

IBM WebSphere Application Server **实用大全**

[美] Ron Ben-Natan
Ori Sasson **著**
熊志辉 **译**

北京·清华大学出版社

(京) 新登字 158 号

内 容 简 介

WebSphere 是电子商务领域最重要的软件厂商 IBM 研发的因特网基础架构软件。WebSphere 使企业能够开发、部署和集成新一代电子商务应用 (如 B2B), 并且支持从简单的 Web 发布到企业级事务处理的电子商务应用。

本书系统、全面地阐明了从开发、装配、部署到管理 WebSphere 电子商务应用程序的各个方面。除了 Web 服务、多语言应用程序与服务器方组件、Web 核心技术 (如 EJB、servlet 与 JSP) 之外, 本书还覆盖了包括 JMS、JSP 标记库、EJB 扩展、Web 服务以及 J2EE 兼容性在内的 IBM WebSphere Application Server 4.0 的新功能。

本书特别适合需要构建企业电子商务应用程序的读者, 本书已成为国际上许多成功的 WebSphere 电子商务应用实例的“圣经”式教材, 许多大型电子商务应用程序开发组都以本书作为设计 WebSphere 电子商务应用程序的指南。此外, 本书也适合那些希望了解并应用 WebSphere 构建服务网站的网站管理人员以及大学的高年级学生。

IBM® WebSphere® Application Server: The Complete Reference

Copyright © 2002 by The McGraw-Hill Companies.

Original English language edition published by McGraw-Hill Education.

All rights reserved.

本书中文简体字版由美国麦格劳-希尔教育出版集团授权清华大学出版社在中国境内(香港、澳门特别行政区和台湾地区除外)出版、发行。未经出版者书面许可, 不得以任何方式复制或抄袭本书的任何部分。

本书封面贴有 McGraw-Hill 防伪标签, 无标签者不得销售。

北京市版权局著作权合同登记号: 图字: 01-2003-1102

版权所有, 盗版必究。

书 名: IBM WebSphere Application Server 实用大全

作 者: Ron Ben-Natan, Ori Sasson

译 者: 熊志辉

出版者: 清华大学出版社 (北京清华大学学研大厦, 邮编 100084)

<http://www.tup.com.cn>

印刷者: 北京市耀华印刷有限公司

发行者: 新华书店总店北京发行所

开 本: 异 16 印张: 51 字数: 1146 千字

版 次: 2003 年 3 月第 1 版 2003 年 3 月第 1 次印刷

印 数: 0001~5000

盘 号: ISBN 7-89494-034-8

定 价: 86.00 元 (1CD)

译者序

Internet 技术以及电子商务应用的蓬勃发展，为许多企业提供了大量新的机会。上世纪 70 年代以来，很多企业都基于电子数据交换（EDI，Electronic Data Interchange）实现业务处理自动化。但 EDI 侧重于商业伙伴之间事务处理的标准化，并且 EDI 标准缺乏灵活性和可伸缩性，使得许多企业希望电子商务应用系统具备如下特征：

- 。 基于标准；
- 。 以服务器为中心；
- 。 可伸缩；
- 。 能利用已有的核心系统；
- 。 可快速部署和易用；
- 。 易管理。

为此，需要一个完整、开放和标准的电子商务基础架构。与此相应，IBM 将使 WebSphere 及其他基础架构软件全面支持开放互联网标准，促进动态电子商务应用的开发。正如 IBM 公司软件部 WebSphere 市场副总裁 Jocelyne Attal 所说：“WebSphere 软件是电子商务基础架构的核心，是通往动态电子商务的极速之路”。

IBM WebSphere 是 Internet 基础架构软件，为企业电子商务提供全面的基础架构。使用 WebSphere，企业可以开发与配置下一代电子商务应用（如 B2B 电子平台），并且支持电子商务应用从简单的 Web 发布到企业级的事务处理，可以说，WebSphere 是构筑新一代电子商务平台的利器。

WebSphere 的基础层包括 WebSphere 应用服务器和 MQSeries，WebSphere 应用服务器包括标准版、高级版及企业版，由单一的 Java 引擎驱动，所以不同版本之间的应用可以相互移植。WebSphere 具有如下特点：

- 。 可扩展性好，操作配置简单；
- 。 WebSphere 支持广泛的业界开放标准，包括 Java/J2EE，XML，LDAP，CORBA，WML 等；
- 。 性能价格比高。

正如其书名指出的那样，本书是关于 IBM WebSphere 应用程序服务器的实用大全。全

书几乎囊括了从开发、装配、部署到管理 WebSphere 电子商务应用程序的每个方面。为了将本书推荐给国内的读者，特别是为了促进国内企业在加入 WTO 之后能够迅速融入参与到电子商务领域的竞争中，清华大学出版社几经曲折向该书原著出版公司购买版权并迅速组织人员进行翻译出版。

本书由胡季红组织翻译，熊志辉译。虽然在本书的翻译过程中，我们查阅了大量的文献资料，力图杜绝翻译中的错误与纰漏。但水平所限，错漏之处在所难免，请广大读者批评指正，谢谢！

第 1 部分 入 门

第 1 章 初识 WebSphere

IBM 被普遍认为是最重要的软件公司之一，在电子商务领域更是如此。实际上，许多人都认为 IBM 是最具影响力的公司，并且在诸如 Web 应用程序开发、Java、中间件以及企业系统等领域都提供有最佳的软件产品。本章将简要介绍 IBM 的电子商务产品。WebSphere 是 IBM 电子商务产品的基石，同时，WebSphere 既是建立 IBM 的电子商务产品蓝图的应用程序服务器平台，又是 IBM 用来创建应用程序的全套产品的产品名统称。本章将讨论这个应用程序服务器，也将讨论属于这个商务产品家族的一些产品，例如 WebSphere Portal Server，WebSphere Studio 以及 WebSphere Voice Server。

WebSphere 就是被称为中间件（middleware）的 Internet 底层软件。它使得企业能够开发、部署与集成下一代电子商务应用程序（例如那些用来进行 business-to-business 电子交易的应用程序）；它还能够支持从简单 Web 发布到企业级事务应用程序在内的各种商务应用程序。WebSphere 改变了企业管理客户、合作者以及雇员关系的方式。例如，能够使用 WebSphere 创建一个非常吸引人的 Web 应用程序，由此提高站点流量的质量与数量；能够扩展应用程序以便集成移动设备（从而使得销售人员能够更快地为客户服务）；或者，能够创建可降低寻求货源所需开销的电子市场。

1.1 WebSphere：是产品家族还是应用程序服务器

WebSphere 是 IBM 的 Internet 策略的基石。从本质上来说，WebSphere 是一个产品线，它由一系列用于开发、部署以及维护 Internet Web 站点与电子商务系统的工具组成。这个产品线的核心部分是 WebSphere Application Server。WebSphere Application Server 是一个应用程序部署环境，用来部署作为 Java 服务器方代码实现的应用程序。WebSphere Application Server 有 3 个不同的版本，三者 in 复杂性 with 覆盖范围方面有所不同。

- 高级单服务器版本（Advanced Single Server Edition）：这个版本提供了高级版本中的大多数 J2EE（Java 2 Platform Enterprise Edition，Java 2 平台企业版）功能，但这

一版本仅适用于包含有一台服务器的环境。这一版本在许多方面都存在限制，部分原因是因为此版本的部署与性能选项，另有一部分原因是此版本的管理特性。

- **完全配置的高级版本 (Advanced Edition, Full Configuration) :** 这是 IBM 提供的标志性应用程序服务器。如果需要在产品环境下运行应用程序，那么就可能要使用这个版本。这个版本包含有对 servlet, JSP (JavaServer Page, JavaServer 页面), XML (Extensible Markup Language, 可扩展标记语言), EJB (Enterprise JavaBean, 企业 JavaBean), 消息传递等功能的支持。
- **开发者许可的高级版本 (Advanced Edition, Developer License) :** 这一版本提供了较低价格的服务器，在功能上与单服务器版本相同，但高级版本不能用于产品运行时环境。高级版本可以作为评估版从 IBM 的网站上下载（但很奇怪的是，完全高级版本 (Full Advanced Edition) 不能下载）。

WebSphere Application Server 于 1998 年秋进入市场。然后陆续发布了几个新的版本；在编写本书时，WebSphere Application Server 的最新版本是 4.0。本书的内容就是与 4.0 版相对应的。4.0 版的一个重要特色就是它是第一个遵从 J2EE 规范的版本。由于是基于 J2EE 设计的，因此 4.0 版能够确保与其他厂商的产品之间的兼容性和互操作性，并且符合大多数开发者所应用的方法学。

作为一个产品系列，WebSphere 是 IBM 电子商务策略的核心元素。实际上，术语“IBM WebSphere 家族”已经被 IBM 拥护者当作电子商务的代名词，并且，IBM WebSphere 家族并不仅仅包括 WebSphere 服务器本身——实际上还包括开发工具、监视组件、配置管理实用程序等产品。这个框架非常广泛，甚至根据 IBM 提供的市场营销文档都难以理解哪些产品真的是 WebSphere 的组成部分，而哪些又是仅仅以“WebSphere”的名义为了使得产品集更完全而提供的。本书将主要关注 WebSphere 作为应用程序服务器的讨论，但笔者也将提供与 WebSphere 家族密切相关的产品信息——用户可以用这个产品来创建与发布真正意义上的电子商务应用程序。其中两个例子就是 WebSphere Studio Advanced Developer 和 VisualAge for Java。这两个产品都是 Web 应用程序的开发环境，并且两者都与 WebSphere 服务器有连接器与集成点。

本章的其余部分将概述 WebSphere 的产品系列。将从非技术性的角度，描述 IBM 以 WebSphere 的名义提供的各种产品。虽然这些产品与本书讨论的主题没有直接关系，但对任何 WebSphere 开发人员来说，至少应该对这些产品的名字比较熟悉。此外，如此众多的以 WebSphere 的名义提供的产品，也反映了 IBM 在创建 WebSphere 这一品牌时所表现出的严肃性。

1.2 WebSphere Application Server

本节关注于 WebSphere Application Server，在此将详细说明各种版本之间的不同之处。所有的版本都提供了一个应用程序服务器所具有的功能。这一术语并没有一个非常精确的定义，但在此上下文中，指的是能够扩展 Web 服务器功能的服务器。传统的 Web 服务器是能够响应 HTTP 协议请求的服务器。应用程序服务器则提供了执行 Java 服务器方代码的能力，从而允许生成动态内容并且实现复杂的企业应用程序。

1.2.1 WebSphere 高级单服务器版本

在 4.0 版中新增加的 WebSphere 的单服务器版本是入门级产品。在 4.0 版之前，WebSphere 标准版本（WebSphere Standard Edition）充当这一功能，其主要目标是开发与部署相对简单的应用程序。虽然标准版本是一个入门级产品，但它确实是一个完整的 WebSphere 服务器，具有所有的核心服务并支持实用程序。标准版本的主要特色是支持 Java servlet 以及 JSP 的 Java 执行引擎。Java servlet 与 JSP 就是能够产生 HTML 或其他内容的 Java 代码段（这些概念将在本书后面进行详细描述）。

在 4.0 版中，WebSphere Application Server 的标准版本没有了，而在 3.5 以及更早的版本中提供了 WebSphere Application Server 的标准版本，IBM 提供了从 3.5 版迁移到 4.0 版的简易途径。停止提供标准版本的主要原因是，由于缺乏 EJB 支持，使得这一版本不能与 J2EE（应用程序服务器的一种 Java 标准）兼容。停止提供标准版本的另一个可能原因就是，它损害了 IBM 从 WebSphere 大量获利的目标——由于标准版本的价位较低，人们更愿意选择使用标准版本。

单服务器版本类似于标准版本，也是作为 WebSphere 的入门级产品而提供的。区别在于，单服务器版本提供了比标准版本更多的功能（其中包括对 EJB 的支持）。它的目标是面向部门级应用程序，并且，单服务器版本对于创建在性能、吞吐率、24/7 操作等方面要求不高的完全电子商务解决方案非常有用。它是一个功能受限的版本，但对于初级入门者非常有用。本书中出现的许多例子都是基于单服务器版本的。只有当需要演示的功能在单服务器版本中不提供时，所给出的例子才是基于完全高级版本（Full Advanced Edition）的。

1.2.2 完全配置的 WebSphere 高级版本

WebSphere 高级版本是 IBM 的标志性应用程序服务器。这一版本完全与 J2EE 兼容，并且最引人注目。它提供了企业版中大多数非常有用的功能，但没有企业版那么复杂。笔者认为，它是世界一流的 Java 应用程序服务器，为开发人员以及管理员提供了非常深入的功能。高级版本包括对 EJB、事务数据库连接以及 Web 服务的支持。本书后面将要详细讨论的 EJB

与 Web 服务，它是开发企业应用程序的令人振奋的重要技术。

完全高级版本提供了比单服务器版本更为丰富的部署环境。它允许用户运行几个应用程序服务器实例（甚至可能不同的主机上运行），并且使得这一过程就像在一个虚拟的服务器上完成。这就意味着，用户能够启动多个服务器，就像只有一个服务器那样协同工作。这一特性使得用户能够无缝地将一系列服务器联系起来共同工作，从而提高系统的性能与可靠性。虽然这对于任何应用环境都可能是一件好事情，但对于电子商务应用程序却是特别关键的，因为创建电子商务应用程序的最基本目标就是在 Web 上向用户开放公司业务。Web 上的大量用户都可以看成是公司业务的潜在客户；但从技术的观点来看，这对应用程序会造成非常重的负担。这种负担主要来自计算负载以及因此而引发的性能问题，另一方面也是因为 Web 环境比典型的客户/服务器环境缺少可控制性。因此，运行电子商务应用程序的部署平台必须具有高度容错性与可扩展性。为了达到这一目标，最自然而又有效的方法就是建立能够共同支持应用程序运行的主机群。但在大多数情况下总是说来容易做起来难，这是因为，为了协调不同机器共同工作，还需要进行许多努力。在完全高级版本中，用户无需任何努力就能够实现这一目标；因此，除了进行正确的配置之外，它是真正意义上的“即插即用”。

1.2.3 WebSphere Enterprise Extension

WebSphere 的企业版本是 IBM 为服务器市场提供的高端解决方案。在 WebSphere 4.0 版中已经停止提供 WebSphere 企业版，因此，WebSphere 企业版最高仅仅在 3.5 版本中存在。WebSphere 企业版最初面向的客户群是拥有电子商务应用程序的组织，并且这些组织的电子商务应用程序对分布性以及事务相关的需求非常迫切。这样的应用程序对系统要求非常高，IBM 解决这一问题的方法是，通过将 WebSphere 高级版本与一系列“功能强大的工具”进行集成，从而进一步丰富 WebSphere 高级版本所提供的功能。现在，如果需要使用这些工具，就必须在一个称为“Enterprise Extension”的包中单独购买。这些工具包括：

- Component Broker
- TXSeries
- MQSeries
- WebSphere Enterprise Service

很难想象，会存在需要同时使用上面列出的所有产品这样的解决方案，因为在有些情况下它们的功能存在重叠。但人们不能忽略这样一个事实，企业版本就是为任何可能存在的高级分布、事务支持、消息传递以及任何可能的应用需求而开发的产品。

IBM 在 4.0 版中进行的一项主要工作，就是将 3.5 版中提供的 3 个不同版本（标准版、高级版与企业版）合并成为一个单独的代码行。在 4.0 版中，先前仅仅能够在企业版上运行的功能，现在只要在 WebSphere 高级版本中安装 Enterprise Extension 就能同样地运行。

Component Broker

Component Broker (组件代理) 是一种支持 CORBA (Common Object Request Broker Architecture, 公共对象请求代理体系结构) 模型以及 EJB 的对象引擎。由于 EJB 规范已经被扩展为能够在 IIOP (Internet Inter-ORB Protocol, Internet ORB 间协议) 上放置 RMI (Remote Method Invocation, 远程方法调用), 因此从定义上就规定了 J2EE 服务器有许多 CORBA 支持。我们相信, Component Broker 仅仅对那些已经使用其他编程语言 (例如 C++) 编写应用程序的组织, 以及对那些在其环境中包含有不同编程语言 (以至于不能迁移到 Java 环境) 的组织存在吸引力。这是因为, Component Broker 中提供的大多数附加功能都与不同于 Java 的其他语言所支持的代码以及功能相关, 或者与不同于 EJB 的对象模型相关。IBM 提供了 C++ CORBA 软件开发工具包, 用来使得将 C 与 C++ 代码集成到 J2EE 环境以及构建与部署 C++ CORBA 客户的过程成为一种标准方法。

除了作为 ORB (Object Request Broker, 对象请求代理) 并支持 EJB 之外, Component Broker 也提供了对大量其他服务的支持, 这些服务对于构建业务应用程序非常有用。这些服务包括并发控制、生命周期、事件、通知、对象识别与命名、安全性、事务、会话、查询 (通过 OOSQL (Object-Oriented Structured Query Language, 面向对象的结构化查询语言) 标准实现)、cache, 以及负载管理服务。

TXSeries

TXSeries 实际上是两个不同的产品——TXSeries CICS (Customer Information Control System, 客户信息控制系统) 与 TXSeries Encina。这两个产品都是具有高事务需求的应用程序所使用的事务监视工具, 并且这两个产品都与 Java 以及 Internet 完全无关。IBM 试图通过 TXSeries 为开发人员提供能够非常方便地将这些产品“粘合”到 WebSphere Application Server 中的连接器。

MQSeries

MQSeries 是 IBM 提供的分布式消息传递解决方案。这个产品以及将这个产品集成到 WebSphere 中的吸引力, 就是它被许多公司的开发人员所使用。MQSeries 类似于 TXSeries, IBM 试图为在 Java 代码中使用这一产品提供优秀的连接器 (或适配器)。MQSeries 也包含在 JMS (Java Messaging Service, Java 消息传递服务) API 环境中使用 MQSeries 提供的支持。

Enterprise Service

Enterprise Extension 的最后一个主要组件就是 WebSphere Enterprise Service, 它在本质上是 IBM 提供的非 J2EE 标准的功能集合。这些服务包括:

- ActiveX Bridge (ActiveX 桥): 允许开发人员基于 Microsoft 的 COM (Component Object Model, 组件对象模型) 技术扩展 J2EE EJB 对已有软件组件的连接性。

- Internationalization Service (国际化服务) : 能够适应位于不同地区与时区的客户关于内容表现与业务逻辑的需要, 提供实现满足此需要的 J2EE 应用程序的机制。需要说明的是, Java 为国际化提供了内置的支持, 但这一功能在 J2EE 应用程序 (在这种应用程序中, 单独的事务可能需要跨越几个不同的地区) 设置中变得更加复杂。
- Extended Messaging Support (扩展的消息传递支持) : 允许 J2EE 服务器通过 JMS 接收器接受传入的异步消息。
- Business Rule Bean (业务规则 Bean) : 使得能够在应用程序代码之外并且在一个外部规则集合中封装业务规则。
- Shared Work Area (共享工作区) : 为 J2EE 应用程序提供了一个关于共享内存的抽象, 从而允许开发人员共享对象之间的信息, 而不必显式调用方法或传递消息。
- Business Process Bean (业务过程 Bean ; 技术预览) : 提供了通过不同事务范型进行高级业务处理的支持。这个框架允许进行配套的 J2EE 事务, 使用基于时间或基于消息的触发器自动初始化事务, 并且允许对 J2EE 组件 (例如 EJB 和连接器) 进行并发调用。

1.2.4 平台支持

目前, WebSphere 服务器的所有产品都能够在大多数主流的操作系统上使用。它可以在 Microsoft 的 Windows NT 与 Windows 2000 上使用, 也可以在大多数 UNIX 变体 (包括 Sun Solaris, IBM AIX, RedHat Linux, HP-UX) 上使用。由于它是 IBM 公司的产品, 因此也可以在 IBM AS/400 上使用。用户可以在 Java 应用程序服务器的 classpath 中包含 JDBC (Java Database Connectivity, Java 数据库互连) 驱动程序, 从而连接到任何拥有这个 JDBC 驱动程序的关系数据库上。由于某些元素需要使用 WebSphere 的内置特性 (例如数据库连接池、自动化的持续保留等), 因此在某些情况下插入此驱动程序是不够的。除了 IBM 的 DB2 之外, WebSphere 服务器支持的主要数据库平台就是 Oracle 8。本书中的例子将使用 DB2 和 Oracle——但利用 JDBC 支持, 它们能够非常容易地在任何其他数据库上实现。支持的主要数据库包括:

- DB2
- Oracle
- Informix
- Sybase
- SQL Server

关于所支持产品的更多详细信息, 请参见如下站点:

http://www-4.ibm.com/software/webservers/appserv/doc/v40/prereqs/ae_v401.htm

1.3 WebSphere 产品家族

本节将介绍除 Application Server 之外的 WebSphere 产品家族的其余产品。

1.3.1 WebSphere Studio Application Developer

WSAD (WebSphere Studio Application Developer) 是 WebSphere 产品线中的新产品,它是 VisualAge for Java 与 WebSphere Studio 的合并。因此,它所面向的目标是 Java 开发人员、Web 网站与脚本构建者等等。

VisualAge for Java 是 IBM 的 Java IDE (Integrated Development Environment, 集成开发环境)。作为功能非常齐全的 Java IDE,它是用来开发、测试与调试 Java 应用程序的高级工具,并包含有对 WebSphere 的内置支持。也就是说,它允许用户开发并且在 VisualAge for Java 中运行 servlet, JSP 以及 EJB。VisualAge for Java 所提供的执行环境是实际 WebSphere 服务器的简化版本。

IBM 意识到大型组织开发资源的不足,为此特别开发了 VisualAge for Java 的 2 种变体,从而允许有一点点或者根本没有任何 Java 技能的开发者生成代码并且开发应用程序。这 2 种变体分别是:

- VisualAge Application Rule: 允许用户快速构建与部署基于规则的应用程序。通过使用 IBM 的其他工具以及中间件,用户就能够相对比较容易地构建大型应用程序。
- VisualAge Generator: 允许快速开发与部署基于 Web 的应用程序,它屏蔽了事务、通信以及数据库的复杂性。

WebSphere Studio 为构建动态 Web 页面与应用程序提供了一个功能强大的工具集。它为设计纯 HTML 页面以及带有 JavaScript, DHTML (Dynamic HTML, 动态 HTML), JSP 的页面提供了可视化的创作环境。正如其他与 WebSphere 相关的产品一样, WebSphere Studio 能够很好地与 WebSphere 服务器以及 VisualAge for Java 协同工作。

另一个为创建带有动态内容的 Web 页面提供支持的可视化编辑环境就是 WebSphere Homepage Builder (以前称为 TopPage)。

1.3.2 WebSphere Business Components

WebSphere Business Components 是一个为节省应用程序开发人员的时间而设计的组件库。这些各种各样的组件具有明确定义的接口并且提供了丰富的功能。在这个包中提供的组件基本上都是 EJB,这大大增加了它们的可用性与可移植性,并且缩短了学习使用它们的过程。

WebSphere Business Components 是基于先前的一个名为 SanFrancisco 的产品开发的。

当几个大客户找到 IBM 公司要求帮助升级一系列企业级、会话关键应用程序到新技术时,就开始启动了 SanFrancisco 工程。这些公司在各种业务领域都有很大的应用程序系列,并且

它们有兴趣从原来的技术（通常就是运行在大型机上的 COBOL）转移到面向对象、以网络为中心的体系结构上来。但是，完成这项工作需要非常庞大的技术迁移以及重写应用程序，这其中隐含的工作量是如此巨大，以至于这些公司不得不向大型软件厂商（IBM）寻求指导与帮助。

在确定了与 IBM 相关的技术迁移内容之后，就进一步指定了 IBM 能够为这些公司提供帮助的一些领域。技术选用、重新培训、知识转让就是 IBM 能够为这些公司提供帮助的领域。之后，分析与设计定义的目标为：来自 IBM 公司的技能熟练的开发人员以及分析员将参与到这些公司的内部开发工程中，从而帮助引导开发过程并且确保正确地进行设计。虽然形成了这些关系，但是，很显然还有许多工作等待处理。正如大多数新技术所遇到的情况一样，人们最初具有较高的期望，但是，在最初的工作中缺乏许多具有市场成熟度的底层结构组件。多层软件体系结构以及以网络为中心的大应用程序底层结构也不例外。这些与 IBM 合作的公司和 IBM 公司很早就认为，如果能够使用一些底层结构组件，并且在以后创建业务系统的工作中重用这些组件，那么就能够使得各方极大地获利。这样就诞生了 SanFrancisco 工程。

在对 SanFrancisco 将要应用的业务系统进行广泛的分析之后，SanFrancisco 工程就开始启动了。然后，它被创建成为一种能够在多个业务系统与领域中重用的业务处理组件集合。从一开始，这个工程的基本底层结构就被定义为采用面向对象框架的思想——例如，不是采用类库的思想。这样，SanFrancisco 就提供了一种底层结构——一致的设计与编程模型，并且提供了能够被修改与扩展的业务逻辑。当然，可重用面向对象组件并不是软件业界的新概念，甚至在框架方面，市场上也已经存在一些非常优秀的实现。但毕竟，SanFrancisco 实现了所有框架中最大的、最完全的框架集合。此外，它是包含高级业务功能的首批框架示例之一；目前使用的大多数框架仅仅解决了较低层的系统问题与用户界面问题。

那么，该如何描述此项工程呢？SanFrancisco 是由 IBM 领导的、多家公司共同协作的工程，涉及到成百上千家愿意将自己的系统体系结构基于 SanFrancisco 框架的国际 ISV（Independent Software Vendor，独立软件供应商），他们愿意帮助维持此产品的稳定性并且形成该产品的独特结构。

虽然 SanFrancisco 包含有许多框架、服务与实用程序，但这一环境的核心部分由三个软件层组成，并且这些软件层与业务应用程序联系越来越紧密：Foundation 层、Common Business Objects 层、Core Business Processes 层。上述每个层都依赖于前一层（见图 1-1）。

Foundation 层实现了可以被 SanFrancisco 环境中所有对象使用的底层结构与服务。它支持对象一致性、事务处理、分布与位置透明性、命名、安全性、所有者等概念。它是最大的层，也是提供对所有应用程序都非常有用的独立于上下文并且与底层结构相关支持的软件层。

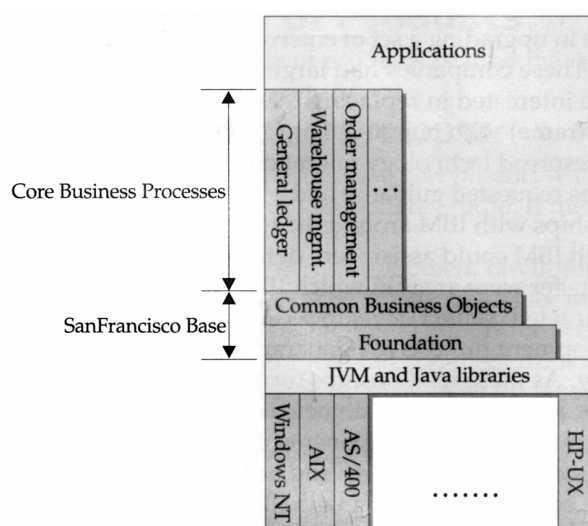


图 1-1 SanFrancisco 框架

Common Business Objects 层实现了大多数业务领域中共同存在的基本的业务对象集合。这些基本业务对象包括地址、货币相关对象、客户与位置等。

最后的 Core Business Processes 层提供了用来形成微应用程序集合的业务对象与过程的完全集合，并且包括对于相关领域非常有用的可用功能。这一软件层实际上并不是一个层，而是一个垂直层的集合，每个垂直层提供了对某一领域的支持。显然，将要被选择的首批垂直层就是那些与 IBM 客户最为相关的；但是，随着时间的推移，还将陆续添加其他更多的垂直层。

通过继承或组合（或同时使用这两种方法），那么用户在理解与使用 Core Business Processes 和 Common Business Objects 的基础上，就能够定制 SanFrancisco 框架，从而使得最终的目标产品非常引人注目。一个最简单的重用情况就是，从这个框架中使用业务对象与过程，并同时设置作为业务对象内置参数的属性。另一个级别的复杂性应用允许用户改变创建对象的方法，并且定义哪个类将被用作实例化某个业务概念。通过为业务类提供自己的扩展，就能够保证仍然在这个框架中的同时自定义一些行为。或者，用户能够通过功能替换、功能继承以及功能修改来完全扩展一个域中的类。

如图 1-1 所示，Foundation 层与 Common Business Objects 层形成了 SanFrancisco Base（SanFrancisco 基础）。这些基础层提供了构建多层、分布式、面向对象应用程序所需的服务与底层对象支持，并且提供了遵循 SanFrancisco 模式的基本对象集合。这样不仅允许应用程序构建者不必过分关心复杂应用程序体系结构所涉及的细节，同时也帮助构建者快速开始开发过程。

Core Business Processes 利用不同领域的丰富功能进一步完善了 SanFrancisco Base，并且所有这些功能都构建在 SanFrancisco Base 上。如图 1-1 所示，应用程序构建者不必局限于框架

应该如何使用。可以在一个或多个 Core Business Processes 中构建应用程序,在这种情况下, SanFrancisco 能够提供最强有力的支持。如果某个业务领域没有被任何过程框架所覆盖,或者,如果提供的功能与所需的功能存在很大的不同,那么应用程序就能够使用 Common Business Objects 层作为在 SanFrancisco 环境中发挥作用的 reusable 业务对象集合。这种用法节约了大量的开发时间以及测试时间,但仍然比直接使用 Core Business Processes 层需要做更多的工作。

最后,有些新的业务对象能够直接在 Foundation 层上创建。在这种情况下,新的业务对象仅仅使用了底层服务,这样也解决了在系统体系结构中面临的大多数复杂问题。显然,上述方法的任何组合运用都是允许的;一个应用程序可能直接使用了 Core Business Processes 或某些 Common Business Objects,同时也使用了 Foundation 层服务的对象集合。一旦应用程序的所有部分都使用了这些服务,这些对象就能够协同工作。

已经说过,只要对象是在 Foundation 层上构建的,它们就能够形成一个一致的应用程序结构。笔者并不推荐构建在 Foundation 之外的对象用于应用程序之中,并且这样做是比较困难的。只要对象使用了 Foundation 层服务并且遵循 SanFrancisco 模式,它们就能够融合到 SanFrancisco 之中并且充分利用 SanFrancisco 带来的便利。没有按照这种方法创建的对象却需要解决许多困难问题,例如一致性、事务等。更为糟糕的是,如果决定脱离 SanFrancisco 模型,还必须同步并维护这两个模型。例如,如果有一个单独的业务事务中包含有使用 SanFrancisco Base 的对象,并且有一些需要自我管理事务的对象,那么就必须将所有这些对象同步到一个事务中——这是一项非专业技术人员根本无法完成的工作。

SanFrancisco 工程并不成功。实际上,与 IBM 所投入的大量资金(以及投入到 Taligent 中的资金;Taligent 是 SanFrancisco 的某些根任务)相比, SanFrancisco 工程是非常不成功的。当 WebSphere 出现并成为 Java 与电子商务平台的领头产品之后,IBM 迅速重新定位并重新创建 SanFrancisco 作为 EJB 的顶层基础框架,并且使得 WebSphere 作为部署环境。WebSphere Business Components 仍然在扩充,很有可能成为具有光明前景的产品线;它当然实现了大量业务功能!

1.3.3 WebSphere Transcoding Publisher

如今的 Internet 开发应用程序面临的一个挑战,就是能够访问 Web 的设备种类繁多。除了传统基于 PC 以及基于 UNIX 的浏览器之外,还有越来越多的 PDA (Personal Digital Assistant, 个人数字助理)以及蜂窝电话提供 Web 访问功能。以一种与这些设备功能相匹配的方式产生内容,要么是必须的,要么是非常有帮助的。也就是说,为了适应这些设备的屏幕显示,必须对内容进行重新格式化,并且可能需要以一种不同的标记语言将这些内容表现出来。例如,蜂窝电话就可能需要使用由 WML (Wireless Markup Language, 无线标记语言)、部分 WAP 标准或 CHTML (Compact HTML, 压缩 HTML) 实现的内容。

WebSphere Transcoding Publisher 为用户提供了在服务器方根据需要重新格式化并适配用户数据的能力。

1.3.4 WebSphere Voice Server

Internet 应用程序的最新发展动向是通过呼叫中心与语音端口提供语音服务。IBM 通过 WebSphere Voice Server 迁移到这一领域。这个服务器基于 WebSphere 服务器以及 VoiceXML (XML 的一种特殊变体, 用来支持基于语音的文档), 并且提供了构建基于 Web 的语音端口以及呼叫中心的工具。

1.3.5 WebSphere Portal Server

就在几年前, 门户还被认为是 Internet 上的“下一个重大事件”。这种热潮现在已经退去, 今天, 任何公司都能够使用诸如 WebSphere Portal Server 这样的产品来构建自己的 Web 门户网站。它允许用户进行注册并接收个性化内容——当然, 也能够被创建成为包含任何用于满足客户、业务伙伴以及雇员需要的特定内容。

1.3.6 WebSphere Everyplace Suite

WebSphere Everyplace Suite 是 IBM 提供的移动计算与移动 Internet 播放工具。虽然它的一些功能与 WebSphere Transcoding Publisher 的功能有重复, 但这个产品的目标是为通过 PDA 或蜂窝电话访问内容的移动用户提供完全独立的解决方案。WebSphere Everyplace Suite 包含有一个称为 WebSphere Everyplace Embedded Edition 的客户方组件。

1.3.7 WebSphere Personalization

WebSphere Personalization 具有为用户提供满足其个人需要与使用偏好的个性化内容的功能。显然, 只要经过相对较小的编程努力, 就能够利用 WebSphere Application Server 达到相同的目的。

1.3.8 WebSphere Edge Server

WebSphere Performance Pack 是 WebSphere Edge Server 的前称, 它提供了大量的性能支持能力。这些性能支持能力包括负载平衡、基于内容的路由 (影响服务质量)、内容过滤以及内容 cache。需要强调的是, 这一产品能够在各种 Web 服务器上工作, 而不仅仅是 IBM 的服务器。

1.3.9 WebSphere Site Analyzer

WebSphere Site Analyzer 提供了分析 Web 站点访问者活动与使用情况的能力。具体表现在, 它提供了用来分析 Web 站点访问者行为的所有必要数据, 同时, 它与 IBM 的其余产品 (例如 WebSphere Application Server 以及 WebSphere Commerce Suite) 紧密集成。

1.3.10 Tivoli Policy Director

Tivoli Policy Director 是 Tivoli 产品家族中的一部分, 它的目标是为电子商务以及基于 Web 的应用程序管理安全性策略。

1.3.11 WebSphere Commerce Suite

WebSphere Commerce Suite 为快速、简单地构建电子商务网站提供了工具。WebSphere Commerce Suite 包括一个名为 WebSphere Payment Manager 的子产品，正如其名称所隐含的那样，这个产品用来管理支付。

1.3.12 WebSphere Business-to-Business Integrator

B2B 是又一个随着泡沫的破灭而丧失光环的名词。WebSphere Business-to-Business Integrator 使得用户能够将公司的企业计算系统与客户、供货者以及业务伙伴联系起来。完成所有这些功能的基本技术就是 XML，但这一产品也组合了 IBM 的集成与事务产品。

1.3.13 WebSphere Partner Agreement Manager

WebSphere Partner Agreement Manager 允许组织自动化与合作伙伴（例如供货者）、客户以及电子市场的交互。这种自动化有助于改善供应链的效率与效能。

1.4 小 结

由于本书是关于使用 WebSphere 工具开发 Web 应用程序并且在 WebSphere Application Server 上部署这些应用程序的参考手册，因此，在书中没有深入详细地讨论本章提及的许多 WebSphere 家族产品。实际上，许多这些产品都应该具有自己的“完全参考手册”。由于本书仅仅关注于 WebSphere Application Server，因此在书中详细讨论了 EJB, servlet, JSP, J2EE 的各方面，以及部署拓扑等。

第 2 章 安装与启动 WebSphere

本章将进入 WebSphere 并讲述 WebSphere 的安装全过程。在这里，将给出在 Windows NT（或 Windows 2000）以及在 Linux 上安装 WebSphere 的全部过程。目前，IBM 并不支持 Windows XP，IBM 计划将在 WebSphere 5.0 产品线中开始正式支持 Windows XP。虽然如此，笔者已经在 Windows XP 系统上运行了 WebSphere 很长一段时间；只要不是试图将 WebSphere 用于产品环境，笔者认为 WebSphere 是可以在 Windows XP 上使用的。在 UNIX 平台上进行 WebSphere 的安装过程与在 Linux 上非常类似。在本章后面，给出了一些疑难解决提示，当然，大多数情况下，安装过程都应该非常顺利。在本章结束时，读者应该能够在自己的机器上毫无问题地安装 WebSphere。

为了开始安装过程，首先需要打开安装程序，这个程序在 CD 上（或者从 www.ibm.com/websphere 下载此程序）。在本书配套 CD 中提供了 WebSphere Application Server 高级单服务器版的评估版本。

2.1 Windows 安装

在启动安装程序时，弹出的第一个窗口就是警告窗口。这个警告窗口除了说明 WebSphere 的版权信息之外，还建议用户在安装过程中关闭任何已经打开的应用程序。如果这时机器正在运行 Web 服务器，或正在运行一个更早版本的 WebSphere，这一点就更为重要。在继续进行安装过程之前，必须关闭这些应用程序。

WebSphere 4.0 并不需要 Web 服务器。这是因为在 4.0 版中带有一个内置的、全功能的 Web 服务器，这个服务器在端口 9080 上运行。并不建议你将这个 Web 服务器用于产品环境，但它确实使得用户能够在不必安装 Web 服务器的条件下测试 WAS（WebSphere Application Server）的安装。

对于产品环境，笔者建议还是安装一个 Web 服务器（或 HTTP 服务器），以便与 Microsoft IIS，iPlanet，Apache 或 Lotus Domino 共同工作。WebSphere 的安装包括默认的应用程序服务器（也就是 IBM HTTP Server）。如果希望将 WebSphere 与一个不同的 Web 服务器一同使用，就必须在进行 WebSphere 的安装之前先安装 Web 服务器。我们将在第 6 章中详细讨论 Web 服务器。

如果已经安装了一个 WebSphere 版本，安装程序将自动检测到这个已有的安装，并且向用户提供进行升级还是进行全新安装的选择（见图 2-1）。当然，如果在此之前没有安装 WebSphere，就不会出现这个选择。

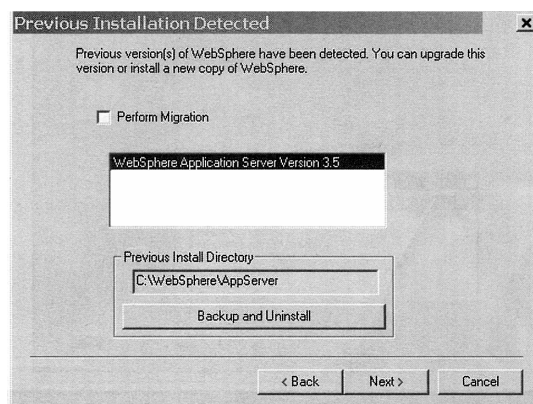


图 2-1 检测到一个已经存在的 WebSphere 安装

在进行选择之后（或者如果以前没有安装 WebSphere），将出现 Installation Options 窗口，这个窗口提供了两种不同的安装方法。第一种安装方法就是 Typical Installation（典型安装），在这种安装方法中将安装 IBM HTTP Server 以及 IBM JDK（Java Developer's Kit）1.3.0。第二种安装方法就是 Custom Installation（自定义安装），这种安装方法允许用户选择安装哪些组件。特别地，它允许用户选择不同的 JDK 并且安装其他 Web 服务器插件（见图 2-2）。需要说明的是，如果使用 JDK，那么首先就必须确保这个 JDK 能够与 WebSphere 正确地协同工作。显然，IBM 的 JDK 1.3.0 能够很好地与 WebSphere 协同工作。

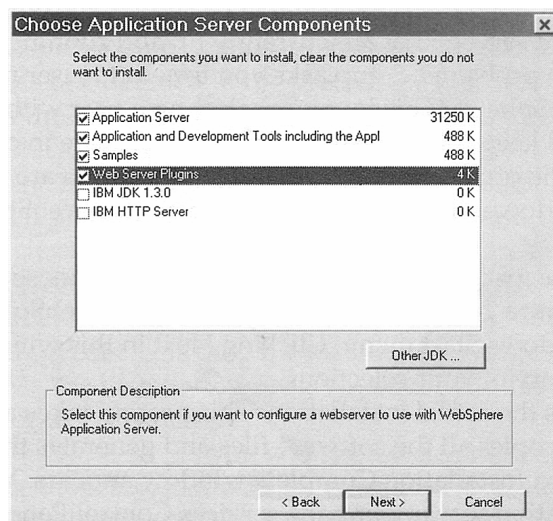


图 2-2 自定义安装

如果不想使用 IBM HTTP Server，请在这个对话框中取消相应的选择。如果选择了 Web Server Plugin，那么 WebSphere 将提示用户进一步选择希望安装的插件，如图 2-3 所示。

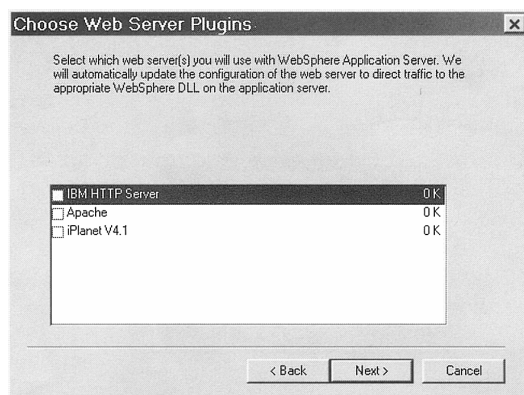


图 2-3 Web 服务器插件

如果所选择的安装目录中已经存在正在工作与运行的 WebSphere，安装程序将警告用户，说明有些文件正在使用，应该关闭这个服务器。如果要为那些已经启动并且正在运行的服务器安装 Web 服务器插件，系统也将给出类似的警告。

如果不存在上述问题，安装程序将要求用户选择用于管理 WebSphere 的用户名与口令（包括选择具有“Administrator”（管理员）权限的用户）。应该将这些信息放置在比较安全的地方，以防忘记。下一步就是选择产品安装目录。如果不安装 IBM HTTP Server，那么这个窗口将仅仅包含一个选项（也就是 WebSphere Application Server 自身）。

在指定了安装目录之后，下一步就需要选择程序文件的文件夹，如图 2-4 所示。这一选择将决定 WebSphere 程序应该出现在 Windows 开始菜单的什么地方。在这个窗口中单击 Next，这样将会给出一个窗口，对所进行的选择进行总结。

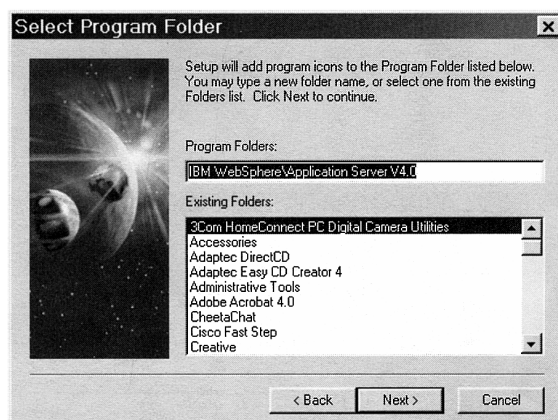


图 2-4 程序文件夹

下一步就是开始实际的安装过程。在这一过程中，安装程序需要拷贝所有的软件文件并且

产生必要的安装文件，因此需要花费几分钟的时间。最后，将会出现一个标准的 Installation Complete (安装结束) 窗口。这时，就可以使用 Control Panel (控制面板) 服务打开 WebSphere 管理服务器。现在，就可以进行本章后面将要讨论的“测试安装”一节。

2.2 Linux 安装

用户可以在 IBM AIX, Sun Solaris, HP-UX 以及 Linux 上使用 WebSphere 4.0。对于所有的 UNIX 安装，用户都需要运行 WebSphere 安装光盘上各自目录下的 install.sh 脚本。如果下载了这个程序，那么首先需要从所得到的 tar 文档中解压这些安装文件。

所有的 UNIX 版本都给出了一个 GUI (Graphical User Interface, 图形用户界面)，用于引导用户按照上一节中与 Windows 安装类似的过程完成 WebSphere 的 Linux 安装。惟一的不同之处就是图形用户界面的外观以及路径名 (在 Linux 中路径名使用的是正斜杠 (/)，而不是反斜杠 (\))。

由于 Linux 的广泛可用性，我们在这里将重点关注 WebSphere 的 Linux 安装。其安装过程与在所有其他平台上的安装过程相似。在 UNIX 安装中需要考虑的关键问题，就是为了保证 WebSphere 能够正确工作，必须检测需要使用哪个操作系统补丁。这一点对于 Solaris 以及 HP-UX，或者对于不同于 RedHat Linux 7.0 的其他 Linux 变体尤为重要。关于所需补丁的最新列表，以及与此安装相关的特殊注意问题，可以在 IBM 提供的产品发布说明中找到。安装程序通常准确反映了任何操作系统的不足或差异。

在运行 install.sh 脚本之后，用户将看到一个欢迎窗口，要求用户单击 Next 以继续安装。正如在 Windows 中进行安装一样，安装程序也将检测任何关于 WebSphere 的先前安装并且提供迁移它们的方法，如图 2-5 所示。

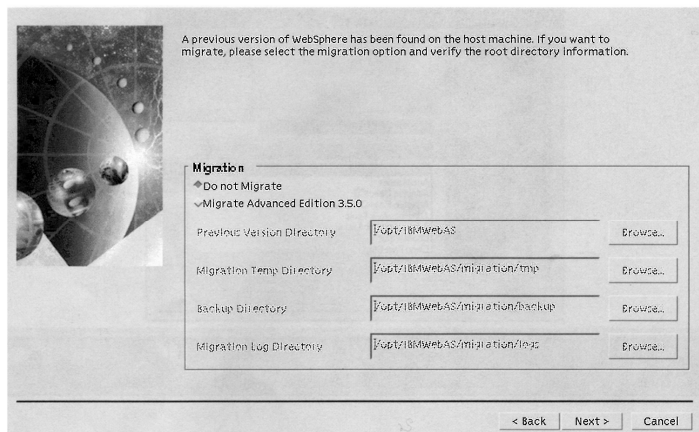


图 2-5 检测到先前已有的安装

下一步是选择典型安装还是自定义安装。正如在 Windows 安装一节中所解释的那样，典型安装将使用 IBM 的 JDK 1.3.0 以及 IBM HTTP Server。如果希望使用一个不同的 JDK 或 Web 服务器，就必须选择自定义安装。特别地，如果已经安装并运行了 Apache 并且希望继续使用 Apache，就要选择自定义安装。

如果选择了自定义安装，那么安装程序将提示用户选择安装组件，如图 2-6 所示。如果选择 Web Server Plugin，安装程序将要求用户进一步选择想安装哪个插件，如图 2-7 所示。

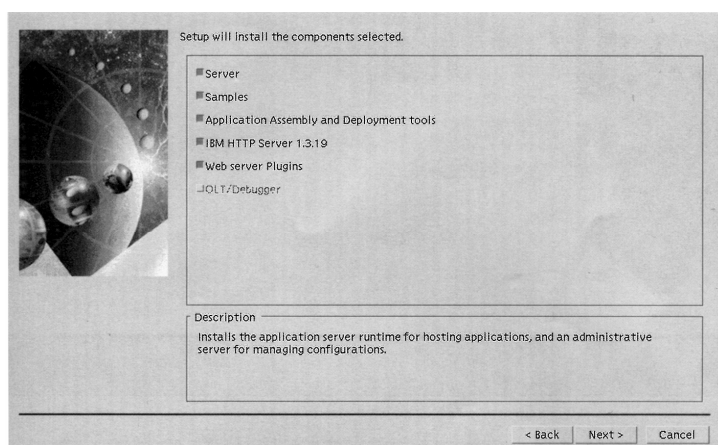


图 2-6 自定义安装

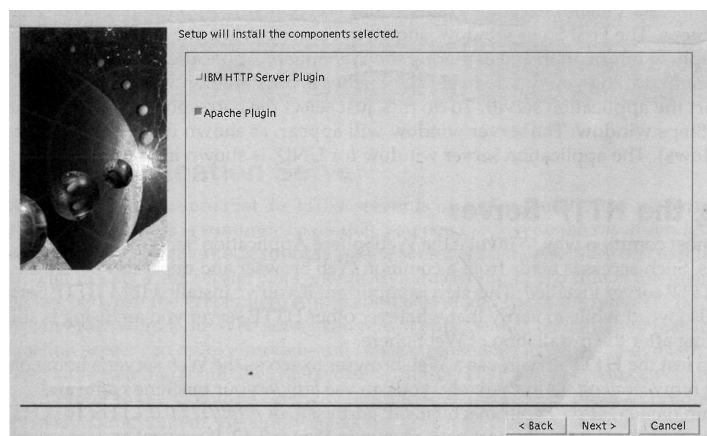


图 2-7 Web 服务器插件

下一步是选择安装目录。如果不安装 IBM HTTP Server，屏幕上将仅仅出现一个表项（就是 WebSphere 服务器）。在选择目录之后，安装程序将给出一个关于用户所做选择的总结窗口，如图 2-8 所示。单击 Install（安装）按钮，将开始安装过程。

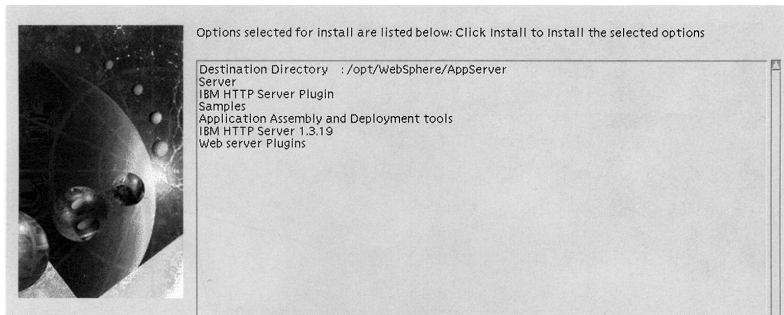


图 2-8 已选择的安装选项的总结

最后，将出现用来说明安装过程结束的窗口。这样就完成了整个安装过程，可以继续进行安装的测试了。

2.3 测 试 安 装

为了测试安装，首先必须启动应用程序服务器。WebSphere 4.0 中包含有一个称为 First Steps 的新功能，用来向用户介绍 WebSphere。在 UNIX 安装过程结束之后，First Steps 窗口将自动启动。在 Windows 中，为了启动 First Steps 窗口，应该在开始菜单上选择 Programs|IBM WebSphere|Application Server V4.0|First Steps。如图 2-9 所示，First Steps 窗口可提供关于 WebSphere 的信息，并且引导用户探索 WebSphere Application Server。

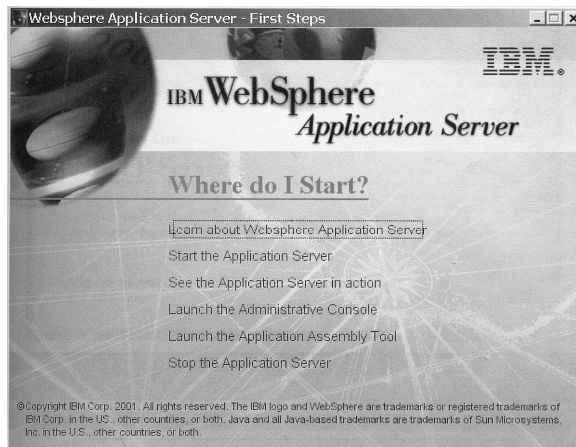


图 2-9 First Steps 窗口

为了测试安装是否成功，可以首先启动应用程序服务器。为此，应该从 First Steps 窗口中选择相应的项。这时，将会出现服务器窗口，如图 2-10 所示（对于 Windows 系统）。UNIX 下

的应用程序服务器窗口如图 2-11 所示。



图 2-10 启动 WebSphere Application Server (在 Windows 环境下)

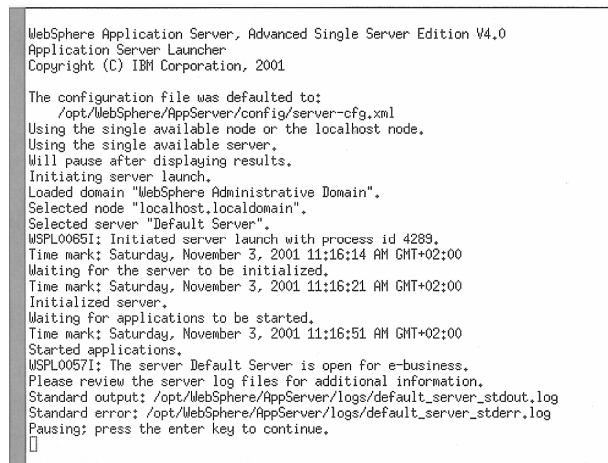


图 2-11 启动 WebSphere Application Server (在 UNIX 环境下)

2.3.1 测试 HTTP 服务器

激活 WebSphere Application Server 的最常用方法就是通过 HTTP 访问。这种访问是由普通的 Web 浏览器发出的, 并且在很大程度上依赖于所安装的 HTTP 服务器。如果已经安装了 IBM HTTP Server, 这个步骤就非常重要。验证正在使用的其他 HTTP 服务器在安装了 WebSphere 之后仍然正常工作也是值得做的。

为了测试 HTTP 服务器，应该使用 Web 浏览器访问 Web 服务器的主页（所使用的地址是 `http://localhost`；或者可以使用 `http://<机器名与域名>`、`http://<机器的 IP 地址>` 或 `http://127.0.0.1`）。对于 IBM HTTP Server，用户将见到如图 2-12 所示的启动页面。如果这时出现了一个错误消息，那么就意味着 Web 服务器没有正确安装，或者没有启动。在这种情况下，就需要参见本章后面的“疑难解决”一节。



图 2-12 IBM HTTP Server 主页

2.3.2 测试应用程序服务器

到目前为止，已经确保 HTTP 服务器启动并正在运行。现在，就需要验证 WebSphere 也正在运行。为此，需要调用 snoop servlet。servlet 就是在服务器方运行的 Java 代码段（将在第 5 部分详细讨论 servlet）。servlet 能够产生动态内容，这也是 WebSphere Application Server 与底层 HTTP 服务器之间的主要区别。换句话说，Web 服务器仅仅能够返回“静态”内容，而应用程序服务器则能够调用 servlet 以及其他机制来产生动态内容。需要说明的是，HTTP 服务器并不是只能产生“静态”内容——实际上，用户也可以配置 Web 服务器使用 CGI（Common Gateway Interface，公共网关接口）脚本或其他技术来产生动态内容。但一般来说，由于引入了 servlet，这些技术正逐渐被淘汰。

那么，应该如何运行 servlet 呢？很简单：只要打开浏览器并指定 URL `http://localhost/servlet/snoop` 即可。这样所得到的结果如图 2-13 所示。这个 servlet 所产生的大多数输出都是动态的（意味着这些内容是在运行时产生的；例如出现在 Request Information 下面的列表中的 Remote Address 字段）。这个字段显示了发送此请求的浏览器的 IP 地址。如果可以使用另一台机器，用户也能够在这台机器上试着运行一个浏览器（这时需要使用适当的主机名激活 localhost），这样能够观察到不同的输出结果。如果由于某些原因导致 snoop servlet

没有启动，那么请参见本章后面的“疑难解决”一节。

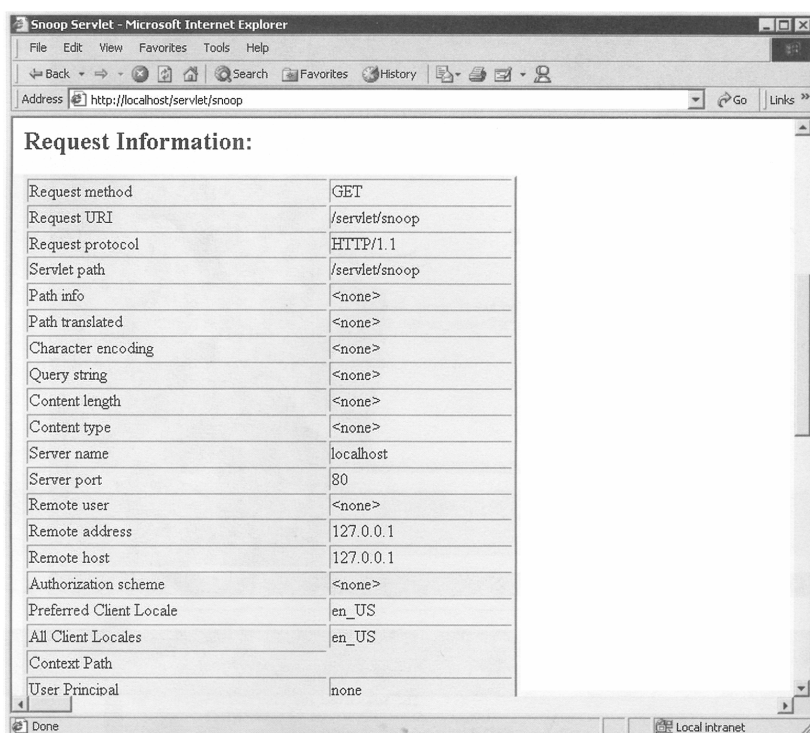


图 2-13 snoop servlet 的输出

如果 Web 服务器已经工作但 snoop servlet 没有工作，那么应该进行另一项测试。如前所述，一个内置的 Web 服务器是 WebSphere 运行时管理服务器的一部分。默认情况下，这个内置的 Web 服务器运行于端口 9080。一旦用户启动了一个应用程序服务器，就可以通过指向 `http://localhost:9080/servlet/snoop` 来测试它是否可以正常工作。如果能够正常工作，但 `http://localhost/servlet/snoop` 不行，那么就有可能是在 Web 服务器与应用程序服务器的通信方面存在问题——这个问题通常与 Web 服务器加载的 WebSphere 插件有关。

2.3.3 WebSphere 示例

再一个测试 WebSphere 安装的方法就是使用 IBM 的 WebSphere 示例。这些示例包括许多游戏应用程序，用来演示 WebSphere 的功能。只有当用户选择将它们作为安装过程的一部分进行安装后，这些示例才能使用。有两种方法可以启动这些 WebSphere 示例，一种是从 First Steps 窗口中选择 See the Application Server in Action，另一种是直接访问 `http://localhost?WSSamples`。

图 2-14 显示了 Samples 主页。用户可以从这个页面中探索各种示例及 WebSphere 的技术与功能。为了确信进行的安装能够正确工作，可以考虑对所有这些示例进行试用。例如，用户可从图 2-15 所示的 YourCo Sample 屏幕中选择 Whitepages。

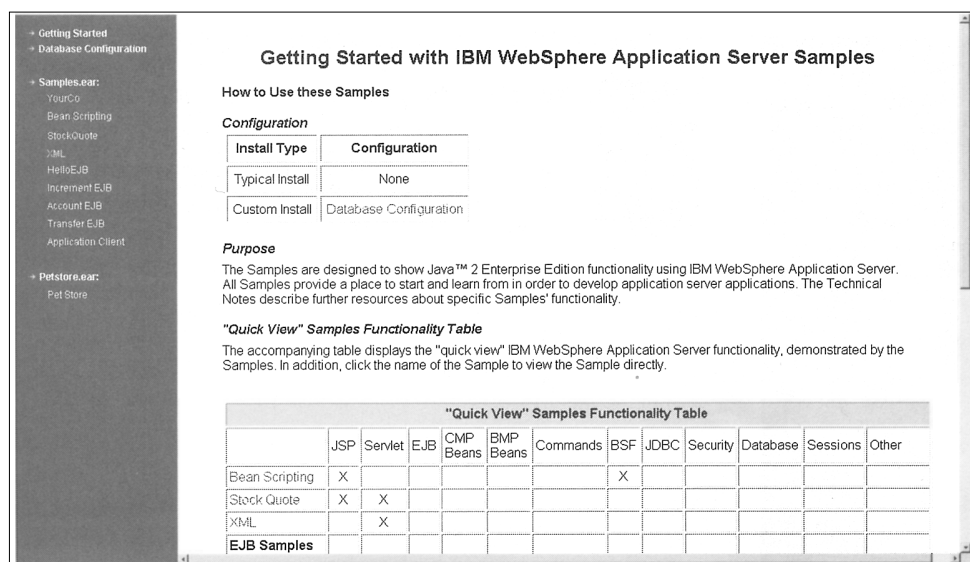


图 2-14 WebSphere Samples 主页

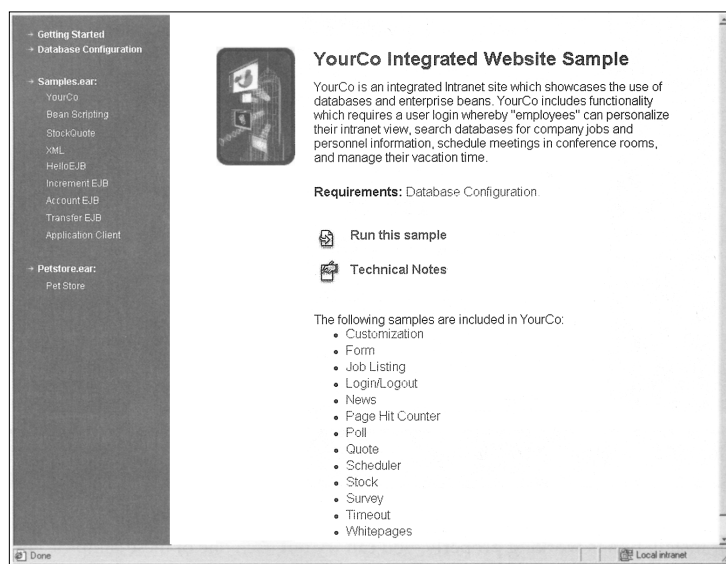


图 2-15 WebSphere YourCo Sample 屏幕显示

在所出现的窗口中，请选择 Run This Sample。然后，单击 Begin Your Search 按钮从而调用

应用程序服务器。图 2-16 给出了示例结果。用户也可以为搜索过程选择不同的参数，从而进一步对这个示例进行测试。

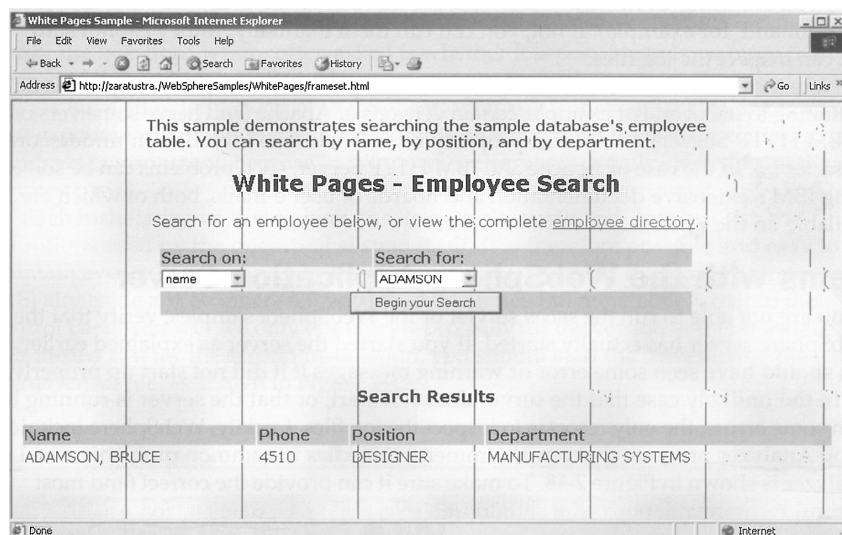


图 2-16 WebSphere White Pages 示例的输出

关于 WebSphere 示例的完全描述并非本章的讨论范围。但是，如果服务器能够正确地运行，用户将会发现这些示例具有很好的说明文档并且很容易操作。

2.4 疑难解决

WebSphere 是一个很成熟的产品，至少从 Java 与 Internet 技术的角度来看是如此。因此，大多数安装问题通常都会在安装程序中给出某种形式的警告。例如，在 UNIX 上进行安装时，安装脚本将自动检测任何操作系统不兼容性。

在任何情况下，如果存在问题，首先必须仔细检查自己的系统是否满足在产品文档中以及发布说明中给出的 WebSphere 使用要求。如果用户使用一个不同的 JDK 或 Web 服务器进行自定义安装，情况更是如此。与 JDK 的和与服务器的不兼容都有可能各种错误。

2.4.1 与 Web 服务器相关的问题

如果不能访问 HTTP 服务器，首先必须确信此服务器正在运行。

为了监视 HTTP 服务器（以及 Windows NT 或 Windows 2000 中的任何其他服务），应该查看 Windows 控制面板下的 Services 面板，如图 2-17 所示。正如图中显示的那样，IBM HTTPD

(Hypertext Transfer Protocol daemon) 服务器应该被自动启动。如果由于某种原因没有启动 HTTPD, 那么应该手工启动它——当然首先需要确信系统在正确的用户帐号下运行并且输入了正确的口令。如果手工启动过程也失败 (这种情况很少见), 则需要检查 HTTP 服务器日志是否存在什么特殊的问题。这些日志文件位于 IBM HTTP Server 安装目录的 logs 目录下, 文件名为 error 和 admin_error。

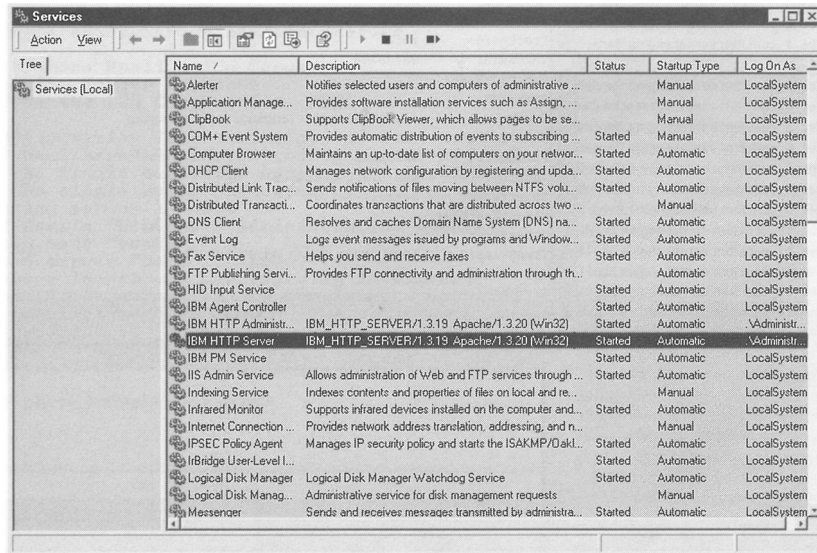


图 2-17 Windows Services 面板

如果正在使用 UNIX, 应该检查 httpd 进程是否正在运行 (例如, 可以使用 ps 命令)。如果没有运行, 就应该手工地运行此进程。如果不能启动运行, 就需要检查日志文件。需要说明的是, 不同的 Web 服务器具有不同的特性, 有时这些不同的特性很难被发现。例如, 如果在主机名中包含有下划线字符 (_), 那么某些版本的 Apache (以及有些版本的 IBM HTTP Server) 就不能正确运行。在 Apache 和 IBM HTTP Server 情况下, 可以使用 IBM 的大量文档与已经存储的大量用户电子邮件 (这两者都能够在 Web 上获取) 来解决这些问题。

2.4.2 与 WebSphere Application Server 相关的问题

如果不能运行 snoop servlet 或 WebSphere 示例, 那么应该验证 WebSphere 服务器已经被启动。按照前面介绍的方法启动这个服务器, 如果该服务器没有被正确启动, 用户就应该见到一些错误或警告消息。

如果出现服务器不能启动, 或出现服务器正在运行但存在错误的情况 (这些情况很少发生), 那么惟一的解决方法就是检查日志文件。幸运的是, WebSphere 包含有一个 Log Analyzer 程序, 能够提供关于一些常见问题的推荐解决方法。图 2-18 显示了 Log Analyzer。为了确

信它能够提供正确的、及时的解决方法，首先应该选择 File|Update Database 命令。

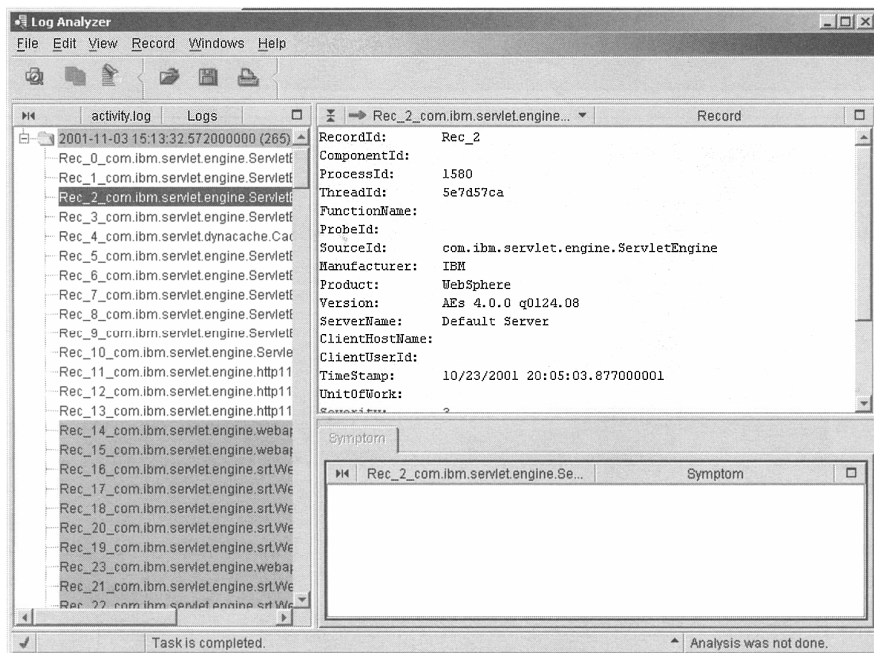


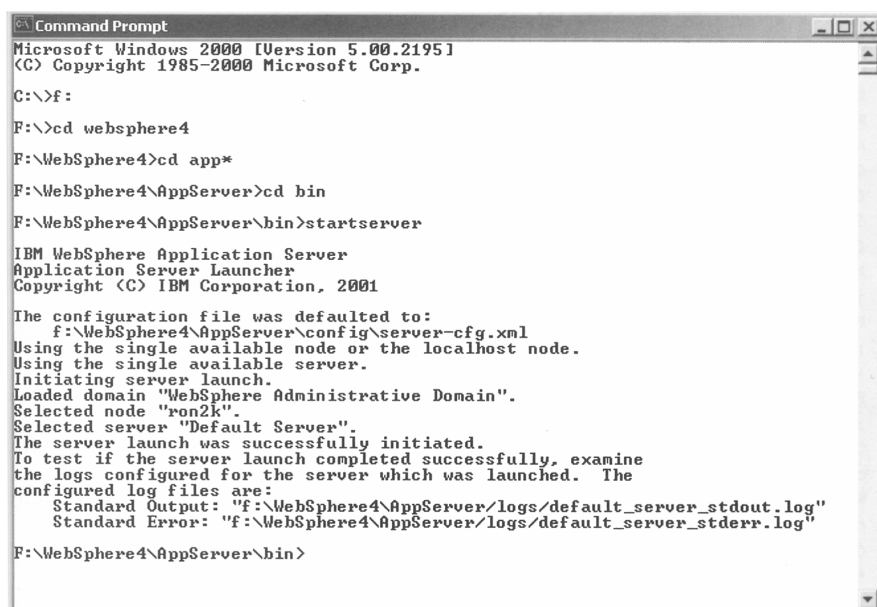
图 2-18 WebSphere Log Analyzer

2.5 多 WebSphere 安装

人们常常会问是否可能在系统中安装多个版本的 WebSphere。关于这个问题的回答是肯定的，如果用户已经有一个老版本的 WebSphere（例如 3.5 版或 3.0.x 版）并且希望开始试着使用 WebSphere 4.0，这种安装方式就显得尤其有用。进行这种安装的问题大多数都来自 Windows 环境，因为在 Windows 中，服务器通常是从 Services 面板中启动的。

WebSphere 的每个安装都有自己的 classpath 并且不使用操作系统使用的定义。因此，就有可能比较容易地同时维持一个 1.1.x JRE（Java Runtime Environment，Java 运行时环境）的 WebSphere 3.0.x 安装以及一个 WebSphere 4.0 安装。需要记住的是，在控制面板中用户将仅仅能够看到一个表项——通常是用户首次安装的表项。这意味着，对于第二个安装，用户应该使用 startserver.bat 文件（如图 2-19 所示）启动服务器。

在进行这项工作时，应该总是查看日志文件，以确信服务器被正确启动。这是非常重要的，因为在控制台上提供的消息并没有多大帮助。进行这些工作的目的，就是为了确信服务器的启动过程完全成功。例如，如果忘记了停止其他 WebSphere 服务器（这是最有可能出现的问题），那么将在 stdout 日志文件中见到下列错误消息：



```

C:\>f:
F:\>cd websphere4
F:\WebSphere4>cd app*
F:\WebSphere4\AppServer>cd bin
F:\WebSphere4\AppServer\bin>startserver

IBM WebSphere Application Server
Application Server Launcher
Copyright (C) IBM Corporation, 2001

The configuration file was defaulted to:
f:\WebSphere4\AppServer\config\server-cfg.xml
Using the single available node or the localhost node.
Using the single available server.
Initiating server launch.
Loaded domain "WebSphere Administrative Domain".
Selected node "ron2k".
Selected server "Default Server".
The server launch was successfully initiated.
To test if the server launch completed successfully, examine
the logs configured for the server which was launched. The
configured log files are:
Standard Output: "f:\WebSphere4\AppServer\logs\default_server_stdout.log"
Standard Error: "f:\WebSphere4\AppServer\logs\default_server_stderr.log"

F:\WebSphere4\AppServer\bin>

```

图 2-19 使用 startserver.bat 启动服务器

IBM WebSphere Application Server, Release 4.0.0
Advanced Single Server Edition for Multiplatforms
Copyright IBM Corp., 1997-2001

```

[01.11.04 02:20:28:581 EST] 566df55e Server
A WSVR0027I: Temporary product license expires on Dec 30, 2001
***** Start Display Current Environment *****
WebSphere AES 4.0.0 q0124.08
running with process name Default Server and process id 4176
Host Operating System is Windows 2000, version 5.0
Java version = J2RE 1.3.0 IBM build cn130-20010502w (JIT enabled: jitc), Java
Compiler = jitc
server.root = f:\WebSphere4\AppServer
Java Home = f:\WebSphere4\AppServer\java\jre
ws.ext.dirs = f:\WebSphere4\AppServer\java\lib;f:\WebSphere4\AppServer
/classes;f:\WebSphere4\AppServer\lib;f:\WebSphere4\AppServer\lib/ext;f:\
WebSphere4\AppServer\web\help;f:\WebSphere4\AppServer\properties
Classpath = f:\WebSphere4\AppServer\lib\bootstrap.jar;f:\WebSphere4\
AppServer\properties
Java Library path = f:\WebSphere4\AppServer\java\bin;.;C:\WINNT\System32;
C:\WINNT;f:\WebSphere4\AppServer\bin;f:\WebSphere4\AppServer\java\bin;f:
\WebSphere4\AppServer\java\jre\bin;f:\Program Files\ibm\gsk5\lib;f:\Perl
\bin;f:\Program Files\IBM\GSK\lib;f:\IBMDebug\bin;f:\IBMCON~1\Encina\bin;

```

```

f:\IBMJava\hpj\bin;f:\IBMJava\eam\bin;F:\oracle\ora81\bin;
F:\oracle\ora81\Apache\Perl\5.00503\bin\mswin32-x86;C:\Program Files\
Oracle\jre\1.1.7\bin;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\Wbem;f
:\IMNnq_NT;f:\IBM
Connectors\CICS\BIN;f:\jdk1.2.2\bin;f:\jdk1.2.2\jre\bin;f:\jdk1.2.2\jre\
bin\classic;f:\WebSphere\AppServer\bin;f:\JDK11~1.8\bin;F:\DOCUSH~1\bin;
f:\
WebSphere4\AppServer\bin;f:\jdk1.3\bin;f:\jdk1.3\jre\bin;f:\jdk1.3\jre\b
-in\
classic;f:\TIBCO\TIBRV\BIN;f:\vitria\bw31\bin\win32
Current trace specification =
***** End Display Current Environment *****
[01.11.04 02:20:34:600 EST] 566df55e Server      U Version : 4.0.0
[01.11.04 02:20:34:690 EST] 566df55e Server      U Edition: AES
[01.11.04 02:20:34:820 EST] 566df55e Server      U Build date: Wed Jun 20
00:00:00 EDT 2001
[01.11.04 02:20:34:920 EST] 566df55e Server      U Build number: q0124.08
[01.11.04 02:20:51:524 EST] 566df55e NameServer  F NMSV0011E: Unable to
start Bootstrap Server. The most likely cause is that the bootstrap port is
already in use. Please ensure that no servers or other processes are already
using the bootstrap server port.
org.omg.CORBA.INTERNAL:  minor code: 8  completed: No
    at com.ibm.CORBA.iiop.GIOPImpl.createListener(GIOPImpl.java:333)
    at com.ibm.CORBA.iiop.GIOPImpl.getEndpoint(GIOPImpl.java:461)
    at com.ibm.CORBA.iiop.GIOPImpl.getEndpoint(GIOPImpl.java:524)
    at com.ibm.rmi.iiop.GIOPImpl.getBootstrapEndpoint(GIOPImpl.java:226)
    at
com.ibm.ejs.ns.CosNaming.BootstrapServer.start(BootstrapServer.java:147)
    at
com.ibm.ws.naming.bootstrap.NameServer.startBootstrapService(NameServer.
java:361)
    at com.ibm.ws.naming.bootstrap.NameServer.doInit(NameServer.java:252)
    at
com.ibm.ws.naming.bootstrap.NameServer.initialize(NameServer.java:104)
    at com.ibm.ws.runtime.Server.initializeNameService(Server.java:1433)
    at com.ibm.ws.runtime.Server.initializeRuntime0(Server.java:791)
    at
com.ibm.ws.runtime.StandardServer.initializeRuntime0(StandardServer.java
:356)
    at com.ibm.ws.runtime.Server.initializeRuntime(Server.java:728)
    at com.ibm.ws.runtime.StandardServer.main(StandardServer.java:522)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.ibm.ws.bootstrap.WSLauncher.main(WSLauncher.java:63)

```

```
java.lang.Throwable
    at com.ibm.ejs.ras.TraceLogger.fatalEvent(TraceLogger.java:219)
    at
com.ibm.ejs.ras.TraceEventGeneratorImpl.fireTraceEvent(Tr.java:1331)
    at
com.ibm.ejs.ras.TraceEventGeneratorImpl.fireTraceEvent(Tr.java:1275)
    at com.ibm.ejs.ras.Tr.fatal(Tr.java:845)
    at
com.ibm.ws.naming.bootstrap.NameServer.startBootstrapService(NameServer.
java:365)
    at com.ibm.ws.naming.bootstrap.NameServer.doInit(NameServer.java:252)
    at
com.ibm.ws.naming.bootstrap.NameServer.initialize(NameServer.java:104)
    at com.ibm.ws.runtime.Server.initializeNameService(Server.java:1433)
    at com.ibm.ws.runtime.Server.initializeRuntime0(Server.java:791)
    at
com.ibm.ws.runtime.StandardServer.initializeRuntime0(StandardServer.java
:356)
    at com.ibm.ws.runtime.Server.initializeRuntime(Server.java:728)
    at com.ibm.ws.runtime.StandardServer.main(StandardServer.java:522)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.ibm.ws.bootstrap.WSLauncher.main(WSLauncher.java:63)
[01.11.04 02:20:51:935 EST] 566df55e SystemOut      U NMSV0011E: Unable to
start Bootstrap Server. The most likely cause is that the bootstrap port is
already in use. Please ensure that no servers or other processes are already
using the bootstrap server port.
org.omg.CORBA.INTERNAL:  minor code: 8  completed: No
    at com.ibm.CORBA.iiop.GIOPImpl.createListener(GIOPImpl.java:333)
    at com.ibm.CORBA.iiop.GIOPImpl.getEndpoint(GIOPImpl.java:461)
    at com.ibm.CORBA.iiop.GIOPImpl.getEndpoint(GIOPImpl.java:524)
    at com.ibm.rmi.iiop.GIOPImpl.getBootstrapEndpoint(GIOPImpl.java:226)
    at
com.ibm.ejs.ns.CosNaming.BootstrapServer.start(BootstrapServer.java:147)
    at
com.ibm.ws.naming.bootstrap.NameServer.startBootstrapService(NameServer.
java:361)
    at com.ibm.ws.naming.bootstrap.NameServer.doInit(NameServer.java:252)
    at
com.ibm.ws.naming.bootstrap.NameServer.initialize(NameServer.java:104)
    at com.ibm.ws.runtime.Server.initializeNameService(Server.java:1433)
    at com.ibm.ws.runtime.Server.initializeRuntime0(Server.java:791)
    at
com.ibm.ws.runtime.StandardServer.initializeRuntime0(StandardServer.java
:356)
```

```

at com.ibm.ws.runtime.Server.initializeRuntime(Server.java:728)
at com.ibm.ws.runtime.StandardServer.main(StandardServer.java:522)
at java.lang.reflect.Method.invoke(Native Method)
at com.ibm.ws.bootstrap.WSLauncher.main(WSLauncher.java:63)

```

此外，用户还将在 logs 目录中见到一个文件名类似于默认的 ServerFatalError100485845-1524.log 的文件，此文件具有如下形式的错误消息：

```

***** Start Display Current Environment *****
WebSphere AEs 4.0.0 q0124.08 running with process name Default Server and
process id 4176
Host Operating System is Windows 2000, version 5.0
Java version = J2RE 1.3.0 IBM build cn130-20010502w (JIT enabled: jitc), Java
Compiler = jitc
server.root = f:\WebSphere4\AppServer
Java Home = f:\WebSphere4\AppServer\java\jre
ws.ext.dirs = f:\WebSphere4\AppServer\java\lib;f:\WebSphere4\AppServer
/classes;f:\
WebSphere4\AppServer\lib;f:\WebSphere4\AppServer\lib\ext;f:\WebSphere4\
AppServer\web\help;f:\WebSphere4\AppServer\properties
Classpath = f:\WebSphere4\AppServer\lib\bootstrap.jar;f:\WebSphere4\
AppServer\properties
Java Library path = f:\WebSphere4\AppServer\java\bin;.;C:\WINNT\System32;
C:\WINNT;f:\WebSphere4\AppServer\bin;f:\WebSphere4\AppServer\java\bin;f:
\WebSphere4\AppServer\java\jre\bin;f:\Program
Files\ibm\gsk5\lib;f:\Perl\bin;f:\Program
Files\IBM\GSK\lib;f:\IBMDebug\bin;f:\IBMCON~1\Encina\bin;f:\IBMVJava\hpj
\bin;f:\IBMVJava\eam\bin;F:\oracle\ora81\bin;F:\oracle\ora81\Apache\Perl
\5.00503\bin\mswin32-x86;C:\Program
Files\Oracle\jre\1.1.7\bin;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\
Wbem;f:\IMNnq_NT;f:\IBM
Connectors\CICS\BIN;f:\jdk1.2.2\bin;f:\jdk1.2.2\jre\bin;f:\jdk1.2.2\jre\
bin\classic;f:\WebSphere\AppServer\bin;f:\JDK11~1.8\bin;F:\DOCUSH~1\bin;
f:\
WebSphere4\AppServer\bin;f:\jdk1.3\bin;f:\jdk1.3\jre\bin;f:\jdk1.3\jre\b
-in\
classic;f:\TIBCO\TIBRV\BIN;f:\vitria\bw31\bin\win32
Current trace specification =
***** End Display Current Environment *****
[01.11.04 02:20:28:581 EST] 566df55e Server      A WSVR0027I: Temporary
product license expires on Dec 30, 2001
[01.11.04 02:20:34:600 EST] 566df55e Server      U Version : 4.0.0
[01.11.04 02:20:34:690 EST] 566df55e Server      U Edition: AEs
[01.11.04 02:20:34:820 EST] 566df55e Server      U Build date: Wed Jun 20

```

```

00:00:00 EDT 2001
[01.11.04 02:20:34:920 EST] 566df55e Server      U Build number: q0124.08
[01.11.04 02:20:51:524 EST] 566df55e NameServer  F NMSV0011E: Unable to
start Bootstrap Server. The most likely cause is that the bootstrap port is
already in use. Please ensure that no servers or other processes are already
using the bootstrap server port.
org.omg.CORBA.INTERNAL:  minor code: 8  completed: No
    at com.ibm.CORBA.iiop.GIOPImpl.createListener(GIOPImpl.java:333)
    at com.ibm.CORBA.iiop.GIOPImpl.getEndpoint(GIOPImpl.java:461)
    at com.ibm.CORBA.iiop.GIOPImpl.getEndpoint(GIOPImpl.java:524)
    at com.ibm.rmi.iiop.GIOPImpl.getBootstrapEndpoint(GIOPImpl.java:226)
    at
com.ibm.ejs.ns.CosNaming.BootstrapServer.start(BootstrapServer.java:147)
    at
com.ibm.ws.naming.bootstrap.NameServer.startBootstrapService(NameServer.
java:361)
    at com.ibm.ws.naming.bootstrap.NameServer.doInit(NameServer.java:252)
    at
com.ibm.ws.naming.bootstrap.NameServer.initialize(NameServer.java:104)
    at com.ibm.ws.runtime.Server.initializeNameService(Server.java:1433)
    at com.ibm.ws.runtime.Server.initializeRuntime0(Server.java:791)
    at
com.ibm.ws.runtime.StandardServer.initializeRuntime0(StandardServer.java
:356)
    at com.ibm.ws.runtime.Server.initializeRuntime(Server.java:728)
    at com.ibm.ws.runtime.StandardServer.main(StandardServer.java:522)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.ibm.ws.bootstrap.WSLauncher.main(WSLauncher.java:63)

```

在停止了老的 WebSphere 服务器之后，用户可能需要等待直到套接字被释放，然后就能够成功启动 WebSphere 4.0 服务器。在启动之后，将会见到下列日志文件。需要查找的句子是“Default Server open for e-business”，在本例中就是所显示的最后一行。

```

IBM WebSphere Application Server, Release 4.0.0
Advanced Single Server Edition for Multiplatforms
Copyright IBM Corp., 1997-2001

[01.11.04 02:45:26:078 EST] 566e784c Server      A WSVR0027I: Temporary
product license expires on Dec 30, 2001
***** Start Display Current Environment *****
WebSphere AEs 4.0.0 q0124.08 running with process name Default Server and
process id 3104
Host Operating System is Windows 2000, version 5.0
Java version = J2RE 1.3.0 IBM build cn130-20010502w (JIT enabled: jitc), Java

```

```

Compiler = jitc
server.root = f:\WebSphere4\AppServer
Java Home = f:\WebSphere4\AppServer\java\jre
ws.ext.dirs = f:\WebSphere4\AppServer\java\lib;f:\WebSphere4\AppServer
/classes;f:\
WebSphere4\AppServer\lib;f:\WebSphere4\AppServer\lib\ext;f:\WebSphere4\
AppServer\web\help;f:\WebSphere4\AppServer\properties
Classpath = f:\WebSphere4\AppServer\lib\bootstrap.jar;f:\WebSphere4\
AppServer\properties
Java Library path = f:\WebSphere4\AppServer\java\bin;.;C:\WINNT\System32;
C:\WINNT;f:\WebSphere4\AppServer\bin;f:\WebSphere4\AppServer\java\bin;f:
\WebSphere4\AppServer\java\jre\bin;f:\Program
Files\ibm\gsk5\lib;f:\Perl\bin;f:\Program
Files\IBM\GSK\lib;f:\IBMDebug\bin;f:\IBMCON~1\Encina\bin;f:\IBMJava\hpj
\bin;f:\IBMJava\eab\bin;F:\oracle\ora81\bin;F:\oracle\ora81\Apache\Perl
\5.00503\bin\mswin32-x86;C:\Program
Files\Oracle\jre\1.1.7\bin;C:\WINNT\system32;C:\WINNT;C:\WINNT\System32\
Wbem;f:\IMNnq_NT;f:\IBM
Connectors\CICS\BIN;f:\jdk1.2.2\bin;f:\jdk1.2.2\jre\bin;f:\jdk1.2.2\jre\
bin\classic;f:\WebSphere\AppServer\bin;f:\JDK11~1.8\bin;F:\DOCUSH~1\bin;
f:\
WebSphere4\AppServer\bin;f:\jdk1.3\bin;f:\jdk1.3\jre\bin;f:\jdk1.3\jre\b
-in\
classic;f:\TIBCO\TIBRV\BIN;f:\vitria\bw31\bin\win32
Current trace specification =
***** End Display Current Environment *****
[01.11.04 02:45:27:720 EST] 566e784c Server      U Version : 4.0.0
[01.11.04 02:45:27:720 EST] 566e784c Server      U Edition: AEs
[01.11.04 02:45:27:720 EST] 566e784c Server      U Build date: Wed Jun 20
00:00:00 EDT 2001
[01.11.04 02:45:27:720 EST] 566e784c Server      U Build number: q0124.08
[01.11.04 02:45:33:038 EST] 566e784c DrAdminServer I WSVR0053I: DrAdmin
available on port 7070
[01.11.04 02:45:33:428 EST] 566e784c ResourceBinde I WSVR0049I: Binding
Session Persistence datasource as jdbc/Session
[01.11.04 02:45:33:588 EST] 566e784c ResourceBinde I WSVR0049I: Binding
Increment Bean Datasource as jdbc/SampleDataSource
[01.11.04 02:45:33:659 EST] 566e784c ResourceBinde I WSVR0049I: Binding
Samples IDB Datasource as jdbc/sample
[01.11.04 02:45:38:165 EST] 566e784c EJBEngine    I WSVR0037I: Starting EJB
jar: Increment Bean Jar
[01.11.04 02:45:39:237 EST] 566e784c SystemOut    U Enhydra InstantDB -
Version 3.26
[01.11.04 02:45:39:247 EST] 566e784c SystemOut    U

```

```
[01.11.04 02:45:39:247 EST] 566e784c SystemOut      U The Initial Developer
of the Original Code is Lutris Technologies Inc.
Portions created by Lutris are Copyright (C) 1997-2001 Lutris Technologies,
Inc.
All Rights Reserved.
[01.11.04 02:45:41:860 EST] 566e784c EJBEngine      I WSVR0037I: Starting EJB
jar: f:\WebSphere4\AppServer\installedApps\Samples.ear/TimeoutEJBBean.jar
[01.11.04 02:45:43:092 EST] 566e784c EJBEngine      I WSVR0037I: Starting EJB
jar: Increment EJB Module
[01.11.04 02:45:44:584 EST] 566e784c EJBEngine      I WSVR0037I: Starting EJB
jar: Hello EJB Module
[01.11.04 02:45:45:496 EST] 566e784c EJBEngine      I WSVR0037I: Starting EJB
jar: Account And Transfer EJB Module
[01.11.04 02:45:46:908 EST] 566e784c ServletEngine A SRVE0161I: IBM WebSphere
Application Server - Web Container. Copyright IBM Corp. 1998-2001
[01.11.04 02:45:46:958 EST] 566e784c ServletEngine A SRVE0162I: Servlet
Specification Level: 2.2
[01.11.04 02:45:46:968 EST] 566e784c ServletEngine A SRVE0163I: Supported
JSP Specification Level: 1.1
[01.11.04 02:45:47:148 EST] 566e784c ServletEngine A Session Manager is
Configured - Initializing...
[01.11.04 02:45:47:789 EST] 566e784c CacheManager A DYNA0011E: Servlet cache
file dynacache.xml not found; caching is disabled
[01.11.04 02:45:47:909 EST] 566e784c ServletEngine A Loading Web Module:
Default Application
[01.11.04 02:45:48:300 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[01.11.04 02:45:48:380 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
InvokerServlet: init
[01.11.04 02:45:48:850 EST] 566e784c ServletEngine A Loading Web Module:
Examples Application
[01.11.04 02:45:48:971 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[01.11.04 02:45:48:991 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
SimpleFileServlet: init
[01.11.04 02:45:49:511 EST] 566e784c ServletEngine A Loading Web Module: admin
[01.11.04 02:45:49:952 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[01.11.04 02:45:49:982 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
SimpleFileServlet: init
[01.11.04 02:45:50:112 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
action: init
[01.11.04 02:45:51:154 EST] 566e784c ServletEngine A Loading Web Module:
YourCo Web Application
```



```
[01.11.04 02:45:51:424 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[01.11.04 02:45:51:434 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
SimpleFileServlet: init
[01.11.04 02:45:51:444 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
InvokerServlet: init
[01.11.04 02:45:51:544 EST] 566e784c ServletEngine A Loading Web Module:
Samples Web Application
[01.11.04 02:45:51:614 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[01.11.04 02:45:51:634 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
SimpleFileServlet: init
[01.11.04 02:45:51:644 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
InvokerServlet: init
[01.11.04 02:45:51:694 EST] 566e784c ServletEngine A Loading Web Module: theme
Web Application
[01.11.04 02:45:51:765 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[01.11.04 02:45:51:845 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
SimpleFileServlet: init
[01.11.04 02:45:51:855 EST] 566e784c WebGroup      I SRVE0091I: [Servlet LOG]:
InvokerServlet: init
[01.11.04 02:45:51:955 EST] 566e784c HttpTransport A Transport http is
listening on port 9080
[01.11.04 02:45:55:220 EST] 566e784c HttpTransport A Transport https is
listening on port 9443
[01.11.04 02:45:55:230 EST] 566e784c HttpTransport A Transport http is
listening on port 9090
[01.11.04 02:45:55:240 EST] 566e784c Server        I WSVR0023I: Server
Default Server open for e-business
```

2.6 小 结

本章描述了一个典型的 WebSphere 安装过程，这个安装过程在单独的一台机器上安装了所有的组件——包括 Web 服务器、应用程序服务器以及管理服务器。在一个产品安装中，这些组件有可能不是安装在一台机器上。目前，为了使用 WebSphere 并且开始构建应用程序，进行典型安装就足够了。在安装了 Web 服务器和应用程序服务器，并且对安装进行测试以确保安装的组件能够正常工作之后，下面就可以准备编写“Hello World”这样的程序了，我们将在第 3、4、5 章给出这些程序的编写方法。

第 3 章 Hello World (Wide Web)

本章将开始对 WebSphere 的探索。第一站是 WebSphere Java Application Server 底层的 Web 服务器。正如将在第 6 章中见到的那样，WebSphere 能够在许多 Web 服务器上使用，例如 Apache Server、IBM 的 HTTP Server、Microsoft 的 IIS (Internet Information Server)、Sun 的 iPlanet 以及 Domino 服务器。本章将关注与 WebSphere 一同发布的 IHS (IBM HTTP Server)。由于启动 HTTP Server 来查看文件是一项很简单的工作，因此本章除此之外还将介绍更多的内容；在这里还将介绍安全服务器——一种使用 SSL (Secure Socket Layer, 安全套接字层) 协议形成的安全连接来传递 Web 网页的服务器，将在第 13 章介绍 SSL 协议。

3.1 HTTP

在开始讨论 Web 上的应用程序之前，首先必须理解 Web 使用的底层协议与传递机制。所有传递给最终用户的浏览器的 Web 网页都是一个完整的请求/响应周期的结果，在这个周期中浏览器与完成这些请求的 Web 服务器通信。Web 服务器与浏览器在相互识别对方并且传递数据（以请求和响应的方式进行传递）的过程中所采用的通信协议被称为 HTTP (Hypertext Transfer Protocol, 超文本传输协议)，HTTP 是 Web 的基础。

由于在 Web 上使用的所有语言、脚本机制以及应用程序都被毫无例外地打包成 HTTP 规范所定义的消息，因此，这一协议的属性将对任何通过 Web 传递的信息施加某种限制与特点。在开始设计将要通过 Web 部署的应用程序与服务之前，理解这些限制是非常重要的。

HTTP 是一种应用程序级协议，它最初是为通过 Internet 传递超文本元素而设计的，自 1990 年以来，HTTP 被用作 Web 协议。虽然 HTTP 现在并非仅仅用于 HTML (Hypertext Markup Language, 超文本标记语言) 页面，但它作为 IP (Internet Protocol) 的特点仍然有效。它们包含有为分布式协作超媒体信息系统提供基础的必要的灵活性与速度。HTTP 是一种无状态 (state-less) 协议，这意味着每个请求-响应会话都是独立的，并且不包含任何关于先前会话的历史信息。无状态协议比有状态协议速度更快，因为需要传递的信息更少。无状态协议也相对比较简单并且容易实现，这样，也就简化了支持无状态协议的软件的复杂性。最后，HTTP 依赖于数据表示的协商，这样就使得能够完全独立于新数据表示方法的开发来构建系统。

HTTP 基于请求/响应范型。一旦服务器提供了响应，相应的事务就结束，这样就确保能够实现无状态操作。一个 HTTP 事务包括以下阶段：

- Connection (连接) : 在这一阶段, 建立从客户到服务器的连接。
- Request (请求) : 在这一阶段, 客户向服务器发送一条请求消息。
- Response (响应) : 在这一阶段, 服务器向客户发送响应。
- Close (关闭) : 服务器关闭连接。

客户请求包括一个方法、一个 URI (Uniform Resource Identifier, 统一资源标识符)、一个协议版本、一条 MIME (Multipurpose Internet Mail Extension, 多用途 Internet 邮件扩展) 消息。服务器的响应包括协议版本、状态代码以及一条 MIME 消息。

HTTP 版本既是请求消息中的一部分, 也是响应消息中的一部分, 其格式如下:

<major version number>.<minor version number> (<主版本号>.<次版本号>)

当协议的修改并没有改动消息解析算法时, 就使用不同的次版本号。消息格式的修改则需要改变主版本号。这个版本号总是作为 HTTP 版本中第一行的一部分。例如, 从服务器向客户发送的响应消息可能以下列行开头:

HTTP/1.0 200 OK

这一行的含义是, 这个响应的协议版本是 1.0, 并且将返回状态 200 (成功完成)。

HTTP 定义的标准字符集是 ISO-8859-1。由于这个字符集是 US-ASCII 的超集, 因此它对使用 7 位字符集定义的消息没有任何影响。在内容类型头部中出现的 charset 参数将覆盖默认的定义, 这一参数被消息发起者使用以指定将要使用的字符集。

3.1.1 URI, URL 与 URN

URI 是用来标识 Internet 资源的字符串。URI 严格地与 Web 上一个对象的 Internet 地址等价。URN (Uniform Resource Name, 统一资源名称) 也用来标识 Internet 上的资源, 但它们通过使用一个独立于位置的名称来实现。另一方面, URL (Uniform Resource Locator, 统一资源定位符) 基于资源的位置, 使用一种方法来标识 Internet 上的资源。

URI, URN 以及 URL 具有类似的目标, 因此可以互用。URI 最抽象并且最概括; 因此, 它们通常用于技术规范之中。例如, URI 的语法定义可以作为 URN 与 URL 语法定义的基础。URN 最为通用, 但它们依赖于命名查找服务, 因此, 也就依赖于并非总是可用的额外服务。URL 最为常用, 并且是 HTTP 与 Web 软件的基础。

URL 的语法非常简单。由于 URL 通过位置来引用资源, 因此它们指定了主机以及在主机上能够发现目标资源的路径名。此外, URL 也可以定义连接到主机上的端口以及连接所使用的协议。因此, URL 具有以下形式:

<protocol> : "://" <host> [:<port>] "/" <absolute path> (<协议> : "://" <主机> [:<端口>] "/" <绝对路径>)

文档可能会使用相对 URL 或虚拟 URL 来引用资源。这些 URL 并不指定协议或主机, 它们仅提供相对路径。这是 Web 服务器提供的一个方便, 用以基于使用此虚拟 URL 的资源的

相对位置来引用资源。这些 URL 对于以一种一致的方式组织大量资源从而能够方便地引用这些资源（例如，能够作为一个单独的元素移动这些资源）是非常重要的。

3.1.2 HTTP 事务

一个 HTTP 事务的“来回”包括客户向服务器发送请求、服务器进行处理以满足此请求、服务器向客户发送响应。HTTP 是一种无状态协议。因此，每个这样的事务过程是完全自治的，在两个连续的事务之间没有任何关系。因此，HTTP 事务是大量的工作并且包括对消息（请求与响应）的处理，在每条消息中都包含有头部（header）与主体（body）。

3.1.3 HTTP 消息

HTTP 消息就是一个通过连接传递的八位字节组流。HTTP 消息是 HTTP 事务的基本组件。从客户向服务器发送的每条请求以及从服务器向客户发送的每条响应都是 HTTP 消息。请求与响应可以很简单，也可能很完全；因此，在 HTTP 中有 4 种消息类型：简单请求、简单响应、完全请求、完全响应。简单请求与简单响应区别于完全请求与完全响应的地方在于，前者不允许使用任何头部信息。

消息头部中包含有一系列定义，用于描述消息体并且提供与 HTTP 事务相关的信息。在一个头部中包含具有以下形式的一系列行：

`<name> ":" <value> (<名称>":"<值>)`

请求

请求是从客户发送给服务器的 HTTP 消息。请求指定了在服务器上应用的方法、这个方法将要应用于的资源、正在被使用的协议版本。一个典型的请求行具有以下形式：

`GET http://www.entity93.com/docRoot.html HTTP/1.0`

上述形式表示，在 URL 上的资源应该被获取并且在服务器的响应中返回。目前，HTTP 定义了下列方法：

- DELETE：这个方法用来删除所提供的 URL 上的资源。
- GET：这个方法用来获取 URL 所指定的资源。如果此资源是一个能够产生数据的程序，那么在响应中将包含由此程序所产生的输出，而不是包含程序本身。
- HEAD：这个方法所执行的动作等价于 GET 方法，惟一的区别在于，当使用此方法时，服务器所返回的数据体为空。对于头部、元信息等，这两个方法是完全等价的。这个方法对于测试资源的存在性而不必获取此资源非常有用。
- POST：在这个方法中，客户将在消息体中发送额外的信息。当服务器为此请求提供服务时，服务器将使用这些额外的信息。这个方法得名于客户需要发送（posting）信息这一事实。
- PUT：这个方法将在指定的 URL 上创建包含所提供数据的资源。

最常用的方法是 GET, POST 以及 HEAD。实际上,并非所有的 HTTP 服务器都一定实现了除这三者之外的其他方法。

此外,关于请求的 HTTP 定义还允许放置任何令牌 (token) 作为一个方法。这样就允许进行动态方法扩展。当消息到达服务器上时,服务器将试图在资源上应用此方法。如果不支持该方法,服务器将返回一个包含适当错误代码的响应。因此,在一个指定服务器上关于指定资源的可接受方法的列表是动态改变的。

响应

服务器的工作完成之后,将要返回给客户的信息就已经准备好了,这时,服务器就可以开始构建响应。HTTP 定义了响应的格式应该包括状态行、头部以及消息体。状态行记录了协议版本、数字状态代码、状态的文本短语描述。一个状态代码具有 3 个十进制数字。HTTP 将状态代码与状态短语描述分为 5 类。每个类分别以一个不同的数字开头:

- Informational: 目前没有使用,编号在 100~199 之间。
- Success: 当请求的动作被应用时,返回这个类中的代码,编号在 200~299 之间。例如,状态代码 200 表示 OK,说明所请求的资源在响应中返回(例如,当请求应用 GET 方法时)。
- Redirection: 当必须采取进一步的措施来完成请求时,返回这个类中的代码,编号在 300~399 之间。例如,如果一个资源已经发生了迁移,并且服务器方程序知道新的 URL,那么服务器将可能会向客户返回一个重定向响应,从而使得浏览器导航到新的 URL。
- Client error: 当在请求中发现语法错误或结构错误时,返回这个类中的代码,编号在 400~499 之间。当一个错误的客户状态到达服务器上时,如果客户还没完成请求的发送动作,应该立刻停止该请求的传输。服务器通常会包含一个关于此错误的简要说明,作为响应的一部分。
- Server error: 当出现一个服务器失败时,返回这个类中的代码,编号在 500~599 之间。

消息实体

当发送了一条请求并且构建了一条响应时,消息的内容将被打包成一个实体。实际包含元信息,元信息由头部以及一个称为主体的内容段组成。实体头部类似于其他头部,表现在它们都包括名称与值。此外,一个特殊的头部定义了主体所使用的格式,这个特殊的头部被称为 Entity-Body header field。

当信息在客户与服务器之间传输时,主体将包含在消息中。这样,在 POST 请求中包含有一个主体,而在 HEAD 以及 GET 请求中不包含主体。响应通常包括一个主体,这是因为它们需要返回某些信息。HEAD 请求的响应以及返回状态代码为 204 与 304 的响应不包含主体。

如果在一条消息中包含有一个主体,那么就需要有许多元信息段。首先,需要指定数据类

型与编码方法。例如，主体可能是一个使用 `gzip` 压缩之后的 HTML 文本。下列形式的完全规范，为消息接收者定义了应该如何从主体中提取数据：

```
Content-Encoding ( Content-Type( <entity data> ) ) (内容编码 (内容类型 (<实体数据>)))
```

这个信息是作为头部字段被包含的。主体的长度通常在另一个额外的头部字段中指定。这种处理方法对于基于 EOF (end-of-file, 文件结尾) 的通信方法是很好的，因为一个通信信道可能会由于某种原因被关闭。对于请求，内容长度是惟一可能有变化的部分；因为如果客户将要关闭连接，服务器将没有任何办法返回响应。

3.2 HTTP 服务器

Web 服务器就是一个在操作系统上注册的服务，该服务默认地侦听机器上的 80 端口，从而为 HTTP 请求提供服务。当启动了一个 Web 服务器时，它将定义一系列从其中接受连接的端口号。这些端口在服务器的配置文件（在本书中就是文件 `httpd.conf`）中定义。不同的 Web 服务器可以有不同的配置文件与方法。IHS (IBM HTTP Server) 基于 Apache Web 服务器，在默认的 WebSphere 安装中提供的 `httpd.conf` 文件的相应段如下：

```
ServerName webspHEREbook.
# This is the main server configuration file.
# See URL http://www.apache.org/ for instructions.

...

# ServerRoot: The directory the server's config, error, and log files
# are kept in

ServerRoot "f:/IHS1_3_19"

...

# Port: The port the standalone listens to.
Port 80

...

# Apache always creates one child process to handle requests. If it
# dies another child process is created automatically. Within the
# child process multiple threads handle incoming requests. The next two
# directives determine the behaviour of the threads and processes.
```

```
# Don't force a server to exit after it has served some number of requests.
# If you do want server's to exit after they have run for a long time (to
# help the system clean up after the process), please set this to a pretty
# large number - like 10,000. What this will do, is, each child server will
# exit after serving 10,000 requests, and another server will take its place.
```

```
MaxRequestsPerChild 0
```

```
# Number of concurrent threads at a time (set the value to more or less
# depending on the responsiveness you want and the resources you wish
# this server to consume).
```

```
ThreadsPerChild 50
```

```
...
```

```
# This should be changed to whatever you set DocumentRoot to.
```

```
<Directory "f:/IHS1_3_19/htdocs">
```

```
# This may also be "None", "All", or any combination of "Indexes",
# "Includes", "ExecCGI", or "MultiViews".
```

```
# Note that "MultiViews" must be named *explicitly* --- "Options All"
# doesn't give it to you.
```

```
Options Indexes
```

```
# This controls which options the .htaccess files in directories can
# override. Can also be "All", or any combination of "Options", "FileInfo",
# "AuthConfig", and "Limit"
```

```
AllowOverride None
```

```
# Controls who can get stuff from this server.
```

```
order allow,deny
allow from all
```

```
</Directory>
```

```
# f:/IHS1_3_19/cgi-bin should be changed to whatever your ScriptAliased
# CGI directory exists, if you have that configured.
```

```
Directory "f:/IHS1_3_19/cgi-bin"
AllowOverride None
Options None
</Directory>

...

# DocumentRoot: The directory out of which you will serve your
# documents. By default, all requests are taken from this directory, but
# symbolic links and aliases may be used to point to other locations.

DocumentRoot "f:/IHS1_3_19/htdocs"

...

LoadModule ibm_app_server_http_module f:/WebSphere4/AppServer/bin/mod
_ibm_app_server_http.dll
Alias /IBMWebAS/ "f:/WebSphere4/AppServer/web/"
Alias /Wssamples "f:/WebSphere4/AppServer/Wssamples/"
WebSpherePluginConfig f:\WebSphere4\AppServer\config\plugin-cfg.xml
```

我们将在本书中详细分析这个文件中的一些伪指令。在 `httpd.conf` 文件中定义的服务器名应该被外部世界所引用的机器的主机名。通常就是安装有 Web 服务器的机器的主机名，但并非总是如此。在大多数情况下，可能存在一个防火墙、路由器或存在一个用来进行网络地址转换（以便通过域名与主机名使用应用程序）的节点，就不一定是安装有 Web 服务器的域名与主机名。我们将在第 41 章进一步分析这个问题。

`ServerRoot` 伪指令定义了安装有 Web 服务器的文件系统根目录。`Port` 伪指令定义了 Web 服务器在其中打开套接字并且侦听输入请求的端口。`DocumentRoot` 伪指令定义了服务器从其中获取信息的文件系统目录。它与 `Alias` 伪指令一道，共同定义了到达 Web 服务器的请求能够访问其中的文档的文件系统根。当在 Web 服务器所侦听的某个端口上建立了一个连接时，操作系统将把请求转交给 Web 服务器。最简单的请求就是访问 Web 服务器根页面的请求（这个请求只要有主机名，而不需提供文档名）。在这种情况下，服务器将使用其配置文件中定义的默认页面进行回答；通常，这个默认页面就是名为 `index.html` 的文件，此文件放置在文档的根目录下。IHS 所提供的回答如图 3-1 所示。

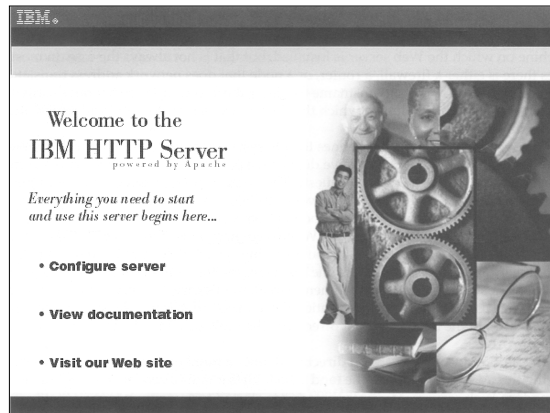


图 3-1 默认的 Web 服务器页面

`MaxRequestsPerChild` 伪指令定义了请求的数目，在超过这个数目之后，提供服务的线程将消亡。Apache (IHS 也一样) 是一个多线程 Web 服务器，正如所有现代的 Web 服务器一样。在服务器中启动了多个线程，用来为页面提供服务，从而确保能够获取高吞吐率。使用 `MaxRequestsPerChild` 伪指令似乎是不合时宜的——实际上在许多情况下确实如此。使用这个伪指令的惟一原因就是软件有时存在 bug (读者可能已经知道这一点)。实际上，有些平台存在众所周知的内存泄漏现象；例如，如果一个线程在 Web 服务器的整个生命周期中都始终活跃，它将有可能最终占据主机的所有内存以至于使得主机无法工作。例如，这种行为在 Solaris 平台上就存在。为了克服这个问题，这条伪指令将能够确保如果为它设立了一个非 0 的正数，线程最终将被杀死，并且重新启动另一个线程，从而使得内存泄漏也被重置。例如，作为经验，当在 Solaris 上运行时，这个值应该设置为 5000。

`ThreadsPerChild` 用来控制同时能够处于活跃状态的线程的数目。当然，这里存在一个这些并发线程的响应性与 HTTP 服务器所用内存量之间的权衡。`ThreadsPerChild` 是一个特别值得重视的参数，并且是用户在进行产品系统优化时所使用的最重要参数之一。实际上，`ThreadsPerChild` 是一个用来控制应用程序服务器应该被如何使用的监视参数。当线程开始在 WebSphere Web 容器和连接池中等待时，系统性能将急剧下降。一种优化方法是确保没有过多的线程在 Web 服务器中产生。通过使用系统的多个克隆，常常能够达到增加吞吐率的目的。

在进行系统优化时，另一个重要参数就是 `MaxClients` 的值——特别是在 AIX 以及 Solaris 上。`MaxClients` 伪指令用来限制 Web 服务器能够服务的并发请求的数量。`MaxClients` 参数的值可能会极大影响应用程序，特别是在这个值的设置过高时更是如此。过犹不及！最佳的值取决于应用程序。`MaxClients` 是一个当用户对一个配置进行性能优化时所修改的 HTTP 参数。它确实是应用程序服务器的“调节阀门”。这个参数的值越低，应用程序被要求完成的工作就越少。这样导致的后果就是，有些试图访问此应用程序的人可能会被拒绝。

Web 服务器是一个由操作系统管理的进程。根据操作系统的不同，Web 服务器在操作系统启动过程中被启动的顺序也可能不一样。例如，在 Solaris 中，Web 服务器的启动可能在一个 rc 文件中完成，而在 Windows NT 或 Windows 2000 下，它将被定义为操作系统的服务。图 3-2 显示了 Windows 2000 Service Manager，用来演示服务器在其中运行的服务。Web 服务器的最初使用目的是提供存储在 Web 服务器的文件系统中的页面；但在本书后面的章中，我们将描述 Web 服务器在构建 Web 应用程序方面的许多其他用法。Web 服务器是通向 WebSphere Application Server 的入口。

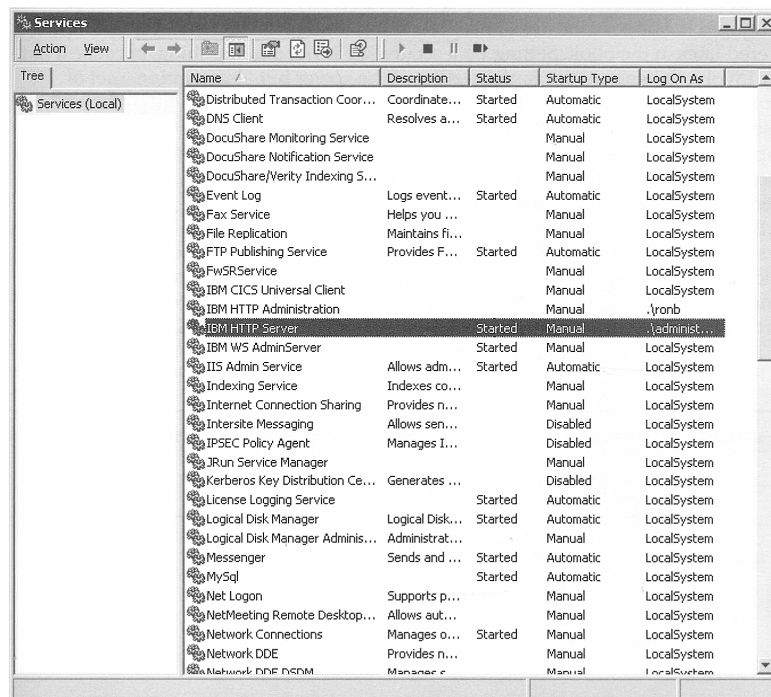


图 3-2 Web 服务器作为 Windows 2000 中的一个服务

3.3 建立一个安全网站

虽然 Web 服务器对于在 Intranet 内部使用以及将普通信息传递给 Internet 上的用户非常有用，一旦用户在 Web 站点上开始传送敏感信息，或者希望传送一些真正的应用程序，Web 服务器（由于其固有的开放性）的使用就可能存在隐患。但幸运的是，大多数 Web 服务器都可以被配置为支持使用 SSL 协议（本书将在后面讨论 SSL 协议）的安全连接。建立一个安全网站并不容易，因为它不仅需要修改服务器的配置，而且需要创建一系列密钥以及一个用于服务器身份认证的方法。第 13 章将详细讨论所有这些概念，但在这里有必要向读者介绍

使用 IHS 建立安全服务器的简单过程。当我们完成这些过程之后，将有一个侦听 80 端口的常规服务器以及一个侦听 443 端口（SSL 的默认端口）的安全服务器。

在开始建立安全服务器之前，需要产生一对密钥。这些密钥用来对会话进行加密，并且是基于公共密钥加密法进行的，公共密钥加密法是所有现代安全性协议的基础。这些密钥可以使用 Key Management 实用程序 IKeyMan 来生成。通过选择 Start|Programs|IBM WebSphere|Application Server V4.0|IKeyMan 来启动这个实用程序。

打开后的 Key Management 应用程序如图 3-3 所示。

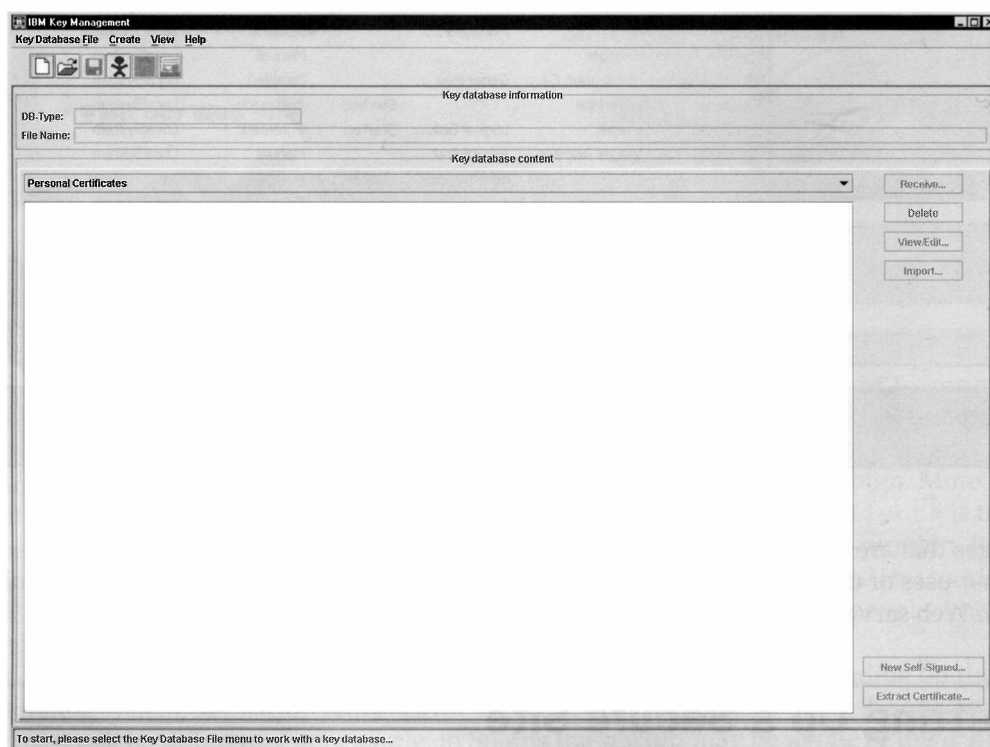
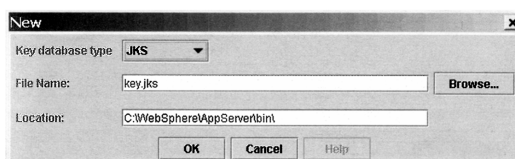


图 3-3 Key Management 应用程序窗口

选择 Key Database File|New，以创建一个新的数据库文件。这时，将打开一个对话框，如下所示：



输入文件名与路径，然后单击 OK。下一步，在图 3-4 所示的窗口中输入口令（请将此口令

记录在一个较安全的地方，以防忘记！)。将这个口令隐藏在一个 .sth 文件中。需要注意的是，这个对话框通过高亮显示的钥匙的数量指出了所提供口令的相对强度。为了改善这个口令，请试着在其中包含数字并且避免使用英语单词。单击 OK 以完成这个过程。这时，Key Management 窗口应该类似于图 3-5。

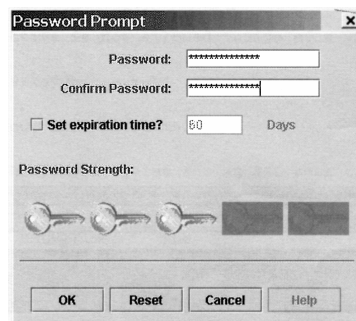


图 3-4 为密钥数据库设置口令

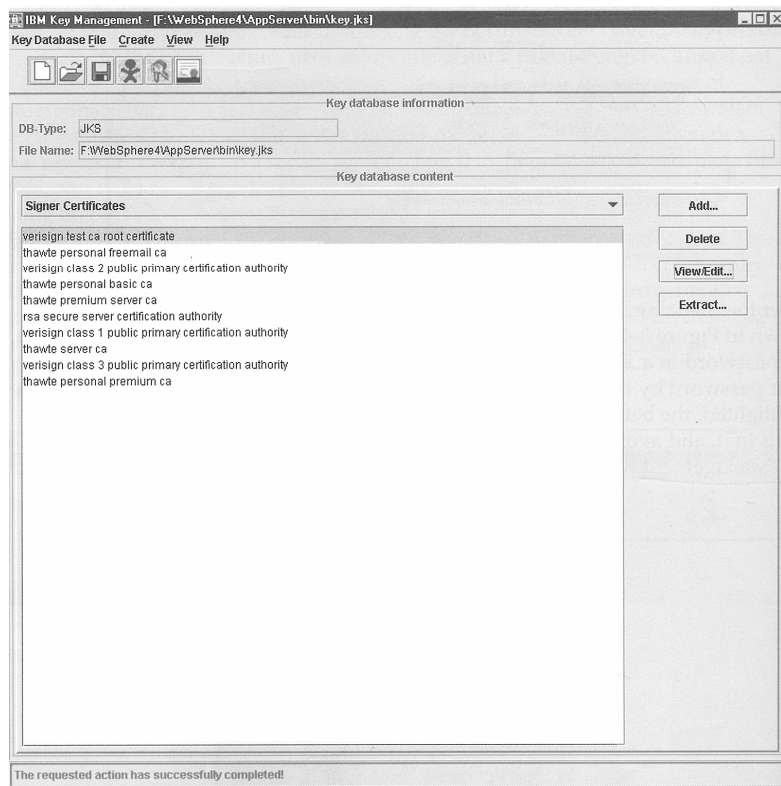


图 3-5 密钥数据库文件的视图

下一步，就需要更新服务器的配置文件（就是服务器的 `conf` 目录下的 `httpd.conf` 文件）。为此，应该在这个配置文件中添加一些伪指令，如下所示：

```
# Load SSL module
# This can be 40,56,128 bit encryption
# For NT
# LoadModule ibm_ssl_module modules/IBModuleSSL128.dll
# For UNIX
# LoadModule ibm_ssl_module      libexec/mod_ibm_ssl128.so
LoadModule ibm_ssl_module modules/IBModuleSSL128.dll

Listen 443
# A virtual host must correspond to a different IP address,
# different port number or a different host
# name for the server, in the latter case the server machine must be
# configured to accept IP packets for multiple addresses.
<VirtualHost websphere.webspherebook.com:443>
# Enter the Document "root"
DocumentRoot "/IHS1_3_19/htdocs"
ErrorLog logs/error-ssl.log
ServerAdmin ronb@webspherebook.com
SSLEnable
SSLClientAuth none
</VirtualHost>
SSLDisable

##      Keyfile directive:
##
##      Specify the names of key files that are available, as defined
##      with the IKEYMAN utility.
##
##      Default: <none>
##      Syntax:  Keyfile <filename.kdb>
##      This directive is not allowed inside of a virtual host stanza

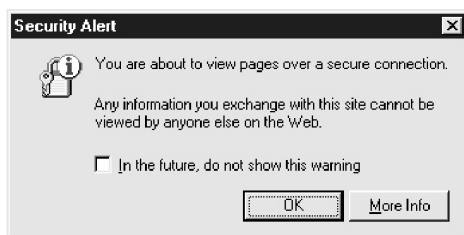
Keyfile "/IHS1_3_19/keydb/key.kdb"

##      SSLV2Timeout and SSLV3Timeout:
##
##      Specify the timeout value for an SSL session. Once the timeout
##      expires, the client is forced to perform another SSL handshake.
##
##      Default:  SSLV2Timeout 100
##                SSLV3Timeout 1000
##      Syntax:   SSLV2Timeout <time in seconds> range 1-100
```

```
##                      SSLV3Timeout <time in seconds>  range 1-86400

SSLV2Timeout 100
SSLV3Timeout 1000
```

在这些配置中包含有：DLL 文件，它们用来实现安全性功能；安全服务器将要侦听的端口（443）；其他许多与 SSL 连接的属性相关的伪指令。一旦更新了配置文件，就需要重新启动 Web 服务器。如果由于某些原因使得该服务器启动失败，用户应该仔细检查对配置文件所作的修改；错误的设置值是导致服务器启动失败的最常见原因。但是，如果重新启动服务器，用户将会发现安全服务器与普通服务器存在不同。例如，如果现在使用 `http://<服务器名称>` 访问这个服务器，用户将会见到一个 Security Alert 对话框，如下所示：



在接收到确认之后，服务器就运行于一个安全连接之上。现在可以结束此过程，去获取一些信用卡号了！

3.4 小 结

本章描述了 HTTP 服务器（Web 服务器）的作用，并且展示了如何建立 Web 服务器以便在非安全连接与安全连接上提供页面服务。Web 服务器在上述整个讨论中都是一个非常重要的组件，因为它是从客户（浏览器）到运行于 WebSphere Application Server 上应用程序组件的入口。实际上，许多体系结构都采用 Web 服务器，并且通过添加各种功能对其进行扩展。一个例子就是 Apache Tomcat，它使用 Apache Web 服务器并且对其添加 servlet 与 JSP 支持。在 WebSphere 解决方案中，打包是不同的——Web 服务器总是管理请求/响应事务，但并不进行其他任何处理（除非文档是静态的）。WebSphere 中的这种处理大多数要么是在 EJB 容器中完成，要么是在 Web 容器中完成。我们将继续介绍这两个主题——在第 4 章中介绍 servlet 与 JSP，在第 5 章中介绍 EJB。

第 4 章 Hello Servlet 与 JSP

大多数 WebSphere 部署都是关于 Web 应用程序的。大多数 Web 应用程序都以 HTML 作为前端并且有一个基于服务器的事务系统。在一个 WebSphere 上开发与部署的 J2EE 应用程序中，后端由一系列技术组成。其中，WebSphere Application Server 支持的、常用来创建将要传递给浏览器的输出的两个主要功能是 servlet 与 JSP。正如其标题所指出的那样，本章将介绍简单“Hello World”程序的实现过程。这一程序的目标是向读者展示关于 servlet 与 JSP 开发的一些简单经历。虽然这些例子不是非常原始，也不是很复杂，但它们将让读者了解到 WebSphere 中的 servlet 与 JSP 是什么。本书中的第 5 部分将集中关注 servlet 与 JSP 的开发及部署，并且在这一部分将提供一些更复杂的例子。

4.1 servlet 基础

在 Web 出现的早期，人们就已经清楚地认识到只有静态 Web 页面是不够的，为了在 Web 上通过一个会话执行某些计算，就必须运行程序。这样，CGI（Common Gateway Interface，公共网关接口）就诞生了，将程序赋给输入到 Web 服务器上的请求。CGI 脚本存在两个主要的缺点。其一是该脚本在服务器之外作为一个单独的进程运行，并且必须为每个请求创建一个这样的进程；其二是该脚本可以用许多不同的脚本语言（例如 Perl、Tcl、Python）来编码。这些脚本语言对于系统管理脚本以及“轻便的”功能非常优秀，但如果将它们应用于非常严肃的应用程序则不太合适，并且它们的可移植性比 Java 差。

为了克服上述缺点，人们推出了 servlet。这样，由于 servlet 作为一个单独的 Java 线程运行，并且在相同的 JVM（Java Virtual Machine，Java 虚拟机）中作为 servlet 服务器，因此解决了上面提到的第一个问题。这种做法比为每个请求创建一个操作系统进程为该请求服务更加高效，因为它需要更少的内存、不需要进行上下文切换并且所需的启动延时更少。

此外，由于 servlet 是用 Java 语言编写的，并且能够调用其他 Java 类——有时调用另一个技术类型（例如 EJB）——来执行复杂的应用程序处理工作（例如数据库访问、复杂算法、文档管理等）。因此第二个问题也得到了妥善解决。

由于 CGI 非常常见，并且由于 Java 作为一种语言对于这样的开发工作非常合适，两者的共同作用使得 servlet 这一概念迅速得到认可并且继续成为 Web 应用程序领域中比较成功的模型之一。目前，市场上关于各种 Java 应用程序服务器的最一致的元素就是 servlet 支持。它们包括 IBM WebSphere，BEA WebLogic，Oracle Application Server，iPlanet（就是以前的 Netscape Server），Apache/Tomcat 等。

在理解了使用 servlet 的动机之后,下面将继续说明 servlet 究竟是什么。servlet 就是一个 Java 代码段,在创建此代码段时遵守 JavaSoft 的 Servlet API 标准。这个 API 标准定义了应该被将要作为 servlet 部署的类所实现的方法集合。显然,这里说的意思也就是 servlet 类应该遵守某些类型的接口或规范,但读者也许会问,Java 不是一直如此吗?事实上,Servlet API 并不仅仅是一个接口集合;它还定义了一种作为 servlet 生命周期中基础的计算模型。

servlet 模型假设 servlet 代码段将被一个遵守 Servlet API 规范的服务器所维护与管理。这个 servlet 服务器通常与一个 Web 服务器(或 HTTP 服务器)协同工作;这种协同是最常见的配置方式,因为大多数 servlet 都是 HTTP servlet,并且它们通过 HTTP 连接被访问(无论是使用浏览器页面还是其他代码段)。这种与 Web 服务器的协同在 WebSphere 中同样存在;WebSphere 安装就包括 Web 服务器(iPlanet 服务器、Microsoft IIS 服务器、Apache 服务器或作为 WebSphere 默认安装一部分的 IBM HTTP Server)以及一个用来管理 servlet 生命周期的服务器。Web 服务器典型情况下都将把请求转发给 servlet 服务器,然后由 servlet 服务器管理并调用在这个服务器上已经注册的服务器。

现在已经有了一个能够从 Web 服务器调用的 servlet 服务器,那么,下一步应该干什么呢?从本质上说,现在的工作实际上已经足够了。简单地说,servlet 就是 Java 代码,它作为能够通过 servlet 服务器调用的服务器方功能的入口点。目前,最常见的 servlet 是 HTTP servlet。一个 HTTP servlet 就是实现了一些入口点方法的 Java 类,通过这些入口点方法就能够启动服务器方处理过程。在有些情况下,servlet 实现了一个非常简单的操作,在这个操作中它将执行某一任务并且返回;在另一些情况下,它仅仅是一个入口点,并且大多数工作都是由 servlet 所调用的类来完成的。由于 servlet 的实例是在自己的线程中被调用,因此 servlet 模型具有很好的可扩展性,使得 servlet 被 Web 应用程序广泛使用。

servlet 模型之所以具有强大的功能,是因为 servlet 为开发人员提供了一个非常直接的计算模型。在接收到一条请求(对于 HTTP servlet,就是 HTTP 请求)之后,就应该产生一个响应。由服务器来处理实际计算与传输的复杂事情。

4.2 “Hello World” servlet

在本章中将要构建与部署的 servlet 可能是最简单的 servlet。这个 servlet 将放置在 Web 容器中,等待一个 HTTP 请求(在本例中是 GET 请求)。在这一请求到达之后,它将编写一个小的 HTML 页面并且将“Hello World”放置在这个页面中。为了使例子更为有趣,我们将为这个 servlet 添加对法语的支持,从而使该 servlet 能够输出“Hello World!!”或“Bonjour tout le monde!!”。

4.2.1 编写 servlet 代码

由于将要通过一个 HTTP 连接从浏览器上访问 servlet 功能，因此，应该将这个 servlet 实现为 HTTP servlet。这就意味着需要为此 servlet 从 `javax.servlet.http.HttpServlet` 类继承一个子类。在本书后面将描述这个类并且提供关于其用法的详细信息，但在这里，需要关心的重要事情是这个类存在一系列当 HTTP 请求到达时被调用的方法。特别地，它有一个当 HTTP GET 请求到达时所调用的 `doGet` 方法以及一个当 HTTP POST 请求到达时所调用的 `doPost` 方法。或者，也可以实现 `service` 方法，当以上两种类型的请求到达时调用此方法（如果没有实现 `do` 方法）。

“Hello World” servlet 的代码如下：

```
package book.websphere.chapter4;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class HelloServlet1 extends HttpServlet {
    Locale usEn;
    ResourceBundle usEnglishMessages;
    Locale frFr;
    ResourceBundle franceFrenchMessages;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        Locale usEn = Locale.US;
        usEnglishMessages = ResourceBundle.getBundle(
            "HelloServletsTextMessages", usEn);
        Locale frFr = Locale.FRANCE;
        franceFrenchMessages = ResourceBundle.getBundle(
            "HelloServletsTextMessages", frFr);
    }

    public void doGet(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");

        ResourceBundle aBundle;
        if (request.getHeader("Accept-Language").compareTo("FR") == 0 ) {
```

```
        aBundle = franceFrenchMessages;
        response.setHeader("Content-Language", "FR");
    }
    else {
        aBundle = usEnglishMessages;
        response.setHeader("Content-Language", "EN");
    }

    PrintWriter out = new PrintWriter (response.getOutputStream());
    out.println("<html>");
    out.println("<head><title>HelloServlets1</title></head>");
    out.println("<body>");
    out.println(aBundle.getString("helloWorld"));
    out.println("</body>");
    out.println("</html>");
    out.close();
}

public String getServletInfo() {
    return "This is a simple servlet activated by an HTTP GET request";
}
}
```

我们在这里感兴趣的主要是 `doGet` 方法以及 `init` 方法。当一个 HTTP GET 方法到达引用此 servlet 的服务器上时，将调用 `doGet` 方法。当 servlet 代码被首次加载到 servlet 服务器中时，将调用 `init` 方法。在这个代码示例中，`init` 方法被用来初始化 Hello World 消息所对应的两个字符串资源，而 `doGet` 方法则根据客户的位置信息输出适当的 Hello World 消息。初始化过程就是要读取在这个简单示例中所使用的两个字符串资源。两个字符串资源的表示方法如下所示，在服务器启动并运行时，它们将维护在内存中以供使用。

```
#Property file for HelloServlets1 text messages
#en_US
helloWorld=Hello World!!

#Property file for HelloServlets1 text messages
#fr_FR
helloWorld=Bonjour tout le monde!!
```

在接收到 HTTP GET 请求时，这个处理方法将激活。基于请求的 URL 并且假设它与所讨论的 servlet 相对应，那么应用程序服务器将调用这个 servlet。由于这样的 URL 是在 GET 请求中形成的，因此将调用 HelloServlet1 servlet 的 `doGet` 方法。这个调用所使用的参数是一个封装了请求参数的 `HttpServletRequest` 以及一个封装了响应所对应对象的 `HttpServletResponse`，其中最重要的参数是结果输出应该被写入的输出流。

这个处理方法进行的第一项工作就是设置响应流的内容类型。在这里，内容类型就是 MIME 类型，它定义了将要提供的输出类型。由于现在的浏览器和其他设备基于 HTML，XML，WML 以及本书后面将要讨论的许多其他格式，因此需要告诉请求者将要提供的输出类型是什么。在本章的这个 servlet 代码中，应该将内容类型设置为 HTML，从而告知请求者输出将以文本的形式书写并且此文本是按照 HTML 格式进行编码的。这就意味着，程序代码在后面将要写入到输出流中的文本是 HTML 格式。

代码需要做的另一项工作是检查请求对象，以决定浏览器所使用的语言。这一工作是通过访问请求中的一个头部来完成的。我们将在第 7 章详细讨论请求头部与请求对象，在这里将主要考虑 servlet 可以使用什么类型的信息。实际上，读者已经在第 2 章的讨论中看见过 WebSphere 发布中所提供的一个 servlet。应该还记得在图 2-13 中显示的 snoop servlet。只要打开 `http://localhost/servlet/snoop` 这个 URL，就能够调用该 servlet。通过观察这个 servlet 的输出，可以发现其中包含有许多关于请求客户的信息。servlet 不仅知道诸如请求方法、浏览器能够接受的语言以及浏览器能够接受的格式这些事情，它也知道客户正在使用的浏览器类型（用户代理）以及客户的主机名与 IP 地址。虽然 HTTP 与 servlet 模型需要如此（例如，输出可能被设置为对用户代理敏感），但作为一个 Web 用户，也应该意识到这些信息已经被传递给服务器。

现在还是返回到 Hello World 示例，请求的输出应该使用何种语言这一决定是基于请求对象中 Content-Language 的值。在本例中，仅仅能够支持法语和英语；除了法语之外，其他语言将显示为英语。在创建了实际的输出之后，输出将被写入到作为响应对象一部分的 `PrintWriter` 对象中。`PrintWriter` 对象封装了输出流，并且写入到这个输出流上的任何信息将被放置在 HTTP 响应体中。由于已经声明了应该将输出书写为 HTML 格式，因此，后面的工作就是给出一些 `println` 语句以便把将要由浏览器显示的 HTML 数据源输出。最为重要的是，应该从前面所选择的字符串资源中提取出正确的“Hello World”字符串。

4.2.2 装配 Web 应用程序

现在，已经在自己所喜爱的 Java IDE（Integrated Development Environment，集成开发环境）——WebSphere Studio Application Developer，WebSphere Studio Site Developer 或 VisualAge for Java——中编写了 servlet 代码，下一步就是在 WebSphere 下部署 servlet。这时，已经有一个 class 文件以及两个 properties 文件。为了使 servlet 在服务器上运行，就需要部署此 servlet；但在此之前，还需要基于这个 servlet 装配出一个应用程序。装配阶段的目标是提供与应用程序及其组件相关的各种参数。

首先，应该产生一个包含上述 class 文件以及两个资源文件的 Java Archive 文件。在这些文件所在的目录中使用下列命令（下面的代码应该在一行中给出）：

```
jar cf HelloServlet.jar HelloServlet1.class
HelloServletsTextMessages_fr_FR.properties
```

HelloServletsTextMessages_en_US.properties

在得到 JAR 文件之后，就可以开始启动 WebSphere Application Assembly Tool。可以从 First Steps 窗口（见第 2 章）启动这个工具，或者从 Windows 开始菜单中启动 WebSphere。或者，可以在 WebSphere Application Server bin 目录中运行 assembly.bat 脚本，从而启动 WebSphere。在加载 Assembly Tool 时，将显示图 4-1 所示的窗口。

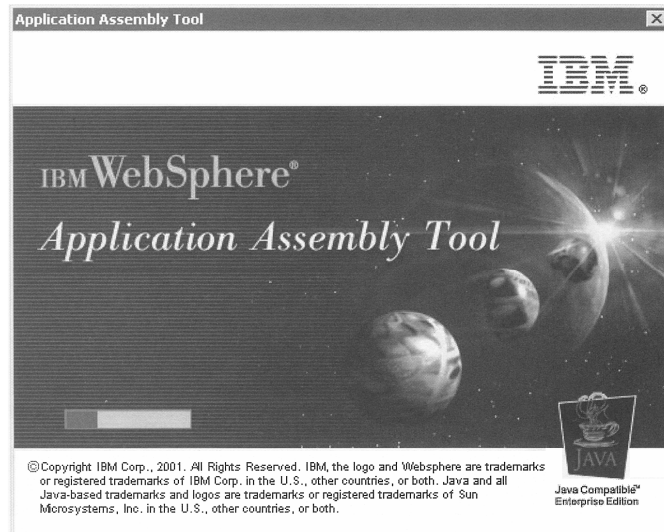


图 4-1 Application Assembly Tool 加载窗口

在工具加载之后，将见到一个 Welcome 窗口。在这个窗口中，请选择 Web Module，这时将出现图 4-2 所示的窗口。在这个窗口中，左边部分给出的是以层次方式组织的关于 Web 模块的组件，而在右边给出的是关于所选择对象的详细描述。在这个 Web 模块中的第一个文件夹是 Web Components。Web 组件可以是 servlet 或 JSP。为了添加新的 servlet，请右击 Web Components 表项，然后选择 New。

在所出现的对话框中，输入关于这个新 servlet 的信息。首先，单击 Browse 以选择这个 servlet 的 class 文件。这时，将打开 class 选择窗口。在这个窗口中，再次单击 Browse。这时将出现一个新的对话框；请选择先前准备好的 JAR 文件。在下一个窗口中，选择 servlet class 文件并单击 OK。

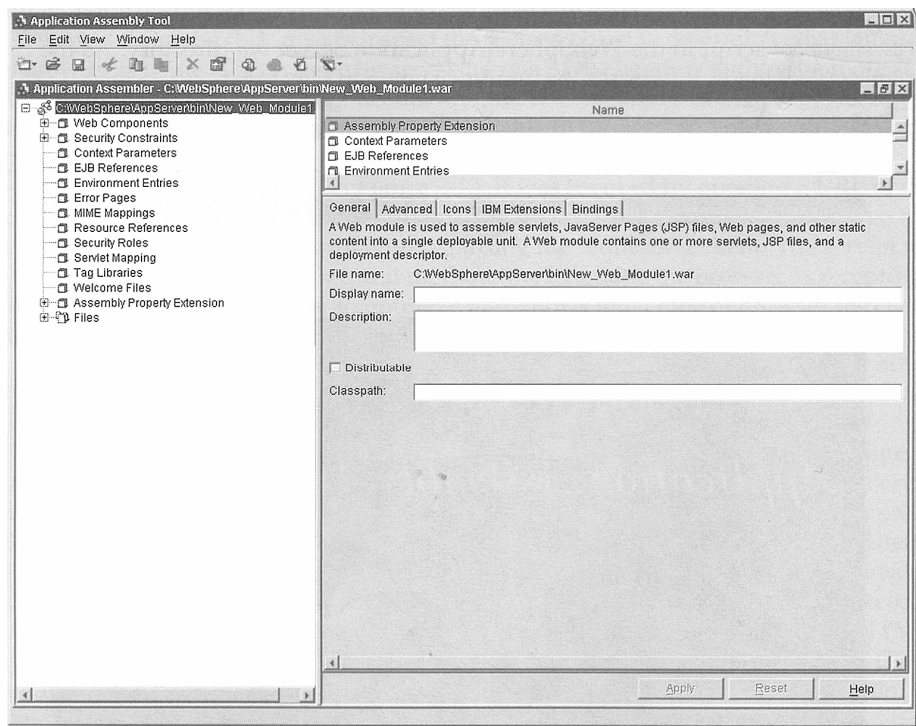


图 4-2 新 Web 模块

这时，将返回到 Web 组件对话框，请输入名称、显示名称、描述并且选中 Load on Startup 复选框。结果如图 4-3 所示。单击 OK。

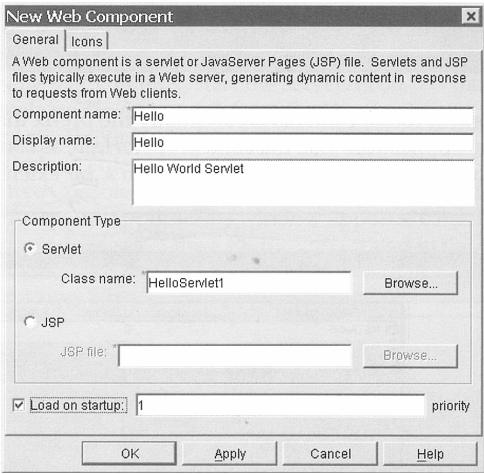


图 4-3 填写好了的 New Web Component 对话框

下一步，右击 Application Assembly Tool 左边的 Servlet Mapping 图标并选择 New。在 New Servlet Mapping 对话框中，利用 URL 模式/Invoke 定义新的映射方法，以及新的 servlet class，如图 4-4 所示。单击 OK。

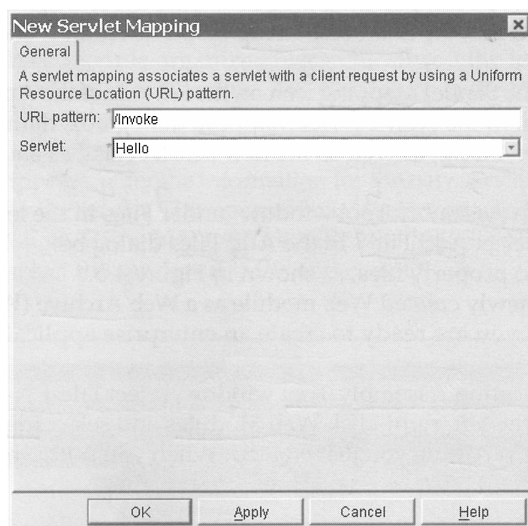


图 4-4 servlet 映射对话框

在 Application Assembly Tool 窗口左边窗格的 Files 下，右击 JAR Files 然后选择 Add Files。在出现的 Add Files 对话框中，选择已经准备好的 JAR 文件以及两个属性文件，如图 4-5 所示。

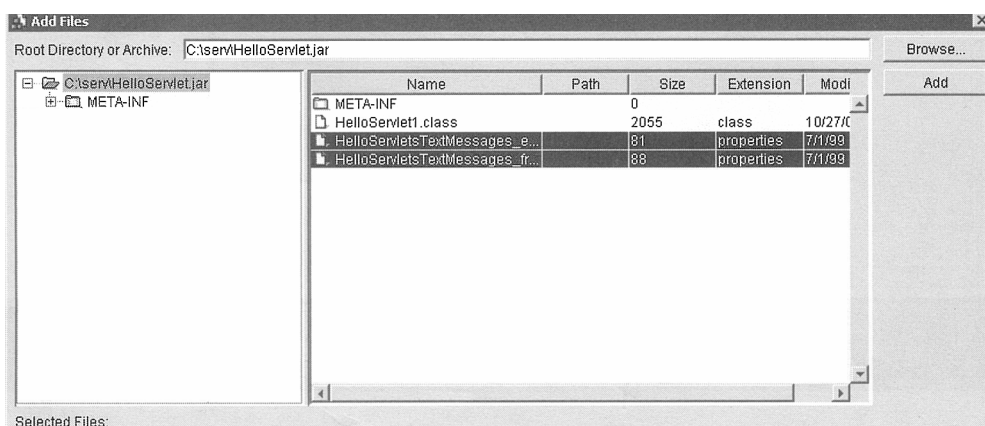


图 4-5 添加资源文件

最后，将新创建的 Web 模块保存为一个 WAR (Web Archive) 文件，其扩展名为 .war。这时，

就已经完成创建包含新模块的企业应用程序的准备工作了。

返回到 Application Assembly Tool 窗口，选择 File|New|Application。在左边的树形窗格中，右击 Web Modules 并选择 Import。在 Open 对话框中，选择刚刚创建的 WAR 文件。在选择 Web 模块时，需要为其选择上下文根。它在本质上就是将要被用于访问此应用程序的 URL 的前缀。如图 4-6 所示，使用上下文根/Hello。这时，就准备好了将应用程序保存为带有.ear 扩展名的文件。

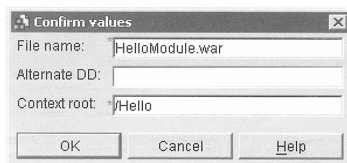


图 4-6 选择上下文根

4.2.3 部署 Web 应用程序

现在，可以开始在服务器上部署应用程序（包含前面的 servlet）。为了完成这项工作，需要调用 Administrative Console。需要说明的是，WebSphere Application Server 高级完全版中所使用的 Administrative Console 与 WebSphere Application Server 单服务器版中的有所不同。下面的例子使用的是单服务器配置中的 Administrative Console。

可以从 Windows 开始菜单启动 Administrative Console，或者在运行 WebSphere 服务器的机器上访问 <http://localhost:9090/admin> 以启动 Administrative Console。需要注意的是，必须启动服务器（可以按照第 2 章中的方法，从 First Steps 窗口启动服务器、从 Windows 开始菜单中启动服务器，或者使用 startserver.bat 脚本启动）。图 4-7 中显示了 Administrative Console 的第一个窗口。在这个窗口中，输入用户名并单击 Submit。

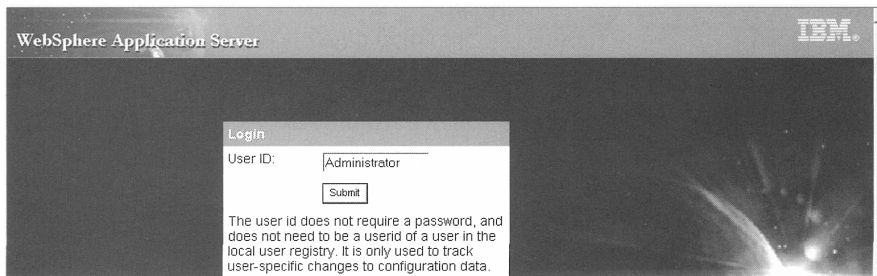


图 4-7 Administrative Console Login 输出显示

这时，就打开了 Administrative Console。在左边的树结构中，选择 Enterprise Applications。这时，将出现 Enterprise Applications 窗口。单击 Install；然后，在下一个窗口的 Path 字段，

选择在上一节中创建的 EAR 文件, 如图 4-8 所示。单击 Next, 从而进入到第二个安装窗口。再次单击 Next, 从而进入到部署过程的最后一个窗口, 如图 4-9 所示。单击 Finish。

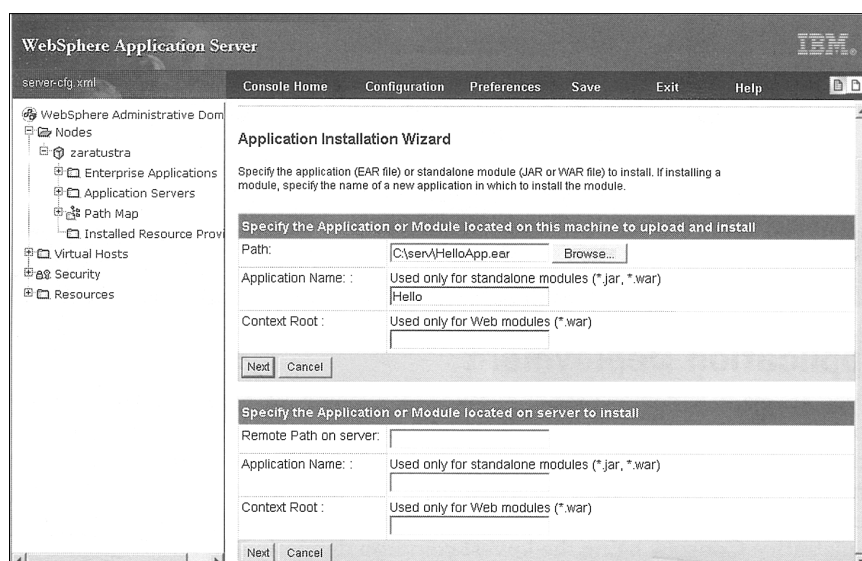


图 4-8 第一个应用程序安装窗口

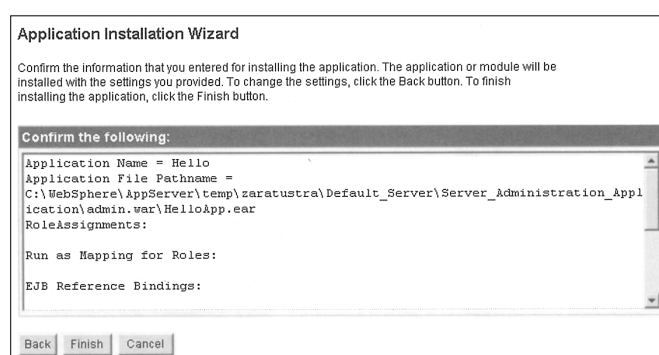


图 4-9 最后出现的应用程序安装窗口

为了完成应用程序的部署, 需要单击 Administrative Console 顶部的 Save, 从而保存设置。此外, 部署过程还要求停止 WebSphere Application Server 并重新启动它。这一过程可以从 First Steps 窗口中完成, 或者使用 sopserver.bat 脚本完成。

4.2.4 测试 servlet

在完成了部署 servlet 所需的全部必要工作之后, 测试 servlet 就比较简单了。只要将 Web 浏

浏览器指向正确的 URL 即可。基于所选择的上下文根以及 Servlet Mapping，所需使用的 URL 应该是 `http://localhost/Hello/Invoke` 这种形式。所得到的结果如图 4-10 所示。

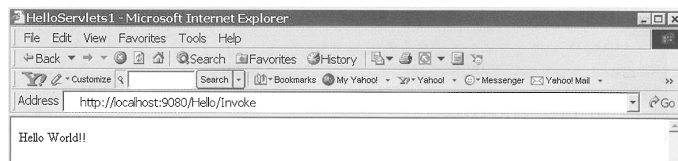


图 4-10 Hello World servlet 的输出

4.3 “Hello World” JSP

读者现在已经了解了关于 servlet 的基本知识，并且学习了如何构建与部署一个简单的 servlet，下面将要学习的是非常简单的 JSP。JSP 采用了一种比较新的 Java 编程模型；但这个模型并不是计算领域的新模型，因为它非常类似于 Microsoft 非常成功的 ASP (Active Server Page) 模型。JSP 的目标是提供与 servlet 相同的有用性，但它使用的是一种将应用程序的处理逻辑完全分离于应用程序表示逻辑的方法。它的另一个目标是提供一个环境，使得在此环境下可以非常容易地构建相对简单的电子商务应用程序，同时又不损害 servlet 模型所具有的强大功能。此处不打算深入讨论 JSP 以及 JSP 模型的细节，因为本书后面将包含这些内容。目前关心的是，实际地构建一个非常简单的示例并且在 WebSphere 服务器上部署它。实际上，要讨论的是 4 个例子，分别演示了稍有不同的各种 JSP 变体。

下面的代码清单显示了 Hello World JSP。它将输出“Hello World”并显示一个“hit”计数器——用来说明自从服务器启动以来，该页面被访问的次数。

```
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<H1>Hello World</H1>
<%= static int count = 0; %>
<H2>This servlet was accessed
<%= ++count %>
times so far. </H2>
<H3>The date is <%= new java.util.Date()%> </H3>
</BODY>
</HTML>
```

这个代码清单是一个普通的 HTML 文件，在其中嵌入了几个 JSP 标记。JSP 标记由 `<%` 和 `%>` 表示。第一个 JSP 标记定义了静态变量 `count`。注意标记是以 `<%!` 开始，当定义变量时，使

用这种标记。另一个 JSP 标记以`<%=`开始，用来定义输出表达式。前两个输出是计数器以及一个 Date 对象。当使用`<%=`标记时，表达式的 `toString()`方法将被调用并且将此表达式的结果输出到页面中。

为了部署这个例子，必须经过一个装配阶段。JSP 的装配阶段类似于 servlet；惟一的不同是，必须在 Web Component Properties 对话框中选择 JSP 单选按钮，如图 4-11 所示。一旦拥有在其中包含 JSP 的 Web 应用程序之后，在 WebSphere 中部署 Web 应用程序的过程就与本章 4.2.2 节“装配 Web 应用程序”中描述的完全相同。

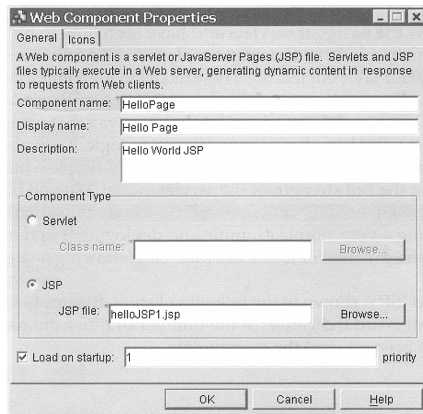


图 4-11 JSP 的 Web Component Properties 对话框

在部署了包含有 JSP 的应用程序（并且重新启动 WebSphere）之后，就可以测试这个 JSP，如图 4-12 所示。

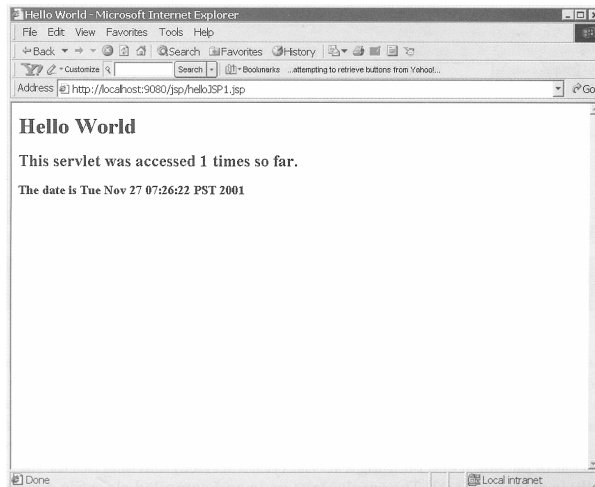


图 4-12 Hello World JSP 的输出

输出页面的 HTML 如下所示：

```
<HTML>
<HEAD>
<TITLE>Hello World</TITLE>
</HEAD>
<BODY>
<H1>Hello World</H1>
<H2>This servlet was accessed
1
times so far. </H2>
<H3>The date is Tue Nov 27 07:26:22 PST 2001 </H3>
</BODY>
</HTML>
```

可以看到，这是 JSP 页面的一个副本，并且使用 JSP 标记各自的输出替换了这些 JSP 标记。重新加载将会增加计数器值，如图 4-13 所示。

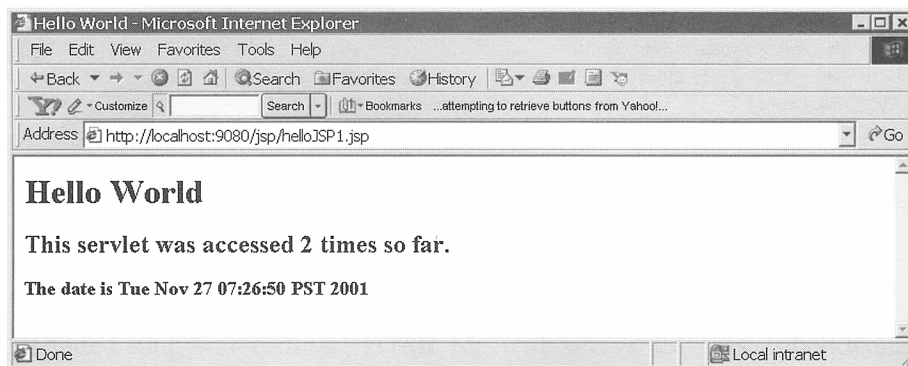


图 4-13 重新加载页面之后的 Hello World JSP 输出

那么，在 JSP 内部究竟发生了什么呢？当 Web 服务器得到一个以 .jsp 为扩展名的 URL 请求时，它将调用 WebSphere Application Server。然后，实际的处理工作就开始了！WebSphere 的 JSP 支持将读取这个 JSP 文件，并且对这个文件进行分析以及构建一个与此 JSP 代码匹配的 servlet（确实是 servlet，没有错！）。所得到的 servlet Java 代码以及编译后的 class 文件将作为 servlet 在服务器上被部署。这个过程仅仅执行一次；而后续的请求将直接导致对已编译 servlet 的快速调用——只要这个 JSP 文件没有被修改。如果 JSP 文件被修改，那么将重新编译适当的 servlet。因此，JSP 提供了一种内置版本和配置控制机制。下面显示了为这个简单 JSP 所创建的 Java 代码：

```
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
```

```
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.io.ByteArrayOutputStream;
import org.apache.jasper.compiler.ibmtsx.*;
import org.apache.jasper.compiler.ibmdb.*;
import java.sql.SQLException;

public class helloJSP1_jsp_1 extends HttpJspBase {
    static char[][] _jspx_html_data = null;
    // begin
    [file="C:\\WebSphere\\AppServer\\installedApps\\mytt.ear\\webjsp.war\\hel
    -loJSP1.jsp";from=(5,3);to=(5,26)]
    static int count = 0;
    // end

    static {
    }

    public helloJSP1_jsp_1( ) {
    }

    private static boolean _jspx_initied = false;

    private static boolean checkedAttributeIgnoreException = false;
    private static boolean throwException = true;

    public final void _jspx_init() throws JasperException {
        ObjectInputStream oin = null;
        int numStrings = 0;
        try {
            InputStream fin = this.getClass().getClassLoader().getResource-
            AsStream("helloJSP1_jsp_1.dat");
            oin = new ObjectInputStream(fin);
            _jspx_html_data = (char[][] oin.readObject());
        } catch (Exception ex) {
            throw new JasperException("Unable to open data file");
        } finally {
```

```
        if (oin != null)
            try { oin.close(); } catch (IOException ignore) { }
    }
}

public void _jspService(HttpServletRequest request, HttpServletResponse
response)
    throws IOException, ServletException {
    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    String _value = null;
    setBooleanIgnoreException();
    try {
        if (_jspx_inited == false) {
            _jspx_init();
            _jspx_inited = true;
        }
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html;charset=Cp1252");
        pageContext = _jspxFactory.getPageContext(this, request,
response"", true, 8192, true);
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();
        // begin [file="C:\\WebSphere\\AppServer\\installedApps
\\mytt.ear\\webjsp.war\\helloJSP1.jsp";from=(0,0);to=(5,0)]
        out.print(_jspx_html_data[0]);
        // end

        // begin [file="C:\\WebSphere\\AppServer\\installedApps
\\mytt.ear\\webjsp.war\\helloJSP1.jsp";from=(5,28);to=(6,30)]
        out.print(_jspx_html_data[1]);
        // end
        // begin [file="C:\\WebSphere\\AppServer\\installedApps
\\mytt.ear\\webjsp.war\\helloJSP1.jsp";from=(6,33);to=(6,43)]
        out.print( ++count );
        // end
    }
```

```

        // begin [file="C:\\WebSphere\\AppServer\\installedApps
\\mytt.ear\\webjsp.war\\helloJSP1.jsp";from=(6,45);to=(8,16)]
        out.print(_jspx_html_data[2]);
        // end

        // begin [file="C:\\WebSphere\\AppServer\\installedApps
\\mytt.ear\\webjsp.war\\helloJSP1.jsp";from=(8,19);to=(8,45)]
        out.print( new java.util.Date() );
        // end

        // begin [file="C:\\WebSphere\\AppServer\\installedApps
\\mytt.ear\\webjsp.war\\helloJSP1.jsp";from=(8,47);to=(13,0)]
        out.print(_jspx_html_data[3]);
        // end
    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } catch ( Throwable t) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(new Exception(t.getMessage()));
    } finally {
        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}

private void setBooleanIgnoreException() {
    if (checkedAttributeIgnoreException) return;
    checkedAttributeIgnoreException = true;
    String initParamIgnoreException =(String)(getServletConfig().
getInitParameter("jsp.repeatTag.ignoreException"));
    if ((initParamIgnoreException != null) && (initParamIgnoreException.
toLowerCase().equals("true"))){
        throwException = false;
    }
}
}
}

```

顺便提一下，如果在这个 JSP 代码中有错误，一个类似于由编译器产生的输出将被重定向到 WebSphere 错误日志中，这个错误日志存储在 WebSphere 安装目录的 logs 目录下，如下所示：

```
org.apache.jasper.JasperException: JSPG0059E: Unable to compile class for JSPC:\WebSphere\AppServer\installedApps\jsp1.ear\jsp1.war\WEB-INF\classes\helloJSP1_jsp_0.java:80: Undefined variable or class name: textProvider
```

```
        out.print( textProvider.getTextMessage() );
```

```
        ^
```

```
1 error
```

```
    at org.apache.jasper.compiler.Compiler.compile(Compiler.java:283)
    at
org.apache.jasper.compiler.ibmtools.JasperUtil.compile(JasperUtil.java:246)
    at
org.apache.jasper.compiler.ibmtools.JspModC.compileDir(JspModC.java:205)
    at
org.apache.jasper.compiler.ibmtools.JspModC.compileApp(JspModC.java:52)
    at
com.ibm.ejs.sm.web.app.SaveAppwizAction.precompileJsps(SaveAppwizAction.
java:1058)
    at
com.ibm.ejs.sm.web.app.SaveAppwizAction.perform(SaveAppwizAction.java:236)
    at
org.apache.struts.action.ActionServlet.processActionInstance(ActionServlet.java:960)
    at
org.apache.struts.action.ActionServlet.process(ActionServlet.java:869)
    at
org.apache.struts.action.ActionServlet.doGet(ActionServlet.java:343)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:740)
    at javax.servlet.http.HttpServlet.service(HttpServlet.java:853)
    at
com.ibm.servlet.engine.webapp.StrictServletInstance.doService(ServletManager.java:718)
    at
com.ibm.servlet.engine.webapp.StrictLifecycleServlet._service(StrictLifecycleServlet.java:160)
    at
com.ibm.servlet.engine.webapp.IdleServletState.service(StrictLifecycleServlet.java:287)
    at
com.ibm.servlet.engine.webapp.StrictLifecycleServlet.service(StrictLifecycleServlet.java:105)
    at
com.ibm.servlet.engine.webapp.ServletInstance.service(ServletManager.java:428)
    at
```

```
com.ibm.servlet.engine.webapp.ValidServletReferenceState.dispatch(ServletManager.java:870)
    at
com.ibm.servlet.engine.webapp.ServletInstanceReference.dispatch(ServletManager.java:788)
    at
com.ibm.servlet.engine.webapp.WebAppRequestDispatcher.handleWebAppDispatch(WebAppRequestDispatcher.java:439)
    at
com.ibm.servlet.engine.webapp.WebAppRequestDispatcher.dispatch(WebAppRequestDispatcher.java:239)
    at
com.ibm.servlet.engine.webapp.WebAppRequestDispatcher.forward(WebAppRequestDispatcher.java:104)
    at
com.ibm.servlet.engine.srt.WebAppInvoker.doForward(WebAppInvoker.java:61)
    at
com.ibm.servlet.engine.srt.WebAppInvoker.handleInvocationHook(WebAppInvoker.java:132)
    at
com.ibm.servlet.engine.invocation.CachedInvocation.handleInvocation(CachedInvocation.java:67)
    at
com.ibm.servlet.engine.invocation.CacheableInvocationContext.invoke(CacheableInvocationContext.java:106)
    at
com.ibm.servlet.engine.srp.ServletRequestProcessor.dispatchByURI(ServletRequestProcessor.java:125)
    at
com.ibm.servlet.engine.oselister.OSEListenerDispatcher.service(OSEListenerDispatcher.java:271)
    at
com.ibm.servlet.engine.http11.HttpConnection.handleRequest(HttpConnection.java:49)
    at
com.ibm.ws.http.HttpConnection.readAndHandleRequest(HttpConnection.java:303)
    at com.ibm.ws.http.HttpConnection.run(HttpConnection.java:232)
    at com.ibm.ws.util.CachedThread.run(ThreadPool.java:122)
```


4.4 小 结

Servlet 与 JSP 是构成 Web 应用程序模型（特别是 WebSphere Web 应用程序模型）的基础的两种技术。这些技术很容易学习与掌握，并且它们在每个 Web 应用程序中都非常重要（与某些时用时不用的更高级技术相比）。特别地，作为 J2EE 应用程序技术基础之一的 EJB 也被广泛使用；但它们还是没有 Servlet 与 JSP 应用的广泛。我们将在第 7 章、第 22~28 章进一步讨论 Servlet 与 JSP。下面，将进入关于 EJB 的介绍。

第 5 章 Hello EJB

本章将简述 WebSphere Application Server 的一项功能，就是创建 EJB（Enterprise JavaBean，企业 JavaBean）。本章的目标是快速概述什么是 EJB 以及它们有什么用处；还将提供一个简单的“Hello World”式的演示程序。第 11 章将更详细并且从更加宽广的视角介绍 EJB，同时，本书的第 6 部分将讨论使用 WebSphere Application Server 开发与部署 EJB。

5.1 EJB 基础

JavaBean 是一个遵守标准接口（用于应用程序编程特别是用户接口编程）的软件组件（更多关于 JavaBean 的介绍见第 10 章）。同样，EJB 就是关于分布式系统的软件组件。正如 JavaBean 一样，EJB 也遵守一个标准接口。

为了理解分布式计算的实质，EJB 是使用关于客户与服务器内在的观念来设计的。其中，服务器就是一个驻留有 EJB 对象的 EJB 容器，客户则访问这些对象来执行某种动作。更为有趣的是，EJB 并不是直接包含在主 EJB 容器中，而是包含在几个子容器中。EJB 容器（container）本质上就是 EJB 的一个执行环境。EJB 容器可能包含几个其他的容器。当然，EJB 容器也是应用程序服务器（在这里就是 WebSphere）的一部分。

5.1.1 EJB 体系结构

客户对 EJB 的访问是通过 Java RMI（Java Remote Method Invocation，Java 远程方法调用）完成的。但是，客户方代码并不直接调用 bean 代码。相反，它将通过两个接口，分别是 EJB 本地接口与 EJB 远程接口。本地接口是与客户进行联系的第一个点，它充当 EJB 的“工厂”（也就是说，当有请求时创建 bean，当不再需要时则删除它们）。远程接口是 EJB 自己的一个代理，客户方代码利用远程接口调用 EJB 上的特定方法。总之，图 5-1 演示了 EJB 的体系结构。

虽然这个体系结构在它的不同组件描述方面有点笼统，但其功能非常强大。这种体系结构的优点是，一旦定义了远程接口与本地接口，就不必担心这些接口的实现，因为这项工作是由 EJB 容器来完成的。在实现一个 EJB 时，需要做的工作就是定义远程接口与本地接口，然后定义 bean。

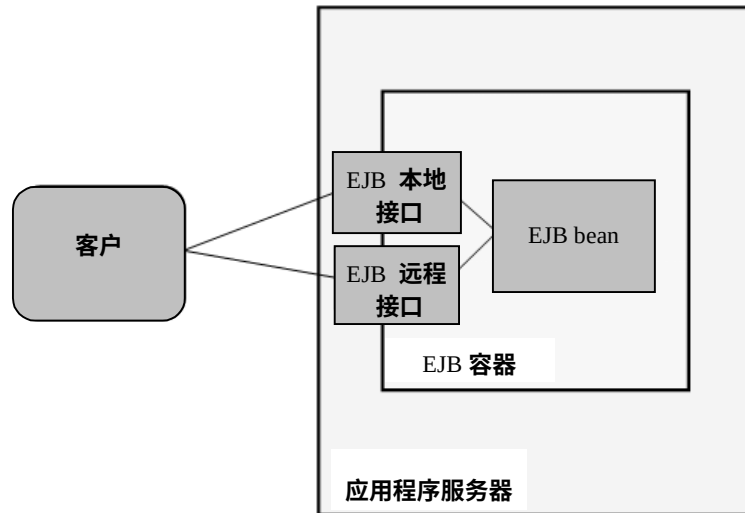


图 5-1 EJB 体系结构

5.1.2 EJB 本地接口

EJB 的本地接口扩展了 EJBHome 接口。表 5-1 列出了本地接口的方法。可以发现，EJBHome 接口仅仅提供了 3 个强制性的方法。其中第一个是 `getEJBMetaData()`，此方法返回一个包含有关于 EJB 信息的对象的引用。这个对象实现了 EJBMetaData 接口，表 5-2 中列出了这个接口的方法。EJBHome 的另外两个方法负责删除 EJB。需要注意的是，在这个接口中没有用来创建 EJB 的方法。读者在下一节将看到，这些方法将在 EJBHome 的 bean 扩展中实现。

表5-1 EJBHome接口的方法

方法	签名	描述
<code>getEJBMetaData</code>	<code>public EJBMetaData getEJBMetaData()</code>	返回关于 bean 对象的元数据
<code>remove</code>	<code>public void remove(Handle handle)</code>	删除一个给定 Handle 的 EJB 对象。Handle 是一个指向 bean 对象的引用，它能够被客户串行化
	<code>public void remove(Object primaryKey)</code>	根据一个 EJB 对象的主键删除这个 EJB 对象（主键仅仅与实体 bean 相关联）

表5-2 EJBMetaData接口的方法

方法	签名	描述
getEJBHome	public EJBHome getEJBHome()	返回实现 EJB 本地接口的对象的引用
getHomeInterfaceClass	public Class getHomeInterfaceClass()	返回实现 EJB 本地接口的类
getPrimaryKeyClass	public Class getPrimaryKeyClass()	返回用来作为这个 EJB 主键的类
getRemoteInterfaceClass	public Class getRemoteInterfaceClass()	返回用来实现 EJB 远程接口的类
isSession	public boolean isSession()	返回一个说明这个 EJB 是否为会话 bean 的布尔值

EJB 的远程接口扩展了 EJBObject 接口，表 5-3 中列出了这个接口的方法。在编写自己的远程接口时，必须包括所有需要远程调用的方法的签名。

表5-3 EJBObject接口的方法

方法	签名	描述
getEJBHome	public EJBHome getEJBHome()	返回实现 EJB 本地接口的对象的引用
getPrimaryKey	public Object getPrimaryKey()	返回 EJB 对象的主键。这个方法仅仅与实体 bean 相关
getHandle	public Handle getHandle()	返回关于对象的 Handle。Handle 是对象的可串行化引用
isIdentical	public boolean isIdentical(EJBObject obj)	测试并查看两个 EJB 对象引用是否指向相同的对象。这个方法主要对实体 bean 特别有用。需要说明的是，具有相同本地接口（并且这个本地接口具有相同的主键）的两个 bean 被认为是相同的
Remove	public void remove()	删除远程 EJB 对象

5.1.3 企业 bean

前面已经讨论了本地接口与远程接口，现在将讨论 EJB。顶级 EJB 类被称为 EnterpriseBean，其定义如下：

```
public interface javax.ejb.EnterpriseBean extends java.io.Serializable
{
};
```

可以看出，在这个定义中并没有什么特别的东西。在开发 EJB 时，显然不要扩展这个接口，因为它没有包含任何内容。相反，应该扩展一个 SessionBean 接口或 EntityBean 接口。这些接口对应于 2 个不同类型的 EJB。第一种 EJB 就是会话 bean，它通常局限于一个单独的客

户会话中。另一种 EJB 就是实体 bean，它具有更长的生命周期，并且通常在几个独立的会话中存在。在本章后面，我们将仅仅讨论会话 bean，因为“Hello World”EJB 就是一个会话 bean。

下面给出了 SessionBean 接口：

```
public interface EntityBean extends javax.ejb.EnterpriseBean
{
    public abstract void ejbActivate() throws
        java.rmi.RemoteException;
    public abstract void ejbPassivate() throws java.rmi.RemoteException;
    public abstract void ejbRemove() throws
        java.rmi.RemoteException,
        javax.ejb.RemoveException;
    public abstract void
        setSessionContext(javax.ejb.SessionContext
            context) throws java.rmi.RemoteException;
}
```

对这些方法的调用是事件驱动的，也就是说，将调用它们来通知 SessionBean 某些外部事件的发生。

5.2 “Hello World”EJB

本节给出的“Hello World”EJB 非常简单，它仅仅提供了一个方法，用来返回问候消息字符串。客户方代码通过查找这个 bean 的本地接口获取对这个 bean 的访问，然后调用这个方法使用。

开发 EJB 的第一个步骤就是定义远程接口。正如上一节中描述的那样，这个接口是对 EJBObject 接口的扩展。远程接口的代码如下所示：

```
package book.websphere.chapter5;
import javax.ejb.*;
import java.rmi.*;
public interface Hello extends EJBObject
{
    String getTextMessage() throws RemoteException;
}
```

为了使用这个 EJB，首先需要导入 EJB 包 javax.ejb。此外，还需要导入 RMI 包 java.rmi，因为在这里需要引用在此包中定义的 RemoteException。当网络出现问题时，系统将发出这个异常，并且客户方代码必须准备好处理这个异常。作为一个设计规则，远程接口中的所有 EJB 方法都应该能够发出这个异常。

在远程接口中定义的惟一方法就是 `getTextMessage()`。并且这个方法的签名必须与它在 bean 中的定义相对应，惟一可能的区别就是在远程接口中，它的定义必须指出能够发出一个 `RemoteException` 异常。

在完成远程接口之后，下面就可以继续进行本地接口的定义。在本地接口中，需要为 bean 定义创建方法。这种方法应该命名为 `create()`，并且在必要时可以带参数。这个创建方法的返回值类型就是远程接口的类型。对于远程接口方法，其创建方法必须被声明使得它能够发出 `java.rmi.RemoteException` 异常。此外，也应该发出 `javax.ejb.CreateException` 异常。当在创建新的 EJB 的过程中出现问题时（例如，EJB 容器没有可使用的内存时），就会发出这个异常。EJB 的本地接口称为 `HelloHome`，代码如下所示：

```
package book.websphere.chapter5;
import javax.ejb.*;
import java.rmi.*;

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

在这里，提供了一个不带任何参数的 `create()` 方法。注意，这个方法必须是 `public` 方法，这是因为它应该在这个包之外可见。

在有了本地接口与远程接口之后，就可以继续讨论 bean 实现类。这也是编写 `EnterpriseJavaBean` 时非常酷的部分；这时就不必提供关于本地接口与远程接口的实现——这一工作将由容器来完成。bean 实现类的名称为 `HelloBean`，其代码如下所示：

```
package book.websphere.chapter5;
import javax.ejb.*;

public class HelloBean implements SessionBean {
    public void ejbCreate()
    {
    }
    public void ejbRemove()
    {
    }
    public void ejbActivate()
    {
    }
    public void ejbPassivate()
    {
    }
    public void setSessionContext(SessionContext ctx)
    {
    }
}
```

```
    }  
    public String getTextMessage()  
    {  
        return "Hello World";  
    }  
}
```

这个代码实现了 SessionBean 接口。由于这里的需求非常简单，因此 SessionBean 接口方法的所有实现都是空的。这就意味着此 bean 将被容器的任何调用所忽略（无论它是被激活还是被钝化）。在第 6 部分读者将见到更为复杂的 EJB 实现。在 HelloBean 类中，惟一能够实际工作的方法就是 getTextMessage()，此方法将返回“Hello World”字符串。需要注意的是，这个方法的签名应该与远程接口中定义的签名相匹配。

在准备好所有必要的代码之后，下一步就是编译这些代码并且创建一个 JAR 文件。为了尽可能使工作简单，可以使用 javac 与 jar 命令。在命令提示方式下，输入以下命令：

```
javac websphere\book\chapter5\*.java  
jar cvf Hello.jar websphere\book\chapter5\*.class
```

上述两个命令都假设用户从源代码文件所在的目录中执行这两个命令。在运行编译之前，应该确信 CLASSPATH 包括 j2ee.jar，csicpi.jar，iwsorb.ar，ns.jar，以及 jts.jar 文件——所有这些文件都在 WebSphere 安装目录的 lib 目录下。

5.2.1 装配与部署

现在，就可以实际地在应用程序服务器上部署 EJB。为此，必须使用 AAT（Application Assembly Tool）。为了启动 Application Assembly Tool，应该运行 WebSphere 安装的 bin 目录下的 assembly.bat 脚本，或者，在 First Steps 窗口中单击 Launch the Application Assembly Tool。在 AAT 启动之后，将会见到一个欢迎窗口。选择 EJB Module。这时，AAT 将打开默认的 EJB 模块并且将它命名为 New_EJB_Jar1.jar，如图 5-2 所示。

为了继续部署示例 EJB，请右击左边面板的 Session Beans，并且单击 New。这时得到的窗口如图 5-3 所示，窗口要求用户为这个新的 EJB 输入信息。单击本地接口（home interface）对应的 Browse。然后，在所出现的窗口中再次单击 Browse，并且在新出现的对话框中选择 Hello.jar 文件。在这个 JAR 文件加载之后，再次进行浏览并选择 HelloHome.class 文件。在对远程类与 EJB bean 类进行类似的处理之后，就可以继续填写这个 EJB 所需的其它字段，如以下列表所示。图 5-4 显示了最终结果。

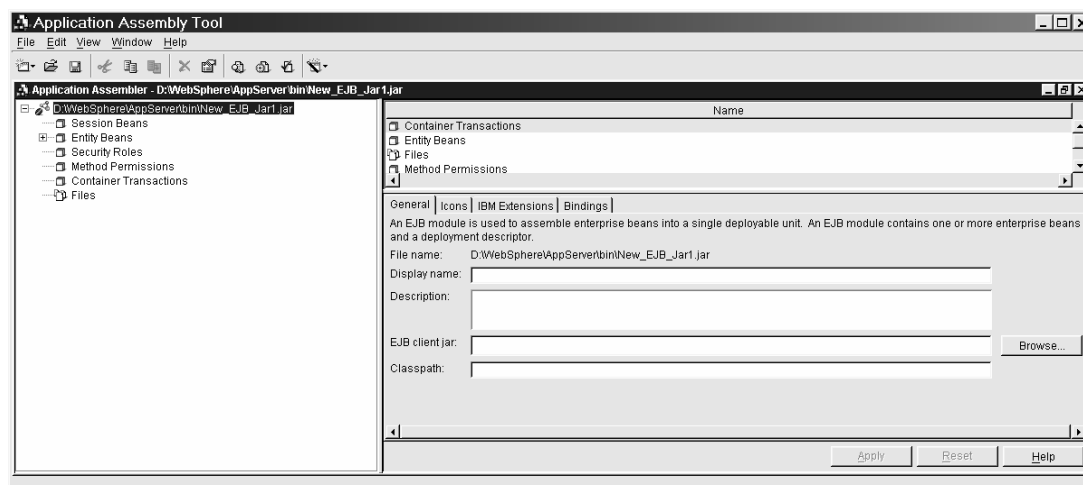


图 5-2 New EJB module 窗口

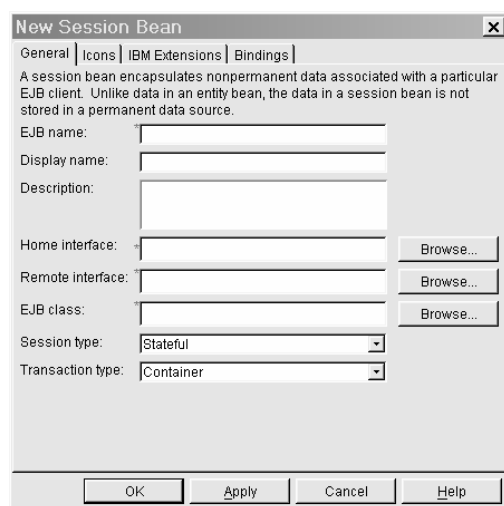


图 5-3 New Session Bean 窗口

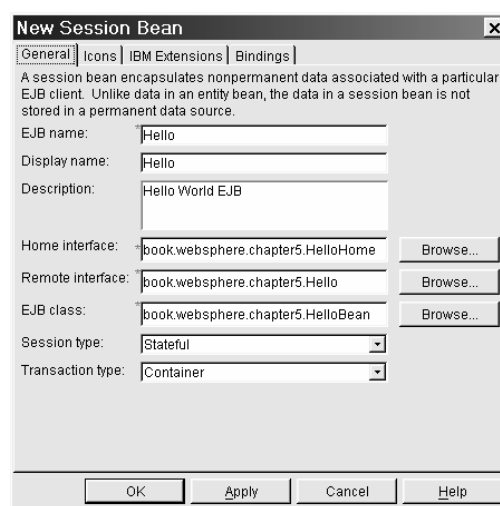


图 5-4 填写之后的 New Session Bean 窗口

- Enterprise Bean Class `book.websphere.chapter5.HelloBean`
- Remote Interface `book.websphere.chapter5.Hello`
- JNDI Home (under Bindings) `HelloBean`
- EJB Name `Hello`
- Display Name `Hello`
- Description `Hello World EJB`

这时，在 New Session Bean 窗口中单击 OK。然后，在左边面板的 JAR 文件上右击鼠标并

选择 Generate Code for Deployment。AAT 将会提示用户保存所做的修改。单击 Yes，并且将这个文件保存到 hello.jar。现在，将会出现 Generate Code for Deployment 窗口。选择 Generate Now，这样就开始了生成过程。如图 5-5 所示，在窗口的底部将显示代码生成过程的进度。这样，装配阶段就完成了。如果有一个可以部署的 JAR 文件，那么就可以继续进行实际的部署过程。

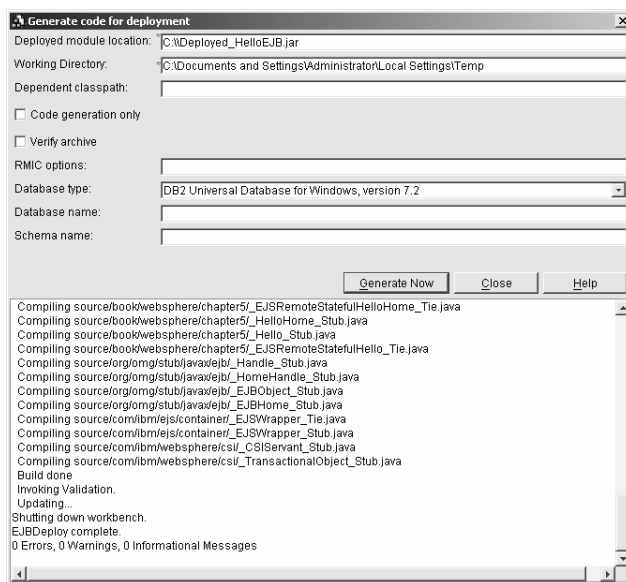


图 5-5 代码生成进度

部署过程可以在 WebSphere Administrative Console 中发生，也可以通过一个名为 ejbdeploy.bat 的脚本发生。在这里，将使用图形界面的 Administrative Console。如果用户更习惯于命令操作，或者希望利用批处理文件（在 UNIX 中为 shell 脚本）自动化部署任务，就可以考虑使用脚本。

可以在 Windows 开始菜单上，从 WebSphere 表项来启动 Administrative Console，或者从 First Steps 窗口启动 Administrative Console。首先，需要在出现的浏览器窗口中输入一个用户名。然后，将出现 Administrative Console 主窗口。在左边面板的 Nodes 下，定位机器名并且在这个机器名上打开 Enterprise Applications，如图 5-6 所示。

单击 Install 以初始化部署过程。这时将出现 Application Install Wizard。选择 AAT 所创建的 JAR 文件并且输入应用程序名。

在单击 Next 时，系统将提示用户输入这个 bean 的 JNDI (Java Naming and Directory Interface, Java 命名和目录接口) 绑定，如图 5-7 所示。这就是后来能够根据它定位到 bean 的名称。将这个名称保持为 HelloBean。在接着出现的窗口中，单击 Next。这时就开始进行应用程序

的安装，安装完成之后，用户将在企业应用程序（enterprise applications）列表中见到这个应用程序。

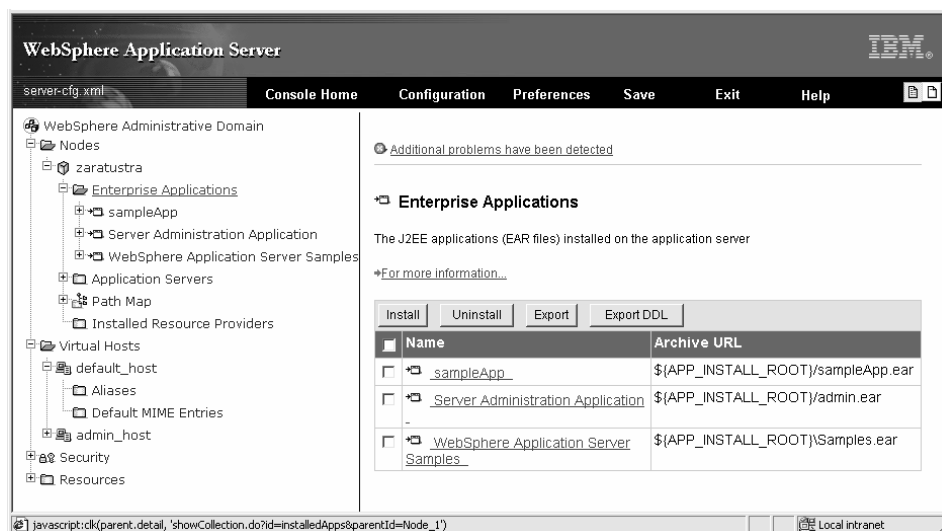


图 5-6 Enterprise Applications 窗口

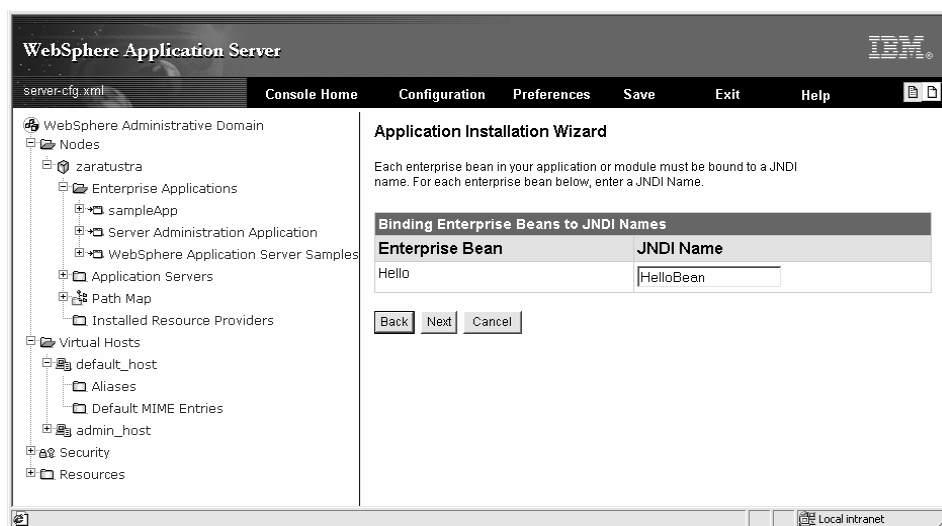


图 5-7 Application Installation Wizard 中的 JNDI 映射

5.2.2 测试 EJB

现在已经完成了 EJB 的部署，下面要进行的工作是编写一个小的客户方程序（如下所示），

来验证这个 EJB 能够正确工作。

