量中存储。通过这些信息，可以生成从用户收集信息的 UI 表单。

3. jBPM 的工作原理

对 jBPM 来说，工作流由一些节点和用户定义的 Handler 组成，一个工作流从 Start-state 开始，中间经过若干个节点（state、fork、milestone、process-state、decision 和 join），最后结束于 end-state，节点之间通过 transition 来连接，指明当前节点下面要经过的节点。经过节点时，jBPM 会触发相应的 Handler 来完成用户指定的工作。Handler 包括各种事件的 ActionHandler；指定节点指定操作 Actor 的 AssignmenHandler；定义分支节点流程走向的 Handler，包括 fork 节点 ForkHandler，Join 节点 JoinHandler，Decision 节点的 DecisionHandler。

jBPM 节点实际上是 XPDL 规范中 Activity 的细化，是一种特殊的 Activity。Start-state 与 End-state 节点由 jBPM 负责处理，jBPM 从 Start-start 节点开始工作流程，在 End-state 结束工作流程。

State 节点要外部的参与才能进入到下一个状态，比如发出订单流程中需要采购主管批准的步骤。

jBPM 默认的 fork 相当于 XPDL 的 ANDFORK，每个分叉都走；默认的 Join 相当于 XPDL 的 ANDJOIN，所有分叉都到达后，才进入下一节点。

Decision 相当于编程语言中的判断，由 DecisionHandler 决定下一个要经过的节点。通过自己定义 DecisionHandler，用户可以借助 decision 节点实现自动节点，即流程到这里，执行一定操作，自动进入下一个节点。

Process-state 相当于 subflow activity，ProcessInvocationHandler 负责处理具体调用子流程的过程，比如子流程名称、传递的参数等。

4. jBPM 工作流引擎的优点

jBPM 具有灵活简单的特点，特别适用于各种嵌入工作流特性的应用，这主要体现在如下方面：

（1）jBPM 是一组标准的 J2SE 程序，整个核心模块仅仅是一个 Jar 包，可以很方便地部署到各种 servlet 容器、EJB 容器等环境，很好地支持 Java 应用、Web 应用、J2EE 应用等调用环境。因此，jBPM 对应用环境的适应性相当强，可以嵌入各种环境的应用系统中。

（2）提供标准的 Java API 的调用方式，且应用方式灵活，可以由 Java 程序直接调用，也可以通过 EJB 容器（Jboss、Websphere、BEA Weblogic 等应用服务器）进行调用。

（3）每个流程节点的业务功能实现了良好的业务流程逻辑与业务操作逻辑的分离，具体机制是在独立地流程定义文件中定义基于事件的逻辑调用用户的 class 和 Web 表单，因此比较容易独立的设计业务处理逻辑、数据操作和界面表单设计，独立于系统的其他部分，比较好地减轻了系统的耦合性。

（4）流程的定义文件基于通用 XML 格式，并和相关程序和其他资源文件打包为一个 zip 格式的 par 包，方便流程在整个项目管理系统应用中进行部署，以后的流程变更和业务处理逻辑的改变都可以通过此 par 包进行发布，而不会产生格式不兼容的问题。

（5）jBPM 工作流引擎内置实现了流程定义文件的版本化管理，有版本管理机制，流程定义可以不断修改，每次发布流程定义文件，对正在进行的工作流实例延用当前的版本，而

新发起的流程实例将由工作流引擎选择最新版本执行，较好地保证了业务操作的完整性和一致性。

（6）jBPM 数据库可以灵活选择，只要 Hibernate 支持的数据库，jBPM 就支持。这样非常方便进行企业的 EAI 集成。

系统应用工作流的一个可行的途径是：在系统的开发中，将设计的业务流程建成 jBPM 工作流定义文件，不但将业务流程逻辑从分散在各个系统功能模块分离出来，集中到一起，便于变更和管理，而且 jBPM 工作流程包的版本化管理机制也保证了业务流程变更的灵活性和可变性。

20.3 jBPM 开发范例

本节将讲解几个具体的实例，这些例子都关注一个特定的主题并包括相关的注释。这些例子也可以在 jbpm 下载包中的 examples 目录中找到。通过具体的实例，介绍如何用 jpdl 构建基本的流程和怎样用 jbpm 的 API 来管理运行的流程。读者可以试图建立一个工程，并对本节介绍的例子做些修改试验着学习。

Eclipse 用户的使用步骤是这样的：①下载 jbpm-3.0-[version].zip 并解压到自己的系统；②选中 File→Import→Existing Project into Workspace；③单击 Next 按钮，然后浏览到 jbpm 的根目录并单击 Finish 按钮，现在在 Eclipse 的工作区中已经有了一个 jbpm 工程了；④然后可以在 src/java.examples/....找到这些例子。当读者打开这些例子时，可以通过选择 Run→Run As→JUnit Test 命令来运行它们。

Jbpm 拥有一个图形化的定制器来完成定制例子中的 XML 文件。

20.3.1 jBPM 安装配置

下面介绍 jBPM 的安装配置方法。

1. 安装 JDK

可以参考 JDK 的安装指南，别忘记设置系统变量 JAVA_HOME。

2. 安装 Ant

Ant 是使用 jBPM 必须的一个工具，jBPM 中的很多操作都要用到 Ant。

安装方法：

（1）下载 Ant，网址为 http://archive.apache.org/dist/ant/binaries/ ，选择其中一个，例如 apache-ant-1.6.5-bin.zip。

（2）解压到 D:\ant（其他目录也可以）。

（3）设置如下系统变量： ANT_HOME=d:\ant 。

（4）把%ANT_HOME%\bin 加入到系统变量 PATH 中。

3. 安装 Eclipse

Eclipse 不是开发 jBPM 必须的工具，但它是对 jBPM 开发很有帮助的工具，特别是 jBPM 提供了一个 Eclipse 插件用来辅助开发 jBPM 。本书使用的版本是 Eclipse3.2。关于 Eclipse 的安装步骤，可以参考相应的安装文档。

4. 安装 jBPM

jBPM 的下载地址：http://www.jboss.com/products/jbpm/downloads。

相关组件如下：

- JBoss jBPM 是 jBPM 的软件包。
- JBoss jBPM Starters Kit 是一个综合包，它包括了 jBPM 软件包、开发插件、一个配置好了的基于 JBoss 的 jBPM 示例、一些数据库配置文件示例。
- JBoss jBPM Process Designer Plugin 是辅助开发 jBPM 的 Eclipse 插件。
- JBoss jBPM BPEL Extension jBPM 关于 BPEL 的扩展包。

本书选择下载 JBoss jBPM Starters Kit，下载后解压到 D:\jbpm-starters-kit-3.1 目录下，含有五个子目录：

- Jbpm，jBPM 的软件包。
- jbpm-bpel，只含有一个网页。
- jbpm-db，各种数据库 hibernate 配置文件示例，有些还包含相应的 jdbc 驱动程序。
- jbpm-designer，辅助开发 jBPM 的 Eclipse 插件，具体在 jbpm-gpd-feature 子目录中。
- jbpm-server，一个已经配置好了的基于 JBoss 的 jBPM 示例。

5. 查看 jBPM 自带的一个范例

在 JBoss jBPM Starters Kit 的 jbpm-server 目录是一个已经配置好的 jBPM 示例，双击 jbpm-server 目录下的 start.bat 文件，启动 JBoss 服务。这时会打开一个 DOS 窗口，启动完成后，日志会不断输出，其中最后一句是 13:55:39,937 DEBUG [StaticNotifier] going to wait for (CMD_EXECUTOR, java.lang.Object@1df59bd)，这表示 jBPM 在开始工作了，它不断进行轮询。

打开网页：http://localhost:8080/jbpm/，显示一个已经用 jBPM 开发好的用户定单流程，具有下单、审核、估价等流程。它所用的数据库是一个内置的数据库。

以 cookie monster 用户登录，选择 create new web sale order 可以创建一个定单。在界面左边是填写的定单情况，右边是整个定货流程的示意图，红色框表示流程进行到哪一步了。填写好定单后，选择 Save and Close Task，完成定单提交。

选择右上角的 Login as another user，以另外一个用户名 ernie 登录。这时可以看到 ernie 用户的任务列表中多了一项。

单击该任务，进入任务画面。这个示例对中文的支持不好，全都显示成了 unicode 码了。在 comment 项填写意见，单击 OK 按钮，进入到下一步。如果选择 more info needed 按钮，则退回给 cookie monster 用户修改定单。

6. 数据库初始化

jBPM 需要数据库支持，jBPM 会把自己的一个初始化数据存储到数据库，同时工作流的数据也是存储到数据库中的。jBPM 使用 Hibernate 来做为自己的存储层，因此只要是 Hibernate 支持的数据库，jBPM 也就支持。

在上面 JBoss 自带的示例中并没有设置数据库，那是因为 jBPM 默认使用的是内存数据库 hsqldb 。

下面通过 MySQL 数据库来介绍 jBPM 的数据库初始化操作。

（1）安装 MySQL。首先参考 MySQL 的安装文档安装 MySQL 程序。本书所用 MySQL

版本为 MySQL 4.1（for windows）。再找一个 MySQL 客户端，目的是方便查看数据库中的数据，可以使用 MySQL 网站上免费提供的 MySQL Query Brower。

（2）建库。在 MySQL 中创建一个库，库名：jbpm。

（3）生成建表的 SQL 语句并建表 。将 jbpm-starters-kit-3.1.1 下的子目录 jbpm 改名为 jbpm.3 ，否则在执行下面的 ant 命令时会报如 jbpm.3 目录不存在的错误：

```
D:\jbpm-starters-kit-3.1.1\jbpm-db\build.xml:361: The following error
      occurred while executing this line:
D:\jbpm-starters-kit-3.1.1\jbpm-db\build.xml:68: Basedir D:\jbpm-starters-
      kit-3.1.1\jbpm.3 does not exist
```

在 DOS 窗下进入 D:\jbpm-starters-kit-3.1.1\jbpm-db 目录，执行如下命令：

```
ant mysql.scripts
```

执行成功后，在 D:\jbpm-starters-kit-3.1.1\jbpm-db\build\mysql\scripts 目录里生成了四个 sql 文件。在 MySQL 客户端中执行 mysql.create.sql 脚本，这样将在 jbpm 库中创建相关的数据表。

20.3.2 Hello World 实例

首先介绍一个 Hello World 实例。一个流程定义是一个由节点（node）和变迁（transition）组成的有向图。变迁也可以叫做转换，所以本书中有的地方也叫转换。Hello World 流程有三个节点。为了看这些组成部分是如何组合的，以一个没有用设计器开发的简单流程开始。图 20-4 就是 Hello World 流程的图形表示。

