



21 世纪网站开发技术丛书 (3)



本光盘内容包括
本版电子书



中文版

JSP Programming Guide

JSP

数据库编程指南

北京希望电子出版社 总策划
布雷恩·赖特 著
赵明昌 译



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

21 世纪网站开发技术丛书 (3)

JSP Database Programming Guide

JSP 数据库编程指南

北京希望电子出版社 总策划

布雷恩·赖特 著 赵明昌 译

本书特点:

- 功能与实例相结合。对 OracleJSP 的基础知识, 编程技巧, 扩展功能等作详细介绍
- 内容新颖、丰富, 有很强的通用性、实用性和指导性

读者对象:

- 数据库开发和应用的广大从业人员自学指导书
- 高等院校相关专业师生自学、教学参考书
- 社会相关领域培训班配套教材

本光盘内容为本版电子书



北京希望电子出版社

Beijing Hope Electronic Press

www.bhp.com.cn

2001

内 容 简 介

这是一本关于 JSP 数据库编程方面的书。详细介绍了 JSP (Java Server Pages) 在大型数据库 Oracle 中应用与开发。当今社会, 信息技术飞速发展, 人们越来越依靠现代网络技术来实现各种价值, 架构自己的个人网站, 组建企业的门户网站, 进行网上营销、交流和宣传, 为满足广大读者需求, 本社特地组织了这套国外最新的网站开发技术丛书。

JSP 技术是企业应用编程中的一部分, 它基于强大的 Java 语言, 具有良好的伸缩性, 与 Java Enterprise API 紧密地集成在一起, 在开发电子商务方面具有得天独厚的优势, 基于 Java 平台构建电子商务平台已经成为当今 IT 领域新时尚。

本版书由 9 章和 3 个附录组成, 主要内容包括: JSP 简介、Oracle 的特定 JSP 实现——OracleJSP 介绍、OracleJSP 的基础知识、编程技巧以及扩展功能、OracleJSP 的解释与发布过程、JSP 标签库和 Oracle 的 JML 标签介绍、OracleJSP 对多种字符集的支持、OracleJSP 的应用范例等。三个附录内容为 OracleJSP 在多种环境下的安装与配置、Servlet 与 JSP 技术的背景知识、编译时 JML 标签支持等。

本版书内容新颖、丰富, 具有很强的技术通用性、实用性和指导性, 尤其是在数据库的二次应用开发方面, 可以说是填补了 JSP 和大型通用数据库平台之间的无缝隙的应用和开发的空白。

本版书不但是从事 JSP 和大型通用数据库平台之间应用和开发的广大编程人员的重要指导书, 同时也是高等院校师生教学与自学的优秀参考书、科研院所图书馆的馆藏读物。

本光盘内容为本版电子书。

系 列 盘 书 名: 21 世纪网站开发技术丛书 (3)

盘 书 名: JSP Database Programming Guide JSP 数据库编程指南

总 策 划: 北京希望电子出版社

文 本 著 者: 布雷恩·赖特 著 赵明昌 译

责 任 编 辑: 王玉玲

C D 制 作 者: 希望多媒体开发中心

C D 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) 010-62547735 (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

CD 生 产 者: 北京中新联光盘有限责任公司

文 本 印 刷 者: 北京媛明印刷厂

开 本 / 规 格: 787 毫米×1092 毫米 1/16 开本 22.75 印张 381 千字

版 次 / 印 次: 2001 年 6 月第 1 版 2001 年 6 月第 1 次印刷

本 版 号: ISBN 7-900071-27-X/TP·26

定 价: 46.00 元 (1CD, 含配套书)

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。

译者的话

本书专门介绍 JSP (JavaServer Pages) 在大型数据库 Oracle 中应用与开发。今天, 越来越多的人使用 JSP 技术来构建高效的电子商务系统, 开发各种中间交易系统, 创建高水平的企业网站。JSP 技术是企业应用编程中的一部分, 它基于强大的 Java 语言, 具有良好的伸缩性, 与 Java Enterprise API 紧密地集成在一起, 在开发电子商务方面具有得天独厚的优势, 基于 Java 平台构建电子商务平台已经成为当今 IT 领域新时尚。这种技术的原理是: 利用 Microsoft SQL Server 7.0、Oracle 8.0 或者 Sybase 等数据库系统作为后台数据仓库, 用 Servlet 等高性能服务器端程序作为后台总控程序, 而 JSP 程序在前台运行。Servlet 接受用户的输入, 分别调用不同的 JSP 程序向客户端反馈信息, JSP/Servlet 通过 HTTP 连接在服务器端和客户端传递数据, JSP/Servlet 并不使用 JDBC 技术直接访问数据库系统, 而是把参数传递给事先编制好的 JavaBeans 和 EJB 组件, 由它们对数据库进行操作, 这样就把系统内部的数据封装保护起来了。JavaBeans 和 EJB 组件还可以把事务分发到另一个组件中去处理, 最后把数据库返回的结果由 JSP/Servlet 送到客户端浏览器显示出来。这样的模式很容易实现分布式网络计算, 因此许多企业应用都能够作成 JavaBeans 组件, 可以重复利用, 这样既封装了某些关键的操作, 又方便了开发者, 提高了开发速度, 网站的伸缩性、安全性也得到了很好的保证。

本书由 9 章和 3 个附录组成, 主要内容包括: JSP 简介、Oracle 的特定 JSP 实现——OracleJSP 介绍、OracleJSP 的基础知识、编程技巧以及扩展功能、OracleJSP 的解释与发布过程、JSP 标签库和 Oracle 的 JML 标签介绍、OracleJSP 对多种字符集的支持、OracleJSP 的应用范例等。三个附录包括 OracleJSP 在多种环境下的安装与配置、Servlet 与 JSP 技术的背景知识、编译时 JML 标签支持等。

本书的特点是内容新, 丰富, 具有很强的技术通用性、实用性和指导性, 尤其是在数据库的二次应用开发方面, 可以说是填补了 JSP 和大型通用数据库平台之间的无缝隙的应用和开发的空白。

本书不但是从事 JSP 和大型通用数据库平台之间应用和开发的广大从业人员的重要指导书, 也是高等院校师生教学与自学的优秀参考书、科研院所图书馆的馆藏读物。

参加本书翻译工作的有赵明、尚春红、魏亚峰, 感谢方敏、刘立峰、吴晓晖、李威为本书校对, 程丽、张云、吴丽为本书录入文稿。由于译者水平有限, 书中错误在所难免, 请读者批评指正。

目 录

第 1 章 概 述

- 1.1 JSP 简介
- 1.2 JSP 的运行
- 1.3 JSP 的语法元素一瞥

第 2 章 Oracle 的 JSP 实现简介

- 2.1 跨 Servlet 环境的可移植性及功能
- 2.2 Oracle 环境对 OracleJSP 的支持
- 2.3 非 Oracle 环境对 OracleJSP 的支持
- 2.4 OracleJSP 的编程扩展概述
- 2.5 OracleJSP 的发行版本和特性小结
- 2.6 OracleJSP 运行模型
- 2.7 Oracle JDeveloper 对 OracleJSP 的支持

第 3 章 基 础 知 识

- 3.1 基本问题
- 3.2 应用程序根目录和文档根目录的功能
- 3.3 JSP 的应用程序和会话概述
- 3.4 JSP-Servlet 之间的交互
- 3.5 JSP 资源管理
- 3.6 JSP 运行时错误处理
- 3.7 JSP 实例 “Starter Sample”

第 4 章 关 键 考 虑

- 4.1 通用的 JSP 编程策略、方法和技巧
- 4.2 关键的 OracleJSP 配置问题
- 4.3 OracleJSP 运行时考虑（非 OSE）
- 4.4 Oracle Servlet Engine 环境应考虑的问题
- 4.5 Apache/Jserv Servlet 环境应考虑的问题

第 5 章 OracleJSP 的扩展功能

- 5.1 可移植的 OracleJSP 编程扩展
- 5.2 Oracle 专用的编程扩展
- 5.3 OracleJSP 对 Servlet 2.0 应用程序和会话的支持

第 6 章 JSP 的解释和发布

- 6.1 OracleJSP 解释器的功能
- 6.2 发布过程中的逻辑与特性概述
- 6.3 解释并发布到 Oracle8i 所使用的工具和命令
- 6.4 发布到 Oracle8i——服务器端解释
- 6.5 发布到 Oracle8i——客户端解释

6.6 其他的 JSP 发布问题

第 7 章 JSP 标签库和 Oracle JML 标签

7.1 标准标签库框架

7.2 JSP 标记语言 (JML) 样例标签库概述

7.3 JSP 标记语言 (JML) 标签描述

第 8 章 OracleJSP 对 NLS 的支持

8.1 页面指令中的内容类型设置

8.2 动态内容类型设置

8.3 OracleJSP 对多字节参数编码的扩展支持

第 9 章 程序实例

9.1 基本实例

9.2 JDBC 实例

9.3 数据库访问 JavaBean 实例

9.4 定制标签实例

9.5 特定 Oracle 程序扩展实例

9.6 在 Servlet 2.0 环境下运用 globals.jsa 的实例

附录 A 安装与配置

A.1 系统需求

A.2 OracleJSP 的安装和 Web Server 的配置

A.3 OracleJSP 的配置

附录 B JSP 和 Servlet 的技术背景

B.1 Servlets 背景

B.2 Web 应用程序分层

B.3 标准的 JSP 接口和方法

附录 C 编译时 JML 标签支持

C.1 JML 编译时与运行时的考虑与逻辑比较

C.2 JML 编译时的语法支持 (1.0.0.6.x 发行版)

C.3 JML 编译时的标签支持 (1.0.0.6.x 发行版)

第 1 章 概 述

本章回顾了 JSP (JavaServer Pages) 技术的标准特征和功能, 如果要获得更进一步的信息, 请参考 Sun 公司的 JSP1.1 技术标准。

(要获得 Oracle 特定的 JSP 实现——OracleJSP 的特征概述, 请参看本书第二章。附录 B 也提供了标准的 Servlet 和 JSP 技术的相关背景。)

本章包括以下几部分内容:

