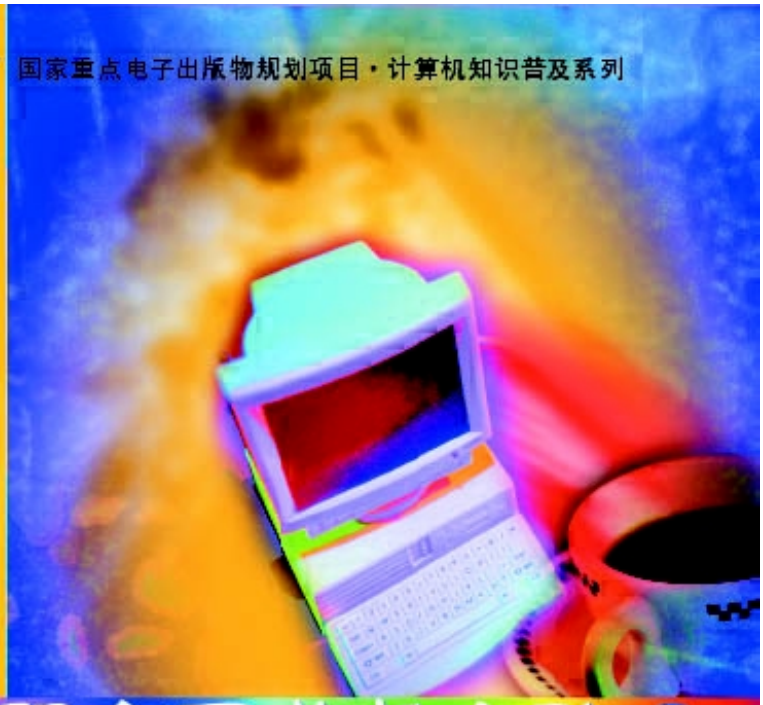
"九五"国家重点电子出版物规划项目·计算机知识普及系列



编程高手成长之路6

Advanced Programming with JavaServer Pages

JSP 高级编程

北京希望电子出版社 总策划
北京大学cem工作室 创作
董 理 陈 亮 曹林有 张 勇等 编著

北京希望电子出版社
Beijing Hope Electronic Press
www.bhp.com.cn

“九五”国家重点电子出版物规划项目　希望计算机知识普及系列

编程高手成长之路（6）

Advanced Programming with JavaServer Pages

JSP 高级编程

北京希望电子出版社　总策划

北京大学 com 工作室　创作

黄理 洪亮 曹林有 张勇等　编著

特点：

- 指导性和实用性强
- 范例丰富、典型
- 附有丰富的实例源码

重点内容：

- JSP 技术与 J2EE 技术
- JSP 技术与 XML 技术
- JDBC 新技术及其在 JSP/Servlet 程序中的应用
- JSP 网络程序设计

北京希望电子出版社
Beijing Hope Electronic Press
www.bhp.com.cn

2001

内 容 简 介

这是一本面向中、高级编程人员的自学指导书，其重点放在 JSP 和其他技术的综合使用方面，全书可分为四大部分：第一部分是 JSP 技术与 J2EE 技术（第一章至第四章），着重介绍 J2EE 技术的代表——EJB 技术的基本原理与开发 EJB 组件的方法。第二部分是 JSP 技术和 XML 技术（第五章至第八章），主要介绍了 XML 技术与 JSP 技术联合的方式之一——Tag Library。第二部分后面的两章是针对市场上最流行的两个 Tag Library 的详细介绍，读者可以把它当作参考手册来使用。本作品的第三部分是 JDBC 新技术及其在 JSP/Servlet 程序中的应用（第九章和第十章），主要介绍最新的 JDBC 技术，如 JDBC 2.0/JDBC 3.0 的新特性，以及鲜为人知而又十分重要的 JDBC Optional Pack。本作品的第四部分是 JSP 网络程序设计，着重介绍如何使用 sun.net 包、JavaMail API 开发访问各种网络服务的 JSP 程序。本作品四个部分之间互为关联又相对独立，读者可以选择阅读某一个部分或者是通读全文。

本版作品是由经验丰富的使用 JSP 组建网站的程序员编著，内文附有丰富的实例源码，供读者学习参考。全书具有语言简明扼要，内容丰富，范例典型，理论与实践相结合的特点，不但是从事用 JSP 进行网站开发和设计的初、中级读者的自学指导书，同时也可作为社会网页设计或编程培训班的教材。

说明：与本书配套的面向初、中级用户的书《JSP 深入编程》也已正式出版，欢迎选购。

本版 CD 为配套书。

系 列 书 名：“九五”国家重点电子出版物规划项目　希望计算机知识普及系列
编程高手成长之路（6）
书　　名：JSP 高级编程　Advanced Programming with JavaServer Pages
总 策 划：北京希望电子出版社
文本著作者：北京大学com工作室 创作　黄理 洪亮 曹林有 张勇等　编著
责任编辑：马红华
CD 制作者：希望多媒体开发中心
CD 测试者：希望多媒体测试部
出版、发行者：北京希望电子出版社
地　　址：北京中关村大街 26 号，100080
网址: www.bhp.com.cn
E-mail: lwm@hope.com.cn
电话: 010-62562329,62541992,62637101,62637102,62633308,62633309（发行）
010-62613322-215（门市）　010-62613322-308（编辑部）
经　　销：各地新华书店、软件连锁店
排　　版：希望图书输出中心　杜海燕
CD 生 产 者：北京中新联光盘有限责任公司
文本印刷者：北京双青印刷厂
开本 / 规格：787 毫米×1092 毫米　1/16 开本　38.75 印张　900 千字
版次 / 印次：2001 年 10 月第 1 版　2001 年 10 月第 1 次印刷
印　　数：0001—5000 册
本 版 号：ISBN 7-980007-78-6
定　　价：55.00 元（本版 CD）

说明：凡我社光盘配套图书若有缺页、倒页、脱页、自然破损，本社负责调换。

声　明

本电子版不包括第 8 章内容，请参看配套图书相关章节。

前　言

JSP（JavaServer Pages）是目前十分流行的一种技术，主要运行于开发服务端的脚本程序和动态生成网站的内容。它与目前同样流行的 ASP 技术、PHP 技术是同样性质的、同一层次的，它们在网站的建设中所起到的作用是一样的。但是 JSP 技术与后面两种技术相比，有着十分突出的优越性。关于 JSP 技术与 ASP 技术、PHP 技术的比较，我们在书中另有论述，这里只想强调一点：JSP 技术有 J2EE 平台支持，发展前途不可限量。众所周知，J2EE 平台提供了 Java 企业应用编程接口（Java Enterprise APIs），为企业计算以及电子商务应用系统提供了有关的技术和强大的类库支持，J2EE 平台包含十几种技术，JSP 技术正是其中的一种核心技术。J2EE 的发展势头十分迅猛，在可以预见的将来，Sun 的 J2EE 平台可能是唯一可以与微软的.Net 构架相互抗衡的平台。在这个意义上说，基于 J2EE 平台的 JSP 技术与基于.Net 平台的 ASP、ASP+技术之争，不正好就是 J2EE 平台与.Net 平台之争的折射吗？因此，JSP 技术以及它的基础——J2EE 技术十分值得我们去关注。在国外，采用 JSP+J2EE 技术构架电子商务网站已经是大行其道了，应用得十分普遍。在国内，这一项技术还是方兴未艾，采用这一项技术架构的网站还不多，不过大致的趋势已经出现了，这真是一个令人兴奋的消息。为了帮助广大读者了解 JSP/J2EE 技术并掌握 JSP/J2EE 技术，我们编写了《JSP 深入编程》和《JSP 高级编程》这两本书。前者侧重于 JSP 技术的基础知识与基本应用，后者侧重于 JSP 技术和其他技术联合使用，本书正是其中第二本书。

本书按顺序讲述了以下知识点：

- JavaBeans 的基础知识。
- EJB 的结构框架。
- 会话 EJB 的开发、部署、测试、应用。
- CMP 模式，BMP 模式的实体 EJB 的开发、部署、测试、应用。
- J2EE 体系结构。
- CORBA,RMI,JNDI 技术入门。
- XML,XSL,CSS 语法介绍。
- WML,XHTML 简介。
- XML+JSP 的开发模式。
- Tag Library 的开发、应用、运行原理。
- javax.servlet.jsp.tagext 包的详细说明。
- JRun Tag Library 的使用手册。
- Jakarta Tag Library 的使用手册。
- JDBC 2.0/3.0 新特性介绍。
- JDBC Optional Pack 介绍（含 RowSet 包、CachedRowSet 包的介绍）。
- JSP 网络程序开发（访问 SMTP,FTP,News 等服务，Socket 技术应用介绍）。
- JavaMail 技术完全指南。

总的来说，本书可以分为四大部分：第一部分是 JSP 技术与 J2EE 技术（第一章至第四章），着重介绍 J2EE 技术的代表——EJB 技术的基本原理与开发 EJB 组件的方法。第二部

分是 JSP 技术和 XML 技术（第五章至第八章），主要介绍了 XML 技术与 JSP 技术联合的方式之一——Tag Library。第二部分后面的两章是针对市场上最流行的两个 Tag Library 的详细介绍，读者可以把它当作参考手册来使用。本书的第三部分是 JDBC 新技术及其在 JSP/Servlet 程序中的应用（第 9 章和第 10 章），主要介绍最新的 JDBC 技术，如 JDBC 2.0/JDBC 3.0 的新特性，以及鲜为人知而又用处极大的 JDBC Optional Pack。本书的第四部分是 JSP 网络程序设计，着重介绍如何使用 sun.net 包、JavaMail API 开发访问各种网络服务的 JSP 程序。本书这四个部分之间互为关联又相互独立，读者可以单独阅读某一个部分或者是通读全书。

顾名思义，本书不是关于 JSP 技术的入门书籍，本书要求读者必须有 JSP,Java 基础，否则阅读起来可能会有很大的困难。作者建议读者不妨参考《JSP深入编程》，因为这两本书是配套编写的，在知识体系结构上有一定的承接性。

本书虽然名为《JSP 高级编程》，但是真正涉及到 JSP 程序编写技巧方面的章节并不多，这是因为 JSP 技术的核心内容很少很少，除了基本语法、编译指令、操作指令和内部对象以外，就没有别的东西了，要发挥 JSP 技术的长处，开发功能强大的 JSP 程序，单单靠 JSP 技术本身是不可能的。JSP 技术必须和其他相关的 Java 技术结合起来，例如 JDBC,EJB,RMI,CORBA,JavaMail 等技术，才有可能开发出功能强大的程序。本书重点介绍的就是上述技术的基本原理和开发方法，至于如何把这些技术和 JSP 技术结合起来，开发运行于服务端的应用程序与 JSP 程序，书中讲的很少，但是读者应该有这方面的经验。况且只要明了这些技术的基本原理与开发的方法，把它们和 JSP 技术结合起来是一件十分简单的事情，不需要浪费过多的笔墨去介绍这方面的知识。

当你读完本书以后，我们不能够保证你一定能够成为 JSP 高手，因为本书提到的技术虽然很多，但是由于篇幅的关系以及其他的原因，这些技术讲的都很肤浅，只是相当于入门的水平。读者如果想有更大的进步，最好是深入研究本书所提到的技术，找几个项目来做，当你能够游刃有余地应用这几种技术于 JSP 程序的开发中时，那时你才是真正的精通 JSP 的高手。本书给读者指出努力的方向以及提供入门的知识，剩下的就靠读者自身的努力了。这就是本书命名为《JSP 高级编程》的原因。

本书由北京大学 com 工作室组织编写。由于时间仓促，笔者的水平有限，书中的谬误一定很多。不足之处，请读者指正。

本书的成功出版，首先归功于本书的主要作者北大黄理同学，他深厚的计算机理论积累和丰富的实践经验才使得本书兼具理论指导及实务操作性，其工作的严谨态度以及出色的语言驾驭功底相信读者在阅读本书时自有体会；其次还有感谢北大洪亮同学，其出色的工作为本书增色不少。也感谢其他许许多多人辛勤的劳动与无私的帮助，轻易便可以列出很多：北大计算机系的李积善、水木清华（smth.org）的 javafancy、北大未名站的 javalover，还有 ROBBY、lz.lan、snowleaf 以及可爱的 Rainbow。

本书技术支持的联系方式：

com_pku@263.net

http：//162.105.106.162：8080 注：访问前，需要事先 mail 联系，以便启动服务器。

目　录

第一部分　JSP 技术与 J2EE 技术

第 1 章　JavaBeans 组件技术

1.1　什么是 JavaBeans
1.2　JSP 中如何使用 JavaBeans
1.3　JavaBeans 的 Scope 属性
1.4　JavaBeans 应用实例
1.5　本章小结

第 2 章　Enterprise JavaBeans

2.1　EJB 技术简介
2.2　EJB 体系结构(一)
2.3　EJB 体系结构(二)
2.4　如何开发 EJB(一)
2.5　如何开发 EJB(二)
2.6　本章小结

第 3 章　EJB 技术进阶

3.1　实体 EJB 的开发技术之一——CMP EJB
3.2　实体 EJB 的开发技术之二——BMP EJB
3.3　EJB 开发实例——封装数据源
3.4　本章小结

第 4 章　JSP 与 J2EE 分布式处理技术

4.1　J2EE 和分布式处理技术
4.2　远程方法调用——RMI 技术
4.3　CORBA 技术
4.4　JNDI 技术
4.5　本章小结 6

第二部分　JSP 技术和 XML 技术

第 5 章　XML 简介

5.1　XML 简介及其语法规则
5.2　DTD 的书写及实例

5.3 CSS 与 XSL 及其实例
5.4 XHTML 简介
5.5 WML 简介
5.6 本章小结
第 6 章 JSP 与 XML 联合开发技术
6.1 XML 与 JSP 技术联合
6.2 在 JSP 中应用 XML
6.3 javax.servlet.jsp.tagext 包介绍
6.4 Tag Library 开发与应用实例
6.5 本章小结
第 7 章 典型 Tag Library 介绍——JRun Tag Library
7.1 JRun Tag Library 简介
7.2 SQL 标记
7.3 J2EE 标记
7.4 Mail 标记
7.5 XML 标记
7.6 其它标记
7.7 本章小结
第 8 章 典型 Tag Library 介绍——Jakarta Tag Library
8.1 Jakarta Tag Librarys 简介
8.2 Application Tag Library
8.3 BSF Tag Library
8.4 DateTime Tag Library
8.5 Input Tag Library
8.6 JDBC Tag Library
8.7 Mailer Tag Library
8.8 Page Tag Library
8.9 Request Tag Library
8.10 Response Tag Library
8.11 Session Tag Library
8.12 本章小结

第三部分 JDBC 新技术及其在 JSP/Servlet 中的应用

第 9 章 JDBC 2.0/3.0 API 的新特性
9.1 JDBC API 2.0 的新特性
9.2 JDBC API 2.0 简介
9.3 JDBC API 3.0 简介

9.4 附录：JDBC 数据类型和 Java 数据类型的映射关系
9.5 本章小结
第 10 章 JDBC Optional Package
10.1 JDBC Optional Package 是什么
10.2 RowSet 包
10.3 CachedRowSet 包
10.4 数据库连接缓冲池
10.5 JNDI 和 RowSet
10.6 本章小结

第四部分 JSP 网络程序设计

第 11 章 JSP 网络程序开发
11.1 配置服务器
11.2 SMTP 服务
11.3 FTP 服务
11.4 News 服务
11.5 Java Socket
11.6 Telnet 服务
11.7 本章小结
第 12 章 Java Mail API
12.1 Java Mail API 简介
12.2 javax.mail 包
12.3 javax.mail.internet 包
12.4 Sun Protocol Privider API 简介
12.5 使用 Java Mail API 访问 Mail 服务器
12.6 本章小结
附录 1 支持 EJB1.0 技术规范的 EJB 平台、开发工具一览表
附录 2 JDBC Driver 一览表
附录 3 WebLogic 服务器的配置方法
附录 4 本书中所用数据库的数据库结构
参考文献

第一部分　JSP 技术与 J2EE 技术

第 1 章　JavaBeans 组件技术

本章将要向读者介绍 JavaBeans 组件技术在 JSP 程序开发中的应用。在《JSP 深入编程》中，我们已经介绍了一点关于 JavaBeans 的知识，但是由于体系结构的原因，我们并没有深入讨论它，也许有的读者对此还有些遗憾，不过不要紧，这一章就来弥补读者的这个遗憾。本章中读者需要重点掌握的内容有：

- JavaBeans 的属性
- JavaBeans 的事件模型
- JSP 中与 JavaBeans 相关的操作指令的语法与用法
- JavaBeans 的开发流程
- JavaBeans 的 Scope 属性
- JavaBeans 封装数据库操作

1.1　什么是 JavaBeans

1.1.1　JavaBeans 简介

软件开发的真正目的之一是利用在程序编码方面的投资，以便在同一公司或者不同公司的其他开发中重用程序编码。近年来，编程人员投入大量精力以便建立可重用的软件、可重用的软件组件。早期用在面向对象编程方面中的投资已经在 Java、C#等编程语言的开发中充分实现，很多软件可以不用做很大的改变就可以运行在各种平台上。

JavaBeans 描述了 Java 的软件组件模型，这个模型被设计成使第三方厂家可以生成和销售能够集成到其他开发厂家或者其他开发人员开发的软件产品的 Java 组件。

应用程序开发者可以从开发厂家购买现成的 JavaBeans 组件，拖放到集成开发环境的工具箱中，再将其应用于应用软件的开发，对于 JavaBeans 组件的属性、行为可以进行必要的修改、测试和修订而不必重新编写和编译程序。在 JavaBeans 模型中，JavaBeans 组件可以被修改或者与其他 JavaBeans 组件组合以生成新的 JavaBeans 组件或完整的 Java 应用程序。

Java 应用程序在运行时，最终用户也可以通过 JavaBeans 组件设计者或应用程序开发者所建立的属性存取方法（setXXX 方法和 getXXX 方法）修改 JavaBeans 组件的属性，这些属性可能是颜色和形状等简单属性，也可能是影响 JavaBeans 组件总体行为的复杂属性。

JavaBeans 组件模型使得软件可以设计成便于修改和便于升级。每个 JavaBeans 组件都

包含了一组属性、操作和事件处理器。将若干个 JavaBeans 组件组合起来就可以生成设计者、开发者所需要的特定运行行为，JavaBeans 组件存放于容器或工具库中，供开发者开发应用程序。

JavaBeans 就是一个可以复用软件模型。JavaBeans 在某个容器中运行，提供具体的操作性能。JavaBeans 是建立应用程序的建筑模块。大多数常用的 JavaBeans 通常是中小型控制程序，但我们也可以编写包装整个应用程序运行逻辑的 JavaBeans 组件 ，并将其嵌入到复合文档中，以便实现更为复杂的功能。

一般来说，JavaBeans 可以表示为简单的 GUI 组件，可以是按钮组件、游标、菜单等等。这些简单的 JavaBeans 组件提供了告诉用户什么是 JavaBeans 的直观方法。但我们也可以编写一些不可见的 JavaBeans，用于接受事件和在幕后工作，例如访问数据库，执行查询操作的 JavaBeans，它们在运行时刻不需要任何可视的界面。在 JSP 程序中所用的 JavaBeans 一般以不可见的组件为主。可见的 JavaBeans 一般用于编写 Applet 程序或者 Java 应用程序。

1.1.2　JavaBeans 属性

JavaBeans 的属性与一般 Java 程序中所指的属性，或者说与所有面向对象的程序设计语言中对象的属性是同一个概念，在程序中的具体体现就是类中的变量。在 JavaBeans 的设计中，按照属性的不同作用又细分为 4 类：Simple 属性、Index 属性、Bound 属性与 Constrained 属性。

Simple 属性

一个 Simple 类型的属性表示一个伴随有一对 getXXX()、setXXX()方法的变量。属性的名称与和该属性相关的 getXXX()、setXXX()方法相对应。例如：如果有 setX()和 getX()方法，则暗指有一个名为"X"的属性，如果有一个方法名为 isX()，则通常暗指"X"是一个布尔类型的属性。请看下面的程序清单 1.1(JavaBean1.java)。

程序清单 1.1

```
//File Name:JavaBean1.java
//Author:fancy
//Date:2001.3.29
//Note:create a simple javabean

public class JavaBean1
{
    String ourString= "Hello";

    public JavaBean1()
    {

    }

    public void setoutString(String newString)
```

```
    {
        ourString=newString;
    }
    public String getoutString()
    {
        return ourString;
    }
}
```

在程序清单 1.1(JavaBean1.java)中，我们定义了一个 JavaBean——JavaBean1，其实也就是定义了一个 JavaBean1 类。JavaBean1 有一个名为 outString 的字符串类型的属性。与这个属性相对应的方法为 setoutString()和 getoutString()，使用这两个方法可以存取 outString 属性的值。

Indexed 属性

一个 Indexed 类型的 JavaBeans 属性表示一个数组值。使用与该属性相对应的 setXXX()方法和 getXXX()方法可以存取数组中某个元素的数值。同时，我们也可以使用另两个同名方法一次设置或取得整个数组的值(即属性的值)。请看程序清单 1.2。

程序清单 1.2

```
//File Name:JavaBean2.java
//Author:fancy
//Date:2001.3.29
//Note:create a indexed javabean

public class JavaBean2
{
    int[] dataSet={1，2，3，4，5，6};

    public void JavaBean2()
    {
    }

    public void setDataSet(int[] x)
    {
        dataSet=x;
    }

    public void setDataSet(int index，int x)
    {
        dataSet[index]=x;
    }

    public int[] getDataSet()
```

北京邮电大学出版社

```
    {
        return dataSet;
    }

    public int getDataSet(int x)
    {
        return dataSet[x];
    }
}
```

在程序清单 1.2(JavaBean2.java)中，定义了 JavaBean——JavaBean2，JavaBean2 具有属性 dataSet，dataSet 属性是一个整型数组，JavaBean2.java 定义了 4 个方法以存取 dataSet 属性的值，它们分别是： setDataSet(int[] x)、setDataSet(int index，int x)、getDataSet(int x)、getDataSet()，其中 setDataSet(int[] x)方法可以一次设定 dataSet 属性的值，getDataSet()方法可以一次获取 dataSet 属性的值，该方法的返回值是一个整型数组，getDataSet(int x)方法可以获取 dataSet 属性中某个指定的元素的值，该方法的返回值为整型数据，与这个方法相对的方法是 setDataSet(int index，int x)方法，使用这个方法可以指定 dataSet 属性中某个特定元素的值。

Bound 属性

一个 Bound 类型的 JavaBean 组件的属性具有这样的特性：当该种属性的值发生变化时，必须通知其它的 JavaBeans 组件对象。每次 JavaBeans 组件对象的属性值改变时，这种属性就引发一个 PropertyChange 事件(属性改变事件，在 Java 程序中，事件也被看作是一个对象)。这个事件中封装了发生属性改变事件的属性名、属性的原值、属性变化后的新值。这个事件将被传递到其它的 JavaBeans 组件中，至于接收事件的 JavaBeans 组件对象应该做什么动作由其自己定义。请看程序清单 1.3(JavaBean3.java)。

程序清单 1.3

```
//File Name:JavaBean3.java
//Author:fancy
//Date:2001.3.29
//Note:create a bound javabean

import java.beans.*;
public class JavaBean3
{
    String ourString= "Hello";
    private PropertyChangeSupport changes = new PropertyChangeSupport(this);
    public void setString(String newString)
    {
        String oldString = ourString;
        ourString = newString;
        changes.firePropertyChange("ourString"，oldString，newString);
```

```
    }

    public String getString()
    {
        return ourString;
    }

    public void addPropertyChangeListener(PropertyChangeListener l)
    {
        changes.addPropertyChangeListener(l);
    }

    public void removePropertyChangeListener(PropertyChangeListener l)
    {
        changes.removePropertyChangeListener(l);
    }
}
```

读者对程序清单 1.3(JavaBean3.java)的运行逻辑一定感到十分迷惑吧，那好，下面我们就来详细解释 JavaBean3.java 程序的含义。 程序首先创建了 PropertyChangeSupport 类型的对象 changes，这是最关键的一步操作，changes 对象主要用于向监听者对象发送信息：当前的 JavaBean 对象已经发生了属性改变的事件。在 JavaBean3.java 程序中，除了普通的存取 JavaBeans 属性值的 setXXX()、getXXX()等方法以外，还定义了如下的方法：

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

第一个方法(addPropertyChangeListener()方法)其实是调用 changes 对象的 addPropertyChangeListener()方法，使一个事件监听者对象和当前 JavaBean 对象绑定起来，并把它添加到监听者队列中去。充当当前 JavaBean 对象的事件监听者，如果当前 JavaBean 对象发生了属性值改变的事件，那么 changes 对象会依次通知监听者队列中的每一个对象，当然也通知了这个事件监听者对象，让它对这个事件做出反映。

第二个方法(removePropertyChangeListener()方法)和前者的作用相反，该方法其实是调用 changes 对象的 removePropertyChangeListener()方法，从监听者队列中移除某个特定的事件监听者对象，此事件监听者对象一旦从监听者队列中删除，那么 changes 对象将不会把属性值改变的事件通知它，它再也没有办法对属性值发生改变的事件作出响应了。

getString()方法可以返回属性值，setString()方法用于设定属性值，setString()方法的代码如下所示：

```
String oldString = ourString;
ourString = newString;
changes.firePropertyChange("ourString"，oldString，newString);
```

在上面的代码中，首先新定义一个字符串 oldString，用于保存属性的原值，然后把新值赋给属性值，这样会产生 JavaBeans 组件属性值改变的事件，最后调用 changes 对象的 firePropertyChange()方法，通知监听者队列里的所有事件监听者对象，当前的 JavaBean 对

象发生了属性值改变的事件，属性的名称、属性的新值、属性的原值，都被作为该方法的参数，一并传给监听者对象，由它们根据这些信息，对此事件作出响应。

Bound 类型的属性就是这样使用的。

Constrained 属性

JavaBeans 组件的 Constrained 类型的属性具有这样的性质：当这个属性的值将要发生变化但是还没有发生变化的时候，与这个属性已经建立了某种监听关系的其它 Java 对象可以否决属性值的改变。此 Constrained 类型的属性的事件监听者对象将会通过抛出 PropertyVetoException 异常事件来阻止该属性值的改变。读者请看程序清单 1.4(JavaBean4.java)。

程序清单 1.4

```
//File Name:JavaBean4.java
//Author:fancy
//Date:2001.3.29
//Note:create a Constrained javabean

    import java.beans.*;
    public class JavaBean4
    {
    private PropertyChangeSupport changes=new PropertyChangeSupport(this);
    private VetoableChangeSupport vetos=new VetoableChangeSupport(this);
    int ourPriceInCents;

    public void setPriceInCents(int newPriceInCents)
        throws PropertyVetoException
    {
        int oldPriceInCents=ourPriceInCents;
        vetos.fireVetoableChange("priceInCents",
            new Integer(oldPriceInCents),
            new Integer(newPriceInCents));
        ourPriceInCents=newPriceInCents;
        changes.firePropertyChange("priceInCents",
            new Integer(oldPriceInCents),
            new Integer(newPriceInCents));
    }

    public void addVetoableChangeListener(VetoableChangeListener l)
    {
        vetos.addVetoableChangeListener(l);
    }

    public void removeVetoableChangeListener(VetoableChangeListener l)
```

```
    {
        vetos.removeVetoableChangeListener(l);
    }

    public void addPropertyChangeListener(PropertyChangeListener l)
    {
        changes.addPropertyChangeListener(l);
    }

    public void removePropertyChangeListener(PropertyChangeListener l)
    {
        changes.removePropertyChangeListener(l);
    }
}
```

程序清单 1.4(JavaBean4.java)比起程序清单 1.3(JavaBean3.java)来说，显得更为晦涩难解，在程序清单 1.4 中，定义了一个 JavaBean——JavaBean4，它有一个 Constrained 类型的属性是 ourPriceInCents，这是一个整型数据。为什么说它是 Constrained 类型的属性呢?请读者注意，在程序的开始部分，我们分别定义了 PropertyChangeSupport 类型的对象 changes 和 VetoableChangeSupport 类型的对象 vetos，changes 对象的作用和程序清单 1.3 中 changes 对象的作用一样，在这里我们就不讨论它的用法了。在这里我们主要讨论 vetos 对象的用法。

vetos 对象主要用于通知事件否决者对象，某个 JavaBean 对象的属性值将要发生变化，让它们投票表决是否允许这个事件的发生。在 JavaBean4.java 中，定义了这样的两个方法，分别是：

public void addVetoableChangeListener(VetoableChangeListener l);

public void removeVetoableChangeListener(VetoableChangeListener l);

前者可以往事件否决者对象队列中添加新的事件否决者对象，作为 JavaBean4 组件对象的事件否决者，一旦成为 JavaBean4 对象的事件否决者，就可以在事件发生之前，否决事件的发生。

第二个方法与第一个方法的作用相反，它可以将某个特定的事件否决者对象从事件否决者对象列表中删除。被删除的事件否决者对象就再也没有权利否决事件的发生，除非它们再次被添加到事件否决者队列中去。

在 JavaBean4.java 程序中，读者需要特别注意 setPriceInCents()方法的实现。在 setPriceInCents()方法中，首先把 ourPriceInCents 属性的原值给保存下来。然后调用 vetos 对象的 fireVetoableChange()方法，通知事件否决者对象队列中的每一个事件否决者对象，告诉它们：JavaBean4 对象即将发生属性改变的事件，发生此事件的属性是 ourPriceInCents，属性的新值为 newPriceInCents，属性的原值为 oldPriceInCents，(实际上还没有改变属性值)。事件否决者对象会根据这些信息，投票表决是否允许该事件的发生，如果有任何一个事件否决者对象否决了这个事件发生的可能性，那么 setPriceInCents()方法将会抛出 PropertyVetoException 异常，程序的运行将会中断，下面的代码将不会执行，也就是说属性值将会保持原来的值。如果事件否决者不否决事件的发生，那么程序将会继续往下执行，

给 ourPriceInCents 属性赋上新值，然后 changes 对象调用 firePropertyChange()方法，通知事件监听者队列中的事件监听者对象，让它们对这个事件作出响应。

总之，某个 JavaBean 组件对象的 Constrained 类型的属性值可否改变取决于其它的事件否决者对象是否允许这种改变。允许与否的条件由其它的事件否决者对象在自己的类中进行定义。

注意：事件监听者和事件否决者的区别在于事件监听者不能够否决事件的发生，但是可以响应事件的发生。而事件否决者正好相反，它可以否决事件的发生，但是不能够响应事件的发生。

1.1.3　JavaBeans 的事件模型

事件处理机制是 JavaBeans 体系结构的核心之一，也是 Java 体系结构的核心之一。通过事件处理机制，我们可以指定一些组件作为事件源，发出可以被系统运行环境或者是其它组件接收的事件。这样，不同的组件就可在某个应用程序内部真正结合在一起，组件之间通过事件的发送、传递、接受进行通信，构成一个完整的逻辑应用。从概念上讲，所谓事件机制，是指一种在“源对象”和“监听者对象”之间，某种状态发生变化时的消息传递机制。事件有许多不同的用途，例如在 Windows 系统中常要处理的鼠标事件、窗口边界改变事件、键盘事件等。在 Java 和 JavaBeans 的事件模型中，则是定义了一个一般的、可扩充的事件机制，这种机制能够：

- 对事件类型和传递的模型的定义和扩充提供一个公共框架，并适合于广泛的应用。
- 与 Java 语言和环境有较高的集成度。
- 事件能被系统运行环境捕获和引发。
- 能使其它开发工具采取某种技术在设计时直接控制事件，以及事件源和事件监听者、事件否决者之间的联系。
- 事件机制本身不依赖于复杂的开发工具。

特别地，还应当：

- 能够发现指定的对象类可以生成的事件。
- 能够发现指定的对象类可以观察（监听）到的事件。
- 提供一个常规的注册机制，允许动态操纵事件源与事件监听者之间的关系。
- 不需要其它的虚拟机和语言即可实现。
- 事件源与监听者之间可进行高效、快速的事件传递。

下面我们就来简单地介绍 JavaBeans 的事件机制是如何运作的。

事件模型概述

事件从事件源到事件监听者的传递是通过对监听者对象的 Java 方法调用进行的。 对每个明确的事件的发生，都必须相应地定义一个明确的 Java 方法。这些方法都集中在事件监听者接口中定义，而且这个接口必须要继承 java.util.EventListener 接口。也就是说，如果我们希望监听事件源发生的事件，我们必须首先定义一个事件监听者接口，定义各种各样的监听方法，以便接收事件源传递来的事件。具体实现了事件监听者接口中一些或全部方

法的类就是事件监听者。伴随着事件的发生，事件源通常把事件及其相应的状态都封装在事件状态对象中，该对象必须继承自 java.util.EventObject。事件状态对象作为参数被传递给应该响应该事件的事件监听者的方法中。

产生某种特定事件的事件源的特征是：遵从规定的编程格式为事件监听者定义注册方法，以便把监听者对象加入当前事件源的事件监听者队列中，并接受对指定事件监听者接口实例的引用。有时，事件监听者不能直接实现事件监听者接口，或者还有其它的额外动作时，就要在一个事件源与其它一个或多个事件监听者之间插入一个事件适配器类的实例对象，来建立它们之间的联系。实际上，事件适配器类就相当于一个过滤器，它可以把事件监听者对象不应该接收的事件或者是不能够接收的事件都过滤掉。

事件状态对象（Event State Object）

与事件有关的状态信息一般都封装在一个事件状态对象中，这种对象必须是 java.util.EventObject 类的子类。按设计习惯，这种事件状态对象类的名应以 Event 结尾。请看程序清单 1.5 (MouseMovedExamEvent.java)。

程序清单 1.5

```
//File Name: MouseMovedExamEvent
//Author:fancy
//Date:2001.3.31
//Note:EventObject-----Mouse Moved Event
import java.awt.Point;

public class MouseMovedExamEvent extends java.util.EventObject
{
    protected int x;
    protected int y;

    public void MouseMovedExampleEvent(Component source，Point location)
    {
        super(source);
        x = location.x;
        y = location.y;
    }

    public Point getLocation()
    {
        return new Point(x，y);
    }
}
```

在程序清单 1.5(MouseMovedExamEvent.java)中，我们定义了一个事件状态对象 MouseMovedExampleEvent，它代表一个鼠标移动的事件，getLocation()方法可以返回鼠标目前的位置。

事件监听者接口与事件监听者

由于 JavaBeans 的事件模型是基于 Java 的方法调用，因而需要一个定义并组织事件操纵方法的方式。在 JavaBeans 事件模型中，事件操纵方法都被定义在继承了 java.util.EventListener 接口的事件监听者接口中，按照一般的规律，事件监听者接口的命名要以 Listener 结尾。任何一个类如果想使用在事件监听者接口中定义的方法都必须扩展这个接口，并且实现其中定义的方法。果真如此，那么这个类也就是事件监听者。请看程序清单 1.6(ArbitraryObject.java)。

程序清单 1.6

```
//File Name: ArbitraryObject.java
//Author:fancy
//Date:2001.3.31
//Note: show JavaBean event model

import java.beans.*;

//定义事件状态对象类
public class MouseMovedExampleEvent extends java.util.EventObject
{
    // 在此类中包含了与鼠标移动事件有关的状态信息
}

//定义了鼠标移动事件的事件监听者接口
interface MouseMovedExampleListener extends java.util.EventListener
{
    //在这个接口中定义了鼠标移动事件监听者所应支持的方法
    void mouseMoved(MouseMovedExampleEvent mme);
}

//定义事件监听者
class ArbitraryObject implements MouseMovedExampleListener
{
    public void mouseMoved(MouseMovedExampleEvent mme)
    {
        //代码省略
    }
}
```

在程序清单 1.6(ArbitraryObject.java)中，首先定义了事件状态对象类 MouseMovedExampleEvent，在此类中包含了与鼠标移动事件有关的状态信息，接着定义了事件监听者接口 MouseMovedExampleListener，在这个接口中定义了鼠标移动事件监听者所应支持的方法 mouseMoved()，该方法以 MouseMovedExampleEvent 类型的对象为参数。ArbitraryObject 类扩展了 MouseMovedExampleListener 接口，实现了 mouseMoved 方法，所

以它是事件监听者。

注意：程序清单 1.5/1.6 只是简单的示例，代码不完整，编译不会通过，也不能够运行，读者务必要注意。

事件监听者的注册与注销

为了把各种可能的事件监听者注册到合适的事件源的监听者队列中，建立事件源与事件监听者间的事件流，事件源必须为事件监听者提供注册和注销的方法。在前面介绍 bound 类型的 JavaBeans 属性时，我们已经提到了这两种方法，在实际编程中，事件监听者的注册和注销方法必须使用标准的设计格式：

public void add< ListenerType>(< ListenerType> listener)；

public void remove< ListenerType>(< ListenerType> listener)；

前者用于注册事件监听者对象，后者用于注销事件监听者对象。ListenerType 代表事件监听者对象的类型。

程序清单 1.7

```
//File Name:EventExam.java
//Author:fancy
//Date:2001.3.31
//Note:show JavaBean event model

import java.util.*;

//首先定义了一个事件监听者接口：
public interface ModelChangedListener extends java.util.EventListener
{
    void modelChanged(EventObject e);
}

//定义事件监听者
class ModelChangedEventObject implements ModelChangedListener
{
    public void modelChanged(EventObject e)
    {
        //代码省略
    }
}

//接着定义事件源类：
public class EventExam
{
    // 定义了一个储存事件监听者的数组
    private Vector listeners = new Vector();
```

北京希望电子出版社

```
    //上面设计格式中的<ListenerType>在此处即是下面的ModelChangedListener
    //把监听者注册入listeners数组中
    public void addModelChangedListener(ModelChangedListener mcl)
    {
        listeners.addElement(mcl);
    }

    //把监听者从listeners中注销
    public void removeModelChangedListener(ModelChangedListener mcl)
    {
        listeners.removeElement(mcl);
    }

    protected void notifyModelChanged()
    {
        //事件源对象使用本方法通知监听者发生了modelChanged事件
        Vector l;
        EventObject e = new EventObject(this);

        //首先要把监听者拷贝到l数组中，冻结EventListeners的状态以传递事件。
        //这样来确保在事件传递到所有监听者之前，已接收了事件的目标监听者的对
        //应方法暂不生效。

        synchronized(this)
        {
            l = (Vector)listeners.clone();
        }

        for (int i = 0; i < l.size(); i++)
        {
            //依次通知注册在监听者队列中的每个事件监听者发生了modelChanged
            //事件，并把事件状态对象e作为参数传递给监听者队列中的每个监听者。
            ((ModelChangedListener)l.elementAt(i)).modelChanged(e);
        }
    }
}
```

在程序清单 1.7(EventExam.java)中，展示了一个完整的 JavaBeans 事件模型，程序首先定义了事件监听者接口 ModelChangedListener，然后又定义了一个事件监听者 ModelChangedEventObject，ModelChangedEventObject 类实现了 ModelChangedListener 接口中定义的 modelChanged()方法，由于该事件监听者所监听的是最普遍的事件对象 EventObject，因此我们就不必定义事件状态对象了。接下来我们定义了事件源 EventExam

类，EventExam 类使用 Vector 数据类型来存储事件监听者队列。EventExam 类定义了 addModelChangedListener()方法用来往事件监听者队列中添加事件监听者对象(表面上添加的是事件监听者接口 ModelChangedListener 对象，但在实际上添加的是事件监听者对象 ModelChangedEventObject)。removeModelChangedListener()方法可以把事件监听者队列中的特定的事件监听者对象注销。事件源对象调用 notifyModelChanged()方法通知事件监听者发生了 modelChanged 事件。notifyModelChanged()方法的方法体中，使用一个 for 循环结构，遍历 Vector 数据结构中保存的每一个事件监听者接口对象，调用它们的 modelChange()方法，通知事件监听者，modelChanged 事件已经发生了，并且把事件状态对象 e 传递给这些事件监听者，这里虽然调用的是 ModelChangedListener 接口的 modelChange()方法，但是这个方法并没有真正实现，所以实际上调用的是 ModelChangedEventObject 类的 modelChange()方法。

事件适配器类

事件适配器类是 Java、JavaBeans 事件模型中极其重要的一部分。在一些应用场合，事件从事件源到事件监听者之间的传递要通过事件适配器类来“转发”。例如：当事件源发出一个事件，而有几个事件监听者对象都可接收该事件，但只有指定的监听者对象可以做出反应时，就要在事件源与事件监听者之间插入一个事件适配器类，由适配器类来指定事件应该是由哪些事件监听者来响应，再由它来转发事件。

注意：JavaBeans 的事件模型实际上用的并不多，尤其是应用于 JSP 程序中的 JavaBeans，很少需要响应或者监听某种事件的产生，但是这并不等于这部分的内容不重要，有时候为了纪录 JavaBeans 都作了哪些敏感的操作，还是需要利用 JavaBeans 的事件模型的。

1.2　JSP 中如何使用 JavaBeans

JavaBeans 被称为是 Java 组件技术的核心。JavaBeans 的结构必须满足一定的命名约定。JavaBeans 类似于 Windows 下的 ActiveX 控件：它们都能提供常用功能并且可以重复使用。JavaBeans 可以在 JSP 程序中应用给我们带来了很大的方便，这使得开发人员可以把某些关键功能和核心算法提取出来，封装成为一个组件对象，增加了代码的重用率、系统的安全性。比如，我们可以将访问数据库的功能、数据处理功能编写封装为 JavaBeans 组件，然后在某个 JSP 程序中加以调用。JavaBeans 技术与 ActiveX 相比，有着很大的优越性，例如 JavaBeans 的与平台无关性，使得 JavaBeans 组件不但可以运行于 Unix 平台，还可以运行在 Windows 平台下面，而且 JavaBeans 从一个平台移植到另外的平台上代码不需要修改，甚至不需要重新编译。但是 ActiveX 就不同了，它只能够应用于 Windows 平台，而且它的代码移植性很差，从 Windows 98 平台移植到 NT 平台就需要重新编译代码，甚至要大幅度改写程序。另一方面，JavaBeans 比 ActiveX 要容易编写得多，用起来也方便得多。起码 JavaBeans 组件在使用以前不需要注册，而 ActiveX 控件在使用以前必须在操作系统中注册，否则在运行的时候，系统将会报错。

本节将介绍在 JSP 程序中如何使用 JavaBeans 组件。要想在 JSP 程序中使用

JavaBeans 组件，必须应用<jsp:useBean>、<jsp:setProperty>、<jsp:getProperty>等 JSP 的操作指令，关于这几个操作指令的用法，我们在《JSP深入編程》中已经有所涉及 但是限于体系结构方面的原因，我们的讨论十分肤浅，而且没有举出具体的例子，这不能不说是一个缺憾。在这一节中，我们会结合实际的例子，再次详细介绍这三个操作指令的用法，顺便帮助读者复习一下 JSP 的基础知识。

1.2.1 <jsp:useBean>操作指令

<jsp:useBean>操作指令用于在 JSP 页面中实例化一个 JavaBean 组件，这个实例化的 JavaBean 组件对象将可以在这个 JSP 程序的其它地方被调用。<jsp:useBean>操作指令的基本语法形式如下所示：

<jsp:useBean id="*name*" scope="*page*|*request*|*session*|*application*" *typeSpec* />
或者
<jsp:useBean id="*name*" scope="*page*|*request*|*session*|*application*" *typeSpec* />
body
</jsp:useBean>

语法参数描述：

- id 属性用来设定 JavaBeans 的名称，利用 id，可以识别在同一个 JSP 程序中使用的不同的 JavaBeans 组件实例。
- class 属性指定 JSP 引擎查找 JavaBeans 代码的路径，一般是这个 JavaBean 所对应的 Java 类名。
- scope 属性用于指定 JavaBeans 实例对象的生命周期。亦即这个 JavaBean 的有效作用范围。scope 的值可能是 page、request、session 以及 application。在下面 1.3 节中，我们会详细讨论这四个属性值的含义与用法。

typeSpec 可能是如下的四种形式之一：

class="className"
或者
class="className" type="typeName"
或者
beanName="beanName" type=" typeName"
或者
type="typeName"

当 JavaBeans 组件对象被实例化以后，你就可以访问它的属性来定制它。我们要获得它的属性值，应当使用<jsp:getProperty>操作指令或者是在 JSP 程序段中直接调用 JavaBeans 对象的 getXXX()方法。<jsp:getProperty>操作指令的语法形式如下所示：

<jsp:getProperty id="Name" property="name" />

使用这个操作指令可以获取将要用到的 JavaBeans 组件实例对象的属性值。实际的值将会被放在输出语句中。

要改变 JavaBeans 的属性，你必须使用<jsp:setProperty>操作指令或者是直接调用 JavaBeans 对象的方法。<jsp:setProperty>操作指令有以下两种语法形式：

<jsp:setProperty id="Name" property="*" />

或者

<jsp:setProperty id="Name" property="propertyNumber" value="string" />

前者的功能是根据已提交的表单中的数据，设置这个 JavaBean 中相应(JavaBeans 属性的名称和表单对象的名称要相同)的属性值。后者的功能是把这个 JavaBeans 的指定的属性设为指定的值。

为了能在 JSP 程序中使用 JavaBeans 组件，你需要特别注意 JavaBeans 类程序的存放问题:为了使应用程序服务器能找到 JavaBeans 类，你需要将其类文件放在 Web 服务器的一个特殊位置。以 JSWDK1.0.1 服务器为例，JavaBeans 的类文件(编译好的 class 文件)应该放在 examples\WEB-INF\jsp\beans 目录下或者是 webpages\WEB-INF\jsp\beans 目录下面，在 resin 服务器中则是放在 doc\WEB-INF\classes 目录下的。至于 JavaBeans 在其他服务器下的存放路径，读者可以参考下文的介绍或者相应服务器的开发文档。

1.2.2 <jsp:setProperty>操作指令

<jsp:setProperty>操作指令被用于指定 JavaBeans 的某个属性的值，它的语法形式如下所示:

```
<jsp:setProperty name="BeanName" PropertyExpr />
PropertyExpr ::= property="*"|
                 property="PropertyName"|
                 property="PropertyName" value="PropertyValue"|
                 property="PropertyName" param="ParameterName"|
```

语法参数说明

- name：name 属性用来指定 JavaBeans 的名称，这个 JavaBeans 必须首先使用<jsp:useBean>操作指令来实例化。
- property：property 属性被用来指定 JavaBeans 需要定制的属性的名称，如果 property 属性的值为*，那么会发生什么情况呢?请参考 1.4.2 小节。
- value：value 属性的值将会被赋给 JavaBeans 的属性。
- param：param 这个属性的作用很微妙，如果客户端传递过来的参数中，有一个参数的名字和 param 属性的值相同，那么这个参数的值将会被赋给 JavaBean 的属性，所以使用了 param 属性就不要使用 value 属性，反之，使用了 value 属性就不要使用 param 属性。这两个属性是互斥的，不过 param 属性必须和 property 属性搭配使用，否则就不知道该赋值给 JavaBeans 的哪一个属性了。

我们不提倡读者使用<jsp:setProperty>操作指令，而应该在 JSP 程序段中直接调用 JavaBeans 组件实例对象的 setXXX()方法，因为后者的代码简单，使用起来比较灵活，相对而言，前一种方法的代码就比较繁琐了，而且灵活性也不好，以 param 属性为例，客户端传递归来的参数值一般不应该直接赋给 JavaBeans 的属性，而应该先转换汉字的内码，再赋值，这一点上 param 属性就无能为力了。

1.2.3 <jsp:getProperty>操作指令

<jsp:getProperty>操作指令搭配<jsp:useBean>操作指令一起使用，可以获取某个 JavaBean 组件对象的属性值，并使用输出方法将这个值输出到页面。<jsp:getProperty>操作指令的语法形式如下所示：

<jsp:getProperty name=”BeanName” Property=”PropertyName” />

语法参数说明

- name：这个属性用来指定 JavaBeans 的名称，这个 JavaBeans 组件对象必须已经使用<jsp:useBean>操作指令实例化了。
- Property：Property 用来指定要读取的 JavaBeans 组件对象的属性的名称。

实际上，我们也可以在 JSP 程序段中直接调用 JavaBeans 对象的 getXXX()方法，来获取 JavaBeans 对象的属性值，我们觉得使用这个方法要比使用<jsp:getProperty>操作指令好，因为前者使用起来比较灵活，而且代码相对比较简单。

1.2.4 JavaBeans 的开发流程

在这一小节里，我们将详细讨论如何开发 JavaBeans 组件，如何把它用到 JSP 程序的开发中去，实现一个完整的 JavaBeans+JSP 的开发流程。

编写 JavaBeans 组件

第一步，应该是编写一个 JavaBeans 组件程序，我们这就根据上面介绍的知识，编写一个十分简单的 JavaBeans 程序。请看程序清单 1.8(HelloWorld.java)。

程序清单 1.8

```
//File Name:HelloWorld.java
//Author:fancy
//Date:2001.3.26
//Note:use this JavaBean to say hello world!

package test;

public class HelloWorld
{
    String Hello="hello world，I am fancy!";

    public void HelloWorld()
    {

    }

    public void setHello(String name)
    {
```

北京希望电子出版社

```
        Hello=name;
    }

    public String getHello()
    {
        return Hello;
    }
}
```

在程序清单 1.8(HelloWorld.java)中，我们编写了一个十分简单的 JavaBean——HelloWorld，它有一个字符串类型的 Hello 属性，用于保存问候信息。在编写 HelloWorld.java 程序时，要注意 HelloWorld 类必须显式声明为 public 类型，其次是 package 语句的使用。请看代码行:

```
package test;
```

这一行代码指示编译器把编译好的类作为 test 包的一部分。HelloWorld.class(类文件)、HelloWorld.java(程序文件)文件必须位于 test 文件夹中。

编译 HelloWorld.java 程序

编写好 HelloWorld.java 程序以后，我们应该把它保存到哪里呢?以 JSWDK1.0.1 服务器为例，应该把它保存到 webpages\WEB-INF\jsp\beans\目录下面，我们必须新建一个文件夹，这个文件夹的名字必须和 package 语句所指定的包名相同。否则服务器无法找到 JavaBean 的类代码。在本例中，这个文件夹的名字应该是 test。保存好 HelloWorld.java 程序后，使用 javac.exe 程序把它编译为 class 文件。

编写 JSP 程序

第三步是编写 JSP 程序，调用我们在上面的步骤中编写好的 HelloWold 组件。请看程序清单 1.9(useBean.jsp)。

程序清单 1.9

```
<%--
File Name:useBean.jsp
Author:fancy
Date:2001.3.26
Note:use javabean to say hello world!
--%>

<jsp:useBean id="hello" scope="page" class="test.HelloWorld" />
<jsp:getProperty name="hello" property="Hello" />
<br>
<%
hello.setHello("Are you want to talk to me?");
%>
```

<%=hello.getHello()%>

在程序清单 1.9(useBean.jsp)中，首先使用<jsp:useBean>操作指令实例化了 HelloWorld 组件对象。在下面的代码中，就可以使用 hello 来引用 HelloWorld 组件对象。读者应该注意 class 属性设为 test.HelloWorld，其中 HelloWorld 代表类的名字，test 有两重含义，第一，HelloWorld 类属于 test 包，第二，HelloWorld 类文件保存在 test 文件夹中。所以 package 语句指定的包名，保存 JavaBeans 类的目标文件夹名，还有<jsp:useBean>操作指令中 class 属性值点号前的部分，这三个值一定要完全相同才行。否则 JSP 服务器都将不能找到相应的 JavaBeans 类。

接下来，使用<jsp:getProperty>操作指令获取 HelloWorld 组件对象 Hello 属性的值并把它输出。然后在 JSP 程序段和 JSP 表达式中分别调用 hello 对象的 setHello()方法和 getHello()方法，设定和获取该对象 Hello 属性的值。

程序清单 1.9 的运行效果如图 1.1 所示。



图 1.1　useBean.jsp 程序的运行效果

到此为止，一个完整的使用了 JavaBeans 组件的 JSP 项目就算开发成功了。这个开发流程虽然简单，不过凡是需要用到 JavaBeans 组件的 JSP 程序的开发，一般都应该遵循这个流程进行开发。

1.2.5　JavaBeans 的保存路径

在这一个小节中，我们将总结 JavaBeans 程序在不同的 JSP 服务器平台中的保存路径。在介绍这些知识以前，我们首先讨论 JavaBeans 程序的存储格式。

JavaBeans 组件被设计出来后，一般是以扩展名为 jar 的压缩格式文件存储，在 jar 文件中包含了与 JavaBeans 有关的信息，并以 MANIFEST 文件指定其中的哪些类是 JavaBeans 的类文件。以 jar 文件格式存储的 JavaBeans 程序在网络中传送时极大地减少了数据的传输数量，并把 JavaBeans 运行时所需要的一些资源捆绑在一起。jar 文件其实是一个 zip 文件，把它的扩展名改为 zip，就可以使用 Winzip 程序打开它。如何才能创建 jar 文件呢?答案是使用 JDK 的 jar.exe 程序，jar.exe 程序一般位于 JDK 的 bin 目录下面，这是一个命令行程序，它的用法如下：

jar {ctxu}[vfm0M] [jar-文件] [manifest-文件] [-C 目录] 文件名 ...

选项：

-c　创建新的归档
-t　列出归档内容的列表
-x　展开归档中的命名的（或所有的）文件
-u　更新已存在的归档
-v　生成详细输出到标准输出上
-f　指定归档文件名
-m　包含来自指定的清单（manifest）文件的清单（manifest）信息
-0　只存储方式；未用 ZIP 压缩格式
-M　不产生所有项的清单（manifest）文件
-i　为指定的 jar 文件产生索引信息
-C　改变到指定的目录，并且包含下列文件：
如果一个文件名是一个目录，它将被递归处理。
清单（manifest）文件名和归档文件名都需要被指定，按'm' 和 'f'标志指定的相同顺序。

示例 1：将两个 class 文件归档到一个名为 'classes.jar' 的归档文件中：

```
jar cvf classes.jar Foo.class Bar.class
```

示例 2：用一个存在的清单（manifest）文件 'mymanifest' 将 foo/ 目录下的所有文件归档到一个名为 'classes.jar' 的归档文件中：

```
jar cvfm classes.jar mymanifest -C foo/ .
```

下面我们总结 JavaBeans 类在不同的 JSP 服务器平台下面的保存位置。

JSWDK1.0.1 服务器

保存路径为：
（1）webpages\WEB-INF\jsp\beans\folderName
（2）examples\WEB-INF\jsp\beans\folderName

JRun 3.0 服务器

保存路径为：
（1）servers\default\default-app\WEB-INF\classes\folderName
（2）servers\default\demo-app\WEB-INF\classes\folderName

Tomcat 3.1/3.2 服务器

保存路径为：
（1）webapps\admin\WEB-INF\classes\folderName
（2）webapps\examples\WEB-INF\classes\folderName
（3）webapps\ROOT\WEB-INF\classes\folderName
（4）webapps\test\WEB-INF\classes\folderName
（5）webapps\XmlServlet\WEB-INF\classes\folderName

Resin1.2 服务器

保存路径为：

（1）doc\WEB-INF\classes\folderName

（2）doc\examples\..\WEB\INF\classes\folderName

限于篇幅，关于 JavaBeans 程序在各个 JSP 服务器平台下的保存路径，我们就介绍到这里，如果读者还希望了解 JavaBeans 程序在其他 JSP 服务器平台下的保存路径，请参考相应的服务器开发文档，或者是与本书作者联系。

1.3　JavaBeans 的 Scope 属性

对于 JSP 程序而言，使用 JavaBeans 组件不仅可以封装许多信息，而且还可以将一些数据处理的逻辑隐藏到 JavaBeans 的内部，除此之外，我们还可以设定 JavaBeans 的 Scope 属性，使得 JavaBeans 组件对于不同的任务，具有不同的生命周期和不同的使用范围。在前面，我们已经提到过 Scope 属性具有四个可能的值，分别是：application、session、request、page，分别代表 JavaBeans 的四种不同的生命周期和四种不同的使用范围。下面我们就分别介绍这四种不同的情况。

1.3.1　Application Scope

如果 JavaBeans 的 Scope 属性被指定为 application，也就是说这个 JavaBean 组件具有 Application Scope，这是什么意思呢?如果一个 JavaBean 组件具有 Application Scope，那么它的生命周期和 JSP 的 Application 对象同步，作用范围也和 Application 对象一样。使用这种类型的 JavaBeans 组件，可以在多个用户之间共享全局信息。具体来说，它的生命周期是这样子的：如果某个 JSP 程序使用<jsp:useBean>操作指令创建了一个 JavaBean 对象，而且这个 JavaBean 组件具有 Application Scope，那么这个 JavaBean 就一直在服务器的内存空间中待命，随时处理客户端的请求，直到服务器关闭为止，它所保存的信息才消失，它所占用的系统资源才会被释放。 在此期间，如果有若干个用户请求的 JSP 程序中，需要用到这个 JavaBean 组件，那么服务器在执行<jsp:useBean>操作指令时，并不会创建新的 JavaBean 组件，而是创建源对象的一个同步拷贝，在任何一个拷贝对象上发生的改变都会影响到源对象，源对象也会做出同步的改变，不过这个状态的改变不会影响其他已经存在的拷贝对象。这种类型的 JavaBeans 组件的功能和 JSP 的 Application 对象十分类似，不过前者的功能要强大得多，而且可以自由扩展，用起来也方便得多。请看程序清单 1.10(Counter.java)、程序清单 1.11(useCounter.jsp)。

程序清单 1.10

```
//File Name:Counter.java
//Author:fancy
//Date:2001.3.26
//Note:use this JavaBean to Counter!

package test;
```

```
public class Counter
{
    int Count=1;

    public void Counter()
    {

    }

    public void addCount()
    {
        Count++;
    }

    public int getCount()
    {
        return Count;
    }
}
```

在程序清单 1.10 中，我们定义了一个 Counter Bean，这个 JavaBean 组件可以用于记录访问者的人数。由于这个程序十分简单，我们就不多做介绍了。

程序清单 1. 11

```
<%--
File Name:useCounter.jsp
Author:fancy
Date:2001.3.26
Note:use javabean to say hello world!
--%>

<jsp:useBean id="counter" scope="application" class="test.Counter" />
<br>
你好，你是第
<%
out.println(counter.getCount());
counter.addCount();
%>位访客。
```

程序清单 1.11(useCounter.jsp)中，首先使用<jsp:useBean>操作指令引入了 JavaBean 组件——Counter，并且声明它的 Scope 为 Application，这一步十分重要，然后调用 Counter 组件的 getCount()方法，获取访问过这个 JSP 程序的人数，如果 Counter 组件刚刚被创建，那么这个方法将会返回缺省值——1，接着调用 Counter 组件的 addCounte()方法，把访问人数加上 1。

程序清单 1.11 的运行效果如图 1.2 所示。

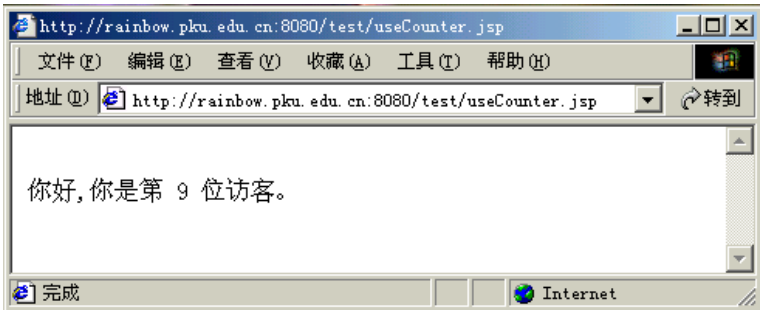


图 1.2　useCounter.jsp 的运行效果

1.3.2　Session Scope

如果一个 JavaBean 组件的 Scope 属性值为 session，那么这个 JavaBean 组件的生命周期、作用范围就和 JSP 的 Session 对象的生命周期、作用范围一样。也就是说：这一类型的 JavaBeans 组件的生命周期就是某个会话过程所经历的时间。也许有的读者对会话过程还不太了解，实际上，会话过程是对于单个用户而言的，会话过程的开始以用户开始访问某个网站为标志，会话过程的结束以用户结束对该网站的访问为标志。不同的用户对应着不同的会话过程，不同的会话过程之间互不干涉，互不影响。假设用户 A 第一次访问了某个网站的某个 JSP 程序，而这个 JSP 程序用到了一个 Scope 属性为 session 的 JavaBean 组件，那么服务器会自动创建这个 JavaBean 组件的实例对象，并且当 A 用户继续访问同一网站其他的 JSP 程序，而其他的 JSP 程序又用到同一个 JavaBean 对象时，那么服务器不会创建新的 JavaBean 对象，而是使用已经存在的 JavaBean 对象实例，也就是说在第一个 JSP 程序中创建的 JavaBean 组件对象在这个用户访问的同一网站的所有的 JSP 程序中都是可用的，而且这个 JavaBean 组件对象的状态保持唯一性。如果有另一个用户 B，访问了用户 A 访问过的 JSP 程序，那么服务器是否会不创建新的 JavaBean 组件对象，而使用由于用户 A 访问而创建的 JavaBean 组件对象呢?答案是否定的。服务器将会为用户 B 创建只属于他的 JavaBean 组件对象。这个新创建的 JavaBean 组件对象在用户 B 访问的同一网站的所有 JSP 程序中都是直接可用的，而不需要创建一个新的组件。并且属于用户 A 的 JavaBean 组件对象和属于用户 B 的组件对象都是唯一的，它们之间互不干涉。这里我们讨论的只是两个用户的情况，其实如果有多个用户在线，情况也一样。

综上所述，Scope 属性为 session 的 JavaBeans 组件的功能、作用范围都和 JSP 的 Session 对象十分类似，不过前者的功能比后者要强大得多，并且使用起来也灵活得多。具有可扩展性，后者没有扩展性。

下面我们就利用这种类型的 JavaBeans 组件，来编写一个特殊的计数器程序，这个计数器并不是统计一个网页的访问人数，而是统计一个用户所访问的页面数目。请看程序清单 1.12(beanPage1.jsp)、程序清单 1.13(beanPage2.jsp)。

程序清单 1.12

```
<%--
File Name:beanPage1.jsp
```

```
Author:fancy
Date:2001.3.26
Note:use Counter to calculate how many pages this user have visited
--%>

<jsp:useBean id="counter" scope="session" class="test.Counter" />
<br>
第一页
<br>
你好，你已经访问了
<%
out.println(counter.getCount());
counter.addCount();
%>个页面。
```

在程序清单 1.12 中，我们使用的 JavaBean 组件仍然是我们在程序清单 1.10 中编写的 Counter，不过这里 Counter 对象的 Scope 属性值是 session，而不是 Application。当用户首先调用 beanPage1.jsp 程序时，Counter 对象被创建了。程序清单 1.12 的运行效果如图 1.3 所示。

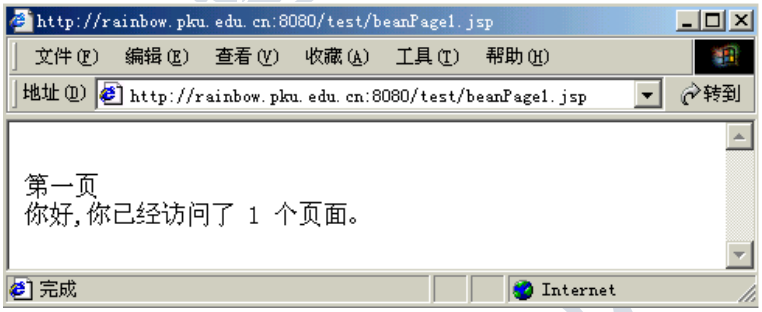


图 1.3　beanPage1.jsp 程序的运行效果

程序清单 1.13

```
<%--
File Name:beanPage2.jsp
Author:fancy
Date:2001.3.26
Note:use Counter to calculate how many pages this user have visited
--%>

<jsp:useBean id="counter" scope="session" class="test.Counter" />
<br>
第二页
<br>
你好，你已经访问了
<%
```

```
out.println(counter.getCount());
counter.addCount();
%>个页面。
```

程序清单 1.13(beanPage2.jsp)程序与程序清单 1.12 基本上是一样的，如果我们首先调用了程序清单 1.12，再访问程序清单 1.13，那么服务器在执行<jsp:useBean>操作指令时，只是返回在程序清单 1.12 中创建的 Counter 对象，而不会创建新的 Counter 对象(即使两个程序文件中的<jsp:useBean>操作指令的 id 属性值不同，也不会创建新的 JavaBean 对象)。程序清单 1.13(beanPage2.jsp)的运行效果如图 1.4 所示。

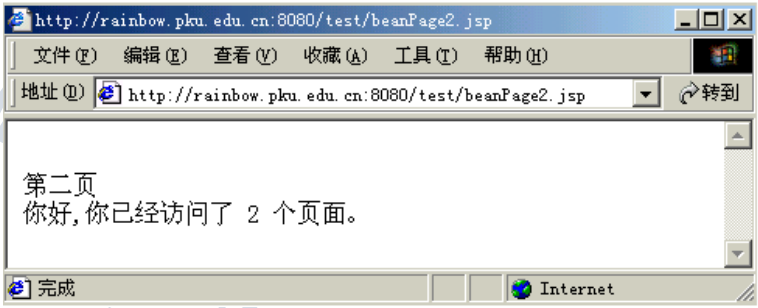


图 1.4　beanPage2.jsp 程序的运行效果

请读者想一想，如果首先调用 beanPage2.jsp 程序，再调用 beanPage1.jsp 程序，那么运行效果还会是这样吗?如果不是，那么又应该如何呢?读者不妨试一试。

1.3.3　Request Scope

如果 JavaBeans 的 Scope 属性值被设为 request，那么这种类型的 JavaBeans 组件对象又有何特性呢?可能读者已经猜到了，这种类型的 JavaBeans 组件对象的生命周期和作用范围和 JSP 的 Request 对象一样。当一个 JSP 程序使用<jsp:forward>操作指令定向到另外一个 JSP 程序或者是使用<jsp:include>操作指令导入另外的 JSP 程序，那么第一个 JSP 程序会把 Request 对象传送到下一个 JSP 程序，而属于 Request Scope 的 JavaBeans 组件对象也将伴随着 Request 对象送出，被第二个 JSP 程序接收。因此，所有通过这两个操作指令连接在一起的 JSP 程序都可以共享一个 Request 对象，共享这种类型的 JavaBeans 组件对象。这种类型的 JavaBeans 组件对象使得 JSP 程序之间传递信息更为容易，不过美中不足的是这种 JavaBeans 不能够用于客户端与服务端之间传递信息，因为客户端是没有办法执行 JSP 程序，创建新的 JavaBeans 对象的。

下面的程序清单 1.14(RequestBean.java)、程序清单 1.15(beanPage3.jsp)和程序清单 1.16(beanPage4.jsp)演示了这种 JavaBeans 组件对象的使用方法。

程序清单 1.14

```
//File Name:RequestBean.java
//Author:fancy
//Date:2001.3.26
//Note:use this JavaBean to transfer info between two jsp program
```

```
package test;

public class RequestBean
{
    String url="index.jsp";

    public void Counter()
    {

    }

    public void setURL(String strURL)
    {
        url=strURL;
    }

    public String getURL()
    {
        return url;
    }
}
```

在程序清单 1.14(RequestBean.java)中创建的 RequestBean Bean 可以用于保存当前页面的名称，缺省值为 index.jsp。在下面的程序清单 1.15(beanPage3.jsp)和程序清单 1.16(beanPage4.jsp)中，将要使用这个 JavaBean 组件实现一个小功能。

程序清单 1.15

```
<%--
File Name:beanPage3.jsp
Author:fancy
Date:2001.3.26
Note:use RequestBean to pass info before two different jsp page
--%>

<jsp:useBean id="reqBean" scope="request" class="test.RequestBean" />
调用页:beanPage3.jsp
<%
reqBean.setURL("beanPage3.jsp");
%>
<br>
<br>
<jsp:include page="beanPage4.jsp" flush="true" />
```

在程序清单 1.15 中(beanPage3.jsp)，首先使用<jsp:useBean>操作指令创建一个新的 RequestBean 组件对象 reqBean。然后在 JSP 程序段中使用 setURL()方法把当前程序的名称

beanPage3.jsp 保存在 reqBean 对象的 url 属性中，接下来使用<jsp:include>操作指令导入 beanPage4.jsp 程序。

程序清单 1.16

```
<%--
File Name:beanPage4.jsp
Author:fancy
Date:2001.3.26
Note:use RequestBean to pass info before two different jsp page
--%>

<jsp:useBean id="reqBean" scope="request" class="test.RequestBean" />
被调用页:beanPage4.jsp
<br>
本页面由
<%
out.println(reqBean.getURL());
%>
调用
```

在程序清单 1.16(beanPage4.jsp)中，同样使用<jsp:useBean>操作指令以便获取 RequestBean 组件对象的实例，由于在执行 beanPage3.jsp 程序的时候，已经创建了 RequestBean 组件的实例对象，所以在这里就不用创建新的 RequestBean 对象，简单地引用已经存在的组件对象即可。然后，在 JSP 程序段中调用 RequstBean 组件的 getURL()方法，获取使用<jsp:include>操作指令的 JSP 程序的名称。在本例中，这个值应该是 beanPage3.jsp，程序清单 1.15(beanPage3.jsp)的运行效果如图 1.5 所示。

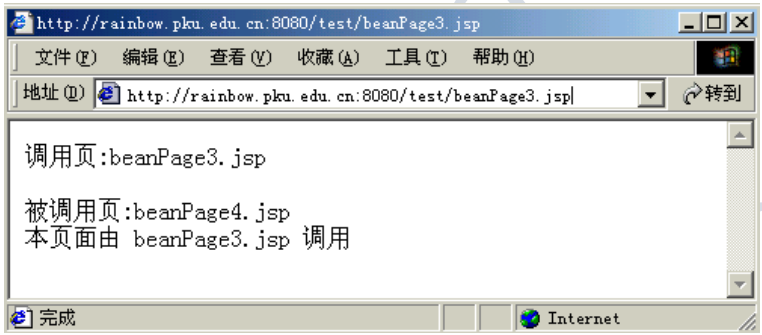


图 1.5　beanPage3.jsp 程序的运行效果

如果不事先运行 beanPage3.jsp，而是直接运行 beanPage4.jsp 程序，那么运行效果应该如何呢?根据我们前面介绍的知识，在执行 beanPage4.jsp 程序中的<jsp:useBean>操作指令时，由于 beanPage4.jsp 没有接受到现存的 RequestBean 组件对象，那么 JSP 服务器将会创建一个新的 RequestBean 对象。接下来，调用 getURL()方法的时候，由于 RequestBean 的 url 属性未经重新赋值，所以 getURL()方法的执行结果是返回缺省值 index.jsp。程序清单 1.16(beanPage4.jsp)的运行效果如图 1.6 所示。

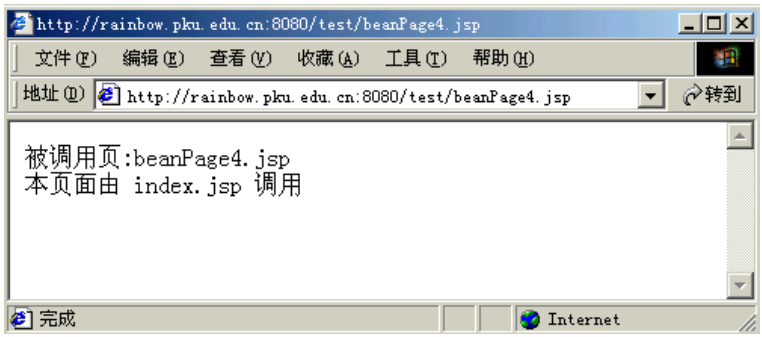


图 1.6　beanPage4.jsp 程序的运行效果

1.3.4　Page Scope

如果一个 JavaBean 的 Scope 属性被设为 page，那么它的生命周期和作用范围在这四种类型的 JavaBean 组件中是最小的。Page Scope 类型的 JavaBeans 组件的生命周期为 JSP 程序的运行周期，当 JSP 程序运行结束，那么该 JavaBean 组件的生命周期也就结束了。Page Scope 类型的 JavaBeans 组件程序的作用范围只限于当前的 JSP 程序中，它无法在别的 JSP 程序中起作用，对应于不同的客户端请求，服务器都会创建新的 JavaBean 组件对象，而且一旦客户端的请求执行完毕，那么该 JavaBean 对象会马上注销，无法为别的客户端请求所使用。

下面的程序清单 1.17(JspCalendar.java)、程序清单 1.18(date.jsp)演示了 Page Scope 的 JavaBeans 组件的用法。

程序清单 1.17

```
//File Name:JspCalendar.java
//Author:fancy
//Date:2001.3.26
//Note:use this JavaBean to get current time

package test;

import java.text.DateFormat;
import java.util.*;

public class JspCalendar
{
    Calendar    calendar = null;

    public JspCalendar()
    {
        calendar = Calendar.getInstance(TimeZone.getTimeZone("PST"));
        Date trialTime = new Date();
        calendar.setTime(trialTime);
```

```
}

public int getYear()
{
    return calendar.get(Calendar.YEAR);
}

public String getMonth()
{
    int m = getMonthInt();
    String[] months = new String []
    {"January", "February", "March",
     "April", "May", "June",
     "July", "August", "September",
     "October", "November", "December"
    };
    if (m > 12)
        return "Unknown to Man";

    return months[m - 1];
}

public String getDay()
{
    int x = getDayOfWeek();
    String[] days = new String[]
    {"Sunday", "Monday", "Tuesday", "Wednesday",
     "Thursday", "Friday", "Saturday"
    };

    if (x > 7)
       return "Unknown to Man";

    return days[x - 1];

 }

public int getMonthInt()
{
    return 1 + calendar.get(Calendar.MONTH);
}

public String getDate()
```

北京希望电子出版社

```
{
        return getMonthInt() + "/" + getDayOfMonth() + "/" +    getYear();

}

public String getTime()
{
        return getHour() + ":" + getMinute() + ":" + getSecond();
}

public int getDayOfMonth()
{
        return calendar.get(Calendar.DAY_OF_MONTH);
}

public int getDayOfYear()
{
        return calendar.get(Calendar.DAY_OF_YEAR);
}

public int getWeekOfYear()
{
        return calendar.get(Calendar.WEEK_OF_YEAR);
}

public int getWeekOfMonth()
{
        return calendar.get(Calendar.WEEK_OF_MONTH);
}

public int getDayOfWeek()
{
        return calendar.get(Calendar.DAY_OF_WEEK);
}

public int getHour()
{
        return calendar.get(Calendar.HOUR_OF_DAY);
}

public int getMinute()
{
        return calendar.get(Calendar.MINUTE);
```

北京希望电子出版社

```
}

public int getSecond()
{
    return calendar.get(Calendar.SECOND);
}

public int getEra()
{
    return calendar.get(Calendar.ERA);
}

public String getUSTimeZone()
{
    String[] zones = new String[]
    {"Hawaii"，"Alaskan"，"Pacific"，
        "Mountain"，"Central"，"Eastern"
    };

    int index = 10 + getZoneOffset();

    if (index <= 5)
    {
        return zones[10 + getZoneOffset()];
    }
    else
    {
        return "Only US Time Zones supported";
    }
}

public int getZoneOffset()
{
    return calendar.get(Calendar.ZONE_OFFSET)/(60*60*1000);
}

public int getDSTOffset()
{
    return calendar.get(Calendar.DST_OFFSET)/(60*60*1000);
}

 public int getAMPM()
```

北京希望电子出版社

```
    {
        return calendar.get(Calendar.AM_PM);
    }
}
```

在程序清单 1.17(JspCalendar.java)中，定义了一个 JavaBean——JspCalendar，JspCalendar 组件中有很多 getXXX()方法，可以获取各种各样的时间信息。关于这些方法的细节问题，我们在这里就不详细讨论了，这不是本书的主题，对此感兴趣的读者，可以参考相应的 Java 文档。

程序清单 1.18

```
<%--
File Name:date.jsp
Author:fancy
Date:2001.4.1
Note:use JspCalendar bean to show the time info.
--%>

    <html>
<body bgcolor="white">
<jsp:useBean id='clock' scope='page' class='test.JspCalendar'/>

<font size=4>
<ul>
<li>Day of month: is <jsp:getProperty name="clock" property="dayOfMonth"/>
<li>Year: is  <jsp:getProperty name="clock" property="year"/>
<li>Month: is  <jsp:getProperty name="clock" property="month"/>
<li>Time: is  <jsp:getProperty name="clock" property="time"/>
<li>Date: is  <jsp:getProperty name="clock" property="date"/>
<li>Day: is  <jsp:getProperty name="clock" property="day"/>
<li>Day Of Year: is  <jsp:getProperty name="clock" property="dayOfYear"/>
<li>Week Of Year: is  <jsp:getProperty name="clock" property="weekOfYear"/>
<li>era: is  <jsp:getProperty name="clock" property="era"/>
<li>DST Offset: is  <jsp:getProperty name="clock" property="dSTOffset"/>
<li>Zone Offset: is  <jsp:getProperty name="clock" property="zoneOffset"/>
</ul>
</font>
</body>
</html>
```

程序清单 1.18(date.jsp)程序十分简单，只是调用 JspCalendar Bean 的各种 getXXX()方法，输出各种各样的时间信息。读者请注意 JspCalendar Bean 的 Scope 属性是 page。这就是说，每次刷新当前页面时，该 JavaBean 对象就会被重新创建，重新获取时间信息，所以每次执行 date.jsp 程序，每次的结果都不一样。程序清单 1.18 的执行结果见图 1.7、图 1.8。

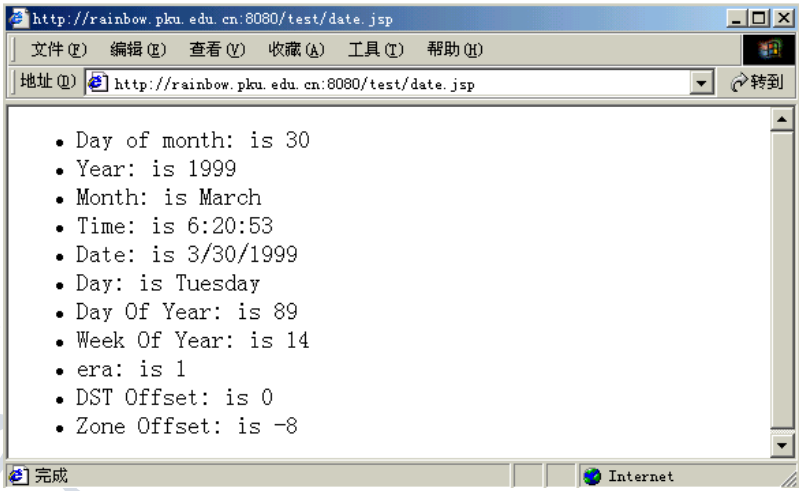


图 1.7　date.jsp 程序的运行结果(第一次运行)

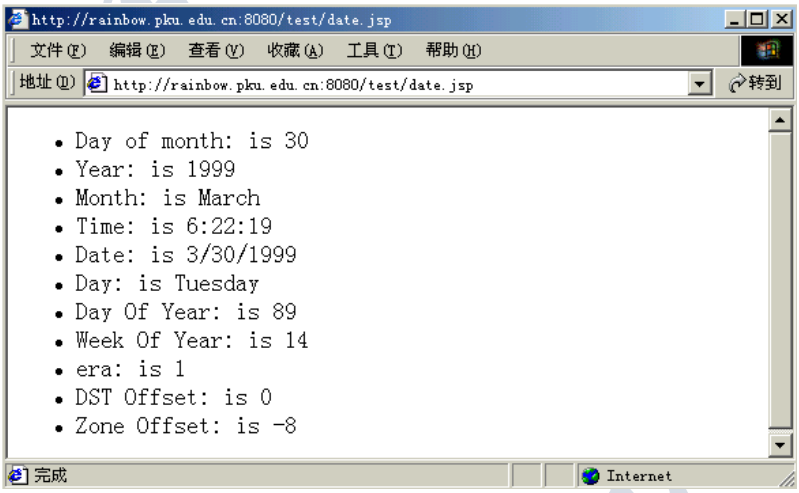


图 1.8　date.jsp 程序的运行结果(第二次运行)

由图 1.7 和图 1.8 不难看出，date.jsp 程序前后两次的运行结果显然不同。接下来请看下面的程序清单 1.19(date1.jsp)。

程序清单 1. 19

```
<%--
File Name:date.jsp
Author:fancy
Date:2001.4.1
Note:use JspCalendar bean to show the time info.
--%>

<html>
<body bgcolor="white">
<jsp:useBean id='clock' scope='application' class='test.JspCalendar'/>
```

```
<font size=4>
<ul>
<li>Day of month: is <jsp:getProperty name="clock" property="dayOfMonth"/>
<li>Year: is  <jsp:getProperty name="clock" property="year"/>
<li>Month: is  <jsp:getProperty name="clock" property="month"/>
<li>Time: is  <jsp:getProperty name="clock" property="time"/>
<li>Date: is  <jsp:getProperty name="clock" property="date"/>
<li>Day: is  <jsp:getProperty name="clock" property="day"/>
<li>Day Of Year: is  <jsp:getProperty name="clock" property="dayOfYear"/>
<li>Week Of Year: is  <jsp:getProperty name="clock" property="weekOfYear"/>
<li>era: is  <jsp:getProperty name="clock" property="era"/>
<li>DST Offset: is  <jsp:getProperty name="clock" property="dSTOffset"/>
<li>Zone Offset: is  <jsp:getProperty name="clock" property="zoneOffset"/>
</ul>
</font>
实际时间:<%=new java.util.Date()%>
</body>
</html>
```

程序清单 1.19 和程序清单 1.18 几乎完全相同。只不过在程序清单 1.19 中 JspCalendar 组件的 Scope 属性值为 application，而程序清单 1.19 中 JspCalendar 组件的 Scope 属性值为 page，并且在程序清单 1.19 的最后还使用了另一种方法获取当前的时间，以便对照，那么程序清单 1.19 的运行效果与程序清单 1.18 相比，究竟有什么不同呢?请看图 1.8 和图 1.9。

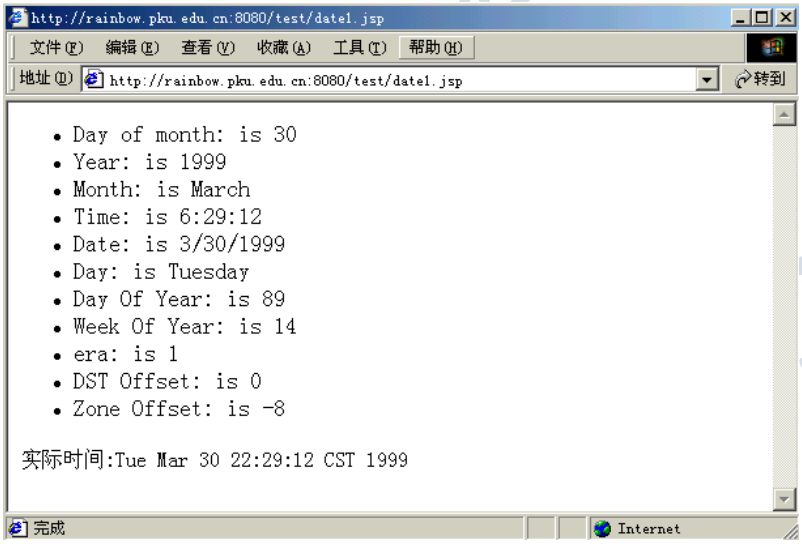


图 1.8　date1.jsp 程序的运行效果(第一次)

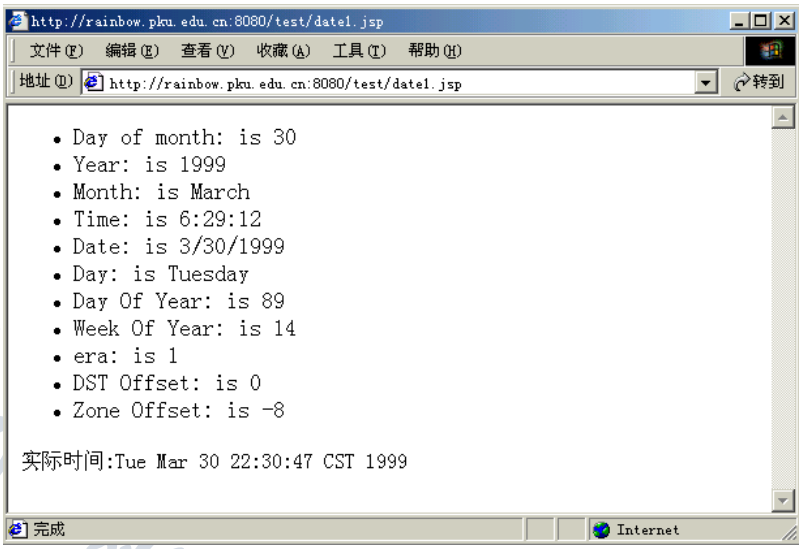


图 1.9　date1.jsp 程序的运行效果(第二次)

由图 1.8 不难看出，两种方法获取的时间一样。在图 1.9 中，第一种方法获取的时间和第二种方法获取的时间有着明显的差距，大概有 1 分半钟左右，即使考虑到代码执行先后的时间间隔，也没有办法解释有这样大的时间差距，但是第一种方法获取的时间(图 1.9)和图 1.8 所显示的时间一模一样。这说明了一个问题，即在 date1.jsp 程序中，JspCalander 组件的代码只被执行一次，不管如何刷新页面，JspCalendar Bean 组件的 getXXX()方法都是返回 JspCalendar Bean 组件第一次被初始化时的时间，这就是 Application Scope 类型的 JavaBeans 与 Page Scope 类型的 JavaBeans 的不同之处。

1.4　JavaBeans 应用实例

在这一节中，我们将结合以前介绍过的知识，讨论 JavaBeans 组件技术如何与 JSP 技术结合在一起，开发功能强大的 JSP 程序。限于篇幅，我们只能够介绍如何用 JavaBeans 封装数据库操作和购物车功能这两个方面的内容，至于其他的方面，读者可以仿照下面的例子，自己探索。

1.4.1　JavaBeans 封装数据库操作

这一小节我们就来介绍如何使用 JavaBeans 封装数据库操作的功能。我们首先需要创建一个可以访问数据库的 JavaBean 组件，然后在 JSP 程序中调用它。请看程序清单 1.20(JDBCBean.java)。

程序清单 1.20

```
//File Name:
//Author:fancy
//Date:2001.4.1
//Note:to visit the database
```

```
package test;

import java.io.*;
import java.sql.*;

public class JDBCBean
{
    String Hello="hello world，I am fancy";

    public void JDBCBean()
    {

    }

    public ResultSet connect()
    {
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Connection conn = DriverManager.getConnection("jdbc:odbc:test"，"sa"，"");
            Statement stmt=conn.createStatement();
            ResultSet rs=stmt.executeQuery("USE fancy SELECT * FROM goods");
            return rs;
        }
        catch(Exception fe)
        {

        }
    return null;
    }
}
```

JDBCBean.java 程序中定义了 JDBCBean 类，使用该类的 connect()方法可以访问数据库。JDBCBean 类的 connect()方法采用标准的流程访问数据库，首先是载入数据库驱动程序，然后是创建数据库连接对象 conn，利用这个连接对象创建 SQL 语句对象 stmt，接下来利用 stmt 对象的 executeQuery()方法向数据库系统发送 SQL 语句，该方法的返回值是 ResultSet 接口的实例对象。

在 JSP 程序中，如何使用这个 JavaBean 组件访问数据库呢？请看程序清单 1.21(JDBCBean.jsp)。

程序清单 1.21

```
<%--
File Name:JDBCBean.jsp
Author:fancy
```

```
Date:2001.3.26
Note:use JDBCBean to access the db System
--%>

<%@ page import="java.sql.*"%>
<jsp:useBean id="db" class="test.JDBCBean" />
<%
ResultSet rs=db.connect();
while(rs.next())
{
    out.println(rs.getObject(1)+"-->"+rs.getObject(3)+"<br>");
}
%>
```

在程序清单 1.21(JDBCBean.java)中，应用 JDBCBean Bean 实现了访问数据库的功能，首先使用<jsp:useBean>操作指令创建 JDBCBean 组件的实例对象 db，然后在 JSP 程序段中直接调用 db 对象的 connect()方法，获取 ResultSet 接口的实例对象 rs，接下来使用一个 while 循环结构遍历这个记录集对象，获取记录集第三个字段的值，并把它们一一输出。

程序清单 1.21 的运行效果如图 1.10 所示。

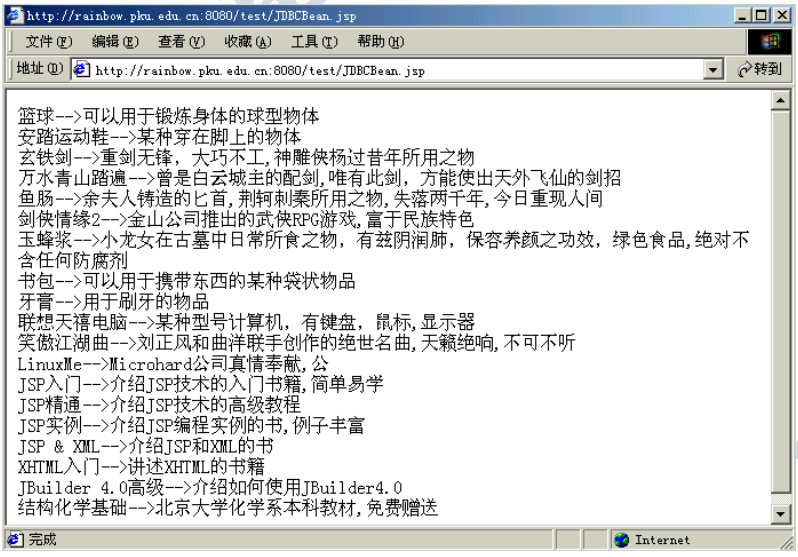


图 1.10　JDBCBean.jsp 程序的运行效果

上面所举的例子十分简单，例如 SQL 语句已经指定了，我们没有办法改变它，除非重新编写 JDBCBean.java 程序。那么如何解决这个问题呢?答案是重载 JDBCBean 类的 connect()方法。让它可以接受更多的数据库访问参数，包括用户名、密码、SQL 语句等参数。这样对于 JDBCBean.jsp 程序来说，无需作多大改动，甚至不需要改动，功能也可以大大扩展。

注意：本例所使用的 JDBC 驱动程序为 JDBC-ODBC 桥，数据库系统为 MS SQL Server7.0 Desktop Edition。操作系统为 Windows Me。关于数据库的结构信息，

读者可以参考本书附录 4 的内容。

1.4.2　JavaBeans 和购物车功能

本小节中我们将要使用 JavaBeans 组件实现一个简单的购物车。请看程序清单 1.22(carts.html)，这是购物车的页面程序。

程序清单 1.22

```
<!—
File Name:carts.html
Author:fancy
Date:2001.4.1
Note:show the goods
-->

<html>
<head>
      <title>carts</title>
</head>
<body bgcolor="white">
<font size = 5 color="#CC0000">
<form type=POST action=carts.jsp>
<BR>
Please enter item to add or remove:
<br>
Add Item:

<SELECT NAME="item">
<OPTION>Beavis & Butt-head Video collection
<OPTION>X-files movie
<OPTION>Twin peaks tapes
<OPTION>NIN CD
<OPTION>JSP Book
<OPTION>Concert tickets
<OPTION>Love life
<OPTION>Switch blade
<OPTION>Rex，Rugs & Rock n' Roll
</SELECT>
<br> <br>
<INPUT TYPE=submit name="submit" value="add">
<INPUT TYPE=submit name="submit" value="remove">
</form>
</FONT>
</body>
```

北京希望电子出版社

</html>

程序清单 1.22(carts.html)的作用是给用户提供一个购物的界面，让他们可以自由地往购物车中添加商品，或者是删除某种商品。程序清单 1.22(carts.html)的运行效果如图 1.11 所示。

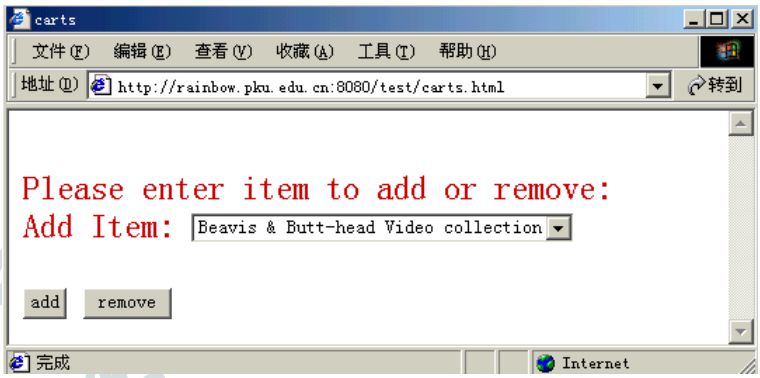


图 1.11　carts.html 的运行效果

接下来请看程序清单 1.23(DummyCart.java)。

程序清单 1. 23

```
//File Name:DummyCart.java
//Author:fancy
//Date:2001.4.1
//Note:a cart

package test;

import javax.servlet.http.*;
import java.util.Vector;
import java.util.Enumeration;

public class DummyCart
{
    Vector v = new Vector();
    String submit = null;
    String item = null;

    private void addItem(String name)
{
        v.addElement(name);
    }

    private void removeItem(String name)
{
```

```
            v.removeElement(name);
        }

        public void setItem(String name)
    {
            item = name;
        }

        public void setSubmit(String s)
        {
            submit = s;
        }

        public String[] getItems()
    {
            String[] s = new String[v.size()];
            v.copyInto(s);
            return s;
        }

        public void processRequest(HttpServletRequest request)
    {
            // null value for submit - user hit enter instead of clicking on
            // "add" or "remove"
            if (submit == null)
                addItem(item);

            if (submit.equals("add"))
                addItem(item);
            else if (submit.equals("remove"))
                removeItem(item);

            // reset at the end of the request
                reset();
        }

        // reset
        private void reset()
    {
            submit = null;
            item = null;
        }
    }
```

北京希望电子出版社

在 DummyCart.java 程序中，定义了购物车的基本模型。DummyCart 类使用 Vector 数据结构来模拟购物车的功能。DummyCart 有三个属性，分别是 submit，item，v，其中，submit 的值如果为 add，那么意味着往购物车中添加商品，如果为 remove，那么表示用户将从购物车中删除商品。Item 代表用户需要添加或者舍弃的商品的名字，v 是一个 Vector 类型的数据，它保存着购物车的所有信息，利用 Vector 类的方法，可以实现往购物车中添加商品或者删除商品的操作。在 DummyCart 类中，最重要的方法是 processRequest()，这个方法判断 submit 的值，然后调用 addItem()方法或者 removeItem()方法，完成基本的购物车操作。

下面请看程序清单 1.24（carts.jsp）。

程序清单 1. 24

```
<%—
File Name:carts.jsp
Author:fancy
Date:2001.4.1
Note:to show cart
--%>

<html>
<jsp:useBean id="cart" scope="session" class="test.DummyCart" />
<jsp:setProperty name="cart" property="*" />
<%
      cart.processRequest(request);
%>
<FONT size = 5 COLOR="#CC0000">
<br> You have the following items in your cart:
<ol>
<%
String[] items = cart.getItems();
for (int i=0; i<items.length; i++) {
%>
<li> <%= items[i] %>
<%
      }
%>
</ol>
</FONT>
<hr>
<%@ include file ="carts.html" %>
</html>
```

用户首先打开 carts.html 页面，选中某种商品，然后单击 submit 按钮，把数据提交到 carts.jsp 程序中，carts.jsp 程序首先使用<jsp:useBean>创建一个新的 Session Scope 类型的 JavaBean 组件对象 cart，如果此 cart 对象已经存在了，那就不用创建了，直接拿过来用就是了。Carts.jsp 程序接着使用<jsp:setProperty>操作指令给 carts 对象赋值，如下面的代码所

示。

例：

<jsp:setProperty name="cart" property="*" />

读者一定注意到了上面的代码行中 property 属性的值为*，这是什么意思呢?这就意味着从客户端传递过来的任何参数，只要名字和 cart 对象的属性名字相符，它们的值就会被赋给 cart 对象的相应的属性。例如如果用户选择了 JSP Book 这种商品，单击了 submit 按钮，那么 JSP 引擎会自动调用 cart 对象的 addItem()方法和 setSubmit()方法，把 item 属性的值设为 JSP Book，把 submit 的值设为 add。

接下来，carts.jsp 程序调用 processRequest()方法，让它根据 submit 属性的值完成相应的购物车操作。最后，carts.jsp 程序使用 cart 对象的 getItems()方法，配合 for 循环结构，把购物车的内容全部输出来。

程序清单 1.24 的运行效果如图 1.12 所示。

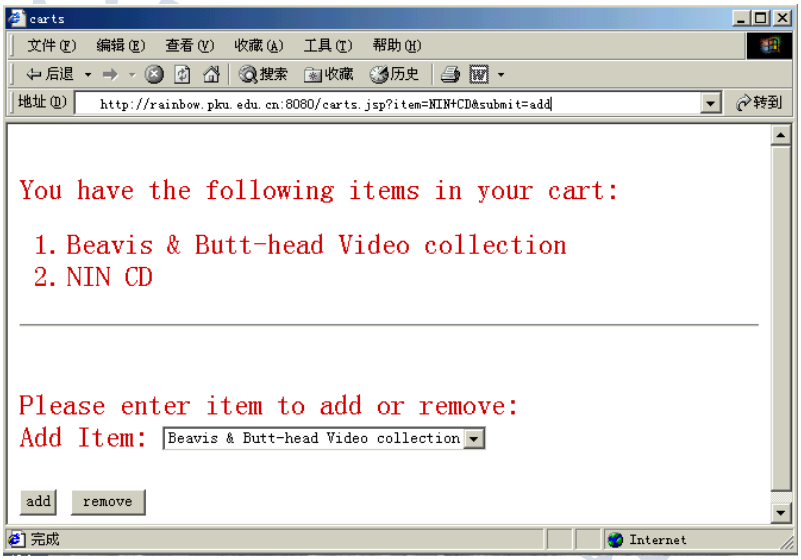


图 1.12　carts.jsp 程序的运行效果

1.5　本 章 小 结

在本章中，我们主要介绍了 JavaBeans 组件技术在 JSP 程序开发中的应用。主要内容包括：

JavaBeans 的属性、JavaBeans 的事件模型、JSP 中与 JavaBeans 相关的操作指令的语法与用法、JavaBeans 的开发流程、JavaBeans 的 Scope 属性、JavaBeans 封装数据库操作等内容。这部分知识十分重要，如果读者能够切实掌握 JavaBean 组件技术，并能够恰当地在 JSP 程序中使用它来封装一些复杂关键的操作，那么你就会发现 JSP 程序原来可以这样简单，但是功能又是这样强大。

在下一章，我们将向读者介绍目前如日中天的 EJB 技术。

第 2 章　Enterprise JavaBeans

EJB(Enterprise JavaBeans)是一个全新的面向服务端的组件框架，用于开发和部署面向对象的分布式的企业级的 Java 应用。用 EJB 技术编写的应用系统可扩展性好，支持并发性、支持事务处理及多用户、多线程环境下的安全性。EJB 应用程序编写一次，就可以在任何支持 EJB 技术的服务器平台上运行。它提供了比以往的中间层解决方案更好的集成性与互操作性，因此，对于高性能、高可扩展性、高安全性的应用系统，通过使用 EJB 组件技术，可以为用户提供更加高效的服务。在本章，我们就来介绍 EJB 技术。本章需要重点掌握的内容如下所示：

- EJB 的体系结构
- EJB 开发环境的配置
- 如何使用 JBuilder 4.0+IAS 4.1 开发 EJB
- 发布 EJB 组件服务
- 编写 EJB 组件服务的客户端

2.1　EJB 技术简介

2.1.1　EJB 技术的产生

EJB（Enterprise JavaBeans）是 Sun 公司推出的 J2EE 产品家族的成员。与其他 J2EE 技术一起，大大增强了 Java 的能力，并推动了 Java 语言在企业级应用系统中的应用。

目前，企业间的电子商务应用系统开发以及分布式计算系统、分布式处理系统的开发成了时尚，企业级应用的编程技术亦在不断发展，企业级应用模式更在不断变化。归纳起来，主要有以下几种发展趋势。

- 从传统的 Client/Server(客户端/服务端)模式向三层、多层、二维分布式应用模式转变。
- 对象、组件技术的不断发展。
- 命名(Naming)和目录(Directory)服务功能增强。
- 大量采用中间件技术。

根据企业级应用的发展趋势，Sun 公司提出了一个名为 JPE 的解决方案(全名为 Java Platform for the Enterprise，译为面向企业的 Java 平台)，为企业级电子商务应用系统的发展提供了坚实的基础。在 JPE 解决方案中，Enterprise JavaBeans(EJB)技术是其核心技术之一，主要用于简化三层或者多层应用系统的开发，进行大量的事务处理和分布式计算。

EJB 技术与其他的 J2EE 技术相互结合，如 JMS、JTS、JNDI、RMI 等技术，为企业级电子商务系统的开发提供了一个安全、可靠、灵活以及伸缩性很强的开发平台。EJB Specification 1.0(EJB 规范 1.0 版本)刚一提出，就得到了包括 IBM、Weblogic、Informix、Oracle、Allaire、Orion、Netscape 等众多厂商的支持，可以说，EJB 将是企业级应用系统开

发的一条“未来之路”。

2.1.2　EJB 组件模型概括

EJB 并不是一个产品。它是 Java 服务器端服务框架的规范之一，软件厂商根据这个规范来实现 EJB 容器(EJB 运行平台)的功能，以便为客户端的应用程序提供 EJB 组件服务。应用程序开发者可以专注于开发应用系统所需的商业逻辑，而不用担心具体服务框架的实现问题。

Sun 公司发布的 EJB 规范详细地解释了一些最小但是必需的服务，如事务处理、安全性、命名和目录服务等等。软件厂商必须遵循这些规范要求，开发支持 EJB 的产品，以便使得任何一个 EJB 组件能够在任意支持 EJB 的平台上使用某个 EJB 规范所规定的必须实现的服务。EJB 规范并没有规定软件厂商如何实现这些服务，它允许软件厂商在不牺牲 EJB 核心服务的可移植性(与平台无关性)的前提下来提供一些增强功能。

EJB 组件模型为中间层应用程序的开发提供了一致的基于组件的开发模型。EJB 组件模型像 Java 技术规范所规定的那样，“write once，run anywhere”(一次编写，处处运行)，具有与平台无关的平台独立性，是一种开放的组件模型。而目前除了 EJB 以外的其他中间层解决方案一般都是基于专有的应用平台，同时，需要有大量的底层代码开发工作，跨平台应用(移植)有困难。EJB 组件模型对这些方案进行了标准化，提供了一个标准的与平台无关的组件运行环境——EJB Server。用户不需要自己去编程，EJB Server 可以负责处理远过程调用、事务处理、安全性操作、多线程应用和组件状态管理等底层操作，简化了应用系统的开发过程。以前，这些处理都需要系统开发者编程实现，开发的难度很大。

EJB 组件模型增强了面向客户端的 JavaBeans 组件模型的功能，适用于整个企业分布式大规模应用系统的开发。我们在第一章已经介绍了一些关于 JavaBeans 组件技术的内容，那么 EJB 组件模型与 JavaBeans 组件模型相比又有何不同呢?

JavaBeans 组件模型特点

JavaBeans 组件模型是面向客户端的组件模型，它支持可移植和可重用的 Java 组件的开发。JavaBeans 组件可以工作于任何 Java 程序应用开发工具中。Java 程序开发者可以利用 Java 集成开发环境开发 Java Application、Java Applet 及 Java Servlet 等程序。我们可以认为，JavaBeans 组件是一个特殊的 Java Class，可以被加入到应用开发工程中，并被 Java 程序开发者所使用。在 Java 集成开发环境中，Java 程序开发者可以通过改变 JavaBeans 组件的属性表或通过特定的 setXXX()方法、getXXX()方法来定制 JavaBean 组件的外观和运行状态。多个 JavaBeans 组件可以组合在一起构成 Java Applet 或 Java Application、JavaBeans。JavaBeans 组件总是在程序运行时被实例化，它支持可视化及非可视化的组件模型。

EJB 组件模型特点

- EJB 是面向服务端的 JavaBeans 组件模型。它是一种特殊的、非可视化的 JavaBeans，运行在服务器上。EJB 组件同普通的 JavaBeans 组件一样，可以和其他 JavaBeans 组件一起建立新的 Java 应用程序。EJB 对象也可以通过改变它的属性表或者使用它的定制方法来进行处理和定制。
- EJB 组件模型主要包括 EJB Server(EJB 服务器)、EJB Container(EJB 容器)、EJB

Object(EJB 对象)以及诸多相关特性。

- EJB Server(EJB 服务器)提供 EJB 组件的运行环境。它负责管理和协调应用程序资源的分配。CORBA 运行系统、Web Server(如 WebLogic 服务器)、数据库系统(例如 Oracle9i 数据库服务器)都可以作为 EJB Server。EJB Server 必须提供 EJB Container(EJB 容器)。
- EJB Container(EJB 容器)是用于管理 EJB Object(EJB 对象)的设备。它负责 EJB 对象的生命周期的管理，实现 EJB 对象的安全性，协调分布式事务处理，并负责 EJB 对象的上下文切换。EJB Container 还可以管理 EJB 对象的状态。某些情况下，EJB 对象数据是短暂的(例如会话 EJB 对象)，只存在于特定的方法调用过程中，另一些情况下，EJB 对象数据是长久的(例如实体 EJB 对象)，多个访问都要调用此 EJB 对象数据。EJB 容器必须同时支持短暂对象数据及长久对象数据。EJB 对象被赋予 EJB Container。当 EJB 对象被使用时，你可以通过修改其环境属性来定制 EJB 对象的运行状态特性。比如，开发者可以使用 EJB Container 用户接口提供的工具来修改 EJB 对象的事务模式及安全属性。EJB 对象一经使用，EJB Container 负责管理 EJB 对象生命周期、EJB 对象安全特性和协调事务处理。

注意：在这里，EJB Object(EJB 对象)的定义是不严格的，指的是 EJB 组件中实现了商业逻辑的部分，严格的 EJB Object 的定义请参考 2.3.1 小节，在 2.3.1 小节以前，我们在文中出现 EJB Object 或者 EJB 对象的地方，意义均如上所述。EJB 远程对象与 EJB 对象的意义是不同的，EJB 远程接口对象指的是 EJB Remote 接口的实例对象。

- EJB 规范提供了这样的一种机制，你可以通过在运行时设置相应的属性值来定义每一个 EJB 对象的运行状态。其实，这种功能和 JavaBeans 组件对象的相应功能是十分类似的，每一个 EJB 对象都需要提供 Deployment Descriptor，包括 Entity Descriptor 或 Session Descriptor。
- Deployment Descriptor 被用于设置 EJB 对象的运行状态。这些设置值告诉 EJB Container 如何去管理和控制 EJB 对象，它们可以在应用程序组装(开发)或应用程序运行时进行设置。典型地针对 EJB 对象的设置属性包括：生命周期、持久性、事务属性和安全特性。

JavaBeans 组件模型与 EJB 组件模型的比较

在前面我们已经分别提到过 JavaBeans 和 EJB 的异同，但是在这里我们仍然要再次比较 JavaBeans 和 EJB 组件模型的一些特征，JavaBeans 是 Java 的组件模型。在 JavaBeans 规范中定义了事件监听、事件引发和 JavaBeans 属性等特征。EJB 也定义了一个 Java 组件模型，但是 EJB 组件模型和 JavaBeans 组件模型是不同的两个模型。首先，这两个模型的侧重点不同，JavaBeans 组件模型的重点在于允许开发者在 Java 集成开发环境中可视化地操纵 JavaBeans 组件，把若干个 JavaBeans 组件拼装成为一个完整的 Java 应用程序。JavaBeans 规范详细地解释了组件间事件监听者的注册、注销、事件的引发、事件的传递、事件的过滤识别和属性使用、JavaBeans 的定制和持久化的应用编程接口等诸多内容。EJB 组件模型

的侧重点则在于定义了一个可以便携地部署 Java 组件的服务框架模型。因此，其中并没提及事件的传递和处理，因为 EJB 组件通常不发送事件和监听事件的发生。同样也没有提及属性的定制——EJB 对象的属性定制并不是在开发时进行，而是在 EJB 组件运行时刻(实际上在部署组件时）通过一个部署描述符(Deployment Descriptor)来定制的。定制的 EJB 对象的属性一般是：生命周期、持久性、事务属性和安全特性等属性。

不要试图寻找 JavaBeans 和 EJB 组件模型之间的相似性。它们都是 Java 组件模型规范，但是前者说明了在集成开发环境中应用程序如何组装比较高效的问题，而后者则侧重于部署 EJB 组件的服务平台的细节。不要错误地认为 JavaBeans 组件是用于客户端程序的开发(实际上，JavaBeans 可以用于客户端程序的开发，例如 Java Applet 小程序中可以使用 JavaBeans，不过这不是 JavaBeans 组件的主要用途)，JavaBeans 组件主要用于服务端程序的开发(如本书第一章所述)，同样，EJB 组件也是用于服务器端的程序开发。JavaBeans 可作为进行非图形化的服务器端的 Java 应用开发的组件使用。区别是当你使用 JavaBeans 创建服务器端应用程序时，你还得设计整个的服务框架。使用 EJB 组件，那么服务框架是现成的，由 EJB 平台自动支持的，你只需要使用 EJB 的 API 即可。对于复杂的服务器端应用程序，显然使用 EJB 比使用 JavaBeans 更简单。

会话 EJB 和实体 EJB

EJB 技术规范规定有两种类型的 EJB 对象，它们分别是：Session EJB(会话 EJB)和 Entity EJB(实体 EJB)。

Session EJB(会话 EJB)是短暂存在的对象，它同样运行在服务器端，并执行一些应用逻辑处理。它的实例对象由客户端应用程序建立，并仅能够被该应用程序所使用，其保存的数据需要开发者编写程序来管理。我们通常使用 Session EJB 来完成数据库访问或数据计算等工作。Session EJB 支持事务属性，但是，当系统因为某种特殊的不可预见的原因崩溃或者关闭后，Session EJB 的状态、数据不会再被系统恢复。Session EJB 的这种特性十分类似于 Page Scope 类型的 JavaBean 组件对象。

Entity EJB(实体 EJB)是持久运行的 EJB 对象，由某个客户端应用程序创建，但是可以被其他对象或者其他的应用程序调用。与 Session EJB 不同，Entity EJB 必须在建立时制定一个唯一的标识，并提供相应的机制允许客户应用程序根据 Entity EJB 标识来定位 Entity EJB 的实例对象。多个用户可以并发访问 Entity EJB，事务之间的协调工作由 EJB Container 或者 EJB 类自身来完成。读者应当注意，Session EJB 只能够被创建它的应用程序所调用，所以不存在事务协调的问题。Entity EJB 支持事务处理，当系统停机时，也可以恢复停机以前的状态，包括 EJB 对象所数据在内。EJB 规范中定义了两种处理 Entity EJB 的持久性模型，即 Beans Managed Peresistent(Bean 自管理模式，亦即 BMP 模式)及 Container Managed Peresistent(容器管理模式，亦即 CMP 模式)。BMP 模式是由 EJB 对象自己来管理持久性，它需要 EJB 开发者来编写数据库或应用程序的处理逻辑，并加入到 EJB 对象类(EJBObject Class)的 ejbCreate()、ejbRemove()、ejbFind()、ejbLoad()和 ejbStore()等方法中。CMP 模式是将 EJB 持久性管理交给 EJB Container 来完成。开发者一般要在 EJB 对象的 Deployment Descriptor 中的 Container Managed 字段属性中指定 EJB 实例对象的持久性作用域。当使用 CMP 模式时，不需要用户知道 Entity EJB 所存储的数据源，也不需要用户参与复杂、烦琐

的编码工作。

EJB Container 的作用

EJB Container 在 EJB 环境下主要起到如下作用：

- EJB Container 负责提供协调管理、事务处理和 RMI(远程方法调用)等功能。
- EJB Container 负责建立 EJB 对象的运行上下文环境(Context)，负责切换协调不同类型的 EJB 对象。

EJB Container 可以有不同类型，如 DBMS、Web Server 等。如果某个厂商宣布支持 EJB 模型，那么一般是提供不同应用的 EJB Container。例如 Oracle 公司支持 EJB 是通过 Oracle 数据库服务器，BEA 公司支持 EJB，是通过它的 WebLogic Web 服务器。

客户端应用程序通常不和 EJB 直接打交道。它们要通过 EJB Container 提供的 Home Object(Home 接口的实现)访问 EJB 对象 ，该接口提供了 EJB 对象的客户方视角。使用 EJB Container 提供的 Home 接口，允许客户建立或删除 EJB 远程对象。对 Entity EJB 对象来说，Home 接口还提供定位特定的 Entity EJB 实例对象的功能。

当用户请求 EJB 服务时，它首先要通过 JNDI 技术来定位对象的 Home 接口，EJB 对象类(EJB Object Class)及其远程接口(Remote Interface)对 Home 接口来说是透明的。Home 接口必须提供建立 EJB 远程接口对象的方法。一旦客户端程序找到并定位所需的 Home 接口以后，它就可以通过调用 Home 接口中的生成方法(create())建立 EJB 远程接口对象。

EJB 对象(EJB Object)是 EJB Container 提供的 EJB 类的一个运行实例，它用来实现 EJB 的远程调用接口(上文已经提到过，EJB 对象有两类，分别是 Session EJB 和 Entity EJB)。用户总是通过调用 EJB 远程接口方法调用来间接调用 EJB 对象的服务。通常是使用 RMI 技术。用户调用 EJB 对象的服务时，由 EJB Container 接受请求，并将任务交给 EJB 对象。这种机制保证为用户及 EJB 提供透明的状态管理、事务控制及安全性服务。

Home 接口

Home 接口定义了创建、查找、删除 EJB 远程接口对象或者 EJB 服务的方法。客户端应用程序通过使用 JNDI 技术定位 Home 接口的位置，一旦获得 Home 接口的实例，就可以利用它的 create()方法创建 EJB 远程接口的实例对象。进而调用 EJB 对象的方法，实现特定的功能。

Remote 接口

远程调用接口 (或者简称为远程接口，Remote Interface)是对 EJB 对象方法的总封装，在远程调用接口中，声明了 EJB 对象的方法，但是并没有具体实现这些方法。EJB 对象具体实现了这些方法。当我们利用 Hmoe 接口实例对象的方法创建 EJB 对象时，实际上创建的却是 EJB 远程调用接口的实例。当我们调用这些实例的方法时，由于远程接口没有实现这些方法，所以调用的却是 EJB 对象相应的方法。利用远程接口的好处在于：这样做可以把某些 EJB 对象的方法屏蔽起来，不让用户访问。

每一个 EJB 都必须有一个 Remote 接口。Remote 接口定义了应用程序规定客户可以调用的逻辑操作。这是一些可以由客户端程序调用的公共的方法，通常由 EJB 对象类来实现。

注意，EJB 的客户并不直接访问 EJB。而是通过 Remote 接口来访问的。

2.1.3　EJB 技术的未来

Sun 公司公布的 EJB1.0 规范是 EJB 规范的第一个版本，随着 J2EE 技术的发展以及建立企业电子商务应用系统的迫切需求，EJB 技术还会进一步的完善与发展。EJB1.0 只规定必须支持 Session EJB。EJB 2.0 规范中，Entity EJB 也必须支持。目前，很多厂商都宣布支持 EJB1.0 /EJB 1.1 规范。并同时提供了对 Session EJB/Entity EJB 的支持。如 IBM 的 San Francisco Project。BEA 公司更是一马当先，推出的 WebLogic 6.0 Web 服务器系统是世界上第一个宣布支持 EJB 2.0 规范的运行平台，其实这就是一个 EJB Server/Container。目前，进程管理、线程池、并发控制和资源管理等功能未包含在 EJB 1.0 规范中。因此，对这些服务的支持程度，也就决定了不同厂商的 EJB 产品的差别与不同。大多数厂商都是将自己原来的中间层解决方案移植到 EJB Container 中。.EJB 技术的发展方向是要支持多种中间层环境，包括：

CORBA 平台，比较典型的如 Borland 公司的 VisiBroker for Java。

DBMS(数据库管理系统)，如 Informix、Oracle、Sybase 等数据库服务器。

Web 服务器，如 BEA WebLogic、Netscape iPlanet Web Server、Oracle Application Server 等。关于市场上流行的支持 EJB 组件模型的开发工具和服务器的列表，请参考本书附录一。

从企业应用多层结构的角度，EJB 是商业逻辑层的构件技术，与 JavaBean 不同，他提供了事务处理的能力，自从三层结构提出，中间层，也就是商业逻辑层，是处理事务的核心，由于从数据存储层分离，它就取代了存储进程的大部分地位。从分布式计算的角度，EJB 像 CORBA 一样，提供了分布式技术的基础，以及对象之间的通讯手段。从 Internet 技术应用的角度，EJB 和 Servlet，JSP 一起成为新一代应用服务器的技术标准。EJB 中的 Bean 可以分为会话 Bean 和实体 Bean，前者维护会话，后者处理事务。现在，Servlet 程序负责与客户端通信，访问 EJB，并把结果通过 JSP 产生页面传回客户端，成为开发的新潮流。从发展的角度看，EJB 完全有可能成为面向对象数据库的新平台，构成企业计算的基础。

总而言之，在日新月异的技术发展和更新中，EJB，甚至 EJB 技术的后继者，将在 J2EE 技术的大旗下不断攻城略地，占领企业计算的大好江山。在下一节，我们将详细介绍 EJB 组件模型的结构。

2.2　EJB 体系结构(一)

2.2.1　EJB 组件如何工作

通过上面的介绍，我们对 EJB 的组件模型应该有了初步的了解，但这是十分肤浅的，如果想凭这一点知识去开发 EJB 应用系统，那无疑是天方夜谭。下面我们首先简要地介绍一下 EJB 组件模型的运行原理。然后再结合运行原理详细介绍 EJB 组件模型。请看图 2.1。

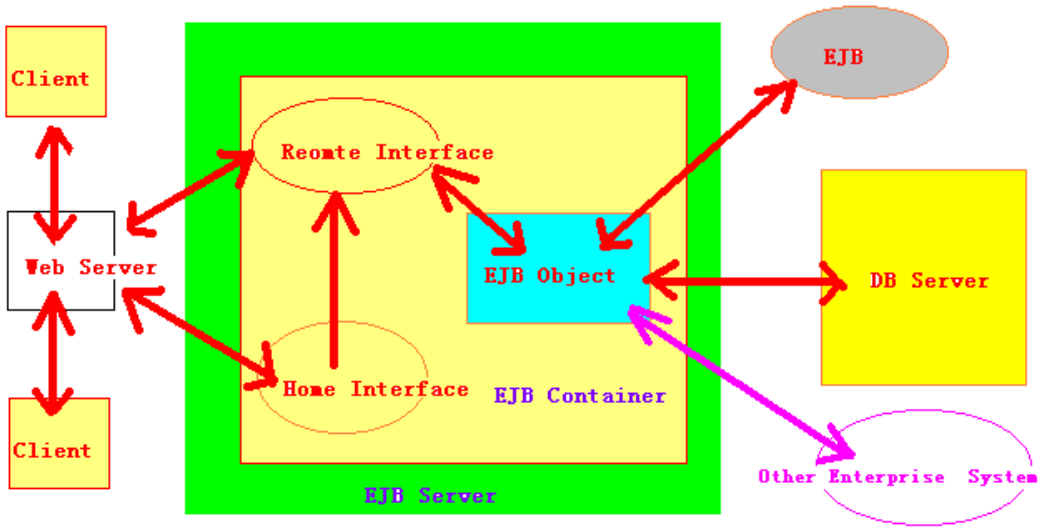


图 2.1　EJB 应用系统运行原理

图 2.1 是一个最最简单的 EJB 应用系统的运行原理图。不会吧，这还是最简单的系统?是的，这确实是最简单的 EJB 应用系统，实际上的 EJB 应用系统要比它还要复杂很多倍。不过读者也不用害怕。EJB 系统刚开始学的时候觉得很复杂，很难，毫无头绪。但是随着学习的深入与开发经验的增加，你就会觉得 EJB 技术原来也是那么简单。

闲话少说，现在我们转入正题。图 2.1 明显地可以分为三个层次，从左到右看。第一层包括 Client 和 Web Server，其实这一层可以再分为两层，分别是 Client 层与 Server 层，但是为了简化系统结构起见，我们还是把它们归为一层。Client 端向 Server 端发送请求，Server 端响应 Client 端的请求，这就是普通的 C/S 模式。Server 可以直接反馈信息到客户端，而不经过任何其他的中间件。但是 Server 端也可以把客户端的请求发送到某个特定的应用系统中，由应用系统对这个请求进行处理，然后再把结果返回 Server 端，Server 端再把结果返回 Client 端，这样做的好处是 Server 端可以专注于响应客户端的请求，而把繁重的计算工作分发到其它的应用系统中进行处理，Server 端只是起了一个信息流交换媒介的作用，这样的处理模式在大流量、重负荷状态下运行具有明显的优越性。实际上，应用系统还可以把客户端的请求按照其性质再次分发到下一级的应用系统中进行处理，这样就可以构造一个高效的树状计算阵列，对输入信息流进行接受、过滤、分发、处理、反馈等操作。EJB 组件就是这种类型的应用系统。

图 2.1 中的第二层是所谓的应用系统层，EJB 服务就主要驻留在这一层。这一层可以是任何可用的中间件解决方案，这与系统平台有很大的关系，但是本章的主题是 EJB 技术，所以我们就假定第二层完全是由 EJB 组件构成的，下面的讨论也都是以这个假定为出发点。第二层由下面的部分组成：EJB Server、EJB Container、Remote Interface、Home Interface、EJB Object 等。应用系统层是如何工作的呢?Web Server 把客户端的请求分发到应用系统层，首要的目标是找到提供特定服务的 EJB 组件，Web Server 透过 EJB Server 层，与 EJB Container 通信，查找并且定位 Home 对象，Home 对象是 EJB 对象与客户端应用程序(这里的客户端是相对而言的，因为第一层相对于第二层就是客户端了，第二层的应用系统相对

于第三层也是客户端了)之间通信的接口，当找到特定的 Home 对象时，我们就可以利用这个对象，创建一个 Remote 对象，这个 Remote 对象封装了 EJB 对象的所有功能，在应用程序中调用 Remote 对象的方法，实际上就是调用 EJB 对象的方法。在这个 EJB 对象的方法中，还可以调用另一个 EJB 对象的方法，第二个 EJB 对象可能存在另一个 EJB Container(同一个 EJB Server)中，甚至还有可能存在于另一个 EJB Server 的 EJB Container 中，不过这两个 EJB 对象是否存在于同一个 EJB Server 中并非问题的关键，关键之处在于第一个 EJB 对象也成了第二个 EJB 对象的客户端程序了，第一个 EJB 对象的还可以调用另外的应用系统来完成特定的任务，例如 CORBA、DBMS 系统，这些另外的应用系统就构成了图 2.1 中的第三层(包含 EJB、Database、Other Enterprise System)。

我们特别需要注意的是：在第一层运行的应用程序通过调用第二层的 Home 对象的 create()方法创建 Remote 对象，并调用 Remote 对象的方法，从应用程序的代码上看，似乎 Remote 对象是驻留在第一层的服务器的内存空间中，但事实上却并非如此，Remote 对象驻留并运行于第二层 EJB Server 的内存空间中，它把执行的结果直接返回第一层的应用程序，不需要再通过 Home 对象了。第一层的应用程序应该分析执行的结果，并把数据送到客户端(这才是真正的客户端，不提供任何服务，只享受服务)的浏览器中。

本章的任务，就是向读者介绍如何使用 EJB 技术构造服务端的第二层——应用系统层。下面我们就来仔细分析一下 EJB 组件模型的具体细节。

2.2.2　EJB Server

EJB 服务器(EJB Server)是管理 EJB 容器(EJB Container)的高端进程或应用程序，并提供对系统服务的访问。EJB 服务器也可以提供厂商自己的特性，如优化的数据库访问接口，对其他服务（如 CORBA 服务）的访问，对 SSL3.0 的支持等。一个 EJB 服务器必须提供对 JNDI 命名和目录服务和事务服务的支持。一些可能的 EJB 服务器的例子如

- 数据库服务器，典型的例子如 Oracle 8i/9i 数据库服务器。
- Web 服务器，典型的例子如 BEA 的 WebLogic Web 服务器，IBM 公司的 WebSphere 服务器。
- 中间件服务器，如 Borland 公司的 Smart Agent。

2.2.3　EJB Container

EJB 容器(EJB Container)是一个管理一个或多个 EJB 类/实例的平台。它通过在规范中定义的接口使 EJB 类访问所需的服务。EJB 容器厂商也可以在容器或服务器中提供额外服务的接口。现在还没有 EJB 服务器和 EJB 容器间接口的规范。因为目前 EJB 容器通常由 EJB 服务器来提供，它们一般都是同一个厂家的产品，所以一旦 EJB 服务器和 EJB 容器之间的接口标准化了，厂商就可能提供可以在任何兼容的 EJB 服务器上运行的 EJB 容器。

Home 接口支持所有定位、创建、删除 EJB 远程对象的方法。Home 对象(Home Object)是 Home 接口的实现。EJB 类开发者必须自己定义 Home 接口。EJB 容器厂商应该实现从 Home 接口中产生 Remote 对象的方法。远程接口（Remote Interface）封装了 EJB 对象中的商业方法。EJB Object 实现了远程接口，并且客户端通过它访问 EJB 实例的商业方法。EJB 类开发者必须自己定义远程接口，EJB 容器开发商必须提供产生相应的 EJB Object 的方法。

客户端程序不能得到 EJB 实例的引用，只能得到它的 EJB Object 实例的引用。当客户端应用程序调用某个方法，EJB Remote 对象接受请求并把它传给 EJB Object，同时提供进程中必要的包装功能。客户端应用程序通过调用 Home 对象的方法来定位、创建、删除 EJB 类的实例(其实是远程接口对象)，通过 Remote 对象来调用 EJB Object 中的商业方法。客户端应用程序可以用 Java 语言来编写，通过 Java RMI 技术来访问 EJB Container 中的 Home 对象和 Remote，或者用其他语言编程并通过 CORBA/IIOP 技术访问已经部署的服务器端组件。

在理解 EJB 规范时，对于 EJB 容器这个术语，我们并不应从字面上简单地把它理解为一个类，而是应该理解为一层代替 EJB Object 访问 EJB Server 并调用相应系统服务的接口。EJB 容器开发商应该提供一套完成这些功能的工具和接口，这些工具和编程接口可以运行在 EJB Server 上。

上面提到的系统服务包括：

- 持久性管理（对于实体 EJB 而言）
- 实现创建和查找服务的 Home 对象的可用性
- 可通过 JNDI 技术访问的 Home 对象对于客户端程序的可视性
- 正确地创建、初始化和删除 EJB Object/Remote 对象
- 保证商业方法能够正确地运行在事务上下文中
- 实现某一基本的安全服务
- 从 Home Interface 和 EJB Object 上产生 stub 类(存根类)和 skeleton 类(框架类)，以实现客户端与服务端的交互。

EJB Object 必须与 EJB 容器通讯来确定调用商业方法的事务上下文。确定以后，EJB Object 在调用商业方法以前建立事务上下文。重要的是 EJB Object 和 EJB 容器的协同工作来实现 EJB 容器所需的事务。EJB 容器厂商提供二者的实现，但对二者的功能分割却是自由的。

目前 EJB 容器厂商都提供专门的 Deploy 工具来读取 Home Interafce，并产生作为 EJB 容器类的 Home Object。在这种情况下 EJB 容器厂商对于每个 EJB Object 类使用不同的 EJB 容器类。EJB 容器厂商还可以使用其它的实现策略，如一个 EJB 容器类实现多个 Home 接口，即产生多个 Home Object。唯一的必要条件是 EJB 容器厂商必须使客户端程序能通过 JNDI 技术访问 Home Object，进而访问 Remote Interface，再由 Remote Interface 访问 EJB Object。客户端应用程序的开发者和 EJB 组件开发者都不需关心 EJB 容器和 Home Object 的实现细节。

2.2.4 Home Interface

客户端的应用程序调用 EJB 组件的方法时，必须首先通过调用 EJB 组件的 Home 接口(Home Interface)的方法创建它(指的是 EJB 组件)的远程实例(Remote Object)。Home 接口包含了一个或多个用来创建 EJB 远程实例的 create()方法。这个 Home 接口的实现不是由 Bean 来实现，而是通过一些称为 Home Object 的类来实现的，这些所谓的 Home Object 类一般不是由开发者编写的，而是由开发工具自动生成的(EJB 容器生产厂家提供这些开发工具)，当然了，如果你希望自己编写 Home Object 类，这没有什么特别的问题，不过你需要透彻

了解不同的 EJB Container 是如何实现 Home 接口的，不同的 EJB Container 的实现方法往往大相径庭，到了下面我们还会提到这个问题，所以我们建议读者使用专门的工具自动产生 Home Object 类，而不是自己编写它们。

Home Object 类的实例在 EJB Server 中实例化，形成 EJB Container，使得客户端的应用程序可以访问并且定位它们。

一个 Home Object 的引用被存放在 JNDI 服务中，客户端的应用程序能够通过 JNDI 技术访问它。EJB Server 一般需要提供某种名字空间的实现，或者是使用外部的名字空间。在这两种情况下客户端应用程序都必须知道名字空间的位置以及 JNDI 的上下文类。例如，一个客户端的 JSP 程序可能接收名字空间和 JNDI 上下文类作为 JSP 程序的参数。除了提供名字空间的位置和 JNDI 上下文类名(Context)，客户端应用程序也必须知道如何在名字空间树状结构中定位 Home Object。这些必须在客户端程序启动时提供。当 EJB 部署者把 EJB 组件发布到 EJB 服务器中时，他必须以参数形式指定名字树，例如 ejb/Test，客户端程序必须获得这个完整的路径名来定位并获得 Test Home Object 的引用。这不是说客户端应用程序通过 JNDI 获得 EJB 容器，而是客户端应用程序使用 JNDI 技术查找(lookup)Home 接口的实现。Home 接口的实现由某个特殊的 EJB Container 来提供(通过 Home Object)，但这是 EJB 容器厂商的细节，EJB 组件开发者和客户端程序的开发者可以忽略它并且应该忽略它。

EJB 开发者在定义 EJB Object 中的 ejbCreate()方法的同时必须在 Home 接口中声明与其相应的 create()方法。实体 EJB 可以包含 Finder 方法以使得客户端程序能够定位已有的实体 EJB 对象。这是因为会话 EJB 只能够被创建它的客户端使用，而且一旦客户端对其的调用结束，那么会话 EJB 会被 EJB Server 自动注销，所以会话 EJB 不支持 finder()方法。实体 EJB 就不同了，一旦被创建，就一直存在于 EJB Server 中，它可以被多个客户端程序同时使用，所以才支持 finder()方法。我们在编写 Home 接口的时候，必须注意到 Home 接口是通过继承 javax.ejb.EJBHome 接口来定义的。EJBHome 接口包含如下的方法：

public void remove(Handle handle) throws java.rmi.RemoteException，RemoveException;

public void remove(java.lang.Object primaryKey) throws java.rmi.RemoteException，RemoveException;

public EJBMetaData getEJBMetaData() throws java.rmi.RemoteException;

public HomeHandle getHomeHandle() throws java.rmi.RemoteException;

一个 EJB 的 Home 接口可以像下面的样子：

程序清单 2.1

```
//File Name:TestHome.java
//Date:2001.4.30
//Author:fancy
//Note:create a Home Interface
import java.rmi.*;
import javax.ejb.*;
public interface TestHome extends EJBHome
{
    public TestRemote create() throws RemoteException，CreateException;
}
```

程序清单 2.1 是一个最简单的 Home Interface。一个 Home Interface 必须扩展 EJBHome 接口，而且必须声明 create()方法，这个 create()方法的返回类型必须是 Remote Interface 的实例，create()方法需要抛出 RemoteException 和 CreateException 这两个异常，否则编译不会通过。

程序还需要导入 java.rmi 包与 javax.ejb 包的支持。

在本小节结束之时，我们再次强调一遍：EJB 容器开发商负责提供实现 Home 接口的 Home Object 类或者提供实现的方法与工具，因为只有 EJB 容器开发商才能实现存储 EJB 的库的编码。

2.2.5 Remote Interface

EJB 开发者已经创建了一个客户端应用程序可以访问 create()方法的 Home Interface，在 Home Interface 中定义的 create()方法在相应的 Bean 中都必须有一个 ejbCreate()方法与之对应。同样，EJB 开发者必须创建封装了客户端应用程序能够访问的商业方法的 Remote Interface。

因为所有的客户端程序调用的方法都必须通过 Remote Interface，因此实现这个接口的是 EJB Object 而不是 Home Object。Remote Interface 中列出的方法名必须和实现 Bean 的方法名相同。

我们在编写 Remote Interafce 的时候，必须注意 Remote Interface 必须扩展 javax.ejb.EJBObject 接口。EJBObject 接口定义的方法如下所示:

```
public EJBHome getEJBHome() throws java.rmi.RemoteException;
public java.lang.Object getPrimaryKey() throws java.rmi.RemoteException;
public void remove() throws java.rmi.RemoteException，RemoveException;
public Handle getHandle() throws java.rmi.RemoteException;
public boolean isIdentical(EJBObject obj) throws java.rmi.RemoteException;
```

以下是一个 Remote Interface 的例子。

程序清单 2.2

```
//File Name:TestRemote.java
//Author:fancy
//Date:2001.4.30
//Note:create a Remote Interface。
import java.rmi.*;
import javax.ejb.*;
public interface TestRemote extends EJBObject
{
    public java.lang.String getAuthor() throws RemoteException;
}
```

所有在 Remote Interface 中声明的方法都必须抛出一个 RemoteException 异常，因为 EJB 规范要求客户端的 stub 类必须是 Java RMI 技术兼容的。但这并不意味着排除了用其它的传输方式的 stub/skeleton 实现，如 CORBA/IIOP。

Remote Interface 继承 javax.ejb.EJBObject 接口，又声明了额外的方法，在上面的例子

中，我们声明了 getAuthor()方法。客户端的应用程序就可以调用这个方法。

2.3　EJB 体系结构(二)

2.3.1　EJB Object

EJB Object(EJB 对象)是网络上的可视对象(对于 Remote Interface 而言是可视的)，它包含了 stub 和 skeleton，作为 EJB 类的代理。EJB 组件的 Remote Interface 继承了 javax.ejb.EJBObject 接口，而 EJB Object 则实现了这个 Remote Interface，使得 EJB 类与 Remote Interface 之间搭起了一座桥梁。对于每个 EJB 都有一个定制的 EJB Object，EJB Object 隔离了 EJB 类，使得 EJB 类在网络中是不可见的，不过 EJB Object 提供了 stub 和 skeleton，Remote Interface 通过这两条路径，与 EJB 类相互通信，所谓的 EJB 类就是真正实现了商业逻辑、商业方法的类，它有两种类型，一种是 Session EJB(会话 EJB)，一种是 Entity EJB(实体 EJB)。在前面，我们对 EJB Object 与 EJB 类不加区分，统一称为 EJB Object 或者 EJB 对象，但是由此开始，我们就需要把它们严格区分开来，打一个不十分贴切的比方：EJB 类相当于一个人，那么 EJB Object 就相当于人身上的衣服。

实现 Remote 接口的 EJB Object 其实是一个 RMI 服务器对象，注意 EJB 类本身不是一个 Remote Object，它在网络上是不可视的。当 EJB 容器实例化了这个 EJB Object 后，EJB 容器会接着初始化 EJB 类的实例引用，使得 EJB Object 能够处理 Remote Interface 传送过来的请求，代理 EJB 类所实现的商业方法的调用。

EJB Object 是由提供 EJB 容器、EJB Server 的厂商实现的，厂商的实现维护了 EJB Object 实例和 EJB 实例的一对一关系。因为 Remote Interface 包含了 EJBObject 接口的方法，所以 EJB 类不必显式地实现这个接口，虽然它提供了 Remote Interface 所列出的商业方法的实现。因为 EJB Object 必须正式地实现 EJB 的 Remote Interface，EJB 容器在部署 EJB 时会自动产生 EJB Object 的源代码，并编译为二进制文件，这些产生的 EJB Object 源代码实现了 Remote Interface。典型的 EJB Object 有一个独特的类名，作为 EJB Object 和 EJB 类的联系。

2.3.2　Session EJB

EnterpriseBean 是 EJB 开发者编写的提供应用程序功能的类，简称为 EJB 类，在上文中我们已经有所提及，EnterpriseBean 是 EJB 组件模型中具体实现商业逻辑、商业方法的部分。EnterpriseBean 有两种，会话 EJB 与实体 EJB。EJB 开发者可以选择创建会话 EJB 或实体 EJB，通过实现不同的接口声明，不同的部署描述符来加以区分。

对于会话 EJB，必须实现 javax.ejb.SessionBean 接口，对于实体 EJB，必须实现 javax.ejb.EntityBean。客户端应用程序不会也不能够直接访问 EnterpriseBean 中的任何方法。客户端应用程序通过 Remote Interface/EJB Object 间接调用 EnterpriseBean 中的方法，EJB Object 就像一个代理服务器一样。在把调用通过 EJB Object 传递时，容器开发商通过包装 EJB Object 的编码插入自己的功能，这称为方法插入。方法插入的一个例子是为每个方法调用创建一个新的事务上下文，当方法返回到 EJB Object 时提交(commit)或回滚(rollback)事务。