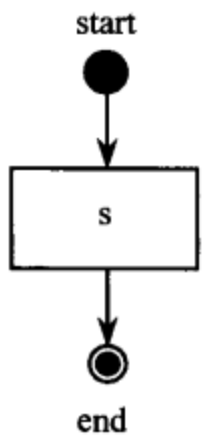


图 20-4 Hello World 流程图

代码如下：

```
public void testHelloWorldProcess() {
  //这个方法显示了流程定义以及流程定义的执行
  //这个流程定义有三个节点：
  //一个未命名的开始状态（start-state），一个 s 状态和一个 end 结束状态（end-state）
  //后面一行解析一行 XML 文本，构造了 ProcessDefiness 对象
  //ProcessDefiness 是一个流程描述的 Java 对象表示
  ProcessDefinition processDefinition = ProcessDefinition.parseXmlString(
    "<process-definition>" +
    "  <start-state>" +
    "    <transition to='s' />" +
    "  </start-state>" +
    "  <state name='s'>" +
```

```
    "    <transition to='end' />" +
    "  </state>" +
    "  <end-state name='end' />" +
    "</process-definition>"
  );

  //下面创建了一个流程定义的执行实例
  //构建之后，流程执行就有一个主执行路径（根令牌），定位在开始状态
  ProcessInstance processInstance =
      new ProcessInstance(processDefinition);

  Token token = processInstance.getRootToken();

   assertSame(processDefinition.getStartState(), token.getNode());

  //开始执行流程，离开开始状态
  token.signal();
  // signal 方法将阻塞，直到流程定义进入等待状态

  // 流程执行进入第一个等待状态，即 s 状态
  //所以执行的主路径将处于 s 状态
  assertSame(processDefinition.getNode("s"), token.getNode());

  //调用 signal 方法，将离开 s 状态
  token.signal();
  //执行 signal 方法之后，进入 end 状态
    assertSame(processDefinition.getNode("end"), token.getNode());
  }
```

20.3.3　数据库实例

jbpm 的一个基本特性就是假如运行的流程处于等待状态，jbpm 可以把它们保存到数据库中。下一个例子演示了如何保存流程实例到 jbpm 的数据库中。这个例子还给出了一个可能发生的处理上下文（context）的建议。比如，Web 应用程序中用户代码的一部分开启了一个流程，并且把运行实例保存到数据库中。后来，一个消息驱动 bean 从数据库中加载了这个流程实例并且继续执行这个流程。

实例代码如下：

```
public class HelloWorldDbTest extends TestCase {

  static JbpmConfiguration jbpmConfiguration = null;

  static {
    //在 src/config.files 目录可以看到配置文件的范例
    //通常在 jbpm.cfg.xml 资源文件包含了配置信息
    //本例使用 XML String 来传入配置信息

    //首先创建一个 JbpmConfiguration 对象
```

```
//因为 JbpmConfiguration 可以由系统所有线程使用，所以创建为静态对象

jbpmConfiguration = JbpmConfiguration.parseXmlString(
  "<jbpm-configuration>" +

  //jbpm-context 机制将 jbpm 核心引擎和 jbpm 从环境中使用的服务分离

  "  <jbpm-context>" +
  "    <service name='persistence' " +
  "  factory='org.jbpm.persistence.db.DbPersistenceServiceFactory' />" +
  "  </jbpm-context>" +

  //jbpm 使用的所有资源文件都要引用 jbpm.cfg.xml 文件

  "  <string name='resource.hibernate.cfg.xml' " +
  "  value='hibernate.cfg.xml' />" +
  "  <string name='resource.business.calendar' " +
  "  value='org/jbpm/calendar/jbpm.business.calendar.properties' />" +
  "  <string name='resource.default.modules' " +
  "  value='org/jbpm/graph/def/jbpm.default.modules.properties' />" +
  "  <string name='resource.converter' " +
  "  value='org/jbpm/db/hibernate/jbpm.converter.properties' />" +
  "  <string name='resource.action.types' " +
  "  value='org/jbpm/graph/action/action.types.xml' />" +
      "  <string name='resource.node.types' " +
      "  value='org/jbpm/graph/node/node.types.xml' />" +
      "  <string name='resource.varmapping' " +
      "  value='org/jbpm/context/exe/jbpm.varmapping.xml' />" +
      "</jbpm-configuration>"
    );
  }

  public void setUp() {
    jbpmConfiguration.createSchema();
  }

  public void tearDown() {
    jbpmConfiguration.dropSchema();
  }

 public void testSimplePersistence() {
//在下边调用的三个方法之间，所有的数据都是通过数据库传输的
//在这个单元测试中，这三个方法依次被调用
//因为我们想测试一个完整的流程
//但是在实际的应用中，这些方法代表对服务器不同的请求
//既然从一个空的干净的内存数据库开始，我们必须从部署流程开始
//在现实中，这只被流程开发人员部署一次
```

```
    deployProcessDefinition();
//假设当在 web 应用程序中一个用户提交了一个表单想开启一个流程实例

    processInstanceIsCreatedWhenUserSubmitsWebappForm();

    //然后，一个异步消息的到达将会使流程继续执行
    theProcessInstanceContinuesWhenAnAsyncMessageIsReceived();
  }

  public void deployProcessDefinition() {
//这个测试程序演示了一个流程定义和这个流程定义的运行
//这个流程定义包括三个节点
//一个未命名的开始状态，一个名字为 s 的状态和一个名字为 end 的结束状态
    ProcessDefinition processDefinition = JpdlXmlReader.parse(
      "<process-definition name='hello world'> +
      "  <start-state name='start'> +
      "    <transition to='s' /> +
      "  </start-state> +
      "  <state name='s'> +
      "    <transition to='end' /> +
      "  </state> +
      "  <end-state name='end' /> +
      "</process-definition>
    );

    //开启一个新的持久化 session
    JbpmSession jbpmSession = jbpmSessionFactory.openJbpmSession();
    //在这个持久化 session 上开启一个事务
    jbpmSession.beginTransaction();

    //保存流程定义到数据库中
    jbpmSession
        .getGraphSession()
        .saveProcessDefinition(processDefinition);

    //提交事务
    jbpmSession.commitTransaction();
    //关闭 jbpmSession
    jbpmSession.close();
}

public void processInstanceIsCreatedWhenUserSubmitsWebappForm() {
    //这个方法中的代码应该在一个 struts-action 里
    //或在一个 jsf 的管理 bean 中（JSF managed bean）
    //开启一个新的持久化 session
    JbpmSession jbpmSession = jbpmSessionFactory.openJbpmSession();
    //并且在这个持久化 session 上开启一个事务
    jbpmSession.beginTransaction();
```

```
//现在可以在数据库中查找我们在上边部署的流程定义
ProcessDefinition processDefinition =
  JbpmSession.getGraphSession().findLatestProcessDefinition("hello world");

//用从数据库中取回的流程定义，可以像在 hello world 例子中那样
//产生流程定义的一个执行实例（那个没有持久化）
ProcessInstance processInstance = new ProcessInstance(processDefinition);
Token token = processInstance.getRootToken();
assertEquals("start", token.getNode().getName());
//开始运行流程实例
token.signal();
//现在流程在状态 s 上
assertEquals("s", token.getNode().getName());
//现在流程实例被保存到数据库中
//所以流程实例的当今状态也被保存到了数据库


jbpmSession
    .getGraphSession()
    .saveProcessInstance(processInstance);
//下边的方法从数据库中取回流程实例
//并且通过提供另外一个外部信号（signal）使流程继续执行

//在这个 Web 应用程序最后的 action 中，事务被提交
jbpmSession.commitTransaction();
//关闭 jbpmSession
  jbpmSession.close();
}

public void theProcessInstanceContinuesWhenAnAsyncMessageIsReceived() {
  //这个方法中的代码应该在一个消息驱动 bean 中开启一个新的持久化 session
  JbpmSession jbpmSession = jbpmSessionFactory.openJbpmSession();
  //并且在这个持久化 session 上开启一个事务
  //请注意在你的应用服务器中，通过 jdbc 连接数据源也是可以的
  jbpmSession.beginTransaction();
  GraphSession graphSession = jbpmSession.getGraphSession();

  ProcessDefinition processDefinition =
      graphSession.findLatestProcessDefinition("hello world");

  //在所有的流程实例中查找这个流程实例
  List processInstances =
      graphSession.findProcessInstances(processDefinition.getId());
  //在这个单元测试的上下文环境中只能有一个运行的实例
  //在现实开发中，流程实例的 id 号可以从到达的信息内容中，或从用户的选择中提取出来

  ProcessInstance processInstance =
      (ProcessInstance) processInstances.get(0);
```

```
    //现在可以继续执行这个流程实例
    //这个流程实例向执行主路径（root token）发出信号
    processInstance.signal();

    //这个 signal 过后，流程实例已经到达结束状态了
    assertTrue(processInstance.hasEnded());

    //现在可以更新数据库中的流程实例的状态
    graphSession.saveProcessInstance(processInstance);

    //在这个消息驱动 bean 的最后，事务被提交
    jbpmSession.commitTransaction();
    //关闭 jbpmSession
    jbpmSession.close();
  }
}
```

20.3.4　上下文实例：流程变量

流程变量包含流程执行时期的上下文信息。流程变量和 java.util.Map 非常相像，也是变量名（name）到变量值（value）的对应关系，就像 java.util.Map 中对应的是 java object。

流程变量被当成流程实例的一部分保存。为了简化，在这个例子中仅仅演示一下和流程变量相关的 API，并没有持久化。

```
//这个例子依旧从 hello world 流程开始
ProcessDefinition processDefinition = JpdlXmlReader.parse(
  "<process-definition>" +
  "<start-state>" +
  "<transition to='s' />" +
  "</start-state>" +
  "<state name='s'>" +
  "<transition to='end' />" +
  "</state>" +
  "<end-state name='end' />" +
  "</process-definition>"
);

ProcessInstance processInstance = new ProcessInstance(processDefinition);

//从流程实例中得到和流程变量一起工作的上下文实例
ContextInstance contextInstance =
  processInstance.getContextInstance();

//在流程离开开始状态之前，要在流程实例的上下文环境中设定一些流程变量
contextInstance.setVariable("amount", new Integer(500));
contextInstance.setVariable("reason", "i met my deadline");
//从现在开始，流程变量和流程实例关联了起来
//流程变量现在可以是用户通过调用列出来的 API 访问
//也可以在动作（action）和节点的执行时访问
```

```
//流程变量被作为流程实例的一部分保存在数据库中

processInstance.signal();

//变量通过上下文环境实例（contextInstance）来访问

assertEquals(new Integer(500), contextInstance.getVariable("amount"));
assertEquals("i met my deadline", contextInstance.getVariable("reason"));
```