```
package book.websphere.chapter5;
import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import javax.naming.*;

public class HelloClient {
    public static void main(String args[])
    {
        try {
            Properties env = System.getProperties();
            env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.websphere.naming.WsInitialContextFactory");
            env.put(javax.naming.Context.PROVIDER_
                "iiop://localhost");
            InitialContext ic = new InitialContext(env);
            HelloHome home = (HelloHome)
                javax.rmi.PortableRemoteObject.narrow(
                    ic.lookup("HelloBean")Hello.class);
            Hello bean = home.create();
            String ret = bean.getTextMessage();
            System.out.println("Returned Message: " + ret);
        } catch (Exception ex) {
            System.out.println("HelloClient Runtime Error:");
            ex.printStackTrace();
        }
    }
}
```

这个程序首先获取一个 `javax.naming.InitialContext` 类型的对象。这个类为命名服务提供了一个接口，将在第 20 章对此接口进行详细说明。一旦 `InitialContext` 对象被初始化，就可以使用它来查找 EJB 的本地接口。初始化过程需要一些指向命名服务提供者的属性。这个命名服务提供者由一个 Java 类（从 `com.ibm.websphere.naming` 获取）以及这个服务器（在这里就是安装了 WebSphere 的本地主机）的 URL 来标识。

在查找完成之后，需要将本地接口映射到正确的类型。这要用到 `javax.rmi.PortableRemoteObject.narrow`，因为此对象将要通过网络连接来传输。

最后，使用本地接口来查找或创建一个远程实例。在这个远程实例上，调用 `getTextMessage()` 消息。

查看客户方代码，可以发现并没有为网络连接进行特别地编码。实际上，这些工作由容器创建的代码为本地接口与远程接口自动完成。虽然这里的代码看起来非常直观，但在此背

后实际上有许多工作要做。例如，当用户调用 `InitialContext` 对象上的查找方法时，实际上是为这个特定的 bean 执行了一次全网络范围的查找。在这个特例中，网络被限制为一个服务器，但在现实世界的应用程序中，情况并非如此。调用 `HelloHome` 引用上的 `create()` 方法涉及 EJB 容器（这个容器负责创建一个新的 `HelloBean` 实例）以及将这个 EJB 容器连接到会话上。最后，在调用 `getTextMessage` 时，请求将通过网络并在 EJB 容器上执行。

现在，需要做的工作就是编译客户方代码并运行它。所得到的结果显然是一条内容为“Hello World”的消息，但在现实世界，这是一个通过网络运行的分布式程序。

5.3 小 结

本章概要地介绍了开发与部署 EJB 的过程。可以说，它们是世界上特别是 WebSphere 中最重要的应用程序框架之一。读者必须掌握在 WebSphere 下开发与部署 EJB 的方法，这一点非常重要。本章仅仅是一个介绍，因此，如果读者对创建自己的、基于 EJB 的应用程序感兴趣，那么这一章的内容是远远不够的。实际上，本书第 11 章、第 15 章以及第 29~34 章都是集中介绍 EJB 的。

第 2 部分 WebSphere 技术

第 6 章 Web 服务器与 Java 应用程序服务器

WebSphere 是一种用来构建电子商务应用程序的运行时环境。由于电子商务应用程序固有的 Web 访问特性，因此，WebSphere 必须在一个 Web 服务器（HTTP 服务器）上部署。然后，WebSphere 添加了构成应用程序服务器的层；通常在通过 HTTP 服务器进行初次访问后（直接与 Enterprise JavaBean 通信的 Java 客户除外），就对这些层进行访问。

本章将讨论一些可以用来部署 WebSphere 产品的 Web 服务器（或 WWW 服务器）。由于 HTTP 服务器目前已经很标准，并且遵守一个既定的结构，因此，WebSphere 能够通过多种不同的 HTTP 服务器被部署。实际上，几乎所有主要的服务器都被支持，因此用户在自己的环境下部署 WebSphere 时不会有什么问题。对于那些还没有可用 HTTP 服务器的底层结构的用户，WebSphere 安装过程包括了一个称为 IBM HTTP Server 的 HTTP 服务器，在从 WebSphere 光盘上安装这个服务器之后，就可以立即使用它（实际上不需要进行任何配置）。

WebSphere 支持下列 Web 服务器：

- Apache Server
- IBM HTTP Server
- Domino
- Lotus Domino Go Webserver
- Microsoft Internet Information Server
- iPlanet Web Server

本章将简要讨论 Apache Server, IBM HTTP Server 以及 Microsoft Internet Information Server。

6.1 Apache Server

Apache Server 属于软件领域一个比较新的公司的产品家族，但同时，Apache Server 又是非凡并且值得赞美的。Apache 可能是最著名的开放源代码软件（至少是同 Linux 一样著名的开放源代码软件），它由分布在全球各地的人员开发——没有任何报酬也不涉及任何公司利

益。进行 Apache 开发的人并没有得到任何报酬，并且使用这个服务器不需要购买许可证。可以说，Apache Server 软件开发程序的结果令人钦佩。

网址为 <http://www.apache.org> 的 Apache HTTP Server Project 是一个协作的软件开发工程，其目标是创建“一个健壮、商业品质、完全功能、可免费获取源代码的 HTTP (Web) 服务器的实现”。来自世界各地的志愿者（也称为 Apache Group），使用 Internet 与 Web 进行通信、规划、开发服务器及其相关文档，从而管理这个工程。此外，还有成百上千的用户为这个工程提供想法、进行编码以及书写文档。

Apache Server 面世于 1995 年，它最初是由美国伊利诺斯大学国家超级计算应用程序中心的 Rob McCool 开发的。它的最初版本获得了较大的成功，并且在不到一年的时间里，McCool 及其他人对这个软件进行了一定的修改之后，Apache Server 就成为 Internet 上的头号服务器。现在，对这个服务器中的代码的修改由用户在本地机器上进行，然后，Apache 的成员提出建议并将它放在邮件列表中。在这个代码修改能够被实现之前，需要经过活跃成员的投票表决。每天大约有 40 条消息出现在邮件列表中，这个组的通信是频繁且非正式的，但投票表决过程却是很规范化的。

在 Apache Group 中可以添加新成员；一个经常为 Apache Server 做出贡献的人将被成员推荐并且被所有投票表决的成员以匿名方式得到认可。在大多数情况下，这个“新”成员至少有 6 个月的时间在积极地为这个组的工作做出贡献，因此，投票表决时的决定是很容易做出的。

根据 Apache Server Project Web 站点的说明，Apache Group 的工作目标是：

……提供关于 HTTP 协议的健壮、商业品质的参考实现。它必须保持为一个平台，在这个平台上个人与组织能够构建可信赖的系统，这个系统既可以用于实验目的，也可以用于任务关键目的。我们相信，在线出版工具将会出现在每个人的手上，并且软件公司能够通过提供增值服务获利，例如特殊模块与支持等等。我们意识到，一个公司“拥有”一个市场可以看作是一种经济优势——在软件业界，这就意味着牢固地控制了一个特殊的渠道，以至于其他所有人都必须支付……只要 World Wide Web 协议没有被单独一个公司所“拥有”，Web 仍然是一个对大公司和小公司都平等的游戏平台。因此，必须阻止协议被某家公司所“拥有”的情况，一个关于此协议的健壮的参考实现的存在性，并且对所有公司都完全免费提供这个参考实现的源代码，这才是一件非常好的事情。

Apache 软件的惊人之处不仅仅因为它是免费开放源代码的，而且也因为它非常成功。全世界已有一半以上的 Web 服务器都基于 Apache，并且这个数字在将来的任何时候都不太可能会改变。在人们的思想中，总是认为非赢利性的复杂软件就是不稳定并且功能不齐全的，但 Apache 并非如此。

Apache Server 是一个成熟、稳定的产品，还有许多开发人员所提供的用户支持。实际上，这种支持甚至比人们通常从软件厂商所期望的支持还要好。简单地说，我们推荐任何希望

使用 Web 服务器的人使用 Apache Server (顺便提一下, Tomcat 是 Apache 组织发起的另一项正在实施的工程, 其目标是构建一个 Java 应用程序服务器。这个成功的工程正在被许多组织所使用; 在许多方面, 它都是 WebSphere 的强大竞争者)。本章讨论的另一个流行服务器就是 IBM HTTP Server, 基本上, 它就是在 Apache Server 的基础上添加了一些插件。

Apache 模块

Apache Server 以模块的形式提供了大量可配置的功能。本质上, 这个服务器的体系结构是模块化的, 它提供了在所有 Web 服务器中最齐全的各种功能模块发布, 同时, 它也可以被配置成为一个瘦的简单服务器。这个发布是以模块集合的形式提供的; 每个模块都可以被编译并且加载到此服务器中, 也可以不加载, 其决定权在用户自己。Apache Server 的 1.3.x 版本能够按照需要动态加载模块, 从而确保内存使用总是最优化, 并且除非提供了许多服务, 否则其映像是比较小的。

下面列出了一些可用的模块以及这些模块的简要描述:

- core: 在这个模块中包含有必须在每个部署中存在的核心 Apache 功能。
- mod_access: 这个模块允许管理员基于正在访问的主机控制对 Web 服务器上所提供的元素的访问。通过使用一系列 allow 伪指令与 deny 伪指令, 管理员能够基于正在提出请求的主机为目录与文件建立访问规则。
- mod_actions: 这个模块允许管理员建立 CGI 脚本, 每当某个 MIME 类型的文件被通过 Web 服务器请求时, 就将调用这个 CGI 脚本。
- mod_alias: 这个模块允许那些不在 Web 服务器根下的文件被 Web 服务器服务。这个模块允许管理员定义关于文件的某种类型的符号链接, 从而使得这个文件就像在 Web 服务器目录结构下一样, 而这同时又维持了建立网站的最大的灵活性。
- mod_asis: 这个模块允许管理员指定 Web 服务器将要服务的某种 MIME 类型的文件——也就是说, 不必关心 HTTP 响应的头部信息。
- mod_auth: 这个模块允许管理员建立一个定义授权组的文本文件。这个文件中的每一行命名了一个组, 并且指定了哪些用户属于这个组。这一信息将用来基于组进行授权。
- mod_auth_anon: 这个模块被管理员用来建立关于匿名访问的 profile (类似于 FTP (File Transfer Protocol, 文件传输协议) 中的匿名用户)。
- mod_auth_db: 这个模块支持按照 Berkeley DB 文件样式 (一个支持带密钥数据访问功能的嵌入式数据库) 进行用户认证; Apache 重用 Berkeley DB 所使用的文件样式。在这种情况下, 一个文本文件定义了组, 这种文本文件的每一行对应一个用户, 并且在这一行里列出了这个用户所属的所有组。有另一个文件维护了每个用户的口令。这些定义然后被用来进行授权与认证。

- `mod_auth_dbm` :这个模块类似于 `mod_auth_db`,但使用 DBM 文件。DBM 是最老(最原始)的 Berkeley DB 样式的数据库。
- `mod_autoindex` :这个模块支持 Web 服务器的文件索引的创建与维护。通过这个模块,既能够支持手工产生的索引文件,也能够支持自动产生的索引文件。
- `mod_cern_meta` :这个模块允许管理员基于正在访问的浏览器设置环境变量值。从浏览器上接收到的每个请求都有一个称为 `user-agent` 的字段,它的值用来标识浏览器——例如, Netscape 与 Internet Explorer 浏览器就拥有稍有不同 `user-agent` 值,使用 HTTP 连接的移动电话则拥有完全不同的 `user-agent` 值。一种比较典型的情况,就是人们要求进行响应的软件能够基于 `user-agent` 表现出不同的行为——也就是说,根据谁发出的请求给出相应的响应。这个模块为这些功能在服务器级提供了内在的支持。
- `mod_cgi` :这个模块负责为 MIME 类型为 `application/x-httpd-cgi` 的文件调用 CGI 脚本。
- `mod_digest` :这个模块允许管理员指定一个包含有用户名以及已编码口令的文件。这个口令用于 MD5 身份认证。MD5 是 R.Rivest 教授在 1991 年开发的算法,用来创建数字签名。
- `mod_dir` :这个模块支持通过手工方法或通过自动过程对目录进行索引。
- `mod_env` :这个模块允许为 Web 服务器设置环境变量,这些环境变量将被传递给服务器调用的 CGI 脚本。
- `mod_example` :这个模块提供了关于 Apache API (Application Programming Interface, 应用程序编程接口) 的例子;这个模块通常并不包含在产品服务器中。
- `mod_expires` :这个模块允许管理员设置与 HTTP 响应中发送的 EXPIRES 值相应的服务器方策略。浏览器将使用 EXPIRES 值来决定在客户 cache 中存在的页面应该在何时作废,从而导致从服务器上获取这个文件。利用该模块,管理员就能够为 Web 服务器所服务的所有页面设置策略。
- `mod_headers` :这个模块允许管理员设置与应该作为响应头部一部分的值相关的服务器方策略。这个模块允许使用诸如 `set`, `add` 以及 `append` 这样的伪指令,用于设置将要被添加到服务器所产生每个响应之中的名称-值对。
- `mod_imap` :这个模块实现了 `imagemap` 实用程序,并且是默认安装的一部分。
- `mod_include` :这个模块用来处理服务器方 `include` 并且处理对 `.shtml` 文件的请求。本书没有使用服务器方 `include`,这是因为 JSP 可用来进行任何这种所要求的处理。
- `mod_info` :这个模块用来为已经安装的服务器提供配置信息相关的服务。
- `mod_isapi` :这个模块提供对 Microsoft ISAPI (Internet Server Application Programming Interface, Internet 服务器应用程序编程接口) 的支持(现在,它是真正意义上的开放——在产品中支持另一个产品的 API!)。
- `mod_log_agent` :管理员使用这个模块来记录进入服务器的所有请求的用户代理信

息。这是一项有用的审计功能。

- `mod_log_config`：管理员使用这个模块来定制服务器日志的各方面，例如，日志文件应该放置在什么地方以及日志文件应该采用的格式。
- `mod_log_referer`：管理员使用这个模块来定制应该对什么进行记录。例如，这个模块支持指定来自某个域的请求不应该被记录的伪指令。
- `mod_mime`：这是一个比较重要的模块，用来处理具有 MIME 类型的属性（以及遵守这些类型的文档）。这个模块还支持伪指令，例如 `AddEncoding`（用来将一个文件扩展名与 MIME 类型相关联）、`AddHandler`（用来将 MIME 类型映射到可处理的应用程序），还有其他更高级的伪指令。
- `mod_mime_magic`：这个模块可以被系统中基于 UNIX 的服务器所使用，该服务器使用 UNIX magic 数来决定一个文件的 MIME 类型。
- `mod_mmap_static`：这个模块允许将一个文件读取到内存中，以便进行快速请求处理。
- `mod_negotiation`：这个模块支持 content selection（内容选择），它允许服务器基于客户发送到请求的头部中的属性来选择尽可能好的文件。
- `mod_proxy`：这个模块为 Apache 实现了 proxy cache，这个模块对于产品服务器是非常重要的。这个模块相对比较复杂，并且具有许多功能；为了得到最佳的结果，Web 服务器管理员应该仔细阅读与这个模块相关的文档。
- `mod_rewrite`：这是另一个重要而又功能强大的模块，它允许基于管理员制定的一系列规则重写 URL。
- `mod_setenvif`：这个模块允许建立被每个请求的头部所评价的规则，并且基于这些规则的满足情况设置某个环境变量的值。
- `mod_so`：这个模块支持在运行时加载模块（在 Windows 上就是 .dll；在 Solaris 上就是 .so；等等）。这就是 HTTP 插件如何在运行时被加载到 Apache 中的原因，并且允许它与 WebSphere 服务器进行通信。
- `mod_spelling`：这个模块可用来自动更正 URL 中细小的输入错误。
- `mod_status`：这个模块用来支持服务器统计功能，这些统计对于监视服务器的执行性能很有帮助。这些值在 HTML 输出中给出。
- `mod_userdir`：这个模块用来为 Web 请求设置用户的主目录。
- `mod_unique_id`：这个模块用来为每个 Web 请求创建一个惟一的标识。
- `mod_usertrack`：这个模块利用 cookie 实现对用户信息的跟踪。
- `mod_vhost_alias`：这个模块支持在相同的机器上定义几个虚拟主机（virtual host）的思想，这是一种建立单一服务器的功能，同时又允许这个服务器能够回答使用不同主机名发出的请求。

如你所见，Apache 提供了非常丰富的服务器功能。使用所有上面提到的功能，用户就能够利用 Apache Server 完成任何可以想象到的事情。

Apache 也提供了一个功能齐全而又方便的 API，以使用户能够编写与 Web 服务器进行接口的应用程序。因此，有超过一半的 Web 服务器基于 Apache Server 这一事实并不奇怪。

6.2 IBM HTTP Server

IBM HTTP Server 是 WebSphere 安装包中的一部分，并且，对于那些希望快速进入角色而又没有可用服务器的人来说，IBM HTTP Server 是一个最简单的选择。使用这个服务器，用户将得到 IBM 提供的最佳 WebSphere 支持，因此，我们建议尽量使用这个服务器。

IBM HTTP Server 是一个基于 Apache Server 但使用 IBM 公司名称的 Web 服务器（当然，Apache 组与 IBM 之间达成了许可协议）。因此，除了一些外部包装上的区别之外，所有关于 Apache Server 的讨论都对 IBM HTTP Server 同样有效。关于安装过程，使用 IBM HTTP Server 则是最简单的，因为其安装是标准安装过程的一部分（当然，这并不是说其他安装过程就很困难——其他安装过程也很直观）。

作为 IBM 与 Apache 组之间许可协议的一部分，IBM 并没有掩饰 Apache Server 完成了所有工作这一事实（关于构成这个服务器代码的动态链接库的数量，请参见图 6-1）。但是，IBM HTTP Server 确实在 Apache Server 的基础上添加了一些细小的功能，这就意味着，虽然 Apache 提供了可以被移植到其他平台的源代码，但并非 IBM HTTP Server 中所有的源代码都是可用的（特别地，这些新添加的功能模块部分就不可用）。

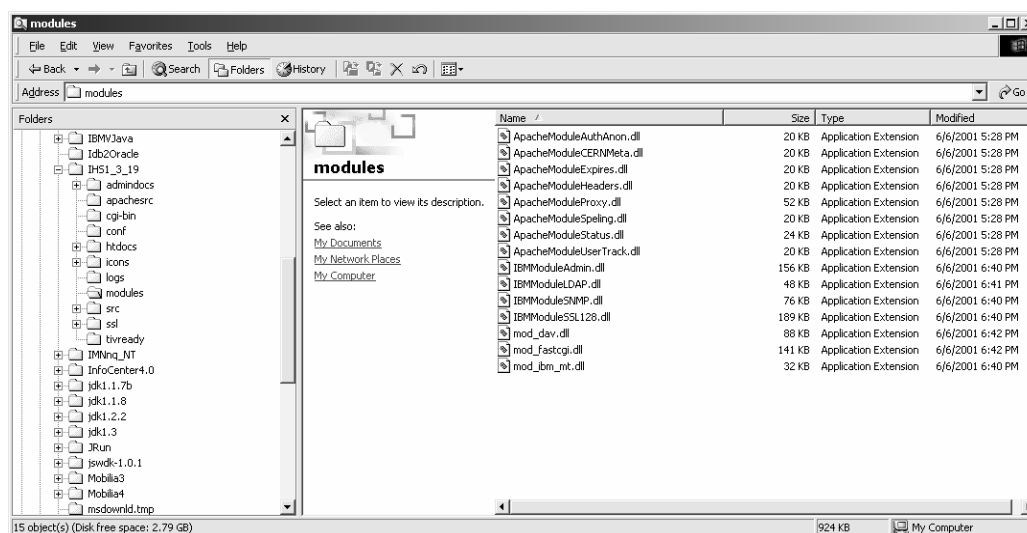


图 6-1 IBM HTTP Server 的一些 DLL（动态链接库）

6.3 Microsoft IIS4.0

IIS (Internet Information Server) 4.0 是作为 Windows NT Server 4.0 安装的一部分而免费提供的 Web 服务器。它最初是 Microsoft 进军 Internet 产品市场的策略的一部分,由于 Windows NT 以及 Windows 2000 的广泛使用,使得存在大量 IIS 安装。实际上,针对 IIS 进行的大量安全性攻击也从一个侧面说明了这个服务器是多么流行。

随着 Windows 2000 的引入,Microsoft 发布了具有不同许可的 IIS 5.0 (特别地,它不再像 4.0 那样免费提供)。本节将关注 IIS 4.0。除了免费之外,IIS 4.0 还很简单——无论从安装(用户几乎不需要做任何事情就可安装它)还是从使用的角度都是如此。所有这些特点有助于那些运行 Windows 操作系统的用户将它作为默认的服务器。当然,这也是 IIS 的一个限制——它只能在 Windows 平台上运行,而 Apache Server 与 IBM HTTP Server 则能够在包括 Windows 与 UNIX 在内的许多平台上运行。

IIS 提供的另一个方便之处就是,使用 Internet Service Manager 工具(可以从 Windows 开始菜单启动这个工具;如图 6-2 所示)能够非常容易地设置服务器的所有属性。

图 6-3 中给出了可以被这个工具管理的服务,并且给出了它们的状态。默认情况下,这些服务就是 HTTP 服务(WWW)、FTP 服务以及 Gopher 服务。图 6-3 显示了一个 Windows NT 服务器,其中只有 WWW 服务可用。这里的每个服务都有一系列属性窗口,用来配置这个服务的特性。图 6-4 到图 6-7 显示了可用来配置 WWW 服务的窗口。

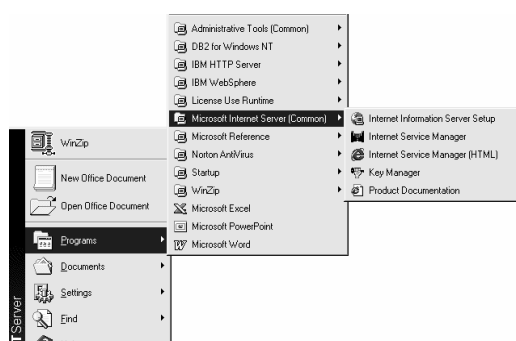


图 6-2 启动 Internet Service Manager



图 6-3 Internet Service Manager

主要的属性在 WWW Service Properties 窗口的 Service 标签中设置,如图 6-4 所示。用户可以设置服务侦听的端口(默认为 80;其他常用的值包括 8080 用于测试服务器)、默认的超时时间、最多允许的并发连接数、匿名帐号 profile 以及服务器所支持的认证方案。

在 Directories 标签中,可以为这个服务器所管理的网站指定参与内容服务的目录。图 6-5 显示了 Directory Properties 窗口,通过双击将要修改其属性的表项,就能够从 Directories 标签访问这个窗口。

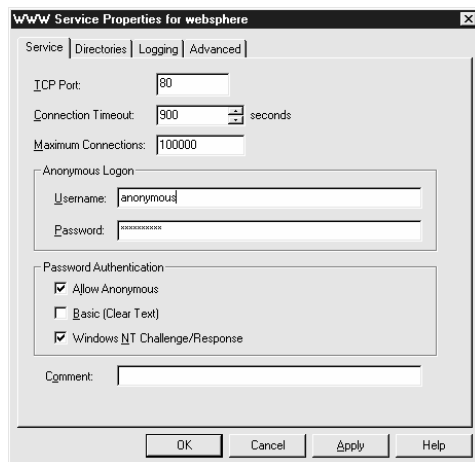


图 6-4 WWW Service Properties 窗口的 Service 标签

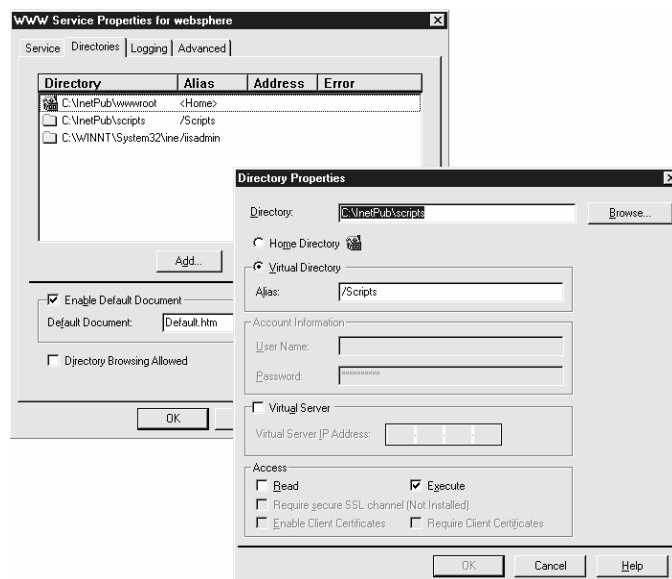


图 6-5 WWW Service Properties 窗口的 Directory Properties 窗口

在 Logging 标签（见图 6-6）中，可以设置日志记录和日志存放属性（包括将存储的日志信息直接放置到一个 SQL/ODBC 数据库中的选项）。

图 6-7 所显示的 Advanced 标签，用来基于 grant 与 deny 伪指令设置访问策略。

除了用来建立服务属性的主要工具之外，Key Manager 工具（见图 6-8）允许添加并管理附加到证书上的密钥。通过选择 Start|Programs|Microsoft Internet Server(Common)|Key Manager

来访问这个工具。这些密钥为电子传输建立安全服务器并且确保服务器认证。

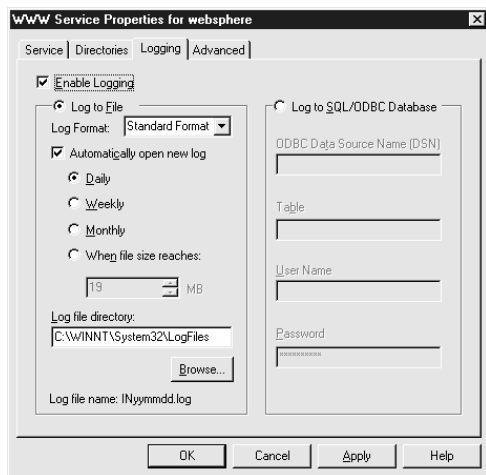


图 6-6 Logging 标签

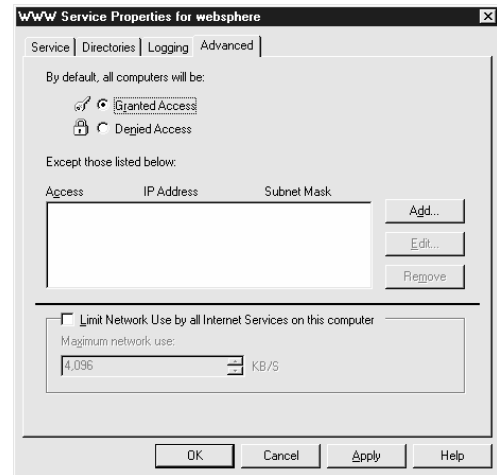


图 6-7 Advanced 标签

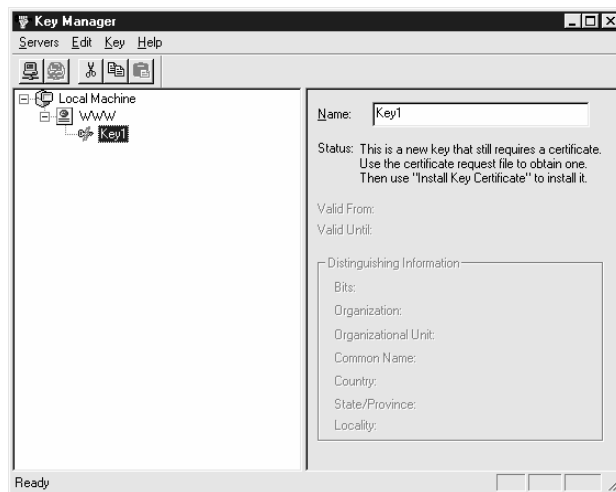


图 6-8 IIS Key Manager

最后，由于需要从一个远程站点启用服务器设置，而不一定需要打开本地主机上的 Service Manager，因此所有工具在基于 HTML 的版本上也可用。

6.4 Microsoft IIS 5.0

继承 IIS 4.0 的特点，IIS 5.0 与其宿主操作系统（Windows 2000）的集成非常紧密。到服务

器的管理是通过控制面板上的 Administrative Tools 文件夹来完成的。Internet Service Manager 为用户提供了管理 IIS 的界面。

Internet Information Services 窗口提供了 Explorer 样式的界面，从而允许非常容易地导航各种服务以及它们的组件。在图 6-9 中，表示所选择机器的节点是 thinkpad。网络上的一个节点可能正在运行多个服务，正如图中所显示的那样，在这里安装了 FTP 服务器、Web 服务器与 SMTP 服务器。用户可以浏览在这个服务器下所显示的文件。

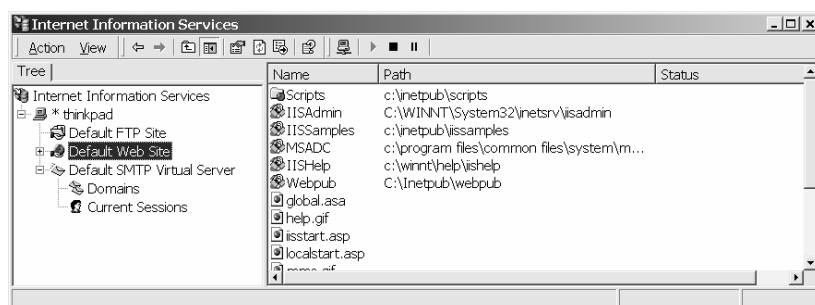


图 6-9 Internet Information Services 的 Default Web Site 视图

控制面板上的 Administrative Tools 文件夹也包括一个称为 Personal Web Manager (见图 6-10) 的项。这个工具使得能够非常容易地管理 IIS Web Server 的内容与设置。

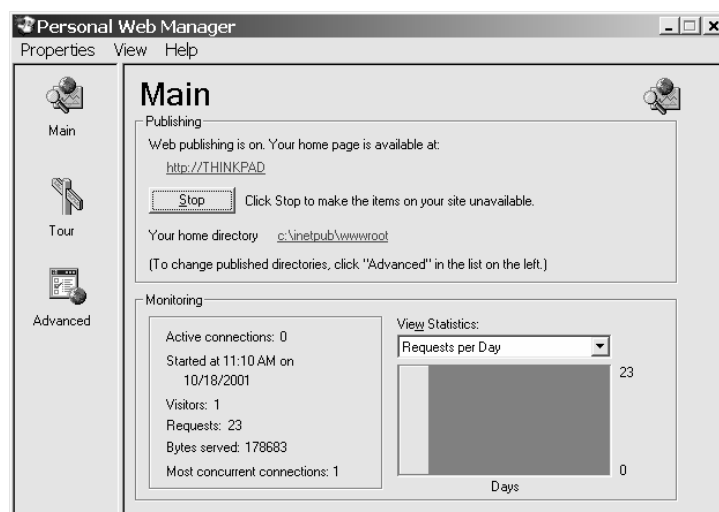


图 6-10 Personal Web Manager 工具

与其他 Web 服务器类似，验证 IIS 安装的最简单方法就是使用一个 Web 浏览器访问它。图 6-11 中显示了 IIS 5.0 主页。

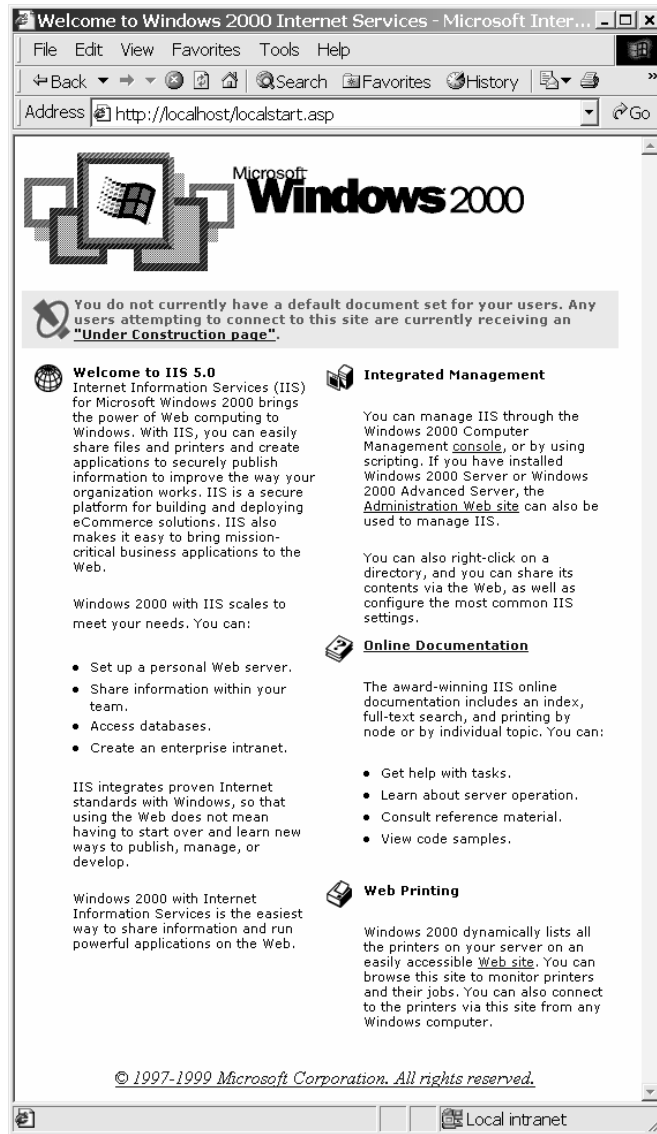


图 6-11 IIS 5.0 的默认主页

6.5 iPlanet Web Server

iPlanet Web Server 是 Netscape 服务器产品线的后续产品，它包括 Netscape Enterprise Server，Netscape FastTrack Server 以及 Kiva 应用程序服务器。它也是 Sun 公司各种应用程序服务器家族（自从 Java Web Server，Net Dynamics 家族以及 Forte 运行时环境以来）的后续产品。

显然，从技术的角度来看，它并不是所有上述产品的直接后继者，但从市场的角度来看确实如此。从技术角度来看，Sun 与 Netscape 毫无疑问地失去了它们作为服务器市场领头羊的地位，并且不能再获取这一地位。

1999 年 3 月，Netscape（后来被 AOL 收购）与 Sun 组成联合小组，试图增加在服务器软件市场的份额，因为 Sun 是 Internet 服务器硬件市场毫无争议的领导者。iPlanet 提供了一个完全功能的 J2EE 服务器，因此它实际上是 IBM WebSphere 的竞争者。但 IBM 提供了对 iPlanet Web Server 的支持，正如它对 Microsoft IIS 的支持一样。

iPlanet Web Server 得到各种 UNIX 变体、Linux 以及 Windows 服务器平台的支持。

iPlanet 管理是基于 Web 的。图 6-12 显示了 Web Server Administration Server 的主页面，从这里用户可以管理多个服务器。

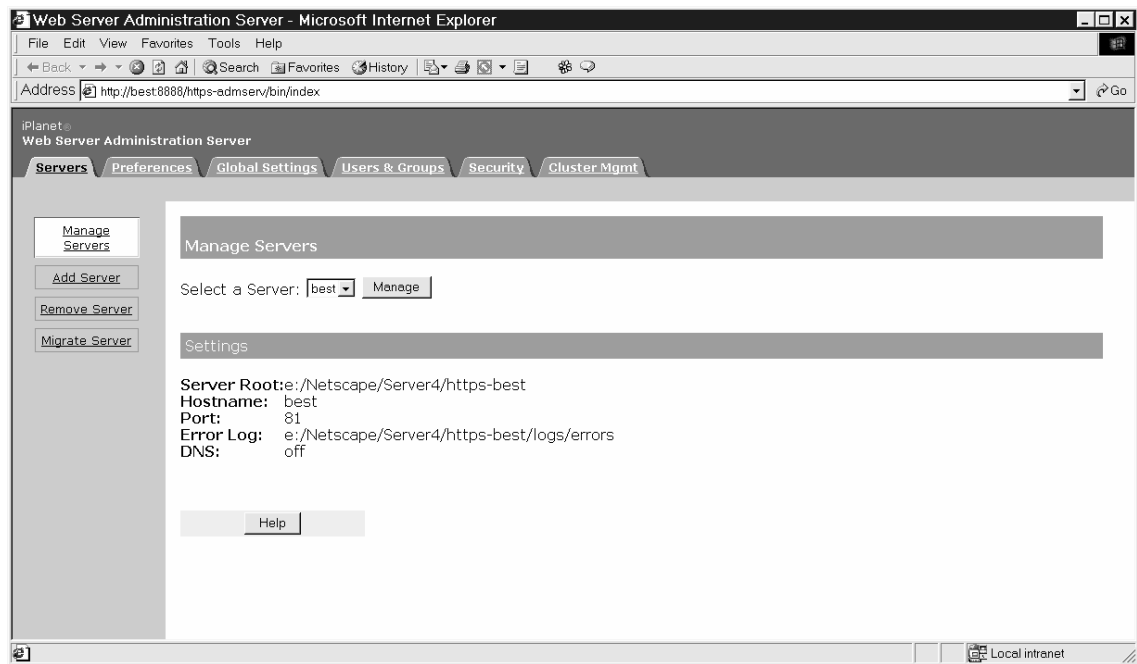


图 6-12 iPlanet Administration Server 页面

用户能够从 Manager Server 页面的 Preferences 标签管理每个服务器，如图 6-13 所示。

与其他 Web 服务器类似，iPlanet 有它自己的默认主页，在安装 iPlanet 服务器完成之后就能够访问这个主页。

最后，iPlanet 中的密钥管理是通过 Server Manager 页面上的 Security 标签进行的，如图 6-14 所示。

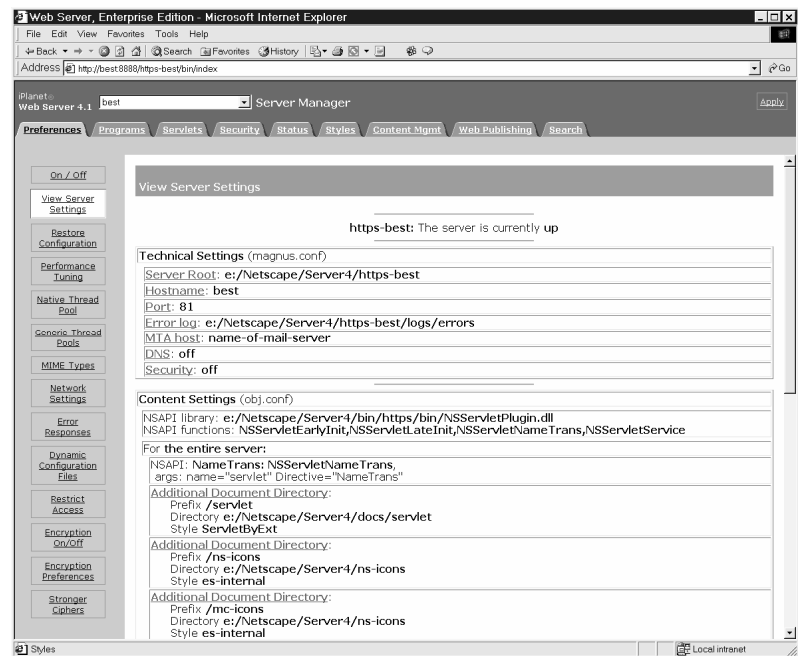


图 6-13 iPlanet Server Manager 的 Preferences 标签

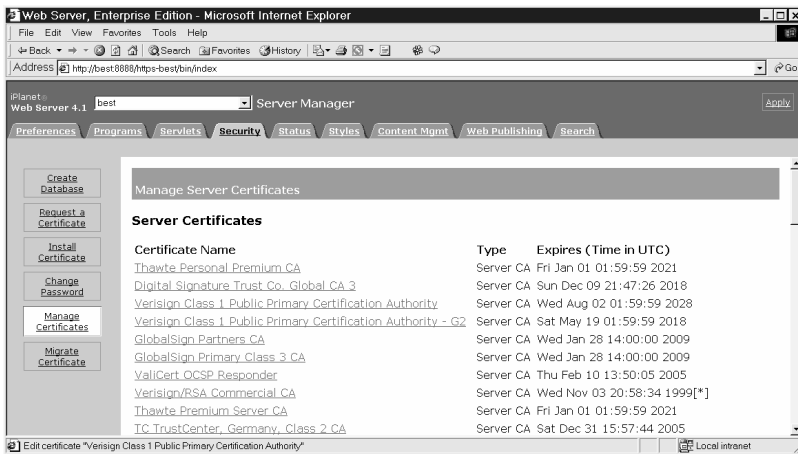


图 6-14 iPlanet Server Manager 的 Security 标签

6.6 Java 应用程序服务器

应用程序服务器（application server）是一个不太容易理解的概念。应用程序服务器可以看

作是 Web 服务器的加强版本。它除了能够支持 HTTP 之外，还提供了一系列 API 与服务用来支持开发人员的应用程序。这些 API 与服务典型地包括对一致性、事务、安全性、XML (Extensible Markup Language, 可扩展标记语言) 以及最新的 Web 服务技术 (例如 SOAP (Simple Object Access Protocol, 简单对象访问协议) 以及 UDDI (Universal Description, Discovery and Integration, 通用描述、发现和集成)) 的支持。

WebSphere 属于基于 Java 的应用程序服务器组。这些服务器的其他比较著名的组是基于 Microsoft.NET 平台的。此外，还有一些服务器使用私有的解决方案 (例如 Cold Fusion)。

从本质上说，Java 应用程序服务器是 Web 服务器的扩展，提供了执行服务器方 Java 代码的能力。Sun 公司引入的第一种服务器方 Java 就是 servlet，它是服务器上第一个 Java 实例的 Java 代码片段。虽然 servlet 是一个笼统的概念，但它们最常见的用法是通过一个 HTTP 会话被调用 (这也就是为什么 servlet 通常从 HTTP Servlet 继承的原因)。

在 servlet 第一次被引入时，CGI 脚本执行了大多数 Web 服务器方处理工作。CGI 是最早被用来创建动态 Web 内容的技术之一。动态地创建 Web 内容的重要性是显而易见的，这也是我们所了解的大多数 Web 站点的基础。CGI 底层的概念就是要求 Web 服务器调用一个外部程序，用以产生内容，并且这些内容将被发送给客户方。

虽然 CGI 迅速成为动态创建内容的事实标准，但它存在一些致命的缺点，特别是在考虑到性能问题时更是如此。在 CGI 编程模式下，Web 服务器需要创建一个新进程来处理每个到达的请求。而创建新进程这种工作通常代价比较高昂。对于 CGI 脚本，在考虑到实现这些脚本的最常用语言是 Perl 时 (因为每个到达的请求都需要加载一个新的 Perl 解释器) 就更是如此。

有几个不同的产品提供了对这个问题的解决方案。由于在这里 CGI 脚本使用 Perl 实现，因此 Apache Server 提供了一个名为 mod_perl 的模块。这个模块将加载一个 Perl 解释器到 HTTP 服务器中，并且这个解释器负责处理请求。其他使用私有 (通常是非标准的) 的 API 的解决方案包括 FastCGI (来自一个名为 OpenMarket 的公司)、Microsoft 的 ASP (Active Server Page) 以及来自 Netscape 的 Server Side JavaScript。

servlet 也进入这一领域，并且迅速成为主流，这是因为它们既提供了 CGI 以及其他服务器方处理技术所具有的优点，又具有 Java 内在的优点 (例如可移植性、安全、对丰富的 Java 库的可用性，并且为广大 Web 编程者所熟知) ——同时，又克服了标准 CGI 编程所具有的性能缺点。

servlet API 规范及其参考实现是由 Sun 的 JavaSoft 形成的 (正如所有与 Java 相关的技术一样)。这个参考实现首次出现在 Sun 的 Java Web Server 中。现在，从各种软件厂商都能够得到 servlet 服务器。其中有些服务器是具有内置 servlet 支持的 Web 服务器，例如 Sun 的参考实现服务器或 Netscape 的服务器。前面已经提及，Apache 也提供了一个名为 JServ 的扩展模块，用来提供 servlet 支持。

Java 应用程序服务器的第二代产品不能被简单地看作在 Web 服务器上进行插件扩充。所有现代 Java 应用程序服务器 (显然包括 IBM 的 WebSphere) 都是综合型的运行时平台，其功

能并不仅仅是提供动态 Web 页面以及作为 HTTP 请求的结果运行 Java 代码。WebSphere 应用程序服务器就是从 servlet 引擎发展演变而来的，并且它基于一个称为 ServletExpress 的先前产品而开发。从这以后，它进行了非常大的演变，以至于最原始的内核中并没有多少仍然被保留。

在 Java 应用程序服务器演变过程中的另一个比较大的步骤是 Sun J2EE 标准的引入。J2EE 是为“Java Application Server”这一概念而作出的第一次标准化努力。这个标准明确定义了 J2EE 服务器应该由什么组成。在 J2EE 中，称为容器（containers）的执行环境存在于服务器中。每个容器都能够执行 servlet，JSP（生成动态 Web 内容的另一种方法）以及 EJB。本书将在第 15 章讨论 J2EE。目前在市场上已经有几个可用的 J2EE 实现。比较值得一提的是：

- IBM 的 WebSphere
- BEA 的 WebLogic
- Oracle 的 Application Server
- Hewlet Packard 的 Bluestone

除此之外，还有其他公司（例如 Sybase，Silverstream 等）的产品可供使用。

6.7 小 结

虽然 IBM 提供 IBM HTTP Server 作为 WebSphere 安装的一部分，用户也可以非常自由地选择另一个不同的 Web 服务器，作为与 WebSphere 应用程序服务器一同工作的部署解决方案。许多组织都已经有了可以使用的 Web 服务器框架，并且它们可能不乐意使用需要抛弃自己已有框架的解决方案。当然，这种决定完全在用户自己，因为 WebSphere 能够支持所有主要的 Web 服务器。

如果还没有可供使用的 Web 服务器，或者如果需要一个独立的 Web 服务器用来在 WebSphere 上部署应用程序，我们建议使用 IBM HTTP Server。这个服务器与其他解决方案具有一样的优点，并且能够非常容易地安装与支持。它也是最常用的组合。

第 7 章 servlet 与 JSP

本章将讨论 servlet 与 JSP (JavaServer Page, JavaServer 页面) 的操作模型的基本原理。前面已经说明了 servlet 与 JSP 如何在 WebSphere 服务器中工作, 并且, 本书将在第 22~28 章更详细地讨论 servlet 与 JSP。这些章包含了一系列例子, 用来介绍这些技术元素的功能。本章将介绍这些例子背后的理论。在这里将描述 servlet 与 JSP 规范所指定的模型。读者必须理解这个模型, 这一点非常重要, 因为它是贯穿所有 servlet 以及 JSP 例子的主线, 并且, 只有理解了 this 模型, 才能更好地理解这些概念。

7.1 servlet 与 JSP——它们相同吗

servlet 是一种遵守 servlet 规范的 Java 类。servlet 是由 Sun 公司引入的, 用来作为服务器上的第一种 Java 实例之一。当时, 大多数 Web 服务器方处理工作都是由 CGI 程序或由使用某种专利的应用程序编程接口 (例如 Netscape Server 的 API, 或称为 NSAPI; Internet Server 的 API, 或称为 ISAPI) 链接到 Web 服务器上的程序来完成的。为了提供服务器方处理具有的所有优点, 同时又充分利用 Java 所固有的优点 (例如可移植性、可以使用大量 Java 库等等), 引入了 servlet。此外, servlet 比 CGI 程序更快, 并且可以在许多 Web 服务器上部署。servlet 迅速流行起来的原因并不仅仅是它们比其他方法更优越, 同时也因为它们简单、易于理解。正如普遍存在的事物一样, 简单的事情总是能够迅速流行。

JSP 是一个比较新的 Java 技术领域。在许多方面, JSP 可以被看作 servlet 的反面——就像“镜像”一样。servlet 通过运行访问数据库或执行某种其他处理的 Java 代码来产生动态 HTML, 然后将 HTML 写入到输出流中。HTML 标记元素作为字符串被嵌入在 Java 代码中。JSP 的作用则相反。从本质上说, JSP 是 Web 服务器所服务的 HTML 页面。它们与普通 HTML 页面的区别表现在两个方面: 它们的文件扩展名为 .jsp; 它们包括一些不是 HTML 标记的额外标记——也就是 JSP 标记。这些 JSP 标记允许在 HTML 页面中嵌入 Java 代码行。因此, 前面所说的“镜像”就是指 JSP 实际上是作为嵌入了 Java 的 HTML (与 servlet 不同, 它是将 HTML 字符串嵌入在 Java 代码中)。

所有这些区别听起来似乎并不重要, 但事实并非如此。JSP 对于将应用程序开发人员编写的函数从网站构建者提供的函数或从负责表示与布局的函数分离出来更为有用。因此, 使用 JSP 的开发组经常采用的一种做法是, 吸收那些构建 HTML 页面的人利用虚构的静态数据进行工作, 然后将这些数据传递给 Java 编程者, Java 编程者在需要使用这些虚构的数据时,

就用可以构建实际动态数据的 Java 代码来替换这些虚构的数据。在有些情况下,使用 servlet 更为合适,而在另一些情况下使用 JSP 则更好;其不同之处在于 HTML 所起的作用。如果大部分 HTML (表示层) 都是固定的,并且 Java 代码主要用来动态产生数据,那么使用 JSP 就更为合适。在 JSP 出现之前,许多 servlet 开发人员通过打包关于 HTML 页面的“模板”并且在相互之间传递动态数据这种方法来模拟 JSP 所做的工作。

7.2 servlet 操作模型

图 7-1 显示了 servlet 的操作模型。正如这个图所显示的那样, servlet 与 servlet 服务器进行交互,并且在整个生命过程中经历了许多阶段。

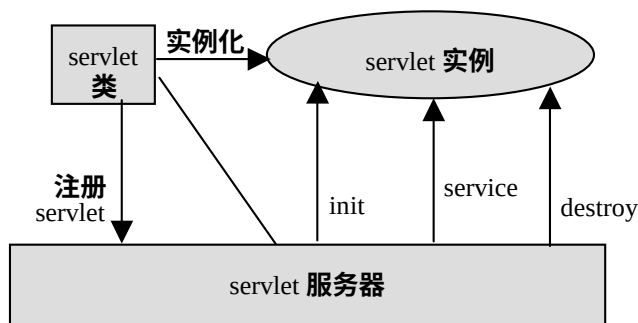


图 7-1 servlet 操作模型

首先, servlet 在服务器上进行注册(也就是它被加入到服务器的配置中)作为任何 servlet 服务器必须支持的功能部分。从某种角度看,服务器使用已经注册的类来实例化 servlet。这一过程通常在服务器启动阶段完成,当然,根据服务器的不同这种情况也可能不一样。

下一步就进入到初始化阶段,这个阶段的动作由 servlet 必须实现(或继承)的 init 方法来完成。因此,在 servlet 服务器实例化这个 servlet 之后,将调用 init 方法并传递封装有环境信息的 ServletConfig 对象。在 servlet 初始化过程中,可能需要使用这些信息。

下一步就进入实际的调用阶段。每个 servlet 必须实现或继承一个 service 方法,当一个需要被 servlet 处理的请求到达时, servlet 服务器将调用这个方法。需要说明的是, HTTP servlet 也能够实现这个 service 方法,但它们也可以实现 doGet, doPost 方法或同时实现这两个方法——这也反映了 HTTP 请求使用 get 或 post 方法这一事实。如果既实现了 service 方法,也实现了 doGet 或 doPost 方法之一,那么首先将调用 service 方法,然后再调用 doGet 与 doPost 方法。因此,覆盖 doGet 与 doPost 方法总是一种安全的做法。

最后,当 servlet 服务器需要回收 servlet 时,它将调用 destroy 方法——它是 servlet 必须实现的另一个方法。

7.2.1 servlet API

servlet 操作模型涉及 servlet 与 servlet 服务器之间的合约。这个合约在 javax.servlet.Servlet 接口中定义，表 7-1 给出了这个接口的方法。从本质上说，servlet 必须提供 3 个最基本的操作步骤：初始化阶段、请求的服务阶段、完成阶段。

表7-1 servlet的方法

方法	签名	描述
init	Void init(ServletConfig config)	用来初始化 servlet。servlet 引擎或 servlet 容器将正好调用一次这个方法。该方法的参数为 servlet 提供了关于主机服务器的信息
service	void service(ServletRequest request, ServletResponse response)	执行客户发出的一个请求。每当客户给出一个请求时，这个方法就被调用一次
destroy	void destroy()	清除被占用的任何资源
getServletInfo	String getServletInfo()	返回一个字符串，在此字符串中包含有关于 servlet 的信息。在 WebSphere 中，可以在部署时从 Administrative Console 设置这个字符串
getServletConfig	ServletConfig getServletConfig()	返回一个 servlet 配置对象，这个对象应该与作为参数传递给 init 方法的对象相同

注意：WebSphere Application Server 4.0 支持 servlet API 2.2。本章（以及本书）的其余部分讨论的内容将与这个版本的 API 相关。

7.2.2 servlet 初始化

当通过 init 方法在服务器上注册 servlet 时，就会发生初始化动作。在 init 方法中包括的主要是：为了保证 servlet 的正确操作所必需的启动代码，需要花费较长执行时间的启动代码——这些事情包括读取文件、创建网络连接等等。常常在实际的请求到达之前就已经创建了实例，因此，当用户在等待响应时便能够有效节省实例化所需的时间。

在现实世界中另一个常见的做法是，存在一个模型，在此模型 servlet 池中的 servlet 在请求开始到达之前就全部被实例化（并且这个 servlet 池中的 servlet 常常共享相同的资源）。servlet 保证了在任何服务器请求到达之前 init 方法将返回。此外，init 方法不会被相同的 servlet 实例再次调用。这样就从根本上确保 init 方法是线程安全的。如果在初始化过程中出现任何错误，init 方法将能够发出 ServletException 异常或 UnavailableException 异常。

7.2.3 servlet 处理

在 servlet 被初始化之后，就可以开始处理请求了。servlet 的主要方法就是 service 方法。通过这个方法，servlet 实现一个请求范型与一个响应范型。service 方法使用一个到达的请求

作为 `ServletRequest` 中的参数，并且创建一个封装在 `ServletResponse` 中的响应。除此之外，servlet 模型并没有更多的要求——可以说，servlet 是一个通用的存在，它服务请求并创建响应。在请求对象中包含有关于服务请求的信息，包括客户提供的参数（如表 7-2 所示）。响应对象用来向客户返回信息，这个对象的方法在表 7-3 中给出。

表7-2 ServletRequest的方法

方法	签名	描述
<code>getAttribute</code>	<code>Object getAttribute(String attribute)</code>	返回关于这个请求的指定名称的属性的值，如果这个指定的属性不存在，则返回 <code>null</code> 。这些属性是特定于服务器的
<code>getAttributeNames</code>	<code>Enumeration getAttributeNames()</code>	返回一个枚举对象，在这个对象中包含有关于此请求的所有属性名
<code>getCharacterEncoding</code>	<code>String getCharacterEncoding()</code>	返回这个请求的输入所使用的字符集编码方法
<code>getContentLength</code>	<code>int getContentLength()</code>	返回这个请求数据的大小，如果其大小未知，则返回 <code>-1</code> 。当为一个 HTTP 发送请求提供服务时，这个方法特别有用
<code>getContentType</code>	<code>String getContentType()</code>	返回请求数据的 MIME 类型，如果其类型未知，则返回 <code>null</code>
<code>getInputStream</code>	<code>ServletInputStream getInputStream()</code>	返回一个输入流，用来读取请求体二进制数据。对于文本数据，应该使用 <code>getReader()</code> 方法，因为此方法考虑了字符集编码问题
<code>getLocale</code>	<code>Locale getLocale()</code>	基于请求中“ <code>AcceptLanguage</code> ”头部的信息，返回客户希望的区域
<code>getLocales</code>	<code>Enumeration getLocales()</code>	基于请求中“ <code>AcceptLanguage</code> ”头部的信息，返回表示客户可接受的区域的 <code>locale</code> 对象的枚举。这些区域是按照优先顺序递减的方式排列的
<code>getParameter</code>	<code>String getParameter(String parameter)</code>	返回一个包含关于指定参数的值的字符串，如果这个参数不存在，则返回 <code>null</code> 。在输入被解析之后，就可以调用这个方法
<code>getParameterNames</code>	<code>Enumeration getParameterNames()</code>	以枚举的形式返回请求的参数名。在输入被解析之后，就可以调用这个方法
<code>getParameterValues</code>	<code>String[] GetParameterValues(String)</code>	返回一个字符串数组，当中放置了参数值。这个方法对于带有多个值的参数非常有用

(续表)

方法	签名	描述
getProtocol	String getProtocol()	返回请求所使用的协议与版本
getReader	BufferedReader getReader()	返回一个带缓冲器的 reader, 用来从请求体中读出文本。这个 reader 被使用适当的字符编码所初始化
getRemoteAddr	String getRemoteAddr()	返回进行请求的客户的 IP 地址
getRemoteHost	String getRemoteHost()	返回进行请求的客户的主机名
getRequestDispatcher	RequestDispatcher getRequestDispatcher(String path)	返回 RequestDispatcher 对象——它是存在于参数路径上的资源的封装器。封装器可用来将请求转发到资源, 或者用来包括来自资源的响应
isSecure	boolean isSecure()	指出请求是否来自一个安全的连接 (例如 HTTPS)
removeAttribute	void removeAttribute(String name)	从请求中删除一个属性
setAttribute	void setAttribute(String name, Object obj)	存储或重置请求中的一个属性

表7-3 ServletRequest的方法

方法	签名	描述
flushBuffer	void flushBuffer()	强制将任何已缓存的内容写入到客户
getBufferSize	int getBufferSize()	返回用于响应的缓冲器的当前大小
getCharacterEncoding	String getCharacterEncoding()	返回响应所使用的字符集编码。通过修改内容类型, 能够改变响应的字符集编码
setContentLength	void setContentLength(int)	设置响应的内容长度, 单位为字节
setContentType	String getContentType()	设置请求数据的 MIME 类型。如果没有设置, 默认为 text/plain
getLocale	Locale getLocale()	返回用于响应的区域
getOutputStream	ServletOutputStream getOutputStream()	返回用于写出二进制数据的输出流。对于文本数据, 应该使用 getWriter()方法
getWriter	PrintWriter getWriter()	返回用于进行文本输出响应的打印写出对象
isCommitted	Boolean isCommitted()	用来指出响应是否被确认
reset	void reset()	清除存储在缓冲器中的任何数据、状态代码以及头部
setBufferSize	void setBufferSize(int sz)	设置响应的缓冲器大小
setLocale	void setLocale(Locale loc)	设置响应的区域, 也将适当地设置头部

7.2.4 servlet 完成

servlet 合约的最后一个部分就是完成阶段，这个阶段通过 `destroy` 方法来实现。当服务器卸载 servlet 时，服务器将调用一次这个方法。在运行 `destroy` 方法之后，就确保了 `service` 方法不会被调用，直到服务器再次加载这个 servlet。当服务器删除一个 servlet 时，它将在所有的服务完成之后，或者在经过一个特定的服务秒数之后，调用 `destroy` 方法。对于需要长时间运行的操作，一个服务请求即使在服务器调用 `destroy` 方法之后仍然可活跃。

servlet 必须负责确保任何仍然在 `service` 阶段的线程工作完成。为了达到这一目标，一种方法就是使 `destroy` 方法设置一个标志并且使 servlet 在每个请求结束之后检查这个标志。如果这个标志已经出现，并且没有其他正在运行的请求，servlet 就将清除自己的资源。实际上，这种策略很少使用，因为在大多数情况下，servlet 执行比较短的操作（至少必须为用户提供可以接受的响应时间这一点就要求 servlet 执行时间比较短）。

7.2.5 线程安全

线程安全是 servlet 编程过程中的一个重要考虑因素。此外，可能有多于一个的活跃线程在使用相同的 servlet 对象。因此，除非特别地在 servlet 中实现 `SingleThreadModel` 接口（这样就明确指出不希望 servlet 是多线程的），否则就必须确保编写的代码是可重入的并且是线程安全的。从这个方面来考虑，应该尽量少用实例变量。当必须使用这种变量时，就必须考虑使用同步方法对它们进行访问，从而保证当多个线程并发地开始运行时不会出错。

7.2.6 HTTP servlet

前面已经说过，最常见的 servlet 类型就是 HTTP servlet。这个 servlet API 定义了 `javax.servlet.http.HttpServlet` 类（它是一个继承于 `javax.servlet.GenericServlet` 的抽象类，而 `javax.servlet.GenericServlet` 又是实现 `javax.servlet.Servlet` 接口的抽象类）。表 7-4 显示了 `HttpServlet` 方法。

表7-4 HttpServlet的方法

方法	签名	描述
<code>doGet</code>	<code>void doGet(HttpServletRequest request, HttpServletResponse response)</code>	实现 HTTP get 操作
<code>doPost</code>	<code>void doPost(HttpServletRequest request, HttpServletResponse response)</code>	实现 HTTP put 操作
<code>doPut</code>	<code>void doPut(HttpServletRequest request, HttpServletResponse response)</code>	实现 HTTP put 操作。这个操作很少使用，因此默认的实现将报告一个 HTTP bad_request 错误。put 操作类似于通过 FTP 发送一个文件
<code>doDelete</code>	<code>void doDelete(HttpServletRequest request)</code>	实现 HTTP delete 操作，这一操作允许客户请求

(续表)

方法	签名	描述
	request,HttpServletResponse response)	从服务器删除一个 URI
doTrace	void doTrace(HttpServletRequest request,HttpServletResponse response)	实现 HTTP trace 操作。默认的实现将送回一条消息、在此消息中包含请求中的所有头部。这个方法很少被覆盖
doOptions	void doOptions(HttpServletRequest request,HttpServletResponse response)	实现 HTTP options 操作，这一操作允许客户决定所允许的 HTTP 操作。默认的实现将检查 servlet 类，以查看已经实现了哪个 do 方法
getLastModified	long getLastModified(HttpServletRequest request)	获取“最后一次”修改的时间，这个时间以自从 1970 年 1 月 1 日凌晨以来的毫秒数表示。这一方法主要用于浏览器 cache。默认的实现将返回一个负数，表示不应该使用任何 cache 机制
service	void service(ServletRequest request,ServletResponse response)	为一个请求提供服务。如果它不是 HTTP 请求，那么将发出一个异常。否则，后面的 service 方法将被调用，并且在调用时提供适当的类型转换
	void service(HttpServletRequest request,HttpServletResponse response)	基于包含在请求中的操作，调用一个 do 方法

在一个 HTTP servlet 中，service 方法已经实现。默认的实现将根据 HTTP 操作调用一个 do 方法以解析 HTTP 请求。每个方法都使用 HttpServletRequest 对象（见表 7-5）以及 HttpServletResponse 对象（见表 7-6）作为参数。

表7-5 HttpServletRequest的方法

方法	签名	描述
isUserInRole	boolean isUserInRole(String role)	指出用户是否起到了指定的作用
isRequestedSession IdValid	boolean isRequestedSession IdValid()	指出与请求相关联的会话 ID 是否有效
isRequestedSession IdFromURL	boolean isRequestedSession IdFromURL()	指出与请求相关联的会话 ID 是否已经进行了 URL 编码
isRequestedSession IdFromCookie	boolean isRequestedSession IdFromCookie()	指出与请求相关联的会话 ID 是否以一个 cookie 出现
getSession	HttpSession getSession(boolean)	获取与请求相关联的当前会话（参见第 22 章）

(续表)

方法	签名	描述
	HttpSession getSession()	等价于 getSession(true)
getServletPath	String getServletPath()	返回与所调用的 servlet 相关联的请求 URI 部分
getRequestURI	String getRequestURI	返回除了任何查询字符串之外的请求 URI
getRequestId	String getRequestedSessionId()	返回与请求相关联的会话 ID
getRemoteUser	String getRemoteUser()	返回发出这一请求的用户名
getQueryString	String getQueryString()	返回包含在请求 URI 中的查询字符串。查询字符串在 URI 中以问号来标识
getPathTranslated	String getPathTranslated()	返回放置在 servlet 路径后面且出现在任何查询参数之前的任何可选的路径信息。此外，还将把这个附加的路径信息转换成实际的路径
getPathInfo	String getPathInfo()	返回放置在 servlet 路径后面并且出现在任何查询参数之前的可选的路径信息
getMethod	String getMethod()	返回请求的 HTTP 操作 (例如 get, post)
getIntHeader	int getIntHeader(String header)	返回一个整型 HTTP 头部的值
getHeaderNames	Enumeration getHeaderNames()	以枚举类型返回 HTTP 头部名
getHeader	String getHeader(String header)	返回所请求的头部的值
getDateHeader	long getDateHeader(string name)	返回请求头部中日期字段的值。这个值以自从 1970 年 1 月 1 日凌晨以来的毫秒数表示
getCookies	Cookie[] getCookies()	获取请求中所发现的关于 cookie 的数组 (请参见第 22 章)
getAuthType	String getAuthType()	返回这个请求所使用的认证方案
getContentPath	String getContextPath()	返回这个请求的上下文 (也就是这个请求的 URI 的前缀)
getUserPrincipal	Principal getUserPrincipal()	返回一个在其中包含有关于当前被认证用户的名称的 java.security.Principal 对象

表7-6 HttpServletResponse的方法

方法	签名	描述
addCookie	void AddCookie(Cookie)	将一个 cookie 添加到这个响应的 HTTP 头部中
addDateHeader	void AddDateHeader(String name,long date)	将给定的名称与日期添加到一个响应的头部中

(续表)

方法	签名	描述
addHeader	void addHeader(String name,String Value)	将给定的名称与值添加到一个响应的头部中
addIntHeader	void addIntHeader(String name,int val)	将给定的名称与整型值添加到一个响应的头部中
containsHeader	boolean ContainsHeader(String header)	检查响应中是否包含有一个指定的头部
encodeRedirectURL	String encodeRedirectURL(String url)	对指定的 URL 进行编码,以便能在 SendRedirect 方法中使用。如果没有要求进行 URL 编码,那么 URL 将不会改变
encodeURL	String encodeURL(String url)	对指定的 URL 进行编码。如果不需要进行编码,那么 URL 将不会改变
sendError	void sendError(int statusCode)	利用默认的消息和指定的状态代码向客户发送一个错误响应
	void sendError(int statusCode,String message)	利用指定的消息和指定的状态代码向客户发送一个错误响应
sendRedirect	void sendRedirect(String url)	使用指定的重定向位置 URL 向客户发送一个临时性的重定向响应
setDateHeader	void setDateHeader(String header,long value)	利用指定的名称与值设置一个响应的头部
setHeader	void setHeader(String header,String value)	利用指定的名称与值设置响应的头部中的一个字段
setIntHeader	void setIntHeader(String header,int value)	利用指定的名称与值设置响应的头部中的一个字段
setStatus	void setStatus(int statusCode)	设置响应的状态代码

HttpServletRequest 封装了 HTTP 请求,而 HttpServletResponse 则封装了相应的响应。每个 do 方法都应该从请求读取数据,在响应中设置 HTTP 头部并且写出响应。通常被设置的头部包括内容类型以及内容长度(如果响应的长度已知)。通常使用输出流或打印输出程序写出响应体。需要说明的是,HTTP servlet 请求以及响应对象的许多方法都与 HTTP 会话、cookie 以及 URL 编码相关。URL 编码与 cookie 是用来维护 HTTP 会话的技术,它们自己也是为 servlet 创建上下文的机制。本书将在第 22 章详细讨论这些问题;为了保持讨论的完整性,笔者在这里给出的方法列表中将它们列了出来。有些响应方法与状态代码相关,这些状态

代码是作为 HTTP 协议一部分的数字代码。SC_OK 表示请求被成功地服务，其他代码则与错误相关，例如，SC_NOT_FOUND（表示请求的页面不存在）或 SC_UNAUTHORIZED（表示请求需要 HTTP 认证）。

get 操作与其他操作之间存在的一个重要的逻辑区别就是，get 操作需要是安全的。也就是说，这个操作对用户不应该有任何副作用。对于那些试图修改已存储数据的请求，应该使用其他的 HTTP 方法（例如 post 或 put）。

7.3 Java Server Page

JSP 页面是一个包含有 Java 代码段以及其他特殊 JSP 标记的文本文档。这个文档自身可以是任何格式（例如 HTML，XML 或任何其他语言）。因此，可以对任何其响应基于某个模板但需要有 Java 处理来创建实际结果的应用程序使用 JSP 页面。正如在 servlet 中提供了通用类但大多数 servlet 都是 HTTP servlet 一样，在 JSP 中大多数都是包含有 Java 的动态生成功能的 JSP HTML 页面。随着时间的推移，我们相信 JSP 也将被大量用来创建 XML 输出。

7.3.1 JSP 操作模型

JSP 操作模型基于 servlet 模型。读者已经知道，servlet 模型定义了如何对一个请求进行处理以便创建这个请求的响应，并且定义了哪些是可以使用的主要对象（请求、响应、输出流、内容类型等）。JSP 与此极为相似，它们也定义了请求被处理并得到响应的范型，并且 JSP 模型具有与 servlet 非常相似的主要对象集合。

激活一个 JSP 意味着需要将 Web 服务器指向文本文件（使用 .jsp 作为扩展名）；对于 JSP 1.1（它是 WebSphere Application Server 4.0 支持的 JSP 版本），这个文件可以是 XML 文件。HTTP 请求将被检查，并且 HTTP 请求是从其中创建请求对象（以及任何所封装的对象）的来源。请求可能来自 Web 客户、其他 servlet 或其他 Java 应用程序。核心 JSP API 定义了一个名为 sendRedirect 的方法，其他 servlet 可以从这个方法发出一个关于 JSP 的请求。在此之前，它们可以创建或修改请求对象，以确保在请求中放置了正确的参数（通过使用 setAttributes 方法）。当 JSP 后来查找某个已经命名的参数时，它就能够在这个请求中找到它们。

在请求被接收之后，处理过程就开始了。这个过程将读取该文本文件并开始处理标记。它的作用是创建响应流——因此，JSP 模型与 servlet 模型是非常相同的。处理过程完成这一动作的方法是，遍历这个文件并查找 JSP 标记。任何不是 JSP 标记的文本部分将被复制到输出流中。任何属于 JSP 标记的部分将被 JSP 引擎处理，并且通常将产生写入到输出流中的输出。其中有些标记并不直接写到输出流，而是影响输出流的输出外观——例如，可以使用 JSP 伪指令来设置输出流的内容类型。

JSP 的大多数用法都是在文本文件中嵌入 Java 代码以便产生动态数据。JSP 引擎能够提取这个 Java 代码并产生数据，这一过程是不同寻常的（至少当第一次接触到这种概念时更是如

此)。JSP 引擎为每个 JSP 文件按照 Java 代码出现的顺序产生一个 servlet。它将产生一个作为真正 servlet 的 .java 文件,然后将这个文件编译为 .class 文件。然后调用这个 servlet 从而产生输出流。HTML 将进入输出流,这是因为 servlet 会以文本形式读取 .jsp 文件并将某些段从这个文件中复制到 servlet 输出流。本书将在第 22~28 章详细说明这一过程并且说明所产生的 servlet。目前,很重要的一点就是必须了解这个事实,因为它完全解释了为什么 JSP 模型与 servlet 模型是如此相似——实际上它们是一个东西并且是相同的。

图 7-2 演示了在 JSP 引擎中发生的主要过程。显然,这种生成与编译过程是需要花费较长时间的,并且,如果每次在访问这个 .jsp 文件时都需要经历这些延时开销,那么是难以接受的。幸运的是,这一过程仅仅出现一次;然后,就能够快速而直接地读取已经编译好的 servlet (只要这个 .jsp 文件自从最后一次编译以来没有被修改)。

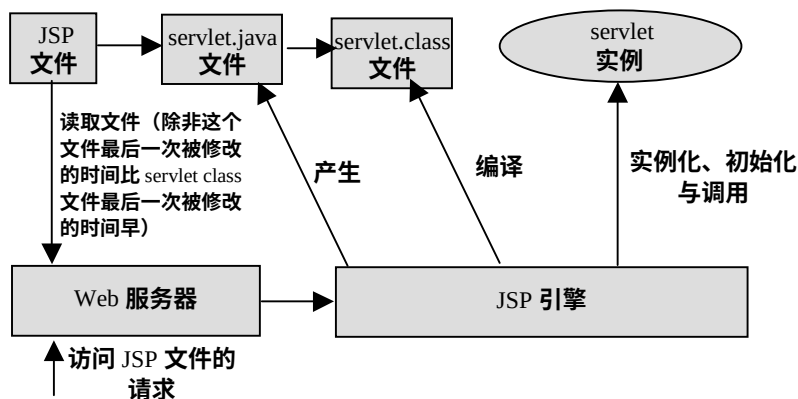


图 7-2 JSP 操作模型

一般来说,在 JSP 页面中使用大量的 Java 代码是不可取的。对于一个好的 JSP 页面,它将使用封装有行为的 bean (或自定义标记库,将在第 27 章讨论)而不是直接将 Java 代码嵌入到 JSP 页面中。

因为模型是相同的,并且实际上 JSP 被转换成 servlet,因此任何适合 servlet 的程序也同样适合于 JSP。例如,可以在文本文件中实现一个称为 `jspInit()` 的方法。如果这样做,当第一次对 JSP 的请求被接收后 (并且在 servlet 产生、编译与实例化之后),所实现的这个方法将会像 servlet 中的 `init` 方法那样在 servlet 实例化后被调用。需要说明的是,任何没有被特别地放置在一个方法中的 Java 代码,默认地属于 servlet 的 `service` 方法。

7.3.2 JSP 语法概述

.jsp 文件是包括 JSP 元素的某种格式的文本文件。JSP 语法定义了一系列可能的元素。任何不属于这些元素类型的代码将不会被 JSP 引擎处理,并且将毫无修改地传递给结果页面。JSP 元素包括:

- scriptlet
- 表达式
- 注释
- 声明
- 动作
- 伪指令
- 隐式对象

对于大多数元素，都有两种不同的格式可用。第一种格式称为 JSP 语法。类似于其他脚本语言，在这种格式中，脚本元素被包围在符号`<%与%>`内部。另一种格式就是标准化 XML 格式 (standardized XML format)，它比较麻烦；脚本元素被包围在符号`<>与</>`内部，例如：`<jsp:element>`与`</jsp:element>`。

Scriptlet

JSP 包括一个很方便的语法，在这个语法中，Java 代码能够写入到称为 scriptlet 的元素中。通过将 Java 代码包围在`<%与%>`中，用来告知 JSP 引擎将这一段代码复制到 servlet 处理方法中（也就是说，如果没有包括任何伪指令，就复制到 service 方法中，或者复制到方法伪指令所定义的方法中）。

在 XML 格式下，这个代码的包围形式如下：`<jsp:scriptlet>`与`</jsp:scriptlet>`。

表达式

一个类似的语法可用来定义表达式，此表达式将被 JSP 求值并写出到输出流中。与 scriptlet 语法的区别在于，表达式的打开标记后面跟随有一个等号（例如`<%=opening tag`）并且所嵌入的代码被当作一个表达式。由于这个表达式被嵌入在 Java 语句中用来执行向输出流的写出动作，因此这个表达式不能用分号（`;`）结束。

在 XML 格式下，表达式的代码包围形式为：`<jsp:expression>`与`</jsp:expression>`。

注释

JSP 文件中的注释与 HTML 中的注释模式相同。通过嵌入一个形式为`<!-- comment-->`的行，就能够向浏览器页面传递注释。另一种注释就是出现在 JSP 中但并不传递给结果页面的隐藏注释 (hidden comment)。可以使用语法`<%-- comment--%>`来嵌入这样的注释。需要注意的是，HTML 与 XML JSP 使用了相同的注释语法。

声明

声明元素用来声明变量或方法。声明元素的语法以标准的 JSP 开标记开始，后面紧跟一个感叹号：`<%!`。在 XML 格式下，声明的代码包围形式为：`<jsp:declaration>`与`</jsp:declaration>`。

包围在声明之中的所有变量与方法将被添加到由 JSP 所产生的 servlet 类中，因此它们在这

个类的任何地方可用。

动作

动作是 JSP 引擎可以使用的预定义函数。一个动作元素可以修改输出流，也可以创建与使用对象。只有一种语法可用于 JSP 动作，这个语法遵守 XML 语法规则。它的格式是 `<jsp:action_name/>`。按照 XML 的约定，动作也可以有属性，这些属性被放置在动作名称的后面，格式为 `attribute="value"`。

动作元素的功能非常强大，除了具有其他优点之外，动作元素还能够用于方便访问 JavaBean 以及包括其他 URL。本书第 25 章将详细讨论各种动作。

伪指令

伪指令是传递给 JSP 引擎的指令。伪指令的格式为 `<%@ directive_name %>`。相应的 XML 语法是 `<jsp:directive.directive_name/>`。关于伪指令的进一步讨论请参见第 25 章。

隐式对象

将 JSP 转换为 servlet 的过程定义了几个不必进行显式声明就能够引用的对象。这些对象形成了 servlet “环境”的一部分。其中一些隐式对象如下：

- out：输出流；javax.servlet.jsp.JspWriter 的一个实例。
- request：HTTP 请求；javax.servlet.http.HttpServletRequest 的一个实例。
- response：HTTP 响应；javax.servlet.http.HttpServletResponse 的一个实例。
- config：servlet 配置；javax.servlet.ServletConfig 的一个实例。

7.4 小 结

servlet 与 JSP 是任何在 WebSphere 上开发与部署的 Web 应用程序的主要部分。至今为止，它们是最常用的结构，并且它们易于理解、使用简单。但它们的易用性并没有损害其功能的强大性——它们是功能非常强大的方法，应该被所有的 Web 开发人员所掌握。在构建 Web 应用程序过程中的另一个重要组件就是 Enterprise JavaBean；本书将在第 8~11 章详细描述 EJB 及其通信底层结构。

第 8 章 公共对象请求代理体系结构

本章将介绍 CORBA (Common Object Request Broker Architecture, 公共对象请求代理体系结构) 的基本原理。将在这里描述基于 CORBA 的系统中的公共组件及其作用。下一章将讨论 CORBA 2.0 并且主要关注 ORB (Object Request Broker, 对象请求代理) 以及 IIOP (Internet Inter-ORB Protocol, Internet ORB 间协议)。

在一些方面, CORBA 与 IIOP 是相对于 EJB 而言的。例如, 在 EJB 1.1 版本的规范中, 要求 EJB 服务器支持 RMI-IIOP (Remote Method Invocation-IIOP) (在 IIOP 上实现的 Java RMI)。通过使用 RMI-IIOP (不同于 EJB 1.0 规范的普通的 Java RMI), 这个规范就打开了与遗留的系统进行互操作的大门。利用 CORBA, 就使得那些使用非 Java 语言 (例如 COBOL, C++) 开发的软件能够继续工作。CORBA 的设计影响了 EJB 的设计, 并且 EJB 以及 J2EE 规范中的大量思想都来自 CORBA。最后, WebSphere 在 IIOP 上实现 EJB 通信, 并且 WebSphere Application Server 企业版也是基于 CORBA 来开发的。

8.1 对象管理组

在 20 世纪 80 年代后半期以及整个 20 世纪 90 年代, 分布式对象成为大多数 IT 组织的主要技术。对分布式对象的需求有 3 个开发方面的原因: 对连接以及计算的可靠网络要求、对服务与协作的远程访问需求、Internet/Web 的巨大变革。虽然每个这样的开发都出现在不同的时间以及出现在不同的技术客户, 但它们都需要对对象、数据以及服务的分布式访问, 以便能够构建功能强大的应用程序。

虽然集成 Web 上独立应用程序以及服务的要求进一步推动了对分布式对象的需求, 但分布式对象并不是从这里开始被使用。实际上, 当 Web 现象最终被认为将会是 IT 新增长点的主要领域, 并且将作为解决 Web 应用程序问题的解决方案时, 对分布式对象的解决方案才准备成熟。CORBA 是 OMG (Object Management Group, 对象管理组) 开发的分布式面向对象体系结构的业界标准, 它已经完全定义了运行于不同平台并且使用不同开发环境与不同编程模型的应用程序应该如何互操作。在许多方面, CORBA 都可以看成是一种全球性的对象中间件——能够非常好地满足 Internet 应用程序的需求。

成立 OMG 的目的, 就是能够在软件业界进行工作, 提供一种开放的体系结构来支持不同厂商的、全球的、基于对象的软件组件的异构网络。OMA (Object Management Architecture, 对象管理体系结构) 指南已经发布, 作为形成对象系统未来的详细规范与底层结构的基础。OMA 定义了许多组件, 它们共同使得能够实现 OMG 所规划的蓝图。在 OMA 中定义的最

重要体系结构规范部分就是 ORB。这个组件是促进完成 OMA 工作的主要机制，它用来在不同的对象和组件之间发送与接收消息。ORB 环境在 CORBA Specification 中定义。

CORBA Specification (1.1 版本) 是在由 Digital Equipment, Hewlett-Packard, HyperDesk, NCR, Object Design 以及 SunSoft 等公司提交的建议基础上形成的。它作为 OMG Document Number 91.12.1 出版，并且作为 OMG 与 X/Open 出版的书籍被发行，它是第一个广泛可用的描述被吸收的 ORB 技术的规范。1995 年中期发布了这个规范的 2.0 修改版并且在 1996 年中期进行了更新。

8.2 对象管理体系结构

对象管理体系结构 (OMA) 是一种通用的体系结构，并且是一个后无可移植环境与可互操作环境的必要组件的分类。OMA 列出了这个体系结构中每个组件都应该遵守的通用技术准则。包括必要的面向对象接口、分布式透明性、作为所有组件基础的通用对象模型、对软件生命周期所有阶段的完全支持、灵活的动态特性、高性能、健壮的实现，以及遵守软件业界的已存在标准。

8.2.1 OMA 参考模型

图 8-1 显示了 OMG 提供的 OMA 参考模型。这个参考模型提供了一种底层框架，用来指导 OMG 技术吸收过程。它定义了为实现 OMG 的目标应该采用的必要组件分类。在进行组件开发的过程中，可以使用这个模型提供指导，指出必须如何进行组件开发。一旦定义了这个体系结构，OMG 的工作就是为在这个模型中定义的组件提供详细的规范。

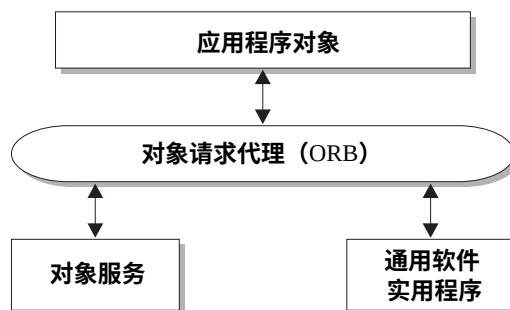


图 8-1 OMG 提供的 OMA 参考模型

如图 8-1 所示，这个模型由 4 个组件类型组成：ORB、对象服务、通用软件实用程序以及应用程序对象。其中，应用程序对象组件代表实际被开发的、用来解决特定领域问题或用来提供现货供应产品的软件。这些对象使用其他 3 个组件类型，并且为它们提供丰富的开发

环境。使用这其余的 3 个组件类型，应用程序开发人员能够快速创建可移植的互操作代码，这些代码可以被其他应用程序组件所重用。从这一角度来看，整个体系结构的目标就是提供尽可能好的关于这类对象的开发环境。

这个体系结构的核心部分是 ORB，它提供了允许对象交互并且共同工作以便达到共同目标的能力。对象服务提供了低级的系统服务，这些系统服务对于开发应用程序是必要的，例如对象一致性、事务能力、安全性。对象服务定义了一个服务集合，这些服务对于任何应用程序都是必要的，从而使得能够在一个比较高级的层次上进行应用程序开发，不需要开发人员为每个应用程序都重新考虑这些服务的实现。通用软件实用程序提供了更高层次的服务，它在语义上与应用程序对象更加接近，例如邮件处理软件实用程序与打印服务软件实用程序。通用软件实用程序还提供了一个更高层次的功能，用来定义许多应用程序所需要的普遍功能。

对象服务与通用软件实用程序之间的区别比较模糊；对象服务强调的是“面向系统”，而通用软件实用程序强调的是“面向应用程序”。另一个重要的区别是，对象服务必须被所有的 ORB 环境在所有的平台上提供支持，而对通用软件应用程序的支持则是自由决定的。

OMA 由对象组成，并且这个体系结构中的每个组件都被定义成提供面向对象的接口（虽然它并不一定要求采用面向对象的方法来实现）。对象通过访问面向对象的接口从任何其他对象请求服务。因此，任何对象都能够为任何其他对象提供服务。这个体系结构比客户/服务器体系结构更加优越，因为每个对象都潜在地可以是客户也可以是服务器。实际上，每个请求调用都在单独一个调用的上下文中定义了一个客户与一个服务器——相同的对象可能被要求为一个请求提供服务（充当服务器），而实现又需要调用一个请求来从另一个对象获得某些信息（充当客户）。这个对象（包括应用程序对象、对象服务对象、通用软件实用程序对象）都将按照这种关系被调用，既可以充当客户又可以充当服务器。

交互使用面向对象的接口。实现（包括非面向对象的实现以及“遗留”软件）都可以被设置为提供面向对象的接口，这常常称为面向对象的封装器或适配器。按照这种方式，OMA 就能够为所有的软件组件（包括那些先前已经开发的软件组件以及那些不是面向对象的软件组件）提供一个底层模型。此外，由于任何非分布式应用程序都是分布式应用程序的一个特例（仅仅使用一台机器），因此这个体系结构能表示所有的系统体系结构。

8.2.2 对象请求代理

对象请求代理是使得这个参考模型能够运作的核心组件。它为任何组件进行通信提供了基本的、必要的对象交互能力。因此，毫无疑问，OMG 应该首先定义这个组件。它允许客户对象发出请求，要求其他对象提供所请求的服务。这些服务请求是由客户以一种独立于位置、语言与平台的方式发出的。因此，ORB 是实现互操作性与可移植性的主要驱动者，也是创建真正的分布式应用程序这一能力的使能者。

对象使用一个由服务提供者所定义的面向对象的接口发出请求。客户通过指定请求将要被

送达的对象的引用、操作名称以及任何其他必要的信息来构建请求。这时还通常包括将要被发送的参数，但也可以包括环境上下文这样的信息。

8.2.3 对象服务

ORB 提供了基本的交互能力，但对于开发实际的应用程序这是远远不够的。应用程序需要假设已经能够从底层环境中获取基本的服务与功能；这是对象服务层起作用的地方。例如，ORB 提供了将请求传递给服务提供者的方法，但并没有定义如何找到这个服务提供者。

对象服务层对于 OMG 参考模型的成功非常重要。如果没有为这个层提供足够的功能，那么 OMG 的蓝图就无法实现——即使 ORB 具有非常强大的功能并且具有很好的可用性，但这样并不能使得创建应用程序成为现实。这是因为，应用程序自己需要进行大量的开发工作，毕竟，ORB 所提供的支持级别太低。大多数应用程序开发人员也缺乏自主开发这些功能所需的专业经验。

因此，对象服务是 OMG 参考模型的主要组件。实际上，一旦有了可以使用的基于 ORB 的体系结构，OMG（通过 Object Services Task Force）就会开始为这一层提供服务。下面列出了一些这样的服务：

- Naming Service（命名服务）：允许通过命名查找 CORBA 对象。
- Event Notification Service（事件通知服务）：用来实现 CORBA 对象之间的异步通信。
- OTS（Object Transaction Service，对象事务服务）：使得 CORBA 对象能够执行事务工作。这个服务有一个较低的层，用来实现事务的结构；还有一个较高层次的接口，用来定义事务的范围与作用域。OTS 构成了 JTS（Java Transaction Service，Java 事务服务）的基础，JTS 是 J2EE 的关键部分。
- Concurrency Control Service（并发控制服务）：允许多个客户通过锁并发地与单独一个资源进行交互。

按照将接口从实现中分离出来这一普遍接受的思想，对象服务仅仅定义了接口并且提供一种关于功能的语义规范。因此，服务定义就为开发人员提供了一个 API 集合以及所提供的功能行为的描述。它并没有为这个服务施加任何与实现相关的限制。实际上，应用程序应该假设为每个服务都提供许多实现。这是完全必要的；因为服务通常比较复杂，并且可能有不同的特点。强调服务的某一个特点常常会导致对另一特点支持的不足。因为服务的用户将可能会要求各种不同的折中方案，因此提供多种实现是必要的。这就涉及 OMG 引入的 QoS（Quality of Service，服务质量）问题，它允许不同的实现强调关于所提供的服务的不同属性。

在 WebSphere 4.0（以及 3.5.x）中，对象服务是由 adminServer 实现的。

8.2.4 通用软件实用程序

通用软件实用程序也提供服务，但它们通常在更高的层次上提供服务。它们比对象服务更接近于应用程序。通用软件实用程序的例子包括混合文档管理、电子邮件、帮助系统。这些功能仍然属于 OMA 的范围；它们可以被许多应用程序使用，并且能够允许快速地以一种可移植、可互操作的方式创建应用程序。与对象服务一样，它们减少了在 OMG 框架中编写软件的复杂性，从而达到 OMG 所定义的总体目标。实际上，由于通用软件实用程序服务所在的层次更高，它们常常比低级的服务涵盖了应用程序的更多功能。例如，一个混合文档框架就可能比支持通用对象图形关系的服务更可取。

8.3 CORBA

基于 ORB 的体系结构支持可移植性与可互操作性。图 8-2 演示了一个客户正在请求将要由对象实现提供的服务。客户已经有了一个对象引用并且激活了这个对象的操作。对象的引用是不透明的，并且客户没有任何关于对象实现的位置、用来实现这个对象的语言以及任何关于实现的其他详细信息。它只有一个关于操作请求的规范，这个规范是接口的一部分。客户发出请求，ORB 负责对对象实现能够响应的格式将请求传递给对象实现。

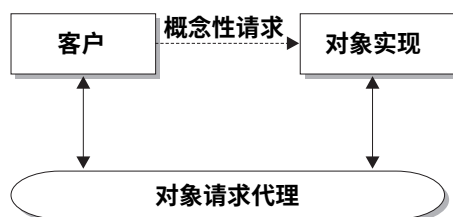


图 8-2 客户请求与对象实现

当客户有了关于对象实现的句柄并且希望发出一个请求时，有两种可能的选择。如果已经使用 IDL (Interface Definition Language, 接口定义语言) 定义了接口并且客户有了关于对象实现类型的定义，那么将发出一个静态调用。这个请求是使用这个目标对象接口特定的 IDL stub 产生的。否则，就可以使用 DII (Dynamic Invocation Interface, 动态调用接口)。这个接口独立于目标对象的实现，因此，即使没有关于目标对象实现的特定知识，也可以使用这个接口。

请求是在运行时构建的，并且没有使用 IDL stub。相反，请求由动态调用模块使用存储在接口库中的信息所处理。图 8-3 演示了两个可能的请求路径。需要说明的是，当请求被传递给对象实现时，它并不知道也并不关心将采用哪条路径。

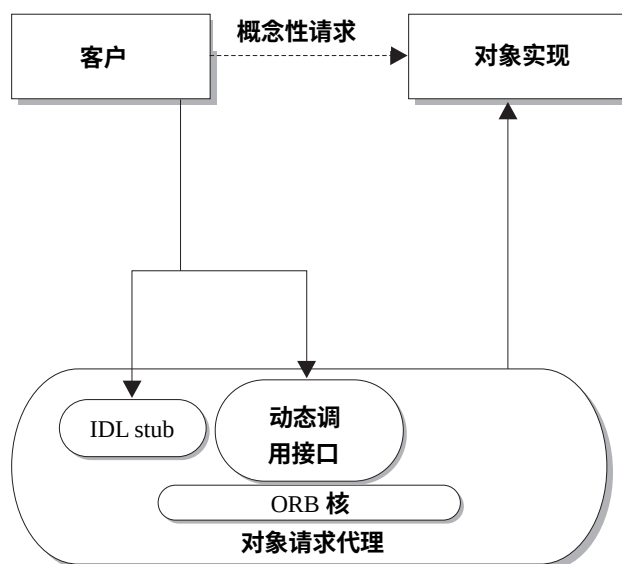


图 8-3 调用一个请求的两条路径

图 8-4 显示了请求传递到对象实现的过程。在定义了使用 IDL 的对象的接口之后，将生成一个 IDL 骨架 (skeleton)。请求将通过这个骨架从 ORB 进行传递。无论请求是通过 IDL stub 还是使用 DII 发出，都将会使用一个 IDL 骨架。通过使用存储在安装时所提供的实现库中的信息，对象实现变为可用的。

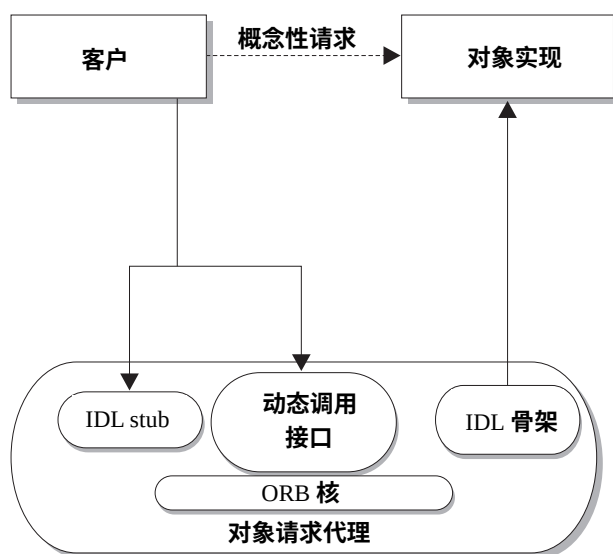


图 8-4 将一个请求传递给对象实现

CORBA 定义了许多 ORB 子组件，这些组件能够相互交互，从而支持所需的功能。图 8-5 演示了一个关于这些组件的简单示意图。这个示意图的目的是给出关于 ORB 的分解说明。实际上，CORBA 并不需要进行任何与 ORB 实现有关的假设或需求。CORBA 仅仅定义了每个 ORB 必须提供的接口。



图 8-5 ORB 中的子组件

8.3.1 ORB 核

ORB 核处理对各种组件的基本通信请求，它可以看作是底层的传输层。CORBA 是为了支持多个系统而设计的，因此被分为一系列模块化组件，从而使得能够使用不同的实现。这样就允许不同的 ORB 核被相同的层所使用（而且对外部组件隐藏这种不同），同时也允许不同层的组件使用相同的核。这个核提供了请求的基本对象表示与通信。

8.3.2 IDL stub 与骨架

IDL 用来定义接口、类型、结构与模块。通过接口的定义，客户就能够被告知对象实现所支持的是哪些服务。在 IDL 中定义的每个接口指定了操作签名，通过这种操作签名产生请求。需要说明的是，在运行时并不需要 IDL 源代码。IDL 定义用来创建 stub 与骨架，并且用来扩充接口库。

客户从他们自己主机的编程语言发出请求。IDL 是一种定义语言，而 CORBA 并没有定义处理语言或调用语言。为了允许用 IDL 编写的接口被从编程语言中使用，需要进行映射（mapping）。映射将提取一个 IDL 定义，并且为每个接口在本地编程语言中生成一个 stub。这些 stub 允许编程者调用被定义为目标对象接口一部分的操作。

系统将为每个接口产生相应编程语言的实现骨架。这些结构实际地调用作为对象实现的一部分而实现的方法。之所以被称为实现骨架，是因为它们是从 IDL 定义中创建的，并且没有包括任何实现细节。对象实现的开发者必须利用在请求被接收时所调用的实际代码来填充骨架。

IDL 骨架通常依赖于对象适配器。相同的接口以及相同的语言可能存在多个骨架——对应不同的对象适配器。甚至有可能提供不使用 IDL 骨架而是当请求到达时“实时地”创建实现的对象适配器。

DII 允许动态创建请求，这样就使得客户能够直接指定对象、操作以及一系列参数并调用这

个请求。所使用的接口对所有对象与所有操作都是共同的，并且它不使用为每个操作在每个接口中所产生的 stub 例程。关于参数以及操作自身的信息通常从接口库中获取。

8.3.3 对象适配器与 ORB 接口

对象适配器提供了一个被对象实现使用的到达 ORB 服务的接口。它也提供了一些服务，当从 ORB 核接收到一个请求，并且这个请求被传递给骨架进行方法调用时，可能需要这些服务。根据对象实现的特点不同，对象实现可以有非常大的差别，因此出现了许多对象适配器。这样就使得能够支持各种各样的情况，但同时又不修改体系结构。

对象实现所使用的其他服务是通过 ORB 接口直接提供的。这个接口对所有 ORB 实现都相同，并且不依赖于对象实现所使用的对象适配器。很自然，这个接口也将被映射到主机的对象实现所使用的编程语言上。这些操作对客户对象也同样可用。

8.3.4 接口库

IR (Interface Repository, 接口库) 维护了关于 IDL 定义的代表。它们是在运行时可用的一致对象。因此，IDL 信息被维护为可以在运行时使用的“活对象”。例如，DII 使用在 IR 中维护的信息来允许程序在编译时发出关于那些不认识的接口的请求。IR 服务也被 ORB 所使用。例如，在执行对参数值的整理与取消整理动作时，就需要用到 IR 信息。IR 也被用作与接口相关的附加信息（例如注解、调试信息等等）的一致存储地。

IR 维护了 ORB 定位与启动对象实现完成某一请求所必须的信息。它也被用于与对象实现相关的附加信息，例如调试信息与安全性规范。IR 是特定于操作环境的，这是因为它在对象实现的构建与活跃过程中被使用，并且它没有被 CORBA 标准化。

8.3.5 对象引用

对象引用 (object reference) 是一个不透明的表示，用来在基于 ORB 的环境中表示对象。客户与对象实现都使用对象引用。ORB 提供了从对象引用到所用编程语言的映射；这种映射必须不依赖于对象引用的 ORB 表示。

CORBA 定义了一个特别的对象引用，用来表示不存在的对象。这个引用已经被确保不同于任何其他对象引用。在有些环境下，它被映射到一个 null (或 nil) 对象的引用。

8.3.6 客户

客户 (client) 从应用程序代码中发出请求。客户代码就像使用任何库例程一样使用 IDL stub。当这个 stub 例程被调用时，关于目标对象的对象引用将被映射到 ORB 所代表的对象引用。

然后，这个 stub 将调用 ORB，由 ORB 负责定位对象实现并将请求路由到这个实现，然后再将任何结果返回给客户。如果在这个方法调用过程中出现了错误，或由于任何原因导致 ORB 不能完成所请求的任务，那么将有一个异常被传递给客户的操作上下文。

如果目标对象的接口完成之前对客户进行了编译，那么就有可能导致目标对象的 stub 不可用。因此，客户代码就不能访问这个对象的接口。在这种情况下，客户可以给出将要被调用的对象和操作的类型名并使用 DII。客户可以使用用来构建调用和参数的调用序列，通过编程的方法构建这个调用，然后调用请求。

客户使用对象引用来发出请求，并且使用对象引用作为参数值。客户通常接收到作为以前的请求的输出参数的对象引用，或者接收到通过请求从其他客户发送到本客户的对象引用。对象引用甚至能够被转换为一个字符串格式，从而被存储在磁盘上以持久保存，以后，这个字符串又能够被相同的 ORB 再次转换回对象引用。

8.3.7 对象实现

对象实现 (implementation) 封装了关于对象的状态与行为。这些对象实现是对象的内部机制，但它们的行为被骨架用来为客户提供服务。CORBA 仅仅定义了完成调用操作的必要机制，它并没有特别地定义这些对象如何被激活、终止、保持，也没有定义访问控制如何处理。但是，由于这些对象实现在真实的环境中必须工作，因此所有这些问题都必须得到解决。在实际中可以自由决定与这些问题相关的实现。

8.3.8 对象适配器

对象适配器 (adapter) 是为对象实现提供的与 ORB 相关的主要功能。ORB 自己也提供了可能被对象实现所使用的接口，但对象适配器提供了对象实现常用的大多数与 ORB 相关的服务。对象适配器提供了一个被所有对象实现所使用的 public 接口，也提供了一个接口骨架所使用的 private 接口。适配器使用与 ORB 相关的接口。

下面是对象适配器所提供服务的例子：

- 与骨架共同用于方法调用。
- 对象实现的激活与去活 (deactivation)。
- 从对象引用到对象实现的映射以及对象引用的生成。
- 注册对象实现，从而使得在接收到请求时能够找到这个对象实现。

CORBA 并不要求所有的对象适配器都支持相同的功能，也不要求它们具有相同的接口。对象适配器可以被特殊化，因为它们是对象实现的主要服务提供者，同时也是请求传递机制中的主要作用者。CORBA 规范仅仅定义了 BOA (Basic Object Adapter, 基本对象适配器)。任何与 CORBA 兼容的实现都必须提供一个 BOA。但是，CORBA 规范并没有对附加的对

象适配器的定义施加任何限制；实际上，这个规范还提供了一些例子，在这些例子中说明了使用附加的对象适配器还可能是很有用的。此外，由于对象实现依赖于对象适配器的接口，因此并不推荐在系统中存在大量的对象适配器。当然这并不是一个严格的限制，因为只有当需要有不同的服务或需要有不同的服务质量时，才会用到不同的对象适配器。这样，每个对象适配器通常都能够支持一个较大的对象实现集合。

在 CORBA 规范中给出的一个例子就是 LOA (Library Object Adapter, 库对象适配器)。BOA 为每个请求启动一个处理线程，或者，甚至为每个对象调用启动一个单独的进程。这对于可能被多个客户共享的库实现是不可接受的。此外，库对象可能实际地在客户的上下文中运行，从而使得不需要进行激活与认证，这样有利于极大地提高性能。如果这些对象实现是通过 BOA 支持的，那么上述所有的优化都不可能进行。通过提供 LOA，能够达到优化行为的目的。

8.4 接口定义语言

CORBA 使得能够开发可移植、可互操作的应用程序。它允许使用多种编程语言，并且使得用一种编程语言所编写的对象能够使用由另一种语言所编写的对象成为可能，使用户能够利用这种方式构建系统。这一工作由 ORB 透明地处理，参与工作的对象并不知道其余对象可能使用的是另一种语言这一事实；CORBA 则是这些对象世界的统一转换者。

在不同的编程语言之间转换对象与调用可以被看作是某种意义上的桥 (bridging) ——在这里，桥涉及将一种编程语言的对象模型转换到另一种编程语言的 (相同) 对象模型这一工作。桥可能涉及两个基本的体系结构：直接 (immediate) 桥与间接 (mediated) 桥。在编程语言互操作性方面，直接桥意味着将每种相关的编程语言直接转换到其他每种相关的编程语言。间接桥则引入一个单独的语言 (并且具有运行时格式)，这种语言描述了对对象模型并且被用来作为每种其他编程语言转换成的中间格式。

CORBA 的最初目标是对任何语言都可以使用，并且能够支持大量编程语言的互操作性。目前，已经存在允许 CORBA 实现 C, C++, Smalltalk, Ada, Objective-C, Java 以及 COBOL 之间互操作性的产品。由于存在大量的编程语言 (并且越来越多)，编程语言可能会改变，不同开发环境可能支持某种语言的稍有不同的变体，这些事实使得不可能支持编程语言的直接桥。

这样，间接桥就是唯一可能的解决方案。但应该使用什么中间语言呢？IDL 就是一种语言，客户使用这种语言来指定发出请求的接口。对象实现提供了方法，这些方法提供接口所定义的必要服务。利用 IDL 语言定义接口，并且这些接口被客户程序所使用。IDL 并不用来编写代码；而只是用来定义规范。因此，客户将不会用到 IDL；它将使用 IDL 接口规范，但实际的调用将要求用自己的编程语言进行书写。这些调用将是 IDL 接口映射到编程语言的结果。对象实现也不能用 IDL 编写，而是用一种 (也可能是另一种) 编程语言编写。然

后，接口操作将映射到实现方法。

8.4.1 CORBA IDL 基础

对 IDL 进行映射涉及在客户方创建 stub 并且在服务器方创建骨架。stub 是编程语言函数(或方法)，它可能被客户代码所调用。调用这样的 stub (使用 CORBA 对象进行调用) 将会导致一个调用动作的发生，即使当实际的对象是远程对象并且该对象由另一种不同的编程语言所实现时也是如此。所有这些动作对客户都是透明的。当发生这样的调用时，就会调用对象实现骨架。这个骨架是在 IDL 映射到对象实现的编程语言时所创建的。由于这样的调用要求执行应用程序的特定行为，因此，开发人员必须为每个作为 IDL 映射结果的骨架提供编程语言实现。所有这些工作都是在对象实现开发人员所使用的普通开发环境中完成的。

正如在 CORBA 规范中定义的那样，从 IDL 到主机本地编程语言的映射必须包括下列每一个动作：

- 所有 IDL 数据类型、常量、对象引用的映射。这包括提供一种用目标编程语言表示的基本数据结构、结构化数据结构以及常量的方法。这个映射过程也必须指出如何在目标编程语言中表示对象。由于这个映射过程必须照顾到主机上编程语言的语义，因此对象表示并不依赖于 ORB 表示。这个表示可以使用一个对 ORB 表示不透明的句柄，或者表示成为真实的编程语言对象。对于第二种情况，映射必须能够支持完全相同的关于继承、操作签名等对象语义。
- 操作签名与调用的映射。这个映射包括输入参数如何被处理、返回值如何被处理、操作如何映射到编程语言函数等等定义。这个映射必须不仅定义 IDL 结构如何被映射，也需要定义它们如何被使用。这涉及处理内存管理与生命周期的任务。
- 异常模型的映射。如果目标编程语言异常处理机制（当这种机制存在时）正在被使用，那么这种映射就应该描述此异常处理机制如何实现 IDL 异常。否则，这个映射就应该描述如何对操作进行扩展以支持异常并且说明需要哪些管理函数。
- IDL 属性的映射。属性被两个 accessor 操作（也就是 get 函数与 set 函数）所建模。
- ORB 接口与其他伪接口的映射。这一过程是必要的，以便允许编程者使用目标编程语言来访问由 ORB 或相关的组件之一（接口库、对象适配器、动态调用接口等）所实现的接口。一种解决方法是将它们定义为一组被打包成库的服务。另一种解决方法则是使用伪对象。伪对象有一个用 IDL 语言定义的接口，但并没有实现为 ORB 管理的对象。只能使用 SII (Static Invocation Interface, 静态调用接口) 来调用操作，而 ORB 可能会截获这种调用并处理它们。在这种解决方法中，ORB 接口将被实际地定义为 IDL 接口。这些接口将作为伪对象的一部分。这样做的优点是，使用为应用程序接口开发的相同机制，ORB 接口可以被映射到许多编程语言。因为它仅仅是

另一种接口，因此能够被映射过程所处理，并且对于 ORB 接口，这种处理过程是默认的。

IDL 的实际语法在某种程度上类似于 C++。下面的程序清单显示了一个关于 IDL 文件（HelloWorld.idl）的简单例子。需要注意的一个小问题是，CORBA 方法签名要求为每个参数指定值是 in，out 还是 inout。这些值分别用来表示此参数是输入参数、输出参数还是输入输出参数。这些语义在 Java 中并不支持；因此，在将 IDL 映射到 Java 的过程中会稍微复杂一点，下面一节将详细讨论这个问题。

```
Module HelloWorld
{
    interface Hello
    {
        string getGreeting();
    };
};
```

8.4.2 CORBA 到 Java 映射

为了演示如何在 Java 中使用 CORBA，考虑刚刚显示的‘Hello World’IDL。为了执行 CORBA 到 Java 的映射，需要使用一个称为 idltojava 的程序，读者可以从 JavaSoft Web 站点获取这个程序。

在下载这个程序之后，运行下列命令：

```
idltojava -fno-cpp HelloWorld.idl
```

需要说明的是，-fno-cpp 标志并不是必要的；它仅仅用来避免使用 C 预处理器对 IDL 进行预处理，用户可能没有安装这个预处理器。

idltojava 程序将产生 5 个不同的 Java 文件。它将自动产生一个名称与 CORBA 模块相同（在本例中就是 HelloWorld）的 Java 包。在这个包中，为每个用 IDL 语言定义的 CORBA 接口创建了一个单独的接口。在这个例子中，下面列出的 Hello.java 文件将被生成。

```
package HelloWorld;
public interface Hello
    extends org.omg.CORBA.Object, org.omg.CORBA.portable.IDLEntity
{
    String getGreeting();
}
```

下面描述了为每个 CORBA 接口生成的其他文件。

- 客户 stub 代码 被客户代码所使用；在本例中就是 _HelloStub.java，如下所示：

```
package HelloWorld;
```

```

public class _HelloStub
    extends org.omg.CORBA.portable.ObjectImpl
    implements HelloWorld.Hello {

    public _HelloStub(org.omg.CORBA.portable.Delegate d) {
        super();
        _set_delegate(d);
    }

    private static final String _type_ids[] = {
        "IDL:HelloWorld/Hello:1.0"
    };
    public String[] _ids() { return (String[]) _type_ids.clone(); }

    //          IDL operations
    //          Implementation of ::HelloWorld::Hello::getGreeting
    public String getGreeting()
    {
        org.omg.CORBA.Request r = _request("getGreeting");
        r.set_return_type(org.omg.CORBA.ORB.init().get_primitive_tc(
            org.omg.CORBA.TCKind.tk_string));
        r.invoke();
        String __result;
        __result = r.return_value().extract_string();
        return __result;
    }

};

```

- **服务器骨架代码** 用来实现对象的抽象类；在我们的例子中就是_HelloImplBase.
java，如下所示。在这个例子中，对象开发人员需要扩充这个类以实现 getGreeting
方法。

```

package HelloWorld;
public abstract class _HelloImplBase extends
org.omg.CORBA.DynamicImplementation implements HelloWorld.Hello {
    // Constructor
    public _HelloImplBase() {
        super();
    }
    // Type strings for this class and its superclass
    private static final String _type_ids[] = {
        "IDL:HelloWorld/Hello:1.0"
    };
};

```

```

        public String[] _ids() { return (String[]) _type_ids.clone(); }

        private static java.util.Dictionary _methods = new
java.util.Hashtable();
        static {
            _methods.put("getGreeting", new java.lang.Integer(0));
        }
        // DSI Dispatch call
        public void invoke(org.omg.CORBA.ServerRequest r) {
            switch (((java.lang.Integer)
_methods.get(r.op_name())).intValue()) {
                case 0: // HelloWorld.Hello.getGreeting
                {
                    org.omg.CORBA.NVList _list = _orb().create_list(0);
                    r.params(_list);
                    String _result;
                    _result = this.getGreeting();
                    org.omg.CORBA.Any _result = _orb().create_any();
                    _result.insert_string(_result);
                    r.result(_result);
                }
                break;
                default:
                    throw new org.omg.CORBA.BAD_OPERATION(0,
org.omg.CORBA.CompletionStatus.COMPLETED_MAYBE);
            }
        }
    }
}

```

- **helper 类** 一个带有用来读取、写入以及收缩 (narrowing) 对象引用的静态方法的类。收缩在本质上是一种类型映射，这个动作在检查对象引用具有正确的类型之前发生。本例中的 helper 类是 HelloHelper.java，如下所示：

```

package HelloWorld;
public final class HelloHolder
    implements org.omg.CORBA.portable.Streamable{
    //          instance variable
    public HelloWorld.Hello value;
    //          constructors
    public HelloHolder() {
        this(null);
    }
    public HelloHolder(HelloWorld.Hello _arg) {

```

```

    value = _arg;
}

public void _write(org.omg.CORBA.portable.OutputStream out) {
    HelloWorld.HelloHelper.write(out, value);
}

public void _read(org.omg.CORBA.portable.InputStream in) {
    value = HelloWorld.HelloHelper.read(in);
}

public org.omg.CORBA.TypeCode _type() {
    return HelloWorld.HelloHelper.type();
}
}

```

- **holder 类** 为一个单独的对象引用实现一个 wrapper，它可以被流化。这个 wrapper 的主要作用是实现 CORBA 的 out 参数与 inout 参数的语义，这是不容易映射到 Java 的部分。这就意味着，如果传递对象的引用到其签名被定义为 out 或 inout 的方法，就必须传递这个 holder，而不是传递实际的对象引用。在我们的例子中，holder 类是 HelloHolder.java，如下所示：

```

package HelloWorld;

class HelloImpl extends _HelloImplBase
{
    public String getGreeting()
    {
        return "Hello world!\n";
    }
}

```

为完成这个例子的实现，需要提供关于这个对象的完全实现。这个工作比较容易，因为只要使用一个方法。其实现在 HelloImpl.java 中完成，如下所示：

```

package HelloWorld;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;

public class HelloServer {

    public static void main(String args[])
    {

```

```
ORB orb = null;
try{
    // Create ORB and our object
    orb = ORB.init(args, null);
    HelloImpl hello = new HelloImpl();

    // Connect new object to ORB
    orb.connect(hello);
    // Access the nameserver
    org.omg.CORBA.Object nsRef =
        orb.resolve_initial_references("NameService");

    // Note the use of the helper class for narrowing
    // (the NamingContext has a helper just like the one
    // automatically created in our example).
    NamingContext ncRef = NamingContextHelper.narrow(nsRef);

    // bind the Object Reference in Naming
    NameComponent nam = new NameComponent("Hello", "");
    NameComponent path[] = {nam};
    ncRef.rebind(path, hello);

    // Wait indefinitely for invocations from clients
    java.lang.Object tmp = new java.lang.Object();
    synchronized (tmp) {
        tmp.wait();
    }
} catch (Exception e) {
    System.err.println("Exception Caught" + e);
    e.printStackTrace();
}
}
```

在这个例子中需要实现的另一个比较关键的代码部分是一个守护进程，用来等待、创建对象的副本，注册这个对象并且等待请求到达。一般来说，运行这个服务器可能需要设置一些运行时属性，用来指定 ORB 服务器。但是，当从一个安装有 WebSphere Application Server 的机器上运行并且这个 WebSphere 正在运行时，就不需要进行任何特殊的设置。这个服务器的代码在 `HelloServer.java` 文件中，如下所示：

```
package HelloWorld;
import org.omg.CORBA.*;
import javax.naming.*;
import javax.rmi.*;
```



```
import java.util.*;

public class HelloORMITest
{
    public static void main(String args[])
    {
        InitialContext initContext = null;

        try{
            Properties props = new Properties();
            props.put(javax.naming.Context.PROVIDER_URL,
                "iiop://localhost");
            props.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.webshpere.naming.WsnInitialContextFactory"
            );
            initContext = new InitialContext(props);

            // Retrieve the object reference using the nameservice
            java.lang.Object ref = initContext.lookup("Hello");
            Hello helloRef = (Hello) PortableRemoteObject.narrow(ref,
                Hello.class);
            // Invoke the Hello object and print out the result
            System.out.println(helloRef.getGreeting());

        } catch (Exception e) {
            System.out.println("Exception : " + e) ;
            e.printStackTrace();
        }
    }
}
```

最后，下列 `HelloTest.java` 文件显示了用来测试这个对象的客户方小程序。这个程序使用 CORBA 命名服务获取一个对象引用，然后为它调用 `getGreeting` 方法并输出结果。

```
package HelloWorld;
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

public class HelloTest
{
    public static void main(String args[])
    {
        ORB orb = null;
        try{
            // create and initialize the ORB
```

```
        orb = ORB.init(args, null);
        // Retrieve the object reference using the nameservice
        // We Fully qualify Object so as not to confuse
        // with java.lang.Object
        org.omg.CORBA.Object nsRef=
            orb.resolve_initial_references("NameService");
        NamingContext ncRef =
        NamingContextHelper.narrow(nsRef);
        NameComponent nc = new NameComponent("Hello", "");
        NameComponent path[] = { nc };
        Hello helloRef =
        HelloHelper.narrow(ncRef.resolve(path));

        // Invoke the Hello object and print out the result
        System.out.println(helloRef.getGreeting());
    } catch (Exception e) {
        System.out.println("Exception : " + e) ;
        e.printStackTrace();
    }
}
}
```

为了测试这个例子，应该编译所有这 8 个 Java 文件（其中 5 个由自动映射过程创建，加上另外 3 个由手工创建）。然后，使用下列命令运行服务器：

```
java HelloWorld.HelloServer
```

最后，使用下列命令运行测试客户程序：

```
java HelloWorld.HelloTest
```

注意，对于这两个命令都需要有正确的 CLASSPATH 设置，以便能找到这些类。

显然，为每个对象实现一个服务器是不现实的，并且手工地启动与停止它们也是不现实的。

EJB 体系结构能够解决这些问题。EJB 服务器能够保留所有的用户对象并允许客户访问它们。

8.5 动态调用接口

DII 允许客户动态地构建与调用请求。客户在调用时只需要知道与接口相关的信息即可；而并不需要知道任何其他信息（例如编译时间）。

虽然结构以及执行请求的过程不同，但 DII 请求的语义与使用映射的 IDL stub 发送的请求的语义完全相同。与静态请求类似，DII 请求也是由操作名称、对象引用以及参数列表组成。

参数被插入到后来被这个调用所使用的名称-值列表中。由于 DII 总是接受一个参数列表（作为一个对象），因此它并不关心存在多少个参数；这允许 DII 支持任何调用。这些结构在运行时被创建与使用（它们仍然可能在运行时进行类型检查并且与 IR 中的信息进行比较）。支持 DII 的基本结构是 NamedValue 结构（用来将一个名称与一个类型为 any 的值相关联）。这个结构代表一个单独的参数。NVList 被用来保存在操作调用中发送的所有参数。在一个关于 NamedValue 结构的实例中包含有参数名、参数值、参数长度以及模式标志（它类似于 IDL 中的 in, out 与 inout 模式标志，分别被称为 ARG_IN, ARG_OUT 与 ARG_INOUT）。这个值被插入到一个类型为 any 的变量中，并且包含有类型代码与实际的数据。

Request 对象用于请求的实际构建与调用。在 Request 接口中可用的操作如下所示：

```
Interface Request {
    ORBStatus      add_arg(.....);
    ORBStatus      invoke(.....);
    ORBStatus      delete();
    ORBStatus      send(.....);
    ORBStatus      get_response(.....);
};
```

8.6 小 结

CORBA 是为了适应在对象之间创建可操作性这一需求而诞生的。它定义了一个远程调用模型，该模型可以被运行于不同平台上的对象所使用，并且它作为更高级服务（例如事务管理）的基础。当 Java 作为一种编程环境与运行时平台出现之后，它的强大功能之一就是支持可移植性。最初，这种可移植性被认可的主要原因是它可以作为一种编写可移植用户接口代码的方法。后来，人们发现可移植性这一特点对于编写服务器方代码非常重要。Java 组件模型最后也演变（关于这个问题的更多信息请参见第 10 章与第 11 章）成作为服务器方组件的最流行的 EJB。需要从运行于许多平台上的许多类型的客户应用程序访问这些服务器方组件。这样就引入了在对象组件模型中的互操作性问题——由此，就需要在 EJB 中使用 CORBA（与 IIOP）。

第 9 章 Internet ORB 间协议

第 8 章已经介绍了作为分布式对象环境下远程调用范型以及作为高级服务的 CORBA。本章将继续讨论 IIOP (Internet Inter-ORB Protocol, Internet ORB 间协议), 它是进行 CORBA 交互的基于 IP 的协议。

9.1 ORB 间能力

虽然 CORBA 1.0 与 1.1 在为分布式面向对象系统定义标准体系结构方面非常成功, 但它们并没有达到 CORBA 创建者最初所定义的与可互操作性相关的所有目标。用户也许并不愿意选择 ORB 作为其任务关键系统的核心中间件。虽然部分原因是某些产品还不是很成熟, 另一些原因是非技术性问题 (例如他们对这些产品缺乏了解), 但更多的原因可能是 CORBA 规范的最初版本存在一些原则性的缺陷, 乃至影响了 ORB 被广泛接受。

在 CORBA 1.x 创建者力图开发一个灵活、开放规范的努力中, 他们并没有解决“ORB 间能力”这一问题——“ORB 间能力”就是不同 ORB 之间的可互操作性能力。CORBA 作为一种体系结构的主要目标是定义一系列机制, 利用这些机制, 可以构建应用程序与服务对象并且使得任何对象能够被任何其他对象所使用。ORB 在对象之间传递请求, 它是 CORBA 这种体系结构的主要支撑者。但是, 在 CORBA 1.x 中并没有解决 ORB 如何与其他 ORB 进行交互的问题。因此, CORBA 1.x 作为一种开放的体系结构, 目前只能用于一个单独的 ORB 中。从一个单独的厂商得到的使用 ORB 的系统的确能够完成 CORBA 最初所定义的目标; 但不幸的是, 如果希望在系统中使用多个 ORB 产品, 可互操作性就难以实现。

这一事实是许多公司在考虑接受 CORBA 作为企业解决方案过程中存在的主要障碍 (由于 CORBA 主要是面向公司市场, 这样也就成为 CORBA 被广泛接受的障碍)。CORBA 1.x 不仅被认为是不完全的, 而且由于 ORB 不能互操作以及基于 ORB 的体系结构只能使用单独一个厂商的产品来构建, 使得许多公司感觉到不能获得独立于厂商且开放的解决方案 (读者也许记得, 在第 8 章的讨论中, 这也是 CORBA 的目标之一)。此外, 由于不同 ORB 产品强调对不同编程语言与编程环境的支持 (例如 Iona 的 Orbix 最初支持 C++, 而 Hewlett Packard 的 DST 支持 Smalltalk——现在属于 ParcPlace), 很自然, 在大多数环境中使用各种 ORB 的混合才是最佳的。

CORBA 与 Web

在 CORBA 2.0 发布以后,由于它集成了 ORB 间能力,使得 ORB 厂商立刻支持这一标准,并且 CORBA 2.0 产品开始以惊人的速度出现。CORBA 2.0 也是公司(用户)开始广泛接受 CORBA 的转折点。在过去的 20 年中,CORBA 被作为分布式面向对象系统的主要支撑者。之所以出现这个局面,很难说是因为 CORBA 2.0 的作用,还是因为 CORBA 像任何技术一样走向成熟的结果。此外,Internet 与 WWW 的迅速兴起,也对 CORBA 作为应用程序级通信的基础起到了非常重要的作用。

Web 上提供的第一代应用程序实际上仅仅是基于表单的信息 accessor,并没有多少应用程序功能。但是,Web 作为信息传递平台,考虑到 Web 上存在的大量观众与客户群,很快就表现出巨大的潜力。这就促使 IT 组织考虑使用 Web 向大量的客户群传递应用程序。公司内联网作为应用程序传递机制的出现,加速了这种趋势,并且大大促进了 CORBA 的应用。

CORBA,特别是 CORBA 2.0 的出现,能够作为填补这一应用领域的解决方案。CORBA 作为一种体系结构,能够成为解决这种互操作性问题的手段。幸运的是,在关于 CORBA 2.0 应用问题的这场竞争中,TCP/IP——Internet 支持者取得了胜利(当然,如果要求将 CORBA 的可互操作性基于 Open Software Foundation 的 DCE (Distributed Computing Environment,分布式计算环境),CORBA 就不可能取得今天这样的地位。实际上,如果那样的话,CORBA 可能会是一个更好但没有人使用的技术)。这就奠定了 CORBA 作为基于 Web 的系统的可互操作体系结构标准这种地位,并且使得 CORBA 能够随着 Internet 的发展而发展。

CORBA 2.0 解决了 CORBA 1.x 所忽略的许多问题。在所有这些问题中,可互操作性不仅是最重要的方面,而且是最出成果的方面。CORBA 2.0 规范不仅定义了 GIOP (General Inter-ORB Protocol,通用 ORB 间协议)与 IIOP,而且解决了基于 DCE 的可互操作性问题以及它们与 Microsoft COM (Component Object Model,组件对象模型)和 OLE Automation 的互操作问题。作为一个成熟的规范,CORBA 2.0 也提出了作为设计级抽象的可互操作性体系结构。CORBA 2.0 深入讨论了体系结构应该提供什么样的可互操作性、桥应该如何工作以及在不同情况下应该使用什么样的解决方案等问题,进一步使得 CORBA 2.0 成为分布式系统(无论是局部分布式系统还是广域网上的分布式系统)的核心底层技术。毫无疑问,CORBA 2.0 规范是分布计算领域的基础文档。

CORBA 2.0 解决了运行于不同 ORB 环境的对象之间的可互操作性问题(CORBA 1.x 仅仅解决了运行于 ORB 环境内部的对象之间的可互操作性问题)。如图 9-1 所示,ORB 可互操作性就是一个在 ORB A 上的客户调用 ORB B 上的对象的 OMG IDL (Interface Definition Language,接口定义语言)定义的操作的能力,其中,ORB A 与 ORB B 都是独立开发的。

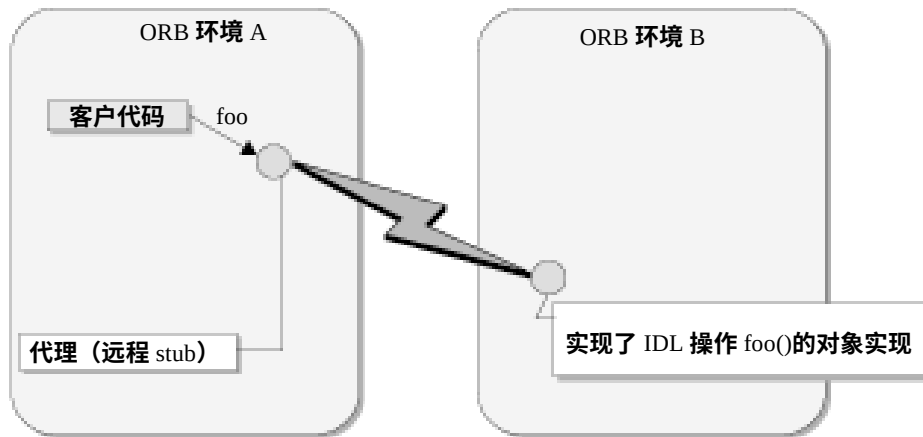


图 9-1 ORB 可互操作性

从 CORBA 2.0 所需的一系列解决方案中可以看出,ORB 可互操作性是一件非常困难的任务。ORB 可互操作性之所以存在这种复杂性,主要是由于以下原因:

- CORBA 是一种在行为级与体系结构级定义的规范。它并没有定义实现级属性。因此,ORB 可能在内部工作上非常不同。由于 ORB 可互操作性的定义并没有对参与的 ORB 做任何假设,因此确保可互操作性的规范必须考虑关于此规范的任何可能实现。
- CORBA 自身的组件(ORB 核、对象服务、对象软件实用程序)就是一个支持大量功能的复杂环境。因此,构成对象与请求的语法上下文的属性集合可能很大并且复杂。当一个请求或对象引用传递到一个 ORB 中时,必须正确管理与维护这些语义。此外,CORBA 作为一种体系结构是不断演化的。可互操作性的规范必须确保能够在进行很少的工作乃至不进行任何工作的条件下就支持未来的服务或软件实用程序——当然,它们必须确保未来添加的内容不损害体系结构。
- 可互操作性是且必须是双向的。一个 ORB 环境能够访问另一个 ORB 环境是不够的。实际上,一个调用过程常常包括作为参数被传递的对象引用,然后这个对象引用被对象实现用来调用所引用对象上的另一个操作(从而颠倒了客户与服务器的角色)。在一个环境中的某个对象能够对第二个环境中的某个对象进行访问的调用,也隐含地表示传递给第一个环境的每个引用对第二个环境可用。

由于 ORB 之间的可互操作性必须处理许多情况,因此 CORBA 2.0 规范定义了两种方法,用来在两个环境之间实现桥。一种方法称为间接桥(mediated bridging),如图 9-2 所示,这种方法强调每个内部 ORB 环境可以维持自己的内部表示,并且要求在外部的使用一种可互操作的格式,用来在两个 ORB 环境之间进行通信。那些需要跨越 ORB 边界的元素被转换成这种可互操作的格式,然后再转换成其他 ORB 的内部表示。IIOP(对于那些内部不是使用

IIOP 的 ORB) 就是一个用来实现间接桥的协议例子。

第二种方法称为直接桥 (immediate bridging), 如图 9-3 所示, 在这种方法中, 实体在两个内部格式之间直接转换。显然, 直接桥更加高效但灵活性较差, 因为必须为每个不同的环境提供内部格式转换例程。

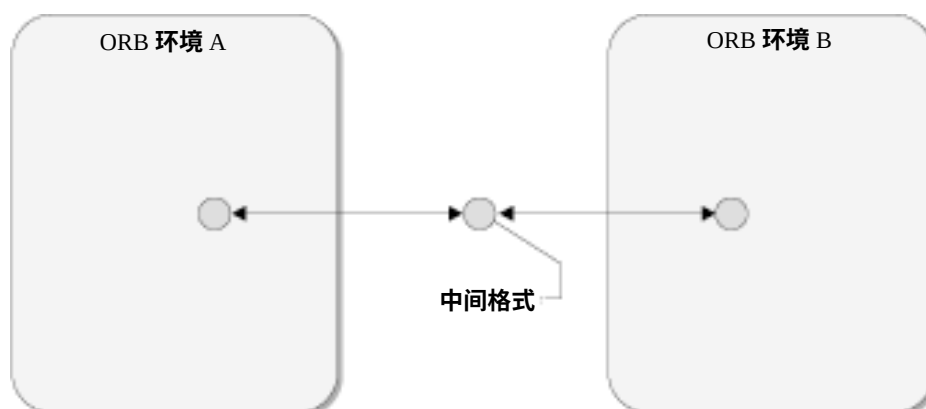


图 9-2 间接桥



图 9-3 直接桥

9.2 域

域 (domain) 是一种集合有许多具有类似特点的组件的组织单位。这是一个比较广泛的概念, 它可以用于物理划分, 也可以用于功能相关的划分。在考虑到可互操作性问题时, 域的特点是非常重要的, 因为域之间的桥需要存在从一种特性集合到另一种特性集合的映射。

并非总是可能在不损失任何信息的条件下定义两个域之间的映射，因为有些域在语义上比另一些域的语义丰富。

读者也许对物理域或管理域的概念比较熟悉。本章关注的是作为独立 ORB 环境定义的域之间的可互操作性。这里所提到的域桥这一概念被限制为 2 个不同 ORB 环境之间的桥。

在定义域桥时，CORBA 2.0 规范对两种桥级别进行了区分。第一种称为请求级桥（request-level bridging），它使用公共的 CORBA API 来执行所需的转换。ORB 与基于 ORB 的组件用来执行这种转换。另一方面，内嵌桥（In-line bridging）在 ORB 内部封装了整个转换过程。

9.2.1 动态骨架接口

在 CORBA 2.0 中用于实现请求级桥的主要组件是 DSI（Dynamic Skeleton Interface，动态骨架接口）。DSI 与 IDL 产生的实现骨架共同存在，如图 9-4 所示。与 IDL 骨架不同，DSI 可以作为任何对象实现的骨架，而不必知道其实际接口是什么。

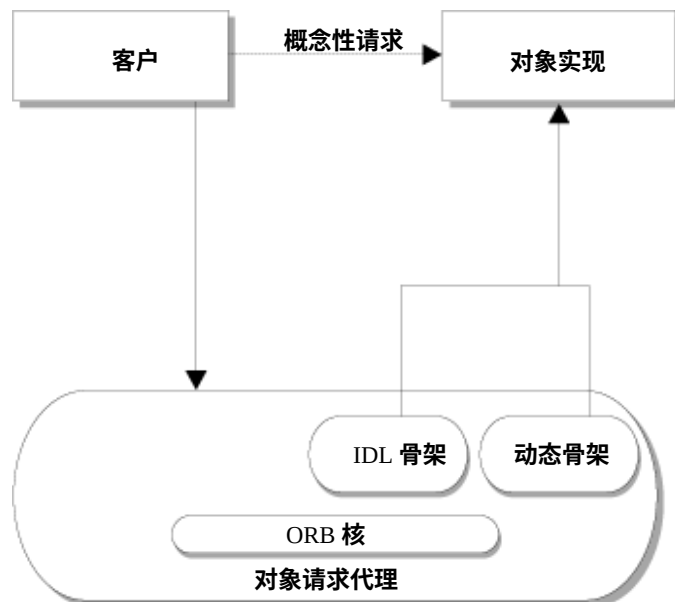


图 9-4 动态骨架接口

DSI 在某种程度上类似于 DII（Dynamic Invocation Interface，动态调用接口）。DII 允许客户指定目标对象引用、操作名称以及参数来创建请求。它提供了一种机制，使得请求在创建完成之后可以发出。DSI 提供了关于此过程的实现方对立面。无论哪个请求到达，将使用相同的方法调用 DSI。这种方法就是 DIR（Dynamic Implementation Routine，动态实现例程），并且它能够接收 DII 中创建的代表任何操作请求的相同结构（也就是，一种指定目标对象、

操作以及参数的结构)。DSI 被定义为伪对象 (pseudo-object)，伪对象的接口是使用 IDL 接口定义的，但伪对象未必被实现为 CORBA 对象，例如，伪对象可以被嵌入作为 ORB 的一部分。代表服务器请求的伪 IDL (PIDL)，如以下 IDL 所示：

```
module CORBA {  
    pseudo interface ServerRequest {  
        Identifier op_name();  
        Context    ctx();  
        void        params(inout NVList params);  
        any         result();  
    };  
};
```

正如对象实现在为请求提供服务时并不知道该请求如何产生一样（也就是，是使用 DII 还是使用 IDL stub 产生该请求），发出请求（以及接收结果）的客户并不知道该请求是通过 IDL 骨架被服务还是通过 DSI 被服务。

这种不透明性就允许 DSI 被用于请求级桥。在这种应用背景下，使用 DSI（它在不需要任何编译时信息的条件下就可以处理任何请求）将请求（当对象引用被另一个 ORB 管理时发出这个请求）传递到桥。在桥的另一方，DII 将被用于向第二个 ORB 环境中的真实对象传递请求。这一过程在图 9-5 中显示。这两个处理请求的抽象与通用模型的组件的组合，使得这种请求级桥的创建相对简单。

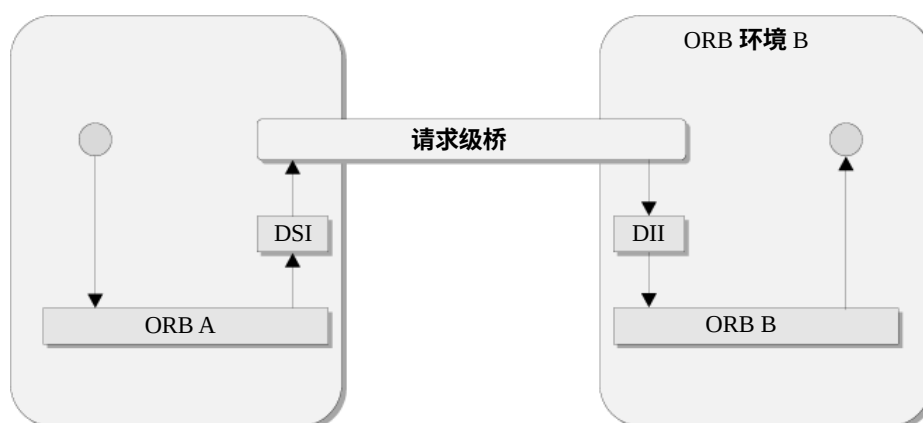


图 9-5 请求级桥

9.2.2 域间引用

如何表示对象引用这一问题可能是可互操作性问题中惟一最重要的。只要客户与对象实现位于相同的 ORB 环境中，对象引用都是合法的；但是，如果它们位于不同的域并且使用不

同的表示时，情况会怎么样呢？在这种情况下，就需要找到一种不依赖于真实对象所在环境的关于对象引用的表示方法。

当目标对象实现存在于另一个 ORB 环境中时，引用表示（reference representation）就是非常关键的一个概念。考虑图 9-6 所表示的情况。客户需要调用另一个 ORB 环境中对象的一个操作。这个操作调用将使用另一个对象作为参数；实际使用的对象则在客户的环境中。当对象实现服务这个请求时，它可能需要使用作为参数传递过来的对象引用——也就是，发送一条消息。这就意味着，域间调用（这个例子中的第一个调用）必须管理对象引用的转换，从而使得对象实现能够透明地使用参数以便进行更多的调用。在一个更复杂的例子中，作为操作参数的对象引用可能位于一个不同于客户环境以及对象实现环境的第三个 ORB 环境中。

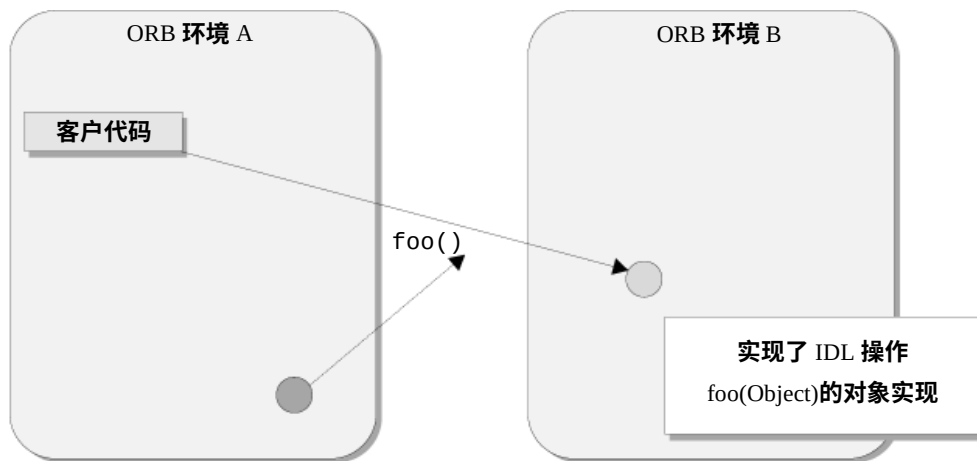


图 9-6 域间引用

随着 CORBA 的发展，显然有时需要通过传引用的方式传递对象，有时又需要通过传值的方式传递对象。因此，对这两种模型的支持都是重要的，WebSphere 使用下列标志支持这两种模型：

```
-Dcom.ibm.CORBA.iiop.noLocalCopies=true
```

这个标志允许通过传引用的方式而不是传值的方式传递对象。

CORBA 2.0 规范定义了下列信息，这些信息对于对象引用的表示与转换是必要的：

- 被引用的对象是否为 null 对象？
- 这个对象的类型是什么？
- 这个环境支持什么协议？
- 有什么 ORB 服务可用？

这些数据对于正确表示对象引用是必要的，它们构成了实现 ORB 可互操作性的关键的 IOR (Interoperable Object Reference, 可互操作对象引用) 结构。

9.3 可互操作对象引用

IOR 是一种对于实现桥 (以支持 ORB 可互操作性) 非常有用的数据结构表示。为了方便，这个数据结构是使用 IDL 定义的，但它不应该被除了桥之外的其他组件所使用。

IOR 包括一个类型指示器以及一个带标记的 profile 指示器集合 (每个所支持的协议对应一项)。虽然许多情况将有一个单独的协议 (例如使用 IIOP 作为内部协议的 ORB 环境)，这个规范支持带有多种协议的多种环境。一个带标记的 profile 包含有一个标识符以及八位字节组流，并且包括标识在那个特定的协议中被引用的对象所需的所有信息 (即它包括这个协议下对象的“不透明”表示)。IOR 包含有多种协议标识信息这一事实，允许即使当底层的协议发生改变时相同的 IOR 仍可以被传递 (只要 IOR 底层的 profile 有相应的八位字节组流的定义)。IOR 使用一个特殊的类型 ID 以及一个空的 profile 集合来表示 null 对象。

使用在 ORB 接口中定义的 object_to_string 操作，可以将 IOR “字符串化” (也就是，能够创建一个表示 IOR 的字符串)。后来，利用 string_to_object 调用，就可以根据这个字符串表示创建一个可直接使用的 IOR。这就意味着，代表 IOR 的字符串在编码过程中包含有 IOR 的所有数据。因此，IOR 的字符串化表示都比较长，如下所示：

```
IOR:00000000000000002849444c3a4a4f4d6573736167696e674c617965722f4a4f4d6573
7361676551756575653a312e3000000000020000000100000064000000000000040000
000a000000107d48e77d8ceb000002500312cd0000000000000b000000107d48e77d8ce6
000002500312cd000000081040000000000c4453543a352e363a0113057b081040030000
001038302e332e31382e3230353b3334363000000000000002c00010000000000c3830
2e332e31382e323035000d84000000000107d48e77d8ceb000002500312cd000000
```

IOR 的字符串化表示对于允许系统有一个公共的开始点是非常重要的。要确定从其中开始访问其他对象引用的第一个对象引用总是很困难的。虽然命名服务很流行，但简单地传递一个字符串就能够创建公共开始点这一事实还是被人们广泛使用。例如，一个可以被嵌入在 applet 参数中的字符串就是启动基于 ORB 事务过程的简便方法。

9.4 ORB 服务

ORB Service (ORB 服务) 是在 CORBA 2.0 可互操作性规范中定义的新概念，读者不应该将它与 CORBA Service 或对象服务混为一谈 (虽然它们有相关性)。ORB 服务是支持特殊的可互操作性转换的 ORB 级模块，用来支持 CORBA Service。

CORBA Service 提供了各种应用程序级功能，例如对事务、安全性以及关系的支持。服务可

以被划分成两组（与 ORB 服务的讨论相对应）：不需要 ORB 级支持并且完全通过 IDL 接口集合定义的服务、需要 ORB 级支持的服务。为了支持第二个服务类型，需要定义 ORB 服务。

例如，事务服务与安全性服务都需要 ORB 级支持。事务代码必须被原子性地（atomically）执行（也就是说，要么所有的代码都执行，要么都不执行）。由于事务可能包括一个远程调用，因此，事务代码可能跨越多台机器上的多个进程。因此，必须将事务上下文信息与远程调用一同传递。但是，对事务上下文的管理必须对应用程序编程者不可见，因此 ORB 服务必须支持事务上下文管理任务。

现在假设运行应用程序的环境不同于对象实现所在的环境。在这种情况下，调用将从一个 ORB 环境中产生，并且必须被转换到第二个 ORB 环境。由于这个调用必须包括适当的事务上下文信息，因此这部分请求也必须被转换到新环境中；被创建用来对 Transaction Service（事务服务）提供可互操作性支持的 ORB 服务就是用来执行这种转换的组件。

Security Service 是另一个关于 ORB 服务必要性的好例子。安全性信息（例如授权信息、签名等）在一个安全环境下通过请求被传递。当一个调用跨越域边界时，必须对这些信息进行转换——一个由适当 ORB 服务所处理的转换。

由于 CORBA 规范并不直接处理 CORBA Service，因此 ORB 服务被定义为体系结构组件。这个规范并不指定任何特定的服务以及如何支持这些服务；相反，它为处理这些服务提供了一个适当的框架，目的是使得这些支持在后来能够扩展到任何其他服务上。它试图确定可能需要这些支持的各种领域，并且确定需要从 ORB 层提供的挂钩（hook）。它也定义了当不同的 ORB 各自支持（可能）不同的 ORB 服务集合时，这些 ORB 之间应该如何协商。

为了支持各种服务可能需要的与服务相关的任何信息的转换，这个可互操作性规范还定义了一个 IDL 类型，这个 IDL 类型在一个结构序列中封装了与服务相关的信息，每个结构都包括一个服务标识符以及一个保存有与服务相关的上下文信息的八位字节组流：

```
module IOP {
    // Each CORBA Service will define its own constant

    typedef unsigned long ServiceID;

    struct ServiceContext {
        ServiceIDcontext    _id;
        sequenceoctet       context_data;
    };

    typedef sequence<ServiceContext> ServiceContextList;
};
```

这些附加的上下文结构是通过正在通信的 ORB 作为请求的一部分发送的，因此允许进行服

务上下文的传输。

9.5 通用 ORB 间协议

GIOP (General Inter-ORB Protocol, 通用 ORB 间协议) 是一个可以被映射到任何面向连接的传输层的 ORB 可互操作性协议。GIOP 通过一个连接与请求进行协商, 并且假设传输层所提供的连接是可靠的, 它负责维护消息的排序。GIOP 还假设传输层提供某种形式的断开连接通知以及传输确认机制。GIOP 是一个普遍的协议, 并且上述这些假设目前还有限制基于连接的传输协议, 人们使用 TCP/IP 构建了 GIOP 作为参考的传输级模型; 因此, 将它们映射到 TCP/IP、代码命名的 IIOP 是一件很自然的事情。

GIOP 规范包括 CDR (Common Data Representation, 公共数据表示) 以及 GIOP 消息规范。CDR 将 IDL 类型映射到形成 GIOP 传输语法的连线级 (wire-level) 表示。GIOP 也定义了与会话参与者之间可以交换哪些消息以及这些消息是如何形成的。

公共数据表示

CDR 定义了 IDL 类型如何被打包到字节流中。CDR 的设计目标是简单、高效与灵活。CDR 是一个连线级传输语法, 它定义了在线上实际传输的内容以及 IDL 如何被打包到这些字节流中。

CDR 不仅映射 IDL 类型; 它也映射为了保证一条请求被正确执行所需传递的全部内容。例如, CDR 也定义了伪对象 (例如 typecode) 以及还没有在 CORBA 核规范中完全指定的数据类型的映射。

CDR 将每种数据都放置在一个八位字节组流中 (也就是通常所说的字节流)。它指定了这些流如何被创建以及如何对它们进行解包。在此过程中, 它对两种类型的流进行了区分: 用来对 GIOP 消息进行打包的流以及用来对封装 (encapsulations) 进行打包的流。封装就是一种流, 它将任何 IDL 数据结构打包到一个独立的字节流 (具有 IDL 类型 `sequence<octet>`)。后来, 这个封装就能够用作插入到其他流中的一个模块, 这些其他流可以是消息流或其他封装。封装对于创建层次式包含结构的流以及支持将数据 marshaling 与 unmarshaling 到流上的“分而治之”策略是非常有用的。CDR 提供的字节排序方法是 receiver-makes-right (由接收者确定)。CDR 并不强制与维护某种字节排序 (big-endian 或 little-endian)。相反, CDR 对数据类型的打包支持这两种方式。当对流的打包结束之后, CDR 流进行编码以说明使用的是哪种字节排序方式。接收者负责直接按照所打包的形式使用这些字节, 或是颠倒它们的顺序 (如果包的字节排序方式与接收者自己的字节排序不匹配)。这个方法更为优越, 因为如果两端使用相同的字节排序方式, 并不需要进行任何转换。

注意: 封装流也有一个排序标记, 并且完全可能出现这种情况: 外围流与嵌入在

其中的封装流具有不同的字节排序方式。

除了 IDL 数据类型与 typecode 之外，CDR 还定义了如何对下列信息进行编码：

- Context（上下文）：一个关于名称-值对的列表，其中包含有产生请求的上下文的信息。上下文传递给对象实现的方式是，将它作为一个字符串序列写入到 CDR 流中，显然，必须具有偶数个字符串。
- Principal（规则）：标识一个调用者，在与安全相关的机制中使用。CDR 仅仅定义了规则必须作为字节流被打包，它没有对这个字节流指定任何语义。
- Exception（异常）：作为一个标识此异常的字符串被打包，后面跟随异常结构的成员。
- Object reference（对象引用）：使用 IOR 结构进行打包。

9.6 GIOP 消息

GIOP 使用一个简单的消息集合，支持 CORBA 所需的所有功能。其中，核集合仅仅包括 7 个消息；这些消息支持所有的调用模型、异常产生与传播、上下文传递及引用管理。

当一个 Request 类型的消息从客户发送到服务器时，就启动了一个典型的 GIOP 会话（这时，客户就是这样的 ORB：保存有对象引用的应用程序客户通过这个 ORB 调用操作；服务器就是这样的 ORB：通过这个 ORB 将激活对象实现）。如果所有的处理工作都顺利，服务器将以 Reply 消息进行响应。为了关闭会话，服务器将发送 CloseConnection。任何时候，客户与服务器都可能发出 MessageError 消息。

客户使用 Request 消息对一个操作调用进行编码。这条请求消息中包含有一个头部与一个消息体。在头部中包含有诸如服务、说明请求是否需要有一个响应被送回客户的标记、为其发出请求消息的这个操作的名称。请求消息体中封装了 in 与 inout 参数（这些参数从客户传递到对象实现）以及一个关于上下文对象的可选编码。

应答消息作为对请求的响应而发出（例如，在有些情况下，单向操作并不需要响应）。应答消息的头部中包含有一个上下文列表、请求 ID、应答状态。应答状态可以是下列状态之一：

- NO_EXCEPTION：表示请求处理过程被正常地完成。在这种情况下，应答消息体中给出了这个返回值以及 out 参数与 inout 参数的值。
- USER_EXCEPTION 或 SYSTEM_EXCEPTION：表示在处理请求的过程中出现了异常。在这种情况下，消息体中将给出所出现的异常。
- LOCATION_FORWARD：表示服务器正在指示客户应该将这个操作重新发送到一个不同的对象引用（例如，由于各种可能的原因，这个对象可能被放置在另一个地方）。这时，消息体中将包含 IOR，这个 IOR 用来再次发送此消息。

CancelRequest 消息将由客户发送给服务器，并且这个消息的头部中包括应该被取消的消息

ID。LocateRequest 与 LocateReply 对于查询一个特定的对象引用非常有用。客户可以发送 LocateRequest 消息，用来判断打包在这个消息中的对象是否合法、这个消息是否被服务器管理等等。服务器将使用 LocateReply 消息进行应答，在这个消息中的位置状态是 UNKNOWN_OBJECT, OBJECT_HERE 或 OBJECT_FORWARD。如果状态为 LOCATE_FORWARD，那么在消息体中将包含要被用来访问对象的新 IOR。

在请求会话过程中，服务器与客户都可能发送 MessageError 消息，用来表示版本号的不一致性或不可认识的消息结构。最后一种消息类型是 CloseConnection，由服务器发送这种消息，用来表示它将不再对请求作出响应。

9.7 IIOP

IIOP (Internet Inter-ORB Protocol, Internet ORB 间协议) 是映射到 TCP/IP 上的 GIOP 的特殊版本。由于 TCP/IP 是如此普遍存在，以至于 IIOP 这一名称变成了 CORBA 2.0 与 GIOP 的代码名称。TCP/IP 以及 Internet (绝大部分由运行 TCP/IP 的网络组成) 使得 IIOP 成为事实上的 ORB 协议。

IIOP 在 ORB 环境中有两个作用。IIOP 被设计作为一种可互操作性协议，表现在当需要跨越 ORB 边界时，它能够被用作将要转换到的目标 ORB 环境的公共表示。由于 IIOP 是 OMG (Object Management Group, 对象管理组) 所拥有的已发布格式，因此，它是许多 ORB 厂商能够转换到的最佳候选。

虽然 IIOP 的目标是作为最常用传输协议 (TCP/IP) 上的公共可互操作性协议，但它的实际应用已经远远超出于此。IIOP 是一种可互操作性协议；但它也是一个能够表示 ORB 环境中所需所有动作的 ORB 协议。因此，IIOP 不仅被用作一种可互操作性协议，它也被许多当今的商业 ORB 用作 ORB 协议。IIOP 不仅用于 ORB 边界内部，它也被用于整个 ORB 域的内部。

由于 IIOP 仅仅是 GIOP 在 TCP/IP 上的实现，因此它仅仅需要一个关于 IIOP profile 的规范，用来在 IOR 中作为“IIOP 代表”。由于 TCP/IP 支持这样的抽象，使用一个主机标识符与一个端口号就能够形成通信信道，因此在 profile 体中包含有这些标识符以及用来表示一种非透明数据结构的附加字节流，执行管理任务的 ORB 使用这种非透明数据结构来表示对象。这个结构由 ORB 软件放置在请求消息内；由于作为接收者的 ORB 是这种非透明结构的所有者与创建者，因此它能够对这种结构进行解释并且为这个请求服务。

9.8 RMI Over IIOP

RMI 是 Java 的远程方法调用机制，用来在 Java 中实现 RPC (Remote Procedure Call, 远程

过程调用)。这在本质上解决了分布式编程环境下的通信问题，但同时又存在其他问题，例如可靠性与安全性（这些问题正是 CORBA 需要解决的）。

RMI 与 CORBA 非常相似，表现在两者都使用 stub 代码与骨架代码。必须从 `java.rmi.Remote` 继承 RMI 接口（并且这里的每个方法都必须发出 `java.rmi.RemoteException` 异常，从而使得客户代码能够应付任何类型的通信错误）。需要从 `java.rmi.UnicastRemoteObject` 继承对象实现。一旦有了这两个接口，并且将实现编译为 JAVA 类文件，那么就可以运行 RMI stub 编译器 `rmic`。这个编译器为远程对象产生客户 stub 与服务器骨架。

使用称为 JRMP（Java Remote Method Protocol，Java 远程方法协议）的协议运行 RMI。为了使用 RMI，必须利用 RMI 命名服务来定位对象，并且在 RMI 注册表中注册 RMI 对象。但是，当使用 RMI over IIOP 时，关于命名服务的所有工作都由 CORBA Service 来完成。

下列 `HelloRMITest.java` 代码显示了一个关于客户方测试程序的例子，这个程序使用 RMI-IIOP，并且基于上一章介绍的例子实现。在这里，JNDI（Java Naming and Directory Interface，Java 命名和目录接口）用来执行搜索。为了发生搜索动作，需使用 `Properties` 对象来指定底层的命名服务实现。除了使用 JNDI 之外，这个例子与前一个测试程序的惟一不同之处在于，这里使用的是 `javax.rmi.RemotePortableObject.narrow`，而不是 CORBA 中用来进行类型转换的收缩操作。这个测试程序与基于 CORBA 的那个测试程序之间关键不同在于，这个测试程序根本没有访问 ORB。这也“证明了”在客户方使用的是 RMI，而不是 CORBA。

```
package HelloWorld;
import org.omg.CORBA.*;
import javax.naming.*;
import javax.rmi.*;
import java.util.*;

public class HelloRMITest
{
    public static void main(String args[])
    {
        InitialContext initContext = null;

        try{
            Properties props = new Properties();
            props.put(javax.naming.Context.PROVIDER_URL, "iiop://localhost");
            props.put(
                javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
            initContext = new InitialContext(props);

            // Retrieve the object reference using the nameservice
```



```
        java.lang.Object ref = initContext.lookup("Hello");
        Hello helloRef = (Hello)
            PortableRemoteObject.narrow(ref, Hello.class);
        // Invoke the Hello object and print out the result
        System.out.println(helloRef.getGreeting());

    } catch (Exception e) {
        System.out.println("Exception : " + e) ;
        e.printStackTrace();
    }
}
```

需要说明的是，RMI-IIOP 可以用于两种不同的情景。第一种情景类似于这里所使用的情況，RMI 被用来访问基于 CORBA 的服务与对象。这与 WebSphere 开发过程相符合，并且与 EJB 体系结构一致。另一种情景涉及 CORBA 客户对 RMI 服务器对象的访问，这样就允许从遗留代码访问 Java 代码。随着 EJB 的引入，这种应用情景变得比较少见。EJB 体系结构允许在服务器上部署基于 Java 的组件并且使用 CORBA 来访问它们。

9.9 小 结

本章描述了 IIOP，它是 CORBA 在 Web 上工作的一般方式，特别是在 WebSphere 中工作的方式。现在读者已经对 CORBA 与 IIOP 有比较好的了解，因此在后面可以开始讨论 JavaBean 与 Enterprise JavaBean。JavaBean 是一种常规的组件模型，而 EJB 是为分布式编程提供的组件模型。EJB 编程模型在 RMI-IIOP 之上实现。在了解关于 EJB 的功能时，应该时刻记住这一原理以及 CORBA 与 IIOP 中引入的范型；这样有利于理解 EJB 的工作原理。

第 10 章 JavaBean

本章描述构成大多数 Java 程序基础的 JavaBean 组件；Java 编程中使用的许多模式都基于在这个模型中所定义的概念。在本章的学习中，读者将知道 JavaBean 组件的构成以及它是如何通过属性、方法与事件来描述的。本章还将介绍如何使用 BeanBox 工具（可从 JavaSoft 免费获取）来创建 bean。这个工具在这里主要用来帮助理解在 IDE（Integrated Development Environment，集成开发环境）中如何使用这些 bean，以及 bean 的元数据是如何被读取与编辑的。本章也将涉及一些更高级的概念，例如事件适配器与 bean 的持续性。

10.1 JavaBean 组件模型

JavaBean 是一种定义 Java 组件结构的组件模型。软件组件模型一般都（JavaBean 尤为如此）面向定义简单而又功能强大的参数这一目标，这些参数允许创建易于重用的代码以及对它们进行打包。组件模型的目标是指定一种软件组件创建者与用户都熟知的严格结构。只要在开发过程中遵循这种结构，开发人员就能够确保所设计的组件能够被很好地理解并且与常见的开发环境、框架等兼容。Java 编程语言由于其接口与类实现相分离，因此这种语言特别适合于开发组件，

任何组件模型的目标都应该允许发展基于组件的产业。为了促进这种情况，仅仅创建一个组件模型是不够的；实际上，也需要提供诸如 IDE、打包工具、第三方组件之类的“关键物质”。虽然 JavaBean 并非第一种组件模型（在比较值得一提的组件模型中，有 Microsoft 的非常成功的 COM 和 Apple/IBM 的 OpenDoc），但在得到广泛应用的速度方面，JavaBean 是创纪录的。目前，JavaBean 是 J2SE（Java 2 Standard Edition，Java 2 标准版）的一部分，并且它几乎是当今市场上任何 Java 库的基础（java.beans 包被集成在 Java 中，作为 JDK（Java Developer's Kit）1.1）。此外，由于 JavaBean 模型是一种“可视化”编程，因此 Java 的标准 GUI 组件（例如 Java Foundation Classes(JFC)/Swing 组件）实际上就是 JavaBean。大多数 IDE（例如 IBM VisualAge for Java 或 Symantec VisualCafe）为创建与修改 bean 的实例提供了可视化工具。JavaBean 也是 EJB 规范的基础，本书将在第 11 章讨论 EJB。

JavaBean 组件模型通过对属性、事件与方法的描述，指定了 Java 组件的结构。除了这个基本结构之外，本章还将讨论持续性、introspection（内省）（类似于 Java 反射）、定制以及打包。那些已经熟悉 JavaBean 的读者可以跳过本章。但是，如果对 JavaBean 还不熟悉，我们建议花一些时间来阅读本章，因为 Java 程序常常使用 JavaBean 核心规范中定义的原理与模式。

10.2 JavaBean 的定义

JavaBean (或 bean) 是一种具有特定结构的可重用 Java 组件, 这种特定的结构构成了由不同开发者开发的组件之间的共同基础——这些组件完成不同的任务并且使用不同的资源。这样, 它就能在开发时与运行时被工具与程序 (在 bean 出现之前就已经写好的程序) 所处理, 并且不需要任何关于 bean 的特殊知识。这些工具并不需要知道任何关于 bean 的特殊信息; 相反, 它们能处理所有 bean, 因为所有 bean 都具有相同的基本结构。这些工具与程序发生作用的抽象级是组件抽象级。通过提供一个公共的继承根, 面向对象编程中的软件层次允许外部用户类似地看待许多不同的实现。JavaBean 通过定义一个通用结构, 在更大程度上做到这一点。JavaBean (以及任何组件模型) 背后的基本原理是: JavaBean 定义了一个抽象级, 用来统一所有软件实体, 并且允许每个这样的实体变成一个组件。通过允许不同来源的独立代码在处理方式上“看上去相同” (无论在开发时还是在运行时都如此), 从而使得 JavaBean 能够实现在 Java 领域内部的软件重用。

10.2.1 属性、方法与事件

为所有 bean 定义共同基础的结构包括属性、方法与事件。其中, 属性 (property) 是一个 bean 的特性。每个特性都有名称和类型, 通过 accessor (get 方法与 set 方法) 对它进行访问。特性通常映射到 bean 所实现的功能, 并且通过 bean 的属性就能够了解关于此 bean 的许多内容。关于属性的例子, 包括诸如可视 bean 的大小、颜色与阴影等特性, 以及诸如 PagerRequest bean 的 pager 类型、pager 数以及重试策略等特定域特性。

读者也许会问, 为什么要将 PagerRequest 对象打包成为一个 bean 呢? 实际上, 将对象打包成 bean 允许系统中的工具利用所有这些对象。例如, 可以编写 (甚至借用) 属性编辑器, 它们能够对任何 bean 进行编辑, 从而允许这个编辑器创建 pager 请求、部分顺序或几乎任何其他域对象。这就意味着, 只要在遵循标准的情况下, 进行很少的工作, 通常就能够创建非常通用的软件。

方法 (method) 是一个对象的主要工作接口。一个 bean 的 public 方法包含有它的 API, 它说明了这个对象知道应该做什么。bean 的开发人员可能会决定仅仅导出 public 方法的部分集合作为 bean 的 API, 在这种情况下, 开发人员就需要为这个 bean 构建一个特殊的“描述符”。

事件 (event) 是 bean 引发的动作。事件是一种比较受欢迎的机制, 通过这种机制, 一个软件组件可以通知另一个软件组件; 它们允许一个组件对由另一个组件所产生的事件作出反应, 从而支持组件交互。JDK 1.1 (以及后续版本) 在很大程度上依赖于组件发出事件并且其他组件通过侦听程序对这些事件进行注册这一概念。JavaBean 要求每个 bean 定义一系列能够通过一系列提交方法发出的事件, 并且通过这些提交方法, 其他组件能够被告知关于

此 bean 的事件。

10.2.2 生命周期问题

bean 的生命过程比“普通”可重用 Java 代码的生命过程稍微复杂一点。这种复杂性主要来自 bean 的可重用性这一事实。在开发时，bean 知道如何在 IDE 中进行动作。因此，它不仅必须提供运行时所使用的 API 集合，也需要提供在开发过程中非常有用的 API 与结构。例如，它必须能够在 IDE 中工作，使得 IDE 能够知道关于这个 bean 的属性、方法与事件。此外，它还必须提供关于自己的足够信息，以便能够嵌入到在 IDE 中可用的工具里（例如，属性编辑器与可视连接管理器）。它必须为开发人员提供定制以及设置其属性的能力，并且它必须提供被定制之后保存自己的机制。为了使一个组件能够成为真正的 bean，所有这些附加的复杂工作都是必要的。一旦解决了这些问题，bean 作为可重用组件的价值就得到提升，因为在程序中使用这些 bean 时所需进行的工作就更少。

为了使得 bean 能够在开发时与运行时都正确工作，bean 开发人员必须注意下列问题。

持续性

与普通 Java 对象不同，bean 必须提供可以被串行化与取消串行化所需的功能。这是必要的，因为 bean 用户通常会设置 bean 的一些属性，并且希望直到运行时才保存这些属性设置。在 JavaBean 模型中，提供给 bean 的机制就是串行化；也就是说，一个 bean 应该提供用来将自己写入到流中以及从流中读出自己的方法。然后，这一方法将被 IDE 以及使用这个 bean 的程序所使用。

introspection

Bean 必须提供关于自己的足够信息，从而使得 IDE 能够帮助开发人员理解这个 bean 能够做什么，并且它必须定义如何与系统中的其他代码段进行交互。JavaBean 定义两种方法，可用来完成上述要求。一种方法依赖于一组约定，第二种方法则要求开发人员构建 helper 类。例如，在基于约定的方法中，如果 PagerRequest bean 有一个类型为 String 名为 TelephoneNumber 的属性，那么 bean 开发人员就应该提供下列形式的两种方法：

```
String getTelephoneNumber()  
void setTelephoneNumber(String aString)
```

它们是一个属性的标准 accessor。一般来说，bean 开发人员只要遵守这些约定（而不必做其他工作）就能够确保 bean 在 IDE 中正常工作。但是，对于复杂的 bean，这还是不够的。例如，这里的命名约定隐含表示所有 public 方法都被看作是组件的 JavaBean 方法。如果 bean 开发人员想要隐藏其中某些方法，但又不能修改除 public 方法之外的其他任何访问类型，这时，开发人员就需要进行稍微复杂的工作。在这种情况下，必须定义并实现一个 BeanInfo 对象。BeanInfo 对象描述了这个 bean 的结构，并且被 IDE 在 introspection 过程中使用；它

是一个元数据对象。

定制

bean 可以定义定制器 (customizer) ——为了能够在 IDE 中更容易地使用 bean 而进行定制的特殊编辑器。定制器常常包括向导与引人注目的用户界面, 用来指引 bean 用户完成定制与使用 bean 的全过程。

打包

由于 bean 是由一些人设计而由另一些人使用的组件, 因此, JavaBean 规范定义了开发人员应该如何对这些 bean 进行打包, 从而保证 bean 用户在自己的开发环境中试图使用这个 bean 时不会遇到不必要的困难。

10.2.3 Juggler 示例

在继续讨论 JavaBean 的更高级特性之前, 首先看一个简单的例子。在这里, 将分析一个简单的“juggler”bean 以及一个简单的按钮 bean, 并且说明它们是 Sun 的 BeanBox (BeanBox 是 BDK (Bean Developer Kit) 的一部分, 可以从 http://java.sun.com/products/javabeans/software/bdk_download.html 获取 BDK) 中的外观。本书在这里将创建一个包括两个按钮的简单应用程序, 这两个按钮用来控制 juggler: 一个用来启动 juggler 过程, 另一个用来停止此过程。我们将在 BeanBox 中通过可视化地挂接事件完成所有的“编程”工作。这里所使用的两个 bean 都是 BeanBox 发布的一部分, 并且作为 bean 开发示例而提供。

BeanBox 由如下工具组成: canvas、BeanBox 工具箱。如图 10-1 所示, canvas 是放置 bean 的工作区。

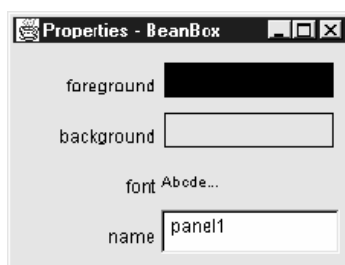


图 10-1 BeanBox canvas

ToolBox 是一个工具选用板，可以从其中拖放出 bean。在启动时，BeanBox 启动过程将分析一系列 JAR (Java Archive) 文件 (bean 的打包格式)，并且将所有可用的 bean 放置在 ToolBox 中，如下图所示。为了将一个新的 bean 添加到这个工具选用板中，需要做的工作就是将相应的 JAR 文件放在正确的目录中（默认为%BDK_HOME%/jars）。



属性编辑器允许设置 bean 的属性值。这个编辑器总是反映关于当前所选择属性的值；在下图的编辑器中，显示的是默认容器的属性，这个容器也是 BDK 发布的一部分。

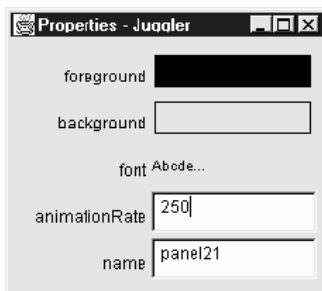


首先，将 3 个 bean 拖放到 canvas 中，如图 10-2 所示。



图 10-2 将 bean 放在 canvas 中

可以注意到，juggler 进行 juggle 的速度过快，因此，用户可能会希望修改执行速度。为此，应该单击 Juggler bean，并查看属性编辑器：



在 juggler 的属性中，有一项是用来设置 juggle 速度的。introspection 机制允许 BeanBox 知道这个属性并且将这个属性放置在编辑器中。为了降低速度，请将 250 修改为 125。然后，分别将按钮的标签设置为“Stop”与“Start”；按钮 bean 的属性中有一项就是它的标签，因此可以使用属性编辑器设置正确的标签。

在单击一个按钮之后，将产生一个动作事件。现在，需要挂接（hook up）这两个按钮被单击时所产生的事件，以便使其调用 juggler 的方法——startJuggling 与 stopJuggling。正如许多现代 IDE 一样，BeanBox 允许可视化地完成这些工作。在提供这种支持的背后，所发生的事情是在 juggler 与每个按钮之间放置了一个适配器。这个适配器用作所执行动作事件（由按钮产生）的侦听程序；当这个事件被触发之后，它将调用适当的 juggler 方法（juggler 定义的 API 的一部分）。

对用户来说，需要做的工作是很简单的。BeanBox 允许用户在不编写一行代码的情况下完

成所有这些任务。首先，选择“Start”按钮，然后选择 Edit|Events|button click|actionPerformed。这样就可以定义自己感兴趣的事件。

然后定位到 Juggler bean。这时，将弹出构成这个 juggler 的 public 接口的方法列表，如图 10-3 所示。选择 startJuggling，这时系统将创建并编译适配器。对 Stop 按钮重复该过程。这时就可以结束这一次小实验。现在，已经有了一个小的可工作应用程序，用户可以保存并运行它；这项技术难道不是很神奇吗？

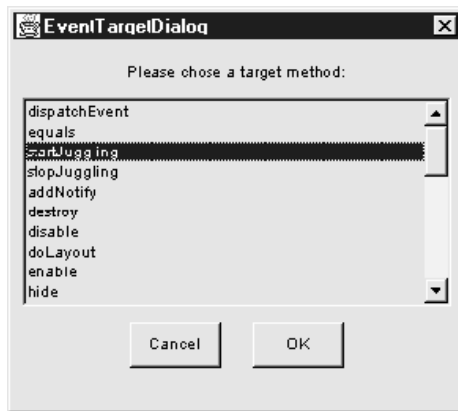


图 10-3 选择将要激活的方法

10.3 introspection

introspection（内省）是一种这样的过程：通过该过程，开发环境能够发现一个 bean 所支持的属性、事件与方法。在上一节的例子中，读者已经见到了这种发现过程的功效。通过了解 bean 的属性，BeanBox 就能够提供正确的属性表、列出 bean 可以执行的动作的事件列表。这个事件列表允许用户选择一个将要被注册的动作，并允许用户知道自己能够选择的方法。BeanBox 对按钮 bean 与 Juggler bean 使用 introspection 来启用这个可视化的编程过程。

JavaBean introspection 的基本目的是提供一种手段，通过这种手段能够检测一个 bean 的功能。这并不是一个全新的概念；在许多环境中，都使用一种定义语言或规范语言来定义功能。例如，IDL 就是一种规范语言，并且在 IDL 规范中指定的信息将作为活对象存储在 IR（Interface Repository，接口库）中，而 IR 能够被具有浏览功能的工具所访问。JavaBean introspection 使用 Java 语言本身，从而不引入一种新的语言 这样就使得整个定义完全由 Java 完成。

可以通过两种机制之一来实现 Java introspection，分别是：低级的实现方法与 BeanInfo 接口实现方法。低级的实现方法依赖于约定。JavaBean 规范定义了一系列约定，用于提供对属性、事件与方法进行访问的命名方法。低级的 introspection 机制使用 Java 反射来访问这些

名称。基于这些约定，就能够在运行时决定 bean 支持的是哪些属性、事件与方法。这种默认的低级机制对于大多数简单 bean 是很有用的。但对于更复杂的 bean，使用这种机制还不够，因为一个复杂的 bean 可能需要分别在不同的环境下提供不同的服务，或者包含有如此丰富的接口以至于对简单编程任务难以有用。为此，JavaBean introspection 定义了第二种机制——BeanInfo 接口——虽然这种方式需要 bean 开发人员进行更多的工作，但它提供了一种很好的控制途径，用于控制 bean 应该为外部世界提供哪些功能。BeanInfo 接口提供了用来描述一个 bean 的属性、事件与方法的方法。bean 开发人员能够创建一个类，用于实现 BeanInfo 接口以描述 bean 的功能。下面的程序清单显示了一个简单类，这个类为 PagerRequest bean 实现 BeanInfo 接口。PagerRequestBeanInfo 类扩展了 SimpleBeanInfo，而 SimpleBeanInfo 是 java.beans 包中的一个实用程序类，它为每个所需的方法提供了一个 null 实现。所创建的 bean 将有 3 个简单的属性（Telephone Number，PIN 与 Message）、一个用于对数值进行 Page 的方法、以及一个当 pager 接收到消息时将会产生的事件。需要说明的是，这里并没有显式地定义这些方法，这意味着可以对 bean 的某些功能使用显式的 BeanInfo 定义，而对于其他功能则使用默认的命名约定。

```
import java.beans.*;

public class PagerRequestBeanInfo extends SimpleBeanInfo {

    public BeanDescriptor getBeanDescriptor() {
        BeanDescriptor _descriptor =
            new BeanDescriptor(PagerRequest.class);
        return _descriptor;
    }

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            PropertyDescriptor _propertyDescriptors[] = {
                new PropertyDescriptor(
                    "Telephone Number" ,
                    String.class),
                new PropertyDescriptor(
                    "PIN" ,
                    String.class),
                new PropertyDescriptor(
                    "Message" ,
                    String.class)
            }
        }
    }
}
```

```
        };
        return _propertyDescriptors;
    } catch (IntrospectionException e) {
        e.printStackTrace();
        return null;
    }
}

public int getDefaultPropertyIndex() {
    return 0;
}

public EventSetDescriptor[] getEventSetDescriptors() {
    try {
        EventSetDescriptor _eventDescriptors[] = {
            new EventSetDescriptor(
                PagerRequest.class,
                "Acknowledge Received",
                java.awt.event.ActionEvent.class,
                "acknowledgeReceived"
            )};
        return _eventDescriptors;
    } catch (IntrospectionException e) {
        e.printStackTrace();
        return null;
    }
}

public int getDefaultEventIndex() {
    return 0;
}

public java.awt.Image getIcon(int iconKind) {
    java.awt.Image img = loadImage("PagerRequest.gif");
    return img;
}
}
```

由于 bean 具有类的形式,并且由于类能够从另一个类进行继承,因此, introspection 是一个多阶段的处理过程,能够遵循继承树的规则。对于 bean 的每个超类,JavaBean introspection 都将试图确定是否有一个 BeanInfo 对象可获取。如果是这样,它将把这些信息加入到 bean 的完全描述中。否则,它将激活低级反射机制并添加基于约定的定义到 bean 描述中。

10.4 属 性

属性就是 bean 的已命名特性。一个属性具有一个名称与类型,并且由 bean 中提供的 accessor 方法所定义。如果对这个方法进行读写,那么就需要实现两个 accessor (get 方法与 set 方法)。只读属性仅仅需要 get 方法。正如 CORBA IDL 特性一样,一个属性并非一定表示 bean 具有一个数据成员与其对应(但通常有一个数据成员与其对应);属性可以被继承(或计算)并且能够涉及复杂的业务逻辑。

对于低级 introspection,与属性相关的约定是:对于一个名为 <PropertyName> 的 <PropertyType> 类型的属性,应该定义两个如下形式的 accessor:

```
public <PropertyType> get<PropertyName>()  
public void set<PropertyName>(<PropertyType> arg)
```

如果这个属性是只读的,那么仅仅需要定义 get accessor。当没有提供 BeanInfo 类,并且 introspection 机制发现一对上述的 public 方法时,它能够知道这个 bean 的属性。

对于一些常见的情况,还可以定义特殊处理方法。例如,如果属性具有 boolean 类型(这是很常见的),那么应该按照以下形式定义 get 方法:

```
public boolean is<PropertyName>()
```

如果一个属性是数组类型(也就是 <PropertyType>[]),那么应该使用的模式是:

```
public <PropertyType> get<PropertyName>(int index)  
public void set<PropertyName>(int index, <PropertyType> arg)
```

JavaBean 还定义了两个属性类型,分别是 bound 属性与约束属性。bound 属性(bound property)支持当一个属性的值发生改变时进行自动通知功能;约束属性(constrained property)允许使用者在当属性值发生改变时执行有效性检查。后面将对 bound 属性与约束属性进行更详细的讨论,因为只有在理解了事件这一概念之后才能使用它们。

10.5 方法与事件

默认情况下,bean 的所有 public 方法都作为 bean 的 API 提供。它们包括所有的属性 accessor 与事件侦听程序订阅方法;因此,一种聪明的做法是在 BeanInfo 中定义方法签名。

JavaBean 事件模型遵守 JDK 事件模型,定义了从源对象到一系列侦听程序的通知传播框架。传播 (propagation) 就是在满足一系列命名约定的侦听程序对象中定义 Java 方法的激活。处理事件的 Java 方法被集中在称为事件侦听程序接口 (event listener interface) 的接口中,这些 Java 方法的签名与命名约定相对应。

将事件用户挂接 (hooking up) 到事件生产者,要求事件源 (生产者) 与侦听方 (用户) 都遵守相同的 JavaBean 事件模式。事件源必须提供事件用户可用来注册感兴趣事件的方法。它们是 add 方法与 remove 方法。例如,如果某个对象希望支持发布 ActionEvent,那么这个对象就应该实现一个 addActionListener 方法与 removeActionListener 方法。利用这些方法,事件用户就能够在事件源对象上注册它们感兴趣的事件类型。为了完成处理,事件用户必须实现在适当的侦听程序接口 (addListener 方法的参数) 中定义的方法;这也是当事件被触发之后源对象将调用的方法。

关于事件如何产生以及该事件如何被传播到侦听程序,这些实际的指定并不重要,在 JavaBean 规范中也未限制具体的实现方法。例如,大多数 bean 的实现都维护有一个事件用户集合,并且使用 add 方法与 remove 方法来管理这个集合;在这种情况下,触发一个事件就意味着在这个集合中进行迭代并分别调用每个用户。但是,在 JavaBean 规范中并没有要求对这样的集合进行管理,并且,可能有许多可选的实现方案。惟一真正重要的规范就是事件发布者与事件用户之间的协议。

源对象触发一个事件就意味着需要创建一个事件对象 (从 java.util.EventObject 继承的类的一个实例)、在这个事件对象中插入关于状态改变的信息、调用感兴趣的每个侦听程序对应的通知方法。事件对象是不可修改的,并且不应该被任何侦听程序对象所改变。由于相同的事件对象将被传递给所有的侦听程序,因此任何对事件对象的修改都可能影响其他侦听程序对该事件的正确处理。

正如属性与方法一样,低级 JavaBean introspection 使用反射与命名约定来决定哪些事件可以被源对象触发。一对下列形式的方法,隐含地表示 bean 创建 <EventType> 事件,该事件通知实现 <EventListenerType> 接口的侦听程序。

```
public void add<EventListenerType>(<EventListenerType> ev)
public void remove<EventListenerType>(<EventListenerType> ev)
```

JavaBean 中支持的一个特殊情况定义了命名约定仅仅用于单点传送事件源。默认的事件处理范型允许处理单独事件源的多个事件用户。如果这个事件源模型仅仅支持一个单独的侦听程序,那么这个方法定义就应该具有下列形式:

```
public void add<EventListenerType>(<EventListenerType> ev)
    throws java.util.TooManyListenersException
```

最后,当事件源产生一个事件时,它将对所有提出申请的侦听程序进行迭代,并且对每个侦听程序调用下列形式的方法:

```
public void <EventOccurrenceMethodName>(<EventStateObjectType> ev)
```

事件对象封装了与状态改变相关的信息。

在 JavaBean 模型中定义的事件传递语义，是一种关于事件源的同步传递。源对象发出的对事件用户的调用将在调用者的线程中完成。如果某个侦听程序失败（以某种方式导致线程失败），那么后续的侦听程序将不能获得通知。

同步事件传递语义是约束属性的基础。约束属性是对 bound 属性的加强——这两种属性都是将属性与事件进行耦合的结构，它们是如此常用以至于在 JavaBean 中拥有特殊的地位。bound 属性是一种对另一个对象有用的属性——这另一个对象通常是 bean 的容器。这一特点对于将定制的行为“绑定到”属性值的修改非常有用。约束属性类似于 bound 属性，表现在某个对象可能对属性的改变有兴趣，但约束属性还能进一步支持属性值的有效性检查（如果认为属性值的改变是不合适的，就允许有兴趣的对象拒绝接受这个改变）。同步语义对于实现约束属性很重要，因为一个侦听程序可能通过产生异常来拒绝属性的改变，从而终止事件处理线程的正常处理过程。

10.5.1 bound 属性

如前所述，bound 属性允许将一个属性与 onChanged 通知类型进行耦合。为了提供 bound 属性，bean 需要为 PropertyChangeListener 类型支持一对多播侦听程序注册方法：

```
public void addPropertyChangeListener(PropertyChangeListener l)
public void removePropertyChangeListener(PropertyChangeListener l)
```

当改变发生时，bean 将调用侦听程序的 propertyChange 方法，并且向这个方法传递一个 PropertyChangeEvent 对象。在 PropertyChangeEvent 对象中包括发生改变的属性的名称、原来的值以及新值。

如果希望注册一个单独的已命名属性，JavaBean 也支持下列方法：

```
public void addPropertyChangeListener(
    String propertyName, PropertyChangeListener l)
public void removePropertyChangeListener(
    String propertyName, PropertyChangeListener l)
```

只有在改变发生以后，才执行对侦听程序的 propertyChange 方法的调用；事件用户对属性的修改并没有任何决定权。下面的例子使用来自 Swing 库（com.sun.java.swing.JInternalFrame）的内部帧的 bound 属性。当一个内部帧被最大化时，属性将发生改变并且用户将被通知到这个改变。

```
//
// When the screen is opened, set up the listener to be
// notified when the internal frame's properties change.
// One of these properties is the isMaximized property, which
// is of interest.
```

```
//

public void postOpenWindow(com.sun.java.swing.JInternalFrame win) {

    OnMaximizeOrMinimize aListener = new OnMaximizeOrMinimize();
    win.addPropertyChangeListener(aListener);

}

//
// When called, check whether this is the property of
// interest to us and, if so, call the handling routine
// (not shown here).
//

class OnMaximizeOrMinimize implements
    java.beans.PropertyChangeListener {

    public void propertyChange(java.beans.PropertyChangeEvent ev)
    {
        if (ev.getPropertyName().compareTo(
            com.sun.java.swing.JInternalFrame.IS_MAXIMUM_PROPERTY) ==
            0) {
            boolean aBoolean =
                ((Boolean)ev.getNewValue()).booleanValue();
            setBasedOn(aBoolean);
        }
    }

    public void setBasedOn(boolean aBoolean) {
        // ...
    }

}
```

10.5.2 约束属性

约束属性对 `bound` 属性进行了扩展，从而支持有效性检查。在此情况下，事件用户不是在属性改变之后才被调用，而是在属性改变之前被调用，并且事件用户被赋予禁止属性发生改变的权利。事件用户能够产生一个 `PropertyVetoException` 异常，从而阻止属性改变的发生。由于使用的是同步事件语义，因此 `bean` 就可以确保直到所有侦听程序都被调用并且为所有

这些侦听程序提供了禁止修改发生之后，才执行属性值的修改动作（显然，不能有任何事件用户禁止修改，否则属性值将不会改变）。约束属性中使用的命名约定包括下列形式的 add 方法与 remove 方法：

```
public void addVetoableChangeListener(VetoableChangeListener l)
public void removeVetoableChangeListener(VetoableChangeListener l)

public void removeVetoableChangeListener(
    String propertyName, VetoableChangeListener l)
public void addVetoableChangeListener(
    String propertyName, VetoableChangeListener l)
```

可以注意到，在侦听程序接口中的事件处理方法已经被定义为能够发出 PropertyVetoException 异常。

约束属性的复杂之处在于，如果某个事件用户禁止属性改变，那么有可能在这时其他事件用户已经被通知该属性的改变并且已经采取了相应的动作。由于这样可能会在拒绝改变发生时导致不正确的应用程序行为，因此，在当属性被修改回原来的值时必须通知这些事件用户。因此，当 bean 处理由某个事件用户发出的禁止改变的异常时，它必须触发另一个用来说明“可以修改”的属性改变事件，从而将属性值修改回原来的值。

10.5.3 事件适配器

事件适配器是一个插入在源对象与事件用户（或者说事件生产者与事件消费者）之间的对象。事件适配器对于事件的正确传播并非必要，但使用它们会使某些工作非常方便。这种方便性表现在两个方面。

首先，事件适配器允许将事件用户与事件源分离出来。这样，就不必强制事件用户实现为了保证事件能够从事件源传递到事件用户所必需的正确的方法集合，因为事件适配器能够充当“转换器”，从而允许事件源与事件用户保持相互独立。由于不同的组件常常是由第三方所开发的，从而使得甚至不可能实现事件源与事件用户之间的耦合；在这种情况下，就必须使用事件适配器。在图 10-4 的顶部，显示了事件适配器的基本构成，图的底部显示了运行时究竟发生了什么。在这个例子中，page 系统是一个由第三方所购买的事件用户。它有一个使用电话号码与 PIN 的 API。这个例子中的源是一个调度队列，当出现调度请求时，该调度队列将发出一个事件。事件适配器实现了侦听程序 API，这些 API 可确保添加侦听程序；并且可确保 page 系统能够调用 API 转换。

其次，事件适配器可以执行应用程序所需的附加操作，例如过滤事件、实现安全性策略、实现排队（例如，CORBA Event Service 有一个称为 Event Channel 的概念，用户在应用程序中部分实现它，用作事件适配器）。另一个可以说明事件适配器有用性的例子就是事务。考虑这样一种情况，在相同的节点上有许多事件用户，当事件源触发一个事件之后，所有事

件用户都必须执行某些操作。同时，还假设所有这些操作都需要在相同的原子性事务代码（也就是，这些事务代码要么全部执行，要么都不执行）中发生。这样，就必须有一种方法来协调这个事务。这个协调器通常不应该是源对象——因为源对象并不一定是内部开发的（也就是不能修改源对象），或者因为相同组件的其他用户并不关心事务。显然，它也不能是任何一个事件用户，因为我们无法预测事件被通知的顺序（任何试图使用某个事件用户来完成这种同步的设计都是凌乱的并且是非可重入的）。这一任务显然由事件适配器来完成是最合适的。正如图 10-5 所示，事件适配器能够为事件用户管理这个事务。

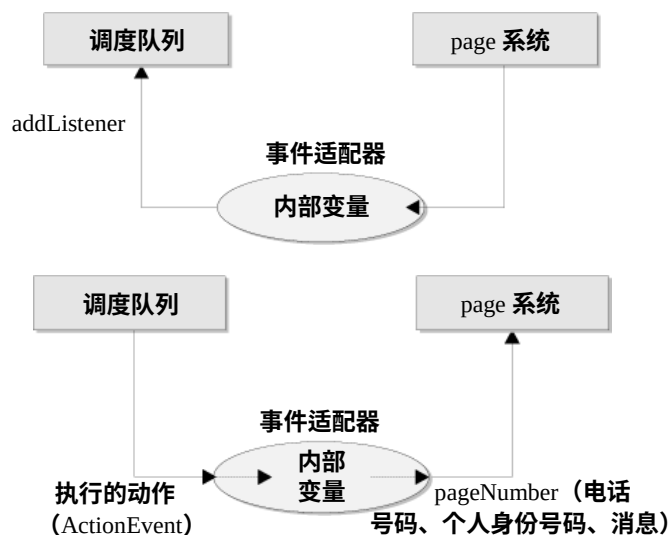


图 10-4 事件适配器设置（顶部），运行时的事件适配器（底部）

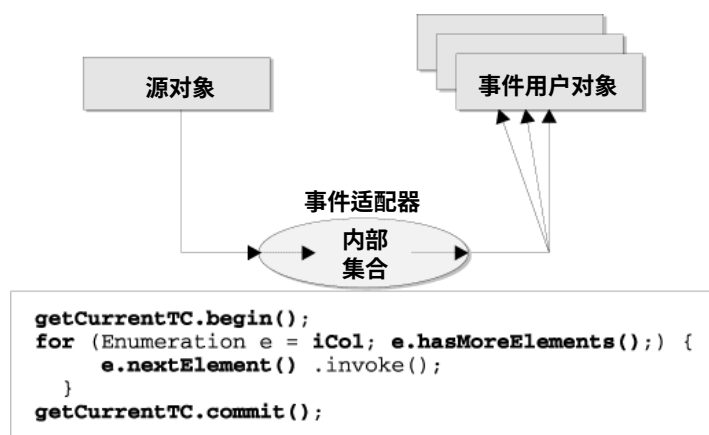


图 10-5 充当事务协调器的事件适配器

10.6 定 制

属性编辑器对于简单 bean 是很有用的。但对于复杂 bean，或者对于具有复杂数据类型作为属性的 bean，默认的属性编辑器可能就不够用。对于这种情况，JavaBean 定义了定制器（customizer）这一概念。定制器是一个类，它为 bean 用户提供了一种方法，用来在设计时为 bean 的属性提供值。定制器可以是一个向导、特殊化的用户界面，或允许输入复杂数据类型的编辑器。

定制器需要从 `java.awt.Component`（直接或间接）继承，因为这个定制器被 bean IDE 用作 GUI。此外，它还必须实现 `java.beans.Customizer` 接口。在这个接口中的方法使用属性值改变事件来管理与 IDE 其他部分的交互。

将一个定制器附加到某个 bean，意味着需要使用 `BeanInfo` 对象。在从 `BeanInfo` 对象获取了关于 bean 的描述之后，使用 `getBeanDescriptor()` 方法来调用 `getCustomizerClass()` 方法，通过调用 `getCustomizerClass()` 方法，就能够获取 bean 的定制器。

10.7 持 续 性

当在 IDE 中使用 bean 时，用户通过设置属性、附加事件等等来定制 bean。这些信息必须被导出到某种可持久存储的介质上，从而使得系统能够在运行时正确工作。JavaBean 使用 Java 对象串行化作为存储与读取 bean 的机制。虽然规范中定义了用来使 bean 保持持续性的其他可选方案，但局限于当前市场上产品的约束，用户只能使用串行化这一方法。

10.8 小 结

在阅读本章之后，读者应该较好地掌握了 JavaBean 组件模型以及用于构建 Java 类的优选方法。只要遵守这个模型中定义的约定，就能够确保用户代码与元数据不仅能够其他工具与框架的上下文中使用，而且（可能这是更重要的）能够确保用户代码更容易阅读与理解。遵守 JavaBean 模型中定义的约定，对于编写可视化组件以及编写那些将被用于可视化创作 IDE 环境的组件显得尤为重要；而且，在我们看来，遵守这个约定总是重要的。下一章将进行更深入的讨论，也就是描述 EJB 组件模型。这个模型更加完善，并且提供更多的服务。

第 11 章 Enterprise JavaBean

读者已经对 JavaBean 有了较好的了解，下面将继续讨论对于使用 Java 构建服务器方功能最为重要的组件模型：EJB（Enterprise JavaBean，企业 JavaBean）。本章将讨论 EJB 体系结构以及构成这个体系结构的许多组件。它描述了需要构建哪些代码，以及由谁来构建这些代码。bean、容器与服务器的共同工作，来支持客户完成任务；本章详细讨论了实体 bean 与会话 bean。读者不应该低估掌握 EJB 的难度。这个模型比较复杂，并且它将要求读者花费一定的时间来理解如何使模型中的各个组成部分共同工作。本章从最基本的内容开始，描述 EJB 模型，并且在后面的章中将提供在 WebSphere 中运行的 EJB 示例。

11.1 服务器方组件

EJB 是一个服务器方组件体系结构，它支持使用 Java 构建企业级分布式应用程序，并且简化这个构建过程。这个如此冗长的定义，其含义就是 EJB 提供了一种框架，允许用户在不必编写代码来处理关于分布式编程的复杂问题（例如通信与同步）的条件下，实现一个分布式 Java 应用程序。EJB 体系结构也管理可伸缩性、安全性以及可靠性等问题。总之，EJB 允许企业编程者关注于应用程序的特色内容，而不必担心应用程序的底层结构。EJB 服务器提供了用户所需的所有底层结构。用户可以选择使用任何厂商的 EJB 服务器，并且体系结构代码在服务器之间是可移植的。本书集中于讨论 WebSphere Application Server，但读者需要记住的重要一点是，EJB 的一个主要优点就是可移植性。这种可移植性来自 Java 是可移植的这一事实，并且它被所有 J2EE 机制（例如 JSP 与 servlet）所共享。

EJB 体系结构是由 Sun 公司引入的，作为将 Java 推进成为用作服务器方编程语言策略的基石。EJB 体系结构的目标是将系统级问题（例如通信、分布、事务、安全性）与实际的业务逻辑以及特定域问题相分离。EJB 的最主要目标就是让开发人员能够全力关注构建域功能。标准 API 以及标准体系结构的使用，允许开发人员依赖于关于系统问题以及体系结构问题的成熟的解决方案，而不必受限于某个特定的厂商。

在 EJB 规范中定义的关于 EJB 体系结构的目标是：

- 为基于 Java 的分布式业务应用程序的开发定义一个组件体系结构。
- 在提供企业级解决方案方面，支持混合使用来自不同厂商的组件。
- 支持专家为事务、安全性、分布与多线程以及其他系统级问题提供解决方案，从而使得域专家以一种他们认为更容易的方法来构建业务应用程序。

- 解决整个软件的生命过程，并且提供一些定义，使得按照这些定义能够在开发时在工具中使用 EJB 组件（非常类似于 JavaBean 所解决的运行时与设计时问题）。
- 定义与可互操作性相关的问题。特别地，EJB 依赖于 CORBA 实现分布式对象的可互操作性；这并不意味着 EJB 可互操作性仅仅与 CORBA 相关，实际上，许多其他 API 映射与传输也是可能的。

11.1.1 EJB 体系结构

EJB 体系结构试图借鉴上一章讨论的 JavaBean 的思想，并且在此基础上更进一步。它不是简单地提供一个用于构建可重用软件组件的框架，而是提供一个用于构建分布式、面向事务组件的框架。这种区别导致非常大的不同。其中之一就是，JavaBean 仅仅是一个为编程者提供服务的开发组件，而 EJB 则是一个可部署的（deployable）组件。在现实的分布式设置环境中，对象在物理机器上的放置通常是在部署时决定的，而不是在开发时决定的。此外，由于这种放置可以随着时间的推移而改变，很显然，能够在不修改软件的情况下对对象的放置进行简易的修改是非常可取的。

作为可部署对象，EJB 必须有一个标准的并且明确定义的部署环境。为此，EJB 体系结构定义了容器与服务器的概念。容器（containers）保存（或包含）有 EJB 实例。在 EJB 规范中定义了容器与 EJB 组件之间的各自职责以及它们的交互。EJB 容器包含在 EJB 服务器中。最初的 EJB 规范并没有给出 EJB 服务器与 EJB 容器的相互关系；而是将这个问题留给厂商自己解决。这样导致的结果是，人们并不清楚容器在何处结束以及服务器从何处开始。实际上，如果服务器仅仅有一个容器，那么在服务器与容器之间并没有什么真正的区别。随着 J2EE 平台的引入，这个问题得到了解决，本书将在第 15 章讨论 J2EE。在 J2EE 中，扩展了容器的概念，并且明确了容器与服务器之间的区别。

关于容器的一个有趣说明：WebSphere 3.5.x 允许创建多个容器，但并不建议这样做。在 WebSphere 3.5.x 中最佳的做法是为每个应用程序服务器提供一个容器。WebSphere 4.0 取消了这一限制，因此它能够提供更健壮的应用程序部署。

企业 bean 只能存在于容器中。不能直接访问企业 bean，也不能直接创建关于企业 bean 的实例。容器管理对企业 bean 功能的访问，并管理企业 bean 实例的创建与析构。由于 EJB 是被远程访问的分布式对象，因此实际的访问是通过名为远程 EJB 对象（remote EJB object）的特殊代理对象进行的。这些特点就允许 bean 开发人员将管理事务、一致性、通信与安全性的职责交给容器来完成。图 11-1 给出了 EJB 体系结构的示意图。

虽然企业 bean 与保存有这个 bean 的容器之间的关系是一个闭合关系，但 EJB 规范要求 EJB 能够在不同容器之间移植。也就是说，EJB 与容器必须是完全分离的。这就意味着，容器为企业 bean 提供的所有服务在 EJB 规范中都是明确定义的。

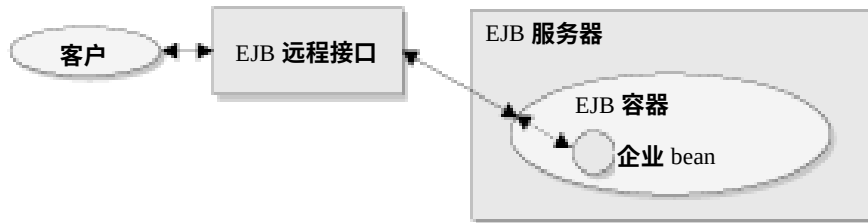


图 11-1 EJB 体系结构概图

为了为分布式面向对象的系统定义一个框架，EJB 体系结构必须在各种相互矛盾的需求之间寻找最佳平衡点。一方面，这个框架必须简单，从而便于理解与使用。另一方面，EJB 是面向企业级系统的——它是大型的、典型的任务关键型系统。特别地，企业系统有严格的性能与失败恢复要求，还有其他特殊的要求。

为了达到这些不同需求的最佳平衡，并且管理这些问题固有的复杂性，EJB 规范采取“分而治之”的策略。与 EJB 生命周期相关的任务与职责被划分给几个不同的功能组。与 JavaBean 不同（在 JavaBean 中只有两个功能组：bean 开发人员与 bean 用户），在 EJB 中有 6 个不同的功能组：bean 提供者（开发人员）、容器提供者、服务器提供者、应用程序开发人员、应用程序部署者以及系统管理员。这些功能组是按照角色进行划分的，其中有些角色是组合的。例如，应用程序部署者与系统管理员可能是同一个人；相同的厂商通常提供 EJB 容器与 EJB 服务器。下一节将详细讨论每个角色的职责。

为了允许不同的功能组之间相互通信，企业 bean 定义了一系列能够影响运行时行为的属性。EJB 属性可以作为元数据（或描述符）来分开，从而允许在设置与部署时进行定制，这类似于 JavaBean 定义 BeanInfo 对象并提供定制。但是，这些属性在本质上是不同的。JavaBean 属性通常与用户接口相关，而 EJB 属性则包括对与分布相关的问题（例如事务、安全性与持续性）的控制。

11.1.2 EJB 角色

EJB 规范提供了每个角色的职责的明确定义。通过明确定义每个功能组的职责，EJB 能够有效地支持分布式而且高度可伸缩应用程序的开发，在这种应用程序中，来自不同厂商的产品能够互操作。下面将解释这些 EJB 角色。

bean 提供者

bean 提供者的职责是开发 EJB。这个提供者可以是独立的开发人员，也可以是可重用软件组件厂商。bean 提供者的最重要职责是以可重用的方式设计 bean。虽然设计 EJB 体系结构的目标是支持可重用，但这并不意味着任何 EJB 就一定是可重用的。只有当正确设计之后，EJB 才是真正可重用的。bean 开发人员的主要设计选择在于“业务”方法（应用程序域中的方法，而不是其他涉及诸如持续性、事务处理等方法）。EJB 方法形成了企业 bean

与 bean 用户之间的合约。这些方法被归组在一个接口中，EJB 开发人员将发布这个接口，并且 bean 实现必须应用这个接口。

bean 开发人员必须提供另一个名为本地接口 (home interface) 的接口，应该为每个 EJB 类型定义本地接口。EJB 本地接口实现了常用的“工厂”设计模式。这个工厂负责实例化对象以及析构对象——在这个例子中就是 EJB 对象。本地接口的另一项职责就是 bean 查找 (bean lookup)。bean 查找用来获取关于一个 bean 的引用。这并非对所有类型的 bean 都是可行的，它仅仅对实体 bean (entity bean) 有用——实体 bean 就是有一个应用程序级标识的持续性 bean。从客户代码的角度来看，对象查找包括 2 个阶段：定位本地接口，然后使用它定位一个特定的对象。

作为对分布式编程支持的固有特性的一部分，客户软件并不直接使用企业 bean。它不是直接使用对象引用，而是保存对远程“proxy”或“stub”的引用，并且由这些引用与容器进行通信。容器为 EJB 提供了执行环境。由于容器负责管理系统级问题，因此它必须知道任何方法调用。

EJB 服务器提供者以及 EJB 容器提供者

从理论上讲，提供服务器的角色与提供容器的角色是不同的，但在实际应用中，服务器是使用容器的实现来提供的。但是，J2EE 体系结构允许将新的容器引入 J2EE 服务器中。

EJB 容器为 EJB 提供了执行环境。它为大型、分布式应用程序提供所需的所有服务，例如安全性服务、事务服务等等。容器处理 bean 生命周期以及方法调用的所有方面。从应用程序开发人员的立场看，由容器来提供所有这些服务这一事实具有极大的优越性。依赖于这些服务，EJB 开发人员能够忽略关于安全性、事务、资源管理、生命周期管理、持续性与多线程的问题。容器能够非常高效地完成这些工作。

为了允许厂商具有某种程度的灵活性，EJB 规范允许服务器或容器提供“扩展的服务”；这些服务不属于 EJB 规范的范畴。这些服务的目标可能是通过负载平衡与透明的失败恢复实现高级性能与可靠性，或通过 EJB 连接器支持遗留系统。WebSphere 提供了到 IBM 遗留系统（例如 CICS (Customer Information Control System, 客户信息控制系统)，Encina, IMS (Information Management System, 信息管理系统)，DB2 以及 MQSeries) 的连接器。显然，依赖于这些扩展服务的 EJB 可能在能够运行的 EJB 容器与服务器范围方面存在限制。需要说明的是，有些这样的服务已经被 J2EE Connector (见第 15 章) 所替代。

应用程序开发人员

应用程序开发人员是使用可用的 EJB 构建企业应用程序的人。可用的 EJB 可能来自商业厂商或独立开发者所提供的 Java 库。虽然可用 EJB 能够提供某些功能，但它们通常都以一种普遍的方式实现，并且，如果 EJB 来自多个不同的来源，它们可能并不能直接达到可互操作。因此，应用程序开发人员应该提供“胶水”，用来连接这些不同的组件。由于 EJB 是中间层软件，因此它们并不提供任何用户界面。应用程序开发人员应该实现某些用户界面，它们可以是任何 Java 界面（例如 JFC/Swing）；在 J2EE 模型中，通常使用 JSP 与 servlet 来

实现用户界面。用户界面代码将调用 EJB。此外，应用程序开发人员可能还需要开发一些 EJB，用来解决特定领域的问题，或者用来连接不同的组件。

应用程序部署者

应用程序部署者可以作为容器/服务器提供者与应用程序开发人员之间的桥梁。应用程序部署者将 EJB 物理地放置在服务器与容器中。部署者拥有部署 bean 的自由，从而使得他或她能够自定义安装以适合可用的资源及其配置。在此过程中考虑的典型因素就是性能。部署者试图这样放置 bean：使得存在尽可能多的“引用局部性”（换句话说，就是使得调用 bean 的客户能够尽可能地接近 bean），但与此同时还必须努力保持不同服务器的负载平衡。其他的考虑可能是技术因素，例如节点之间的通信带宽、防火墙的存在性。

部署者的另一个重要职责就是进行安全性设置。部署者能够为不同的 EJB 方法以及 Web 资源赋予不同的访问权限。这些权限的设置应该与组织的安全策略一致，并且不损害存储在系统中的数据的安全性。

系统管理员

系统管理员负责处理部署之后的应用程序在使用时所出现的任何问题。管理员可以使用各种监视工具；在 WebSphere 中，Administrative Console 提供了这样的工具。在执行过程中出现的问题可能来自各个方面，例如硬件、操作系统、Web 服务器、应用程序服务器，或应用程序自身。系统管理员应尽可能正确定位这些问题，然后解决问题，或者，按照需要将问题转发给部署者或应用程序开发人员。

11.2 EJB 组件

EJB 体系结构为构建分布式面向事务的组件提供了一个框架。与 JavaBean 一样，EJB 也是一个组件模型；例如，它定义了 EJB 组件的结构并且描述了 EJB 组件所在的环境——这是一个非常重要的概念。哲学中的一个著名命题就是，为了完全描述一个实体，就必须描述其周围的环境以及这个实体如何与该环境交互。实际上，这也是 EJB 体系结构所采用的方法。EJB 组件是一个实现业务逻辑的 Java 组件。当一个组件在 EJB 环境中存在时，这个 EJB 组件必须假设已经存在一个明确定义的环境以及一系列与其发生的交互。例如，EJB 规范定义了容器的职责以及由容器所管理的 EJB 组件的职责。此外，它也完全定义了这个容器与 EJB 组件之间的交互，从而定义了一个结构化环境可供 EJB 构建者使用。这就引出这样的任务：必须按照工程实现的原则（而不是按照艺术的原则）来构建 EJB 组件与基于 EJB 的系统；这就是计算机科学领域的最大目标，名为软件工程。

EJB 组件环境的基本原理如下：

- 开发一个企业 bean，用来实现业务功能。

- 会话 bean 通常处理域信息并执行业务功能，而实体 bean 则存储这些信息。
- 企业 bean 并非孤立的。它位于一个部署在服务器中的容器里。企业 bean 并不直接访问；容器负责管理对企业 bean 功能的访问。这样就允许对服务（如事务、安全性）进行管理的职责从企业 bean 转移到容器和服务上。例如，容器能够执行企业 bean 实例的创建工作。这样就允许容器来管理关于这个类的生命周期与事务等问题。
- 企业 bean 与容器是完全分离的（decoupled），这就意味着可以使用多个 EJB 兼容的容器来管理企业 bean 的实例。在 EJB 规范中明确定义了容器为企业 bean 提供的所有服务。
- 企业 bean 定义了影响其运行时行为的属性集合。这些属性在部署描述符（deployment descriptor）中捕获。

为了使客户软件能够使用 EJB 组件，它必须提供关于下列实现领域的支持：

- 对象标识：它是可用来惟一地标识对象的标识符（并非所有的企业 bean 都有一个在应用程序级可见的对象标识）。
- 方法调用：形成企业 bean 与客户之间合约的业务方法。这些方法被归组到一个接口中，bean 开发人员必须满足这个接口。
- 本地接口：为每个 EJB 类型定义的接口。在本地，只要这个 bean 有一个应用程序级标识，那么使用主键就能够标识每个 bean。因此，对象查找包括两个阶段：定位本地接口以及使用主键定位一个特定的对象。本地接口是由命名服务存储的，客户可以使用标准 JNDI（Java Naming and Directory Interface，Java 命名和目录接口）来访问本地接口。企业 bean 的本地接口是对 javax.ejb.EJBHome 接口的扩展。

客户软件并不直接使用企业 bean，并且不直接访问这个 bean 的功能（它通过一个管理大多数系统级问题的中间容器进行访问），因此，它必须时刻掌握在企业 bean 中发生的事情。这样带来的影响是，企业 bean 的每个容器也必须支持 EJB 规范中定义的接口。这种支持通常需要容器完成某些工作。例如，容器希望运行原子性事务代码或修改此代码，然后，它们将这一工作委托给企业 bean 中适当的方法。

使用企业 bean 的客户能够依赖于企业 bean 有惟一标识符（identifier）这个事实。创建标识符这一工作，有时是企业 bean 的职责，有时又是容器的职责。例如，如果 bean 代表一个持续性域对象，那么它将为这个对象提供标识符，容器则仅仅负责将这个标识符传递给客户（若被请求）。另一方面，bean 也可能是临时（transient）对象，这个对象被客户用来执行某些任务。这样的企业 bean 可能是客户在应用程序服务器上的扩展，并且当客户不再使用它时将超期作废。在这种情况下，bean 的生命周期取决于客户的生命周期，并且它的标识并没有被定义为 bean 自己的一个属性；但实际上，容器可能产生了一个可供客户使用的对象标识——通常在为客户构建 bean 时产生。

本地接口定义了一个或多个 create 方法以及零个或多个 find 方法。create 方法允许客户创建

关于 EJB 组件的新实例，而 find 方法则允许客户查找一个特定的 EJB。显然，企业 bean 必须支持这两种方法；例如，企业 bean 开发人员必须为每个 create 方法启动一个 ejbCreate 方法，并且必须为本地接口中定义的每个 find 方法提供一个 ejbFind 方法。此外，容器还提供了被客户所访问的系统级功能；其中有些这样的功能要求企业 bean 开发人员必须进行一些工作。

由于企业 bean 存在于 EJB 容器中，因此 EJB 规范也精确地定义了容器在管理企业 bean 时所使用的 API。容器是 EJB 的重要部分，因为它允许企业 bean 开发人员依赖于容器所处理的标准服务。在这些服务中，最重要的就是生命周期管理。虽然生命周期管理是容器的职责，但如果没有企业 bean 的完全协作，容器就无法进行生命周期管理。这种发生在企业 bean 及其容器之间的协作（或称为合约），在一系列必须由企业 bean 实现的接口中定义。特别地，企业 bean 必须实现一系列状态管理 callback（容器调用这些 callback 来通知企业 bean 关于某些重要改变的发生）作为生命周期操作的一部分。这些接口要么是 javax.ejb.SessionSynchronization 接口，要么是 javax.ejb.EntityBean 接口，具体取决于这个企业 bean 是会话 bean 还是实体 bean。企业 bean 及其容器之间合约的其他元素还包括 API（通过这些 API，一个容器就能够通知 bean 关于其当前上下文与环境的信息）。

最后，由于 EJB 面向组件开发的高端市场，因此存在一个关于从企业 bean 接口到 CORBA 的标准映射。这样就打开了 EJB 体系结构，使其能够成为真正的多厂商、多语言以及多平台体系结构。从 EJB 到 CORBA 的映射包括许多重要的类型：从接口到 CORBA IDL 的映射、EJB 原子性事务代码作为 OTS（Object Transaction Service，对象事务服务）事务上下文的传播、对 CORBA 命名服务的支持，以及在安全性上下文中对安全性信息的传播。通过定义这些类型的映射，EJB 使自己具有完全语义并且具有连线级可互操作性。

11.2.1 会话企业 bean

企业 bean 被划分为两类：会话企业 bean 与实体企业 bean。这两者的主要区别在于持续性。会话 bean 代表服务器方组件，用来实现业务逻辑但不是持续的，而实体 bean 则保证其状态的完全持续性。本节重点讨论会话 bean；下一节将详细讨论实体 bean。

会话 bean 代表客户实现业务逻辑。会话 bean 不是持续的，并且不被多个客户所共享；从这个角度看，会话 bean 是提供正确应用程序划分的方便机制。应用程序划分是分布式计算中的一个重要概念，即业务逻辑常常需要在不同的地点执行而且又必须为客户软件提供位置透明支持。关于这个问题的一个主要例子就是为了提高性能而进行应用程序划分。例如，假设有一个业务函数，它接收一小部分输入并且产生一个输出，但这个函数需要大量访问数据库。显然，图 11-3 中所示的应用程序划分将比图 11-2 所示的应用程序划分执行效果更好。会话 bean 允许应用程序开发人员对应用程序进行划分，从而使得某些业务逻辑处理过程在服务器上代表客户发生；因此，会话 bean 是客户进程的逻辑扩展。

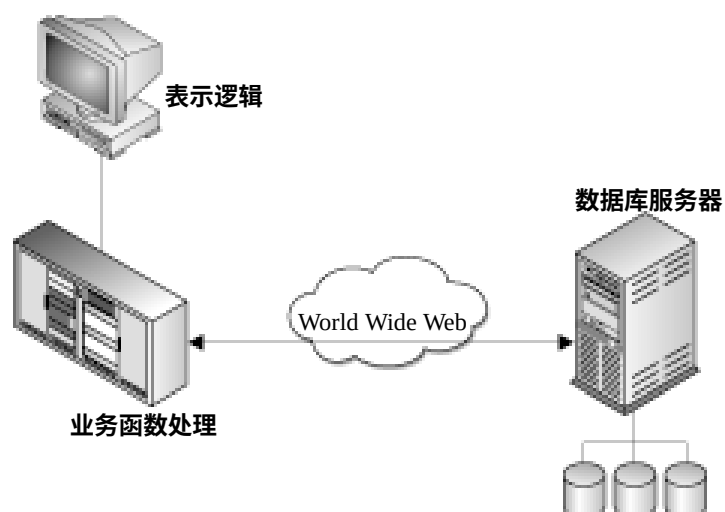


图 11-2 应用程序划分：方案 1

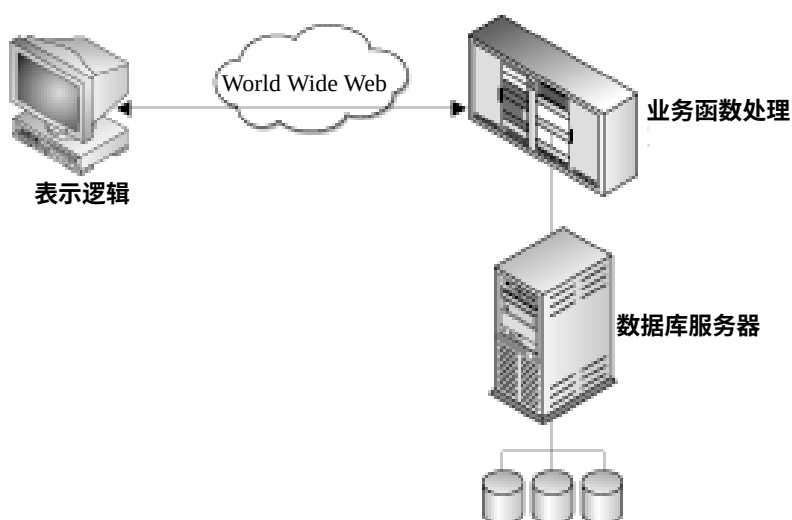


图 11-3 应用程序划分：方案 2

会话 bean 有对象标识；但作为一个非持续性实体，其对象标识也是非持续性的。因此，一个会话 bean 可能存活一会儿，为某些客户提供服务，并且在后来又重新用另一个标识复活。因此，会话 bean 的对象标识并不是一个在应用程序层次上有意义的描述符；这种对象标识最多只能被客户为了临时的方便性而使用，例如，为了便于将它放在一个由应用程序所管理的散列（hash）表中。需要注意的是，虽然会话 bean 在逻辑上可能再次复活并且为相同

的客户提供服务，但实际上，它是一个具有不同标识的不同对象，不能将它与能够支持复杂生命周期的实体对象混为一谈。会话 bean 的标识不能用作主键，并且一般来说不能提供给客户。

对会话 bean 的功能的访问总是通过 EJB 对象来进行的，这个对象使用 Java 的 RMI 将会话 bean 的接口作为远程接口提供出来，并且位于 CORBA 的 IIOP 层次之上。在 EJB 对象上进行的所有调用都被传递给企业 bean。会话 bean 的用户并不需要关心这个 bean 的精确位置，因为 RMI 与 CORBA 使得这些调用的远程特性是透明的。除了将域级方法调用传递给企业 bean 之外，EJB 对象还提供了一些方法来管理 EJB 对象自己（例如对象比较）。

与所有企业 bean 一样，会话 bean 存在于一个 EJB 容器中。这个容器负责管理该 bean 的生命周期、对 bean 方法的访问、安全性、事务等。

使用容器的查找功能，客户就能够访问企业 bean 的本地接口。本地接口类似于 BeanInfo 对象，提供了用来创建新 EJB 对象、删除 EJB 对象、访问企业 bean 元数据信息（实现 javax.ejb.EJBMetaData 接口）的函数，在设计时，这些函数对于工具是非常有用的。例如，下面显示了创建一个新 PartOrder 对象所需的代码段。需要注意的是，实际的创建方法（构造函数）特定于部分顺序，但所有创建方法都是使用本地接口调用的；实际的创建工作总是由 EJB 容器来管理。

```
Context clientContext = new InitialContext();

PartOrderHome partOrderHome = javax.rmi.PortableRemoteObject.narrow(
    clientContext.lookup("applications/services/parts/orders"),
    PartOrderHome.class);

PartOrder aPartOrder = partOrderHome.create(
    new Date(),           // creation date
    "AT&T",               // company name
    "Jane Doe",           // person ordering
    ...);
```

图 11-4 显示了一个会话 bean 的生命周期中通常经历的阶段的示意图。当一个本地接口的创建方法被调用时，就创建了一个会话 bean。企业 bean 在服务器上创建，并且将向调用者（EJB 对象）返回一个对象引用（可能是远程引用）。然后，客户将使用这个 bean 的句柄开始调用方法。

当客户使用完这个 bean 并且希望释放它时，就应该调用一个删除方法以析构这个 bean。可以使用 EJB 对象或本地接口调用删除方法。在两种情况下，所得到的结果都是这个企业 bean 不再被客户所引用；然后，客户应该释放这个引用。如果在企业 bean 不再存在时客户仍然使用这个句柄，那么将出现一个 NoSuchObjectException 异常。如果客户在调用某个删除方法之前已经释放了引用，那么这个 bean 将一直存在直到服务器退出——通过正常的关闭动作退出或者由于系统崩溃导致退出。在此期间，客户能够使用对 EJB 对象（句柄）的

getEJBObject 调用再次获取该句柄，这个调用将把状态返回到该企业 bean 存在并被引用时的状态。

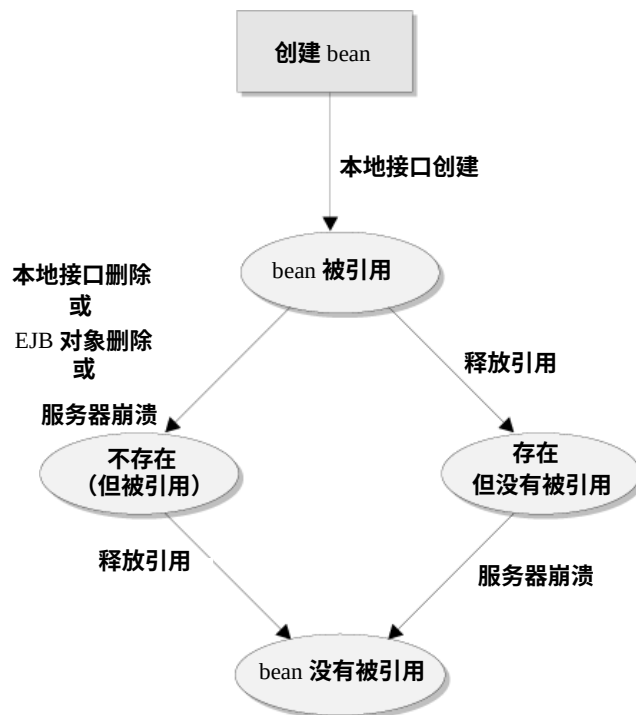


图 11-4 会话 bean 的生命周期

通过 EJB 对象以及通过容器进行的调用最终将到达企业 bean。一个会话企业 bean 应该实现在 SessionBean Java 接口中定义的下列方法：

- setSessionContext：被容器用来设置 bean 的上下文；这个上下文由容器来管理。SessionContext 对象提供了下列被企业 bean 使用的函数：
 - getEJBObject：返回句柄对象。
 - getHome：返回本地接口。
 - getEnvironment：返回环境属性。
 - getCallerPrincipal：返回调用客户的标识。这是一个允许 bean 基于调用对象的标识配置其应答的函数。例如，价格信息可能根据订单实体是内部客户还是外部客户而有所不同。因此，一个价格 bean 就可能需要知道关于请求价格信息的对象的详细信息。
 - isCallerInRole：为 bean 提供了一个测试方法，用来将调用者与某种角色匹配。
 - setRollbackOnly：允许 bean 确保当前事务可能不被确认，并且在这个原子性事务

代码中做的任何修改最终都会被一个回滚 (rollback) 动作所放弃。

- `getRollbackOnly`：返回一个 `boolean` 值，用以说明这个事务是否被标记为仅仅用来回滚。
- `getUserTransaction`：返回 `javax.jts.UserTransaction`，它提供了对事务的直接访问并且允许对原子性事务代码进行手工的处理。
- `ejbRemove`：允许 bean 执行任何清除或终止过程。由于会话 bean 不是持续性实体，因此这样的清除通常比较简单。容器负责 bean 整个生命周期的管理；因此，当容器删除一个 bean 时，它将调用 `ejbRemove` 方法。
- `ejbPassivate`：当容器试图取消一个 bean 的活性时，将调用这个方法，从而允许关闭资源或执行其他的准备工作，并且在再次激活这个 bean 之后，其状态仍然是正确的。
- `ejbActivate`：当一个 bean 被激活时，容器将调用此方法。然后，bean 就可以执行任何逆操作以准备它的状态。

会话 bean 也可以实现在 `SessionSynchronization` 接口中定义的附加方法。这些方法提供了同步挂接，从而允许会话 bean 优化其事务上下文方面的行为。

- `afterBegin`：当事务开始时被调用。
- `beforeCompletion`：在确认阶段开始之前、并且客户完成其工作之后调用这个方法。这样就允许 bean 在进行确认之前完成任何准备工作。
- `afterCompletion`：在事务完成后调用此方法，无论通过回滚动作还是通过成功的确认进行调用（根据作为参数传递的标志决定）。

虽然会话 bean 不是持续性实体，但它仍然能代表其客户管理状态；会话 bean 的非持续性特点仅仅表示这些状态在该 bean 被析构之后将不复存在。分布式对象应用程序通常可以被分类为遵守两种模型之一：无状态模型与有状态模型。在无状态模型中，服务器并不代表客户维护状态；在每次调用时，客户必须向服务器传递足够的信息，以确保服务器能够代表客户成功地执行操作。HTTP Web 模型就是一个无状态模型——也就是说，浏览器所发出的每个用来从 Web 服务器上获取信息的请求都是独立的，并且不依赖于先前的请求。在有状态模型中，服务器为客户维护一个会话，并且服务器维护着一些关于客户的状态参数。这两种分布式交互模型都有其存在的合理性，每种模型都有其更为适用的领域。因此，EJB 支持这两种模型。

当部署一个会话 bean 时，它将定义这个会话 bean 的容器应该如何对它进行管理。如果将这个 bean 指定为无状态，那么这个容器就被告知不需要维护任何对话状态，并且相同的 bean 可以被多个客户所使用（没有任何客户需要关心这个事实，因为在服务器上没有维护任何状态信息）。或者，容器可以被告知该 bean 是有状态的，在这种情况下，容器将代表客户管理 bean 的对话状态。对话状态 (conversational state) 被定义为 bean 的字段值以及从一个字段能够到达的每个对象。显然，由于基于对象应用程序管理非常复杂的对象服务，因此这

一个对话状态的定义能够非常容易地包括大量的对象。因此，对话状态独立于事务支持。一个会话 bean 的对话状态并不是事务性的，也不能在回滚 bean 的事务时对对话状态进行回滚。bean 开发人员负责在事务上下文中正确地管理对话状态。

需要说明的是，虽然会话 bean 不是持续性的，但从操作的原子性角度来看，它们是事务性的；惟一的不同在于事务输出并不保证持续性——持久的（durable），事务的首字母缩写定义（ACID（Atomicity, Consistency, Isolation, and Durability，原子性、一致性、隔离性、持久性））中的第 4 个单词。

无状态 bean 应该被看作服务提供者池中的工人。将工人指派到客户上是随意的，并且创建无状态 bean 的新实例将简单地被添加到池中。在本地接口中用来创建无状态会话 bean 的惟一被允许的创建方法是不使用任何参数的那个创建方法。工人池的大小由容器所控制，并且基于负载参数和可用时间来决定。

有状态会话 bean 具有更精细的生命周期，如图 11-5 所示。当客户调用本地接口上的一个 create 方法时，将创建 bean。然后，容器将调用 newInstance 方法来创建这个 bean 并使用 setSessionContext 设置上下文。最后，ejbCreate 被调用，从而向客户返回一个 EJB 对象。这时，该 bean 就可以被客户所使用。

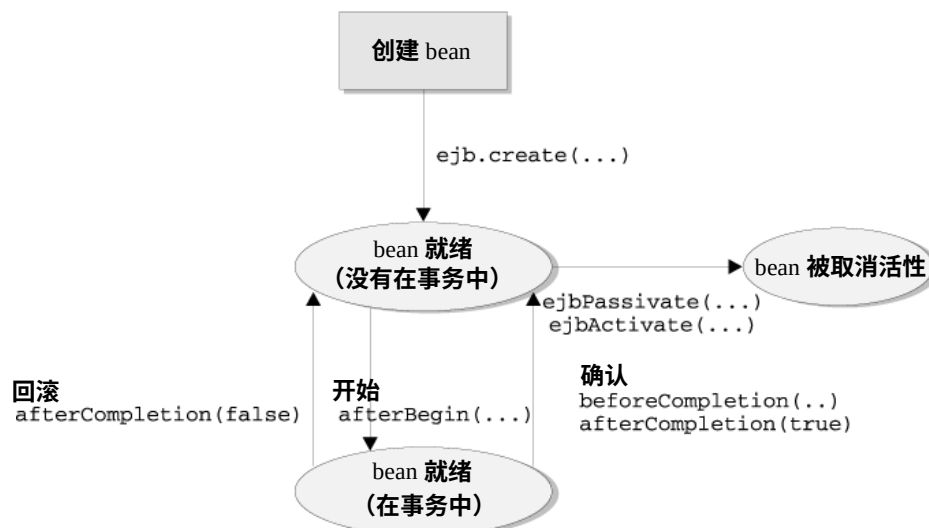


图 11-5 有状态会话 bean 的生命周期

只要客户进行非事务调用，那么这个 bean 将维持在非事务状态。当一个事务方法被调用时，这个 bean 就被包括在事务中；这时，afterBegin 被调用，从而允许该 bean 执行任何 bean 特定处理。如果其他事务方法被调用，这个 bean 将维持在事务状态。如果在该 bean 处于事务状态时调用了非事务方法，那么将标记出一个错误。

如果事务将要确认，容器将进行 `beforeCompletion` 调用，用来确保在进行实际的确认之前被传递。当事务将要回滚时，也将发生相同的事情。最后的事务状态将作为参数传递给 `afterCompletion` 调用。

虽然 bean 处于非事务状态，容器可能决定将 bean 移出内存。在这种情况下，它将调用 `ejbPassivate` 以允许 bean 释放任何当前占用的资源并且准备将该 bean 移出内存。当客户后来请求这个 bean 所提供的服务时（记住，这里讨论的是有状态 bean），容器将再次激活这个 bean 并调用 `ejbActivate` 方法，从而使得该 bean 能够准备就绪。

11.2.2 实体企业 bean

实体企业 bean 代表以某种形式存储在持续性介质上的对象。实体 bean 包括在对象的不同使用之间可以保存的状态信息。作为持续性对象，实体 bean 具有一个在对象创建的同时所创建的对象标识，并且这个对象标识在该对象的整个生命周期中保持不变。对象标识可用来查找这个对象，并且作为对对象进行识别与比较的主键。

实体 bean 生活在管理生命周期、事务与安全性的 EJB 容器中。此外，容器还管理实体 bean 的持续性。多个客户可以并发地使用相同的实体 bean。它们之间的同步由实体 bean 所在的容器管理。客户使用本地接口访问实体 bean，这时，将返回作为 bean 的句柄的 EJB 对象。在这个方面，实体 bean 的框架与会话 bean 相同。

与会话 bean 一样，实体 bean 也需要提供创建方法与查找方法的本地接口。但与会话 bean 不同的是，实体 bean 的本地接口必须实现 `findByPrimaryKey` 方法。该方法使用主键提供对实体 bean 的访问，这可能是因为实体 bean 是一个能够由主键唯一地标识的持续性对象。典型的例子包括帐号编码、偏序编码、国家名。主键自身是一个对象，并且可以是任何类型，但它必须是可串行化的。需要说明的是，主键仅仅在本地接口中具有唯一性。因此，本地接口与主键值的组合能够唯一地标识一个对象；实际上，如果两个对象具有相同的本地接口与相同的主键值，那么这两个对象被认为是相同的。

通过调用由 EJB 对象所实现的 `getPrimaryKey()` 方法，能够访问主键。对于给定的两个 EJB 对象，可以使用下列代码来判断它们是否相同。

```
obj1.isIdentical(obj2)
```

如前所述，实体 bean 代表一个持续性实体，并且是驻留在另一个系统（该系统处理关于持续性的细节）中的对象的应用程序扩展。例如，存储在关系数据库中作为记录集的对象、与负责持续性细节的遗留应用程序进行交互的对象。

实体 bean 的持续性有两种形式：BMP（Bean-Managed Persistence，Bean 管理的持续性）与 CMP（Container-Managed Persistence，容器管理的持续性）。对于 BMP，bean 自己负责实现对象的持续性。例如，bean 开发人员可以使用 JDBC（Java Database Connectivity，Java 数据库互连）或 JSQL（Java Structured Query Language，Java 结构化查询语言）调用来直接在

bean 中实现加载与保存功能。对于 CMP，容器代表 bean 执行数据库访问。为了使容器能够构建适当的数据访问调用，它需要从 bean 得到描述性元数据。EJB 规范利用实体 bean 提供的 `containerManagedFields` 部署描述符来完成此功能。

上述每种持续性机制都有其优点和缺点。BMP 对于某些特殊的行为是必要的；但是，用户必须自己进行所有的工作。BMP 同时也意味着需要在 bean 中进行大量困难的编程工作，并且不能维持原有的持续性。CMP 延迟与数据存储介质的耦合直到 bean 被部署在某些容器中，这样就使得实体 bean 更具可重用性。

图 11-6 演示了实体企业 bean 的生命周期。当容器调用 `newInstance` 方法时，就创建一个实体企业 bean，然后，通过调用 `setEntityContext` 方法来设置上下文。这时，该实例就变成池的一部分；它已经被创建，但还没有与某个 EJB 对象相关联，也没有任何特殊的数据。在这个池中的所有实例都是等价的。下一个变化就是进入就绪态；可以通过使用 `ejbCreate` 方法，或者在进行 `ejbFind` 方法的调用之后调用一个 `ejbActivate` 方法，从而进入就绪态。在调用 `ejbCreate` 方法之后，容器将立即调用 `ejbPostCreate` 方法。在 `ejbPostCreate` 方法被调用时，对象的标识就已经可用；这个调用允许 bean 执行可能需要使用对象标识的进一步动作。这里所讨论的第一种情况将创建一个新对象来为客户提供服务，而第二种方法则允许使用一个已有的对象。当该对象进入就绪态之后，容器将激活 `ejbLoad` 方法以便从持续性存储介质中获取 bean 的数据；此后，容器可能周期性地使用 `ejbStore` 保存数据，或者使用 `ejbLoad` 更新数据。当这些工作完成之后，bean 将使用 `ejbPassivate` 方法或 `ejbRemove` 方法以重新返回到池状态。一旦返回到池中，实体 bean 就不再有对象标识，并且在下一次活动中可能被分配给另一个新对象。

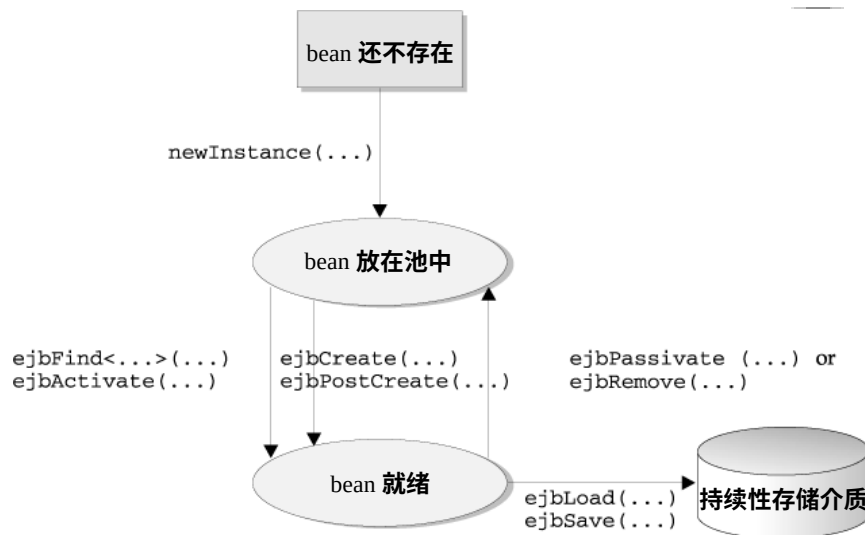


图 11-6 实体 bean 的生命周期

由于实体 bean 代表域对象，因此一个实体 bean 可能被 2 个并发的客户同时访问。并发访问的同步是容器提供给 bean 用户与 bean 开发人员的服务之一。容器必须负责确保使用相同实体 bean 的并发客户之间的正确同步。图 11-7 显示了两种可能的管理策略。在这个图的顶部方案中，容器将管理通过相同 EJB 对象访问的两个实体 bean。这两个 bean 生活在不同的事务上下文中——也就是，每个事务都属于不同的客户。最终的同步是调用 ejbLoad 方法与 ejbStore 方法时由数据存储介质执行的。对于这个图的底部情况，容器通过 EJB 对象管理对 bean 的访问。只有一个 bean 存在，并且通过这个 EJB 对象对其功能的访问是串行化的。在大多数情况下，第一种策略更加可行，这是因为：

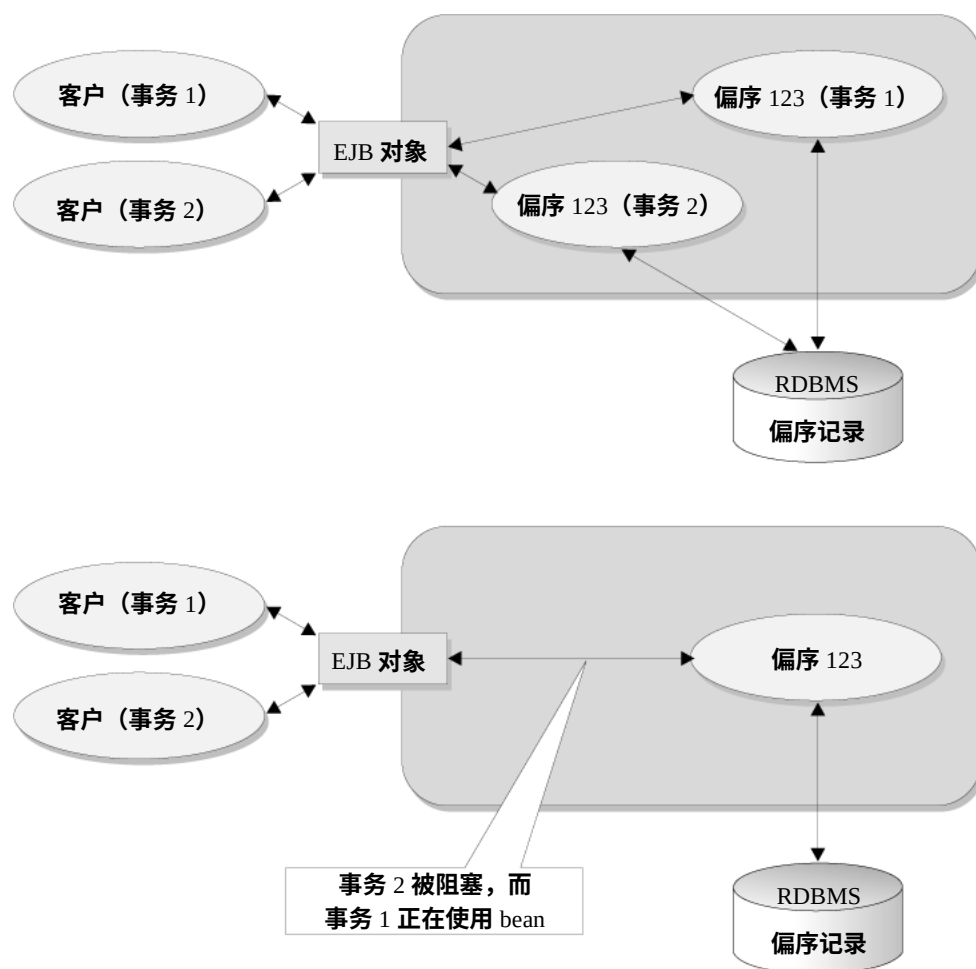


图 11-7 管理对实体 bean 并发访问的两种方法

- 可以达到更好的性能。

- 数据库同步通常比容器同步更加健壮并且具有更多的功能。
- 在复杂应用系统中，数据库常常被系统中的许多部分所使用，例如报表模块、数据抽取过程等等。因此，建立在数据库层次上的事务与锁定管理功能比建立在 EJB 基础上更加安全。
- 容器不需要加入复杂的锁定与串行化代码。

11.3 EJB 事务模型

对分布式事务的支持，可能是 EJB 最吸引人的地方。在应用程序中创建对分布式事务的支持是一项非常困难的任务，但使用 EJB 能够减轻这一问题的难度。通过将管理事务这一职责从 bean 开发人员转移到容器与服务器提供者，bean 能够简化应用程序开发并且确保涉及事务这一困难的问题由专家来解决。

EJB 事务支持可以与 CORBA OTS (Object Transaction Service, 对象事务服务) 对应，并且能够支持平面事务模型。OTS 为平面事务模型定义了必要的支持，并且为嵌套事务定义了可选的支持（嵌套允许一个事务在另一个事务的上下文中启动）。每个事务管理它自己的修改集合。当一个嵌套事务回滚时，那么在这个嵌套事务中的所有修改将被废弃。如果这个嵌套事务被确认，那么在此事务中所作的修改将变成包围该嵌套事务的那个事务修改集合的一部分，并且由这个外围事务进行确认或回滚。在一个嵌套模型中，顶级事务通常与数据库事务相对应。在这种情况下，被嵌套的事务是受限的，表现在它们不具有持续性，这是因为，当顶级事务确认发生时，被嵌套事务的信息并没有保存在数据库中。EJB 事务支持仍然是在平面事务模型上提供；这一决定基于当今的数据库系统与事务监视器支持平面事务模型这一事实。

EJB 使用 JTS (Java Transaction Service, Java 事务服务)，更特别地说就是使用 JTA (Java Transaction API, Java 事务 API)。JTS 将 OTS API 映射到 Java 编程语言；也就是说，它使用 IDL-to-Java 语言映射将 OTS IDL 定义映射到 Java。此外，JTS 还是一个为 X/Open XA 接口标准提供的 Java 映射。JTA 提供了一个方便的 Java 接口集合，用于事务。只要遵守这些标准，就能够确保 JTS 仍然作为 Java 的事务标准。

JTS javax.transaction 包定义了应用程序级限界 API，从而允许应用程序开发人员定义原子性事务任务。EJB 将依赖于这个接口，或者更特殊地说，就是依赖于 javax.transaction.

UserTransaction 接口 (EJB 容器提供者需要实现这个接口)。bean 开发人员与客户使用这个包中的 API 进行限界。

事务限界可以由客户、bean 或容器来进行。由容器执行的限界是最简单的，这是因为客户与 bean 对事务不可知。显然，容器必须使用 bean 所定义的策略信息，这是因为容器是一个通用的实体，它必须使自己的行为适应 bean。对于事务，bean 使用事务属性来定义限界策略。容器利用对 bean 功能的访问必须经过容器这一事实来管理事务限界。关于应该做什么

的实际决定是基于 bean 所定义的事务属性的。需要说明的是，这个策略可能是为整个 bean 全局地定义的，也可能是为某个 bean 方法定义的；因此，对于不同的方法就可能有不同的事务行为发生。例如，bean 可能设置自己的事务属性，使得每个方法调用必须在它自己的事务中进行。在这种情况下，容器将为每个发生在 bean 上的调用启动一个新的事务，并且如果在进行原始调用处理的过程中 bean 进行了附加的调用，容器还必须管理事务上下文的传递。图 11-8 显示了这种情况。

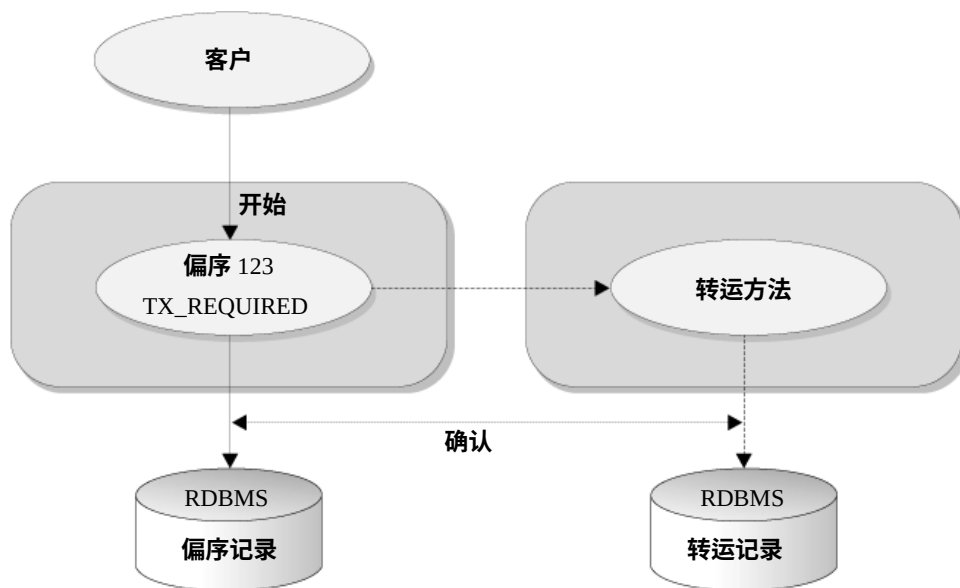


图 11-8 容器限界的例子

可能的事务属性值如下：

- TX_NOT_SUPPORTED：通过使用这个值，bean 通知容器自己不支持事务。这样，通过容器的调用将不包含在事务中。容器通过在对 bean 调用的过程中挂起任何活跃的事务来保证这一点。
- TX_BEAN_MANAGED：当 bean 希望使用显式的限界，并且希望通知容器不要代表 bean 管理事务时，bean 将使用这个属性。
- TX_REQUIRED：当 bean 希望确保任何处理都在一个事务中执行时，就使用这个属性。如果该调用是在一个由客户所启动的事务中作出，那么将在事务中执行这个 bean 调用。否则，容器就负责在将控制权交给 bean 之前启动一个事务。容器也负责在将控制返回给调用者之前对事务进行确认。
- TX_SUPPORTS：当 bean 希望自己在能够与客户相同的作用域内被调用时，使用这个属性。如果客户在调用这个 bean 时已经有一个事务打开，那么 bean 将在调用者

的事务中执行，否则，它将在没有任何活跃事务的情况下执行。

- **TX_REQUIRES_NEW**：使用这个属性值的 bean，将告知容器需要在一个新事务的范围中调用本 bean。无论客户的事务状态是什么，容器总是打开一个新事务，这个新事物处理在 bean 的整个操作过程中活跃。当 bean 完成时，容器将确认事务并恢复被挂起的客户事务。
- **TX_MANDATORY**：这一属性类似于 TX_SUPPORTS，表现在 bean 也是在客户的事务中被激活。但是，如果一个没有已打开事务的客户试图进行调用，将发出一个 TransactionRequired 异常。如果 bean 需要在客户事务的上下文中被调用，而且希望在没有事务的情况下不能被调用，那么就应该使用这个属性。

为了使容器能够代表 bean 管理事务，bean 需要定义的另一个属性就是用来控制读锁定选项的事务隔离等级。这些值对应于 java.sql.Connection 接口中可用的选项。事务隔离等级可能的值包括：

- **TRANSACTION_READ_UNCOMMITTED**：可能读取被标记为脏（脏读）的数据；可以读出未确认的数据。
- **TRANSACTION_READ_COMMITTED**：不允许进行脏读；只有确认的数据才能被读取。
- **TRANSACTION_REPEATABLE_READ**：不允许进行脏读和非可重复读。
- **TRANSACTION_SERIALIZABLE**：事务被串行化；只有当一个事务处理完成之后，另一个事务才能工作。
- **TRANSACTION_NONE**：不使用事务。

虽然容器限界对客户与 bean 开发人员来说是最简单的模型，但有时事务属性的可能值的集合对于事务限界的精确控制并不足够。在 EJB 事务模型中，允许使用 bean 管理的限界以及客户管理的限界。对于 bean 管理的限界，将使用 TX_BEAN_MANAGED 事务属性来定义这个 bean，它告诉容器应该代表 bean 管理事务。然后，bean 就可以使用 javax.transaction.UserTransaction 接口。没有定义 TX_BEAN_MANAGED 作为其事务属性的 bean 不允许对底层的事务管理器进行直接调用。或者，客户可以使用 javax.transaction.UserTransaction 接口开始、确认与回滚事务。

由于容器与服务器能够管理事务，因此可以发出与事务相关的异常（TransactionRolledbackException，TransactionRequiredException，InvalidTransactionException）。任何异常处理机制通常都是为这样的情况而设置的：错误检测者（detector；发出异常的对象）与错误处理程序（handler；捕获该异常并试图采取纠正措施的对象）可能是不同的对象。因此，在一个容器管理的事务上下文中发出异常，并不意味着这个事务必须被标记为只能回滚。还有可能出现这种情况，就是捕获该异常的客户能够修复问题并继续进行后续的业务方法调用。客户能够假设当前事务被标记为回滚的惟一情况就是，当发出 TransactionRolled-

backException 异常时。或者，异常处理程序可以使用 `getRollbackOnly()` 进行显式测试，从而验证当前活跃事务的状态。管理自己的事务进程的 bean（就是那些使用 `TX_BEAN_MANAGED` 的 bean）应该在发出异常之前调用 `setRollbackOnly()`，具体取决于所发现的错误是否应该允许到目前为止所做的修改被确认。

11.4 EJB 分布模型

EJB 的应用目标是大规模的分布式对象应用程序。因此，分布就成为 EJB 定义的基础。在 EJB 中进行应用程序开发涉及使用 bean 所实现的业务功能构建企业 bean 与客户的问题。容器、服务器以及其他 EJB 底层结构对象都由厂商提供，用来简化分布式应用程序的构建。因此，EJB 分布模型必须足够简单从而应用程序编程者能够使用它，但又必须具有足够强大的功能从而用于构建任务关键的大规模应用程序。

为此，EJB 分布模型使用一项久经验证的基于 proxy 与对象 stub 的技术。EJB 分布模型并不是一个新模型；它使用基于 CORBA 分布模型的标准 Java RMI。对应用双方来说，分布都是完全透明的。可以在本地或远程访问企业 bean 的功能，并且客户能够不关心企业 bean 的实际物理位置而访问企业 bean。无论是客户还是企业 bean 都不能意识到对方所在的位置。所创建的 Java RMI stub 涉及 EJB 对象与本地接口对象。由于 EJB 对象是用于访问功能的接口对象，因此，使用它作为创建 Java RMI stub 的对象是最合适的。此外，本地接口将获得自己的 RMI stub；这对于通过创建过程或查找过程初始化与企业 bean 的连接是必要的。使用 Java RMI 作为 EJB 中支持分布的 stub 与接口并不会影响底层的连线级协议。实际上，在网络协议方面，更适合 EJB 的解决方案是 CORBA IIOP。通过使用 IIOP，EJB 能够确保它的分布模型与非 Java 对象以及遗留系统具有可互操作性，并且为它们提供开放的接口。将 EJB 基于 CORBA IIOP 是非常自然的，毕竟，它们具有相同的分布模型，EJB 事务基于 CORBA OTS 模型，并且两者使用相同的对象模型。通过将 CORBA 命名服务（COS（Common Object Service，公共对象服务）的一部分）映射到 JNDI API，甚至能够解决 EJB 中的命名服务问题。最后，下一节将要讨论的 EJB 安全性也可以映射到 CORBA 安全性——使用 `CORBA::Principal` 来映射用户 ID，使用 IIOP over the SSL（Secure Socket Layer，安全套接字层）进行用户认证与加密，使用 SECIOP（Secure IIOP）来映射一个特殊的安全性机制（例如 GSSKerbero）。

11.5 EJB 安全性

正如在事务中一样，EJB 在安全性底层结构方面的主要目标，是提供一个接口集合，从而允许将实现安全性的职责从企业 bean 开发人员转移到容器与服务器实现者。这并不意味着企

业 bean 不能控制安全性；而只是意味着，提供这种机制使得典型的企业 bean 能够明确定义应该达到的安全性属性，而关于这些安全性的实现细节则由服务器与容器来处理。与事务一样，这是通过在部署时使用一个安全性属性集合来实现的。

EJB 部署描述符中的安全性属性定义了一个关于授权用户的列表，每个授权用户的表项都具有 `java.security.Principal` 类型——Java 中用于对标识与授权进行访问控制的标准接口。

与安全性相关的更多部署属性包括 `RunAsMode` 属性与 `RunAsIdentity` 属性。`RunAsMode` 属性定义了当 bean 为客户请求服务并作出更多调用时将要被传递的安全性标识。这个属性可能值包括：

- `CLIENT_IDENTITY`：当 bean 作出的调用应该假设已经存在进行原始调用的客户的安全性信息时，将使用这个值；为第二个调用提供服务的 bean 将认为原始的客户作出了这个调用（在安全性方面）。如图 11-9 所示。
- `SYSTEM_IDENTITY`：这个调用将使用特权系统帐号的标识。
- `SPECIFIED_IDENTITY`：将使用一个特定的用户帐号作出该调用。

只有当 `RunAsMode` 属性具有 `SPECIFIED_IDENTITY` 值时，`RunAsIdentity` 属性才是有意义的；在这种情况下，`RunAsIdentity` 属性定义了在进行第二个调用时将使用的实际用户标识。

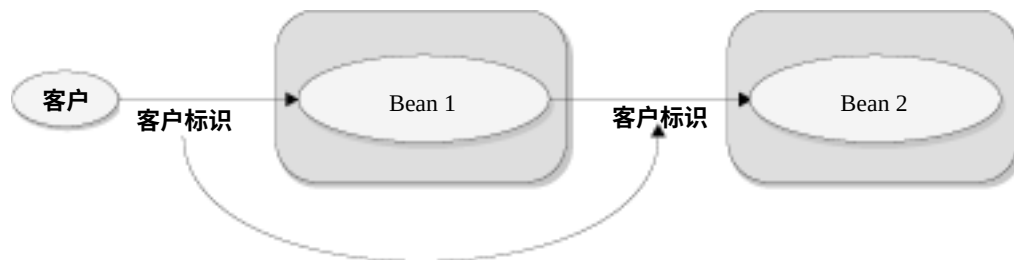


图 11-9 说明 `CLIENT_IDENTITY` 属性的使用

11.6 小 结

EJB 组件模型是服务器方应用程序使用的主要编程模型。它将结构与一致性带入到企业应用程序的最复杂方面——服务器方处理，包括事务管理与持续性管理。本章讨论了实体 bean 与会话 bean 的生命周期，并且指出在何时应该使用相应的 bean。本章也从 EJB 容器、EJB 服务器、客户、开发人员的角度描述了 EJB 体系结构。本书第 29~34 章对如何在 WebSphere 下开发与使用 EJB 进行了更详细的讨论。

第 12 章 XML

XML (Extensible Markup Language, 可扩展标记语言) 是 WebSphere 所采用技术集合中的一个新成员。从 3.x 版本开始, 所有 WebSphere Application Server 都包括许多 XML 库并且维护有大量用 XML 编写的元数据。4.0 版本更是将它提升到一个新的境界; WebSphere Application Server 4.0 支持 XML 解析与生成并且使用 XML 作为更高级技术 (例如 Web 服务) 的基础, 从而在整个产品中都使用 XML。

本章提供了在使用 XML 时必须熟悉的各种库的信息。后面的章节将讨论使用 XML 构建与部署应用程序, 并且讨论 DOM, SAX (Simple API for XML), 以及 XSLT (XSL Transformation) 的用法。虽然在 WebSphere 中使用 XML 意味着需要使用用于创建与处理 XML 文档的 Java 库, 但本章所讨论的话题对所有 XML 处理都有效, 而与所使用的编程语言无关。

12.1 新 “silver bullet”

XML 是 IT 领域的新 “silver bullet” —— 也就是, 它是许多困难的 IT 问题的解决方案。现在, XML 已经被用作集成与自动化的基石, 集成与自动化是对复杂信息系统进行有效部署的两个重要手段。此外, XML 可能是电子商务应用程序的最重要底层组件, 因为它允许由不同厂商所创建的系统相互交流与共享数据。它也是 EAI (Enterprise Application Integration, 企业应用程序集成) 系统 (例如 Vitria, TIBCO, MQSeries, Web Methods, Cross Worlds 等) 的基础。

现在, 世界各地的厂商与开发人员都转向使用 XML —— 这样做是有充分理由的。XML 是一个由 W3C (World Wide Web Consortium) 开发的标准, 用来改善通过 Web 进行数据共享以及数据格式化处理的工作; 一般来说, 它通过保存域的语义来支持标记数据的概念。XML 是 SGML (Standard Generalized Markup Language, 标准通用标记语言) 的应用程序 (子集), 但其使用更简单。它比 HTML 功能更强大; 实际上, HTML 是 XML 的子集。因此, 可以说 XML 是 SGML 的强大表达能力与 HTML 的简单性两者之间有效的结合体, 并且它已经成为 Web 上的混合语言。此外, 它还在其他方面存在优势, 例如 business-to-business 应用程序、EDI (Electronic Data Interchange, 电子数据交换) 以及文档管理方面。

XML 已经成为 W3C 和整个业界研究与开发的热点。作为一个标准, 它是 W3C 开发的许多其他标准的基础 —— 例如 XLL (Extensible Linking Language, 可扩展链接语言), DOM, SAX 与 XSLT。在 W3C 的网站 <http://www.w3.org> 以及其他网站 (例如为 OASIS 提供信息的网站以及包括 IBM, Microsoft, Sun 以及 Oracle 在内的大多数厂商的网站) 上都可以找到这些

规范。

XML 已经是用于电子商务应用程序的重要技术，并且我们相信，这一趋势将会有效地加速 XML 与电子商务的融合。目前，有非常大量的应用程序使用 XML，并且为这些应用程序提供的资源不再局限于 W3C 与商业厂商。例如，FpML (Financial Products Markup Language, 财务产品标记语言) 就是一个利用 XML 编写的应用程序，它描述了财务产品如衍生、换兑。FpML 由 JPMorgan 与 PricewaterhouseCoopers 开发，并且在 Mozilla 许可下发布；程序清单 12-1 给出了一个 FpML 文档的例子。

注意：一个明显的问题是，“JPMorgan 为什么发布自己所投资的源代码，万一这个源代码被竞争者用来作为技术优势怎么办”？答案是，这并非因为 JPMorgan 的人都“很好”。实际上，这是因为公司（例如 JPMorgan）已经理解到，它们存在于一个新的 Web 和电子商务范型下，开放标准就能够支持以前不可设想的服务与系统。而 XML 则是这种潮流的领导者。

程序清单 12-1 FpML 示例