- JSP 简介
- JSP 的运行
- JSP 的语法元素一瞥

1.1 JSP 简介

JSP 技术由 Sun 公司提出, 利用它可以很方便地在页面中生成动态的内容, 使网络应用程序可以输出多姿多彩的动态页面。JSP 技术通常与 Java Servlet 技术相结合, 可以在 HTML 页面 (或者其他标记语言, 如 XML) 中内嵌了 Java 代码段并且调用外部 Java 组件。它作为一个前端处理工具, 可以使用 JavaBeans 和 EJBs (Enterprise JavaBeans) 完美地实现复杂的商业逻辑和动态功能。

JSP 代码与 JavaScript 等网页脚本语言是不同的, 在标准的 HTML 页面中可以出现的任何内容都可以在 JSP 页面中出现。

在一个典型的数据库应用中, JSP 页面将会调用某些 JavaBean 或者 Enterprise JavaBean 组件, 这些组件可以通过 JDBC 或者 SQLJ 直接或间接地访问数据库。

JSP 页面在运行之前要被解释成 Java Servlet, 解释过程是按需进行的, 有时可能会提前进行), 然后它可以处理 HTTP 请求并生成响应信息, JSP 技术为编写 Servlet 程序提供了更为便利的途径。

另外, JSP 页面和 Servlets 程序是可以相互操作的, 也就是说 JSP 页面可以包含从 Servlet 程序输出的内容, 可以将内容输出到 Servlet 程序, 反过来 Servlet 程序也可以包含从 JSP 页面输出的内容, 并且可以将内容输出到 JSP 页面中。

1.1.1 JSP 页面的外观

下面是一个简单的 JSP 页面例子 (里面所使用的 JSP 语法元素的解释可以参看本书相关部分)。

```
<HTML>
<HEAD><TITLE>The Welcome User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>
<B>Enter name:</B>
```

```

<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>

```

如本例所示，在一个 JSP 页面中，JSP 元素以标签<%开始，以标签%>结束。本例中，Java 代码从 HTTP request 对象中获取用户名信息，接着打印出用户名并且得到当前的日期。在此例中，比如用户输入的名字是“Amy”，那么 JSP 页面的输出如下图所示。



1.1.2 JSP 编码与 Servlet 编码，哪个更方便

在 HTML 页面中内嵌 Java 代码和 Java 调用与直接在 Servlet 程序中编写 Java 代码相比起来更为方便。JSP 语法允许你更快捷地编写动态 Web 页面内容，与 Java Servlet 语法相比需要更少的代码。

下面是一个用来对比 Servlet 代码和 JSP 代码的例子：

Servlet 代码：

```

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

public class Hello extends HttpServlet
{
    public void doGet(HttpServletRequest rq, HttpServletResponse rsp)
    {
        rsp.setContentType("text/html");
        try {
            PrintWriter out = rsp.getWriter();
            out.println("<HTML>");
            out.println("<HEAD><TITLE>Welcome</TITLE></HEAD>");
            out.println("<BODY>");
            out.println("<H3>Welcome!</H3>");
            out.println("<P>Today is "+new java.util.Date()+".</P>");
            out.println("</BODY>");
            out.println("</HTML>");
        } catch (IOException ioe)
        {
            // (error processing)
        }
    }
}

```



```
}  
}
```

（有关 `HttpServlet` 抽象类 `Response`，`HttpServletRequest` 接口和 `HttpServletResponse` 接口的背景信息，请参看本书“Servlet 接口”部分。）

JSP 代码:

```
<HTML>  
<HEAD><TITLE>Welcome</TITLE></HEAD>  
<BODY>  
<H3>Welcome!</H3>  
<P>Today is <%= new java.util.Date() %>.</P>  
</BODY>  
</HTML>
```

要知道 JSP 的语法是多么地简单，它可以省略掉传统 Java 代码中的 `imports` 语句和 `try...catch` 代码块，并且 JSP 的解释器会为你自动处理大量 servlet 编码工作，比如它直接或间接地实现了标准的 `javax.servlet.jsp.HttpJspPage` 接口并且添加必要的代码以获得一个 HTTP Session 对象。

另外，JSP 页面里面的 HTML 代码可以直接出现，而不必像 Servlet 代码那样必须嵌在 Java 的 `print` 语句里，因此你可以使用任何流行的 HTML 创作工具来创建 JSP 页面。

1.1.3 分隔商业逻辑与页面内容——JavaBeans 的调用

JSP 技术的最大优点就是可以将网页的静态内容（HTML 代码）开发与网页的动态内容（Java 代码）开发分隔开来，从而可以使得精通 HTML 但对 Java 不很精通的开发人员专门负责网页静态内容的开发，而那些对 Java 很在行但却不熟悉 HTML 的开发人员就可以专注于网页的动态内容的开发。

在一个典型的 JSP 页面中，大部分的 Java 代码和商业逻辑将不会在内嵌的代码段中，相反，它主要通过调用 JavaBeans 或 Enterprise JavaBeans 组件来实现。

JSP 提供了下面的语法来定义和创建一个 JavaBeans 对象的实例：

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

本例创建了类 `mybean.NameBean` 的一个实例：`PageBean`（`scope` 参数将会在本章的后面作出解释。），在 JSP 页面中此句的后面，你就可以如下例般使用这个实例了：

```
Hello <%= pageBean.getNewName() %> !
```

（如果 `pageBean` 的 `newName` 属性为“Julie”，那么上句输出“Hello Julie!”。）

将商业逻辑从页面内容中分隔出来将有利于更便利的分工合作，Java 专家可以专门负责商业逻辑和动态内容，比如编写和维护 `NameBean` 的代码；而 HTML 专家可以负责页面的布局和内容的表达，比如编写和维护 `.jsp` 文件的代码。这样做可以做到责任明确，提高开发的效率。

JavaBeans 使用标签 `useBean` 去声明 JavaBean 的实例对象，并且 `getProperty` 和 `setProperty` 去访问 bean 的属性，关于这方面的详细内容将在“JSP 的 Actions 和 `<jsp:>` 标签集”中讨论。

1.1.4 JSP 页面和其他标记语言

JSP 技术最典型的应用是动态 HTML 输出，但是 Sun 公司的 JSP1.1 技术标准也支持其他类型结构下的、基于文本的文档输出。由于 JSP 解释器不处理 JSP 元素以外的文本，因此任何选用于页面的文本也同样可以出现在 JSP 页面中。

JSP 页面从 HTTP request 对象得到相关信息，通过 SQL 数据库查询从数据服务器上得到相关数据，然后处理这些信息并将它们组合在一起输出到 HTTP response 对象，输出的内容可以是 HTML 格式、DHTML 格式、XHTML 格式或者 XML 格式。

1.2 JSP 的运行

本节简单地介绍了 JSP 页面的运行过程，包括按需解释（在 JSP 页面第一次运行时需要解释），JSP 容器和 Servlet 容器的作用以及出错处理。

注意：术语 JSP 容器（JSP container）出现于 Sun 公司的 JSP 1.1 技术标准中，用以替代早期标准中的术语 JSP 引擎（JSP engine），这两个术语是同义的。

1.2.1 JSP 容器概论

一个 JSP 容器就是一个程序实体，它用来解释、执行和处理 JSP 页面并且将请求信息传递给 JSP 页面。

JSP 容器的组成随着 JSP 的不同实现也不尽相同，但本质上它是由许多 servlet 和 serlets 的集合构成的，因此 JSP 容器是被 Servlet 容器所运行的。

如果 Web Server 是用 Java 语言开发的话，那么 JSP 容器可以被集成到 Web Server 中，否则此容器可以与 Web Server 建立关联并被其使用。

1.2.2 JSP 页面和按需解释

假设在一个典型的按需解释场合下，JSP 页面通常是通过下面步骤来运行的：

1. 用户通过 HTTP 请求一个后缀名是.jsp 的 URL，此 URL 连接到了一个 JSP 页面上。
2. Web Server 的 Servlet 容器发现在 URL 中有.jsp 文件扩展名，就调用 JSP 容器来进行处理。
3. 如果此页面是第一次被请求，JSP 容器要定位此 JSP 页面文件并解释它，解释的过程包括将 JSP 源文件处理成 Servlet 代码文件(.java)以及编译.java 文件生成 Servlet 的.class 文件。

JSP 解释器生成的 servlet 类是实现了 javax.servlet.jsp.HttpJspPage 接口的类（由 JSP 容器提供）的一个子类。这个 servlet 类被叫做页面实现类，本书中将页面实现类的实例称为 JSP 页面实例。

将 JSP 页面解释成 servlet 的同时会自动生成一些 servlet 代码，如 javax.servlet.jsp.HttpJspPage 的实现代码以及它的 service 方法代码，这将极大地减轻编程工作量。

4. JSP 容器运行页面实现类的实例，此时 servlet（即 JSP 页面实例）就会处理 HTTP

请求，生成对应的 HTTP 响应并传回给客户端。

注意：上面的步骤粗略地描述了 JSP 页面的运行过程，正如前面所提到过的，每个销售商都有自己的 JSP 容器实现，但是它都是由一个 servlet 或者一套 servlet 来组成的。例如，可能有一个前端的 servlet 专门用来定位于 JSP 页面文件，一个解释的 servlet 专门用来解释和编译；以及一个封装的 servlet 类，它可以被每个页面实现类所继承（因为一个解释后的页面并不是一个纯 servlet，所以不能直接由 servlet 容器来运行）。这些 servlet 要想正常运行都必须要有个 servlet 容器。

1.2.3 请求一个 JSP 页面

一个 JSP 页面可以通过两种方法来请求：要么是直接通过 URL 请求，要么是通过另一个网页或 servlet 来间接请求。

直接请求 JSP 页面

在一个 servlet 或 HTML 页面中，最终用户可以直接通过 URL 来请求一个 JSP 页面，例如，假设在 myapp 应用程序的根目录下存在 HelloWorld.JSP 页面文件，如下所示：

```
myapp/dir1/HelloWorld.jsp
```

如果 Web Server 使用端口 8080，那么可以通过下面 URL 来请求此页面：

```
http:// hostname:8080/myapp/dir1/HelloWorld.jsp
```

（应用程序的根目录是在应用程序的 servlet 环境中指定的。）

当最终用户第一次请求 HelloWorld.jsp 文件时，JSP 容器将解释并运行此页面，对以后的再次请求，JSP 容器只运行此页面，解释过程已经不再需要。

间接请求 JSP 页面

与 servlet 一样，JSP 页面也可以间接地运行，例如从一个标准的 HTML 页面连接至 JSP 页面，或者从其他的 JSP 页面、servlet 中引用另一个 JSP 页面。