20.3.5 任务分配实例

在下一个例子中介绍怎样分配给用户一个任务。因为 jbpm 工作流引擎和组织模型是分离的，因此，必须完成 AssignmentHandler 接口的一个具体实现来分配参与者(actor)的任务(task)。

```
public void testTaskAssignment() {
//下边的流程是基于 hello world 流程的。状态节点被替换为任务节点
//任务节点在 jpdl 中代表一个等待状态，并且它能产生待完成的任务
//这些任务必须被完成流程才能继续执行
  ProcessDefinition processDefinition = JpdlXmlReader.parse(
    "<process-definition name='the baby process'>" +
    "<start-state>" +
    "<transition name='baby cries' to='t' />" +
    "</start-state>" +
    "<task-node name='t'>" +
    "<task name='change nappy'>" +
    "<assignment class='org.jbpm.tutorial.taskmgmt.NappyAssignmentHandler' />" +
    "</task>" +
    "<transition to='end' />" +
    "</task-node>" +
    "<end-state name='end' />" +
    "</process-definition>"
  );

//产生流程定义文件的一个运行实例
  ProcessInstance processInstance = new ProcessInstance(processDefinition);
  Token token = processInstance.getRootToken();
  //开始执行流程，通过默认的变迁（transition）离开开始状态（start-state）
token.signal();
// signal 方法将一直被阻塞，一直到执行的流程进入一个等待状态
//在这里，它是个任务节点（task-node）

  assertSame(processDefinition.getNode("t"), token.getNode());
//当执行的流程到达任务节点时，一个“换尿布”的任务被创建
//并且 NappyAssignmentHandler 将被调用，来决定这个任务应该分给谁
//这里 NappyAssignmentHandler 返回的是“爸爸”

//实际上，任务应该是通过 org.jbpm.db.TaskMgmtSession 中的方法从数据库中获取出来的
//既然这里并不想把复杂的持久化引入这里例子
//仅仅获取流程实例中的第一个任务实例（在这个测试当中只有一个任务）

  TaskInstance taskInstance = (TaskInstance)
```

```
    processInstance.getTaskMgmtInstance().getTaskInstances().iterator().next();

    //现在检查一下流程实例是不是真正地分给了“爸爸”
    assertEquals("papa", taskInstance.getActorId() );

    //现在假定“爸爸”已经完成了他的任务，并且把任务标记为已完成
    taskInstance.end();
    //既然这是最后（惟一）一个要完成的任务
  //这个任务的完成将触发流程实例的继续执行

    assertSame(processDefinition.getNode("end"), token.getNode());
  }
```

20.3.6　自定义 action 实例

Action 是把开发人员编写的 Java 代码集成到 jbpm 流程中的一套机制。Action 可以和它自己的节点（node）关联（如果它们在流程图中是相关的）。Action 也可以被放置在以下事件中，如变迁（transition）发生，离开节点或进入一个节点。这种情况下，action 并不是流程图表示的一部分，但是在流程的执行过程中，当执行激活这些事件时，action 也会被执行。

先看一下在实例中将要用的 action 的实现：MyActionHandler。这个 action 处理动作的实现并没有做很复杂的事情，它仅仅是把 boolean 变量 isExecuted 设置为 true。isExecuted 变量是静态的，这样就可以像在 action 外部检验它的值一样在 action 处理动作的内部访问它。

```
//MyActionHandler 展示了一个在 jbpm 流程执行时可以执行一些用户代码的类
public class MyActionHandler implements ActionHandler {
//在每个测试的开始（在 setup 方法中），成员 isExecuted 都被设置成 false
public static boolean isExecuted = false;
//action 将要把 isExecuted 设置为 true
//这样当 action 被执行时就可以在这个单元测试中显现出来

public void execute(ExecutionContext executionContext) {
    isExecuted = true;
  }
}
```

在每个测试的开始，都要把静态变量 MyActionHandler.isExecuted 设置为 false。

```
//每个测试开始之前，都把 MyActionHandler 的成员静态变量 isExecuted 设置为 false
public void setUp() {
MyActionHandler.isExecuted = false;
}
```

下面在一个变迁（transition）上开始 action。

```
public void testTransitionAction() {
//下边的流程是 hello world 流程的一个变种
//在由状态 s 向结束状态变迁上放了一个 action
//这个测试的目的是想展示把 jbpm 流程和 Java 代码集成起来是多么的容易
    ProcessDefinition processDefinition = JpdlXmlReader.parse(
      "<process-definition>" +
      "<start-state>" +
```

```
      "<transition to='s' />" +
      "</start-state>" +
      "<state name='s'>" +
      "<transition to='end'>" +
      "<action class='org.jbpm.tutorial.action.MyActionHandler' />" +
      "</transition>" +
      "</state>" +
      "<end-state name='end' />" +
      "</process-definition>"
    );
//开启流程定义的一个执行实例
    ProcessInstance processInstance =
      new ProcessInstance(processDefinition);
    //下一个 signal 将使执行的流程实例离开开始状态并进入状态 s
    processInstance.signal();
    //这里是为了展示 MyActionHandler 没有被执行
    assertFalse(MyActionHandler.isExecuted);
    //并且执行的主路径被定位在状态 s 上
    assertSame(processDefinition.getNode("s"),
            processInstance.getRootToken().getNode());
    //下一个 signal 方法将激活 root token 的执行
//这个 token 将要带着这个 action 发生转换（transition）
//并且这个 action 在 signal 方法被调用时会被执行
    processInstance.signal();
    //这里可以看到当调用 signal 方法时，MyActionHandler 被执行
    assertTrue(MyActionHandler.isExecuted);
  }
```

下一个例子演示了相同的 action，但是 action 分别放置在 enter-node 和 leave-node 事件上。注意一个节点（node）和变迁（transition）相比拥有多于一个的事件类型，变迁（transition）只拥有一个事件。因此放置在 node 上的 action 应该被放置在一个事件元素上。

```
ProcessDefinition processDefinition = JpdlXmlReader.parse(
  "<process-definition>" +
  "<start-state>" +
  "<transition to='s' />" +
  "</start-state>" +
  "<state name='s'>" +
  "<event type='node-enter'>" +
  "<action class='org.jbpm.tutorial.action.MyActionHandler' />" +
  "</event>" +
  "<event type='node-leave'>" +
  "<action class='org.jbpm.tutorial.action.MyActionHandler' />" +
  "</event>" +
  "<transition to='end'/>" +
  "</state>" +
  "<end-state name='end' />" +
  "</process-definition>"
);
```

```
ProcessInstance processInstance = new ProcessInstance(processDefinition);

assertFalse(MyActionHandler.isExecuted);
//下一个 signal 将会使流程执行，离开开始状态进入名字为 s 的状态
//这样进入 s 状态因此执行 action
processInstance.signal();
assertTrue(MyActionHandler.isExecuted);

//重新设置 MyActionHandler.isExecuted 为 false
MyActionHandler.isExecuted =;
//下一个 signal 将要触发流程的执行来离开状态 s
//这样 action 将会再执行一次
processInstance.signal();
//结果应该如先前所预料的一样
assertTrue(MyActionHandler.isExecuted);
```

20.4　小结

本章介绍了工作流中间件的技术原理和开发实例，包括如下内容：

- 工作流的概念工作流中间件的基本功能。
- 开源工作流引擎 jBPM 项目，它的体系结构和特点。
- jBPM 的安装配置开发实例。

第 21 章　中间件技术的最新进展

中间件技术发展极快，各种中间件类型不断涌现，本章对中间件技术的最新进展做概要的介绍。

21.1　RFID 中间件

21.1.1　RFID 技术

RFID 是 Radio Frequency IDentification 的缩写，即射频识别，俗称电子标签。RFID（射频识别）是一种非接触式的自动识别技术，它通过射频信号自动识别目标对象并获取相关数据，识别工作无须人工干预，可工作于各种恶劣环境。RFID 技术可识别高速运动物体，并可同时识别多个标签，操作快捷方便。

RFID 系统的基本组成部分如下：

（1）标签（Tag）：由耦合元件及芯片组成，每个标签具有惟一的电子编码，附着在物体上标识目标对象。

（2）阅读器（Reader）：读取（有时还可以写入）标签信息的设备，可设计为手持式或固定式。

（3）天线（Antenna）：在标签和读取器间传递射频信号。

（4）RFID 中间件：扮演 RFID 标签和应用程序之间的中介角色。

RFID 技术的基本工作原理是：标签进入磁场后，接收解读器发出的射频信号，凭借感应电流所获得的能量发送出存储在芯片中的产品信息（Passive Tag，无源标签或被动标签），或者主动发送某一频率的信号（Active Tag，有源标签或主动标签）；解读器读取信息并解码后，送至中央信息系统进行有关数据处理。

RFID 应用的范围遍及制造、物流、医疗、运输、零售、国防等。Gartner Group 认为，RFID 是 2005 年建议企业可考虑引入的十大策略技术之一，然而其成功的关键除了标签（Tag）的价格、天线的设计、波段的标准化、设备的认证之外，最重要的是要有关键的应用软件（Killer Application），才能迅速推广。而中间件（Middleware）可称为是 RFID 运作的中枢，因为它可以加速关键应用的问世。

21.1.2　RFID 中间件

看到目前各式各样 RFID 的应用，企业最想问的第一个问题是：“我要如何将我现有的系统与这些新的 RFID Reader 连接？”这个问题的本质是企业应用系统与硬件接口的问题。因此，通透性是整个应用的关键，正确抓取数据、确保数据读取的可靠性，以及有效地将数据传送到后端系统都是必须考虑的问题。传统应用程序与应用程序之间（Application to Application）数据通透是通过中间件架构解决，并发展出各种 Application Server 应用软件；同理，中间件的

架构设计解决方案便成为 RFID 应用的一项极为重要的核心技术。

RFID 中间件扮演 RFID 标签和应用程序之间的中介角色，从应用程序端使用中间件所提供一组通用的应用程序接口（API），即能连到 RFID 读写器，读取 RFID 标签数据。这样一来，即使存储 RFID 标签情报的数据库软件或后端应用程序增加或改由其他软件取代，或者读写 RFID 读写器种类增加等情况发生时，应用端不需修改也能处理，省去多对多连接的维护复杂性问题。

RFID 中间件是一种面向消息的中间件（Message-Oriented Middleware，MOM），信息（Information）是以消息（Message）的形式从一个程序传送到另一个或多个程序。信息可以以异步（Asynchronous）的方式传送，所以传送者不必等待回应。面向消息的中间件包含的功能不仅是传递（Passing）信息，还必须包括解译数据、安全性、数据广播、错误恢复、定位网络资源、找出符合成本的路径、消息与要求的优先次序以及延伸的除错工具等服务。

RFID 中间件可以从架构上分为两种：

（1）以应用程序为中心（Application Centric）的设计概念是通过 RFID Reader 厂商提供的 API，以 Hot Code 方式直接编写特定 Reader 读取数据的 Adapter，并传送至后端系统的应用程序或数据库，从而达成与后端系统或服务串接的目的。

（2）以架构为中心（Infrastructure Centric）。随着企业应用系统的复杂度增高，企业无法负荷以 Hot Code 方式为每个应用程式编写 Adapter，同时面对对象标准化等问题，企业可以考虑采用厂商所提供标准规格的 RFID 中间件。这样一来，即使存储 RFID 标签情报的数据库软件改由其他软件代替，或读写 RFID 标签的 RFID Reader 种类增加等情况发生时，应用端不做修改也能应付。

21.1.3　RFID 中间件的功能

总的来说，RFID 中间件是将底层 RFID 硬件和上层企业应用结合在一起的粘合剂。虽然原则上的中间件是横向的软件技术，但在 RFID 系统中，为使其更适用于特定行业，RFID 中间件往往会针对行业做一定的适配工作。