当使用 EJB 容器厂商的发布工具在分发部署 EJB 时，发布工具会产生 EJB 组件的 EJB Object 的一个 stub 和 skeleton。实际上发布工具并不创建 EnterpriseBean 本身的 stub 和 skeleton，也并不需要创建，因为 EnterpriseBean 不会通过网络直接被访问到。EJB Object 才是真正的网络对象。

EJB 容器也可以调用 EnterpriseBean 中的某个方法。例如，EJB 容器保证当一个 EnterpriseBean 实例生成后，Home Object 中的 create()的任何参数会传递到 EnterpriseBean 相应的 ejbCreate()方法。EnterpriseBean 还有其它的接口和要求。然而，会话 EJB 和实体 EJB 的要求是不同的。下面我们就来介绍会话 EJB(Session EJB)，下一小节再介绍实体 EJB(Entity EJB)。

会话 EJB 是一种通过 Home Interface 创建并对客户端连接专有的 EnterpriseBean。会话 EJB 的实例一般不与其它客户端共享。这允许会话 EJB 维护客户端的状态。会话 EJB 的一个例子是购物车，众多顾客可以同时购物，往他们自己的购物车中添加商品，而不是向一个公共的购货车中加私人的货物。

我们可以通过定义一个实现 javax.ejb.SessionBean 接口的类来创建一个会话 EJB。该接口(SessionBean 接口)的定义如下：

public void setSessionContext(SessionContext ctx) throws EJBException , java.rmi.RemoteException;

public void ejbRemove() throws EJBException，java.rmi.RemoteException;

public void ejbActivate() throws EJBException，java.rmi.RemoteException;

public void ejbPassivate() throws EJBException，java.rmi.RemoteException;

javax.ejb.EnterpriseBean 是一个空接口，是会话 EJB 和实体 EJB 的超类。下面是一个简单的会话 EJB 的例子。

程序清单 2.3

```
//File Name:TestBean.java
//Author:fancy
//Date:2001.5.1
//Note:to create a session EJB
import java.rmi.*;
import javax.ejb.*;
public class TestBean implements SessionBean
{
    private SessionContext sessionContext;
    public void ejbCreate()
    {
    }
    public void ejbRemove()
    {
    }
    public void ejbActivate()
    {
```

```
    }
    public void ejbPassivate()
    {
    }
    public void setSessionContext(SessionContext context)
    {
        sessionContext = context;
    }
    public String getAuthor()
    {
        return "Rainbow";
    }
}
```

在上面的例子中，geAuthor()方法是 EJB 开发者自己定义的商业逻辑。

会话 EJB 的交换(切换)

EJB 容器开发商可以实现把会话 EJB 的实例从主存移到二级存储中的交换机制，这可以增加在特定的一段时间内实例化的会话 EJB 的总数。EJB 容器维护一个会话 EJB 实例的时间期限，当某个会话 EJB 的不活动状态时间达到这个阈值，EJB 容器就把这个会话 EJB 实例拷贝到二级存储中并从主存中删除。EJB 容器可以使用任何机制来实现会话 EJB 的持久性存储。最常用的方式是通过 EJB 的串行化。这个交换机制与 Windows 操作系统的内存交换机制有一定的相似之处。

活化和钝化

为了支持 EJB 容器厂商提供会话 EJB 的交换，EJB 规范定义了钝化——把会话 EJB 从主存转移到二级存储的过程，活化——把会话 EJB 恢复到主存中去的过程。在 javax.ejb.SessionBean 接口中声明的 ejbActivate()和 ejbPassivate()方法，允许 EJB 容器通知已经被活化的会话 EJB 的实例：它将要被 EJB 容器钝化。EJB 开发者可以用这些方法释放或者恢复处于钝化状态的会话 EJB 所占有的值、引用和系统资源。一个可能的例子是数据库连接，作为有限的系统资源，不能被处于钝化状态的会话 EJB 所使用。

有了这些方法就使得不必再使用 transient。事实上，使用 transient 可能是不安全的，因为串行化机制自动地把 transient 的值设为 null 或 0。而通过 ejbActivate()和 ejbPassivate()方法显式地设置这些参数更好一些。依靠 Java 的串性化机制把 transient 的值设成 null 也是不可移植的，因为会话 EJB 有可能被部署在不使用 Java 的串性化机制获得持久性的 EJB 容器中，这时该行为会发生改变。如果 EJB 容器不提供实现 EJB 交换的机制，那么这些方法将永远不会被调用。

当客户端程序调用 EJB 的商业方法时，钝化的会话 EJB 被自动激活。当 EJB Object 收到方法调用的请求时，它唤醒 EJB 容器所需要活化的会话 EJB。当活化完成时，EJB Object 代理对会话 EJB 的方法调用。如果会话 EJB 参与一个事务，那么它不能被钝化。把会话 EJB 放在主存中更有效率，因为事务通常在很短的时间内完成。如果会话 EJB 没有钝化前

必须释放或活化前必须重置的状态，那么这些方法(指的是 ejbActivate()和 ejbPassivate()方法)可置空。在大多数情况下，EJB 开发者不必在这些方法中做任何事。

会话 EJB 的状态管理

会话 EJB 的部署描述符必须声明该 EJB 是有状态或者是无状态的。一个无状态(Stateless)的会话 EJB 是在方法调用间不维护任何状态信息的 EJB。通常，会话 EJB 的优点是代替客户端维护状态。然而，让会话 EJB 无状态也有一个好处。无状态 EJB 不能被钝化。因为它不维护状态，所以没有需要保存的信息。EJB 容器可以删除会话 EJB 的实例。客户端应用程序永远不会知道无状态 EJB 的删除过程。客户端的引用是通过 EJB Object。如果客户端应用程序稍后又调用了一个商业方法，则 EJB Object 通知 EJB 容器再实例化一个新的会话 EJB 实例。因为没有状态，所以也没有信息需要恢复。

无状态 EJB 可以在不同的客户端应用程序间共享，只是在某一时刻只能有一个客户端执行一个方法(独占状态)。因为在方法调用间没有需要维护的状态，所以客户端应用程序可使用任何无状态会话 EJB 的实例。这使得 EJB 容器可以维护一个较小的可用 EJB 的缓冲池，以节省主存。因为无状态 EJB 在方法调用间不能维护状态，因此从技术上讲在 Home Interface 的 create()方法不应有参数。在创建 EJB 时向 EJB 传递参数意味着在 ejbCreate()返回时需要维护 EJB 的状态。而且，EJB 实例一旦删除，经由 EJB Object 调用商业方法的结果使得 EJB 容器必须重新创建一个无状态的 EJB，这时在最开始创建 EJB 时所用的参数就不存在了。EJB 容器开发厂商的部署分发工具应该能检查 Home Interface，如果是无状态对话的 EJB，应该保证其不包含带参数的 create()方法。

2.3.3　Entity EJB

实体 EJB 的角色

实体 EJB 用来代表底层的对象。最常见的是用实体 EJB 代表关系型数据库中的数据。一个简单的实体 EJB 可以定义成代表数据库表的一个记录，也就是每一个实例代表一个特殊的记录。更复杂的实体 EJB 可以代表数据库表之间的关联视图。在实体 EJB 中还可以考虑包含厂商的增强功能，如对象-关系映射的集成。

通常用实体类代表一个数据库表比代表多个相关联的表更简单且更有效。反过来可以轻易地向实体类的定义中增加关联，这样可以最大地复用 cache 并减小旧数据的表现。

实体 EJB 和会话 EJB 的比较

看起来会话 EJB 好象没什么用处，尤其对于数据驱动的应用程序。当然事实并不是这样。因为实体 EJB（譬如说）代表底层数据库的一行纪录，则实体 EJB 实例和数据库记录间就是一对一的关系。因为多个客户端程序必须访问底层记录，这意味着，不同于会话 EJB，客户端程序必须共享实体 EJB(事实上，这个功能也可以使用会话 EJB 来完成，不过效率不高，消耗系统资源过多，有多少个客户端程序在运行，就需要多少个会话 EJB 在服务端运行，而使用实体 EJB 就好多了，只需要一个实体 EJB 在运行即可)。因为实体 EJB 是共享的，所以实体 EJB 不允许保存每个客户端的信息。会话 EJB 允许保存客户端的状态信息，客户端应用程序和会话 EJB 实例间是一对一的。实体 EJB 允许保存记录的信息，实体 EJB

的实例和记录间是一对一的。一个理想的情况是客户端通过会话 EJB 连接服务器，然后会话 EJB 通过实体 EJB 访问数据库。这使得既可以保存客户端的信息又可以保存数据库记录的信息。

没有会话 EJB，应用程序开发者（客户端开发者）就必须理解 EJB 类的事务要求，并使用客户端的事务划分来提供事务控制。EJB 的主要好处就是应用开发者不需知道 EJB 类的事务需求。一个会话 EJB 可以代表一个商业操作，进行事务控制，不需要在客户端程序中进行事务划分。

EntityBean 接口

实体 Bean 类必须继承 javax.ejb.EntityBean 接口，下面我们应该来看看 EntityBean 接口所定义的方法。

public void setEntityContext(EntityContext ctx) throws EJBException，java.rmi.RemoteException

public void unsetEntityContext() throws EJBException，java.rmi.RemoteException

public void ejbRemove() throws RemoveException，EJBException，java.rmi.RemoteException

public void ejbActivate() throws EJBException，java.rmi.RemoteException

public void ejbPassivate() throws EJBException，java.rmi.RemoteException

public void ejbLoad() throws EJBException，java.rmi.RemoteException

public void ejbStore() throws EJBException，java.rmi.RemoteException

Finder 方法

通过 Home Interface 或者 Remote Interface 创建和删除 EJB 的实例，对实体 EJB 和会话 EJB 来说有不同的含义。对会话 EJB 来说，删除意味着从 EJB 容器中删除，不能够再次使用，并且其状态信息也丢失了(也许没有状态信息)。对于实体 EJB，删除意味着底层数据库记录被删除了。因此，一般不把删除作为实体 EJB 生命周期的一部分。

创建一个实体 EJB 意味着一个记录被插进数据库中。与删除操作类似，创建操作通常也不作为实体 EJB 生命周期的一部分。客户端访问实体 EJB 需要先找到它。除了 create()方法以外，一个实体 EJB 的 Home Interface 通常还有 Finder 方法(指 findByPrimaryKey()方法)。客户端需要根据应用程序的限制来识别一个特殊的数据库记录，亦即识别一个实体 EJB，例如程序清单 2.4。

程序清单 2.4

```
//File Name:TestEntityEJBHome.java
//Author:fancy
//Date:2001.5.1
//Note:create a EntityBean'Home Interface
import java.rmi.*;
import javax.ejb.*;
public interface TestEntityEJBHome extends EJBHome
{
```

```
    public TestEntityEJBRemote create() throws RemoteException，
        CreateException;
    public TestEntityEJBRemote findByPrimaryKey(String primKey)
        throws RemoteException，FinderException;
}
```

当客户端程序调用 Remote Object 的任何方法时，EJB 容器会把调用传递到实体 EJB 的相应方法中。一个简单的 Entity EJB 代码实例如程序清单 2.5。

程序清单 2.5

```
//File Name:TestEntityEJB.java
//Author:fancy
//Date:2001.5.1
//Note:create a EntityBean Class
import java.rmi.*;
import javax.ejb.*;
public class TestEntityEJB implements EntityBean
{
    EntityContext entityContext;
    public String ejbCreate() throws CreateException
{
        /**@todo: Implement this method*/
        return null;
    }
    public void ejbPostCreate() throws CreateException
    {
    }
    public void ejbRemove() throws RemoveException
    {
    }
    public String ejbFindByPrimaryKey(String primKey)
    {
        //@todo: Implement this method
        return null;
    }
    public void ejbActivate()
    {
    }
    public void ejbPassivate()
    {
    }
    public void ejbLoad()
    {
    }
    public void ejbStore()
```

```
    {
    }
    public void setEntityContext(EntityContext entityContext)
    {
       this.entityContext = entityContext;
    }
    public void unsetEntityContext()
    {
       entityContext = null;
    }
    public String getAuthor()
    {
       return "fancy";
    }
}
```

一个较好的方法是把 Finder 方法当成数据库的 SELECT 语句，而动态 SQL 参数相当于方法的参数。请注意 Home Interface 中的 Finder 方法向客户端返回一个 EJB 的远程接口对象(请参考程序清单 2.4)。而实体 EJB 中的 Finder 方法向 EJB 容器返回一个唯一的标识符(请参考程序清单 2.5)，称为主键(Primary Key)。EJB 容器用这个主键实例化一个选定的 EJB Object。不论如何实现 Finder 方法，EJB 容器都用这个主键代表这个选定的记录(亦即选定的实体 EJB)，由实体 EJB 类来决定如何用这唯一的标识符来代表记录。

可能由一个 Finder 方法得到满足 SELECT 语句条件的多个记录。这种情况下实体 EJB 的 Finder 方法将返回一个主键的枚举类型，Home Interface 的 Finder 方法返回值就是 EJB 远程接口对象的枚举类型。

主键

主键这个词有可能被曲解，把它理解为唯一的标识符更恰当些。当实体 EJB 代表一个数据库记录时，主键可能是该记录的组合键。对于每个实体 EJB 的实例，都有一个相应的 EJB Object，当一个 EJB Object 与一个实体 EJB 实例对应时，该 EJB 实例的主键保存在 EJB Object 中。这时就说该实体 EJB 的实例有一个标识符。当客户端程序调用 Home Object 的 Finder 方法时，EJB 容器会用没有标识符的实体 EJB 的实例来执行这个请求。EJB 容器可能为此维持一个甚至多个匿名的实例。不论如何实现 Finder 方法，都必须向 EJB 容器返回底层数据的主键，如数据库的记录。如果多个记录满足条件，那么就返回多个主键。当 EJB 容器得到主键后，它会用该主键初始化一个 EJB Object。EJB 容器也可以初始化一个与 EJB Object 关联的实体 EJB 的实例，因为底层记录的标识符在 EJB Object 中保存着，因此在 EJB 实例中没有状态。因此，EJB 容器可以在 EJB Object 上调用商业方法时再实例化 EJB，以节省内存资源。

当 Finder 方法向 EJB 容器返回主键时，EJB 容器首先会检查该主键的 EJB Object 是否已经存在。如果该主键的 EJB Object 已经存在，那么 EJB 容器不会创建一个新的 EJB Object，而是向客户端返回这个已存在的 EJB Object 的远程接口对象。这样就保证了每个记录只有

一个 EJB Object 的实例，所有的客户端程序共享同一个 EJB Object。

主键只是在 EJB 类中唯一地标识实体 EJB 的实例，EJB 容器负责保证其范围。应该明确 Finder 方法只是从数据库中取出数据的主键，而不包含其它的数据项。也可能调用 Finder 方法后不产生任何实体 EJB 的实例，只产生包含该主键的 EJB Object，当客户端调用 EJB Object 的方法时再产生并导入实体 EJB 的实例。

客户端能在任何时候获得实体 EJB 的主键，并且以后可以使用该主键通过 Home Interface 重建对实体 EJB 远程接口的引用。主键类的类型在部署描述符中指定。EJB 开发者可以用任何类型来表示主键。唯一的要求是类必须实现 serializable(串行化)，因为该主键可能在客户和服务器之间进行传递。

实体 EJB 的内外存交换

实体 EJB 也有所谓的活化和钝化过程，与会话 EJB 类似。然而，不在事务过程中的实体 EJB 是无状态的;也可以说其状态总是和底层的数据同步的。如果我们像钝化会话 EJB 那样钝化实体 EJB，则当钝化无状态实体 EJB 时只会删除它。但是因为 EJB 容器调用 Finder 方法需要匿名的实体 EJB，EJB 容器可能为此把不活动的实体 EJB 钝化到一个私有内存池中。一旦从 EJB Object 中删除了实体 EJB，则同时也删除了标识符（主键关联）。

当客户端程序调用没有相关的实体 EJB 的 EJB Object 的商业方法时，EJB 容器就可能用这个私有内存池重新分配实体 EJB。注意这个内存池中的 EJB 没有标识，可以被任何 EJB Object 重用。EJB 容器可以不维护任何有 EJB Object 的实体 EJB，除非有一个商业方法在通过 EJB Object 被调用。如果实体 EJB 在事务中运行则需保持其与 EJB Object 的关联。

自管理的持久性

实体 EJB 的管理有两种模式，一种是自管理模式，另一种是容器管理模式。因为实体 EJB 代表底层的数据，所以我们需要把数据从数据库中取出然后放在 EJB 中。当 EJB 容器第一次把一个实体 EJB 的实例与 EJB Object 关联时，它就开始了一个事务并调用了这个 EJB 的 ejbLoad()方法。在这个方法中开发者必须实现从数据库中取出正确的数据并把它放在 Bean 中。当 EJB 容器要提交一个事务时，它首先调用 EJB 的 ejbStrore()方法。这个方法负责向数据库中回写数据。我们称之为自管理持久性，因为实体 EJB 类方法中的代码提供了这种同步。

当 ejbLoad()方法完成时，EJB 有可能与底层数据库不一致。商业方法的调用触发了与 EJB Object 关联的 EJB 实例的分配，在事务中执行的 ejbLoad()方法必须在部署描述符中声明。根据接收到的方法调用请求，EJB Object 和 EJB 容器一起建立一个事务上下文，EJB 容器分配与 EJB Object 关联的 EJB 实例并调用 EJB 实例的 ejbLoad()方法，这个方法现在运行在事务上下文中。这个事务上下文传递给数据库，根据部署描述符中指定的孤立性级别，这个事务锁定数据库中被访问的数据。

只要事务上下文活动，数据库中的数据就一直保持锁定状态。当客户端或 EJB 容器提交事务时，EJB 容器首先调用 bean 的 ejbStore()方法，把 EJB 中的数据回写到数据库中。相应的数据库记录在 ejbLoad()和 ejbStore()间保持锁定，这个模式保证了 EJB 和数据库间的同步。其间可以进行不同的商业方法调用。而且，ejbLoad()和 ejbStore()明确地区分了事务

边界，事务中可以进行任何商业方法调用。事务的持续时间由部署描述符决定，也可能由客户端决定。

容器管理的持久性

如果部署描述符声明实体 EJB 使用容器管理持久性，则我们不需要自己调用 ejbLoad()和 ejbStore()来访问数据库。EJB 容器会自动把数据从数据库中导入到 EJB 中，然后调用 EJB 的 ejbLoad()方法完成从数据库中接收数据。

同样地，EJB 容器调用实体 EJB 的 ejbStore()方法来完成把数据回写到数据库中。这些方法实际上没有执行任何数据库操作。当 EJB 容器开发商用复杂的工具来提供容器管理持久性时，如自动产生能进行对象-关系映射的实体 EJB 类，EJB2.0 规范规定了厂商必须提供的容器管理实体持久性的最小需求集。

2.3.4　部署描述符

部署描述符可以指定 EJB 的一个 public 域来实现与数据库列简单映射。EJB 容器使用 EJB 部署描述符读出这个 EJB 的 public 域并写到相应的列，或从数据库列中读出数据写到 public 域中。EJB 容器管理的持久性对 EJB 开发者来说是非常好的服务，且不需对象-关系影射等其他复杂的机制，开发者会发现它比 EJB 自管理的持久性更有效率。

EJB 开发中两个主要的角色是 EJB 开发者和 EJB 部署者。EJB 有很多属性为开发者所不能预知，如数据库的网络地址、使用的数据库驱动程序等等。部署描述符作为由 EJB 开发者定义的特性表，由 EJB 部署者添入正确的值。EJB 部署描述符有标准的格式(一般是一个 XML 文件)，在开发和部署环境中是可移植的，甚至在不同 EJB 平台间也是可移植的。

除了为开发 EJB 和部署 EJB 的协同提供一个标准的属性单，部署描述符也应包含 EJB 应如何执行有关事务和安全的细节信息。一些如访问控制链（ACL）等属性，也应该由 EJB 部署者来调整，以保证适当的用户能在运行时使用 EJB。其它属性，如事务信息，有可能完全由 EJB 开发者事先指定，因为一般由 EJB 开发者创建数据库访问代码，并熟知 EJB 的方法应如何运行事务。部署描述符是一个标准的 Java 类。我们需要创建一个部署描述符的实例，导入数据，然后串行化。这个串行化的部署描述符放在一个 jar 文件中并和 EJB 的其他部分一起送到部署环境。EJB 部署者利用某种部署工具读取这个串行化的部署描述符，可能修改一些属性值，然后使用这个修改后的部署描述符来分发部署 EJB。

部署一个 EJB 时，分配对应的部署描述符，然后初始化、串行化，再将其与其它 EJB 类一起放入到一个 jar 文件中。不同厂商在编辑部署描述符时可能有不同的方式。例如，一个厂商可能使用文本方式，而另一厂商可能提供图形工具。但最后结果的部署描述符是一个标准的格式，并且在不同平台间是可移植的。

为了包装一个 EJB，该 EJB 的类，接口和串行化的部署描述符放在一个 jar 文件中。这个 jar 文件必须含有一个 manifest 文件，manifest 文件提供 jar 文件中所有文件的完整描述。例如：

```
Manifest-Version: 1.0
Name: testbean/_TestRemote_Stub.class
Name: testbean/TestHomeHelper.class
```

Name: testbean/TestHomeOperations.class
Name: testbean/_TestHome_Stub.class
Name: testbean/TestHomeHolder.class
Name: testbean/TestRemotePOA.class
Name: testbean/TestRemote.class
Name: META-INF/ejb-jar.xml
Name: testbean/TestRemoteOperations.class
Name: META-INF/ejb-inprise.xml
Name: testbean/TestHomePOAInvokeHandler.class
Name: testbean/TestHome.class
Name: testbean/TestRemoteHelper.class
Name: testbean/TestRemotePOAInvokeHandler.class
Name: testbean/TestBean.class
Name: testbean/TestRemoteHolder.class
Name: testbean/TestHomePOA.class

开发者不必关心 EJB jar 文件的创建。EJB 容器的开发商一般会提供一个工具来帮助开发者创建部署描述符，然后把所有必需的文件打包进一个 jar 文件中。

2.4　如何开发 EJB(一)

2.4.1　EJB 开发工具简介

由本小节开始，我们将介绍如何开发 EJB，如何把 EJB 技术与 JSP 技术结合起来。首先，我们需要了解一些关于 EJB 开发工具的知识。

目前，市场上流行的 EJB Server/Container 有 BEA 公司的 WebLogic、IBM 公司的 WebSphere、Borland 公司的 Inprise Application Server、Allaire 公司的 JRun、Oracle 公司的 Oracle 8i/9i，前面 4 个都是 Web 服务器，Oracle8i/9i 则是数据库服务器，它们都支持 EJB1.0/1.1 技术规范，其中 WebLogic 6.0 服务器支持 EJB 2.0 规范。在这几种 EJB Server/Container 中，我们建议读者选用 WebLogic 作为 Web 服务器端的 EJB Server，以 Oracle 8i/9i 作为数据库服务端的 EJB Server。

市场上流行的 EJB 开发工具有 Borland 公司的 JBuilder 3.5/4.0、IBM 公司的 VisualAge 3.0/3.5、Oracle 公司的 JDeveloper 3.2、Allaire 公司的 JRun Studio、Sun 公司的 Forte 2.0、Sybase 公司的 PowerJ。这里没有什么好与坏之分，我们建议读者选用的开发工具最好和选用的 EJB Server 相匹配，例如如果你选用了 Borland 公司的 IAS 服务器做为 EJB Server，那么你最好选用 Borland JBuilder 4.0 作为 EJB 的开发工具，如果你选用 IBM 公司的 WebSphere 作为 EJB Server，那么你最好选用 VisualAge 3.0/3.5 作为 EJB 开发工具，当然了，也不是非得如此不可，只是这样做至少不会产生 EJB 与 Container/Server 不相匹配的问题。

在本书中，我们选用 Borland 公司的 JBuilder 4.0 Enterprise Edition 作为 EJB 的开发工具，以 Borland 公司的 Inprise Application Server 4.1.1(简称为 IAS)服务器的 Smart Agent 作为测试 EJB 性能的临时 EJB Server，以 BEA 公司的 WebLogic5.1 服务器作为最终分发部署

EJB 组件的 EJB Server/Container，我们还选用 WebLogic 5.1 作为 JSP 服务器，利用它来测试 EJB 的客户端(指基于 Web 模式的客户端，基于 Application 模式的客户端程序在 JBuilder4 中直接测试)。

2.4.2　JBuilder 4.0+IAS 4.1 的开发环境配置

在本节中，我们将介绍如何配置 JBuilder 4.0 的开发环境，使得它可以开发 EJB 组件。

首先，你必须安装 JBuilder 4.0 Foundation Edition，然后再安装 JBuilder 4.0 Enterprise Edition，这没有什么好说的。

启动 JBuilder4，依次选择 File→New…→Enterprise Tab，我们会看到与 EJB、CORBA 工程有关的图标都处于不可用的状态，这说明暂时我们还无法创建一个 EJB 工程。

安装 IAS 4.11 服务器，可以安装在任意目下面，我们建议读者把 IAS 安装到 JBuilder4 的程序目录下面。

安装 IAS4.11 完毕后，我们在 JBuilder4 中，选择 Tools→Configure　Libraries…，出现如图 2.1 所示对话框。

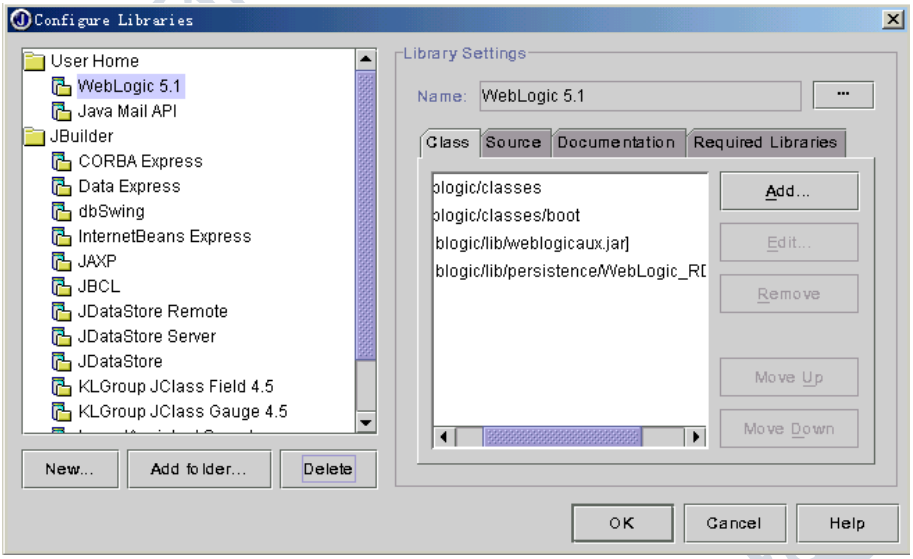


图 2.1　JBuilder4 的 Configure Libraries 对话框

单击 New…，把 IAS 服务器所含的库文件(指 JBuilder4\IAS\lib 目录下面的 jar 文件)都添加到 JBuilder4 的库列表中，作为一个新库，库名为 IAS。如图 2.2 所示。

单击 OK→OK，完成 Configure Libraries 的工作。

在 JBuilder4 中，单击 Tools→Enterprise Setup….→CORBA Tab，如图 2.3 所示。

选择 VisiBroker→Edit….，出现了 Edit Configuration 对话框，如图 2.4 所示。

在 Path for ORB tools 下面的文本框中，设定 IAS 服务器安装目录的 bin 文件夹的路径，然后把 Library for projects 下面的文本框设为 IAS，这样使得当前的工程可以使用 IAS 库，IAS 库含有 EJB API。一切就绪后，单击 OK。

接着上面一步，选择 Application Server Tab→IAS 4.1 Tab，如图 2.5 所示。

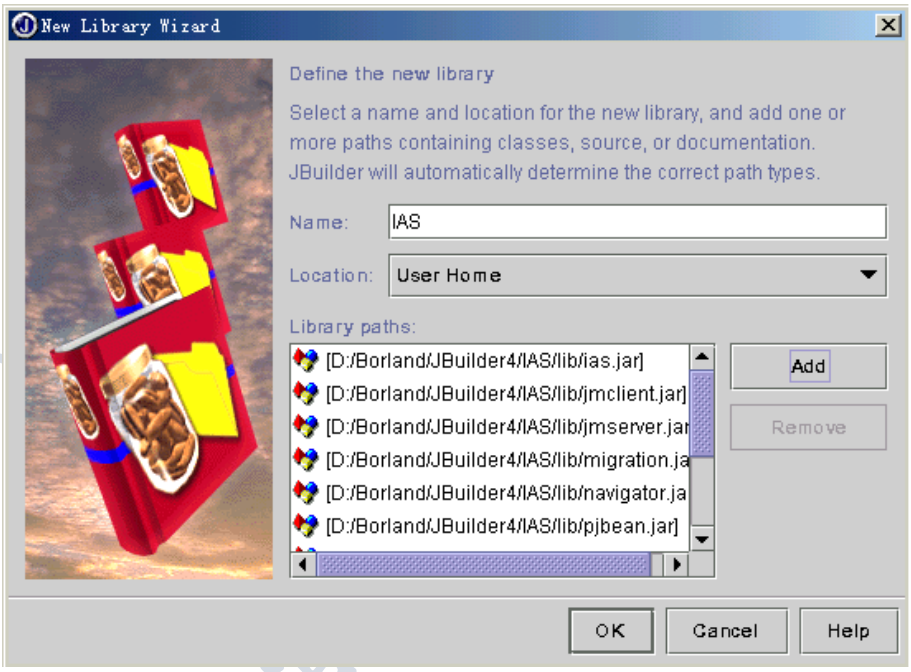


图 2.2　JBuilder4 的 New Library Wizard 对话框

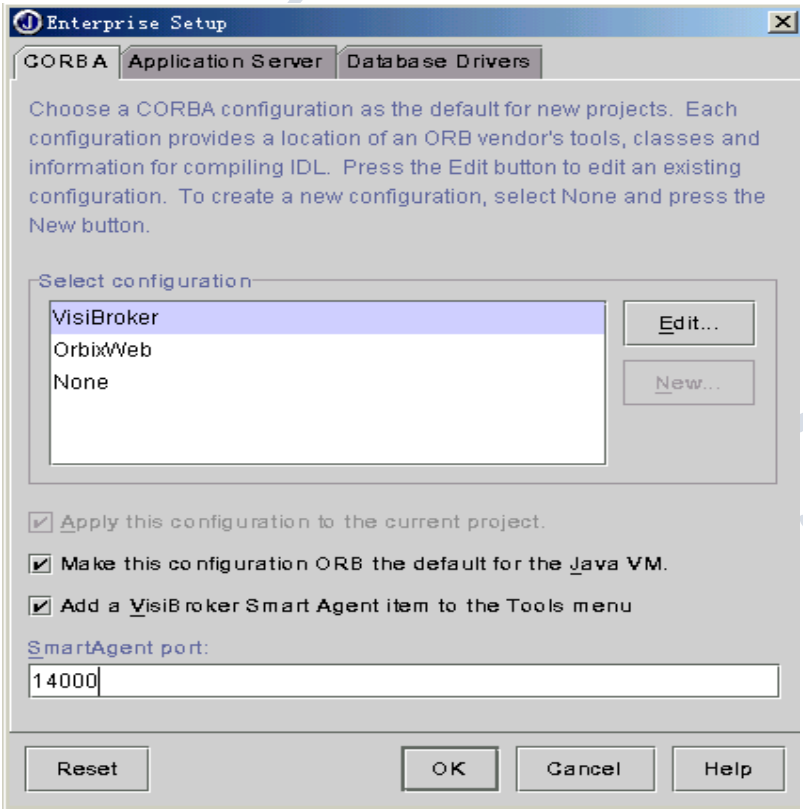


图 2.3　JBuilder4 的 Enterprise Setup 对话框

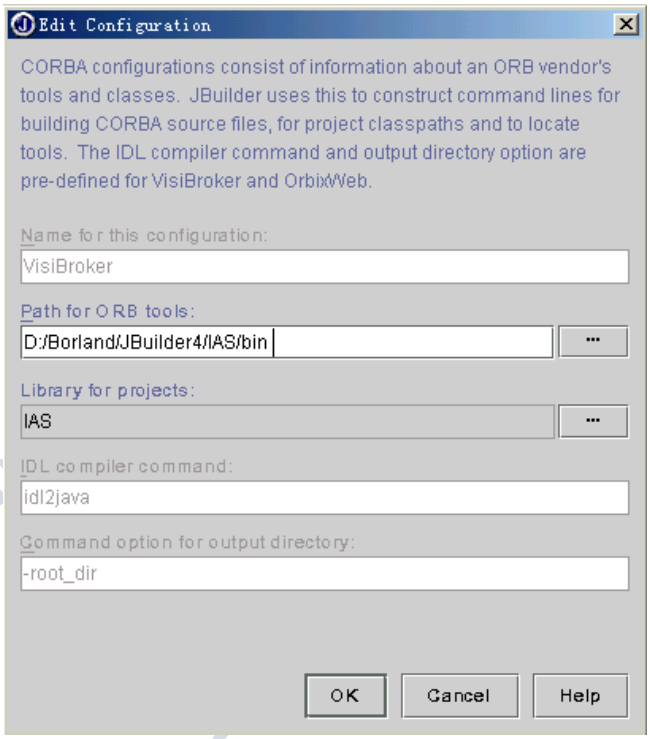


图 2.4　JBuilder4 的 Enterprise Setup 对话框

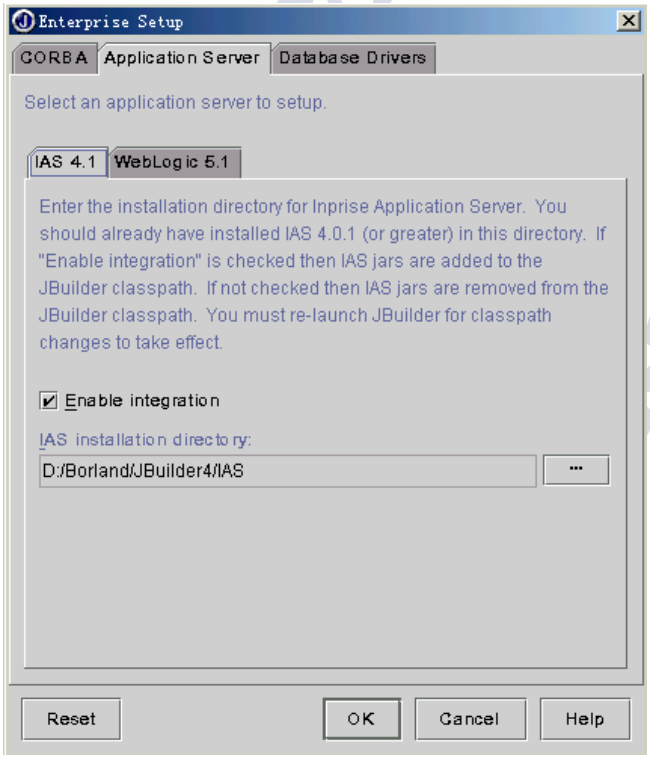


图 2.5　JBuilder4 的 Enterprise Setup 对话框

在 Enable integration 前面的框中打勾，在 IAS installation directory 下面的文本框中，设定 IAS 服务器的安装目录。单击 OK→OK，回到 JBuilder4 的主窗口。

关闭 JBuilder 4，再次启动 JBuilder 4，配置开发环境的工作就告一段落了。

2.4.3　创建 EJB 工程

由本小节开始，我们将一步一步的引导读者创建一个完整的 EJB 项目。

在 JBuilder4 中，依次选择 File→New…→Enterprise Tab→Empty EJB Group，如图 2.6 所示。

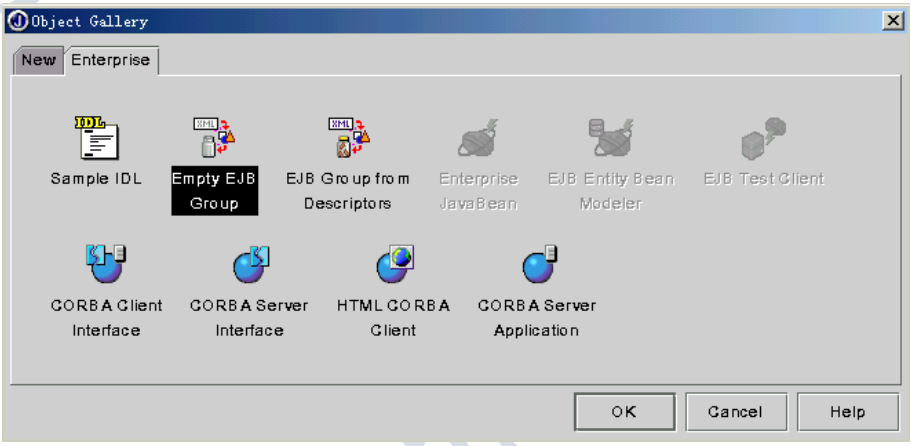


图 2.6　JBuilder4 的 Object Gallery 对话框

单击 OK，出现下一个对话框，设定 Project name 为 HelloWorld，单击 Next，又出现新对话框。

接着上一步，不要做任何改动，单击 Next，出现新对话框，输入 Title、Author、Company、Project Description 等注释信息，单击 Finish。

接着上一步，这时候出现了一个名为 Empty EJB Group Wizard 的对话框。在 Name 一栏中输入 HelloWorld，如图 2.7 所示。

单击 OK，就回到了 JBuilder4 的主界面。

在 JBuilder4 中，依次选择 File→New…→Enterprise Tag→Enterprise JavaBeans→OK，出现了 Enterprise JavaBeans Wizard 的对话框，在第一个对话框中，什么也不要改动，Available EJB groups 下面的下拉列表框的值应该缺省为 HelloWorld.ejbgrp，单击 Next，出现了下一个对话框。

接上一步，在 Class Information→Class Name 一栏中，改变缺省值 EnterpriseBean1 为 HelloWorld，其它地方不要改动(package 为 helloworld，Base Class 为 java.lang.Object，options 选中 Stateless session bean，这表明将要创建一个无状态的 Session EJB)，单击 Next，出现下一个对话框，如图 2.8 所示。

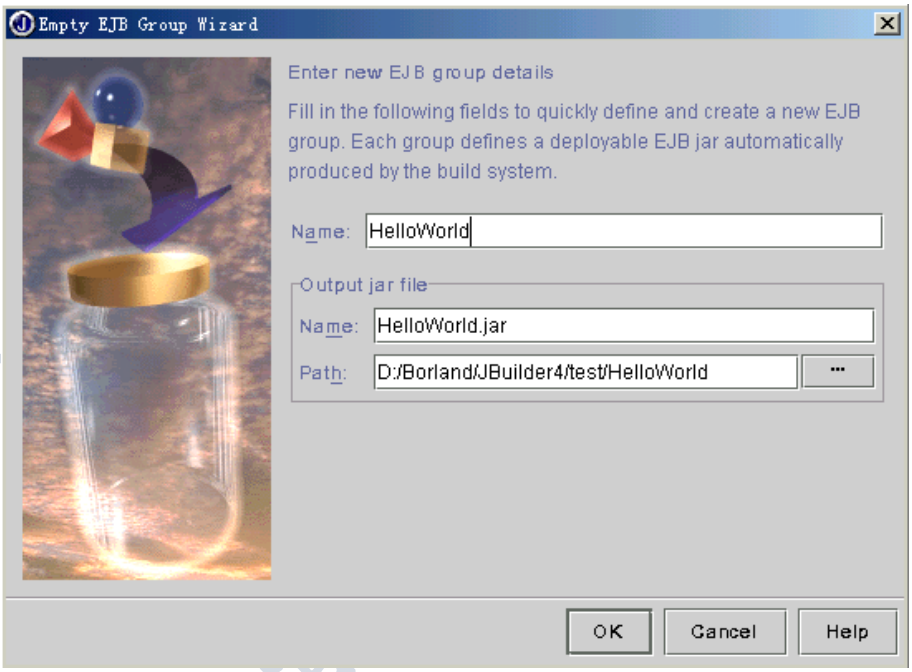


图 2.7　JBuilder4 的 Empty EJB Group Wizard 对话框

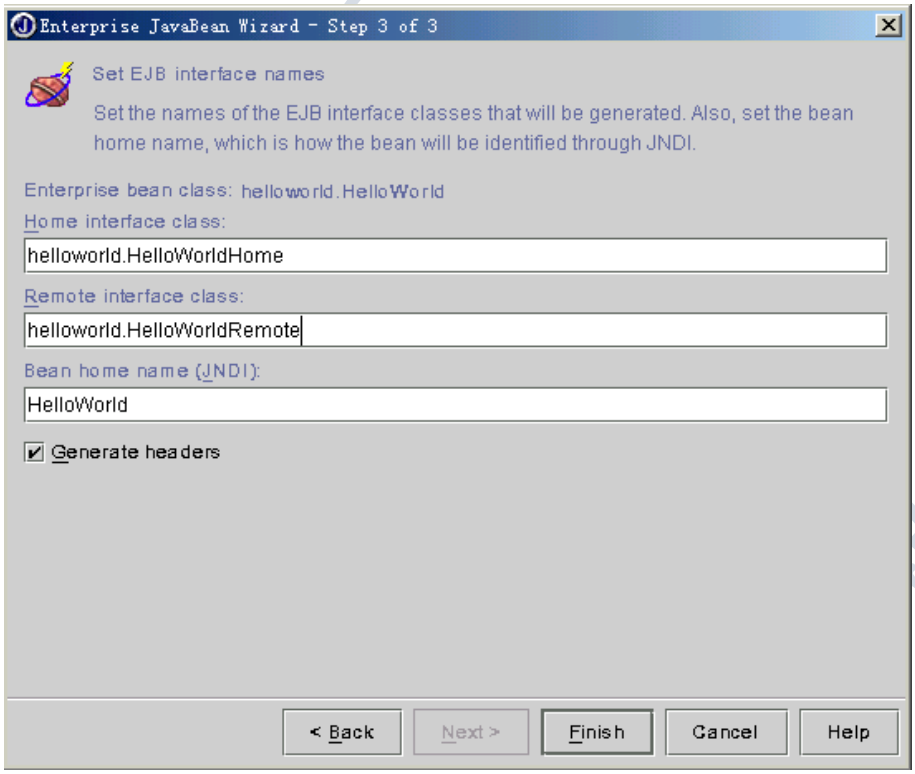


图 2.8　JBuilder4 的 Enterprise JavaBeans 对话框

在图 2.8 中，读者不难看出，EJB Home Interface 的名字为 HelloWorldHome，Remote

Interface 的名字为 HelloWorldRemote，Session EJB 的名字为 HelloWorld，读者可以任意改变这些接口/类的名称，只是不要相同就可以了，不过我们建议读者不要做任何改动。

接着上一步，单击 Finish，回到 JBuilder4 的主开发界面。

在 JBuilder4 中，依次选择 Project→Make Project，编译项目，应该没有任何错误，如果出现错误，请检查是否把 IAS 库添加到当前 Project 的 CLASSPATH 中去了(使用菜单 Project→Project Properties…进行设置)，还需要按照上面的步骤，一步步检查，看看哪一步做错了，改正过来，再编译一遍。

如果项目编译成功，那说明 EJB 已经基本开发成形了。

2.4.4　开发 EJB 类

EJB 类是实现商业逻辑、商业方法的地方，在 2.4.3 小节中，我们已经利用 JBuilder4 提供的 Wizard，自动生成了一个最简单的 EJB，自然也包括了一个最简单的 EJB 类，我们需要再此基础上进行二次开发，添加我们所需要的方法代码。

在 JBuilder4 的主窗口中，左侧，双击 HelloWorld.java，然后就可以在右侧的文件编辑区编辑 HelloWorld.java 的源代码。HelloWorld.java 的初始源代码如程序清单 2.6。

程序清单 2.6

```
package helloworld;

import java.rmi.*;
import javax.ejb.*;

/**
* Title:HelloWorld.java
* Description:create session ejb class
* Copyright:      Copyright (c) 1999
* Company:Peking University
* @author:fancy
* @version 1.0
*/

public class HelloWorld implements SessionBean
{
    private SessionContext sessionContext;
    public void ejbCreate()
    {
    }
    public void ejbRemove()
    {
    }
    public void ejbActivate()
    {
```

北京希望电子出版社

```
}
public void ejbPassivate()
{
}
public void setSessionContext(SessionContext context)
{
        sessionContext = context;
}
}
```

在 HelloWorld.java 文件中，添加如下的代码段：

```
public String getMessage()
{
    return “Hello World”;
}
```

方法 getMessage()也就是所谓的商业方法、商业逻辑。当然了，我们只是举一个最最简单的例子，实际上 EJB 类的代码十分复杂，绝不会如此简单，但是我们可以在此基础上继续开发，添加更多的方法与代码，完成更复杂的功能。

在 JBuilder4 主窗口的右下方，单击 HelloWorld Tab→Bean Tab→Methods，在 JBuilder4 主窗口右侧中间出现了 HelloWorld 类的方法列表，把滚动条移动到最下方，找到 getMessage()方法，getMessage()方法前面有一个小方框，我们在这个小方框上打勾，然后保存所有的文件，如图 2.9 所示。

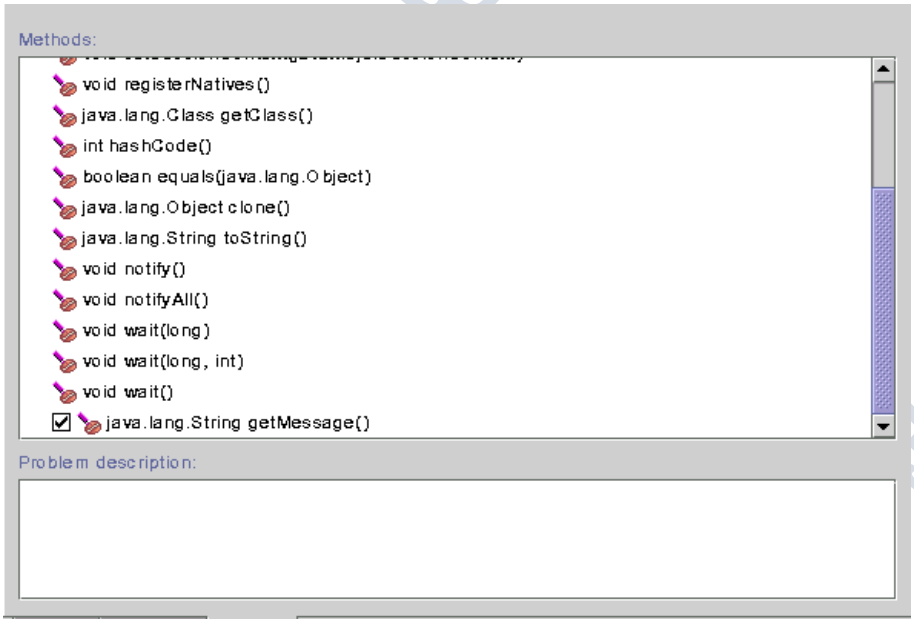


图 2.9　使商业方法(getMessage())与 EJB 的 Remote 接口建立关联

在 JBuilder4 中，依次选择 Project→Make Project，编译器提示下面的错误：

Specification compliance error in "HelloWorld.jar": Please run Verify in the EJB DD Editor

"HelloWorld.ejbgrp": Stateless Session Bean: "HelloWorld" The Java-IDL reverse mapping does not support Java type names which have a package name and a type name which are the same ignoring case: helloworld.HelloWorld

"HelloWorld.ejbgrp": Stateless Session Bean: "HelloWorld" There must be a transaction-attribute associated with the method: public abstract java.lang.String helloworld.HelloWorldRemote.getMessage() throws java.rmi.RemoteException

不要管这些错误，请看下一小节。

2.4.5　开发 Remote Interface

根据上文的描述，Remote Interface 是客户端程序与 EJB 实例之间的桥梁，客户端正是通过 Remote 接口的实例对象，由 EJB Object 代理对 EJB 商业逻辑、商业方法的调用。Remote Interface 实际上是对 EJB 商业方法的一个封装，客户端程序可以调用的方法都在 Remote Interface 中得到了声明，真正的实现却是在 EJB 类/EJB Object 中。

在 JBuilder4 主界面中，左侧，双击 HelloWorldRemote.java，在右侧的编辑区就可以编辑 HelloWorldRemote.java 文件了。HelloWorldRemote.java 的初始代码如程序清单 2.7。

程序清单 2.7

```
package helloworld;

import java.rmi.*;
import javax.ejb.*;

/**
* Title:HelloWorldRemote.java
* Description:create a Remote Interface class
* Copyright:      Copyright (c) 1999
* Company:Peking University
* @author:fancy
* @version 1.0
*/

public interface HelloWorldRemote extends EJBObject
{
    public java.lang.String getMessage() throws RemoteException;
}
```

在上面的代码中，可以看到 EJB 的 Remote Interface 声明了 getMessage()方法，我们在 HelloWorld 类中已经实现了这个商业方法。

我们一般不直接编辑 Remote Interface 的代码，建议读者采取的方法是：在 HelloWorld 类中加入商业方法的实现代码，然后单击 Bean→Methods，在商业方法名字前的小方框中打勾，保存所有文件，那么这些商业方法就会自动在 Remote Interface 中声明了。

2.4.6　开发 Home Interface

接下来，我们该看看 Home Interface 如何开发了。在上文中，我们已经提到过，Home Interface 十分重要，客户端的应用程序必须定位 Home 接口，然后通过它的 create()方法来创建 EJB 的远程接口对象。

我们使用 JBuilder4 的 EJB Wizard 自动创建的 Home Interface 的初始代码如程序清单 2.8。

程序清单 2.8

```
package helloworld;

import java.rmi.*;
import javax.ejb.*;

/**
* Title:HelloWorldHome.java
* Description:to create a Home Interface class
* Copyright:      Copyright (c) 1999
* Company:Peking University
* @author:fancy
* @version 1.0
*/

public interface HelloWorldHome extends EJBHome
{
    public HelloWorldRemote create() throws RemoteException，CreateException;
}
```

如果我们开发的是会话 EJB，那么一般不需要对 Home Interface 的源代码进行修改。

2.4.7　编辑部署文件

下面我们介绍如何利用 JBuilder4 编辑 EJB 的部署描述符。

在 JBuilder4 窗口的左侧，双击 HelloWorld.ejbgrp，左侧窗口被分成了上下两部分，下面的部分为 EJB Deployment Descriptor，在下面的部分中展开 HelloWorld，如图 2.10 所示。

单击 Container Transactions，JBuilder4 主界面的右侧出现了部署描述符的图形编辑界面，单击 Add，保存全部文件，如图 2.11 所示。

然后单击图 2.11 上方打勾的图标，检验部署描述符的语法，没有任何错误。这样部署描述符就算开发完成了，在图 2.11 中，我们可以通过单击 EJB DD Editor 来编辑部署描述符，通过单击 EJB DD Source 面板来查看部署描述符的源文件。JBuilder4 为我们创建了两个 XML 文件作为部署描述符，分别是 ejb-jar.xml 和 ejb-inprise.xml，这两个文件的代码如程序清单 2.9 及 2.10。

程序清单 2.9(ejb-jar.xml)

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

```
<!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>
<enterprise-beans>
<session>
    <ejb-name>HelloWorld</ejb-name>
    <home>helloworld.HelloWorldHome</home>
    <remote>helloworld.HelloWorldRemote</remote>
    <ejb-class>helloworld.HelloWorld</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
</session>
</enterprise-beans>
<assembly-descriptor>
<container-transaction>
    <method>
    <ejb-name>HelloWorld</ejb-name>
    <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
</assembly-descriptor>
</ejb-jar>
```

程序清单 2.10(ejb-insprise.xml)

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

<!DOCTYPE inprise-specific PUBLIC '-//Inprise Corporation//DTD Enterprise
    JavaBeans 1.1//EN"http://www.borland.com/devsupport/appserver/dtds/
    ejb-inprise.dtd'>

<inprise-specific>
<enterprise-beans>
    <session>
        <ejb-name>HelloWorld</ejb-name>
        <bean-home-name>HelloWorld</bean-home-name>
    </session>
</enterprise-beans>
</inprise-specific>
```

北京希望电子出版社

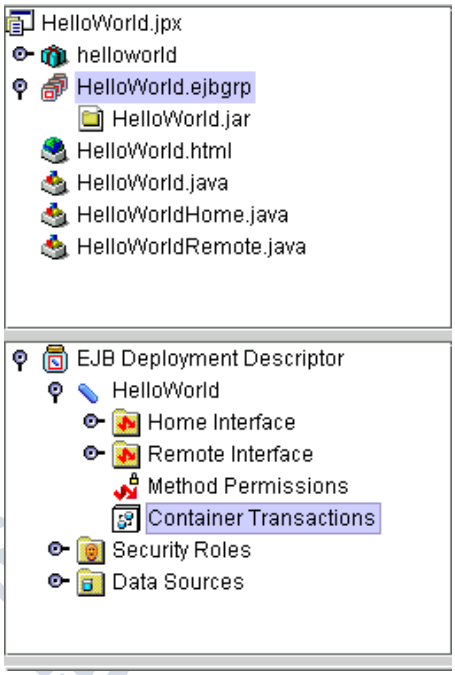


图 2.10　JBuilder4 的 EJB Deployment Descriptor 视图

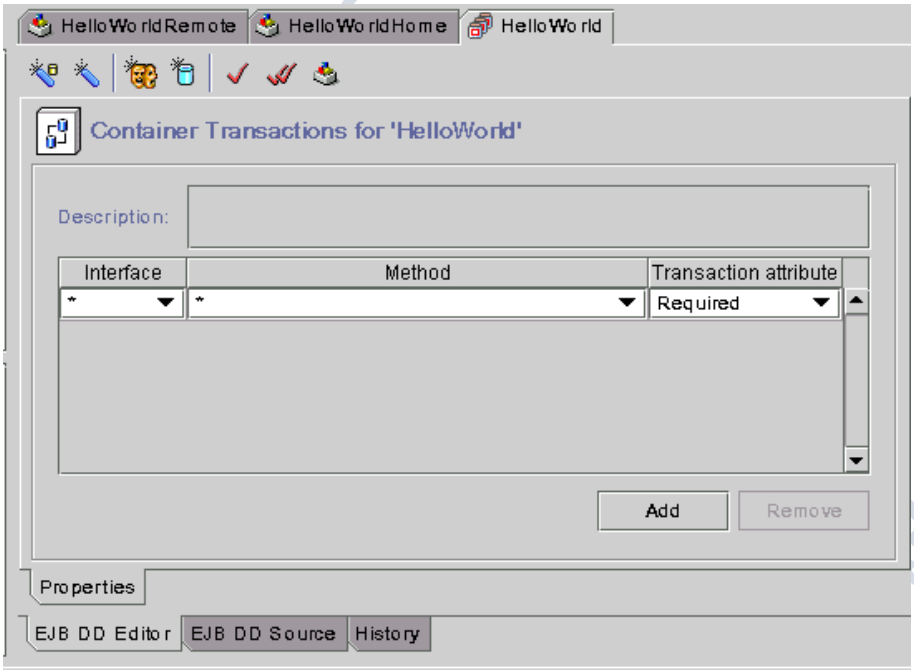


图 2.11　JBuilder4 的 EJB Deployment Descriptor 图形化编辑界面

我们既可以在 EJB DD Editor 面板中以图形化的方式编辑部署描述符，也可以在 EJB DD Source 面板中直接以源代码级的方式编辑部署描述符文件，编辑完以后，千万不要忘了对部署描述符进行验证(Verify)。

2.5　如何开发 EJB(二)

2.5.1　运行环境配置

本节我们接着介绍如何开发 EJB 及如何配置 EJB 组件的运行环境。在 JBuilder4 的主窗口中，左侧，右键单击 HelloWorldHome.java→选择 Properties….→选择 Build Tab→VisiBroker，在 Generate IIOP 前面的小方框前打勾，如图 2.12 所示。

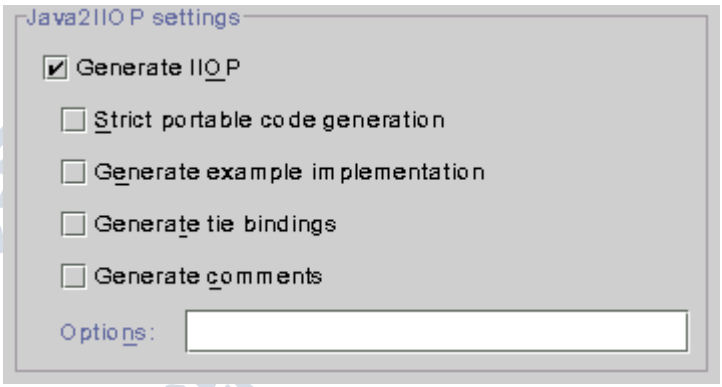


图 2.12　配置 Java2IIOP 的属性

单击 OK。然后选择 Run→Configuration…..，如图 2.13 所示。

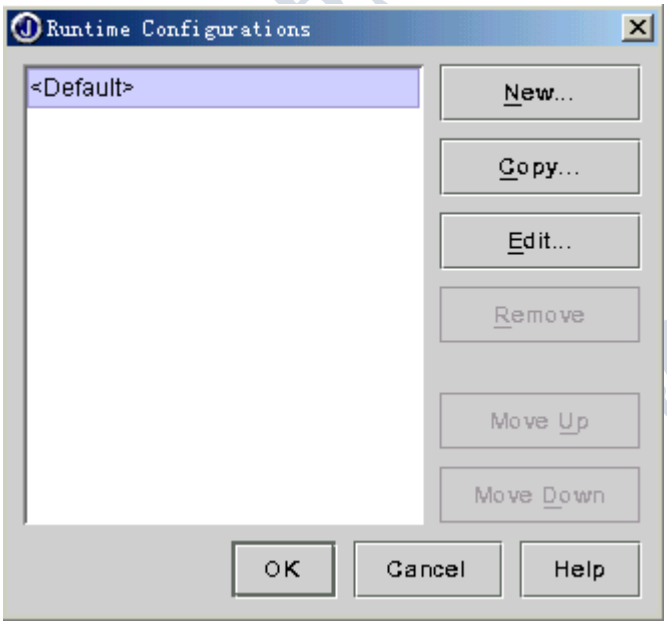


图 2.13　配置 EJB 的运行环境

在图 2.13 中，选中<Default>，单击 OK 即可。这样 EJB 的运行环境就算配置好了。

2.5.2　创建 EJB Container

本小节中，我们将创建一个 EJB Container，方法很简单，只需要 Make Project 就可以了，JBuilder4 会根据 Remote Interface 和 Home Interface 自动产生十几个 Java 类文件，这些文件的名字如下所示：

HelloWorldRemoteHelper.java
HelloWorldRemoteHolder.java
HelloWorldHomePOA.java
_HelloWorldHome_Stub.java
HelloWorldRemotePOAInvokeHandler.java
HelloWorldRemoteOperations.java
HelloWorldRemotePOA.java
_HelloWorldRemote_Stub.java
HelloWorldHomeHelper.java
HelloWorldHomeHolder.java
HelloWorldHomePOAInvokeHandler.java
HelloWorldHomeOperations.java

这些文件的作用就是产生 Home Object 和 EJB Object，须知 EJB Container 就是由 Home Object 构成的。在上面列举的文件中，以 _HelloWorldHome_Stub.java 和 _HelloWorldRemote_Stub.java 最为重要，前者实现了 Home Interface 所定义的 create()方法，后者实现了 Remote Interface 所定义的商业方法，在本例中是 getMessage()方法。这两个文件的源代码如程序清单 2.11。

程序清单 2.11(_HelloWorldHome_Stub.java)

```
package helloworld;

public class _HelloWorldHome_Stub extends javax.ejb._EJBHome_Stub
    implements HelloWorldHome，HelloWorldHomeOperations
{
    final public static java.lang.Class _opsClass =
        helloworld.HelloWorldHomeOperations.class;

    public java.lang.String[] _ids ()
    {
        return __ids;
    }

    private static java.lang.String[] __ids =
    {
        "RMI:helloworld.HelloWorldHome:0000000000000000",
        "RMI:javax.ejb.EJBHome:0000000000000000"
    };
```

```
public helloworld.HelloWorldRemote create ()
    throws java.rmi.RemoteException，  javax.ejb.CreateException
{
    try
    {
        while (true)
{
            if (!_is_local())
{
            org.omg.CORBA.portable.OutputStream _output = null;
            org.omg.CORBA.portable.InputStream    _input    = null;
            helloworld.HelloWorldRemote _result;
            try
            {
            _output = this._request("create"，true);
            _input = this._invoke(_output);
            //FIX: Cannot use helper class
            //because of potential stub downloading
            _result= (helloworld.HelloWorldRemote)
                javax.rmi.PortableRemoteObject.narrow
                    (_input.read_Object
                        (helloworld.HelloWorldRemote.class)，
                            helloworld.HelloWorldRemote.class);
            return _result;
            }
            catch (org.omg.CORBA.portable.ApplicationException
                _exception)
            {
            final org.omg.CORBA.portable.InputStream in =
                _exception.getInputStream();
            java.lang.String _exception_id =
                _exception.getId();
            if (_exception_id.equals("IDL:javax/ejb/CreateEx:1.0"))
            {
                _exception_id = in.read_string();
                throw (javax.ejb.CreateException)
                    ((org.omg.CORBA_2_3.portable.InputStream)in).
                        read_value();
            }
            //FIX: Wrap original Exception here?
            throw new java.rmi.UnexpectedException
                ("Unexpected User Exception: " + _exception_id);
```

北京希望电子出版社

```
                }
                catch (org.omg.CORBA.portable.RemarshalException
                    _exception)
                {
                continue;
                }
                finally
                {
                    this._releaseReply(_input);
                }
            }
            else
            {
                final org.omg.CORBA.portable.ServantObject _so =
                    _servant_preinvoke("create"，_opsClass);
                if (_so == null)
                {
                    continue;
                }
                final helloworld.HelloWorldHomeOperations _self =
                    (helloworld.HelloWorldHomeOperations)_so.servant;
                try
                {
                    return _self.create();
                }
                finally
                {
                    _servant_postinvoke(_so);
                }
            }
        }
    }
                catch (org.omg.CORBA.portable.UnknownException ex)
                {
                    if (ex.originalEx instanceof java.lang.RuntimeException)
{
                        throw (java.lang.RuntimeException) ex.originalEx;
                }
                else if (ex.originalEx instanceof Exception)
                {
                    throw new java.rmi.ServerException(ex.getMessage()，
                        (java.lang.Exception)ex.originalEx);
                }
```

北京希望电子出版社

```
                else
                {
                    throw new java.rmi.ServerError
                        (ex.getMessage()，(java.lang.Error) ex.originalEx);
                }
            }
            catch (org.omg.CORBA.SystemException ex)
            {
                throw javax.rmi.CORBA.Util.mapSystemException(ex);
            }
        }
    }
```

在程序清单 2.11 中，实现了 create() 方法，该方法是在 Home Interface(HelloWorldHome.java)中声明的，该方法的返回值为 HelloWorldRemote 对象。

程序清单 2.12(_HelloWorldRemote_Stub.java)

```
package helloworld;

public class _HelloWorldRemote_Stub extends javax.ejb._EJBObject_Stub
    implements HelloWorldRemote，HelloWorldRemoteOperations
{
    final public static java.lang.Class _opsClass =
        helloworld.HelloWorldRemoteOperations.class;

    public java.lang.String[] _ids ()
    {
        return __ids;
    }

    private static java.lang.String[] __ids =
    {
        "RMI:helloworld.HelloWorldRemote:0000000000000000"，
        "RMI:javax.ejb.EJBObject:0000000000000000"
    };

    public java.lang.String getMessage () throws java.rmi.RemoteException
    {
        try
        {
            while (true)
            {
                if (!_is_local())
                {
                    org.omg.CORBA.portable.OutputStream _output = null;
                    org.omg.CORBA.portable.InputStream  _input  = null;
                    java.lang.String _result;
```

```
            try
            {
                _output = this._request("_get_message"，true);
                _input = this._invoke(_output);
                _result = org.omg.CORBA.WStringValueHelper.read
                    (_input);
                return _result;
            }
            catch (org.omg.CORBA.portable.ApplicationException
                _exception)
            {
                final org.omg.CORBA.portable.InputStream in =
                    _exception.getInputStream();
                java.lang.String _exception_id =
                    _exception.getId();
                //FIX: Wrap original Exception here?
                throw new java.rmi.UnexpectedException("Unexpected
                    User Exception: " + _exception_id);
            }
            catch (org.omg.CORBA.portable.RemarshalException
                _exception)
            {
                continue;
            }
            finally
            {
                this._releaseReply(_input);
            }
        }
        else
        {
            final org.omg.CORBA.portable.ServantObject _so =
                _servant_preinvoke("_get_message"，_opsClass);
            if (_so == null)
            {
                continue;
            }
            final helloworld.HelloWorldRemoteOperations
                _self = (helloworld.HelloWorldRemo
                teOperations)_so.servant;
            try
            {
                return _self.getMessage();
            }
            finally
            {
```

```
                        _servant_postinvoke(_so);
                    }
                }
            }
        }
        catch (org.omg.CORBA.portable.UnknownException ex)
        {
            if (ex.originalEx instanceof java.lang.RuntimeException)
            {
                throw (java.lang.RuntimeException) ex.originalEx;
            }
            else if (ex.originalEx instanceof Exception)
            {
                throw new java.rmi.ServerException(ex.getMessage(),
                    (java.lang.Exception)ex.originalEx);
            }
            else
            {
                throw new java.rmi.ServerError(ex.getMessage(),
                    (java.lang.Error) ex.originalEx);
            }
        }
        catch (org.omg.CORBA.SystemException ex)
        {
            throw javax.rmi.CORBA.Util.mapSystemException(ex);
        }
    }
}
```

在程序清单 2.15(_HelloWorldRemote_Stub.java)中，实现了商业方法——getMessage()方法，该方法是在 Remote Interface(HelloWorldRemote.java)中声明的，至于实现的细节，我们就不在这里讨论了。

2.5.3　发布 EJB 服务

经过上面这么多步的工作，EJB 组件的开发应该告一段落了，下面我们应该发布 EJB 服务，因为我们还不知道 EJB 能不能够按照我们的设计正确运行，所以暂时不把 EJB 发布到真正的 EJB Server 上去，而是使用 IAS 服务器所带的 Smart Agent 作为临时的 EJB Server/Container，发布我们刚刚编写好的 EJB，以便进行测试，测试成功以后再把它分发到真正的 EJB Server 上去。

首先需要运行 Smart Agent，即 osagent.exe 程序，osagent.exe 程序在 IAS 安装目录的 bin 文件夹内。

然后转回到 JBuilder4 的主界面，选择 Run→Run Project。JBuilder4 首先会编译 EJB，然后把它分发到 Smart Agent 中。JBuilder4 会出现类似于下面的提示信息。

```
Inprise EJB Container
```

```
=====================
        server version : 4.1.1
    server build date : Aug 18，2000
          java version : 1.3.0
        java vendor : Sun Microsystems Inc.
            heap size : 1984 Kb
    java class path : D:\Inprise\AppServer\lib\navigator.jar
                    : D:\Inprise\AppServer\lib\vbdev.jar
                    : D:\Inprise\AppServer\lib\vbejb.jar
                    : D:\Inprise\AppServer\lib\vbjdev.jar
                    : D:\Inprise\AppServer\lib\vbjorb.jar
                    : D:\Borland\JBuilder4\test\HelloWorld\classes
                    : D:\Borland\JBuilder4\IAS\lib\ias.jar
                    : D:\Borland\JBuilder4\IAS\lib\jmclient.jar
                    : D:\Borland\JBuilder4\IAS\lib\jmserver.jar
                    : D:\Borland\JBuilder4\IAS\lib\migration.jar
                    : D:\Borland\JBuilder4\IAS\lib\navigator.jar
                    : D:\Borland\JBuilder4\IAS\lib\pjbean.jar
                    : D:\Borland\JBuilder4\IAS\lib\servlet.jar
                    : D:\Borland\JBuilder4\IAS\lib\vbdev.jar
                    : D:\Borland\JBuilder4\IAS\lib\vbejb.jar
                    : D:\Borland\JBuilder4\IAS\lib\vbjdev.jar
                    : D:\Borland\JBuilder4\IAS\lib\vbjorb.jar
                    : D:\BORLAND\JBUILDER4\JDK1.3\demo\jfc\Java2D\Java2Demo.jar
                    : D:\BORLAND\JBUILDER4\JDK1.3\jre\lib\i18n.jar
                    : D:\BORLAND\JBUILDER4\JDK1.3\jre\lib\jaws.jar
                    : D:\BORLAND\JBUILDER4\JDK1.3\jre\lib\rt.jar
                    : D:\BORLAND\JBUILDER4\JDK1.3\jre\lib\sunrsasign.jar
                    : D:\BORLAND\JBUILDER4\JDK1.3\lib\dt.jar
                    : D:\BORLAND\JBUILDER4\JDK1.3\lib\tools.jar
=====================
Initializing ORB..... done
Initializing JNS..... done
Initializing JTS.... done
Initializing JSS.....Developer's License (no connection limit)
Copyright (c) 1996-2000 Inprise Corporation.    All rights reserved.
License for JDataStore development only - not for redistribution
Registered to:
Inprise Application Server Development Licensee
Inprise Application Server Customer
..... done
Initializing JDB..... done
Initializing EJBs..... done
```

北京希望电子出版社

```
Container [ejbcontainer] is ready
        EJB Container Statistics
        ========================
        Time              Tue Mar 16 00:00:21 CST 1999
        Memory (used)     1714 Kb    (max 1714 Kb)
        Memory (total)    2680 Kb    (max 2680 Kb)
        Memory (free)     36.0%
        ------------------------
        Home              HelloWorld
        Total in memory   0
        Total in use      0
        ========================
```

如果出现了上面的信息，那么就表明，这个 HelloWorld EJB 已经成功地部署到 Smart Agent 中了。下面我们应该编写一个简单的 EJB 客户端程序来测试这个 EJB 了。

2.5.4　测试 EJB 服务

本小节我们将介绍如何利用 JBuilder4 来编写一个 EJB 客户端测试我们刚才所发布的 EJB。在 JBuilder4 中，选择 File→New…→Enterprise Tab→EJB Test Client→OK，将出现如图 2.14 所示的窗口。

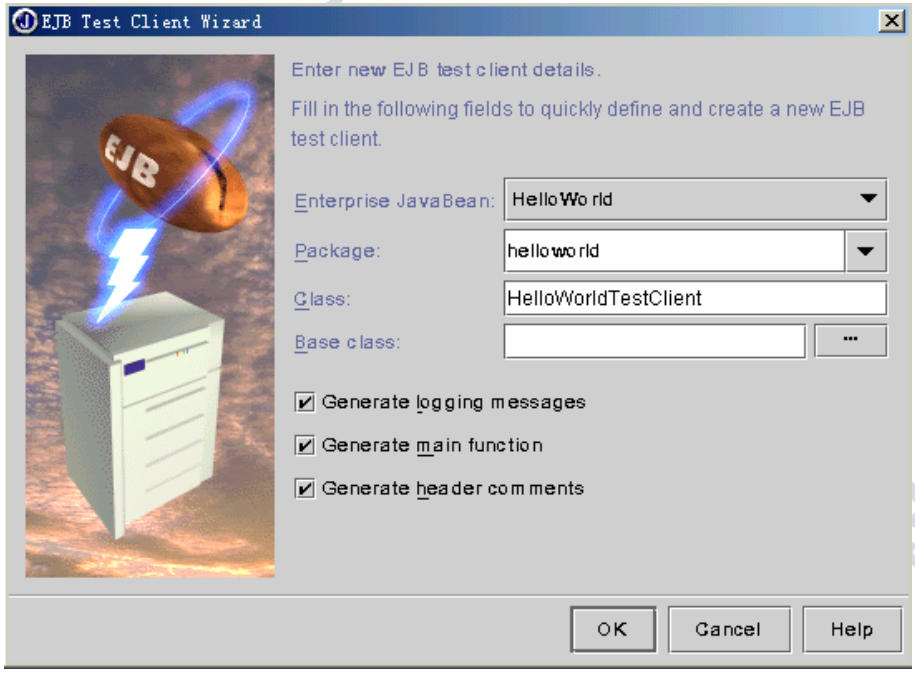


图 2.14　JBuilder4 的 EJB Test Client Wizard 对话框

在图 2.14 中，我们把 Class 框的缺省值 HelloWorldTestClient1 改为 HelloWorldTestClient，其他的一切都不要改动，单击 OK，回到 JBuilder4 的主界面，JBuilder4 为我们自动创建了 HelloWorldTestClient.java 程序。编辑 HelloWorldTestClient.java，使得它如下面的程序清单

2.13 所示，其中黑体的部分是我们自己加上去的，其余的部分是 JBuilder4 自动产生的。

程序清单 2.13

```
package helloworld;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;

/**
* Title:HelloWorldTestClient.java
* Description:test the EJB
* Copyright:      Copyright (c) 1999
* Company:Peking University
* @author:fancy
* @version 1.0
*/

public class HelloWorldTestClient
{
     private static final String ERROR_NULL_REMOTE = "Remote interface reference is null.It must be
          created by calling one of the Home interface methods first.";
     private static final int MAX_OUTPUT_LINE_LENGTH = 50;
     private boolean logging = true;
     private HelloWorldHome helloWorldHome = null;
     private HelloWorldRemote helloWorldRemote = null;

     /**Construct the EJB test client*/
     public HelloWorldTestClient()
     {
          long startTime = 0;
          if (logging)
          {
               log("Initializing bean access.");
               startTime = System.currentTimeMillis();
          }

          try
          {
               //get naming context
               Context ctx = new InitialContext();

               //look up jndi name
               Object ref = ctx.lookup("HelloWorld");

               //cast to Home interface
               helloWorldHome = (HelloWorldHome)
                    PortableRemoteObject.narrow(ref，HelloWorldHome.class);
```

```
            HelloWorldRemote hwr=create();
            String msg=hwr.getMessage();
            System.out.println(msg);
            if (logging)
            {
                long endTime = System.currentTimeMillis();
                log("Succeeded initializing bean access.");
                log("Execution time: " + (endTime - startTime) + " ms.");
                }
        }
        catch(Exception e)
        {
            if (logging)
            {
                log("Failed initializing bean access.");
            }
            e.printStackTrace();
        }
}

//-------------------------------------------------------------------------
// Methods that use Home interface methods to generate a Remote interface reference
//-------------------------------------------------------------------------

public HelloWorldRemote create()
{
    long startTime = 0;
    if (logging)
    {
        log("Calling create()");
        startTime = System.currentTimeMillis();
    }
    try
    {
        helloWorldRemote = helloWorldHome.create();
        if (logging)
        {
            long endTime = System.currentTimeMillis();
            log("Succeeded: create()");
            log("Execution time: " + (endTime - startTime) + " ms.");
        }
    }
    catch(Exception e)
    {
        if (logging)
        {
```

北京希望电子出版社

```
                log("Failed: create()");
            }
            e.printStackTrace();
        }

        if (logging)
        {
            log("Return value from create(): " + helloWorldRemote + ".");
        }
        return helloWorldRemote;
    }

    //--------------------------------------------------------------------------
    // Methods that use Remote interface methods to access data through the bean
    //--------------------------------------------------------------------------

    public String getMessage()
    {
        String returnValue = "";
        if (helloWorldRemote == null)
        {
            System.out.println("Error in getMessage(): " + ERROR_NULL_REMOTE);
            return returnValue;
        }
        long startTime = 0;
        if (logging)
        {
            log("Calling getMessage()");
            startTime = System.currentTimeMillis();
        }
        try
        {
            returnValue = helloWorldRemote.getMessage();
            if (logging)
            {
                long endTime = System.currentTimeMillis();
                log("Succeeded: getMessage()");
                log("Execution time: " + (endTime - startTime) + " ms.");
            }
        }
        catch(Exception e)
        {
            if (logging)
            {
                log("Failed: getMessage()");
            }
```

```
            e.printStackTrace();
        }

        if (logging)
    {
            log("Return value from getMessage(): " + returnValue + ".");
        }
        return returnValue;
    }

    //--------------------------------------------------------------------------
    // Utility Methods
    //--------------------------------------------------------------------------

    private void log(String message)
    {
        if (message.length() > MAX_OUTPUT_LINE_LENGTH)
        {
            System.out.println("-- " +
                message.substring(0，MAX_OUTPUT_LINE_LENGTH) + " ...");
        }
        else
        {
            System.out.println("-- " + message);
        }
    }
    /**Main method*/

    public static void main(String[] args)
    {
        HelloWorldTestClient client = new HelloWorldTestClient();
        // Use the client object to call one of the Home interface wrappers
        // above，to create a Remote interface reference to the bean.
        // If the return value is of the Remote interface type，you can use it
        // to access the remote interface methods.  You can also just use
        // the client object to call the Remote interface wrappers.
    }
}
```

HelloWorldTestClient.java 的运行流程如下，首先调用构造函数 HelloWorldTestClient()，在构造函数中，先是使用 lookup()方法，定位 Home Interface 的位置，获取 Home Object，亦即 helloWorldHome。然后调用 create()方法，该方法返回一个 Remote 对象(hwr)，接下来，调用 hwr 对象的 getMessage()方法，这其实是调用 HelloWorld EJB 的商业方法 getMessage()，getMessage()方法的返回值为“Hello World”。我们把返回值赋给一个字符串，然后输出。

客户端程序的运行方法如下：

在 JBuilder4 的主界面中，左侧，右键单击 HelloWorldTestClient.java→Run，运行输出

如下所示：

```
-- Initializing bean access.
-- Calling create()
-- Succeeded: create()
-- Execution time: 380 ms.
-- Return value from create(): Stub[repository_id=RMI ...
Hello World
-- Succeeded initializing bean access.
-- Execution time: 10660 ms.
```

读者也许已经发现了，这个 EJB 客户端程序的输出结果显示：调用这个 HelloWorld EJB 组件的商业方法(getMessage())花费了很长的时间，一共是 10660ms，亦即 10.66 秒，这并非是 EJB 组件的运行效率不佳，而是因为这个 EJB 组件的服务端(SmartAgent)和客户端(JBuilder4)都在笔者的电脑上面运行，与此同时，笔者的电脑上还运行了一个数据库服务器(MS SQL Server)、一个 Web 服务器(IAS)，还有一个字处理软件(Word)，系统资源有限，所以运行速度才这么慢，如果 EJB Server 在别的电脑上运行，EJB Client 在本地机上运行，真正实现了分布式处理的梦想，那时候 EJB 组件的运行效率还是相当高的。闲话少说，现在转入正题，由上面的运行结果可见，HelloWorld EJB 已经成功发布，成功运行了，怎么样，是不是很简单呢?下面我们该把这个 EJB 正式部署到商业服务器上去了，我们首先选用的服务器是 IAS4.1。

2.5.5　打包分发 EJB 服务

在这一小节中，将介绍如何把上面编写好的 HelloWorld EJB 分发到 IAS 服务器上，作为一项服务发布。

启动 IAS 服务器

第一步是启动 IAS 服务器，你可以从开始菜单启动 IAS 服务器，也可以进入 IAS 的 bin 目录下面，双击执行 ias.exe 程序，这样也可以运行 IAS 服务器，当出现了类似于下面的字样，那说明 IAS 服务器已经成功了。

```
Inprise Application Server 4.1.1
Copyright (c) 1999-2000 Inprise Corporation. All Rights Reserved.
Portions Copyright (c) 1997-1999 Sun Microsystems，Inc. All Rights Reserved.
Server rainbow Started
```

运行 EJB Deployment

在 JBuilder4 的主运行界面中，选择 Tools→EJB Deployment…，出现了如图 2.15 所示的窗口。

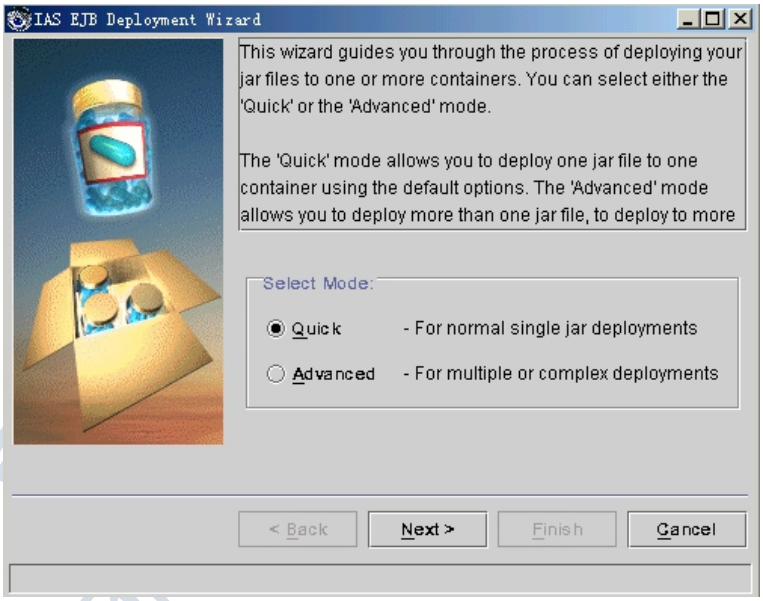


图 2.15　JBuilder4 的 IAS EJB Deployment Wizard 窗口

在图 2.15 中，有两个选项，Quick 和 Advanced，Quick 选项适合于发布单个 EJB 组件，而 Advanced 选项适合于发布多个 EJB 组件，在这个例子中，我们选择 Quick，然后单击 Next，出现了如图 2.16 所示的窗口。

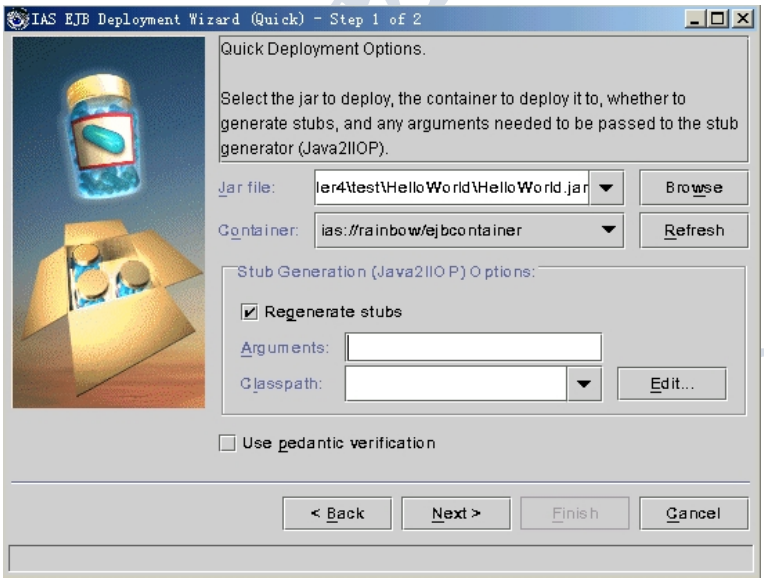


图 2.16　JBuilder4 的 IAS EJB Deployment Wizard 窗口

选择需要发布的 EJB 组件

在这一步，我们将选择需要发布的 EJB 组件和 EJB Container，如图 2.16 所示。

在图 2.16 中，Jar file 一栏用于填写你所希望发布的 EJB 组件所在的 jar 文件的路径，

你可以单击 Browse 按钮修改它，在 Container 一栏，列出了目前可用的 EJB Container，缺省状态下是 ias://localhost/ejbcontainer，如果这一栏为空，那么请确定 IAS 服务器是否已经运行了，如果 IAS 服务器不是出于运行状态，那是没有办法分发 EJB 服务的。

设置停当后，单击 Next，出现如图 2.17 所示的窗口。

EJB 组件分发成功

如果你上面的步骤没有出什么漏子的话，那么你将会看到如图 2.17 所示的画面：EJB 组件已经分发成功了。在图 2.17 中，单击 Finish 按钮，关闭这个窗口，回到 JBuilder4 的主界面，再次运行 EJB 的客户端程序(右键单击 HelloWorldTestClient.java→Run)，我们会看到这次的运行结果和上面使用 SmartAgent 作为 EJB Container 得到的结果几乎没有不同，如果有不同，那就是运行所耗费的时间不同。

到此为止，我们已经成功地开发了一个 EJB 组件，成功地测试了这个 EJB 组件，并且成功地把它发布到一个 EJB Container(IAS)中去了。我们是否应该到此结束呢?不!事情还没有结束呢，我们应该选用一个更好的 EJB Container，把 EJB 组件发布到哪里去，IAS 服务器作为 EJB Container 还不是最好的选择，对于 EJB Container，BEA 的 WebLogic、IBM 的 WebSphere 才是更好的选择。好了，你现在可以关闭 JBuilder4，进入下一小节的学习，在下一小节，我们将介绍如何把 EJB 组件分发到 WebLogic 服务器上，以 WebLogic 服务器作为 EJB 组件的 EJB Server/Container。

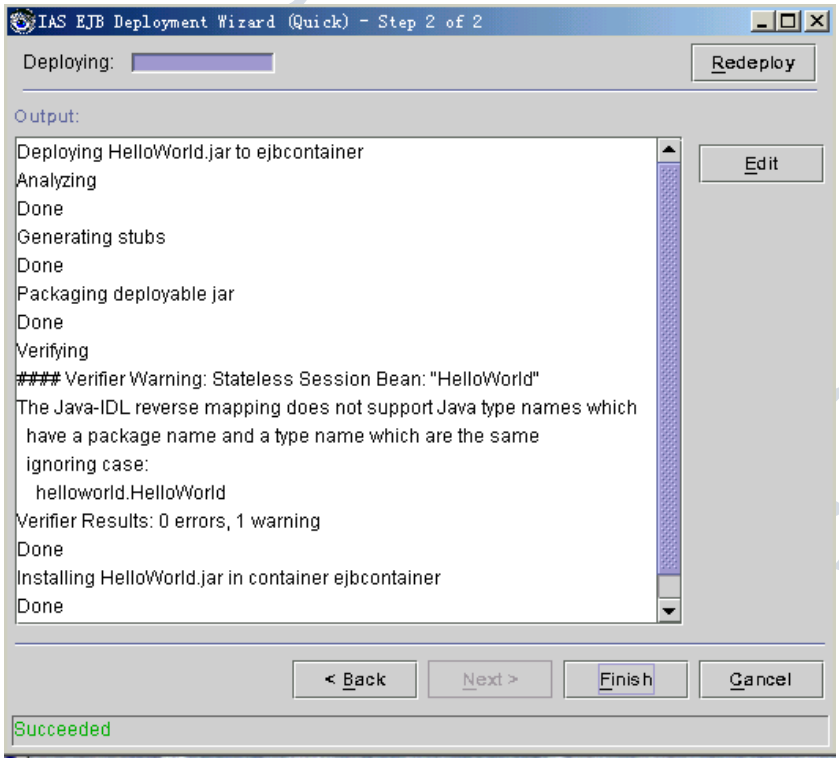


图 2.17　EJB 组件分发成功了

2.5.6　使用 WebLogic 服务器分发 EJB 服务

在本小节，我们将介绍如何把使用 JBuilder4 开发的 EJB 组件分发到 WebLogic 服务器上，笔者认为，WebLogic 服务器是最好的商用 J2EE 运行环境，也是最好的 EJB Server/Container，这是有一定根据的，例如：WebLogic 支持最新的 EJB 技术规范，WebLogic 的市场占有率最高等。我们之所以首先介绍如何把 EJB 组件分发到 IAS 服务器上面去，那是因为 JBuilder4 和 IAS 都是 Borland 公司的产品，利用 JBuilder4 开发的 EJB 组件最容易发布到 IAS 服务器中，并不是 IAS 服务器最适合做 EJB Server/Container。

现在让我们开始吧。

安装 WebLogic 服务器

第一步应该安装 WebLogic 服务器，笔者推荐使用 WebLogic 5.1/6.0，你可以到 www.weblogic.com 去下载一个试用版。然后配置好 WebLogic 服务器，使它支持 JSP。关于如何配置 WebLogic 服务器的方法，请参考本书的附录 3。

封装 EJB 组件

第二步是把 EJB 组件所用到的 class 文件都打包到一个 jar 文件中，如果我们是使用 JBuilder4 开发的 EJB 组件，那么在编译 EJB 工程的时候，JBuilder4 会自动把所有相关的文件都打包到一个 jar 文件中，这个 jar 文件的名字就是工程的名字。

转移 jar 文件

把 jar 文件从 JBuilder4 的开发目录转移到 WebLogic 服务器安装目录的 myserver 文件夹的根目录下面，这样做的目的是为了便于管理 EJB 组件，如果略过这一步也可以。

配置 EJB Deployer

从开始菜单运行 WebLogic 附带的 EJB Deployer，如图 2.18 所示。

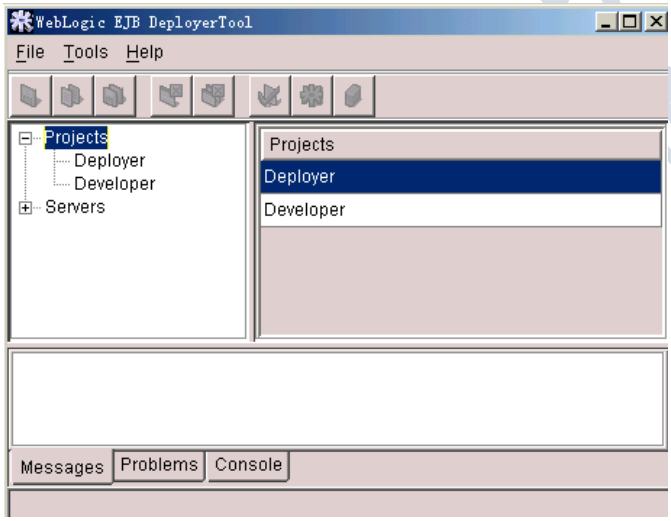


图 2.18　EJB Deployer

图 2.18 显示了 EJB Deployer 的主运行界面，在左边的视图中，单击 Server，接着在右边的窗口里，单击 Add 按钮，右边的窗口将如图 2.19 所示。

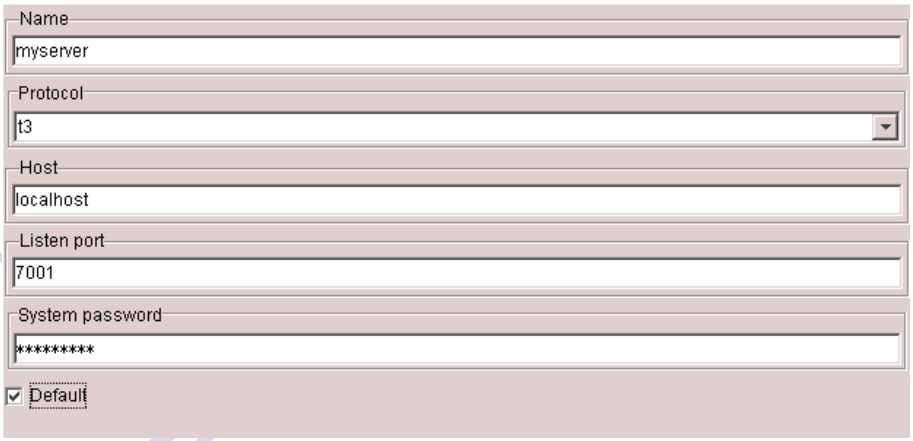


图 2.19　WebLogic 服务器的设置

我们需要在如图 2.19 所示的窗口中设置 WebLogic 服务器的信息，前面 4 项都是预先设定的，不需要改动，第 4 项 System password 需要我们自己输入，然后在 Default 前面的框中打上勾。这样我们就把 WebLogic 服务器设为缺省的 EJB Container。

接下来请选择 Tools→Preferences...，会出现如图 2.20 所示的对话框，我们需要在这个对话框中指定 ejbc.exe 程序的属性，也就是指定 Java 编译器——javac.exe 程序所在的路径，然后单击 OK 按钮，关闭这个对话框，回到 EJB Deployer 的主界面。

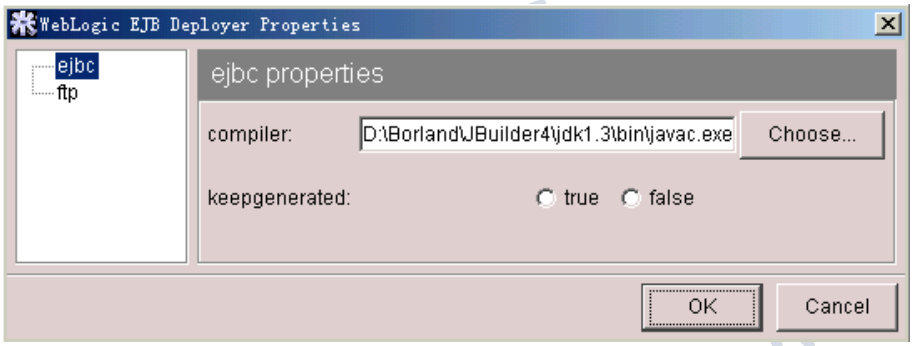


图 2.20　WebLogic EJB Deployer Properties 对话框

编辑 EJB 部署描述符

我们在 EJB Deployer 的主运行界面中，选择 File→Open...，打开在上文第二步中封装好的 jar 文件(在这个例子中，这个 jar 文件就是 HelloWorld.jar，也就是我们在 2.3、2.4 节中开发的 HelloWorld EJB 组件所对应的 jar 文件)，EJB Deployer 将出现如图 2.21 所示的对话框，这时，请你稍微耐心一点，这可能需要花比较长的时间。



图 2.21　WebLogic EJB DeployTool 对话框

当 EJB Deployer 把 jar 文件完全载入以后，如图 2.21 所示的对话框将会消失，同时在 EJB Deployer 主运行界面的视图区(左侧)，会出现此 EJB 组件的树状视图，我们可以把这个树状视图一级一级展开，然后在右侧的窗口编辑该 EJB 组件的部署描述符。

在左侧的视图区，单击 Files 节点，右边的窗口将显示刚刚载入的 jar 文件中都含有哪些文件，如图 2.22 所示。

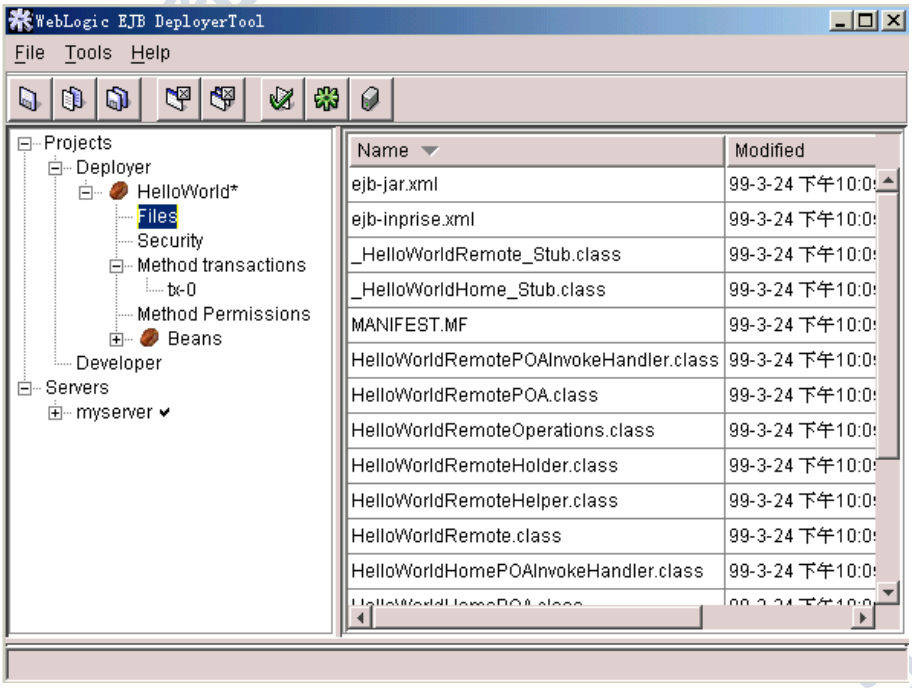


图 2.22　查看 EJB jar 文件所包含的文件列表

实际上，我们在 JBuilder4 中开发 HelloWorld EJB 组件时，已经编辑了它的 ejb-jar.xml 文件，ejb-jar.xml 文件保存了 EJB 组件部署描述符的大部分信息，所以在 EJB Deployer 中，我们不用对部署描述符做太多的修改(对部署描述符所做的任何改动都将修改 ejb-jar.xml 文件)，不过有一点，我们必须指定 HelloWorld EJB 组件所使用的 JNDI 服务名，EJB 组件的客户端程序将凭借这个服务名定位 EJB 的 Home Interface，进而调用 EJB 组件的商业方法。那么如何设定 HelloWorld EJB 组件在 WebLogic 服务器中的 JNDI 服务名呢?在 EJB Deployer

的主运行界面的左侧视图区，展开 Beans 节点，点击 HelloWorld，在 EJB Deployer 主窗口的右侧，出现了如图 2.23 所示的窗口，选择 Classes Tab，在 Home JNDI name 一栏下面的文本框中，填入 HelloWorld，这就是 HelloWorld EJB 的 JNDI 服务名，现在我们可以保存所做的工作了。

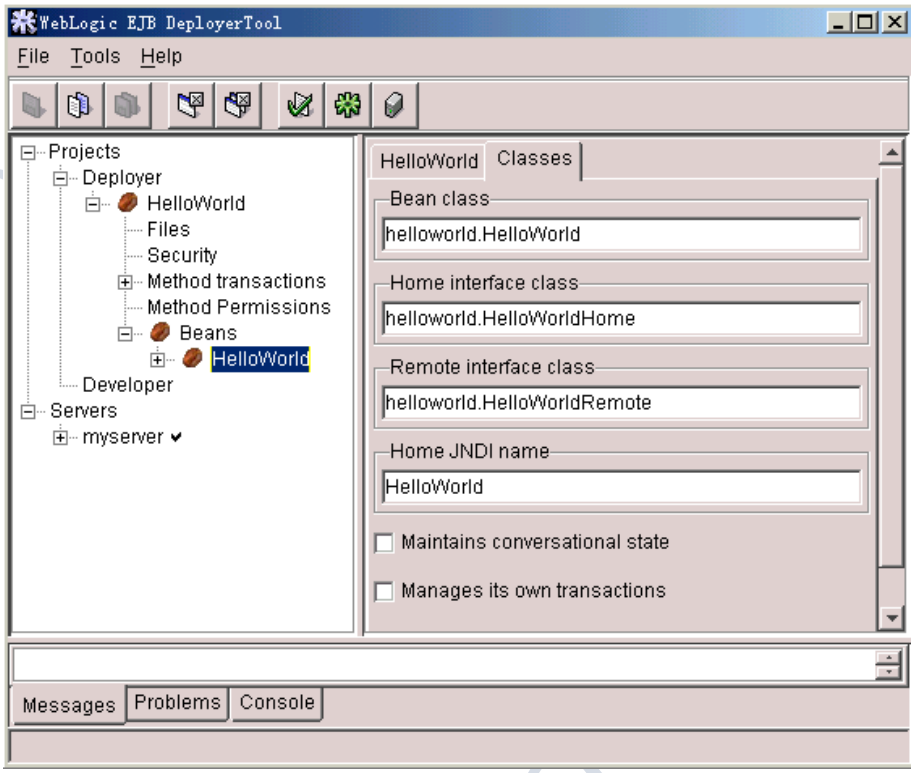


图 2.23　设定 EJB 组件的 JNDI 服务名

创建 EJB Container

单击工具栏中的 Generate container…按钮，就是那一个类似于齿轮模样的按钮，创建 HelloWorld EJB 组件的 EJB Container，在这一步中，EJB Deployer 将调用 ejbc.exe 程序，产生该 EJB 组件的 stub 类(存根类)和 skeleton 类(框架类)，也就是 Home Object 类和 EJB Object 类，EJB Deployer 还将调用 javac.exe 程序把它们编译为 class 文件，并加入到前面的 jar 文件中，在这一步中，EJB Deployer 还会自动产生 weblogic-ejb-jar.xml 文件，并且也会打包到 jar 文件中。EJB Deployer 在执行这一步的时候，会消耗很多的系统资源，要花大概一分多钟的时间，在此期间，会出现如图 2.24 所示的窗口，并且还可能弹出几个 MS-DOS 窗口，这是 EJB Deployer 在调用 ejbc.exe 程序和 javac.exe 程序，读者不要去管它，也不要随便关闭这些 MS-DOS 窗口，它们会自动关闭的。当创建 HelloWorld EJB 的 EJB Container 完成以后，会出现如图 2.25 所示的对话框，单击 Close 按钮，把它关闭掉，返回 EJB Deployer 的主界面。

我们虽然已经创建了 HelloWorld EJB 的 EJB Container，但是 HelloWorld EJB 组件仍然没有被发布到 WebLogic 服务器中，我们也可以利用 EJB Deployer 临时把 HelloWorld EJB

发布到 WebLogic 服务器中，这只需要单击 Deploy…按钮就可以了，但这只是临时的发布，如果 WebLogic 服务器关闭了，再次启动的时候，它不会自动把 HelloWorld EJB 组件载入的。如果我们希望 WebLogic 服务器在启动的时候自动载入 HelloWorld EJB，必须修改 WebLogic 服务器的配置文件。



图 2.24　Generator EJB Container



图 2.25　创建 EJB Object /HomeObject 成功

修改 weblogic.properties 文件

这一步，我们将修改 weblogic.properties 文件，使得 WebLogic 服务器能够在启动的时候自动把 HelloWorld EJB 载入，当作一个 JNDI 服务发布，如果 WebLogic 服务器是处于运行状态，那么应该首先关闭 WebLogic 服务器，然后使用记事本打开 WebLogic 安装目录(根目录)下的 weblogic.properties 文件，找到下面的代码行。

```
#weblogic.ejb.deploy=\
#          D:/weblogic/myserver/ejb_basic_beanManaged.jar，\
#          D:/weblogic/myserver/ejb_basic_containerManaged.jar，\
#          D:/weblogic/myserver/ejb_basic_statefulSession.jar，\
#          D:/weblogic/myserver/ejb_basic_statelessSession.jar，\
#          D:/weblogic/myserver/ejb_extensions_finderEnumeration.jar，\
#          D:/weblogic/myserver/ejb_extensions_readMostly.jar，\
#          D:/weblogic/myserver/ejb_subclass.jar，\
#          D:/weblogic/myserver/jolt_ejb_bankapp.jar
```

然后在这些代码行的后面，添加上下面的代码行(请注意/和\)：

```
weblogic.ejb.deploy=\
D:/weblogic/myserver/HelloWorld.jar，\
```

在上面的代码中，HelloWorld.jar 文件就是我们使用 JBuilder4 所开发的 HelloWorld EJB 的 jar 文件，而且这个文件已经从 JBuilder4 的开发目录转移到 WebLogic 的 myserver 文件夹下面，还使用 EJB Deployer 程序处理过。保存好 weblogic.properties 文件，然后启动 WebLogic 服务器。当 WebLogic 服务器的运行窗口(wlserver)出现类似于下面的信息时，那么恭喜你，这说明你已经成功地把 HelloWorld EJB 组件发布到 WebLogic 服务器中，WebLogic 服务器已经自动地把它载入，当作一个 JNDI 服务发布，而且 WebLogic 服务器也成功启动了。

```
星期四 三月 25 16:15:40 GMT+08:00 1999:<I> <EJB> 1 EJB jar files loaded，containing 1 EJBs
星期四 三月 25 16:15:40 GMT+08:00 1999:<I> <EJB> 1 deployed，0 failed to deploy.
星期四 三月 25 16:15:41 GMT+08:00 1999:<I> <ZAC> ZAC ACLs initialized
星期四 三月 25 16:15:41 GMT+08:00 1999:<I> <ZAC> ZAC packages stored in local di
rectory exports
星期四 三月 25 16:15:41 GMT+08:00 1999:<I> <ListenThread> Listening on port: 7001
星期四 三月 25 16:15:41 GMT+08:00 1999:<A> <Posix Performance Pack> Could not in
itialize POSIX Performance Pack.
星期四 三月 25 16:15:41 GMT+08:00 1999:<E> <Performance Pack> Unable to load per
formance pack，using Java I/O.
星期四 三月 25 16:15:42 GMT+08:00 1999:<I> <WebLogicServer> WebLogic Server star
ted
```

2.5.7　编写 JSP 程序访问 EJB 服务

既然 HelloWorld EJB 组件已经成功地发布到 WebLogic 服务器中，那么我们怎么才能够在 JSP 程序中调用这个 EJB 组件的商业方法呢?也就是如何编写一个基于 JSP 程序的 EJB 客户端呢?在上文中，我们虽然已经实现了 EJB 的客户端程序，不过那是一个 Java Application，不能够用于 JSP 程序中。本书第一部分的主旨，一方面在于介绍 J2EE 应用的开发技术，另一方面则是这些 J2EE 技术如何与 JSP 技术结合起来，开发功能强大的应用系统。因为 J2EE 技术比较难掌握，而利用 JSP 技术开发 J2EE 应用程序的客户端就相对容易得多，所以我们在第一方面花费了大量的笔墨，而第二方面就介绍比较少，这不是说第二方面不重要，实际上，两方面都十分重要，都不可偏颇 (毕竟，我们开发出来的 EJB 组件，主要还是在 JSP 程序中调用)，只不过第一方面的知识实在太多，太难，而我们以前又很少接触罢了。

下面我们就来介绍该如何编写 JSP 程序访问 EJB 组件。请看下面的程序清单 2.14。

程序清单 2.14

```
<%--
File Name:ejb.jsp
Author:fancy
Date:2001.5.10
Note:ejb client
--%>
```

```
<%@ page import=" java.rmi.RemoteException"%>
<%@ page import=" java.util.Properties"%>

<%@ page import=" javax.ejb.CreateException"%>
<%@ page import=" javax.ejb.RemoveException"%>
<%@ page import=" javax.naming.Context"%>
<%@ page import=" javax.naming.InitialContext"%>
<%@ page import=" javax.naming.NamingException"%>
<%@ page import=" javax.rmi.PortableRemoteObject"%>
<%@ page import="helloworld.*" %>

<%
HelloWorldHome testHome = null;
HelloWorldRemote testRemote = null;
Context ctx = new InitialContext();
//look up jndi name
Object ref = ctx.lookup("t3://localhost:7001/HelloWorld");
//cast to Home interface
testHome = (HelloWorldHome) PortableRemoteObject.narrow(ref，HelloWorldHome.class);
testRemote=testHome.create();
String author=testRemote.getMessage();
out.println(author);
%>
```

读者对照程序清单 2.14 和程序清单 2.13，不难发现这两个程序的流程几乎一样，都是查找 Home Interface，获取 Home Interface 以后，创建 Remote Interface，最后利用 Remote Interface，调用 EJB 的商业方法，把结果输出，程序就差不多了，实际上，基于 JSP 的 EJB 客户端与基于 Java Application 的 EJB 客户端在本质、原理上没有任何区别，我们可以很容易地把一个基于 Java Application 的 EJB 客户端改写为 JSP 程序。

在程序清单 2.14 中，读者需要注意两个问题。

（1）需要导入 helloworld 包，请看下面的代码：

```
<%@ page import="helloworld.*" %>
```

我们在编写 HelloWorld EJB 的时候，所有的类都是属于 helloworld 包的，如果我们不把这个包导入 JSP 程序中，那么 WebLogic 的 JSP 引擎将因为无法识别 HelloWorldHome 接口、HelloWorldRemote 接口而报错。helloworld 包其实就是 HelloWorld.jar 文件中包含的那些 class 文件，我们之所以把 HelloWorld.jar 文件从 JBuilder4 的开发目录转移到 WebLogic 的 myserver 文件夹下面，其中一个很重要的原因就是便于 JSP 引擎找到 helloworld 包中所定义的那些类/接口。

（2）请看下面的代码，在定位 Home 接口时，必须使用这样的格式。

```
Object ref = ctx.lookup("t3://localhost:7001/HelloWorld");
```

如果我们使用 WebLogic 作为 EJB Server，那么 EJB 的 JNDI 服务名会是这样的格式：

t3://host:port/JNDIName

如果我们使用 IAS 作为 EJB Server，那么 EJB 的 JNDI 服务名会是另外的格式，如：

ias://host/ejbcontainer/JNDIName

host 代表主机的名字，可以是本地机，也可以是远程服务器，这没有任何限制，port 是服务器的服务端口名，对于 WebLogic 服务器，这个值一般是 7001，JNDIName 就是 JNDI 服务的名字，我们可以通过编辑部署描述符的方式来指定它，例如 HelloWorld。

程序清单 2.17 的运行效果如图 2.26 所示。

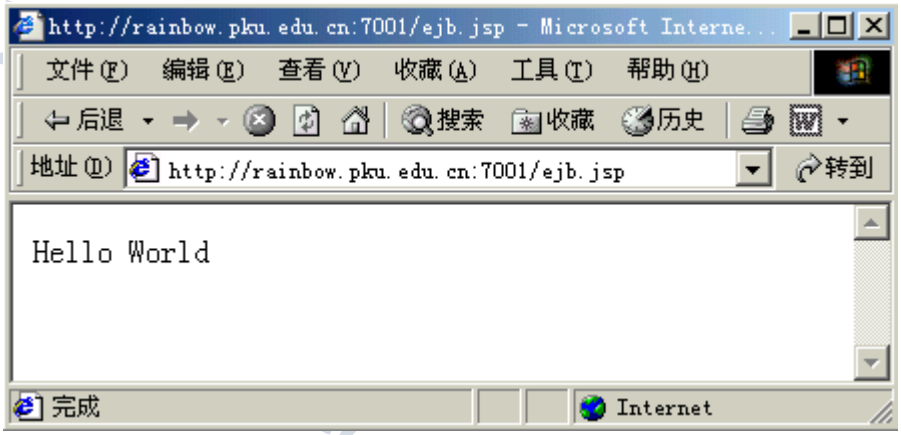


图 2.26　ejb.jsp

2.6　本 章 小 结

本章重点介绍的内容有：EJB 的体系结构、EJB 开发环境的配置、如何使用 JBuilder 4.0+IAS 4.1 开发会话 EJB、发布 EJB 服务，这主要针对 IAS 服务器和 WebLogic 服务器、编写 EJB 服务的客户端、调用 EJB 组件的商业方法，不知道读者都掌握了没有，应该说，本章是本书最难明白的章节之一，光看一次是很难完全领会的，只有按照书中的方法，一步一步自己去试验，才有可能掌握好 EJB 技术。

本章所介绍的 EJB 技术还比较基本，还没有涉及到实体 EJB 的开发方法，在下一章中，我们将接着介绍如何开发 CMP 模式、BMP 模式的实体 EJB。

第 3 章　EJB 技术进阶

在上一章中，我们已经介绍了 EJB 的基础知识，还详细描述了 Session EJB 的开发、运行和测试技术，在本章，我们将继续介绍 EJB 的开发技术，但是重点将转移到实体 EJB 的开发技术上。本章需要重点掌握的内容有：

- CMP 类型的实体 EJB 的开发
- BMP 类型的实体 EJB 的开发
- 利用 BMP 类型的实体 EJB 封装数据源

3.1　实体 EJB 的开发技术之一——CMP EJB

在上一章中我们已经提到过：EJB 技术规范支持两种类型的 EJB 对象，它们分别是 Session EJB(会话 EJB)和 Entity EJB(实体 EJB)。第 3 章所开发的 EJB 组件属于会话 EJB。

Session EJB(会话 EJB)是短暂存在的对象，它同样运行在服务器端，并执行一些应用逻辑处理。它的实例对象由客户端应用程序建立，并仅能够被创建它的应用程序所使用，其保存的数据需要开发者编写程序来管理。Session EJB 支持事务属性，但是，当系统因为某种特殊的不可预见的原因崩溃或者关闭后，Session EJB 的状态、数据不会再被系统恢复。Session EJB 的这种特性十分类似于 Page Scope 类型的 JavaBeans 组件对象。

Entity EJB(实体 EJB)是持久运行的 EJB 对象，由某个客户端应用程序创建，但是可以被其他对象或者其他的应用程序调用。与 Session EJB 不同，Entity EJB 必须在建立时制定一个惟一的标识，并提供相应的机制允许客户应用程序根据 Entity EJB 标识来定位 Entity EJB 的实例对象。多个用户可以并发访问 Entity EJB，事务之间的协调工作由 EJB Container 来完成。读者应当注意，Session EJB 只能够被创建它的应用程序所调用，所以不存在事务协调的问题。Entity EJB 支持事务处理，当系统关闭时，也可以恢复关闭以前的状态，包括 EJB 对象所包含的数据在内。EJB 规范中定义了两种管理 Entity EJB 的持久性模型，即 Beans Managed Persistence(Beans 自管理方式，简称为 BMP)及 Container Managed Persistence (容器代管理方式，简称为 CMP)。BMP 管理方式是由 EJB 对象自己来管理持久性，它需要 EJB 开发者来编写数据库或应用程序的处理逻辑，并加入到 EJB 对象类(EJBObject Class)的 ejbCreate()、ejbRemove()、ejbFind()、ejbLoad()和 ejbStore()、Finder 等方法中。CMP 管理方式是将 EJB 持久性管理交给 EJB Container 来完成，开发者不用过多操心。开发者一般要在 EJB 对象的 Deployment Descriptor(部署描述符)中的 Container Managed 属性中指定 EJB 实例对象的持久性作用域。当使用 CMP 模式时，开发者不需要知道 Entity EJB 如何存取数据源，开发者也很少需要参与复杂、烦琐的编码工作，不过 CMP 模式的实体 EJB 功能比较单一，使用起来不灵活，扩展起来比较难，而采用 BMP 模式，那就需要开发者事必躬亲，对每一个细小的问题都必须考虑周到，但是 BMP 模式的实体 EJB 开发起来很灵活，开发者有很大的自由度，可以定制复杂的处理逻辑，扩展起来也比较容易。在实际的开发过程

中，如果需要使用实体 EJB，那么比较单一、固定的任务可以交给 CMP 模式的实体 EJB 完成，而灵活、复杂的系统最好还是采用 BMP 模式的实体 EJB。读者需要注意的是，在会话 EJB 中没有 CMP 与 BMP 模式之分，只有 Stateless 与 Stateful 之分。

下面我们首先介绍 CMP Entity EJB 的开发方法。

3.1.1　CMP EJB 简介

CMP 模式的实体 EJB 非常简单，因为有很多的细节问题我们都不需要去关心，EJB Container 会自动实现它们，例如 EJB 类中定义的 ejbCreate()、ejbRemove()、ejbPassivate()、ejbLoad()、ejbStore()、getXXX()、setXXX()等方法，还有 Home 接口中声明的 findByPrimaryKey()、findAll()等方法，我们只需要简单地声明它们，或者简单地返回，具体的实现要依靠 EJB Container，还有一个更为重要的问题，那就是——DataSource，在 CMP 模式下面，我们根本就不需要理会 DataSource 在哪里，DataSource 如何与实体 EJB 绑定在一起，实体 EJB 如何存取数据，如何 Load，如何 Store，如何 Create，还有事务的问题，还有持久性的问题，还有并发执行的问题等等，所有的问题都归 EJB Container 管理，开发者不用费半点心思，我们只要把 Home Interface、Remote Interface、EJB 类、部署描述符做出来，利用 EJB Deploy 工具就能够自动生成完备的、可以完成上述任务的 EJB Container，怎么样，是不是很轻松，很简单呢?下面我们就来看看如何利用 JBuilder4 开发一个 CMP 模式的实体 EJB。

3.1.2　创建 EJB 工程

我们还是按照上一章的方法，首先利用 JBuilder4 建立一个 EJB 的框架架构，然后在这个基础之上进行第二次开发。在开发 CMP Entity EJB 之前，我们同样要做好配置开发环境的工作，具体来说，就是配置好 IAS 服务器和 JBuilder4，关于配置的方法，读者可以参考第 2 章的介绍，因为我们在上一章已经做好了配置工作，所以在这里就把这一步给掠过了，不过读者最好还是从头检查一遍开发环境，开发过程中出现的问题多半是由于开发环境没有配置好而造成的。

配置好开发环境以后，我们还有一步额外的工作要做，那就是配置 JDBC 数据源，实体 EJB 的主要作用就是映射数据库的结构，并提供一个间接访问数据库中数据的方法，所以我们在开发实体 EJB 之前，必须首先指定 JDBC 数据源。

那么如何指定 JDBC 数据源呢?请按照下面的步骤去做：

（1）创建一个数据库，我们所使用的数据库系统是 MS SQL Server 7.0，至于数据库的结构，请参考附录 4，ODBC 数据源的名称为 test。

（2）启动 JBuilder，选择 Tools→JDBC Explorer，将出现如图 3.1 所示的窗口。

（3）在 JDBC Explorer 中，选择 File→New…，出现一个新窗口，在这个窗口中，我们指定 JDBC Driver 为 sun.jdbc.odbc.JdbcOdbcDriver，读者可以改为其他的 JDBC Driver，JDBC URL 为 jdbc:odbc:test(因为 ODBC 数据源的名称为 test)，读者也可以改为其他的 JDBC URL。结果如图 3.2 所示 。

（4）在图 3.2 中，单击 OK，出现了如图 3.3 所示的窗口，我们输入的用户名为 sa，密码为空，然后单击 OK 按钮，那么 JDBC Explorer 程序将试图连接到数据库中。

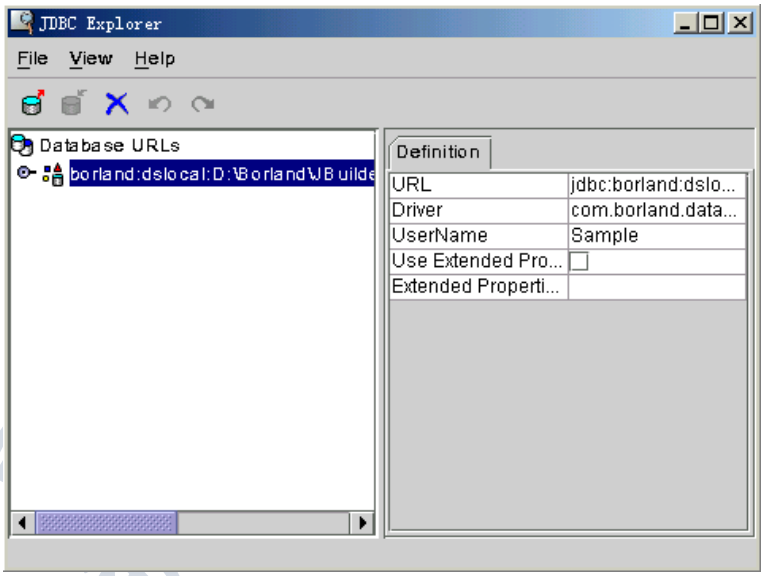


图 3.1　JDBC Explorer 的窗口

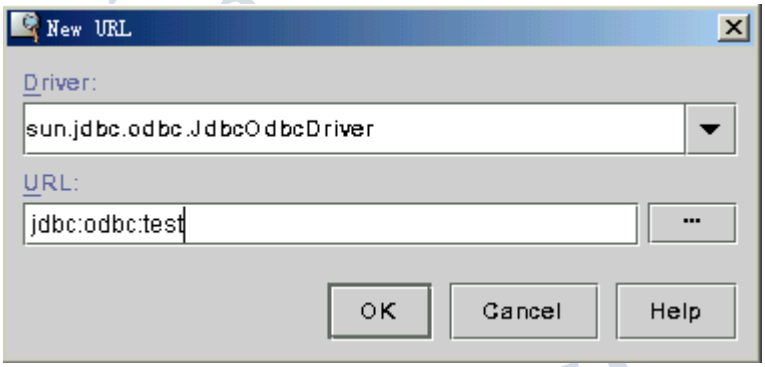


图 3.2　创建新的 JDBC DataSource

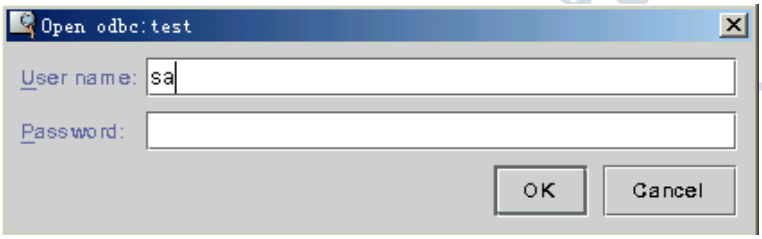


图 3.3　输入数据库的用户名和密码

（5）在图 3.3 中，单击 OK 按钮之后，如果 JDBC Explorer 连接数据库系统成功，那么 JDBC Explorer 的主窗口将如图 3.4 所示，我们应该逐个展看窗口左侧的视图，明了数据库的字段结构，同时也可以在右侧的窗口中输入 SQL 语句，并执行它，改变数据库的结构或者更新里面的数据。

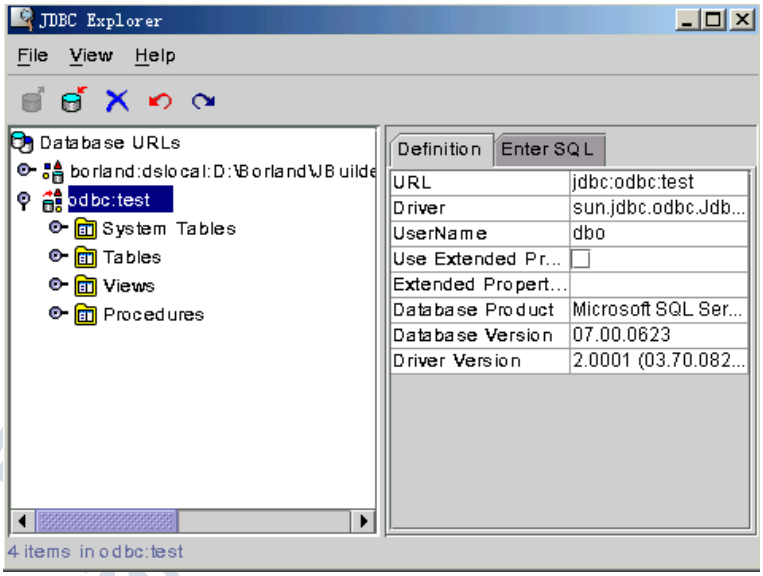


图 3.4　利用 JDBC Explorer 查看数据库的结构

（6）关闭 JDBC Explorer，这时候程序提示你是否保存所做的改变，你应该选择保存，要不然你所做的一切工作都白费了，又得从头再来。

配置好 JDBC DataSource 以后，我们的下一步就是创建一个实体 EJB 工程，读者请跟着下面的步骤一步一步尝试，一定会成功的。

（7）在 JBuilder4 的主运行界面中，选择 File→New…→Enterprise Tab→Empty EJB Group，JBuilder4 会自动创建一个新的 Project，我们把这个 Project 的名字指定为 JDBCTest，如图 3.5 所示。

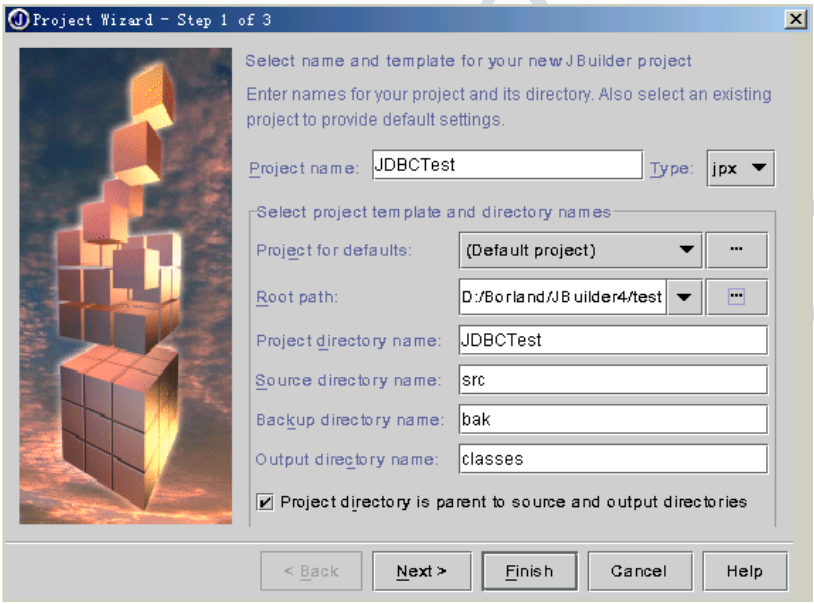


图 3.5　JBuilder4 的 Project Wizard

（8）创建好 JDBCTest Project 以后，Empty EJB Group Wizard 会自动运行，我们指定这个 EJB Group 的名字为 JDBCTest，包含此 EJB 组件的 jar 文件名为 JDBCTest.jar，如图 3.6 所示。

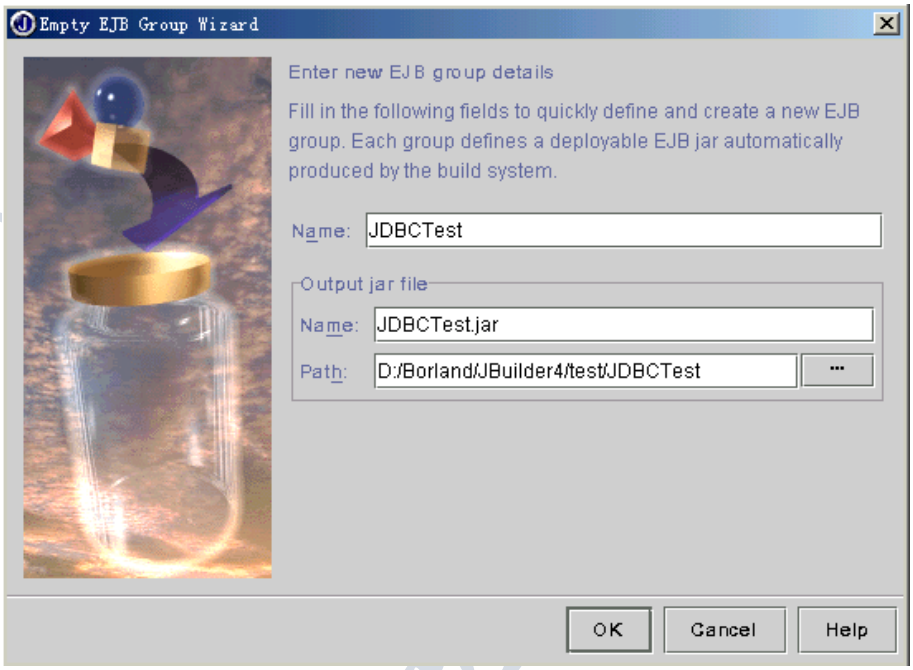


图 3.6　JBuilder4 的 Empty EJB Group Wizard

（9）创建好 EJB Group 以后，下面该创建一个 Entity EJB 了，在 JBuilder 的主运行界面中，选择 File→New…→Enterprise Tab→EJB Entity Bean Modeler，然后单击 OK 按钮，如图 3.7 所示。

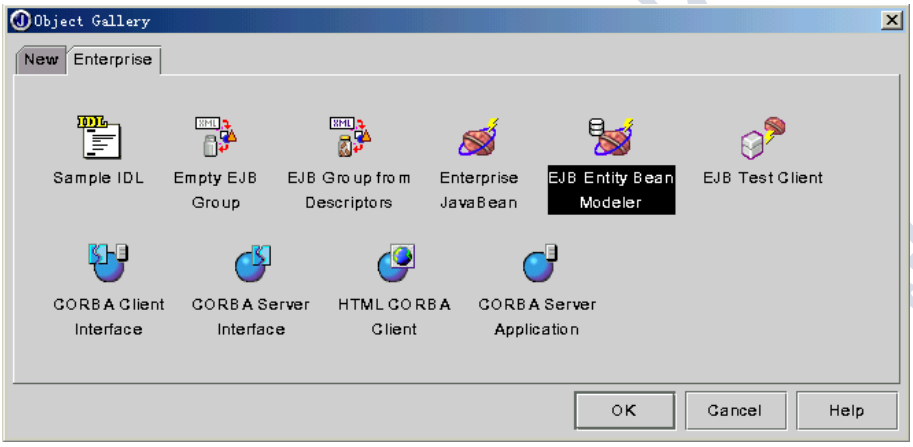


图 3.7　创建 EJB Entity Bean Modeler

（10）在图 3.7 的窗口中，单击 OK 按钮之后，出现如图 3.8 所示的对话框，确保 Available EJB groups 下拉列表中选的是 JDBCTest.ejbgrp，然后单击 Next 按钮。

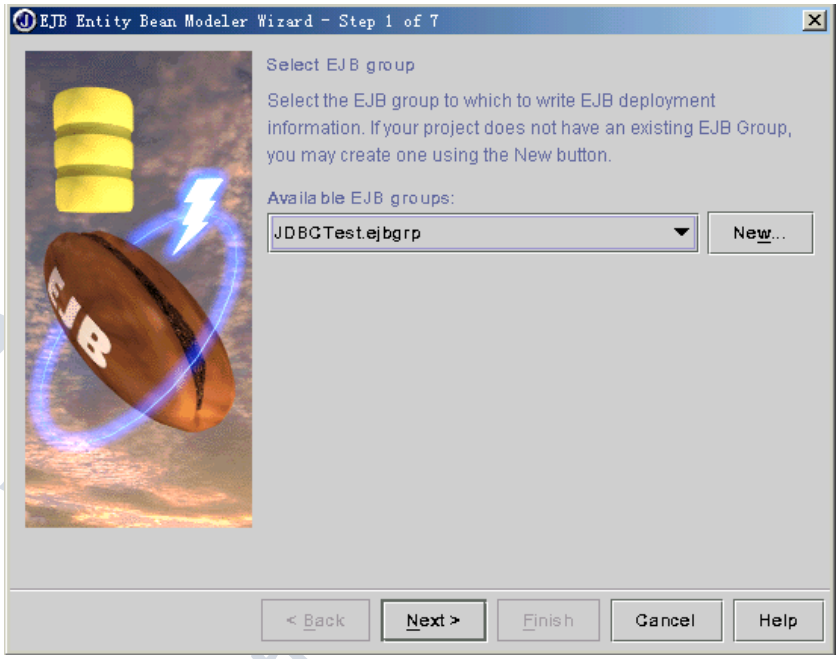


图 3.8　JBuilder4 的 EJB Entity Bean Modeler Wizard

（11）接着上一步，将出现如图 3.9 所示的对话框。

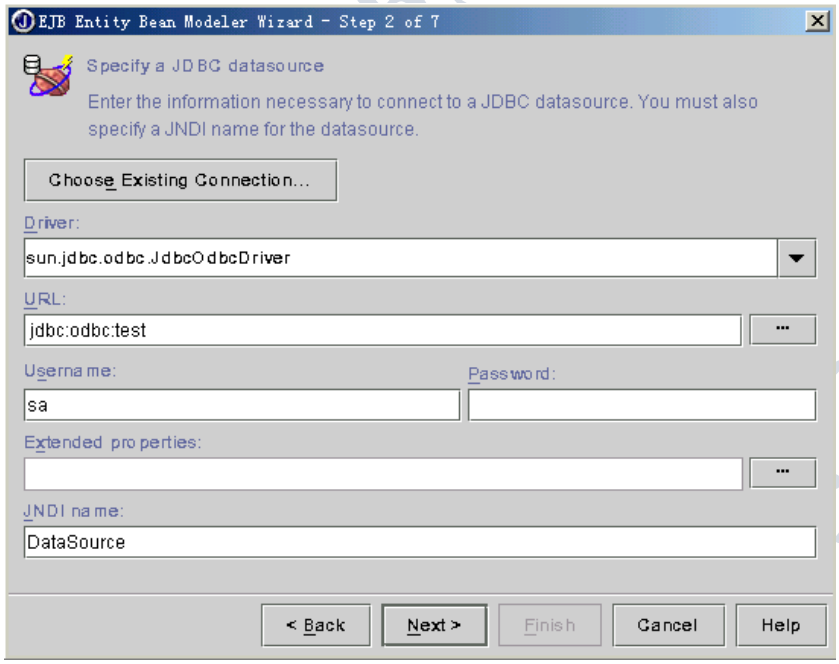


图 3.9　设定 JDBC DataSource

在图 3.9 中，我们需要点击 Choose Existing Connection…按钮，选择已经存在的 JDBC 数据源，也就是一开始就设定的那一个 JDBC 数据源——jdbc:odbc:test。然后在 Username

处填入 sa，让 Password 空着，JNDI name 保持 DataSource 不变。单击 Next 按钮，进入下一步。

（12）在这一步中，设定 Entity EJB 需要映射的 Table，JBuilder4 会根据 Table 的字段名字，自动产生 setXXX()方法和 getXXX()方法。jdbc:odbc:test 数据源所对应的数据库一共有 4 个用户表，分别是 goods 表、tbuser 表、tborder 表、dtproperties 表，我们选定 goods 表作为 Entity EJB 映射的 Table，方法是在左侧的窗口中，选定需要映射的 Table，然后单击“>”按钮，那么这个 Table 的名字就会移到右边的窗口中，这样就可以了，如图 3.10 所示。

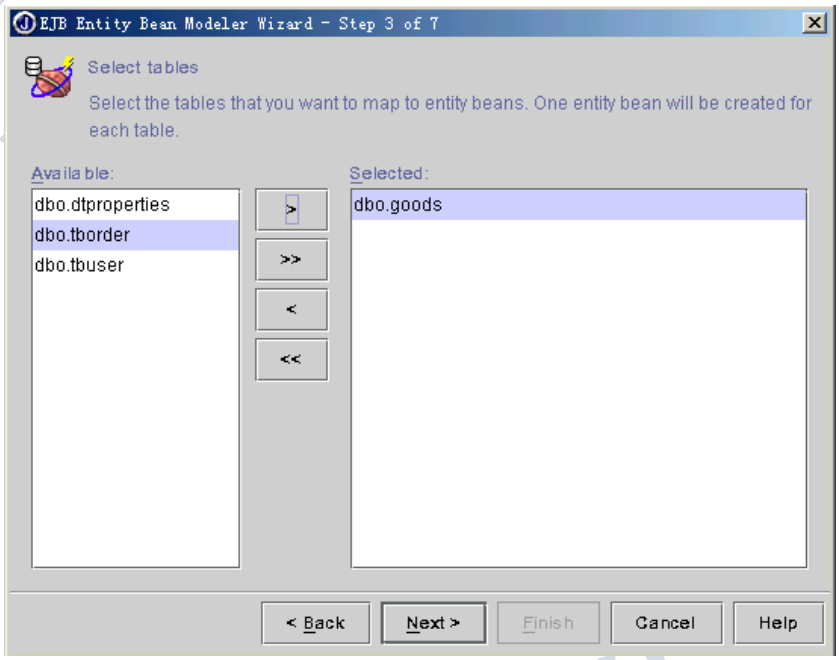


图 3.10　选定 Entity EJB 所映射的 Table

（13）在图 3.10 的对话框中，单击 Next 按钮，进入一个新的对话框，不要做任何改动，再次单击 Next 按钮，将显示如图 3.11 所示的对话框，在图 3.11 的对话框中，我们将设定 Entity EJB 的 Primary Key，亦即主键。在 Entity EJB 的 Home Interface 中，声明了 findPrimaryKey()方法，利用这个方法，可以获取特定的 Remote 对象。我们所说的主键，其实就是 Table 的某一个或者某几个特定的字段，但是这些字段的值必须是唯一的，亦即在数据库的记录中，不能够有相同的值，如果某一个字段被设为主键，那么 JBuilder4 只会为它自动产生 getXXX()方法，而不会产生 setXXX()方法，因为主键是标识 Entity EJB 的唯一的标志，当然不能够被改动，而且主键的值必须是唯一的。不被设定为主键的字段既可以有 setXXX()方法，也可以有 getXXX()方法。其实 Entity EJB 的主键与数据库的主键有一定的相似之处，Entity EJB 就相当于记录集中的一个行纪录。

在图 3.11 的对话框中，我们选定字段 id 作为此 Entity EJB 的主键，这只需要在 id 字段前面的那一个小框中打勾即可。选中 id 字段以后，单击 Next 按钮，进入下一步。

（14）在图 3.11 中，单击 Next 按钮以后，会进入如图 3.12 所示的对话框，JBuilder4

根据 Table 的名字(goods)自动生成这个 Entity EJB 的 Home Interface、Remote Interface、EJB Class、JNDI 服务的名字，我们可以修改这些名字，把它们改成希望的值，在这个例子中，Bean 的名字为 Goods，JNDI 服务名为 Goods，Home Interface 的名字为 jdbctest.GoodsHome，Remote Interface 的名字为 jdbctest.GoodsRemote，EJB Class 的名字为 GoodsBean，主键的类型为 int，也就是 goods Table 的 id 字段。一切都设置好以后，清单击 Next 按钮，进入下一步。

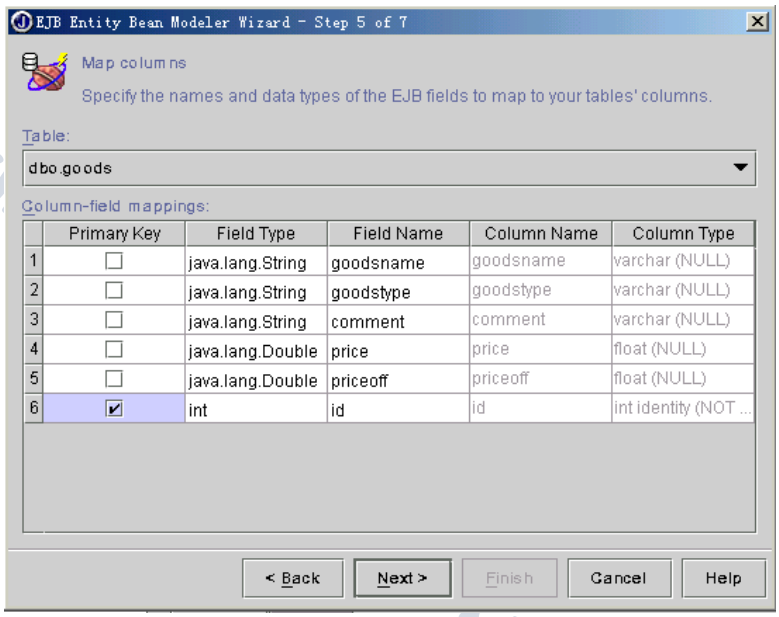


图 3.11　选定 Entity EJB 的主键

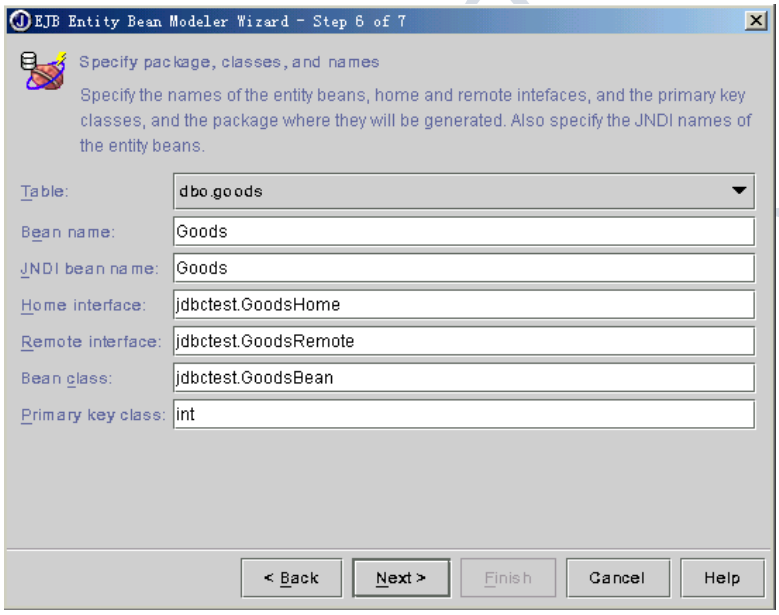


图 3.12　设定 Entity EJB 的 Home、Remote Interface 和 EJB Class 的名字

（15）在图 3.12 的对话框中，单击 Next 按钮，将进入如图 3.13 所示的对话框，在该对话框中，我们将设定 Goods EJB 的模式是 CMP 还是 BMP。

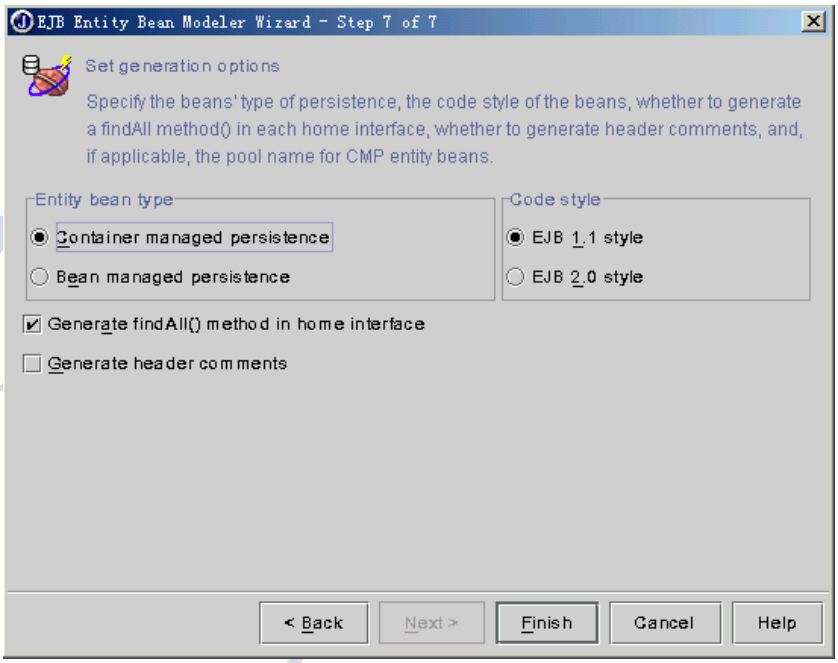


图 3.13　设定 Entity EJB 的模式

如图 3.13 所示，我们选中了 Container managed persistence，还选中了 Generate findAll() method in home interface，这样 JBuilder4 将为我们自动生成一个 Entity EJB 的运行框架，这个 Entity EJB 的模式是 CMP，亦即容器管理模式，JBuilder4 还将在 Home Interface 中声明一个 findAll()方法，利用这个方法，可以获取在 EJB Container 中所有可用的同类 Remote Interface。

单击 Finish 按钮，这样就算搭建起一个 Entity EJB 的框架了。实际上，我们不用做任何修改，只需要编译这个 Project，Goods EJB 就能够成功运行了。

下面，我们将简要地分析一下究竟 JBuilder4 都给我们生成了哪些代码，Entity EJB 和 Session EJB 都有些什么不同。

3.1.3　Home Interface 和 Remote Interface

这一小节中，我们分析 Goods EJB 的 Home Interface 和 Remote Interface，首先是 Home Interface，请看程序清单 3.1。

程序清单 3.1