```
<?xml version="1.0" standalone="no"?>
<!-- Copyright (c) 1999 by J.P.Morgan and PricewaterhouseCoopers.
PricewaterhouseCoopers refers to the individual member firms of the world
wide PricewaterhouseCoopers organization. All rights reserved.
-->
<!-- version 1.0b2 : August 6, 1999 -->
<fpml:FpML xmlns:fpml="urn:fpml-FpML"
  xmlns:t="urn:fpml-type"
  xmlns:d="urn:fpml-date"
  xmlns:m="urn:fpml-money"
  xmlns:r="urn:fpml-rate">

  <fpml:Trade>
    <fpml:tradeIDs>
      <tid:TradeIDs xmlns:tid="urn:fpml-TradeID">
        <tid:TradeID>
          <tid:partyReference>ABC Trust</tid:partyReference>
          <tid:transactionReferenceNumber>237732</tid:transactionReferenceNumber>
        </tid:TradeID>
        <tid:TradeID>
          <tid:partyReference>XYZ Group</tid:partyReference>
          <tid:transactionReferenceNumber>1230</tid:transactionReferenceNumber>
        </tid:TradeID>
        <tid:TradeID>
          <tid:partyReference>CDF Inc.</tid:partyReference>
          <tid:transactionReferenceNumber>237732</tid:transactionReferenceNumber>
        </tid:TradeID>
      </tid:TradeIDs>
    </fpml:tradeIDs>
  </fpml:Trade>
</fpml:FpML>
```

```
</tid:TradeID>
<tid:TradeID>
  <tid:partyReference>TUV Ltd.</tid:partyReference>
  <tid:transactionReferenceNumber>1230</tid:transactionReferenceNumber>
</tid:TradeID>
<tid:TradeID>
  <tid:partyReference>MNO LLC</tid:partyReference>
  <tid:transactionReferenceNumber>237732</tid:transactionReferenceNumber>
</tid:TradeID>
<tid:TradeID>
  <tid:partyReference>HIJ.com</tid:partyReference>
  <tid:transactionReferenceNumber>1230</tid:transactionReferenceNumber>
</tid:TradeID>
</tid:TradeIDs>
</fpml:tradeIDs>
</fpml:Trade>

</fpml:FpML>
```

12.2 可扩展标记语言

XML 于 1996 年由 Sun 公司的 Jon Bosak 领导的工作组所创建。XML 被设计作为 SGML 的子集，在设计过程中考虑以下目标：

- XML 应该足够简单，以便在 Internet 上使用。
- XML 应该支持各种类型的应用程序。
- XML 应该使得应用程序开发人员构建处理 XML 文档的软件这一工作更简单。
- XML 文档必须以自然的 XML 方式被人类可读。
- 创建 XML 文档的工作应当是简单的。

可以说，XML 的设计目标就是作为 SGML 的子集——因此保持了 SGML 所具有的强大功能（但 XML 同时必须易于使用）。

XML 是一种允许文档被结构化的格式。因此，XML 标准主要描述了 XML 文档。XML 文档有一个由实体组成的物理结构。每个 XML 文档正好有一个实体，称为根实体（root entity）；所有其他实体都必须属于这个根实体。利用一个开始标记与一个结束标记来定义 XML 实体，这与使用 HTML 定义许多元素的方法相同。当一个实体嵌入另一个实体时，被嵌入的实体的开始标记与结束标记都必须位于外围实体的开始标记与结束标记的内部。例如，如果希望描述拥有一个电话号码的 contact，那么可能需要按如下方式嵌入该实体：

```
<contact>
```



```
<name>Jane Doe</name>
<phone>201-861-7533</phone>
</contact>
```

可以使用开始标记与结束标记指定一个在这两个标记之间没有任何内容的空标记（也就是不包含任何数据的元素），或者使用一个简单的自关闭（self-closing）开始标记来指定空标记。例如，一个 `contact` 可能有两个可用的电话号码；如果这些标记为空，那么就可以使用下列 XML 实体：

```
<contact>
  <name>Jane Doe</name>
  <phone>201-861-7533</phone>
  <altPhone1></altPhone1>
  <altPhone2/>
</contact>
```

XML 文档可以包括许多类型的实体。大多数实体都反映出嵌入在 XML 文档中的信息以及正在被该文档描述的域语义。许多其他结构（包括元素）构成一个典型的 XML 文档。例如，可以使用 HTML 开发人员熟悉的语法在 XML 文档中嵌入注释，如下所示：

```
<!-- note that the <contact> here is not really a tag;
it is part of the comment and will be ignored by
applications, parsers, and other XML tools (except
to view it as a comment).
-->
```

12.2.1 XML 示例：API 集成

在继续后面的讨论之前，首先看几个较为复杂的例子，第一个例子就是 API 集成（毕竟，没有任何东西能够击败现实世界的代码）。图 12-1 显示了一条消息，消息体中包含有 XML；它被发送给 ViryaNet Integration Server 作为 HTTP 请求。XML 代表为激活 ViryaNet 系统中定义的 API 所需的数据。在这个例子中，一个应用程序将向 ViryaNet 服务中心发送一条消息，以创建一个调用。发送该消息的应用程序可以是 call-center 应用程序、trouble-ticketing 应用程序或 provisioning 系统。通过允许消息以 XML 的形式表示，就使得 ViryaNet Integration Server 能够对任何支持 XML 的平台可用。

如图 12-1 所示，消息有 2 个主要部分——消息头部与消息体。这些概念不能与 HTTP 头部和 HTTP 体相混淆；所有 XML 代码都作为 HTTP 体的一部分存在。头部包括关于消息本身的信息，这些信息对于 ViryaNet Integration Server 的传输级是有用的（例如，这个消息是什么，以及何时发送该消息）。嵌入在 `<input>` 标记中的第二部分是要作为消息的一部分发送的参数。

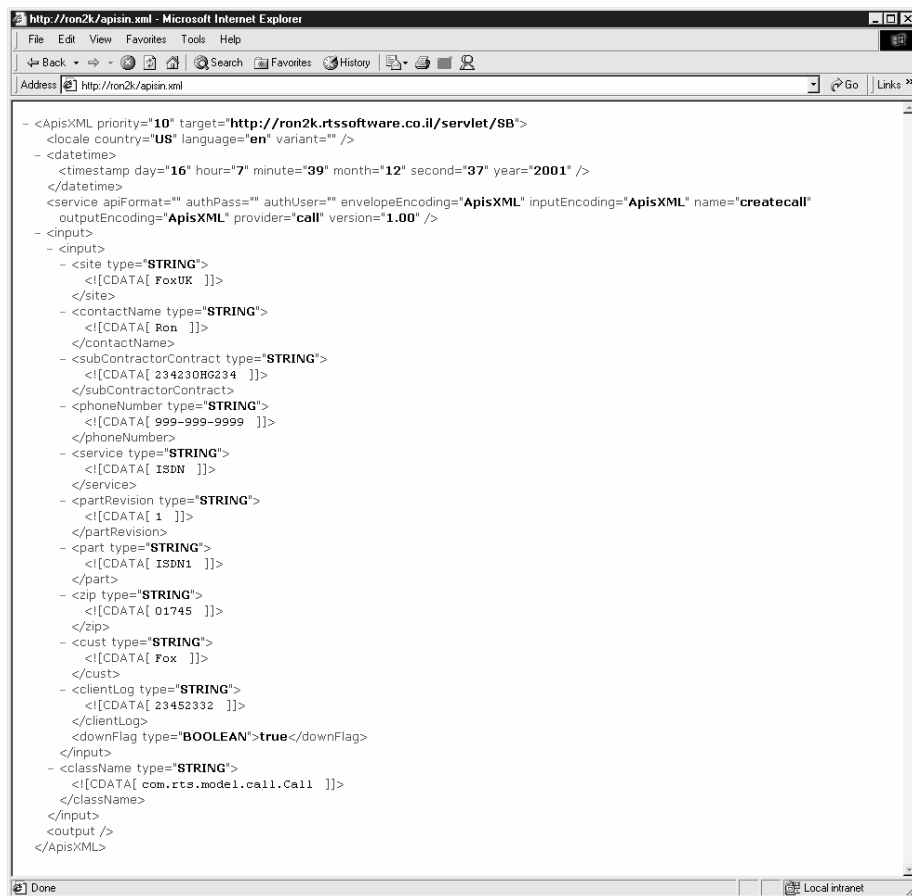


图 12-1 XML 封装 API 调用的例子

在图 12-1 中，还需要说明的一点是这个文件中的所有数据都封装在一个 CDATA 元素中。这个元素表明，其中的内容应该作为原始数据被读取，而不能进行任何解释。这也就意味着，可以在其中包括任何形式的控制代码与不可见字符，这些数据可能对各种语法分析程序具有特殊的含义。如果希望内容被作为数据读取，并且不控制发送者如何对这些数据进行打包，那么最安全的做法就是将这些内容封装在 CDATA 元素中。

12.2.2 XML 示例：数据文件

第二个例子（见图 12-2）是一个消息文件。ViryaNet 服务中心的所有消息都存储在一个 XML 文件中，当服务器需要向 Web 浏览器上的用户显示一条消息时，服务器将使用这个文件。这里需要说明的是，这个文件被命名为 PortalMessages_en，表示它是一个 XML 文件，用于为来自英语地区的消息请求服务。从 Web 浏览器或者通过 ViryaNet Integration Server 得到

的请求都将包括一个区域值（注意，请求的头部（见图 12-1）有一个 locale 参数）。当某个环节出现问题并且需要显示一条错误消息时，就将基于区域从这个文件中取出一条错误消息。图 12-2 显示了为 en locale 提供的 XML 文件。

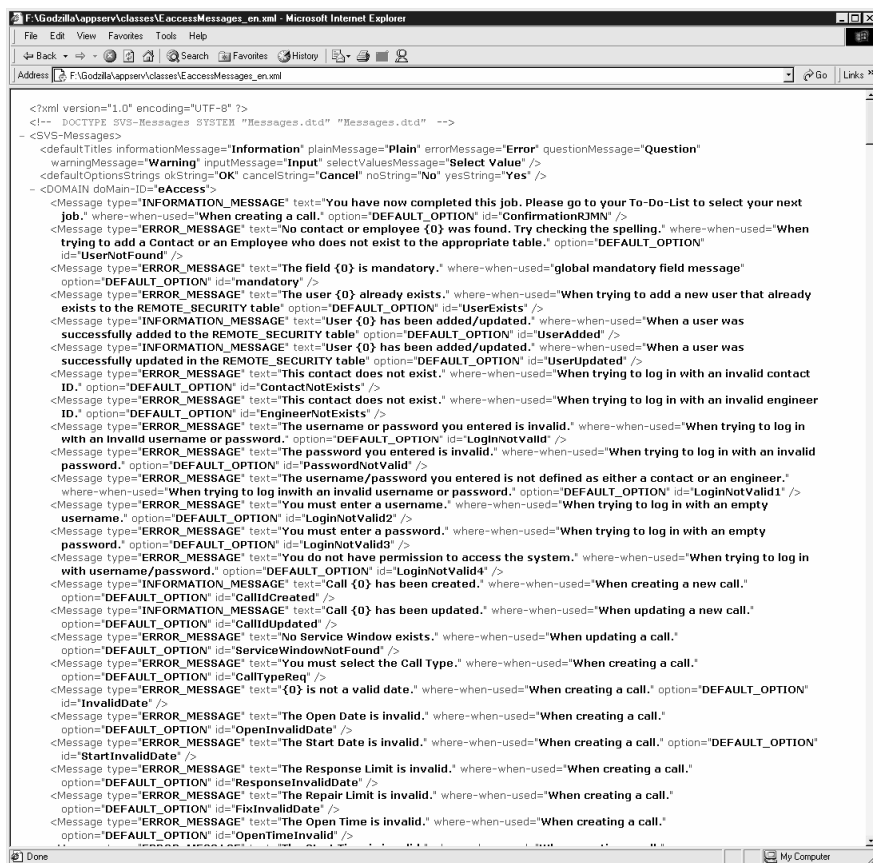


图 12-2 关于错误消息的 XML 文件——en locale

例如，如果希望支持日语地区的错误消息，那么就需要创建名为 PortalMessages_jp 的另一个文件，在这个文件中，错误消息以日语形式给出。然后，如果有一条请求消息通过 Web 到达，并且其浏览器所选择的区域为日语 (jp_JP)，那么_jp 将从适当的文件中引用这个消息。

12.2.3 文档类型定义

XML 的最基本概念就是 XML 中的标记集合是可扩展的 (extensible；这也是 X 的来由)，而不是固定的。这就意味着，使用 XML 编写的不同应用程序可以定义自己的标记集合，并


```

        (selectValuesDocumentation?,(
        parametersDocumentation?)*,
        (buttonTitles?)*)>
<!ATTLIST Message
    id                ID                #REQUIRED
    text              CDATA            #REQUIRED
    where-when-used   CDATA            #IMPLIED
    title             CDATA            #IMPLIED
    type              (ERROR_MESSAGE|INFORMATION_MESSAGE|
                      WARNING_MESSAGE|QUESTION_MESSAGE|
                      PLAIN_MESSAGE|INPUT_MESSAGE|
                      SELECT_VALUES_MESSAGE)
                      #REQUIRED
    option            (CANCEL_OPTION|NO_OPTION|YES_OPTION|
                      YES_NO_OPTION|YES_NO_CANCEL_OPTION|
                      OK_OPTION|OK_CANCEL_OPTION|DEFAULT_OPTION)
                      'DEFAULT_OPTION'
    confirm-key       CDATA            #IMPLIED>
<!ELEMENT buttonTitles EMPTY>
<!ATTLIST buttonTitles
    id                (okString|cancelString|yesString|noString)
                      #REQUIRED
    text              CDATA            #REQUIRED>
<!ELEMENT parametersDocumentation EMPTY>
<!ATTLIST parametersDocumentation
    id                CDATA            #REQUIRED
    text              CDATA            #REQUIRED>
<!ELEMENT selectValuesDocumentation EMPTY>
<!ATTLIST selectValuesDocumentation
    text              CDATA            #REQUIRED>

```

12.2.4 XML schema

计算机业界定义 XML 的新趋势是使用 XML schema。XML schema 是一个描述 XML 代码的 XML 文件。XML schema 是一个元数据元素，它使用 XML 的灵活性来提供引用这个 XML schema 的 XML 的完全描述，由于 XML schema 是一个高级概念，并且由于本书并不使用 schema，因此这里不对 schema 进行深入的讨论。一般来说，可以将 XML schema 看作类似于 DTD 但比 DTD 更健壮、功能更强大的描述元素。目前，W3C 正在对 XML schema 进行标准化，并且许多主要的厂商（例如 Microsoft，IBM）都支持 XML schema。

元素都有用来描述这些元素的特性集合，也就是说，这些特性集合用来描述元素的属性。可以添加特性，并且添加它们的值作为开始标记的一部分，从而将某个特性与元素相关联。这些特性有时被误认为是嵌入在实体中的附加标记。实际上，很容易对这两者进行区分，

其方法是 将 XML 代码看作类似于对象模型 的描述。在这种情况下，它们之间的区别就等价于对象的属性与所引用的其他对象的属性之间的区别。可以定义特性作为 DTD 的一部分，并且可以将它们定义为强制性的、具有默认值的。例如，如果希望在 `contact` 实体中添加一个用来表示“VIP level”（VIP 等级）的特性，那么就应该使用下列 XML 代码：

```
<contact vipLevel="6">
  <name>Jane Doe</name>
  <phone>201-861-7533</phone>
  <altPhone1></altPhone1>
  <altPhone2/>
</contact>
```

12.3 文档对象模型

DOM (Document Object Model, 文档对象模型) 是 HTML 文档与 XML 文档的 API，它为获取对这个文档所表示的对象的访问提供了编程范型。DOM 提供了用于构建文档、在文档中导航文档与实体、查看与编辑属性以及任何希望对文档进行工作所需的接口。DOM 标准是由 W3C 定义与支持的。通过将其提升为一个标准，W3C 确保有一个标准的 API 用来管理 XML 文档与 HTML 文档，从而允许开发人员构建可移植的代码，而这进一步促进了 XML 的应用。

DOM 将每个 XML (与 HTML) 文档进行解析，并将它们分析成为一个关于对象的层次化结构。它定义了如何从 XML 文档中创建森林 (树的集合)。由于树结构是由 DOM 定义的对象，因此 DOM 提供了用来管理这些树的 API，这样也就能够管理文档自己。由于 DOM 主要由这个用来处理对象的 API 组成，因此它属于 CORBA IDL。IDL 是一种可以映射到多种编程语言的规范语言，因此可以使用许多编程语言处理 DOM API 以创建库的集合，同时又维护有一个单独的、一致性的 API。

12.3.1 XML 的 DOM

DOM 最初来自 HTML，创建 DOM 的目的是提供一个可用来为浏览器开发客户方应用程序 (例如 JavaScript 应用程序) 的对象模型。在此之后，DOM 迅速地发展，但它仍然包含大量关于 HTML 对象的定义，以及浏览器中公用对象的定义。因此，DOM 规范有两个主要部分——与 XML 匹配的核接口集合的定义，以及为 HTML 对象特别提供的接口集合的定义。下面给出了 DOM 为 XML 给出的主要接口：

- Document 整个 XML 文档或 HTML 文档。这个接口有诸如 `docType`, `implementation`, `documentElement` 这样的属性，并且有诸如 `createElement`, `createDocumentFragment`,

createTextNode, createComment, createCDATASection, createAttribute 以及 createEntityReference 这样的方法。

- Node：整个 DOM 的主要对象。读者应该记得，DOM 在本质上是一个树，因此，其中每个元素就是一个节点。节点具有属性，例如 `nodeType`, `parentNode`, `childNodes`, `firstChild`, `lastChild`, `previousSibling`, `nextSibling` 以及 `ownerDocument`；并且，节点具有方法，例如 `insertBefore`, `replaceChild`, `removeChild`, `appendChild` 以及 `hasChildNodes`。
- Element：一个节点，代表文档中被标记的对象。在对树结构进行遍历的过程中，元素是最常见节点类型。元素接口从节点接口继承，并且添加了新的方法，例如 `getAttribute`, `setAttribute`, `removeAttribute`, `getAttributeNode`, `setAttributeNode`, `removeAttributeNode` 以及 `getElementsByTagName`。
- Attr：元素的特性（也就是在构建 XML 时插入在开始标记中的内容）。DOM 中的特性也是节点，因此，在 DOM 中创建的树包括元素与特性。这样，具有特性的元素以及嵌入的元素将被映射到 DOM 树的一个节点上，并且这个节点具有用于作为嵌入元素的子节点；但它与那些代表特性的节点存在更多的关系。
- Comment：文档中的注释。

12.3.2 HTML 的 DOM

有一个特定于 HTML 对象的接口集合，包括：

- HTMLDocument：添加特性，例如 `title`, `URI`, `body`, `image`, `applet`, `link`, `form`, `anchor` 以及 `cookie`；添加方法，例如 `open`, `close`, `getElementById` 以及 `getElementsByName`。
- HTMLElement：从 Element（第一个类型）继承；添加属性，例如 `id`, `title`, `lang`, `dir` 以及 `className`。
- HTMLHeadElement：从 HTMLElement 继承；代表 HTML 文档的头部。
- HTMLLinkElement：添加特性，例如 `href`, `rel`, `rev`, `target` 以及 `type`。
- HTMLTitleElement：从 HTMLElement 继承；代表 HTML 文档的 `title` 元素。
- HTMLStyleElement：代表一个样式引用，例如用来定义层叠样式表或样式伪指令的引用。
- HTMLBodyElement：添加特性，例如 `Link`, `background`, `bgColor`, `link`, `text` 以及 `vLink`。
- HTMLFormElement：添加特性，例如 `action`, `method`, `target`；添加方法，例如 `submit` 与 `reset`。
- HTMLSelectElement：添加特性，例如 `selectedIndex`, `value`, `form`, `options`, `disabled`,

name；添加方法，例如 add, remove, blur, focus。

- **HTMLInputElement**：添加特性，例如 defaultValue, defaultChecked, form, accept, align, checked, disabled, value；添加方法，例如 blur, focus, select, click。
- **HTMLTextElement**：添加特性，例如 form, disabled, rows, tabIndex, type, value；添加方法，例如 blur, focus, select。
- **HTMLButtonElement**：添加特性，例如 form, name, disabled, value。
- **HTMLLabelElement**：添加特性，例如 form 与 htmlForm。
- **HTMLUListElement**：用来创建无序列表（UL 标记）。
- **HTMLOListElement**：用来创建有序列表（OL 标记）。
- **HTMLLIElement**：用来创建列表中的元素（LI 标记）。
- **HTMLHeadingElement**：用来创建头部元素（H1 到 H6 标记）。
- **HTMLBRElement**：用来创建断行元素（BR 标记）。
- **HTMLAnchorElement**：用来插入锚元素（A 标记）；添加特性，例如 href 与 target。
- **HTMLImageElement**：用来插入图像元素（IMG 标记）；添加特性，例如 name, align, alt, height, isMap, src 与 width。
- **HTMLAppletElement**：用来插入 applet（APPLET 标记）；添加特性，例如 archive, code, codebase, height, object 与 width。
- **HTMLScriptElement**：用来插入一个脚本（可以是文件引用，也可以是内嵌代码）；代表 HTML 4.0 中的 SCRIPT 元素；添加特性，例如 event, src 与 type。
- **HTMLTableElement**：添加特性，例如 tHead, tFoot, rows, align, bgColor, border 与 frame；添加方法，例如 createTHead, deleteTHead, createTFoot, deleteTFoot, createCaption, insertRow 与 deleteRow。

除了这些元素列表之外，HTML 的 DOM 还提供许多其他元素，例如，HTMLTableCaptionElement, HTMLTableColElement, HTMLTableSectionElement 与 HTMLTableRowElement。

下面的程序清单演示了 DOM 的使用。这个例子显示了 DOM 解析程序的两种使用形式。第一种形式是读取一个 XML 字符串（这个字符串可能来自一个文件，可能从 Internet 上接收，也可能从一个数据库中提取），然后从其中提取节点、元素、特性。第二种形式就是读取一个使用 DOM API 构建的 DOM 树，并且输出一个包括元素与特性的 XML 字符串。

```
// Assume xml is a variable holding an XML string.
// Create a reader with the string so that you can send
// it to the parser.
// create an input source - used by the parser API.
    java.io.StringReader xmlStringReader =
new java.io.StringReader (xml);
    org.xml.sax.InputSource xmlInputSource =
```



```
new org.xml.sax.InputSource(xmlStringReader);

// Create a parser. Use the Simple API for XML, but
// ask for a DOM parser to be created.
    org.xml.sax.Parser parser =
org.xml.sax.helpers.ParserFactory.makeParser(
"com.ibm.xml.parsers.TXDOMParser");

// Give the parser the input source.
    parser.parse(xmlInputSource);

    // Cast the parser object so you can directly access
    // the document object. This is not the conventional
    // way of accessing things, because usually a higher-
    // level API is used.
    com.ibm.xml.parsers.TXDOMParser actualParser =
(com.ibm.xml.parsers.TXDOMParser) parser;

// Get the document object. This is the most important
// object when you work with DOM, because from it you can
// access all nodes, elements, and attributes that are
// anywhere in the DOM tree (the XML).
    TXDocument doc =
(TXDocument)(actualParser.getDocument());

    // Now assume that path is a variable holding a string[]
    // with the path inside the XML file you are looking for.
    // The following code searches and returns the node
    // within the DOM that is in that path.
    int len = path.length;
    if (len <1)
        return null;

    Node node = doc;

    for ( int i = 0; i < len; i++)
    {
        if (node == null)
            break;
        NodeList childNodesList = node.getChildNodes();
        node = null;
        if (childNodesList == null)
            break;
        {for ( int j = 0; j < childNodesList.getLength(); j++)
```

```

        if (childNodesList.item(j).getNodeName().
            trim().equals(path[i].trim()))
        {
            node = childNodesList.item(j);
            break;
        }
    }

    // The variable node now holds the desired element.

    //-----

    // This part shows the flip. Given a document object
    // that is created by composing elements, attributes, and
    // nodes, you can write out the XML as a file or a string.

    // You start with a hash table with name-value pairs.
    // You then create a set of elements that encapsulate these values.
    Iterator keys = hashtable.keySet().iterator();
    Element child;
    while (keys.hasNext())
    {
        String nextElementName = (String)keys.next();
        Object nextElement = hashtable.get(nextElementName);
        Element child =
            new TXDocument().createElement(nextElementName);
        doc.appendChild(child);
        fillDomData(child, nextElement);
    }

    StringWriter stWriter = new StringWriter();
    doc.printWithFormat(stWriter);
    xml = stWriter.toString();

```

下面的程序清单显示了 `Domable` 类（实际上是一个关于这个类的简单版本），它是一个在 ViryaNet XML 底层结构中使用的实用程序类，并且这个程序清单演示了如何使用 DOM 执行 XML 处理。

```
package com.rts.apis.common;
```

```
import com.sun.java.util.collections.Map;
import com.sun.java.util.collections.HashMap;
import com.sun.java.util.collections.Iterator;
import java.io.FileInputStream;
import java.net.URL;
import java.util.Calendar;
import java.util.Date;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.Vector;
import java.io.*;
import org.w3c.dom.*;
import com.ibm.xml.parser.*;
import com.ibm.xml.parsers.*;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.Statement;
import java.sql.Types;
import java.text.SimpleDateFormat;
import java.util.Enumeraion;
import java.net.MalformedURLException;

public class Domable
{
    // Many definitions omitted - not important for these examples.

    // The DOM document object is most central to this class.
    TXDocument selfDoc = null;

    /**
     * Parse the string into a DOM document.
     * @param xml java.lang.String
     */
    public Domable(String xml) throws IOException , org.xml.sax.SAXException{
        try{
            xml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + xml;
            java.io.StringReader xmlStringReader = new java.io.StringReader
(xml);
```

```

        org.xml.sax.InputSource xmlInputSource = new org.xml.sax. Input-
Source (xmlStringReader);
        org.xml.sax.Parser parser = org.xml.sax.helpers.ParserFactory.
makeParser ("com.ibm.xml.parsers.TXDOMParser");
        parser.parse(xmlInputSource);
        com.ibm.xml.parsers.TXDOMParser actualParser = (com.ibm.xml.
parsers.TXDOMParser) parser;
        setSelfDoc((TXDocument)actualParser.getDocument());
    } catch (Exception ex) {
        Log.debug("MessageFormat", ex);
    }
}

/**
 * Create an element to be added into the document with the appropriate tag.
 * @return org.w3c.dom.Element
 * @param aNode org.w3c.dom.Node
 * @param aTag java.lang.String
 */
public static Element createElement(TXDocument aDoc, String aTag) {
    Element childFirst = aDoc.createElement(aTag);
    aDoc.appendChild(childFirst);
    return childFirst;
}

/**
 * Create an element to be added as a child node.
 * @return org.w3c.dom.Element
 * @param aNode org.w3c.dom.Node
 * @param aTag java.lang.String
 */
public static Element createElement(Node aNode, String aTag) {
    Element childFirst = new TXDocument().createElement(aTag);
    aNode.appendChild(childFirst);
    return childFirst;
}

/**
 * This method is used to get the value of an attribute of a specific

```

```
* node that is pointed to by a path.
* @param path the path to the node.
* @param attributeName the name of the attribute.
* @return the value of the required attribute.
*/
public String getAttributeValue(String[] path, String attributeName)
    throws Exception
{
    Node node = getNode(path);

    if (node == null)
        throw new Exception("Not found path.");

    String result = getAttributeValue(node, attributeName);
    if (result == null)
        throw new Exception("Not found attribute " + attributeName);
    return result;
}

/**
 * This method is used to get the value of an attribute of a specific
 * node.
 * @param node the node.
 * @param attributeName the name of the attribute.
 * @return the value of the required attribute.
 */
public static String getAttributeValue(Node node, String attributeName)
{
    NamedNodeMap nodeMap = node.getAttributes();
    if (nodeMap == null)
        return null;
    Node attributeNode;

    for (int i = 0; i < nodeMap.getLength(); i++){
        attributeNode = nodeMap.item(i);
        if
(attributeNode.getNodeName().equalsIgnoreCase(attributeName))
            return attributeNode.getNodeValue();
    }

    return null;
}
```

```
/**
 * Parse the string into a DOM document.
 * @param xml java.lang.String
 */
public static TXDocument getDocument(String xml) throws IOException,
org.xml.sax.SAXException
{
    TXDocument retDoc = null;
    try
    {
        xml = "<?xml version=\"1.0\" encoding=\"UTF-8\"?>" + xml;
        java.io.StringReader xmlStringReader = new
        java.io.StringReader(xml);
        org.xml.sax.InputSource xmlInputSource = new
        org.xml.sax.InputSource(xmlStringReader);
        org.xml.sax.Parser parser = org.xml.sax.helpers.ParserFactory.
            makeParser("com.ibm.xml.parsers.TXDOMParser");
        parser.parse(xmlInputSource);
        com.ibm.xml.parsers.TXDOMParser actualParser =
        (com.ibm.xml.parsers.TXDOMParser) parser;
        retDoc = ((TXDocument) actualParser.getDocument());
    }
    catch (Exception ex)
    {
        Log.debug("MessageFormat", ex);
    }
    return retDoc;
}
```

```
/**
 * Used to get a node according to the path from the root to the node.
 * @param path the path to the node.
 * @return returns the node or null if not found.
 */
public Node getNode(String[] path)
{
    int len = path.length;
    if (len < 1)
        return null;
}
```

```
Node node = this.getSelfDoc();

if (node == null)
    return null;

for ( int i = 0; i < len; i++)
{
    if (node == null)
        break;
    NodeList childNodesList = node.getChildNodes();
    node = null;
    if (childNodesList == null)
        break;
    for ( int j = 0; j < childNodesList.getLength(); j++){ if
(childNodesList.item(j).getNodeName().trim().equals(path[i].trim()))
        {
            node = childNodesList.item(j);
            break;
        }
    }
}

return node;
}
```

12.4 XML 的简单 API

DOM 提供了处理任何 XML 文档所必需的功能。编程者可以使用 DOM API 来构建 XML 文档的树结构，然后遍历这些树，以处理元素、提取信息或执行任何其他操作。虽然按照这种方式使用 DOM 非常容易，但它们在有些时候是非常低效的。例如，如果 XML 文档表示公司所有的 contact（联系），那么这个 XML 代码将是一个非常大的文档。如果要对这个文档进行解析，创建树，然后遍历这些树，那么可能需要等待很长的时间。如果确实需要对 XML 文档中的所有元素进行遍历，那么只有采用这种方法；DOM 提供的解决方案总是尽可能地高效。但是，如果需要进行的工作是对文档中元素的子集（并且可能是一个非常小的子集）进行处理，那么应该考虑另一种更好的可选方法。

SAX（Simple API for XML）是一种用于处理 XML 文档的基于事件的 API。基于事件就意味着 SAX 为应用程序开发人员提供了一种方法，使得在开始进行应用程序级处理之前，不必对所有的元素进行预构建，而仅仅对那些开发人员感兴趣的元素进行预构建。这样做的

好处是，那些不必要的结构将不被构建；相反，当任何开发人员感兴趣的事件发生时，在应用程序代码中产生一个 callback。需要说明的是，当需要处理元素的子集时，SAX 不仅比 DOM 更加高效，而且如果需要处理大型的 XML 文档（例如有成百上千甚至成千上万个元素），它还是一个非常有用的工具。如果使用 DOM，即使需要处理所有的元素，系统都可能会出现内存问题，因为构建如此多的对象需要大量的内存。相反，SAX 则不需要构建任何东西；其处理过程是实时进行的，而不是采用“构建后处理”范型进行的。

SAX 的使用是简单的，涉及两个阶段。首先，需要使用 SAX API 定义自己感兴趣的元素。例如，可能仅仅对 contact 的名称感兴趣，或者，仅仅对应该在给定的月份进行更新的 contact 感兴趣。定义了自己的兴趣范围之后，下一步就需要激活解析程序，用来解析 XML 文档。在解析程序对整个 XML 文档进行处理的过程中，它将遇到元素标记并产生事件（例如，START ELEMENT CONTRACT，END ELEMENT CONTRACT，START ELEMENT CONTACT）。当中有些事件是不需要的，因此不会发生任何事情（也就是说，解析程序将继续工作）。当感兴趣的事件发生时，这个事件将导致应用程序 callback 被调用，然后，对这个特殊元素的处理过程就开始了。

显然，只有当使用支持 SAX 的解析程序时，使用 SAX 才是有意义的。这个解析程序就是需要遍历树并产生事件的组件；因此，它需要知道如何完成这一工作。SAX 定义了一个称为 org.xml.sax 的包，提供了许多接口，例如 Parser，AttributeList 与 Locator。为了支持 SAX 处理，支持 SAX 的 XML 工具就需要实现这些接口。

SAX 在 org.xml.sax 包中定义了另一个接口集合，应用程序开发人员必须实现这些接口。包括：

- DocumentHandler：在这个接口中定义了解析程序用来通知应用程序某事件已经发生的方法。从应用程序开发人员的角度来看，这是最重要的接口。
- ErrorHandler：在文档被处理的过程中，如果发生了某些错误，将使用这个接口来调用应用程序级代码。

12.5 XSL 转换

XSLT（XSL Transformation，XSL 转换）是 XSL（Extensible Stylesheet Language，可扩展样式表语言）的子集，定义了用来将 XML 文档转换成其他 XML 文档的语言（实际上，XSLT 是 XSL 的一部分；XSL 的另一部分是格式处理语言）。XSLT 在 XML schema 中很重要。XML 文档基于文档的创建者所定义的标记集合产生。常常有这种情况，就是两个功能组之间希望以 XML 文档的形式交换数据，但在所使用的标记集合上不完全相同。在这种情况下，通过 XSLT 在这两个 XML 标记集合之间执行一个映射常常是可行的。

XSLT 是 XML 的一个应用程序，它导致一个标记集合允许定义转换规则。这些规则定义了如何对一个从 XML 文档构建的源树进行处理，使其变成结果树，然后，利用这个结果树产

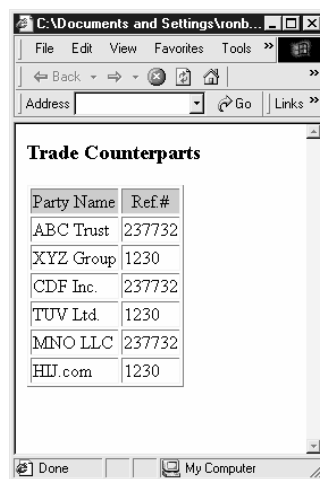
生 XML 结果文档。每个转换规则都有一个模式与模板。模式（pattern）用来与源树进行匹配，用以确定规则（rule）所定义的结构，规则是转换的基础。在模式被匹配之后，就对源树应用模板（template），用以产生结果树的一个元素。显然，上述动作将循环进行，从而使得在应用模板时，其他转换规则也将被引用。

模板可能非常复杂，XSLT 对模板具有较强的表达性功能。可以独立地创建树元素，也可以从源树创建树元素；除了使用 XSLT 定义的大量元素类型之外，也可以使用 XSLT 支持的处理指令。

每条规则都有一个模式，它定义了规则何时起作用，从而能够被应用到源树以创建结果树。模式还可以有基于 XPath 规范的表达式。这些表达式能够选择适当的节点进行处理。例如，一个表达式可以对节点（如节点的类型）、节点的特性、节点的祖先、节点的后代以及它们的组合施加条件判断。

一旦一个表达式被求值，那么将开始进行模式的匹配以及模板的处理。当表达式被匹配之后，总是伴随着标识了一个当前节点；这个节点就是由模式所匹配的那个节点。对模板的所有处理都基于这个节点进行。例如，如果包括一条用于遍历所有子节点的伪指令，那么这条伪指令将被应用到表达式所匹配的节点上。

XSLT 是一种复杂的语言，需要花费一定的时间去学习理解。我们并不对这种语言的所有功能进行描述；实际上，对 XSLT 的描述可能占据一本书的篇幅。有兴趣的读者可以访问 <http://www.w3.org/TR/WD-xslt>，以获取关于 XSLT 的详细信息。我们以两个简单的例子来结束本章的讨论。第一个例子将把一个描述交易伙伴的 XML 文档转换成一个 HTML 表（读者可能已经认识这个文档，它是基于 FpML 的，在程序清单 12-1 中进行过说明）。XSLT 样式表在第一个程序清单中给出，而得到的结果 HTML 源代码在第二个程序清单中给出。最后，在图 12-3 中显示了这个结果 HTML 页面。



Party Name	Ref.#
ABC Trust	237732
XYZ Group	1230
CDF Inc.	237732
TUV Ltd.	1230
MNO LLC	237732
HIJ.com	1230

图 12-3 应用 XSLT 样式表

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

<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/XSL/Transform/1.0"
  xmlns="http://www.w3.org/TR/REC-html40"
  xmlns:fpml="urn:fpml-FpML"
  xmlns:tid="urn:fpml-TradeID"
  result-ns=""
  indent-result="yes">

  <xsl:template match="fpml:Trade">
    <HTML><BODY>
      <H3>Trade Counterparts</H3>
      <TABLE BORDER="1">
        <TR align="center" bgcolor="CCCCCC">
          <TD>Party Name</TD>
          <TD>Ref.#</TD>
        </TR>
        <xsl:apply-templates/>
      </TABLE>
    </BODY></HTML>
  </xsl:template>

  <xsl:template match="fpml:TradeIDs">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="tid:TradeID">
    <TR>
      <xsl:apply-templates/>
    </TR>
  </xsl:template>

  <xsl:template match="tid:transactionReferenceNumber">
    <TD>
      <xsl:apply-templates/>
    </TD>
  </xsl:template>

  <xsl:template match="tid:partyReference">
    <TD>
      <xsl:apply-templates/>
    </TD>
  </xsl:template>
```

```
</xsl:template>
```

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

下面给出了结果 HTML 页面：

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

```
<HTML>
```

```
<BODY>
```

```
<H3>Trade Counterparts</H3>
```

```
<TABLE BORDER="1">
```

```
<TR align="center" bgcolor="CCCCCC">
```

```
<TD>Party Name</TD>
```

```
<TD>Ref.#</TD>
```

```
</TR>
```

```
<TR>
```

```
<TD>ABC Trust</TD>
```

```
<TD>237732</TD>
```

```
</TR>
```

```
<TR>
```

```
<TD>XYZ Group</TD>
```

```
<TD>1230</TD>
```

```
</TR>
```

```
<TR>
```

```
<TD>CDF Inc.</TD>
```

```
<TD>237732</TD>
```

```
</TR>
```

```
<TR>
```

```
<TD>TUV Ltd.</TD>
```

```
<TD>1230</TD>
```

```
</TR>
```

```
<TR>
```

```
<TD>MNO LLC</TD>
```

```
<TD>237732</TD>
```

```
</TR>
```

```
<TR>
```

```
<TD>HIJ.com</TD>
```

```
<TD>1230</TD>
```

```
</TR>
```

```
</TABLE>
</BODY>
</HTML>
```

第二个例子显示了关于使用 XML 的更详细应用程序集成。正如人们所说的那样，并且从实际交易结果也可以清楚地看到，XML 正成为应用程序集成的“silver bullet”。当今的大多数应用程序都有一个基于 XML 的接口。但不幸的是，每个应用程序都使用 XML 的不同“方言”。在第 38 章将介绍 Web 服务，它是用来标准化使用 XML 进行集成这一工作的新技术集合。

由于我们现在还处在 Web 服务及其标准（SOAP，WSDL，UDDI）的开始阶段，因此当前给出的用来消除 XML 方言间差异的解决方案常常就是 XSLT 转换程序。当一个应用程序创建了一个 XML 文档时，就使用 XSLT 转换程序将它转换成另一个应用程序可理解的 XML 文档。下面的程序清单给出了一个这样的例子。其中，第一个程序清单显示了用 SXP (Service Exchange Protocol，服务交换协议)“方言”编写的 XML 文档。这个文档被第二个程序清单中显示的 XSLT 表进行转换，成为最后一个程序清单所显示的 XML 文档。

下面给出了由应用程序 A 所产生的 SXP 文档：

```
<AssignmentAdd Destination="ViryanetTest">
  <Assignment>
    <Key>290819</Key>
    <Start>2001-08-07 06:30</Start>
    <Finish>2001-08-07 09:30</Finish>
  </Assignment>
  <Task>
    <Key>33892352</Key>
    <CallID>te435</CallID>
    <Number>1</Number>
    <Status>
      <Name>Open</Name>
    </Status>
  </Task>
  <Engineers>
    <Engineer>
      <ID>JP3</ID>
      <Name>jp tEST TERMINATE</Name>
    </Engineer>
  </Engineers>
</AssignmentAdd>
```

下面给出了 XSLT 样式表：

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

```

<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" language="VBS-
cript">
  <xsl:template match="/">
    <SXPCreateGeneralEvent Revision="1.00">
      <input>
        <xsl:choose>
          <xsl:when
            test="AssignmentAdd"><eventDefinition>ALDMST</eventDefinition></xsl:when>
          <xsl:when
            test="AssignmentUpdate"><eventDefinition>ALDMST</eventDefinition></x-
            sl:when>
          <xsl:when test="AssignmentDelete"><eventDefinition>REJNA</eventDefinition>
          </xsl:when>

          <xsl:if test="*/Task/CallID"><call><xsl:value-of select="*/Task/CallID"/>
            </ call></xsl:if>
          <xsl:if test="*/Task/Number"><action><xsl:value-of select=
            "*/Task/Number"/ ></action></xsl:if>
          <xsl:if test="*/Engineers/Engineer/ID"><employee><xsl:value-of select=
            "*/ Engineers/Engineer/ID"/></employee></xsl:if>
          <xsl:if test="*/Assignment/Start"><dateFrom><xsl:value-of select=
            "*/ Assignment/Start"/></dateFrom></xsl:if>
          <xsl:if test="*/Assignment/Finish"><dateTo><xsl:value-of select=
            "*/ Assignment/Finish"/></dateTo></xsl:if>

            <allocator>
              CLICK
            </allocator>
            <userId>
              CLICK
            </userId>

          </input>
        </SXPCreateGeneralEvent>
      </xsl:template>
    </xsl:stylesheet>

```

最后，给出了应用程序 B 所使用的 XML 文档：

```

<SXPCreateGeneralEvent Revision="1.00">
  <input>
    <eventDefinition>
      ALDMST
    </eventDefinition>
    <call>

```

```
        te435
    </call>
    <action>
        1
    </action>
    <employee>
        JP3
    </employee>
    <dateFrom>
        2001-08-07 06:30
    </dateFrom>
    <dateTo>
        2001-08-07 09:30
    </dateTo>
    <allocator>
        CLICK
    </allocator>
    <userId>
        CLICK
    </userId>
</input>
</SXPCreateGeneralEvent>
```

12.6 小 结

本章描述了 XML 的使用，并且描述了其他相关的包与技术，例如 DOM，SAX 与 XSLT。现在，读者应该较好地理解了什么是 XML 文档、如何使用 XML 文档、如何创建 XML 文档、如何解析 XML 文档，并且理解了能够应用于这些文档的处理工作。当我们在第 35~38 章讨论在 WebSphere 中如何使用 XML 来构建与部署应用程序时，就需要掌握本章所讨论的这些“如何”。

第 13 章 Java 安全性

本章的目标是从较高的层次介绍 Java 安全性以及电子商务应用程序中常见的安全性方案。安全性问题是一个比较广泛的话题，它是 Web 应用程序存在性的核心问题。本章将向读者介绍一些在 WebSphere 上开发与部署应用程序所必需熟知的概念。如果本章讨论的话题激发了读者的学习兴趣，那么还有许多更深入地介绍该话题的书籍可供阅读。

Java 的设计目标是可移植的解释性、分布式、可靠、健壮与安全 (secure)、高性能与多线程、简单、动态、并且支持运行时环境的面向对象的编程语言。本章的第一部分将讨论 Java 编程语言所具有的安全性特点。

Java 从一开始就把安全作为重要的设计目标。作为一种用于构建健壮分布式系统并且可用于在 Web 上进行应用程序部署的语言，在设计该语言时必须时刻牢记安全性问题——不顾及安全将会使得 Java 在实际的 Web 系统上无法应用。Java 从一开始就支持安全性，并且在其中集成了大量与安全性相关的服务，这些服务由 JDK (Java Developer's Kit) 或支持包的最新发行所提供，因此，Java 完全能够将分布式 Web 对象变成现实。

Java 安全性由许多部分组成。首先，作为一种编程语言，Java 是这样定义的：这种方式消除了使用语言自身的特点入侵到用 Java 语言编写的系统中的可能性。Java 消除了指针的概念，并且对编程者隐藏了内存的布局与结构。因此，编程者就不能使用该语言自己侵入运行时代码中。Java 是类型安全的，并且产生的异常将被标记，从而消除了对 type 系统进行攻击的可能性。Java 编程语言的这些属性，不仅在消除 bug 以及使编程模型更易于使用等方面显得非常重要，而且能够使 Java 在防黑客方面更安全。

虽然该语言存在的这些特点能够提高安全性，但它们并不能自己确保这些安全性。实际上，可以说，没有任何功能集合、服务以及能力能够保证 Web 环境的完全安全性——毕竟，黑客们机敏过人，具有创新性并且动机明确。但是，通过本章的学习可以了解到，Java 提供了特别有用并且非常完善的安全性功能，加上它所具有的约定集合以及人工的有效参与，Java 理所当然的能够使得创建安全的 Web 应用程序成为现实。

13.1 Java 安全性模型

安全性是 Java 编程环境的内在部分，这也是为什么在 Java JDK 的第一个发行版本中就集成了安全性特点的原因。除了 Java 编译器所提供的类型安全支持，以及逻辑结构与物理实现相分离这些安全性策略之外，基本的 Java 模型还通过如下 4 个主要支持确保安全性：

- sandbox 模型
- class loader
- 字节代码验证程序
- 安全性管理器

13.1.1 Java sandbox

sandbox 模型可能是 Java 安全性模型中最著名的部分，当然，也是最重要的部分之一。Java 采取了一个不同于传统操作环境的方法。例如，在大多数操作系统中，应用程序能够对系统中的所有资源进行访问，包括文件、打印机以及网络。在 Java 中，从 Web 上下载的非可信代码在一个称为 sandbox 的受限环境中运行。通过使非可信代码在受限环境中运行，并且严格限制该代码可访问的资源，就能够使得任何恶意行为仅仅在 sandbox 中造成损害。需要说明的是，之所以这个模型能够支持该功能，是因为 Java 使用了虚拟机（virtual-machine）方法（也就是，一种运行时体系结构，在这种体系结构下，应用程序根据由另一个程序（Java 虚拟机，用来将指令转换到本地机器代码指令）所读取的指令进行编译）。

Java sandbox 可以称作是“非可信代码的运动场”。它定义了在其中运行非可信代码的环境，并且能够确保代码（例如所下载的 applet）不能对这个受限环境之外的地方造成损害以致破坏客户机器。它为 applet 或其他下载的代码提供了所需的所有功能，而不必允许它们访问系统的外部组件。这个 sandbox 实际上就使得 JVM（从技术上说，实际上是 JVM 的一部分）成为 applet 所见的全部世界。

class loader、字节代码验证程序以及安全性管理器进一步加强了 sandbox 模型。sandbox 模型定义了 JVM 状态机中必须始终存在的常量与断言。在将控制交给非可信代码时可能出现的任何状态都必须在 Java sandbox 模型的约束之下。

Java sandbox 模型定义了下列关于 JVM 状态机的状态迁移，作为 Java 安全性的基础：

1. 当一个 class loader 的构造函数被调用时，将调用安全性管理器来验证这个线程是否能够创建一个 class loader。这样就确保非可信代码不能创建更多的 class loader（即使有字节代码被修改）。
2. 当 JVM 调用 class loaders 方法加载一个类时，就进入下一状态。首先，将使用 class loader 的 findSystemClass 方法找到所有的系统类，从而确保非可信代码不能伪装成系统代码。
3. 在代码被加载之后，将调用验证程序来检查代码的有效性。验证程序确保代码遵守语法与结构规则。静态成员被初始化为默认值。
4. 然后，class loader 将加载的类映射到它所管理的名称空间上。

sandbox（假设在 sandbox 的实现中不存在任何 bug 与安全性漏洞）的概念是非常好的。它允许用户下载非可信 applet 并在客户机器上毫无忧虑地运行它们。这与那些必须依赖于所下载代码的创建者及其数字签名的其他模型完全不同（说实在的，sandbox 模型有时是过于

严格的。在这些情况下，数字签名就允许某些代码比其他代码“更加可信”。

当然，sandbox 模型也有缺点。在 JDK 1.0.x 中定义的 sandbox 是非常严格的。例如，从非可信资源下载 applet 不能访问系统资源、系统属性等等，而这种非常苛刻的限制可能使得这些程序没有用处。因此，开发人员与用户就不得不面临这样的选择：或者避开 sandbox 模型（这样就使得系统不安全），或者剥夺应用程序所需的必要功能。

幸运的是，在 JDK 1.1 中给出了关于这个问题的一个解决方案，并且在 J2SE (Java 2 Platform Standard Edition, Java 2 平台标准版) 中对此解决方案进行了扩展。其思想就是设置所提供的控制级别。特别地，为此定义了关于可信级别的概念。可以在客户方基于数字签名定义这些可信级别，这样就允许被签名的、下载的代码能够在各种虚拟 sandbox 中运行，每种虚拟 sandbox 都有不同的可用资源与许可集合。

13.1.2 class loader

Java 通过一个正在运行的程序按照需要动态加载类文件。当在 Web 上工作时，这些类文件可以从最初的 Web 站点下载，并且被动态添加到工作环境中（这在某些方面类似于 DLL (Dynamic-linked Library, 动态链接库) 的动态加载）。这个加载过程是由一个特定的对象 (class loader) 执行的。这个对象形成了安全性防护的第一道屏障，表现在它能够控制与限制应该加载什么、从什么地方加载以及何时加载。class loader 应用命名规则并且确保不适当的类不被加载（例如，class loader 不允许加载可能在其中嵌入安全性漏洞的其他 class loader）。

class loader 负责将名称映射到 Java 字节代码、负责从类文件加载适当的字节代码，并且负责调用验证程序（若必要）。这对于维护安全性常量从而使得 class loader 成为 Java 安全性防护的第一道屏障是非常关键的。例如，class loader 不会试图通过网络加载任何来自 java.* 包中的类，这样就保证 JVM 不会被欺骗而使用 Java 内核类的不安全表示。

Java 2 除了加载程序员定义的所有 class loader 外，还额外启动 3 个 class loader。所提供的默认 class loader 分别是引导程序 class loader、扩展 class loader 以及应用程序 class loader。应用程序 class loader 是惟一使用 CLASSPATH 环境变量的 class loader。CLASSPATH 环境变量可以被命令行选项-classpath java 所覆盖。

JDK 将 class loader 定义为一个抽象类（也就是，它不能被实例化）；这个抽象类的子类可以定义用来动态加载类的实例。默认的行为是以平台相关的方式从 CLASSPATH 加载，并且需要网络访问允许字节代码被下载与实例化。

13.1.3 验证程序

验证程序 (verifier) 在代码被实际执行之前使用。class loader 所加载的非可信代码必须经过一个验证阶段，这个阶段用来“证明”代表 Java 代码的所加载字节代码遵守关于该语言的所有规则。由于字节代码是 JVM 的“机器语言”，并且由于加载该代码的系统并不知道谁创建了这些字节代码（也不知道它们是如何被创建的），因此，安全性模型必须确保这些字

节代码没有隐藏可能对 Java 类型实现造成危险的恶意程序。验证程序能够确保访问的严格性、确保不执行非法的类型转换、没有进行伪造的内存访问。

显然，验证程序在一个较低的层次上工作，并且确保字节代码遵守 Java 规则；但它并不能执行任何应用程序级分析或执行启发性的安全性检查。

如果能够确信 Java 字节代码总是由一个合法的 Java 编译器所产生，那就可以不需要验证程序。由合法 Java 编译器（例如作为 JDK（或 J2SE 发布）的一部分的编译器）创建的字节代码总是遵守 Java 语言规则的。虽然绝大多数 Java 字节代码都是使用这样的编译器创建的，但我们并不能完全依赖于这一事实。验证程序的目的，就是完全杜绝可能破坏 JVM 的、通过恶意方法创建的字节代码渗入到系统中。例如，可能手工地构造一些字节代码，并且一旦运行这些字节代码，将导致 JVM 的栈溢出，从而使 JVM 崩溃。验证程序就是这样的安全性组件，它负责阻止这种非法行为的发生。

13.1.4 安全性管理器

Java 安全性模型中的最后一个组件就是安全性管理器，它作为 sandbox 的看门狗，是 Java 安全性技术中最重要的组件。安全性管理器负责限制非可信代码的动作。简单地说，它就是用来保证非可信代码在运行时不会离开 sandbox 的组件。安全性管理器控制对资源的运行时访问，这些资源包括构成 Java 安全性框架的内部对象。例如，安全性管理器不允许非可信代码创建可能被用来巧妙避开所在层次上的安全性检查的 class loader。安全性管理器必须允许对物理资源（例如套接字、文件）的所有访问。

安全性管理器既是主动的，又是被动的。它对某些方法执行运行时有效性检查，同时也被其他与安全性相关的组件所使用（要求安全性管理器许可一个被请求的操作）。如果安全性管理器不想允许该操作，它将产生一个安全性异常，从而拒绝该操作的发生。

安全性管理器进行的测试项目包括：

- 对操作系统的访问
- 对套接字的访问
- 对文件系统的访问
- 对 Java 包的访问
- 对安全性模型的访问；例如，安全性管理器需要确信没有安装新的 class loader
- 对线程与线程组的访问
- 对剪贴板的访问
- 对打印机以及假脱机程序的访问

13.1.5 applet 安全性

施加在 applet 上的安全性限制比较严格，显然这是必要的，因为所下载的 applet 很可能包含

有恶意代码。因此，在默认情况下，sandbox 必须尽可能地严格。但是，在 Web 上支持企业应用程序常常暗示着所下载的代码要求比较宽松的运行时环境；JDK 1.1 与 JDK 1.2 引入了一个更为丰富的模型，它能够基于数字签名提供更多的可选择控制。

applet 不允许对本地（甚至任何）文件系统上的文件进行读取与写入。它们不能检查某个文件的存在性、不能修改文件名、不能创建目录、不能检查文件属性。这样就消除了所下载的 applet 删除硬盘上的内容、读取硬盘上的关键信息并将它们在 Internet 上广播等危险。在更高的 JDK 版本中，对这些限制进行了一定的放松，并且包括了更多高级的功能。例如，Sun 的 appletviewer（在 JDK 1.1 中）使用访问控制列表来在限制文件访问方面提供更精细的控制（使用属性文件中的 acl.read 属性）。

applet 能够创建的网络连接也受到了限制。一个 applet 仅仅能够与从其中下载该 applet 的主机建立网络连接。例如，只能打开到达提供该 applet 的 Web 服务器所在主机的套接字。

此外，还对 applet 能够访问的系统属性进行了限制。系统属性提供关于 Java 代码段运行环境的信息，包括操作系统名称、主机名以及其他环境属性。对系统属性的访问是通过 getProperty，getProperties 与 setProperties 方法来提供的，所有这些方法都在 System 类中实现。

13.2 J2SE 安全性

JDK 1.0 定义了早期 Java applet 使用的基本安全性模型，但如果考虑到真实应用程序在 Web 上部署这一问题，这个基本的安全性模型是远远不够的。如前所述，苛刻的限制将使得实现某些任务与应用程序变得困难。

JDK 1.1 将数字签名引入 Java 中。签名、密钥、证书、提供者以及算法都是用 Java 实现的，从而允许 applet 被它们的创建者所标记。然后，运行这个 Java 代码的客户就可以决定为某些已标记的代码放宽安全性限制。

JDK 1.1 也引入了 JAR（Java Archive）文件的概念。JAR 文件是能够将类文件、图像、区域特定数据以及其他的组件打包在一起的档案文件。这些 JAR 文件可用来形成一个关于某个下载网站运行应用程序所需全部内容的独立表示。在安全性方面，JDK 1.1 提供了利用数字签名标记 JAR 文件、并且使用这个签名能够惟一地标识该 JAR 文件以消除中间人攻击的支持。

此外，JDK 1.1 还引入了下列功能：

- 访问控制（access control）：ACL（Access Control List，访问控制列表）被用来作为管理 principal 及其许可的机制。
- 消息摘要（message digest）：提供了用来创建消息摘要作为数字指纹的支持。
- 密钥管理（key management）：为管理安全性数据库中的密钥与证书提供了原始的支持。

J2SE 还有更加丰富和完全的安全性体系结构。J2SE 安全性模型不仅具有更多的功能，而且具有更好的一致性以及更合理的组织方式，这个模型使用少量的原语概念来定义一个参考模型。在功能方面，J2SE 安全性模型对 JDK 1.1 模型进行了下列改进：

- 细粒度的访问控制：虽然应用程序可以定义并实现自己的 class loader 与安全性管理器，但这是一件非常困难的工作。Java 2 提供了更简单的方法，用来使应用程序修改与安全性相关的控制机制。
- 可配置的安全性策略：在 Java 2 中，安全性策略可以被配置。
- 可扩展的访问控制：可以在不必将 check 方法添加到 SecurityManager 类中的条件下，提供对访问控制模型的扩展。
- 检查的一致性：可信代码与非可信代码（applet 与应用程序）相互之间更加接近，并且能够使用相同的办法管理（通过使用相应的安全性域来实现不同的安全性属性）。这些域就是安全性原语的支持者，用来确定使用的是哪种类型的 sandbox（若存在）。

J2SE 安全性体系结构的核心概念就是保护范围（protection domain）。保护范围将代码进行混合，并且有一系列许可与其相关联（见图 13-1）。每个代码段都包含在一个保护范围中，并且假设从与这个范围相关的许可获取安全性特性。一个单独的线程能够完全被嵌入在一个单独的保护范围中，或者，它也可以使得不同的段属于不同的保护范围。很自然，将代码段从一个范围移动到另一个范围（特别是当第二个范围有更多许可时）通常涉及一些安全性约束，这些安全性约束可能会改变调用者可用的许可。

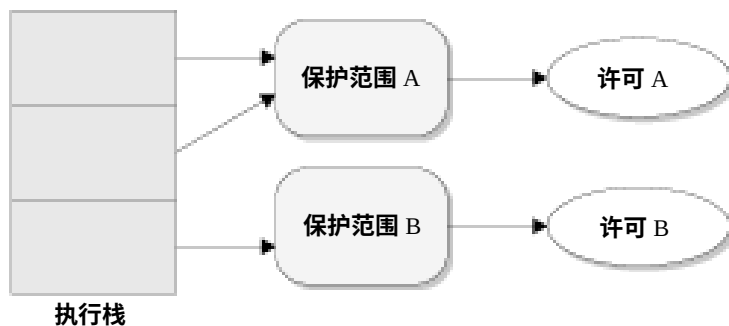


图 13-1 J2SE 保护范围

一般来说，保护范围可以是整个系统范围（包括有更多与其相关联的许可的系统代码），或者是各种应用程序范围。很自然，只有在受限的约束下，应用程序代码才可访问系统范围。

J2SE 安全性 API

Java 的安全性 API 大多数集中于 java.security 与 java.security.*包中。

许可

J2SE 安全性中最基本的抽象就是由称为 `java.security.Permission` 的基类所表示的许可。从这个类所继承的类的实例代表执行某种操作的许可。`java.security.Permission` 类的子类在各种包中使用，表示在各自相关上下文中的访问许可。例如，`java.net.SocketPermission` 代表通过套接字对网络的访问许可，而 `java.io.FilePermission` 则代表对某个文件的访问许可。

通常使用带有两个 `String` 参数的构造函数创建许可类的实例。其中，第一个参数定义了与许可相关的对象，另一个参数保存有关于这个对象的许可（以逗号分隔的操作的列表）。下面将分析几个与前面所提到的两个许可类相对应的示例。下面的行将创建一个许可对象，它允许与服务器 `www.viryanet.com` 的端口 80 进行连接：

```
Permission p1 = new SocketPermission("www.viryanet.com:80",
    "connect");
```

为了许可对文件 `/tmp/x` 的读写，可以使用下列代码行：

```
Permission p2 = new FilePermission("/tmp/x", "read,write");
```

实际上，许可自身并不能获取对系统资源与对象的访问。应该将许可设置在一个数据库中，并且这个数据库中的表项称为策略 (policy)。默认的数据库是文件 `java.policy`，它通常在 JDK 安装目录的 `jre\lib\security` 目录下（它的位置由一个称为 `java.security` 的属性文件定义，该文件位于与上相同的目录中）。

JDK 发布的默认文件如下：

```
// Standard extensions get all permissions by default
grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};

// default permissions granted to all domain
sgrant {
    // Allows any thread to stop itself using the java.lang.Thread.stop()
    // method that takes no argument.
    // Note that this permission is granted by default only to remain
    // backwards compatible.
    // It is strongly recommended that you either remove this permission
    // from this policy file or further restrict it to code sources
    // that you specify, because Thread.stop() is potentially unsafe.
    // See "http://java.sun.com/notes" for more information.
    permission java.lang.RuntimePermission "stopThread";

    // allows anyone to listen on un-privileged ports
    permission java.net.SocketPermission "localhost:1024-", "listen";
```

```
// "standard" properties that can be read by anyone
permission java.util.PropertyPermission "java.version","read";
permission java.util.PropertyPermission "java.vendor","read";
permission java.util.PropertyPermission "java.vendor.url","read";
permission java.util.PropertyPermission "java.class.version","read";
permission java.util.PropertyPermission "os.name", "read";
permission java.util.PropertyPermission "os.version", "read";
permission java.util.PropertyPermission "os.arch", "read";
permission java.util.PropertyPermission "file.separator","read";
permission java.util.PropertyPermission "path.separator","read";
permission java.util.PropertyPermission "line.separator","read";
permission java.util.PropertyPermission
"java.specification.version", "read";
    permission java.util.PropertyPermission
"java.specification.vendor", "read";
        permission java.util.PropertyPermission
"java.specification.name", "read";

    permission java.util.PropertyPermission
"java.vm.specification.version", "read";
        permission java.util.PropertyPermission
"java.vm.specification.vendor", "read";
            permission java.util.PropertyPermission
"java.vm.specification.name", "read";
                permission java.util.PropertyPermission "java.vm.version","read";
                permission java.util.PropertyPermission "java.vm.vendor","read";
                permission java.util.PropertyPermission "java.vm.name","read";
};
```

这个文件包含有一系列 `grant` 语句。`grant` 语句的一般形式包含有一个 `sign-by` 表项（在这个例子中没有）以及一个 `codebase` 表项（在第一个 `grant` 语句中显示）。共享相同 `codebase` 并且由相同实体所标记的类被认为具有相同的保护范围。例如，下列代码为 `http://viryanet.com` 中的由 `viryanet` 标记的所有代码授予对 `/tmp/viryanet` 下的文件的读/写许可。

```
grant signedBy "viryanet", codeBase "http://viryanet.com/" {
    permission java.io.FilePermission "/tmp/viryanet/*",
    "read,write";
};
```

当然，为了使代码被“标记”，还需要除了签名者（在这个例子中就是 `viryanet`）名称之外的其他东西。这些所谓的其他东西就是数字签名（digital signature）。

需要说明的是，这里所显示的默认许可文件仅仅提供了对 JDK 安装目录下 `lib\ext` 目录中代

码的无限制访问，而且非可信代码只能读取系统属性。

虽然能够手工地编辑策略，但标准 JDK 发布包括一个用于编辑策略的 Policy Tool。图 13-2 显示了该工具应用于默认的 java.policy 文件。

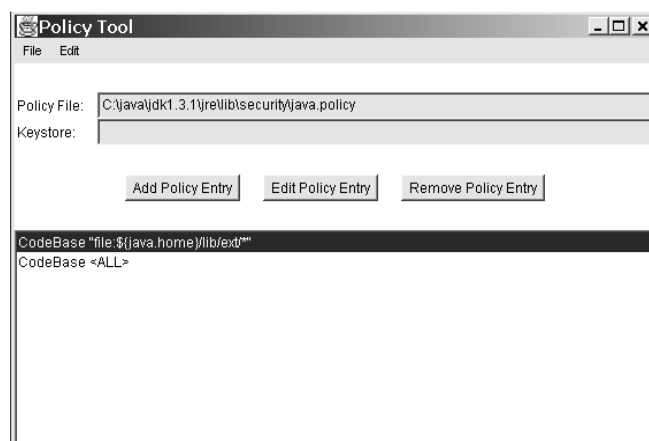


图 13-2 Java Policy Tool 的主窗口

用于数字签名的密钥是使用 keytool 实用程序管理的，这个实用程序是一个命令行实用程序，它也是 JDK 标准发布的一部分。keytool 实用程序用来创建密钥对（公共密钥或私有密钥）或存储来自其他功能组的证书。所有的密钥与证书都存储在一个称为 keystore 的数据库中。keystore 自己也是口令保护的，用来维护其内容的安全性。

访问控制列表

ACL API 提供了一个框架，通过该框架，用户能够说明是否授予给定用户对某些资源的访问权限。用户或用户组由 java.security.Principal 接口表示。

ACL API 在 java.security.acl 包中定义，这个包定义了下列接口：

- Acl：对某个资源的访问许可列表。
- Permission：授予某个用户对某种特定资源的许可。需要说明的是，这里的 java.security.acl.Permission 不同于前面讨论的 java.security.Permission。
- Group：封装一个 principal 列表。
- Owner：代表一个 ACL 列表（Acl 接口从这个接口继承）的所有者，可以是一个或多个 principal。
- AclEntry：描述某个 Principal 的访问许可。这个表项由极性（允许或拒绝）、Principal（这个特定的表项与谁相关）、Permission（用来代表一个动作）而组成。

在给定 Acl 对象之后，用户就可以检查某些 principal 是否被授予或拒绝某些操作。

更多的安全性 API：JAAS，JCE，JSSE

本节讨论某些附加的与安全性相关的 API，它们最初是作为可选 Java 包提供的；在 J2SE 1.4 中，它们变成了标准组件。下列组件为 Java 提供了更多安全性：

- JAAS (Java Authentication and Authorization Service, Java 认证与授权服务)：提供了一个用于用户认证的框架。
- JSSE (Java Secure Socket Extension, Java 安全套接字扩展)：提供了 SSL 与 TLS (Transport Layer Security, 传输层安全性) 支持。
- JCE (Java Cryptology Extension, Java 加密扩展)：提供了一个用于支持密码技术的框架。

虽然 J2SE 与 JCE 在本质上都是支持密码技术的 API，但由于美国出口控制的限制，它们是分离的。

Java Authentication and Authorization Service：JAAS 是一个 Java 包，提供了进行用户认证以及加强用户访问控制的框架。JAAS 认证是以一种“可插的”方式实现的，从而允许 Java 应用程序与底层的认证技术相分离。JAAS 授权扩展了 Java 安全性体系结构，并且提供了一种授予访问控制的方法，该方法不仅能够基于正在运行的代码（也就是基于代码的来源以及谁标记该代码），也能够基于是谁在运行该代码进行访问控制的授予。

Java Secure Socket Extension：JSSE 提供了 TLS 的 SSL 实现。大多数 JSSE 类都包含在 `javax.net.ssl` 包中，但有一些包含在 `javax.security.cert` 与 `javax.net` 包中。

JSSE 提供了 `SSLSocket` 类与 `SSLServerSocket` 类，分别对应于非安全性 `Socket` 类与 `ServerSocket` 类。为了使代码修改最小化，JSSE 提供了套接字工厂（分别是 `SocketFactory`、`ServerSocketFactory`，`SSLSocketFactory` 与 `SSLServerSocketFactory`）。

后面的内容将对 SSL 进行更详细的描述。

Java Cryptography Extension：JCE 提供对加密、密钥生成、密钥 agreement 以及 MAC (Message Authentication Code, 消息认证码) 算法的支持，从而加强 Java 安全性。

JCE 是为了扩展作为 Java 2 平台一部分的 JCA (Java Cryptography Architecture, Java 密码体系结构) API 而创建的。为了遵守美国出口控制，JCE 被单独提供。后来，人们开发了一个关于 JCE 的、不受出口控制的变体，并且将它集成到 J2SE 1.4 中。

JCE 类位于 `java.security`，`javax.crypto`，`javax.crypto.interface` 与 `javax.crypto.spec` 包中。

JCE 使用不同的加密算法对数据流进行加密，并且在给出了所需的密钥之后，对这些数据流进行解密（取决于密钥的可用性）。

13.3 安全多用途 Internet MIME 扩展

S/MIME (Secure Multipurpose Internet MIME Extension, 安全多用途 Internet MIME 扩展) 是一个用于安全电子邮件的规范,安全电子邮件为 MIME 格式的电子邮件消息引入安全性选项。S/MIME 是 RSA Data Security 的第一步,它使用数字签名提供认证手段,并且使用加密提供保密性。S/MIME 已经被提交给 IETF (Internet Engineering Taskforce, Internet 工程任务小组),并且它是安全电子邮件的事实标准。与大多数其他协议一样,S/MIME 使用公共密钥算法建立对称密钥,然后,使用这个对称密钥进行加密。X.509 数字证书用于认证。

注意:RSA 代表 Rivest, Shamir 与 Adleman (他们是公共密钥加密技术的发明者)。

安全电子事务规范

SET (Secure Electronic Transaction, 安全电子事务) 规范是 MasterCard International, Visa International, GTE, Microsoft, Netscape, SAIC, Terisa Systems 与 VeriSign 开发的开放工业标准。SET 是为了允许消费者与交易商通过 Web 进行安全的银行卡交易而设计的。

SET 是为了在开放网络 (特别是 Internet) 进行安全的支付卡事务而设计的一个技术规范。SET 的目标是允许在 Internet 上进行银行卡事务就像在零售店进行买卖一样简单和安全。由于目前有超过 60 000 000 的人使用 Web (这些人大多数都有较高的收入),因此,每个人都对 SET 存在极大的兴趣。这个规范由三部分组成:业务需求文档、开发人员信息文档以及协议规范文档。SET 的核心部分是支付协议,当消费者选择使用银行卡支付 Internet 上购买的商品与服务时,这个协议可用来保护消费者的银行卡信息。

SET 最初目标是设计成为一个开放的规范,用来确保正确的安全性模型与开放的可互操作标准。SET 使用加密的概念来确保支付的完整性并且对购买者与交易商进行认证。在标准化方面,形成一个标准规范是为了确保一个厂商所开发的应用程序能够与另一个厂商开发的应用程序进行互操作。因此,SET 不仅定义了操作的语义,而且定义了将要被使用的连线级协议。

下面给出了 SET 提出的业务需求 (在 SET Business Description (Book 1) 中定义):

- 提供关于支付信息与订单信息的保密性
- 确保数据完整性
- 认证持卡者就是这个帐户的合法用户
- 认证交易商与所使用的卡的发行公司存在财务关系
- 确保事务处理信息的安全传输
- 确保安全性机制的灵活性与可扩展性
- 确保可互操作性

有一点很有意思，需要说明一下。支持 SET 事务的系统并不是一个数字现金体制。持卡者与交易商进行卡信息的通信。交易商使用这些信息与金融机构通信，并且获得进行事务的授权。支付过程本身并不属于 SET 的范围。

证书被 SET 大量使用，一方面用来证明持卡者（实际上，证书就是我们平常所使用的银行卡），另一方面用来证明交易商与金融机构存在关系（类似于交易商的窗口上所贴的标签，用来告知在那里可以使用的银行卡）。SET 证书是通过一个层次式的信任方案进行验证的。每个证书都链接到提供证明的组织，从而允许信任树被以层次化方式遍历。

13.4 安全套接字层

SSL (Secure Socket Layer, 安全套接字层) 是一个支持通过 Web 进行保密、认证通信的安全性协议。这个协议位于应用程序级协议（例如，HTTP 或 CORBA 调用）之间，并且在 TCP/IP 之间，用来阻止窃听、篡改或消息伪造。SSL 协议提供了数据加密与认证功能，并且提供了数据完整性验证功能。由于这个协议在应用程序级协议之下，因此，它可用于任何基于 TCP/IP 的应用程序级协议（例如 FTP, IIOP 等等）。此外，SSL 对于应用程序级是透明的，这一特点使得 SSL 能够很容易地被集成并且很容易将 SSL 添加到当前可用的 Web 软件中。SSL 是事实上的工业标准。

SSL 主要是为了提供通过 Internet 进行的应用程序级通信的保密性与可靠性而设计的。SSL 被设计成为两个互相作用的层：较低级的层是位于 TCP/IP 上的 SSL Record Protocol，它为第二个层次（称为 SSL Handshake Protocol）提供低级服务。这个协议为应用程序级协议提供了对必要安全性机制的访问。SSL Handshake Protocol 允许两个相互作用的应用程序互相认证（若必要）并且协商加密算法与加密密钥。

13.4.1 SSL 安全性特点

SSL 有 3 个主要的、与安全性相关的特点：

- **保密性**：使用 SSL 连接的应用程序会话是保密的。在 SSL 的初始化阶段，允许两个功能组协商用于数据流加密的保密密钥。SSL 数据加密使用诸如 DES (Data Encryption Standard, 数据加密标准) 与 RC4 (Ron's Code 4) 这样的算法来完成；RC4 是一个由 Ron Rivest 发明的可变密钥大小的加密算法。
- **认证**：SSL 连接的每个功能组都可以使用公共密钥算法对对方的身份进行认证。
- **可靠性**：SSL 连接是可靠的，并且在其中包括完整性检查机制，用来对付恶意的篡改攻击。

此外，SSL 还是高效和可扩展的。SSL 包括一个 cache 机制，允许多个会话使用相同的连接，

从而节省初始建立时间。SSL 被设计成为一种能够在其中集成更多加密算法（只要使用双方都支持该算法）的框架，因此 SSL 能够提供可扩展性。

显然，为了进行 SSL 事务，会话双方都必须使用 SSL。在 Web 会话中，这意味着为了保证一个安全的 SSL HTTP 会话发生，浏览器与 Web 服务器都必须支持 SSL。在基于 CORBA 的分布式系统中，这意味着 ORB（Object Request Broker，对象请求代理）双方都必须支持 SSL。

SSL 操作是一个两阶段的会话，这两个阶段是：初始化阶段与数据传输阶段。此外，各种警告处理程序能够确保安全性攻击不会破坏会话或损害对方的安全性。TCP/IP 连接的初始化涉及安全性握手过程，在这个过程中，客户与服务器就会话的安全性达成一致，并且在这个过程中可能进行认证（客户对服务器的认证以及服务器对客户的认证）。在数据传输过程中，将使用某种对称密钥加密算法。此外，数据完整性机制将用来确保数据没有被意外事件或恶意攻击所破坏。

SSL 为消息传输以及客户/服务器认证提供了数据加密支持。SSL 用于数据加密的技术主要是 RSA 数据安全性技术。对于事务的某一端位于美国本土之外的情况，将使用具有 40 位密钥的 RC4 流加密算法。使用该算法加密的一条消息，通常需要一台典型的工作站花费大约一年的时间对该消息进行“蛮力计算”攻击。在美国本土，使用的是 128 位密钥，利用蛮力计算方法几乎不能对它进行破坏。SSL 认证基于可信证书机构发布的数字证书并且使用公共密钥进行密码测试。

SSL 支持如下 4 种加密操作：

- 数字签名：SSL 或者使用 RSA 进行签名，这种方法利用一个保密密钥对一个 36 字节的结构进行加密（也就是数字签名）；或者使用 DDS（Defense Security System，防御安全性系统），它激活一个对 20 字节的结构进行数字签名的算法。
- 块密码加密：在这种方式下，每个数据块（明文）都被加密成密文数据块。
- 流密码加密：数据流与一个由密码安全生成器所产生的密码流进行异或（XOR）。
- 公共密钥加密：一个带有“陷阱门”的单向函数被用于加密数据段。公共密钥对连接是不保密的（即对所有人可用），但对所传递的数据段进行解密需要保密信息。

13.4.2 SSL 连接

SSL 连接的第一个阶段将建立一个 SSL 会话。SSL 不是无状态协议，并且在整个连接甚至多个连接（这样允许 SSL 提供改进的性能）之间都维护信息。会话状态包括：

- 会话标识符
- 对等证书
- 压缩方法
- 用来指定所用数据加密算法的密文指示

- 功能组之间使用的主要（对称）保密密钥
- 用来说明会话是否可用于支持多个连接的标志

连接状态包括：

- 随机字节序列，每个都是由连接的一方所选择的
- 用于 MAC 操作的服务器保密密钥
- 服务器用于已加密数据的密钥
- 用于 MAC 操作的客户保密密钥
- 客户用于已加密数据的密钥
- 一个初始化向量集合
- 用于确保正确的消息顺序的顺序号

SSL 连接中第一个也是最重要的阶段就是握手（handshake）阶段。在这个阶段，客户与服务器相互通信，从而建立会话与连接状态；如果双方不能就加密与认证达成一致，那么就不会发生数据传输。握手定义了所使用的协议版本以及将要使用的加密算法。此外，作为握手的一部分，任何一方都可能要求另一方对自己进行认证。最后，握手阶段使用公共密钥加密技术创建一个共享的保密密钥集合。一旦握手完成，则直接进行数据通信（而不是处理异常事件）。

握手阶段以客户向服务器发送一条 `CLIENT_HELLO` 消息开始。然后，服务器将以一条 `SERVER_HELLO` 消息进行响应。每条这样的消息都包括一个由一方产生并且被传递给另一方的随机值，此外，还使用这两条消息进行公共密钥算法的协商。握手的下一个部分就是进行可选的身份认证。假设认证被成功地执行（或假设不需要在第一个阶段进行认证），那么服务器将发送 `SERVER_HELLO_DONE` 消息。然后，客户使用先前选择的公共密钥发送 `CLIENT_KEY_EXCHANGE` 消息。

最后，客户选择一个加密算法并且向服务器发送 `CHANGE_CIPHER_SPEC` 消息，从而安装这个新的加密算法来替换（到目前为止）`NULL` 算法。需要说明的是，后来也可能使用这种消息将一个加密方法替换为另一个。在进行实际的加密算法替换之前，服务器将向客户发送 `CHANGE_CIPHER_SPEC`；只有这时，才进行实际的替换。当服务器成功地替换原来的加密算法之后，将向客户发送一条 `FINISHED` 消息（这时使用的是新加密算法）。这样就确保客户与服务器双方都使用新的加密算法。

SSL 压缩

除了加密之外，SSL 对通过连线传递的所有数据进行压缩。使用作为会话定义的一部分的算法对 SSL 记录进行压缩。默认的算法并不修改流（也就是不执行压缩），但客户与服务器达成一致的任意压缩算法以及相匹配的解压算法都可以使用。在结构方面，`SSLPlainText` 结构被 `SSLCompressed` 结构所替换。

然后，使用加密函数将被压缩的结构转换成 `SSLCiphertext` 结构以及消息认证代码。这样做

是可能的，因为在握手阶段已经对密钥进行了协商。然后，这些结构通过连线被传递，与此一同传递的还有用来监视未到达的报文或以错误顺序到达的报文的顺序号。需要说明的是，加密在压缩之后执行，不能反过来。在接收方，其顺序是相反的：首先进行解密，然后进行解压。

13.4.3 高级特性

SSL 协议定义了一些高级特性，从而使其适合 Web 事务并适合作为 IIOP 会话的基础。SSL 支持一个加密算法修改协议，从而允许客户和服务器重新协商一个加密算法。当通信双方中有一方认为安全性可能会被损害时，就可以使用这个协议进行协商。然后，一方将发出一条特殊的 SSL 消息来通知另一方修改加密算法。

关于高级 SSL 消息的另一个例子就是 Alert Protocol。可以使用警告消息来告知通信一方关于某些异常情况的出现。在某些情况下（例如当警告级别是 fatal 时），将会导致通信会话的终止。

其他异常情况包括：

- 非预期消息
- 记录中包含不正确的消息认证代码
- 解压函数失败
- 握手失败
- 没有证书可用来完成一个证书请求
- 证书被破坏
- 证书过期

13.5 小 结

本章描述了 Java 安全性模型以及 SSL，它们是构成在 WebSphere 上部署的安全应用程序的基础。Java 安全性 API 允许用户定义 ACL 并且保护运行时环境免受入侵。SSL 是大多数人用来为 Web 应用程序实现加密与认证的技术。本书第 19 章与第 44 章将介绍 WebSphere 安全性模型，并讨论如何使用 WebSphere Application Server 实现与管理应用程序的安全性。

第 14 章 Java 消息传递服务

分布式软件系统管理大量复杂的任务，涉及可靠性、安全性、容错、吞吐率以及网络连接。解决这些复杂问题的一种方法就是使用系统组件之间松耦合的异步交互。这与 RMI 以及 CORBA 的概念相反，在这两者中的交互大多数都是同步的（虽然 CORBA 开发了不少异步范型）。同步交互与异步交互之间的关键区别在于，后者对网络延迟与网络故障有更大的适应。

一直以来，企业消息传递系统都是企业计算系统范围的一部分。消息传递系统（例如 IBM 的 MQSeries）已经赢得广泛的欢迎，并且是许多公司分布式软件系统的骨干。消息传递本质上是一种用来达到对等通信松耦合（loosely coupled）目标的技术。换句话说，消息发送者与接收者并不需要对通信进行同步，特别地，它们甚至不需要同时可用。而且，消息双方不需要对对方有任何了解；惟一的要求就是它们需要使用相同的消息格式。

为了给 Java 企业开发人员提供使用 MQSeries 时挂接到遗留系统所需的工具，或在构建使用 MQSeries 的系统时所需的工具，IBM 提供了用于 MQSeries 的 Java 连接器（connector）。它们在本质上是为 MQSeries 提供 Java 接口的封装器。

虽然这样的接口非常有用，但它与用来构建 Java API 所使用的方法并不一致。这就是出现 JMS（Java Messaging Service，Java 消息传递服务）的原因。JMS 是 API 的 Java 扩展，其目标是简化集成面向消息的中间件以及简化集成 Java 应用程序。JMS 是一个基于接口的 API，它允许不同的厂商提供自己的、类似于 JDBC 或 JNDI 的关于 JMS 服务的实现。通过将 API 与实现相分离，JMS 允许开发人员使用消息传递系统（例如 MQSeries），而且允许开发人员编写可以在消息传递系统之间移植的通用代码。JMS 的另一个优点是标准化的 API 更易于学习。

JMS 标准是由 Sun 的 JavaSoft 引入的，但它类似于其他 J2EE 组件，其他厂商（尤其是 IBM）参与了这个标准的定义与改进。

14.1 JMS 体系结构

任何基于 JMS 的应用程序的骨干都是底层的消息传递系统，消息是通过这个消息传递系统传递的。消息传递系统又被称作 JMS 提供者（provider）。JMS 是以独立于平台的方式定义的，这就意味着为某个消息传递系统所编写的代码应该能够在任何其他 JMS 兼容的消息传递系统中工作（可能需要对代码进行某些细小的修改）。JMS API 对消息传递系统进行抽象，从而去除了它们之间的区别。

JMS 应用程序的应用程序级组件称为 JMS 客户 (client)。JMS 客户就是使用 JMS API 生产与消费消息的 Java 代码段。不是 Java 实现的客户称为本地客户 (native client)。这样的客户将直接使用底层的消息传递系统。

JMS 体系结构的两个其他组件分别是消息 (message；在客户之间进行信息通信的对象) 与被管理对象 (administered object；预先配置的对象，通常由管理工具而不是 JMS API 所创建和管理)。有 2 类被管理对象：目标与连接工厂 (connection factories)。

JMS 支持两种消息传递范型：点对点消息传递与发布/订阅消息传递。在 JMS 的行话中，每种这样的消息传递范型被称为 domain (域)。点对点消息传递意味着每条消息都仅有一个消费者，并且消息传递是真正异步的（也就是说，当消息到达时，接收者就能够取出该消息，而不必关心它所在的状态）。点对点消息传递类似于有一个邮箱；收信人可以选择何时打开这个邮箱并接收消息。图 14-1 显示了具有一个发送者与一个接收者的典型点对点消息传递情景。一般来说，可能有几个发送者与几个接收者。

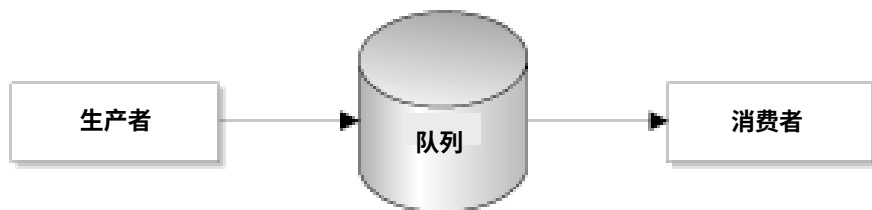


图 14-1 点对点消息传递

发布/订阅消息传递允许一条单独的消息被发送给多个客户。这是通过客户订阅某个消息传递话题来实现的。客户将接收到自己所订阅的关于某话题的所有被发布的消息，如图 14-2 所示。一个消息传递话题类似于电台和电视广播的频道；只要调到该频道，就能够接收到任何被发送的信息，并且任何调到该频道的人将获得相同的内容。

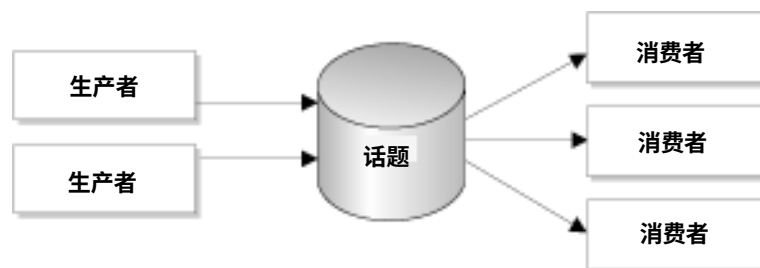


图 14-2 发布/订阅消息传递

客户仅仅接收到自己所订阅的话题的消息，这意味着客户在订阅某一话题之前不能接收到所发送的关于该话题的消息。但是，使用持久订阅 (durable subscription) 能够克服这种不足，持久订阅允许客户接收在它们订阅之前发送的消息。

JMS 支持两种消费消息的方法：通过调用 `receive` 方法同步地消费消息，该方法将阻塞调用者直到消息可用（或者直到超过了某个时限）；也可以通过定义消息听众（`message listener`）异步地消费消息。消息听众类似于事件听众，表现在它们都对感兴趣的事件进行注册；当消息被接收时，它们的 `onMessage` 方法被调用。

14.2 JMS 编程模型

JMS 编程模型引入了下列抽象：

- 连接工厂：通过客户代码用来连接到 JMS 提供者的类。
- 连接：封装了一个到达 JMS 提供者连接的对象。这个连接可以在 TCP/IP 上实现，也可以在任何其他通信介质上实现。
- 会话：关于生产消息与消费消息的上下文。会话可以在连接之上被创建，并且它们被用来创建消息、消息生产者以及消息消费者。
- 目标：封装了一个消息的目标或消息源的对象，在点对点消息传递中就是消息队列，在发布/订阅消息传递中就是话题。
- 消息生产者：为会话创建的对象，用来向目标发送消息。
- 消息消费者：为会话创建的对象，用来从目标获取消息。
- 消息：由头部、可选的属性集合以及可选的主体组成的对象。JMS 的实际目标就是传递消息对象。

上述抽象都有用来代表它的 JMS 接口。除了消息之外，每个接口都有两个接口对其进行扩展，每个 domain 对应一个扩展接口。例如，JMS 为会话定义了一个 `Session` 接口。`TopicSession` 接口为发布/订阅消息传递对这个接口进行了扩展，而 `QueueSession` 接口则为点对点消息传递对这个接口进行了扩展。表 14-1 对这些抽象所对应的接口进行了总结。

表14-1 点对点接口与发布/订阅接口

JMS 父类	点对点 domain	发布/订阅 domain
<code>ConnectionFactory</code>	<code>QueueConnectionFactory</code>	<code>TopicConnectionFactory</code>
<code>Connection</code>	<code>QueueConnection</code>	<code>TopicConnection</code>
<code>Session</code>	<code>QueueSession</code>	<code>TopicSession</code>
<code>Destination</code>	<code>Topic</code>	<code>Queue</code>
<code>MessageProducer</code>	<code>TopicPublisher</code>	<code>QueueSender</code>
<code>MessageConsumer</code>	<code>TopicSubscriber</code>	<code>QueueReceiverQueueBrowser</code>
<code>Message</code>		

14.2.1 连接工厂

连接工厂实现了一个非常著名的工厂模式，它通过客户代码来产生连接。连接工厂是被管理对象，意味着用户需要使用 WebSphere Administrative Console 中的管理工具生成这些被管理对象。

为了访问连接工厂，需要执行一个 JNDI 查找。读者应该记得，在第 5 章为“Hello World”bean 本地接口执行了一个类似的查找。在第 20 章将更详细地讨论 JNDI。

那么，应该如何执行查找呢？实际上，需要实例化一个初始上下文对象，如下所示：

```
Properties env = System.getProperties();
/* Possibly set the naming context URL and factory */
InitialContext ic = new InitialContext(env);
```

现在，就能够访问特定的连接工厂对象了。在 J2EE SDK (Software Development Kit) 中预配置了两个默认的连接工厂对象，因此可以立即访问它们。其中第一个是队列连接工厂，其实例化方法如下：

```
QueueConnectionFactory qFactory = (QueueConnectionFactory)
    ic.lookup("QueueConnectionFactory");
```

另一个是话题连接工厂，其实例化方法如下：

```
TopicConnectionFactory topicFactory = (TopicConnectionFactory)
    ic.lookup("TopicConnectionFactory");
```

14.2.2 连接

连接对象封装了一个到达 JMS 提供者的通信链接。这个链接是一个到达消息传递系统的连接，而不是到达消息目标的连接。基于消息的通信范型是无连接的；任何时候都不存在对等连接。

那么，如何获得一个连接呢？在给出了前面讨论的一类连接工厂之后，就可以创建连接。创建连接所使用的方法名称取决于希望创建队列连接还是创建话题连接。如果需要创建队列连接，那么使用下列语句：

```
QueueConnection qConn = qFactory.createQueueConnection();
```

如果需要创建话题连接，那么使用下列语句：

```
TopicConnection tConn = topicFactory.createTopicConnection();
```

使用这些连接，就能够创建会话对象，这将在下一小节中讨论。

当打开连接时，位于连接另一方的 JMS 提供者将分配某些系统资源。由于这一过程通常发生在一个独立的 JVM 中，也可能发生在一台独立的机器上，因此垃圾收集器并不能管理这

些资源的释放。用户必须使用 `close` 方法显式地进行这一工作，例如：

```
/* Close Queue Connection */
qConn.close();

/* Close Topic Connection */
tConn.close();
```

14.2.3 会话

会话对象用来生产消息与消费消息。需要提醒的是，会话是一个单线程的消息传递上下文，因此在多线程应用程序中可能需要创建几个这样的对象，每个对象对应一个线程。

正如表 14-1 显示的那样，根据所对应的 domain 不同，会话有两种形式。可以从话题连接创建话题会话，如下所示：

```
TopicSession tSession = tConnection.createTopicSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

其中第一个参数说明是否使用事务；如果这个参数为 `false`，那么表示不进行事务会话。第二个参数指出消息应答所需的行为，它仅仅对不进行事务的会话有意义。对于进行事务的会话，在确认事务时自动进行应答（关于这个问题的更多信息，请参见 14.4.3 节“JMS 与事务”）。下面说明了可能的应答值以及它们的含义：

- `Session.AUTO_ACKNOWLEDGE`：一旦客户获取消息，会话就自动在客户方进行应答。
- `Session.CLIENT_ACKNOWLEDGE`：客户必须使用消息的应答方法显式地应答消息。任何这样的应答方法将被会话应用于所有被消费的其他消息（在应答之前进行）。
- `Session.DUPS_OK_ACKNOWLEDGE`：允许会话较迟钝地应答消息，这样就可能创建重复的消息。在某些情况下，这个选项能够改善性能，但这种方式只能被那些能够接受重复消息的客户所使用。

创建队列会话与创建话题会话非常相似，并且参数也是相同的。下面给出了一个例子：

```
QueueSession qSession = qConnection.createQueueSession(false,
    Session.AUTO_ACKNOWLEDGE);
```

14.2.4 目标

目标是消息被发送的目的地以及消息的来源。正如表 14-1 显示的那样，Queue 对象代表点对点消息传递的目标，Topic 对象代表发布/订阅消息传递的目标。

必须使用 JNDI 查找目标对象，这类似于查找连接所用的方法。例如，使用为连接定义的代码段的命名上下文，就可以利用下列代码行查找一个称为 Queue1 的队列：

```
Queue queue1 = (Queue) ic.lookup("Queue1");
```

类似地，可以按以下方法查找一个话题：

```
Topic topic = (Topic) ic.lookup("ImportantTopic");
```

14.2.5 消息生产者与消息消费者

消息生产者与消息消费者是会话为了发送与接收消息而创建的对象。这些对象将作为客户方消息传递队列（或话题）的代理。

在点对点消息传递中，QueueSender 与 QueueReceiver 分别代表消息生产者与消息消费者。可以使用会话对象创建消息生产者与消息消费者。例如：

```
QueueSender qSender = qSession.createSender(queue1);  
QueueReceiver qReceiver = qSession.createReceiver(queue2);
```

通过调用 send 方法，就能够发送一条消息，例如：

```
qSender.send(message)
```

当然，首先必须有消息对象。可以使用 receive 方法接收一条消息，如下所示：

```
"message="/qReceiver.receive();" 
```

这个方法实现了同步消息消费。也可以使用消息听众异步地消费消息。消息听众（message listener）是一个对象，该对象实现了 MessageListener 接口。可以在消息消费者（或者点对点消息传递中的 QueueReceiver）上注册该对象；然后，当一条消息被接收时，就调用听众的 onMessage 方法。

在创建消息消费者时，能够可选地指定感兴趣的消息类型。应该使用本章后面 14.4.2 节“消息选择器”将要讨论的 SQL（Structured Query Language，结构化查询语言）表达语法的子集来指定感兴趣的消息。

JMS 为发布/订阅消息传递定义了类似的生产者与消费者（见表 14-1）。消息听众的概念可应用于发布/订阅消息传递的范型。

14.2.6 消息

消息是 JMS 中最基本的抽象，它们是定义整个 JMS 体系结构的目标。一条 JMS 消息由头部、（可选的）属性以及（可选的）主体组成。

消息头部

消息头部包含一个固定的字段集，用来标识该消息以及帮助对该消息进行路由。每条消息都有一个惟一的消息 ID（由 JMSMessageID 头部字段表示）以及一个目标（队列或话题）；

由 JMSDestination 字段表示)。其他字段还包括：

- JMSPriority：消息优先级。
- JMSTimestamp：消息发送时所放置的时间戳。
- JMSExpiration：消息的过期时间。
- JMSDeliveryMode：传递方式 (DeliveryMode.PERSISTENT 或 DeliveryMode.NON_PERSISTENT)。前一种方式具有更大的开销，但能够确保消息传递，即使消息传递系统失败时也可工作。
- JMSCorrelationID：将消息与其他消息链接起来的应用程序级的值。
- JMSReplyTo：一个目标对象，其中给出了对消息进行应答的地址。
- JMSType：一个说明消息类型的字符串；它通常依赖于 JMS 提供者。
- JMSRedelivered：一个 boolean 值，JMS 提供者用它来说明该消息是第二次被传递。

消息属性

消息属性本质上是应用程序级的头部字段。属性名是 Java String 对象，并且它们的值是 Java 的原始类型，例如 boolean, byte, short, int, long, float, double 与 String。

只能在发送方设置属性值。消息属性对于消息接收者是只读的。

使用 setObjectPropertyMethod 方法设置消息属性，使用 getObjectProperty 方法获取消息属性。

消息体

消息体是可选的，它可以有 6 种不同的形式。对于每种形式，JMS 定义了一个消息类型，该消息类型是 Message 接口的扩展接口。对于每个这样的接口，都提供了相应的设置方法与获取方法。下面给出了这些消息类型：

- Message：消息体为空。
- TextMessage：消息体由 java.lang.String 对象组成。
- BytesMessage：消息体由字节流组成。
- StreamMessage：消息体由 Java 原始类型值的流组成。
- ObjectMessage：消息体由可串行化对象组成。
- MapMessage：消息体由相关联数组组成，其中，数组由名称-值对组成（名称是 Java 字符串，值是 Java 原始对象）。

14.3 编写一个 JMS 客户

一个典型的 JMS 客户编写过程如下：

1. 使用 JNDI 查找获得 ConnectionFactory 对象。

2. 建立 JMS 连接对象。
3. 创建 JMS 会话对象。
4. 使用 JNDI 查找 JMS 目标对象。
5. 创建 JMS 消息生产者、消息消费者或同时创建这两个对象。
6. 发送或接收消息。

14.3.1 点对点消息传递客户

为了演示上述步骤，在这里考察最简单的 JMS 客户，它将发送一条文本消息。为简单起见，下面例子中的消息将局限于单个单词。

```
import javax.naming.*;
import javax.jms.*;

public class HelloQueueSender
{
    public static void main(String[] args)
    {
        Context                ic = null;
        QueueConnectionFactory qConnFactory = null;
        QueueConnection        qConn = null;
        Queue                  queue = null;
        QueueSession           qSession = null;
        QueueSender            qSender = null;
        TextMessage            message = null;
        if (args.length != 2) {
            System.out.println("Usage: java HelloQueueSender " +
                               "<queue-name> <message>");
            System.exit(1);
        }
        String qName = new String(args[0]);
        String fName = "QueueConnectionFactory";
        try {
            /* Lookup connection factory and queue */
            ic = new InitialContext();
            qConnFactory = (QueueConnectionFactory)
                           ic.lookup(fName);
            queue = (Queue) ic.lookup(qName);
        } catch (Exception e) {
            System.err.println("JNDI Lookup failed" + e);
            System.exit(1);
        }
    }
}
```

```

    try {
        /* Create connection, session, sender, message */
        qConn = qConnFactory.createQueueConnection();
        qSession = qConn.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);
        qSender = qSession.createSender(queue);
        message = qSession.createTextMessage();
        /* Set message text */
        message.setText(new String(args[1]));
        /* Send message */
        qSender.send(message);
        /* Close connection */
        qConn.close();
    } catch (Exception e) {
        System.err.println("Error on Sending Message" + e);
    }
}
}

```

现在，在接收方考察这条消息。大多数代码是相同的；不同的代码部分用来实现实际的接收操作。

```

import javax.naming.*;
import javax.jms.*;

public class HelloQueueReceiver
{
    public static void main(String[] args)
    {
        Context                ic = null;
        QueueConnectionFactory qConnFactory = null;
        QueueConnection        qConn = null;
        Queue                  queue = null;
        QueueSession           qSession = null;
        QueueReceiver          qReceiver = null;
        if (args.length != 1) {
            System.out.println("Usage: java HelloQueueReceiver " +
                "<queue-name>");
            System.exit(1);
        }
        String qName = new String(args[0]);
        String fName = "QueueConnectionFactory";
        try {
            /* Lookup connection factory and queue */

```

```

        ic = new InitialContext();
        qConnFactory = (QueueConnectionFactory)
                        ic.lookup(fName);
        queue = (Queue) ic.lookup(qName);
    } catch (Exception e) {
        System.err.println("JNDI Lookup failed" + e);
        System.exit(1);
    }
    try {
        /* Create connection, session, receiver */
        qConn = qConnFactory.createQueueConnection();
        qSession = qConn.createQueueSession(false,
                Session.AUTO_ACKNOWLEDGE);
        qReceiver = qSession.createReceiver(queue);
        /* Receive message */
        qConn.start();
        TextMessage m = (TextMessage)qReceiver.receive(1);
        if (m != null) {
            System.out.println("Read message: " + m.getText());
        }
        /* Close connection */
        qConn.close();
    } catch (Exception e) {
        System.err.println("Error on Sending Message" + e);
    }
}
}

```

14.3.2 发布/订阅消息传递客户

为了演示发布/订阅消息传递，这里给出了一个涉及文本消息的简单例子。前面小节中的例子的精华是所使用的两个类，其中一个用于发送消息，另一个用于接收消息。下面的程序清单显示了用于发送消息的类。`main` 方法对用户输入进行循环，并且发送每一行作为在命令行中指定的话题的一条消息。空字符串表示发送过程的结束。

```

import java.io.*;
import java.util.*;
import javax.naming.*;
import javax.jms.*;

public class HelloTopicSender
{
    private TopicSession tSession;

```

```
private TopicPublisher publisher;
private TopicConnection conn;
private InitialContext ic;

public HelloTopicSender(String topicName)
{
    try {
        /* Lookup connection factory and queue */
        ic = new InitialContext();
        String tName = "TopicConnectionFactory";
        tConnFactory = (TopicConnectionFactory) ic.lookup(tName);
    } catch (Exception e) {
        System.err.println("JNDI Lookup failed" + e);
        System.exit(1);
    }
    /* Create connection, sessions */
    try {
        conn = tConnFactory.createTopicConnection();
        tSession = conn.createTopicSession(false,
                                           Session.AUTO_ACKNOWLEDGE);
        Topic topic = (Topic)jndi.lookup(topicName);
        publisher = tSession.createPublisher(topic);
    } catch (Exception e) {
        System.err.println("Message topic initialization failed" + e);
        System.exit(1);
    }
}

/* Create and send message using topic publisher */
protected void send(String text)
{
    TextMessage message = session.createTextMessage( );
    message.setText(text);
    publisher.publish(message);
}

/* Close the JMS connection */
public void shutdown()
{
    try {
        conn.close( );
    }
    catch (Exception e) {
        /* Report error */
    }
}
```



```
    }  
}  
  
public static void main(String [] args) {  
    try {  
        if (args.length != 1) {  
            System.err.println("Usage: java HelloTopicSender " +  
                                "<topic>");  
            System.exit(1);  
        }  
        HelloTopicSender sender = new HelloTopicSender(args[0],  
                                                         args[1],args[2]);  
        BufferedReader userInput =  
            new BufferedReader(new InputStreamReader(System.in));  
  
        // Loop until an empty line is entered  
        while(true){  
            String s = commandLine.readLine( );  
            if (s.equals("")){  
                sender.shutdown();  
                System.exit(0);  
            } else  
                sender.send(s);  
        }  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}
```

这个例子的第二个部分是用来订阅某条消息的代码段。为了演示消息听众的用法，从 `MessageListener` 继承类。在这个特定的例子中，实现了一个不断空闲并等待消息的循环查询，但在其他情况中，也可以使用基于事件的处理程序。

```
import java.io.*;  
import java.util.*;  
import javax.naming.*;  
import javax.jms.*;  
  
public class HelloTopicReceiver implements javax.jms.MessageListener  
{  
    private TopicSession tSession;  
    private TopicSubscriber subscriber;  
    private TopicConnection conn;
```

```
private InitialContext ic;

public HelloTopicReceiver(String topicName)
{
    try {
        /* Lookup connection factory and queue */
        ic = new InitialContext();
        String tName = "TopicConnectionFactory";
        tConnFactory = (TopicConnectionFactory) ic.lookup(tName);
    } catch (Exception e) {
        System.err.println("JNDI Lookup failed" + e);
        System.exit(1);
    }
    /* Create connection, sessions */
    try {
        conn = tConnFactory.createTopicConnection();
        tSession = conn.createTopicSession(false,
            Session.AUTO_ACKNOWLEDGE);
        Topic topic = (Topic)jndi.lookup(topicName);
        subscriber = tSession.createSubscriber(topic);
        subscriber.setMessageListener(this);
        conn.start();
    } catch (Exception e) {
        System.err.println("Message topic initialization failed" + e);
        System.exit(1);
    }
}

/* Recive message */
public void onMessage(Message msg)
{
    try {
        TextMessage message = (TextMessage) msg;
        System.out.println(msg.getText());
    }
}

public static void main(String [] args) {
    try{
        if (args.length != 1) {
            System.err.println("Usage: java HelloTopicReceiver " +
                "<topic>");
            System.exit(1);
        }
    }
}
```

```
    }  
    HelloTopicSender sender = new HelloTopicSender(args[0]);  
    /* Idle waiting for messages */  
    while(true){  
    }  
} catch (Exception e) {  
    e.printStackTrace( );  
}  
}  
}
```

14.4 高级 JMS 特性

用来发送消息的 `send` 和 `publish` 方法，有一个详细的模式，用来指定除了消息之外的另外 3 个参数：传递方式、优先级、过期时间（在方法中使用的顺序也是如此）。

正如本章前面提到的那样，JMS 消息头部包括一个传递方式字段，该字段使用 `Delivery-Mode` 接口中定义的两个值之一：`PERSISTENT` 和 `NON_PERSISTENT`。持续传递是默认方式，它确保消息没有丢失；为了达到这一要求，必须将消息存储到稳定的存储介质中。可以使用 `setDeliveryMode` 方法（为消息生产者定义的）或使用前面提到的 `send` 方法的详细形式设置传递方式。

消息的优先级决定消息被传递的顺序，并且允许用户指定某些消息比另一些消息更为紧急。优先级是一个介于 0 到 9 之间的整数，0 表示最低优先级，9 表示最高优先级。默认的优先级值是 4。可以在创建会话时使用 `MessageProducer` 接口的 `setPriority` 方法，或者在发送消息时指定优先级。

消息的过期时间对于在将来的某个时刻必须作废的任何消息是非常有用的。默认地，消息永不过期。因此，若使用消息生产者的 `send` 方法或 `setTimeToLive` 方法指定一个过期时间（以毫秒表示），那么如果在指定的时间结束之前消息还没有被传输，该消息将被析构。

考虑下面的例子：

```
qSender.send(message, DeliveryMode.NON_PERSISTENT, 9, 1000);
```

这个代码利用最高的优先级发送一条非持续消息；但是，如果没有在 1 秒之内传输出去，该消息将被析构并且在后来的时间里不会被传输。

14.4.1 持久订阅

使用持续传递方式能够确保发布者在发布/订阅方式下发送的消息不会丢失。但是，只有订阅者处于活跃状态时才能接收到这些消息。为了演示这种情况，可以考虑蜂窝电话的 SMS

(Short Message Service, 短消息服务) 消息。要求订阅者处于活跃状态, 就意味着蜂窝电话被关闭时到达的消息将丢失。

虽然这种行为在某些情况下是可以接受的, 但并非对所有情况都适用。这一问题的解决方案就是定义持久订阅。持久订阅 (durable subscription) 有一个惟一的标识, 一旦某个标识被订阅, 那么 JMS 提供者将维护这些订阅, 即使在请求方不再活跃时 (也就是连接被关闭) 也是如此。毫无疑问, 这样会带来额外的代价 (也就是增加消息传递系统的开销)。其优点是, 订阅者 (订阅者指定了一个先前所注册持久订阅的 ID) 一旦注册, 该订阅者将接收到关于该话题的所有被发布的消息。

为了建立持久订阅, 需要为连接定义一个连接 ID 并为订阅者定义一个订阅名称。这两个定义的组合就能够标识订阅者。

连接 ID 是用管理工具设置的; 也就是, 需要为目标客户 ID 添加一个新连接工厂。然后, 使用具有该 ID 的连接就可以创建订阅。例如:

```
TopicSubscriber durableSubscriber =  
    tSession.createDurableSubscriber(importantTopic, "Sub1");
```

由于持久订阅会对消息传递系统带来一些开销, 并且由于消息传递系统不能确定什么时候不再需要持久订阅, 因此用户必须显式地取消订阅。为此, 需要使用会话的 `unsubscribe` 方法, 但只有在关闭订阅之后才能进行这个调用。下面给出了在先前代码行中所创建的持久订阅的例子:

```
durableSubscriber.close();  
tSession.unsubscribe("Sub1");
```

14.4.2 消息选择器

在许多情况下, 消息接收者并不是对发送给他们的所有消息都有兴趣, 就像读者可能并不是对家里所接收到的所有邮件都感兴趣一样。JMS 提供了类似于 SQL 的表达式的消息过滤机制, 从而选择消息。在创建消息消费者时可以定义这个机制。用来创建消息消费者的每个会话方法都有一个完全的形式, 在这种形式中可以指定消息选择器。例如, 在创建持久订阅时可以使用下列代码:

```
TopicSubscriber durableSub =  
    tSession.createDurableSubscriber(topic,  
        "Sub2", "dollarAmount > 100", true);
```

这个调用中的前二个参数分别是话题对象与订阅名称 (这与前面持久订阅的例子相同)。第三个参数是消息选择器, 最后一个参数指定了订阅者是否忽略本地发布的消息 (`true` 值表示本地发布的消息将被忽略)。

消息选择器是一个字符串, 它能够指定一个利用消息头部与属性定义的表达式。如果读者

对 SQL 比较熟悉,那么可以想象有一个关于所有消息的表,表中的列表示消息头部和属性,然后,用户就为 SELECT 语句指定 WHERE 短语。

14.4.3 JMS 与事务

事务提供了一种可靠的执行操作的方式。这就是它们在几乎每个企业应用程序中都发挥作用的原因。某些 J2EE 技术解决了事务问题,例如, JDBC Connection 对象、EJB、JTS (Java Transaction Service, Java 事务服务) 与 JTA (Java Transaction API, Java 事务 API)。

在消息传递系统中事务的重要性表现在,在某些情况下要求几条消息一同被原子性 (atomicity) 地传输。也就是说,这些消息要么全部被传输,要么全部不传输。在出现失败的情况下,发送者能够回滚该操作。

一般来说,事务支持原子性更新的概念,在这种更新方式中,操作要么全部完成,要么根本没有进行,但不会出现操作被执行一部分的情况;事务支持一致性,从而使得失败的事务不会对数据造成任何影响;事务支持持久性,从而使得修改是真正持续的;事务还支持隔离,从而使得事务进行的修改对其他事务不可知,直到该事务被确认。

JMS 支持二种不同类型的事务:本地事务与分布式事务。本地 (Local) 事务的作用范围上被限制在一个单独的客户上,并且它们提供了将几条消息归组在一起的能力。本地事务是通过 Session 接口管理的。在创建会话时,需要将第一个参数指定为 true (这样就指定该会话是支持事务的)。然后,就可以使用 Session.commit 方法与 Session.rollback 方法。需要强调的是,这种事务被局限于单个客户的作用范围;因此,在其中发送消息的事务与接收相同消息的其他客户没有任何关系。实际上,如果不是很谨慎,使用本地事务具有严重的缺陷。例如,如果发送了一条消息然后等待应答,所有的工作都在相同事务中进行的话,那么什么也不会发生。换句话说,程序将被挂起。其原因就是消息在被确认之前并没有接收到,因此其他功能组不会给出应答。

如果需要实现分布式 (distributed) 事务,最佳的方法就是使用 EJB 体系结构。第 32 章将详细讨论被称为消息 bean (message bean) 的特殊 EJB 的用法,它在本质上将 EJB 的功能与 JMS 的功能进行融合。

14.5 小 结

消息传递系统已经成为构建分布式系统的最广泛使用的底层结构。目前市场上已经有许多自称为最佳的这样的产品。这些产品可以归结为两个类型:中间件或 EAI (Enterprise Application Integration, 企业应用程序集成)。这些系统为多种编程语言 (包括 Java) 提供了 API。

最初,所有这些产品都有自己专用的 API。随着该商业领域的成熟以及消息传递模型被明确

定义，JMS 最终发展成一个标准 API，通过它能够对任何上述系统进行访问。这种标准化也使得构建使用消息传递模型的分布式应用程序更为容易，并且这些应用程序更具有可移植性。因此，它现在理所当然地成为 WebSphere Application Server 不可分割的一部分。

本章主要从技术角度讨论了 JMS。在第 32 章，将从消息 bean 与 EJB 的角度以及从部署利用 JMS 的应用程序的角度进一步讨论 JMS。在构成 WebSphere 的基础技术方面，还需要讨论 J2EE 模型；下一章将讨论这一话题。

第 15 章 Java 2 企业版

Java 推出了一个平衡的瘦客户，从而迈出了作为主流编程语言的第一步。这主要是通过引入 applet 来实现的，applet 是嵌入在 HTML 页面中的小程序。但是，Java 作为编程语言所存在的特点，也使得它成为开发服务器方代码与基于 Web 的客户的最佳编程语言。Java 对服务器方应用程序的重要性，并不亚于对客户方应用程序的重要性。随着该语言逐渐成熟，它越来越多地被用于服务器上，而不是客户上。它在面向对象语言、安全性功能、可移植性、网络连接等方面的强势，已经使得它成为企业软件市场（在服务器上）的重要语言。但是，由于 Java 还不是很成熟，应用程序服务器开发人员有时在考虑使用 Java 充当这种核心角色时还有所犹豫。在 1997 年，已经明显地出现这种情况：除非有某种结构可用来支持诸如事务、安全性以及分布式之类的问题，否则 Java 将成为编写 applet 的语言（如果这样的话，Java 将逐渐消失）。

现在介绍 EJB（Enterprise JavaBean，企业 JavaBean）。正如在构建客户软件时需要框架和标准来确保某人能够使用由其他人员开发的工具和服务一样，在构建服务器方组件时也面临同样的问题。服务器上的业务逻辑应该集中关注的是构建领域功能，而不是系统级功能（例如事务管理与协调、安全性、命名服务）。如果没有上述问题的解决方案，那么任何严肃的应用程序都难以正常工作，因此，需要有一个标准的 API 集合，来确保服务器上的应用程序组件可以为系统与体系结构问题选择解决方案，而不必受到任何限制，也不必被束缚在某个单独的解决方案上。这种 API 的集合就是 EJB 规范。

后来，Java 2 发布之后，Sun 公司就定义了 J2EE（Java 2 Platform Enterprise Edition，Java 2 平台企业版）。该平台是一个用来在分布式环境下开发、部署与执行应用程序的体系结构。这个平台为应用程序提供了系统级服务，例如事务管理、安全性、客户连接性、数据库访问。虽然 Sun 的 JavaSoft 发布了 J2EE 规范，但这个软件主要是为了与其他厂商（特别是 IBM）实现紧密协同而开发的。J2EE 规范受到许多在设计各种应用程序服务器（例如 IBM 的 WebSphere）过程中所积累的实际经验的影响。

J2EE 的重要性在于，在构建实际的（特别是大型）系统时，开发人员的工作可以被划分为两个主要方面：领域级应用程序编程与系统级问题。这种划分印证了几乎任何大型编程工程都有两种编程者这一事实，其中一种编程者面向实现业务逻辑与领域特定功能的需求，另一种编程者则负责构建“底层结构”（用来处理诸如分布、通信、事务、命名服务等问题）。EJB 试图将这两种类型分开，并且提供了一个框架，在此框架下，应用程序开发人员能够集中精力关注于领域功能，而使用由专家所构建的服务器与容器来解决他们在系统级的需求。这样，企业软件开发人员不仅能够节省大量的时间，并且能够得益于现有的解决方案，这

些解决方案可能远远比他们自己构建的解决方案更为优秀。J2EE 平台允许由厂商开发的底层结构有一个标准的 API 以及标准的框架，从而使其能够被多个领域级应用程序所使用。它给用户带来的好处是改善了可伸缩性与性能。

需要指出的是，虽然 Sun 的 EJB 与 J2EE 定义是拥有一个标准化机制（该机制用来为 Java 服务器方代码定义底层结构）的第一步，但它并不是为 Java 定义底层结构软件的第一步。J2EE 是一个业界标准，其他公司也为 J2EE 的制定做出了重大贡献，尤其是 IBM。例如，IBM 开发了一个称为 SanFrancisco 的产品，它在本质上是一个较大的底层结构软件包。这个产品可用来处理 EJB 规范中的许多问题，并且，J2EE 的许多思想都借鉴了 SanFrancisco。2001 年 8 月，WebSphere 4.0 实现了 J2EE 规范 1.2 版本。J2EE 是一个正在发展的标准，目前 J2EE 1.3 版本已经最终形成。

本章剩余部分将概述 J2EE。第 18~38 章将详细讨论构成 J2EE 的技术。

15.1 J2EE 组件技术

J2EE 规范定义了一个相关的 API 集合，这些 API 提供了一个用于开发与部署基于 Web 的应用程序（包括 EJB，servlet，JSP）的平台，本书将在第 5 部分和第 6 部分讨论这三种技术。J2EE SDK 是 Sun 提供的参考实现。需要提醒的是，从年代上看，J2EE 是在引入 EJB，servlet 与 JSP 很长时间之后才出现的。因此，虽然 J2EE SDK 提供了参考实现，但 IBM 的 WebSphere（以及来自其他厂商的产品）甚至在 J2EE 规范出台之前就实现了该规范的大部分。但这并不是说 J2EE 的定义就没有价值。实际上，它提供了对先前许多独立 API 的集成。不同的 Java API 通常是独立开发与部署的。利用 J2EE，Sun 提供了一个紧密集成的 API，从而使其软件更加“成熟”。表 15-1 给出了构成 J2EE 的 API 列表。当然，所有这些 API 都是在 J2SE 之上定义的，并且丰富了 J2SE。

表15-1 J2EE技术

API	描述
EJB (Enterprise JavaBean, 企业 JavaBean)	用来构建服务器方业务对象的框架
JDBC (Java Database Connectivity, Java 数据库互连)	用来对 RDBMS (Relational Database Management System, 关系数据库管理系统) 进行独立于数据库的访问的一致 API
JMS (Java Messaging Service, Java 消息传递服务)	用于分布式异步消息传递
JavaMail	用于构建独立于特定协议与平台的邮件与消息传递应用程序框架

(续表)

API	描述
JNDI (Java Naming and Directory Interface, Java 命名和目录接口)	分布式命名与目录服务, 提供对位置透明的数据仓库的一致访问
JSP (JavaServer Page, JavaServer 页面) Java Servlet	用于实现常用于 Web 客户的动态 HTML 或 XML 页面的框架 用于为通过 HTTP 到达的请求产生响应的服务器方代码
事务	JTA (Java Transaction Server, Java 事务服务器), 用来管理 分布式事务的 API JTS (Java Transaction Service, Java 事务服务), 遵守 JTA 规范的 Transaction Manager 的实现规范
XML (Extensible Markup Language, 可扩展标记语言)	作为 Web Service 支持的一部分, J2EE 重点放在 XML 上。 它包括 JAXP (XML Processing) 的 Java API
CORBA (Common Object Request Broker Architecture, 公共对象请求代理体系结构)	Java IDL, 支持 CORBA 可互操作性, 并且包括简化的 Object Request Broker 支持 Internet Inter-ORB Protocol Remote Method Interface (RMI) over IIOP, 用于分布式对象通信 同样, JTS 也是基于 CORBA OTS (Object Transaction Service, 对象事务服务)
连接器	用来集成遗留信息系统 (通常不是用 Java 实现) 的标准 API 与体系结构。连接器不是 J2EE 1.2 的部分, 但它是 J2EE 1.3 的部分

除了 API 之外, J2EE 包括一个测试集合, 用于验证 API 兼容性。J2EE 所使用的技术集合可以被大致划分为 3 个不同类型:

- 组件: 应用程序级软件构建块, 例如 EJB, servlet, JSP, 客户方 applet 与应用程序。
- 服务: 提供了与其他系统集成能力。这些服务包括 JDBC, JNDI 与 JTA。
- 通信: 用来提供通信的技术, 例如 RMI-IIOP (Remote Method Invocation-IIOP), JMS 与 JavaMail。HTTP 也可以被认为是这些技术之一, 但它并不是 J2EE 规范的一部分。

15.2 J2EE 应用程序

J2EE 应用程序可以从三种不同类型的构建块构建: EJB、Web 组件与 J2EE 应用程序客户。J2EE 作为服务器方技术的一个优点是, 与任何 Java 技术一样, 它提高了代码的可移植性与可重用性。

J2EE 应用程序可以包含 0 个或多个企业 bean、Web 组件或 J2EE 应用程序客户组件。企业 bean 由 3 个类文件组成：EJB 类、远程接口与本地接口。Web 组件可能包含下列类型的文件：servlet 类、JSP、HTML 页面或 GIF (Graphics Interchange Format) 图像。J2EE 应用程序客户是一个 Java 应用程序，该应用程序运行于一个允许它访问 J2EE 服务的环境中。

J2EE 应用程序，以及每个 Web 组件、EJB 或 J2EE 应用程序客户，都有一个部署描述符。部署描述符是一个 XML 文件，它描述了组件及其部署时行为。例如，简单地说，EJB 的部署描述符声明了事务属性与安全性授权。由于这些信息具有说明性，因此可以在不修改源代码以及不进行重新编译的情况下就可以对它们进行改动。相反，J2EE 服务器在运行时读取这些信息并且相应地对 bean 进行动作。

J2EE 体系结构的主要优点之一是它提供了中间件（中间层软件，本质上就是 EJB）与客户方软件（通常是 Web 组件）之间的清晰分离。非编程者（例如 Web 设计者）也能够进行客户软件的准备工作。J2EE 体系结构提供了另一个重要的区分就是实际组件（例如 EJB）与它们的部署描述符之间的区分。这样，服务器管理员就能够在不需要对实际 Java 类文件进行修改的情况下对某些问题（例如对象放置、安全性策略）进行改动。

在大多数常见的设置中，客户方代码可以在 applet 也可以在应用程序里，在这两种情况下，客户都在客户机器的本地 JVM 中执行。当在客户方使用 Web 浏览器时，服务器方的 Web 组件将被调用。动态 Web 组件（也就是 servlet 与 JSP）将生成 HTML 与 XML 页面。在进行这一工作时，它们通常访问 EJB 或其他 J2EE 服务器方 API，例如 JDBC，RMI，JavaMail 或 JMS。

尽管 J2EE 框架对服务器方组件技术与客户方组件技术进行了区分，但实际上它仍然支持真正意义上的 n 层体系结构。用来支持这一特点的技术是 EJB。利用 EJB，就可能以各种方式对软件进行划分。虽然 EJB 基本上是服务器方实体，但也可以作为其他 EJB 的客户。

15.3 J2EE 应用程序部署

对任何应用程序开发，打包与部署都是重要的考虑因素，而对于大规模的企业应用程序，它们就显得更加关键。在 Java 编程语言发展的早期阶段，就出现了将几个 Java 类文件打包在一起的需求。最先的解决方案是使用 ZIP 文件（基于 Phillip Katz 提出的标准 ZIP 压缩格式）。使用 ZIP 文件，就能够在不复制文件目录的需求下部署 Java applet 或应用程序，因为它们全部被打包在一个单独的 ZIP 文件中。

虽然 ZIP 文件为小的应用程序与 applet 提供了较好的解决方案，但仍然有一些信息被丢失（例如为文件提供某些元数据从而使包文件具有完全自描述性的能力）。这样，后来就采用 JAR (Java Archive) 文件格式。在这种格式下，文件仍然被打包并压缩在一个扩展名为 .jar 的单独文件中，并且该格式能够以独立于平台的方式支持层次式目录（实际上，压缩方法与打包方法仍然使用的是 ZIP 格式）。但是，JAR 文件的目录结构包括一个名为 manifest.mf

文件的 META-INF 目录。在这个文件中定义了元数据。使用 jar 命令处理 JAR 文件，这个命令是 JDK 发布的一部分，并且该命令的参数类似于 UNIX tar 命令。

这个 manifest 文件由一系列行的序列构成，每一行具有以下形式：

```
Attribute: value
```

例如，包含有 JavaBean 的 JAR 文件可以有 Java-Bean:true 属性。另一个例子是 Sealed 属性，通过指定这个属性，就能够阻止来自其他文件的类访问这个 JAR 文件中的类。

利用 Main-Class 属性，就可能指定一个类，使得当把这个 JAR 文件作为 jre 或 java 命令的参数时就将调用这个类的 main 方法。

此外，也可以使用 Name 属性，将某个属性集合限制为对一个单独的包有效。例如，如果某个 JAR 文件包含有几个包，并且我们希望对某个单独的包（假设为 com.viryanet.package）进行密封，那么这个 manifest 文件就应该包含有如下所示的行：

```
Name: com/viryanet/package/  
Sealed: true
```

15.3.1 Web 档案文件

Java 应用程序打包的下一个发展阶段就是 WAR (Web Archive) 文件。WAR 文件是扩展名为 .war 的 JAR 文件。WAR 文件是作为 servlet API 规范的一部分提出的，用于部署基于 Web 的应用程序。WAR 文件通常包含 servlet、JSP 以及相关的资源、静态 HTML 文件或 XML 文件。

WAR 文件有一个称为 WEB-INF 的目录，还有一个从 JAR 格式继承而来的 META-INF 目录。在 WEB-INF 目录中可能包含有一个名为 classes 的目录（在其中存储了 Java 类文件以及适当的目录结构）和一个名为 lib 的目录（在其中存储了 JAR 文件和 ZIP 文件以及 Java 类文件）。在 WEB-INF 目录中也包含一个名为 web.xml 的文件，这个文件保存有 Web 应用程序的元数据。

为了演示 WAR 文件的结构，下面考虑一个仅仅包含有名为 test.jsp 的 JSP 文件的最简单 Web 应用程序。可以使用第 4 章描述的 AAT (Application Assembly Tool) 生成 WAR 文件（注意，在第 4 章中还描述了如何创建包含 servlet、JSP 以及资源文件的 WAR 文件）。

在得到 WAR 文件之后，就可以使用下列命令检查其内容：

```
jar tf <filename>
```

例如，对于一个小的 WAR 文件，下面给出了其内容：

```
test.jsp  
META-INF/MANIFEST.MF  
WEB-INF/web.xml  
WEB-INF/ibm-web-bnd.xmi
```

WEB-INF/ibm-web-ext.xmi

在这里，档案文件包括 JSP 文件（因为它需要作为应用程序的一部分被部署）与一些元数据文件。下面给出了 manifest.mf 文件：

```
Manifest-Version: 1.0
Main-Class: null
Class-Path: ./
```

可以看出，这个文件没有什么实际内容。因为，大多数有意义的数据都放在 web.xml 文件中。下面给出了 web.xml 文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.2//EN"
"http://java.sun.com/j2ee/dtds/web-app_2_2.dtd">
  <web-app id="WebApp_1">
    <display-name>test-app</display-name>
    <servlet id="Servlet_1">
      <servlet-name>test</servlet-name>
      <jsp-file>test.jsp</jsp-file>
      <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping id="ServletMapping_1">
      <servlet-name>test</servlet-name>
      <url-pattern>test</url-pattern>
    </servlet-mapping>
  </web-app>
```

可以看出，在这个文件中包括了装配 Web 应用程序时在 AAT 中所指定的大多数参数。DOCTYPE 标记包括关于 Web 应用程序的 JavaSoft 的 DTD（Data Type Definition，数据类型定义）引用。遵守这个 DTD 就能够确保所有的 WAR 文件的确是标准的，并且能够被用于遵守 J2EE 规范的服务器上。

这个 WAR 文件示例的最后就是两个 IBM 的对 WAR 格式进行了扩展的文件。在这些文件中出现的某些参数与设置将在本书第 28 章讨论。

15.3.2 企业档案文件

尽管 JSP 与 servlet 是 J2EE 应用程序中的重要组件，但还需要提及 EJB。当 EJB 被引入时，它们的部署与打包是基于包括某些特殊 manifest 属性的 JAR 文件进行的。这种机制不像 WAR 文件那样灵活，其中的元数据是使用 XML 表达的。

为了提供关于企业应用程序部署问题的全面解决方案，J2EE 标准引入了 EAR（Enterprise Archive）文件（用来充当多个 WAR 文件与 JAR 文件的容器的 JAR 文件）。

按照 WAR 文件的思想, EAR 文件包括一个称为 application.xml 的 META-INF 文件, 在这个文件中包含有应用程序级元数据。例如, 考虑一个包括前一节讨论的 WAR 文件的简单应用程序。这个应用程序的 application.xml 文件如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC "-//Sun Microsystems, Inc.//DTD
J2EEApplication 1.2//EN"
"http://java.sun.com/j2ee/dtds/application_1_2.dtd">
  <application id="Application_ID">
    <display-name>test.ear</display-name>
    <module id="WebModule_1">
      <web>
        <web-uri>test.war</web-uri>
        <context-root>/test</context-root>
      </web>
    </module>
  </application>
```

同样, DOCTYPE 标记中的 DTD 引用也用来确保这个文件的结构遵守 J2EE 规范。大型应用程序的 application.xml 文件可能难于理解(读者将在第 34 章看到这一点), 这就是 AAT 是 WebSphere 环境中一个关键组件的原因。值得一提的是, JavaSoft 的 J2EE SDK 包括一个称为 deploytool 的程序, 可用来创建与处理 JAR, WAR 与 EAR 文件。但是, 这个工具没有 WebSphere 的 AAT 功能强大, 也没有后者易于使用。

15.4 J2EE 体系结构

与其他 Java 对象不同, J2EE 组件(无论是 servlet, EJB 还是 JSP)自己不能运行, 也不能使用 JVM 运行。它需要某种执行环境。J2EE 容器提供了这种执行环境, 同时, J2EE 容器又在 J2EE 服务器中运行。

服务器与容器的概念是在 EJB 规范中引入的。服务器对应于应用程序服务器, 例如 WebSphere。容器是 J2EE 组件的运行环境, 并且容器存在于 J2EE 服务器中。容器为 J2EE 组件提供各种服务, 例如生命周期服务、通信服务。在 J2EE 被引入之前, EJB 规范关于容器与服务器的区分是模糊的。

在 J2EE 体系结构中, 指定了多种类型的容器。除了 EJB 容器之外, 还有 servlet 引擎、JSP 引擎与 Web 容器(Web 组件的容器)。这些容器为 HTTP 与任何其他所需的 API 提供了必要的支持。

J2EE 服务器提供了一个完全的 API 集合, 并且提供了相应的组件实现。J2EE 规范也使得容器与服务器的混合与匹配成为可能。例如, 用户可以将厂商 X 开发的 servlet 引擎安装到厂商 Y 开发的应用程序服务器。实际上, 在 WebSphere 中使用的许多 Apache 元素就是基于这

些功能实现的。

图 15-1 显示了 J2EE 体系结构的主要组件以及它们在一个典型的 J2EE 应用程序中所起的作用。

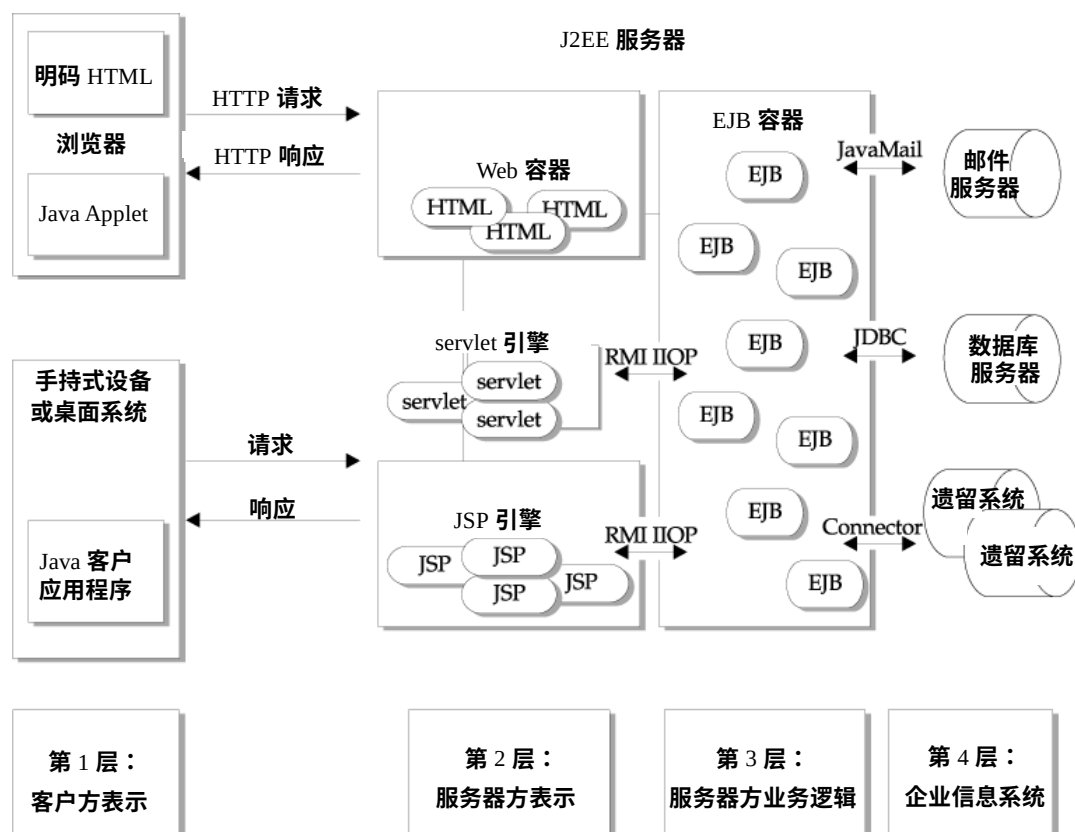


图 15-1 J2EE 体系结构组件

15.4.1 J2EE 服务器

J2EE 服务器提供下列服务：

- HTTP：使得 Web 浏览器能够访问 servlet 与 JSP 文件。
- EJB：允许客户调用企业 bean 上的方法。
- 命名与目录服务：允许程序通过 JNDI API 定位服务与组件（通常就是 EJB）。
- 认证：通过要求用户登录来加强安全性。
- 其他 API：CORBA，JavaMail，Transcation，JMS 与 Connector。

15.4.2 Web 组件容器

Web 组件在 Web 组件容器中存在并运行。Web 容器可能很简单,就是一个关于 HTML,XML 与 GIF 文件的仓库。但是,对于动态 Web 页面,可以使用更复杂的容器,也就是 servlet 引擎与 JSP 引擎。

servlet 容器为 servlet 提供网络服务,例如支持 HTTP (或其他请求-响应协议)。它负责生命周期管理 (例如 servlet 实例的创建与析构) 并提供对 J2EE API (如 RMI) 的访问。

JSP 容器是能够将 JSP 页面转换成 servlet (通过编译) 的 servlet 容器。

15.4.3 EJB 容器

EJB 实例在 EJB 容器中运行, EJB 容器是一个控制企业 bean 并提供重要的系统级服务的运行时环境。EJB 容器为企业 bean 提供了下列服务:

- 生命周期管理
- 事务管理
- 安全性
- 远程客户互连
- 数据库连接池

生命周期管理

EJB 容器能够完全控制 bean 的生命周期, 并且执行与生命周期相关的操作作为对客户请求的响应。一个 EJB 在其生命过程中将经历几个阶段。容器创建企业 bean, 将它转移到可用实例池与活跃状态之间, 最后删除这个企业 bean。

事务管理

当客户调用企业 bean 中的一个 EJB 方法时, 由 EJB 容器负责确保 bean 的事务的一致性。由于容器管理事务, 因此开发人员不需要实现企业 bean 中的事务代码。相反, bean 的事务行为在一个称为部署描述符 (deployment descriptor) 的外部文件中提供。

安全性

安全性是分布式应用程序 (特别是基于 Web 的应用程序) 的必要考虑因素。EJB 容器仅仅允许授权客户调用企业 bean 的方法。类似于事务行为, 安全性定义并不是开发人员需要编码的部分, 而是在部署时定义的。

远程客户互连

EJB 容器基于 Java RMI, 管理客户与企业 bean 之间的低级通信。在创建企业 bean 之后, 客户调用这个 bean 上的方法时就像这个 bean 在相同的虚拟机上一样。

数据库连接池

在任何现实的设置中，数据库连接都是代价较高的资源。获取数据库连接的过程比较耗时，并且连接的数目可能有限制。由于任何具有实用价值的应用程序都需要数据库访问，因此 EJB 容器试图解决这一问题。为此，容器管理了一个数据库连接池，企业 bean 能够快速从这个池中获取一个连接。在 bean 释放连接之后，该连接又可以被另一个 bean 所重用。

15.4.4 servlet 与 JSP

servlet 为生成动态 Web 内容提供了一个简单而功能强大的 API。虽然 servlet 是为了用作支持请求-响应协议的通用机制而设计的，但它们最常见的用法是处理关于 Web 内容的 HTTP 请求。JSP 提供了一种不同的机制，用来生成动态 Web 内容。利用 JSP，Java 代码将不包含普通的 Java 类；相反，这个代码被嵌入在 HTML（或其他内容）页面内部，它类似于一种脚本语言。

15.4.5 EJB

EJB 是使用 Java 编程语言编写的可重用服务器方组件。企业 bean 包括用户应用程序的业务逻辑。EJB 有两种类型：

- 会话 bean
- 实体 bean

会话 bean 在 J2EE 服务器中代表客户进行动作。客户通过调用属于企业 bean 的方法与 J2EE 服务器通信。会话 bean 与客户进行对话，并且可以被看作客户的扩展。会话 bean 可以是无状态的，也可以是有状态的。无状态会话 bean 可以有許多不同的客户并且可以被放置在池中。有状态会话 bean 只有一个客户。当这个客户终止时，相应的会话 bean 也终止。在任何情况下，会话 bean 都是临时性的，而不是持续性的。

实体 bean 代表持续性业务对象。实体 bean 的信息通常存储在关系数据库中，但并非总是如此（也可以存储在对象数据库、遗留应用程序、文件或其他任何存储机制中）。所使用的存储机制的类型依赖于 EJB 技术的特定实现。参考实现（J2EE SDK）使用关系数据库，IBM WebSphere 也使用关系数据库。

15.4.6 J2EE Connector

J2EE Connector 体系结构的目标是为遗留系统或任何不是基于 Java 的企业系统与基于 Java 的代码的集成提供一个解决方案。这样的系统的例子包括大型机事务、ERP（Enterprise Resource Planning，企业资源规划）、数据库系统。在 J2EE Connector 体系结构引入之前，Java 平台并没有包括任何组件可用来解决遗留系统的集成问题。这样就导致出现大量由不同厂商提供的产品，用来提供对它们自己的非 Java 系统的连接。例如，IBM 提供了自己的连接

器，用来连接到 CICS（Customer Information Control System，客户信息控制系统）。

J2EE Connector 体系结构提供了一个框架，在此框架下，EIS（Enterprise Information System，企业信息系统）的提供者能够为它的 EIS 实现一个资源适配器。资源适配器被插入到 J2EE 服务器中，由它通过 J2EE 服务器提供企业应用程序与 EIS 之间的连接。任何正确实现的适配器都能够确保 J2EE 服务器中的应用程序及其各自 EIS 之间的无缝通信。这个体系结构是可伸缩的，并且多个资源适配器可以被同时插入到 J2EE 中。

资源适配器类似于 JDBC 驱动程序。它是一个系统级软件驱动程序，用来连接到一个特定的 EIS。J2EE 服务器与 EIS 协同工作，从而实现事务、安全性与通信，并且这些功能都以无缝、透明的方式完成。为此，有几个合约定义了 J2EE 服务器与 EIS 之间的关系：

- 连接管理合约：提供对底层 EIS 的连接并负责 EIS 连接池功能。
- 事务管理合约：允许在多个资源管理器之间正确地执行事务。
- 安全性合约：结合底层安全性机制，确保安全访问 EIS。

15.4.7 JavaMail

JavaMail 是用来处理邮件消息传递的 Java API。JavaMail API 提供了一个用来对邮件系统建模的抽象类的集合。使用这些 API，就能够以独立于平台、独立于协议的方式访问邮件。第 22 章给出了一个使用 JavaMail 的示例。

JavaMail API 是一个关于电子邮件消息传递的开放框架的规范。它提供了一个可以被不同厂商实现的通用 API。这种一致性能大大缩短开发人员的学习时间并且减少厂商面临的设计困难。

JavaMail API 在 javax.mail 包中提供。这个 API 的顶级表项是 Session 类，用来管理访问邮件系统所需使用的配置选项与用户认证。特别地，它用来控制与访问代表特定厂商实现的邮件服务的类。这些服务的提供者实现了几个类。

Store 类允许访问、读取、写入、监视与搜索邮件。Store 对象包含一个可从其中检索邮件的邮件文件夹对象数据库。Store 实现了这个特定的访问控制（例如 IMAP（Internet Message Access Protocol，Internet 消息访问协议）、POP3（Post Office Protocol，邮局协议）），并且有用来建立连接和关闭连接的方法。

Folder 类提供了一个关于邮件消息以及对封装在 Message 对象中的各条消息进行访问的层次式结构。

Message 类对单独的电子邮件消息的细节进行建模，这些细节包括主题行、发信人、回复地址、收信人以及日期。这个类实现了关于消息的“缓慢”取（也就是说，尽可能地延迟取回）以改进网络性能。JavaMail 支持多部分 Message 对象，其中的每个部分都可以有自己的属性与内容类型（这对于实现带有附件的邮件消息是有用的）。

Transport 类用来在一个特定的协议上发送消息。这里的主要方法是 Transport.send，它实现消息的发送。

15.4.8 Java 消息传递服务

JMS 促进了分布式异步消息传递 (distributed asynchronous messaging)，分布式异步消息传递是构建分布式应用程序的成熟技术。这个领域中处于领先地位的产品就是 IBM 的 MQSeries。JMS 为这些系统提供了一个独立于产品的 Java API，从而允许开发人员构建对消息传递中间件进行应答的应用程序，而不必被束缚在单独的一个实现中。已经在第 14 章详细讨论了 JMS。

15.4.9 Java 命名和目录接口

JNDI 为命名与目录服务提供了一个基于 Java 的 API，主要用来访问 LDAP (Lightweight Directory Access Protocol，轻量级目录访问协议) 目录。本书将在第 20 章对它进行讨论。

15.5 Java 数据库互连

JDBC (Java Database Connectivity，Java 数据库互连) API 为 Java 开发人员提供了以独立于平台的方式连接到关系数据库系统的能力。实际上，JDBC 本身就可以写一本书，因此，本节的目的是让读者对它有一个基本的熟悉，以防今后可能会用到它。

JDBC 定义了一个开放的标准 API。厂商可以提供它们自己对数据库进行访问的“JDBC 驱动程序”。JDBC 标准允许厂商在维护基本的语义的条件下，对 JDBC 驱动程序的实现与功能进行选择。JDBC 驱动程序的实现可以基于 ODBC (Open Database Connectivity，开放数据库互连) (实际上，Sun 公司提供了一个 JDBC-ODBC 桥，从而使得任何符合 ODBC 的数据库可以被 JDBC 所使用)，也可以使用厂商私有的本地代码实现或者使用通过网络连接到数据库的私有代码实现。

除了配置与性能问题之外，开发人员对驱动程序的底层实现通常并不感兴趣。例如，基于 ODBC 的驱动程序通常比厂商的本地代码所实现的驱动程序速度慢，也没有成为应用的主流。

15.5.1 使用 JDBC

JDBC API 包含在 `java.sql` 包中。为了执行一个数据库操作，需要下列步骤：

1. 加载 JDBC 驱动程序。
2. 建立一个数据库连接。
3. 构造一个 SQL 查询或更新语句。
4. 执行这个语句。

5. 关闭连接。

其他动作还包括查询数据库以获取概要信息或获取其他元数据。

加载 JDBC 驱动程序

JDBC 驱动程序由 `java.sql` 包中的 `DriverManager` 类管理。可以用两种方法加载 JDBC 驱动程序：实例化驱动程序或显式地加载这个类。例如，如果要实例化一个 Oracle 驱动程序，应该使用下列语句：

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

如果要显式地加载这个类，使用下列语句：

```
Class.forName("oracle.jdbc.driver.OracleDriver");
```

连接到数据库

使用 `DriverManager.getConnection` 方法获取数据库连接。这个方法使用的参数包括数据库 URL、用户名、口令。例如，假设有一个名为 `testdb` 的 Oracle 数据库放在名为 `dbhost` 的机器上，那么调用这个方法的形式就是：

```
Connection conn = DriverManager.getConnection(
    "jdbc:oracle:thin@dbhost:1521:testdb", "dba", "dbaspass");
```

建立连接可能包括建立网络连接的动作（如果数据库在另一台机器上）。

执行查询

下一步就是使用连接对象构造一个 SQL 语句，其形式如下：

```
Statement statement = con.createStatement();
```

这个语句对象可使用 `executeQuery` 方法执行 SQL 查询。例如：

```
ResultSet rs = statement.executeQuery("SELECT * from MYTABLE");
```

从 `executeQuery` 返回的对象类型为 `ResultSet`。`ResultSet` 本质上是一个数据库表中行的集合。可以使用 `ResultSet.next` 方法对这些行进行迭代。对于每一行，都可以使用 `ResultSet` 对象的 `get` 方法获取各个列上的值。`get` 方法可以使用列的序数，也可以使用列名称。例如，如果第一列称为 `COLUMN1`，并且这个列中保存的是整数值，那么就可以使用下列形式之一获取这个列上的值：

```
int way1 = rs.getInt("COLUMN1"); // By column name
```

或

```
int way2 = rs.getInt(1); // By ordinal
```

类似于 `getInt` 方法，用户也可以使用 `getString`，`getObject` 以及其他 `get` 方法。

执行更新

执行更新类似于执行查询，不同之处是使用了 `fix-width` 对象的一个不同方法。例如：

```
Statement statement = con.createStatement();
statement.executeUpdate("INSERT MYTABLE VALUES (1,2,3)");
```

关闭连接

关闭连接比较简单，只要调用连接对象的 `close` 方法即可：

```
conn.close();
```

虽然这是一个只有一行代码的语句，但这个语句非常重要，因为保持打开连接就需要加载数据库与 JVM。如果总是打开新连接并且忘记关闭它们，那么就可能会出现性能问题乃至可用性问题。

15.5.2 更多的 JDBC 功能

现在读者已经对如何在一个启用 JDBC 的 Java 应用程序中使用数据库有了基本的了解。本节将继续讨论 JDBC 的高级功能选项。

prepared 语句

为了改善应用程序的性能，JDBC 允许使用 `prepared` 语句以相同的方式编写语句（除了相同的参数具有不同的变元之外）。例如，假设需要从一个表中选出其第一列都具有某个值 (`val`) 的所有行。这时，就可以使用下列语句：

```
ResultSet rs = statement.executeQuery("SELECT * from MYTABLE WHERE" +
"COLUMN1 = " + val);
```

但是，这样可能不是一种高效的办法，因为每次都必须重复地重新生成这个 SQL 查询。另一种方法就是构造一条 `prepared` 语句，如下所示：

```
PreparedStatement ps = conn.prepareStatement("SELECT * from MYTABLE WHERE
COLUMN1 = ?");
```

以后，当需要（在一个循环或其他结构中）使用值 `val` 时，就可按下列方法执行查询：

```
ps.setInt(1, val);
ResultSet rs = ps.executeQuery();
```

JDBC 2 ResultSet

JDBC 2（它是 J2SE 中使用的标准）中的 `ResultSet` 接口提供了一些新功能，包括滚动。除了 `next()` 方法之外，还可以使用 `previous()` 方法进行回退，或使用 `first()` 与 `last()` 方法分别转移到第一行和最后一行。

ResultSet 的另一个新功能是 update 方法。应该记得，ResultSet 支持诸如 getInt 这样的方法用以获取值。它也提供了相应的 update 方法，从而能够在结果集之上立即执行更新，而不必使用一条单独的 update 语句。例如：

```
int val = rs.getInt(1);
rs.updateInt(1, val+1);
```

JDBC 事务

JDBC 支持事务 (transaction) 的概念。对于每个连接，在任何给定的时刻只能存在一个事务，不允许出现多个嵌套的事务。但是，客户可能使用多个连接。

第 11 章简要讨论了 EJB 事务，本书将在第 31 章对 JDBC (以及其他) 事务管理进行更详细的讨论。

15.6 小 结

自从考虑将 Java 用于构建应用程序特别是企业级应用程序以来，为了使其发展更成熟并更适合上述应用程序的要求，Java 已经走过了一段很长的路程。目前，Java 主要有 3 条发展轨迹，其中 2 条与 WebSphere 上运行的应用程序相关：标准版 (J2SE) 与企业版 (J2EE)。作为应用程序服务器，WebSphere 主要面向企业市场。它用来构建最经常被多个用户访问的高端系统。因此，WebSphere 作为服务器使用 J2EE 的 API 集合也就不足为怪。正如本章描述的那样，J2EE 是一个定义 API 集合的包集合。J2EE 通常被称为一个标准，这是因为这些 API 的组合定义了一个任何应用程序开发人员都可使用的环境 (无论是从 API 还是从语义方面都是如此)。这就意味着，应用程序开发人员能够重用大量底层服务与需求，从而使得他或她能够编写更少的代码，但同时又确保所重用的代码能够在任何符合 J2EE 规范的服务器上使用。这是软件界以前从来没有达到过的可移植性新层次。

WebSphere 版本 4 是一个完全符合 J2EE 规范的版本，并且这是 WebSphere 发展历程中的重要一步。

本章将结束第 2 部分的讨论，这一部分讨论的是 WebSphere 的技术基础。本书的其余部分将描述如何开发在 J2EE 服务器 (在这里就是 WebSphere) 上运行的应用程序以及如何在服务器上部署这些应用程序，并且提供了在以 WebSphere 为中心的环境下应用程序开发人员所使用的开发工具的概述。其中最后一个问题将是下面两章 (在第 3 部分中) 的讨论话题。

第 3 部分 WebSphere 应用程序开发环境

第 16 章 WebSphere Studio Application Developer

WebSphere Studio 产品家族提供了一个集成的工具集合，可用来开发 Web 站点。它将简化网站设计、开发与维护的任务。

WebSphere Studio 产品家族中的最新成员是 WSAD (WebSphere Studio Application Developer)。WSAD 是 J2EE 应用程序的 IDE (Integrated Development Environment, 集成开发环境)。它囊括了 J2EE 应用程序的构建、测试与部署的整个生命周期。WSAD 吸引人的地方表现在它结合了 Web 设计工具的易用性与 Java IDE (例如 VisualAge for Java) 的强大性。而且，WSAD 还为 WebSphere Application Server 提供了完全的测试环境。

WSAD 代表了 VisualAge for Java 与 WebSphere Studio 家族早期产品的结合。在 WSAD 被引入之前，一些非常严肃的开发工作都是在 VisualAge for Java 中完成的，而小型的开发 (例如将已有的组件集成) 以及 Web 站点设计则是使用 WebSphere Studio 完成的。WSAD 为几乎所有的服务器方组件提供了一个单独的开发环境，这些服务器方组件包括 HTML 文件、XML 文件、JSP、servlet、EJB 以及 Web 服务。WSAD 提供了非常广的功能，使得它对创建 J2EE 应用程序过程中的各类开发人员都存在强大的吸引力，包括从 Web 设计者与 Java 开发人员到部署与管理应用程序的各类人群。通过提供自定义机制，WSAD 能够为充当某一角色的用户分离那些对他或她无用的功能。此外，虽然这个产品提供了大量的功能，但并没有对该工具在性能上的强大性造成损害。

值得一提的是，WSAD 基于称为 Eclipse 的开放源代码软件平台，这一平台由 IBM 领导开发。读者可以在 <http://www.eclipse.org> 上了解更多关于 Eclipse 工程的信息。

16.1 安 装

可以直接从 IBM 的 Web 站点 <http://www.software.ibm.com/websphere/studio> 下载 WSAD 以及其他 WebSphere Studio 产品。也可以在本书的配套光盘上使用 WSAD。与 WebSphere 家族的其他成员相似，WSAD 支持多平台 (在编写本书时，能够支持 Windows (98/Me/2000/XP) 以及 Linux)。

WSAD 的安装过程很简易。如果从 Internet 上下载了 WSAD 的安装版本,那么首先应该解压,然后运行 setup.exe 程序,这个程序将带领用户完成安装的全过程。如果用户从光盘上进行安装,那么可以直接运行 Setup 程序。

安装过程是很标准的。Setup 程序的第一个窗口是欢迎屏幕,然后出现的是许可协议。用户需要选择产品安装目录、主用户角色以及版本控制软件。主用户角色是下列之一:J2EE 开发人员、Web 开发人员、Java 开发人员或 Web 服务开发人员。主用户角色的设置决定了当用户启动 WSAD 时所见到的内容。版本控制软件设置允许用户将 WSAD 集成到已有的版本控制软件中(例如 CVS 或 Rational ClearCase)。在进行这些选择之后,就可以启动安装过程,该过程将需要几分钟的时间。

在启动 WSAD 时,将见到类似于图 16-1 所示的窗口。

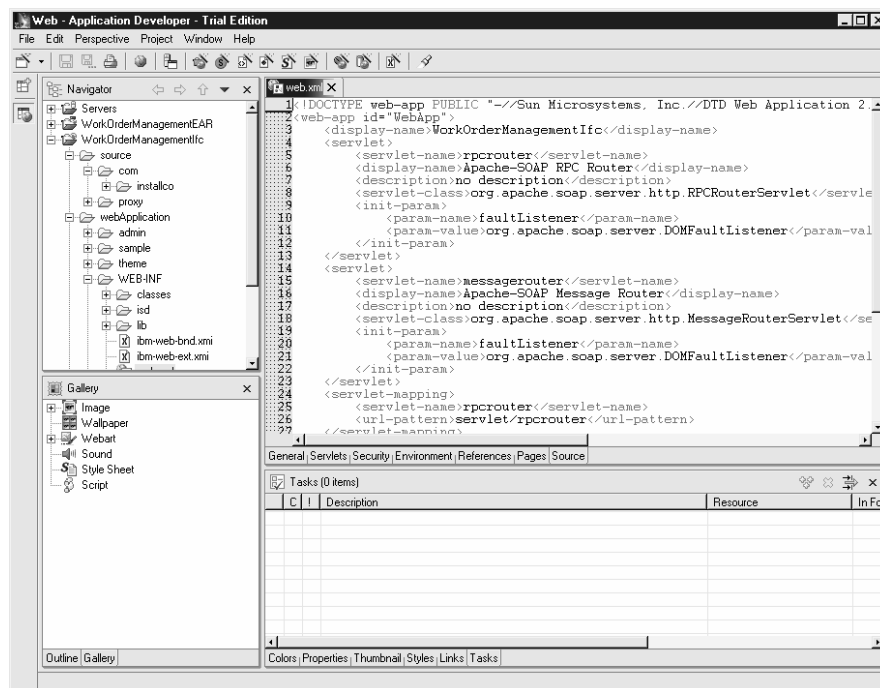


图 16-1 WSAD 的欢迎页面

16.2 WSAD 概览

WSAD 的主用户界面称为 Workbench。Workbench 引入了透视 (perspective) 这一概念,透视就是在某一时刻可用的显示器与编辑器的基于角色的自定义。在工作周期中,用户可以在透视之间进行切换,例如,使用一个透视进行编码,而使用另一个透视进行调试。

透视对相关的视图和编辑器进行组合。视图（view）就是查看某种资源的方法，而编辑器（editor）用来创建与修改代码。WSAD 包括各种编辑器，分别适合特定的文件类型，例如 HTML 文件、XML 文件、Java 资源文件等。可以使用左边的工具栏或 Perspective 菜单在不同的透视之间切换。透视工具栏提供了用来打开新透视的按钮，也为所有当前打开的透视提供了相应的图标。

注意：可以有相同类型的两个已打开透视。例如，可以打开两个 Java 透视，分别用于不同的包。

WSAD 中的透视类型

表 16-1 给出了透视、它们的图标以及描述。每个透视都是可自定义的，并且可以控制所显示的元素以及它们的布局。而且，如果访问一个特定的资源（例如文件），那么就可以使用相应的编辑器对其进行编辑，而不管透视。例如，如果从资源视图或 J2EE 视图双击一个 HTML 文件，那么将出现 HTML 编辑器。

表16-1 WSAD透视

图标	透视	描述
	J2EE	开发 J2EE 应用程序与 EJB 模块
	Web	开发 Web 应用程序或 Web 服务并且设计静态或动态 Web 页面
	Server（服务器）	内置的 WebSphere Application Server 测试环境，可用来测试在其他透视中开发的组件
	Help（帮助）	访问 WSAD 的集成 Help 实用程序，该实用程序提供教程与参考材料
	Resource（资源）	访问文件与文件夹
	Team（组）	访问内置的协作支持——对配置管理系统提供的集成支持，例如 CVS 与 Rational ClearCase
	Java	开发 Java 代码
	Debug（调试）	调试 Java 代码（包括 servlet 与 JSP）
	Data（数据）	访问数据库，从其中导入数据并且执行 SQL 查询
	Scripts（脚本）	编写与查阅脚本
	XML	设计并查看 XML 文件、schema 文件以及文档类型定义
	Java type hierarchy（Java 类型层次）	浏览 Java 类型层次并查看方法与成员变量
	Trace（跟踪）	profile 应用程序与特定的类

(续表)

图标	透视	描述
	Plug-in development (插件开发)	开发用来扩展 WSAD 的 WebSphere Studio Workbench 插件

Workbench 的一个重要组件就是导航窗口，该窗口出现在主窗口的左边窗格。导航窗口提供了一个关于可用资源（工程、文件夹、文件）的层次视图。工程是用来构建、共享、组织资源的管理单元。文件夹是文件的集合，它就是文件系统目录的同义语。文件是底层文件系统的最基本资源。

在 WSAD 中存在 7 类工程，分别是：

- Web 工程：代表 Web 应用程序。这种工程通常包含静态 HTML 文件、图像、Java 文件、servlet 与 JSP。
- EAR 工程：代表企业应用程序。EAR 工程将 EJB 与 Web 应用程序结合起来。
- J2EE 应用程序客户工程：代表 J2EE 应用程序的客户程序。
- Java 工程：代表 Java 应用程序，类似于 VisualAge for Java 中的工程。
- EJB 工程：代表 EJB 组件的集合。
- Server 工程：用来捕获服务器配置信息。
- Plug-in 工程：代表 WebSphere Studio Workbench 插件。

所提供的另一种工程类型是 Example Project（示例工程）。WSAD 包括大量的例子，用来演示 J2EE 应用程序或 Web 应用程序的概念。

WSAD 的另一个重要功能就是其中包含有大量的向导，用来引导用户完成创建新资源或实体的工作。通过选择 File|New，就可以使用这些向导。向导的列表取决于正在工作的透视，但在列表中总包含 Other 表项；如果选择了这个表项，那么用户将见到完全的向导列表。向导、功能强大的编辑器以及调试环境的组合，使 WSAD 成为进行开发的强大工具。面对这么多功能，读者也许会感到有点无所适从。本章的其余部分将帮助读者熟悉某些用来开发 JSP，Java 类与 EJB 的功能。

16.3 开始进行 Web 开发

前面已经说过，WSAD 使用工程来组织文件。实际上，这是开发工具的一个比较标准的概念，这一思想在包括 Microsoft Visual Studio 到任何 Java IDE 的开发工具中都得到应用。工程可能代表 Web 站点，或者代表一个企业应用程序。

1. 为了开始一个新工程，应该选择 File|New Project。这时，将出现一个对话框，该对话框就是新工程向导的第一个窗口。在这个对话框中，选择希望创建的工程类型。

2. 在左边窗格中选择 Web，然后单击 Next。
3. 选择工程名以及希望将工程文件放在的位置。可以指定一个与该工程相关联的企业应用程序。用户可以使用默认的 EAR（也就是默认的企业应用程序档案文件）或定义自己的档案文件。在这个例子中，我们将创建一个称为 Ch16 的新工程，并且将它放在一个称为 Ch16Ear 的新 EAR 文件中。然后，单击 Next。
4. 在图 16-2 所示的窗口中，选择 Java 编译设置，这些设置被组织在 4 个不同的标签中：Source（源文件位置）；Project（相关的工程）；Libraries；Order（控制 classpath 库的顺序）。完成这些设置之后，单击 Finish。

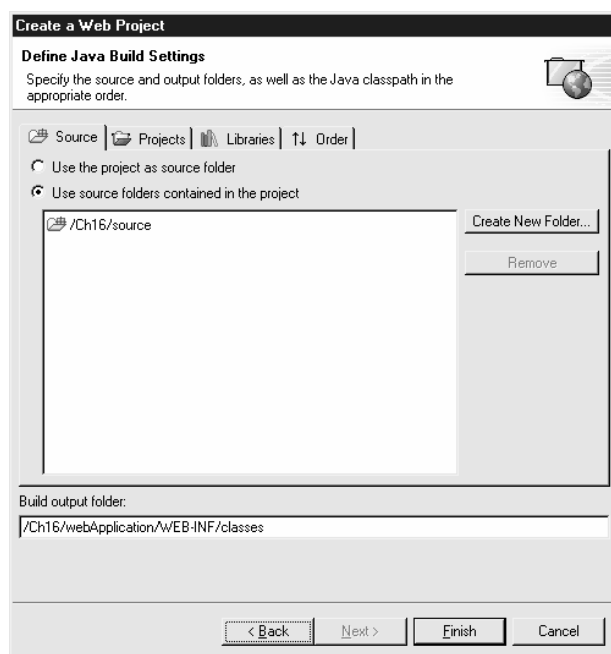
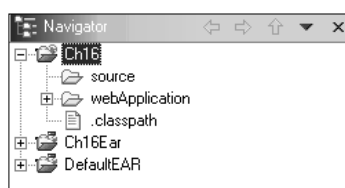
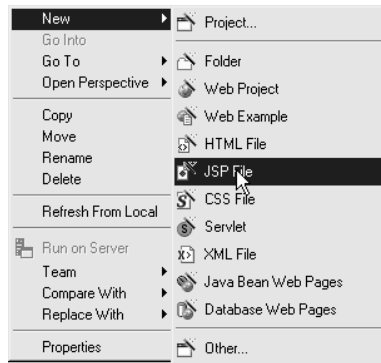


图 16-2 Java 编译设置

在创建新工程之后，它将出现在 Navigator 窗格的左边，如下图所示。所创建的 EAR 文件也显示在这个 Navigator 窗格中。



在 Navigator 窗格上，右击 Ch16 文件夹（或它下面的 source 文件夹），然后从如下图所示的弹出菜单中添加新元素（例如 JSP 文件、HTML 文件或 Web 工程）。



下面将向应用程序中添加 JSP 文件。

1. 当右击 Navigator 窗格中的一个文件夹图标时，将出现一个弹出菜单，从菜单中选择 JSP File 选项。这样，将出现 Create a JSP File 对话框，如图 16-3 所示。

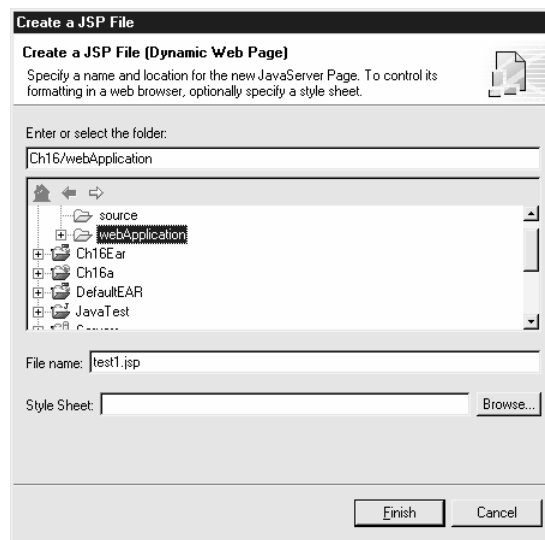


图 16-3 Create a JSP File 对话框

2. 在这个对话框中，可以选择 JSP 文件名以及将要在其中创建 JSP 的文件夹（如果选择了无效的文件夹，那么将在对话框的上面出现一条错误消息）。
3. 在进行这些选择并单击 Finish 按钮之后，将在主工作区（位于 Navigator 窗格的右边）出现这个新的 JSP，其标题为 Place test1.jsp's content here，如图 16-4 所示。

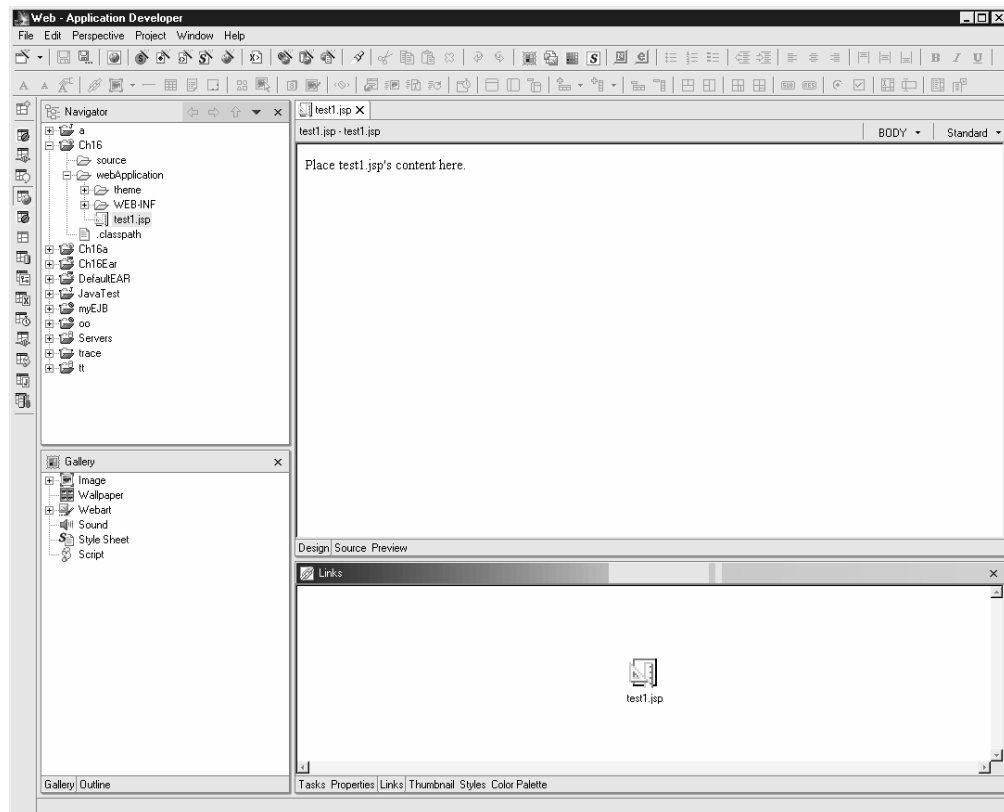


图 16-4 新的 JSP 出现在主工作区

4. 将这些文本替换为 This is a test。需要说明的是，这里的显示是一个 Design 显示，正如在主工作区左下角说明的那样。如果转移到 Source 显示，如图 16-5 所示，那么将可以见到 JSP 源代码。Design 显示用来为页面提供高级视图。可以使用这个显示添加各种元素（例如脚本、图像、表单元素、框架等）。Source 显示给出了底层的源代码，用户也可以直接对其进行修改。Preview 显示给出的是页面的外观。但是，它并不执行服务器方代码（JSP 的）或脚本，因此，它们的效果并没有表现出来。

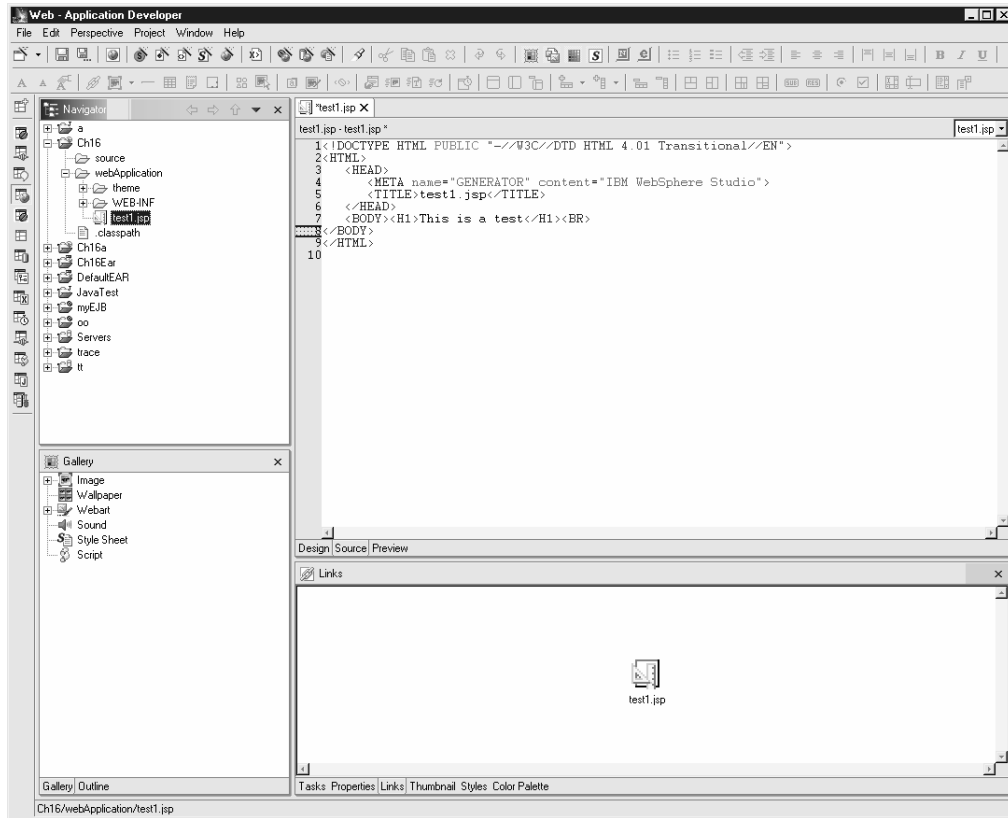
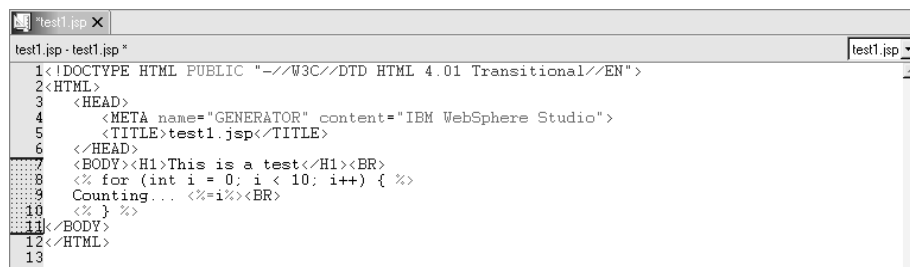


图 16-5 JSP 源代码

为了使这个页面更有趣，可以在其中添加一个小的 for 循环来加强它，如下图所示。



在添加这个 JSP 之后，就可以执行代码。右击 test1.jsp 图标并从弹出的菜单中选择 Run From Server。这时，显示器将显示测试环境，也就是 Server 透视。它显示了一个包含有 WebSphere 服务器消息的控制台窗口。在服务器启动（可能需要几秒钟）之后，就可以见到这个 JSP。Server 透视包括一个浏览器窗口，它显示了运行 JSP 的结果，如图 16-6 所示。

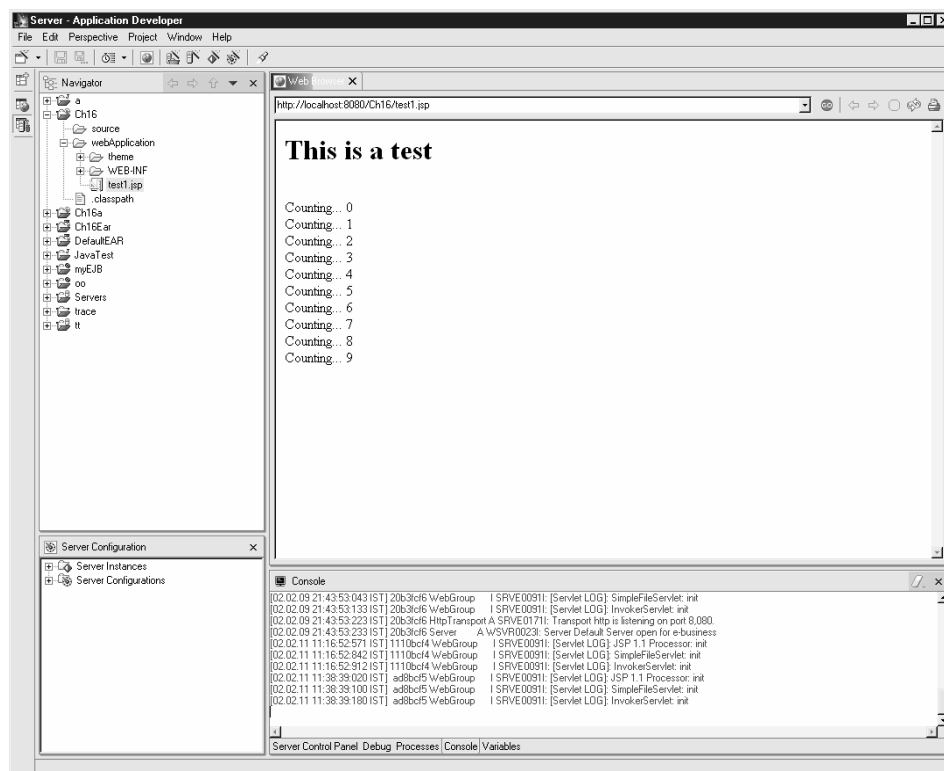
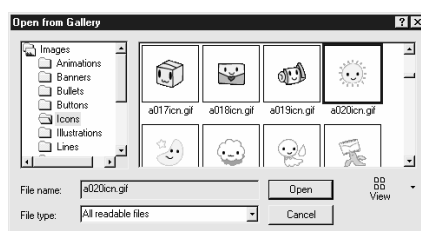


图 16-6 运行 JSP 的结果

为了演示修改这个页面并重新查看该页面的操作简易性，现在再次回到前一个显示（Web 透视），在这里编辑 JSP。可以使用最左边的工具栏或选择 **Perspective|Open|Web** 打开这个显示。

在 Web 透视中，单击 test1.jsp 窗口中的 **Design** 标签（或选择 **Page|Design**），从而选择 Design 视图。将光标放在页面的底部，然后选择 **Insert|Image File|From Gallery**。

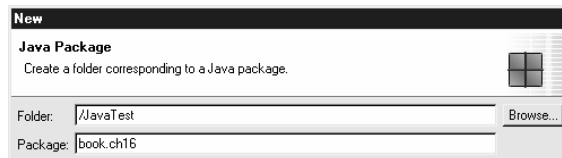
WebSphere Studio Application Developer 包括一个关于图像、样式表以及其他 Web 艺术形式的大集合，被组织在“gallery”（图库）中。为了将一个图像添加到 JSP 源代码中，应该从这个图库中选择一个图像。例如，我们在这里选择了一个太阳图标。



这个选择将立即反映到 Design 显示中。也可以打开 Source 显示，这时将发现一个新图像标记被添加到 JSP 源代码中。为了在测试环境中见到新图标，需要保存这个 JSP 文件并返回到 Server 透视，这时，将显示这个页面的新版本（在刷新浏览器之后）。

16.4 Java 开发

Java 开发主要是通过 Java 透视管理的。在创建一个新 Java 工程之后，就可以开始在其中添加 Java 组件。例如，可以添加一个新包，如下图所示。



在添加一个包之后，就可以添加 Java 类，如图 16-7 所示。

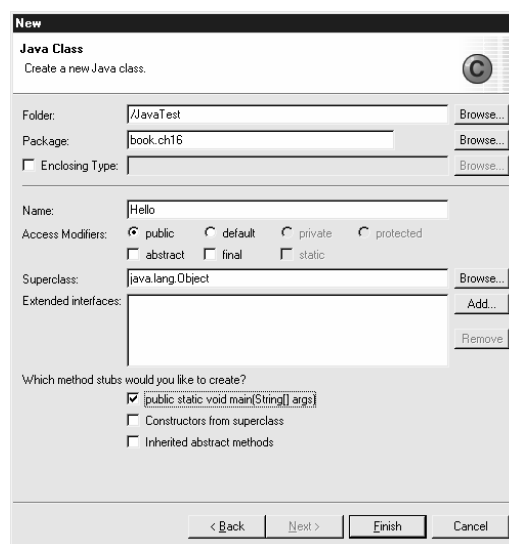


图 16-7 添加 Java 类

在添加这个类之后，将在 Application Developer 中出现一个编辑器，可以在此编辑器中修改这个类的 Java 代码。例如，可以添加一个简单的“Hello World”输出语句，如图 16-8 所示。

在 Application Developer 右下部窗格中显示了这个类以及它所包括的方法。小图标表示成员修饰。例如，附加在 main 图标旁边的 S 表示该成员是静态的，附加在 Hello 图标旁边的运行符号表示这是一个可运行类。成员的可见性是由带颜色的形状来表示的（绿色的圆圈表

示 public，黄色的菱形表示 protected，红色的正方形表示 private)。附加在成员旁边的字母 F 表示 final 修饰。

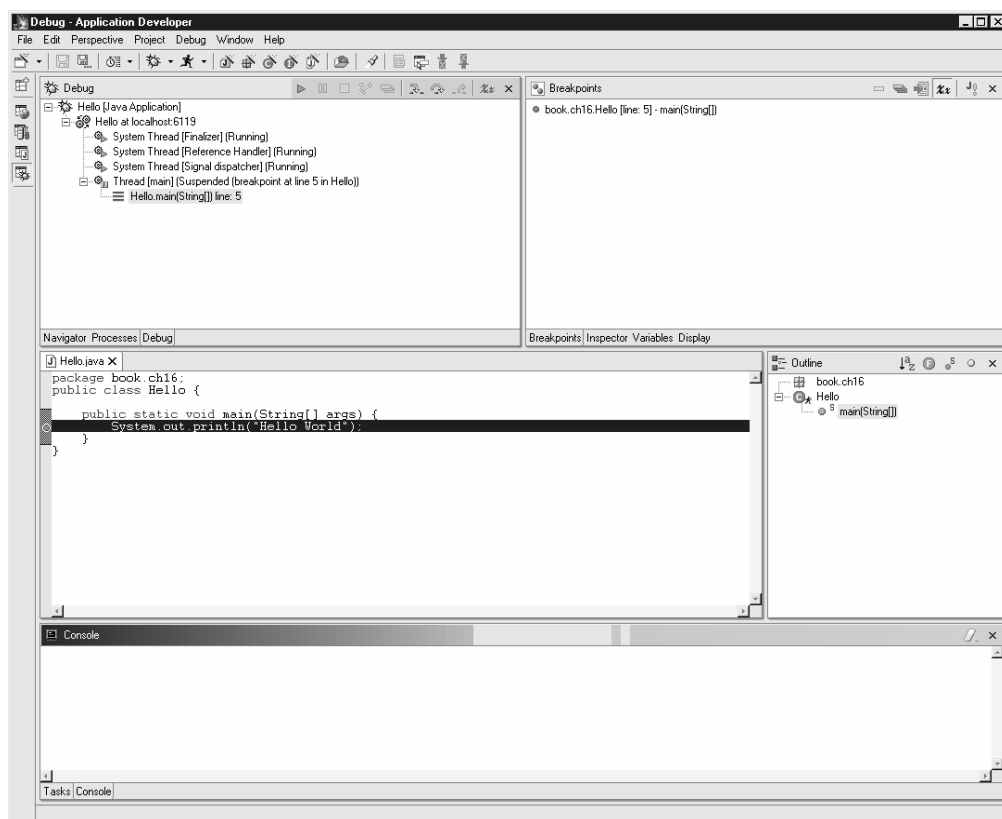


图 16-8 添加一条“Hello World”语句

下面开始讨论调试问题。为了添加一个断点，请右击断点所在的位置并从弹出菜单中选择 Add|Breakpoint。这里在 Hello 类中的打印输出行添加一个断点。选择 Project|Build All（以确保这个类被编译）；然后，就可以使用工具栏中各自的菜单运行或编译。例如，如果调试 Hello 类，那么它将在所请求的断点停下来。

注意，在进行调试时，系统将立刻切换到 Debug 透视，在这里用户可以检查变量与表达式，并查看和编辑被调试代码。也可以查看正在运行的进程。

在功能方面，VisualAge for Java 与 WSAD 提供了几乎相同的功能集合。但一个关键的不同在于，后者提供了功能强大的工具用于 Web 设计与制作。从可用性角度来看，主要的不同在于，VisualAge for Java 为各种目的（例如 Console, Debugger 等等）打开新窗口，而 WSAD 有一个主窗口，用户在不同的透视之间切换。虽然用户可能需要花费一定的时间来习惯这种工作方式，但 WSAD 提供了一种更好的用户界面理念——当需要并发地处理许多不同组

件时更是如此。

注意：在 WSAD 中，调试功能可以与 Server 透视一同被调用。换句话说，可以将一个断点设置于正在内置服务器中运行的代码上，然后，用户可以像调试任何其他代码一样进行调试。

16.5 EJB 开发

EJB 开发是使用 J2EE 透视完成的。为了添加一个新 EJB，通过选择 File|New 调用 New EJB 向导。

在如图 16-9 所示的 Create EJB 对话框中，在两个选项之间进行选择：创建一个新 EJB 或从仓库中添加 bean。为了创建一个新 EJB，首先需要指定 bean 名称。然后，需要选择 bean 的类型——Session，Entity with BMP（Bean Managed Persistence，Bean 管理的持续性）或 Entity with CMP（Container Managed Persistence，容器管理的持续性）。其他参数还包括新 bean 所属的 EJB 对象、bean 类的包、本地接口与远程接口、bean 类的实际名称、（可选的）bean 超类与超类型。

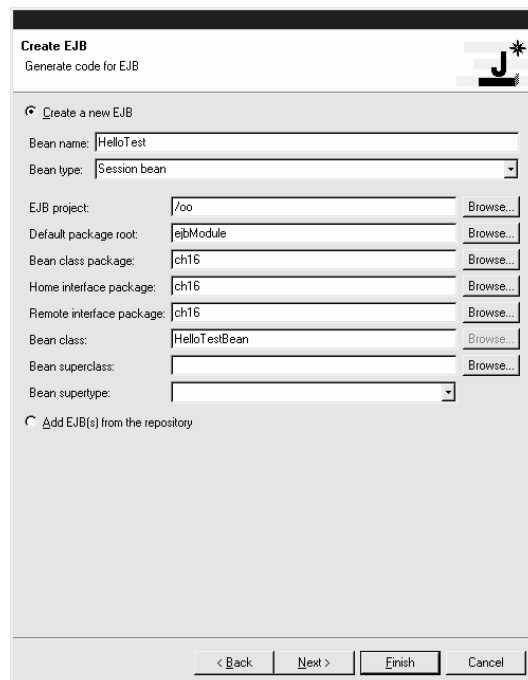
The image shows the 'Create EJB' dialog box in WebSphere Studio. It has a title bar 'Create EJB' and a subtitle 'Generate code for EJB'. There are two radio buttons: 'Create a new EJB' (selected) and 'Add EJB(s) from the repository'. Under 'Create a new EJB', there are several text fields and dropdown menus: 'Bean name' (HelloTest), 'Bean type' (Session bean), 'EJB project' (/oo), 'Default package root' (ejbModule), 'Bean class package' (ch16), 'Home interface package' (ch16), 'Remote interface package' (ch16), 'Bean class' (HelloTestBean), 'Bean superclass' (empty), and 'Bean supertype' (empty). Each field has a 'Browse...' button next to it. At the bottom, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'.

图 16-9 创建一个新 EJB

在单击 Next 按钮后，将见到一个窗口，该窗口允许用户选择本地接口名、远程接口名与密钥接口名（仅仅可用于实体 bean）。也可以选择远程接口应该扩展的接口，以及希望在 bean 实现中导入哪些语句。对于 CMP bean，也可以选择持续性字段。

单击 Finish 按钮，这时就创建了一个新 EJB（在图 16-10 中显示的例子）。向导将自动创建所有必要的类，以及在这些类中所有必要的方法。例如，bean 实现类定义了所有 EJB 所需的方法，以及保存有 EJB 上下文的成员变量和用来设置与获取这些变量的方法。

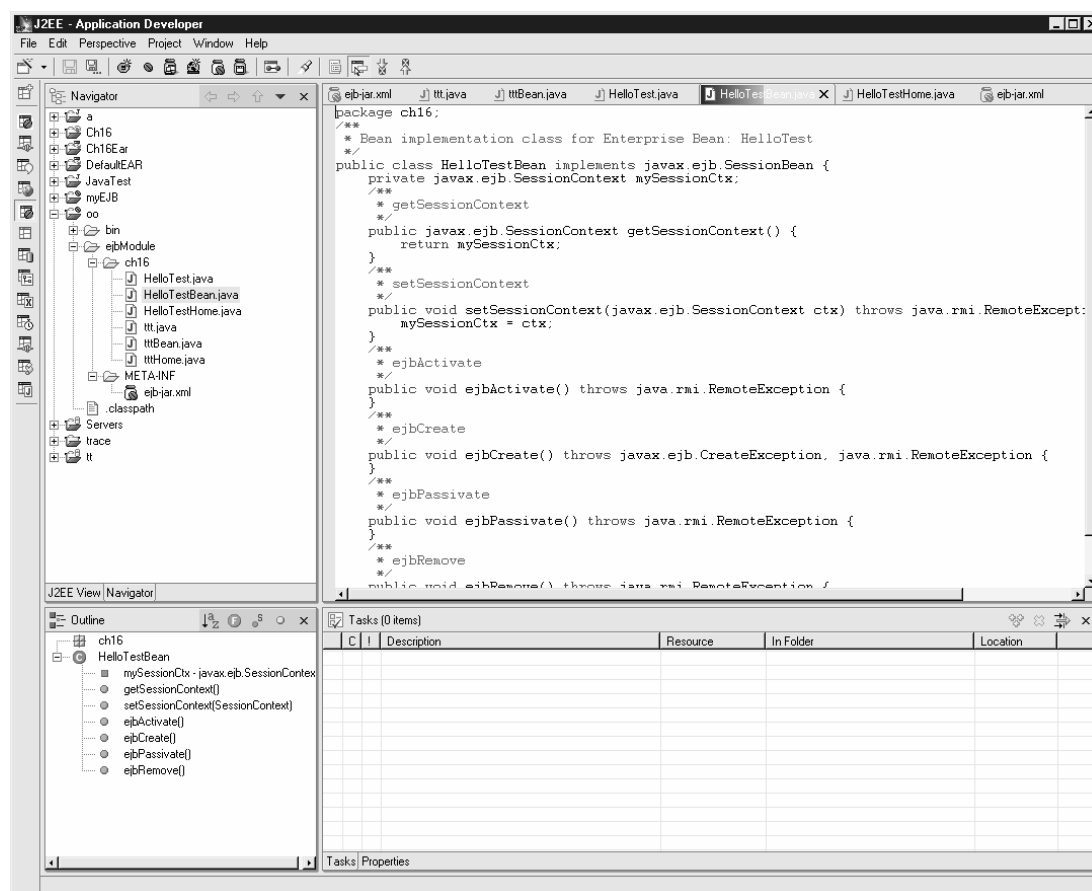


图 16-10 一个可供使用的新 EJB

当使用 Application Developer 添加新 EJB 时，用户 EJB 模块中的 ejb-jar.xml 文件将不断变大并包括关于该 EJB 模块的所有必要的定义。在 Application Developer 的 Bean 标签上，可以查看所定义的 bean，如图 16-11 所示。也可以使用属性编辑器修改 bean 的属性；这样就构建了部署描述符 XML。

在 Security 与 Transaction 标签上，可以为 bean 指定所需的事务行为以及安全性行为。这些

行为可以在单个方法上指定。这一功能等价于 AAT 所提供的功能,但它被集成到 Application Developer 工具中,因此易于使用。EJB 事务与安全性将在第 31 章详细讨论。

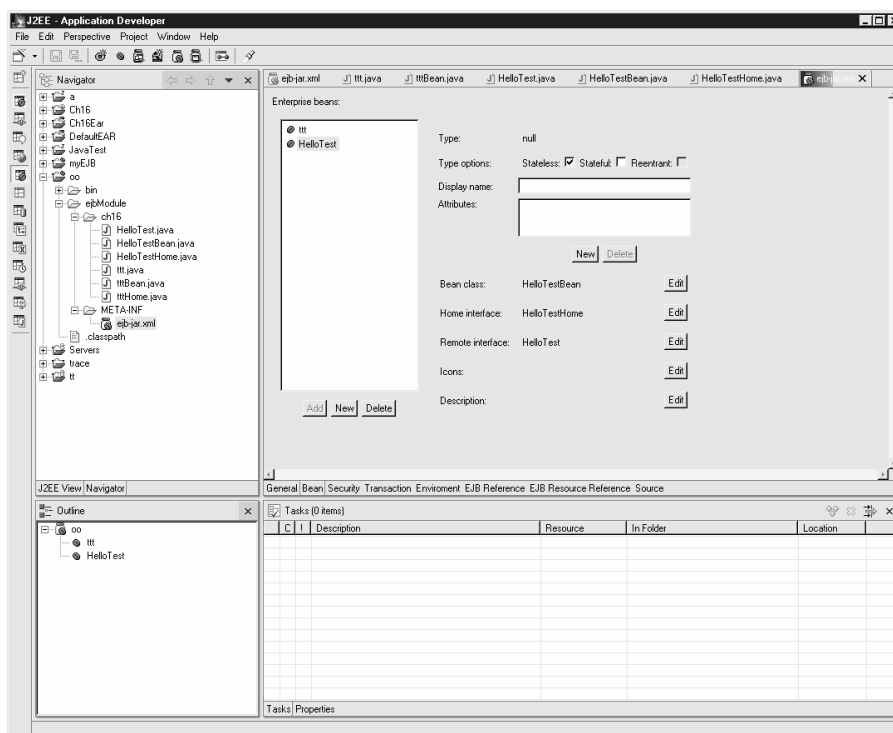


图 16-11 检查已定义的 bean

需要说明的是，在 J2EE 透视下，可以观察 application.xml 文件，该文件定义了一个 J2EE 应用程序。对这个应用程序，用户可以定义作为该应用程序一部分的 EJB 和 Web 模块，也可以为这个应用程序定义安全性。

16.6 小 结

本章在较高的层次上概述了 WebSphere Studio Application Developer。WSAD 是 WebSphere 工具系列中的一个新工具,并且最终将取代 VisualAge for Java 与 WebSphere Studio。可以说,WSAD 是为使用 WebSphere Application Server 的所有网站与应用程序开发人员提供的“一步到位”的解决方案。

读者已经发现，WSAD 基于透视的概念，透视映射到应用程序生命过程中的各个角色。通过本章的例子，读者已经了解了一部分透视，并且已经了解了一些能够使用该工具开发的模块类型。

WSAD 作为一个工具，它覆盖了开发周期中的两个维。通过包括这些维，WSAD 提供了一个统一集成的工具来使应用程序开发小组具有更高的生产率。其中第一维就是组角色，第二维就是活动。Web 应用程序与企业应用程序在所包含的组件以及所涉及的用来构建这些应用程序的人员方面可能非常复杂。网站设计者、Java 开发人员、应用程序部署者都是 J2EE 应用程序的开发生命周期中的一部分。它们构成了开发生命周期中的一个维，它完全包含在 WSAD 中。在这一周期中的活动包括开发、装配、调试与打包——它们构成了 WSAD 所支持的第二维。

虽然 WSAD 是 WebSphere 家族中的标志性开发工具，但有许多开发人员仍然使用（可能将继续使用）VisualAge for Java。因此，将在第 17 章讨论 VisualAge for Java。

第 17 章 VisualAge for Java

VisualAge for Java 是 IBM 提供的用来开发 Java 软件的 IDE。本章的目标是让读者熟悉 VisualAge for Java 环境。除了作为 Java 开发的强大工具之外，VisualAge for Java 在用于开发 WebSphere 服务器方代码（例如 servlet，JSP，EJB）时也有许多优点。之所以具有这些优点，是因为 WebSphere 与 VisualAge for Java 之间的紧密集成，从而提供一个特殊的“WebSphere 测试环境”（WebSphere Test Environment）。这个环境允许用户不必在应用程序服务器上部署软件，就能够对该软件进行测试。因此，就有效地缩短了开发-编译-调试周期。VisualAge for Java 还提供了更多的优点，例如内置的配置管理与大量用来添加不同组件的向导。

17.1 安装 VisualAge for Java

VisualAge for Java 有 3 种不同版本：入门版（Entry Edition）、专业版（Professional Edition）、企业版（Enterprise Edition）。为了开发 WebSphere servlet 与 JSP，需要使用专业版或企业版。而为了开发 EJB（本书第 6 部分的话题），则必须使用企业版。只有企业版才加入了版本控制与配置控制功能，并且加入了共享代码仓库。可以从 IBM Web 站点 <http://www.software.ibm.com/ad/vajava> 免费下载入门版，但其他版本则需要购买。

在获得 VisualAge for Java 安装软件之后（在光盘上或从 Web 上下载），就可以运行 setup.exe 程序以启动安装过程。Setup 程序是标准的程序，可以指导用户完成安装过程。安装过程需要解决的一个小问题就是为 Web 浏览器修改 proxy 设置。这一步是必要的，以便允许 Web 浏览器访问在 WebSphere Test Environment 中运行的 servlet 与 JSP。所需的修改将自动发生作用，只有当机器中的 proxy 设置是自动的时（例如，如果这些 proxy 设置由系统管理员集中管理），才需要手工参与。

一旦安装过程完成，就可以从 Windows 开始菜单中启动 VisualAge for Java 以进行测试。这时将见到闪烁的屏幕，几秒钟之后，该屏幕将被 Welcome 屏幕所替换，如右图所示。



在这个窗口中，就可以开始创建一个软件组件。现在，单击 Close 按钮，或选择 Go to the

Workbench，从而访问 VisualAge for Java Workbench，如图 17-1 所示。

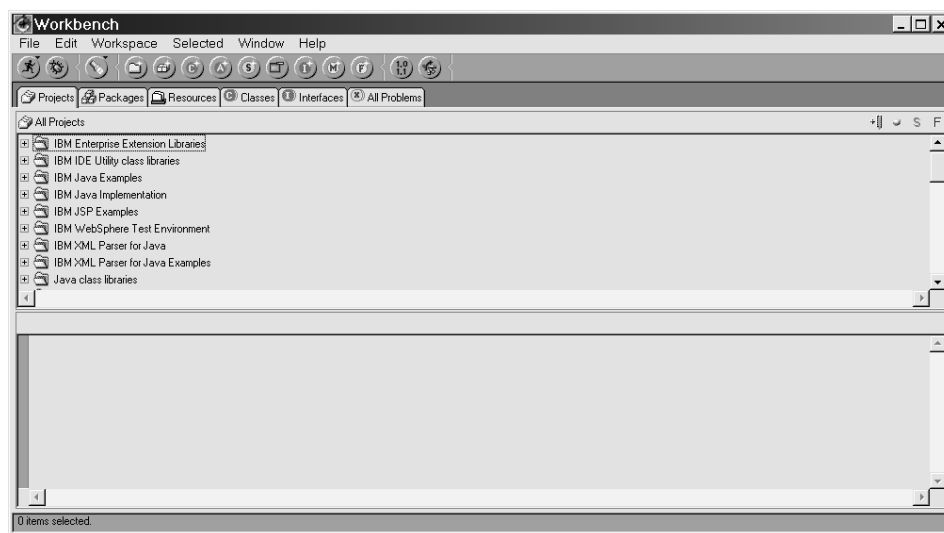
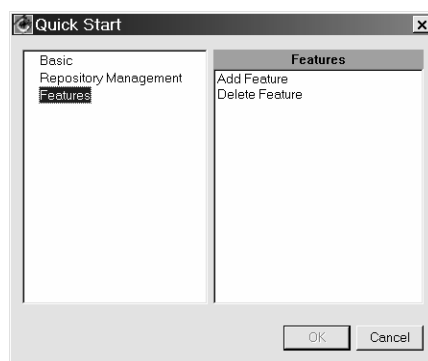


图 17-1 Workbench 是 VisualAge 的主窗口

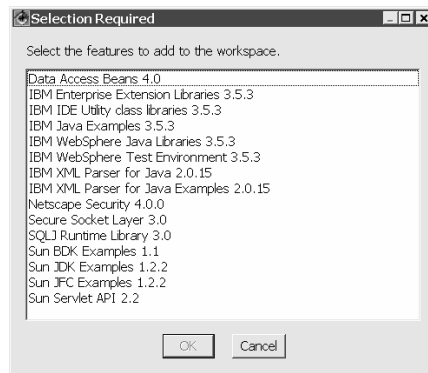
在 Workbench 中需要执行的第一个动作就是在系统中添加新功能。对于使用 VisualAge 开发 servlet, JSP 与 EJB，这是必要的准备（将在第 28 章和第 34 章讨论）。例如，为了在安装中包括 WebSphere Test Environment 功能，就需要添加这个功能。

下面说明了如何向 VisualAge 添加 WebSphere Test Environment 功能：

1. 选择 File|Quick Start。这时将见到 Quick Start 窗口：



2. 从左边的列表中选择 Features，然后从右边的列表中选择 Add Feature。这时，将在 SelectionRequired 对话框中见到一个功能列表：



3. 选择 IBM WebSphere Test Environment 3.5.3 (在这种情况下,也能够添加任何其他功能)。或者,可以按 F2 按钮以添加该功能。然后单击 OK。

VisualAge 有一些非常有用的组件。例如,最近我们需要与 SAP (Service Advertising Protocol, 服务广告协议) 系统进行接口,但我们惊喜地发现, VisualAge 的企业版已经有一个称为 Access Builder for SAP R/3 的功能,这样,我们花了不到一个星期的时间就完成了与 SAP 的接口!

VisualAge for Java 将所有编程数据存储储在两个文件中,分别是:仓库与工作区。仓库(repository)是用户或编程组成员所使用的所有类的中心数据存储地。如果用户单独工作,那么这个文件就位于用户的本地机器上,但在一个典型的工程中,仓库则驻留在另一个共享服务器上,这个服务器被经常备份。一旦某个类被加载到仓库中,它将不会删除,除非显式地请求一个清除动作(purge)。可以从 Repository Explorer 执行这个动作,为了访问 Repository Explorer,应选择 Window|Repository Explorer (实际上,即使如此,仓库也没有真正清除这个类,因此可以进行恢复——除非选择了对仓库进行压缩)。

工作区(workspace)包含当前正在 VisualAge 环境中使用的所有类。当向 VisualAge 添加新功能时,也将加载某些工程与类到工作区中。通常,用户会希望将所有正在使用的类加载到工作区,因此,可以方便地认为工作区就是 CLASSPATH 环境。但是, VisualAge for Java 也有一个工作区 classpath,可以选择 Window|Options 并从出现的 Options 对话框中进行 classpath 的设置。这个 classpath 允许用户引用那些没有加载到工作区中的类。

17.2 应用 VisualAge for Java 创建一个“Hello World” applet

为了熟悉 VisualAge for Java 的使用,下面将添加一个简单的“Hello World” applet。虽然与本书讨论的主题没有直接关系,但创建这个 applet 将演示使用 VisualAge 的简易性。

VisualAge for Java 提供了用来添加新组件的向导集合,从而帮助加速开发过程。虽然这一

过程看上去似乎比较长，但它确实能够节省大量的时间，因为它自动实现了 applet 中所需的大多数通用功能的代码。

1. 选择 File|Quick Start。
2. 在 Quick Start 窗口中，从左边的列表选择 Basic。
3. 在右边给出的组件列表中，选择 Create Applet。
4. VisualAge 将非常详细地指导用户完成 applet 的创建。从 SmartGuide 对话框中，选择工程名（单击 Browse 进行搜索，或在文本框中输入名称）、包名称、applet 的类名称，如图 17-2 所示。

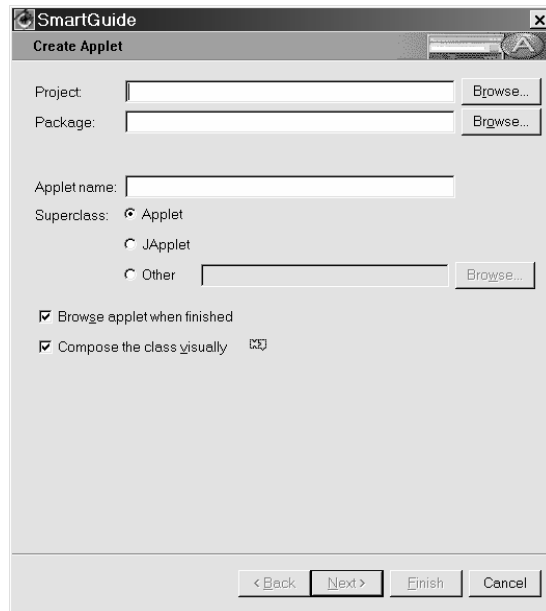


图 17-2 在 SmartGuide 对话框中添加 applet 信息

5. 在相同的屏幕上，选择是否希望在完成时浏览这个 applet，或可视化地构造这个类。单击 Next。
6. 在 Applet Properties 屏幕中，选择 applet 是否实现一个 main() 方法以及该 applet 是否需要在一个线程中运行。单击 Next。
7. 在 Events 屏幕中，选择希望实现的侦听程序。单击 Next。
8. 在 Code Writer 屏幕中，选择是否希望在 applet 中给出样例代码，并且选择将要实现的任何其他方法。单击 Next。
9. 在 Parameters 屏幕中，定义 applet 参数。单击 Next。
10. 在最后一个屏幕（Applet Info）中，如图 17-3 所示，为 getAppletInfo 方法编辑字符串值。然后单击 Finish 按钮。



图 17-3 填写 applet 信息

这样，applet 就准备好了，下面可以“可视化地编辑”它。

1. 在 HelloApplet 窗口左边的工具栏上，可以添加一个标注，如图 17-4 所示。

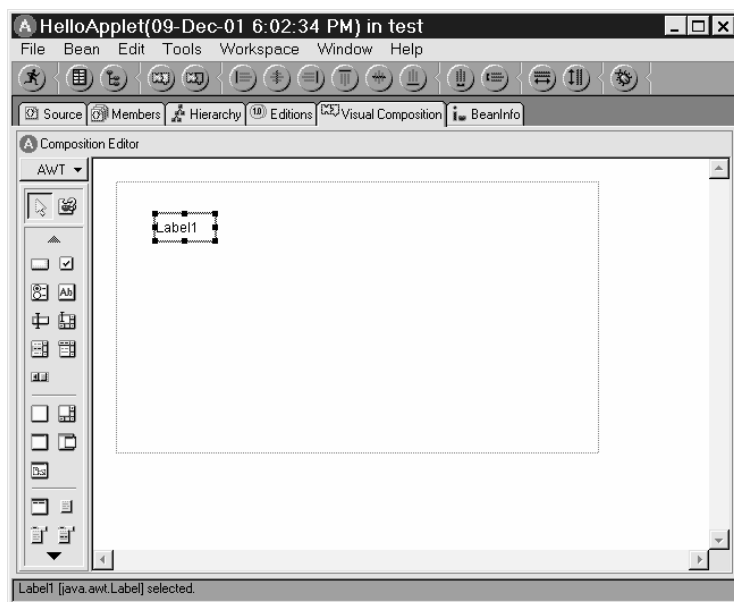


图 17-4 添加一个标注

可以双击这个标注以查看其属性，如下所示：



2. 在标注字符串中添加文本：HelloWorld。使这个字体的大小更大一点，这时，显示将更新以反映新的字体大小。
3. 最后，由于正在使用 IDE 的图形界面，可以“拉伸”这个标注以显示全部文本，如图 17-5 所示。

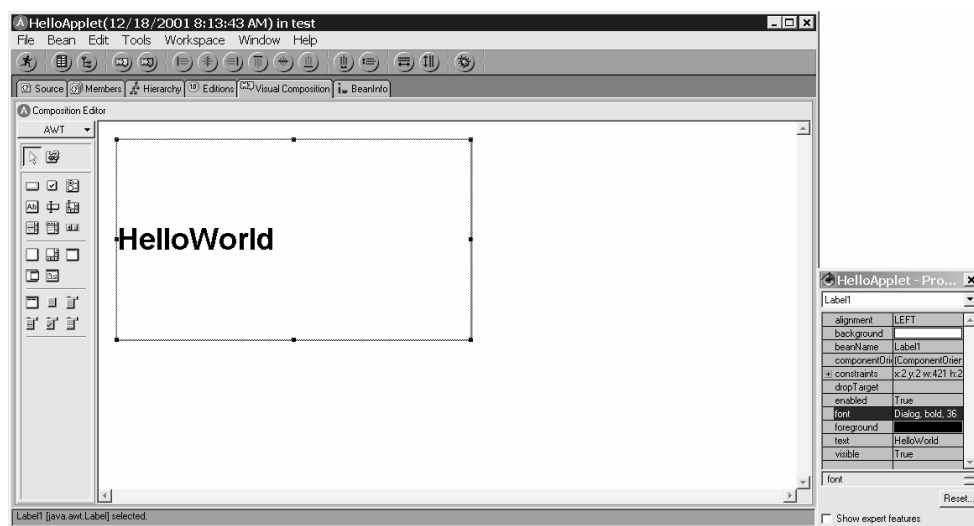


图 17-5 拉伸标注以显示整个文本

17.2.1 VisualAge 窗口

VisualAge 窗口包括几个标签：Source，Members，Hierarchy，Editions，Visual Composition 与 BeanInfo。

单击 Source 标签，就可以检查这个新 applet 的源代码，如图 17-6 所示。

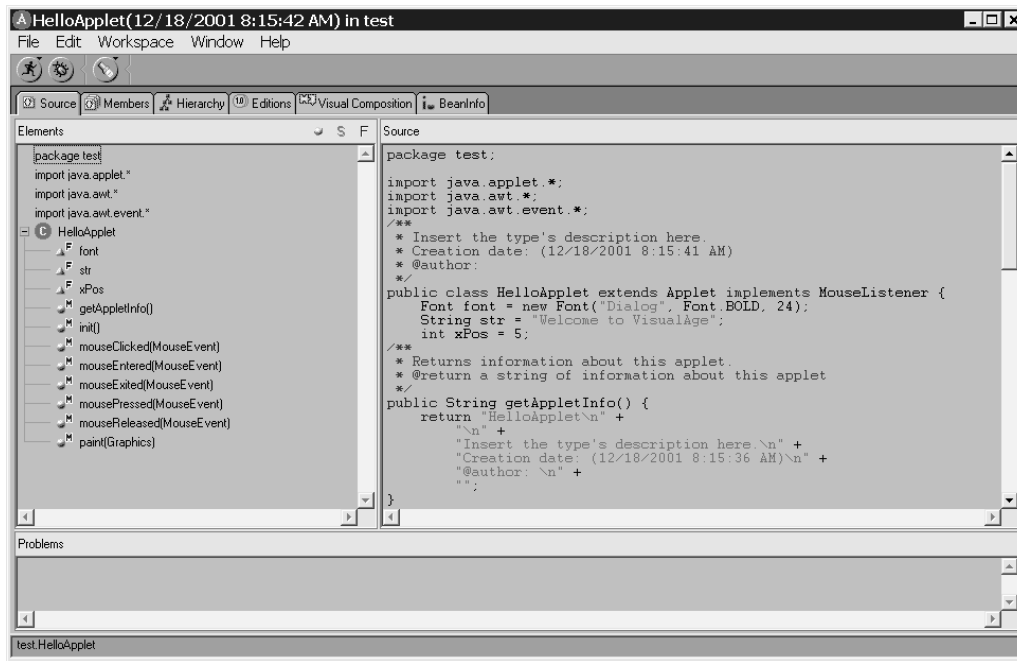


图 17-6 查看新 applet 的源代码

Source 标签左边的 Elements 列表显示了类元素。需要说明的是，每个类成员都被编码为 M（表示它是一个方法）或 F（表示是一个字段）。附加在字母旁边的形状用来表示访问修饰：三角形表示默认访问、正方形表示 private 访问、菱形表示 protected 访问、圆形表示 public 访问。其他修饰也用字母表示（但它们在图 17-6 中不可见）。其中，A 表示 abstract、F 表示 final、N 表示 native、S 表示 static、T 表示 transient、V 表示 volatile。其他可见的表示记号还包括：文件夹符号用来表示一个工程，包图标用来表示一个包，带圆圈的 A 用来表示 applet，带圆圈的 C 用来表示类（class），带圆圈的 I 用来表示一个接口（interface）。

Members 视图（在 Members 标签上）表现的是类成员信息，如图 17-7 所示。

Hierarchy 标签表现的是类层次结构，如图 17-8 所示。在所有这些标签中，可以使用不同的表项点来浏览并查看代码。在 Source 标签上，可以编辑代码。

一个比较好的 VisualAge 编辑方式就是自动完成（automated completion）。如果输入一个实例变量的名称，并且紧跟着出现了一个圆点（.），那么将见到可用于该对象类型的所有可能的方法列表，只要简单地按 CTRL-SPACEBAR 即可。

Editions 标签，在图 17-9 中显示，提供了配置管理（在这个例子中它是一个新 applet，因此没有许多版本被显示出来），并且可视化的编辑窗口与刚刚使用的相同。

最后，在图 17-10 中显示了 BeanInfo 标签，它允许编辑 JavaBean。第 10 章已经快速介绍了 JavaBean。在这个标签中，Features 列表出现在左边；每个功能或者是一个方法，或者是一

个属性（分别由 M 或 P 来区分）。对于每个属性，R 表示它能够被 Get，W 表示它能够被 Set。

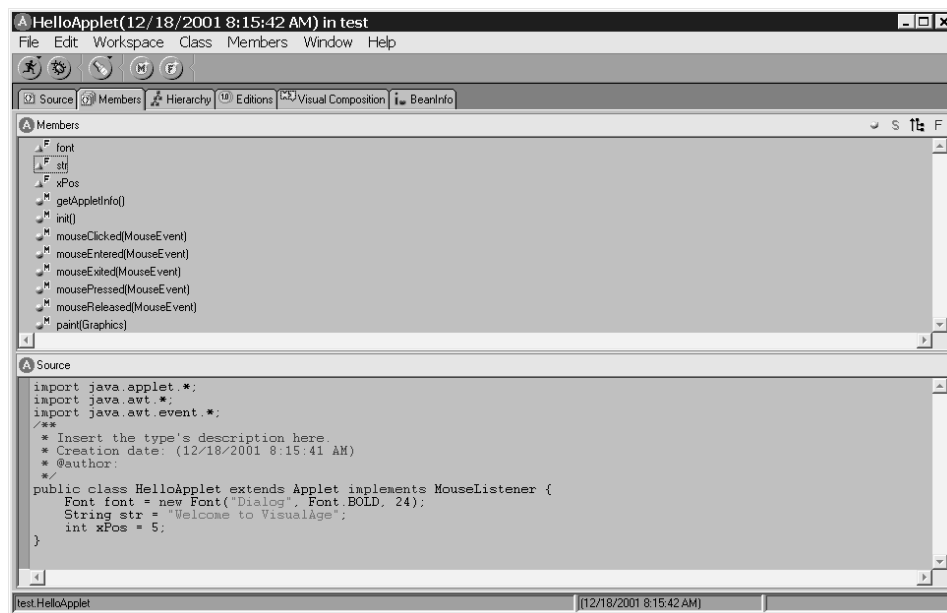


图 17-7 Members 标签

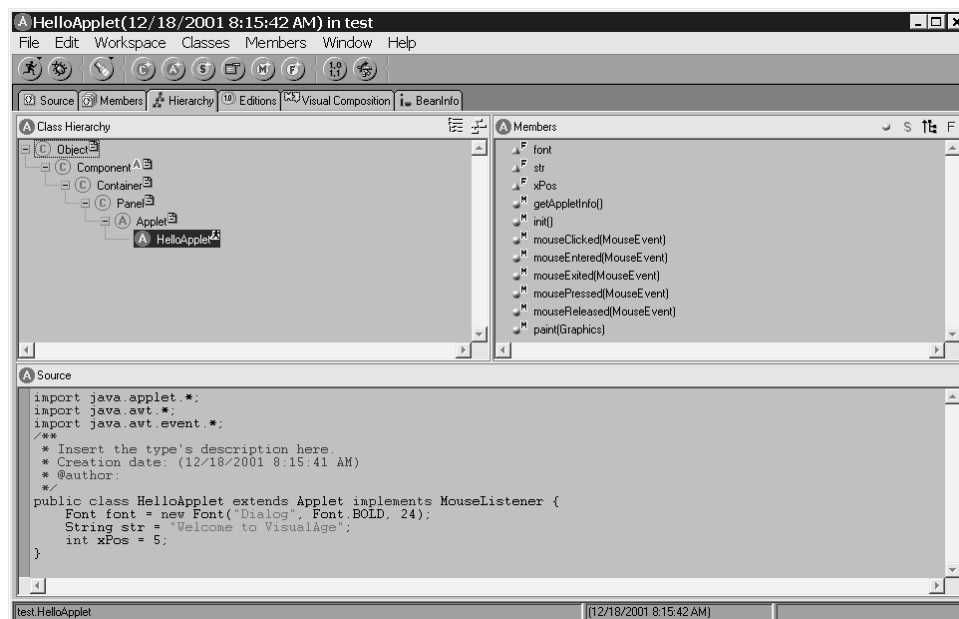


图 17-8 Hierarchy 标签

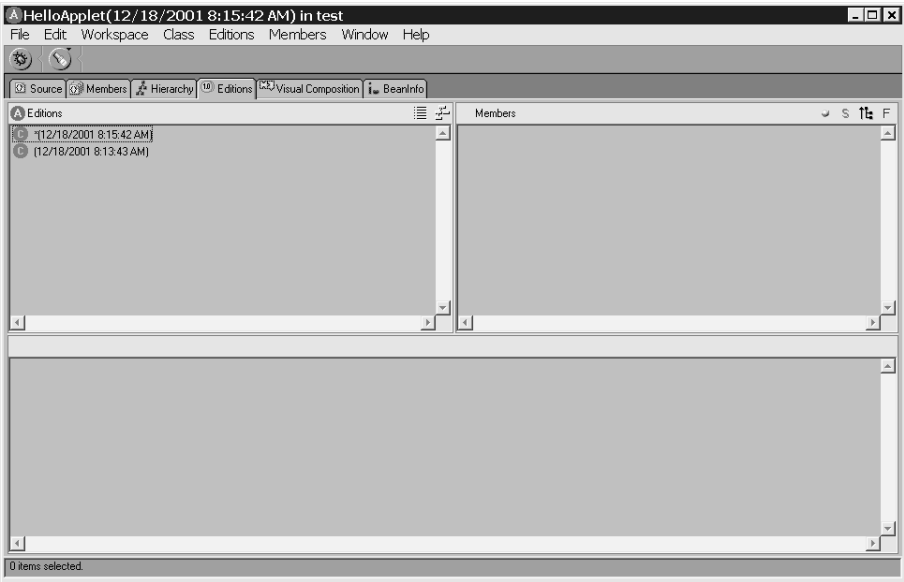


图 17-9 Editions 标签

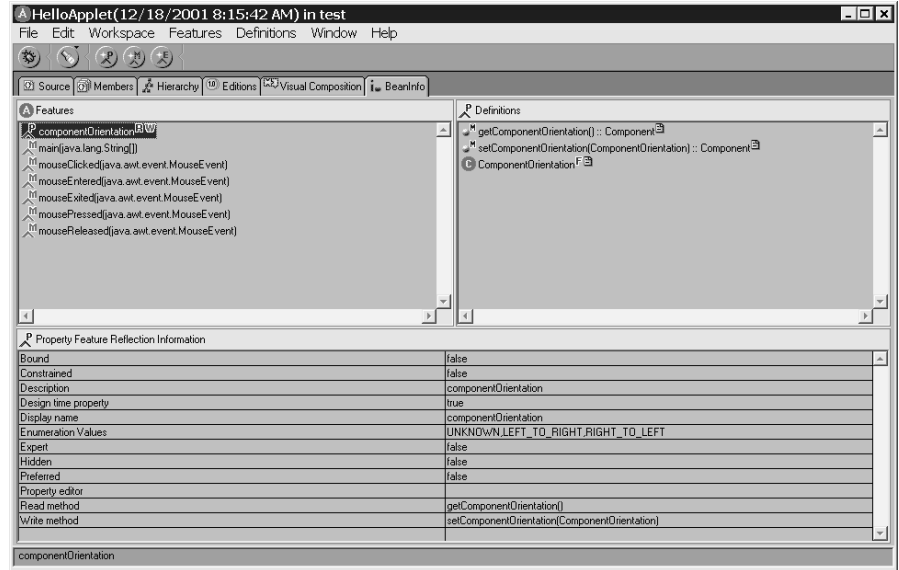
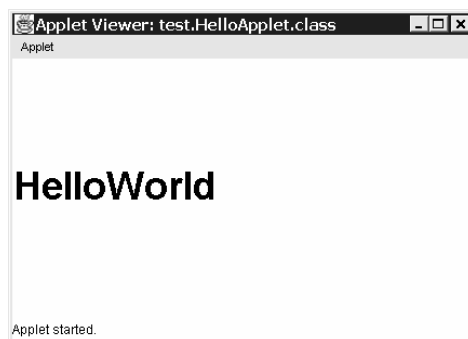


图 17-10 BeanInfo 标签

17.2.2 运行 applet

在 Workbench 中工具条的左边有一个带有小运行图形的按钮；它就是 Run 按钮。单击这个

按钮，将执行该 applet。这时，将见到一个提示，用来告知该 applet 正在运行。然后，将在 applet 查看器中见到这个 applet：



17.3 VisualAge for Java Workbench

图 17-11 所显示的 Workbench 是 VisualAge 开发的中心。这个 Workbench 中的标签提供了关于代码的不同视图——通过工程、包、资源、类与接口。

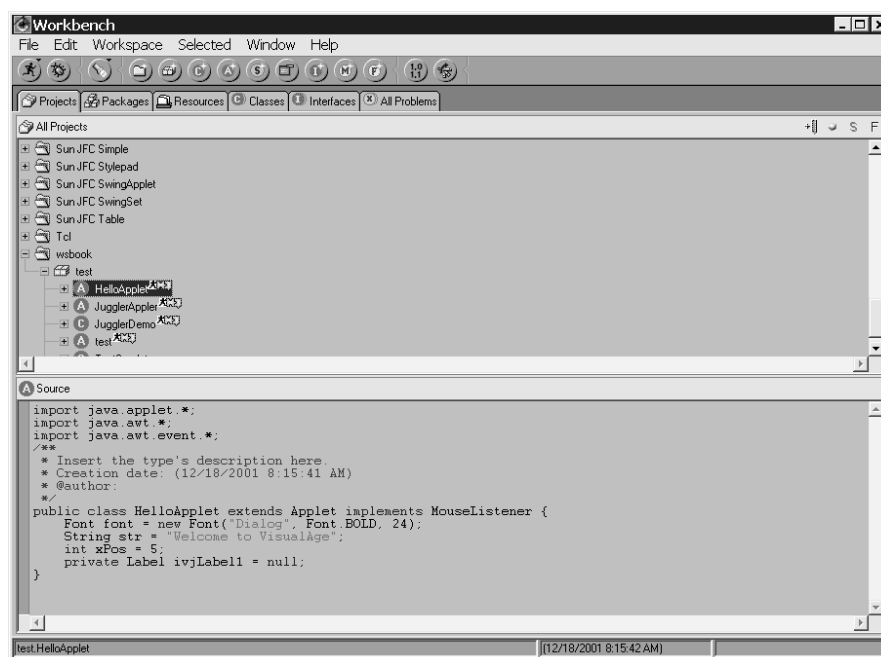


图 17-11 Workbench

All Problems 标签可用来查看与当前代码相关的语法错误以及其他类型的错误（虽然在理论上应该总是能够确保代码中不存在错误）。VisualAge 允许用户甚至在不编译的情况下保存代码，并且将使用一个图标来标记出这个问题，这个图标就是在方法、类或 Workbench 标签上打一个红色的 X。库之间的不兼容性也将作为错误在 Workbench 中标记出来。

与大多数基于 Windows 的应用程序一样，菜单条出现在窗口的顶部，在菜单条下是工具栏；工作区占据了窗口的大部分区域，并且在窗口底部显示了两个状态栏。在工作区中给出的是标签，从而使用户能够在 Projects, Package, Classes, Interfaces 与 All Problems 之间进行切换。当用户在标签控件中进行选择时，菜单条也跟着进行相应的改变。

注意：VisualAge for Java 的企业版还有两个标签，分别是 Managing（用来管理代码所有权（也就是在一个组中进行配置管理））和 EJB（用来管理 Enterprise JavaBean）。Workbench 中的各种标签与菜单允许你浏览代码，以查找关于方法、成员或类的引用。用户也可以添加新的类、方法与字段。一般地说，菜单具有自解释性。另外值得说明的是，许多操作都可以使用不同的标签以多种方式来完成。

下列菜单对所有选择都是固定的。

- File：用于创建文件、加载文件、保存文件与退出。
 - Quick Start：访问不同的创建向导，并且用来为 VisualAge for Java 添加/删除功能。
 - Import：将来自目录、JAR 文件或仓库中的代码插入到 VisualAge 工程中。
 - Export：保存 VisualAge 工程或包到一个目录、JAR 文件或仓库中。
 - Print Setup：设置用于打印的页面选项。
 - Quick Access to Recent Files：列出最近访问的类、包与工程的名称。
 - Save Workspace
 - Exit VisualAge
- Edit：编辑源代码。
 - Revert to Saved：退回到某个文件的已保存版本。
 - Undo, Redo, Cut, Copy, Paste, Select All：执行剪贴板操作。
 - Format Code：修改代码缩进、花括号位置以及其他格式问题。
 - View Reference Help
 - Open on Selection
 - Find/Replace
 - Print Text
 - Save

- Save Replace
- Breakpoint：设置一个断点，用于调试器跟踪。
- Workspace
 - Open Project Browser：通过从列表中进行选择，或者使用搜索字符串来打开一个特定的工程。
 - Open Package Browser：通过从列表中进行选择，或者使用搜索字符串来打开一个特定的包。
 - Open Type Browser：通过从列表中进行选择，或者使用搜索字符串来打开一个特定的类型。
 - Search：在工作区中搜索一个给定的字符串。
 - Text Search/Replace：在工作区中搜索并替换一个字符串。
 - Search Again：访问先前作出的搜索。
 - Search Results：显示搜索结果窗口。
 - Tools：调用工作区工具，例如 Fix Manager 或 SQLJ 工具。出现在这个子菜单中的工具集合取决于在 VisualAge 中安装了哪些功能。例如，如果安装了 SAP Access Builder，那么 Tool 菜单中将包括这个功能。
- Window：窗口处理。
 - Clone：创建当前活跃窗口的一个副本。
 - Switch to：激活一个不同的窗口。
 - Refresh：刷新当前窗口。
 - Lock Window：确保该窗口一直打开；有助于防止意外地关闭该窗口。
 - Maximize Pane：最大化所选择的面板，以占据全部工作区（一旦被选择，这个表项将变为 Restore Pane）。
 - Flip Orientation：将工作区中组件的对齐方式从竖直对齐转换为水平对齐，或者相反。
 - Show Edition Names：显示组件（工程、包、类）的版本名。
 - Workbench：使 Workbench 成为活跃窗口。
 - Scrapbook：打开 Scrapbook 以查看并编辑代码段，包括导入的代码。
 - Repository Explorer：以类似于 Explorer 的方式浏览类仓库。
 - Console：为使用文本 I/O 的程序显示 Java 控制台。
 - Log：显示执行日志窗口与错误日志窗口。
 - Debug：访问调试器、断点与异常。
 - Options：通过多种选项来自定义 VisualAge 的显示与行为。

- Help：联机帮助。

除了这些固定的并且在所有 VisualAge for Java 窗口中都可用的菜单之外，还有一些随着所选择标签不同而不同的菜单。

17.3.1 Projects 标签

Projects 标签工作区显示了工程层次，它允许用户展开或收缩每个文件夹。当这个标签被选中时，下列菜单将出现在 Workspace 菜单与 Window 菜单之间。

- Selected：对当前所选择的工程进行处理。
 - Open：打开所选择的工程。
 - Open To：打开所选择的工程并进入一个特定的标签——Packages, Classes, Interfaces, Editions, Resources 或 Problems。
 - Go To：通过从列表中进行选择，或者使用搜索字符串，转移到某个 Project, Package 或 Type；添加或处理书签。
 - Add：增加一个新元素：Project, Package, Class, Applet, Application, Interface, Method 或 Field。这个子菜单在不同的上下文中使用，并且并非所有的表项都可应用于所有的实例。例如，在 Projects 标签中出现的 Selected 菜单中，最后两个表项为不可用，因为不能添加方法或字段到一个工程中。
 - Attribute Filters：选择将要被显示的元素（类，方法与成员）应该具有的条件。可以按照可见性（public, protected, private）或其他修饰（final, static 等）进行过滤。
 - Import：将代码导入到选择的工程中。
 - Export：从选择的工程中导出代码。
 - Replace With：将工程的当前版本替换为先前版本或一个不同版本。
 - Delete：删除所选择的工程。
 - Reorganize：复制、移动、重命名所选择的工程。
 - Manage：控制版本名的设置方法。
 - Compare With：将所选择的工程与先前版本（或一个不同版本）进行比较。
 - Run：执行所选择的工程；当在 Applet Viewer 中运行时，将出现可应用的可用表项。在可应用的地方运行 main 方法，并且检查 CLASSPATH 环境变量（以确保它被正确地设置从而代码能够真正运行）。
 - Document：打印出源代码或创建一个 JavaDoc。
 - Tools：访问上面提到的外部工具。
 - Properties：显示所选择工程的属性，包括名称、状态、创建时间与版本。

17.3.2 Packages 标签

在 Packages 标签上，工作区的上半部分被划分为 3 个部分：Packages，Types 与 Members，如图 17-12 所示。如果选择一个包，那么包含在这个包中的所有类型（接口与类）都在中间的面板中显示出来。如果选择了一个特定的类型，那么它的成员将在第三个面板中显示出来，并且代码将出现在工作区的下半部分。

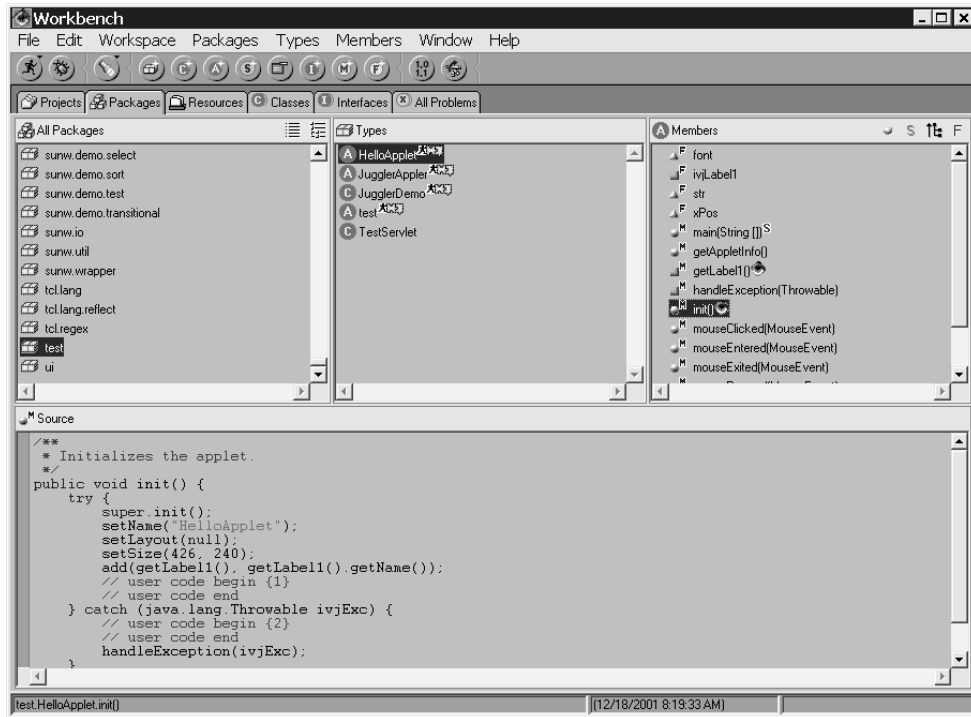


图 17-12 Packages 标签的 3 个部分

Types 区域说明了类型是接口（用 I 标记）还是类（用 C 标记）。出现在名称旁边的图标用来说明该类是否可运行（用正在运行的图标表示），并且说明它是否能够被可视化编辑（用 puzzle 图标表示——类似于一个小人头像的图标）。类似地，在 Members 区域，一个小的 M 表示方法，F 表示字段，出现在名称尾部的图标显示访问模式（如果不是 public）。

当 Packages 标签被选中时，菜单条将在 Workspace 与 Window 菜单之间显示 Packages、Types 与 Members 菜单。这些菜单提供的命令类似于工程的 Selected 菜单，因此这里只介绍它们的不同之处。其中一个不同就是在 Packages 标签的每个菜单上都有一个 Go To 表项（例如 Members 标签上的 Go To Member），该表项特定于从这个菜单可处理的对象类型。Packages 标签菜单有一个 Layout 表项，允许用户在关于包的树布局与列表布局之间进行选择。

Attribute Filters 表项仅仅出现在 Members 标签菜单中。

Types 菜单可用来处理类型。它不同于工程的 Selected 菜单，表现在提供了对类型的特殊支持。Packages 标签的 Types 菜单包括下列特殊表项。

- Generate：为 accessor 以及必要的方法自动生成代码。
- Method Template：创建一个新方法。
- References To：查找对类型的引用，或者对类型中特定字段的引用。
- Externalize Strings：提取出类中硬编码的字符串或作为 JavaBean 属性的字符串，并且将它们输出到资源束（这对于本地化与国际化，以及对于从外部自定义控件的标题非常有用）。
- Tools：在这个子菜单中包括有更多的项目，例如用来生成远程方法调用 stub 的 Generate RMI。

Members 菜单可用来处理菜单。除了标准命令之外，在这个菜单上还提供了下列命令。

- Inheritance Filters：根据它们在继承层次中何处被声明，过滤继承成员与方法的显示。
- References：查阅 Workspace，Hierarchy 或 Working Set（用户定义的 Java 代码元素集合）中所选择的成员。
- Declarations：创建 Workspace，Hierarchy 或 Working Set 中所选择成员的声明。
- References To：列出所选择方法对方法、字段与类型进行的引用（仅仅对方法有效）。
- Declarations Of：列出被所选择的方法声明的方法、字段或类型（仅仅对方法有效）。

17.3.3 Classes, Interfaces 与 All Problems 标签

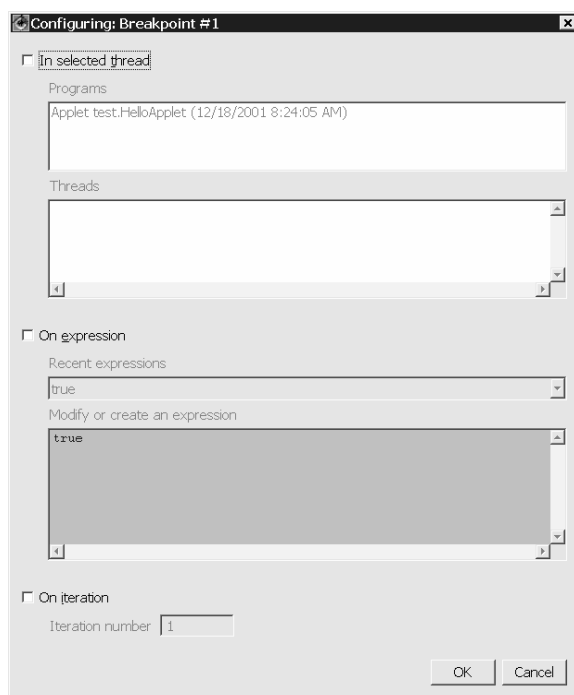
Classes 标签与 Interfaces 标签可用来在不必关心包或工程归属的情况下查看所有的类或接口。在这些标签上，工作区的上半部分被划分为两个部分——左边显示了类或接口的层次，右边显示了方法。变为可用的 Classes 菜单与 Interfaces 菜单，和出现在 Packages 标签中的 Types 菜单几乎相同。

All Problems 标签可用来对所有组件中出现的错误进行快速访问。这是一个关于问题的集中显示视图，也可用来查看特定的包或类的错误。

17.4 使用调试器

VisualAge 调试器简单易用。在 Source 标签上右击源代码，将打开一个弹出菜单，该菜单中包括 Breakpoint 选项。如果选择这个选项，就将为代码添加一个断点。断点用来告知调试器当程序执行到这一位置时，执行过程应该挂起；这时，控制将交给编程者，由他或她按

照需要检查变量值、步进到一个方法中，运行并跳过一个方法，或执行其他解决方案。当选择 Breakpoint 选项时，将打开下列窗口：



在这个窗口中，可以将断点限制在一个特定的线程或程序中，或者，在其中施加一个表达式。甚至可以要求断点在指定次数地经过所选择代码行之后变成有效。

在设置断点并运行代码之后，调试器将自动被调用（如果遇到了一个有效的断点）。Debugger 窗口在下半部分显示了源代码，在上半部分显示了每个活跃线程（以及断点被打开的线程）的方法调用栈，同时在上半部分显示了变量以及变量值，如图 17-13 所示。

调试器提供了标准的功能，例如 Step Into，Step Over 与 Step Out，通过单击 Debugger 窗口顶部的按钮能够访问这些功能。也可以使用这些按钮继续执行、挂起或终止一个线程。在对代码进行单步执行时，将要执行的下一行被高亮显示。

17.5 小 结

VisualAge for Java 是一个优秀的开发环境。自发布以来，我们就一直在使用它。从其第一次发布起，VisualAge 就比所有其他 IDE 在配置管理环境方面高出一筹。在过去的几年中，VisualAge for Java 不断成熟并且（我们认为）一直都是大规模 Java 开发的最佳 Java IDE。

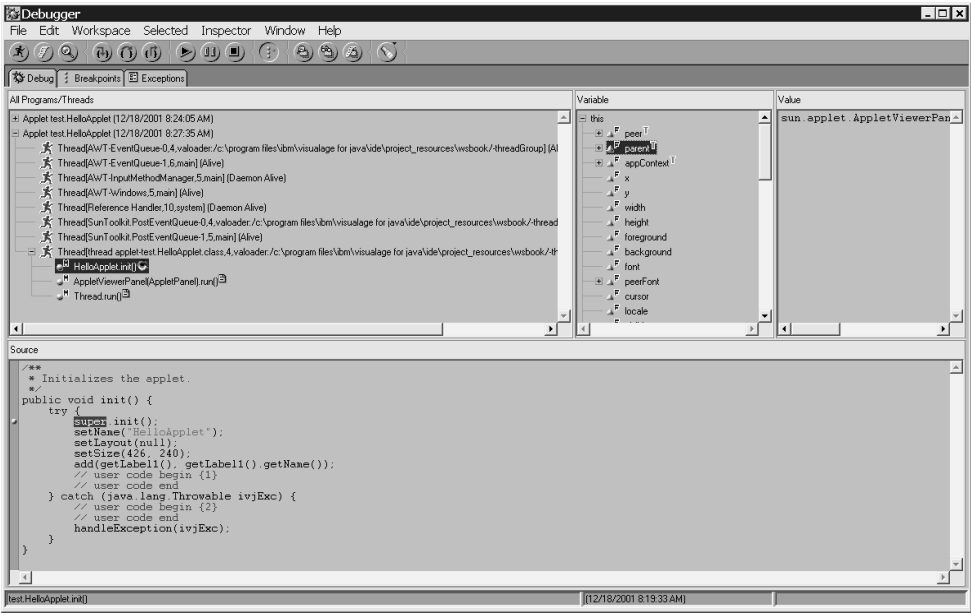


图 17-13 Debugger 窗口

第 4 部分 核 心 服 务

第 18 章 连接池与数据访问 bean

数据访问是任何业务应用程序都必然面临的问题。数据库服务器可能是任何业务应用程序的最重要的组件，它负责数据的持续性存储。在基于 Web 的应用程序中，与数据库的交互也是一个关键问题。无论是搜索引擎、在线购物网站（e-tailer），还是传统的保存有关于客户与供货商记录的业务应用程序，几乎都要求处理大量的信息。

本章关注数据库互连管理问题。Java 应用程序中的数据库互连几乎就是 JDBC（Java Database Connectivity，Java 数据库互连）的同义词，JDBC 是一个 Java 扩展 API，它为编程者提供了基于 SQL 查询的数据访问能力。JDBC 是一个开放的 API，意味着不同厂商可以提供各自的实现（而这又意味着基于 JDBC 的代码可以在数据库之间移植）。

注意：第 15 章简要说明了 JDBC 是什么。关于 JDBC 的详细介绍，不是本书讨论的范围，实际上，有些书是专门介绍 JDBC 的。如果需要了解更多关于 JDBC 的信息，可以访问 Sun 的 JDBC Web 页面，网址为 <http://java.sun.com/products/jdbc>。

但是，通过 JDBC 连接访问数据库服务器的能力，对于产品级应用程序还是不够的。数据库连接是一种必须由应用程序服务器底层结构管理的昂贵资源。本章讨论的第一个话题就是连接池，连接池的目标就是改善依赖于数据库的 Java 服务器方代码（例如 servlet）的性能与可靠性。第二个话题提供了一个关于 JDBC API 的替换方法，试图简化开发人员在构建与开发依赖于数据库的服务器方代码时所涉及的工作，从而提高编程者的效率。这是通过一个称为数据访问 bean（data access bean）的 JavaBean 集合来实现的。

18.1 连 接 池

开发 WebSphere 应用程序（或基于 Web 的应用程序）的一个最常见范型（也是本书反复讨论的一个话题）就是三层应用程序：

- 第一层负责用户界面与表示，它实际上就是 Web 浏览器。
- 第二（中间）层是 Web 服务器（或应用程序服务器），负责应用程序的业务逻辑。
- 第三层负责管理应用程序所使用的数据。这一层可以是关系数据库系统（例如 Oracle 或 DB2），也可以是能够进行数据管理的其他系统。

在三层应用程序结构被引入之前，大多数应用程序体系结构都是客户-服务器体系结构，在这种体系结构下，客户负责控制用户界面与大多数应用程序逻辑。客户通常打开一条与数据库服务器的连接，并且在客户的整个生命过程中维护该连接一直为打开状态。此外，如果在客户上出现某些意外情况（例如 bug），那么并不会产生什么严重的后果，因为用户通常会杀死该应用程序，而这时数据库连接将关闭（当用户重新启动应用程序时，数据库连接又再次打开）。

在三层模型（特别是 Web 应用程序模型）下，必须管理数据库连接。数据库是从应用程序服务器访问的。应用程序服务器被许多用户所使用，并且在用户与数据库连接之间没有直接映射。实际上，客户-服务器体系结构与应用程序-服务器体系结构之间的一个主要区别就是它们对数据库访问方式的不同。Web 应用程序能够非常轻松地成千上万个用户提供服务，某些应用程序甚至为上百万个用户服务。

这时，数据库连接显然不能与每个用户相关联；必须要求有一个可伸缩性更好的模型。在这个模型下，数据库连接可以被许多用户所重用，并且该数据库的连接由应用程序服务器集中管理。连接不再像客户-服务器模型中那样属于客户。当某个客户需要访问数据时，它请求并获取一个连接，然后使用这个连接一段时间，最后将这个连接返回给连接池。

在这种三层体系结构中存在的主要问题之一就是中间层与第三层之间的通信开销。其问题就是“连接到”数据库是一个相对比较昂贵的操作。在非基于 Web 的环境下，这并不会导致什么问题，因为用户登录到系统进行自己的工作，并不介意等待几秒钟来建立连接。这一过程通常只发生一次，因为只要客户正在运行，连接就一直保持打开。相反，基于 Web 的应用程序需要处理大量的用户请求，并且这些用户的请求模式不具有可预测性，并且访问通常要求在非常短的交互中完成。如果需要为每个操作创建新数据库连接，那是很麻烦的。

而且，基于 Web 的应用程序常常是无状态的。即使为 HTTP 请求建立了一个 TCP/IP 连接，该连接也将在响应之后关闭。这就意味着，服务器方代码并不能知道最终用户是否需要在 1 秒钟、1 小时之后（还是永远不会）启动下一个交互。如果中间层为每个 HTTP 请求初始化一条连接，那么就可能出现用在连接数据库与断开数据库上的时间比实际接收数据的时间更长。

由于性能与可伸缩性是 Web 应用程序的关键问题（用户的耐心是很容易消失的），因此 WebSphere 包括了一个内置的连接池功能。池（pooling）本质上是一种资源共享形式。连接池不需要每个用户的请求都经历一个连接操作的开销，因为连接池是预先准备好的（或按照需要动态创建的），并且该连接池由容器（Web 容器或 EJB 容器）管理。连接池被容器中

所有元素共享。这样，servlet 就不等待数据库连接。如果一个连接可用，那么该连接立刻就处于准备状态。如果没有连接可用，那么将创建一个新连接。这时将会增加由于建立连接所带来的一些开销，但在后来的请求中，就又有有一个连接添加到连接池中。在最坏的情况下（就是当到达连接池允许的最大连接数时），正在请求的 servlet 就需要等待正在使用连接的 servlet 释放连接并将该连接返回到连接池中。由于大多数数据库交互都是很短的，因此这个等待时间不会很长。同样，连接池也避免了断开连接操作的开销，因为连接是一直打开的。

池曾一度被认为在数据库服务器开销方面存在优点。这是因为池允许数据库资源被重用，它常常允许建立很少数目的连接（或并发用户）。应用程序服务器厂商之所以把它鼓吹为一个优点，是因为数据库服务器通常是基于并发用户的数量进行许可的。随着三层体系结构的流行，池的发展趋势导致了数据库服务器许可方式的转变，使得基于每 CPU 付费或基于每功率单位付费。

WebSphere 中的连接池机制也对 prepared 语句进行 cache。prepared 语句与一条连接相关联；当应用程序请求 prepared 语句时，池机制将确定该语句是否已经被 cache。由于创建新 prepared 语句这一操作代价相对较高，因此 cache 机制能够改善那些反复创建相同 prepared 语句的应用程序的性能。

18.1.1 使用 JDBC 获取连接

在 WebSphere 中，池机制的实现对正在使用的 servlet 是透明的。实际上，这是 JDBC 2.0 Optional Package API 的一部分。在这里提供了一个称为 DataSource 的抽象用来获取连接，这个对象隐藏了池的细节。本质上，DataSource 是一个连接的工厂。它不同于 JDBC 2.0 Core API（以及 JDBC 1.0）处理连接的方法，在后者中，连接是使用 JDBC 驱动程序管理器的 getConnection 方法建立的（并且相同的连接在用完之后需要显式地关闭）。下面的例子演示了如何使用 JDBC 2.0 Core 建立连接（不采用任何连接池机制）：