当一个 JSP 页面中通过 JSP 语句来调用另外一个 JSP 页面时，路径的指定可以有两种方法：一种是相对于应用程序的路径——叫做 context-relative 或者 application-relative 路径；另外一种相对于调用页面的路径——叫做 page-relative 路径。application-relative 路径以 “/” 起始，而 page-relative 路径则没有 “/”。

应该注意到，这两种路径与在 URL 或 HTML 链接中指定的路径都不相同。接着上一小节的例子，与直接 URL 请求具有相同效果的 HTML 链接路径如下所示：

```
<a href="/myapp/dir1/HelloWorld.jsp" /a>
```

具有相同效果的使用 application-relative 路径的 JSP 语句如下：

```
<jsp:include page="/dir1/HelloWorld.jsp" flush="true" />
```

具有相同效果的使用 page-relative 路径的 JSP 语句如下：（调用和被调用的页面处于同一目录下。）

```
<jsp:forward page="HelloWorld.jsp" />
```

（“JSP 的 Actions 和 <jsp:> 标签集”中将讨论 jsp:include 和 jsp:forward 语句。）

1.3 JSP 的语法元素一瞥

在“JSP 页面的外观”中曾经有一个简单的例子，其中使用了一部分 JSP 语法。本节将进一步介绍 JSP 的各种语法元素，包括下面的内容：

- 指令语句，用来表示应将 JSP 作为一个整体来看待。
- 脚本元素，也就是 Java 代码段，用来声明变量，添加表达式、代码段以及为程序编写注释等。
- 对象和作用域，Java 的对象可以显式创建也可以隐式创建，每个对象都有一个作用域，在此作用域里可以访问这个对象的属性和方法，出了此作用域，对象就会被释放。作用域一般有页面作用域及会话作用域等。
- 动作（action），用来创建对象或者在 JSP 的响应中改变输出流，也可既创建对象又改变输出流。

上面所列主题的基本语法和例子在本节后续部分将有进一步介绍，要获得更多的信息，请参看 Sun 公司的 JSP1.1 技术标准。

注意：JSP 的指令、声明、表达式以及脚本等的，都有对应的与 XML 兼容的语法，请参看“可替换 XML 的 JSP 语法”以获得更多信息。

1.3.1 指令语句

指令语句所设置的属性是针对整个 JSP 页面的，它主要用来控制解释和运行页面时所需的参数信息。指令语句的基本语法如下：

```
<%@ directive attribute1="value1" attribute2=" value2"... %>
```

JSP 1.1 标准支持的指令语句有：

- **page**——使用此语句来设置任意与页面相关的属性，比如使用的脚本语言，需要继承的类，引入的包，错误处理页面以及页面输出缓存的大小等，并且它一次可以设置多个属性。下面是一个例子：

```
<%@ page language="java" import="packages.mypackage"
errorPage="boof.jsp" %>
```

如果要将 JSP 页面的输出缓冲大小设为 20kb（缺省值为 8kb），请参照下例：

```
<%@ page buffer="20kb" %>
```

如果在页面中不使用缓冲，请参照下例：

```
<%@ page buffer="none" %>
```

注意：

- 使用错误处理页面的 JSP 页必须被缓冲，如果出错从而转向错误处理页面时，可以清除缓冲内容，不向浏览器输出。
- 在 OracleJSP 中，java 是缺省的语言设置，但是在程序中显式地声明是个良好的编程习惯。
- **include**——使用此语句可以将另外一个包含有内容和代码的 JSP 文件插入到当前的 JSP 页面中。指定 JSP 文件的路径时必须使用相对于当前 JSP 页面的 URL 路径，

下面是一个例子：

```
<%@ include file="/jsp/userinfopage.jsp" %>
```

include 语句既可以指定相对于页面的路径，也可以指定相对于环境的路径。

注意：

- 此处所讲的 include 语句是所谓的“静态包含”，在本章后续部分还将讨论使用 jsp:include 语句的动态包含。静态包含在 JSP 页面被解释的时候生效，而动态包含在 JSP 页面被请求的时候生效。
- include 语句只能在相同的 servlet 环境下使用。
- taglib——使用此语句可以在 JSP 页面中添加定制的 JSP 标签库。一些厂商为了扩展 JSP 功能而提供他们自己的标签集，使用 taglib 语句就可以使用这些定制的标签了。Taglib 是要指定标签库的描述文件的路径以及为了使用它们而加的前缀，请看下面的例子：

```
<%@ taglib uri="/oracustomtags" prefix="oracust" %>
```

在此页面的后面，如果想使用 oracustomtags 标签库中的标签，那么就必须使用 Oracust 后缀，如下例所示（假设标签库中有一个标签叫 dbaseAccess）：

```
<oracust:dbaseAccess>
```

```
...
```

```
</oracust:dbaseAccess>
```

从这些例子中可以看到，它们使用了 XML 风格 start-tag 和 end-tag 语法。JSP 标签库和标签库描述文件将在本章后面部分介绍，并且在第 7 章中会有更详细的讨论。

1.3.2 脚本元素

JSP 脚本元素包括以下几种，它们都是在 JSP 页面中可以出现的 Java 代码段：

- 声明——用来声明 JSP 页面所使用的方法或成员变量的语句。
JSP 声明语句出现在 <%!...%> 声明标签中，它使用标准的 Java 语法来声明成员变量或方法，并且导致在生成的 servlet 代码中出现对应的声明语句。请看下面的一个例子：

```
<%! double f1=0.0; %>
```

这个例子声明了一个成员变量 f1，同时在 JSP 解释器生成的 servlet 类代码中，变量 f1 也在顶层类中被声明。

注意：相对于成员变量来说，方法变量是在 JSP 小代码段中声明的（有关小代码段，后面将对其作出进一步解释）。

- 表达式——标准的 Java 表达式，首先经过计算，然后将结果转换成字符串类型，显示在页面中它们出现的地方。JSP 表达式不以分号（;）结束，并且需要包含在 <%...%> 标签中。下面是一个实例：

```
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! </B></P>
```

注意：在请求期属性（如 jsp:setproperty 语句）中的 JSP 表达式不需要转换成字符串类型。

- 小代码段——所谓小代码段，就是 Java 代码和标记语言混合在一块形成的页面的

一部分。

在小代码段中，可以包括多种形式的 Java 代码，你可以使用一行完整的代码，也可以使用连续多行代码，甚至还可以使用不完整的半行代码，使它们与 JSP 页面中的 HTML 代码交叉出现，以完成一定的功能。

JSP 小代码段必须出现在`<%...%>`标签中，并且使用标准的 Java 语法。

例子 1:

```
<% if (pageBean.getNewName().equals("")) { %>
    I don't know you.
<% } else { %>
    Hello <%= pageBean.getNewName() %>.
<% } %>
```

在这个例子中，有三个单行的 JSP 代码与两行 HTML 代码混合出现，在第四行的 HTML 代码中还包括一个 JSP 表达式，注意到它不需要分号作为结束标记，并且 JSP 语法允许 HTML 代码插入到 if 和 else 分支语句之间，如果满足条件的话，HTML 内容才会被输出。

上面的例子中假设使用了一个 JavaBean 对象 `pageBean`。

例子 2:

```
<% if (pageBean.getNewName().equals("")) { %>
    I don't know you.
    <% empmgr.unknownemployee();
} else { %>
    Hello <%= pageBean.getNewName() %>.
    <% empmgr.knownemployee();
} %>
```

这个例子在小代码段中添加了更多 Java 代码，并且它假设有如下的前提条件：使用 JavaBean 对象 `pageBean`，事先已给定义并实例化了对象 `empmgr`；`empmgr` 对象有合适的方法去处理已知或未知的雇员信息。

注意：相对于声明成员变量来说，要使用 JSP 小代码段以声明方法变量，如下例所示：

```
<% double f2=0.0; %>
```

本例的小代码段声明了一个方法变量 `f2`，与成员变量不同的是，在 JSP 解释器所生成的 servlet 类代码中，`f2` 被声明成 service 方法的一个变量，而在前面曾经讲到过，成员变量是 servlet 类的一个变量。

要获得这两种变量之间比较的内容，请参考“方法变量声明与成员变量声明的比较。”

- 注释——即开发人员在 JSP 代码里所做的注释，它和 Java 代码里的注释是相同的。注释语句要出现在`<%...%>`标签中，与 HTML 注释不同的是，当用户查看页面的源代码时，这些注释语句是不可见的。

实例：

```
<!-- Execute the following branch if no user name is entered. --%>
```

1.3.3 JSP 对象与作用域

在本书中，术语 JSP 对象指的是在 JSP 页面中声明或访问的 Java 类的实例。JSP 对象可以是下列情况中的一种：

- 显式的——显式对象在 JSP 页面的代码中声明并创建，根据你选择的作用域设置，它可以被本页面或其他页面所访问。
 - 隐式的——隐式对象由潜在的 JSP 机制所创建，根据特定对象类型的继承作用域设置，它可以被 JSP 页面中的小代码段或表达式所访问。
- 作用域将在后面的“对象作用域”一部分详细讨论。

显式对象

显式对象通常是使用 `jsp:useBean` 语句来声明和创建的一个 `JavaBean` 实例。关于 `jsp:useBean` 语句和其他的动作语句的详细描述，请参看“JSP 的 Actions 和 `<jsp:>` 标签集”，但是本处也提供了如下的例子：

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

在这个例子中 `jsp:useBean` 语句定义一个对象 `pageBean`，它是 `mybeans` 包里名字叫“`NameBean`”类的一个实例。有关 `scope` 参数的详细讨论请参看下一节“对象作用域”。

在 Java 小代码段或声明中也可以创建对象，所用语法和在 Java 程序中创建一个 Java 类实例相同。

对象作用域

在 JSP 中声明的对象，不管它是显式对象还是隐式对象，都只能在某一特定的作用域下被访问。如果是用 `jsp:useBean` 语句创建的显式 `JavaBean` 对象，你可以通过下面所示的语法来显式地设置它的作用域：

```
scope=" scopevalue"
```