在 RFID 系统这种具体情况下，中间件层除通常的功能外，还有以下特定功能：

（1）使阅读/写入更加可靠。

（2）把数据通过读卡器网络推或拉到正确位置（类似路由器）。

（3）监测和控制读卡器。

（4）提供安全读/写操作。

（5）降低射频干扰。

（6）处理标签型和读卡器型事件。

（7）应用通知。

（8）接受并且转发来自应用的中断指令。

（9）给用户提供异常告警。

从体系结构上讲，RFID 中间件还可以分为子层，包括边缘层和集成层。边缘与集成层的分离可以提高可伸缩性并降低客户成本，因为边缘层是轻量级的，成本又低。

边缘层定期轮询读卡器删除复本，并进行筛选和设备管理。边缘服务器还负责创建 ALE 事件并将其分派至集成层。

集成层接收多个 ALE 事件并将其合并到涉及各种系统和人员的工作流中，这些系统和人员是更大的业务流程的一部分。集成层通过基于标准的 JCA 适配器与打包应用程序（如仓库管理系统或产品信息管理系统）交互。通过一些提供抽象层的控件和开源框架，该层也可以与系统一起工作，抽象层将后端组件公开为可重用组件。集成层也可以通过 Web 服务接口与对象名解析服务进行通信、利用 B2B 消息通过防火墙中的网关与外部系统进行通信。

（1）边缘层。边缘层通常提供的功能有标准的设备支持和管理、高效的捕获数据和过滤数据、创建 ALE 事件并将其分派至集成层等。

边缘层应该支持丰富的设备，包括流行的 RFID 读卡器和打印机，以及各类条形码识别器、指示灯、LED 显示、电眼和可编程逻辑控制器（PLC）。它可以运行在单独的计算机上，也可以嵌入新出现的其他设备（如路由器）中。应该符合 EPCglobal 应用级别事件（ALE）标准，提供易于使用的标签写入和其他类型设备的扩展功能，并支持 ISO 和 EPCglobal 标签标准（包括 Gen2）。

随着 RFID 技术的应用日益广泛，企业需要处理分布在全球各个供应链中数以千计的读卡器的输入信息。快速发展将会挑战可伸缩性。需要处理的数据量非常庞大，这样就产生了更大的挑战。

要处理这种级别的数据流量，需要使用非阻塞 I/O 机制。当众多用户同时使用 RFID 访问同一个应用程序时，大多数中间件解决方案为每个客户端打开一个插口，并为每个用户建立独有的线程。这种阻塞 I/O 技术严重限制了性能和可伸缩性。与此相反，非阻塞 I/O 可以使 BEA WebLogic Server 之类的中间件能够在多个并发用户中复用少量的读卡器线程，确保较高的性能和可伸缩性。

在处理读卡器的大流量数据流和进行消息传递时，需要大量使用 I/O 和网络。边缘服务器的 CPU 利用主要用于边缘服务器的复本检测和模式匹配。在要处理的数据量确定的情况下，网络带宽也会成为一个问题。“批量数据传输”即将多个请求包装在一个数据包中可以舒缓网络堵塞问题。它还可以减少多个请求通过安全层及其他代码层所需的时间。

（2）集成层。集成层接收多个 ALE 事件并将其合并到涉及各种系统和人员的工作流中，这些系统和人员是更大的业务流程的一部分。它通常提供的功能有安全性、互操作性、管理、消息传递和集成等。

对于 RFID 来说，大量相关的潜在敏感数据使得安全性成为 RFID 系统至关重要的一个方面。最低级别，安全管理可以防止读卡器被关闭以及记录项被窃取。因此，必须通过验证、授权或审计来保护管理接口，这也许会通过 SSL（Secure Socket Layer，安全套接字层）来实现。

互操作性对于确保 RFID 的成功实现具有多重重要意义。或许，最迫切的需求是基于标准的 JCA 适配器要有效连接到诸如仓库管理系统或运输管理系统之类的应用程序。仅仅能够以私有格式发布 JMS 消息或事件是远远不够的；应用程序供应商，如 SAP、Yantra 和 Manhattan，要求事件以确定的格式呈现。适配器可以填平鸿沟，将信息以可接受的格式传播至恰当的应用程序。中间件解决方案应能够提供和支持适用于关键应用程序的适配器。

在其他方面，开箱即用的互操作性同样至关重要。例如，中间件应能够与防火墙提供者、身份验证、授权和审计提供者、负载均衡系统和 JMS 供应商进行互操作。读卡器的互操作性也非常重要。尽管读卡器通信协议的标准化一直在进行，但在出现一个占据主导地位的标准之

前，每个中间件供应商都必须提供一个读卡器抽象层和互操作性解决方案。

设计良好的架构可以将读卡器抽象层置于边缘层，使得集成层具有读卡器无关性。也就是说，集成层无需考虑特定的读卡器协议或格式。

随着 RFID 在各个供应链中启用，管理整个架构的能力成为必要。以高级别来看，RFID 的监控和管理包括两个方面：设备管理和对读卡器的配置。管理员需要一个管理整个架构的接口，该接口应该包含在一个集中式的门户框架中。

RFID 管理解决方案还应与现有的管理提供者（如 HP OpenView 或 Tivoli）无缝集成，需要支持 SNMP 和 JMX 之类的标准协议。理想的情况是，一个中央配置主机应能够将配置推行至边缘和整个供应链中的读卡器。

保证的 exactly-once（只发送一次）消息处理语义非常难以实现。即使在干预式消息传输过程中，发送方和接收方也都存在着消息中断的可能性。大部分中间件解决方案没有考虑确保 exactly-once 消息语义的需求。但是，如果不考虑这个问题会产生一系列问题——例如，单次交付报告会被无意地交付多次。仓库管理员就会认为向合作伙伴发送了两份报告而非一份；在不同的时间和地点多次发生这种情况，其效果就会非常惊人。

另一个重要因素是确保对消息排队和出队的事务性保证。如果消息没有按事务顺序排队，队列就没有保证；类似地，出队的消息也无法保证经过完全处理。其他方面的考虑主要是围绕操作幂等性——重新执行已部分完成的操作是否安全。

有时需要进行连接的计算，特别是在发送方和接收方地理位置较远时。在这种情况下，如果一方依赖于另一方的同步响应，则网络中断就会带来整个操作的终止。这种情况下应该设为异步通信。

通常使用 JMS 进行异步通信。但是，如果 JMS 提供者在接收方，发送方如果无法对消息进行排队就会阻塞（或者引发错误并负责重新尝试发送）。因此，在发生这些问题的情况下，将 JMS 放在接收方不会对发送方有任何帮助。但是，如果要使用存储—转发消息传递机制，其中的许多问题都可以解决。这样，异步通信就可以恢复，因为存储—转发系统会负责继续发送消息、重试等。由于这个原因，JMS Bridge 或存储—转发技术就显得至为重要。

需要进行某种形式的企业应用集成（Enterprise Application Integration，EAI）才能实现 RFID 事件的全部价值。仅仅将事件从边缘服务器分派至一系列的应用程序还不能成为完美的解决方案，因为它会产生与安全性、可靠消息传递、性能、可用性、适配器连接、业务流程界定等相关的问题。

比较而言，EAI 解决方案可提供对一个问题的全面概览。例如，一个在达拉斯和旧金山具有不同边缘服务器的组织，可以将事件发送至共同的 EAI 解决方案。涉及连接至不同边缘服务器的读卡器或天线的事件需要组合并关联到一个统一的 EAI 层。而且，复杂的事件组合不适用于这种情况，因为边缘层需要占用 CPU 周期。随着业务流程涉及到组织内部和外部越来越多的系统和人员，EAI 层变得更为关键。

其他一些方面也使得集成解决方案更为必要。要连接至后端应用程序，需要使用基于标准的适配器；在可视化环境下汇编、监控和管理流程的能力也非常重要。通过通用抽象层（如控件），在业务流程、门户、Web 服务、RFID 读卡器和其他元素之间构成复杂交互的能力可以大大提高。最后，在传递事件时，必须在边缘层和实际集成层之间实现无缝集成。

21.1.4 RFID 中间件的特征

一般来说，RFID 中间件具有下列特点：

（1）独立于架构（Insulation Infrastructure）。RFID 中间件独立并介于 RFID 读写器与后端应用程序之间，并且能够与多个 RFID 读写器以及多个后端应用程序连接，以减轻架构与维护的复杂性。

（2）数据流（Data Flow）。RFID 的主要目的在于将实体对象转换为信息环境下的虚拟对象，因此数据处理是 RFID 最重要的功能。RFID 中间件具有数据的搜集、过滤、整合与传递等特性，以便将正确的对象信息传到企业后端的应用系统。

（3）处理流（Process Flow）。RFID 中间件采用程序逻辑及存储再转送（Store-and-Forward）的功能来提供顺序的消息流，具有数据流设计与管理的能力。

（4）标准（Standard）。RFID 为自动数据采样技术与辨识实体对象的应用。EPCglobal 目前正在研究为各种产品的全球惟一识别号码提出通用标准，即 EPC（产品电子编码）。EPC 是在供应链系统中以一串数字来识别一项特定的商品，通过无线射频辨识标签由 RFID 读写器读入后，传送到计算机或是应用系统中的过程，称为对象命名服务（Object Name Service，ONS）。对象命名服务系统会锁定计算机网络中的固定点，抓取有关商品的消息。EPC 存放在 RFID 标签中，被 RFID 读写器读出后，即可提供追踪 EPC 所代表的物品名称及相关信息，并立即识别及分享供应链中的物品数据，有效率地提供信息透明度。

从发展趋势看，RFID 中间件可分为 3 大类：

（1）应用程序中间件（Application Middleware）发展阶段。RFID 初期的发展多以整合、串接 RFID 读写器为目的，本阶段多为 RFID 读写器厂商主动提供简单 API，以供企业将后端系统与 RFID 读写器串接。以整体发展架构来看，此时企业的导入须自行花费许多成本去处理前后端系统连接的问题，通常企业在本阶段会通过 Pilot Project 方式来评估成本效益与导入的关键议题。

（2）架构中间件（Infrastructure Middleware）发展阶段。本阶段是 RFID 中间件成长的关键阶段。由于 RFID 的强大应用，Wal Mart 与美国国防部等关键使用者相继进行 RFID 技术的规划并进行导入的 PilotProject，促使各国际大厂持续关注 RFID 相关市场的发展。本阶段 RFID 中间件的发展不但已经具备基本数据搜集、过滤等功能，同时也满足企业多对多（Devices-to-Applications）的连接需求，并具备平台的管理与维护功能。

（3）解决方案中间件（Solution Middleware）发展阶段。未来在 RFID 标签、读写器与中间件发展成熟过程中，各厂商针对不同领域提出各项创新应用解决方案，例如 Manhattan Associates 提出 RFID in a Box，企业不需再为前端 RFID 硬件与后端应用系统的连接而烦恼，该公司与 Alien TechnologyCorp 在 RFID 硬件端合作，发展 Microsoft .Net 平台为基础的中间件，针对该公司 900 家的已有供应链客户群发展 Supply Chain Execution（SCE）Solution，原本使用 Manhattan Associates SCE Solution 的企业只需通过 RFID in a Box，就可以在原有应用系统上快速利用 RFID 来加强供应链管理的透明度。