```
package book.ws.ch18;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.sql.*;
import java.text.*;

public class JDBCServlet extends HttpServlet {

    // Other methods
```



```

public void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = new PrintWriter (response.getOutputStream());
    out.println("<HTML>")
    out.println("<HEAD><TITLE> Table </TITLE></HEAD>")
    out.println("<BODY>");
    try {
        Class.forName("com.ibm.db2.jdbc.app.DBDriver");
        Connection con = DriverManager.getConnection(
            "jdbc:db2:myDb:5001",
            "db2admin",
            "db2passwd");
        Statement stmt = con.createStatement();
        ResultSet rs = stmt.executeQuery("select * from Tbl")
        while (rs.next()) {
            // Assume we have two columns in the table.
            out.println("<P >" + rs.getString(1) + "," +
                rs.getString(2));
        }
        con.close();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    out.close();
}
}

```

18.1.2 连接池编码

JDBC 2.0 Optional Package API 是 JDBC 2.0 Core API 的扩展。这一扩展的目标是支持某些最新的 Java 扩展，例如 JNDI（将在第 20 章讨论）以及 EJB。JDBC 2.0 Optional Package 还通过 DataSource 接口提供了对连接池的支持。查看该接口用法的最佳方法就是通过一个例子来说明。下面将返回到前面演示的那个例子，但在这里将使用 DataSource（也就是对连接池提供支持）：

```

package book.ws.ch18;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.sql.*;

```

```
import java.text.*;

public class JDBC2Servlet extends HttpServlet {

    // Other methods

    public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = new PrintWriter (response.getOutputStream());
        out.println("<HTML>");
        out.println("<HEAD><TITLE> Table </TITLE></HEAD>");
        out.println("<BODY>");
        try {
            Properties env = new Properties();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.websphere.naming.WsnInitialContextFactory");
            Context ctx = new InitialContext(env);
            DataSource ds =
                (DataSource)ctx.lookup("java:comp/env/jdbc/mydb");
            java.sql.Connection con = ds.getConnection();
            Statement stmt = con.createStatement();
            ResultSet rs = stmt.executeQuery("select * from Tbl")
            while (rs.next()) {
                // Assume we have two columns in the table.
                out.println("<P >" + rs.getString(1) + "," +
                    rs.getString(2));
            }
            con.close();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        out.close();
    }
}
```

为了使用连接池，系统将用 JNDI 查找 DataSource 对象（这与第 5 章中“Hello World”例子中查找 EJB 的方法相同。关于 JNDI 的更多信息，请参见第 20 章）。在这个例子中的数据源名称是 mydb，并且在其前面使用 java:comp/env/jdbc/进行修饰。需要说明的是，也可以将数据源看作 jdbc/mydb，但这个方法主要是为了保持向后兼容而提供的，并且它与 J2EE 1.2 不一致。另外，在这里不存在数据库服务器的直接命名服务；这一操作由连接池处理。

init 方法中执行的代码就是查找 DataSource 对象。为此，应该预先使用 WebSphere

Administrative Console（将在第 42 章讨论）创建 DataSource 对象并存储在命名服务中。为了进行这个查找，需要创建一个 JNDI 命名上下文，并且通过属性容器为它提供参数。需要记住的是，init 方法是执行 JNDI 查找的最佳位置，因为在某个 servlet 实例的生命周期过程中数据源几乎不会改变。这样，在为 HTTP 请求提供服务时，只需要执行一次这个操作，从而节省时间。

作为 doGet 方法的一部分，连接从数据源获取，而不是从驱动程序管理器获取。虽然看上去似乎从数据源获取连接和从驱动程序管理器获取连接是相似的，但这两个选项对应用程序具有不同的意义。如果 DataSource 对象的设置正确，那么它将以透明的方式实现连接池，从而改善性能、可伸缩性与可靠性。虽然在这里对连接调用了 close 方法，但数据源并没有真正从数据库断开，因为可能后来还会使用数据库。总之，其思想就是在 init 方法中访问 DataSource 对象，然后使用该对象来获取连接，隐式地使用池。

在使用从数据源获取的连接时，可能需要处理以下两个异常：

- ConnectionPreemptedException：如果 servlet 在很长的时间里一直没有使用连接，那么将发生该异常。在这种情况下，底层的机制将控制这个连接并更好地使用它。
- ConnectionTimeoutException：当发出一个 getConnection 请求时，如果池为空（也就是在进行这个调用时没有连接可用，并且在指定的超时时间里没有连接被设置为可用），那么将发生该异常。

由于数据库连接是保证应用程序服务器平滑操作的核心可重用资源，因此 WebSphere 不能依赖于应用程序代码必须正确执行这一点。如果某个应用程序代码出现了错误的行为并且没有向池返回连接，那么连接必须被“强制性地”收回。在客户-服务器环境下，客户的错误行为仅仅影响到一个用户。但在三层体系结构中，数据库资源是最重要的共享资源，应用程序代码的错误行为在理论上可能会导致整个服务器消耗所有的资源。因此，服务器必须提供自我保护能力，当服务器判断应用程序代码应该释放连接时，即使应用程序代码没有主动释放连接，服务器也将强制性地收回该连接。在这样的情况下，将发出 ConnectionPreemptedException 异常。WebSphere 关于池的实现有一些自己的异常，将在下一节讨论它们。

需要说明的重要一点是，DataSource 对池的支持并不是标准的一部分；这个支持的实现是有依赖性的。因此，当使用 JDBC 2.0 的其他实现时，用户不能指望该功能一定可用。另外，老版本的 WebSphere 通过连接管理器（connection manager；在 com.ibm.servlet.connmgr 包中实现）实现了自己的连接池机制。在 WebSphere 4.0 中不再支持连接管理器。如果用户有一些老的 WebSphere 应用程序使用了连接管理器，应该说，将它们升级到使用 DataSource 对象这一工作并不难，因为底层的设计模型是相似的，并且可以将先前被称为 IBMConnSpec 的对象替换为 DataSource 对象。

已经讨论过，使用连接池的一个主要动机就是改善性能。因此，设置连接池的大小是非常重要的。使用一个大的连接池能够改善响应时间与可用性，因为这样就提高了 servlet 在请

求时从池中获取自由连接的可能性。另一方面，维护一个大的连接池需要占据大量的内存，这样又可能降低性能。

18.1.3 WebSphere 连接池

WebSphere 对 JDBC 2.0 DataSource 接口的实现提供了一些属性，可用来调整连接池的性能。这些属性是特定于 WebSphere 的，并且与 `com.ibm.websphere.advanced.cm.factory.DataSourceFactory` 相关联。表 18-1 列出了这些属性及其含义。

表18-1 连接池属性

属性	含义
CONN_TIMEOUT	应用程序在超时并发出 <code>ConnectionWaitTimeoutException</code> 异常之前等待连接的最长时间（单位为秒）。默认值是 180 秒。如果将这个值设置为 0，那么将禁用连接超时异常
DATASOURCE_CLASS_NAME	WebSphere 用来连接到数据库的特定厂商 DataSource 对象。这个类应该实现 <code>javax.sql.ConnectionPoolDataSource</code> 或 <code>javax.sql.XADataSource</code>
DISABLE_AUTO_CONN_CLEANUP	用来说明在每个事务结束时，连接是否应该关闭。默认的设置是 false，表示在完成一个事务时连接将返回池（此后，任何试图使用该连接的行为都将导致出现 <code>StaleConnectionException</code> 异常）。将这个值设置为 true 则意味着在事务结束之后连接不返回池，并且应用程序必须调用 <code>close()</code> 显式地返回该连接
ERROR_MAP	这个属性用来将 <code>SQLException</code> 对象映射到 WebSphere 异常；应用程序开发人员很少对它感兴趣
IDLE_TIMEOUT	一个连接能够在池中不被使用的最长时间（单位为秒）。当这段时间结束之后，连接将被关闭并且被清除出池。默认值是 1800 秒。如果将这个值设置为 0，那么就表示禁用超时。需要说明的是，如果从池中删除某个连接将使得池中的总连接数低于 <code>MIN_POOL_SIZE</code> ，那么池中的连接即使超时也不会被删除
MAX_POOL_SIZE	连接池中允许保存的最多连接数。默认值为 10。池的大小对池的效率有直接影响。使用较小的池，将导致请求连接的代码等待时间更长，而使用较大的池将增加开销，因为有更多的连接需要维护。对这个值的设置应该与数据源的能力相对（例如，如果数据库所支持的连接数被限制为 5，那么这个值就必须设置为 5 或更小）

(续表)

属性	含义
MIN_POOL_SIZE	在池中应该存在的最少连接数。默认值为 1。最小池大小可能影响性能。一个小的池需要更少的开销，但当做出了连接的第一个请求之后，响应就较慢。池中开始时并没有达到最小连接数。当为正在请求的应用程序创建连接之后，它们就被放置在连接池中
ORPHAN_TIMEOUT	在应用程序不使用某个连接的情况下，该连接能够被应用程序所拥有的最长时间（单位为秒）。一旦某个连接在这个时间之内没有被使用，那么该连接将被标记为孤立（orphaning）。如果在超时时间结束之后还没有被使用，它将被返回给池。任何后来对该连接的使用都将导致出现 <code>StaleConnectionException</code> 异常。默认值是 1800
STATEMENT_CACHE_SIZE	为池中所有连接准备的 <code>prepared</code> 语句 <code>cache</code> 的大小。默认值为 100

正如表 18-1 所说明的那样，WebSphere 的连接池可发出两种异常：

- `com.ibm.ibm.websphere.ce.cm.StaleConnectionException`：当一个连接由于数据库失败、应用程序关闭连接或连接被孤立而不可用时，将发出这个异常。
- `com.ibm.ejs.cm.pool.ConnectionWaitTimeoutException`：当在经过所定义的秒数之后还没有获取一个连接时，将发出这个异常（这个超时是由 `CONN_TIMEOUT` 属性定义的）。

这两个异常都对 JDBC 的 `SQLException` 进行扩展，因此，可用来处理该异常的 `catch` 语句也可以用来处理 WebSphere 的连接池异常。

18.1.4 连接生命周期

在任何连接池方案（特别是 WebSphere 的连接池方案）中，一个连接可以有 3 个状态之一：不存在、池状态、正在使用。图 18-1 演示了连接的生命周期。

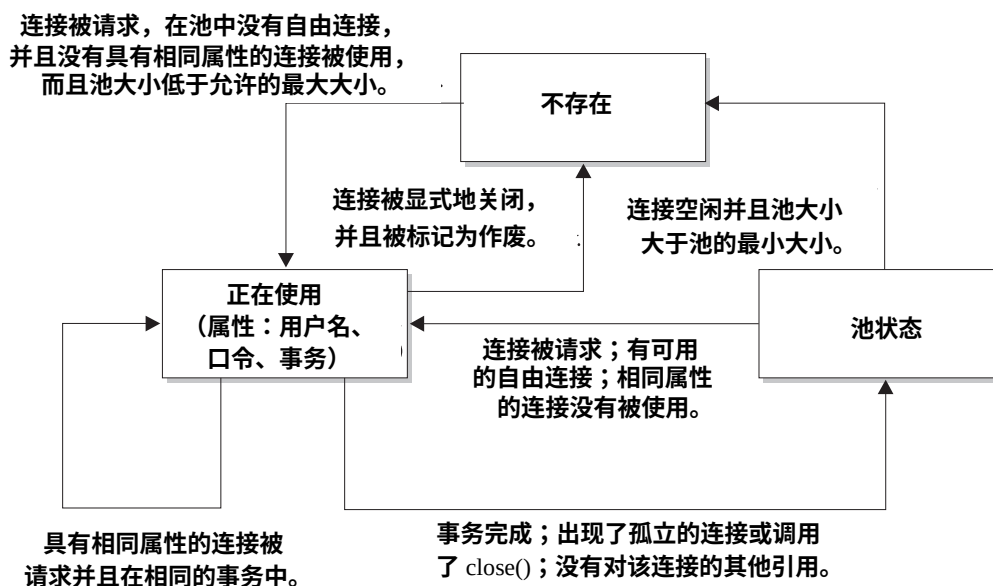


图 18-1 连接的生命周期

18.1.5 连接池的局限性

虽然连接池对于改善依赖于数据库的应用程序的性能非常有用，但它并没有减轻用户管理连接这一工作。特别地，用户必须记住在完成对连接的使用之后关闭所获取的任何连接。不要将这一工作拖延到 `finalize` 方法中进行，因为这并不能确保在垃圾收集器意识到对象不再需要之前就经过了指定的时间。实际上，最好的做法就是在打开连接的相同方法中关闭该连接（一旦数据库访问完成）。

注意：不要忘了将关闭连接动作作为在应用程序代码中处理异常的一部分。

另一个重要问题是，要避免将连接用作静态成员变量。如果这样使用连接，那么就可能出现两个不同的线程使用相同连接的情况，这样就会导致不可预测的结果。相同的要求也同样应用于 `servlet` 实例变量，因为一个单独的 `servlet` 实例被多个线程所使用。读者应该记得有一个相同的 `servlet` 实例被多个客户请求所使用；这样，在 `servlet` 中使用一个连接作为实例变量本质上与在任何其他类中拥有一个静态成员的效果相同。

18.1.6 管理连接池

连接池是通过 WebSphere Administrative Console 管理的。第 42 章给出了这个工具的详细描述，但在这里需要指出的是，连接池的设置与数据源相关联。对于每个 JDBC 数据源，都

可以在 Connection Pooling 标签上设置连接池属性。

18.2 数据访问 bean

本章的其余部分将讨论中间层（Web/应用程序服务器的业务逻辑）与第三层（数据库服务器）之间关系的一个不同方面。为了访问数据源，中间层应用程序代码通常使用 JDBC API（假设数据源符合 JDBC）。数据访问 bean 可以作为 JDBC（也就是所谓的 java.sql 包）的一个替代。

不使用 JDBC 这一想法看上去似乎很奇怪，因为 JDBC 已经成为 Java 数据库互连的事实标准。取代 JDBC 隐含着使用 IBM 的数据访问 bean。JavaBean 编程模型具有多种优点，而不仅仅是在可视化编程过程的 IDE（例如 VisualAge for Java）中能够使用 bean 这一功能。可视化编程减少了需要编写的代码量，提高了开发人员的生产率。数据访问 bean 的其他功能还包括：

- **cache 查询结果：**SQL 查询结果被一次性获取并放在 cache 中。这样就允许 servlet（或应用程序）在 cache 中自由地前后移动，或者跳转到 cache 结果中的任何一行。当在 JDBC 中进行 SQL 查询时，有时是一次获取一个结果，而且只允许执行向前迭代（虽然 JDBC 2 不再如此）。
- **通过 cache 进行更新：**cache 中保存的查询结果可以被处理，以添加、删除或修改行。然后，这些修改可以被写回关系数据库中。当使用这一功能时，用户就不再需要利用 SQL 来更新数据库。
- **参数化查询：**在许多情况下，提交给数据库的 SQL 查询可能包括一些仅仅在运行时才能得到的参数。这样，数据访问 bean 不必显式地在运行时构造这些字符串，它们允许定义基 SQL 查询字符串，并且该字符串可以使用变量来代替某些真正的值。在查询被执行之前，这些变量将被相应的值所代替。实际上，基查询就是带有由问号（?）表示的变量元素的普通 SQL 查询。
- **元数据：**将参数化查询的思想更进一步，考虑参数不一定是字符串的情况，而是任意的 Java 对象。元数据规范定义了如何将 Java 对象转换成 SQL 数据类型，反之亦然；这个相同的规范还定义了如何将 SQL 数据类型转换为 Java 类型。元数据使得用户在处理数据库信息时不必花费时间和代码执行类型转换与字符串构造。

现在，数据访问 bean 已经成为 WebSphere 的一部分。自从 4.0 版本以来，这些 bean 都是用 JDBC 2.0 所实现，并且能够充分发挥数据源接口与连接池的优势。

下面给出一个简单的 servlet，用来介绍不同的数据访问 bean。本例使用与上一个 servlet 例子相同的技术初始化数据源。为了得益于连接池，这里也使用 DataSource 对象。数据访问 JavaBean 包含在 com.ibm.db 包中，因此需要导入这个包。此外，还必须确保 classpath 包括

了 databeans.jar 文件, 该文件对相关的类进行了打包 (这个文件位于 WebSphere 安装目录的 lib 目录下)。

```
package book.ws.ch18;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.sql.*;
import java.text.*;
import javax.naming;
import com.ibm.db.*;
import com.ibm.ejs.dbm.jdbcext.*;

public class DataBeanServlet extends HttpServlet {

    private static DataSource ds = null;

    // Other methods

    public void init(ServletConfig js) throws ServletException {
        super.init(js);
        Properties env = new Properties();
        env.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming.WsnInitialContextFactory");
        Context ctx = new InitialContext(env);
        ds = (DataSource) ctx.lookup("java:comp/env/jdbc/mydb"
        )
    public void doGet(HttpServletRequest request, HttpServletResponse
        response) throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = new PrintWriter (response.getOutputStream());

        out.println("<HTML>")
        out.println("<HEAD><TITLE> Table </TITLE></HEAD>");
        out.println("<BODY>");
        try {
            Connection con = ds.getConnection();
            DatabaseConnection dconn = new DatabaseConnection(conn);
            String query = "select PRODUCT, QTY, PRICE from Tbl"+
                "where PRICE >= ?";
            metaData = new StatementMetaData();
            metaData.setSQL(query);
```



```

        metaData.addParameter("minPrice", Integer.class,
                                Types.SMALLINT);
        metaData.addColumn("PRODUCT", String.class, Types.VARCHAR);
        metaData.addColumn("QTY", Integer.class, Types.SMALLINT);
        metaData.addColumn("PRICE", Integer.class, Types.SMALLINT);
        metaData.addTable("Table");

        SelectStatement stmt = new SelectStatement();
        stmt.setConnection(dconn);
        stmt.setMetaData(metaData);
        // Here we set the minPrice parameter.
        // We omit the details of how the price
        // is determined. The point is that it is not fixed.
        selectStatement.setParameter("minPrice", getMinPrice());
        selectStatement.execute();

        SelectResult rs = stmt.getResult();
        while (rs.nextRow()) {
            // Assume we have two columns in the table.
            out.println("<H1> " + (String)rs.getColumnValue("ID")
                        + ", " + (Integer) rs.getColumnValue("QTY")
                        + "</H1>");
        }
        stmt.close();
        dconn.disconnect();
        con.close()
        out.println("</BODY></HTML>");
    } catch (Exception ex) {
        ex.printStackTrace();
    }
    out.close();
}
)

```

这个例子使用名为 `dconn` 的 `DataBaseConnection` bean，在数据访问 bean 层与实际的数据库之间建立链接。这个 bean 封装了一个 JDBC 连接；其中的一些方法可用来连接到数据库、从数据库断开连接、获取/设置事务隔离级别。

然后，创建一个用来为查询存储元数据信息的 `StatementMetaData` bean。这些信息包括 SQL 查询字符串、查询中使用的数据库列、各自的 Java 数据类型与 SQL 数据类型、查询参数（由查询中的问号表示）及其 Java 与 SQL 数据类型、表名称。对于多个参数的情况，这些参数将按照它们在 `StatementMetaData` bean 中被引入的顺序分别与查询字符串的问号相关联。`StatementMetaData` 类允许使用多种方法添加列与参数。

SelectStatement bean 代表 StatementMetaData bean 中定义的参数化查询的实际实例。这时，它将请求连接，请求元数据 bean 并且要求对参数赋值。一旦 SelectStatement bean 的执行方法被调用，那么就使用 SelectResult bean 来提取结果。这个 bean 提供了 nextRow 与 previousRow（在这个例子中没有使用）对结果行进行遍历，并且提供了 getColumnValue 与 setColumnValue 分别用来获取与设置列值。此外，这个 bean 还允许用户以一种在后来才影响数据库的方式对行进行更新。

总之，数据访问 bean 提供了非常好的方法用来对数据库接口进行编程。它们允许对数据库进行高效的遍历，并且允许非常容易地使用在编译时其值未知的参数构造查询。虽然在效率方面，对查询结果的 cache 未必会立刻发生作用（当查询的量非常大时）。在这种情况下，cache 被分为独立读取的块，但这种方式对使用数据访问 bean 的编程者是透明的。

另外要注意的是，数据访问 bean 是 IBM 的专利代码，因此它们不能在 J2EE 服务器之间移植。如果使用它们，就意味着用户必须使用 WebSphere（或其他 IBM 产品），并且在这种情况下不推荐迁移到其他 J2EE 服务器。

18.3 小 结

管理共享资源是任何应用程序服务器的最重要职责之一。数据库资源是任何电子商务应用程序使用的最有价值、最重要的资源之一。因此，管理数据库资源是 WebSphere Application Server 的核心任务之一。

本章说明了使用数据库连接池的必要性并且介绍了数据库连接的生命周期。在此描述了数据库连接池所使用的 API，以及可用来自定义与优化连接池行为的属性。同时，还给出了使用池中的连接时需要处理的异常。

此外，本章还简要讨论了称为数据访问 bean 的 WebSphere 专利技术。这些 bean 使得比利用 JDBC API 进行数据库访问更简单，同时又不损害系统的可伸缩性与管理。当然，某些高级功能只能通过直接使用 JDBC 与连接池得到，但许多应用程序开发人员都能够从使用数据访问 bean 获得实际的好处。

数据库访问管理是 WebSphere Application Server 提供的核心服务之一。下一章将讨论另一个重要的核心服务（应用程序安全性）。

第 19 章 WebSphere 安全性服务

第 15 章讨论了 J2EE 与操作模型。J2EE 的一个主要目标就是定义开发人员能够使用的应用程序模型。这个模型后来被应用程序服务器提供商所采用（例如 IBM 提供的 WebSphere 产品，在这些产品上可以运行应用程序）。当然，它主要强调的是在服务器平台中构造许多底层结构，从而使应用程序开发人员在遵守 API 以及应用程序框架的情况下利用这些功能。

第 15 章重点讨论的是 J2EE 中的许多底层结构，但并没有讨论安全性问题。安全性是一个复杂的话题，目前已经得到越来越多的关注。Web 应用程序一直是安全性问题的焦点，但在 WebSphere 中，也为应用程序开发人员提供了一个较好的安全性模型。第 13 章概括性地讨论了安全性的概念。可以说，以理论概念为背景对于理解 WebSphere 中的某些功能是很重要的。本章说明了 WebSphere 所实现的安全性模型以及 WebSphere 为开发人员提供了什么安全性功能。它主要关注于应用程序装配阶段，在这一阶段定义了许多应用程序安全性 profile。第 44 章将进一步解释如何在部署时将这些功能组织到一起，并且显示安全性是如何在应用程序服务器上被管理的。

19.1 J2EE 安全性模型

WebSphere 4.0 遵守 J2EE 安全性模型。这个模型主要用来处理授权与认证的问题。加密是通过 SSL 与 HTTPS 实现的，它并不直接与 J2EE 模型相关。通过一种独立于应用程序的从角色到方法许可的映射机制来管理授权。J2EE 模型中的认证支持会话，它使用某种登录方法要求用户进行登录。WebSphere 也支持签名功能与非认证访问。

J2EE 安全性最强调的是基于角色以及基于策略的模型，通过该模型，应用程序装配者与应用程序部署者能够控制对应用程序功能的访问。这与 EJB 中首次定义的模型相符合。在 J2EE 应用程序模型中，容器就像管理其他功能一样管理安全性。应用程序开发人员可以使用 J2EE API 集合通过编程的手段控制所有的安全性，或者，可以使用由容器管理与维护的安全性信息。对于后一种情况，影响安全性的实际值（例如访问控制列表）在部署时定义。下面给出了来自 EJB 规范 1.1 版本中的一些定义：

- bean 提供者负责在部署描述符的 security-role-ref 元素中声明企业 bean 代码中使用的所有安全性角色的名称。
- 应用程序装配者能够定义包含在 ejb-jar 文件中的企业 bean 安全性视图。
- 如果 bean 提供者已经使用 security-role-ref 元素声明了任何安全性角色引用，那么应

用程序装配者就必须将 security-role-ref 元素中列出的所有安全性角色引用链接到 security-role 元素中定义的安全性角色。

- 应用程序装配者为每个安全性角色定义方法许可。
- 部署者的工作是，将应用程序装配者指定的安全性视图映射到目标操作环境中安全性域所使用的机制与策略上。

19.2 WebSphere 安全性模型

WebSphere 安全性模型直接基于 J2EE 安全性模型。它使用许多相互交互的安全性组件解决端到端应用程序安全性问题，以向应用程序提供端到端安全性服务。图 19-1 显示了这些组件。

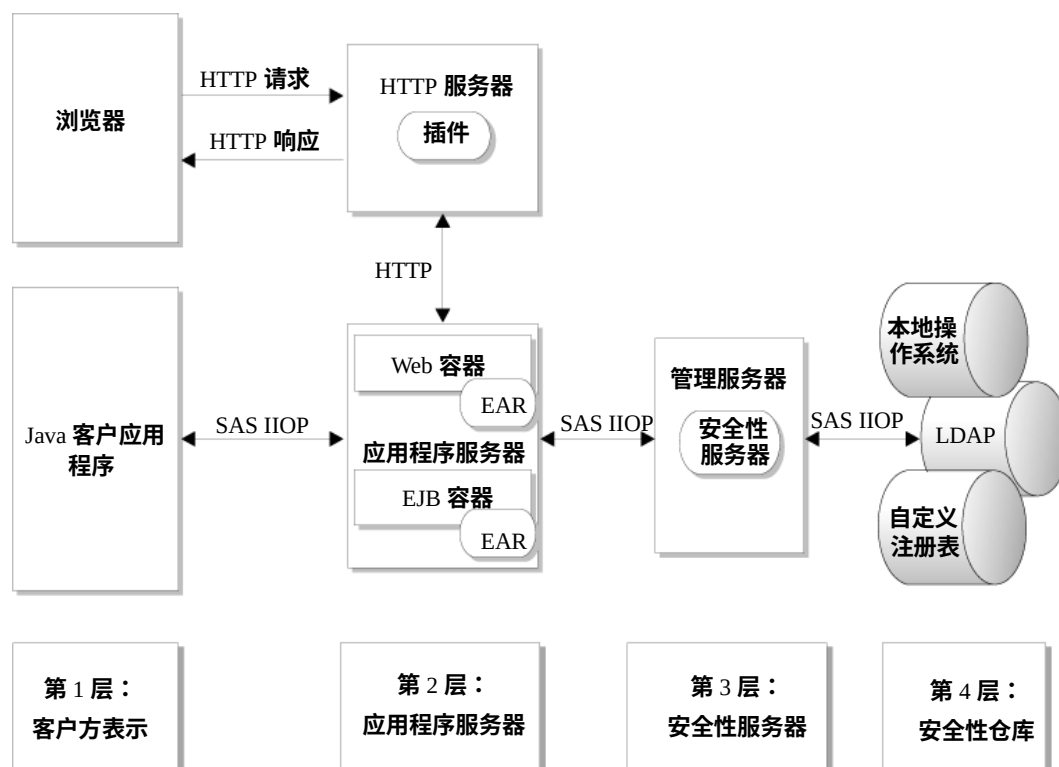


图 19-1 Web 安全性模型组件

请求通过 Web 服务器（如果客户是浏览器）或直接通过 IIOP 到达应用程序服务器。安全性模型以一致的方式支持这两种访问路径。如果请求是通过 Web 服务器到达的，那么它可以使用 HTTP 或 HTTPS。如果使用 HTTPS，则会话利用 SSL 加密，并且认证可通过内置在

SSL 中的机制实现，这已经在第 13 章描述过。在这种情况下，如果客户已经通过认证，那么客户信息就对应用程序服务器可用。

如果客户是一个 Java 客户，那么请求直接通过 IIOP 到达应用程序服务器与 EJB 容器。正如图 19-1 显示的那样，这个会话在 SAS (Secure Association Service, 安全性关联服务) 上运行。SAS 在客户与服务器双方运行。当客户需要从服务器获取信息时，它首先通过 SAS。SAS 在向服务器发送请求之前，将客户的安全性身份信息附加到请求上并且将它们作为请求的安全性上下文的一部分。在服务器方，SAS 从到达的请求提取出安全性上下文，对客户身份进行认证并且将该信息传递给 EJB 容器。这时，EJB 将直接处理授权。

任何资源不仅可通过 EJB 加密，也可以通过 WebSphere 加密。这些资源包括驻留在 Web 服务器上的资源以及那些不是直接由应用程序服务器所管理的资源。一个例子就是作为 Web 应用程序一部分的 HTML 资源。安全性插件被加载到 Web 服务器中，并且该插件使用安全性服务器中的服务保护 Web 资源。

容器使用在应用程序服务器上运行的名为安全性合作者的组件。安全性合作者 (security collaborator) 负责基于从安全性服务器获取的信息，为容器提供与安全性相关的服务。安全性合作者既为 EJB 容器提供服务，也为 Web 容器提供服务，并且在应用程序服务器上可能存在多个安全性合作者。合作者由安全性策略管理，读者将在后面了解到如何定义附加到容器上的安全性策略。

Web 合作者负责认证与授权，并且负责某些其他的 service，例如日志安全性信息。一个客户请求可能已经被认证，或者作为会话中前一请求的一部分，或者使用 SSL 的客户方认证功能。如果没有被认证，那么 Web 合作者就能够要求客户经过一个授权过程，通常包括提供用户名与口令。

EJB 合作者不执行认证；它们仅仅关注于授权。EJB 合作者依赖于 SAS 执行对 Java 客户的认证。但 EJB 合作者支持非常健壮的委托安全性模型，并且为已经通过 SAS 认证的客户设置“run-as”标识，或者基于作为部署描述符一部分的其他定义设置这一标识。授权属性属于作为 EAR 一部分被打包的安全性策略。安全性信息放置在 XML 部署描述符中，并且通常由使用 AAT 的应用程序装配者定义。EAR 包包含下列安全性信息：

- 角色定义
- 方法许可
- run-as 模式或委托策略
- 认证质询类型
- 数据安全性设置

安全性合作者与运行在管理服务器上的安全性服务器交互。当一个合作者需要执行认证时，它与安全性服务器通信并传递认证请求。安全性服务器检查用户注册表并执行认证。WebSphere 的安全性服务支持操作系统、LDAP 服务器或通过 API 自定义的注册表所管理的用户注册表。认证过程的结果是一个关于用户身份的集合，它们将在会话阶段用于进一步

的请求。

19.2.1 认证模型

WebSphere 支持多种认证机制。其中最重要的两种机制就是 LDAP 与本地操作系统认证。这些机制的核心问题就是消除为每个应用程序构建自定义用户注册表的需要，以及以用户名/口令和更复杂的用户 profile 的形式重用用户信息。这样不仅节省了编程时间，而且通过减少必须在组织中维护的用户 profile 数降低所有者的总开销。用户常常在本地操作系统注册表（无论是 UNIX 系统还是 Windows 系统）中定义，并且 IT 部门常常不习惯于在自己的应用程序中重复这些定义，因为这意味着必须管理诸如口令修改等问题。WebSphere 允许这些信息被用于应用程序认证机制。通过访问 LDAP 服务来获取用户信息以及作为节点属性定义的完全用户 profile，已经逐步成为主流并且被用于企业应用程序中。

WebSphere 支持许多认证方法。这些方法是用户向 WebSphere 服务器认证自己的手段，而不用考虑服务器如何对用户注册表处理认证。这些方法包括：

- 使用基本的 HTTP 认证（即使用 HTTP 响应代码进行质询）询问用户名/口令。
- 使用 HTML 表单来收集用户的信息并且通过普通的 HTTP 请求将这些信息发送给服务器，这种方法询问用户名/口令。
- 询问数字 (X.509) 证书，在这种方式下，将在用户的客户机器上安装证书或读取证书，并且使用证书来认证用户。

图 19-2 显示了如何用 AAT 来为 Web 模块设置质询类型。在这里，使用一个负责实际登录过程的登录表单进行质询。

19.2.2 授权模型

大多数 WebSphere 安全性模型都致力于管理授权。WebSphere 遵守 J2EE 授权模型，表现在它基于通过应用程序装配者创建安全性角色与许可。图 19-3 显示了一个在 Web 应用程序中为 Web 模块定义安全性角色的例子。

通过定义角色被允许访问哪些资源与方法这一映射来为方法设置许可。首先，定义一个安全性约束集合，如图 19-4 所示。这些约束定义了哪些安全性角色具有访问权限，如图 19-5 所示，在这里为调度程序 (Dispatcher) 提供了对 Web 资源的访问权限，也为服务管理员 (ServiceManager) 提供了这一访问权限。在每个这样的约束中，可以创建资源组并定义哪些资源应该在某个特定的安全性约束下管理，如图 19-6 与图 19-7 所示。

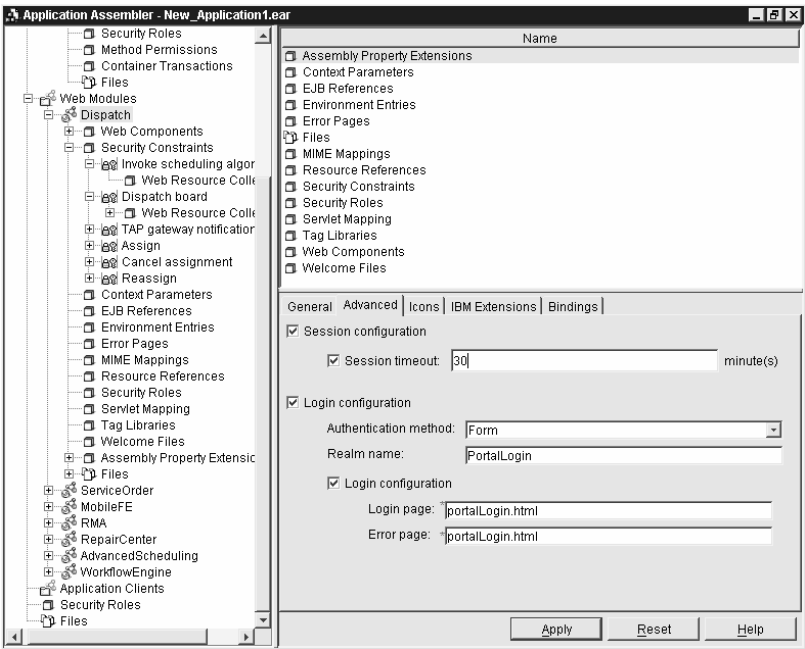


图 19-2 用于认证用户对 Web 应用程序访问的质询类型

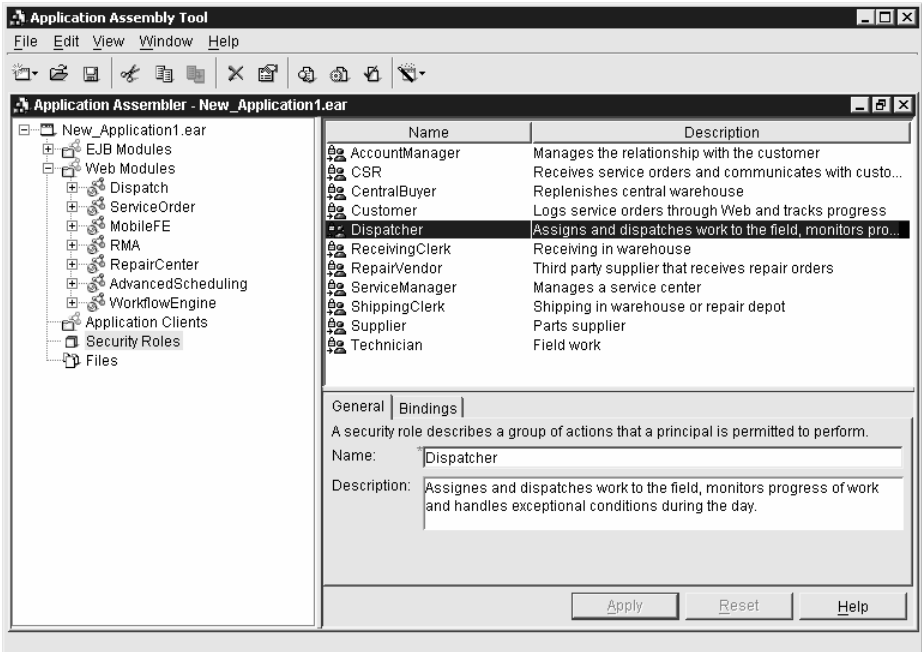


图 19-3 Web 模块中的安全性角色

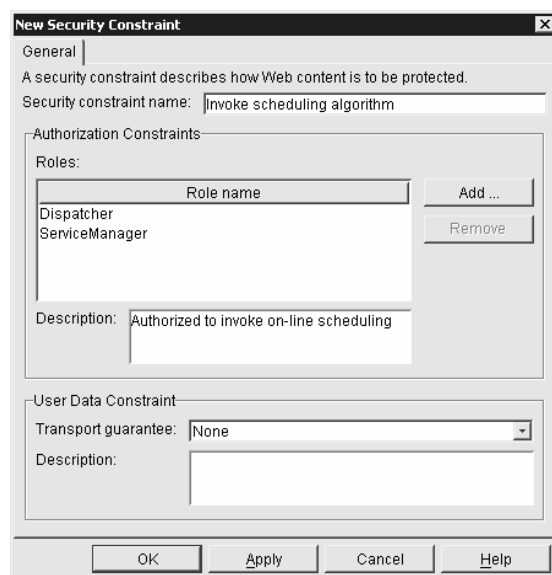


图 19-4 定义安全性约束

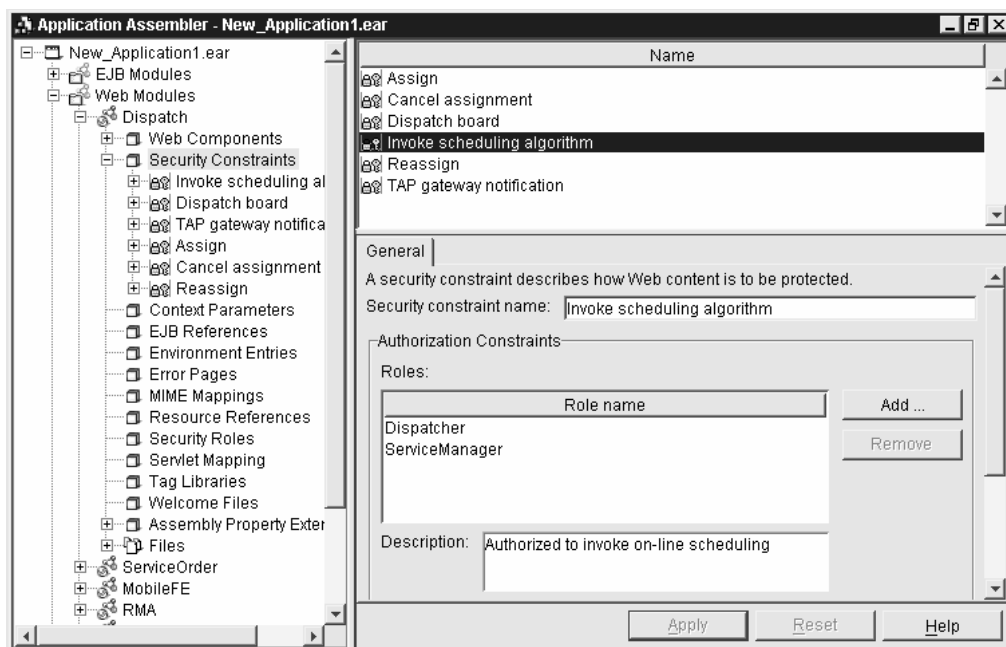


图 19-5 安全性约束中的安全性角色访问权限

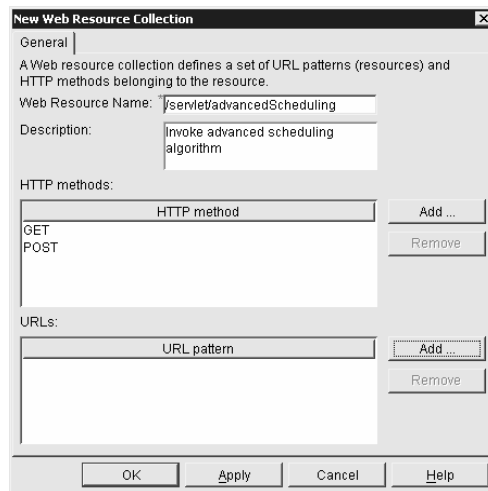


图 19-6 添加一个正在调度的 Web 资源由安全性约束管理

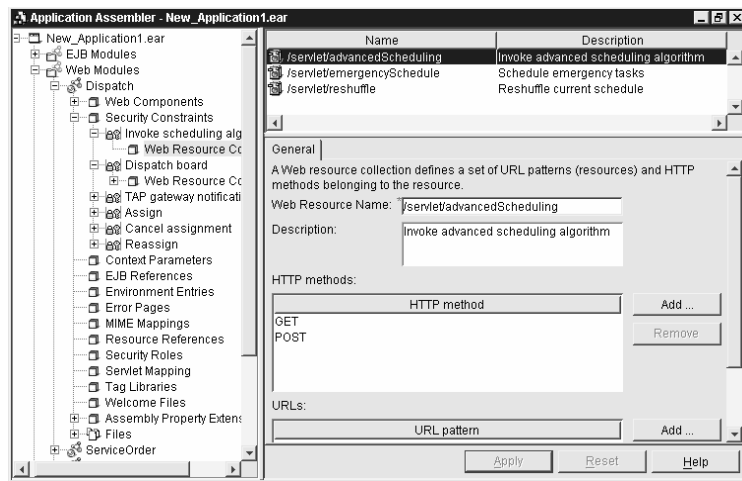


图 19-7 安全性约束管理的 Web 资源集合

当应用程序被部署时，管理员将定义用户与用户组。这些定义是动态的，并且在应用程序的生命过程中也可以进行定义。服务器管理员使用 Administration Console 将用户与用户组映射到安全性组。这些映射也可以使用 AAT 进行，如图 19-8 所示。这样就定义了哪个用户可以访问哪些方法。当某个请求到达服务器时，服务器将查看用户的标识并确定哪些安全性角色被赋给用户。基于这种信息，服务器就能够验证用户是否有权访问所调用的方法。图 19-9 显示了这一方案，其中，虚线代表在应用程序装配过程中执行的映射，实线代表由管理员执行的映射。

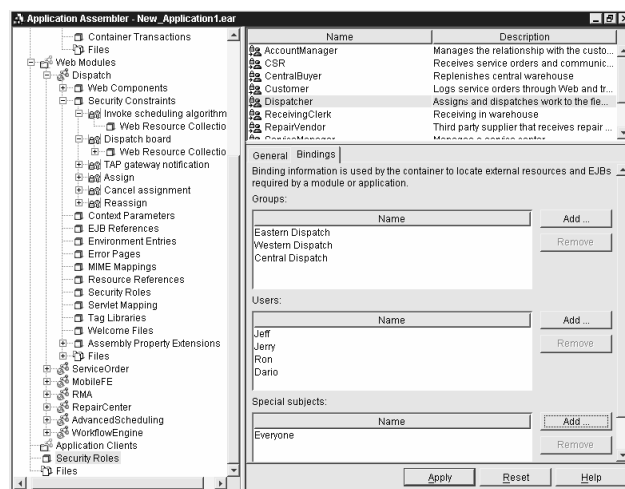


图 19-8 利用 AAT 映射用户与用户组

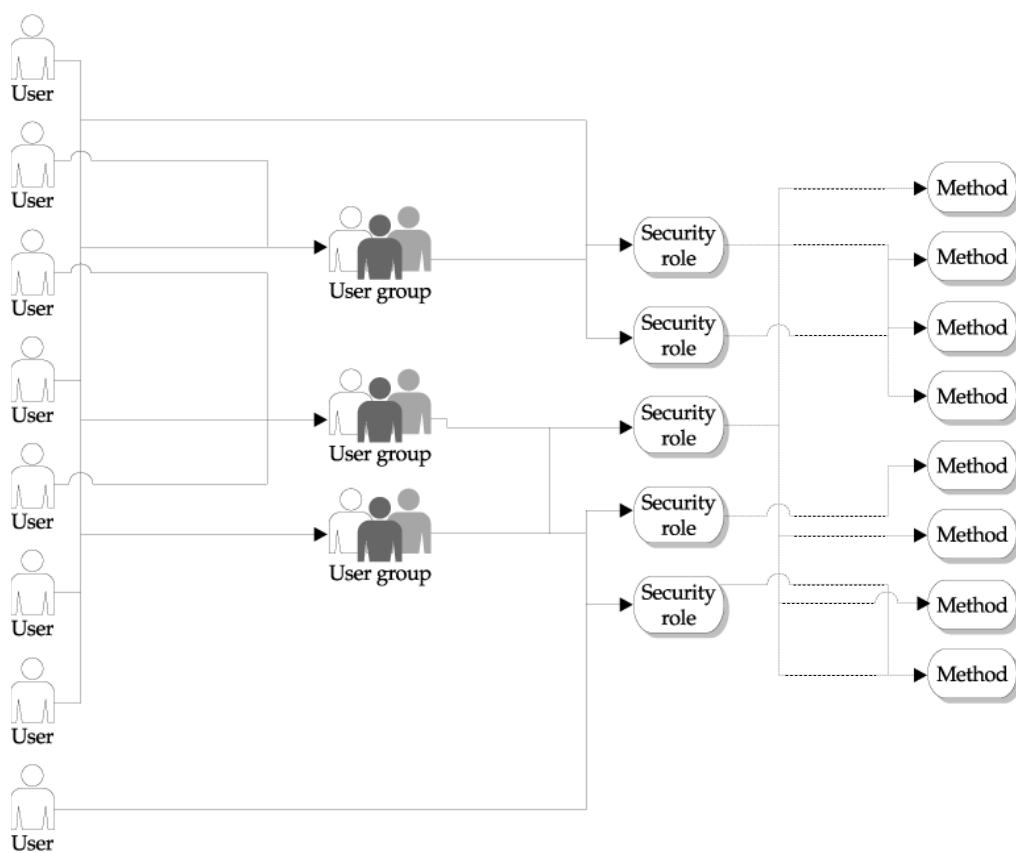


图 19-9 将安全性角色映射到方法

19.2.3 委托模型

委托是 J2EE 安全性模型中的另一个重要元素。它允许用户定义一个委托，用来代表用户执行某个方法调用。读者应该记得，一个服务器组件（例如 EJB）也可以作为从另一个组件请求服务的客户。在这种情况下，就可以基于原始请求发送者的授权 profile 或基于某些其他标准定义某个方法调用是否应该被执行。

WebSphere 支持对委托策略的定义与管理。当没有定义委托策略时，默认地总是假设使用原始调用者的安全性标识与 profile。但这并非总是正确的；有时，用户可能会构建一个组件，为了运行该组件，需要不同的许可。此外，用户还可能希望提供组件的一部分接口，从而能够访问该组件的功能。这时，就需要修改安全性属性以允许这种委托，其方法是使用 AAT 设置 Run-As 模式。可能的 RA 模式值包括：

- Run as caller（作为调度者运行；默认值）
- Run as system（作为系统运行）
- Run as an assigned specific identity（作为被赋给的特定标识运行）

图 19-10 显示了如何为高级调度 EJB 中的所有方法设置 Run-As 模式，以作为 Dispatcher 安全性角色运行。

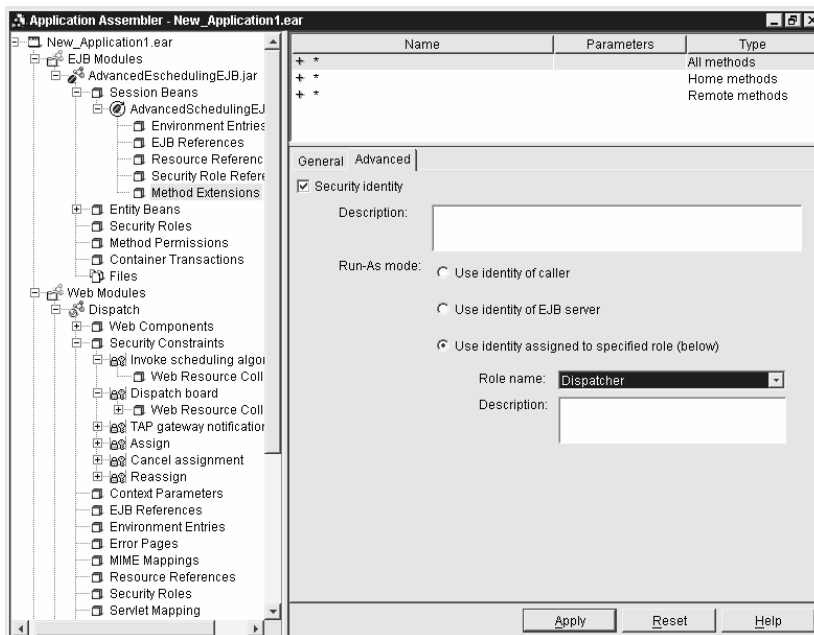


图 19-10 将方法的安全性委托策略设置为 run-as Dispatcher

19.2.4 部署描述符

安全性属性在部署描述符中维护。这些 XML 文件包含安全性信息，并且为每个所管理的资源包括附加的属性。作为部署描述符的一部分被维护的安全性属性包括：

- 安全性角色与方法许可
- 委托策略（也就是 Run-As 模式）
- 认证的质询类型
- 数据安全性设置

19.2.5 安全性 profile

WebSphere 维持了全局安全性 profile 以及特定应用程序安全性 profile。应用程序安全性可以为每个应用程序单独定义。这一安全性 profile 将覆盖这个特定应用程序的全局 profile。安全性相关的信息通常以 XML 维护在诸如 web.xml 这样的文件中。下列 XML 元素就是关于这些安全性信息的例子：

- 安全性角色：

```
<security-role id="Dispatcher">
  <description>Role assigning tasks to resources and dispatching
    roles to the field</description>
  <role-name>Dispatcher</role-name>
</security-role>
```

- 被管理的资源：

```
<security-constraint id="DispatchBoard">
  <web-resource-collection id=" DispatchBoard ">
    <web-resource-name>DispatchBoard</web-resource-name>
    <url-pattern>/servlet/dispatchBoard</url-pattern>
    <http-method>GET</http-method>
    <http-method>POST</http-method>
  </web-resource-collection>
```

- 认证/质询：

```
<login-config id="PortalLogin ">
  <auth-method>FORM</auth-method>
  <realm-name>PortalLogin</realm-name>
  <form-login-config id="PortalLogin">
    <form-login-page>portalLogin.html</form-login-page>
    <form-error-page>portalLogin.html</form-error-page>
```

```
</form-login-config>  
</login-config>
```

19.3 安全性管理模型

安全性是通过 Administrative Console 与 Security Center 工具管理的。可以从 Administrative Console 打开安全性中心。相同的过程和定义集合被用来管理所有资源，而无论这些资源的类型是什么。第 44 章着重介绍安全性管理，且重点在 Administrative Console。这里，仅仅给出管理什么以及为下列资源需要设置哪些安全性属性：

- Web 资源（HTML，JSP，servlet 等等）：Web 资源的安全性属性由 AAT 定义并且在 Web 档案文件、应用程序部署描述符以及 web.xml 文件中维护。Web 资源的安全性 profile 包括安全性角色定义、角色引用以及安全性约束。
- EJB：EJB 安全性模型很详细，WebSphere 基于 J2EE 安全性模型支持对它的管理。这个信息在 EAR 文件、应用程序部署描述符、安全性服务器的仓库以及各种 XML 与 XMI 文件（特别是 ejb-jar.xml，ibm-ejb-jar-ext.xmi 与 ibm-application-bnd.xmi）中维护。对安全性信息的设置是从 AAT 或 Administrative Console 中进行的，具体取决于设置属性的人是应用程序装配者还是部署者。EJB 的安全性 profile 包括关于角色、角色引用、方法许可、Run-As 标识以及 Run-As 映射等信息。
- 全局安全性设置：这是为服务器上部署的所有应用程序与组件设置的安全性属性。
- SSL 设置：图 19-11 与图 19-12 显示了如何使用 Security Center 设置超时与 SSL 密钥文件参数。

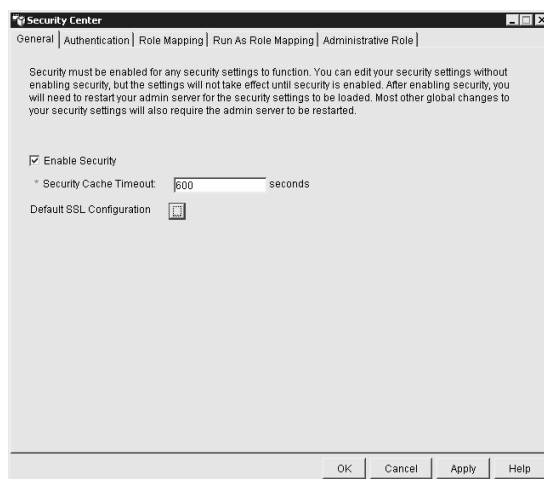


图 19-11 在 Security Center 中启用安全性并设置超时

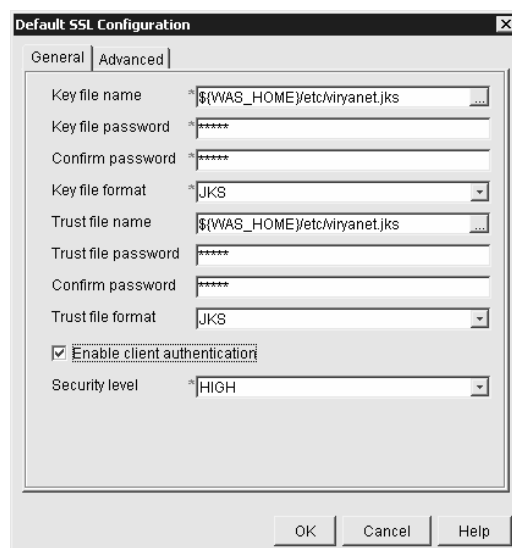


图 19-12 在 Security Center 中设置 SSL 密钥文件值

- **认证设置：**图 19-13 与图 19-14 显示了如何使用 Security Center 设置用户注册表源——认证是通过操作系统的用户注册表完成还是通过 LDAP 源完成。

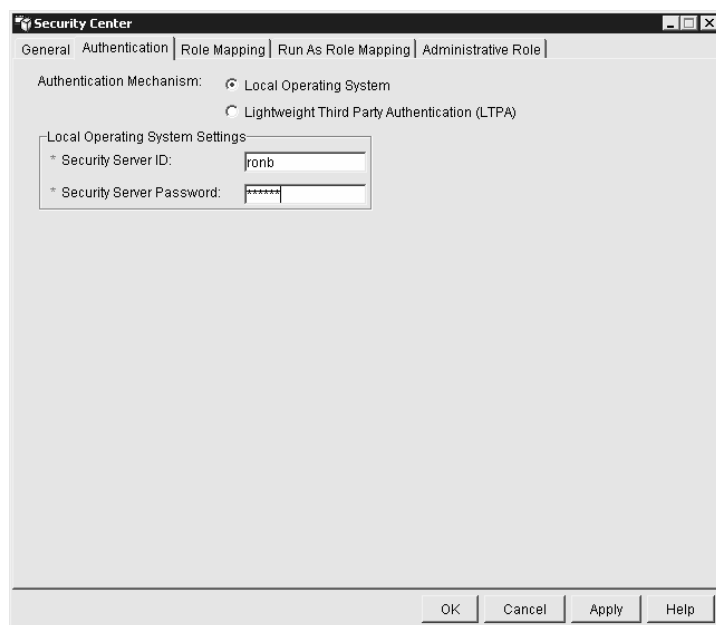


图 19-13 在 Security Center 中设置本地操作系统认证

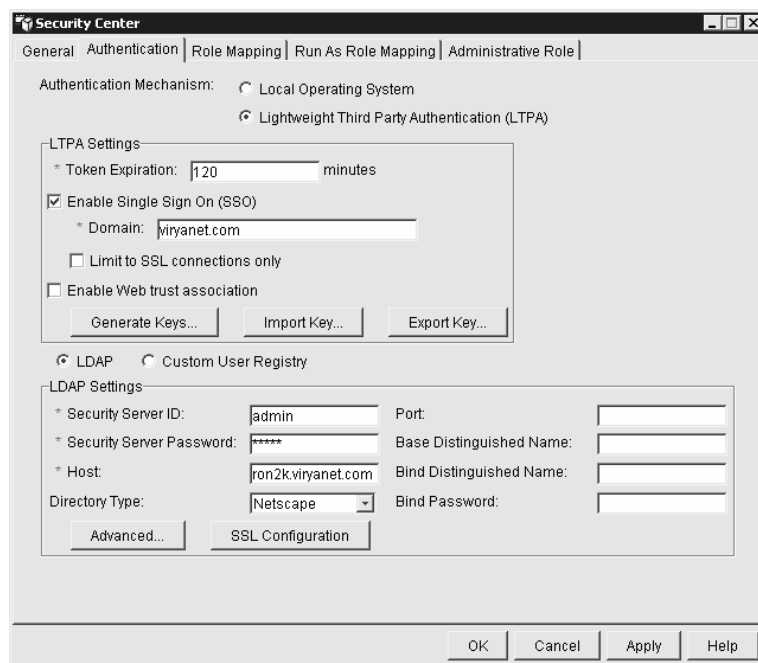


图 19-14 在 Security Center 中配置 SSL

- 安全性角色：当为 Web 资源与 EJB 设置 profile 时，它们被重用。

19.4 小 结

本章简要描述了 WebSphere 安全性模型并且描述了 WebSphere 如何实现 J2EE 安全性模型。详细讨论了认证与授权模型，并且描述了委托模型。这里讨论的重点是在应用程序装配过程中需要进行的定义(第 44 章重点关注运行时的安全管理以及应用 Administrative Console 进行与安全性相关的活动)。

在描述认证时，我们提到了一种可能的认证方法，就是使用 LDAP 服务器，在这种情况下，实际的认证动作是使用 LTPA (Lightweight Third Party Authentication，轻量级第三方认证)进行的。LDAP 协议可用来访问目录服务器中的信息，并且逐步成为存储用户信息与认证用户的一种选择。本书将在第 20 章讨论 LDAP。

第 20 章 LDAP 支持

现在，任何提到有关安全性的问题，都必须包括对 LDAP (Lightweight Directory Access Protocol, 轻量级目录访问协议) 的讨论。安全性对于标识用户 (例如通过用户名与口令)、认证用户身份、获取与用户相关的许可与 profile 都是必要的。在所有其他事情中，用户所属的用户组 (或角色) 信息必须被提取出来。

这些信息几乎需要在所有应用程序中维护，并且这也可能是导致工作复杂化的一个原因。维护这些信息的每个应用程序都需要能够访问该信息以正确地工作。但是，用户并不希望必须每次都输入这些安全性信息；因此，有必要提出一种更好的方法来维护安全性信息并且在多个应用程序上对安全性信息进行映射，从而使这些工作就像进行一次签名一样方便。这样做的关键就是目录服务，允许用户以所有应用程序都能访问的方式维护信息。

20.1 目录服务

目录 (directory) 就是按某种顺序组织的关于对象的信息集合。日常生活中关于目录的例子包括电话号码目录与库目录。从软件的角度看，目录就是为带有关键字的对象列出信息的数据库或数据仓库。普通数据库与目录之间的主要区别在于，与普通数据库相比，目录有更多的查找请求但有相对较少的更新请求。这种类型的用法为专用化与优化提供了可能，并且这也是目录服务器需要进行的工作。

目录允许用户或应用程序根据某个特定任务所需的特点查找资源。例如，关于用户的目录可用来查找某个人的电子邮件地址或传真号码。目录也可以被搜索以查找用户所属的用户组，然后对这个目录进行查询以确定施加到这个用户组或这个用户上的属性。

目录服务通常被用来访问信息，包括用来查找赋给某个被管理实体 (例如，为用户维护的 profile) 的属性信息、更新这些信息 (这种动作较少发生)、在给定了某些选择标准之后查找一个实体。目录服务是由目录服务器提供的，软件组件管理该目录并实现服务。图 20-1 显示了使用目录客户的应用程序访问目录服务的情况。

20.2 命名服务

任何类型的分布式编程所面临的一个关键问题就是命名 (naming)。当客户需要获取资源的访问时，它们必须有一个方法来将“逻辑”名转换为实际的对象引用，并且它们必须透明

地完成这一工作（因为对象有可能在另一台远程机器上）。Java JNDI 就是用来访问命名与目录服务的 Java API。在第 5 章讨论并使用 JNDI 在 Web 服务器上访问 EJB，在那里，实际的命名服务是由各自的 CORBA Service 提供的。通过使用 JNDI，就屏蔽了关于命名服务底层实现的细节。一个值得研究的关于命名服务的底层实现就是 LDAP。

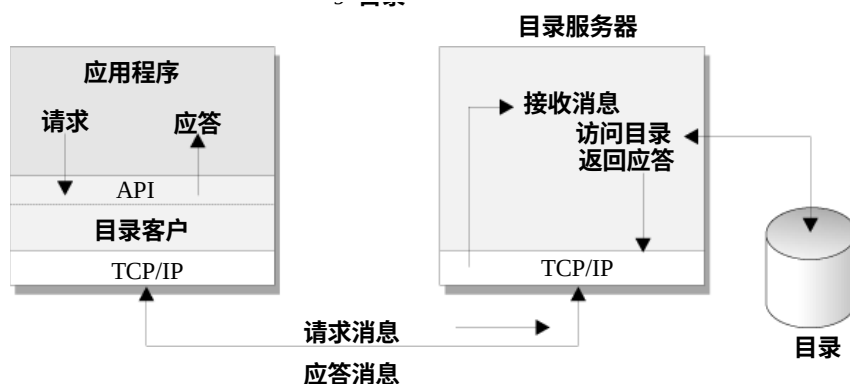


图 20-1 应用程序访问目录服务

LDAP 是开放的工业标准，目标是提供一种方法来管理与文件、用户、打印机以及其他可通过不同的网络共享的资源的信息。LDAP 就是一个通过 JNDI 访问的目录服务的例子。

命名服务用来标记任何实体，该实体具有一个后来可被用于查找该资源的通用名称。在安全性上下文中，这个需要标记的实体就是代表用户的实体，而这个用户正在试图访问 WebSphere 环境所管理的应用程序。

WebSphere Application Server Advanced Edition 允许在 LDAP 服务器上存储用户注册表（主要是为了进行 WebSphere 的认证）。我们推荐使用这种环境，因为它能够减轻用户进行安全性管理的工作，并且提供了一个可以在安全性级别上（使用 LDAP 目录）进行应用程序集成的可用环境。但是，为了使用 LDAP，需要安装 LDAP 服务器，LDAP 服务器不是 WebSphere Application Server 发布的一部分。用户可以使用 IBM SecureWay 目录服务器，这个服务器包括 LDAP 的完全实现。也可以使用第三方提供者的 LDAP 服务器与 WebSphere Application Server。需要说明的是，这个功能在高级单服务器版中不可用。

20.3 LDAP 是什么

随着计算机作为支持业务处理的工具这一趋势越来越流行，以及 Internet 作为业务处理信息的传输桥梁，导致出现了目录服务这一需求，利用目录服务，用户能够在某个网络上查到所需的人员或资源。这种使用方法与库目录或电话号码目录的用法并没有什么不同。

LDAP 出现于 20 世纪 90 年代早期，作为 DAP（Directory Access Protocol，目录访问协议）

的轻便的可选方案，DAP 是 X.500 协议的规范的一部分。X.500 是由几个国际组织（特别是 ISO（International Standards Organization，国际标准化组织））发起的标准化工作的一部分。这些标准化努力的结果就是出现了所谓的 OSI（Open Systems Interconnect，开放系统互连），它一度被认为是 Internet 协议的另一个替代。OSI 模型是一个参考模型，它定义了关于数据通信的七层，其中以物理层作为最低层，逐步向上，以应用程序协议作为最高层。OSI 参考模型以及相应的协议完全是新开发的，使用这些协议通常能够得到更干净的应用程序设计。但是，其相对较慢的开发过程以及很慢的部署与安装过程导致 OSI 协议的最终失败，特别是由于基于 TCP/IP 的 Internet 的快速发展，更加速了其失败的进程。

与其他 Internet 标准相似，在达到目前的状态之前，LDAP 也经过了几个发展阶段。现在，它已经成为业界标准，并且大多数厂商都支持 LDAP。实际上，所有面向底层结构软件的厂商都提供了一个 LDAP 产品以及许多可用于存储与检索标注信息的公共域服务器。虽然 LDAP 曾一度被认为将成为 Internet 上的中央注册处，但后来出现了许多极具竞争力的标准，这些标准之间的竞争使得没有任何一个标准能够胜出（正如许多人所说的那样：“关于标准的最好事情就是同时存在多个标准”）。

LDAP 基于客户-服务器范型。虽然初看上去 LDAP 似乎是一个“目录”协议，但它实际上定义了 LDAP 客户与 LDAP 服务器之间的通信协议。通过这个协议，以及所涉及的消息结构，就导致了目录模型。换句话说，通信协议就隐含着施加在 LDAP 服务器设计者上的约束，并且这些约束抽象地定义了 LDAP 目录是什么。

LDAP 在 TCP/IP 之上实现，同样，客户与服务器的交互方式代表了其他基于 TCP/IP 的协议，例如 HTTP 或 FTP。会话是由“绑定”到服务器上的客户初始化的（这对应于 TCP/IP 中请求为 LDAP 服务器指定主机 IP 地址以及 TCP/IP 端口的操作）。在进行（可选的）用户名与口令认证过程之后，实际的会话就开始了。如果没有认证，客户可以假设“匿名”标识并且提供默认的访问权限。会话自身也包含一系列关于目录数据的操作——也就是搜索与更新操作。操作是通过客户发出请求，服务器进行响应的方式完成的。当客户完成动作之后，会话将关闭。

20.3.1 区别名与相对区别名

LDAP 客户请求的所有搜索与更新操作主要是对基本数据结构进行处理，LDAP 服务器在这个数据结构中存储信息（被称为目录表项（directory entry））。一个单独的目录表项描述了现实世界中的一个对象（例如人、打印机、计算机）。目录表项由一个称为 DN（Distinguished Name，区别名）的名称来标识，这个名称又由一些更小的部分（称为 RDN（Relatively Distinguished Name，相对区别名））组成。这种情况非常类似于在大多数现代操作系统（例如 UNIX 或 Windows）中用来组织文件的目录层次。

例如，考虑一个名为 E:\WebSphere\AppServer\readme.txt 的文件。这个名称由 4 个用反斜杠分开的小名称组成。这些引用驱动器、目录或文件名的小名称等价于 RDN。除了表项名称

之外，每个表项还包括一个或多个属性用来构成表项描述。例如，一个关于计算机的目录表项可能包括类型、操作系统、IP 地址、物理位置以及其他信息。这些属性列表，以及每个属性的类型和可能的语法限制，都在 schema 中描述。schema 与目录表项之间的关系与面向对象语言中类和对象之间的关系相同——也就是，目录表项可以被看作是 schema 的一个实例。新的 schema 可以从已有的 schema 继承，这就像面向对象语言中的类继承一样。有一个称为 objectClass 的特殊属性，它在每个目录表项中存在，用来标识该目录表项所属的 schema。这个属性在对象类的“顶部”定义，在本质上就是根类，所有的 schema 都从它继承。

20.3.2 属性

在一个特定的目录表项中，有 0 个或多个值与每个属性 (attribute) 相关联。0 个值表示没有值与这个属性相关联，这对于可选的属性是可能的 (属性是否为可选的是 schema 定义的一部分)。最常见的情况是属性有一个值，但也可能出现一个属性有多个值的情况。多个值对于某些属性 (例如电话号码) 是有用的，在这种情况下，就有多个值与一个对象 (在电话号码的例子中，这个对象就是人) 相关联。

属性类型或语法用来定义一个属性可以使用什么类型的值。类型系统非常灵活，它允许定义通用的二进制数据，也可以是定义大小写敏感或大小写不敏感的可打印字符串。也可以定义属性语法的新类型，并且限制值或内容的长度。

属性名与值用来指定 RDN。应该记得，表项名 (DN) 由一些称为 RDN 的小名称组成。它们通常具有形式 <attribute name>=<value> (<属性名>=<值>)，但一般来说，可以指定一个 RDN 作为以正号分开的属性/值对。下面给出了一个两个属性/值对的例子：

```
<attribute name>=<value1>+<attribute2 name>=<value>
```

20.3.3 目录信息树

目录表项是以称为 DIT (Directory Information Tree, 目录信息树) 的类似于树的结构组织的，该结构以一个虚拟的根节点开始。对于一个给定的表项，构成 DN 的 RDN 对应于树中从虚拟根节点到表项节点的路径。DN 是一个以逗号分开的 RDN 的列表，但不同于文件系统路径名，RDN 被指定的顺序是从表项节点逐步向上到根节点。

图 20-2 显示了一个简单的 DIT。在这个例子中，目录由属于某个部门与组织的人组成。因此，Akitami 的销售部门的 Ed Lopez 的 DN 就是 cn=Ed Lopez, ou=Sales 与 o=Akitami。在这个图中显示的目录根是虚拟的，没有相应的目录表项与之对应。

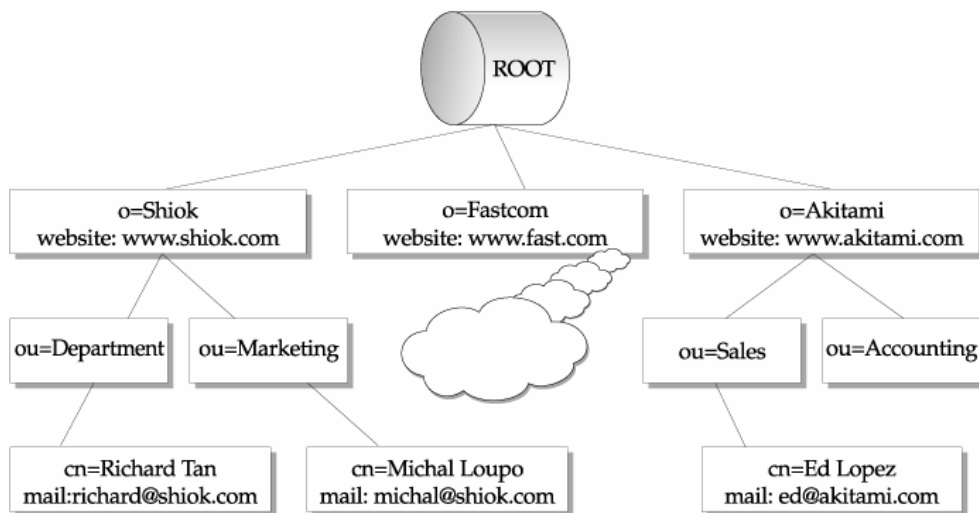


图 20-2 一个关于 DIT 的简单例子

在 DIT 内部，也可以创建别名 (alias)，它可以看作节点之间的链接。这些别名也就是 DIT 是一种类似于树的结构而不是真正的树结构的原因。别名可用来表示某些实体属于两个类的情况 (如一个人可能属于两个部门)。使用别名的另一个目的就是确保向后兼容 (也就是，当对 DIT 进行修改时，可能希望原来的 DN 仍然有效 (至少临时有效))。这很容易实现，只要将该目录表项移动到新位置，但同时在原来的 DN 上留下一个别名即可。

20.3.4 引用

考虑到 LDAP 所面向的应用程序类型，可伸缩性 (scalability) 是一个重要问题。特别地，这意味着只有一个服务器对于维护整个 DIT 是不够的，为了支持可伸缩性，DIT 应该包括对其他服务器的引用 (referral)，从而允许利用多个服务器创建分布式 DIT。服务器中的顶级节点称为服务器后缀 (suffix) ——它是这个服务器中所有 DN 的最长公共后缀。实际上，一个单独服务器可以有几个后缀，分别对应 DIT 的几棵子树。对其他服务器的引用由一个特殊类型的表项来表示，并且可以按以下两种方法之一进行处理：正在请求的客户可以获取引用并访问被引用的服务器，或者，发生服务器链接 (chaining)。对于后一种情况，客户最初联系的服务器将向被引用的服务器发出请求。

20.3.5 操作

现在将描述如何在 DIT 上执行操作。LDAP 支持的最重要操作就是搜索 (search) 操作。LDAP 是基于所执行的大多数操作都是搜索操作这一 (合理的) 假设而设计的。只有非常少的请

求是更新操作。如果不是这样，那么任何标准的数据库系统都可能比 LDAP 服务器更好。搜索操作允许客户在 DIT 的某一部分进行搜索，以查找表项。

搜索操作的参数如下：

- Base (基)：DIT 中的一个节点，搜索过程从这个节点开始。
- Scope (范围)：在 DIT 中进行的搜索的深度。这里有 3 个选项：
 - BaseObject (基对象)：仅仅对基节点进行搜索。
 - SingleLevel (单级)：仅仅对基节点的直接子节点进行搜索。
 - WholeSubTree (整个子树)：对基节点的所有后代进行搜索。
- Search Filter (搜索过滤器)：所返回的表项必须满足的搜索标准。搜索标准由一系列与属性相关的断言组成，并且具有属性操作符的形式。操作符包括=、>=、<=、=*以及~=。前 3 个比较操作符的意义很明确，=*操作符不带任何值地使用，并且将返回为该属性设置了一个值的所有表项；~=操作符提供了一个近似的匹配。如果提供了多个断言，那么应该在这些断言之间使用 Boolean 操作符，这些操作符可以是 &、|或!，分别表示 AND (与)、OR (或)、NOT (非)。这些操作符是以前缀方式表示的，也就是操作符出现在参数之前。例如，“&(cn=Ed Lopez)(website=www.akit-ami.com)”。
- Returned Attributes (返回的属性)：将要返回的属性列表。
- Alias Dereferencing (别名复引用)：是否对别名进行跟踪（如果不对别名进行跟踪，那么对“别名”自己进行检查并返回；如果对别名进行跟踪，那么搜索将继续到达链接所指的地方）。
- Size and Time Limit (大小与时间限制)：将返回最大表项数目以及最长搜索时间。这样可以限制搜索引入的负载量。

更新操作包括向目录中添加新表项、删除表项（必须是叶节点）、修改已有的表项。修改已有表项的一个特殊情况就是修改 DN，这就意味着可能修改 DN 的第一个（最低有效）组件，也可能修改节点的位置到一个不同的子树。

所有操作，特别是更新操作，可能需要授权。LDAP 支持各种级别的用户认证（分别对应于不同的安全性级别），包括从通过基本授权（类似于 HTTP 使用的授权方式）的无授权，到功能更加强大的认证方案（可能使用 SSL 以及类似的安全性机制）等不同级别。

20.4 Java 命名与目录接口

本章讨论基于 Java 的 API（称为 JNDI），它是非特定于 LDAP 的通用 API，并且也应用于其他命名与目录服务（例如 DNS (Domain Naming Service, 域名服务)）。开发 JNDI 是为了创建一致的模型访问在不同目录服务器中维护的不同资源。这听上去有点像创建 LDAP 的

目标。但是，JNDI 能够包括进行资源管理的其他方法（例如 CORBA，DNS），但它并不包含目录管理的逻辑。

JNDI 是 J2EE 的一部分，它包含在下列包中：

- `javax.naming`：用来在命名服务中查找对象与值的基本 API。
- `javax.naming.directory`：用于执行过滤搜索与更新的更复杂 API。
- `javax.naming.spi`：服务提供者开发人员使用的接口。应用程序开发人员对这些接口几乎没有兴趣，因为它们是为了开发对特殊服务（例如 LDAP）的支持而设计的。

为了给 EJB 提供命名服务，WebSphere 包括了一个关于 JNDI 服务器的内置实现。在第 5 章中的“Hello EJB”示例就使用了 JNDI（虽然是基本的使用）。在 WebSphere 4.0 的高级版本中，这一功能是由 `adminServer` 提供的，而在高级单服务器版本中，则由应用程序服务器自己实现。

20.4.1 命名服务包

`javax.naming` 包为了给出通用的命名服务，提供了一个接口与类的集合。在命名服务包中的最基本抽象就是上下文（context）。上下文可以被看作在其中发生操作的名称空间。关于上下文的例子有 LDAP 节点（及其相关联的子树）或文件系统目录（以及该目录中的所有目录）。关于上下文的一个重要属性就是它们能够包含其他上下文（也就是说，它们能够形成一个递归（recursive）结构。例如，在文件系统中，目录可以包含其他目录。当使用 JNDI 时，原始上下文（initial context）被认为是使用该上下文的所有命名操作的根。因此，如果文件系统上的原始上下文被设置为 `E:\WebSphere\AppServer`，那么文件 `E:\WebSphere\AppServer\readme.txt` 的路径就是 `\readme.txt`。

为了使用 JNDI，首先需要创建一个上下文，使所有的操作都基于该上下文。上下文由命名服务包中的 `Context` 接口表示。为了创建一个上下文，需要为 JNDI 建立环境变量。表 20-1 列出了 JNDI 所使用的某些属性，这些属性与 WebSphere 客户创建上下文的动作相关。JNDI 还支持许多其他环境变量，用来定义安全性、国际化以及与服务相关的参数。对于那些没有显式设置的属性，将使用系统默认值。为了设置属性，可以从文件中读取（使用 `Properties` 对象的 `load` 方法，或提供一个在属性文件上打开的 `InputStream`），或者，可以在运行时使用 `properties` 对象的 `put` 方法。

表20-1 JNDI环境

环境属性	定义常量	描述
<code>java.naming.factory.initial</code>	<code>Context.INITIAL_CONTEXT_FACTORY</code>	原始上下文工厂的类名。在 WebSphere Server Advanced Edition 中，应该被设置为 <code>com.ibm.</code>

(续表)

环境属性	定义常量	描述
java.naming.provider.url	Context.PROVIDER_URL	WebSphere.naming.WsnInitial ContextFactory 指定配置信息，供提供商使用。对于 WebSphere 高级版，应该为 iiop://servername:portnum。可以使用 WebSphere Administration Client 确定端口号，默认值为 900
java.naming.dns.url	Context.DNS_URL	指定 JNDI URL 上下文使用的 DNS 主机与域名

类 `javax.naming.InitialContext` 实现了 `Context` 接口，并且被用来创建上下文对象。利用该上下文，就可以检索命名信息。`Context` 接口的检索方法使用 `String` 或 `Name.name` 作为参数，这个参数是一个用来表示 0 个或多个元素的有序序列的接口，对于需要使用多个名称空间的情况非常有用。`Context` 接口支持 4 种检索方法（每种方法都有两个变体，一个对应于 `String`，一个对应于 `Name`）：

- `list`：返回作为参数（parameter 是当前上下文中一个上下文的名称）被传递的上下文上所绑定的对象的名称以及对象的类名称对应的一个 `NamingEnumeration`（`Enumeration` 的扩展）。返回的集合中包括 `NameClassPair` 对象，从这些对象中可以检索名称与类名称。
- `listBindings`：返回名称的 `NamingEnumeration` 以及在参数上下文中绑定的实际对象。在所返回的集合中包括 `Binding` 对象，从这些对象中可以检索名称与对象。
- `lookup`：返回指定名称的对象。
- `lookupLink`：跟踪链接，并返回指定名称的对象。

在上下文中对对象的管理是使用绑定对象完成的。`Context` 接口提供了下列与绑定相关的方法：

- `bind`：在这个上下文中将名称绑定到对象（如果该名称已经被使用，那么将产生一个异常）。
- `createSubcontext`：创建并绑定一个新上下文到名称（等价于文件系统（例如 MS-DOS）中的 `mkdir`）。
- `destroySubContext`：析构上下文并从名称空间中删除它（等价于文件系统（例如 MS-DOS）中的 `rmdir`）。
- `rebind`：将名称绑定到对象，覆盖任何已有的绑定。
- `rename`：获取绑定到一个名称上的对象，并将该对象与一个新名称绑定。
- `unbind`：从名称空间中解除指定名称的对象的绑定（等价于一个删除操作）。

20.4.2 目录服务包

`javax.naming.directory` 对命名服务包进行扩展，从而提供了访问与管理目录服务的功能。它支持的主要功能是对象属性，作为搜索操作的基础。目录服务包的主要思想来自 LDAP 规范中目录搜索的概念。

为了在目录中搜索一个对象，需要创建一个实现了 `javax.naming.directory.DirContext` 接口的对象。该接口扩展了 `Context` 接口，使得所有命名服务包操作也可以在这里应用。

`InitialDirContext` 类是 `DirContext` 接口的实现。

目录服务包引入了下列两个接口用来表示属性：

- **Attribute**：代表一个与指定名称的对象相关联的属性。这个属性自身是关于 0 个或多个对象的序列。这个属性也可以以 `DirContext` 的形式提供一个 schema，用来描述属性的结构。还可以提供另外一个 `DirContext`，用来保存这个属性中的值的语法（语法的定义与提供商相关——也就是，它们可能是其意义依赖于底层服务的字符串）。
- **Attributes**：代表一个与指定名称的对象相关联的 `Attribute` 对象的集合。

这两个接口分别为添加/删除值和属性提供了方法。

`DirContext` 接口提供了关于搜索方法的 4 种形式。最简单的形式就是使用一个名称以及一个 `Attributes` 对象作为参数。它搜索包含所提供属性集合的对象。搜索的结果是一个 `NamingEnumeration`，这与命名服务包类似。`NamingEnumeration` 中的元素是 `Binding` 类 `SearchResult` 的子类，它提供了对搜索属性的访问。

下面给出了一个使用 `BasicAttribute` 与 `BasicAttributes` 类完成目录搜索的示意代码，它实现了相关的接口。

```
Properties props = new Properties();
props.put(Context.INITIAL_CONTEXT_FACTORY,
    "com.ibm.jndi.LDAPCtxFactory");
props.put(Context.PROVIDER_URL, "ldap://host");
try {
    // Create Search Context
    DirContext ctx = new InitialDirContext(props);

    // Define attributes
    Attributes attrs = new BasicAttributes(true);
    attrs.put(new BasicAttribute("cn"));
    attrs.put(new BasicAttribute("website"));

    // Perform search
    NamingEnumeration result = ctx.search("ou=shioK", attrs);
```



```

        // Iterate over results
        while (result.hasMore()) {
            ...
        }
    }
    catch (NamingException ex) {
        ...
    }
}

```

一个更高效的搜索方法允许用户预先指定需要获取的是哪些属性（例如，如果正在搜索关于人员的目录，那么可能仅仅关心他们的电话号码）。

最后两个搜索方法允许用户执行带有过滤的搜索以及带有控制的搜索。这两种思想在前面关于 LDAP 的描述中已经说明。为了进行过滤，为搜索方法提供了一个过滤字符串，其格式与 LDAP 搜索过滤器相同。

搜索控制由 SearchControls 对象表示，该对象定义了如下内容：

- Search scope（搜索范围）：OBJECT_SCOPE，ONELEVEL_SCOPE 或 SUBTREE_SCOPE（分别与基对象、单级或整个子树对应）。
- Maximum number of entries（最大表项数）：对返回的表项的数目进行限制。
- Maximum time（最长时间）：搜索的时间限制，单位为毫秒。
- Dereference flag（复引用标志）：指定在搜索过程中是否对链接进行复引用处理。
- Returned attributes（返回的属性）：指定返回哪些属性（同样，当仅仅返回所需要的属性部分时有助于提高效率）。
- Return object values（返回对象值）：指定应该返回对象值还是返回名称与类。

下面是一个使用搜索过滤与搜索控制的示意性例子：

```

Properties props = new Properties();
props.put(Context.INITIAL_CONTEXT_FACTORY,
           "com.ibm.jndi.LDAPCtxFactory");
props.put(Context.PROVIDER_URL, "ldap://host");
try {
    // Create Search Context
    DirContext ctx = new InitialDirContext(props);

    String[] attrs = { "email" };
    String filter = "(&(cn=E*)(website=www.shiok.com))"

    // Define Search Controls
    SearchControls sc = new SearchControls(SUBTREE_SCOPE, 1000,
                                           0, attrs, false, false);

    // Perform search
}

```

```
NamingEnumeration result = ctx.search("ou=shio", filter, sc);
// Return results
while (result.hasMore()) {
    ...
}
}
catch (NamingException ex) {
    ...
}
```

目录服务包允许通过 `DirContext` 接口修改属性，这对于将属性设置到对象上是非常重要的，只有这样，搜索方法才有可使用的搜索对象。这一动作是通过 `modifyAttributes` 方法实现的，该方法使用一个名称、一个属性数组以及将要执行的操作为参数。操作可以是下列之一：

- `DirContext.ADD_ATTRIBUTE`
- `DirContext.REPLACE_ATTRIBUTE`
- `DirContext.REMOVE_ATTRIBUTE`

或者，可以使用一个名称以及一个关于 `ModificationItem` 对象的数组调用 `modifyAttributes`，这些对象实际上就是上述操作与 `Attributes` 对象的配对。

JNDI 目录服务包的最后一项功能，就是它允许使用 `getSchema` 方法与 `getSchemaClassDefinition` 方法获取一个目录的 schema，这两个方法都返回 `DirContext` 对象（类似于前面提到的 `Attribute` 中的 `schema` 方法）。

20.5 WebSphere 中的 LDAP

正如本章开始时描述的那样，为了在 WebSphere 中使用 LDAP，需要 LDAP 服务器。一种选择就是使用 IBM SecureWay Directory，其优点是能够与 WebSphere 更好地集成。

在 WebSphere 上下文中使用 LDAP 的一种可能用法就是维护用户注册表，用户注册表是认证过程的基础。换句话说，WebSphere 的安全性服务能够使用基于 LDAP 的用户数据库。由于 WebSphere 是一个应用程序服务器，它需要维护认证信息以及访问控制信息。最起码，它需要维护用户名与口令。

对于维护用户注册表，WebSphere 支持 3 个可选的方案。一个就是使用本地注册表，也就是说，注册表由底层操作系统（例如 Windows 2000 或 UNIX）维护。第二个就是使用客户定义的注册表，可以使用 IBM 提供的 API 实现该注册表。第三个方案（在这里将要讨论的方案）就是在 LDAP 用户注册表上执行用户 ID 与基于口令的认证。

20.5.1 IBM SecureWay Directory

IBM 的 SecureWay 产品实现了 LDAP 并提供了遵守 LDAP 的目录服务器。其底层的数据存

储器是 DB2 (记住,最基本的目录服务器就是一个其中大部分活动都是查找操作的数据库)。从这个角度来看,SecureWay 并没有通过专用化提供更高的性能。但是,DB2 是目前最快的关系数据库之一,因此,使用 SecureWay 应该不会出现性能问题。此外,IBM 还提供了 SecureWay Directory Client SDK,用来提供与目录服务器交互所必需的 API。然后,这些 API 又被 WebSphere 目录服务器用来访问该目录服务器,如图 20-3 所示。

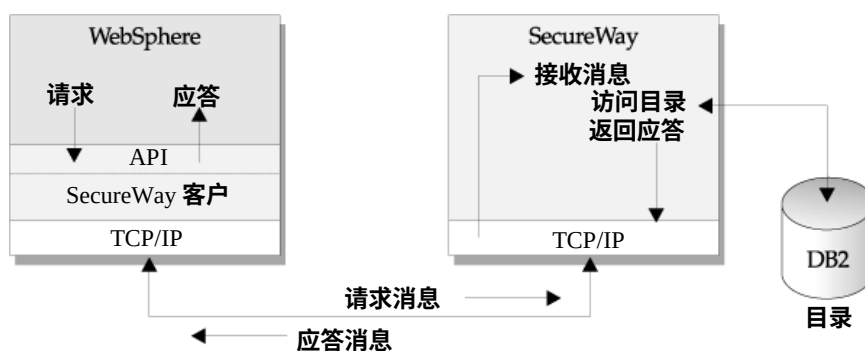


图 20-3 WebSphere 访问 SecureWay

为了使用 SecureWay 来管理 WebSphere 用户注册表,需要对 SecureWay 进行一些设置。首先,需要建立一个根,从这个根就能够管理目录服务器上的所有用户。典型地,这将代表用户或应用程序所属的组织。可以使用 Directory Administration Tool 完成这项工作,如图 20-4 所示。

为了添加 WebSphere 使用的根,需要添加一个后缀:

1. 从 Directory Administration Tool 打开 Suffixes 文件夹并选择 Add a Suffix。
2. 将后缀的 DN 设置为 o=<your company name>与 c=<your country>。
3. 打开图 20-5 所示的 Java 管理客户。可以忽略所出现的警告——该警告告诉用户这个根之下的 dtree 为空。
4. 需要 rebind 树。这时将会询问用户名与口令;请使用管理员口令。
5. 可以继续操作,在这个根中添加用户。记住,这些用户构成了 WebSphere 中的用户注册表。

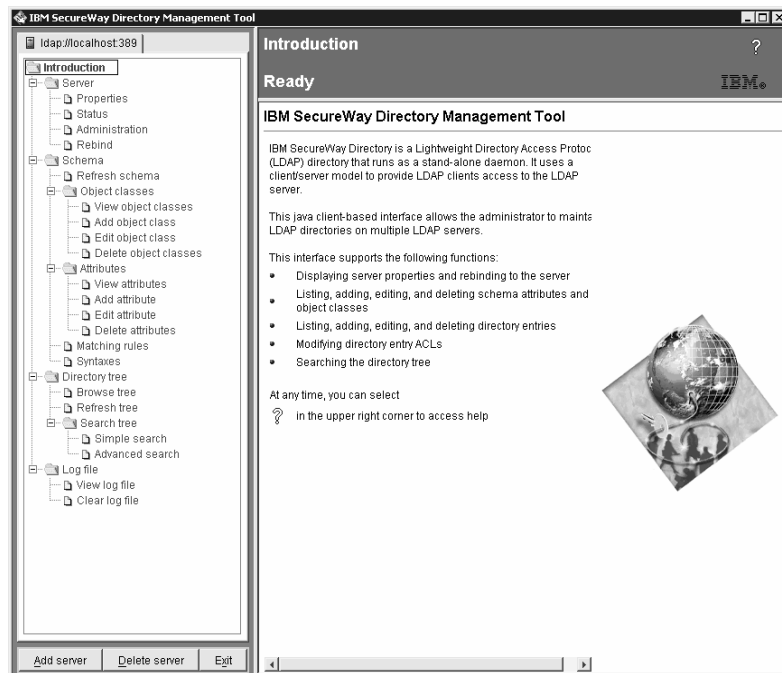


图 20-4 SecureWay Directory Administration Tool

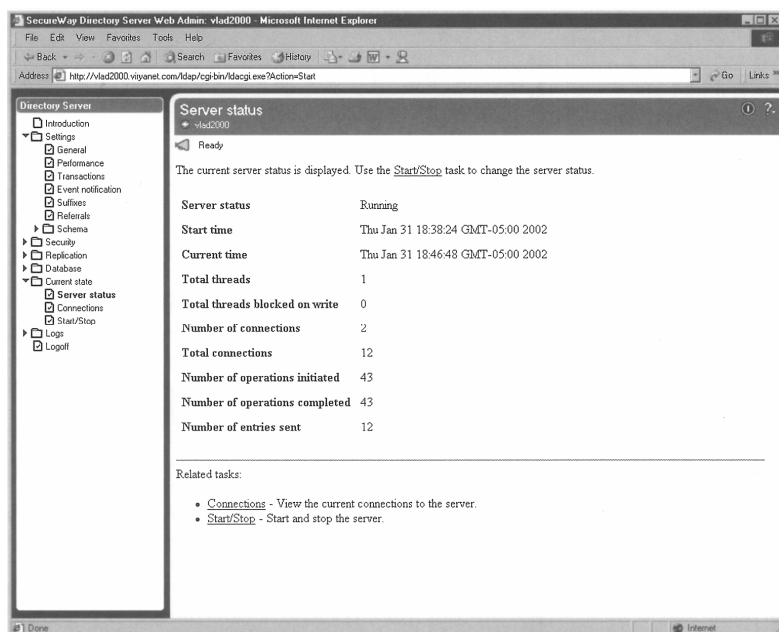


图 20-5 SecureWay 管理客户

20.5.2 WebSphere 用户注册表

现在的目标是要确保在前一节中创建的所有用户都可以作为 WebSphere 用户被访问。需要做的工作就是告知 WebSphere 应该使用 SecureWay 服务器作为自己的用户注册表。为此，需要告知 WebSphere 使用 LDAP 进行认证。

1. 在 Directory Management Tool 中，进入 Security Center 并选择 Authentication 标签，如图 20-6 所示。

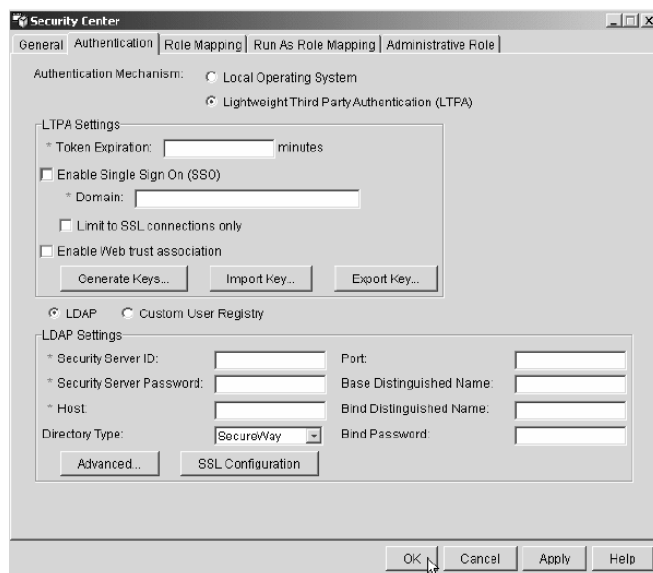


图 20-6 为 WebSphere 用户注册表指定 SecureWay

2. 将 Authentication Mechanism 设置为 Lightweight Third Party Authentication(LTPA)。
3. 然后，选中 LDAP 选项，将 Directory Type 设置为 SecureWay，并且填写设置，包括 SecureWay 主机名、Port（默认为 389）以及在前一节中建立的后缀。

20.6 小 结

LDAP 用来访问联机目录服务。它直接在 TCP 上运行，并且可用来访问独立的 LDAP 目录服务，或者访问 X.500 支持的目录服务。目录类似于数据库，但目录倾向于包含更具描述性、基于属性的信息。通常，目录中的信息更经常被读取，而较少被写入。因此，目录一般不实现普通数据库中为了执行大量复杂的更新操作而使用的事务或回滚方案。

LDAP 目录服务模型基于表项 (entry)。表项是一个带有区别名的属性集合，区别名用来明

确地指定表项。表项的每个属性都有一个类型 (type) 与一个或多个值。LDAP 目录服务基于客户-服务器模型。一个或多个 LDAP 服务器包含组成 LDAP 目录树的数据。LDAP 客户连接到 LDAP 服务器，并向 LDAP 服务器询问问题。服务器作出响应，或者给出一个指针进行响应，客户能够根据该指针获取更多信息 (指针通常是另一个 LDAP 服务器)。无论连接到哪个 LDAP 服务器上，客户都看到关于目录的相同视图：在一个 LDAP 服务器上表示的名称与另一个 LDAP 服务器上相同的名称所引用的表项是相同的。这是全局目录服务 (例如 LDAP) 的一个重要特点。

LDAP 非常重要，因此在 WebSphere 中实现了 LDAP 接口，并且使用 SecureWay 实现了利用 LDAP 实现的安全性 (在第 19 章已详细讨论)。由于 LDAP 不仅在维护安全性信息以及用户 profile 方面越来越重要，而且在维护任何需要在多个应用程序之间共享的属性方面也变得越来越重要，使得 LDAP 成为 WebSphere 的核心服务。虽然目前还有许多 WebSphere 用户没有使用 LDAP，但我们预测，在今后的 2 年中这种情况会得到改变。特别是对于那些在复杂企业环境中实现应用程序的用户，由于在这种复杂企业环境下许多应用程序需要以协作的方式共同工作，使这种情况更是如此。

第 21 章将讨论日志 API，它是 WebSphere 提供的另一种核心服务。

第 21 章 WebSphere 应用程序日志

软件开发过程实际上就是一个著名的“编码-测试-调试”周期。在利用 WebSphere 开发 Web 应用程序与企业应用程序时，通过利用诸如 WebSphere Studio Application Developer 与 VisualAge for Java 这样的工具，能够在一定程度上缩短这种周期。

这种内置的调试器对于测试与调试代码具有很重要的价值。但不幸的是，在部署之后才能检测到一些 bug。目前，使用诸如 VisualAge for Java 这样的工具通常比较困难。其原因是多样的，例如，该工具并非总是在部署网站时可用，使用 VisualAge for Java 不能使问题完全重复的出现（例如，有的问题仅仅在集群环境下出现），需要调试一个远程应用程序……但最基本的要求是，在实际使用 WebSphere Application Server 而不仅仅是使用开发环境时，可能需要使用某些调试工具。

最简单也可能是最常用的调试方法就是在代码中放置一些输出语句。当编写服务器方代码时，使用这种方法并不好（多么保守的说法！），因为有几个应用程序并发地运行并且将输出到相同的日志文件。

一个更系统的方法就是使用日志实用程序进行调试。为了解决这一需求，IBM 提供了 Logging Toolkit for Java，它是可以从 www.alphaworks.ibm.com/tech/loggingtoolkit4j 获取的 alphaWorks 的一部分。Logging Toolkit for Java 提供了一个包括日志程序、处理程序、过滤器、格式化程序在内的全面的体系结构。日志程序用来产生将要被日志的数据，它可以有两种类型：消息日志程序与跟踪日志程序。消息日志程序（message logger）由终端用户或系统管理员所使用，而跟踪日志程序（trace logger）则由开发人员在代码中使用。处理程序接收将要被日志的数据，处理程序可以附加在一个日志程序、文件或套接字上。过滤器（filter）用来控制哪些数据被显示，从而允许用户关心那些自己感兴趣的数据。格式化程序负责以某种特定的格式表现数据。

WebSphere 包括一个关于该日志工具箱的特殊版本，称为 JRas。需要指出的是，IBM 并不支持在 WebSphere 中使用非 WebSphere 版本的该日志工具箱。

21.1 JRas 消息日志与跟踪实用程序

JRas 对两种可以由代码产生的文本信息进行区分，分别是：消息与跟踪。消息（Message）是默认地产生的，主要由终端用户和系统管理员使用。可以对这些消息进行本地化，使得消息所用的语言与用户的区域相匹配。跟踪（Trace）主要由开发人员所使用。跟踪包括关于应用程序当前状态或关于某个运行时错误的详细技术信息。跟踪信息通常不是由应用程

序产生的，而且只有在调试过程中才启动跟踪功能。

JRas 包实现了日志程序、处理程序与格式化程序。它也提供了称为 `manager` 的对象，该对象在实质上包含关于日志程序、处理程序与格式化程序的配置。

消息与跟踪文本都记录在日志文件中。所有的消息都被记录在一个单独的文件中，该文件被 Web 服务器与任何应用程序所共享。WebSphere 日志记录在 `default_server_stdout` 文件中，该文件驻留在 WebSphere 安装目录的 `logs` 目录下。下面给出了一个日志文件样例：

```
IBM WebSphere Application Server, Release 4.0.1
Advanced Single Server Edition for Multiplatforms
Copyright IBM Corp., 1997-2001

[02.01.15 19:11:29:961 PST] 5dd676de Server      A WSVR0027I: Temporary
product license expires on Jul 13, 2002
***** Start Display Current Environment *****
WebSphere AEs 4.0.1 a0131.07 running with process name THINKPAD/Default Server
and process id 2432
Host Operating System is Windows 2000, version 5.0
Java version = J2RE 1.3.0 IBM build cn130-20010609 (JIT enabled: jitc), Java
Compiler = jitc
server.root = C:\WebSphere\AppServer
Java Home = C:\WebSphere\AppServer\java\jre
ws.ext.dirs =
C:\WebSphere\AppServer\java\lib;C:\WebSphere\AppServer\classes;C:\WebSphere\
AppServer\lib;C:\WebSphere\AppServer\lib\ext;C:\WebSphere\AppServer\
web\help;.\lib\ifxjdbcx.jar;.\lib\ifxjdbc.jar
Classpath = C:\WebSphere\AppServer\properties;C:\WebSphere\AppServer\
lib\bootstrap.jar
Java Library path = C:\WebSphere\AppServer\java\bin;.;C:\WINNT\System32;
C:\WINNT;C:\WebSphere\
AppServer/bin;
Current trace specification =
***** End Display Current Environment *****
[02.01.15 19:11:32:304 PST] 5dd676de Server      U Version : 4.0.1
[02.01.15 19:11:32:304 PST] 5dd676de Server      U Edition: Advanced Single
Server Edition for Multiplatforms
[02.01.15 19:11:32:304 PST] 5dd676de Server      U Build date: Thu Aug 09
00:00:00 PDT 2001
[02.01.15 19:11:32:304 PST] 5dd676de Server      U Build number: a0131.07
[02.01.15 19:11:42:459 PST] 5dd676de DrAdminServer I WSVR0053I: DrAdmin
available on port 7000
[02.01.15 19:11:43:059 PST] 5dd676de ResourceBinde I WSVR0049I: Binding
Session Persistence datasource as jdbc/Session
[02.01.15 19:11:44:121 PST] 5dd676de ResourceBinde I WSVR0049I: Binding
```


Increment Bean Datasource as jdbc/SampleDataSource
[02.01.15 19:11:44:191 PST] 5dd676de ResourceBinde I WSVR0049I: Binding
Samples IDB Datasource as jdbc/sample
[02.01.15 19:11:44:251 PST] 5dd676de ResourceBinde I WSVR0049I: Binding
Petstore IDB Datasource as ps/PetStoreDataSource
[02.01.15 19:11:48:517 PST] 5dd676de EJBEngine I WSVR0037I: Starting EJB
jar: Increment Bean Jar
[02.01.15 19:11:52:783 PST] 5dd676de SystemOut U Enhydra InstantDB -
Version 3.26
[02.01.15 19:11:52:783 PST] 5dd676de SystemOut U
[02.01.15 19:11:52:783 PST] 5dd676de SystemOut U The Initial Developer
of the Original Code is Lutris Technologies Inc.
Portions created by Lutris are Copyright (C) 1997-2001 Lutris Technologies,
Inc.
All Rights Reserved.
[02.01.15 19:11:54:616 PST] 5dd676de EJBEngine I WSVR0037I: Starting EJB
jar: C:\WebSphere\AppServer\installedApps\Samples.ear/TimeoutEJBBean.jar
[02.01.15 19:11:57:440 PST] 5dd676de EJBEngine I WSVR0037I: Starting EJB
jar: Increment EJB Module
[02.01.15 19:11:57:610 PST] 5dd676de EJBEngine I WSVR0037I: Starting EJB
jar: Hello EJB Module
[02.01.15 19:12:03:299 PST] 5dd676de ServletEngine A SRVE0161I: IBM WebSphere
Application Server - Web Container. Copyright IBM Corp. 1998-2001
[02.01.15 19:12:03:359 PST] 5dd676de ServletEngine A SRVE0162I: Servlet
Specification Level: 2.2
[02.01.15 19:12:03:359 PST] 5dd676de ServletEngine A SRVE0163I: Supported
JSP Specification Level: 1.1
[02.01.15 19:12:03:569 PST] 5dd676de ServletEngine A SRVE0167I: Session
Manager is Configured - Initializing...
[02.01.15 19:12:04:220 PST] 5dd676de CacheManager A DYNA0011E: Servlet cache
file dynacache.xml not found; caching is disabled
[02.01.15 19:12:04:370 PST] 5dd676de ServletEngine A SRVE0169I: Loading Web
Module: Default Application.
[02.01.15 19:12:05:011 PST] 5dd676de WebGroup I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[02.01.15 19:12:05:091 PST] 5dd676de WebGroup I SRVE0091I: [Servlet LOG]:
InvokerServlet: init
[02.01.15 19:12:05:191 PST] 5dd676de ServletEngine A SRVE0169I: Loading Web
Module: Examples Application.
[02.01.15 19:12:05:442 PST] 5dd676de WebGroup I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[02.01.15 19:12:05:462 PST] 5dd676de WebGroup I SRVE0091I: [Servlet LOG]:
SimpleFileServlet: init
[02.01.15 19:12:05:482 PST] 5dd676de WebGroup I SRVE0091I: [Servlet LOG]:

```

DirectoryBrowsingServlet: init
[02.01.15 19:12:05:782 PST] 5dd676de ServletEngine A SRVE0169I: Loading Web
Module: admin.
[02.01.15 19:12:06:333 PST] 5dd676de WebGroup      I SRVE0091I: [Servlet LOG]:
JSP 1.1 Processor: init
[02.01.15 19:12:06:343 PST] 5dd676de WebGroup      I SRVE0091I: [Servlet LOG]:
SimpleFileServlet: init
[02.01.15 19:12:06:403 PST] 5dd676de WebGroup      I SRVE0091I: [Servlet LOG]:
action: init
[02.01.15 19:12:09:457 PST] 5dd676de HttpTransport A SRVE0171I: Transport
http is listening on port 9,080.
[02.01.15 19:12:12:722 PST] 5dd676de HttpTransport A SRVE0171I: Transport
https is listening on port 9,443.
[02.01.15 19:12:12:732 PST] 5dd676de HttpTransport A SRVE0171I: Transport
http is listening on port 9,090.
[02.01.15 19:12:12:752 PST] 5dd676de Server        A WSVR0023I: Server Default
Server open for e-business

```

类似地，所有跟踪都被记录在一个单独的文件中，无论这些跟踪来自服务器还是来自应用程序。这两种文件都可以使用 WebSphere Log Analyzer 进行检查，该工具在第 2 章简要地提到过，本章后面的 21.6 节“Log Analyzer”将对该工具进行详细描述。

21.2 JRas 管理器

WebSphere 中关于 JRas 的代码在 `com.ibm.websphere.ras.Manager` 类中。这个类负责创建与维护日志程序，并且它提供了一个称为 `getManager()` 的静态方法，用来检索该类的单个实例。使用这个实例，就可以创建消息日志程序或跟踪日志程序。该管理器负责维护所创建的日志程序，并且如果某个请求需要先前所创建的日志程序，那么它将返回该日志程序，而不是创建新日志程序。

日志程序是一个指定名称的对象，它的名称由 4 个不同的子部件字符串组成，分别是：组织、产品、组件、日志程序名。管理器也支持一个更简单的名称空间，它仅仅由组件与日志程序名组成，但这个名称空间在 IBM 内部使用。在给定这 4 个部件字符串之后，就可以使用下列方法创建一个消息日志程序：

```

com.ibm.ras.RASMessageLogger createRASMessageLogger(String organization,
String product, String component, String name);

```

可以使用下列方法创建一个跟踪日志程序：

```

com.ibm.ras.RASTraceLogger createRASTraceLogger(String organization,
String product, String component, String name);

```

显然，在设置名称时必须非常仔细，以避免名称空间冲突。推荐用户为每个类仅仅使用一个日志程序，并且将这个类的名称与日志程序名相关联。

跟踪日志程序可以被归组从而形成逻辑组。管理器为此提供了下列方法：

```
void addLoggerToGroup(com.ibm.ras.RASTraceLogger logger, String groupName);
```

这种归组允许用户同时跟踪几个相关的组件。组名称必须对应用程序是惟一的，并且必须不能与其他应用程序相关联的组冲突。

21.3 JRas 日志程序

消息日志程序与跟踪日志程序都从 `RASILogger` 接口继承。该接口提供了日志程序的基本功能，并且继承了下列用于检索与设置日志程序名的方法：

```
public String getName();  
public void setName(String name);
```

它们也定义了 `isLoggable` 方法，用来判断该日志程序是否为某个类型的事件而启用。这个方法签名如下：

```
public Boolean isLoggable(long type)
```

参数类型中包含一个有效的消息或跟踪事件类型。JRas 支持 3 种消息类型，均在 `RASIMessageEvent` 接口中定义。表 21-1 列出了这些类型，并且给出了它们到 WebSphere 消息类型的映射。

表21-1 WebSphere中的JRas消息类型

JRas 消息类型		相应的 WebSphere 消息类型	
全名	缩写	高级版	企业版
TYPE_INFORMATION	TYPE_INFO	Audit	Informational
TYPE_WARNING	TYPE_WAN	Warning	Warning
TYPE_ERROR	TYPE_ERR	Error	Error

JRas 跟踪类型由 `RASITraceInterface` 提供。这个接口提供了两种跟踪类型：带等级的与不带等级的。有 3 个带等级的跟踪类型：`level1`、`level2` 与 `level3`。不带等级的跟踪事件对应于特定类型的跟踪事件。表 21-2 列出了这两种跟踪事件，并且给出了在 WebSphere 中的相应跟踪类型。

表21-2 WebSphere中的JRas跟踪类型

JRas 跟踪类型	相应的 WebSphere 跟踪类型
TYPE_LEVEL1	Event
TYPE_LEVEL2	Entry/Exit
TYPE_LEVEL3	Debug
TYPE_ERROR_EXC	Event
TYPE_OBJ_CREATE	Event
TYPE_OBJ_DELETE	Event
TYPE_SVC	Event
TYPE_API	Entry/Exit
TYPE_CALLBACK	Entry/Exit
TYPE_ENTRY_EXIT	Entry/Exit
TYPE_PRIVATE	Entry/Exit
TYPE_PUBLIC	Entry/Exit
TYPE_STATIC	Entry/Exit
TYPE_MISC_DATA	Debug

21.3.1 消息日志程序

消息日志程序不同于跟踪日志程序，表现在它们支持消息文本的本地化。RASIMessageLogger 接口提供了支持消息日志本地化的方法。可本地化的消息日志是基于标准 Java 资源束得到的，将在第 39 章中详细讨论。资源束是能够为不同区域提供的消息目录。每条消息都有一个用来标识该消息的关键字，并且每条消息都可以有嵌入在消息文本中的可选对象。下面给出了 RASIMessageLogger 接口的方法：

- String getMessageFile()：返回当前使用的资源束的文件名。
- void setMessageFile(String)：设置消息文本资源束。
- void message (long type, Object obj, String methodName, String key 与 Object[]-params)：发出一条消息。这个方法参数如下：
 - type 是表 21-1 中显示的一个类型。
 - obj 代表一个类名（除非首先传递一个字符串，否则日志程序将调用 obj.getClass().getName()）。
 - methodName 是一个方法名。
 - key 是可本地化消息的消息关键字，它与 setMessageFile 方法中指定的消息文件结合使用。

- param 代表将要被替换到消息文本中的对象。
- void msg (long type, Object obj, String methodName, String key, String file 与 Object[]params) : 发出一条消息。该方法的参数与前一个方法的参数完全相同, 惟一的不同就是可以使用参数 file 指定资源束。
- void textMessage (long type, Object obj, String methodName, String text 与 Object[]params) : 发送一条消息, 该消息的文本在参数 text 中指定, 而不是通过资源束指定。当然, 这就意味着不能提供本地化支持。这个方法的其他参数的含义与 message 方法相同。

需要说明的是, 为了使用方便, message, msg 与 textMessage 方法还提供了多种变体, 这些变体的不同表现在 params 参数上。这个 API 为这三方法分别定义了关于不使用参数、使用一个参数、使用两个参数以及使用参数数组的各种变体。其中, 最后一种形式 (使用参数数组) 就是分别为上面这三个方法讨论的情况。

21.3.2 跟踪日志程序

跟踪日志程序由 RASITraceLogger 接口表示。这个接口提供了用来生成不同类型的跟踪的几个方法, 包括: entry, exit, exception 与 trace。

entry 方法与 exit 方法的参数和 RASIMessageLogger.textMessage 方法的参数非常相似。这些参数包括跟踪类型 (表 21-2 所显示的类型之一)、在其上可调用 getClass().getName() 的类名或对象、方法名、附加的可选参数 (可以有 0 个、1 个、2 个或关于这些参数的数组)。

trace 方法将文本消息写为跟踪事件。其参数类似于 entry 与 exit, 不同之处是, text 字符串参数添加在可选的参数之前。

exception 方法使用跟踪类型、类名字符串 (或一个相应的对象)、方法名、异常对象作为参数。然后, 该方法将把异常记录到此跟踪日志中。

RASITraceLogger 接口使用一个内部的掩码来指定哪些跟踪事件被日志, 从而支持跟踪过滤的概念。可以分别使用 getTraceMask 方法与 setTraceMask 方法获取和设置这个掩码。其中后一个方法由 WebSphere Application Server 基于指定的跟踪设置来调用。

21.4 WebSphere 中的跟踪设置

消息总是连续不断地被日志, 而在默认情况下, 跟踪不被日志。为了控制对跟踪的日志, WebSphere Administrative Console 在应用程序服务器下提供了 Trace Service 表项。这个表项允许用户控制跟踪机制的行为, 如图 21-1 所示。

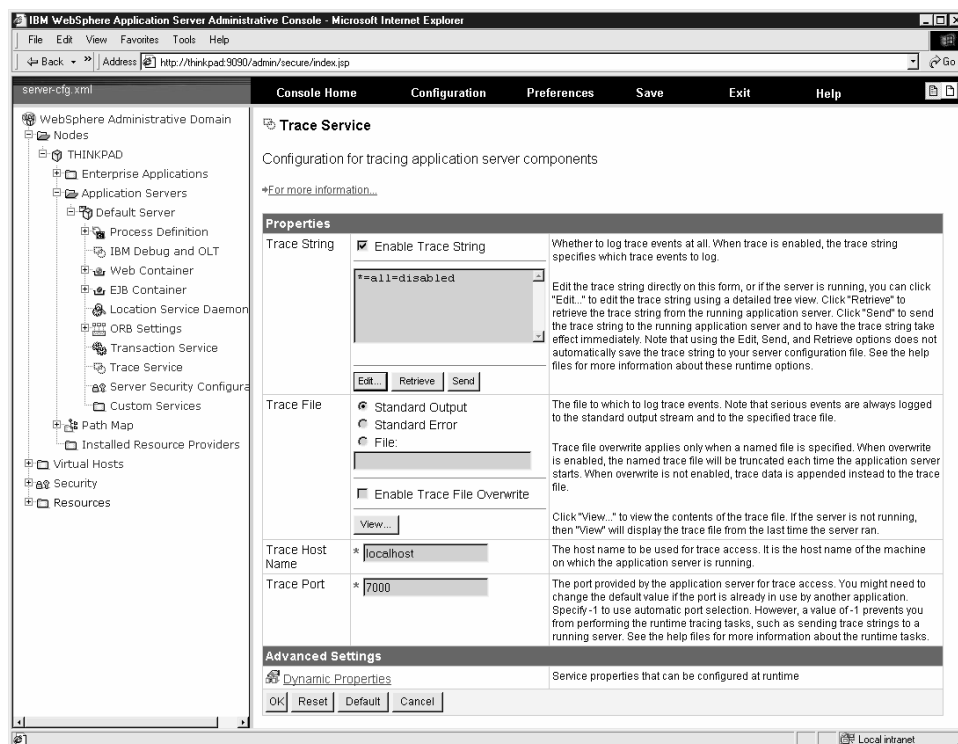


图 21-1 WebSphere Administrative Console 中的 Trace Service

下面给出了可以设置的参数。

- **Trace String（跟踪字符串）**：这个字符串决定什么跟踪被日志。字符串的格式为 `component=type=action`，其中，`component` 是组件名，`type` 是消息的类型（`debug`，`entry/exit` 或 `event`），`action` 可以是 `enable`，`disable` 或 `inherit`。例如，为了日志所有与 IBM 的 `servlet` 个性化功能相关的调试跟踪，就应该添加 `com.ibm.servlet.personalization.*=debug=enabled` 这一行。虽然用户可以直接编辑这个字符串，但操作台提供了一个友好的用户界面用来进行该字符串的编辑。
- **Trace File（跟踪文件）**：决定了跟踪信息将要被写入到的文件。可以选择标准输出、标准错误或任何其他文件。
- **Trace Host Name（跟踪主机名）**：应用程序服务器正在其中运行的主机名。
- **Trace Port（跟踪端口）**：用于跟踪的端口。默认为 7000，但如果任何其他应用程序正在使用该端口，那么可以对它进行修改。

为了简化为不同组件指定跟踪级别的任务，Administrative Console 提供了一个特殊的界面，如图 21-2 所示。在 Trace String 域（见图 21-1）上单击 Edit 按钮，就将启用该界面。

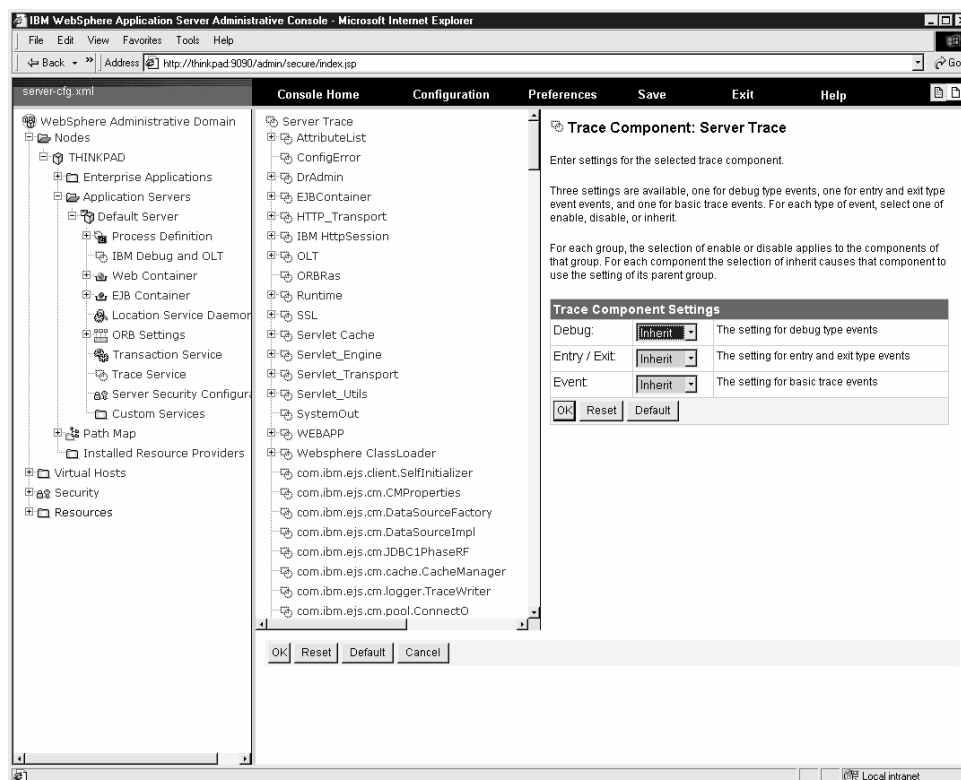


图 21-2 跟踪服务设置

在编辑跟踪字符串时，Administrative Console 提供了一棵树，用来浏览可跟踪的各种组件。一旦选择了某个组件，就可以启用、禁用或“继承”三个不同级别的消息（对应于表 21-2 中所显示的级别）。选择 Inherit 就意味着组件的行为与组件层次（例如，对于类，就是相关联的包）中直接父组件的行为相同，或者与该组件所属的组行为相同。

21.5 Web 应用程序日志示例

为了演示 Web 应用程序日志的用法，下面将分析一个使用跟踪的 servlet 的简单例子。这个 servlet 识别在 HTTP 请求中的 2 个参数：enum 与 denom。它将输出 enum 被 denom 除之后得到的比率。

在 servlet init 方法中创建跟踪日志程序，使其具有组织名、产品名、组件名与完全的包类名。并且将这个日志程序添加到一个称为 CompleteReferenceBooks 的组中。

在 doGet 方法中，包括了用于日志 entry 与 exit 的代码。这一代码包围在 if 语句中，if 语句

检查 entry/exit 跟踪是否应该被日志。这样就使得在运行时能够节省一些操作（例如它节省了 entry 跟踪中 params 数组的构建操作）。如果在 doGet 方法中捕获了一个异常，那么使用跟踪日志程序的 exception 方法将该异常记录下来。

```
package book.websphere.ch21;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import com.ibm.ras.*;
import com.ibm.Websphere.ras.*;

public class TracerServlet extends HttpServlet {
    RASITraceLogger traceLog;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        com.ibm.websphere.ras.Manager manager =
            com.ibm.websphere.ras.Manager.getManager();
        traceLog = mgr.createRASITraceLogger("Osborne", "CompleteReference",
            "websphere", "book.websphere.ch21.TracerServlet");
        manager.addLoggerToGroup(traceLog, "CompleteReferenceBooks");
    }

    public void doGet(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        try {
            String enum = request.getParameter("enum");
            String denom = request.getParameter("denom");
            if (traceLog.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT)) {
                Object[] parms = { enum, denom };
                traceLog.entry(RASITraceEvent.TYPE_ENTRY_EXIT, this, "doGet",
                    parms);
            }
            response.setContentType("text/html");
            int enumInt = (new Integer(enum)).toInt();
            int denomInt = (new Integer(denom)).toInt();
            PrintWriter out = new PrintWriter (response.getOutputStream());
            out.println("<html>");
            out.println("<head><title>TracerServlet</title></head>");
        }
    }
}
```



```
        out.println("<body>");
        out.println("<H1>" + enum + " divided by " + denom + " is: <BR>");
        out.println((enum / denom) + "</H1>");
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
    catch (Exception ex) {
        traceLog.exception(RASITraceEvent.TYPE_ERROR_EXC, this,
            "doGet", ex);
    }
    if (traceLog.isLoggable(RASITraceEvent.TYPE_ENTRY_EXIT)) {
        traceLog.exit(RASITraceEvent.TYPE_ENTRY_EXIT, this, "doGet");
    }
}
}
```

21.6 Log Analyzer

虽然由 JRas 产生的日志与跟踪能够被人类直接阅读，但 WebSphere 还是提供了一个图形化的日志分析工具。Log Analyzer（如图 21-3 所示）读取一个或多个活动或跟踪日志，合并所有这些跟踪，然后显示所有归组在 UOW（UnitOfWork）元素下的表项。需要说明的是，Log Analyzer 不能分析消息日志；因为这些消息日志的内容是由资源束控制的，并且具有本地化特点，使得这些日志可能有任意的格式。

除了标准的状态栏、工具栏与菜单栏之外，Log Analyzer 的主窗口还包括 3 个面板。在这个窗口的左边，该面板显示了日志记录列表，这些记录在默认情况下是根据它们的 Units of Work 进行归组的。每个 UOW 都有一个文件夹图标，在该图标下显示了所有与它相关联的跟踪记录。没有 UOW 标识的记录被归组在树的一个单独文件夹中。UOW 文件夹是按照时间戳进行排序的，其中最近出现的 UOW 位于树的顶部。类似地，UOW 中的表项是按照相反的顺序显示的。注意，时间排序是在 Log Analyzer 合并了来自几个文件（如果有几个文件被加载）的表项之后进行的。也可以按照某个字段（而不是 UOW）对日志记录进行排序，其方法是在 Analyzer Preferences 对话框（见图 21-4）中进行一个不同的选择。在这个对话框中，可以选择按照进程 ID、线程 ID、服务器名、函数名等选项进行文件归组。

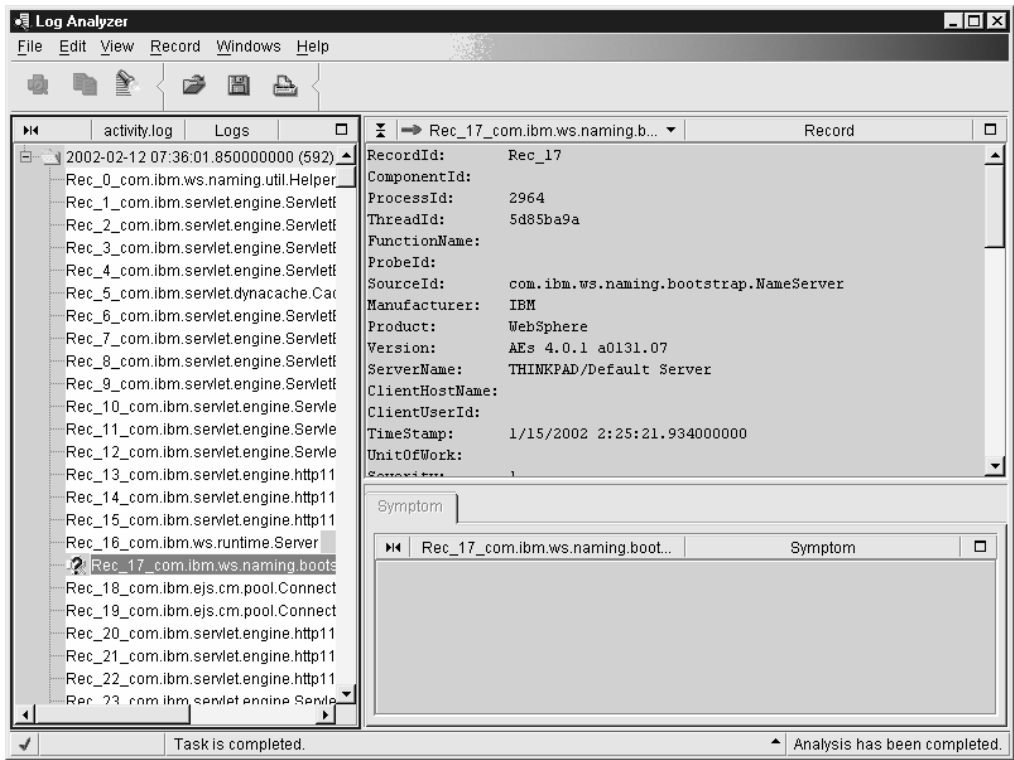


图 21-3 在 Log Analyzer 中查看合并后的跟踪信息

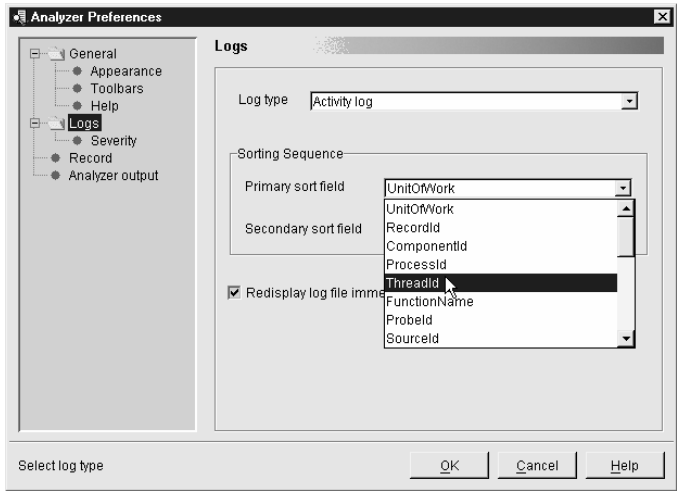


图 21-4 使用 Analyzer Preferences 对话框设置排序序列

为了便于快速标识，跟踪消息被使用不同的颜色编码，下表总结了这些颜色编码。用户也

可以修改这些颜色设置,方法是在左边面板中选择 General 选项,然后在 Analyzer Preferences 对话框中设置外观值。

安全性级别	被选中	没有被选中
1	Pink (粉红色)	Red (红色)
2	Yellow (黄色)	Green (绿色)
3	White (白色)	Blue (蓝色)

Log Analyzer 主窗口中的其他两个面板分别是记录面板与分析面板。记录面板显示了关于所选择记录的详细描述,从而提供所有被记录的信息。分析窗口显示在调用 Analyze 函数(在 Record 菜单下)时需要的附加信息,并且显示用户需要的某些其他信息。这些附加信息可能是 Log Analyzer 从通用 WebSphere 错误数据库中找到的某些错误故障信息。

注意,在应用 Analyze 函数之后,树中每个被分析的记录表项都有一个小的指示图标。每个图标的含义如下表所示。

图标	含义
	在 Analysis 窗口中可以得到某些分析信息
	在 Analysis 窗口中可以得到某些分析信息,但这个表项与前面出现的某个问题相关。可能需要参看前面的日志文件以对问题进行诊断
	这个表项的问题严重性级别为 1 或 2,但它没有任何分析信息
	这个表项的问题严重性级别为 3,但它没有任何分析信息

21.7 小 结

本章讨论了 WebSphere 与 JRas 工具箱中可用的日志功能与跟踪功能,它是本书第 4 部分的最后一章。下面,我们将讨论转移到在 WebSphere 上开发与部署应用程序时所使用的结构上。第 5 部分讨论的是 WebSphere 中的 servlet 与 JSP,在第 6 部分将讨论 EJB。这两个部分关注的是开发人员可用的应用程序框架与支持。

第 5 部分 开发 servlet 与 JSP

第 22 章 HTTP servlet

通过本书前面对 servlet 生命周期的介绍以及对一个简单“Hello World”示例的描述，相信读者已经熟悉了什么是 servlet 以及它们是如何工作的，那么，让我们转移到关于 servlet 的下一个学习阶段。本章给出了最常见 servlet（通过 HTTP 连接访问的 HTTP servlet）的 3 个示例。本章的目标有两点：显示在现实世界中如何使用 servlet 的例子；说明如何使用 WebSphere servlet。每个例子都演示了一个端到端的过程，包括生成某种形式的用户界面以及访问存储在邮件服务器或 RDBMS（Relational Database Management System，关系数据库管理系统）中的数据。

22.1 读取邮件的 servlet

本章第一个例子是现实生活中基于 Web 的服务的实现，该服务允许用户使用 Web 浏览器（而不是使用电子邮件程序，例如 Microsoft Outlook）通过 Web 读取电子邮件。这些服务在 Web 上是免费的，例如在 www.mail2web.com 上就是如此。

为了使这个邮件读取服务样例对任何 Web 浏览器可用，需要构建一个 HTTP servlet 来为邮件请求服务。这个 HTTP servlet 将接收电子邮件请求，访问 POP3 邮件服务器以检查邮件，并返回某些用户界面用以查看邮件。这个 servlet 使用 JavaMail API，它是 J2EE 的一部分，已经在第 15 章对它进行了介绍。

该 servlet 使用 4 个不同的参数，分别用于电子邮件用户名、POP3 服务器、POP3 口令以及操作（servlet 用来标识被请求的操作）。操作参数可以不要，或使用 detail 值（用以检索消息体）。servlet 参数出现在 servlet URL 中，并且可以通过 HttpServletRequest 对象被访问。下面给出了此 servlet 的代码：

```
package book.websphere.chapter22;

import javax.mail.*;
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import java.util.*;
import java.io.*;

public class MailReader extends HttpServlet {
    public void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        performTask(request, response);
    }

    public void doGetMail(HttpServletRequest req,
        HttpServletResponse resp)
    {
        PrintWriter pw = null;
        try {
            pw = resp.getWriter();
        }
        catch (Exception ex) {
            return;
        }
        try {
            String userName = req.getParameter("username");
            String pop3 = req.getParameter("pop3");
            String password = req.getParameter("password");

            session = Session.getInstance
(System.getProperties(),null);
            store = session.getStore("pop3");
            store.connect(pop3, userName, password);
            folder = store.getFolder("INBOX");
            folder.open(Folder.READ_ONLY);
            messages = folder.getMessages();

            resp.setContentType("text/html");
            pw.println("<HTML><HEAD><TITLE>Your" +
                "Mail</TITLE></HEAD><BODY><TABLE>" +
                "BORDER=\"1\"><TR>");

            pw.println("<TH>From</TH><TH>Date</TH><TH>Subject</TH>");
            for (int i = 0; i< messages.length; i++) {
                pw.println("<TR><TD>" +
                    messages[i].getFrom()[0] + "</TD>");
                pw.println("<TD>" + messages[i].getSentDate())
```

```
        + "</TD>");
        pw.println("<TD><A HREF='" +
            req.getRequestURI() +
            "?operation=detail&msgnum=" + i +
            "'>" +
            messages[i].getSubject() +
            "</A></TD></TR>");
    }
    pw.println("</TABLE><P><A HREF='" +
        HttpUtils.getRequestURL(req) +
        "?operation=logoff'" +
        ">Log-Off</A></P></BODY></HTML>");
    pw.close();
}
catch (Exception ex) {
    ex.printStackTrace(pw);
}
}

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

public void performTask(HttpServletRequest request,
    HttpServletResponse response) {
    try {
        String operation = request.getParameter("operation");
        String userName = request.getParameter("username");

        if (operation == null) {
            if (userName == null)
                sendOutPage(response.getWriter());
            else
                doGetMail(request, response);
        }
        else if (operation.equals("detail")) {
            doGetDetail(request, response);
        }
        else if (operation.equals("logoff")) {
            doLogoff(request, response);
        }
    }
}
```

```
catch(Exception ex)
{
    ex.printStackTrace();
}
}

public void sendOutPage(PrintWriter pw)
{
    pw.println("<HTML><HEAD><TITLE>Mail Reading " +
        "Servlet</TITLE></HEAD>");
    pw.println("<BODY><H1>Welcome to the Mail"+
        " Reading Servlet</H1>");
    pw.println("<FORM ACTION=" +
        "'http://yourhost/ch22/MailReader'"+
        "METHOD='GET'>");
    pw.println("<B>Username: </B><INPUT TYPE='TEXT' " +
        "NAME='username'><BR>");
    pw.println("<B>POP3 Server: </B>" +
        "<INPUT TYPE='TEXT' NAME='pop3'><BR>");
    pw.println("<B>Password:</B><INPUT TYPE='TEXT' " +
        "NAME='password'><BR>");
    pw.println("<INPUT TYPE='SUBMIT'><INPUT " +
        "TYPE='RESET'></FORM></BODY></HTML>");
}

private Folder folder;
private Message[] messages;
private Session session;
private Store store;

public void doGetDetail(HttpServletRequest req,
                        HttpServletResponse resp)
{
    PrintWriter pw = null;

    try {
        pw = resp.getWriter();
    }
    catch (Exception ex) {
        return;
    }
    try {
        int num = Integer.parseInt(
            req.getParameter("msgnum"));
    }
```

```

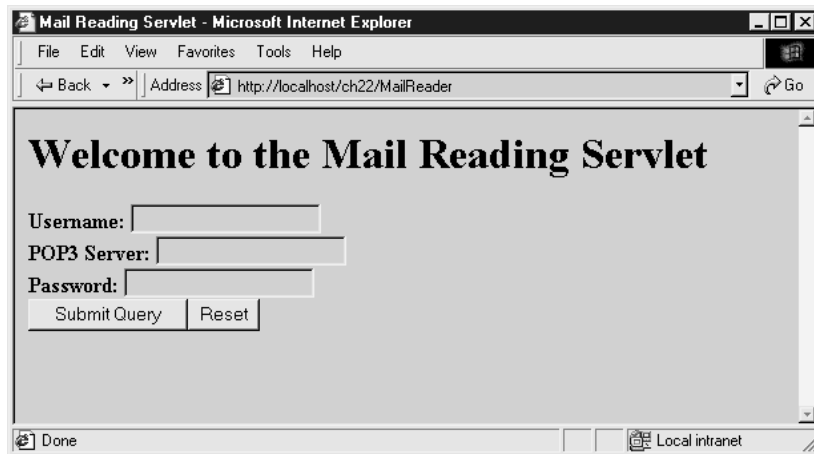
        String s;
        Message msg = messages[num];
        resp.setContentType("text/html");
        pw.println("<HTML><HEAD><TITLE>Reading Message:" +
            msg.getSubject() + "</TITLE>");
        pw.println("</HEAD><BODY><B>From:</B>" +
            msg.getFrom()[0]);
        pw.println("<B>To:</B>" +

            msg.getRecipients(Message.RecipientType.TO)[0]);
        pw.println("<B>Date:</B>" + msg.getSentDate());
        pw.println("<P>");
        BufferedReader rd = new BufferedReader(new
            InputStreamReader(msg.getInputStream()));
        while ((s = rd.readLine()) != null)
            pw.println(s);
        pw.println("</P></BODY></HTML>");
        pw.close();
    }
    catch (Exception ex) {
        ex.printStackTrace(pw);
    }
}

public void doLogoff(HttpServletRequest req,
    HttpServletResponse resp)
{
    try {
        folder.close(false);
        store.close();
    }
    catch (Exception ex)
    {
    }
}
}

```

当没有提供操作参数并且没有提供用户名时，程序将假设用户是第一次访问该 servlet，然后发出一个首次登录页面（如下图所示），它是一个简单的 HTML 表单，可以在这个表单中填写参数。



在单击 Submit Query 按钮之后，相同的 servlet 将再次被调用。在这第二次调用中，servlet 已经有了一个用户名（用作请求参数）。使用 JavaMail API，这个 servlet 就能够连接到 POP3 服务器并下载用户的电子邮件。

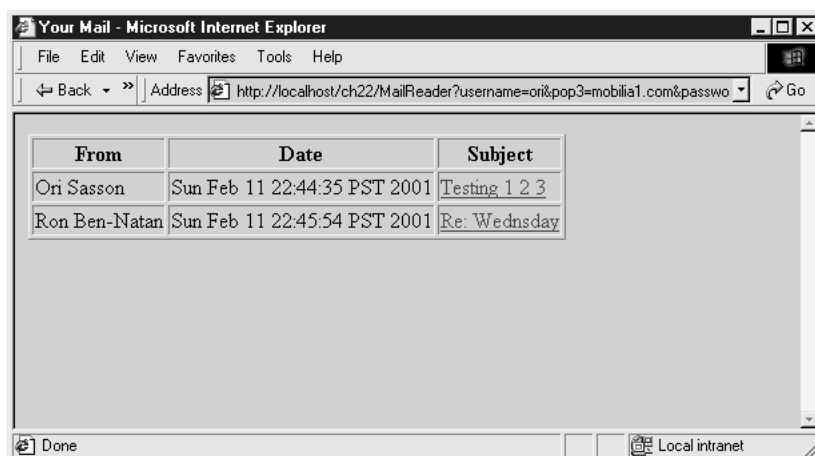
使用 Session 对象

可以使用 `Session.getInstance` 方法访问 Session 对象。注意，也可能为这个方法提供 Properties 参数。该参数可用来指定自己感兴趣的协议或实现。在本例中使用的是默认属性，只要在系统中没有特殊的设置存在，使用默认属性是很好的；这些设置所在的文件是 `java mail.providers`，`javamail.default.providers`，`javamail.address.map` 与 `javamail.default.address.map`。另外需要说明的是，`getInstance` 方法还有另一个参数——`authenticator`，在本例中没有使用。这个参数可用来提供更高级的认证方案，从而提高电子邮件安全性。Session 接口也提供了 `getDefaultInstance` 方法，该方法给出对默认（和共享）会话实例的访问。

利用 Session 对象，程序就能够获取 POP3 协议的一个 Store 对象。利用适当的参数（服务器、用户名、口令）调用 store 上的 `connect` 方法，就允许 servlet 从 store 中获取 INBOX 文件夹。在 Folder 对象被打开之后，可以使用 `getMessage` 方法获取电子邮件消息。

到达的电子邮件显示在一个表中，每个电子邮件消息占据一行。每行信息基于一个 Message 对象。消息 accessor 方法用来检索电子邮件的发信人、日期与主题。提供了一条 HTML 链接从消息主题连接到消息体。这个链接用来获取电子邮件消息的详细信息，并且需要与服务器进行另一个往返。为了构造这个请求的原始 URL（包括所有的参数），需要使用 `javax.servlet.http.HttpUtils` 类中的 `getRequestURL` 方法。这个类提供了一些静态实用程序方法，可用来编写 HTTP servlet 代码。下图演示了一个关于接收到的电子邮件的表，在发送表单数据之后，服务器输出这个表作为响应。需要说明的是，这仅仅是一个示例；在实际的

邮件读取程序中，不会将用户口令显示在 URL 上。



22.2 用于配置的 servlet

第二个例子来自层次式配置的应用。这个例子显示了一个关于复杂机械部件的配置。该机械按照一种层次式结构组成，在这种结构中，每个部分都可以被划分为更小的部分集合，而这些更小的部分又可以被继续划分。由于大多数人在某一时刻只能见到一部分细节，因此现实世界中几乎所有的对象都是按照这种层次式结构划分的。每个级别都可以被管理，并且观察者能够决定自己希望查看的部件级别。

例如，计算机硬件由主板、网卡、磁盘等组成。主板又可以进一步分为 CPU、内存芯片、控制器等。正如计算机硬件一样，软件也由一些部件组成：模块、过程、函数等。这样的层次式视图并不仅仅局限于 IT 相关的问题。任何事情，从工程到发电站，到飞机、城市——实际上几乎任何事情——都是以某种层次的结构维护的。

这样的划分通常维护在某种形式的数据库中，编程者需要提供关于这一配置的视图。显然，编程者希望将这些信息显示为层次式的——最佳的方法就是使用 tree widget。

在给定了表示层次式结构的数据模型之后，需要构建一个支持 Web 的界面，用来在浏览器中将这种层次显示为树。这样就允许任何有权访问网站的人以一种方便的形式查看配置。由于需要为任何浏览器提供此服务，因此可以构建一个 HTTP servlet 来服务这些请求。HTTP servlet 将接收请求，访问数据库中的数据，并返回某种用户界面以显示层次信息。

在开始编程之前，需要进行一些决策。服务器模型是什么？这个问题比较简单。最佳的模型就是使用 HTTP servlet。在这个例子中，将构建一个可接受 GET 请求的 servlet。在接收到这样的请求（以及标识配置根所需的参数集合）时，servlet 将访问数据库并获取与整个树相关的信息。然后，它把这些信息写入输出流中。

下一个问题就是：应该如何写这些信息？是否应该将它直接写为 HTML？对此问题的回答要困难一点。是否可以考虑将它放到一个 HTML 表中，并基于树的深度进行缩进？这样也许会好一点，但还不是很好。实际上，由于需要提供动态的用户界面，因此这里需要某些代码能够传递给浏览器，从而使树可以被显示和处理（也就是，它的节点可合上、展开等）。因此，能够进行的选择范围就缩小了：要么使用 JavaScript 与 DHTML 的组合，要么使用 Java applet。由于本书的主题与 Java 技术紧密结合，因此在这里将选择后者。

然后，下一个步骤就是编程了。人在本性上都是懒惰的，因此，当需要编写代码时，总是花费一定的时间试图在 Web 上找到可供借用的例子——甚至花钱购买。半个小时之后，就能够找到 10 个可用来完成所需任务的 applet（Web 是不是很神奇？）。在对可用的包进行一番比较后，决定选用 <http://www.jpowered.com/tree> 上称为 Advanced Tree Explorer 的包。这是一个最佳的 applet。因此，可以下载它（或一个 30 天的评估版）并阅读它的使用说明，这大约要 15 分钟的时间，这样就可以开始真正的工作了。

22.2.1 初始化 servlet

首先，构建 servlet 类与该类的 init 方法，如下所示：

```
package book.websphere.chapter22;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;
import java.sql.*;
import java.text.*;

public class ConfigServlet extends HttpServlet {
    static ResourceBundle aBundle = null;
    static String serialFormat;
    static String nonSerialRootFormat;
    static String serialRootFormat;
    static String nonSerialFormat;
    static String bgImage = "";
    static String codebase;
    static String fontString;

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        aBundle = ResourceBundle.getBundle("ConfigServlet");
        serialFormat = aBundle.getString("serialFormat");
        nonSerialFormat = aBundle.getString("nonSerialFormat");
    }
}
```

```

        serialRootFormat = aBundle.getString("serialRootFormat");
        nonSerialRootFormat =
            aBundle.getString("nonSerialRootFormat");
        codebase = aBundle.getString("codebase");
        fontString = aBundle.getString("font");
        bgImage = aBundle.getString("bgImage");
        DBConnection.init(
            aBundle.getString("dburl"),
            aBundle.getString("dbuser"),
            aBundle.getString("dbpass"));
        try {
            Connection conn = DBConnection.getConnection();
            CallableStatement stmt = conn.prepareCall(
                "alter session set NLS_DATE_FORMAT = 'DD MON YYYY'"
            );
            stmt.execute();
            stmt.close();
        } catch (Throwable ex) {
            ex.printStackTrace();
        }
    }
}

```

每次当这个 servlet 被加载时（也就是当应用程序服务器被启动时）都将调用 `init` 方法。由于该 servlet 需要访问数据库，因此必须创建一个到达数据库的连接。实际上，由于创建数据库连接是要花费一定的时间的（事实上这也是耗时最长的操作之一），这样，在 servlet 启动过程中打开连接（这时没有用户在等待该操作）也就不足为怪了。在 `init` 中也处理仅仅发生一次的其他事情，例如读取属性文件以及设置后来在实际处理过程中使用的所有常量。需要说明的是，数据库连接通过一个简单的连接类进行，这将在后面显示。通常，当使用 WebSphere 时，将使用从连接池获取的连接，而不必由 servlet 管理这些连接。这一功能已在第 18 章中进行了详细的说明。

```

package book.websphere.chapter22;

import java.sql.*;

public class DBConnection {
    static Connection conn = null;
    static String aURL;
    static String aUserName;
    static String aPassword;

    public static void init(String url, String userName, String
        password) {

```

```
// Load the JDBC driver.
if (conn == null) {
    try {
        DriverManager.registerDriver(new
            oracle.jdbc.driver.OracleDriver());
    } catch (Exception e) {
        System.err.println("Could not load driver ");
        e.printStackTrace();
    }
    aURL = url;
    aUserName = userName;
    aPassword = password;
}

public static Connection createConnection() {
    try {
        // Establish the connection to the database.
        conn = DriverManager.getConnection(aURL, aUserName,
            aPassword);
    } catch (Exception e) {
        System.err.println("Could not create connection " +
            e.getMessage());
        e.printStackTrace();
        conn = null;
    }
    return conn;
}

public static Connection getConnection() {
    Connection toRet = null;
    if (conn == null)
        toRet = createConnection();
    else toRet = conn;
    return toRet;
}
}
```

这个例子使用来自 Oracle 的 JDBC 瘦驱动程序 (thin driver)；必须确保 WebSphere 服务器能够访问这个类，为此，需要在 Administrator 操作台将这个 JDBC 注册为 JDBC Resource，或者更新用来部署此 servlet 的 Web 模块的 classpath。如果没有执行这些动作，那么在试图向应用程序服务器注册这个 servlet 时将出现异常。当然，如果正在使用一个不同的数据库，那么应该使用相应的 JDBC 驱动程序。

22.2.2 为请求服务

下一步就是构建完成所有实际工作的方法。由于我们希望以最灵活的方式提供此服务，因此实现了 servlet 的 doGet 方法。这个方法从请求对象接受参数并继续构建配置树。该方法的代码如下：

```
public void doGet(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = new PrintWriter(response.getOutputStream());
    String nodeKey = request.getParameter("nodeKey");
    String rootSerialId = request.getParameter("serialsT");
    String rootPartId = request.getParameter("partId");
    String rootPartDesc = request.getParameter("partDesc");
    String rootRevision = request.getParameter("rev");
    String quantity = request.getParameter("qty");

    out.println("<html>");
    out.println("<head><title>Configuration View</title></head>");
    out.println("<body>");
    out.println("<applet code=\"TreeApplet\" width=\"250\"
        height=\"300\" +
        \"archive=\"Treemenu.jar\">");
    out.println("<param name=\"nodeOffset\" value=\"25\">");
    out.println("<param name=\"image1\" +
        \"value=\"closed|./IconImages/closedfolder.gif\">");
    out.println("<param name=\"image2\" +
        \"value=\"open|./IconImages/openfolder.gif\">");
    out.println("<param name=\"image3\" +
        \"value=\"document|./IconImages/document.gif\">");
    Stack displayStringStack = new Stack();
    Stack levelStack = new Stack();

    String rootName = formatRootBySerial(
        rootPartId, rootPartDesc, rootRevision, rootSerialId);
    out.println("<param name=\"node1\" value=\"conf|root|\" +
        rootName + \"|Helvetica|N|10|0|0|0|true|closed
        |open\"");
    displayStringStack.push(rootName);
    levelStack.push(new Integer(1));
```

```
// ... Build the tree in the database and read it in
// rs is the result set object returned from the
// database query - holding the nodes that need to
// go into the tree structure

while (rs.next()) {
    // read in the data per row and place it in variables
    // to be used in the following code
    // then, based on the level and the values on the stack
    // decide where in the tree the node belongs
    if (configLevel == atTop) {
        String itemString = "item" +
            Integer.toString(itemNum++);
        String dispString = ((String)displayStringStack.peek())
            + "/" + dString;
        out.println("<param name=\"" + itemString +
            "\" value=\"" +
            dispString);
    } else if (configLevel < atTop) {
        while (configLevel <
            ((Integer)levelStack.peek()).intValue()) {
            displayStringStack.pop();
            levelStack.pop();
        }
        String itemString = "item" +
            Integer.toString(itemNum++);
        String dispString = ((String)displayStringStack.peek())
            + "/" + dString;
        out.println("<param name=\"" + itemString +
            "\" value=\"" +
            dispString);
    } else { // configLevel > atop
        String itemString = "item" +
            Integer.toString(itemNum++);
        String dispString = ((String)displayStringStack.peek())
            + "/" + dString;
        out.println("<param name=\"" + itemString +
            "\" value=\"" +
            dispString);
        displayStringStack.push(dispString);
        levelStack.push(new Integer(configLevel));
    }
}
```

```

        stmt.close();
    } catch (Throwable ex) {
        ex.printStackTrace();
    }

    out.println("</applet>");
    out.println("</body></html>");
    out.close();
}

```

注意：此处忽略了一些数据库访问代码，因为此处并不关心它们；读者只要理解通过请求对象传递而来的参数是用来调用数据库操作（用于构建树）的即可。然后，将从数据库中读取此树并用来产生 applet 所需的参数。

这个方法首先读取发送给 servlet 的参数，以确定什么节点作为配置树的根。然后，servlet 开始设置输出流的内容类型（一个 HTML 页面）并写出 applet 标记。Advanced Tree Explorer applet 接收通过 PARAM 标记集影响其显示的参数。它也接收使用 PARAM 标记的树的数据。因此，这个 servlet 需要将常规的显示参数（由方法的第一部分完成）以及封装数据的参数（方法的其余部分）写到输出流中。

在常规参数写出之后，数据库就被访问以获取节点。数据库调用将以显示顺序返回这些节点，而不是按照树的排序（例如深度优先搜索或宽度优先搜索）返回这些节点。有一个栈对象被用来构建树结构（基于从数据库所返回的行上所标记的配置级）。然后，一个 PARAM 标记被添加到名为 nodeX 的 applet 上，其中，X 每行增加 1。通过将每个节点的名称连接成完全的路径（其前缀由栈所管理），就能够得到此树结构。这样，得到的结果 HTML 源代码如下：

```

<html>
<head>
<title>Install Base Configuration</title>
</head>
<body>
<applet code="TreeApplet" width="250" height="300" archive="Treemenu.jar">
<param name="nodeOffset" value="25">
<param name="image1" value="closed|./IconImages/closedfolder.gif">
<param name="image2" value="open|./IconImages/openfolder.gif">
<param name="image3" value="document|./IconImages/document.gif">
<param name="node1" value="conf|root|Root Configuration|Helvetica|
        N|10|0|0|0|true|closed|open">
<param name="node2" value="serialeee|conf|Serial #EEE|Helvetica|
        N|10|0|0|0|true|closed|open">
<param name="node3" value="pentium|serialeee|Pentium IV 2000|Helvetica|
        N|10|0|0|0|true|closed|document">

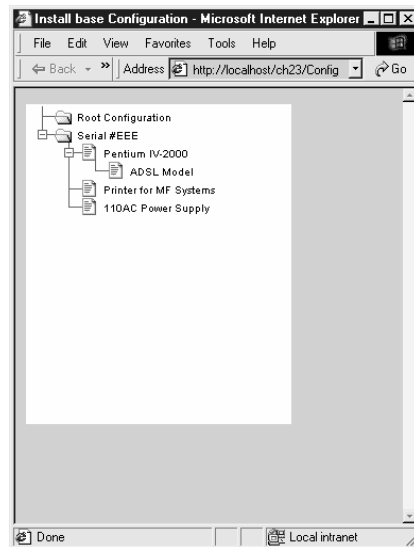
```



```
<param name="node4" value="modem|pentium|ADSL Modem|Helvetica|
    N|10|0|0|0|true|closed|document">
<param name="node5" value="printer|serialeee|Printer for MF Systems|
    Helvetica|N|10|0|0|0|false|document">
<param name="node6" value="powersup|serialeee|110AC Power supply|
    Helvetica|N|10|0|0|0|false|document">

</applet>
</body>
</html>
```

相应的 Web 页面如下所示：



22.3 FastTicker servlet

下面继续描述第三个（也是本章的最后一个）servlet。实际上，这个例子比较复杂一点，并且包括了一个形成前端与后端服务器进程的 servlet。这个例子是从一个完全不同的领域（金融服务）拿来的。

这个例子涉及一个名为 FastTicker.com 的假想的公司，该公司提供在线股票通知服务。实际生活中存在许多这种服务的例子；在本章，读者将看到构建这种软件是多么简单（并且可能会意识到为什么那些曾经在华尔街非常成功的 Internet 公司后来会倒闭）。显然，我们在这里实现的程序不会得到热烈的欢迎，但它的功能还是很完全的。

这个服务允许用户定义一个 profile，其中包括用户希望进行跟踪的股票代号，以及一个关于阈值的定义的集合。对于用户想要跟踪的每个股票代号，都可以分别定义一个下限和一个

上限。如果定义了一个下限，并且股票价格降到这个阈值之下，那么将通知用户。如果定义了一个上限，并且股票价格上升到这个阈值之上，那么也将通知用户。通知是基于向用户发送电子邮件实现的，它也是 profile 的一部分。

由于现在的传呼机与蜂窝电话公司允许用户使用电子邮件发送消息到传呼机或蜂窝电话上（通过 SMS（Short Message Service，短消息服务）），因此电子邮件通知是非常有用的。这个 profile 的最后一个部分是用户建立的事件间隔，用来定期检查股票价格并发出通知。这里所显示的实现是有限的，例如，如果用户指定了 5 分钟为间隔，并且股票价格没有变化但在所设置的上限之上，那么也会每隔 5 分钟发一个通知。如果股票价格没有变化，确实没有理由一直发送消息，但现实问题是，没有人利用这个实现去赚钱。

用户 profile 的维护、检查股票值以及执行通知就组成了这个 servlet 的实现。下面首先描述 profile 维护。

22.3.1 profile 的维护

由于用户 profile 将在 Web 上更新，因此需要 servlet 产生并管理用户 profile。我们将保持这个 servlet 足够的简单，这样导致的结果是用户界面比较简陋。对系统的首次访问是通过如下登录表单进行的：



这个表单使用一个 GET 请求连接到 servlet。doGet 方法在数据库中查找用户；如果用户被找到（并且口令正确），那么它将继续读取用户的 profile 并使用用户的值创建一个 HTML 表单。这些将被写入到 servlet 输出流中。图 22-1 显示了一个由此 servlet 创建的表单示例。

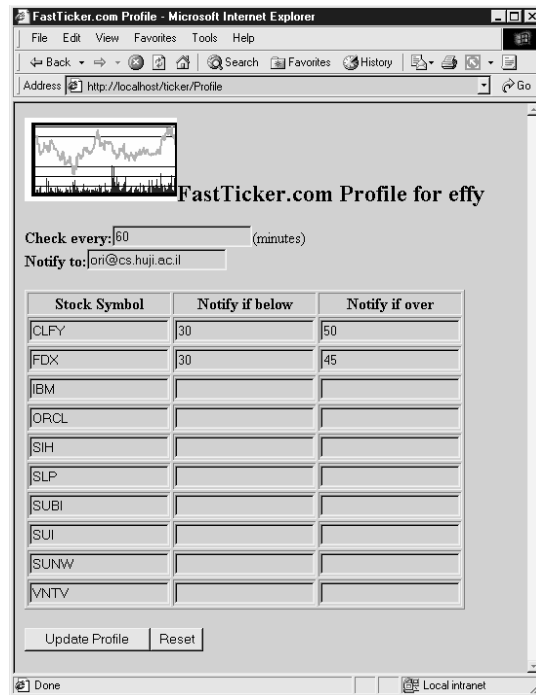


图 22-1 FastTicker servlet 创建的表单示例

下面显示了 doGet 方法创建的 HTML 源代码：

```
<HTML>
<HEAD>
<TITLE>FastTicker.com Profile
</TITLE>
</HEAD>
<BODY>
<H2><IMG SRC="http://yourhost/ticker.gif" BORDER>FastTicker.com
Profile for
effy</H2>
<FORM ACTION="http://yourhost/ticker/Profile" METHOD=POST>
<INPUT TYPE="HIDDEN" NAME="userName" VALUE="effy">
<B>Check every:</B><INPUT NAME="frequency" VALUE="60">(minutes)
<BR><B>Notify to:</B><INPUT NAME="email" VALUE="ori@cs.huji.ac.il">
<P><P><TABLE BORDER>
<TR><TH>Stock Symbol</TH><TH>Notify if below</TH>
    <TH>Notify if over</TH></TR>
<TR>
<TD><INPUT NAME="symbol0" VALUE="CLFY"></TD>
<TD><INPUT NAME="low0" VALUE="30"></TD>
```

```
<TD><INPUT NAME="high0" VALUE="50"></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol1" VALUE="FDX"></TD>
<TD><INPUT NAME="low1" VALUE="30"></TD>
<TD><INPUT NAME="high1" VALUE="45"></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol2" VALUE="IBM"></TD>
<TD><INPUT NAME="low2" VALUE=""></TD>
<TD><INPUT NAME="high2" VALUE=""></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol3" VALUE="ORCL"></TD>
<TD><INPUT NAME="low3" VALUE=""></TD>
<TD><INPUT NAME="high3" VALUE=""></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol4" VALUE="SIH"></TD>
<TD><INPUT NAME="low4" VALUE=""></TD>
<TD><INPUT NAME="high4" VALUE=""></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol5" VALUE="SLP"></TD>
<TD><INPUT NAME="low5" VALUE=""></TD>
<TD><INPUT NAME="high5" VALUE=""></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol6" VALUE="SUBI"></TD>
<TD><INPUT NAME="low6" VALUE=""></TD>
<TD><INPUT NAME="high6" VALUE=""></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol7" VALUE="SUI"></TD>
<TD><INPUT NAME="low7" VALUE=""></TD>
<TD><INPUT NAME="high7" VALUE=""></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol8" VALUE="SUNW"></TD>
<TD><INPUT NAME="low8" VALUE=""></TD>
<TD><INPUT NAME="high8" VALUE=""></TD>
</TR>
<TR>
<TD><INPUT NAME="symbol9" VALUE="VNTV"></TD>
```

```

<TD><INPUT NAME="low9" VALUE=""></TD>
<TD><INPUT NAME="high9" VALUE=""></TD>
</TR>
</TABLE><BR>
<INPUT TYPE="SUBMIT" VALUE="Update Profile"><INPUT TYPE="RESET">
</FORM>
</BODY>
</HTML>

```

需要说明的是, 表单的动作使用相同的 servlet, 但使用 POST 方法。这是因为在这个例子中, servlet 有两个作用: GET 方法用来获取用户 profile, POST 方法用来更新用户 profile。doGet 方法的实现如下:

```

public void doGet(
    HttpServletRequest request, HttpServletResponse
    response)
    throws ServletException, IOException {
    try {
        String loginName = request.getParameter("loginName");
        String passwd= request.getParameter("passwd");

        PrintWriter out = response.getWriter();
        response.setContentType("text/html");
        out.println("<HTML><HEAD><TITLE>Ticker.com"
            + "Profile</TITLE></HEAD>");
        out.println("<BODY>");
        out.println("<H2><IMG SRC=\"http://yourhost/ticker.gif\" " +
            "BORDER>FastTicker.com Profile for " + loginName +
            "</H2>");
        Statement stmt = null;
        try {
            Connection conn = DBConnection.getConnection();
            stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(
                "select passwd__t, frequency_c, "+"
                "email_c from monitor_users where user__t = '" +
                loginName + "'");
            if (rs.next()) {
                String p = rs.getString(1);
                int f = rs.getInt(2);
                String email = rs.getString(3);
                if (email == null)
                    email = "";
                if (p.compareTo(passwd) != 0)

```

```

        out.println("<H3>Incorrect password. " +
            "Please back up and try again</H3>");
    else {
        stmt.close();
        stmt = conn.createStatement();
        rs = stmt.executeQuery("select symbol__t, low_c,
            + "high_c " +
            "from monitors where user__t= '" +
            loginName + "' order by symbol__t");
        String[][] profileValues = new String[10][3];
        for (int i = 0; i < 10 ; i++)
            for (int j = 0 ; j < 3; j++)
                profileValues[i][j] = "";
        int i = 0;
        while ((i < 10) && rs.next()) {
            profileValues[i][0] = rs.getString(1);
            profileValues[i][1] = rs.getString(2);
            profileValues[i][2] = rs.getString(3);
            if (profileValues[i][1] == null)
                profileValues[i][1] = "";
            if (profileValues[i][2] == null)
                profileValues[i][2] = "";
            i++;
        }
        out.println("<FORM ACTION=" +
            "\"http://yourhost/servlet/Ticker\"METHOD=POST>");
        out.println("<INPUT TYPE=\"HIDDEN\" " +
            "NAME=\"userName\"VALUE=\"\" + loginName +
            "\">");
        out.println("<B>Check every:</B><INPUT
            + "NAME=\"frequency\""+
            "VALUE=\"\" + f + "\">(minutes)");
        out.println("<BR><B>Notify to:</B><INPUT"
            + "NAME=\"email\" +
            "VALUE=\"\" + email + "\">");
        out.println("<P><P><TABLE BORDER>");
        out.println("<TR><TH>Stock Symbol</TH><TH>Notify " +
            "if below</TH><TH>Notify if
            + "over</TH></TR>");
        for (int k = 0 ; k < 10 ; k++) {
            out.println("<TR>");
            out.println("<TD><INPUT NAME=\"symbol\" + k +
                "\"VALUE=\"\" +
                profileValues[k][0] + "\"></TD>");

```

```

        out.println("<TD><INPUT NAME=\"low\" + k +
                    \"\"VALUE=\"\" +
                    profileValues[k][1] + \"\"></TD>");
        out.println("<TD><INPUT NAME=\"high\" + k +
                    \"\"VALUE=\"\" +
                    profileValues[k][2] + \"\"></TD>");
        out.println("</TR>");
    }
    out.println("</TABLE><BR>");
    out.println("<INPUT TYPE=\"SUBMIT\" VALUE=\"Update\" +

                \"Profile\"><INPUT TYPE=\"RESET\">");
    out.println("</FORM>");
}
}
else
    out.println("<H3>No user named " + loginName +
                ". Please back up and try again</H3>");
} catch (Throwable ex) {
    out.println("<H3>Server error; please try again
                later</H3>");
    ex.printStackTrace();
}
if (stmt != null)
    stmt.close();

out.println("</BODY></HTML>");
out.close();

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

```

在验证用户名与口令之后，servlet 继续读取该用户的所有行。对于每一行，在 HTML 表单中都产生了一个字段集，并且在这些字段上有适当的值作为初始值。表单本身的构建方式是：当单击 Submit 按钮时，servlet 的 doPost 方法被调用。doPost 方法更新数据库中的用户 profile：

```

public void doPost(
    HttpServletRequest request, HttpServletResponse
                                response)
    throws ServletException, IOException
    response.setContentType("text/html");

```

```
PrintWriter out = response.getWriter();
out.println("<HTML><HEAD><TITLE>Ticker Profile
           Updated</TITLE></HEAD>");
out.println("<BODY>");

String[] symbols = new String[10];
String[] lows = new String[10];
String[] highs = new String[10];
String frequencyS = request.getParameter("frequency");
String emails = request.getParameter("email");
String userName = request.getParameter("userName");
for (int i = 0 ; i < 10 ; i++) {
    symbols[i] = request.getParameter("symbol" + i);
    lows[i] = request.getParameter("low" + i);
    highs[i] = request.getParameter("high" + i);
}
Statement stmt = null;
try {
    Connection conn = DBConnection.getConnection();
    stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(
        "update monitor_users set email_c = '" + emails +
        "', frequency_c = " + frequencyS + "where user__t = '"
        + userName + "'");
    rs.next();
    stmt.close();
    System.out.println("delete from monitors ");
    stmt = conn.createStatement();
    stmt.executeUpdate(
        "delete from monitors where user__t = '" +
        userName + "'");
    stmt = null;
    for (int i = 0 ; i < 10 ; i++)
        if (symbols[i].trim().length() > 0) {
            System.out.println("Inserting for symbol " +
                               symbols[i]);
            stmt = conn.createStatement();
            StringBuffer sqlString = new StringBuffer(100);
            sqlString.append(
                "insert into monitors(user__t, symbol__t");
            if (lows[i].trim().length() > 0)
                sqlString.append(", low_c");
            if (highs[i].trim().length() > 0)
                sqlString.append(", high_c");
```



```

        sqlString.append(") values ('" + userName +
            "',''" + symbols[i].trim() + "'");
        if (lows[i].trim().length() > 0)
            sqlString.append(", " + lows[i]);
        if (highs[i].trim().length() > 0)
            sqlString.append(", " + highs[i]);
        sqlString.append(")");
        String sqlS = sqlString.toString();
        System.out.println(sqlS);
        stmt.executeUpdate(sqlS);
        stmt.close();
        stmt = null;
    }
    out.println("<H3>Profile successfully updated</H3>");
} catch (Throwable ex) {
    out.println("<H3>Error while updating profile</H3>");
    ex.printStackTrace();
}

out.println("</BODY></HTML>");
out.close();
}

```

22.3.2 通知

此系统的第二部分就是执行实际股票查找并发出通知的进程：

```

package book.websphere.chapter22;

import java.util.*;
import java.sql.*;
import java.net.*;
import java.io.*;
import javax.mail.*;
import javax.mail.internet.*;

public class TickerDaemon implements Runnable {

    final static int dbPollingInterval = 30000; // 30 seconds

    Hashtable workerThreads;

    void updateWorkerThreads() {
        Statement stmt = null;
    }
}

```

```
try {
    Connection conn = DBConnection.getConnection();
    stmt = conn.createStatement();
    ResultSet rs = stmt.executeQuery(
        "select user__t, frequency_c, email_c from monitor_users");
    while (rs.next()) {
        String uName = rs.getString(1);
        int f = rs.getInt(2);
        String email = rs.getString(3);
        if (email != null) {
            if (!workerThreads.contains(uName)) {
                System.out.println("Adding worker thread for " +
                                    uName);
                WorkerThread nt = new WorkerThread(uName, f, email);
                workerThreads.put(uName, nt);
                (new Thread(nt)).start();
            }
        }
    }
} catch (Throwable ex) {
    ex.printStackTrace();
}

public TickerDaemon() {
    workerThreads = new Hashtable(100);
}

public void run() {
    updateWorkerThreads();
    try {
        Thread.currentThread().sleep(dbPollingInterval);
    } catch (Throwable ex) {
        ex.printStackTrace();
    }
}

public static void main(String[] args) {
    DBConnection.init("jdbc:oracle:thin:@websphere:1521:ORCL",
                     "ws", "ws");
    Thread t = new Thread(new TickerDaemon());
    t.start();
}
}
```

```
class WorkerThread implements Runnable {

    String userName;
    String email;
    int frequency;

    public WorkerThread(String uName, int f, String e) {
        userName = uName;
        frequency = f;
        email = e;
    }

    void updateBaseData() {
        try {
            Connection conn = DBConnection.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(
                "select frequency_c, email_c from monitor_users where
                user__t = '" +
                userName + "'");
            rs.next();
            frequency = rs.getInt(1);
            email = rs.getString(2);
            stmt.close();
        } catch (Throwable ex) {
            ex.printStackTrace();
        }
    }

    void checkQuoteData() {
        updateBaseData();
        StringBuffer urlBuf = new StringBuffer(1000);
        urlBuf.append("http://quotes.nasdaq.com/Quote.dll?mode=stock");
        Vector values = new Vector(10);
        try {
            Connection conn = DBConnection.getConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(
                "select symbol__t, low_c, high_c from monitors where"
                + "user__t = '" +
                userName + "' order by symbol__t");
            while (rs.next()) {
                String[] aValue = new String[3];
```

```
        aValue[0] = rs.getString(1);
        aValue[1] = rs.getString(2);
        aValue[2] = rs.getString(3);
        values.addElement(aValue);
        urlBuf.append("&symbol=");
        urlBuf.append(aValue[0]);
    }
} catch (Throwable ex) {
    ex.printStackTrace();
}
urlBuf.append("&x=0&y=0");
String urlString = urlBuf.toString();
String urlResponse = getHTMLFor(urlString);
if (urlResponse != null)
    parseAndNotify(urlResponse, values);
}

String getHTMLFor(String url) {
    try {
        URLConnection aConnection = (new URL(url)).openConnection();
        aConnection.setDoInput(true);
        aConnection.setDoOutput(true);
        aConnection.setUseCaches(false);
        InputStream inS = aConnection.getInputStream();
        StringBuffer buffer = new StringBuffer(4096);
        boolean atEnd = false;
        while(!atEnd) {
            int len = inS.available();
            byte[] buf = new byte[len];
            if (inS.read(buf, 0, len) == -1)
                atEnd = true;
            if (len > 0)
                buffer.append(new String(buf));
            Thread.sleep(10);
        }
        String retString = buffer.toString();
        return retString;
    } catch (Throwable ex) {
        ex.printStackTrace();
    }
    return null;
}

void notifyQuote(String messageText) {
```

```
Vector messagesToSend = new Vector(5);
int beginAt = 0;
while (beginAt < messageText.length()) {
    int endAt = beginAt + 150;
    if (endAt > messageText.length())
        endAt = messageText.length();
    messagesToSend.addElement(messageText.substring(beginAt,
                                                    endAt));

    beginAt = endAt + 1;
}
for (int i = 0; i < messagesToSend.size() ; i++) {
    String parts = (new Integer(i + 1)).toString() + "/" +
        (new
            Integer(messagesToSend.size())).toString();
    sendEmail((String)messagesToSend.elementAt(i), parts);
}
}

void sendEmail(String messageText, String part) {
    Properties props = new Properties();
    props.put("mail.smtp.host", "mail.fastticker.com");
    try {
        Message msg = new
            MimeMessage(Session.getDefaultInstance(props,
                                                    null));

        InternetAddress from = new InternetAddress("fastticker.com");
        msg.setFrom(from);
        InternetAddress[] to = { new InternetAddress(email) };
        msg.setRecipients(Message.RecipientType.TO, to);
        msg.setSubject("Stock quote from FastTicker.com for " +
            userName + " " + part);
        msg.setContent(messageText, "text/plain");
        Transport.send(msg);
        System.out.println("Sent email to " + email);
    } catch (MessagingException ex) {
        ex.printStackTrace();
    }
}

void parseAndNotify(String urlResponse, Vector values) {
    Hashtable prices = new Hashtable(values.size());
    Vector symbolLocations = new Vector(50);
    Vector symbols = new Vector(50);
    int beginAt = 0;
```

```
int endAt = 0;
int dollarAt = 0;

for (int i = 0 ; i < values.size() ; i++ ) {
    String[] aValue = (String[])values.elementAt(i);
    String symbol = aValue[0];
    beginAt = urlResponse.indexOf(symbol, endAt);
    dollarAt = urlResponse.indexOf("$", beginAt);
    endAt = urlResponse.indexOf("\\"", dollarAt);
    prices.put(symbol, urlResponse.substring(dollarAt+5, endAt));
}
StringBuffer messageText = new StringBuffer(200);
for (int i = 0 ; i < values.size() ; i++ ) {
    String[] aValue = (String[])values.elementAt(i);
    boolean sendQuote = ((aValue[1] == null) && (aValue[2] ==
                                                                    null));

    double aPrice = (new
        Double((String)(prices.get(aValue[0])))).doubleValue();
    if (aValue[1] != null)
        if ((new Double(aValue[1])).doubleValue() >= aPrice)
            sendQuote = true;
    if (aValue[2] != null)
        if ((new Double(aValue[2])).doubleValue() <= aPrice)
            sendQuote = true;
    if (sendQuote) {
        messageText.append(aValue[0]);
        messageText.append("=");
        messageText.append(prices.get(aValue[0]));
        messageText.append(" ");
    }
}
String text = messageText.toString();
if (text.length() > 0)
    notifyQuote(text);
}

public void run() {
    while(true) {
        checkQuoteData();
        try {
            Thread.currentThread().sleep(frequency*60*1000);
        } catch (Throwable ex) {
            ex.printStackTrace();
        }
    }
}
```

```
    }  
  }  
}
```

这个进程是作为一个 Runnable（可运行）类实现的。从另一个 servlet 的 init 方法调用此进程——这主要是为了方便，以允许注册此进程，从而能够在 WebSphere Application Server 启动后立即启动此进程。这个进程管理一个工作线程池，使得能够处理许多需要跟踪 profile 的用户。对于每个用户，数据库信息一直被读取。重新读取信息是必要的，这是因为可能出现当工作线程正在运行时，用户登录到系统并修改阈值的情况。然后，就需要获取股票价格信息，此 servlet 访问 NASDAQ 站点 www.nasdaq.com 以提供实际的股票信息。这个网站很好用，因为实际的价格查找操作可通过 URL 进行。基于用户的 profile，我们能够很简单地构造此 URL。

在使用 URLConnection 访问 NASDAQ 站点之后，可以读取所返回的 HTML 页面并基于一个解析方法提取价格信息。显然，任何实际的应用程序都应该更好地完成这一工作（这里所采用的方法的一个重要缺点就是它对 NASDAQ 站点关于 Web 设计的任何修改都很敏感）。然后，将提取的价格值与阈值进行比较，并且，若需要，将使用 JavaMail 包创建并发送一条电子邮件消息。我们有理由相信，这整个具有实际执行价值的服务的例子，只要用约一天的时间就可以构建并部署完毕。

22.4 小 结

本章介绍了 3 个 servlet 例子的代码，每个例子都执行一定的功能并创建一个 Web 页面，应用程序利用此 Web 页面可向用户显示信息。这些例子都比较简单，目的是帮助读者初步了解如何在 WebSphere 上构建并部署 servlet。由于 Web 应用程序模型是基于独立的 Web 请求/应答方式的，因此应用程序上下文并不天生地受内核 servlet 的支持。并且，任何应用程序都维护有一些应用程序状态。因此，对任何应用程序（即使是本章介绍的简单 servlet，如果没有某些应用程序状态是无法工作的），我们都需要描述如何在 servlet 模型中管理这些状态；这就是第 23 章要讨论的话题。

第 23 章 servlet 会话

本章继续讨论 servlet，重点描述在 servlet 中进行会话管理的问题。WebSphere 作为一个应用程序服务器，支持两个主要的应用程序模型：servlet（作为 servlet 的扩展，也包括 JSP）与 EJB。虽然 EJB 更适用于复杂的应用程序，servlet 更适用于 Web 部署的、短期的、面向事务的应用程序，但在必须管理应用程序状态的情形下，也可以使用 servlet。servlet 是当今许多支持 Web 的应用程序的基础，因此，大多数使用 servlet 的开发工程都需要以某种方式管理应用程序状态。

23.1 会话与状态

servlet 驻留在 WebSphere Application Server 中，并且等待 HTTP 请求（或者通过某些其他协议到达的请求）。由于 HTTP servlet 模型与 Web 服务器模型存在固有的关系，因此每条请求通常都没有相关的历史信息。这通常称为 Web 模型的无状态性（statelessness）——Web 天生就是无状态的。Web 模型无状态，这一事实是 Web 成为多维介质的重要因素。一个无状态模型比需要管理状态的模型能够更好地执行。在无状态模型下执行的软件，其构建过程更简单，因为它不需要解决诸如可以管理多少会话、如何知道通信断开、多个服务器如何被使用以达到负载均衡等问题。因此，无状态模型能够满足 Web 的基本需求。

但是，应用程序常常确实需要代表使用其服务的客户管理状态。考虑一个简单的电子商务应用情况，在这里，某个客户通过 Web 购买商品，并且将商品添加到他或她的购物车中。显然，所有的功能组都希望能够存储整个购物车，当购物过程继续时，更新购物车，并且当用户打算结帐时再次使用购物车。

无连接技术形成面向连接的技术的基础，这一情况并不少见。TCP/IP 就是这种例子。IP 与 UDP（User Datagram Protocol，用户数据报协议）并不支持连接的概念；它们仅仅实现了一个面向报文的通信协议。相反，TCP 就有连接的概念，并且在一个较低级的协议上管理正在运行的报文，其方式是向应用程序提供一个面向连接的视图。同样，无状态模型也可以用作必须维护状态的模型的基础。这样的做法是可能的（甚至是简单的）：以一种应用程序状态能够被维护的方式，添加某些使用无状态模型的应用程序级处理。实际上，由于这是所有应用程序如此基本的需求，使得这一额外的层已经被添加到所有应用程序服务器（WebSphere 也不例外）中。

本章描述了许多用来代表应用程序管理状态的方案，并且为每个方案的使用提供了例子。本章关注的是引入到 servlet 规范中的会话管理 API，因为它是 servlet 的最佳解决方案。下

一章将给出与会话管理相关的更多例子，并且将更详细地分析 WebSphere servlet 支持。

23.2 HTTP cookie

在开始描述 servlet 会话管理之前，需要理解 cookie 是什么。HTTP cookie 就是由服务器创建并传递到客户（通常是一个浏览器）的状态对象。这些封装了状态的实体最初由 Netscape Navigator 支持，后来逐渐被所有浏览器采用。这样，cookie 已经成为一种最常用方法，在这种方法下，服务器程序创建的状态信息可以被传回给客户，从而使得后来来自该客户的调用可以包含这些信息。cookie 允许对会话进行区分（demarcated），也就是，可以按照是否存在 cookie 来定义会话，并且客户与服务器之间所有涉及某个 cookie 的交互都可以被标记为一个单独会话的一部分。

cookie 有许多属性，如下：

- 名称和值：值就是一个常用来作为不透明结构的（例如一个标识）字符序列。
- 过期日期：定义一个超时，在此时间之后，cookie 的值就不再有意义。
- 域和路径：用来定义 cookie 的范围。域定义了与 cookie 相关的 Internet 域。默认的域值就是创建该 cookie 的服务器（cookie 从这个服务器到达客户）名，但服务器也可以指定一个更大的域（若需要）。
- 路径属性：当需要缩小范围时，这一属性有用。

最后，cookie 可以被标记为安全 cookie（secure cookie；通过一个附加的属性），这意味着该 cookie 应该通过一个安全的连接在客户与服务器之间传输。这是一个重要的功能，若没有该功能，大多数电子商务应用程序就不可能存在。例如，可以考虑在线交易应用程序。如果该应用程序很简单，并且仅涉及一个屏幕，那么可能不需要安全 cookie。但是，典型的在线交易应用程序是比较复杂的，并且在用户能够继续使用安全应用程序之前需要系统登录。管理该应用程序状态的一种可能方案就是 cookie，它以某种形式封装用户的标识。在应用程序构建时，应该考虑任何持有这个 cookie 的用户都能够继续一个会话并进行交易。显然，在这种情况下，应用程序必须保护 cookie 的值，就像保护用户名与口令一样。

虽然现在 cookie 得到了非常广泛的使用，但对 cookie 还存在一些争议，主要集中在两个问题上：保密性与安全性。cookie 允许应用程序跟踪状态。但是，它们也可能被用来执行许多其他任务。例如，它们提供了一种机制，使网站能够不断跟踪访问该网站的用户。这样，一个高级的网站就可能基于所得到的跟踪信息自定义将要提供给不同用户的信息。虽然这在我们看来是一件好事情，但显然这样的跟踪也可能被滥用，并且这种机制违背了 Web 匿名性这一基本特点。

例如，考虑下面这个在笔者的硬盘上发现的 cookie 文件：

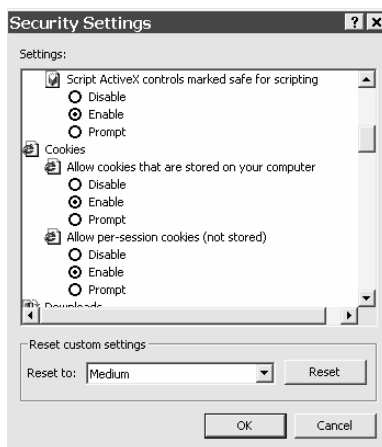
```
SITESERVER
```