```
//File Name: GoodsHome.java
//Author:fancy
//Date:2001.5
//Note:the Home Interface
package jdbctest;
```

```
import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface GoodsHome extends EJBHome
{
    public GoodsRemote create(String goodsname，String goodstype，String comment，
        Double price，Double priceoff，int id) throws RemoteException，CreateException;
    public GoodsRemote create(int id) throws RemoteException，CreateException;
    public GoodsRemote findByPrimaryKey(int primaryKey) throws RemoteException，
        FinderException;
    public Collection findAll() throws RemoteException，FinderException;
}
```

在程序清单 3.1 中，定义了 Goods EJB 的 Home Interface，Home Interface 必须继承 EJBHome 接口，这一点，无论是 Session EJB 还是 Entity EJB 都不能够违背。

GoodsHome 接口中声明了 4 个方法，第一个 create()方法的返回值是一个 GoodsRemote 对象，调用这个方法，就相当于往数据库中插入了一个新的行纪录。第二个 create()方法也返回一个 GoodsRemote 对象，也相当于往数据库中插入了一个新的行纪录，不过这行纪录只有主键被赋值，其余的字段都是空值。findByPrimaryKey()方法可以根据指定的主键的值，返回特定的 GoodsRemote 对象，这个 GoodsRemote 对象代表了某个特定的行纪录。最后一个方法是 findAll()方法，findAll()方法的返回值是 Collection 对象，这个 Collection 对象包含的是 EJB Container 中所有可用的同类的 Remote 接口对象，相当于一个完整的记录集，Entity EJB 所映射的 Table 有多少行，那么此 Collention 对象就包含有多少个元素，EJB Container 中就有多少个可用的同类的 Remote 接口对象。利用 findPrimaryKey()方法和 findAll()方法，就可以实现检索数据库的功能了。

下面我们来看看 Remote Interface 的代码，请看程序清单 3.2。

程序清单 3. 2

```
//File Name: GoodsRemote.java
//Author:fancy
//Date:2001.5
//Note:the Remote Interface

package jdbctest;

import java.rmi.*;
import javax.ejb.*;

public interface GoodsRemote extends EJBObject
{
    String getGoodsname() throws RemoteException;
```

```
    void setGoodsname(String goodsname) throws RemoteException;
    String getGoodstype() throws RemoteException;
    void setGoodstype(String goodstype) throws RemoteException;
    String getComment() throws RemoteException;
    void setComment(String comment) throws RemoteException;
    Double getPrice() throws RemoteException;
    void setPrice(Double price) throws RemoteException;
    Double getPriceoff() throws RemoteException;
    void setPriceoff(Double priceoff) throws RemoteException;
    int getId() throws RemoteException;
}
```

Remote Interface 必须继承自 EJBObject 接口，在程序清单 3.2 中，定义了一套 getXXX()方法和 setXXX()方法，这些 getXXX()方法和 setXXX()方法都是 JBuilder4 根据 goods Table 的各个字段的名字和 JDBC 数据类型自动生成的，用于获取与设定 Entity EJB 的属性值，实际上是在存取行纪录的各个字段的值。关于这些方法的用法，读者可以参考程序清单 3.6。

3.1.4　EJB 类

本小节中，我们分析 EJB 类的实现代码，请看程序清单 3.3。

程序清单 3.3

```
//File Name: GoodsBean.java
//Author:fancy
//Date:2001.5
//Note:the Remote Interface

package jdbctest;

import java.rmi.*;
import javax.ejb.*;

public class GoodsBean implements EntityBean
{
    EntityContext entityContext;
    public String goodsname;
    public String goodstype;
    public String comment;
    public Double price;
    public Double priceoff;
    public int id;
    public int ejbCreate(String goodsname，String goodstype，String comment，Double
        price，Double priceoff，int id) throws CreateException
    {
        this.goodsname = goodsname;
        this.goodstype = goodstype;
        this.comment = comment;
```

北京希望电子出版社

```
        this.price = price;
        this.priceoff = priceoff;
        this.id = id;
        return 0;
}
public int ejbCreate(int id) throws CreateException
        {
        return ejbCreate(null，null，null，null，null，id);
}
public void ejbPostCreate(String goodsname，String goodstype，String comment，
        Double price，Double priceoff，int id) throws CreateException
{
}
public void ejbPostCreate(int id) throws CreateException
{
        ejbPostCreate(null，null，null，null，null，id);
        }
public void ejbRemove() throws RemoveException
{
}
public void ejbActivate()
{
}
public void ejbPassivate()
{
}
public void ejbLoad()
{
}
public void ejbStore()
{
}
public void setEntityContext(EntityContext entityContext)
{
        this.entityContext = entityContext;
}
public void unsetEntityContext()
{
        entityContext = null;
}
public String getGoodsname()
{
        return goodsname;
}
public void setGoodsname(String goodsname)
{
```

北京希望电子出版社

```
        this.goodsname = goodsname;
    }
    public String getGoodstype()
    {
        return goodstype;
    }
    public void setGoodstype(String goodstype)
    {
        this.goodstype = goodstype;
    }
    public String getComment()
    {
        return comment;
    }
    public void setComment(String comment)
    {
        this.comment = comment;
    }
    public Double getPrice()
    {
        return price;
    }
    public void setPrice(Double price)
    {
        this.price = price;
    }
    public Double getPriceoff()
    {
        return priceoff;
    }
    public void setPriceoff(Double priceoff)
    {
        this.priceoff = priceoff;
    }
    public int getId()
    {
        return id;
    }
}
```

Entity EJB 的 EJB Class 必须继承自 EntityBean 接口，而 Session EJB 的 EJB Class 必须继承自 SessionBean 接口。在程序清单 3.3 中，读者不难发现，ejbLoad()、ejbStore()、ejbRemove()、ejbActivate()、ejbPassivate()等方法的方法体都是空的，ejbCreate()方法的方法体也只是简单地把结果返回，具体实现的程序逻辑被隐藏起来了，这就是所谓的 CMP 模式，EJB Container 把一切的事情都包办了，我们只要会调用这些方法就可以了，不需要知道它们是如何被实现的。

北京希望电子出版社

在程序清单 3.3 中，还定义了很多 setXXX()方法与 getXXX()方法，这些就是所谓的商业方法，亦即在 Remote Interface 中声明的那些方法。我们可以利用 Home Interface 创建一个 Remote Interface，然后再利用 Remote Inteerface 通过 stub 类和 skelton 类调用这些商业方法。具体的步骤，读者可以参考程序清单 3.6。

3.1.5　部署描述符

下面我们就来介绍 Entity EJB 的部署描述符。在第 2 章的时候我们已经提到过，使用 JBuilder4 开发的 EJB 组件的部署描述符包含在两个 XML 文件中，这两个 XML 文件分别是 ejb-jar.xml 文件和 ejb-inprise.xml 文件，我们先来看看 Goods EJB 的 ejb-jar.xml 文件，请看程序清单 3.4 (ejb-jar.xml)。

程序清单 3.4(ejb-jar.xml)

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

<!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>
    <enterprise-beans>
    <entity>
        <ejb-name>Goods</ejb-name>
        <home>jdbctest.GoodsHome</home>
        <remote>jdbctest.GoodsRemote</remote>
        <ejb-class>jdbctest.GoodsBean</ejb-class>
        <persistence-type>Container</persistence-type>
        <prim-key-class>int</prim-key-class>
        <reentrant>False</reentrant>
        <cmp-field>
            <field-name>goodsname</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>goodstype</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>comment</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>price</field-name>
        </cmp-field>
        <cmp-field>
            <field-name>priceoff</field-name>
        </cmp-field>
        <cmp-field>
```

```
            <field-name>id</field-name>
        </cmp-field>
        <primkey-field>id</primkey-field>
        <resource-ref>
            <res-ref-name>jdbc/DataSource</res-ref-name>
            <res-type>javax.sql.DataSource</res-type>
            <res-auth>Container</res-auth>
        </resource-ref>
    </entity>
    </enterprise-beans>
    <assembly-descriptor>
    <container-transaction>
    <method>
        <ejb-name>Goods</ejb-name>
        <method-name>*</method-name>
        </method>
        <trans-attribute>Required</trans-attribute>
        </container-transaction>
        </assembly-descriptor>
</ejb-jar>
```

在 ejb-jar.xml 文件中，定义了 Goods EJB 的具体信息，包括 Remote Interface、Home Interface 的名字、管理模式(CMP)、主键的数据类型，还定义了 Goods EJB 所映射的 goods Table 的字段名。下面的这段代码定义了 Goods EJB 所使用的 JDBC 数据源的名字。

例：

```
<resource-ref>
    <res-ref-name>jdbc/DataSource</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
</resource-ref>
```

下面我们再来看看 ejb-inprise.xml。请看程序清单 3.5(ejb-inprise.xml)。

程序清单 3.5(ejb-inprise.xml)

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

<!DOCTYPE inprise-specific PUBLIC '-//Inprise Corporation//DTD Enterprise JavaBeans
    1.1//EN' 'http://www.borland.com/devsupport/appserver/dtds/ejb-inprise.dtd'>

<inprise-specific>
    <enterprise-beans>
    <entity>
        <ejb-name>Goods</ejb-name>
        <bean-home-name>Goods</bean-home-name>
        <resource-ref>
```

```
            <res-ref-name>jdbc/DataSource</res-ref-name>
            <jndi-name>DataSource</jndi-name>
        </resource-ref>
        <cmp-info>
        <database-map>
            <table>goods</table>
        </database-map>
        <finder>
            <method-signature>findAll()</method-signature>
            <where-clause />
            <load-state>True</load-state>
        </finder>
        </cmp-info>
    </entity>
    </enterprise-beans>
    <datasource-definitions>
    <datasource>
        <jndi-name>DataSource</jndi-name>
        <url>jdbc:odbc:test</url>
        <username>sa</username>
        <password></password>
        <driver-class-name>sun.jdbc.odbc.JdbcOdbcDriver</driver-class-name>
    </datasource>
    </datasource-definitions>
</inprise-specific>
```

在程序清单 3.5 中，主要定义的是 Goods EJB 所使用的 JDBC 数据源的信息，例如用户名是什么，密码是什么，JDBC URL 地址是什么等等，还定义了 Goods EJB 所映射的 Table 为 goods。ejb-inprise.xml 是 Borland 公司特有的 XML 文件，如果是利用别的公司的产品开发 EJB 组件，那么这个 XML 文件的名字可能有所不同，例如 jboss.xml 就是 JBoss 所指定的 EJB 组件的部署描述符，weblogic-ejb-jar.xml 就是 WebLogic 服务器所需要的 EJB 组件的部署描述符，而 ejb-jar.xml 文件是 EJB 规范规定的必须要有的 XML 文件，无论使用哪种工具开发 EJB 组件，都必须包含这个 XML 文件。

3.1.6　创建 EJB Container

在上文中，我们已经介绍了 Home Interface、Remote Interface、EJB Class，现在我们应该利用 JBuilder4 来生成 Goods EJB 的 EJB Container，方法如第 2 章所述。

在 JBuilder 的主窗口中，左侧，右键单击 GoodsHome.java，选择 Properties….→选择 Build Tab→VisiBroker，在 Generate IIOP 前面的小方框前打勾。

然后编译 Project，JBuilder4 会自动产生十几个 java 文件，这些 java 文件就构成了 EJB Container 和 Home Object，这一节中开发的是 CMP 类型的 Entity EJB，所以在 EJB Container 中应该具体实现 ejbLoad()、ejbStore()、ejbRemove() 、ejbActivate()、ejbPassivate()等方法，

至于如何实现这些方法，我们就不用关心了，JBuilder4 会自动处理这些细节问题的。编译完 Project 以后，千万不要忘了保存所有的项目文件。

3.1.7　创建 EJB 客户端

在上面的几个小节中，我们已经开发了一个 CMP 类型的实体 EJB，现在应该编写一个 EJB 客户端程序对它进行测试，为了节省篇幅和方便起见，我们仅仅介绍如何编写基于 Java Application 的 EJB 客户端，至于基于 JSP 的 EJB 客户端，读者可以自行编写，实际上，编写 EJB 比使用 JSP 技术编写 EJB 的客户端要简单很多倍，而且我们只要会编写基于 Java Application 的 EJB 客户端，那么在此基础上稍微改写一下，就是基于 JSP 的 EJB 客户端程序了，这一项技术想必读者早已经掌握了，即使读者还没有掌握这门技术，那么读者可以参考第 2 章最后的 EJB 客户端程序，那就是基于 JSP 的 EJB 客户端程序。

如何开发基于 Java Application 的 EJB 的客户端程序呢？Entity EJB 的客户端程序与 Session EJB 的客户端程序有何不同呢?我们的回答是这两者没有任何不同的地方，都是查找 Home Interface、创建 Remote Interface、调用商业逻辑这三部曲。对于第一个问题的回答如下，请读者在 JBuilder 中，选择 File→New…→Enterprise Tab→EJB Test Client→OK，将出现如图 3.14 所示的窗口。

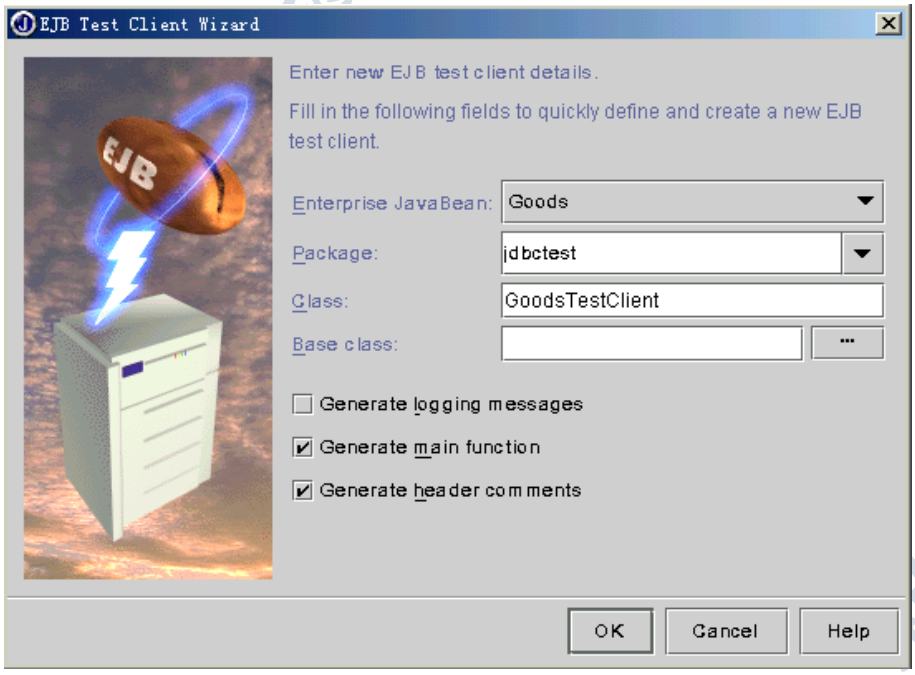


图 3.14　JBuilder4 的 EJB Client Test Wizard

在图 3.14 中，我们把 Class 框的缺省值 GoodsTestClient1 改为 GoodsTestClient，其他的一切都不要改动，单击 OK，回到 JBuilder 的主界面，JBuilder 为我们自动创建了 GoodsTestClient.java 程序。编辑 GoodsTestClient.java，使得它如下面的程序清单 3.6 所示，其中黑体的部分是我们自己加上去的，其余的部分是 JBuilder 自动产生的。

程序清单 3.6

```
//File Name:GoodsTestClient.java
//Author:fancy
//Date:2001.5.10
//Note:test the client

package jdbctest;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.util.Collection;
import java.util.Iterator;

public class GoodsTestClient
{
    private static final int MAX_OUTPUT_LINE_LENGTH = 50;
    private GoodsHome goodsHome = null;

    /**Construct the EJB test client*/
    public GoodsTestClient()
    {
        try
        {
            //get naming context
            Context ctx = new InitialContext();
            //look up jndi name
            Object ref = ctx.lookup("Goods");
            //cast to Home interface
            goodsHome = (GoodsHome)
                PortableRemoteObject.narrow(ref，GoodsHome.class);
            GoodsRemote gr=goodsHome.findByPrimaryKey(22);
            System.out.println("name:"+gr.getGoodsname());
            System.out.println("type:"+gr.getGoodstype());
            System.out.println("id:"+gr.getId());
            System.out.println("price:"+gr.getPrice());
            System.out.println("off:"+gr.getPriceoff());
            System.out.println("comment:"+gr.getComment());
            System.out.println("is session ejb:"+
                goodsHome.getEJBMetaData().isSession());
    }
    catch(Exception e)
    {
        e.printStackTrace();
     }
}
```

北京希望电子出版社

```
//--------------------------------------------------------------------------
// Utility Methods
//--------------------------------------------------------------------------

public GoodsHome getHome()
{
        return goodsHome;
}

public void executeFindAll()
{
        try
        {
            Collection collection = goodsHome.findAll();
            Iterator iterator = collection.iterator();
            while (iterator.hasNext())
            {
            Object object = iterator.next();
            GoodsRemote goodsRemote = (GoodsRemote)
                PortableRemoteObject.narrow(object，GoodsRemote.class);
            log(" id="+goodsRemote.getId()+" "+goodsRemote.toString());
            }
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    private void log(String message)
    {
        if (message.length() > MAX_OUTPUT_LINE_LENGTH)
        {
            System.out.println("-- " +
                message.substring(0，MAX_OUTPUT_LINE_LENGTH) + " ...");
        }
        else
        {
                System.out.println("-- " + message);
        }
    }
    /**Main method*/

    public static void main(String[] args)
    {
        GoodsTestClient client = new GoodsTestClient();
```

北京希望电子出版社

```
        client.executeFindAll();
        // Use the getHome() method of the client object to call Home interface
        // methods that will return a Remote interface reference.    Then
        // use that Remote interface reference to access the EJB.
        // For quick access to all available records，you can use the
        // executeFindAll() method in this class.
        }
}
```

程序清单 3.6 很简单，但是我们有两点要说明。

（1）首先找到下面的代码行，这几行代码首先利用 Home Interface 的 findPrimaryKey()方法，查找主键等于 22 的 Remote Interface，亦即 GoodsRemote 对象，一旦获取了 GoodsRemote 对象，就可以调用在 Remote Interface 中声明的各种 getXXX()方法与 setXXX()方法，获取 GoodsRemote 对象的属性值或者设定它的属性值，这些 getXXX()方法与 setXXX()方法实质上是在存取数据库某个 Table 的行纪录，这个 Table 被 Entity EJB 所映射，在本例中，这个 Table 是 goods，这个 Entity EJB 是 Goods EJB。

例：

```
GoodsRemote gr=goodsHome.findByPrimaryKey(22);
System.out.println("name:"+gr.getGoodsname());
System.out.println("type:"+gr.getGoodstype());
System.out.println("id:"+gr.getId());
System.out.println("price:"+gr.getPrice());
System.out.println("off:"+gr.getPriceoff());
System.out.println("comment:"+gr.getComment());
System.out.println("is session ejb:"+goodsHome.getEJBMetaData().isSession());
```

（2）读者请留意 executeFindAll()方法，这个自定义的方法调用 Home Interface 声明的 findAll()方法，遍历 EJB Server/Container 中所有可用的 Goods EJB 对象(实际上是 GoodsRemote 对象)，并且输出它们的主键信息。

3.1.8　运行和测试

下面我们该运行和测试 Goods EJB 了，在运行 Goods EJB 之前，首先要对 Project 做一些设定。选择 Run→Configuration…，选中<Default>，然后单击 OK 即可。别忘了要启动 Smart Agent，如果读者对这一部分的内容不熟悉，不妨参考第 2 章的内容。

首先应该运行 Goods EJB，运行 Goods EJB 的输出结果如下所示(前半段输出已经省略了)：

```
Initializing EJBs..... done
Container [ejbcontainer] is ready
    EJB Container Statistics
        ========================
        Time              Thu Mar 25 23:52:59 CST 1999
        Memory (used)       1704 Kb    (max 1704 Kb)
        Memory (total)      2736 Kb    (max 2736 Kb)
        Memory (free)       37.0%
```

```
------------------------
Home                    Goods
Total in memory         0
Total in use            0
==========================
```

接着应该运行 Goods EJB 的客户端程序——GoodsTestClient.java，在 JBuilder 的主界面中，左侧，右键单击 GoodsTestClient.java→Run，运行输出如下所示：

```
name:小楼一夜听春雨
type:兵器
id:22
price:3000.8
off:0.8
comment:小楼一夜听春雨，深巷明朝卖杏花，好优美的诗意啊，可惜它是一把弯刀，又名圆月弯刀，曾经是青青所用之物，后归丁鹏所有，和万水青山踏遍齐名。
is session ejb:false
--  id=1 Stub[repository_id=RMI:jdbctest.GoodsRemote: ...
--  id=2 Stub[repository_id=RMI:jdbctest.GoodsRemote: ...
--  id=3 Stub[repository_id=RMI:jdbctest.GoodsRemote: ...
--  id=4 Stub[repository_id=RMI:jdbctest.GoodsRemote: ...
--  id=22 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=6 Stub[repository_id=RMI:jdbctest.GoodsRemote: ...
--  id=7 Stub[repository_id=RMI:jdbctest.GoodsRemote: ...
--  id=8 Stub[repository_id=RMI:jdbctest.GoodsRemote: ...
--  id=9 Stub[repository_id=RMI:jdbctest.GoodsRemote: ...
--  id=10 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=11 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=12 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=13 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=14 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=15 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=16 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=17 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=18 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=19 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
--  id=20 Stub[repository_id=RMI:jdbctest.GoodsRemote ...
```

这时，如果再查看 Goods EJB 服务端的输出，将会有下面的结果：

```
EJB Container Statistics
    ==========================
    Time              Fri Mar 26 00:04:36 CST 1999
    Memory (used)         1522 Kb    (max 1736 Kb)
    Memory (total)        2664 Kb    (max 2664 Kb)
    Memory (free)         42.0%
    ------------------------
    Home                  Goods
    POOLED                1
```

```
Total in memory        1
Total in use           1
========================
EJB Container Statistics
========================
Time                Fri Mar 26 00:04:51 CST 1999
Memory (used)       1670 Kb    (max 1736 Kb)
Memory (total)      2664 Kb    (max 2664 Kb)
Memory (free)       37.0%
------------------------
Home                Goods
READY               1
POOLED              1
Total in memory     2
Total in use        2
========================
EJB Container Statistics
========================
Time                Fri Mar 26 00:04:56 CST 1999
Memory (used)       1692 Kb    (max 1736 Kb)
Memory (total)      2664 Kb    (max 2664 Kb)
Memory (free)       36.0%
------------------------
Home                Goods
READY               20
Total in memory     20
Total in use        20
========================
```

上面的输出结果说明了在 EJB Server/Container 中，可用的 Goods EJB 对象一共有 20 个，刚好等于 goods Table 纪录的行数，这是巧合吗?读者不妨考虑一下。

3.2　实体 EJB 的开发技术之二——BMP EJB

上一节，我们已经介绍了 CMP 模式的 Entity EJB 的开发技术，在这一节中，我们将介绍 BMP 模式的 Entity EJB 的开发技术。

3.2.1　BMP EJB 简介

BMP EJB，全名是 Beans Managed Persistence EJB，中文译名为 Bean 自管理模式的 EJB，BMP 模式的 EJB 与 CMP 模式的 EJB 最大的不同之处在于 ejbLoad()、ejbStore()、ejbRemove()、ejbActivate()、ejbPassivate()、ejbCreate()等方法都必须由开发者编码实现，EJB Container 不会再为我们代劳，也就是说我们必须直接和数据源打交道，必须自己编写存取数据库纪录集的代码，如果没有开发工具的帮助，这真是一项十分繁琐复杂的任务。

使用 BMP EJB 的好处在于它的灵活性，如果我们要实现一个条件查询，查询条件会经常变化，这时候，使用 CMP EJB 不是一个最好的选择，这种情况下，BMP EJB 就比 CMP

EJB 要优越得多。我们可以在编码实现上述方法的同时，添加自己的商业逻辑，这就是 BMP 模式的一个突出之处——扩展性，如果你对于 EJB 编程十分熟练的话，你就会发现，BMP 模式带给你的便利远远超过了带给你的不便。

我们建议读者在必要的时候采用 BMP 模式的 EJB 完成你的开发项目。

3.2.2　创建 EJB 工程

开发 CMP 模式的 EJB 与开发 BMP 模式的 EJB 有何不同?应该说，在创建 EJB 的框架时，这两种模式没有什么不同，但是在开发的后期，CMP 模式的 EJB 与 BMP 模式的 EJB 的 EJB Class 大不相同，同时，它们的部署描述符也有所不同。

在本小节中，我们将利用 JBuilder4 创建一个 Entity EJB Modeler Project，这个 Project 的名字为 JDBCBean，前面的步骤和开发 Goods EJB(JDBCTest Project)一模一样，读者可以参照 3.2 节的说明。

在 EJB Entity Bean Modeler Wizard 的第六步，我们需要修改此 EJB 的 Home Interface、Remote Interface 的名字，如图 3.15 所示，这个 Entity EJB 的名字仍然是 Goods，映射的 Table 仍然是 goods，d 代表主键的字段仍然是 id 字段，主键的数据类型为 int。

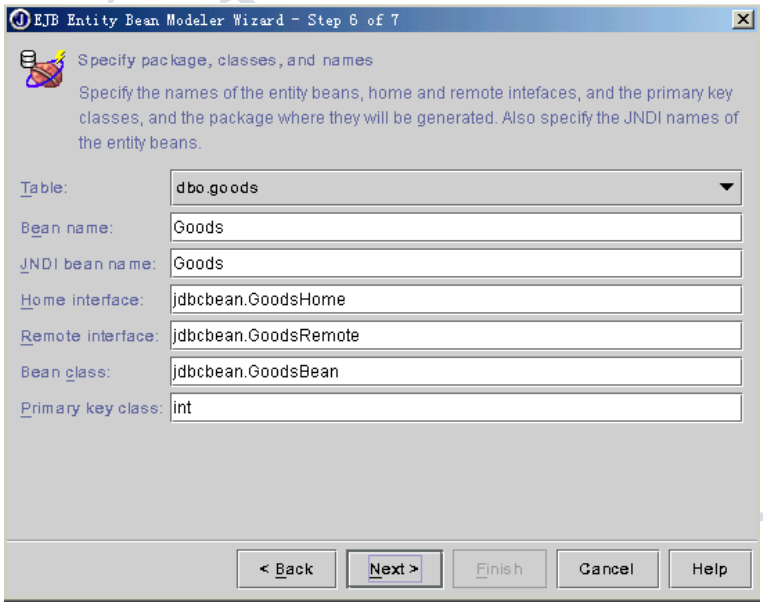


图 3.15　JBuilder4 的 EJB Entity Bean Modeler Wizard

在图 3.15 的窗口中，都设定好以后，单击 Next 按钮，将出现如图 3.16 所示的窗口，在这个窗口中，我们将要设定 Entity EJB 的开发模式，本节开发的是 BMP 模式的 Entity EJB，所以我们应该选中 Bean managed peresistence，然后单击 Finish，JBuilder4 会根据我们的设定，自动产生 Home Interface、Remote Interface、EJB Class、部署描述符等文件，这样就算完成 EJB 框架的开发工作，下面就来看看 JBuilder4 为我们自动产生的代码。

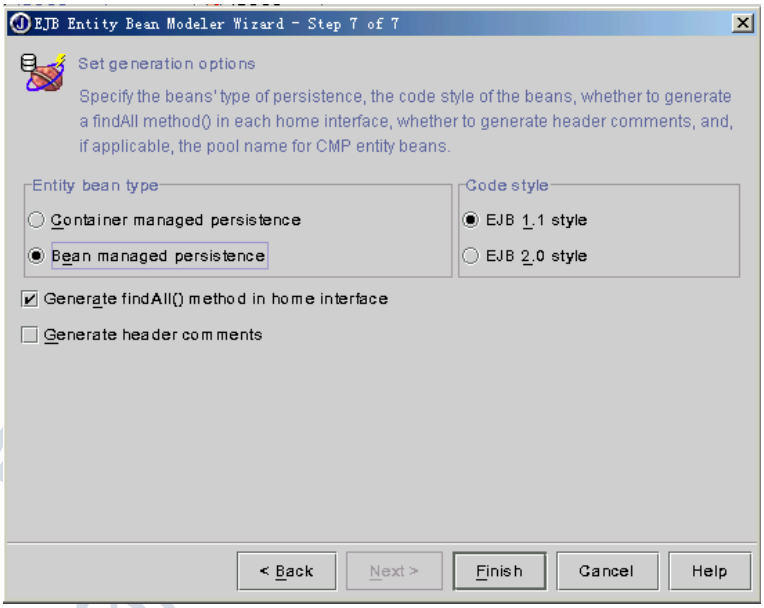


图 3.16　JBuilder4 的 EJB Entity Bean Modeler Wizard

3.2.3　Home Interface 和 Remote Interface

读者请看程序清单 3.7 和程序清单 3.8，这就是 BMP 模式的 Goods EJB 的 Home Interface、Remote Interface 的源代码，读者不妨把它们和程序清单 3.1、3.2 比较一下，看看有没有差别，实际上是没有任何差别的。BMP 模式的 EJB 与 CMP 模式的 EJB 的差别不在于 Home Interface 或者是 Remote Interface，而在于 EJB Class 与 EJB Container 的不同。

程序清单 3.7

```
//File Name: GoodsHome.java
//Author:fancy
//Date:2001.5
//Note:the Home Interface

package jdbcbean;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;

public interface GoodsHome extends EJBHome
{
    public GoodsRemote create(String goodsname，String goodstype，String comment，
        Double price，Double priceoff，int id) throws RemoteException，CreateException;
    public GoodsRemote create(int id) throws RemoteException，CreateException;
    public GoodsRemote findByPrimaryKey(int primaryKey) throws RemoteException，
        FinderException;
```

```
    public Collection findAll() throws RemoteException，FinderException;
}
```

程序清单 3.8

```
//File Name: GoodsRemote.java
//Author:fancy
//Date:2001.5
//Note:the Remote Interface

package jdbcbean;

import java.rmi.*;
import javax.ejb.*;

public interface GoodsRemote extends EJBObject
{
    String getGoodsname() throws RemoteException;
    void setGoodsname(String goodsname) throws RemoteException;
    String getGoodstype() throws RemoteException;
    void setGoodstype(String goodstype) throws RemoteException;
    String getComment() throws RemoteException;
    void setComment(String comment) throws RemoteException;
    Double getPrice() throws RemoteException;
    void setPrice(Double price) throws RemoteException;
    Double getPriceoff() throws RemoteException;
    void setPriceoff(Double priceoff) throws RemoteException;
    int getId() throws RemoteException;
}
```

正因为程序清单 3.7 和 3.1 没有差别，程序清单 3.8 和程序清单 3.2 也没有差别，所以我们就不再重复解释这两个程序了，读者可以参考上文的叙述。

3.2.4　EJB 类

在上文我们已经不止一次提到过，BMP 模式的 EJB 与 CMP 模式的 EJB 的结构最大的不同之处在于 EJB Class 与 EJB Container，BMP 模式的 EJB 的 EJB Class 分为两个部分，一个是普通的 EJB Class(GoodsBean.java)，继承自 EntityBean 接口，这部分只是简单地声明了 ejbLoad()、ejbStore()、ejbRemove()、ejbActivate()、ejbPassivate()、ejbCreate()等方法，并没有真正地去实现它们，请看程序清单 3.9(GoodsBean.java)，这一部分和 CMP 模式的 EJB 的 EJB Class 一模一样，读者如果不信，可以比较程序清单 3.9 和程序清单 3.3 的异同，怎么样，找到了吗?实际上，这两个程序没有任何不同，只不过 CMP 模式的 EJB 把这些方法的实现放到了 EJB Container，而在 BMP 模式的 EJB 中，我们必须新定义一个 EJB Class，继承 GoodsBeans 类，并且编码实现 ejbLoad()、ejbStore()、ejbRemove() 、ejbActivate()、ejbPassivate()、ejbCreate()等方法，读者请看程序清单 3.10 (GoodsBeanBMP.java)。

程序清单 3.9

```
//File Name:GoodsBean.java
//Author:Fancy
//Date:2001.5
//Note:the EJB Class
package jdbcbean;

import java.rmi.*;
import javax.ejb.*;

public class GoodsBean implements EntityBean
{
    EntityContext entityContext;
    public String goodsname;
    public String goodstype;
    public String comment;
    public Double price;
    public Double priceoff;
    public int id;
    public int ejbCreate(String goodsname，String goodstype，String comment，Double
        price，Double priceoff，int id) throws CreateException
    {
        this.goodsname = goodsname;
        this.goodstype = goodstype;
        this.comment = comment;
        this.price = price;
        this.priceoff = priceoff;
        this.id = id;
        return 0;
    }
    public int ejbCreate(int id) throws CreateException
    {
        return ejbCreate(null，null，null，null，null，id);
    }
    public void ejbPostCreate(String goodsname，String goodstype，String comment，
        Double price，Double priceoff，int id) throws CreateException
    {
    }
    public void ejbPostCreate(int id) throws CreateException
    {
        ejbPostCreate(null，null，null，null，null，id);
    }
    public void ejbRemove() throws RemoveException
```

北京希望电子出版社

```
{
}
public void ejbActivate()
{
}
public void ejbPassivate()
{
}
public void ejbLoad()
{
}
public void ejbStore()
{
}
public void setEntityContext(EntityContext entityContext)
{
    this.entityContext = entityContext;
}
public void unsetEntityContext()
{
    entityContext = null;
}
public String getGoodsname()
{
    return goodsname;
}
public void setGoodsname(String goodsname)
{
    this.goodsname = goodsname;
}
public String getGoodstype()
{
    return goodstype;
}
public void setGoodstype(String goodstype)
{
    this.goodstype = goodstype;
}
public String getComment()
{
    return comment;
}
public void setComment(String comment)
```

北京希望电子出版社

```
    {
        this.comment = comment;
    }
    public Double getPrice()
    {
        return price;
    }
    public void setPrice(Double price)
    {
        this.price = price;
    }
    public Double getPriceoff()
    {
        return priceoff;
    }
    public void setPriceoff(Double priceoff)
    {
        this.priceoff = priceoff;
    }
    public int getId()
    {
        return id;
    }
}
```

程序清单 3.10

```
//File Name:GoodsBeanBMP.java
//Author:Fancy
//Date:2001.5
//Note:the EJB Class

package jdbcbean;

import java.sql.*;
import javax.ejb.*;
import javax.sql.DataSource;
import java.util.*;

public class GoodsBeanBMP extends GoodsBean
{
    DataSource dataSource;
     public int ejbCreate(String goodsname，String goodstype，String comment，Double price，
        Double priceoff，int id) throws CreateException
    {
```

北京希望电子出版社

```
super.ejbCreate(goodsname，goodstype，comment，price，priceoff，id);
try
{
    //First see if the object already exists
    ejbFindByPrimaryKey(id);
    //If so，then we have to throw an exception
    throw new DuplicateKeyException("Primary key already exists");
}
catch(ObjectNotFoundException e)
{
    //Otherwise we can go ahead and create it...
}
Connection connection = null;
PreparedStatement statement = null;
try
{
    connection = dataSource.getConnection();
    statement = connection.prepareStatement("INSERT INTO goods (goodsname，
        goodstype，comment，price，priceoff，id) VALUES (?，?，?，?，?，?)");
    statement.setString(1，goodsname);
    statement.setString(2，goodstype);
    statement.setString(3，comment);
    statement.setDouble(4，price.doubleValue());
    statement.setDouble(5，priceoff.doubleValue());
    statement.setInt(6，id);
    if (statement.executeUpdate() != 1)
    {
        throw new CreateException("Error adding row");
    }
    statement.close();
    statement = null;
    connection.close();
    connection = null;
    return id;
}
catch(SQLException e)
{
    throw new EJBException("Error executing SQL INSERT INTO goods
        (goodsname，goodstype，comment，price，priceoff，id) VALUES
            (?，?，?，?，?，?): " + e.toString());
}
finally
{
```

```
                try
                {
                        if (statement != null)
                        {
                                statement.close();
                        }
                }
                catch(SQLException e)
                {
                }
                try
                {
                        if (connection != null)
                        {
                                connection.close();
                        }
                }
                catch(SQLException e)
                {
                }
        }
}
public int ejbCreate(int id) throws CreateException
{
        return ejbCreate(null，null，null，null，null，id);
}
public void ejbRemove() throws RemoveException
{
        super.ejbRemove();
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
                connection = dataSource.getConnection();
                statement = connection.prepareStatement("DELETE FROM goods WHERE
                        id = ?");
                statement.setInt(1，id);
                if (statement.executeUpdate() < 1)
                {
                        throw new RemoveException("Error deleting row");
                }
                statement.close();
                statement = null;
```

北京希望电子出版社

```
                connection.close();
                connection = null;
            }
            catch(SQLException e)
            {
                throw new EJBException("Error executing SQL DELETE FROM goods
                    WHERE id = ?: " + e.toString());
            }
            finally
            {
                try
                {
                    if (statement != null)
                    {
                        statement.close();
                    }
                }
                catch(SQLException e)
                {
                }
                try
                {
                    if (connection != null)
                    {
                        connection.close();
                    }
                }
                catch(SQLException e)
                {
                }
            }
        }
        public void ejbLoad()
        {
            id = ((Integer) entityContext.getPrimaryKey()).intValue();
            Connection connection = null;
            PreparedStatement statement = null;
            try
            {
                connection = dataSource.getConnection();
                statement = connection.prepareStatement("SELECT goodsname，goodstype，
                    comment，price，priceoff FROM goods WHERE id = ?");
                statement.setInt(1，id);
```

北京希望电子出版社

```
        ResultSet resultSet = statement.executeQuery();
        if (!resultSet.next())
        {
            throw new NoSuchEntityException("Row does not exist");
        }
        goodsname = resultSet.getString(1);
        goodstype = resultSet.getString(2);
        comment = resultSet.getString(3);
        price = new Double(resultSet.getDouble(4));
        priceoff = new Double(resultSet.getDouble(5));
        statement.close();
        statement = null;
        connection.close();
        connection = null;
    }
    catch(SQLException e)
    {
        throw new EJBException("Error executing SQL SELECT goodsname，
            goodstype，comment，price，priceoff FROM goods WHERE id = ?: "
                +e.toString());
    }
    finally
    {
        try
        {
            if (statement != null)
            {
                statement.close();
            }
        }
        catch(SQLException e)
        {
        }
        try
        {
            if (connection != null)
            {
                connection.close();
            }
        }
        catch(SQLException e)
        {
        }
```

北京希望电子出版社

```
        }
        super.ejbLoad();
    }
    public void ejbStore()
    {
        super.ejbStore();
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
            connection = dataSource.getConnection();
            statement = connection.prepareStatement("UPDATE goods SET goodsname
                = ?，goodstype = ?，comment = ?，price = ?，priceoff = ?
                    WHERE id = ?");
            statement.setString(1，goodsname);
            statement.setString(2，goodstype);
            statement.setString(3，comment);
            statement.setDouble(4，price.doubleValue());
            statement.setDouble(5，priceoff.doubleValue());
            statement.setInt(6，id);
            if (statement.executeUpdate() < 1)
            {
                throw new NoSuchEntityException("Row does not exist");
            }
            statement.close();
            statement = null;
            connection.close();
            connection = null;
        }
        catch(SQLException e)
        {
            throw new EJBException("Error executing SQL UPDATE goods SET
                goodsname = ?，goodstype = ?，comment = ?，price = ?，priceoff = ?
                    WHERE id = ?: " + e.toString());
        }
        finally
        {
            try
            {
                if (statement != null)
                {
                    statement.close();
                }
```

```
            }
            catch(SQLException e)
            {
            }
            try
            {
                if (connection != null)
                {
                    connection.close();
                }
            }
            catch(SQLException e)
            {
            }
        }
}
public int ejbFindByPrimaryKey(int key) throws ObjectNotFoundException
{
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
            connection = dataSource.getConnection();
            statement = connection.prepareStatement("SELECT id FROM goods WHERE
                id = ?");
            statement.setInt(1，key);
            ResultSet resultSet = statement.executeQuery();
            if (!resultSet.next())
            {
                throw new ObjectNotFoundException("Primary key does not exist");
            }
            statement.close();
            statement = null;
            connection.close();
            connection = null;
            return key;
        }
        catch(SQLException e)
        {
            throw new EJBException("Error executing SQL SELECT id FROM goods
                WHERE id = ?: " + e.toString());
        }
        finally
```

```
        {
            try
            {
                if (statement != null)
                {
                    statement.close();
                }
            }
            catch(SQLException e)
            {
            }
            try
            {
                if (connection != null)
                {
                    connection.close();
                }
            }
            catch(SQLException e)
            {
            }
        }
    }
    public Collection ejbFindAll()
    {
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
            connection = dataSource.getConnection();
            statement = connection.prepareStatement("SELECT id FROM goods");
            ResultSet resultSet = statement.executeQuery();
            Vector keys = new Vector();
            while (resultSet.next())
            {
                int id = resultSet.getInt(1);
                keys.addElement(new Integer(id));
            }
            statement.close();
            statement = null;
            connection.close();
            connection = null;
            return keys;
```

北京希望电子出版社

```
        }
        catch(SQLException e)
        {
            throw new EJBException("Error executing SQL SELECT id FROM goods: "
                + e.toString());
        }
        finally
        {
            try
            {
                if (statement != null)
                {
                    statement.close();
                }
            }
            catch(SQLException e)
            {
            }
            try
            {
                if (connection != null)
                {
                    connection.close();
                }
            }
            catch(SQLException e)
            {
            }
        }
    }

    public void setEntityContext(EntityContext entityContext)
    {
        super.setEntityContext(entityContext);
        try
        {
            javax.naming.Context context = new javax.naming.InitialContext();
            dataSource = (DataSource)
                context.lookup("java:comp/env/jdbc/DataSource");
        }
        catch(Exception e)
        {
            throw new EJBException("Error looking up dataSource:" + e.toString());
```

北京希望电子出版社

```
            }
        }
    }
```

看完程序清单 3.10 了吗?程序清单 3.10 虽然很长，但是很简单，主要是错误处理的部分占的篇幅比较多，实际上真正的实现代码并不是很多。程序清单 3.10 主要是实现在程序清单 3.9 中定义的那些 ejbLoad()、ejbStore()、ejbRemove() 、ejbActivate()、ejbPassivate()、ejbCreate()方法，其实，这些方法的实现代码十分类似，都是建立数据库连接，创建 SQL 句柄对象，然后根据方法的输入参数，设定 SQL 语句的输入参数，执行 SQL 语句，把结果返回。

我们就以 ejbCreate()方法为例，说明如何做到 BMP，其他方法的实现都和 ejbCreate()方法的实现差不多，我们就不再介绍了，读者可以模拟得知。ejbCreate()方法的实现代码如下例所示。

例：

```
public int ejbCreate(String goodsname，String goodstype，String comment，Double price，Double priceoff，
    int id) throws CreateException
{
    super.ejbCreate(goodsname，goodstype，comment，price，priceoff，id);
    try
    {
        //First see if the object already exists
        ejbFindByPrimaryKey(id);
        //If so，then we have to throw an exception
        throw new DuplicateKeyException("Primary key already exists");
    }
    catch(ObjectNotFoundException e)
    {
        //Otherwise we can go ahead and create it...
    }
    Connection connection = null;
    PreparedStatement statement = null;
    try
    {
        connection = dataSource.getConnection();
        statement = connection.prepareStatement("INSERT INTO goods (goodsname，
            goodstype，comment，price，priceoff，id) VALUES (?，?，?，?，?，?)");
        statement.setString(1，goodsname);
        statement.setString(2，goodstype);
        statement.setString(3，comment);
        statement.setDouble(4，price.doubleValue());
        statement.setDouble(5，priceoff.doubleValue());
        statement.setInt(6，id);
        if (statement.executeUpdate() != 1)
```

```
            {
                throw new CreateException("Error adding row");
            }
            statement.close();
            statement = null;
            connection.close();
            connection = null;
            return id;
        }
        catch(SQLException e)
        {
            throw new EJBException("Error executing SQL INSERT INTO goods
                (goodsname，goodstype，comment，price，priceoff，id) VALUES
                    (?，?，?，?，?，?): " + e.toString());
        }
        finally
        {
            try
            {
                if (statement != null)
                {
                    statement.close();
                }
            }
            catch(SQLException e)
            {
            }
            try
            {
                if (connection != null)
                {
                    connection.close();
                }
            }
            catch(SQLException e)
            {
            }
        }
}
```

这个 ejbCreate()方法的流程如下：

首先调用父类的 ejbCreate()方法，如：

super.ejbCreate(goodsname，goodstype，comment，price，priceoff，id);

判断有没有同样主键存在，如果有，那么会抛出 DuplicateKeyException 异常。

北京希望电子出版社

获取数据库连接对象，如：

```
connection = dataSource.getConnection();
```

建立 SQL 句柄对象，如：

```
statement = connection.prepareStatement("INSERT INTO goods (goodsname，
goodstype，comment，price，priceoff，id) VALUES (?，?，?，?，?，?)");
```

利用 PreparedStatement 对象的 setXXX()方法和输入参数，组装 SQL 语句，如：

```
statement.setString(1，goodsname);
    statement.setString(2，goodstype);
…
```

调用 PreparedStatement 对象的 executeUpdate()方法，执行 SQL 语句，如：

```
if (statement.executeUpdate() != 1)
    {
        throw new CreateException("Error adding row");
    }
```

关闭 statement、connection 等对象，如：

```
statement.close();
    statement = null;
    connection.close();
    connection = null;
```

返回新的 EJB 对象的主键(标识符)

```
return id;
```

这就是 ejbCreate()方法的执行流程，剩余的代码都是错误处理代码。读者不难发现，ejbCreate()方法就是往数据库里插入新的纪录，Entity EJB 就代表一个数据库行纪录，Goods EJB 对象其实就代表了 goods 表的一个行纪录。

3.2.5　部署描述符

程序清单 3.11 和 3.12 就是 BMP 模式的 Goods EJB 的部署描述符。BMP 模式的 Entity EJB 的部署描述符与 CMP 模式的 Entity EJB 的部署描述符几乎一模一样，不过还是有一点点小区别。读者不妨比较程序清单 3.11 和程序清单 3.4，看看这两个部署描述符都有哪些差别。

程序清单 3.11(ejb-jar.xml)

```
<?xml version="1.0" encoding="GBK"?>
<!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>
<enterprise-beans>
    <entity>
        <ejb-name>Goods</ejb-name>
```

```
            <home>jdbcbean.GoodsHome</home>
            <remote>jdbcbean.GoodsRemote</remote>
            <ejb-class>jdbcbean.GoodsBeanBMP</ejb-class>
            <persistence-type>Bean</persistence-type>
            <prim-key-class>int</prim-key-class>
            <reentrant>False</reentrant>
            <resource-ref>
                <res-ref-name>jdbc/DataSource</res-ref-name>
                <res-type>javax.sql.DataSource</res-type>
                <res-auth>Container</res-auth>
            </resource-ref>
        </entity>
    </enterprise-beans>
    <assembly-descriptor>
    <container-transaction>
        <method>
            <ejb-name>Goods</ejb-name>
            <method-name>*</method-name>
            </method>
            <trans-attribute>Required</trans-attribute>
    </container-transaction>
    </assembly-descriptor>
    </ejb-jar>
```

怎么样，找到了这两个程序之间的不同了吗?让我们来告诉你吧，程序清单 3.4 定义了很多 Entity EJB 所映射的 goods Table 的字段，它们都是 Goods EJB 对象的一个属性，而在程序清单 3.11 中就没有这些定义，更重要的是，在程序清单 3.4 中，有这样的一行代码：

<persistence-type>Container</persistence-type>

这说明这个 Entity EJB 的管理模式为 CMP 模式。

而在程序清单 3.11 中，对应的代码为：

<persistence-type>Bean</persistence-type>

这表明对应 Entity EJB 的管理模式一定是 BMP 模式。

下面是程序清单 3.12(ejb-inpries.xml)，它和程序清单 3.5 类似，读者不妨也比较一下这两者的差别，我们就不再多说了。

程序清单 3.12(ejb-inprise.xml)

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE inprise-specific PUBLIC '-//Inprise Corporation//DTD Enterprise JavaBeans
    1.1//EN' 'http://www.borland.com/devsupport/appserver/dtds/ejb-inprise.dtd'>

<inprise-specific>
<enterprise-beans>
    <entity>
        <ejb-name>Goods</ejb-name>
```

```
            <bean-home-name>Goods</bean-home-name>
            <resource-ref>
                <res-ref-name>jdbc/DataSource</res-ref-name>
                <jndi-name>DataSource</jndi-name>
            </resource-ref>
        </entity>
    </enterprise-beans>
    <datasource-definitions>
        <datasource>
            <jndi-name>DataSource</jndi-name>
            <url>jdbc:odbc:test</url>
            <username>sa</username>
            <password></password>
            <driver-class-name>sun.jdbc.odbc.JdbcOdbcDriver</driver-class-name>
        </datasource>
    </datasource-definitions>
    </inprise-specific>
```

3.2.6　创建 EJB Container

这一节中，我们需要创建 EJB Container，步骤和 3.1.6 节所介绍的步骤一模一样，但是结果却很不一样，3.1.6 节中创建的 EJB Container 必须实现 ejbLoad()、ejbRemove()等方法，必须管理 EJB 和 JDBC 数剧源之间的交互操作，但是在这一节中创建的 EJB Container 却不需要实现这些功能，因为我们已经在程序清单 3.10 中实现了 BMP 模式的 EJB Class，这些任务都在其中完成了。

3.2.7　创建 EJB 客户端

本小节，我们将编写 BMP 模式的 Goods EJB 的客户端程序，与 CMP 模式的 Goods EJB 的客户端程序一样，我们只介绍如何编写基于 Java Application 的 EJB 客户端程序，至于基于 JSP 的 EJB 客户端程序，读者可以自己尝试。

下面请看程序清单 3.13(GoodsTestClient.java)，其中黑体的部分是我们自己加上去的代码。

程序清单 3.13

```
//File Name:GoodsTestClient.java
//Author:fancy
//Date:2001.5.10
//Note:to test the ejb

package jdbcbean;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;
```

```
import java.util.Collection;
import java.util.Iterator;

public class GoodsTestClient
{
    private static final int MAX_OUTPUT_LINE_LENGTH = 50;
    private GoodsHome goodsHome = null;

    /**Construct the EJB test client*/
    public GoodsTestClient()
    {
        try
        {
            //get naming context
            Context ctx = new InitialContext();
            //look up jndi name
            Object ref = ctx.lookup("Goods");
            //cast to Home interface
            goodsHome = (GoodsHome)
                PortableRemoteObject.narrow(ref，GoodsHome.class);
            GoodsRemote gr=goodsHome.findByPrimaryKey(22);
            System.out.println("name:"+gr.getGoodsname());
            System.out.println("type:"+gr.getGoodstype());
            System.out.println("id:"+gr.getId());
            System.out.println("price:"+gr.getPrice());
            System.out.println("off:"+gr.getPriceoff());
            System.out.println("comment:"+gr.getComment());
            System.out.println("is session ejb:"
                +goodsHome.getEJBMetaData().isSession());
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
    }

    //-------------------------------------------------------------------------
    // Utility Methods
    //-------------------------------------------------------------------------

    public GoodsHome getHome()
    {
        return goodsHome;
```

```
}

public void executeFindAll()
{
    try
    {
        Collection collection = goodsHome.findAll();
        Iterator iterator = collection.iterator();
        while (iterator.hasNext())
        {
            Object object = iterator.next();
            GoodsRemote goodsRemote = (GoodsRemote)
                PortableRemoteObject.narrow(object，GoodsRemote.class);
            log(goodsRemote.toString());
        }
    }
    catch(Exception e)
    {
        e.printStackTrace();
    }
}
private void log(String message)
{
    if (message.length() > MAX_OUTPUT_LINE_LENGTH)
    {
        System.out.println("-- "
            + message.substring(0，MAX_OUTPUT_LINE_LENGTH) + " ...");
    }
    else
    {
        System.out.println("-- " + message);
    }
}
/**Main method*/

public static void main(String[] args)
{
    GoodsTestClient client = new GoodsTestClient();
    //client. executeFindAll();
    // Use the getHome() method of the client object to call Home interface
    // methods that will return a Remote interface reference.    Then
    // use that Remote interface reference to access the EJB.
    // For quick access to all available records，you can use the
```

北京希望电子出版社

```
            // executeFindAll() method in this class.
        }
    }
```

不知道读者发现没有，程序清单 3.13 和程序清单 3.6 其实是一模一样的，这说明了一个问题，那就是 CMP EJB 的客户端与 BMP EJB 的客户端程序并没有实质上的区别，尽管服务端的实现方法可能不一样，但是所对应的客户端程序却是一样的。这也是分布式处理的好处之一，服务端可以改变服务实现的方法，但是只要提供服务的接口没有变化，客户端的程序就不需要变化。

3.2.8　运行和测试

下面我们按照 3.1.8 小节的方法运行 Goods EJB(BMP 模式下)的服务端与客户端程序，客户端程序(GoodsTestClient.java)的输出结果如下所示：

```
name:小楼一夜听春雨
type:兵器
id:22
price:3000.8
off:0.8
comment:小楼一夜听春雨，深巷明朝卖杏花，好优美的诗意啊，可惜它是一把弯刀，又名圆月弯刀，曾经是青青所用之物，后归丁鹏所有，和万水青山踏遍齐名
is session ejb:false
```

至于服务端的输出结果读者可以参照 3.1.8 小节的结果。

3.3　EJB 开发实例——封装数据源

在这一节中，我们将在 3.2 节的基础上，对 Goods EJB 进行改写，实现动态查询的功能，实际上，这就相当于完成了封装数据源的功能，因为在 3.2 中，我们已经可以利用 Goods EJB 完成插入新纪录、更新行纪录、删除行纪录、返回特定行纪录的功能，现在再加上动态查询的功能，那么一个数据源的功能也就全了。我们为什么不在 3.1 小节的基础上进行再开发呢？这是因为 3.1 小节开发的是 CMP 模式的 Goods EJB，要扩充起来很困难。3.2 小节开发的是 BMP 模式的 Goods EJB，扩展起来容易得多了。

那么如何扩展 BMP 模式的 Goods EJB 呢?

首先需要修改 GoodsBeanBMP.java 程序，定义一个执行动态查询的方法。这个方法的名字是 ejbFindAll(String name)，它的主要作用是查询 goods Table 的 comment 字段，返回所有符合条件的 Goods EJB 对象的集合，ejbFindAll(String name)方法其实是从 ejbFindAll()方法改写过来的。请看程序清单 3.14(GoodsBeanBMP.java)，其中黑体部分就是我们在程序清单 3.10 的基础上加进去的代码。

程序清单 3.14

```
//File Name:GoodsBeanBMP.java
//Author:Fancy
//Date:2001.5
```

```
//Note:the EJB Class

package jdbcbean;

import java.sql.*;
import javax.ejb.*;
import javax.sql.DataSource;
import java.util.*;

public class GoodsBeanBMP extends GoodsBean
{
    DataSource dataSource;
    public int ejbCreate(String goodsname，String goodstype，String comment，Double price，Double
        priceoff，int id) throws CreateException
    {
        super.ejbCreate(goodsname，goodstype，comment，price，priceoff，id);
        try
        {
            //First see if the object already exists
            ejbFindByPrimaryKey(id);
            //If so，then we have to throw an exception
            throw new DuplicateKeyException("Primary key already exists");
        }
        catch(ObjectNotFoundException e)
        {
            //Otherwise we can go ahead and create it...
        }
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
            connection = dataSource.getConnection();
            statement = connection.prepareStatement("INSERT INTO goods (goodsname，
                goodstype，comment，price，priceoff，id) VALUES (?，?，?，?，?，?)");
            statement.setString(1，goodsname);
            statement.setString(2，goodstype);
            statement.setString(3，comment);
            statement.setDouble(4，price.doubleValue());
            statement.setDouble(5，priceoff.doubleValue());
            statement.setInt(6，id);
            if (statement.executeUpdate() != 1)
            {
                throw new CreateException("Error adding row");
```

北京希望电子出版社

```
            }
            statement.close();
            statement = null;
            connection.close();
            connection = null;
            return id;
        }
        catch(SQLException e)
        {
            throw new EJBException("Error executing SQL INSERT INTO goods
                (goodsname，goodstype，comment，price，priceoff，id) VALUES
                    (?，?，?，?，?，?): " + e.toString());
        }
        finally
        {
            try
            {
                if (statement != null)
                {
                    statement.close();
                }
            }
            catch(SQLException e)
            {
            }
            try
            {
                if (connection != null)
                {
                    connection.close();
                }
            }
            catch(SQLException e)
            {
            }
        }
    }
    public int ejbCreate(int id) throws CreateException
    {
        return ejbCreate(null，null，null，null，null，id);
    }
    public void ejbRemove() throws RemoveException
    {
```

北京希望电子出版社

```
super.ejbRemove();
Connection connection = null;
PreparedStatement statement = null;
try
{
    connection = dataSource.getConnection();
    statement = connection.prepareStatement("DELETE FROM goods WHERE
        id = ?");
    statement.setInt(1， id);
    if (statement.executeUpdate() < 1)
    {
        throw new RemoveException("Error deleting row");
    }
    statement.close();
    statement = null;
    connection.close();
    connection = null;
}
catch(SQLException e)
{
    throw new EJBException("Error executing SQL DELETE FROM goods
        WHERE id = ?: " + e.toString());
}
finally
{
    try
    {
        if (statement != null)
        {
            statement.close();
        }
    }
    catch(SQLException e)
    {
    }
    try
    {
        if (connection != null)
        {
            connection.close();
        }
    }
    catch(SQLException e)
```

北京希望电子出版社

```
            {
            }
        }
    }

    public void ejbLoad()
    {
        id = ((Integer) entityContext.getPrimaryKey()).intValue();
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
            connection = dataSource.getConnection();
            statement = connection.prepareStatement("SELECT goodsname，goodstype，
                comment，price，priceoff FROM goods WHERE id = ?");
            statement.setInt(1，id);
            ResultSet resultSet = statement.executeQuery();
            if (!resultSet.next())
            {
                throw new NoSuchEntityException("Row does not exist");
            }
            goodsname = resultSet.getString(1);
            goodstype = resultSet.getString(2);
            comment = resultSet.getString(3);
            price = new Double(resultSet.getDouble(4));
            priceoff = new Double(resultSet.getDouble(5));
            statement.close();
            statement = null;
            connection.close();
            connection = null;
        }
        catch(SQLException e)
        {
            throw new EJBException("Error executing SQL SELECT goodsname，
                goodstype，comment，price，priceoff FROM goods WHERE id = ?: "
                    +e.toString());
        }
        finally
        {
            try
            {
                if (statement != null)
                {
```

```
                    statement.close();
                }
            }
            catch(SQLException e)
            {
            }
            try
            {
                if (connection != null)
                {
                    connection.close();
                }
            }
            catch(SQLException e)
            {
            }
        }
        super.ejbLoad();
}

public void ejbStore()
{
        super.ejbStore();
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
            connection = dataSource.getConnection();
            statement = connection.prepareStatement("UPDATE goods SET goodsname
                = ?，goodstype = ?，comment = ?，price = ?，priceoff = ?
                    WHERE id = ?");
            statement.setString(1，goodsname);
            statement.setString(2，goodstype);
            statement.setString(3，comment);
            statement.setDouble(4，price.doubleValue());
            statement.setDouble(5，priceoff.doubleValue());
            statement.setInt(6，id);
            if (statement.executeUpdate() < 1)
            {
                throw new NoSuchEntityException("Row does not exist");
            }
            statement.close();
            statement = null;
```

北京希望电子出版社

```
            connection.close();
            connection = null;
        }
        catch(SQLException e)
        {
            throw new EJBException("Error executing SQL UPDATE goods SET
                goodsname = ?，goodstype = ?，comment = ?，price = ?，priceoff = ?
                    WHERE id = ?: " + e.toString());
        }
        finally
        {
            try
            {
                if (statement != null)
                {
                    statement.close();
                }
            }
            catch(SQLException e)
            {
            }
            try
            {
                if (connection != null)
                {
                    connection.close();
                }
            }
            catch(SQLException e)
            {
            }
        }
}

public int ejbFindByPrimaryKey(int key) throws ObjectNotFoundException
{
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
            connection = dataSource.getConnection();
            statement = connection.prepareStatement("SELECT id FROM goods WHERE
                id = ?");
```

北京希望电子出版社

```
                statement.setInt(1，key);
                ResultSet resultSet = statement.executeQuery();
                if (!resultSet.next())
                {
                        throw new ObjectNotFoundException("Primary key does not exist");
                }
                statement.close();
                statement = null;
                connection.close();
                connection = null;
                return key;
        }
        catch(SQLException e)
        {
                throw new EJBException("Error executing SQL SELECT id FROM goods
                        WHERE id = ?: " + e.toString());
        }
        finally
        {
                try
                {
                        if (statement != null)
                        {
                                statement.close();
                        }
                }
                catch(SQLException e)
                {
                }
                try
                {
                        if (connection != null)
                        {
                                connection.close();
                        }
                }
                        catch(SQLException e)
                        {
                        }
        }
}
public Collection ejbFindAll()
        {
```

北京希望电子出版社

```
Connection connection = null;
PreparedStatement statement = null;
try
{
        connection = dataSource.getConnection();
        statement = connection.prepareStatement("SELECT id FROM goods");
        ResultSet resultSet = statement.executeQuery();
        Vector keys = new Vector();
        while (resultSet.next())
        {
                int id = resultSet.getInt(1);
                keys.addElement(new Integer(id));
        }
        statement.close();
        statement = null;
        connection.close();
        connection = null;
        return keys;
}
catch(SQLException e)
{
        throw new EJBException("Error executing SQL SELECT id FROM goods: "
                + e.toString());
}
finally
{
        try
        {
                if (statement != null)
                {
                        statement.close();
                }
        }
        catch(SQLException e)
        {
        }
        try
        {
                if (connection != null)
                {
                        connection.close();
                }
        }
```

北京希望电子出版社

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

public void setEntityContext(EntityContext entityContext)
{
        super.setEntityContext(entityContext);
        try
        {
                javax.naming.Context context = new javax.naming.InitialContext();
                dataSource = (DataSource)
                        context.lookup("java:comp/env/jdbc/DataSource");
        }
        catch(Exception e)
        {
                throw new EJBException("Error looking up dataSource:" + e.toString());
        }
}

public Collection ejbFindAll(String name)
{
        Connection connection = null;
        PreparedStatement statement = null;
        try
        {
        connection = dataSource.getConnection();
        statement = connection.prepareStatement("SELECT id FROM goods
                where comment like ?");
        statement.setBytes(1，name.getBytes("GBK"));
        ResultSet resultSet = statement.executeQuery();
        Vector keys = new Vector();
        while (resultSet.next())
        {
                int id = resultSet.getInt(1);
                keys.addElement(new Integer(id));
        }
        statement.close();
        statement = null;
        connection.close();
        connection = null;
        return keys;
```

北京希望电子出版社

```
            }
            catch(Exception e)
            {
                throw new EJBException("Error executing SQL SELECT id FROM
                    goods: " + e.toString());
            }
            finally
            {
            try
            {
                if (statement != null)
                {
                statement.close();
                }
            }
            catch(SQLException e)
            {
            }
            try
            {
                if (connection != null)
                {
                connection.close();
                }
            }
            catch(SQLException e)
            {
            }
            }
        }
    }
```

ejbFindAll(String name)方法的实现逻辑很简单，关键之处在于构造 SQL 语句，以及根据方法参数(name)设定 SQL 语句的输入参数，还需要注意数据库的中文问题，此外就没有什么特别的了。

下面我们需要修改 Goods EJB 的 Home Interface，在 Home Interface 中声明 ejbFindAll(String name)方法，修改过的 Home Interface 的源代码如程序清单 3.15 所示，其中黑体部分是我们添加上的代码。

程序清单 3.15

```
package jdbcbean;

import java.rmi.*;
import javax.ejb.*;
```

北京邮电大学出版社

```
import java.util.*;

public interface GoodsHome extends EJBHome
{
    public GoodsRemote create(String goodsname，String goodstype，String comment，
        Double price，Double priceoff，int id) throws RemoteException，CreateException;
    public GoodsRemote create(int id) throws RemoteException，CreateException;
    public GoodsRemote findByPrimaryKey(int primaryKey) throws RemoteException，
        FinderException;
    public Collection findAll() throws RemoteException，FinderException;
    public Collection findAll(String name) throws RemoteException，FinderException;
}
```

我们把整个 Project 再编译一遍，就可以运行 Goods EJB 的服务端了。下面我们应该编写新的 Goods EJB 的客户端了，这是基于 Java Application 的客户端，基于 JSP 的 EJB 客户端我们就不介绍了，读者可以把前者改写一下就行了。请看程序清单 3.16，其中黑体部分是我们在 JBuilder4 自动产生的代码的基础上添加的代码。

程序清单 3.16

```
package jdbcbean;

import javax.naming.*;
import javax.rmi.PortableRemoteObject;
import java.util.Collection;
import java.util.Iterator;

public class GoodsTestClient
{
private static final int MAX_OUTPUT_LINE_LENGTH = 50;
    private GoodsHome goodsHome = null;

    /**Construct the EJB test client*/
    public GoodsTestClient()
    {
        try
        {
            //get naming context
            Context ctx = new InitialContext();

            //look up jndi name
            Object ref = ctx.lookup("Goods");

            //cast to Home interface
            goodsHome = (GoodsHome)
                PortableRemoteObject.narrow(ref，GoodsHome.class);
```

北京希望电子出版社

```
            GoodsRemote gr=goodsHome.findByPrimaryKey(22);
            System.out.println("name:"+gr.getGoodsname());
            System.out.println("type:"+gr.getGoodstype());
            System.out.println("id:"+gr.getId());
            System.out.println("price:"+gr.getPrice());
            System.out.println("off:"+gr.getPriceoff());
            System.out.println("comment:"+gr.getComment());
            System.out.println("is session ejb:"
                  +goodsHome.getEJBMetaData().isSession());
        }
        catch(Exception e)
        {
            e.printStackTrace();
        }
}

//-------------------------------------------------------------------------
// Utility Methods
//-------------------------------------------------------------------------

public GoodsHome getHome()
{
        return goodsHome;
}

public void find()
{
        try
        {
        Collection collection = goodsHome.findAll("%小楼%");
        Iterator iterator = collection.iterator();
        System.out.println(collection.size());
        while (iterator.hasNext())
        {
            Object object = iterator.next();
                GoodsRemote goodsRemote = (GoodsRemote)
            PortableRemoteObject.narrow(object，GoodsRemote.class);
            System.out.println("name:"+goodsRemote.getGoodsname());
            System.out.println("type:"+goodsRemote.getGoodstype());
            System.out.println("id:"+goodsRemote.getId());
            System.out.println("price:"+goodsRemote.getPrice());
            System.out.println("off:"+goodsRemote.getPriceoff());
            System.out.println("comment:"+goodsRemote.getComment());
        }
        }
```

北京希望电子出版社

```
        catch (Exception fe)
        {
        fe.printStackTrace();
        }
}

public void executeFindAll()
{
        try
        {
                Collection collection = goodsHome.findAll();
                Iterator iterator = collection.iterator();
                while (iterator.hasNext())
                {
                        Object object = iterator.next();
                        GoodsRemote goodsRemote = (GoodsRemote)
                                PortableRemoteObject.narrow(object，GoodsRemote.class);
                        log(goodsRemote.toString());
                }
        }
        catch(Exception e)
        {
                e.printStackTrace();
        }
}

private void log(String message)
{
        if (message.length() > MAX_OUTPUT_LINE_LENGTH)
        {
                System.out.println("-- "
                        + message.substring(0，MAX_OUTPUT_LINE_LENGTH) + " ...");
        }
        else
        {
                System.out.println("-- " + message);
        }
}
/**Main method*/

public static void main(String[] args)
{
        GoodsTestClient client = new GoodsTestClient();
        client.find();
        // Use the getHome() method of the client object to call Home interface
```

```
            // methods that will return a Remote interface reference.   Then
            // use that Remote interface reference to access the EJB.
            // For quick access to all available records，you can use the
            // executeFindAll() method in this class.
        }
    }
```

在程序清单 3.16 中，我们仿照 executeFindAll()方法，定义了一个 find()方法，在 find()方法体中，利用了 Home Interface，调用 findAll()方法，然后再对返回的结果集进行处理。程序清单 3.16 十分简单，我们就不多做解释了。程序清单 3.16 的运行输出如下所示：

```
name:小楼一夜听春雨
type:兵器
id:22
price:3000.8
off:0.8
comment:小楼一夜听春雨，深巷明朝卖杏花，好优美的诗意啊，可惜它是一把弯刀，又名圆月弯刀，曾经是青青所用之物，后归丁鹏所有，和万水青山踏遍齐名
is session ejb:false
1
name:小楼一夜听春雨
type:兵器
id:22
price:3000.8
off:0.8
comment:小楼一夜听春雨，深巷明朝卖杏花，好优美的诗意啊，可惜它是一把弯刀，又名圆月弯刀，曾经是青青所用之物，后归丁鹏所有，和万水青山踏遍齐名
```

好了，关于 Goods EJB 的开发我们就介绍到这里，至于如何分发 Goods EJB 到应用程序服务器上，如何编写 Entity EJB 的 JSP 客户端，请读者参考第 2 章的相关内容，限于篇幅，我们就不多说了。说到这里，想必读者对于 EJB 的强大功能及其开发方法已经有了一定程度的了解，但是 EJB 的技术体系博大精深，不是我们这两章书就能够概括的了的，读者如果希望对 EJB 技术有进一步的了解，关键之处在于多加练习，还需要多参考相关的英文文档。

3.4　本 章 小 结

本章主要介绍了两种 Entity EJB 的开发方法，并比较了它们的异同，这两种 Entity EJB 分别是 CMP 模式的 Entity EJB 与 BMP 模式的 Entity EJB，本章的第三节，改写了 Goods EJB，使得它封装了 JDBC 数据源的功能。

在第 4 章中，我们将介绍 RMI、CORBA、JNDI 等技术在 JSP 程序中的应用。

第 4 章　JSP 与 J2EE 分布式处理技术

在 Java 平台上，有三大分布式处理技术，其一就是我们前两章所介绍的 EJB 组件技术，除此之外，就是 CORBA 技术(for java)和 RMI 技术，在本章中，我们将简要介绍 CORBA 和 RMI 应用的基础知识及开发方法，并试图把它们和 JSP 技术结合起来，编写功能强大的应用程序。

本章需要重点掌握的内容有：

- J2EE 平台的结构
- RMI 模型的开发方法和应用
- CORBA 的结构模型、开发方法和应用
- JNDI 技术简介

4.1　J2EE 和分布式处理技术

4.1.1　J2EE 体系结构

近来的一段时间，J2EE 这个词似乎十分热门，相信有很多读者都对 J2EE 技术不甚了了，那么什么是 J2EE？J2EE 是 Java 2 Enterprise Edition 的缩写，意为 Java 2 平台企业版。J2EE 是一种利用 Java 2 平台来简化诸多与多级企业解决方案的开发、部署和管理相关的复杂问题的体系结构。J2EE 技术的基础就是核心 Java 平台或 Java 2 平台的标准版(简称为 J2SE)，J2EE 不仅具有了 J2SE 中的许多优点，例如“编写一次、到处运行”的特性、访问数据库存取数据的 JDBC API、CORBA 技术等等，同时还提供了对 EJB（Enterprise JavaBeans）、JMS、JTS、Java Servlet API、JSP 以及 XML 技术的全面支持。J2EE 平台的 Servlet、RMI、JDBC 和 Java IDL 等强大的分布式计算技术使 Java 具备了在企业计算中一展身手的能力。一般认为，现代的企业计算解决方案除了企业的业务逻辑（business logic）外，还需要提供对 8 种基本服务的支持，这些服务分别是：

（1）命名/目录服务（Naming and Directory Service）。在企业范围内的信息共享（包括计算机、用户、打印机、应用程序等所有资源）过程中，名/目录服务扮演着重要的角色，它维护着名字和实际资源之间的连接关系，以便使用者能通过名字实现对实际资源的透明访问。在一个大型的企业中可能有多种命名/目录服务并存，如何将所有的命名/目录服务在现有的计算架构完整地统一起来，以便让应用程序透明地访问所有的命名/目录服务，就成了企业计算解决方案必须解决的问题之一。

（2）数据访问服务(Data Access Service)。大部分的应用都需要访问数据库。企业计算解决方案必须提供一种统一的界面对所有的数据库进行访问。

（3）分布式对象服务(Distributed Object Service)。在一个分布式的环境中，构成整个应用的所有组件可能位于不同的服务器上，这些组件通常通过 CORBA 进行互联。在企业计算环境里必须提供一种统一的方法访问这些组件，以保护企业的投资。

（4）企业管理服务(Enterprise Management Service)。在任何分布式环境中，管理都是一个关键的需求，需要应用程序提供远程管理的能力，以防止出现非法操作和访问以及对设备、应用程序状态的管理。

（5）事务处理服务(Transaction Processing Service)。一些关键部门在日常工作中有大量的事务处理，事务处理的开发是一件极其复杂的工作。

（6）消息服务(Messaging Service)。在一些企业，特别是一些对同步传输要求不高的企业通常采用消息机制在应用程序之间进行通讯。

（7）安全服务(Security Service)。应用程序的安全性是任何企业都关心的焦点之一，任何企业计算解决方案都不能回避。

（8）Web 服务(Web Service)。大部分企业级应用都以 Web 服务器为中心。Web 服务成为企业计算解决方案的重要组成部分之一。

为了实现上述服务，在 J2EE 平台中，包含了若干个新的 API，与这些服务一一对应，J2EE 的重要组成部分如下所示：

- JDBC(Java Database Connectivity)提供了连接各种数据库的统一接口。
- EJB(Enterprise JavaBeans)使得开发者方便地创建部署和管理跨平台的基于组件的企业应用。
- Java RMI(Java Remote Method Invocation)用来开发分布式 Java 应用程序。一个 Java 对象的方法能被远程 Java 虚拟机调用，这样，远程方法激活可以发生在对等的两端，也可以发生在客户端和服务器之间，只要双方的应用程序都是用 Java 写的。
- Java IDL(Java Interface Definition Language)提供与 CORBA(Common Object Request Broker Architecture)的无缝集成，这使得 Java 能够集成异构的商务信息资源。
- JNDI(Java Naming and Directory Interface)提供从 Java 平台到系统的无缝连接，这个接口屏蔽了企业网络所使用的各种命名和目录服务。
  在一个企业内部往往多种命名/目录服务并存，如何将它们统一起来，以使企业在开发应用程序时能忽略各命名／目录服务之间的区别而透明地访问这些命名／目录服务，成为企业需要考虑的问题。JNDI 技术解决了这一难题。JNDI 技术命名／目录服务本身的特性出发提出了一些统一的访问名／目录服务的界面，由各命名／目录服务开发商对这些界面加以实现，使用者就可以通过这些统一的界面去访问底层的各种服务。因此在 JNDI 技术中其实存在两种角色：一个是 JNDI API，另一个是 JNDI SPI(Service Provider Interface)。SPI 是各命名／目录服务开发商对 JNDI API 的实现，以便用户通过 JNDI API 访问他们所提供的服务。
  通过 JNDI API，开发人员只要了解 JNDI 就可以开发适用于所有实现了 JNDI 的命名/目录服务的应用，保证了应用程序的可移植性，且免去了大量的学习精力。
- JMAPI(Java Management API)为异构网络上的系统、网络和服务管理的开发提供一整套丰富的对象和方法。JMAPI 是一组与协议无关的 API，开发的应用不但可以在不同的硬件平台上运行，而且适用于多种网管协议。
- JMS(Java Message Service)提供企业消息服务，如可靠的消息队列、发布和订阅通讯，以及有关推拉(Push/Pull)技术的各个方面。

消息机制在一些异步传输系统中被普遍使用。同 JNDI 服务相似，市场上有许多命名 / 目录服务产品实现这种应用，如 Tibco、MessageQ 等。JMS 将这些所有的产品统一于一个界面，且获得了大量的中间件厂商的支持，包括 BEA System、Etsee Soft、IBM MQSeries、Novell 和 Sybase 等在内的中间件厂商都已宣称要实现 JMS，有的如 Etsee Soft、BEA WebLogic 已经实现。

- JTS(Java Transaction Service)提供存取事务处理资源的开发标准，这些事务处理资源包括事务处理应用程序、事务处理管理程序。

短短的几年多时间，Java 已经从当初只能开发 applet 等小小的应用、发展到今天在企业计算中的大量运用。相信随着 Java 技术的进一步发展，一定会有越来越多的企业受益于 Java 技术革命带来的好处。

- JSA(Java Security API)。企业计算的安全性主要体现在 4 个方面：认证（Authentication）、授权(Authorization)、信息在传输过程中的完整性(Integrity)和私有性(Privacy)。Java 2 平台提供了包括公钥（public key）加密、数字签名（DSA）等功能在内的加密框架，实现各加密功能，以保证 Java 计算的安全性。

综合 JTS、JMS、JNDI 等 API 的特点可以发现，J2EE 平台所包含的 Java Enterprise API 不是自己去重新开发实现这些应用系统，而只是在这些产品之上又增加了一个抽象层，以此来提升 Java 和各类应用软件在企业中的应用，保证了用 Java 开发的企业计算应用能够在多种平台上运行。

4.1.2　目前流行的分布式计算解决方案

目前市场上流行的分布式计算解决方案除了 EJB 技术之外，还有 CORBA 技术、RMI 技术、微软的 COM/DCOM 技术了，在这一小节中，我们就对这 3 种解决方案做一个简单的对比。首先是 CORBA 技术。

CORBA 技术

CORBA 构件模型的底层结构为 ORB。一个 CORBA 构件采用 IDL 进行描述。CORBA 提供了 IDL 到 C、C ＋＋、Java、COBOL 等语言的映射机制——IDL 编译器。IDL 编译器可以生成 Server 方的 Skeleton(框架) 和 Client 方的 Stub(存根)代码，通过分别与客户端和服务端程序的联编(指的是 Stub、Skeleton 代码分别与客户端程序、服务端程序的联编)，即可得到相应的 Server 和 Client 程序。

CORBA 提供了一系列的公共对象服务规范——COSS(Common Object Service Specification)，其中包括名字服务、永久对象服务、生命周期服务、事务处理服务、对象事件服务和安全服务等，它们相当于一类用于企业级计算的公共构件。此外，CORBA 还针对电信、石油等典型的应用行业提供了一系列的公共设施。

CORBA 是一种语言中性的软件构件模型，可以跨越不同的网络、不同的机器和不同的操作系统，实现分布对象之间的互操作。

CORBA 有几个基本的优点：与开发语言无关的独立性，与开发者无关的独立性和与操作系统无关的独立性。CORBA 的 ORB 在当前每一种主流操作系统上均有实现 (仅就 Microsoft 的各种操作系统来说，CORBA 获得的支持甚至超越了微软自己推荐的 DCOM 模

型)。除此之外，CORBA ORB 可以访问多种语言实现的对象（包括 C++、COBOL、Smalltalk 和 Java）。借助于 CORBA 的 IIOP 协议(CORBA's Internet Interoperability Protocol)，某一开发者（比如说我）开发的 CORBA ORB 能够获取并操作存在于远程机器上的由其他的开发者（比如说你）开发的分布式对象。Java ORB 允许客户端在没有安装任何特别软件的情况下实现 Java 客户端应用程序（Java ORB 的类可以像 Java Applet 一起动态下载，也可能与浏览器捆绑在一起）。

DCOM 技术

目前，Microsoft 的分布式组件对象模型 DCOM（Distributed Componont Object Model）仅运行于 Windows 操作系统之上。Microsoft 正在与第三方开发商协作，以将 DCOM 移植到其它的操作系统上(包括 MVS 和几种 UNIX 操作系统)。像 CORBA 一样，DCOM 是独立于语言的，它用 Microsoft 的对象描述语言 ODL)通过接口对对象加以描述。

与 CORBA 相比，DCOM 有三个重大缺点。首先，它由单一开发者(微软)定义并控制，这大大限制了 DCOM 使用者的选择范围(比如 DCOM 开发工具)。其次，DCOM 缺乏众多的平台支持，这极大程度地制约了代码的可重用性和 DCOM 应用的可扩展性。最后，与 CORBA 相比，DCOM 是一种相对不十分成熟的技术。尽管微软目前正为 DCOM 加入消息和事务支持，但这些功能在 1994 年的 CORBA 2.0 就已经实现了，并且正由几家不同的 CORBA 软件开发商所发行。

为了使得应用程序能够访问服务端的 DCOM 对象，开发者不得不使用 Internet Explorer 浏览器和 Windows 平台，只有这样才能支持 DCOM 的客户端程序。这样的限制当然削弱了 DCOM 客户端应用程序的可用性。而另一方面，DCOM 的一个重大优势在于，对 Microsoft Windows 的用户是免费的。DCOM 的发展经历了一个相当曲折的过程。DCOM 起源于动态数据交换(DDE)技术，通过剪切/粘贴(Cut/Paste)实现两个应用程序之间共享数据的动态交换。对象连接与嵌入(OLE)就是从 DDE 模型中引伸而来的。随后，Microsoft 引入了构件对象模型——COM，形成了 COM 对象之间实现互操作的接口标准。COM 模型规定了对象模型和编程要求，使 COM 对象可以与其他对象相互操作。这些对象可以用不同的语言实现，其结构也可以不同。基于 COM，微软进一步将 OLE 技术发展到 OLE2。其中 COM 实现了 OLE 对象之间的底层通信工作，其作用类似于 CORBA/ORB。不过此时的 COM 只能作用在单机 Wintel 平台上。在 OLE2 中，也出现了我们今天熟知的拖放技术以及 OLE 自动化。与此同时，微软在 VB 中引入了可以嵌入任何可视构件的通用模型 VBX(OCX 的前身)。VBX 的主要局限在于它并不是一个开放的结构，也没有为第三方软件开发商提供 VBX 集成的标准。最后，微软将上述思想集中在一起，以 COM 作为构件之间通信框架。VBX 也发展为 OLE 控件 OCX 的形式。DCOM 是 COM 在分布计算方面的自然延续，它为分布在网络不同节点的两个 COM 构件提供了互操作的基础结构，而所有以 OLE 为标志的技术如今也已挂上了 ActiveX 标志。

从 CORBA 的观点来看，我们可以粗略地说，ActiveX 控件与 DCOM 的关系相当于 CORBA 构件与 ORB 的关系。当然，按照微软一贯的产品开发逻辑，微妙的思想都退到了幕后，而提供给开发者的是一个以 Wizard 方式生成各种应用的可视化开发环境。在公共服务方面，微软提出了自己的事务服务器 MTS(Microsoft Transaction Server)和消息队列服务

器 MSMQ(Microsoft Message Queue Server)。 前者与 CORBA 对象事务服务目标类似，后者则是为了保证应用系统之间进行可靠的消息通讯和管理。 此外，微软在网络安全方面也有一整套实用的解决方案。

RMI 技术

按照 Javasoft 对 Java 的界定，Java 是一个应用程序开发平台，它按照高性能、可移植、可解释的原则，提供面向对象的编程语言和运行环境。Java 计算的本质就是利用分布在网络中的各类对象协同完成相应的任务。EJB 组件模型就是一个最有说服力的例子，还有：Java Applet 可按用户的需求从服务器上动态地下载到客户机的浏览器上，完成 HTML 页 面的动态变化。

远程方法调用机制——RMI 是构成 Java 分布对象模型的基础结构。RMI 系统包括存根/框架层、远程引用层和传输层。目前，RMI 的传输层是基于 TCP 协议实现的，将来的 RMI 体系结构将建立在 IIOP 协议之上，可以实现 Java 技术与 CORBA 技术的深层融合。应用层建立在 RMI 系统之上。

RMI(Remote Method Invocation) 是 JDK 中的重要特色。RMI 使得 Java 客户能够访问远程机器上的服务对象。这听起来似乎十分类似于 CORBA，但两者并不一样。其关键在于服务器端的应用程序也必须用 Java 编写，且只能使用 JDK 中提供的工具或者与 JDK 兼容的工具。你根本无法把过去编制的代码加到新程序中去，除此之外，RMI 还有许多其它缺陷。与 CORBA 不同，RMI 没有服务这一概念。另外，根据 RMI 写出的 Java 服务端对象往往性能低劣，这个缺点源于 Java 虚拟机(有趣的是，Java CORBA 服务器比 RMI 服务器表现出更好的性能)。RMI 也不包括像 CORBA ORB 那样的对象激活功能。

实际上，RMI 及 Java 技术更可能向 OMG 的标准靠拢，而不是背道而驰。Sun 已经宣布，Java 事务服务（Java Transaction Services）将建立在 OMG 的对象事务服务（Object Transaction Services）。该公司还曾发布其长远计划：使 RMI 对象可以通过 IIOP 协议相互通讯(现在已经实现了，通过 RMI_IIOP 桥)。

总而言之，RMI 对于用纯 Java 书写的小规模的应用程序来说，是一种可行方案。但 CORBA 提供了集成的基础，这种集成是指新开发的代码和已有对象的集成，同时允许将来加以扩展。在做出取此舍彼的选择之前，你必须权衡上面的各种因素，并仔细审视每种技术的现状。

4.1.3　JSP/Java 分布式计算模型

长期以来，Client/Server 结构一直是企业计算中使用最普遍的计算架构之一。但广大用户在享受 C/S 结构带来的利益时，也忍受着越来越大的成本投入和使用与管理上的麻烦。

在 C/S 结构当中，几乎所有的应用逻辑都在客户端实现，导致客户端的应用程序越来越复杂，也给开发人员带来了大量的移植工作；同时要维护如此“肥”且节点众多的客户机更是一件庞杂的工作。

Java 给上述问题的解决带来了曙光。跨平台和从服务器下载执行等特性使 Java 技术在企业计算中得到越来越多的应用。C/S 正逐渐退出舞台，代之而起的是一种新的分布式计算架构：三层次（3—tier）企业计算架构。

三层架构中处于第一层的是客户端表示层。与 C/S 结构中的“肥”客户端不同，客户层仅仅是整个应用系统中的图形用户界面表示，不表示任何应用逻辑，其某些运行代码可以从位于第二层的 Web 服务器下载到本地的浏览器中执行(如 Java Applet 小程序)，几乎不需要任何管理工作，这就是我们平时说的“瘦”客户机，JSP/Servlet 程序就是客户端表示层的代表。处于第二层的是应用服务层，由一或多台服务器（Web 服务器也位于这一层）组成，处理企业应用中的所有业务逻辑和对数据库的访问等工作，该层具有良好的可扩充性，可以随着应用的需要任意增加服务器的数目（构成“宽”的服务器层），EJB 组件、CORBA 应用、RMI 应用就是第二层的代表。处于第三层的是数据中心层，由数据库系统和企业中的已有系统组成。

比三层计算更新的概念是多层计算，主要是将三层计算中的应用服务层继续细分为 Web 服务器层和应用服务层，Web 服务器不再处理任何业务逻辑，而只是将处理请求转发到相应的应用服务单元，由应用服务器调用中间件具体实现商业逻辑，并把处理结果通过 Web 服务器发送到客户端表示层。

在多层计算中，用户的开发工作主要集中在应用服务层进行，许多中间件(Middleware)厂商推出了各自的应用服务器中间件产品，负责提供较底层的服务如负载均衡、状态监测等，以便用户能够将大部分精力集中在业务逻辑的开发。

Java 为多层计算中的每个层次都提供了强大的技术。Java 2 平台的 JFC 所提供的客户端技术，成为开发客户端图形用户界面的首选，JSP/Servlet 程序功能强大，运行效率高，能够快速生成动态页面，反馈回客户端。在应用服务层和数据中心层，包括 EJB、RMI、JDBC、Java IDL 在内的四种技术能够让企业快速地开发他们的分布式应用；对于市场上各式各样的中间件产品，Java 也提供了独特而有效的解决方案。Java 技术正逐渐成为企业计算的潮流。

当前有许多企业采用 CORBA 作为它们的分布式计算对象模型，它们需要 Java 提供开发 CORBA 应用的能力。Java IDL 正是为了满足这一需求而推出的，Java 成为又一种能够开发 CORBA 对象的语言。

Java IDL 与 CORBA/IIOP2.0 规范兼容，包括一个 ORB（Object Request Broker）和一个 IDL 语言编译器用于将用 IDL 语言定义的界面编译成 Java 源代码，然后用 Java 来实现该 CORBA 对象。Java IDL 还提供了一个名服务器用于通过名称寻找 CORBA 对象。开发人员通过 Java IDL API 可以快速开发 CORBA 应用。

上面所介绍的就是 Java 的分布式计算模型，实际上，JSP 的分布式计算模型和 Java 的分布式计算模型没有什么两样，它们的第二、第三层都是一样的，只是在第一层客户端表示层有些小的区别。在 JSP 的分布式计算模型中，客户端表示层是 JSP/Servlet 程序，也许还有 XML、XLST 程序，而在 Java 的分布式计算模型中，客户端表示层是 Java Applet，或者是基于 JFC 组件的 Java Application。JSP 分布式计算模型中的客户端表示层一般是在服务端运行的，而 Java 分布式计算模型中的客户端表示层一般是在客户端运行的，其运行环境是浏览器或这 JRE。

下面我们就来详细描述一下 JSP/Java 分布式计算模型的各个细节部分。

客户层（Client Tier）

J2EE 应用可以是基于 Web 的，也可以是不基于 Web 的。在一个基于 Web 的 J2EE 应用中，用户的浏览器在客户层中运行，并从一个 Web 服务器上下载 Web 层中的静态 HTML 页面或由 JSP 或 Servlets 等程序生成的动态 HTML 页面，也许还有 XML、XSL、XSLT 等程序。一个不基于 Web 的 J2EE 应用系统中，一个独立的客户端程序，或者不运行在一个 HTML 页面中，而是运行在其它一些基于网络的系统（比如手持设备或汽车电话）中的 Java 程序，在客户层中运行，并在不经过 Web 层的情况下访问中间件对象(例如 EJB) 。该不基于 Web 的客户端程序可能也包括一个 JavaBeans 类来管理用户输入，并将该输入发送到在企业层中运行的 EJB 对象来处理。根据 J2EE 规范，JavaBeans 类不被视为组件 。为 J2EE 平台编写的 JavaBeans 类有实例变量和用于访问实例变量中的数据的 getXXX() 和 set XXX()方法。以此种方式使用的 JavaBeans 类在设计和实现上通常都是简单的，但是它们必须符合 JavaBeans 规范中列出的命名和设计约定。

Web 层

Web 层是一个特殊的客户层，J2EE Web 层的 Web 组件可以由 JSP/Servlet 程序、基于 Web 的 Java Applets 组成。调用 Servlets 或者 JSP 页面的 HTML 页面在应用程序组装时与 Web 组件打包在一起。就像客户层一样，Web 层可能包括一个 JavaBeans 类来管理用户输入，并将输入发送到在业务层中运行的 EJB 对象来处理 。运行在客户层的 Web 组件依赖 Web 服务器来支持响应客户的请求。

业务层

作为解决或满足某个特定业务领域（比如银行、零售或金融业）需要的逻辑的业务代码由运行在业务层的 EJB 组件或者其他的中间件来执行。一个 EJB 对象从客户程序处接收数据，对数据进行处理（如果需要），再将数据发送到企业信息系统层(可能是数据库系统)存储。一个 EJB 对象还可以从企业信息系统层中检索数据，并将数据送回客户程序。运行在业务层的 EJB 对象依赖于 EJB 容器来为诸如事务、生命期、状态管理、多线程及资源存储池提供通常都非常复杂的系统级代码。

业务层经常被称作 Enterprise JavaBeans（EJB）层。业务层构成了 3 层 J2EE 应用的中间层，而其它两层是客户层和企业信息系统层。

查询服务（lookup services）

因为一个 J2EE 应用程序的组件是单独运行的，并且往往在不同的设备上运行，因此，需要一种能让客户层和 Web 层代码查询并引用其他代码和资源的方法。客户层和 Web 层代码使用 Java 命名和目录接口（JNDI ）来查询用户定义的对象（例如 EJB）、环境条目（例如一个数据库驱动器的位置）、企业信息系统层中用于查找资源的 JDBC DataSource 对象，以及消息连接。

安全和事务管理（Security and Transaction Management）

诸如安全和事务管理这样的应用行为可以在部署时在 Web 和 EJB 组件上进行配置。这

个特征将应用逻辑从可能随装配而变化的配置设定中分开了。

安全

J2EE 安全模型允许配置一个 Web 或 EJB 组件，使系统资源只能由授权的用户访问。例如，一个 Web 组件可以被配置成提示输入用户名和密码。一个 EJB 组件可以被配置成只让特定团体中的成员调用其某些方法。或者，一个 Servlet 程序可以被配置成让某个组织中的所有人都能访问其某些方法，同时只让该组织中的某些享有特权的人访问一些方法。同样是该 Servlet 程序，可以针对另外一个环境而被配置成让每个人都能访问其所有方法，或者仅让选定的少数人访问其所有方法 。

事务管理(Transaction Management)

J2EE 事务模型使得能够在部署时定义构成一个单一事务的方法之间的关系，以使一个事务中的所有方法被处理成一个单一的单元。这是我们所希望的，因为一个事务是一系列步骤，这些步骤要么全部完成，要么全部取消。例如，一个 EJB 对象可能有一组方法，使我们可以通过从第一个账户借出并存入第二个账户的方式而将钱从第一个账户转移到第二个账户。我们希望全部的操作被作为一个单元对待，这样，如果在借出之后存入之前发生了故障，该借出操作被取消。事务属性是在装配期间定义在一个组件上的。这使得能将来自多个应用组件的方法归到一个事务中，这说明，我们可以轻易变更一个 J2EE 应用程序中的应用组件，并重新指定事务属性，而不必改变代码或重新编译 。在设计应用组件时，要记住，尽管 EJB 对象有一个可使应用组件的容器自动启动多步事务的机制，但是 Java Applet 和应用的客户容器可能并不支持这一点。然而，Java Applet 和应用客户容器总是能够调用支持这一点的一个 EJB 对象。还应当注意，JSP 和 Servlets 程序没有被设计成是支持事务的，它们通常应当将事务工作交给一个 EJB 对象来完成。然而，如果事务工作在一个 JSP 或 Servlet 程序中是必需的，那么此种工作也应当是非常有限的 。

可再用应用组件（Reusable Application Components ）

J2EE 组件（Java Applets、EJB 组件、JSP 、Servlets）都被打包成模块，并以 Java Archive（JAR）文件的形式交付。一个模块由相关的组件、相关的文件及描述如何配置组件的配置描述文件组成。例如，在组装过程中，一个 HTML 页面和 Servlet 程序被打包进一个模块之中，该模块包含该 HTML 文件、Servlet 程序及相关的配置描述文件，并以一个 Web Archive（WAR）文件的形式交付，该 WAR 文件是一个带.war 扩展名的标准 JAR 文件。使用模块使得利用相同的组件中的某些组件来组装不同的 J2EE 应用程序成为可能 。例如，一个 J2EE 应用程序的 Web 版可能有一个 EJB 组件，还有一个 JSP 程序。该 EJB 组件可以与一个应用客户组件结合，以生成该应用程序的非 Web 版本。这不需要进行额外的编码，只是一个装配和部署的问题。并且，可再用组件使得将应用开发和部署过程划分成由不同的角色来完成成为可能，这样不同的人或者公司就能完成封装和部署过程的不同部分。

设计用户界面和引擎（Designing the User Interface and Engine ）

在为 J2EE 应用程序设计用户界面和后端引擎时，我们需要决定是让该程序基于 Web，

还是不基于 Web。在做出这个决定时，我们可能希望考虑平台配置、下载速度、安全、网络流量和网络服务。例如，包含有用户界面并且经常被大量用户访问的一个 Java Applet 可能需要花很长的时间才能被下载下来，这让用户沮丧。然而，如果知道该 Java Applet 要运行在公司的内部网内的受控环境中，那么，在这种情况下，该 Applet 将拥有一个完全可接受的下载速度。

注意： 所谓的三层模型有多种看法，上面的叙述只是代表我们的看法。有人认为 JSP/Servlet 程序应该算是业务层，也就是第二层，我们认为，应该把商业逻辑从 JSP/Servlet 程序中最大限度的剥离，让 JSP/Servlet 程序只负责数据的显示与接受用户请求，这样它们就是属于客户端表示层的了。

4.2　远程方法调用——RMI 技术

在这一节中我们将介绍 RMI 技术的基础知识、结构、开发和应用。

4.2.1　RMI 概述

在大型的企业计算中，需要将整个应用系统分解成若干子系统并运行于相应的部门当中，以提高整个系统的应用效率和可维护性及安全性。由于不同的工作由不同的系统来完成，在它们之间需要一种安全便利的通讯机制。从面向对象的角度来说，企业计算需要一种分布式的对象模型，使得运行于不同主机之间的对象能够互相进行方法调用。RMI 正是基于 Java 的能够满足上述计算的分布式计算模式，实现了在运行于不同虚拟机的对象之间的方法调用。

RMI 的研究工作其实在 1994 年（Java 诞生之前）就已在 Sun 公司的实验室里开始了。研究人员在充分评估了分布计算的趋势并比较了多种分布式系统以后开发了 RMI 对象模式，在 1997 年 JDK1.1 发布时正式推出，成为 Java 分布式计算的重要技术之一。JDK1.1 为 RMI 的开发提供了一系列的 API，简化了 RMI 的开发工作。

从调用方式来看，RMI 和 RPC（远过程调用）有很大的相似之处，但二者之间最大的不同在于：RMI 是面向对象，而 RPC 是基于过程调用的。由于 RMI 面向对象的特性，RMI 调用可以直接将对象在调用的两端之间进行传递，不但可以传送数据，而且还可以传递方法(mobile behavior)，扩展了 RMI 的使用；另外 RMI 还支持两个 RMI 对象之间的方法回调(callback)。

RMI 赋予每个 RMI 对象一个唯一的名字，并将之与实际的对象捆绑在一起（binding），这种对象关系登记在 RMI 的注册登记表中，调用者通过对象的名字找到相应的对象后调用它的方法，而不需要考虑该对象的实际物理存储位置。这不但更符合人们的使用习惯，而且提高了系统的可扩充性和鲁棒性。RMI 将多个 RMI 对象的名字登记在同一张登记表中（监听于某端口），一个对象有一或多个方法以供远程调用，从而使一个端口可以提供多种服务，节约了系统的端口资源。

目前，RMI 不限于用 Java 语言编写的 RMI 对象之间的调用，它们之间通过 JRMP（Java Remote Method Protocol，译为 Java 远程方法协议）进行通信；对于另一种对象模式 CORBA，RMI 已经实现同 CORBA 对象之间的互相调用，由 Sun 公司开发的进行二者之间

相互通讯的产品（RMI over IIOP）已经正式发布。

4.2.2　开发 RMI 应用

在这一小节中，我们将介绍如何开发 RMI 应用。要开发 RMI，必须构造 4 个主要的类，它们分别是：远程对象的本地接口、RMI 客户、远程对象实现和 RMI 服务程序。RMI 服务程序生成远程对象实现的一个实例，并用一个特殊的 URL 注册它，RMI 客户在远程服务器上查找对象，若找到就把它转换成本地接口类型，然后像一个本地对象一样使用它。下面是一个简单的 RMI 例子，远程对象只返回一个消息字符串。要使这个例子更有价值，我们需要做的就是完善远程对象实现类。

远程对象的本地接口

该类仅仅是一个接口，而不是实现，它声明了 RMI 客户端程序可以调用的方法。RMI 客户机可以直接使用它，RMI 服务程序必须产生一个远程对象来实现它，并用某个 URL 注册它的一个实例。请看程序清单 4.1，这是一个简单的远程对象的本地接口。

程序清单 4.1

```
//File Name:Rem.java
//Author:fancy
//Date:2001.5
//Note:Remote interface
import java.rmi.*;
public interface Rem extends Remote
{
public String getMessage() throws RemoteException;
public String getAuthor() throws RemoteException;
}
```

在编写远程对象的本地接口的时候，读者应该注意本地接口（Rem）必须是公共的，否则客户机在加载一个实现该接口的远程对象时就会出错。此外，它还必须从 java.rmi.Remote 接口继承而来，远程对象的本地接口中的每一个方法都必须抛出远程异常 java.rmi.RemoteException。

远程对象实现类

这个类真正实现 RMI 客户所调用的远程对象本地接口，它必须从 UnicastRemoteObject 继承，其构造函数应抛出 RemoteException 异常，而且每一个实现方法都必须抛出 RemoteException 异常。请看程序清单 4.2。

程序清单 4.2

```
//File Name:RemImpl.java
//Author:fancy
//Date:2001.5
//Note:implemention remote interface

import java.rmi.*;
```

```
import java.rmi.server.UnicastRemoteObject;
public class RemImpl extends UnicastRemoteObject implements Rem
{
    public RemImpl() throws RemoteException
    {
    }
    public String getMessage() throws RemoteException
    {
        return("Here is a remote message.");
    }
    public String getAuthor() throws RemoteException
    {
        return("fancy.");
    }
}
```

RMI 服务器类

该类将创建远程对象实现 RemImpl 的一个实例，然后用一个特定的 URL 来注册它，所谓注册就是通过调用 Naming.bind()方法或 Naming.rebind()方法来将 RemImpl 的实例对象绑定到特定的 URL 上。 请看程序清单 4.3。

程序清单 4.3

```
//File Name:RemServer.java
//Author:fancy
//Date:2001.5
//Note: remote server

import java.rmi.*;
import java.net.*;

public class RemServer
{
    public static void main(String[] args)
    {
        try
        {
            RemImpl localObject = new RemImpl();
            Naming.rebind("rmi://localhost/Rem", localObject);
        }
        catch(RemoteException re)
        {
            System.out.println("RemoteException:"+re);
        }
        catch(MalformedURLException mfe)
```

```
            {
                System.out.println("MalformedURLException: "+mfe);
            }
        }
    }
```

RMI 客户类

RMI 客户使用 Naming.lookup()方法在指定的远程主机上查找对象，若找到就把它转换成本地接口类型(在本例中是 Rem 类型)，然后像一个本地对象一样使用它。与 CORBA 不同之处在于 RMI 客户必须知道提供远程服务主机的 URL，这个 URL 可以通过 rmi://host/path 或 rmi://host:port/path 来指定，如果省略端口号，就使用 1099。Naming.lookup()方法可能产生 3 个异常：RemoteException、NotBoundException、MalformedURLException，这 3 个异常都需要捕获。RemoteException、Naming 和 NotBoundException 在 java.rmi 包中定义，MalformedURLException 在 java.net 包中定义，所以我们要在应用程序中导入这两个包，另外，客户机将向远程对象传递串行化对象 Serializable，所以还应在程序中导入 java.io 包。请看程序清单 4.4。

程序清单 4. 4

```
//File Name: RemClient.java
//Author:fancy
//Note:rmi client
//Date:2001.5

import java.rmi.*;
import java.net.*;
import java.io.*;

public class RemClient
{
    public static void main(String[] args)
    {
        try
        {
            String host = (args.length > 0) ? args[0] : "localhost";
            //通过URL在远程主机上查找对象，并把它转化为本地接口Rem类型
            Rem remObject=(Rem)Naming.lookup("rmi://" + host + "/Rem");
            System.out.println(remObject.getMessage()); //调用远程对象的方法
            System.out.println(remObject.getMessage()); //调用远程对象的方法
        }
        catch(RemoteException re)
        {
            System.out.println("RemoteException: " + re);
        }
```

```
            catch(NotBoundException nbe)
            {
                System.out.println("NotBoundException: " + nbe);
            }
            catch(MalformedURLException mfe)
            {
                System.out.println("MalformedURLException:"+ mfe);
            }
        }
    }
```

如上所述，RMI 应用已经编写好了，下面我们应该编译运行这个 RMI 应用，步骤如下：

（1）编译 RMI 客户和服务器，这将自动编译远程对象的本地接口和远程对象实现，命令行代码如下

```
javac RemClient.java
javac RemServer.java
```

（2）生成客户存根模块(stub)和服务器框架(skeleton) 。

```
rmic RemImpl
```

这将构造 RemImpl_Stub.class 和 RemImpl_Skeleton.class。

（3）请将 Rem.class、RemClient.class 和 RemImpl_Stub.class 拷贝到 RMI 客户机，将 Rem.class、RemImpl.class、RemServer.class 和 RemImpl_Skeleton.class 拷贝到 RMI 服务器。

（4）启动 RMI 注册程序。在命令行状态下，运行 rmiregistry 程序，rmiregistry 程序在 JDK 的 bin 文件夹内。这个程序在服务器上执行，不论有多少个 RMI 远程对象需要运行，本操作只需做一次。

（5）运行 RMI 服务程序。在命令行下执行下面的命令：

```
java RemServer.class
```

这一步的作用是启动 RMI 服务器(在服务器上执行) 。

在命令行下执行下面的命令：

```
java RemClient.class
```

这一步的作用是启动 RMI 客户程序(在客户端计算机上执行)。输出结果为：

```
Here is a remote message.
fancy
```

4.2.3　使用 JSP 编写 RMI 应用客户端

我们可以编写基于 JSP 的 RMI 应用客户端，使得在 JSP 程序中也能够调用远程 RMI 对象的商业方法，这种技术无疑大大地扩展了 JSP 程序的能力，也是利用 JSP+J2EE 技术构建企业应用系统中关键的一环。

实际上，基于 JSP 的 RMI 应用客户端可以从程序清单 4.4 改写而来，请看程序清单 4.5。

程序清单 4.5

```
<%--
File Name:rem.jsp
Author:fancy
```

```
Date:2001.5
Note:the client for rmi object
--%>

<%@ page import=" java.rmi.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="Rem.class" %>
<%
try
{
    String host = "localhost";
    //通过URL在远程主机上查找对象，并把它转化为本地接口Rem类型
    Rem remObject=(Rem)Naming.lookup("rmi://" + host + "/Rem");
    out.println(remObject.getMessage()+"<br>"); //调用远程对象的方法
out.println(remObject.getMessage()); //调用远程对象的方法
}
catch(RemoteException re)
{
    out.println("RemoteException: " + re);
}
catch(NotBoundException nbe)
{
    out.println("NotBoundException: " + nbe);
}
catch(MalformedURLException mfe)
{
    out.println("MalformedURLException:"+ mfe);
}
%>
```

程序清单 4.5 和程序清单 4.4 的输出结果一致。

4.3　CORBA 技术

4.3.1　CORBA 技术简介

公用对象请求代理程序体系结构（Common Object Request Broker Architecture），缩写为 CORBA，是对象管理组织（Object Management Group，简称为 OMG）对应当今快速增长的软硬件的协同工作能力的要求而提出的方案。简而言之，CORBA 允许应用程序和其他的应用程序通讯，而不论它们在什么地方或者由谁来设计。CORBA 1.1 标准由对象管理组织(OMG)在 1991 年发布。它定义了接口定义语言（IDL）和应用编程接口（API），从而通过实现对象请求代理（ORB，Object Request Broker）来激活客户/服务器的交互。CORBA 2.0 标准于 1994 年的 12 月发布。它定义了如何跨越不同的 ORB 提供者而进行通讯。

ORB (Object Request Broker)是一个中间件，它在对象间建立客户-服务器的关系。通过 ORB，一个客户可以很简单地使用服务器对象的方法而不论服务器是在同一机器上还是通过一个网络访问。ORB 截获调用，然后负责找到一个对象实现这个请求，传递参数和方法，最后返回结果。客户不用知道对象在哪里，是什么语言实现的，远程对象所在的操作系统以及其它和远程对象接口无关的东西。

在传统的客户/服务器程序中，开发者使用他们自己设计的或者公认的标准定义设备/对象之间通信的协议。协议的定义依赖于实现的语言，网络的传输和其他许许多多因素。ORB 将这个过程简单化。使用 ORB，协议定义是通过应用接口，而该接口是接口定义语言（IDL）的一个实现，它和使用的编程语言无关的。并且 ORB 提供了很大的灵活性。它让程序员选择最适当的操作系统，运行环境和设计语言来建设系统中每个组件。更重要的是，它允许集成已经存在的组件。

CORBA 是在面向对象标准化和互操作性道路上的一个信号。通过 CORBA，用户不必要知道软硬件的平台和他们处在企业网的什么地方就可以操作。

Java 提供了一个概念清晰、结构紧凑的分布计算模型和构件互操作的方法，为构件应用开发提供了相当的灵活性。但由于它还处于发展时期，因此其形态很难界定。CORBA 是一种集成技术，而不是编程技术。它提供了对各种功能模块进行构件化处理并将它们捆绑在一起的粘合剂。Java 技术和 CORBA 技术在很大的程度上可以看作是互补的。为了适应 Web 应用的发展要求，许多软件厂商都急于促成 CORBA 技术和 Java 技术的结合，RMI 可以建立在 IIOP 协议之上。CORBA 不只是对象请求代理 ORB，也是一个非常完整的分布式对象平台。CORBA 可以扩展 Java 在网络、语言、组件边界、操作系统中的各种应用。其次，Java 也不仅仅是与 CORBA 捆绑的语言，它还是一个易变的对象系统，也是一个运行对象的便携式 OS，而且允许 CORBA 对象在主机、网络计算机和蜂窝电话等应用上运行。Java 还简化了大型 CORBA 系统中的代码分配，其中内置的多线程和垃圾收集使编写可靠的网络对象更为方便。此外，Java 能创建可移动对象并将它们分发出去，而采用 CORBA 可以将它们连接在一起，并与计算环境中的数据库、遗留系统、其他语言编写的对象和应用相互集成。

CORAB 与 Java 的基础结构也可以很好地互补。CORBA 处理网络透明性，Java 处理实现透明性。CORBA 提供了 Java 便携应用环境与对象间的连接。由此看来 CORAB/Java 技术紧密结合的趋势是势不可挡的。而且二者的结合将成为 Object Web 技术的主要形态之一。

4.3.2　CORBA 模型

OMG 是一个开放组织，成立于 1989 年，到今天已有 600 多名成员，其主要目标是：使用对象技术，使基于对象的软件成员在分布的异构环境中可重用、可移植、可互操作。OMG 成立不久便制定了对象管理体系结构（Object Management Architecture，OMA），它由以下 4 个部分组成。

- 对象请示代理 ORB（Object Request Broker），使对象在分布式环境中透明地收发请求和响应。它是构造分布式对象应用，使应用在不同的层次的异构环境下相互操作的基础。

- 对象服务 OS(Object Services)，是为使用和实现对象而提供的基本服务集合。对于分布式应用来说是基本服务，同时这些服务独立于应用领域。如生命循环服务定义了对象创建、删除、拷贝、移动的方式，它不考虑对象在应用中如何实现的。
- 通用设施 CE(Common Facilites)，是许多应用提供的共享服务集合，如系统管理、电子邮件等。
- 应用对象 AO(Application Objects)，它相当于传统的应用，应用对象可由独立的厂商生产，是参考模型中的最高层，OMG 对此不作规定。

由此可见，ORB 在对象管理系统中处于最关键的位置，是对象间的“通信总线”。为 ORB 制定的规范称为 CORBA 规范。CORBA 是一种规范，它定义了分布式对象如何实现相互操作。在 World Wide Web 盛行之前，特别是 Java 编程语言风靡之前，C＋＋开发者基本将 CORBA 作为其高端分布式对象的解决方案。最初，OMG 在 1990 年制订了对象管理体系（Object Management Architecture，即 OMA，来描述应用程序如何实现互操作）的时候，作为其中(OMA)的一部分，需要有一个标准规范应用程序片段即对象的互操作，这导致了 CORBA 的诞生。OMA 定义了组成 CORBA 的 4 个主要部分，分别与上面提到过的 OMA 的 4 个部分相对应，它们是：

- Object Request Broker：ORB，作为对象互通讯的软总线。
- CORBA Services：CORBA 服务，定义加入 ORB 的系统级服务，如安全性、命名和事务处理，是开发分布式应用所必需的模块。这些服务提供异步事件管理，事务、持续、并发、名字、关系和生存周期的管理。
- CORBA Facilities： CORBA 工具，定义应用程序级服务，如复合文档等，对于开发分布式应用不是必需的，但是在某些情况下是有用的。这些工具提供信息管理、系统管理、任务管理和用户界面。
- Business Objects 或者是 Application Objects： 商业对象或者应用对象，主要用于定义现实世界的对象和应用，如飞机或银行帐户。

上述 4 部分的关系如图 4.1 所示。

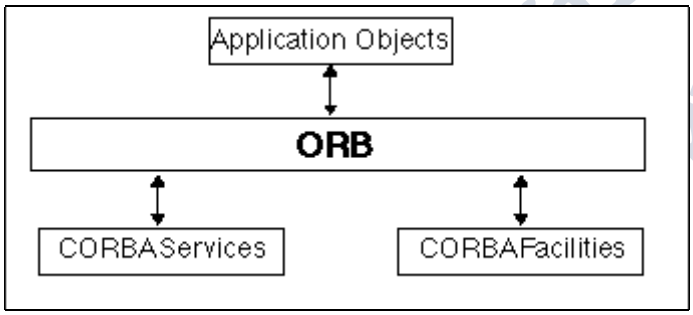


图 4.1　CORBA 的组成

CORBA 组件包括 ORB 核心、ORB 接口、IDL 存根(stub)、DII(Dynamic Invocation Interface，动态调用接口)、对象适配器、IDL 框架(skeleton)、DSI(Dynamic Skeleton Interface，动态框架界面)。CORBA 运行结构（ORB 核心）是由特定开发商决定的，不是由 CORBA 定义的。不管怎样，ORB 接口是一个标准的功能接口，是由所有的 CORBA 兼容 ORB 提

供的。IDL 编译器为每个界面产生存根(stub)。这就屏蔽了低层次的通信，提供了较高层次的对象类型的特定 API。DII 是相对于 IDL 存根的另一种方法，它为运行时构建请求提供了一个通用的方法。对象适配器，为把可选的对象技术集成进 OMA(Object Management Architecture)提供支持。IDL 框架类似于 IDL 存根，但是，它是工作于服务器端的（对象实现）。对象适配器发送请求给 IDL 框架，然后，IDL 框架就为之调用合适的对象实现中的方法。

CORBA 接口和数据类型是用 OMG 接口定义语言(OMG IDL)定义的。每个接口方法也是用 OMG IDL 声明的。IDL 是 CORBA 体系结构的关键部分，它为 CORBA 和特定程序设计语言的数据类型之间提供映射(例如从 CORBA 到 Java 之间的映射)。IDL 也允许对原有代码实行封装。IDL 是一个面向对象的接口定义语言，具有和 C++相类似的语法。由于所有的接口都是通过 IDL 定义的，CORBA 规范允许客户端程序和服务端对象用不同的程序设计语言书写，彼此的细节都是透明的。CORBA 在不同对象请求代理(ORB)之间使用 IIOP 协议进行通信，使用 TCP 协议作为网络通信协议。

使用接口描述语言(IDL，Interface Description Language)编写的对象接口，使得与语言无关的独立性成为可能。IDL 使得所有 CORBA 对象以一种方式被描述，仅仅需要一个由本地语言（C/C++、CORBA、Java）到 IDL 的“桥梁”。

CORBA 被设计和架构为服务端对象使用不同程序语言书写、运行于不同平台上的对象系统。CORBA 依赖于 ORB 中间件在服务器和客户之间进行通信。ORB 扮演一个透明地连接远程对象的对象。每个 CORBA 对象提供一个调用接口，并且有一系列的方法与之相联。ORB 负责为请求发现相应的实现，并且把请求传递给 CORBA 服务器。ORB 为客户提供透明服务，客户永远都不需要知道远程对象的位置以及用何种语言实现的。OMA 最重要的部分就是 ORB。为了创建一个遵从 CORBA 规范的应用程序，ORB 是 CORBA 4 大部分中唯一必须提供的。许多 ORB 产品根本不带 CORBA Services 或是 CORBA Facilities，你可以自己编写商用对象。但是，没有 ORB，CORBA 应用程序绝对无法工作。

既然 ORB 如此重要，那么我们就来讨论一些 ORB 的细节问题。CORBA ORB 最显而易见的功能，就是对你的应用程序或是其它 ORB 的请求予以响应。在 CORBA 应用程序运行期间，你的 ORB 可能被请求做许多不同的事情，包括：

- 查找并调用远程计算机上的对象。
- 负责不同编程语言之间的参数转换（如 C＋＋到 Java）。
- 可超越本机界限的安全管理。
- 为其它的 ORB 收集并发布本地对象的元数据(metadata)。
- 用下载的客户端存根代码（stub）中描述的静态方法调用去激活远程对象中的方法。
- 用动态方法调用激活远程对象。
- 自动激活一个当前没有装入内存运行的对象。
- 将回调方法导引向其(当前 ORB 对象)管理之下的本地对象。

实现细节对软件开发者的透明性，是 ORB 的一个杰出的特性。用户只须在代码中提供相应的 hooks(钩子)，用于初始化 ORB 并向 ORB 登记该应用程序，就可以将该应用程序和大量分布式对象建立联系。

目前，对于较为流行的编程语言（包括 C＋＋，Smalltalk，Java 和 Ada 95），已经有了

许多第三方的 ORB。随着其他语言的逐渐流行，CORBA 开发商毫无疑问地要做出相应的 ORB 来支持它们。

下面我们简单描述 CORBA 应用的工作流程，如图 4.2 所示。

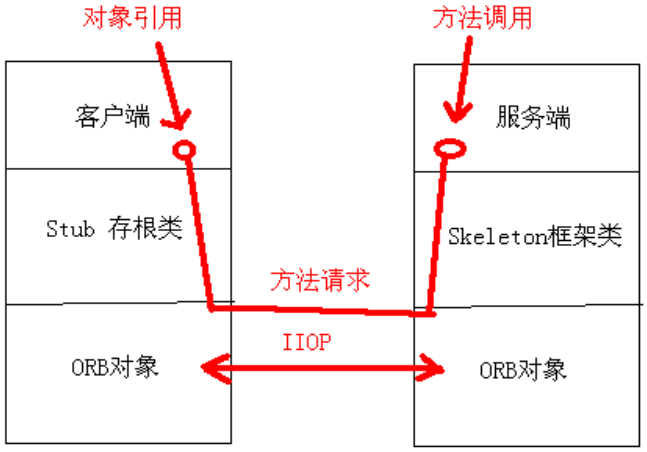


图 4.2　CORBA 的工作模型

图 4.2 说明了从客户端到服务器发送一个消息请求的过程，其中客户端也可以是一个 CORBA 对象，客户机不需要了解 CORBA 对象的位置与实现细节，也不需要了解哪个 ORB 用于存取对象。在客户端，应用系统包括远程对象的引用，对象引用使用 stub 类(存根)方法作为远程方法的代理，这个方法事实上是在 ORB 中的，所以调用 stub 类方法会自动调用 ORB 对象的连接功能，ORB 对象会把对 stub 类方法的调用传递到服务器端。

在服务器端，ORB 对象利用框架类(skeleton)代码把远程调用转换成本地对象的方法调用，框架类需要对调用和参数的格式进行转换。同时，当方法返回时，框架类对结果进行变换，然后通过 ORB 对象把结果返回给客户机。

4.3.3　接口定义语言 IDL 以及 Java 语言映射

Java IDL（Interface Definition Language）可实现网络上不同平台上的对象相互之间的交互，该技术基于通用对象请求代理体系结构 CORBA 规范说明。IDL 是不依赖于语言的接口定义语言，所有支持 CORBA 的语言都有 IDL 到该语言的映射。就像其名字所表示的那样，Java IDL 支持 IDL 到 Java 语言的映射。CORBA 规范说明和 IDL 映射是由 OMG 组织（Object Management Group）定义的，Sun 公司是其成员之一，它在定义 IDL 到 Java 映射的工作中起了主要作用。

为了保持 CORBA 的商业中立性和语言中立性，必须有一个中介，存在于像 C++ CORBA 服务器代码和 Java CORBA 客户机这样的实体之间，这就是 IDL。一个底层对象的若干相关方法和属性被 IDL 封装为一个单一的接口。一旦 IDL 接口定义完成，它可以以 stub 类(存根类)或 skeleton(框架类)的形式编译成你所使用的开发语言。在所有的 ORB 中都有 IDL 编译器，例如，JDK 中提供了 idlj.exe 编译器。

有一点值得注意的是 IDL 不同于其它的面向对象程序设计语言，我们不能用它具体实

现它所声明的类或者是方法，因此，将它仅仅作为一种定义底层对象接口的语言要好得多。就像在 Java 中将属性和方法封装到相关的类中一样，上述各项均包含在 IDL 的模块之中。在一个模块之中可以定义一个或多个接口。

接口定义语言 IDL 用于定义 CORBA 对象的接口，所有的 CORBA 对象都必须支持 IDL。IDL 的语法和 C++的语法非常类似，利用 Java IDL，可以在 Java 中定义、实现、存取 CORBA 对象。对于每个 IDL，idlj.exe 程序生成一个 Java 接口和其他一些必要的.java 文件，包括一个客户端 stub（Stub）和服务器端 skeleton（Skeleton）。

为了使 Java 支持 CORBA 规范说明，需要一个把 IDL 中的元素映射为 Java 中元素的标准途径，Sun 已经制订了两者之间的映射规则，并提供了编译器 idlj.exe，以便从 IDL 得到相应的 stub 类和 skeleton 类。

表 4.1 列出了一些 IDL 中元素和 Java 中元素的映射关系。

表 4.1　IDL 和 Java 的映射关系

IDL 元素	Java 元素
Module	package
Interface	Interface，help class，hold class
Constant	public static final
Boolean	boolean
Char，wchar	Char
String，wstring	java.lang.String
Short，unsigned short	Short
Long，unsigned long	Long
Float	float
Double	double
Enum，struct，union	class
Sequence，array	array
Read only attribute	method for accessing value of attribute
Read write attribute	Methods for accessing and setting value of attribute
Operation	Method

下面我们就来定义一个简单的 IDL 程序，请看程序清单 4.6(hello.idl)。

程序清单 4.6(hello.idl)

```
module HelloApp
{
    interface Hello
    {
        string sayHello();
    };
};
```

hello.idl 中声明了 sayHello()方法，sayHello()方法的返回值是一个字符串。我们使用 IDL

编译器 idlj.exe 程序(idlj.exe 程序是 JDK 自带的程序，在 bin 文件夹中)把这个 hello.idl 文件映射为 Java 源代码文件，用以下命令编译 Hello.idl 文件：

idlj hello.idl

根据命令行中的不同选项，idlj.exe 程序将产生不同的文件，上面这条命令将创建子目录 HelloApp 并在其中产生 5 个文件：HelloOperations.java、_HelloStub.java、Hello.java、HelloHelper.java、HelloHolder.java。

这 5 个文件的程序清单如程序清单 4.7～4.11。

程序清单 4.7(HelloOperations.java)

```
package HelloApp;
/**
* HelloApp/HelloOperations.java
* Generated by the IDL-to-Java compiler (portable)，version "3.0"
* from hello.idl
* 1999年3月27日  18时22分21秒  CST
*/
public interface HelloOperations
{
    String sayHello ();
} // interface HelloOperations
```

程序清单 4.8(_HelloStub.java)

```
package HelloApp;

/**
* HelloApp/_HelloStub.java
* Generated by the IDL-to-Java compiler (portable)，version "3.0"
* from hello.idl
* 1999年3月27日  18时22分21秒  CST
*/
public class _HelloStub extends org.omg.CORBA.portable.ObjectImpl
    implements HelloApp.Hello
{
    // Constructors
    // NOTE:   If the default constructor is used，the
    //             object is useless until _set_delegate (...)
    //             is called.
    public _HelloStub ()
    {
        super ();
    }

    public _HelloStub (org.omg.CORBA.portable.Delegate delegate)
```

北京希望电子出版社

```
{
    super ();
    _set_delegate (delegate);
}

public String sayHello ()
{
    org.omg.CORBA.portable.InputStream _in = null;
    try
    {
        org.omg.CORBA.portable.OutputStream _out = _request ("sayHello"，true);
        _in = _invoke (_out);
        String __result = _in.read_string ();
        return __result;
    }
    catch (org.omg.CORBA.portable.ApplicationException _ex)
    {
        _in = _ex.getInputStream ();
        String _id = _ex.getId ();
        throw new org.omg.CORBA.MARSHAL (_id);
    }
    catch (org.omg.CORBA.portable.RemarshalException _rm)
    {
        return sayHello ();
    }
    finally
    {
        _releaseReply (_in);
}
    } // sayHello

    // Type-specific CORBA::Object operations
    private static String[] __ids = {"IDL:HelloApp/Hello:1.0"};
    public String[] _ids ()
    {
        return (String[])__ids.clone ();
    }

    private void readObject (java.io.ObjectInputStream s)
    {
        try
        {
            String str = s.readUTF ();
```

北京希望电子出版社

```
                org.omg.CORBA.Object obj =
                    org.omg.CORBA.ORB.init ().string_to_object (str);
                org.omg.CORBA.portable.Delegate delegate =
                    ((org.omg.CORBA.portable.ObjectImpl) obj)._get_delegate ();
                _set_delegate (delegate);
            }
            catch (java.io.IOException e)
            {
            }
        }

        private void writeObject (java.io.ObjectOutputStream s)
        {
            try
            {
                String str = org.omg.CORBA.ORB.init ().object_to_string (this);
                s.writeUTF (str);
            }
            catch (java.io.IOException e)
            {
            }
        }
} // class _HelloStub
```

程序清单 4.9(Hello.java)

```
package HelloApp;

/**
* HelloApp/Hello.java
* Generated by the IDL-to-Java compiler (portable)，version "3.0"
* from hello.idl
* 1999年3月27日  18时22分21秒  CST
*/

public interface Hello extends HelloOperations，org.omg.CORBA.Object，
    org.omg.CORBA.portable.IDLEntity
{
} // interface Hello
```

程序清单 4.10(HelloHelper.java)

```
package HelloApp;

/**
* HelloApp/HelloHelper.java
* Generated by the IDL-to-Java compiler (portable)，version "3.0"
```

北京希源电子出版社

```
* from hello.idl
* 1999年3月27日  18时22分21秒  CST
*/

abstract public class HelloHelper
{
    private static String  _id = "IDL:HelloApp/Hello:1.0";

    public static void insert (org.omg.CORBA.Any a，HelloApp.Hello that)
        {
        org.omg.CORBA.portable.OutputStream out = a.create_output_stream ();
        a.type (type ());
        write (out，that);
        a.read_value (out.create_input_stream ()，type ());
    }

    public static HelloApp.Hello extract (org.omg.CORBA.Any a)
    {
        return read (a.create_input_stream ());
    }

    private static org.omg.CORBA.TypeCode __typeCode = null;
    synchronized public static org.omg.CORBA.TypeCode type ()
    {
        if (__typeCode == null)
        {
            __typeCode = org.omg.CORBA.ORB.init ().create_interface_tc
                (HelloApp.HelloHelper.id ()，"Hello");
        }
        return __typeCode;
    }

    public static String id ()
    {
        return _id;
    }

    public static HelloApp.Hello read (org.omg.CORBA.portable.InputStream istream)
    {
        return narrow (istream.read_Object (_HelloStub.class));
    }

    public static void write (org.omg.CORBA.portable.OutputStream ostream，
```

北京希望电子出版社

```
            HelloApp.Hello value)
        {
            ostream.write_Object ((org.omg.CORBA.Object) value);
        }

        public static HelloApp.Hello narrow (org.omg.CORBA.Object obj)
        {
            if (obj == null)
                return null;
            else if (obj instanceof HelloApp.Hello)
                return (HelloApp.Hello)obj;
            else if (!obj._is_a (id ()))
                throw new org.omg.CORBA.BAD_PARAM ();
            else
            {
                org.omg.CORBA.portable.Delegate delegate =
                    ((org.omg.CORBA.portable.ObjectImpl)obj)._get_delegate ();
                return new HelloApp._HelloStub (delegate);
            }
        }
}
```

程序清单 4.11(HelloHolder.java)

```
package HelloApp;

/**
* HelloApp/HelloHolder.java
* Generated by the IDL-to-Java compiler (portable)，version "3.0"
* from hello.idl
* 1999年3月27日  18时22分21秒  CST
*/

public final class HelloHolder implements org.omg.CORBA.portable.Streamable
{
  public HelloApp.Hello value = null;

  public HelloHolder ()
  {
  }

  public HelloHolder (HelloApp.Hello initialValue)
  {
    value = initialValue;
  }
```

北京希望电子出版社

```
    public void _read (org.omg.CORBA.portable.InputStream i)
    {
      value = HelloApp.HelloHelper.read (i);
    }

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

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

idlj.exe 程序自动产生的代码只是一个基本的框架，我们若要完成该应用程序，还要分别在这 5 个文件的基础上提供服务器端和客户机端的实现。

至于如何开发一个完整的 CORBA 应用，我们将在下一小节中进行介绍。

4.3.4　编写 CORBA 应用

在这一小节中，我们将描述如何开发一个完整的 CORBA 应用系统，利用 Java IDL 开发 CORBA 应用系统一般可分为下面的 5 个步骤：

定义远程接口

用 IDL 定义远程对象的接口，使用 IDL 而不是 Java 语言是因为 idlj.exe 编译器可以自动地从 IDL 文件中产生 Java 语言的 stub 和 skeleton 源文件，以及和 ORB 对象连接时所需要的一些代码。使用 Java IDL，开发人员可以用 Java 语言来实现客户机和服务器。如果要为一个已经存在的 CORBA 服务实现客户机，或为一个已经存在的客户机实现服务，则首先要给出 IDL 接口，然后运行 idlj.exe 编译器产生 stub 和 skeleton，在此基础上再编码实现。

编译远程接口

对 IDL 文件运行 idlj.exe 编译器，产生 Java 版本的接口，以及 stub 和 skeleton 代码文件，这些代码文件使得应用程序可以和 ORB 相连接。

实现服务器

把 idlj.exe 编译器产生的 skeleton 和服务器应用程序集成在一起，除了要实现远程接口中的方法之外，服务器代码还要包括启动 ORB 以及等待远程客户机的调用等部分。

CORBA 服务器程序的结构和大部分 Java 应用程序的结构一样，即导入所需要的包，声明服务器类，定义 main()方法，处理一些例外等。另外，CORBA 服务器要有一个 ORB 对象，每个服务器实例化一个 ORB 对象，并向其注册服务对象，因此当 ORB 对象接收到

调用请求时可以寻找到 CORBA 服务器。最后是 CORBA 服务对象的管理，服务器是一个进程，实例化了一个或多个服务对象，CORBA 服务对象具体实现接口中说明的操作。CORBA 服务对象和命名服务一起工作，使得 CORBA 服务对象对于客户机来说是可用的，CORBA 服务器需要对名字服务的对象引用，可以向名字服务注册，并保证向 CORBA 接口的调用被路由到其服务对象上，最后是等待客户机的调用。

实现客户机

类似地，以 stub 作为客户端应用程序的基础，客户机建立在 stub 之上，通过 Java IDL 提供的名字服务查询服务器，获得远程对象的引用，然后调用远程对象中的方法。

CORBA 客户机的结构和大部分 Java 应用程序的结构基本相似，即导入所需要的包、声明应用类、定义 main()方法、处理一些例外等。另外和服务器程序一样，一个 CORBA 客户机也需要本地的 ORB 对象来执行所有的配置工作，每个客户机实例化一个 org.omg.CORBA.ORB 对象，然后向该对象传递一些必要的信息以进行初始化，最后是利用 ORB 对象取得所需要的服务。一旦一个应用程序有了 ORB 对象，即可通过 ORB 对象来确定应用所需要的服务的位置。为了调用 CORBA 对象中的方法，客户端应用要有对该对象(指的是 CORBA 对象)的引用，有很多种方法可以得到这种引用，如命名服务等方法。

启动应用程序

一旦实现了服务器和客户机，就可以启动名字服务，接着启动服务器，然后运行客户机。

下面我们简要介绍一下开发 CORBA 应用的步骤。

（1）使用 IDL 创建 CORBA 接口。程序清单 4.12 使用 OMG IDL 描述一个 CORBA 对象。

程序清单 4.12(Show.idl)

```
module About
{
    interface Show
    {
        string ShowName();
    };
};
```