下面列举了四种可能的作用域范围：

- `scope="page"` ——对象只能在创建它的 JSP 页面中被访问。
值得注意的是，当用户为了运行一个 JSP 页面而刷新它时，所有具有页面作用域的对象都要重新生成新的实例。
- `scope="request"` ——对象可以在与创建它的 JSP 页面监听的 HTTP 请求相同的任意一个 JSP 页面中被访问。
- `scope="session"` ——对象可以在与创建它的 JSP 页面共享相同的 HTTP 会话的任意一个 JSP 页面中被访问。
- `scope="application"` ——对象可以在与创建它的 JSP 页面属于相同的网络应用程序的任意一个 JSP 页面中被访问。

隐式对象

在 JSP 页面中，有很多的隐式对象可以直接使用，它们是由 JSP 机制所自动创建的 Java 类实例，通过这些隐式对象，JSP 页面可以实现与用户的动态交互。

下面列出了可用的隐式对象，关于这些对象的方法及如何使用它们，请参看 Sun 公司

的 Java 文档，这些文档可以在如下 URL 中找到：

<http://java.sun.com/products/servlet/2.2/javadoc/index.html>

- **page**
page 对象是 JSP 页面实现类的一个实例，它在页面被解释的时候自动创建，在 JSP 页面中，page 与 this 是相同的。
- **request**
request 对象是 javax.servlet.http.HttpServletRequest 接口实现类的一个实例，它代表着一个 HTTP 请求。（javax.servlet.http.HttpServletRequest 接口扩展了 javax.servlet.ServletRequest 接口。）
- **response**
response 对象是 javax.servlet.http.HttpServletResponse 接口实现类的一个实例，它代表着一个 HTTP 响应。（javax.servlet.HttpServletResponse 接口扩展了 javax.servlet.ServletResponse 接口。）
- **pageContext**
pageContext 对象是 javax.servlet.jsp.PageContext 类的一个实例，它代表着 JSP 页面的上下文环境，用来存储和访问 JSP 页面中所具有 page 作用域的对象。
Pagecontext 对象的作用域为 page，也就是说它只能在与它关联的 JSP 页面内被访问，而不能在其他地方访问。
- **session**
session 对象是 javax.servlet.http.HttpSession 类的一个实例，它代表着一个 HTTP 会话。
- **application**
application 对象是 javax.servlet.ServletContext 类的一个实例，它代表着网络应用程序的 servlet 环境。
application 对象具有全局作用域，只要和应用程序在同一个 Java 虚拟机中，任何 JSP 页面都可以访问 application 对象。（程序员应该具有 Java 虚拟机和服务端架构的基础知识。例如，在 Oracle Servlet Engine 架构中，每个用户的应用程序都在他（或她）自己的 Java 虚拟机中运行。）
- **out**
out 对象是 javax.servlet.jsp.JspWriter 类的一个实例，它用来将页面内容写到输出流中（javax.servlet.jsp.JspWriter 类扩展了 java.io.Writer 类）。
对于一个特定的请求来说，out 对象被关联到此请求的 response 对象上。
- **config**
config 对象是 javax.servlet.ServletConfig 类的一个实例，它代表着一个 JSP 页面的 servlet 配置信息。（一般来讲，servlet 容器在初始化期间使用 ServletConfig 类的实例为 servlet 提供信息，这些信息中有一部分就是 ServletContext 的实例。）
- **exception**（仅用于 JSP 错误处理页面）
exception 对象是 java.lang.Exception 类的一个实例，它代表着 JSP 页面中没有被捕捉到的异常，这些异常被另外一个 JSP 页面抛出并且导致错误处理页面被调用。

此对象只适用于 JSP 错误处理页，所谓错误处理页就是当一个 JSP 页面抛出异常时所自动转向的页面。在错误处理页的实例中访问，提供一些关于异常的信息。

隐式对象的使用

上一小节所讨论的隐式对象是非常有用的，下面的例子介绍了如何使用 request 对象从 HTTP 请求参数中查询并显示用户名。

```
<H3> Welcome <%= request.getParameter("username") %> ! </H3>
```

1.3.4 JSP 的 Actions 和<jsp:> 标签集

当 JSP 页面运行时，JSP 的动作元素会导致某些类型的动作被触发，比如初始化一个 Java 对象并使它在此页面中可用，这样的动作元素包括下面几项：

- 创建一个 JavaBean 实例并且访问它的属性。
- 将页面转移到另外一个 HTTP 页面、JSP 页面或 servlet 中。
- 在 JSP 页面中包括一个外部的资源。

动作元素使用一套标准的 JSP 标签，它以<jsp: 做为开始标记。尽管使用在本章的前面所讨论的以<%语法开始的标签已经足够用来编写 JSP 页面，但是<jsp:标签仍然提供了更强大的功能和更方便的用法。

动作元素使用的语法和 XML 语句所使用的语法是相似的，都有开始和结束标签，如下例所示：

```
<jsp:sampletag attr1=" value1" attr2=" value2" ... attrN=" valueN">
... body...
</jsp:sampletag>
```

如果没有正文的话，动作元素以空标签来作为结束标记，如下例所示：

```
<jsp:sampletag attr1=" value1", ..., attrN=" valueN" />
```

JSP 标准包括下列标准的动作标签，它们在这儿只是被简单介绍和讨论了一下：

- **jsp:useBean**

jsp:useBean 动作创建一个指定 JavaBean 类的实例，并且给实例指定一个名字，定义它的作用域。

例子：

```
<jsp:useBean id="pageBean" class="mybeans.NameBean" scope="page" />
```

这个例子创建了类 mybeans.NameBean 的一个实例的属性（这些实例必须已经用 useBean 语句定义过）。

- **jsp:setProperty**

jsp:setProperty 动作设置一个或多个 bean 属性（bean 必须是已在 UseBean 动作中指定的）。你可以直接指定某一个属性的值，或者从一个相关联 HTTP 请求中得到一个参数值赋给某一个属性，甚至还可以循环查询 HTTP 请求参数的值，把它们分别赋给多个属性。

下面的例子将 pageBean 实例（在前面的 useBean 例子中定义）的 user 属性设成“Smith”：

```
<jsp:setProperty name="pageBean" property="user" value="Smith" />
```

下面的例子根据 HTTP 请求中名字叫 username 的参数的值来设置 pageBean 的 user 属性:

```
<jsp:setProperty name="pageBean" property="user" param="username" />
```

或者, 如果 HTTP 请求的参数名字和 pageBean 的属性名字相同 (都为 user), 那么可以如下简单地设置 user 的属性:

```
<jsp:setProperty name="pageBean" property="user" />
```

下面的例子将导致循环查询 HTTP 请求参数, 如果发现 HTTP 请求参数中有和 pageBean 的属性名字相同的, 就将 pageBean 的那个属性值设为 HTTP 请求参数中对应的那个值:

```
<jsp:setProperty name="pageBean" property="*" />
```

- **jsp:getProperty**

jsp:getProperty 语句读出一个 JavaBean 实例的属性值, 将它转换为 Java 字符串, 并且将字符串的值放入输出缓冲中以便它能被显示出来。(此语句所谈的 JavaBean 字符串必需已经用 jsp:useBean 语句定义过。)为了使字符串转换能正确进行, 简单类型 (如 int、char 等) 可以直接转换, 而对象类型则必须调用在 java.lang.Object 类中指定的 toString() 方法来转换。

下面的例子将 pageBean 的 user 属性的值填入输出对象中:

```
<jsp:getProperty name="pageBean" property="user" />
```

- **jsp:param**

jsp:param 语句一般和 jsp:include, jsp:forward 或 jsp:plugin 语句联合使用, 关于 jsp:include, jsp:forward, jsp:plugin 语句的用法将在稍后介绍。

对于 jsp:forward 和 jsp:include 语句来说, jsp:param 语句提供了名字/值这一可选语法来设置 HTTP 请求对象的参数值, 如果指定了新的参数和值, 它们将被添加到 HTTP 请求对象中, 并且新的值将覆盖掉旧的值。

下面的例子将 HTTP 请求对象中的 username 参数的值设为 “Smith”:

```
<jsp:param name="username" value="Smith" />
```

注意: 在 JSP 1.0 标准中, jsp:include 或者 jsp:forward 不支持 jsp:param 标签。

- **jsp:include**

jsp:include 语句将外部的静态或者动态资源插入到页面中, 当此页面被请求显示时, 插入的外部资源也将被显示出来。在指定资源文件时应该使用相对 URI 路径。

(要么使用页面相对路径, 要么使用应用程序相对路径。)

在 Sun 公司的 JSP 1.1 标准中规定, 使用 jsp:include 语句时必须将 flush 属性设为 true, 从而使得当 jsp:include 动作被执行时, 缓冲区里的输出马上被送到浏览器里 (在当前的 JSP 实现里, 将 flush 属性设成 false 是无效的)。

你也可以在 jsp:include 的正文区用 jsp:param 来设定一些参数值, 如下面两个例子所示:

例子一:

```
<jsp:include page="/templates/userinfopage.jsp" flush="true" />
```

例子二:

```
<jsp:include page="/templates/userinfopage.jsp" flush="true" >
  <jsp:param name="username" value="Smith" />
  <jsp:param name="userempno" value="9876" />
</jsp:include>
```

注意，例子二中的功能也可以用下例所示的语法来实现：

```
<jsp:include page="/templates/userinfopage.jsp?username=Smith&userempno=9876" flush="true" />
```

注意：

- `jsp:include` 动作语句是动态包含的，它在实质上与本章前面所讨论的 `include` 指令语句是相同的，但不同的是 `jsp:include` 在请求期生效，而 `include` 则是在解释期生效。
- `jsp:include` 动作只能在同一个 `servlet` 环境中使用。
- `jsp:forward`
`jsp:forward` 语句将当前页面的执行过程终止，忽略它的输出，并且转向一个新的页面——要么是一个 HTML 页面，一个 JSP 页面或者一个 `servlet`。
- 使用 `jsp:forward` 语句时，JSP 页面必须使用缓冲机制（不能将 `buffer` 属性设为“none”），它将清空缓存里的内容（浏览器里将得不到任何输出。）。
- 和 `jsp:include` 语句一样，你也能在 `jsp:forward` 的正文区用 `jsp:param` 来设定参数值，如下面两个例子所示：