```
ID=da4fcbc400eeb9b3d9190a88570eb57a
barnesandnoble.com/
0
642859008
31887777
3660024432
29271958
*

userid
3MMN8PH9NX
barnesandnoble.com/
0
358989312
29785928
3660324432
29271958
*

browserid
BITS=2&OS=3&VERSION=5%2E0&AOLVER=0&BROWSER=2
barnesandnoble.com/
0
1841121280
29308070
3660724432
29271958
*
```

显然, 这个 cookie 文件最初来自 barnesandnoble.com (顺便说明, 在 Windows 2000 上, cookie 文件放置在用户 profile 文件夹的一个称为 Cookies 的文件夹中)。第二个 cookie (称为 userid), 就是由 Web 站点放在这里; 它不是由用户 (而是由笔者) 显式地创建的 ID。此后, 当访问该网站时, 就利用这个 ID 标识我们, 并且这一信息很可能被用于我们并不知道 (甚至是我们不愿意) 的其他目的。为了允许“保密的人”同样能够使用 Web, 而不损失使用功能 (或公司的策略), 所有浏览器都支持禁用 cookie 的能力。下图显示了 Microsoft Internet Explorer 5.0 中的 Security Settings 对话框。可以在 Internet Explorer 中选择 Tools|Internet Options, 访问该对话框。然后, 打开 Security 标签并单击 Custom Level 按钮。这时就可以看到, 可以从这个对话框中禁用 cookie 支持。



警告：禁用 cookie 支持，可能会影响对许多 Web 应用程序的使用。

关于 cookie 的另一种反对意见就是，cookie 被存储在硬盘上，以后可以被用于访问某个网站。这种对 cookie 的使用与那些用于管理应用程序会话或状态的使用稍有不同，后者常常被用来标识返回的用户并且维护用户的状态。因此，必须对会话中使用的 cookie 与存储在硬盘上的 cookie 进行区分。现在再分析 Security Settings 对话框，可以注意到，用户可以允许用于会话管理（在浏览器会话中）的 cookie，而禁止持续性 cookie。

23.3 “传统的”会话管理

由于会话管理是应用程序固有的基础，使得自从在 Web 上出现第一个应用程序时（比 servlet 出现还早）就已经存在会话管理问题。已经存在许多管理会话的方法。

但是，这里并不打算对所有这些方法都进行完全的概述，因为这种讨论超出了本书的范围。此外，我们也不希望让读者养成‘坏习惯’。我们认为，编写 servlet 的人最好使用内置在 servlet API 中的会话管理功能。为了保持完整性，这里将讨论某些“传统的”会话管理机制，然后再讨论“主流的”会话管理机制。

23.3.1 被隐藏的字段

跟踪会话状态的最古老方法可能就是在表单中隐藏字段。在第 22 章，当把用户名嵌入到 FastTicker.com profile 更新屏幕中时，使用了被隐藏的字段。这个字段被 servlet 的 GET 处理过程所插入，并且被 POST 更新过程所使用。在典型情况下，会话与状态信息通过隐藏在表单中的字段集合所传递。由于所有字段（隐藏字段或非隐藏字段）可以在任何提交操作中从客户传递到 servlet，因此这些信息能够被有效地从一个请求处理线程传递到下一个请求

处理线程。这种方法的主要优点是简单，并且被所有浏览器支持，且对操作环境没有任何要求（因此，从该方法总是能正确工作这一点看，它是最安全的）。

23.3.2 重写 URL

另一种方法就是重写 URL，类似于被隐藏的字段，表现在它使用一种为其他工作而设计的机制顺便传递状态信息。在这种方法中，状态信息被附加在 servlet 路径上，并将在后来使用 request 对象的 `getPathInfo` 方法获取该信息。

23.4 使用 cookie 管理信息

在讨论 servlet 提供的对会话管理的内置支持之前，值得一提的最后一种方法就是使用 cookie。servlet 提供了一种方法，可用来通过 `javax.servlet.http.Cookie` 类、`HttpServletResponse` 中的 `addCookie` 方法以及 `HttpServletRequest` 中的 `getCookie` 方法手工地创建与使用 cookie。这个 API 提供了一种易于使用并且很巧妙（当然是比被隐藏的字段更巧妙，但不如将在下一节中讨论的方法）的信息管理途径。

下面将说明一个简单的 cookie 示例，它提供 Web 上的测试服务。将向用户提出一个问题，并请求给出回答。每次用户提交一个答案时，它将与正确的答案进行比较。如果所接收的答案不正确，那么这个问题将再次返回给用户，并且给出一个 cookie 值（当中保存了接收到错误回答的次数）以及所有不正确答案的 cookie。这些会话信息用来对问题进行评分（因为得分与试图回答的次数成反比）并产生最终的输出（在这个代码中将省略实际的测试评分部分）。图 23-1 显示了提供给用户的屏幕。

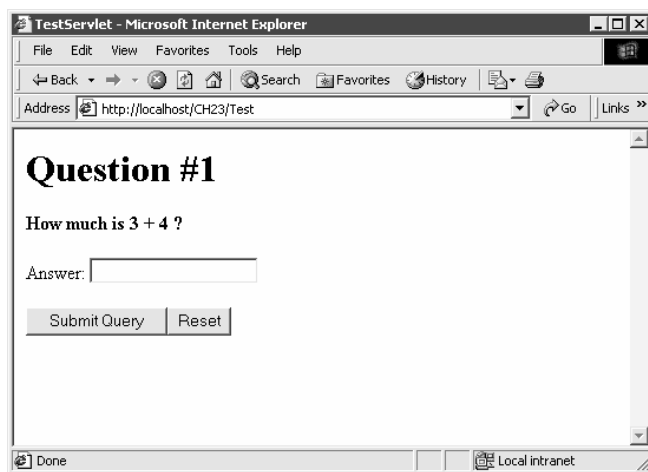


图 23-1 Question 屏幕

Question 屏幕是由 servlet 的 GET 处理方法创建的。在 doGet 方法中,除了生成 HTML 之外,一个名为 FAILED_ATTEMPTS 的 cookie 的值被设置为 0。如果用户给出了答案 5,那么 doPost 方法将让这个 cookie 的值增加 1,并且为这次失败的过程重新设置值。此外,它将添加一个关于此不正确答案的 cookie。

例如,在用户进行一次错误的回答之后,所得到的输出如图 23-2 所示。如果再出现一次错误的回答,那么所得到的输出如图 23-3 所示。

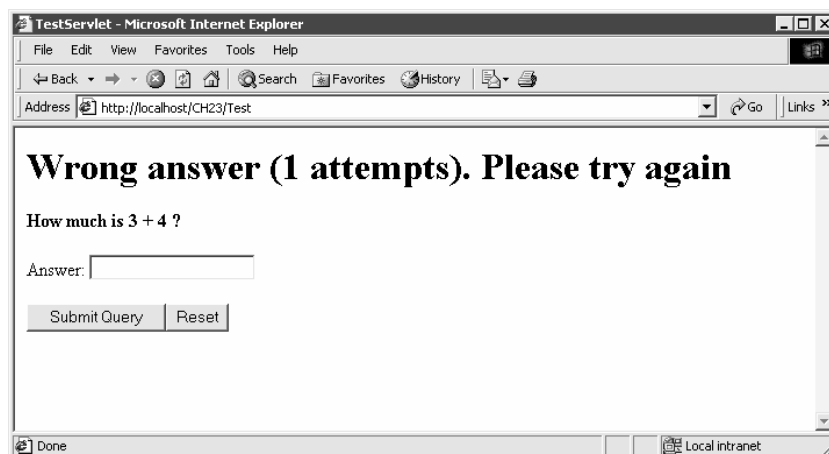


图 23-2 Wrong Answer 屏幕

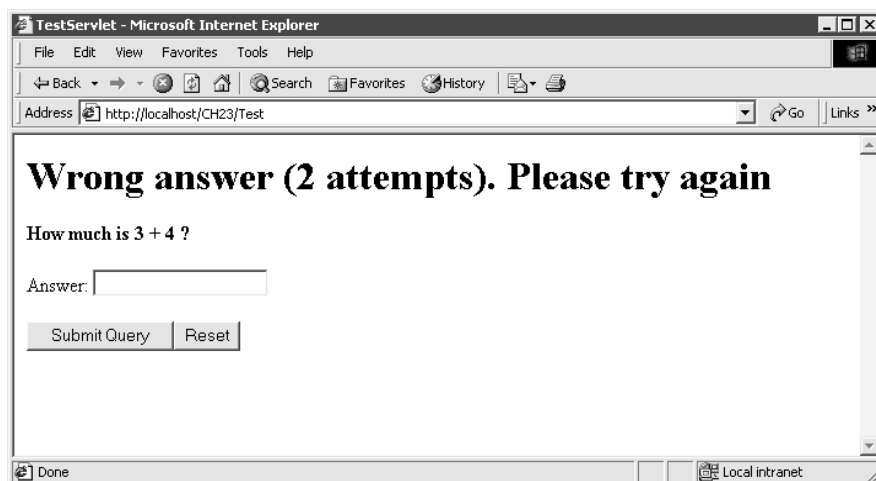


图 23-3 第二次出现 Wrong Answer 屏幕

可以注意到,每次这样的输出都给出了本次回答已累计的次数;这样是可能做到的,因为构成应用程序状态的 cookie 信息总是包含有失败回答的次数。当用户最后作出正确的回答

时，所得到的输出如图 23-4 所示。

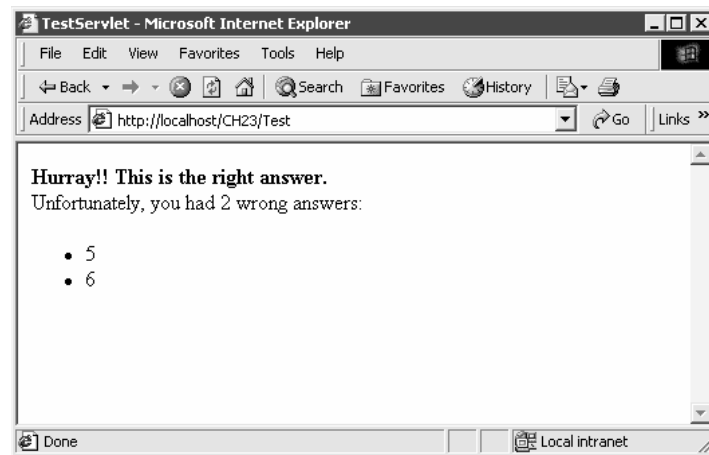


图 23-4 得到正确回答之后的屏幕

用户所有的错误回答都能够被显示，这是因为它们是会话状态（state）的一部分。下面是关于这个例子的 servlet 代码：

```
package book.websphere.chapter23;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class TestServlet extends HttpServlet {
    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
    }

    public void doGet(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = new PrintWriter (response.getOutputStream());

        Cookie myCookie = new Cookie("FAILED_ATTEMPTS", "0");
        response.addCookie(myCookie);
    }
}
```

```
        printOutForm(out, "Question #1");
        out.close();
    }

    void printOutForm(PrintWriter out, String extraString) {
        out.println("<html>");
        out.println("<head><title>TestServlet</title></head>");
        out.println("<body><H1>" + extraString + "</H1>");
        out.println("<B>How much is 3 + 4 ?</B>");
        out.println("<FORM ACTION=\"http://yourhost/ch23/Test\" +
            \"\"METHOD=POST>");
        out.println("Answer: <INPUT TYPE=\"TEXT\"\"
            + "NAME=\"answer\"><P>");
        out.println("<INPUT TYPE=\"SUBMIT\"><INPUT
            + "TYPE=\"RESET\"</FORM>");
        out.println("</body></html>");
    }

    public void doPost(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = new PrintWriter (response.getOutputStream());

        int currentNumberOfWrongAnswers = 0;
        Vector wrongAnswers = new Vector();
        Cookie[] myCookies = request.getCookies();
        for (int i = 0 ; i < myCookies.length; i++) {
            String name = myCookies[i].getName();
            String value = myCookies[i].getValue();
            System.err.println("Cookie named " + name + " has value " +
                value);
            if (name.compareTo("FAILED_ATTEMPTS") == 0)
                currentNumberOfWrongAnswers = Integer.parseInt(value);
            else
                wrongAnswers.addElement(value);
        }
        System.err.println("Finished reading cookies");
        int currentAnswer =
            Integer.parseInt(request.getParameter("answer"));
        if (currentAnswer != 7) {
```

```

        currentNumberOfWrongAnswers++;
        Cookie anotherWrongAnswer = new Cookie(
            "WRONG_ANSWER_" + currentNumberOfWrongAnswers,
            request.getParameter("answer"));
        response.addCookie(anotherWrongAnswer);
        Cookie myCookie = new Cookie(
            "FAILED_ATTEMPTS",
            Integer.toString(currentNumberOfWrongAnswers));
        response.addCookie(myCookie);
        printOutForm(out,
            "Wrong answer (" + currentNumberOfWrongAnswers +
                " attempts). Please try again");
    } else {
        out.println("<html>");
        out.println("<head><title>TestServlet</title></head>");
        out.println("<body>");
        out.println("<B>Hurray!! This is the right answer.</B><BR>");
        if (currentNumberOfWrongAnswers > 0) {
            out.println("Unfortunately, you had " +
                currentNumberOfWrongAnswers +
                " wrong answers:<BR>");
            out.println("<UL>");
            for (Enumeration e = wrongAnswers.elements() ;
                e.hasMoreElements() ; ) {
                out.println("<LI>");
                out.println(e.nextElement());
            }
            out.println("</UL>");
            out.println("</body></html>");
        }
    }
    out.close();
}

public String getServletInfo() {
    return "book.websphere.chapter23.TestServlet Information";
}
}

```

需要指出的有趣一点是：如果在浏览器中运行这个例子，并且打开另一个浏览器窗口（从File 菜单中进行选择，而不是打开关于该应用程序的又一个实例），那么两个浏览器窗口将共享相同的会话上下文，因此显示的企图回答次数是这两个窗口各自企图回答次数之和。另一方面，如果打开浏览器的另一个实例，那么两个结果窗口不共享相同的会话，并且系

系统将跟踪两个独立的企图序列。

23.5 会话跟踪 API：“可取的”会话管理

现在，讨论基于 servlet 的应用程序中跟踪与管理会话状态的更可取方法：会话跟踪 API。前面对 cookie 的讨论是重要的，不仅因为它们 Web 应用程序中被广泛使用，而且因为 cookie 是 servlet 会话跟踪的底层基础。servlet 会话跟踪机制与 cookie 之间的不同在于，在会话跟踪机制中，cookie 被用来指向服务器上所维护的结构，该结构管理构成应用程序状态的信息。这比让状态信息自己以 cookie 的形式在网络上传输更好。

cookie 允许服务器添加一个值的字典（dictionary）（名称-值对集合）到服务器与浏览器之间传递的消息中。servlet 会话跟踪 API 仅仅在服务器上维护这个字典，并且提供一个惟一的标识符来检索该字典；这个标识符放置在发送给浏览器的 cookie 中。这种管理方式在两个方面都表现最佳：所支持的功能类似于 cookie 的完全功能，但这种方式存在更小的通信开销、更高的保密性、更简单的维护性与更好的封装等优点。

利用会话跟踪 API 测试 servlet

下面继续使用前一节中用到的小测试 servlet，并且试图利用 servlet 会话对其进行改写。所得到的结果如下，其中新代码用黑体标示：

```
package book.websphere.chapter23;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.*;

public class TestServlet2 extends HttpServlet {

    public void init(ServletConfig config) throws ServletException {
        super.init(config);
    }

    public void doGet(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = new PrintWriter (response.getOutputStream());
```

```
HttpSession aSession= request.getSession(true);
aSession.setAttribute("FAILED_ATTEMPTS", new Integer(0));

printOutForm(out, "Question #1");
out.close();
}

void printOutForm(PrintWriter out, String extraString) {
    out.println("<html>");
    out.println("<head><title>TestServlet</title></head>");
    out.println("<body><H1>" + extraString + "</H1>");
    out.println("<B>How much is 3 + 4 ?</B>");
    out.println("<FORM ACTION=\"http://yourhost/ch23/Test2\" +
        \"\"METHOD=POST>");
    out.println("Answer: <INPUT TYPE=\"TEXT\" \" +
        "NAME=\"answer\"><P>");
    out.println("<INPUT TYPE=\"SUBMIT\"INPUT\" +
        "TYPE=\"RESET\"</FORM>");
    out.println("</body></html>");
}

public void doPost(
    HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = new PrintWriter (response.getOutputStream());

    int currentNumberOfWrongAnswers = 0;
    Vector wrongAnswers;

    HttpSession aSession = request.getSession(true);

    if (aSession.getAttribute("WRONG_ANSWERS") == null)
        wrongAnswers = new Vector();
    else {
        wrongAnswers =Vector}(aSession.getAttribute("WRONG_ANSWERS"));
        currentNumberOfWrongAnswers =

((Integer)(aSession.getAttribute("FAILED_ATTEMPTS"))).intValue();
    }
    int currentAnswer =
```

```

        Integer.parseInt(request.getParameter("answer"));
    if (currentAnswer != 7) {
        currentNumberOfWrongAnswers++;
        wrongAnswers.addElement(request.getParameter("answer"));
        aSession.setAttribute(
            "FAILED_ATTEMPTS", new Integer(currentNumberOfWrongAnswers));
        aSession.setAttribute(
            "WRONG_ANSWERS", wrongAnswers);
        printOutForm(out,
            "Wrong answer (" + currentNumberOfWrongAnswers +
            " attempts). Please try again");
    } else {
        out.println("<html>");
        out.println("<head><title>TestServlet</title></head>");
        out.println("<body>");
        out.println("<B>Hurray!! This is the right answer.</B><BR>");
        if (currentNumberOfWrongAnswers > 0) {
            out.println("Unfortunately, you had " +
                currentNumberOfWrongAnswers +
                " wrong answers:<BR>");
            out.println("<UL>");
            for (Enumeration e = wrongAnswers.elements() ;
                e.hasMoreElements() ; ) {
                out.println("<LI>");
                out.println(e.nextElement());
            }
            out.println("</UL>");
            out.println("</body></html>");
        }
    }
    out.close();
}

public String getServletInfo() {
    return "book.websphere.chapter23.TestServlet Information";
}
}

```

这段代码提供了与前一例子相同的功能，给出该代码的目的是说明当希望使用会话跟踪时应该如何对代码进行修改。

这个例子使用了会话跟踪 API，在此情况下，会话对象被用来保存一些数据结构。由于这些对象是在 WebSphere Application Server 中维护的，因此任何对象都可以存储在会话中。需要说明的是，应该仅仅将可串行化对象存储在会话中。否则，就不能使用持续性会话选

项。当应用这个选项之后，存储在会话对象中的对象就能够保存在硬盘上，并且在需要时从硬盘上重构该对象。这样就要求放置在会话中的每个对象都是可串行化的；我们强烈推荐仅仅将这样的对象放在会话中（即使没有任何使用持续性会话的打算）。因此，在原来老的 cookie 例子中，每个错误的答案都存储在一个字符串中，但在新代码里，所有错误的答案可以被存储在一个向量中，这个向量可以存储在会话中作为一个单独的实体。这样就消除了需要为被跟踪的消息使用人造名称的需要（正如我们在利用 cookie 时所做的那样）。

使用会话跟踪 API 的代码是简单的。首先，利用请求的 `getSession` 方法获取会话对象。传递给这个方法的参数指定了（在这个例子中）如果某个会话对象还没有被创建，就应该创建一个会话对象作为该调用的结果。然后，这个会话对象被用作使用 `getAttribute` 与 `setAttribute` 方法的相关联数组。

会话跟踪 API 显然比这个例子中所显示的功能还更强。最重要的是，会话对象可用来维护服务器上的对象；传递给浏览器的一个单独 cookie 可用来提供对该结构的下标。

查看这个会话所使用的 cookie 的简易方法，就是在使用这个测试 servlet（指的是第二个版本）之后调用 snoop servlet。只有当此 snoop servlet 与测试 servlet 被打包在相同的 Web 应用程序中时，此方法才能正确工作，因为该会话仅仅在应用程序边界内维护。在 Client Cookie 下，会话 ID 可以很容易地确定，如图 23-5 所示。

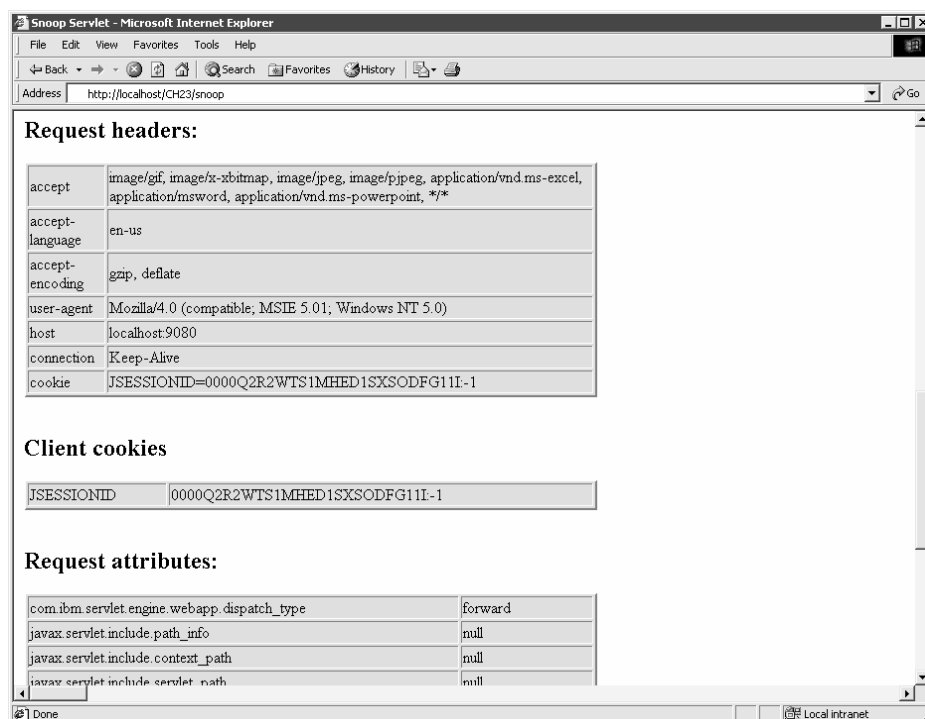


图 23-5 会话 ID 位于 Client Cookie 下

由于 cookie 是浏览器使用 jsessionid cookie 维护的, 因此在该测试 servlet 完成之后, 这个 cookie 仍然存在。这就是为什么能够通过激活 snoop servlet 来查看此 cookie 的原因。显然, 除了维护对象字典这一工作之外, 会话还有其他任务。在 Servlet API 2.2 中, HttpSession 对象支持的其他方法如下:

- `getCreationTime`: 获取会话被创建的时间。
- `getId`: 获取赋给每个会话的标识符。这个标识符在 WebSphere 中是惟一的, 并且它就是通过 cookie 传递给浏览器的那个标识符。
- `getLastAccessTime`: 返回客户最后一次发送带有会话标识符的消息的时间。
- `getSessionContext`: 返回会话的上下文。
- `getAttributeNames`: 返回与该会话关联的对象的所有关键字。
- `getMaxInactiveInterval`: 返回 Web 容器在客户请求之间保持该会话的最长时间间隔 (单位为秒)。
- `setMaxInactiveInterval`: 设置 Web 容器在客户请求之间保持该会话的最长时间间隔 (单位为秒)。
- `invalidate`: 告知 WebSphere 不再维护某个会话 (请参见后面的讨论)。
- `isNew`: 如果某个会话已经被服务器创建, 但客户还没有确认加入到该会话中, 那么将此会话标识为 new (新的)。
- `removeAttribute`: 从会话中删除一个对象。

一旦 `invalidate()` 方法被调用, 那么为这个会话维护的所有对象都将被废弃并且该会话不再被使用。对于 Web 工作模型, 提供这个方法是非常重要的, 因为在 Web 模型下, 浏览器可能在使用服务器的某时刻“突然消失”。发生这种情况可能有许多原因, 例如, 用户可能关闭浏览器并执行另一个任务, 或者, 连接可能丢失。与此同时, 我们可能还维护了这个浏览器会话的对象。除非希望服务器的内存使用变得越来越大 (直到用完所有的内存空间, 或直到性能大幅度下降以至于需要重新启动), 否则这些信息就需要在适当的时间间隔之后被丢弃。一种方法就是通过注销例程, 它将作废会话。另一种方法就是使用后台进程, 周期性地作废那些有一段时间没有被使用的会话 (在第 24 章, 读者还将见到另一种管理清除的方法)。

23.6 小 结

应用程序状态是 servlet 编程中最重要的方面。不同于大多数普通的 servlet, 所有应用程序 servlet 都需要管理状态信息, 在构成用户会话的许多请求/响应对之间就这些状态信息进行通信。由于应用程序状态对 Web 应用程序模型 (特别是对 servlet) 如此关键, 使得应用程序状态已经成为核心 servlet API 的重要部分。

第 24 章 WebSphere 特定的 servlet 问题

本章继续讨论 servlet，并重点说明 WebSphere 特定的 servlet 支持。WebSphere 扩展了 Java servlet API，并提供了一些非标准的功能。这些 IBM 扩展包括对更简易的会话管理支持、个性化支持、错误报告支持与 servlet chain 支持。

24.1 WebSphere 会话跟踪

从上一章可以了解到，servlet 中的会话跟踪基于 HttpSession 接口实现。WebSphere 发布利用 com.ibm.servlet.session.IBMSession 接口扩展了标准的 HttpSession。这个接口支持多 WebSphere 服务器、安全性与持续性，从而加强了“常规的”会话跟踪功能。

24.1.1 Administrative Console 中的会话跟踪设置

第 23 章讨论了进行会话管理的方法。显然，会话管理对使用会话跟踪 API 的 Java servlet 是可用的。但是，在实际应用中会话的用法取决于 WebSphere 管理工具定义的配置。图 24-1 显示了在每个应用程序服务器下都可用的 Session Manager 页面（这个图只显示了一个服务器：默认的应用程序服务器）。

Session Manager 工具包括许多可以被修改的属性。用户能够通过重写 URL 与 cookie 支持来启用或禁用会话管理（在第 23 章中已经说明，这是维护会话的两种方法）。如果通过重写 URL 或 cookie 支持禁用会话管理，那么将不会发生会话跟踪——但总是可以使用被隐藏的字段或通过其他自己创建的途径执行会话管理。使用 URL 重写的优点是即使用户在 Web 浏览器中禁用 cookie 时，它也能正常工作。但是，URL 重写的一个重要缺点就是静态 HTML 文件可能不能正确工作，并且在这些情况下，必须利用 URL 数据手工地编写 JSP 与 servlet。这就意味着需要进行大量的工作，因此 URL 重写仅仅在希望支持不带 cookie 的浏览器时才有价值。

如果启用 URL 重写与 cookie，那么会话管理方式取决于 servlet 的编码方式。如果 servlet 中包括对 URL 进行编码的逻辑，那么所有的会话管理将基于 URL 重写，而不考虑 Enable Cookies 设置。如果在 servlet 中没有显式地包含对 URL 重写的代码，并且选择了 Enable Cookies，那么只要客户 Web 浏览器能够接受 cookie，就将使用 cookie。如果 Web 浏览器禁用 cookie，并且在 servlet 中没有实现 URL 重写代码，那么 Enable Cookies 与 Enable URL Rewriting 都被忽略，就像两者都不能使用一样。

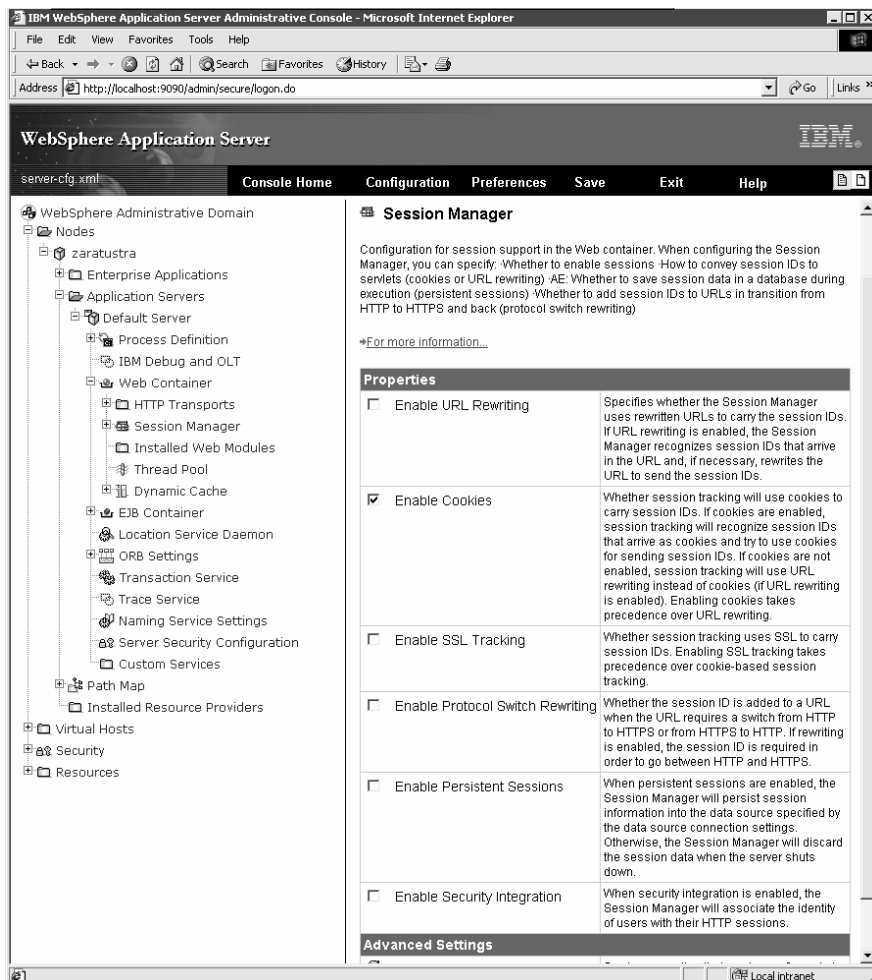


图 24-1 Administrative Console Session Manager 页面

也可以启用或禁用协议切换重写 (protocol switch rewriting)，当需要进行从安全连接到非安全连接的切换 (或相反的切换) 时，协议切换重写用来告知 WebSphere 是否应该传递会话信息。

WebSphere 支持的另一个选项就是 SSL (Secure Socket Layer, 安全套接字层) 会话管理，在 SSL 会话管理下，通过 SSL 连接传输会话信息。使用这种方法还有更多的好处 (也就是安全性集成 (security integration))；已认证的用户与他们的会话信息相关联。

最后，Persistent Session 选项意味着持续性会话存储在一个数据库中；这样，它们的生命长度就超过了服务器进程的生命长度。因此，在遇到服务器关闭这样的情况时会话仍然持续。换句话说，如果服务器关闭，后来又重新启动，启用 Persistent Session 这个选项就使得服务

器能够在不必破坏客户会话的条件下恢复到服务器关闭之前的状态。持续性会话对于长时间地维护用户会话也很有用，并且，正如在本章后面的“集群环境下的会话”一节中解释的那样，持续性会话在集群环境下对于会话跟踪也起着重要作用。

由于可能在一个会话中导致性能降低，使得人们常常认为使用持续性会话不好。毕竟，持续性会话中并非所有的会话管理都发生在内存中，而是有些管理工作必须访问硬盘。但是，持续性会话在许多业务应用程序以及业务情景中是很重要的，因为即使服务器关闭，此会话仍然存在；当用户会话信息是关键信息时，这一功能是非常重要的。有时，会话对象仅仅维护了应用程序为用户服务的上下文。另有一些时候，会话信息也包括了一个在长的应用程序会话中建立的数据，或者包括了几个应用程序屏幕之间建立的数据。在这种情况下，除非应用程序特别地管理这种持续性，否则，使用持续性会话就可以作为避免系统减速以至于挫伤用户信心的一种方法。

持续性会话应用的另一种情形就是移动设备。移动计算平台通常有两种方式：在线工作并且不断对中央服务器进行访问的应用程序，在移动设备上具有足够的逻辑并使用同步机制与中央数据库通信的应用程序。在后一种情况下，用户可能在上午 9:00 与中央服务器同步，然后在断开连接的方式下在移动设备上进行自己的工作。然后，在中午时，用户可能再次与服务器同步。从用户的角度看，第二次同步与第一次同步在相同的应用程序上下文中进行。但是，大多数应用程序服务器的配置中，会话过期时间可能远远小于 3 个小时。因此，除非应用程序特别地处理这个问题，否则，就应该使用持续性会话。

最后，需要使用持续性方式而不是使用完全在内存中方式的另一种情形与过期时间值有关。大多数在 WebSphere 以及其他应用程序服务器上部署的 Web 应用程序的配置中都有一个 15~45 分钟的会话过期时间。由于在 Web 应用程序模型中，用户可能在不注销系统的情况下就直接关闭浏览器，而且会话对象是可能消耗大量服务器内存的昂贵资源，因此，大多数 Web 应用程序都假设如果一个用户在相对较短的时间内没有使用这个系统，该用户将被取消。电子商务应用程序通常就是这样做的。但是，Web 应用程序现在被用于所有类型的设备，包括那些为内部用户服务以及为外部用户服务的设备。在有些情况下，系统管理员需要确保用户即使在很长的时间段内没有动作的情况下也不能取消该用户（实际上，我们就曾经遇到过必须在 9 小时内不能取消用户的情况！）。在这种情形下，将所有的会话对象纯粹维持在内存中就可能導致服务器失效或出现其他问题。

现在回去讨论使用 cookie 用作标识会话的方法。在 Session Manager 页表项中，可以发现 Default Cookie Settings 页，如图 24-2 所示。

在这个页中，可以为保持有会话标识符的 cookie 设置默认值。与 WebSphere 的先前发行版不同，在 WebSphere 4.0 中不能设置 cookie 名，因为它现在是 J2EE 规范的一部分（但现在，cookie 名已不再是一个有用的功能了）。现在常用的是限制这种 cookie 将被发送到的域的个数的能力（例如，如果已经知道应用程序服务器仅仅支持内部网上的应用程序，则可以使用此功能）。Maximum Age 域保持有一个整数值（单位为秒），定义了 cookie 被浏览器作废的时间期限。读者不要将这个时间期限与可能发生在服务器上的会话作废期限混淆；这个

cookie 值放置在响应头部中作为 cookie 信息的一部分（正如在第 23 章讨论的那样），并且由浏览器执行作废动作。最后，Path 域允许输入信息，用来限制这个会话 cookie 将要发送到的服务器的路径。

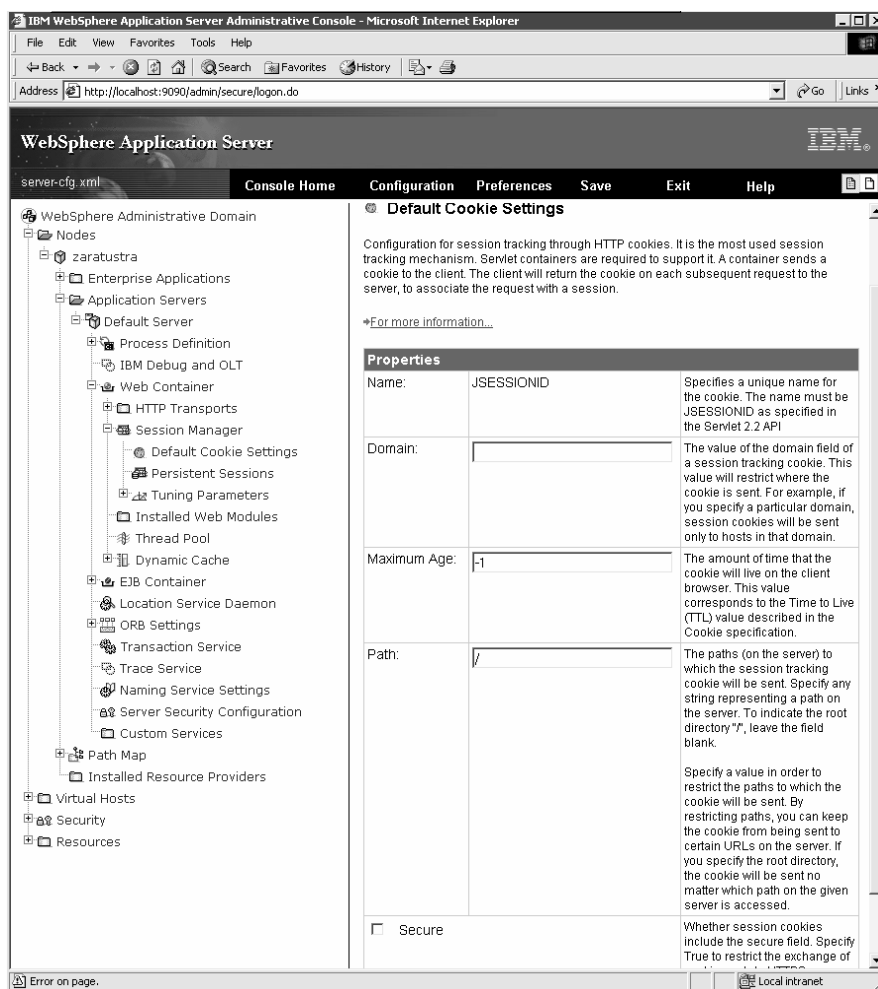


图 24-2 Administrative Console Default Cookie Settings 页面

图 24-3 所示的 Persistent Session 页，用来配置 Session Manager 将用于获取数据库连接的数据源。这个页面上的设置将决定服务器如何使会话信息持续化。只有当启用持续性会话时，这个页才可应用。也可以设置 JDBC 数据源，并且设置用户名与口令，当为了存储会话信息而连接到数据库时将使用用户名与口令。如果正在使用 IBM DB2，就需要指定会话的大小（但当使用 Oracle 时，可以使用 32K 会话，而不需要任何特殊的设置）。如果正在使用

DB2 并且需要多于 4K 的表空间, 那么就需要指定会话表所需的表空间 (使用 DB2 中的 CREATE TABLESPACE 命令)。

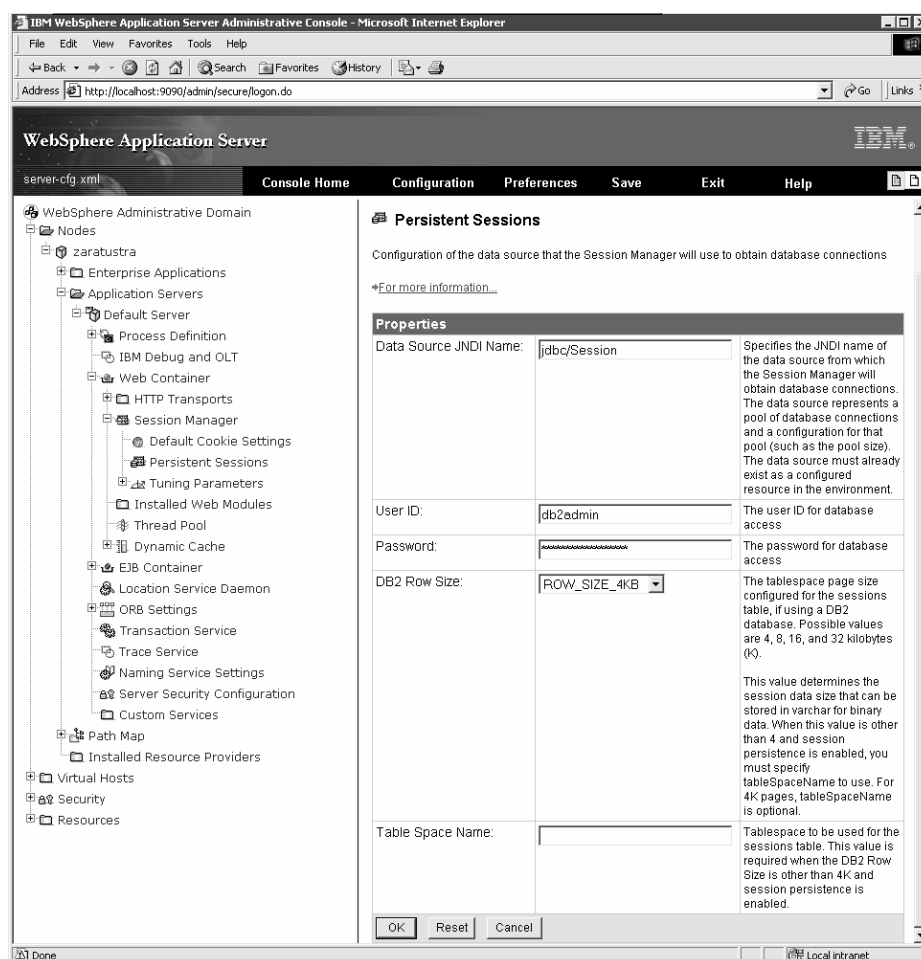


图 24-3 Administrative Console Persistent Sessions 页面

最后, Administrative Console 还提供了可以在其中优化会话管理行为的页面。在 Tuning Parameters 页面 (如图 24-4 所示) 中, 可以控制存储在内存中的最大会话数以及没有使用的会话的生命周期 (在最后一次被使用之后, 某个会话最多能够存在的时间)。Allow Overflow 设置用来指定是否允许内存中的会话对象数暂时性地超过所指定的最大会话数。

Tuning Parameters

Tuning parameters for session support

For more information...

Properties

☐ Using Multitrow Schema

Whether to place each instance of application data in a separate row in the database, allowing larger amounts of data to be stored for each session. This can yield better performance in certain usage scenarios. If using multitrow schema is not enabled, instances of application data can be placed in the same row.

Max In Memory Session Count:

1000

Specifies the maximum number of sessions to maintain in memory.

The meaning differs depending on whether you are using in-memory or persistent sessions. For in-memory sessions, this value specifies the number of sessions in the base session table. Use the Allow Overflow property to specify whether to limit sessions to this number for the entire Session Manager, or to allow additional sessions to be stored in secondary tables.

For persistent sessions, this value specifies the size of the general cache. If the Cache property is enabled, this value specifies how many session updates will be cached before the Session Manager reverts to reading session updates from the database automatically.

This value holds when you are using in-memory sessions, persistent sessions with caching, or persistent sessions with manual updates. (The manual update cache keeps the last n time stamps representing "last access" times, with n being the Max In Memory Session Count value).

☒ Allow Overflow

Whether to allow the number of sessions in memory to exceed the value specified by Max In Memory Session Count property

☐ Schedule Invalidation

Enables the scheduled invalidation process for cleaning up the invalidated HttpSession from the sessions database. When enabled, this setting specifies the two hours of a day in which the invalidator process cleans up the invalidated persistent sessions in the database.

Write Frequency:

END_OF_SERVLET_SERVICE

When the session is written to the database. If you specify END_OF_SERVLET_SERVICE, a session is written to the database after the servlet completes execution. If you specify MANUAL_UPDATE, programmatic sync on the HttpSession object is required in order to write the session data to the database. If you specify TIME_BASED_WRITE, session data is written to the database based on the specified Write Interval value.

Write Interval:

120

If the Write Frequency is specified as TIME_BASED_WRITE, this value specifies how often the session data will be written to the database. The value must be a number of seconds, specified as a positive integer in the range from 5 to 9999. At minimum, a configured Invalidation Timeout should be twice as large as the configured write interval.

Write Contents:

ONLY_UPDATED_ATTRIBUTES

Whether only updated attributes should be written to the database. Otherwise, all of the session attributes will be written to the database, whether or not they have changed.

Invalidation Timeout:

30

Specifies how long a session is allowed to go unused before it will be considered valid no longer. The value is in minutes. Use a -1 to specify that the session will not be invalidated.

The value of this setting is used as a default when the session timeout is not specified in a Web module deployment descriptor.

Note, to preserve performance, the invalidation timer is not accurate "to the second." It is safe to assume that the timer is accurate to within two minutes. When the Write Frequency is time based, this value should be at least twice as large as the write interval.

OK

Reset

Cancel

图 24-4 Administrative Console Tuning Parameters 页面

还可以指定持续性会话信息应该在何时被写出到数据库中。可能的值如下：

- END_OF_SERVLET_SERVICE：表示当 servlet 运行完成时写出持续性信息。
- MANUAL_UPDATE：信息更新的频率由 servlet 编程者决定。这样，就必须使用 HttpSession 接口（请参见本章后面的 24.1.3 节“WebSphere 会话跟踪 API”）。
- TIME_BASED_WRITE：允许在固定的时间间隔上保存会话。这一间隔可通过 Write Interval 参数进行选择。

也可以控制被写入的实际内容——是整个会话对象还是那些被修改的属性。
最后一个选项用来进行作废调度。使用这个选项，就能使用一个垃圾收集进程，从数据库的持续性存储器中删除被作废的会话。可以在 Invalidation Schedule 页面上设置这种作废在什么时候发生，如图 24-5 所示，它位于 Tuning Parameters 下。
会话作废进程被设置为每天发生两次，并且在这个页面上，可以选择作为动作发生的时间；这里定义的时间是以 0~23 之间的整数来指定的。

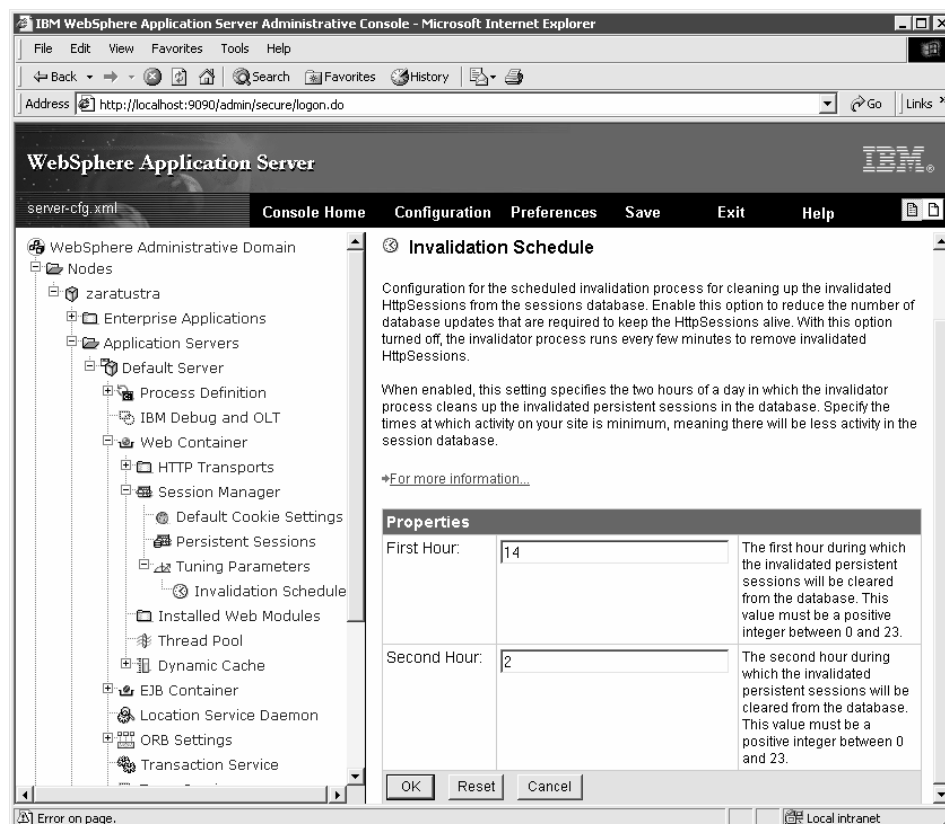


图 24-5 Administrative Console Invalidation Schedule 页面

24.1.2 集群环境下的会话

对于一个支持大量用户的典型电子商务应用情形，任何单独的处理器都没有如此强大的功能，可以支持在理论上无限的用户数。因此，为了支持可扩展性很好的高性能网站，典型安装都包括多个应用程序服务器实例，它们共同工作并给人留下一种只有单个服务器实例正在运行的假象。这可以在一个多处理器机器上配置，每个处理器运行一个服务器，或在多个机器上，每个机器可能运行多个服务器。

无论配置如何，得到的结果都是多个 WebSphere 服务器在行为上就像一个逻辑服务器一样工作。这样的逻辑服务器就称为集群 (cluster)。WebSphere Application Server 高级版 (与单站点版) 支持使用集群。它自动管理对发送给集群实例的请求委托。这种一个逻辑服务器的情况，就隐含地要求会话管理必须在概念上是集中式的管理。因此，所有会话都必须对集群中所有服务器可用。

使用 WebSphere 高级版进行集群配置有许多优点。首先,这种部署类型是可容错的体系结构,在这种体系结构下,如果由于失败或预计划的维护策略导致某个应用程序服务器关闭,那么还有一个备份存在。由于所有应用程序服务器都能处理任何会话,因此用户并不能意识到这一点(除了可能会感觉到对现有的服务器的负载似乎更高之外)。这种体系结构也支持高可用性 (high availability),高可用性的概念用来描述一个体系结构不仅提供了容错支持,而且使所有资源(包括后备资源)可同时被使用。提供容错支持的一种方法就是设置一个主服务器,同时提供一个后备服务器,当主服务器出现问题时,后备服务器才开始活跃。在这种配置下,整个系统在性能方面并没有受益。高可用性用来描述一个系统具有后备服务器,但这个后备服务器也用于普通的操作,因此能够实现更短的响应时间与更高的吞吐率。

WebSphere 支持会话信息不在集群实例之间共享的情况,也支持会话信息在集群实例之间共享的情况。对于第一种方案,会话信息没有被共享,也就是说,每个 WebSphere 实例管理自己的会话信息,WebSphere 实例之间信息不共享。如果会话被大量使用,并且如果用户需要潜在地被多个服务器服务,那么这种方案显然是没有用的。

第二种方案具有可伸缩性与容错性,表现在它支持会话跟踪的分布式特点并且不会产生瓶颈,同时还支持所有会话信息对所有实例的完全可用性。在这种方案下,使用的是持续性会话。当某个实例需要访问一个会话时,它与数据库进行交互以访问会话信息。需要说明的有趣一点是,几乎任何需要高级功能的软件,都有数据库的参与。另一个有趣问题是,使用持续性会话实际上是转移了工作负载。在这种情况下,减轻了数据库的繁重工作,用户需要确保数据库服务器也具有高可用性解决方案,例如 Oracle 的 Parallel Server 或 IBM 的 HACMP。

24.1.3 WebSphere 会话跟踪 API

已经说过,com.ibm.servlet.session.IBMSession 接口提供了对 IBM 扩展 servlet 会话的编程接口。这个接口只提供了 3 个方法:

- getUsername: 返回这个会话相关的已认证用户名。如果没有用户与这个会话相关,将返回 anonymous。
- isOverflow: 指出会话管理所需的内存是否溢出(也就是,是否超过了内存中应该存在的会话数限制)。
- sync: 与数据库同步(仅仅对持续性会话有用)。

虽然这个编程接口并不丰富,但 IBM 在这个接口的实现类的背后进行了许多工作。这个类实现了在 Administrative Console 中定义的持续性行为。它也加强了某些安全性,表现在一个 servlet 不能访问与某个已认证用户相关联的会话,除非该 servlet 在相同的用户身份下操作(也就是说,除非已经为相同的用户进行了认证)。

24.2 个性化 WebSphere

个性化 (personalization) 就是为一个特定的用户自定义 Web 应用程序所提供的内容与行为的过程。自定义可以基于显式地选择的偏好, 或者基于以前与应用程序进行的交互。前一种自定义方法被广泛应用于 Web 门户 (例如 Yahoo!), 而后一种方法被用于电子商城 (如 Amazon.com)。

这两种个性化方法的基础都是搜集并维护与用户相关的信息。所有与用户相关的信息的集合, 称为用户 profile (user profile)。WebSphere 提供了对维护用户 profile 的支持。用户 profile 由 `com.ibm.websphere.userprofile.UserProfile` 接口表示。这个接口的基本实现包括下列数据项: Address, Name, Phones, City, Nation, Employer, Fax number, Language, E-mail address, State/Province 与 Postal Code (地址、姓名、电话、城市、国籍、雇主、传真、语言、电子邮件地址、州/省与邮编)。这些数据项代表了最常用的人口统计信息, 可以用 accessor 方法 (例如 `getCity` 或 `getEmail`) 读取这些信息。

显然, 这些基本数据项并不能满足所有应用情况的需要。为了添加更多的字段, 需要有一个对 `UserProfile` 接口进行扩展的类, 并实现这个 `UserProfileExtender` 接口。后一个接口要求实现 `getNewColumns` 方法, 该方法提供了一个字符串数组, 用来保存新添加的列的名称。另一种实现自己的用户 profile 的方法就是扩展 `UserProfileProperties` 接口。这在本质上是一个散列表, 其中存储了名称-值对, 类似于 `java.util.Dictionary`。

还有一种实现自己的用户 profile 的方法就是利用底层的实现。WebSphere 将用户 profile 实现为实体 EJB, 并且用户可以对这个实体 bean 进行扩展。在这里笔者不详细描述这个方法, 因为本书在后面将详细讨论 EJB。但值得说明的是, 扩展用户 profile bean 具有一个重要的优点, 就是可以将存储在关系数据库中的遗留数据导入到 WebSphere Application Server 所管理的用户 profile 中。

在 4.0 版中, 可以手工地使用 XML 配置文件管理用户 profile (与 3.5 版相比, 这简直是一个倒退, 因为在 3.5 版中, 用户 profile 是由 WebSphere Administrative Console 管理的, 管理的方式类似于管理会话跟踪)。为了启用用户 profile 支持, 需要将 `userprofile.xml` 文件添加到 WebSphere 安装的 `properties` 目录中。

```
<?xml version="1.0"?>
<userprofile>
  <userprofile-enabled>true</userprofile-enabled>
  <userprofile-wrapper-class>
    <classname>
      com.ibm.servlet.personalization.userprofile.UserProfile
    </classname>
  </userprofile-wrapper-class>
  <userprofile-manager-name>
```

```
User Profile Manager
</userprofile-manager-name>
<userprofile-bean>
  <readonly-interface>
    com.ibm.servlet.personalization.userprofile.UP_ReadOnly
  </readonly-interface>
  <readwrite-interface>
    com.ibm.servlet.personalization.userprofile.UP_ReadWrite
  </readwrite-interface>
  <readonlyhome-interface>
    com.ibm.servlet.personalization.userprofile.UP_ReadOnlyHome
  </readonlyhome-interface>
  <readwritehome-interface>
    com.ibm.servlet.personalization.userprofile.UP_ReadWriteHome
  </readwritehome-interface>
  <readonly-JNDI-lookupName>UP_ReadOnlyHome</readonly-JNDI-lookupName>
  <readwrite-JNDI-lookupName>UP_ReadWriteHome</readwrite-JNDI-lookupName>
</userprofile-bean>
<userprofile-store>
  <database-userid>db2admin</database-userid>
  <database-password>db2passwd</database-password>
  <database-datasource>jdbc/Profiles</database-datasource>
</userprofile-store>
</userprofile>
```

这个文件很容易理解，并且在文件中包括对数据 wrapper 类的定义，这个类默认为 UserProfile，但也可以是已经说明的任何其他扩展。此外，用户可以选择底层的 EJB 实现。实际上，提供了两个独立的实现：一个用于只读访问，另一个用于读/写访问。对于每种情况，都可以指定本地接口、远程接口、JNDI 名（这些术语都与 EJB 部署相关，在本书的第 11 章与第 6 部分讨论）。最后，需要提供数据库用户、ID、口令以及用于用户 profile 持续性的 JDBC 数据源。

在安装 userprofile.xml 文件之后，就可以在企业应用程序中访问用户 profile。需要说明的是，这些应用程序应该使用默认的（或用户自己的）bean 来表示用户 profile。profile 的默认 EJB 在 WebSphere 安装目录的 lib/userprofile.jar 中。

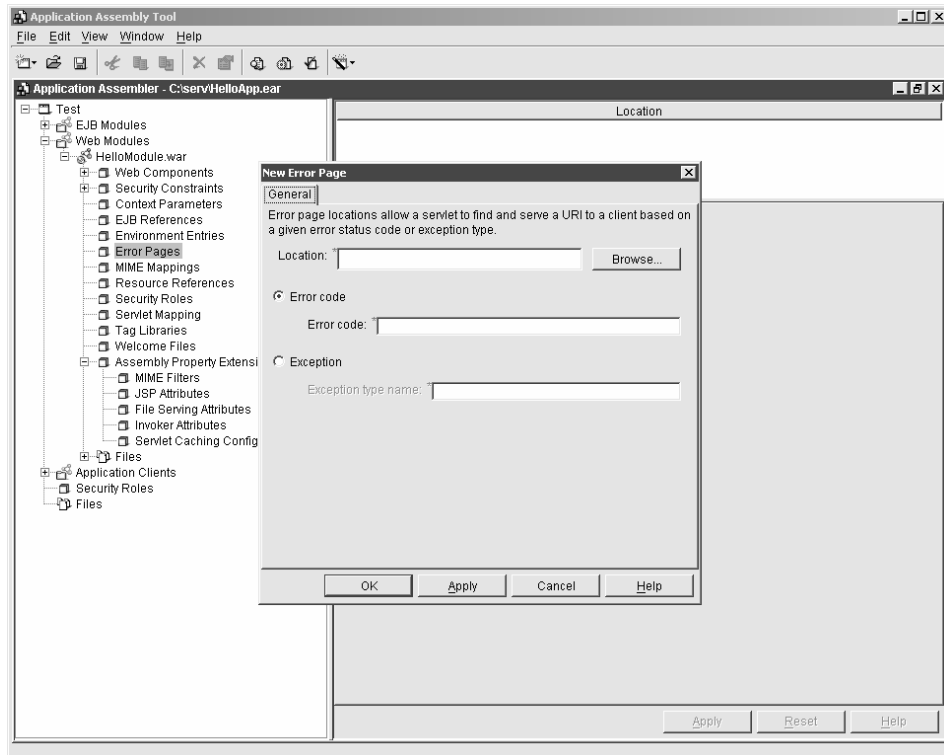
24.3 其他 IBM servlet 扩展

在构建 servlet 时，应该注意到 WebSphere Application Server 提供的某些附加的支持。这些附加功能不是 servlet 规范的一部分，但它们有效地节省用户做大量重复工作的时间。

24.3.1 错误处理

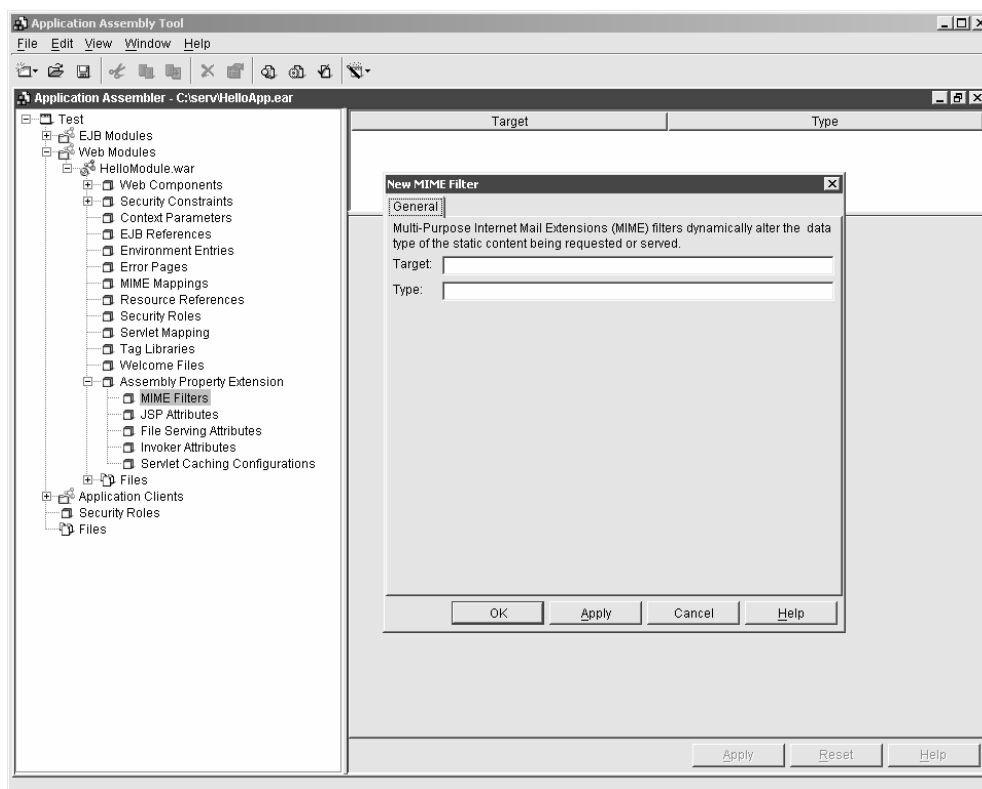
当 servlet 发出一个 `uncaught` 异常，或者当它显式地调用 `ServletResponse.sendError` 时，将报告 servlet 错误。默认情况下，WebSphere 将显示一个错误，用来指出发生错误的线程栈。虽然这一信息对于开发人员很有价值，但它们对用户没有帮助（实际上，对于一个典型的用户，当面向这些错误页面时，通常都比较迷惑或感到束手无策）。

为了解决这个问题，WebSphere 提供了 `com.ibm.websphere.servlet.error.ServletError-Reporter` 类，由这个类负责错误处理。作为开发人员，不需要直接与这个类交互。相反，可以在装配应用程序时在 Application Assembly Tool 中指定 New Error Page，如下图所示。



24.3.2 过滤 servlet 与 servlet chain

基于 MIME 的过滤需要配置 Web 容器，将某个特定 MIME 类型的 HTTP 响应转发到一个指定的 servlet 以进一步处理。可使用 Application Assembly Tool 的 New MIME Filter 对话框来指定这个 servlet：



servlet chain 是 servlet 将输出与响应转发到其他 servlet 进行处理的过程。显然，这样有利于提高模块化。servlet chain 的机制，就是一个 servlet 的输出作为下一个 servlet 的输入，而这个 chain 中的最后一个 servlet 的响应被返回给客户。为了使用 servlet chain，应用程序需要使用 WebSphere 内置的 servlet——com.ibm.websphere.servlet.filter.ChainerServlet。在这个 servlet 的初始化参数中，需要在 chainer.pathlist 参数中指定将要调用的 URL 列表。

24.3.3 servlet 事件

com.ibm.websphere.servlet.event 包允许跟踪与 servlet 相关的事件，主要是那些来自 servlet 生命周期中的事件。

为了在 servlet 中使用这个包，需要使用 ServletContextEventSource 接口，如下所示：

```
ServletContextEventSource source = (ServletContextEventSource)
    getServletContext().getAttribute(
        ServletContextEventSource.ATTRIBUTE_NAME);
source.addServletErrorListener(listener);
```

listener 对象应该实现 ServletListener 接口,它提供了与 servlet 生命周期事件相关的方法 init 方法的开始与结束、destroy 方法的开始与结束、加载、卸载。

需要说明的是,可以通过实现 ApplicationListener 接口为应用程序级事件定义侦听程序。这个接口提供了允许侦听程序跟踪应用程序可用性的方法。

24.3.4 代理请求与响应

com.ibm.websphere.servlet.request 包允许 servlet 生成转发给另一个 servlet 的请求。当一个请求需要被转发给另一个 servlet,但首先需要对这个请求进行某些修改时,这是很有用的。例如,假设用户希望修改从 GET 发送到 POST 的请求。由于 HttpServletRequest 并没有提供用来修改请求的接口,但 WebSphere 却允许利用 HttpServletRequestProxy 来完成这一工作。用户可以对这个类进行任意扩展,以定义自己的请求。为了使这个请求是一个 POST 请求,需要实现 getMethod()方法以确保它总是返回 POST。这个包也提供了 ServletInputStreamAdapter 类,允许对任何 Java 流进行封装,从而使得能够被用作使用这个请求的 servlet 的输入流。类似地,com.ibm.websphere.servlet.response 包支持使用底层 HttpServletResponse 调用的代理。这种代理没有请求代理作用那么大,但它能够对那些不会经常修改的 servlet 响应进行 cache。为了使用这个包,需要实现一个扩展 HttpServletResponseProxy 的响应对象。这个代理能够使用 ServletOutputStreamAdapter 类提供一个输出流,用来捕获输出。然后,后续的请求就可以使用 cache 的输出。

24.3.5 按照名称与服务文件调用 servlet

WebSphere 包括两个内置的 servlet,它们为 Web 应用程序提供可选的功能。第一个 servlet 允许使用某个 servlet 的类名(例如 SnoopServlet)访问此 servlet。为了支持这一功能,需要在 Application Assembly Tool 中启用 Web 模块的 IBM 扩展表项相对应的属性,如图 24-6 所示。但并不推荐这样做,因为这样暴露了应用程序的内部信息,并且为潜在的黑客提供了可能对他们有用的信息。

另一种方便来自文件服务 servlet。为了启用这一功能,应该在 Application Assembly Tool 中选中 File Serving Enabled 复选框,如图 24-6 所示。文件服务的效果是,在 Web 应用程序的文档根下的所有文件将自动被服务;不需要进行任何附加的配置步骤。这样就减少了在 Application Assembly Tool 中设置应用程序时必须做的大量工作,因为不需要显式地配置应用程序中的 servlet 与 HTML 页面(只要它们存储在应用程序的文档根下)。

警告: File Serving Enabled 选项虽然为静态页面的服务提供了极大的方便,但它也可能是一个安全性风险,因为任何放置在这个文档根中的文件都将被服务。因此,必须采取一些安全性步骤(适当的操作系统文件/目录许可,等等),以确保只有适当的文件被放置在这个文档根中。

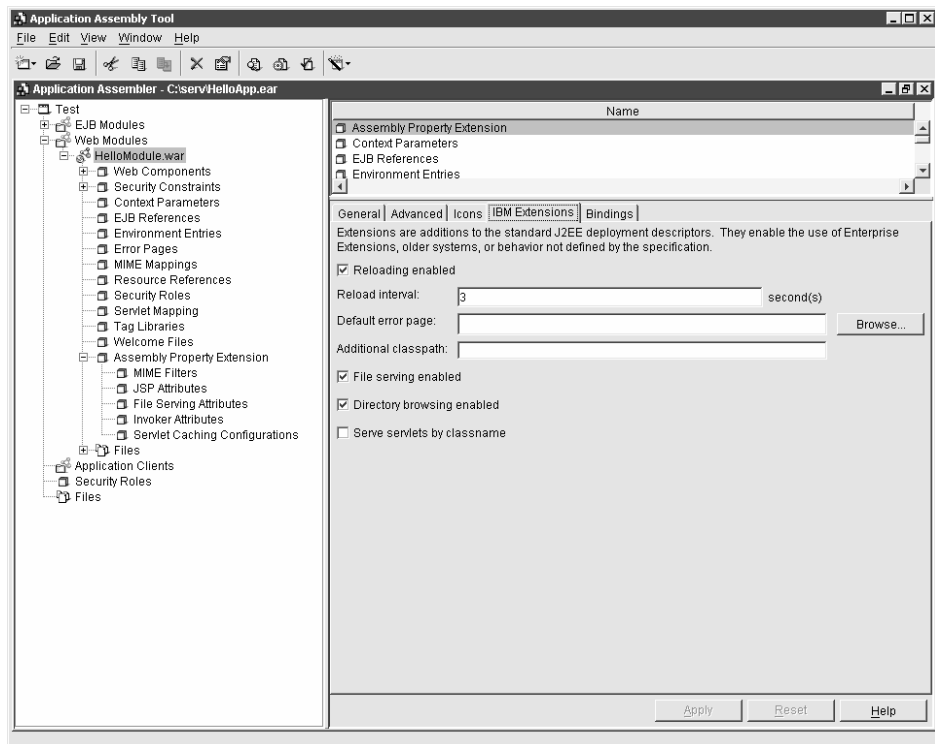


图 24-6 Application Assembly Tool，IBM 对 Web 模块的扩展

24.4 小 结

servlet 是 WebSphere Application Server 与 J2EE 框架支持的最重要结构之一。它们为几乎所有的 Web 应用程序提供了切入点。毫不奇怪，在 WebSphere 中提供了许多功能，用来在 servlet 上下文下，帮助开发人员管理应用程序内容与会话信息。本章覆盖了所有这些功能——其中某些是 IBM 特定的，另一些则是 J2EE 规范的一部分。

WebSphere 4 中的 servlet 常常被部署作为中央访问点并且作为后端功能的接口。在不久以前，这些 servlet 还是 Web 应用程序的表示层——用来动态地创建为浏览器服务的 HTML 的层。但现在已不是这样；在今天的 WebSphere 应用程序中，这一任务是使用 JSP 完成的，本书将在第 26~27 章讨论 JSP。

第 25 章 JSP 元素

JSP 是最适合为基于 Web 的应用程序构建动态用户界面的 Java 技术。这些应用程序使用表现技术（例如 HTML，DHTML，XML 与 Java applet）构造用户界面，使用 Java 服务器方代码进行业务处理。JSP 就是为了支持这些技术之间更方便、更平滑的集成而特别设计的，它允许使用 Java 服务器方代码产生动态内容。在前面的章中已经说明了 JSP 的功能模型与处理基础；在本章与下一章，将详细讨论如何编写 JSP 以及如何在 WebSphere 中部署 JSP。首先，将对 JSP 标记集合以及功能进行详细的讨论并给出示例，而与此同时还将讨论 JSP 引擎的强大功能。下一章将接着讨论关于 JSP 的一些例子。

25.1 JSP 语法

JSP 文件是文本文件，除了包括遵守 JSP 语法的元素集合之外，还包括 HTML（或 XML）标记、脚本以及其他标记集合。这里所描述的语法遵守 JSP 1.1 规范。在 WebSphere 4.0 中使用的 JSP 规范就是 JSP 1.1。JSP 规范定义了两个语法——标准语法，此语法基于类似于 ASP（Active Server Page）的标记伪指令，另一个语法则定义了标记集合作为 XML 的应用程序，并使用 XML 名称空间。在 XML 语法中，标记隐含地说明它们是什么，这不同于标准语法，标准语法仅仅在 HTML 代码与 Java 代码之间切换。例如，遵守 JSP 标准语法的 include 伪指令的标记如下：

```
<%@ include file="..." %>
```

而对于 JSP XML 语法，其形式为：

```
<jsp:directive.include file="..." />
```

虽然一般来说将 JSP 文件表示为 XML 文档并使用标记这种方法更好一些，但它们也存在一些缺点。例如，一个 XML 文档必须有一个单独的根元素，但 JSP 页面却没有这样一个自然存在的根；虽然可以定义这样的一个节点，但它毕竟是人造的。此外，XML 语法使用起来比较麻烦，而使用 JSP 对 Web 表示进行编码则相对快速、容易，因此这这也是一个严重的缺点。

JSP 页面的 XML 语法存在的优点就是，文档具有很好的格式并且它们类似于标记与属性的概念。在开始使用标记库时，由于标记库已经成为业界的发展方向，因此这一点是至关重要的。本书将在第 27 章更多地讨论标记库。虽然 XML 语法可能对工具与程序更有用，但标准语法（本章的其余部分将把标准语法称为 JSP 语法）对于简单的程序更易于使用，并

且对那些长期编写 JSP 的人员来说，使用标准语法几乎成了一种规范。

JSP 语法定义了许多元素。任何不属于这些元素的代码都不会被 JSP 引擎所处理，并且将被毫无修改地传递给结果页面。这些元素通常称为模板元素（template element），因为它们通常构成了将要在其中插入动态内容的表示代码。每个 JSP 元素属于下列分类之一：

- 注释
- 标准隐式对象
- 动作
- 伪指令
- 声明
- scriptlet
- 表达式
- bean 引用

25.2 注 释

JSP 文件中的注释遵守 HTML 中注释的一般模式。通过嵌入下列形式的行，就可以在浏览器页面中传递一个注释。

```
<!-- put comment here -->
```

如果要求注释出现在 JSP 中，但不传递给结果页面，那么可以嵌入一个遵守 scriptlet 中 Java 规范的注释，编写下列代码：

```
<%-- put comment here --%>
```

需要说明的是，这并不意味着注释被 JSP 引擎传递到 Java 文件中。

还有一种放置注释的方法就是在 scriptlet 元素中使用普通的 Java 注释，如下所示：

```
<% // put comment here %>
```

或者，编写如下形式的代码：

```
<% /* put comment here */ %>
```

25.3 标 准 对 象

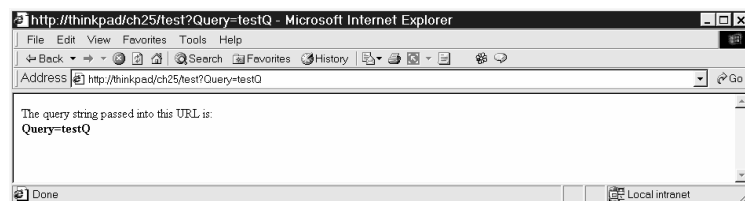
由于 JSP 模型遵守 servlet 模型（并且由于 JSP 实际上被编译成 servlet 代码），因此一个标准的对象集合总是可用的。可以在没有显式地声明的情况下就引用这些对象，并且这些对象形成了 servlet 中的“环境”。

在第 4 章，已经见到过几个隐式地定义的 JSP 对象，例如 out，request 与 response。为了保持讨论的完整性，这里将再次简要描述这些对象。

request 对象是一个 javax.servlet.HttpServletRequest 类型的对象(假设 JSP 通过 HTTP 被访问)。out 所引用的对象在规范中被定义为具有 javax.servlet.jsp.JspWriter 类型，此对象类似于 servlet 的打印输出对象。下面的例子使用这两个对象，实际上，这个例子可能是关于使用这些对象的最简单例子：

```
<HTML>
<BODY>
The query string passed into this URL is:<BR><B>
<%
    out.println(request.getQueryString());
<%
</B>
</BODY>
</HTML>
```

访问这个 JSP 所得到的结果如下：



response 对象的类型为 javax.servlet.HttpServletResponse。一个同样非常简单的、将客户重定向到另一个 URL 的例子如下所示：

```
<HTML>
<BODY>
<%
    response.sendRedirect("http://www.ibm.com/websphere");
%>
</BODY>
</HTML>
```

上面所提到的类型都是在 JSP 规范中定义的接口。由于 WebSphere 是 JSP 标准的实现，因此其中使用的实际类有些不同。为了查看输出，可以使用一个 JSP，如下所示：

```
<HTML>
<BODY>
<% out.println("Request is of type " + request.getClass()); %>
<BR>
<% out.println("Response is of type " + response.getClass()); %>
```

```
<BR><% out.println("Out is of type " + out.getClass()); %>
</BODY>
</HTML>
```

所得到的输出显示如下：



可以看到，request 与 response 是作为 com.ibm.servlet.engine.webapp 包的一部分实现的，它也被用于 servlet。out 对象来自 org.apache.jasper.runtime 类，表示 WebSphere 的 JSP 引擎从 Apache/Tomcat 对 JSP 的实现中继承而来。

JSP 模型还提供了下列隐式对象，它们在本质上都是从底层的 servlet 模型继承而来的：

- page：page 对象可以被看作本地的 this 对象。当页面被实例化时，page 对象确实与 this 对象相等。在 JSP 的生命周期中，this 对象可能不表示页面（例如，它可能表示包含在其中的一个子页面）。page 对象总是被确保表示整个 JSP。
- session：session 对象是 HttpSession 的一个实例，用于会话跟踪以及存储在整個会话中共享的对象。
- application：application 对象是所产生的 servlet 的 ServletContext 对象。
- config：config 对象是 ServletConfig 的一个实例，提供关于 JSP 的初始化参数的访问。
- exception：exception 对象是一个仅仅在被标记为错误页面（使用下一节中描述的 errorPage 伪指令）的页面中可用的 Java 对象。对于这些页面，exception 对象保存了从前一个页面发出的异常。
- pageContext：pageContext 对象是一个保存有 request，response，application，config，session 与 out 对象的容器。页面上下文提供了关于该页面当前执行上下文的一个已更新视图。

25.4 JSP 伪指令

JSP 伪指令是向 JSP 引擎发出的指令。它们构成了一个规范集合，用来告诉 JSP 引擎应该将什么编译到所产生的 servlet 中，以及如何进行编译。伪指令可以使用 JSP 语法，也可以使用 XML 语法。其中 JSP 语法为：

```
<%@ directive_name attribute="value" %>
```

XML 语法为：

```
<jsp:directive.directive_name attribute="value" />
```

其中最重要的就是 `page` 伪指令，这个伪指令可用来设置页面特定的属性。`language` 属性指定了脚本中所用的语言。对于 JSP，`language` 属性的惟一合法值就是 `java`；任何其他值都将导致 JSP 引擎失败。这也是默认值；因此，除非支持其他的语言，否则这个标记就没有什么实际用处。这个伪指令的其余属性定义了所产生的 `servlet` 的外观，以及这个 `servlet` 具有什么样的行为。`page` 伪指令的一般语法如下：

```
<%@ page
    language=<scriptingLanguage>
    extends=<className>
    import=<importList>
    session=<true|false>
    buffer=<none|sizekb>
    autoFlush=<true|false>
    isThreadSafe=<true|false>
    info=<info_text>
    errorPage=<error_url>
    isErrorPage=<true|false>
    contentType=<ctinfo>
%>
```

`import` 属性用来告知 JSP 引擎应该在所生成的代码中包括什么 `import`（导入）语句。由于 JSP 引擎首先处理 JSP 文件并且为 `servlet` 创建一个 Java 类，然后才对它进行编译，因此导入语句是有用的，表现在它们允许在后来向 JSP 文件中插入代码以使用简写（当它引用 Java 类时）。默认情况下，`java.lang.*`，`java.servlet.*`，`javax.servlet.http.*`与 `javax.servlet.jsp.*`被导入。例如，下列伪指令允许在 JSP 中使用 IBM XML 解析程序类与 DOM 类：

```
<%@ page import="com.ibm.xml.parser.*,org.w3c.dom.*" %>
```

需要说明的是，与 Java 代码一样，导入语句应该引用能够在 `classpath` 中发现的类与包。类似地，`extends` 属性用来告知 JSP 引擎某个类是否应该作为所产生的 `servlet` 类的超类。一般不推荐使用这个属性，因为它限制了 Web 容器的应用，使得必须与 JSP 相应地实现 `servlet`。`contentType` 属性用来告知 JSP 引擎在响应流中应该使用什么内容类型。对响应流的设置等价于调用 `response.setContentType`。

`page` 伪指令的其他属性包括：

- `autoFlush`：当缓冲器满或超标时，这个属性决定页面的行为。其默认值为 `true`，表示当缓冲器超标时内容被刷新。如果为 `false`，那么当缓冲器超标时将产生一个异常。
- `buffer`：用来决定是否对页面进行缓存，以及如果被缓存，缓冲器大小为多少。如果

这个值设置为 0，那么页面不被缓存。缓冲器大小以 kB 来指定，例如，32k。

- `errorPage`：指定当出现 `uncaught` 异常时应该调用的 URL。
- `info`：定义了一个由所产生的 `Servlet` 的 `getServletInfo` 方法所返回的字符串。
- `isErrorPage`：默认设置为 `false`。如果设置为 `true`，表示该页面用作一个错误页面。在这种情况下，页面就能够使用隐式异常对象访问错误信息。
- `isThreadSafe`：用来告知 JSP 编译器关于页面的线程安全性。默认为 `true`。如果为 `false`，那么对相同页面的多个调用将基于这些调用到达的顺序以串行的方式处理。
- `session`：指出当前页面是否参与会话。这个属性默认设置为 `true`。如果设置为 `false`，那么会话隐式对象就没有被定义。

在 JSP 1.1 规范中引入的一个特殊伪指令就是 `taglib` 伪指令，用来扩展 JSP 的标记集合。这个伪指令有两个属性：`uri` 与 `prefix`。前者是一个 URI (Uniform Resource Identifier，统一资源标识符)，它指向 JAR (Java Archive) 文件，在这个文件中包含了一个用来描述新标记的标记库描述符。`prefix` 属性指出用来表示页面中新标记的关键字。例如，下列伪指令允许使用一个前缀为 `viryanet` 的标记集合。

```
<% taglib uri="http://www.viryanet.com/taglib/viryatags.jar"
    prefix="viryanet" %>
```

如果在标记库描述符中定义了一个名为 `serviceCall` 的标记，那么就可以使用下列代码：

```
<viryanet:serviceCall>Call 911</viryanet:serviceCall>
```

注意：第 27 章将详细描述标记库与 `taglib` 伪指令。

25.5 声 明

Java 声明（变量与方法）可以被添加到 JSP 文件中。声明可以具有 JSP 语法，也可以具有 XML 语法。JSP 语法为：

```
<%! java code %>
```

XML 语法为：

```
<jsp:declaration> code </jsp:declaration>
```

所有这样的代码都将被添加到从 JSP 生成的 `Servlet` 类中，因此，这些代码在类的任何地方可用。与 `scriptlet` 和表达式不同，这些代码不是放在 `Servlet` 的服务方法中，而是作为类定义的一部分。这样就允许向所产生的 `Servlet` 类添加方法与类成员变量。

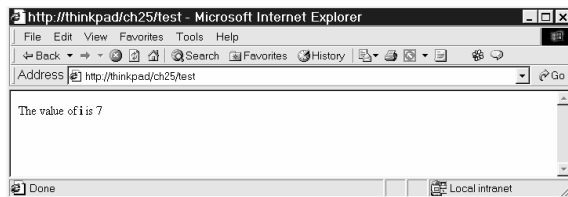
25.6 scriptlet 和表达式

JSP 包括一个方便的语法，可以在其中写 Java 代码。通过将 Java 代码包围在 `<%与%>` 标记之间，就能够告诉 JSP 引擎将该代码复制到 servlet 处理方法中。这种 Java 代码段被称为 scriptlet。此外，一个类似的语法也允许定义表达式，对表达式进行求值并且将值写入到输出流中。表达式不同于 scriptlet，表现在前者的打开标记使用的是 `<%=`，并且嵌入的代码被看成是一个能够产生值的表达式。由于表达式被嵌入在 Java 代码中，因此编写表达式到数据流上时不应该使用分号（`;`）。

例如，考虑下列 JSP：

```
<HTML><BODY>
<% int i = 6; %>
<% i = i + 1; %>
The value of <B>i</B> is
<%= i %>
</BODY>
</HTML>
```

这个 JSP 使用简单的 Java 代码来设置变量 `i`，并且对这个变量进行自增。它使用一个表达式来输出值。这个 JSP 的输出如下所示：



关于 scriptlet，需要说明的一个有趣事情就是能够真正将 HTML 输出与 Java 源代码混合在一起。例如，考虑下面的页面：

```
<HTML>
<BODY>
This is an example of what happens to a scriptlet in the output.
Notice how the line will be missing when we observe the resulting HTML's
source.
<% int i = 1; %>
<BR/>
This is an even better example.
Based on how the scriptlet code segments evaluate
we get different lines that are really part of the HTML:
<BR/>
```

```
<% if (i == 1) { %>
i really does equal 1
<% } else { %>
i does not equal 1
<% } %>
</BODY>
</HTML>
```

在下面这个简单的 JSP 文件中，需要注意两点。第一，将 JSP 文件与结果 HTML 页面进行比较时，可以发现这些行几乎完全匹配，但有一些空白行，最初是存在 scriptlet 的：

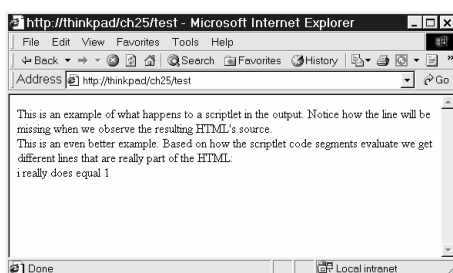
```
<HTML>
<BODY>
This is an example of what happens to a scriptlet in the output.
Notice how the line will be missing when we observe the resulting
HTML's
source.

<BR/>
This is an even better example.
Based on how the scriptlet code segments evaluate
we get different lines that are really part of the HTML:
<BR/>

i really does equal 1

</BODY>
</HTML>
```

第二个有趣的事情就是，“if”短语实际上是利用一个 scriptlet 集合来编写的，但与之相对应的 then 和 else 的输出都是由 HTML 写的。因此，在一个短语中的代码不仅使用不同的编程语言进行组合，而且使用了完全不同的处理模型。这个 JSP 的输出如下所示：



关于如何理解这种 JSP 在所产生的 Java servlet 代码中运行，其关键如下：

```
import javax.servlet.*;
```

```
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.PrintWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;
import java.io.ByteArrayOutputStream;
import org.apache.jasper.compiler.ibmtsx.*;
import org.apache.jasper.compiler.ibmdb.*;
import java.sql.SQLException;

public class test_jsp_10 extends HttpJspBase {

    static char[][] _jspx_html_data = null;

    static {
    }
    public test_jsp_10( ) {
    }

    private static boolean _jspx_inited = false;

    private static boolean checkedAttributeIgnoreException = false;

    private static boolean throwException = true;

    public final void _jspx_init() throws JasperException {
        ObjectInputStream oin = null;
        int numStrings = 0;
        try {
            InputStream fin =
this.getClass().getClassLoader().getResourceAsStream("test_jsp_10.dat");
            oin = new ObjectInputStream(fin);
            _jspx_html_data = (char[][] ) oin.readObject();
        } catch (Exception ex) {
            throw new JasperException("Unable to open data file");
        } finally {
            if (oin != null)
```

```

        try { oin.close(); } catch (IOException ignore) { }
    }
}

public void _jspService(HttpServletRequest request, HttpServletResponse
response)
    throws IOException, ServletException {

    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    String _value = null;
    setBooleanIgnoreException();
    try {

        if (_jspx_inited == false) {
            _jspx_init();
            _jspx_inited = true;
        }
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html;charset=Cp1252");
        pageContext = _jspxFactory.getPageContext(this,
            request, response,
                "", true, 8192, true);
        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();

        // begin
        [file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.
jsp";from=(0,0);to=(5,0)]
            out.print(_jspx_html_data[0]);
        // end
        // begin
        [file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.
jsp";from=(5,2);to=(5,14)]
            int i = 1;
        // end
        // begin

```

```
[file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.jsp";from=(5,16);to=(11,0)]
    out.print(_jspx_html_data[1]);
    // end
    // begin
[file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.jsp";from=(11,2);to=(11,17)]
    if (i == 1) {
        // end
        // begin
[file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.jsp";from=(11,19);to=(13,0)]
    out.print(_jspx_html_data[2]);
    // end
    // begin
[file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.jsp";from=(13,2);to=(13,12)]
    } else {
        // end
        // begin
[file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.jsp";from=(13,14);to=(15,0)]
    out.print(_jspx_html_data[3]);
    // end
    // begin
[file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.jsp";from=(15,2);to=(15,5)]
    }
    // end
    // begin
[file="C:\\WebSphere\\AppServer\\installedApps\\ch25.ear\\ch25.war\\test.jsp";from=(15,7);to=(18,0)]
    out.print(_jspx_html_data[4]);
    // end

} catch (Exception ex) {
    if (out.getBufferSize() != 0)
        out.clearBuffer();
    pageContext.handlePageException(ex);
} catch ( Throwable t) {
    if (out.getBufferSize() != 0)
        out.clearBuffer();
    pageContext.handlePageException(new Exception(t.getMessage()));
} finally {
```

```

        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}
private void setBooleanIgnoreException() {
    if (checkedAttributeIgnoreException) return;
    checkedAttributeIgnoreException = true;
    String initParamIgnoreException =
        (String)(getServletConfig().getInitParameter
            ("jsp.repeatTag.ignoreException"));
    if ((initParamIgnoreException != null) &&
        (initParamIgnoreException.toLowerCase().equals("true"))){
        throwException = false;
    }
}
}
}

```

下面将分析 `_jspService` 方法的实现，这个方法代表 JSP 执行动作并写输出。此方法调用 `out.print` 输出 HTML 数据，这些数据存储在一个称为 `_jspx_html_data` 的特殊数组中，此数组在 `_jspx_init` 方法中初始化。从效果上看，这个方法提供的是一系列命令，这些命令是 Java 命令（JSP 中 scriptlet 代码）与 `out.print` 调用的混合。`out.print` 调用在效果上将 HTML 从 JSP 源文件写到 servlet 的输出流。这种 Java 代码与复制代码的组合是 servlet 引擎所执行的代码生成过程的重要部分。需要说明的有趣一点是，为 JSP 生成的 Java servlet 代码在许多方面都是 JSP 的镜像。在 JSP 中，嵌入在 Java 代码中的 scriptlet 对 HTML 代码是“外来的”。在所生成的 servlet 中，来自 scriptlet 的 Java 代码可以看成“本地”代码，而 HTML 代码是从 JSP 源文件中复制过来的。

25.7 JSP 动作

JSP 动作仅仅在基于 XML 的语法中使用，如下所示：

```
<jsp:action-name attribute="value" />.
```

虽然不同的动作有不同的属性，但所有动作都有两个共同的属性：`id` 与 `scope`。`id` 属性允许 JSP 开发人员惟一地对动作元素命名，从而提供了一种从页面中引用动作的方法。`scope` 属性用来决定动作元素的生命时间。这个属性也可以有 4 个不同的值：`page`，`request`，`session` 与 `application`。可以想象，这些值分别对应 JSP 中的隐式对象。当 `scope` 属性被设置为 `page` 时，由这个动作标记所创建的对象仅仅在它创建该对象的页面中可用。设置为 `request` 时，所创建的对象能够被服务于相同请求的页面共享，并且在该请求完成时释放此对象。设置为 `session` 时，所创建的对象与页面中的会话相关联，并且在此会话中创建了该对象，因此

这些对象被相同会话中处理某个请求的所有页面共享。一旦会话完成，这些对象将被释放。设置为 application 时，创建的对象在所有属于相同 Web 应用程序的页面中可用。

JSP 动作提供了一种实例化并使用 JavaBean 的方法。JavaBean 是可重用的 Java 组件，已经在第 10 章讨论了 JavaBean。如果读者对 JavaBean 的概念还不熟悉，为了这里的讨论需要，可以将它们理解为具有一系列属性的 Java 对象。每个这样的属性都可以使用 get 方法来获取，并使用 set 方法来设置。useBean 动作用来查找或创建一个新 JavaBean 实例。需要说明的是，这个动作并不是简单地实例化一个 bean；它首先将查找在相同的范围中是否存在一个已经定义的 bean（基于指定的属性进行查找）。这样就提供了一种简单而又强大的机制，实现在一个会话或 Web 应用程序中对象的共享。useBean 动作使用在 id 属性中指定的名称创建一个新脚本变量。然后，可以在此页面中使用这个变量。

除了常见的 id 与 scope 属性之外，useBean 动作还有 3 个属性：

- class：这个 bean 的 Java 实现类。
- type：bean 应该遵守的 Java 类型（也就是，所实现的类、超类或一个接口）。
- beanName：由 bean 开发人员所选择的 bean 名称，它是 JavaBean 模型的一部分，并且在一定程度上与实现类相对应。

必须指定 type 属性或 class 属性。如果指定了这两个属性，那么所提供的 class 必须遵守所提供的 type。当使用 beanName 时，就不能使用 class 属性，但必须使用 type 属性。也就是说，不允许仅仅指定 beanName 而不指定任何 type 属性。

还有两个附加的动作，可用来设置与获取 bean 的属性。分别是 getProperty 动作与 setProperty 动作。在使用 setProperty 动作时，可以利用 name 属性标识 bean，使用 property 属性来指定应该被修改的属性，并且利用 value 属性来指定所使用的值。需要说明的是，值也可以是 JSP 表达式，例如：

```
<%= "Today is " + new Date() %>
```

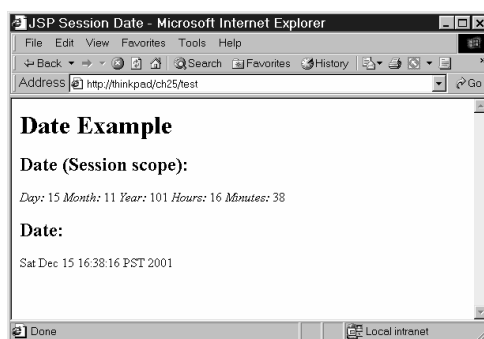
另一种使用 setProperty 动作的方法，就是使用 param 属性，而不是 value 属性。这样，就从请求（HttpServletRequest 对象）参数各自的名称中获取值。当希望设置一个 bean 的属性为请求参数并且这个请求参数正好与这个 bean 的属性名称相同时，那么 setProperty 动作的 property 属性就可以被忽略。setProperty 动作的一个很棒的功能就是它允许使用星号（*）作为通配符，这就意味着，所有其名称与请求参数相匹配的属性都将被赋给相同的参数值。getProperty 动作的使用更简单，它只有一种形式。这个动作使用两个必需的参数，分别是 bean 名与属性名。动作将转换成一个字符串：如果属性不是字符串类型，那么将隐式地使用 toString 方法，把此对象转换成一个字符串。

另一种动作就是 include 动作。这个动作用来将其他页面的输出插入到当前页面中。这些页面可能是静态页面（例如 HTML 页面），也可能是动态页面（例如 JSP 页面）。当遇到这个动作时，JSP 的执行将被暂停，并且被包括的页面接着进行处理。include 伪指令有两个必

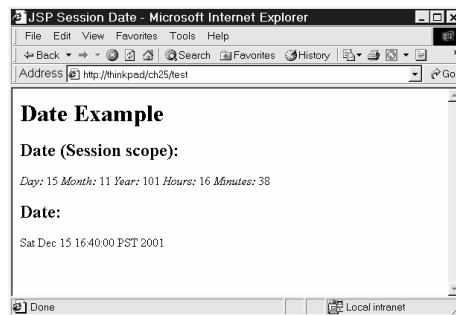
备的属性：page 与 flush。第一个属性定义了将要被包括的页面（它必须是相同服务器上的一个相对 URL）；第二个属性用来说明在包括页面之前，是否需要刷新这个被包括的页面及其缓冲器。在 JSP 1.0 中，这个值必须为 true，因为在其底层 servlet 模型中存在某些限制。在前面的“JSP 语法”一节中，已经讨论了 include 伪指令，这个伪指令完全不同于 include 动作。include 伪指令在 JSP 被编译之前，获取一个指定的页面并将它插入到 JSP 中。为了演示这些动作，考虑下面的 JSP：

```
<HTML>
<HEAD><TITLE>JSP Session Date</TITLE></HEAD>
<BODY><H1>Date Example</H1>
<jsp:useBean id="mydate" scope="session" class="java.util.Date" />
<jsp:useBean id="pagedate" scope="page" class="java.util.Date" />
<H2>Date (Session scope): </H2>
<I>Day:</I>
<jsp:getProperty name="mydate" property="date" />
<I>Month:</I>
<jsp:getProperty name="mydate" property="month" />
<I>Year: </I>
<jsp:getProperty name="mydate" property="year" />
<I>Hours:</I>
<jsp:getProperty name="mydate" property="hours" />
<I>Minutes:</I>
<jsp:getProperty name="mydate" property="minutes" />
<H2>Date: </H2>
<%= pagedate.toString() %>
</BODY>
</HTML>
```

这个 JSP 声明了一个实例化 java.util.Date 类的 JavaBean 对象，它使用 session 范围。然后，这个 JSP 又声明了另一个具有 page 范围的 date 对象，并且输出这两个对象。这样，所得到的效果就是当会话启动时能够捕获日期及当前日期。这个 JSP 的输出如下所示：



如果单击浏览器的 Refresh 按钮，将会见到一个类似的输出；第一个时间是相同的，而第二个时间则改变为当前时间。



下面考虑将这个 JSP 分为两部分。第一部分就是第 7 章中 SessionDate.jsp 页面的修改版，它使用了 include 动作，如下所示：

```
<HTML>
<HEAD><TITLE>JSP Session Date</TITLE></HEAD>
<BODY><H1>Date Example</H1>
<jsp:include page="/PrintDate.jsp" />
<H2>Date (Session scope): </H2>
<I>Day:</I>
<jsp:getProperty name="mydate" property="date" />
<I>Month:</I>
<jsp:getProperty name="mydate" property="month" />
<I>Year: </I>
<jsp:getProperty name="mydate" property="year" />
<I>Hours:</I>
<jsp:getProperty name="mydate" property="hours" />
<I>Minutes:</I>
<jsp:getProperty name="mydate" property="minutes" />
<H2>Date: </H2>
<%= pagedate.toString() %>
</BODY>
</HTML>
```

PrintDate.jsp 页面用来获取日期：

```
<HTML><BODY>
<jsp:useBean id="mydate" scope="session" class="java.util.Date" />
</BODY>
</HTML>
```

为了获取称为 mydate 的 bean 并从其中获取日期，后一个页面使用了 useBean 动作与

getProperty 动作。第二个页面能够获取此信息，因为它与第一个页面共享相同的会话。这个页面的输出是相同的，但在背后实现的机制完全不一样。PrintDate.jsp 页面首先被执行，然后被嵌入到整个页面的输出中。这个关于 include 动作的例子看上去不是很有用处；但一般来说，这个动作与 include 伪指令对提升页面以及页面段的可重用性是非常有用的。

另外两个动作是 forward 与 plugin。forward 动作可用于将请求转发到另一个页面上。它有一个称为 page 的参数，在其中包含有请求将要被转发到的目标 URL。同样，这也必须是一个在相同服务器上的相对 URL。plugin 动作用来将 Java 组件（JavaBean 或 applet）插入到页面中。这些组件被嵌入在页面中，并且在客户方的客户 JRE（Java Runtime Environment，Java 运行时环境）（通常是浏览器）中执行。plugin 动作类似于将 applet 嵌入到 HTML 页面中。由于某些组件在不同的浏览器上行为有所不同，因此这个动作实际上对特定浏览器定义提供支持。plugin 动作具有下列属性：

- type：类型可以是 applet 或 JavaBean。
- jreversion：需要在客户方对 JRE 进行细小的修改（默认为 1.1）。
- iepluginurl：一个 URL，在这个 URL 上发现了与 Microsoft Internet Explorer 兼容的组件版本（如果它在客户 cache 中不可用，那么将下载它）。
- nspluginurl：一个 URL，在这个 URL 上发现了与 Netscape Navigator 相对应的版本。
- code, align, archive, height, hspace, name, vspace, title 与 width：这些属性对应于 applet 的 codebase 标记中所用的 HTML 参数。

25.8 IBM JSP 支持

与 WebSphere Application Server 的先前版本不同，在 4.0 版中没有包括许多 IBM 特定的 JSP 功能。这可能是为了遵守 J2EE，应用程序服务器的所有方面都转向某种标准化。但这并不是说 IBM 就没有提供任何特殊的功能，而是说这种特殊的功能比较少。对于初学者来说，WebSphere 中提供的所有特殊 servlet 功能（在第 24 章详述）也同样应用于 JSP，因为所有 JSP 页面都将转换成 servlet。

25.8.1 IBM JSP 标记

在 WebSphere Application Server 3.5 版本中，IBM 引入了一个特殊的标记家族，可用于变量数据。它们在本质上是对标准 JSP bean 动作的扩充，允许更容易地使用从数据库获取的数据。这些标记的前缀是 tsx，在 4.0 版中支持这些标记。虽然 WebSphere 3.5 使用一个不同的 JSP 规范（也就是 1.0 规范），但 JSP 1.1 规范需要对 JSP 1.0 向后兼容；因此这些工作在 4.0 版中没有改变。

tsx:dbconnect 标记用来建立数据库连接，这个标记的属性如下：

- id：代表该连接的 bean 的 id。
- userid：数据库用户 id，也可以使用 tsx:userid 标记来指定。
- passwd：数据库用户口令，也可以使用 tsx:userid 标记来指定。
- url：数据库的 JDBC URL。
- driver：数据库的 JDBC 驱动程序。

tsx:dbquery 标记用来使用一个给定的连接执行一个 SQL 查询。这个标记的属性有：id，代表输出 ResultSet 的 bean 的 id；connection，提供关于连接 bean 的 id；limit，这是一个可选的属性，用来指定将要被返回的最大行数。tsx:dbquery 标记的标记体中包括将要被执行的 SQL 查询。

下列 JSP 段演示了这两个标记的使用：

```
<tsx:dbconnect id="con", userid="db2admin", passwd="db2admin"
  url="jdbc:db2:database", driver="com.ibm.db2.jdbc.app.DB2Driver"/>
<tsx:dbquery id="query" connction="con">
  select name,phone from phonetable
</tsx:dbquery>
```

现在有了一个连接与一个查询，那么下一步干什么呢？这时可以使用 tsx:repeat 标记与 tsx:getProperty 标记从查询中实际地获取信息。tsx:repeat 标记对查询结果的每一行进行迭代，tsx:getProperty 标记则允许从当前行中获取某个列的值。tsx:repeat 标记有 3 个可选的参数：index，用来标识重复块中的索引（当前行）；start，用来指定这个块的起始索引值（默认为 0）；end，索引的结束值（可选），它必须比 start 高。tsx:getProperty 标记有一个 name 属性，用来指定与此查询相关的属性以及属性名。

继续分析前面的 JSP 段，也可以使用查询来显示姓名与电话号码列表，如下所示：

```
<tsx:repeat index=ind>
  <B>Name:</B>
  <tsx:getProperty name="query" property="name" />
  <B>Phone:</B>
  <tsx:getProperty name="query" property="phone" />
</tsx:repeat>
```

这个标记集合在本质上提供了非常简易的方法，用来避免显式地使用 JDBC 访问数据库。这些标记集合具有很强的功能，表现在可以嵌套重复标记并使用 tsx:modify 标记修改数据库，这类似于 tsx:dbquery，只是前者将更新数据库。并且，它与标记库具有相同格式和用法，将在第 27 章讨论标记库。

25.8.2 部署问题

IBM JSP 处理程序为每个 JSP 文件创建并编译一个 servlet。与某个 JSP 相对应的 Java 源文件放置在下列目录中：

```
<WebSphere Installation  
Dir>\temp\<hostname>\<servername>\<webmodulename>
```

相关联的.class 文件与.dat 文件也放置在相同的目录中(.dat 文件保存了 JSP 的静态部分——相同的部分存储在_jsp_xhtml_data 数组中,这在前面已经说明过)。

可以使用 Application Assembly Tool 为每个 JSP 设置下列属性。这些属性存储在 IBM 为 Web 模块扩展的 XML 文件 (ibm-web-ext.xml) 中:

- keepgenerated: 指定所产生的 Java 文件是否被保存 (可能值是 true 或 false)。需要说明的是,即使将这个值设置为 false,也将会产生 Java 文件并放置在 ibm-web-ext.xml 目录中。如果迅速查看,将会发现该文件出现在此目录中然后消失。
- scratchdir: 可用来指定一个不同的目录,用于放置.class, .java 与.dat 文件,而不是将这些文件放在 WebSphere 安装目录的 temp 目录下。
- defaultEncoding: 如果没有 contentType 伪指令,则用该属性来指定所使用的默认编码方式。如果这个元素不存在,那么将使用系统属性 file.encoding。
- jsp.repeatTag.ignoreException: 允许控制先前发布中<tsx:repeat>标记所表现出来的一些特异行为。在这个标记的实现中,捕获了各种异常,但没有发出这些异常,从而会导致出现不可预测的结果。这个标记默认设置为 false,但如果为了给在老版本的 WebSphere 中开发的页面提供向后兼容能力,那么就需要将这个值设置为 true。

另外需要说明的有趣事情是,一旦 Web 应用程序被部署,也可以在 Web 模块目录中修改 JSP 页面。这样将会导致 JSP 引擎重新生成 servlet 并重新编译它们。这一点在开发过程以及代码调试过程中是非常方便的。但是,在产品环境下存在这种特点则是危险的。

IBM 提供的另一个非标准功能就是对 JSP 文件的批量编译。其主要思想就是为第一次对 JSP 页面的访问提供更快的响应;这样做是有必要的,因为在默认情况下,初次请求将导致启动生成 servlet 并编译它的过程。在 Application Assembly Tool 中设置 Load on Startup 之后,就启用了这一功能。

25.9 小 结

在所有 WebSphere 技术中,JSP 可能是开发人员使用最多的技术之一。JSP 页面在大多数 Web 应用程序中被用作用户界面层;因此,它们被非常频繁地使用。大多数工程都创建大量的 JSP 页面。这样,对 JSP 的管理就是 WebSphere 开发工作的重要部分。本章提供了许多关于 JSP 页面的信息,并且特别描述了 JSP 页面的标记集合。

JSP 技术正在不断发展,很重要的一点就是必须跟上最新的 JSP 技术。为此,本书将在下面的两章中继续讨论 JSP。下一章讨论关于 JSP 编程与部署的更复杂例子;第 27 章则讨论另一个重要的内容——标记库。

第 26 章 JSP 示例

前几章详细讨论了关于 JSP 的各个方面。本章将提供一些基于 JSP 应用程序的示例。第 4 章的“Hello World”示例以及第 25 章的那些例子已经给出了如何开发 JSP 的基本概念。本章目标是进一步加深对 JSP 编程的理解。

26.1 对一个数进行因式分解的 JSP

第一个例子先考虑一个简单的算术问题：对整数进行因式分解。将要构建的这个 JSP 要求用户输入一个整数，然后求出从 2 开始直到这个数的所有整数因子。

这个例子演示了 Java 控制流结构如何与 JSP 标记非常整洁地结合在一起。可以在 for 语句或 while 语句的中间关闭 scriptlet，放置一些 HTML 文本，然后继续此 scriptlet 以完成循环语句。在出现标记库之前，这是编写 JSP 的一种首选方法，本书将在第 27 章讨论标记库，并且，我们建议尽可能将 Java scriptlet 与 HTML 元素分开，只要这样做不会使得代码完全不可读即可。

```
<HTML>
<HEAD><TITLE>Number Factoring</TITLE></HEAD>
<BODY>
<%
    String param = request.getParameter("number");
    if (param == null) {
%>
        <H1>Enter a number:</H1>
        <FORM action="factor.jsp">
            Number:<INPUT type="text" name="number" title="Number:">
        </FORM>
<%
    }
    else {
        int number;

        try {
            number = new Integer(param).intValue();
        }
        catch (Exception ex) {
```

```

        number = 0;
    }

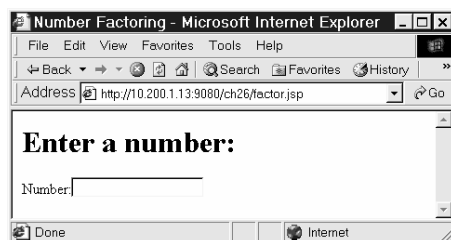
    if (number <= 0) {
%>
        <H1><FONT color="0x0000ff">Please Enter a Positive
            Number</FONT></H1>
        <FORM action="factor.jsp">
        Number:<INPUT type="text" name="number" title="Number:">
        </FORM>
    <%

    }
    else {
%>
        <H1>Factors of numbers up to <%= number %> are:</H1>
        <TABLE cellpadding="0" border="1">
    <%
            for (int i = 2; i < number; i++) {
                boolean isPrime = true;
%>
            <TR><TD>Factors of <%= i %></TD>
    <%
                for (int j = 2; j <= i / 2; j++) {
                    if (((i / j) * j) == i) {
                        isPrime = false;
%>
                    <TD><%= j %></TD>
    <%
                }
            }
            if (isPrime) {
%>
                <TD><FONT color="0xff0000">Prime</FONT></TD>
    <%
            }
%>
        </TR>
    <%
    }
    }
%>
</BODY>

```

</HTML>

当这个 JSP 被调用时，得到的输出是带有单独一个字段的表单，如下所示：



当用户在文本字段中输入数值 26 之后，JSP 将提供如图 26-1 所示的输出。可以看到，对于每个素数，在其中显示了标注 Prime，对于其余的数，则显示了它们的整数因子。

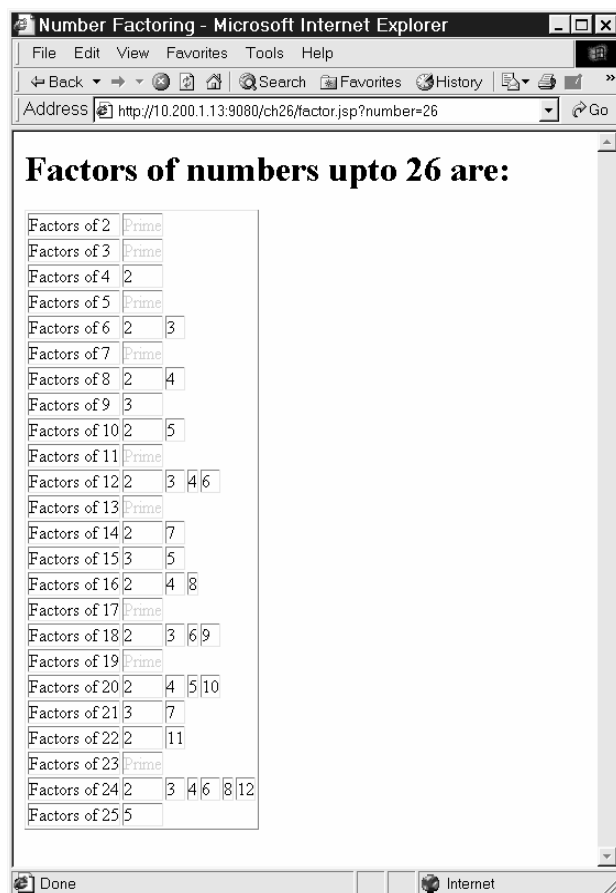


图 26-1 素数 JSP 的结果 Web 页面

26.2 生物学序列对齐 JSP

下一个 JSP 示例来自生物信息学领域。生物信息学 (Bioinformatics) 是一门学科, 它组合了生物学、计算机科学与数学, 目标是分析近年来收集的大量生物学数据。

这个例子主要关注蛋白质。蛋白质是聚合物。换句话说, 蛋白质是由称为氨基酸的更小块构成的。表示蛋白质的标准方法就是使用一个字母序列, 每个字母代表一个氨基酸。

考虑一个数据库表, 在其中存储了蛋白质的集合。这个表的列如下所示:

- ID: 蛋白质的整数 ID 编码, 它是本数据库中内部的表示。
- LEN: 序列的长度。
- SEQ: 序列 (长度为 LEN 的字符串)。
- NAME: 蛋白质的全名。
- IDINSOURCE: 蛋白质在源数据库 (也就是, 将要从其中提取信息的数据库) 中的代码名。

本例所使用的源数据库称为 SWISSPROT, 可以在 <http://www.expasy.ch/sprot> 获取。

本节中的第一个 JSP 使用 JDBC 从数据库表获取信息, 并且以 HTML 格式显示这些信息。当这个页面被调用时, 它将输出一个表单, 用户可以在该表单中填写蛋白质 ID。当这个表单被提交之后, 相同的 JSP 被再次调用, 并且在调用时使用将要显示的蛋白质作参数, 然后, 执行数据库搜索以获取数据。

```
<%@ page import="java.sql.*" %>
<%@ page import="java.net.URL" %>
<%@ page import="java.io.*" %>

<HTML>
<HEAD><TITLE>Protein Data</TITLE></HEAD>
<BODY>
<%
    Connection con;
    java.sql.Statement state = null;
    String param = request.getParameter("proteinid");
    if (param == null) {
%>
        <H1>Enter Protein ID:</H1>
        <FORM action="protein.jsp">
            ID:<INPUT type="text" name="proteinid">
            <BR>
            <INPUT type="submit">
        </FORM>
```

```
<%
    }
    else {
        int number;

        try {
            number = new Integer(param).intValue();
        }
        catch (Exception ex) {
            number = 0;
        }

        if (number <= 0) {
%>
    <H1>Enter Protein ID:</H1>
    <FORM action="protein.jsp">
    ID:<INPUT type="text" name="proteinid">
    <BR>
    <INPUT type="submit">
    </FORM>
<%
    }
    else {
        int protein=number;

        try {
            String url = "jdbc:oracle:thin@dbhost:1521:proteindb";

            // Load Oracle driver
            Class.forName("oracle.jdbc.driver.OracleDriver");

            // Perform DB connection
            con = DriverManager.getConnection(url,"protuser","");

            // Create a statement
            state = con.createStatement();
        }
        catch (SQLException ex) {
            return;
        }
        ResultSet rs = state.executeQuery(
            "select * from proteins where id =" + number + ";")
        if (!rs.next() ) {
%>
```

```

<FONT color="#FF2020">Error: this protein does not exist</FONT>
<%
    }
    else {
        String protIDinSource = rs.getString("idinsource");
        String protName = rs.getString("name");
        int protLen = rs.getInt("len");
        String sequence = rs.getString("seq");
%>

<BODY bgcolor="# E9F800" topmargin="0" leftmargin="8"
    rightmargin="8" marginheight="0" marginwidth="8">
<BR>
<TABLE cellpadding="0" cellspacing="0" width="100%"
    border="0">
    <TR><TD bgcolor="#418A76">
        <TABLE cellpadding="4" cellspacing="1"
            width="100%" border="0">
            <TR bgcolor="#DCF0E4">
                <TD colspan="2">
                    <B></H2>Protein ID <%= protein%></H2>
                    </B></TD>
                </TR>
                <TR bgcolor="# E9F800">
                    <TD><B>Swissprot ID</B></TD>
                    <TD><%= protIDinSource%></TD>
                </TR>
                <TR>
                <TR bgcolor="# E9F800">
                    <TD><B>Name</B></TD>
                    <TD><%= protName%></TD>
                </TR>
                <TR bgcolor="#E9F800">
                    <TD><B>Length in amino acids</B></TD>
                    <TD><%= protLen%></TD>
                </TR>
            </TABLE>
        </TD></TR>
    </TABLE>
<%
    }
}
%>
</BODY>

```



```

import java.util.*;

class Blastd
{
    private static final int daemonPort = 4004;
    private static ServerSocket MainSocket = null;
    static void main(String args[]) {
        try {
            MainSocket = new ServerSocket(daemonPort);
            while (true)
            {
                Socket ClientSocket = MainSocket.accept();
                Blaster thd = new Blaster(ClientSocket);
                thd.start();
            }

        }

        catch (Exception e)
        {}

        finally
        {
            try
            {
                MainSocket.close();
            }
            catch (Exception exc)
            {
            }
        }
    }
}

::::::::::::
Blaster.java
::::::::::::
import java.net.*;
import java.io.*;
import java.util.*;

public class Blaster extends Thread
{
    Socket socket = null;          // Socket to client

```

```
public Blaster(Socket clientSocket)
{
    socket = clientSocket;
}

public void run()
{
    BufferedReader input = null;
    PrintWriter output = null;

    try {
        input = new BufferedReader(new InputStreamReader
                                   (socket.getInputStream()));
        output = new PrintWriter(socket.getOutputStream(), true);
        String seq1 = input.readLine();
        String seq2 = input.readLine();

        long pid = System.currentTimeMillis();
        String fn1 = "/tmp/s1_" + pid;
        String fn2 = "/tmp/s2_" + pid;
        String fn3 = "/tmp/oo_" + pid;
        PrintWriter pw = new PrintWriter(new FileOutputStream
                                           (fn1));

        pw.println(seq1);
        pw.close();
        pw = new PrintWriter(new FileOutputStream (fn2));
        pw.println(seq2);
        pw.close();
        String cmd = "./bl2seq -i " + fn1 + " -j " + fn2 +
                     " -p blastp -T -o " + fn3;
        System.out.println(cmd);
        Process p = Runtime.getRuntime().exec(cmd);
        p.waitFor();

        BufferedReader fileReader = new BufferedReader(new
                                                         InputStreamReader(new FileInputStream(fn3)));
        while (true) {
            String res = fileReader.readLine();
            if (res == null)
                break;
            output.println(res);
        }
    }
```

```
        output.close();
        socket.close();

    }
    catch (Exception ex) {
        System.out.println("Exception " +ex );
    }
}
}
```

在编译并运行这个程序之后，将实现下列 Java 类，这个类用来调用此程序。这个类使用 Java 套接字与该程序连接并将序列传递给这个程序。实质上，这里所做的工作就是实现一个简单的 RPC（Remote Procedure Call，远程过程调用）。

```
import java.net.*;
import java.io.*;

public class Blast {
    public String doIt(String seq1, String seq2) {
        String ret = "";
        int port = 4004;
        String server = "proteinhost";
        Socket socket = null;
        String lineToBeSent;
        BufferedReader input;
        PrintWriter output;
        int ERROR = 1;

        // connect to server
        try {
            socket = new Socket(server, port);
        }
        catch (UnknownHostException e) {
            System.err.println(e);
        }
        catch (IOException e) {
            System.err.println(e);
        }
        try {
            output = new PrintWriter(socket.getOutputStream(),true);
            input = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
```

```
        output.println(seq1);
        output.println(seq2);
        output.flush();

        boolean done = false;
        while (!done) {
            try {
                String instr = input.readLine();
                if (instr == null) {
                    done=true;
                    break;
                }
                System.out.println("In = "+instr);
                ret = ret + instr;
            }
            catch (Exception ex)
            {
                ret = ret + "Exception" + ex;
                done = true;
            }
        }

    }
    catch (IOException e) {
        System.out.println(e);
    }
    try {
        socket.close();
    }
    catch (IOException e) {
        System.out.println(e);
    }
    return ret;
}
}
```

最后，将分析这个例子的 JSP，它使用两个蛋白质 ID 作为参数，并输出关于它们的序列对齐。这个 JSP 从数据库中检索蛋白质序列，然后调用 Blast 类，而这个类又连接到前面提到的小 Java 程序上。

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



```

<HEAD>
<TITLE>Sequence Alignment</TITLE>
</HEAD>
<%
    String prot1 = request.getParameter("protein1").toString();
    String prot2 = request.getParameter("protein2").toString();
    if (prot1==null)
        prot1="";
    if (prot2==null)
        prot2="";
    int protein1=0;
    int protein2=0;

    try {
        protein1 = new Integer(prot1).intValue();
        protein2 = new Integer(prot2).intValue();
    }
    catch (Exception ex) {
        // Handle Error...
    }
%>
<BODY bgcolor="#E8F7EE" topmargin="0" leftmargin="7" rightmargin="7"
marginheight="0" marginwidth="7">
<FONT color="#6FB5A2"><H3>Sequence Alignment for Proteins
<%= protein1%>, <%=protein2%></H3></FONT>
<%
    int protLen1 = 0;
    String sequence1 = null;
    int protLen2 = 0;
    String sequence2=null;
    Connection con ;
    java.sql.Statement state ;
    try {
        String url = "jdbc:oracle:thin@dbhost:1521:proteindb";
        Class.forName("oracle.jdbc.driver.OracleDriver");
        con = DriverManager.getConnection(url,"protuser","");
        state = con.createStatement();
        ResultSet rs = state.executeQuery(
            "select * from proteins where id =" + protein1 + ";" ) ;
        if (!rs.next() ) {
%>
<FONT color="#FF2020">Error: this protein does not exist</FONT>
<%
        }

```

```
        else {
            protLen1 = rs.getInt("len");
            sequence1 = rs.getString("seq");
        }
        rs = state.executeQuery(
            "select * from proteins where id =" + protein2 + ";" );
        if (!rs.next() ) {
%>
<FONT color="#FF2020">Error: this protein does not exit</FONT>
<%
        }
        else {
            protLen2 = rs.getInt("len");
            sequence2 = rs.getString("seq");
        }
        state.close();
        con.close();
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}
%>
<%
    Blast b = new Blast();
    String str = b.doIt(sequence1, sequence2);
%>
<FONT face='courier'>
<%=str%>
</FONT>
</BODY>
</HTML>
```

图 26-2 显示了这个 JSP 的输出样例。

26.3 读取邮件的 JSP

在最后一个例子中，将重新分析第 22 章讨论的读取邮件的 servlet 示例。这里所提供的 Web 应用程序的功能与第 22 章中那个 servlet 非常相似，但现在这个例子的实现是基于 JSP 的。其不同之处是，这个例子的实现中使用会话对象来维护状态（应该记得，在第 23 章详细讨论了 HTTP 会话对象）。通过与第 22 章显示的那个 servlet 比较，可以发现 JSP 模型的诱人之处：可以在保持 servlet 模型所有功能的情况下，避免使用那些长而乏味的 HTML 标记的

println 调用。

这个 Web 应用程序的第一个页面是登录页面，该页面将显示一个简单的表单。

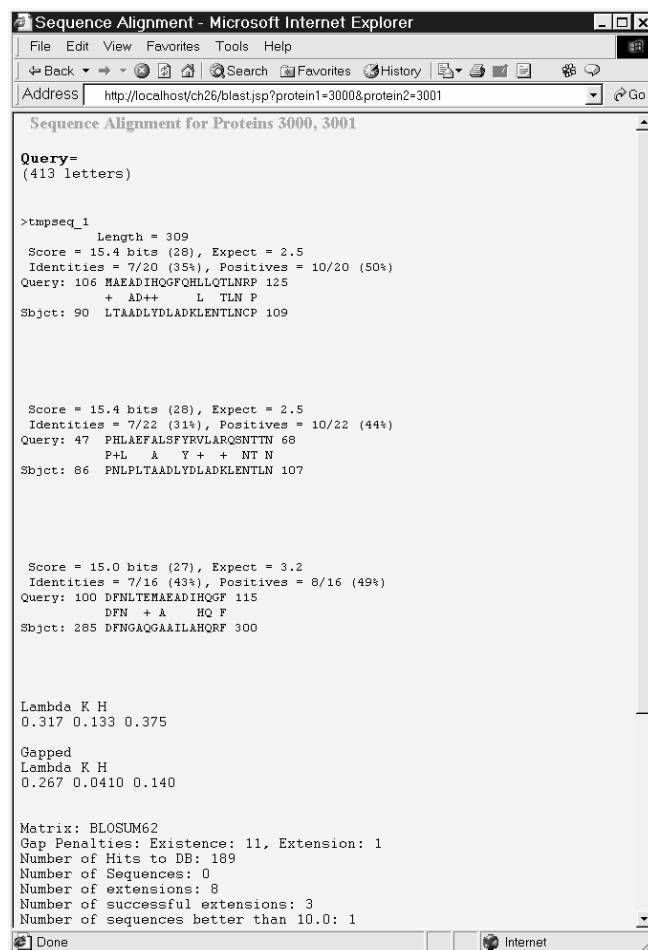


图 26-2 蛋白质序列对齐的 blast JSP 输出

```
<HTML>
<HEAD>
<TITLE>Mail Reader JSP
</TITLE>
</HEAD>
<BODY>
<H1>Welcome to the JSP-based Mail Client</H1>
<FORM ACTION="getMail.jsp" METHOD="POST">
<B>Username: </B>
```

```
<INPUT TYPE='TEXT' NAME='username'><BR>
<B>POP3 Server: </B>
<INPUT TYPE='TEXT' NAME='pop3'><BR>
<B>Password: </B><INPUT TYPE='TEXT'
NAME='password'><BR>
<INPUT TYPE='SUBMIT'><INPUT TYPE='RESET'>
</FORM>
</BODY>
</HTML>
```

这个页面将把用户指向下列代码所显示的 `getMail.jsp` 页面。这个新页面使用登录参数连接到邮件服务器并且将连接对象放置到 servlet HTTP 会话中。然后,显示一个在其中包含有邮件消息列表的 HTML 表。

```
<% page import="java.util.*" %>
<% page import="java.io.*" %>
<% page import="javax.mail.*" %>
<% page session="true" %>
<%
    Folder folder = null;
    Message[] messages = null;
    Session mailsession = null;
    Store store = null;
    String userName = null;

    if (session.getAttribute("username") == null) {
        userName = req.getParameter("username");
        String pop3 = req.getParameter("pop3");
        String password = req.getParameter("password");
        mailsession = Session.getInstance(System.
getProperties()null);
        store = mailsession.getStore("pop3");
        store.connect(pop3, userName, password);
        folder = store.getFolder("INBOX");
        folder.open(Folder.READ_ONLY);
        messages = folder.getMessages();
        session.setAttribute("username", userName);
        session.setAttribute("folder", folder);
        session.setAttribute("messages", messages);
        session.setAttribute("store", store);
        session.setAttribute("session", mailsession);
    }
    else {
        username = (String) session.getAttribute("username");
```

```

        folder = (Folder) session.getAttribute("folder");
        messages = (Message []) session.getAttribute("messages");
        mailsession = (Session) session.getAttribute("session");
        store = (Store) session.getAttribute("store");
    }

    %>
    <HTML>
    <HEAD>
    <TITLE>
    Welcome <%=username %>! Your Mail
    </TITLE>
    </HEAD>
    <BODY>
    <TABLE BORDER="1">
    <TR>
    <TH>From</TH>
    <TH>Date</TH>
    <TH>Subject</TH>
    <%
        for (int i = 0; i < messages.length; i++) {
    %>
    <TR><TD>
    <%= messages[i].getFrom()[0] %>
    </TD>
    <TD><%= messages[i].getSentDate() %>
    </TD>
    <TD><A HREF="getDetail.jsp?msgnum=<%=i%>">
    <%= messages[i].getSubject() %>
    </A>
    </TD>
    </TR>
    <%
        } %>
    </TABLE><P><A HREF="doLogout.jsp">
    Log-Off</A>
    </P>
    </BODY>
    </HTML>

```

下一个页面包括到两个其他页面的链接。其中第一个页面就是 doLogout.jsp 页面，该页面比较直观。它显示一条消息，作废此会话，并且关闭 JavaMail 文件夹与 store 对象。

```

<% page import="java.util.*" %>
<% page import="java.io.*" %>
<% page import="javax.mail.*" %>

```

```
<% page session="true" %>
<%
    Folder folder = null;
    Message[] messages = null;
    Session mailsession = null;
    Store store = null;
    String userName = null;

    username = (String) session.getAttribute("username");
    folder = (Folder) session.getAttribute("folder");
    messages = (Message []) session.getAttribute("messages");
    mailsession = (Session) session.getAttribute("session");
    store = (Store) session.getAttribute("store");
%>
<HTML>
<HEAD>
<TITLE>
Mail Reader JSP Logout
</TITLE>
</HEAD>
<BODY>
<H1>
Thank you for using the JSP mail reader
</H1>
</BODY>
</HTML>
<%
    try {
        folder.close(false);
        store.close();
        session.invalidate();
    }
    catch (Exception ex)
    {
    }
%>
```

最后一个页面是 `getDetail.jsp`，它获取并显示某个电子邮件消息的实际内容。

```
<% page import="java.util.*" %>
<% page import="java.io.*" %>
<% page import="javax.mail.*" %>
<% page session="true" %>
<%
```

```
Folder folder = null;
Message[] messages = null;
Session mailsession = null;
Store store = null;
String userName = null;
username = (String) session.getAttribute("username");
folder = (Folder) session.getAttribute("folder");
messages = (Message []) session.getAttribute("messages");
mailsession = (Session) session.getAttribute("session");
store = (Store) session.getAttribute("store");
int num = Integer.parseInt(req.getParameter("msgnum"));
String s;
Message msg = messages[num];

%>
<HTML>
<HEAD>
<TITLE>
Reading Message Number <%= num %>
</TITLE>
</HEAD>
<BODY>
</H1><Reading Message: <%= msg.getSubject() %>
<B>From:</B>
<%= msg.getFrom()[0] %>
<B>To:</B>
<%= msg.getRecipients(Message.RecipientType.TO)[0] %>
<B>Date:</B>
<%= msg.getSentDate() %>
<P>
<%
        BufferedReader rd = new BufferedReader(new
            InputStreamReader(msg.getInputStream()));
        while ((s = rd.readLine()) != null)
            out.println(s);
%>
</P>
</BODY>
</HTML>
```

26.4 小 结

本章分析了几个简单的 JSP 示例。可以看出，使用 JSP 并没有什么困难的地方，这也是 JSP

的主要优点之一；JSP 经常被用来构建耐用的应用程序。可以说，简单编程范型与复杂应用程序之间界限已经越来越模糊；随着应用程序复杂度的提高，编程的复杂性也在不断增加并且简单的模型已经难以胜任此工作。实际上，JSP 也面临这些问题。但幸运的是，JSP 模型正在不断发展；当所使用的应用程序已经难以在此模型中适合时，JSP 也就相应地改进了，并且抽象出了那些复杂性。

这方面的最新发展趋势是，使用自定义标记库封装并抽象 JSP 上使用的复杂行为。第 27 章将讨论自定义标记库。

第 27 章 自定义标记库

Servlet 与 JSP 是 WebSphere 开发平台上每个 Web 应用程序开发人员使用的主要工具。实际上, JSP 与 Servlet 是一个东西并且是相同的, 因为每个 JSP 都将被 JSP 引擎转化为一个 Servlet。JSP 生命周期与 Servlet 生命周期大部分都是相同的, 用户可以自由地决定在应用程序中使用 Servlet 还是使用 JSP, 这取决于应用程序的侧重点和需求。

实际上, 是使用 JSP 还是使用 Servlet (或两者都用) 几乎总是取决于开发人员在开发组织中的角色。如果正在构建后端功能, 那么可以使用 Servlet 向外部提供所开发的功能。如果工作于应用程序的表示层, 主要是构建访问某些后端数据并将这些数据提供给浏览器的动态 Web 屏幕, 那么可能使用的是 JSP。从角色方面看, 后端编程者通常编写 Servlet, 而页面设计者总是使用 JSP。用于编辑 JSP 的工具比较成熟, 并且遵守 Web 页面开发工具的相同范型。相反, 除了作为 Java IDE 的一部分而提供的模板与调试器之外, Servlet 并没有包括任何开发工具。

虽然 JSP 是用于构建用户界面的首选方法, 但它们也可以变得很复杂。本章的前面部分将分析 JSP 固有的一些问题与复杂性。

在本章后面, 将介绍自定义标记库。业界正在越来越多地使用这些库, 在 WebSphere 中也一样。自定义标记库这一技术可能会引导 JSP 使用的革命, 并且导致构建 JSP 更加简单。使用标记库还将更好地区分 Web 应用程序开发人员的角色。对标记库的讨论足以占据一本书的篇幅; 但本章并不涉及自定义标记库的所有选项、生命周期、API 集合这些问题, 本章将介绍自定义标记库, 并且描述在 WebSphere 部署中如何使用标记库。

27.1 JSP 复杂性问题

正如第 7 章分析的那样, Servlet 与 JSP 的历史是不断发展的。其主要发展历程是: 当 Servlet 第一次出现, 作为编写 CGI 脚本的更好替代方法以来, 它就受到极大的欢迎。后来, 随着大量 HTML 被嵌入作为 Java print 命令中的字符串, 并且这种局面变得越来越难以控制, 就导致 Servlet 变得非常复杂。许多工具都不能使用, 因为作为字符串, 元素失去了所有的语义和结构。然后, 出现了 JSP 并颠倒了这一角色, 使页面被构建作为“常规的”HTML 页面, 并且在这些 HTML 页面中嵌入一些 scriptlet, 用来提取动态数据或执行某些服务器方处理。由于 JSP 在 Servlet 的基础上进一步加强了某些功能, 使得 JSP 成为最好的页面编写工具。但不幸的是, 随着时间的推移, JSP 被广泛用于复杂业务应用程序, 这样又出现了下一个“复杂性曲线”——因为编程结构变得非常庞大。在许多业务应用程序中, 页面都非常

巨大并难以阅读。更为严重的问题是，这些页面变成了 HTML 与 Java 的混合，这就意味着在页面设计者（HTML 开发人员）与 Java 开发人员之间已经不存在明确的区别。这样，构建复杂的 Web 用户界面现在还需要 Java 开发人员的参与，而这种代价是很高的。

27.1.1 封装与抽象

计算机科学中几乎每个问题最终都可以被归结为缺乏封装（lack of encapsulation）或缺乏抽象（lack of abstraction）。事实证明，JSP 所面临的问题可以说就是由于在封装与抽象方面都存在不足。当人们首次开始编写 JSP 时，常常会被这个模型的简单所折服。他们通常简单地构建一个 Web 页面，然后就开始使用 Java 代码填写动态内容。这种方法对于简单使用还可以，但开发人员并没有总是将他们的页面当作编程结构来看待。因此，他们常常忽略了关于程序模块性与封装的问题，并且最终导致代码大而又难以管理。应该记得，Web 页面自己被编译成一个 servlet，并且在这个过程中生成服务方法。

理解 JSP 模型的最佳方法是看一个复杂的页面。图 27-1 显示了一个比较简单的 Web 屏幕，它基于一个工作流引擎所管理的业务规则集合，根据一系列动态行为来控制使用该屏幕可完成的工作。在这里并不列出这个 JSP 的内容，因为其代码较长（它的 HTML 与 Java 部分的代码总和超过了 2000 行）。更麻烦的问题是，在生成的 servlet 中该服务方法的大小也很大，有超过 1100 行的代码！

虽然仅仅考虑所生成方法的大小具有一定的误导（毕竟，这些代码是由代码生成器产生的），但这确实能够帮助读者理解 JSP 所面临的复杂性问题有多么严重。这种对抽象与模块化的缺乏，正是大多数 Java 开发人员力图解决的问题。

实际上，关于这个问题的解决方案对 HTML 与 Java 代码都是简单而又自然的。HTML 代码中的重复部分可以被封装成一个单独的 JSP，并且嵌入到其他页面中。封装不仅减少了代码大小，而且支持可重用性并加强了可维护性，因为修改只要在一个地方进行，然后被复制到其他地方。不仅如此，开发人员也不需要再在 HTML 方做很多的工作，因为 HTML 常常可以利用编辑工具生成。

对于动态部分（Java 代码），其解决方案就更加简单而自然。由于 Java 是一种面向对象的语言，使 JSP 更具模块化的方法就是，封装那些通常被嵌入到 JSP 中的 Java 代码作为 Java 对象的方法。然后，这些类与方法就能够在多个 JSP 中使用，从而大幅度减少代码量并且使其维护过程流水线化。

Report Activities - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://ron2k.itsoftware.co.il/servlet/RequestHandler?processId=1260000001144&expectedStepName=WorkStationISP&user=RonG&newProcess=

Report Activities

RMA 1525 Job 1525-1 Status Received

Action 2 Repair Inspection

Product BASE STA ASSY 3850 Serial #

Service Symptom Courtesy Visit Repeat Repair No

Problem

Notes

Faults

Fault Cause Remedy

Choose a Fault Choose a Cause Choose a Remedy View/Add Faults

Labor

Activity From To Duration

Choose a Labor View/Add Labor

Parts Activities

Activity	Product	Find Part	Qty.	Serial #	Room	Aisle	Bin
Choose an Activity		Look In					
		Look In					
Choose an Activity		Look In					
		Look In					

Complete Action

Choose an Action

Submit Clear View Activities Add New Action

图 27-1 JSP 所生成的 Web 页面示例

27.1.2 附件示例

下面将分析一个例子，以了解在封装之前与封装之后同一个例子代码间的区别。这个例子与文档附件相关。在图 27-1 中，可以在屏幕的右上方看到一个图纸夹图标。如果单击这个图纸夹，那么将出现下列屏幕：

Attachment for 1525

To add an attachment, type in a path or hit the Browse button. Then hit the Add Attachment button to add the attachment.

Browse... Description Add Attachment

Diagnostics manual - IBM_POS.htm Delete

console_printout - console.log Delete

TOC - toc.doc Delete

这个屏幕显示了附加在图 27-1 上的工作的所有附件。每个文档都可以查看，并且可以将一个附件添加到对象（在这里，就是一项工作）中。这种机制是很普遍的，因为用户可能需要将一个附件添加到这个系统所管理的任何对象中。因此，这个机制需要包括在许多不同的 JSP 中。

在开发过程中常常是这样，通过从先前的失败中吸取教训而变得更聪明。因此，当第一次开发了一个附件功能之后，后来就可能会直接将实现该附件功能的代码嵌入到 JSP 中。

在下一次试图开发此功能时，用户就可能会创建一个新类，用来封装附件功能。这样做的结果是，能够为任何实体与任何 JSP 非常容易地添加附件功能。这样就在系统中创建了一个新的抽象，在这个抽象中对附件的实现进行了非常整洁的打包；后来就可以利用这个新抽象在 JSP 中添加功能。从维护的角度看，这也是一条好消息，因为现在能够仅仅修改实现附件功能的 Java 类，而不必对使用该功能的 JSP 代码进行修改。

每个需要嵌入附件功能的 JSP 都需要包括下列代码：

```
...

<%@ import = "com.rts.attachment.*" %>
...
<%

    HttpSession session = request.getSession(true);

    AttachmentInfo ai = new AttachmentInfo();
    Map keysAndValues1 = new HashMap(3);
    keysAndValues1.put("id", 1234);
    ai.setKeysAndValues(keysAndValues1);
    ai.setAccessLevel("1");
    ai.setClassName("ServiceOrder");

    AttachmentCreator ac = new AttachmentCreator(ai);

    session.putValue(
        PortalConstants.getSessionVariable(
            "RTS_Attachment_Creator"), ac);

    // *** ATTACHMENT END ***
    %>
...
<!-- Start Attachment Link -->
<A class="WinletLinkFont" href="Attachment.jsp">
    <IMG BORDER="0" ALT="Attachment" SRC=attachment.gif/ >
</A>
<!-- End Attachment Link -->
```

这里主要关心的是此代码中的两个部分。第一个就是 attachment creator 对象的创建。attachment creator 获取一个或多个附件信息对象。然后，就可以将附件添加到一个实体中，或者修改属于某个实体的附件。这些实体需要被附件信息对象所引用。

附件 JSP 使用 attachment creator 对象显示附件并添加或删除附件，如下所示：

```
...
<%
    HttpSession session = getSession(request);
    noCacheResponse(response);

    String root = Attachment.getRootURLLocation();

    AttachmentCreator attachCreator = (AttachmentCreator)
        session.getValue("RTS_Attachment_Creator");

    String errorMessage = null;
    String prompt = "";
    String title = "";
    String objectInd = "";
    String description = (String) request.getAttribute
        ("description");
    AttachmentInfo[] attachInfo = null;

    if (attachCreator == null)
    {
        errorMessage = "Error";
    }
    else
    {
        attachInfo = attachCreator.getInfo();
        errorMessage = (String) request.getAttribute
            ("errorMessage");

        if (attachCreator.hasSingleObject())
        {
            title = " for ";
            objectInd = "0";
        }
        else
        {
            prompt = " select the object,";
        }
    }
}
```

```
    }  
    ...  
%>  
...  
<!-- Begin title -->  
<tr>  
  <td>  
    <table width="100%" border="0" cellspacing="0"  
      cellpadding="0" class="WidgetsBackground">  
      <tr>  
        <td>  
          <table>  
            <tr>  
              <td class="WindowHeading">  
                Attachment  
              </td>  
              <td class="WindowHeadingDynamicData">  
                <%= title + attachCreator.getTitle() %>  
              </td>  
            </tr>  
          </table>  
        </td>  
      </tr>  
    </table>  
  </td>  
<!-- End title -->  
...  

```

可以看到，这确实是一个关于附件机制的良好封装。

27.2 标 记 库

前一节中描述的解决方案能够很好地工作。它封装了行为并创建一个新抽象。但问题是，这个新抽象是否仍然是 Java 抽象呢？虽然这个例子在 Java 编程者看来非常简单，但它对于那些仍然希望在 HTML（或 DHTML）中工作的页面设计者可能毫无用处。毕竟，页面是属于页面设计者的工作，而不是属于 Java 编程者的工作，但如果需要利用前面一节中的代码，情况就不再如此。

实际上，真正需要创建的应该是一个封装了所需的行为并且能够被页面设计者所使用的抽象。为此，就出现了自定义标记库。HTML 范型实际上就是所有标记的范型。所使用的标记通常是由 HTML 规范定义的。自定义标记库允许定义新标记，并且将 Java 类与实现行为相关联。然后，这些新标记就可以被页面构建者用来调用封装并抽象在每个标记中的功能。正如页面设计者能够使用 HTML TABLE 标记来构建一个表，并且使用 ANCHOR 标记（带

有一个 HREF 属性) 来构建一条链接一样, 使用 JSP 编辑器的页面设计者需要能够添加附件按钮, 用来激活附件的所有必备功能。通过使用标记库, Java 开发人员能够以一种简单的“像 HTML 一样”的方法构建提供功能的 Java 类。

在附件工程中, 这些类允许页面设计者通过使用一个简单的元素来集成一条附件链接 (包括附件的图像以及其后的处理程序), 如下所示:

```
...
< vn_attachment:attach_to_single_object
    entity="ServiceOrder"
    key="id"
    value="1234"
/>
```

自定义标记库允许利用自己的代码扩展基本 JSP 标记集合, 从而提供了可供页面设计者使用的附加的抽象。标记库使 JSP 开发人员达到了一种新的生产率、可重用性以及可维护性层次。

27.2.1 在 JSP 中使用标记

标记库是 JSP 1.1 规范的一部分, 并且任何支持 JSP 的应用程序服务器的实现都需要以相同方式支持这些标记库 (至少在使用模型上应该如此)。这样就提供了一个标准化约束, 用来确保许多标记库可以被创建与重用。

为了使用标记库中的标记, 需要做下列工作:

1. 从 JSP 中引用标记库。
2. 在 JSP 中使用标记。

为了使 JSP 能够在运行时正确工作, 需要在 WebSphere 服务器上部署这个标记库。为此需要进行下列工作:

1. 构建一个标记库描述符 (TLD) 文件。
2. 在 Web 应用程序上下文中部署这个 TLD 文件。
3. 确保实现自定义标记的 Java 类对 JVM 可访问 (即确保它们包含在 classpath 中)。

下面给出了一个使用自定义附件 (attachment) 标记的例子:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HEAD>
...
<%@ taglib
    uri="http://www.viryanet.com/taglibs/attachment-taglib"
    prefix=vn_attachment %>
```

```

</HEAD>
...
<vn_attachment:attach_to_single_object
    entity="ServiceOrder"
    key="id"
    value="1234"
/>
...

```

从这个例子中可以看出，只有在声明了 taglib 的位置之后，在这个库中定义的标记才可以被引用。同样可以看出，使用标记库是很简单而且很自然的。但在服务器能够与这个标记的代码相关联之前，需要进行一点设置工作（标记功能的编码实现问题将在下一节“实现自定义标记”讨论，关于代码部署方面的详细讨论将在 27.2.3 节“在 WebSphere 中部署标记库”给出）。

作为部署工作的一部分，首先要将这些类放置到服务器的 classpath 中（或将这些类打包成 Web 模块）。然后，就需要构建一个 TLD 文件，这个文件中存放了关于标记库的元数据，服务器使用这个文件进行有效性检查（当然还有其他用处）。TLD 描述了库中每个标记的结构、属性等信息。例如，附件的 taglib TLD 如下所示：

```

<?xml version="1.0" encoding="iso-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>vn_attachment</shortname>
    <uri>http://www.viryanet.com/taglibs/attachment-taglib</uri>
    <info>
        A tag library that defines the tag elements for the attachment framework
    </info>

    <tag>
        <name> attach_to_single_object</name>

<tagclass>com.rts.attachments.SingleObjectAttachment</tagclass>
    <bodycontent>empty</bodycontent>
    <info>
        Encapsulates attachments associated with a single entity.
        The entity is referenced by a key and the value of the key.
    </info>

```



```

    <attribute>
      <name>entity</name>
      <required>true</required>
    </attribute>

    <attribute>
      <name>key</name>
      <required>true</required>
    </attribute>

    <attribute>
      <name>value</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>

  </tag>

  ...

</taglib>

```

在这个 TLD 中，惟一不具有自解释性的元素是 `rtexprvalue` 元素。把这个元素的默认值设置为 `false`，它在 `attach_to_single_object` 标记中，实体与关键字都将这个值设置为 `false`。当这个值被设置为 `true` 时，就意味着这个属性在运行时所使用的值是一个需要引擎计算的运行时表达式。在这个例子中，关键字的值就是工作编号，这个值显然不是固定的；传递给用户的每个页面都可能有不同的工作，并且用户需要为每个工作获取相应的附件，而不是为一个具有固定值的工作获取相应的附件。

对于在 JSP 中该标记的使用，相应的 JSP 代码如下所示：

```

...
<vn_attachment:attach_to_single_object
  entity="ServiceOrder"
  key="id"
  value="<%= jobBean.getId() %>"
/>

```

在准备好 TLD 之后，就需要在 Web 应用程序上下文中部署它。可以从管理控制台完成这个工作，或者确保这个文件被 Web 描述符文件所引用。如果使用 Web 描述符文件（`web.xml`），就需要添加一个 `taglib` 元素，如下所示：

```

<taglib>
  <taglib-uri> http://

```

```
www.viryanet.com/taglibs/attachment-taglib </taglib-uri>
<taglib-location>WEB-INF/
attachment-taglib.tld</taglib-location>
</taglib>
```

27.2.2 实现自定义标记

现在读者可能能够猜测到自定义标记是如何被 JSP 引擎所处理的。标记就是被 JSP 引擎所处理的元素。JSP 引擎使用 TLD 来查找实现标记所代表行为的类。这个类需要实现一个接口，这个问题将在后面讨论。在引擎找到了与这个标记相匹配的类之后，将调用这个类中的方法。此方法通常将一些数据返回给页面，因此页面的上下文必须对实现这个标记的类可用。然后，标记的实现将基于 taglib API 所定义的某种协议，告知 JSP 引擎如何继续处理这个页面。

一个没有体的简单标记

下面是一个先前所显示的附件（attachment）标记的基本实现：

```
package com.rts.attachments;

import java.io.IOException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.http.*;
import com.rts.infrastructure.EntityMapper;

public class SingleObjectAttachment
    extends TagSupport {

    String key;
    String entity;
    String value;

    public void setKey(String key) {
        this.key = key;
    }

    public void setEntity(String entity) {
        this.entity = entity;
    }
}
```

```
public void setValue(String value) {
    this.value = value;
}

public int doStartTag()
    throws JspException
{
    try {

        AttachmentInfo ai = new AttachmentInfo();
        Map keysAndValues1 = new HashMap(3);
        keysAndValues1.put("id", key);
        ai.setKeysAndValues(keysAndValues1);
        ai.setAccessLevel("1");
        ai.setClassName(EntityMapper.mapToClass(entity));

        AttachmentCreator ac = new AttachmentCreator(ai);

        pageContext.setAttribute("RTS_Attachment_Creator", ac, Page-
Context.SESSION_SCOPE);

        pageContext.getOut().println("<A " +
"class=\"WinletLinkFont\" href=\"Attachment.jsp\">");
        pageContext.getOut().println("<IMG BORDER=\"0\" " +
"ALT=\"Attachment\" SRC=attachment.gif/>");
        pageContext.getOut().println("</A>");

    } catch(Throwable ex) {
        throw new JspTagException("Exception: " + ex);
    }
    return SKIP_BODY;
}
}
```

这个类非常直观。它从 `TagSupport` 继承而来，这样就继承了许多作为 `taglib` 框架一部分的方法，并且遵守 `taglib` API 集合。这个类也提供了对许多所需对象的访问。

下一个问题就是属性。`entity` 属性用来帮助了解对象的类型，以及实例化该对象所需的關鍵字与值的属性。这些属性需要被传递给这个标记类。将值传递到标记类的工作是使用 `JavaBean` 定义的约定进行的。出现在标记中的每个属性都应该有一个 `set` 方法。在调用 `doStartTag` 方法之前，要调用这些 `set` 方法，从而使得在处理所调用的方法时，这些属性的值被设置到实例变量中。

然后,doStartTag 方法被调用。这样,在会话上下文中,就产生并保存了一个 attachment creator 对象,这个对象将被另一个 JSP 所使用,用来显示实际的附件。最后,发送给这个页面的输出被创建成为超链接。这个方法最后将返回 SKIP_BODY,因为在上面给出的例子中没有标记体。

带有体的标记

在前面,实现了一个从 TagSupport 类继承而来的类。TagSupport 类是规范中 2 个标记基类之一,另一个是 BodyTagSupport 类。TagSupport 类实现了 Tag 接口,BodyTagSupport 类则实现了 BodyTag 接口。两个类的主要区别在于 BodyTagSupport 关注于元素的体,TagSupport 则不是这样。因此,BodyTagSupport 影响到自定义标记的生命周期、功能以及复杂性。

一般来说,调用一个单独的后端操作的标记通常没有体。它有一个标记属性的集合,将作为参数传递给后端操作。在各种书籍中常常见到的关于标记库的例子就是 send_email。这个标记允许页面设计者指定应该基于某些动态属性(这些动态属性从 JSP 的相同动态对象中提取出来)来发送电子邮件。

标记的第二个主要用途就是在页面被处理时为应用程序提供标记功能。在构建 JSP 之后,HTML 元素向浏览器提供伪指令,用来指示如何向用户表现信息。这些信息需要动态地创建;因此,需要使用 JSP。许多页面有时可能需要做较多的动态内容生成工作,并且,对一个页面体的处理常常依赖于动态属性。这些属性可以(也应该)被放置在自定义标记的属性中,从而有效地确保在整个页面中避免使用 scriptlet。

一个带有体的标记通常从标记的属性中提取某些值(或者从前面定义的上下文中提取这些值),并且使用这些值来读取标记体并生成内容。这样的例子如:translations,在这里,一个标记的属性用来替换体中的某些单词;stripping,在这里,用户的标识可用来帮助屏蔽那些用户无权看到的信息;还有许多其他非常有用的功能例子。

体标记可以被分为两组:第一种体标记可以在标记开始时执行一个动作,并且在页面上下文中设置某些变量,然后,这些变量被标记体中的其他标记所使用。这些标记可以控制执行流程,但并不实际地使用体内容;体内容使用标记已经准备好的信息。第二种体标记可以读取体内容,对这些内容进行某种处理,然后将修改后的内容作为新体内容写回。

关于体标记的一个典型例子就是 iteration 标记(实际上,在 JSP 1.2 中,这个标记是内核规范的一部分)。许多 JSP 都遵守一种模式,在这种模式下,它们需要显示一个对象集合。图 27-2 给出了一个 HTML 页面,这个页面显示了一个关于服务调用的集合;对于每个调用,还提供了诸如状态、用户、位置以及使用次数等信息。

生成图 27-2 所示内容的 JSP 将创建一个集合对象,并且在这个集合对象中进行迭代,然后写出每个调用所需的数据。例如,这个 JSP 可能如下所示:

Call ID	Call Type	Action	Symptom	Customer	Site	Agreed ETA	Start Date	Respond By	Fix by	Assigned To	Priority	Seq	Status	Update
3848	Customer initiated	1	Cracked Unit	112500 Com Corp	SITE site 2	26 Nov 2001 11:00	26 Nov 2001 00:35	26 Nov 2001 10:00	27 Nov 2001 09:00	112001 Lorenzo Kinney	90	0	Completed	Update
3838	Customer initiated	1	Cracked Unit	112500 Com Corp	SITE site 2	26 Nov 2001 03:00	25 Nov 2001 19:20	26 Nov 2001 10:00	27 Nov 2001 09:00	112001 Lorenzo Kinney	90	0	Completed	Update
3837	Electric Outage	1		U- AC71 Agence Centrale Electrique	US- 71/1 Le Mans	04 Dec 2001 08:53	25 Nov 2001 19:19	26 Nov 2001 10:00	27 Nov 2001 09:00	112001 Lorenzo Kinney	90	0	Completed	Update
3836	Technical Support	1		112500 Com Corp	SITE site 2		25 Nov 2001 19:17	26 Nov 2001 16:00	27 Nov 2001 15:00	112001 Lorenzo Kinney	90	0	Dispatched	Update
3835	Installation	1		112500 Com Corp	SITE site 2		25 Nov 2001 19:16	26 Nov 2001 16:00	27 Nov 2001 15:00	112001 Lorenzo Kinney	90	0	Wait for Ack	Update
3834	Electric Outage	1		U- AC71 Agence Centrale Electrique	US- 71/1 Le Mans	26 Nov 2001 02:31	24 Nov 2001 03:11	26 Nov 2001 10:00	27 Nov 2001 09:00	112001 Lorenzo Kinney	90	0	Completed	Update
3833	Electric Non Outage	1		U- AC71 Agence Centrale Electrique	US- 71/1 Le Mans	24 Nov 2001 09:47	24 Nov 2001 02:47	26 Nov 2001 10:00	27 Nov 2001 09:00	112001 Lorenzo Kinney	90	0	Completed	Update
3832	Gas Leak	1		U- AC71 Agence Centrale Electrique	US- 71/1 Le Mans	20 Nov 2001 15:01	20 Nov 2001 08:00	21 Nov 2001 07:00	22 Nov 2001 06:00	112001 Lorenzo Kinney	90	0	Completed	Update
3831	Gas Leak	1		U- AC71 Agence Centrale Electrique	US- 71/1 Le Mans	20 Nov 2001 14:43	20 Nov 2001 07:43	21 Nov 2001 06:43	22 Nov 2001 05:43	112001 Lorenzo Kinney	90	0	Completed	Update
3830	Gas Leak	1		U- AC71 Agence Centrale Electrique	US- 71/1 Le Mans	20 Nov 2001 14:37	20 Nov 2001 07:36	21 Nov 2001 06:36	22 Nov 2001 05:36	112001 Lorenzo Kinney	90	0	Completed	Update

图 27-2 查看一个对象集合的细节

...

```

<%
for (int i = 0;
    i < (BnEngCallSearchRes.getBottom() -
        BnEngCallSearchRes.getTop() + 1); ++ i)
{
    int top = BnEngCallSearchRes.getTop()-1;
    Map item = BnEngCallSearchRes.getCallItem(i + top);
    int index = i+top;

    if (CallParam.getObject().getExternalCallNumberFlag()){
        out.write("<TD class=\"TextField\">" +
            item.get("externalSystemId") + "</TD>");
        out.write("<TD class=\"TextField\">" +
            item.get("callId") + "</TD>");
    }
    else

```

```

    {
        out.write("<TD class=\"TextField\">" +
            item.get("callId") + "</TD>");
    }
    out.write("<TD class=\"TextField\">" +
        item.get("callType") + "</TD>");
    out.write("<TD class=\"TextField\">" +
        item.get("actionNo") + "</TD>");
    if (!BnEngCallSearchRes.isAppointment())
    {
        out.write("<TD class=\"TextField\">" +
            item.get("symptom") + "</TD>");
    }
    out.write("<TD class=\"TextField\">" +
        item.get("customerId") + "</TD>");
    out.write("<TD class=\"TextField\">" +
        item.get("customerName") + "</TD>");
    out.write("<TD class=\"TextField\">" +
        item.get("siteId") + "</TD>");
    out.write("<TD class=\"TextField\">" +
        item.get("siteName") + "</TD>");

    ...

}
%>

...

```

虽然可以对这个特殊的代码示例进行改进（的确，这个代码写得不好），但它是许多编写 JSP 的人员写出来的典型示例。也许有人会说，体的内部应该使用带有 scriptlet 的 HTML 编写，因为在 HTML 中才真正需要动态代码。但经验表明，这样只会使事情更糟。将许多 TD 与 TR 这样的 HTML 元素以及许多小 scriptlet 混合在一起将会使页面完全不可读；在现在这种情况下，这些代码还能够被 Java 开发人员所阅读，但对于页面设计者，这些代码的可读性还是很差的。

在将页面交还给页面设计者之前，需要为一个调用封装那些写出特殊信息的代码，从而能够使用一个简单的标记写出对象，并且用户能够使用一个标记进行迭代。这里的第一项工作可以由 bodyless（不带体）标记完成，由于这不是讨论的重点，因此在这里不对它进行说明。这样做的最大好处是页面设计者能够使用如下形式的标记完成所需的工作：

```
<vn:call object="currentCall" format="CALL_VIEW"/>
```

在这里，format 指定了对象应该如何被显示。不同的屏幕可能以不同的方式显示相同的对象，而用户则希望将这些封装在实现这个调用实体的类中。

然后，需要添加一个迭代标记（iteration）来消除 for 循环，它也允许为显示标记准备好调用 ID。下面给出了这个 JSP 的形式：

```
...

<vn:iterate
  collection="BnEngCallSearchRes"
  object="currentCall"
>
  <vn:call object=<%= currentCall %>" format="CALL_VIEW"/>
</vn:iterate>

...
```

在这个例子中，迭代标记并没有在体中做任何实际的处理。它只需要执行对自己的迭代，并且确保 object 属性连续不断地更新，从而被内部的标记所使用。需要说明的是，在这里去除了 from 与 to 功能，因为它们对于说明迭代标记没有什么用处，也就没有必要使代码变得复杂化。

下面分析这个标记的相关代码：

```
package com.rts.infrastructure;

...

import javax.servlet.jsp.*;

public class SimpleIterator extends BodyTagSupport {

    Iterator elementsList = null;

    String collection = null;
    String object = null;

    public void setCollection(String collection)
    {
        this.collection = collection;
    }

    public void setObject(String object)
    {
        this.object = object;
    }
}
```

```
public int doStartTag()
    throws JspException
{
    Iterator elementsList =
        (Iterator)pageContext.getAttribute("collection");
    if(elementsList.hasNext()) {
        return EVAL_BODY_TAG;
    }
    return SKIP_BODY;
}

public void doInitBody()
    throws JspException
{
    pageContext.setAttribute(object, elementsList.next());
}

public int doAfterBody()
    throws JspException
{
    if(elementsList.hasNext()) {
        pageContext.setAttribute(object, elementsList.next());
        return EVAL_BODY_TAG;
    }

    return SKIP_BODY;
}
}
```

在这里可以注意到的第一件事情就是，标记现在是从 `BodyTagSupport` 继承而来。

紧接着可以看到用来从页面上下文中访问 `iterator` 对象的标记属性，以及用来保存对象的名称，这些对象将用于处理标记体。

当第一次进入这个标记时，调用 `doStartTag` 方法。这个方法从页面上下文获取 `iterator` 对象并获取这个集合中的第一个元素。如果这个集合为空，那么它将返回 `SKIP_BODY`，用来告知 JSP 引擎不要对体进行处理。这正是迭代循环提供的必要行为，因为在体中可能会存在静态 HTML，并且这些静态 HTML 在该集合为空时不应该被写出去。

在处理标记的开始部分之后，JSP 引擎将处理标记体；这在 JSP 规范中已经定义，作为 `BodyTag` 生命周期的一部分。然后，将调用 `doInitBody`，这时就可以将当前对象插入到页面上下文中。这个对象后来将被 `vn:call` 标记所使用。在 JSP 处理完标记体之后，它将调用 `doAfterBody` 方法。在这里，将再次测试其中是否还有对象需要迭代，如果有的话，将返回 `EVAL_BODY_TAG`，用来告知 JSP 引擎再一次处理标记体。在进行这一动作之前，需要重

新准备内部标记所使用的对象。如果没有更多的元素需要处理，那么将返回 SKIP_BODY，并且 JSP 引擎完成对标记的处理。

为了在 WebSphere 节点上部署这个标记，也需要使用 TLD。vn:iterate 的标记描述符稍有不同，因为这是一个体标记：

```
<tag>
  <name>iterate</name>
  <tagclass>com.rts.infrastructure.SimpleIterator</tagclass>
  <teiclass>
    com.rts.infrastructure.SimpleIteratorExtraInfo
  </teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
    Iterates over a collection
  </info>

  <attribute>
    <name>collection</name>
    <required>true</required>
    <rtexprvalue>>false</rtexprvalue>
  </attribute>
  <attribute>
    <name>object</name>
    <required>true</required>
    <rtexprvalue>>false</rtexprvalue>
  </attribute>
</tag>
```

在这个 TLD 中，有两个问题值得注意。第一个就是 bodycontent 的值。由于这是一个带有体的标记，因此需要指出存在一个体并且说明它的类型。可能的值是 JSP 与 tagdependent。默认值为 JSP，这个值意味着体的内容应该被 JSP 引擎所处理，这与前面讨论的相同；当体的内容应该被标记所读取并处理时，应该使用 tagdependent，也就是，当体确实是关标记（enclosing tag）的一部分而不是页面的一部分时，应该使用 tagdependent。当使用属性将创建许多繁杂的代码时，这个选项是非常有用的。例如，在附件的那个例子中，我们提到过，有时可能希望给出所有与 multiple 对象相关联的附件（例如，与工作相关联、与用户相关联，或与负责这一工程的组织相关联）。由于希望从其中提取出附件的对象数量是动态的，因此使用一个类似于 XML 的嵌入元素进行处理将更好，如下所示：

```
<vn_attachment:attach_to_multiple_objects >
  <attach_to
    entity="ServiceOrder"
    key="id"
```

```

        value="1234"/>
    <attach_to
        entity="User"
        key="id"
        value="100222"/>
    <attach_to
        entity="Department"
        key="id"
        value="QA"/>

/>

```

在这个例子中，内部的元素不是自定义标记；相反，它们仅仅是关标记所使用的内容，用来从多个对象中动态地提取附件。JSP 引擎不应该对这个体进行处理，因为这样将会失败；相反，标记处理方法将使用返回 `BodyContent` 对象的 `getBodyContent()` 读取体内容。

这个 TLD 中另一个有趣的事情就是 `teiclass` 元素。它代表标记附加信息 (tag extra info)，这是一个可以为每个标记添加的类，用来描述此标记实现的元数据属性。在本例中需要使用元数据类，因为此标记动态地创建新属性并且将它们插入到页面上下文中。在这种情况下，标记的附加信息类 (extra info class) 就定义了所插入属性的数据类型与作用范围：

```

package com.rts.infrastructure;

import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;
import javax.servlet.jsp.tagext.VariableInfo;
...

public class SimpleIteratorExtraInfo
    extends TagExtraInfo {

    public VariableInfo[] getVariableInfo(TagData data)
    {
        VariableInfo[] rc = new VariableInfo[1];

        rc[0] = new VariableInfo("currentCall",
                                com.rts.call.Call,
                                true,
                                VariableInfo.NESTED);

        return rc;
    }
}

```

`NESTED` 表示这个变量仅仅在自定义标记的开始与结束之间可用。这是元数据的主要作用。

在大多数情况下，TagExtraInfo 对象也用来保证标记使用的有效性。标记实现者可以实现 isValid 方法，并且将所有有效性检查代码放在其中（而不是放在 doStart 方法或 doInit 方法中），从而确保在对标记的处理开始之前就能够确定其有效性。

27.2.3 在 WebSphere 中部署标记库

在 WebSphere 下部署 taglib 很简单，并且更重要的是，它遵守 JSP 1.1 与 J2EE 规范定义的标准。在部署 taglib 的过程中涉及的元素是 JSP、实现此标记的类、TLD 以及 Web 应用程序。可以单独地管理所有这些实体，或者，将它们打包到一个 WAR 文件中。虽然手工部署工作仅仅涉及将文件放置到正确的位置并更新 web.xml 文件，但我们还是建议在 Web 模块中部署 taglib。

使用 WAR 文件的优点是，Web 应用程序能够在一个地方被管理，并且使用 WAR 进行管理与配置控制更简单。可以创建一个 WAR 文件，并且将它作为一个 Web 模块发布到应用程序服务器上。在这个例子中，WAR 文件必须包含在一个 EAR（Enterprise Archive）文件中，然后将这个文件加载到服务器。

为了创建 WAR 文件，需要使用 AAT（Application Assembly Tool）。可以从 Administration Console 启动这个工具，或者直接从命令行 Programs 菜单启动该工具。

1. 在 WebSphere Administrative Console 中，选择 Tools|Application Assembly Tool。
2. 在 Application Assembly Tool 窗口中，选择一个向导，用来指导完成装配过程。在这个例子中，应该选择 Create Web Module Wizard 图标。



3. 在这个向导中填写 WAR 信息，如图 27-3 所示。当后面将 Web 模块添加到企业应用程序中时，也将用到这个屏幕显示。通过使用下拉列表，可以设置一个容器应用程序（假设已经有这样的应用程序）。在这个过程的后面，将会创建企业应用程序。单击 Next 按钮进入下一个阶段。
4. 添加类资源。选择适当的 JAR 文件，并且将所有这些文件添加到 WAR 中。为了添加一个 JAR 文件，应该单击 Add Jar Files 然后选择这个文件。每个 JAR 文件都添加一个类集合，如图 27-4 所示。标记实现类必须被包括作为一个资源。在完成添加资源的工作之后，单击 Next 按钮。

为了创建 Web 模块，还有许多附加的步骤；这些步骤与其他装配过程没有什么不同。现在这里关注的是添加标记库，如图 27-5 与图 27-6 所示。将使用向导的这个步骤来添加 TLD。可以添加任何数目的 TLD 到 WAR 中。由于 TLD 包括了指向 JSP, Java 类等的元数据，因此仅仅需要将 TLD 添加到 WAR 中。

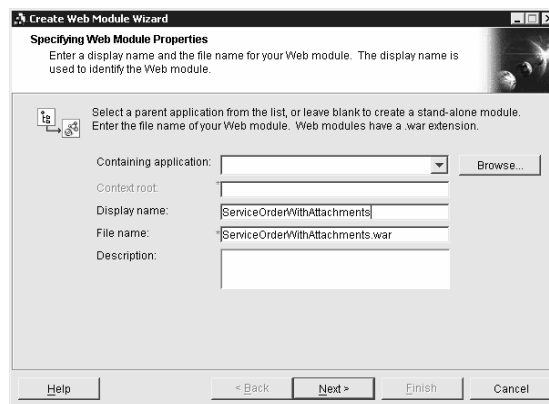


图 27-3 指定 WAR 信息

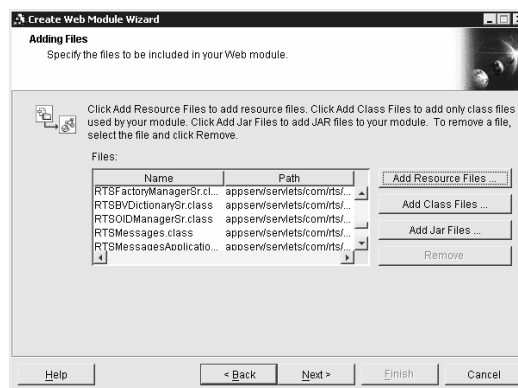


图 27-4 添加资源与 JAR 文件

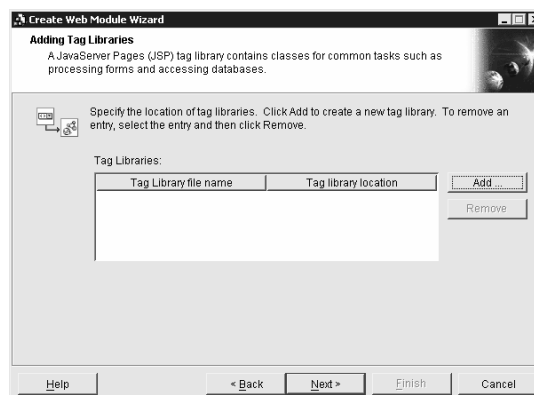


图 27-5 添加标记库



图 27-6 添加 TLD

5. 添加所有的 TLD（在这个例子中，只有一个被添加）。
6. 现在就可以完成 WAR 的装配工作，并保存它。
7. 下一步，将创建 EAR，这一工作也可以使用装配工具中某个适当的向导来完成。
8. 在这里不给出关于创建 EAR 的所有步骤，需要记住的重要一点是，在向导的 Adding Web Modules 步，需要将 WAR 加载到 EAR 中，如图 27-7 所示。

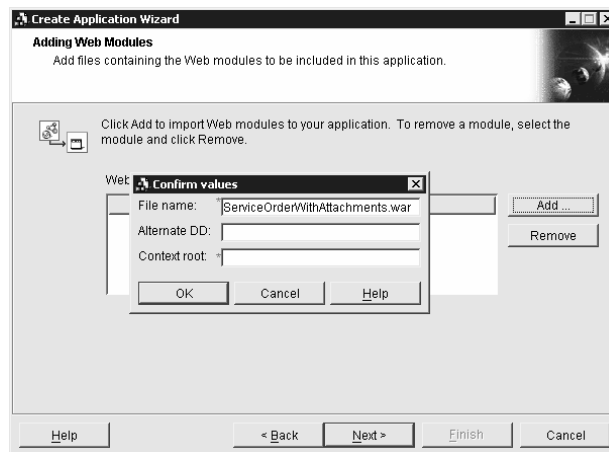


图 27-7 将 WAR 添加到 EAR 中

9. 在保存了 EAR 之后，退出 Application Assembly Tool 并返回到 Administration Console。在这里，需要安装企业应用程序。通过右击树面板中的 Enterprise Application 文件夹并选择 Install Enterprise Application 选项来完成这一工作。
10. 在 Administration Console 中，右击将要部署的节点并选择 Regen Webserver Plugin 选项，再次生成 Web 服务器插件。这是此过程的最后一步。

这样，标记库以及应用程序就安装到了 WebSphere 服务器中，可以准备运行了。

27.2.4 标记库资源

标记库是 Web 应用程序领域一个相对较新的事物，我们相信，它们的广泛使用将会导致构

建 JSP 方法的革命。在理想的编程世界中，所有 JSP 将回到它们本应具有的面目中来，也就是它们仅仅是 Web 页面。随着标记库在 Java 开发群体中得到越来越多的使用，标记库将成为一种主流技术，并且它将不仅仅用于内部开发方案。可以预见，开发人员将会开始使用商业化的标记库，而不是自己去构建这些标记库。

为此，有必要熟悉 Web 上的一些资源，它们为了解最新的标记库信息提供了很好的开端。下面列出了部分这样的站点：

- <http://jakarta.apache.org/taglibs/>：Jakarta taglib 工程网站是读者必须了解的。其中有一个工程就是标准标记库（JSTL，<http://jakarta.apache.org/taglibs/doc/standard-doc/intro.html>），这个标记库的作用并不亚于标准模板库对 C++ 的作用。
- <http://jsptags.com/tags/>：这里提供了大量的标记库集合，并且给出了到达其他相关网站的链接。
- <http://www.taglib.com>：在这里为 JSP 开发人员提供了资源与新闻。
- <http://www.jspin.com/home/tags>：这个站点提供了超过 80 个的标记库。
- <http://www.javaskyline.com/dev.html>：这是关于标记库的集合。
- <http://www.jspinsider.com/index.view>：这是一个关于 JSP 与标记库的好资源。

27.3 小 结

在过去三年中，自定义标记库已经成为“JSP 模型中最重要的发展”，并且，它们必将引导 JSP 开发的革命。最后，对自定义标记库的使用将持久地改变 JSP 编程的风格，并且使得在构建 JSP 过程中的分工更加明确，构建可重用组件的 Java 应用程序开发人员与创建实际 Web 页面 JSP 页面设计者之间的分工将更加清晰。

本章重点讨论了标记库如何改变 JSP 编程的方式。下一章将讨论如何调试 servlet 与 JSP。

第 28 章 使用 VisualAge 调试 servlet 与 JSP

WebSphere 运行时平台将支持 servlet 与 JSP 作为核心技术。前面已经分析了一些如何在 WebSphere 下部署 servlet 与 JSP 的例子。关于 servlet 与 JSP 的开发与部署就成为本章的话题。

可以使用任何开发环境来开发 servlet 与 JSP,甚至可以使用普通的 JDK 来进行这项工作(利用 TextPad 或其他更好的文本编辑器),但 IBM 的 VisualAge for Java 提供了开发 servlet 与 JSP 的更好环境。VisualAge for Java 有一个内置的 WebSphere 测试环境,可以在 VisualAge 中激活这个环境,从而能够在 VisualAge IDE 内部完成“编程-测试-调试”周期的全过程。另一种可选的方法是,在其他 Java IDE 中使用一个文本编辑器(或使用站点构造器创建 JSP 页面)构建 servlet,然后,将它们安装到 WebSphere 环境中并查看结果(或“试着”查看结果)。

通过使用 VisualAge,开发生产率能够大幅度地提高。原因就是,VisualAge 提供了统一的环境与调试工具。读者将会见到,当使用 VisualAge for Java 的 WebSphere 测试环境时,编码、运行与测试所有这些工作都能够在此 IDE 中完成,而不必启动 WebSphere 服务器或管理工具。在使用 WebSphere 时,测试-调试过程是很长并且很困难的,特别当需要为代码中的每次修改都进行从装配到部署的所有过程时更是如此。此外,如果使用的是 VisualAge for Java 的企业版,那么版本控制与配置管理的所有问题都将自动作为 servlet 开发周期的一部分。

使用 WebSphere 测试环境的第二个优点是对调试工具的访问。如果使用另一个环境或编辑器,那么,为了知道程序正在干什么,惟一的方法就是使用一系列输出语句,用来检查代码运行到何处、变量的值是什么以及编译其他信息。而如果使用 WebSphere 中的测试环境(VisualAge IDE),那么 servlet 与 JSP 开发人员就能够使用其中所有的调试工具。这意味着可以在 servlet 代码中插入断点并且使 WebSphere 测试环境在这个断点停止下来,从而允许在构建一个普通、老的 Java 应用程序时可以做到的事情在这个测试环境下也能够做到。

虽然 WebSphere 与 VisualAge 之间的紧密集成还有许多工作要做,但基于 Java 的服务器的标准化趋势,意味着其他 IDE 最终将成为使用 VisualAge for Java 的可选方法。

28.1 在 VisualAge 中开发 servlet

VisualAge for Java 提供了创建 servlet 的向导,它类似于第 17 章中用来创建 applet 的向导。为了调用 Create Servlet Wizard,应该在工作区中右击鼠标,然后在弹出菜单中选择

AddServlet。这时，向导就打开了，如图 28-1 所示。

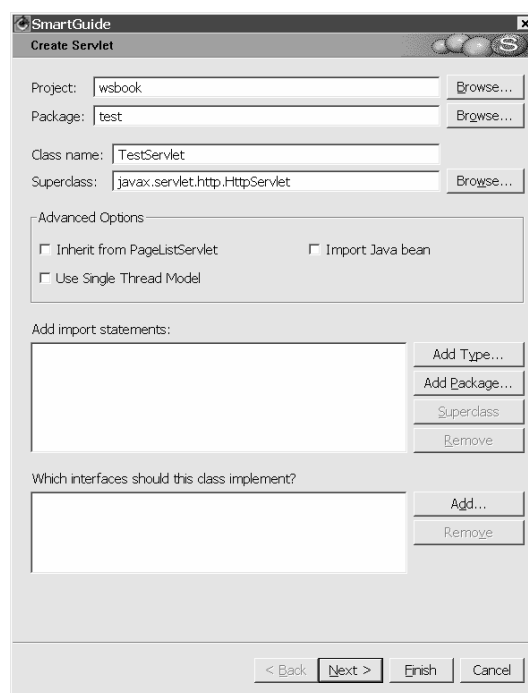
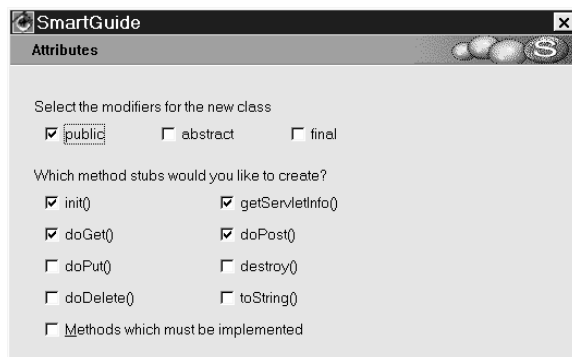


图 28-1 Create Servlet Wizard 打开屏幕

在这里，应该指定工程名、包名、类名。默认情况下，新创建的 servlet 从 HttpServlet 继承，但如果需要使用一个不同的超类，也可以对这个值进行修改。此外，还可以添加导入语句，并指定 servlet 应该实现什么接口（不同于 Servlet 接口）。

在单击 Next 之后，向导屏幕将要求为新类指定 Java 修饰，并且指出需要实现哪些方法。默认的选择（如下图所示）将实现 init，doGet，doPost 与 getServletInfo 方法。



单击 Finish，可以看到新的 servlet 就创建好了。对 doGet 与 doPost 的默认实现都调用一个

performTask 方法，它将接收 HTTP 请求与响应对象。在第 4 章讨论的“Hello World”servlet 中，在这里将添加一个代码段，用来产生一条“Hello World”消息。但是在这里我们将不考虑支持英语和法语两种情况。在插入新代码之后，performTask 方法如图 28-2 所示。

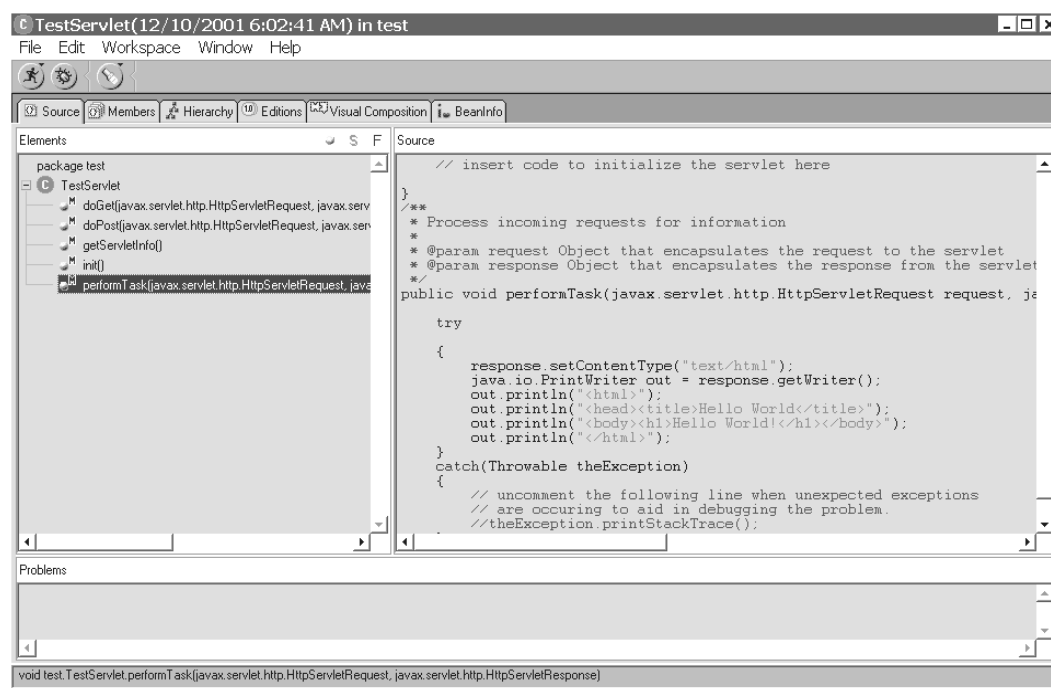


图 28-2 performTask 代码段

显然，VisualAge for Java 所有的功能都可应用于开发 servlet。在定义了 servlet（手工定义或使用向导）之后，用户能够在 VisualAge for Java 中添加或删除方法、浏览代码以及执行其他任务。但是，servlet 开发中功能最强大以及最有用的就是 WebSphere 测试环境。

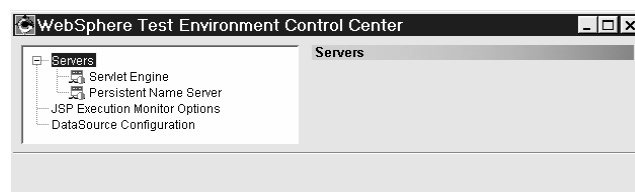
28.1.1 WebSphere 测试环境

WebSphere 测试环境是一个与 VisualAge for Java IDE 集成的瘦 Web 服务器与应用程序服务器。实际上，它并不是那么瘦，这也是其受批评的主要地方。它需要占据很多的内存，需要花费较长的初始化时间，并且在运行时比较慢。但是，在编码-测试-调试周期中，它比任何可用的工具都优越。

WebSphere 测试环境可用来在 Web 服务器下部署 servlet、服务 HTML 页面，以及处理传递给浏览器的包含 HTML 页面的 JSP 页面。实际上，它几乎包括了在将 servlet/JSP 电子商务应用程序在 WebSphere 下部署之前，为了确保这些电子商务应用程序正确工作所需的所有组件。这些工具被方便地打包到 IDE 中，因此它们的激活是很简单的。

下面说明了如何使用 WebSphere 测试环境：

1. 为了从 Workbench 中启动 WebSphere 测试环境，应该选择 Workspace|Tools 并选择 servlet 名。
2. 测试环境的启动过程需要几秒钟的时间，在完成启动时，将出现 WebSphere Test Environment Control Center 窗口，如下图所示。



这个窗口相当于测试环境的“Administrative Console”。IBM 使用与 WebSphere Administrative Console 中服务器所用颜色编码相同的颜色编码（蓝色表示正在运行，红色表示停止运行）。

3. 单击窗口左边面板中的 Servlet Engine 图标，这时，将见到与图 28-3 类似的窗口。

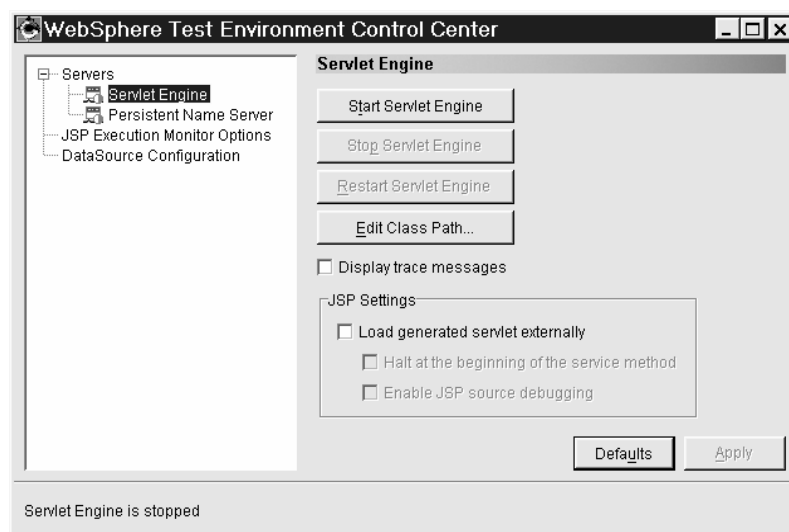


图 28-3 Servlet Engine

4. 在开始使用 servlet 引擎之前，需要为它设置 classpath。为了打开 Servlet Engine ClassPath 对话框（如图 28-4 所示），应该在 Test Environment Control Center 窗口中单击 Edit Class Path 按钮。

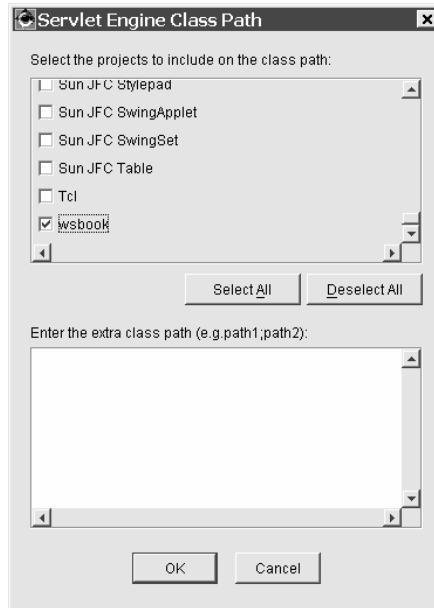
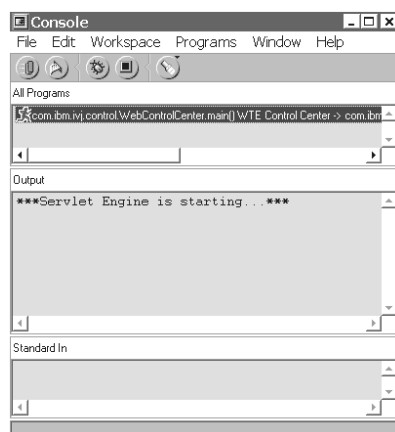


图 28-4 选择在其中放置了新 servlet 的工程

5. 选择在其中放置了新 servlet 的那个工程。这是一个很有用的功能，它允许以 VisualAge for Java 工程的方式来设置 classpath，而不是直接设置目录与文件（毕竟，这是用来开发与调试代码的环境）。在这个对话框中，也可以添加任何 VisualAge 外部的 classpath。
6. 在完成此工作之后，单击 OK 以返回到 Environment Control Center 窗口。
7. 在 Control Center 窗口中，单击 Start Servlet Engine。在 servlet 引擎初始化的过程中，VisualAge for Java Console 窗口将出现，并提供相应的进度信息，如下图所示：

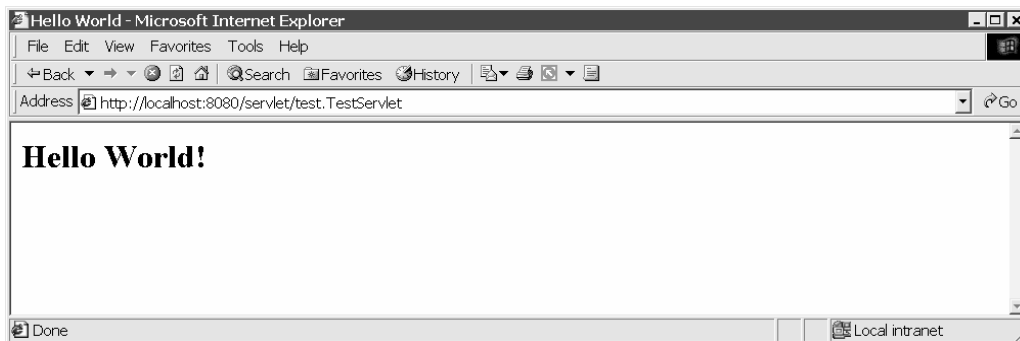


在初始化完成之后，将更新显示并反映 servlet 引擎正在运行。

现在，servlet 引擎正在运行，就可以调用小的测试 servlet 了。为此，请打开 Web 浏览器并使其指向 servlet URL。在这个 URL 中包括该 servlet 的全名。例如，上面给出的 TestServlet 所对应的 URL 是 `http://localhost:8080/servlet/test.TestServlet`。

关于这个 URL，有 3 点值得说明的地方。首先，主机名为 `localhost`，它就是正在运行 IDE 的机器（或者，可以使用与此相等价的 IP 地址 `127.0.0.1`）。并没有排除在一台不同的机器上远程访问 IDE 的可能性，但这并不是典型的情况，因为 WebSphere 测试环境通常被作为开发人员使用的个人服务器。默认情况下，端口号为 8080，因为这是“开发服务器”的一个约定。当然，也可以通过 VisualAge for Java Options 窗口对此进行修改。默认情况下，WebSphere 使用端口 80，因此这两个工具之间通常不会发生冲突，并且可以在正在运行 VisualAge 的相同机器上运行 WebSphere 服务器的实例（当然，假设在该机器上有足够的内存）。最后需要注意的是，URL 路径的命名规则是基于 servlet 与 servlet 所在包的名称的组合而得到的。在这里不能像使用 WebSphere 那样执行完全的应用程序部署。至少，在这里没有包括与注册 servlet 或部署 servlet 相关的问题。一旦对 servlet 完成了编码与编译工作，就能够在这里访问到它。

下面演示了这个 servlet 的输出：



28.1.2 调试 servlet

在安装、加载、启动测试环境之后，剩余的工作就很简单。这个环境的优点在于，所有的 IDE 工具现在都可用于调试 servlet。正如在第 17 章描述的那样，可以插入断点、检查变量以及执行其他任务。例如，可以在 `performTask` 方法上设置一个断点，如图 28-5 所示。在下次 servlet 被访问时（例如，通过单击浏览器上的 refresh 或 reload 按钮），测试环境就开始执行 servlet 并将遇到这个断点，然后，就显示出调试器，如图 28-6 所示。

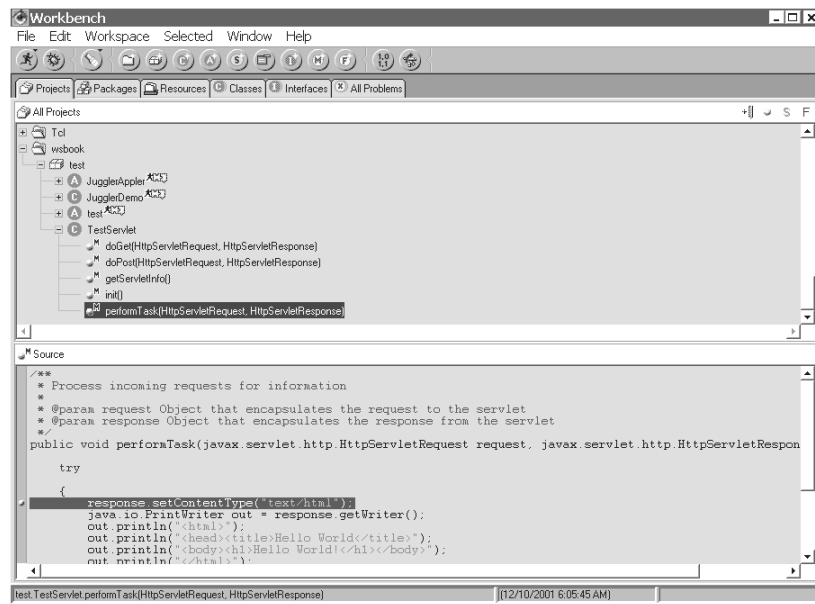


图 28-5 设置一个断点

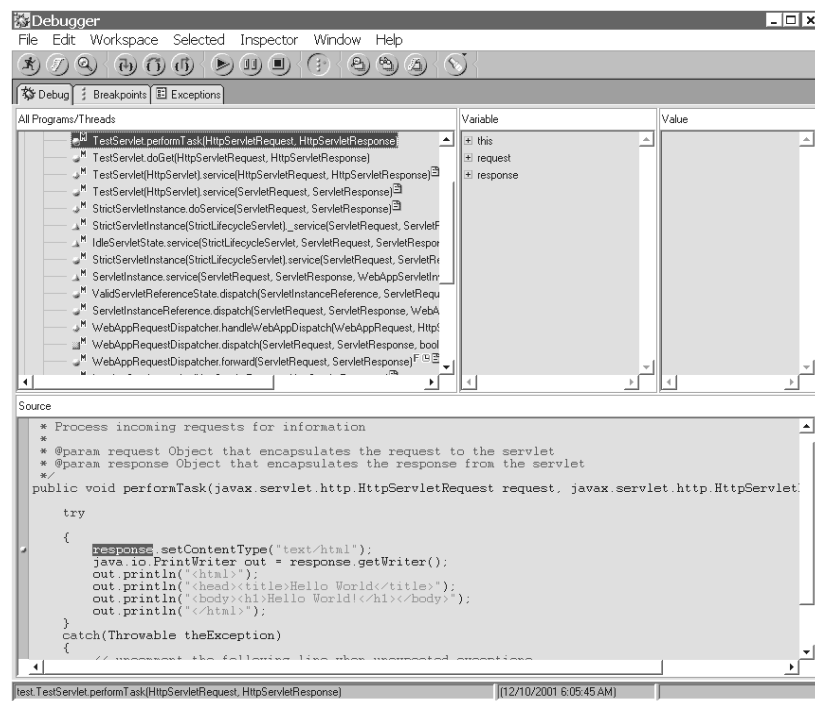


图 28-6 调试器

在调试器中，可以像 IDE 中所做的那样自由地进行任何工作（可以检查这个调试器能够访问的任何对象）。例如，可以检查请求对象 URI，如图 28-7 所示。

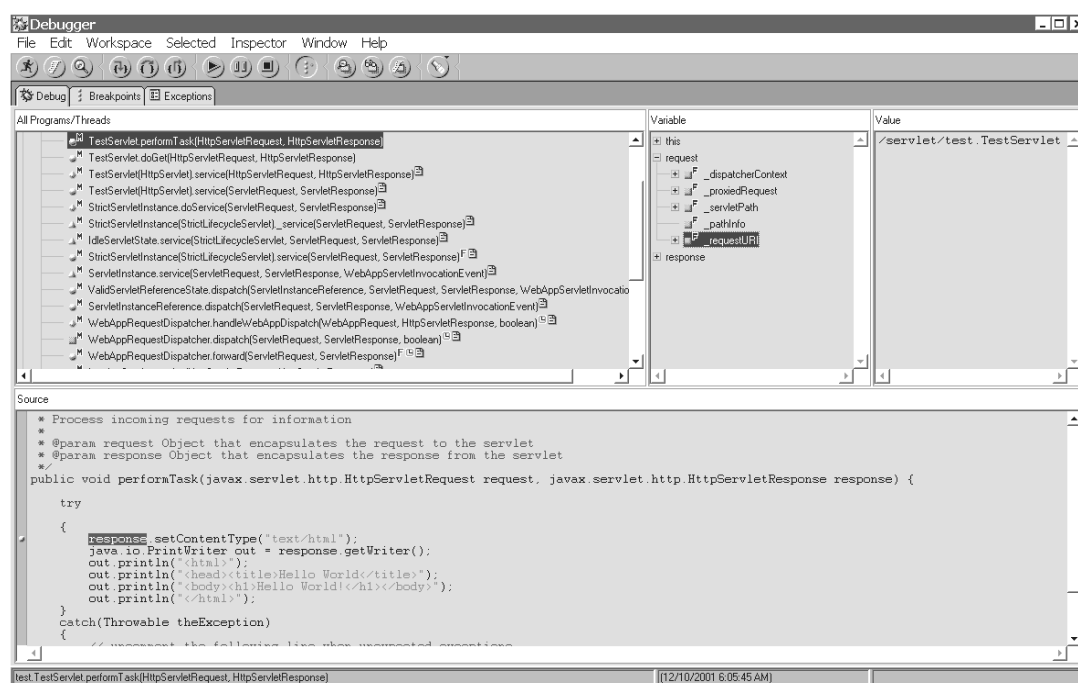
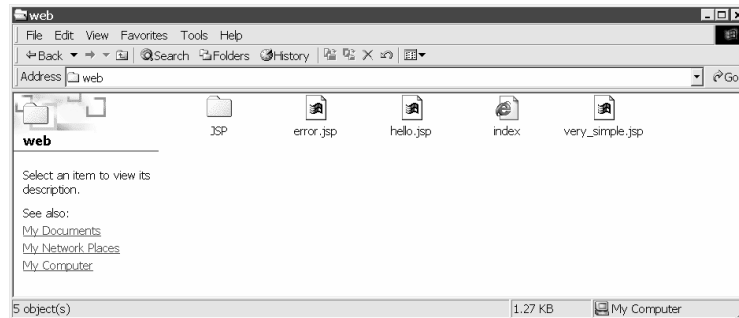


图 28-7 检查请求对象 URI

与使用真实的 WebSphere 安装进行软件测试与调试相比，这些调试功能能够节省大量的工作。它不仅能够大大缩短开发周期，而且使许多开发工作变得大不一样。

28.1.3 调试 JSP

本章还介绍可用来进行 JSP 开发与调试的工具。与 servlet 的开发和执行环境不同，JSP 需要进行一些细小的部署。如前所述，VisualAge for Java WebSphere Test Environment 运行有一个小的 Web 服务器，就像实际的 IBM HTTP 服务器一样，这个 Web 服务器也有一个“文档根”。默认情况下，这个根在 VisualAge for Java 安装目录的 ide\project_resources\IBM WebSphere Test Environment\hosts\default_host\default_app\web\目录中。下面显示了这个目录的默认内容。



为了在 VisualAge for Java 中运行 JSP，首先需要将这些 JSP 放置在这个目录中。例如，可以将第 4 章中的那个“Hello World”JSP 放置在前述的目录中，作为 hello.jsp 文件，然后从 Web 浏览器上访问这个文件，如图 28-8 所示。需要说明的是，这里的 URL 使用与前面相同的主机名与端口，并且路径是相对于 web 目录的。



图 28-8 从 WebSphere Test Environment 运行 hello.jsp

如果需要调试 JSP（通常都要求这样），那么必须启用 JSP 的执行监视功能，这个功能在 Environment Control Center 窗口的 JSP Execution Monitor Options 中设置，如图 28-9 所示。

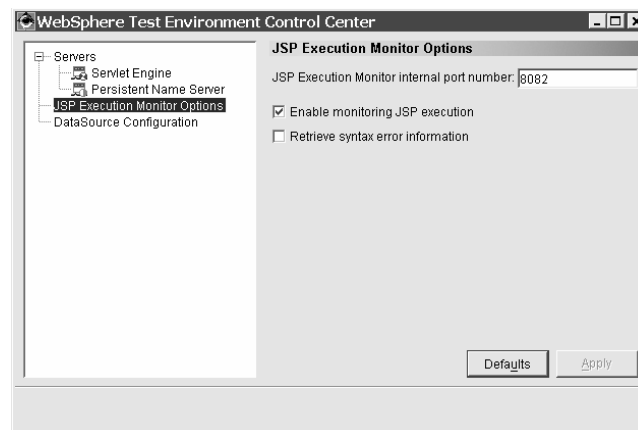


图 28-9 JSP Monitor Options

在启用 JSP 监视功能之后，当调用一个 JSP 时（例如，在浏览器窗口上单击 refresh 按钮），VisualAge for Java 将自动为 JSP 启动 Execution Monitor 窗口，如图 28-10 所示。JSP Execution Monitor 是 JSP 的一个特殊调试环境。应该记得，JSP 将被转换成 servlet。

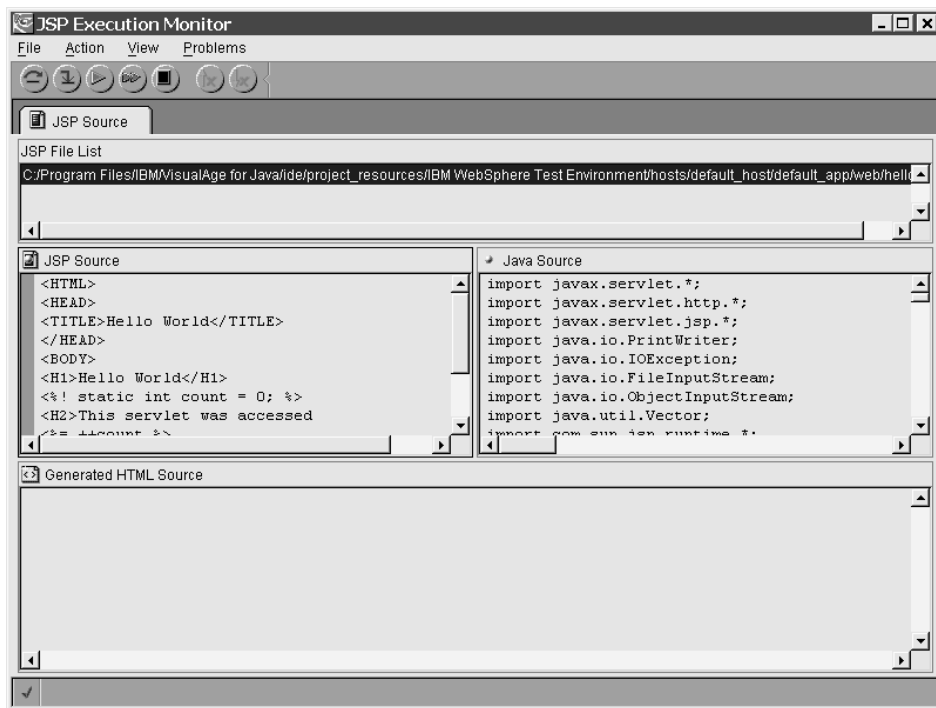


图 28-10 JSP Execution Monitor

JSP Execution Monitor 显示 4 个子窗口，分别为：

- 浏览器所启动的 JSP 列表
- JSP 源代码
- servlet Java 源代码
- 所生成的 HTML

JSP Monitor 允许用户单步执行 Java 代码，并且查看动态生成的 HTML 内容。它并没有提供 VisualAge for Java 调试器的全部功能，但它对于调试 JSP 仍然是有用的。为了运行 JSP，必须单击 JSP 监视工具栏上的 Step，Run 或 Fast-Forward 按钮。

也可以利用 JSP Execution Monitor Options 控制 JSP Monitor 的设置。例如，可以启用或禁用 JSP 监视功能、语法错误检查功能，也可以设置用于 JSP 监视的端口（需要说明的是，这些端口是用于 VisualAge for Java 内部通信的，与浏览器中用于 HTTP 连接的端口 8080 没有关系）。

28.2 小 结

IBM 是 Java、Java 应用程序服务器以及电子商务的主要提供商之一。VisualAge for Java（这对于使用企业版的多人开发组提供了最佳的支持）与 WebSphere 应用程序服务器的集成，从而形成 WebSphere Test Environment，这对于 WebSphere 应用程序开发人员是最重要的。如果计划使用 WebSphere Application Server 部署应用程序，我们强烈推荐使用 VisualAge 进行应用程序开发并使用本章介绍的工具。

本章将结束对 servlet 与 JSP 的讨论。下面将进入 Enterprise JavaBean。

第 6 部分 开发 EJB

第 29 章 会话 bean

本章开始深入讨论 EJB (Enterprise JavaBean, 企业 JavaBean), EJB 是一个服务器方组件体系结构, 它支持并简化使用 Java 构建企业级分布式应用程序的过程。正如在第 11 章讨论的那样, EJB 分为两类: 会话 bean 与实体 bean。这两种分类之间的主要区别在于持续性 (persistence) 这一概念。会话 bean 代表实现业务逻辑的服务器方组件, 它是非持续性的; 当需要在一个单独的用户会话中保持 bean 状态的持续性时, 就应该使用实体 bean。在第 5 章已描述了一个简单的无状态 “Hello World” 会话 bean。第 11 章详细讨论了 EJB、EJB 体系结构、角色与框架, 并且描述了会话 bean 与它们的生命周期。本章将给出一些会话 EJB 的示例。通过这些示例, 读者将了解到如何在 WebSphere Application Server 中构建与部署会话 bean。在这里的讨论中, 既包括无状态会话 bean 也包括有状态会话 bean, 此外, 还包括 EJB 的所有方面 (构建 EJB、装配 EJB 与部署 EJB)。

29.1 PageFetch: 无状态会话 bean

无状态会话 bean 不维护对话状态, 换句话说, 当无状态会话 bean 被调用时, 它并不维护任何用于后续调用的上下文。这样导致的一个结果就是, EJB 容器能够将无状态会话 bean 放置在池中并重用它们。例如, 可以使用相同的实例为不同客户的不同调用服务。这样做是可能的, 因为无状态会话 bean 的实例是完全相同的。不过, 虽然无状态会话 bean 没有维护对话状态, 但这并不意味着它不能有任何成员变量。但是, 这些变量的使用有一定的限制, 因为客户必须不依赖于它们的存在性 (也就是, 客户不能假设自己后来的方法调用实际上使用了相同的 bean 实例)。如果某个无状态会话 bean 有状态 (即它包含有成员变量), 那么这些成员变量应该被声明为 final。如果没有将它们声明为 final, 那么当两个或多个 bean 客户同时访问无状态会话 bean 的状态时, 就可能出现竞争冲突。

本节给出的无状态会话 bean 的例子是 PageFetch bean。从第 11 章的讨论已经了解到, 使用会话 EJB 的一个目的就是支持应用程序划分。这里所讨论的 bean 就是如此。这个 bean 代表客户并获取一个 Web 页面。然后, 对这个 Web 页面的内容进行压缩并返回给客户。假设在

其中部署了这个 bean 的 EJB 服务器与保存有页面内容的 Web 服务器在位置上很接近(例如, 两者都在相同的 LAN 上), 这种 bean 对于获取“繁忙的”Web 页面是有用的。

29.1.1 PageFetch 远程接口

开发新企业 bean 的第一步就是定义远程接口, 远程接口用来定义与 bean 用户的合约。在这个例子中, bean 提供了一个名为 getPage 的“业务”方法。这个方法使用一个 HTTPURL 作为参数, 然后获取页面并以压缩的形式返回该页面。这个接口如下所示:

```
import javax.ejb.*;
import java.rmi.*;
import java.io.*;

public interface PageFetch extends EJBObject
{
    byte[] getPage(String url) throws RemoteException, IOException;
}
```

这个远程接口对作为 javax.ejb 包一部分的 EJBObject 接口进行了扩展。实现这个接口的类是由 EJB 容器产生的。在这个远程接口中定义的每个方法都必须能够发出 RemoteException 异常, 用来处理由于某些通信或其他问题阻止方法被调用的情况。这一要求主要来自使用 Java RMI 调用 EJB 方法这一事实。此外, 这个方法的每个参数与返回的值都必须是 Java 原始类型或必须实现 Serializable 接口。

getPage 方法也能够发出 IOException 异常。当它访问由 url 所指定的 Web 页面时, 可能出现某些网络失败。导致这种失败的原因, 可能是 URL 书写错误, 或者是服务器或网络停机等各种原因。所有这些情况都能够通过发出异常表现出来。

29.1.2 PageFetch 本地接口

下一步就是为 EJB 定义本地接口。本地接口用来创建与析构 EJB 实例。EJB 本地接口对 javax.ejb 包中定义的 EJBHome 接口进行扩展。

```
import javax.ejb.*;
import java.rmi.*;
import java.io.*;

public interface PageFetchHome extends EJBHome
{
    PageFetch create() throws RemoteException, CreateException;
}
```

`create` 方法不需要任何参数，它返回前面定义的远程接口。一般来说，本地接口的 `create` 方法可以使用构建 `bean` 对象所需的任何参数。对于无状态 `bean`，通常不需要参数。`create` 方法能发出 `RemoteException` 异常（由于 RMI 问题）或发出 `CreateException` 异常，用来表示虽然通信不存在问题，但服务器仍然不能创建 `bean` 实例（例如，由于内存限制）。

29.1.3 PageFetch bean 实现

在定义了上述两个接口之后，现在就可以讨论这个 `bean` 的实现了。首先需要说明的是，`bean` 实现了 `javax.ejb.SessionBean` 接口。它没有实现 `PageFetch` 接口（远程接口），因为这个接口已经由 EJB 容器所生成的一个 `stub` 类所实现。

这个 `bean` 实现也可以实现其他接口或扩展一个类。为了能够访问下一节描述的有状态 `bean`，有必要将这个 `bean` 的实现分为两部分。首先，定义 `PageBean` 类，这个类实现 `SessionBean` 接口。在这个类中有两个辅助方法，用来允许加载 Web 页面与压缩一个字符串：

```
import javax.ejb.*;
import java.rmi.*;
import java.io.*;
import java.util.*;
import java.net.*;
import java.util.zip.*;

public class PageBean implements SessionBean
{
    // Implementation of "business" method
    public byte[] getPage(String url) throws IOException
    {
        // Get the page content.
        StringBuffer buf = loadPage(url);
        // Compress the page.
        return pack(buf);
    }

    // EJB methods
    public void ejbCreate()
    {
    }

    public void ejbRemove()
    {
    }

    public void ejbActivate()
```

```
{
}

public void ejbPassivate()
{
}

public void setSessionContext(SessionContext context)
{
}

// Given a URL, access the page and return it in a string buffer.
protected StringBuffer loadPage(String urlString) throws IOException
{
    URL finalUrl = new URL(urlString);
    StringBuffer stringBuffer = new StringBuffer();
    URLConnection urlconnection = finalUrl.openConnection();
    urlconnection.setDoInput(true);
    urlconnection.setRequestProperty("User-Agent",
        "Mozilla/4.0(compatible; MSIE 5.5; Windows 98)");
    InputStreamReader reader = new InputStreamReader
        (urlconnection.getInputStream());
    int i = urlconnection.getContentLength();
    int j;
    char[] inbuf = new char[4096];
    for (; (j = reader.read(inbuf)) != -1; ) {
        stringBuffer.append(inbuf, 0, j);
        int k = stringBuffer.length();
        if (i <= 0) {
            continue;
        }
        if (i <= k) {
            break;
        }
    }
    return stringBuffer;
}

/*
 * Compress the byte stream before sending it. It is worth doing
 * a bit of extra work because the bytes are normally sent over
 * a slow wireless line.
 */
protected byte[] pack(StringBuffer stringBuffer) throws IOException
```

```

    {
        String pageStr = stringBuffer.toString();
        ByteArrayOutputStream s = new ByteArrayOutputStream();
        ByteArrayOutputStream st2 = new ByteArrayOutputStream();
        ObjectOutputStream out = new ObjectOutputStream(s);
        out.writeBytes(pageStr);
        out.flush();
        byte[] full = s.toByteArray();
        int fullsz = s.size();

        Deflater def = new Deflater();
        def.setStrategy(Deflater.DEFAULT_STRATEGY);
        def.setLevel(Deflater.BEST_COMPRESSION);
        def.setInput(full);
        DeflaterOutputStream os = new DeflaterOutputStream(st2, def);
        int len = fullsz;
        int st = 0;
        int jump = 1024;
        byte[] buf = new byte[1024];

        while (st < len) {
            if (len - st < 1024)
                jump = len - st;
            System.arraycopy(full, st, buf, 0, jump);
            os.write(buf, 0, jump);
            st += 1024;
        }
        os.close();
        return st2.toByteArray();
    }
}

```

EJB 方法的实现是空的，主要是因为这个 bean 是一个无状态 bean，因此在它被创建、删除、激活或钝化之后并没有什么工作要做。在这里实现了两个 `protected` 辅助方法，可以被这个类的子类所使用。实际上，这两个辅助方法是这个类的主要部分。

`loadPage` 方法加载指定 URL 的 Web 页面，并返回一个保存有这个页面的内容的 `StringBuffer`。这个方法将建立一条 URL 连接，并通过这个连接读取指定的 Web 页面。`pack` 方法使用 `java.util.zip` 包压缩 `StringBuffer`。

下一个类就是本会话 bean 的实际实现。它扩展 `PageBean` 并实现了业务方法 `getPage`。通过实现这个方法以及 `SessionBean` 接口所需的方法，`PageFetchBean` 将编译出 EJB 容器希望从这个 bean 得到的实现。需要说明的是，这个方法必须被声明为 `public`，并且必须与远程接口中为它定义的签名相匹配。在签名方面惟一可允许的不同在于 bean 实现方法不需要发出

RemoteException 异常。如果它自己的逻辑需要执行 RMI，那么可能就需要发出此异常，在这种情况下，方法的签名就完全相同。但是，在这里并不是这种情况。

```
import javax.ejb.*;
import java.rmi.*;

public class PageFetchBean extends PageBean
{
    // Implementation of "business" method
    public byte[] getPage(String url) throws IOException
    {
        // Get the page content.
        StringBuffer buf = loadPage(url);
        // Compress the page.
        return pack(buf);
    }
}
```

正如前面说明的那样，bean 的超类实现了两个 protected 辅助方法，它们由这个类中定义的那个单独方法所使用。这样做是允许的，因为 bean 类是一个普通的 Java 类。但是，在 EJB 模型中给出了一些限制：

- bean 不能启动新线程或使用同步的关键字。
- bean 必须不是抽象的。
- 静态变量只有当被声明为 final 时（即所有静态变量实际上就是常量）才能使用。

编译器、装配工具或 EJB 容器并没有强制执行这些限制。因此，开发人员就存在出现这些错误的可能性；因此在大多数情况下，仍然需要非常仔细地设计 EJB 应用程序。

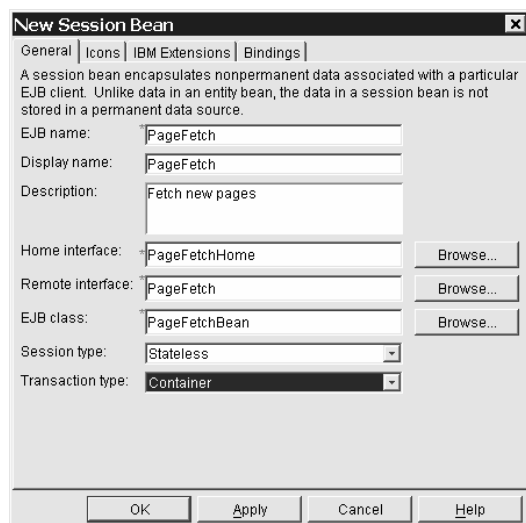
29.1.4 PageFetch bean 的装配与部署

bean 的装配与部署分别是使用 AAT 与 Administrative Console 完成的。

在添加一个新 bean（作为新 EJB 模块的一部分）时，需要使用 AAT 来设置 bean 参数。可以使用 New Session Bean 对话框中的 General 标签来指定 EJB 名、显示名与描述，如图 29-1 所示。这三个设置将用来辅助进行 bean 的部署。应该记得，部署 EJB 模块的人可能与应用程序装配者不是同一个人（甚至可能来自不同的组织）。其他参数还包括远程接口、本地接口与 bean 类。最后，可以选择 bean 是无状态的还是有状态的；在本例中是无状态的。此外，还为 bean 设置事务属性。关于事务属性，将在第 31 章讨论。但在这里值得说明的是，AAT 并不能自动地确定 bean 是否有状态，因为作为无状态 bean 并不意味着在其中就不存在成员变量。

可以使用 Icons 标签（在这里没有给出来）将一个图标与 bean 相关联。此后，这个图标就

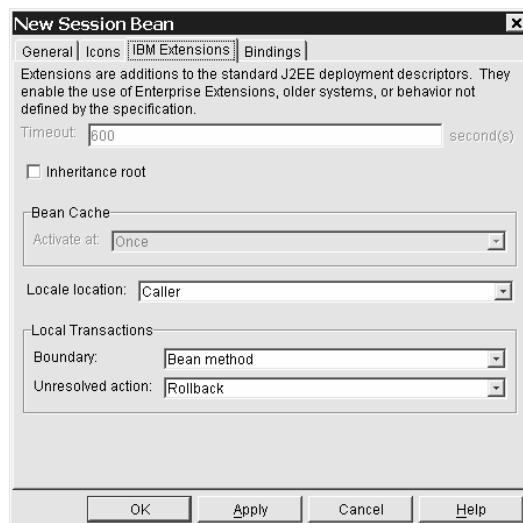
将在装配时与部署时被显示。图 29-2 所显示的 IBM Extensions 标签用来定义某些 WebSphere 特定属性，例如超时、cache 与本地事务。这将在第 33 章详细讨论。



The 'New Session Bean' dialog box, General tab. It contains the following fields and options:

- General | Icons | IBM Extensions | Bindings
- A session bean encapsulates nonpermanent data associated with a particular EJB client. Unlike data in an entity bean, the data in a session bean is not stored in a permanent data source.
- EJB name: PageFetch
- Display name: PageFetch
- Description: Fetch new pages
- Home interface: PageFetchHome (with a Browse... button)
- Remote interface: PageFetch (with a Browse... button)
- EJB class: PageFetchBean (with a Browse... button)
- Session type: Stateless (dropdown)
- Transaction type: Container (dropdown)
- Buttons: OK, Apply, Cancel, Help

图 29-1 New Session Bean 的 General 标签

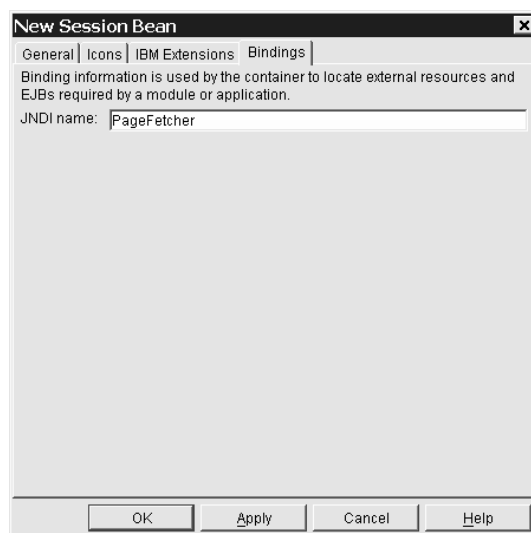


The 'New Session Bean' dialog box, IBM Extensions tab. It contains the following fields and options:

- General | Icons | IBM Extensions | Bindings
- Extensions are additions to the standard J2EE deployment descriptors. They enable the use of Enterprise Extensions, older systems, or behavior not defined by the specification.
- Timeout: 600 second(s)
- ☐ Inheritance root
- Bean Cache: Activate at: Once (dropdown)
- Locale location: Caller (dropdown)
- Local Transactions: Boundary: Bean method (dropdown), Unresolved action: Rollback (dropdown)
- Buttons: OK, Apply, Cancel, Help