将其存为 Show.idl 文件。

（2）编译接口并生成 CORBA 支持文件。在 MS-DOS 状态下，我们用以下命令编译这个 IDL 文件：

```
idlj.exe Show.idl
```

idlj.exe 是 Sun 公司的 IDL 编译器，包含在 JDK 中，你可以在 bin 文件夹中找到它。

编译后将在当前目录下生成 About 子目录，其中会包括一些支持文件，这些文件和程序清单 4.7——程序清单 4.11 十分相似，读者如有兴趣可以看一下，但一定不要修改。

（3）实现服务器(ShowServer.java)。请看程序清单 4.13(showServer.java)。

程序清单 4.13(showServer.java)

```
import About.*;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import org.omg.CORBA.*;

public class ShowObject extends ShowOperations
{
public String ShowName()
    {
        return "\nMy name isfancy!!\n";
    }

}

public class ShowServer
{
    public static void main(String args[])
    {
        try
        {
            // 创建和初始化 ORB
            ORB orb = ORB.init(args，null);
            // 创建服务对象并将其向 ORB 注册
            ShowObject ShowRef = new ShowObject();
            orb.connect(ShowRef);
            // 获取根命名上下文
            org.omg.CORBA.Object objRef =
                orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            // 绑定命名中的对象引用

            NameComponent nc = new NameComponent("About"，"");
            NameComponent path[] = {nc};
            ncRef.rebind(path，ShowRef);
            // 等待来自客户机的调用

            java.lang.Object sync = new java.lang.Object();
            synchronized (sync)
            {
                sync.wait();
            }
        }
```

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

ShowServer 类的 main()方法，可完成以下任务：

1）创建一个 ORB 实例(orb) ，如下面的代码。

```
ORB orb = ORB.init(args，null);
```

2）创建一个服务对象实例（CORBA About 对象的实现）并通知 ORB 对象，如下面的代码。

```
ShowObject ShowRef = new ShowObject();
orb.connect(ShowRef);
```

3）获取一个命名上下文的 CORBA 对象引用，在该命名上下文中注册新的 CORBA 对象。

4）在命名上下文中将新对象注册在“About”名下，如下面的代码。

```
NameComponent nc = new NameComponent("About"，"");
NameComponent path[] = {nc};
ncRef.rebind(path，ShowRef);
```

5）等待客户端程序对新对象的调用。

（4）实现客户机 (ShowClient.java)

请看程序清单 4.14(ShowClient.java)。

程序清单 4.14(ShowClient.java)

```
import About.*;
import org.omg.CosNaming.*;
import org.omg.CORBA.*;

public class ShowClient
{
    public static void main(String args[])
    {
        try
        {
            // 创建和初始化 ORB
            ORB orb = ORB.init(args，null);
            // 获取根命名上下文
            org.omg.CORBA.Object objRef =
            orb.resolve_initial_references("NameService");
            NamingContext ncRef = NamingContextHelper.narrow(objRef);
            //解析命名中的对象引用
            NameComponent nc = new NameComponent("About"，"");
```

北京希望电子出版社

```
                NameComponent path[] = {nc};
                About.Show ShowRef = ShowHelper.narrow(ncRef.resolve(path));
                // 调用 Show 服务对象并打印结果
                String show = ShowRef.ShowName();
                System.out.println(show);
            }
            catch (Exception e)
            {
                System.out.println("ERROR : " + e) ;
                e.printStackTrace(System.out);
            }
        }
}
```

在 ShowClient 类中完成了以下任务：

1）创建一个 ORB，如下面的代码。

```
ORB orb = ORB.init(args，null);
```

2）获取一个指向命名上下文的引用，如下面的代码。

```
org.omg.CORBA.Object objRef =
orb.resolve_initial_references("NameService");
NamingContext ncRef = NamingContextHelper.narrow(objRef);
```

3）在命名上下文中查找“Show”并获得指向该 CORBA 对象的引用，如下面的代码。

```
NameComponent nc = new NameComponent("About"，"");
NameComponent path[] = {nc};
About.Show ShowRef = ShowHelper.narrow(ncRef.resolve(path));
```

4）调用对象的 ShowName() 操作并打印结果，如下面的代码。

```
String show = ShowRef.ShowName();
System.out.println(show);
```

（5）构建和运行 ShowName 程序。

1）编译所有的 java 文件，包括在 About 目录中 CORBA 支持文件(stub 和 skeleton)。

```
javac *.java
```

2）启动一个 MS-DOS 窗口，输入以下命令，确保 JNDI 命名服务器处于运行状态：

```
tnameserv -ORBInitialPort 1050
```

3）启动另一个 MS-DOS 窗口，输入以下命令，启动 ShowServer 服务程序：

```
java ShowServer -ORBInitialPort 1050
```

4）再启动一个 MS-DOS 窗口，运行客户机的 ShowClient 应用程序：

```
java ShowClient -ORBInitialPort 1050
```

ShowClient.java 程序的输出为：

```
My name is fancy!
```

这说明 CORBA 服务已经测试成功了。

4.3.5　使用 JSP 编写 CORBA 应用客户端

和 RMI 技术一样，我们同样可以使用 JSP 编写 CORBA 应用的客户端程序，就以上文编写的 ShowServer CORBA 服务为例，我们编写一个基于 JSP 的客户端程序，请看程序清单 4.15。

程序清单 4.15

```
<%--
File Name:show.jsp
Author:fancy
Date:2001.5
Note:jsp client for corba
--%>

<%@ page import=”About.*” %>
<%@ page import=”org.omg.CosNaming.* ”%>
<%@ page import=”org.omg.CORBA.*”%>

<%
try
{
    // 创建和初始化 ORB
    ORB orb = ORB.init(args，null);
    // 获取根命名上下文
    org.omg.CORBA.Object objRef =orb.resolve_initial_references("NameService");
    NamingContext ncRef = NamingContextHelper.narrow(objRef);
    //解析命名中的对象引用
    NameComponent nc = new NameComponent("About"，"");
    NameComponent path[] = {nc};
    About.Show ShowRef = ShowHelper.narrow(ncRef.resolve(path));
    // 调用 Show 服务对象并打印结果
    String show = ShowRef.ShowName();
    out.println(show);
}
catch (Exception e)
{
    out.println("ERROR : " + e) ;
    e.printStackTrace(System.out);
}
%>
```

不难看出 show.jsp 和 ShowClient.java 程序的代码十分相似，实际上，ShowClient.java 程序只要稍微修改一下就是 show.jsp 程序了，两者之间没有本质的区别，所以，只要我们熟悉 JSP 的语法，并且掌握基于 Java Application 的 CORBA 客户端程序的编写方法，那么

我们就不难编写出基于 JSP 的 CORBA 客户端程序。

4.4　JNDI 技术

在本书中，我们已经多次提到，而且还将多次提到 JNDI 这个术语，那么 JNDI 到底是什么概念呢?JNDI 指的是 Java Naming and Directory Interface，亦即 Java 命名和目录服务接口。

在介绍 JNDI 之前，我们先让读者对 Naming 与 Directory 有最基本的认识，进而了解使用 JNDI 的原因。

Naming 在电脑系统上是最基本的服务之一，借助名字(names)与对象 (objects)之间的绑定 (binding)，对象使用者通过正确地描述上下文环境(Context)来存取调用欲使用的对象。例如：DNS 服务器将主机名称 Rainbow.pku.edu.cn 对应到 IP 地址 162.105.106.162，以方便使用者记忆。

Directory 可视为 Naming 概念的扩充，让特定对象拥有属性 (attributes)以记录额外的信息。于是，我们可以通过命名来查看(look up)对象并获得该对象的属性信息，或是利用属性作为搜寻的过滤条件 (search filter)。Directory 目前在电脑系统上较著名的应用系统与架构有 Novell 公司的 NDS(Novell Directory Services)、Sun 公司的 NIS(Network Information Service)，与即将成为网络上一个新标准的 LDAP(Lightweight Directory Access Protocol) 。

JNDI 有什么作用呢?利用 JNDI，我们可以在 Java 程序中(包括 Application、Applet、JSP、Servlet、EJB、RMI 等)存取或检索在 Directory 内的对象及其属性。例如以 Java 开发的电子邮件客户端程序能够通过 Directory 的方式来管理其中的通讯录；然而借助扩充 Directory 的信息，例如将打印的服务对象存入 Directory，Java 程序就可以检索网络上符合要求的打印对象，然后把数据传送至该打印对象，再调用物理上的打印机进行打印操作。换言之，支援 Directory 的 Java 程序将能把 Directory 中的对象当成 Java 的对象来使用。

JNDI(Java Naming and Directory Interface)是一套提供 Naming 和 Directory 功能的 API，Java 应用程序开发者通过使用 JNDI，在 Naming 和 Directory 方面的应用上就有了通用的标准。

JNDI 包含一组 API 和一组 SPI(Service Provider Interface)。Java 程序通过 JNDI API 存取各种 Naming 和 Directory 服务；JNDI SPI 则使得各种 Naming 和 Directory 服务透明化，允许 Java 程序通过 JNDI API 来存取这些服务。

4.5　本 章 小 结

本章重点讲述了下面的内容：J2EE 平台的结构；RMI 模型的开发方法和应用；CORBA 的结构模型、开发方法和应用；JNDI 技术简介，与 EJB 技术相比，这些技术的重要性相对较低，所以我们介绍得就比较简略。

在下一章，我们将开始一个新的主题，那就是 XML 技术与 JSP 技术的结合。

第二部分　JSP 技术和 XML 技术

第 5 章　XML 简介

本章前三节主要介绍关于 XML 的基本知识，包括 XML 概念的简单介绍、XML 语法规则的叙述，XML 相对于传统的网页标记语言的突出优点，XML 的文档类型定义（DTD）、XML 的数据模式问题，以及 XML 的形式方面的问题。XML 的形式问题是指 XML 文档的显示，本章主要介绍了风格样式表语言 CSS 以及可扩展的风格样式表语言 XSL。通过这一章的学习，读者应当对 XML 的基本概念和特点有所了解，并且能够书写结构良好的 XML 文档，并将文档通过加入风格样式表文件在浏览器中显示出来。

本章后两节主要介绍了有关 HTML 到 XML 的过渡版本（XHTML）和新的无线应用协议及其脚本语言（WML）的一些知识。XHTML 由 W3C 制定，于 2000 年 1 月成为 W3C 的推荐标准。它是 XML 显示语言的一种，是用 XML 重新定义了的 HTML。与 XHTML 一样，WML 其实也是 XML 的一种显示语言。它基于无线应用协议 WAP 的基础之上，而 WAP 是一种主要是为了实现无线通信工具的快捷的信息获取而制定的协议。WML 就是针对这种要求而设计的，使用无线通信设备浏览 WML 文件。

实际上，本书不应该介绍 XML 语言的语法，但是因为本书的其他部分涉及了很多关于 XML 的知识，并要大量使用 XML，况且 XML 技术已经与 Java/J2EE/JSP 技术紧密地耦合在一起，为了使得本书的体系结构完整起见，我们特意以一章的篇幅来介绍 XML 的基础知识。

XHTML 是 HTML 的扩展，是下一代网页语言的标准，JSP 技术同样可以与 XHTML 结合在一起，XHTML 与 XML 的关系十分密切，所以我们在这里也简单地介绍 XHTML 的概貌，让读者了解这种新技术，实际上，在 XHTML 中使用 JSP 与在 HTML 中使用 JSP 没有任何区别，所以，我们没有就如何使得 XHTML 与 JSP 技术结合做更多的讨论。

WML 是 XML 的一个子集，它使得无线设备也可以像普通的电脑一样，上网浏览页面，只不过这些页面是由 WML 构建的，而不是使用 HTML 的语法。WML 技术也可以与 JSP 技术结合在一起，我们可以使用 JSP 程序生成动态的 WML 文件，再传送到无线设备中。

应当说明，本章所介绍的内容是极其概括与肤浅的，目的在于帮助读者了解本书其它部分的内容，它充其量只能够起到复习的作用，读者千万不要抱有这样的想法，以为读完本章以后就可以掌握 XML、XHTML、WML 了。读者如果希望完整地了解 XML 的全貌，还是需要阅读有关的专著。

5.1　XML 简介及其语法规则

5.1.1　XML 简介

XML 是 eXtensible Markup Language 的缩写，意为可扩展的标记语言。XML 是一套定义语义标记的规则，这些标记将文档分成许多部件并对这些部件加以标识。它也是元标记语言，即定义了用于定义其他与特定领域有关的、语义的、结构化的标记语言的句法语言。XML 由全球互联网协会(World Wide Web Consortium)在 1996 年底提出。有许多人会认为，XML 是由 HTML 所延伸出来的，是 HTML 的增强版。

事实并非如此，XML 与 HTML 有着很大的差异，主要的区别如下：

（1)XML 是元标记语言。XML 不只是像超文本标记语言(Hypertext Markup Language，HTML）或是格式化的程序。这些语言定义了一套固定的标记，用来描述一定数目的元素。如果标记语言中没有所需的标记，用户也就没有办法了。这时只好等待标记语言的下一个版本，希望在新版本中能够包括所需的标记，但是这样一来就得依赖于软件开发商的选择了。

XML 是一种元标记语言。用户可以定义自己需要的标记。这些标记必须根据某些通用的原理来创建，但是在标记的意义上，也具有相当的灵活性。例如，要处理与菜谱有关的事情，需要描述菜的成分、数量、特色、价格、名称等，这就必须创建用于每项的标记。新创建的标记可在文档类型定义（Document Type Definition，在以后的篇幅中常简称为 DTD）中加以描述。现在，只需把 DTD 看作是一本词汇表和某类文档的句法。

XML 定义了一套元句法，与特定领域有关的标记语言如 MusicML、MathML 和 CFML、WML 等都必须遵守。如果一个应用程序可以理解这一元句法，那么它也就自动地能够理解所有的由此元语言扩展建立起来的语言。浏览器不必事先了解多种不同的标记语言使用的每个标记。事实是，浏览器在读入文档或者是它的 DTD 文档时才了解了给定文档所使用的每一个标记。

至于在浏览器中或者在其他设备上如何显示这些标记的内容，是通过附加在文档上的另外的样式单描述语言提供的，如 CSS 或者 XSL。

（2）XML 描述的是结构和语义。XML 标记描述的是文档的结构和意义。它不描述页面元素的格式化。可用样式单(CSS)或者 XSL 为文档增加格式化信息。XML 文档本身只说明文档包括什么标记，而不是说明文档看起来是什么样的。作为对照，HTML 文档包括了格式化、结构和语义的标记。<B>就是一种格式化标记，它使其中的内容变为粗体。<STRONG>是一种语义标记，意味着其中的内容特别重要。<TD>是结构标记，指明内容是表中的一个单元。事实上，某些标记可能具有所有这三种意义。<H1>标记可同时表示 20 磅的 Helvetica 字体的粗体、第一级标题和页面标题。

例如，在 HTML 中，一首歌可能是用定义标题、定义数据、无序的列表和列表项来描述的。但是事实上这些项目没有一件是与音乐有关的。用 HTML 定义的歌曲可能如下：

程序清单 5.1(song.html)

```
<dt>Hot Cop
<dd> by Jacques Morali Henri Belolo and Victor Willis
```

```
<ul>
<li>Producer: Jacques Morali
<li>Publisher: PolyGram Records
<li>Length: 6:20
<li>Written: 978
<li>Artist: Village People
</ul>
```

而在 XML 中，同样的数据可能标记为：

程序清单 5.2(song.xml)

```
<SONG>
<TITLE>Hot Cop</TITLE>
<COMPOSER>Jacques Morali</COMPOSER>
<COMPOSER>Henri Belolo</COMPOSER>
<COMPOSER>Victor Willis</COMPOSER>
<PRODUCER>Jacques Morali</PRODUCER>
<PUBLISHER>PolyGram Records</PUBLISHER>
<LENGTH>6:20</LENGTH>
<YEAR> 978</YEAR>
<ARTIST>Village People</ARTIST>
</SONG>
```

在程序清单 5.2 中没有使用通用的标记如<dt>和<li>，而是使用了具有意义的标记，如<SONG>、<TITLE>、<COMPOSER>和<YEAR>等。这种用法具有许多优点，包括源码易于被人阅读，使人能够看出作者的含义。可以选择 XML 的元素名称，以便使其在附加的上下文中具有额外的意义。例如，元素名称可以是数据库的域名。XML 比 HTML 更为灵活而且适用于各种应用，因为有限数目的标记不必用于许多不同的目的。

5.1.2　XML 的语法规则及其良构性

与 HTML 不同，XML 对于语法有着严格的规定，只有当一个 XML 文档符合“格式良好”的基本要求时，处理程序才能对它加以分析和处理。“格式良好的”这一标准通过对 XML 文档的各个逻辑成分和物理成分进行语法规定，保证了 XML 文档严密的条理性、逻辑性和良好的结构性，从而大大提高了 XML 应用处理程序处理 XML 数据的准确性和效率。实际上，“格式良好”的要求就是 XML 规范的语法要求，一个简单的检验方法就是用 Internet Explorer 5.01 以上版本的浏览器打开正在编辑的 XML 文档，如果报错，这个文档就不是“格式良好的”。XML 文档的结构包括逻辑结构和物理结构。

逻辑结构

一个 XML 文档通常以一个 XML 声明开始，通过 XML 元素来组织 XML 数据。XML 元素包括标记和字符数据。为了组织数据更加方便、清晰，还可以在字符数据中引入 CDATA 数据块，并可以在文档中引入注释。此外，由于有时需要给 XML 处理程序提供一些指示信息，XML 文档中可以还包含处理指令。

具体说来，各个逻辑元素的作用和形式如下：

1. XML 声明

XML 声明是处理指令的一种，一个 XML 文档最好以一个 XML 声明作为开始。下面是一个完整的 XML 声明：

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

在一个 XML 的处理指令中必须包括 version 属性，指明所采用的 XML 的版本号，而且它必须在属性列表中排在第一位。standalone 属性表明该 XML 文档是否和一个外部文档类型定义 DTD 配套使用。encoding 属性则指明了数据所采用的编码标准。如果需要显示中文，那么编码应该是 GB2312 或者 GBK。

2. 元素

元素是 XML 文档内容的基本单元。从语法上讲，一个元素包含一个起始标记、一个结束标记以及标记之间的数据内容。其形式是：

<元素标记>数据内容</元素标记>

对于标记有以下语法规定：

(1) 标记必不可少。任何一个格式良好的 XML 文档中至少要有一个元素。

(2) 大小写有别。

(3) 要有正确的结束标记。结束标记除了要和起始标记在拼写和大小写上完全相同，还必须在前面加上一个斜杠“/”。当一对标记之间没有任何文本内容时，可以不写结束标记，而在起始标记的最后冠以斜杠“/”来确认。这样的标记称为“空标记”。

(4) 标记要正确嵌套。例如：<学生><姓名></学生><姓名>就是一个错误的嵌套。

(5) 标记命名要合法。标记名应该以字母、下划线“_”或冒号“:”开头，后面跟字母、数字、句号“.”、冒号、下划线或连字符“-”，但是中间不能有空格，而且任何标记名不能以“xml”（或者“xml”大小写的任何组合，如“XML”、“xML”、“xmL”等等）起始。

(6) 有效使用属性。标记中可以包含任意多个属性，属性以名称/取值对出现，属性名不能重复，名称与取值之间用等号“=”分隔，且取值用引号引起来。

3. CDATA 节

在标记 CDATA 下，所有的标记、实体引用都被忽略，而被 XML 处理程序当作字符数据看待。CDATA 的形式如下：

<! [CDATA[文本内容]>

CDATA 的文本内容中不能出现字符串“]]>”，CDATA 不能嵌套。

4. 注释

在 XML 中，注释的方法与 HTML 完全相同，用“<! --”和“-->”将注释文本引起来。

对于注释还有以下规定：

(1) 在注释文本中不能出现字符“-”或字符串“—”。

(2) 不要把注释文本放在标记之中，类似地，不要把注释文本放在实体声明之中或之前。

(3) 注释不能被嵌套。例如：<学生><!--学习成绩优秀--></学生>就是错误的。

5. 处理指示

处理指示是用来给处理 XML 文件的应用程序提供信息的。所有的处理指示应该遵循

下面的格式：

<? 处理指示名 处理指示信息？ >

也就是说，XML 分析器可能对它并不感兴趣，而把这些信息原封不动地传给 XML 应用程序。然后，这个应用程序来解释这个指示，遵照它所提供的信息进行处理，或者再把它原封不动地传给下一个应用程序。

实际上前面例子中的 XML 声明就是一个处理指示：

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

6. 样式表声明

从本质上讲，样式表声明和 XML 声明一样，这两者都是处理指示的一种。但只有需要对 XML 文档进行解析和显示的时候，才需要给出这两个处理指示。而且 XML 的语法规定，XML 文档的第一行必须是 XML 声明，两者都出现时文档类型声明和样式表声明必须分别位于第三行和第二行。例如下面的 XML 文档就是格式良好的，具有良构性：

程序清单 5.3(test.xml)

```
<?xml version="1.0" encoding="GB2312" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="mystyle.xsl"?>
<学生>
<姓名>许国华</姓名>
</学生>
```

7. 文档类型声明

为了对 XML 文档进行文档类型定义，在 XML 文档中必须进行文档类型的说明，一般是在 XML、风格样式表声明之后。如下的 XML 文档就是有效的：

```
<?xml version="1.0" encoding="GB2312" standalone="no"?>
<?xml-stylesheet type="text/xsl" href="mystyle.xsl"?>
<!DOCTYPE　根元素名　SYSTEM　"mydtd.dtd" >
<学生>
<姓名>方中星</姓名>
</学生>
```

物理结构

从物理结构上讲，XML 文档是由一个或多个存储单元构成的，这些存储单元就是所谓的实体。所有的 XML 文档都包含了一个“根实体”，又称作“文档实体”。这个实体是由 XML 本身给出的，无需显式定义就可以使用，它指的其实就是整个文档的内容，是 XML 语法分析器处理的起点。除此之外，可能还需要用到其他一些实体，这些实体都用名字来标识，在文档类型定义 DTD 中给出定义。

简单地说，实体充当着和别名类似的角色，即，一个简单的实体名称可以用来代表一大段文本内容。像任何计算机别名系统一样，实体引用简化了录入工作，因为每当要使用同样一大段文本时，只需使用它的别名就可以了，处理器会自动把这个别名替换为相应的文本。

实体引用有以下几点规则：

（1）除了 XML 标准规定的预定义实体以外，在 XML 文档引用一个实体之前，必须

已经对此实体进行过声明。

（2）实体引用中不能出现空格。

（3）尽管在一个实体中可以再引用其他实体，但是不能出现循环引用。

（4）实体引用的文档必须符合 XML 语法的种种要求，任何一个独立的逻辑要素，比如元素、标记、注释、处理指令、实体引用等等，都不能开始于一个实体，而结束于另一个实体。

至于这一部分的详细内容，我们在后面的 DTD 的学习会深入讨论，在这里就不再赘述了。

5.2　DTD 的书写及实例

5.2.1　什么是 DTD

XML 的精髓就是基于信息描述的、能够体现数据信息之间逻辑关系的、可以确保文件的易读性和易搜索性的自定义标记。为支持词汇表，XML 提供了一种定义文档结构与元素的机制，这就是文档类型定义。对 XML 的元素、属性及实体等读者都已经似曾相识了，而 DTD 正是把元素、元素属性、实体等包容进来，给整个文档提供声明保证了文档的有效性。DTD 是 Document Type Definition 的简写，中文意思是文档类型定义。DTD 是“元标记”这个概念的产物，它描述了标记语言的语法和词汇表，也就是定义了文件的整体结构以及文件的语法。简而言之，DTD 规定了一个语法分析器为了解释一个“有效的”XML 文件所需要知道的所有规则的细节。

DTD 可以是一个完全独立的文件，也可以在 XML 文件中直接设定。所以，DTD 分为外部 DTD（在 XML 文件中调用另外已经编辑好的 DTD）和内部 DTD（在 XML 文件中直接设定 DTD）两种。比如，有几十家相互联系的、合作伙伴关系的公司、厂商，他们相互之间的交换电子文档都是用 XML 文档。那么我们可以将这些 XML 文档的 DTD 放在某个地方，让所有交换的 XML 文档都使用此 DTD，这是最方便的做法，同时也适用于公司内部的 XML 文件使用。

5.2.2　一个 DTD 的实例的简单分析

在 XML 所描述的指标语言中，DTD 便提供了语法规定，以便给各个语言要素赋予一定的顺序。为了说明特定的语法规则，DTD 采用了一系列正则式，语法分析器将这些正则式与 XML 文件内部的数据模式相匹配，从而判别一个文件是否是有效的。匹配被严格执行，因此，如果 XML 文件中有任何信息不符合 DTD 的规定，都不会通过。读者可以先看下面一个实例，从后面的简单分析可以了解 DTD 的基本设计格式：(此文件只是 DTD 定义，在浏览器中会显示一个树状列表)。

程序清单 5.4（book.dtd）

```
<!DOCTYPE BOOKS[
<!ELEMENT BOOKS(BOOK+)>
<!ELEMENT BOOK(AUTHOR，TITLE，DESCRIPTION，PRICE)>
<!ATTLIST BOOK language CDATA #REQUIRED>
```

```
<!ELEMENT AUTHOR (#PCDATA)+>
<!ELEMENT DESCRIPTION (#PCDATA)>
<!ELEMENT PRICE(QUANTITY，CURRENCY)>
<!ELEMENT QUANTITY (#PCDATA)>
<!ELEMENT CURRENCY (#PCDATA)>
]>
```

从上面的例子，读者不难看出，DTD 的基本格式是：

<!DOCTYPE　　根元素名　　[……]>

其中!DOCTYPE 是关键字表示文档类型定义开始，而[　　]中则包括了文档类型定义的实际内容。在上面这个例子中，我们还看到了!ELEMENT 、!ALLIST 这两个关键字，它们分别承担了元素组成、类型的定义和元素属性、类型的定义。

5.2.3　DTD 语法

DTD 有 4 种类型的声明，即元素类型声明、属性列表声明、实体声明、符号声明。

元素类型声明（ETD）

下面是两个典型的元素类型声明：

<!ELEMENT STUDENT (NAME，ADDRESS)>

<!ELEMENT STUDENT　(#PCDATA)>

第一个元素类型声明定义了一个<STUDENT>元素，并规定该元素必须有 NAME 和 ADDRESS 两个子元素，而且 NAME 的位置必须在 ADDRESS 的前面；第二个元素类型声明也定义了一个<STUDENT>，并规定该元素由纯文本的一般字符组成。元素类型声明不但说明了每个文件中可能存在的元素，给出了元素的名字，而且给出了元素的具体类型。元素类型声明应该采用如下的结构：

<!ELEMENT 元素名 （元素内容描述）>

上面的例子中读者可以发现一个 XML 元素即可以是子元素的集合也可以是一段纯文本字符。事实上根据 XML 的标准可以将元素按内容划分为 4 类：空元素类型、ANY 元素类型、父元素类型、混合元素类型。

1．空元素类型

空元素类型定义方式为：

<!ELEMENT 元素名 EMPTY>

这类元素在 XML 文档中使用空元素标记，元素中没有内容。例如程序清单 5.5 的 XML 文档就是有效的（该代码可运行）。

程序清单 5.5（student1.xml）

```
<?xml version = "1.0" encoding="GB2312" standalone = "yes"?>
<!DOCUMENT [
<!ELEMENT 学生 EMPTY>
]>
<学生></学生>
```

2. ANY 元素类型

ANY 元素类型定义方式为：

<!ELEMENT 元素名 ANY>

XML 文档里该元素中可以包含任何内容。建议一般只把文档的根元素规定为 ANY 类型。例如程序清单 5.6 的 XML 文档是有效的。

程序清单 5.6(student2.xml)

```
<?xml version = "1.0" encoding="GB2312" standalone = "yes"?>
<!DOCTYPE 学生[
<!ELEMENT 学生ANY >
 ]>
<学生>
</学生>
```

这个 DTD 定义了一个 XML 文件，它只有一个根元素，名为<学生>，这个元素可以有任何类型的子元素，也可以包含纯文本字符，还可以为空元素。上面的 XML 文档中的<学生>元素就为空元素。这容易造成读者的错觉，由于 ANY 元素类型的元素被定义为可以包含其它元素，是不是就可以对要加入这个根元素的子元素不加定义就进行使用。回答是否定的，因为尽管元素<学生>被定义为可以包含其它元素，但实际上这个 DTD 除了<学生>元素本身外没有定义任何其它元素,所以也就没有其它元素可以用作<学生>元素的子元素。“有效的”XML 文件规定文件中所使用的任何元素都必须在 DTD 中给出定义。

不过并不是所有这种根元素的组成部分都需要另加说明，纯文本字符就是一个例外，ANY 定义下的根元素使用纯文本都是不需要说明的。在相同的 DTD 定义下，程序清单 5.7 所示的 XML 文件则是合法的。

程序清单 5.7(student3.xml)

```
<?xml version = "1.0" encoding="GB2312" standalone = "yes"?>
<!DOCTYPE 学生[
<!ELEMENT 学生  ANY     >
 ]>
<学生>
姓名：李超群      性别：男        电话：62764586
</学生>
```

3. 父元素类型

这类元素中可以包含子元素，在 DTD 中通过正则表达式规定子元素出现的顺序和次数。语法分析器将这些正则式与 XML 文档内部的数据模式相匹配，判别出一个文档是否是“有效的”。DTD 尽管要求严格，但也有它的灵活性。使用正则表达式，我们就可以解决上述问题，描述父元素与子元素之间非常复杂的关系。例如，你可以对一个元素作如下任何一种类型的定义：它有一个子元素，有一个或多个子元素，有零个或多个子元素，至少有一个子元素。你还可以定义复合关系，比如“元素 X 是有效的，如果它含有一个或多个子元素 Y，或一个子元素 Z”。你还可以定义复合关系，举个例子就明白了，例如：<个人资料>中必须包括<姓名><性别>，可以包括一个或多个<联系方式>，可以有<个人奖励>也可以没有。

由于以上的特点，父元素类型的元素的声明就可以有各种灵活而多样的形式。各种形式之间主要是在子元素出现的顺序、出现的次数以及各个子元素之间的复合关系这几方面存在着差异。要搞清楚它们的区别，熟悉下面的表格是必不可少的。

表 5.1　正则表达式中的元字符的含义

正则表达式中的元字符	元字符所对应的含义
Space(空格)	不要求严格遵从顺序要求
,	AND，要求严格遵从顺序要求
+	出现一次或多次
*	出现零次或多次
?	可选，不出现或出现一次
（）	一组要共同匹配的表达式
|	OR，或
元素 A 、元素 B、元素 C	元素列表，无需遵从顺序要求

程序清单 5.8 有助于读者更好地把握这种元素类型下的声明形式。例如程序清单 5.8 中 XML 文档就是有效的，该 DTD 并没有对< 学生>的子元素的顺序作出要求。

程序清单 5.8(student5.xml)

```
<?xml version = "1.0" encoding="GB2312" standalone = "yes"?>
<!DOCUMENT  学生>
[<!ELEMENT 学生(姓名 EMAIL)>
    <!ELEMENT 姓名(#PCDATA)>
    <!ELEMENT EMAIL(#PCDATA)>]>
< 学生>
        <EMAIL>zhang@etang.com</EMAIL>
            <姓名>张三</姓名>
    </ 学生>
```

程序清单 5.9 中的 XML 文档是无效的，因为元字符“,”要求在 DTD 中定义的两个子元素都必须出现，而文档中只出现了子元素<姓名>，而把<EMAIL>抛之脑后，显然是偷工减料的做法，是语法分析器所不能容忍的。

程序清单 5.9(studenterror.xml)（元字符的错误用法示例，不能编译执行，将报错）

```
<?xml version = "1.0" encoding="GB2312" standalone = "yes"?>
<!DOCUMENT  学生
[<!ELEMENT 学生(姓名，EMAIL)>
    <!ELEMENT 姓名(#PCDATA)>
    <!ELEMENT EMAIL(#PCDATA)>]>
< 学生>
        <姓名>张三</姓名>
    </ 学生>
```

还可以用括号把各个子元素并成一组，然后用元字符对其进行操作，例如

```
<!ELEMENT 联系人(姓名，EMAIL)+>
```

```
<!ELEMENT 姓名(#PCDATA)>
<!ELEMENT EMAIL(#PCDATA)>
```

程序清单 5.10 所示的 XML 文件是符合该 DTD 文件的要求的。

程序清单 5.10(temp.xml)

```
<学生>
    <姓名>张三</姓名>
    <EMAIL>zhang@aaa.com</EMAIL>
  <姓名>李四</姓名>
    <EMAIL>li@bbb.org</EMAIL>
    <姓名>王五</姓名>
    <EMAIL>wang@ccc.org</EMAIL>
</学生>
```

仅仅因为元字符“+”的位置的改变，DTD 文件的含义就发生了很大的变化。

元字符“|”的作用是符号“|”描述了一个 OR 操作。因此，下面的 DTD 片段所规定的 XML 元素是所有的<学生>元素应该有一个<姓名>子元素，同时，在此之后还应该有一个<电话>或一个<EMAIL>元素，但不能同时有<电话>和<EMAIL>两个元素。

例：

```
<!DODUMENT 学生[
<!ELEMENT 学生(姓名，(电话|EMAIL))>
    <!ELEMENT 姓名(#PCDATA)>
    <!ELEMENT 电话(#PCDATA)>
    <!ELEMENT EMAIL(#PCDATA)>
]>
```

XML 正则表达式的匹配原则不允许循环逻辑。所以，OR 的意思是或者选这个或者选那个，但不能两个都选，也不能两个都不选。

例：

```
< 学生>
    <姓名>张三</姓名>
</学生>
```

这个例子是一个“无效的”XML 片段，因为 DTD 中规定或者有一个<电话>元素，或者有一个<EMAIL>元素，但上面这个例子中两种元素都没有。

字符“?”说明一个子元素是可选的，它可以出现，也可以不出现。因此，在下面的 DTD 片断中，我们规定，每一个<学生>都必须有一个<姓名>子元素，同时或者有一个<电话>子元素，或者有一个<EMAIL>子元素，此外，它还可以包含一个<地址>子元素，也可以不包含这种元素。

例：

```
<!ELEMENT 学生(姓名，(电话|EMAIL)，地址?)>
 <!ELEMENT 姓名(#PCDATA)>
 <!ELEMENT 电话(#PCDATA)>
 <!ELEMENT EMAIL(#PCDATA)>
 <!ELEMENT 地址(学校，系，宿舍)>
```

```
<!ELEMENT 学校(#PCDATA)>
<!ELEMENT 系 (#PCDATA)>
<!ELEMENT 宿舍(#PCDATA)>
```

根据这个 DTD 的描述，下面的 XML 片段是有效的：

```
<学生>
    <姓名>张风林</姓名>
    <EMAIL>zhang@etang .com</EMAIL>
    <地址>
        <学校>北京科技大学</学校>
        <系>机械工程系</系>
        <宿舍>34楼345号</宿舍>
    </地址>
</学生>
```

4．混合元素类型

这类元素中即可以包含子元素，也可以包含纯文本字符，同样也可以在 DTD 中通过正则表达式规定子元素、纯文本字符出现的顺序和次数。语法分析器通过同样的方式验证文档的有效性。程序清单 5.11 中，<学生>就是一个混合元素。

程序清单 5.11（student5.xml）

```
<?xml version = "1.0" encoding="GB2312" standalone = "yes"?>
<!DOCTYPE CONTACTS [
  <!ELEMENT 学生列表 ANY>
  <!ELEMENT 学生(姓名|电话|EMAIL|#PCDATA)*>
  <!ELEMENT 姓名(#PCDATA)>
  <!ELEMENT 电话(#PCDATA)>
  <!ELEMENT EMAIL(#PCDATA)>
]>
<学生列表>
    <学生>
            <姓名>雷雨风</姓名>
            <电话>(010)62345678</电话>
            <EMAIL>zhang@etang.com</EMAIL>
            最喜爱的运动：游泳；
最喜欢的歌曲：一天到晚游泳的鱼；
最喜欢的菜：酸菜鱼。
</学生>
</学生列表>
```

在定义元素时，在下还要提醒读者注意以下问题：

（1）ETD(元素类型声明)的顺序是无关紧要的，如下例：

```
<!ELEMENT 姓名(#PCDATA)>
        <!ELEMENT 联系人列表 ANY>
        <!ELEMENT 联系人(姓名)>
```

和

例：

```
<!ELEMENT 联系人列表 ANY>
<!ELEMENT 联系人(姓名)>
<!ELEMENT 姓名(#PCDATA)>
```

所定义的文件结构是完全相同的。

（2）不能对不同的元素使用相同的元素名，即便这些元素的内容、包含的子元素不同也不行，因为它只会引起文件各个元素的混淆，使文件的可读性大打折扣。

例：

```
<!ELEMENT 联系人列表 ANY>
<!ELEMENT 联系人(姓名)>
<!ELEMENT 联系人(EMAIL)>
<!ELEMENT 姓名(#PCDATA)>
<!ELEMENT EMAIL(#PCDATA)>
```

在这个例子中，对<联系人>的重复定义，会引起错误。

属性列表声明

例：

```
<!ELEMENT BOOK(AUTHOR，TITLE，DESCRIPTION，PRICE)>
< !ATTLIST BOOK language CDATA #REQUIRED >
```

上面例子的第二行实际就是一个属性列表的声明。DTD 的属性列表声明采用了如下的格式：

<!ATTLIST 元素名 （属性名 属性类型 缺省值）*>

结合前面的知识我们可以知道，对于某一个元素它的属性可以有零个或者多个，每个属性的声明都应当包括属性缺省值、属性类型及属性名称等三项内容。缺省值说明在 XML 文件中，如果没有特别说明属性的取值，语法分析器默认它具有的取值；属性类型则用来指定该属性是属于 10 个有效属性类型中的哪种类型。

1．元素属性的缺省值

我们先从元素属性的缺省值说起，要了解这一点读者需要浏览一下表 5.2，这会对读者有所帮助。

表 5.2　属性缺省值的意义

属性缺省值	对于缺省值的描述
#required	表示在标记中必须出现此属性
#implied	标记中可以不出现此属性
#fix	属性的值是固定的某个值
字符串	标记中如没有指定属性的值，那么此字符串就是此属性的值

（1）必须赋值的属性。关键字 REQUIRED 说明 XML 文件中必须为这个属性给出一个属性值。例如，假设你想定义一个“页面作者”元素，并把这个元素加入所有网站中的每一个页面。之所以定义这个元素，是为了页面编辑者能够提供他的联系信息，以便当发现页面错误或无效链接时，可以及时地通知他。在这种情况下，每个页面作者都有不同的

个人信息，所以你无法事先知道应该用什么作为缺省值，但你又的确需要提供每个人的信息。这时候，你就可以把与联系信息相关的属性定义为必需的（REQUIRED），而且不用提供缺省值。

（2）属性值可有可无的属性。当使用 IMPLIED 关键字时，文法解释器不再强行要求你在 XML 文件中给该属性赋值，而且也无需在 DTD 中为该属性提供缺省值。可以说，这是对属性值有无的最低要求，现实中经常用到。

（3）固定取值的属性。还有一种特殊情况，你需要为一个特定的属性提供一个缺省值，并且不希望 XML 文件的编写者把你的缺省值替代掉。这时候，就应该使用 FIXED 关键字，同时为该属性提供一个缺省值。

（4）定义缺省值的属性。如果不使用上面任何一种关键字的话，该种属性就是属于这种类型。对于这种属性，你需要在 DTD 中为它提供一个缺省值。而在 XML 文件中可以为该属性给出新的属性值来覆盖事先定义的缺省值，也可以不另外给出属性值，后一种情况下它就默认为采用 DTD 中给出的缺省值。

下面举一个简单的 XML 代码片断：

```
<!ATTLIST    大四学生
          身高  #CDATA #IMPLIED
          体重  #CDATA #IMPLIED
          联系信息  #CDATA #REQUIRED
          政治面貌#CDATA #FIXED "群众"
          家庭出身  #CDATA "独生子女">
```

2．属性类型

对于属性类型，读者在前面的章节可能已经接触了一些。在这里，我想结合 DTD 的实例再和读者一起熟悉一遍。读者都知道，属性类型主要有以下 10 种：

CDATA、ENYMERATED、ID、IDREF、IDREFS、ENTITY 、ENTITIES 、NMTOKEN 、NMTOKENS、NOTATION。我们将一一分析它们的实例。

CDATA 指的是纯文本，即由字符、符号“&”、小于号“<”和引号“""”组成的字符串。请看程序清单 5.12 这个节目单的例子。

程序清单 5.12(program.xml)

```
<?xml version = "1.0"        encoding="GB2312"      standalone = "yes"?>
<!DOCTYPE  节目单[
   <!ELEMENT  节目单  ANY>
   <!ELEMENT  节目  (#PCDATA)>
   <!ATTLIST  节目  演员  CDATA>
   ]>
<节目单>
     <节目  演员="赵本山">卖拐</节目>
     <节目  演员="冯巩  郭东林">的寸进尺</节目>
     <节目  演员="郭达  蔡明">西厢记<节目>
</节目单>
```

属性也可以被描述为一组可接受的取值的列表，XML 文件中对属性的赋值将从这个列

表中选取一个值。这类属性属于枚举类型 ENUMERATED，不过，关键字 ENUMERATED 是不出现在 DTD 定义中的。这种属性类型在处理现实中本身受到很大限制的属性时，优越性尤为突出，例如性别、政治面貌、最高学历诸如此类的属性的处理，就可以采用这种方式。

程序清单 5.13（shop.xml）

```
<?xml version = "1.0"        encoding="GB2312"       standalone = "yes"?>
<!DOCTYPE 购物篮 [
   <!ELEMENT 购物篮 ANY>
   <!ELEMENT 肉 EMPTY>
   <!ATTLIST 肉 类型( 鸡肉 | 牛肉 | 猪肉 | 鱼肉 ) "鸡肉">
   ]>
<购物篮>
     <肉 类型 = "鱼肉"/>
     <肉 类型 = "牛肉"/>
     <肉/>
</购物篮>
```

程序清单 5.13 就是一个枚举类型的属性，其中第 3 个<肉>的类型属性的缺省值为“鸡肉”，不知读者理解得如何。

ID 是用属性值的方式为文件中的某个元素定义唯一标识的方法，它的作用类似于 HTML 文件中的内部链接。在大多数情况下，ID 由处理文件的程序或脚本语言使用。一般而言，不要给 ID 类型的属性事先指定缺省值，这很容易引起不同的元素具有相同的标识的情况，更不能使用 FIXED 型的缺省值。此类属性经常使用 REQUIRED 缺省类型，当然，这也不是必需的。有的应用并不要求每个元素都有自己的标识，所以，也可以使用 IMPLIED 缺省类型。

程序清单 5.14（student6.xml）

```
<?xml version = "1.0"      encoding="GB2312"       standalone = "yes"?>
<!DOCTYPE 学生列表[
   <!ELEMENT 学生列表 ANY>
   <!ELEMENT 学生姓名，EMAIL]>
   <!ELEMENT 姓名(#PCDATA)>
   <!ELEMENT EMAIL(#PCDATA)>
   <!ATTLIST 学生 编号 ID #REQUIRED>
   ]>

<学生列表>
   <学生 编号="1">
     <姓名>雷雨风</姓名>
     <EMAIL>lei@etang.com</EMAIL>
   </学生>

   <学生 编号="2">
```

```
        <姓名>路朝天</姓名>
        <EMAIL>lu@sina.com</EMAIL>
    </学生>
</学生>
```

IDREF 类型允许一个元素的属性使用文件中的另一个元素，方法就是把那个元素的 ID 标识值作为该属性的取值。例如程序清单 5.15。

程序清单 5.15（student7.xml）

```
<?xml version = "1.0"    encoding="GB2312"    standalone = "yes"?>
<!DOCTYPE  学生列表[
    <!ELEMENT  学生列表  ANY>
    <!ELEMENT  学生(姓名，EMAIL)>
    <!ELEMENT  姓名(#PCDATA)>
    <!ELEMENT EMAIL(#PCDATA)>
    <!ATTLIST  学生  编号  ID #REQUIRED>
    <!ATTLIST  学生  导师  IDREF #IMPLIED>
    ]>

<学生列表>
    <学生  编号="2">
            <姓名>路朝天</姓名>
            <EMAIL>lu@sina.com</EMAIL>
    </学生>

    <学生  编号="1"  导师="58">
            <姓名>雷雨风</姓名>
            <EMAIL>lei@etang.com</EMAIL>
    </学生>
</学生列表>
```

NMTOKEN 是 Name token （即命名符号）的缩写，该类属性值是一种受限制的字符串表示，属性值只能由“字母”、“数字”、“下划线”、“.”、“:”、“-”这些符号组成。一般地，一个 NMTOKEN 属性必须由单个单词构成，但该单词不必符合其他约束，比如不必匹配其他的属性或实体。 NMTOKENS 类型的属性值可以包含多个由空格（或制表符、换行符）分隔的 NMTOKEN 值。请看程序清单 5.16。

程序清单 5.16（data.xml）

```
<?xml version = "1.0"    encoding="GB2312"    standalone = "yes"?>
<!DOCUMENT  数据
[<!ELEMENT  数据(#PCDATA)>
<!ATTLIST  数据
    安全性( ON | OFF ) "OFF">
    授权用户  NMTOKENS #IMPLIED
>]>
<数据  安全性="ON"  授权用户  = "Toms Cattuso">
```

芝麻开门，123321
</数据>

实体声明

在 DTD 中，还可以声明一些称为 Entity(实体)的东西让 DTD 和 XML 文件使用，这就是实体声明。实体实际上是属性类型的一种，可以把 Entity 看作是一个常量，它有一定的值。在 DTD 中，Entity 的声明语法为：

<!ENTITY entity-name entity-definition>

例如：我们在 DTD 中声明<!ENTITY PC "(#PCDATA)"> ，那么在后面的元素设定中，就可以使用这个 Entity 来代替“(#PCDATA)”这个字符串，如：<!ELEMENT 作者 (#PCDATA)>可以写成<!ELEMENT 作者 &PC;>。引用 Entity 的时候，必须要在 Entity 名称前面加上“&”符号，后面加上“；”符号。实体可以分为：内部实体、外部实体和参数实体 3 种，下面是 3 个典型的实体：

< !ENTITY entity1 " 文本 ">
< !ENTITY entity2 SYSTEM "/standard/legalnotice.xml">
< !ENTITY entity3 SYSTEM "/standard/logo.gif" NDATA GIF87A>

内部实体是对文本字符串的命名如上例 3 个实体中的第一个，此后使用&entity1 就相当于插入了字符串“文本”。内部实体经常用于定义重复出现的文本，或经常要求改动的文本（如文档的修订情况记录）。

外部实体是对其他外部文件的命名，它使得 XML 文件可以引用外部文件的内容。外部文件可以包含文本，也可以包含二进制数据。与内部实体一样，外部文件将被插入实体引用出现的位置，而且将随同文档的其余部分一起被解析器处理（对二进制数据的引用只能出现于元素属性，因为二进制数据不可被解析）。对二进制数据的引用通常用来提供图片等“非 XML ”的内容。前面示例中的 2 、 3 两个实体声明即为外部实体声明。

与前面两种实体不同，参数实体只能出现于文档类型定义中，包括定义和引用。声明参数实体时，名字前面要加一个“ % ”和空格；在引用参数实体时，引用其他实体时使用的“ & ”符号也要改为“ % ”。参数实体引用将直接展开为它所代表的文本，而其他实体引用并不立即展开。 例如我们声明了如下参数实体：

< !ENTITY % personcontent "#PCDATA | quote">

则用下面这种形式引用参数实体：

< !ELEMENT allen (%personcontent;)*>

符号声明

和实体声明一样，符号声明也是属性声明的一种，实际上就是 NOTATION 类型。符号声明允许属性值为一个 DTD 中声明的符号，这个类型对于使用非 XML 格式的数据非常有用。现实世界中存在着很多无法或不易用 XML 格式组织的数据，例如图像、声音、影像等等。对于这些数据，XML 应用程序常常并不提供直接的应用支持。通过为它们设定 NOTATION 类型的属性，可以向应用程序指定一个外部的处理程序。例如，当你想要为一个给定的文件类型指定一个演示设备时，可以用 NOTATION 类型的属性作为触发。要使用 NOTATION 类型作为属性的类型，首先要在 DTD 中为可选用的记号作出定义。定义的方

式有两种，一种是使用 MIME 类型，形式是

<！NOTATION 记号名 SYSTEM "MIME 类型">

另一种是使用一个 URL 路径，指定一个处理程序的路径。

<！NOTATION 记号名 SYSTEM "URL 路径名">

在程序清单 5.17 中，为<视频文件>元素指定了两种可选设备：一种是 RealPlayer.exe，用来播映.mov 文件，另一种则用来显示.gif 图像。

程序清单 5.17(fuhao.dtd)

```
<?xml version = "1.0"        encoding="GB2312"   tandalone = "yes"?>
<!DOCTYPE 文件[
   <!ELEMENT 文件 ANY>
   <!ELEMENT 视频文件 EMPTY>
   <!ATTLIST 视频文件 演示设备 NOTATION ( mp | gif ) #REQUIRED>
   <!NOTATION mp SYSTEM "RealPlayer.exe">
   <!NOTATION gif SYSTEM "Image/gif">
   ]>

<文件>
   <视频文件 演示设备 = "mp"/>
</文件>
```

条件段

条件段是文档类型声明外部子集的一部分，取决于相应的关键字，它们或被包含在 DTD 逻辑结构之内，或被排除在 DTD 逻辑结构之外。同内部或外部 DTD 子集一样，条件段可以包含一个或多个完整的声明、注释、处理指令或嵌套的条件段，其间可以夹杂空白。

如果条件段的关键字是 INCLUDE，那么条件段的内容是 DTD 的一部分，如果条件段的关键字是 IGNORE，那么条件段的内容逻辑上不是 DTD 的一部分。如果一个关键字为 INCLUDE 的条件段出现在更大的关键字为 IGNORE 的条件段中，内外两个条件段都被忽略。如果条件段的关键字是一个参数实体引用，处理器在决定是否包含或忽略此条件段前，必须先将该参数实体置换成其内容。

例：

```
<ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >

<![%draft;[
<!ELEMENT book (comments*，title，body，supplements?)>
]]>
<![%final;[
<!ELEMENT book (title，body，supplements?)>
]]>
```

在上面的例子中，考虑条件段的作用，最终该 DTD 片段应当理解为定义了一个元素<book>，其中可以包括零个或多个子元素<comments>，还必须包含<title><dody >子元素各

一个，有一个可选子元素<supplements>，而且各子元素必须按照前面的顺序出现。

5.2.4　如何使用 DTD 文件

将 DTD 与 XML 联系起来就要用到文档类型声明。文档类型声明必须紧跟 XML 声明，类型定义信息可以来自外部 DTD （称为外部子集），也可直接嵌入 XML 文档内（称为内部子集），还可以同时提供外部子集和内部子集。使用外部 DTD 的 XML 文档的结构为：

```
<?xml version = "1.0" encoding="GB2312" standalone = "no"?>
<!DOCTYPE  根元素名SYSTEM "外部DTD文件的URL">
文档体.....
```

读者可以看出要使用外部 DTD 只需将 XML 声明中的 standalone 属性声明改为“no”，然后在文档的第二行加入文档类型声明就 OK 了。如程序清单 5.18、5.19 所示的 XML 文档在相应的 DTD 文件的作用下就是有效的。

程序清单 5.18(Reference.dtd)

```
<?xml version="1.0" encoding="GB2312" ?>
<!ELEMENT Reference (书籍*)>
    <!ELEMENT  书籍  (名称，作者，价格)>
    <!ELEMENT  名称  (#PCDATA)>
    <!ELEMENT  作者  (#PCDATA)>
    <!ELEMENT  价格  (#PCDATA)>
        <!ATTLIST  价格  货币单位  CDATA #REQUIRED>
```

程序清单 5.19(Reference.xml)

```
<?xml version="1.0" encoding="GB2312" standalone="no"?>
<!DOCTYPE Reference SYSTEM "Recipe.dtd">
<Refenence>
    <书籍>
    <名称>二〇〇二年研究生入学考试指南（英语分册）</名称>
    <作者>朱泰琪</作者>
    <价格　货币单位＝"元" >35.00<价格>
    </书籍>
    <书籍>
    <名称>一九九九Java词汇 </名称>
    <作者>黄敏红</作者>
    <价格　货币单位＝"元" >50.00<价格>
    </书籍>
</Reference>
```

5.2.5　XML 的数据模式问题

XML 是一种完全面向数据语义的标识语言，取消了 HTML 的显示样式与布局描述能力，突出了数据的语义与元素结构描述能力。简单地说，数据模式就是指元素的嵌套形式，它并没有像元素那样在 DTD 中单独指出，它是以元素之间的嵌套关系来表现的，是一种逻辑而已。数据模式强调元素之间的逻辑嵌套关系，而不是元素、标记或者元素属性的定义

和描述。

XML 的数据模式（数据逻辑结构）的特点

XML 提供了一个将大量信息组织成为具有确定意义的整体结构的功能，这种具有确定意义的整体称为文档。XML 文档是数据的容器，它并不关心数据的显示样式与布局效果。构造 XML 文档的基本成分是元素（Element）。

XML 文档的元素形成一种层次结构，处于顶端的元素称为根元素，根元素包含了所有其他元素。XML 支持元素的嵌套，但不像 HTML 那样允许元素交迭，即 XML 中元素结束标记出现次序应该和开始标记出现次序严格相反。XML 通过元素的这种层次关系支持丰富数据结构（Rich Data Structure）。换句话说，XML 文档中的元素之间不是简单的前后次序关系，而是具有明确的从属、依赖等关系。如图 5.1 所示，根元素 Student（学生）包含了 Contaction（联系方式）元素，Contaction 又包含了 Email(电子邮件)元素，最后形成一个树状结构。元素的名字描述了元素的内容，而分层结构描述的是元素之间的关系。例如，元素 Contaction 包含元素 Email 表明 Email 是 Contaction 的子元素。

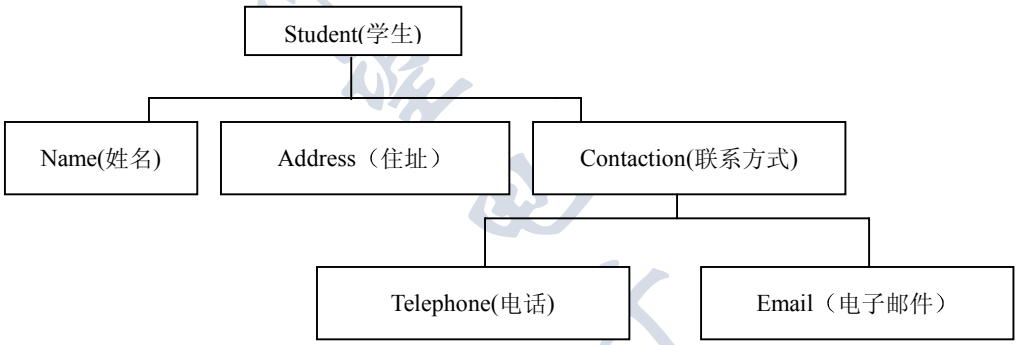


图 5.1　XML 文档中的元素关系

XML 数据的输出与存储

XML 描述的是数据的内容或语义，而不像 HTML 那样描述显示样式和布局。那么，如何将 XML 描述的内容“展现”给用户呢？

XML 文档除了可以用文本编辑器浏览外，由于它具有天然的层次结构，许多工具还可以将 XML 文档显示为一个可扩展的树形结构。更为复杂的输出样式需要用到过滤器。例如，对于一部 XML 格式的小说，如果要将它以传统的纸张方式、Web 页面格式和适合掌上设备阅读的格式发布，就需要分别为这 3 种不同媒体提供输出样式说明，但描述内容的 XML 文档无需任何改动，这就实现了内容与显示样式的分离。

为了向用户提供基于 XML 的内容，查询与存储技术非常重要。假设某个用户需要在一个包含大量医疗设备信息的 XML 文档中访问有关助听器的数据，如何才能以足够的精确度查找到目标数据？这就需要有一种合适的存储系统和查询语言。虽然目前还没有专门用于 XML 的标准查询语言，但有关查询语言规范的工作正在进行当中。一些厂商已经向 W3C 提交了建议，W3C 在 1998 年 12 月成立了一个工作组以讨论查询语言需求。期间，

已经有对象数据库（ODBMS）供应商为填补这个空白开发相关工具，包括访问 XML 资源子集的查询接口及用来管理大容量 XML 资源的一些工具。ODBMS 可谓 XML 数据的天然存储工具。XML 本质上是元素与对象的一个分层结构，而 ODBMS 又特别适合于存储层次型数据；此外，ODBMS 能够在元素这一层次上管理和操纵数据，并在这一层次上提供了极为完善的加锁模式。

除了 ODBMS 之外，将信息存储在传统的关系数据库，然后将查询结果转换为 XML 也是一种可行的方法。例如，在 Web 开发中使用 ADO 查询 ODBC 数据源然后将结果用 XML 来描述。使用这种方法的好处在于可以用标准 SQL 来获得结果集的 XML 表达。也许在不远的将来，许多关系数据库会提供将查询结果直接转换为 XML 的机制。

面向Web的数据挖掘技术——XML 的数据模式优势在网络应用中的体现

由于 XML 能够使不同来源的结构化的数据很容易地结合在一起，因而使搜索多样的不兼容的数据库能够成为可能，从而为解决 Web 数据挖掘难题带来了希望。XML 的扩展性和灵活性允许 XML 描述不同种类应用软件中的数据，从而能描述搜集的 Web 页中的数据记录。同时，由于基于 XML 的数据是自我描述的，数据不需要有内部描述就能被交换和处理。

Web 上的数据与传统的数据库中的数据不同，传统的数据库都有一定的数据模型，可以根据模型来具体描述特定的数据。而 Web 上的数据非常复杂，没有特定的模型描述，每一站点的数据都各自独立设计，并且数据本身具有自述性和动态可变性。因而，Web 上的数据具有一定的结构性，但因自述层次的存在，从而是一种非完全结构化的数据，这也被称之为半结构化数据。半结构化是 Web 上数据的最大特点。

Web 数据挖掘技术首要解决半结构化数据源模型和半结构化数据模型的查询与集成问题。解决 Web 上的异构数据的集成与查询问题，就必须要有一个模型来清晰地描述 Web 上的数据。针对 Web 上的数据半结构化的特点，寻找一个半结构化的数据模型是解决问题的关键所在。除了要定义一个半结构化数据模型外，还需要一种半结构化模型抽取技术，即自动地从现有数据中抽取半结构化模型的技术。面向 Web 的数据挖掘必须以半结构化模型和半结构化数据模型抽取技术为前提。

以 XML 为基础的新一代 WWW 环境是直接面对 Web 数据的，不仅可以很好地兼容原有的 Web 应用，而且可以更好地实现 Web 中的信息共享与交换。XML 可看作一种半结构化的数据模型，可以很容易地将 XML 的文档描述与关系数据库中的属性一一对应起来，实施精确地查询与模型抽取。

1．XML 对异源结构化数据的处理

XML 给基于 Web 的应用软件赋予了强大的功能和灵活性，因此它给开发者和用户带来了许多好处。比如进行更有意义的搜索，并且 Web 数据可被 XML 唯一地标识。没有 XML，搜索软件必须了解每个数据库是如何构建的，但这实际上是不可能的，因为每个数据库描述数据的格式几乎都是不同的。由于不同来源数据的集成问题的存在，现在搜索多样的不兼容的数据库实际上是不可能的。XML 能够使不同来源的结构化的数据很容易地结合在一起。软件代理商可以在中间层的服务器上对从后端数据库和其它应用处来的数据进行集成。然后，数据就能被发送到客户或其他服务器做进一步的集合、处理和分发。XML 的扩展性

和灵活性允许它描述不同种类应用软件中的数据，从描述搜集的 Web 页到数据记录，从而通过多种应用得到数据。同时，由于基于 XML 的数据是自我描述的，数据不需要有内部描述就能被交换和处理。利用 XML，用户可以方便地进行本地计算和处理，XML 格式的数据发送给客户后，客户可以用应用软件解析数据并对数据进行编辑和处理。使用者可以用不同的方法处理数据，而不仅仅是显示它。XML 文档对象模式(DOM)允许用脚本或其他编程语言处理数据，数据计算不需要回到服务器就能进行。XML 可以被利用来分离使用者观看数据的界面，使用简单灵活开放的格式，可以给 Web 创建功能强大的应用软件，而原来这些软件只能建立在高端数据库上。另外，数据发到桌面后，能够用多种方式显示。

2．XML 对数据简单开放式的描述

XML 还可以通过以简单开放扩展的方式描述结构化的数据，XML 补充了 HTML，被广泛地用来描述使用者界面。通过 XML，数据可以粒状地更新。每当一部分数据变化后，不需要重发整个结构化的数据。变化的元素必须从服务器发送给客户，变化的数据不需要刷新整个使用者的界面就能够显示出来。XML 也允许加进其他数据，比如预测的温度。加入的信息能够进入存在的页面，不需要浏览器重新发一个新的页面。XML 应用于客户需要与不同的数据源进行交互时，数据可能来自不同的数据库，它们都有各自不同的复杂格式。但客户与这些数据库间只通过一种标准语言进行交互，那就是 XML。由于 XML 的自定义性及可扩展性，它足以表达各种类型的数据。客户收到数据后可以进行处理，也可以在不同数据库间进行传递。总之，在这类应用中，XML 解决了数据的统一接口问题。但是，与其他的数据传递标准不同的是，XML 并没有定义数据文件中数据出现的具体规范，而是在数据中附加 TAG 来表达数据的逻辑结构和含义。这使 XML 成为一种程序能自动理解的规范。

3．XML 对运算负荷的分布式处理

XML 应用于将大量运算负荷分布在客户端，即客户可根据自己的需求选择和制作不同的应用程序以处理数据，而服务器只需发出同一个 XML 文件。如按传统的“Client/Server”工作方式，客户向服务器发出不同的请求，服务器分别予以响应，这不仅加重服务器本身的负荷，而且网络管理者还需事先调查各种不同的用户需求以做出相应不同的程序，但假如用户的需求繁杂而多变，则仍然将所有业务逻辑集中在服务器端是不合适的，因为服务器端的编程人员可能来不及满足众多的应用需求，也来不及跟上需求的变化，双方都很被动。应用 XML 则将处理数据的主动权交给了客户，服务器所作的只是尽可能完善、准确地将数据封装进 XML 文件中，正是各取所需、各司其职。XML 的自解释性使客户端在收到数据的同时也理解数据的逻辑结构与含义，从而使广泛、通用的分布式计算成为可能。

5.3　CSS 与 XSL 及其实例

在 XML 文件中，使用的基本上是自定义的标记，显然一个浏览器是无法理解这些标记的，现在，浏览器仅仅是作为一个 XML 文件的解析器——只要你的 XML 文件是 Well-Formed 的，那么它就将文件原封不动地给你显示出来。在 XML 中内容与表现形式是分开的。这就使得不同的用户可以根据他们自己的需要来定义数据的表现形式。在一个 XML 的源文件中并没有关于它表现形式的信息。

XML 文件的所有表现信息多发放在了 stylesheet（样式表）文件当中。stylesheet 文件全权负责 XML 源文件的表现形式。所以说如果一个 XML 源文件对应不同的 stylesheet 文件它就会有不同的表现形式. 有了 stylesheet 文件我们可以对文件表现型始终的大小、颜色、空白作特定的规定。样式单(Style Sheet)是一种描述结构文档表现方式的文档，它既可以描述这些文档如何在屏幕上显示，也可以描述它们的打印效果，甚至声音效果。与传统使用的<font>等标记相比，样式单有许多突出的优点：表达效果丰富、文档体积小、便于信息检索、可读性好。

迄今为止，W3C 已经给出了两种样式单语言的推荐标准，一种是层叠样式单 CSS（Cascading Style Sheets），另一种是可扩展样式单语言 XSL（eXtensible Stylesheet Language）。CSS 可以展现 HTML 和 XML 文件，而 XSL 可以展现 XML 和 Transformation（转型语言）。

5.3.1　CSS 简介

CSS 就是一种叫做样式表（stylesheet）的技术。也有的人称之为层叠样式表（Cascading Stylesheet）。在主页制作时采用 CSS 技术，可以有效地对页面的布局、字体、颜色、背景和其它效果实现更加精确的控制。只要对相应的代码做一些简单的修改，就可以改变同一页面的不同部分，或者页数不同的网页的外观和格式。CSS 是通过对页面结构的风格控制的思想，控制整个页面的风格的。如果站点上所有的网页风格都使用一个 CSS 文件进行控制，只要修改这个 CSS 文件中相应的行，那么整个站点的所有页面都会随之发生变动。

CSS 的主要特点和对 XML 的贡献如下：

- 实现了所有浏览器和平台之间的兼容性。
- 使页面的字体变得更漂亮，而且可以选择各种不同的字体、颜色、背景等，使页面不再像单纯的 XML（包括 DTD）文档那样枯燥单调。
- 使得网页中的布局变得轻松而简单，并且更合理、更清晰 。
- 只要这些网页套用的都是一个风格样式表，可以将许多网页的风格格式同时更新，不用再一页一页地更新了。
- 更少的编码、更少的页数和更快的下载速度。
- 样式表只是简单的文本，就像 XML 那样。它不需要图像，不需要执行程序，不需要插件，不需要流式。它就像 XML 指令那样快。

5.3.2　CSS 的基本格式问题

定义 CSS 的基本格式如下：

part{property : value ; property : value ; ...}

part：选择符，被施加样式的元素，可以是标记（tag）、类(class)、标识(id)等。

property：样式属性，可以是颜色、字体、背景等等。

value：样式属性取值，决定样式结果。

对于每个被施加样式的元素，都采用了命令括号 {……} 的格式封闭成为一个独立的单元。样式单一般不包含在 XML 文档内部，以独立的文档方式存在。可以通过如下格式在 XML 文档内部引用 CSS 风格样式表：

< ? xml-stylesheet　type = " text /　css "　href="example1.css"?>

其中，xml-stylesheet　为关键字，type="text /css "表示风格样式表为 CSS 类型的，href="example1.css"表示所引用的风格样式表文件为 example1.css，而且该文件与引用它的 XML 文档位于同一目录下。如果二者不在同一目录下，则必须给出该文件的绝对路径。

CSS 风格样式表对文本、图片、超链接等各种网页元素都可以进行控制，下面我们主要介绍一下对文字和超链接的控制，其他元素的控制与此类似，读者可以自己类比学习，掌握这部分内容。

文字控制

请看下面的简单例子。

例：

P art1{FONT－FAMILY: "宋体"; FONT－SIZE: 9pt; LINE－HEIGHT: 12pt; color: ＃000000}

说明：“FONT－FAMILY: "宋体";”用来指定网页文字的字体；“FONT－SIZE:9pt;”用来指定网页文字的字号大小，pt 是表示大小的单位；“LINE－HEIGHT:12pt;”用来指定行与行的垂直距离，即行高；“color: ＃000000”指定网页文字的颜色，＃000000 代表黑色，为十六进制数。

链接色彩变化及下划线的控制

例：

A:hover {BACKGROUND－COLOR:　＃ffccff; COLOR:　＃0080ff}

说明：hover 表示鼠标指示时，链接文字背景色为＃ffccff、前景色为＃0080ff。

例：

A:link {color: ＃000000;TEXT－DECORATION: none}

说明：link 表示未被访问时，链接颜色为黑色，链接无下划线。

例：

A:visited {color:gray;TEXT－DECORATION: none}

说明：visited 表示被访问后，链接颜色为灰色，链接无下划线。

例：

A:active {color:green;text－decoration: none}

说明：active 表示鼠标点击时，链接颜色为绿色，链接无下划线。

例：

A:hover {TEXT－DECORATION: underline}

说明：hover 表示鼠标指示时，链接显示下划线。

注释：

none——没有下划线

underline——下划线

overline——上划线

line-through——中划线

5.3.3　一个应用 CSS 的 XML 文档的实例说明

程序清单 5.20所示的是一个简单的 XML 文档和它的样式单例子 。

程序清单 5.20(information.xml)

```
<?xml version="1.0" encoding="gb2312" ? Standalone="no" >
<?xml-stylesheet type="text/css" href="example1.css"?>
<!DOCTYPE  information  SYSTEM "customer.dtd">
<information>
    <customer>
        <name>李超群</name>
        <name>雷雨风</name>
        <id>1</id>
        <address>当代商城</ address >
        <email>lichaoqun@sina.com</email>
        <email>leiyufeng@sohu.com</email>
        <goods>钻石主板2代</goods>
</customer>
<customer>
        <name>朱峰</name>
        <id>3</id>
        <address>上海市淮海路35号</address>
        <email>zhufeng@hotmail.com</email>
        <email>zhufeng1@hotmail.com</email>
        <telephone>021-76483218</telephone>
        <goods>轻骑兵组合音响</goods>
</customer >
</information>
```

上面是一个完整的 XML 文档，描述的是一个客户信息表，其中有两个客户的资料。有关客户信息表的 DTD 定义如程序清单 5.21。

程序清单 5.21(customer.dtd)

```
<!ELEMENT information(customer*) >
<!ELEMENT customer(name+，id，address telephone? email*，goods) >
<!ELEMENT name(#PCDATA) >
<!ELEMENT id(#PCDATA) >
<!ELEMENT address(#PCDATA) >
<!ELEMENT telephone(#PCDATA) >
<!ELEMENT email(#PCDATA)>
<!ELEMENT goods(#PCDATA)>
```

下面看一下如何利用 CSS 将上述资料显示在浏览器中（见程序清单 5.22）。

程序清单 5.22(example1.css)

```
information，customer
{
    font-family: "宋体";
```

```
        font-size:10.5pt;
        font-weight:bold;
        color:black;
        display:block;
        margin-bottom:5pt;
}

id，address，telephone
{
        font-family: "隶书";
        font-weight:normal;
        font-size:10.5pt;
        display:block;
        color:blue;
        margin-left:20pt;
}

name，email
{
        font-family: "行楷"
        font-weight:bold;
        font-size:10.5pt;
        display:block;
        color:red;
        margin-top:5pt;
        margin-left:8pt;
}
```

利用上面的 CSS 风格样式文件，我们可以得到像普通网页一般的浏览效果，这种效果和单纯的 XML（包括 DTD）显然不可同日而语，读者结合前面一节的例子就可以发现这一点。

5.3.4　XSL 简介

XSL(eXtensible Stylesheet Language，可扩展样式语言)是为 XML 文件定义的一种标识语言，它将提供远远超过 CSS 的强大功能，如将元素再排序等。实际上简单的 XML 已可被 CSS 所解释，然而复杂的高度结构化的 XML 数据或 XML 文档则只能依赖于 XSL 极强的格式化的能力呈现给用户。正如 XML 介于 HTML 和 SGML 之间一样，XSL 标准介于 CSS 和 SGML 的 DSSSL(Document Style　Semanticsand　Specification Language，文档样式语义和规范语言)之间。DSSSL 定义格式化对象的全特征模式。由于 DSSSL 使用框架语法，而且是很复杂的，所以 DSSSL 未能得到推广应用。XSL 支持 DSSSL 流对象和 CSS 对象，并对复杂的任务提供进入脚本语言的通道，而且允许扩展。

XSL（可扩展样式语言）也是一种显示 XML 文件的规范。和 CSS 不同的是：XSL 是

遵循 XML 的规范来制定的。也就是说，XSL 文件本身符合 XML 的语法规定。XSL 在排版样式的功能上要比 CSS 强大。比如：CSS 适用于那些元素顺序不变的文件，它不能改变 XML 文件中元素的顺序——元素在 XML 文件中是什么顺序排列的，那么通过 CSS 表现出来顺序不能改变。对于那些需要经常按不同元素排序的文件，我们就要用 XSL。XSL 是怎样工作的呢？XML 文件在展开后是一种树状结构，称为“原始树”，XSL 处理器（现在只有 IE 5 支持 XSL，在 IE 5 中的处理器叫 XSL Stylesheet Processor）从这个树状结构读取信息，根据 XSL 样式的指示对这个“原始树”进行排序、复制、过滤、删除、选择、运算等操作后产生另外一个“结果树”，然后在“结果树”中加入一些新的显示控制信息，如表格、其他文字、图形以及一些有关显示格式的信息。XSL 处理器根据 XSL 样式表的指示读取 XML 文件中的信息，然后再重新组合后转换产生一个 Well-Formed 的 HTML 文件。浏览器显示 HTML 文件肯定是没问题的，这样，XML 文件中的信息就会以一定的形式显示在我们面前了。

下面提供一个 XSL 的简单实例，读者可以通过浏览体会 XSL 的实际效果。DTD 文件读者可以参照前面的讲述来书写，并把该 DTD 文件和下面的 XML 文件存放在一个目录下即可（见程序清单 5.23）。

程序清单 5.23(employee.xml)

```
<?xml version="1.0" encoding="GB2312"? standalone="no">
<?xml:stylesheet type="text/xsl" href="employee.xsl"?>
<!DOCTYPE 打工人员名单 SYSTEM  "employee.dtd">
<打工人员名单>
  <打工人员>
    <姓名>梅幽明</姓名>
    <id>897</id>
    <性别>男</性别>
    <出身>劳苦大众</出身>
    <爱好>在夕阳下散步 </爱好>
    <自我描述>
      鞋儿破，帽儿破，身上的袈裟破，南无阿弥驼佛。
    </自我描述>
  </打工人员>
  <打工人员>
      <姓名>艾奔驰</姓名>
      <id>888</id>
      <出身> 高干子弟 </出身>
      <鸣谢>这次活动得到了，**市**大学**系的支持，在此诚表谢意！ </鸣谢>
    </打工人员>
</打工人员名单>
```

对应的 XSL 文件如程序清单 5.24。

程序清单 5.24(employee.xsl)

```
<?xml version="1.0" encoding="GB2312"?>
<html  xmlns:xsl="http://www.w3.org/TR/WD-xsl">
```

```
<body    style="font-size:12pt;background-color:blue">
<xsl:for-each    select="打工人员名单/打工人员">
<div style="background-color:green;color:white;padding:4px">
<span syle="font-weight:bold;color:white">
姓名: <xsl:value-of    select="姓名"/>
</span>
<p>id: <xsl:value-of    select="id"/></p>
<p>性别: <xsl:value-of    select="性别"/></p>
<p>出身: <xsl:value-of    select="出身"/></p>
<p>爱好: <xsl:value-of    select="爱好"/></p>
</div>
 <div    style="background-color:yellow;color:red; font-size:9pt;margin-bottom:10pt;">
<p><center><xsl:value-of    select="自我描述"/></center></p>
 </div>
 <div    style="background-color:orange;color:green;font-size:9pt;">
 <center><xsl:value-of    select="鸣谢"/></center>
</div>
</xsl:for-each>
</body>
</html>
```

在浏览器中打开 employee.xml 文件时，我们就可以看到施加了 XSL 风格样式表作用的网页的实际效果了。

5.3.5　XSL 语法

前面我们只是给出了一个 XSL 的实例，下来我们要学习 XSL 的书写方法及其规则。要利用 XSL 将 XML 文档中提取相关数据，就要用到 XSL 提供的模式化查询语言。所谓模式化查询语言，就是通过相关的模式匹配规则表达式从 XML 里提取数据的特定语句，然后利用 XSL 中的 HTML 标记将所选择的网页元素（数据）包围起来，也就是用表现信息把内容信息包围起来，我们也可以说把“原始树”的元素添加到“结果树”中。

在 XSL 中我们可以采用 3 种不同的模式来实现上述过程：

选择模式< xsl:for-each >、< xsl:value-of >

测试模式< xsl:if > <xsl:choose><xsl:otherwise>和< xsl:when >

匹配模式< xsl:template >和<xsl:apply-templates>

选择模式

1. < xsl:for-each >

XSL 提供了这样的具有程序语言性质的语句< xsl:for-each>来选定唯一特定的标记或者具有相同名称的一系列标记称为结点集，而且这种选定可以循环进行直到 XML“原始树”的结束。它的具体语法如下：

< xsl:for-each select="pattern" order-by="sort-criteria-list">

属性说明：

select:表示满足 select 条件，使用此样式描述。

order-by:表示对由 select 选定的标记的子标记进行排序。"sort-criteria-list"是该标记的子标记列表。在列表元素前添加加号（+）表示选定标记的子标记该列表的内容以升序排序，添加减号（-）表示逆序排序。

当然上述操作只是对 XML 的原始树的元素或者说是标记进行了，而并没有施加样式。如果要对其施加样式，则应该按照如下格式进行：

```
< xsl:for-each select="xml_mark" >
< !--样式定义-- >
< /xsl:for-each >
```

我们可以看如下一个简单的例子：

```
< xsl:for-each select="东方不败" >
<TR>
<BR><xsl:process-children/></BR>
</TR>
< /xsl:for-each >
```

参照 HTML 标记，上述的 XSL 语句的含义是不难理解的。当然我们也可以采取另外一种方式。

例:

```
<TR>< xsl:for-each select="东方不败" ><TR>
```

2. < xsl:value-of >

< xsl:for-each >模式只是选取节点，并没有取出节点的值，而< xsl:value-of >能够取出节点的值。它的具体语法如下：

```
< xsl:value-of select="pattern" >
```

属性说明：

select:用来与当前上下文匹配的 XSL 式样。简单地讲，如果要在 XSL 文档某处插入某个 XML 标记（假定是 xml_mark 标记）的内容，可用如下方式表示：

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

同样我们通过这一语句实现对网页元素（标记）的风格样式的施加，可以采用下列方式。

```
< xsl:for-each select="东方不败" >
<P>
<TR>
<BR><xsl:value-of select= "风云再起" ></BR>
</TR>
<P>
< /xsl:for-each >
```

有了简单的选择模式，我们已经可以对 XML 文档的数据进行简单的格式化输出了，我们可以选取前面的 Reference.xml(程序清单 5.19)文档作为这种格式化输出的例子。在实现这种输出之前，需要对 Reference.xml 做少许修改。

将其中的第二行

<?xml-stylesheet type="text/css" href="example1.css"?>

改为如下的语句：

<?xml-stylesheet type="text/xsl" href="example1.xsl"?>

然后在同一目录下书写如程序清单 5.25 所示的 XSL 文件就可以了。

程序清单 5.25(example1.xsl)。

```
< ?xml version="1.0" encoding="GB2312"? >
< HTML xmlns:xsl="http:www.w3.org/TR/WD-xsl">
< HEAD >
< TITLE >客户信息 < /TITLE >
< /HEAD >
< BODY >
< xsl:for-each select="customer" >
< P/ >
< TABLE border="1" cellspacing="0" >
< CAPTION style="font-size: 150%; font-weight: bold" >
客户信息记录
< /CAPTION >
< TR >
< TH >姓名< /TH >< TD >< xsl:value-of select="name"/ >< /TD >
< TH >姓名< /TH >< TD >< xsl:value-of select="name"/ >< /TD >
< TH >编号< /TH >< TD >< xsl:value-of select="id"/ >< /TD >
< TH >地址< /TH >< TD >< xsl:value-of select="address"/ >< /TD >
< TH >电子邮件< /TH >< TD >< xsl:value-of select="email"/ >< /TD >
< TH >电子邮件< /TH >< TD >< xsl:value-of select="email"/ >< /TD >
< /TR >
< TR >
< TH >订购的货物名称及描述< /TH >< TD colspan="5" >< xsl:value-of select="goodsl"/ >< /TD >
< /TR >
< /TABLE >
< /xsl:for-each >
< xsl:for-each select="customer" >
< P/ >
< TABLE border="1" cellspacing="0" >
< CAPTION style="font-size: 150%; font-weight: bold" >
客户信息记录
< /CAPTION >
< TR >
< TH >姓名< /TH >< TD >< xsl:value-of select="name"/ >< /TD >
< TH >编号< /TH >< TD >< xsl:value-of select="id"/ >< /TD >
< TH >地址< /TH >< TD >< xsl:value-of select="address"/ >< /TD >
< TH >电子邮件< /TH >< TD >< xsl:value-of select="email"/ >< /TD >
< TH >电子邮件< /TH >< TD >< xsl:value-of select="email"/ >< /TD >
< TH >电话< /TH >< TD >< xsl:value-of select="telephone"/ >< /TD >
```

北京希望电子出版社

```
< /TR >
< TR >
< TH >订购的货物名称及描述< /TH >< TD colspan="5" >< xsl:value-of select="goodsl"/ >< /TD >
< /TR >
< /TABLE >
< /xsl:for-each >

< /BODY >
< /HTML >
```

测试模式

XML 技术的优势之一就在于数据输出的可选择性，即选择需要的数据输出。如果我们对 XML 数据不需要全部输出，而只需要其中的满足某条件的部分数据，那么条件判断<xsl:if>与多条件判断<xsl:choose>及<xsl:when>则迎合了这种需要，这种模式与程序设计中的流程控制的思想有异曲同工之妙。

1. <xsl:if>

XSL 元素<xsl:if>的语法结构如下：

```
<xsl:if expr="script-expression" language="language-name" test="pattern">
```

属性说明：

expr: 脚本语言表达式，计算结果为“真”或“假”；如果结果为“真”，且通过 test，则在输出中显示其中内容（可省略此项属性）。

language: expr 属性中表达式的脚本语言类型，其取值与 HTML 标记 SCRIPT 的 LANGUAGE 属性的取值相同，缺省为“JScript”。test——源数据测试条件。

test : 设定条件的叙述表达式。只有当 script 属性的设定值传回“true”时（或 test 属性设置的条件成立），<xsl:if>中的内容才会被 XSL 处理器处理。

我们可以看下面一个简单的 XSL 片段，来加深对这个语句的理解。

例:

```
<xsl:if test=".[@sex='男']">
    <td>男<xsl:value-of /></td>
</xsl:if>
```

该片段的含义是当 test 属性设置的条件".[@sex='男']"成立时，XSL 语句

```
<td>男<xsl:value-of /></td>
```

才会被应用到该测试所在的标记上。

2. <xsl:choose> 、<xsl:when>和<xsl:otherwise>

这 3 个元素是用来设定较为复杂的条件式，一般共同配合使用。

（1）< xsl:choose >

语法：

```
< xsl:choose >
```

属性说明：

无属性值，表示一个多选测试的开始。

（2）< xsl:when >

语法：

< xsl:when expr="script-expression" language="language-name" test="pattern" >

属性说明：

expr：脚本语言表达式，计算结果为“真”或“假”；如果结果为“真”，且通过 test，则在输出中显示其中内容（可省略此项属性）。

language：expr 属性中表达式的脚本语言类型，其取值与 HTML 标记 SCRIPT 的 LANGUAGE 属性的取值相同，缺省为“JScript”。

test: 源数据测试条件。

（3）< xsl:otherwise >

语法：

< xsl:otherwise >

属性说明：

无属性值。

在一个多选测试中，如果没有不满足< xsl:when >规定的条件，如果在最后有此标记，则输出此标记中的内容。请看下面简单的 XSL 片断。

例:

```
<xsl:choose>
<xsl:when test="mountain">
<td>高山<xsl:value-of /></td>
 </xsl:when>
<xsl:when test="valley">
<td>峡谷<xsl:value-of /></td>
</xsl:when>
<xsl:when test="plain">
<td>平原<xsl:value-of /></td>
</xsl:when> <xsl:otherwise>
<td>盆地</td>
</xsl:otherwise>
</xsl:choose>
```

XSL 模板与匹配模式

< xsl:for-each >、< xsl:value-of >虽然可以对 XML 数据实现简单的格式化输出，但如果遇到比较复杂的 XML 格式输出，将 XSL 按照要求依次写下来的话，一是设计困难，可扩展性差，不利于人员之间的分工协作；另外，可修改性很差，可能会出现牵一发而动全军的情况，不利于维护。因此，在这里我们就需要采用程序中模块化设计逐步细化的方法，来克服这种困难的局面。

1．书写模板 < xsl:template >

它的具体语法如下：

< xsl:template match="node-context" language="language-name" >

属性说明：

match:确定什么样的情况下执行此模板。作为一种简化的说明，在此处使用标记的名字；其中最上层模板必须将 match 设为“/”。

language:　确定在此模板中执行什么脚本语言，其取值与 HTML 中的 SCRIPT 标记的 LANGUAGE 属性的取值相同，缺省值是 JScript。

< xsl:template > 用 match 属性从 XML 选取满足条件的节点，针对这些特定的节点形成一个特定输出形式的模板。

2. 调用模板 <xsl:apply-templates>

它的具体语法如下：

< xsl:apply-templates select="pattern" order-by="sort-criteria-list" >

属性说明：

select: 确定在此上下文环境中应执行什么模板，即选取用< xsl:template >标记建立的模板（块）。

order-by:以分号（;）分隔的排序标准，通常是子标记的序列。具体的含义可以参照选择模式中的说明。

下面我们通过一个例子来对这种模板的使用，加以说明。对程序清单 5.27 的 XML 文档采用 example2.xsl 风格样式表。

程序清单 5.26(Reference1.xml)

```
<?xml version="1.0" encoding="gb2312" ? Standalone="no" >
<?xml-stylesheet type="text/css" href="example2.xsl"?>
<!DOCTYPE  information  SYSTEM "customer.dtd">
<information>
    <customer>
    <name>李超群</name>
    <name>雷雨风</name>
    <id>1</id>
    <address>当代商城</ address >
    <email>lichaoqun@sina.com</email>
    <email>leiyufeng@sohu.com</email>
    <goods>钻石主板2代</goods>
    </customer >
</information>
```

程序清单 5.27(example2.xsl)

```
< ?xml version="1.0" encoding="GB2312"? >
< xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" >
< !--根模板-- >
< xsl:template match="/" >
< HTML >
< HEAD >< TITLE >客户信息< /TITLE >< /HEAD >
< BODY >
< xsl:apply-templates select=" information/ customer "/ >
```

```
< /BODY >
< /HTML >
< /xsl:template >

< !—客户信息模板-- >
< xsl:template match="customer" >
      < TABLE border="1" cellspacing="0" >
      < CAPTION >
      客户信息< xsl:eval >formatIndex(childNumber(this)，"I")< /xsl:eval >
      < /CAPTION >
      < xsl:apply-templates select="name" / >
      < xsl:apply-templates select="id" / >
      < xsl:apply-templates select="address" / >
      < xsl:apply-templates select="telephone" / >
      < xsl:apply-templates select="email" / >
      < TR />
      < TD >订购货物< /TD >
      < TD COLSPAN="5" >
      < TABLE cellspacing="0" >
      < xsl:apply-templates select="goods"/ >
      < /TABLE >
      < /TD >
      < /TABLE >
      < BR/ >
< /xsl:template >

<!--姓名模板-- >
< xsl:template match="name" >< TD >姓名< /TD >
< TD >< xsl:value-of/ >< /TD >
< /xsl:template >

< !--客户编号模板-- >
< xsl:template match="id" >< TD >客户编号< /TD >
      < TD >< xsl:value-of/ >< /TD >
< /xsl:template >

< !--客户住址模板-- >
< xsl:template match="address" >< TD >住址< /TD >
      < TD >< xsl:value-of/ >< /TD >
< /xsl:template >

< !--客户电话模板-- >
< xsl:template match="telephone" ><TD>电话号码</TD>
```

北京希望电子出版社

```
        < TR >< TD >< xsl:value-of/ >< /TD >< /TR >
    < /xsl:template >

    < !--客户电子邮件模板  -- >
    < xsl:template match="email" ><TD>电子邮件</TD>
        < TR >< TD >< xsl:value-of/ >< /TD >< /TR >
    < /xsl:template >

    < !--客户订购货物模板-- >
    < xsl:template match="goods" ><TD>订购货物</TD>
        < TR >< TD >< xsl:value-of/ >< /TD >< /TR >
    < /xsl:template >

    < /xsl:stylesheet >
```

在浏览器我们可以看到这种风格样式表作用下的网页的实际效果。

XSL 模式化查询语言其他模式的简单介绍

除了上面的 3 种基本模式以外，XSL 还有其他一些特殊的模式，来增强它本身的灵活性。主要的有下面几个：

（1）<xsl:comment>。在此元素中的内容，XSL 将它作为注释信息，并不显示在浏览器中。

（2）<xsl:copy>。从 XML 文件中拷贝标记中的信息到输出的文件中。无属性。利用这种模式可以将 XML 文档中的非标记信息读取出来，利用该标记将其显示出来。

（3）<xsl:attribute>和<xsl:element>。可以在标记中附加一个属性名称或新建一个标记。XSL 样式表不但可以引用 HTML 标记，而且也可以建立新的标记和属性，然后一起结合 XML 文件中的信息来显示。其中，<xsl:attribute>是在标记中新增一个属性，<xsl:element>是新建一个标记。它们有共同的属性 name，XML 文档没有提供的标记可以通过<xsl:element name="element_name">的格式添加；同样的也可以为标记添加一个新属性 ，<xsl:attribute name="attribute_name">可以实现这一功能。其中 name 代表了所添加的元素或属性的名称。

关于 XSL 的其他问题

讲到这里，我们对 XSL 的基本问题可以说已经搞清楚了，但是 XSL 的内容远远不止这些，读者要想对 XSL 做更进一步的了解，还需要学习有关 XSL 的运算符、关系与逻辑运算符、特殊字符，以及 XSL 的各种函数，多个 XSL 对于 VBScript、JScript 增加的方法、属性等方面做深入的学习，鉴于篇幅对 XSL 的介绍也只能到此为止。

当然如果读者对 HTML 非常熟悉，那么对于 XSL 的掌握是很有帮助的，因为 XSL 文件本身就是 HTML 文档，二者在某种程度上是相通的。比如我们如果掌握了常用的文字相关属性表常用的版面排列属性，那么在书写 XSL 文件时就会胸有成竹，而且使自己的 XML 文档表现得更加丰富多彩。

因为 XSL 的书写格式非常严格，因此建议读者在书写 XSL 文件时注意如下问题：

（1）任何 XSL 文件的根元素为< xsl:stylesheet>，文件必须以< /xsl:stylesheet>结束。

（2）任何 XSL 文件的题头必须为

```
< ?xml version="1.0" encoding="GB2312"? >
< xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl" >
```

其中 xmlns:xsl="http://www.w3.org/TR/WD-xsl"这一句主要用来说明该 XSL 样式表是使用 W3C 所制定的 XSL，设定值就是 XSL 规范所在的 URL 地址。

（3）XSL 中出现的任何 HTML 标记必须成对出现，否则会引起错误。

5.4　XHTML 简介

HTML 应用至今已有多年的历史，尤其是近几年来，HTML 在 Internet 中扮演着越来越重要的角色。然而随着网络技术的不断发展，出现了各种手持设备和 PC 边缘产品访问 Internet 的需求，此时，人们发现现有的 HTML 的功能显得越来越薄弱，甚至根本无法满足人们的需求。于是，技术的发展使得人们不得不去寻找一种新的、能更好地满足移动设备上网需求的标记语言，这种新的标记语言就是 XHTML。

5.4.1　什么是 XHTML

XHTML 可以简单地理解为 HTML in XML，但是 XHTML 并不代表一种标记语言，而是一系列标记语言的统称，因为它们都源自且兼容 HTML 4，并扩展了 HTML 4，以更好地满足人们的需求，因此称为 XHTML。在 W3C 的 XHTML 的规范说明中是这么定义 XHTML 的：“XHTML 包括一系列的文档类型说明（DTD）和模块，这些 DTD 和模块是从 HTML 4 中重新生成的，是 HTML 4 的子集，并做了扩展。XHTML 的 DTD 都是基于 XML 的，并且最终的目的是作为各种 XML 应用的粘合剂。”

具体地说：将现在的 HTML 4 规范划分成不同的子集，例如：专门处理文本（Text）的 DTD、专门处理表格的 DTD 等等，并将这些子集用 XML 规则重新编写，即所有这些从 HTML 4 中划分出来的 DTD 同 HTML 4 中关于各部分的定义是一致的，不同的是这些 DTD 是用 XML 的 DTD 定义规则重新编写的，并且加进了 XML 中的一些优良特性，比如：一致性、严密性等。

HTML 与 XHTML 的主要差别是后者的可扩展性。XHTML 的扩展性试图提供一个定制 XHTML 的标准方法，以满足文档或者特殊客户的要求。为了提供可扩展性，XHTML 是在可结合和扩展元素的子集中定义的，这使得网页设计人员和网页编程人员的工作量大为降低，从记忆大量的 HTML 标记中解放出来。只要为实际的网页内容选择适当的模块，拥有客户的配置文件信息，网页制作人员在获得这两方面的信息之后，就能使开发出来的网页在客户机上获得广泛的支持。

5.4.2　XHTML 的作用

最初 W3C 研究 XHTML 的目的是打算让 XHTML 充当各种 XML 语言应用的粘合剂，例如：在一种 XML 应用中，可以用 XHTML 结合 SVG 标记或其他 XML 标记，如 SMIL、CFML。

但是 W3C 在制定 XHTML 标准的过程中发现，XHTML 可以成为一种标准，使得只要是符合这个标准制作的 Web 页面就可以用各种浏览器浏览，包括桌面浏览器和各种手持设备；而且用 XHTML 制作的 Web 页面在遵循少量原则的情况下，可以直接用现有的桌面浏览器浏览，不必为 XHTML 研制专门的浏览器，很容易地就做到了向下兼容。

由于 XHTML 是用 XML 规则编写的，因此 XHTML 页面的制作要遵守 XML 语法，如：

（1）区分大小写：如 Windows 平台下，一个页面若是包含< ?xml version="1.0" encoding="GB2312"? >，表示这个页面是 XML 页面，浏览器将调用 Windows XML Parser 解释器显示 XML 页面，如果将指令中的 xml 换成 XML，则会出现“无法显示 XML 页面”错误。

（2）任意一个标记都必须有结束标记相对应。

（3）对于空标记要在“>”前加“/”。

（4）对于所有的属性值都必须用引号引起来等。

5.4.3　XHTML 的模块划分

因为 XHTML 是用 XML 重写了 HTML 4，根据 XML 的制作规则，要为每个 HTML 4 支持的数据类型重新定义，因而出现了另一个文档——modularization of　XHTML。

这个文档说明了 XHTML 由一组模块组成，每个模块描述了一种数据类型，并包括应用在这种数据类型上的标记。应用这些模块，可以标记各种文档元素，如：标题、段落、列表等。每个模块都由一组标记组成，例如：对于文本模块，应该包括 H1-H6、P、PRE、BLOCKQUOTE 等 HTML 标记。对于其它 HTML 内容，应该有相应的模块，如：列表模块——包括专门制作列表的标记；基本的表单模块；表格模块；脚本模块等。

5.4.4　XHTML 中关于表单（Form）的定义

除了将 HTML 4 中的标记进行了分类，XHTML 规范中还包括一个新的关于 Web 表单的规范——XForms。

在电子商务应用中，一个订单往往要在多个部门或多个机构之间传递，并且每个部门都可能要在订单中加入自己部门的认可信息，因此，对于像电子商务这样的工作流应用来说，Web Form 应该更小巧，具备可携带性、易处理等特性。显然这些特性用 HTML 4 的 Form 机制是不可能实现的，目前用 HTML 4 制作的 Form 都是将脚本、内容、显示样式以及注释等混合在一起，既庞大又不清晰，因此 W3C 针对目前 Web Form 的现状以及应用的需要，制定了新的 Form 标准——XForms。

XForms 最主要的特性就是将表单的数据、处理逻辑和显示控制分开，XForms 的所有内容只和定义好的数据模型相关，而不涉及任何显示标记，包括处理逻辑。对于这一点，一个现成的例子就是 XML，一个用 XML 编制的 Web 页面，至少包括 3 个文件：一个是文档类型定义（.dtd），即此页面中共用到了几个元素，各自的名字是什么，分别有什么样的属性；另一个是内容文件（.xml），用前一个文件定义的元素标记填写内容，有点像目前的 HTML 文件；最后一个文件是显示格式文件（CSS 和 XSL），定义内容文件中的内容按什么样式显示。

W3C 关于 XForms 的应用需求说明还包括：随着嵌入式技术的发展，在各种 PC 外设中嵌入 Web Server 已经不是难事，例如：在一个打印机中嵌入 Web Server，某个用户如果需要用这个打印机打印某些东西，那么这个用户可以使用嵌入打印机的 Web Server 中 XForms 提供的用户配置界面，调整打印机的一些参数，如：页边距、页码等，而此时，这个用户可能是通过 Internet 访问到这台打印机的。

5.4.5　XHTML 的前景

未来的 Web 技术将是除 PC 之外的各种设备都能访问 Web，如：蜂窝式移动电话、寻呼机、汽车中的计算机、掌上电脑等。随着这些能访问 Web 页面设备的出现，Web 页面的作者自然要考虑如何将为桌面电脑设计的页面正确显示在一次只能显示几行信息的小型设备上，现在流行的 WAP 就是用 WML 语言将现有的 HTML 页面内容转换成手机能接收、显示的技术。除了手机，其他设备也需要有适合自己的专用 HTML 语言，当然，针对不同设备制定不同标记语言的做法是不现实的，为了实现多种设备的 Web 访问能力，W3C 在 XHTML 规范中提出了一种解决方案。

用标记区分不同的访问设备

在 W3C 的解决方案中，首先要给各种不同的设备一个不同的标记。假设：用字符 pc 代表台式机，mp 代表移动电话，这样做的目的在于，当一个设备访问某一 Web 页面时，首先将这种设备的标记告诉 Web 页面，那么 Web 页面将根据这种设备的特性，如：有无显示装置，有无存储设备，一次可以接收多少字节等，选择这种设备能够接受的标记语言，重新建立 Web 页面，然后发送给这个设备。这样一来，无形之中为各种可访问 Web 的设备建立了一个统一的标准，所有的 PC 机用桌面浏览器的标准，所有的手机用手机的标准。

设备描述文件（Device Profile）

怎样根据设备的特性重建 Web 页面呢？这就需要设备描述文件（Device Profile）来帮忙。XHTML 标记集并不知道哪个设备能正确显示它，因此，需要我们事先为所有的设备定义好描述文件，这个描述文件将告诉 Web 系统某个设备支持 XHTML 的哪个模块，甚至包括该使用哪种语言显示页面。

那么使用设备描述文件告诉 Web 系统某个设备使用 XHTML 的哪个模块后，Web 页面就能正确地在这种设备上显示吗？当然不行。为什么？因为 XHTML 使用的是 XML 语法，对于 Web 页面的显示是由专门控制显示输出的文件来控制的。在现在的 XML 页面里，如果没有显示的样式单，浏览器就会像是一面镜子，XML 文件中写的文本是什么，就显示什么。显然，在 XHTML 中除了设备描述文件，还需要一个像 CSS 和 XSL 一样的文件，规定在某种设备中如何显示 XHTML 页面，这个文件就是文档描述文件（Document Profile）。

文档描述文件（Document Profile）

文档描述文件和设备描述文件一起，指出访问 Web 页面设备支持的 XHTML 模块集，使用何种显示样式单，以此保证 Web 站点为指定设备提交正确的 Web 页面。即使某种设备没有文档描述文件，W3C 规定，要么提供用户配置界面，由用户选择一种最适合这个设备

的文档描述文件，要么干脆根据用户的需要生成一个新的文档。

通过以上关于设备标记、设备描述文件和文档描述文件的说明，我们可以总结出 W3C 关于不同设备浏览同一 Web 页面的工作过程: 当某个设备想浏览某一 Web 站点时，在同站点建立连接的过程中，站点首先获取这种设备的设备标记，以决定使用哪种设备描述文件和文档描述文件；然后，站点调出和这种设备匹配的设备描述文件和文档描述文件，使用某种转换工具，用嵌入在这种设备中的浏览器能识别的标记语言重建 Web 页面，然后将页面提交给这个设备。整个工作过程的一个应用实例就是目前炒得很热的 WML 标记语言和 WAP 协议。

上面提到的转换工具完成这样的功能：由于不同的设备使用的浏览器不一样，自然标记语言也不一样，在站点确定这个设备的设备描述文件和文档描述文件后，转换工具根据这两个文件使用设备支持的标记语言重新生成 Web 页面。同时，工具的位置既可以位于 Web 站点上，也可以位于 ISP 的代理服务器上，还可以内嵌在客户端的浏览器中。

可以预言，如果 XHTML 能迅速成长起来的话，那么不久的将来，不仅整个互联网将会呈现出崭新的面貌，而且互联网经济也将有飞跃性的发展。

5.4.6　XHTML 文档的一个简单的实例

现在所使用的 HTML 是用 SGML 语言定义的，是 SGML 的一个应用实例。XHTML 是用 XML 语言重新定义了的 HTML，是 HTML 的 XML 版本，有人称其为“HTML in XML”。XHTML 中的标记定义与 HTML 中的相同，但语法比 HTML 要严格的多，因为它基于 XML 语言，必须和 XML 地其他规范相吻合。

为了使读者对 XHTML 的特性有所了解，下来我们举一个实例说明。源代码如程序清单 5.28。

程序清单 5.28(example.xhtml)

```
<? xml version="1.0" encoding="gb2312"?>
<!DOCTYPE html PUBLIC"-//W3C//DTD XHTML 1.0 Strict//EN"   "DTD/strict.dtd"   >
<html>
<head>
<title>
    XHTML：The Extensible HyperText Markup language!
</title>
<link rel="stysheet" href="W3C-WD.css" type="text/css"/>
<?xml-stylesheet href="lg.css" type="text/css"?>
</head>
<body>
    <p>
        <a href="http://www.pku.edu.cn/"><img src="pku.bmp" alt="pku"/></a>
</p>
<h1>XHTML Sample </h1>
<h2 class="spec">
                <a href ="http://www.w3.org/TR/xhtml1/">XHTML<sup>&#8482;</sup>1.0:
```

```
                    The Extensible HyperText Markup Language
                </a>
</h2>
<p>
This page will be revised to include errata for the XHTML 1.0 specification，
See:
<a href="http://www.w3.org/TR/xhtml1/"> htttp://www.w3.org/TR/xhtml1/</a>
</p>
</body>
</html>
```

其中样式表文件 lg.css 代码如程序清单 5.29。

程序清单 5.29（lg.css）

```
body
{
   backpround : top left no-repeat url(WD.gif)#FFFFFF;
   margin-left:10%;
   margin-right:10%
   font-family:sans-serif
}
h1，h2，h3，h4，h5，h6
{
   margin-left:-4%;
   font-family:sans-serif;
   color:rgb(0，122，255);
}
h1
{
   margin-top:3em
}
A:link{color:rgb(0，0，153)}
A:visited{color:rgb(153，0，153)}
A:active{color:rgb(255，0，122)}
A:hover{color:rgb(0，96，255)}
.spec a {text-decoration:none}
img
{
   color:white;border;none;
}
code
{
            font-family:monospace;
            font-weight:bold;
}
```

北京希望电子出版社

```
em
{
   font-style:italic;
   font-weight:bold;
}
strong
{
   text-transform:uppercse;
   font-weight:bold;
}
u，ins，.add
{
   text-decoration:underline;
   color:rgb(153，0，0);
}
s，del，.remove
{
   text-decoration:line-through;
   color:rgb(192，192，0);
}
pre
{
   font-family:monospace;
   color:#006600;
   font-weight:bold;
}
p.caption
{
   font-weight:bold;
   font-style:italic;
}
blockquote
{
   color:navy;
   font-family: "Comic Sans MS"，sans-serif;
}
div.contents li
{
list-style-type:none
}
.contents
{
   background-color:rgb(204，204，255);
```

北京希望电子出版社

```
    padding:0.5em;
    border:none;
    width:100%;
    margin-right:5%;
}
.good
{
    border:solid:
    border-width:2px;
    border-color:green;
    color:#006600;
    margin-right:5%;
    margin-left:0;
    margin-bottom:10px;
    padding:0.3em;
}
.bad
{
    border:solid;
    margin-right:5%;
    border-color:red;
    margin-left:0;
    color:rgb(192，101，101)
    margin-bottom:10px;
    padding:0.3em;
}
td，th
{
    font-family:sans-serif;
    background:rgb(255.255，153);
    border-color:rgb(255，255，255);
}
```

将 example.xhtml 文件保存为 text.html，然后就可以在浏览器中查看运行结果了。

5.5　WML 简介

5.5.1　WML 的基础 WAP

无线应用协议 WAP(Wireless Application Protocal)是制定在无线通讯设备(例:行动电话、PDA 等)来执行 Internet 网路存取服务的开放标准。这是一个使用户借助无线手持设备，如掌上电脑、手机、呼机、双向广播、智能电话等，获取信息的安全标准 WAP 所使用的协议，类似 HTTP 协议，但主要针对无线通讯设备所开发的，因为无线设备频宽有限、屏幕也较小，因此，要制定专门的协定来支援。

北京希望电子出版社

只要手机能支援 WAP，就可以经由无线通讯网络存取网络资源，未来可透过手机在户外进行各项交易。WAP 支持绝大多数无线网络，包括 GSM，CDMA，CDPD，PDC，PHS，TDMA，FLEX，ReFLEX，iDen，TETEA，DECT，DataTAC 和 Mobitex。

所有操作系统都支持 WAP，其中专门为手持设备设计的有 Palm OS，EPOC，Windows FLEXOS，OS/9 及 Java OS。一些手持设备，如掌上电脑，安装微型浏览器后，可借助 WAP 接入 Internet。 微型浏览器文件很小，可较好地解决手持设备内存小和无线网络带宽不宽的限制。虽然 WAP 能支持 HTML 和 XML，但 WML 才是专门为小屏幕和无键盘手持设备服务的语言。WAP 也支持 WMLScript。这种脚本语言类似于 JavaScript，但对内存和 CPU 的要求更低，因为它基本上没有其他脚本语言所包含的无用功能。

WAP 规范是一种无线应用程序的编程模型和语言，它第一次定义了一个开放的标准结构和一套用来实现无线 Internet 接入的协议。WAP 规范的要素主要包括：WAP 编程模型、遵守 XML 标准的无线标记语言（WML）、用于无线终端的微浏览器规范、轻量级协议栈、无线电话应用（WTA）框架。

这个模型在很大程度上利用了现有的 WWW 编程模型，应用开发人员可以从这种模型中得到许多好处，包括可以继续使用自己熟悉的编程模型、能够利用现有的工具（如 Web 服务器、XML 工具）等。另外，WAP 编程模型还针对无线环境的通信特点，对原有的 WWW 编程模型进行了优化和扩展。

WAP 规范优化和扩展了现有的 Internet 标准。WAP 组织针对无线网络环境的应用对 TCP/IP、HTTP 和 XML 进行了优化，现在它已经将这些标准提交给了 W3C 联合会作为下一代的 HTML（HTML -NG）和下一代的 HTTP（HTTP-NG）。

在设计 WAP 网页时不论你使用的 WAP 开发工具是 UP .SDK 或 NOKIA WAP TOOKIT 或者 ERICSSON WAP IDE，你都必须配置进行 Web 服务器，使得 Web 服务器可以支持 WML 文件类型。故在此将常见的几种 Web 服务器的设置介绍如下：

（1）Windows NT 平台的 IIS 服务器。

<1>启动菜单->程序->WINDOWS NT OPTION PACK ->IIS5.0。

<2>右击 Internet Information Server 的子项计算机名，在快捷菜单上选择“属性”。

<3>在属性页面的下部，有一个“文件类型（F）”按钮，单击此按钮，会出现文件类型界面。

<4>单击“新增类型”按钮，然后在相关的扩展名栏中填写 .wml ，在内容类型（MIME）栏中填写 text/vnd.wap.wmle、单击“确定”按钮。

（2）Apache Web Server。

<1>不管是 NT 平台 还是 UNIX 或 LINUX 平台，都是修改 Apache 安装目录下的 conf/mime.types 文件。

<2>在该文件中增加以下内容：

text/vnd.wap.wml .wml image/vnd.wap.wbmp .wbmp

application/vnd.wap.wmlc .wmlc text/vnd.wap.wmls.wmls

application/vnd.wap.wmlsc .wmlsc

<3>存盘。

<4>重新启动 Apache Web Server 即可。

5.5.2　WML 入门

程序清单 5.30 是一个 WML 的简单的例子，读者可以结合前面的 HTML 以及 XML 的知识，对 WML 的特点和内容有一个初步的印象。

程序清单 5.30(example1.wml)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
  "http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
  <template>
      <do type="prev" label="back">
        <prev/>
        <!--provide a button you can clink to back a step-->
      </do>
  </template>
<card id="friends" title="Hot link">
  <p>
      <a href="http://wap.sian.com.cn/">Sina WAP</a><br/>
      <a href="#nextcard">Next Card</a>
  </p>
</card>
<card id="nextcard">
  <p>
    this is the second card.
  </p>
</card>
</wml>
```

通过上面的示例读者应该了解到以下内容。

语法：WML 的语法与 HTML 相似，仍然是一种标记语言，而且延续了 XML 语法规则。

元素：在 XML 和 WML 语言中，语言的基本元素称之为“标签”。标签必须被“<”和“>”括起来。大多数标签都包括“起”、“止”两部分，例如： … 。某些特殊标签可以只有一个标签，但是必须有结束标记，例如：<prev/>。

属性：XML 语言的标签可以包含很多属性，给标签提供必要的附加信息属性内容通常在起始标签内使用。属性只作为参数为标签提供必要的信息，不会被浏览器显示。属性的值需要被引号括起来，可以是单引号或者双引号，引号可以成对嵌套使用。例如：<card id="friends" title="Hot link">。

注释：注释内容是方便制作者阅读源代码，不会被浏览器显示。和 XML 一样，WML 不支持注释嵌套。注释的格式与 HTML 以及 XML 相一致。

文档结构：WML 文档是由 Card 和 Deck 构成的，一个 Deck 是一个或多个 Card 的集合。在得到客户终端的请求之后，WML 从网络上把 Deck 发送到客户的浏览器，访问者可

以浏览 Deck 内包含的所有 Card，而不必从网上单独下载每一个 Card。

5.5.3　WML 语法

下来将详细地讲述 WML 文档的各个方面，使读者掌握 WML 文档的用法。

由于前面的 WAP 我们已经介绍过了，相信读者应当能够理解。因此，这里直接叙述 WML 语言。WML 语法主要有如下几部分。

基本规则

<1>WML 使用 XML 文档字符集，目前支持 Unicode 2.0，WML 的所有标签、属性和规定的可接收值必须小写，CARD 的名字和变量也是区分大小写的。对于连续的空字符，只显示一个空格。

<2>标签内属性的值必须用符号“或”括起来，属性名、“＝”和值之间不能有空格。对于不成对出现的标签，必须在“>”前加“/”，比如
。

<3>对保留字符的处理，对应的取代字符见表 5.3。

表 5.3　保留字符

保留字符	对应的取代字符
<	&lt
>	&gt
?	&apos
"	&quot
&	&amp
$	$$
空格	
-	&shy

基本格式和文件头

WML 文件的一般格式如下。

例:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
<head>
        <access/>
        <meta..../>
</head>
    <card>
        Some contents...
    </card>
```

<wml>

<1> WML 结构看上去和 HTMl 文件很类似，对于每一个 DECK，在其文档开头必须指明以下的类型声明。

例:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
```

千万注意字母的大小写。对于一个 DECK，其文件大小最好不要超过 1.2K。

<2><wml>标签和 HTML 中的<html>标签一样，用来表明这是一个 WML 的 DECK，它有一个可选的 xml:lang 属性来制定文档的语言，比如<wml xml:lang="zh">表示文档语言为中文。

<3><head>标签包含了该 DECK 的相关信息。<head>标签之间可以包含一个<access>标签和多个<meta>标签。<access domain="域" path="/路径" />相当于 HTML 中的<BASE>标签，指定该 DECK 的访问控制信息，它用两个可选的属性，domain 用来指定域，默认值为当前域，path 用来指定路径，默认值为“/”，即根目录。由于<access>单独使用，所以要用“/”结尾，以后对于类似的情况不再赘述。

<meta 属性 content="值" scheme="格式" forua="true|false"/>和 HTML 中的类似，提供了该 DECK 的 meta 信息。属性是必选的，包括一下三种情况:

name="name": UP.Link Server 忽略 meta 数据。

http-equiv="name": UP.Link Server 将 meta 数据转为 HTTP 响应头(同 HTML) 。

user-agent="agent":UP.Link Server 直接将 meta 数据传给手机设备。

content 属性也是必选的，其内容根据属性而定。scheme 属性目前尚不支持。forua 为可选属性。

目前支持的 meta 属性如下所示:

```
<meta http-equiv="Cache-Control" content="max-age=3600"/>
```

指定 DECK 在手机内存缓存中的存储时间段，默认的为 30 天(除非内存耗尽)，在该期间，手机对于访问过的 DECK 直接从缓存里调用。如果信息是对时间敏感的，可以用 max-age 指定 DECK 在缓存里的生存期，最小单位是秒，如果指定为 0，则每次都需通过连接服务器来调用该 DECK。

```
<meta user-agent="vnd.up.markable" content="false"/>
```

和

```
<meta user-agent="vnd.up.bookmark" content="指定的 URL"/>
```

类似于普通浏览器的书签功能。当用户将一个 CARD 做了书签后，手机浏览器首先用一个标记记录该 CARD，这个标记默认的是<card>标签中的 title 属性(以后会讲到)，然后当用户选择了该书签以后，浏览器就会打开被记录的 URL。但是因为在默认的情况下，手机会记录所有的 DECK，所以，一般<meta>被用来使手机不要记录当前的 URL，即<meta user-agent="vnd.up.markable" content="false"/>。此外，如果要为书签指定不同于当前 DRECk 的 URL，用<meta user-agent="vnd.up.bookmark" content="指定的 URL"/>。

<4>一个 DECK 可以包含多个 CARD，每个 CARD 的内容可能不止一屏显示，注意

DECK，CARD 和屏幕显示范围的关系。一个 CARD 用<card>和</card>包含。<card>可以包含以下可选的属性。

<card id="name" title="label" newcontext="false" ordered="true" onenterforward="url" onenterbackward="url" ontimer="url">

id 属性用来指定 CARD 的名字，可用来在 CARD 间跳转，相当于在 HTML 中在页内跳转时用<A NAME="jumpHere">。

title 属性用来作为书签的标记，该属性一般不会显示在屏幕上。

newcontext 属性，默认值为 false，用来指示当跳转到本 CARD 时，手机是不是要清除以前保留的信息，包括变量，堆栈里的历史记录，重新设置手机状态等。

ordered 属性，默认值是 true，表明该 CARD 里的内容是按固定的顺序显示，还是按用户的选择来显示。这样做是为了方便填表单，当 ordered 设置为 true 时，如果一个表单的内容不能在一屏里显示完，就分成多屏显示；当 ordered 设置为 false 时，手机可以显示一个概要 CARD 来总结有效的选项，用户可以从中选取表单选项来填写。

onXXX 属性，类似于 HTML 标签中的 onXXX 属性，用来捕捉事件，当事件被触发时执行指定的操作(任务)，在这里是转向某个 URL。

显示文本

在文本的显示上 WML 基本和 HTML 相同。文字段落包含在 和 之间，align 属性指定该段文字的对齐方式，默认的是 left，其他可选择 right 和 center；mode 属性指定当一行显示不下所有的文字时是否自动换行，默认的是自动换行 wrap，如果选 nowrap，则在一行中显示，浏览器会通过类似于水平滚动条的机制来显示所有文字。

换行标签也一样为
，这里先提一下，在标单中如果有多个<input>或者<select>，其间不要用
，否则会使手机浏览器认为是断点而把表单分页显示。

文字的修饰标签有<b>、<i>、<u>、、、<big>和<small>，意义和 HTML 里的相同。

表格的显示标签也和 HTML 相近，使用<table title="name" align="left|right|center" columns="列数">、<tr>和<td>来显示，<table>的 title 属性用来给表格取个名字，columns 属性指定表的列数，不能为 0，可选的 align 属性和前面提到的一样是对齐方式。表格中可以包含文字和图片。

程序清单 5.31(example2.wml)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
<card>
<p align="center">
<i>Hello</i><br/>
<b><i>World!</i></b>
<table title="mytable" align="right" columns="2">
```

```
<tr>
    <td>1-1</td>
    <td>1-2</td>
</tr>
<tr>
    <td>2-1</td>
    < 1000 ;td>2-2</td>
</tr>
</table>
</p>
</card>
</wml>
```

显示图片

显示图片(1 位黑白 BMP 图片)的标签一样类似于 HTML:

```
<img alt="text" src="url" localsrc="icon" align="left" height="n" width="n" vspace="n" hspace="n"/>
```

属性中 alt 和 src 是必须要有的，其他可选。另外要注意的是<img>要放在<p>里，不能放在<do>和<option>等功能键标签和选单标签里。

alt 属性用来指定当手机不支持图片显示时用来显示的文字。

src 属性指定图片的 URL，但当有了以下的 localsrc 属性时，手机浏览器就忽略 src 属性。

localsrc 属性用来指定显示存在手机 ROM 中的图标，如果找不到，则到 UP.Link Server 上去找。

可选的 align 属性用来表明图片同当前行文本的对齐方式，默认值为 bottom，可选 to 和 middle。

height、width、vspace、hspace 属性分别指定图片的长宽和距环绕文字的间距，目前不是所有的 WAP 手机都支持这些属性。

程序清单 5.32(example3.wml)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
<card>
    <p align="center"><img alt=":)" src="xxx.bmp" localsrc="smileyface"/></p>
</card>
</wml>
```

锚和任务

超链接是 HTML 页面里最基本的功能，在 WML 里也一样用和来包括用来建立连接的文字，必选属性 href 指定了要打开的 URL，可选的 title 属性给该连接取个标记名字，这个名字将作为软按钮之一的 ACCEPT 键的标记显示在屏幕的软

按钮区，所以通常可以将属性作为提示文字使用。

然而，以上的连接在 WML 里只是任务的一种情况，为了能够使用其它任务，所以引进了新的标签<anchor title="label">任务标签 文本</anchor>，<a>其实是当任务标签为<go/>时的简单表示方式。

程序清单 5.33(example5.wml)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
<card>
    <p>
    <anchor title="Link1"><go href="test1.wml"/>News</anchor><br/>
    <a title="Link2" href="test2.wml">Sports</a>
    </p>
</card>
</wml>
```

WML 里的任务标签有以下几种，除了用于<anchor>，还可以用在事件中：

<go>用来指示浏览器显示指定的 URL，包括 DECK 的第一个 CARD 或者指定的 CARD。语法如下:

```
<go href="url" sendreferer="false|true" method="get|post" accept-charset="charset">
    <postfield name="name" value="value"/>
    <setvar name="name" value="value"/>
</go>
```

其中 href 属性为必选，其他为可选。

sendreferer 属性表示是否传递调用 href 所指定的 URL 的页面的 URL，也就是当前页的 URL，即 HTTP 头中的 HTTP_REFERER，默认值为 false，可选值为 true。

Method 属性和 HTML 中的表单 FORM 的 method 属性一样，指定表单是以 get 的方式还是 post 的方式递交，以便 cgi 处理，默认的值为 get，但如果未指定 method 而<go></go>间存在<postfield>，手机自动以 post 方式传递。

accept-charset 属性可覆盖在 HTTP 头里指定的字符集，可以写多个字符集，如 accept-charset="US-ASCII，ISO-8859-1"。

<postfield name="name" value="value"/>类似于 HTML 表单 FORM 中的<INPUT TYPE="HIDDEN" NAME="变量名" VALUE="值">，通过它可以向指定的 URL 传递以“变量名/值”形式的数据。name 和 value 属性都是必选的。注意只有这里的变量是用来递交给 CGI 程序的。

除了<postfield>，还可以在<go>和</go>间加入一句或者多句<setvar name="name" value="value"/>，该语句的意思是，当触发某一事件时，给变量赋值。

要注意的是，当<go>和</go>之间没有任何语句时，要用<go/>的形式，这点比较特别。

例:

```
<anchor title="Link1"><go href="test.wml"/>News</anchor>。
```

<prev>用来将当前页面的 URL 压入 URL 历史堆栈，并打开此前的 URL，若该 URL 不存在，则<prev>无效。语法类似<go>：

```
<prev><setvar name="name" value="value"/></prev>
```

<prev>和</prev>之间可加入一句或多句<setvar name="name" value="value"/>，若不加，则必须变成<prev/>的形式。

<refresh>用来刷新当前的页面，从而使得页面内的变量刷新，语法为：

```
<refresh><setvar name="name" value="value"/></refresh>
```

<noop>，表示什么也不做，该标签不能用在<anchor>中，一般用在覆盖 DECK 级的<do>。

WML 中的控件

对程序设计有所涉及的读者应该很清楚，一般来说表单是应用程序的图形用户界面，在 WML 中，表单提供了用户与网页的交互功能。对表单的控制能力可以证明一个 HTML 设计者是否够专业，而且很多交互功能也必须依赖表单。

WML 没有表单属性，但是 WML 可以直接使用控件，同样可以达到使用表单的效果。因此，使用控件的水平可以体现一个 WML 设计者的制作水平。因此下面我们介绍的重点是 WML 支持的各种类型的控件。

WML 控件有 Select List 和 Input Box 两个系列，每个系列另外包含几个子系列，基本可以满足表单设计的需求。

1．选择列表控件（Select List）

Select 有两对很重要也很容易混淆的属性：name，value，iname，ivalue。这 4 个属性的区别和用途不太容易描述清除，看了后面的例子会很容易理解。

每个 Select 是一个或多个 Option 的集合，Option 地结果返回给 Select 元素的 name 和 iname。例如下面的代码。

例:

```
<select name="name" iname="iname" value="value" ivalue="ivalue">
<option value="S">sina</option>
<option value="Y">yahoo</option>
</select>
```

相关属性:

multiple 这个布尔变量的值决定是否允许多重选择，值为 True 时 Select 控件允许复选，否则相反。

name 和 Value: 这一组变量的主要作用是获取于该 option 的返回值，value 提供 name 的缺省值。

iname 和 ivalue: 与上一组参数功能相似，不同的是 ivalue 返回有效 Option 的序列号。被选中的控件用它的序号表示，0 代表没有 option 被选中，假如第二个和第三个同时被选中就表示为 2;3。

title 作为标题参数提供给浏览器，但是不同的浏览器处理方式有所不同，有些浏览器直接显示选项内容不显示标题，有的浏览器显示标题，按选择键进入选择界面。

tabindex 提供给浏览器的控件序号参数。

2．选项控件（Option）

Option 只有包含在 Select 内才有意义，无法单独使用。

相关属性:

value: Option 的返回值，假如当前 Option 被选择，这个 Value 的值会被传送到 Select 元素的 Name 变量。

title :供浏览器显示的选项标题。

onpick:如果当前 Option 被点选，浏览器跳转到指定的 URL。如下例所示。

例:

```
<card>
    <p>Please choice your favourite Web.<br/>
    <select name="X">
      <option value="S">sina</option>
      <option value="Y">yahoo</option>
    </select>
    <p>
</card>
```

上例是一个基本的单选列表，选择的结果被赋值给 X，下面我们来学习前面所说的两组属性的用法。

例:

```
<card>
<p>Please choice all your favourite Web.<br/>
<select name="X" iname="I" ivalue="1;3" multiple="true">
    <option value="S">sina</option>
    <option value="Y">yahoo</option>
    <option value="N">netease</option>
</select>
<p>
</card>
```

这里是一个使用了 iname 和 ivalue 的多选列表，I 被预置为 1;3 。假如用户选择了 sina 和 yahoo，X 被赋值为 S;Y，I 被赋值为 1;2。假如用户不做任何选择，I 等于 1;3，X 内容为空。下面我们将学习 onpick 属性的功能，如下。

例:

```
<p>Jump to your favourite Web.<br/>
<select>
<option onpick="http://wap.sina.com.cn">sina</option>
<option onpick="http://wap.chnmobile.net">china mobile</option>
</select>
<p>
</card>
```

这里演示了 Option 的 onpick 功能，不管 Option 的状态如何，只要它被点选，浏览器

就会跳转到指定的 URL。

3．文本框控件（Input）

文本框控件（Input）用来输入文本，控件的使用格式如下：

<input name="variable" title="label" type="type" value="value" default="default" format="specifier" emptyok="false|true" size="n" maxlength="n" tabindex="n"/>，。

相关属性:

title，该输入框的标题。

type，默认值为 text，如选择 password，则输入的数据显示为*。

name，指定了用来存储该输入文本的变量名字。

value，与 select 的相同属性很相似，name 用于存储变量数据，value 用于提供缺省值。

format，用来格式化输入的数据，可用的标记如下，使用时可用“一位数字标记”和“*标记”的形式，前者代表 N 个标记型字符，如 3X，后者代表任意个(小于 maxlength 属性的值)标记型字符。format 属性所用到的标记性字符如表 5.5。

表 5.5　format 属性所用到的标记

标记	描述
A	任何符号或者大写字母(不包括数字)
A	任何符号或者小写字母(不包括数字)
X	任何符号、数字或者大写字母(不可改变为小写字母)
X	任何符号、数字或者小写字母(不可改变为大写字母)
N	任何数字(不包括符号或者字母)
M	任何符号、数字或者大写字母(可改变为小写字母)或者多个字符，默认为首字大写
M	任何符号、数字或者小写字母(可改变为大写字母)或者多个字符，默认为首字小写

maxlength 属性，指定了用户可输入的最大字符长度，最大限制为 256 个字符。

emptyok 属性，表示用户可否不填输入框，默认为 false，即要填。

size 属性，输入框显示长度，目前未被支持。

tabindex 属性，类似于在 HTML 表单中按 Tab 键后，焦点落在哪个选项上，该值决定了这个选择顺序，数字大的排在后面。目前未被支持。读者可以看下面一个简单的代码片断，考虑一下它的含义。

例:

```
<card>
<p>First name:
<input    type="text" name="first"/><br/>
Last name:
<input    type="text" name="last"/><br/>
Age:
<Input    type="text" name="age" format="3N"/>
</p>
```

```
</card>
```

4．计时器控件（Timer）

计时器控件 timer 可以用来在用户不进行任何操作的一段时间后，自动执行一个任务，任何激活 CARD 页面的任务和用户操作都会启动 timer，而任务进行时，timer 就停止。每个 CARD 只能有一个 timer，一个 timer 只能触发一个任务。语法如下：

```
<timer value="value"/>
```

其中 value 为必选属性，用来设置定时器的定时值，最小单位为 0.1 秒。

相关属性:

value: 倒计时的点数，每一单位等于 0.1 秒。具体用法如下面的代码片断所示。

例:

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
<card id="first" ontimer="#next">
    <timer value="100"/>
    <p>Wait ten seconds</p>
</card>
<card id="next">
<onevent type="timer">
    <go href="#first"/>
</onevent>
<timer value="10"/>
<p>Wait one second</p>
</card>
</wml>
```

事件

WML 的事件基本上分为两大类，一类是键盘(包括软硬按钮)输入事件，用<do>标签来处理，另一类是相关页面内部的事件，用<onevent>标签来处理。

1．外部事件

<do>标签的语法如下：

```
<do type="type" label="label" name="name" optional="false|true">任务</do>
```

任务就是以前提到的 4 种任务。<do>的属性中，type 是必选的，其他为可选。相关的属性说明如下：

label 属性，指定了软按钮在屏幕上的显示文本。目前 type 属性为 delete，help，prev 时该属性无效。

name 属性，为<do>取个名字，同一的 CARD 里的<do>不能重名。如果 CARD 级的<do>和 DECK 级的<do>同名，则覆盖 DECK 级的<do>。

optional 属性，指定手机是不是可以忽略这个事件，默认值是 false。

type 属性，指定触发的事件，常见的事件类型如表 5.6 所示。

表 5.6　事件类型

type 值	触发原因
accept	调用 ACCEPT 按钮机制
delete	调用 DELETE 按钮机制
help	调用 HELP 按钮机制
options	调用选择按钮机制
prev	调用 PREV 按钮机制
reset	调用清除和重新设定手机状态时的 RESET 机制(目前不支持)

通过分析下面的例子，相信读者会对这些有所了解。

程序清单 5.34(example11.wml)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
<head>
<meta http-equiv="Cache-Control" content="max-age=0"/>
</head>
<card id="card0" ordered="false">
<do type="accept" label="InputName" name="do1">
<go href="#card01"/>
</do>
<p>
NAME:<input name="userName" title="User Name" type="text" format="*M" emptyok="false"
maxlength="12"/>
</p>
</card>
<card id="card01">
<p>
You name is $(userName:noesc).
</p>
</card>
</wml>
```

2．内部事件

内部事件主要有<onevent>与<timer>两种，现在分别介绍如下。

（1）<onevent>的语法如下：

<onevent type="type">任务</onevent>

必选属性 type 的取值如表 5.7 所示。

北京希望电子出版社

表 5.7　<onevent>标签 type 属性的取值

type 值	如果用户执行了以下操作就执行任务
onpick	用户选择或不选一个<option>项时
Onenterforward	用户使用<go>任务到达一个 CARD 时
onenterbackward	用户使用<prev>任务返回到前面的 ARD 时，或者按 BACK 按钮时
ontimer	当<timer>过期时

具体的用法如程序清单 5.35 所示。

程序清单 5.35(example12.wml)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.com/DTD/wml_1.1.xml">
<wml>
<!-- this deck can't use in Ericsson r320sc ， because r320sc haven't accept button-->
<card id="start">
<do type="accept" label="next">
    <go href="#two"/>
</do>
<p>This is the first card.</p>
</card>
<card id="two">
<do type="accept" label="next">
    <go href="#three"/>
</do>
<onevent type="onenterbackward">
    <go href="#temp"/>
</onevent>
<p>This is the second card.</p>
</card>
<card id="three">
<do type="accept" label="back">
    <prev/>
</do>
<p>This is the thired card.</p>
</card>
<card id="temp">
<do type="accept" label="start">
    <go href="#first"/>
</do>
<p>haha，you are lost!</p>
</wml>
```

（2）<timer/>事件。<timer>事件可以用来在用户不进行任何操作的一段时间后，自动

执行一个任务，任何激活 CARD 页面的任务和用户操作都会启动<timer/>，而任务进行时，<timer/>就停止。每个 CARD 只能有一个<timer/>，一个<timer/>只能触发一个任务。语法如下：

<timer name="variable" value="value"/>

其中 name 为可选属性，指定为一个变量名，当退出该 CARD 时，该变量存储此时定时器的值，当定时器超时时，手机将该变量设为 0；value 为必选属性，用来设置定时器的定时值，最小单位为 0.1 秒。

程序清单 5.36(example13.wml)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
"http://www.wapforum.org/DTD/wml_1.1.xml">
<wml>
<head>
<meta http-equiv="Cache-Control" content="max-age=0"/>
</head>
<card id="card1" ontimer="#card2">
<timer name="time1" value="50"/>
<p align="center">
After 5s，goto card2
</p>
</card>
<card id="card2">
<onevent type="ontimer">
<go href="#card1"/>
</onevent>
<timer name="time2" value="50"/>
<p align="center">
Here is card2!
</p>
</card>
</wml>
```

需要注意的是，<onevent>　<timer>　<do>三者必须严格按以上顺序写，否则将会出错，因为和 XML 一样 WML 的格式要求也是相当严格的。

5.5.4　用 JSP 创建 WAP 应用

使用 ASP 或者 JSP 来创建动态 WML 内容，是非常容易的。唯一要注意的就是配置服务器使它的脚本输出类型为"text/vnd.wap.wml"，或者在 JSP 脚本中直接设置输出类型。

程序清单 5.37 是一个用 JSP 输出动态 WML 内容的例子。

程序清单 5.37(wap.jsp)

```
<?xml version="1.0"?>
<!DOCTYPE wml PUBLIC "-//WAPFORUM//DTD WML 1.1//EN"
```

北京希望电子出版社

```
"http://www.wapforum.org/DTD/wml_1.1.xml">
<% response.setContentType("text/vnd.wap.wml"); %>
<wml>
<card id="start">
<do type="accept">
<go href="index.jsp#test"/>
</do>
<p>用JSP输出动态WML:<br/>
选择Accept继续!<br/>
</p>
</card>
<card id="test">
<do type="prev">
<prev/>
</do>
<%
out.println("<p>");
out.println("你好，测试成功!这是使用JSP程序产生的输出<br/>");
out.println("</p>");
%>
</card>
</wml>
```

在上面的程序中，关键之处在于使用了 JSP Response 对象的 setContentType()方法设定输出文件类型为 text/vnd.wap.wml，这样尽管在服务端运行的是 JSP 程序，但是传送到无线上网设备中的确是标准的 WML 文件，自然可以被上网设备的浏览装置浏览。

5.6　本 章 小 结

这一章中，我们主要对 XML 技术做了初步的了解。还简要地介绍了 XHTML 的概况，WML 的语法知识，其中，后面两部分的内容不是十分重要，读者不需要花太多的时间去学习它们，至于 JSP 技术如何与 XHTML、WML 等技术结合起来，相信很多读者都十分关心这个问题，我们认为，只要掌握了 XML 技术与 JSP 技术是如何结合起来的，那么其他的问题就可以触类旁通、迎刃而解了，毕竟 XHTML、WML 都是 XML 的子集，在本质上与 XML 没有太大的区别。学完本章后，如果读者可以读懂 XML 文件，并且可以自己编写一些简单的 XML 文件，那么我们的目标就已经达到了。

在下一章，我们将介绍一个十分关键的问题，那就是 XML 技术如何与 JSP 技术结合起来。

第 6 章　JSP 与 XML 联合开发技术

在第 5 章中，我们已经介绍了 XML 的相关知识，在本章中，我们将进入正题，介绍 JSP 技术如何与 XML 技术结合起来，联合开发一个动态网站。本章中需要重点掌握的内容有：

- JSP 技术与 XML 技术结合的一些概念。
- taglib 操作指令的使用方法。
- Tag Library 的运行原理。
- tagext 包简介。
- 自定义 Tag Library。
- 应用自定义的 Tag Library 开发 JSP 程序。

下面我们就开始讨论 XML 技术与 JSP 技术如何结合起来。

注意：本章讨论的 JSP 技术与 XML 技术的结合，不涉及其他 J2EE 技术与 XML 技术的结合，尽管 JSP 技术也是 J2EE 核心技术的一部分，对 J2EE 技术与 XML 技术相互结合感兴趣的读者可以参考有关的专著。

6.1　XML 与 JSP 技术联合

6.1.1　XML 与 JSP 技术联合的模式

Java 2 平台企业版（J2EE）架构在厂商市场和开发者社区中备受推崇。J2EE 技术的核心是 Enterprise JavaBeans 技术和 JavaServer Pages 技术，以及它对 XML 语言的支持，正是源于这些核心技术，使 Java 技术计算扩展到了企业级应用领域。

作为建立在 Java Servlets 模型之上的表达层技术，JSP 技术使得产生动态网页变得更简单，它允许将静态 HTML 内容与服务器端脚本混合起来生成动态输出。JSP 把 Java 作为默认的脚本语言，同时 JSP 技术提供了大量的服务器端标签（Tag，例如<jsp:include>等操作指令），获得了一个健壮的 Web 应用平台和一种简单易用的语言工具。另一方面，作为一种开发工具，可扩展标记语言（XML）简化了数据交换、进程间消息交换这一类的事情，因而对开发者逐渐变得有吸引力，并开始流行起来。自然，在 J2EE 架构中利用 JSP 访问或集成 XML 解决方案的想法十分诱人。因为这将是强大系统架构，应用平台同高度灵活的数据管理方案的结合。

那么 XML 技术与 JSP 技术集成的方案都有哪些呢?总的来说，XML 技术与 JSP 技术结合有两个大的趋势，第一个大的趋势就是以 XML 技术为前端显示层或者是后端数据层，JSP/JMS/Servlet/EJB 等 J2EE 技术为中间处理层，JSP 等 J2EE 技术接受客户端的请求，从后端数据层中获取数据，经过加工处理之后，以 XML/XSL/XSLT 的形式返回客户端，实际上客户端的请求也需要通过 XML 显示层才传递到 JSP 技术所在的中间处理层，在这个模型中，JSP 技术充当了逻辑控制、计算处理的角色，而 XML 技术则充当了显示数据、存储

数据、传递信息流的功能。体现这一趋势的例子很多，具体如下所述。

XML 技术充当数据层

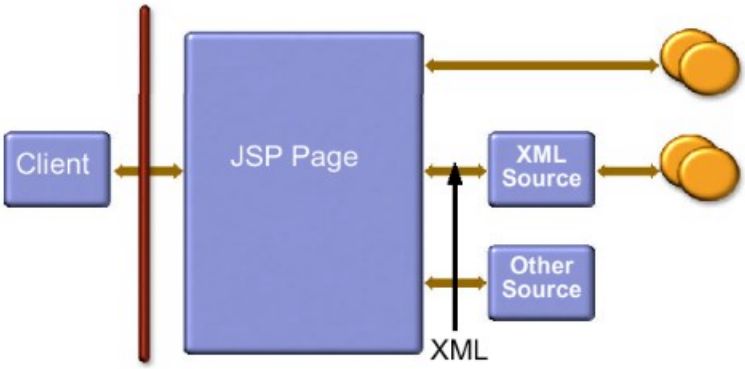


图 6.1　XML 技术充当数据层

如图 6.1 所示，这是一个最简单的 JSP 与 XML 技术集成的模型，在这里，JSP 程序从 XML 数据源或者是其他的数据源中提取数据，经过加工处理，然后返回客户端。

等一下!有的读者也许会问:为什么用 XML 文件存储数据吗?为什么不使用数据库呢？这个问题问得很好。我们的回答是，对很多用途来说，用数据库太过浪费了。要使用一个数据库，你必须安装和支持一个分离的服务器处理进程（a separate server process，通常指的是数据库服务进程，多个服务进程驻留于同一台服务器上，很容易产生效率问题），它常要求有安装和支持它的 Administrator(可以理解为管理界面)。你必须学习 SQL 语言，并用 SQL 写查询，然后获取记录集，再返回。而如果你用 XML 文件存储数据，将可减少额外的服务器的负荷，这个负荷通常十分巨大，以至于我们通常不得不采用所谓的分布式处理技术。还有，你还找到了一个编辑数据的简单方法。你只要使用文本编辑器，而不必使用复杂的数据库工具。XML 文件很容易备份，和朋友共享，或下载到你的客户端。同样的，你可以方便地通过 Ftp 工具上载新的数据文件(XML 文件)到你的站点。

XML 还有一个更抽象的优点,即作为层次型格式得数据比关系型格式的数据更好。它可以用一种很直接的方式来设计数据结构来符合你的需要。你不需要使用一个实体——关系编辑器，也不需要使你的图表（schema）标准化。 如果你有一个元素（element）包含了另一个元素，你可以直接在数据格式中表示它，而不需要使用表的关联。

注意：读者请注意，我们并非在排斥使用关系型数据库系统，我们仅仅是建议在不需要使用数据库系统的地方不要使用数据库系统，而是用 XML 技术存储数据和信息。但是在很多应用系统中，仅仅依靠文件系统是不够充分的。如果更新改写数据很频繁，文件系统会因为同时写入而受到破坏。数据库则通常支持事务处理，可以应付并行发生的请求而不至于损坏。对于复杂的查询统计要有反复、及时的更新，此时数据库表现都很优秀。当然，关系型数据库还有很多优点，包括丰富的查询语言、图表化工具、可伸缩性、存取控制等等，这是目前 XML 技术所不具备的。

注意：目前，XML 的结构化查询标准——XQL 语言已经出台了，这是一种类似于 SQL 的语言，可以实现对 XML 文档进行有效的查询。

在上面这样的案例中，正如大多数中小规模的、基于发布信息的站点一样，你可能涉及的大多数数据存取都是读，而不是写，数据虽然可能很大，但相对来说并没有经常的更新变化，你也不需要做很复杂的查询，即使你需要做，也将用一个独立的查询工具，那么成熟的 RDBMS(关系型数据库管理系统)的优点消失了，而面向对对象/层次型的数据模型的优点则可以得到体现。最后，如果你使用的数据库系统提供一个查询器外壳来进行 SQL 查询并将结果集转化进入 XML 文件，这也是完全有可能的。所以你可以选择这两种方式之一，使用 XML 或者使用 RDBMS 存储数据。事实上，XML 正变成一种非常健壮的，便于编程的工具，作为某个成熟的数据库的前端工具来进行存储和查询。Oracle 的 XSQL Servlet 即是这种技术的一个很好的例子。

XML 充当显示层

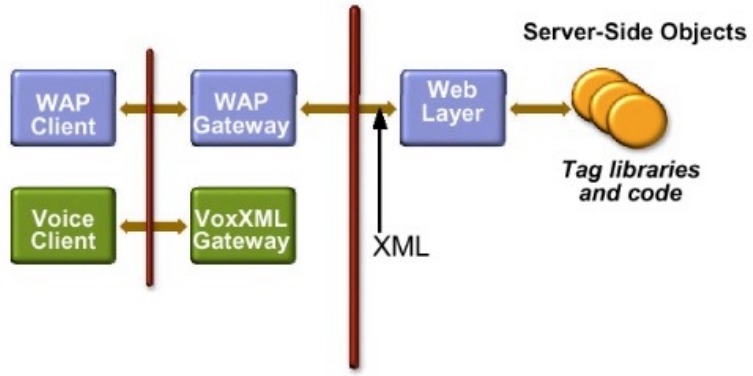


图 6.2　XML 充当显示层

如图 6.2 所示，由 XML 技术充当显示层，通过 WAP、VoxXML 网关为 WAP 客户端、Voice 客户端提供服务，其实，这里不仅仅限于 WAP 客户端和 Voice 客户端，还可以是其它的客户端，例如 HTTP 客户端或者是其他的无线设备客户端等等。Web Layer 由 JSP/Servlet 等技术构成，JSP/Servlet 程序把加工处理后的结果以 XML 的形式输出，通过特定的网关，发送到特定的客户端设备中去。例如在第 4 章的末尾，我们实现了这样的一个例子：以 JSP 程序输出一个动态的 WML 文件，而 WML 文件其实就是特殊的 XML 文件。

XML 技术做为显示层是有很大的优势的，因为 XML 技术把数据类型定义、数据与显示格式分开了，只要定义好了数据类型与若干个显示格式，就可以对不同的数据类型或者是不同的客户端套用不同的显示格式，甚至可以根据不同的客户端决定把那些数据发送到客户端，那些数据应该屏蔽起来。这种解决方法有着很好的伸缩性。在 XML 中，显示格式由 CSS 或者是 XSL 技术来实现。现在，又有了一种 XSLT 技术，是 XSL 技术的扩展，利用这种技术，我们可以把显示格式做成模板(Template)的形式，JSP 或者 Servlet 程序只需要把处理结果填充到模板中就可以了，这种方法的伸缩性更强，也更容易开发，如图 6.3 所示。

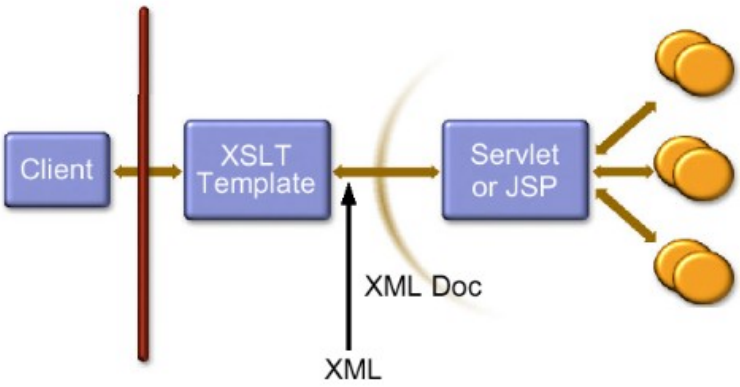


图 6.3　XML 充当显示层

JSP 技术与 XML 技术集成的另一个趋势就是所谓的 Tag Libraries 在 JSP 程序中的大规模应用。我们经常遇到的都是第一种趋势的例子，实际上，第二种集成方式对 JSP 技术的扩展更为巨大，功能也更为强大，影响也更为深远，毕竟，对于扩展处理层的计算能力，简化开发步骤，缩短开发时间的问题而言，如何显示结果，如何存储数据是些相对次要的问题。本章重点介绍的是 JSP 技术与 XML 技术的第二种集成方式，亦即关于 Tag Libraries 的内容。

什么是 Tag Libraries 呢?实际上 JSP 的操作指令，如<jsp:forward>、<jsp:include>等都是些封装好的 XML Tag，它们都属于 jsp 这个 Tag Library，这个 Tag Library 为 JSP 引擎所直接支持，关于这一点，在 JSP1.2 技术规范中有详细的说明，在 JSP1.2 规范中，还定义了一套 DTD，利用这一套 DTD，JSP 文件可以有两种编写方式，一种是普遍的 HTML 代码加上 JSP 代码段，另一种方式是全部以 XML 文件的格式编写，后者的结构性，层次性比第一种要好很多，不过第二种编写方式应用的还不是很多，但是这两种方式编写出来的文件都必须以 jsp 为扩展名，都需要经过 JSP 引擎的编译运行。JSP 页本身可被翻译为 XML 结构的文档。JSP 1.2 规范中定义了后者(XML 结构的文档)适用的标签和不适用标签可供选择。比如，JSP 代码段(写在<%...%>中)同样可以写在 XML 适用标签(指<jsp:sriptlet>和</jsp:scriptlet>)之间。显然，当你手工书写 JSP 代码时，前一种方法比编写 XML 格式的文档更容易。然而，在 IDE（集成开发环境）或者其他 JSP 开发工具中，使用 XML 书写格式能够更容易地生成有效的 JSP 程序。关于 JSP 和 XML 关系的详细资料，请参阅 JSP 1.2 规范，我们在这里就不详细论述了。

利用 JSP 内建的操作指令、编译指令，我们已经可以实现很多常用的功能，但是读者有没有想过对这些指令进行扩充、定制，编写一些新的指令，在 JSP 程序中使用，使得我们在实现一些更强大、更特殊的功能时更为方便快捷呢? JSP 规范支持这样的功能。我们可以定义一些新的 XML 标记，再编写一些 Java 类——标记执行句柄(Tag Handler)，以实现这些标记的功能，经过一番配置以后，就能够在 JSP 程序中直接运用这些标记完成相应的功能，而不必每次都写上同样的一大段代码。例如我们可以编写一个 mail 标记，把 Java Mail API 封装起来，这样我们在需要收发邮件的时候，只需要使用这个 mail 标记就可以了，而不必在每个 JSP 程序中都使用繁琐的 Java 代码访问邮件服务器，并收发邮件。我们还可以

把自己自定义的新标记分门别类，按照不同的类型分发打包为 jar 文件，部署到 JSP 服务器上去，这些 jar 文件就是所谓的 Tag Libraries 了。

本章的目的，就在于讨论如何开发 Tag Libraries 和如何在 JSP 程序中使用 Tag Libraries，扩展 JSP 技术的功能。至于 JSP 技术与 XML 技术集成的第一个趋势，由于涉及 XML、XSL、XSLT 的知识过多，与 XML 的关系十分密切，但是与 JSP 的关系不是十分密切，而且技术性不如第二种方法强，所以我们在本书中将不过多讨论这方面的问题，对此感兴趣的读者请参考相关的专著，本章下面的讨论都是以 JSP 技术与 XML 技术的第二种集成方式为讨论起点的。

6.1.2　XML 与 JSP 技术联合的优越性

XML 技术与 JSP 技术集成(按照第二种方式)在一起有何优越性呢?