21.1.5 RFID 中间件的两个应用方向

根据 ABI Research Inc.的预测，2008 年之前全球各产业的需求所创造出来的 RFID 市场规

模可达到200亿美元，其中软件市场约占47亿美元，2007年RFID的整合服务收入将超越RFID产品收入。随着硬件技术逐渐成熟，庞大的软件市场商机促使国内外信息服务厂商莫不持续注意与提早投入，RFID中间件在各项RFID产业应用中居于神经中枢，特别受到国际大厂的关注，未来在应用上可朝下列方向发展：

（1）Service Oriented Architecture Based RFID中间件。面向服务的架构（SOA）的目标就是建立沟通标准，突破应用程序对应用程序沟通的障碍，实现商业流程自动化，支持商业模式的创新，让IT变得更灵活，从而更快地响应需求。因此，RFID中间件在未来发展上，将会以面向服务的架构为基础的趋势，提供企业更弹性灵活的服务。

（2）Security Infrastructure。RFID应用最让外界质疑的是RFID后端系统所连接的大量厂商数据库可能引发的商业信息安全问题，尤其是消费者的信息隐私权。通过大量RFID读写器的布置，人类的生活与行为将因RFID而容易追踪，Wal Mart、Tesco（英国最大零售商）初期RFID Pilot Project都因为用户隐私权问题而遭受过抵制与抗议。为此，飞利浦半导体等厂商已经开始在批量生产的RFID芯片上加入“屏蔽”功能。RSA Security也发布了能成功干扰RFID信号的技术“RSA Blocker标签”，通过发射无线射频扰乱RFID读写器，让RFID读写器误以为搜集到的是垃圾信息而错失数据，达到保护消费者隐私权的目的。目前Auto-ID Center也正在研究Security机制以配合RFID中间件的工作。相信Security将是RFID未来发展的重点之一，也是成功的关键因素。

21.2　企业应用集成（EAI）中间件

本节介绍企业应用集成（Enterprise Application Integration，EAI）中间件。

21.2.1　电子商务就是EAI

电子商务是一种商业技术，同时也是一种信息技术。然而，电子商务的实施中遇到许多的问题，包括信息流的不通畅、库存管理的问题、物流配送的迟缓，以及财务的不一致等。要解决这些问题，需要打造一个零延迟企业（ZLE，Zero Latency Enterprise）。

零延迟企业是一种理念，或者说是一种方法和技术，这种企业可以即时地更新和访问企业中的数据和信息。企业数据和信息一旦进入企业信息系统，就可以到处使用，而不必重新导入，无需冗余。

零延迟企业可以快速地为客户提供服务，更好地部署企业的资源，给企业带来利润的提高。然而，要实现零延迟企业，就必须克服企业应用分割带来的困难，而企业应用分割的解决方案就是EAI（企业应用集成）。

21.2.2　企业应用分割带来的问题

企业应用分割是由于历史的原因，包括技术的不断进步以及人们对信息系统的理解不全面的后果。企业应用分割是企业架构方面的问题，包括如下方面：

（1）多种孤立的，不兼容的遗留系统。

（2）不兼容的硬件系统和设备。

（3）异构平台，无法彼此通信。

（4）不兼容的、无法移植的开发语言。

（5）不兼容的数据格式。

（6）异构网络。

（7）不同供应商的成套应用程序（如 CRM、ERP 系统等）。

由于企业应用分割，企业拥有多个封闭的系统，多个系统之间具备冗余的数据和重复的功能。各个系统之间信息传输的困难带来企业反应的迟钝、成本的提高和效益的下降。

要解决企业应用分割的问题，就需要求助于 EAI（企业应用集成，Enterprise Application Integration）技术。

21.2.3 EAI 的定义

EAI 并不是一个典型“计算机应用系统”。EAI 是将基于各种不同平台、采用不同方案建立的异构应用集成起来的一种方法和技术。

一般说来，一个企业的“应用系统”由数据库、业务逻辑和用户界面三个层次组成（如图 21-1 所示）。虽然应用系统的设计架构从三位一体发展到今天的多层结构，但它本身的定义没有发生很大的改变。而 EAI 则起着将多个“孤立”的应用系统相互“粘接”的作用，是一个“中间插件”，很像一个“中间人”的角色，如图 21-2 所示。

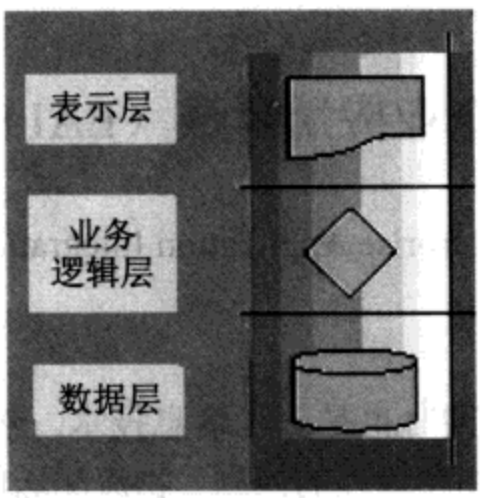


图 21-1 企业应用的三层结构（数据层、业务逻辑层和表示层）

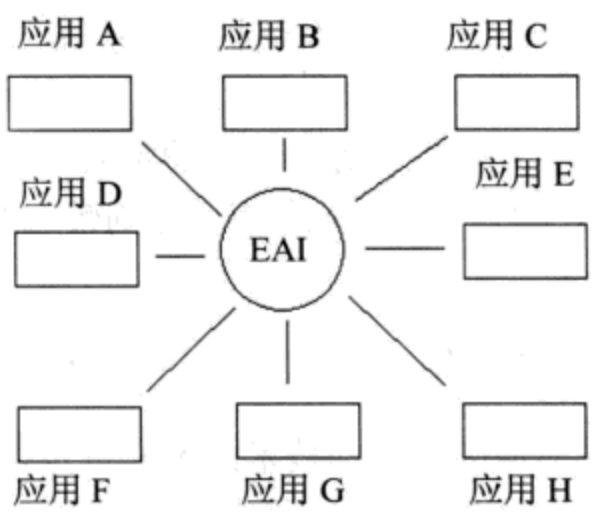


图 21-2 EAI（企业应用集成）整合企业内部的应用

当然，如果一个“中间人”可以协调多于两个人的关系，那么这个“中间人”就必须具

有多方面的协调能力，比如会讲多种“语言”，但他仍然只能是一个“中间人”，是“躲在”企业“应用系统”后面的，企业的最终“用户”并没有觉察到它的真实存在；如果“反客为主”，“从后台走向前台”，并且还带有自己的数据库，那么它就不应该称为 EAI 了，而应该称为一个“经过大量 EAI 整合的新的应用系统”了。

针对上述企业应用的三个层次（或者也可以是更多的层次），企业应用集成（EAI）也可以有不同的实现。

21.2.4　EAI 的目标

EAI 存在两个领域：企业内的 EAI 和企业间的 EAI。实现企业内的 EAI 是实现企业间的 EAI 的前提条件。从结构上来看，企业内 EAI 是一个存在于企业间 EAI 顶层的抽象层。正是通过企业间 EAI，才使给企业带来长远效益的电子商务得以出现。实际上，供应链集成就是一种企业间的 EAI 技术。

EAI 就是在各个应用系统的接口之间共享数据和功能。EAI 的一个原则就是集成多个系统而不干扰每个系统。EAI 的终极目标就是将多个企业和企业内部的多个应用集成到一个虚拟的、统一的应用系统。

因此，实施 EAI 必须遵循如下原则：

- 应用程序的独立性。
- 面向商业流程。
- 独立于技术。
- 与平台无关。

21.2.5　EAI 的类型

EAI 解决方案可以呈现许多种形式并以多种级别出现。EAI 合适的级别依赖于许多因素，包括公司的大小、公司的行业类别、公司应用的集成度或是项目的复杂度以及预算等。

这里列出了 EAI 的中间件解决方案的 4 个类型：用户界面集成、数据集成、商务流程集成和函数/方法集成。

在本节中介绍这些类型的特征和集成的方式。

（1）用户界面集成（界面重组）。界面重组是一个面向用户的整合，将遗留系统的终端窗口和 PC 的图形界面使用一个标准的界面（有代表性的例子是使用浏览器）来替换。一般的，应用程序终端窗口的功能可以一对一地映射到一个基于浏览器的图形用户界面。新的表示层需要与现存的遗留系统的商业逻辑或者一些封装的应用，如 ERP、CRM 以及 SCM 等进行集成。

企业门户应用（Enterprise Portal）也可以被看成是一个复杂的界面重组的解决方案。一个企业门户合并了多个企业应用，同时表现为一个可定制的基于浏览器的界面。在这个类型的 EAI 中，企业门户框架和中间件解决方案是一样的。

（2）数据集成。数据集成发生在企业内的数据库和数据源级别。通过从一个数据源将数据移植到另外一个数据源来完成数据集成。数据集成是现有 EAI 解决方案中最普遍的一个形式。然而，数据集成的一个最大的问题是商业逻辑常常只存在于主系统中，无法在数据库层次去响应商业流程的处理，因此这限制了实时处理的能力。

此外还有一些数据复制和中间件工具来推动在数据源之间的数据传输，一些是以实时方

式工作的，一些是以批处理方式工作的。

数据集成的方法包括：批传输、数据合并、数据复制和析取转换装载解决方案（ETL Solution）。

（3）业务流程集成。虽然数据集成已经证明是 EAI 的一个流行的形式，然而，从安全性、数据完整性、商务流程角度来看，数据集成仍然存在着很多问题。组织内大量的数据是被商业逻辑所访问和维持的。商业逻辑应用并加强了必须的商业规则、商务流程和安全性，而这些对于下层数据都是必需的。

商务流程集成产生于跨越了多个应用的商务流程层。通常通过使用一些高层的中间件来表现商务流程集成的特征。这类中间件产品的代表是消息中介，消息中介使用一个总线模式或者是 HUB 模式来对消息处理标准化并控制信息流。

（4）函数/方法集成。函数和方法集成包括直接的和严格的在网络环境中的跨平台应用程序之间的应用到应用（A2A）的集成。它涵盖了普通的代码（COBOL、C++、Java）撰写、应用程序接口（APIs）、远端过程调用（RPCs）、分布式中间件，如 TP 监控、分布式对象、公共对象访问中介（CORBA）、Java 远端方法调用（RMI）、面向消息的中间件（MOM，Message Oriented Middleware）以及 Web 服务等各种软件技术。

面向函数和方法的集成一般来说是处于同步模式的，即基于客户（请求程序）和服务器（响应程序）之间的请求响应交互机制。

21.2.6 EAI 架构模式

在通常情况下，企业应用集成的困难来自于大量的商业需求和技术的变化。快速的业务变化和发展需要应用系统能更快地发布和使用，而且，业务上的需求需要更多的、高度的应用集成。在加速 EAI 开发的时候，会面对大量的中间件。然而，许多中间件并非像宣传所讲的那么好用。

架构模式（architecture pattern）是 IT 业界人士用来处理应用集成复杂性的一种重要方法。EAI 架构模式提供了对 EAI 在结构上的一种视图，是成功地实施 EAI 的一个基础。这些模式帮助 IT 人士选择合适的方案和工具来扫除应用集成的障碍和困难。