图 29-2 IBM Extensions 标签

图 29-3 所示的 Bindings 标签用来决定这个 bean 的 JNDI 名。这个名称很重要，因为客户方代码使用这个名称作为访问 bean 的关键字。



The 'New Session Bean' dialog box, Bindings tab. It contains the following fields and options:

- General | Icons | IBM Extensions | Bindings
- Binding information is used by the container to locate external resources and EJBs required by a module or application.
- JNDI name: PageFetcher
- Buttons: OK, Apply, Cancel, Help

图 29-3 Bindings 标签

在将这个 bean 打包到一个 EJB 模块中之后（重复第 5 章所描述的步骤），向 WebSphere

Application Server 部署这个 bean 就很容易了。下一节给出一个使用 PageFetch bean 的小 JSP 页面，因此，将 EJB 与 JSP 打包在一起形成 EAR 文件是有意义的。EAR 文件中可以包括 0 个或多个 WAR 文件以及 0 个或多个 EJB 模块。

创建 EAR 文件

为了创建新 EAR 文件，应该选择 File|New|Application。然后，在 Application Assembly Tool 的左边面板中，通过右击 EJB Modules 图标并选择 Import，就可以将 EJB 模块导入 EAR 文件中。然后，也可以导入 Web 模块（通过右击 Web Modules 图标并选择 Import）。在添加 EJB 模块与 WAR 模块之后，AAT 的显示应该如图 29-4 所示。

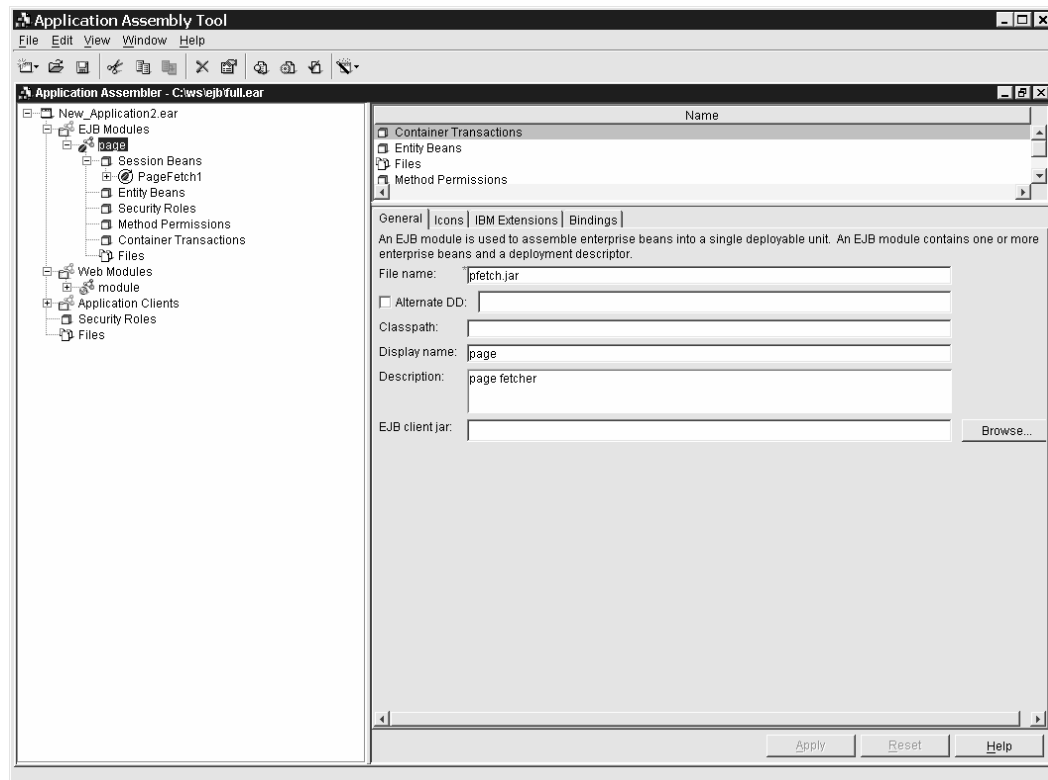


图 29-4 带有企业应用程序（包含 PageFetch EJB 和 Web 模块）的 AAT

可以注意到，Web 模块包含有一个单独的 JSP。29.1.5 节“可动作的 PageFetch”中将讨论这个 JSP。

AAT 将生成一个基于 XML 的部署描述符。关于这个 bean 的部署描述符如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
```

```

"-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
"http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar id="ejb-jar_ID">
  <description>page fetcher</description>
  <display-name>page</display-name>
  <enterprise-beans>
    <session id="Session_1">
      <description>Page Fetch1</description>
      <display-name>PageFetch1</display-name>
      <ejb-name>PageFetch1</ejb-name>
      <home>PageFetchHome</home>
      <remote>PageFetch</remote>
      <ejb-class>PageFetchBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor id="AssemblyDescriptor_1">
  </assembly-descriptor>
</ejb-jar>

```

在这个 XML 文件中，可以清楚地看到先前在 AAT 中所选择的 bean 属性：bean 描述、显示名、远程接口与本地接口、bean 实现类、无状态会话类型与事务类型。

29.1.5 可动作的 PageFetch

为了演示 PageFetch bean，在这里将实现一个简单的 JSP，这个 JSP 将访问一个特定的页面并且对数据进行解压，然后将这个页面显示给用户：

```

<%@page import="javax.ejb.*" %>
<%@page import="javax.naming.*" %>
<%@page import="java.util.*" %>
<%@page import="java.rmi.*" %>
<%@page import="java.util.zip.*" %>
<%
    String str = null;
    byte[] ret = null;
    byte[] full = null;
    try {
        Properties env = System.getProperties();
        env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming.WsnInitialContextFactory");

```

```
env.put(javax.naming.Context.PROVIDER_URL, "iiop://localhost");
InitialContext ic = new InitialContext(env);
PageFetchHome home = (PageFetchHome)
    javax.rmi.PortableRemoteObject.narrow(
        ic.lookup("PageFetcher"),
        PageFetchHome.class);
PageFetch bean = home.create();
ret = bean.getPage("http://www.viryanet.com");
Inflater inf = new Inflater();
inf.setInput(ret);
full = new byte[2000000];
len = inf.inflate(full);
str = new String(full, 0, len);
} catch (Exception ex) {
    %>
    Error <%= ex %>
    <%
}
    %>
<%= str %>
```

这个 JSP 的实现非常直观，在对内容进行解压并从 bean 返回之后，将输出一个页面。在这里已经指定要求 JSP 访问一个单独的常量 URL，当然，这样的做法是不实际的，但对于在这里演示 EJB 的使用已经足够了。

29.2 DiffPageFetch：一个有状态会话 bean

现在开始讨论有状态会话 bean。有状态 (state-ful) 会话 bean 保存了多个方法调用的对话状态。这意味着，在这个 bean 中有一些成员变量用来保存状态，并且，容器必须为每个客户提供一个单独的实例，因为有状态 bean 实例不再是可互换的。对于无状态 bean，容器可以为所有客户提供相同的实例，但对于有状态 bean，必须为每个客户管理一个实例。容器所做的工作通常就是钝化 bean（将它们放在持续性存储器中，并且为不同的客户使用对应的实例）。

这里给出的有状态会话 bean 的示例类似于前面的无状态会话 bean 的示例。同样，假设需要访问一个 Web 页面，但现在希望每次访问页面都有所不同。bean 必须能够向客户发送这些改变，这样就降低了网络流量。为此，bean 必须维护页面最新的状态。对于无状态 bean，实现这一功能的惟一方法就是在每次调用时向这个 bean 发送先前的页面内容，作为方法调用的参数。但这显然是对网络流量的浪费，并且它违背了计算页面差别并压缩数据的目标。有状态会话 bean 的状态是由它的非瞬时性成员变量值决定的。这些变量必须是原始的 Java

类型，或者，它们必须实现了 `Serializable` 接口，从而使容器能够将它们放置到外部存储器中。有状态会话 bean 节省了设计工作并使程序更简单。无状态会话 bean 能够产生可以更好地执行的系统，并且具备固有的可伸缩性。无状态会话 bean 在一个池中管理并且被多个客户所重用。它们创建了一个适合于企业应用程序的环境，在这种环境下，可伸缩性与性能也是最重要的因素。另一方面，有状态会话 bean 不能支持高负载，因此，很少用于产品应用程序。

29.2.1 DiffPageFetch 远程接口

与 `PageFetch` bean 一样，首先定义远程接口。这里所定义的 bean 提供了两个业务方法：第一个称为 `getPage`，用来获取页面并以压缩形式返回该页面。这个方法用来对页面进行完全刷新，或者用来获取页面的原始拷贝。不过，这个方法没有任何参数。bean 的一个实例对应于一个单独的 URL，因此，这个 URL 在构建时被传递给 bean。第二个方法是 `getDiff`，这个方法用来返回当前页面与最近使用 `getPage` 或 `getDiff` 所获取的页面之间的差别。

```
import javax.ejb.*;
import java.rmi.*;
import java.io.*;

public interface DiffPageFetch extends EJBObject
{
    byte[] getPage() throws RemoteException, IOException;
    byte[] getDiff() throws RemoteException, IOException;
}
```

关于有状态会话 bean，应该注意的一件事就是存储在其中的数据并不是真正持续性的，表现在当服务器失败时数据将丢失。这一点不同于实体 bean，实体 bean 能够完全确保数据的持续性，它使用外部数据存储器（例如数据库）来存储数据。因此，如果状态具有长期有效性的重要性，那么就不应该使用有状态会话 bean。例如，如果 bean 保存了某些与财务事务相关的数据，那么就不应该冒数据可能丢失的风险。在本例中，数据的丢失并不会导致严重后果，因为如果 bean 状态丢失，可以再次调用 `getPage` 来重新加载这个 Web 页面。

29.2.2 DiffPageFetch 本地接口

现在来分析本地接口，本地接口定义了 `create` 方法。在这个例子中，`create` 方法使用一个参数（页面的 URL），用于后续的 `getPage` 和 `getDiff` 调用。

```
import javax.ejb.*;
import java.rmi.*;
import java.io.*;
```

```
public interface DiffPageFetchHome extends EJBHome
{
    PageFetch create(String url) throws RemoteException, CreateException;
}
```

29.2.3 DiffPageFetch bean 实现

这个类也实现了 `javax.ejb.SessionBean` 接口，因为这个 bean 也是一个会话 bean。通过直接扩展 `PageBean` 类来实现 `SessionBean` 接口。

这个 bean 有两个成员变量，用来保存状态。其中一个变量保存了 bean 所接收的 HTTP URL，另一个变量保存了所接收页面的最新副本。

为了完整性，在这里实现了 `SessionBean` 接口的所有方法。需要说明的是，`ejbActivate` 方法与 `ejbPassivate` 方法仍然为空（因为在这里并没有使用任何需要被释放的资源），`ejbCreate` 接口使用一个参数（也就是 URL），并且将它存储在相关的成员变量中。`ejbRemove` 方法清除这个变量。

```
import javax.ejb.*;
import java.rmi.*;

public class PageFetchBean extends PageBean
{
    // Conversational State
    StringBuffer content;
    String url;

    // Implementation of "business" method
    public byte[] getPage() throws IOException
    {
        StringBuffer buf = loadPage(url);
        content = buf;
        return pack(buf);
    }

    public byte[] getDiff() throws IOException
    {
        if (content == null)
            return null;
        StringBuffer buf = loadPage(url);
        StringBuffer diff = doDiff(content, buf);
        content = buf;
        return pack(diff);
    }
}
```

```
}

// EJB methods
public void ejbCreate(String theUrl)
{
    url = theUrl;
    content = null;
}

public void ejbRemove()
{
    content = null;
}

public void ejbActivate()
{
}

public void ejbPassivate()
{
}

public void setSessionContext(SessionContext context)
{
}

// Auxiliary Method
protected String doDiff(String orig, String next)
{
    int ind1 = 0;
    int origLen = orig.length();
    int nextLen = next.length();
    String diff = "";

    while (orig.charAt(ind1) == next.charAt(ind1) &&
           ind1 < nextLen && ind1 < origLen)
        ind1++;
    if (ind1 == origLen)
    {
        if (ind1 == nextLen)
            return diff;
        return "start " + ind1 + " end 0%%" +
            next.substring(origLen);
    }
}
```

```

    }
    int ind2 = 0;
    while (orig.charAt(origLen - ind2 - 1) == next.charAt(nextLen -
        ind2 - 1) && ind2 < (origLen - ind1) &&
        ind2 < (nextLen - ind1))
        ind2++;
    return "start" + ind1 + " end " + ind2 + "%%" +
        next.SubString(ind1, nextLen-ind2);
}
}

```

这个 bean 有两个业务方法。第一个就是 `getPage` 方法，这个方法用来加载页面，其行为非常类似于 `PageFetch` bean 的 `getPage` 方法。不同之处在于，它不需要使用参数，相反，使用状态变量来确定目标 URL。

第二个方法是 `getDiff`。这个方法加载页面的当前版本并与老版本的页面进行比较，然后向客户发送两者的区别。为了简单，在这里实现了一个对两者进行比较的简陋方法——就是在两者之间查找最长的公共前缀与最长的公共后缀，并且认为所有的修改都发生在中间部分。当然，这并不是普遍适用的，但对于当中只有一个字段（例如状态）发生改变的页面有用。在任何情况下，这个区别计算方法满足最低的需求，就是能够确保如果没有对页面进行修改，那么区别为空。如果没有首先调用 `getPage` 方法，并且如果在某个时候调用了 `getDiff` 方法，那么该方法就将返回 `null`。

对这个 bean 进行应用程序装配与部署是非常相似的，只是在 `New Session Beans` 对话框的 `General` 标签上应该将这个 bean 指定为有状态。

客户方代码可以通过调用 `getPage` 方法，然后反复调用 `getDiff` 方法来查找是否对这个页面进行了更新，从而使用这个 bean。在接收到这个方法的返回值之后，客户方代码需要对其进行解压并对前缀与后缀长度进行某些简单的分析。

29.3 理解会话 bean 的生命周期

第 11 章讨论了会话 bean 的生命周期。本节将使用第 5 章中的“Hello World” bean 作为例子，来仔细分析会话 bean 的生命周期，并且在各种 EJB 方法中添加一些调试输出语句。然后，将分析一个关于此 bean 的有状态变体，以进一步明确无状态会话 bean 与有状态会话 bean 的生命周期之间的区别。

29.3.1 无状态调试 Hello World bean

这个无状态 bean 的远程接口如下：

```
import javax.ejb.*;
import java.rmi.*;
public interface HelloDbg extends EJBObject
{
    String getTextMessage() throws RemoteException;
}
```

其本地接口如下：

```
import javax.ejb.*;
import java.rmi.*;

public interface HelloDbgHome extends EJBHome
{
    HelloDbg create() throws RemoteException, CreateException;
}
```

这两个接口类似于第5章所讨论的无状态会话 bean。其 bean 实现也很相似，只是在这里插入了一些输出语句，如下所示：

```
import javax.ejb.*;
import java.rmi.*;

public class HelloDbgBean implements SessionBean
{
    // Implementation of "business" method
    public String getTextMessage()
    {
        return "Hello World";
    }

    // EJB methods

    public void ejbCreate()
    {
        System.out.println("ejbCreate");
    }

    public void ejbRemove()
    {
        System.out.println("ejbRemove");
    }

    public void ejbActivate()
    {

```



```
        System.out.println("ejbActivate");
    }

    public void ejbPassivate()
    {
        System.out.println("ejbPassivate");
    }

    public void setSessionContext(SessionContext context)
    {
        System.out.println("setSessionContext");
    }
}
```

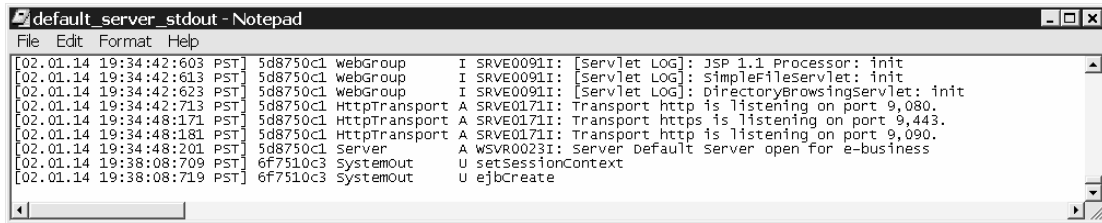
我们将使用下列简单的客户程序来调用这个 bean，并且在末尾使用一个无限循环，从而使程序不会退出。这样，就能够确保资源没有被释放（至少在超时结束之前如此）。

```
import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import javax.naming.*;

public class HelloClient {
    public static void main(String args[])
    {
        try {
            Properties env = System.getProperties();
            env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.websphere.naming.WsnInitialContextFactory");
            env.put(javax.naming.Context.PROVIDER_URL, "iiop://localhost");
            InitialContext ic = new InitialContext(env);
            HelloDbgHome home = (HelloDbgHome)
                javax.rmi.PortableRemoteObject.narrow(
                    ic.lookup("HelloDbgHome"), HelloDbgHome.class);
            HelloDbg bean = home.create();
            System.out.println(bean.getTextMessage());
            for (;;)
            {
                catch (Exception ex) {
                    System.out.println("HelloClient Runtime Error:");
                    ex.printStackTrace();
                }
            }
        }
    }
}
```

在部署这个 bean 并运行此客户程序之后，接着查看 WebSphere 安装目录根下 logs 目录中的

WebSphere 日志文件。如果工作正常（并且客户方代码输出“Hello World”消息），那么 WebSphere 的 stdout 日志应该与下面类似：



```
default_server_stdout - Notepad
File Edit Format Help
[02.01.14 19:34:42:603 PST] 5d8750c1 webgroup I SRVE0091I: [Servlet LOG]: JSP 1.1 Processor: init
[02.01.14 19:34:42:613 PST] 5d8750c1 webgroup I SRVE0091I: [Servlet LOG]: SimpleFileServlet: init
[02.01.14 19:34:42:623 PST] 5d8750c1 webgroup I SRVE0091I: [Servlet LOG]: DirectoryBrowsingServlet: init
[02.01.14 19:34:42:713 PST] 5d8750c1 HttpTransport A SRVE0171I: Transport http is listening on port 9,080.
[02.01.14 19:34:48:171 PST] 5d8750c1 HttpTransport A SRVE0171I: Transport https is listening on port 9,443.
[02.01.14 19:34:48:181 PST] 5d8750c1 HttpTransport A SRVE0171I: Transport http is listening on port 9,090.
[02.01.14 19:34:48:201 PST] 5d8750c1 Server A WSVR0023I: Server Default Server open for e-business
[02.01.14 19:38:08:709 PST] 6f7510c3 Systemout U setSessionContext
[02.01.14 19:38:08:719 PST] 6f7510c3 Systemout U ejbcreate
```

现在，使用不同的窗口反复调用相同的程序。这样就创建了许多都没有死的并发客户。再次查看日志文件，并且将注意到这个文件没有改变。这就意味着，WebSphere 实现了实例池机制。由于这个 bean 是无状态的，因此相同的实例可以为任何数目的客户服务。

29.3.2 有状态调试 Hello World bean

继续使用前一个 bean，并且在其中进行下列修改：在构建时向这个 bean 传递一条消息字符串，并且当调用 getMessageText 方法时返回这个字符串。

这个 bean 的远程接口如下：

```
import javax.ejb.*;
import java.rmi.*;
public interface HelloDbg2 extends EJBObject
{
    String getTextMessage() throws RemoteException;
}
```

它的本地接口如下：

```
import javax.ejb.*;
import java.rmi.*;
public interface HelloDbgHome2 extends EJBHome
{
    HelloDbg2 create(String m) throws RemoteException, CreateException;
}
```

这个 bean 的实现以一种独特的方式输出消息：

```
import javax.ejb.*;
import java.rmi.*;

public class HelloDbgBean2 implements SessionBean
{
```

```
public String msg;

// Implementation of "business" method
public String getTextMessage()
{
    return msg;
}

// EJB methods

public void ejbCreate(String m)
{
    msg = m;
    System.out.println("ejbCreate" + msg);
}

public void ejbRemove()
{
    System.out.println("ejbRemove 2");
}

public void ejbActivate()
{
    System.out.println("ejbActivate 2");
}

public void ejbPassivate()
{
    System.out.println("ejbPassivate 2");
}

public void setSessionContext(SessionContext context)
{
    System.out.println("setSessionContext 2");
}
}
```

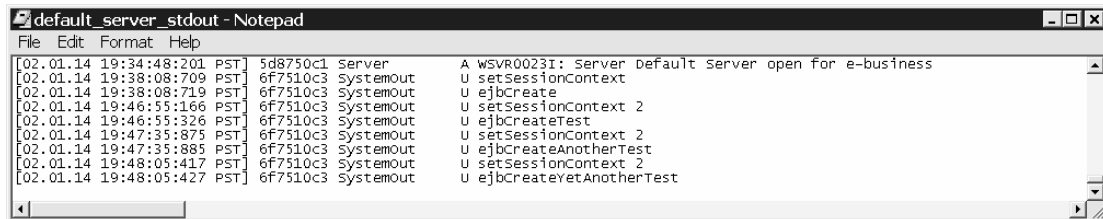
下面给出了测试这个 bean 的客户代码。它从命令行参数得到用来初始化这个 bean 的字符串。

```
import javax.ejb.*;
import java.rmi.*;
import java.util.*;
import javax.naming.*;
```

```
public class HelloClient2 {
    public static void main(String args[])
    {
        try {
            Properties env = System.getProperties();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.websphere.naming.WsnInitialContextFactory");
            env.put(javax.naming.Context.PROVIDER_URL, "iiop://localhost");
            InitialContext ic = new InitialContext(env);
            HelloDbgHome2 home = (HelloDbgHome2)
                javax.rmi.PortableRemoteObject.narrow(
                    ic.lookup("HelloDbgHome2"), HelloDbgHome2.class);
            HelloDbg2 bean = home.create(args[0]);
            System.out.println(bean.getTextMessage());
            for (;;)
        } catch (Exception ex) {
            System.out.println("HelloClient Runtime Error:");
            ex.printStackTrace();
        }
    }
}
```

在编译所有的代码并部署这个 bean 之后，就可以开始重复进行测试了（运行几个客户并检查日志）。

在这里分别使用参数 Test，Another Test 与 Yet Another Test 三次运行客户方程序。在下图中给出了日志的结果。可以注意到，前一次执行的输出仍然显示在日志中，但可以通过发生在 setSessionContext 调用之后的数字 2 以及跟随在 ejbCreate 调用之后的消息字符串来区分新输出。可以在这个日志中见到，每个客户都有自己的 bean 实例，并且容器不能在 3 个 bean 之间共享相同的实例。



29.4 小 结

本章对会话 bean 进行了更详细的讨论，无论是对于一个客户的扩展（有状态 bean），还是

对于由多个客户所使用的支持池管理机制的可重用组件（无状态 bean），会话 bean 都是用来实现这些服务器方代码的重要结构。在两种情况下，客户都通过 RMI 与 bean 进行通信，并且 EJB 容器管理对 bean 的访问。

本章给出了关于无状态会话 bean 与有状态会话 bean 的例子，并且描述了它们在生命周期方面存在的不同以及 WebSphere 如何对它们进行管理。此外，还讨论了在实现会话 bean 时构建的类与方法，并且描述了这些 bean 的装配过程与部署描述符。

会话 bean 常被用来编写服务器方处理代码。会话 bean 通常不代表业务对象，因为它们是链接到会话上（因此链接到它们的名称上）而不是链接到持续性数据。第 30 章将详细讨论实体 bean，实体 bean 通常用来实现持续性业务对象。

第 30 章 实体 bean

第 29 章讨论了 EJB 会话 bean。在原始的 EJB 规范中，另有一部分定义了实体 (entity) bean。实体 bean 不同于会话 bean，在于实体 bean 是持续性对象。持续性对象存储在外部存储器中（例如硬盘或数据库）。本章将讨论什么是实体 bean 以及如何构建、装配与部署实体 bean。重点关注封装在实体 bean 中的数据的持续性。此外，本章还将比较实体 bean 与会话 bean，以帮助读者能够在未来的工程中选择正确的 bean 类型，然后给出了几个关于如何使用实体 bean 的示例。

30.1 持 续 性

对象的持续性是一个重要概念，并且有许多技术可用来实现持续性。持续性的重要性，主要来自这样的事实：大型软件系统通常要求长时间地处理并存储大量的数据。Java 提供了对象串行化机制，用作实现持续性的基本方法。Java 中的串行化 (Serialization) 是 Java 输入/输出机制的扩展，但它允许这些机制支持对象。虽然串行化对于网络上的对象很有用，但对于在数据库中存储对象并不是非常有用。现在，在计算机业界的持续性几乎毫无例外地都是指使用关系数据库，这意味着数据需要被存储在一个表格格式中。有时，对象串行化可能直接将数据存储在文件系统中作为关系数据库的原始数据，或者存储在对象数据库中，但对于大型、任务关键的应用程序，使用这种方法就不是很有用。

实现 Java 持续性的最流行方法（实际上，这也是存储任何类型的数据的最流行方法）就是使用传统的关系数据库，例如 Oracle 或 IBM 的 DB2。但是，在关系数据库中存储对象面临着关于数据表示的一个概念问题，就是：如何将对象数据映射到关系数据库中？在某些情况下，这种映射是简单的，只要将每个类映射到一个表并且将每个成员变量映射到数据库表的列上即可。但对于另一些情况，在对象中可能包含其他对象引用或数组，或者，它们可能需要被映射到多个表；这样，映射就变得很复杂。

在 EJB 体系结构下，实体 bean 提供了对持续性的支持。它们能够使用 BMP (Bean Managed Persistence, Bean 管理的持续性) 实现自己的持续性，或者，它们能够依赖于 EJB 容器使用 CMP (Container Managed Persistence, 容器管理的持续性) 实现持续性。WebSphere 中（甚至 EJB 中）的 CMP 是利用关系数据库实现的。

已经指出，关系数据库是存储大量数据的最常用方法（至少在未来 15 年内构建的应用程序都将如此）。EJB 规范也采用关系数据库模型，这样就能够确保与遗留数据的兼容性。WebSphere EJB 服务器允许用户使用现有的数据库，从而支持这种兼容性。关系数据库已经

被极大地优化,使得这些数据库在处理查询以及更新数据方面的效率高于其他解决方案几个数量级。虽然有时会出现使用其他存储器范型存在某些优点的情况,但关系数据库仍然(并且在未来的很长一段时间内依然如此)在性能、可靠性、灵活性以及业界可接受性方面都是最佳的选择。

EJB 在支持关系数据库的同时,也允许通过使用 BMP 在 EJB 实现中使用任何其他的对象存储技术。例如,可以挂接对象数据库系统与 XML 数据库到 J2EE 应用程序中。对象数据库通过保持基于对象的数据表示,来避免将对象数据映射到数据库上时所出现的相关问题。

为了演示与实体 bean 相关的概念,下面将分析一个 bean,这个 bean 能够 cache 一个 Web 页面的内容。这个例子在实质上与前一章的例子相同,但有一个关键的区别:页面数据是持续性的。这种持续性可用来确保服务器方的页面内容与客户方的页面内容同步。

30.2 会话 bean 与实体 bean

使用实体 bean 访问 Web 页面的方式,与在第 29 章描述的使用会话 bean 访问 Web 页面的方式比较容易混淆。但毕竟,会话 bean 与实体 bean 在概念上还是不同的。而选择最佳的 bean 类型则取决于 EJB 所需解决的应用程序需求。考虑一个更复杂的应用程序,在这个应用程序中,需要 cache 页面的内容以允许在客户方脱机访问 Web 浏览器。换句话说,希望客户下载页面的当前副本(实际上是一系列页面的副本),并且在后来从服务器上更新页面。在这种情况下,服务器就需要了解到客户方所保存的数据的最新拷贝,从而判断哪些页面已经更新。更进一步,这个过程还可能持续较长的一段时间,甚至比一个典型的“会话”存在的时间更长。因此,对于这种应用程序,使用实体 bean 是最佳的选择。

将 bean 实现为实体 bean 的最明显好处就是它的持续性,相应地,这个好处也表现为能够容忍服务器失败。实体 bean 在几个方面不同于会话 bean。其中一个关键的不同在于,对于有状态会话 bean,创建这个 bean 的客户通常就是这个 bean 的惟一用户。而对于实体 bean,一个 bean 可以被某个客户所创建并且被另一个客户所使用。显然,这一功能是非常重要的,它使得实体 bean 可以被用作业务对象,这些业务对象代表存储在数据库中的业务数据。

实体 bean 模型提供了对使用由其他客户所创建的 bean 的内置支持,表现在,可以使用一个 finder 方法来“发现”这个实体 bean。使用这样的方法,就使得查找并访问某个特定的实体 bean 成为可能。为了实现这个功能,实体 bean 有与它们相关联的关键字(关于关键字,将在下一节讨论)。

会话 bean 与实体 bean 之间的另一个有趣不同在于,后者可能被 EJB 体系结构之外的程序所修改。由于持续性通常是通过关系数据库来实现的,因此对存储在数据库中的数据进行修改将隐含的表示,当客户方代码访问这些实体 bean 时它们已经发生了改变。实际上,可以说实体 bean 在关系数据库中提供了一个不同的“视图”。当然,实体 bean 远不止提供“视图”这么简单,因为它们封装了复杂的业务逻辑。

会话 bean 与实体 bean 之间的另一个区别在于它们的生命周期；这一区别主要是由于实体 bean 是持续性的。会话 bean 具有有限的生命跨度，并且在经过一段时间之后，会话 bean 将被析构。相反，实体 bean 有很长的生命跨度，并且只有在远程接口或本地接口显式地调用 remove 方法时，才析构实体 bean。需要说明的是，在另一些方面，实体 bean 的行为类似于会话 bean。例如，与无状态会话 bean 一样，实体 bean 实例也放置在池中；相同的内存中对象可用于几个 bean 对象。

实体 bean 与会话 bean 的 EJB 上下文是不同的。在第 11 章已经知道，EJB 上下文对象将告知 bean 关于周围环境的信息。对于实体 bean，bean 上下文添加了两个新方法，分别是 getEJBObject 与 getPrimaryKey。会话 bean 也存在 getEJBObject 方法，并且它允许 bean 实现获取一个对“远程”接口的引用（对自己的引用），因此，在某种程度上可以将它看作 Java 中 this 对象的远程版本。乍看起来这样并不会有什么好处，但这是必要的，因为 bean 实例被放置在池中并且被重用。无论何时，只要实体 bean 需要查找其主键是什么（特别是在 ejbLoad、ejbStore 与 ejbRemove 方法中），那么就可以使用 getPrimaryKey 来完成这一工作。

30.3 CachedPage bean 设计

CachedPage bean 的一个单独实例代表 HTTP URL 所指定的一个单独的 Web 页面。通过与相关的 HTTP 服务器联系，这个 bean 的用户就能够要求该 bean 检查这个页面是否修改。这样的 bean 对于具有有限的网络带宽（例如移动客户）但又希望连续不断地监视更新数据的客户有用。在这里忽略了这样一个事实，就是相同的 URL 可能基于进行调用的用户而创建不同的页面内容。在实际的应用程序中，内容可能会由于安全性约束以及用户角色而改变。这也可以很容易地使用 EJB 安全性模型以及 run-as 属性来实现，它允许 bean 根据调用者的标识进行动作。在这个例子的其余部分，我们将假设一个 URL 惟一定义所生成的内容，而不管是谁请求这个页面。

这个 bean（以及相应的服务器）负责更新页面，而实际的终端用户则只有在存在更新时才消耗带宽。为了实现这一功能，bean 必须能够存储访问 Web 页面的 URL，以及这个页面的实际内容。这个内容是必要的，因为只有这样，bean 才能够知道是否已经对该页面进行过修改。

为了实现一个实体 bean，就需要实现前面已经描述过的所有组件：远程接口、本地接口与 bean 实现。还有一个以前没有讨论过的新组件，就是关键字，下面将讨论。

30.3.1 CachedPage bean 关键字

当设计实体 bean 时，必须考虑关键字。实体 bean 中的关键字类似于关系数据库中的惟一关键字（惟一关键字就是惟一地定义一个行的一个或多个列集合）。换句话说，在相同的表中

没有两个不同的行具有相同的关键字值。在 EJB 中与此相对应的概念就是惟一地定义 EJB 实例的关键字对象。可以从 EJB 实例获取 EJB 关键字，并且可以使用关键字来搜索该 EJB 实例。

对于这里所讨论的例子，可以将关键字指定为被 cache 的 URL。为了使事情更有趣并且更实际，可以考虑将关键字设置为包括 URL 与一个用户名。这样就允许几个用户访问相同的 Web 页面，而不会出现相互干扰，并且，它对于作为使用 run-as 行为的功能的扩展也非常有用。将用户名与 URL 组合起来也满足了关键字的主要需求，因为这样是惟一的，意味着不存在相同的用户多次 cache 相同页面的可能性。CachedPagePK 类，代表这里所讨论的 CachedPage bean 的关键字，这个类的定义如下：

```
package book.ch30;

public class CachedPagePK implements java.io.Serializable
{
    public String userName;
    public String pageURL;

    public boolean equals(Object obj) {
        if (obj instanceof CachedPagePK) {
            CachedPage pk = (CachedPagePK) obj;
            return (userName.equals(pk.userName) &&
                    pageURL.equals(pk.pageURL));
        }
        return false;
    }
    public int hashCode() {
        String concat = userName + pageURL;
        return concat.hashCode();
    }
}
```

关键字应该实现 Serializable 接口，因为用作 bean 远程接口的方法参数的所有类型以及用作这些方法的返回值的所有类型都必须遵守 Java RMI 约定。这些约定要求上面所描述的类型或者是 Java 原始类型，或者是实现了 java.io.Serializable 或 javax.rmi.Remote 的类型，或者是前述类型的数组。需要说明的是，如果没有遵守这些约定，在程序编译时未必会出现问题，但当相关的方法被调用时就可能会导致出现 RemoteException 异常。此外，WebSphere 对主键的定义还添加了另外一个限制：主键必须覆盖 java.lang.Object 的 equals 方法与 hashCode 方法。在实现 BMP 时，不存在这个限制。

30.3.2 CachedPage 远程接口

为了完成这里所描述的 bean，需要提供一个远程接口、本地接口与 bean 实现。这个 bean 的远程接口如下面的代码所示，它提供了一个标准的类似于 JavaBean get 的方法。为了设置属性，也可以提供相应的 set 方法。这里所讨论的 bean 中没有 set 方法，因为所有的属性或者是关键字的一部分（用户名与 Web 页面），或者是由 bean 内部决定的（例如最后修改日期与页面内容）。不能为构成关键字的属性定义 set 方法，因为如果修改这种属性，将会导致 bean 标识的改变。

```
import java.util.*;
import javax.ejb.*;

public interface CachedPage extends EJBObject
{
    public Boolean    hasChanged()        throws RemoteException;
    public String    getUserName()       throws RemoteException;
    public String    getPageURL()        throws RemoteException;
    public Date      getLastModified()   throws RemoteException;
    public String    getContent()        throws RemoteException;
};
```

30.3.3 CachedPage 本地接口

这个 bean 的本地接口如下：

```
import java.rmi.*;
import java.util.*;
import javax.ejb.*;

public interface CachedPageHome extends EJBHome
{
    public CachedPage create(String userName, String pageURL);

    public CachedPage findByPrimaryKey(CachedPagePK key)
        throws RemoteException, FinderException;
    public Enumeration findByUserName(String userName)
        throws RemoteException, FinderException;
};
```

这个本地接口不同于第 29 章为“Hello World”会话 bean 所提供的那个本地接口，主要是这

里提供了 finder 方法 (findByXXX)。第一个 finder 方法是 findByPrimaryKey。这个方法使用主键作为参数并查找这个 bean 的实例。这个方法的实现由容器负责。这个本地接口中定义的其他 finder 方法是 findByUserName。这个方法演示了如何定义不使用关键字进行查找的 finder 方法。这些方法可能会找到几个满足搜索标准的 bean，并且返回 java.util Enumeration。在本例中，允许使用用户名为一个单独的用户查找所有被 cache 的页面。

对于 BMP，findByPrimaryKey 方法的实现由开发人员负责。对于 CMP，相同方法的实现则由容器负责。显然，容器不能仅仅通过这个方法的签名来实现此 finder 方法。对于 BMP，必须为执行搜索提供实际的代码。而在使用 CMP 时，附加的信息通过 finder helper 接口得到，这个接口本质上为容器提供了查询逻辑。finder helper 不是 JSP 规范的一部分，因此，它是不可移植的。这个例子的 finder helper 类如下：

```
public interface CahcedPageBeanFinderHelper
{
    String findByUserNameWhereClause = "userName = ?";
}
```

使用一个称为<bean-name>BeanFinderHelper 的接口来指定 finder 逻辑，并且，查询逻辑必须包含在一个称为<find-method-name>WhereClause 的字符串常量中。这个字符串包含一个类似于 SQL 查询语句的 Where 短语。在这个 Where 短语中，问号用来代表变量，这些变量将被 finder 方法参数中的实际值所代替（以参数被传递的顺序）。可以在 Where 短语中使用 bean 实现类中的字段。在本例中，使用的是 CachedPage bean 实现类的 userName 字段。需要说明的是，在老版本的 WebSphere 中，finder helper 接口包括了一个完全的 Select 查询；这种方式现在已经被淘汰，因为很难对这种查询进行优化并且它允许用户选择太多或太少的 bean。

利用 Application Assembly Tool，用户可以在部署描述符中提供 Where 短语。

30.3.4 CachedPage bean 实现

关于 bean 实现类的一个重要技术细节，并且是必须遵守的实现细节，就是必须将 bean 中的所有持续性字段（实际上，只有那些其持续性被容器所管理的字段）定义为 public，从而使容器能够访问这些字段。EJB 开发人员也可以定义非持续性字段以及这些字段的可见性，这完全由开发人员自己决定。

```
import java.util.*;
import javax.ejb.*;
import java.net.*;
import java.io.*;
public class CachedPageBean implements EntityBean
{
    public String userName;
```

```
public String pageUrl;
public boolean changeFlag;
public Date lastModified;
public String content;

private void refresh() throws IOException
{
    StringBuffer stringBuffer = new StringBuffer();
    String urlString = pageUrl;
    URL finalUrl;
    URLConnection urlconnection;
    InputStreamReader reader;

    try {
        finalUrl = new URL(urlString);
        urlconnection= finalUrl.openConnection();
        urlconnection.setDoInput(true);
        urlconnection.setRequestProperty("User-Agent",
            "Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)");
        reader = new InputStreamReader
            (urlconnection.getInputStream());
    }
    catch (Exception ex) {
        System.err.println("Cannot connect to URL " + pageUrl +
            " Exception " + ex);

        return;
    }
    int i = urlconnection.getContentLength();
    if (i == -1)
        return;
    int j;
    String s2;
    char[] inbuf = new char[4096];
    for (; (j = reader.read(inbuf)) != -1; ) {
        stringBuffer.append(inbuf, 0, j);
        int k = stringBuffer.length();
        if (i <= 0) {
            continue;
        }
        if (i <= k) {
            break;
        }
    }
    String newContent = stringBuffer.toString();
}
```

```
        if (content == null || !content.equals(newContent)) {
            changeFlag = true;
            content = newContent;
            lastModified = new Date();
        }
    }

    public boolean    hasChanged()
    {
        try {
            refresh();
        }
        catch (Exception ex) {
            System.err.println("Error reading " + pageUrl +
                               " Exception" + ex);
        }
        return changeFlag;
    }

    public String     getUserName()
    {
        return userName;
    }

    public String     getPageURL()
    {
        return pageUrl;
    }

    public Date       getLastModified()
    {
        return lastModified;
    }

    public String     getContent()
    {
        return content;
    }

    public void ejbCreate(String user, String url)
    {
        changeFlag = false;
        lastModified = new Date();
        userName = user;
    }
}
```

```

        pageUrl = url;
    }

    public void setEntityContext(EntityContext c) {}
    public void unsetEntityContext() {}
    public void ejbRemove() {}
    public void ejbPassivate() {}
    public void ejbActivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
};

```

在 bean 实现中包括 `hasChanged` 方法，这个方法将检查并确定某个 Web 页面是否在 HTTP 服务器上发生了改变。这是此例子中这个 bean 的主要“业务”方法，并且它必须为对象封装主要的业务逻辑。在这个 bean 中实现的业务逻辑，使用 `java.net.URLConnection` 对象来获取 Web 页面，并且将这个 Web 页面与该页面的最后一个版本进行比较。如果在内容上没有改变，那么就返回 `false`。否则，页面的新内容被存储起来，并且最后一次修改日期被更新，然后返回 `true`。最后，可以使用 `getContent` 方法访问这个页面的实际内容。

注意，这个 bean 必须实现在本地接口中定义的所有关于 `ejbCreate` 的签名。并且所有这些实现都必须是 `public` 的，而且返回值为 `void`。

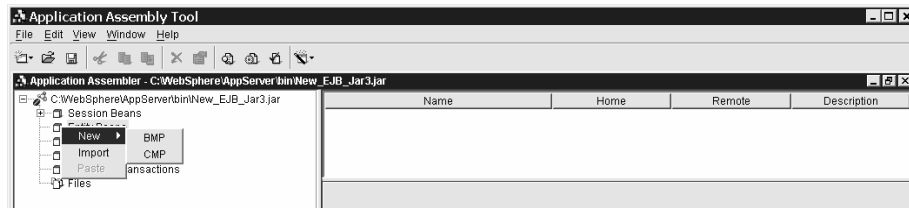
正如在第 11 章解释的那样，容器使用 `ejbCreate`，`ejbActivate`，`ejbPassivate`，`ejbRemove`，`ejbLoad`，`ejbStore`，`ejbPostCreate`，`setEntityContext` 与 `unsetEntityContext` 方法来通知 bean 生命周期相关的事件。这些事件包括创建、激活、钝化、删除、加载、存储与后创建（后创建表示在关键字已经被设置之后进行的初始化）。类似于会话 bean，EJB 服务器能够使用激活与钝化临时地“交换出”实体 bean。需要说明的是，不同于会话 bean，实体 bean 的创建方法是可选的。

在这个特定的例子中，将让所有这些方法为空。另一个要解决的问题就是在 `ejbPostCreate` 方法中自动地加载页面，从而使得对 `hasChanged` 方法的第一次调用能够有某个版本的内容可供与当前页面内容进行比较。如果一个 bean 实现了多个 `create` 方法，那么它必须为每个 `create` 方法实现一个 `post-create` 方法（后创建方法）；`post-create` 方法所使用的参数与 `create` 方法所使用的参数相同。

30.4 CachedPage bean 装配与部署

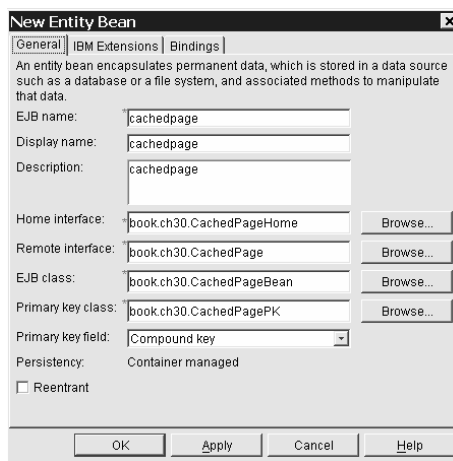
实体 bean 的部署过程类似于会话 bean，如第 29 章所述。其中最主要的区别是，实体 bean 的部署描述符包含一些不同的信息。

在 AAT 中创建了新 EJB 模块之后，就可以右击左边面板的 Entity Beans 表项，添加新实体 EJB。不同于会话 bean，在这里有两个选择（BMP 或 CMP），如下图所示：

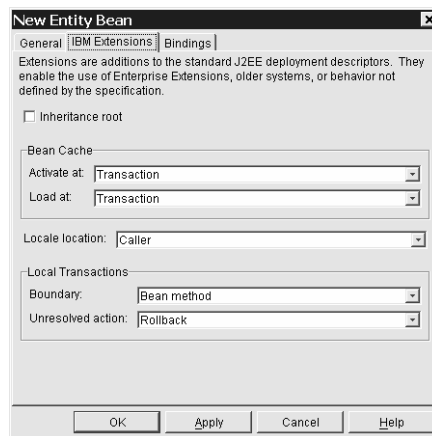


这时，请选择 CMP（BMP 将在下一节讨论）。

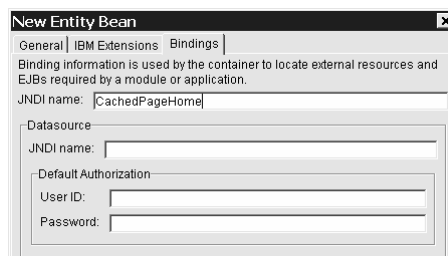
类似于用于添加新会话 bean 的对话框，添加新实体 bean 的对话框也有几个标签。其中，General 标签包括了用于指定主键类与关键字字段（如果它是一个单独的字段）的域。在所讨论的例子中，关键字不是单独的字段，因此 Primary key field 表项显示 Compound key。



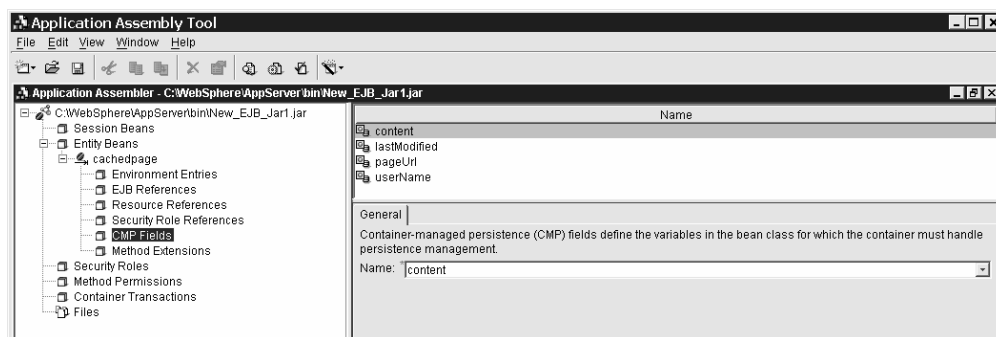
IBM Extensions 标签与会话 bean 的相应标签相同，但在这里，Timeout 字段被删除并且可以配置 cache 选项。关于 IBM EJB 扩展，将在第 33 章详细讨论。



Bindings 标签中包括了 JNDI 名与用于存储 bean 的数据源。



为了告知容器哪些字段应该是持续性的,就需要在 AAT 中右击 CMP Fields 表项并选择 New, 从而将这些字段指定为 CMP 字段。这时,将打开 New CMP Field 对话框,在这个对话框中可以选择应该被设置为持续性的字段。在选择其持续性由容器所管理的字段之后, CMP Fields 表项将列出所有这些字段,如下图所示:



在 EJB 装配描述符中,可以很直观地看到在 AAT 中对 bean 进行的设置:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN"
"http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar id="ejb-jar_ID">
  <description>cachedpage</description>
  <display-name>cachedpage</display-name>
  <enterprise-beans>
    <entity id="ContainerManagedEntity_1">
      <description>cachedpage</description>
      <display-name>cachedpage</display-name>
      <ejb-name>cachedpage</ejb-name>
      <home>book.ch30.CachedPageHome</home>
      <remote>book.ch30.CachedPage</remote>
```



```
<ejb-class>book.ch30.CachedPageBean</ejb-class>
<persistence-type>Container</persistence-type>
<prim-key-class>book.ch30.CachedPagePK</prim-key-class>
<reentrant>False</reentrant>
<cmp-field id="cachedpage_userName">
  <field-name>userName</field-name>
</cmp-field>
<cmp-field id="cachedpage_lastModified">
  <field-name>lastModified</field-name>
</cmp-field>
<cmp-field id="cachedpage_pageUrl">
  <field-name>pageUrl</field-name>
</cmp-field>
<cmp-field id="cachedpage_content">
  <field-name>content</field-name>
</cmp-field>
</entity>
</enterprise-beans>
<assembly-descriptor id="AssemblyDescriptor_1">
</assembly-descriptor>
</ejb-jar>
```

在装配阶段完成之后，就可以进行 bean 的部署。bean 的部署过程类似于其他 EJB 模块的部署过程。惟一的不同在于绑定与数据库设置。如前所述，这里的绑定除了包括 JNDI 名之外，也包括数据源属性，在部署时可以对绑定设置进行修改。也可以选择存储 bean 的数据的数据库类型。

在 bean 被部署之后，就没有必要执行任何数据库设置操作。WebSphere 将自动创建一个名为 ejb.beanNamebeantbl 的数据库表，其中，beanName 就是 bean 名。因此，对于这里所描述的例子，数据库表的名称就是 ejb.book.ch30.CachedPagebeantbl。AAT 生成用于创建这个表的 SQL 代码，并且在部署时将执行这些 SQL 代码。

需要记住的是，WebSphere 4.0 不会自动创建这些表。它仅仅创建 DDL (Data Definition Language, 数据定义语言) 文件，这个文件特定于所选择的 DB (数据库) 类型，这样就使得用户能够创建这个表。

30.5 BMP 示例

为了演示 BMP，现在考虑将 CachedPage bean 修改为由 bean 所管理。这样做的典型目的就是需要使用非关系数据存储，或者，需要使用特别设计的机制（例如，优化的查询与存储过程）来改善性能。

为了迁移到 BMP，不需要修改远程接口与本地接口；只要修改 bean 实现。这样做是很合理

的，因为 bean 的用户不需要关心 bean 的内部实现细节（包括持续性实现细节）。同样需要说明的是，在这里不需要 finder helper 类。

EJB 规范并没有对如何实现 BMP 进行限制。可以使用 ASCII 文件、Java 对象串行化、遗留事务系统或任何其他存储机制来实现 BMP（这完全由自己决定），只要它有用 Java 编写的实现或某个接口即可。

在本例中，将考虑使用 JDBC 连接到关系数据库的情况，这是基于 Java 的持续性的最典型情况。下面给出了面向 BMP 的 bean 实现代码：

```
import java.util.*;
import javax.ejb.*;
import java.sql.*;
import javax.sql.*;
import java.net.*;
import java.io.*;

public class CachedPageBean implements EntityBean
{
    public String userName;
    public String pageUrl;
    public boolean changeFlag;
    public java.util.Date lastModified;
    public String content;
    transient protected EntityContext context;

    private void refresh() throws IOException
    {
        StringBuffer stringBuffer = new StringBuffer();
        String urlString = pageUrl;
        URL finalUrl;
        URLConnection urlconnection;
        InputStreamReader reader;

        try {
            finalUrl = new URL(urlString);
            urlconnection= finalUrl.openConnection();
            urlconnection.setDoInput(true);
            urlconnection.setRequestProperty("User-Agent",
            "Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)");
            reader = new InputStreamReader
                (urlconnection.getInputStream());
        }
        catch (Exception ex) {
            System.err.println("Cannot connect to URL " +
```

```
        pageUrl + " Exception " + ex);
        return;
    }
    int i = urlconnection.getContentLength();
    if (i == -1)
        return;
    int j;
    String s2;
    char[] inbuf = new char[4096];
    for (; (j = reader.read(inbuf)) != -1; ) {
        stringBuffer.append(inbuf, 0, j);
        int k = stringBuffer.length();
        if (i <= 0) {
            continue;
        }
        if (i <= k) {
            break;
        }
    }
    String newContent = stringBuffer.toString();
    if (content == null || !content.equals(newContent)) {
        changeFlag = true;
        content = newContent;
        lastModified = new java.util.Date();
    }
}

public boolean    hasChanged()
{
    try {
        refresh();
    }
    catch (Exception ex) {
        System.err.println("Error reading " + pageUrl +
            " Exception " + ex);
    }
    return changeFlag;
}

public String    getUserName()
{
    return userName;
}
```

```
public String    getPageURL()
{
    return pageUrl;
}

public java.util.Date    getLastModified()
{
    return lastModified;
}

public String    getContent()
{
    return content;
}

public void ejbLoad()
{
    Connection con = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {
        CachedPagePK key = (CachedPagePK) context.getPrimaryKey();
        userName = key.userName;
        pageUrl = key.pageURL;

        String query = "select * from CACHEDPAGES set " +
            "where USER = '" + userName + "', '" +
            "PAGE = '" + pageUrl + "'";
        con = DriverManager.getConnection(
            "jdbc:db2:myDb:5001",
            "db2admin",
            "db2passwd");
        stmt = con.prepareStatement();
        rs = stmt.executeQuery(query);
        boolean more = rs.next();
        if (more) {
            changeFlag = true;
            content = rs.getString(3);
            lastModified = new java.util.Date(rs.getString(4));
        }
    }
    catch (Exception ex) {
        System.err.println("Error in ejbLoad " + pageUrl +
            " Exception " + ex);
    }
}
```

```
    }
    finally {
        rs.close();
        stmt.close();
        con.close();
    }
}

public void ejbStore()
{
    Connection con = null;
    PreparedStatement stmt = null;
    try {
        String query = "update CACHEDPAGES set " +
            "CONTENT = " + content + ",MODIFIEDDATE = " +
            lastModified + "where USER = '" + userName + "'," +
            "PAGE = '" + pageUrl + "'";
        con = DriverManager.getConnection(
            "jdbc:db2:myDb:5001",
            "db2admin",
            "db2passwd");
        stmt = con.prepareStatement();
        int count = stmt.executeUpdate(query);
    }
    catch (Exception ex) {
        System.err.println("Error in ejbStore " + pageUrl +
            " Exception " + ex);
    }
    finally {
        stmt.close();
        con.close();
    }
}

public void ejbRemove()
{
    Connection con = null;
    PreparedStatement stmt = null;
    try {
        String query = "delete from CACHEDPAGES where USER = '" +
            userName + "'," + "PAGE = '" + pageUrl +
            "'";
        con = DriverManager.getConnection(
```

```
        "jdbc:db2:myDb:5001",
        "db2admin",
        "db2passwd");
    stmt = con.prepareStatement();
    int count = stmt.executeUpdate(query);
}
catch (Exception ex) {
    System.err.println("Error in removing " + pageUrl +
        " Exception " + ex);
}
finally {
    stmt.close();
    con.close();
}
}

public void setEntityContext(EntityContext ctx)
{
    context = ctx;
    try {
        Class.forName("com.ibm.db2.jdbc.app.DBDriver");
    }
    catch (Exception ex) {
    }
}

public void unsetEntityContext()
{
    context = null;
}

public void ejbPostCreate(String user, String url)
{
}

public CachedPagePK ejbCreate(String user, String url)
{
    userName = user;
    pageUrl = url;
    CachedPagePK key = new CachedPagePK();
    key.userName = userName;
    key.pageURL = pageUrl;
    changeFlag = false;
    lastModified = new java.util.Date();
}
```

```

    try {
        refresh();
    }
    catch (Exception ex) {
        System.err.println("Error in reading " + pageUrl +
            " Exception " + ex);
    }

    String query = "insert into CACHEDPAGES (USER, PAGE, " +
        "CONTENT, LASTMODIFIED)" +
        "values ('" + userName + "', '" +
        pageUrl + "', '" + content + "', '" +
        lastModified + "')";

    Connection con = null;
    PreparedStatement stmt = null;
    try {
        con = DriverManager.getConnection(
            "jdbc:db2:myDb:5001",
            "db2admin",
            "db2passwd");
        stmt = con.prepareStatement();
        int count = stmt.executeUpdate(query);
    }
    catch (Exception ex) {
        System.err.println("Error in creating " + pageUrl +
            " Exception " + ex);
    }
    finally {
        stmt.close();
        con.close();
    }
    return key;
}

public CachedPagePK ejbFindByPrimaryKey(CachedPagePK key) throws
    FinderException
{
    String name = key.userName;
    String url = key.pageURL;

    String query = "select * from CACHEDPAGES set " +
        "where NAME = '" + name + "', '" +
        "PAGE = '" + url + "'";

```

```
Connection con = null;
PreparedStatement stmt = null;
ResultSet rs = null;
try {
    con = DriverManager.getConnection(
        "jdbc:db2:myDb:5001",
        "db2admin",
        "db2passwd");
    stmt = con.prepareStatement();
    rs = stmt.executeQuery(query);
    boolean more = rs.next();
    boolean toomany = false;
    if (more)
        toomany = rs.next();
    if (more)
    {
        if (toomany) {
            throw new FinderException("Too many rows");
        }
        else {
            return key;
        }
    }
    else {
        throw new FinderException("Not found");
    }
}
catch (Exception ex) {
    throw new FinderException("Exception Caught: " + ex);
}
finally {
    rs.close();
    stmt.close();
    con.close();
}

}

public Enumeration ejbFindByUserName(String name) throws
    FinderException
{
    Vector v;
    Connection con = null;
    PreparedStatement stmt = null;
    ResultSet rs = null;
```



```

    try {
        String query = "select * from CACHEDPAGES set " +
            "where USER = '" + name + "'";
        con = DriverManager.getConnection(
            "jdbc:db2:myDb:5001",
            "db2admin",
            "db2passwd");
        stmt = con.prepareStatement();
        rs = stmt.executeQuery(query);
        v = new Vector();
        CachedPagePK key;
        boolean more = rs.next();

        while (more) {
            key = new CachedPagePK();
            key.userName = name;
            key.pageURL = rs.getString(1);
            v.addElement(key);
            more = rs.next();
        }
    }
    catch (Exception ex) {
        throw new FinderException("Exception Caught: " + ex);
    }
    finally {
        rs.close();
        stmt.close();
        con.close();
    }
    return v.elements();
}

public void ejbPassivate() {}
public void ejbActivate() {}
}

```

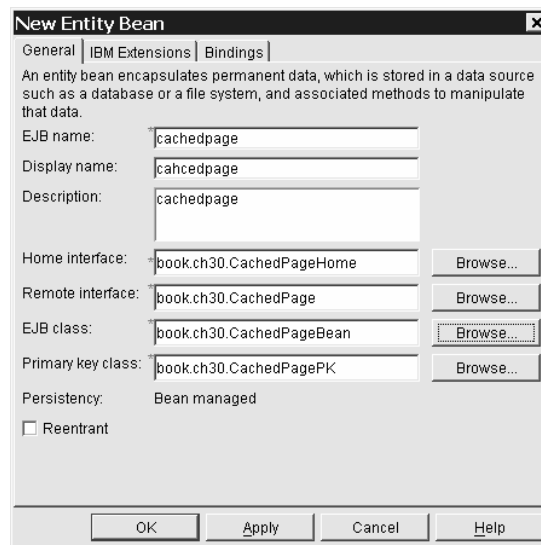
需要为 `ejbLoad`, `ejbStore`, `ejbRemove` 与 `ejbCreate` 方法建立 JDBC 连接。这些连接在 `finder` 方法中也是需要的, 这个方法用来在数据库中查找行。如果没有找到任何内容, 应该从 `findByPrimaryKey` 方法中发出 `FinderException` 异常。`findByUserName` 方法不会发出此异常。这似乎有点混淆; 之所以在 `findByPrimaryKey` 方法中能够发出此异常, 是因为在调用该方法之后, 总是希望能够从这个方法中返回刚好一个对象 (远程引用)。相反, `findByUserName` 方法则返回 `Java Enumeration` 对象, 而这个枚举值可以为空。如果没有任何 bean 实例与方法的参数相匹配, 那么所返回的枚举值的确为空。

在这个例子中实现数据库连接的方法似乎有点古怪。一般来说，有 4 种方法可解决这个问题，分别是：

- 在 `setEntityContext` 中建立数据库连接，并在 `unsetEntityContext` 中释放这个数据库连接。这种方法最容易实现，但仅仅对于数据库连接池（请参见第 34 章）有用。
- 可以在 `ejbActivate` 与 `ejbCreate` 方法中建立数据库连接，在 `ejbFind` 方法中建立与释放数据库连接，在 `ejbRemove` 与 `ejbPassivate` 方法中释放数据库连接。这个方法能够提高性能，因为只有活跃的 bean 有数据库连接，但这个方法的实现比较困难。
- 最麻烦的方法，也是在前面一个例子中所使用的方法，就是在每个需要数据库连接的方法中建立然后释放数据库连接。这样，就能够确保只有当需要时才打开数据库连接，这能够使得代码更加高效并且更加具有可伸缩性。
- 最好的方法就是在 `ejbActivate`，`ejbCreate` 等方法中要求获得一个对数据源（使用 JNDI）的引用，然后为每个方法获取并释放数据库连接。

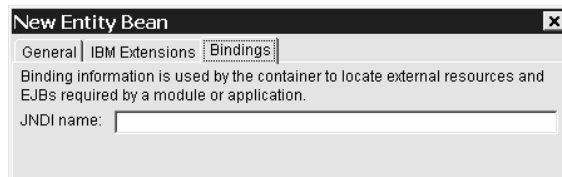
为了使一个单独操作的数据库连接开销最小化，可以通过重用已有的连接池对数据库连接进行池操作。需要说明的是，EJB 服务器自己实现了线程池机制以及 EJB 实例池机制，从而提高 EJB 的性能。第 18 章说明了 WebSphere 的连接池，第 34 章将描述带有连接池的 BMP 的实现。

现在返回到应用程序装配阶段。当向 EJB 模块中添加一个新 BMP 实体 bean 时，所使用的参数与 CMP 所使用的参数有所不同。General 标签将指出持续性是由 bean 管理的，并且不包括 Primary Key Field 表项。



IBM Extensions 标签（在这里没有显示出来）没有改变；在 Bindings 标签中没有数据源规范

字段，因此它实际上与会话 bean 相同。



在使用 BMP 时，bean 实现中存在的区别是非常明显的：需要进行更多的编码。只有在有特殊的持续性需求时，才使用 BMP。

30.6 小 结

实体 bean 是任何使用 EJB 的应用程序体系结构的核心。实体 bean 代表业务对象，并且允许构建应用程序，在这个应用程序中，数据访问与业务功能都封装在一个面向对象的框架中。然后，就可以使用面向对象的抽象来编写应用程序。

实体 bean 可以使用容器管理的持续性，在这种情况下，可以延迟数据映射的执行，并且利用容器来处理数据持续性问题。在另一些情况下，用户可能希望自己管理持续性，这在某些情况下可能会达到更高效的数据库访问目的。无论对于哪种情况，都应该隐藏关于数据持续性实现的细节，并且在一个高级的抽象层次工作，在这种层次上，对象可以被其他组件非常容易地访问与使用，这些组件通常是客户端应用程序的会话 bean 扩展。

数据库访问要求进行查询以获取数据，并要求进行插入与更新以存储数据。插入与更新意味着需要定义事务以及原子性事务操作，以支持 ACID（原子性、一致性、隔离性、持久性）属性。在这里还没有讨论事务属性并且只是简单地提到了 EJB 安全性的 run-as 功能。这些都是第 31 章讨论的话题。

第 31 章 EJB 事务与安全性

第 29 章与第 30 章已经对 EJB 体系结构中的两个核心问题进行了简单说明——事务与安全性。实际上，它们对于企业应用程序的设置具有非常重要的作用。

EJB 的一个基本点，就是它是一个服务器方应用程序框架，这个框架能够帮助开发人员在构建复杂服务器方代码的过程中避免一些非常头痛的问题。相反，开发人员能够使用伪指令（directive），依赖于容器来处理大量复杂代码编写工作。如果在容器管理的 EJB 框架下工作，bean 开发人员就不需要考虑关于事务与安全性的复杂问题。

EJB 体系结构提供了非常灵活与强大的解决方案，用来映射 EJB 的事务行为与安全性功能。与这些问题相关的大多数属性都能够在装配时（或部署时）设置，这样就将业务逻辑以及 bean 实现与这些问题相分离。当然，EJB 体系结构也允许有兴趣的开发人员实现自己的事务管理策略或安全性管理策略。

31.1 Java 中的事务

事务在业务应用程序中扮演重要的角色。可以考虑资金从一个帐号转移到另一个帐号的情况。简单地说，这一动作包括从一个帐号取出资金，然后将这些资金放置到另一个帐号上。显然，这样的动作应该原子性地执行，也就是说，这个动作必须完全完成，或者完全没有完成。如果允许这种动作被部分地完成，那么就破坏了数据的一致性；在这种情况下，资金可能会“突然不见了”。

事务是一系列发生在共享资源上的操作，这些操作被归组成一个单独的工作单元并且维护了 ACID 属性。缩略语 ACID 代表 Atomicity（原子性）、Consistency（一致性）、Isolation（隔离性）与 Durability（持久性），它们是事务必须具有的属性。

- Atomicity（原子性）：作为事务的一部分的动作被原子性地看待，表现在或者所有的动作都被执行，或者没有任何动作被执行。如果出现了失败，在这个事务过程中执行的修改能够完全撤消。
- Consistency（一致性）：如果出现失败，系统状态将返回到事务之前的原始状态。
- Isolation（隔离性）：一个事务所做的任何修改在这个事务完成之前对事务系统中的任何其他用户都是不可见的。
- Durability（持久性）：一旦事务完成，该事务的影响将持久存在；特别地，它们能够容忍系统失败。

注意：关于事务理论的完全讨论，不是本书关注的内容。关于这方面的更多信息，读者可以参见“Databases and Transaction Processing, by Lewis, Bernstein, and Kifer (Addison Wesley, 2001)”。

在分布式系统下，可能出现失败的机会更大，特别是可能影响到动作的一部分的本地失败。例如，在执行资金转移操作时，可能有三方参与：资金转移的发起者、源帐号、目标帐号。这三方中的每一方都可能独立于其他二方而出现失败。有可能接收帐号在线，但由于某种原因使得源帐号不可用。在这种情况下，执行资金转移将导致不可预测的后果（例如，创建新资金）。

分布式事务处理系统在 IT 领域已经存在几十年的时间。一些专利的解决方案（例如 IBM 的 CICS (Customer Information Control System, 客户信息控制系统)）直到 20 世纪 80 年代都一直占据市场统治地位，后来，这一趋势发展为某些组织（例如 X/Open 组）提供的基于 UNIX 的开放式解决方案，它们最后成为事务的标准草案。但这并不是说那些专利系统就失去了市场份额，实际上，像 CICS 这样的产品目前仍占据了非常重要的市场份额。

作为关于事务标准化努力的一部分，OMG (Object Management Group, 对象管理组) 为分布式事务定义了一个面向对象的框架，这个框架被引入到 CORBA 中。实际上，这个框架就是本书第 8 章简单描述过的 OTS (对象事务服务)。

在将事务引入到 Java 编程语言中时，关于事务管理的问题被分为两个独立的部分。第一部分就是 JTS (Java Transaction Service, Java 事务服务)。JTS 通过在第 8 章中详细描述的 IDL Java 映射提供了 OTS 功能。另一个部分就是 JTA (Java 事务 API)。

JTA 为映射事务提供了一个更高层次的 API，它包含在 `javax.transaction` 包中。这个包为管理事务提供了几个接口。`javax.transaction.UserTransaction` 包将一个事务初始化为方法，从而提供可供使用的基本功能原语，这些方法如下：

- `begin`：创建一个新事务，并且将这个事务与实际的线程相关联。
- `commit`：完成与当前线程相关联的事务。
- `getStatus`：返回与当前线程相关联的事务的状态（可能的状态列表是 `javax.transaction.Status` 接口的一部分）。
- `rollback`：回滚与当前线程相关联的事务。
- `setRollbackOnly`：使与当前线程相关联的事务成为一个只能被回滚的事务。
- `setTransactionTimeout`：为与当前线程相关联的事务设置一个超时（单位为秒）。在过了这个时间之后，事务将自动回滚。

对于大多数应用目标，应用程序级代码并不关心 JTS 以及事务管理的实现。实际上，应用程序级代码仅仅对与它相关的事务的处理过程感兴趣；`UserTransaction` 接口可用来完成这个目标。

`UserTransaction` 的对象通常是通过 JNDI 查找获得的。例如，在 WebSphere 环境下，可以使

用 JNDI 名 `jta/usertransaction` 来访问服务器的事务服务。在获取事务上下文之后，客户代码就能够使用 `begin` 方法初始化事务，然后就可以在事务范围内执行动作。最后，将调用 `commit` 方法来完成该事务。下面给出了一个例子：

```
import javax.transaction.*;
class ClientCode {
    // ...
    void transactionMethod() {
        // ...
        Context initialContext = new InitialContext();
        UserTransaction tcon =
            (UserTransaction) initialContext.lookup(
                "jta/usertransaction");

        tcon.begin();
        // Use resources within the transaction scope
        tcon.commit();
    }
}
```

显然，通过编程的方法来管理事务是比较麻烦的。EJB 规范允许通过声明的方法来管理事务，也允许用户自己编写代码来管理事务，完全取决于用户个人的需要。

31.2 启用事务支持

与 EJB 持续性相似，EJB 的事务行为可以由容器管理，也可由开发人员管理。EJB 规范在定义 bean 的事务行为的 EJB 部署描述符中定义了两个部署属性。正如在第 11 章讨论的那样，它们分别是 `transaction` 属性与 `isolation` 属性。前者用来告知容器应该将哪个事务行为应用于方法调用上。这可以为整个 bean 定义，也可以为独立的方法定义。`isolation` 属性用来说明事务在容器中相互之间如何区分，同样，可以为整个 bean 以及为单独的方法定义这个属性。从第 11 章可以知道，可能的事务设置是 `TX_NOT_SUPPORTED`，`TX_BEAN_MANAGED`，`TX_REQUIRED`，`TX_SUPPORTS`，`TX_REQUIRES_NEW` 与 `TX_MANDATORY`。

当为实体 bean 使用 CMP 时，就不能用 `TX_NOT_SUPPORTED`，因为容器要求事务是持续性数据。

这两个事务属性都可以在部署时为整个 bean 或为单个的方法定义。惟一的例外就是，`TX_BEAN_MANAGED` 规范不能用于单个的方法，而只能用于整个 bean。当使用 `TX_BEAN_MANAGED` 设置时，这个 bean 就能够分别从 `setSessionContext` 方法或 `setEntityContext` 方法中提供的 `SessionContext` 或 `EntityContext` 对象中获取事务。这个动作可以利用实体上下文的 `getUserTransaction` 方法来完成。这个方法的返回值是

`javax.transaction.UserTransaction`，用于事务的划分。需要说明的是，除非被指定为 `TX_BEAN_MANAGED`，否则 `bean` 就不应该直接访问事务服务。

类似于事务属性，在第 11 章也详细讨论了隔离等级。可能的隔离等级值包括 `TRANSACTION_READ_UNCOMMITTED`，`TRANSACTION_READ_COMMITTED`，`TRANSACTION_REPEATABLE_READ`，`TRANSACTION_SERIALIZABLE` 与 `TRANSACTION_NONE`。隔离等级可用来控制事务相互之间在查看数据库方面隔离的方式。如果在这个过程中没有特别谨慎，那么当有几个事务访问数据库时，就可能出现三个常见的问题：脏读、不可重复读与虚读。

当一个事务从已经被另一个事务所写但还没有确认的数据库数据中读数据时，就发生了脏读（dirty read）。如果后一个事务回滚，前一个事务实际上使用的是错误数据。如果事务从数据库中读取数据，然后，在后来的某时刻，它读取相同数据但这时数据已发生修改，在这种情况下，就出现了不可重复读（unrepeatable read）。之所以会发生这种情况，可能是由于另一个事务对该数据进行了动作。当一个事务查询数据库并获取某个数据集合时，另一个事务对数据库进行更新，并且在后来的某个时刻，前一个事务又执行相同的查询并获得新的记录（就是虚幻记录），这时，就发生了一个虚读（phantom read）。

更加“松弛的”EJB 隔离级别就是 `TRANSACTION_READ_UNCOMMITTED`；这一级别没有施加任何隔离保证，但它具有最好的性能。使用这个隔离级别将使系统更加容易出现上面所描述的三种问题（脏读、不可重复读与虚读）。`TRANSACTION_READ_COMMITTED` 隔离级别能够解决脏读问题，并且保证不会发生脏读。`TRANSACTION_REPEATABLE_READ` 隔离级别能够阻止脏读与不可重复读这两个问题发生。`TRANSACTION_SERIALIZABLE` 隔离级别提供了最佳的保证，在这个隔离级别下，能够确保不发生脏读、不可重复读与虚读。

由于隔离与数据库访问相关，当它应用于 CMP `bean` 时所得到的效果有所不同。对于 BMP 会话 `bean` 或实体 `bean`，容器在事务开始时为 `bean` 所使用的每个数据库连接设置隔离级别。对于 CMP `bean`，持续化对象按照遵守特定隔离级别的方式实现数据库访问代码。

对隔离级别的选择，首先而且最重要的是取决于底层数据库支持的隔离级别。否则，在性能与数据一致性风险之间就没有明确的折中。这种折中来自这样一个简单的事实：对某个数据记录上锁的时间越长，出现数据一致性问题的风险就越小，但与此同时整体性能将降低（由于并发的事务被强制等待）。在极端情况下，甚至可能导致死锁。`Serializable` 隔离级别是最严格也是最安全的，但在某些情况下并不需要如此。例如，不执行更新而仅仅读数据的代码，就可以使用 `ReadUncommitted` 隔离级别。

底层数据库支持什么样的隔离级别并没有任何理论依据，因为并非所有的数据库都支持所有的隔离级别。例如，Oracle 仅仅支持 `ReadCommitted` 与 `Serializable` 隔离级别。这意味着必须在这两种情况之间进行选择：或者让系统存在虚读以及不可重复读这种危险，或者让系统的性能较低。

31.2.1 利用 AAT 控制 EJB 事务

WebSphere AAT 为设置 EJB 的事务行为提供了一个方便的方法。在会话 bean 的 General 标签上，可以设置事务的行为是由 bean 管理还是由容器管理，如图 31-1 所示。开发工具（例如第 16 章描述的 WSAD（WebSphere Studio Application Developer））也可以用来设置 EJB 的事务行为。这一选择对实体 bean 是不需要的，因为容器为这种 bean 管理事务。

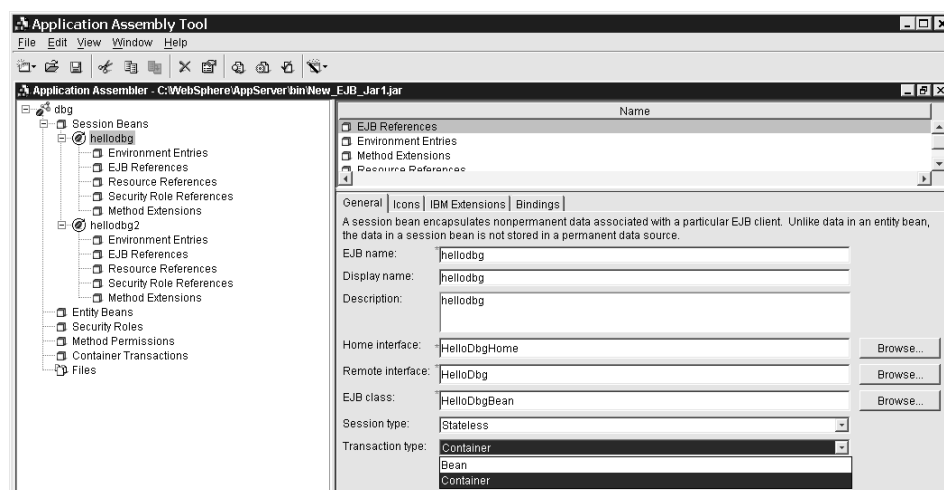
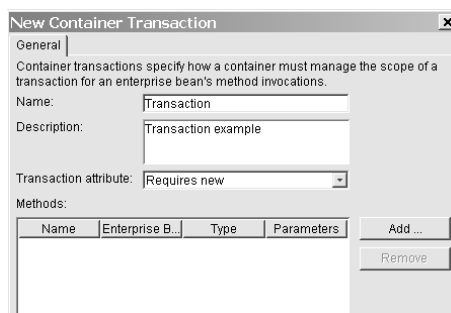
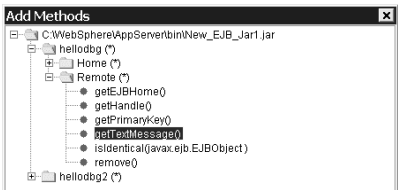


图 31-1 使用 AAT 设置事务行为

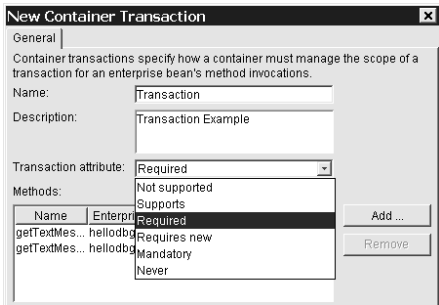
当使用容器管理的事务时，事务属性是通过 EJB 模块的 Container Transaction 表项来设置的。如果需要为一个或多个方法定义事务属性，应该右击这个实体并选择 New。这时，将打开如下的对话框：



单击 Add 按钮之后，将打开一个新窗口，在这个窗口中可以选择模块中任何 bean 的 EJB 方法，而且这个模块使用了容器管理的事务。



在这里可以选择所需的任何数目的方法，并且为这些方法选择相应的事务属性，如下图所示：



在每个 EJB 的 Method Extensions 表项下可以选择隔离级别属性，如图 31-2 所示。在这里，可以选择任何方法并设置这个方法的隔离级别。对于某个接口的所有方法，或者对于与这个 bean 相关的所有方法，可以包括通配符。

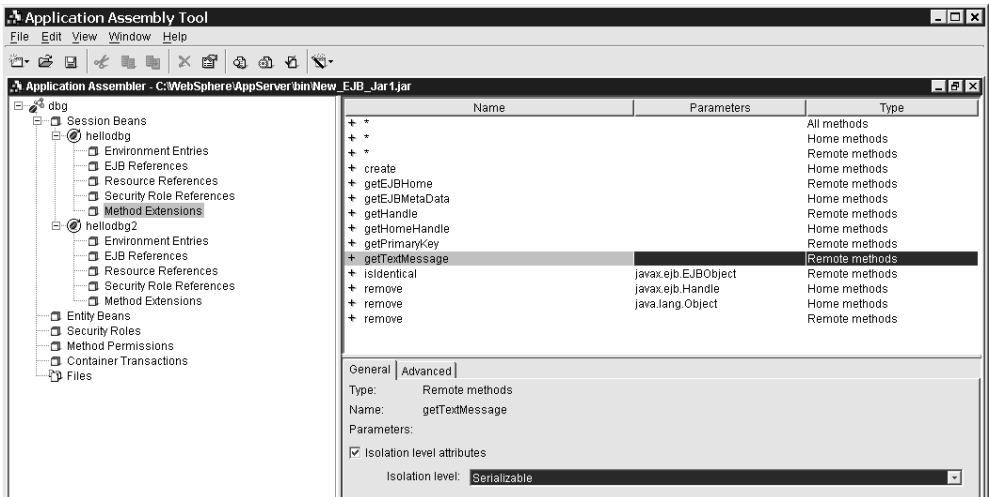


图 31-2 使用 AAT 设置隔离级别

在上面所进行的任何设置都将反映到部署描述符中。例如，在为第 29 章介绍的 bean 中的 2

个 `getTextMessage` 方法添加了事务 `Required` 属性之后,所得到的 XML 部署描述符将如下所示 (与事务相关的表项用黑体表示):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar id="ejb-jar_ID">
  <display-name>dbg</display-name>
  <enterprise-beans>
    <session id="Session_1">
      <description>hellodbg</description>
      <display-name>hellodbg</display-name>
      <ejb-name>hellodbg</ejb-name>
      <home>HelloDbgHome</home>
      <remote>HelloDbg</remote>
      <ejb-class>HelloDbgBean</ejb-class>
      <session-type>eStateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
    <session id="Session_2">
      <description>hellodbg2</description>
      <display-name>hellodbg2</display-name>
      <ejb-name>hellodbg2</ejb-name>
      <home>HelloDbgHome2</home>
      <remote>HelloDbg2</remote>
      <ejb-class>HelloDbgBean2</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  <assembly-descriptor id="AssemblyDescriptor_1">
    <container-transaction id="MethodTransaction_1">
      <description>Transaction:+:Example Transaction</description>
      <method id="MethodElement_1">
        <ejb-name>hellodbg</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getPrimaryKey</method-name>
        <method-params>
        </method-params>
      </method>
      <method id="MethodElement_2">
        <ejb-name>hellodbg2</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getTextMessage</method-name>
      </method>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

```
        <method-params>
        </method-params>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

31.2.2 EJB 事务的局限性

虽然事务非常重要,但必须意识到的重要一点是,它们并没有包括 EJB 中所有的实例变量。在一个有状态会话 bean 中,bean 状态就不是事务性的。换句话说,如果一个事务在有状态 EJB 中开始,并且后来又执行了一个回滚操作,那么 EJB 状态将不会改变。必须在事务开始之前,显式地修改状态以反映状态的变化。正如在第 11 章说明的那样,可以利用 SessionSynchronization 接口告知会话 bean 与其相关的事务状态。

在实体 bean 中,bean 的状态是事务性的,但这也仅仅是对持续性字段(由 CMP 管理或 BMP 管理)如此。非持续性变量不存储在数据库中。并且,这些变量不能用于维护方法调用之间 EJB 客户相关联的状态,因为这些值可能由于另一个客户的调用而修改,或者当 bean 被钝化时这个值丢失。

一个非常重要的限制就是,利用任何来自容器的事务性支持所声明的 EJB,都不能使用 JDBC 确认与回滚操作。这些操作预留给容器使用。

当运行于容器管理的事务中时,如果出现了异常,容器可能会回滚事务。如果一个系统异常被捕获,那么在异常被传递给客户之前,将进行事务的回滚。但是,如果一个应用程序异常被捕获,容器将假设这个事务仍然有效。在这种情况下,事务没有被回滚。实际上,如果它是一个新事务,并且这个事务的范围就是发出这个异常的方法调用的范围,那么这个事务将被确认。当然,如果应用程序显式地为这个事务调用了 `setRollbackOnly`,那么该事务将被回滚。

需要说明的是,一个没有被捕获的应用程序异常可能会导致回滚。考虑这样一种情况:某个 bean 方法声明了它将发出一个应用程序异常,并且它的事务属性为 `TX_REQUIRES_NEW`。那么,如果这个方法发出了一个应用程序异常,那么即使正在进行调用的方法捕获并处理了这个异常,该事务仍将回滚。

虽然没有被捕获的应用程序异常可能会导致一个事务回滚,但如果正在使用容器管理的事务,通过发出异常来回滚事务并不是一种值得推荐的方法。因为,容器可能不能确定这个异常是否‘足够严重’以至于需要进行回滚。相反,应该在 EJB 上下文中调用 `setRollbackOnly`。调用这个方法的一种典型情况就是,当对另一个 EJB 的调用失败并且有一个异常被捕获时。在某些情况下,可能会希望重试这个 EJB 调用。例如,可能是由于某种临时性的网络连接问题而导致发出异常。在一次或多次这种重试失败之后,或者如果被捕获的异常已经毫无

疑问就是异常时（例如 `TransactionRolledbackException`），那么应该将 EJB 上下文设置为只能回滚。

如果代码实现了涉及几个 EJB 实例的冗长事务，那么不可避免存在这样的风险：事务过程中早期的某个 EJB 调用了 `setRollbackOnly`。在这样的情况下，就可能存在一个潜在的陷阱：事务过程中后来的 bean 不断地浪费 CPU 周期来处理一个注定失败的事务。为了避免出现这种情况，可以使用 `getRollbackOnly` 方法，从而使得 bean 实现能够检查它是否在一个有用的事务中执行。

31.2.3 bean 管理的事务

BMT（Bean Managed Transaction，Bean 管理的事务）可用于这样的情况：bean 需要在事务中保持活跃。如前所述，bean 管理的事务局限于会话 bean。需要说明的是，BMT 是真正为专家级开发人员提供的，并且，应该有足够好的理由来说明为什么不使用由容器管理的事务。

bean 可以从 EJB 上下文获取一个事务上下文（由 `UserTransaction` 对象表示）。如果这个 bean 是一个无状态会话 bean，那么它不能在 EJB 客户调用的多个方法中使用相同的事务上下文。原因很清楚：对于不同的方法的调用，不能指望使用会话 bean 中的任何成员变量。正如在第 29 章见到的那样，容器可以为不同的客户使用相同的 bean。对于有状态会话 bean，情况则不同。有状态会话 bean 可以将事务上下文当作一个实例变量，从而实现在几个方法调用之间重用相同的事务上下文。在这种情况下，也可能实现 `javax.ejb.SessionSynchronization` 接口来同步与事务的对话状态（当然，这仅仅是一种方便，因为事务被代码直接处理）。

注意：不应该让一个事务跨越多个方法（也就是说，不能在一个方法中启动某个事务，而在另一个方法中进行动作，然后在第三个方法中进行事务的确认）。

为了实现 BMT，在 bean 中必须包括由 `setSessionContext` 方法传递给它的 `SessionContext` 对象。后来，它可以从会话上下文对象中获取事务上下文。下列代码段给出了实现 BMT 的方法：

```
import javax.ejb.*;
import javax.transaction.*;

public class StatelessBeanManagingTransactions implements SessionBean
{
    private SessionContext ctx = null;

    // EJB required methods...
    public void setSessionContext(SessionContext c)
    {
        ctx = c;
    }
}
```

```
}

// Business method implementation
public void invoke(int a, int b, int c)
{
    UserTransaction trans = ctx.getUserTransaction();
    // Perform non-transactional work
    ...
    try () {
        trans.begin();
        // Perform transactional work
        ...
        trans.commit();
    } catch (RollbackException ex1) {
        ...
    } catch (HeuristicRollbackException ex2) {
        ...
    } catch (HeuristicMixedException ex3) {
        ...
    } catch (NotSupportedException ex4) {
        ...
    } catch (SystemException ex5) {
        ...
    }
    // Perform non-transactional work
    ...
}
...
}
```

在使用 BMT 时,可能会用到 `UserTransaction` 对象的 `getStatus` 方法,来确定当前事务是否注定失败。对于失败的情况,BMT bean 可以调用 `UserTransaction` 对象的 `setRollbackOnly` 方法。只有在任何重试都进行过之后,才应该调用这个方法。

31.2.4 从客户方代码管理事务

虽然 EJB 编程模型允许客户参与事务管理,但这样就使客户方编程工作复杂化。从系统设计的角度看,它违背了瘦客户的原则,因为瘦客户应该关注于表示,而使用服务器方服务来处理诸如事务这样的问题。在 WebSphere 环境下,服务器的事务服务是使用 JNDI 名 `jta/usertransaction` 来实现的。在获取了一个事务上下文之后,就可以使用 `begin` 方法与 `commit` 方法进行事务的划分。事务所包含的 EJB 调用(以及它们对 bean 的数据的影响)显然是这个事务的一部分。关于这种调用的示意代码非常类似于前面所显示的代码段:

```
import javax.transaction.*;
import javax.ejb.*;

class ClientCode {
    // ...
    void transactionMethod() {
        // Obtain EJB reference
        ...
        // Perform non-transactional work
        ...
        // Lookup transaction
        Context initialContext = new InitialContext();
        UserTransaction tcon =
            (UserTransaction) initialContext.lookup(
                "jta/usertransaction");

        tcon.begin();
        // Perform EJB invocations that are part of the transaction
        tcon.commit();
        // Non-transactional work
    }
}
```

并不推荐这种从客户方直接激活事务的做法。一个更好的设计选择应该是，将所有事务管理封装在一个会话 bean（按照定义，这个 bean 代表客户进行动作）中。然后，这些 bean 就可以使用 EJB 的事务属性，并且充分利用 EJB 容器的事务功能。

31.3 EJB 安全性

当发生 EJB 方法调用时，调用方将向 EJB 容器传递一个用户上下文（user context），用来代表先前被认证的调用者标识（认证并不是此处讨论的话题，第 19 章与第 44 章讨论了认证）。EJB 安全性关注于授权（authorization）。

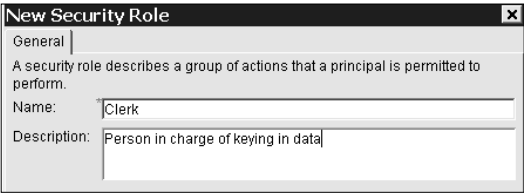
可以通过编程或者通过声明来实现 EJB 安全性。通过编程管理安全性意味着需要显式地编写代码来处理安全性检查。这样就使得安全性被硬编码到 bean 中，同时将增加 bean 中的代码量。通过声明进行授权就是将授权的工作委托到 EJB 容器进行，而使得 bean 实现关注于业务逻辑而不是安全性。这对于维护由于将 bean 开发人员与 bean 装配者和部署者之间的职责分离而带来的灵活性也是很重要的。

31.3.1 通过声明进行 EJB 授权

通过声明进行 EJB 授权在 AAT 中定义，这类似于其他部署属性。在 EJB 中的授权依赖于安

全性角色 (security role)。安全性关系与客户标识或一组客户标识有比较松的对应关系。安全性角色以及相关的许可在部署时定义。
在 AAT 中添加新安全性角色是很容易的。

1. 右击 EJB 模块的 Security Roles 表项，然后选择 New。这时将出现下列对话框：



2. AAT 将显示 EJB 模块中可用的角色的列表。例如，在添加两个角色之后，所得到的显示如图 31-3 所示。
3. 为了定义方法调用许可，应该右击 Method Permissions 表项并选择 New，这时将出现如下的 New Method Permission 对话框：

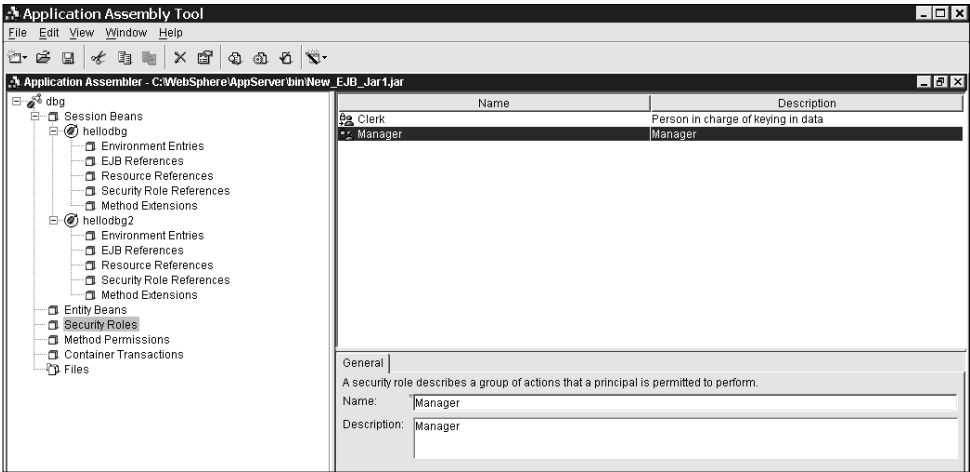
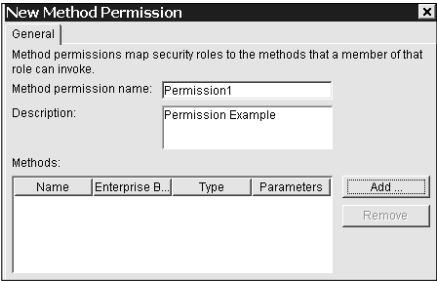
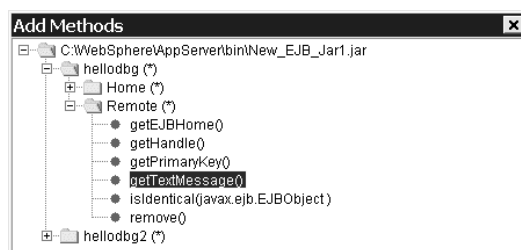
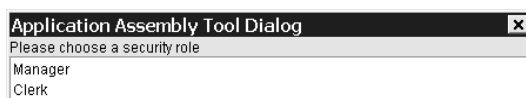


图 31-3 使用 AAT 设置安全性角色

4. 在这个对话框中，可以选择一个或多个方法，并且可以选择一个或多个安全性角色。为了选择一个方法，应该单击 Add 按钮以打开 Add Methods 窗口：



5. 可以类似地从可用的角色列表中选择安全性角色：



6. 在从可用的角色中进行选择之后，就完成了方法许可表项（包括方法选择与角色选择），如图 31-4 所示。这种方法许可的意义就是所有选定的安全性角色都允许调用所有被选择的方法。

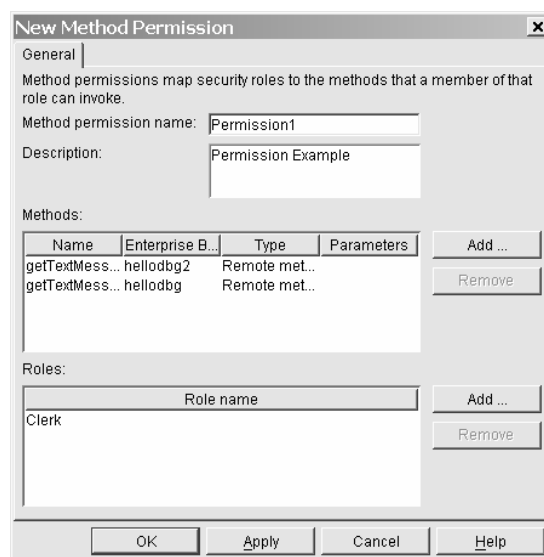


图 31-4 定义方法许可

除了方法许可之外，也可以决定安全性标识在方法调用之间传递的方式（有时将这个问题称为安全性委托（security delegation））。这一任务可以在 Method Extensions 对话框的

Advanced 标签中完成，在这里，可以选择 Run-As Mode。这个模式决定了某个方法被调用时以什么标识运行该方法。这个方法可以假设使用调用者标识、SYSTEM 标识（也就是 EJB 服务器）或另一个“特定的”安全性角色。为了选择一个特定的安全性角色，需要在可用的安全性角色列表（图 31-4 使用的同一个列表）中选择 Run-As Identity 选项。

虽然 EJB 模块的安全性角色仅仅是一些带有描述的名称，但在企业应用程序级，它们就与实际的用户以及用户组绑定。这种绑定是使用 AAT 中应用程序 Security Roles 表项的 Bindings 标签完成的，如图 31-5 所示。

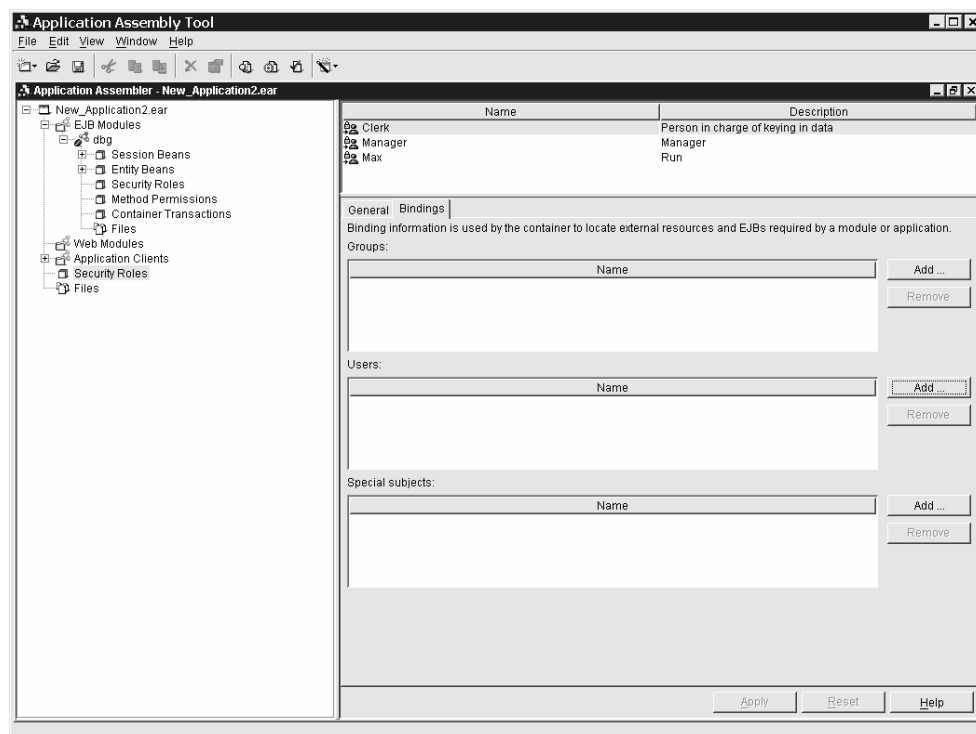


图 31-5 使用 AAT 绑定安全性角色

31.3.2 通过编程进行 EJB 授权

通过编程进行 EJB 授权依赖于调用 EJB 上下文的 isCallerInRole 方法与 getCallerIdentity 方法。前者用来检查调用者是否是某个角色的一部分，后者用来返回调用者的安全性标识。

下列代码段演示了这两个方法的用法：

```
import javax.ejb.*;  
import java.security.Principal;
```

```
public class SecureSessionBean implements SessionBean
{
    private SessionContext ctx;

    void setSessionContext(SessionContext c)
    {
        ctx = c;
    }

    // Other methods
    ...

    // Method which makes sure the user is running under the role
    // 'SYSOP'
    public void sensitiveMethod()
    {
        if (ctx.isCallerInRole("SYSOP")) {
            // Do important stuff
            ...
        }
        else {
            // Don't do anything or throw exception
        }
    }

    // Method which returns the caller identity
    String void getCallerIdent()
    {
        Principal p = ctx.getCallerPrincipal();
        return p.getName();
    }
}
```

使用这些代码的一个重要前提就是假设在部署描述符中将包括 **SYSOP** 安全性角色。当这个角色在安全性角色列表中可用时，应用程序部署者就能够自由地将用户映射到这个角色上。如果开发人员没有将这个安全性角色包括在部署描述符中，那么这个代码将不能正确运行（或者，它至少不能运行 `if` 语句中的代码段，这个代码段用来测试调用者角色）。

31.3.3 部署描述符中的安全性属性

与其他部署属性一样，安全性属性也在 XML 部署描述符中给出。下面给出了在添加用户与许可之后（请参见图 31-4）的 XML 描述符。其中，与安全性相关的属性用黑体表示。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
    "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 1.1//EN"
    "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">
<ejb-jar id="ejb-jar_ID">
    <display-name>dbg</display-name>
    <enterprise-beans>
        <session id="Session_1">
            <description>hellodbg</description>
            <display-name>hellodbg</display-name>
            <ejb-name>hellodbg</ejb-name>
            <home>HelloDbgHome</home>
            <remote>HelloDbg</remote>
            <ejb-class>HelloDbgBean</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
        <session id="Session_2">
            <description>hellodbg2</description>
            <display-name>hellodbg2</display-name>
            <ejb-name>hellodbg2</ejb-name>
            <home>HelloDbgHome2</home>
            <remote>HelloDbg2</remote>
            <ejb-class>HelloDbgBean2</ejb-class>
            <session-type>Stateless</session-type>
            <transaction-type>Container</transaction-type>
        </session>
    </enterprise-beans>
    <assembly-descriptor id="AssemblyDescriptor_1">
        <security-role id="SecurityRole_1">
            <description>Manager</description>
            <role-name>Manager</role-name>
        </security-role>
        <security-role id="SecurityRole_2">
            <description>Person in charge of keying in data</description>
            <role-name>Clerk</role-name>
        </security-role>
        <method-permission id="MethodPermission_1">
            <description>Permission1+:Permission Example</description>
            <role-name>Clerk</role-name>
            <method id="MethodElement_1">
                <ejb-name>hellodbg2</ejb-name>
                <method-intf>Remote</method-intf>
                <method-name>getTextMessage</method-name>
            </method>
        </method-permission>
    </assembly-descriptor>
</ejb-jar>
```

```
        <method-params>
        </method-params>
    </method>
    <method id="MethodElement_2">
        <ejb-name>hellodbg</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>getTextMessage</method-name>
        <method-params>
        </method-params>
    </method>
</method-permission>
</assembly-descriptor>
</ejb-jar>
```

31.4 小 结

事务与安全性可能是服务器方业务逻辑开发人员面临的最困难问题。所有业务应用程序都需要处理事务与安全性问题。因此，这两个话题都在 EJB 框架中处理，这样就创建了一个统一的应用程序框架并且可能为业务系统开发人员节省大量的工作。

EJB 模型中的事务与安全性都有两种方式——其中一种就是 EJB 容器管理这些问题，另一种就是开发人员使用一系列 API 直接处理事务和/或安全性。第一种方法更为可取，这有许多原因。开发人员只需要做更少的工作，由专家（在这里就是 WebSphere 内核开发组）构建的容器能够更好地处理这些问题，并且对事务与安全性行为的决定具有很大的灵活性，以至于应用程序部署者能够根据应用程序行为进行相关的决策。第 32 章将继续讨论 EJB 模型，并且介绍 WebSphere 上为 EJB 添加的最新功能——消息 bean。

第 32 章 消息驱动的 bean

EJB 体系结构的目标是在服务器方扮演一个重要角色，因此它必须解决关于消息传递的问题。由于 EJB 是用 Java 实现的，因此就可能在 EJB 中使用 JMS (Java Messaging Service, Java 消息传递服务)。但是，EJB 方法调用是通过 RMI-IIOP 实现的，这意味着与远程客户的通信以同步 (synchronous) 的形式发生。因此，将 EJB 与 JMS 异步通信结合起来的工作并不简单。将 EJB 与 JMS 结合在一起的完全解决方案必须解决与 EJB 相关的各种问题，例如生命周期与事务管理。许多这样的复杂问题都是由于 JMS 被用于异步通信而导致的。由于异步调用是分离的分布式应用程序的核心，因此将 EJB 模型与 JMS 模型进行合并能够为应用程序开发人员提供更好的支持。

消息传递是企业软件开发的重要工具，在第 14 章已经讨论了消息传递。EJB 规范的开发者们已经意识到对消息传递的支持必须加入到规范中。为了满足基于消息的服务器方软件的需求，已经对 EJB 规范进行了扩展，从而定义一种称为消息驱动的 (message-driven) bean 的特殊类型的 EJB。这些 bean 部分遵守 EJB 的概念，但同时又添加了一些特殊的功能与行为用来支持消息传递。消息驱动的 bean 是 EJB 2.0 规范中新加的部分，因此，消息驱动的 bean 并不是 WebSphere 4.0 高级版的组成部分。但它们将在 WebSphere 5.0 中可用。

WebSphere 4.0 企业版包括一个非标准的消息 bean 实现。本章将描述标准的消息驱动的 bean。我们相信，IBM 将在未来的 WebSphere 发布中包括对消息驱动的 bean 支持，作为遵守 J2EE 规范的一部分，并且用来加强自己的 MQSeries 产品。本章最后一节将描述 WebSphere 4.0 企业版中对消息驱动的 bean 的支持。

32.1 消息驱动的 bean 简介

消息驱动的 bean 是特殊的 EJB 组件，其目标是为消费 JMS 消息提供支持。如前所述，将 EJB 与 JMS 结合在一起应用会引发一些概念性的困难。这是因为 JMS (更广泛地说就是消息传递) 是基于所涉及的功能组 (例如，客户与服务器) 分离 (decoupling) 这一思想，而客户调用 EJB (特别是有状态 bean) 的方式是不合适的。而且，本地接口与远程接口对于通过 JMS 调用的 bean 没有用处，因为 JMS 自己提供了进行调用的机制。

消息驱动的 bean 可以看作无状态会话 bean 与 JMS 侦听程序的一种混合。消息驱动的 bean 没有对话状态，从而允许容器基于客户的需要管理这些 bean 的生命周期。这些 bean 只有一个业务方法，称为 onMessage，这个方法使用一个单独的 Message 对象作为参数并且返回 void 值。值得说明的是，这个消息是弱类型的，它就像 JMS 侦听程序相应的方法一样。

`onMessage` 方法必须使用 `instanceof` 仔细地检查这个消息，以确定该消息的实际类型。这一点不同于实体 bean 与会话 bean，它们可用来定义几个业务方法，且每个方法都使用强类型参数。而且，实体 bean 与会话 bean 能够有返回值，但异步消息传递范型则严禁消息驱动的 bean 有返回值。相反，消息驱动的 bean 能够通过发送一条封装有响应的消息进行响应。由于异步通信的共性以及消息传递特有的原因，使得消息驱动的 bean 中的 `onMessage()` 方法不能发出应用程序级异常。当然，可能发出系统异常，并且这些异常可以被 EJB 容器捕获并处理。

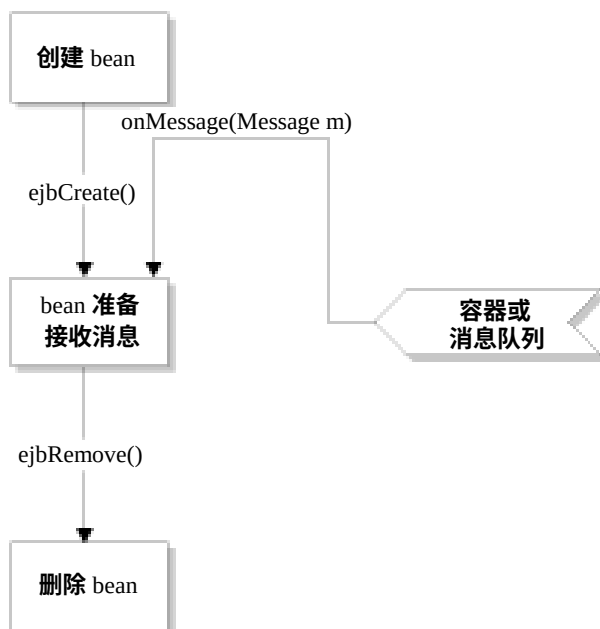
32.2 开发消息驱动的 bean

消息驱动的 bean 需要实现两个单独的接口：`javax.ejb.MessageBean` 与 `javax.jms.MessageListener`。这也说明，消息驱动的 bean 是 EJB 与 JMS 侦听程序的混合。

在 EJB 方，消息驱动的 bean 类能够实现下列方法。所有这 3 个方法都是必需的：

- `void ejbCreate()`：一个被声明为 `public` 但不是 `static` 的创建方法，并且返回值为 `void`（类似于会话 bean 与实体 bean）。但是，在这个创建方法中不使用任何参数。可以在这个方法中查找其他 EJB 引用或创建 JMS 连接（不同于用来接收被转发给 `onMessage` 方法的消息的连接）。
- `void ejbRemove()`：容器在析构 bean 之前将调用的一个删除方法。任何在创建方法中创建的数据库连接或 JMS 连接都应该在这个方法中被关闭。
- `void setMessageDrivenContext(MessageDrivenContext ctx)`：这个方法用来为 bean 提供 EJB 上下文，从而给出关于周围环境的信息。`MessageDrivenContext` 是 `EJBContext` 的扩展，并且为编程者提供了访问事务信息的方法，这在第 31 章已经描述。不能使用 `EJBContext` 接口的其他方法，因为这个 bean 没有本地接口或远程接口。与安全性相关的方法是没有意义的，因为并没有使用 EJB 方法调用来调用 `onMessage` 方法。

除了这些方法，`MessageListener` 接口还要求实现本章前面所描述的 `onMessage` 接口。如前所述，不必为消息驱动的 bean 提供本地接口或远程接口。这个 bean 实现自身就提供了容器所需的所有内容。因此，这种 bean 的实现比实体 bean 或会话 bean 的实现更简单。正如其实现一样，消息驱动的 bean 的生命周期也是非常简单的。当容器实例化（它然后调用 `ejbCreate` 方法）这个 bean 时，此 bean 的生命周期就开始了。然后，这个 bean 就可以接收消息了，并且，在后来使用 `onMessage` 反复地调用此方法。最后，当容器决定不再需要这个 bean 时（例如，当容器关闭时），它将调用 `ejbRemove` 方法来删除这个 bean。下图演示了消息驱动的 bean 的生命周期。



32.3 使用消息驱动的消息驱动 bean 的优点

EJB 体系结构的一个主要优点就是 EJB 容器厂商为应用程序实现了底层结构，从而使得开发人员只需要处理应用程序级细节，例如特殊的业务逻辑。这一办法也在消息驱动的消息驱动 bean 的实现中反映出来。

容器将代表 bean 自动执行几个动作：基于在部署时进行的配置，创建用来接收消息的消息消费者；注册消息侦听程序（实际上，用户不必调用 `setMessageListener`）；容器也基于 bean 部署参数，指定一个消息确认模式。

最重要的是，容器负责管理 bean 实例的池机制。换句话说，容器为处理消息而创建一个单独的 bean 实例，并且，如果消息负载很高，它将创建几个这样的实例来并行地为到达的消息服务。

而且，容器也提供了对容器管理的事务的支持。但用户必须知道，这一功能并不像为会话 bean 或实体 bean 提供的容器管理的事务功能那么强大。在这里存在的主要区别是，消息处理程序并不能代表发送该消息的那一方执行相同的分布式事务。当然，其原因就在于异步消息传递处理是为了将发送者与接收者分开而设计的，因此，使用相同的事务实际上就使得又回到了同步通信。在一个单独的事务上下文中，这实际上意味着如果允许一个事务既包括消息发送者又包括消息接收者，那么应用程序就可能出现死锁，因为消息发送者在直到事务被确认之前不会将消息放置在目标（例如队列）上。而消息接收者在消息出现在目

标上之前不会确认，因此，这样就可能导致整个应用程序被挂起。

那么，当请求容器管理事务时，容器应该如何工作呢？它将使用一个单独的事务范围读取消息并执行 `onMessage` 业务逻辑。如果出现了任何失败，事务将被回滚，并且特别是，确认将不会被发送给消息发送方。消息驱动的 bean 的容器管理的事务类似于其他 EJB 的事务，表现在代码中不需要显式地包含对 `rollback` 与 `commit` 的调用。

除了容器管理的事务之外，也可以使用 bean 管理的事务或者不使用任何事务。对 bean 管理的事务，容器将让用户自己实现原子性事务代码。在消息驱动的 bean 接收到消息之后，事务过程就开始了，然后，就可以在部署描述符中选择确认方式。实际上，当显式地调用 `UserTransaction` 对象的 `begin` 方法时，事务就开始，可以从 bean 上下文中获取这个对象。由于事务只能在接收到消息之后开始，因此对事务的回滚并不会导致消息被重新传送。这一点与容器管理的事务不同，在容器管理的事务中，`MessageDrivenContext` 方法对 `setRollbackOnly` 的调用（或者由于任何外部原因而导致的回滚）将导致消息被重新传送。当不使用事务支持时，容器对于消息确认的行为就没有定义。EJB 规范确实保证容器将发送一条消息确认。但是，容器是否对消息进行确认是由自己决定的；它并没有提供任何时间保证（也就是，在 bean 的 `onMessage` 方法返回之后，可能发送也可能不发送确认）。

32.4 消息驱动的 bean 示例

下面这个简单的例子演示了消息驱动的 bean 的概念。这个 bean 将接收文本消息并打印出这条消息，它在第 5 章“Hello World” EJB 提供的消息之后完成该动作。

这个 bean 实现了消息驱动的 bean 所需的每个方法：

- `ejbCreate` 方法查找 Hello bean 的本地接口并为这个 bean 创建一个远程接口。
- `ejbRemove` 方法删除会话 bean。
- `setMessageDrivenContext` 方法存储成员变量 `ctx` 中的上下文。
- `onMessage` 方法接收消息。如果这个消息不是文本消息，那么将打印出一条错误消息。否则，远程方法被调用以获取这个文本消息，然后，这个文本消息与输入消息的文本进行合并之后被打印出来。

下列例子是伪造的，表现在远程 bean 提供了一条常量消息，但它演示了调用会话 bean 的消息 bean 的思想。这是一个非常常见的组合，其中会话 bean 用作应用程序逻辑的入口点，而消息驱动的 bean 则提供了在异步消息传递环境下给出其功能的一种方法。

```
import javax.jms.*;
import javax.ejb.*;
import book.websphere.chapter5;

public class HelloMsgBean implements MessageDrivenBean, MessageListener {
```



```
private MessageDrivenContext ctx;
private Hello helloBean;

public void ejbCreate()
{
    try {
        Properties env = System.getProperties();
        env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        env.put(javax.naming.Context.PROVIDER_URL, "iiop://localhost");
        InitialContext ic = new InitialContext(env);
        HelloHome home =
            (HelloHome) javax.rmi.PortableRemoteObject.narrow(
                ic.lookup("HelloBean"));

        helloBean = home.create();
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}

public void ejbRemove()
{
    try {
        Properties env = System.getProperties();
        env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        env.put(javax.naming.Context.PROVIDER_URL, "iiop://localhost");
        InitialContext ic = new InitialContext(env);
        HelloHome home = (HelloHome) javax.rmi.PortableRemoteObject.narrow(
            ic.lookup("HelloBean"));

        home.remove(helloBean);
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}

public void setMessageDrivenContext(MessageDrivenContext c)
{
    ctx = c;
}
```

```

public void onMessage(Message msg)
{
    if (!(msg instanceof TextMessage)) {
        System.out.println("HelloMsgBean received wrong message type" +
            msg.getClass().getName());
        return;
    }
    try {
        TextMessage textMsg = (TextMessage) msg;
        String msgText = msg.getText();
        String helloText = null;
        helloText = helloBean.getTextMessage();
        System.out.println(helloText + textMsg);
    }
    catch (Exception ex) {
        ex.printStackTrace();
    }
}

```

32.5 消息驱动的 bean 的部署描述符

类似于会话 bean 与实体 bean，利用基于 XML 的部署描述符来指定部署参数。

下例为前面讨论的消息驱动的消息驱动的 bean 提供了一个部署描述符。其中，消息驱动的 bean 的标记以黑体表示。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC
    "-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans 2.0/EN"
    "http://java.sun.com/j2ee/dtds/ejb-jar_2_0.dtd">
<ejb-jar id="ejb-jar_ID">
    <description>Hello Message Bean Module</description>
    <display-name>HelloMsgBeanModule</display-name>
    <enterprise-beans>
        <message-driven id="MessageDriven_1">
            <description>Hello Message Bean</description>
            <display-name>HelloMsgBean</display-name>
            <ejb-name>HelloMsgBean</ejb-name>
            <ejb-class>book.websphere.chapter32.HelloMsgBean</ejb-class>
            <transaction-type>Container</transaction-type>
            <message-driven-destination>
                <destination-type>javax.jms.Queue</destination-type>
                <subscription-durability>nondurable</subscription-durability>
            </message-driven-destination>
        </message-driven id="MessageDriven_1">
    </enterprise-beans>
</ejb-jar id="ejb-jar_ID">

```

```
        </message-driven-destination>
    </message-driven>
</enterprise-beans>
<assembly-descriptor id="AssemblyDescriptor_1">
    </assembly-descriptor>
</ejb-jar>
```

destination-type 描述符定义了消息如何通过 JMS 被传递,例如,使用队列或话题。subscription-durability 标记用来控制 bean 的持久性属性;只有当 bean 是一个话题 bean 时,这个 bean 才是持久的。transaction-type 在消息驱动的 bean 中具有不同的含义。可选的<acknowledgement-mode>标记可用来指定当使用了 bean 管理的事务划分时,消息将如何被确认。消息驱动的 bean 支持(但没有在这里给出)的另一个标记就是<message-selector>标记。这个标记实现了在 14.4.2 一节中详细讨论的关于 EJB 规范的消息选择器的概念。使用这个标记,就能够指定可用于引用消息类型与属性的类似于 SQL 的语句。为了将被部署的 bean 附加到一个特定的消息队列上,还需要使用一些额外的标记,但这些标记并不是作为 EJB 规范的一部分而提供的,它们可以由厂商特定的扩展来解决。

32.6 WebSphere 4.0 中 Enterprise Extension 的消息驱动的 bean

WebSphere Application Server 4.0 包括一个称为 Extended Messaging Support 的 Enterprise Extension,它提供了非标准的消息驱动的 bean。

为了快速将消息驱动的 bean 引入到 WebSphere 中,WebSphere 设计者采用了一条捷径。它不是扩展 EJB 容器使其支持这 3 种类型的 bean,而是将消息驱动的 bean 作为会话 bean 实现。

这样看上去似乎不好理解,特别是考虑到为消息驱动的 bean 提供支持的 EJB 2.0 规范时更是如此。但是,对于实际的应用目标,它提供了一个比较合理的折中:目前市场上任何 EJB 2.0 实现在稳定性与可靠性方面都比不上 WebSphere 4.0。在这里所采用的引入消息 bean 的方法虽然是非标准方法,但它为这些 bean 的使用提供了极大的方便。

WebSphere Extended Messaging Service 扮演了消息驱动的 bean 的 EJB 容器相同的角色。它建立连接与消息队列(或话题),在接收到消息时调用消息 bean。为了使用这个服务,需要进行以下两步工作。

首先,需要开发自己的消息 bean(一个带有本地接口与远程接口的普通会话 bean)。这个 bean 定义并实现了 onMessage(javax.jms.Message)方法。类似于“真实”EJB 2.0 消息驱动的 bean 的要求,这个方法也必须是 public 的且非静态的,并且这个方法的返回值为 void,而且不应该发出任何应用程序异常。

下一步就是配置消息传递服务。为此,需要提供一个用来描述该配置的 XML 文件。可以通过 Administrative Console(将在第 42 章详细讨论),编辑应用程序服务器 Custom 标签下的

Extended Messaging Support Service 表项，从而将这个文件引入 WebSphere 中。只有当使用 Enterprise Extension 时，这个选项才可用。

这个 XML 文件详细定义了用来实现 JMS 连接池（以及相关的 bean）的参数以及消息 bean 的参数。其中，连接池的参数包括一个空闲超时（单位为毫秒），在经过这一超时之后，如果在池中没有使用的连接数超过了允许的最大数，那么这个连接将被废弃。

bean（或侦听程序）参数包括：

- bean 本地接口的 JNDI 名
- JMS 连接工厂
- JMS 目标名
- JMS 目标类型（队列或话题）
- 持久性（可持久或不可持久）
- 订阅者名（对于持久话题订阅）
- JMS 选择器
- JMS 确认模式
- 试图获取某条消息时最大的重试次数
- 侦听程序对正在侦听的目标具有的最大会话数
- 事务标志

下面给出了一个这种配置的例子：

```
<Config>
  <Pooling>
    <Timeout>30000</Timeout>
    <Threshold>10</Threshold>
  </Pooling>
  <Listener>
    <HomeJNDIName>msgBeanHome</HomeJNDIName>
    <JMSConnectionFactory>ivtQCF<JMSConnectionFactory>
    <JMSDestination>MessageQueue</JMSDestination>
    <JMSDestinationType>javax.jms.Queue</JMSDestinationType>
    <JMSSubscriptionDurability>nondurable</JMSSubscriptionDurability>
    <JMSAcknowledgeMode>auto-acknowledge</JMSAcknowledgeMode>
    <MaxRetries>0</MaxRetries>
    <MaxSessions>1</MaxSessions>
    <Transactional>True</Transactional>
  </Listener>
</Config>
```

32.7 小 结

消息传递与异步通信迅速成为构建分离、分布式系统的首选体系结构。因此，消息传递范型成为通过 WebSphere 部署的电子商务与应用程序的核心。在认识到这一事实之后，EJB 定义过程将消息驱动的 bean 引入 EJB 2.0 规范。WebSphere 4.0 还没有完全实现这个功能，但由于 IBM 一直遵守标准，毫无疑问，消息驱动的 bean 将被包含在 WebSphere 的未来版本中（可能是 WebSphere 5.0，或者是 WebSphere 4.x）。此外，WebSphere 中包括了一个提供类似功能的支持 JMS 的会话 bean 实现。本章对两者都进行了介绍，这是因为我们相信这一功能对于许多 WebSphere 应用程序都非常有用。

本章将结束对 WebSphere 中支持的不同 EJB 类型的讨论。第 33 章将给出 WebSphere 为 EJB 开发人员提供的某些专利性功能；然后，将进入到一个具有总结性质的章中，在这一章将使用本书介绍的许多概念，构建一个例子，演示 EJB 在多层应用程序中的使用。

第 33 章 IBM EJB 扩展

虽然 J2EE 是服务器方开发的标准，但 WebSphere 仍然提供了一些不是 EJB 1.1 规范的特定 EJB 功能。这实际上是一把双刃剑：一方面，这些功能常常非常有用并且能够节省大量的工作；但另一方面，这样又使得代码不可移植。显然，一个标准越成熟，就有越少的产品会偏离这个标准。WebSphere Application Server 的先前版本提供了更多偏离标准的功能。在先前的章中，我们已经在 AAT 中遇到了几个部署参数，这些参数与 IBM EJB 的扩展相关。本章将描述这些参数，并且描述那些不是 EJB 1.1 标准的其他功能。

33.1 IBM WebSphere 特定的部署参数

在一个 EJB 模块的装配阶段，当添加一个新 bean（会话 bean 或实体 bean）时，这个新 bean 的对话框中就包括一个 IBM Extensions 标签。在这些对话框中，扩展是标准 J2EE 部署描述符的附加部分。它们能够支持使用 Enterprise Extension、原有系统或规范中没有定义的行为。其中有些扩展是会话 bean 与实体 bean 公用的（如图 33-1 与图 33-2 所示）。但另有一些参数仅仅对一种 bean 有用（如图 33-2 所示）。

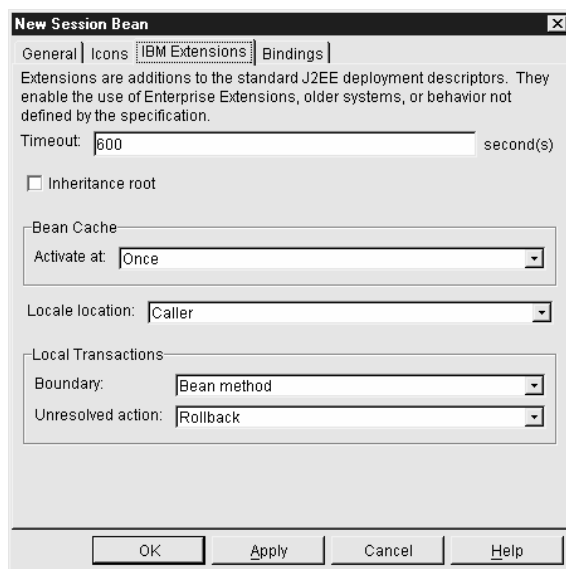


图 33-1 会话 bean 的 IBM Extensions

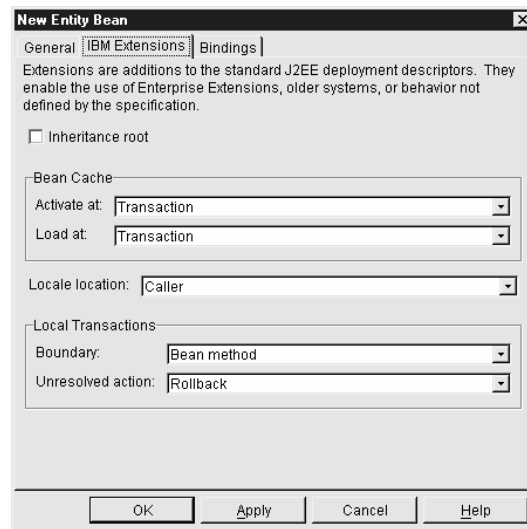


图 33-2 实体 bean 的 IBM Extensions

会话 bean 与实体 bean 都能够引用本地事务。当一个 EJB 方法在 EJB 规范称为未指定的 (unspecified) 上下文中运行时, 那么就创建一个本地 (local) 事务上下文。未指定的事务上下文能够在 bean 创建、删除、钝化、激活或 `setSessionContext` 方法调用过程中活跃。类似地, 对于那些不支持事务的 bean, 或者对于那些当客户没有在某个事务上下文中时不支持事务的 bean, 事务上下文就是本地事务。虽然这些情况并不是很重要 (也就是, 典型代码并不会在 `setSessionContext` 中开始一个事务), 但 WebSphere 使用本地事务解决了这些问题。这样就确保在任何可能的情况下, 事务都具有正确业务行为。可以被激活的动作包括回滚与确认, 其中默认的总是回滚动作。

33.1.1 会话 bean 的 IBM 扩展部署参数

IBM 为会话 bean 提供的特定部署参数如下:

- Timeout: 会话 bean 的超时时间。当发生超时时, 这个 bean 实例将被从系统中删除。
- Inheritance Root: 指定企业 bean 是否是一个继承层次的根。
- Activate at: 指定一个 bean 被激活的时机以及被放置在 bean cache 中的时机。可能的值包括:
 - Once: 这是默认值。当这个 bean 被服务器进程访问时, 它就被激活了一次, 并且由容器来决定在什么时机将这个 bean 去活 (换出)。有时这种方式称为懒惰的 (lazy) 激活。
 - Transaction: bean 在事务开始时被激活, 并且只有在事务完成之后才被去活。

这个参数仅仅对有状态会话 bean 有用（它对无状态 bean 无用，因为容器可以在不同的客户之间共享这些 bean）。

- Locale Location：为 bean 指定区域设置。当获取消息目录时，所使用的语言可以基于正在进行调用的客户的区域，或者基于服务器的区域。可能的值是 Server 与 Caller。
- Local Transactions：包括下列情况：
 - Boundary：用来指定一个本地事务从什么地方开始。这个属性不能应用于会话 bean，即使 AAT 允许它被设置。
 - Unresolved Action：如果一个应用程序在本地事务中“取消确认”资源，这个设置用来指定容器将要采取的动作。可能的值是 Rollback（默认）与 Commit。

33.1.2 实体 bean 的 IBM 扩展部署参数

对于实体 bean，下列参数具有与会话 bean 相同的含义：

- Inheritance Root
- Locale Location
- Local Transactions——Boundary 与 Unresolved Action

Activate at 的 Bean Cache 属性类似于有状态会话 bean。Load at 属性指定了 bean 应该在什么时候从数据库中加载状态。有效值是 Activation 与 Transaction。前者表示当激活时加载 bean，后者表示在开始事务时加载 bean。Load at 属性的值用来指出 EJB 容器是应该独占数据库访问还是共享数据库访问（Activation 表示独占访问，Transaction 表示共享访问）。

实体 bean 的 Activate at 与 Load at 属性的设置决定了将要使用的确认选项。确认选项是 EJB 1.1 规范的一部分，并且这两个属性允许应用程序装配者决定使用哪个选项。表 33-1 显示了 EJB 1.1 中的确认选项以及相应的 bean cache 设置。

表33-1 EJB 1.1中的确认选项以及相应的bean cache设置

确认选项	数据库访问的隐含模式	bean cache 设置	说明
A	独占	Activate at=Once	避免进行 ejbLoad 调用，从而有效地最小化数据库的 I/O。
		Load at=Activation	这种方式的缺点是访问这个 bean 实例的所有事务在数据库中都被串行化（这不同于容器），如果 bean 实例被多个事务并发访问，这样会导致比较低性能。而且，只能有一个容器（没有克隆/WLM、其他容器或任何其他 DB 应用程序）能够访问这些表
B	共享	Activate at=Once	由于有更多的对象在 cache 中，因此这种方式将增加内存需求。可能出现一个单独的 bean 实例在 cache 中有几个
		Load at=Transaction	

(续表)

确认选项	数据库访问的隐含模式	bean cache 设置	说明
C	共享	Activate at=Transaction Load at=Transaction	副本的情况（每个事务对应一个副本）。只有当这个 bean 在事务过程中被连续不断地访问时（它将被转换为多个 ejbActivate 调用，由于 bean 已经在 cache 中，因此可以自由地进行这些调用），才应该考虑将这个选项设置为 A 通过最小化在 cache 中存在的实体 bean 数，来减少内存需求。但是，它允许内存的相同实例存在多个副本（每个事务对应一个副本，这与选项 B 相同）。如果被并发访问的 bean 实例很少更新，使用这种方式很有用

33.2 管理数据库连接

在实现 BMP 时，大多数 EJB 方法都需要与底层数据库交互。为了支持连接池，需要建立数据库连接的 EJB 代码应该使用第 18 章描述的 DataSource 接口。

WebSphere 节点的管理员能够创建一系列特殊的实体 bean，这些实体 bean 封装关于数据库以及数据库驱动程序的信息。可以使用这些 bean 来获取 DataSource 对象。DataSource 接口提供了对 WebSphere 服务器连接池的透明访问，从而有助于提高应用程序性能。

为了访问管理员建立的数据源，所编写的代码需要使用简单的 JNDI 查找。例如，如果数据源名称为 DataSourceName，那么可以使用下列代码段：

```
InitialContext ic = new InitialContext();
DataSource ds =
    (DataSource) ic.lookup("java:comp/env/jdbc/DataSourceName");
```

注意，应用程序部署者必须将本地 JNDI 引用（"java:comp/env/jdbc/Data- SourceName"）映射到全局资源（DataSourceName）。

在获取 DataSource 对象之后，就可以使用这个对象来建立一条数据库连接，如下：

```
Connection con = ds.getConnection()
```

当使用 DataSource 建立连接时，可能仍然需要使用第 30 章描述的方法来建立这个连接。最容易实现的方法就是在 setEntityContext 方法中建立连接并且在 unsetEntityContext 方法中释放这个连接。当使用连接管理器时，这是一个非常好的选择，因为不需要担心使用太多连接的问题，连接管理器负责对连接实现池机制。

为了演示这种方法，下面将再次分析第 30 章的 BMP bean 例子，并且修改其实现以使用这一技术。所需做的修改如下：

1. 为 JNDI 以及 DataSource 接口创建一个新 import 语句。

```
import javax.naming.*;  
import javax.sql.DataSource;
```

2. 添加数据源作为这个类的一个成员：

```
public class CachedPageBean implements EntityBean  
{  
    public String userName;  
    public String pageUrl;  
    public boolean changeFlag;  
    public Date lastModified;  
    public String content;  
    transient protected EntityContext context;  
    private DataSource ds = null;  
    ...  
}
```

3. 在 setEntityContext 方法中设置 ds 成员：

```
public void setEntityContext (EntityContext ctx)  
{  
    context = ctx;  
    try {  
        InitialContext ic = new InitialContext();  
        DataSource ds = (DataSource)  
            ic.lookup("java:comp/env/jdbc/CachedPageDataSource");  
    }  
    catch (Exception ex) {  
    }  
}
```

4. 在 unsetEntityContext 方法中清除这个成员设置：

```
public void unsetEntityContext()  
{  
    context = null;  
    ds = null;  
}
```

5. 最后，需要建立连接的每个方法都可以使用 ds：

```
public void ejbLoad  
{  
    try {  
        CachedPagePK key = (CachedPagePK)context.getPrimaryKey();
```

```
        userName = key.userName;
        pageUrl = key.pageURL;

        String query = "select * from CACHEDPAGES set " +
            "where USER = '" + userName + "', '" +
            "PAGE = '" + pageUrl + "'";
        Connection con = ds.getConnection();
        PreparedStatement stmt = con.prepareStatement();
        ResultSet rs = stmt.executeQuery(query);
        boolean more = rs.next();
        if (more) {
            changeFlag = true;
            content = rs.getString(3);
            lastModified = new java.util.Date(rs.getString(4));
        }
        rs.close();
        stmt.close();
        con.close();
    }
    catch (Exception ex) {
        System.err.println("Error in ejbLoad " + pageUrl +
            " Exception " + ex);
    }
}
```

33.3 小 结

这一章真的很短，不过却是一个不错的暂停。WebSphere 4 是一个真正基于标准的服务器，并且提供了对 EJB 1.1 规范的完全支持。其中所提供的少量扩展只在某些极端情况下使用，大多数开发人员都不会遇到这种情况。从本章起，我们将基本结束对 WebSphere 中 EJB 的讨论。下一章将详细总结本书第 29~33 章所介绍的内容，并且给出一系列例子，用来演示如何将这些功能组织在一起，并应用于常见的多层分布式应用程序。

第 34 章 在多层应用程序中使用 EJB

分布式对象的出现以及 Web 应用程序的机会使得在企业计算领域发生了重大的范型转变。常用的客户-服务器范型已经被三层或多层方案所取代。

在多层体系结构下，组件是按照这样的方式进行层次划分的：每一层在物理上都与其他层隔开，并且每一层都使用它下面的层。特别地，最顶层通常关注于用户界面或表示，中间层负责处理业务逻辑，最底层（后端）是数据存储层（通常是数据库，有时也可以是遗留系统的封装器）。由于不同的层可能在共同的或不同的内存空间上，因此它们需要使用某种机制（例如第 8 章与第 9 章讨论的 CORBA/IIOP）进行通信。

多层体系结构与客户-服务器方法之间最关键的差别在于，业务逻辑从用户界面层分离出来，这样就使得瘦客户方法（客户可能是瘦客户、肥客户或介于两者之间）成为可能。实际上，多层体系结构与客户-服务器方法相比具有几个明显的优点。首先也是最重要的优点就是，多层方法能够更好地支持模块性（modularity），这样，在体系结构中就能够将用户界面、业务逻辑与数据处理清晰地分开。而这同时也意味着，每个这样的组件都能够在更低代价的条件下被修改。在模块化结构下，修改用户界面组件或修改业务逻辑显然比在单独的应用程序中更为容易。

多层方法的其他优点主要与网络流量和部署相关。用户界面与业务逻辑之间的通信量通常比业务逻辑与数据库之间的通信量要少得多。通过将业务逻辑从用户界面中分离出来，业务逻辑就能够放得更靠近服务器，从而去除了在更长、更慢的线路上进行网络通信的高开销。当然，这也就使得用户的应用体验更好。多层体系结构比客户-服务器系统的可伸缩性更好。它们减少了服务器方需要服务的元素数量。服务器不必直接服务于每个客户，而是被服务器层次之上的元素所访问。

34.1 多层与 J2EE

J2EE 与 EJB 体系结构都支持多层体系结构。这主要是通过表示、业务逻辑与数据仓库之间明确定义的分离来实现的。表示逻辑基于 Web 应用程序，Web 应用程序由静态内容（例如 HTML 页面、图像与 PDF）、JSP 与 servlet 组成。业务逻辑由 EJB 捕获，数据库逻辑通常基于关系数据库。能够具有容器管理的 bean 持续性，就意味着可以在不必对数据库逻辑进行任何编码的情况下开发应用程序。

Web 应用程序与 EJB 之间的分离，并不一定意味着在表示层就没有封装任何业务规则。实际上，这并不是被显式地禁止的。表示层可以由多种方法实现，其中有些方法还可能捕获

所有的业务逻辑。但是，为了使应用程序更具有可伸缩性并且更健壮，推荐使用 EJB 作为中间层。但有些业务逻辑总是发生在表示层（例如输入字段有效性检查）。

J2EE 通过提供表示逻辑、业务逻辑与数据库之间明确的分离，从而在定义上形成了一种三层体系结构。EJB 固有的特性也使得有必要实现包括多层的复杂体系结构。实际上，为来自客户（例如，表示逻辑）的请求提供服务的 EJB，自己也可能是其他 EJB 的客户。

在一个真正的多层体系结构（多于三层）中，可能有几层 EJB 用于提供中间层。这对于复杂应用程序是合适的，在这种应用程序中，需要多个层次提供模块性、可靠性或性能。对于复杂的业务逻辑，最好是注重模块化，将代码划分到几个 EJB 中实现，而不是将所有的业务逻辑代码都放在一个单独的 EJB 中完成。当然，使用几个 EJB 并不意味着就实现了多层，但为了简化客户方编程，应该提供一个明确定义的入口点（利用一个或少数几个 EJB 来提供中间层接口）。

J2EE 使用的 EJB 连接器（connector）结构，为多层体系结构提供了进一步的支持。正如在第 15 章说的那样，EJB 连接器是一种特殊的组件，它允许已有的系统挂接到 EJB 平台上。这种连接器允许遗留系统在任何与 J2EE 兼容的应用程序服务器上实现“即插即用”。这种遗留系统可能是基于大型机的事务系统（例如 CICS），或者是后端应用程序（例如，ERP（Enterprise Resource Planning，企业资源规划）或 CRM（Customer Relationship Management，客户关系管理）系统）。IBM 在前面一版本的 WebSphere 中提供了这种连接器，能够与 IMS（Information Management System，信息管理系统）、CICS、Encina 与 MQSeries 实现连接。在编写本书时，这种连接器还不能被用作 EJB 连接器，但在不久的将来，EJB 连接器可能也会存在。EJB 连接器体系结构的主要优点是，它提供了与外部系统进行连接的标准方法，这样就使得 EJB 代码与连接器在 J2EE 系统之间更容易移植。

本章的其余部分将提供一个利用 WebSphere 构建的多层应用程序的详细例子。

34.2 示例：Onamix 在线食品商店

为了演示 WebSphere 中的多层体系结构，下面将开发一个简单的电子交易应用程序。许多电子交易应用程序都采用了多层范型的核心思想。这种应用程序要求多层体系结构具有很好的可伸缩性，因为可能有不定数量的用户（Web 冲浪者）访问 Web 应用程序。这些用户的动作将通过中间层传递给应用程序数据库。这个例子将关注一个提供日本食品的在线商店，下面将把这个在线商店称为“Onamix”。

34.2.1 Onamix 组件

图 34-1 显示了 Onamix 应用程序中组件的高层次视图。其中，最高层负责表示（或用户界面），由 servlet 与 JSP 组成。这些 servlet 与中间层进行交互，中间层由食物碟（食物）、顾

客、订单与购物车组成。有些 EJB 是会话 EJB，用来捕获与会话相关的数据以及封装业务逻辑。在这个例子中，这样的业务逻辑可用来提供折扣或送货到户费。其他 bean 用来捕获持续性数据，并且使用实体 bean 来实现。

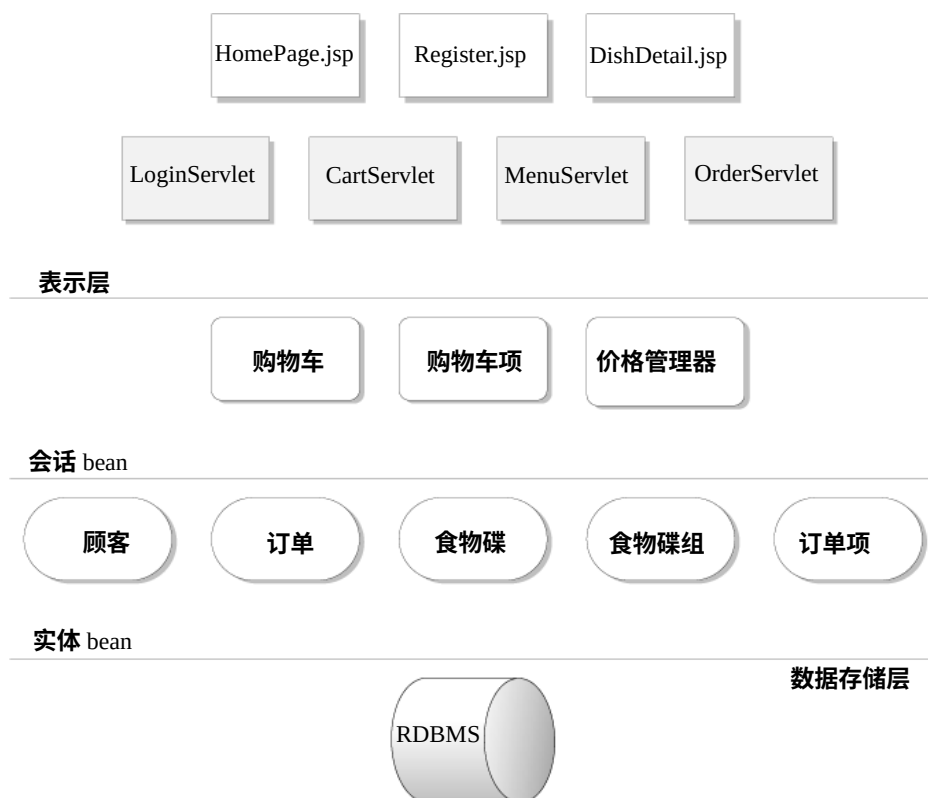


图 34-1 Onamix 应用程序的高层体系结构

表示层包括下列元素：

- HomePage.jsp：用来提供关于本商店“门面”的 JSP，通常显示（还有其他功能）当天的特价商品。
- LoginServlet：用于顾客登录的 servlet。
- Register.jsp：用于新顾客注册的 JSP。
- MenuServlet：用来允许顾客浏览菜单的 servlet。
- DishDetail.jsp：用来显示特定食物碟详细信息的 JSP。
- CartServlet：用来显示当前购物车的 servlet。
- OrderServlet：用来执行实际订货动作的 servlet。

这个例子中使用的会话 EJB 包括下列部分：

- Cart：购物车，由 CartItem 元素组成。
- CartItem：单个购物车项目。
- PricingManager：在其中封装了送货到户费与打折的逻辑。

Cart 与 CartItem bean 都是有状态 bean，因为它们维护了一个用户会话。

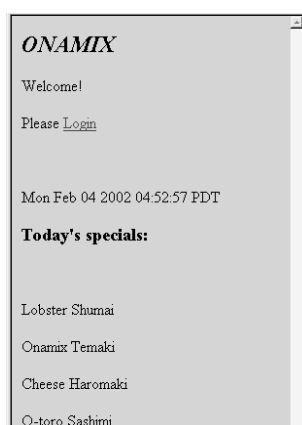
下面将要描述的实体 bean，用来表示应用程序域中的对象，例如顾客、食物碟与订单。这个例子中的所有实体 bean 都使用容器管理的持续性。

- Customer：系统的一个用户，将要向这个用户传递一些食物碟。
- Dish：一个在菜单上可用的单独的食物碟。
- DishGroup：一组食物碟。菜单由 DishGroup 对象组成。
- Order：顾客作出的在其中包括 OrderItem 元素的订单。
- OrderItem：订单中的一个单独项。

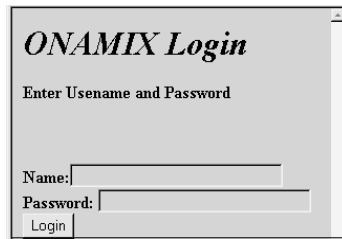
需要强调的是，为了实现一个真正可用的在线商店，还需要提供一些附加的服务程序。例如，可用一组功能——通常称为在线目录管理（online catalog management）——来建立（在这个例子中）关于食物碟的数据库。为了简洁，而且由于对这些功能的实现还常常涉及可重用、现货可用的类库，因此在这个例子中不给出这样的服务程序。

34.2.2 Onamix 用户透视

用户对这个应用程序的体验从 Onamix 主页开始，这个主页显示当前日期与时间，以及当天的特价商品：



在进行订货或浏览菜单之前，顾客需要登录。



ONAMIX Login

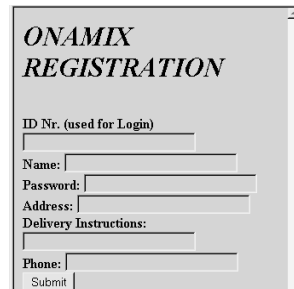
Enter Username and Password

Name:

Password:

Login

对于新用户，在进入该系统之前，需要注册。



ONAMIX REGISTRATION

ID Nr. (used for Login)

Name:

Password:

Address:

Delivery Instructions:

Phone:

Submit

一旦完成注册，用户就能够浏览菜单：



ONAMIX MENU

Appetizers

[Asadashi Tofu](#)
[Beef Takake](#)
[Cheese Haromaki](#)

Salads

[Tofu Salad](#)
[Wakame Salad](#)

Sashimi

[Sunaki Sashimi](#)
[O-toro Sashimi](#)
[Sake Sashimi](#)

Soups

[Miso Soup](#)
[Vegetable Ramen](#)

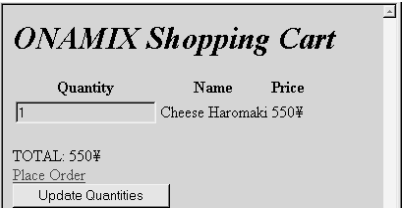
Sushi

[Onamix Temaki](#)
[Plain Inari](#)
[Maguru Nigiri](#)

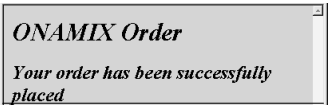
用户也可以选择特定的食物碟：



特定的食物碟可以被添加到购物车中，购物车将显示所有选择的项目总和：



在放置订单之后，将发出一条确认消息，用来说明订单已经被应用程序处理：



34.3 Onamix 应用程序实现

Onamix 应用程序的实现，将按照从底向上的方式进行说明，首先说明的是实体 EJB（实体 EJB 保存了应用程序的实际数据），然后说明实现业务逻辑的会话 bean 以及与会话相关的信息（例如购物车），最后说明用户界面组件。

34.3.1 Onamix 实体 bean

Onamix 应用程序中使用的实体 bean 直接与现实世界应用程序中的域实体（Dish，Customer 与 Order）相关。

Dish 实体 bean

Dish 实体 bean 代表菜单上一个单独的食物碟。每个食物碟都有一个名称（将被用作主键）、

描述以及一个整数价格（因为这里假设给出的是日元价格）。每个食物碟也有 3 个 boolean 属性，分别用来表示该食物是热的还是冷的、是否是当天的特价商品以及是否缺货。最后，这个 bean 还能够包括一个可选的图像，可以使用关于此图像的 URL 字符串来指定该图像。这个 bean 的远程接口如下：

```
package com.onamix;

import javax.ejb.*;
import java.rmi.RemoteException;

public interface Dish extends EJBObject {
    public String getName() throws RemoteException;
    public String getDescription() throws RemoteException;
    public void setDescription(String description) throws RemoteException;
    public int getPrice() throws RemoteException;
    public void setPrice(int price) throws RemoteException;
    public String getPicture() throws RemoteException;
    public void setPicture(String pic) throws RemoteException;
    public boolean getIsWarm() throws RemoteException;
    public void setIsWarm(boolean isWarm) throws RemoteException;
    public boolean getIsOutOfStock() throws RemoteException;
    public void setIsOutOfStock(boolean isOut) throws RemoteException;
    public boolean getIsSpecial() throws RemoteException;
    public void setIsSpecial(boolean isSpec) throws RemoteException;
}
```

这个 Dish bean 的主键基于食物碟名称得到，下面显示了这个主键对象的类的实现：

```
package com.onamix;
public class DishPK implements java.io.Serializable {
    public String name;
    public DishPK(String nm)
    {
        name = nm;
    }
    public String toString() {
        return name;
    }
    public boolean equals(Object d)
    {
        return ((DishPK) d).name.equals(name);
    }
    public int hashCode()
    {

```

```
        return name.hashCode();  
    }  
}
```

本地接口提供了各种各样的 `finder` 方法，用来允许用户按照名称、价格、描述以及冷热来获取食物碟，并且允许用户获取当天的特价商品。虽然所有这些 `finder` 方法在这个示例应用程序中都没有被实际使用，但提供尽可能丰富的接口是有用的。这样做能够使得所设计的 EJB 更具有开放性并且更具有可重用性。如果在设计的一开始就考虑尽可能丰富的接口，那么所设计的 EJB 就可重用，有时候，甚至原始的开发人员在设计之初都不能预见到这种可重用性。

```
package com.onamix;  
import javax.ejb.*;  
import java.rmi.RemoteException;  
import java.util.*;  
  
public interface DishHome extends EJBHome {  
    Dish create(String name, String description, int price,  
                String picture,  
                boolean warm, boolean outOfStock, boolean special)  
        throws CreateException, RemoteException;  
  
    public Dish findByPrimaryKey(DishPK key)  
        throws FinderException, RemoteException;  
  
    public Dish findByName(String name)  
        throws FinderException, RemoteException;  
    public Enumeration findBySpecial()  
        throws FinderException, RemoteException;  
    public Enumeration findAllDishes()  
        throws FinderException, RemoteException;  
}
```

这个 bean 的实现是很直观的。在这里使用的是容器管理的持续性。这种方法很简单，并且对这种方法使用不存在什么争论，因为这个 bean 中的所有数据元素都是原始元素。由于在这里让容器管理任何事情，因此其实现主要是一些 `get` 方法与 `set` 方法，如下所示：

```
package com.onamix;  
import javax.ejb.*;  
  
public class DishBean implements EntityBean {  
    private EntityContext ctx;  
    private String name;  
    private String description;  
    private boolean isOutOfStock;
```

```
private boolean isWarm;
private boolean isSpecial;
private String picture;
private int price;

public String getName()
{
    return name;
}

public String getDescription()
{
    return description;
}

public void setDescription(String desc)
{
    description = desc;
}

public boolean getIsOutOfStock()
{
    return isOutOfStock;
}

public void setIsOutOfStock(boolean b)
{
    isOutOfStock = b;
}

public boolean getIsWarm()
{
    return isWarm;
}

public void setIsWarm(boolean b)
{
    isWarm = b;
}

public boolean getIsSpecial()
{
    return isSpecial;
}

public void setIsSpecial(boolean b)
```

```
{
    isSpecial = b;
}

    public String getPicture()
    {
        return picture;
    }
    public void setPicture(String pic)
    {
        picture = pic;
    }
    public int getPrice()
    {
        return price;
    }
    public void setPrice(int pr)
    {
        price = pr;
    }
}

public void setEntityContext(EntityContext ct)
{
    ctx = ct;
}

public void unsetEntityContext() {
    ctx = null;
}

public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbStore() {}
public void ejbLoad() {}
public void ejbRemove() {}

public void ejbCreate(String nm, String desc, int pr, boolean warm)
{
    name = nm;
    description = desc;
    price = pr;
    isWarm = warm;
}
```

```
        public void ejbPostCreate(String nm, String desc, int pr,
                                   boolean warm) {}
    }
```

当然，还需要为 Dish bean 指定 finder 逻辑类，从而使得容器能够实现 finder 方法。在这里定义一些字符串，这些字符串将被用作 SELECT 语句中的 WHERE 短语，容器使用这些语句查询数据库（在这个数据库中查询食物碟），如下所示：

```
package com.onamix;
public interface DishBeanFinderHelper
{
    String findByNameWhereClause = "name = ?";
    String findBySpecialWhereClause = "isSpecial = TRUE";
    String findAllDishesWhereClause = "TRUE";
}
```

Customer 实体 bean

Customer bean 用来表示单个顾客。为了简单，这里将假设顾客有一个特殊的 ID 号（例如司机的驾驶许可证编号或护照编号），用来在应用程序中标识该顾客。为一个顾客存储的信息包括姓名、地址、电话号码以及特殊的送货说明。这个 bean 的远程接口如下：

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface Customer extends EJBObject
{
    public String getID() throws RemoteException;
    public void setID(String id) throws RemoteException;
    public String getPassword() throws RemoteException;
    public void setPassword(String password) throws RemoteException;
    public String getName() throws RemoteException;
    public void setName(String name) throws RemoteException;
    public String getPhone() throws RemoteException;
    public void setPhone(String phone) throws RemoteException;
    public String getAddress() throws RemoteException;
    public void setAddress(String address) throws RemoteException;
    public String getDeliveryInstruction() throws RemoteException;
    public void setDeliveryInstruction(String inst)
                                   throws RemoteException;
}
```

主键基于 ID，它提供了一种简单的方法用来在数据库中标识顾客：

```
package com.onamix;
```

```
import java.io.Serializable;

public class CustomerPK implements java.io.Serializable {
    public String ID;
    public CustomerPK(String id) {
        this.ID = id;
    }
    public CustomerPK() {
    }
    public String toString() {
        return ID;
    }

    public int hashCode() {
        return ID.hashCode();
    }

    public boolean equals(Object c) {
        return ((CustomerPK)c).ID.equals(ID);
    }
}
```

本地接口提供了创建新顾客（由注册页面所使用）以及按照姓名或 ID 查找顾客的功能：

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.*;

public interface CustomerHome extends EJBHome {
    Customer create(String ID, String name,
                    String passwd, String address,
                    String deliveryInst, String phone)
        throws CreateException, RemoteException;
    public Customer findByPrimaryKey(CustomerPK key)
        throws FinderException, RemoteException;
    public Enumeration findByName(String name)
        throws FinderException, RemoteException;
    public Enumeration findAllCustomers()
        throws FinderException, RemoteException;
}
```

同样，这个 bean 的实现也很直观，在这里将省略其中某些 set/get 方法。

```
package com.onamix;
```

```
import javax.ejb.*;

public class CustomerBean implements EntityBean {
    private EntityContext ctx;
    private String ID;
    private String name;
    private String password;
    private String address;
    private String deliveryInstructions;
    private String phone;

    public String getName() {
        return name;
    }
    public void setName(String nm) {
        name = nm;
    }

    ...

    public void ejbActivate()      {}
    public void ejbPassivate()     {}
    public void ejbLoad()          {}
    public void ejbStore()         {}
    public void ejbRemove()        {}

    public void ejbCreate(String id, String nm, String passwd,
                          String addr, String deliveryInst, String ph)
    {
        ID = id;
        name = nm;
        password = passwd;
        address = addr;
        deliveryInstructions = deliveryInst;
        phone = ph;
    }

    public void ejbPostCreate(String id, String nm, String passwd,
                             String addr, String deliveryInst, String ph)
    {}
}
```

同样，为了简单，在这里也使用容器管理的持续性。这就要求实现一个 finder helper 类，用

来创建适当的 WHERE 短语：

```
package com.onamix;

public interface CustomerBeanFinderHelper
{
    String findByNameWhereClause = "name = ?";
    String findByIdWhereClause = "id = ?";
    String findAllCustomersWhereClause = "TRUE";
}
```

DishGroup 实体 bean

DishGroup 将几个食物碟组合在一起，因此比前一个 bean 稍微复杂一点。在这里，食物碟被赋给一个组，从而在菜单中提供一种层次结构。DishGroup EJB 是关于实体 EJB 包容的非常好的例子。一个 DishGroup 包括一个或多个 Dish bean，并且这个 DishGroup 提供了容器所具有的功能。这就意味着，可以向其中添加元素，或者从其中删除元素。远程接口定义了用来添加及删除元素的方法，也定义了用来访问元素的方法：

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.*;

public interface DishGroup extends EJBObject {
    public String getName() throws RemoteException;
    public Enumeration getElements() throws RemoteException;
    public void addElement(Dish aDish) throws RemoteException;
    public void removeElement(Dish aDish) throws RemoteException;
}
```

主键类基于组名称（用来构建菜单层次结构的相同名称）：

```
package com.onamix;

public class DishGroupPK implements java.io.Serializable
{
    public String name;

    public DishGroupPK(String nm)
    {
        name = nm;
    }

    public String toString()
```

```
{
    return name;
}

public boolean equals(Object dish)
{
    return ((DishPK) dish).name.equals(name);
}

public int hashCode()
{
    return name.hashCode();
}
}
```

Dish 组的本地接口仅仅包括一个基于组名称的 finder 方法：

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.*;

public interface DishGroupHome extends EJBHome
{
    DishGroup create(String name)
        throws CreateException, RemoteException;
    public DishGroup findByPrimaryKey(DishGroupPK key)
        throws FinderException, RemoteException;
    public Enumeration findAllGroups()
        throws FinderException, RemoteException;
}
```

Dish 组的 bean 实现维护了一个关于 Dish 组实例的向量,并且实现了用于向这个向量中添加元素以及从这个向量中删除元素的方法。ejbLoad 方法用来确保当这个 bean 被加载到容器中时此向量将可用：

```
package com.onamix;
import javax.ejb.*;
import java.util.*;
import java.io.*;
import java.naming.*;
import java.rmi.RemoteException;
public class DishGroupBean implements EntityBean
{
    private EntityContext ctx;
```

```
private String name;
private Vector dishesPK;
private Vector dishes;
public void setEntityContext(EntityContext ct)
{
    ctx = ct;
}
public void unsetEntityContext()
{
    ctx = null;
}

public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbStore() {}
public void ejbRemove() {}
public void ejbLoad() throws RemoteException;
{
    try {
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming.WsnInitialContextFactory");
        p.put(Context.PROVIDER_URL, "iiop://localhost");
        InitialContext ic = new InitialContext(p);
        Object homObj = ic.lookup("DishHome");
        DishHome dishHome = (DishHome)
            javax.rmi.PortableRemoteObject.narrow(
                homObj, DishHome.class);
        Enumeration e = dishesPK.elements();
        while (e.hasMoreElements()) {
            DishPK pk = (DishPK) e.nextElement();
            dishes.addElement(dishHome.findByPrimaryKey(pk));
        }
    }
    catch (Exception ex) {
        throw new RemoteException(ex.toString());
    }
}

public void ejbCreate(String nm)
{
    name = nm;
    dishesPK = new Vector();
    dishes = new Vector();
}
```

```
    }

    public void ejbPostCreate(String nm) {}

    public String getName()
    {
        return name;
    }

    public Enumeration getElements()
    {
        return dishes.elements();
    }

    public void addElement(Dish dish)
    {
        dishes.addElement(dish);
        dishesPK.addElement((DishPK) dish.getPrimaryKey());
    }

    public void removeElement(Dish dish)
    {
        DishPK pk = (DishPK) dish.getPrimaryKey();
        for (int i = 0; i < dishesPK.size(); i++) {
            if (dishesPK[i].equals(pk)) {
                dishesPK.removeAt(i);
                dishes.removeAt(i);
                i--;
            }
        }
    }
}
```

OrderItem 实体 bean

订单项对应于订单中的一行，由 OrderItem bean 表示。这个 bean 保存了商品、商品的数量以及（可选的）打折。这个订单项的关键字是一个由订单号与行号组合而成的编号。这个订单项 bean 的远程接口如下：

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface OrderItem extends EJBObject
{
```

```
public Dish getDish() throws RemoteException;
public void setDish(Dish dish) throws RemoteException;
public int getPrice() throws RemoteException;
public int getQuantity() throws RemoteException;
public void setQuantity(int quantity) throws RemoteException;
public int getDiscount() throws RemoteException;
public void setDiscount(int discount) throws RemoteException;
}
```

OrderItemPK 类基于一个惟一的订单项 ID 实现了这个订单项的主键（类似于 GroupDishPK 类）。这个订单项 bean 的本地接口如下：

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;
import java.util.*;

public interface OrderItemHome extends EJBHome {
    public OrderItem create(String orderItemId, Order order,
                           Dish dish, int quantity, int discount)
        throws CreateException, RemoteException;
    public OrderItem findByPrimaryKey(OrderItemPK key)
        throws FinderException, RemoteException;
    public Enumeration findByOrderID(String orderID)
        throws FinderException, RemoteException;
}
```

同样，OrderItem bean 的实现也很简单，在这里将忽略其中某些 set/get 方法：

```
package com.onamix;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;
import javax.naming.*;

public class OrderItemBean implements EntityBean {
    private EntityContext ctx;
    private String orderItemID;
    private DishPK dishPK;
    private OrderPK orderPK;
    private Dish dish;
    private Order order;
    private int quantity;
    private int discount;
}
```

```
public void ejbActivate() {}
public void ejbPassivate() {}
public void ejbRemove() {}
public void ejbStore() {}
public void setEntityContext(EntityContext ct)
{
    ctx = ct;
}

public void unsetEntityContext()
{
    ctx = null;
}

public void ejbLoad() throws RemoteException
{
    try {
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming.WsnInitialContextFactory");
        p.put(Context.PROVIDER_URL, "iiop://localhost");
        InitialContext ic = new InitialContext(p);
        Object homObj = ic.lookup("OrderHome");
        OrderHome orderHome = (OrderHome)
            javax.rmi.PortableRemoteObject.narrow(
                homObj, OrderHome.class);
        order = orderHome.findByPrimaryKey(orderPK);
    }
    catch (Exception e) {
        throw new RemoteException(e.toString());
    }
}

public void ejbCreate(String oiid, Order ord, Dish aDish,
    int qty, int disc) throws RemoteException
{
    orderItemID = orderItemID;
    quantity = qty;
    discount = disc;
    order = ord;
    dish = aDish;
    orderPK = (OrderPK) order.getPrimaryKey();
}
```

```

        dishPK = (DishPK) dish.getPrimaryKey();
    }

    public void ejbPostCreate(String oiid, Order ord, Dish aDish,
                              int qty, int disc) {}

    ...
    public int getPrice() throws RemoteException
    {
        return (100 - discount) * quantity * dish.getPrice() / 100;
    }
}

```

同样，仍然使用 CMP 并使用下列 finder helper 类：

```

package com.onamix;
public interface OrderItemBeanFinderHelper
{
    String findByOrderIDWhereClause = "orderPK = ?";
}

```

Order 实体 bean

订单将顾客与一个或多个订单项联系起来。一个订单包括订单项、客户引用以及日期。一个订单记录了打折费用与送货到户费。订单的主键是一个序列编号，用来标识该订单。Order bean 的远程接口如下：

```

package com.onamix;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;

public interface Order extends EJBObject
{
    public Enumeration getItems() throws RemoteException;
    public Customer getCustomer() throws RemoteException;
    public int getTotal() throws RemoteException;
    public int getDeliveryCharges() throws RemoteException;
    public void setDeliveryCharges(int delivery) throws RemoteException;
    public Date getDate() throws RemoteException;
}

```

订单的主键基于一个惟一的订单 ID，下面显示了这个主键的类实现：

```

package com.onamix;

```

```
public class OrderPK implements java.io.Serializable
{
    public String orderID;
    public OrderPK(String oid)
    {
        orderID = oid;
    }
    public String toString()
    {
        return orderID.toString();
    }
    public boolean equals(Object order)
    {
        return ((OrderPK)order).orderID.equals(orderID);
    }

    public int hashCode()
    {
        return orderID.hashCode();
    }
}
```

Order bean 的本地接口基于主键与顾客定义了 finder 方法，因此可以查看与某个顾客相关的订单。

```
package com.onamix;
import java.rmi.RemoteException;
import javax.ejb.*;
import java.util.*;

public interface OrderHome extends EJBHome {
    public Order create(String orderID, Customer cust)
        throws CreateException, RemoteException;
    public Order findByPrimaryKey(OrderPK orderKey)
        throws FinderException, RemoteException;
    public Enumeration findByCustomer(CustomerPK customer)
        throws FinderException, RemoteException;
    public Enumeration findAllOrders()
        throws FinderException, RemoteException;
}
```

这个 bean 实现包括 get/set 方法（同样在这里将省略那些不感兴趣的部分），用于加载、创建与删除的框架方法以及一个用于计算订单总价格的业务方法：

```
package com.onamix;
```



```
import java.util.*;
import java.io.*;
import java.rmi.RemoteException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.ejb.*;

public class OrderBean implements EntityBean {
    private EntityContext ctx;
    private Customer customer;
    private Vector items;
    private String orderID;
    private CustomerPK customerPK;
    private Date date;
    private int deliveryCharges;

    public Enumeration getItems() throws RemoteException
    {
        OrderItemHome home = null;
        try {
            Properties p = new Properties();
            p.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.websphere.naming.WsnInitialContextFactory");
            p.put(Context.PROVIDER_URL, "iiop://localhost");
            InitialContext ic = new InitialContext(p);
            Object homObj = ic.lookup("OrderItemHome");
            home = (OrderItemHome)
                javax.rmi.PortableRemoteObject.narrow(
                    homObj, OrderItemHome.class);
            return home.findByOrderID(orderID);
        } catch (Exception e) {
            throw new RemoteException(e.toString());
        }
    }
    ...
    public int getTotalPrice() throws RemoteException
    {
        int total = 0;
        for (int i = 0; i < items.size(); i++) {
            try {
                OrderItem item = (OrderItem) items.elementAt(i);
                total += item.getPrice();
            }
            catch (Exception ex) {
```

```
        throw new RemoteException(ex.toString());
    }
}
total += deliveryCharges;
return total;
}

public void setEntityContext(EntityContext ct)
{
    ctx = ct;
}

public void unsetEntityContext()
{
    ctx = null;
}

public void ejbActivate()    {}
public void ejbPassivate()   {}
public void ejbStore()       {}
public void ejbLoad() throws RemoteException
{
    try {
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.ejs.ns.jndi.CNInitialContextFactory");
        p.put(Context.PROVIDER_URL, "iiop://localhost");
        InitialContext ic = new InitialContext(p);
        Object hom1obj = ic.lookup("CustomerHome");
        Object hom2obj = ic.lookup("OrderItemHome");
        CustomerHome customerHome = (CustomerHome)

            javax.rmi.PortableRemoteObject.narrow(
                hom1obj, CustomerHome.class);
        OrderItemHome orderItemHome = (OrderItemHome)

            javax.rmi.PortableRemoteObject.narrow(
                hom2obj, OrderItemHome.class);
        customer = customerHome.findByPrimaryKey(customerPK);
        Enumeration e = orderItemHome.findByOrderId(orderID);
        items = new Vector();
        while (e.hasMoreElements()) {
            items.addElement((OrderItem) e.nextElement());
        }
    }
}
```

```

    }
    catch (Exception e) {
        throw new RemoteException(e.toString());
    }
}

public void ejbCreate(String oid, Customer c) throws RemoteException
{
    orderID = oid;
    customer = c;
    customerPK = (CustomerPK)customer.getPrimaryKey();
    date = new Date();
    items = new Vector();
}

public void ejbPostCreate(String orderItemID, Customer c)
    throws CreateException, RemoteException {
}

public void ejbRemove() throws RemoteException, RemoveException
{
    Enumeration e = getItems();
    while (e.hasMoreElements()) {
        OrderItem item = (OrderItem) e.nextElement();
        item.remove();
    }
}
}

```

finder helper 类类似于前面所显示的相应类：

```

package com.onamix;

public interface OrderBeanFinderHelper
{
    String findByCustomerWhereClause = "customerPK = ?";
    String findAllOrdersWhereClause = "TRUE";
}

```

34.3.2 部署参数

现在，已经为实体 bean 编写了代码，下一步就需要决定部署描述符中的部署参数。对于这个应用程序中的所有 bean，都假设安全性 run-as 模式被设置为 CLIENT_IDENTITY。这样

就足够了，因为在这里并不希望任何 bean 执行任何安全的或特权的操作。这就意味着，所有的授权都基于发起这个操作的客户进行。

事务行为稍微复杂一点，并且不同的 bean 有不同的事务模式。Customer bean 通常只创建一次，并且不会经常修改（即使被修改，也仅仅被一个用户修改），因此可以将这个 bean 设置为具有最低的隔离级别（TRANSACTION_READ_UNCOMMITTED）的 TX_SUPPORTS 事务模式。类似的设置可以应用于食物碟和食物碟组。

另一方面，对于订单与订单项，为其设置 TX_REQUIRES 事务模式并且设置最高的隔离级别（TRANSACTION_SERIALIZABLE）。这样就避免了应用程序中的某些并发性问题（但还需要注意死锁）。

34.3.3 Onamix 会话 bean

Onamix 应用程序使用会话 bean 封装业务逻辑以及会话特定的数据。

Cart 会话 bean

在这个应用程序中，购物车维护了所有会话特定的数据。购物车由 Cart bean 表示，这个 bean 的远程接口包括购物车特定的方法（例如 clear 与 getTotalPrice），也包括用于管理购物车上下文的方法（例如 getCustomer）：

```
package com.onamix;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;
public interface Cart extends EJBObject
{
    public int getCount() throws RemoteException;
    public Enumeration.getItems() throws RemoteException;
    public void addDish(Dish dish) throws RemoteException;
    public void setDishQuantity(Dish dish, int quantity)
        throws RemoteException;
    public void clear() throws RemoteException;
    public Customer getCustomer() throws RemoteException;
    public int getTotalPrice() throws RemoteException;
    public Order placeOrder() throws RemoteException;
    public void setDiscount(int disc) throws RemoteException;
}
```

购物车包含一个 CartItem 对象的数组。此外，它还包括一个购物车所属的客户的引用。Cart bean 提供了方法，用来设置送货费用和打折数额；这些方法由 PricingManager bean 在外部决定。Cart bean 也提供了一个方法，可用来根据购物车中的项目创建一个订单。即利用 OrderItem bean 为购物车中的每个 CartItem bean 创建一个新 Order bean。

Cart bean 的本地接口仅仅是一个 create 方法。

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface CartHome extends EJBHome {
    public Cart create(Customer customer)
        throws CreateException, RemoteException;
}
```

下面显示了这个 bean 的实现。一个有趣的细节是，购物车 bean 的 ejbRemove 方法删除了购物车项目 bean。虽然从技术上说，可以利用会话 bean 的超时机制让容器释放没有被使用的会话 bean。但是，依靠这种服务并不好，因为这样将会使应用程序不具有很好的可伸缩性并且可能会存在大量的 bug。应该总是显式地释放由 bean 所使用的对象（当然，除非能够在客户调用之间重用这些 bean）。与购物车项目相反，从购物车中引用的 Customer bean 并没有被删除，因为这是一个持续性实体，它的存在性并不依赖于会话或购物车。

```
package com.onamix;
import javax.naming.*;
import javax.ejb.*;
import java.util.*;
import java.io.*;
import javax.naming.*;
import java.rmi.*;

public class CartBean implements SessionBean
{
    private SessionContext ctx;
    private Customer customer;
    private Vector items;

    public void setSessionContext(SessionContext ctx)
    {
        this.ctx = ctx;
    }

    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbCreate(Customer cust)
    {
        customer = cust;
        items = new Vector();
    }
}
```

```
public void ejbRemove()
{
    Enumeration e = items.elements();
    while (e.hasMoreElements()) {
        try {
            CartItem item = (CartItem) e.nextElement();
            item.remove();
        }
        catch (Exception ex) {
        }
    }
}

public void setDiscount(int d) throws RemoteException
{
    Enumeration e = items.elements();
    while (e.hasMoreElements()) {
        try {
            CartItem item = (CartItem) e.nextElement();
            item.setDiscount(d);
        }
        catch (Exception ex)
        {
            throw new RemoteException(ex.toString());
        }
    }
}

private CartItem getItem(Dish dish)
{
    Enumeration e = items.elements();
    while (e.hasMoreElements()) {
        try {
            CartItem item = (CartItem) e.nextElement();
            Dish otherDish = item.getDish();
            if (dish.isIdentical(otherDish)) {
                return item;
            }
        }
        catch (Exception ex)
        {
        }
    }
}
```

```
        return null;
    }

    public Customer getCustomer()
    {
        return customer;
    }

    public Enumeration getItems()
    {
        return items.elements();
    }

    public int getCount()
    {
        return items.size();
    }

    public void addDish(Dish dish, int qty) throws RemoteException
    {
        try {
            CartItem item = getItem(dish);
            if (item != null) {
                item.setQuantity(item.getQuantity() + qty);
                return;
            }
            Properties p = new Properties();
            p.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.websphere.naming.WsnInitialContextFactory");
            p.put(Context.PROVIDER_URL, "iiop://localhost");
            InitialContext ic = new InitialContext(p);
            Object homObj = ic.lookup("CartItemHome");
            CartItemHome cartItemHome = (CartItemHome)
                javax.rmi.PortableRemoteObject.narrow(
                    homObj, CartItemHome.class);
            item = cartItemHome.create(dish, qty);
            items.addElement(item);
        }
        catch (Exception e) {
            throw new RemoteException(e.toString());
        }
    }

    public void setDishQuantity(Dish dish, int quantity)
```

```
throws RemoteException
{
    CartItem item = getItem(dish);
    if (item != null)
        item.setQuantity(quantity);
}

public void clear()
{
    for (int i=0; i < items.size(); i++) {
        CartItem item = (CartItem) items.elementAt(i);
        try {
            item.remove();
        }
        catch (Exception e) {
            e.printStackTrace();
        }
    }
    items = new Vector();
}

public int getTotalPrice() throws RemoteException
{
    int total = 0;
    for (int i = 0; i < items.size(); i++) {
        try {
            CartItem item = (CartItem) items.elementAt(i);
            total += item.getPrice();
        }
        catch (Exception ex) {
            throw new RemoteException(ex.toString());
        }
    }
    return total;
}

public Order placeOrder() throws RemoteException
{
    Order order = null;
    try {
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming.WsnInitialContextFactory");
        p.put(Context.PROVIDER_URL, "iiop://localhost");
    }
}
```



```

        InitialContext ic = new InitialContext(p);
        Object homObj1 = ic.lookup("OrderItemHome");
        Object homObj2 = ic.lookup("OrderHome");
        Object homObj3 = ic.lookup("PricingManagerHome");
        OrderItemHome orderItemHome = (OrderItemHome)
            javax.rmi.PortableRemoteObject.narrow(
                homObj1, OrderItemHome.class);
        OrderHome orderHome = (OrderHome)
            javax.rmi.PortableRemoteObject.narrow(
                homObj2, OrderHome.class);
        PricingManagerHome pricingManagerHome =
            (PricingManagerHome)
            javax.rmi.PortableRemoteObject.narrow(
                homObj3, PricingManagerHome.class);
        PricingManager pm = pricingManagerHome.create();
        String oid = Long.toString(System.currentTimeMillis())
+
            customer.getPrimaryKey().toString();
        order = orderHome.create(oid, customer);

        order.setDeliveryCost(
            pm.getDeliveryCost(customer.getAddress()));
        Vector orderItems = new Vector();
        for (int i=0; i < orderItems.size(); i++) {
            CartItem item = (CartItem) orderItems.elementAt(i);
            Dish dish = item.getDish();
            String id = oid + dish.getPrimaryKey().toString();
            OrderItem oitem = orderItemHome.create(id,
                order, dish,
                item.getQuantity(), item.getDiscount());
        }
    }
    catch (Exception ex) {
        throw new RemoteException(ex.toString());
    }
    return order;
}
}

```

CartItem 会话 bean

Cart bean 使用了代表购物车中一个单独项目的 CartItem bean。这个 bean 保存了购物车中一个单独表项的相关数据。这些数据与食物碟及其数量相关。这个 bean 提供了一个方法，用来获取食物碟价格，这是通过将方法调用委托给 Dish bean 完成的。CartItem bean 的远程接

口与本地接口如下：

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface CartItem extends EJBObject
{
    public Dish getDish() throws RemoteException;
    public int getPrice() throws RemoteException;
    public int getDiscount() throws RemoteException;
    public void setDiscount() throws RemoteException;
    public int getQuantity() throws RemoteException;
    public void setQuantity(int quantity) throws RemoteException;
}
```

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface CartItemHome extends EJBHome {
    public CartItem create(Dish dish, int qty)
        throws CreateException, RemoteException;
}
```

这个 bean 的实现也不是很有意思，并且主要是一些 get/set 方法与 EJB 框架方法；因此，在这里将忽略它。

PricingManager 会话 bean

PricingManager bean 是一个仅提供业务逻辑的会话 bean。在本例中，这个 bean 负责计算订单的总价格（并且应用任何可用的打折）并计算送货到户费。在这个 bean 中实现的业务逻辑被大大简化。对于超过 100 000 日元的订单，将使用%5 的打折，并且在东京范围之内，不收取送货到户费（对于在东京之外的地方，收取 1000 日元的送货到户费）。这些“业务规则”仅仅是关于在会话 bean（例如 PricingManager）中能够完成的工作的演示。显然，在现实的应用程序中所实现的规则将更加复杂。PricingManager 的远程接口如下：

```
package com.onamix;
import javax.ejb.*;
import java.util.*;
import java.rmi.RemoteException;

public interface PricingManager extends EJBObject
{
}
```

```
        public int getDeliveryCost(String address) throws RemoteException;
        public void determinePrice(Cart order) throws RemoteException;
    }
```

这个 bean 的本地接口仅仅包括 create 方法：

```
package com.onamix;
import javax.ejb.*;
import java.rmi.RemoteException;

public interface PricingManagerHome extends EJBHome
{
    public PricingManager create()
        throws CreateException, RemoteException;
}
```

这个 bean 实现了用于计算费用的逻辑：

```
package com.onamix;

import javax.naming.*;
import javax.ejb.*;
import java.util.*;
import java.io.*;
import javax.naming.*;
import java.rmi.*;

public class PricingManagerBean implements SessionBean
{
    public void setSessionContext(SessionContext ctx) {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbCreate() {}
    public void ejbRemove() {}
    public int getDeliveryCost(String address)
    {
        if (address.indexOf("Tokyo") >= 0)
            return 0;
        return 1000;
    }

    public void determinePrice(Cart order) throws RemoteException
    {
        try {
            int price = order.getTotalPrice();
        }
    }
}
```

```

        if (price > 100000)
            order.setDiscount(10);
        else
            order.setDiscount(0);
    }
    catch (Exception ex) {
        throw new RemoteException(ex.toString());
    }
}

```

一种可以用来替代 `determinePrice()` 方法中所使用的“if 块”的可选方案就是使用 Enterprise Extension 所提供的业务规则 bean。这样，就可以基于业务需求在运行时修改业务规则。会话 bean 的事务行为是简单的，因为它与会话相关。因此，不需要考虑任何与并发性相关的问题。事务行为可以被设置为 `TX_SUPPORTS`，并且使用最低的隔离级别 (`TRANSACTION_READ_UNCOMMITTED`)。

34.3.4 Onamix 表示层

Onamix 表示层是一个 Web 应用程序，它为顾客提供了一个用户界面，通过这个用户界面，顾客就能够浏览菜单并且放置订单。表示层的所有元素都打包在 `root/onamixapp` 下的一个模块中。为了能够使用第 5 部分与第 6 部分讨论的所有元素，在这里选择使用 JSP 与 servlet 的混合。这种混合在一定程度上是任意的，并且有些 servlet 包括了嵌入的 HTML。一般来说，应该避免这种用法，并且将登录划分到一个 servlet 与一个 JSP 中，正如在第 5 部分讨论的那样。

主页

主页实现为一个 JSP，它使用 `DishHome` bean 获取并显示当天的所有特价商品：

```

<%@ page import="java.util.*" %>
<%@ page import="javax.ejb.*" %>
<%@ page import="javax.naming.*" %>
<%@ page import="com.onamix.*" %>
<html>
<head>
<title>ONAMIX</title>
</head>
<body>
<p><b><H2><i>ONAMIX</i></H2></b> </p>
<p>Welcome!</p>
<p>Please <a
href="http://onamix.com/onamixapp/LoginServlet">Login</a></p>

```

```

<br><br>
<%= new Date() %><br>
<H3>Today's specials: </H3><br>
<br>
<%
try{
    Properties p = new Properties();
    p.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.websphere.naming.WsnInitialContextFactory");
    p.put(Context.PROVIDER_URL, "iiop://localhost");
    InitialContext ic = new InitialContext(p);
    Object homObj = ic.lookup("DishHome");
    DishHome dishHome = (DishHome)
        javax.rmi.PortableRemoteObject.narrow(
            homObj, DishHome.class);
    Enumeration e = dishHome.findBySpecials();
    while (e.hasMoreElements()) {
        Dish d = (Dish) e.nextElement();
        out.println("<p>" + d.getName() + "</p>");
    }
}
catch (Exception ex) {
    out.println("Error " + ex.toString());
}
%>
</body>
</html>

```

LoginServlet servlet

LoginServlet servlet 用于用户登录。这个 servlet 要求用户提供用户名与口令。在用户被认证之后，将创建一个新的购物车会话 bean，并且将这个会话 bean 与当前 HTTP 会话相关联。需要说明的是，在所有的 servlet 中，都是在 init 方法上构建所需的本地对象。这个方法仅仅被调用一次，并且，其中应该仅仅包含与 servlet 的所有线程相关的代码。通过存储本地对象，就能够节省到达 servlet 的每个请求的处理时间。

```

package com.onamix;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;

public class LoginServlet extends HttpServlet {

```

```

    private CustomerHome customerHome;
    private CartHome cartHome;
    public void init(ServletConfig config) throws ServletException {
        super.init(config);
        try {
            Properties p = new Properties();
            p.put(Context.INITIAL_CONTEXT_FACTORY,
                "com.ibm.websphere.naming." +
                "WsnInitialContextFactory");

            p.put(Context.PROVIDER_URL, "iiop://localhost");
            InitialContext ic = new InitialContext(p);
            Object homObj = ic.lookup("CustomerHome");
            customerHome = (CustomerHome)
                javax.rmi.PortableRemoteObject.narrow(
                    homObj, CustomerHome.class);
            Object homObj2 = ic.lookup("CartHome");
            cartHome = (CartHome)
                javax.rmi.PortableRemoteObject.narrow(
                    homObj2, CartHome.class);
        }
        catch (Exception e) {
            throw new ServletException(e.toString());
        }
    }

    private void sendOutForm(HttpServletResponse response) throws IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><HEAD><TITLE>ONAMIX</TITLE></HEAD>" +
            "<BODY><H1><I>ONAMIX Login</I><BR></H1>");
        out.println("<P><B>Enter Username and Password</B><P><BR>");
        out.println("<form action=onamix.com/onamixapp/LoginServlet" +
            "\" method=\"get\">" +
            "<BR><B>Name:</B><input type=\"text\" name=\"login\" " +
            "value=\"\" size=\"30\"><BR><B>Password:</B>" +
            "<input type=\"text\" name=\"password\" " +
            "value=\"\" size=\"30\"><BR><input type=\"submit\" " +
            "value=\"Login\"></form>");
        out.println("</BODY></HTML>");
        out.close();
    }
}

```

```
public void doPost (HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException
{
    HttpSession session = request.getSession(true);
    String ID = request.getParameter("id");
    String name = request.getParameter("name");
    String passwd = request.getParameter("password");
    String address = request.getParameter("address");
    String deliveryInst = request.getParameter("deliveryInst");
    String phone = request.getParameter("phone");
    try {
        Customer c = customerHome.create(ID, name, passwd, address,
            deliveryInst, phone);
        Cart cart = cartHome.create(c);
        session.putValue("cart", cart);
        response.sendRedirect(response.encodeUrl("/MenuServlet"));
        sendOutForm(response);
    }
    catch (Exception e) {
        throw new ServletException(e.toString());
    }
}

public void doGet (HttpServletRequest request,
                    HttpServletResponse response)
                    throws ServletException
{
    HttpSession session = request.getSession(true);
    String login = request.getParameter("login");
    String password = request.getParameter("password");

    try {
        Enumeration customers = null;
        Customer c = customerHome.findByPrimaryKey(new
CustomerPK(login));
        String realPassword = c.getPassword();
        if (realPassword.equals(password)) {
            Cart cart = cartHome.create(c);
            session.putValue("cart", cart);
            response.sendRedirect(response.encodeUrl("/MenuServlet"));
            return;
        }
        sendOutForm(response);
    }
}
```

```

        catch (Exception e) {
            throw new ServletException(e.toString());
        }
    }
}

```

Registration 页面 (注册页面)

注册页面允许新顾客进行注册。在注册之后，用户就自动登录到网站中——也就是说，注册页面将调用 login servlet (登录 servlet)。

```

<HTML><HEAD><TITLE>ONAMIX</TITLE></HEAD>
<BODY><H1><I>ONAMIX REGISTRATION</I><BR></H1>
<form action=onamix.com/onamixapp/LoginServlet" method="post">
<BR><B>ID Nr. (used for Login)</B>
<input type="text\" name="id" value="" size="30">
<BR><B>Name:</B>
<input type="text\" name="name" value="" size="30">
<BR><B>Password:</B>
<input type="text\" name="password" value="" size="30">
<BR><B>Address:</B>
<input type="text\" name="address" value="" size="30">
<BR><B>Delivery Instructions:</B>
<input type="text\" name="deliveryinst" value="" size="30">
<BR><B>Phone:</B>
<input type="text\" name="phone" value="" size="30">
<BR><input type="submit" value="Submit"></form>
</BODY></HTML>

```

MenuServlet servlet

MenuServlet 允许用户浏览菜单并查看项目。这个 servlet 使用 DishGroup bean 来显示菜单部分。

```

package com.onamix;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;

public class MenuServlet extends HttpServlet
{
    private DishGroupHome dishGroupHome;
    public void init(ServletConfig config) throws ServletException {
        try {

```



```

        super.init(config);
        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming.WsnInitialContextFactory");
        p.put(Context.PROVIDER_URL, "iiop://localhost");
        InitialContext ic = new InitialContext(p);
        Object homObj = ic.lookup("DishGroupHome");
        dishGroupHome = (DishGroupHome)
            javax.rmi.PortableRemoteObject.narrow
            homObj, DishGroupHome.class};
    }
    catch (Exception e) {
        throw new ServletException(e.toString());
    }
}

public void doGet (HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException
{
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    String groupName = request.getParameter("group");
    out.println("<HTML><HEAD><TITLE>ONAMIX</TITLE></HEAD>" +
        "<BODY><H1><I>ONAMIX MENU</I><BR></H1>");
    try {
        if (groupName == null) {
            Enumeration groups = dishGroupHome.findAllGroups();
            while (groups.hasMoreElements()) {
                DishGroup group = (DishGroup)
                    groups.nextElement();
                printOutGroup(group, out);
            }
        }
        else {
            DishGroup group =
                dishGroupHome.findByPrimaryKey(new
                    DishGroupPK(groupName));
            printOutGroup(group, out);
        }
    }
    catch (Exception e) {
        throw new ServletException(e.toString());
    }
    out.println("</BODY></HTML>");
}

```

```

        out.close();
    }
    public void printOutGroup(DishGroup group, PrintWriter out)
        throws IOException
    {
        out.println("<BR><H3>" + group.getName()
            + "</H3>");
        Enumeration items = group.getElements();
        while (items.hasMoreElements()) {
            Dish dish = (Dish) items.nextElement();
            out.println("<BR><a href=\"\" +
                \"http://onamix.com/onamixapp/DishDetail?\" +
                \"dish=\" + dish.getName() +
                \"\">\" + dish.getName() + "</a>");
        }
    }
}

```

Dish 页面

下面的 JSP 显示了特定食物碟的详细信息。menu servlet 为所选择的食物碟调用这个 JSP。这个 JSP 使用此 bean 获取所有的信息，并且添加一个图像到食物碟。

```

<%@ page import="java.util.*" %>
<%@ page import="javax.ejb.*" %>
<%@ page import="javax.naming.*" %>
<%@ page import="com.onamix.*" %>
<html>
<head>
<title>ONAMIX</title>
</head>
<body>
<%
try{
    Properties p = new Properties();
    p.put(Context.INITIAL_CONTEXT_FACTORY,
        "com.ibm.websphere.naming.WsnInitialContextFactory ");
    p.put(Context.PROVIDER_URL, "iiop://localhost");

    InitialContext ic = new InitialContext(p);
    Object homObj = ic.lookup("DishHome");
    DishHome dishHome = (DishHome)
        javax.rmi.PortableRemoteObject.narrow(
            homObj, DishHome.class);
    String dishName = request.getParameter("dish");

```

```
String dishGroupName = request.getParameter("dishgroup");
Dish dish = dishHome.getPrimaryKey(new DishPK(dishName));
}
catch (Exception ex) {
    out.println("Error " + ex.toString());
}
%>
<form action="onamix.com/onamixapp/CartServlet?operation=add+dish="
<%= dish.getName() %>
method="get">
<p><b><font size="6"><i>ONAMIX</i></font></b> </p>
<p><a href="http://onamix.com/onamixapp/MenuServlet">Menu</a> :
<a href="http://onamix.com/onamixapp/MenuServlet?group=<%= dishGroupName
%>"><%= dishGroupName %></a>
: </p>
<p></p>
<p><b><%= dish.getName()%>
</b>&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
<%= dish.getPrice() %>¥
<input type="button" value="Add To Cart" name="B1"></p>
<p><i><%= dish.getDescription()%></i></p>
</body>
</html>
```

CartServlet servlet

CartServlet 使用 CartItem bean 显示购物车。它也使用价格管理器计算将要显示的价格。

```
package com.onamix;
import java.io.*;
import java.util.*;
import java.text.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;

public class CartServlet extends HttpServlet
{
    private PricingManagerHome pricingManagerHome;
    private DishHome dishHome;
    public void init(ServletConfig config) throws ServletException {
        super.init(config);

        try {
```

```

        Properties p = new Properties();
        p.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.ibm.websphere.naming." +
            "WsnInitialContextFactory");
        p.put(Context.PROVIDER_URL, "iiop://localhost");
        InitialContext ic = new InitialContext(p);
        Object homObj = ic.lookup("PricingManager");
        pricingManagerHome = (PricingManagerHome)
            javax.rmi.PortableRemoteObject.narrow(
                homObj, PricingManagerHome.class);
        Object homObj1 = ic.lookup("PricingManager");
        dishHome = (DishHome)
            javax.rmi.PortableRemoteObject.narrow(
                homObj1, DishHome.class);
    }
    catch (Exception e) {
        throw new ServletException(e.toString());
    }
}

public void doPost (HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
    try {
        PrintWriter out = response.getWriter();
        HttpSession session = request.getSession(false);
        Cart cart = (Cart) session.getValue("quote");
        Enumeration e = request.getParameterNames();
        while (e.hasMoreElements()) {
            String paramName = (String) e.nextElement();
            if (paramName.startsWith("updateqty")) {
                int ind = Integer.parseInt(paramName.substring(8));
                int val =
                    Integer.parseInt(request.getParameter(paramName));
                Enumeration items = cart.getItems();
                int count = 0;
                while (items.hasMoreElements()) {
                    CartItem cartItem = (CartItem)
                        items.nextElement();
                    if (count == ind)
                        cartItem.setQuantity(val);
                    count++;
                }
            }
        }
    }
}

```

```

        PricingManager pm = pricingManagerHome.create();
        pm.determinePrice(cart);
        response.sendRedirect(response.encodeURL("/CartServlet"));
    }
    catch (Exception ex) {
        throw new ServletException(ex.toString());
    }
}

public void doGet (HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();
    try {
        HttpSession session = request.getSession(false);
        if (session == null || session.getValue("cart") == null) {
            response.sendRedirect(
                response.encodeURL("/LoginServlet"));
            return;
        }
        Cart cart = (Cart) session.getValue("quote");
        out.println("<HTML><HEAD><TITLE>ONAMIX</TITLE></HEAD>" +
            "<BODY><H1><I>ONAMIX Shopping" +
            " Cart</I><BR></H1>");
        String operation = request.getParameter("operation");
        String dishName = request.getParameter("dish");
        if (operation == null) {
            out.println(
                "<form action=\"" +
                response.encodeURL("/CartServlet") +
                "\" method=\"" + "put\" ">" +
                "<TABLE><TR><TH> Quantity </TH>" +
                "<TH> Name </TH><TH> Price </TH></TR>");
            Enumeration items = cart.getItems();
            int i = 0;
            while (items.hasMoreElements()) {
                CartItem cartItem = (CartItem)
                    items.nextElement();
                Dish dish = cartItem.getDish();
                out.println(
                    "<TR>" +
                    "<TD><input type=\"" + "text\" " + "name=\"" +
                    "updateqty" + i +

```

```

        "\"" value=\"\" +
        cartItem.getQuantity() +
        "\"">\" +
        "</TD>\" + dish.getName() + "</TD>\" +
        "</TD>\" + dish.getPrice() +
        "¥</TD></TR>\";
        i++;
    }
    out.println("</TABL><EB>RTOTAL:\" +
        cart.getTotalPrice() + "¥");
    out.println(
        "<BR><a href=\"
        \"http://onamix.com/onamixapp +
        \"/OrderServlet\">\" +
        \"Place Order</A>\" +
        "<BR><input type=\"submit\" \" +
        \"value=\"Update \" +
        \" Quantities\">\"");
    }
    else if (operation.equals("empty")) {
        cart.clear();
        out.println("<H2><I>Your cart is now empty</I></H2>");
    }
    else if (operation.equals("add")) {
        Dish dish = dishHome.findByPrimaryKey(new
DishPK(dishName));
        cart.addDish(dish);
        PricingManager pm = pricingManagerHome.create();
        pm.determinePrice(cart);
        response.sendRedirect(
            response.encodeURL("/CartServlet"));
    }
    }
    catch (Exception e) {
        throw new ServletException(e.toString());
    }
    out.println("</body> </html>");
    out.close();
}
}

```

OrderServlet servlet

OrderServlet **负责处理订货过程。它从当前购物车中创建一个持续性订单。**

```
package com.onamix;
import java.io.*;
import java.util.*;
import java.text.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.naming.*;

public class OrderServlet extends HttpServlet
{
    public void doGet (HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        try {
            HttpSession session = request.getSession(false);
            if (session == null || session.getValue("cart") == null) {
                response.sendRedirect(response.encodeURL("/LoginServlet"));
                return;
            }
            Cart cart = (Cart) session.getValue("quote");
            cart.placeOrder();
            out.println("<HTML><HEAD><TITLE>ONAMIX</TITLE></HEAD>" +
                        "<BODY><H1><I>ONAMIX Order</I><BR></H1>");
            out.println("<H2><I>Your order has been successfully" +
                        "placed</I></H2>");
        }
        catch (Exception e) {
            throw new ServletException(e.toString());
        }
        out.println("</body> </html>");
        out.close();
    }
}
```

34.4 小 结

本章给出了一个简单电子交易应用程序的完全示例，这个应用程序使用了 servlet、JSP、会话 bean 与实体 bean。其中的代码段比较长，但很简单，相信你也会这样认为。并且，虽然由于复杂的业务逻辑可能会导致应用程序的复杂性，但 WebSphere 所实现的技术框架是清

晰、干净与有用的。

本章是本书第 6 部分的最后一章，实际上，它也可以看成是第 5 部分与第 6 部分的“总结性工程”。

第 7 部分 使用 XML

第 35 章 编写 XML 电子商务应用程序

欢迎进入第 7 部分的学习。本书第 6 部分讨论了 EJB，它是 WebSphere 应用程序的重要技术基础。这一部分将讨论 XML（Extensible Markup Language，可扩展标记语言），它是 Web 应用程序中另一个重要的技术。关于 XML 的介绍，请参见第 12 章。

我们将通过一些简单的示例（首先给出的是一些在应用程序中使用 XML 的示例）演示 XML 以及相关的技术。第 36 章使用 XSLT（XSL Transformation）对打包在 XML 文档中的数据进行处理。第 37 章与第 38 章重点描述相对较新的技术（Web 服务），这些技术基于 XML，并且有可能从根本上改变我们使用业务系统的方式。

正如第 12 章描述的那样，XML 被许多软件专家看成是用来开发电子商务应用程序（在消费市场以及 business-to-business 市场）的最重要技术之一。XML 是一种相对较新的技术，因此，目前在市场上还没有见到关于这种技术的可能应用程序；实际上，业界还没有定义这种技术所适用的目标应用类型。尽管如此，还是存在一些应用背景以 XML 技术为主体，包括集成、数据交换与存储等等。业界已经开发了许多用来使用 XML 的常见工具，包括 XML 解析程序、DOM 结构与 SAX（Simple API for XML）。

本章将开发一个完整的应用程序，用来说明这些内容的使用，并且演示如何使用 WebSphere 中提供的 XML 工具。此外，还将给出一个来自更大业务应用程序的例子，这个应用程序使用 XML 进行表示层转码（transcoding）。

35.1 内线交易应用程序

本章首先将构建一个应用程序，用来满足人们在股票市场上新发现的输钱爱好。许多人都有这种谬误的想法，认为只要根据“内线”的行为就能够确定应该购买哪支股票。但由于存在数量众多的股票，人们很难手工地检查内线购买了哪些股票。因此，这里讨论的应用程序能够执行一个自动扫描，以确定哪个公司的内线决定购买该公司的股票。这里的假设是，如果内线使用自己的钱购买某支股票，那么他们必然知道某些不为人知的事实，并且我们也可以购买这支股票。

这个应用程序能够对 NASDAQ 交易所列出的公司进行工作,并且使用两个主要的信息来源。第一个信息来源就是来自 NASDAQ 网站的信息,用来确定在这个交易所中所有的股票代码。图 35-1 显示了一个列出 NASDAQ 上市公司的典型的 Web 页面。因此,这个应用程序需要做的第一件工作就是访问 NASDAQ 网站,查找其中所有的页面并从中提取出股票代码。

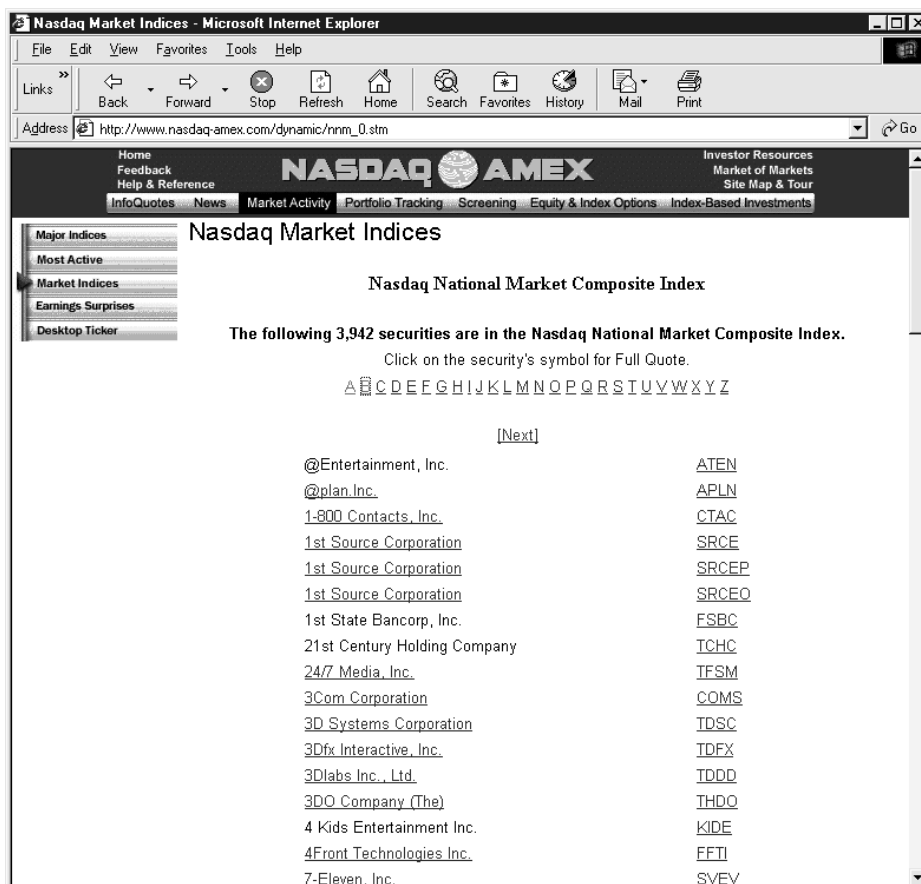


图 35-1 NASDAQ 上市公司

在获得了所有的股票代码之后,就将访问另一个数据源。Yahoo! Finance 部分列出了任何股票代码的内线交易信息。图 35-2 显示了一个页面例子。Yahoo!数据源为每个股票代码列出了所有的内线交易(我们所选择的这个页面显示了许多内线购买交易)。从这个 URL (<http://biz.yahoo.com/t/i/IACO.html>) 可以知道,只要给定了一个股票代码(例如 MSFT, 代表 Microsoft),就可以知道需要访问的 URL 是 <http://biz.yahoo.com/t/m/MSFT.html>。因此,只要有了所有的股票代码(在第一个阶段得到),就能够访问所有这些页面。

Date	Name	Action	Shares	Type	Price	Value
30-Jun-99	GAARD MICHAEL B	Bought (P)	9,500	Common	4.09	\$38,855
30-Jun-99	GAARD MICHAEL B	Acquired (B)	1,000	Common	3.48	-
30-Jun-99	BOEHME CHERIL L	Acquired (B)	1,000	Common	3.48	-
30-Jun-99	PARKER RICHARD S	Acquired (B)	1,000	Common	3.48	-
30-Jun-99	TERRIEN RORY C	Acquired (B)	1,000	Common	3.48	-
30-Jun-99	TRICK MARY K	Acquired (B)	1,000	Common	3.48	-
30-Jun-99	FORD LARRY J	Acquired (B)	1,000	Common	3.48	-
30-Jun-99	FURTNEY MARK	Acquired (B)	1,000	Common	3.48	-
30-Jun-99	FURTNEY MARK	Holdings	200 (j)	Common	-	-
30-Jun-99	PEDERSON ROBIN	Acquired (B)	1,000	Common	3.48	-
25-Jun-99	GAARD MICHAEL B	Planned Sale (144)	20,000	Common	-	\$77,391
25-Jun-99	GAARD MICHAEL B	Bought (P)	5,000	Common	3.69	\$18,450
2-Jun-99	GAARD MICHAEL B	Bought (P)	5,000	Common	4.00	\$20,000
2-Jun-99	PARKER RICHARD S	Exercised (M)	32,300	Common	1.13	-
2-Jun-99	PEDERSON ROBIN	Exercised (M)	36,316	Common	1.13	-

图 35-2 Yahoo! Finance 内线交易信息

下面就需要对这些页面进行分析，并且在 Bought 或 Acquired 部分查找表项，如果发生了交易动作，就从其中提取数据并且提取股票交易时的价格。然后，所有的结果都被打包并且写出作为一个 XML 文档，在后来的某个时刻，可以在另一个应用程序中查看或使用这个 XML 文档。所产生的文件是一个 XML 文档，这就使得在另一个应用程序中使用起来非常简单；这也是 XML 相对于传统的格式所表现出来的主要优点之一。

应用程序的所有部分都被编写为独立的。这样，第一个模块将访问 NASDAQ 页面，并以 XML 文档的形式输出股票代码。第二个应用程序模块解析这个文档并且找出所有的代号。将这两个过程分离到两个应用程序中进行，除了是一种良好的软件工程实践之外，也能够用于演示 XML 工具的使用。

在这里使用 IBM 的 Java XML Parser，它是 WebSphere 的一部分，用来解析 XML 文档、构建 DOM 树以及访问信息。在第一个例子中，我们将使用这个包来构建 DOM 结构并将它们输出为 XML。TreeViewer（见图 35-3）应用程序可用来在调试过程中查看 XML 文档。

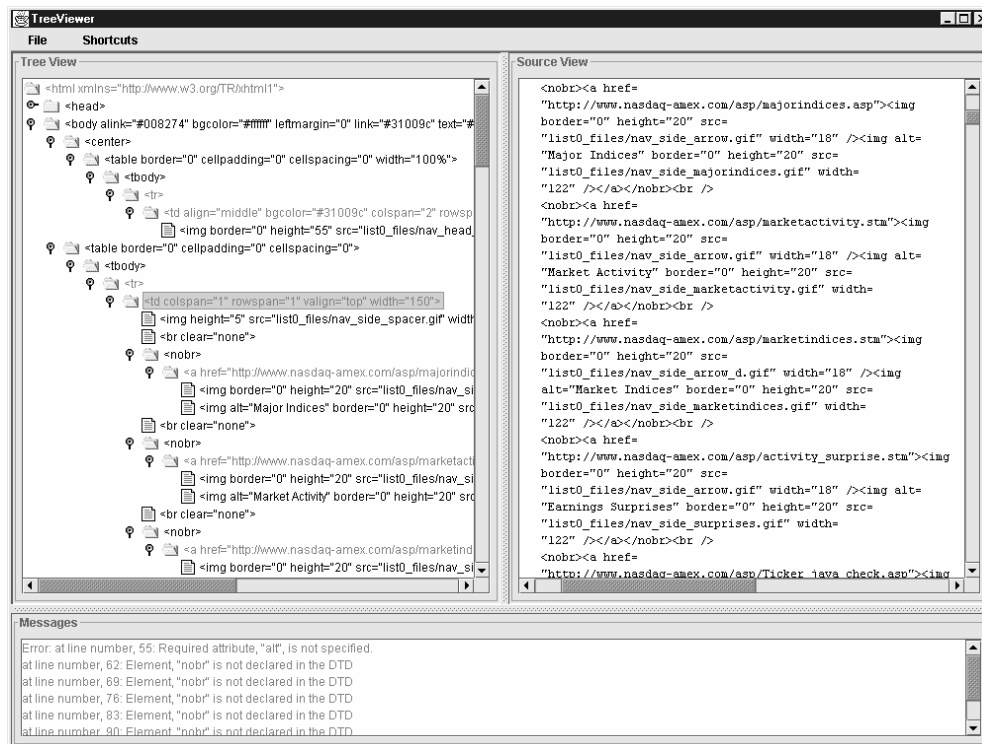


图 35-3 TreeViewer 应用程序

35.1.1 生成股票代号

这个应用程序的第一部分很简单，如下所示：

```
import java.util.*;
import java.io.*;
import org.w3c.dom.*;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.*;
import org.xml.sax.*;

public class SymbolGenerator {

    public SymbolGenerator() {
    }

    public static void main(String[] args) {
```

```
SymbolGenerator gen = new SymbolGenerator();

Vector symbolVector = new Vector(1000);
for (int i = 0 ; i <= 50 ; i++) {
    // Get the HTML and process it
    StringBuffer htmlText = HTMLFetcher.getHTMLFor(
        "http://www.nasdaq-amex.com/dynamic/nnm_" +
        Integer.toString(i) + ".stm");
    gen.extractSymbolInformationInto(htmlText, symbolVector);
}
gen.outputSymbolInformation(symbolVector);
}

// Return a vector with the symbols that the page displays
void extractSymbolInformationInto(
    StringBuffer aBuf, Vector symbolVector) {
    int index = 0;
    String str = aBuf.toString();
    while (index != -1) {
        index = str.indexOf("symbol=", index);
        if (index != -1) {
            int nextIndex = str.indexOf(">", index);
            String aSymbol = str.substring(index+7, nextIndex);
            symbolVector.addElement(aSymbol);
            index = nextIndex + 1;
        }
    }
}

// Write an XML document by building an element per symbol
void outputSymbolInformation(Vector symbolVector) {
    com.ibm.xml.parser.TXDocument myDocument =
        new com.ibm.xml.parser.TXDocument();

    try {
        ProcessingInstruction pi = myDocument.
            createProcessingInstruction("xml", "version=\"1.0\"");
        myDocument.appendChild(pi);

        Element root = myDocument.createElement("stockSymbols");
        root.setAttribute("exchange", "NASDAQ");
        myDocument.appendChild(root);
    }
}
```

```

        for (Enumeration e = symbolVector.elements();
             e.hasMoreElements(); ) {
            String aSymbol = (String)e.nextElement();
            Element child = myDocument.createElement("symbol");
            Text t = myDocument.createTextNode(aSymbol);
            child.appendChild(t);
            root.appendChild(child);
        }

        FileWriter fw = new FileWriter("symbols.xml");
        myDocument.toXMLString(fw);
        fw.close();

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

```

这个部分对 NASDAQ 上列出所有代号的 50 个页面进行迭代，并且为每个代号获取 HTML 页面。然后，对这些 HTML 页面进行查找，以找到“代号”字符串出现的位置并提取出股票代码。为此，需要使用一个称为 HTMLFetcher 的实用类，本章的许多其他代码例子中都将重用这个类。这个类的代码如下所示：

```

import java.net.*;
import java.io.*;

public class HTMLFetcher {

    // Given a URL get the HTML page. If there is an exception
    // the method returns null. Use the net package to open a URLConnection
    // to get the HTML response
    static public StringBuffer getHTMLFor(String url) {
        try {
            URLConnection aConnection = (new URL(url)).openConnection();
            aConnection.setDoInput(true);
            aConnection.setDoOutput(true);
            aConnection.setUseCaches(false);
            InputStream inS = aConnection.getInputStream();
            StringBuffer buffer = new StringBuffer(4096);
            boolean atEnd = false;
            while(!atEnd) {
                int len = inS.available();
                byte[] buf = new byte[len];

```

```

        if (inS.read(buf, 0, len) == -1)
            atEnd = true;
        if (len > 0)
            buffer.append(new String(buf));
        Thread.sleep(10);
    }
    return buffer;
} catch (Throwable ex) {
    ex.printStackTrace();
}
return null;
}
}

```

下一步工作就是对向量中的所有代号进行迭代，并构建 XML 文件。在这里使用 DOM 类构建一个文档，然后将这个文档转储到一个 XML 文件中。构建文档的工作很简单，只要实例化节点并创建这些节点之间的关系即可。一旦该文档被创建，就可以使用 `toXMLString` 方法创建 XML 字符串并将这个字符串写出到一个文件中。图 35-4 显示了结果 XML 文件的一部分。

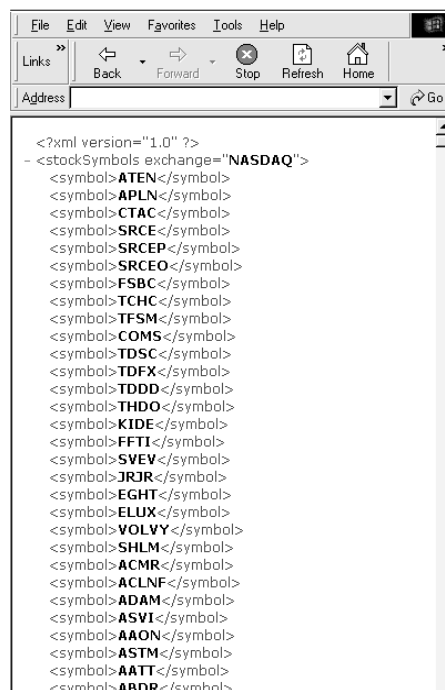


图 35-4 股票代号的 XML

35.1.2 获取内线信息

这个应用程序的第二个模块稍微复杂一点，其代码如下：

```
import java.util.*;
import java.io.*;

import org.w3c.dom.*;
import org.xml.sax.helpers.ParserFactory;
import com.ibm.xml.parsers.*;
import org.xml.sax.*;

public class InsiderInspector implements ErrorHandler{

    public static void main(String[] args) {

        InsiderInspector gen = new InsiderInspector();

        com.ibm.xml.parser.TXDocument myDocument =
            new com.ibm.xml.parser.TXDocument();

        try {
            ProcessingInstruction pi = myDocument.
                createProcessingInstruction("xml", "version=\"1.0\"");
            myDocument.appendChild(pi);

            Element root = myDocument.createElement("insiderTrading");
            root.setAttribute("asOf", (new Date()).toString());
            myDocument.appendChild(root);

            Vector symbolVector = gen.retrieveAllSymbols();
            for (Enumeration e = symbolVector.elements() ; e.hasMoreElements() ; )
            {
                String aSymbol = (String)e.nextElement();
                StringBuffer htmlText = HTMLFetcher.getHTMLFor(
                    "http://biz.yahoo.com/t/" + aSymbol.substring(0,1) +
                    "/" + aSymbol + ".html");
                StringBuffer aBuf = HTMLTidier.tidyHTML(htmlText);
                if (aBuf != null) {
                    Element symbolRoot = myDocument.createElement(aSymbol);
                    root.appendChild(symbolRoot);
                    gen.extractInsiderInfo(aBuf, myDocument, symbolRoot);
                }
            }
        }
```



```
    }

    FileWriter fw = new FileWriter("insiderTrading.xml");
    myDocument.toXMLString(fw);
    fw.close();
} catch (Exception ex) {
    ex.printStackTrace();
}
}

Vector retrieveAllSymbols() {
    Vector symbols = new Vector(4096);
    BufferedReader br = null;
    try {
        br = new BufferedReader(new FileReader("symbols.xml"));
    } catch (Exception ex) {
        ex.printStackTrace();
        return null;
    }
    Parser parser = null;
    try {
        parser = ParserFactory.makeParser("com.ibm.xml.parsers.NonValidatingDOMParser");
        parser.setErrorHandler(this);
    } catch (Exception ex) {
        ex.printStackTrace();
        return null;
    }

    try
    {
        parser.parse(new org.xml.sax.InputSource(br));

        org.w3c.dom.Document d = ((NonValidatingDOMParser)parser).
        getDocument();

        org.w3c.dom.Element rootElement = d.getDocumentElement();
        NodeList nodes = rootElement.getChildNodes();
        for (int i = 0 ; i < nodes.getLength() ; i++)
        {
            org.w3c.dom.Node aNode = nodes.item(i);
            if ((aNode instanceof org.w3c.dom.Element) &&
                aNode.getNodeName().equalsIgnoreCase("symbol")) {
                org.w3c.dom.Element aSymbolElement = (org.w3c.dom.Element)aNode;
```

```
        String symbolName =
            ((org.w3c.dom.Text)aSymbolElement.getFirstChild()).getData();
        symbols.addElement(symbolName);
    }
}
} catch (Throwable ex) {
    ex.printStackTrace();
}
return symbols;
}

void extractInsiderInfo(
    StringBuffer aBuf,
    com.ibm.xml.parser.TXDocument myDocument,
    Element symbolRoot) {
    Parser parser = null;
    try {
        parser = ParserFactory.makeParser(
            "com.ibm.xml.parsers.NonValidatingDOMParser");
        parser.setErrorHandler(this);
    } catch (Exception ex) {
        ex.printStackTrace();
    }

    try {
        parser.parse(new org.xml.sax.InputSource(
            new ByteArrayInputStream(aBuf.toString().getBytes())));
        org.w3c.dom.Document d = ((NonValidatingDOMParser)parser).
getDocument();
        org.w3c.dom.Element inputElement = d.getDocumentElement();
        recursivelyDescend(inputElement, myDocument, symbolRoot);
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

void recursivelyDescend(
    org.w3c.dom.Node aNode,
    com.ibm.xml.parser.TXDocument myDocument,
    Element symbolRoot) {
    if (aNode instanceof org.w3c.dom.Text) {
        String theText = ((org.w3c.dom.Text)aNode).getData().trim();
        if (theText.length() >= 6) {
            String subS = theText.substring(0,6);
```

```
if ((subS.compareTo("Bought") == 0) ||
    (subS.compareTo("Acquir") == 0)) {
    Node aParent = aNode.getParentNode().getParentNode();
    Element newElement = myDocument.createElement("insiderPurchase");

    // This processing is very tightly coupled with the XML
    // structure and normally a more flexible pattern would
    // be used - but it serves the purpose of illustrating the
    // relationship between the XML document and the Java APIs
    try {
        Node dateNode = aParent.getFirstChild().getNextSibling().
            getFirstChild();
        newElement.setAttribute(
            "date",
            ((Text)dateNode).getData().trim());
        Node priceNode = aParent.getFirstChild().getNextSibling().
            getNextSibling().getNextSibling().
            getNextSibling().getNextSibling().getNextSibling().
            getNextSibling().getNextSibling().getFirstChild();
        newElement.setAttribute(
            "price",
            ((Text)priceNode).getData().trim());
    } catch (Throwable ex) {
        newElement.setAttribute(
            "error",
            "There was an inside trade that could not be parsed");
    }

    symbolRoot.appendChild(newElement);
}
}
}
NodeList nodes = aNode.getChildNodes();
for (int i = 0 ; i < nodes.getLength() ; i++) {
    org.w3c.dom.Node subNode = nodes.item(i);
    recursivelyDescend(subNode, myDocument, symbolRoot);
}
}

public void warning(SAXParseException exception) throws SAXException {
    // In our case we can ignore any parsing exception since it is
    // not part of the elements we need to process
}
```

```

    public void error(SAXParseException exception) throws SAXException {
        // In our case we can ignore any parsing exception since it is
        // not part of the elements we need to process
    }

    public void fatalError(SAXParseException exception) throws SAXException {
        // In our case we can ignore any parsing exception since it is
        // not part of the elements we need to process
    }

}

```

首先，读取带有代号信息的 XML 文件。这一过程比较简单，因为已经知道了这个 XML 文件的结构，因此一般来说不会出错。在获取了代号之后，就对所有的代号进行迭代，并且为每个代号打开一个到达 Yahoo! 网站的连接，然后使用 `HTMLFetcher` 类获取 HTML。

下一步就比较有意思。在这里不是像前一部分那样遍历 HTML 源，而是将这个 HTML 文件看作一个 XML 文档，并且使用文档节点遍历这个 XML 文档。在以后，将 HTML 页面当作明确定义的 XML 文档这种处理方法将很常用——目前，将这些页面当作 XML 文档进行处理还比较困难，因为 Web 上的大多数 HTML 页面都不是按照 XML 文档的要求构建的。

因此，首先需要将这些 HTML 页面转换成遵守 XML 规则的页面。在这里使用了一个称为 TIDY（由 Hewlett Packard 的 Dave Raggett 创建）的程序，该程序可以从 <http://www.w3.org/People/Raggett/tidy> 下载。这个应用程序能够对 HTML 文档进行修改，从而使该文档能够被 XML 解析程序所解析。下列代码显示了可完成这一工作的实用类，当然，这并不是一个非常优秀的方法，但它能够完成该工作。

```

import java.io.*;

public class HTMLTidier {

    public HTMLTidier() {
    }

    public static StringBuffer tidyHTML(StringBuffer inBuf) {

        try {
            File outputFile = new File("insiders.html");
            FileWriter out = new FileWriter(outputFile);
            out.write(inBuf.toString());
            out.close();
        } catch (Throwable ex) {