例子一：

```
<jsp:forward page="/templates/userinfopage.jsp" />
```

例子二：

```
<jsp:forward page="/templates/userinfopage.jsp" >
  <jsp:param name="username" value="Smith" />
  <jsp:param name="userempno" value="9876" />
</jsp:forward>
```

注意：

- 此处的 `jsp:forward` 例子与上面的 `jsp:include` 例子的不同之处在于：`jsp:include` 例子是将 `useinfopage.jsp` 的内容插入到当前页面的输出中，而 `jsp:forward` 例子则停止当前页面的处理，转向显示 `userinfopage.jsp` 页面。
- `jsp:forward` 语句只能在同一个 `servlet` 环境中使用。
- `jsp:plugin`
`jsp:plugin` 语句将导致在客户端浏览器中运行一个指定的 `applet` 或者 `JavaBean`，并且如果在必要的时候还需要下载一个 Java 插件。
如果要指定待运行的 `applet` 的配置信息，可以使用 `jsp:plugin` 属性。JSP 容器一般会提供一个缺省的下载 URL，但是你也可以通过指定属性 `nspluginurl` = “url”（对 Netscape 浏览器）或者 `ieplugin` = “url”（对于 Internet Explorer 浏览器）来提供一个下载插件的 URL。
在 `<jsp:param>` 开始标签和 `</jsp:params>` 结束标签中使用嵌套的 `jsp:param` 语句来指定待运行的 `applet` 或 `JavaBean` 的参数。（请注意在 `jsp:include` 和 `jsp:forward` 语句中

使用 `jsp:param` 时，并不需要这些 `jsp:params` 开始、结束标签。)

当指定的 `plugin` 不能运行时，可以使用 `<jsp:fallback>` 和 `</jsp:fallback>` 起始和结束标签来显示一些提示信息。

下面的例子是从 Sun 公司的 JSP 1.1 标准上摘录的，它解释了如何使用一个 applet 插件：

```
<jsp:plugin type=applet code="Molecule.class" codebase="/html" >
  <jsp:params>
    <jsp:param name="molecule" value="molecules/benzene.mol" />
  </jsp:params>
  <jsp:fallback>
    <p> Unable to start the plugin. </p>
  </jsp:fallback>
</jsp:plugin>
```

在 `jsp:plugin` 语句中还允许指定很多的额外参数，如 `ARCHIVE`，`HEIGHT`，`NAME`，`TITLE` 和 `WIDTH` 等，这些参数的用法请参考相关的文档。

1.3.5 标签库

除了本节前面所讨论的标准的 JSP 标签以外，JSP 标准还允许开发厂商它们自己的标签库，并且允许开发厂商实现一个框架，从而使得普通用户也能定义他们自己的标签库。

标签库定义了一套定制的标签，你可以把它想像为 JSP 的一个子语言。开发者在手工编写 JSP 页面时可以直接使用标签库，但是 Java 开发工具也能自动地调用这些标签库。一个标准的标签库必须能在两个不同的 JSP 容器实现之间进行移植。

要将一个标签库导入到 JSP 页面中需要使用 `taglib` 指令，请参看“指令语句”。

支持 JSP 标签库的标准的 JavaServer Pages 包括以下几个关键概念：

- 标签处理器

标签处理器描述了使用定制标签所导致的动作的意义，它是实现了标准的 `javax.servlet.jsp.tagext.package` 中 `Tag` 或 `BodyTag` 接口(取决于这个定制标签是否在开始标签和结束标签中使用正文)的 Java 类的一个实例。

- 脚本变量

定制标签通过它自身或者别的脚本元素(如小代码段)可以创建服务器端对象以供使用，这是由创建或更新脚本变量来实现的。

一个标签所定义的脚本变量的细节必须在标准的 `javax.servlet.jsp.tagex.TagExtraInfo` 抽象类的子类中指定，本书将这样的子类叫做 `tag-extra-info` 类。

JSP 容器在解释页面时要使用这些类的实例。

- 标签库描述文件

标签库描述(TLD)文件是一个 XML 文档，它包含着标签库及库内每一个定制标签的信息，标签库描述文件的文件名具有 `.tld` 的扩展名。

JSP 容器在遇到标签库内的定制标签时，需要使用标签库描述文件来决定如何处理。

- 标签库的 web.xml 文件用法

Sun 公司的 Java Servlet 标准（版本 2.2）描述了 servlets 的一个标准的发行描述器——web.xml 文件。JSP 应用程序可以使用这个文件来定位 JSP 标签库。

对 JSP 标签库来说，web.xml 文件能包含一个 taglib 元素和两个子元素：taglib-uri 和 taglib-location。

要得到关于这些主题的信息，请参看“标准标签库框架”。

要得到 OracleJSP 所提供的样例标签库的信息，请参看“JSP 标记语言（JML）样例标签库概述”。

要得到更进一步的信息，请参看 Sun 公司所提供的 JSP 1.1 技术标准。

第 2 章 Oracle 的 JSP 实现简介

在 Oracle8i 的 8.1.7 发行版 (OracleJSP 1.1.0.0.0) 中, Oracle 完整实现了符合 Sun 公司的 JSP 1.1 标准的 JavaServer Pages。

本章介绍了 OracleJSP 的特性, 并且讨论了多种环境对 OracleJSP 的支持, 尤其是在 Oracle Servlet Engine (OSE) 环境下。所谓 OSE, 即是 Oracle8i JServer 所提供的的一个 Web Server (一种服务器软件, 而不是硬件), 更精确地说, OSE 就是一个 servlet 容器。

要获得标准的 JSP 特性的介绍, 请参看第一章“概述”。

本章包含以下几个主题:

- 跨 Servlet 环境的可移植性及功能
- Oracle 环境对 OracleJSP 的支持
- 非 Oracle 环境 OracleJSP 的支持
- OracleJSP 的编程扩展概述
- OracleJSP 发行版本和特性小结
- OracleJSP 的运行模型
- OracleJSP 的 Oracle JDeveloper 支持

2.1 跨 Servlet 环境的可移植性及功能

Oracle 的 JSP 实现是可以高度跨服务器平台和 servlet 环境移植的, 它也为旧的 servlet 环境下的网络应用程序提供了一个框架。

2.1.1 OracleJSP 的移植性

OracleJSP 能够在符合 Sun 公司的 Java Servlet 标准 2.0 或更高版本的 Servlet 环境中运行, 而大部分的 JSP 实现只能在 Servlet 2.1(b)或更高的版本中运行。正如在下一节中所解释的那样, OracleJSP 为旧的 Servlet 环境所不具备的功能都提供了相应等价的功能。

并且, OracleJSP 容器是不依赖于服务器环境和它的 servlet 实现的, 但是大部分的开发厂商则将他们的 JSP 实现作为他们的 servlet 实现的一部分, 而不是使 JSP 作为一个独立的产品。

OracleJSP 的这些可移植性上的优点使它更容易地在你的开发环境和目标环境下都能正常运行。相对于 OracleJSP, 其他一些 JSP 实现则由于服务器或 servlet 平台的限制而不得不在开发环境和目标平台中使用两套不同的 JSP 实现。通常情况下, 开发平台和目标平台使用同一个 JSP 容器是最好的选择, 开发过程将从中受益, 因为在实际情况中, 在不同的环境中总是存在一些差异, 所以使用不同的 JSP 容器将很可能导致问题的出现。

2.1.2 OracleJSP 对 Servlet 2.0 环境的扩展

Servlet 标准和 JSP 功能之间的互相依赖性, 导致 Sun 公司的 JSP 标准与 Java Servlet

标准的版本号也牢牢系在一起。按照 Sun 公司的说法, JSP 1.0 需要 Servlet 2.1(b)实现, 而 JSP 1.1 需要 Servlet 2.2 实现。

Servlet 2.0 标准局限于它只能为每个 Java 虚拟机提供一个 Servlet 环境, 而不是每个应用程序都拥有一个 Servlet 环境; Servlet 2.1 标准允许(但不能控制)每个应用程序有一个单独的 Servlet 环境; Servlet 2.1(b)和 Servlet 2.2 标准可以控制单独的 Servlet 环境。(要得到 Servlets 和 Servlet 环境的有关信息, 请参看“Servlets 背景”。)

OracleJSP 容器提供了扩展的功能去模拟 Servlet 2.1(b)标准所提供的应用程序支持, 从而允许在诸如 Apache/JSer 这样的 Servlet 2.0 环境中提供一个完整的应用程序框架, 包括为不同的应用程序提供不同的 ServletContext 和 HttpSession 对象。

OracleJSP 是通过一个文件——globals.jsa 来提供这些扩展的支持的。Globals.jsa 文件可以被认为是 JSP 应用程序标记器, 应用程序和会话的事件处理器以及应用程序的全局变量和指令的声明处。

正因为具有这些扩展功能, OracleJSP 可以不受限于某一个特定的 servlet 环境。

2.2 Oracle 环境对 OracleJSP 的支持

本节简单概述一下提供并支持 OracleJSP 的 Oracle 环境, 包括以下几部分内容:

- Oracle Servlet Engine (OSE)概述
- Oracle Internet Application Server 概述
- Oracle HTTP Server 所扮演的角色
- Oracle 网络应用程序的数据库访问策略
- 其他 OracleJSP 环境一览

Oracle Servlet Engine (OSE)作为一个 Web Servlet 和 Servlet 容器, 需要运行在 Oracle8i 数据库中, 它支持 JSP 预解释模型, 也就是说 JSP 页面在发布浏览数据库之前或者正在发布到数据库的过程中就被解释成 Servlet 类, 以后将运行在数据库的地址空间中。

对其他的 Oracle 环境来说, OracleJSP 容器支持典型的按需解释模型, 也就是说 JSP 页面在运行的时候被解释。OracleJSP 的设计目标就是在上面所讲两种模型中都能高效地运行, 并且无论你选择的 Web Servlet 是什么, 它都提供统一的语法。

2.2.1 Oracle Servlet Engine (OSE)概述

如果你的 JSP 页面被设计用来访问 Oracle8i 数据库, 那么你就能直接在数据库里运行它们, 因为 Oracle Servlet Engine (OSE) (它在 Oracle8i Jserver 中提供) 和 OracleJSP 容器协作使得 JSP 页面可以被预解释到数据库地址空间中去。与 JSP 页面在中间层运行的技术相比, 它减少了中间层与数据服务器之间的通信压力。在 OSE 环境下访问数据库是通过服务器端内置的 Oracle JDBC 驱动来完成的。