便携性(Portable)

所谓的便携性，指的是 Tag Libraries 可以包装成 jar 文件，按照一定的步骤分发部署到可用的 JSP 服务器中去，这些 JSP 服务器可以是跨平台的不同的服务器，但是对于 Tag Libraries 的开发与使用没有任何的影响，这充分体现了 Java 技术跨平台的本质，这对于 Tag Libraries 开发者来说是一个福音。

简单性(Simple)

简单性是 Tag Libraries 的一大特性。我们都知道在 HTML 语法中，<I>与</I>标记对可以使得一段文本以斜体的形式显示，但是我们有谁知道这种效果是如何实现的呢?不，我们不需要知道这个问题，这个问题应该交给微软的程序员去解决，我们只需要知道<I>与</I>标记可以产生斜体的效果就可以了，这就是 HTML 的简单性。同样，我们如果获得了一个 mail Tag，我们知道利用它可以发送邮件，使用它的时候，只需要注明主机，邮件内容，收信人，主题就可以了，而不需要知道这个 mail Tag 是如何封装了 Java Mail API 的，这些细节问题由 Tag 的开发者来解决，Tag 使用者无需对此了解太多，Tag 使用者只需要专注于 Tag Libraries 的部署与商业逻辑的实现即可，这就是 Tag Libraries 的简单性，毕竟，需要用到邮件服务的 JSP 开发者很多，但是不是每个 JSP 程序员都熟悉 Java Mail API，这也没有必要，只要有人编写出 mail Tag，并且愿意无偿提供别人使用，那么我们就可以下载它，用到我们的 JSP 程序中。终有一日，我们可以像编写 HTML 文件一样，使用各种各样的 Tag 来编写 JSP 文件，到那时，JSP 程序中将不会再有晦涩难懂的大段 Java 脚本了，取而代之的是一个个简单明了，易于使用的 Tag 与 Tag 所包含的数据，这时候，开发 JSP 程序不再是专业程序员干的活了，一个普普通通的甚至对 Java 语言一无所知的人都可以利用 Tag 开发出功能强大的 JSP 程序来。

可扩展性

可扩展性也是 XML 技术与 JSP 技术集成(第二种方式)的重要特性。其实简单性与可扩展性的意义差不多，不过后者着重于功能方面，而前者着重于易用性方面。正是借助这种可扩展性，使得 JSP 技术在与 PHP、ASP 等技术的比较中，占了上风。什么是可扩展性呢?

我们都知道 JSP 中有一个<jsp:useBean>操作指令，可以在 JSP 程序中使用封装好 JavaBeans，功能十分强大，但是我们还不满足，我们希望在 JSP 程序中实现 EJB 的客户端，按照普通的做法，我们只能够循规蹈矩地写上一大段繁琐的 EJB 客户端代码，但是，我们也可以定义一个新的 useEJB Tag，这个标记实现了 EJB 的客户端，在使用 useEJB Tag 时，我们只需要指定 EJB 的 JNDI 服务名称就可以访问 EJB Server 中的 EJB 的商业方法了，而且我们一旦实现了 useEJB Tag，我们就可以在千千万万的 JSP 程序中使用它，而且代码没有太大的差异。如果 EJB 技术有了进步，比如 Sun 公司发布了 EJB 3.0 规范，EJB Server 中的 EJB 也按照 EJB 3.0 规范来编写了，这时客户端的 JSP 程序有可能全部陷于瘫痪，该怎么办呢?别着急，我们只需要改写一下实现 useEJB Tag 的 Java 类，重新分发部署 useEJB Tag 就可以了，所有使用了 useEJB Tag 的 JSP 程序的代码都不需要改变(只要 useEJB 标记的使用方法、属性、参数不发生改变)。这对于 JSP 程序开发者来说无疑是一个天大的喜讯。

多样性

所谓的多样性指的是 Tag Libraries 可以用于多种脚本语言中，这不是说 Tag Libraries 可以为 PHP、ASP 所用，Tag Libraries 仍需要在 JSP 框架下使用，当前的 JSP 规范仅仅支持以 Java 语言为 JSP 的脚本语言，但是有些服务器支持别的脚本语言，比如 JRun 支持以 JavaScript 为脚本语言，这种情况下，利用 JavaScript 编写的 JSP 程序与利用 Java 编写的 JSP 程序相比，一样能够使用 Tag Libraries，而且使用方法一样，功能没有任何差别。

上面所述的 4 个方面就是 JSP 技术与 XML 技术以第二种方式集成时最突出的优点，还有很多优点，读者不妨参考 JSP 技术规范。

6.1.3　XML 与 JSP 技术联合的前景

XML 技术与 JSP 技术集成在一起有着广阔的前景，第一种集成方法可以与第二种集成方法集成起来，届时，网络服务端的后台数据层是基于 XML 存储技术的数据层，而且基于 XML 存储技术的数据层的功能可以与基于 RDBMS 的数据层的功能并驾齐驱了。在中间处理层，JSP 程序全部实现了 Tag 化、JavaBeans 化，相关的功能都封装到 Tag Libraries 或者 JavaBeans 中，开发者只需要专注于商业逻辑的实现，而不必关心特定功能是如何实现的，JSP 程序的编写完全 XML 化了，程序的格式类似于 XML 文档，不但层次性好，可读性强，而且可以利用 IDE 以图形化的方式迅速地开发出 JSP 程序来，这正如今天我们利用 Dreamweaver 开发网页一样方便快捷。中间处理层的输出结果完全是 XML 文件的格式，在前端显示层，主要以 XML、XSLT、XSL 等技术为核心，前端显示层获取处理层的输出数据，并把它们与显示格式结合起来，填充到特定的显示模板中，再根据特定的客户端设备，往客户端发送不同的显示模板，例如，如果客户端是无线上网设备，那么显示层就通过网关往客户端发送 WML 文件，由客户端的处理程序解释。如果客户端是普通的 Web 浏览器，那么就往客户端发送 XML 文件(起数据承载功能)和 DTD 文件(起数据类型描述功能)，还有 XSL 文件(起格式化数据的功能)，浏览器根据这三个文件，就可以正确解析 XML 文档，并把处理结果显示出来。在如客户端是 Real Player，那么显示层就往客户端发送 SMIL 格式的文件，Real Player 可以解析这种格式的文件，这正如 IE 可以解析 XML 文件一样。这样一来，服务端显示层就可以根据不同的客户端，使用同样的处理结果，结合不同的显

示模板，服务于不同类形的客户端，而后台处理程序与数据存储模式却是一样的。客户端的设备也由此而具有学习功能，由于 XML 技术的开放性，客户端的处理程序不再需要支持特定的文件格式，例如 HTML 之类的标准，它只要读取不同的 DTD 文件，就能够识别解析不同的 XML 文件了，再嵌入 XSL 文件，格式化数据的显示效果，这样客户端的程序可以做得很小，再不会有庞大的浏览器了，浏览器之间也不会因为支持的标准不一样而产生不兼容的现象了，当然，这需要有一个前提，那就是所有客户端的设备都必须严格遵守 XML 的标准，这样执行结果才是无差异的。

综上所述，XML 技术与 JSP 技术一旦完全集成起来，那么不但服务端可以通过 XML 技术整合在一起，而且客户端与服务端之间也可以通过 XML 技术整合在一起了，真正实现了跨平台的梦想，这里所说的跨平台不只是对于某几种操作系统而言，而是对于不同的上网设备而言的，这时候打印机与手机、掌上电脑等设备通过 XML 技术相互通讯将不再是梦想了。另一方面，服务端又可以通过 JSP/EJB 等 J2EE 技术，构建一个高性能、支持并发性、支持事务的分布式处理系统，快速高效地对客户端的请求做出响应。这就是 XML 技术与 JSP 技术整合的前景。

6.2　在 JSP 中应用 XML

在本节中，我们将介绍 JSP 技术与 XML 技术整合的第二种模式的实现细节，它的核心就是 Tag Libraries 在 JSP 程序中的使用，那么在 JSP 程序中如何使用 Tag Libraries 呢?

6.2.1　taglib 编译指令

在 JSP 程序中，我们可以通过 taglib 编译指令在 J 中导入 Tag Libraries，关于 taglib 操作指令的知识，我们在本系列第一本书中已经提到过，为了本书的结构完整起见，我们把这部分的内容复述如下，供读者参考。

JSP 程序可用的标记/指令集可以通过使用“标记库”(Tag Library)的形式来进一步扩充。在 JSP 程序中，用 taglib 编译指令声明该程序中可以使用的标记库。用 URI 参数唯一地标识这个标记库，并指定一个相应的标记库识别符(prefix)用来辨别不同标记库标记的使用。如果一个 JSP 编译器不能定位一个标记库(即在 URI 位置处没有找到相应的标记库)，那么会引发一个致命的编译错误。如果在 taglib 指令之前使用了它所引入的标记库识别符，也会引发致命的编译错误。

标记库可以包含一个确认方法(Validation Method)，用这个方法可以确定 JSP 页面是否正确使用了该标记库的功能。taglib 指令的语法形式如下所示：

<%@ taglib uri="*tagLibraryURI*" prefix="*tagPrefix*" %>

其中的属性含义是

- uri：唯一的指定标记库的绝对 URI 或相对 URI。URI 用于定位这个标记库的位置。
- tagPrefix：指定所用的标记库的识别符，定义了<*prefix*:*tagname*>标记中的 *prefix*，用以区别用户所使用的不同的标记库。

例：

```
<leaPrefix:myTag>
```

注意：jsp，jspx，java，servlet，sun，和 sunw 等是保留的标记库识别符，用户不能定义这样的识别符。在目前的规范里，空识别符是不允许的。

下面的 JSP 代码片断演示了在 JSP 程序中如何使用自定义的标记库。

例：

```
<%@ taglib uri="leaApp/mytags" prefix="leaTags" />
...
<leaTags:playPingPong>
...
</leaTags:playPingPong>
```

在上面的代码中，taglib 编译指令引入了一个自定义标记库，该标记库的相对地址(URI)为 leaApp/mytags，该标记库的识别符是 leaTags。这个 JSP 程序不能够再使用这个识别符以引入其它的标记库。本例中我们假设此标记库中有 playPingPong 这个 Tag，并在 JSP 程序中使用它。

注意：如果 Java 编译器碰到一个形如 tagName 的标记，这个标记是由 taglib 编译指令引入的，但是 tagName 在相应的标记库里不存在，那么这也将导致致命的编译错误。假设在上面的例子中，leaApp/mytags 标记库里没有定义 playPingPong 这个标记，那么将发生致命的编译错误。

6.2.2　Tag Library 和 Tag 的原理

本小节将介绍 Tag Library 与 Tag 的工作原理，这个问题涉及到两大方面，第一个方面是 JSP 程序的执行原理，第二个方面就是 Tag 的执行原理。

我们首先介绍第一个方面——JSP 程序的运行原理，根据 JSP1.2 规范的描述，JSP 程序的运行可以分为这样的几个步骤。

Parsing

在这一步，JSP 引擎将检查 JSP 程序的源代码，看看是否有违反语法的地方，如果有，那么抛出错误信息，不运行此 JSP 程序，如果一切正常，那么就把这个 JSP 程序翻译为 XML 文件的格式，也就是把这个 JSP 程序按照 XML 文件的格式重新写一遍，我们在前面已经提到过，JSP 程序有两种书写方式，一种方式是普通的 HTML 标记加上 JSP 指令与 Java 脚本，第二种书写方式就是把 JSP 指令替换为特定的 XML 标记，这些标记在一个 DTD 文件中给出了定义，所谓的 Parsing 过程就是把按第一种方式书写的 JSP 程序翻译为以第二种方式书写的 JSP 程序，并加上一些 debug 信息，例如：

```
<%@ include file=”copyright.hmtl” %>
```

将会被替换为

```
<jsp:directive.include file=”copyright.html” />
```

在这一过程中，JSP 引擎还会处理 include 进来的文件，识别那些是由 JSP 指令转换过来的 XML 标记，那些是程序员自定义的标记。实际上，我们可以直接以第二种方式书写 JSP 程序，这样 Parsing 过程会被 JSP 引擎忽略掉，JSP 程序的运行速度将有所加快的。

Validation

这一步所作的工作就是验证 JSP 程序中所使用的自定义标记库，检查程序员是否正确使用了该标记库的功能。JSP 引擎会按照所使用的自定义标记库的出现顺序，逐个验证它们，查看每一个自定义 Tag 是否有对应的 TagExtraInfo 类(亦即下文的 TEI 类)，如果有，那么就调用相应的 TagExtraInfo 类的 isValid()方法，检查程序员是否正确使用了该标记的属性与参数。

Translation

在这一步中，JSP 程序(XML 格式的)被 JSP 引擎翻译为一个标准的 Java 类，该 Java 类一般继承了 HttpJspPage 接口，覆盖了_jspService()方法。该类编译执行的结果就是我们所看到的 JSP 程序的执行结果。请看程序清单 6.1。

程序清单 6.1(test.jsp)

```
<%
out.println("你好");
%>
```

程序清单 6.1 经过 JSP 引擎翻译后所产生的 Java 类文件如下所示(程序清单 6.2，有少量改动，主要是代码缩进方面)。

程序清单 6.2

```
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.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002ftest_0002ejsptest_jsp_1 extends HttpJspBase
{
    static
    {
    }
    public _0002ftest_0002ejsptest_jsp_1( )
    {
    }

    private static boolean _jspx_inited = false;
```

北京希望电子出版社

```
public final void _jspx_init() throws JasperException
{
}

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;
     try
     {

          if (_jspx_inited == false)
            {
               _jspx_init();
               _jspx_inited = true;
          }
          _jspxFactory = JspFactory.getDefaultFactory();
          response.setContentType("text/html;charset=8859_1");
          pageContext = _jspxFactory.getPageContext(this，request，response，
                    ""，true，8192，true);

          application = pageContext.getServletContext();
               config = pageContext.getServletConfig();
          session = pageContext.getSession();
          out = pageContext.getOut();

          //begin [file="E:\\temp\\Down\\Tomcat3.2\\webapps\\ROOT\\test.jsp";
               //from=(0，2);to=(2，0)]
           out.println("脛茫潞脙");
          // end
     }
     catch (Exception ex)
     {
          if (out.getBufferSize() != 0)
               out.clearBuffer();
          pageContext.handlePageException(ex);
```

北京希望电子出版社

```
            }
            finally
            {
                out.flush();
                _jspxFactory.releasePageContext(pageContext);
            }
        }
    }
```

注意：程序清单 6.1/6.2 所使用的 JSP 服务器为 Tomcat 3.2，操作系统平台为 Win Me，JDK 版本为 1.3。

根据程序清单 6.2，我们应该获得这样的信息，该 Java 类继承自 HttpJspPage 接口，HttpJspPage 接口又继承于 JspPage 接口，JspPage 接口继承于 Servlet 接口，Servlet 接口继承于 GenericServlet 类，可见，关于 JSP 程序被编译为一个 Servlet 类文件的说法还是有一定道理的。Servlet 程序几乎都继承了 HttpServelt 类，而 HttpServlet 类正是继承于 GenericServlet 类，所以 HttpJspPage 接口和 HttpServlet 还是有点血缘关系的。在程序清单 6.2 中，最重要的方法是_jspService()方法，该方法实现了 JSP 程序(程序清单 6.1 中的 JSP 程序)的所有功能，把“你好”输出到客户端。

Execution

这一步就是编译上面一步所产生的 Java 文件，产生 class 文件，并调用 Java 虚拟机执行它，结果首先存放在缓冲区中，程序执行完毕，再一并输出到客户端。

有的读者可能会问 JSP 程序的执行流程与 Tag 的工作原理又有何关系呢?我们的回答是：有很大的关系，请接着往下看，我们介绍这个问题的第二个方面——Tag 的工作原理。

要说 Tag 的工作原理，首先必须提到 Tag 的编写方法。Tag/Tag Library 的编写方法如下所述。

首先确立编写的目标

我们首先需要明确都需要实现那些目标，需要封装什么功能，Tag 的名字是什么，初步设计 Tag 的使用方法，Tag 是否需要参数，这个 Tag 都需要那些属性，需要用到哪个 Java API，开发平台是什么，这些问题我们都需要考虑得十分清楚。而且这些问题在考虑清楚以前不要轻易编写程序。

编写 TLD 文件

所谓的 TLD 文件，指的是 Tag Library Descriptor Files，中文意义为标记库描述符文件。这是一个 XML 格式的文件，它描述了 Tag Library 的详细信息，它的具体作用有以下几个方面。

（1）指定此 Tag Library 的版本号。

（2）指定 Tag Library 所遵守的 JSP 规范的版本号，至少是 1.1。

（3）指定此 Tag Library 的名字，例如 Application。

（4）指定此 Tag Library 的 URI，普通资源定位符。

（5）指定此 Tag Library 的验证类(validator class)。

（6）指定此 Tag Library 的事件监听类(listener)。

（7）一些注释信息。

对于特定的 Tag，TLD 文件给出了下面的信息。

（1）指定了 Tag 的名字。

（2）指定了 Tag Hander，即真正实现了该 Tag 功能的 Java 类。

（3）指定了 TEI 类，利用该类，可以获取 Tag 的附加信息，也就是获取 Tag 的某些属性值。

（4）指定了 Tag 标记体内的数据类型。亦即<Tag>标记与</Tag>标记之间内容的数据类型。

（5）一些注释信息。

（6）指定了 Tag 的属性信息。

编写 Tag Handler

当编写好 TLD 文件以后，我们就可以针对特定的 Tag 编写特定的 Tag Handler 了，真正实现 Tag 的功能，当然了，Tag Handler 的编写有一套规则，不是随便一个 Java 类都可以作为 Tag Handler 的，我们在下文还要详细讨论这个问题。

打包 Tag Library

我们为每一个 Tag 都编写好 Tag Handler 以后，就可以把所有的类文件与 TLD 文件打包到一个 jar 文件中。然后把它分发部署到 JSP 服务器上去，这需要对服务器的配置做一点小小的改动。

测试 Tag Library

在这一步中，我们需要编写一个 JSP 程序，使用该 Tag Library 的 Tag，看看是否实现了原来设计的功能。请看下面的程序清单 6.3，这是一个 Tomcat 3.2 服务器所自带的例子，有少量改动。

程序清单 6.3(foo.jsp)

```
<html>
<body>
<%@ taglib uri="http://java.apache.org/tomcat/examples-taglib" prefix="eg" %>
Radio stations that rock:
<ul>
<eg:foo att1="98.5" att2="92.3" att3="107.7">
<li><%= member %></li>
</eg:foo>
</ul>
<eg:log>
Did you see me on the stderr window?
```

```
</eg:log>
<eg:log toBrowser="true">
Did you see me on the browser window as well?
</eg:log>
</body>
</html>
```

在程序清单 6.3 中，首先使用 taglib 操作指令导入一个 Tag Library——simple，指定了该 Tag Library 的 URI 地址与标识符(prefix=eg)，在下面的代码段中，则分别验证了 foo 标记与 log 标记的使用，foo 标记与 log 标记都是属于 simple Tag Library 的标记。

到此为止，一个 Tag Library 就算开发成功了，我们除了可以自己使用这个 Tag Library，也可以把它上载到网络中，供别人使用。

现在，我们可以对 Tag 的工作原理进行讨论了。我们参考上文所介绍的 JSP 程序的执行流程，看看在每一步中都发生了哪些事情。

Parsing

在这一过程中，如果 JSP 程序用到了 Tag Library，那么 JSP 引擎会读入该 Tag Library 的 TLD 文档，对其进行分析，以便把标准的 XML 标记(指的是由 JSP 指令转化而来的 XML 标记)与自定义的 Tag 识别开来。

Validation

在这一过程中，JSP 引擎会根据 TLD 文件的描述信息，对 Tag Library 进行验证工作。如果有某些标记指定了 TEI class，那么 JSP 引擎会利用这些类的 isValid()方法对 Tag 的属性值进行检验。

Translation

这一步是十分关键的一步，JSP 引擎会根据 taglib 编译指令指定的 Tag Library 的 URI 地址，把含有 Tag Handler 的 jar 文件载入，把 JSP 文件中使用了 Tag(指自定义的 Tag)的地方用 Tag Handler 所定义的方法代替，例如 doStartTag()方法、doAfterBody()方法等。然后 JSP 引擎再把整个 JSP 程序翻译为标准的 JSP 扩展类，亦即一个 Java 类。

Execution

在这一步，JSP 引擎调用 Java 虚拟机对翻译好的 JSP 程序文件进行编译运行，无论 JSP 程序中有没有使用 Tag Library，这一步都是一样的，没有什么区别。

读者都应该明白了吧，下面我们讨论一些细节性的问题，即应该如何开发一个 Tag Library，并把它应用到 JSP 程序的开发中去。

注意：本章所有开发的例子的开发环境：操作系统为 Windows Me 中文版，JDK 为 JDK 1.3 SE，JSP 服务器为 Tomcat 3.2 服务器，安装路径为 E:\temp\Down\Tomcat3.2，下文中一律以 TOMCATHOME 代替。

6.2.3　编写 TLD

在定好了开发目标以后，第一件要做的工作就是开发 TLD 文件。在上文已经提到过，TLD 文件的作用就是描述 Tag Library 和 Tags 的详细信息，TLD 文件在 Tag Library 中的地位十分重要，一个 Tag Library 的成败几乎全部取决于 TLD 文件的成败，而且如果出现错误多半是因为 TLD 文件编写不规范造成的，因为我们编写 Tag Handler 时必须参照 TLD 文件的定义。

TLD 文件本质上是一个 XML 文件，该文件可以使用的标记在另外的一个 DTD 文件中有定义，这个 DTD 文件的名字为 web-jsptaglibrary_1_1.dtd，读者可以在下面的网址中找到这个文件：

http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd

读者如果熟悉这个 DTD 文件，对于正确编写一个 TLD 文件是十分有好处的，下面我们就把这个 DTD 文件列出来，并结合具体的 TLD 文档，解释每个标记的含义，并介绍如何编写一个 TLD 文件。

程序清单 6.4(web-jsptaglibrary_1_1.dtd:有部分删节)

```
<!--
This is the DTD defining the JavaServer Pages 1.1 Tag Library
descriptor (.tld) (XML) file format/syntax.
A Tag Library is a JAR file containing a valid instance of a Tag Library
Descriptor (taglib.tld) file in the META-INF subdirectory，along with the
appropriate implementing classes，and other resources required to
implement the tags defined therein.
-->
<!--
The taglib tag is the document root，it defines:
tlibversion     the version of the tag library implementation
jspversion      the version of JSP the tag library depends upon
shortname       a simple default short name that could be used by
                a JSP authoring tool to create names with a mnemonic
                value; for example，the it may be used as the prefered
                prefix value in taglib directives
uri             a uri uniquely identifying this taglib
info            a simple string describing the "use" of this taglib，
                should be user discernable
-->
<!ELEMENT taglib (tlibversion，jspversion?，shortname，uri?，info?，tag+) >
<!ATTLIST taglib id ID #IMPLIED xmlns CDATA #FIXED
      "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<!--
Describes this version (number) of the taglibrary (dewey decimal)
#PCDATA ::= [0-9]*{ "."[0-9] }0..3
-->
```

```
<!ELEMENT tlibversion     (#PCDATA) >
<!--
Describes the JSP version (number) this taglibrary requires in
order to function (dewey decimal)
The default is 1.1
#PCDATA ::= [0-9]*{ "."[0-9] }0..3
-->
<!ELEMENT jspversion    (#PCDATA) >
<!--
Defines a short (default) shortname to be used for tags and
variable names used/created by this tag library.    Do not use
white space，and do not start with digits or underscore.
#PCDATA ::= NMTOKEN
-->
<!ELEMENT shortname         (#PCDATA) >
<!--
Defines a public URI that uniquely identifies this version of
the taglibrary Leave it empty if it does not apply.
-->
<!ELEMENT uri          (#PCDATA) >
<!--
Defines an arbitrary text string descirbing the tag library
-->
<!ELEMENT info      (#PCDATA) >
<!--
The tag defines a unique tag in this tag library，defining:
- the unique tag/element name
- the subclass of javax.servlet.jsp.tagext.Tag implementation class
- an optional subclass of javax.servlet.jsp.tagext.TagExtraInfo
- the body content type (hint)
- optional tag-specific information
- any attributes
-->
<!ELEMENT tag (name，tagclass，teiclass?，bodycontent?，info?，attribute*) >
<!--
Defines the subclass of javax.serlvet.jsp.tagext.Tag that implements
the request time semantics for this tag. (required)
#PCDATA ::= fully qualified Java class name
-->
<!ELEMENT tagclass (#PCDATA) >
<!--
Defines the subclass of javax.servlet.jsp.tagext.TagExtraInfo for
this tag. (optional)
```

北京希望电子出版社

```
If this is not given，the class is not consulted at translation time.
#PCDATA ::= fully qualified Java class name
-->
<!ELEMENT teiclass (#PCDATA) >
<!--
Provides a hint as to the content of the body of this tag. Primarily
intended for use by page composition tools.
There are currently three values specified:
tagdependent      The body of the tag is interpreted by the tag
        implementation itself，and is most likely in a
        different "language"，e.g embedded SQL statements.
JSP               The body of the tag contains nested JSP syntax
empty             The body must be empty
The default (if not defined) is JSP
#PCDATA ::=    tagdependent | JSP | empty
-->
<!ELEMENT bodycontent (#PCDATA) >
<!--
The attribute tag defines an attribute for the nesting tag
An attribute definition is composed of:
- the attributes name (required)
- if the attribute is required or optional (optional)
- if the attributes value may be dynamically calculated at runtime
        by a scriptlet expression (optional)
-->
<!ELEMENT attribute (name，required? ，rtexprvalue?) >
<!--
Defines the canonical name of a tag or attribute being defined
#PCDATA ::= NMTOKEN
-->
<!ELEMENT name     (#PCDATA) >
<!--
Defines if the nesting attribute is required or optional.
#PCDATA ::= true | false | yes | no
If not present then the default is "false"，i.e the attribute
is optional.
-->
<!ELEMENT required       (#PCDATA) >
<!--
Defines if the nesting attribute can have scriptlet expressions as
a value，i.e the value of the attribute may be dynamically calculated
at request time，as opposed to a static value determined at translation
time.
```

北京希望电子出版社

```
#PCDATA ::= true | false | yes | no
If not present then the default is "false"，i.e the attribute
has a static value
-->
<!ELEMENT rtexprvalue (#PCDATA) >
<!ATTLIST tlibversion id ID #IMPLIED>
<!ATTLIST jspversion id ID #IMPLIED>
<!ATTLIST shortname id ID #IMPLIED>
<!ATTLIST uri id ID #IMPLIED>
<!ATTLIST info id ID #IMPLIED>
<!ATTLIST tag id ID #IMPLIED>
<!ATTLIST tagclass id ID #IMPLIED>
<!ATTLIST teiclass id ID #IMPLIED>
<!ATTLIST bodycontent id ID #IMPLIED>
<!ATTLIST attribute id ID #IMPLIED>
<!ATTLIST name id ID #IMPLIED>
<!ATTLIST required id ID #IMPLIED>
<!ATTLIST rtexprvalue id ID #IMPLIED>
```

在程序清单 6.4 所列出来的 DTD 文件中，定义了所有在 TLD 文件中可用的 XML 标记，在这个 DTD 文档中，有很多英文注释，这些注释解释了每个标记的用途。看不懂!不要紧，下面我们就来结合一个具体的 TLD 文件实例，逐个解释这些标记的意义与用途。

考虑到读者都是 Tag Library 技术的初学者，要从头编写一个 TLD 文件有很大的难度，我们就以一个现成的例子为基础。本章所选用的 JSP 服务器为 Tomcat 3.2，Tomcat 3.2 服务器自带了一个使用 Tag Library 的例子，我们就以这个例子为基础，再进行扩展，这样不但见效快，降低了难度，读者也更容易理解些，这个例子的 JSP 文件如程序清单 6.3 所示，为了正确运行这个 JSP 程序(foo.jsp)我们需要把 TOMCATHOME\webapps\examples\WEB-INF 文件夹全部替代 TOMCATHOME\webapps\ROOT\WEB-INF 文件夹，并且把程序清单 6.3（foo.jsp）保存 ROOT 文件夹的根目录下面，这样就可以正确运行这个 JSP 程序了，运行效果如图 6.4 所示。

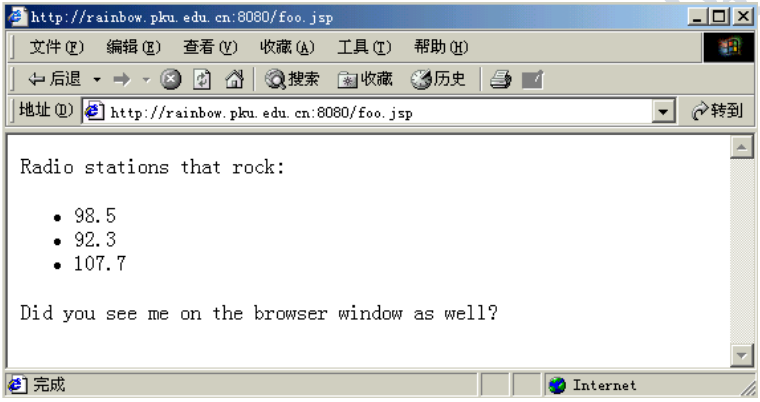


图 6.4　foo.jsp

我们在上文已经提到过，程序清单 6.3 所使用的 Tag Library 为 simple，先看看这个 Tag Library 的 TLD 文件，使用记事本打开 TOMCATHOME\webapps\ROOT\WEB-INF\jsp\example-taglib.tld 文件，如程序清单 6.5。

程序清单 6.5(example-taglib.tld)

```
<?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">
<!-- a tag library descriptor -->

<taglib>
<!-- after this the default space is
"http://java.sun.com/j2ee/dtds/jsptaglibrary_1_2.dtd"
-->

<tlibversion>1.0</tlibversion>
<jspversion>1.1</jspversion>
<shortname>simple</shortname>
<uri></uri>
<info>
    A simple tab library for the examples
</info>

<tag>
    <name>ShowSource</name>
    <tagclass>examples.ShowSource</tagclass>
    <info> Display JSP sources </info>
    <attribute>
        <name>jspFile</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>

<!-- A simple Tag -->
<!-- foo tag -->
<tag>
    <name>foo</name>
    <tagclass>examples.FooTag</tagclass>
    <teiclass>examples.FooTagExtraInfo</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>
        Perform a server side action; uses 3 mandatory attributes
    </info>
```

北京邮电大学出版社

```
        <attribute>
            <name>att1</name>
            <required>true</required>
        </attribute>
        <attribute>
            <name>att2</name>
            <required>true</required>
        </attribute>
        <attribute>
            <name>att3</name>
            <required>true</required>
        </attribute>
    </tag>

    <!-- Another simple tag -->
    <!-- log tag -->
    <tag>
        <name>log</name>
        <tagclass>examples.LogTag</tagclass>
        <bodycontent>TAGDEPENDENT</bodycontent>
        <info>
            Perform a server side action; Log the message.
        </info>
        <attribute>
            <name>toBrowser</name>
            <required>false</required>
        </attribute>
    </tag>

    </taglib>
```

下面我们就来详细分析程序清单 6.5，粗粗看来，程序清单 6.5 可以分为三大部分，第一部分就是前面四行，这是所谓的 XML 指令了，这一部分必不可少，但是和本章的关系不密切，在这里我们不用讨论它。第二部分描述了 Tag Library 的详细信息。这部分的代码如下：

```
<tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>simple</shortname>
    <uri></uri>
    <info>
        A simple tab library for the examples
</info>
```

读者应该对上面的代码所使用的标记有印象吧，不错，它们都在程序清单 6.4 中定义

了，下面我们就来讨论这几个标记的意义。

（1）tlibversion：指定此 Tag Library 的版本号，这里的值为 1.0。

（2）jspversion：指定此 Tag Library 所遵守的 JSP 规范的版本号，可以是 1.1 或者 1.2。

（3）shortname：指定此 Tag Library 的名字，此处的值为 simple。

（4）uri：指定了 Tag Library 的 URI，即到哪里可以获得此 Tag Library。

（5）info：对于 Tag Library 的一些注释。

还有一些标记，在程序清单 6.5 中没有出现，我们也一并解释如下：

（6）validatorclass：指定了 Tag Library 的 TagLibraryValidator class，TagLibraryValidator class 用于校检 Tag Library。

（7）listener：指定了 Tag Library 的事件监听者类，利用事件监听者类我们可以实现基于 Tag Library 的事件模型。

（8）small-icon：指定了代表 Tag Library 的小图标，如果 Tag Library 被某个开发工具封装起来，那么指定的图标可以显示在开发工具的工具栏中。

（9）large-icon：指定了代表 Tag Library 的大图标，其余的意义同上。

程序清单的第三部分是关于 Tag 的部分，这一部分详细地描述了所有 Tag 的信息。Simple Tag Library 包含有下面几个 Tag：foo Tag、log Tag、ShowSource Tag，我们来看看描述 ShowSource Tag 的代码段。

```
<tag>
    <name>ShowSource</name>
    <tagclass>examples.ShowSource</tagclass>
    <info> Display JSP sources </info>
    <attribute>
            <name>jspFile</name>
            <required>true</required>
            <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
```

由上面的代码读者不难看出，描述 Tag 的代码段也可以分为两大部份，第一部分描述 Tag 的基本信息，第二部分描述 Tag 的属性信息。整个代码段必须包含在<tag>标记与</tag>标记之间，<attribute>标记与<attribute>标记之间包含的就是 Tag 属性的信息，也就是第二部分，剩下的就是第一部分了。下面我们来分析一下这些标记的含义。

（1）name：指定 Tag 的名称，这里的值为 ShowSource，这说明我们在 JSP 程序中，如果希望使用 ShowSource 标记，必须用这样的语法。<prefix:ShowSource jspFile=”…”>，其中 prefix 指的是标记库标识符。

（2）tagclass：指定此 Tag 的 Tag Handler，即实现这个 Tag 功能的 Java 类，在此处的值为 examples.ShowSource，即 examples 包的 ShowSource 类就是 ShowSource Tag 的 Tag Handler。

（3）info：可以写一些对于本 Tag 的注释。

还有其他的标记，但是在上面的代码中没有用到，我们一并解释如下。

（4）teiclass：指定此 Tag 的 TagExtraInfo 类，tei 即是 Tag Extra Info 三个打头字母的

缩写。利用 TagExtraInfo 类(基类为 javax.servlet.jsp.tagext.TagExtraInfo，实际中必须覆盖这个类)，可以获取一些关于 Tag 的额外信息。

（5）bodycontent：指定在 Tag 对之间可以包含的内容的类型，比如在<prefix:ShowSource>标记与</prefix:ShowSource>标记之间可以包含哪些类型的内容？这是由 bodycontent 决定的。bodycontent 可以取以下值： tagdependent、JSP、empty。

tagdependent：标记体内的内容将被送到 Tag Handler 中，赋给 Tag，标记体内的内容可以为空。

JSP：标记体内的内容可以是 JSP 代码段，也可以是普通的文本，用于往客户端输出，也可以是空值。

empty：标记体内不能含有任何内容。

读者一定要注意这三个值的大小写。

（6）small-icon：代表该 Tag 的小图标，参见上文。

（7）large-icon：代表该 Tag 的大图标，参见上文。

我们再来看看描述 Tag 属性信息的那一部分代码，看看都可以用哪些标记，这些标记如下所示。

（8）attribute：<attribute>标记与</attribute>标记之间描述 Tag 的属性信息，而且一个 attribute 标记对之间只能够描述一个属性的信息。一个 Tag 可以含有多个属性，也就是可以含有多个 attribute 标记对。

（9）name：指定 Tag 某个属性的名字，在本例中，该值为 jspFile，亦即 ShowSource 标记有一个名为 jspFile 的属性。

（10）require：指定 Tag 某个属性是否必须指定的，可以使用的值为 false、true、yes、no。缺省值为 false。在本例中，jspFile 是必须指定的属性。

（11）rtexprvalue：指定 Tag 某个属性的值是否可以通过 JSP 代码段动态生成。可以使用的值为 false、true、yes、no。缺省值为 false。在本例中，jspFile 的值可以由 JSP 代码动态生成。

（12）type：指定 Tag 某个属性的 Java 类型。用法可能如下。

例：

```
<type>
java.lang.Object
</type>
```

缺省值为 java.lang.String。

读者都看明白程序清单 6.4 和 6.5 了吗?好的，现在我们该对 simple Tag Library 进行扩充了，按照通俗的做法，我们编写一个 hello Tag，这个 Tag 的唯一功能就是输出问候语：Hello World。TLD 文件该怎么写呢?

我们可以在程序清单 6.5 的后面加上这样的一段代码(必须在</tag>标记与</taglib>标记之间)：

```
<tag>
    <name>hello</name>
    <tagclass>examples.hello</tagclass>
```

```
    <bodycontent>JSP</bodycontent>
    <info>say hello world</info>
  </tag>
```

在上面的这段代码中，我们定义了 hello Tag，这个 Tag 的 Tag Hander 为 examples 的 hello 类。hello 标记对之间可以包含 JSP 代码段用于输出。

好了，simple Tag Library 的 TLD 文件已经编写好了。我们把这个文件保存在下面的目录：

TOMCATHOME\webapps\ROOT\WEB-INF\jsp\example-taglib.tld

下面我们该编写 hello Tag 的 Tag Handler 了。

6.2.4　编写 Tag Hander

在这一节中，我们将介绍如何编写 Tag Handler。本小节以 hello Tag 为例，编写一个 hello.java(这是由 TLD 文件决定的，不可能是其他的值)。hello.java 的源代码如程序清单 6.6。

程序清单 6.6

```
//File Name:hello.java
//Author:fancy
//Date:2001.6.3
//Note:Tag Handler for hello tag
package examples;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class    hello extends BodyTagSupport
{
    public final int doEndTag() throws JspException
    {
        try
        {
            pageContext.getOut().write("hello world");
        }
        catch(Exception e)
        {
            throw new JspException("IO Error: " + e.getMessage());
        }
        return EVAL_PAGE;
    }
}
```

在程序清单 6.6 中，定义了 hello 类，程序首先导入 Java Servlet API 的所有包和 java.util

包，hello 类继承自 BodyTagSupport 类，这是必需的，Tag Handler 可以扩展 Tag 接口或者 BodyTag 接口，还可以继承 TagSupport 类或者 BodyTagSupport 类，两者必居其一。

在 hello 类中，仅仅定义了 doEndTag()方法，doEndTag()方法在 BodyTagSupport 类中定义。在这里把父类的方法覆盖掉了。除了 doEndTag()方法以外，BodyTagSupport 类还定义了 doStartBody()方法、doInitBody()方法、doAfterBody()方法，这些方法在 Tag 的生命周期中依次执行，doEndTag()方法最后执行，在下文我们还将详细地讨论这个问题。

此 doEndTag()方法的返回值为 EVAL_PAGE，这是一个整型常数，其意义是：使得 JSP 引擎继续执行并输出该 Tag 后面的内容，如果返回值为 SKIP_PAGE，那么 JSP 引擎将会忽略该 Tag 后面的内容。类似的常数还有 EVAL_BODY_INCLUDE、EVAL_PAGE、EVAL_BODY_TAG 等，我们在下文还要谈到这个问题。

为了使读者增加对 Tag Handler 的理解，我们再来看看 foo Tag 的 Tag Handler，请看程序清单 6.7。

程序清单 6.7

```
package examples;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.util.Hashtable;
import java.io.Writer;
import java.io.IOException;

/**
* Example1: the simplest tag
* Collect attributes and call into some actions
*
* <foo att1="..." att2="...." att3="...." />
*/

public class FooTag extends ExampleTagBase    implements BodyTag
{
      private String atts[] = new String[3];
      int i = 0;

      private final void setAtt(int index，String value)
      {
            atts[index] = value;
      }

      public void setAtt1(String value)
      {
            setAtt(0，value);
      }

      public void setAtt2(String value)
```

北京希望电子出版社

```
    {
        setAtt(1，value);
    }

    public void setAtt3(String value)
    {
        setAtt(2，value);
    }

    /**
    * Process start tag
    *
    * @return EVAL_BODY_INCLUDE
    */
    public int doStartTag() throws JspException
    {
        return EVAL_BODY_TAG;
    }

    public void doInitBody() throws JspException
    {
        pageContext.setAttribute("member"，atts[i]);
        i++;
    }

    public int doAfterBody() throws JspException
    {
        try
        {
            if (i == 3)
            {
                bodyOut.writeOut(bodyOut.getEnclosingWriter());
                return SKIP_BODY;
            }
            else
                pageContext.setAttribute("member"，atts[i]);
            i++;
            return EVAL_BODY_TAG;
        }
        catch (IOException ex)
        {
            throw new JspTagException(ex.toString());
        }
    }
}
```

我们来分析一下 FooTag.java 程序的结构，FooTag 类继承自 ExampleTagBase 类，扩展

了 BodyTag 接口，其中 ExampleTagBase 类是自定义的类，请参考程序清单 6.8。FooTag.java 定义了下面几个方法 setAtt()方法、setAtt1()方法、setAtt2()方法、setAtt3()方法，这 4 个方法最先被调用，设定 foo Tag 的 3 个属性 att1、att2、att3。还有 doStartTag()方法，这个方法紧接着被调用。这个 doStartTag()方法没有做任何事情，仅仅返回一个整型常量：EVAL_BODY_TAG，其意义是使得 JSP 引擎继续执行这个标记并输出执行结果，doStartTag()方法还可能返回整型常量 SKIP_BODY，那么 JSP 引擎将会忽略这个 Tag 的执行，也就是不再继续执行这个 Tag，没有任何结果输出。doStartTag()方法紧接着 setXXX()方法被调用。

还有 doInitBody()方法，这个方法紧接着 doStartTag()方法执行。在本例中，doInitBody()方法的作用是把数组 atts[]的第一个值赋给了 JSP 程序变量 member，member 的生命周期为 Page。

还有 doAfterBody()方法，这个方法紧接着 doInitBody()方法执行。这是一个自调用式的方法，当它的返回值为 SKIP_BODY 时，JSP 引擎将跳出对这个 Tag 的执行，执行 Tag 后面的内容。如果 doAfterBody()方法的返回值为 EVAL_BODY_TAG，那么 JSP 引擎会自动调用 doAfterBody()方法，直至返回值不是 EVAL_BODY_TAG 时，JSP 引擎才会跳出这个执行循环。在本例中，doAfterBody()方法依次把数组 atts[]的每一个值赋给了 JSP 程序变量 member，如果全部赋值完毕，那么返回 SKIP_BODY 值，如果还没有赋值完毕，返回 EVAL_BODY_TAG 值，计数加一，再次执行 doAfterBody()方法。从执行 doAfterBody()方法的循环中跳出以后，JSP 引擎接下来将执行 doEndTag()方法，因为 FooTag.java 程序中没有定义 doEndTag()方法，JSP 引擎会调用父类的 doEndTag()方法。doEndTag()方法的返回值也是一个整型常数，有可能为 SKIP_PAGE，如果是这样的话，那么 Tag 后面的所有内容都将被 JSP 引擎自动跳过，doEndTag()方法的返回值也有可能为 EVAL_PAGE，如果是这样的话，那么 JSP 引擎会继续执行该 Tag 后面的值。执行完 doEndTag()方法后，会调用 release()方法，释放系统资源。这样，一个 Tag 就算执行完毕，如果 doStartTag()方法或者 doAfterBody()方法的返回值为 SKIP_BODY，那么这两个方法的后面各步都没有机会被执行，JSP 引擎自动把它们跳过了，例如当 doStartTag()方法的返回值为 SKIP_BODY 时，那么 doInitBody()方法、doAfterBody()方法方法都不会被执行，而 release()方法、doEndTag()方法无论在什么时候都将被执行。这就是 Tag 的执行流程。

上面的解释十分重要，读者一定要仔细体会，我们在下文中还会反复提及 Tag 的执行流程，并对它进行扩充。

下面请看程序清单 6.8(ExampleTagBase.java)。

程序清单 6.8(ExampleTagBase.java)

```
package examples;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public abstract class ExampleTagBase implements Tag
{
```

```
public void setParent(Tag parent)
{
    this.parent = parent;
}

public void setBodyContent(BodyContent bodyOut)
{
    this.bodyOut = bodyOut;
}

public void setPageContext(PageContext pageContext)
{
    this.pageContext = pageContext;
}

public Tag getParent()
{
    return this.parent;
}

public int doStartTag() throws JspException
{
    return SKIP_BODY;
}

public int doEndTag() throws JspException
{
    return EVAL_PAGE;
}


// Default implementations for BodyTag methods as well
// just in case a tag decides to implement BodyTag.
public void doInitBody() throws JspException
{
}

public int doAfterBody() throws JspException
{
    return SKIP_BODY;
}

public void release()
```

北京希望电子出版社

```
    {
        bodyOut = null;
        pageContext = null;
        parent = null;
    }

    protected BodyContent bodyOut;
    protected PageContext pageContext;
    protected Tag parent;
}
```

ExampleTagBase 类是 FooTag 类的父类，它定义了 doStartTag()方法、doInitBody()方法、doAfterBody()方法、doEndTag()方法、release()方法，如果 FooTag 类中没有相应的方法，那么 JSP 引擎会执行父类的方法，亦即 ExampleTagBase 类的相应方法，如 FooTag 类中没有定义 release()方法，实际运行中，JSP 程序调用的是 ExampleTagBase 类的 release()方法。

6.2.5　自定义 Tag Library 的应用

在这一节中，我们将介绍如何在 JSP 程序应用在上文中所开发的 simple Tag Library，我们还将测试新开发的 hello Tag。

步骤如下。

（1）把程序清单 6.6(hello.java)保存到下面的目录：

TOMCATHOME\webapps\ROOT\WEB-INF\classes\examples\hello.java

（2）编译程序清单 6.6(hello.java)。在 MS-DOS 窗口，输入下面的命令。

```
cd e:\temp\down\tomcat3.2\webapps\root\web-inf\classes\examples
SET CLASSPATH=%CLASSPATH%;e:\temp\Down\Tomcat3.2\lib\servlet.jar
javac hello.java
```

（3）检查 TLD 文件。检查 TOMCATHOME\webapps\ROOT\WEB-INF\jsp\example-taglib.tld 文件，看看是否已经添加上对 helloTag 的定义。

（4）编辑 web.xml 文件。编辑 TOMCATHOME\webapps\ROOT\WEB-INF\web.xml 文件，添加如下的代码段：

```
<taglib>
    <taglib-uri>
        http://java.apache.org/tomcat/examples-taglib
    </taglib-uri>
    <taglib-location>
        /WEB-INF/jsp/example-taglib.tld
    </taglib-location>
</taglib>
```

如果这段代码已经存在了，那么不需要对 web.xml 文件进行任何改动。保存好 web.xml 文件。

注意：在上文中我们提到过把 TOMCATHOME\webapps\examples\WEB-INF 文件夹全部

替代 TOMCATHOME\webapps\ROOT\WEB-INF 文件夹，因此 ROOT\WEB-INF 目录下面的 web.xml 文件可能不需要改动。

（5）改写程序清单 6.3(foo.jsp)。改写程序清单 6.3(foo.jsp)，使它如下所示。

程序清单 6.9(foo.jsp，有删节)

```
<html>
<body>
<%@ taglib uri="http://java.apache.org/tomcat/examples-taglib" prefix="eg" %>
Radio stations that rock:
<ul>
<eg:foo att1="98.5" att2="92.3" att3="107.7">
<li><%= member %></li>
</eg:foo>
</ul>
<eg:hello/>
</body>
</html>
```

把程序清单 6.9 保存为 TOMCATHOME\webapps\ROOT\foo.jsp，在编写 foo.jsp 的时候，有一点读者必须特别留意，taglib 编译指令的 uri 属性可以乱写，但是这个 uri 属性的值必须和 web.xml 文件中 <taglib-uri>标记对所指定的 uri 的值一致，也就是说只要这两个值一致，不管如何取值都可以，实际上 taglib 编译指令 uri 属性的值是无关紧要的，至少对于 Tomcat 服务器而言是如此，Tomcat 服务器碰到 taglib 编译指令时，第一反应是读取 web.xml 文件，根据 uri 属性值的匹配原则，找到这个 Tag Library 所对应的 TLD 文件(根据 web.xml 文件中<taglib-location>标记对之间的值)，再根据这个 TLD 文件，按图索骥，就能够找特定 Tag 的 Tag Handler 了。

（6）运行程序清单 6.9。启动 Tomcat 服务器，在浏览器的地址栏中输入：

http://Rainbow.pku.edu.cn:8080/foo.jsp

运行效果如图 6.5 所示。

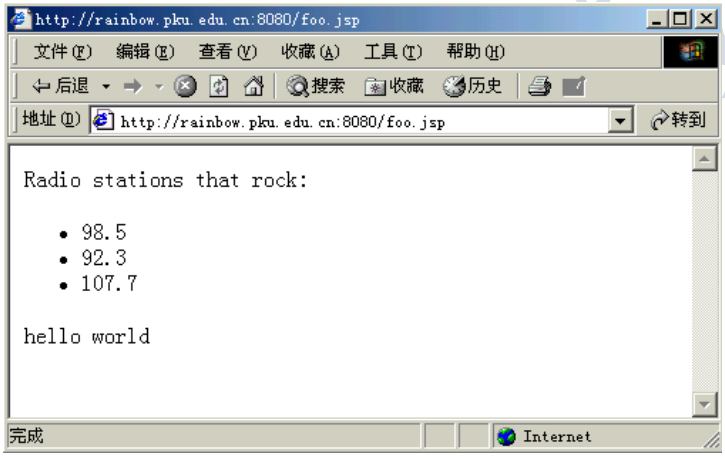


图 6.5　foo.jsp

如图 6.5 所示，hello Tag 已经正确运行了，输出了 hello world。这就是我们编写的第一个 Tag 了。怎么样？效果如何?

6.2.6　代码解析

虽然 Tag Library 已经测试成功了，但是我们对 Tag 的运行流程还是不太清楚，下面我们就来分析由程序清单 6.9 翻译过来的 Java 文件，借以明了 Tag 的运行流程。请看程序清单 6.10，这是由 Tomcat 服务器自动产生的文件。

程序清单 6.10

```
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.FileInputStream;
import java.io.ObjectInputStream;
import java.util.Vector;
import org.apache.jasper.runtime.*;
import java.beans.*;
import org.apache.jasper.JasperException;

public class _0002ffoo_0002ejspfoo_jsp_6 extends HttpJspBase
{
    static
    {
    }
    public _0002ffoo_0002ejspfoo_jsp_6( )
        {
    }
    private static boolean _jspx_inited = false;
    public final void _jspx_init() throws JasperException
        {
    }
    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;
try
{
    if (_jspx_inited == false)
    {
        _jspx_init();
        _jspx_inited = true;
    }
    _jspxFactory = JspFactory.getDefaultFactory();
    response.setContentType("text/html;charset=8859_1");
    pageContext = _jspxFactory.getPageContext(this，request，response，
        ""，true，8192，true);
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();

    out.write("<html>\r\n<!--\r\n    Copyright (c) 1999 The Apache Software
        Foundation. All rights \r\n    reserved.\r\n-->\r\n<body>\r\n");
    out.write("\r\n\r\nRadio stations that rock:\r\n\r\n<ul>\r\n");

    /* ----    eg:foo ---- */
    examples.FooTag _jspx_th_eg_foo_0 = new examples.FooTag();
    _jspx_th_eg_foo_0.setPageContext(pageContext);
    _jspx_th_eg_foo_0.setParent(null);
    _jspx_th_eg_foo_0.setAtt1("98.5");
    _jspx_th_eg_foo_0.setAtt2("92.3");
    _jspx_th_eg_foo_0.setAtt3("107.7");
    try
    {
        int _jspx_eval_eg_foo_0 = _jspx_th_eg_foo_0.doStartTag();
        if (_jspx_eval_eg_foo_0 == Tag.EVAL_BODY_INCLUDE)
            throw new JspTagException("Since tag handler class
            examples.FooTag implements BodyTag，
            it can't return Tag.EVAL_BODY_INCLUDE");
            if (_jspx_eval_eg_foo_0 != Tag.SKIP_BODY)
        {
            try
            {
                if (_jspx_eval_eg_foo_0 != Tag.EVAL_BODY_INCLUDE)
                {
                    out = pageContext.pushBody();
```

北京希望电子出版社

```
                    _jspx_th_eg_foo_0.setBodyContent((BodyContent) out);
                }
                _jspx_th_eg_foo_0.doInitBody();
                do
                {
                    String member = null;
                    member = (String) pageContext.findAttribute("member");
                    out.write("\r\n<li>");
                    out.print( member );
                    out.write("</li>\r\n");
                }
                while(_jspx_th_eg_foo_0.doAfterBody() ==
                    BodyTag.EVAL_BODY_TAG);
            }
            finally
            {
                if (_jspx_eval_eg_foo_0 != Tag.EVAL_BODY_INCLUDE)
                    out = pageContext.popBody();
            }
        }
        if (_jspx_th_eg_foo_0.doEndTag() == Tag.SKIP_PAGE)
            return;
}
finally
{
        _jspx_th_eg_foo_0.release();
}
out.write("\r\n</ul>\r\n");

        /* ----    eg:hello ---- */
        examples.hello _jspx_th_eg_hello_0 = new examples.hello();
        _jspx_th_eg_hello_0.setPageContext(pageContext);
        _jspx_th_eg_hello_0.setParent(null);
        try
        {
            int _jspx_eval_eg_hello_0 = _jspx_th_eg_hello_0.doStartTag();
            if (_jspx_eval_eg_hello_0 == Tag.EVAL_BODY_INCLUDE)
                throw new JspTagException("Since tag handler class examples.hello
                implements BodyTag，
                it can't return Tag.EVAL_BODY_INCLUDE");
            if (_jspx_eval_eg_hello_0 != Tag.SKIP_BODY)
            {
                try
```

北京希望电子出版社

```
                                {
                                    if (_jspx_eval_eg_hello_0 != Tag.EVAL_BODY_INCLUDE)
                                    {
                                        out = pageContext.pushBody();
                                        _jspx_th_eg_hello_0.setBodyContent((BodyContent)out);
                                    }
                                    _jspx_th_eg_hello_0.doInitBody();
                                    do
                                    {
                                    }
                                    while (_jspx_th_eg_hello_0.doAfterBody() ==
                                        BodyTag.EVAL_BODY_TAG);
                                }
                                finally
                                {
                                    if (_jspx_eval_eg_hello_0 != Tag.EVAL_BODY_INCLUDE)
                                        out = pageContext.popBody();
                                }
                            }
                            if (_jspx_th_eg_hello_0.doEndTag() == Tag.SKIP_PAGE)
                            return;
                        }
                        finally
                        {
                            _jspx_th_eg_hello_0.release();
                        }
                        out.write("\r\n</body>\r\n</html>\r\n");
                    }
                    catch (Exception ex)
                    {
                    if (out.getBufferSize() != 0)
                        out.clearBuffer();
                    pageContext.handlePageException(ex);
                    }
                    finally
                    {
                        out.flush();
                        _jspxFactory.releasePageContext(pageContext);
                    }
                }
            }
```

程序清单 6.10 是一个 JSP 扩展类，该类继承了 HttpJspPage 类，覆盖了_jspService()方法，_jspService()方法是最重要的方法。我们来研究它的流程：

北京希望电子出版社

流程 1：首先是获取 application、session、pageContext、config、out 等 JSP 内部对象。

流程 2：创建 Tag 的实例。

对于 foo Tag，有

```
examples.FooTag    _jspx_th_eg_foo_0 = new examples.FooTag();
```

当执行到这一行时，说明 JSP 引擎已经开始处理 foo 标记了。同理可有：

```
examples.hello _jspx_th_eg_hello_0 = new examples.hello();
```

流程 3：利用 Tag 的实例对象，执行 setPageContext ()方法、setParent()方法。

对于 foo Tag，有

```
_jspx_th_eg_foo_0.setPageContext(pageContext);
_jspx_th_eg_foo_0.setParent(null);
```

对于 hello Tag，有：

```
_jspx_th_eg_hello_0.setPageContext(pageContext);
_jspx_th_eg_hello_0.setParent(null);
```

前者的作用是设定 Tag 的输出内容，后者的作用是设定 Tag 的父 Tag。

流程 4：调用 setXXX()方法。

接下来该调用 setXXX()方法，对 Tag 的属性赋值，对于 foo Tag，有

```
_jspx_th_eg_foo_0.setAtt1("98.5");
_jspx_th_eg_foo_0.setAtt2("92.3");
_jspx_th_eg_foo_0.setAtt3("107.7");
```

如果该 Tag 没有属性，也没有 setXXX()方法，那么这一步会被省略，读者可以参考 hello Tag 与 foo Tag 的不同结果，根据 foo Tag 翻译过来的代码段有三个 setXXX()方法，而根据 hello Tag 翻译过来的代码段没有一个 setXXX()方法，这是因为 hello Tag 没有属性，而 foo Tag 有属性。

至于流程 3，两个标记都有相应的代码段。

流程 5：执行 doStartTag()方法。

对于 foo Tag，有：

```
int _jspx_eval_eg_foo_0 = _jspx_th_eg_foo_0.doStartTag();
```

对于 hello Tag，有

```
int _jspx_eval_eg_hello_0 = _jspx_th_eg_hello_0.doStartTag();
```

然后根据不同的返回值，选择不同的执行流程，如果返回值为 Tag.EVAL_BODY_INCLUDE，那么抛出错误(进入流程 12)，如果返回值为 Tag.SKIP_BODY，那么 JSP 引擎立即返回，执行该 Tag 的 doEndTag()方法(进入流程 10)。如果是其他值，那么执行流程 6。

流程 6：执行 pushBody()方法和 setBodyContent()方法

对于 foo Tag，有

```
out = pageContext.pushBody();
_jspx_th_eg_foo_0.setBodyContent((BodyContent) out);
```

对于 hello Tag，有

```
out = pageContext.pushBody();
_jspx_th_eg_hello_0.setBodyContent((BodyContent) out);
```

流程 7：执行 doInitBody()方法。

对于 hello Tag，有

```
_jspx_th_eg_hello_0.doInitBody();
```

对于 foo Tag，有

```
_jspx_th_eg_foo_0.doInitBody();
```

流程 8：循环执行 doAfterBody()方法。

在流程 8 中，循环执行 doAfterBody()方法与 Tag 对之间的 JSP 代码段，只要 doAfterBody()方法的返回值为 BodyTag.EVAL_BODY_TAG，流程 8 就会反复执行，否则进入流程 9。

对于 foo Tag，使用 foo Tag 的 JSP 程序段为(参考程序清单 6.9)：

```
<eg:foo att1="98.5" att2="92.3" att3="107.7">
<li><%= member %></li>
</eg:foo>
```

在 foo Tag 对之间有 HTML 代码与 JSP 表达式： <%= member %>，所以翻译过来的 Java 代码段为：

```
do
{
      String member = null;
      member = (String) pageContext.findAttribute("member");
      out.write("\r\n<li>");
      out.print( member );
      out.write("</li>\r\n");
while(_jspx_th_eg_foo_0.doAfterBody() == BodyTag.EVAL_BODY_TAG);
}
```

对于 hello Tag，使用 foo Tag 的 JSP 程序段为(参考程序清单 6.9)：

```
<eg:hello/>
```

在 hello Tag 对之间没有 HTML 代码与 JSP 表达式，所以翻译过来的 Java 代码段为：

```
do
{
}
while (_jspx_th_eg_hello_0.doAfterBody() == BodyTag.EVAL_BODY_TAG);
```

hello Tag 的流程 8 是一个 do-while 空循环。

流程 9：执行 popBody()方法

对于 foo Tag 与 hello Tag，均有

```
out = pageContext.popBody();
```

流程 10：执行 doEndTag()方法

如果 doEndTag()方法的返回值为 Tag.SKIP_PAGE，那么程序将跳转到流程 12，否则进入流程 11。

对于 hello Tag，有

```
if (_jspx_th_eg_hello_0.doEndTag() == Tag.SKIP_PAGE)
      return;
```

对于 foo Tag，有

```
if (_jspx_th_eg_foo_0.doEndTag() == Tag.SKIP_PAGE)
    return;
```

流程 11：执行 release()方法

执行 release()方法，释放系统资源。然后进入流程 12。

对于 hello Tag，有

```
…
finally
{
    _jspx_th_eg_hello_0.release();
}
```

对于 foo Tag，有

```
…
finally
{
    _jspx_th_eg_foo_0.release();
}
```

流程 12：程序结束。

从流程 1 到流程 12，就是一个 Tag 的完整运行流程，请注意，我们这里讨论的是 Tag 的运行流程，而不是该 JSP 程序的运行流程，这里尽管只讨论了 foo Tag 与 hello Tag，但是这个运行流程对于每一个 Tag 都是适用的。关于流程之间如何跳转的问题，我们在下文还会讨论。

6.3　javax.servlet.jsp.tagext 包介绍

在本节，我们对 javax.servlet.jsp.tagext 包做一个简要介绍，如果你希望更好地了解 Tag 的工作流程，编写功能更为强大的 Tag，而不仅仅是简单的 hello Tag，那么你一定要对 tagext 包有深入了解，我们在此抛砖引玉，希望能够帮助读者更好地掌握 Java Tag Extension API 的知识。

注意：本节介绍的知识完全基于 JSP 1.2 规范中所描述的 Java Tag Extension API。

6.3.1　Tag 接口与 BodyTag 接口

本小节介绍 Tag 接口与 BodyTag 接口。

Tag 接口是十分重要的接口，它定义了 Tag Handler 与 JSP page implementation class 之间通讯的基础协议。主要用于处理 Tag 自身的 Action。

Tag 接口中定义了 4 个整型常数，可以控制 Tag 的执行流程，这些常数分别是：

- SKIP_BODY。忽略执行当前的 Tag，只有 doStartTag()方法与 doAfterBody()方法才有可能返回这个值。
- EVAL_BODY_INCLUDE。把 Tag 的执行结果合并到某个输出流中，只有 doStartTag()方法才有可能返回这个值，如果 Tag Handler 扩展了 BodyTag 接口，那

么 doStartTag()方法不能够返回这个值。

- SKIP_PAGE。忽略 Tag 后面的所有内容，JSP 引擎结束对此 JSP 程序的处理，这样 Tag 后面的代码将没有机会被虚拟机执行到。只有 doEndTag()方法才能够返回这个值。
- EVAL_PAGE。不忽略 Tag 后面的所有内容，JSP 引擎继续对此 JSP 程序的处理，只有 doEndTag()方法才能够返回这个值。

Tag 接口中还定义了一些重要的 Tag 生命周期方法的原型，也就是在 Tag 的执行流程中，顺序执行的那些方法，例如 doStartTag()等方法，我们在下面就把这些方法简要地列出来，读者在实际编程中，如果 Tag Handler 扩展的是 Tag 接口，那么根据实际情况可以覆盖这些方法。

- public void setPageContext(PageContext pc); 设定当前的 pageContext 对象，创建了 Tag Handler 的实例后，紧接着就是执行这个方法，对应于 Tag 执行流程 3。
- public void setParent(Tag t); 设定当前 Tag 的父 Tag 对象，在调用 setPageContext()方法以后就会调用这个方法，对应于 Tag 执行流程 3，这个方法对于处理嵌套的 Tag 很有帮助。
- public Tag getParent(); 获取当前 Tag 的父 Tag 对象，和 setParent()方法对应。
- public int doStartTag() throws JspException; 处理开始标记，例如 <eg:foo attr1="…." …..>，doStartTag()方法对应于 Tag 执行流程 5。
- public int doEndTag() throws JspException; 处理结束标记，例如 </eg:foo>，doEndTag()方法对应于 Tag 执行流程 10。
- public void release(); 作用是释放执行 Tag 所占用的资源，删除 Tag 对象，对应于 Tag 执行流程 11。

下面我们介绍 BodyTag 接口，BodyTag 接口继承自 Tag 接口，它在 Tag 接口的基础之上添加了一些处理 Tag Body 的方法，这样 BodyTag 接口既可以处理 Tag 自身的 Action，也可以处理 Tag Body 的 Action。如果我们需要定义一个类似于 foo Tag 的 Tag，那么此 Tag Handler 一般需要扩展 BodyTag 接口，而不能够扩展 Tag 接口，因为 foo 标记对之间可以加入别的代码。但是 hello Tag 就不同了，hello Tag 的 Body 是 Empty，也就是说不能够有 hello 标记对，hello 只能够单独使用，所以类似于 hello Tag 的 Tag 的 Tag Handler 一般扩展的是 Tag 接口，当然了也可以扩展 BodyTag 接口，因为 BodyTag 接口具有 Tag 接口的全部功能。

BodyTag 接口定义了整型常量：EVAL_BODY_TAG，该常量的意义是继续执行 Tag Body 中的代码，只有 doAfterBody()方法和 doStartTag()方法可以返回这个值，如果 doAfterBody()方法的返回值一直为 EVAL_BODY_TAG，那么 doAfterBody()方法和 Tag Body 中的代码会被反复执行，这是一个死循环，除非 doAfterBody()方法的返回值不是 EVAL_BODY_TAG，读者可以参考上文有关 foo Tag 的说明以及程序清单 6.7/6.10。

注意：只有扩展了 BodyTag 接口的 Tag Handler 的 doAfterBody()方法和 doStartTag()方法才有可能使用此常数。

BodyTag 接口还定义了下面的方法：

- public void setBodyContent(BodyContent b)。setBodyContent()方法对应于 Tag 执行

流程 6，用于设定 Tag 的 BodyContent 属性，所谓的 BodyContent，其实代表的是 Tag Body 中代码段的 JspWriter 对象，BodyContent 可以输出这些代码段的执行结果，也可以读取这些代码段的执行结果(BodyContent 类继承了 JspWriter 类)。

- public void doInitBody() throws JspException。初始化 Tag Body，对应于 Tag 执行流程 7。没有任何返回值。
- public int doAfterBody() throws JspException。doAfterBody()方法是十分重要的方法，对应于 Tag 执行流程 8，我们需要注意的重点在于 doAfterBody()方法的返回值，返回值不同，Tag 执行的流程也会有很大的改变，关于这一点，我们在上文已经反复强调多次了，在这里就不再重复了。

6.3.2　TagSupport 类与 BodyTagSupport 类

TagSupport 类是十分重要的类，它实现了 Tag 接口中定义的方法。在 TagSupport 类中实现的方法如下所示：

- public static final Tag findAncestorWithClass(Tag from，java.lang.Class klass);
- public int doStartTag() throws JspException;
- public int doEndTag()throws JspException;
- public void release();
- public void setParent(Tag t);
- public Tag getParent();
- public java.lang.String getId();
- public void setPageContext(PageContext pageContext);
- public void setValue(java.lang.String k，java.lang.Object o);
- public java.lang.Object getValue(java.lang.String k);
- public void removeValue(java.lang.String k);
- public java.util.Enumeration getValues();

上面列出来的方法中，大部分都是在 Tag 接口中声明过的，意义没有变化，除此以外的 setXXX()方法和 getXXX()方法乃是用于设定或者读取特殊的 Tag 属性的，例如 Tag 的 id，至于 findAncestorWithClass()方法，它的作用是查找和某个类相匹配的 Tag Handler。

BodyTagSupport 类继承了 TagSupport 类，直接实现了 BodyTag 接口，间接实现了 Tag 接口。因此，我们无论编写什么样的 Tag 的 Tag Handler，使它继承 BodyTagSupport 类，总是可行的。

BodyTagSupport 类中实现的方法如下所示，这些方法几乎全部在 Tag 接口、BodyTag 接口、TagSupport 类中声明过或者有定义，因此我们就不再详细介绍了。

- public int doStartTag() throws JspException;
- public int doEndTag() throws JspException;
- public void setBodyContent(BodyContent b);
- public void doInitBody() throws JspException;
- public int doAfterBody() throws JspException;
- public void release();

- public BodyContent getBodyContent();
- public JspWriter getPreviousOut();

6.3.3　TagInfo 类 TagExtraInfo 类

TagInfo 类和 TagExtraInfo 类映射了 Tag 的描述信息，而 Tag 的描述信息在 Tag Library 的 TLD 文件中定义，所以 TagInfo 类和 TagExtraInfo 类正是通过读取 TLD 文件，从而获取 Tag 的描述信息。

在 TagInfo 类中，所定义的重要方法有：

- 获取 Tag 的名字。
- public TagAttributeInfo[] getAttributes(); 获取一个 TagAttributeInfo 数组，TagAttributeInfo 对象包含了 Tag 特定的属性的信息。
- public boolean isValid(TagData data); 验证赋给 Tag 特定属性的值是否有效。在 JSP 程序的执行流程中，Translation 阶段和 Validation 阶段，JSP 引擎会自动调用这个方法校检 Tag。
- public TagExtraInfo getTagExtraInfo(); 获取一个 TagExtraInfo 对象，这个 TagExtraInfo 对象包含了 Tag 的额外信息，至于 TagExtraInfo 类的名字，可以从 TLD 文件中获取，也就是 teiclass 标记所指定的值。
- public java.lang.String getTagClassName(); 获取此 Tag 的 Tag Handler 类的名字，也就是在 TLD 文件中，tagclass 标记所指定的值。
- public java.lang.String getBodyContent(); 获取 Tag Body 可能的类型，返回值是一个字符串常量，可能有 3 种情况：
  BODY_CONTENT_JSP
  BODY_CONTENT_TAG_DEPENDENT
  BODY_CONTENT_EMPTY

这 3 个值和 TLD 文件中 bodycontent 标记所指定的值是一一对应的，读者可以参考上文介绍 TLD 文件时关于 bodycontent 标记的介绍。

注意，这个 getBodyContent()方法和 BodyTagSupport 类的同名方法不一样，读者不要混淆。

- public java.lang.String getInfoString(); 获取对 Tag 的注释信息，亦即 TLD 文件中 tag 标记所包含的 info 标记所指定的值。
- public TagLibraryInfo getTagLibrary(); 返回一个 TagLibraryInfo 对象，利用此对象，可以获取 Tag Library 的信息。

TagExtraInfo 类映射了 Tag 的一些信息，在其中没有什么特别重要的方法。有一个 getTagInfo()方法，可以反过来获取 TagInfo 对象，至于其他的方法，我们就不介绍了，读者可以参考相关的文档。

6.3.4　TagLibraryInfo 类与 TagAttributeInfo 类

TagLibraryInfo 类映射了 Tag Library 的详细信息。TagLibraryInfo 类其实也是通过读取 TLD 文件，才能够获取 Tag Library 的信息。在该类中定义了下面的重要方法。

- public java.lang.String getURI(); 获取 Tag Library 的 URI 属性值。
- public java.lang.String getPrefixString(); 获取 Tag Library 的标识符——prefix，此 prefix 在 taglib 编译指令中指定。
- public java.lang.String getShortName(); 获取 Tag Library 的名字，Tag Library 的 Short Name 在 TLD 文件中指定。
- public java.lang.String getInfoString(); 获取 Tag Library 的注释信息。Tag Library 的 Info 也在 TLD 文件中指定。
- public TagInfo[] getTags(); 获取一个 TagInfo 数组，这个 TagInfo 数组包含了此 Tag Library 中所包含的所有 Tag 的所有详细描述信息。
- public TagInfo getTag(java.lang.String shortname); 获取一个 TagInfo 对象，此 TagInfo 对象包含了此 Tag Library 中特定 Tag 的详细描述信息。方法参数 shortname 指定了 Tag 的名字。

TagAttributeInfo 类映射了 Tag 的特定属性(Attribute)的信息。也就是 TLD 文件中被 attribute 标记所包含的部分。

TagAttributeInfo 类定义的重要方法如下所示：

- public boolean canBeRequestTime(); 获知此属性是否可以通过 JSP 程序段或者 JSP 表达式动态设定。在 TLD 文件中，由 rtexprvalue 标记所指定。
- public boolean isRequired(); 获取此属性是否必须的属性。在 TLD 文件中，由 require 标记所指定。
- public java.lang.String getName(); 获取此属性的名字。在 TLD 文件中，由 tag 标记包含下的 name 标记所指定。
- public java.lang.String getTypeName(); 获取此属性的 Java 类型名称。在 TLD 文件中，由 type 标记所指定。

也许有的读者要问，应该如何使用这些 XXXInfo 类呢?TagLibraryInfo 类有一个构造函数如下所示：

protected TagLibraryInfo(java.lang.String prefix，java.lang.String uri);

我们只要调用这个构造函数，就能够获取 TagLibraryInfo 对象，那么由 TagLibraryInfo 类的 getTags()方法、getTag()方法，又可以获取 TagInfo 对象，再由 TagInfo 对象，也可以轻易地获取 TagExtraInfo 对象和 TagAttributeInfo 对象，这样一来，无论是 Tag Library 的信息，还是 Tag 的信息，甚至是 Tag 特定属性的信息，都尽入我们的掌握之中。

6.3.5　BodyContent 类

BodyContent 类代表了 Tag Body 的内容，BodyContent 类继承自 JspWriter 类，利用它既可以读取 Tag Body 的内容，也可以输出 Tag Body 的内容，我们上面经常说 Tag Body，但是对 Tag Body 一直没有下一个定义，请看下面的例子。

例：

```
<prefix:tag>
    body
</prefix:tag>
```

在上面的例子中，body 即是 Tag Body，利用 BodyContent 类的 getReader()或者 getString()方法，就可以读取这部分内容。如果是调用 writeOut()方法或者是 flush()方法或者是 getEnclosingWriter()方法，就可以把这部分内容输出或者是写入到输出流中。

BodyContent 类定义的方法如下所示，详细的意义我们就不再解释了。

- protected BodyContent(JspWriter e);
- public void flush() throws java.io.IOException;
- public void clearBody();
- public abstract java.io.Reader getReader();
- public abstract java.lang.String getString();
- public abstract void writeOut(java.io.Writer out) throws java.io.IOException;
- public JspWriter getEnclosingWriter();]

关于 Tag Extension API 我们就介绍到这里，对此感兴趣的读者不妨参考相关的文档。

6.4　Tag Library 开发与应用实例

在本节中，我们将运用上文介绍的知识，开发一个完整的 Tag Library。这个 Tag Library 封装了 Java Servlet API 中 ServletContext 接口的方法，功能和 JSP 内部对象 Application 一模一样，不过比后者要更容易使用。这个 Tag Library 是在 Jakarta Tag Libraries 的 Application Tag Library 的基础上加以简化改写而来。在第 7 章，我们介绍了 Jakarta Tag Libraries 的用法，自然也包括了 Application Tag Library 的用法，读者可以参考。

6.4.1　Application Tag Library 的开发目标

我们要开发的 Tag Library 的名字是 Application，要求是达到类似于 JSP 内部对象 Application 的功能，封装 ServletContext 接口的方法。

此 Tag Library 计划开发下面几个 Tag：

（1）attribute：作用是获取某个服务器变量的值。

语法：

<prefix:attribute name="…"/>

属性：

name：必需的。服务器变量的名字。

（2）attributes：作用是循环列出现存的所有服务器变量，还可以获得特定服务器变量的详细信息。

语法：

```
<prefix:attributes id="…">
        …<jsp:getProperty name="…" property="name"/>
…<jsp:getProperty name="…" property="value"/>
</prefix:attributes>
```

属性：

id：必需的。这个参数的值必须和<jsp:getProperty>操作指令中 name 属性的值一样，

这样一来，我们在 attributes 标记体内使用该操作指令(指<jsp:getProperty>)就可以分别获取任何一个服务器变量的名称和值了。

（3）removeattribute：作用是删除某个服务器变量。

语法：

<prefix:removeattribute name="…"/>

属性：

name：必需的。需要移去的服务器变量的名字。

（4）setattribute：作用是设定某个服务器变量的值。

语法：

<prefix:setattribute name="attrname">Value</app:setattribute>

属性：

name：必需的。需要设定新值的服务器变量的名字，在 setattrbiute 标记体内所包含的数据将会被赋给该服务器变量。

6.4.2　定义 TLD 文档

本小节将定义 Application Tag Library 的 TLD 文件——application.tld。

程序清单 6.11(application.tld)

```
<?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">

<!-- a tag library descriptor -->

<taglib>
<!-- after this the default space is
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"
-->

    <!-- The version number of this tag library -->
    <tlibversion>1.0</tlibversion>

    <!-- The JSP specification version required to function -->
    <jspversion>1.1</jspversion>

    <!-- The short name of this tag library -->
    <shortname>application</shortname>

    <!-- Public URI that uniquely identifies this version of the tag library -->
    <uri>http://jakarta.apache.org/taglibs/application</uri>

    <!-- General information about this tag library -->
```

```
<info>
    The APPLICATION custom tag library contains tags which can be used to
    access information contained in the ServletContext for a web application.
</info>

<!-- ******************** Defined Custom Tags ******************-->
<!-- ******************** Defined attribute Tag ******************-->
<tag>
    <name>attribute</name>
    <tagclass>application.AttributeTag</tagclass>
    <bodycontent>empty</bodycontent>
    <info>Get the value of a single application attribute.</info>
    <attribute>
    <name>name</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>

<!-- ******************** Defined attributes Tag ******************-->
<tag>
    <name>attributes</name>
    <tagclass>application.AttributesTag</tagclass>
    <teiclass>application.AttributesTEI</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>Loop through all application attributes.</info>
    <attribute>
        <name>id</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>

<!-- ******************** Defined setattribute Tag ******************-->
<tag>
    <name>setattribute</name>
    <tagclass>application.SetAttributeTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <info>
        Sets the value of an application attribute to the
        content in the body of the tag.
    </info>
    <attribute>
```

北京希望电子出版社

```
            <name>name</name>
            <required>true</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>
    </tag>

    <!-- ******************** Defined removeattribute Tag ******************-->
    <tag>
        <name>removeattribute</name>
        <tagclass>application.RemoveAttributeTag</tagclass>
        <bodycontent>empty</bodycontent>
        <info>Removes an attribute from the application.</info>
        <attribute>
            <name>name</name>
            <required>true</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>
    </tag>
</taglib>
```

6.4.3　编写 Tag Hander

在本小节中，我们将根据 application.tld 文件所描述的信息，分别编写上述 4 个 Tag 的 Tag Handler。

attribute Tag

在 application.tld 文件中指定 attribute Tag 的 Tag Handler 为 AttributeTag 类，下面我们就来编写这个 Tag Handler。请看程序清单 6.12(AttributeTag.java)。

程序清单 6.12

```
//File Name:AttributeTag.java
//Author Morgan Delagrange，modify:fancy
//Note:the attribute Tag's Tag Handler

package application;

import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class AttributeTag extends TagSupport
{
```

北京希望电子出版社

```
    // Name of application attribute
    private String name = null;

    /**
    * Method called at end of Tag to output attribute value
    *
    * @return EVAL_PAGE
    */
    public final int doEndTag() throws JspException
    {
        Object value =pageContext.getServletContext().getAttribute(name);
        if( value == null )
            value = "Not found this attribute or it is null ";

        try
        {
            pageContext.getOut().write(value.toString());
        }
        catch(Exception e)
        {
            throw new JspException("IO Error: " + e.getMessage());
        }
        return EVAL_PAGE;
    }

    /**
    * Set the required tag attribute <b>name</b>.
    *
    * @param String name of application attribute
    */
    public final void setName(String str)
    {
        name = str;
    }
}
```

在程序清单 6.12 中，定义了 setName()方法和 doEndTag()方法，在 attribute Tag 的执行流程中，setName()方法将首先被调用，读取 name 属性的值(是 str)，赋给 Tag Handler 的一个成员变量(也是 name)，在 doEndTag()方法中，利用下面的代码获取相应的服务器变量的值：

```
Object value =pageContext.getServletContext().getAttribute(name);
```

注意，pageContext 是 TagSupport 类的成员变量，因为 Tag Handler 继承了 TagSupport 类，所以可以直接使用使用，获取服务器变量的值以后，把这个值输出：

pageContext.getOut().write(value.toString());

setattribute Tag

在 application.tld 文件中指定 setattribute Tag 的 Tag Handler 为 SetAttributeTag 类，下面我们就来编写这个 Tag Handler。请看程序清单 6.13(SetAttributeTag.java)。

程序清单 6.13

```
//File Name: SetAttributeTag.java
// Author Morgan Delagrange modify:fancy
//Note:setattribute tag's tag handler

package application;

import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class SetAttributeTag extends BodyTagSupport
{
    private String name = null;

    /**
    * Returns EVAL_BODY_TAG so the body of the tag can be evaluated.
    *
    * @return EVAL_BODY_TAG
    */
    public final int doStartTag() throws JspException
    {
    return EVAL_BODY_TAG;
    }

    /**
    * Read the body of the setattribute tag to obtain the attribute value,
    * then set the attribute with <b>name</b> to that value.
    *
    * @return SKIP_BODY
    */
    public final int doAfterBody() throws JspException
    {
        // Use the body of the tag as attribute value
        BodyContent body = getBodyContent();
        String s = body.getString();
```

```
            // Clear the body since we only used it as input for the attribute
            // value
            body.clearBody();
            // set the attribute
            pageContext.getServletContext().setAttribute(name，(Object)s);
            return SKIP_BODY;
        }

        /**
        * Set the required tag attribute <b>name</b>.
        *
        * @param String name of application attribute to set value for
        */
        public final void setName(String str)
        {
            name = str;
        }
    }
```

在程序清单 6.13 中定义了 setattribute Tag 的 Tag Handler——SetAttributeTag 类，SetAttributeTag 类最主要的方法是 doAfterBody()方法，在 setattribute Tag 的执行流程中，首先利用 setName()方法，获取需要设定新值的服务器变量的名字，然后执行 doStartTag()，这个方法没有做任何事情，接着执行 doAfterbody()方法，在这个方法中，首先创建一个 BodyContent 类的实例对象(body)，接着利用该实例对象的 getString()方法读取 setattribute 标记体内的内容，然后把这些内容赋给特定的服务器变量，完成这一个任务的代码如下：

```
pageContext.getServletContext().setAttribute(name，(Object)s);
```

在上面的代码中，name 是服务器变量的名字，s 是要赋给此服务器变量的值。

doAfterBody()方法的返回值为 SKIP_BODY，这说明 doAfterBody()方法只执行一遍，就进入到下一个执行流程中去了，而不会反复执行。

removeattribute Tag

在 application.tld 文件中指定 removeattribute Tag 的 Tag Handler 为 RemoveAttributeTag 类，下面我们就来编写这个 Tag Handler。请看程序清单 6.14(RemoveAttributeTag.java)。

程序清单 6. 14

```
//File Name: RemoveAttributeTag.java
//Author Morgan Delagrange
//Note:the removeattribute Tag's Tag Handler
package application;

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

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

public class RemoveAttributeTag extends TagSupport
{
    private String name = null;

    /**
    * Removes an attribute from the ServletContext.
    *
    * @return SKIP_BODY
    */
    public final int doStartTag() throws JspException
    {
        pageContext.getServletContext().removeAttribute(name);
        return SKIP_BODY;
    }

    /**
    * Set the required tag attribute <b>name</b>.
    *
    * @param String name of application attribute to remove
    */
    public final void setName(String str)
    {
        name = str;
    }
}
```

在 RemoveAttributeTag.java 中，定义了两个方法，分别是 setName()方法和 doStartTag()方法，这两个方法都十分简单，前者是从 removeattribute Tag 的属性中读取需要删除的服务器变量的名字，而后者则是执行删除服务器变量的操作，这一操作使用了下面的代码：

```
pageContext.getServletContext().removeAttribute(name);
```

在上面的代码中 name 是需要删除的服务器变量的名字。

attributes Tag

在 application.tld 文件中指定 attributes Tag 的 Tag Handler 为 AttributesTag 类，TEI class 为 AttributesTEI.java 下面我们就来编写这个 Tag Handler 和 TEI class。请看程序清单 6.15(AttributesTag.java)和程序清单 6.16(AttributesTEI.java)。

程序清单 6.15

```
//File Name:AttributesTag.java
//Author:Morgan Delagrange modify:fancy
//Note:the attributes Tag's Tag Handler
package application;
```

```
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class AttributesTag extends BodyTagSupport
{
    private String name = null;
    private ServletContext app = null;
    private Enumeration attributes = null;
    private String attribute = null;

    /**
    * Loops through all the attributes that came within the ServletContext.
    *
    * @return SKIP_BODY if no attributes are found，
    *EVAL_BODY_TAG if an attribute exists
    */
    public final int doStartTag() throws JspException
    {
        // Get the ServletContext
        app = pageContext.getServletContext();

        attributes = app.getAttributeNames();
        if( attributes == null || !attributes.hasMoreElements() )
            return SKIP_BODY;
        attribute = (String)attributes.nextElement();
        if( attribute == null )
            return SKIP_BODY;

        pageContext.setAttribute(id，this，PageContext.PAGE_SCOPE);
        return EVAL_BODY_TAG;
    }

    /**
    * Method called at end of each attributes tag.
    *
    * @return EVAL_BODY_TAG if there is another attribute，
    *or SKIP_BODY if there are no more attributes
    */
    public final int doAfterBody() throws JspException
```

北京希望电子出版社

```
{
        // See if this is the last attribute
        if( !attributes.hasMoreElements() )
        return SKIP_BODY;
        // There is another attribute，so loop again
        attribute = (String)attributes.nextElement();
        if( attribute == null )
        return SKIP_BODY;
                return EVAL_BODY_TAG;
}

/**
* Method called at end of Tag
* @return EVAL_PAGE
*/
public final int doEndTag() throws JspException
{
        try
        {
                if(bodyContent != null)
                bodyContent.writeOut(bodyContent.getEnclosingWriter());
        }
        catch(java.io.IOException e)
        {
                throw new JspException("IO Error: " + e.getMessage());
                }
                return EVAL_PAGE;
}

/**
* Returns the name of the attribute.
* <p>
* &lt;jsp:getProperty name=<i>"id"</i> property="name"/&gt;
*
* @return String - attribute name
*/
public final String getName()
{
        return attribute;
}

/**
* Returns the value of the attribute.
```

```
     * <p>
     * &lt;jsp:getProperty name=<i>"id"</i> property="attribute"/&gt;
     *
     * @return String - value of the attribute
     */
     public final String getAttribute()
     {
          Object value = app.getAttribute(attribute);
          if( value == null )
          return "";
          return "" + value.toString();
     }

     /**
     * Remove the script variable after attributes tag closed out
     */
     public final void release()
     {
          if( id != null && id.length() > 0 )
          pageContext.removeAttribute(id，PageContext.PAGE_SCOPE);
     }
}
```

应该说，程序清单 6.15 是我们遇到的最复杂的 Tag Handler 了，下面我们来逐一分析此 Tag Handler 中所定义的方法。

在 doStartTag()方法中，首先获取一个服务器对象 app，代码如下所示：

```
app = pageContext.getServletContext();
```

接下来把所有服务器变量的列表赋给了一个枚举对象 attributes，代码如下：

```
attributes = app.getAttributeNames();
```

接下来分析这个枚举对象，看看是否为空，如果为空，那么返回 SKIP_BODY，跳出对这个 Tag 的执行流程。

在 doAfterBody()方法中，每执行一次该方法，便取得枚举对象中下一个元素的值，返回 EVAL_BODY_TAG，使得 doAfterBody()方法连同 Tag Body 中的代码被反复执行，直到枚举对象中所有的元素都被遍历过，这时 doAfterBody()方法的返回值为 SKIP_BODY，跳出 Tag 的执行流程。

然后是 doEndTag()方法和 release()方法。

还有两个方法，分别是 getName()方法和 getAttribute()方法，前者返回某个服务器变量的名字，后者返回某个服务器变量的值，这两个方法何时被调用呢?请看上文 attributes Tag 的语法，我们可以在此 Tag Body 中使用<jsp:getProperty>操作指令分别获取服务器变量的名字和服务器变量的值，这两个方法就在此时被执行，由于 Tag Body 内的代码和 doAfterBody()方法一样，是循环执行的，每循环一次，doAfterBody()方法就会更新 Tag Handler 成员变量 attribute 的值，而 attribute 变量保存的是服务器变量的名字，调用 doAfterBody()方法后，调

用 getName()方法，返回 attribute，又调用 getAttribute()方法，此方法根据 attribute 的值，返回特定服务器变量的值，接着又调用 doAfterBody()方法，就这样一直循环下去，直到满足退出条件。所以使用 attributes Tag，就可以把所有的服务器变量的名字和值都一一输出。

程序清单 6.16

```
//File Name: AttributesTEI.java
//Author Morgan Delagrange
//Note:TEI class for attributes Tag
package application;

import javax.servlet.jsp.tagext.*;

public class AttributesTEI extends TagExtraInfo
{
    public final VariableInfo[] getVariableInfo(TagData data)
    {
        return new VariableInfo[]
        {
            new VariableInfo(data.getAttributeString("id"),
            "application.AttributesTag",
            true,
            VariableInfo.NESTED),
        };
    }
}
```

6.4.4　部署 Application Tag Library

在上面的两个小节中，我们已经把 Application Tag Library 开发好了，现在的问题是如何把它们部署到服务器上面去，请按照下面的步骤进行。

把程序清单 6.11 保存为

TOMCATHOME\webapps\ROOT\WEB-INF\application.tld

修改 TOMCATHOME\webapps\ROOT\WEB-INF\web.xml

加上这么一段代码：

```
<taglib>
<taglib-uri>
        http://java.apache.org/tomcat/application-taglib
    </taglib-uri>
    <taglib-location>
        /WEB-INF/jsp/application.tld
    </taglib-location>
</taglib>
```

保存文件

把程序清单 6.12 至 6.16 保存在下面的目录：

TOMCATHOME\webapps\ROOT\WEB-INF\classes\application\

在 MS-DOS 状态，执行下面的命令：

```
SET CLASSPATH=%CLASSPATH%;TOMCATHOME\lib\servlet.jar
javac AttributeTag.java
javac AttributesTag.java
javac AttributesTEI.java
javac RemoveAttributeTag.java
javac SetAttributeTag.java
```

现在 Application Tag Library 就算部署成功了，但是能不能够使用呢?这还是一个问题。

6.4.5　测试 Application Tag Library

在本小节中，我们将编写一个简单的 JSP 程序，测试我们开发的 Tag Library，源代码如程序清单 6.17。

程序清单 6.17

```
<%--
File Name:application.jsp
Author:fancy
Date:2001.6.4
Note:test the application tag library
--%>
<html>
<body>
<%@ taglib uri="http://java.apache.org/tomcat/application-taglib" prefix="app" %>
set attribute:myatt1，myatt2，myatt3，myatt4，myatt5<br>
<app:setattribute name="myatt1">AAbb1</app:setattribute>
<app:setattribute name="myatt2">AAbb2</app:setattribute>
<app:setattribute name="myatt3">AAbb3</app:setattribute>
<app:setattribute name="myatt4">AAbb4</app:setattribute>
<app:setattribute name="myatt5">AAbb5</app:setattribute>
list attribute:<br>
<app:attributes id="att">
    <jsp:getProperty name="att" property="name"/> =
<jsp:getProperty name="att" property="attribute"/><br>
</app:attributes>
change attribute:myatt1<br>
<app:setattribute name="myatt1">AAbbx</app:setattribute>
get attribute:myatt1<br>
myatt1=<app:attribute name="myatt1"/><br>
remove attribute:myatt1<br>
<app:removeattribute name="myatt1"/>
get attribute:myatt1<br>
myatt1=<app:attribute name="myatt1"/>
```

北京希望电子出版社

```
</body>
</html>
```

程序清单 6.17(application.jsp)十分简单，相信我们不用解释读者也可以看得懂，其运行效果如图 6.6 所示。

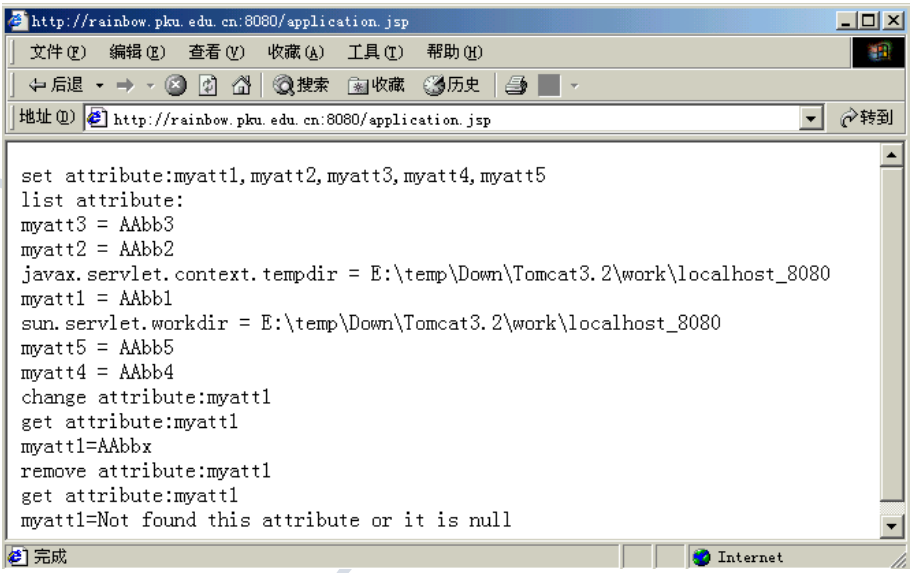


图 6.6　application.jsp

到此为止，一个完整的 Tag Library 就算开发成功了，这么样?读者朋友，你会开发 Tag Library 了吗?我们还可以把上面所开发的 TLD 文件和 Tag Handler，TEI class 都打包到某个 jar 文件中，这样就可以把我们开发的 Tag Library 提供给别人使用了。

6.5　本 章 小 结

本章重点讨论的内容有：XML 技术和 JSP 技术集成的两种模式，并着重描述了第 2 种模式——Tag Library，读者读完了本章，应该能够自己开发 TLD 文件，开发 Tag Handler，分发 Tag Library 到服务器中，并能够编写 JSP 程序使用 Tag Library。本章还讨论了 Tag 的执行流程，这是十分重要的内容，读者只有对其十分明了，才有可能正确编写 Tag Handler。本章还介绍了一些关于 Tag Extension API 的知识，可供读者参考。

我们自己开发的 Tag Library 毕竟功能有限，那么我们为什么不使用别人开发的 Tag Library 呢？在第 7 章和第 8 章，就是一些纯粹手册性的资料，分别介绍了 JRun Tag Library 的使用方法和 Jakarta Tag Libraries 的使用方法，可供读者参考，如果读者对这方面不感兴趣，那么这两章完全可以跳过不读，但是我们相信，如果读者能够应用这两个 Tag Library 于 JSP 程序的开发当中，那么一定可以收到事半功倍的效果。

第 7 章　典型 Tag Library 介绍——JRun Tag Library

在第 6 章我们已经讨论了如何自定义一个 Tag Library，并且把它应用到 JSP 程序开发中去，但是我们自己开发的 Tag Library 功能毕竟有限，现在市场上有不少可供商业应用的 Tag Library，这些 Tag Library 编写合理，功能强大，使用方便，涵盖了 JSP 编程的方方面面，合理使用这些 Tag Library，可以让我们迅速开发出功能强大的 JSP 程序来。这些 Tag Library 之中的佼佼者就是 Allaire 公司开发的 Tag Library 和 Apache Jaktara 开发组开发的 Tag Library。

本章将介绍 JRun Tag Library 的用法。在下一章我们将介绍 Jakarta Tag Library 的用法。

本章没有任何需要硬性掌握的内容，读者完全可以把本章当作一本手册来阅读，在需要的时候查阅一下即可，不需要详细通读全章。

注意：本章的部分内容参考了 Allaire 公司的技术文档(JRun Tag Library Reference)，参考的部分是定义具体标记的 TLD 文件以及标记的语法形式，有些标记的应用实例是由原文档该改写而来。

7.1　JRun Tag Library 简介

7.1.1　JRun Tag Library 介绍

JRun Tag Library 是 Allaire 公司开发的一套标记库，支持多种功能，例如常用的 SQL 操作、收发电子邮件、使用 JMS、JTS、JNDI 等服务、支持 XML 和 Servlet 等功能。我们可以使用 SQL 标记执行实际的数据库查询操作，然后利用 Query2Xml 标记把数据库返回的查询结果转换为 XML 格式的文件，最后利用 Xslt 标记，调用指定的 xsl 文件，把这个 XML 文件显示出来。我们只是聊举一例，说明 JRun Tag Library 的作用，其实 JRun Tag Library 的功能远不止此，读者如果想一窥全豹，不妨继续往下看。

7.1.2　JRun Tag Library 列表

JRun Tag Library 中支持的 Tag 如表 7.1 所列。

表 7.1　JRun Tag Library 所含的标记列表

标记名	描述
Sql	执行数据库查询操作
SqlParam	和 Sql 标记一起使用，用于创建动态的 SQL 语句
SendMsg	用于发送同步的消息
MsgParam	和 SendMsg 标记一起使用，设定标记的属性
GetMsg	接收消息
Transaction	执行与事务相关的操作

（续表）

标记名	描述
Jndi	查找目录服务或者对象，例如 EJB、CORBA 等
SendMail	发送 email
MailParam	和 SendMail 标记一起使用，用于设定邮件的属性，如附件等
GetMail	获取邮件信息
Servlet	调用 Java Servlet 程序
ServletParam	和 Servlet 标记一起使用，设定 Servlet 程序的属性
Query2Xml	把数据库查询返回的结果集转换为 XML 文件的形式
Xslt	处理 XSLT 文件
Form	自动产生 JavaScript 代码在客户端验证用户的输入(针对 Form 结构)
Input	自动产生 JavaScript 代码在客户端验证用户的输入(针对 Input 结构)
Select	自动产生 JavaScript 代码在客户端验证用户的输入(针对 Select 结构)
Param	声明程序段变量
ForEach	循环过程控制，类似于 Visual Basic 的 ForEach 语句
If	执行条件判断，用于程序流程控制
Switch	类似于 Java 语言中的 Switch 关键字，功能也类似
Case	和 Switch 标记一起使用

7.1.3　如何使用 JRun Tag Library

如何使用 JRun Tag Library 呢?我们的建议是你最好选用 JRun 服务器，那么你就可以在 JSP 程序中自由使用 JRun Tag Library 了，如果你使用的是其他的服务器，那么你需要到 Allaire 公司的站点(www.allaire.com)去下载包含 JRun Tag Library 的 Jar 文件，并且把它添加到 CLASSPATH 的路径中去。在 JSP 程序中没有什么特别的设置。

7.1.4　本章体例

本章的内容是这样组织的，对于每个具体的标记，首先介绍它的用途，然后是它的语法形式，每个属性的意义如何，接下来是定义这个标记的 TLD 文件，最后结合一个具体的例子来介绍这个标记的用法。如果是比较复杂的例子则会有代码解释，反之，就是简单地把程序代码列出来就完了。

7.2　SQL 标记

7.2.1　Sql 标记

用途：

Sql 标记主要用于处理与数据库相关的操作，在<Sql>与</Sql>标记之间可以插入任何静态的 SQL 语句，这些 SQL 语句将会被发送到数据库引擎中去，并被执行，同时 Sql 标记还可以配合 SqlParam 标记一起使用，创建动态的 SQL 语句，Sql 标记可以通过三种方式与

数据库系统建立连接。第一种方式是直接使用现存的 JDBC Connection 对象(java.sql.Connection 接口)，第二种方法是使用 J2EE Data Source(javax.sql.DataSource)对象，这是一个 JNDI 服务对象，第三种方法是使用指定的 JDBC 数据库驱动程序与数据库服务的 URL 地址与数据库服务器建立连接。这三种方法我们都会在下面的例子中涉及。

Sql 标记可以包含在 Transaction 标记中，这时，Sql 标记体内的操作被当作事务的一部分执行，这个事务中的操作一起被执行，如果每一个操作都成功了，那么所有的状态改变都将成为事实，如果有任何一个操作失败了，那么整个操作就会失败，这个事务所涉及的状态的改变都不会起任何作用，而保持原状。

语法：

```
<sql ...>
sql statement with optional <sqlparam .../>
...
</sql>
```

属性：

connection：指定 Connection 对象的名称。

id：执行 SQL 语句所返回的记录集的名字就是 id 属性的值，如果执行 SQL 语句后不返回任何有效的记录集，那么这个属性将会被忽略。

scope：该属性类似于 JavaBeans 的 Scope 属性，可以取的值为 Page、Session、Request、Application 等。

datasrc：指定数据源的名字，该名字代表一个 JNDI 服务，也就是说我们可以把数据源作为一个 JNDI 服务发布，在 datasrc 属性中指定这个 JNDI 服务的名字，就可以访问此数据源了。

username：指定访问数据库所需要的用户名。

password：指定访问数据库所需要的密码。

driver：指定访问数据库所需要的驱动程序名，例如 jdbc.odbc.JdbcOdbcDriver。

url：指定数据源的标示，例如 jdbc:odbc:test。

TLD 文件：

```
<tag>
<name>sql</name>
<tagclass>allaire.taglib.SqlTag</tagclass>
<teiclass>allaire.taglib.SqlTei</teiclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>id</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>scope</name>
<required>false</required>
```

```
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>driver</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>url</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>datasrc</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>connection</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>username</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>password</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ page import="java.sql.*" %>
<%@ page import="javax.sql.*，allaire.taglib.*" %>
<%@ taglib uri="jruntags" prefix="jrun" %>

<%
Class.forName("jdbc.odbc.JdbcOdbcDriver");
Connection con = DriverManager.getConnection("jdbc:odbc:test"，"sa"，"");
%>
```

北京希望电子出版社

```
<jrun:sql connection="<%= con %>" id="q1">
SELECT * FROM    goods
</jrun:sql>

<jrun:sql driver=" jdbc.odbc.JdbcOdbcDriver " url=" jdbc:odbc:test" id="q2"
username=”sa” password=””>
SELECT * FROM goods
</jrun:sql>

<%-- sql uses java:comp/env/jdbc/dsn1 to lookup a datasource --%>
<jrun:sql datasrc="dsn1" id="q3">
SELECT * FROM goods
</jrun:sql>

<%-- you can enumerate the QueryTable by: --%>
<jrun:param id="q3" type="QueryTable"/>
<jrun:foreach item="x" group="<%= q3.Names %>">
<%= x %><br>
</jrun:foreach>

<jrun:foreach group="<%= q3 %>">
<jrun:foreach item="y" group="<%= q3.Values %>">
<%= y %><br>
</jrun:foreach>
</jrun:foreach>

<%-- OR --%>
<jrun:param id="q3" type="QueryTable"/>
<jrun:foreach group="<%= q3 %>">
<%
int count = q3.getColumnCount();
for (int i = 0;i < count;i += 1)
{
%>
<%= q3.get(i) %><br>
<%
}
%>
</jrun:foreach>

<%-- OR --%>
<jrun:param id="q3" type="QueryTable"/>
<%while (q3.next())
```

北京希望电子出版社

```
 {
%>
<jrun:foreach item="y" group="<%= q3.Values %>">
<%= y %><br>
</jrun:foreach>
<%
}
%>

<%-- OR，if you want to use column names... --%>
<jrun:param id="q3" type="QueryTable"/>
<jrun:foreach group="<%= q3 %>">
<%= q3.get("id") %><br>
<%= q3.get("lastname") %><br>
<%= q3.get("firstname") %><br>
</jrun:foreach>

<%-- OR，if you want to use column index... --%>
<jrun:param id="q3" type="QueryTable"/>
<jrun:foreach group="<%= q3 %>">
<%= q3.get(1) %><br>
<%= q3.get(2) %><br>
<%= q3.get(3) %><br>
</jrun:foreach>
```

代码解释：

在上面的实例中，读者只需要了解 Sql 标记是如何与数据源建立连接的就可以了。第一种情况，首先创建一个 Connection 接口的实例对象，然后指定 Sql 标记的 connection 属性的值为这个 Connection 对象。第二种情况，直接指定 Sql 标记的 driver 属性和 url 属性的值，Sql 标记就利用这些信息与数据库建立连接。第三种情况指定 datasrc 属性的值为 JNDI 服务名 dsn1，JSP 引擎会自动寻找(lookup)这个服务，然后通过它来访问数据源。与数据源建立连接以后，就可以把 SQL 语句发送到数据源中去，执行的结果将会保存到一个记录集对象中，这个记录集对象的名字就是 Sql 标记 id 属性的值。接下来，就是利用各种方法把记录集对象中的数据输出来，这里用到了 ForEach 标记和 SqlParam 标记的功能，我们将在下面分别介绍这两个标记的功能。

7.2.2　SqlParam 标记

用途：

SqlParam 标记主要是配合 Sql 标记一起使用，创建动态的 SQL 语句，SqlParam 标记特别适合于指定 SQL 语句的输入属性。

语法：

```
<sqlparam ... />
```

属性：

value：指定 SQL 输入属性的值。可以是一个 JSP 表达式也可以是普通的字符串。

sqltype：指定 SQL 输入属性的 SQL 数据类型。可能是下面的值：ARRAY，BIGINT，BINARY，BIT，BLOB，CHAR，CLOB，DATE，DECIMAL，DISTINCT，DOUBLE，FLOAT，INTEGER，JAVA_OBJECT，LONGVARBINARY，LONGVARCHAR，NULL，NUMERIC，OTHER，REAL，REF，SMALLINT，STRUCT，TIME，TIMESTAMP，TINYINT，VARBINARY，VARCHAR。这些值在 java.sql.Types 类中有定义。

scale：如果 SQL 语句的输入属性是一个数字，那么 scale 属性将指定小数点后的数字的位数。

TLD 文件：

```
<tag>
<name>sqlparam</name>
<tagclass>allaire.taglib.SqlParamTag</tagclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>value</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>sqltype</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>scale</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:sql driver="jdbc.odbc.JdbcOdbcDriver" url="" id="result">
SELECT * FROM goods where id =<jrun:sqlparam Sqltype=” INTEGER”
value="<%=idnumber %>"/>
</jrun:sql>
```

北京希望电子出版社

7.3　J2EE 标记

7.3.1　SendMsg 标记

用途：

SendMsg 标记可以利用 JMS 服务(Java Message Service)发送文本消息。SendMsg 标记可以用于 Transaction 标记之中，作为某个事务的特定操作而执行，如果整个事务的所有操作都成功了，那么消息也就成功地发送出去了，如果事务的某个操作失败了，那么整个事务都失败了，消息的发送也失败了。在这一点上，SendMsg 标记和 Sql 标记是十分类似的。SendMsg 标记可以和 MsgParam 标记一起协同使用，以便设定附加消息的属性。

语法：

<sendmsg ...>
message body with optional <msgparam ... />...
</sendmsg>

属性：

msgsrc：必需。根据 msgsrc 属性的值，可以利用 JNDI 服务的查找方法获取到消息队列源的连接。

queue：必需。消息队列服务所对应的 JNDI 服务的名字。

username：可选属性。访问消息队列所需要认证的用户名。

password：可选属性。访问消息队列所需要认证的用户密码。

delivery：可选属性。指定消息分发的模式，合法的值为 PERSISTENT 或者 NON_PERSISTENT，缺省值为前者。

priority：可选属性。该属性值决定消息的优先级，缺省值为 4。

expire：可选属性。指定消息的失效时间，缺省值为永不失效，这个属性的单位是微秒。

TLD 文件：

```
<tag>
<name>sendmsg</name>
<tagclass>allaire.taglib.SendMsgTag</tagclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>msgsrc</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>queue</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
```

```
<name>username</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>password</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>delivery</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>priority</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>expire</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>

<%-- sendmsg uses java:comp/env/jms/QueueConnectionFactory to lookup
a queue connection factory and uses java:comp/env/jms/Queue1 to lookup
a queue. --%>

<jrun:sendmsg msgsrc="QueueConnectionFactory" queue="Queue1">
HELLO WORLD!!.
</jrun:sendmsg>

<%-- OR，if the factory and the queue can't be found from the
default InitialConext，it is necessary to use jndi to specify
a custom provider class name and url. --%>
<jrun:jndi provider="..." url="..."
name="java:comp/env/jms/QueueConnectionFactory" id="f"/>
<jrun:jndi provider="..." url="..."    name="java:comp/env/jms/Queue1" id="q"/>
```

北京希望电子出版社

```
<jrun:sendmsg msgsrc="<%= page.getAttribute("f") %>"
queue="<%= page.getAttribute("q") %>">
This is a text message.
</jrun:sendmsg>
```

7.3.2　MsgParam 标记

用途：

MsgParam 标记主要和 SendMsg 标记一起使用，指定 JMS 消息的属性。

语法：

<msgparam ... />

属性：

name：必选的属性，指定 JMS 消息的属性名字。

value：必选的属性，指定 JMS 消息的属性值，与前者相对。

TLD 文件：

```
<tag>
<name>msgparam</name>
<tagclass>allaire.taglib.MsgParamTag</tagclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>name</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>value</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>

<%-- sendmsg uses java:comp/env/jms/QueueConnectionFactory to lookup
a queue connection factory and uses java:comp/env/jms/Queue1 to lookup
a queue. --%>
<jrun:sendmsg msgsrc="QueueConnectionFactory" queue="Queue1">
<jrun:msgparam name="foo" value="bar"/>
This is a text message.
</jrun:sendmsg>
```

北京希望电子出版社

7.3.3　GetMsg 标记

用途：

GetMsg 标记主要用于从消息队列中提取特定的消息。在 GetMsg 标记体内，我们可以使用 SQL-SELECT 语句完成这个任务，SQL-SELECT 语句借鉴了 SQL 语言的语法，但是又有些微小的差别。SQL-SELECT 语句的语法如下：

```
SELECT {a comma-separated list of property names or *}
FROM {message queue name}
WHERE {a valid JMS message selector string as defined in JMS spec.}
```

GetMsg 标记可以在 Transaction 标记体内使用。

语法：

```
<getmsg ...>
SQL-SELECT filter string...
</getmsg>
```

属性：

msgsrc：必需。根据 msgsrc 属性的值，可以利用 JNDI 服务的查找方法获取到消息队列源的连接。

username：可选属性。访问消息队列所需要认证的用户名。

password：可选属性。访问消息队列所需要认证的用户密码。

id：必选的属性，利用这个属性，我们可以在当前页面任意的 JSP 程序段内引用这个 JMS 消息。

scope：类似于 JavaBeans 的 Scope 属性，可以取 Application、Session、Request、Page 等 4 个值。

TLD 文件：

```
<tag>
<name>getmsg</name>
<tagclass>allaire.taglib.GetMsgTag</tagclass>
<teiclass>allaire.taglib.GetMsgTei</teiclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>msgsrc</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>username</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>password</name>
```

北京希望电子出版社

```
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>id</name>
<required>true</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>scope</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ page import="allaire.taglib.*" %>
<%@ taglib uri="jruntags" prefix="jrun" %>

<jrun:getmsg msgsrc="QueueConnectionFactory" id="table">
SELECT * FROM MsgQueue
</jrun:getmsg>

<%-- You can enumerate the JMS messages by using the same technique as
described in sql section. --%>
<jrun:param id="table" type="MessageTable"/>
<jrun:foreach group="<%= table %>">
...
</jrun:foreach>
```

7.3.4　Transaction 标记

用途：

利用 Transaction 标记可以若干个操作都封装在一个事务中，当这个事务被执行时，只有其中所有的操作都成功了，那么这个事务才算成功，所有的状态改变才会生效，如果有任意一个操作失败，那么系统将会执行回滚操作，已经完成的操作所产生的状态改变将被取消，一切将恢复原状。Transaction 标记可以包含的标记包括 Sql 标记、SendMsg 标记、GetMsg 标记等。

语法

```
<transaction>
<sql ...>
sql statement
</sql>
...more <sql>，<sendmsg> or <getmsg> tags...
```

北京希望电子出版社

```
</transaction>
```

属性：

无。

TLD 文件：

```
<tag>
<name>transaction</name>
<tagclass>allaire.taglib.TransactionTag</tagclass>
<bodycontent>JSP</bodycontent>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:transaction>
<jrun:sql datasrc="s1">
<%-- java:comp/env/jdbc/s1 --%>
INSERT INTO user(username，password) VALUES(“fancy”，”fancy”)
</jrun:sql>

<jrun:sql datasrc="s2">
<%-- java:comp/env/jdbc/s2 --%>
SELECT * FROM goods
</jrun:sql>

<%--
java:comp/env/jms/QueueConnectionFactory
java:comp/env/jms/Queue1
--%>
<jrun:sendmsg msgsrc="QueueConnectionFactory" queue="Queue1">
Successful transaction...
</jrun:sendmsg>
</jrun:transaction>
```

7.3.5　Jndi 标记

用途：

Jndi 标记允许 JSP 程序在 J2EE 服务器环境下处理分布式的对象。Jndi 标记支持 4 个主要的操作，分别是：lookup、list、search、attributes。

语法：

```
<jndi ... />
```

属性：

action：必需的属性。这个属性指定了一个 JNDI 操作，它的值可能是 lookup、list、search、attributes 四者之一。如果 action= “lookup”，那么这个标记将从 JNDI Server 返回一个对象。如果 action=“list”，那么 Jndi 标记将会返回所有匹配的对象的列表。如果 action=“attributes”，

那么 Jndi 标记将从指定的目录服务(例如 LDAP)中返回一个属性的列表。如果 action=“search”，那么 Jndi 标记将从指定的目录服务返回所有与该服务绑定对象的一个枚举。

name：必需的属性。这个属性描述了 JNDI lookup/search 操作的目标对象。

provider：必需的属性。提供 JNDI lookup/Directory 服务的类名。

url：必需的属性。提供 JNDI lookup/Directory 服务的 URL 地址。

id：必需的属性。在下面的 JSP 程序段中可以凭借这个属性来引用 JNDI 服务。

scope：可选的属性。类似于 JavaBeans 的 Scope 属性。

attributes：必需的属性。attribute 属性的值必须是 java.util.Dictionary，java.util.Map 或者 javax.naming.Attributes 对象。 Jndi 标记的 search 操作就以这个属性的值为检索条件。

TLD 文件：

```
<tag>
<name>jndi</name>
<tagclass>allaire.taglib.JndiTag</tagclass>
<teiclass>allaire.taglib.JndiTei</teiclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>id</name>
<required>true</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>scope</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>action</name>
<required>true</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>name</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>provider</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>attributes</name>
```

北京希望电子出版社

```
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>url</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ page import="allaire.taglib.*" %>
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:jndi action="lookup" name="java:comp/UserTransaction" id="txn"/>
<jrun:jndi action="list" name="java:comp/env/jdbc" id="enum"/>
```

7.3.6　Servlet 标记

用途：

Servlet 标记可以使用 Java Servlet API 的 RequestDispatcher 接口的 include()方法调用外部的 Servlet 程序，外部 Servlet 程序的运行结果会嵌在调用它的 JSP 程序中。Servlet 标记可以配合 ServletParam 标记一起使用。设定外部 Servlet 程序的调用属性。

语法：

```
<servlet ...>
optional <servletparam ... />
</servlet>
```

属性：

code：必需的属性。code 属性的值就是外部 Servlet 程序的路径和名字。

TLD 文件：

```
<tag>
<name>servlet</name>
<tagclass>allaire.taglib.ServletTag</tagclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>code</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:servlet code="SnoopServlet">
</jrun:servlet>
```

7.3.7　ServletParam 标记

用途：

ServletParam 标记必须和 Servlet 标记一起使用，设定调用外部 Servlet 程序所需的属性。

语法：

<servletparam ... />

属性：

name：必选属性。Servlet 程序所需属性的名字。

value：必选属性。Servlet 程序所需属性的值。

TLD 文件：

```
<tag>
<name>servletparam</name>
<tagclass>allaire.taglib.ServletParamTag</tagclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>name</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>value</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:servlet code="SnoopServlet">
<jrun:servletparam name="key" value="<%= new Hashtable() %>"/>
</jrun:servlet>
```

7.4　Mail 标记

7.4.1　SendMail 标记

用途：

SendMail 标记主要用于发送邮件。单独使用 SendMail 标记可以发送普通邮件，如果配合 MailParam 标记使用就可以发送带有附件的邮件了。在 SendMail 标记体内就是邮件的正文，在 SendMail 标记中可以设定邮件服务器的信息。SendMail 标记就根据这些信息与邮件服务器建立连接，并且使用它把邮件发送到目标地址。我们还可以使用 GetMail 标记把邮箱中的信件取回。

北京希望电子出版社

语法：

<sendmail ...>
mail body with optional <mailparam ... />
</sendmail>

属性：

host：必需的属性。邮件服务器的名字，例如 smtp.263.net、263.net

port：可选的属性。邮件服务器的服务端口号。缺省值为 25，SMTP 服务的缺省端口号为 25，POP3 服务的端口号为 110。

timeout：可选的属性。邮件服务器的超时时间，如果客户端连接上邮件服务器后，在指定的时间范围内没有采取任何操作，那么服务器会自动断开这个连接。timeout 的数据类型是字符串型，单位是微秒，缺省值为 3000。

session：必需的属性。session 属性表示客户端与邮件服务端的会话过程。赋给这个属性的值必须是某个现存的 Java Mail Session 对象或者这个 Session 对象的名字。如果把 Session 对象的名字以字符串的形式赋给 session 属性，那么 JSP 引擎会自动调用 JNDI 服务的 lookup()方法查询是否存在 java:comp/env/mail/[session]这个对象，关于 session 属性的用法，读者可以参考下面的代码实例。关于 Java Mail Session 对象的知识，读者可以参考第 12 章的相关内容。

注意：使用了 session 属性就不能够使用 host 属性了，反之亦然。

sender：必需的属性。发送邮件者的邮箱名称。

recipient：必需的属性。邮件接收者的邮箱地址。

cc：可选的属性。指 carbon copy e-mail addresses。

bcc：可选的属性。指 blind carbon copy e-mail addresses。

subject：可选的属性。subject 属性指定邮件的主题。

TLD 文件：

```
<tag>
<name>sendmail</name>
<tagclass>allaire.taglib.SendMailTag</tagclass>
<teiclass>allaire.taglib.SendMailTei</teiclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>host</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>port</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
```

```
<attribute>
<name>timeout</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>session</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>sender</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>recipient</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>cc</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>bcc</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>subject</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:sendmail host="smtp.263.net" sender="fancy@263.net" recipient="fancy@263.net">
Hello world!
</jrun:sendmail>

<%-- java:comp/env/mail/session1 --%>
```

北京希望电子出版社

```
<jrun:sendmail session="session1" sender="fancy@263.net" recipient="fancy@263.net">
Hello world.
</jrun:sendmail>
```

7.4.2　MailParam 标记

用途：

MailParam 标记需要和 SendMail 标记一起配合使用，发送带有附件的特殊邮件，利用 MailParam 标记我们还可以设定特殊的邮件信息头(e-mail headers)。

语法：

```
<mailparam ... />
```

属性：

attachurl：必需的属性。attachurl 属性指定邮件附件的地址，既可以是绝对地址，也可以是相对地址。

name：必需的属性。name 属性指定特殊的 email 信息头的名称。

value：必需的属性。指定特殊的 email 信息头的值，与上面的 name 属性相对。

注意：attachurl 属性一般不和 name、value 属性同时使用。

TLD 文件：

```
<tag>
<name>mailparam</name>
<tagclass>allaire.taglib.MailParamTag</tagclass>
<teiclass>allaire.taglib.MailParamTei</teiclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>attachurl</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>name</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>value</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
```

北京希望电子出版社

```
<jrun:sendmail host="smtp.263.net" sender="fancy@263.net" recipient="fancy@263.net">
Hello world.
<jrun:mailparam attachurl="http://www.pku.edu.cn/index.html" />
</jrun:sendmail>
```

7.4.3　GetMail 标记

用途：

顾名思义，GetMail 标记主要用于从 POP3 服务器上的邮箱中取回属于自己的信件。

语法：

```
<getmail ...>
SQL-SELECT e-mail retrieval string.
</getmail>
```

在 GetMail 标记体内，我们使用一种类似于 SQL 语句的语法，检索服务器上的邮箱，看看有没有符合要求的邮件。这种检索语句的语法如下：

```
SELECT {a comma-separated list of mail headers or *}
FROM {mail folder name}
WHERE {optional search terms available in JavaMail API}
```

mail folder name 指的是邮箱的名字，例如 INBOX。

search terms 指的是可供检索的条件，这些条件在 Java Mail API 中都有规定，如表 7.2 所示。

表 7.2　Search Terms 列表

Valid Search Terms for getmail Retrieval String		
Search Term	Valid Operators	Data Type/format
Sender	=，contains	String literal
Recipient	=，contains	String literal
Cc	=，contains	String literal
Bcc	=，contains	String literal
MessageID	=，contains	String literal
MessageNumber	=，>=，>，<=，<，!=	integer
Subject	=，contains	String literal
SentDate	=，>=，>，<=，<，!=	MM/dd/yyy hh:mm:ss
ReceivedDate	=，>=，>，<=，<，!=	MM/dd/yyy hh:mm:ss
Size	=，>=，>，<=，<，!=	integer
Flag	=，!=	One the following: • ANSWERED • DELETED • DRAFT • FLAGGED • RECENT • SEEN • USER
Body	contains	contains

逻辑操作符“and”、“or”、“not”也可以用在检索语句中，以便把多个检索条件组合起来。例如：

```
SELECT * FROM InBox
WHERE (Sender contains 'java.sun.com' or Sender contains '263.net') and Size < 1000
```

属性：

host：必需的属性。邮件服务器的名字，例如 smtp.263.net、263.net。

port：可选的属性。邮件服务器的服务端口号。缺省值为 110，SMTP 服务的缺省端口号为 25，POP3 服务的端口号为 110。

timeout：可选的属性。邮件服务器的超时时间，如果客户端连接上邮件服务器后，在指定的时间范围内没有采取任何操作，那么服务器会自动断开这个连接。timeout 的数据类型是字符串型，单位是微秒，缺省值为 3000。

session：必需的属性。session 属性表示客户端与邮件服务端的会话过程。赋给这个属性的值必需是某个现存的 Java Mail Session 对象或者这个 Session 对象的名字。如果把 Session 对象的名字以字符串的形式赋给 session 属性，那么 JSP 引擎会自动调用 JNDI 服务的 lookup()方法查询是否存在 java:comp/env/mail/[session]这个对象，关于 session 属性的用法，读者可以参考下面的代码实例。关于 Java Mail Session 对象的知识，读者可以参考第 12 章的相关内容。

username：必需的属性。邮箱用户的名字。

password：必需的属性。进入邮箱所需要的用户密码。

id：必需的属性。邮件的 id 号，一旦使用 GetMail 标记取得了邮件的信息，那么在下面的 JSP 程序中就可以利用这个 id 号来引用邮件的信息。

scope：类似于 JavaBeans 的 Scope 属性。

protocol：所使用的邮件协议的名称，合法值为 pop3 和 imap，smtp 不能用在这里，因为 smtp 协议提供发送信件的服务，不提供邮箱的服务。

TLD 文件：

```
<tag>
<name>getmail</name>
<tagclass>allaire.taglib.GetMailTag</tagclass>
<teiclass>allaire.taglib.GetMailTei</teiclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>host</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>port</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
```

```
<attribute>
<name>timeout</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>session</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>id</name>
<required>true</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>scope</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>username</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>password</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>protocol</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ page import="allaire.taglib.*" %>
<%@ taglib uri="jruntags" prefix="jrun" %>

<jrun:getmail host="263.net" username="fancy" password="fancy"
id="e-mails" protocol="pop3">
SELECT * FROM INBOX/Save WHERE Subject contains '小妹你好'
```

北京希望电子出版社

```
</jrun:getmail>

<%-- java:comp/env/mail/session1 --%>
<jrun:getmail session="session1" id="e-mails" protocol="pop3">
SELECT * FROM INBOX/Save WHERE Subject contains '小妹你好'
</jrun:getmail>
<jrun:param id="e-mails" type="e-mailTable"/>
<jrun:foreach group="page.e-mails">
<%=e-mails.get("...") %><br>
</jrun:foreach>
```

7.5　XML 标记

7.5.1　Query2Xml 标记

用途：

Query2Xml 标记可以把规则的数据格式化转化为 XML 格式的数据。

语法：

<query2xml ... />

属性：

query：必需的属性。java.sql.ResultSet，javax.sql.RowSet 等对象的名字。关于这两个借口的信息，读者可以参考本书的第三部分，query 属性其实是指定了 Query2Xml 标记所使用的数据的来源。

id：可选的属性。在下面的 JSP 程序中，就需要靠这个 id 号才能引用被转化为 XML 格式的数据。

type：可选的属性。设定原始数据应该转化为何种类型的 XML 模型。合法的值是 DOM 和 TEXT。如果该值是 DOM，那么 org.w3c.dom.Document 对象将会被创建、如果该值是 TEXT，那么 java.io.BufferedReader 对象将要被创建。这两个对象都可以读取规范的 XML 文档。

scope：可选的属性。类似于 JavaBeans 的 Scope 属性。

rootname：可选的属性。设定 XML 文档中 query 标记的名称，缺省值是 table。

rowname：可选的属性。设定 XML 文档中 row 标记的名称，缺省值是 row。

TLD 文件：

```
<tag>
<name>query2xml</name>
<tagclass>allaire.taglib.Query2XmlTag</tagclass>
<teiclass>allaire.taglib.Query2XmlTei</teiclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>query</name>
<required>true</required>
```

```
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>id</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
      </attribute>
<attribute>
<name>scope</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>type</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>rootname</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>rowname</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ page contentType"text/xml" %>
<%@ page import=”java.sql.*” %>
<%@ page import=”java.io.*” %>

<?xml version="1.0"?>
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:sql driver="Jdbc.Odbc.JdbcOdbcDriver" url="jdbc:odbc:test"
id="rs" scope="session" username=”sa” password=””>
SELECT * FROM goods
</jrun:sql>
<jrun:query2xml query="session.rs" />

<%
Class.forName("jdbc.odbc.JdbcOdbcDriver");
```

北京希望电子出版社

```
Connection con = DriverManager.getConnection("jdbc:odbc:test"，”sa”，””);
Statement stmt=con.createStatement();
ResultSet rs1=stmt.executeQuery(“SELECT * FROM goods”);
%>

<jrun:query2xml query="rs1" />
```

7.5.2　Xslt 标记

用途：

Xslt 标记可以执行 xsl 文件，把 XML 格式的文档格式化输出。

语法 1：

```
<xslt ... >
xml input...
</xslt>
```

属性 1：

xsl：必选的属性。指定目标 xsl 文件的地址。

id：可选的属性。在下面的 JSP 程序中仍然可以凭借这个 id 号来引用这个 XML 文件的输出。

scope：可选的属性，类似于 JavaBeans 的 Scope 属性，缺省值是 page。

语法 2：

```
<xslt ... />
```

属性 2：

xml：可选的属性指定需要输出的 XML 文件的地址。

xsl：必选的属性。指定需要使用的目标 xsl 文件的地址。

id：可选的属性。在下面的 JSP 程序中仍然可以凭借这个 id 号来引用这个 XML 文件的输出。

scope：可选的属性，类似于 JavaBeans 的 Scope 属性，缺省值是 page。

TLD 文件：

```
<tag>
<name>xslt</name>
<tagclass>allaire.taglib.XsltTag</tagclass>
<teiclass>allaire.taglib.XsltTei</teiclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>xsl</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>id</name>
<required>false</required>
```

```
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>scope</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>xml</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%--
Example The xslt syntax allows JSP developers to markup the XML data in the JSP for
transformation，for example，output from query2xml could be used directly in xslt
without creating extra scripting variable. For those who do not prefer embedding XML
data into JSP，the empty tag syntax (URL to both XML and XSL) meets their
requirements.
--%>

<%@ taglib uri="jruntags" prefix="jrun" %>

<jrun:xslt xml='<%= new URL("http://localhost/article.xml") %>'    xsl="format.xsl"/>
<jrun:sql driver="Jdbc.Odbc.JdbcOdbcDriver" url="jdbc:odbc:test"
      username="sa" password=""    id="rs">
SELECT * FROM goods
</jrun:sql>
<jrun:xslt xsl="format2.xsl">
<jrun:query2xml query="page.rs"/>
</jrun:xslt>
```

7.6　其它标记

7.6.1　Form 标记

用途：

Form 标记扩展了 HTML 语言中的 Form 标记，Form 标记还可以包含其他的标记，例如 Input 标记，Select 标记等。

语法：

```
<form ... >
...
```

北京希望电子出版社

(optional) <input .../>|<select ...>...</select>
...
</form>

属性：

name：必要的属性，这个 Form 结构的名字。

action：可选的属性，和 HTML 标准中的定义一样，就是提交 Form 结构中所包含的数据的目标程序地址。

onSubmit:onSubmit 属性指定 JavaScript 脚本函数的名称，当 Submit 按钮被点击的时候，浏览器会自动调用这个 JavaScript 脚本程序，然后才把数据发送出去。

other HTML 4.0 attributes...：其他的 HTML 4.0 语法规范规定的 Form 标记的属性。

TLD 文件：

```
<tag>
<name>form</name>
<tagclass>allaire.taglib.FormTag</tagclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>name</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>action</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>onSubmit</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:form name="form1" action="form.jsp">
<input type="submit" value="submit">
</jrun:form>
```

7.6.2　Input 标记

用途：

Input 标记扩展了 HTML 标准的 Input 标记。Input 标记一般需要在 Form 标记中使用。

北京希望电子出版社

Input 标记支持 onClick 事件的验证。Input 标记支持 radio buttons、checkboxes、text boxes 等构件。

语法：

<input ... />

属性：

name：必需的属性。和 HTML 语法的定义一样。

type：可选的属性。和 HTML 语法的定义一样。可能是 radio、checkbox、text、file、hidden、password 等值。

value：可选的属性。和 HTML 语法的定义一样。

required：可选的属性。和 HTML 语法的定义一样。

onError：可选的属性。JavaScript 脚本函数的名字。当错误产生的时候，浏览器将会自动调用这个 JavaScript 脚本函数。

onValidate：可选的属性。JavaScript 脚本函数的名字。当文本域的内容发生改变的时候，浏览器将会自动调用这个 JavaScript 脚本函数。

other HTML 4.0 attributes...：可选的属性。其他的 HTML 4.0 语法规范规定的 Input 标记的属性。

TLD 文件：

```
<tag>
<name>input</name>
<tagclass>allaire.taglib.InputTag</tagclass>
<bodycontent>empty</bodycontent>
<info>Input Tag</info>
<attribute>
<name>name</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>type</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>value</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>required</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
```

```
</attribute>
<attribute>
<name>onError</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>onValidate</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>

<%-- Built-in phone format validation --%>
<jrun:form name="form1" action="input.jsp">
<jrun:input name="input1" type="phone" required="true" />
<input type="submit" value="validate">
</jrun:form>

<%-- Custom input field validation and error message --%>

<script language="javascript">
<!--
function customValidate(obj_value)
{
if (!(obj_value.toString() == "PROMOTION"))
{
return false;
}
return true;
}

function customErrorMsg(form_obj，input_obj，obj_value，error_msg)
{
alert("custom error message: Invalid Promotion Code: "+obj_value);
return false;
}
//-->
</script>

<jrun:form method="post" name="form1" action="form4.jsp">
```

北京希望电子出版社

```
<jrun:input name="t1" required="true" onValidate="customValidate"
onError="customErrorMsg" />
<jrun:input name="t2" type="password" required="true"/>
<input type="submit" value="Submit"/>
</jrun:form>
```

7.6.3　Select 标记

用途：

Select 标记扩展了 HTML 标准的 Select 标记。

语法：

<select ...>...</select>

属性：

name：必需的属性。和 HTML 语法的定义一样。

size：可选的属性。和 HTML 语法的定义一样。

hashtable：必需的属性。如果指定了这个属性，那么将会利用 Java Hashtable 对象中的数据自动生成 Select 列表。

query：必需的属性。query 属性和 hashtable 属性的作用差不多，这不过前者使用 java.util.Hashtable 对象中所包含的数据自动生成下拉列表，而 query 属性则是利用 java.sql.ResultSet 对象或者 javax.sql.RowSet 对象所包含的数据自动生成下拉列表。请参看下面的程序实例。

value：可选的属性。和 HTML 语法的定义一样。

display：可选的属性。和 HTML 语法的定义一样。

required：可选的属性。和 HTML 语法的定义一样。

onError：可选的属性。JavaScript 脚本函数的名字。当错误产生的时候，浏览器将会自动调用这个 JavaScript 脚本函数。

selected：可选的属性。和 HTML 语法的定义一样。

TLD 文件：

```
<tag>
<name>select</name>
<tagclass>allaire.taglib.SelectTag</tagclass>
<teiclass>allaire.taglib.SelectTei</teiclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>name</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>size</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
```

```
</attribute>
<attribute>
<name>hashtable</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>query</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>required</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>onError</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>value</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>display</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
<name>selected</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
<%@ page import=”java.sql.*” %>
<%@ page import=”java.io.*” %>
<jrun:param id="rs" type="allaire.taglib.QueryTable"/>
<jrun:sql id="rs" driver="Jdbc.Odbc.JdbcOdbcDriver" url="jdbc:odbc:test"
```

北京希望电子出版社

```
username=”sa” password=””>
SELECT * FROM goods
</jrun:sql>

<jrun:form name="form1" action="select.jsp">
<jrun:select name="box1" required="true" query="<%= rs %>"
value="ID" display="LastName">
</jrun:select>
<input type="submit" value="validate">
</jrun:form>
```

7.6.4　Param 标记

用途：

Param 标记主要用于定义程序段变量，这些变量一旦被定义，就可以在任意的 JSP 代码段中和任何的 JRun 标记体内使用，当然了，这有一个前提条件，就是必须在这个变量的作用范围内。Param 标记定义的变量不能够覆盖已经存在的程序变量。

注意：Param 标记是一个顶级标记，也就是说它不能够被别的 JRun 标记所包含。

语法：

<param ... />

属性：

id：必需的属性。Param 标记所定义的变量的名称。这个名称不要和任何现存的变量相冲突。

scope：可选的属性。scope 属性设定该变量的作用范围，十分类似于 JavaBeans 的 Scope 属性。可以有下面的 4 种取值：page、session、request、application。

type：可选的属性，指定该变量的数据类型。

default：可选的属性。设定该变量的初始值。

TLD 文件：

```
<tag>
<name>param</name>
<tagclass>allaire.taglib.ParamTag</tagclass>
<teiclass>allaire.taglib.ParamTei</teiclass>
<bodycontent>empty</bodycontent>
<attribute>
<name>id</name>
<required>true</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>scope</name>
<required>false</required>
```

```
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>type</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>default</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%--
Example The param tag must be a top-level tag so the scripting variable it generates is visible byall the other custom actions in the same page.
--%>

<%@ taglib uri="jruntags" uri="jrun" %>

<%-- Declares an Integer object and stores it in the session scope. --%>
<jrun:param id="x" type="java.lang.Integer" default="<%= new Intger(1) %>"
scope="session"/>
<%= session.getAttribute("x") %>

<jrun:param id="x" type="java.lang.Integer" default="<%= new Integer(1) %>"
scope="application"/>
<%= application.getAttribute("x") %>
<%
application.setAttribute("x"，new Integer(2));
%>

<%-- Declares a Date object and stores it in the page scope. --%>
<jrun:param id="y" type="java.util.Date" default="<%= new Date() %>" scope="page"/>
<%= y %>
```

7.6.5　ForEach 标记

用途：

ForEach 标记类似于 Visual Basic 语言中的 Foreach 语句，主要起着循环的作用。

语法：

```
<foreach ... >
```

北京希望电子出版社

...