```

```
        ex.printStackTrace();
        return new StringBuffer(32);
    }

    String[] cmdLine = {
        "tidy.exe", "-f", "errs.txt", "-im", "-asxml", "insiders.html"};
    try {
        Runtime.getRuntime().exec(cmdLine);
        // Yes - it is ugly to wait a few seconds
        // but it's just a demo after all
        Thread.sleep(5000);
    } catch (Throwable ex) {
        ex.printStackTrace();
        return new StringBuffer(32);
    }

    try {
        File inputFile = new File("insiders.html");
        FileReader in = new FileReader(inputFile);
        int justRead = 0;
        int index = 0;
        StringBuffer aBuf = new StringBuffer(4096);
        while(justRead != -1) {
            char[] buf = new char[1024];
            justRead = in.read(buf, index, 1024);
            aBuf.append(buf);
        }
        in.close();
        return aBuf;
    } catch (Throwable ex) {
        ex.printStackTrace();
    }
    return null;
}
}
```

在创建一个构造合理的 XML 文档之后，就可以通过解析程序运行这个文档，并且查找在其中有 Bought 或 Acquired 的 Text 节点。在这里，需要查找的是带有日期信息与价格信息的兄弟节点（每个这样的节点都有图 35-5 中突出显示的节点所示的结构），并且将它添加到 XML 文档中。

图 35-6 显示了最终结果（在检查所有可能的股票代号之后）。显然，这需要运行比较长的时间，但它能为第二天早上的交易提供帮助。让我们一起去赚钱吧！



图 35-5 带有 Text, Date 与 Price 的 DOM 结构

```

- <insiderTrading example="1">
- <ATEN>
  <insiderPurchase date="27-Jan-99" price="-" />
  <insiderPurchase date="27-Jan-99" price="-" />
  <insiderPurchase date="27-Jan-99" price="-" />
  <insiderPurchase date="4-Nov-98" price="7.25 - 9.00" />
  <insiderPurchase date="2-Nov-98" price="7.25" />
  <insiderPurchase date="2-Nov-98" price="7.25" />
  <insiderPurchase date="22-Oct-98" price="3.94" />
  <insiderPurchase date="22-Oct-98" price="4.00" />
  <insiderPurchase date="21-Oct-98" price="4.25" />
  <insiderPurchase date="20-Oct-98" price="4.31 - 4.38" />
  <insiderPurchase date="19-Oct-98" price="4.63" />
  <insiderPurchase date="14-Oct-98" price="4.75" />
  <insiderPurchase date="15-Sep-98" price="10.38" />
  <insiderPurchase date="14-Sep-98" price="9.88" />
  <insiderPurchase date="4-Sep-98" price="9.19" />
  <insiderPurchase date="31-Aug-98" price="8.80" />
  <insiderPurchase date="31-Aug-98" price="8.63 - 8.75" />
  <insiderPurchase date="4-Aug-98" price="-" />
  <insiderPurchase date="4-Aug-98" price="-" />
  <insiderPurchase date="4-Aug-98" price="-" />
  <insiderPurchase date="4-Aug-98" price="-" />
  <insiderPurchase date="4-Aug-98" price="-" />
  <insiderPurchase date="4-Aug-98" price="-" />
  </ATEN>
- <APLN />
- <CTAC>
  <insiderPurchase date="17-Mar-99" price="12.50 - 12.63" />
  <insiderPurchase date="30-Sep-98" price="5.94" />
  <insiderPurchase date="29-Sep-98" price="6.00" />
  <insiderPurchase date="9-Sep-98" price="6.50" />
  <insiderPurchase date="9-Sep-98" price="6.50" />
  <insiderPurchase date="26-Aug-98" price="5.75" />
  <insiderPurchase date="18-Aug-98" price="6.06" />
  <insiderPurchase date="14-Aug-98" price="6.00" />
  <insiderPurchase date="13-Aug-98" price="6.06" />
  </CTAC>

```

图 35-6 内线交易 XML

35.2 转码方案

本节提供一个来自表示级转码领域的例子。这个例子来自一个电子商务应用程序，需要为多种设备（包括 PC、PDA 与电话）提供应用程序屏幕显示。每个这样的设备都有一个浏览器（或微型浏览器），用于接收以 XML 的某种方言（HTML、WML（Wireless Markup Language，无线标记语言）与 CHTML（Compact HTML，压缩 HTML））打包的信息。

在这个例子中，假设应用程序能够产生合理构成的 XML，这种 XML 能够由 servlet、JSP 或任何其他组件生成。转码程序（transcoder）是一个用来解析 XML 的过滤器，并且应用一些简单的规则来创建另一个 XML，这个新创建的 XML 最终将被传递给设备。转码程序可用来过滤图像、删除空白、进行元素替换以及执行其他类似的表示级修改。下面给出一个例子来说明转码程序的使用，在通过 LAN 或 WAN 传递内容时，人们可能会在编写页面时留有很多空白，如下所示：

```
<html>

<head>

    <meta content="text/html"http-equiv="Content-Type">
    <title class="MultiLingual">
My Calls
    </title>
</head>
<body>

    <table>

        <tr>
            <td>
                <b>

                    Call:

                </b>
                <a
href="/servlet/RequestHandler;jsessionid=GVYTQ2IAAAAAC5YAAAARGUQ?task=
ActionDetails&action=runQuery&skipLogin=yes&callId=3529&actionId=1&formB
ean=MobileMainMenu">
```

```

3529
/

1
    </a>
  </td>
  <td>
    <b>

    Resp:

    </b>
    Aug 08,2001 15:00
  </td>
</tr>
...
</body>
</html>

```

对于每一个字节都比较关键的慢速无线连接，可能希望删除其中所有的空白，例如：

```

<html><head><meta content="text/html"http-equiv="Content-Type"><title
class="MultiLingual"> My Calls</title></head><body><div align="right"><a
href="/RTS_HTML_Pages/RTS_General/OnLineHelp/en_US/MobileHelp/MyCallsPag
-e.jsp"name="Help"> ?</a></div><table><tr><td><b> Call:</b><a href="
/servlet/RequestHandler;jsessionId=GVYTQ2IAAAAAC5YAAAARGUQ?task=ActionDe
-tails&action=runQuery&skipLogin=yes&callId=3529&actionId=1&formBean=Mob
-ileMainMenu">3529/1</a></td><td><b> Resp:</b> Aug 08,2001 15:00</td></tr>
<tr><td><b>Sched. From:</b> Aug 03,2001 10:00</td><td><b> Sched. To:</b> Aug
03,2001 10:29</td></tr><tr><td><b> Status:</b> Acknowledged Appointment
</td></tr><tr><td><b> Priority:</b> 90</td><td><b> Site:</b> site 2</td>
</tr><tr><td><b> Product:</b> ISDN2</td></tr><tr><td><hr color="black">
</td></tr><tr><td><b> Call:</b><a href="/servlet/RequestHandler;jsess-
ionid=GVYTQ2IAAAAAC5YAAAARGUQ?task=ActionDetails&action=runQuery&skipLog
in=yes&callId=3695&actionId=1&formBean=MobileMainMenu"> 3695/1</a></td>
<td><b>Resp:</b> Sep 21,2001 13:00</td></tr><tr><td><b> Sched. From:</b>
Sep 27,2001 12:00</td><td><b> Sched. To:</b> Sep 27,2001 13:00</td></tr>
<tr><td><b> Status:</b> Wait for Ack</td></tr><tr><td><b> Priority:</b>
90</td><td><b> Site:</b> site 2</td></tr><tr><td><b> Product:</b> ISDN</td>
</tr><tr><td><hr color="black"></td></tr><tr><td><b>Call:</b><a href="
/servlet/RequestHandler;jsessionId=GVYTQ2IAAAAAC5YAAAARGUQ?task=ActionDe
-tails&action=runQuery&skipLogin=yes&callId=3721&actionId=1&formBean=Mob
-ileMainMenu"> 3721/1</a></td><td><b> Resp:</b> Oct 16,2001 11:00</td></tr>

```



```

<tr><td><b> Sched. From:</b> Oct 16,2001 06:30</td><td><b> Sched. To:</b>
Oct 16,2001 06:59</td></tr><tr><td><b> Status:</b> In work</td></tr>
<tr><td><b> Priority:</b> 90</td><td><b> Site:</b> site 2</td></tr>
<tr><td><b> Product:</b> ISDN</td></tr><tr><td><hr color="black"></td>
</tr><tr><td><b> Call:</b><a href="/servlet/RequestHandler;jsessionId=
GVYTQ2IAAAAC5YAAAARGUQ?task=ActionDetails&action=runQuery&skipLogin=yes
&callId=3809&actionId=1&formBean=MobileMainMenu"> 3809/1</a></td><td><b>
Resp:</b> Nov 12,2001 10:53</td></tr><tr><td><b> Sched. From:</b> Nov 12,
2001 02:53</td><td><b> Sched. To:</b> Nov 12,2001 03:23</td></tr><tr><td><b>
Status:</b> In work</td></tr><tr><td><b> Priority:</b> 90</td><td><b>
Site:</b> Le Mans</td></tr><tr><td><b> Product:</b> U-187-A</td></tr>
<tr><td><hr color="black"></td></tr></table><div align="left"> Calls: 1 to
4 of 4<br></div><table><tr><td><div align="center"><a href="/servlet/
RequestHandler;jsessionId=GVYTQ2IAAAAC5YAAAARGUQ?task=SearchCalls&actio
-n=runQuery&skipLogin=true&login=true&formBean=MobileMainMenu"name="Sear
-ch Calls"><b> Search for Calls</b></a></div></td></tr><tr><td><div align="
center"><a href="/servlet/RequestHandler;jsessionId=
GVYTQ2IAAAAC5YAAAARGUQ?task=OtherCalls&action=runQuery&skipLogin=yes"><
b>Activity - Other Calls</b></a></div></td></tr><tr><td><div align=
"center"><a href="/servlet/RequestHandler;jsessionId=GVYTQ2IAAAAC5
YAAAARGUQ?task=MobileMainMenu&action=runQuery&skipLogin=true"><b> Main
Menu</b></a></div></td></tr></table><hr color="black"></body></html>

```

在第一个例子中，所生成的 HTML 包括 7442 个字符，而第二个页面仅仅包括 3125 个字符。

转码程序

现在开始描述 Transcoder 类，这个类中的许多工作都是由辅助类完成的。下面的例子显示了转码程序的代码。其中一个比较重要的方法是 parse 方法。发生在这个方法中的第一件事情就是基于发出该请求的设备创建一个处理程序（handler）对象。转码程序使用一系列属性文件，并且这些属性文件可以用来标识应该创建什么处理程序（基于 HTTP 请求中的用户代理（user-agent）值）。一个例子就是，对于 PDA 上的常规浏览器，将创建一个 HTML 处理程序；而对于 WAP（Wireless Application Protocol，无线应用协议）电话，将创建 WML 处理程序。为了基于输出来管理转码过程，使用处理程序是必要的。处理程序是最重要的元素，因为它们执行了大部分工作。在转码框架中的处理程序甚至能够执行一些附加的任务，例如提供多语言支持以及安全性支持。

```

public class Transcoder {
    public final static String SUPPORTED_TAGS = "supportedTagList";
    public final static String HANDLER_CLASS = "handlerClass";
    public final static String PAGE_HEADER = "pageHeader";
    public final static String PARSER_CLASS = "transcoder.parser";
}

```

```
public final static java.util.Stack transcoders =
    new java.util.Stack();
public final static String ERROR_MESSAGE_PREFIX = "Html filter: ";

protected PrintWriter        writer;
protected TranscoderHandler   handler;
protected TranscoderParser    parser;
protected Map                 deviceProfile;
protected String              deviceName;
protected InputStream          inputStream;
protected String              pageName;
protected List                 modifiers = new ArrayList();
protected boolean              disableTagRemoving;
protected boolean              isMobileApplication = true;
protected Locale               locale;

protected HttpServletRequest   servletRequest;
protected HttpServletResponse servletResponse;
protected HttpServlet          servlet;

// ....

public void parse() {
try {
    String handlerClass = (String)deviceProfile.get(HANDLER_CLASS);
    if(handlerClass == null) {
        handlerClass = HtmlTranscoderHandler.class.getName();
    }

    try {
        handler = (TranscoderHandler)RTSFactoryManager.
getInstanceOfAbstract(handlerClass);
    } catch(Exception e) {
        RTSTracker.handleException(ERROR_MESSAGE_PREFIX,e);
        copyStreams();
        return;
    }

    handler.setWriter(writer);
    handler.modifiers = modifiers;
```

```
String supportedTagList = (String)deviceProfile.  
get(SUPPORTED_TAGS);  
if(supportedTagList == null) {  
    RTSTracker.displayString(ERROR_MESSAGE_PREFIX +  
        "Failed to get " + SUPPORTED_TAGS +  
        " property for " + deviceName);  
    copyStreams();  
    return;  
}  
  
Map tagList = TagListLoader.getTagList(supportedTagList);  
if(tagList == null) {  
    copyStreams();  
    return;  
}  
  
handler.setTagList(tagList);  
handler.setDeviceProfile(deviceProfile);  
handler.setPageName(pageName);  
handler.setTagRemovingDisabled(disableTagRemoving);  
handler.setIsMobileApplication(isMobileApplication);  
handler.setLocale(locale);  
handler.setTranscoder(this);  
  
parser.setDocumentHandler(handler);  
parser.setErrorHandler(handler);  
parser.setCommentHandler(handler);  
parser.setXmlTags(handler.isXml());  
  
parser.parse(inputStream);  
return;  
}  
catch(SAXNotRecognizedException e) {  
    RTSTracker.handleException(ERROR_MESSAGE_PREFIX +  
        "Unrecognized SAX2 feature",e);  
}  
catch(SAXException e) {  
    RTSTracker.handleException(ERROR_MESSAGE_PREFIX +  
        "SAX exception",e);  
}  
catch(IOException e) {  
    RTSTracker.handleException(ERROR_MESSAGE_PREFIX +  
        "IO Exception",e);
```

```

    }
    catch(Throwable e) {
        RTSTracker.handleException(ERROR_MESSAGE_PREFIX,e);
    }
}

// ...
}

```

在创建一个处理程序之后,将实例化几个其他对象,并且由这个处理程序所拥有。modifier 对象被创建并保存在处理程序中,作为可用标记的列表。所有这些动作都是基于一系列 XML 文件完成的,这些文件表示了转码程序应该如何处理元素的元数据。在这些文件中定义了合法的标记以及为了创建正确的表示形式可能需要的修改。甚至那些用来说明转码程序应该如何工作的元数据都是基于 XML 实现的。

下面两个 XML 文件显示了两个此类元数据的例子,其中第一个用于 HTML,第二个用于 WML。第一个例子如下:

```

<ROOT>
  <tag name="A" attributes="accesskey,charset,coords,href,hreflang,
name,onblur,onfocus,rel,rev,shape,tabindex,target,type"/>
  <tag name="ABBR,ACRONYM,ADDRESS"/>
  <tag name="APPLET" attributes="height,width,align,alt,archive,code,
hspace, name,vspace"/>
  <tag name="AREA" attributes="alt,accesskey,coords,href,$nohref,
onblur,onfocus,shape,tabindex,target"/>
  <tag name="B"/>
  <tag name="BASE" attributes="href,target"/>
  <tag name="BASEFONT" attributes="size,color,face,id"/>
  <tag name="BDO" attributes="dir,id,style,title"/>
  <tag name="BIG"/>
  <tag name="BLOCKQUOTE" attributes="cite"/>
  <tag name="BODY"
attributes="bgcolor,background,text,link,vlink,alink,onload,onunload"/>
  <tag name="BR" attributes="clear" properties="empty"/>
  <tag name="BUTTON" attributes="accesskey,$disabled,name,onblur,onfocus,
tabindex,type,value"/>
  <tag name="CAPTION" attributes="align"/>
  <tag name="CENTER,CITE,CODE"/>
  <tag name="COL" attributes="align,char,charoff,span,valign,width"/>
  <tag name="COLGROUP"
    attributes="align,char,charoff,span,valign,width"/>
  <tag name="DD"/>
  <tag name="DEL" attributes="cite,datetime"/>

```

```

<tag name="DFN"/>
<tag name="DIR" attributes="$compact"/>
<tag name="DIV" attributes="align"/>
<tag name="DL" attributes="$compact"/>
<tag name="DT,EM,FIELDSET"/>
<tag name="FONT" attributes="size,color,face"/>
<tag name="FORM" attributes="action,method, enctype, accept-charset,
onreset,onsubmit,target,name,id"/>
<tag name="FRAME" attributes="frameborder,longdesc,marginheight,
marginwidth,$noresize, scrolling,src"/>
<tag name="FRAMESET" attributes="cols,onload,onunload,rows"/>
<tag name="HEAD" attributes="profile"/>
<tag name="H1,H2,H3,H4,H5,H6" attributes="align"/>
<tag name="HR" attributes="align,size,width,$noshade,color"/>
<tag name="HTML" attributes="version"/>
<tag name="I"/>
<tag name="IFRAME" attributes="align,frameborder,height,longdesc,
marginheight, marginwidth, name,scrolling,src,width"/>
<tag name="IMG" attributes="src,align,hspace,vspace,alt,border,usemap,
ismap, width,height"/>
<tag name="INPUT" attributes="accept,accesskey,align,alt,$checked,
$disabled,maxlength,name,onblur,onchange,onfocus,onselect,$readonly,size,
src,tabindex,type,usemap,value"/>
<tag name="INS" attributes="cite,datetime"/>
<tag name="ISINDEX" attributes="prompt"/>
<tag name="KBD"/>
<tag name="LABEL" attributes="accesskey,for,onblur,onfocus"/>
<tag name="LEGEND" attributes="accesskey,align"/>
<tag name="LI" attributes="type,value"/>
<tag name="LINK" attributes="href,rel,rev,urn,methods,title,id"/>
<tag name="LISTING" properties="preformatted"/>
<tag name="MAP" attributes="name"/>
<tag name="MENU" attributes="$compact"/>
<tag name="META" attributes="name,http-equiv,content"/>
<tag name="NEXTID" attributes="n"/>
<tag name="NOFRAMES,NOSCRIPT"/>
<tag name="OBJECT"
attributes="align,archive,border,classid,codebase,codetype,data,$declare,
height,hspace,name,standby,tabindex,type,usemap,vspace,width"/>
<tag name="OL" attributes="type,start,$compact"/>
<tag name="OPTGROUP" attributes="label,$disabled"/>
<tag name="OPTION" attributes="$selected,value,$disabled,label"/>
<tag name="P" attributes="align"/>
<tag name="PARAM" attributes="name,id,type,value,valuetype"/>

```

```

    <tag name="PLAINTEXT,XMP" properties="preformatted"/>
    <tag name="PRE" attributes="width" properties="preformatted"/>
    <tag name="Q" attributes="cite"/>
    <tag name="S,SAMP"/>
    <tag name="SCRIPT" attributes="type,charset,$defer,language,src,
title"/>
    <tag name="SELECT" attributes="name,size,$multiple,$disabled,onblur,
onchange,onfocus,tabindex"/>
    <tag name="SMALL,STRIKE,STRONG"/>
    <tag name="STYLE" attributes="type,media,title"/>
    <tag name="SUB,SUP"/>
    <!-- tag name="SPAN" attributes="class,id"/> -->
    <tag name="TABLE"
attributes="align,border,width,cellspacing,cellpadding,frame,summary,
bgcolor"/>
    <tag name="TBODY" attributes="align,char,charoff,valign"/>
    <tag name="TD" attributes="abbr,align,axis,bgcolor,char,charoff,
colspan,headers,height,$nowrap,rowspan,scope,valign,width"/>
    <tag name="TEXTAREA" attributes="cols,rows,accesskey,
$disables,name,onblur,onchange,onfocus,onselect,$readonly,tabindex"
properties="alwaysClosed"/>
    <tag name="TFOOT" attributes="align,char,charoff,valign"/>
    <tag name="TH" attributes="abbr,align,axis,bgcolor,char,charoff,
colspan,headers,height,$nowrap,rowspan,scope,valign,width"/>
    <tag name="THEAD" attributes="align,char,charoff,valign"/>
    <tag name="TITLE"/>
    <tag name="TR" attributes="align,bgcolor,char,charoff,valign"/>
    <tag name="TT,U"/>
    <tag name="UL" attributes="type,$compact"/>
    <tag name="VAR"/>
</ROOT>

```

第二个例子如下：

```

<ROOT>
    <tag name="WML" attributes="xml:lang"/>
    <tag name="HEAD"/>
    <tag name="ACCESS" attributes="domain,path"/>
    <tag name="META" attributes="http-equiv,name,forua,content,scheme"/>
    <tag name="TEMPLATE"
        attributes="onenterforward,onenterbackward,ontimer"/>
    <tag name="CARD" attributes="onenterforward,onenterbackward,ontimer,
title,newcontext,ordered,xml:lang,id"/>
    <tag name="SELECT" attributes="multiple,name,value,iname,ivalue,

```

```

title,tabindex,xml:lang"/>
  <tag name="OPTION" attributes="value,title,onpick,xml:lang"/>
  <tag name="OPTGROUP" attributes="title,xml:lang"/>
  <tag name="INPUT" attributes="name,value,type,format,emptyok,size,
maxlength,title,xml:lang,tabindex,accesskey" properties="empty"/>
  <tag name="FIELDSET" attributes="title,xml:lang"/>
  <tag name="TIMER" attributes="name,value"/>
  <tag name="EM,STRONG,I,B,U,BIG,SMALL" attributes="xml:lang"/>
  <tag name="P" attributes="align,mode,xml:lang"/>
  <tag name="BR" properties="empty"/>
  <tag name="TABLE" attributes="title,align,columns,xml:lang"/>
  <tag name="TR"/>
  <tag name="TD" attributes="xml:lang"/>
  <tag name="PRE" attributes="xml:space" properties="preformatted"/>
  <tag name="IMG" attributes="alt,src,localsrc,vspace,hspace,
align,height,width,xml:lang"/>
  <tag name="POSTFIELD,SETVAR"
    attributes="name,value" properties="empty"/>
  <tag name="GO"
    attributes="href,sendreferer,method,enctype,accept-charset"/>
  <tag name="PREV,REFRESH,NOOP"/>
  <tag name="DO" attributes="type,label,name,optional,xml:lang"/>
  <tag name="ANCHOR" attributes="title,accesskey,xml:lang"/>
  <tag name="A" attributes="href,title,accesskey,xml:lang"/>
  <tag name="ONEVENT" attributes="type"/>
</ROOT>

```

在处理程序准备好之后，它将被设置到解析程序的一个实例变量中。然后，`parse` 方法被调用，从而使得解析程序开始工作。解析程序的一个例子如下：

```

public class SimpleParser extends TranscoderParserImpl {
    protected SimpleLexer lexer = new SimpleLexer();
    public void parse (InputStream stream) throws Exception {
        lexer.reset(stream);

        documentHandler.startDocument();
        while (lexer.nextToken() != lexer.TOKEN_EOF) {
            int type = lexer.getTokenType();
            if(lexer.mWhiteSpace.count != 0) {
                documentHandler.characters(
                    lexer.mWhiteSpace.data,0,lexer.mWhiteSpace.count);
            }

            switch(type) {

```

```

        case lexer.TOKEN_TAG:
            TagInfo tag = lexer.mCurrentTag;
            if(tag.isEndTag()) {
                documentHandler.endElement(tag.mName);
                break;
            }
            attributeList.clear();
            for(int i = 0; i < tag.parameterCount(); i++) {
                attributeList.addAttribute(
                    tag.getParameterName(i),
                    null, tag.getParameterValue(i));
            }
            documentHandler.startElement(tag.mName, attributeList);
            if(tag.isStartEndTag()) {
                documentHandler.endElement(tag.mName);
            }
            break;

        case lexer.TOKEN_SCRIPT:
        case lexer.TOKEN_TEXT:
            if(lexer.mBuffer.count != 0) {
                documentHandler.characters(lexer.mBuffer.data, 0,
                                           lexer.mBuffer.count);
            }
            break;

        case lexer.TOKEN_COMMENT:
            if(lexer.mBuffer.count != 0) {
                commentHandler.comment(lexer.mBuffer.toString());
            }
    }

    }
    documentHandler.endDocument();
}
public void setXmlTags(boolean flag) {
}
}
}

```

SimpleParser 类从 TranscoderParserImpl 类继承，如下所示：

```

public abstract class TranscoderParserImpl implements TranscoderParser {
    protected DocumentHandler documentHandler;
    protected ErrorHandler errorHandler;
}

```



```
        protected AttributeListImpl attributeList;
        protected CommentHandler commentHandler;
    public TranscoderParserImpl() {
        attributeList = new AttributeListImpl();
    }
    public void parse (String systemId) throws SAXException, IOException {
        try {
            parse(new FileInputStream(systemId));
        } catch (Exception e) {
            if(e instanceof IOException) {
                throw (IOException)e;
            }
        }
    }

    public void parse (InputSource src) throws SAXException, IOException {
        try {
            parse(src.getByteStream());
        } catch (Exception e) {
            if(e instanceof IOException) {
                throw (IOException)e;
            }
        }
    }

    public void setCommentHandler(CommentHandler commentHandler) {
        this.commentHandler = commentHandler;
    }
    public void setDocumentHandler (DocumentHandler handler) {
        documentHandler = handler;
    }
    public void setDTDHandler (DTDHandler handler) {
    }
    public void setEntityResolver (EntityResolver resolver) {
    }
    public void setErrorHandler (ErrorHandler handler) {
        errorHandler = handler;
    }
    public void setLocale(java.util.Locale locale) throws SAXException {
    }
}
```

这个类就是转码解析程序层次的根，但并不一定被实例化，这是因为正如其代码显示的那样，它并没做什么实际的工作。基类实现了 `TranscoderParser`，它是对 `org.xml.sax.Parser` 接

口的扩展：

```
public interface TranscoderParser extends org.xml.sax.Parser {
    public void parse (InputStream is) throws Exception;
    public void setCommentHandler(CommentHandler commentHandler);
    public void setXmlTags(boolean isXml);
}
```

现在此应用程序的工作过程就变得更加清晰了；转码程序基于对事件进行解析的概念——即 SAX。它将由处理程序对象（通过解析程序）注册事件并处理代码。如果查看 SimpleParser 中 parse 方法的实现，就可以看到它使用一个“lexer”对象（词法解析对象）来识别权标（token）。基于权标被识别成为标记、脚本还是文本，它分别执行不同的任务——但总是通过处理程序对象（这是所有的智能所在）进行工作。

35.3 小 结

本章提供了两个例子，用来说明如何在应用程序中使用 XML。实际上，这些例子仅仅是冰山一角；已经证明，在应用程序中使用 XML 非常有价值。其主要原因是 XML 具有动态性与层次性。计算机科学中的大多数数据结构都是层次性的。随着时间的推移，用来处理与使用 XML 的解析程序和工具正在变得更快、更标准并且质量更高。这也使得 XML 的应用更加广泛，几乎包括了人们能够想到的任何领域。实际上，这也很自然——因为一旦使用了这些库，就很有可能对那些具有层次性、递归性、动态性或任何其他数据结构处理类型的事物都使用 XML。

第 36 章 XSL Transformation 的应用

XML 在电子商务应用程序中所起的核心作用由于 XSLT 而得到进一步加强，XSLT 是 XSL (Extensible Stylesheet Language, 可扩展样式表语言) 的一个子集，它定义了如何将一种 XML 方言转换成另一种。XSLT 对于 XML 在电子商务应用程序中的成功非常重要。由于不同的系统以及不同的网站几乎都使用不同的 XML 结构，因此，对各种结构中不匹配的语义提供桥接从而实现信息交换的能力，可能就成为决定 XML 成败的主要因素。为了实现这种桥接，就应该以 XML 转换的形式定义变换规则。

本章描述 XSLT 处理的两个方面。首先，说明了 XSL 样式表如何用于处理一个 XML 文档。然后，说明了如何在普通的 XSLT 处理程序中使用 Java 代码作为接口层的一部分，接口层能够接受任何 XML 格式的数据，并将所接收的数据转换成应用程序能够理解的本地 XML 格式。

36.1 更多的内线交易

本节继续讨论第 35 章描述的内线交易例子，用来说明如何使用 XSLT 来处理 XML 文档，并作为电子商务应用程序的一部分。这个例子使用包含在 WebSphere 库中的 LotusXSL。

36.1.1 去除错误

在第 35 章，我们创建了一个关于内线购买者列表的 XML 文件。但不幸的是，这个文件很大（大约有 2MB），而且很难使用。实际上，这个文件对一个典型的用户也不是非常有用。本节与下一节将提供一些 XSLT 示例，用来说明如何处理这些数据，直到得到满意的格式与内容。

需要做的第一件事情就是去除无用的数据，例如，如果任何股票的 insiderPurchase 节点具有 error 属性（见图 36-1），则删除此股票的 XML 节点。当在内线交易程序中对节点遍历的解析失败时，就产生 error 属性。由于这类节点没有包含任何信息，因此可以删除。

下列代码显示了删除这些节点所必需的一条处理指令、一个文档根以及 XSLT 规则：

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

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

  <xsl:template match="/">
```


`<xsl:stylesheet>` 类型的元素，并且所有其他元素都嵌入在这个文档根中。这个文档根中包含一个名称空间属性，用来定义所有属于 XSL 名称空间的元素所使用的 URI。

由于 XSLT 的语法和语义并不是很简单，因此下面将详细描述每条规则并提供尽可能多的解释。

本例中的文档有 3 条规则。每条规则都定义了一个模式，XSLT 处理程序试图匹配这个模式，并且在模式匹配之后将节点标记为应该被处理的节点。第一条规则将匹配源 XML 文件的根节点（用反斜杠（/）表示）。第二条规则匹配 `insiderTrading` 节点的任何直接后代节点。模式匹配规则的语法类似于 UNIX 中的目录结构。第三条规则匹配任何不具有 `error` 属性的 `insiderPurchase` 节点。在 XSL 中，属性以一个前导的“at”（@）符号表示。因此，第三条规则的模式匹配定义应该为“所有不存在 `error` 属性名的 `insiderPurchase` 节点”。

这些规则的处理部分比较简单。对于根节点，将创建一个带有单个属性（称为 `example`）的名为 `insiderTrading` 的 XML 元素。这个元素将作为结果 XML 文件的根元素。然后，对源文件中文档的其余部分应用所有的规则。对股票（`stock`）节点的处理是最容易的。只要将这个节点复制过来并继续处理其子孙节点即可。最后，当遇到一个不是错误节点的购买（`purchase`）节点时，将在输出中创建一个适当的节点。需要说明的是，语法中包含有已经被计算的值，这个值嵌入在带有花括号的字符串中。

36.1.2 删除那些没有内线购买者的代号

现在可以着手编写第一个 XSLT 文件了，这个文件大小从原来的 1942KB 降低到现在的 1316KB（当然，还可以做得更好）。实际上，还存在许多根本没有进行内线交易的节点（也就是那些满足规则的公司），因此也可以将这些节点删除。为此，对上面的第二条规则所定义的模式匹配进行一点小修改即可：只有当 `insiderTrading` 节点的子节点满足第二条规则所定义的模式匹配，并且至少具有一个 `insiderPurchase` 节点时，这个规则才能满足：

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

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

  <xsl:template match="/">
    <insiderTrading example="2">
      <xsl:apply-templates/>
    </insiderTrading>
  </xsl:template>

  <xsl:template match="insiderTrading/*[insiderPurchase]">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

```
</xsl:template>

<xsl:template match="insiderPurchase[not(@error)]">
  <insiderPurchase date="{@date}" price="{@price}"/>
</xsl:template>

</xsl:stylesheet>
```

36.1.3 删除那些没有价格的节点

这样确实删除了一些节点，但并没有很大的帮助，因为文件大小仅仅改变到 1306KB。现在，将删除那些没有价格的节点；如果不知道内线购买者在购买股票时的价格，实际上也不能判断当前价格是高还是低。下列 XSLT 文件将使文件大小进一步降低到 1148KB：

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

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

  <xsl:template match="/">
    <insiderTrading example="3">
      <xsl:apply-templates/>
    </insiderTrading>
  </xsl:template>

  <xsl:template match="insiderTrading/*[insiderPurchase]">
    <xsl:copy>
      <xsl:apply-templates/>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="insiderPurchase[not(@price=' - ')]">
    <insiderPurchase date="{@date}" price="{@price}"/>
  </xsl:template>

</xsl:stylesheet>
```

36.1.4 排序

现在文件大小已经降到可接受的程度了。下一步开始进行排序。在下面的例子中，将按照价格的值进行排序。对于每个股票代号，都要求按照价格值所决定的顺序进行排序并列出这些内线交易；毕竟，这里所关心的是尽可能低的价格，因此，可能还需要将当前价格与内线所购买的最低价格进行对比。下面显示了执行排序的 XSLT 文件。需要说明的是，sort

元素具有一个可选的 `data-type` 属性，用来指定如何执行排序。在这个例子中，不希望使用字母数字顺序进行排序；否则，价格 10 就可能出现在价格 3 之前。

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

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

  <xsl:template match="/">
    <insiderTrading example="4">
      <xsl:apply-templates/>
    </insiderTrading>
  </xsl:template>

  <xsl:template match="insiderTrading/*[insiderPurchase]">
    <xsl:copy>
      <xsl:apply-templates select="insiderPurchase">
        <xsl:sort select="@price" data-type="number"/>
      </xsl:apply-templates>
    </xsl:copy>
  </xsl:template>

  <xsl:template match="insiderPurchase">
    <insiderPurchase date="{@date}" price="{@price}"/>
  </xsl:template>

</xsl:stylesheet>
```

36.1.5 将 XML 文档转换为 HTML

下一步对于电子商务应用程序很有实际意义。这一步将把 XML 文件转换为 HTML 文档，从而能够利用一个表来显示所有数据。这一过程听起来似乎很简单，但是，这里存在一个比较大的问题：不能获取 XSL 中实际标记的值，因为在 XML 中，股票代码是标记而不是数据。这是一个重大的教训，因此我们希望在学完本章之后，读者应该记得不要将数据当作标记处理。

为了完成这个程序，需要修改第 35 章所创建的 `servlet` 的代码，如下所示：

```
public static void main(String[] args) {

  InsiderInspector gen = new InsiderInspector();

  com.ibm.xml.parser.TXDocument myDocument =
    new com.ibm.xml.parser.TXDocument();
```

```
try {
    ProcessingInstruction pi = myDocument.
        createProcessingInstruction("xml", "version=\"1.0\"");
    myDocument.appendChild(pi);

    Element root = myDocument.createElement("insiderTrading");
    root.setAttribute("asOf", (new Date()).toString());
    myDocument.appendChild(root);

    Vector symbolVector = gen.retrieveAllSymbols();
    for (Enumeration e = symbolVector.elements() ;
        e.hasMoreElements() ; ) {
        String aSymbol = (String)e.nextElement();
        StringBuffer htmlText = HTMLFetcher.getHTMLFor(
            "http://biz.yahoo.com/t/" + aSymbol.substring(0,1) +
            "/" + aSymbol + ".html");
        StringBuffer aBuf = HTMLTidier.tidyHTML(htmlText);
        if (aBuf != null) {
            Element symbolRoot = myDocument.createElement("stock");
            symbolRoot.setAttribute("symbol", aSymbol);
            root.appendChild(symbolRoot);
            gen.extractInsiderInfo(aBuf, myDocument, symbolRoot);
            System.err.println("Finished processing " + aSymbol);
        }
    }

    FileWriter fw = new FileWriter("insiderTrading.xml");
    myDocument.toXMLString(fw);
    fw.close();
} catch (Exception ex) {
    ex.printStackTrace();
}
}
```

现在，当重新运行 XML 生成代码时，应该将 XML 源修改为如图 36-2 所示。这样就允许使用下列 XSLT 代码来产生最终的结果，如图 36-3 所示。


```

- <insiderTrading example="4">
- <stock symbol="ATEN">
  <insiderPurchase date="22-Oct-98" price="3.94" />
  <insiderPurchase date="22-Oct-98" price="4.00" />
  <insiderPurchase date="21-Oct-98" price="4.25" />
  <insiderPurchase date="20-Oct-98" price="4.31 - 4.38" />
  <insiderPurchase date="19-Oct-98" price="4.63" />
  <insiderPurchase date="14-Oct-98" price="4.75" />
  <insiderPurchase date="4-Nov-98" price="7.25 - 9.00" />
  <insiderPurchase date="2-Nov-98" price="7.25" />
  <insiderPurchase date="2-Nov-98" price="7.25" />
  <insiderPurchase date="31-Aug-98" price="8.63 - 8.75" />
  <insiderPurchase date="31-Aug-98" price="8.80" />
  <insiderPurchase date="4-Sep-98" price="9.19" />
  <insiderPurchase date="14-Sep-98" price="9.88" />
  <insiderPurchase date="15-Sep-98" price="10.38" />
</stock>
- <stock symbol="CTAC">
  <insiderPurchase date="26-Aug-98" price="5.75" />
  <insiderPurchase date="30-Sep-98" price="5.94" />
  <insiderPurchase date="29-Sep-98" price="6.00" />
  <insiderPurchase date="14-Aug-98" price="6.00" />
  <insiderPurchase date="18-Aug-98" price="6.06" />
  <insiderPurchase date="13-Aug-98" price="6.06" />
  <insiderPurchase date="9-Sep-98" price="6.50" />
  <insiderPurchase date="9-Sep-98" price="6.50" />
  <insiderPurchase date="17-Mar-99" price="12.50 - 12.63" />
</stock>
- <stock symbol="SRCE">
  <insiderPurchase date="28-Aug-98" price="31.56" />
  <insiderPurchase date="25-Aug-98" price="34.00" />
  <insiderPurchase date="28-Jan-99" price="35.00" />
  <insiderPurchase date="1-Feb-99" price="36.00" />
</stock>
- <stock symbol="FSBC">
  <insiderPurchase date="27-Apr-99" price="19.00" />
  <insiderPurchase date="29-Apr-99" price="19.25" />
  <insiderPurchase date="26-Apr-99" price="19.38" />

```

图 36-2 改进后的新 XML

ABGX	5-Mar-99; 14.50	14-Jun-99; 15.25	24-Jun-99; 15.31	15-Jun-99; 15.63															
AANB	30-Apr-99; 10.75	23-Apr-99; 11.00	30-Mar-99; 11.00	11-Feb-99; 11.75	22-Jun-98; 12.60	31-Aug-98; 13.75	14-Sep-98; 13.88	2-Dec-98; 14.80	1-Dec-98; 14.80	26-Aug-98; 15.13	24-Aug-98; 15.27	23-Oct-98; 17.63							
ABBK	12-Apr-99; 14.86																		
ABMD	5-Apr-99; 12.25	1-Apr-99; 12.25	29-Mar-99; 12.25	31-Mar-99; 12.50															
ABTE	3-Sep-98; 2.75 - 3.03	4-Sep-98; 2.88 - 3.38	2-Sep-98; 3.00	9-Sep-98; 3.25	1-Sep-98; 3.25 - 4.53	31-Aug-98; 5.38 - 5.56	11-Sep-98; 7.00												
BOUT	24-Mar-99; 25.00	24-Mar-99; 25.00	24-Mar-99; 25.00	24-Mar-99; 25.00	24-Mar-99; 25.00	24-Mar-99; 25.00													
ABOV	15-Dec-98; 13.00	15-Dec-98; 13.00	15-Dec-98; 13.00	15-Dec-98; 13.00	11-Dec-98; 13.56														
ABRX	24-Dec-98; 16.77																		
ACRI	20-Aug-98; 4.44	10-Aug-98; 4.63	10-Aug-98; 4.75	12-Aug-98; 5.00															

图 36-3 内线购买模式的 HTML 显示

```

<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/XSL/Transform/1.0">

  <xsl:template match="/">
    <HTML>
      <HEAD>
        <TITLE>
          Insider Buyer patterns
        </TITLE>
      </HEAD>
      <BODY>
        <H3>
          This table lists all insider purchases for Nasdaq stock.
        </H3>
        <TABLE BORDER="1" WIDTH="100">
          <xsl:apply-templates/>
        </TABLE>
      </BODY>
    </HTML>
  </xsl:template>

```

```

<xsl:template match="stock">
  <TR>
    <TD>
      <xsl:value-of select="@symbol"/>
    </TD>
    <xsl:apply-templates/>
  </TR>
</xsl:template>

<xsl:template match="insiderPurchase">
  <TD>
    <xsl:value-of select="@date"/>
    <xsl:text>
      ;
    </xsl:text>
    <xsl:value-of select="@price"/>
  </TD>
</xsl:template>

</xsl:stylesheet>

```

最后，下列 XSLT 文件可用来将 XML 转换到 WML，如图 36-4 所示。

```

<?xml version="1.0" ?>
<wml>
  <card id="n1" title="ATEN">
    <do type="options" label="First">
      <go href="#n1" />
    </do>
    <do type="options" label="Second">
      <go href="#n2" />
    </do>
    <p>
      22-Oct-98 ; 3.94
      <br />
      22-Oct-98 ; 4.00
      <br />
      21-Oct-98 ; 4.25
      <br />
      20-Oct-98 ; 4.31 - 4.38
      <br />
      19-Oct-98 ; 4.63
      <br />
      14-Oct-98 ; 4.75
      <br />
      4-Nov-98 ; 7.25 - 9.00
      <br />
      2-Nov-98 ; 7.25
      <br />
      2-Nov-98 ; 7.25
      <br />
      31-Aug-98 ; 8.63 - 8.75
      <br />
      31-Aug-98 ; 8.80
      <br />
      4-Sep-98 ; 9.19
      <br />
      14-Sep-98 ; 9.88
      <br />
      15-Sep-98 ; 10.38
      <br />
    </p>
  </card>
</wml>

```

图 36-4 结果 WML

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

  <xsl:template match="/">
    <xsl:pi name="xml">
      version="1.0"
    </xsl:pi>
    <wml>
      <xsl:apply-templates/>
    </wml>
  </xsl:template>

  <xsl:template match="stock">
    <xsl:element name="card">
      <xsl:attribute name="id">
        <xsl:textn/xsl:text>
          <xsl:number/>
        </xsl:attribute>
      <xsl:attribute name="title">
        <xsl:value-of select="@symbol"/>
      </xsl:attribute>
      <do type="options" label="First">
        <go href="#n1"/>
      </do>
      <do type="options" label="Second">
        <go href="#n2"/>
      </do>
      <p>
        <xsl:apply-templates/>
      </p>
    </xsl:element>
  </xsl:template>

  <xsl:template match="insiderPurchase">
    <xsl:value-of select="@date"/>
    <xsl:text>
      ;
    </xsl:text>
    <xsl:value-of select="@price"/>
    <br/>
  </xsl:template>

</xsl:stylesheet>
```

WML 的底层模型引入了卡片的概念。在某一时刻，蜂窝设备显示器的显示就是一个 WML 卡片，并且允许用户使用电话的控制键在卡片之间进行移动。图 36-5 显示了在蜂窝电话上得到的结果显示：两个卡片包围在屏幕的周围，用来在卡片之间进行导航。需要说明的是，在这里采用了一点欺骗技术，因为这个 XML 文件仅仅支持 3 种股票代码。显然，一般并不希望通过蜂窝连接来传递 1MB 的数据，实际上，人们一般也不希望在这个设备上存在成千上万个卡片。



图 36-5 内线购买模式的 WML 显示

36.2 XML 到 XML 转换

这里讨论的第二个例子是一个实际的应用示例。虽然目前大多数应用程序都能够读取 XML 文件，并且在它们自己的接口层使用这个文件，但不幸的是，大多数应用程序都使用它们自己的 XML 方言。这使得不同应用程序之间的对接就如同出现 XML 之前一样困难，这也是出现 XSLT 的一个原因。由于 XSLT 能够在 XML 结构与另一个 XML 结构之间进行转换，因此使用 XSLT 作为中间层是很有价值的。

特别地，可以插入一个转换层，用来解析 XML 文档，并且在 XML 根节点（每个 XML 文档只有一个根节点）之间进行映射，然后选择 XSLT 表单并应用到另一个 XML 文档上。这就意味着，在处理应用程序自己的 XML 文档之前，首先将它转换为应用程序能够理解的另一个 XML 文档。转换过程需要使用 XSLProcessor 对象，这个对象是 pcom.lotus.xsl 包的一部分。实际工作在下列方法中完成，可以看到，其工作很简单。

```
public Document customize(String _xslPath)
    throws SAXException
{
    XSLProcessor processor = new XSLProcessor();
    Throwable ex = null;
    // Tries to customize the Document by using XML instruction
    // with references to an XSL file
    try
    {
        outDocument = processor.process(inDocument);
    }
    catch(MalformedURLException e){ex = e;}
    catch(FileNotFoundException e){ex = e;}
    catch(IOException e){ex = e;}
    catch(SAXException e){ex = e;}

    if(ex != null)
        System.err.println("XslCustomizer" + ex);

    // if customization using XML instruction failed and _xslPath
    // has value (i.e. there is a mapping between the root node
    // and an XSL file then customize the document
    // using XSL file from '_xslPath' location
    if(_xslPath.length() > 0 && outDocument == null)
        outDocument = processor.process(inDocument,
            XmlDomGenerator.readDocument(_xslPath), null);

    //if customization failed outDocument equals inDocument
    if(outDocument == null)
        outDocument = inDocument;
    return outDocument;
}
```

36.3 小 结

XSLT 是一个强大的工具，因此，它也是读者必须熟悉的重要工具之一。但不幸的是，XSLT 难于使用。当然，从 Java 代码的观点来看，进行这些处理又不是很困难，实际上还比较简单。其困难在于编写 XSLT 表单，这是一个基于模式匹配范型的处理过程，而不是基于过程范型。XSLT 所固有的困难使得它并不像原来想象的那样得到了广泛使用。

人们曾经广泛地这样认为：构建应用程序的方法就是创建一个业务逻辑层，由此来产生 XML 格式的数据，并且利用 XSLT 表单来创建表示层，以适合不同类型浏览器的使用。但构建 XSLT

表单（即使是使用工具构建）所存在的困难使得这一技术并没有当初希望的那样实用。

XSLT 还没有解决的另一个问题就是 XML 方言之间的桥接。在前一节所显示的代码是很简单的，但实际上，用来完成这种桥接所需的 XSLT 表单非常复杂并且难于编写。因此，业界转向创建支持应用程序可互操作性的标准，这将在第 37 章与第 38 章讨论。

第 37 章 Web 服务：SOAP，WSDL 与 UDDI

Web 服务是当前计算领域最热门的话题之一。厂商与用户等等都对这些新技术表现出极大的兴趣，这些技术为分布式独立的系统提供简单、基于标准的可互操作性。实际上，它已经成为软件技术领域最受关注的话题，并且人们在 Web 服务器技术方面进行的投资也是非常可观的。Microsoft 已经定义了它在 Web 服务方面的整个 .NET 策略（这也是其作为技术厂商的整个策略）。

本章介绍 Web 服务的概念，并且描述构成 Web 服务的 3 个主要技术：SOAP (Simple Object Access Protocol, 简单对象访问协议)，用来进行远程调用以及消息传递；WSDL (Web Services Description Language, Web 服务描述语言)，用来描述服务端点；UDDI (Universal Description, Discover and Integration, 通用描述、发现和集成)，用来使用一个通用的注册表发布与发现服务。本章将描述这些技术，然后分析 WebSphere Application Server 中是如何支持这些技术的。

37.1 Web 服务模型

电子商务的应用正在从根本上改变系统的外观，并且更重要的是（对开发人员来说），它正在改变人们如何构建系统的方式。为了支持电子商务，系统常常是“从外部向内部”发展的，表现在，Web 站点能够很快建立，然后，慢慢地延伸到公司的后端系统中。电子商务的另一个焦点问题是合作系统之间的互联，这种互联有时是企业内部的连接，有时则需要连接到外部。在如今的电子商务策略中，e-bonding 的概念已经不再是遥遥无期的。在这里，e-bonding 就是指一个系统能够灵活、快速地连接到顾客以及业务伙伴的系统上的能力。例如，ViryNet 的 Service Hub 平台不仅支持内部雇员，而且支持供货商、第三方以及顾客——这个服务社区中的所有成员。这就意味着，明天的电子商务应用程序需要连续不断地相互交互，并且不需要为了点对点集成进行巨大的投资。显然，这样就需要某些标准的支持。使用一个新模型以协作的方式运行电子商务应用程序并不是一个新概念。实际上，学术领域已经对这个问题进行了多年的理论探索。但如今技术发展已经使得这种方式成为可能，并且成为一种主流的方式。这对于构建这种新模型的人来说都是值得惊喜的。实际上，我们也是为了将这种业务功能模型介绍给用户而工作。这个新模型称为 Web 服务 (Web service)。实际上，对 Web 服务的支持可能是 WebSphere 4.0 中增加的最重要功能。

Web 服务模型解决了使用标准 Internet 协议可以访问的功能元素（在这里很难使用像“组件”或“模块”这样的词，否则很容易覆盖其原有的含义）的相关问题。Web 服务模型是基于

组件的开发过程中所得到的经验与基于 Web 的访问和调用被广泛使用的结合。这个模型的基本思想并不是新的；这些思想类似于早在 10 年前就已经为人所知的各种技术。

但是，Web 服务模型的独特之处表现在两个方面：首先，它基于 Internet 协议，因此非常适合于建立 Internet 连接。在理论上，这就使得能够获取无限的可用信息、功能与计算能力。其次，Web 服务模型得到 Microsoft 阵营与非 Microsoft 阵营双方的支持（出现这种现象还是第一次）。

这个新模型的潜力是巨大的。对于用户，它意味着廉价、快速部署并且功能齐全的系统，能够连续不断地改进系统、灵活性与可扩展性。对于解决方案与应用程序提供者，它意味着更多的获利机会以及更低的发展成本。并且最重要的是，对于底层结构厂商，这个新模型意味着新的许可证获利机会。因此，这个模型对各方面都有好处。

Web 服务模型的元素

支持 Web 服务模型所需的基本元素实际上很简单。Web 服务是一些部署在节点上的功能元素，这个节点可以连接在任何网络上，并且能够通过 Internet 进行访问。对于 Web 服务，并没有关于编程语言（利用编程语言来编写功能元素）的限制，也没有对操作平台（在操作平台上的功能元素）的限制。因此，为了支持 Web 服务模型，需要 3 个元素：

- 一种标准的方法，用来代表数据与消息，或者代表激活这些功能元素的调用。
- 一种标准的方法，用来描述 Web 服务完成的工作如何被服务的用户（典型地，是另一种使用 Web 服务的功能元素）所理解。
- 一种标准的方法，用来发现 Web 服务的提供者。

幸运的是，所有这些元素都已经实现。这 3 种能够共同支持 Web 服务的技术分别是 SOAP、WSDL 与 UDDI。并且更为有趣的是，所有这 3 种技术都是 XML 应用程序，这也进一步证明了 XML 是“Web 的事实语言标准”。由于 WebSphere 是 IBM 支持电子商务的平台，因此毫无疑问，IBM 是提供 Web 服务完全开发与部署环境（在 WebSphere 4.0 中）的第一批厂商。

Web 服务的出现，不仅改变了如何做某件事，而且改变了应该做什么。应用于 HTTP 上的 XML 仅仅是一个用来指定系统之间如何通信的模式。但 Web 服务则覆盖了更广泛的内容。系统能够通信之后，下一步需要解决的问题就是通信的内容以及决定与谁进行通信。Web 已经从一个静态模型发展成一个动态、可交互的模型。在这个模型中，Web 需要有一种方法能够将所有服务集成在一起。这不仅意味着需要具有查找这些服务的能力，而且需要确定某个网站能够提供哪些服务，以及如何注册与发现这些服务。所有这些问题得到解决之后，所产生的结果就是一个完全互联、具有通用的服务传递与使用功能的大系统（直到现在，人们也只能想象这种情况）。的确，所有这些标准目前还处于其发展的早期阶段，这种通用的服务还没有成为现实；但是，面向这一目标而开发的工具越来越成熟，并且发展迅

速。所有构建这种系统的人员都应该对这些工具与标准有深刻的理解。

37.2 创建新标准

在应用程序可互操作性领域的发展前沿充满了激烈的竞争，并且有大量的失败。仅仅在 5 年前，OMG 的 CORBA 与 Microsoft 的 DCOM（Distributed Component Object Model，分布式组件对象模型）之间进行的那场竞争就非常激烈。虽然这些技术对于 LAN 上的应用程序可互操作性支持很出色，但它们仍然不适用于 Internet 上的可互操作性。更为重要的是，所有这些技术都更加适合于紧密耦合的环境。但在 Web 环境下，只有通过松耦合的模型才能够实现其所具有的优点。因此，IIOP 可能适合于 EJB 通信，但不适合于 e-bonding。在这个环境中，HTTP 必须是底层协议，而 XML 必须是所选择的语言。

这一事实已经被包括 IBM 与 Microsoft（这两个公司确实在就此进行协作，并且它们是推动 Web 标准形成的主要动力）在内的厂商所确认。IBM 与 Microsoft 就一个技术基础进行协作这种情况是不常见的，但它们的这种协作得到了广泛的欢迎，并且其结果是导致业界的广泛接受。例如，许多其他厂商（包括 Ariba，BEA，HP，Iona，SAP 与 Software AG）都已经开始购买 IBM/Microsoft 关于 Web 服务的这种领先服务。但是，对这些标准造成威胁并且试图打破这种局面的是 Sun 宣布对 SOAP 与 UDDI 的支持，作为 Sun ONE（Open Network Environment，开发网络环境）的一部分。甚至 ebXML（Electronic Business XML，电子商务 XML）阵营也开始支持 SOAP，并且在未来将进一步支持 WSDL 与 UDDI，这样就使得 SOAP/WSDL/UDDI 成为事实的协议并且成为 Web 服务的标准基础。

分析家们也就这些问题达成了一致。Gartner Group 在 2001 年 4 月发布的研究报告预测，IBM 与 Microsoft 有 80% 的可能性成为定义 Web 标准的领导者。任何信服这些分析家的人都知道，80% 就意味着实际上已经成为事实。这种分析报告常常能够产生市场，而不是仅仅告知业务部门关于市场的状态。

37.2.1 从 XML 到 SOAP

HTTP 与 XML 是基础，但它们还远远不够。HTTP 主要用来从一个地方向另一个地方传递文件（也可能是请求）；它对于完全的应用程序可互操作性是不够的。因此，第一步就是要对 HTTP 进行一些扩展：添加一个 HTTP 头部集合；将请求封装到一个新的 HTTP POST 命令（称为 M-POST）类型中；在 HTTP 请求体中添加一个丰富的 XML 结构，从而能够支持通过 Internet 进行的复杂应用程序到应用程序之间的通信。这也就是 SOAP。

SOAP 使用 HTTP 传输 XML 格式的消息。它提供了一个构建于 Internet 标准之上的标准对象调用协议，使用 HTTP 作为传输层并且使用 XML 作为编码层。它在本质上是可扩展的，因为在消息体中使用 XML，并且因为它添加头部使得消息能够不断随着时间而发展。SOAP

在 HTTP 客户与服务器的层次上工作。它并不关注操作系统、编程语言或其他类似的问题；因此，它确实能够作为可互操作性的基础。由于 SOAP 在 HTTP 与 XML（两种很简单的技术）的基础上构建，因此 SOAP 也很简单，至少比前面提到的可互操作性技术（例如 CORBA 与 DCOM）简单得多。

37.2.2 添加 WSDL

一旦有了分离系统之间的通信基础，就可以开始讨论 Web 服务了。显然，SOAP 提供了进行调用的能力，但 SOAP 对于创建 Web 服务的构成或本质还是不够的。因此，IBM、Microsoft 与 Ariba 进行的下一步工作就是创建 WSDL。WSDL 定义了一个 XML 语法，用来将网络服务描述成为能够交换消息的通信端点集合。这个语法描述了面向文档的服务以及面向过程的服务。WSDL 为消息的抽象提供了定义，并且为可能的通信所需的特殊协议提供了定义。所有这些工作都是在不限消息类型、协议、服务的内容的条件下进行的。利用 WSDL，就能够定义用于通信的系统的端点，并且能够将这些端点收集到集合中，这些集合就是向外部世界提供的 Web 服务。

37.2.3 添加 UDDI

最后一步工作就是 IBM、Microsoft 与 Ariba 开发的 UDDI。成功的电子商务要求公司之间能够互相发现对方，使本公司的需求与能力被其他公司所了解，并且能够使用不同的进程集成服务。UDDI 支持的目标模型就是，在这个模型下，公司能够连接到在 Internet 上部署的注册表上，从而定位到其他公司，了解这些公司所具有的能力，并且连续不断地发现更多的服务。UDDI 定义了支持这种业务功能所需的要求。它使得公司能够快速、容易、动态地相互发现并相互协作。它创建了一种手段，使一个公司能够描述其服务与过程，并且动态地发现其他公司并与之交互，而且所有这些动作都通过 Internet 完成。当然，UDDI 也基于 SOAP、XML 与 HTTP。UDDI 支持 UDDI Programmer's API Specification 中的描述，这是一个定义了 30 多个 SOAP 消息的文档，这些消息被 Web 服务信息发布者所使用，也被那些查询各种可重用 Web 服务的公司所使用。UDDI 定义的 SOAP 消息允许 Web 服务提供者在 UDDI 注册表中注册自己。它们也能够帮助 Web 服务消费者发现与自己特定需求相匹配的服务提供者。最后，UDDI 有一个非常实际的发展模型。这些开发 UDDI 的公司承诺，不仅负责管理创建这些标准（一系列文档）的工作，而且负责实现一个公司注册表，作为中心网站。而且，它们承诺在自己的核心系统（例如 WebSphere 4.0）中实现这个注册表。

现在说明一个 UDDI 描述的实际内容。一个 UDDI 描述具有 4 个层次：businessEntity 层、businessService 层、bindingTemplate 层与 tModel 层。businessEntity 包括用来提供服务发布方信息的元素。每个服务发布方都可能发布多个 Web 服务，因此一个 businessEntity 可能与多个 businessService 实体相关联。每个 businessEntity 描述了某种技术服务体系，这些服务

通常通过 SOAP 消息传递接口提供。businessService 实体可能与多个 bindingTemplate 元素链接，分别提供关于某个表项的服务提供者的技术信息。bindingTemplate 元素有对 tModel 元素的引用，用来提供关于此服务用法的描述与规范（包括各种用来描述服务的属性），并且可用于搜索过程与资格认证过程。

图 37-1 显示了一个关于 UDDI 中 4 个层次的例子。在这个例子中，ServiceCompany 为顾客提供各种劳务相关的服务。假设 ServiceCompany 负责管理电缆公司、数字碟形卫星天线提供者以及 DSL 提供者所使用的劳务技工。ServiceCompany 的顾客将这些服务提供到家庭，但它们自己并没有用来从事这些工作的劳务技工。相反，它们将这些工作实际地承包出去。由于 ServiceCompany 处理许多顾客，并且每个顾客都有不同的系统，因此它使用 ViryaNet Service Hub 给出自己的在线供给捕获系统作为 Web 服务。因此，它发布了一个 businessEntity 元素，当中包括公司的所有信息。这样就可能使得更多提供者发现 ServiceCompany 并最终成为其顾客。ServiceCompany 主要有 3 种 Web 服务，分别对应它的 3 个工作领域（也就是，每个领域的劳务技工以及在线订单捕获系统分别对应一个 Web 服务）。最后，在 DSL 家族，存在 3 个 bindingTemplate，用来允许 ServiceCompany 的顾客发出 DSL 安装、维修与取消请求。其中每个都可能涉及不同的过程与/或数据结构，并且这些都可以通过单独的 tModel 元素定义。

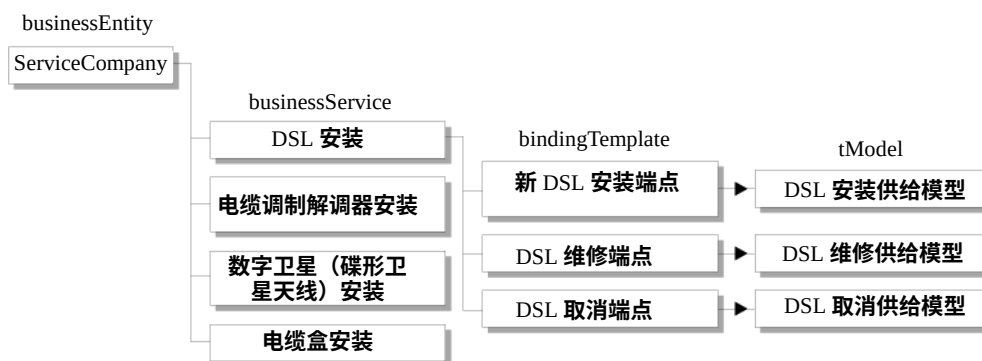


图 37-1 UDDI 实体层示例

37.3 WebSphere 对 Web 服务的支持

IBM 是本章所描述的 3 种技术（SOAP，WSDL 与 UDDI）的领导者。这 3 种技术形成了 IBM 面向服务的体系结构，如图 37-2 所示。作为软件厂商，IBM 承诺在 WebSphere 产品家族中为每个标准提供实现，包括提供开发工具以及提供运行时系统（WebSphere Application Server）。在 WebSphere 4.0 以及新版本的 WebSphere Studio 与 VisualAge for Java 中提供了对 SOAP，WSDL 与 UDDI 的支持。WebSphere 是业界能够支持所有这 3 个标准的第一个全面

的 XML 与 Web 服务开发环境。



图 37-2 IBM 的 Web 服务蓝图

本章的其余部分将概述 SOAP, WSDL 与 UDDI 机制。目标是提供关于这三者如何共同工作的基本概念，说明如何使用它们以及每个技术集合所具有的结构。

这些例子来自 ViryaNet 的 Service Hub 平台。它们所涉及的公司包括服务提供公司（例如那些在家庭与公司进行安装的公司）、那些提供维修设备与更换设备的技工的公司，以及那些管理分布式劳工与其他服务人员的公司。这些公司都在不断地寻找更有效的业务模型，在这种模型下，常常需要独立的组织之间进行业务过程的合作，这实际上就是利用 Web 服务能够获得的好处。

在复杂的服务业务中，组织需要更加廉价并且更加高效的业务传递模型。这意味着，组织常常需要外购部分服务传递业务。例如，有的公司需要节省人力资源但同时要保持原有的顾客关系，这样就需要从那些能够管理劳务的公司外购劳动力。这样，就意味着劳动力信息需要从一个组织传递到另一个组织。在过去，这需要涉及各种系统之间的点对点集成，可以想象，那是一件很痛苦的事情。这实际上就是绑定到一个劳务输出公司，并且形成了一种阻碍，使得这种模型排除了劳务竞标机制，难以与多个劳务服务提供者进行合作，这实际上也不符合劳务市场的概念。由于所有这些概念是 B2B 交互的关键，并且它们是业务效率的推动者，因此它们对服务业具有非常重要的作用。这里所显示的例子涉及“服务订单”信息。在一种可能的情况下，某个组织可能捕获了与订单相关的信息（例如顾客位置、要求执行的服务，以及各种权利（例如应该何时提供服务））。一旦拥有顾客关系的公司捕获了这些信息，它们就能够与劳务提供者通信并创建一个呼叫实体。这些动作是通过发送 SOAP 请求来完成的，也就是说，激活由第二家公司提供的 Web 服务。利用 WSDL 定义 Web 服务，并且使用 UDDI 注册表发现这个 Web 服务。

37.4 SOAP 消息传递

程序清单 37-1 显示了在 ViryaNet Service Hub 中创建服务订单的一条 SOAP 消息。需要使用这个文档的应用情况是，有一个外部系统向 Service Hub（特别地，就是向劳务管理系统）发送一条消息。这个外部系统可能是物资供应系统、远程通信业的故障单管理系统、其他纵向市场的帮助桌面或呼叫中心系统。传递给 Service Hub 的信息描述了顾客、设备与所需的服务。这些数据将被发送给劳务管理系统，从而使现场工程师能够被派到现场去服务订单。

程序清单 37-1 SOAP 消息示例：请求

```
POST /CreateCall HTTP/1.1
Host: virya.viryanet.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 953
SOAPAction:
http://virya.viryanet.com/servlet/ServletBroker?provider=Call&service=createCall

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <viryanet:CreateCall xmlns:viryanet="http://virya.viryanet.com
      /servlet/ApiRepository">
      <problemDescription type="STRING">
        <![CDATA[BURNING SMELL]]>
      </problemDescription>
      <warrantyFlag type="BOOLEAN">false</warrantyFlag>
      <subContractorFlag type="BOOLEAN">false</subContractorFlag>
      <replacementTerm type="STRING">
        <![CDATA[123123]]>
      </replacementTerm>
      <equipmentNumber type="STRING">
        <![CDATA[1234]]>
      </equipmentNumber>
      <externalSystemId type="STRING">
        <![CDATA[2143234324]]>
      </externalSystemId>
      <contactName type="STRING">
        <![CDATA[Jane Doe]]>
      </contactName>
    </viryanet:CreateCall>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

```
<site type="STRING">
  <![CDATA[ML1]]>
</site>
<repeatCallFlag type="BOOLEAN">false</repeatCallFlag>
<responseOriginalValue type="STRING">
  <![CDATA[12]]>
</responseOriginalValue>
<responseTerm type="STRING">
  <![CDATA[1]]>
</responseTerm>
<serviceWindowTerm type="STRING">
  <![CDATA[1]]>
</serviceWindowTerm>
<centerLogged type="STRING">
  <![CDATA[US East]]>
</centerLogged>
<loanOriginalValue type="STRING">
  <![CDATA[12]]>
</loanOriginalValue>
<node type="LONG">123213</node>
<overheadPrice type="FLOAT">12.0</overheadPrice>
<contract type="STRING">
  <![CDATA[CONT1]]>
</contract>
<alertColorCode type="STRING">
  <![CDATA[RED]]>
</alertColorCode>
<subContractorContract type="STRING">
  <![CDATA[1]]>
</subContractorContract>
<phoneNumber type="STRING">
  <![CDATA[123-123-1234]]>
</phoneNumber>
<chargeFlag type="BOOLEAN">false</chargeFlag>
<quoteNumber type="INTEGER">1233123123</quoteNumber>
<preventiveMaintenanceCallFlag type="BOOLEAN">
  false
</preventiveMaintenanceCallFlag>
<acceptFlag type="INTEGER">1</acceptFlag>
<b0Tag type="STRING">
  <![CDATA[1]]>
</b0Tag>
<assetNumber type="STRING">
  <![CDATA[2343214]]>
```

```
</assetNumber>
<faultGrade type="STRING">
  <![CDATA[FAULT1]]>
</faultGrade>
<loanUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</loanUnitOfMeasureCode>
<loanValue type="STRING">
  <![CDATA[12]]>
</loanValue>
<service type="STRING">
  <![CDATA[24x7]]>
</service>
<responseValue type="STRING">
  <![CDATA[12]]>
</responseValue>
<partRevision type="STRING">
  <![CDATA[1]]>
</partRevision>
<fixOriginalValue type="STRING">
  <![CDATA[12]]>
</fixOriginalValue>
<alsoCreateAction type="BOOLEAN">false</alsoCreateAction>
<serviceOrderId type="STRING">
  <![CDATA[1]]>
</serviceOrderId>
<fixUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</fixUnitOfMeasureCode>
<downtime type="STRING">
  <![CDATA[1]]>
</downtime>
<callStatusCode type="STRING">
  <![CDATA[0]]>
</callStatusCode>
<symptomCode type="STRING">
  <![CDATA[1]]>
</symptomCode>
<quantity type="INTEGER">10</quantity>
<part type="STRING">
  <![CDATA[ROUTER1]]>
</part>
<quoteFlag type="BOOLEAN">false</quoteFlag>
<priority type="INTEGER">1</priority>
```

```
<entitlementRemarks type="STRING">
  <![CDATA[scratched]]>
</entitlementRemarks>
<purchaseOrder type="STRING">
  <![CDATA[21384098]]>
</purchaseOrder>
<zip type="STRING">
  <![CDATA[01745]]>
</zip>
<callTypeCode type="STRING">
  <![CDATA[0]]>
</callTypeCode>
<outstandingStatusCode type="BOOLEAN">false
</outstandingStatusCode>
<serialNumber type="STRING">
  <![CDATA[3475987435]]>
</serialNumber>
<subContractorPo type="STRING">
  <![CDATA[1]]>
</subContractorPo>
<user type="STRING">
  <![CDATA[rbn]]>
</user>
<cust type="STRING">
  <![CDATA[ML]]>
</cust>
<fixValue type="STRING">
  <![CDATA[12]]>
</fixValue>
<projectId type="STRING">
  <![CDATA[2134]]>
</projectId>
<actualResponseTime type="STRING">
  <![CDATA[17 Nov 2000 7:00]]>
</actualResponseTime>
<loanTerm type="STRING">
  <![CDATA[1]]>
</loanTerm>
<fixTerm type="STRING">
  <![CDATA[1]]>
</fixTerm>
<responseUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</responseUnitOfMeasureCode>
```



```

        <clientLog type="STRING">
            <![CDATA[1]]>
        </clientLog>
        <downFlag type="BOOLEAN">false</downFlag>
    </viryanet:CreateCall>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

在程序清单 37-1 与 37-2 中可以看到，SOAP 消息通常被封装在 HTTP 请求与响应中。数据本身是一个 XML 文档，这个 XML 文档被封装在定义了路由选择与消息传递信息的 XML 封皮（envelope）中。

程序清单 37-2 SOAP 消息示例：响应

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 423

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <viryanet:CreateCall xmlns:viryanet="http://virya.viryanet.com
      /servlet/ApiRepository">
      <problemDescription type="STRING">
        <![CDATA[BURNING SMELL]]>
      </problemDescription>
      <warrantyFlag type="BOOLEAN">false</warrantyFlag>
      <subContractorFlag type="BOOLEAN">false</subContractorFlag>
      <replacementTerm type="STRING">
        <![CDATA[123123]]>
      </replacementTerm>
      <equipmentNumber type="STRING">
        <![CDATA[1234]]>
      </equipmentNumber>
      <externalSystemId type="STRING">
        <![CDATA[2143234324]]>
      </externalSystemId>
      <contactName type="STRING">
        <![CDATA[Jane Doe]]>
      </contactName>
      <site type="STRING">
        <![CDATA[ML1]]>
      </site>
    </viryanet:CreateCall>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

```

<repeatCallFlag type="BOOLEAN">false</repeatCallFlag>
<responseOriginalValue type="STRING">
  <![CDATA[12]]>
</responseOriginalValue>
<responseTerm type="STRING">
  <![CDATA[1]]>
</responseTerm>
<serviceWindowTerm type="STRING">
  <![CDATA[1]]>
</serviceWindowTerm>
<centerLogged type="STRING">
  <![CDATA[US East]]>
</centerLogged>
<loanOriginalValue type="STRING">
  <![CDATA[12]]>
</loanOriginalValue>
<node type="LONG">123213</node>
<overheadPrice type="FLOAT">12.0</overheadPrice>
<contract type="STRING">
  <![CDATA[CONT1]]>
</contract>
<alertColorCode type="STRING">
  <![CDATA[RED]]>
</alertColorCode>
<subContractorContract type="STRING">
  <![CDATA[1]]>
</subContractorContract>
<phoneNumber type="STRING">
  <![CDATA[123-123-1234]]>
</phoneNumber>
<chargeFlag type="BOOLEAN">false</chargeFlag>
<quoteNumber type="INTEGER">1233123123</quoteNumber>
<preventiveMaintenanceCallFlag
type="BOOLEAN">false</preventiveMaintenanceCallFlag>
<acceptFlag type="INTEGER">1</acceptFlag>
<b0Tag type="STRING">
  <![CDATA[1]]>
</b0Tag>
<assetNumber type="STRING">
  <![CDATA[2343214]]>
</assetNumber>
<faultGrade type="STRING">
  <![CDATA[FAULT1]]>
</faultGrade>

```

```
<loanUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</loanUnitOfMeasureCode>
<loanValue type="STRING">
  <![CDATA[12]]>
</loanValue>
<service type="STRING">
  <![CDATA[24x7]]>
</service>
<responseValue type="STRING">
  <![CDATA[12]]>
</responseValue>
<partRevision type="STRING">
  <![CDATA[1]]>
</partRevision>
<fixOriginalValue type="STRING">
  <![CDATA[12]]>
</fixOriginalValue>
<alsoCreateAction type="BOOLEAN">false</alsoCreateAction>
<serviceOrderId type="STRING">
  <![CDATA[1]]>
</serviceOrderId>
<fixUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</fixUnitOfMeasureCode>
<downtime type="STRING">
  <![CDATA[1]]>
</downtime>
<callStatusCode type="STRING">
  <![CDATA[0]]>
</callStatusCode>
<symptomCode type="STRING">
  <![CDATA[1]]>
</symptomCode>
<quantity type="INTEGER">10</quantity>
<part type="STRING">
  <![CDATA[ROUTER1]]>
</part>
<quoteFlag type="BOOLEAN">false</quoteFlag>
<priority type="INTEGER">1</priority>
<entitlementRemarks type="STRING">
  <![CDATA[scratched]]>
</entitlementRemarks>
<purchaseOrder type="STRING">
```

```
<![CDATA[21384098]]>
</purchaseOrder>
<zip type="STRING">
  <![CDATA[01745]]>
</zip>
<callTypeCode type="STRING">
  <![CDATA[0]]>
</callTypeCode>
<outstandingStatusCode type="BOOLEAN">
  false
</outstandingStatusCode>
<serialNumber type="STRING">
  <![CDATA[3475987435]]>
</serialNumber>
<subContractorPo type="STRING">
  <![CDATA[1]]>
</subContractorPo>
<user type="STRING">
  <![CDATA[rbn]]>
</user>
<cust type="STRING">
  <![CDATA[ML]]>
</cust>
<fixValue type="STRING">
  <![CDATA[12]]>
</fixValue>
<projectId type="STRING">
  <![CDATA[2134]]>
</projectId>
<actualResponseTime type="STRING">
  <![CDATA[17 Nov 2000 7:00]]>
</actualResponseTime>
<loanTerm type="STRING">
  <![CDATA[1]]>
</loanTerm>
<fixTerm type="STRING">
  <![CDATA[1]]>
</fixTerm>
<responseUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</responseUnitOfMeasureCode>
<clientLog type="STRING">
  <![CDATA[1]]>
</clientLog>
```

```

        <downFlag type="BOOLEAN">false</downFlag>
    </viryanet:CreateCall>
    <viryanet:CreateCallResult>
        <date null="" type="NULL"/>
        <status type="STRING">
            <![CDATA[0]]>
        </status>
        <id type="STRING">
            <![CDATA[886]]>
        </id>
    </viryanet:CreateCallResult>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

SOAP 消息通常由 3 部分组成：封皮、头部与体。封皮与体是 SOAP 消息中必须有的元素，头部是可选的。封皮是 XML 文档的根元素，体是消息的有效负载。体是封皮的子元素。头部元素是一种用来为消息添加属性的通用机制，并且不要求在通信双方之间（或消息传递各方的联盟之间）预先达成一致。头部属性用来告知接收者如何处理该消息。

SOAP 消息中的头部可以具有多种属性。其中，有一个称为 `mustUnderstand` 的属性。如果在头部元素中将这个属性的值设置为 1，那么消息发送者实际上就要求使用元素应该遵守的语义进行处理。如果消息接收者不能遵守这个指令，它在处理该消息时就必将失败。头部元素的另一个重要属性就是 `actor` 属性。从发送者向最终接收者发出的 SOAP 文档可能在传输过程中经过了多个组织，它们可能基于这个 SOAP 文档进行了某些处理（也可能没有）。因此这些动作者（actor）就可能实现了一些简单的功能，例如路由，或者它们可能解释了某些消息语义。`actor` 属性允许对头部元素进行标记，并且标记方法是强制要求中介动作者处理某个元素，然后将这个元素从 SOAP 文档中删除。

在 SOAP 消息中进行的数据编码是比较直观的；它类似于大多数信息系统（例如数据库与面向对象语言）所使用的编码方式。类型可以是原始类型，也可以是复合类型；复合元素可以由几部分构成。这是一种递归的结构，在这种结构下，复合类型可能由其他复合类型构成，或者由原始类型构成。复合类型最终变成了一棵大树（或层次），其中的叶节点就是简单（原始）类型。类型本身遵守 XML Schema 方法学，并且允许消息是真正的自描述的。这里假设 SOAP 都建立在这种关于类型的层次结构上，但它们并不能说是 SOAP 的内在部分。SOAP 更关注于消息传递元素与路由元素。它在消息体中使用 XML 与 XML Schema，消息体就是所谓的消息有效负载（message payload）。下面给出了常见的编码方法示例（在这里没有使用 Schema）：

```

<SOAP-ENV:Body>
    <ServiceOrder>
        <ID>213408798</ID>
        <CustomerRef>Fleet</CustomerRef>
    </ServiceOrder>
</SOAP-ENV:Body>

```

```

        <TimeWindow>
            <From>2001:4:6:13:0:0</From>
            <To>2001:4:6:15:0:0</To>
        </TimeWindow>
        ...
    </ServiceOrder>
</SOAP-ENV:Body>

```

SOAP 并不直接与 HTTP 相关。它能够在使用多种协议（而不仅仅是 HTTP）的多种传送设备上被传递。但是，HTTP 在 SOAP 的生命历程中扮演了一个非常重要的角色。应该记得，其主要原型就是 HTTP XML。因此，虽然 SOAP 消息能够通过许多其他协议（当然包括，邮件协议）被传递，但最常用的方法还是将 SOAP 绑定在 HTTP 上。因此，SOAP 规范文档中的大部分内容都用来描述如何在 HTTP 上使用 SOAP。

SOAP 在许多方面都继续使用 HTTP 头部，但它添加了一些特定于 SOAP 的头部。SOAP 规范要求，HTTP 请求/响应对中的 SOAP 消息总是使用 text/xml 内容类型。除此之外，大多数 SOAP 头部都与 HTTP 头部相同。另一个附加的头部就是 SOAPAction 头部字段，另外指定一个 URI，这个 URI 可帮助告知该消息的意图。SOAP 响应（包括头部）使用与 HTTP 请求相同的结构。

除了使用普通的 HTTP 请求与响应之外，SOAP 也定义了一个 HTTP 扩展框架。按照这种方式发送的消息将强制 SOAP 消息接收者将请求作为 SOAP 操作进行处理，而不是让接收者自己决定如何处理该请求。其基本区别在于，SOAP 请求使用 M-POST 请求类型而不是 POST 请求类型。例如，如果在 HTTP 扩展框架中调用程序清单 37-1 所显示的消息，那么它将成为程序清单 37-3 所示的形式。

程序清单 37-3 WSDL 的服务文档结构

```

M-POST /CreateCall HTTP/1.1
Man: "http://schemas.xmlsoap.org/soap/envelope/"; ns=VN
Content-Type: text/xml; charset="utf-8"
Content-Length: 953
VN-SOAPAction:
http://virya.viryanet.com/servlet/ServletBroker?provider=Call&service=createCall

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
    <viryanet:CreateCall xmlns:viryanet="http://virya.viryanet.com
      /servlet/ApiRepository">
      <problemDescription type="STRING">
        <![CDATA[BURNING SMELL]]>

```

```
</problemDescription>
<warrantyFlag type="BOOLEAN">false</warrantyFlag>
<subContractorFlag type="BOOLEAN">false</subContractorFlag>
<replacementTerm type="STRING">
  <![CDATA[123123]]>
</replacementTerm>
<equipmentNumber type="STRING">
  <![CDATA[1234]]>
</equipmentNumber>
<externalSystemId type="STRING">
  <![CDATA[2143234324]]>
</externalSystemId>
<contactName type="STRING">
  <![CDATA[Jane Doe]]>
</contactName>
<site type="STRING">
  <![CDATA[ML1]]>
</site>
<repeatCallFlag type="BOOLEAN">false</repeatCallFlag>
<responseOriginalValue type="STRING">
  <![CDATA[12]]>
</responseOriginalValue>
<responseTerm type="STRING">
  <![CDATA[1]]>
</responseTerm>
<serviceWindowTerm type="STRING">
  <![CDATA[1]]>
</serviceWindowTerm>
<centerLogged type="STRING">
  <![CDATA[US East]]>
</centerLogged>
<loanOriginalValue type="STRING">
  <![CDATA[12]]>
</loanOriginalValue>
<node type="LONG">123213</node>
<overheadPrice type="FLOAT">12.0</overheadPrice>
<contract type="STRING">
  <![CDATA[CONT1]]>
</contract>
<alertColorCode type="STRING">
  <![CDATA[RED]]>
</alertColorCode>
<subContractorContract type="STRING">
  <![CDATA[1]]>
```

```

</subContractorContract>
<phoneNumber type="STRING">
  <![CDATA[123-123-1234]]>
</phoneNumber>
<chargeFlag type="BOOLEAN">false</chargeFlag>
<quoteNumber type="INTEGER">1233123123</quoteNumber>
<preventiveMaintenanceCallFlag type="BOOLEAN">
  false
</preventiveMaintenanceCallFlag>
<acceptFlag type="INTEGER">1</acceptFlag>
<b0Tag type="STRING">
  <![CDATA[1]]>
</b0Tag>
<assetNumber type="STRING">
  <![CDATA[2343214]]>
</assetNumber>
<faultGrade type="STRING">
  <![CDATA[FAULT1]]>
</faultGrade>
<loanUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</loanUnitOfMeasureCode>
<loanValue type="STRING">
  <![CDATA[12]]>
</loanValue>
<service type="STRING">
  <![CDATA[24x7]]>
</service>
<responseValue type="STRING">
  <![CDATA[12]]>
</responseValue>
<partRevision type="STRING">
  <![CDATA[1]]>
</partRevision>
<fixOriginalValue type="STRING">
  <![CDATA[12]]>
</fixOriginalValue>
<alsoCreateAction type="BOOLEAN">false</alsoCreateAction>
<serviceOrderId type="STRING">
  <![CDATA[1]]>
</serviceOrderId>
<fixUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</fixUnitOfMeasureCode>

```



```
<downtime type="STRING">
  <![CDATA[1]]>
</downtime>
<callStatusCode type="STRING">
  <![CDATA[0]]>
</callStatusCode>
<symptomCode type="STRING">
  <![CDATA[1]]>
</symptomCode>
<quantity type="INTEGER">10</quantity>
<part type="STRING">
  <![CDATA[ROUTER1]]>
</part>
<quoteFlag type="BOOLEAN">false</quoteFlag>
<priority type="INTEGER">1</priority>
<entitlementRemarks type="STRING">
  <![CDATA[scratched]]>
</entitlementRemarks>
<purchaseOrder type="STRING">
  <![CDATA[21384098]]>
</purchaseOrder>
<zip type="STRING">
  <![CDATA[01745]]>
</zip>
<callTypeCode type="STRING">
  <![CDATA[0]]>
</callTypeCode>
<outstandingStatusCode type="BOOLEAN">
  false
</outstandingStatusCode>
<serialNumber type="STRING">
  <![CDATA[3475987435]]>
</serialNumber>
<subContractorPo type="STRING">
  <![CDATA[1]]>
</subContractorPo>
<user type="STRING">
  <![CDATA[rbn]]>
</user>
<cust type="STRING">
  <![CDATA[ML]]>
</cust>
<fixValue type="STRING">
  <![CDATA[12]]>
```

```
</fixValue>
<projectId type="STRING">
  <![CDATA[2134]]>
</projectId>
<actualResponseTime type="STRING">
  <![CDATA[17 Nov 2000 7:00]]>
</actualResponseTime>
<loanTerm type="STRING">
  <![CDATA[1]]>
</loanTerm>
<fixTerm type="STRING">
  <![CDATA[1]]>
</fixTerm>
<responseUnitOfMeasureCode type="STRING">
  <![CDATA[1]]>
</responseUnitOfMeasureCode>
<clientLog type="STRING">
  <![CDATA[1]]>
</clientLog>
<downFlag type="BOOLEAN">false</downFlag>
</viryanet:CreateCall>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

37.5 WSDL

WSDL 是一种用来描述 Web 上的服务的 XML 格式。WSDL 将服务看作是一系列由发送与接收 XML 文档的客户应用程序所调用的端点集合。WSDL 以一种抽象的兼具实际实现的方式描述服务。由于 WSDL 遵守与 SOAP 相同的一般模式，因此 WSDL 规范描述了如何在 SOAP 环境中使用 WSDL。虽然 WSDL 并没有被限制为必须使用 SOAP，但 WSDL 并不知道有关协议与平台的信息，因此，编写遵守 WSDL 软件的人有可能使用的是 SOAP。

37.5.1 WSDL 中的抽象

WSDL 定义了 7 种抽象，分别是：类型、消息、操作、端口类型、绑定、端口与服务。服务是 WSDL 所关注的实体，并且 WSDL 文档的目标是描述服务的结构。服务（service）被描述为端口的集合，端口代表网络端点（也就是，能够通过 Web 请求并提供服务的例程）。端口（port）被定义为 Web 上的一个地址，它与一个绑定组合在一起，构成端点所支持的协议与数据格式。绑定由一个更抽象的实体（端口类型，port type）来描述，它被定义成操作

的集合。通过端点交换的数据也被定义成消息的抽象。最后，在消息中传递的数据的定义封装在类型定义之中。

WSDL 中的类型根据 XML Schema 规范来描述，WSDL 除了定义一般的绑定方法之外，还定义了一系列特定的绑定。对这些具体绑定的扩展主要为 SOAP、HTTP GET 请求与 HTTP POST 请求、MIME 服务。

程序清单 37-4 显示了 WSDL 规范中定义的典型的服务文档结构。

程序清单 37-4 典型的 WSDL 文档

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri">?

  <import namespace="uri" location="uri"/>*

  <wsdl:documentation/>?

  <wsdl:types>?
    <wsdl:documentation/>?
    <xsd:schema/>*
    extensibility element -->*
  </wsdl:types>

  <wsdl:message name="nmtoken">*
    <wsdl:documentation/>?
    <part name="nmtoken" element="qname"? type="qname"?/>*
  </wsdl:message>

  <wsdl:portType name="nmtoken">*
    <wsdl:documentation/>?
    <wsdl:operation name="nmtoken">*
      <wsdl:documentation/>?
      <wsdl:input name="nmtoken"? message="qname"?>?
        <wsdl:documentation/>?
      </wsdl:input>
      <wsdl:output name="nmtoken"? message="qname"?>?
        <wsdl:documentation/>?
      </wsdl:output>
      <wsdl:fault name="nmtoken" message="qname">*
        <wsdl:documentation/>?
      </wsdl:fault>
    </wsdl:operation>
  </wsdl:portType>

  <wsdl:binding name="nmtoken" type="qname">*
```

```

<wsdl:documentation/>?
extensibility element -->*
<wsdl:operation name="nmtoken">*
  <wsdl:documentation/>?
  extensibility element -->*
  <wsdl:input name="nmtoken"?>?
    <wsdl:documentation/>?
    extensibility element -->
  </wsdl:input>
  <wsdl:output name="nmtoken"?>?
    <wsdl:documentation/>?
    extensibility element -->*
  </wsdl:output>
  <wsdl:fault name="nmtoken">*
    <wsdl:documentation/>?
    extensibility element -->*
  </wsdl:fault>
</wsdl:operation>
</wsdl:binding>

<wsdl:service name="nmtoken">*
  <wsdl:documentation/>?
  <wsdl:port name="nmtoken" binding="qname">*
    <wsdl:documentation/>?
    extensibility element -->
  </wsdl:port>
  extensibility element -->
</wsdl:service>

extensibility element -->*

</wsdl:definitions>

```

程序清单 37-5显示了一个关于 WSDL 文档的例子,这个例子描述了在 ViryaNet 的 ServiceHub 平台上的订单创建服务。需要说明的是,服务利用下面 5 个主要的元素类定义:

- **Types**: 定义了消息中所使用的数据类型。
- **Messages**: 逻辑地定义了应该在端点之间传递什么。
- **Port types**: 指定操作的抽象。
- **Bindings**: 定义底层的协议与数据格式。
- **Ports**: 指定绑定的地址。

最后,还有一个关于服务自身的定义,它结合使用上述 5 种元素类型。

程序清单 37-5 createCall 消息的 WSDL 定义

```
<?xml version="1.0"?>
<definitions name="Call"

    targetNamespace=
        "http://viry.viryanet.com/servlet/ApiRepository/call.wsdl"
    xmlns:tns=
        "http://viry.viryanet.com/servlet/ApiRepository/call.xsd"
    xmlns:xsd_vn=
        "http://viry.viryanet.com/servlet/ApiRepository/call.wsdl"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns="http://schemas.xmlsoap.org/wsdl/">

    <types>
        <schema targetNamespace="http://viry.viryanet.com/servlet/
            ApiRepository/call.wsdl"
            xmlns="http://www.w3.org/1999/XMLSchema">
            <element name="CallRequest">
                <complexType>
                    <all>
                        <element name="problemDescription " type="string"/>
                        <element name="warrantyFlag" type="boolean"/>
                        <element name="subContractorFlag" type="boolean"/>
                        <element name="replacementTerm" type="string"/>
                        <element name="equipmentNumber" type="string"/>
                        <element name="externalSystemId" type="string"/>
                        <element name="contactName" type="string"/>
                        <element name="site" type="string"/>
                        <element name="repeatCallFlag" type="string"/>
                        <element name="responseOriginalValue" type="string"/>
                        <element name="responseTerm" type="string"/>
                        <element name="serviceWindowTerm" type="string"/>
                        <element name="centerLogged" type="string"/>
                        <element name="loanOriginalValue" type="string"/>
                        <element name="node" type="long"/>
                        <element name="overheadPrice" type="float"/>
                        <element name="contract" type="string"/>
                        <element name="alertColorCode" type="string"/>
                        <element name="subContractorContract" type="string"/>
                        <element name="phoneNumber" type="string"/>
                        <element name="chargeFlag" type="boolean"/>
                        <element name="quoteNumber" type="integer"/>
                        <element name="acceptFlag" type="integer"/>
                    </all>
                </complexType>
            </element>
        </schema>
    </types>

```

```

    <element name="b0Tag" type="string"/>
    <element name="assetNumber" type="string"/>
    <element name="faultGrade" type="string"/>
    <element name="loanUnitOfMeasureCode" type="string"/>
    <element name="loanValue" type="string"/>
    <element name="service" type="string"/>
    <element name="responseValue" type="string"/>
    <element name="partRevision" type="string"/>
    <element name="fixOriginalValue" type="string"/>
    <element name="alsoCreateAction" type="boolean"/>
    <element name="serviceOrderId" type="string"/>
    <element name="fixUnitOfMeasureCode" type="string"/>
    <element name="downtime" type="string"/>
    <element name="callStatusCode" type="string"/>
    <element name="symptomCode" type="string"/>
    <element name="quantity" type="integer"/>
    <element name="part" type="string"/>
    <element name="quoteFlag" type="boolean"/>
    <element name="priority" type="integer"/>
    <element name="entitlementRemarks" type="string"/>
    <element name="purchaseOrder" type="string"/>
    <element name="purchaseOrder" type="string"/>
    <element name="zip" type="string"/>
    <element name="callTypeCode" type="string"/>
    <element name="outstandingStatusCode" type="string"/>
    <element name="serialNumber" type="string"/>
    <element name="subContractorPo" type="string"/>
    <element name="user" type="string"/>
    <element name="cust" type="string"/>
    <element name="fixValue" type="string"/>
    <element name="projectId" type="string"/>
    <element name="actualResponseTime" type="string"/>
    <element name="loanTerm" type="string"/>
    <element name="fixTerm" type="string"/>
    <element name="loanTerm" type="string"/>
    <element name="clientLog" type="string"/>
    <element name="downFlag" type="boolean"/>
  </all>
</complexType>
</element>
<element name="CallResponse">
  <complexType>
    <all>
      <element name="date" type="string"/>
    </all>
  </complexType>
</element>

```

```

        <element name="status" type="string"/>
        <element name="id" type="string"/>
    </all>
</complexType>
</element>

</schema>
</types>

<message name="InputCall">
    <part name="body" element="xsd_vn:CallRequest"/>
</message>

<message name="OutputCall">
    <part name="body" element="xsd_vn:Response"/>
</message>

<portType name="CreateCallPortType">
    <operation name="CreateCall">
        <input message="InputCall"/>
        <output message="OutputCall"/>
    </operation>
</portType>

<portType name="UpdateCallPortType">
    <operation name="CreateCall">
        <input message="InputCall"/>
        <output message="OutputCall"/>
    </operation>
</portType>

<binding name="CreateCallSoapBinding" type="CreateCallType">
    <soap:binding style="document"
transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="CreateCall">
        <soap:operation soapAction=
"http://viry.viryanet.com/servlet/
ServletBroker?provider=Call&service=createCall"/>
        <input>
            <soap:body use="literal" namespace="
http://viry.viryanet.com/servlet/ApiRepository/call.xsd"
encodingStyle="http://schemas.xmlsoap.org/soap/encoding/" />
        </input>
    </operation>
</binding>

```

```

        <output>
            <soap:body use="literal" namespace="
http://viry.viryanet.com/servlet/ApiRepository/call.xsd"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        </output>
    </operation>
</binding>

    <binding name="UpdateCallSoapBinding" type="UpdateCallType">
        <soap:binding style="document" transport=
"http://schemas.xmlsoap.org/soap/http">
            <operation name="CreateCall">
                <soap:operation soapAction="
http://viry.viryanet.com/servlet/
ServletBroker?provider=Call&service=updateCall">
                    <input>
                        <soap:body use="literal" namespace="http://viry.
viryanet.com/servlet/ApiRepository/call.xsd"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                    </input>
                    <output>
                        <soap:body use="literal" namespace="
http://viry.viryanet.com/servlet/ApiRepository/call.xsd"

encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
                    </output>
                </operation>
            </binding>

    <service name="CallService">
        <documentationCall creation/update service/documentation>
        <port name="CreateCallPort" binding="tns:CreateCallSoapBinding ">
            <soap:address location="http://viry.viryanet.com/
servlet/ServletBroker">
        </port>
        <port name="UpdateCallPort" binding="tns:UpdateCallSoapBinding ">
            <soap:address location="http://viry.viryanet.com/
servlet/ServletBroker">
        </port>

    </service>

```



```
</definitions>
```

37.5.2 WSDL 中的类型

WSDL 中的类型是 XML Schema 类型。一条消息包括一个或多个逻辑部分，其中每个部分都有一个类型。消息部分有一个名称与一个元素；其中，元素（element）属性指定了在这个文档中定义的类型。因此，消息定义了在一个通信系统之间使用的数据类型的抽象。应该记得，端口类型是一个指定名称的抽象操作与消息的集合。每个端口类型都有一个名称与几个可选的属性，例如用来定义端点是否支持单向、请求-响应、要求-响应还是通知类型的通信的属性。

如果端口类型定义了一个单向操作，那么其定义形式如下：

```
<wsdl:portType>
  <wsdl:operation name="createCall">
    <wsdl:input .../>
  </wsdl:operation>
</wsdl:portType>
```

如果端口类型定义了一个请求-响应或一个要求-响应操作，那么其定义形式如下：

```
<wsdl:portType>
  <wsdl:operation name="createCall">
    <wsdl:input ... />
    <wsdl:output ... />
    <wsdl:fault ... />
  </wsdl:operation>
</wsdl:portType>
```

如果端口类型定义了一个通知操作，那么其定义形式如下：

```
<wsdl:portType>
  <wsdl:operation name="createCall">
    <wsdl:output .../>
  </wsdl:operation>
</wsdl:portType>
```

绑定（binding）为一个特定端口类型所定义的操作与消息描述消息格式与协议细节。一个单独的端口类型可能存在多种绑定。端口是物理端点描述符，并且它们指定了一个物理位置，可以从这个物理位置接收服务。最后，服务将一系列相关的端口归组到一起，并且定义一个提供服务的域。

37.5.3 WSDL 中的 SOAP 绑定

由于 WSDL 在许多方面都与 SOAP 相似，因此，在 WSDL 规范中有专门描述 SOAP 绑定。

在 SOAP 相关的定义中，包括一个用来定义绑定到 SOAP 操作的方法、一个用来指定 SOAP 端点的地址的方法、一个用来在操作定义上使用 SOAPAction URL 的方法、一系列关于头部的定义（这些头部作为 SOAP 封皮的一部分被传递），以及一个用 XSD 指定 SOAP 根的方法。这些规范都使用 SOAP 名称空间。例如，一个 SOAP 绑定元素可以指定 SOAP 被用作底层协议。在这种情况下，WSDL 绑定部分具有如下形式：

```
<binding ... >
    <soap:binding ... />
</binding>
```

类似地，当使用 SOAP 扩展定义一个操作时，这个操作具有下列形式：

```
<operation ... >
    <soap:operation soapAction="..." ... />
</operation>
```

WSDL 规范的另一部分定义了 HTTP GET/POST 绑定，它允许支持一个更原始的实现。在这里，HTTP 用于消息传递，并且服务定义者可以使用 HTTP GET 或 HTTP POST 指定一个绑定、可以指定端口被实现为一个 HTTP 端点、还可以指定相对于 HTTP 端点的每个操作的地址。

37.6 UDDI

UDDI 定义了一个发布与发现关于 Web 服务的信息的方法。UDDI 定义了这样的底层结构，它强制电子交易、协作市场以及其他在线业务操作按照这个底层结构实现。UDDI 主要是为开发人员与设计者提供的工具，同时，它也是一个相对较低层次的定义，能够帮助标准化如何定义服务与发现服务。因此，UDDI 可以看作是 WSDL 与 SOAP 的补充，表现在将服务定义为利用 WSDL 来定义、利用 SOAP 来激活、然后使用 UDDI 来注册与发现，这样就更符合逻辑。

UDDI 关注的是服务的注册与发现。UDDI 依赖于分布式注册表的存在，使用 XML 实现并访问这个分布式注册表。UDDI 业务注册是通过一个 XML 文件完成的，这个文件描述了一个业务实体及其相关的 Web 服务。这一定义包括 3 个部分，它们分别等价于现实世界中由黄页提供的服务（允许地址与联系信息被发现）、黄页（允许进行分类）与绿页（用来提供关于已发布服务的技术信息）。所有这些信息都维护在 Web 上的 UDDI 注册表中。UDDI 定义了 XML 标准，允许开发人员在注册表中注册信息，并且发现与使用存储在注册表中的信息。在一个典型的 UDDI 使用情景中，有一方注册关于某个系统能够支持的 Web 服务的信息。通过网站，或者通过使用 UDDI 规范中定义的编程 API 的工具，将这些信息添加到 UDDI 注册表中（第 38 章将进一步讨论这个问题）。UDDI 注册表在逻辑上是一个数据库，但在物理上可能是（而且通常是）关于一系列物理注册表的分布式结构；毕竟，它应该具有好的

可伸缩性。UDDI 在本质上并不是一个发现工具，其目标也不是替代搜索引擎与门户。实际上，它仅仅定义一种结构，使得程序通过这种结构能够访问信息。

UDDI 中的结构

从技术上说，UDDI 包含一个关于 SOAP 消息的 XML Schema (或 XSD)，以及一个关于 API 的定义 (这些 API 用来执行 UDDI 操作)。Schema 定义允许编程者定义业务信息、服务信息、绑定信息以及服务规范信息。业务信息使用 businessEntity 元素，它支持“黄页”分类从而能够执行搜索。businessEntity 元素的子结构也定义了支持“黄页”类型的功能所需的信息。利用 businessService 元素来描述服务信息。这个元素是更高层次的元素 (例如，在每个 businessService 元素中都可能存在许多 Web 服务描述符)。这种元素允许在更高的层次上进行分段与分类。bindingTemplate 元素允许编程者提供关于地址的信息，通过这个地址能够联系到所需的 Web 服务。

businessEntity 结构通常代表业务信息以及这个业务能够提供的服务的信息。它们包括业务名称、业务的惟一标识符、业务实体的描述、联系信息以及 (最重要的) 所支持的 businessService 列表。businessService 有一个名称、描述、惟一关键字以及 bindingTemplate 列表。在每个 bindingTemplate 中有一个描述符的集合以及一个称为 accessPoint 的必要元素，这个元素描述了访问端点 (例如，一个 http:URL 或 mailto:URL)。这个元素也指向 tModel 实体，它定义了关于此服务的实际技术蓝图。

UDDI API 自身是基于 SOAP 的，意味着每个调用都具有 SOAP 消息的形式。调用通常定义了需要使用哪个函数。这个请求将被传递给 UDDI 注册表提供者，提供者使用一个 SOAP 文档作为回答。API 中包含 30 多条 SOAP 消息，这些消息可以划分为以下 3 组：

- 浏览 API：这些 API 用来发现元素并且发现与 UDDI 注册表中表项相关联的信息。在这个类型中包含的是 find_xx API。
- drill-down API 这些 API 是 UDDI 中被组织为层次结构的元素。一旦使用 find_xx API 发现了元素，就可以使用 get_xx API 进一步导航这个层次。
- 发布 API：这些 API 允许编程者和系统管理存储在 UDDI 注册表中的信息。

37.7 小 结

现在，读者已经对 Web 服务模型及其底层技术 (SOAP, WSDL 与 UDDI) 有所了解，这样就可以进入下一章的学习。下一章描述了 WebSphere 开发人员在 WebSphere Application Server 上开发与部署 Web 服务时可用的工具集合。

第 38 章 在 WebSphere 上部署 Web 服务

IBM 是推动 Web 服务发展的主要厂商之一。因此，IBM 所提供的许多 Web 服务都集成在其标志性产品（WebSphere）中，这一点毫不奇怪。在 WebSphere 4.0 版本中，对 Web 服务的支持已经深入到每个方面。这就意味着，有一些 Web Service 库是随着 WebSphere Application Server 一同发布的（特别是 Apache 的 SOAP 库与 UDDI4J 库，这些库用来实现 Web 服务、为 SOAP 进行的调用服务、调用其他 Web 服务，并允许通过 WebSphere 部署的应用程序与 UDDI 注册表交互，它们都用来发现与发布服务。

WebSphere 中提供的 Web 服务支持仅仅关注于部署方面。关于 Web 服务的工具以及开发人员生产率是 WebSphere Studio 的一部分，在 WebSphere Studio 中包括一个 WSDL 生成器。这些工具最初是作为 Web Services Toolkit 包提供的（现在仍然可以通过 AlphaWorks 获取），但目前已经对它们进行了改进并且打包作为 WebSphere Studio 的一部分。这些工具能够在 WebSphere 4.0（实际上，WebSphere 3.5 也可以）中使用，也可以在许多 UDDI 仓库中使用。本章将讨论 WebSphere 的部署方面，并且描述这样一个全过程：在这个过程中，一个实现了重要业务功能的 JAR 文件被作为 Web 服务部署在 WebSphere 服务器上。

38.1 调度服务

这里所讨论的 Web 服务示例涉及劳务管理领域中一个重要的业务功能。在这里对此服务进行简单的描述，因为理解 Web 服务在真实业务上下文中的使用很重要。我们很失望地发现，几乎遇到的每个关于 Web 服务的例子都在一定程度上与股票交易相关。这样很容易给人造成一种错觉，认为 Web 服务仅仅在这个领域有存在价值，但事实当然并非如此。

本章描述的这个例子来自一个称为 ViryaNet Service Hub 的真实应用程序，它将其核心功能作为支持 XML 的 API 与 Web 服务发布。

在本例中，将要作为一个 Web 服务而部署的是调度服务。服务的输入是一系列在某一天需要完成的任务（对于一个通用问题，并没有关于一天的限制，在这里主要是为了简化问题）。每个任务都有一个位置，用来代表应该在什么地方提供服务。可以将这想象为需要在某个地方安装或维修设备。每个任务也有一个预约窗口，在这个窗口中包括开始时间与结束时间，用来定义资源应该何时到位。这个系统的其他输入就是一系列资源（通常是一些现场工作人员）。每个资源都有一个家庭位置以及一些用来定义该资源可用的限制，这些限制用来表示该资源何时可以工作。服务的输出是一个任务分配列表。在这个列表中也包括一个顺序，用来告知每个资源应该按什么顺序去完成分配的任务。其目标是找到的一个有效的

分配方案，使得能够在满足所有可能的约束的同时，最小化行程时间。不同的调度意味着不同的代价，因此，一个好的调度算法能够节省成本。

调度服务就是应该被部署为 Web 服务的一个好例子。但这个问题是比较困难的；实际上，它是旅行推销员问题的一个更复杂版本，而旅行推销员问题实际上是一个 NP 完全问题。NP 完全性是复杂性理论中的核心概念之一。对于某个问题，如果给出了一个解决方案，而用来判断这个解决方案是否有效的算法中，如果存在多项式复杂度的算法，那么这个问题就属于 NP 类问题。如果这个算法也可以用于任何其他 NP 类问题的求解，那么这个问题就是一个 NP 完全问题。需要说明的是，这个多项式复杂度的算法仅仅用来验证一个解决方案，而不是用来找到这个解决方案。对于大多数人来说，这还不是非常有用的。但是，由于存在“完全性”，并且由于成千上万个非常优秀的人面对这样的 NP 完全问题，但仍然不能给出发现这些问题的最佳解决方案的快速算法，因此，可以假设没有一个算法能够对每种情况都提供最佳的解决方案，但可以使用启发信息来获得好的解决方案。这种算法在一定程度上还是具有重大价值的（但构建这种算法有一定的困难），并且这种算法对于较小的公司（这种公司不可能拥有自己构建这种算法所需的资源）具有重要的作用。但是，比创建这种算法更好的情况是通过一个中心网站提供一项服务，这项服务使得顾客能够非常容易地提交自己的问题数据，然后就得到顾客所需的调度方案。显然，这项服务是很超值的。

图 38-1 与图 38-2 显示了关于某一天（具有 30 个资源）两个可能的分配方案。图 38-1 给出的是没有应用智能优化启发信息的驾驶模式分配方案。每个带有阴影的行代表一个单独资源的驾驶路线，并且路线上的点代表任务的位置。图 38-2 显示了由相同的资源集合所处理的相同任务集合，但在这里的调度算法使用了智能启发信息。从图 38-3 所示的条形图中可以看出，在利用优化的调度算法之后，平均能够提高 25% 的生产率（这就意味着，公司能够节约 25% 的费用，或者，能够使用相同的费用完成更多的工作）。在许多情况下，这样就能够节省大量的成本，从而有效地确定 Web 服务的价格。

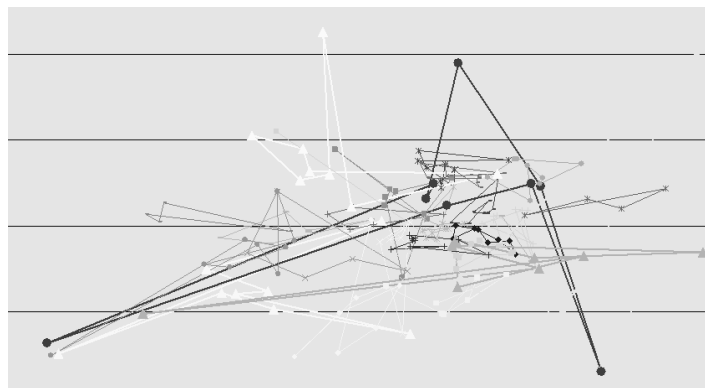


图 38-1 在应用优化的调度算法之前的行车路线

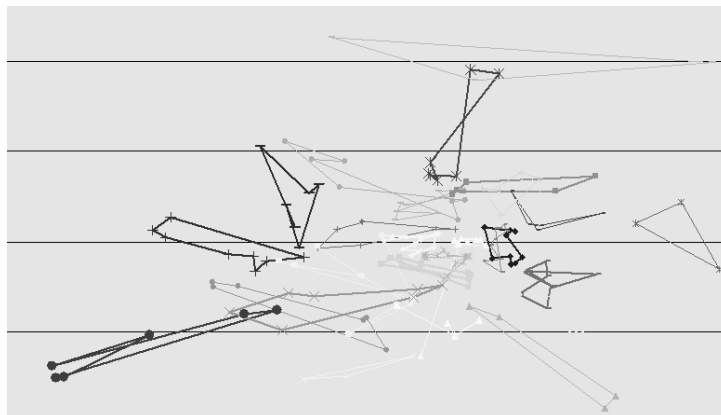


图 38-2 在利用优化的调度算法之后的行车路线

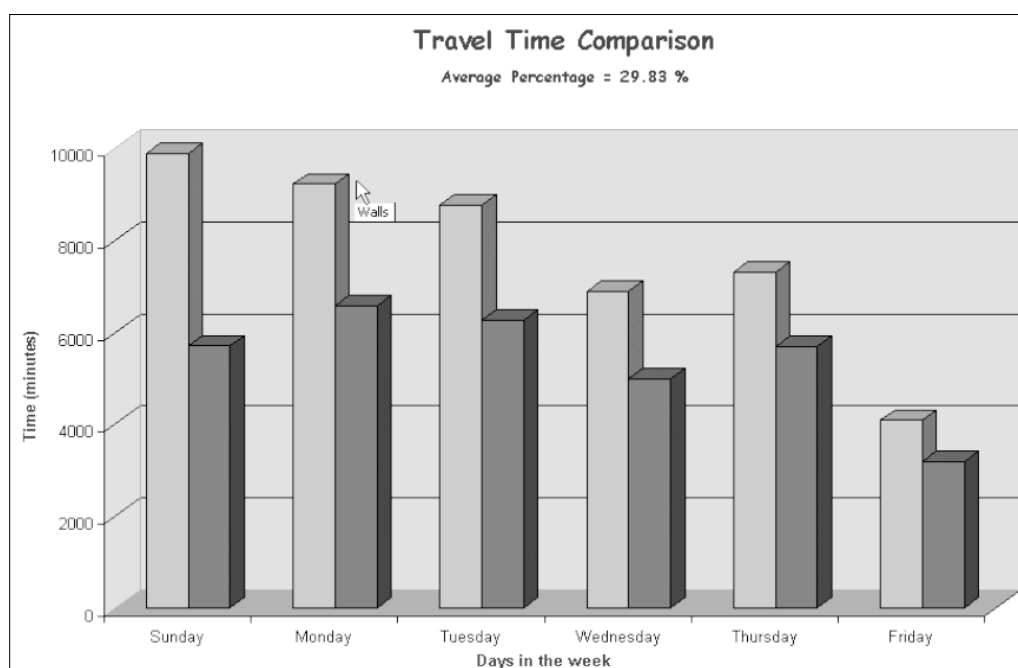


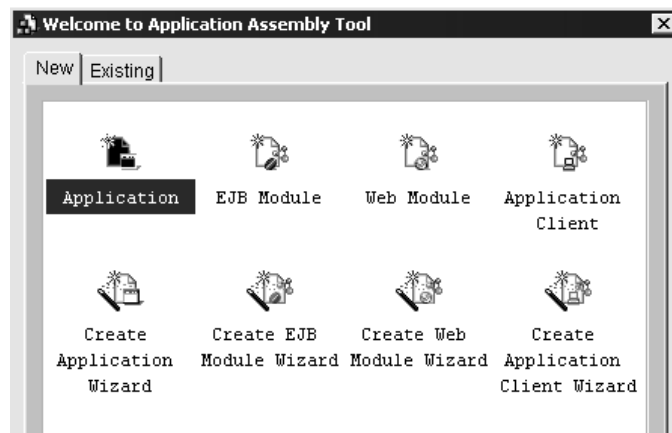
图 38-3 通过使用优化的调度算法，实现行车时间的节约（因此也表现为总工作时间的节约）

38.2 构建 EAR/WAR

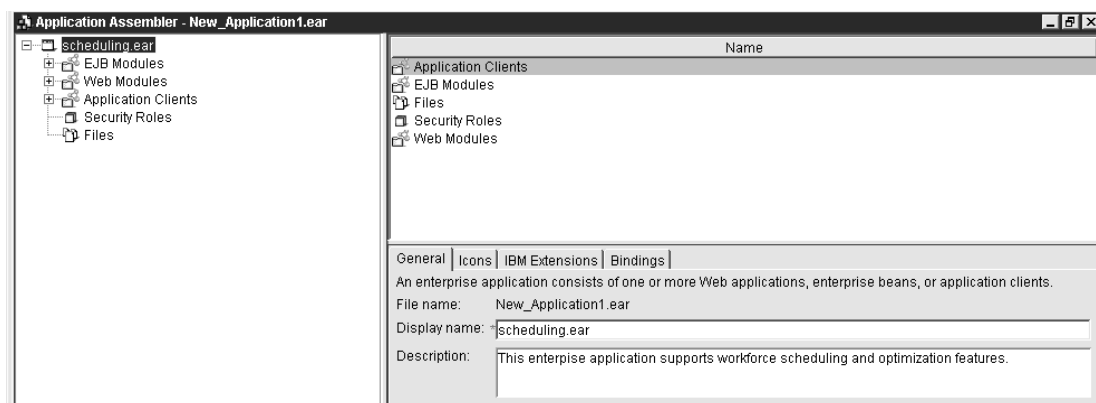
在这里并不讨论这个算法的细节，因为它很复杂，并且具有更高的专利价值。现在假设已

经将这个调度算法打包成一个 JAR (Java Archive) 文件中的 Java 类的集合。为了在 WebSphere 中将这个调度算法作为一个 Web 服务提供出来，首先就需要将它部署为一个应用程序与一个 Web 模块。

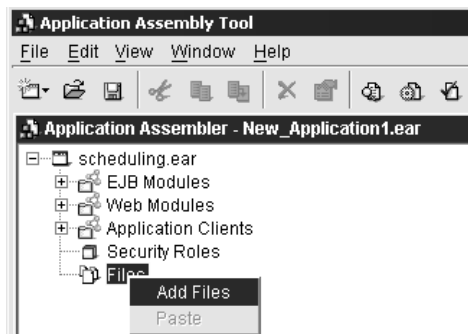
1. 为了打开应用程序装配工具，请选择 Programs | IBM WebSphere | Application Server v4.0 | Application Assembly Tool。然后，在窗口中双击 Application 选项，如下所示：



2. 由于在这里需要创建一个新 EAR (Enterprise Application Archive)，所以应该在 Application Assembly Tool 中填写显示名与描述，如下所示：



3. 在 EAR 中需要添加适当的 Java 类。因此，应该将有关的 Java 文件添加到 EAR 中。右击 Files 子菜单，并且选择 Add Files：



4. 在 Add Files 对话框中，选择添加 scheduling.jar 文件。在选择文件之后，所得到的结果如图 38-4 所示。

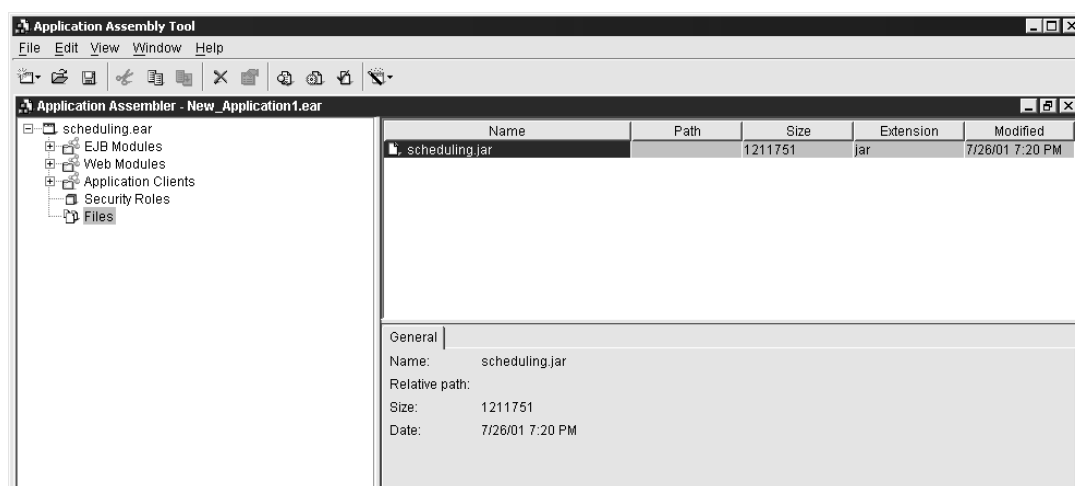


图 38-4 EAR 描述中的 JAR 文件

5. 然后，选择 File | New | Web Module 以创建一个新 Web 模块。这个新 Web 模块包括一系列 servlet 与 JSP。在 Classpath 字段中输入路径名，从而设置这个 Web 模块的 classpath。

Web 模块并不直接与 Web 服务的调用相关，但一个真正的调度包不仅仅需要服务。它还将包括一个界面，用户通过这个界面能够查看与修改任务分配，查看突出的任务以及查看位置。图 38-5 显示了一个关于这个用户界面的例子；它包括一个调度板，从而使得调度者能够查看 Gantt 图表，以了解谁正在做什么并且获得关于哪些任务与资源能够重叠的说明图。

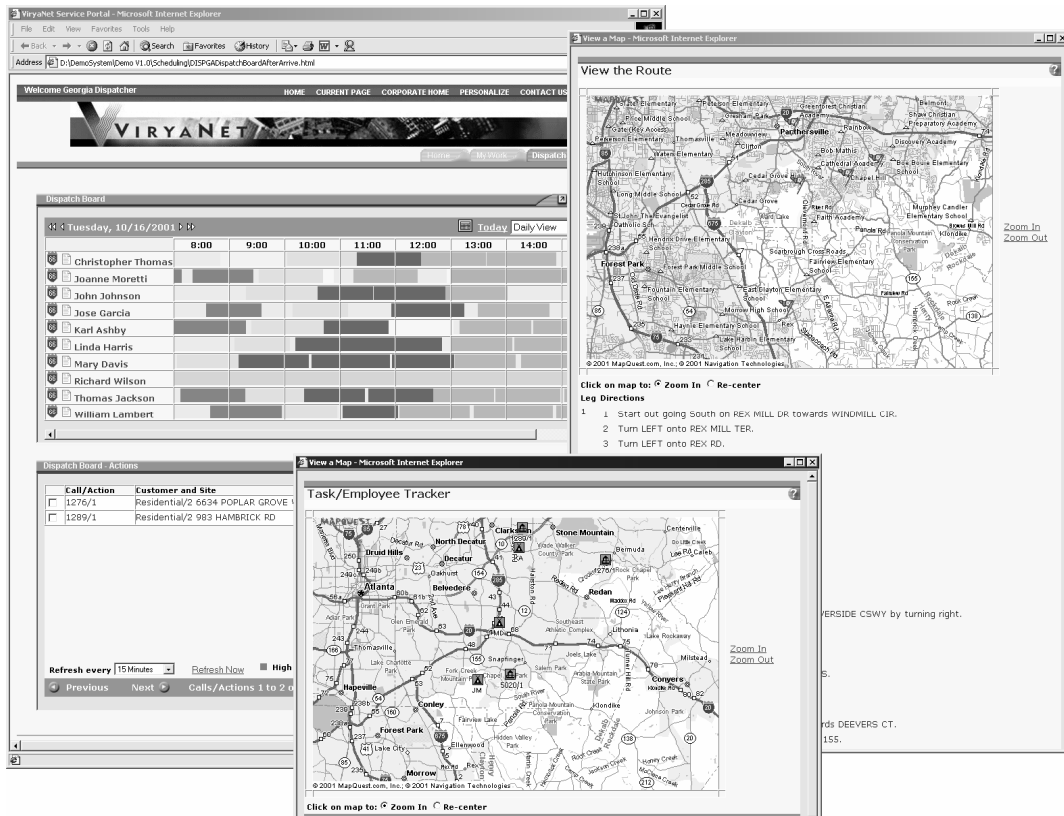
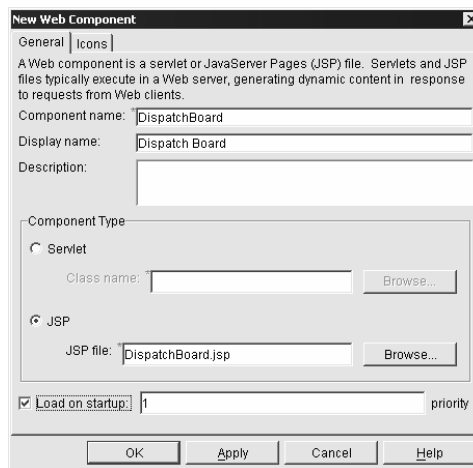
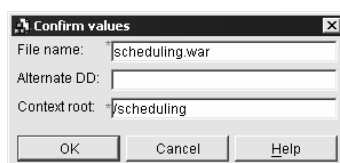


图 38-5 调度板用户界面

6. 然后，就需要定义一系列 JSP 与 servlet，并且将它们添加作为 Web 模块的一部分：



7. 完成这一工作之后，就能够保存新创建的 WAR（Web Application Archive），关闭并返回到 EAR。在进入 EAR 之后，需要右击目录树中的 Web Modules 并选择 Import，从而导入这个 Web 模块。这时，将显示如下所示的 Confirm values 对话框：



8. 所有这些都完成之后，Web 模块就成为企业应用程序的一部分，如图 38-6 所示，并且可以保存这个 EAR。

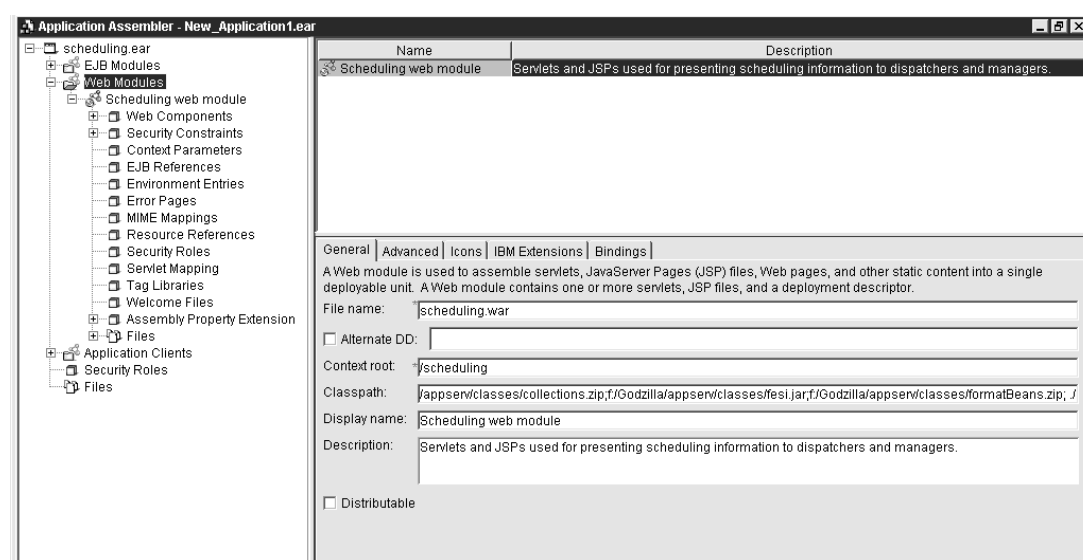


图 38-6 Web 模块作为企业应用程序的一部分

这样，所有的工作都完成了。已经将算法封装到一个 EAR（包括用户界面组件）中，下一步就可以开始创建 SOAP 封装器并将它安装到 WebSphere 服务器上，用作 Web 服务。

38.3 创建 SOAP 部署描述符

为了使服务器能够将应用程序的功能作为 Web 服务提供出来，需要创建 SOAP 层。作为 WebSphere 一部分安装的库与工具能够使这一工作很容易。

在这个例子中，需要提供出来的是一个 Java 类及其方法。因此，需要指定哪个类以及哪个方法提供这种服务。可以使用下列部署描述符完成这项工作：

```

<isd:service
  xmlns:isd="http://xml.apache.org/xml-soap/deployment"
  id="urn:viryaset-advanced-scheduling">
  <isd:provider
    type="java"
    scope="application"
    methods="runAdvancedScheduling">
  <isd:java class="com.rts.scheduling.AdvancedScheduling"/>
  </isd:provider>
  <isd:faultListener>
    org.apache.soap.server.DOMFaultListener
  </isd:faultListener>
</isd:service>

```

部署描述符是 Apache SOAP 包的一部分。部署描述符是一些 XML 文档，用来向 SOAP 运行时环境提供关于客户调用可以使用的服务的信息。其实际的内容取决于提供这个服务的“人工作品”(artifact)。例如，提供这一服务的 EJB 服务器的部署描述符应该在 isd:provider 元素中具有 isd:option 元素。关于部署描述符的简要介绍，请参见 <http://xml.apache.org/soap/docs/guide/deploy.html>。

在准备好部署描述符之后，就需要运行 SoapEarEnabler 工具。这个工具是 WebSphere 发布的一部分，并且它在 WAS_HOME/AppServer/bin 中。为 EAR 运行这个工具是很简单的，其控制台会话如下所示：

```

IBM WebSphere Application Server Release 4
SOAP Enterprise Archive enabler tool.
Copyright IBM Corp.,1997-2001

Please enter the name of your ear file:
f:\Godzilla\appserv\classes\scheduling.ear

***Backing up EAR file to: f:\Godzilla\appserv\classes\scheduling.ear~

How many services would you like your application to contain (1...n)?1

Now prompting for info for service 1:
Please enter the filename of the SOAP deployment descriptor xml file:
f:\Godzilla\appserv\classes\scheduling.xml
Is this service an EJB;(y =yes /n =no)?n
How many jarfiles are required for this service (0...n)?1
Classpath requirement 1:Please choose a file:([1 ]scheduling.war):1
Should this service be secured;(y =yes /n =no)?n

Please enter a context root for your non-secured services (e.g./soap):

```

/scheduling

Do you wish to install the administration client?
Warning,you should not install this client in a production ear unless
you intend to secure the URI to it.

Install the administration client;(y =yes /n =no)?y

38.4 安 装 服 务

现在已经安装了包，下一步就需要将它部署到 WebSphere 上，这是将包部署为 EAR 的标准部署过程。

Administrator Console 安装 EAR 到 Enterprise Applications 文件夹中，如图 38-7 所示。

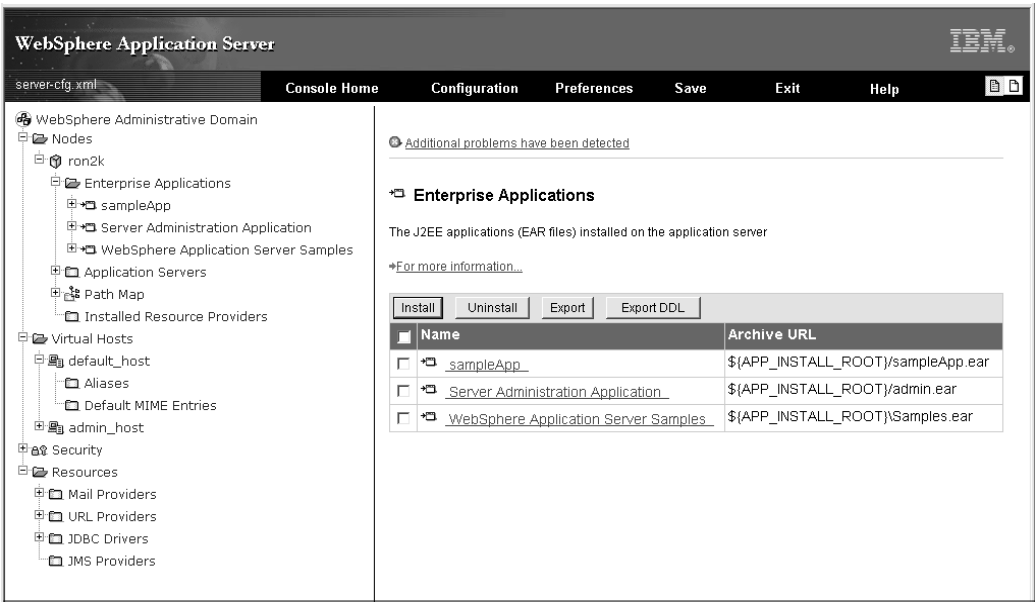


图 38-7 安装在服务器上的企业应用程序

- 1. 单击 Install 按钮以打开 Application Installation Wizard。然后输入 EAR 应用程序文件名：

Application Installation Wizard

Specify the application (EAR file) or standalone module (JAR or WAR file) to install. If installing a module, specify the name of a new application in which to install the module.

Specify the Application or Module located on this machine to upload and install

Path:

Application Name: Used only for standalone modules (*.jar, *.war)

Context Root: Used only for Web modules (*.war)

Specify the Application or Module located on server to install

Remote Path on server:

Application Name: Used only for standalone modules (*.jar, *.war)

Context Root: Used only for Web modules (*.war)

2. 选择主机（在这个例子中，只要让它保持为默认值即可），如下所示：

Application Installation Wizard

Each Web module in your application must be mapped to a virtual host and may have JSPs precompiled. For each Web module below, enter a virtual host name and select precompiled JSP option.

Specifying Virtual Host names and Precompiled JSP option for Web Modules

Web Module Name	Virtual Host Name	Precompile JSPs
Scheduling web module	default_host	Yes

3. 确认应用程序的细节，然后单击 Finish 按钮：

Application Installation Wizard

Confirm the information that you entered for installing the application. The application or module will be installed with the settings you provided. To change the settings, click the Back button. To finish installing the application, click the Finish button.

Confirm the following:

Application Name = Scheduling
Application File Pathname = f:\WebSphere4
\AppServer\temp\ron2k\Default_Server\Server_Administration_Application\admin.war
\scheduling.ear
RoleAssignments:
Run as Mapping for Roles:
EJB Reference Bindings:

4. 现在返回到企业应用程序列表并选择新安装的企业应用程序（在左边面板的树形视图中单击应用程序的名称,或者在右边面板上单击一条超链接）。启动这个应用程序，如图 38-8 所示。

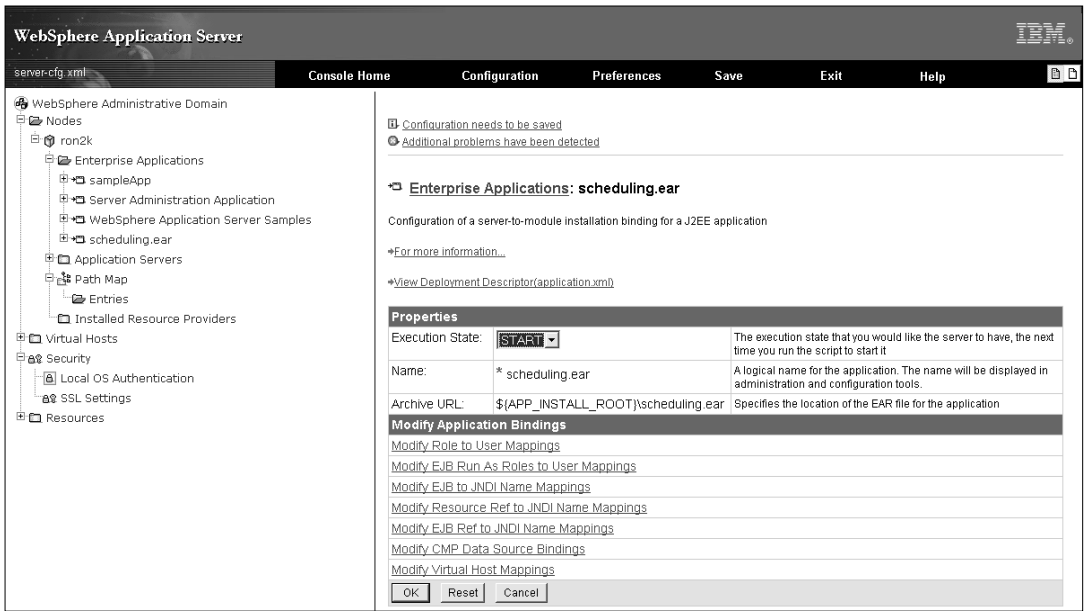


图 38-8 启动用于调度的企业应用程序

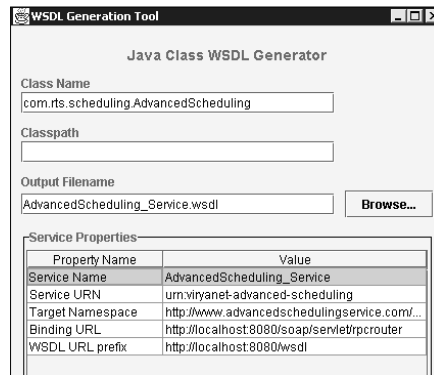
- 5. 最后，单击 Generate 以重新产生 Web 插件配置，从而允许对企业应用程序的访问。这样，部署的工作就完成了（这时的 WebSphere 实例就能够提供使用高级算法的调度服务了）。

38.5 创建 WSDL 文件

虽然已经在服务器上安装了服务，但除非别人知道这个服务在何处并且知道如何调用这个服务，否则就没有人能够真正使用它。为了完全结束部署过程，需要创建这个服务的描述（也就是，必须产生一个 WSDL 文件），然后向 UDDI 注册表发布这个服务以及相关的描述元数据。

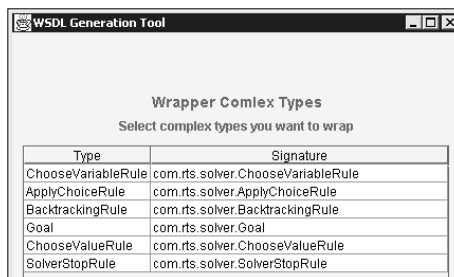
这个例子将使用 Web Services Toolkit 并且利用 wsdlgen 实用程序产生一个 WSDL 文件。需要说明的是，在试图完成这些动作之前，需要正确地设置 classpath，因为这个实用程序为了创建适当的类型信息将从类定义中获取元数据。

在启动这个实用程序之后，将出现 WSDL Generation Tool 的第一个屏幕。在其中填写类名称并且对 URN（Uniform Resource Name，统一资源名称）进行修改，然后填写服务名称，如下所示：

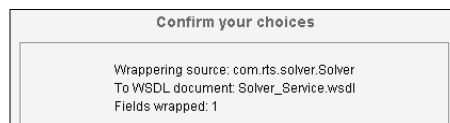


然后，这个实用程序将给出类中可用的方法。它允许对可用的 `public` 方法进行简单的包装，并且允许以 WSDL 文件的形式为这些方法创建元数据。如果方法签名只有简单类型，那么就不需要进行包装。在这种情况下，可以立刻结束工作。

但不幸的是，真实的业务函数并不是仅仅有一个 `int` 和一个 `string` 作为它们的参数。复杂的服务通常需要有复合的数据结构。例如，在这里的调度服务中，服务的输入就包括一个复合的、递归的数据类型。在这种情况下，方法的左边将被标记上一个红色的点。如果选择这个方法，系统将显示这个复合的类型，并要为此复合的类型定义附加的映射：



这时可以选择任何（甚至所有的）复合类型，并为它们产生一个包装器，如下所示：



在这个过程结束时，就在没有编写一行代码的情况下得到了一个完整的 WSDL 文件（这个文件很长）。

38.6 与 UDDI 注册表交互

WebSphere 并没有提供一个私有的 UDDI 注册表。为了支持 UDDI，WebSphere 提供了一个称为 UDDI4J 的 Java 库。这个库允许 Java 程序与 UDDI 注册表交互，用于发布服务以及发现服务。这就意味着能够将简单的 Java 程序放在一起从而在私有注册表或公共注册表上发布服务。

虽然这已经足够了，但人们可能会希望（并相信）WebSphere 的未来版本中将确实包括一个 UDDI 注册表。但目前，可以使用一个公共注册表，它是可以安装与管理的第三方注册表，可以从 <http://www7b.boulder.ibm.com/wsdd/downloads/UDDIregistry.html> 下载这个 IBM 的 UDDI 注册表的预览发布。下面继续讨论第三种选择。

WebSphere UDDI Registry Preview 运行于 WebSphere Application Server 4.0。它支持 UDDI 1.0 规范中定义的 20 个基于 SOAP 的 API，并且通过 IBM DB2 为发布的实体提供持续性。因此，对整个包的下载是很大的（DB2 组件就有大约 300MB），并且安装过程也需要一定的时间。

在这个包中包含一个基于 Web 的图形用户界面，它能够支持业务、服务以及其他与 UDDI 兼容的实体的发布与查询，而不需要编写任何代码。现在开始进行这项工作。

在登录到这个工具中之后，就可以开始发布服务。首先定义一个用于调度服务的新业务实体（见图 38-9）。作为业务实体的一部分，然后就应该定义联系信息、地址信息等。

IBM WebSphere UDDI Registry Preview
Universal Description, Discovery and Integration

UDDI Register
UDDI Login
UDDI Logout
UDDI Publish
UDDI Find

Add a new Business

The fields indicated with an asterisk (*) are required; other fields are optional. If you don't wish to add the business press the **Cancel** button to return to the main menu. When you are satisfied with the information you have entered press the **Continue** button to verify your business information.

Name	Action
Scheduler	Edit

Business Description(s)
Add a new description

Business Contact(s)
Add a new contact

Business Locator(s)
Add a new locator

Cancel Continue Help

图 38-9 添加一个新业务实体

下一步是非常重要的部分：定义服务的端点（endpoint）。它不仅指定了各种方法（这些方法可用来发现关于如何通过服务描述符调用服务的细节），而且定义了服务的访问点（access point；也就是在何处发现这些服务）。这样就需要定义方法，使得能够通过这个方法创建一条到达安装了包的 WebSphere 服务器的请求。通过在图 38-10 所示的 Add a new Service 窗口中设置访问点，就能够定义这个服务端点。按下 Add a new access point（添加一个新访问点）链接，并且填写 URL 与这个访问点的细节。

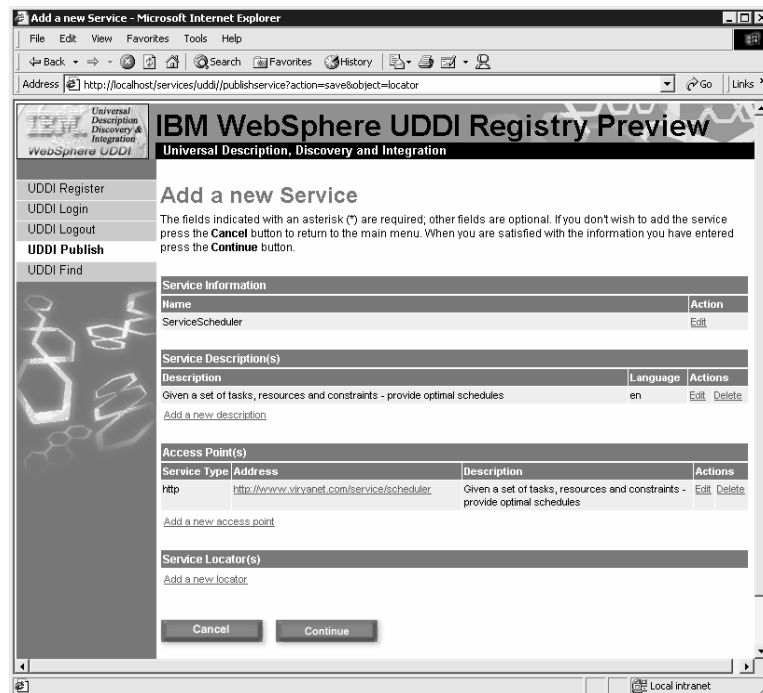


图 38-10 包括访问点的服务细节

38.7 小 结

Web 服务是未来发展的趋势，WebSphere（以及其他主要平台，例如.NET）能够提供对 Web 服务的完全支持。本章讨论了 WebSphere 提供的工具与服务。在这里给出的例子，应该能够帮助理解服务作为业务功能的概念，这对于远程、离散的应用是非常有用的，我们也描述了如何将实现打包成一个企业应用程序、如何通过 Apache SOAP 启用这个服务、如何创建 WSDL 并且将它发布到一个 UDDI 注册表中。

本章是第 7 部分的最后一章，这一章致力于描述 WebSphere 中可用的 XML 相关技术。本书的第 8 部分将继续讨论另一个重要功能：国际化与本地化应用程序的支持。

第 8 部分 国际化与本地化

第 39 章 Java 支持的国际化功能

本章讨论 Java 提供的用于构建国际化应用程序的支持。它描述了 Java 所固有的国际化功能，例如，对 Unicode 标准的支持、对资源束的使用以及许多其他类似的机制。在这里还给出了一些例子，用来说明如何在 WebSphere 中编写 Java 应用程序，才能更好地支持本地化。因此，本章讨论的重点是一个以机制为中心的话题，也就是，本章将向读者介绍 Java 中为国际化与本地化提供的大量支持。

本章以及下一章的所有例子都来自 ViryaNet 的 Service Hub，这一产品在为日本市场进行本地化之后，在日本的销售很成功。

39.1 Unicode

首先开始介绍 Unicode，它是 IT 领域几乎所有厂商都认可的标准。这个标准由一个称为 Unicode Consortium 的非赢利组织创建，其目标是为所有语言中的所有字符创建一个统一的编码方法。Unicode Consortium 作为一个标准委员会，是最成功的组织之一；没有任何其他编码标准能够如此受到重视。图 39-1 显示了 Unicode Consortium 的主页，其地址为 <http://www.unicode.org>。可以注意到，在左边面板中给出了一系列多语言文本。

Java 基于 Unicode。实际上，在 Java（以及所有从 Java 继承而来的技术平台，例如 WebSphere）中的字符表示使用 Unicode（也就是，JVM 中字符的本地表示使用 2 个字节）。这个前提是 Java（以及任何 WebSphere 应用程序）在本质上能够支持国际化与本地化应用程序这一事实的基础。关于 Unicode 的详细讨论不是本书的范围。关于 Unicode 这个概念的更详细描述，请访问 <http://www.unicode.org/unicode/standard/principles.html>。

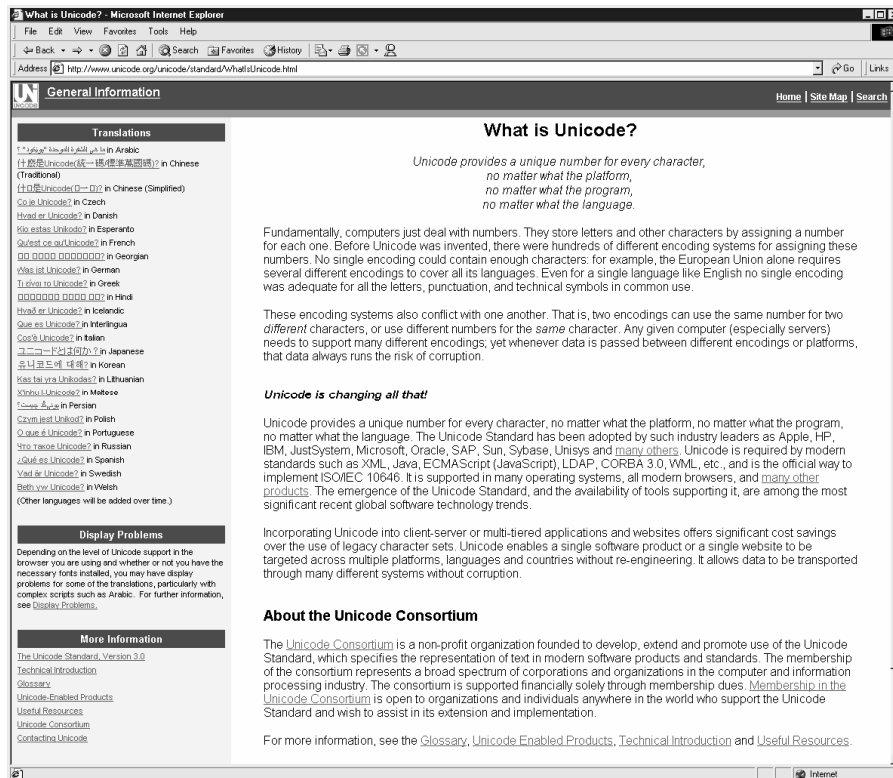


图 39-1 Unicode Consortium 的主页

39.2 资源 束

每个应用程序都需要使用字符串，这些字符串通常为两种类型之一（具体取决于应用程序如何使用这些字符串）。其中一种就是作为应用程序所管理的数据的一部分的字符串。例子包括产品目录中的产品描述以及顾客主数据库中的公司名。另一种就是应用程序在提示、错误消息、提供信息的信息以及界面中的其他消息中所使用的字符串。

第一种类型的字符串，除了能够在不同的持续性结构中存储各种描述（例如产品的描述）以及基于用户的语言使用正确的描述之外，不需要应用程序的干预（例如，如果正在使用关系数据库存储数据，那么一种方便的方法就是保存一些列，其中这个应用程序所支持的每种语言使用一列作为该语言下的产品描述。并且使应用程序基于当前活跃的语言提取（插入/更新）适当的列。下一章将继续讨论这种解决方案）。

第二种类型的字符串包括应用程序所使用的消息、提示、标注与常量字符串。在这种情况下，需要支持的字符串集合是有限的，并且由应用程序完全控制。它不涉及数据表项的问

题，也不涉及在数据存储器中维护这些字符串的问题。为这些字符串提供语言级支持是比较简单的。

Java 通过资源束 (resource bundle) 提供对国际化与语言级的完全支持。正如其名称所隐含的那样，资源束维护了一个关于资源的束——每个区域对应一个资源（后面将进一步讨论区域，现在把区域理解成一种语言）。资源束可用来为每种语言（每个区域）维护不同的字符串与常量集合。在给定了将要为其服务的区域（例如，将要使用某种语言为其创建显示的用户）之后，就可以非常方便地访问与这个区域相关的所有资源。使用资源束，能够提供另一种级别的间接 (indirection)，这就意味着不需要对所使用的字符串、消息与其他常量进行硬编码；相反，可以访问与当前区域相匹配的束并且从束中获取所有资源。

示例：错误消息

现在来分析一个例子。下面给出的类提供了 API Repository 以及用于处理数据库连接的全部主要的 API。使用通用的 `com.rts.apis.core.Log` 类对消息进行日志。其中大多数构造程序都是多余的，因为大多数方法都是静态方法。`main()` 方法在这里用于测试的目的。这个类使用 2 个属性文件：`APIRepository.properties` 与 `APIRepositoryMessages.properties`。

```
package com.rts.apis.common;

import java.sql.*;
import java.util.*;
import java.io.*;
import java.text.*;
import com.rts.apis.common.Log;

public class APIRepository {
    private static Locale currentLocale = null;
    private static ResourceBundle apiRepositoryMessages = null;
    /** Class variable that maintains the connection to the database.
     * It is accessed via the public class method
     * getRepositoryConnection.
     * getRepositoryConnection.*/
    private static Connection con = null;

    /** Hashtable used for caching components when querying
     * during the API Server processing.
     * There is no caching when using the API Repository definition tool
     */
    public static Hashtable componentHash = new Hashtable(100);
```

```
/** Hashtable used for caching services when querying during
 * the API Server processing.
 * There is no caching when using the API Repository definition tool
 */
public static Hashtable serviceHash = new Hashtable(100);
/** Hashtable used for caching component bindings when querying
 * during the API Server processing.
 * There is no caching when using the API Repository definition tool
 */
public static Hashtable componentBindingHash = new Hashtable(100);
/** Hashtable used for caching argument bindings when querying
 * during the API Server processing.
 * There is no caching when using the API Repository definition tool
 */
public static Hashtable argumentBindingHash = new Hashtable(100);

private static ResourceBundle apiRepositoryProps = null;

    static {
        if (apiRepositoryProps == null) {
            initProps();
        }
    }

/**
 * Closes the repository connection
 * Exceptions must be handled by caller of this method
 *
 * @exception SQLException An sql exception if one occurs
 */
public static void closeRepositoryConnection() throws SQLException {
    //Try to disconnect.
    if (con != null)
        con.close( );
    con = null;
}

/**
 * Get API Repository Messages Bundle for all locale sensitive messages
 * @return ResourceBundle API Repository messages
 * @exception FrontEndException If the message file is missing
 */
public static ResourceBundle getApiRepositoryMessages()
    throws FrontEndException {
    try {
```

```

        if (apiRepositoryMessages == null) {
            apiRepositoryMessages = ResourceBundle.getBundle(
                "APIRepositoryMessages", getLocale());
        }
    }
    catch (MissingResourceException ex) {
        //do not use the resource bundle for this message
        // because clearly it will not be found
        Log.debug("APIRepository", ex);
        throw new FrontEndException(
            "Missing Resource Exception: " + ex.getMessage());
    }
    return apiRepositoryMessages;
}

/**
 * Get API Repository Properties
 * @return Properties API Repository properties
 */
public static ResourceBundle getApiRepositoryProps() {
    if (apiRepositoryProps == null) {
        initProps();
    }
    return apiRepositoryProps;
}

/**
 * Get locale based on language, country properties in the
 * APIRepository properties file
 * @return Locale current locale
 * @exception FrontEndException If an error occurs
 */
public static Locale getLocale() throws FrontEndException {
    if (currentLocale == null) {
        String lang = apiRepositoryProps.getString("language");
        String cntry = apiRepositoryProps.getString("country");
        currentLocale = new Locale(lang, cntry);
    }
    return currentLocale;
}

/**
 * Gets the next version string based on the input argument last version
 * examples: argument: 1.00, returns 1.01
 *           argument: 1.0a, return 1.0b
 *           argument: 1, returns 2
 * @param String last version

```

```
* @return String next version
*/
public static String getNextVersion(String lv) {
    //Find last period and determine if after it is numeric only...
    // if so add 1, if not increment character
    String strValue = "";
    StringTokenizer st = new StringTokenizer(lv, ".");
    while (st.hasMoreTokens()) {
        strValue = st.nextToken();
    }
    if (strValue.equals(""))
        return "1.00";
    boolean allDigits = true;
    char [] c = new char[strValue.length()];
    strValue.getChars(0, strValue.length(), c, 0);
    for (int i=0; i < c.length; i++) {
        allDigits = Character.isDigit(c[i]);
    }
    if (allDigits) {
        String s = lv.substring(0, lv.length() - strValue.length()) +
            ((new Integer(strValue)).intValue() + 1);
        return s;
    }
    else {
        char lastChar = c[c.length-1];
        int cv = Character.getNumericValue(lastChar) + 1;
        String newChar = String.valueOf(
            Character.forDigit(cv, Character.MAX_RADIX));
        return lv.substring(0, lv.length() - 1) + newChar;
    }
}

public static Connection getPortalConnection()
    throws SQLException, FrontEndException {
    Connection con = null;
    if (con == null)
    {
        String driverClassName =
            apiRepositoryProps.getString("dbDriver");
        String databaseURL =
            apiRepositoryProps.getString("dbDriverType") +
            apiRepositoryProps.getString("dbURL");
        Properties properties = new Properties( );
        properties.put("user",
            apiRepositoryProps.getString("portal_name"));
    }
}
```

```

        properties.put("password",
            apiRepositoryProps.getString("portal_pass"));
    try {
        String ns = apiRepositoryProps.getString(
            "driverNeedsServer");
        properties.put(
            "server",
            apiRepositoryProps.getString("portal_env"));
    } catch (MissingResourceException ex) {}
    try {
        Class c = Class.forName(driverClassName);
        DriverManager.registerDriver((Driver)c.newInstance());
        con = DriverManager.getConnection(
            databaseURL, properties);
    } catch (ClassNotFoundException ex) {
        System.err.print(
getApiRepositoryMessages().getString("ClassNotFoundException");
        System.err.println(ex.getMessage());
        throw new FrontEndException(
getApiRepositoryMessages().getString(
            "ClassNotFoundException")+ ex.getMessage();
    } catch (IllegalAccessException ex) {

System.err.print(getApiRepositoryMessages().getString("IllAccEx");
    System.err.println(ex.getMessage());
    throw new FrontEndException(
getApiRepositoryMessages().getString("IllAccEx")
        + ex.getMessage());
    } catch (InstantiationException ex) {
    System.err.print(
getApiRepositoryMessages().getString("InstEx");
    System.err.println(ex.getMessage());
    throw new FrontEndException(
getApiRepositoryMessages().getString("InstEx")+
        ex.getMessage());
    }
}
return con;
}
/**
 * Get the repository connection allows a single place
 * for establishing a DB connection
 * Exceptions must be handled by those that call this method
 * The username/password/environment are taken from the properties file

```



```

* The exception messages need to be internationalized...
*
* @exception SQLException An sql exception if one occurs
* @exception FrontEndException If another error occurs in
* methods called from here
* @return Returns a Connection instance*/
public static synchronized Connection getRepositoryConnection()
    throws SQLException, FrontEndException {
    if (con == null)
    {
        //Supply the information specific to the local system.
        //Enter your JDBC driver name and URL.
        // Using Weblogic
        String driverClassName =
            apiRepositoryProps.getString("dbDriver");
        String databaseURL =
            apiRepositoryProps.getString("dbDriverType") +
            apiRepositoryProps.getString("dbURL");
        Properties properties = new Properties( );
        //Enter your user name and password.
        //Additional properties may be required.

        properties.put("user",
            apiRepositoryProps.getString("dbUser"));
        properties.put("password",
            apiRepositoryProps.getString("dbPassword"));
        //Some drivers need the server property and others don't
        //so have property to say whether needs,
        //its value is irrelevant. If the driver does not need
        //the server the property is commented out
        try {
            String ns = apiRepositoryProps.getString("driverNeedsServer");
            properties.put("server",
                apiRepositoryProps.getString("dbEnvironment"));
        } catch (MissingResourceException ex) {}

        try {
            Class c = Class.forName(driverClassName);
            DriverManager.registerDriver((Driver)c.newInstance());
            //Try to connect to the database through the driver.
            con = DriverManager.getConnection(databaseURL, properties);
        } catch (ClassNotFoundException ex) {
            Log.debug("APIRepository",
                getApiRepositoryMessages().getString("ClassNotFoundExcepti"),2);

```

```

        Log.debug("APIRepository", ex.getMessage(), 2);
        throw new FrontEndException(
            getApiRepositoryMessages().getString("ClassNotFoundExcepti")+
            ex.getMessage());
    } catch (IllegalAccessException ex) {
        Log.debug("APIRepository",
            getApiRepositoryMessages().getString("IllAccEx"), 2);
        Log.debug("APIRepository", ex.getMessage(), 2);
        throw new FrontEndException(
            getApiRepositoryMessages().getString("IllAccEx")+
            ex.getMessage());
    } catch (InstantiationException ex) {
        Log.debug("APIRepository",
            getApiRepositoryMessages().getString("InstEx"), 2);
        Log.debug("APIRepository", ex.getMessage(), 2);
        throw new FrontEndException(
            getApiRepositoryMessages().getString("InstEx")+
            ex.getMessage());
    }
}
try {
    con.setAutoCommit(true);
} catch (SQLException e) {
    try {
        String driverClassName =
            apiRepositoryProps.getString("dbDriver");
        String databaseURL =
            apiRepositoryProps.getString("dbURL");
        Properties properties = new Properties( );
        properties.put("user",
            apiRepositoryProps.getString("dbUser"));
        properties.put("password",
            apiRepositoryProps.getString("dbPassword"));
        try {
            String ns =
                apiRepositoryProps.getString("driverNeedsServer");
            properties.put("server",
                apiRepositoryProps.getString("dbEnvironment"));
        } catch (MissingResourceException ex) {}
        Class c = Class.forName(driverClassName);
        DriverManager.registerDriver((Driver)c.newInstance());
        //Try to connect to the database through the driver.
        con = DriverManager.getConnection(
            databaseURL, properties);
    }
}

```

```
        con.setAutoCommit(true);
    } catch (Exception e1) {
        con = null;
    }
}
return con;
}

private static void initProps () {
    apiRepositoryProps = ResourceBundle.getBundle(
        "ApisInits", Locale.getDefault());
}
/**
 * main Method is used for testing purposes only
 */
public static void main(String[] args) {

}

/**
 * Class method to return an argument binding from the repository
 * based on the name, version
 *
 * @exception FrontEndException propagates exceptions to caller
 * @param String service name
 * @param String service version
 * @param boolean cachingEnabled
 * @return An instance of ArgumentBinding or null if none.*/
public static ArgumentBinding queryArgumentBinding(
    String name, String version, boolean cachingEnabled)
    throws FrontEndException{
    return ArgumentBinding.query(name, version, cachingEnabled);
}

/**
 * Class method to return a component from the repository based on the
 * name, version
 *
 * @exception FrontEndException propagates exceptions to caller
 * @param String service name
 * @param String service version
 * @param boolean cachingEnabled
 * @return An instance of Component or null if none.*/
public static Component queryComponent(
    String name, String version, boolean cachingEnabled)
```

```

        throws FrontEndException{
            return Component.query(name, version, cachingEnabled);
        }
    /**
     * Class method to return a component binding from the repository
     * based on the name, version
     *
     * @exception FrontEndException propagates exceptions to caller
     * @param String service name
     * @param String service version
     * @param boolean cachingEnabled
     * @return An instance of ComponentBinding or null if none.*/
    public static ComponentBinding queryComponentBinding(
        String name, String version, boolean cachingEnabled)
        throws FrontEndException{
        return ComponentBinding.query(name, version, cachingEnabled);
    }
    /**
     * Class method to return a service from the repository based on the
     * name, version and provider. If the version is an empty string
     * then it will bring the latest version of the service found.
     * Caching will be used based on the cachingEnabled argument
     *
     * @exception FrontEndException propagates exceptions to caller
     * @param String service name
     * @param String service version
     * @param String service provider
     * @param boolean cachingEnabled
     * @return An instance of Service or null if none.*/
    public static RepositoryService queryService(
        String name, String version,
        String provider, boolean cachingEnabled)
        throws FrontEndException{
        return RepositoryService.query(
            name, version, provider, cachingEnabled);
    }
}

```

这个例子演示了静态错误消息的使用。静态对象由一个称为 `apiRepositoryMessages` 的类维护。当程序需要获取一条错误消息时，它并不包括常量字符串，而是从资源束对象中获取一个字符串。因此，使用资源束的 `getString` 调用的所有字符串仅仅是查找过程中所用的关键字，而不是字符串本身。

这样就允许将实际的消息字符串输出到一个文件中，如下所示：

```
#Sun Mar 21 13:28:53 GMT 1999
ErrorCode=Error Code:
ClassNotFoundException=Class Not Found Exception:
IllegalAccessException=Illegal Access Exception:
InstantiationException=Instantiation Exception:
DBException=DB Exception: {0}
Message=Message:
Transaction_rollback=Transaction is being rolled back...
SQLException=-----SQLException-----
Connecting_to_the=Connecting to the database...
Service_exists=Validation Error: Service already exists for this version.
Change the version number to save this Service.
Try_to_connect_to=Trying to connect to {0}/{1}@{2}
Not_found=Not found.
This_is_the_info=This is the information retrieved.
Structures_must_be_defined=Validation Error: Input, Output, Error and
Exception Structures must be defined.
Exception_caught_due=Exception caught due to: {0}
DOM_Creation=DOM Creation Exception: {0}
SQLState=SQLState:
CB_exists=Validation Error: Component Binding already exists for this
version. Change the version number to save this Component Binding.
input_invalid=Validation Error: The input component must be a valid
component.
output_invalid=Validation Error: The output component must be a valid
component
CB_invalid=Validation Error: The component binding must be valid.
AB_exists=Validation Error: Argument Binding already exists for this
version. Change the version number to save this Argument Binding.
prim_prop=Invalid Request: The property is a primitive type.
version_not_allowed=Validation Error: This is a primitive type. It cannot
have a version number.
invalid_version=Validation Error: This is a non-primitive type. Please give
it a valid version number.
invalid_comp=Validation Error: The structure properties must be valid
components.
cond_not_allowed=Validation Error: Output and Error Structures must have
conditions. Please add one.
missing_cond=Validation Error: Input and Exception Structures cannot have
conditions.
NumberFormatException=Invalid number format for version:
type_required=Validation Error: Please enter a property type.
Comp_exists=Validation Error: A component already exists for this version.
Please change the version number to save this Component.
```

```
Only_one_default_cond=Validation Error: Only one default condition is
allowed on output/error structures.
Req_default_cond=Validation Error: One default condition is required on
output/error structures.
Name_prov_required=Validation Error: Please enter a service name and
provider.
Name_required=Validation Error: Please enter a property name.
duplicate_name=Validation Error: You cannot duplicate a property name.
default_only=Validation Error: Default conditions cannot have a reference
or match.
reference_required=Validation Error: Please enter a reference for this
condition.
match_required=Validation Error: Please enter a match for this condition.
stp_name_req=Validation Error: Please enter the stored procedure name.
db_req=Validation Error: Please enter either a database class/method or a
database driver/URL/name/password.
Invalid_DB_URL=Error: You have entered an invalid URL format to extract
database information.
Target_required=Validation Error: Please enter the target.
Source_required=Validation Error: Please enter the source.
bind_not_matching=Validation Error: The target and source should be of the
same type/version, or a component binding should be provided.
Both_must_be_array=Validation Error: The source and target must both be
arrays.
static_info_req=Validation Error: Please enter a static class name and
method for this static mechanism.
corba_info_req=Validation Error: Please enter all the mandatory information
for this corba mechanism.
corba_array_error=Validation Error: A corba service property can't be array
```

上面显示的文件提供了美国英语表示的错误消息。需要说明的是，这个文件以 `en_US` 作为文件名末端的扩展，表示美国英语（`en_UK` 表示英国英语）。例如，如果希望为日语本地化这个应用程序，那么需要将等号（=）右边的字符串进行翻译，并使用 `APIRepositoryMessages_jp_JP` 文件名，如图 39-2 所示。

通过使用不同区域获得对应的资源束，就能够在甚至不修改任何应用程序代码的情况下获取适当的字符串；这就是国际化应用程序的实质！

在资源束中包含有区域特定的对象。当程序需要区域特定的资源（例如 `String`）时，程序就能够从适合于当前用户区域的资源束中加载这种资源。这样，就能够编写完全独立于用户区域的程序代码，在资源束中隔离大部分（如果不是全部）特定区域信息。这就使得能够编写具有下列特点的程序：

- 很容易被本地化（或翻译）到不同的语言

- 能够同时处理多个区域
- 后来很容易被修改以支持更多的区域



图 39-2 Japanese 区域对应的属性文件示例

资源束在概念上就是一系列从 ResourceBundle 继承而来的相关类。ResourceBundle 的每个相关子类都有相同的基名，再加上一个用来标识其区域的附加组件。例如，假设资源束命

名为 `MyResources`。那么，编写的第一个类应该是默认的资源束，它具有与其家族相同的名称 (`MyResources`)。然后，也可以提供任意数量的特定区域类：例如，可以为德语提供一个名为 `MyResources_de` 的类。

`ResourceBundle` 的每个相关子类都包含相同的项，但这个项已经被翻译成 `ResourceBundle` 子类所代表的那个区域对应的语言。例如，`MyResources` 与 `MyResources_de` 都有一个 `String`，用来标注一个表示取消操作的按钮。在 `MyResources` 中，这个 `String` 可能包含的是 `Cancel`，而在 `MyResources_de` 中包含的则是 `Abbrechen`。

如果有不同的资源应用于不同的国家，那么也可以进行一些特殊化处理：例如，`MyResources_de_CH` 代表瑞士 (CH) 的德语 (de)。如果希望仅仅修改这个特殊化资源束中的某些资源，也可以这样做。

当程序需要一个区域特定对象时，将使用 `getBundle` 方法加载 `ResourceBundle` 类：

```
ResourceBundle myResources =
    ResourceBundle.getBundle("MyResources", currentLocale);
```

其中第一个参数指定了包含所需对象的资源束的家族名。第二个参数指定所需的区域。`getBundle` 使用这两个参数来构造 `ResourceBundle` 子类的名称，并且在后来将加载这个类。资源束查找机制将基于所需的区域、由 `Locale.getDefault()` 返回的当前默认区域以及资源束根 (baseclass)，按照从低级 (更特殊) 到父级 (较少特殊化) 的顺序搜索带有各种后缀的类：

```
baseclass + "_" + language1 + "_" + country1 + "_" + variant1
baseclass + "_" + language1 + "_" + country1 + "_" + variant1 + ".properties"
baseclass + "_" + language1 + "_" + country1
baseclass + "_" + language1 + "_" + country1 + ".properties"
baseclass + "_" + language1
baseclass + "_" + language1 + ".properties"
baseclass + "_" + language2 + "_" + country2 + "_" + variant2
baseclass + "_" + language2 + "_" + country2 + "_" + variant2 + ".properties"
baseclass + "_" + language2 + "_" + country2
baseclass + "_" + language2 + "_" + country2 + ".properties"
baseclass + "_" + language2
baseclass + "_" + language2 + ".properties"
baseclass
baseclass + ".properties"
```

例如，如果当前默认的区域是 `en_US`，而调用者感兴趣的区域是 `fr_CH`，资源束名称为 `MyResources`，那么资源束查找机制将按照下列顺序搜索以下类：

```
MyResources_fr_CH
MyResources_fr
MyResources_en_US
```



```
MyResources_en
MyResources
```

资源束查找结果是一个类，但这个类可能在硬盘上存在一个支持它的属性文件。也就是说，如果 `getBundle` 并没有找到给定名称的类，那么它将附加 `.properties` 到这个类名上，搜索具有这个名称的属性文件。如果它找到了这个文件，那么将创建一个新 `PropertyResourceBundle` 对象来保存这个文件。继续分析前面的那个例子，`getBundle` 将返回类与文件，并且其优先顺序如下：

```
(class) MyResources_fr_CH (file) MyResources_fr_CH.properties (class) MyResources_fr (file) MyResources_fr.properties (class) MyResources_en_US (file) MyResources_en_US.properties (class) MyResources_en (file) MyResources_en.properties (class) MyResources (file) MyResources.properties
```

如果查找失败，`getBundle()` 将发出一个 `MissingResourceException` 异常。

基类必须完全可用（例如，`myPackage.MyResources`，而不仅仅是 `MyResources`）。而且，它必须能够被用户代码所访问；它不能是 `ResourceBundle.getBundle` 调用的那个包所私有的。

注意：`ResourceBundles` 在内部被用来访问 `NumberFormats`, `Collations` 等等。这个查找策略也是如此。

资源束中包含关键字/值对。其中，关键字用来惟一标识资源束中区域特定的对象。下面是一个包含有两个关键字/值对的 `ListResourceBundle` 例子：

```
class MyResource extends ListResourceBundle {
    public Object[][] getContents() {
        return contents;
    }
    static final Object[][] contents = {
        // LOCALIZE THIS
        {"OkKey", "OK"},
        {"CancelKey", "Cancel"},
        // END OF MATERIAL TO LOCALIZE
    };
}
```

关键字总是 `Strings`。在这个例子中，关键字是 `OkKey` 与 `CancelKey`。在上一个例子中，它的值也是 `Strings` (`OK` 与 `Cancel`)，但这不是一成不变的。值可以是任何对象类型。

使用适当的 `get` 方法从资源束中获取一个对象。由于 `OkKey` 和 `CancelKey` 都是字符串，因此需要使用 `getString` 来获取它们：

```
button1 = new Button(myResourceBundle.getString("OkKey"));
button2 = new Button(myResourceBundle.getString("CancelKey"));
```

`get` 方法都需要使用关键字作为参数，并且如果找到了对象，则返回该对象。如果没有找到

这个对象，那么 `get` 方法将发出一个 `MissingResourceException` 异常。

除了 `getString` 之外，`ResourceBundle` 还支持许多其他方法用来获取不同类型的对象，例如 `getStringArray`。如果没有对象与其中的某个方法相匹配，那么可以使用 `getObject` 方法，并且将结果转换为适当的类型。例如：

```
int [] myIntegers = (int []) myResources.getObject ("intList");
```

注意：应该总是给出一个没有带任何后缀的基类。如果所请求的区域不存在，这个基类将作为“最后的选择”。实际上，必须对为其提供资源的任何给定的继承链提供所有的类。例如，如果提供 `MyResources_fr_BE`，那么就必须提供 `MyResources` 与 `MyResources_fr`，否则资源束查找机制就不能正确工作。

Java 2 平台提供了 `ResourceBundle` 的两个子类（分别是 `ListResourceBundle` 与 `PropertyResourceBundle`），它们提供了一个比较简单的方法用于创建资源。正如在前一个例子中见到的那样，`ListResourceBundle` 将资源管理为一个关键字/值对列表。`PropertyResourceBundle` 使用一个属性文件来管理资源。

如果 `ListResourceBundle` 或 `PropertyResourceBundle` 不能满足应用需要，也可以自己编写 `ResourceBundle` 子类。在所编写的子类中必须重载两个方法：`handleGetObject()` 与 `getKeys()`。下面给出了一个关于 `ResourceBundle` 子类的简单例子，就是 `MyResources`，用来管理两个资源（对于更多数目的资源，可能需要使用散列表）。应该注意到，如果没有找到这个关键字，`handleGetObject` 必须返回 `null`。如果关键字是 `null`，那么应该发出 `NullPointerException` 异常。另外，如果父级 `ResourceBundle` 处理相同的关键字并且使用相同的值（正如下面 `German` 对应的那样），那么就不需要提供值。还可以注意到，由于指定了一个 `g_GE` 资源束，因此也必须提供一个默认的 `en` 资源束（虽然这个资源束从根资源束继承所有的数据）。下面就是一个例子：

```
// default (English language, United States)
abstract class MyResources extends ResourceBundle {

    public Object handleGetObject(String key) {

        if (key.equals("okKey")) return "Ok";
        if (key.equals("cancelKey")) return "Cancel";
        return null;
    }
}

// German language
public class MyResources_de extends MyResources {
    public Object handleGetObject(String key) {
        // don't need okKey, since parent level handles it.
    }
}
```

```
        if (key.equals("cancelKey")) return "Abbrechen";  
        return null;  
    }  
}
```

不必在应用程序中限制为只使用一个 ResourceBundles 家族。例如，可以建立一个关于异常消息 ExceptionResources 的资源束 (ExceptionResources_fr, ExceptionResources_de 等等)，也可以建立 WidgetResource 的资源束 (WidgetResources_fr, WidgetResources_de 等等)，实际上，可以对资源进行任意形式的划分。

当然，还有许多有关国际化应用程序的问题需要解决 (例如日期格式、地址格式、字符串连接、强校对以及排序)。幸运的是，Java 提供了内在的机制来处理所有这些问题。关于 Java 中国际化问题的讨论本身就可以占据一本书的篇幅，因此，这里不讨论所有这些问题。但读者在学完本章与下一章之后，就能够编写一个多语言应用程序。

39.3 小 结

本章讨论了 Java 为构建国际化应用程序提供的支持。特别地，在这里给出了 Java 内在的某些国际化功能，例如对 Unicode 标准的支持、使用资源束以及许多其他类似的机制。但仅仅有机制还是不够的。所有这些机制都必须集成在一起使用，从而提供一个一致的框架，使得在这种框架下能够构造多语言应用程序。这是第 40 章讨论的话题，在那一章将使用本章讨论的许多机制来描述一个完全国际化的应用程序。

第 40 章 构建多语言应用程序

第 39 章描述了用来构建国际化应用程序的一些元素。Java 是为了国际化使用而构建的（这也是它在 Internet 上取得成功的原因之一），这一事实也是 Java 可用于非常简易地构建多语言应用程序的基础。但仅仅这样还是不够的，本章将提供一些关于在 WebSphere 中构建多语言应用程序的方法，同时给出相关的源代码示例。

多语言问题虽然很重要，但它也仅仅是 WebSphere 完全国际化与本地化工作中的一个子集。其他元素还包括数据格式、货币格式与地址格式，在某些语言中，甚至还包括更高级的元素（例如 widget，允许从右向左进行书写，或允许从上到下进行书写）。

本章主要关注 WebSphere 的多语言方面，这是因为数据/地址/货币/数值格式等问题都相对比较简单，并且通常可以按照支持多语言类似的方式进行处理（关于 widget 的讨论不是本书的范围，因为这种问题通常不是由应用程序开发人员来解决的）。

40.1 多语言问题的分类

在解析多语言应用程序时，支持多种语言的概念可以划分为 3 大类：

- 屏幕表示
- 数据
- 消息

用户所访问的屏幕（在浏览器、客户-服务器应用程序或无线设备上）全部都包括格式信息，例如提示、标题与标注。当传递以不同语言出现的屏幕时，屏幕的布局（或者说它的表示特点）需要进行一定的调整，从而使得发送给用户的提示与其他消息能够适合该语言对应的大小（例如，德语中的文本通常比英语中的相同文本更长）。

在多语言应用程序中，相同的数据常常需要以不同的语言提供出来。例如，一个公司在日本的销售代表可能需要与阿根廷的销售代表访问相同的销售信息。这样，相同的数据就必须能够被全球各地的人所理解。多语言数据问题并不仅仅是简单地将屏幕信息翻译成许多语言；实际上，它们需要在多种语言下维护相同的数据。维护数据是一个更复杂的过程，因为它常常需要能够在一个屏幕上显示关于相同数据的多个语言版本。

消息（message）就是在多种语言中显示错误与其他类型的信息（包括对话框消息、调试消息、日志消息）。这是最简单的问题，因为 Java 资源束（在第 39 章讨论）就是为了支持这一功能而特别构建的。本章将不对这个问题进行进一步的讨论。

40.2 屏 幕 表 示

有 3 种不同的方法与实现可用来支持多语言表示：

- 多个 JSP 方法：适合于小规模应用程序，在这种应用程序中，应用程序的大小（以及后来的维护需求）不能与代价更昂贵的转码程序方法的初期投资相比。
- 字符串注入方法：适合于这样的应用程序：不需要进行大的底层结构投资，牺牲开发人员的易用性来获取维护一个单独的代码行的能力。
- 转码程序方法：适合于大型应用程序，在这种应用程序中，开发时进行的一次性投资是值得的，因为在后来的维护周期中能够大量节省费用。

40.2.1 多个 JSP 方法

在开发过程中，WebSphere 应用程序的表示层包含有一个 JSP 层，它获取通常由内部 servlet 或 EJB 所创建的数据，并且对这些数据进行格式化，从而创建 HTML，WML 或其他标记语言的脚本。然后，通过某些表示软件（例如浏览器、微型浏览器）将这些脚本提供给用户。

为了使用这种方法，需要创建多个 JSP（所需支持的每种语言对应一个）。这是一种简单、经济的方法，由于其具有的简单性，因此推荐在小应用程序中使用这种方法。这个方法的另一个优点就是所得到的布局不会受到影响；因为每个 JSP 都可以被优化，从而创建对应于这种语言的最佳表示。

图 40-1 显示了 ViryaNet 的 Service Process 模块的日语和英语两种语言的登录屏幕。需要说明的是，这个 HTML 是从不同的 JSP 生成的（在这个例子中，所使用的 JSP 很简单）。



图 40-1 日语登录屏幕和英语登录屏幕

下面给出了用来创建这两个登录屏幕的 JSP 脚本。可以看到，日语 JSP 与英语 JSP 中包含的所有代码几乎都相同（因此，这种方法不适合非常复杂的应用程序）。

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

<%@ page extends = "com.rts.eAccess.framework.EAccessBaseServlet" %>
<head>

    <link href="/RTS_HTML_Pages/RTS_General/Css/Customization.css"
rel="stylesheet" title="Customization style sheet" type="text/css" />

    <meta content="text/html" http-equiv="Content-Type">
    <title>
        &#12532;&#12451;&#12522;&#12450;&#12493;&#12483;&#12488;
        &#12469;&#12540;&#12499;&#12473;&#12503;&#12525;&#12475;&#12473;
    </title>
    <link href="/RTS_HTML_Pages/RTS_General/Css/rts.css"
        rel="stylesheet"
        title="ViryaNet Style Sheet"
        type="text/css" />
    <link href="/RTS_HTML_Pages/RTS_General/Css/rts_remote.css"
        rel="stylesheet"
        title="ViryaNet Service Process Style Sheet"
        type="text/css" />
    <script language="JavaScript" type="TEXT/JAVASCRIPT">

        function fnTrapKD()
        {
            if(event.keyCode == 13)
            {
                formLogin.submit();
            }
        }
    </script>
</head>
    <body onkeydown="fnTrapKD()" onload="showMsg()"
        onresize="window.location.href = window.location.href;">
    <jsp:scriptlet> if (isExplorer(request)){ </jsp:scriptlet>
        <DIV id="images" CLASS="outerLayer">
    <jsp:scriptlet>}
    else{</jsp:scriptlet>
    <DIV align="center">
    <jsp:scriptlet> } </jsp:scriptlet>

        
```

[illegible]

使用多个 JSP 就意味着需要为不同的用户群利用相应的 JSP 来构建页面，但现在的问题是，系统如何知道应该为哪个用户应用哪个 JSP 呢？一种方案（这种方案可能在几年前被使用）就是维护关于用户的信息（例如，在一个用户 profile 数据库或在一个.ini 文件中），并且当用户登录时，开始将服务器代码转移为使用适当的 JSP。但在如今电子商务的时代，访问系统的用户数量可能是成百上千乃至成千上万的，并且系统需要支持那些未必在系统中已经存在记录的用户，因此，就必须有一种更具可伸缩性的方法来解决这个问题。

在确定应该为用户创建的表示时，可以使用这样的事实：浏览器记录了关于它们所接收的编码的信息，并且浏览器将这一信息传递给服务器。每个浏览器都可以让用户选择他或她所用的编码方式。实际上，所选择的编码方式就等价于用户的区域。图 40-2 显示了 Microsoft Internet Explorer 的编码方式选择。

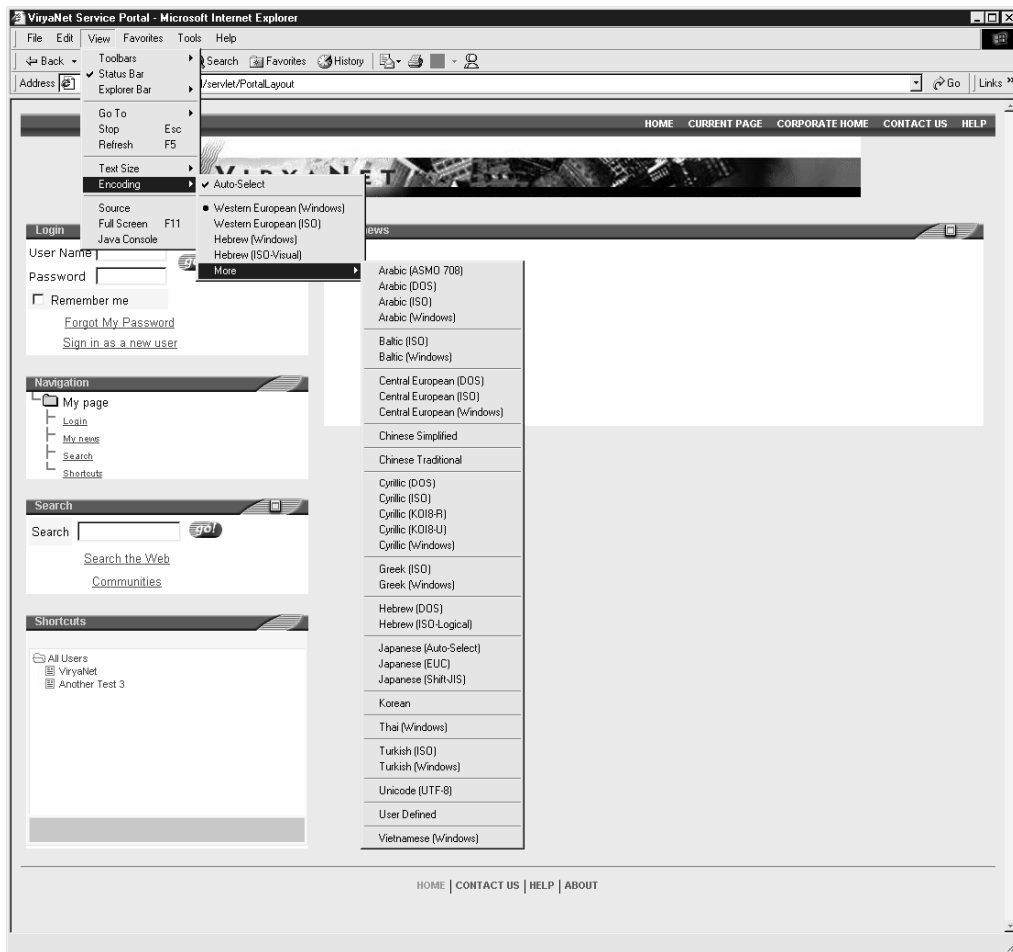


图 40-2 选择浏览器所用的编码方式

每当浏览器向服务器发送一条 HTTP 请求时，就在这条请求的 HTTP 头部中向服务器传递这个区域信息。服务器将使用这个信息（例如，用来选择适当的 JSP）。图 40-3 显示了 WebSphere 的 snoop servlet 所产生的输出，它在 HTTP 请求中输出项（请注意 Request Headers 下称为 accept-language 的字段）。

Request method	GET
Request URI	/servlet/snoop
Request protocol	HTTP/1.1
Servlet path	/servlet/snoop
Path info	<none>
Path translated	<none>
Character encoding	<none>
Query string	<none>
Content length	<none>
Content type	<none>
Server name	ron2k.rtssoftware.co.il
Server port	80
Remote user	<none>
Remote address	127.0.0.1
Remote host	127.0.0.1
Authorization scheme	<none>
Preferred Client Locale	en_US
All Client Locales	en_US
All Client Locales	iw
Context Path	
User Principal	none

Request headers

accept	*/*
accept-encoding	gzip, deflate
accept-language	en-us,he;q=0.5
connection	Keep-Alive
cookie	AG_LOGIN999999=1llong; sesessionid=0BR002IAAAAAQ5YAAAA4XPI
host	ron2k.rtssoftware.co.il
user-agent	Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0; ConversayVoiceSurfer, Conversa Web 3.2.0; Voice Surfer 1.0.0; Conversa Web 4.0.0)

Client cookies

AG_LOGIN999999	1llong
sesessionid	0BR002IAAAAAQ5YAAAA4XPI

图 40-3 snoop servlet 输出

下列程序清单显示了 ViryaNet Service hub 中的一个代码段，这个代码段负责将区域信息转

发到服务器,从而使得服务器能够使用正确的 JSP。这个代码使用 HTTP 请求中存储的区域。该方法将查找在匹配过程中最接近于这个区域的 JSP。首先,它试图使用完全的区域规格说明(例如 jp_JP_WIN),但如果这一匹配没有成功,它将基于部分区域(例如 jp_JP,甚至 jp)搜索 JSP。

```
private final static String findBestFile (
    String jspBaseFolder, String jspName, Locale locale) {
    String jspFolder = getJspFileFolder();

    String localeName = locale.toString();

    String jspPath;
    File jspFile;
    int lastUnderscoreIndex;

    while (true) {
        jspPath = jspBaseFolder + localeName + "/" + jspName;
        jspFile = new File (jspFolder + jspPath);

        if (jspFile.exists()) {
            RTSTracker.displayString(
                JspRedirector.class.toString() +
                ".findBestMatch() - found jsp: " + jspPath +
                " for locale " + locale);
            return jspPath;
        }

        lastUnderscoreIndex = localeName.lastIndexOf("_");
        if (lastUnderscoreIndex == -1)
            break;

        localeName = localeName.substring(0, lastUnderscoreIndex);
    }

    // try again with the default locale
    localeName = PortalConstants.getDefaultLocale().toString();
    while (true) {
        jspPath = jspBaseFolder + localeName + "/" + jspName;
        jspFile = new File (jspFolder + jspPath);

        if (jspFile.exists()) {
            RTSTracker.displayString(
                JspRedirector.class.toString() +
```

```

        ".findBestMatch() - found jsp: " +
        jspPath + " for locale " + locale);
    return jspPath;
}

lastUnderscoreIndex = localeName.lastIndexOf("_");
if (lastUnderscoreIndex == -1)
    break;

localeName = localeName.substring(0, lastUnderscoreIndex);
}

jspPath = jspBaseFolder + jspName;
jspFile = new File (jspFolder + jspPath);

if (jspFile.exists()) {
    RTTracker.displayString(JspRedirector.class.toString() +
        ".findBestMatch() - found jsp: " + jspPath +
        " for locale " + locale);
    return jspPath;
}

RTTracker.displayString(JspRedirector.class.toString() +
    ".findBestMatch() - not found jsp file for: pagename= " +
    jspName + ", basefolder=" + jspBaseFolder + ", locale=" +
    locale + " & file folder=" + jspFolder);
return null;
}

```

为了使这个方案能够正确工作，需要按照区域来组织 JSP。因此，在 ViryaNet Service hub 中，所有 JSP 都存储在包括区域规格说明的目录结构中（如图 40-4 所示）。

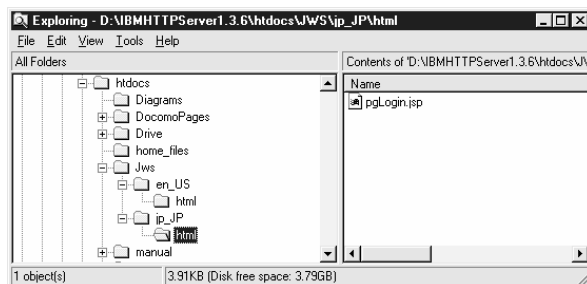


图 40-4 Windows Explorer 中基于区域的目录层次

40.2.2 字符串注入方法

这种方法在一个单独的 JSP 中使用字符串注入方法。在编写这个 JSP 时，JSP 中没有直接包含将要被表示出来的字符串。相反，为了查找实际被表示的字符串，将基于一个关键字调用 Java 例程（为了简单，这个关键字通常就是 en_US 特定区域文件中的字符串）。系统在一个数据库中为所有支持的区域维护字符串。对 Java 例程的调用过程中，使用浏览器传入的区域信息作为关键字来查找正确的字符串，这些字符串将被放置在传递给浏览器的生成页面中。下面显示了一个关于登录屏幕的脚本示例：

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

<%@ page
extends = "com.rts.eAccess.framework.EAccessBaseServlet"
import=com.rts.ml.translate.MultiLingual %>
<head>
<link href="/RTS_HTML_Pages/RTS_General/Css/Customization.css"
rel="stylesheet" title="Customization style sheet"
type="text/css" />
<bean create="NO" introspect="NO" name="SelfServiceException"
scope="REQUEST" type="com.rts.selfservice.SelfServiceException">
</bean>
<meta content="text/html" http-equiv="Content-Type">
<title class="MultiLingual">

<jsp:expression
MultiLingual.translate("pgLogin.jsp", "TITLE", "H",
"Login to ViryaNet Service Process", "NA", "true")
</jsp:expression>
</title>
<link href="/RTS_HTML_Pages/RTS_General/Css/rts.css"
rel="stylesheet" title="ViryaNet Style Sheet" type="text/css" />
<link href="/RTS_HTML_Pages/RTS_General/Css/rts_remote.css"
rel="stylesheet" title="ViryaNet Service Process Style Sheet"
type="text/css" />
<script language="JavaScript" type="TEXT/JAVASCRIPT">
function showMsg()
{
<jsp:scriptlet>
if ((SelfServiceException.getClassName() != null) ||
(SelfServiceException.getOverride() != null))
out.write("alert(\"" + SelfServiceException.toString() +
```

```

        "\");");
    </ jsp:scriptlet >
}

function fnTrapKD()
{
    if (event.keyCode == 13)
    {
        formLogin.submit();
    }
}
</script>
</head>
<body onkeydown="fnTrapKD()" onload="showMsg()"
    onresize="window.location.href = window.location.href;">
    < jsp:scriptlet> if (isExplorer(request)){ </ jsp:scriptlet >
        <DIV id="images" CLASS="outerLayer">
        < jsp:scriptlet >}
    else{ </ jsp:scriptlet >
    <DIV align="center">
    < jsp:scriptlet >} </ jsp:scriptlet >

        
        <br />
        
        <br />
        

    < jsp:scriptlet> if(isNetscape(request)){ </ jsp:scriptlet >
        <div align="center" class="nestedLayer" id="login" name="login">
            ViryaNet Service Process
        < jsp:scriptlet> } else{ </ jsp:scriptlet >
            <div align="center" class="nestedLayer" id="login" name="login">
                <span class="MultiLingual" id="100">

    <jsp:expression >
        MultiLingual.translate("pgLogin.jsp", "100", "T",
            "ViryaNet Service Process ", "NA", "false")
    </jsp:expression >
    <jsp:expression> "<!--" </jsp:expression>

```

```
<jsp:expression> "-->" </jsp:expression>
</span>
```

```
<jsp:scriptlet > </jsp:scriptlet >  
    <form action="/servlet/RequestHandler" method="POST"  
        name="formLogin" target="_top">  
        <input name="task" type="HIDDEN" value="BnLogin" />  
        <input name="action" type="HIDDEN" value="run" />  
        <table align="center" border="0">  
            <tr>  
                <td class="TableBodyFontBold">  
                    <jsp:scriptlet>  
                        int j1 =9;  
                        if (isNetscape(request))  
                            for (int ii = 0; ii < j1; ii++)  
                                {  
                                    </jsp:scriptlet >  
                                    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~  
                                    <jsp:scriptlet>  
                                        }  
                                    </jsp:scriptlet >  
                                }  
                            }  
                    <span class="MultiLingual" id="200">  
  
        <jsp:expression >  
            MultiLingual.translate(  
                "pgLogin.jsp", "200", "T", "User Name", "NA", "false")  
        </jsp:expression >  
        <jsp:expression> "<!--" </jsp:expression>  
        User Name  
        <jsp:expression> "-->" </jsp:expression>  
        </span>  
    </td>  
    <td>  
        <input class="TableBodyFont" name="username" size="10"  
            type="TEXT" value="" />  
    </td>  
</tr>  
<tr>  
    <td class="TableBodyFontBold">  
        <jsp:scriptlet>  
            int j2 =9;  
            if (isNetscape(request))  
                for (int ii = 0; ii < j2; ii++)
```



```

        </span>
    </div>
</td>
</tr>
</table>
</form>
</div>
</div>
</div>
</body>
</html>

```

下面给出了执行实际查找动作的代码。这种方法的优点是使用一个单独的 JSP 处理所有的区域，并且很容易执行字符串的转换，而且字符串转换是集中维护的。这种方法的缺点（而且是一个很大的缺点）就是 JSP 的构成不好（这并不是说它们存在什么错误，而是说在所有的字符串都被删除之后，如果通过一个 WYSIWYG 编辑器查看这个 JSP，那么其脚本的查看对于开发人员是不方便的）。

```

package com.rts.ml.translate;

import com.sun.java.util.collections.*;
import java.util.Locale;
import java.sql.*;
import com.rts.sql.*;
import com.rts.gen.RTSTracker;
import com.rts.gen.RTS;
import java.util.PropertyResourceBundle;
import java.util.MissingResourceException;
import com.rts.util.RTSUtil;
import com.rts.ml.RTSMLManager;

/**
 * Call stored function to translate string.
 * Reads a translation on/off switch from server.properties
 * If switch is not found in server.properties - sets a switch to OFF
 */
public class MultiLingual {

    public static Locale TRANSLATION_NOT_NEEDED = Locale.US;
    public static final String NBSP = "&nbsp;";

    NBSP_WRONG = "&nbsp;" ;
    private static Connection _connection;

```

```
private static Map _cache;
private static boolean _filled = false;

private static String _translation = null;

public MultiLingual() {
    super();
}
private static void clearCache() {
    _cache = null;
}
private static void display(String s) {
    RTSTracker.displayString(s);
}
private static void fillCacheFromDb(String jspName) {

    display("Fill cache for =" + jspName );

    com.rts.sql.RTSConnection con = null;
    com.rts.sql.RTSPreparedStatement stmt = null;
    String id = null,
           type = null,
           textJsp = null,
           textMlt = null;

    String sql =
        " SELECT ID,TYPE,TEXT_JSP,TEXT_MLT FROM JSP_TEXT where JSP_NAME=?";

    try {
        con = com.rts.sql.RTSDB.getSelectConnection();
        stmt = con.prepareStatement(sql);
        stmt.getStmt().setString(1, jspName);
        ResultSet res = stmt.executeQuery();
        boolean b = false;
        while (res.next())
        {
            id = res.getString(1);
            type = res.getString(2);
            textJsp = res.getString(3);
            textMlt = res.getString(4);
            if( id != null ) {
                MltCacheObject o = new MltCacheObject();
                o.setId(id);
            }
        }
    }
}
```

```
        o.setType(type);
        o.setTextJsp(textJsp);
        o.setTextMlt(textMlt);

        setToCache(jspName , id , o);
        b = true;
    }
}
if( b) {
    RTSTracker.displayString("Translation Screen Cache filled");
}
}
catch (SQLException ex) {
    com.rts.gen.RTSTracker.handleException(
        "MultiLingual : fillCacheFromDb", ex);
}
finally {
    com.rts.gen.RTS.close(con, stmt, null);
}
}

private static Map getCache() {

    if( _cache == null ) {
        _cache = new Hashtable();
    }

    return _cache;
}

private static MltCacheObject getFromCache(String s) {

    return (MltCacheObject) getCache().get(s);
}

private static String getTranslationFromCache(
String jspName, String fieldId, String fieldType, String text) {

    String translated = null;

    StringBuffer sb = new StringBuffer(jspName);
    sb.append(".");
    sb.append(fieldId);
    MltCacheObject o = getFromCache(sb.toString());
```

```

        if (o != null) {
            translated = o.getTextMlt();
        }
        return translated;
    }
    private static String getTranslationFromDb(
        String jspName, String fieldId, String fieldType, String text, String language)
    {

        display(" getTranslationFromDb JspName=" + jspName + " fieldId=" +
            fieldId + " text=" +text);

        String translated = text;

        com.rts.sql.RTSConnection con = null;
        com.rts.sql.RTSCallableStatement stmt = null;
        String old = text;
        String pureText = stripNbsp(text);

        try {
            String sql = "{call JSP_TEXT_TRANS(?,?,?,?,?) }";
            con = com.rts.sql.RTSDB.getSelectConnection();
            stmt = con.prepareCall(sql);

            stmt.bindValue(1, jspName);
            stmt.bindValue(2, fieldId);
            stmt.bindValue(3, fieldType);
            stmt.bindValue(4, pureText);
            stmt.getStmt().registerOutParameter(
                5, java.sql.Types.VARCHAR);

            stmt.execute();
            translated = stmt.getStmt().getString(5);
            RTSTracker.displayString("translation - OK");
        }
        catch (SQLException ex) {
            com.rts.gen.RTSTracker.handleException(
                "MultiLingual : translate", ex);
        }
        finally {
            com.rts.gen.RTS.close(con, stmt, null);
        }

        // if there were nbsp in the original string

```

```
// replace in the original string the non-bnsp value
// with the new value saving nbsp-aces
if (!pureText.equals(old)) {
    String str = RTSUtil.replace(
        old, pureText, translated);
    translated = str;
}

display("Translation result=" + translated);
return translated;
}

private static boolean getTranslationSwitch() {

    boolean _switch = true;
    if( _translation == null ) {
        PropertyResourceBundle configBundle =
            (PropertyResourceBundle)
                PropertyResourceBundle.getBundle("server");
        try {
            _translation = configBundle.getString("translation");
        }
        catch(MissingResourceException ex) {
            _translation = "off" ;
        }
    }

    if( "0".equalsIgnoreCase(_translation) ||
        "off".equalsIgnoreCase(_translation) ||
        "false".equalsIgnoreCase(_translation) ||
        "no".equalsIgnoreCase(_translation) )
    {

        _switch = false;
    }

    return _switch;
}

public static void main(String[] args) {
    String s = MultiLingual.translate(
        "JSpName", "1", "T","Site","NA", "true");
}
```

```

        RTSTracker.displayString(s);
    }

    private static boolean needTranslation() {
        boolean need = true;

        boolean _switch = getTranslationSwitch();

        if( ! _switch ) {
            return false;
        }

        if (
TRANSLATION_NOT_NEEDED.equals(RTSMLManager.getClientLocale())){
            need = false;
        }
        else if(
TRANSLATION_NOT_NEEDED.getDisplayLanguage().equalsIgnoreCase(
RTSMLManager.getClientLocale().getDisplayLanguage())){
            need = false;
        }

        return need;
    }
    private static String realTranslate_NEW(
String jspName, String fieldId, String fieldType, String text, String language)
    {

        display("TrealTranslate_NEW =" + jspName + " fieldId=" +fieldId +
            " text=" +text);
        String translated = getTranslationFromCache( jspName,
            fieldId, fieldType, text );

        if( translated == null ) {

            translated = getTranslationFromDb(
                jspName, fieldId, fieldType, text, language);
            MltCacheObject o = new MltCacheObject();
            o.setId(fieldId);
            o.setType(fieldType);

```

```

        o.setTextJsp(text);
        o.setTextMlt(translated);

        setToCache(jspName , fieldId , o);
        RTSTracker.displayString(
            "Translation Screen Cache appended by "+ text +
            " - " + translated);
    }

    return translated;

}

private static void setToCache(String key, MltCacheObject o) {
    getCache().put(key, o);
}

private static void setToCache(
    String jspName, String fieldId, MltCacheObject o) {
    StringBuffer sb = new StringBuffer(jspName);
    sb.append(".");
    sb.append(fieldId);
    setToCache(sb.toString(), o);
}

/**
 * Sets a Locale for which the translation is not needed </br>
 * The default is Locale.US
 * @param Locale locale
 *
 */
public static void setTranslationNotNeeded(Locale l) {
    TRANSLATION_NOT_NEEDED = l;
}

/**
 * Removes nbsp from the string.
 * Also removes a wrong nbsp (without ; at the end )
 * @param String string
 * @return String string stripped from nbsp
 */
private static String stripNbsp(String s) {

    String r = new String();

    String stripped = null;
    int i = s.indexOf(NBSP);
    if (i > -1) {

```

```

        stripped = RTSUtil.replace(s, NBSP, r);
    }
    else {
        stripped = s;
    }

    i = stripped.indexOf(NBSP_WRONG);
    String stripped1 = null;
    if (i > -1) {
        stripped1 = RTSUtil.replace(stripped, NBSP_WRONG, r);
        display("Wrong NBSP occured in line:" + s );
    }
    else {
        stripped1 = stripped;
    }

    return stripped1;
}
/**
 * The API method. Translates a given string into the default client
 * language if that language is not US English.</br>
 * Uses an STP JSP_TEXT_TRANS in its translation process.</br>
 *
 * @param String jsp screen name (like pgMyFavoriteLovelyScreen.jsp)
 * @param String field id - the unique identifier of the field within
 * the screen (usually a number which is built automatically)
 * @param String field type (T - text, H - title, B - botton, O - option)
 * @param String text to translate
 * @param String language - currently optional & may be null
 * @return String translated string
 */
public static String translate(
    String jspName, String fieldId, String fieldType,
    String s, String language, String fillCache) {

    if ( needTranslation() ) {
        display("start translating..");
        if( fillCache.equalsIgnoreCase("true") ) {
            display("start building a cache ..");
            fillCacheFromDb(jspName);

```



```

    }
    return realTranslate_NEW(
        jspName, fieldId, fieldType, s, language);
}
else {
    return s;
}
}
}

```

40.2.3 转码程序方法

第三种方法（我们认为优于其他两种方法的方法）维护了关于 JSP 的“自然”形式，表现在将字符串插入到 JSP 中，但字符串被打上了标记。它利用 SPAN 标记来表示某种字符串应该被解释为需要进行多语言查找。然后，有一个称为转码程序（transcoder；请不要与 IBM WebSphere Transcoder 混淆）的组件将负责基于区域信息把 SPAN 标记中的字符串替换为另一个字符串。因此，这种方法具有所有的优点（一个单独的 JSP、转换字符串放在数据库中、“看上去更好”的 JSP）。

下列脚本显示了这种方法对应的 JSP。下面的第二个脚本则显示了负责将 SPAN 元素中的字符串进行替换的转码程序代码。

```

<% com.rts.transcoder.servlet.RTSMobileTranscoderServlet
    .preparePageForFiltering(request,response); %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<html>
<%@ page
extends = "com.rts.eAccess.framework.EAccessBaseServlet" %>
<head>
<link href="/RTS_HTML_Pages/RTS_General/Css/Customization.css"
rel="stylesheet" title="Customization style sheet" type="text/css"
/>
<bean create="NO" introspect="NO" name="SelfServiceException"
scope="REQUEST" type="com.rts.selfservice.SelfServiceException">
</bean>
<meta content="text/html" http-equiv="Content-Type">
<title class="MultiLingual">
Login to ViryaNet Service Process
</title>
<link href="/RTS_HTML_Pages/RTS_General/Css/rts.css"rel="stylesheet"
title="ViryaNet Style Sheet" type="text/css" />
<link href="/RTS_HTML_Pages/RTS_General/Css/rts_remote.css"
rel="stylesheet" title="ViryaNet Service Process Style Sheet"

```

```

        type="text/css" />
        <script language="JavaScript" type="TEXT/JAVASCRIPT">
function showMsg()
{
    <jsp:scriptlet>
    /* if ((SelfServiceException.getClassName() != null) ||
       (SelfServiceException.getOverride() != null))

        out.write("alert(\"" + SelfServiceException.toString() +
                   "\");"); */
    </ jsp:scriptlet >
}
function fnTrapKD()
{
    if (event.keyCode == 13)
    {
        formLogin.submit();
    }
}
</script>
</head>
<body onkeydown="fnTrapKD()" onload="showMsg()"
    onresize="window.location.href = window.location.href;">
< jsp:scriptlet> if (isExplorer(request)){ </ jsp:scriptlet >
    <DIV id="images" CLASS="outerLayer">
< jsp:scriptlet >}
else{</ jsp:scriptlet >
<DIV align="center">
< jsp:scriptlet >} </ jsp:scriptlet >
    
    <br />
    
    <br />
    
< jsp:scriptlet> if(isNetscape(request)){ </ jsp:scriptlet >
    <div align="center" class="nestedLayer" id="login"
        name="login">ViryaNet Service Process
< jsp:scriptlet> } else{ </ jsp:scriptlet >
    <div align="center" class="nestedLayer" id="login" name="login">
        <span class="MultiLingual" id="100">
            ViryaNet Service Process

```

```
</span>
< jsp:scriptlet> } </ jsp:scriptlet >
    <form action="/servlet/RequestHandler" method="POST"
        name="formLogin" target="_top">
        <input name="task" type="HIDDEN" value="BnLogin" />
        <input name="action" type="HIDDEN" value="run" />
        <table align="center" border="0">
            <tr>
                <td class="TableBodyFontBold">
                    < jsp:scriptlet>
                        int j1 =9;
                        if (isNetscape(request))
                            for (int ii = 0; ii < j1; ii++)
                                {
                                    </ jsp:scriptlet >
                                        &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
                                    < jsp:scriptlet>
                                        }
                                </ jsp:scriptlet >
                                <span class="MultiLingual" id="200">
                                    User Name
                                </span>
                            </td>
                            <td>
                                <input class="TableBodyFont" name="username" size="10"
                                    type="TEXT" value="" />
                            </td>
                        </tr>
                        <tr>
                            <td class="TableBodyFontBold">
                                < jsp:scriptlet>
                                    int j2 =9;
                                    if (isNetscape(request))
                                        for (int ii = 0; ii < j2; ii++)
                                            {
                                                </ jsp:scriptlet >
                                                    &nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&nbsp;&~
                                                < jsp:scriptlet>
                                                    }
                                            </ jsp:scriptlet >
                                            <span class="MultiLingual" id="300">
                                                Password
                                            </span>
                                        </td>
```



```
import java.lang.System.*;
import com.commerceone.xdk.standards.sax.*;
import com.rts.transcoder.taglist.*;
import com.sun.java.util.collections.*;
import com.rts.gen.RTSTracker;
import java.util.ResourceBundle;
import java.util.PropertyResourceBundle;
import com.rts.ml.translate.MultiLingual;
import org.xml.sax.helpers.AttributeListImpl;
import java.util.Locale;

public class MultilingualHandler implements
    TranscoderModifier, Tags,
    com.rts.transcoder.taglist.Attributes {
    public final static int MAX_JSP_NAME_SIZE = 50;
    public final static String ML_CLASS = "MultiLingual";

    boolean translateMode;
    String translateId;
    String translateType;
    String specialId;
    int translateIndex;
    TranscoderHandler handler;
    String language;
    String jspName;

    public void end(){}
    public String mlTranslate(
String fieldId, String fieldType, String s, String fillCache) {
        s = s.trim();
        String src = specialId;
        if(src == null) {
            src = s;
        } else {
            fieldId = "T" + specialId.hashCode();
        }

        String dst = MultiLingual.translate(jspName, fieldId, fieldType, src,
language, fillCache);
        //System.out.println("Translate: <<" +jspName+" , "+fieldId+" ,
            "+fieldType+" , "+src+" , "+language+" , "+fillCache);

        if(specialId == null) {
            return dst;
        }
    }
}
```

```
}

String s1 = specialId.trim();
String s2 = s.trim();
String s3 = dst.trim();
specialId = null;

StringBuffer sb = new StringBuffer();
for(int i = 0; i < s1.length(); i++) {
    char ch = s1.charAt(i);
    if(ch == '%') {
        i++;
        if(s1.charAt(i) != '%') {
            ch >= '\n';
        }
    }
    sb.append(ch);
}

s1 = sb.toString();
java.util.StringTokenizer st = new java.util.StringTokenizer(s1, "\n");
java.util.Vector v = new java.util.Vector();
boolean first = true;
s2=s2.trim();

while(st.hasMoreTokens()) {
    String str = st.nextToken().trim();
    int idx = s2.indexOf(str);
    if(idx >= 0) {
        if(idx != 0) {
            v.addElement(s2.substring(0,idx).trim());
        } else
        if(first) {
            first = false;
            if(s1.indexOf('\n') == 0) {
                v.addElement("");
            }
        }
        s2 = s2.substring(idx + str.length());
    }
}

if(s2.trim().length() > 0) {
    v.addElement(s2.trim());
}
```

```

        sb.setLength(0);
        for(int i = 0; i < s3.length(); i++) {
            char ch = s3.charAt(i);
            if(ch == '%') {
                i++;
                if(s3.charAt(i) != '%') {
                    if(v.size() > 0) {
                        String str = (String)v.elementAt(0);
                        sb.append(str);
                        v.removeElementAt(0);
                    }
                    continue;
                }
            }
            sb.append(ch);
        }
        return sb.toString();
    }
    /**
     * printing the start tag
     */
    public AttributeList modifyAttributes(AttributeList atts, TagDescription td)
    {
        AttributeListImpl attributes = (AttributeListImpl)atts;
        if(td == null) {
            return atts;
        }

        int id = td.getId();

        if(!translateMode) {
            if(id == t_TITLE) {
                String mlClass = atts.getValue(an_CLASS);
                if(mlClass != null &&
                    mlClass.equalsIgnoreCase(ML_CLASS)) {
                    translateMode = true;
                    translateType = "H";
                    translateId = "TITLE";
                    translateIndex = -1;
                }
                specialId = atts.getValue(an_ID);
            } else
                if(id == t_SPAN) {

```

```

        String mlClass = attributes.getValue(an_CLASS);
        translateId = attributes.getValue(an_ID);
        if(translateId != null && mlClass != null &&
            mlClass.equalsIgnoreCase(ML_CLASS)) {
            translateMode = true;
            translateType = null;
            translateIndex = -1;
        }
    }
} else {
    if(id == t_INPUT) {
        String type = attributes.getValue(an_TYPE);
        String value = attributes.getValue(an_VALUE);

        if(type != null && (type.equalsIgnoreCase("button") ||
            type.equalsIgnoreCase("submit") ||
            type.equalsIgnoreCase("reset"))) {
            value = mlTranslate(
                translateId, "B", value, "false");
            attributes.removeAttribute(an_VALUE);
            attributes.addAttribute(an_VALUE, "", value);
        }
    } else
    if(id == t_OPTION) {
        specialId = atts.getValue(an_ID);
        translateType = "O";
        translateIndex ++;
    }
}
return attributes;
}
/**
 * processing the end tag
 */
public String modifyTag(String tagName, TagDescription td, boolean isEndTag){
    if(isEndTag && td != null) {
        if(td.getId() == t_TITLE || td.getId() == t_SPAN) {
            translateMode = false;
        }
    }
    return tagName;
}
public String modifyText(String s) {
    if(translateMode && s.trim().length() > 0) {

```



```

        String id = translateId;
        if(translateIndex >= 0) {
            id += "." + translateIndex;
        }
        String flag = (translateType != null && translateType.equals("H"))
            ? "true" : "false";
        s = mlTranslate(id, translateType == null
            ? "T" : translateType, s, flag);
    }
    return s;
}

public void reStart(TranscoderHandler handler) {
    start(handler);
}

public void start(TranscoderHandler handler) {
    this.handler = handler;
    language = "NA";
    translateMode = false;
    translateId = null;
    translateType = null;
    specialId = null;
    translateIndex = -1;

    if(handler.pageName != null) {
        jspName = handler.pageName;
        if(jspName.length() > MAX_JSP_NAME_SIZE) {
            jspName = jspName.substring(
                jspName.length() - MAX_JSP_NAME_SIZE);
        }
    }
}
}

```

40.3 多语言数据

对数据的多语言支持虽然比较麻烦，但却很简单。同样，在这里给出两种不同的方法：一种方法是用应用程序服务器代码，另一种方法就是用数据库视图。

下面的脚本显示了一种方法，这个方法就是引入一个称为 `MLString` 的新封装。对于某个数据字符串，在这个类中维护了多个字符串，分别对应于此应用程序（或者说是数据库方案）所支持的不同语言。这就意味着当从数据库中读入 `MLString` 时，所有语言的所有值都被读入。然后，当用户想要修改区域（或者在不进行修改的情况下以不同的区域浏览数据）时，

这个数据在服务器中就已经可用，并且用户的请求能够很容易得到满足。

```
package com.rts.ml;

//Title:      FW ML
//Version:
//Copyright:   Copyright (c) 2001
//Author:      ViryaNet
//Company:     ViryaNet
//Description: FW ML

import com.sun.java.util.collections.*;
import com.rts.fw.RTSBusinessObject;
import com.rts.session.RTSEncodable;
import com.rts.session.RTSEncDec;
import com.rts.session.RTSEncoder;
import com.rts.session.RTSDecoder;
import com.rts.gen.RTS;
import com.rts.gen.RTSTracker;
//import com.rts.util.RTSUtil;
//import com.rts.util.RTSIterator;
import java.util.Locale;
import com.rts.gen.RTSResultTable;
import com.rts.dbl.*;
import com.rts.xml.schema.description.SchemaDescriptor;

abstract public class MLString implements
    Cloneable, RTSEncodable, SchemaDescriptor {

    /** dictionary with all pairs of language - String
     */
    private Map _stringInLang;

    /**
     * Class constructor
     */

    public MLString() {
        this((String[])null, (String[])null);
    }

}
```

```

/**
 * Class constructor
 *
 * @param languages
 *   array of language codes. The codes can be
 *   a language code with optional country code.br
 *   For example:<br><ul><tt>
 *   <li>enus -- US English
 *   <li>fr   -- French
 *   <li>ru   -- Russian
 *   </ul></tt><br>
 *   Language codes are taken from:
 *   <a href=
 *   "http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_639.html">
 *   ISO 639</a>.
 *   <br>
 *   Country codes are taken from:
 *   <a
href="http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html">ISO
3166</a>.
 * @param values
 *   array of values. a value for each language code
 */
public MLString(String[] languages, String[] values) {
    initialize(languages, values);
}

/**
 * Class constructor: Use the given business class and the name of
 * the business variable in that class, to get the list of supported
 * languages. String values are null.
 *
 * @param aClass a business class
 * @param bvName a business variables path
 */
public MLString(Class aClass, String path) {

    this( RTSMLManager.getSupportedLanguageCodes(aClass, path), null);
}

/**
 * adds new language + String to the dictionary
 * @param language a language (or language + country) code
 * @param value    a value (can be null)
 */
public void add(String language,String value)

```

```

{
    getLanguagesDict().put(language,value);
}
public Object asXml( String vName, Object value )
{

    return null;

}

/**
 * Clone this object. The returned object is a deep copy of this object.
 */
public Object clone()
{
    MLString obj;

    try
    {
        obj = (MLString) super.clone();
    }
    catch(CloneNotSupportedException ex)
    {
        // Ignore
        obj = null;
    }

    obj.setLanguagesDict( new HashMap(getLanguagesDict()) );
    return obj;
}
/**
 * Decode the given object.
 * <br>
 * <blockquote><pre>
 * enc:
 * [ ENC_TYPE_MLSTRING class-name languages-dictionary ]
 *
 * </pre></blockquote>
 * <br>
 *
 * @param enc encoded object
 * @return the decoded object
 */
public Object decode(Object [] enc, RTSDecoder decoder)
{

```

```
        _stringInLang = (Map)decoder.decode((Object[])enc[2]);
        return this;
    }
    /**
     * Answers a new empty MLString
     */
    public MLString empty()
    {
        String [] langs = getLanguages();

        MLString emptyML = (MLString)clone();
        for( int i = 0 ; i < langs.length ; ++i )
        {
            emptyML.setValue(langs[i], null);
        }

        return emptyML;
    }
    /**
     * Encode the given object.
     */
    public Object [] encode(RTSEncoder encoder)
    {
        Object [] enc = new Object[3];

        enc[0] = RTSEncDec.ENC_TYPE_MLSTRING;
        enc[1] = getClass().getName();
        enc[2] = encoder.encode(_stringInLang);

        return enc;
    }
    /**
     * Returns the default language
     */
    private static String getDefaultLanguage()
    {
        return "en";
    }
    /**
     * Returns the default languages
     */
```

```
public static String [] getDefaultLanguages()
{
    String [] langs = new String[1];
    langs[0] = getDefaultLanguage();
    return langs;
}
/**
 * Get the languages dictionary
 */

public String[] getLanguages()
{
    Set s = getLanguagesDict().keySet();
    String[] _langs = new String[s.size()];
    Iterator it =s.iterator();
    int i = 0;
    while( it.hasNext() ) {
        _langs[i++] = (String) it.next();
    }

    return _langs;
}
/**
 * Get the languages dictionary
 */

protected Map getLanguagesDict()
{
    return _stringInLang;
}
/**
 * Returns String for the current language
 * @param language java.lang.String
 * added by Zeev 26/9/99
 */
public String getValue() {
    return (String) _stringInLang.get( getDefaultLanguage() );
}
/**
 * Returns String for the given language
 * @param language a language code, or null if not found
 */
```

```
public String getValue (String lang)
{
    return (String)getLanguagesDict().get(lang);
}
/**
 * SchemaDescriptor implementation
 */
public Object getVariableValue(String vName)
{
    return getLanguagesDict().get( vName);
}
/**
 * Fills a dictionary of languages.
 * If no string given - null will be taken
 * @param String[] languages
 * @param String[] strings
 */
private void initialize(String[] langs, String[] strings)
{
    int min;

    if( langs == null )
        langs = getDefaultLanguages();

    Map dict = new HashMap(langs.length);
    setLanguagesDict(dict);

    if( strings == null )
    {
        min = 0;
    }
    else
    {
        min = langs.length;
        if( strings.length < min )
            min = strings.length;
    }

    int i = 0;

    for (i = 0; i < min; i++)
```

```
{
    dict.put(langs[i], strings[i]);
}

for ( ; i < langs.length ; i++ )
{
    dict.put(langs[i], null);
}
}
/**
 * Check if this object is empty.
 * MLString object is empty if all values of all languages are empty.
 */
public boolean isEmpty()
{
    Iterator itr = getLanguagesDict().values().iterator();
    while(itr.hasNext())
    {
        String str = (String)itr.next();
        if( str != null && !str.equals("") )
            return false;
    }
    return true;
}
/**
 * Returns true if the given language is supported by the mls
 */
public boolean isLanguageIncluded(String language)
{
    return getLanguagesDict().containsKey(language);
}
/**
 * List all supported languages
 */
public String[] listLanguages()
{
    Map dict = getLanguagesDict();
    String [] langs = new String[ dict.size() ];

    int i = 0;
    Iterator itr = dict.keySet().iterator();
    while (itr.hasNext())
    {
        langs[i++] = (String)itr.next();
    }
}
```



```
    }

    return langs;
}
/**
 * Set the languages dictionary
 */

protected void setLanguagesDict(Map dict)
{
    _stringInLang = dict;
}
/**
 * Set a new value for the default language
 */
public void setValue(String value)
{
    getLanguagesDict().put(getDefaultLanguage(), value);
}
/**
 * Set a new value for an existing language
 */
public void setValue(String language, String value)
{
    getLanguagesDict().put(language, value);
}
/**
 * SchemaDescriptor implementation
 */
public void setVariable (String vName, Object val)
{
    getLanguagesDict().put(vName, val);
}