OSE 运行模型需要开发者执行一些特殊的步骤以将 JSP 页面发布到 Oracle8i 数据库中, 包括解释页面, 将它们加载到服务器端, 并且发行它们以使它们可用来被运行。

在 Oracle 8.1.7 发行版的安装过程中, Oracle HTTP Servlet 被设置为默认的 Web Server, 作为 JSP 和 servlet 应用程序在 OSE 环境下运行的前端解释工具。如果你想改变这个缺省的

设置，请参看相关的安装说明文档。

Oracle Servlet Engine 8.1.7 发行版支持 Servlet 2.2 和 JSP1.1 标准，并且提供 OracleJSP 8.1.7 发行版（1.1.0.0.0）。

2.2.2 Oracle Internet Application Server 概述

Oracle Internet Application Server 是一个灵活、安全的中间层应用程序服务器。它可以用来投递 Web 内容，主管 Web 应用程序，连接到 back-office 应用程序，并且使这些服务器对任何客户端浏览器都可以访问。用户可以在 Internet 上访问信息、执行商业分析以及交流合作。

为了实现这些内容和服务，Oracle Internet Application Server 1.0.x 发行版合并了下面一些组件：Oracle HTTP Server 1.0.0 发行版，用来处理数据库缓冲和中间层应用程序的 iCach，支持 Oracle 基于表单的应用程序和报表生成的 Oracle Forms Services 和 Oracle Reports Services，支持 Enterprise JavaBeans 和存储过程多种商业逻辑运行环境以及 Oracle Bussiness Components for Java。

对数据库访问来说，Oracle HTTP Server 能将 HTTP 请求传递给 servlets 或 JSP 页面，并且它们能运行在以下场景中：

- 直接在 Oracle8i 数据库中（请求传递是通过 Apache 的 mod_ose 模块来完成的）。在此种场景下，数据库访问是通过服务器端 JDBC 驱动（需要使用 JDBC 代码，要么使用 SQLJ 代码）来实现的。
- 在 Apache/JServ 环境（请求参数是通过 Apache 的 mod_jserv 模块来实现的）。在此种场景下，数据库访问是通过客户端/中间层的 JDBC 驱动（需要使用 JDBC 代码，要么使用 SQLJ 代码）来实现的。

Oracle Internet Application Server 1.0.x 发行版提供的 servlet 和 JSP 环境如下：

- 在 1.0.0 和 1.0.1 发行版中都提供了 Apache/JServ 环境，支持 servlet 2.0 标准。
- 在 1.0.0 发行版提供了 OracleJSP 1.0.0.6.1 发行版，支持 JSP1.0 标准。
- 在 1.0.1 发行版提供了 OracleJSP 1.0.0.0.0 发行版，支持 JSP1.1 标准。

要得到关于 Oracle Internet Application Server 更多的信息，请参看它的相关文档库。

注意：在 Oracle Internet Application Server 将来的发行版本中，将有可能用其他的 Servlet 环境来取代 Apache/JServ 环境。

2.2.3 Oracle HTTP Server 所扮演的角色

建立在 Apache Web Server 基础之上的 Oracle HTTP Server 1.0.0 发行版是由 Oracle Internet Application Server 1.0.x 发行版或 Oracle 8.1.7 发行版提供的，它是网络应用程序访问 Oracle8i 数据库的 HTTP 入口点。

不管是运行在 Oracle8i 数据库环境里面还是外面，应用程序都可以通过 Oracle HTTP Server 来访问数据库。Oracle HTTP Server 对数据库的访问是通过合适的 Apache add-on 模块来完成的。

本节的剩余部分将涵盖以下主题：

- Apache Mods 的使用
- 关于 mod_ose 的更多信息
- 关于 mod_jserv 的更多信息

注意:

- 尽管 Oracle Servlet Engine 自身具有 Web Server 的功能,但是建议最好将它作为一个 Servlet 容器与 Oracle HTTP Server 联合使用,尤其是在包含有静态 HTML 的应用程序中。使用 mod_ose 模块可以通过 Oracle HTTP Server 来访问 OSE。
- 在 Oracle Internet Application Server 环境中,你可以通过 Oracle HTTP Server 来访问 Oracle iCache(适用于被缓冲的只读数据),或者访问后台 Oracle8i 数据库。但是在 Internet Application Server 1.0.0 和 1.0.1 发行版中,你不能使用 mod_ose/OSE 模块,因为在这些发行版中的 iCache 不包含 Oracle Servlet Engine。

Apache Mods 的使用

在 Oracle HTTP Server 中,动态的内容是通过各种 Apache mod 组件来实现的,这些 mod 组件可以是 Apache 提供的,也可以是诸如 Oracle 这样的开发厂商所提供的。(一般来说,静态的内容是由文件系统来实现的。)

一个 Apache mod 其实就是一个典型的 C 代码模块,它在 Apache 的地址空间中运行,并且将请求信息传递给特定的由 mod 指定的处理器。(对 mod 软件来说,必须针对特定的处理器来写相应的代码。)

OracleJSP 开发者所关心的 Apache Mods 如下所列:

- mod_ose, 由 Oracle 提供,它将需处理的 JSP 页面和 Servlets 先发布到 Oracle8i 数据库中,然后由 Oracle Servlet Engine 在数据库地址空间中来运行它们。
- mod_jserv, 由 Apache 提供,使用它可以使运行在 Apache/Jserv 环境下的 JSP 页面和 Sevlets 在中间层的 Java 虚拟机中访问 Oracle8i 数据库。

注意: 在 Apache 环境下还有很多别的 Apache “mod” 组件可供使用,它们或者是由 Apache 提供作为通用的组件,或者是由 Oracle 提供用来针对 Oracle 特定的功能,但是它们跟 JSP 应用程序的开发都是不相关的。

关于 mod_ose 的更多信息

由 Oracle 所提供的 mod_ose 组件用来将 HTTP 请求信息传递给运行在 OSE 环境下的 JSP 页面或 servlets。它通过 Net8 协议使用 HTTP 与 OSE 进行通信,并且可以处理无状态的或全状态的请求。在 Oracle HTTP Server 中配置的每个虚拟域都和一个数据库连接字符串(Net8 名字-值)相关连,用来指示在哪里创建连接来处理请求信息。连接直接使用 Net8 协议,提供了与 OCI 相同的负载均衡和热备份的功能。

如果在 Oracle Internet Application Server 环境下运行的应用程序使用了 mod_ose,那么 Internet Application Server 的 Apache/Jserv 所使用的 Servlet 2.0 环境就不再可用了,反之要

使用 Oracle Servlet Engine 自己的 Servlet 2.2 环境。

为了快速访问数据库，在 OSE 环境下运行的 JSP 应用程序和 Servlets 使用服务器端内部的 Oracle JDBC 驱动。要获得 OSE 的概述信息，请参看“Oracle Servlet Engine (OSE)概述”。

你可以使用 JServer 的 shell 命令 `exportwebdomain` 来配置 `mod_ose` 去查找已经发布到数据库里的 servlet 和 JSP 页面。

要获得关于 `mod_ose` 的更深一步的信息以及 `exportwebdomain` 命令的用法信息，请参看 Oracle8i 所提供的“Oracle Servlet Engine 用户指南”。

关于 `mod_jserv` 的更多信息

由 Apache 所提供的 `mod_jserv` 组件用来将 HTTP 请求信息传递运行在中间层 Java 虚拟机里 Apache/Jserv 环境下的 JSP 页面或 Servlets。Oracle Internet Application Server 1.0.x 发行版中包含了 Apache/Jserv Servlet 容器，它支持 Servlet 2.0 标准，以及 JDK 1.1.8 或 JDK1.2.2。中间层环境可以与后台 Oracle8i 数据库共存于一台物理主机，也可以分存于两台物理主机上。

`Mod_jserv` 和中间层 Java 虚拟机通过 Apache/JServ 的基于 TCP/IP 的协议来进行通信，并且通过缓冲池 `mod_jserv` 组件可以同时将 HTTPs 请求传递给多个 Java 虚拟机并提供负载均衡机制。

运行在中间层 Java 虚拟机中的 JSP 应用程序使用 Oracle JDBC OCI 驱动或者 Thin 驱动来访问数据库。

Servlet 2.0 环境（相对于 Servlet 2.1 或 2.2 环境）有一些需要特殊对待的问题，请参看“Apache/JServ 环境应考虑的问题”。

要获得 `mod_jserv` 的配置信息，请参看 Apache 文档。（此文档在 Oracle8i 和 Oracle Internet Application Server 中都有提供。）

2.2.4 Oracle 网络应用程序的数据库访问策略

JSP 应用程序的开发者如果要使用 Oracle8i 数据库作为后台数据库的话，那么可以有下面所列的几种选择：

- 1、通过 Oracle HTTP Server 在 Apache/Jserv Servlet 容器下运行（使用 `mod_jserv`）。
- 2、通过 Oracle HTTP Server 在 Oracle Servlet Engine 下运行（使用 `mod_ose`）。
- 3、直接将 OSE 作为 Web Server 并在其下运行（不过一般建议使用 Oracle HTTP Server）。

注意：当你使用 Oracle HTTP Server 的时候，应该意识到 Apache/JServ 环境与 Oracle Servlet Engine 环境有不同的缺省文档根目录（对静态文件）。要获得相关信息，请参看“Oracle Internet Application Server 与 Oracle Servlet Engine 的文档根目录之比较”。

如果你想使用 JDBC OCI 驱动器或者应用程序需要 Oracle8i JServer 环境所没有提供的 Java 特性（比如 JNI）时，就必须在 Apache/Jserv 环境下运行，因为它使用了标准的 Java

虚拟机（当前版本为 JDK1.2.2 或 1.1.8）。

然而，在 Apache/JServ 环境下运行也有缺点，多重 Java 虚拟机所需要的缓冲池机制必须用手工配置。（要获得更多的信息，请参看 Oracle8i 或 Oracle Internet Application Server 所提供的 Apache mod_jserv 文档。）