</foreach>

属性：

item：可选的属性。代表循环过程中，中间变量的名称。每次从循环变量(group)中取出来的值都会赋给这个中间变量。

type：可选的属性。定义中间变量的数据类型。这样我们在 ForEach 标记体内就可以使用相应的方法对中间变量进行处理。

group：必选的属性。代表需要循环操作的对象。

TLD 文件：

```
<tag>
<name>foreach</name>
<tagclass>allaire.taglib.ForEachTag</tagclass>
<teiclass>allaire.taglib.ForEachTei</teiclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>item</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>type</name>
<required>false</required>
<rtexprvalue>false</rtexprvalue>
</attribute>
<attribute>
<name>group</name>
<required>true</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ page import="allaire.taglib.*" %>
<%@ taglib uri="jruntags" prefix="jrun" %>

<%-- Syntax 1 --%>
<jrun:foreach item="c" type="Cookie" group="<%= request.getCookies() %>">
<%= c.getName() %>
</jrun:foreach>

<%-- Syntax 2 --%>
<%-- java:comp/env/jdbc/dsn1 --%>
<jrun:sql datasrc="dsn1" id="rs">
```

北京希望电子出版社

```
SELECT * FROM goods
</jrun:sql>
<jrun:param id="rs" type="QueryTable"/>
<jrun:foreach group="<%= rs %>">
<%= rs.get("goodsname") %>
<%= rs.get("goodstype") %>
<%= rs.get("goodsprice") %>
</jrun:foreach>
```

7.6.6　If 标记

用途：

相当于 Java 语言中的 if 关键字。与 If 标记类似的还有 Switch 标记和 Case 标记等。

语法：

```
<if ...>
</if>
```

属性：

expr：一个合法的 Java 表达式，它的值必须是 true 或者 false(布尔值)。if 标记就凭借 expr 表达式的值，判断应不应该执行 if 标记体内的代码(无论是 JSP 代码还是 HTML 代码)。

TLD 文件：

```
<tag>
<name>if</name>
<tagclass>allaire.taglib.IfTag</tagclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>expr</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>
<jrun:if expr="<%= "fancy".equals(request.getParameter("username") %>">
the current user is fancy<br>
</jrun:if>
```

7.6.7　Switch 标记

用途：

Switch 标记类似于 Java 语言中的 switch 关键字，它起着判断条件，执行相应代码段的作用。Switch 标记一般需要和 Case 标记一起使用，在 Switch 标记体内，可以包含若干个 Case 标记块。

北京希望电子出版社

语法：

<switch>
one or more <case> tags...
</switch>

属性：

无。

TLD 文件：

```
<tag>
<name>switch</name>
<tagclass>allaire.taglib.SwitchTag</tagclass>
<bodycontent>JSP</bodycontent>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>

<jrun:switch>
<jrun:case expr="<%=2>1%>">
<%
out.println("2 is larger than 1。");
%>
</jrun:case>

<jrun:case expr="<%=2<1%>">
<%
out.println("2 is smaller than 1。");
%>
</jrun:case>
<jrun:case>
....
</jrun:case>
</jrun:switch>
```

7.6.8　Case 标记

用途：

Case 标记类似于 Java 语言中的 case 关键字，Case 标记必须和 Switch 标记配合使用，而不可以单独使用，否则会抛出错误。

语法：

<case ...>
...
</case>

北京希望电子出版社

属性：

expr：这是任意合法的 Java 表达式，它的值必须是 true 或者 false。如果它的值为 true，那么 Case 标记体内的 JSP 代码或者 HTML 代码将会被执行和输出。反之则不会被处理。expr 是可选的属性，缺省值为 true。请参考 Switch 标记的部分。

TLD 文件：

```
<tag>
<name>case</name>
<tagclass>allaire.taglib.CaseTag</tagclass>
<bodycontent>JSP</bodycontent>
<attribute>
<name>expr</name>
<required>false</required>
<rtexprvalue>true</rtexprvalue>
</attribute>
</tag>
```

用法示例：

```
<%@ taglib uri="jruntags" prefix="jrun" %>

<jrun:switch>
<jrun:case expr="<%=2>1%>">
<%
out.println("2 is larger than 1。");
%>
</jrun:case>
<jrun:case expr="<%=2<1%>">
<%
out.println("2 is smaller than 1。");
%>
</jrun:case>
<jrun:case>
....
</jrun:case>
</jrun:switch>
```

7.7　本 章 小 结

在本章中，我们详细讨论了 JRun Tag Library 标记库每一个标记的用法。读者如果需要使用这个标记库的强大功能，那么最好选用 JRun 服务器。但并不是说只有使用 JRun 服务器，才能够使用 JRun Tag Library。在下一章，我们将介绍 Jakarta 标记库的详细用法。

北京希望电子出版社

第 8 章　典型 Tag Library 介绍——Jakarta Tag Library



第三部分　JDBC 新技术及其在 JSP/Servlet 中的应用

第 9 章　JDBC 2.0/3.0 API 的新特性

在《JSP 深入编程》的第 10 章中，我们介绍了 JDBC 技术的基本原理和使用 JDBC API 访问数据库的方法，其实，那里介绍的只是 JDBC API 1.0 的知识，Java 程序员在使用了 JDBC API 1.0 一段时间后，一般都会感到比较的不便。例如 ResultSet 接口，用于处理数据库操作返回来的记录集，可是这个接口连返回记录集所包含行数的函数都没有，而且没有在记录集中定位的方法，这样使得 JSP/Servlet 程序员在编写分页显示的数据库程序时十分不便，两年过去了，Sun 公司 JavaSoft 部门在 JDBC API 1.0 的基础上，相继发布了 JDBC API 2.0、JDBC Option Pack(RowSet Package、CachedRowSet Package)、JDBC API 3.0 等新的数据库操作 API，使得 JDBC API 的功能越来越强大，也越来越方便使用了，2000 年 6 月，Sun 公司发布 JDO(Java Data Object) 0.8 规范，这个技术规范是参照 Microsoft 公司的 ADO(ActiveX Data Object)技术开发的，目的是使得利用 Java 语言开发数据库应用程序更为快捷，功能更强。用过 ASP 的读者一定会知道 ADO 技术，它用若干个 COM 组件封装了几乎所有的基于 ODBC 技术的数据库的访问操作，用起来十分方便，而且运行效率十分高。而且，这些内建的组件内部包含了很多有用的数据库函数，可以实现一些常用的功能，例如对分页显示的支持，ASP 程序员只需要设定几个简单的参数，很容易就实现了分页显示的功能，不用写多少行代码。JDO 技术和 ADO 技术十分相似，不过它比 ADO 的功能还要强大的多，至少它是与平台无关的，关于 JDO 技术的细节，在后面的第 10 章将会详细讨论。

本章的主要任务是向读者介绍 JDBC API 2.0/3.0 相对于 JDBC API 1.0 的新特性、新方法、新功能，以及如何使用这些先进的技术开发 JSP/Servlet 程序，事实上，Java 程序员不仅可以将 JDBC API 2.0/3.0 用于开发 Java 应用程序(Application)，也可以用于开发 JSP/Servlet 等类型的程序。本章所列出的代码段，除非特别声明，否则均可以用于 Application/JSP/Servlet 等类型的程序。

本章需要掌握的主要内容有：

- 全新的记录集接口(ResultSet 接口)。
- 如何处理 BLOB、CLOB 类型的数据(Blob、Clob 接口)。
- 如何处理新的 SQL 数据类型。
- 如何处理自定义 SQL 数据类型。

- JDBC 3.0 API 简介。

9.1　JDBC API 2.0 的新特性

与 JDBC API 1.0 相比，JDBC API 2.0 在以下的几个方面做了比较大的改进：

- 修改了记录集接口(ResultSet 接口)的方法，使它支持可以滚动的记录集，即数据库游标可以在返回的记录集对象中自由地向前或向后滚动，或者定位到某个特殊的行。例用 ResultSet 接口中定义的新方法，JSP/Servlet 程序员可以用 Java 语言来更新记录集，比如插入记录，更新某行的数据，而不是靠执行 SQL 语言，这样就大大方便了程序员的开发工作。
- 新的 SQL 语句接口(Statement 接口)支持批操作，就是 Java 应用程序可以向数据库服务器发送几个 SQL 语句，但是数据库引擎并不立即执行它，而是把它们加入到一个块(Batch)中，到了最后才执行这个 SQL 语句块(Batch)，这个功能大大扩展了编程的灵活性，为编制出功能更强大的、更灵活的数据库程序提供了可能。
- 支持最新的 SQL3 数据类型，特别是对 BLOB、CLOB 等类型的数据提供了很好的支持。
- 允许程序员自定义数据类型，并提供了若干个接口以支持存取程序员自定义的数据类型。如 SQLData、SQLInput、SQLOutput 等接口。

9.2　JDBC API 2.0 简介

9.2.1　新的记录集接口(ResultSet 接口)

在 JDBC API 2.0 中，ResultSet 接口有了很大的变化，增加了很多行操作、行定位的新方法，功能也强大许多。最主要的变化有以下几方面：

1．新定义了若干个常数

这些常数用于指定 ResultSet 的类型，游标移动的方向等性质，如下所示：

```
public static final int FETCH_FORWARD;
public static final int FETCH_REVERSE;
public static final int FETCH_UNKNOWN;
public static final int TYPE_FORWARD_ONLY;
public static final int TYPE_SCROLL_INSENSITIVE;
public static final int TYPE_SCROLL_SENSITIVE;
public static final int CONCUR_READ_ONLY;
public static final int CONCUR_UPDATABLE;
```

FETCH_FORWORD：该常数的作用是指定处理记录集中行的顺序是由前到后，即从第一行开始处理，一直到最后一行。

FETCH_REVERSE：该常数的作用是指定处理记录集中行的顺序是由后到前，即从最后一行开始处理，一直到第一行。

FETCH_UNKNOWN：该常数的作用是不指定处理记录集中行的顺序，由 JDBC 驱动

程序和数据库系统决定。

TYPE_FORWARD_ONLY：该常数的作用是指定数据库游标的移动方向是向前，不允许向后移动，即只能使用 ResultSet 接口的 next()方法，而不能使用 previous()方法，否则会产生错误。

TYPE_SCROLL_INSENSITIVE：该常数的作用是指定数据库游标可以在记录集中前后移动，并且当前数据库用户获取的记录集对其他用户的操作不敏感，就是说，当前用户正在浏览记录集中的数据，与此同时，其他用户更新了数据库中的数据，但是当前用户所获取的记录集中的数据不会受到任何影响。

TYPE_SCROLL_SENSITIVE：该常数的作用是指定数据库游标可以在记录集中前后移动，并且当前数据库用户获取的记录集对其他用户的操作敏感，就是说，当前用户正在浏览记录集，但是其它用户的操作使数据库中的数据发生了变化，当前用户所获取的记录集中的数据也会同步发生变化，这样有可能会导致非常严重的错误产生，建议慎重使用该常数。

CONCUR_READ_ONLY：该常数的作用是指定当前记录集的协作方式(concurrency mode)为只读，一旦使用了这个常数，那么用户就不可以更新记录集中的数据。

CONCUR_UPDATABLE：该常数的作用是指定当前记录集的协作方式(concurrency mode)为可以更新，一旦使用了这个常数，那么用户就可以使用 updateXXX()等方法更新记录集中的数据。

在 JSP/Servlet 程序中如何使用这些预定义的常数呢?这是读者很关心的问题，在 9.2.2 节中将有介绍。

2．ResultSet 接口提供了一整套的定位方法

这些可以在记录集中定位到任意一行，具体有：

public boolean absolute(int row);　该方法的作用是将记录集中的某一行设定为当前行，亦即将数据库游标移动到指定的行，参数 row 指定了目标行的行号，这是绝对的行号，由记录集的第一行开始计算，不是相对的行号。

public boolean relative(int rows);　该方法的作用也是将记录集中的某一行设定为当前行，但是它的参数 rows 表示目标行相对于当前行的行号，例如当前行是第 3 行，现在需要移动到第 5 行去，既可以使用 absolute()方法，也可以使用 relative()方法，代码如下。

例：

```
rs.absolute(5);
```

或者

```
rs.relative(2);
```

其中 rs 代表 ResultSet 接口的实例对象。

又如当前行是第 5 行，需要移动到第 3 行去，代码如下。

例：

```
rs.absolute(3);
```

或者

```
rs.relative(-2);
```

其中 rs 代表 ResultSet 接口的实例对象。

读者需要注意的问题是，传递给 relative()方法的参数，如果是正数，那么数据库游标向前移动，如果是负数，那么数据库游标向后移动。

注意：在本章中所说的数据库游标向前移动是指向行号增大的方向移动，向后移动是指向行号减少的方向移动。

public boolean first(); 该方法的作用是将当前行定位到数据库记录集的第一行。

public boolean last(); 该方法的作用刚好和 first()方法相反，是将当前行定位到数据库记录集的最后一行。

public boolean isFirst(); 该方法的作用是检查当前行是否记录集的第一行，如果是，返回 true，否则，返回 false。

public boolean isLast(); 该方法的作用是检查当前行是否记录集的最后一行，如果是，返回 true，否则，返回 false。

public void afterLast(); 该方法的作用是将数据库游标移到记录集的最后，位于记录集最后一行的后面，如果该记录集不包含任何的行，该方法不产生作用。

public void beforeFirst(); 该方法的作用是将数据库游标移到记录集的最前面，位于记录集第一行的前面，如果记录集不包含任何的行，该方法不产生作用。

public boolean isAfterLast(); 该方法检查数据库游标是否处于记录集的最后面，如果是，返回 true，否则，返回 false。

public boolean isBeforeFirst(); 该方法检查数据库游标是否处于记录集的最前面，如果是，返回 true，否则，返回 false。

public boolean next(); 该方法的作用是将数据库游标向前移动一位，使得下一行成为当前行，当刚刚打开记录集对象时，数据库游标的位置在记录集的最前面，第一次使用 next()方法，将会使数据库游标定位到记录集的第一行，第二次使用 next()方法，将会使数据库游标定位到记录集的第二行，以此类推。

注意：如果在当前行打开了一个输入流(Input Stream)，那么再次使用 next()方法时，将会自动关闭该输入流。

public boolean previous(); 该方法的作用是将数据库游标向后移动一位，使得上一行成为当前行。

下面以程序清单 9.1(dbScroll.jsp)为例，演示如何使用 ResultSet 接口的方法在记录集中定位到特定的行。

程序清单 9.1

```
<%--
File Name:Application.java
Author: fancy
Date:2001.2.17
Note:a java program use jdbc technology to visit the database server
--%>

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

```
<%
String url = "jdbc:weblogic:mssqlserver4:rainbow:1433";
Connection con;
Statement stmt;
try
{
    Class.forName("weblogic.jdbc.mssqlserver4.Driver ");
}
catch(java.lang.ClassNotFoundException e)
{
    out.print("ClassNotFoundException: ");
    out.println(e.getMessage());
}

try
{
    con = DriverManager.getConnection(url，"sa"，"");
    stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE，
        ResultSet.CONCUR_READ_ONLY);
    ResultSet srs = stmt.executeQuery("USE fancy SELECT * FROM goods");
    srs.absolute(4);
    int rowNum = srs.getRow(); // rowNum should be 4
    out.println("rowNum should be 4 " + rowNum);
    srs.relative(-3);
    rowNum = srs.getRow(); // rowNum should be 1
    out.println("rowNum should be 1 " + rowNum);
    srs.relative(2);
    rowNum = srs.getRow(); // rowNum should be 3
    out.println("rowNum should be 3 " + rowNum);
    srs.absolute(1);
    out.println("after last? " + srs.isAfterLast() );
    if (!srs.isAfterLast())
    {
        String name = srs.getString("goodsname");
        float price = srs.getFloat("price");
        out.println(name + "         " + price);
    }
    srs.afterLast();
    while (srs.previous())
    {
        String name = srs.getString("goodsname");
        float price = srs.getFloat("price");
```

北京希望电子出版社

```
                out.println(name + "       " + price);
            }
            srs.close();
            stmt.close();
            con.close();
        }
        catch(BatchUpdateException b)
        {
            out.println("-----BatchUpdateException-----");
            out.println("SQLState:" + b.getSQLState());
            out.println("Message:" + b.getMessage());
            out.println("Vendor:" + b.getErrorCode());
            out.print("Update counts:    ");
            int [] updateCounts = b.getUpdateCounts();
            for (int i = 0; i < updateCounts.length; i++)
            {
                out.print(updateCounts[i] + "      ");
            }
            out.println("");
        }
        catch(SQLException ex)
        {
            out.println("-----SQLException-----");
            out.println("SQLState:" + ex.getSQLState());
            out.println("Message:" +ex.getMessage());
            out.println("Vendor:" + ex.getErrorCode());
        }
        %>
```

程序清单 9.1(dbScroll.jsp)的运行环境是 BEA WebLogic Server 5.1，数据库服务器是 Microsoft SQL Server 7.0 Desktop Edition，主机名为 rainbow，数据库服务器侦听端口为 1433，数据库的名称是 fancy，创建 fancy 数据库的 SQL 文件是 db.sql，请参考本书附录 4，数据库的 JDBC 驱动程序采用 WebLogic 的 mssqlserver4，当然，读者也可以使用别的 JDBC 驱动程序，但是这里千万不能使用 JDBC-ODBC 桥驱动程序，否则会出错。

dbScroll.jsp 的流程十分简单，在这里就不详细讨论了，但是请读者特别留心以下的几行代码：

```
String url="jdbc:weblogic:mssqlserver4:rainbow:1433";
```

该行指定数据源的位置，如果读者不是使用 WebLogic 的 mssqlserver4　JDBC 驱动程序，这行代码需要稍作修改，至于如何修改，请读者参考相关的文档。

```
Class.forName("weblogic.jdbc.mssqlserver4.Driver");
```

该行代码载入 WebLogic 的 JDBC 驱动程序类，如果读者不是使用 WebLogic 的 JDBC 驱动程序，那么这行代码也需要修改一下。

```
stmt =con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
```

ResultSet.CONCUR_READ_ONLY);

该行代码十分重要，它创建了一个 SQL 语句接口(Statement 接口)的实例对象。并且指定：凡是由该对象执行 SQL 语句所返回的记录集(ResultSet)都可以前后移动数据库游标，而且记录集中的数据不可以修改。如果调用无参数的 createStatement()方法创建 SQL 语句接口的实例对象，那么 stmt 对象执行 SQL 语句所返回的记录集仅可以向前移动数据库游标，如果这时调用 ResultSet 接口的 previous()、absolute()、relative()等方法就会出错。

3．ResultSet 接口添加了对行操作的支持

使用 JDBC API 2.0，不仅可以任意将数据库游标定位到记录集中的特定行，而且还可以使用 ResultSet 接口新定义的一套方法更新当前行的数据，在以前，如果 Java 程序员希望更新记录集中某行的数据，必须发送 SQL 语句给数据库，程序员需要在 Java 代码中嵌入冗长的 SQL 语句，用以执行 UPDATE、DELETE、INSERT 等数据库操作，但是，当 JDBC API 2.0 出现时，一切就都改变了，程序员已经可以部分抛开 SQL 语言，享受 Java 编程的乐趣了。ResultSet 接口中新添加的部分方法如下所示：

public boolean rowDeleted();　如果当前记录集的某行被删除了，那么记录集中将会留出一个空位，调用 rowDeleted()方法，如果探测到空位的存在，那么就返回 true，如果没有探测到空位的存在，就返回 false 值。

public boolean rowInserted();　如果当前记录集中插入了一个新行，该方法将返回 true，否则返回 false。

public boolean rowUpdated();　如果当前记录集的当前行的数据被更新，该方法返回 true，否则返回 false。

public void insertRow();　该方法将执行插入一个新行到当前记录集的操作。

public void updateRow();　该方法将更新当前记录集当前行的数据。

public void deleteRow();　该方法将删除当前记录集的当前行。

public void updateString(int columnIndex，String x);　该方法更新当前记录集当前行某列的值，该列的数据类型是 String(指 Java 数据类型是 String，与之对应的 JDBC 数据类型是 VARCHAR 或 NVARCHAR 等数据类型)，该方法的参数 columnIndex 指定所要更新的列的列索引，第一列的列索引是 1，以此类推，第二个参数 x 代表新的值。这个方法并不执行数据库操作，需要执行 insertRow()方法或者 updateRow()方法以后，记录集和数据库中的数据才能够真正更新。

public void updateString(String columnName，String x);　该方法和上面介绍的同名方法差不多，不过该方法的第一个参数是 columnName，代表需要更新的列的列名，而不是 columnIndex。

ResultSet 接口中还定义了很多个 updateXXX()方法，都和上面的两个方法相类似，由于篇幅的原因，在这里就不详细描述了，对此感兴趣的读者，可以参考相关的文献。

往数据库当前记录集插入新行的操作流程如下：

（1）调用 moveToInsertRow()方法。

（2）调用 updateXXX()方法，指定插入行各列的值。

（3）调用 insertRow()方法，往数据库中插入新的行。

在程序清单 9.2(insertRow.jsp)中，演示了如何应用上面的方法往数据库中插入新的行

(亦即新的记录)。

程序清单 9.2

```
<%--
File Name:insertRow.jsp
Author:fancy
Date:2001.2.17
Note:to insert some rows ro the current database
--%>
<%@page import="java.sql.*" %>
<%
String url = " jdbc:weblogic:mssqlserver4:rainbow:1433";
Connection con;
Statement stmt;
try
{
    Class.forName("weblogic.jdbc.mssqlserver4.Driver ");
}
catch(java.lang.ClassNotFoundException e)
{
    out.println("ClassNotFoundException: ");
    out.println(e.getMessage());
}

try
{
    con=DriverManager.getConnection(url，"sa"，"");
    stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE，
        ResultSet.CONCUR_UPDATABLE);
    ResultSet uprs = stmt.executeQuery("USE fancy SELECT * FROM tbuser");
    uprs.moveToInsertRow();
    uprs.updateString("username"，"peking");
    uprs.updateString(2，"peking");

    uprs.insertRow();

    uprs.updateString(1，"lijishan");
    uprs.updateString("password"，"lijishan");

    uprs.insertRow();
    uprs.beforeFirst();
    out.println("Table tbuser after insertion:");
    while (uprs.next())
    {
```

```
            String name = uprs.getString("username");
            String pass = uprs.getString("password");
            out.println("username:"+name+"<br>");
            out.println("password:"+pass+"<br>");
        }
        uprs.close();
        stmt.close();
        con.close();
    }
    catch(SQLException ex)
    {
        out.println("SQLException:" + ex.getMessage());
    }
    %>
```

程序清单 9.2 往 fancy 数据库的 tbuser 表插入了两行，亦即两个记录，然后执行数据库查询，检查 INSERT 操作对数据库的影响。insertRow.jsp 程序应用了上面讲述的方法，比较简单，在这里就不重复介绍程序中所用到的各个方法了。读者如果想执行这个 JSP 程序，请将用黑体标出的代码行稍作修改，配置好 JSP 服务器、数据库服务器、JDBC 数据库驱动程序即可。

更新数据库中某个记录的值(某行的值)的方法是：

（1）定位到需要修改的行(使用 absolute()、relative()等方法定位)。

（2）使用相应 updateXXX()方法设定某行某列的新值，XXX 所代表的 Java 数据类型必须可以映射为某列的 JDBC 数据类型，如果希望 rollback 该项操作，请再调用 updateRow()方法以前使用 cancelRowUpdates()方法，这个方法可以将某行某列的值复原。

（3）使用 updateRow()方法，完成 UPDATE 的操作。

删除记录集中某行(亦即删除某个记录)的方法：

（1）定位到需要修改的行(使用 absolute()、relative()等方法定位)。

（2）使用 deleteRow()方法。

4．新的 ResultSet 接口添加了对 SQL3 数据类型的支持

SQL3 技术规范中添加了若干个新的数据类型，如 REF、ARRAY 等，ResultSet 接口扩充了 getXXX()方法，添加了获取这些数据类型的数据的 getXXX()方法，如 getArray()、getBlob()、getBigDecimal()、getClob()、getRef()等方法，这些方法既可以接收列索引为参数，也可以接收列名(字段名)为参数，这些方法分别返回对应的 Java 对象实例，如 Clob、Array(JDBC Array)、Blob、BigDecimal、Ref 等，使用起来十分方便，至于这些方法的用法，在下面还会涉及，这里就不再赘述了。

5．获取记录集行数的方法

（1）首先使用 last()方法，将数据库游标定位到记录集的最后一行。

（2）使用 getRow()方法，返回记录集最后一行的行索引。该索引就等于记录集所包含记录的个数，也就是记录集的行数。getRow()方法是在 JDBC API 2.0 中才定义的，在 JDBC API 1.0 中没有这个方法。

9.2.2　新的 SQL 语句接口(Statement 接口)

在 JDBC API 2.0 中，SQL 语句接口(Statement 接口)也有了很大的改进，功能更加强大，PreparedStatement 接口和 CallableStatement 接口都继承了 Statement 接口，因此本小节也介绍这两个接口相对于 JDBC API 1.0 的改进之处。因为上述的三个接口都由 Connection 接口的方法创建，所以本小节也顺便提一提 Connection 接口。

1．Statement 接口、CallableStatement 接口、PreparesStatement 接口的创建

这三个接口分别由 Connection 接口的 createStatement()、prepareStatement()、prepareCall()等方法创建，这几个方法的定义如下：

```
public Statement createStatement();
public Statement createStatement(int  resultSetType，int  resultSetConcurrency);
    public CallableStatement prepareCall(String sql);
public CallableStatement prepareCall(String sql，int  resultSetType，
    int  resultSetConcurrency);
    public PreparedStatement prepareStatement(String sql);
    public PreparedStatement prepareStatement(String sql，int  resultSetType，
    int  resultSetConcurrency);
```

上面列出的方法中，参数 sql 代表需要执行的 SQL 语句 ，这些 SQL 语句不是完整的 SQL 语句，一般带有 IN/OUT/INOUT 参数，参数 resultSetType 代表该方法创建的 SQL 语句接口执行 SQL 语句所返回的 ResultSet 的类型，例如：是否允许数据库游标前后移动，是否对其他用户的数据库更新操作敏感等，它们都是一些整型的常数，在 ResultSet 接口中定义了，读者可以参考 9.2.1 小节的相关内容，参数 resultSetConcurrency 代表该方法创建的 SQL 语句接口执行 SQL 语句所返回的 ResultSet 的协同模式，如允许更新记录集的数据 、或者仅仅只读，不能更新等，它们也是一些整型的常数，在 ResultSet 接口中定义了，读者可以参考 9.2.1 小节的相关内容。下面的代码段是创建 Statement 接口对象的示例(数据库连接代码已经省略了，con 是 Connection 接口的实例对象)。

例：

```
stmt=con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE，
ResultSet.CONCUR_UPDATABLE);
```

上面的代码创建了一个 SQL 语句接口(Statement)的实例对象，该实例对象允许它执行 SQL 语句所返回的记录集中的数据库游标前后移动，允许更新记录集中的数据。

2．支持批操作

Statement 接口、PreparedStatement 接口、CallableStatement 接口都支持数据库批操作，就是将若干个 SQL 语句添加到一个 SQL 语句块(Batch)中，一并发送到数据库服务器去，数据库引擎执行完 SQL 语句块中的语句后，会将所有的结果一并返回，这种功能特别适用于大批量的数据库 INSERT 操作。为了实现这样的功能，必须用到的 Statement 接口的方法如下所示：

public void addBatch(String sql);　该方法用于将 SQL 语句添加到 SQL 语句块中。

public void clearBatch();　该方法用于将 SQL 语句块中的所有 SQL 语句全部删除。

public int[] executeBatch();　该方法用于将 SQL 语句块发送到数据库服务器去，并执行它，返回的结果是一个整型数组，数组中的元素是数据库服务器执行 SQL 语句块中 SQL 语句所返回的更新计数，SQL 语句块中含有多少个 SQL 语句，返回的整行数组中就含有多少个元素。

使用 JDBC API 执行数据库批操作的方法是：

（1）创建 Statement 接口的实例对象。

（2）调用 addBatch()方法，往 SQL 语句块中添加若干个 SQL 语句。

（3）使用 executeBatch()方法，完成数据库批操作。

程序清单 9.3(batch.jsp)演示了如何运用 Statement 接口的批操作功能往数据库中插入 4 条新记录。

程序清单 9.3

```
<%--
File Name:batch.jsp
Author:fancy
Date:2001.2.17
Note:insert some records to the database using sql batch .
--%>

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

<%
ResultSet rs = null;
PreparedStatement ps = null;

String url = " jdbc:weblogic:mssqlserver4:rainbow:1433";
Connection con;
Statement stmt;
try
{
    Class.forName("weblogic.jdbc.mssqlserver4.Driver");
}
catch(java.lang.ClassNotFoundException e)
{
    out.println("ClassNotFoundException: ");
    out.printlnln(e.getMessage());
}

try
{
    con = DriverManager.getConnection(url，"sa"，"");
    con.setAutoCommit(false);
    stmt = con.createStatement();
```

北京希望电子出版社

```
        stmt.addBatch("USE fancy INSERT INTO tbuser VALUES('tttt'，'tttt') ");
        stmt.addBatch("USE fancy INSERT INTO tbuser VALUES('ppp'，'ppp') ");
        stmt.addBatch("USE fancy INSERT INTO tbuser VALUES('javafancy'，'javafancy') );
        stmt.addBatch("USE fancy INSERT INTO tbuser VALUES('hh'，'hh') ");

        int [] updateCounts = stmt.executeBatch();
        con.commit();
        con.setAutoCommit(true);
        ResultSet uprs = stmt.executeQuery("USE fancy SELECT * FROM tbuser");
        out.println("Table COFFEES after insertion:");
        while (uprs.next())
        {
            String name = uprs.getString("username");
            String pass = uprs.getString("password");
            int id = uprs.getInt("id");
            out.println("name:"+name+"      "+" pass:"+pass+"    id:"+id+"<br>");
        }

        uprs.close();
        stmt.close();
        con.close();
    }
    catch(BatchUpdateException b)
    {
        out.printlnln("-----BatchUpdateException-----");
        out.printlnln("SQLState:    " + b.getSQLState());
        out.printlnln("Message:    " + b.getMessage());
        out.printlnln("Vendor:    " + b.getErrorCode());
        out.println("Update counts:    ");
        int [] updateCounts = b.getUpdateCounts();
        for (int i = 0; i < updateCounts.length; i++)
        {
            out.println(updateCounts[i] + "      ");
        }
        out.printlnln("");
    }
    catch(SQLException ex)
    {
        out.printlnln("-----SQLException-----");
        out.printlnln("SQLState:" + ex.getSQLState());
        out.printlnln("Message:" + ex.getMessage());
        out.printlnln("Vendor:" + ex.getErrorCode());
    }
```

北京希望电子出版社

```
%>
```

注意：读者若想运行程序清单 9.3，请根据实际情况，先修改用黑体加粗的代码行。

上面介绍的执行数据库批操作的方法仅仅适用于 Statement 接口，不适用于 PreparedStatement 接口、CallableStatement 接口，后面两个接口定义了新的 addBatch()方法，该方法不需要任何参数，但是在 Statement 接口中定义的 addBatch()方法却需要参数，参数是一个 SQL 语句，数据类型是 String。

在 PreparedStatement 接口、CallableStatement 接口中实现数据库批操作的方法是：

（1）创建 PreparedStatement 接口或者 CallableStatement 接口的实例对象。

（2）使用 PreparedStatement 接口中定义的 setXXX()方法设定 SQL 语句(该 SQL 语句是在创建 PreparedStatement 接口或者 CallableStatement 接口的实例对象时初始化的)的 IN/OUT/INOUT 参数的值(CallableStatement 接口并没有定义任何 setXXX()方法，它的 setXXX()方法全部继承自 PreparedStatement 接口)。

（3）使用 executeBatch()方法，该方法在 Statement 接口中定义，不过 PreparedStatement 接口和 CallableStatement 接口都继承了这个方法。

程序清单 9.4(batch2.jsp)演示了如何使用 PreparedStatement 接口的方法完成数据库批操作，往数据库中插入四条新记录。

程序清单 9.4

```
<%--
File Name:batch2.jsp
Author:fancy
Date:2001.2.17
Note:inser some records to the database using sql batch .
--%>

<%@page import="java.io.*" %>
<%@page import="java.sql.*" %>
<%@page import="java.net.*" %>
<%@page import="java.util.*" %>
<%
String url = " jdbc:weblogic:mssqlserver4:rainbow:1433";
Class.forName("weblogic.jdbc.mssqlserver4.Driver ");

// Here sa is the username
Properties prop = new Properties ();
prop.put("user"，"sa");
prop.put("password"，"");
prop.put("serverVersion"，"Sql7.0");

Connection ctn= DriverManager.getConnection(url，prop);
```

```
//Creates a PreparedStatement object with single parameter
PreparedStatement prepStmt = ctn.prepareStatement("USE fancy INSERT INTO goods
    (goodsname，goodstype，comment，price，priceoff)+□ □
    VALUES(□?□，□?□，□?□，3000，0.8) ");

prepStmt.setString(1，“JSP精通”);
prepStmt.setString(2，“书籍”);
prepStmt.setString(3，“介绍JSP技术的高级教程”);
prepStmt.addBatch();

prepStmt.setString(1，“结构化学基础”);
prepStmt.setString(2，“书籍”);
prepStmt.setString(3，“北京大学化学系本科教材，免费赠送”);
prepStmt.addBatch();

prepStmt.setString(1，“万水青山踏遍”);
prepStmt.setString(2，“兵器”);
prepStmt.setString(3，“曾是白云城主的配剑，唯有此剑，方能使出天外飞仙的剑招”);
prepStmt.addBatch();

prepStmt.setString(1，“小楼一夜听春雨”);
prepStmt.setString(2，“兵器”);
prepStmt.setString(3，“一把弯刀，又名圆月弯刀，曾经是青青所用之物，后归丁鹏所有，
和万水青山踏遍齐名”);
prepStmt.addBatch();

prepStmt.executeBatch();

prepStmt.close();
ctn.close();
%>
```

注意：读者若想运行程序清单 9.5，请根据实际情况，先修改用黑体加粗的代码行。可能会产生 JSP 的中文问题，如果没有办法解决，读者可以将插入数据库的数据全部变为英文。

9.2.3　处理 BLOB、CLOB 类型的数据(Blob、Clob 接口)

BLOB、CLOB 是 SQL3 标准支持的新数据类型，主要用于保存大型超长的数据，如图片、视频、CD 等。Oracle 等数据库系统支持 BLOB、CLOB 数据类型。在 JDBC API 1.0 中，不支持直接存取 BLOB、CLOB 等类型的数据，必须通过输入/输出流来操作，这样做十分不方便。在 JDBC API 2.0 中，新定义了 Blob 接口、Clob 接口，对这两种类型数据的操作大大简化了。下面就来介绍这两个接口，以及如何使用这两个接口的方法来操作

BLOB、CLOB 类型的数据。

如何获取和设定 BLOB、CLOB 类型的数据

ResultSet、CallableStatement、PreparedStatement 等接口都定义了 getBlob()方法、getClob()方法、setBlob()方法、setClob()方法，用以获取或者设定 BLOB、CLOB 类型的数据，具体方法的定义，请读者参考相应的文档，这里就不再介绍了。

Blob 接口的方法

在 Blob 接口中，定义了下面的方法：

public long length();　该方法可以获取 Blob 数据的长度。

public byte[] getBytes(long pos，int length);　该方法可以从 Blob 数据中获取其中的某一段，将其赋给一个 byte 数组，参数 pos 是开始截取数据的位置，参数 length 是截取数据的长度。

public InputStream getBinaryStream();　该方法从 Blob 数据中获取一个输入流(InputStream)。

public long position(byte[] pattern，long start);　该方法获取特定字节在 Blob 数据中的位置，参数 pattern 是查找的目标字节，参数 start 指的是开始查找的位置。

public long position(Blob pattern，long start);　该方法可以获取特定 Blob 类型的数据在当前 Blob 数据中的开始位置，参数 pattern 代表需要查找的 Blob 数据，参数 start 代表开始查找匹配的位置。

下面的两个 JSP 代码段演示了如何获取 Blob 类型的数据，并将其输出(数据库连接代码段已经省略了，其中，rs 是 ResultSet 接口的实例对象)。

例：

```
<%
rs.absolute(4);
Blob blob=rs.getBlob( " image " );
java.io.InputStream in = blob.getBinaryStream();
byte b;
while ((in.read()) > -1)
{
      b = in.read();
      out.println(b);
}
%>
```

例：

```
<%
rs.absolute(4);
Blob blob=rs.getBlob( " image " );
long len = blob.length();
byte [] data = blob.getBytes(1，len);
for (int i = 0; i < len; i++)
```

```
{
    byte b = data[i];
    out.println(b);
}
%>
```

Clob 接口的方法

在 Clob 接口中，定义了以下的方法：

public long length(); 该方法可以获取 Clob 类型数据的长度。

public String getSubString(long pos，int length); 该方法获取 Clob 类型数据的一部分，并将结果赋给一个字符串，参数 pos 就是开始截取数据的位置，参数 length 代表需要截取数据的长度。该方法和 String 类的 substring()方法差不多。

public Reader getCharacterStream(); 该方法从 Clob 对象中返回一个 Reader 类的实例对象。该方法可以用于读取 Clob 对象所包含的数据。

public InputStream getAsciiStream(); 该方法从 Clob 对象中返回一个输入流(InputStream)。该方法也可以用于读取 Clob 对象的数据，getAsciiStream()方法和 getCharacterStream()方法的区别在于，前者是以 ASCII 码输入流的方式读取数据，后者是以字符流的方式读取数据。

public long position(String searchstr，long start); 该方法从 Clob 类型的数据中获取特定字符串出现的位置，参数 seacherstr 就是目标字符串，参数 start 代表开始检索的位置。

public long position(Clob searchstr，long start); 该方法的作用是从 Clob 类型的数据中获取另一个 Clob 类型数据出现的位置，其中，参数 searchstr 代表需要匹配的目标 Clob 对象，参数 start 代表开始检索的位置。

下面的 JSP 代码段演示了如何将一个 Clob 类型的数据插入到数据库中去(数据库连接代码已经省略了，rs 代表 ResultSet 接口的实例对象，con 是 Connection 接口的实例对象)。

例：

```
<%
Clob notes = rs.getClob("NOTES");
PreparedStatement pstmt = con.prepareStatement("UPDATE MARKETS SET COMMENTS = ?
    WHERE SALES < 1000000");
pstmt.setClob(1，notes);
pstmt.executeUpdate();
%>
```

BLOB、CLOB 数据类型的区别

BLOB、CLOB 数据类型虽然都可以存储大量超长的数据，但是两者是有区别的。BLOB 其实是 Binary Large Object 的缩写，BLOB 类型的数据以二进制的格式保存于数据库中，特别适用于保存程序文件、图片、视频文件、音频文件等，CLOB 是 Character Large Object 的缩写，CLOB 类型的数据以 Character 的格式保存于数据库中，比较适合于保存比较长的文本文件，当然，并非一定要如此，程序员必须根据实际条件来决定采用哪种数据类型。

9.2.4　处理新的 SQL 数据类型(ARRAY、REF)

在新的 SQL92/99(SQL3)规范中，添加了若干新的数据类型，除了上面提到的 BLOB、CLOB 数据类型外，还有 REF、Structured Types、ARRAY 等复杂数据类型。这些新的数据类型给数据库开发者以很大的自由度，使得他们可以设计十分复杂而且有效的数据库结构，但是这就给 Java 语言程序员出了一个大难题，即如何访问这些使用新数据类型的数据呢?在 JDBC API 2.0 出现以前，Java 程序员几乎可以说是束手无策，因为 JDBC API 1.0 对新的 JDBC 数据类型的支持几乎为零，JDBC API 2.0 出现以后，一切就都不同了。

本小节介绍如何使用 JDBC API 2.0 处理 ARRAY、REF 等复杂数据类型。

ARRAY 数据类型

数据类型 ARRAY 主要用于保存一些类似于数组结构的数据，例如，创建一个公司(company)数据库，假设该数据库有两个字段，一个字段是公司的名称(name)，name 的数据类型可以是普通的 VARCHAR 类型，另一个字段是该公司所有雇员的名字(employee)，employee 字段就需要用到 ARRAY 数据类型，它把所有的雇员的名称都保存在一个类似于数组的结构中去。那么如何处理 ARRAY 类型的字段数据呢?

（1）ResultSet 接口、CallableResultSet 接口定义了 getArray()方法可以获取 ARRAY 数据类型的数据，返回值是一个 java.sql.Array 接口的实例对象，请看下面的代码段：

例：

```
Array a=rs.getArray(1);
```

其中，rs 是 ResultSet 接口的实例对象。

getArray()方法返回的 Array 接口的实例对象仅仅包含了对原 ARRAY 类型数据的一个逻辑指针(logical pointer)，并不包含任何实质的数据。为了真正获得源数据，必须使用 Array 接口的 getArray()方法或者 getResultSet()方法。Array 接口的 getArray()方法不同于 ResultSet 接口和 CallableStatement 接口的 getArray()方法，前者的作用是根据 Array 接口实例对象所包含的逻辑指针，将源数据取回，并转化为一个 Java 数组(Java Array)。Java 程序员可以使用 Java 语言对该数组进行处理。同样，Array 接口的 getResultSet()方法和 Statement 接口的 getResultSet()方法也有不同，前者的作用是根据 Array 接口实例对象所包含的逻辑指针，将源数据取回，并转化为一个 ResultSet 接口的实例对象，后者的作用也是获取 ResultSet 接口的实例对象，不过这个记录集对象是通过执行 SQL 语句而来的，并不是由某个数据结构转换而来的。

Array 接口的 getArray()方法有可选参数，如 index，count，因为 ARRAY 数据类型的数据的保存方式类似于数组结构，因此必然有元素索引，index 参数指定从某个索引开始获取源数据的元素，参数 count 指定需要获取元素的个数。Array 接口的 getResultSet()方法的情况也差不多，详细的方法说明，读者可以参考相关的文档。

（2）存储 ARRAY 类型的数据。

PreparedStatement 接口的 setArray()方法和 setObject()方法可以将一个 ARRAY 类型的数据作为 IN 参数传递给 PreparedStatement 对象。如下面的 JSP 代码段所示(数据库连接代码已经省略了)：

例：

```
<%
PreparedStatement pstmt=conn.prepareStatement("INSERT INTO dept (name，members)
      VALUES(?，?)");
pstmt.setString(1，"biology");
pstmt.setArray(2，member_array);
pstmt.executeUpdate();
%>
```

CallableStatement 接口也有类似的方法。

（3）更新记录集中 ARRAY 类型的数据。

ResultSet 接口的 updateArray()方法可以更新数据类型为 ARRAY 的数据。如下面的 JSP 代码所示(数据库连接代码已经省略了)：

例：

```
<%
// retrieve a column containing an SQL ARRAY value from ResultSet rs
java.sql.Array num = rs.getArray("NUMBERS");
// update the column "LATEST_NUMBERS" in a second ResultSet
// with the value retrieved...
rs2.updateArray("LATEST_NUMBERS"，num);
rs2.updateRow();
%>
```

REF 数据类型

SQL REF 数据类型主要用于保存具有复合数据结构的数据，例如，创建一个记录雇员信息的表，如果希望在一个字段内保存雇员的所有信息，如年龄、名称、职务、性别等等，该如何做呢?一个比较好的方法就是：先创建一个新的数据类型(person)，person 数据类型包括了 name(VARCHAR)、age(INT)、home(VARCHAR)等子数据，然后创建 employee 表，该表仅仅含有一个名为 id 的字段，id 的 SQL 数据类型指定为 REF，而且和 person 数据类型关联起来。

当我们往 employee 表中输入数据时，其实数据是分两个地方存放的，employee 表的 id 字段仅仅包含对真实数据位置的一个索引，一个指针，凭借这个指针或者索引，可以找到真实的数据，真实的数据则以 person 结构的形式存放在另一个表中，虽然数据是分开存放的，但是并不防碍对数据库的操作，INSERT、SELECT、UPDATE、DELETE 等数据库操作的 SQL 语句和数据都保存在同一个表(多个字段)的时候没有任何差别。JDBC API 1.0 中不支持 SQL REF 数据类型，在 JDBC API 2.0 中，专门定义了 Ref 接口用于处理 REF 类型的数据。

在 Java 语言中，操作 REF 类型数据的方法如下：

（1）获取 REF 对象。

CallableStatement 接口和 ResultSet 接口都定义了 getRef()方法，该方法的返回值是 Ref 接口的实例对象。如下面的 JSP 代码段所示(数据库连接代码已经省略了)。

例：

```
<%
ResultSet rs = stmt.executeQuery("SELECT oid FROM dogs WHERE "+"name = rover");
rs.next();
Ref ref = rs.getRef(1);
%>
```

读者需要注意的是：Ref 接口的实例对象并不包含实际的数据，而是仅仅包含 REF 结构对与它相关联的保存实际数据的数据类型的一个引用指针。要想获得实际的数据，必须事先根据保存实际数据的自定义数据结构(即为与 REF 数据结构相关联的那一个数据结构)，定义一个 Java 类，然后将 Ref 接口的实例对象强制类型转换为先前定义好的 Java 类，这样就可以获取实在的数据了，在 Java 类中我们还可以定义一套 setXXX()、getXXX()方法，运用这一套方法就可以对获取的实际数据进行操作了，如下面的 JSP 代码段所示：

例：

```
<%
Ref ref = rs.getRef(1);
Address addr = (Address)ref.getObject();
%>
```

（2）更新/存储 REF 类型的数据

更新/存储 REF 类型数据有几种方法，第一种方法是使用 ResultSet 接口的 updateRef()方法，然后使用 updateRow()方法更新当前记录，或者使用 insertRow()方法插入一个新的纪录。第二种方法是使用 CallableStatement 接口(或者 PreparesStatement 接口)的 setRef()方法，设定 IN 参数，然后使用 execute()方法更新数据库中的数据。第三种方法是使用 Ref 接口的 setValue()方法，请看下面的 JSP 代码实例(数据库连接代码已经省略了)。

例：

```
<%
ResultSet rs = stmt.executeQuery("SELECT OID FROM DOGS " +"WHERE NAME
= 'ROVER'");
rs.next();
Ref rover = rs.getRef("OID");
Dog dog = (Dog)rover.getValue(map);
// manipulate instance of Dog
dog.setAge(14);
//...
// store updated Dog
rover.setValue((Object)dog);
%>
```

上面的代码段的含义是：首先执行 SQL 语句，获取 ResultSet 接口的实例对象 rs。接着使用 getRef()方法获取 Ref 接口的实例对象 rover，记录集的字段名 OID 是 REF 类型，它和一个名为 dog 的自定义 SQL 数据类型相关联，在另外的程序中，已经定义了一个 Dog 类，该类继承了 SQLData 接口，Dog 类是根据 SQL 数据类型 dog 定制的，它提供了一组方法以存取 dog 类型的数据，例如 setAge()方法，程序中使用 Ref 接口的 getValue()方法获取

了实在的数据，并且强制转换为 Dog 类的实例对象，getValue()方法的参数 map 是 java.util.Map 接口的实例对象，map 从那里来?在下面另有论述，在此处，读者只需要清楚，这个 Map 对象(map)定义了一组映射，指明 SQL 数据类型 dog 可以映射为 Java 类 Dog。程序接着使用 Dog 类的 setAge()方法更新数据，这里更新的数据仅仅是 Dog 实例对象(dog)中保存的数据，并非是数据库中的数据，最后调用 Ref 接口的方法 setValue()，完成更新数据库的操作，setValue()方法的参数是经过更新的 Dog 对象 dog(需要将它强制转换为 java.lang.Object 类型)。

好了，关于如何使用 JDBC API 2.0 操作 SQL99 规范新定义的数据类型，由于篇幅所限，就介绍到这里，在 9.2.3 节介绍了 BLOB、CLOB 数据类型的操作方法，在本节，介绍了 ARRAY、REF 数据类型的操作方法，这 4 个数据类型比较常用，至于其它不太常用的 SQL 数据类型，如 Structured Type、Datalink 等，本书就不再介绍了，对此感兴趣的读者，请参考 Sun 公司 JavaSof 部门编写的 JDBC API 2.0 规范原文，下一小节将介绍如何使用 JDBC API 2.0 操作自定义的数据类型。

9.2.5　如何处理自定义 SQL 数据类型(SQLData、SQLInput、SQLOutput 接口)

在 9.2.3，9.2.4 小节，分别介绍了如何使用 JDBC API 2.0 操作 BLOB、CLOB、ARRAY、REF 等 SQL92/99 规范新定义的数据类型，读者也许会问，处理 BLOB、CLOB 类型的数据，我会，但是如何处理自定义的数据类型呢?这个问题问的好，幸亏 JDBC API 2.0 的开发人员早已考虑到了这个问题。这个问题的答案是使用 SQLData、SQLInput、SQLOutput 接口。

解决问题的思路

解决这个问题的思路如下：假设数据库管理员创建了一种新的 SQL 数据类型，例如 dog，并且使用这种数据类型(dog 类型)创建了若干个数据库，目前的问题是需要 Java 程序员编写一些 Java 程序，这些 Java 程序可以是 JSP 程序，也可以是 Servlet 程序或者 Java 应用程序，访问使用这种数据类型的数据库(数据表)。Java 程序员应该做的第一件事就是创建一个 Java 类，例如 Dog 类，Dog 类必须继承自 java.sql.SQLData 接口，而且映射 SQL 数据类型 dog，就是说，dog 数据类型含有几个成员，Dog 类就必须要有多少个成员变量，它们之间需要一一对应，相互匹配，Dog 类还要覆盖 java.sql.SQLData 接口的 writeSQL()方法、readSQL()方法，这两个方法主要用于读取或者指定 Dog 类的成员变量。

接下来，程序员可以开始编写访问数据库的程序了。应该先建立数据库连接，返回 Connection 接口的实例对象，接着使用 Connection 接口的 getTypeMap()方法，获取当前数据库连接缺省的，Java 对象映射数据库 SQL 数据类型的映射地图(Map)，这是一个 Map 接口(java.util.Map 接口)的实例对象(map)，程序员应该将 Dog 类的名称添加到这个实例对象中去，最后再调用 Connection 接口的 setTypeMap()方法，重新设定当前数据库连接的映射地图。完成这几步的 Java 代码如下：

例：

```
java.util.Map map = conn.getTypeMap();
map.put("dog"，Class.forName("Dog"));
conn.setTypeMap(map);
```

这几行代码的作用如下，当 JDBC 驱动程序操作自定义数据类型的数据时，在本例中是 dog 类型，它(JDBC Driver)必然要将它(dog)映射为 Java 对象，要不然，就无法对 dog 类型的数据进行操作。那么如何将 dog 类型的数据映射为 Java 对象呢?JDBC 驱动程序将会查看当前数据库连接的映射地图，并且一一试验，看看有那些现存的 Java 类可以和 dog 数据类型相互映射，因为 Dog 类是专门为映射 dog 数据类型编写的，而且又加入了映射地图，因而 Dog 类的名称一定会被 JDBC 驱动程序找到，因为 dog 类不是标准的 Java 数据类型，所以不能用 setInt()、getInt()、setString()、updateBlob()等普通的 setXXX()方法、updateXXX()方法或者 getXXX()方法来存取 dog 类型的数据，那该怎么办呢?可以使用 getObject()、setObject()、updateObject()等方法，但必须进行类型转换，强制转换为 Dog 对象，以 get 过程为例：调用 getObject()方法，并强制转化为 Dog 类的实例对象时，因为 JDBC 驱动程序已经知道了 Dog 类就是 dog 数据类型所映射的 Java 对象，所以 JDBC 驱动程序会自动调用 Dog 类的 readSQL()方法，初始化 Dog 对象，将 dog 类型的数据映射为 Dog 类的实例对象，然后程序员可以使用 Dog 类的成员方法存取 Dog 对象的成员变量。当需要更新 dog 类型的数据时，需要使用 updateObject()方法，将 Dog 对象作为参数传递给该方法。JDBC 驱动程序将会根据映射地图，首先将 Dog 对象映射为 SQL dog 类型的数据，然后再传递给数据库引擎，完成更新数据库操作。

代码实例

程序清单 9.5/9.6/9.7/9.8/9.9/9.10 向读者演示了如何使用 JDBC API 2.0 操作自定义数据类型的数据，请先看程序清单 9.5(createType.sql)。

程序清单 9.5

```
--File Name:createType.sql
--Author:fancy
--Date:2001.2.18
--Note:to create three data type

CREATE TYPE RESIDENCE AS
(
    DOOR NUMERIC(6),
    STREET VARCHAR(100),
    CITY VARCHAR(50),
    OCCUPANT REF(PERSON)
) NOT FINAL

CREATE TYPE FULLNAME AS
(
    FIRST VARCHAR(50),
    LAST VARCHAR(50)
) NOT FINAL

CREATE TYPE PERSON AS
```

```
(
    NAME FULLNAME，
    HEIGHT NUMERIC，
    WEIGHT NUMERIC，
    HOME REF(RESIDENCE)
) NOT FINAL
```

程序清单 9.5(createType.sql)的作用是创建了 PERSON、FULLNAME、RESIDENCE 等三个新的数据类型，读者应该注意到 RESIDENCE 类型的 OCCUPANT 字段是 REF 类型的，而且和 PERSON 类型相互关联，PERSON 类型的字段 HOME 也是 REF 类型的，并且和 RESIDENCE 类型相互关联，PERSON 类型的 NAME 字段的 SQL 数据类型还是刚刚定义的 FULLNAME 类型，接下来请看程序清单 9.6(createTable.sql)。

注意：程序清单 9.5～9.11 使用的数据库系统是 Oracle 8i，SQL 语言的语法服从 PL/SQL 语言的语法。

程序清单 9.6

```
--File Name:createTable.sql
--Author:fancy
--Date:2001.2.18
--Note:to create two table use UDTs

CREATE TABLE HOMES OF RESIDENCE
(
    REF IS OID SYSTEM GENERATED，
    OCCUPANT WITH OPTIONS SCOPE PEOPLE
)

CREATE TABLE PEOPLE OF PERSON
(
    REF IS OID SYSTEM GENERATED，
    OCCUPANT WITH OPTIONS SCOPE HOMES
)
```

程序清单 9.6(createTable.sql)创建了两个新表 HOMES 和 PEOPLE，两表都有 OID 字段，都是 REF 类型的字段，分别和 RESIDENCE、PERSON 数据类型相关联。

程序清单 9.7

```
--File Name:insertTable.sql
--Author:fancy
--Date:2001.2.18
--Note:insert data into table

INSERT INTO PEOPLE (NAME，HEIGHT，WEIGHT) VALUES
(
    NEW FULLNAME('DAFFY'，'DUCK')，
```

```
        4，
        58
);

INSERT INTO HOMES (DOOR，STREET，CITY，OCCUPANT) VALUES
(
        1234，
        'CARTOON LANE'，
        'LOS ANGELES'，
        (SELECT OID FROM PEOPLE P WHERE P.NAME.FIRST = 'DAFFY')
)

UPDATE PEOPLE SET HOME = (SELECT OID FROM HOMES H WHERE
        H.OCCUPANT->NAME.FIRST = 'DAFFY') WHERE FULLNAME.FIRST = 'DAFFY'
```

程序清单 9.8(insertTable.sql)的作用是分别往 HOMES、PEOPLE 表中插入一个新的纪录，并且在这两个记录中建立关联。现在，建立数据的工已经完成了，下一步的任务是为三个自定义的数据类型编写三个 Java 类，让它们可以相互映射，请看程序清单 9.8～9.10。

程序清单 9.8

```
//File Name:Residence.java
//Author:fancy
//Date:2001.2.16
//Note:for maping the SQL UDT:RESIDENCE

import java.sql.*;
import java.io.*;

public class Residence implements SQLData
{
        public int door;
        public String street;
        public String city;
        public Ref occupant;
        private String sql_type;

        public String getSQLTypeName()
        {
                return sql_type;
        }

        public void readSQL (SQLInput stream，String type) throws SQLException
        {
                sql_type = type;
                door = stream.readInt();
```

北京希望电子出版社

```
            street = stream.readString();
            city = stream.readString();
            occupant = stream.readRef();
        }

        public void writeSQL (SQLOutput stream) throws SQLException
        {
            stream.writeInt(door);
            stream.writeString(street);
            stream.writeString(city);
            stream.writeRef(occupant);
        }
    }
```

程序清单 9.8(Residence.java)定义了 Residence 类，该类用于映射 SQL 数据类型 RESIDENCE，该类继承了 java.sql.SQLData 接口，覆盖了 SQLData 接口的 readSQL()方法和 writeSQL()方法、getSQLType()方法，定义了成员变量 door(int 类型)、street(String 类型)、city(String 类型)、occupant(Ref 类型)、sql_type(String 类型)，读者对照程序清单 9.7 和 9.9，应该不难发现：Residence 类的成员变量除了多出一个 sql_type 外，其余的成员变量和 SQL 数据类型 RESIDENCE 的成员字段不但名称相似，而且数据类型也是相互映射的。这不是巧合，而是必须这样做。

把 Residence 类添加到当前数据库的映射地图以后，如果 JDBC 驱动程序碰到不认识的 SQL 数据类型 RESIDENCE，如何处理呢?它(JDBC Driver)必然要求助于映射地图，寻找可以映射 RESIDENCE 类型数据的 Java 类，经过一一试验，JDBC 驱动程序发现 Residence 类可以映射 RESIDENCE 数据类型，于是 Residence 类就可以派上用场了。对记录集数据的操作，可以分为获取和设定两种方式，对于自定义的数据类型，执行第一种操作一般使用 getObject()方法，并且将获取的数据强制转换为与该 SQL 数据类型(本例中是 RESIDENCE)相互映射的 Java 对象(本例中是 Residence 类)。JDBC 驱动程序碰到这种情况，就会自动调用 Residence 类的 readSQL()方法，将获取的数据分别赋给 Residence 类的成员变量，完成初始化 Residence 对象的工作。请读者注意 readSQL()方法，该方法有两个参数，第一个参数 stream，是 SQLInput 接口的实例对象，该对象用于从 SQL 输入流中读取数据，stream 参数由 JDBC 驱动程序自动传递给 readSQL()方法。第二个参数是 type，字符串类型，它表示目前需要映射的 SQL 数据类型的名称，在本例中，type 的值是“RESIDENCE”，type 参数也是由 JDBC 驱动程序自动传递给 readSQL()方法，readSQL()方法体中，不但使用 SQLInput 接口的 readXXX()方法，将获取的数据初始化 Residence 类的成员变量，而且将 type 的值赋给了成员变量 sql_type，这样一来 sql_type 中就保存了与 Residence 类相互映射的 SQL 数据类型 RESIDENCE 的名称。当使用 Residence 类的 getSQLType()方法时，将 sql_type 返回，据此可以知道 Residence 究竟映射了哪一个 SQL 数据类型。

当执行设定记录集数据的操作时，情况十分类似，这时应该使用 setObjec()方法，将 Residence 对象作为参数传递给 setObject()方法。JDBC 驱动程序执行这一操作，将会自动调用 Residence 类的方法 writeSQL()，将 Residence 对象中保存的数据都写到 SQL 输出流去，

剩下的任务就由数据库服务器完成了，我们不需要过多关心，writeSQL()方法的参数 stream，是 SQLOutput 接口的实例对象，该参数由 JDBC 驱动程序自动传递给 writeSQL()方法，

writeSQL()方法体中，调用 SQLOutput 接口的方法 writeXXX()，将 Residence 类的成员变量的值都写入到 SQL 输出流中。

getSQLType()、writeSQL()、readSQL()等方法都在 SQLData 接口中定义了原型，在 SQLData 接口的子类中，必须将它们覆盖掉，用自己编写的同名方法代替。

程序清单 9.9、9.10 和程序清单 9.8 的情况差不多，我们就不再详细分析了，接下来请看程序清单 9.9(Fullname.java)。

注意：在 Residence 类中，也可以定义别的成员方法，用以操作成员变量，并非只能重载 SQLData 接口的三个方法，而不能新定义别的方法。

程序清单 9.9

```
//File Name:Fullname.java
//Author:fancy
//Date:2001.2.18
//Note:for maping the SQL UDT:FULLNAME

import java.sql.*;
import java.io.*;

public class Fullname implements SQLData
{
    public String first;
    public String last;
    private String sql_type;
    public String getSQLTypeName()
    {
        return sql_type;
    }

    public void readSQL (SQLInput stream，String type) throws SQLException
    {
sql_type = type;
        first = stream.readString();
        last = stream.readString();
    }

    public void writeSQL (SQLOutput stream) throws SQLException
    {
        stream.writeString(first);
        stream.writeString(last);
    }
```

北京希望电子出版社

```
}
```

程序清单 9.9(Fullname.java)定义了 Java 类 Fullname，主要用于映射 SQL 数据类型 FULLNAME。接下来请看程序清单 9.10(Person.java)。

程序清单 9.10

```
//File Name:Person.java
//Author:fancy
//Date:2001.2.18
//Note:for maping the SQL UDT:PERSON

import java.sql.*;
import java.io.*;

public class Person implements SQLData
{
    Fullname name;
    float height;
    float weight;
    Ref home;
    private String sql_type;

    public String getSQLTypeName()
    {
        return sql_type;
    }

    public void readSQL (SQLInput stream，String type) throws SQLException
    {
        sql_type = type;
        name = (Fullname)stream.readObject();
        height = stream.readFloat();
        weight = stream.readFloat();
        home = stream.readRef();
    }

    public void writeSQL (SQLOutput stream) throws SQLException
    {
        stream.writeObject(name);
        stream.writeFloat(height);
        stream.writeFloat(weight);
        stream.writeRef(home);
    }
}
```

北京希望电子出版社

程序清单 9.10(Person.java)定义了 Java 类 Person，主要用于映射 SQL 数据类型 PERSON。接下来请看程序清单 9.11(useUDT.jsp)。

程序清单 9.11

```
<%--
File Name:useUDT.jsp
Author:fancy
Date:2001.2.18
Note:this jsp program can get/store the UDT data
--%>

<%@ page import="java.io.*"□%>
<%@ page import="java.sql.*"□%>

<%
// set up mappings for the connection
try
{
      Class.forName("JDBC Driver Class");
}
catch(Exception fe)
{
      //to do it
}

try
{
      Connection con=DriverManage.getConnection(JDBC URL□，User□，Pass);
      java.util.Map map = con.getTypeMap();
      map.put(S.RESIDENCE"，Class.forName("Residence"));
      map.put("S.FULLNAME"，Class.forName("Fullname"));
      map.put("S.PERSON"，Class.forName("Person"));
      con.setTypeMap(map);
}
catch (ClassNotFoundException ex) {}
PreparedStatement pstmt;
ResultSet rs;
pstmt = con.prepareStatement("SELECT OCCUPANT FROM HOMES");
rs = pstmt.executeQuery();
rs.next();
Ref ref = rs.getRef(1);
pstmt = con.prepareStatement("SELECT FULLNAME FROM PEOPLE WHERE OID = ?");
pstmt.setRef(1，ref);
rs = pstmt.executeQuery();
```

```
rs.next();
Fullame who = (Fullname)rs.getObject(1);
// prints "Daffy Duck"
out.println(who.first + " " + who.last);
%>
```

程序清单 9.11(useUDT.jsp)的关键之处在于将 Residence 类、Fullname 类、Person 类加入了映射地图，当 JDBC 驱动程序碰到自定义的 SQL 数据类型 RESIDENCE、FULLNAME、PERSON 时，会自动将这三种类型的数据映射为对应的 Java 对象。

SQL 数据类型到 Java 类型的缺省映射

前面已经提到，当前的数据库连接有一个缺省的映射地图，JDBC 驱动程序会根据该地图将 JDBC 数据类型(亦即 SQL 数据类型)映射为相对应的 Java 类型，同样 JDBC 驱动程序也会将 Java 类型映射为对应的 SQL 数据类型，然后再将数据发送到数据库引擎中，在 9.4 节中列出了 SQL 数据类型到 Java 类型的缺省映射。

SQLData 接口

SQLData 接口是 JDBC API 2.0 中新定义的接口，它提供了 getSQLType()方法、readSQL()方法、writeSQL()方法的原型，当程序员定义映射 SQL 数据类型的类时，必须继承自 SQLData 接口，而且覆盖这三个方法，在这里，我们要再次提醒读者：readSQL()、writeSQL()方法由 JDBC 驱动程序自动调用，不是由程序员编程调用的。上述三个方法的定义如下：

```
public java.lang.String getSQLTypeName();
public void readSQL(SQLInput  stream，java.lang.String  typeName);
public void writeSQL(SQLOutput  stream);
```

SQLInput 接口

SQLInput 接口主要用于从 SQL 输入流中读取数据。SQLInput 接口没有构造函数，它的实例对象由 JDBC 驱动程序创建，然后作为参数，传递给 readSQL()方法。SQLInput 接口定义了一整套的 readXXX()方法，可以读取各式各样的数据，包括 BLOB、CLOB、ARRAY、REF 类型的数据在内，下面是 readBlob()、readClob()方法的定义，至于其他的 readXXX()方法，读者可以参考 JDBC 的文档。

```
public Blob readBlob();
public Clob readClob();
```

SQLOutput 接口

SQLOutput 接口主要用于往 SQL 输出流中写入数据，SQLOutput 接口没有构造函数，它的实例对象由 JDBC 驱动程序创建，然后作为参数，传递给 writeSQL()方法。SQLOutput 接口定义了一整套的 writeXXX()方法，可以写入各式各样的数据，包括 BLOB、CLOB、ARRAY、REF 类型的数据在内，下面是 writeBlob()、writeClob()、writeArray()方法的定义，至于其他的 writeXXX()方法，读者可以参考 JDBC 的文档。

```
public void writeBlob(Blob  x);
```

public void writeClob(Clob x);
public void writeArray(Array x)

9.3　JDBC API 3.0 简介

JDBC API 3.0 和 JDBC API 2.0 相比较，改进的地方不是特别多，主要有以下几个方面的变化：

- 将 RowSet 包捆绑到 JDBC API 3.0 中，在 JDBC API 2.0 中，不包括 RowSet 包，RowSet 包只是 JDBC Optional Package 的一部分。
- 在 ResultSet 接口中添加了 updateArray()、updateBlob()、updateClob()、updateRef()等方法。
- 新定义了 ParameterMetaData 接口，应用这个接口，可以获取预编译 SQL 语句(prepared statements)的 IN/OUT 参数的属性。
- 新支持 SQL 数据类型 DATALINK，映射 DATALINK 类型的 Java 对象是 java.net.URL 类。

由于 JDBC API 3.0 将在 Java 1.4 中被完全支持，目前用得非常少，所以本节仅仅简单介绍新增加的 ParameterMetaData 接口，关于 JDBC API 3.0 详细的信息，读者可以参考 Sun 公司的 JDBC 3.0 技术规范文档。

下面介绍 ParameterMetaData 接口。

我们都知道，PreparedStatement 接口的实例对象在创建的时候，必然含有带若干个未知参数的一条 SQL 语句。SQL 语句在执行前就被预编译，优化，执行起来效率较高，那若干个未知参数在执行 SQL 语句前可以使用 PreparedStatement 接口的 setXXX()方法指定。ParameterMetaData 接口的主要作用就是获取那些未知参数的信息。

9.3.1　获取 ParameterMetaData 接口实例对象的方法

在 PreparedStatement 接口中定义了 getParameterMetaData()方法，这个方法将返回 ParameterMetaData 接口的实例对象，getParameterMetaData()方法的定义如下：

public ParameterMetaData getParameterMetaData();

CallableStatement 接口继承了 PreparedStatement 接口，因此也能使用这个方法以获取 ParameterMetaData 接口的实例对象，如下面的代码所示(数据库连接代码已经省略，cstmt 是 CallableStatement 接口的实例对象)。

例：

```
ParameterMetaData pmd=cstmt.getParameterMetaData();
```

9.3.2　ParameterMetaData 接口的方法

ParameterMetaData 接口定义的主要方法如下所示：

public int getParameterCount();　该方法用于获取 SQL 语句中未知参数的数量。

public int isNullable(int param);　该方法可以知道某个特定的位置参数是否允许空值(null)，参数 param 是未知参数的索引，第一个未知参数的索引是 1，第二个未知参数的

索引是 2，以此类推。

public int getParameterType(int param);　该方法可以知道特定未知参数所对应的 SQL 数据类型，参数 param 指定了目标未知参数的位置，该方法的返回值是一个整型常数，这些常数在 java.sql.Types 类中定义。

public java.lang.String getParameterTypeName(int param);　该方法可以知道特定未知参数所对应的 SQL 数据类型，和 getParameterType()方法的功能一样，不过这个方法的返回值是字符串，而不是预定义的整型常数。

public java.lang.String getParameterClassName(int param);　这个方法主要用于处理自定义的数据类型(UDT)，因为有些未知参数的 SQL 数据类型可以是自定义的数据类型，程序员自然应该定义一个 Java 类来映射自定义的 SQL 数据类型，使用这个方法可以返回与特定未知参数的 SQL 数据类型相互映射的 Java 类的类名，其实，这个方法并非只对自定义的 SQL 数据类型有效，对于一般的 SQL 数据类型也是有效的，只不过 Java 程序员都应该熟知普通常用的 SQL 数据类型与 Java 对象之间的映射关系(见表 9.1)，再调用这个方法，没有什么意义。

9.4　附录：JDBC 数据类型和 Java 数据类型的映射关系

表 9.1　JDBC Type 到 Java Type 的映射关系

JDBC 类型	Java 类型
CHAR	String
REAL	float
DATALINK	java.net.URL
REF	Ref
STRUCT	Struct
DISTINCT	mapping of underlying type
ARRAY	Array
BLOB	Blob
CLOB	Clob
TIMESTAMP	java.sql.Timestamp
TIME	java.sql.Time
DATE	java.sql.Date
LONGVARBINARY	byte[]
VARBINARY	byte[]
BINARY	byte[]
DOUBLE	double
FLOAT	double
VARCHAR	String
LONGVARCHAR	String

（续表）

JDBC 类型	Java 类型
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	boolean
BOOLEAN	boolean
TINYINT	byte
SMALLINT	short
INTEGER	int
BIGINT	long

注意：表 9.1 的映射关系将被 ResultSet 等接口的 getObject()方法所使用，用以将数据库返回的数据映射为本地的 Java 数据类型的数据。

表 9.2　Java Type 到 JDBC Type 的映射关系

Java 类型	JDBC 类型
String	CHAR，VARCHAR，orLONGVARCHAR
java.math.BigDecimal	NUMERIC
boolean	BIT or BOOLEAN
byte	TINYINT
short	SMALLINT
int	INTEGER
long	BIGINT
float	REAL
double	DOUBLE
byte[]	BINARY，VARBINARY，LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
Clob	CLOB
Blob	BLOB
Array	ARRAY
Struct	STRUCT
Ref	REF
java.net.URL	DATALINK
Java class	JAVA_OBJECT

注意：表 9.2 的映射关系将被 ResultSet 等接口的 setObject()方法所使用，可以把本地的 Java 数据类型的数据映射为数据库支持的数据类型。

北京希望电子出版社

表 9.3　JDBC Type 到 Java Object Type 的映射关系

JDBC 类型	Java 对象类型
FLOAT	Double
DOUBLE	Double
BINARY	byte[]
VARBINARY	byte[]
JAVA_OBJECT	underlying Java class
DATALINK	java.net.URL
REF	Ref
STRUCT	Struct or SQLData
ARRAY	Array
DISTINCT	Object type of underlying type
TIMESTAMP	java.sql.Timestamp
TIME	java.sql.Time
DATE	java.sql.Date
LONGVARBINARY	byte[]
CLOB	Clob
BLOB	Blob
REAL	Float
BIGINT	Long
INTEGER	Integer
SMALLINT	Integer
TINYINT	Integer
CHAR	String
VARCHAR	String
LONGVARCHAR	String
NUMERIC	java.math.BigDecimal
DECIMAL	java.math.BigDecimal
BIT	Boolean
BOOLEAN	Boolean

注意：表 9.3 的映射关系将被 ResultSet 等接口的 getObject()方法所使用，用以将数据库返回的数据映射为本地的 Java 对象。

表 9.4　Java Object Type 到 JDBC Type 的映射关系

Java 对象类型	JDBC 类型
Java class	JAVA_OBJECT
Long	BIGINT

（续表）

Java 对象类型	JDBC 类型
Integer	INTEGER
String	CHAR，VARCHAR，orLONGVARCHAR
Boolean	BIT or BOOLEAN
java.math.BigDecimal	NUMERIC
Float	REAL
Double	DOUBLE
byte[]	BINARY，VARBINARY，LONGVARBINARY
java.sql.Date	DATE
java.sql.Time	TIME
java.sql.Timestamp	TIMESTAMP
Clob	CLOB
Blob	BLOB
Array	ARRAY
Struct	STRUCT
Ref	REF
java.net.URL	DATALINK

注意：表 9.4 的映射关系将被 ResultSet 等接口的 setObject()方法所使用，可以把本地的 Java 对象映射为数据库支持的数据类型。

9.5　本 章 小 结

在本章中，我们介绍了 JDBC API 2.0 相对于 JDBC API 1.0 的全新特性，包括全新的记录集接口(ResultSet 接口)、如何处理 BLOB、CLOB 类型的数据(Blob、Clob 接口)、如何处理新的 SQL 数据类型、如何处理自定义的 SQL 数据类型等全新的内容，在 9.3 节，我们还简单地介绍了 JDBC API 3.0 的情况，并探讨了 ParameterMetaData 接口的用法和该接口的主要方法。经过本章的学习，相信读者对 JDBC 技术的理解和应用能力，都会跃上一个新的台阶。

在第 10 章，将要介绍 JDBC Optional Package 的 RowSet 包、CachedRowSet 包。

第 10 章　JDBC Optional Package

在第 9 章，已经介绍了 JDBC API 2.0/3.0 的基础知识，在本章，我们将要介绍 JDBC Optional Package 2.0 包的知识。JDBC Optional Package 2.0 包是 JDBC API 的扩展，它在 JDBC API 的基础上添加了很多有用而强大的功能，因此，如果你希望成为一个 JSP 高手或者是 J2EE 技术高手，那么你对 JDBC Optional Package 2.0 包的用法就一定要有所了解。

本章讲述的主要内容是：

- JDBC Optional Package 的来龙去脉
- RowSet 包介绍
- CachedRowSet 包介绍
- RowSet 和数据库连接缓冲池
- RowSet 和 JNDI 技术

下面的第一节首先向读者介绍什么是 JDBC Optional Package。

10.1　JDBC Optional Package 是什么

JDBC Optional Package 是什么?相信很多读者都怀有这个问题。这也难怪，现在市场上介绍 Java 及其相关技术的书籍很少，介绍 JDBC API 的书籍/技术文章更是寥寥可数，更别提什么 JDBC Optional Package 了。本节将向读者介绍 JDBC Optional Package 的来龙去脉，以及 JDBC Optional Package 在 J2EE 技术体系中的重要作用。

1997 年 6 月 Java Software(JavaSoft)和 JDC(Java Developer Community)开始联合制订 JDBC API 2.0 的技术规范，从一开始，JDBC API 2.0 就分为两部分，第一部分是 JDBC Core API 2.0，这就是我们通常所说的 JDBC API 2.0，第 9 章介绍的其实就是 JDBC Core API 2.0，这部分 API 存在于 java.sql 包中，JDBC Core API 2.0 包含于 JDK 1.2/1.3 中，是 Java2 平台的一部分。JDBC API 2.0 的第二部分就是 JDBC Standard Extension API 2.0（JDBC 标准扩展 API2.0），亦即本章开头所说的 JDBC Optional Package 2.0 包，JDBC Optional Package 2.0 包其实也包含了两个部分，第一部分 API 内含于 javax.sql 包中，由于这部分 API 的最重要的接口是 RowSet，所以这部分 API 通常就称为 RowSet 包，JDBC Optional Package 2.0 包的第二部分 API 内含于 sun.jdbc.rowset 包中，由于这个包得最重要的类是 CachedRowSet 类，因此，sun.jdbc.rowset 包又称为 CachedRowSet 包，CachedRowSet 包和 RowSet 包合起来统称 JDBC Optional Package。在 Java Software 的规划中，以 java 开头的包是 Java 核心包，以 javax 开头的包是 Java 标准扩展包，以 sun、sunw 开头的包是 Sun 公司的扩展包，因此 JDBC Optional Package 也是 Java 标准扩展包的一部分。

除了 JDBC Optional Package 以外，Java Software 还开发了 sun.jdbc.odbc 包，这个包主要用于处理 ODBC 数据源，如果你的数据库系统使用 Microsoft SQL Server 7.0，那么这个包就相当有用了。

Java Software 在 JDBC API 2.0 的基础上开发了 JDBC API 3.0，JDBC API 3.0 不但包含了 java.sql 包，还包含了 javax.sql 包，就是说：JDBC Optional Package 的 RowSet 包已经融合到 JDBC API 3.0 中去了，不过 CachedRowSet 包仍然游离在 JDBC API 3.0 之外，不知道 Java Software 此举是何用意。

Java Software 和 JDC 之所以开发 JDBC Optional Package，乃是出于以下的目的：

- 保持 JDBC Core API 的紧凑和易用性。JDBC Core API 只完成一些最基本、最简单、最有用的功能，将一些强大的、不是必需的数据库访问功能放到 JDBC Optional Package 中实现。
- 使 JDBC 技术可以和其他的 Java 扩展 API 有效地融合。在 JDBC Optional Package 中，提供了对 JavaBeans、EJB、JNDI、JTA(Java Transaction API)数据库缓冲池的支持。
- 提供了许多特别的处理数据的功能。如利用 CachedRowSet 包，可以实现从 XML 文件中读取数据，并将数据输入数据库的功能。

从目前发布的 JDBC Optional Package 2.0 版本来看，Java Software 和 JDC 基本达到上面所列的目标，而且，JDBC Optional Package 2.0 的发布，还促进了 J2EE 技术的大融合，这对于 Java 技术的发展，实在是起着至关重要的作用。下面第二节将向读者介绍 JDBC Optional Package 2.0 的 RowSet 包。

10.2　RowSet 包

10.2.1　RowSet 包简介

RowSet 包含有下面的类/接口：

javax.sql.ConnectionEvent(Class);
javax.sql.ConnectionEventListener(Interface);
javax.sql.ConnectionPoolDataSource(Interface);
javax.sql.DataSource(Interface);
javax.sql.PooledConnection(Interface);
javax.sql.RowSet(Interface);
javax.sql.RowSetEvent(Class);
javax.sql.RowSetInternal(Interface);
javax.sql.RowSetListener(Interface);
javax.sql.RowSetMetaData(Interface);
javax.sql.RowSetReader(Interface);
javax.sql.RowSetWriter(Interface);
javax.sql.XAConnection(Interface);
javax.sql.XADataSource(Interface);

在上面列出来的类/接口中，DataSource 接口主要用于处理 JNDI 和 JDBC 的相互调用问题; ConnectionEvent 类、ConnectionEventListener 接口、ConnectionPoolDataSource 接口、PooledConnection 接口主要用于处理数据库连接缓冲池(Connection Pooling);RowSet 接口、

RowSetEvent 接口、RowSetInternal 接口、RowSetListener 接口、RowSetMetaData 接口、RowSetReader 接口、RowSetWriter 接口主要用于完成 RowSet 的功能(即 JDBC 技术应用于 JavaBeans 组件等功能)。XAConnection 接口、XADataSource 接口主要用于分布式事务处理(Distributed Transactions)。

在 10.2 节中，我们将主要介绍如何使用 RowSet 包完成 RowSet 功能，这也是本章所要重点介绍的内容之一，至于如何使用 RowSet 包处理数据库连接缓冲池、如何与 JNDI 数据服务相交互的知识，我们将分别在 10.4，10.5 节中做简要介绍。

10.2.2　RowSet 接口

RowSet 接口是 RowSet 包中最重要的接口，RowSet 接口为 JDBC API 添加了对 JavaBeans(TM)组件模型的支持，在一个可视化的 JavaBeans 开发环境中，使用 RowSet 对象可以像使用 JavaBeans 组件那样方便。RowSet 对象可以在设计时刻被创建和指定属性，在程序的运行时刻被执行，RowSet 接口定义了一套 JavaBeans 组件的属性方法，RowSet 实例对象可以使用这些 setXXX()方法指定这些属性，以便和数据库建立连接，并且从数据库中读取数据，setXXX()方法还可以设定 SQL 语句的 IN 参数，也就是说，RowSet 实例对象执行的 SQL 语句是预编译过的，运行效率相当高。RowSet 接口支持 JavaBeans 的事件模型，允许同一个应用程序的其他组件在运行时刻，并且 RowSet 接口的实例对象发生了某种变化/事件的情况下(例如，记录集的数据库游标向前移动了一位)，调用 RowSet 接口的 setXXX()方法，修改 RowSet 对象的属性。

RowSet 接口继承自 java.sql.ResultSet 接口，它不但具有 ResultSet 接口的强大的记录集功能，而且比 ResultSet 接口更容易使用。下面我们就来介绍如何使用 RowSet 接口。

（1）创建 RowSet 接口的实例对象。

在 JDBC API 中，ResultSet 接口的实例对象一般不是调用构造函数直接创建，而是由其它方法的返回结果初始化，例如 Statement 接口的 executeQuery()方法的返回结果就是 ResultSet 接口的实例对象。RowSet 接口的创建方法如下。

例：

```
RowSet rs=new RowSet();
```

（2）设定 RowSet 对象的数据库连接属性。

创建了 RowSet 接口的实例对象以后，必须设定 RowSet 对象的数据库连接属性，以便 RowSet 对象可以和数据库服务器建立连接，进而向数据库发送 SQL 语句，访问数据库中的数据，并获取结果记录集。RowSet 接口定义了下面的方法(仅列出比较重要的方法)用以指定数据库连接属性，如用户名、访问密码、数据源 URL 等属性。

public void setConcurrency(intconcurrency);　该方法可以指定 RowSet 对象和数据库建立的连接的协同级别，通俗地讲，就是指 RowSet 对象所包含的结果记录集的数据是否可以更新，方法参数 concurrency 是一个整型常数，具体的常数定义，读者可以参考第 9 章关于 ResultSet 接口部分，这些常数在 ResultSet 接口中被定义，RowSet 接口继承自 ResultSet 接口，自然可以使用这些预定义的常数。

public void setDataSourceName(java.lang.Stringname);　RowSet 接口的实例对象可以使用这个方法查询(lookup)特定的(由参数 name 指定)，存在于 JNDI 命名服务(JNDI

naming service)中的 JDBC DataSource 对象，在 10.5 节中我们将介绍与此有关的内容。

public void setMaxRows(intmax);　该方法可以指定 RowSet 对象中所包含的记录集的最大记录数。

public void setPassword(java.lang.Stringpassword);　该方法可以指定 RowSet 对象和数据库建立连接所需的用户密码。

public void setReadOnly(booleanvalue);　该方法指定 RowSet 对象中所包含的记录集是否只读，方法参数为布尔值，如果为 true，那么结果集只读，反之亦然。

public void setTransactionIsolation(intlevel);　该方法指定事务处理的隔离级别。

public void setType(inttype);　RowSet 对象可以使用此方法指定记录集游标的类型，例如可以前后移动的数据库游标，或者是仅仅可以往前移动的游标。方法参数 type 是一个这整型常数，这些常数在 ResultSet 接口中定义，读者可以参考本书第 9 章介绍 ResultSet 接口的内容，或者是查阅 JDBC 的帮助文档。

public void setTypeMap(java.util.Mapmap);　RowSet 对象可以使用此方法指定当前 RowSet 对象与数据库连接时所使用的映射地图，关于映射地图的知识，读者可以参考第 9 章 9.2.5 节的相关内容，这个方法在需要处理自定义的 SQL 数据类型时特别重要，读者一定要特别留意。

public void setUrl(java.lang.Stringurl);　该方法用于指定 RowSet 对象需要与之建立连接的数据源的 JDBC URL 地址，例如：

jdbc：odbc：test。

public void setUsername(java.lang.Stringname);　该方法用于指定 RowSet 对象访问数据库服务器时所需的用户名。

（3）使用 RowSet 对象与数据库建立连接。

如何使用 RowSet 对象和数据库建立连接呢?请看下面的 JSP 代码段。

例：

```
<%
Class.forName(sun.jdbc.odbc.JdbcOdbcDriver);
rset.setUrl(jdbc：odbc：test);
rset.setUser(sa);
rset.setPassword();
rest.setType(ResultSet.TYPE_SCROLL_SENSITIVE);
%>
```

在上面的代码段中，首先需要载入 JDBC 驱动程序，然后分别使用 setUrl()、setUser()、setPassword()等方法指定 JDBC 数据源 URL、数据库用户名、数据库访问密码等访问参数，代码段的最后使用 setType()方法指定 RowSet 实例对象所包含的结果记录集的类型是：数据库游标可以前后移动，并且结果记录集的数据对其他用户的数据库更新操作敏感。关于常数 ResultSet. TYPE_SCROLL_SENSITIVE 的意义，读者可以参考第 9 章介绍 ResultSet 接口的内容。其实，上面的 JSP 代码段并没有真正与数据库系统建立了连接，它仅仅指定了最重要的数据库连接参数，使得 RowSet 处于就绪状态，随时都可以和数据库建立连接。若需要与数据库建立连接，除了指定数据库连接参数以外，第二步就必须向数据库发送 SQL

命令，设置 SQL 命令的 IN 参数，最后调用 execute()方法，该方法的内部流程是：建立真正的数据库连接，调用数据库引擎，完成 SQL 命令，并且用数据库系统返回的结果数据填充 RowSet 对象的记录集。这时，RowSet 对象和 ResultSet 对象差不多，也可以编历记录集每一个记录，或者定位到某一个记录，使用 getXXX()、updateXXX()等方法对记录集中的数据进行操作。

（4）使用 RowSet 对象向数据库服务器发送命令。

在设定了 RowSet 对象与数据库连接的属性以后，就可以利用 RowSet 对象向数据库发送命令和命令参数，使用 RowSet 接口的 setCommand()可以向数据库系统发送预编译的 SQL 命令，使用其它的 setXXX()方法可以设定 SQL 命令的 IN 参数，这种方式十分类似于 PreparedStatement 接口和 CallableStatement 接口发送预编译 SQL 命令的方式。RowSet 接口中定义的相关方法如下所示：

public void setCommand(java.lang.Stringcmd); 使用该方法可以向数据库发送预编译的 SQL 语句，SQL 语句中可以含有输入参数，参数的位置以?号代替，第一个输入参数的索引为 1，第二个参数的索引为 2，以此类推。

public void setArray(inti，java.sql.Arrayx); 该方法可以设定 SQL 语句中 SQL 数据类型是 SQL ARRRAY 的参数，方法参数 i 指定输入参数的索引，方法参数 x 代表替换目标参数的 java.sql.Array 对象。

public void setBlob(inti，java.sql.Blobx); 该方法和 setArray()差不多，主要用于设定 SQL 语句中数据类型为 SQL BLOB 类型的输入参数。

public void setString(intparameterIndex，java.lang.Stringx); 该方法主要用于设定 SQL 语句中数据类型为 VARCHAR、NVARCHAR 等类型的输入参数，parameterIndex 代表需要替换的输入参数的索引，x 代表需要引入的 java String 对象。

还有很多 setXXX()方法都可以指定 SQL 语句的输入参数的值，但是由于篇幅问题，在此就略过不提了，读者可以参考相关的文献，使用 RowSet 接口的 clearParameter()方法可以清除所有已经指定的 SQL 输入参数的值。

下面是 JSP 代码实例，向读者演示如何使用 RowSet 对象的 setCommand()方法向数据库发送 SQL 命令，并且使用 setXXX()方法指定 SQL 命令的输入参数(rset 是 RowSet 接口的实例对象)。

例：

```
<%
rset.setCommand(SELECT * FROM tbuser where name=? and pass=?);
rset.setString(1，fancy);
rset.setString(2，fancy);
%>
```

其实，setCommand()方法和前面讲述的 setPassword()、setUrl()等方法都没有和数据库系统建立事实上的数据库连接，它们都用于设定 RowSet 对象的属性。SQL 命令也可以看作 RowSet 对象的属性之一。

（5）使用 RowSet 对象执行数据库操作。

如果读者已经按照上面介绍的步骤设置好 RowSet 对象的连接属性，SQL 命令属性，

输入参数，那么现在就可以使用 RowSet 接口的 execute()方法执行数据库操作，execute()方法首先应用数据库连接属性与数据库建立事实上的连接，然后向数据库系统发送 SQL 命令，接受数据库返回的数据，使用这些数据实例化 RowSet 对象内部的结果记录集。execute()方法并不返回任何 ResultSet 接口的实例对象，它仅用数据库返回的数据填充 RowSet 对象，因为 RowSet 接口继承自 ResultSet 接口，它具有 ResultSet 接口的全部功能，在执行了 execute()方法以后，可以将 RowSet 对象当作一个 ResultSet 对象来使用，这是没有任何问题的，请看下面的 JSP 代码段，该代码段向读者演示了如何使用 RowSet 对象执行数据库操作，如何使用 RowSet 接口的方法获取结果记录集的信息(rset 是 RowSet 接口的实例对象)。

例：

```
<%
rset.execute();
while(rset.next())
{
        out.println(username：+rset.getString(name)+<br>);
        out.println(password：+rset.getString(pass)+<br>);
      }
rset.close();
%>
```

（6）给 RowSet 对象添加事件监听者。

RowSet 对象可以和某个 RowSet 事件监听者(RowSetListener)绑定在一起，RowSet 事件监听者可以对 RowSet 对象中发生的事件作出响应，例如数据库游标的移动事件。

addRowsetListener()方法可以是一个 RowSet 事件监听者和 RowSet 对象绑定在一起，removeRowSetListener()方法则可以解除这种绑定关系，RowSet 事件监听者一旦和 RowSet 对象解除了绑定关系，那么它对 RowSet 对象内部所发生的事件就不能做出响应了。

这两个方法的定义如下：

public void addRowSetListener(RowSetListenerlistener);

public void removeRowSetListener(RowSetListenerlistener);

参数 listener 是 RowSetListener 接口的实例对象。关于 RowSetListener 接口的知识，我们将在下一小节做个简单介绍。

10.2.3　RowSetListener 接口

RowSetListener 接口主要用于监听 RowSet 对象中发生的事件，在必要的时候，可以对这些事件作出响应。例如：我们可以将一个 RowSetListener 对象绑定在 RowSet 对象上，响应 RowSet 对象中发生的任何事件，并将事件记录到某个特定的 log 文件中，当应用程序结束时，通过查看 log 文件，我们就可以知道应用程序在运行时，都做了哪些数据库操作，更新了多少个记录的数据。RowSetListener 接口可以响应的事件有三种：

（1）cursorMove：数据库游标移动事件。

（2）rowChanged：当新的记录被插入，或者是记录被删除、更新的事件。

（3）rowSetChanged：当 RowSet 对象被改变的事件。

RowSetListener 接口定义了三种方法以响应这三个事件，分别如下所示：

public void rowSetChanged(RowSetEventevent);

public void rowChanged(RowSetEventevent);

public void cursorMoved(RowSetEventevent);

这些方法的参数 event 是 RowSetEvent 类的实例对象，代表 RowSet 对象中产生的事件。这三个方法一般由 JDBC 数据库驱动程序自动调用，Java 程序员不需要关心这些方法何时被调用，如何被用。那么如何使用 RowSetListener 接口呢?普通的方法是：程序员需要为 RowSet 对象定义一个事件监听类(如 EchoListener 类)，EchoListener 类必须继承 RowSetListener 接口，在 EchoListener 类的定义中，程序员必须覆盖上述的三个方法，EchoListener 类定义好以后，我们就可以编程将它绑定到某个 RowSet 对象上。请读者参考代码清单 10.1(EchoListener.java)。

代码清单 10.1

```
//File Name：EchoListener.java
//Author：fancy
//Date：2001.2.25
//Note：for   class EchoListener   extends RowSetListener
import javax.sql.*;
import java.io.*;

public class EchoListener implements RowSetListener
{

    public void rowSetChanged(RowSetEvent event)
    {
        System.out.println("The rowset changed");
    }

    public void rowChanged(RowSetEvent event)
    {
        System.out.println("Row changed");
    }

    public void cursorMoved(RowSetEvent event)
    {
        System.out.println("Rowset cursor moved");
    }
}
```

在程序清单 10.1(EchoListener.java)中，定义了一个 RowSet 事件监听器类 EchoListener，这个类继承自 RowSetListener 接口，同样响应 cursorMoved、rowChanged、rowSetChanged 事件，例如，如果发生了 cursorMoved 事件，EchoListener 类就会调用 cursorMoved()方法，输出 Rowset cursor moved 的字样，如果发生了其他的事件，相应方法也差不多。

程序清单 10.2 中，演示了如何使用 EchoListener 类监听 RowSet 对象的事件。

程序清单 10. 2

```
//File Name：RowSetExam.java
//Author：：fancy
//Date：2001.2.19
//Note：test the EchoListener    class

import java.sql.*;
import javax.sql.*;
import java.io.*;

public class RowSetExam
{
    Connection conn = null;
    Statement stmt = null;
    RowSet    rs = null;

    public static void main(String Args[])
    {
        RowSetExam    rsExam = new RowSetExam();
        //rsExam.setup();
        rsExam.go();
    }

    public void go()
    {
        // Register the driver.

        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch (ClassNotFoundException ex)
        {
            System.err.println("SQLException：  " + ex.getMessage());
        }

        try
        {
            EchoListener    echo = new EchoListener();
            rs = new RowSet();
            rs.addRowSetListener(echo);
            rs.setUrl("jdbc：odbc：test");
```

北京希望电子出版社

```
            rs.setUsername("sa");
            rs.setPassword("");
            rs.setCommand("SELECT * FROM goods");
            rs.execute();

            // a scroll in the park...
            scrollCursor(rs);
            rs.close();
        }
        catch (SQLException ex)
        {
            System.err.println("SQLException：    " + ex.getMessage());
        }
    }

    public void scrollCursor(RowSet    rs) throws SQLException
    {
        System.out.println("Fetching from RowSet...");
        String v1;
        String    v2;
        try
        {
            while (rs.next())
            {
                try
                {
                    v1 = crs.getString(1);
                    if (rs.wasNull() == false)
                    {
                        System.out.println("v1 is " + v1);
                    }
                    else
                    {
                        System.out.println("v1 is null");
                    }
                }
                catch (SQLException ex)
                {
                    System.err.println("Unable to get：    " + ex.getMessage());
                }

                v2 =rs.getString(2);
                if (rs.wasNull() == false)
```

```
                    {
                        System.out.println("v2 is " + v2);
                    }
                    else
                    {
                        System.out.println("v2 is null");
                    }
                }
            }
            catch (SQLException ex)
            {
                ex.printStackTrace();
            }
        }
    }
```

在程序清单 10.2(RowSetExam.java)中，演示了 RowSet 接口和 RowSetListener 接口(EchoListener 类)的用法。程序清单 10.2 是一个 Java 应用程序，它有两个函数，go()函数是程序的主运行逻辑 ，scrollCursor()函数主要用于移动 RowSet 对象的数据库游标，遍历整个结果记录集，并将记录集的数据输出。程序一旦运行，调用了 RowSetExam 类的构造函数后 ，马上就调用 go()函数，go()函数首先载入 JDBC 驱动程序，本例使用的 JDBC 驱动程序是 JDBC-ODBC 桥驱动程序，然后创建 EchoListener 类和 RowSet 接口的实例对象 echo 和 rs。rs 对象使用 addRowSetListener()方法将监听器 echo 对象与自身绑定，这样 echo 对象将监听 rs 对象上发生的任何事件，并自动响应。rs 对象接着调用 setXXX()方法设定与数据库建立连接的连接参数，本例中，使用的数据库是在第 9 章中建立的 fancy 数据库，JDBC URL 是 jdbc：odbc：test，用户名为 sa，密码为空，当这些连接参数都设定以后，使用 setCommand()方法向数据库发送 SQL 语句，然后使用 execute()方法，与数据库建立事实上的连接，执行 SQL 语句，并且使用数据库返回的结果数据填充 rs 对象，程序的最后将 rs 对象作为参数传递给自定义函数 scrollCursor()，从 scrollCursor()函数返回以后，关闭 rs 对象，程序结束。scrollCursor()函数执行的是移动数据库游标的操作，它的流程十分简单，仅仅简单地使用 while 循环结构遍历整个结果记录集，将结果输出，但在这个过程中，不断地引发 cursorMoved 事件，这些事件触发了 rs 对象的监听器——echo 对象，JDBC 数据库驱动程序会自动调用 EchoListener 类的 cursorMoved()方法，输出 Rowset cursor moved 的字样，rs 对象的数据库游标每向前移动一位，EchoListener 类的 cursorMoved()方法就会被调用一次。所以，这个程序的输出结果将会是每输出两个字段，就会出现 Rowset cursor moved 字样。

虽然程序清单 10.2 是 Java 应用程序，但是我们可以很容易地将它改写为 JSP/Servlet 程序，在改写程序清单 10.2 的同时，我们需要首先改写程序清单 10.1，因为 EchoListener 类的三个方法都使用了 System.out.prinln()方法输出信息，这无论对于客户端的程序调用者还是处于服务端的程序开发者来说，都没有任何意义 ，一种比较好的改写方法是将 EchoListener 类所有可能的输出都定向到某个 log 文件中，这样一来，运行程序清单 10.2

对 fancy 数据库所作的任何操作都会被记录下来，这些记录下来的信息对于整个数据库系统的安全运行有时候是十分有用的。

10.2.4　RowSetEvent 类

在 RowSetListener 接口中，定义了三个方法，分别是 cursorMoved()、rowChanged()、rowSetChanged()方法，这三个方法的参数都是 RowSetEvent 类的实例对象 event，RowSetEvent 类的对象代表了 RowSet 对象生命周期中所发生的各种各样的事件。在 RowSet 对象的生命周期中，会不断地引发很多事件，比较典型的事件有：数据库游标的移动，新记录的插入等等，每一个事件都对应着一个 RowSetEvent 对象，如果 RowSet 对象没有和任何一个监听器绑定，那么 JDBC 数据库驱动程序将会忽略这些事件，但是如果 RowSet 对象和某个监听器相绑定了，那么当 RowSet 对象产生了一个事件，对应的 RowSetEvent 对象也会被创建，JDBC 数据库驱动程序将会把 RowSetEvent 对象传递给监听者，监听者会根据 RowSetEvent 对象的类型，自动调用相应的方法对 RowSet 事件作出响应。

10.3　CachedRowSet 包

10.3.1　CachedRowSet 包简介

CachedRowSet 包就是前面提到的 sun.jdbc.rowset 包，CachedRowSet 包目前并不直接包含在 JDBC Standard Extension API 2.0 中，因此，它和 RowSet 包并不一起提供，Java Software 提供的 JDBC Standard Extension API 2.0 仅仅包含 RowSet 包，你不能在 Java Software 的网站上直接下载 CachedRowSet 包的 jar 文件，只有 JDC 的成员才可以下载使用 CachedRowSet 包，所以，如果你希望体验 CachedRowSet 包的强大功能，那么你必须首先申请成为 JDC 的成员，然后进入 JDC 组织的页面下载 CachedRowSet 包。

CachedRowSet 包有三个十分重要的类，分别是 CachedRowSet 类、JdbcRowSet 类、WebRowSet 类，CachedRowSet 类可以为瘦客户端提供处理数据库数据的方法，如果 JDBC 数据库驱动程序不支持记录集数据库游标的前后移动或者行更新，那么使用 CachedRowSet 类就可以达到你的目的。JdbcRowSet 类的主要作用是包装 JDBC Driver 和 ResultSet，使得它们可以作为一个可视化的 JavaBeans 组件被使用于 Windows 界面中。WebRowSet 类继承自 CachedRowSet 类，它的最重要的功能是将记录集中的数据以 XML 文件的形式输出，或者是从 XML 文件中读入数据并输入到数据库中。在本节，我们将会详细介绍这三个类的主要用法。在 CachedRowSet 包中，有一个抽象类 BaseRowSet，它是 CachedRowSet 类和 JdbcRowSet 类的直接父类，也是 WebRowSet 类的间接父类，也比较重要，10.3.2 节将介绍 BaseRowSet 类，然后，在接下来的几个小节中，再分别介绍余下的三个类。

10.3.2　BaseRowSet 类

BaseRowSet 类继承自 java.lang.Object，它是一个抽象类(abstract)，也就是说，在 BaseRowSet 类的定义中，它仅仅声明了成员方法名，没有实现成员方法的功能，这些功能需要在继承 BaseRowSet 类的类或者接口中实现。CachedRowSet 类和 JdbcRowSet 类就都实现了 BaseRowSet 类中声明的成员方法，所以它们都不是抽象类。

在 BaseRowSet 类中，主要定义了以下的成员方法和成员变量：

（1）设置或者获取 XXXRowSet 对象与数据库建立连接的连接参数。

首先需要声明一下，这里所说的 XXXRowSet 对象，指的是 CachedRowSet 类、JdbcRowSet 类、WebRowSet 类的实例对象，而不是指 javax.sql.RowSet 接口的实例对象，对于这一点，读者一定要特别注意。BaseRowSet 声明了 setPassword()、getPassword()、setUsernme()、getUsername()、setUrl()、getUrl()等方法名，这些方法主要用于指定或者获取 RowSet 对象与数据库建立连接的连接参数，类似的方法还有很多，例如：setType()、setTypeMap()、getType()、getTypeMap()等。读者应该不难发现，这些方法和 javax.sql..RowSet 接口中定义的方法一模一样，不过这两者之间似乎没有任何联系，BaseRowSet 类继承自 java.lang.Object，而 RowSet 接口继承自 java.sql.ResultSet 接口。

（2）XXXRowSet 对象的监听器。

BaseRowSet 类中，声明了 addRowSetListener() 方法和 removeRowSetListener()方法，这两个方法分别用于将一个事件监听器绑定在 XXXRowSet 对象上和解除这种绑定关系。javax.sql.RowSet 接口也定义了类似的方法。

（3）发送 SQL 命令和设定 SQL 命令的输入参数(IN 参数)。

BaseRowSet 类同样声明了 setCommand()、execute()方法分别用于向数据库系统发送 SQL 命令和执行 SQL 命令，并用数据库返回的数据填充 XXXRowSet 对象。BaseRowSet 类也声明了一整套的 setXXX()方法用于指定 SQL 命令的输入参数(IN 参数)，例如：setString()、setBlob()、setClob()、setObject()等方法。javax.sql.RowSet 接口也定义了十分类似的方法。

（4）流成员变量。

BaseRowSet 类还定义了若干个流类成员变量。用于保存 getXXXStream()方法返回的流类对象，这些成员变量如下所示：

protected java.io.InputStream asciiStream; 成员变量 asciiStream 用于保存 getAsciiStream()方法返回的 ASCII 流，这是 java.io.InputStream 类型的变量。

protected java.io.Reader charStream; 成员变量 charStream 用于保存 getCharacterStream()方法返回的 char 流，这是 java.io.Reader 类型的变量。

protected java.io.InputStream unicodeStream; 成员变量 unicodeStream 用于保存 getUnicodeStream()方法返回的 Unicode 流，这是 java，io.InputStream 类型的变量。

protected java.io.InputStream binaryStream; 成员变量 binaryStream 主要用于保存 getBinaryStream()方法返回的 binary 流，这是 java.io.InputStream 类型的变量。

10.3.3　CachedRowSet 类

CachedRowSet 类是 CachedRowSet 包中最重要的类。它继承自 BaseRowSet 类，CachedRowSet 对象为规范的数据提供了一个无连接的(disconnected)、可串行化的(serializable)、可以滚动的(scrollable，指数据指针可以前后移动)的容器。CachedRowSet 对象可以简单地看作是一个与数据库断开连接的结果记录集，它被缓存在数据源之外，因为所有的数据都被缓存在内存之中，所以 CachedRowSet 对象不适合于处理含有海量数据的记录集。

CachedRowSet 对象的重要作用是：它可以在作为数据容器，在不同应用程序的不同的组件之间传送数据。例如，一个运行于 Application Server 上的 Enterprise JavaBeans 组件可以使用 JDBC API 访问数据库，然后，可以使用 CachedRowSet 对象将数据库返回的数据通过网络发送到运行于客户端浏览器上的 Java Applet 程序或者 JavaBeans 组件。

如果客户端由于资源的限制或者出于安全上的考虑，没有办法使用 JDBC 数据库驱动程序，例如 PersonalJava Clients、Personal Digital Assistant (PDA)、Network Computer(NC)等客户端，这时，使用 CachedRowSet 类就可以提供一个 Java Client，用以处理数据库的规范数据。

CachedRowSet 类的第三个作用是：它可以通过使用数据源以外的空间缓存记录集的数据，从而不需要 JDBC 数据库驱动程序帮助，而实现了结果记录集的数据库游标的前后移动，还可以执行行更新的操作。CachedRowSet 对象在读入数据库的数据时，采用 get rows in 的机制，当它更新记录集的数据时，采用 get changed rows out 的机制。CachedRowSet 对象获取数据以后，就断开了和数据源的连接，只有执行行更新操作时，才再度与数据库建立连接。某些 JDBC 驱动程序目前仍然不支持结果记录集的数据库游标的前后移动，这时使用 CachedRowSet 类就可以实现你的愿望。

注意：如果你使用的 JDBC 数据库驱动程序是 JDBC-ODBC 桥驱动程序，用 ResultSet 接口、Statement 接口以通常的方法访问数据库，似乎无法实现记录集的数据库游标的前后移动，特别是不能向后移动，亦不能定位到任意行去，如果读者碰到了类似的问题，除了更换 JDBC 驱动程序以外，不妨使用 CachedRowSet 类。

下面我们详细介绍如何使用 CachedrowSet 类。

创建 CachedrowSet 类的实例对象

如果想使用 CachedRowSet 类的强大功能，那么你必须首先创建 CachedRowSet 类的实例对象。如何创建呢?可以使用 CachedRowSet 类的构造函数。CachedRowSet 类的构造函数是 CachedRowSet()，该函数没有任何参数。CachedrowSet()函数初始化了下面的属性值：

```
onInsertRow = false,
insertRow = null
cursorPos = 0
numRows = 0
showDeleted = false
queryTimeout = 0
maxRows = 0
maxFieldSize = 0
rowSetType = ResultSet.TYPE_SCROLL_INSENSITIVE
concurrency = ResultSet.CONCUR_READ_ONLY
readOnly = false
isolation = Connection.TRANSACTION_READ_COMMITTED
escapeProcessing = true
absolutePos = 0
```

新创建的 CachedRowSet 对象缺省可以容纳 100 个记录所包含的数据。

指定 CachedRowSet 对象和数据库建立连接的连接属性

创建了 CachedRowSet 对象，就可以使用 setPassword()、setUsername()、setUrl()等方法指定 CachedRowSet 对象和数据库建立连接的连接参数(当然了，需要首先载入 JDBC 驱动程序)。setPassword()setUsername()等方法都是在 BaseRowSet 类中被声明，然后在 CachedRowSet 类中实现了方法的功能。这里设定连接参数的过程和 javax.sql.RowSet 接口几乎一模一样，读者可以参考 10.2 节的内容。

使用记录集数据填充 CachedRowSet 对象

如何使用记录集的数据填充 CachedRowSet 对象呢?一般说来，有三种较为常用的方法。第一种方法和 javax.sql.RowSet 接口所使用的方法如出一辙，设定了 CachedRowSet 对象的数据库连接参数以后，调用 setCommand()方法指定 SQL 命令，再使用 setXXX()方法设定 SQL 命令的输入参数(如果有输入参数的话)，接着就可以使用 execute()方法，首先利用设定好的连接参数和数据库建立连接，发送 SQL 命令，执行 SQL 命令，获取数据库返回的数据，并用这些 数据填充 CachedrowSet 对象的内部记录集结构，注意：这种方法所使用的 execute()方法不带任何参数。第二种方法是：首先载入 JDBC 数据库驱动程序，然后与数据库建立连接，创建 Connection 接口的实例对象，接着用 setcommand()方法指定 SQL 命令，如果存在 SQL 输入参数，则可以使用 setXXX()方法指定 IN 参数，一切就绪后，就可以调用 execute()方法，请读者注意，此 execute()方法非彼 execute()方法，后面介绍的 execute()方法需要参数，参数就是 Connection 接口的实例对象，execute() 方法可以利用这个对象往数据库发送 SQL 命令，并用数据库服务器返回的数据填充 CachedRowSet 对象。两个 execute()方法的定义如下：

```
public void execute(java.sql.Connection connection);
public void execute();
```

使用记录集数据填充 CachedRowSet 对象的第三种方法是：首先载入 JDBC 数据库驱动程序，然后分别创建 Connection 接口的实例对象、Statement 接口的实例对象，接着调用 Statement 对象的 execute()方法，执行数据库操作，返回一个 ResultSet 接口的实例对象，然后就可以使用 CachedRowSet 类的 populate()方法将 ResultSet 对象的数据填充 CachedRowSet 对象内部的记录集结构。populate()方法接受 ResultSet 接口的实例对象为方法参数。populate()方法的定义如下：

```
public void populate(java.sql.ResultSet data)
```

下面是三个 JSP 代码段，向读者演示如何使用上述三种方法填充 CachedRowSet 对象内部的记录集结构(防错代码已经省略了，crs 是 CachedRowSet 类的实例对象)。

例：方法一

```
<%
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
crs.setUrl("jdbc:odbc:test");
crs.setUsername("sa");
crs.setPassword("");
```

```
crs.setCommand("SELECT * FROM goods");
crs.execute();
%>
```

例：方法二

```
<%
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn=DriverManager("jdbc:odbc:test","sa","");
crs.setCommand("SELECT * FROM goods");
crs.execute(conn);
%>
```

例：方法三

```
<%
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection conn=DriverManager("jdbc:odbc:test","sa","");
Statement stmt=conn.createStatement();
ResultSet rs=stmt.executeQuery("SELECT * FROM goods");
crs.populate(rs);
%>
```

在上面的三个方法中，以第三种方法最为常用，读者一定要特别熟悉这种方法。

获取 CachedRowSet 对象内部记录集结构的数据

在上面已经介绍了如何用数据库的数据填充 CachedRowSet 对象内部记录集结构，那么如何访问这些存储于 CachedRowSet 对象内部的数据呢?不用担心，CachedRowSet 类定义了一套 getXXX()方法可以获取 CachedRowSet 对象内部记录集结构的数据，如 getString()、getBlob()、getClob()等方法，这些方法的用法和 java.sql.ResultSet 接口 javax.sql.RowSet 接口的同名方法十分相似，我们在这里就不详细介绍了。读者如果对此感兴趣，可以参考 CachedRowSet 包的说明文档。CachedRowSet 类同样支持自定义 SQL 数据类型，至于如何支持自定义的 SQL 数据类型，读者可以参考第 9 章的相关内容。

CachedRowSet 类可以将内部记录集结构的数据拷贝到 RowSet 对象中去，有两个方法可以实现这个功能，分别如下所示：

```
public javax.sql.RowSet createCopy();
public javax.sql.RowSet createShared();
```

这两个方法都返回 RowSet 接口的实例对象，初初看起来，这两种方法似乎没有什么差别，不就是将 CachedRowSet 对象的内部数据备份到 RowSet 对象中去吗?其实这两个方法有很大的差别，第一个方法仅仅是将 CachedRowSet 对象的内部数据备份到 RowSet 对象中，如果后来 CachedRowSet 对象所包含的数据发生了变化，例如某个记录的数据被更新或者被删除了，但是 RowSet 对象不会受到任何的影响，即 RowSet 对象内部记录集的数据保持原样，没有发生变化。createShare()方法同样将 CachedRowSet 对象内部记录集的数据复制到 RowSet 对象中去，这两个对象的数据是共享的。如果 CachedRowSet 对象修改了内部记录集的数据，那么 RowSet 对象内部记录集的数据也会同步发生变化。

CachedRowSet 对象除了可以将内部记录集的数据复制到 RowSet 对象上，还可以将数据复制到 Collection 对象上，这需要使用 toCollection()方法，该方法的定义如下：

public java.util.Collection toCollection();

public java.util.Collection toCollection(int column);

上面的两个方法同名，都返回一个 java.util.Collection 对象，第一个方法返回的 Collection 对象含有 CachedRowSet 对象内部记录集的全部数据，但是第二个方法需要参数 column(整型)，这个方法仅仅将 CachedrowSet 对象内部记录集的第 cloumn 列的数据返回，并赋给 Collection 对象，而不是返回全部的数据。

在记录集中定位

java.sql.ResultSet 接口、javax.sql.RowSet 接口、CachedRowSet 类都可以用 first()、last()、relative()、absolute()、next()、previous()等方法在数据库记录集中任意定位。但是由于某些较老的 JDBC 数据库驱动程序不支持数据库游标的前后移动，如果程序员使用 ResultSet 接口或者 RowSet 接口的上述方法，就会出现异常，因为这两个接口的实例对象和数据库一直保持着连接状态，而 JDBC 驱动程序又不支持前后滚动数据库游标的，这样就会返回“数据库驱动程序不支持 XXX 操作”的错误。

在 CachedRowSet 类出现以前，程序员有三种解决方法：第一种方法是改用其他的技术，例如 ASP 的 ADO 技术，这样整个平台的构架将要改变，以前所作的一切都要推倒重来，很显然，这样做的代价太大，不到万不得已的情况，一般不会采用这种方法。第二种方法是将记录集的数据转化为某个 Java 数据结构类的实例对象，利用这些数据结构类的方法实现数据指针的滚动，例如可以将 ResultSet 对象的数据都复制到 java.util.Vector 类或者 java.util.LinkedList 类中，这样就可以任意获取某个特定的记录数据了，不过，当数据库记录集的容量很大的情况下，例如 1 万条记录，使用这种方法的花销就太大，所以，这种方法用的人也比较少。第三种方法是更换 JDBC 数据库驱动程序，这种方法最方便，也最有效，对程序代码几乎没有任何影响，能够使用这种方法自然最好了，不过万一找不到合适的数据库驱动程序，那又该如何是好呢?虽然目前的主流数据库系统都有好几种驱动程序可供选用，但是如果你使用的数据库系统并不十分流行，可以使用的 JDBC 数据库驱动程序仅有一种，没有任何选择的余地，那可真是无计可施。现在各位读者不必担心了，因为我们可以使用 CachedRowSet 类达到我们目的，轻松实现滚动数据库游标的功能。

因为 CachedRowSet 对象一旦从数据库中获取数据，就和数据库断开了连接，暂时处于一种无连接(disconnected)的状态，它把记录集的数据都缓存在数据库系统以外的内存空间中，可以不依靠特定的 JDBC 数据库驱动程序而实现滚动数据库游标的功能，任意定位于记录集的任意行。

下面的程序清单 10.3(CachedRowSetExam1.java)演示了如何使用 CachedRowSet 类和一个不支持游标前后移动的 JDBC 数据库驱动程序，实现在记录集中的任意定位。

程序清单 10.3

```
//File Name：CachedrowSetExam1.java
//Author：fancy
//Date：2001.2.26
```

```
//Note：to show    how to scroll the data

import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;

public class CachedRowSetExam1
{
    public static void main(String args[])
    {
        Connection conn; // Hold the connection to the database
        Statement stmt;
        ResultSet rs;

        // Register the driver.
        try
        {
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        }
        catch (ClassNotFoundException ex)
        {
            System.err.println("ClassNotFoundException：  " +
                ex.getMessage());
        }

       // Connect to the database.
            // NOTE：  this will *not* work as is. You will need to
            // supply userid，password，machine and database in the
            // appropiate place before you can connect
        try
        {
            conn = DriverManager.getConnection("jdbc：odbc：test"，"sa"，"");
            stmt = conn.createStatement();
            // clean up previous runs
            /*
            * now create the resultset that we will use to populate
            * the rowset.
            */
            rs = stmt.executeQuery("SELECT * FROM goods");
            System.out.println("Query executed");

            // create a new rowset and populate it...
            CachedRowSet crs = new CachedRowSet();
```

```
crs.populate(rs);
System.out.println("RowSet populated.");

/*
* Now that the rowset has been populated we can
* close the connection to the database.
*/
stmt.close();
System.err.println("stmt closed.");
conn.close();
System.err.println("conn closed.");

/*
* From this point on we are fetching from the rowset
*/
System.out.println("Fetching from RowSet...");

String v1;
String   v2;

while (crs.next())
{
      v1 = crs.getString(1);
      if (crs.wasNull() == false)
      {
            System.out.println("v1 is " + v1);
      }
      else
      {
            System.out.println("v1 is null");
      }

      v2 = crs.getString(2);
      if (crs.wasNull() == false)
      {
            System.out.println("v2 is " + v2);
      }
      else
      {
            System.out.println("v2 is null");
      }

}
```

北京希望电子出版社

```
if (crs.isAfterLast() == true)
{
        System.out.println("We have reached the end");
        System.out.println("This is row：  " + crs.getRow());
}

System.out.println("And now backwards...");

while (crs.previous())
{

    v1 = crs.getString(1);
    if (crs.wasNull() == false)
    {
        System.out.println("v1 is " + v1);
    }
    else
    {
        System.out.println("v1 is null");
    }

    v2 = crs.getString(2);
    if (crs.wasNull() == false)
    {
        System.out.println("v2 is " + v2);
    }
    else
    {
        System.out.println("v2 is null");
    }

}

if (crs.isBeforeFirst() == true)
    System.out.println("We have reached the start");

crs.first();
if (crs.isFirst() == true)
    System.out.println("We have moved to first");

System.out.println("This is row：  " + crs.getRow());
    if (crs.isBeforeFirst() == false)
```

北京希望电子出版社

```
                    System.out.println("We aren't before the first row.");

            crs.last();
                if (crs.isLast() == true)
                    System.out.println("...and now we have moved to the last");

                System.out.println("This is row：  " + crs.getRow());

                if (crs.isAfterLast() == false)
                    System.out.println("we aren't after the last.");
            }
            catch (SQLException ex)
            {
                System.err.println("SQLException：  " + ex.getMessage());
            }
        }
    }
```

程序清单 10.3(CachedRowSetExam1.java)的流程十分简单，它首先用常规的方法与数据库建立连接，然后使用 Statement 接口的 executeQuery()方法访问数据库，返回记录集对象，本例使用的 JDBC 数据库驱动程序是 JDBC-ODBC 桥驱动程序，一般情况下，不可能实现记录集数据的滚动访问，于是程序接着创建了一个 CachedRowSet 类的实例对象 crs，使用 populate()方法，将记录集对象(指 ResultSet 接口的实例对象 rs)的数据填充到 crs 的内部记录集结构中。为了证明 CachedRowSet 对象与数据库系统的无连接特性，程序接着关闭了数据库连接对象，实际上就是关闭了当前的数据库连接。然后程序使用 while 循环结构，使得数据库游标从记录集的头部，滚动到记录集的尾部，并输出每行前两列的数据。程序并没有到此终止，而是使用 while()循环结构，从记录集的尾部到头部又一次遍历记录集，输出每一行的前两列数据。接着调用 first()函数，定位到记录集的头部，再接着使用 last()方法，使得数据库游标又移动到记录集的尾部，程序这才结束。

注意：程序清单 10.3(CachedRowSetExam1.java)虽然是 Java 应用程序的运行模式，但是很容易将它改写为 JSP/Servlet 类型的程序。读者不妨试试看。其实，本章所介绍的知识并不针对特定类型的 Java 程序，无论是 Java Applet、Java Application 亦或者是 JSP、Servlet 程序甚至是 JavaBeans 组件，都可以使用 JDBC Standard Extension API，这当然也包括 CachedRowSet 包在内了。我们之所以要使用 Java Application 的形式向读者演示 CachedRowSet 类的功能，主要是因为手头上有这个例子，不用就太浪费了。

使用 CachedRowSet 对象更新记录集的数据

使用 CachedRowSet 对象可以更新记录集的数据，而无需 JDBC 数据库驱动程序的支持，CachedRowSet 对象更新记录集数据的方法和 ResultSet 接口、RowSet 接口的方法十分相似，但是稍有不同，不同在哪里呢?ResultSet 接口的实例对象若要更新记录，除了直接往

数据库发送 SQL 语句以外，另一种方法就是使用 updateXXX()方法指定新的值，再调用 updateRow()方法，完成更新数据库数据操作；若要插入新的数据，必须先调用 moveToInsertRow()方法，再调用 updateXXX()方法指定新纪录的值，最后调用 insertRow()方法，完成往数据库中插入记录的工作。CachedRowSet 对象执行插入新纪录或者更新某行的记录时，所用的步骤和上面的步骤几乎一模一样，只是程序的最后必须调用 acceptChanges()方法。

因为 CachedRowSet 对象和数据库是无连接的，它的记录集的数据缓存在数据库系统以外的内存空间中，即使调用了 insertRow()、updateRow()、deleteRow()等方法，对实际数据库中的数据却没有任何的改变，被改变的只是被缓存的记录集中的数据，这些改变可以被当前的数据库用户觉察到，但实际数据库中的数据并没有改变，所以需要调用 acceptChanges()方法，使得实际数据库中的数据和被缓存的记录集中的数据保持同步。

acceptChanges()方法的定义如下：

```
public void acceptChanges();
public void acceptChanges(java.sql.Connection con);
```

第一个 acceptChanges()方法方法不接受任何输入参数，第二个 acceptChanges()方法需要 Connection 接口的实例对象作为输入参数。什么时候使用第一种方法?什么时候又使用第二种 acceptChanges()方法呢?这和 CachedRowSet 对象内部记录集数据的填充方式有关，如果这些数据是通过调用 CachedRowSet 类的 execute()方法(指不使用输入参数的 execute()方法)而填充的，那么应该使用第一种 acceptChanges()方法，如果数据是通过执行 CachedRowSet 类的 populate()或者 execute()方法(需要输入参数的 execute()方法)而填充的，那么应该使用第二个 acceptChanges()方法。

请看下面的 JSP 代码段，它们演示了 acceptChanges()方法的用法。

例：

```
<%
crs.moveToInsertRow();
crs.updateString("Name"，"Shakespeare");
crs.updateInt("ID"，10098347);
crs.updateShort("Age"，58);
crs.updateInt("Sal"，150000);
crs.insertRow();
crs.moveToCurrentRow();
…
crs.acceptChanges();
%>
```

下面的程序清单 10.4(CachedRowSetExam2.java)是一个比较综合的例子，它全面展示了 CachedRowSet 类的各种各样重要的用法，包括如何在记录集中前后移动数据库游标，和如何更新特定记录的数据，这个例子几乎概括了本节的主要内容，读者如果完全掌握了这个程序的编写技巧，那么你对 CachedRowSet 类的用法也就掌握了一大半。

程序清单 10.4

```
//File Name：CachedRowSetExam2.java
```

```
//Author：fancy
//Date：2001.2.27
//Note：show the using of CachedRowSet class

import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;

    public class CachedRowSetExam2
    {

        public static void main(String args[])
        {
            CachedRowSetExam2 crs = new CachedRowSetExam2();
            try
            {
                Class.forName("sun.jdbc.odbc.JdbcOdbc.Driver");
                crs.go(false);
                crs.go(true);
            }
            catch (SQLException ex)
            {
                System.err.println("SQLException: " + ex.getMessage());
            }
        }

        private void go(boolean populated) throws SQLException
        {
            CachedRowSet crs;

            // create a new row set
            crs = new CachedRowSet();

            // set some properties of the rowset
            crs.setUrl("jdbc：odbc：test");
            crs.setUsername("sa");
            crs.setPassword("");
            crs.setCommand("SELECT * FROM goods");
            crs.setTableName("goods");

            crs.execute();

            if (populated == false)
```

```
        {
            // insert into the rowset
            populateRowset(crs);
            System.err.println("Rowset populated");


            System.err.println("Transfering changes to database...");
            crs.acceptChanges();
            System.err.println("Changes transfered.");
        }
        else
        {
            scrollCursor(crs);
        }

}


private void scrollCursor(CachedRowSet crs) throws SQLException
{
    System.out.println("Fetching from RowSet...");

    String v1;
    String   v2;

    while (crs.next())
    {

        v1 = crs.getString(1);
        if (crs.wasNull() == false)
        {
            System.out.println("v1 is " + v1);
        }
        else
        {
            System.out.println("v1 is null");
        }

        v2 = crs.getString(2);
        if (crs.wasNull() == false)
        {
            System.out.println("v2 is " + v2);
        }
```

北京希望电子出版社

```
        else
        {
            System.out.println("v2 is null");
        }

    }

    if (crs.isAfterLast() == true)
    {
        System.out.println("We have reached the end");
        System.out.println("This is row：  " + crs.getRow());
    }

    System.out.println("And now backwards...");

    while (crs.previous())
    {

        v1 = crs.getString(1);
        if (crs.wasNull() == false)
        {
            System.out.println("v1 is " + v1);
        }
        else
        {
            System.out.println("v1 is null");
        }

        v2 = crs.getString(2);
        if (crs.wasNull() == false)
        {
            System.out.println("v2 is " + v2);
        }
        else
        {
            System.out.println("v2 is null");
        }

    }

    if (crs.isBeforeFirst() == true)
    {
        System.out.println("We have made it back to the start!");
```

北京希望电子出版社

```
            }

            crs.first();
            if(crs.isFirst() == true)
                System.out.println("We have moved to first");

            System.out.println("This is row：  " + crs.getRow());

            if (crs.isBeforeFirst() == false)
                System.out.println("We aren't before the first row.");

            crs.last();
            if (crs.isLast() == true)
            System.out.println("...and now we have moved to the last");

            System.out.println("This is row：  " + crs.getRow());

            if (crs.isAfterLast() == false)
            System.out.println("we aren't after the last.");
        }

        private void populateRowset(CachedRowSet crs) throws SQLException
        {
            int i;

            for (i=0; i < 10 ; i++)
            {
                crs.moveToInsertRow();
                crs.updateString(1，"How to programing with JSP");
                crs.updateString(2，"books");
                crs.insertRow();
                crs.moveToCurrentRow();
            }

    }

}
```

程序清单 10.4(CachedRowSetExam2.java) 的流程基本和程序清单 10.3(CachedRowSetExam1.java)相同，不同的是程序清单 10.4 中新定义了 populateRowset()方法，这个方法往数据库中插入十条新记录，在方法 populateRowset() 方法体内，没有调用 acceptChanges()方法，当程序从 populateRowset()方法中返回，回到程序的主运行逻辑时，才调用 acceptChanges()方法，对数据库中的数据做出更新，这样作的效率比每插入一个新

的记录就调用一次 acceptChanges()方法要高效得多，建议读者采用这种方法更新数据库中的数据。

10.3.4　JdbcRowSet 类

JdbcRowSet 类是 CachedRowSet 类的姊妹类，它们都是继承自 BaseRowSet 类，不过用法上却有很大的不同，CachedRowSet 对象和数据库之间不会保持长久的连接状态，数据都缓存在内存空间中，使用 updateXXX()方法和 updateRow()等方法都不会直接修改数据库中的数据，但是 JdbcRowSet 类就有所不同了，它的实例对象和数据库保持着连接，而且是否可以执行记录集行更新，或者在记录集中前后移动数据库游标这些操作都依赖于 JDBC 技术认可的数据库驱动程序。JdbcRowSet 类的主要作用就是使得 JDBC API 和 JavaBeans 嵌合在一起，在这一点上，JdbcRowSet 类和 javax.sql.RowSet 接口的用法可以说是基本一样，可以互相换用。

JdbcRowSet 对象的创建方法和 RowSet 对象的创建方法一样，请看下面的 JSP 代码段。

例：

```
<%
JdbcRowSet jrs = new JdbcRowSet();
jrs.setCommand("SELECT * FROM goods WHERE    name LIKE '?' ");
jrs.setURL("jdbc：odbc：test");
jrs.setUsername("sa");
jrs.setPassword("");
jrs.setString(1，"小楼一夜听春雨");
jrs.execute();
%>
```

JdbcRowSet 类也定义了 exexute()方法，这个方法将执行三个步骤：

- 根据 setUrl()、setUsername()、setPassword()这些方法设定的数据库连接参数，创建 rowset 和数据库之间连接，即实例化 JdbcRowSet 类的内部私有变量 Connection 对象。
- 实例化内部私有变量 PreparedStatement 对象，执行带有输入参数的 SQL 语句。
- 实例化内部私有变量 ResultSet 对象，将 JdbcRowSet 类的内部记录集结构填充数据库返回的数据。

实际上，除了 JdbcRowSet 类的 execute()方法如此之外，RowSet 接口和 CachedRowSet 类的 execute()方法的运行机理也大概如此。

关于 JdbcRowSet 类的用法，读者可以比照 RowSet 接口的用法，这里就不多说了。

10.3.5　WebRowSet 类

WebRowSet 类是 CachedRowSet 类的子类，它除了具有 CachedRowSet 类的所有功能，还具有一项十分特殊的功能，就是 WebRowSet 对象可以将记录集中的数据输出为 XML 文件，或者将保存于 XML 文件中的数据读入 WebRowSet 对象的内部记录集结构中。我们首先来看看 WebRowSet 类都定义了那些重要的方法：

public WebRowSet(); 这是 WebRowSet 类的缺省构造函数，它初始化了内部的 CachedRowSet 对象和一个 XmlReaderImpl 对象、XmlWriterImpl 对象 ，其中 CachedRowSet 对象用于执行数据库操作，XmlReaderImpl 对象、XmlWriterImpl 对象分别用于读取和写入 XML 文件数据。

public static void writeXml(java.sql.ResultSet rs，java.io.Writer writer); 该方法可以将一个记录集对象中包含的数据写入到某个 XML 文件中。参数 rs 代表记录集对象，参数 writer 代表对目标 XML 的写入数据流。这个方法该如何使用呢?请看下面的 JSP 代码段(数据库连接过程已经省略，wrs 是 WebRowSet 类的实例对象，rs 是 ResultSet 接口的实例对象)

例：

```
<%
java.io.FileWriter FW = new java.io.FileWriter("fancy.xml");
wrs.writeXml(rs，FW);
response.sendRedirect("fancy.xml");
%>
```

在上面的代码段中，首先将记录集的数据写入到 fancy.xml 文件中，然后程序将当前的页面重定向到 fancy.xml，向客户端的用户输出完整的结果。

public void writeXml(java.io.Writer writer); 本方法和上面的方法同名但是参数不同，本方法仅有 writer 参数，而没有 rs 参数，这个方法的作用是将当前 WebRowSet 对象内部包含的数据写入到某个 XML 文件中。

public void readXml(java.io.Reader reader); 使用这个方法可以从某个 XML 文件中读入数据到 WebRowSet 对象的内部记录集结构中，一旦调用了这个方法，就可以像访问数据库记录集一样，使用 getXXX()方法获取 XML 文件中记录的数据。与此同时，也可以使用 updateXXX()方法更新这些数据，不过此时不需要使用 acceptChanges()方法了，因为原始数据保存于 XML 文件中，而非数据库中，仅仅需要使用 writeXml()方法就可以更新原 XML 文件的数据，下面的 JSP 代码段就演示了这一过程(wrrs 是 WebRowSet 类的实例对象)。

例：