在《设计模式：可重用面向对象的软件设计》一书中，Erich Gamma 等对设计模式给出了如下定义：

设计模式是不断发展和改进的设计方案的一种抽象。它反映了软件设计人员为了提高软件重用性和灵活性而得到的一种设计结构。

EAI 架构模式也是为提高应用系统的重用性和灵活性的一种设计方案。这种做法对于快速变化的商业时代来说是至关重要的。只有这种可变性强的 IT 集成方案才能跟上业务变化的步伐。

EAI 架构的模式主要是针对系统或者应用程序的整体结构。架构模式被定义为软件系统的基础的结构化模式。它提供了一系列定义好的子系统，定义了它们之间的关系和组织这些关系的规则。

EAI 的架构模式和这个定义是一致的。架构模式包括集成适配器（Integration Adapter）、集成消息器（Integration Messenger）、集成面（Integration Facade）及集成媒介器（Integration Mediator），和设计模式是一致的。架构模式和设计模式不同的地方是：架构模式指明的是系

统级的结构属性，并且会对子系统的结构产生影响；而设计模式却不会影响子系统的结构。

EAI 架构模式的共同特点是解藕——将系统之间的相互依赖最小化，这样就提供了更好的灵活性和更强的独立性。

EAI 的架构有 4 种模式：集成适配器模式、集成消息器模式、集成正面模式和集成媒介器模式。

下面将详细讲述这四种模式。

（1）集成适配器模式。对 IT 组织来说，将一个遗留的封闭系统的服务对其他应用系统开放的做法越来越常见。集成适配器描述的是将一个未加以集成的应用系统服务导出到其他应用程序的一种架构模式。一旦这种服务被导出后，这个应用程序的功能就变成可以满足业务需求的可重用的资产。

集成适配器模式提供了一种将可重用的应用程序导出来的灵活方式。这种架构模式和适配器的设计模式有同样的出发点——将一个已有的服务器端应用接口转换成一个客户端程序所期望的接口。这种架构模式的另一个出发点是可以为多个客户端应用程序提供一个统一的可重用的接口。

如图 21-3 所示，集成适配器将一个特定的接口转换为一个开放的、可重用的接口。这种模式的参与者是一个或者多个客户端应用程序和服务器端应用程序。客户端应用程序通过适配器接口调用服务器上的应用程序的服务。这个适配器将导出的公共应用程序接口（API）转换为服务器端的 API。适配器并不需要知道客户端应用程序的存在。对于服务器端应用程序来说，它可能并不知道这个适配器的存在；而另外一种情况，服务器端应用程序会由于适配器的存在而做一定程度的修改。

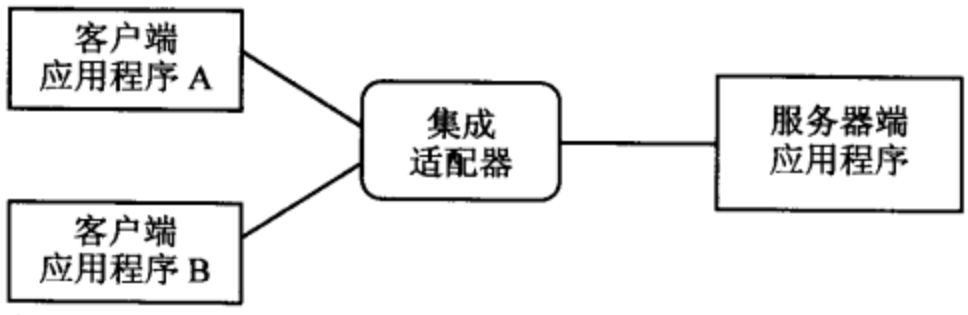


图 21-3　集成适配器模式

（2）集成消息器模式。集成消息器是指在提供集成的同时将应用程序之间的交互逻辑解藕的一种架构模式。这种架构模式带来的益处是使应用程序之间的通信相互依赖性降低到最小。要实现这种目标就需要依赖于这种灵活的集成方式。这种模式支持如下三种通信模型：

- 一对一同步（请求/响应）。这种模型涉及了单个客户端应用程序和单个服务器端应用程序。客户端应用程序等待服务器端应用程序的响应。
- 一对一异步（消息队列）。这种模型涉及了单个客户端应用程序和单个服务器端应用程序。客户端应用程序并不等待服务器端应用程序的响应。
- 一对多异步（发布和预定）。这种模型涉及了一个客户端应用程序和一个或者多个服务器端应用程序。

虽然这种模式的通信模型是多样化的，但其目的是一样的，使得应用程序之间的通信具备尽可能小的相互依赖性，如图 21-4 所示。

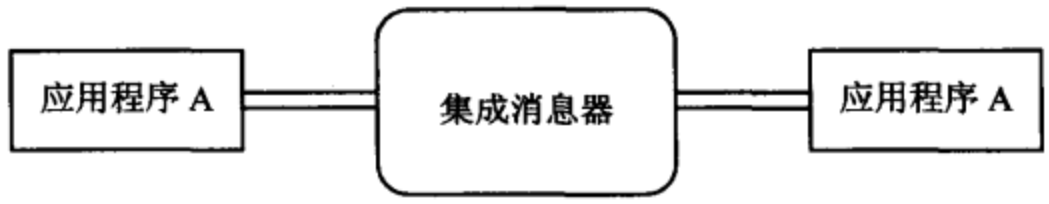


图 21-4 集成消息器模式

这种模式的参与者是被集成的应用程序和集成消息器。集成消息器负责在应用程序间发布消息，并且提供透明的消息定位服务。

（3）集成正面模式。Facade 指的是建筑物的正面。而在设计模式中，Facade 指的是通过统一的、简化的接口来隐藏接口背后的设施。

集成正面模式描述的是将客户端应用程序和服务器端应用程序集成起来的一种集成方案。这种架构模式和 Facade 设计模式的出发点是一样的。不过，这种架构模式提供了级别更高的、更加简化的接口供客户端应用程序使用，以使客户端应用程序和服务器端应用程序的依赖性、相关性降至最小，这样就获得了应用程序的灵活性和重用性。集成正面的模式如图 21-5 所示。

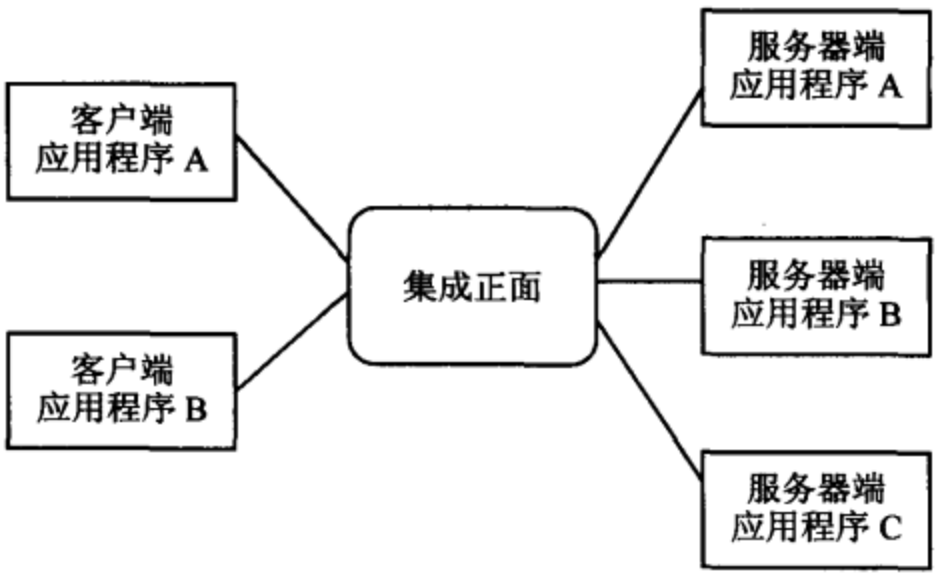


图 21-5 集成正面模式

这种架构模式可以为一个或者多个客户端应用程序提供统一的、简化的接口。这种集成正面模式的参与者是一个或者多个客户端应用程序、一个或者多个服务器端应用程序以及集成正面。客户端应用程序可以调用集成正面的服务。这种模式抽象了服务器端应用程序的功能，使其更易于使用。集成正面将其接口转换成为服务器端应用程序的接口。实际上，应用程序执行了具体的工作，而集成正面的作用是将其自身的接口转换为服务器端应用程序的接口。在这种架构模式中，集成正面不知道客户端应用程序的存在，服务器端应用程序也无须知道集成正面的存在。

（4）集成媒介器模式。集成媒介器模式指的是将应用程序的交互逻辑封装起来然后从应用程序中剥离出来加以集成。这种方案的优点在于：

- 将应用程序之间的依赖性以及对现有的应用程序的影响最小化。
- 由于应用程序的交互逻辑不是分布在应用程序中，而是集中起来，这样维护的工作量将达到最小。

集成媒介器模式如图 21-6 所示。这种模式包括一个集成媒介器以及两个或者多个应用程

序。集成媒介器包含的是系统的交互逻辑；参与的应用程序直接和集成媒介器交互，而不是和其他的应用程序交互。

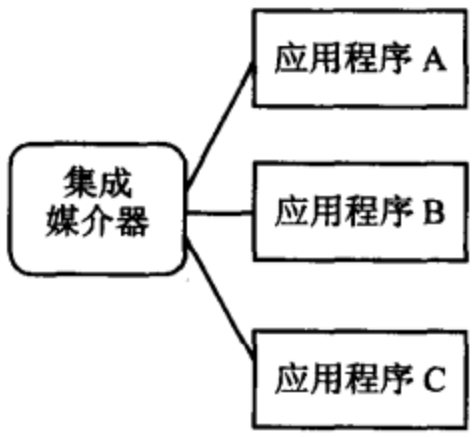


图 21-6　集成媒介器模式

由于交互逻辑被集中到了集成媒介器，这种模式使系统获得了更好的灵活性，提高了业务的敏捷性。

通过以上论述，可以发现使用架构模式，可以提高系统的重用性和灵活性，降低 EAI 集成的复杂性和 EAI 实施可能存在的风险。

21.2.7　EJB、应用程序服务器与应用程序集成（EAI）

应用服务器是一种事务性中间件，它提供了一种集中化的服务器架构，能连接企业的各种资源，共享商业方法和商业逻辑，在实现企业分布式计算以及方法级别的企业应用集成方面具备明显的优势。

事务性中间件，如 TP 监视器和应用程序服务器（Application Server，以下简称应用服务器），为实施 EAI 和企业的分布式计算带来很大的便利。可扩展性、容错能力以及将企业应用程序流程集中化的架构是事务性中间件的重要特征。

应用服务器对 EAI 来说是一种很好的工具，因为这种架构提供了一个集中化的服务器，可以处理不同来源的数据和信息，如数据库和应用程序。另外，通过应用服务器可以将信息从一个应用程序发送到另一个应用程序，并且支持分布式计算。不过，使用应用服务器可能要对现有的应用程序做一定的修改。