如果你不需要诸如 JNI 这样的 Java 特性，特别是你的应用程序使用了大量的 SQL 语句时，在 OSE 环境下运行通常是更好的，原因如下：

- OSE 提供了更快的数据库访问速度，因为它在 Oracle8i 数据库地址空间里运行，并且使用服务器端的内容 Oracle JDBC 驱动。
- OSE 提供了更好的安全性。
- OSE 为状态管理提供了更强的支持。

尽管将 Oracle Servlet Engine 直接作为 Web Server 是可行的，并且在许多情况下还会带来益处，但是 Oracle 仍然建议 OSE 与 Oracle HTTP Server 联合使用，应用程序通过 Oracle HTTP Server 和 mod_ose 来访问 OSE。

并且，Oracle HTTP Server 和 mod_ose 能处理下面这些 OSE 自身所不能处理的情况：

- 通过 Net8 认证的防火墙来进行数据库访问。
- 实现一个使用多重数据库的容错系统。
- 通过端口 80 来访问数据库。

这是一种最典型的直接使用 OSE 作为 Web Server 所办不到的情况，因为在 UNIX 环境下端口 80 只能由具有 root 权力的用户来访问，但是最终用户并没有 root 的权力。

- 对无状态的应用程序提供连接池，以使大多数情况下能够避免会话启动时的过载。

Oracle8i 8.1.7 发行版的缺省安装将 Oracle HTTP Server 作为运行在 OSE 环境下的 JSP 页面和 Servlets 的前端 Web Server。

2.2.5 其他 Oracle JSP 环境一览

除了 Oracle Servlet Engine 和 Oracle Internet Application Server 之外，下面的 Oracle 环境也支持 OracleJSP：

- Oracle Application Server
- Oracle Web-to-go
- Oracle JDeveloper

Oracle Application Server

Oracle Application Server (OAS) 是一个灵活的、标准的中间层环境，它适用于应用程序逻辑，提供数据库集成，支持合作式和电子商务式的企业应用程序开发。

新的用户将更倾向于使用 Oracle Internet Application Server 来代替 OAS，但是对现有的 OAS 用户来说，Oracle Application Server 4.0.8.2 发行版也提供了 Servlet 2.1 环境和 OracleJSP 1.0.0.6.0 发行版（支持 JSP1.0 标准）。

要获得更多信息，请参看“Oracle Application Server 开发者指南：Jservlet 和 JSP 应用程序”。

Oracle Web-to-go

Oracle Web-to-go 是 Oracle8i Lite 版的一个组件，它由一系列模块和服务构成，用来更便利地开发、发布和管理移动网络应用程序。

Web-to-go 可以让开发者轻松地将基于网络的应用程序扩展到移动网络应用程序，并且不需处理与设备相关的复制，同步以及其他网络问题。与传统的依赖于某一定制或私有 API 的移动计算技术不同，Web-to-go 使用了作为工业标准的 Internet 技术。

Web-to-go 1.3 发行版提供了 Servlet 2.1 环境和 OracleJSP 1.0.0.6.1 发行版（支持 JSP1.0 标准），未来的发行版本中将包括 Servlet 2.2 环境和 OracleJSP 1.1.x。

要获得更多的信息，请参看“Oracle Web-to-go 实现指南”。

Oracle JDeveloper

JDeveloper 是一个 Java 开发工具，而不像前面所讨论的其他 Oracle 产品那样都是平台。它包含了一个网络监听器，一个 Servlet 运行器，还有一个 OracleJSP 容器用来运行和测试 JSP 应用程序。

要获得更多的信息，请参看“Oracle JDeveloper 对 OracleJSP 的支持”。

JDeveloper 3.1 版本提供了 Servlet 2.1 环境和 OracleJSP 1.0.0.6.1 发行版（支持 JSP 1.0 标准），未来发行版中将提供 Servlet 2.2 环境和 OracleJSP 1.1.x。

2.3 非 Oracle 环境对 OracleJSP 的支持

你能够在任何支持 Servlet 2.0 或更高版本的服务器环境中安装并运行 OracleJSP，并且 OracleJSP 8.1.7 发行版已经在下列环境中被测试过：

- Apache Software Foundation 的 Apache/JServ 1.1
Apache/JServ 是一个没有 JSP 环境的 Web Server 和 Servlet 2.0 平台，要在此环境下运行 JSP 页面，必须首先安装一个 JSP 环境。
- Sun 公司的 JSWDK1.0 (JavaServer Web Developer's Kit)
JSWDK1.0 是一个拥有 Servlet 2.1 环境以及 JSP1.0 参考实现的 Web Server，不过你仍然可以在它上面安装 OracleJSP 以替代原来的 JSP 环境。
- Apache Software Foundation 的 Tomcat 3.1
Tomcat 3.1 是 Apache Software Foundation 与 Sun 公司合作开发的结果，它是一个拥有 Servlet 2.2 环境以及 JSP 1.1 参考实现的 Web Server，不过你仍然可以在它上面安装 OracleJSP 以代替原来的 JSP 环境。你也可以联合使用 Tomcat 和 Apache 作为 Web Server，以替代原来的 Tomcat Web Server。

2.4 OracleJSP 的编程扩展概述

本节提供了 OracleJSP 所支持的扩展编程特性的概述。

OracleJSP 通过定制标签库和定制 JavaBeans 提供了下面所列的扩展功能，所有的扩展都可以很容易地移植到其他 JSP 环境：

- 扩展的数据类型，由 JavaBeans 实现，并且有指定的作用域。
 - 与 XML 和 XSL 集成。
 - 可以访问数据库的 JavaBeans。
 - Oracle JSP Markup Language(JML)定制标签库，它不再要求 JSP 开发人员具有很熟练的 Java 技术，减轻了开发难度。
 - 定制的 SQL 功能标签库。
- OracleJSP 也提供了下列专门用于 Oracle 环境的扩展功能：
- 支持 SQLJ，允许将 SQL 语句直接内嵌到 Java 代码中。
 - 扩展的 NLS 支持。
 - 用来处理事件的 JspScopeListener。
 - 用于应用程序支持的 globals.jsa 文件。

下面将首先讨论这些扩展功能，然后简要地介绍 OracleJSP 页面如何与 Oracle PL/SQL Server Pages 交互。

2.4.1 可移植的 OracleJSP 扩展概述

本小节所讨论的 Oracle 扩展要么是通过 OracleJSP 的 JML 标签库来实现的，要么是通过定制的 JavaBeans 来实现的，因此它们都可以很容易地被移植到其他 JSP 环境中。

OracleJSP 的扩展数据类型

一般情况下，JSP 页面使用核心的 Java 数据类型来表达数量值，但是下列两种标准的方法都不能完全满足 JSP 页面应用的需要：

- 简单数据类型，如 int，float，double 等。
- 在标准的 java.lang 包中提供的封装类，如 Integer，Float，Double 等。

简单数据类型不具有指定作用域的功能，也就是说它们不能在 JSP 作用域对象（如 page，request，session，application 等）中保存，因为只有对象才能被保存在作用域对象中。

封装类的值是对象，因此理论上可以存储在 JSP 作用域对象中，但是它们不能使用 jsp:useBean 语句来声明，因为封装类不遵循 JavaBean 模型并且不提供无参数的构造函数。

并且，封装类的实例是不可改变的，要改变一个值，就必须创建一个新的实例并给它赋上适当的值。

为了解决这些局限性，OracleJSP 提供了 JmlBoolean，JmlNumber，JmlFPNamber 以及 JmlString 等 JavaBean 类，它们被封装在 oracle.jsp.jml 包中，包装了大部分通用的 Java 数据类型。

要获得更详细的信息，请参看“JML 扩展数据类型”。

与 XML 和 XSL 的集成

你可以使用 JSP 语法来生成任何基于文本的 MIME 类型的内容，而不仅仅是 HTML 代码。并且你还可以动态地创建 XML 输出，不过当你使用 JSP 页面生成 XML 文档时，常常希望 XML 数据在被送到客户端输出之前先应用样式表 (stylesheet)。但是要做到这一点，对 JSP 技术来说是很难的，因为 JSP 页面所使用的标准的输出流是通过服务器直接地写回

到客户端的。

OracleJSP 在它所提供的 JML 样例标签库中实现了一个特殊的标签，通过它可以指定所有或部分的 JSP 页面在输出之前必须先被传送到 XSL 样式表中进行转换。在单个的 JSP 页面中你可以多次使用 JML 标签，从而可以使一个页面中不同的部分具有不同风格的样式，尤其重要的是 JML 标签库可以很容易地移植到其他的 JSP 环境中。

并且，OracleJSP 的解释器支持 Sun 公司的 JSP1.1 技术标准中所规定的 XML 可交换语法。

要获得更多的信息，请参看“OracleJSP 对 XML 和 XSL 的支持”。

定制的数据库访问 JavaBeans

OracleJSP 提供了一套定制的 JavaBeans 来访问 Oracle 数据库，下面列出了封装在 oracle.jsp.dbtil 包中的数据库访问 JavaBeans：

- ConnBean，用来打开一个数据库连接。
- ConnCacheBean，用 Oracle 的连接缓冲机制来实现数据库连接。
- DBBean，执行一个数据库查询。
- CursorBean，提供通用的 DML 支持，如 UPDATE，INSERT，DELETE 语句以及查询语句等。

要获得更详细的信息，请参看“Oracle 的数据库访问 JavaBeans”。

OracleJSP 的 SQL 定制标签库

在 OracleJSP 的 8.1.7 发行版中提供了一个定制的标签库来实现 SQL 功能，在库中提供了如下标签：

- dbOpen——打开一个数据库连接。
- dbClose——关闭一个数据库连接。
- dbQuery——执行一个数据库查询。
- dbCloseQuery——在查询时关闭光标。
- dbNextRow——在查询结果集合中将光标移动到下一行。
- dbExcute——执行任何 SQL 语言的 DML 或 DDL 语句。

要获得更详细的信息，请参看“OracleJSP 的 SQL 标签库”。

Oracle JML 定制的标签库

尽管 Sun 公司的 JSP 1.1 标准中支持 Java 以外的脚本语言，但是 Java 仍然是首选的语言，并且在许多情况下是唯一考虑的语言。尽管 JSP 技术的设计目标就是将动态内容的开发（Java 编码）与静态内容（HTML 编码）的开发分隔开来，但对一个不懂任何 Java 语言的网络开发者来说，仍然存在着很多障碍，因为 JSP 技术仍然需要开发者对 Java 语言有一定程度的了解。对一个小的开发小组来说，如果里面没有 Java 专家，情况就相当的糟糕。