```
<%
java.io.FileReader FW = new java.io.FileReader("fancy.xml");
wrs.readXml(FW);
out.println("Now print the default data of fancy.xml")
while(wrs.next())
{
      out.println(wrs.getString(1)+"<br>");
}
out.println("Now change the data of   fancy.xml");

wrs.first();
while(wrs.next())
{
      wrs.updateString(1，"the data had been changed by fancy，HaHa");
}
```

```
java.io.FileWriter FW = new java.io.FileWriter("fancy.xml");
wrs.writeXml(rs，FW);
response.sendRedirect("fancy.xml");
%>
```

在上面的 JSP 代码段中，首先从 fancy.xml 文件中读入数据并填充 WebRowSet 对象的内部记录集结构，然后遍历这个记录集，将第一列的数据输出，接着程序依次更新记录集的数据，最后将更新后的记录集的数据写回到原来的 fancy.xml 文件中，完成源数据的更新，并且令页面重定向到 fancy.xml 文件，使客户可以看到程序运行的结果。

public void setXmlReader(XmlReader reader); 使用这个方法可以将一个特定的 XmlReader 对象绑定到当前的 WebRowSet 对象中，在 WebRowSet 类 的缺省构造函数中，已经指定了缺省的 XmlReaderImpl 对象，用于读取 XML 文件的数据。

public void setXmlWriter(XmlWriter writer); 使用这个方法可以将一个特定的 XmlWriter 对象绑定到当前的 WebRowSet 对象中，在 WebRowSet 类 的缺省构造函数中，已经指定了缺省的 XmlWriterImpl 对象，用于写入数据到目标 XML 文件中。

public XmlReader getXmlReader(); 使用这个方法可以获取和当前 WebRowSet 对象绑定在一起的 XmlReader 对象。

public XmlWriter getXmlWriter(); 使用这个方法可以获取和当前 WebRowSet 对象绑定在一起的 XmlWriter 对象。

WebrowSet 类的主要方法就以上这些，下面，程序清单 10.5(WebRowSetExam.java) 将演示如何使用 WebRowSet 类的强大功能操作 XML 文件。

程序清单 10.5

```
//File Name：WebRowSetExam.java
//Author：fancy
//Date：2001.3.1
//Note：to    show how to operate the XML file with WebRowSet object

import java.sql.*;
import javax.sql.*;
import sun.jdbc.rowset.*;
import java.io.*;


public class WebRowSetExam
{

    public static void main(String args[])
    {
        WebRowSetExam    rs = new WebRowSetExam();
        try
        {
            rs.go();
```

```
        }
        catch (SQLException ex)
        {
            System.err.println("SQLException: " + ex.getMessage());
        }
    }

    private void go() throws SQLException
    {
        WebRowSet wrs;

        // create a new row set
        wrs = new WebRowSet();

        // set some properties of the rowset
        wrs.setUrl("jdbc：odbc：test");
        wrs.setUsername("sa");
        wrs.setPassword("");

        wrs.setCommand("select col1，col2 from test_table");

        // populate the rowset.
        wrs.execute();
        System.out.println("RowSet populated.");

        try
        {
            java.io.FileWriter FW = new java.io.FileWriter("Example6.xml");

            scrollCursor(wrs);
            wrs.writeXml(FW);

            java.io.FileReader FR = new java.io.FileReader("fancy1.xml");

            WebRowSet wrs2 = new WebRowSet();
            wrs2.readXml(FR);

            java.io.FileWriter FW2 = new java.io.FileWriter("fancy2.xml");

            wrs2.writeXml(FW2);

        }
        catch (Throwable ex)
```

北京希望电子出版社

```
        {
            System.out.println(ex.getMessage());
        }
    }

    private void scrollCursor(WebRowSet wrs) throws SQLException
    {
        System.out.println("Fetching from RowSet...");

        String v1;
        String v2;

        wrs.beforeFirst();

        while (wrs.next())
        {

            v1 = wrs.getString(1);
            if (wrs.wasNull() == false)
            {
                System.out.println("v1 is " + v1);
            }
            else
            {
                System.out.println("v1 is null");
            }

            v2 = wrs.getString(2);
            if (wrs.wasNull() == false)
            {
                System.out.println("v2 is " + v2);
                if (v2 == 2)
                {
                    System.out.println("Updating!");
                    wrs.updateInt(4，9999);
                    wrs.updateString(1，"WPS2000");
                    wrs.updateRow();
                    wrs.deleteRow();

                    wrs.moveToInsertRow();
                    wrs.updateInt(4，50);
                    wrs.updateString(1，"Microsoft Word 2000");
                    wrs.insertRow();
```

北京希望电子出版社

```
                    wrs.moveToCurrentRow();
                    wrs.next();
                    wrs.updateString(1， "DosMe");
                    wrs.updateRow();
                    wrs.previous();
                }
            }
            else
            {
                System.out.println("v2 is null");
            }

        }
        if (wrs.isAfterLast() == true)
        {
            System.out.println("We have reached the end");
            System.out.println("This is row：  " + wrs.getRow());
        }

        System.out.println("And now backwards...");

        while (wrs.previous())
        {

            v1 = wrs.getString(1);
            if (wrs.wasNull() == false)
            {
                System.out.println("v1 is " + v1);
            }
            else
            {
                System.out.println("v1 is null");
            }

            v2 = wrs.getString(2);
            if (wrs.wasNull() == false)
            {
                System.out.println("v2 is " + v2);
            }
            else
            {
                System.out.println("v2 is null");
```

北京希望电子出版社

```
            }

        }

        if (wrs.isBeforeFirst() == true)
        {
            System.out.println("We have made it back to the start!");
        }

            wrs.first();
            if (wrs.isFirst() == true)
                    System.out.println("We have moved to first");

            System.out.println("This is row：  " + wrs.getRow());

            if (wrs.isBeforeFirst() == false)
                System.out.println("We aren't before the first row.");

            wrs.last();
            if (wrs.isLast() == true)
                System.out.println("...and now we have moved to the last");

            System.out.println("This is row：  " + wrs.getRow());

            if (wrs.isAfterLast() == false)
                System.out.println("we aren't after the last.");
    }
}
```

程序清单 10.5(WebRowSetExam.java) 几乎涵括了 WebRowSet 类的强大功能，它的主要运行流程是，首先建立数据库连接，使用数据库返回的数据填充 WebRowSet 对象的内部记录集结构，接着调用自定义的 scrollCursor()函数演示数据库游标的前后移动，从 scrollCursor()函数返回以后，创建一个新的 java.io.Reader 对象，利用这个对象和 writeXml()方法往 XML 文件(fancy1.xmlt 对象内部记录集结构所包含的数据，然后，程序又使用 readXml()方法，从刚才的 XML 文件中(fancy1.xml)读入数据到 WebRowSet 对象的内部记录集对象中去，接着再次使用 writeXml()方法把数据写入到另外的 XML 文件中(fancy2.xml)。

注意：如果希望正确编译运行程序清单 10.5，你必须将 org.xml 包加入到编译路径中去，即添加 JAXP 支持，否则程序将可以编译，但是不能运行。

程序清单 10.5 编译后运行产生的 XML 文件如程序清单 10.6。

程序清单 10.6(部分内容有删节)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE RowSet PUBLIC '-//Sun Microsystems，Inc.//DTD RowSet//EN' 'http：//java.sun.com/
```

```
        j2ee/dtds/RowSet.dtd'>

<RowSet>
<properties>
        <command><null/></command>
        <concurrency>1007</concurrency>
        <datasource><null/></datasource>
        <escape-processing>true</escape-processing>
        <fetch-direction>0</fetch-direction>
        <fetch-size>0</fetch-size>
        <isolation-level>2</isolation-level>
        <key-columns>
        </key-columns>
        <map></map>
        <max-field-size>0</max-field-size>
        <max-rows>0</max-rows>
        <query-timeout>0</query-timeout>
        <read-only>true</read-only>
        <rowset-type>1004</rowset-type>
        <show-deleted>false</show-deleted>
        <table-name><null/></table-name>
        <url><null/></url>
</properties>
<metadata>
        <column-count>6</column-count>
        <column-definition>
                <column-index>1</column-index>
                <auto-increment>false</auto-increment>
                <case-sensitive>false</case-sensitive>
                <currency>false</currency>
                <nullable>1</nullable>
                <signed>false</signed>
                <searchable>true</searchable>
                <column-display-size>100</column-display-size>
                <column-label>goodsname</column-label>
                <column-name>goodsname</column-name>
                <schema-name></schema-name>
                <column-precision>100</column-precision>
                <column-scale>0</column-scale>
                <table-name></table-name>
                <catalog-name></catalog-name>
                <column-type>12</column-type>
                <column-type-name>varchar</column-type-name>
```

```
 </column-definition>
 <column-definition>
      <column-index>2</column-index>
      <auto-increment>false</auto-increment>
      <case-sensitive>false</case-sensitive>
      <currency>false</currency>
      <nullable>1</nullable>
      <signed>false</signed>
      <searchable>true</searchable>
      <column-display-size>100</column-display-size>
      <column-label>goodstype</column-label>
      <column-name>goodstype</column-name>
      <schema-name></schema-name>
      <column-precision>100</column-precision>
      <column-scale>0</column-scale>
      <table-name></table-name>
      <catalog-name></catalog-name>
      <column-type>12</column-type>
      <column-type-name>varchar</column-type-name>
</column-definition>
<column-definition>
      <column-index>3</column-index>
      <auto-increment>false</auto-increment>
      <case-sensitive>false</case-sensitive>
      <currency>false</currency>
      <nullable>1</nullable>
      <signed>false</signed>
      <searchable>true</searchable>
      <column-display-size>100</column-display-size>
      <column-label>comment</column-label>
      <column-name>comment</column-name>
      <schema-name></schema-name>
      <column-precision>100</column-precision>
      <column-scale>0</column-scale>
      <table-name></table-name>
      <catalog-name></catalog-name>
      <column-type>12</column-type>
      <column-type-name>varchar</column-type-name>
</column-definition>
<column-definition>
      <column-index>4</column-index>
      <auto-increment>false</auto-increment>
      <case-sensitive>false</case-sensitive>
```

北京希望电子出版社

```
            <currency>false</currency>
            <nullable>1</nullable>
            <signed>true</signed>
            <searchable>true</searchable>
            <column-display-size>24</column-display-size>
            <column-label>price</column-label>
            <column-name>price</column-name>
            <schema-name></schema-name>
            <column-precision>15</column-precision>
            <column-scale>0</column-scale>
            <table-name></table-name>
            <catalog-name></catalog-name>
            <column-type>6</column-type>
            <column-type-name>float</column-type-name>
        </column-definition>
        <column-definition>
            <column-index>5</column-index>
            <auto-increment>false</auto-increment>
            <case-sensitive>false</case-sensitive>
            <currency>false</currency>
            <nullable>1</nullable>
            <signed>true</signed>
            <searchable>true</searchable>
            <column-display-size>24</column-display-size>
            <column-label>priceoff</column-label>
            <column-name>priceoff</column-name>
            <schema-name></schema-name>
            <column-precision>15</column-precision>
            <column-scale>0</column-scale>
            <table-name></table-name>
            <catalog-name></catalog-name>
            <column-type>6</column-type>
            <column-type-name>float</column-type-name>
        </column-definition>
        <column-definition>
            <column-index>6</column-index>
            <auto-increment>true</auto-increment>
            <case-sensitive>false</case-sensitive>
            <currency>false</currency>
            <nullable>0</nullable>
            <signed>true</signed>
            <searchable>true</searchable>
            <column-display-size>11</column-display-size>
```

北京希望电子出版社

```
            <column-label>id</column-label>
            <column-name>id</column-name>
            <schema-name></schema-name>
            <column-precision>10</column-precision>
            <column-scale>0</column-scale>
            <table-name></table-name>
            <catalog-name></catalog-name>
            <column-type>4</column-type>
            <column-type-name>int identity</column-type-name>
        </column-definition>
</metadata>
<data>
    <row>
        <col>JSP入门</col>
        <col>书籍</col>
        <col>介绍JSP技术的入门书籍，简单易学</col>
        <col>30.2</col>
        <col>0.8</col>
        <col>1</col>
    </row>
    <row>
        <col>结构化学基础</col>
        <col>书籍</col>
        <col>北京大学化学系本科教材，免费赠送</col>
        <col>0.0</col>
        <col>0.8</col>
        <col>7</col>
    </row>
    <row>
        <col>万水青山踏遍</col>
        <col>兵器</col>
        <col>曾是白云城主的配剑，唯有此剑，方能使出天外飞仙的剑招</col>
        <col>1300.96</col>
        <col>0.8</col>
        <col>11</col>
    </row>
    <row>
        <col>小楼一夜听春雨</col>
        <col>兵器</col>
        <col>又名圆月弯刀，曾经是青青所用之物，后归丁鹏所有，和万水青山踏遍齐名
            </col>
        <col>3000.8</col>
        <col>0.8</col>
```

北京希望电子出版社

```
            <col>12</col>
        </row>
        <row>
            <col>笑傲江湖曲</col>
            <col>CD</col>
            <col>刘正风和曲洋联手创作的绝世名曲，天籁绝响，不可不听</col>
            <col>10.0</col>
            <col>0.8</col>
            <col>19</col>
        </row>
    </data>
    </RowSet>
```

程序清单 10.6(fancy1.xml)是由 WebRowSetExam.java 程序自动生成的 XML 文件之一，另外一个文件是 fancy2.xml，内容和 fancy1.xml 一模一样，这里就不列出了。fancy1.xml 文件的结构大致可以分为三个部分，第一部分是 WebRowSetRowSet 对象的信息，这部分内容由标记<properties></properties>包起来。第二部分内容由标记<metadata>和</metadata>包起来，主要描述数据库表各个字段的详细信息。第三部分就是记录集数据的具体内容，这部分内容被标记<data>和</data>包起来，每一行的内容由标记<row>和</row>包起来，每一列的内容由<col>标记和</col>标记包起来。关于 fancy1.xml 文件，有两个需要特别注意的地方，请找到如下行：

```
<?xml version="1.0" encoding="UTF-8"?>
```

这一行定义了当前网页的字符编码为 UTF-8 格式，如果 XML 文件中含有中文，这样设定是不合适的，IE 浏览器将会报错，应该将它改为：

```
<?xml version="1.0" encoding="GB2312"?>
```

再找到如下行：

```
<!DOCTYPE RowSet PUBLIC '-//Sun Microsystems，Inc.//DTD RowSet//EN' 'http：//java.sun.com/
j2ee/dtds/RowSet.dtd'>
```

这一行定义了从哪个位置可以找到本 XML 文件所用到的 XML 标记的定义文件(DTD 文件)，这个文件的 URL 地址是 http：//java.sun.com/j2ee/dtds/RowSet.dtd，如果你的机器没有上网，但是安装了 CachedRowSet 包，那么在 CachedRowSet 包的安装目录下面会有 RowSet.dtd 文件(RowSet.dat 文件的详细内容请参考程序清单 10.7)，请将它拷贝到和 fancy1.xml 文件相同的目录下面，然后将上面的代码修改为：

```
<!DOCTYPE RowSet PUBLIC '-//Sun Microsystems，Inc.//DTD RowSet//EN'
'RowSet.dtd'>
```

一切就绪后，用 IE 浏览器打开 fancy1.xml 文件，将会出现如图 10.1 所示的效果。

程序清单 10. 7

```
<!ELEMENT RowSet (properties，metadata，data)>

<!ELEMENT properties (command，concurrency，datasource，
escape-processing，fetch-direction，fetch-size，isolation-level，key-columns，
map，max-field-size，max-rows，query-timeout，read-only，rowset-type，
```

```
show-deleted，table-name，url)>

<!ELEMENT command (#PCDATA)>
<!ELEMENT concurrency (#PCDATA)>
<!ELEMENT datasource (#PCDATA)>
<!ELEMENT escape-processing (#PCDATA)>
<!ELEMENT fetch-direction (#PCDATA)>
<!ELEMENT fetch-size (#PCDATA)>
<!ELEMENT isolation-level (#PCDATA)>
<!ELEMENT key-columns (column)*>
<!ELEMENT column (#PCDATA)>
<!ELEMENT map (type，class)*>
<!ELEMENT type (#PCDATA)>
<!ELEMENT class (#PCDATA)>
<!ELEMENT max-field-size (#PCDATA)>
<!ELEMENT max-rows (#PCDATA)>
<!ELEMENT query-timeout (#PCDATA)>
<!ELEMENT read-only (#PCDATA)>
<!ELEMENT rowset-type (#PCDATA)>
<!ELEMENT show-deleted (#PCDATA)>
<!ELEMENT table-name (#PCDATA)>
<!ELEMENT url (#PCDATA)>

<!ELEMENT metadata (column-count，(column-definition*))>

<!ELEMENT column-definition
(column-index，auto-increment，case-sensitive，currency，nullable，
signed，searchable，column-display-size，column-label，column-name，
schema-name，column-precision，column-scale，table-name，catalog-name，
column-type，column-type-name)>

<!ELEMENT column-count (#PCDATA)>

<!ELEMENT column-index (#PCDATA)>
<!ELEMENT auto-increment (#PCDATA)>
<!ELEMENT case-sensitive (#PCDATA)>
<!ELEMENT currency (#PCDATA)>
<!ELEMENT nullable (#PCDATA)>
<!ELEMENT signed (#PCDATA)>
<!ELEMENT searchable (#PCDATA)>
<!ELEMENT column-display-size (#PCDATA)>
<!ELEMENT column-label (#PCDATA)>
<!ELEMENT column-name (#PCDATA)>
```

北京希望电子出版社

```
<!ELEMENT schema-name (#PCDATA)>
<!ELEMENT column-precision (#PCDATA)>
<!ELEMENT column-scale (#PCDATA)>
<!ELEMENT catalog-name (#PCDATA)>
<!ELEMENT column-type (#PCDATA)>
<!ELEMENT column-type-name (#PCDATA)>

<!ELEMENT data (row*，ins*，del*，insdel*)*>

<!ELEMENT row (col|upd)*>
<!ELEMENT ins (col|upd)*>
<!ELEMENT del (col|upd)*>
<!ELEMENT insdel ((col|upd)*)>

<!ELEMENT col (#PCDATA)>
<!ELEMENT upd (#PCDATA)>
```



图 10.1

10.3.6　XML 操作相关类

在 CachedRowSet 包中，定义了若干个类和接口以帮助 WebRowSet 对象读取和写入 XML 文件，这些类和接口有 XmlReader 接口、XmlWriter 接口、XmlReaderImpl 类、XmlWriterImpl 类等，前面两个接口是抽象接口，分别定义了 readXML()方法和 writeXML()方法的名称，但是没有真正实现这两个方法的功能，XmlReaderImpl 类、XmlWriterImpl 类

分别继承了相应的接口，实现了相应的方法。

writeXML()方法和 readXML()方法的定义如下所示：

public void writeXML(WebRowSet caller，java.io.Writer writer);

public void readXML(WebRowSet caller，java.io.Reader reader);

这两个方法都需要 WebRowSet 对象和 java.io.Writer 对象作为参数，事实上，我们很少需要直接使用这两个方法读取数据或者写入数据到 XML 文件，读取或者写入数据到 XML 文件的途径一般是使用 WebRowSet 类的 readXml()方法和 writeXml()方法。当调用 readXml()方法和 writeXml()方法的时候，WebRowSet 对象在内部自动调用 XmlReaderImpl 类的 readXML()方法和 XmlWriterImpl 类的 writeXML()方法完成对应的功能。

如果你想单独使用 writeXML()方法或者 readXML()方法，请参考下面的 JSP 代码段。

例：

```
<%
java.io.Writer wr=new java.io.Writer("fancy.xml");
WebRowSet wrs=new WebRowSet();
wrs.populate(rs);
XmlWriterImpl xwi=new XmlWriterImpl();
xmi.writeXML(wrs，wr);
%>
```

例：

```
<%
java.io.Reader re=new java.io.Reader("fancy.xml");
WebRowSet wrs=new    WebRowSet();
XmlReaderImpl xri=new XmlReaderImpl();
xri.readXML(wrs，re);
while(wrs.next())
{
      out.println(wrs.getString(1)+"<br>");
}
%>
```

10.4　数据库连接缓冲池

一个数据库缓冲池指的是缓存于内存空间中的数据库物理连接，这些数据库连接可以被重复使用。数据库缓冲池对于提高 Java 数据库应用程序的性能十分重要，尤其是当这个 Java 数据库应用程序运行于中间层服务器环境时。

数据缓冲池存在于中间层服务器环境当中，可以被不同的 Java 应用程序所调用。javax.sql.RowSet 包添加了对缓冲数据源的支持，即可以将缓冲池缓冲的数据库连接看作是一个是实实在在的数据源服务来使用。RowSet 包提供了好几个接口用于处理数据库缓冲池，主要的接口有：

DataSource 接口，DataSource 接口的实例对象代表了存在于中间层服务器中的缓冲数据源服务。使用它可以返还数据库缓冲池中现存的数据库连接，DataSource 接口的实例对

象实际上是某个 JNDI 服务的提供者，在使用它之前，该 JNDI 服务对象必须先在中间层服务器环境中注册，并且和某个服务名绑定在一起，然后它才能被别的 Java 应用程序调用。

ConnectionPoolDataSource 接口，该接口可以用于创建一个被缓冲于缓冲池的数据库物理连接，它有可能会被 DataSource 接口的实例对象调用。

PooledConnection 接口，该接口代表被缓冲的数据库连接。它定义了一个 getConnection()方法，使用这个方法，可以返回 java.sql.Connection 接口的实例对象。

如何使用 RowSet 包定义的类和接口操作数据库缓冲池呢?下面的 JSP 代码段提供了一个简单的不完整的例子。

例：

```
<%
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup(“jdbc/EmployeeDB”);
// First，get a Connection. Connection pooling is done
// internally by the DataSource object.
Connection con = ds.getConnection(jdbc/webDatabase”，"sa”，“");
// Do all the work as a single transaction (optional).
con.setAutoCommit(false);
// The actual work (queries and updates) would go here.
// Work is done using standard JDBC code as defined in the
// rest of the JDBC API.
// Commit the transaction.
con.commit();
// Close the connection. This returns the underlying physical
// database connection to the pool.
con.close();
%>
```

上面的 JSP 代码段的运行机理如下：

- 首先，程序代码获取初始化的 JNDI 环境，并且调用 Context.lookup()方法，从 JNDI 服务提供者那里获一个 DataSource 对象。
- 中间层 JNDI 服务提供者返回一个 DataSource 对象给当前的 Java 应用程序，这个 DataSource 对象代表了中间层服务上现存的缓冲数据源。
- 应用程序调用 DataSource 对象的 getConnection()方法。
- 当 DataSource 对象的 getConnection()方法被调用时，中间层服务器将查询数据库连接缓冲池中有没有 PooledConnection 接口的实例对象，这个 PooledConnection 对象将被用于与数据库建立物理上的数据库连接。
- 如果在缓冲池中命中了一个 PooledCoonection 对象，那么连接缓冲池将简单地更新内部的缓冲连接队列，并将该 PooledConnection 对象返回。如果在缓冲池内没有找到现成的 PooledConnection 对象，那么 ConnectionPoolDataSource 接口将会被用来产生一个新的 PooledConnection 对象，并将它返回，以便应用程序使用。
- 中间层服务器调用 PooledConnection 对象的 getConnection()方法，以便返还一个

java.sql.Connection 对象给当前的 Java 应用程序。

- 当中间层服务器调用 PooledConnection 对象的 getConnection()方法时，JDBC 数据库驱动程序将会创建一个 Connection 对象，并且把它返回中间层服务器。
- 中间层服务器将 Connection 对象返回给应用程序。Java 应用程序可以认为这个 Connection 对象是一个普通的 JDBC Connection 对象，使用它可以和数据库建立事实上的连接，与数据库引擎产生交互操作。
- 当应用程序不需要使用 Connection 对象时，可以调用 Connection 接口的 close()方法，请注意，这种情况下，close()方法并没有关闭事实上的数据库连接，仅仅是释放了被应用程序占用的数据库连接，并将它还给数据库连接缓冲池，数据库连接缓冲池会自动将这个数据库连接交给请求队列中下一个的应用程序使用。

10.5　JNDI 和 RowSet

本节将简单地向读者介绍如何在 JDBC API 中使用 JNDI 技术，JNDI 技术提供了一种程序运行模式，使得 Java 应用程序可以通过网络发现和请求远端计算设备上的服务，如数据库服务等。

关于 JNDI 技术在 JSP 技术中的应用，在前面的相关章节已经有所介绍。本节将着重介绍 JNDI 技术如何与 JDBC 技术结合起来，构建强大的访问数据库的 Java 应用程序。究竟 JNDI 技术和 JDBC 技术的结合会有何好处呢?

（1）应用 JNDI 技术，可以实现真正的分布式处理，数据库服务器、提供数据库连接服务的组件、执行数据库操作的 Java 应用程序可以分布在不同地址的异质计算设备上。

（2）将 JNDI 技术与 JDBC 技术结合，可以实现数据库连接缓冲池的功能，请参考 10.4 节的介绍。

（3）将 JNDI 技术与 JDBC 技术结合，可以在 Java 应用程序中屏蔽掉很多繁琐的建立数据库连接的代码，这样有有助于系统的安全性。

JNDI 技术与 JDBC 技术的结合既然有那么多的优点，那么我们该如何做呢?RowSet 包提供了这方面的功能。JNDI 技术与 JDBC 技术结合的原理如下：

使用 JDBC 和 JNDI 技术，编写一个提供数据库服务的组件，然后将它出版到某个中间层服务器中，侦听服务请求。接下来我们可以编写 Java 应用程序，查找提供数据库服务的组件，一旦查找到它(组件)，就可以利用它提供的各种方法执行特定的数据库操作。RowSet 包定义的 DataSource 接口提供了建立数据库连接的 JNDI 支持。具体的方法如下：

（1）首先创建一个 Java 类 SampleDataSource，该类扩展了 javax..sql.DataSource 接口，具体实现了获取数据库连接的功能。该类的代码请参考程序清单 10.8(SampleDataSource.java)。

程序清单 10. 8

```
//File Name：SampleDataSource.java
//Author：fancy
//Date：2001.3.4
//Note：to create tht datasource connection
```

```
import java.sql.*;
import javax.sql.*;
import javax.naming.*;

public class SampleDataSource implements javax.sql.DataSource，
    javax.naming.Referenceable，java.io.Serializable
{
    /* Constructors*/
    public SampleDataSource()
    {
        // This constructor is needed by the object factory
    }
    /** Properties*/
    public String getServerName()
    {
        return serverName;
    }
    public void setServerName(String serverName)
    {
        this.serverName = serverName;
    }
    public String getDatabaseName()
    {
        return databaseName;
    }
    public void setDatabaseName(String databaseName)
    {
        this.databaseName = databaseName;
    }
    /*
    * Methods inherited from DataSource
    */
    public Connection getConnection() throws SQLException
    {
        //vendor specific code to create a JDBC Connection goes here
        try
        {
            Class.forName("jdbcDriver");
            Connection conn=DriverManager.getConnection("jdbcURL"，"user"，"pass");
            return conn;
        }
        catch(Exception fe)
        {
```

北京希望电子出版社

```
                //to do nothing
            }
            return null;
        }
        public Connection getConnection(String username，String password) throws SQLException
        {
            //vendor specific code to create a JDBC Connection goes here
            String user=username;
            String pass=password;
            try
            {
                Class.forName("jdbcDriver");
                Connection conn=DriverManager.getConnection("jdbcURL"，user，pass);
                return conn;
            }
            catch(Exception fe)
            {
                //to do nothing
            }
            return null;

        }
        public java.io.PrintWriter getLogWriter() throws SQLException
        {
            //vendor specific code goes here
        }
        public void setLogWriter(java.io.PrintWriter out) throws SQLException
        {
            //vendor specific code goes here
        }
        public void setLoginTimeout(int seconds) throws SQLException
        {
            //vendor specific code goes here
        }
        public int getLoginTimeout() throws SQLException
        {
            //vendor specific code goes here
        }

        private String serverName = null;
        private String databaseName = null;
}
```

在 SampleDataSource 类中，定义了 getConnection()方法，利用该方法，可以获取

北京希望电子出版社

java.sql.Connection 接口的实例对象，除了 getConnection()方法以外，SampleDataSource.java 程序还定义了若干个其他相关方法，为了节省篇幅，这些方法的定义实体都省略了。

（2）接下来，我们应该编写一个 Java 应用程序，将上面编写好的 SampleDataSource 组件出版到中间层服务器中去。假设这个应用程序名为 JNDIExam.java，那么它应该含有下面的代码段。

例：

```
SampleDataSource sds = new SampleDataSource();
sds.setServerName(“rainbow”);
sds.setDatabaseName("fancy");
Context ctx = new InitialContext();
ctx.bind(“jdbc/EmployeeDB”，sds);
```

上面的代码中，第一行创建了一个 SampleDataSource 对象，接下来的两行代码分别调用 setServerName()方法和 setDatabaseName()方法，初始化 SampleDataSource 对象的属性，然后创建 JNDI 命名环境对象 ctx，将 SampleDataSource 类的实例对象 sds 绑定到 JNDI 名 jdbc/EmployeeDB 中。

编译运行 JNDIExam.java 程序，将 JNDI 服务发布出去。除了使用这个方法，也可以使用某些 GUI 工具可视化出版 JNDI 服务。

（3）下一步，我们应该编写客户端(此处指相对的客户端，某台服务器相对于另一台服务器来说，可以是服务端，也可以是客户端)的 Java 应用程序，如 JSP 程序等，以便使用 JNDI 服务，请看下面的 JSP 代码段(try/catch 模块已经省略了)，关于这一段 JSP 代码的解释，读者可以参考 10.4 节的相关解释。

例：

```
<%
Context ctx = new InitialContext();
DataSource ds = (DataSource)ctx.lookup(“jdbc/EmployeeDB”);
Connection con = ds.getConnection(jdbc/webDatabase”，"sa”，“");
con.setAutoCommit(false);
Statement stmt=con.createStatement();
ResultSet rs=stmt.executeQuery("SELECT * FROM goods");
while(rs.next())
{
      out.println(rs.getString(1));
}
con.commit();
con.close();
%>
```

10.6　本章小结

本章主要向读者介绍了 JDBC Standard Extension API 2.0 的技术，以及如何将这种技术和 JNDI、JavaBeans、数据库连接缓冲池等技术融合在一起，编写强大的 Java 应用程序，

应该指出，本章介绍的技术不但可以用于 Java Application 而且还可以用于 Java Servlet 和 JSP 程序。本章所举的例子大部分基于 Java Application，其实可以很容易将它们改写为 JSP/Servlet 程序，关于这一点，读者可以自己试一试。

在第 11 章，我们将向读者介绍 JSP 网络应用程序的编写方法，即如何编写 JSP 程序访问 FTP、SMTP、News 等网络服务。

北京希望电子出版社

第四部分　JSP 网络程序设计

第 11 章　JSP 网络程序开发

本章将向读者介绍如何编写访问 FTP、SMTP、News 等服务的 JSP 程序。在 Internet 中，除了 WWW 服务以外，FTP、SMTP、News 等服务同样十分重要，我们如果想访问这些服务，一般需要通过标准的客户端程序。但是，Java 语言最擅长的就是编写网络应用程序了，因此我们可以使用 Java 网络类库，编写 JSP 程序来访问这些服务。

本章的内容要点如下：

- 配置 Mail 和 FTP 务器
- 访问 SMTP 服务
- 访问 FTP 服务
- 访问 News 服务
- 使用底层的 Socket

我们也可以编写 Java 应用程序来访问 Telnet 服务，但是 Telnet 服务要求客户端与服务端有着持续不断的网络连接，这在 JSP 程序中实现有一定的难度，比较复杂，为了减小读者阅读上的困难，我们将这部分内容删除了。对此感兴趣的读者可以参考相关的文档。

11.1　配置服务器

在本节中，我们将向读者介绍如何把自己的 PC 机配置成为 Mail 服务器和 FTP 服务器，因为在下面的编写程序中，有很多实例都需要用到 Mail 服务器和 FTP 服务器。其实我们也可以使用外部的 Mail 服务器和 FTP 服务器，不过这样做实在太慢，而且被管理员发现就糟了，所以我们还是自己配一个服务器比较好。

11.1.1　配置 Mail 服务器

本小节介绍如何配置 Mail 服务器。可用的 Mail 服务器很多，例如在 Unix 系统下有 Sendmail，在 Windows 系统下面有 Exchange Server、SMTP Service，还有很多免费的邮件服务器软件，例如 ArGoSoft，EasyMail 等。笔者的操作系统平台是 Windows Me，所以 Exchange Server、SMTP Service 自然是用不了了，因为这两种软件都必须安装在 Windows NT 平台下面，而且不是免费软件，所以只能从免费软件中寻找了，经过一番比较，我们选中了_ArGo Software 公司开发的 ArGoSoft Mail Server 1.4 作为邮件服务器。这款软件的功能十分强大，设置简单，十分便于初学者使用。读者如果需要使用这款软件，不妨到下面

的网址去下载一个试用。

国外：

http：//www.argosoft.com

国内：

http：//software.fanso.com

下面我们介绍配置 Mail Server 的步骤。

首先安装 ArGoSoft Mail Server，然后在程序组中启动服务器，如图 11.1 所示。

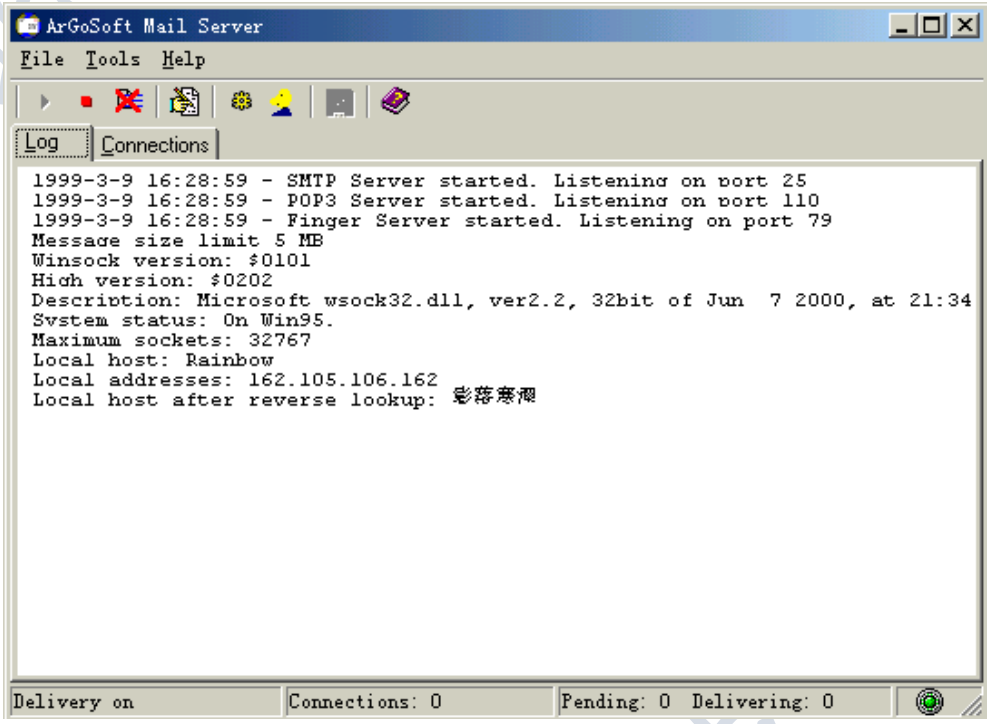


图 11.1　ArGoSoft Mail Server 的运行窗口

（1）设定 DNS 服务器的 IP 地址。

选择 Tools→Options→General，设定 DNS 服务器的 IP 地址，这样你就可以在远程计算机上访问这台邮件服务器了，也就是说，这台邮件服务器不仅可以在局域网中使用，而且可以在 Internet 上使用。如果你不知道 DNS 服务器的地址，可以使用 winipcfg 程序查看本机的网络信息，如图 11.2 所示。

（2）设定主机的名字(Domain)。

选择 Tools→Options→Local Domains，填上本机的名字，然后单击 Add 按钮，这里需要注意，填写本机的名字时，不但要写上机器的名字，而且还要加上本机所在的域名，例如笔者电脑的名字为 Rainbow，所在的域名为 pku.edu.cn，那么本机的名字应该是 Rainbow.pku.edu.cn，如图 11.3 所示。

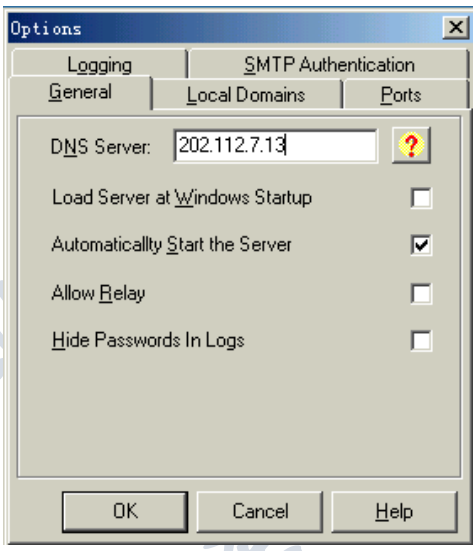


图 11.2　设定 DNS 服务器的地址



图 11.3　设定主机的名字

（3）设定 SMTP、POP3、Finger 服务的端口号。

SMTP 服务的端口号一般是 25，POP3 服务的端口号一般是 110，Finger 服务的端口号一般是 79，你如果希望改变缺省的端口号，可以选择 Tools→Options→Ports 面板，然后做相应的改变，如图 11.4 所示。

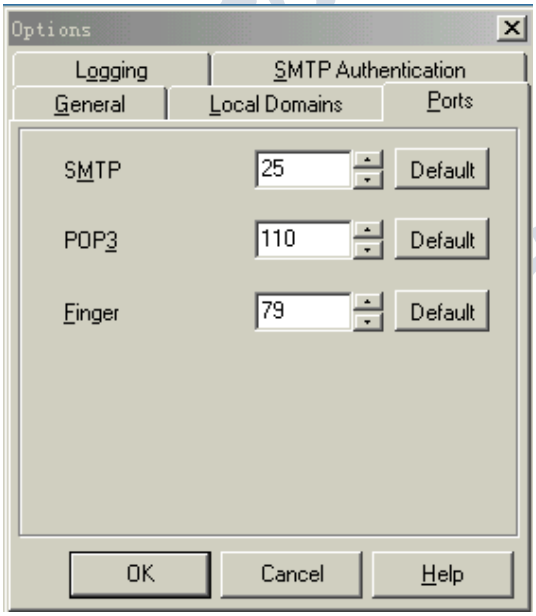


图 11.4　设定 SMTP、POP3、Finger 服务的端口号

不过我们建议读者，如果不是十分有必要，最好不要随便改变缺省的邮件服务端口号。

（4）创建新邮箱用户。

下面我们应该创建一个新的邮箱用户，选择 Tools→Users→Add New User，出现添加

用户的窗口，如图 11.5 所示。我们在 User Name 文本框中输入 Rainbow，在 Real Name 文本框中输入 Rainbow，然后在 Password 文本框和 Confirm Password 文本框中输入相同的密码，最后单击 OK，这样就创建了一个名为 Rainbow 的邮箱用户，这个用户的邮箱地址就是 Rainbow@Rainbow.pku.edu.cn，如果我们创建了一个名为 fancy 的邮箱用户，那么这个用户的邮箱地址就是 fancy@Rainbow.pku.edu.cn，@字符后面就是我们在前两步所设定的邮件服务器主机名(如图 11.6 所示)。

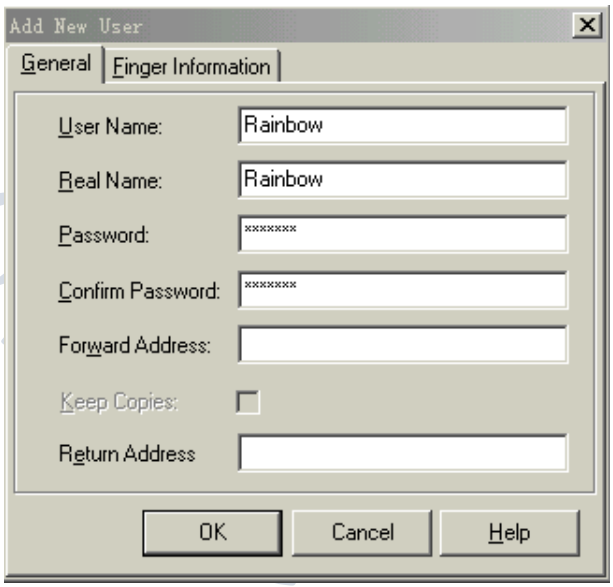


图 11.5　创建邮箱新用户

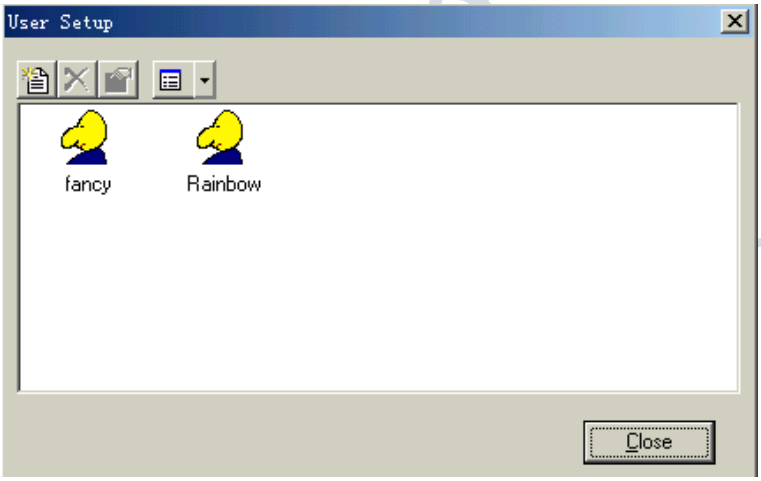


图 11.6　创建邮箱用户 fancy 和 Rainbow

（5）测试邮件服务器。

至此为止，我们的邮件服务器就算配置好了，还有一些比较高级的配置功能，限于篇幅，我们就不在这里介绍了，读者可以自行摸索。我们可以使用 Outlook、Foxmail 等邮件

服务器客户端软件来读取邮件服务器上的信件，或者是使用这个邮件服务器发信通知自己的好友，怎么样，配置一个邮件服务器是不是很简单啊，你会了吗?

如果你的邮件服务器不能够正常工作，请按照上面的步骤仔细检查一遍，一般都是主机名填错了或者是用户名有误。

11.1.2　配置 FTP 服务器

本小节中我们将向读者介绍如何配置一个 FTP 服务器。配置 FTP 服务器比配置 Mail 服务器还要简单。在本书中，我们选用的 FTP 服务器软件为 Serv_U 2.5e，这是一个小巧玲珑的 FTP 服务器软件，据说性能十分稳定，而且功能比微软的 IIS 服务器内嵌的 FTP 服务要强得多，配置也不难掌握，尤其是新版的 Serv_U 服务器，内建了一个 Wizard，可以帮助用户快速定制自己的 FTP 站点。

Serv_U 软件在教育网内很多站点上面都可以找到。例如：

ftp：//ftp.cs.pku.edu.cn

ftp：//ftp.lib.pku.edu.cn

http：//software.fanso.com

如果你希望获得最新版本的 Serv_U 软件，不妨访问下面的网址：

http：//www.ftpserv-u.com

Serv_U 的配置方法如下：

（1）下载安装 Serv_U。

这一步没有什么可说的，安装的路径最好和你希望发布 FTP 服务的路径相同。

（2）运行 FTP 服务器。

从开始菜单中运行 Serv_U FTP 服务器，如图 11.7 所示。

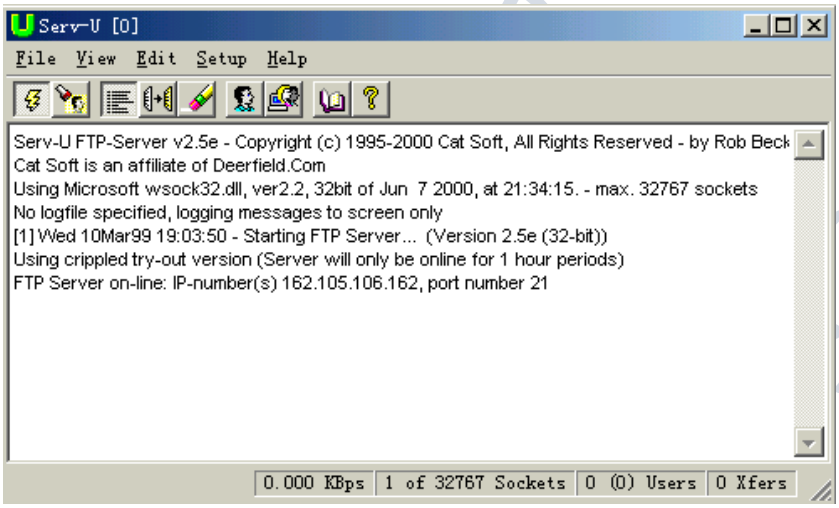


图 11.7　运行 FTP 服务器

（3）创建用户。

请选择 Setup→Users...→New，输入用户名，然后单击 OK，输入密码，在 Home directory

文本框中输入 FTP 发布服务的根目录，然后再设定 File/Directory access rules，单击 store 按钮即可，如图 11.8 所示。

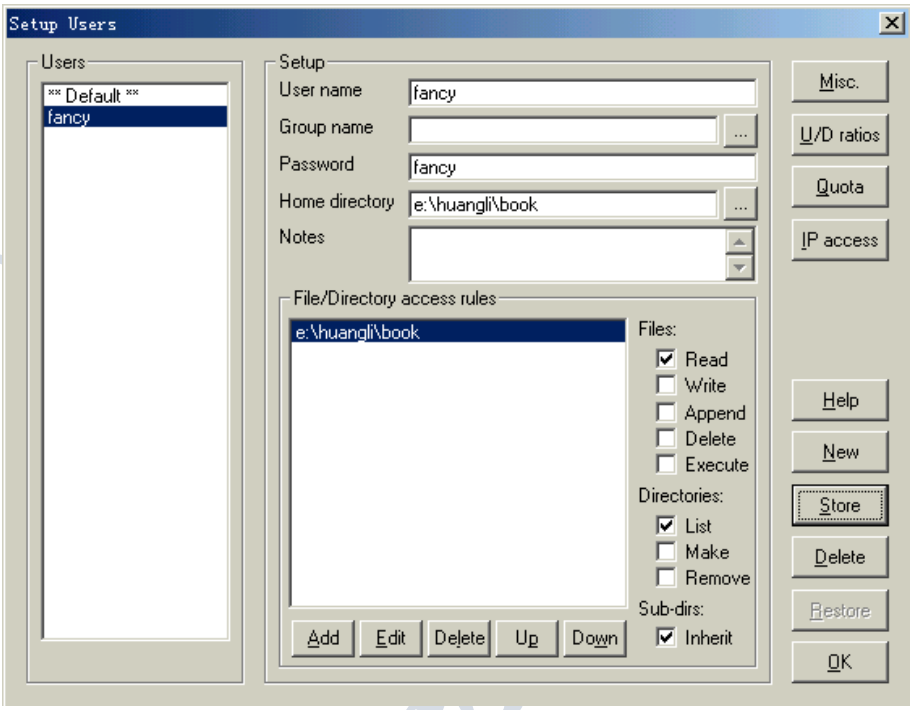


图 11.8　创建 FTP 用户

在图 11.8 中，我们创建了一个名为 fancy 的用户，该用户的密码为 fancy，FTP 服务的根目录为 e：\huangli\book，该用户只有下载的权限，而没有上载文件，删除文件或者目录、创建目录，执行程序的功能。

（4）测试 FTP 服务器。

下面我们可以随便选用一个 FTP 客户端软件来测试这个 FTP 服务器，以 Windows 系统自带的 ftp 程序为例，在 DOS Shell 中运行 ftp.exe 程序，请记住 IP 地址是 162.105.106.162，用户名是 fancy，密码是*******，这几项一定要正确，否则不可能连上服务器。测试情况如下：

```
ftp> open 162.105.106.162
Connected to 162.105.106.162.
220-Serv-U FTP-Server v2.5e for WinSock ready...
220 This FTP server is an unregistered 45 day try-out version of Serv-U
User (162.105.106.162：(none))：  fancy
331 User name okay，need password.
Password：
230 User logged in，proceed.
ftp> ls
200 PORT Command successful.
150 Opening ASCII mode data connection for /bin/ls.
```

doc
doc2
image
image2
perface.txt
226 Transfer complete.
ftp：39 bytes received in 0.06Seconds 0.65Kbytes/sec.

这说明我们的 FTP 服务器已经配置成功了。

下面我们就可以使用这两个 FTP 服务器和 Mail 服务器来测试我们编写的 JSP、Java 程序了。

注意：除了 ArGoSoft Mail Server 和 Serv_U 软件以外，读者亦可以使用别的软件来构建 Mail 服务器和 FTP 服务器，或者根本就不用自己配置服务器，直接使用 Internet 上运行的 Mail、FTP 服务器也可以，这对下面的 JSP、Java 程序的运行没有任何影响。

11.2　SMTP 服务

11.2.1　SMTP 协议和 POP3 协议

SMTP 协议（Simple Mail Transfer Protocol）是被普遍使用的最基本的 Internet 邮件服务协议。SMTP 协议支持的功能很简单，并且在安全性方面存在着问题。经过它传递的所有电子邮件都是以普通文本形式进行的。它不能够传输诸如图像等非文本信息。在网络上传输非文本时，任何人都可以在中途截取并复制这些邮件，甚至对邮件内容进行窜改。邮件在传输过程中可能丢失。别有用心的人甚至很容易就以冒名顶替的方式伪造邮件。由于安全性的缘故，后来出现了 ESMTP（Extended SMTP，扩展的 SMTP 协议）。

POP 协议（Post Office Protocol）是一种允许用户从邮件服务器收发邮件的协议。它存在有 2 种版本，即 POP2 和 POP3 两者都具有简单的电子邮件存储转发功能。它们在本质上很相似，都属于离线式工作协议。只是由于使用了不同的协议端口，使两者无法兼容。在与 SMTP 协议相结合方面，POP3 是目前最常用的电子邮件服务协议。

POP3 在支持离线工作方式之外，也支持在线工作方式。

在离线工作方式下，用户收发邮件时，首先通过 POP3 客户程序登录到支持 POP3 协议的邮件服务器，然后发送邮件及附件，其后，邮件服务器将把该用户收存的邮件传送给 POP3 的客户程序，并将这些邮件从服务器上删除；最后，邮件服务器将用户提交的发送邮件，转发到运行 SMTP 协议的计算机中，通过它实现邮件的最终发送。在为用户从邮件服务器收取邮件时，POP3 是对该用户当前存储在服务器上全部邮件进行操作，一次性将它们下载到用户端计算机中。在客户的邮件下载完毕后，邮件服务器对这些邮件的暂存的使命即告完成。

使用 POP3，用户不能对他们存储在邮件服务器上的邮件进行部分传输。离线工作方式则适合那些从固定计算机上收发邮件的用户使用。

当使用 POP3 在线工作方式收发邮件时，用户在所用的计算机与邮件服务器保持连接

的状态下读取邮件。用户的邮件将保留在邮件服务器上。

在本节中，我们将介绍使用 sun.net.smtp 包来访问 SMTP 服务器。下面先介绍 sun.net.smtp 包的一些基本情况。

11.2.2　sun.net.smtp 包简介

sun.net.smtp 包主要用于处理 SMTP 服务，它包含了 SmtpProtocolException 类、SmtpClient 类、SmtpPrintStream 类。SmtpProtocolException 类主要用于处理访问 SMTP 服务过程中的错误。SmtpClient 类代表 SMTP 服务的客户端。SmtpPrintStream 类相当于一个输出流，用于向 SMTP 服务器发送信息。下面我们简要介绍 SmtpClient 类的用法。

SmtpClient 类的主要方法如下所示：

public SmtpClient(String p0) throws IOException　这是 SmtpClient 类的构造函数。方法参数 p0 代表邮件服务器的名字。如果使用这个构造函数，那么将会建立客户端和邮件服务器之间的连接。

public SmtpClient() throws IOException　这也是 SmtpClient 类的构造函数，不过调用这个构造函数不会建立客户端与邮件服务器之间的连接，如果我们需要建立与服务器的连接，必须显示调用 openServer()方法。

public void closeServer() throws IOException　顾名思义，closeServer()方法将会关闭客户端和邮件服务器之间的连接。

public void to(String p0) throws IOException　to()方法指定发件人的地址，例如 fancyrainbow@263.net。

public void from(String p0) throws IOException　from()方法和 to()方法相对应，这个方法指定邮件的目标地址，例如 Rainbow@Rainbow.pku.edu.cn。

private void openServer(String p0) throws IOException　这个方法用于打开邮件服务器和客户端的连接，参数 p0 指定了邮件服务器的名字，例如 Rainbow.pku.edu.cn。

public PrintStream startMessage() throws IOException　startMessage()方法返回对服务端的输出流，利用 PrintStream 对象的 write()方法或者 print()方法可以向服务端输出各种各样的数据。其实这些数据就是邮件的正文了，邮件服务器会把它们转发到邮件的目标地址中。

void closeMessage() throws IOException　这个方法会关闭对邮件服务器的输出流，一旦执行了这个方法，那么邮件就算发送完毕了。不过客户端与服务端的连接并没有关闭，我们还可以再次调用 to()方法和 startMessage()方法，发送第二甚至第三封信。

public String getMailHost()　使用这个方法将会返回邮件服务器的名字。

在下面的小节中，我们将使用 SmtpClient 类，编写简单的一个发送邮件的 JSP 程序。

11.2.3　访问 SMTP 邮件服务器

请先看程序清单 11.1(smtp.jsp)。

程序清单 11.1

```
<%--
File Name：smtp.jsp
```

```
Author：fancy
Date：2001.4.8
Note：Use smtp service to send mail
--%>

<%@page import="sun.net.smtp.*" %>
<%@page import="java.io.*" %>
<%@page import="sun.net.*" %>

<%
String host="Rainbow.pku.edu.cn";
String from="fancyrainbow@263.net";
String to="fancy@Rainbow.pku.edu.cn";

SmtpClient smtp = new SmtpClient(host);
out.println("login the mail server...."+"<br>");
out.println("System Reply："+smtp.getResponseString()+"<br>");

out.println("set from address"+"<br>");
smtp.from(from);
out.println("System Reply："+smtp.getResponseString()+"<br>");

out.println("set to address"+"<br>");
smtp.to(to);
out.println("System Reply："+smtp.getResponseString()+"<br>");

out.println("the first letter to fancy@Rainbow.pku.edu.cn"+"<br>");
PrintStream msg=smtp.startMessage();
out.println("System Reply："+smtp.getResponseString()+"<br>");

out.println("senting message"+"<br>");
msg.println("To：  " + to);
msg.println("Subject：  Hello SmtpClient");
out.println("System Reply："+smtp.getResponseString()+"<br>");
// blank line between headers and message
msg.println();
out.println("System Reply："+smtp.getResponseString()+"<br>");
msg.println("This is a test message.");
out.println("System Reply："+smtp.getResponseString()+"<br>");

out.println("close printstream"+"<br>");
msg.close();
out.println("System Reply："+smtp.getResponseString()+"<br>");
```

北京希望电子出版社

```
out.println("get the mail host"+"<br>");
out.println(smtp.getMailHost()+"<br>");
out.println("System Reply："+smtp.getResponseString()+"<br>");

out.println("close the connect to mail host"+"<br>");
smtp.closeServer();
out.println("System Reply："+smtp.getResponseString()+"<br>");
%>
```

程序清单 11.1 中，利用邮件服务器 Rainbow.pku.edu.cn，发送邮件到 fancy@Rainbow.pku.edu.cn 中，程序首先创建了 SmtpClient 类的实例对象 smtp，和邮件服务器建立连接，然后使用 from()，to()方法设定邮件的来源和目标地址，一切就绪之后，就可以调用 startMessage()方法，获取对服务端的输入流对象 msg，接着利用这个输入流，往服务端中写入数据，服务器会把这些数据转发到邮件的目标地址中。最后，使用 closeServer()方法关闭客户端和服务端的连接，释放系统资源。在程序中，读者应该注意到我们多次调用了 getResponseString()方法，这个方法是从 SmtpClient 的父类中继承过来的，使用它可以获取服务器的应答信息。

程序清单 11.1(smtp.jsp)的运行效果如图 11.9 所示。

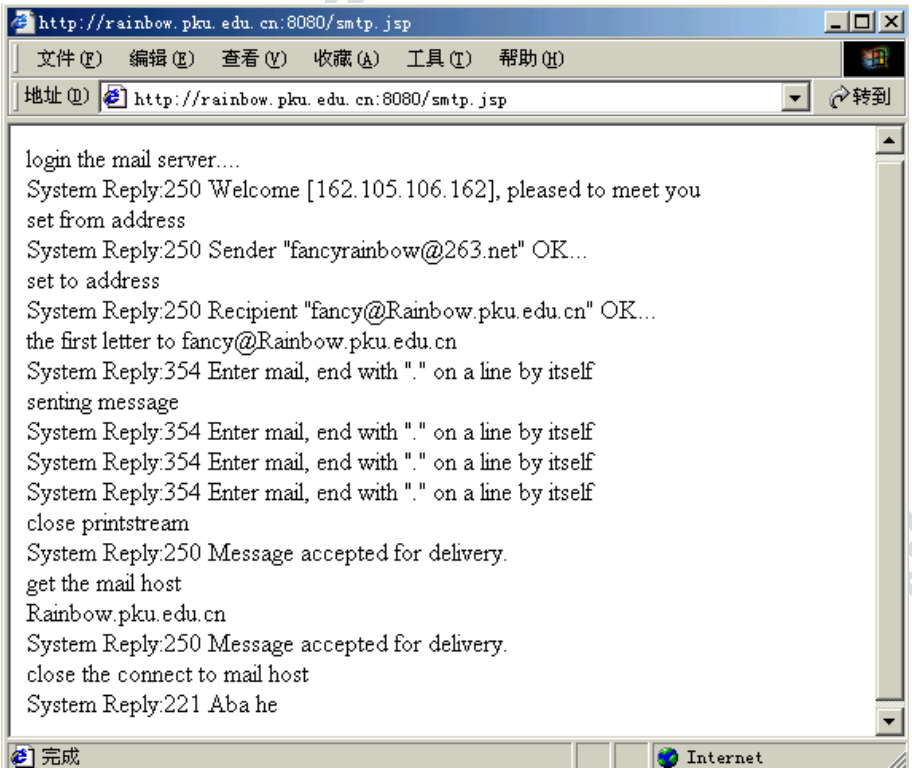


图 11.9　smtp.jsp 程序的运行效果

如果我们使用 Outlook 查看账号：fancy@Rainbow.pku.edu.cn，会看到这样的信息，如

图 11.10 所示。

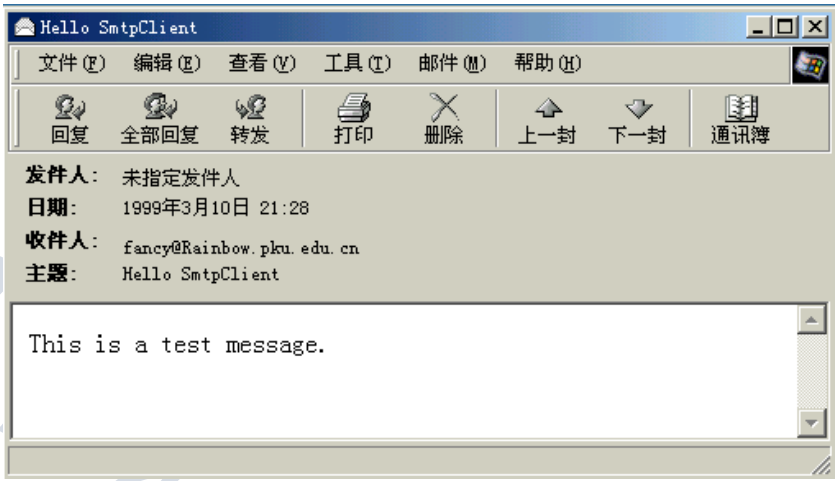


图 11.10　收到新邮件啦

关于使用 sun.net.smtp 包访问 SMTP 服务的内容，我们就暂时介绍到这里，在后面，我们还会介绍如何使用更为底层的 Socket 类来访问 SMTP 服务。在第 12 章，我们还会介绍功能更为强大的 Java Mail API。

11.3　FTP 服务

11.3.1　FTP 协议

FTP 协议(File Transfer Protocol)是 Internet 上用来传送文件的协议（文件传输协议）。它是为了我们能够在 Internet 上互相传送文件而制定的的文件传送标准，规定了 Internet 上文件如何传送。也就是说，通过 FTP 协议，我们就可以跟 Internet 上的 FTP 服务器进行文件的上传（Upload）或下载（Download）等动作。

和其他 Internet 应用一样，FTP 也是依赖于客户程序/服务器关系的概念。在 Internet 上有一些网站，它们依照 FTP 协议提供服务，让网友们进行文件的存取，这些网站就是 FTP 服务器。网上的用户要连上 FTP 服务器，就要用到 FPT 的客户端软件，通常 Windows 都有“ftp”命令，这实际就是一个命令行的 FTP 客户程序，另外常用的 FTP 客户程序还有 CuteFTP、Ws_FTP、FTP Explorer 等。

要连上 FTP 服务器（即“登录”），必须要有该 FTP 服务器的帐号。如果是该服务器主机的注册客户，你将会有一个 FTP 登录帐号和密码，就凭这个帐号密码连上该服务器。但 Internet 上有很大一部分 FTP 服务器被称为“匿名”（Anonymous）FTP 服务器。这类服务器的目的是向公众提供文件拷贝服务，因此，不要求用户事先在该服务器进行登记注册。

Anonymous（匿名文件传输）能够使用户与远程主机建立连接并以匿名身份从远程主机上拷贝文件，而不必是该远程主机的注册用户。用户使用特殊的用户名“anonymous”和“guest”就可有限制地访问远程主机上公开的文件。现在许多系统要求用户将 Emai1 地址作

为口令，以便更好地对访问进行跟综。出于安全的目的，大部分匿名 FTP 主机一般只允许远程用户下载（download）文件，而不允许上载（upload）文件。也就是说，用户只能从匿名 FTP 主机拷贝需要的文件而不能把文件拷贝到匿名 FTP 主机。另外，匿名 FTP 主机还采用了其他一些保护措施以保护自己的文件不至于被用户修改和删除，并防止计算机病毒的侵入。在具有图形用户界面的 World Wild Web 于 1995 年开始普及以前，匿名 FTP 一直是 Internet 上获取信息资源的最主要方式、在 Internet 成千上万的匿名 PTP 主机中存储着无以计数的文件，这些文件包含了各种各样的信息、数据和软件。人们只要知道特定信息资源的主机地址，就可以用匿名 FTP 登录获取所需的信息资料。虽然目前使用 WWW 环境已取代匿名 FTP 成为最主要的信息查询方式，但是匿名 FTP 仍是 Internet 上传输分发软件的一种基本方法。

这一节里，我们将会使用 sun.net.ftp 包的类来实现访问 FTP 服务的功能。首先我们先介绍 sun.net.ftp 包的功能。

11.3.2　sun.net.ftp 包简介

sun.net.ftp 包有五个类，分别是 FtpClient 类、FtpLoginException 类、FtpProtocolException 类、FtpInputStream 类、IftpClient 类。FtpClient 类代表的是 FTP 服务的客户端，FtpInputStream 流类继承自 TelnetInputStream 类，用于读取 FTP 服务器的应答信息。lftpClient 类用于处理使用代理服务器访问 FTP 服务器的情况。FtpLoginException 类和 FtpProtocolException 类用于处理访问 FTP 服务器过程中的异常和错误。

下面我们简要介绍 FtpClient 类的用法。FtpClient 类的主要方法如下所示：

public FtpClient(String p0) throws IOException;　这是 FtpClient 类的构造函数，利用它可以和 FTP 服务器建立有效的连接，参数 p0 代表服务器的 IP 地址。但是还需要调用 login()方法，输入用户名和密码，才能够登录 FTP 服务器。

public FtpClient() throws IOException;　这也是 FtpClient 类的构造函数，但是这个函数没有和 FTP 服务器建立实质上的连接。需要调用 openServer()方法打开客户端与 FTP 服务器的连接，然后再使用 login()方法登录 FTP 服务器。

public void closeServer() throws IOException;　这个方法关闭客户端与服务端的连接。

public void openServer(String p0) throws IOException;　这个方法将建立客户端与 FTP 服务器之间的连接。参数 p0 是 FTP 服务器的名字。

public void openServer(String p0，int p1) throws IOException;　这个方法的作用和前者一样，参数 p1 指定需要连接的服务器的 FTP 服务的端口号，FTP 服务的端口号一般是 21，但是出于某些原因，可能将这个端口号改为别的数值了，这时就需要使用这个方法和服务器建立有效的连接。

public void login(String p0，String p1) throws IOException;　这个方法的作用是登录 FTP 服务器，参数 p0 代表用户名，参数 p1 代表相应的密码，在调用 login()方法以前，必须首先调用 openServer()等方法，打开客户端与服务端之间的连接。

public TelnetInputStream get(String p0) throws IOException　这个方法用于从 FTP 服务器上下载文件，参数 p0 代表文件或者目录的名字。这个方法的返回值是一个

TelnetInputStream 对象。下面的代码段演示了如何使用这个方法从 FTP 服务器上获取文件。

例：

```
InputStream tis=(InputStream)fc.get("参考文献.txt");
BufferedReader filein=new BufferedReader(new InputStreamReader(tis));
String temp;
while((temp=filein.readLine())!=null)
{
    out.println(temp+"<br>");
}
```

在上面的代码中，我们首先把 get()方法的返回值强制转换为 InputStream 对象，然后从这个 InputStream 对象创建 BufferedReader 对象 filein，利用 filein 来读取数据流，这是因为 BufferedReader 对象有 readLine()方法，可以一次读入一行数据，支持中文的读取，而且不会发生网络阻塞，TelnetInputStream 对象就没有这样的功能。

public TelnetOutputStream put(String p0) throws IOException　put()方法和 get()方法相对，put()方法可以往服务器中写入数据，参数 p0 代表服务端相应的文件名，这个方法将打开一个 Telnet 输出流对象，利用 TelnetOutputStream 对象的方法，可以把数据写入到 p0 参数指定的文件中，在调用这个方法的同时，首先需要确定你是否有权限往 FTP 服务器上写入数据。

public TelnetInputStream list() throws IOException　这个方法就相当于标准的 FTP 命令 LIST，它可以返回 FTP 服务器上当前目录的信息，例如有哪些文件夹，有哪些文件，大小如何，什么时间创建的等等。list()方法的返回值是一个 TelnetInputStream 对象，下面的代码段演示了如何使用 list()方法列出 FTP 服务器上的目录信息。

例：

```
InputStream tis=(InputStream)fc.list();
BufferedReader in=new BufferedReader(new InputStreamReader(tis));
String temp;
while((temp=in.readLine())!=null)
{
    out.println(temp+"<br>");
}
```

由上面的代码可见，list()方法与 get()方法的使用技巧基本一样。

public void cd(String p0) throws IOException　cd()方法十分类似于 DOS 系统的 cd 命令，利用 cd()方法可以改变 FTP 服务器上的当前目录。参数 p0 表示文件夹的名字，特别地，cd("..")表示进入当前目录的上一级目录。利用 cd()方法和 list()方法相互配合，我们就可以遍历 FTP 服务器上的所有目录和文件了。

public void binary() throws IOException　binary()方法指定客户端与服务端之间的数据传输以二进制的形式进行。

public void ascii() throws IOException　ascii()方法指定客户端与服务端之间的数据传输以 ascii 吗的形式进行。

在下面的小节中，我们将以两个完整的例子为代表，向读者介绍如何使用上面列出来

的方法访问 FTP 服务器。

11.3.3　访问 FTP 服务器

请看程序清单 11.2(ftp.jsp)，这个 ftp.jsp 程序演示了如何获取服务器上的目录信息。

程序清单 11.2

```
<%--
File Name：ftp.jsp
Author：fancy
Date：2001.4.8
Note：Use ftp service to list file or directory
--%>

<%@ page contentType="text/html;charset=gb2312" %>
<%@page import="sun.net.ftp.*" %>
<%@page import="java.io.*" %>
<%@page import="sun.net.*" %>

<%
FtpClient fc=new FtpClient();
fc.openServer("162.105.106.162");
out.println(fc.getResponseString()+"<br>");
fc.login("fancy"，"fancy");
out.println(fc.getResponseString()+"<br>");

fc.cd("doc");
InputStream tis=(InputStream)fc.list();
out.println(fc.getResponseString()+"<br>");
BufferedReader in=new BufferedReader(new InputStreamReader(tis));
String temp;
while((temp=in.readLine())!=null)
{
      out.println(temp+"<br>");
}
fc.closeServer();
out.println(fc.getResponseString()+"<br>");
%>
```

程序 ftp.jsp 首先创建 FtpClient 类的实例对象 fc，然后依次调用 openServer()方法和 login()方法登录服务器。调用 cd()方法把当前目录设为 doc，然后利用 list()方法获取目录信息，接下来就是如何读取输入流的信息了。

程序清单 11.2(ftp.jsp)的运行效果如图 11.11 所示。

北京希望电子出版社

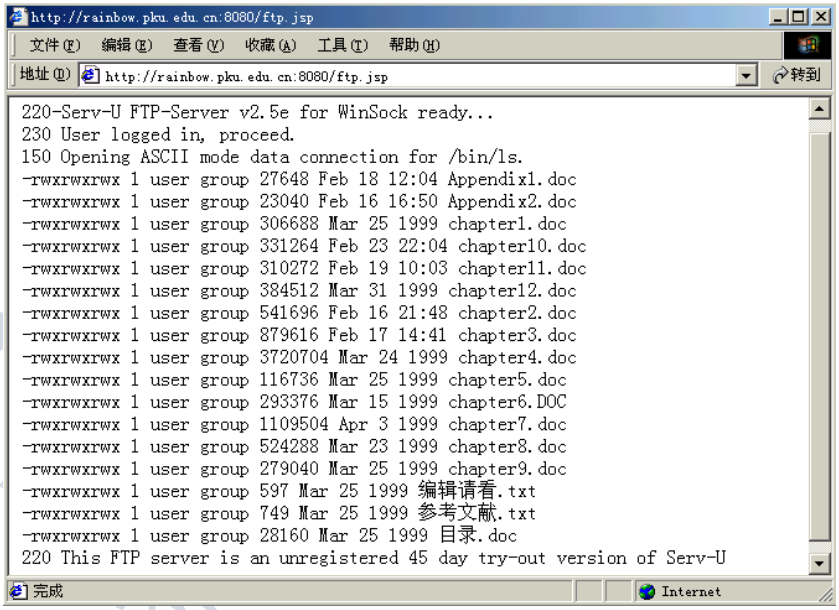


图 11.11　ftp.jsp 程序的运行效果

程序清单 11.3

```
<%--
File Name：ftp1.jsp
Author：fancy
Date：2001.4.8
Note：Use ftp service to get file
--%>

<%@ page contentType="text/html;charset=gb2312" %>
<%@page import="sun.net.ftp.*" %>
<%@page import="java.io.*" %>
<%@page import="sun.net.*" %>
<%!
public String getStr(String str)
{
    try
    {
        String temp_p=str;
        byte[] temp_t=temp_p.getBytes("ISO8859-1");
        String temp=new String(temp_t);
        return temp;
    }
    catch(Exception e)
    {
        //to do nothing
    }
    return "null";
```

```
}
%>
<%
FtpClient fc=new FtpClient();
fc.openServer("162.105.106.162");
//out.println(fc.getResponseString()+"<br>");
fc.login("fancy", "fancy");
//out.println(fc.getResponseString()+"<br>");
fc.cd("doc");
out.println(fc.getResponseString()+"<br>");
InputStream tis=(InputStream)fc.get("参考文献.txt");
out.println(getStr(fc.getResponseString())+"<br>");
BufferedReader filein=new BufferedReader(new InputStreamReader(tis));
String temp;
while((temp=filein.readLine())!=null)
{
      out.println(temp+"<br>");
}
fc.closeServer();
out.println(fc.getResponseString()+"<br>");
%>
```

在程序清单 11.3 中(ftp1.jsp)，前半部分和 ftp.jsp 程序一样，后半部分演示了如何利用 get()方法获取 FTP 服务器上的文件。在这里我们遇到一个问题，就是 FTP 服务器如果往客户端发送的应答信息中含有中文，那么在 JSP 程序中将不能够正确显示，所以我们编写了一个 getStr()函数，这个函数可以转换字符串的内码，含有中文字符的字符串经过调用这个函数，就能够正确显示中文字符了。 ftp1.jsp 程序的运行效果如图 11.12 所示。



图 11.12　ftp1.jsp 程序的运行效果

11.4　News 服务

11.4.1　NNTP 协议

网络新闻是指在 Internet 上拥有相同爱好的用户进行意见交换的一种无形的用户交流网络。与邮件用户组相类似，它也是按照不同的专题来组织的，每个专题称为一个网络新闻组。目前，在 Internet 上存在着为数众多的专题新闻组。网络新闻服务器是提供网络新闻服务的计算机，它使用 NNTP 协议（Network News Transfer Protocol）。志趣相同的用户可借助网络新闻服务器，展开各种类型的专题讨论。只要用户加入某一网络新闻组后，就可以读取到加入这个组的其他用户关于某一专题发表的意见；加入者可以参加讨论，将自己的见解送给本组的其他用户阅读；当遇到困难时，也可以向本组的其他用户求助，这将得到热心的帮助。由此可见，新闻组按照自己的独特方式运行。虽然人们常常对别人的攻击能够做出快速反应，但是在这里很难就任何事都达成一致意见。

网络新闻组为某些特定的用户提供一个论坛，它从小组成员那里或其它来源获得消息，又把一则一则消息提供给成员用户。一个新闻组（News Group）相当于一个电子公告板。一则新闻被称为一篇文章（article），以电子邮件的方式发给网络新闻组。从表面上看，电子邮件通信组似乎足以适应网络用户之间进行讨论和交流的需要。但是，参加电子邮件通信组，需要不断参与，时常会遇到接收和处理大量信件的烦脑。参加网络新闻组，则可以进行更多的讨论，甚至只限于旁观，对于兴趣不浓的问题，无需订阅新闻，偶尔查阅一下最新讨论的结果。

网络新闻组是一种用户完全自由参与的活动，它与邮件用户组不同，要参加时用户无须事先申请，不感兴趣时也不用声明退出。只在用户的计算机拥有支持 NNTP 协议的“新闻阅读器”程序，就可通过 Internet 随时阅读新闻服务器提供的分门别类的消息。用户能读到的专题种类取决于用户使用的新闻阅读程序访问的新闻服务器。每个新闻服务器在收集和发布网络消息方面是“各自为政”的。用户可以将自己的见解提供给新闻服务器，只要对某个网络新闻组讲座的专题感兴趣，用户就可以将自己的意见通过计算机网络送入该新闻组。用户提交的每条好信息都将作为一条网络新闻，通过计算机网络而传播开去，并最终作为过时的消息消失在网络中。

我们可以使用客户端软件来阅读新闻组中的新闻，例如 Outlook，但是，我们既然学习了 JSP 技术和 Java 语言，为何不利用它们来编写一个属于自己的新闻组阅读程序呢?下面我们将介绍如何使用 Java 语言的 sun.net.nntp 包来编写一个简单的 JSP 程序，以访问北京大学的 NEWS 服务器。我们将首先介绍 sun.net.nntp 包的一些基础知识。

11.4.2　sun.net.nntp 包简介

sun.net.nntp 包含五个类，它们分别是 NntpClient 类、NewsgroupInfo 类、NntpInputStream 类、NntpProtocolException 类、UnknownNewsgroupException 类。后面的三个类不太重要，因此我们不介绍这三个类的详细用法，对此感兴趣的读者可以参考相应的文档。

NntpClient 类代表 News 服务器的客户端，它定义的重要方法如下所示。

public NntpClient();　这是 NntpClient 类的构造函数，创建一个 NntpClient 类的实例，但是客户端与News服务器之间没有建立有效的连接。如果需要建立与News服务器的连接，必须接着调用 openServer()方法。

public NntpClient(String p0) throws IOException;　这也是 NntpClient 类的构造函数，参数 p0 表示 News 服务器的名字，例如 news.pku.edu.cn，这个构造函数将建立客户端与 News 服务器之间的连接，并且登录服务器。

public void openServer(String p0，int p1) throws IOException;　openServer()方法建立客户端与 News 服务端的连接，参数 p0 代表 News 服务器的地址，参数 p1 代表 NNTP 服务的端口号，一般是 119。

String[] tokenize(String p0);　tokensize()方法可以把一个完整的字符串分解为若干个字符串，并且保存在一个字符串数组中。例如字符串“This is a test”，传递给 tokenize()方法后，将被分解为“This”、“is”、“a”、“test”等四个字符串，并被保存到指定的字符串数组中。

public NewsgroupInfo getGroup(String p0) throws IOException;　这个方法将返回一个 NewsgroupInfo 类型的对象，这个对象包含有当前新闻组的信息。在下面我们还会介绍 NewsgroupInfo 类的用法。

public void setGroup(String p0) throws IOException;　这个方法用于设定当前新闻组的名字，例如 comp.lang.java 新闻组，这是专门讨论 Java 语言的新闻组，或者 comp.lang.java.corba 新闻组，这是专门讨论 CORBA 开发的新闻组。news.pku.edu.cn 服务器上有下列新闻组：

news.*：　有关 news 本身的讨论组。

sci.*：　有关自然科学的讨论组。

soc.*：　有关社会科学的讨论组。

bionet.*：　有关生物学的讨论组。

alt.*：　其他话题的混合讨论组。

comp.*：关于计算机技术的讨论组。

public InputStream getArticle(int p0) throws IOException;　这个方法可以获取特定新闻的全文信息。参数 p0 指定该新闻的编号。

public InputStream getArticle(String p0) throws IOException;　这个方法和上面的方法的作用是一样的，都是获取特定新闻的全文信息，不过方法的参数不同而已，这个方法的参数是相应新闻的标题。

读者也许会问，新闻的正文是什么样子的?下面就是一个新闻消息的全文(包括数据头在内)。

例：

From：　Genady Beryozkin

Newsgroups：　comp.lang.java.help，comp.lang.java，comp.lang.java.misc

Subject：　Re：　HTML parser

Date：　Sun，08 Apr 2001 13：42：22 +0200

Organization：　The Hebrew University of Jerusalem

Lines：　10

Message-ID： <3AD04E9E.9EFCA764@inter.removeme.net.il>
References： <3AC5E29B.E5A74176@adcc.alcatel.be>
NNTP-Posting-Host： ccdis-11.technion.ac.il
Mime-Version： 1.0
Content-Type： text/plain; charset=us-ascii
Content-Transfer-Encoding： 7bit
X-Trace： news.huji.ac.il 986730126 28262 132.68.253.11 (8 Apr 2001 11：42：06 GMT)
X-Complaints-To： abuse@news.huji.ac.il
NNTP-Posting-Date： Sun，8 Apr 2001 11：42：06 +0000 (UTC)
X-Mailer： Mozilla 4.7 [en] (WinNT; U)
X-Accept-Language： en
Xref：sunlight.pku.edu.cn comp.lang.java.help：38477 comp.lang.java：30485 comp.lang.java.misc：7475

The also is a general parser called ANTLR，(www.antlr.org)
that has a demo HTML grammar defined.

Shanmuganathan Thiagarajan wrote：

> Hello，
> Is there any class available in Java to parse a HTML dource
> file into a webpage document.And if possible，how it can be done.
> Can you clear it with a sample code.

在上面的例子中，读者可以看到，第一行是作者的名字，第二行是新闻组的名字，第三行是本新闻的主题，第四行是日期信息，第五行是作者的单位，下面的代码行就是一些编码方面的信息，这些行我们不用过多去关心，不过我们需要记住这些行的行数(一共 13 行，中间一行太长了，分为两行写)。因为用户是不会关心这些信息的，程序中需要把这 13 行的数据全部过滤掉，不要让用户看到。接下来以黑体强调的行就是新闻的正文了，这是真正有用的部分，我们必须把这部分内容正确显示出来。

上面我们仔细分析了 News 的结构，这对于我们编写阅读新闻的 JSP 程序是十分有帮助的，读者一定要注意到这一点。

getArticle()方法的返回值是一个 InputStream 类型的对象，我们一般需要把它转换为 BufferedReader 类型的对象，以便提高读取效率、避免网络阻塞和中文问题。

public InputStream getHeader(int p0) throws IOException; 这个方法可以返回特定新闻的数据头，在我们前面介绍的 News 的结构中，News 数据头就是黑体字以外的部分。这个方法的参数 p0 是新闻的 ID 号。

public InputStream getHeader(String p0) throws IOException; 这个方法的作用与上面的方法的作用一样，也是获取特定新闻的数据头，不过就是方法参数不同，这个方法的参数 p0 表示新闻的标题。getHeader()方法一般来说很少会用到。读者也不用太重视这个方法。

下面我们介绍 NewsgroupInfo 类的变量和方法，NewsgroupInfo 类定义了三个变量，分别是：

public String name; 这个变量代表当前新闻组的名字。

public int firstArticle; 这个变量表示当前新闻组第一条新闻的编号。

public int lastArticle;　这个变量表示当前新闻组最后一条新闻的编号，一个新闻组的新闻的编号都是连续的，也就是说 lastArticle 减去 firstArticle 的值就是当前新闻组中新闻的总数了。

public void reload(NntpClient p0) throws IOException;　reload()方法可以更新当前的 NewsgroupInfo 对象的信息，该方法的参数是 NntpClient 类的实例对象。

11.4.3　访问 NEWS 服务器

在本小节中，我们将使用上面介绍的知识，编写一个简单的阅读新闻的 JSP 程序，请看程序清单 11.4(nntp.jsp)。

程序清单 11.4

```
<%--
File Name：nntp.jsp
Author：fancy
Date：2001.4.8
Note：Use nntp service to read news
--%>

<%@ page contentType="text/html;charset=gb2312" %>
<%@page import="sun.net.nntp.*" %>
<%@page import="java.io.*" %>
<%@page import="sun.net.*" %>

<%
NntpClient nc=new NntpClient("news.pku.edu.cn");
out.println(nc.getResponseString()+"<br>");
nc.setGroup("comp.lang.java");
out.println(nc.getResponseString()+"<br>");
NewsgroupInfo ni=nc.getGroup("comp.lang.java");
//out.println(nc.getResponseString()+"<br>");
int first=ni.firstArticle;
//out.println(nc.getResponseString()+"<br>");
int last=ni.lastArticle;
//out.println(nc.getResponseString()+"<br>");
//out.println(first+"<br>");
//out.println(last+"<br>");
InputStream is=nc.getArticle(first);
BufferedReader in=new BufferedReader(new InputStreamReader(is));
String temp;
in.readLine();
out.println(in.readLine()+"<br>");
out.println(in.readLine()+"<br>");
out.println(in.readLine()+"<br>");
out.println(in.readLine()+"<br>");
out.println(in.readLine()+"<br>");
```

北京希望电子出版社

```
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
in.readLine();
while((temp=in.readLine())!=null)
{
    out.println(temp+"<br>");
}
nc.close();
%>
```

nntp.jsp 程序首先登录 News 服务器，在本例中，我们使用了北京大学的 News 服务器，news.pku.edu.cn，然后利用 setGroup()方法设定当前新闻组为 comp.lang.java，获取 comp.lang.java 新闻组的 NewsgroupInfo 对象，由 NewsgroupInfo 对象的 lastArticle 变量和 firstArticle 变量的值，得知 comp.lang.java 新闻组的新闻数，接下来使用 getArticle()方法获取最后一篇新闻的正文，在使用 BufferedReader 对象读取新闻的时候，要特别注意把无用的信息过滤掉，也就是不把它们输出来。

程序清单 11.4 的运行效果如图 11.13 所示。

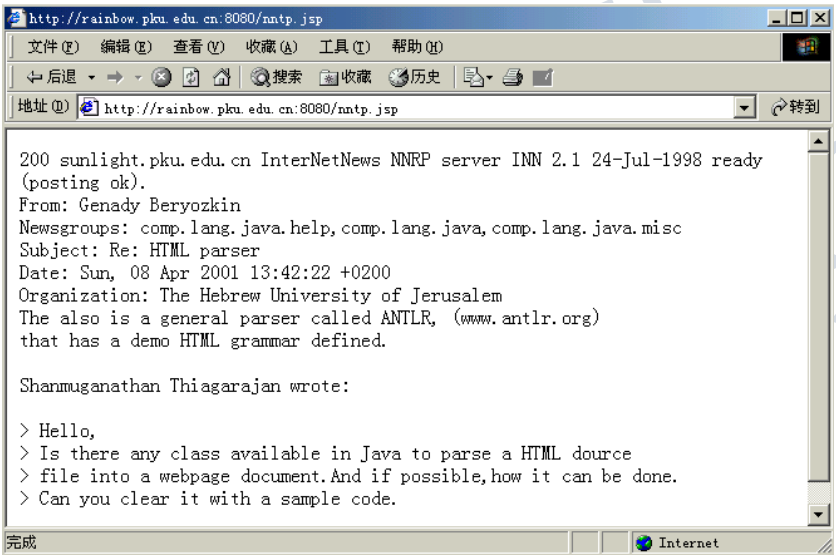


图 11.13　nntp.jsp 程序的运行效果

11.5　Java Socket

在前面的几节中，我们已经讨论了如何使用高级的类(如 FtpClient、SmtpClient、NntpClient 类)来访问相应的网络服务，其实，这些高级类都是以 Socket 的方式与服务端建立连接的。在这一节中我们将向读者介绍如何使用底层的 Socket 类来访问网络服务。

11.5.1　java.net 包简介

java.net 包就是 Java 语言的标准网络编程类库，Java 语言中的网络编程，一般可以分为网络资源的使用、Socket 类和数据报三个部分，java.net 包的类也就是按照这三个部分来进行组织的。在这一小节中，我们将对 java.net 包的类做一个简要的介绍，与 Socket 相关的类则放到下一个小节中介绍。

数据报

数据报是一种无连接的通信方式，类似于现实生活中的广播，网络中某台主机发出数据报类型的信息包，信息包所能到达的区域的所有计算机都能够接收并处理这个信息包。数据报的速度比较快，但是由于发送者和接收者之间不建立网络连接，不能保证所有数据都能送到目的地。所以一般用于传送非关键性的数据。发送和接收数据报需要使用 java.net 包中的 DatagramPacket 类和 DatagramSocket 类。

（1）DatagramPacket 类。该类是进行数据报通信的基本单位，它包含了需要传送的数据、数据报的长度、IP 地址和端口等。DatagramPacket 类的构造方法有两种。

DatagramPacket(byte []，int)构造一个用于接收数据报的 DatagramPacket 类，byte []类型的参数是接收数据报的缓冲，int 类型的参数是接收的字节数。

DatagramPacket(byte []，int，InetAddress，int)构造一个用于发送数据的 DatagramPacket 类，byte []类型参数是发送数据的缓冲区，int 类型参数是发送的字节数，InetAddress 类型参数是接收机器的 Internet 地址，最后一个参数是接收方接收数据包的端口号。

（2）DatagramSocket 类。DatagramSocket 类是用来发送数据报的 Socket，它的构造方法有两种。

DatagramSocket()构造一个用于发送的 DatagramSocket 类。

DatagramSocket(int p0) 构造一个用于接收的 DatagramSocket 类，参数 p0 表示接收方监听的网络端口号。

构造完 DatagramSocket 类后，就可以发送和接收数据报。发送和接收数据包的方法和本书的主题关系不是太大。我们就不再多说了，对此感兴趣的读者可以参考别的专著。

URL 类和 URLConnection 类

利用 URL 类和 URLConnection 类几乎可以访问所有的网络资源，包括 FTP、SMTP、News 等服务在内。

（1）URL 类。在 Internet 上的所有网络资源都是用 URL(Uniform Resource Locator，译为统一资源定位符)来表示的，URL 类在 Java 的网络编程中是十分重要的，许多网络程序都以它为基础。URL 类的构造方法有六种。我们在这里只列出最常用的四种方法。

URL(String，String，int，String)构造一个 URL 类。第一个 String 类型的参数是网络协议的类型，可以是 http，ftp，file，telnet，mailto 等。第二个 String 类型参数是主机名，例如 www.pku.edu.cn，int 类型参数是指定端口号，例如 80，最后一个参数是给出服务器上的文件名或路径名，例如 index.html。

URL(String，String，String)构造一个 URL 类。参数含义与上相同，使用缺省端口号，一般是 80。

URL(URL，String)构造一个 URL 类。使用给出的 URL 和相对路径，String 类型参数代表相对路径和文件的名字，使用缺省的协议 http，缺省的端口号 80。

URL(String)使用 URL 字符串构造一个 URL 类，使用缺省的协议 http，缺省的端口号 80，缺省的文件名 index.htm。

在构造 URL 类时，必须有相应的异常处理。因此应写成：

例：

```
URL url;
String home = "http：//www.pku.edu.cn/";
try
{
    url = new URL(home);
}
catch (MalformedURLException e)
{
    ……  //出错处理
}
```

在构造完一个 URL 类后，如果与服务器建立了网络连接，就可以使用 URL 类中的 openStream()方法与服务器上的文件建立一个流的连接，但是这个流是输入流对象(InputStream)，只能读取服务器的应答信息而不能写。读者如果希望往服务上写入数据，则必须创建 URLConnection 类，调用 URLConnection 类的 getOutputStream()方法，获取输出流对象。

（2）URLConnection 类。使用 URL 类中 openConnection()方法可以构造一个 URLConnection 类。这个类中包含了更丰富的方法，可以对服务器上的文件进行更多的处理。URLConnection 类的构造方法是 URLConnection(URL)，可以构造一个对指定 URL 的连接对象。如果直接用 URLConnection 类的构造方法来构造 URLConnection 对象时，并未建立与指定 URL 的连接，所以还必须使用 URLConnection 类中的 connect()方法建立连接。而用 URL 类中的 openConnection()方法来构造 URLConnection 对象时，就已经建立了网络连接，就不需要使用 connect()方法了。

URLConnection 类最重要的方法就是 getInputStream()方法和 getOutputStream()方法，使用这两个方法就可以与服务器进行交互操作了。

利用 URL 类和 URLConnection 类其实也可以访问 FTP、News、SMTP 等网络服务。

11.5.2　Socket 和 ServerSocket

这一小节中我们将重点介绍 Socket 类和 ServerSocket 类。Socket 原先是 Unix 系统中

的概念，以后在网络编程中被广泛使用，并被推广到了 Windows 系统中，产生了所谓的 Windows Socket API。Java 语言中也引入了 Socket 的概念。在 Java 程序中我们可以把 Socket 类和 ServerSocket 类分别用于 Client 端和 Server 端，在任意的两台机器间建立点对点的连接(PTP)。可以说，Socket 类与 ServerScoket 类是 Java 网络类库中最底层的类了，表示网络服务客户端的类如 NewworkClient 类、FtpClient 类、SmtpClient 类无不是直接或者间接从 Socket 类继承而来的。下面我们就来介绍这两个类的使用方法。

Socket 类

Socket 类用在用户端，用户通过构造一个 Socket 类来建立与服务器的连接。Socket 连接可以是流连接，也可以是数据报连接，这取决于构造 Socket 类时使用的构造方法。一般使用的流连接，流连接的优点是能所有数据都能准确、有序地送到接收方，缺点是速度较慢。Socket 类的构造方法有很多，最主要的有下面这四种。

Socket(String，int)构造一个连接指定主机、指定端口的数据流类型的 Socket。第一个参数指定主机的名字，第二个参数指定端口号。什么服务就用什么服务的端口号，例如 HTTP 服务的端口号为 80，FTP 服务的端口号为 21，NNTP 服务的端口号为 119。

Socket(String，int，boolean)构造一个连接指定主机、指定端口的 Socket 类，boolean 类型的参数用来设置 Socket 对象的类型，是数据流 Socket 还是数据报 Socket。其余参数的意义与第一个方法相同。

Socket(InetAddress，int)构造一个连接指定 Internet 地址、指定端口的数据流类型的 Socket。在这个构造函数中，主机地址以 InetAddress 对象来表示。

Socket(InetAddress，int，boolean)构造一个连接指定 Internet 地址、指定端口的 Socket 类，boolean 类型的参数用来设置是流 Socket 还是数据报 Socket。

在构造完 Socket 类后，就可以通过 Socket 类来建立输入流、输出流，通过数据流来传送数据。

Socket 类的最主要的方法如下所示。

public InputStream getInputStream() throws IOException; 这个方法用于获取网络连接的输入流对象，利用这个流对象的 read()方法可以读取服务端的应答信息。

public OutputStream getOutputStream() throws IOException; 这个方法用于获取网络连接的输出流对象，利用这个流对象的 write()方法可以往服务端写入数据。服务端会解析这些数据，并返回应答信息到客户端。

public void close() throws IOException; 这个方法可以关闭当前的 Socket 连接。

ServerSocket 类

ServerSocket 类用在服务器端，接收用户端传送的数据。ServerSocket 类主要的构造方法有如下两种。

ServerSocket(int)在指定端口上构造一个 ServerSocket 类。唯一的参数指定监听的端口号。

ServerSocket(int，int)在指定端口上构造一个 ServerSocket 类，并进入监听状态，第二个 int 类型的参数是监听时间长度。

ServerSocket 类的主要方法如下所示。

public Socket accept() throws IOException

ServerSocket 对象一旦创建完毕，就需要调用 accept()方法，监测网络连接，如果在指定的端口中探测到了客户端的请求，那么 accept()方法就会返回一个 Socket 对象，服务端就可以利用这个 Socket 对象与客户端的 Socket 对象建立点对点的连接，然后就可以自由地交换数据了。

public void close() throws IOException　这个方法将会关闭 ServerSocket 对象，网络连接会强行中断。

在 JSP 程序中，一般很少会用到 ServerSocket 类，因为我们编写的是访问网络服务的客户端程序，服务端程序不需要我们去编写。我们只要关心如何才能够与服务器建立连接，并向服务器发送数据，获取服务器的应答信息，至于服务器是如何处理客户端的请求的，服务端是否用 Java 语言实现的这些方面，我们根本不用去关心。

下面有两个 Java 代码片断，演示了 ServerSocket 类与 Socket 类的用法。

例：服务器上的程序。

```
ServerSocket server;
Socket ssocket;
try
{
    server = new ServerSocket(8080);
    ssocket = server.accept();
    OutputStream out = ssocket.getOutputStream();
    out.write("Test");
}
catch (IOException e)
{
    //to do it
}
```

例：用户端的程序。

```
Socket csocket;
try
{
    csocket = new Socket ("Rainbow.pku.edu.cn"，8080);
    InputStream in = csocket.getInputStream();
    System.out.println("client reads：  " + in.read());
}
catch (UnknownHostException e)
{
    //to do it
}
catch (IOException e)
{
```

北京希望电子出版社

```
        //to do it
    }
```

限于篇幅，关于 Socket 类和 ServerSocket 类的知识，我们就介绍到这里，对此感兴趣的读者可以参考相关的专著。在 11.5.3 小节，我们会介绍 SMTP 协议的具体命令，在第 11.5.4 小节，我们将会综合运用 Socket 和 SMTP 协议的知识，编写一个访问 SMTP 服务的 JSP 程序。

11.5.3　再谈 SMTP 协议

在前面我们已经简单地介绍了 SMTP 协议的知识，但在本小节里，我们还将详细地介绍 SMTP 协议的具体命令语法，因为在下一小节里，我们将会用到这些语法知识来编写一个访问 SMTP 服务的 JSP 程序。本小节的内容可能会有些晦涩难解，不过读者最好还是反复多看几遍，掌握这些知识对于提高你的编程能力是很有帮助的。

SMTP 协议的特点

简单邮件传输协议（SMTP）的目标是可靠高效地传送邮件，它独立于传送子系统而且仅要求一条可以保证传送数据单元顺序的通道(在 Java 程序中，这条通道由客户端的 Socket 类和服务段提供)。

SMTP 服务的一个重要特点是它能够在传送中接力传送邮件，传送服务提供了进程间通信环境（IPCE），此环境可以包括一个网络、几个网络或一个网络的子网。邮件可以通过连接在不同 IPCE 上的进程跨网络进行邮件传送。更特别的是，邮件可以通过不同网络上的主机接力式传送。

SMTP 协议的模型

SMTP 协议的设计基于以下通信模型：针对用户的邮件请求，发送 SMTP 邮件端建立与接收 SMTP 邮件端之间建立一个双向传送通道。接收 SMTP 邮件端可以是邮件的最终接收者也可以是中间传送者。SMTP 命令由发送 SMTP 邮件端发出，由接收 SMTP 邮件端接收，而 SMTP 应答则反方面传送。

一旦传送通道建立，SMTP 邮件发送者发送 MAIL 命令指明邮件发送者。如果 SMTP 邮件接收者可以接收邮件则返回 OK 应答。SMTP 邮件发送者再发出 RCPT 命令确认邮件是否接收到。如果 SMTP 邮件接收者接收，则返回 OK 应答；如果不能接收到邮件，则发出拒绝接收应答（但不中止整个邮件操作），双方将如此重复多次。当邮件接收者收到全部邮件后会接收到特别的序列，如果邮件接收者成功处理了邮件，则返回 OK 应答。

SMTP 协议提供传送邮件的机制，如果邮件接收方与邮件发送方连接在同一个传送服务下时，邮件可以直接由邮件发送方主机传送到邮件接收方主机；或者，当两者不在同一个传送服务下时，通过中继 SMTP 服务器传送。为了能够对 SMTP 服务器提供中继能力，它必须拥有最终目的主机地址和邮箱名称。

MAIL 命令参数是回复路径，它指定邮件从何处来；而 RCPT 命令的参数是转发路径的，它指定邮件向何处去。关于 MAIL 命令的详细信息，我们在后面还会介绍。

当同一个邮件要发往不同的接收者时，SMTP 遇到了向不同邮件接收者发送同一份数

据的复制品的问题，SMTP 命令和应答有一个比较奇怪的语法，应答也有一个数字代码。命令与应答对大小写不敏感，也就是说，SMTP 命令和应答可以是大写、小写或两者的混合，但这一点对用户邮件名称却不一定是对的，因为有的邮件主机对用户名大小写是敏感的，邮件主机的名称对大小写不敏感。

SMTP 命令

SMTP 命令定义了邮件传输或由用户定义的系统功能。它的命令是由<CRLF>结束的字符串。而在带有参数的情况下，命令本身由<SP>和参数分开，如果未带参数可以直接和<CRLF>连接。下面讨论 SMTP 命令和应答。

发送邮件操作涉及到不同的数据对象，它们由不同的参数相互连接。回复路径就是 MAIL 命令的参数，而转发路径则是 RCPT 命令的参数，邮件日期是 DATA 命令的参数。这些参数或者数据对象必须跟在命令后。这种模式也就要求有不同的缓冲区来存储这些对象，也就是说，有一个回复路径缓冲区，一个转发路径缓冲区，一个邮件内容缓冲区。特定的命令产生自己的缓冲区，或使一个或多个缓冲的内容被清除。

1. HELLO (HELO)

此命令用于向接收 SMTP 邮件端确认发送 SMTP 邮件。参数域包括发送 SMTP 邮件的主机名。接收 SMTP 邮件端通过连接确认命令来向发送 SMTP 邮件端确认接收 SMTP 邮件。

2. MAIL (MAIL)

此命令用于开始将邮件发送到一个多个邮箱中。参数域包括回复路径。返回路径中包括了可选的主机和发送者邮箱列表。当有主机列表时，它是一个回复路径源，它说明此邮箱是由在表中的主机一一传递发送（第一个主机是最后一个接收到此邮件的主机）过来的。此表也有作向发送者返回非传递信号的源路径。因为每个传递主机地址都被加在此表起始处，它就必须使用发送 IPCE 而不是接收 IPCE（如果它们不是一个 IPCE 的话）清楚的名称。一些出错信息的回复路径可能就是空的。

此命令清除回复路径缓冲区，转发路径缓冲区和邮件内容缓冲区，并且将此命令的回复路径信息插入到回复路径缓冲区中。

3. RECIPIENT (RCPT)

此命令用于确定邮件内容的唯一接收者；多个接收者将由多个此命令指定。转发路径中包括一个可选的主机和一个必需的目的邮箱的地址。当出现主机列表时，这就是一个源路径，它指明邮件必须向列表中的上一个主机发送。如果接收 SMTP 邮件端未实现邮件的传递发送，就会返回如未知本地用户（550）的信息给用户。

当邮件被传递发送时，传递主机必须将自己的名称由转发路径的开始处移至回复路径的结束处。当邮件最终到达目的地时，接收 SMTP 邮件端将以它的主机邮件格式自己的名称插入目标邮件中。例如，由传递主机 A 接收的带有如下参数的邮件时，

FROM：<USERX@HOSTY.ARPA>

TO：<@HOSTA.ARPA，@HOSTB.ARPA：USERC@HOSTD.ARPA>

将会变成如下形式：

FROM：<@HOSTA.ARPA：USERX@HOSTY.ARPA>

TO：<@HOSTB.ARPA：USERC@HOSTD.ARPA>.

此命令导致它的转发路径参数加入转发路径缓冲区中。

4. DATA (DATA)

SMTP 邮件接收端将跟在该命令后的行作为邮件内容。此命令导致此命令后的邮件内容加入邮件内容缓冲区。邮件内容可以包括所有 128 个 ASCII 码字符。邮件内容由只包括一个句号的行结束，也就是如下的字符序列："<CRLF>.<CRLF>"，它指示了邮件的结束。

邮件内容的结束指示要求邮件接收者现在就处理保存的邮件内容。此过程将回复路径缓冲区，转发路径缓冲区和邮件内容缓冲区的内容全部清空。如果操作成功，邮件接收者必须返回 OK 应答；如果失败也必须返回失败应答。

当接收 SMTP 邮件端收到一条信息时，无论是用作转发还是此邮件已经到达目的地，它都必须在邮件内容的开始处加上时间戳这一行，这一行指示了接收到邮件主机和发出此邮件主机的标识，以及接收到邮件内容的时间和日期。转发的信件将有多行这样的时间戳。当接收 SMTP 邮件端作最后一站的传送时，它将返回路径信息行插入邮件中。此行包括了发送命令中的<reverse-path>的信息。在这里，最后一站传送的意思是邮件将被送到目的用户手中，但在一些情况下，邮件可能需要更进一步的加工并由另外的邮件系统传送。

可能在返回路径中的邮箱与实际发送的邮件不一致，这个情况可能发生在需要传送一个特定的错误处理信箱而不是信件发送者那里。上面所述说明了，最后的邮件内容由一个返回路径行，和在其后的一个或多个时间戳行构成。这些行后面是邮件内容的头和体信息。

当处理后面的邮件数据指示部分成功时就需要特定的说明。这种情况可能发生在发送 SMTP 邮件端发现当邮件需要传送给多个用户时，只能够成功地向其中的一部分发送信息这种情况下。在这种情况下，必须对 DATA 命令发送 OK 应答，而接收 SMTP 邮件端组织并发送一个“不可传递邮件”信息到信息的发送者。在此信息中或者发送一个不成功接收者的列表，或者每次发送一个不成接收者，而发送多次。所有不可传递邮件信息由 MAIL 命令发送。

5. SEND (SEND)

此命令用于开始一个发送命令，将邮件发送到一个或多个终端上。参数域包括了一个回复路径，此命令如果成功就将邮件发送到终端上了。

回复路径包括一个可选的主机列表和发送者邮箱。当出现主机列表时，表示这是一个传送路径，邮件就是经过这个路径上的每个主机发送到这里的（列表上第一个主机是最后经手的主机）。此表用于返回非传递信号到发送者。因为每个传递主机地址都被加在此表起始处，它就必须使用发送 IPCE 而不是接收 IPCE（如果它们不是一个 IPCE 的话）清楚的名称。一些出错信息的回复路径可能就是空的。

此命令清除回复路径缓冲区，转发路径缓冲区和邮件内容缓冲区，并且将此命令的回复路径信息插入到回复路径缓冲区中。

6. SEND OR MAIL (SOML)

此命令用于开始一个邮件操作将邮件内容传送到一个或多个终端上，或者传送到邮箱中。对于每个接收者，如果接收者终端打开，邮件内容将被传送到接收者的终端上，否则就送到接收者的邮箱中。参数域包括回复路径，如果成功地将信息送到终端或邮箱中此命令成功。

回复路径包括一个可选的主机列表和发送者邮箱。当出现主机列表时，表示这是一个

传送路径，邮件就是经过这个路径上的每个主机发送到这里的（列表上第一个主机是最后经手的主机）。此表用于返回非传递信号到发送者。因为每个传递主机地址都被加在此表起始处，它就必须使用发送 IPCE 而不是接收 IPCE（如果它们不是一个 IPCE 的话）清楚的名称。一些出错信息的回复路径可能就是空的。

此命令清除回复路径缓冲区，转发路径缓冲区和邮件内容缓冲区，并且将此命令的回复路径信息插入到回复路径缓冲区中。

7. SEND AND MAIL (SAML)

此命令用于开始一个邮件操作将邮件内容传送到一个或多个终端上，并传送到邮箱中。如果接收者终端打开，邮件内容将被传送到接收者的终端上和接收者的邮箱中。参数域包括回复路径，如果成功地将信息送到邮箱中此命令成功。

回复路径包括一个可选的主机列表和发送者邮箱。当出现主机列表时，表示这是一个传送路径，邮件就是经过这个路径上的每个主机发送到这里的（列表上第一个主机是最后经手的主机）。此表用于返回非传递信号到发送者。因为每个传递主机地址都被加在此表起始处，它就必须使用发送 IPCE 而不是接收 IPCE（如果它们不是一个 IPCE 的话）清楚的名称。一些出错信息的回复路径可能就是空的。

此命令清除回复路径缓冲区，转发路径缓冲区和邮件内容缓冲区，并且将此命令的回复路径信息插入到回复路径缓冲区中。

8. RESET (RSET)

此命令指示当送邮件操作将被放弃。任何保存的发送者，接收者和邮件内容应该被抛弃，所有缓冲区和状态表应该被清除，接收方必须返回 OK 应答。

9. VERIFY (VRFY)

此命令要求接收者确认参数是一个用户。如果这是（已经知道的）用户名，返回用户的全名和指定的邮箱。此命令对回复路径缓冲区、转发路径缓冲区和邮件内容缓冲区没有影响。

10. EXPAND (EXPN)

此命令要求接收者确认参数指定了一个邮件发送列表，如果是一个邮件发送列表，就返回表中的成员。如果这是（已经知道的）用户名，返回用户的全名和指定的邮箱。此命令对回复路径缓冲区，转发路径缓冲区和邮件内容缓冲区没有影响。

11. HELP (HELP)

此命令导致接收者向 HELP 命令的发送者发出帮助信息。此命令可以带参数，并返回特定的信息作为应答。此命令对回复路径缓冲区，转发路径缓冲区和邮件内容缓冲区没有影响。

12. NOOP (NOOP)

此命令不影响任何参数和已经发出的命令。它只是说明没有任何操作而不是说明接收者发送了一个 OK 应答。此命令对回复路径缓冲区，转发路径缓冲区和邮件内容缓冲区没有影响。

13. QUIT (QUIT)

此命令指示接收方必须发送 OK 应答然后关闭传送信道。接收方在接到 QUIT 命令并做出响应之前不应该关闭通信信道。发送方在发送 QUIT 命令和接收到响应之前也不应该

关闭信道。即使出错，也不应该关闭信道。如果连接被提前关闭，接收方应该像接收到 RSET 命令一样，取消所有等待的操作，但不恢复原先已经做过的操作。而发送方应该像接收到暂时错误（4XX）一样假定命令和操作仍在支持之中。

14. TURN (TURN)

此命令指定接收方要么发送 OK 应答并改变角色为发送 SMTP 邮件端，要么发送拒绝信息并保持自己的角色。如果程序 A 现在是发送 SMTP 邮件端，它发出 TURN 命令后接收到 OK（250）应答，它就变成了接收 SMTP 邮件端。程序 A 就进入初始状态，好象通信信道刚打开一样，这时它发送 220 准备好服务信号。如果程序 B 现在是接收 SMTP 邮件端，它发出 TURN 命令后接收到 OK（250）应答，它就变成了发送 SMTP 邮件端。程序 A 就进入初始状态，好象通信信道刚打开一样，这时它准备接收 220 准备好服务信号。

若要拒绝改变角色，接收方可以发送 502 应答。

（1）SMTP 命令规则。

对于这些命令的顺序有一定的限制。对话的第一个命令必须是 HELLO 命令，此命令在此后的会话中也可以使用。如果 HELLO 命令的参数不可接受，必须由返回一个 501 失败应答，同时接收到的 SMTP 必须保持在与刚才一致的状态下。

NOOP，HELP，EXPN 和 VRFY 命令可以在会话的任何时候使用。MAIL，SEND，SOML 或 SAML 命令开始一个邮件操作。一旦开始了以后就要发送 RCPT 和 DATA 命令。邮件操作可以由 RSET 命令终止。在一个会话中可以有一个或多个操作。

如果在操作开始参数不可接受，必须返回 501 失败应答，同时接收到的 SMTP 邮件必须保持在与刚才一致的状态下。如果操作中的命令顺序出错，必须返回 503 失败应答，同时接收到的 SMTP 邮件必须保持在与刚才一致的状态下。

会话的最后一个命令必须是 QUIT 命令。此命令在会话的其它时间不能使用。

命令是由命令码和其后的参数域组成的。命令码是四个字母组成的，不区别大小写。因为下面命令的作用是相同的：

MAIL Mail mail MaIl mAIl

这对于引导任何参数值的标记也是适用的，如 TO 和 to 就是一样的。命令码和参数由一个或多个空格分开。然而在回复路径和转发路径中的参数是区别大小写的。特别是在一些主机上，"smith"和"Smith"就根本不是一个用户。

参数域由不定长的字符串组成，它由<CRLF>结束，接收方在完全接收到此序列前不会采取任何行动。方括号代表可选的参数域。如果不选择的话，系统选择默认的设置。

对 SMTP 命令的响应是多样的，它确定了在邮件传输过程中请求和处理的同步，也保证了发送 SMTP 邮件端知道接收 SMTP 邮件端的状态。每个命令必须有且只有一个响应。

SMTP 响应由三位数字组成，其后跟一些文本。数字帮助决定下一个应该进入的状态，而文本对人是有意义的。三位的响应已经包括了足够的信息，不用再阅读文本，文本可以直接抛弃或者传递给用户。特别的是，文本是与接收和环境相关的，所以每次接收到的文本可能不同。在附录 E 中可以看到全部的响应码。正规的情况下，响应由下面序列构成：三位的数字，<SP>，一行文本和一个<CRLF>，或者也可以是一个多行响应。只有 EXPN 和 HELP 命令可以导致多行应答，然而，对所有命令，多行响应都是允许的。

（2）SMTP 服务器的响应码列表。

REPLY CODES BY FUNCTION GROUPS
500 格式错误，命令不可识别（此错误也包括命令行过长）
501 参数格式错误
502 命令不可实现
503 错误的命令序列
504 命令参数不可实现
211 系统状态或系统帮助响应
214 帮助信息
220 <domain> 服务就绪
221 <domain> 服务关闭传输信道
421 <domain> 服务未就绪，关闭传输信道（当必须关闭时，此应答可以作为对任何命令的响应）
250 要求的邮件操作完成
251 用户非本地，将转发向<forward-path>
450 要求的邮件操作未完成，邮箱不可用（例如，邮箱忙）
550 要求的邮件操作未完成，邮箱不可用（例如，邮箱未找到，或不可访问）
451 放弃要求的操作；处理过程中出错
551 用户非本地，请尝试<forward-path>
452 系统存储不足，要求的操作未执行
552 过量的存储分配，要求的操作未执行
553 邮箱名不可用，要求的操作未执行（例如邮箱格式错误）
354 开始邮件输入，以<CRLF>.<CRLF>结束
554 操作失败

注意：

<CRLF>表示回车

<SP>表示空格

11.5.4　使用 Socket 访问 SMTP 服务

SMTP 协议是目前网上流行的发送 E-mail 的协议，上面已经提到，SMTP 协议一共支持 14 条命令。不过，一般来说，发一封 E-mail 只需用如下 5 条命令就足够了。

HELO <SP> <domain> <CRLF> 与SMTP服务器握手，传送本机域名
MAIL <SP> FROM：<reverse-path> <CRLF>传送发信者的信箱名称
RCPT <SP> TO：<forward-path> <CRLF> 传送接收者的信箱名称
DATA <CRLF> 发送信件数据（包括信头和信体）
QUIT <CRLF> 退出与SMTP服务器的连接

我们如何把 Socket 和 SMTP 协议结合起来，编写访问 SMTP 服务的 JSP 程序呢?我们的思路是这样的，首先创建一个 Socket 对象，与 SMTP 服务端建立连接，然后使用输出流对象向服务端发出上述 SMTP 命令，使用输入流对象读取 SMTP 服务器的应答信息。请看程序清单 11.5(socketsmtp.jsp)。

程序清单 11.5

```
<%--
File Name：smtp.jsp
Author：fancy
```

```
Date：2001.4.8
Note：Use smtp service to send mail
--%>

<%@page import="sun.net.smtp.*" %>
<%@page import="java.io.*" %>
<%@page import="java.net.*" %>

<%
Socket csocket;
try
{
      csocket = new Socket ("Rainbow.pku.edu.cn"，25);
      InputStream is=csocket.getInputStream();
      OutputStream os=csocket.getOutputStream();
      BufferedReader in=new BufferedReader(new InputStreamReader(is));
      out.println("client reads-->" +in.readLine()+"<br>");
      PrintStream ps=new PrintStream(os);
      ps.println("HELO");
      out.println("client reads-->" +in.readLine()+"<br>");
      ps.println("MAIL FROM：fancyrainbow@263.net");
      out.println("client reads-->" +in.readLine()+"<br>");
      ps.println("RCPT TO：Rainbow@Rainbow.pku.edu.cn");
      out.println("client reads-->" +in.readLine()+"<br>");
      ps.println("RCPT TO：fancy@Rainbow.pku.edu.cn");
      out.println("client reads-->" +in.readLine()+"<br>");
      ps.println("DATA");
      out.println("client reads-->" +in.readLine()+"<br>");
      ps.println("Hello World!");
      //out.println("client reads-->" +in.readLine()+"<br>");
      ps.println(".");
      out.println("client reads-->" +in.readLine()+"<br>");
      ps.println("QUIT");
      out.println("client reads-->" +in.readLine()+"<br>");
      //ps.close();
      csocket.close();
}
catch (Exception fe)
{
      out.println(fe.getMessage());
}
%>
```

在程序清单 11.6 中，实现了我们原来的构想，使用 PrintStream 对象的 println()方法发

送完邮件正文后，必须要向服务器发送一个点号(.)，标志邮件已经结束，否则邮件服务器会一直处于等待状态。在程序的最后，向邮件服务器发送 QUIT 命令，结束这个服务进程。

socketsmtp.jsp 程序的运行效果如图 11.14 所示。

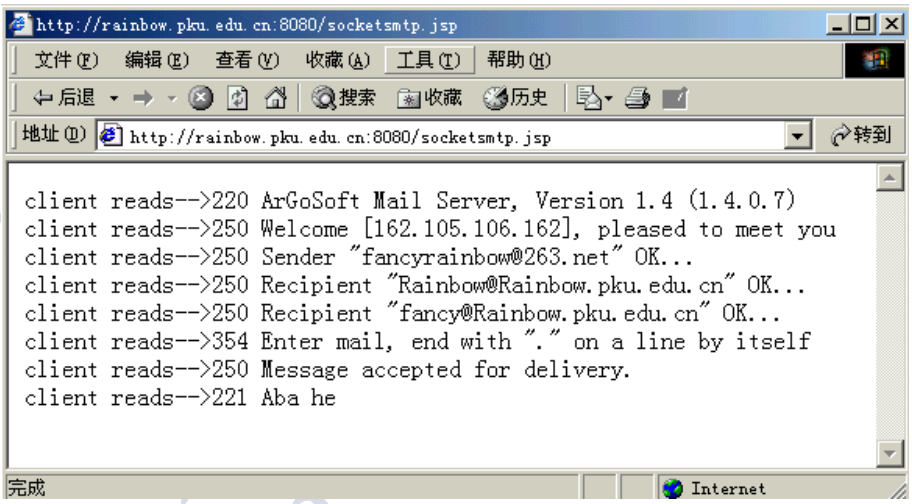


图 11.14　socketsmtp.jsp 程序的运行效果

使用 Outlook 软件查看你的账号，就会看到这封邮件了，如图 11.15 所示。

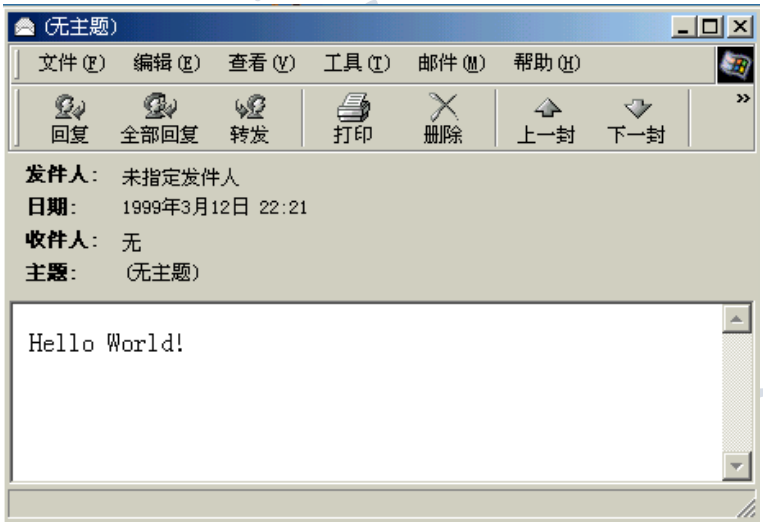


图 11.15　收到新邮件啦

其实，我们只要知道了 Internet 协议的具体内容，就可以按照上面的方法，编写类似的程序了。不过我们不建议读者使用这种方法，因为这种方法需要考虑的底层细节太多了，开发起来效率不高，如果我们不是做底层系统的开发，最好不要用 Socket 直接访问服务器，而应该使用比较高层的客户端类，如 FtpClient 类、SmtpClient 类。

11.5.5　使用 Socket 访问 FTP 服务

在本节中，我们将演示如何利用底层的 Socket 类和 FTP 协议编写一个访问 FTP 服务的 JSP 程序。由于篇幅的关系，我们只是把程序清单和运行结果列出来，至于 FTP 协议的内容和程序的解说就略过不提了。不过这个程序和程序清单 11.5(socketsmtp.jsp)十分相近，FTP 协议和 SMTP 协议的命令也有一定的相似性，因此读者应该不难看懂这个程序。请看程序清单 11.6(ftp2.jsp)。

程序清单 11.6

```
<%--
File Name：ftp2.jsp
Author：fancy
Date：2001.4.9
Note：Use socket to visit ftp service
--%>

<%@ page contentType="text/html;charset=gb2312" %>
<%@ page import="java.io.*"%>
<%@ page import="java.util.*"%>
<%@ page import="java.net.*"%>
<%
System.out.println("Hello World!");
String host="ftp.lib.pku.edu.cn";
String user="anonymous";
String pass="fancyrainbow@263.net";
Socket client;
PrintStream ps;
BufferedReader in;
try
{
    client=new Socket(host，21);
    ps=new PrintStream(client.getOutputStream()，true);
    in=new BufferedReader(new InputStreamReader(client.getInputStream()));
    String c;
    ps.println("USER anonymous");
    ps.println("PASS fancyrainbow@263.com");
    ps.println("SYST");
    ps.println("REST 100");
    ps.println("REST 0");
    ps.println("PWD");
    ps.println("TYPE A");
    ps.println("PORT 162，105，106，162，6，164");
    ps.println("CWD pub1");
    ps.println("NLST");
```

北京希望电子出版社

```
        ps.println("PASV");
        ps.println("QUIT");
        while ((c=in.readLine())!=null)
        {
              out.println(c+"<br>");
        }
        in.close();
        out.close();
        client.close();
}
catch(Exception fe)
{
        System.out.println(fe.getMessage());
}
%>
```

程序清单 11.6 的运行效果如下所示：

```
220-
220- Welcome to FTP.LIB.PKU.EDU.CN
220- ============================
220- HTTP：//FTP.LIB.PKU.EDU.CN
220- HTTP：//FTP.LIB.PKU.EDU.CN：81
220-
220-
220 sun3000 FTP server (Version wu-2.6.0(1) Thu Oct 28 12：59：35 CST 1999) ready.
331 Guest login ok，send your complete e-mail address as password.
230-
230- Welcome to FTP Server of Peking University Library
230- ================================================
230-
230- HTTP：//FTP.LIB.PKU.EDU.CN：81
230- HTTP：//SUN3000.LIB.PKU.EDU.CN：81
230- FTP：//FTP.LIB.PKU.EDU.CN
230- FTP：//SUN3000.LIB.PKU.EDU.CN
230-
230- 40 Anonymous users allowed，3 users online.
230-
230- Use http：//pccms.pku.edu.cn：8000/ftps.htm to search.
230-
230-  本站严禁使用5个进程以上下载，违者禁止其C类地址的访问权限
230-  请大家给与配合，谢谢。
230-
230- User anonymous from 162.105.106.162，Tue Apr 10 08：54：07 2001
230-
```

北京希望电子出版社

```
230-
230 Guest login ok，access restrictions apply.
215 UNIX Type：  L8
350 Restarting at 100. Send STORE or RETRIEVE to initiate transfer.
350 Restarting at 0. Send STORE or RETRIEVE to initiate transfer.
257 "/" is current directory.
200 Type set to A.
200 PORT command successful.
250 CWD command successful.
550 pub1：  No such file or directory.
227 Entering Passive Mode (162，105，140，202，222，255)
221-You have transferred 0 bytes in 0 files.
221-Total traffic for this session was 1654 bytes in 0 transfers.
221-Thank you for using the FTP service on sun3000.
221 Goodbye.
```

11.6　Telnet 服务

Telnet 服务，亦即远程登录服务，利用 Telnet 服务，我们可以远程登录主机进行管理操作，Telnet 服务相当于给我们提供了一个 Shell。Telnet 服务要求客户端与服务端建立不间断的连接，以便进行交互操作，所以，利用 JSP 程序来编写 Telnet 客户端程序并不合适，但是这并不是说在 JSP 程序没有办法访问 Telnet 服务，在 JSP 程序中仍然有办法访问 Telnet 服务，请看下面的程序清单 11.7(telnet1.jsp)。

程序清单 11. 7

```
<%--
File Name：telnet1.jsp
Author：fancy
Date：2001.5.4
Note：to telnet the server
--%>
<%@ page import="java.net.*"%>
<%@ page import="java.io.*"%>
<%@ page import="sun.net.*"%>
<%
    try
    {
        String host="162.105.106.162";
        NetworkClient nc=new NetworkClient(host，23);
        PrintStream ps=nc.serverOutput;
        InputStream is=nc.serverInput;
        ps.println();
        ps.println("c：");
        //ps.println("cd progra~1");
```

```
ps.println("dir");

BufferedReader in=new BufferedReader(new InputStreamReader(is));
String temp;
temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
//out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
//out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
//out.println("response info："+temp+"<br>");

temp=in.readLine();
//out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
//out.println("response info："+temp+"<br>");
temp=in.readLine();
//out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
ps.println("exit");

temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
//out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
temp=in.readLine();
out.println("response info："+temp+"<br>");
```

```
            temp=in.readLine();
            out.println("response info："+temp+"<br>");
            temp=in.readLine();
            out.println("response info："+temp+"<br>");
            temp=in.readLine();
            out.println("response info："+temp+"<br>");

            temp=in.readLine();
            out.println("response info："+temp+"<br>");
            temp=in.readLine();
            out.println("response info："+temp+"<br>");
            temp=in.readLine();
            out.println("response info："+temp+"<br>");
            ps.println("exit");
            in.close();
            ps.close();
        }
        catch(Exception fe)
        {
            out.println("here");
            out.println(fe.getMessage());
        }
%>
```

程序清单 11.7 的运行效果如图 11.16 所示。

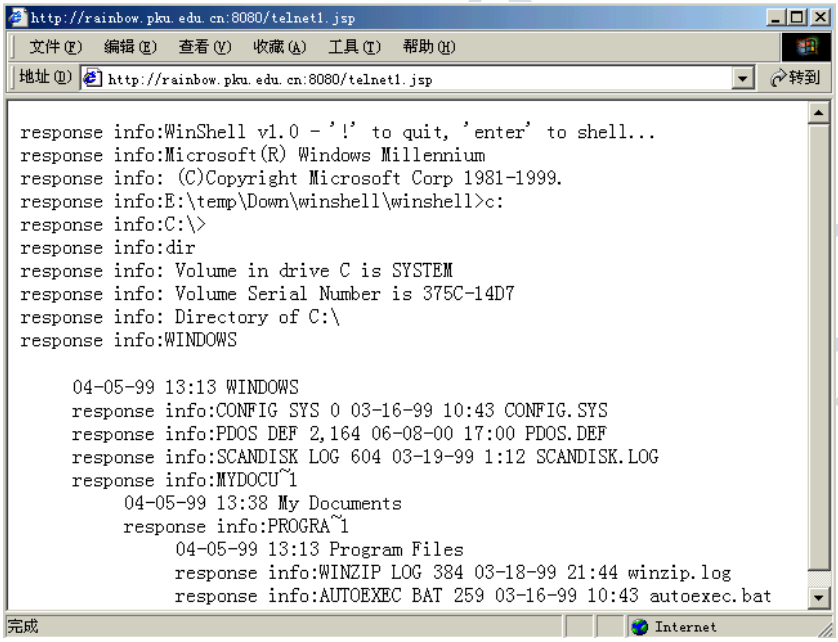


图 11.16　telnet1.jsp 程序访问 telnet 服务

程序清单 11.7 的流程和前面几个程序差不多，都是建立一个客户端对象，然后获取对服务端的输入流与服务端对客户端的输出流，利用输入流发送命令，利用输出流读取数据，所以我们就不详细介绍这个程序了。我们建议读者不要用 JSP 程序来实现 Telnet 客户端，可以使用 Applet 程序或者 Application 类型的程序来实现 Telnet 服务的客户端，我们之所以举这个例子只是为了说明利用 JSP 技术能干什么，而不是推荐使用 JSP 技术干什么。

11.7　本 章 小 结

在本章中，我们介绍了如何利用 Java 语言的 sun.net.*.*类库，编写访问 Internet 服务的 JSP 程序。还讨论了 Mail、FTP 服务器的配置问题，在本章的后半部分，我们结合 SMTP 协议与 Java 语言的 Socket 类，编写了一个访问 SMTP 服务的 JSP 程序，这种编程技巧不足取，但是这种解决问题的方法的确很重要，因为有一些 Internet 服务还没有对应的 Java 客户端类，例如 echo 服务、finger 服务等，这时我们就需要利用到 Socket 类和相应的协议，自己编写可用的客户端了。

在第 12 章，也就是本书的最后一章，我们将向读者介绍强大的 Java Mail API，读者将会学到如何利用 Java Mail API 构建强大的邮件系统。

北京希望电子出版社

第 12 章　Java Mail API

Java Mail API 是 Sun 公司最新开发的邮件 API，可以完成复杂的邮件处理，这个 API 的功能比我们前面所介绍的 sun.net.smtp 包的功能强大多了，它不但可以访问 SMTP 服务，而且可以访问 POP3 服务，IMAP 服务。在本章中，我们将要介绍 Java Mail API 的基本知识。当读者学完本章以后，就可以独自建立一个类似于 freemail.263.net 的邮件系统接口了。

本章需要重点掌握的内容如下所示：

- javax.mail 包
- javax.mail.internet 包
- Sun Protocol Privider API
- 使用 Java Mail API 访问 Mail 服务器

12.1　Java Mail API 简介

Java Mail API 的开发是 Sun 公司为 Java 程序开发者提供公用 API 框架的持续努力的良好例证。提倡公用框架，反对受限于特定供应商的解决方案，充分预示着一个日益开放的开发环境的建立。

在 email 通讯领域，面向最终应用的开发者（以及用户）已经能够购买到最适合他们使用的公用 API 框架实现，而低层开发者能够提供有效访问特定邮件服务的解决方案。其意义在于：小型开发组能够集中精力于开发高性能的客户端邮件软件，而为它提供不同邮件环境下的支持则变得相当容易;大型开发组织则侧重于为新开发的企业级邮件服务提供公用的访问支持，并由此获得丰富的应用软件支持。最大的赢家是信息系统用户，它能够按照要求（性能、投资等）方便地集成由不同厂商提供的产品和方案。

开发高度可重用的开放 API 框架的关键之一在于强调抽象接口技术，即在支持现有标准的基础上，支持未来可能的扩展。Java Mail API 体现了这一思想。Sun 公司和其它开发商正在为大多数现有的公用邮件服务系统标准和邮件协议提供缺省实现和工具。已经可用的支持至少包括：POP3、SMTP、IMAP 等邮件协议。

Java Mail API 的结构本身证明了它的开发者的基本目标之一——软件开发的工作量应该取决于应用程序本身的复杂程度以及开发者所要求的控制程度。换句话说，Java Mail API 尽可能地保持简单。本章的示例程序充分说明了这一点。乍看起来，Java Mail API 所拥有的类总数以及类之间的关系可能让人误解为需要漫长的学习时间。实际上，一旦正式开始使用，你就会发现该 API 不失为在应用程序中加入健壮的邮件/通讯支持的简单工具。

Java Mail API 包括的类数量远远大于此处涉及的类数量。但是核心类却不多，如下所示：

javax.mail.Session　Session 类是 Java Mail API 最高层入口类。它最常用的方法用于为不同邮件协议控制和装载 SPI（即 Service Provider Implementation，译为服务提供接口）。如 Store 类(代表邮箱)是通过 Session 类获得的。

javax.mail.Store　Store 类实现特定邮件协议上的读、写、监视、查找等操作。通过 Store 类可以访问 Folder 类(代表邮箱文件夹)。

javax.mail.Transport　Transport 类也是由服务提供者提供的类，实现用特定协议发送消息/邮件。

javax.mail.Folder　Folder 类用于分级组织邮件，并提供访问 email 的能力，这个能力是通过获取 Message 类(代表邮件)的实例而实现的。

javax.mail.Message　Message 类模型化实际 email 消息的所有细节，如标题、发送/接收地址、发送日期等等。

值得一提的是，Java Mail API 实际上依赖于另外一个 Java 扩展——JAF，即 JavaBean 活动框架（JavaBean Activation Framework）。JAF 的目的在于统一处理不同数据格式的方法(不管数据格式为简单文本还是由图片、声音、视频甚至其它“活动”内容共同组成的复合文档)。在这个意义上，JAF 对 Java 的作用正如插件(Plug-ins)对 Web 浏览器的作用。

12.2　javax.mail 包

这一节我们将要介绍 javax.mail 包。Java Mail API 其实分为两大部份，第一部分是 Java Mail 1.2 API，含有 javax.mail 包、javax.mail.internet 包、javax.mail.event 包、javax.mail.search 包。第二部分是 Sun Protocol Provider API，含有 com.sun.mail.pop3 包、com.sun.mail.imap 包、com.sun.mail.smtp 包。

javax.mail 包主要用于模拟一个邮件系统。

12.2.1　Session 类

在上文中已经提到过 Session 类是一个十分重要的类，Session 类映射了客户端与 Mail Server 之间的会话过程，利用 Session 类，我们就可以在客户端与 Mail Server 之间建立会话过程，并进而访问邮箱、文件夹、邮件;或者是利用 SMTP Server 发送邮件，可以说 Session 类就是邮件系统的访问入口。

在 Session 类中定义的重要方法如下所示：

public boolean getDebug();　获取当前会话过程关于 debug 的设置信息。如果当前会话过程允许 debug，那么客户端与服务端之间的交互过程信息会在程序控制台中输出，如果是 JSP、Servlet 等程序，那么这些信息会在 Web Server 的控制台中输出的。我们建议读者利用 setDebug()方法把 debug 设置为 true，这样一来就可以看到客户端是如何往服务端发送命令的，而服务端又是如何响应客户端请求的，这对于我们学习 SMTP 协议和 POP3 协议有很大的帮助。

public static Session getDefaultInstance(java.util.Properties props);　该方法可以利用缺省的系统属性，建立一个缺省的会话过程，该方法的返回值是 Session 对象，用法示例如下：

```
Properties props = System.getProperties();
Session sess= Session.getDefaultInstance(props);
Store store = null;
```

```
store = sess.getStore(protocol);
store.connect(host，user，password);
```

读者请注意：创建了 Session 对象并不就意味着已经与服务端建立连接了，我们还需要调用 Store 类的 connect()方法才能够与服务器真正建立起有效的连接。

该方法的参数是一个 Properties 对象，主要用于传递环境属性信息，除了系统(System)的环境属性信息以外，Java Mail API 中还规定了另外几个与 Mail Server 相关的属性，如下所示：

mail.store.protocol：访问邮箱的时候使用那种协议，POP3 或者是 IMAP。

mail.transport.protocol：发送邮件的时候使用那种协议。

mail.host：邮件服务器的主机名。

mail.user：邮箱用户名。

mail. protocol.host：同 mail.host 属性。这里 protocol 可以用 smtp、pop3 或者 imap 代替。

mail. protocol.user：同 mail.user。这里 protocol 可以用 smtp、pop3 或者 imap 代替。

mail.from：邮件的发信人地址。

mail.debug：会话过程是否处于 debug 状态。

我们可以使用 Properties 对象的 put()方法，根据需要设定这些属性的值，然后再把 Properties 对象传递给 getDefaultInstance()方法。

例：

```
Properties props = System.getProperties();
props.put("mail.smtp.host"，"smtp.263.net");
Session sess= Session.getDefaultInstance(props);
```

public static Session getDefaultInstance(java.util.Properties props，Authenticator authenticator); 该方法和上文的同名方法功能一样，都是创建一个缺省的会话过程，不过有些服务器为了防止某些别有用心的用户利用 SMTP 服务大量发送垃圾邮件，管理员就给服务器加上了要求身份验证的控制，只有经过服务器身份验证的用户，才有权利用 SMTP 服务发送邮件。这样一来，上文的同名方法只能够对不要求身份验证的邮件主机使用，而这个方法既适用于要求身份验证的情况，又适用于不要求身份验证的情况。这个方法的第一个参数仍然是 Properties 对象，第二个参数是 Authenticator 对象，该对象包含了通过身份验证所需要的全部信息，如果服务器不要求验证身份，那么把这个参数设为 null 值即可，此时这两个方法没有任何区别。

我们在 12.5 节会介绍如何进行身份验证的知识。

public Folder getFolder(URLName url) throws MessagingException; 获取邮箱中文件夹，返回值是一个 Folder 对象。该方法参数是 URLName 对象，关于 URLName 类的知识，读者可以参考 Java Mail API 的文档，我们在本章中不予介绍。

在实际编程中，我们一般是通过 Store 对象的 getFolder()方法获得 Folder 对象，很少利用 Session 对象的 getFolder()方法直接获得 Folder 对象。

public static Session getInstance(java.util.Properties props); 该方法和 getDefaultInstance()方法(指单参数的 getDefaultInstance())一样，不过有少许区别，前者直接

创建一个新的 Session 对象，而后者却首先检查有没有已经存在的 Session 对象，如果有符合要求的 Session 对象存在，那么就返回这个 Session 对象，否则，才创建一个新的 Session 对象。

public static Session getInstance(java.util.Properties props，Authenticator authenticator); 该方法和 getDefaultInstance()方法(指双参数的 getDefaultInstance())一样，不过有少许区别，前者直接创建一个新的 Session 对象，而后者却首先检查有没有已经存在的 Session 对象，如果有符合要求的 Session 对象存在，那么就返回这个 Session 对象，否则，才创建一个新的 Session 对象。

public java.util.Properties getProperties(); 获取会话过程的属性信息。该方法的返回值为 Properties 对象，其实这个方法将返回我们在创建 Session 对象时，作为参数传递给 getInstamce()、getDefaultInstance()等方法的 Properties 对象。

public String getProperty(String name); 设定当前会话过程的某些属性的值，方法参数 name 指的是属性的名称，例如 mail.host。

public Provider getProvider(String protocol) throws NoSuchProviderException; 该方法可以获知某种协议对应哪一种服务，在 Java Mail API 中，定义了三种服务，分别是 SMTP Transport 服务、IMAP Store 服务、POP3 Store 服务，第一个是邮件传输服务，第二、第三个是邮箱服务，这三种服务分别对应的协议是：SMTP 协议、IMAP 协议、POP3 协议。示例代码段如下所示。