1. 应用服务器与事务

应用服务器将复杂的应用分解为多个事务。从事务开始到结束，从客户端到服务器端，应用服务器始终控制着事务。在这一点上，事务可能是全部，也可能什么也不是。一个事务不可能是未完成的，因此应用服务器使系统处于一个稳定的状态。事务具有原子性、独立性、一致性和持久性。事务的这些特性为开发人员提供了一种一致、可靠的编程模式，这使得应用服务器成为实现分布式计算的最好选择，因为分布式计算必须处理不同的数据库、队列以及运行在异构平台之上的批处理接口。

应用服务器可以以客户端或者节点的方式来处理事务并且可以在不同系统中对事务进行路由，来实现 EAI 问题论域的需求。应用服务器同时提供负载均衡、线程池、对象的回收以及自动恢复等功能。

（1）数据库连接共享。使用应用服务器的一个益处是通过管理事务可以减小数据库连接

的数量，减小大系统对数据库带来的负载。有了应用服务器，在增加客户端数量的同时，不用增加数据库服务器的大小。

通过“过滤”客户端请求，应用服务器不用对每一个客户端进行单独处理。通过数据库连接的“池化”，一个客户端请求只须简单地触发 TP 监视器上的事务服务，这些请求可以共享同一个数据库连接。当一个数据库连接负载过重时，只需启动一个新的数据库连接。这是三层结构的基础，也是三层结构可以承受高用户负载的原因。

（2）负载均衡。当客户请求的数量超过了系统所能处理的共享进程的数量时，其他的进程就自动启动，这就是负载均衡的概念。一些应用服务器可以动态地将进程负载分布到多个服务器上，或者在多处理器环境下将进程分配给多个处理器。

应用服务器的负载均衡机制也可以为进程设置优先级。这样开发人员可以为重要客户提供更优越的服务。另外，开发人员也可以根据应用程序类型、响应时间、事务的容错等设置进程优先级，同时也能通过设置参数控制每个事务可获得的进程的数量。

（3）容错。应用服务器提供了一个强壮的应用程序部署环境，使应用可以从许多故障中得到恢复。应用服务器通过使用冗余系统来获得高可用性。这种能力使得应用服务器在 EAI 问题论域获得不少优势，因为应用服务器能确保发出的信息在发送方和接收方应用程序之间被共享。

（4）通信。应用服务器可以通过多种方式通信，如 RPC、DPL、MOM 等。应用服务器也可以和多种中间件通信，如消息代理等，能更好地满足 EAI 问题论域的需求。

2. 管理事务性组件

应用服务器和传统的事务性组件（如 TP 监视器）的一个不同点就是应用服务器能够处理事务性组件。大部分使用事务性组件架构的应用服务器都采用 EJB 作为标准。

应用服务器对事务性组件提供事务的支持，同样也可以从客户端程序来控制事务性组件的事务。应用服务器可以构建 EJB 的集群，提供负载均衡的支持，实现良好的可扩展性。通过 EJB 可以访问来自数据库、遗留系统以及其他资源的信息和商业方法，提供了一种新的共享机制，以及一种新的集成机制，如图 21-7 所示。

应用服务器的优势在于方法级别的应用集成，或者说是 EAI 公共业务逻辑的共享。对 EAI 的架构设计师或开发人员来说，消息代理或者面向消息的中间件，对于数据级别或者应用程序接口级别的信息共享是一个好的选择。这并不是说应用服务器不适合 EAI,而是说应用服务器在方法级别的应用集成方面有更多的优势。从 EAI 的观点来看，架构设计师和开发人员必须明白应用服务器的优势和不足。

应用服务器提供了应用程序代码（也就是业务处理和应用程序对象）运行的场所，同时也提供了一个可以在应用程序之间共享方法的场所。应用服务器提供了一个部署和管理事务性组件的环境，同时还可以连接多种资源和数据，具备集成后台应用的能力。

3. 应用服务器与 EAI

应用服务器的将来会是什么样？又将如何应用于 EAI 领域呢？

简单地说，应用服务器将继续是 Web 应用开发和方法/过程级后台集成的平台。

不过，应用服务器还必须继续发展，包括可扩展性、适应更大的负载、更好地管理状态的能力以及更好地管理事务。

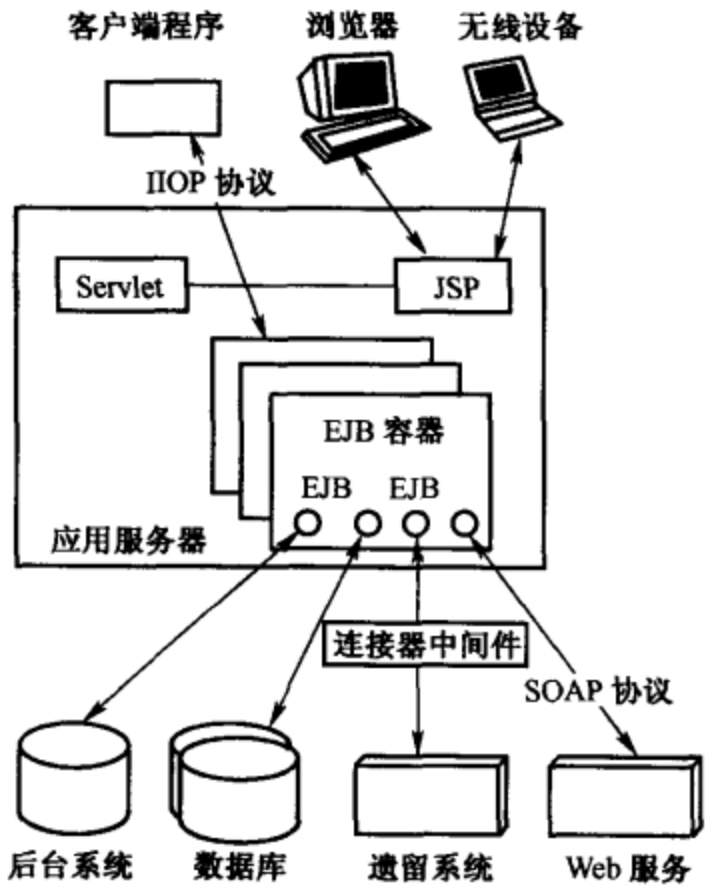


图 21-7　EJB 与企业应用集成

EJB 是一个很好的方向，但是 EJB 规范在许多方面（如线程池、并发控制、资源管理等方面）还没有详细的定义。这样不同的供应商就可能推出不太一致的产品，可能会使用不同的机制，会有不同的发展方向。Sun 公司意识到了这一点，因此建立了一套程序来确保这些产品是基于 EJB 规范的；反过来又加速了 EJB 的发展，这样可以使用不同的工具来开发 EJB,发布到遵循 EJB 规范的不同容器。

应用服务器通过共享商业方法提供了一种比较紧密的集成方式。目前，应用服务器逐渐适应松散耦合的集成方式，这要归功于新出现的一些技术，如 XML 技术可以统一信息的格式，便于通过应用服务器实现应用程序之间的信息共享和传递；基于 SOAP 协议的 Web 服务技术，已经成为一种可以通过应用服务器访问 Internet 的商业方法。另外，Java 消息服务（JMS）提供了一种新的技术来将新的应用和旧的后台系统进行集成。JMS 是用于和面向消息的中间件相互通信的应用程序接口（API）。它既支持点对点的域，又支持发布/订阅（publish/subscribe）类型的域，并且提供对下列类型的支持：经认可的消息传递、事务型消息的传递、一致性消息和具有持久性的订阅者。

应用服务器将在 EAI 领域发挥越来越明显的作用。EAI 问题论域需要在应用程序之间支持事务管理，共享业务逻辑以及数据，并且允许所连接的应用程序所做的更改。使用应用服务器的集成解决方案对此来说是比较合适的。而其他的应用（如 CRM、SCM、ERP 等）也越来越依赖于应用服务器的这种架构，基于应用服务器的综合方案和补充性的解决方案也越来越多。

21.2.8　EAI 中间件的定义

企业应用集成中间件通常指的是企业应用集成服务器。企业应用集成服务器不是一个新

的概念，实际上，作为“消息中介”或“EDI”支撑软件，EAI 服务器在国内早就有应用。

消息中间件、事务处理中间件、Web 应用服务器、门户服务器等中间件产品都有支持集成企业信息系统（EIS）的能力，并在实际的企业信息系统集成中实实在在地发挥着作用。与这些产品不同，EAI 服务器强调的是应用系统之间相互访问与集成的需求和能力，访问是双向的。集成的中心是作为枢纽的 EAI 服务器，枢纽向外的辐射（通过消息中间件等技术）把各个应用系统连接集成在一起。枢纽执行应用之间格式的转换、消息传递的路由选择与控制及传输任务。

EAI 服务器通常是一个有着重量级“价格”的软件系统。软件通常运行在消息中间件之上，可以和Web 应用服务器等中间件集成。

21.3 数字电视中间件

21.3.1 数字电视中间件的概念

数字电视中间件是指位于数字电视机顶盒内部实时操作系统与应用程序之间的软件部分，它以应用程序接口 API 的形式存在，整个 API 集合被存储在机顶盒的闪存 FLASH 中。针对机顶盒的应用程序基于 API 进行开发，能够支持丰富的应用。采用中间件系统，可以跨越技术、标准等复杂的内容，用简单的方法定制具有自己特色的应用软件，从而在提高开发效率、减少开发成本的同时能够跟上技术的发展，将应用的开发变得更加简捷，使产品的开放性和可移植性更强。虽然中间件对机顶盒硬件资源的要求较高，但在开发大量应用时成本大大下降。

中间件技术是一个纯软件系统技术，它也是一个比较笼统的概念。一般地说，它是建立在数字电视接收设备驱动层之上，为交互应用提供一个完整的应用编程接口的软件系统。它包含一系列的功能，如内存管理、通信管理、图形系统管理、SI 数据装载、系统资源管理以及与前端系统间的通信及控制等。图 21-8 描述了中间件在数字电视机顶盒中的位置。

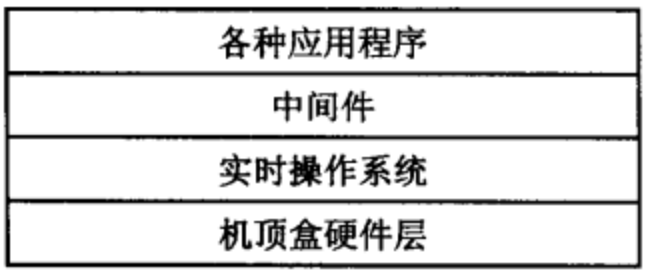


图 21-8 中间件在数字电视机顶盒中的位置

21.3.2 数字电视中间件的技术标准

不同的中间件系统会提供不同的与下层驱动资源模块的接口以及与上层应用编程的接口，因此不同的系统之间是不能互通的。因而人们在数字电视业务的不断拓展过程中，认识到了制定统一的应用程序接口的重要性。可以说，中间件技术的发展与中间件标准的制定进程是同步的。数字电视领域的两大标准化组织——欧洲的 DVB 和美国的 ATSC 在制订数字电视广播标准、数据广播和交互业务标准之后，开始制定中间件的标准。DVB 已经提出了基于 Java 虚拟机的中间件标准——多媒体家庭平台(MHP)，我国的数字电视中间件标准主要参考 MHP，