/** returns value by current locale,
 * if doesn't exist, then English version
 */
public String toString()
{
    return RTSTracker.objToString(getLanguagesDict());
}
}
```

第二种方法就是使用数据库视图。数据库中的“真实”表有多个列，分别用于所支持的每种语言。此外，对于系统所支持的每种语言，都存在一个数据库视图。这个视图映射没有在多语言方案中涉及的“真实”表的所有列，也映射区域语言所对应的列。对支持的每种语言都创建一个数据库用户；在这个数据库用户中包含有适当区域的所有视图。

在 WebSphere 应用程序服务器中，可以创建多个连接池，其中每个数据库用户（也就是每个支持的区域）对应一个。当需要为一个请求提供服务时，就提取出这个区域。然后，基于所请求的区域，从适当的池中请求一个数据库连接。由于这个连接映射到与终端用户所用区域相匹配的数据库用户上，因此能够获取到正确的数据字符串。这样，相同的查询（例如 `SELECT PART_DESC FROM PARTS WHERE...`）将基于请求中给出的区域，从“真实”表中获取不同的数据。如图 40-5 所示。

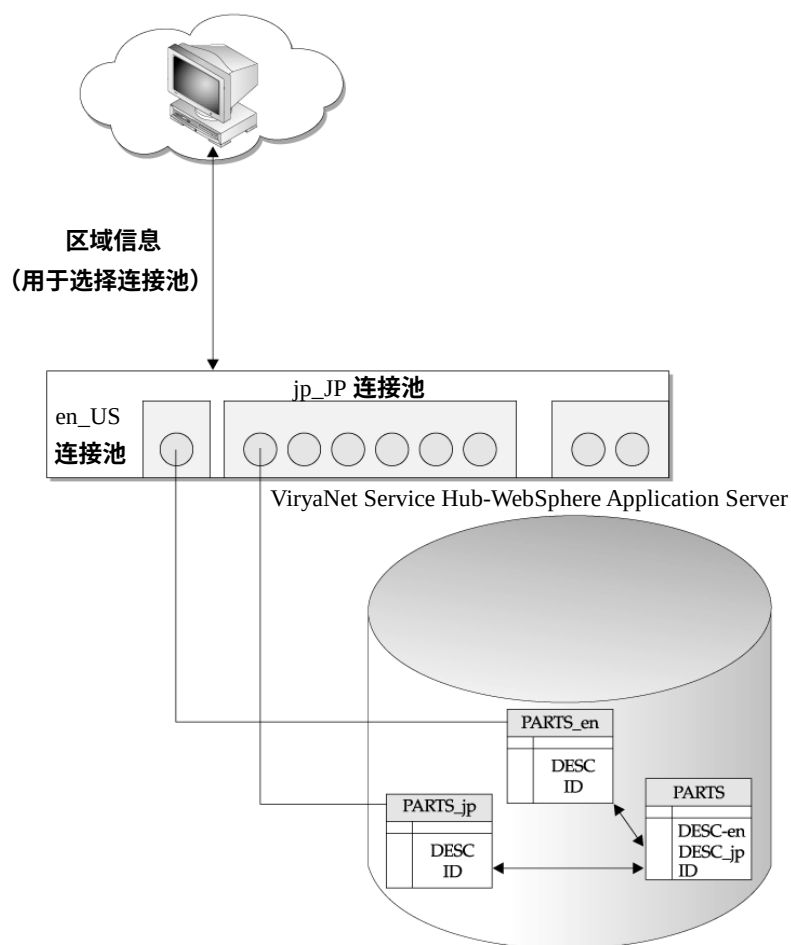


图 40-5 使用数据库视图支持多语言数据

40.4 小 结

现在，读者已经知道了如何使用这些机制与方法：利用这些机制与方法，就可以构建能够在 WebSphere 应用程序服务器中部署的全球化应用程序。记住，在开始使用商业 Web 应用程序之后，并不能预测到什么人会使用这个 Web 应用程序。因此，如果需要将应用程序本地化到不同区域的不同用户群，那么就需要使用这些机制来确保数据被正确地表示。

在转入下一个话题之前，还需要说明的是：千万不能低估本地化的工作量。实际上，进行翻译需要花费比较长的时间，并且实现是比较昂贵的。此外，其他问题（例如数据格式、地址格式、约定、颜色、文化蕴含等等）使得本地化工作更为复杂并且应该引起更多的注意，特别是对于那些没有在受控的业务环境下使用的应用程序以及那些可能为顾客市场提供服务的应用程序更是如此。

第 9 部分 管理 WebSphere 网站

第 41 章 为可伸缩性与容错性部署 WebSphere

WebSphere 是一个用来部署 Web 应用程序的应用程序服务器。本书重点讨论了应该如何开发这些 Web 应用程序。而且在前面详细说明了服务器以及部署在这些服务器上的 Web 应用程序的配置与管理，并且给出了许多相应的例子，但是，关于最终配置的高层视图将留给本章来讨论。本章将仔细讨论一个典型的 WebSphere 部署，其中包括对所有相关组件的讨论。

可以说，一个关于电子商务系统的完全部署要求包括 WebSphere Application Server 在内的一整套技术组件集合。后面将对这些组件进行讨论。本章目标是帮助读者将前面已经学习的不同信息（以及与工作相关的经验）串连在一起，从而管理一个完整的部署方案。

本章的另一个主要目标就是让读者对完全产品部署的服务器拓扑有一个清晰的视图。当设计一个系统时，就必须决定应该购买哪些硬件以及如何进行服务器配置。当然，硬件的需求取决于底层应用程序的性能，本章在这一方面没有进行过多的讨论（因为这个问题的症结在于每个应用程序都有它自己的性能需求与特点）。而且，不考虑实际的性能需求，大多数系统在拓扑方面具有很大的共同点。因此，有的系统可能需要一个双 CPU 服务器，并且使用 866Mhz 的 CPU 以及 Windows 2000 操作系统；而另一个系统则可能需要 4CPU 的服务器，并且使用 SPARC 处理器与 Solaris 操作系统；但这两个系统的拓扑是相似的。

本章描述在任何工程的初始阶段都非常有用的最常用拓扑——无论这个工程是一个电子商务应用程序还是一个内部开发工程。

41.1 服务器拓扑

首先分析比较明显的情况（这样总是开始做某件事的好方法）。为了运行大多数应用程序，至少需要 4 个常用的组件：一个存储数据的数据库服务器，一个运行代码的应用程序服务器，一个处理 HTTP 请求的 WWW 服务器（或称为 Web 服务器）以及一个解释标记语言并将数据表现给用户的浏览器。浏览器归用户所有，因此不必考虑它，但其他 3 个组件是需要工程设计者负责管理的。

图 41-1 显示了一个服务器拓扑，用来演示哪个服务器应该与其他哪些服务器通信。浏览器首先产生一个 HTTP 请求。需要说明的是，这种情景才是应用焦点，而不是 Java 客户在应用程序服务器上直接与 EJB 通信，因为前者被更加广泛地使用。

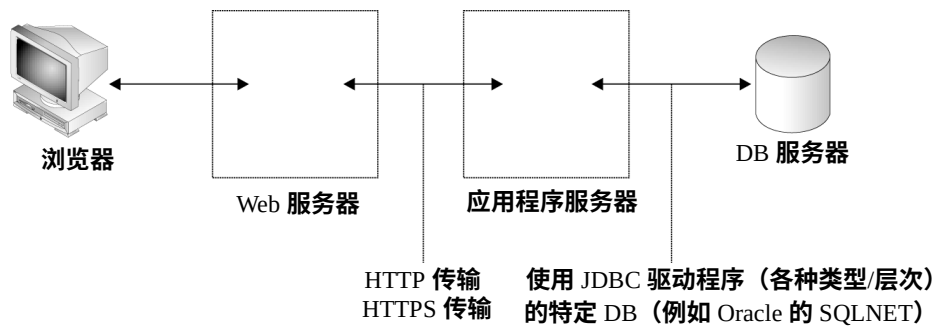


图 41-1 概念性拓扑

Web 服务器为 HTTP 请求服务。如果它请求的是一个静态页面，那么处理就结束了（Web 服务器只要用这个页面进行响应即可），但这种情形并不多见。大多数页面在某种程度上都是动态的，而在这种情况下，请求将被传递给应用程序服务器。直到 WebSphere 4.0 版本之前，这一动作都是使用一些专利的方法（例如本地管道或通过套接字实现的 OSE（Open Servlet Engine，开放 Servlet 引擎）专利协议）完成的。在 WebSphere 4.0 中，请求通过 HTTP 传递（或通过 HTTPS 传递（如果需要加密的通信通道）；甚至在 DMZ（Demilitarized Zone，隔离区）中传递，4.1.2 节中将讨论 DMZ）。

一旦到达了应用程序服务器，就将调入应用程序代码并执行应用程序逻辑从而执行动作与事务。典型地，应用程序服务器代码也将使用某种形式的标记语言创建将要被传递给浏览器的下一个表示。为了执行这两个任务，应用程序代码需要访问数据库。这是通过与数据库服务器通信来完成的。由于大多数数据库服务器都是关系数据库，因此要使用 JDBC 驱动程序，并且，取决于 JDBC 连接的类型，调用或者通过一个网关直接在 Java 中被处理，或者通过本地库处理。无论哪种情况，应用程序服务器与数据库服务器之间的通信都是取决于数据库厂商的。例如，如果正在访问的数据库是 Oracle，那么在网络上的通信将是 SQLNET。其中，应用程序服务器可能使用标准 API 直接与 JDBC 驱动程序通信（然后由驱动程序负责管理与数据库服务器的通信）。

在这种上下文中，有必要提到 WebSphere Connection Manager。WebSphere 的另一个优点就是提供了对关系数据库的透明访问（只要有 JDBC 驱动程序）。正如在第 18 章见到的那样，从应用程序的角度看，Connection Manager 提供了比直接数据库调用更好的性能。

41.1.1 两服务器拓扑

图 41-1 描述的拓扑仅仅是概念性的。在本章后面将会见到，它是不精确的，因为所描述的

组件可能会产生许多同时运行的线程与进程，从而提高吞吐率和降低响应时间。此外，它还没有精确地定义这些元素应该如何分布在物理硬件上。下面将讨论这个问题。

可以想象的最简单拓扑如图 41-2 所示（读者甚至可能会想到将所有的组件放置在一个单独服务器上的情况，但应该记住的是，永远不要将数据库服务器与应用程序服务器放置在相同的计算机上！）。这种两个服务器的拓扑意味着应用程序服务器与 Web 服务器运行在相同的机器上。这是一种合理的拓扑，并且在开发系统中几乎总是使用这种拓扑。它还有一个优点，就是具有较小的延迟，因为在应用程序服务器与 Web 服务器之间没有网络开销。在早期版本的 WebSphere 中，这种拓扑被广泛地采用，实际上，在 3.0 版本之前，这是惟一可能的拓扑，因为此前的 WebSphere 并不支持将 Web 服务器从应用程序服务器分离的情况。通过使用专利的 OSE 协议（IBM 基于套接字的专利协议），在 WebSphere 3.x 中可以支持这种分离，但在 WebSphere 4.0 中，使用 HTTP 与 HTTPS 连接方案取代了 3.x 中所使用的方法。这样就不仅允许实现安全的连接，而且使用了防火墙能够理解（这样就不必定义一套新的称为 WebSphere 的协议并且为其定义特殊的规则）的标准协议。

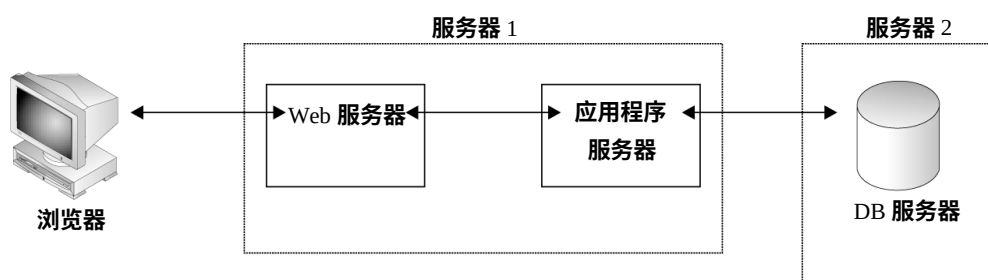


图 41-2 两服务器拓扑

41.1.2 三服务器拓扑

虽然两服务器拓扑是合理的，但使用三服务器拓扑通常更好。图 41-3 显示了三服务器拓扑。

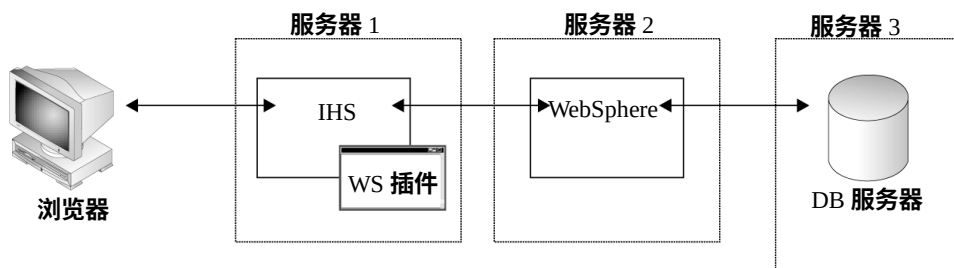
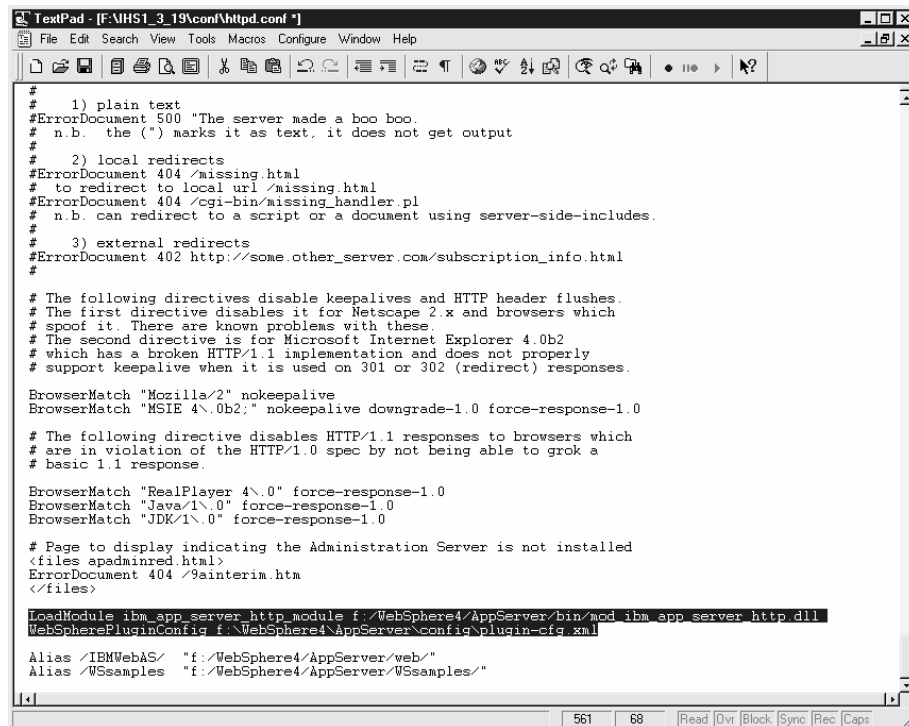


图 41-3 三服务器拓扑

在这种拓扑中，Web 服务器运行在一台物理服务器上，并且 WebSphere 应用程序 JVM 运行在

另一台物理服务器上。为了使这种拓扑能够工作，必须在 WWW 运行时加载 HTTP 插件。在大多数部署中，使用的 Web 服务器都是 Apache 服务器，IBM 公司将这个服务器再次注册为 HIS (IBM HTTP Server)。在这种情况下，WebSphere 客户被打包为一个 Apache 模块并且被加载到 Apache 运行时环境中，如图 41-4 所示，在这个图中也显示了对 XML 配置文件的引用，本章将在“设置一个三服务器的配置”一节中对这个配置文件进行详细描述。



```
#
# 1) plain text
#ErrorDocument 500 "The server made a boo boo."
# n.b. the (') marks it as text, it does not get output
#
# 2) local redirects
#ErrorDocument 404 /missing.html
# to redirect to local url /missing.html
#ErrorDocument 404 /cgi-bin/missing_handler.pl
# n.b. can redirect to a script or a document using server-side-includes.
#
# 3) external redirects
#ErrorDocument 402 http://some.other_server.com/subscription_info.html
#
# The following directives disable keepalives and HTTP header flushes.
# The first directive disables it for Netscape 2.x and browsers which
# spoof it. There are known problems with these.
# The second directive is for Microsoft Internet Explorer 4.0b2
# which has a broken HTTP/1.1 implementation and does not properly
# support keepalive when it is used on 301 or 302 (redirect) responses.
BrowserMatch "Mozilla/2" nokeepalive
BrowserMatch "MSIE 4\..0b2;" nokeepalive downgrade-1.0 force-response-1.0
#
# The following directive disables HTTP/1.1 responses to browsers which
# are in violation of the HTTP/1.0 spec by not being able to grok a
# basic 1.1 response.
BrowserMatch "RealPlayer 4\..0" force-response-1.0
BrowserMatch "Java/1\..0" force-response-1.0
BrowserMatch "JDK/1\..0" force-response-1.0
#
# Page to display indicating the Administration Server is not installed
<files apadmainred.html>
ErrorDocument 404 /9ainterim.htm
</files>
LoadModule ibm_app_server_http_module f:/WebSphere4/AppServer/bin/mod_ibm_app_server_http.dll
WebSpherePluginConfig f:/WebSphere4/AppServer/config/plugin-cfg.xml
Alias /IBMWebAS/ "f:/WebSphere4/AppServer/web/"
Alias /WSsamples "f:/WebSphere4/AppServer/WSsamples/"
```

图 41-4 加载 Apache httpd.conf 中的 WebSphere 客户

正如前面说过的那样，三服务器拓扑优于两服务器拓扑。其原因如下：有时，特别是当 WebSphere 运行于 SMP (Symmetric Multiprocessing, 对称多进程) 体系结构下时，可能有几个应用程序服务器的实例以及几个 WebSphere 服务器的实例并发地运行。这种方案的优点就是使用更少的硬件、更容易的配置与更好的性能，缺点就是降低了安全性。实际上，有许多原因可以说明三服务器拓扑比两服务器拓扑更好。其中一个原因就是，从非常简单的情况说（可能这还不同于直觉），三服务器拓扑比两服务器拓扑执行更好。Web 服务器与 JVM 之间的通信是很快的，并且，为 JVM 提供一个单独的物理服务器对于整体的性能至关重要（但不幸的是，JVM 仍然比任何其他组件慢很多）。此外，三服务器的配置在应用冗余方面具有更大的灵活性，这一点将在本章后面的 41.2 节“高可用性”见到。但这两种拓扑之间最重要的不同，在于三服务器拓扑能够非常自然地支持 DMZ 配置。

DMZ 配置

DMZ 是一个概念，它定义了如何在包括有防火墙的环境中部署应用程序，并定义了如何确保抵御外部入侵者的安全性。公司所部署的 Internet 应用程序及电子商务应用程序是相对较新的。当这种应用程序第一次出现时，显然，其安全性是一个必须立刻解决的问题。目前，防火墙解决方案已经被用于保护内部网络，并且防火墙很适合电子商务部署模型。

在部署 Internet 应用程序时，公司通常在 Web 服务器之前放置一个防火墙，并打开端口 80（HTTP 的默认端口）用于为所有的客户机器提供访问。这就使所有用户（无论它们在 Internet 上的位置是什么）都能访问应用程序，而与此同时又维持（相对）内部网络仍然受保护。但不幸的是，黑客在进行破坏时总是表现出丰富的创造力与动力。人们很快就发现，虽然从理论上讲防火墙能够阻止除了端口 80 之外的任何破坏，但由于在底层结构软件中存在 bug（例如在 Microsoft IIS 中就发现了许多 bug）以及在应用程序中存在 bug，使得黑客总是能够找到入侵途径。这些安全性漏洞导致的后果可能是不具破坏性的“我在这里”类型的说明，也可能是极具破坏性的“格式化硬盘”类型的程序，或者更坏的情况是，所有关键的、保密的业务数据都被窃。这样，就诞生了第二个防火墙以及 DMZ 的概念。

为了理解这个概念，首先需要理解防火墙的作用，以及除了检查名称之外还有没有更好的方法。物理世界中的防火墙（例如一栋建筑物）主要是用来防止火灾的蔓延。这也是网络部署中防火墙软件解决方案的主要作用。外部防火墙（放置在公共 Internet 与 Web 服务器之间的防火墙）主要用来防止来自 Internet 的“火势蔓延”（或者，在现在的例子中就是防止未授权个人的访问）。但是，如果“火”想尽办法力图攻击进来该怎么办呢？其答案就是放置另一个防火墙来隔离“火”。这也就是 DMZ 概念的重要性。

图 41-5 显示了一个 DMZ 方案。第二个防火墙放置在 Web 服务器与内部网络之间，用来确保如果 Web 服务器受到安全性攻击，未授权访问这种“火”不会蔓延到整个内部网络。如果确实出现对 Web 服务器的这种访问，那么只有这个服务器的安全性受到伤害。第二个防火墙通常具有非常严格的规则库。例如，外部防火墙可能定义只要使用端口 80 或端口 443（对于安全连接），所有的主机都能够访问这个 Web 服务器，但第二个防火墙则仅仅允许对 Web 服务器主机并且在特定端口上的访问。

那么，与两服务器拓扑相比，三服务器拓扑有没有什么缺点呢？缺点当然存在，实际上，生活中没有任何事情是完全绝对的。其缺点就是整个部署更加复杂，并且需要为这个工程使用更多的物理硬件。但是，由于 Web 服务器处理过程非常高效，因此它们在硬件方面并不需要很大的投资，实际上，许多网络工程师（包括作者）都认为，三服务器拓扑在硬件上需要的投资并不是很多，因为可以使用一台并不昂贵的服务器用于 Web 服务器，因此仅仅需要为运行 WebSphere JVM 的服务器投资即可。因此，两服务器拓扑惟一的优点就是复杂性似乎更低——与三服务器拓扑所具有的显著优点相比，这是微不足道的。

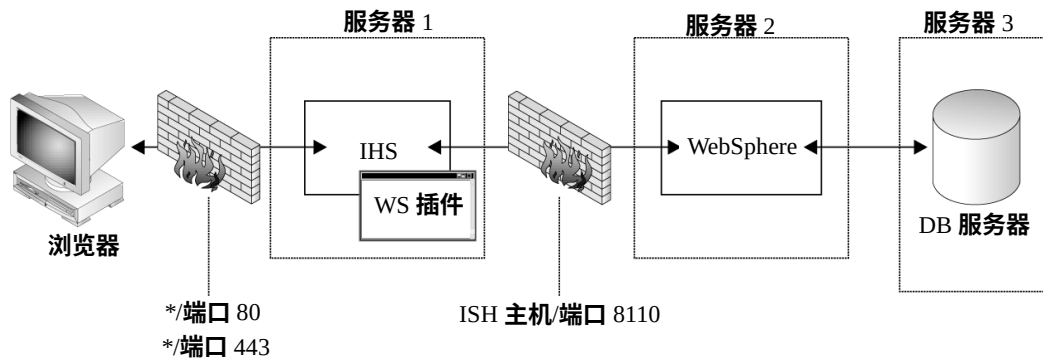


图 41-5 三服务器拓扑与 DMZ

设置一个三服务器的配置

图 41-6 显示了三服务器拓扑所需的配置。其中，Web 服务器放置在 DMZ 中，并且引用 plugin-cfg.xml 文件，如图 41-4 所示，此文件在.../WebSphere/AppServer/config 目录下。这个文件用来告知 HTTP 插件，WebSphere 服务器在什么地方以及应该与这个服务器的哪个端口进行通信。当然，在这个文件中定义的端口必须与被允许通过内部防火墙的端口相同。在这个文件中也包含了路由信息与 servlet URI 映射，这也进一步说明了为什么在每次安装新的企业应用程序时都必须重新生成插件信息。这个文件替换在 WebSphere 3.x 中存在的 queues.properties, vhosts.properties, rules.properties 文件（这些文件被很具有策略性地放置在 WebSphere/AppServer/temp 目录中）。这个文件的内容如下：

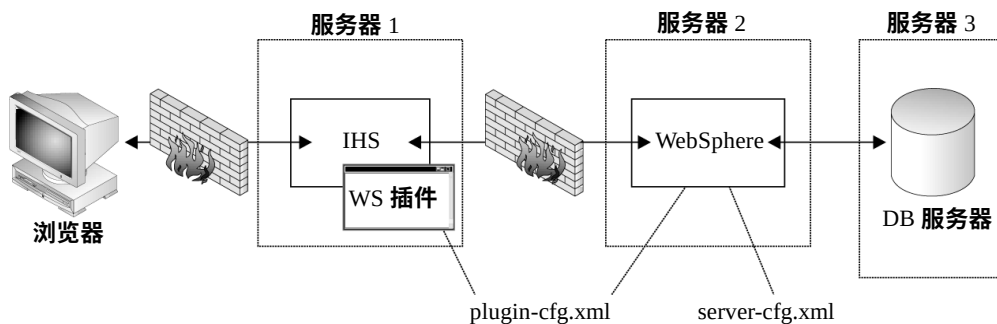


图 41-6 三服务器拓扑的属性文件

```
<Config>
  <!-- The LogLevel controls the amount of information that
        is written to the plug-in log file. Possible values
        are Error, Warn, and Trace. -->
  <Log Name="f:/WebSphere4/AppServer/logs/native.log" LogLevel="Error"/>

  <!-- Server groups provide a mechanism of
```

```

        grouping servers together. -->
<ServerGroup Name="default_group">
    <Server CloneID="vn1" Name="default_server">
        <!-- The transport defines the hostname and port
            value that the Web server
            plug-in will use to communicate
            with the application server. -- >
        <Transport Hostname="washost" Port="8110" Protocol="http"/>
    </Server>
</ServerGroup>

<!-- Virtual host groups provide a mechanism of
    grouping virtual hosts together. -->
<VirtualHostGroup Name="default_host">
    <VirtualHost Name="*:*/>
</VirtualHostGroup>

<!-- URI groups provide a mechanism for grouping URIs together. Only
    the context root of a Web application needs to be specified unless
    you want to restrict the request URIs that are passed to the
    application server. -->
<UriGroup Name="default_host_URIs">
    <Uri Name="/servlet/*"/>
    <Uri Name="/webapp/examples"/>
    <Uri Name="*.jsp"/>
    <Uri Name="/ErrorReporter"/>
    <Uri Name="/j_security_check"/>
    <Uri Name="/tradetheme"/>
    <Uri Name="/theme"/>
    <Uri Name="/WebSphereSamples/*"/>
</UriGroup>

<!-- A route ties together each of the above components. -->
<Route ServerGroup="default_group" UriGroup="default_host_URIs"
    VirtualHostGroup="default_host"/>
</Config>

```

plugin-cfg.xml 文件被 Web 服务器所使用。应用程序服务器使用的主要文件是 server-cfg.xml；在 WebSphere 4.0 安装中的默认文件在下面的清单中列出。需要说明的是，这个文件将所有的配置属性一致地集成到一起，并且使用清晰地打包、具有自描述能力的格式表示这个文件。这个文件中包括了 queues.properties, vhosts.properties, rules.properties 文件以及许多其他属性文件中存在的信息。最后，如果建立的是一个安全传输（HTTPS），那么在这个插件

中将包含每个应用程序服务器对应的关键字与证书信息。

```
<applicationserver:Domain xmi:version="2.0" xmlns:xmi="http://www.omg
.org/XMI"xmlns:applicationserver="applicationserver.xmi" xmlns:security=
"security.xmi"xmlns:resources="resources.xmi" xmlns:server="server.xmi"
xmi:id="Domain_1" name="WebSphere Administrative Domain">
  <virtualHosts xmi:id="VirtualHost_1" name="default_host">
    <aliases xmi:id="HostAlias_1" hostname="*" port="8110"/>
    <aliases xmi:id="HostAlias_2" hostname="*" port="9443"/>
    <aliases xmi:id="HostAlias_3" hostname="*" port="80"/>
    <aliases xmi:id="HostAlias_4" hostname="*" port="443"/>
    <defaultMimeEntries xmi:id="MimeEntry_1" type="image">
      <extensions>gif</extensions>
    </defaultMimeEntries>
    <defaultMimeEntries xmi:id="MimeEntry_2" type="text/plain">
      <extensions>txt</extensions>
      <extensions>java</extensions>
      <extensions>wsdl</extensions>
    </defaultMimeEntries>
    <defaultMimeEntries xmi:id="MimeEntry_3" type="text/html">
      <extensions>htm</extensions>
      <extensions>html</extensions>
    </defaultMimeEntries>
    <defaultMimeEntries xmi:id="MimeEntry_4" type="x-application">
      <extensions>exe</extensions>
    </defaultMimeEntries>
    <defaultMimeEntries xmi:id="MimeEntry_5" type="x-application/zip">
      <extensions>zip</extensions>
    </defaultMimeEntries>
    <defaultMimeEntries xmi:id="MimeEntry_6" type="application/pdf">
      <extensions>pdf</extensions>
    </defaultMimeEntries>
  </virtualHosts>
  <virtualHosts xmi:id="VirtualHost_2" name="admin_host">
    <aliases xmi:id="HostAlias_5" hostname="*" port="9090"/>
  </virtualHosts>
  <security xmi:id="Security_1" cacheTimeout="30000" enabled="false"
  activeAuthMechanism="LocalOSAuthentication_1">
    <authMechanisms xmi:type="security:LocalOSAuthentication"
    xmi:id="LocalOSAuthentication_1">
      <userRegistry xmi:type="security:LocalOSUserRegistry"
      xmi:id="LocalOSUserRegistry_1" serverId="demouser"
      serverPassword="{xor}0zoyMC8+LCwoOw==" name="LOCALOS">
        <properties xmi:id="UserRegProperty_1" name="myCustomProperty">
```

```

        value="myCustomValue"/>
    </userRegistry>
</authMechanisms>
<defaultSSLSettings xmi:id="SecureSocketLayer_1"
    keyFileName="{WAS_ROOT}/etc/DummyKeyring.jks"
    keyFilePassword="{xor}CDo9Hgw="
    keyFileFormat="JKS" trustFileName="{WAS_ROOT}/etc/DummyKeyring.jks"
    trustFilePassword="{xor}CDo9Hgw=" clientAuthentication="false"
    securityLevel="HIGH" enableCryptoHardwareSupport="false">
    <cryptoHardware xmi:id="CryptoHardwareToken_1"
        tokenType="PKCS#11" libraryFile="" password="{xor}"/>
    </defaultSSLSettings>
</security>
<resourceProviders xmi:type="resources:JDBCDriver" xmi:id="JDBCDriver_1"
    name="Db2JdbcDriver" description="DB2 JDBC Driver"
    implementationClassName="COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource"
    urlPrefix="jdbc:db2">
    <factories xmi:type="resources:DataSource" xmi:id="DataSource_1" name=
    "Session Persistence datasource" jndiName="jdbc/Session" description="Data
    source for session persistence" category="Samples" jtaEnabled="false"
    databaseName="Session" minimumPoolSize="1" maximumPoolSize="30" connect-
    ionTimeout="1000" idleTimeout="2000" orphanTimeout="3000">
        <propertySet xmi:id="DataSource_1_ps"/>
    </factories>
    <propertySet xmi:id="JDBCDriver_1_ps"/>
</resourceProviders>
<resourceProviders xmi:type="resources:JDBCDriver" xmi:id="JDBCDriver_2"
    name="idbJdbcDriver" description="instantDB JDBC Driver"
    implementationClassName="com.ibm.ejs.cm.portability.IDBConnectionPoolDat-
    aSource" urlPrefix="jdbc:idb">
    <factories xmi:type="resources:DataSource" xmi:id="DataSource_2"
    name="Increment Bean Datasource" jndiName="jdbc/SampleDataSource"
    description="Increment Bean Datasource" category="Samples1" jtaEnabled=
    "false" minimumPoolSize="1" maximumPoolSize="10" connectionTimeout="1000"
    idleTimeout="2000" orphanTimeout="3000">
        <propertySet xmi:id="DataSource_2_ps">
            <resourceProperties xmi:id="J2EEResourceProperty_1" name="url"
            type="java.lang.string" value="jdbc:idb:f:\WebSphere4\AppServer/bin/
            sampleDB.prp"/>
        </propertySet>
    </factories>
    <factories xmi:type="resources:DataSource" xmi:id="DataSource_3" name=
    "Samples IDB Datasource" jndiName="jdbc/sample" description="Samples
    Gallery IDB Datasource" category="Samples" jtaEnabled="false" minimum-

```

```

PoolSize="1" maximumPoolSize="10" connectionTimeout="1000" idleTimeout=
"2000" orphanTimeout="3000">
  <propertySet xmi:id="DataSource_3_ps">
    <resourceProperties xmi:id="J2EEResourceProperty_2" name="url"
type="java.lang.string"
value="jdbc:db:f:\WebSphere4\AppServer\installedApps\Samples.ear/IDBDat
abase/>
sample.prp"/>
  </propertySet>
</factories>
  <propertySet xmi:id="JDBCdriver_2_ps"/>
</resourceProviders>
  <resourceProviders xmi:type="resources:URLProvider" xmi:id="URLP
rovider_1" name="Default URL Provider" description="The default internal URL
Provider that can be used to create URL Factories. This provider's
StreamHandler class name and protocol must be set to 'unused' in order for
it to be valid" streamHandlerClassName="unused" protocol="unused">
  <propertySet xmi:id="URLProvider_1_ps"/>
</resourceProviders>
  <resourceProviders xmi:type="resources:MailProvider" xmi:id="MailPro-
vider_1" name="Default Mail Provider" description="The default internal Mail
Provider that can be used to create Mail Sessions">
  <propertySet xmi:id="MailProvider_1_ps"/>
</resourceProviders>
  <nodes xmi:id="Node_1" name="ron2k">
    <servers xmi:type="applicationserver:ApplicationServer"
xmi:id="ApplicationServer_1" desiredExecutionState="START" name="Default
Server" id="-1" moduleVisibility="COMPATIBILITY">
      <processDefinition xmi:type="server:JavaProcessDef" xmi:id="Proc-
essDef_1" executableName="{JAVA_HOME}/bin/java" commandLineArguments=""
workingDirectory="{WAS_ROOT}/bin" executableTargetKind="JAVA_CLASS"
executableTarget="com.ibm.ws.bootstrap.WSLauncher">
        <executionSettings xmi:id="ProcessExecution_1" umask="777"
runAsUser="root"/>
        <ioRedirect xmi:id="OutputRedirect_1"
stdoutFilename="{LOG_ROOT}/default_server_stdout.log"
stderrFilename="{LOG_ROOT}/default_server_stderr.log"/>
        <jvmSettings xmi:id="JavaVirtualMachine_1"
classpath="{WAS_ROOT}/lib/bootstrap.jar;{WAS_ROOT}/properties"
bootClasspath=""
verboseModeClass="false" verboseModeGarbageCollection="false"
verboseModeJNI="false" initialHeapSize="4" maximumHeapSize="256"
runHProf="false" hprofArguments="" debugMode="false" debugArgs=""
genericCommandLineArgs="com.ibm.ws.runtime.StandardServer"

```

```

disableJIT="false">
    <systemProperties xmi:id="SystemProperty_1" name="server.root"
value="${WAS_ROOT}"/>
    <systemProperties xmi:id="SystemProperty_2" name="ws.ext.dirs"
value="${JAVA_HOME}/lib;${WAS_ROOT}/classes;${WAS_ROOT}/lib;${WAS_ROOT}/
lib/ext;
${WAS_ROOT}/web/help;${WAS_ROOT}/properties"/>
    <systemProperties xmi:id="SystemProperty_3"
name="com.ibm.CORBA.ConfigURL"
value="file:${WAS_ROOT}/properties/sas.server.props"/>
    </jvmSettings>
    </processDefinition>
    <performanceMonitoring xmi:id="PerformanceMonitor_1" enable="false"
specification=""/>
    <objectLevelTraceSettings xmi:id="ObjectLevelTrace_1" enable="false"
hostname="localhost" port="2102" debug="false" sourcePath=""/>
    <locationServiceDaemon xmi:id="LocationServiceDaemon_1" hostname=
"localhost"port="9000" mode="NONE"/>
    <transactionService xmi:id="TransactionService_1"
transactionLogFile="${TRANLOG_ROOT}/tran1.log,${TRANLOG_ROOT}/tran2.log"
totalTranLifetimeTimeout="60000" clientInactivityTimeout="30000"/>
    <traceService xmi:id="TraceServiceConfig_1" enable="true"
traceSpecification="*=all=disabled" traceOutputFilename="stdout"
diagThreadPort="7000"/>
    <namingServiceSettings xmi:id="NamingServiceProvider_1" enable=
"true"providerHost="localhost" providerPort="9070"
providerClassname="com.ibm.ejs.naming.transient.NamingServiceProvider"
enablePersistentNaming="true"/>
    <orbSettings xmi:id="ORBConfig_1" enable="true" bootstrapHost=
"localhost" bootstrapPort="900">
    <threadPool xmi:id="ThreadPool_2" minimumSize="10" maximumSize="50"
inactivityTimeout="3500" isGrowable="false"/>
    </orbSettings>
    <webContainer xmi:id="WebContainer_1"
installedWebModules="WebModuleRef_1 WebModuleRef_2 WebModuleRef_3
WebModuleRef_4 WebModuleRef_5 WebModuleRef_6">
    <transports xmi:type="applicationserver:HTTPTransport"
xmi:id="HttpTransport_1" hostname="*" port="9080"/>
    <transports xmi:type="applicationserver:HTTPTransport"
xmi:id="HttpTransport_2" hostname="*" port="9443" sslEnabled="true"/>
    <transports xmi:type="applicationserver:HTTPTransport"
xmi:id="HttpTransport_3" hostname="*" port="9090" external="false"/>
    <sessionManager xmi:id="SessionManager_1" enableUrlRewriting=
"false"enableCookies="true" enableSSLTracking="false"

```

```

enableProtocolSwitchRewriting="false" enablePersistentSessions="false"
enableSecurityIntegration="false">
    <defaultCookieSettings xmi:id="Cookie_1" domain="" maximumAge="-1"
path="/" secure="false"/>
    <persistentSessions xmi:id="SessionPersistence_1"
datasourceJNDIName="jdbc/Session" userId="db2admin"
password="{xor}0z1tPjsyNjF7dxoRHBAbGht2" db2RowSize="ROW_SIZE_4KB"
tableName=""/>
    <tuningParams xmi:id="TuningParams_1" usingMultiRowSchema="false"
allowOverflow="true" scheduleInvalidation="false"
writeFrequency="END_OF_SERVLET_SERVICE" writeInterval="120"
writeContents="ONLY_UPDATED_ATTRIBUTES" invalidationTimeout="30">
        <invalidationSchedule xmi:id="InvalidationSchedule_1"
firstHour="14" secondHour="2"/>
    </tuningParams>
</sessionManager>
    <dynamicCache xmi:id="DynamicCache_1" enable="false"
cacheSize="1000" defaultPriority="1"/>
    <threadPool xmi:id="ThreadPool_1" minimumSize="10" maximumSize="50"
inactivityTimeout="10" isGrowable="false"/>
</webContainer>
    <ejbContainer xmi:id="EJBContainer_1"
passivationDirectory="{WAS_ROOT}/temp" inactivePoolCleanupInterval="30000"
installedEJBModules="EJBModuleRef_2 EJBModuleRef_1 EJBModuleRef_3
EJBModuleRef_4 EJBModuleRef_5" defaultDatasource="DataSource_1">
        <cacheSettings xmi:id="EJBCache_1" cleanupInterval="1000"
cacheSize="2000"/>
    </ejbContainer>
    <serverSecurityConfig xmi:id="ServerSecurityConfig_1"
useDomainQualifiedUserNames="false"/>
</servers>
    <installedResourceProviders xmi:id="ResourceProviderRef_1"
classpath="" resourceProvider="JDBCdriver_1"/>
    <installedResourceProviders xmi:id="ResourceProviderRef_2"
classpath="{WAS_ROOT}/lib/idb.jar" resourceProvider="JDBCdriver_2"/>
    <installedApps xmi:id="ApplicationRef_1" name="sampleApp"
archiveURL="{APP_INSTALL_ROOT}/sampleApp.ear">
        <modules xmi:type="applicationserver:WebModuleRef"
xmi:id="WebModuleRef_1" uri="default_app.war"/>
        <modules xmi:type="applicationserver:WebModuleRef"
xmi:id="WebModuleRef_2" uri="examples.war"/>
        <modules xmi:type="applicationserver:EJBModuleRef"
xmi:id="EJBModuleRef_2" uri="Increment.jar"/>
    </installedApps>

```

```

    <installedApps xmi:id="ApplicationRef_2" name="Server Administration
Application" archiveURL="${APP_INSTALL_ROOT}/admin.ear">
      <modules xmi:type="applicationserver:WebModuleRef"
xmi:id="WebModuleRef_3" uri="admin.war"/>
    </installedApps>
    <installedApps xmi:id="ApplicationRef_3" name="WebSphere Application
Server Samples" archiveURL="${APP_INSTALL_ROOT}\Samples.ear">
      <modules xmi:type="applicationserver:WebModuleRef"
xmi:id="WebModuleRef_4" uri="YourCo.war"/>
      <modules xmi:type="applicationserver:WebModuleRef"
xmi:id="WebModuleRef_5" uri="Samples.war"/>
      <modules xmi:type="applicationserver:WebModuleRef"
xmi:id="WebModuleRef_6" uri="theme.war"/>
      <modules xmi:type="applicationserver:EJBModuleRef"
xmi:id="EJBModuleRef_1" uri="TimeoutEJBean.jar"/>
      <modules xmi:type="applicationserver:EJBModuleRef"
xmi:id="EJBModuleRef_3" uri="IncrementEJBean.jar"/>
      <modules xmi:type="applicationserver:EJBModuleRef"
xmi:id="EJBModuleRef_4" uri="HelloEJBean.jar"/>
      <modules xmi:type="applicationserver:EJBModuleRef"
xmi:id="EJBModuleRef_5" uri="AccountAndTransferEJBean.jar"/>
    </installedApps>
    <pathMap xmi:id="PathMap_1">
      <entries xmi:id="PathMapEntry_1" symbolicName="APP_INSTALL_ROOT"
path="${WAS_ROOT}/installedApps" description="The filesystem path to
the directory that will contain installed enterprise applications."/>
      <entries xmi:id="PathMapEntry_2" symbolicName="LOG_ROOT"
path="${WAS_ROOT}/logs" description="The filesystem path to the directory
that will contain server log files."/>
      <entries xmi:id="PathMapEntry_3" symbolicName="TRANLOG_ROOT"
path="${WAS_ROOT}/tranlog" description="The filesystem path to the directory
that will transaction log files."/>
      <entries xmi:id="PathMapEntry_4" symbolicName="PRODUCT_INSTALL_ROOT"
path="${WAS_ROOT}" description="The filesystem path to the product
installation directory"/>
      <entries xmi:id="PathMapEntry_5" symbolicName="WAS_ROOT"
path="f:\WebSphere4\AppServer" description="The filesystem path to the
product installation directory"/>
      <entries xmi:id="PathMapEntry_6" symbolicName="JAVA_HOME"
path="${WAS_ROOT}/java" description="The filesystem path to the product Java
home directory"/>
    </pathMap>
  </nodes>
</applicationserver:Domain>

```


41.2 高 可 用 性

现在，假设在网络上将使用 3 个物理服务器，但这样就足够了吗？实际上，这很可能还是不够的。如果不包括需要满足高性能需求的应用程序，或者不包括对容错性有需求的大应用程序，那么使用 3 个物理服务器可能是足够了。但是，如果确实使用具有上述要求的应用程序，那么还需要进行更大的投资。Java 仍然是一个相对较慢的运行环境，并且部署在 WebSphere 上的应用程序通常为大量的用户提供服务。此外，还可能面对 24 小时不间断的在线需求。所有这些需求都要求在部署中存在冗余 (redundancy)，这意味着要求有比最低需求更多的硬件。冗余用来在负载高峰时通过负载均衡机制将负载分布到所有部署的服务器上，此外，冗余还用于支持容错（如果某些服务器被关闭，其余的服务器仍然可用于处理请求）。

关于 WebSphere 4.0 的失败恢复，建议访问 <http://www7.software.ibm.com/vadd-bin/ftpd1?1/vadc/wsdd/pdf/modjeski.pdf> 上的白皮书。

41.2.1 纵向伸缩与横向伸缩

图 41-7 显示了两种主要形式的伸缩。纵向伸缩 (vertical scaling) 用来描述多个 JVM 放置在一个单独的物理服务器上的配置。服务器可能有一个或多个 CPU。JVM 支持一个线程模型 (threading model)。根据 JVM 的不同，线程模型可能使用本地操作系统的线程，也可能在 JVM 中模拟线程（被模拟的线程有时称为绿线程 (green thread) 或轻进程 (lightweight process)）。如果使用的是操作系统线程，那么即使单个 JVM 都可能耗尽服务器所有的资源。在 3.x 版本（由于 JDK 1.1 存在的可伸缩性问题）中，情况并非总是如此；如果有一个具有 4CPU 的机器，那么使用 4 个 JVM 运行相同的代码行远远比使用一个 JVM 要好得多。但在 4.0 版本中，这个问题已经不再存在，并且作为一条一般的原则，就是使用越少的 JVM 越好。

例如，在一个 4 路的机器中，不应该让相同的应用程序运行于多于 2 个的 JVM 上。使用 2 个 JVM 可用于处理某些进程失败恢复的情况。如果不是为了容错的需要，一个单独的 JVM 就能够完全利用 4 路的机器，甚至在大多数情况下可以使用 8 路。最新的 JDK（也就是 JDK 1.3x）需要大的内存模型支持，并且不存在先前 JVM 版本中出现的那些问题。

在 3.x 版本中，使用图 41-8 所示的部署方案能够获得最佳的性能与容错性。在这种方案下，有 2 个物理服务器分别运行 2 个 WebSphere JVM。其中，物理服务器通常是 2CPU 的机器。常用的规则是，每个 CPU 运行一个 JVM（这里假设服务器仅仅被用于 WebSphere，并且不进行任何其他进程）。

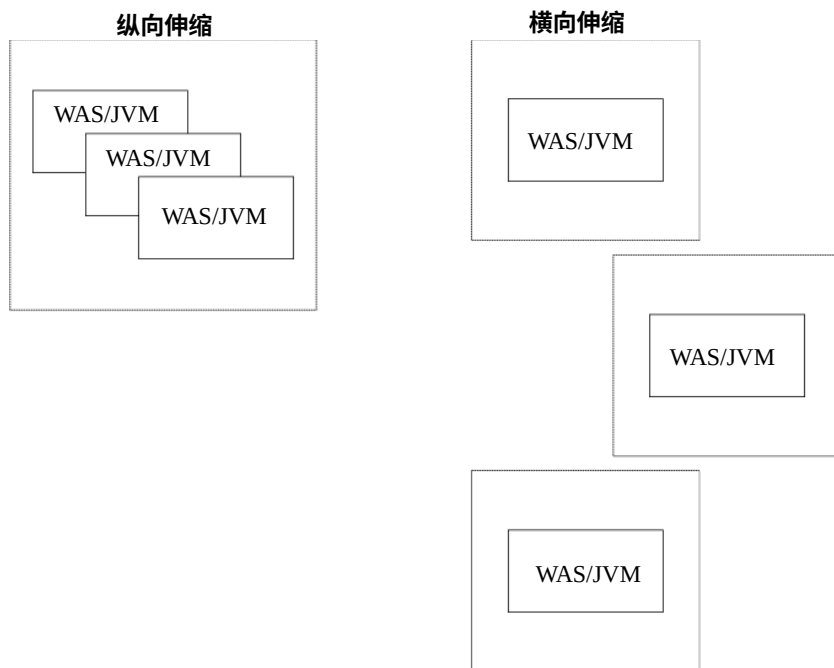


图 41-7 纵向伸缩与横向伸缩

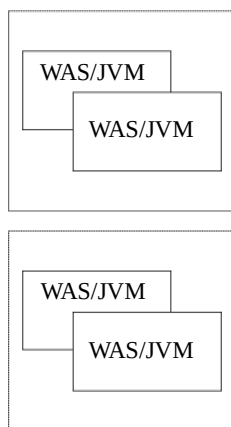


图 41-8 WebSphere 3.x 与 4.x 的优选伸缩

在 4.0 版本中，最佳的配置比较简单，并且常见的规则是使用越少的 JVM 越好。但为了高可用性（当然不是为了性能），应该使用多于一个的 JVM。值得庆幸的是，这种配置的构建与维护都更加简单。

41.2.2 设置 Web 服务器

那么如何部署 Web 服务器呢？使用多个 JVM 来执行 EJB 与 JSP 处理是一件好事情，但也需要有一个 Web 服务器处理来自浏览器的 HTTP 会话。图 41-9 显示了多 JVM 与一个 Web 服务器组合的最简单部署。图中，所有的克隆都由一个单独的 Web 服务器访问。

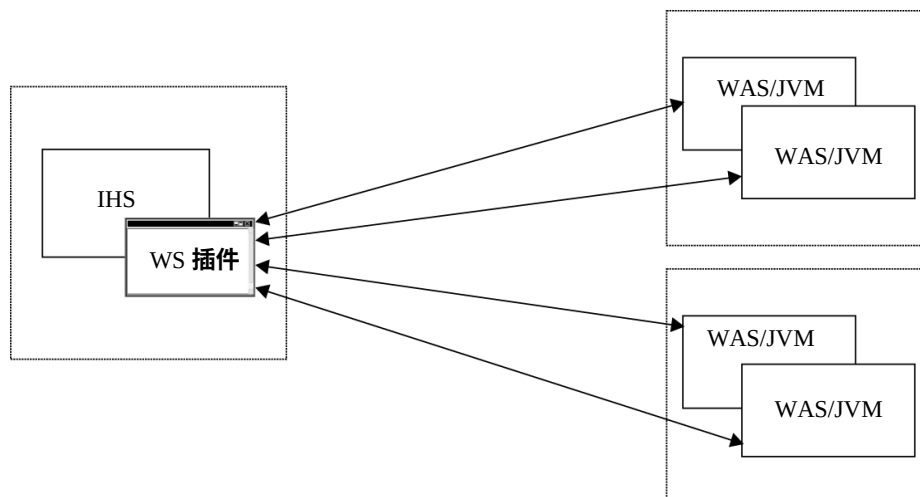


图 41-9 一个单独的 Web 服务器对应多个克隆

为了支持 4 个这样的克隆，必须配置 Web 服务器的 HTTP 插件，如以下代码所示。在这种情况下，有 2 个服务器分别运行 2 个独立的 JVM。

```

<Config>
  <!-- The LogLevel controls the amount of information that is
        written to the plug-in log file. Possible values are
        Error, Warn, and Trace. -- >
  <Log Name="f:/WebSphere4/AppServer/logs/native.log" LogLevel="Error"/>

  <!-- Server groups provide a mechanism of grouping servers together. -->
  <ServerGroup Name="default_group">
    <Server Name="washost1">
      <!-- The transport defines the hostname and port value that
            the web server plug-in will use to communicate with the
            application server. -- >
      <Transport Hostname="washost1" Port="8110" Protocol="http"/>
    </Server>
    <Server Name="washost2">
      <!-- The transport defines the hostname and port value that
  
```

```
        the web server plug-in will use to communicate with the
        application server. -->
        <Transport Hostname="washost2" Port="8120" Protocol="http"/>
    </Server>
</ServerGroup>

<!-- Virtual host groups provide a mechanism of grouping
      virtual hosts together. -->
<VirtualHostGroup Name="default_host">
    <VirtualHost Name="*:*/>
</VirtualHostGroup>

<!-- URI groups provide a mechanism for grouping URIs together. Only
      the context root of a Web application needs to be specified unless
      you want to restrict the request URIs that are passed to
      the application server. -->

<UriGroup Name="default_host_URIs">
    <Uri Name="/servlet/*"/>
    <Uri Name="/webapp/examples"/>
    <Uri Name="*.jsp"/>
    <Uri Name="/ErrorReporter"/>
    <Uri Name="/j_security_check"/>
    <Uri Name="/tradetheme"/>
    <Uri Name="/theme"/>
    <Uri Name="/WebSphereSamples/*"/>
</UriGroup>

<!-- A route ties together all of the above components. -->
<Route
    ServerGroup="default_group" UriGroup="default_host_URIs"
    VirtualHostGroup="default_host"/>
</Config>
```

WebSphere 4.0 支持一些很有用的属性，可以将这些属性添加到插件 XML 属性文件中，这些属性包括：

- **LoadBalance**：这个属性用来指定进行独立服务器调度的算法。所支持的算法包括循环算法（round-robin）或随机算法（random）。在循环算法中，HTTP 插件循环查询可用的 WebSphere JVM（跳过那些不能到达的 JVM）。在随机算法中，每条请求都被随机地发送给一个 JVM。这个属性是 XML 文件中 ServerGroup 元素的一部分，它也定义了重试间隔（插件第一次意识到服务器已经被关闭到插件下一次试图使用

这个服务器的时间之间的间隔)。

- **Transport** :这个属性指定了插件如何与 WebSphere 服务器通信 (HTTP 或 HTTPS), 并且指定了通信所用的端口号与服务器名。

除了这些新属性之外, plugin-cfg.xml 文件中还包括 WebSphere 3.x 中定义的各种属性文件 (queues.properties, rules.properties 与 vhosts.properties)。VirtualHost 元素允许定义希望服务器响应的目标虚拟主机名。这些属性也可以归组在 VirtualHostGroup 元素中。

插件信息不是静态的, 甚至在服务器正在运行时这些信息也可以改变。Config 元素 (XML 文件中的根元素) 有一个称为 RefreshInterval 的属性。这个属性的值定义了插件重新读取 plugin-cfg.xml 文件的时间间隔 (单位: 秒)。

先前版本的 WebSphere 存在的一个主要问题就是它们不能正确地处理会话仿射。在 4.0 版本中修复了这个问题, 并且提供了一个全新的实现——基于将 cloneID 添加到 sessionID cookie 中。会话仿射 (session affinity), 有时又称为会话持续性 (session persistence) 或粘性会话 (sticky session), 可用来确保如果某个浏览器会话已经被一个 JVM 所启动, 那么所有来自相同浏览器的请求将总是被插件转发到最初为这个请求提供服务的服务器上。在 WebSphere 3.5.2 之前, 会话仿射还不存在, 即使在 3.5.x 中, 会话仿射也并没有完全实现。最后, 在 WebSphere 4.0 中完全实现了这个功能并且允许进行部署, 如下列代码所示。

```
<?xml version="1.0"?>
<Config>
  ...
  <ServerGroup Name="ronhome/Default Server">
    <Server CloneID="tbajpkr1" Name="Default Server">
      <Transport Hostname="ronhome1" Port="9080" Protocol=
        "http"/></Server>
    <Server CloneID="tbajpkr2" Name="Default Server">
      <Transport Hostname="ronhome2" Port="9080" Protocol=
        "http"/></Server>
    </ServerGroup>
  ...
</Config>
```

为了进一步进行澄清, 我们强烈建议除非使用的是 WebSphere 4.0 或更新版本, 否则不应该依赖于 WS 客户提供负载均衡或容错支持。

需要说明的是, 虽然应该为 JVM 进行较大的硬件投资, 但并不需要过多关心 Web 服务器 (假设运行的是 Apache 或 IHS), 因为 Apache 在运行时速度很快。而且, 它的配置文件允许用户调整将被用于服务 HTTP 请求的进程数。

```
##
## httpd.conf -- Apache HTTP server configuration file
##
```

```
...

# Server-pool size regulation. Rather than making you guess how many
# server processes you need, Apache dynamically adapts to the load it
# sees --- that is, it tries to maintain enough server processes to
# handle the current load, plus a few spare servers to handle transient
# load spikes (e.g., multiple simultaneous requests from a single
# Netscape browser).

# It does this by periodically checking how many servers are waiting
# for a request. If there are fewer than MinSpareServers, it creates
# a new spare. If there are more than MaxSpareServers, some of the
# spares die off. These values are probably OK for most sites ---

MinSpareServers 5
MaxSpareServers 10

# Number of servers to start --- should be a reasonable ballpark figure.

StartServers 5

# Limit on total number of servers running, i.e., limit on the number
# of clients who can simultaneously connect. If this limit is ever
# reached, clients will be LOCKED OUT, so it should NOT BE SET TOO LOW.
# It is intended mainly as a brake to keep a runaway server from taking
# Unix with it as it spirals down...

MaxClients 150

# MaxRequestsPerChild: the number of requests each child process is
# allowed to process before the child dies.
# The child will exit so as to prevent problems after prolonged use when
# Apache (and maybe the libraries it uses) leaks. On most systems, this
# isn't really needed, but a few (such as Solaris) do have notable leaks
# in the libraries.

MaxRequestsPerChild 500

...
```

前面的代码清单显示了在 Solaris 部署中 IHS（或 Apache）上使用的一些重要参数。其中，StartServers 的值就是当服务器首次被初始化时将启动的 HTTPD（HTTP Daemon）进程数。在这里，如果调用

```
apachectl start
```

然后，在命令行输入

```
ps -ef | grep http
```

那么就将见到 5 个这样的进程。MinSpareServers 与 MaxSpareServers 被 Apache 运行时环境用来维护一个关于就绪状态的进程的池。

假设正在运行 WebSphere 4.0，用户可能会确实按照这里所描述的那样来部署系统。但不幸的是，Web 服务器就成了单点失败，因此这个系统并不是真正的容错系统（如果这个单独的服务器被关闭，那么整个系统将不能工作）。实际上，使用作为 WebSphere 一部分的软件，有许多可选的方案可用来实现负载平衡，但我们强烈推荐使用其中的一种方案。

41.2.3 消除单点失败

为了消除单点失败，最佳的方法就是创建两个完全克隆（不仅对 JVM 进行克隆，而且对 Web 服务器进行克隆）。图 41-10 显示了这种使用两套 IHS+WebSphere 的配置。但这样还不够。大多数情况下都不允许用户进行这样的选择：“使用 www1.mycompany.com，如果它不能工作，请使用 www2.mycompany.com”。应该让所有的用户都使用 www.mycompany.com，并且使得系统具有足够的智能，从而将用户重定向到正确的克隆上。

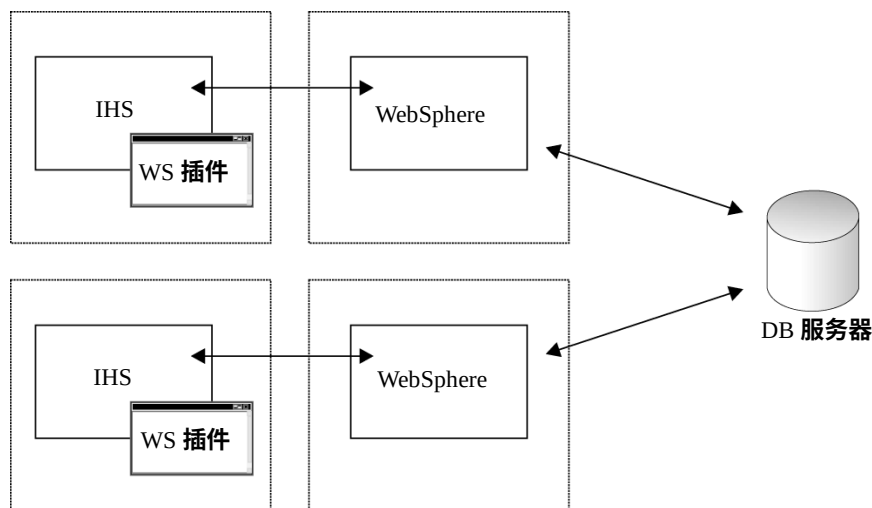


图 41-10 消除 Web 单点失败

在这种部署拓扑中所需的元素是路由器 (router)。在这个方案中，路由器通过将网络通信重定向到适当的服务器上，来实现负载平衡与容错功能。显然，如果想要应用程序正确地工作，就必须保留会话仿射功能，因此应该选择一个“智能路由器”（智能路由器就是能够被

配置，从而确保正确的网络通信重定向功能的路由器)。所有主要的路由器厂商都有支持这种功能的产品，我们并不是这一领域的专家，因此在这里不能向读者推荐某个厂商的产品。同时，这里需要讨论某些关于路由器的基本知识，并且将描述图 41-11 所示的拓扑，这个图中使用的是运行 WebOS 8.x 的 Alteon AD3 路由器。本书在许多部署示例中都使用这个路由器，它挺好用的。

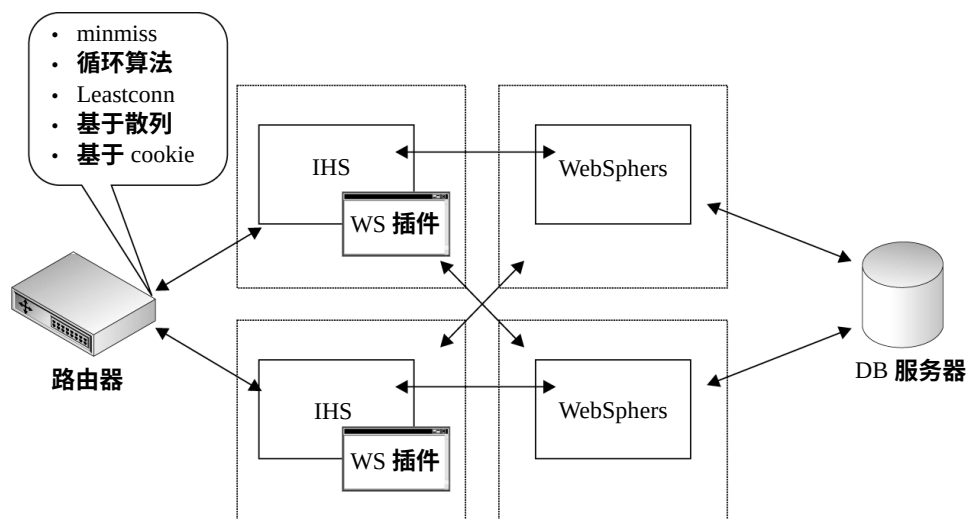


图 41-11 在拓扑中添加路由器

41.2.4 设置服务器负载均衡

Alteon 路由器的一个重要使用就是 SLB（Server Load Balancing，服务器负载均衡）。由于这个功能是如此重要，使得 Alteon 路由器支持多种 SLB 的部署度量。其中有下列算法：

- minmiss
- 循环算法
- Leastconn
- 散列

minmiss 将所有来自某个客户 IP 的网络通信指引到一个单独的服务器。这个算法确保了会话仿射，因为，它实际上确保了更强的功能：所有来自一个单独客户的请求总是到达相同的 Web 服务器。因此，即使在客户机上打开很少的浏览器，所有的请求（甚至那些没有共享一个会话的请求）都将被转发到相同的服务器。这个算法对于使用 cache（因此其名称为 minmiss）的应用程序能够更好地工作，并且在一般情况下，对于在线应用程序其工作

也很好（当然，对于某些特定应用程序，使用这个算法可能存在缺点）。

循环算法（round-robin）就是一个简单的循环模式，将每个请求转发到下一个服务器上。只有当在 Alteon 路由器上激活了会话持续性功能时，才能使用这个算法。

Leastconn 算法对每个服务器上已经打开的连接数进行跟踪，并且将请求转发到具有较小打开连接数的服务器上。如果两个克隆具有类似的处理能力并且运行相同的代码，那么这种算法的工作性能最好。同样，这种算法也只有在打开了会话持续性支持之后才能使用。

散列（hash）算法对正在被访问的 URL 与/或客户 IP 应用一个散列函数。除非这个散列函数应用于客户 IP（在这种情况下需要会话仿射，正如在 minmiss 中一样），否则这个方法也需要会话持续性。

为了支持使用 Alteon 路由器的会话仿射，需要设置它的会话持续性属性。由于正在使用标准的约定在 WebSphere 上运行应用程序，因此使用 jsessionid cookie 维护会话上下文。这个 cookie 由 Web 应用程序生成（通常在登录时生成），并且用来访问在其中维护了应用程序上下文的服务器方会话对象。例如，只要检查 HTTP 头部，就可以发现类似于以下形式的代码：

```
cookie: jsessionid= LBAZISTAAAAAFR4LOVUAAAA
```

为了将所有具有某个 cookie 值的 HTTP 网络通信指引到相同的服务器，需要确保 Alteon 路由器使用了相同的 cookie 名。利用下列 WebOS 命令可以很容易地实现这个要求：

```
# /cfg/slb/virt 1/service 80/pbind cookie passive "cookienam" jsessionid  
1 23 disable
```

这个配置命令很简单。它为虚拟系统 1（也就是/cfg/slb）配置服务器的负载平衡功能。它为 HTTP 网络通信（也就是 service 80）定义属性。并且它告知路由器在第 1 个字符与第 23 个字符之间使用名为 jsessionid 的 cookie 的值。任何 HTTP 请求，如果其第 1~23 个字符具有与此相同的值，那么这个请求将被转发给相同的服务器。

现在将上面讨论的内容组合起来，如图 41-12 所示。路由器就是所有请求都应该到达的地址；它可能由 DNS 名称表定义为 www.mycompany.com（或者类似的形式）。基于上面讨论的某种 SLB 算法，网络通信将被转发给某个 Web 服务器。然后，每个 Web 服务器将向一系列 JVM（有时是一个 JVM，有时是多个 JVM，具体取决于 WebSphere 负载平衡要求）发出请求。最后，数据库服务器通常有一个热备份正在运行，从而使得如果这个数据库服务器关闭，替代服务器能够在几秒钟之内在线工作，从而确保系统继续正确运行。

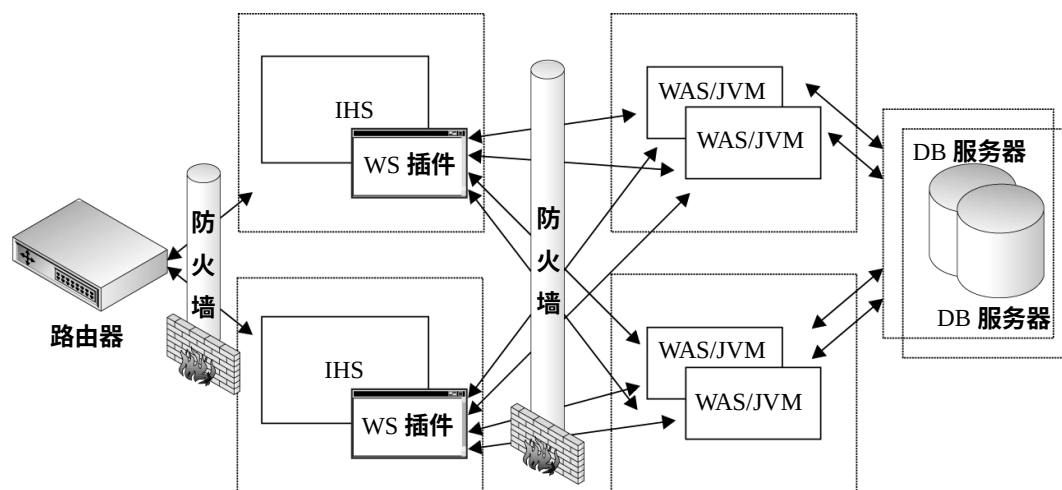


图 41-12 完全配置

41.3 小 结

本章详细讨论了各种可能的部署拓扑。学完本章之后，读者应该能够构建一个完全容错的解决方案，并且能够理解如何设置各种服务器与路由器。其中某些设置将在第 42 章继续讨论，那一章将深入描述 WebSphere 管理工具。

第 42 章 WebSphere Administrative Console

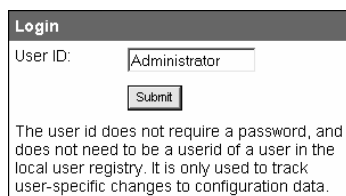
WebSphere Application Server 是一个复杂而又完善的产品。因此，配置与管理 WebSphere 并不是一件简单的任务。为了帮助完成这些任务，IBM 提供了一个用户友好的界面，称为 Administrative Console。

Administrative Console 是区别 WebSphere Advanced Edition Single-Server (AEs) 和 WebSphere Advanced Edition (AE) 产品的功能之一。在 AEs 中，Administrative Console 被实现为基于 Web 的界面，通过端口 9090 对其进行访问。相反，AE 提供了一个实现此控制台的 Java 客户应用程序。后者功能更强，并且更具有交互性（例如，控制台能够在不需要用户刷新的情况下输出错误消息）。另一个区别就是 AE 的 Administrative Console 提供了对管理多个网站的真正支持。从历史的观点看，需要说明的有趣一点是，“古老”版本的 WebSphere（2.0 版本以及更早的版本）有一个基于 Web 的界面（使用 applet），而后来的版本（3.x 版本以及后续版本）提供了基于 Java 的 Administrative Console。

本章将描述这两种 Administrative Console。本书配套光盘上提供的 WebSphere 安装是 AEs 版本，因此，如果使用了这个版本的 WebSphere，那么读者可能仅仅对本章的前半部分感兴趣，前半部分主要关注基于 Web 的控制台。但是，如果正在运行（或打算运行）一个产品网站，那么很可能需要使用 AE，这时读者将会对本章后半部分的信息更感兴趣。

42.1 AEs Administrative Console

我们在前面已经介绍了 AEs Administrative Console，它主要用来完成诸如部署 Web 应用程序与企业应用程序这样的任务。AEs Administrative Console 是基于 Web 的，并且使用 URL <http://localhost:9090/>（假设 WebSphere 在本地运行）对它进行访问。当访问这个页面时，要求输入用户名，如下图所示。用户名并不需要口令，它仅仅用来跟踪对配置所做的修改。



在输入用户名之后，将出现 AEs Administrative Console 的主页，如图 42-1 所示。

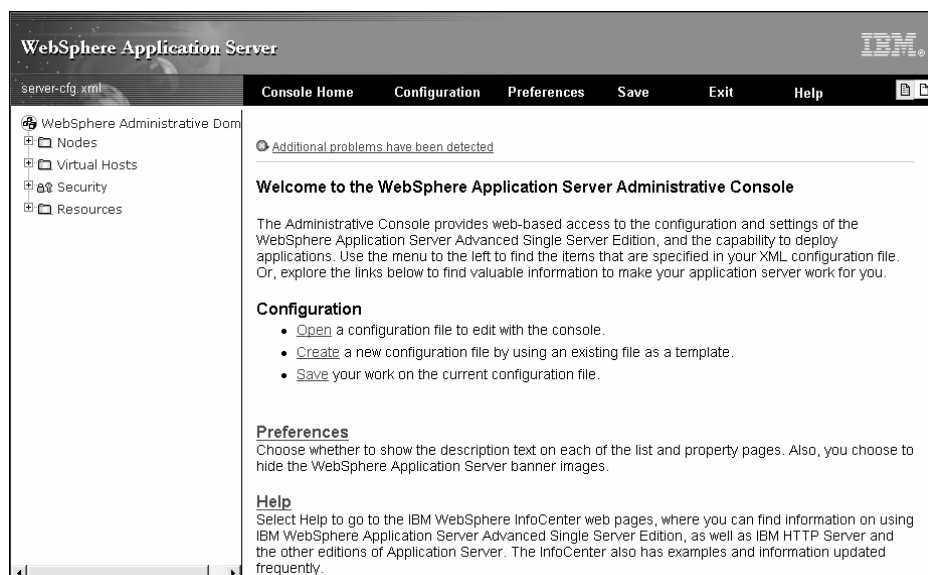


图 42-1 AEs Administrative Console

这个控制台的左边窗格包含一个树形控件，可用来对各种配置组件导航。在树形控件的上方是当前活跃的配置文件名。WebSphere 配置信息存储在 XML 文件中，这个文件位于 WebSphere 安装根目录的 config 目录下。



在使用 AEs Administrative Console 时，这个树形控件总是可用的。它模拟 Windows Explorer 与类似的应用程序的标准行为，其中，树形控件用来导航，并且可以在这个树形控件上选择一个节点，从而在主应用程序窗口（这个窗口出现在右边窗格中，在这里没有显示出来）中查看其细节。在右边窗格的顶部，AEs Administrative Console 提供了一个菜单条，如下所示：



这个菜单条具有下列选项：

- Console Home：用来返回到控制台主页。
- Configuration：用来选择一个配置文件，从而对其进行编辑。默认情况下，编辑的是 config/server-cfg.xml 文件，但也可以加载其他文件，如图 42-2 所示。

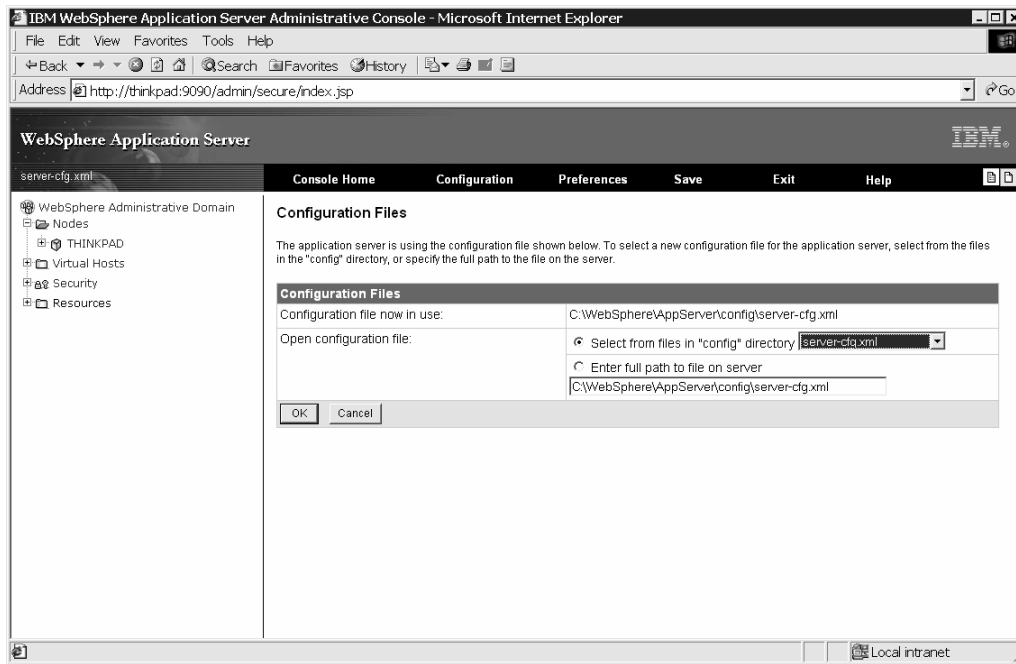


图 42-2 在 Configuration Files 页面中可以加载其他文件进行编辑

- Preferences：用来控制控制台的行为。它提供了下列选项：
 - Show Descriptions：在打开了描述功能之后，控制台为每个字段提供详细的描述。在菜单条的右端有两个按钮，用来快速启用与禁用描述。
 - Show IBM Banner：触发在控制台的顶部显示蓝色条幅，这个条幅用来显示产品名与 IBM 标志。
 - Enable Validation After Every Edit：使控制台对每个动作进行有效性检查。对于富有经验的管理员，在执行批处理动作时，可能会关闭这个功能，以提高响应时间。
 - JavaScript or no JavaScript Tree：允许选择树形控件的底层实现。默认情况下，控制台显示一个 JavaScript tree。可以选择 Load NoJavaScript Tree 选项，从而使用普通的 HTML 树形控件。HTML tree 总是以完全展开的方式出现，如下页右图所示（只有在浏览器不支持 JavaScript 时，这个选项才有用）。
- Save：保存对配置文件所做的编辑，这一步非常关键，因为如果没有保存所做的修改，那么这些修改将丢失。为了提示用户保存修改，控制台在发现存在没有保存的修改时，将显示一条消息，告知用户应该保存配置修改。
- Exit：允许正常地终止 Administrative Console 会话。当通过这个菜单条退出时，系统将提示用户保存所做的任何修改。

- Help：打开一个浏览器窗口，并且从产品在线文档中显示与上下文相关的 Help 页面。

Administrative Console 中的一个典型会话包括一系列编辑操作后跟一个保存所做修改的操作。树形控件用来导航到感兴趣的对象，并且编辑工作是在控制台右边窗格的主窗口中完成的。下一节将详细描述树形控件中的每个元素。

42.2 AEs Administrative Console 拓扑

在控制台左边窗格上的树形控件中，WebSphere Administrative Domain 的下面出现有 4 个文件夹，分别是：Nodes，Virtual Hosts，Security 与 Resources。

Nodes 代表在其中安装了 WebSphere 的物理机器（或主机）。当使用 AEs 时，在 Nodes 文件夹下总是存在一个这样的节点（毕竟，这是一个单网站版本）。

Virtual Hosts 文件夹允许管理虚拟主机。虚拟主机（virtual host）是一个提供几个机器外在表现的单独机器，其中每个虚拟机器都有一个不同的逻辑名（以及一个不同的 DNS）。虚拟主机并不共享 Web 资源。

Security 文件夹包含与安全性相关的设置，例如对底层操作系统与 SSL 进行的认证。

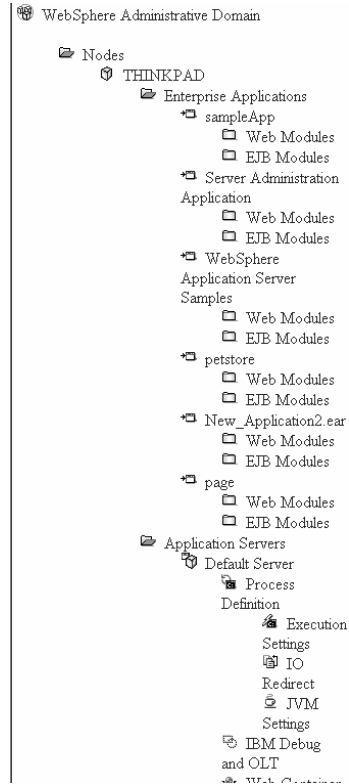
最后一个文件夹是 Resources，在这里可以管理 J2EE 资源，例如 JavaMail 提供者、JDBC 驱动程序与 JMS 提供者。

42.2.1 Nodes

在 Nodes 文件夹中，可以通过 Enterprise Applications, Application Servers, Path Map 与 Installed Resource Providers 项访问每个主机的大多数配置设置。

Enterprise Applications

单击 Enterprise Applications 项，将打开 Enterprise Applications 页面，在这个页面中可以访问安装在服务器上的应用程序。有一个图形标识用来指出这些应用程序是否正在运行（在图 42-3 所示的例子中，箭头表示应用程序正在运行；New_application2.ear 应用程序没有在运行，它用一个小 x 图标表示）。从这个窗口中可以启动、停止、重新启动、卸载或导出每个已经安装的应用程序。为了执行上述任何动作，需要选择一个或多个与应用程序相关联的复选框，并单击相应的按钮。



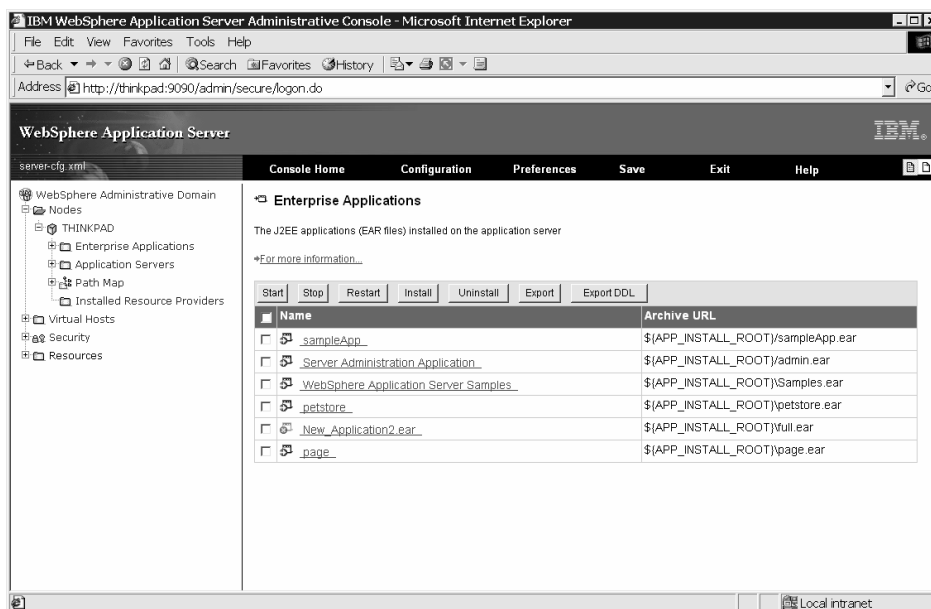
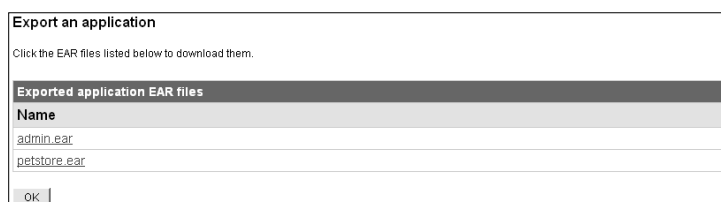


图 42-3 Enterprise Applications 页面

例如，如果选择两个应用程序并单击 **Export** 按钮，那么将打开下列页面。在这个页面中，可以单击与所选择的应用程序相对应的 **EAR** 文件，并将它们导出到一个文件中。



安装新应用程序或模块并不需要选择任何已安装的应用程序。在 **Enterprise Applications** 页面中单击 **Install** 按钮，将打开 **Application Installation Wizard**，在这里可以选择一个本地或远程 **Web** 模块、**EJB** 模块或一个企业应用程序。然后，就可以在 **WebSphere** 中部署所做的选择。第 4 章与第 5 章已详细讨论了这个过程。需要说明的是，对于 **Web** 模块或 **EJB** 模块，需要指定希望在其中部署该模块的应用程序，因为模块只能作为 **WebSphere** 中应用程序的一部分运行。

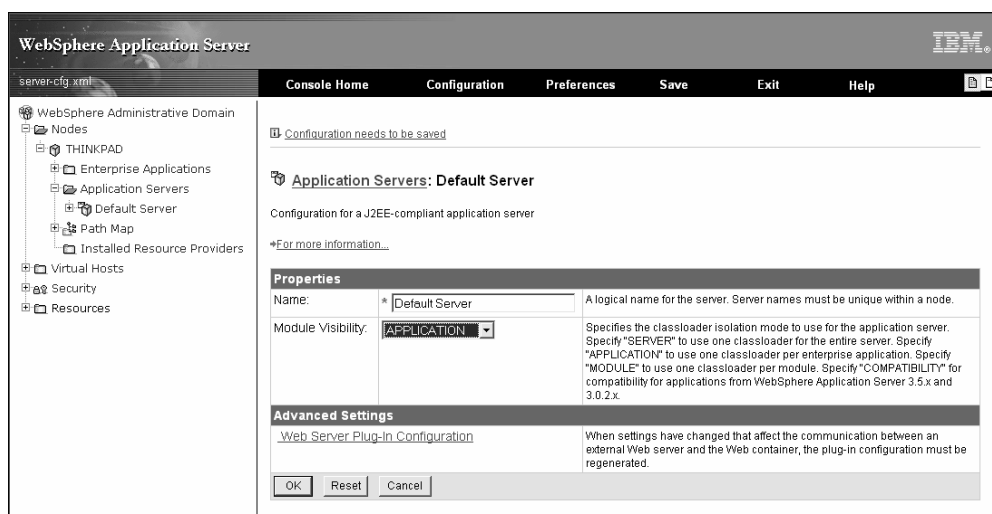
Application Servers

Nodes 下的 **Application Servers** 表项可用来控制 **WebSphere** 服务器本身的行为。

Default Server

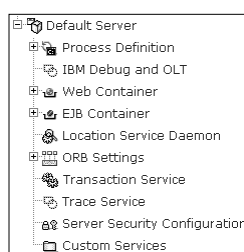
在 **Application Servers** 表项下有几个表项，其中包括 **Default Server**。在 **Default Server** 页面中，

可以 PING、启动与停止应用程序服务器：



在 Default Server 页面中，可以选择服务器名并控制模块可见性。模块可见性（module visibility）是一个技术话题；它控制了应用程序操作的名称空间。模块可见性选项 MODULE，APPLICATION 和 SERVER 用来决定 Java 类加载程序是为每个模块、每个应用程序还是为整个服务器提供。另一个选项是 COMPATIBILITY，用来确保与 WebSphere 3.x 具有相同的行为。如果正在运行一个 J2EE 兼容的应用程序，那么推荐使用 MODULE，因为它遵守 J2EE 规范。

在 Advanced Settings 下，可以重新生成 Web 服务器插件，这是一个重要的功能，它允许用户通过底层的 HTTP 服务器访问应用程序。在部署了新应用程序之后，就可以利用指向这个 WebSphere 服务器（例如，一个使用 9080 端口的服务器）的 URL 来访问此应用程序。重新生成插件，就允许 HTTP 服务器将请求重定向到与新应用程序相关的 WebSphere 上。如果跳过这个阶段，HTTP 服务器就不能识别指向新应用程序的 URL，并且当指定这些 URL 时，将返回一个错误页面。因此，只要对应用程序进行了 URI 路径或虚拟主机的修改，就需要重新生成插件。在 Default Server 下可用的附加的选项如下所示：



- **Process Definition**：可用来指定 Java 可执行程序、应用程序服务器的工作目录以及环境属性。这个表项具有下列子表项：
 - **Execution Settings**：控制应用程序服务器进程的操作系统设置：Priority，Umask（用于所创建文件的文件访问许可），Run-as User 和 Run-as Group。这些设置对应于 UNIX 系统（只要检查属性以及默认的 Run-as User 作为根（这在基于 Windows 的系统中通常并不是这样设置的）就可以发现这些设置对应于 UNIX 系统）。
 - **IO Redirect**：用来选择应用程序服务器 JVM 标准输出与标准错误被重定向到的文件。默认情况下，这些文件在安装根的 logs 目录中。
 - **JVM Settings**：控制与 JVM 相关的技术设置，例如 CLASSPATH、各种 JVM 组件（类加载程序、JNI（Java Native Interface，Java 本地接口）和垃圾收集器）的详细模式、堆大小设置、调试设置（例如要求使用 java_g 以后用调试功能）以及系统属性设置。
 - **IBM Debug and OLT**：启动一个分布式调试器，并且启动可以在 WebSphere 中使用的对象级跟踪机制。本书中没有描述这些工具，但它们是 WebSphere 发布的一部分。利用这个选项，还可以决定对象级跟踪查看器所在的主机名与端口，如下所示：

IBM Distributed Debugger and Object Level Trace Settings

Configuration for IBM Object Level Trace and IBM Distributed Debugger. Complimentary copies of these products are provided for tracing and debugging the applications running on your application server. See the InfoCenter for extensive information.

[*For more information...](#)

Properties	
☐ Enable Object Level Trace (OLT)	Enables the tracing of your application with OLT. If you want to use OLT for tracing and debugging of your application, ensure to also enable the IBM Distributed Debugger below.
Host Name: * localhost	Specifies the fully qualified host name of the machine on which the OLT trace viewer is located. Trace results will be directed to the machine.
Port: * 2102	The port number on which the OLT tool is receiving the tracing information
☐ Enable IBM Distributed Debugger	This makes the execution of your application enabled for the IBM Distributed Debugger. With this selection you can debug your application in standalone mode or through OLT
Source Path:	The root directories from which to obtain the source code of your application for debugging purposes if not already set in your CLASSPATH

Advanced Settings	
Dynamic Properties	Service properties that can be configured at runtime

OK Reset Cancel

Web Container

Web 容器包容 Web 应用程序并为 Web 应用程序服务。在这个选项下有下列项：HTTP Transports，Session Manager，Installed Web Modules 与 Thread Pool。

HTTP Transports 控制绑定在 Web 容器上的端口列表。默认情况下，这些端口是 9080（用于访问 Web 应用程序）、9090（用于 Administrative Console）与 9443（用于 Web 服务器插件）。对于每个端口，都可以指定是否启用 SSL。如果启用，还可以指定与 SSL 相关的属性，如

下图所示。这些属性用来决定与加密相关的设置，例如密钥文件与信任文件。

SSL Settings

Configuration settings for using Secure Socket Layer (SSL)

[For more information...](#)

Properties

Key File Name:

{WAS_ROOT}/etc/DummyServ

The fully qualified path to the key file that contains public keys and perhaps private keys

Key File Password:

The password for accessing the key file

Key File Format:

JKS

The format of the key file

Trust File Name:

{WAS_ROOT}/etc/DummyServ

The fully qualified path to a trust file containing the public keys

Trust File Password:

A password for accessing the trust file

☐ Enable Client Authentication

Specifies that the server and client should prove their identities through an exchange of keys.

Security Level:

HIGH

One of a preconfigured set of security levels

☐ Enable Crypto Token Support

Whether cryptographic token support is enabled

Advanced Settings

[Dynamic Properties](#)

[Crypto Token](#)

Specifies information about the cryptographic tokens related to SSL support

OK

Reset

Cancel

在指定一个新 HTTP 传输之后，就需要定义一个相应的虚拟主机别名。

HTTP 传输设置,对于修改用于直接访问(没有经过 Web 服务器)Web 应用程序的内部 HTTP 传输端口以防 9080 端口被占用的情况非常有用。对于一个产品环境，可能希望禁用这个端口,因为内部传输的性能与可伸缩性不能与 Web 服务器所使用的 Web 插件的相应端口匹配。

Session Manager (在下图中显示) 控制 servlet 会话机制，这正如第 23 章描述的那样。Session Manager 属性允许启用会话跟踪的两个方法 :重写 URL 和 cookie。也可以启用 SSL 跟踪 (也就是，在 SSL 会话中使用会话)、协议切换 (当从 HTTP 切换到 HTTPS，或者进行其他切换时维护会话)、持续性会话与安全性集成 (将已经认证的用户名与会话相关联)。此外，还可以添加将要被 Web 容器服务所使用的 Java 属性。

Session Manager

Configuration for session support in the Web container. When configuring the Session Manager, you can specify: Whether to enable sessions How to convey session IDs to servlets (cookies or URL rewriting) AE: Whether to save session data in a database during execution (persistent sessions) Whether to add session IDs to URLs in transition from HTTP to HTTPS and back (protocol switch rewriting)

[For more information...](#)

Properties

☐ Enable URL Rewriting

Specifies whether the Session Manager uses rewritten URLs to carry the session IDs. If URL rewriting is enabled, the Session Manager recognizes session IDs that arrive in the URL and, if necessary, rewrites the URL to send the session IDs.

☒ Enable Cookies

Whether session tracking will use cookies to carry session IDs. If cookies are enabled, session tracking will recognize session IDs that arrive as cookies and try to use cookies for sending session IDs. If cookies are not enabled, session tracking will use URL rewriting instead of cookies (if URL rewriting is enabled). Enabling cookies takes precedence over URL rewriting.

☐ Enable SSL Tracking

Whether session tracking uses SSL to carry session IDs. Enabling SSL tracking takes precedence over cookie-based session tracking.

☐ Enable Protocol Switch Rewriting

Whether the session ID is added to a URL when the URL requires a switch from HTTP to HTTPS or from HTTPS to HTTP. If rewriting is enabled, the session ID is required in order to go between HTTP and HTTPS.

☐ Enable Persistent Sessions

When persistent sessions are enabled, the Session Manager will persist session information into the data source specified by the data source connection settings. Otherwise, the Session Manager will discard the session data when the server shuts down.

☐ Enable Security Integration

When security integration is enabled, the Session Manager will associate the identity of users with their HTTP sessions.

Advanced Settings

[Dynamic Properties](#)

Service properties that can be configured at runtime

OK

Reset

Cancel

Session Manager 具有下列子表项：

- **Default Cookie Settings**：控制用于会话跟踪的 cookie 的行为。对于 J2EE 规范，这个 cookie 名是固定的。可以修改与 cookie 相关联的域和路径，并且修改其存活时间。
- **Persistent Sessions**：指定用于存储持续性会话的数据源与数据库参数（例如用户 ID、口令与表空间名）（只有当启用持续性会话时，这个表项才有意义）。
- **Tuning Parameters**：指定与 Session Manager 相关联的技术参数，包括会话的无效超时、内存中维护的最大会话数，以及持续性会话的调整参数（例如用于决定何时将会话写入到数据库中的策略——在服务方法的结尾、用手工的硬编码调用，或在固定的时间间隔写入）。

Installed Web Modules 显示一个关于已安装的 Web 模块的列表；对于每个已经安装的模块，都提供了一条到达包含这个模块的企业应用程序的链接，并且可以查看这个模块的部署描述符。Web 容器使用池维护正在运行的 servlet 所使用的线程。利用 Thread Pool 子表项，可以指定池的大小、不活跃超时（线程被删除之前经历的时间间隔）以及池的最大大小是否可以被超过。

WebSphere 能够 cache 动态 servlet 与 JSP 文件的输出，并且这一功能能够极大地提高 Web 应用程序的性能（但是，从 cache 中取出的数据并不一定是当前数据）。完善的 cache 机制基于对 service 方法调用的截取来实现，并且这一机制考虑请求参数、URI，cookie 和会话信息。

Dynamic Cache 表项可用来启用 cache、控制 cache 大小以及为被 cache 的 servlet 设置默认优先级（代表 LRU 算法中的周期数，在这个算法中有一个表项被确保仍然在 cache 中；推荐值为 1）。

cache 机制也可以使用外部 cache，这可以用 Cache Groups 子表项来指定（这一功能仅仅在 AE 中完全支持，因此，将在本章后面对其进行详细讨论）。

EJB Container

EJB 容器包容并调用 EJB，它提供了下列参数：

- **Passivation Directory**：允许指定一个目录，被钝化的会话将存储在这个目录中——EJB 容器的“交换空间”。
- **Inactive Pool Cleanup Interval**：允许指定一个时间间隔（单位为毫秒），在经过这个时间间隔之后，不活跃的 bean 池将减少到最小数目。
- **Default Data Source**：可用来指定 EJB 容器使用的默认的和 JDBC 兼容的数据源（例如，对容器管理的持续性）。

在 Cache Settings 下还有其他可用的参数；这些参数可用来设置 cache 大小与清除间隔。这

个间隔决定了容器试图从 cache 中删除没有被使用的项目的频率。

在 EJB 容器下的其他子表项是 Installed EJB Modules。与 Web 模块的相应表项一样，可以使用这个选项来查看模块的部署描述符，以及添加一条到达包含这个模块的企业应用程序的链接。

ORB Settings

可以在这里控制 ORB (Object Request Broker, 对象请求代理)。可以启用这个服务 (默认情况下它是启用的, 因为 EJB 需要使用这一服务) 并且指定引导过程的主机与端口。如果 ORB 与其他 ORB 一同使用, 并在默认情况下使用了 localhost 及其 900 端口, 那么就可以使用这个主机与端口。此外, 还可以指定 com.ibm.CORBA 包中定义的其他属性。

在 ORB Settings 下有 Pool 与 SSL Settings 表项。除了所指定的参数与不同的池相关之外, Pool 表项完全与用来控制 Web 模块线程池的选项相同。EJB Container 维护自己的线程池。SSL Settings 表项允许为使用 SSL 中的 IIOP 而定义 SSL 参数。这些参数与用来设置 SSL HTTP 传输端口的参数相同。

ORB Settings 下有以下子表项：

- Transaction Service：用来控制事务服务的行为。在这里可以启用 EJB 事务，以及指定一个日志文件。此外，容器还允许指定事务生命周期的最长期限（单位为秒）。在这个时间结束之后，事务被强制回滚。这一功能对于防止死锁是重要的。类似地，可以指定一个客户不活跃超时，为此，容器将计算自从最后一次与客户交互以来所经历的时间。如果出现了用户不活跃超时的情况，这个事务将被异常终止。

Transaction Service 功能如下图所示。

Configuration needs to be saved

Transaction Service

Configuration of the transaction service

[*For more information...](#)

Properties	
☐ Enable	Whether the server will attempt to start the specified service
Transaction Log File:	\${TRANLOG_ROOT}/tran1.log The log file for transaction service logging. If specifying multiple log file names, separate them with commas. For example, specify: "W:\WebSphere\AppServer\logs\ran1.log;W:\WebSphere_0\AppServer\logs\ran2.log" If this property is blank, the server will use in-memory logging. In a production environment, be sure to set a value because transaction rollback capability requires transaction logging.
Total Tran Lifetime Timeout:	120 The container will rollback transactions whose total lifetime (ie. timeperiod since beginning of transaction) exceeds the value of this setting. This value should be specified in seconds.
Client Inactivity Timeout:	60 The container uses client inactivity timeout to forcibly abort transaction if a client becomes inactive. Basically, assume that a client calls an EJBServer and invokes some method. When the call returns to client, the timer is started. If client does not come back within the client inactivity timeout interval, the transaction in the server is rolledback by the container. This value should be specified in seconds.

Advanced Settings

[Dynamic Properties](#)

Service properties that can be configured at runtime

OK Reset Cancel

- Trace Service：控制跟踪的日志，在第 21 章已详细讨论过。

- **Server Security Configuration**：这里的惟一选项就是 **Use Domain Qualified User Names**。当这个选项被启用时，用户名总是被它所在的安全性域所检查。这一功能可应用于从方法（例如 `getPrincipal()`）所返回的用户名。
- **Custom Services**：允许将自己的服务附加到应用程序服务器上。这些服务在本质上是用户提供的类，并且在服务器启动时被调用（当服务器关闭时，这些服务也被关闭）。

在这里可以定义自定义服务，为此，需要指定名称、描述与类名。这些类应该实现 `com.ibm.websphere.runtime.CustomService` 接口。当指定一个自定义服务时，也可以为这个服务指定一个外部配置文件（以 URL 的形式指定），以及指定可以在何处找到这个类的 classpath。

`com.ibm.websphere.runtime.CustomService` 比较直观，它仅仅提供了 2 个方法。其中，`initialize` 方法使用从所指定的外部配置文件构造的 `java.util Properties` 对象作为参数，并且当应用程序服务器被启动时调用这个方法。当服务器关闭时调用 `shutdown` 方法。

Path Map

Path Map 表项是 Nodes 的下一个部分。它汇总了其他设置中的逻辑路径列表，如下所示：

Entries			
A list of file system roots			
*For more information...			
New Delete			
Symbolic Name	Path	Description	
APP_INSTALL_ROOT	\${WAS_ROOT}/installedApps	The filesystem path to the directory which will contain installed enterprise applications.	
LOG_ROOT	\${WAS_ROOT}/logs	The filesystem path to the directory which will contain server log files.	
TRANLOG_ROOT	\${WAS_ROOT}/tranlog	The filesystem path to the directory which will transaction log files.	
PRODUCT_INSTALL_ROOT	\${WAS_ROOT}	The filesystem path to the product installation directory	
WAS_ROOT	C:\WebSphere\AppServer	The filesystem path to the product installation directory	
JAVA_HOME	C:\WebSphere\AppServer\java	The filesystem path to the product java home directory	

当需要指定一个目录时，就可以使用任何上述路径。这样就使得能够在一个集中的位置控制各种文件所在的目录。例如，事务服务日志的默认值是相对于 `TRANLOG_ROOT` 路径指定的：

```
$ {TRANLOG_ROOT}/tran1.log, ${TRANLOG_ROOT}/tran2.log
```

Installed Resource Providers

这是 Nodes 下的最后一个子表项。可用来指定应用程序服务器可用的 J2EE 资源（例如 JDBC 驱动程序）列表以及与它们相关联的 classpath。可以使用 Resources 表项添加新资源，它将在 42.2.4 节“Resources”中讨论。

42.2.2 Virtual Hosts

Virtual Hosts 表项允许添加和删除一个虚拟主机，然后，可以为其设置名称、DNS 别名（包括与其绑定在一起的端口号）以及默认 MIME 表项。

默认情况下，使用一个端口为 80 与 443 的 default_host 表项，9080 端口和 9443 端口用于 Web 服务器插件。admin_host 使用 9090 端口。

注意：虚拟主机别名必须与 HTTP 传输所使用的每个端口相对应。

42.2.3 Security

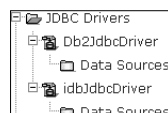
使用 Security 表项，可以启用安全性以及为认证 cache 设置一个超时。需要说明的是，默认情况下安全性是禁用的，因此，对于产品环境，需要做的第一件事情就是启用安全性。在 Security 表项下提供的是对底层操作系统认证的选项以及进行 SSL 设置的选项。第 44 章完全集中讨论安全性管理，并且，该章将介绍如何使用 Administrative Console 来完成这些任务。

42.2.4 Resources

在 Resources 表项中，可以将下列 J2EE 资源添加到应用程序服务器中：JDBC 提供者、JMS 提供者、URL 提供者以及 JavaMail 提供者。一旦某个资源被选中，那么这个资源的表项将显示一个可用的提供者列表。

例如，如果选择 JDBC Drivers 表项，那么将显示已经配置的驱动程序列表。可以使用适当的按钮来添加或删除驱动程序。当添加一个驱动程序时，预先设置的选项包括为 Oracle，DB2 与 Microsoft SQL Server 提供的驱动程序，但也可以添加任何与 JDBC 兼容的数据库驱动程序。

也可以展开树形控件中的 JDBC Drivers 表项，直接访问可用的驱动程序列表。在每个驱动程序表项下是一个 Data Sources 表项，如以下演示所示。其中每个表项包括为其定义了 JNDI 名称的特定数据源、数据库表、数据库用户名与口令以及连接池参数。



对于其他资源，使用类似的概念：例如，在 Resources 文件夹下是特定的提供商（类似于 JDBC 驱动程序），并且在它们下面是对象列表（类似于数据源）。这些对象是邮件提供者的会话，URL 提供者的 URL 以及 JMS 提供者的 JMS 连接。

第 14 章与第 15 章已讨论了 JavaMail, JDBC 与 JMS。URL 提供者用来处理新协议。JDK (Java 开发者工具包) 支持默认的提供者处理协议，例如 http, ftp 与 file 协议。对于其他协议，需要自己添加提供者，并且指定用来处理这些协议的数据流的类。

也可以定义自己的 JavaMail 会话对象,在这种情况下,需要指定邮件传输与邮件存储的细节(例如协议、主机名、用户名与口令)。

AEs 提供了对 JMS 资源的有限支持。它仅仅支持两种类型的 JMS 提供者(MQSeries 和 SonicMQ)。对于每种提供者,可以定义连接工厂与目标。正如在第 14 章描述的那样,即 JMS 代码在很大程度上依赖于定义连接工厂和消息目标的管理设置。

42.3 AE Administrative Console

WebSphere Application Server 的 AE Administrative Console 以 Java 应用程序的形式提供。此外,这个控制台与管理服务器通信,而管理服务器能够启动与停止应用程序服务器(可能是多个服务器)。这不同于 AEs,在 AEs 中,Administrative Console 只能与应用程序服务器交互。AE 中所使用的方法更具有可伸缩性,表现在它能够管理多个应用程序服务器。而且,它在执行诸如重启应用程序服务器这样的动作时更好用。

AE Administrative Console(如图 42-4 所示)的启动方法包括:从 WebSphere 的 First Steps 窗口中启动、从开始菜单启动或使用 adminclient.bat 脚本启动。

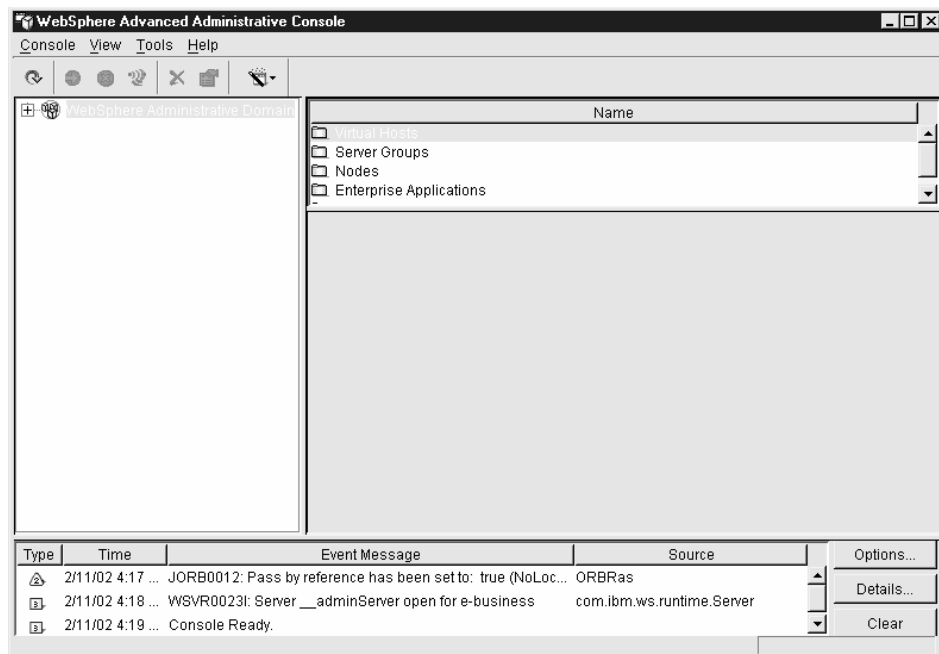


图 42-4 AE Administrative Console

当启动这个控制台时,将出现一个在其中包括 IBM 标志与 WebSphere 标志的窗口。控制台

需要与管理服务器通信，在启动控制台之前必须启动管理服务器。否则，控制台将不能启动，并且将出现下列消息：

```
ADGEU2008E: The Administration Client failed to connect to the
Administration Server. Start the local or remote Administration Server
service before launching the Administration Client.
```

当从命令行启动管理服务器时，所使用的是不同的脚本，该脚本称为 `adminserver.bat`（如果在 UNIX 上运行，则称为 `adminserver`）。在 Windows 上，管理服务器也可以作为 Windows NT 服务被启动。

AE Administrative Console 是一个 Java 应用程序。它包含一个菜单条、一个工具条、在左边包含一个树形控件以及在右边包含一个细节窗口，并且在底部包含一个消息区。

树形控件用来浏览各种服务器及其组件，在右边的细节窗口中可查看与编辑属性。与 AEs Administrative Console 中基于 Web 的界面相比，消息区是一个重要的功能改进，因为它可以立刻通知用户任何事件或问题的发生。

树形控件的根是 WebSphere Administrative Domain（与 AEs 中的相同）。在这个根下有 5 个表项。其中 4 个表项（Virtual Hosts, Nodes, Enterprise Applications 与 Resources）与 AEs 中的相同。另一个新表项是 Server Groups，这个表项用来管理多个 WebSphere 实例，这些 WebSphere 实例协同工作并向外界提供类似于一个单独服务器的功能。

如果应用程序服务器没有启动并运行，需要启动这个服务器。为了避免混淆，应该记住，为了启动控制台，首先要保证管理服务器正在运行，但没有对应用程序服务器提出这样的要求。展开 Nodes 文件夹，并访问 Applications Servers 文件夹。在这里，应该见到 Default Server，它是在安装过程中自动创建的。右击 Default Server，并且从弹出菜单中选择 Start。当这个服务器在运行时，它将在消息区中发出一条“open for e-business”（打开电子商务）消息。

42.4 使用 AE Administrative Console

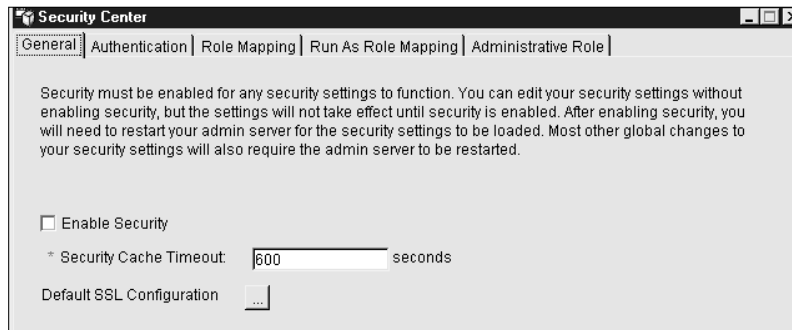
在 AE Administrative Console 中，可以浏览管理域中的对象树，以及查看并编辑这些对象——类似于使用 AEs 控制台时的过程。为了减轻管理工作，AE 控制台提供了一系列向导，用来指导用户完成常见的管理任务。

42.4.1 AE Administrative Console 菜单条

AE Console 菜单提供了下列命令：

- New：创建新对象。
- Wizards：为完成常见的任务提供向导。

- Trace：控制跟踪机制。
- Security Center：打开 Security Center 对话框（如下图所示），在这个对话框中汇总了安全性设置。需要说明的是，默认情况下安全性是禁用的，因此如果需要使用与安全性相关的功能，应该记住启用安全性（管理安全性是第 44 章讨论的话题）。



- Import from XML：导入一个 XML 配置文件。
- Export to XML：将配置导出到一个 XML 文件中。
- Start：启动所选择的对象。
- Stop：停止所选择的对象。
- Force Stop：强制所选择的对象停止。
- Ping：Ping 所选择的对象（以判断其是否正在工作）。
- Remove：删除所选择的对象。
- Properties：显示所选择对象的属性。
- Find：基于类型或名称标准，搜索一个对象。
- Command History：显示一个关于所有已执行命令的表，包括它们的状态、完成时间以及完成结果。如果失败，还可以查看与这个命令相关联的错误。
- Exit：退出控制台。

View 菜单提供了下列选项：

- Show Property Pane：可控制用来显示所选择对象属性的属性面板是否打开。
- Show Message Pane：可控制是否打开消息面板。
- Runtime Inspector：可控制是否打开运行时检查工具。运行时检查工具是一个独立于控制台主窗口的非模态对话框。它显示运行时信息，例如进程 ID 与当前正在使用的设置。

Tools 菜单调用与管理任务相关的外部工具：

- Application Assembly Tool：装配 Web 模块、EJB 模块与企业应用程序（请参见第 4

章与第 5 章)。

- Log Analyzer：分析跟踪文件（请参见第 21 章）。
- Resource Analyzer：分析资源。

Help 菜单中可以使用下列选项：

- Concept Help：根据 IBM InfoCenter 为 WebSphere 提供的文档定义所选择的对象。
- Task Help：来自 IBM InfoCenter 为 WebSphere 提供的文档中的特定任务帮助。
- Field Help：从 IBM InfoCenter 为 WebSphere 提供的文档中给出上下文敏感的（特定字段）在线帮助。
- Information Center：打开 IBM InfoCenter 为 WebSphere 提供的文档中的 AE Administrative Console 页面。
- About：显示产品信息。

42.4.2 AE Administrative Console 工具条

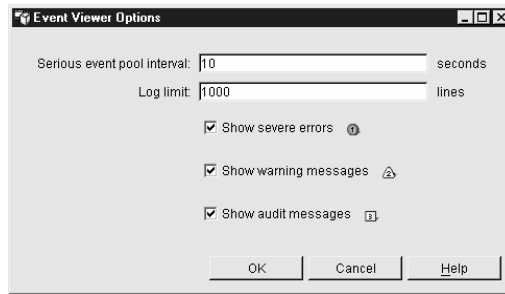
AE Administrative Console 工具条提供了如表 42-1 所示的按钮。可以从 Console 菜单或从右击某个对象时弹出的菜单中获取每个按钮的功能。

表42-1 AE Administrative Console工具条按钮

按钮	功能
	刷新所选择的对象（及其子树）
	启动所选择的对象
	停止所选择的对象
	Ping 所选择对象（以检查该对象是否是活的）
	删除所选择的对象
	在一个新对话框中显示所选择对象的属性（与出现在属性面板中的属性相同）
	显示 Wizards 菜单（与 Console 菜单中的 Wizards 表项相同）

42.4.3 AE Administrative Console 消息面板

AE Administrative Console 消息面板中的每条消息都包括一个类型、时间、消息文本以及来源。消息可以是 Severe，Warning 与 Audit。可以对消息进行过滤；并且可以通过消息面板的 Event Viewer Options 对话框（如下图所示）控制池间隔以及消息日志的长度。可以单击消息面板上的 Options 按钮来访问这个对话框。



为了查看关于某个特定消息的细节，应该在控制台消息面板中选择这个消息并且单击 **Details** 按钮。这时，事件细节窗口将显示事件 ID、时间、类型、节点、服务器、线程 ID、源与完全文本。最后，可以单击 **Clear** 按钮来清除消息日志。

Virtual Hosts 文件夹与 Nodes 文件夹

Virtual Hosts 文件夹中包含的信息类似于 AEs 控制台的 Virtual Hosts 文件夹中可用的信息。可以在这里定义一个虚拟主机列表，并且为每个主机定义由主机名与端口号组成的别名。在 Virtual Hosts 页面的 **Advanced** 标签下，可以定义 MIME 扩展。

与 AEs 一样，AE 控制台的 Nodes 文件夹包含一个关于物理机器的列表。在 Nodes 之下有 Application Servers 与 Generic Servers 表项，将在后面的两个小节中讨论它们。节点属性包括节点名（不能被修改，因为这是只有底层操作系统才具有的特权）以及已经安装的 J2EE 资源集合。每种类型的资源都有一个标签与 AE 控制台中的 Installed Resources 表项相对应。Nodes 页面中的每个标签显示了相应类型的资源列表，允许用户安装新资源。如果安装新提供者，可以从 Resources 文件夹的相应类型的提供者列表选择一个提供者。

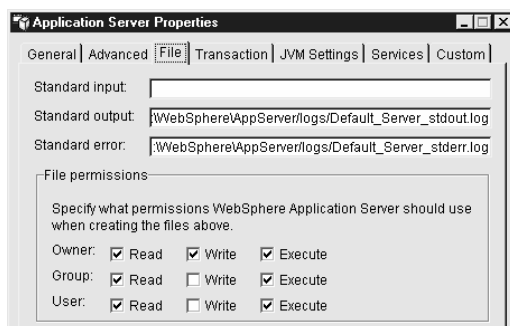
Application Servers 文件夹

出现在 Nodes 文件夹中的每个节点都有两个子文件夹：Application Servers 文件夹与 Generic Servers 文件夹。在 Application Servers 文件夹中，可以发现在这个节点上正在运行的每个应用程序服务器实例。右击应用程序服务器图标，在弹出菜单中可以启动、停止、删除或 ping 这个服务器（其中有些选项可能变灰，例如，如果这个服务器已经被启动，那么就不能启动这个服务器了）。在这个弹出菜单中还包括一个 Transactions 项，它允许用户查看一系列当前正在运行的事务。这对于处理死锁问题以及其他事务相关问题是很有用的。在 Transactions 窗口中，可以确认或回滚每个事务，也可以查看其细节。

应用程序服务器属性提供了几个不同的标签。其中，General 标签包括服务器逻辑名、它所在的节点、工作目录以及环境。可以在这里指定启动状态，它决定了当节点被启动时是否自动启动这个服务器。默认情况下，WebSphere 服务器不会自动启动。这个设置决定了当管理服务器启动时，应用程序服务器将干什么。General 标签中的最后一个选项就是 Module Visibility，它提供了与 AEs 控制台中所描述的相同的选项。

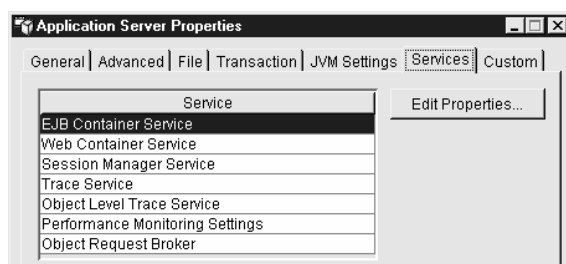
在 Advanced 标签中，可以为 ping 操作设置参数（间隔时间与超时）。也可以为进程设置操作系统设置（进程优先级、用户 ID 与组 ID）。最后一个选项就是可以使用域认可的用户名（在 AEs 的 Security 下）。

File 标签如下所示，可以在这里为服务器指定标准输入、标准输出与标准错误文件。它也为给所创建的文件设置掩码提供了图形界面。



在 Transaction 标签中可以设置事务超时与事务非活跃超时（单位为秒）。JVM Settings 标签可用于控制 JVM 堆大小、classpath、系统属性以及其他设置（例如启用垃圾收集器详细模式）。

Services 标签中包含服务的列表，其中大部分服务在本章前面对 AEs 控制台的讨论时已经说明。在这个列表中的惟一个新服务与性能监视相关。在这里，各种服务的设置与其他服务设置相分离（例如 Process Definition），并且参数在标签中的归组并没有反映这些参数在 AEs 中的分类。当然还存在一些其他细微的区别。例如，在 Web Container 服务下，可以启用 servlet cache 以及使用外部 cache 组。



对外部 cache 组的支持比 AEs 中的支持程度更深。在这里可以为外部组指定适配器 bean。适配器 bean 充当 WebSphere 与外部 cache 之间的桥。Custom 标签允许定义实现 CustomService 接口的自定义服务。

与 AEs 的另一个不同在 AE Session Manager 服务中。当使用持续性会话时，控制台提供了一系列预定义的配置用来调整参数，如图 42-5 所示。

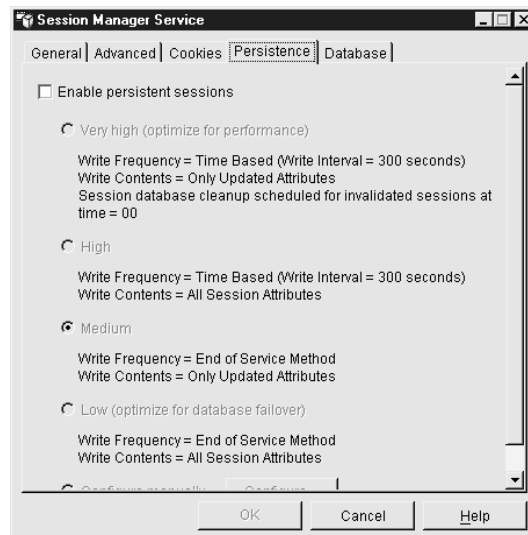


图 42-5 Session Manager Service 的 Persistence 标签

也可以手工地调整参数，然后在 AEs 控制台中为这一功能指定相同的参数。EJB 容器、跟踪服务、OLT 服务与 ORB 服务的设置与 AEs 中所讨论的相应设置相同。在 Performance Monitoring Settings 对话框中（如图 42-6 所示），可以监视服务器组件以及应用程序的性能。

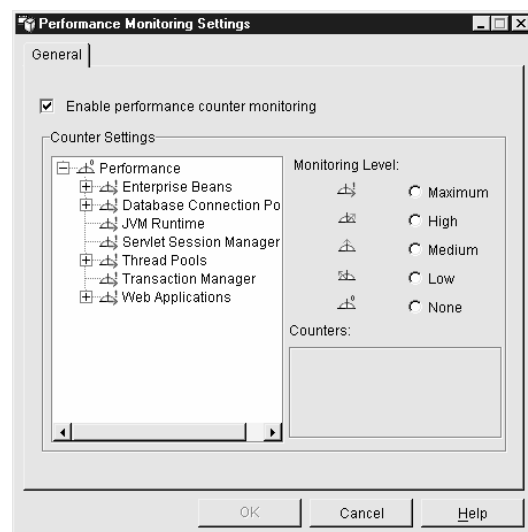


图 42-6 Performance Monitoring Settings 对话框

Performance Monitoring Settings 对话框提供了一个关于应用程序服务器组件的层次化显示，

通过可用来定位与选择一个单独 servlet 或 EJB 的树形控件，然后选择一个监视级别，就可以获得这种层次化显示。如图 42-6 所示，这个对话框显示了为每个监视级别维护的计数器列表。被监视数据的显示是通过 Resource Analyzer 工具（这个工具也提供了定义监视设置的能力）完成的。应该记住的是，设置高级别的监视将极大地影响性能，并且在启用监视功能时，不应该运行产品系统。

在 Application Server 文件夹下，有两个表项分别代表已安装的 Web 模块与 EJB 模块。它们分别对应于 AEs 控制台中的 Web 容器与 EJB 容器的表项。

EJB Modules 表项提供了一个关于所有已安装模块的列表。通过选择一个模块，就可以查看它的部署属性。与 AEs 控制台不同，在这里可以实际地编辑部署属性。显然，这一功能提供了极大的灵活性，并且使得应用程序的部署更为容易。

类似地，Web Modules 表项列出了可用的 Web 模块，并且允许用户查看与编辑它们的属性。但不能编辑一个 Web 模块的 Context Root——这是由应用程序部署者通过 Application Assembly Tool 工具完成的。

注意，在图 42-7 中，Enterprise Application 下列出了 Web 模块与 EJB 模块的细节。在 Enterprise Application 下为每个模块显示的属性，与在 Application Server 文件夹的相应模块文件夹下列出的属性完全相同。

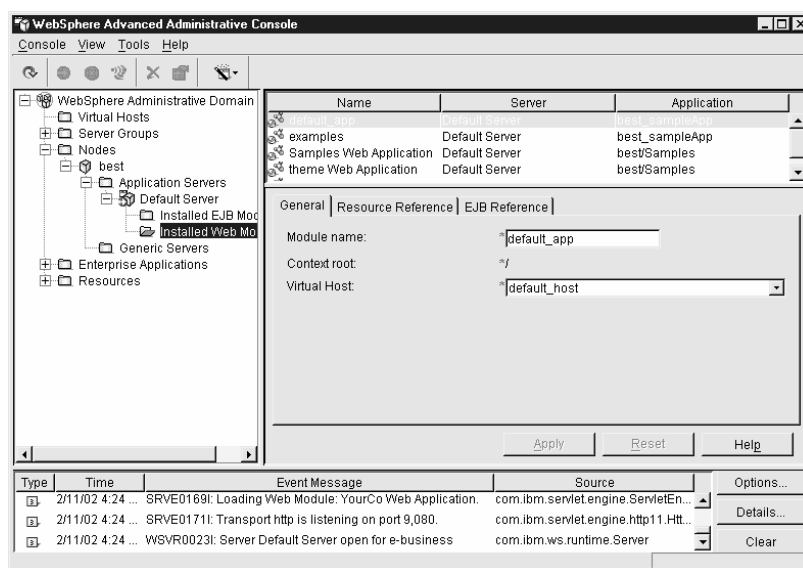


图 42-7 Web 模块与 EJB 模块的细节

Generic Servers 文件夹

WebSphere 中的普通服务器就是常见的可执行程序，它们可以是库、脚本或通过 JRE（Java Runtime Environment，Java 运行时环境）访问的 Java 代码调用。能够附加这种普通服务器

的能力，进一步补充了具有 CustomService 接口的 Java 类的自定义服务这一概念。

在 Generic Servers 下，可以定义普通服务器并编辑它们的设置。这些服务器是包含在 Administrative Console 中的，用来为定义与应用程序服务器一同启动与停止的外部进程提供一种简单的方法。例如，企业应用程序通常需要被打包为能够在 WebSphere JVM 之外部署的 RMI 服务器，但这样又需要它们与应用程序组件一同启动与停止。

Enterprise Applications 文件夹

Enterprise Applications 文件夹包含所有已部署的应用程序。在这里，可以查看每个应用程序的 EJB 模块与 Web 模块。

企业应用程序的属性组织在 3 个标签中：

- General：包含应用程序名。
- User/Role Mappings：将应用程序角色映射到特定的用户与组。当指定用户时，需要指定一个有效的口令。
- Run-as Mappings：将 Run-as 角色映射到特定的用户。当指定用户时，需要给出一个有效的口令。

右击一个企业应用程序，在弹出菜单中可以查看部署描述符与导出应用程序。这两个功能与 AEs 控制台中的相应功能具有类似的行为。在相同的菜单中，还可以启动、停止或删除应用程序。

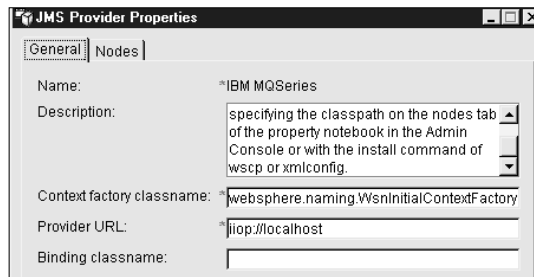
Resources 文件夹

在 Resources 文件夹中包含 JDBC 提供者、JavaMail 会话、URL 提供者、J2C (J2EE Connector Architecture, J2EE 连接器体系结构) 资源适配器以及 JMS 提供者。AE 控制台提供了类似于 AEs 界面的功能，但 AE 控制台技术性更强并且更加详细。例如，在添加一个 JDBC 驱动程序时，可以选择实现类，而不是指定数据库产品。另一个不同就是可以选择 Console | New，或者右击资源类型表项并选择 New，就可以很容易地添加资源。

在添加 JDBC 驱动程序之后，就可以定义特定的 JDBC 数据源。每个这样的数据源都是由一个 JNDI 名标识的，并且被映射到一个特定的数据库、表、用户名与口令。

对于 JavaMail 会话，其功能类似于 AEs 中的相应功能。可以指定自己的会话，包括指定输出邮件的设置。也可以像在 AEs 中那样添加 URL 提供者。为此，需要右击 URL Providers 图标，并选择 New 以打开 URL Provider 资源窗口。在这里，可以定义特定的 URL 并将它们绑定到 JNDI 名，从而可以被服务器方代码所使用。

在 JMS Providers 下，可以为新的 JMS 兼容的消息传递系统添加提供者（与 AEs 不同，在这里并没有被限制为使用支持的产品列表）。对一个新 JMS 提供者，应该指定名称、描述、上下文工厂类名与提供者 URL，如下所示：



同样，在 JMS Providers 下是 JMS 连接工厂与 JMS 目标。可以对这两个列表执行添加、删除与编辑操作。对于每个列表，都需要指定一个显示名、描述、JNDI 名、可选的自由文本类型以及附加的自定义属性。相关联的 JMS 提供者作为只读属性被显示（它与在其下创建连接工厂的提供者相匹配）。对于目标，需要指定目标类型（是话题还是队列），对于连接，需要指定连接类型（是点对点连接还是发布/订阅连接）。

J2C 资源适配器为 WebSphere 提供了用来与遗留系统或 EIS（Enterprise Information System，企业信息系统）交互的方法。对于 J2C 适配器，需要指定一个名称和它所在的档案文件（一个 .rar 文件），以及 classpath、描述和可选的特定厂商属性。

Server Groups 文件夹

AE 与 AEs 之间最主要的区别就是 AE 支持多服务器（毕竟，小写字母 s 代表 Single-Server（单服务器））。为了减轻大型网站的管理工作，AE 引入了服务器组（server group）的概念，服务器组是应用程序服务器实例的集合，这些应用程序服务器协同工作并向外界提供一个单独应用程序服务器的功能。这种重复能够大大提高性能与可靠性。服务器组具有与一个单独应用程序服务器相同的属性，此外，它还有下列附加的属性（当单击一个服务器组时，在服务器组属性编辑器中可以见到这些属性）。

- Server Group Name：显示这个组的名称。
- Workload Management Policy：将客户绑定到这个组的服务器上。有下列选项：
 - Random：每个请求被随机地分配给一个服务器。
 - Round Robin：应用程序服务器被随机地排序，客户根据这个应用程序服务器的排序列表绑定到服务器上，当列表中的所有服务器都被选择时，整个列表将重新开始。当然，如果这个序列中的某个服务器实例不可用，那么这个服务器将被跳过。
 - Random Prefer Local：指定除非请求可以被本地服务（在这种情况下，将随机地选择一个本地应用程序服务器），否则就基于随机选择方式进行分配。
 - Round Robin Prefer Locally：为本地应用程序服务器组织请求。如果没有本地应用程序服务器能够为给定的请求服务，那么将根据 round-robin（循环）策略选择一个远程主机。

在使用这些设置创建一个服务器组之后，它还不能使用。因为还没有在这个组中参与工作

的实际应用程序服务器。为了添加这些应用程序服务器实例，需要右击服务器组图标并选择 New | Clone。然后，为这个克隆选择一个名称，并且选择安装这个克隆的节点。在 WebSphere 中，克隆 (clone) 是指作为服务器组的一部分而运行的应用程序服务器实例。

42.5 AE Administrative Console 向导

AE Administrative Console 提供了许多向导，可用来减轻管理 WebSphere 安装的任务。本节将详细讨论这些向导。

42.5.1 Install Enterprise Application Wizard

Install Enterprise Application Wizard 可以指引用户完成部署一个新应用程序的全过程。出现的第一个窗口类似于 AEs 为安装新应用程序所提供的窗口，如图 42-8 所示。

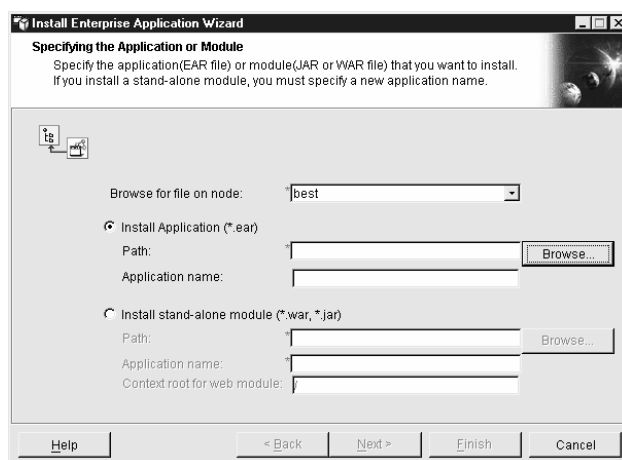


图 42-8 Install Enterprise Application Wizard

用户可以选择 EAR 文件，或者选择一个独立的 (Web 或 EJB) 模块。如果选择独立的模块，WebSphere 将创建一个包含新模块的企业应用程序。在下一个窗口中，可以指定应用程序中的逻辑角色与实际用户之间的映射。

通过为每个角色选择一系列用户与/或组来定义映射。可以选择不进行认证，或者允许所有被认证的用户都可以充当某种特定的角色。

在下一个窗口中，需要将 EJB RunAs 角色映射到用户。RunAs 角色是为特定的 EJB 方法定义的角色 (在 EJB 方法扩展的 Advanced 标签下)。可以映射 RunAs 角色 (其中包括在先前窗口中出现的相同角色定义) 到一个单独的用户。

在定义了与安全性相关的配置之后，下一个窗口就涉及 EJB 引用。在 AAT 中，有一个模块

可以定义指向模块外部的 bean 的 EJB 引用。在 AAT 中，这些引用被绑定在特定的 bean 上，但在 Administrative Console 中，可以修改这些绑定。可以将一个 bean 引用绑定到在服务器上安装的 EJB 上，或者绑定到一个自由文本的 JNDI 名称上。

与 EJB 引用非常相似，一个模块可以包含对 URL、数据源、JavaMail 或 JMS 对象的资源引用。这个向导的下一步就是将这样的引用绑定到各自类型的已安装资源。在继续下一步工作之前，必须进行这些引用的绑定。

下一步工作就是选择默认数据源，以防存在包含带 CMP（Container Managed Persistence，容器管理的持续性）的 EJB 这样的模块。在这里，可以为单独的 bean 选择数据源。

在这个向导的最后两步，应该决定应用程序放置在何处。首先，需要为每个 Web 模块选择一个虚拟主机。然后，需要为应用程序中的每个模块选择应用程序服务器实例。默认情况下，所有的模块都被设置为在默认服务器上运行。这时就可以单击 Finish 按钮了，如果单击 Next 按钮，那么将出现一个确认窗口，这是该向导的最后一步。

42.5.2 Create Application Server Wizard

图 42-9 显示了 Create Application Server Wizard，它在启动时给出一个窗口，可以在这个窗口中指定服务器名以及安装应用程序服务器的节点。

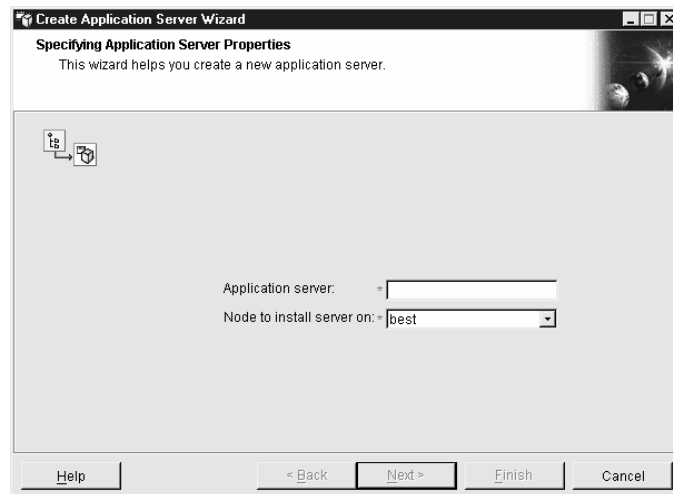


图 42-9 Create Application Server Wizard 的打开窗口

下一步，就能够编辑服务属性。其过程与编辑应用程序服务器的 Services 标签相同。最后一个步骤就是给出一个确认窗口。

42.5.3 Datasource Wizard

用于创建新数据源的 Datasource Wizard，在启动时显示的窗口如图 42-10 所示，在这个窗口

中可以指定数据源的名称与描述，也可以指定目标数据库的名称。

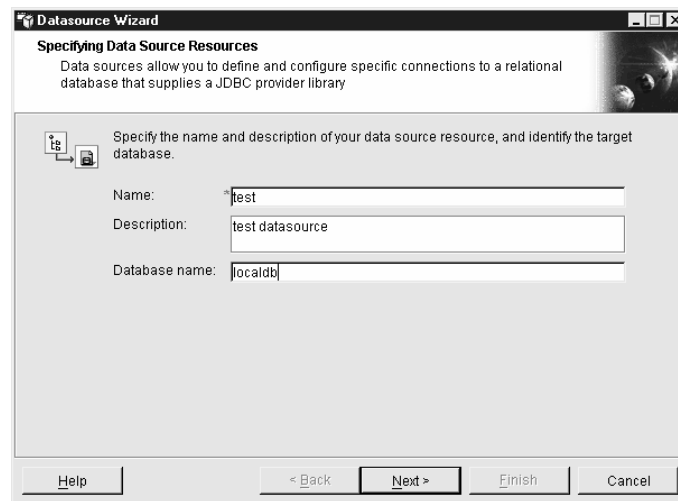


图 42-10 Datasource Wizard 的打开窗口

然后，就需要选择 JDBC 提供者。可以选择一个已有的提供者（在主窗口的 Resources 文件夹下可用），或者指定一个新提供者。

对于新 JDBC 提供者，必须指定名称、描述与实现类。不论是选择已有的 JDBC 提供者，还是为一个新 JDBC 提供者指定了参数，确认窗口都将为这个向导完成动作序列。

42.5.4 JMS Resources Wizard

用于创建 JMS 资源的向导要求用户选择希望创建的资源的类型：连接工厂还是目标。然后，将要求用户输入资源的参数并选择 JMS 提供者。类似于 Datasource Wizard，这个向导允许用户定义一个新 JMS 提供者。在选择 JMS 提供者之后，向导将显示确认窗口。

42.5.5 J2C Connection Factory Wizard

J2C Connection Factory Wizard 的工作流程类似于 Datasource 与 JMS 向导。首先为连接工厂选择名称与描述。然后，选择一个资源适配器，或者定义自己的资源适配器。当定义一个新适配器时，必须指定名称、描述与完全路径。

42.5.6 URL Resource Wizard

URL Resource Wizard 允许用户定义一个 URL，并且为这个 URL 指定名称与描述，如图 42-11 所示。

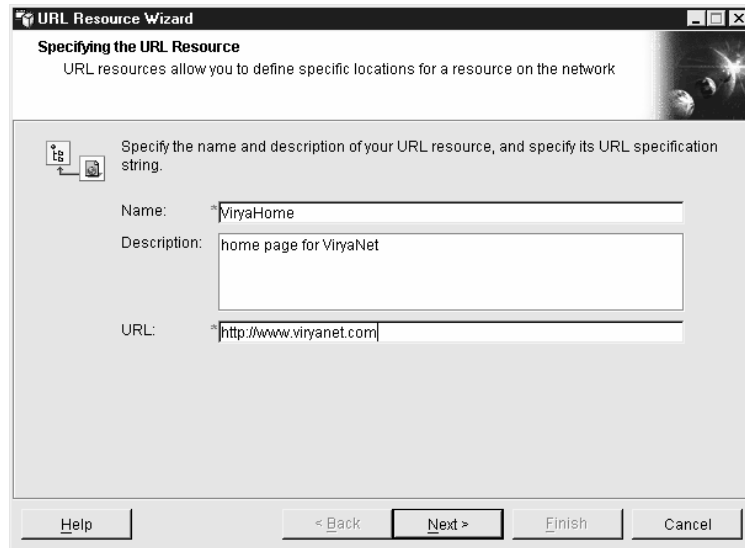


图 42-11 URL Resource Wizard

在单击 **Finish** 按钮之后，就创建了一个新 URL；它的 JNDI 名是 `url/name`，其中 `name` 就是为这个新 URL 指定的名称。

42.5.7 Performance Tuner

Performance Tuner Wizard 可以指引用户完成应用程序服务器的非常重要的性能相关设置。这个向导的第一步就是选择希望进行性能调整的应用程序服务器或服务器组。

让人感兴趣的第一个参数就是 Web 容器的线程池大小。这个向导显示了一个对话框，在这个对话框中可以指定池中的最大线程数。

下一个对话框用于 ORB。可以在这里选择它的线程池大小，并定义使用 RMI/IIOP (Remote Method Invocation-IIOP) 请求的对象将按照引用被传递。在使用后一个选项时应该非常小心，只有当确信它适合于自己的应用程序时才使用它，因为它破坏了远程调用的语义。特别地，它允许 EJB 方法修改调用方的对象。

后面的几个步骤与数据库连接相关。在这里，可以选择一个数据源，然后调整与其相关联的连接池大小和 prepared 语句 cache 大小。在调整数据库参数之后，进一步将允许用户调整 JMS 堆大小。

42.5.8 Create Server Group Wizard

这个向导可帮助用户创建一个服务器组。它指引用户完成指定服务器组名称以及指定负载管理策略的过程；然后，可以选择服务属性，最后为这个组创建克隆。为了节省时间，可以基于一个已有的应用程序服务器设置来创建新服务器组。或者，可以选择复选框（如图

42-12 所示) 并选择一个已有的应用程序服务器实例。

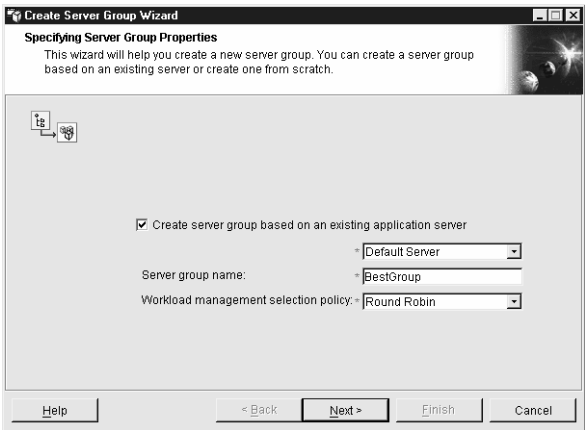


图 42-12 选择一个已有的应用程序服务器

42.6 AE Resource Analyzer

WebSphere Application Server AE 提供了用于监视服务器以及应用程序的工具，以确定潜在的瓶颈。Resource Analyzer 可用来获取任何所请求的监视的统计信息。图 42-13 显示了 Resource Analyzer 窗口。

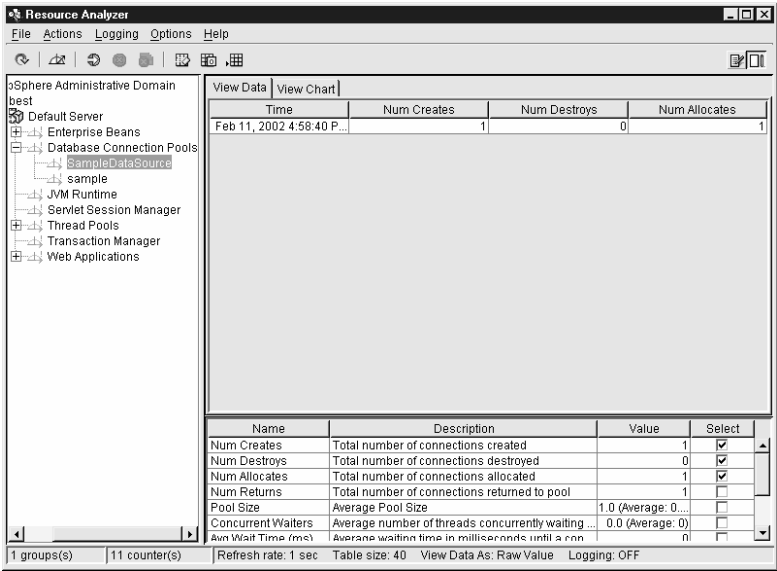


图 42-13 Resource Analyzer

Resource Analyzer 连续不断地为所请求的计数器对每个被监视的元素进行轮询。然后，它将以表格的形式给出计数结果。可以使用左边的树形控件浏览各种被监视的元素。树中的表项包括服务器组件以及应用程序组件。在每个表项的图标旁边，有一个小图标用来显示它的当前监视级别。

在监视性能时，为了使监视开销最小化，应该在另一台单独的机器上启动 Resource Analyzer，并且将它指向被监视的应用程序服务器。

可以单击某个元素在窗口底部的表格显示中的选项，从而过滤掉这个元素的某些信息。

Resource Analyzer 能够提供图形，用来显示所选择元素的计数进程。这些图形可以帮助用户确定性能问题与瓶颈。

42.7 小 结

本章分析了为管理员提供的工具。WebSphere AE（或者是 AEs）可用来部署应用程序。为了有效地建立应用程序与管理服务器，管理员需要使用一系列工具。理解能够利用这些工具做什么事情，这对于管理员是非常重要的。即使应用程序的开发过程很好，但除非进行了正确的配置，否则也不能为用户提供这个应用程序的功能。

WebSphere 的每个主要版本都极大地改进了所提供的部署模型与管理工具。从 3.5 版本到 4.0 版本的飞跃就很好地证明了这一点；服务器模型与 Administrative Console 都在 4.0 版本中得到了改进。控制台具有更多的功能并且更直观。它也遵守 J2EE 模型，并且更加可靠，执行性能也更好。

这是到目前为止本书最长的一章——这可能是因为，无论使用哪种技术来构建应用程序，都需要管理部署。本章提供了利用控制台可以管理的所有组件的概述，并且我们力图提供对所有管理元素的概述。在进行日常的管理工作时，这一章也可以作为参考。

现在，已经学完了本书关于 WebSphere 管理的讨论。第 43 章将继续讨论脚本环境与安全性管理的使用。

第 43 章 WebSphere 管理脚本

在第 42 章，我们学习了如何使用管理客户来配置与管理 WebSphere 服务器。无论正在使用哪个版本的 WebSphere（也即，无论正在使用哪个版本的管理工具），使用这个工具进行日常的管理任务都存在优缺点。其优点就是这个工具具有直观性，易于使用，并且具有一个好的用户界面。但对于重复性工作（无论需要完成的工作是由许多任务组成，还是需要执行一系列重复的任务（例如，每周一次或每天一次）），这种启动工具、导航到正确的位置然后执行任务的过程都是很繁琐的。

43.1 高级版本中的管理工具

配置与管理任务常常是重复的，并且通常是机械性的。为了支持这种配置与管理任务，WebSphere 发布包括两个重要的工具：WSCP（WebSphere Control Program，WebSphere 控制程序）与 XMLConfig 工具。WSCP 在 Tcl 脚本编程环境上构建，这个环境允许使用一系列用 Tcl 定义的对象与动作来管理 WebSphere。这个工具允许使用脚本语言执行任何配置与管理动作。由于 Tcl 是一种功能强大的语言，并且被许多系统管理员所熟知，因此它是用于管理 WebSphere 服务器的有用工具。作为一个脚本编程环境，WSCP 可用来创建脚本文件，从而对当前的任务重复不断地使用这个文件，并且由于 WSCP 程序是从命令行激活的，因此可以使用工具（例如 UNIX 上的 cron）调度自动执行管理任务。

第二款工具是 XMLConfig，自从 WebSphere 3.x 产品以来这个工具就存在。这个工具使用插入到 XML 文件中的声明来支持对服务器的配置与管理。这个 XML 文件定义了应该对哪个工具进行动作以及应该执行什么动作。这个工具对于重复性任务以及复合任务也非常有用。与 WSCP 一样，这个工具也是从命令行激活的。XMLConfig 提供了备份、恢复与重复管理仓库的方法，在管理仓库中保存了所有的 WebSphere 配置信息。

每个工具都特别适合于某种目的。WSCP 是一个脚本编程环境，因此它在本质上是过程性的。相反，XMLConfig 遵守一个声明范型，这对于大多数人的使用有一定的困难，但如果需要在服务器中创建对象的结构时，这个工具就非常有用。当然，也不能忽略了 GUI 工具。所有这些工具都与管理服务器交互，并且最终操纵管理数据库（如图 43-1 所示），它就是控制应用程序服务器的地方。本章将描述 WSCP 与 XMLConfig，从而继续讨论关于配置与管理的一般话题。在精通了 WebSphere 管理之后，读者将会发现，自己几乎很少使用 GUI 工具，相反，主要是使用本章描述的工具之一进行工作。

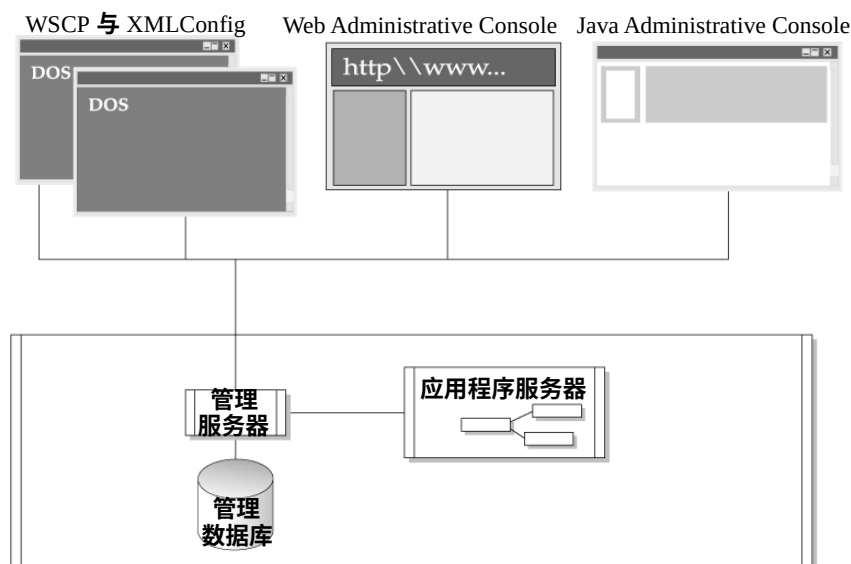


图 43-1 操纵管理数据库

43.2 WebSphere 控制程序

WebSphere 控制程序 (WebSphere Control Program) 是一个 Tcl 程序。在 Windows 下运行 wscp.bat 文件，或者在 UNIX 下运行 wscp.sh 文件，将激活这个程序；这些文件在 WebSphere/AppServer/bin 目录中。WSCP 可以被调用作为一个交互性程序。如果输入 wscp，用户就进入了一个交互式 shell，在这里可以发出命令，从而配置、管理与显示在应用程序服务器中所管理的对象。

但是，这并不是 WSCP 的典型用法；毕竟，如果希望以交互式会话的方式管理服务器，那么使用管理 GUI 则更好。WSCP 程序的一个更常用用法是使用一个文件，文件中包含有应该被这个程序所运行的命令。这个文件在一次构建好了之后，就可以被多次使用。这样就使得执行重复性任务的效率更高，并且能够很容易地对已经构建的过程进行细微的修改。为了在运行 WSCP 时使用一个文件，应该使用 wscp -f <file_name> 选项。可以使用 wscp -h 显示所有的命令行选项，如下所示：

```
D:\WebSphere\AppServer\bin>wscp -h
```

```
Usage: wscp [-h] [-c command] [-f file] [-p file] [-x extension] [-- options]
```

```
-c command : evaluate the specified Tcl command
-h          : display this usage information
-f file     : evaluate the specified file of Tcl commands
```



```
-p file      : evaluate the specified properties file
-x extension: load the specified Tcl extension class
-- options   : Tcl argc and argv variables are set as specified
```

The WSCP shell evaluates Tcl commands in the order specified on the command line, so any extensions should be loaded prior to invoking commands dependent on those extensions. If no command-line commands or files are specified, an interactive shell is invoked, which is terminated by the exit command. Command-line options not supported by WSCP or specified after -- on the command line, are used to set the Tcl argc and argv variables, which may be interpreted by Tcl extensions or other commands.

其中，-f 选项可用来控制哪些命令进入 WSCP 会话。-p 选项可用来设置将影响 WSCP 会话环境的属性。这个选项的最常见用法就是设置管理服务器，可以通过 WSCP 会话对这个管理服务器进行配置。默认情况下，使用的是相同机器上的管理服务器。但是，可以很容易地管理另一台机器上的应用程序服务器。为此，需要设置 wscp.hostname 属性。

例如，假设需要管理在两台不同机器（假设名称为 host1.mydomain.com 与 host2.mydomain.com）上的两个服务器，如图 43-2 所示。

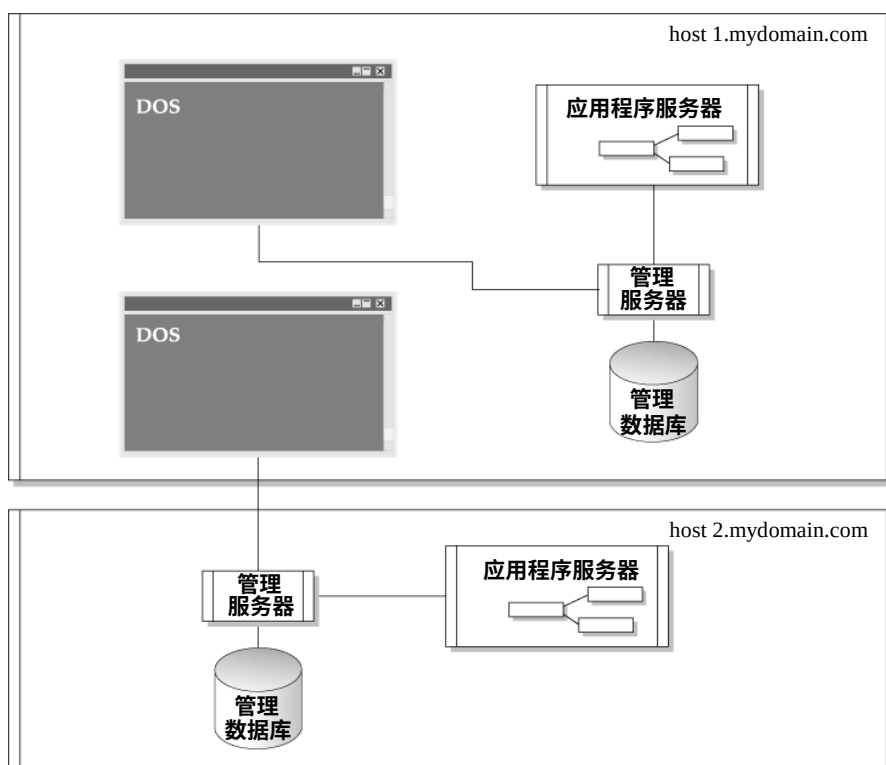


图 43-2 使用 WSCP 管理多个服务器

假设这两个服务器是相同的，并且要在两个服务器上执行相同的操作。同时，还假设服务器上的所有对象都是完全相同的，并且惟一的区别就是主机名内部所管理的对象名不同。在这种情况下，就需要将所有 WSCP 命令放置到一个称为 `wscp_commands1.tcl` 的文件中。如果这个文件是为 `host1.mydomain.com` 主机提供命令的文件，那么在将这个文件中的 `host1.mydomain.com` 替换为 `host2.mydomain.com` 之后，为第二个主机提供的文件名就是 `wscp_commands2.tcl`。然后，创建一个称为 `host2.prop` 的文件，在这个文件中包含有一个单独的行，如下所示：

```
wscp.hostname=host2.mydomain.com
```

然后，在 `host1.mydomain.com` 上运行下面 2 个 WSCP 会话，如图 43-2 所示。

```
wscp -f wscp_commands1.tcl
wscp -f wscp_commands2.tcl -p host2.prop
```

43.2.1 WSCP 对象与动作

WebSphere 服务器由一系列对象组成。这些对象包括应用程序服务器、节点、servlet、EJB 等等。每个这样的对象都可以通过发出命令由 WSCP 会话所管理；每个命令就是应用于对象上的动作。每个对象有一个类型，并且每个类型都有一系列对象可作为这个类型的实例。每个命令以类型名与动作名开始。如果这个动作将应用于一个实例，那么紧接着应该给出这个实例的标识符。如果动作应用于类型对象（或应用于一个外延），那么就不需要实例标识符。WSCP 所管理的对象类型以及 WSCP 会话中每个命令的格式如下所示：

```
D:\WebSphere\AppServer\bin>wscp
wscp> Help
```

```
The general format of all WSCP actions is:
    <object-type> <action> [name] [options]
```

```
The following is a list of the supported objects:
```

```
ApplicationServer
Context
DataSource
DrAdmin
EnterpriseApp
GenericServer
Help
J2CConnectionFactory
J2CResourceAdapter
JDBCdriver
```

```
JMSConnectionFactory
JMSDestination
JMSProvider
MailSession
Module
Node
PmiService
Remote
SecurityConfig
SecurityRoleAssignment
ServerGroup
URL
URLProvider
VirtualHost
XMLConfig
```

To list all actions an object supports: <object> help

To list all the options for an action: <object> help <action>

For verbose information on an action: <object> help <action> -verbose

有些动作被发出并作用于一个特定的对象，而有些动作则作用于外延。外延 (extent) 是具有某种类型的所有实例的集合。实际上，外延与类型是一个相同的概念，表现在可以将类型当作一个对象——某些动作被应用于类型（或外延），另有一些动作被应用于实例。应用于类型（或外延）对象的一个例子就是用来列出具有某种类型的所有实例的命令。应用于类型对象的另一个例子就是使用 `create` 动作来创建一个具有某种类型的新对象。在这种情况下，类型对象（或外延对象）实际上是充当一个工厂对象。

如果想要在一个实例上执行某个动作，那么应该指定类型名与动作，还需要指定这个对象的惟一标识符。例如，使用下列代码查看 `host1` 节点的所有属性：

```
wscp> Node show /Node:host1/ -all
```

```
{FullName /Node:host1/} {Name host1} {CurrentState Running} {DesiredState
Running} {StartTime 1005070676469} {DependentClasspath
{}}{DeployedJar Directory {d:\WebSphere\AppServer\deployedEJBs}} {HostName
host1} {HostSystemType x86} {InstallRoot {d:\WebSphere\AppServer}}
{ProcessId 1756}
```

每个对象类型都有一个属性集合。服务器配置就包括对这些属性设置值。属性遵守一个名称-值范型，但值可以是一个复合类型。一个例子就是 `MIME` 类型的表，它是一个与 `VirtualHost` 对象的属性相关联的值。每个属性都有一个默认值 如果创建了某个类型的对象，属性的初始值就是由这个默认值所定义的。为了查看某个对象类型的所有属性及其默认值，应该使用下列动作：

```
wscp> VirtualHost attributes
```

```
FullName Name AliasList IsDefault MimeTable
```

```
wscp> VirtualHost defaults
```

```
{IsDefault False} {MimeTable {{wk4 application/vnd.lotus-1-2-3}{wk3
application/vnd.lotus-1-2-3}{aiff audio/x-aiff}{ief image/ief}{txt
text/plain}{wk1 application/x-123}{aifc audio/x-aiff}{xsl
application/xml}{cc text/plain}{mpg video/mpeg}{SOL application/solids}
{mpe video/mpeg}{asm text/x-asm}{t application/x-troff}{cpio application/
x-cpio}{s text/css}{UNV application/i-deas}{htmls text/x-ssi-html}{tiff
image/tiff}{CCAD application/clariscad}{mjpg video/x-motion-jpeg}{sh
application/x-sh}{rtx text/richtext}{jpeg image/jpeg}{xwd image/
x-xwindowdump}{VDA application/vda}{m text/plain}{snd audio/basic}{gif
image/gif}{h text/plain}{for text/plain}{mov video/quicktime}{xbm image/
x-xbitmap}{uil text/x-uil}{src application/x-wais-source}{STEP
application/STEP}{dtd text/xml}{dxf application/dxf}{js application/
x-javascript}{c text/plain}{tcl application/x-tcl}{tsv text/tab-separated-
values}{set application/set}{rtf application/x-rtf}{MPG video/mpeg}{sam
application/vnd.lotus-wordpro}{MPE video/mpeg}{IGES application/iges}
{pdf application/pdf}{ppm image/x-portable-pixmap}{zip multipart/x-zip}
{shar application/x-shar}{nc application/x-netcdf}{avi video/x-msvideo}
{mpeg video/mpeg}{au audio/basic}{list text/plain}{DXF application/dxf}
{mime www/mime}{igs application/iges}{org application/vnd.lotus-
organizer}{sv4crc application/x-sv4crc}{SET application/set}{latex
application/x-latex}{ksh application/x-ksh}{dwg application/acad}{rgb
image/x-rgb}{gtar application/x-gtar}{C text/plain}{etx text/x-setext}
{pclapplication/x-pcl}{ms application/x-troff-ms}{ai application/
postscript}{eps application/postscript}{qt video/quicktime}{pgm image/
x-portable-graymap}{java text/plain}{conf text/plain}{vew application/
vnd.lotus-approach}{xml text/xml}{uu text/x-uuencode}{bsh application/
x-bsh}{aif audio/x-aiff}{step application/STEP}{sv4cpio application/
x-sv4cpio}{me application/x-troff-me}{stp application/STEP}{movie video/
x-sgi-movie}{123 application/vnd.lotus-1-2-3}{mar text/plain}{IGS
application/iges}{*. * www/unknown}{tex application/x-tex}{stl
application/SLA}{cxxxtext/plain}{DWG application/acad}{man application/
x-troff-man}{cdf application/x-netcdf}{tar multipart/x-tar}{* www/
unknown}{dvi application/x-dvi}{iges application/iges}{JPEG image/jpeg}
{prz application/vnd.lotus-freelance}{pbm image/x-portable-bitmap}{ustar
multipart/x-ustar}{sdml text/plain}{DRW application/drafting}{der
application/x-x509-ca-cert}{jpg image/jpeg}{ps application/postscript}
{prt application/pro_eng}{xpm image/x-xpixmap}{jpe image/jpeg}{apr
```

```

application/vnd.lotus-approach}{prs application/x-freelance}{scm
application/vnd.lotus-screencam}{lst text/plain}{tr application/
x-troff}{ras image/cmu-raster}{hh text/plain}{pnm image/x-portable-
anymap}{tif image/tiff}{STP application/STEP}{PS application/
postscript}{shtml text/x-ssi-html}{hdf application/x-hdf}{roff
application/x-troff}{or3 application/vnd.lotus-organizer}{bcpio
application/x-bcpio}{lwp application/vnd.lotus-wordpro}{STL
application/SLA}{def text/plain}{or2 application/vnd.lotus-organizer}
{wrl x-world/x-vrml}{htm text/html}{log text/plain}{class application/
octet-stream}{pre application/vnd.lotus-freelance}{f90 text/plain}{unv
application/i-deas}{oda application/oda}{gz multipart/x-gzip}{css text/
css}{com text/plain}{texinfo application/x-texinfo}{wav audio/x-wav}{vda
application/vda}{bin application/octet-stream}{texi application/
x-texinfo}{bmp image/bmp}{JPG image/jpeg}{PRT application/pro_eng}{JPE
image/jpeg}{csh application/x-csh}{MPEG video/mpeg}{html text/html}}

```

每个动作都是对象类型定义的一部分。不同的对象类型可能有不同的动作。为了查看某个类型的可用动作有哪些，应该使用下列帮助命令：

```

wscp> ApplicationServer help
The following actions are available for ApplicationServer

```

attributes	Display the attributes of the object
containment	Display the containment hierarchy for the object
create	Create the specified object
defaults	Display or set attribute defaults
help	Display this help message
list	Display all the instances of this type
modify	Modify the attributes of the specified object
operations	List all the actions available on the object type
remove	Remove the specified object
show	Display the attributes of specified object
start	Start the specified object
stop	Stop the specified object

```

wscp> Context help
The following actions are available for Context

```

bind	Bind the object to the specified name
cd	Change directory to the specified name
create	Create the specified object
env	List the current environment
help	Display this help message

init	Create the initial context
list	List contents of the context
operations	List all the actions available on the object type
rebind	Rebind the object to the specified name
remove	Remove the specified object
rename	Rename the object to the specified name
unbind	Unbind the specified object

```
wscp> JDBCDriver help
```

The following actions are available for JDBCDriver

attributes	Display the attributes of the object
containment	Display the containment hierarchy for the object
create	Create the specified object
defaults	Display or set attribute defaults
help	Display this help message
install	Install a JDBC driver
list	Display all the instances of this type
modify	Modify the attributes of the specified object
operations	List all the actions available on the object type
remove	Remove the specified object
show	Display the attributes of specified object
uninstall	Uninstall the JDBC driver

有些动作在 WSCP 中是对于所有的对象类型可用的。例如，对一个单独对象进行操作的动作（如 `remove` 动作）以及那些对外延对象进行操作的动作（如 `list` 动作）。此外，不同的对象类型还有特定于这个对象的动作。这些动作就是对一个对象进行操作的动作，例如 `start` 动作与 `stop` 动作，这两个动作可分别用来启动与停止某个应用程序服务器。

43.2.2 一个 WSCP 示例

WSCP 是一个强大的工具，一旦掌握了 Tcl 语法，就能够使用它来非常容易地管理整个环境。实际上，就我们的经验来看，只要有一定 Tcl 背景的管理员都倾向于在 WSCP 中完成所有的管理工作，而不管所管理的是 WebSphere 3.5 还是 4.0。我们的另一个经验就是，从长远角度看，开发复杂的脚本能够节省大量的管理 WebSphere 的时间。在开始讨论 XMLConfig 之前，将描述一个使用 WSCP 的简单例子。

首先为系统定义虚拟主机。为此需要使用工厂对象创建一个新虚拟主机对象，然后设置其必需的属性。在这里需要设置的就是这个主机的别名列表，如下所示：

```
wscp> VirtualHost create /VirtualHost:vhost/ -attribute {{AliasList {host1
host1.mydomain.com 127.0.0.1 10.10.10.40}}}
```

然后，需要创建 JDBC 驱动程序并安装它。创建驱动程序的工作也需要使用工厂对象。安装驱动程序实际上就是应用于新创建的对象的动作，因此需要使用在驱动程序创建过程中给出的标识符。这里，将使用一个 Oracle 瘦驱动程序，因此创建与安装这个驱动程序的代码形式如下：

```
wscp> JDBCDriver create /JDBCDriver:myOracleDriver/ -attribute {{ImplClass  
oracle.jdbc.driver.OracleDriver} {Description "Oracle Thin Driver"}}
```

```
wscp> JDBCDriver install /JDBCDriver:myOracleDriver/ -node /Node:host1/  
-jarfile "f:/WebSphere/Appserver/lib/classes111.zip"
```

接下来，需要创建应用程序服务器实例并为这个应用程序服务器配置会话管理属性。需要说明的是，在 4.0 版本中，会话管理属性是应用程序服务器的一部分；在 3.x 版本中，它们是 SessionManager 对象的一部分，这个对象在 4.0 版本中不再存在。

```
Wscp> ApplicationServer create /Node:host1/ApplicationServer:myAppServer/  
-attribute {{JVMConfig {{InitialHeapSize 64} {MaxHeapSize 128}}}   
{WebContainerConfig {SessionManagerConfig {{EnablePersistentSessions false}  
{TuningScheduleInvalidation true} {TuningInvalidationTimeout 15}}}}
```

使用 WSCP 创建克隆是一件比较简单的事情。ServerGroup 对象有一个用于创建克隆的动作。例如，如果按照前面的列表定义了一个应用程序服务器，那么可以从其中创建一个服务器组对象，然后在这个服务器组上使用 clone 动作。在下面代码样例中，将创建一个名为 Clone1 的服务器组，然后创建一个名为 Clone2 的克隆：

```
wscp> ServerGroup create /ServerGroup:Clone1/ -baseInstance  
/Node:host1/ApplicationServer:myAppServer/
```

```
wscp> ServerGroup clone /ServerGroup:Clone1/ -node /Node:host1/ -cloneAttrs  
{{Name Clone2}}
```

在这个例子中，如果再次列出所有的服务器组（克隆），将看到下列内容：

```
wscp> ServerGroup list
```

```
{/Node:host1/ServerGroup:Clone1/} {/Node:host1/ServerGroup:Clone2/}
```

43.3 XMLConfig

XMLConfig 是作为 WebSphere 高级版本一部分的另一个实用程序，早在 WebSphere 3.0 版本中就引入了 XMLConfig。从命令行调用这个实用程序，并且它接受遵守 DTD (Document Type Definition, 文档类型定义) 的 XML 文件，DTD 定义了这个实用程序允许的结构模式。

XMLConfig 可用来完成 WSCP 能够完成的任何任务，包括创建环境、修改属性以及启动与停止服务器。实际上，这两个实用程序可以看作是互补的，因为每个实用程序都有自己更适用的领域。例如，在我们看来，当一个应用程序解决方案被打包作为一个 WebSphere 应用程序并且被发布为产品时，使用 XMLConfig 进行管理就更加容易。这是因为应用程序安装能够包括一个 XMLConfig 文件，这个文件可以被导入到 WebSphere 应用程序服务器中用于部署。另一方面，当一个工程组正在内部开发应用程序并且有一个系统管理员负责管理服务器时，WSCP 就更加常用，因为它更加类似于其他脚本编程环境。我们更喜欢使用 XMLConfig，但这仅仅是个人偏好。

为了调用 XMLConfig，应该到 WebSphere/AppServer/bin 目录中并使用命令行程序 XMLConfig（在 Windows 下是 XMLConfig.bat，在 UNIX 下是 XMLConfig.sh）。可能的参数如下：

```
{ ( -import <xml data file> ) ||  
[ ( -export <xml output file> [-partial <xml data file>] ) ]  
-adminNodeName <primary node name>  
[ -nameServiceHost <host name> [ -nameServicePort <port number> ] ]  
[-traceString <trace spec> [-traceFile <file name>]]  
[-generatePluginCfg <true || false>]  
[-substitute <"key1=value1[;key2=value2[...]]">]]  
In input xml file, the key(s) should appear as $key$ for substitution.
```

我们喜欢使用 XMLConfig 的一个原因就是，它遵守在构建软件过程中所使用的传统方法：复制与粘贴。配置与建立服务器配置的最简单、最安全方法就是从现有的服务器中复制配置，然后按照需要进行适当的修改。XMLConfig 以非常自然的方式支持这一点，因为它允许导出一个 XML 描述文件，对这个文件进行修改，然后导入这个文件。这样，由于总是从一个可以正常工作、具有正确语法的 XML 文件开始，就能够减少对命令进行仔细调试的必要性，只需要对所作的修改进行仔细的检查即可。

43.3.1 XMLConfig 命令

XMLConfig 会话接受一个 XML 文件，这个文件遵守 XMLConfig DTD，在其中定义了一系列将要发出的命令。每个命令都被一个 XML 标记所包装，这个 XML 标记定义了 WebSphere 包容层次中的元素，并且定义了命令本身。命令可以是控制命令（例如 start 与 stop）；结构命令（例如 create 与 update）；或 export 命令。

下面给出了一些控制命令：

- start：启动一个资源（例如节点或应用程序服务器）。
- stop：停止资源。
- restart：停止资源，然后重新启动这个资源。

- disable：挂起一个 Web 应用程序。
- enable：恢复一个 Web 应用程序。
- stopforrestart：可能停止一个节点，以便重新启动。

下面给出了一些结构命令：

- create：在管理服务器中创建一个元素，并且基于这个创建命令在 XML 文件中的位置将所创建的元素放在层次中。
- update：使用 XML 结构中定义的属性与内部元素，更新在管理服务器中已经定义的一个元素。
- delete：从管理服务器中删除一个元素。
- locate：在管理服务器中标识一个元素。用来指向包含层次中的一个元素，从而使得结构命令（例如 create 与 delete）能够在适当的上下文中被执行。例如，如果需要向 Web 应用程序中添加一个资源，那么首先需要定位这个 Web 应用程序，然后使用一个 create 命令。

下面给出了 Export 命令：

- export：将包含层次中的一棵子树导出到一个 XML 导出文件中。

export 命令与结构命令组合使用，就能够支持配置的复制与粘贴。例如，如果需要创建一个附加的企业 bean，就可以导出一个已有的 bean，然后修改某些属性以适合新的类与参数，然后在结构命令（最可能就是 update 命令）的导入模式下使用这个文件。这是目前为止最简单、最安全的创建一个新企业 bean 的方法，我们推荐使用这种方法。

43.3.2 XMLConfig 示例

下面将描述一个与 WSCP 中使用的例子相似的例子。首先，创建一个带有别名列表与 MIME 表的虚拟主机。需要说明的是，即使这个虚拟主机不存在，也可以使用 update 命令；如果一个元素不存在，那么 update 命令将创建这个新元素，如果这个元素已存在，那么这个命令将更新此元素中的值。因此，使用这个命令是最安全的。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE websphere-sa-config SYSTEM
"$XMLConfigDTDLocation$$dsep$xmlconfig.dtd" >
<websphere-sa-config>
  <virtual-host action="update" name="vhost">
    <mime-table >
      <mime type="application/vnd.lotus-1-2-3">
        <ext>wk4</ext>
        <ext>wk3</ext>
```

```

        </mime>
        <mime type="audio/x-aiff">
            <ext>aiff</ext>
        </mime>
        <mime type="image/ief">
            <ext>ief</ext>
        </mime>
        <mime type="text/plain">
            <ext>txt</ext>
        </mime>
        ...
    </mime-table>
    <alias-list>
        <alias>host1</alias>
        <alias>host1.mydomain.com</alias>
        <alias>127.0.0.1</alias>
        <alias>10.10.10.40</alias>
    </alias-list>
</virtual-host>
</websphere-sa-config>

```

然后，就像在 WSCP 示例中一样，创建一个 JDBC 驱动程序：

```

<jdbc-driver action="update" name="myOracleDriver">
    <implementation-class>
        com.oracle.jdbc.driver.OracleDriver
    </implementation-class> <description/>
    <description>Oracle Thin Driver</description>
    <install-info>
        <node-name>host1</node-name>
        <jdbc-zipfile-location>
            f:/WebSphere/Appserver/lib/classes111.zip
        </jdbc-zipfile-location>
    </install-info>
</jdbc-driver>

```

最后，创建这个应用程序服务器并且设置一些调整参数。需要再次说明的是，这里总是使用 update 命令。这能够照顾到创建与更新两种情景。

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE websphere-sa-config SYSTEM
"file:/// $XMLConfigDTDLocation$ $dsep$ xmlconfig.dtd">
<websphere-sa-config>
    <node name="host1" action="locate" >
        <application-server action="update" name="Host1">

```

```
<executable>java</executable>
<command-line-arguments/>
<environment/>
<user-id/>
<group-id/>
<working-directory/>
<umask>18</umask>
<stdin/>
<stdoutf>:/WebSphere/AppServer/logs/host1_stdout.log</stdout>
<stderrf>:/WebSphere/AppServer/logs/host1_stderr.log</stderr>
<process-priority>20</process-priority>
<maximum-startup-attempts>2</maximum-startup-attempts>
<ping-interva>l60</ping-interval>
<ping-timeout>200</ping-timeout>
<ping-initial-timeout>300</ping-initial-timeout>
<selection-policy>roundrobinpreferlocal</selection-policy>
<trace-specification/>
<trace-output/>
<transaction-log-file/>
<olt-enabled>>false</olt-enabled>
<system-properties/>
<debug-enabled>>false</debug-enabled>
<transaction-timeout>120</transaction-timeout>
<transaction-inactivity-timeout>
    60000
</transaction-inactivity-timeout>
<thread-pool-size>20</thread-pool-size>
<security-enabled>>false</security-enabled>
<admin-agent-ior/>
<cache-config>
    <cache-size>4097</cache-size>
    <cache-sweep-interval>1000</cache-sweep-interval>
    <passivation-directory/>
</cache-config>
<log-file-spec/>
<performance-monitor-spec>
    pmi=none:beanModule=maximum:connectionPoolModule=medium:
    j2cModule=medium:jvmRuntimeModule=none:jvmpiModule=low:
    servletSessionsModule=none:threadPoolModule=high:
    transactionModule=none:webAppModule=maximum
</performance-monitor-spec>
<olt-server-host>localhost</olt-server-host>
<olt-server-port>7001</olt-server-port>
<selection-policy>roundrobinpreferlocal</selection-policy>
```

```
<source-path/>
<wlm-template-ior/>
<thread-pool-size>15</thread-pool-size>
<jvm-config>
  <initial-heap-size>64</initial-heap-size>
  <max-heap-size>128</max-heap-size>
  <generated-command-line-arguments>
    -Xms64m -Xmx128m
  </generated-command-line-arguments>
  <system-properties/>
  <additional-command-line-arguments/>
  <debug-mode>false</debug-mode>
  <debug-string/>
  <run-hprof>false</run-hprof>
  <hprof-args/>
  <disable-jit>false</disable-jit>
  <verbose-class-loading>false</verbose-class-loading>
  <verbose-jni>false</verbose-jni>
  <verbose-gc>false</verbose-gc>
  <boot-classpath-replace/>
  <boot-classpath-append/>
  <boot-classpath-prepend/>
</jvm-config>
<web-container>
  <dynamic-cache-config>
    <enabled>false</enabled>
    <cache-size>0</cache-size>
    <default-priority>0</default-priority>
  </dynamic-cache-config>
  <transport name="http">
    <transport-host>*</transport-host>
    <transport-port>7007</transport-port>
    <http-transport>
      <connection-timeout>5</connection-timeout>
      <backlog-connections>50</backlog-connections>
      <keep-alive-timeout>5</keep-alive-timeout>
      <maximum-keep-alive>25</maximum-keep-alive>
      <maximum-req-keep-alive>300</maximum-req-keep-alive>
      <ssl-enabled>false</ssl-enabled>
    </http-transport>
  </transport>
  <thread-maximum-size>50</thread-maximum-size>
  <thread-minimum-size>25</thread-minimum-size>
  <thread-inactivity-timeout>10</thread-inactivity-timeout>
```

```
<thread-is-growable>true</thread-is-growable>
<session-manager>
  <enable-security-integration>
    false
  </enable-security-integration>
  <enable-ssl-tracking>
    false
  </enable-ssl-tracking>
  <invalidation-schedule-first-hour>
    0
  </invalidation-schedule-first-hour>
  <invalidation-schedule-second-hour>
    0
  </invalidation-schedule-second-hour>
  <persistent-db2-row-size>
    0
  </persistent-db2-row-size>
  <maximum-inmemory-session-count>
    1000
  </maximum-inmemory-session-count>
  <write-contents>0</write-contents>
  <write-frequency>0</write-frequency>
  <write-interval>10</write-interval>
  <enable-sessions>>false</enable-sessions>
  <enable-url-rewriting>>false</enable-url-rewriting>
  <enable-cookies>true</enable-cookies>
  <enable-protocol-switch-rewriting>
    false
  </enable-protocol-switch-rewriting>
  <cookie name="JSESSIONID">
    <domain/>
    <maximum>-1</maximum>
    <path>/</path>
    <secure>>false</secure> </cookie>
  <tuning-invalidation-time>15</tuning-invalidation-time>
  <persistent-sessions>>false</persistent-sessions>
  <data-source name="">
    <default-user/>
    <default-password/>
  </data-source>
  <using-multi-row>>false</using-multi-row>
  <allow-overflow>true</allow-overflow>
</session-manager>
</web-container>
```

```

<orb-config>
  <bootstrap-host-name/>
  <bootstrap-port>900</bootstrap-port>
  <comm-trace-enabled>false</comm-trace-enabled>
  <connection-cache-maximum>240</connection-cache-maximum>
  <connection-cache-minimum>100</connection-cache-minimum>
  <external-config-url/>
  <force-tunnel>whenrequired</force-tunnel>
  <listener-port>0</listener-port>
  <locate-request-timeout>180</locate-request-timeout>
  <local-host-name/>
  <lsd-host-name/>
  <no-local-copies>false</no-local-copies>
  <request-retries-count>1</request-retries-count>
  <request-retries-delay>0</request-retries-delay>
  <request-timeout>180</request-timeout>
  <thread-pool-size>20</thread-pool-size>
  <tunnel-agent-url/>
  <rmi-remote-code-base/>
  <ssl-listener-port>0</ssl-listener-port>
</orb-config>
<custom-service-config-list>
  <custom-service-config name="AutoGen">
    <description>Generated when started</description>
    <classname>
com.ibm.websphere.plugincfg.initializers.AEPluginCfgService
    </classname>
    <classpath/>
    <external-config-url/>
    <enabled>false</enabled>
  </custom-service-config>
</custom-service-config-list>
</application-server>
</node>
</websphere-sa-config>

```

43.4 小 结

大多数管理员并不使用 GUI 客户进行连续性的日常管理与配置任务。他们使用可重复的、易于学习的并且易于使用的工具。在大多数复杂的分布式环境下，采用基于脚本的管理是必要的。此外，防火墙、安全性以及性能问题常常导致 Administrative Console 被废弃，而

是在产品环境下使用这些工具。

我们极力推荐使用 WSCP 与 XMLConfig。至于哪一个工具更适合于用户自己，这完全取决于个人偏好，也可以同时使用这两个工具，因为它们都遵守相同的模型。如果用户已经掌握了 Tcl，那么可能更倾向于使用 WSCP。如果用户喜欢 XML 并且能够熟练地使用 XML 编辑器（例如 XML Spy），那么可能更倾向于使用 XMLConfig。但两个工具都不错。

本章介绍了这两个工具，并且解释了主要的管理模型与层次。还有一个没有讨论的话题就是安全性管理。实际上，这个话题比较复杂，因此需要单独一章的篇幅，本书将在第 44 章讨论这个话题。

第 44 章 管理 WebSphere 安全性

欢迎进入本书最后一章的学习，虽然这是最后一章，但这当然不意味着它是最不重要的。应用程序的安全性总是非常重要的，Web 应用程序与服务器的安全性已经成为人们关注的焦点，这不仅因为在 Internet 上发生了许多安全性破坏事件，而且由于人们对安全性问题的日益重视（2001 年 9 月 11 号发生的恐怖袭击，更加剧了人们对安全性的关注）。

本书在第 13 章已经对安全性相关的概念进行了简要的介绍，并且在第 19 章描述了 WebSphere Security Services，这一章将继续讨论 WebSphere 管理员最重要的任务——也就是，在 WebSphere Application Server 上管理安全性。本章将简要说明管理员能够控制的安全性功能，并且说明如何使用 Administrative Console 与 Security Center 工具执行这些安全管理任务。需要说明的是，本章关注的是最常见的安全性管理活动，而不是全面说明安全性管理与配置问题。实际上，IBM 出版了一本非常优秀的红皮书，名为：WebSphere Advanced Edition:Security。这本书对所有的安全性功能与安全性管理问题进行了讨论，有 500 多页的篇幅。

本章讨论的话题包括如何使用口令确保服务器安全、如何建立一个自定义的用户注册表、如何将用户分配到安全性角色上、如何在 DMZ（Demilitarized Zone，隔离区）中加密网络通信等等。其中许多安全性话题已经在其他章（包括第 3 章、第 13 章、第 19 章与第 20 章）讨论，这一章将加深对它们的学习。

44.1 确保应用程序服务器安全

如果没有确保服务器的安全性，那么任何人都可以使用 Administrative Console 屏蔽所创建的任何安全性设置（在此，不讨论机器本身的安全性——包括机器的物理安全性与作为管理员登录的安全性。显然，这是确保应用程序服务器安全的前提）。需要做的第一件事情就是使用 Console 菜单从 Administrative Console 中打开 Security Center。然后，选择 Security Center 选项并打开 General 标签。最后，选中 Enable Security 复选框并应用修改。

如果试图仅仅应用这个修改，系统将会提示给出用户注册表，利用用户注册表可以对用户名/口令进行有效性检查，正如第 19 章讨论的那样。可以做的最简单事情就是使用操作系统的注册表，如图 44-1 所示。

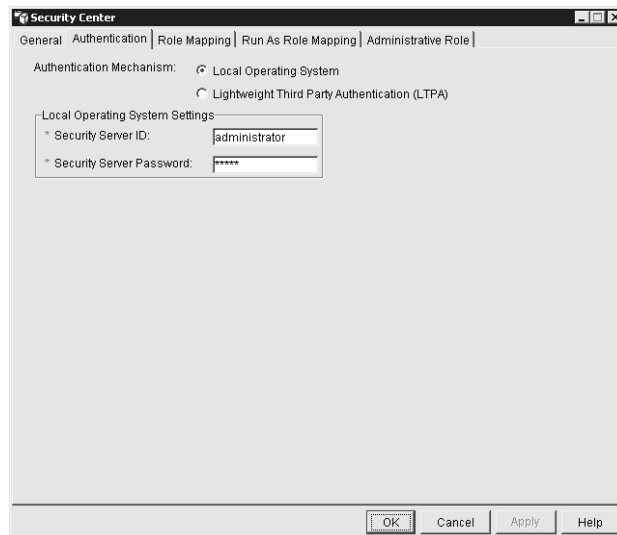
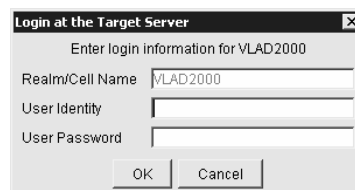


图 44-1 使用操作系统的注册表

这时，需要重新启动管理服务器，管理服务器是安全性服务器所运行的地方。在重新启动管理服务器之后，这个服务器就是拥有安全性保证的。如果试图打开 Administrative Console，将出现一个对话框，要求输入口令：



这样，服务器就能够确保每次访问的安全性。现在已经修改了服务器的全局设置，并且它将影响由这个管理服务器所管理的所有应用程序服务器。即使最著名的 snoop servlet 也同样受影响；如果试图访问它，首先将遇到 Enter Network Password 对话框。

但是，这通常并不是用户想要的。实际上，可以单独地为每个应用程序服务器设置禁用安全性或设置特定安全性功能。为此，需要打开希望不应用安全性访问限制的应用程序服务器的属性表。

然后，需要选择 JVM Settings 标签并修改（或添加）某些属性，如下所示：

```
com.ibm.CORBA.SSLTypeIClientAssociationEnabled → false
com.ibm.CORBA.LocalOSClientAssociationEnabled → false
com.ibm.CORBA.LTPAClientAssociationEnabled → false
com.ibm.CORBA.DCEClientAssociationEnabled → false
com.ibm.CORBA.SSLTypeIServerAssociationEnabled → true
```

`com.ibm.CORBA.LocalOSServerAssociationEnabled` → `true`

`com.ibm.CORBA.LTPAServerAssociationEnabled` → `false`

这些属性假设正在使用的是操作系统注册表；对于 LTPA（Lightweight Third-Party Authentication，轻量级第三方认证），这些属性则相反。最后得到的结果应该如图 44-2 所示。

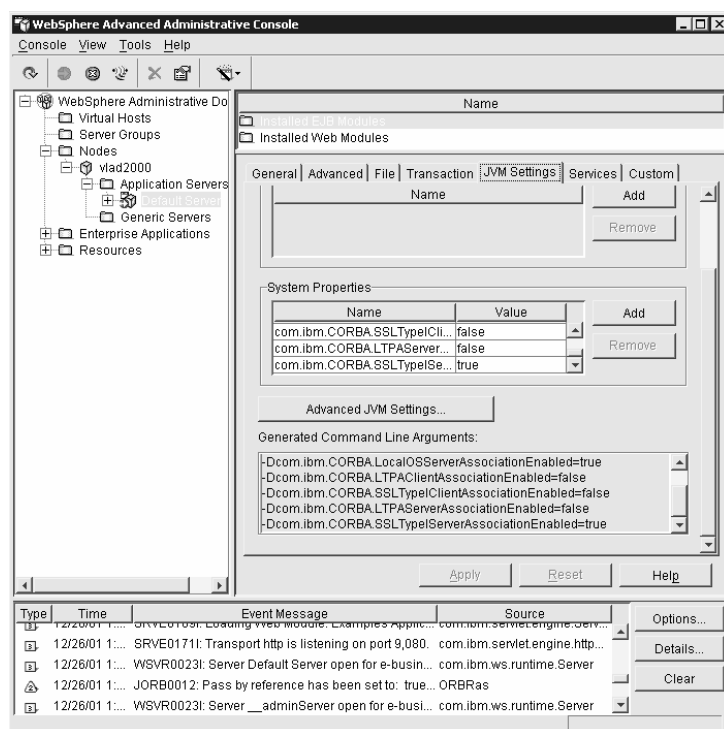


图 44-2 Default Server 的 JVM Settings

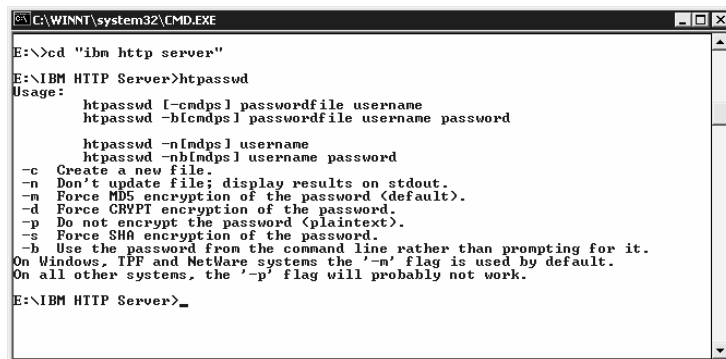
现在，需要使用 Administrative Console 重新启动应用程序服务器（而不是管理服务器）。这时，管理服务器运行在安全模式下，而这个应用程序服务器则运行在非安全模式下。

44.2 为 HTTP Server 管理安全性

第 3 章解释了如何使用 IBM HTTP Server 建立一个安全网站。在那里学习了如何使用 IKEYMAN 实用程序生成一个密钥对，以及如何通过修改配置文件来启用 SSL。在这里不再重复这些内容，因此，在继续学习后面的内容之前重新阅读第 3 章最后一节的内容也许会有所帮助。本章关注的是管理工具以及如何使用这些管理工具来配置安全性。因此，在这里详细说明了如何使用 IBM HTTP Server 的管理控制台（它不同于 WebSphere Administrative Console）确保 HTTP 服务器的安全性以及如何建立 SSL（这里将假设已经生成并安装了 SSL

握手的服务器认证中将要使用的证书)。为了继续学习,请按照下列步骤操作:

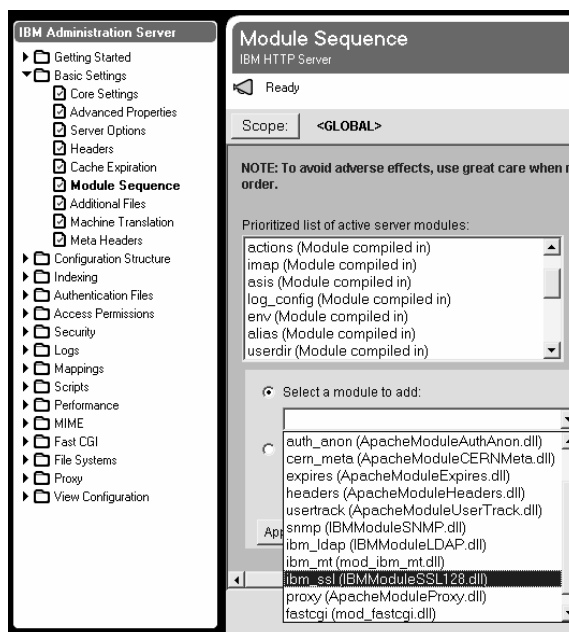
1. 第一件事情就是使用 IBM HTTP Server 的根安装目录下的 `htpasswd` 实用程序设置管理口令。需要修改的文件名为 `admin.passwd`, 它位于 `conf` 目录下。这个实用程序的可能参数如下:



```
C:\WINNT\system32\CMD.EXE
E:\>cd "ibm http server"
E:\IBM HTTP Server>htpasswd
Usage:
    htpasswd [-cmdpsl] passwordfile username
    htpasswd -b[cmdpsl] passwordfile username password

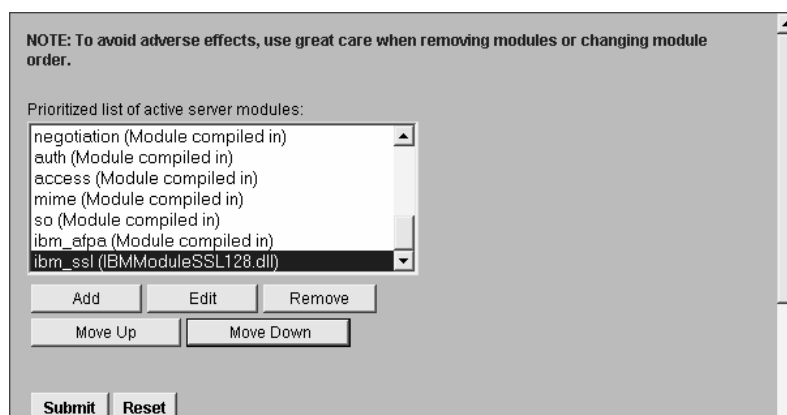
    htpasswd -n[mdpsl] username
    htpasswd -nb[mdpsl] username password
    -c Create a new file.
    -n Don't update file; display results on stdout.
    -m Force MD5 encryption of the password (default).
    -d Force CRYPT encryption of the password.
    -p Do not encrypt the password (plaintext).
    -s Force SHA encryption of the password.
    -b Use the password from the command line rather than prompting for it.
On Windows, TPF and NetWare systems the '-n' flag is used by default.
On all other systems, the '-p' flag will probably not work.
E:\IBM HTTP Server>
```

2. 从 `http://localhost` 开始工作, 它将给出 IBM HTTP Server 的索引页面, 单击 `Configure Server`。使用刚才为服务器设置的用户名与口令。这时, 将出现 HTTP 管理控制台。
3. 加载 SSL 模块。单击树视图上的 `Module Sequence` 选项, 并且单击 `Add` 按钮。然后选择一个将要添加的模块, 从组合框中选择 `ibm_ssl` 模块:

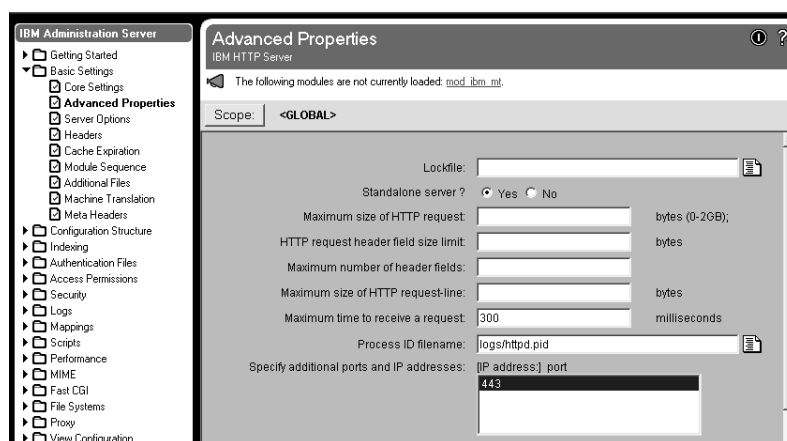


4. 在添加这个模块之后, 可以修改模块加载的顺序。通常, 这样不会影响其他的模块;

但为了安全起见，应该使用 Move Down 按钮确保这个 SSL 模块是最后被加载的：



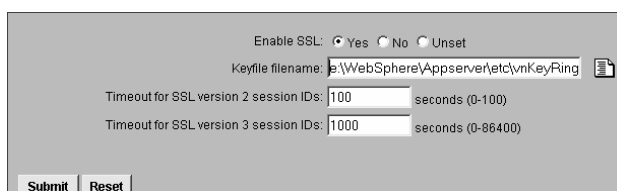
5. 现在需要进行设置以确保主机端口的安全。使用 Basic Settings 选择中的 Advanced Properties 选项，然后单击 Add 按钮并输入安全服务器所使用的端口号。在这里选择的是 443 端口，这个端口是 HTTPS 的默认端口。其结果如下：



6. 现在开始设置虚拟服务器与 SSL 将要在其中被激活的范围。选择 Configuration Structure | Create Scope，并输入虚拟主机名、端口与服务器名。然后向下滚动到中间面板，并单击 Submit。
7. 在创建范围之后，就选择 Basic Settings | Core Settings 以设置文档根（任何虚拟主机都需要一个文档根；这与安全服务器没有直接关系）。
8. 最后，需要为新创建的范围激活这个 SSL。首先，应该确信 SSL 对主服务器是禁用的。并且确信正在工作于 GLOBAL 范围，以及确信 SSL 被禁用。
9. 然后单击 Scope 按钮，并且选择刚刚为安全服务器所创建到的那个范围：



10. 现在执行的每个动作将仅仅影响所选择的范围——也就是，影响正在设置的安全服务器。打开 SSL，并且选择想要使用的密钥文件（这个文件通常由 IKEYMAN 所生成，这在第 3 章已经描述），并且设置超时值：



11. 最后，提交并重新启动这个服务器。现在，就有了一个与普通服务器并行工作的 SSL 安全服务器。

44.3 自定义用户注册表

第 19 章讨论了 WebSphere 安全性模型上下文中的认证。有两个选项可用于认证机制：使用本地操作系统或使用 LTPA。如果决定使用本地操作系统，那么仅仅需要指定用户名与口令（用来访问这个应用程序）以及与这个用户相关联的许可，如图 44-1 所示。如果使用 LTPA，那么需要在两种可能的用户信息源中选择一个，如图 44-3 所示。

LTPA 最典型的使用情况就是当用户信息来自一个 LDAP 服务器时。正如在第 20 章讨论的那样，在这种情况下，需要设置主机位置、端口与基 DN（Distinguished Name，区别名）。这样就允许 WebSphere 服务器与 LDAP 服务器（例如，一个 SecureWay 服务器）通信以认证用户。

或者，也可能需要使用一个自定义注册表，其原因可能是这个注册表已经作为公司标准而存在，或由于其他内部的原因。在这种情况下，需要提供一个类（这个类可能是自己编码实现的，也可能是接收到的），这个类实现了 WebSphere 定义的一个协议。这个类将充当实际注册表信息与 WebSphere 安全性服务器之间的接口。需要利用 Security Center 注册这个类，如图 44-4 所示。这个类实现中所需的属性通过使用 Special Custom Settings 按钮来提供，如图 44-5 所示。

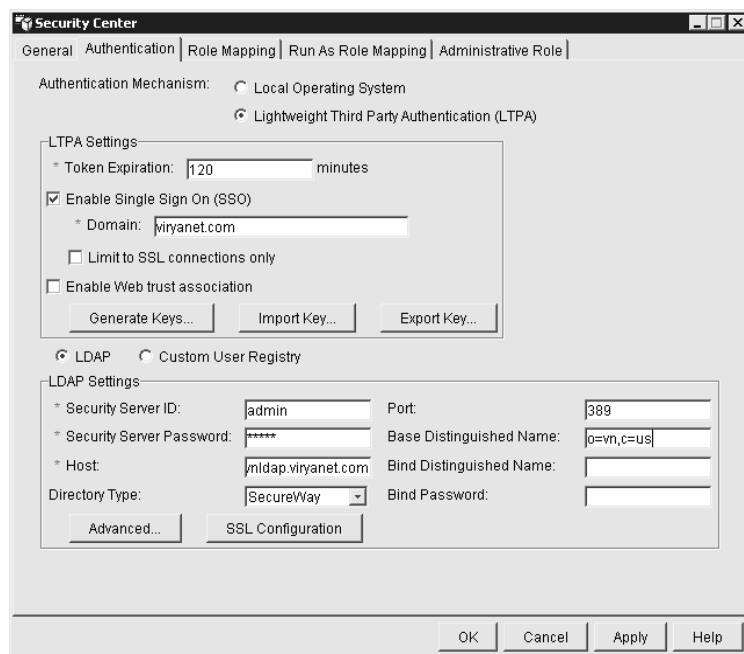


图 44-3 选择使用 LDAP 用户注册表的 LTPA 认证

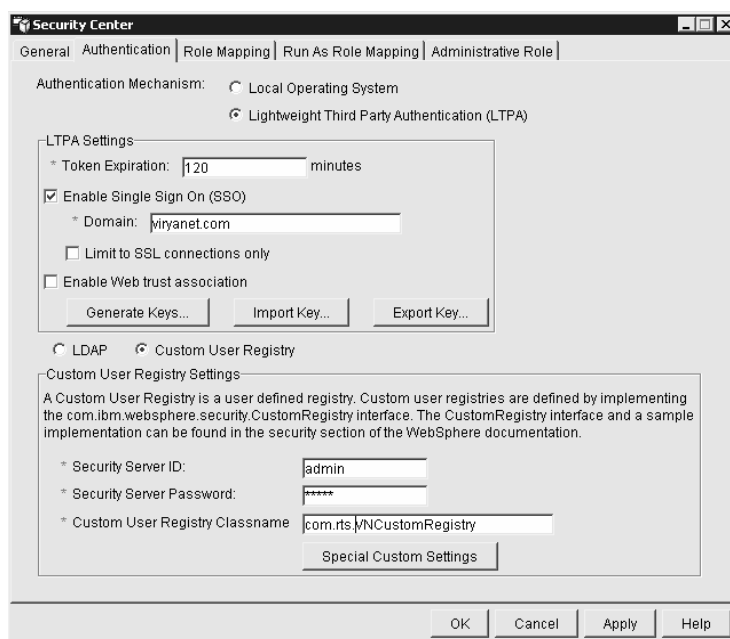


图 44-4 使用自定义用户注册表的 LTPA

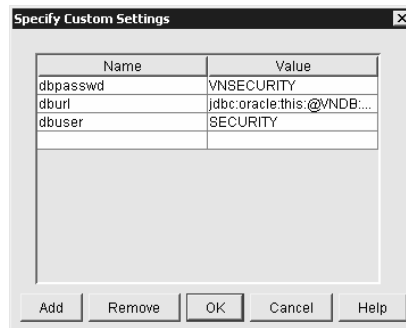


图 44-5 为自定义注册表接口类设置属性

这个类放置在 WebSphere 的 classpath 下，并且被应用程序服务器用来执行认证。协议在 CustomRegistry 接口中定义，如下所示：

```
public interface CustomRegistry
{
    // General methods
    public void initialize(java.util.Properties props)
    throws CustomRegistryException;

    public String getRealm()
    throws CustomRegistryException;

    // User-related methods
    public boolean isValidUser(String userName)
    throws CustomRegistryException;

    public List getUsers()
    throws CustomRegistryException;

    public List getUsers(String pattern)
    throws CustomRegistryException;

    public String getUniqueUserId(String userName)
    throws CustomRegistryException,
    EntryNotFoundException;

    public String getUserSecurityName(String uniqueUserId)
    throws CustomRegistryException,
    EntryNotFoundException;

    public String getUserDisplayName(String securityName)
    throws CustomRegistryException,
```

```
EntryNotFoundException;

public List getUsersForGroup(String groupName)
throws CustomRegistryException,
EntryNotFoundException;

public List getUniqueUserIds(String uniqueGroupId)
throws CustomRegistryException,
EntryNotFoundException;

// Group-related methods
public boolean isValidGroup(String groupName)
throws CustomRegistryException;

public List getGroups()
throws CustomRegistryException;

public List getGroups(String pattern)
throws CustomRegistryException;

public String getUniqueGroupId(String groupName)
throws CustomRegistryException,
EntryNotFoundException;

public String getGroupSecurityName(String uniqueGroupId)
throws CustomRegistryException,
EntryNotFoundException;

public String getGroupDisplayName(String groupName)
throws CustomRegistryException,
EntryNotFoundException;

public List getGroupsForUser(String userName)
throws CustomRegistryException,
EntryNotFoundException;

public List getUniqueGroupIds(String uniqueUserId)
throws CustomRegistryException,
EntryNotFoundException;

// Authentication methods
public String checkPassword(String userId, String password)
throws PasswordCheckFailedException,
CustomRegistryException;
```



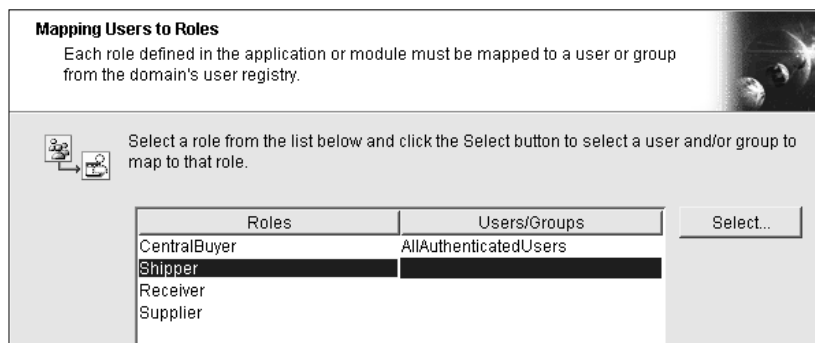
```
public String mapCertificate(X509Certificate cert)
throws CertificateMapNotSupportedException,
CertificateMapFailedException,
CustomRegistryException;
}
```

当这个类首次被使用时，自定义属性将被传递给 `initialize` 方法。在这个例子中，用户注册表信息实际上维护在一个 Oracle 数据库中，并且自定义属性告知自定义注册表类如何访问这个数据库。

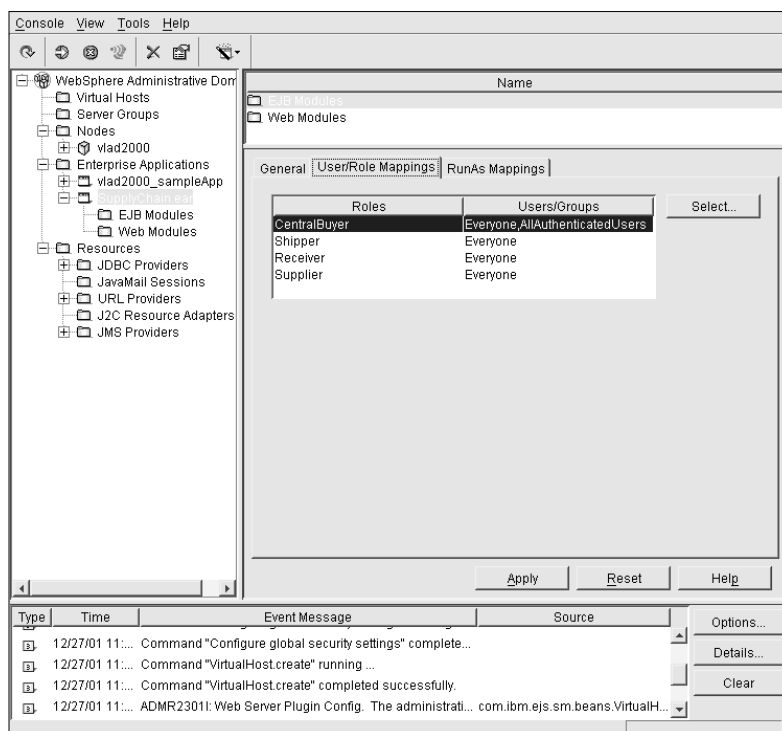
44.4 将用户与组映射到安全性角色

正如在第 19 章讨论的那样，应用程序装配涉及安全性角色的定义。然后，这些角色将被分配调用方法以及访问资源的许可。然后，就需要将用户与用户组映射到这些安全性角色上，从而完成授权 profile。基于这个映射，每个访问系统的用户都将与安全性角色相关联（或不关联）以及被授权（或没有被授权）访问。这种映射可以在安装应用程序服务器时完成，也可以在后来使用 Administrative Console 或 Security Center 完成。

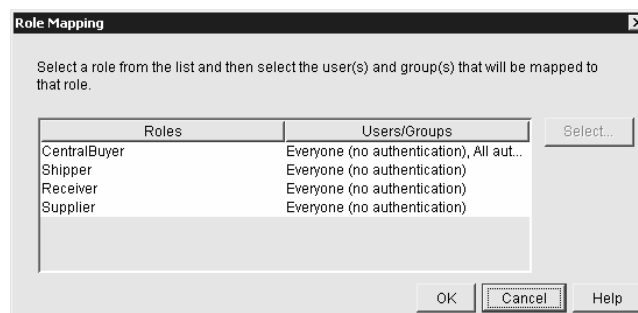
在安装企业应用程序时，向导中有一个步骤如下所示：



对于每个角色，需要选择一系列用户和/或组。这一任务通过单击 **Select** 按钮来完成，它允许选择各种用户与组。需要为应用程序定义的每个安全性角色重复这一工作。在应用程序安装之后，可以在 Administrative Console 中直接修改这个映射：



也可以从 Security Center 中修改这个映射，在这里，应该单击 Edit Mappings 从而打开 Role Mapping 对话框：



44.5 Web 服务器与应用程序服务器之间的安全连接

第 41 章分析了在 DMZ 环境下的一个部署。正如在图 44-6 中显示的那样，这种情况包括了两个防火墙——其中一个在公共 Internet 与 Web 服务器之间，另一个在 Web 服务器与应用程序服务器之间。

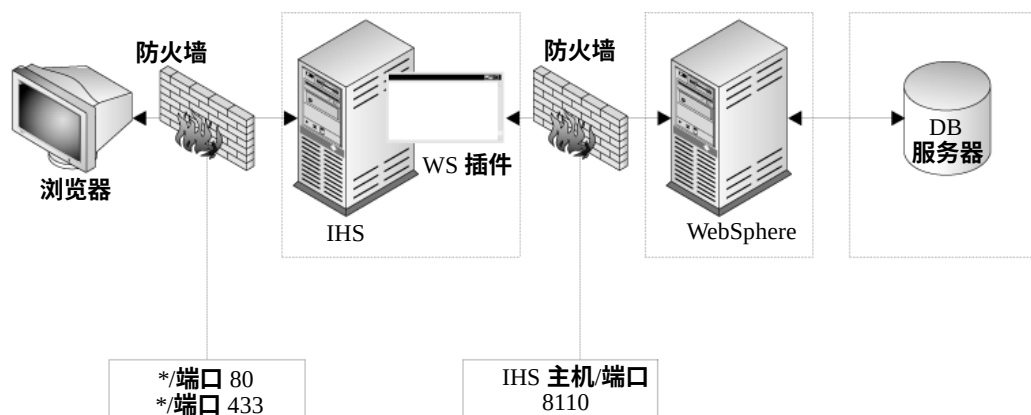


图 44-6 DMZ 环境

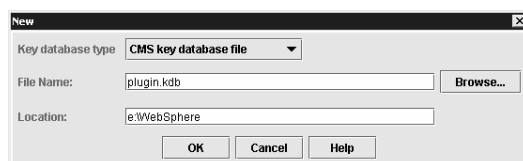
在 WebSphere 4.0 中，WebSphere 插件与 WebSphere Application Server 之间的通信是通过 HTTP 实现的。

正如在第 41 章提到的那样，这是对 OSE（Open Servlet Engine，开放 Servlet 引擎）协议的一个重大改进，这个协议基于 WebSphere 3.x。其中的一个好处就是，在使用 SSL 时，Web 服务器与应用程序服务器之间的通信可以被加密，从而加强了整个系统的安全性。当使用 SSL 时，Web 服务器与应用程序服务器之间使用的协议是 HTTPS，这个协议被所有防火墙、路由器等支持。

为了使用 SSL 在 WebSphere 插件与应用程序服务器之间建立安全通信，必须为这两个组件创建证书，并且将它们安装在各自相应的密钥数据库中，然后为插件与 Web 容器修改配置。由于这一通信发生在 2 个内部服务器之间，因此不需要使用 CA（Certificate Authority）证书；相反，可以生成自己的证书。

为了测试，可以利用默认的 SSL 配置与假密钥（WebSphere 携带的）。为此，需要在 Transport Properties Data 页面中选中正确的项。如果 HTTP 服务器没有在 DMZ 中，甚至可以在产品中这样使用。在这种情况下，需要为 Web 服务器插件生成一个证书并且为应用程序服务器中的 Web 容器生成一个证书。

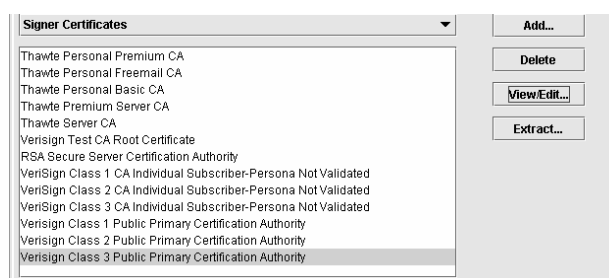
对于 Web 服务器插件，可以使用 IHS（IBM HTTP Server）安装中提供的密钥管理实用程序创建一个密钥文件。一旦这个实用程序被打开，就可以创建一个 CMS 密钥数据库文件：



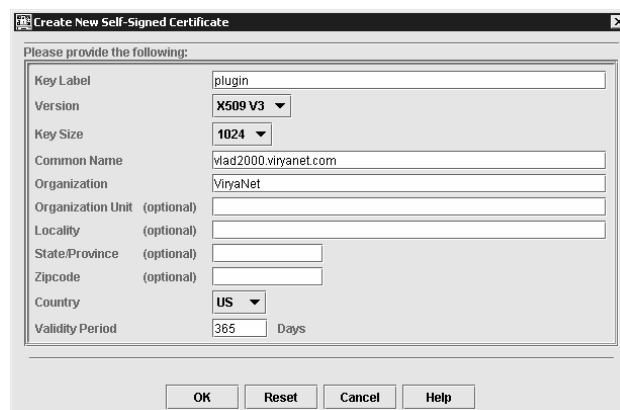
然后，需要选择一个口令。需要说明的是，这个实用程序能够帮助选择口令，其方法就是

对用户所选择的口令强度（口令强度就是这个口令被猜中或破解的容易程度）进行评估并打上标记。

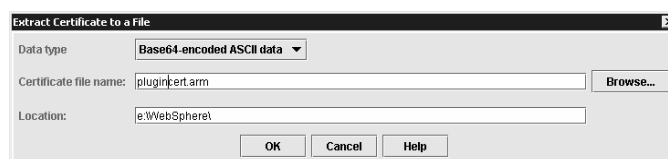
当这个密钥文件首次创建时，它包括了相当数量的 CA 信任的签名者证书，如下所示：



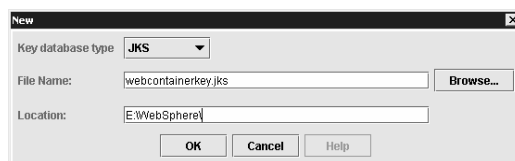
由于插件仅仅需要向 Web 容器验证自己，因此可以删除所有那些证书并创建自己的证书。后来，就可以直接将这个证书导入到容器的密钥文件中。在这个阶段，将删除前一个演示中显示的所有行，并创建一个新的、自己签名的证书：



然后，需要将这个证书提取到一个文件中，并且在后来的过程中将使用这个文件：



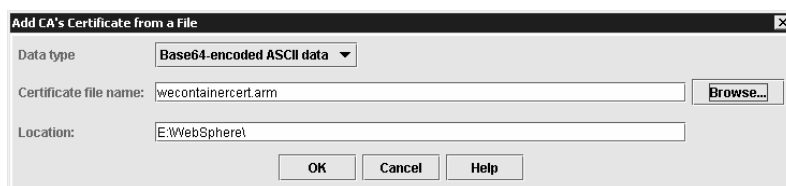
现在，可以关闭这个密钥文件并退出这个实用程序。下一个阶段就是启动作为 WebSphere 安装一部分的密钥管理实用程序。同样，首先将创建一个密钥文件，但这次它是一个 JKS（Java Key Store）文件。



在设置其口令之后，将再次遇到一系列签名者证书。同样，可以删除所有这些证书并添加一个新签名者证书，提取出这个证书并输出到一个文件中。

在完成了上面这些步骤（第一步）之后，就可以继续进行下面两个步骤。现在，已经创建了两个密钥文件并且将每个文件的证书分别提取到一个 .arm 文件中。下面需要将它们导入到各自的密钥数据库中。需要做的实际工作是，确保当 Web 容器接收到一个来自插件的请求时，它有这个插件的证书并且能够对其进行认证。

为此，需要再次打开来自 IHS 安装的密钥管理实用程序，然后打开插件密钥文件。从下拉菜单中选择 Signer Certificates，并且单击 Add 按钮。选择从容器中提取的 .arm 文件，如下所示：



这时，新的密钥数据库内容将出现在 IBM Key Management 对话框中。可以关闭插件密钥数据库并退出这个实用程序。调用 WebSphere 的密钥管理实用程序，可以对插件的证书执行相同的动作，并且将 plugin.arm 文件导入到容器的密钥文件中。

需要执行的最后两个步骤涉及插件与容器本身。为了请求 HTTPS 通信，需要修改其配置。对插件配置的修改是通过修改 plug-in.xml 文件来实现的。应该将下列代码

```
<ServerGroup Name="default_group">
  <Server Name="default_server">
    <Transport Hostname="ihs" Port="9080" Protocol="http"/>
  </Server>
</ServerGroup>
```

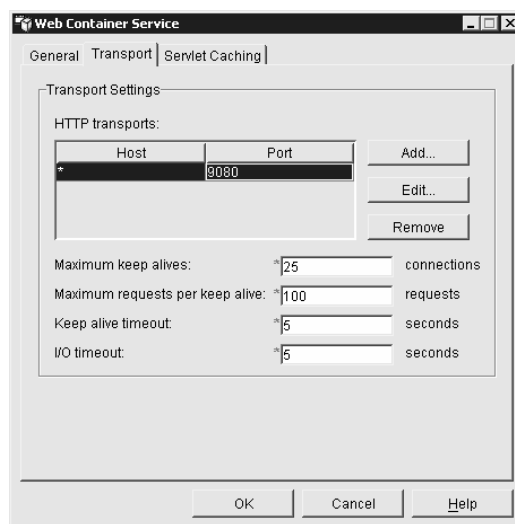
替换成：

```
<ServerGroup Name="default_group">
  <Server Name="default_server">
    <Transport Hostname="ihs" Port="9080" Protocol="https"/>
    <Property name="keyring" value="e:\WebSphere\plugin.kdb"/>
    <Property name="stashfile" value="e:\WebSphere\plugin.sth"/>
  </Server>
</ServerGroup>
```

```
</Transport>
</Server>
</ServerGroup>
```

将 Web 容器修改为使用 HTTPS 是从 Administrative Console 中完成的。

1. 启动这个控制台并选择希望为其设置安全通信的应用程序服务器。右击这个应用程序服务器，然后选择 Properties。
2. 从 Services 标签中选择 Web Container Service 然后单击 Edit Properties 按钮。
3. 在下面这个对话框中，选择 Transport 标签并单击那个惟一可用的传输：



4. 单击 Edit 按钮，然后修改这个传输的属性。触发 Enable SSL 选项并填写密钥文件名与口令。
5. 在所有的对话框中都单击 OK 之后，重新启动这两个服务器，这样，就完成了设置工作；服务器之间的通信现在就是通过 HTTPS 完成的。

最后需要说明的是：管理员可以为相同的应用程序使用 HTTP 与 HTTPS。只要打开这两个端口，WebSphere 就能够非常智能地将非安全请求通过 HTTP 传输，而安全请求通过 HTTPS 传输。

44.6 小 结

设置一个安全的 Web 应用程序涉及大量的工作。本章描述的任务仅仅是其中的一部分。此外，还需要设置防火墙底层结构与路由器（所有这些都属于安全性结构的范畴）。所有这些问题都很复杂，并且应该预留足够的时间来处理这些问题，因为安全性问题常常是由于错

误的配置而引发的。

现在就结束了本书最后一章的讨论。我们希望本书具有较好的可读性，并且希望读者从书中获取了大量有益的信息，可用来帮助开发、部署与管理运行在 WebSphere Application Server 上的应用程序。同时，我们还建议安装并熟悉 IBM 的 info center，因为它是对完成日常任务非常有用的工具。此外，IBM 还出版了许多很好的红皮书，它们覆盖了 WebSphere 开发与管理的方方面面。这些书加起来有超过 5000 多页的篇幅——因此，除非希望对 WebSphere Application Server 的某个方面进行深入的研究，否则很难用到它们；但如果这就是您的工作目标，我们还是强烈推荐使用这些参考资料。