例：

```
<%
Provider []pr=sess.getProviders();
out.println(pr[i].getClassName()+"<br>");
%>
```

public Provider[] getProviders(); 该方法和 getProvider 方法差不多，不过返回值为一个 Provider 类型的数组，代码实例如下所示。

例：

```
<%
Provider []pr=sess.getProviders();
for(int i=0;i<pr.length;i++)
{
    out.println(pr[i].getClassName()+"---------->");
    out.println(pr[i].getProtocol()+"---------->");
    out.print(pr[i].getVendor()+"<br>");
}
%>
```

执行该代码的输出结果为：

```
com.sun.mail.imap.IMAPStore----------> imap----------> Sun Microsystems，Inc
com.sun.mail.pop3.POP3Store----------> pop3----------> Sun Microsy stems，Inc
com.sun.mail.smtp.SMTPTransport----------> smtp----------> Sun Microsystems，Inc
com.sun.mail.imap.IMAPStore----------> imap----------> Sun Microsystems，Inc
```

北京希望电子出版社

com.sun.mail.smtp.SMTPTransport----------> smtp----------> Sun Microsystems，Inc

com.sun.mail.pop3.POP3Store----------> pop3----------> Sun Microsystems，Inc

public Store getStore() throws NoSuchProviderException;　该方法可以获取一个缺省的邮箱对象，至于是 IMAP 邮箱还是 POP3 邮箱，那就要看当前会话过程的 mail.store.protocol 属性的值是 imap 还是 pop3 了。

public Store getStore(Provider provider) throws NoSuchProviderException;　作用和上文的同名方法一样，只不过方法参数略有差异。

public Store getStore(String protocol) throws NoSuchProviderException;　作用和上文的同名方法一样，只不过方法参数略有差异。该方法的参数为协议的名字，如果参数为 pop3，那么这个方法将返回 POP3 的邮箱对象，如果参数为 imap，那么这个方法将返回 IMAP 的邮箱对象。

public Store getStore(URLName url) throws NoSuchProviderException;　作用和上文的同名方法一样，只不过方法参数略有差异。这里的参数为 URLName 对象，这个方法较少被用到。

public Transport getTransport() throws NoSuchProviderException;　该方法返回一个 Transport 对象，Transport 对象用于发送邮件。

public Transport getTransport(Address address) throws NoSuchProviderException; 同上，不过这个方法通过传递 Address 对象指定了发送邮件的目标地址。

public Transport getTransport(String protocol) throws NoSuchProviderException; 该方法返回一个 Transport 对象，Transport 对象用于发送邮件。使用参数 protocol 指定了传输邮件所使用的协议名称。

public Transport getTransport(Provider provider) throws NoSuchProviderException; 同上，不过这个方法指定的是 Provider 而不是 Protocol。

public Transport getTransport(URLName url) throws NoSuchProviderException; 该方法返回一个 Transport 对象，Transport 对象用于发送邮件。这个方法通过传递 URLName 对象指定了发送邮件的目标地址。

public void setDebug(boolean debug);　设定当前会话过程是否处于会话状态，如果为 true，那么会话过程处于 debug 状态，客户端与服务端的交互操作信息都会在程序控制台中输出，如下所示。

例(仅仅是输出信息的一部分)：

```
C: TOP 13 0
S: +OK core mail
Received: from www8.kaxiu.com (unknown [202.103.25.226])
        by mx8.263.net (Postfix) with ESMTP id DFA551C66456B
        for <fancyrainbow@263.net>; Tue,1 May 2001 13:48:02 +0800 (CST)
Received: (from nobody@localhost)
        by www8.kaxiu.com (8.9.3/8.9.3) id NAA09901;
        Tue,1 May 2001 13:35:20 +0800
Date: Tue,1 May 2001 13:35:20 +0800
Message-Id: <200105010535.NAA09901@www8.kaxiu.com>
```

To: fancyrainbow@263.net
Subject: 您的朋友 新蓝 给您寄来贺卡
From: lz.lan@163.net
Reply-To: lz.lan@163.net
Sender: nobody@www8.kaxiu.com

C: QUIT
S: +OK core mail

在上面的例子中，C 代表 Client，S 代表 Server，通过察看这些信息，可以帮助我们了解客户端是如何与服务端交互的，也可以帮助我们更好地掌握各种邮件协议。

如果方法参数为 false，那么会话过程将不处于 debug 状态，这些交互信息自然也不会出现在控制台中。

public void setProvider(Provider provider) throws NoSuchProviderException; 设定当前会话过程的 Provider，读者请参考 getProvider()方法和 getProviders()方法的说明。

12.2.2 Store 类

Store 类映射了 Mail Server 上面的邮箱系统，利用 Store 类，我们就可以访问用户邮箱中的文件夹，进而访问邮件的信息了。Store 类继承自 Service 类，其中定义的重要方法如下所示：

public abstract Folder getFolder(String name) throws MessagingException; 获取邮箱中的文件夹，例如 POP3(Post Office Protocol - Version 3)邮箱的 INBOX 文件夹，根据 Java Mail API 的说明，POP3 邮箱只支持 INBOX 收件夹。参数 name 指定了收件夹的名字，该方法的返回值是 Folder 对象。代码示例如下。

例：

```
<%
Properties props = System.getProperties();
Session sess= Session.getDefaultInstance(props，null);
sess.setDebug(true);
Store store = null;
store = sess.getStore(protocol);
store.connect(host，user，password);
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
%>
```

读者需要注意的是，在调用此方法以前，Store 对象必须首先调用 connect()方法和邮件服务器建立连接，否则会出现错误。connect()方法是在 Service 类中定义的，Store 类继承了这个方法，connect()方法有好几个版本，如下所示：

```
public void connect() throws MessagingException;
public void connect(java.lang.String host，java.lang.String user，java.lang.String password)
    throws MessagingException;
public void connect(java.lang.String host，int port，java.lang.String user，
```

java.lang.String password) throws MessagingException;

读者可以根据实际情况自行选用，在上面的代码片断中，我们选用了第二个 connect()方法，三个参数的意义分别为：主机名、用户名、密码。

public abstract Folder getFolder(URLName url) throws MessagingException; 同上，不过这两个方法的参数略有不同，前一个方法指定的是文件夹的名字，后一个方法指定的却是代表该文件夹的 URLName 对象。有些读者可能要问 URLName 对象究竟是什么样子呢?我们可以使用下面的 JSP 代码段来返回 URLName 对象的信息。

例：

```
<%
out.println(store.getURLName().toString()+"<br>");
out.println(store.getURLName().getUsername()+"<br>");
out.println(store.getURLName().getPassword()+"<br>");
out.println(store.getURLName().getPort()+"<br>");
out.println(store.getURLName().getProtocol()+"<br>");
%>
```

运行结果如下所示：

```
pop3：//fancyrainbow@263.net
fancyrainbow
null
-1
pop3
```

在上面的代码中，getURLName()方法是 Service 类定义的方法，getUsername()、getPort()、getPassword()、getProtocol()等方法是 URLName 类定义的方法。

public void addStoreListener(StoreListener l)； 该方法的作用是添加一个邮箱事件监听者，当某个邮箱事件发生了，例如收到新邮件了，某个邮件被阅读了，那么邮箱事件监听者会对这些事件进行反应。

关于事件处理的知识，读者可以参考本书第一章关于 JavaBeans 事件模型的叙述。我们认为如果客户端与服务端保持着长时间的连接，这时候采用事件监听者才有用武之地，而 JSP 程序只是在执行的时候才与 Mail 服务器建立连接，当数据被发送到客户端时，已经和 Mail Server 断开了连接，会话过程已经结束，这段时间是很短的，使用事件监听者根本没有意义，因此我们建议读者在编写 JSP 程序中，最好不要建立事件监听模型。

public void removeStoreListener(StoreListener l); 这个方法和 addStoreListener()方法的作用相反，是移除某个邮箱事件监听者。

protected void notifyStoreListeners(int type，String message); 这个方法的作用是通知所有的当前邮箱的邮箱事件监听者，某个事件发生了，并把消息发给它们，让它们做出响应。

public void addFolderListener(FolderListener l); 这个方法和 addStoreListener()方法的作用类似，不过前者的作用是添加邮箱事件监听者，而后者的作用是添加文件夹事件监听者。

public void removeFolderListener(FolderListener l); 这个方法和 removeStoreListener()

方法的作用类似，不过前者的作用是移除邮箱事件监听者，而后者的作用是移除文件夹事件监听者。

protected void notifyFolderListeners(int type， Folder folder); 这个方法和notifyStoreListener()方法的作用类似，不过前者的作用是通知邮箱事件监听者，而后者的作用是通知文件夹事件监听者。

protected void notifyFolderRenamedListeners(Folder oldF，Folder newF); 这个方法用于通知文件夹事件监听者，某个文件夹发生了文件夹改名的事件，让它们做出响应。这个方法有两个参数，前者是改名前的文件夹对象 oldF，后者是改名后的文件夹对象 newF。

12.2.3 Transport 类

Transport 类主要用于承担发送邮件的服务。它定义的重要方法如下所列：

public static void send(Message msg) throws MessagingException; 该方法将发送邮件，邮件的信息，如主题，收信人地址，发信人地址等都被封装到 Message 对象中去了。

下面是一段发送邮件的 JSP 代码，适用于不需要身份验证的邮件服务器。

例：

```
<%
String to="Rainbow@Rainbow.pku.edu.cn";
String from="fancyrainbow@263.net";
String subject="Test";
Properties props = System.getProperties();
props.put("mail.smtp.auth"，"false");
props.put("mail.smtp.host"，"Rainbow.pku.edu.cn");
Session sess = Session.getInstance(props，null);
sess.setDebug(true);
Message msg = new MimeMessage(sess);
msg.setFrom(new InternetAddress(from));
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to，false));
msg.setSubject(subject);
msg.setSentDate(new Date());
msg.setText("Hello World");
Transport.send(msg);
%>
```

因为 send()方法的类型是 static，所以我们不需要构造 Transport 类的实例对象，便能够直接调用 send()方法。读者可以把这一段代码与第 11 章利用 SmtpClient 类发送邮件的代码相比较，自然是这里的代码较为复杂，不过功能也更为强大些。

关于 Message 类的知识，我们还会在下文提及。

public static void send(Message msg，Address[] addresses)throws MessagingException; 这个方法和上文的同名方法的作用类似，不过利用这个方法可以把一封邮件发送到多个收信人的信箱中。Address 类型的数组保存了邮件的多个投递目标地址。

public abstract void sendMessage(Message msg ， Address[] addresses)throws

MessagingException;　这个方法和上面的方法类似，不过上文的两个 send()方法都是 static 类型的方法(静态方法)，因此可以直接以 Transport.send(param)的形式调用，但是这样做有一点不妥，就是我们无法监听邮件发送过程中可能发生的事件，比如邮件无法投递等事件。使用 sendMessage()方法就不同了，我们必须首先创建 Transport 对象，然后才能调用 sendMessage()方法，不过这个时候可以添加 Transport 事件监听对象，监听投递邮件过程中所产生的事件。sendMessage()方法也可以按照一个地址列表发送邮件。

public void addTransportListener(TransportListener l);　这个方法的作用是添加 Transport 事件监听对象，这个 Transport 事件监听对象可以监听发送邮件过程中可能出现的事件，并对其做出响应。

public void removeTransportListener(TransportListener l);　这个方法和 addTRansportListener()方法刚好相反，它的作用是移除 Transport 事件监听对象。

protected void notifyTransportListeners(int type , Address[] validSent , Address[] validUnsent，Address[] invalid，Message msg);　这个方法的作用是通知 Transport 事件监听对象特定的事件已经发生了，让它们做出响应。这个方法的参数很特殊，有三个是 Address 数组类型的参数，它们的意义如下：

validSent：地址正确，已经发送了邮件的地址列表。

validUnSent：地址正确，但是还没有发送邮件的地址列表。

invalid：地址不正确，不能够发送邮件的地址列表。

这个方法被声明为 protected，类型，一般由系统自动调用，程序员对此不必关心。

12.2.4　Folder 类

Folder 类也是十分重要的类，它映射了邮箱中的文件夹系统，例如 POP3 邮箱中的 INBOX 文件夹。Folder 可以包含 Message，亦即邮件，也可以包含 Folder，也可以被 Folder 包含，这样就形成了一个树状结构。利用 Folder 类，可以处理上下级的文件夹，也可以处理邮件。Folder 类尤以处理邮件的功能特别强大。如何获取 Folder 对象呢?最常用的方法是调用 Store 类的 getFolder()方法，读者可以参考关于 Store 类的介绍，我们在这里就不再重复了。

在 Folder 类中定义的重要方法如下所列：

public abstract String getName();　获取文件夹的名字。

public abstract String getFullName();　获取文件夹的全名。

public URLName getURLName();　获取关联当前文件夹的 URLName 对象，请看下面的 JSP 代码段。

例：

```
<%
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
out.println(folder.getName()+"<br>");
out.println(folder.getFullName()+"<br>");
out.println(folder.getURLName().toString()+"<br>");
```

```
%>
```

我们所使用的文件夹为 INBOX，上面这段 JSP 代码的返回值为：

```
INBOX
INBOX
pop3：//fancyrainbow@263.net/INBOX
```

public Store getStore();　获取当前文件夹所属的 Store 对象，我们可以由 Store 对象获取 Folder 对象，又可以反过来由 Store 对象获取 Folder 对象。

public abstract Folder getParent() throws MessagingException;　该方法可以获取树状结构中上一级的父文件夹。

public abstract boolean exists() throws MessagingException;　该方法可以判断某个文件夹是否存在。请看下面的 JSP 代码段。

例：

```
<%
Folder folder = store.getFolder("INBOXX");
out.println(folder.exists());
%>
```

运行结果为 false。

public abstract Folder[] list(String pattern) throws MessagingException;　该方法可以返回所有属于当前文件夹的子文件夹，方法的返回值是一个 Folder 数组，参数为匹配模式。

假设存在着这样的一个树状目录结构：

```
Personal/
        Finance/
                Stocks
                Bonus
                StockOptions
        Jokes
```

那么有：

（1）list(“*”)方法所返回的 Folder 数组将包含 Personal、Finance、Stocks、Bonus、StockOptions、Jokes 等 Folder 对象。

（2）list(“%”)方法将返回 Fiance、Jokes 这两个 Folder 对象。

（3）list(“Jokes”)方法将返回的 Folder 数组中仅仅包含 Jokes 这个 Folder 对象。

（4）list(“Stock*”)方法返回的 Folder 数组中包含 Stocks、StockOptions 这两个 Folder 对象。

public Folder[] listSubscribed(String pattern) throws MessagingException;　这个方法和前一个方法基本类似，但是列出来的文件夹都是被预定的文件夹。参数 pattern 的意义同上。

public Folder[] list() throws MessagingException;　利用这个方法可以列出当前 Folder 的所有子文件夹，相当于调用 list(“*”)的情况。这个方法可以在当前文件夹处于关闭状态下

调用。

public Folder[] listSubscribed() throws MessagingException;　利用这个方法可以列出当前 Folder 的所有被预定的子文件夹，相当于调用 listSubscribed (“*”)的情况。这个方法可以在当前文件夹处于关闭状态下调用。

public abstract int getType() throws MessagingException;　返回当前文件夹的类型，返回值是一个整型常数，可以取下面的值：

HOLDS_FOLDERS：此文件夹可以包含子文件夹。

HOLDS_MESSAGES：此文件夹可以保存邮件。

上述两个常数在 Folder 类中被定义。

public abstract boolean create(int type) throws MessagingException;　创建一个新文件夹。似乎 POP3 类型的邮箱不允许创建新的文件夹。参数 type 是文件夹的类型，也就是上文所说的两种类型。该方法的返回值为布尔型，成功则返回 true，否则返回 false。

public boolean isSubscribed();　判断当前文件夹是否被预定了。不是所有的 Store 对象都支持预定。

public void setSubscribed(boolean subscribe) throws MessagingException;　设定当前的文件夹是否被预定。不是所有的 Store 对象都支持预定，这个方法可以在文件夹处于关闭状态下调用。

public abstract boolean hasNewMessages() throws MessagingException;　判断当前文件夹是否有了新邮件了。这个方法的返回值不一定正确。

public abstract Folder getFolder(String name) throws MessagingException;　获取文件夹对象，参数为文件夹的名字。返回值为 Folder 对象。

public abstract boolean delete(boolean recurse) throws MessagingException;　这个方法删除当前文件夹，只有当文件夹处于关闭状态时才能够调用这个方法。参数 recurse、决定了 delete 操作对子文件夹的影响，如果 recure 的值为 true，那么所有的子文件都要被删除，当前文件夹自然会被删除。如果 recure 的值为 false，那么有三种大的情况：

第一种情况：Folder 的类型为 HOLDS_MESSAGES，那么该文件夹包括其中所有的邮件都会被删除。

第二种情况：Folder 的类型为 HOLDS_FOLDERS，但是这个 Folder 是空的，那么它被删除，如果它含有其他的文件夹，那么 delete()方法执行失败，没有一个文件夹被删除。

第三种情况：Folder 的类型既为 HOLDS_FOLDERS 也是 HOLDS_MESSAGES，如果 Folder 是空的，那么它被删除，如果 Folder 含有 Message，但是不含 Folder，那么 Message 被删除，Folder 本身也被删除，如果 Folder 既含有 Message，也含有 Folder，那么情况就更为复杂了，我们就不在这里深入讨论了。读者可以参考 Java Mail API 的文档。

public abstract boolean renameTo(Folder f) throws MessagingException;　这个方法可以重命名当前的文件夹。这个方法可以在文件夹处于关闭状态下进行。

public abstract void open(int mode) throws MessagingException;　打开当前文件夹，使得它处于 Open 状态，文件夹只有处于 Open 状态，才能够从中读取邮件信息。参数 mode 为打开文件夹的模式，有两种可能的取值：

READ_ONLY：此文件夹是只读的，不能够执行删除邮件、转移邮件等操作。

READ_WRITE：此文件夹是读写的，能够执行删除邮件、转移邮件等操作。

这两个都是整型常数，在 Folder 类中有定义。

public abstract void close(boolean expunge) throws MessagingException; 关闭当前的文件夹。有些操作必须在文件夹处于关闭状态下进行，例如删除文件夹。如果参数 expunge 的值为 true，那么在关闭文件夹的同时将删除所有已经打上删除标记的邮件。

public abstract boolean isOpen(); 这个方法可以判断当前文件夹是否处于 Open 状态。

public int getMode(); 这个方法可以判断当前文件夹的打开模式是 READ_ONLY 或者是 READ_WRITE，读者请看下面的 JSP 代码段。

例：

```
<%
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
out.println(folder.isSubscribed()+"<br>");
out.println(folder.isOpen()+"<br>");
out.println(folder.getMode()+"<br>");
%>
```

运行结果如下所示：

```
true
true
1
```

public abstract int getMessageCount() throws MessagingException; 该方法将返回当前文件夹中邮件的数目。如果在文件夹处于 Close 状态时刻调用这个方法，那么将返回-1。

public int getNewMessageCount() throws MessagingException; 该方法将返回当前文件夹中新邮件的数目。如果在文件夹处于 Close 状态时刻调用这个方法，那么将返回-1。

public int getUnreadMessageCount() throws MessagingException; 该方法将返回当前文件夹中没有阅读过的邮件的数目。如果在文件夹处于 Close 状态时刻调用这个方法，那么将返回-1。

public abstract Message getMessage(int msgnum) throws MessagingException; 这个方法将获取特定编号的邮件，参数 msgnum 代表该邮件在当前文件夹所有邮件列表中的位置。

public Message[] getMessages(int start，int end) throws MessagingException; 这个方法将返回一个 Message 数组，包含了特定编号范围内的所有邮件。

public Message[] getMessages(int[] msgnums) throws MessagingException; 这个方法将返回一个 Message 数组，参数 msgnums 是一个整型数组，里面存储了所需要提取的邮件在当前文件夹邮件列表中的编号。

public Message[] getMessages() throws MessagingException; 这个方法将返回一个 Message 数组，此 Message 数组包含了当前文件夹中所有的邮件。下面的代码段演示了如何获取所有的邮件。

例：

```
<%
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
for (int i=0，n=message.length; i<n; i++)
{
    out.println(i+1+"："+message[i].getFrom()[0]+"--->"+message[i].getSubject()+"<br>");
}
%>
```

public void fetch(Message[] msgs，FetchProfile fp) throws MessagingException; 这个方法的作用是根据预定义的 FetchProfile 对象来 Prefetch 一个邮件数组，请看下面的代码。

例：

```
<%
Message msg[] = folder.getMessages();
FetchProfile fp = new FetchProfile();
fp.add(FetchProfile.Item.ENVELOPE);
fp.add("X-mail");
folder.fetch(msg，fp);

for (int i = 0; i < folder.getMessageCount(); i++)
{
        out.println(msg[i].getFrom());
        out.println(msg[i].getSubject());
        out.println(msg[i].getHeader("X-mailer"));
}
%>
```

执行结果大致如下：

```
[Ljavax.mail.internet.InternetAddress;@301b07c 您的朋友 sonia 给您寄来贺卡 null
[Ljavax.mail.internet.InternetAddress;@239b07c 您的朋友 新蓝 给您寄来贺卡 null
[Ljavax.mail.internet.InternetAddress;@2b1b07c 您的朋友 sonia 给您寄来贺卡 null
[Ljavax.mail.internet.InternetAddress;@d55b07c 五一快乐 null
```

public void setFlags(Message[] msgs ， Flags flag ， boolean value) throws MessagingException; 该方法的作用是给一个邮件数组里面所有的邮件都打上一样的标记。参数 msgs 就是需要打上标记的邮件数组，flag 是一个 Flags 对象，代表邮件标记，value 是一个布尔值，如果为 true，那么数组里面所有的邮件都打这个标记，如果为 false，那么邮件数组里面所有的邮件都不打上这个标记。

public void setFlags(int start ， int end ， Flags flag ， boolean value) throws MessagingException; 这个方法同样是给邮件打上标记，不过需要打上标记的邮件是从编号 start 开始，编号 end 结束。

public void setFlags(int[] msgnums ， Flags flag ， boolean value) throws MessagingException; 这个方法同样是给邮件打上标记，不过需要打上标记的邮件的编号

存储在整型数组 msgnums 中。

public void copyMessages(Message[] msgs，Folder folder) throws MessagingException; 这个方法的作用是把一个邮件数组中所包含的邮件拷贝到另一个文件夹中。

public abstract Message[] expunge() throws MessagingException; 这个方法将删除所有已经打上删除标记的邮件，返回值是被删除的邮件的数组。

public Message[] search(SearchTerm term) throws MessagingException; 这个方法的作用是查找所有符合要求的邮件。查询范围是当前的文件夹，查询条件由 SearchTerm 对象 term 决定。

public Message[] search(SearchTerm term , Message[] msgs) throws MessagingException; 这个方法的作用是查找所有符合要求的邮件。查询范围是邮件数组 msgs，邮件数组中的邮件必须属于当前文件夹，查询条件由 SearchTerm 对象 term 决定。

public void addFolderListener(FolderListener l); 添加一个文件夹事件监听对象——FolderListener，FolderListener 可以监听文件夹中发生的事件，并且响应它们。

public void removeFolderListener(FolderListener l); 和 addFolderListener()方法相对，删除某个文件夹事件监听对象。

protected void notifyFolderListeners(int type); 这个方法可以用广播的方式通知与当前文件夹绑定在一起的 FolderListener：某种类型的事件发生了，让它们做出响应。参数 type 是一些预定义的常数，在 FolderEvent 类中定义，可以取的值如下所示：

FolderEvent.CREATED：文件夹被创建。

FolderEvent.DELETED：文件夹被删除。

FolderEvent.RENAMED：文件夹被改名。

这个方法一般被系统自行调用。

protected void notifyFolderRenamedListeners(Folder folder); 这个方法通知文件夹事件监听者，某个文件夹发生了重命名的事件。

12.2.5　Message 类

Message 类映射了邮件的结构，利用 Message 类，我们可以读取邮件的详细信息，也可以设定邮件的状态，还可以对邮件进行各种各样的操作。Message 类实现了 Part 接口的方法。

public abstract Address[] getFrom() throws MessagingException; 这个方法返回一个 Address 数组，该数组包含了当前邮件的 Form 属性的值，也就是发信人的邮箱地址列表。

public abstract void setFrom() throws MessagingException; 这个方法不设定邮件的发信人地址。这样收信人在打开邮件的时候，将看不到发信人的地址。

public abstract void setFrom(Address address) throws MessagingException; 这个方法可以设定当前邮件邮件的发信人地址。这样收信人在打开邮件的时候，将看到发信人的地址。setFrom()方法一般用在发送邮件的程序中，一般在发送邮件以前被调用。请看下面的代码段。

例：

<%

```
String from="fancyrainbow@263.net";
Message msg = new MimeMessage(sess);
msg.setFrom(new InternetAddress(from));
%>
```

public abstract void addFrom(Address[] addresses) throws MessagingException; 这个方法可以往当前邮件的发信人邮箱地址列表添加更多的发信人邮箱地址，一般在发送邮件以前被调用，方法参数是一个 Address 数组，包含了需要添加的发信人的邮箱地址。

public Address[] getAllRecipients() throws MessagingException; 获取当前邮件的投递目标邮箱地址列表，返回值是一个 Address 数组。

public abstract void setRecipients(Message.RecipientType type，Address[] addresses) throws MessagingException; 我们都知道，邮件的投递目标邮箱地址有多种类型，有直接投递的邮箱地址，还有抄送的邮箱地址等等，在 Java Mail API 中，规定了邮件的投递目标邮箱地址由三种类型，由三个常量表示，分别是：

Message.RecipientType.BCC：指 blind carbon copy 类型的目标邮箱地址。

Message.RecipientType.CC：指 carbon copy 类型的目标邮箱地址，相当抄送的邮箱地址。

Message.RecipientType.TO：指的是直接投递的目标邮箱地址。

上述三个常量在 Message.RecipientType 类中定义，它们都是 Message.RecipientType 类型的常量。这个 setRecipients()方法就用来设定一个邮箱数组中所有邮箱地址的投递类型，是 BCC 或者是 CC 还是 TO。该方法的第一个参数就是投递目标邮箱地址的类型，第二个参数是一个 Address 数组，里面包含了需要设置邮箱地址类型的邮箱地址。如果我们获得了一个邮件，我们想知道这个邮件的所有目标地址中投递地址类型为 CC 的邮箱地址都有哪些; 投递地址类型为 BCC 的邮箱地址都有哪些;投递地址类型为 TO 的邮箱地址都有哪些，那么我们应该如何做呢?下面的代码段解决了这个问题。

例：

```
<%
Message msg = folder.getMessages(1);
Address[] a = m.getRecipients(Message.RecipientType.TO);
Address[] b = m.getRecipients(Message.RecipientType.BCC);
Address[] c = m.getRecipients(Message.RecipientType.CC);
…
%>
```

getRecipients(Message.RecipientType type)方法和 getRecipients()方法一样，也是在 Message 类中定义的。

public void setRecipient(Message.RecipientType type， Address address) throws MessagingException; 这个方法和上文的同名方法类似，不过需要设置投递地址类型的邮箱地址只有一个(一个 Address 对象)，而不是一个邮箱数组。

public abstract void addRecipients(Message.RecipientType type，Address[] addresses) throws MessagingException; 这个方法可以往当前邮件的投递目标地址列表中添加一系列设定了投递地址类型的邮箱地址。这一系列邮箱地址以 Address 数组的形式提交，而且

它们的投递地址类型由参数 type 指定，type 是一个 Message.RecipientType 类型的常量，可以取的值为：

Message.RecipientType.BCC

Message.RecipientType.CC

Message.RecipientType.TO

关于这三个常量，我们已经在上文提到过，这里我们就不再重复了。

这个方法一般只在发送邮件以前调用。

public void addRecipient(Message.RecipientType type， Address address) throws MessagingException; 这个方法和上面的同名方法类似，不过本方法每次只添加一个设定了投递地址类型的目标邮箱地址，而上面的方法一次就添加一个数组的邮箱地址。这个方法一般只在发送邮件以前调用。

public Address[] getReplyTo() throws MessagingException; 获取回复邮件时需要回复的邮箱地址的列表。返回值是一个 Address 数组。

public void setReplyTo(Address[] addresses) throws MessagingException; 设定回复当前邮件时需要回复的邮箱地址的列表，方法参数是一个 Address 数组，里面包含了需要回复的邮箱地址的列表。setReplyTo()方法和 getReplyTo()方法对应，前者一般在发送邮件以前指定。

public abstract String getSubject() throws MessagingException; 获取当前邮件的主题。邮件的主题在发送邮件以前由 setSubject()方法指定。

public abstract void setSubject(String subject) throws MessagingException; 设定邮件的主题，该方法一般只在发送邮件以前调用。

public abstract java.util.Date getSentDate() throws MessagingException; 获取发送此邮件的时间。请看下面的 JSP 代码段。

例：

```
<%
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
for (int i=0，n=message.length; i<n; i++)
{
    out.println(i+1+"："+message[i].getFrom()[0]+"--->"+message[i].getSubject()+"---->");
    out.println(message[i].getSentDate()+"<br>");
}
%>
```

运行效果如下所示：

```
1：hljudge@263.net--->稿子怎么样了----> Wed May 02 18：21：27 CST 2001
2：fancyrainbow --->sdfsdf----> Sat May 05 21：24：49 CST 2001
3：fancyrainbow --->sdfsdfsd----> Sat May 05 21：26：45 CST 2001
```

public abstract void setSentDate(java.util.Date date) throws MessagingException; 设定发送此邮件的时间，并非是设定什么时间，这封邮件就什么时候发送出去，所谓的发送

时间只是邮件信息头的一部分，没有什么实际意义，当 Transport 对象的 send()方法被调用时，那才是邮件的发送时间，对于这一点，读者千万不要误解。

public abstract java.util.Date getReceivedDate() throws MessagingException; 这个方法获取收到此信件的时间。

public abstract Flags getFlags() throws MessagingException; 获取邮件被打上的标记，我们可以给邮件打上各种各样的标记，以便标识邮件的状态，如新邮件、已经阅读过的邮件、已经回复的邮件、将要被删除的邮件等等。该方法的返回值是 Flags 对象，利用这个对象，我们可以判断此邮件所处的状态。请看下面的代码。

例：

```
<%
out.println(msg.getFlags().toString());
%>
```

输出的结果类似于：

```
javax.mail.Flags@0。
```

public boolean isSet(Flags.Flag flag) throws MessagingException; 这个方法可以判断某个邮件是不是被打上什么标记。请看下面的代码段。

例：

```
<%
Message m = folder.getMessage(1);
m.setFlag(Flags.Flag.DELETED，true); // set the DELETED flag
// Check if DELETED flag is set of this message
if (m.isSet(Flags.Flag.DELETED))
    out.println("DELETED message");
%>
```

上面的代码的输出结果自然是 DELETED message 了。

public abstract void setFlags(Flags flag，boolean set) throws MessagingException; 这个方法的作用是给邮件打上某种标记或者去掉某种标记，这依赖于 set 参数的值，如果 set 的值为 true，那么就是打上某个标记，如果 set 的值为 false，那么就是去掉某个标记。

public void setFlag(Flags.Flag flag，boolean set) throws MessagingException; 这个方法也是给邮件打上某个标记，和上面的同名方法类似，不过这个方法使用 Flags.Flag 对象打标记，而上面的方法使用 Flags 对象打标记。请看下面的 JSP 代码段。

例：

```
<%
Message message[] = folder.getMessages();
message[6].setFlag(Flags.Flag.ANSWERED，true);
message[5].setFlag(Flags.Flag.DELETED，true);
message[4].setFlag(Flags.Flag.DRAFT，true);
message[3].setFlag(Flags.Flag.FLAGGED，true);
message[2].setFlag(Flags.Flag.RECENT，true);
message[1].setFlag(Flags.Flag.SEEN，true);
message[0].setFlag(Flags.Flag.USER，true);
```

```
for (int i=0，n=message.length; i<n; i++)
{
    out.println(i+1+"："+message[i].getSubject()+"---->");
    out.println(message[i].getFlags().toString()+"<br>");
}
%>
```

运行结果如下所示：

1：稿子怎么样了----> javax.mail.Flags@80000000
2：sdfsdf----> javax.mail.Flags@20
3：sdfsdfsd----> javax.mail.Flags@10
4：sdfsdf----> javax.mail.Flags@8
5：sdfsdf----> javax.mail.Flags@4
6：您的朋友 新蓝 给您寄来贺卡----> javax.mail.Flags@2
7：您的朋友 新蓝 给您寄来贺卡----> javax.mail.Flags@1
8：您的朋友 sonia 给您寄来贺卡----> javax.mail.Flags@0

public int getMessageNumber(); 利用这个方法获得当前邮件在文件夹邮件列表中的编号。

protected void setMessageNumber(int msgnum); 利用这个方法设定当前邮件在文件夹邮件列表中的编号。

public Folder getFolder(); 这个方法可以获得当前邮件所处的文件夹，返回值是一个 Folder 对象。

public boolean isExpunged(); 判断邮件是否给打上了删除标记。

protected void setExpunged(boolean expunged); 设定当前邮件被打上删除标记，或者是不被打上删除标记，采取何种操作，取决于 expunged 参数的值，如果邮件被打上了删除标记，那么在调用 Folder 对象的 expunge()方法时，它将会被立刻删除。

public abstract Message reply(boolean replyToAll) throws MessagingException; 这个方法的作用是：根据源邮件的信息，自动产生一个回复邮件(Message 对象)，这个回复邮件的各项属性都设定好了，如目标地址，发信人邮箱地址等，回复邮件的主题也被设定为在源主题的前面加上“Re:”字样，我们可以用 setXXX()方法对这些属性进行微调，再调用 Transport.send()方法，就可以实现回信的操作了。参数 replyToAll 指定是否回复源邮件的所有发信人。这是一个抽象方法，也就是该方法没有实现。必须在其子类中实现。

public abstract void saveChanges() throws Messaging Exception; 在关闭文件夹以前调用这个方法，保存对文件夹中的邮件所作的改变，例如改变了邮件的标题或者是给邮件打上了某个标记等等。请注意，文件夹的打开方式必须设定为 READ_WRITE，否则这个方法无效。这是一个抽象方法，也就是该方法没有实现。必须在其子类中实现。

public boolean match(SearchTerm term) throws MessagingException; 这个方法判断当前的邮件是否匹配查询条件，至于查询条件，则被参数 term 封装起来了，term 是 SearchTerm 类的实例对象。

Message 类还实现了很多在 Part 接口中定义的方法，为了避免重复，我们会在介绍 Part

接口时着重介绍这些方法。

12.2.6　Part 接口

Part 接口映射了邮件的信息结构，Part 接口中声明的方法在 Message 类中实现，也就是说，当我们创建一个 Message 对象时，也能够使用在 Part 接口中声明的方法。Part 接口中声明的重要方法如下所示。

public int getSize() throws MessagingException;　获取邮件的大小，返回值是字节数。

public int getLineCount() throws MessagingException;　返回邮件的行数，如果没有办法获取这个数值，那么返回-1。

public String getContentType() throws MessagingException;　返回邮件文本的类型。一般返回值是 text/plain。

public boolean isMimeType(String mimeType) throws MessagingException;　判断邮件是否所指定的 Mime Type。

public String getDisposition() throws MessagingException;　这个方法将返回邮件的 Disposition 信息，Disposition 信息描述了邮件是以何种方式送到收信人手里的，请参考 RFC 2183 文件。

public void setDisposition(String disposition) hrows MessagingException;　这个方法用于设定邮件的 Disposition 信息，是通过附件的方式发送，还是以邮件正文的形式发送。参数可能取的值为 Part.ATTACHMENT 或者 Part.INLINE，前者指定邮件以附件的形式发送，后者指定邮件以邮件正文的方式发送。

public String getFileName() throws MessagingException;　如果当前的 Part 对象是一个附件的话，那么此方法可以获取附件所对应的文件名。

public void setFileName(String filename) throws MessagingException;　此方法可以指定邮件的附件的文件名。

public InputStream getInputStream() throws IOException，MessagingException;　此方法返回一个输入流对象，利用这个输入流对象，可以流式地从邮件中读入数据，包括邮件信息头和邮件正文等。

public Object getContent() throws IOException，MessagingException;　利用这个方法可以返回邮件的正文。

public void setContent(Object obj，String type) throws MessagingException;　这个方法的作用是把数据写入邮件的正文，第一个参数就是需要写入到邮件正文的数据，第二个参数指定邮件的类型，是普通的文本邮件，或者是 HTML 格式的邮件，还是其他类型的邮件。

public void setText(String text) throws MessagingException;　这个方法的作用是把一些文本数据写入到邮件的正文中，并把邮件的类型设为 text/plain。

public void setContent(Multipart mp) throws MessagingException;　这个方法同样是设定邮件的正文，不过这次不是什么文本数据了，而是一个 Multipart 对象。关于 Multipart 类的知识，我们将在下文介绍。

public void writeTo(OutputStream os) throws IOException，MessagingException;　指

定一个输出流，利用这个输出流对象，往邮件正文中写入数据，这个方法如何使用呢?首先是创建一个 Message 对象，然后创建一个输出流对象，接着调用 writeTo()方法，把这个输出流对象与 Message 对象关联在一起，利用输出流对象的 write()方法，就可以往邮件正文中写入数据了。

public String[] getHeader(String header_name) throws MessagingException; 这个方法的作用是获取邮件的投递信息头，邮件的投递信息头类似于 HTTP Header。一个 邮件投递信息头有多个值，所以此方法的返回值为字符串数组。

public void setHeader(String header_name , String header_value) throws MessagingException; 这个方法的作用是设置某个邮件投递信息头的值，如果原来已经赋值，那么原值会被覆盖掉。

public void addHeader(String header_name , String header_value) throws MessagingException; 添加新的邮件投递信息头。

public void removeHeader(String header_name) throws MessagingException 删除某个邮件投递信息头。

public Enumeration getAllHeaders() throws MessagingException; 获取所有邮件投递信息头的列表，返回值是一个枚举对象。这个方法如何使用呢?请看下面的代码片断。

例：

```
<%
Enumeration enum=msg.getAllHeaders();
while(enum.hasMoreElements())
{
    out.println(enum.nextElement());
}
%>
```

上面代码段的作用是把所有的邮件投递信息头的名称一一输出。

12.2.7 Multipart 类

Multipart 类用于构造一个复合类型的邮件主体(Multiple Body Part)。它的用法如下，首先构造若干个 BodyPart 对象，再构造 Multipart 对象，利用 Multipart 对象的 addBodyPart()方法把 BodyPart 对象添加到 Multipart 对象中，最后用 Part 接口声明的 setContent()方法把 Multipart 对象设定为邮件的主体，亦即正文。下面我们就来看看在 Multipart 类中定义的重要方法。

public String getContentType(); 返回 Multipart 对象的 Content-Type 属性值，可能的值为 multipart/alternative 、 multipart/mixed 、 multipart/related 、 multipart/parallel 、 multipart/signed。

public int getCount() throws MessagingException; 获取 Multipart 对象中所包含的 BodyPart 对象的数目。

public BodyPart getBodyPart(int index) throws MessagingException; 返回 Multipart 对象中所包含的特定编号的 BodyPart 对象。BodyPart 对象都是按顺序包含在 Multipart 对象

中，每一个 BodyPart 对象都有它特定的编号。

public boolean removeBodyPart(BodyPart part) throws MessagingException; 删除 Multipart 对象中所包含的特定的 BodyPart 对象，返回值是一个布尔值。

public void removeBodyPart(int index) throws MessagingException; 删除 Multipart 对象中所包含的特定编号的 BodyPart 对象，没有返回值。

public void addBodyPart(BodyPart part) throws MessagingException; 往 Multipart 对象中添加 BodyPart 对象。

public void addBodyPart(BodyPart part，int index) throws MessagingException; 往 Multipart 对象中插入 BodyPart 对象，插入位置由第二个参数指定。

public Part getParent(); getParent()方法可以获取包含此 Multipart 对象的 Part 对象(或者 Message 对象、或者 BodyPart 对象)。如果此 Multipart 对象还没有被设定为邮件的主体，那么此方法的返回值为 NULL。

public void setParent(Part parent); 利用这个方法同样可以把当前的 Multipart 对象设定为邮件的主体。它的作用和 Part 接口中声明的 setContent()方法一样。

12.2.8　Flags 类

Flags 类封装了给邮件所打的标记，利用 Flags 类和 Flags.Flag 类，我们可以把邮件分成几种类型，新收到的邮件、没有阅读过的邮件、已经阅读过的邮件、将要被删除的邮件等等，同时，我们也可以按照邮件的类型，把它们加以分类，把 INBOX 文件夹分为逻辑上的几个部分，例如收件箱、发件箱、垃圾箱、存信箱等等。这样一来十分便于对邮件和文件夹的管理。

使用 Message 类的 getFlags()方法可以获取 Flags 对象。还可以利用 Message 类的 setFlags()方法给邮件打上标记。Flags 类的作用主要是处理标记自身的结构，并不是用于操作标记。Flags 类定义的重要方法如下所示：

public void add(Flags.Flag flag); 这个方法可以在当前标记的基础上加上新的标记，比如一封邮件既可以是刚刚收到的，也可以同时是被打上删除标记的邮件。

public void add(String flag); 作用同上，参数不同。这个方法主要用于添加自定义的标记。

public void add(Flags f); 作用同上，参数不同，这个方法以 Flags 对象作为参数，前者以字符串为参数。

public void remove(Flags.Flag flag); 这个方法可以在当前复合标记的基础上去掉某些标记。

public void remove(String flag); 作用同上，参数不同。这个方法主要用于删除自定义的标记。

public void remove(Flags f); 作用同上，参数不同。

public boolean contains(Flags.Flag flag); 这个方法的作用是判断当前的复合标记中有没有包含指定的标记。

public boolean contains(String flag); 作用同上，参数不同。这个方法对于自定义的标记有效。

public boolean contains(Flags f);　作用同上，参数不同。

public Flags.Flag[] getSystemFlags();　这个方法可以获取邮件系统所支持的标记类型，返回值是一个 Flags.Flag 数组。

public String[] getUserFlags();　这个方法获取用户自定义的标记，返回值是一个字符串数组。

下面是 Flags.Flag 类所定义的一些常量，它们是标记的抽象表示。

public static final Flags.Flag ANSWERED;　回复标记，表明该邮件已经被回复。

public static final Flags.Flag DELETED;　删除标记，该邮件即将被删除。

public static final Flags.Flag DRAFT;　草稿标记，该邮件处于草稿状态，还没有发出。

public static final Flags.Flag FLAGGED;　该邮件处于 FLAGGED 状态。

public static final Flags.Flag RECENT;　新邮件标记，最新收到的邮件。

public static final Flags.Flag SEEN;　已经被阅读过的邮件。

public static final Flags.Flag USER;　属于用户自定义标记。

12.3　javax.mail.internet 包

在上一节中我们已经介绍了 javax.mail 包中若干个重要的类和接口的用法，在这一节中，我们将接着介绍 javax.mail.internet 包的知识。javax.mail 包的作用是模拟一个邮件系统，并提供收发邮件的支持，而 javax.mail.internet 包的侧重点则在于对 Internet 邮件系统的支持，但是没有提供收发邮件的功能。javax.mail.internet 包在 javax.mail 包的基础上做了进一步的扩展，对于 MIME 标准提供了完善的支持，所谓的 MIME 标准，指的是 Multipurpose Internet Mail Extensions，POP3、IMAP4、SMTP 等协议都是建立在 MIME 的标准之上，关于 MIME 标准的信息，读者可以查阅 RFC2045 文件。RFC2045 在下面的网址可以找到：

http：//www.ietf.org/rfc/rfc2045.txt

在实际的程序开发中，我们一般是利用 javax.mail.internet 包的类来处理邮件信息，例如读取邮件内容，组装邮件等，而利用 javax.mail 包的类来收发邮件、管理邮箱、管理文件夹等，凡是与具体每封邮件有关的任务都应该交给 javax.mail.internet 包来完成。

在 javax.mail.internet 包中所定义的类/接口几乎全都是对 javax.mail 包中相应类/接口的扩展，它们的方法、用法都十分相似，所以说，读者只要掌握了 javax.mail 包的用法，那么 Java Mail API 的用法就掌握了八成了。为了简明起见，我们就不详细介绍 javax.mail.internet 包的各个类/接口的详细用法了，本节仅仅介绍 MimePart 接口、MimeMessage 类的用法，至于其他的类/接口的用法，读者可以参照上一节的介绍，再看看 Java Mail API 的文档，就可以很容易地掌握 javax.mail.internet 包的用法了。

12.3.1　MimePart 接口

在上文中，我们已经提到过 javax.mail 包定义了一个 Part 接口，映射邮件的结构信息，在 javax.mail.internet 包中，也定义了一个与之相对应的接口——MimePart 接口，MimePart 接口与 Part 接口相比，增加了对 MIME 标准的支持。功能也更为强大。MimePart 接口继承自 Part 接口，能够使用在 Part 接口中声明的方法。下面是在 MimePart 接口中所声明的重

要方法。

public void addHeaderLine(String line) throws MessagingException;　给邮件添加信息头，可以使用 getHeader()方法或者 getAllHeaderLines()获取邮件的信息头，这两个方法也是在 MimePart 接口中声明的。

public String getEncoding() throws MessagingException;　这个方法可以获取邮件的编码方式。

public String getContentID() throws MessagingException;　这个方法可以获取邮件的 Content_ID。如果这个值没有设置，那么返回 NULL。

public String getContentMD5() throws MessagingException;　这个方法可以获取邮件的 Content_MD5 摘要信息。

public void setContentMD5(String md5) throws MessagingException;　这个方法和 getContentMD5()方法的作用相反，是设定邮件的 Content_MD5 摘要信息。

public String[] getContentLanguage() throws MessagingException;　这个方法可以获知在邮件信息头 Content-Language 所指定的语言信息。

public void setText(String text) throws MessagingException;　在组装邮件的时候，可以利用 setText()方法把一段普通文本设为邮件的正文，与此同时，邮件的 Content-Type 属性也被设为 text/plain。

public void setText(String text，String charset) throws MessagingException;　这个方法的作用同上，只不过还同时设定了邮件正文所使用的字符集，例如 ISO-8859-1，第二个参数 charset 就用于指定邮件内容所使用的字符集。

上面介绍的就是在 MimePart 接口中定义的重要方法，读者可能会问：如何才能够使用这些方法呢?因为 MimePart 是一个接口，它本身并不直接实现这些方法，所以我们需要构造一个 MimeMessage 对象，然后再利用 MimeMessage 对象调用 MimePart 接口的方法，因为 MimeMessage 类实现了 MimePart 接口的方法。

12.3.2　MimeMessage 类

MimeMessage 类实现了 MimePart 接口的方法，继承了 Message 类的方法，我们若想发送或者察看 MIME 编码类型的邮件，那么就必须使用 MimeMessage 类的方法。读者读完本小节以后，就会发现在 MimeMessage 类中定义的很多方法的名字都和 Part 接口、Message 类的方法的名字一样，这不足为怪，MimeMesage 类覆盖了这些方法，以实现对 MIME 标准的支持。如何获取 MimeMessage 对象呢?固然可以直接调用 MimeMessage 类的构造方法，但是一个简便的方法是强制把 Message 对象转变为 MimeMessage 对象。例如下面的代码。

例：

```
<%
Message message[] = folder.getMessages();
MimeMessage mm=(MimeMessage)message[9];
out.println(mm.getContentLanguage());
%>
```

直接构造 MimeMessage 对象的例子是：

例：

```
<%
Properties props = new Properties();
props.put("mail.smtp.host"，host);
Session sess = Session.getDefaultInstance(props，null);
sess.setDebug(debug);
MimeMessage msg = new MimeMessage(sess);
%>
```

在 MimeMessage 类中定义的重要方法如下所示：

public Address[] getFrom() throws MessagingException　获取邮件的发件人邮箱地址列表。

public void setFrom(Address address) throws MessagingException　设置邮件的发信人地址列表。

public void setFrom() throws MessagingException　设置邮件的发信人地址，如果原来已经设定了一个值，那么原值会被新值覆盖掉。

public void addFrom(Address[] addresses) throws MessagingException;　在邮件的发信人地址中再添加一个发件人邮箱地址的列表。

public Address[] getRecipients(Message.RecipientType type) throws MessagingException;　利用这个方法可以获取此邮件的目的邮箱地址列表，但是对于目的邮箱地址的类型有一定的限制，比如说是抄送的邮箱地址还是直接送达的邮箱地址，参数 type 就是对目的邮箱地址类型的限制条件。关于目的邮箱地址的类型，读者可以参考上文介绍 Message 类的部分，也可以参阅 Java Mail API 文档中关于 Message.RecipientType 类的介绍。

public Address[] getAllRecipients() throws MessagingException;　这个方法也是获取邮件的目的邮箱地址列表，但是对目的邮箱地址的类型没有限制，所有的邮箱地址都被存储在一个 Address 数组结构中返回。

public void setRecipients(Message.RecipientType type，Address[] addresses) throws MessagingException;　这个方法用于指定邮件的目的邮箱地址列表，第一个参数 type 指定了目的邮箱地址的类型，是抄送的，或者是直接送达的，第二个参数就是邮箱地址列表，如果原来已经指定了一个幕的邮箱地址列表，那么原来的值会被覆盖掉。

public void setRecipients(Message.RecipientType type ， String addresses) throws MessagingException;　这个方法用于指定邮件的目的邮箱地址，与上一个方法的不同之处在于：前一个方法一次就指定一个地址列表，可能含有很多邮箱地址，而这个方法只能够指定一个目的邮箱地址，如果原来已经指定了一个目的邮箱地址，那么原值被覆盖。

public void addRecipients(Message.RecipientType type，Address[] addresses) throws MessagingException;　这个方法和前面的几个方法都有相似之处，不过这个方法不会覆盖原来的值，它只是在原来目的邮箱地址列表的基础上再加上一个目的邮箱地址列表。

public Address[] getReplyTo() throws MessagingException;　利用这个方法可以获取此邮件所指定的回复邮箱地址列表。

public void setReplyTo(Address[] addresses) throws MessagingException;　利用这个方法可以指定此邮件的回复地址列表，当收信人想回复这封邮件的时候，设想他点击了某个按钮，进入了写信的界面，那么这个回复地址列表会自动出现在此写信界面的收件人邮箱地址栏中。

public String getSubject() throws MessagingException;　获取信件的主题。

public void setSubject(String subject) throws MessagingException;　设置邮件的主题。

public void setSubject(String subject，String charset) throws MessagingException; 设置邮件的主题，同时还指定了主题字符串所使用的字符集。

public java.util.Date getSentDate() throws MessagingException;　获取邮件的发送日期，所谓的发送日期，是指由发信方指定的一个邮件信息头，并非是邮件系统发送这封信的真正时间，利用 setSentDate()方法可以设定邮件的发送日期。

public void setSentDate(java.util.Date d) throws MessagingException;　设定此邮件的发送日期。

public java.util.Date getReceivedDate() throws MessagingException;　获取此邮件的收信日期。

public int getSize() throws MessagingException;　获取邮件的大小，返回值为此邮件的字节数。

public int getLineCount() throws MessagingException;　返回邮件正文的行数，如果无法获取，那么返回-1。

public String getContentType() throws MessagingException;　获取此邮件的 Content-Type 信息，例如 text/plain、text/html 等等。

public boolean isMimeType(String mimeType) throws MessagingException;　判断邮件是否属于某种 MIME 类型，参数 mimeType 的值可能是类似于 text/plain 的形式。例：

```
<%
Message message[] = folder.getMessages();
MimeMessage mm=(MimeMessage)message[9];
out.println(mm. isMimeType(“text/plain”));
%>
```

public String getMessageID() throws MessagingException;　这个方法的作用是返回此邮件的标识符，每封邮件都有一个唯一的标识符，请看下面的 JSP 代码片断。

例：

```
<%
Message message[] = folder.getMessages();
MimeMessage mm=(MimeMessage)message[9];
out.println(mm.getMessageID());
%>
```

运行的结果为：

```
<200105010524.NAA07980@www8.kaxiu.com>
```

public InputStream getInputStream() throws IOException，MessagingException;　此

方法返回一个输入流对象，利用这个输入流对象，可以流式地从邮件中读入数据，包括邮件正文和邮件信息头等。

protected InputStream getContentStream() throws MessagingException; 此方法返回一个输入流对象，利用这个输入流对象，可以流式的从邮件正文中读入数据，注意，仅仅是读入正文的数据。

public Object getContent() throws IOException，MessagingException; 利用这个方法可以返回邮件的正文，返回值是一个 Java 对象。

public void setContent(Object o，String type) throws MessagingException; 这个方法的作用是把数据写入邮件的正文，第一个参数就是需要写入到邮件正文的数据，第二个参数指定邮件的类型，是普通的文本邮件，或者是 HTML 格式的邮件，还是其他类型的邮件。

public void setText(String text) throws MessagingException; 使用这个方法可以设定 text/plain 类型的邮件的正文。

public void setText(String text，String charset) throws MessagingException; 使用这个方法可以设定 text/plain 类型的邮件的正文，同时还指定了正文字符串的编码方式，例如 GB2312。

public void setContent(Multipart mp) throws MessagingException; 这个方法同样是设定邮件的正文，不过这次不是什么文本数据了，而是一个 Multipart 对象。利用 Multipart 对象，一封邮件可以包含若干个复合的部分，请参考本章 12.5 节的例子。

public Message reply(boolean replyToAll) throws MessagingException; 这个方法的作用是：根据源邮件的信息，自动产生一个回复邮件(Message 对象)，这个回复邮件的各项属性都设定好了，如目标地址、发信人邮箱地址等，回复邮件的主题也被设定为在源主题的前面加上”Re：”字样，我们可以用 setXXX()方法对这些属性进行微调，再调用 Transport.send()方法，就可以实现回信的操作了。参数 replyToAll 指定是否回复原邮件的所有发信人。

public void writeTo(OutputStream os) throws IOException，MessagingException; 指定一个输出流，利用这个输出流对象，往邮件正文中写入数据，这个方法如何使用呢?首先是创建一个 MimeMessage 对象，然后创建一个输出流对象，接着调用 writeTo()方法，把这个输出流对象与 MimeMessage 对象关联在一起，利用输出流对象的 write()方法，就可以往邮件正文中写入数据了。

public String[] getHeader(String name) throws MessagingException; 这个方法的作用是获取邮件的投递信息头，邮件的投递信息头类似于 HTTP Header。一个邮件投递信息头有多个值，所以此方法的返回值为字符串数组。

public void setHeader(String name，String value) throws MessagingException; 这个方法的作用是设置某个邮件投递信息头的值，如果原来已经赋值，那么原值会被覆盖掉。

public void addHeader(String name，String value) throws MessagingException; 添加新的邮件投递信息头。

public void removeHeader(String name) throws MessagingException; 删除某个邮件投递信息头。

public Enumeration getAllHeaders() throws MessagingException; 获取所有邮件投

递信息头的列表，返回值是一个枚举对象。这个方法如何使用呢?请看下面的代码片断。

例：

```
<%
Message msg=folder.getMessage(1);
MimeMessage mm=(MimeMessage)msg;
Enumeration enum=msg.getAllHeaders();
while(enum.hasMoreElements())
{
    out.println(enum.nextElement());
}
%>
```

上面代码段的作用是把所有的邮件投递信息头的名称一一输出。

public Flags getFlags() throws MessagingException; 使用这个方法可以获取此邮件被打上的标记，返回值为 Flags 对象，一旦获取了 Flags 对象，我们就可以利用上一节中所介绍的 Flags 类的方法来对这个标记进行细致处理。利用 setFlags()方法可以给某封邮件打上标记。

public boolean isSet(Flags.Flag flag) throws MessagingException; 这个方法的作用是判断此邮件有没有被打上某种标记。

public void setFlags(Flags flag，boolean set) throws MessagingException; 给邮件打上某种标记。

public void saveChanges() throws MessagingException; 在关闭文件夹以前调用这个方法，保存对文件夹中的邮件所作的改变，例如改变了邮件的标题或者是给邮件打上了某个标记等等。请注意，文件夹的打开方式必须设定为 READ_WRITE，否则这个方法无效。

12.4　Sun Protocol Privider API 简介

在上文中我们已经提到过，Java Mail API 实际上分为两部分，第一部分是通用的的 API，适用于不同的邮件系统，无论是 POP3 系统还是 IMAP4 系统，都能够正确执行。第二部分就是扩展 API，分别对不同的邮件系统提供了不同的解决方案。这第二部分就是 Sun Protocol Privider API，它又包含了三个包，单独提供了对 POP3 协议、IMAP4 协议、SMTP 协议的支持。这三个包的名字如下所示：

com.sun.mail.pop3 包

com.sun.mail.smtp 包

com.sun.mail.imap 包

这三个包所定义的类、方法绝大部分都是继承了 javax.mail 包、javax.mail.internet 包的相应的类或者接口。例如 com.sun.mail.smtp 包定义了 SMTPMessage 类和 SMTPTransport 类，前者继承了 MimeMessage 类(javax.mail.internet)，后者继承了 Transport 类(javax.mail 包)。

如果我们开发的是跨平台、跨系统的邮件管理系统，那么我们建议你不要使用 Sun Protocol Privider API，因为这很容易导致不兼容的情况发生，如果你是针对特定的邮件系

统开发程序，例如 IMAP4(INTERNET MESSAGE ACCESS PROTOCOL - VERSION 4rev1)邮件系统，那么使用 Sun Protocol Privider API 中的 com.sun.mail.imap 包有可能获得较好的运行效率。

12.5　使用 Java Mail API 访问 Mail 服务器

在本节中，我们将使用十个完整的例子程序说明如何使用 Java Mail API 访问 Mail 服务器。这十个例子涵括了最常用的功能，例如粘贴附件、发送 HTML 格式的邮件等等。至于一些复杂高级的功能，例如 Java Mail API 如何与 JavaBeans、EJB、JNDI 、Tag Library 等技术结合起来，或者是 Mail System 的事件处理技术，不包括在本节的范围之内，有兴趣的读者可以自己研究。稍微综合一些的例子，读者可以参考《JSP深入編程》 。

12.5.1　发送普通邮件

下面是一个发送普通类型邮件的 JSP 程序，请看程序清单 12.1(mail.jsp)。

程序清单 12.1

```
<%--
File Name：mai.jsp
Author：fancy
Date：2001.5.10
Note：how to use java mail api to send email
--%>

<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>

<%
out.println("hehe");
String to="Rainbow@Rainbow.pku.edu.cn";
String from="fancyrainbow@263.net";
String subject="Test";

Properties props = System.getProperties();
props.put("mail.smtp.auth"，"false");
props.put("mail.smtp.host"，"Rainbow.pku.edu.cn");

Session sess = Session.getInstance(props，null);
sess.setDebug(true);
```

```
Message msg = new MimeMessage(sess);
msg.setFrom(new InternetAddress(from));
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to，false));
msg.setSubject(subject);
msg.setSentDate(new Date());
msg.setText("Hello World!");
Transport.send(msg);
%>
```

程序清单 12.1 的流程是这样的，首先获取一个 Session 对象，然后利用这个 Session 对象构造一个 Message 对象，此 Message 对象就代表了一个完整的邮件模型，调用 Message 对象的 setXXX()方法，设定邮件信息的方方面面，如目的邮箱地址、发信人邮箱地址、发送日期、主题、正文等等，一切就绪之后，就可以调用 Transport.send()方法，把这个 Message 对象发送出去，也就是把邮件发送出去。请注意，这个方法仅仅适用于不需要身份验证的 SMTP 服务器。如果 SMTP 服务器需要身份验证信息，那么这个程序在运行时会出现错误。把邮件发送出去后，我们使用 Outlook 等客户端软件查看邮箱，将会发现我们又收到一封新邮件，这正是我们利用 mail.jsp 程序给自己发送的邮件，如图 12.1 所示。

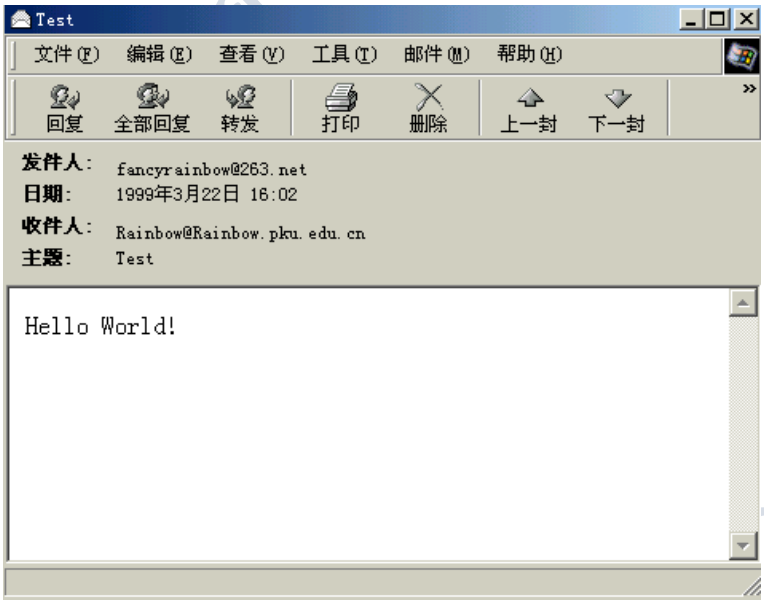


图 12.1　mail.jsp

12.5.2　发送 HTML 格式的信件

程序清单 12.2

```
<%--
File Name：mail1.jsp
Author：fancy
Date：2001.5.10
Note：how to use java mail api to send email
```

```
--%>

<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>

<%
out.println("hehe");
String mailhost="Rainbow.pku.edu.cn";
String to="Rainbow@Rainbow.pku.edu.cn";
String from="fancyrainbow@263.net";
String subject="笑敖江湖主题曲";
Properties props = System.getProperties();
props.put("mail.smtp.auth"，"false");
props.put("mail.smtp.host"，"Rainbow.pku.edu.cn");
Session sess = Session.getInstance(props，null);
sess.setDebug(true);
Message msg = new MimeMessage(sess);
msg.setFrom(new InternetAddress(from));
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to，false));
msg.setSubject(subject，"ISO-8859-1");
msg.setSentDate(new Date());
StringBuffer sb = new StringBuffer();
sb.append("<HTML>\n");
sb.append("<HEAD>\n");
sb.append("<TITLE>\n");
sb.append(subject + "\n");
sb.append("</TITLE>\n");
sb.append("</HEAD>\n");
sb.append("<BODY>\n");
sb.append("<H1><Font color='red'>笑敖江湖主题曲</font></H1>" + "\n");
sb.append("英雄肝胆两相照，江湖儿女日见少，<br>"+"\n");
sb.append("心还在，人去了，回首一片风雨飘摇，<br>"+"\n");
sb.append("心还在，人去了，回首一片回首一片风雨飘摇。<br>"+"\n");
sb.append("</BODY>\n");
sb.append("</HTML>\n");
msg.setDataHandler(new DataHandler(sb.toString()，"text/html"));
Transport.send(msg);
%>
```

在 12.5.1 小节中，我们举了一个例子，演示如何利用 Java Mail API 发送普通类型的邮

件，在本小节所举的例子中，我们利用 Java Mail API，实现了发送 HTML 格式邮件的功能，这与上一小节相比，可以说是一个很大的进步。究竟如何发送 HTML 格式的邮件呢?请看程序清单 12.2(mail1.jsp)。

程序清单 12.2 和程序清单 12.1 在结构上基本相同，只是在设定邮件正文内容时不太一样，在程序清单 12.1 中，我们通过使用 Message 对象的 setText()方法，直接指定邮件的文本正文，但是在程序清单 12.2 中，这一步骤就复杂多了。我们首先需要创建 StringBuffer 对象，然后往 Buffer 里面写入一个完整的 HTML 网页(利用 StringBuffer 对象的 append()方法)，这一步做完以后，利用下面的代码把 StringBuffer 对象所包含的数据和 Message 对象结合在一起：

```
msg.setDataHandler(new DataHandler(sb.toString()，"text/html"));
```

这一行代码其实就是设定了邮件的正文内容以及邮件的类型——text/html。我们可以使用 Outlook 查看程序清单 12.2 的运行结果，如图 12.2 所示，这显然是一封 HTML 格式邮件，因为邮件正文前面的七个字尺寸很大，而且又是红色的文本，普通文本格式的邮件做不到这个效果。

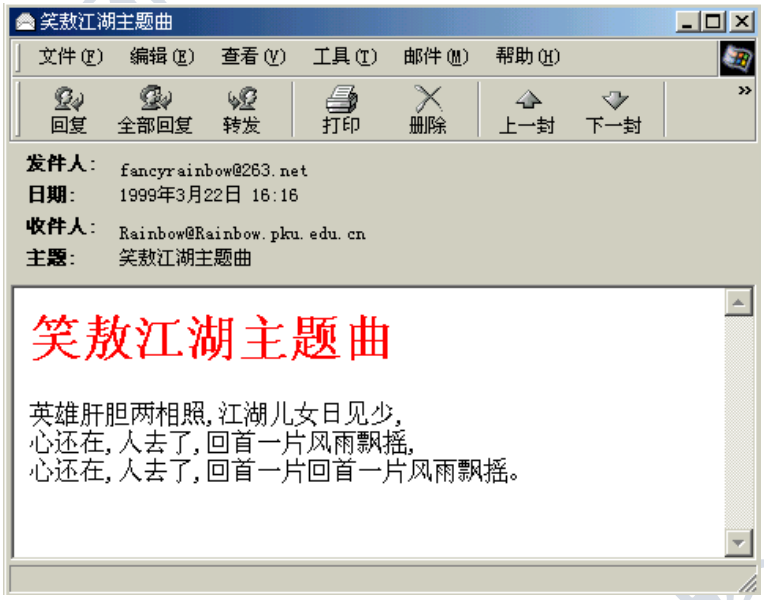


图 12.2　mail1.jsp

读者需要特别注意一个问题：那就是 Java Mail API 的中文问题，Java Mail API 也有中文问题吗?是的，如果你使用 setSubject()方法直接设定中文的主题，或者使用 setText()方法直接指定中文内容，那么在使用 Outlook 等客户端软件查看邮件时，你看到的将是一团乱码。如何解决这个问题呢?我们只需要指定邮件主题和邮件正文的字符集为 ISO-8859-1 即可。请看下面的代码片断。

例：

```
<%
msg.setSubject(subject，"ISO-8859-1");
msg.setText(text，” ISO-8859-1”);
```

%>

这样一来，利用此程序发送的中文邮件就能够在 Outlook 等客户端软件中正确显示了，不过如果是利用 StringBuffer 对象发送 HTML 格式的邮件，对于邮件的正文似乎就没有中文问题，对于邮件的主题，仍有中文问题。

12.5.3　发送含有附件的邮件

程序清单 12.3

```
<%--
File Name：mail2.jsp
Author：fancy
Date：2001.5.10
Note：how to use java mail api to send email
--%>

<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>

<%
out.println("hehe");
String mailhost="Rainbow.pku.edu.cn";
String to="Rainbow@Rainbow.pku.edu.cn";
String from="fancyrainbow@263.net";
String subject="附件";
Properties props = System.getProperties();
props.put("mail.smtp.auth"，"false");
props.put("mail.smtp.host"，"Rainbow.pku.edu.cn");
Session sess = Session.getInstance(props，null);
sess.setDebug(true);
MimeMessage msg = new MimeMessage(sess);
msg.setFrom(new InternetAddress(from));
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to，false));
msg.setSubject(subject，"ISO-8859-1");
msg.setSentDate(new Date());

// create and fill the first message part
MimeBodyPart mbp1 = new MimeBodyPart();
mbp1.setText("请看附件"，"ISO-8859-1");
```

北京希望电子出版社

```
// create the second message part
MimeBodyPart mbp2 = new MimeBodyPart();

File file=new File("D：\\Borland\\JBuilder4\\Tomcat\\webapps\\ROOT\\mail2.jsp");
// attach the file to the message
FileDataSource fds = new FileDataSource(file);
mbp2.setDataHandler(new DataHandler(fds));
mbp2.setFileName(fds.getName());
out.println(fds.getName());

// create the Multipart and its parts to it
Multipart mp = new MimeMultipart();
mp.addBodyPart(mbp1);
mp.addBodyPart(mbp2);

// add the Multipart to the message
msg.setContent(mp);
Transport.send(msg);
%>
```

程序清单 12.3 演示了如何使用 Java Mail API 发送带有附件的邮件，我们可以归纳程序清单 12.2 的主要流程如下：

（1）首先创建一个 MimeMessage 类型的对象 msg，在这里使用 Message 类不是很合适。

（2）调用 setXXX()方法指定 MimeMessage 对象(msg)的各种属性。

（3）创建 MimeBodyPart 类型的对象 mbp1，此 MimeBodyPart 对象就相当于邮件的主体部分。MimeBodyPart 类是在 javax.mail.internet 包中定义的，继承自 BodyPart 类。

例：

```
<%
// create and fill the first message part
MimeBodyPart mbp1 = new MimeBodyPart();
mbp1.setText("请看附件"，"ISO-8859-1");
%>
```

（4）创建第二个 MimeBodyPart 类型的对象 mbp2，此 MimeBodyPart 对象就相当于邮件的附件部分。通过下面代码把这个 MimeBodyPart 对象与附件关联在一起。

例：

```
<%
File file=new File("D：\\Borland\\JBuilder4\\Tomcat\\webapps\\ROOT\\mail2.jsp");
// attach the file to the message
FileDataSource fds = new FileDataSource(file);
mbp2.setDataHandler(new DataHandler(fds));
mbp2.setFileName(fds.getName());
%>
```

在上面的代码中，我们首先创建一个 File 对象，然后创建一个 FileDataSource 对象，把 File 对象作为 FileDataSource 类的构造函数参数传入，接下来顺序调用 setDataHandler()方法和 setFileName()方法，把附件和 MimeBodyPart 对象绑定在一起。

（5）创建一个 Multipart 类型的对象 mp，两次调用 addBodyPart()方法，把上面所创建的两个 MimeBodyPart 对象添加到 Multipart 对象(mp)中去，这样做的目的是把邮件主体与附件部分组装起来，构成一个复合的整体。请看下面的代码。

例：

```
<%
// create the Multipart and its parts to it
Multipart mp = new MimeMultipart();
mp.addBodyPart(mbp1);
mp.addBodyPart(mbp2);
%>
```

（6）最后一步是调用 MimeMessage 对象的 setContent()方法，参数为 Multipart 对象，然后使用 Transport.send()方法把邮件发送出去。

我们还可以按照上面介绍的方法，创建多个 MimeBodyPart 对象，粘贴多个附件，这无论是在技术上还是在原理上都是可能的。程序清单 12.3 的运行效果如图 12.3 所示，读者应该注意到在附件栏中，已经注明了有一个附件(mail2.jsp)，大小为 1.62KB。

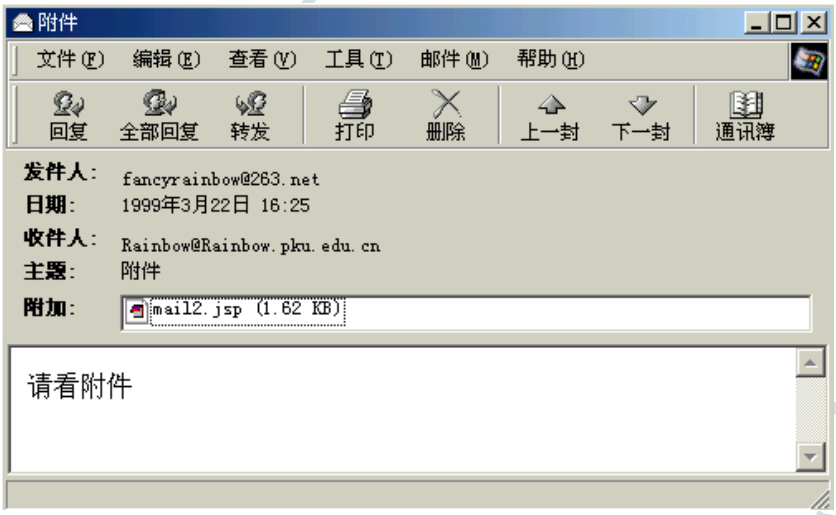


图 12.3　mail2.jsp

12.5.4　发送复合邮件

程序清单 12.4

```
<%--
File Name：mail3.jsp
Author：fancy
Date：2001.5.10
Note：how to use java mail api to send email
```

```
--%>

<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>

<%
out.println("hehe");
String mailhost="Rainbow.pku.edu.cn";
String to="Rainbow@Rainbow.pku.edu.cn";
String from="fancyrainbow@263.net";
String subject="Test";
Properties props = System.getProperties();
props.put("mail.smtp.auth"，"false");
props.put("mail.smtp.host"，"Rainbow.pku.edu.cn");
Session sess = Session.getInstance(props，null);
sess.setDebug(true);
MimeMessage msg = new MimeMessage(sess);
msg.setFrom(new InternetAddress(from));
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to，false));
msg.setSubject(subject);
msg.setSentDate(new Date());
// create and fill the first message part
MimeBodyPart mbp1 = new MimeBodyPart();
mbp1.setText("hello world");

// create and fill the second message part
MimeBodyPart mbp2 = new MimeBodyPart();
// Use setText(text，charset)，to show it off !
mbp2.setText("你好"，"ISO-8859-1");

// create the Multipart and its parts to it
Multipart mp = new MimeMultipart();
mp.addBodyPart(mbp1);
mp.addBodyPart(mbp2);

// add the Multipart to the message
msg.setContent(mp);
Transport.send(msg);
%>
```

北京希望电子出版社

在程序清单 12.4 中，我们演示了如何发送一个复合邮件，所谓的复合邮件，指的是邮件由多个部分组成。读者不难发现，程序清单 12.4 和程序清单 12.3 十分相似，确实如此，程序清单 12.4 和程序清单 12.3 的流程完全一样，前者比后者还要简单，在程序清单 12.3 中，第二个 MimeBodyPart 对象包含的是一个作为附件的文件，而在程序清单 12.4 中，第二个 MimeBodyPart 对象包含的是普通的文本，它也是邮件主体的一部分，但不是邮件的附件。

程序清单 12.4 的运行效果如图 12.4 所示，我们可以看到，在邮件的两个部分之间，被一条横线分开了。

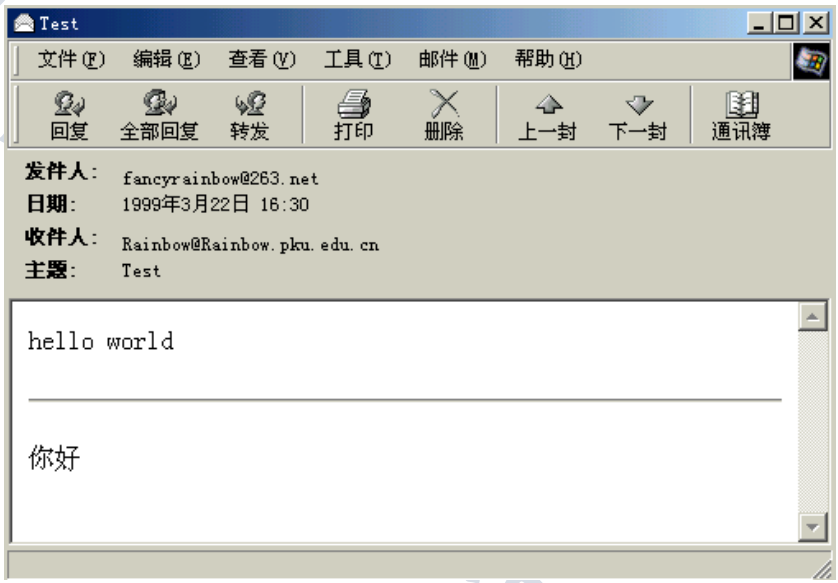


图 12.4　mail3.jsp

12.5.5　多个邮件投递地址

程序清单 12.5

```
<%--
File Name：mai4.jsp
Author：fancy
Date：2001.5.10
Note：how to use java mail api to send email
--%>

<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>
```

```
<%
out.println("hehe");
String to="Rainbow@Rainbow.pku.edu.cn";
String to1="Fancy@Rainbow.pku.edu.cn";
String from="fancyrainbow@263.net";
String subject="Test";

Properties props = System.getProperties();
props.put("mail.smtp.auth"，"false");
props.put("mail.smtp.host"，"Rainbow.pku.edu.cn");

Session sess = Session.getInstance(props，null);
sess.setDebug(true);

Message msg = new MimeMessage(sess);
msg.setFrom(new InternetAddress(from));
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to，false));
msg.addRecipients(Message.RecipientType.TO，InternetAddress.parse(to1，false));
msg.addRecipients(Message.RecipientType.CC，InternetAddress.parse(to1，false));
msg.setSubject(subject);
msg.setSentDate(new Date());
msg.setText("Hello World!");
Transport.send(msg);
%>
```

程序清单 12.5 演示了一次如何向多个信箱发送同一封信。诀窍在于使用 MimeMessage 类/Message 类的 addRecipients()方法。请看下面的代码。

例：

```
<%
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to，false));
msg.addRecipients(Message.RecipientType.TO，InternetAddress.parse(to1，false));
msg.addRecipients(Message.RecipientType.CC，InternetAddress.parse(to1，false));
%>
```

在上面的代码中，给当前的邮件指定了两个需直接送达的邮箱地址，一个需要抄送的邮箱地址。

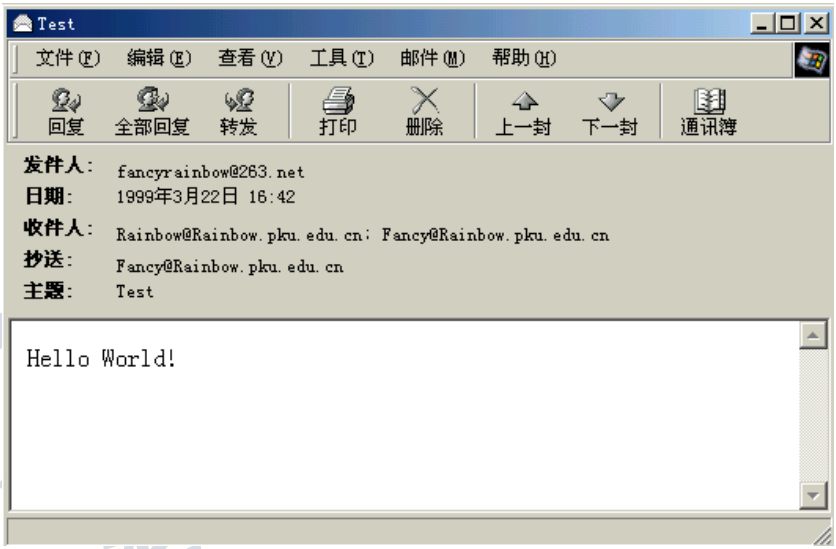


图 12.5　mail4.jsp

程序清单 12.5 的运行效果如图 12.5 所示，读者请特别注意收件人栏、抄送栏的信息。

12.5.6　SMTP 服务器身份验证

程序清单 12.6

```
<%--
File Name：mail5.jsp
Author：fancy
Date：2001.5.10
Note：how to use java mail api to send email
--%>

<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>
<%!
public class SmtpAuthenticator extends Authenticator
{
    protected PasswordAuthentication getPasswordAuthentication()
    {
        return new PasswordAuthentication("fancy"， "fancy");
    }
}
%>

<%
out.println("hehe");
```

```
String to="Rainbow@Rainbow.pku.edu.cn";
String to1="Rainbow@Rainbow.pku.edu.cn";
String from="fancyrainbow@263.net";
String subject="Test";

Properties props = System.getProperties();
props.put("mail.smtp.auth"，"true");
props.put("mail.smtp.host"，"Rainbow.pku.edu.cn");

SmtpAuthenticator sa=new SmtpAuthenticator();
Session sess = Session.getInstance(props，sa);
sess.setDebug(true);

Message msg = new MimeMessage(sess);
msg.setFrom(new InternetAddress(from));
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to，false));
msg.setRecipients(Message.RecipientType.TO，InternetAddress.parse(to1，false));
msg.setSubject(subject);
msg.setSentDate(new Date());
msg.setText("Hello World!");
Transport.send(msg);
%>
```

程序清单 12.6 演示了如何通过 SMTP 服务器的身份验证，并利用 SMTP 服务器发送邮件。

为了防止恶意用户利用 SMTP 服务器搞破坏，所以 SMTP 服务器一般需要身份验证，只有通过了身份验证的用户才有权利利用 SMTP 服务器发送邮件。我们总结了这样的一个流程，可以通过 SMTP 服务器的身份验证。

（1）在 JSP 声明段中，声明一个 SmtpAuthenticator 类，SmtpAuthenticator 类必须继承自 Authenticator 类，注意，这是 javax.mail 包中定义的 Authenticator 类，而不是 java.net 包的 Authenticator 类。这个 SmtpAuthenticator 类还必须覆盖 Authenticator 类的 getPasswordAuthentication()方法，getPasswordAuthentication()方法必须声明为 protected 类型，返回一个 PasswordAuthentication 类型的对象，此 PasswordAuthentication 对象包含了通过 SMTP 服务器身份验证所需的用户名和密码。

例：

```
<%!
public class SmtpAuthenticator extends Authenticator
{
    protected PasswordAuthentication getPasswordAuthentication()
    {
        return new PasswordAuthentication("fancy"，"fancy");
    }
}
%>
```

（2）创建一个 SmtpAuthenticator 类的实例。

例：

```
<%
SmtpAuthenticator sa=new SmtpAuthenticator();
%>
```

（3）调用 Session 类的 getInstance()方法，获取一个 Session 类型的对象。这个方法的第一个参数是一个 Properties 类型的对象，第二个参数就是 SmtpAuthenticator 对象。JSP 引擎在执行这个方法的时候，首先获取 SmtpAuthenticator 对象，然后自动调用此对象的 getPasswordAuthentication()方法，取得用户名和密码，再把它们发送到 SMTP 服务器去，以便通过身份验证。如果身份验证通过了，那么 getInstance()方法就会返回一个 Session 对象。

例：

```
<%
Session sess = Session.getInstance(props，sa);
%>
```

（4）除了上面所说的三点以外，程序清单 12.6 和本章别的程序在发送邮件的流程上没有任何区别。

如何配置 ArGoSoft Mail Server，使它支持身份验证

在上一章，我们已经介绍了 ArGoSoft Mail Server 的配置方法，在这一章，我们再做一点补充，使得 Mail Server 要求身份验证。

在 ArGoSoft Mail Server 的管理界面中，选择 Tools→Options→SMTP Authentication。

在 Enable SMTP Authentication 和 Use POP3 User Names and Passwords 这两栏后面的小框中打上勾，单击 OK 即可，这样用户在通过 SMTP 服务器身份验证时，所需要的用户名和密码就是他所拥有的 POP3 邮箱的用户名和密码，如图 12.6 所示。

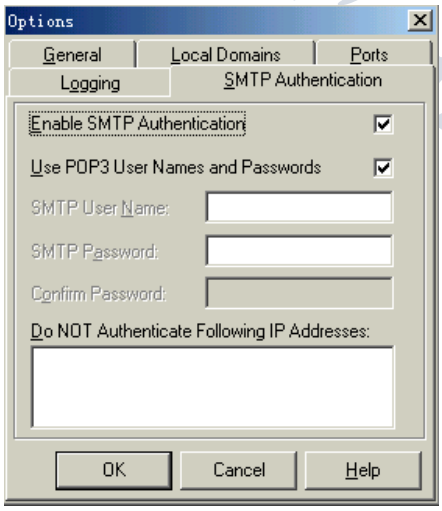


图 12.6　配置 ArGoSoft Mail Server

注意：ArGoSoft Mail Server 是本章程序所使用的 Mail Server。

12.5.7　文件夹邮件列表

程序清单 12.7

```
<%--
File Name：folder.jsp
Author：fancy
Date：2001.5.10
Note：list mail
--%>

<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>

<%
// Get a Store object
String mailhost="Rainbow.pku.edu.cn";
String user="Rainbow";
String password="Rainbow";
String protocol="pop3";
String url=null;
String root=null;

Properties props = System.getProperties();
Session sess= Session.getDefaultInstance(props，null);
sess.setDebug(true);
Store store = null;
store = sess.getStore(protocol);
store.connect(mailhost，user，password);
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);

Message message[] = folder.getMessages();
for (int i=0，n=message.length; i<n; i++)
{
      out.println(i+1+"："+message[i].getSubject()+"---->");
      out.println(message[i].getSize()+"<br>");
}
store.close();
%>
```

在程序清单 12.7(folder.jsp)中，演示了如何获取邮箱中邮件的列表。步骤如下。

（1）创建 Session 对象，这一步不需要再多说了吧。

（2）创建 Store 对象，Store 对象映射了邮箱的结构。请看下面的代码。

北京希望电子出版社

例：

```
<%
Store store = null;
store = sess.getStore(protocol);
store.connect(mailhost，user，password);
%>
```

在调用 Session 对象的 getStore()方法时，需要指定所使用的协议，如果你的邮箱是 POP3 邮箱，那么应该使用 getStore(“pop3”)，如果你的邮箱是 IMAP4 类型的，那么应该使用 getStore(“imap”)。

（3）获取 INBOX 文件夹。请看下面的代码。

例：

```
<%
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
%>
```

getFolder()方法的参数就是邮箱的名字，POP3 协议支持 INBOX 文件夹。调用 getFolder()方法以后，需要 Open 文件夹，然后才能够从文件夹中读取邮件。

（4）获取邮件列表。打开文件夹以后，使用 Folder 对象的 getMessages()方法，就可以返回一个 Message 数组，里面包含了当前文件夹中所有邮件的列表。请看下面的代码。

例：

```
<%
Message message[] = folder.getMessages();
%>
```

（5）处理邮件。剩下的事情，就是对每封具体的邮件进行处理了，这没有什么好介绍的。

程序清单 12.7 的运行效果如图 12.7 所示。

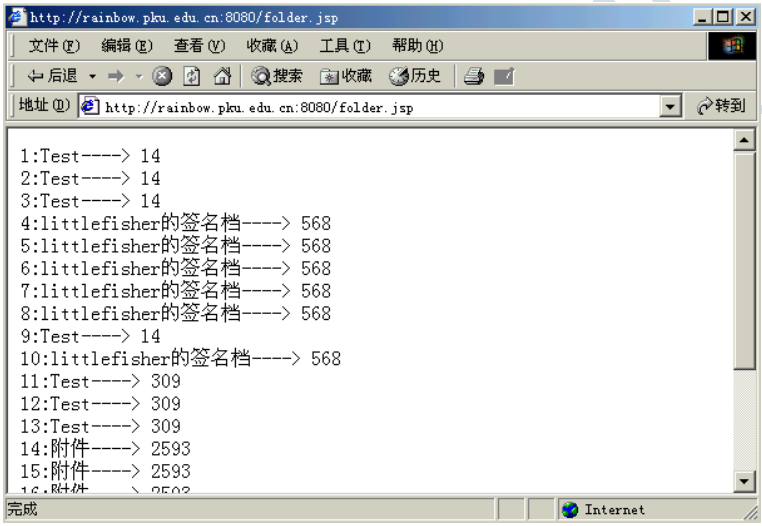


图 12.7　folder.jsp

12.5.8　查看邮件信息

程序清单 12.8

```
<%--
File Name：readmail.jsp
Author：fancy
Date：2001.5.10
Note：read mail
--%>

<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>

<%
// Get a Store object
String mailhost="Rainbow.pku.edu.cn";
String user="Rainbow";
String password="Rainbow";
String protocol="pop3";
String url=null;
String root=null;

Properties props = System.getProperties();
Session sess= Session.getDefaultInstance(props，null);
sess.setDebug(true);
Store store = null;
store = sess.getStore(protocol);
store.connect(mailhost，user，password);
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);

Message message[] = folder.getMessages();
MimeMessage mm=(MimeMessage)message[1];

out.println("Subject："+mm.getSubject()+"<br>");
out.println("Encoding："+mm.getEncoding()+"<br>");
out.println("Message ID："+mm.getMessageID()+"<br>");
out.println("Message Number："+mm.getMessageNumber()+"<br>");
out.println("Sent Date："+mm.getSentDate()+"<br>");
out.println("Received Date："+mm.getReceivedDate()+"<br>");
out.println("Size："+mm.getSize()+"byte<br>");
```

北京希望电子出版社

```
out.println("Flag：  "+"");
mm.setFlag(Flags.Flag.RECENT，  true);
Flags flag=mm.getFlags();

if (flag.contains(Flags.Flag.ANSWERED))
{
      out.println("ANSWERD");
}
else if (flag.contains(Flags.Flag.DELETED))
{
      out.println("DELETED");
}
else if (flag.contains(Flags.Flag.DRAFT))
{
      out.println("DRAFT");
}
else if (flag.contains(Flags.Flag.FLAGGED))
{
      out.println("FLAGGED");
}
else if (flag.contains(Flags.Flag.RECENT))
{
      out.println("RECENT");
}
else if (flag.contains(Flags.Flag.SEEN))
{
      out.println("SEEN");
}
else if (flag.contains(Flags.Flag.USER))
{
      out.println("USER");
}
else
{
      out.println("non flag");
}
out.println("<br>");
out.println("Folder：  "+mm.getFolder().getURLName().toString()+"<br>");
out.println("From：  "+"");
Address []addr=mm.getFrom();
for(int i=0;i<addr.length;i++)
{
      out.println(""+addr[i].toString()+"  ");
}
out.println("<br>");
```

北京希望电子出版社

```
out.println("To & CC & BCC：  "+"");
Address []addrTo=mm.getAllRecipients();
for(int i=0;i<addrTo.length;i++)
{
      out.println(""+addrTo[i].toString()+"  ");
}
out.println("<br>");

out.println("Reply To：  "+"");
Address []addrReply=mm.getReplyTo();
for(int i=0;i<addrTo.length;i++)
{
      out.println(""+addrReply[i].toString()+"  ");
}
out.println("<br>");
out.println("Content Language：  "+mm.getContentLanguage()+"<br>");
out.println("Content：  "+mm.getContent()+"<br>");
store.close();
%>
```

程序清单 12.8 的作用就是阅读邮件的信息，这适用于不带附件的邮件，程序清单 12.8 虽然比较长，但是十分简单，主要就是调用 MimeMessage 类、Message 类的 getXXX() 方法，没有什么难的地方，因此我们就不多做介绍了，读者自己领会吧。程序清单 12.8 的运行效果如图 12.8 所示。

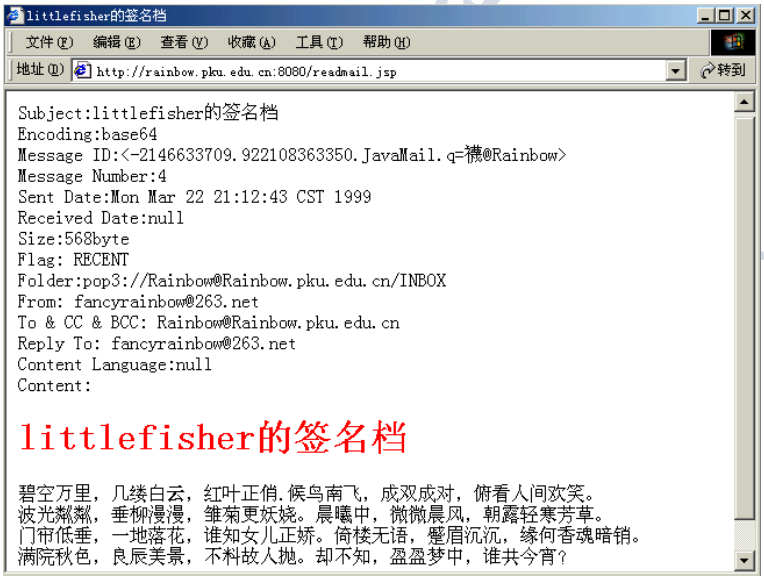


图 12.8　readmail.jsp

12.5.9　查看邮件附件

程序清单 12.9

```
<%--
File Name：readattr.jsp
Author：fancy
Date：2001.5.10
Note：how to use java mail api to send email
--%>

<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>

<%!
public static void handleMultipart(Multipart multipart，JspWriter out)
{
    try
    {
        int n=multipart.getCount();
        for(int i=0;i<n;i++)
        {
            handlePart(multipart.getBodyPart(i)，out);
        }
    }
    catch (Exception fe)
    {
        //to do it
    }
}
%>

<%!
public static void handlePart(Part part，JspWriter out)
{
    try
    {
        String disposition = part.getDisposition();
        String contentType = part.getContentType();
        if (disposition == null)
```

北京希望电子出版社

```
            {
                // When just body
            }
            else if (disposition.equals(Part.ATTACHMENT))
            {
                System.out.println("Attachment：  " + part.getFileName() + "  ：  "
                    + contentType);
                saveFile(part.getFileName()，part.getInputStream()，out);
            }
            else if (disposition.equals(Part.INLINE))
            {
                System.out.println("Inline：  " + part.getFileName() + "  ：  " + contentType);
                saveFile(part.getFileName()，part.getInputStream()，out);
            }
            else
            {    // Should never happen
                System.out.println("Other：  " + disposition);
            }
        }
        catch (Exception fe)
        {

        }
}
%>

<%!
public static void saveFile(String filename，InputStream input，JspWriter out)
{
        try
        {
            if (filename == null)
            {
                filename = File.createTempFile("xx"，".out").getName();
            }
            // Do no overwrite existing file
            filename="tmp"+filename;
            File file = new File(filename);
            //out.println(filename);
            for (int i=0; file.exists(); i++)
            {
                file = new File(filename+i);
            }
```

北京希望电子出版社

```
                out.println(file.getAbsolutePath());
                FileOutputStream fos = new FileOutputStream(file);
                BufferedOutputStream bos = new BufferedOutputStream(fos);
                BufferedInputStream bis = new BufferedInputStream(input);
                int aByte;
                while ((aByte = bis.read()) != -1)
                {
                        bos.write(aByte);
                        myOut.write(aByte);
                }
                bos.flush();
                bos.close();
                bis.close();
        }
        catch (Exception fe)
        {
                // to do it
        }
}
%>

<%
// Get a Store object
String mailhost="Rainbow.pku.edu.cn";
String user="Rainbow";
String password="Rainbow";
String protocol="pop3";
String url=null;
String root=null;

Properties props = System.getProperties();
Session sess= Session.getDefaultInstance(props，null);
sess.setDebug(true);
Store store = null;
store = sess.getStore(protocol);
store.connect(mailhost，user，password);
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
MimeMessage mm=(MimeMessage)message[15];
Object content = mm.getContent();
if(content instanceof Multipart)
{
```

北京希望电子出版社

```
        handleMultipart((Multipart)content，out);
    }
    else
    {
        handlePart(mm，out);
       }
    store.close();
    %>
```

程序清单 12.9(readattr.jsp)演示了如何读取邮件的附件。在 readattr.jsp 程序中，定义了三个方法，分别是 handleMultipart()方法、handlePart()方法、saveFile()方法，handleMultipart()方法的作用是把一个复合邮件分解为几个部分，每一部分都传递给 handlePart()方法进行处理，handlePart()方法的作用是具体处理邮件，在 handlePart()方法中，对传入的参数(邮件的一部分)进行判断，如果属于附件，那么调用 saveFile()方法，把附件保存起来，如果是邮件的主体，那么什么事也不做，跳出 handlePart()方法，saveFile()方法的主要作用就是创建一个临时文件，把附件作为一个输入流读入，再作为一个输出流写入到这个临时文件中。该临时文件所在的位置一般是服务器 bin 文件夹内。readattr.jsp 程序其实只是把邮件的附件保存起来，并没有把它显示出来，我们可以修改 saveFile()程序，输出邮件附件的保存目录，并自动生成一个超链接，当用户点击这个超链接，就可以下载此附件到客户端了。

读者应该注意到，在 readattr.jsp 程序中所定义的三个方法中，都有一个类型为 JspWriter 的参数，这个参数的作用是代替 JSP 的 Out 对象，把信息输出到客户端。

12.5.10　给 INBOX 划分文件夹

程序清单 12.10

```
<%--
File Name：list.jsp
Author：fancy
Date：2001.5.10
Note：list mail
--%>

<%@ page import="java.net.*" %>
<%@ page import="java.io.*" %>
<%@ page import="javax.activation.*" %>
<%@ page import="java.util.*" %>
<%@ page import="javax.mail.*" %>
<%@ page import="javax.mail.internet.*" %>

<%
// Get a Store object
String mailhost="Rainbow.pku.edu.cn";
String user="Rainbow";
```

```
String password="Rainbow";
String protocol="pop3";
String url=null;
String root=null;

Properties props = System.getProperties();
Session sess= Session.getDefaultInstance(props，null);
sess.setDebug(true);
Store store = null;
store = sess.getStore(protocol);
store.connect(mailhost，user，password);
Folder folder = store.getFolder("INBOX");
folder.open(Folder.READ_ONLY);
Message message[] = folder.getMessages();
message[0].setFlag(Flags.Flag.DELETED，true);
//MimeMessage mm=(MimeMessage)message[0];
//mm.setExpunged(true);
message[1].setFlag(Flags.Flag.DRAFT，true);
message[2].setFlag(Flags.Flag.DELETED，true);
message[3].setFlag(Flags.Flag.DELETED，true);
message[4].setFlag(Flags.Flag.RECENT，true);
message[5].setFlag(Flags.Flag.SEEN，true);
message[6].setFlag(Flags.Flag.RECENT，true);
message[7].setFlag(Flags.Flag.SEEN，true);
message[8].setFlag(Flags.Flag.SEEN，true);
message[9].setFlag(Flags.Flag.RECENT，true);
message[10].setFlag(Flags.Flag.RECENT，true);
message[11].setFlag(Flags.Flag.DRAFT，true);
message[12].setFlag(Flags.Flag.DELETED，true);
message[13].setFlag(Flags.Flag.RECENT，true);
message[14].setFlag(Flags.Flag.DRAFT，true);
message[15].setFlag(Flags.Flag.DELETED，true);
message[16].setFlag(Flags.Flag.RECENT，true);
message[17].setFlag(Flags.Flag.DRAFT，true);
out.println("收件箱<br>");
for(int i=0;i<message.length;i++)
{
      Flags flag=message[i].getFlags();
      if(flag.contains(Flags.Flag.RECENT))
            out.println(message[i].getMessageNumber()+"："+message[i].getSubject()+"<br>");
}

out.println("<br>草稿箱<br>");
```

北京希望电子出版社

```
for(int i=0;i<message.length;i++)
{
    Flags flag=message[i].getFlags();
    if(flag.contains(Flags.Flag.DRAFT))
        out.println(message[i].getMessageNumber()+"："+message[i].getSubject()+"<br>");
}

out.println("<br>保存箱<br>");
for(int i=0;i<message.length;i++)
{
    Flags flag=message[i].getFlags();
    if(flag.contains(Flags.Flag.SEEN))
        out.println(message[i].getMessageNumber()+"："+message[i].getSubject()+"<br>");
}

out.println("<br>垃圾箱<br>");
for(int i=0;i<message.length;i++)
{
    Flags flag=message[i].getFlags();
    if(flag.contains(Flags.Flag.DELETED))
        out.println(message[i].getMessageNumber()+"："+message[i].getSubject()+"<br>");
}

store.close();
%>
```

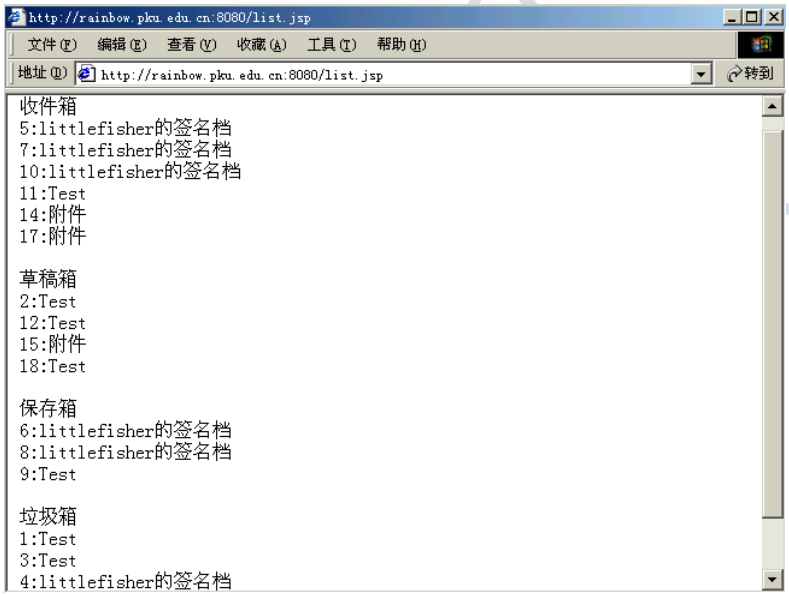


图 12.9　list.jsp

程序清单 12.10(list.jsp)演示了两大功能：给邮件打上标记和根据这些标记给邮件分类。第一个功能只要调用 Message 类的 setFlag()方法就可以实现了。第二个功能也比较简单，只要利用 getFlags()方法获取一个 Flags 对象，再调用 Flags 对象的 contains()方法就可以判断给邮件打的是何种标记了。利用这第二个功能，我们可以把 INBOX 文件夹分为收件箱、垃圾箱、保存箱、草稿箱等若干个邮箱，方便了邮箱的管理。

程序清单 12.10 十分简单，我们也不多做介绍了，读者可以自己研究程序的代码。

注意：本章所使用的 JSP 服务器为 Tomcat 3.2，Mail 服务器为 ArGoSoft Mail Server，操作系统平台为 Windows Me，JDK 为 JDK 1.3 版本，Java Mail API 为 1.2 版本，JAF 为 1.1 版本，Tomcat 3.2 服务器缺省状态下并不支持 Java Mail API，需要对其进行配置。方法是把包含有 Java Mail API、JAF 的 jar 文件拷贝到 Tomcat 的 lib 文件夹下面，并把它们的路径添加到系统变量 CLASSPATH 中。例如：打开 tomcat.bat 文件，添加下面的几行代码。

```
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\mail.jar
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\mailapi.jar
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\activation.jar
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\imap.jar
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\pop3.jar
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\smtp.jar
```

保存文件，重新启动服务器，就可以运行上面提供的例子程序了。

12.6　本 章 小 结

本章首先对 Java Mail API 做了简单介绍，然后重点介绍了若干个类、接口的用法，在最后一节，举了十个完整的例子，演示了如何使用 Java Mail API 访问具体的 Mail Server。学完本章以后，读者应该能够利用 Java Mail API 编写出简单的发送邮件、读取邮件的 JSP、Java 程序。

附录 1　支持 EJB1.0 技术规范的 EJB 平台、开发工具一览表

下表数据截止日期:20-Mar-01

公司名称	产品名称	网址
应用程序服务器		
Allaire Corporation	JRun Server	www.allaire.com
ATG	Dynamo Application Server	www.atg.com
BEA Systems	BEA WebLogic	www.bea.com
Bluestone Software，Inc.	Total-e-Server 7.2	www.bluestone.com
Borland Corporation	Borland Application Server	www.borland.com
BROKAT Technologies	Brokat Server Technologies	www.brokat.com
Compaq	NonStopTM Enterprise Application Server	www.compaq.com
Evidian	JOnAS	www.evidian.com
ForteTM Software	SynerJ/Server	www.sun.com/forte/
Fujitsu-Siemens	BeanTransactions Application Server	www.fujitsu-siemens.com
Fujitsu Software Corporation	Interstage	www.interstage.com
Gemstone Systems，Inc.	GemStone/J	www.gemstone.com
Haht Commerce，Inc.	HAHTsite Scenario Server	www.haht.com
IBM Corporation	CICS Transaction Server	www.ibm.com
IBM Corporation	Component Broker	www.ibm.com
IBM Corporation	TXSeries	www.ibm.com
IBM Corporation	WebSphere Application Server	www.ibm.com
In-Q-My Technologies	EnterpriseBeans Server	www.inqmy.com
IONA	The IONA iPortal Server	www.iona.com
iWay Software	iWay Application Server	www.iwaysoftware.com
IPlanet	iPlanet TM Web Server Enterprise Edition 6.0	www.iplanet.com
ObjectSpace	Voyager Application Server	www.objectspace.com
Oracle Corporation	Oracle Application Server 9i	www.oracle.com
Persistence Software	PowerTier for EJB	www.persistence.com
Pramati Technologies	Pramati Server	www.pramati.com
Progress Software	Progress Apptivity Application Server	www.progress.com
Secant Technologies	Secant Extreme Enterprise Server for EJB	www.secant.com
Silverstream	SilverStream Application Server 3.0	www.silverstream.com
Sun Microsystems	NetDynamics Application Server Platform	www.netdynamics.com
Sybase，Inc.	Sybase EAServer	www.sybase.com
Unify	Unify eWave Engine 4.0 Enterprise Edition	www.unifyewave.com

（续表）

公司名称	产品名称	网址
Versata	Versata E-Business Automation System	www.versata.com
EJB 开发工具		
Allaire Corporation	JRun Studio	www.allaire.com
BEA Systems	BEA WebLogic	www.bea.com
Bluestone Software，Inc.	Total-e-Server 7.2	www.bluestone.com
Borland Corporation	Borland JBuilder	www.borland.com
Computer Associates	COOL:Joe	www.ca.com
Forte Software	ForteTM for Java	www.sun.com/forte/
IBM Corporation	VisualAge for Java	www.ibm.com
iWay Software	iWay Application Server	www.iwaysoftware.com
JFX Software	JFX Studio	www.jfxsoftware.com
ObjectSpace	Voyager Application Server	www.objectspace.com
Oracle Corporation	Oracle JDeveloper	www.oracle.com
Persistence Software	PowerTier for EJB	www.persistence.com
Pramati Technologies	Pramati Server	www.pramati.com
Silverstream	SilverStream Application Server 3.0	www.silverstream.com
Softera Ltd.	SoftModeler/Business	www.softera.com
Sybase，Inc.	Sybase Power J	www.sybase.com
TogetherSoft	Together Enterprise	www.togethersoft.com
Versant Corporation	Versant enJin	www.versant.com
WebGain	Structure Builder	www.webgain.com
WebGain	Visual Cafe	www.webgain.com
组件和应用系统		
abaXX Technology GmbH	abaXX E-Business Suite	www.abaXX.com
Digital Harbor	WorkSpace NG	www.dharbor.com
IBM Corporation	IBM SanFrancisco	www.ibm.com
iWay Software	iWay Application Server	www.iwaysoftware.com
Macadamian Technologies Inc.	Syndeo Collaboration Suite	www.macadamian.com
ObjectFX Corporation	SpatialFX Developer Suite 3.0	www.objectfx.com
Xenosys Corporation	LiveBiz JOFX Open Financial Exchange Toolkit	www.livebiz.com
数据库服务器		
IBM Corporation	DB2	www.ibm.com
Informix	Informix Dynamic Server	www.informix.com
Object Design，Inc.	Javlin EJB Data Server	www.odi.com
Oracle Corporation	Oracle 8i/9i	www.oracle.com
Versant Corporation	Versant enJin	www.versant.com

北京希望电子出版社

附录 2　JDBC Driver 一览表

条件

（1）支持 JDBC 1.x/2.x API。

（2）Type 4 类型的 Driver。

公司名	URL	支持的 DBMS
Ashna Inc.	http://www.jturbo.com/	MS SQL Server
ATINAV, INC	http://www.atinav.com/	MS SQL Server
Borland	http://www.borland.com/jbuilder/	JDataStore
Borland	http://www.borland.com/	Paradox
ChipData	http://www.chipdata.com/	DB2
Cloudscape	http://www.cloudscape.com/	Cloudscape
FrontBase	http://www.frontbase.com/	FrontBase
Frontline Software	http://www.frontbase.com/	FrontBase
Fujitsu Siemens Computers	http://www.fujitsu-siemens.de/	SESAM/SQL-Server
HiT Software, Inc.	http://www.hitsw.com/	DB2
Hitachi	http://www.hitachi.co.jp/	DABroker
HOB electronic GmbH & Co. KG	http://www.hob.de/	DB2 DL/I MS SQL Server Oracle VSAM
HOB electronic GmbH & Co. KG	http://www.hobsoft.com/	DB2 DL/I MS SQL Server Oracle VSAM
HXKJ	http://members.tripod.com/	FoxPro VFP xbase
i-net Software	http://www.inetsoftware.de/	MS SQL Server
i-net Software	http://www.inetsoftware.de	Oracle
IBM	http://www.ibm.com/iSeries/	IBM AS/400
Imaginary	http://www.imaginary.com/	mSQL
Informix Corporation	http://www.informix.com/	Informix

（续表）

公司名	URL	支持的 DBMS
jxDBCon	http://jxdbcon.sourceforge.net/	PostgreSQL
Kobu.Com	http://www.kobu.com/jdbshare/	JDBC
MERANT	http://www.merant.com/products/	DB2
MERANT	http://www.merant.com/products/	Sybase
MM.MySQL	http://www.worldserver.com/	MySQL
NetDirect	http://www.j-netdirect.com/	DB2 dBase FoxPro Informix Ingres MS Access MS SQL Server MySQL ODBC Oracle Sybase
NetDirect	http://www.j-netdirect.com/	MS SQL Server
Open Text Corporation	http://www.opentext.com/basis/	BASIS
OpenBase International	http://www.openbase.com/	OpenBase
Oracle	http://technet.oracle.com/	Oracle
Pervasive Software	http://www.pervasive.com/	Pervasive.SQL
Pointbase	http://www.pointbase.com/	PointBase
PostgreSQL Development Group	http://jdbc.postgresql.org/	PostgreSQL
Quadcap Software	http://www.quadcap.com/	Quadcap
SAP AG	http://www.sap.com/	SAP DB
SilverStream	http://www.silverstream.com/	Oracle
Software AG	http://www.softwareag.com/	ADABAS
Solid Information Technology	http://www.solidtech.com/	SOLID Embedded Engine SOLID SynchroNet
Sybase, Inc.	http://www.sybase.com/	Sybase
Uniset	http://www.uniset.ru/	InterBase MS Access

附录 3　WebLogic 服务器的配置方法

WebLogic 是 BEA 公司开发的支持 JSP/Servlet/EJB 的应用程序服务器，BEA 公司在开发 Java 应用系统方面一直领先，WebLogic 在各种支持 JSP 的服务器中也是一枝独秀。WebLogic 已经发展到了 6.0 版本。WebLogic 是一款强大的商用应用程序服务器，完全可以承担企业应用的重担。WebLogic 服务器具有以下优点：

- 支持最新的 J2EE 标准(RMI、JMS、JNDI、RMI-IIOP)，支持 XML、WAP、JOLT、WebLogic COM、JHTML、CORBA，支持 JSP 1.1、Servlet 2.2 规范，支持 EJB 1.1 规范，WebLogic 6.0 在众多服务器中率先支持 EJB2.0 规范，又一次走在各大厂商的前头。
- WebLogic 支持数据库缓冲池。
- WebLogic 有一个 Console(控制台)，管理比较方便，不好之处在于这个管理程序使用 Java 语言编写，运行起来较慢，很消耗系统资源。
- WebLogic 支持 TIME、FILE、ZAC 等服务。
- 发布/删除 EJB 服务十分方便，可以通过窗口界面可视化完成。
- WebLogic 支持 jConnect、jDriver、T3 等数据库驱动程序，可以连接 Oracle、SQL Server 等大型数据库。

下面介绍 WebLogic 5.1 的配置。

首先安装 WebLogic，安装完 WebLogic 后，WebLogic 缺省支持 Servlet，但是无法直接支持 JSP，我们需要对服务器做一些配置，WebLogic 在安装目录下有一个名为 weblogic.properties 的文件，用文本处理器把它打开，由于该文件太长了，为了节省篇幅，在这里就不把整个文件都列出来了，仅仅挑出比较重要的，或者需要修改的代码行进行介绍：

该文件记录着 WebLogic 服务器的全部配置信息，服务器在启动时将会将它读入，并按其配置服务器的各项功能，读者可以看到，该文件中很多行前面都有“#”，这表示注释，注释的行会被 WebLogic 忽略，没被注释的行将会被执行。

读者可以查找以下行和代码，了解其作用，必要的话可以对其进行相应的修改。

weblogic.system.listenPort=7001

该行表明服务器侦听的端口为 7001，读者也可以将它改为其他的端口，如 80。

weblogic.password.system=javafancy

这一行设定管理员密码，这个密码也可以在安装 WebLogic 时设定，读者在使用 WebLogic Console 控制服务器时，需要使用这个密码。

weblogic.system.enableConsole=true

该行允许管理员使用 WebLogic Console。

weblogic.system.maxLogFileSize=1024

该行设定 weblogic.log 文件的最大长度。

weblogic.system.minPasswordLen=8

该行设定管理员密码的最小长度为 8 个字符。

weblogic.httpd.session.enable=true

该行生效后，在服务端允许使用 Session，Session 可以在服务端保持客户端的信息，在开发电子商务程序时十分有用，在第 7 章中已经进行了详细介绍。

```
# MIME types
weblogic.httpd.mimeType.text/html=html，htm
weblogic.httpd.mimeType.image/gif=gif
weblogic.httpd.mimeType.image/jpeg=jpeg，jpg
weblogic.httpd.mimeType.application/pdf=pdf
weblogic.httpd.mimeType.application/zip=zip
weblogic.httpd.mimeType.application/x-java-vm=class
weblogic.httpd.mimeType.application/x-java-archive=jar
weblogic.httpd.mimeType.application/x-java-serialized-object=ser
weblogic.httpd.mimeType.application/octet-stream=exe
weblogic.httpd.mimeType.text/vnd.wap.wml=wml
weblogic.httpd.mimeType.text/vnd.wap.wmlscript=wmls
weblogic.httpd.mimeType.application/vnd.wap.wmlc=wmlc
weblogic.httpd.mimeType.application/vnd.wap.wmlscriptc=wmlsc
weblogic.httpd.mimeType.image/vnd.wap.wbmp=wbmp
```

下面的代码段列出 WebLogic 服务器支持的 MIME 类型：

weblogic.jdbc.enableLogFile=false

weblogic.jdbc.logFileName=jdbc.log

这两行代码指定一个名为 jdbc.log 的文件记录调用 JDBC 数据库驱动程序产生的事件或者错误。

weblogic.httpd.documentRoot=public_html/

该行的作用是指定 WebLogicInstallPath\myserver\public_html\目录作为应用程序发布目录，JSP 代码可以存放到该目录中去。

welogic.httpd.servlet.classpath=D:/weblogic/myserver/servletclasses

该行指定 Servlet 程序的 class 文件存放的地方，我们需要将编译好的 class 文件放置到 myserver\servletclasses 文件夹内。

```
#weblogic.httpd.register.*.jhtml=\
#        weblogic.servlet.jhtmlc.PageCompileServlet
#weblogic.httpd.initArgs.*.jhtml=\
#        pageCheckSeconds=1，\
#        packagePrefix=examples.jhtml，\
#        compileCommand=c:/java/bin/javac.exe，\
#        workingDir=C:/weblogic/myserver/classfiles，\
#        verbose=true
```

这十几行代码设定服务器对 JHTML 技术的支持，在缺省状态下是关闭的，即这些行都被注释掉，如果希望服务器支持 JHTML，必须将注释去掉，同时还需要做一些改变，关键是设定机器上安装的 JDK 的路径，请看行：

```
#          compileCommand=c:/java/bin/javac.exe，\
```

该行设定 javac 编译器的路径，所以必须将它改为机器上安装的 JDK 的 bin 目录所在的路径，如在笔者的计算机上，就把它改为：

```
compileCommand=d:/Borland/JBuilder35/jdk1.2.2/bin/javac.exe，\
```

找到代码段：

```
#weblogic.httpd.register.*.jsp=\
#          weblogic.servlet.JSPServlet
#weblogic.httpd.initArgs.*.jsp=\
#          pageCheckSeconds=1，\
#          compileCommand=c:/java/bin/javac.exe，\
#          workingDir=C:/weblogic/myserver/classfiles，\
#          verbose=true
```

这十几行代码设定服务器对 JSP 技术的支持，在缺省状态下是关闭的，即这些行都被注释掉，如果希望服务器支持 JSP，你必须将注释去掉，同时还需要做一些改变，关键是设定机器上安装的 JDK 的路径，请看行：

```
#          compileCommand=c:/java/bin/javac.exe，\
```

该行设定 javac.exe 编译器的路径，所以必须将它改为机器上安装的 JDK 的 bin 目录所在路径，如在笔者的计算机上，就把它改为：

```
compileCommand=d:/Borland/JBuilder35/jdk1.2.2/bin/javac.exe，\
```

找到代码段：

```
####################################
# USER-WRITTEN AND DEMO SERVLET REGISTRATIONS
# ------------------------------------------------------
# Set ACLs for these as desired
# ------------------------------------------------------
weblogic.httpd.register.phone=examples.servlets.PhoneServlet
# Set initArgs phonelist value to the fully-qualified
# file specification of the phonelist file
weblogic.httpd.initArgs.phone=\
          phonelist=D:/weblogic/examples/servlets/phonelist
weblogic.httpd.register.snoop=examples.servlets.SnoopServlet
weblogic.httpd.register.cookies=examples.servlets.CookieCounter
weblogic.httpd.register.error=examples.servlets.ErrorServlet
weblogic.httpd.register.applet=examples.servlets.AppletServlet
weblogic.httpd.register.helloWorld=examples.servlets.HelloWorldServlet
weblogic.httpd.register.helloKona=examples.servlets.HelloKonaServlet
weblogic.httpd.register.page=examples.servlets.PageEvent
```

```
weblogic.httpd.initArgs.page=\
        imageurl=D:/weblogic/examples/images/trans.gif
weblogic.httpd.register.session=examples.servlets.SessionServlet
weblogic.httpd.register.simple=examples.servlets.SimpleServlet
weblogic.httpd.register.simpleFormServlet=examples.servlets.SimpleFormServlet
weblogic.httpd.register.survey=examples.servlets.SurveyServlet
weblogic.httpd.initArgs.survey=\
    resultsDir=D:/weblogic/myserver
```

这些代码段的作用是注册 Servlet 程序，以及指定 Servlet 程序的初始参数。前面已经提到，Servlet 的 class 文件必须存放在 servletclasses 目录中，但是这样还不行，必须在 weblogic.properties 文件中对 Servlet 进行注册，以便 WebLogic 服务器在启动的时候将 Servlet 程序的二进制代码载入内存。例如，我们编写了一个名为 test 的 Servlet 程序，class 文件为 test.class，存放在 servletclasses 目录下，要使服务器支持该 Servlet 程序，则必须在上面的代码段后面加上一句：

```
weblogic.httpd.register.test=test
```

保存 weblogic.properties 文件，启动 WebLogic 服务器，在浏览器的地址栏中输入 http://localhost:7001/test，回车，就能看到 test Servlet 运行的效果了。

找到代码段：

```
# Administration servlet registrations
# ------------------------------------------------
# For managing your WebLogic Server with a web browser.
# Do not modify these registrations.
weblogic.httpd.register.AdminEvents=admin.AdminEvents
weblogic.httpd.register.AdminClients=admin.AdminClients
weblogic.httpd.register.AdminConnections=admin.AdminConnections
weblogic.httpd.register.AdminJDBC=admin.AdminJDBC
weblogic.httpd.register.AdminLicense=admin.AdminLicense
weblogic.httpd.register.AdminMain=admin.AdminMain
weblogic.httpd.register.AdminProps=admin.AdminProps
weblogic.httpd.register.AdminRealm=admin.AdminRealm
weblogic.httpd.register.AdminThreads=admin.AdminThreads
weblogic.httpd.register.AdminVersion=admin.AdminVersion
```

以上的代码段是指定用于管理 WebLogic 服务器的 Servlet 程序。其中 AdminMain 是主管理程序。启动服务器，在浏览器的地址栏中输入：

http://localhost:7001/AdminMain，回车，将会出现类似于图 1 的窗口。

北京希望电子出版社



图 1　AdminMain 程序的运行效果(已经连接到 AdminThreads 程序去了)

找到代码段：

```
# Modify these ACLs for these servlets to
# control access to the administration servlets.
weblogic.allow.execute.weblogic.servlet.AdminEvents=system
weblogic.allow.execute.weblogic.servlet.AdminClients=system
weblogic.allow.execute.weblogic.servlet.AdminConnections=system
weblogic.allow.execute.weblogic.servlet.AdminJDBC=system
weblogic.allow.execute.weblogic.servlet.AdminLicense=system
weblogic.allow.execute.weblogic.servlet.AdminMain=system
weblogic.allow.execute.weblogic.servlet.AdminProps=system
weblogic.allow.execute.weblogic.servlet.AdminRealm=system
weblogic.allow.execute.weblogic.servlet.AdminThreads=system
weblogic.allow.execute.weblogic.servlet.AdminVersion=system
```

以上代码段的作用是指定仅有 system 用户可以使用管理程序(Servlet)。

注意：如果你不想用 JSP 的扩展名*.jsp，而想使用别的扩展名，例如，*.hsp，可以在文件末尾加上如下的一段代码：

```
weblogic.httpd.register.*.hsp=\
weblogic.servlet.JSPServlet
weblogic.httpd.initArgs.*.hsp=\
pageCheckSeconds=1，\
#compileCommand=c:/java/bin/javac.exe，\
compileCommand=d:/Borland/JBuilder35/jdk1.2.2/bin/javac.exe，\
workingDir=C:/weblogic/myserver/classfiles，\
verbose=true
```

这样你就可以使用*.hsp 的扩展名了，而且 HSP 文件使用的是 JSP 文件的语法。

到此为止，已经把 WebLogic 服务器配置完成了，可以使用菜单选项 weblogic server 来启动服务器，当你看到如图 2 所示画面时，那么，祝贺你，服务器已经成功启动了。

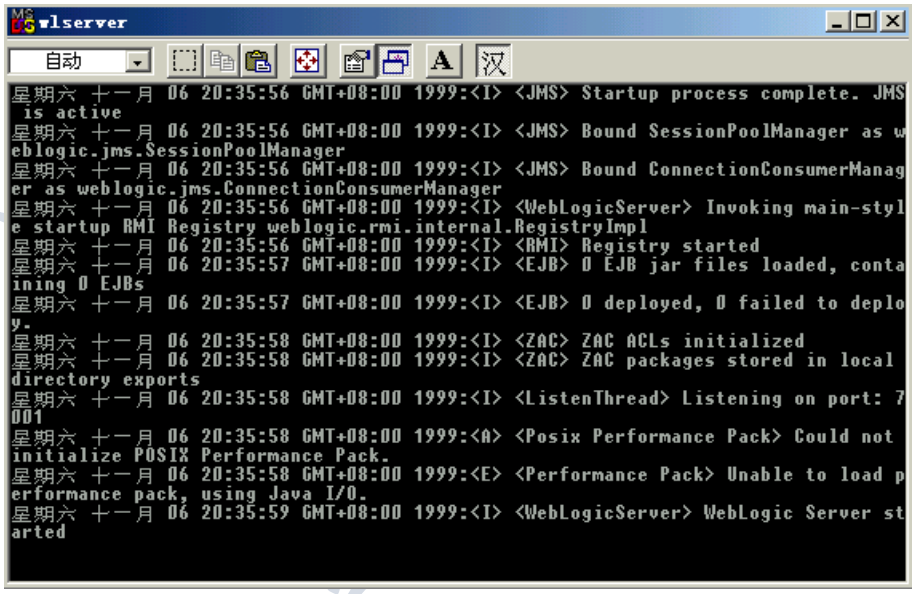


图 2　启动 WebLogic 服务器

在浏览器的地址栏中，输入 http://localhost:port/，回车，将会出现类似图 3 的窗口。



图 3　WebLogic 服务器的缺省文档

在启动了 WebLogic 服务器以后，你就可以使用 WebLogic Console 来管理服务器了，也可以使用 Console 来注册 Servlet 程序，如图 4 所示。

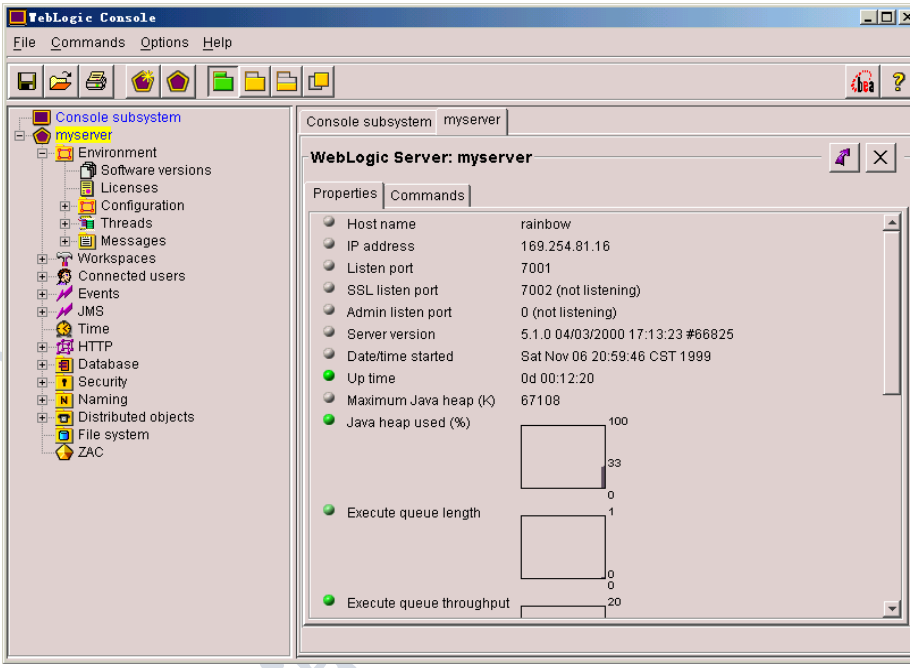


图 4　WebLogic Console 的运行界面

我们还可以使用 EJB Deployer 程序来发布 / 删除 EJB 服务，如图 5 所示。

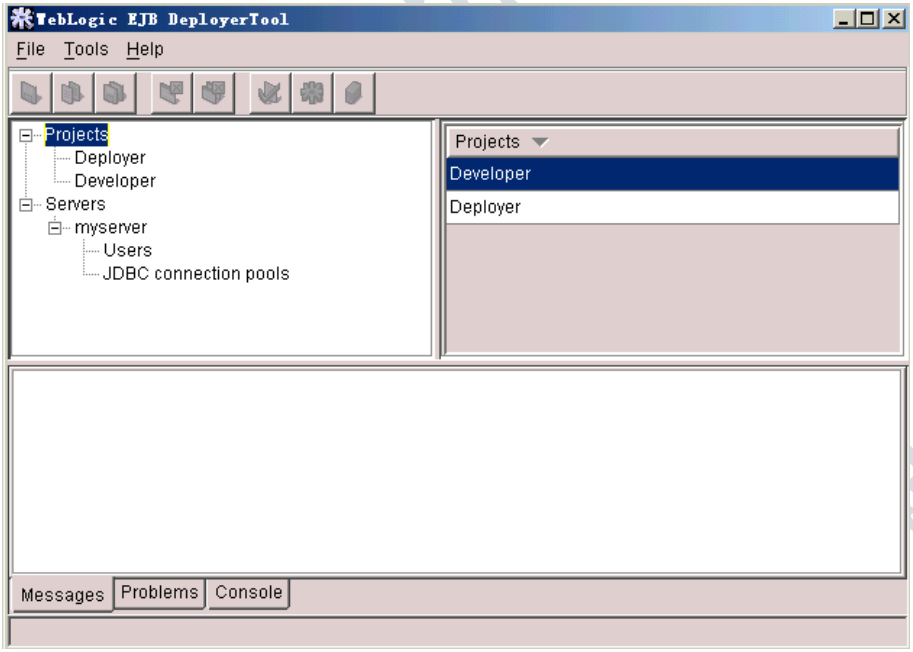


图 5　EJB Deployer 的运行效果

附录 4　本书中所用数据库的数据库结构

注意：下面的程序使用的是 T-SQL 语言的语法，该程序只能在 SQL Server 下面运行，如果希望用于其他的数据库系统，如 Access、Oracle 8i 等，需要做一定的修改。

```
--File Name:db.sql
--Author:fancy
--Date:2001.2.12
--Note:to create the database

CREATE DATABASE fancy
USE fancy

/*创建用户表tbuser */
/*id字段是自动增加的，每次增加数目为1 */

CREATE TABLE tbuser
(
      username varchar(100)，
      password varchar(100)，
      id int not null identity(1，1)
)

/*    输入9个缺省的网站用户，只有网站的用户，才可以完成购物过程 */

INSERT INTO tbuser(username，password) VALUES('fancy'，'fancy')
INSERT INTO tbuser(username，password) VALUES ('rainbow'，'rainbow')
INSERT INTO tbuser(username，password) VALUES ('lea'，'lea')
INSERT INTO tbuser(username，password) VALUES ('lijishan'，'lijishan')
INSERT INTO tbuser(username，password) VALUES ('huang'，'huang')
INSERT INTO tbuser(username，password) VALUES ('huangli'，'huangli')
INSERT INTO tbuser(username，password) VALUES ('peking'，'peking')
INSERT INTO tbuser(username，password) VALUES ('university'，'university')
INSERT INTO tbuser(username，password) VALUES ('tempuser'，'tempuser')

/*创建商品表(goods)，用于保存商品的信息 */
/* 字段priceoff 表示商品打折的幅度 */
/*字段goodstype表示商品的类型 */
/*字段 comment 表示对商品的简单介绍 */
```

北京希望电子出版社

```
CREATE TABLE goods
(
    goodsname varchar(100)，
    goodstype varchar(100)，
    comment varchar(100)，
    price float，
    priceoff float，
    id int not null identity(1，1)
)

/* 下面的代码插入二十条纪录到商品表(goods)中，进行初始化工作 */

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('JSP入门'，'书籍'，'介绍JSP技术的入门书籍，简单易学'，30.2，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('JSP精通'，'书籍'，'介绍JSP技术的高级教程'，35.0，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('JSP实例'，'书籍'，'介绍JSP编程实例的书，例子丰富'，40.0，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('JSP & XML'，'书籍'，'介绍JSP和XML的书'，38.0，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('XHTML入门'，'书籍'，'讲述XHTML的书籍'，20，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('JBuilder 4.0高级开发教程'，'书籍'，'介绍如何使用JBuilder4.0'，85，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('结构化学基础'，'书籍'，'北京大学化学系本科教材，免费赠送'，0.0，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('篮球'，'体育器材'，'可以用于锻炼身体的球型物体'，20.5，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('安踏运动鞋'，'体育器材'，'某种穿在脚上的物体'，130.1，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('玄铁剑'，'兵器'，'重剑无锋，大巧不工，神雕侠杨过昔年所用之物'，3000.01，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
```

```
VALUES ('万水青山踏遍'，'兵器'，'曾是白云城主的配剑，唯有此剑，方能使出天外飞仙的剑招
'，1300.96，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES('小楼一夜听春雨'，'兵器'，'一把弯刀，又名圆月弯刀，曾经是青青所用之物，后归丁鹏所有，和万水青山踏遍齐名'，3000.8，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('鱼肠'，'兵器'，'余夫人铸造的匕首，荆轲刺秦所用之物，失落两千年，今日重现人间
'，430.21，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('剑侠情缘2'，'游戏软件'，'金山公司推出的武侠RPG游戏，富于民族特色'，38.0，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('玉蜂浆'，'药品'，'小龙女在古墓中日常所食之物，有兹阴润肺，保容养颜之功效，绿色食品，绝对不含任何防腐剂'，354.1，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('书包'，'日常用品'，'可以用于携带东西的某种袋状物品'，15.21，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('牙膏'，'日常用品'，'用于刷牙的物品'，9.0，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('联想天禧电脑'，'计算机'，'某种型号计算机，有键盘，鼠标，显示器'，6500.9，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('笑傲江湖曲'，'CD'，'刘正风和曲洋联手创作的绝世名曲，天籁绝响，不可不听'，10.0，0.8)

INSERT INTO goods(goodsname，goodstype，comment，price，priceoff)
VALUES ('LinuxMe'，'操作系统'，'Microhard公司真情奉献，公开全部源代码，仅发行100套，世纪珍藏版'，28.7，0.8)

/*创建订单表(tborder)，用于保存订单信息 */
/*字段username表示发出订单的用户名 */
/*字段password表示用户的密码，凭此密码，用户可以修改订单*/
/*字段 ordertime表示用户发出订单的时间 */
/*字段status 表示订单的状态，如果订单没有被处理，那么status=1，如果订单已经被处理了，那么*/
/*status=2 */
/*字段 totalprice表示本订单的所有涉及商品的总金额 */
/*字段 totalprice记录了用户订单的详细信息，包括定购商品的种类、数量、打折、金额等 */
```

```
CREATE TABLE tborder
(
    username varchar(100)，
    password varchar(100)，
    ordertime datetime，
    status     varchar(100)，
    orderlist varchar(4000)，
    totalprice float，
    id int not null identity(1，1)
)

/* 下面的代码检索数据库的信息，看看有没有错误 */

SELECT * FROM tbuser
SELECT * FROM goods
SELECT * FROM tborder
```

建立好数据库以后，必须将数据库服务发布出去，读者可以使用 Windows 操作系统的 ODBC 数据源管理程序，将 fancy 数据库发布出去，数据库的别名是 test，用户名是 sa，密码为空，必须将数据库服务发布为系统数据源，不能发布为文件数据源，发布数据库服务以后，请测试数据库的连接是否有效，如果一切正常，说明数据库服务已经成功发布了，我们可以开始编写 Servlet/JSP 程序使用 JDBC 访问 fancy 数据库了。本书所使用的数据库系统是 MS SQL Server 7.0 DE 版，操作系统平台是 Windows Me，JDBC 的版本为 2.0，JDBC 数据库驱动程序一般是 JDBC——ODBC 桥。

北京邮电大学出版社

参考文献

[1] 黄理，李积善，曹林有，张勇，用 JSP 轻松开发 Web 网站，北京希望电子出版社，2001.1，第一版

[2] Sun Microsystem Inc.Java(TM) Platform 1.2 API Specification

[3] Sun Microsystem Inc.JavaServer Pages(TM) (JSP) Specification Version 1.2

[4] Sun Microsystem Inc.Java(TM) Servlet Specification v2.3

[5] Caucho Technology, Resin 1.1 Reference Guide

[6] Allaire Corporation,JRun Setup Guide

[7] Jaktara Group,Some Frequently Asked Questions (FAQ) on Tomcat

[8] BEA Systems, Inc.,Installation Guide BEA WebLogic Enterprise

[9] BEA Systems, Inc.,BEA WebLogic Server Introduction to WebLogic Server 5.1

[10] 杨光，沈建男，JSP 程序设计实务，中国青年出版社，2001，2，第一版

[11] 施汝军，网站 JSP 后台解决方案，人民邮电出版社，2000.12，第一版

[12] 姜天戬，东海大学资讯科学系，JNDI 技术简介，来源：网络

[13] Wilbur Lang，CORBA 入门，来源：网络

[14] 沈加翔 编译，CORBA 与 Java 的结合使用(学习使用 CORBA 编写访问服务器对象的分布式 Java 小应用)，来源：网络

[15] 用分布式对象技术构造 Java 应用，来源于：
http://javabar.silversand.net/technology/talkapp/c-app-003.htm

[16] 刘江宁、吴泉源、周立，几种构件模型的比较分析，来源：
http://home.online.tj.cn/~yangdy/computer/corba1.htm

[17] 分布计算环境，来源于：
http://cit.sjtu.edu.cn/Rdirectory/corba.htm

[18 施晓军，解读 COM 与 CORBA ，来源于：
http://search.ccidnet.com/Detail.wct?RecID=2&SelectID=1&ChannelID=3947&Page=1

[19] 施晓军，解读 COM 与 CORBA（下），来源于：
http://search.ccidnet.com/Detail.wct?RecID=1&SelectID=1&ChannelID=3947&Page=1

[20] 经乾、郭镇、赵伟，如何在 Java 中实现远程方法调用，来源于：
http://search.ccidnet.com/Detail.wct?RecID=8&SelectID=1&ChannelID=3947&Page=1

[21] 伊文涛，使用 JAVA 开发 CORBA 应用，来源于：网络

[22] 潘逸群，中国计算机报 1999 年第 20 期，企业计算中的 Java，来源于：
http://member.netease.com/~zzy71616/jsp/javaprg/javatech.htm

[23] www.erry.com ，CORBA 技术和 Java 技术的结合——Java IDL，来源于：
http://member.netease.com/~zzy71616/jsp/javaprg/javaidl.htm

[24] POP3 邮局协议-版本 3，来源于：网络

[25] TELNET 协议规范，来源于：网络

[26] 文件传输协议文件传输协议，来源于：网络

[27] 简单邮件传输协议，来源于：网络

[28] 小锋，Java 网络编程，来源于：网络

[29] EJB 导论 1——EJB 导论 8，来源于：ytht.net

[30] Borland，EJB Development Using JBuilder4 and Inprise Application Server 4.1 A Step-by-step Tutorial

[31] Allaire Corporation,JRun Tag Library Reference

[32] JavaSoft,Tag Libraries Tutorial

[33] Sun Microsystems Inc.,JSP Technology and XML

[34] Sun Microsystems Inc., JavaMail 1.2 API documentation

[35] Sun Microsystems Inc.,JavaMail TM API Design Specification Version 1.2

[36] Sun Microsystems Inc.,Enterprise JavaBeans TM Specification,Version 2.0

[37] Sun Microsystems Inc., JDBC ™ 3.0 Specification

[38] 新浪科技，WML 教程 1：从第一个实际应用全面了解 WML，来源于：http://www.sina.com.cn

[39] 禹希初，跟我学 XSL（一），来源于：网络

[40] 其他引用，原作者可以和本书作者联系

北京希望电子出版社