目前正在制定之中。ATSC 成立了 T3/S17 技术专家小组委员会，致力于机顶盒软件环境的定义，称之为数字电视应用软件环境（DASE）。下面就对 MHP 和 DASE 做简要介绍。

（1）欧洲的 MHP。由于 DVB 采取市场驱动的框架，因此初期的工作重点是广播基础设施，在大量的数字广播传输标准取得成功后，DVB 的工作重点转移到交互业务这一层次，代表成果是数据广播标准和交互业务标准。由于看到广播、计算机和消费电子在家庭中的聚合，DVB 又迈出了第三步，即建立多媒体家庭平台（Multimedia Home Platform，MHP）标准，按照 MHP 组主席 Georg Luetteke 博士的说法，MHP 所带来的数字电视和相关多媒体业务的融合是 WWW 出现以来最大的技术奇迹，也是现在能够想象到的广播技术的终极。

MHP 有关的工作开始于 1997 年初，工作模式遵循 DVB 的工作原则，即先研究市场需求，再进行技术标准开发。MHP 作为 DVB 商业部（CM）的一个工作组，由 DVB 指导委员会（SB）于 1997 年 12 月正式批准成立，其任务是调查多媒体家庭平台的商业需求。其后，在 DVB 技术部成立了 MHP 技术面工作组 TAM（Technical Aspects of MHP），其任务是编写 MHP 技术规范。与此同时，MHP 组仍在进行 MHP 商业需求的细化和辨别工作，两个组相互协作，联系密切，共同完成多媒体家庭平台这项牵涉面广的复杂工作。

MHP API 的最早竞争者有三个：来自 Sun Microsystems 和 Thomson Multimedia 的 OpenTV 提案，来自数字视听工业论坛 DAVIC（Digital Audio Video Industries Council）的 MHEG 5/MHEG 6 + Java 提案和来自法国地面数字电视广播商 Canal Plus 的 Multimedia Highway 提案。不久， Multimedia Highway 就提出愿意采取一定策略来容纳 MHEG 5 和 Java。

从 1998 年到现在，DVB 内部关于 MHP 的辩论十分激烈，原因是 DVB 各成员的想法各不相同，而这样的一个标准涉及到欧洲的情况不同的多个国家。争论的核心主要集中在六个方面：开放标准还是封闭标准；增强电视还是交互电视；Internet 的中心地位问题；欧洲和美国的不同；继承还是创新；知识产权问题。其中对增强电视的情有独钟、难以达成一致意见和知识产权问题是阻碍标准制定的三个主要障碍。

尽管争论激烈，甚至有人怀疑 MHP 这样的统一标准是否必要，即使制定了能否实现，但 DVB 的 MHP 项目还是取得了不少成果，目前达成一致的是定义一套兼容 Java 的应用编程接口（API）。此 API 以三个类别（profile）的形式定义，或者说三个层次，其中每个上层类别都包含下层类别。第一个类别是增强电视类别（Enhanced TV profile），它是基于 Java 的，而且其中不包括 HTML 兼容。第二个类别称为带反向通道的交互电视，定义为两种可选格式：一种是“纯 Java”方案，源自 Sun 的 PersonalJava 规范和/或 JavaTV 规范，这种方案定义为缺省类别，另一种可选的格式是在默认类别的基础上增添兼容 HTML。第三个类别称为 Internet 接入，在交互电视类别的基础上增加 IP 和兼容 HTML。

API 的目的是通过互操作性提供统一的应用平台，但上述基于 Java 的 API 方案并非如此，尤其是在兼容 HTML 方面模棱两可的规定使得互操作性大打折扣。最热心兼容 HTML 的是有线电视运营商，他们关心快速 Internet 接入能够成为自己的一项关键业务，而其竞争对手（卫星运营商和数字地面电视广播公司）对兼容 HTML 并不热心，因此 MHP 交互电视类别的两种选项实际上在有线电视和卫星与地面广播之间埋下了一道潜在裂痕。

欧洲的统一 API 分三步走，不是技术问题，而是考虑到欧盟成员国在政策方面的差异。政策差异是其社会文化环境差异的反映，特别在电视广播领域这种差异更加明显，各成员国联系到现有公共广播制度，在国家层面上会采取不同行动。因此，现行的 MHP 标准是一个结构

冗杂的大系统，它在具有先进性和可扩展性的同时，各个程序包之间并不和谐，存在功能上的叠加和重复。

（2）美国的 DASE。在 ATSC 1983 年成立伊始，双向增强电视就是构想之一，但直到 1996 年 FCC 大部分采纳 ATSC 标准时，双向交互电视问题都一直悬而未决，没有人能说清楚呈现在观众面前的交互电视到底是什么样。在此期间，时代华纳曾开展交互电视实验，但没取得成功；相反，交互式的 Internet 异军突起，取得了巨大成功。

消费电子制造商（包括数字电视芯片制造商）和内容生产商都迫切需要制定统一的接收机软件平台，这是交互电视取得成功的关键。因此，1997 年末，ATSC 成立了数字电视应用软件环境（DASE）特别工作组，它源于 ATSC T3/S16（交互业务）专家组内的一个小组，后来独立成一个特别工作组，并很快成为 T3 的直属专家组，编号为 T3/S17。S17 之下又细分为多个小组：系统业务组、总体组、类别（profile）组、安全组和呈现引擎组，随着工作的深入，小组的划分也有所变化。

作为应用软件环境，DASE 包括四个组成部分：应用执行引擎（Application Execution Engine，AEE）、呈现引擎（Presentation Engine，PE）、内容解码器（Content Decoder）和应用程序接口（API）。这四个部分构成一个有机整体，这一体系结构是随着 DASE 的工作进展而提出并不断完善的。

应用执行引擎 AEE 直接在接收机的操作系统和库之上实现，以平台无关的方式解释或执行应用代码中的程序部分，为内容解码器和呈现引擎提供插件平台，作为底层软硬件平台的抽象，扮演 DASE 环境集成者的角色，同时负责管理多个应用可能竞争的资源。

呈现引擎 PE 的任务是：①执行描述屏幕如何显示的（可能是说明性的）代码；②提供屏幕的空间布局；③提供屏幕上小部件（widget）或对象的时间同步；④允许小部件或对象在屏幕上的合成（透明和半透明等）；⑤在屏幕上对已知类属的小部件或对象进行实例化。PE 必须对用户产生的事件及其手势或动作作出反应。PE 可以是平台有关的，即使用接收机操作系统的私有库，如果制造商允许，可以下载平台无关的 PE（只依赖于 AEE 和 API），通过注册可以代替制造商的 PE。在 DASE 接收机中，将只有一个 PE。推荐的 DASE PE 为 XHTML。

面向系统服务的应用程序接口 API 是 DASE 为应用程序提供的接口标准，应用通过它访问下层操作系统和接收机硬件提供的系统服务，是操作系统自有库之上的抽象层，是 DASE 的核心。

21.3.3 数字电视中间件 Java 技术平台的种类

数字电视中间件的 Java 平台采用 Personal Java（个人应用型程序），目前全球数字电视的中间件基本都采用这种规格。随着应用领域的不断细分和明确，Personal Java 正在进一步模块化，以适应不同嵌入式设备的资源能力要求。

因此，Java 联盟提出新的嵌入式 Java 规格，称为 J2ME（Java 2 Micro Edition，新型个体应用版本），将以前 Personal Java 的功能进行模块化组合，使嵌入式 Java 平台具有可剥离的能力。

目前适用于个人电脑和更大型应用的有 Enterprise Edition（企业型）和 Standard Edition（标准型）两个版本，采用的标准为 JVM（Java 虚拟机）。嵌入式应用则划分更细，形成不同的配置，有 CDC（Connected Device Configuration，连接设备配置）和 CLDC（Connected Limited Device Configuration，连接有限设备配置）两种，分别采用 CVM（Compact Virtual Machine，紧凑式

虚拟机）和 KVM（Kit Virtual Machine，微小虚拟机）两种虚拟机。

J2ME 经过这样的模块化配置，成为 Personal Java 的替代形式，今后嵌入式 Java 平台将逐步过渡到 J2ME 平台。由于历史原因，数字电视中间件在当初只在 Personal Java 选择，由于 J2ME 完全包含 Personal Java，所以 J2ME 取代 Personal Java 在中间件的地位成为必然。

（1）Personal Java 平台。Personal Java 作为构建可连接网络的消费类电子设备而设计的 JAE（Java 应用环境），是 Sun 公司推出的一系列 Java 产品中专门面向消费类电子设备的一个平台。它由 JVM 和一系列 Java API（Application Programming Interface，应用编程接口）组成，其中 Java API 包括核心、选项和类库。另外，Personal Java 在资源有限的环境中包括消费类电子产品中应用需要的一些特定的特征。

（2）J2ME 平台。J2ME 是 Java 2 这个平台专门针对家电类产品和嵌入式设备制定的。组成 J2ME 的 Java 虚拟机和 API 是与那些针对家电类产品和嵌入式产品的运行环境相匹配的。

J2ME 针对小型设备设计，具有丰富的用户接口和网络交互模型，编程和可移植性好。

J2ME 各应用平台由 Configuration（配置）和 Profile（档次）来组成，配置是为 JVM 定义最小的能力和库，使 JVM 足够小并且能够运行在相同级别的设备上。由于同一级别设备的配置对内存和处理能力的要求相近似，所以 J2ME 将所有的嵌入式应用设备大体上区分为两种：一种是运算功能有限、电源供应也有限的嵌入式设备，如 PDA（Personal Digital Assistant，个人数字助理）和手机等，并把它们定义在 CLDC 规格之中；另一种是运算能力相对较佳、并在电源供应上相对比较充足的嵌入式装置，并把它们规范为 CDC 规格。

档次基于 Java 技术的 API 集合，由不同功能的 Java 类库组成。档次构建在配置之上，作为配置的补充，为特定的设备提供足够的运行环境。档次是为了要更明确地区分出各种嵌入式设备上 Java 程序该如何开发，具有哪些功能。因此档次之中定义了与特定嵌入式设备非常相关的扩充类，而 Java 程序在各种嵌入式装置的应用接口该如何表达则通过档次的定义来实现。各档次中所定义的扩充类库是根据底层配置中所定义的核心类库建立的。

（3）J2ME CDC 平台。CVM 是构成 CDC 的基础，它在传统 Java 虚拟机基础上对 Byte-Code 解释器进行了改进，具有动态编译器功能，使运行 Java 程序更为有效和快速。CVM 的运行环境需要大于 2M 字节的程序存储器和大于 512 字节的运行存储器。

（4）J2ME CLDC 平台。CLDC 适用于一些硬件资源有限的接收机。在 J2ME 规范中，CLDC 主要是针对运算功能有限、电力供应也有限的嵌入式装置（如 PDA、手机）定义的。它所需要的运行资源在 CPU 处理能力、内存、网络带宽和电源等方面的资源需求要低于 CDC 对资源的需求。

21.4　小结

本章介绍了中间件技术的最新进展，包括如下内容：

- RFID 中间件的定义、功能和特征。
- 企业应用集成（EAI）中间件的定义、EAI 的类型和 EAI 架构模式。
- 数字电视中间件的概念、技术标准，以及数字电视中间件 Java 技术平台的分类。