OracleJSP 提供了一套定制的标签——JSP 标记语言（JML）作为可选择的解决方案，在 Oracle JML 样例标签库中有一整套的 JSP 标签，你可以利用它来编写你的 JSP 页面，而不用使用 Java 语句。JML 提供的标签可以用来声明变量、控制流程、条件分支、迭代循环、

参数设置以及调用对象等。

JML 标签库也支持 XML 功能，如前面所述。

下例解释了 `jml:for` 和 `jml:print` 标签的用法：

```
<jml:for id="i" from="1" to="5" >
  <H<jml:print eval="i" />
    Hello World!
  </H<jml:print eval="i" />
</jml:for>
```

要获得更多的信息，请参看“JSP 标记语言（JML）样例标签库概述”。

注意：在 JSP1.1 标准之前的 OracleJSP 版本中使用了一个 Oracle 专用的 JML 标签库的编译时实现，这个实现在现在的版本中仍然支持，并作为标准的运行时实现的一个可替换的选择。要获得相关的信息，请参看附录 C“编译时 JML 标签支持”。

2.4.2 Oracle 的专用扩展概述

本节所列出的 OracleJSP 扩展是 Oracle 专用的，不能移植到其他 JSP 环境下。

OracleJSP 对 SQLJ 的支持

动态的 JSP 页面常常需要从数据库中提取数据，但是标准的 JSP 技术没有提供内建的数据库访问支持，JSP 开发者一般必须依赖于标准的 Java 数据库连接（Java DataBase Connectivity，简称 JDBC）API 或者一套定制的数据库 JavaBeans 来访问数据库。

SQLJ 是一套标准的语法，它用来将静态的 SQL 指令直接嵌入到 Java 代码中，极大地简化了数据库访问的编程工作。OracleJSP 和 OracleJSP 解释器都支持在 JSP 代码段中的 SQLJ 程序。

SQLJ 语句用 `#sql` 来标示，你可以使用 JSP 源文件的文件扩展名 `.sqljsp` 来触发 OracleJSP 解释器调用 Oracle SQLJ 解释器以及处理此文件。

要获得相关信息，请参看“OracleJSP 对 Oracle SQLJ 的支持”。

OracleJSP 对扩展 NLS 的支持

OracleJSP 提供了扩展的 NLS 支持，主要用在那些不能对多字节的请求参数和 JavaBean 属性设置进行编码的 servlet 环境下。

在这样的环境下，OracleJSP 提供了 `translate_params` 配置参数，它允许 OracleJSP 直接取代 servlet 容器来进行编码工作。

JspScopeListener 和事件处理

Oracle 提供了 `JspScopeListener` 接口来对 JSP 应用程序中具有各种作用域的对象的生命周期进行管理。

标准的 Servlet 和 JSP 的事件处理是通过 `javax.servlet.http.HttpSessionBindingListener` 接口来提供的，但是它只能处理基于 session 的事件。相反，Oracle 的 `JspScopeListener` 还能处理基于 page、基于 request 以及基于 application 的事件。

要获得更多的信息，请参看“OracleJSP 事件处理——`JspScopeListener`”。

globals.jas 文件和应用程序支持 (Servlet 2.0)

在 Servlet 2.0 环境下, Servlet 并没有被完全定义, OracleJSP 定义了 globals.jsa 文件以扩展对 Servlet 应用程序的支持。

在任意一个 Java 虚拟机中, 每个应用程序都可以拥有一个 globals.jsa 文件 (或者用等价的说法, 每个 Servlet 环境都拥有一个 globals.jsa 文件)。globals.jsa 文件通过使用应用程序在 Servlet 环境下就可以模拟没有完全定义的 servlet 环境和 HTTP 会话的动作。

globals.jas 文件也为全局的 Java 声明和 JSP 指令提供了空间, 这些全局变量和指令可以在应用程序中的所有 JSP 页面中使用。

2.4.3 Oracle PL/SQL Server Pages 与 OracleJSP 联合使用

Oracle 提供了一个叫做 PL/SQL Server Page (PSP) 的产品, PSP 技术允许将 PL/SQL 代码段和存储过程调用内嵌在 HTML 页面中, 提供了与 JSP 技术相类似的开发优势。也就是说, 页面的动态内容开发可以和页面的静态内容开发相互独立, HTML 专家可以专管页面的静态部分, 而 PL/SQL 专家可以专管页面的动态部分。

在 PSP 页面中用来区分 PL/SQL 代码段的语法与在 JSP 页面中用来区分 Java 代码段的语法是相同的。

本节的剩余部分将讨论对 JSP-PSP 交互的支持, 并且包含一些有关 PSP URL 的背景知识。

JSP 页面和 PSP 页面的交互

当最终用户运行一个 PSP 应用程序时, PSP 页面被解释成存储过程, 然后 PL/SQL 网关调用它以生成网络浏览器的输出。因为 PL/SQL 网关在 Oracle8i 里是在 servlet 环境下运行的, 所以运行在 Oracle Servlet Engine 下的 JSP 页面就可以以下面两种方式来和 PSP 页面交互:

- 在 JSP 页面中使用动态包含 (jsp:include) 来包含一个 PSP 页面。
- 在 JSP 页面中动态链接到 (jsp:forward) 一个 PSP 页面。

不过, PSP 页面没有相应的功能可以动态包含或链接到一个 JSP 页面, 并且你不能在 JSP 页面中静态地包含 (利用 %<include%> 指令在解释时静态包含一个文件) 一个 PSP 页面。

PSP 页面的 URL

当每个 PSP 页面在数据库里被加载和编译的时候, 它变成一个 PL/SQL 的存储过程。PSP 页面存储过程的名字要么在页面里面显式地声明 (使用 %<pls sql procedure="proc-name"%> 语法), 要么从 PSP 文件的名字导出。

如果给定了 PL/SQL 存储过程的名字, 页面的 URL 就由下面的通用语法来确定:

http://host[:port]/some-prefix/dad/[schema.]proc-name

其中 <some-prefix> 对内嵌的 PL/SQL 其中来说是 pls sql, 并且 <dad> 是运行存储过程的数据库访问描述符 (database access descriptor)。

2.5 OracleJSP 的发行版本和特性小结

OracleJSP 1.1.0.0.0 发行版是由 Oracle8i 8.1.7 发行版所提供的，它全面支持 JSP 1.1 标准。在本书中，“OracleJSP 8.1.7 发行版”与“OracleJSP 1.1.0.0.0 发行版”是统一的。

一些其他的支持 OracleJSP 的 Oracle 平台还没有包含最新的 OracleJSP 发行版本，但是它们都集成了 OracleJSP 1.1.0.6.1 或 1.1.0.6.0 发行版，全面支持 JSP 1.0 标准。

2.5.1 Oracle 平台所提供的 OracleJSP 发行版本

表 2-1 总结了 OracleJSP 的发行版本与提供它的 Oracle 平台的发行版本。

表中“OracleJSP 特性注解”一栏对 OracleJSP 1.1.0.0.0 发行版没有意义，因为它只能应用于特定的 Oracle 平台。

表 2-1 Oracle 平台的发行版本和 OracleJSP 的发行版本

Oracle 平台	Servlet 环境	OracleJSP 的发行版本	OracleJSP 特性注解
Oracle Servlet Engine (Oracle8i), 8.1.7 发行版	Servlet 2.2	OracleJSP 1.1.0.0.0 (JSP 1.1)	n/a
Oracle Internet Application Server, 1.0.1 发行版	Servlet 2.0 (Apache/JServ)	OracleJSP 1.1.0.0.0 (JSP 1.1)	n/a
Oracle Internet Application Server, 1.0.0 发行版	Servlet 2.0 (Apache/JServ)	OracleJSP 1.0.0.6.0 (JSP 1.0)	globals.jsa 配置参数 JML 限制
Oracle Application Server, 4.0.8.2 发行版	Servlet 2.1	OracleJSP 1.0.0.6.0 (JSP 1.0)	配置参数 JML 限制
Oracle Web-to-go, 1.3 发行版	Servlet 2.1	OracleJSP 1.0.0.6.1 (JSP 1.0)	配置参数 JML 限制
Oracle JDeveloper, 3.1 发行版	Servlet 2.1	OracleJSP 1.0.0.6.1 (JSP 1.0)	配置参数 JML 限制

对上表所提及的 Oracle 平台，都可以下载，包含并使用更新的 OracleJSP 版本，OracleJSP 版本文档一般作为产品的一部分提供。

为了检查某一个特定的环境中所使用的 OracleJSP 版本，可以使用隐式对象 application 来得到它的发行版本号，例子如下：

```
<%= application.getAttribute("oracle.jsp.versionNumber") %>
```

2.5.2 OracleJSP 1.0.0.6.x 发行版特性注解

下面描述了表 2-1 中“OracleJSP 特性注解”一栏中所列特性的重要性，只适用于 OracleJSP 1.0.0.6.x 发行版本：

- Servlet 2.0 标准没有提供一套完整的网络应用程序框架。对 Servlet 2.0 环境来说，比如 Apache/JServ 和 Oracle Internet Applications Server（它使用 Apache/JServ），所

以的 OracleJSP 发行版本都通过 `globals.jsa` 机制提供了扩展的支持以实现一个完整的应用程序框架。

- 一些在 OracleJSP 1.1.0.0.0 发行版中所支持的配置参数在 1.0.0.6.x 发行版中并不支持，请参看“配置参数列表”。
- OracleJSP 1.0.0.6.x 发行版是在 JSP 1.0 标准下开发的，因此不能支持 JSP 1.1 标准中的定制标签库机制。这些 OracleJSP 发行版只能通过 Oracle 专用的编译时实现以及对 OracleJSP 解释器的扩展来支持 JML 标签。

在 OracleJSP1.0.0.6.x 发行版中，使用 JML 需要 `taglib` 指令，并且指令必须指定包含标签库的类，如下例所示：

```
<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>
```

相比来说，当使用符合 JSP1.1 标准的 JSP 实现时（诸如 OracleJSP 1.1.0.0.0 发行版），`taglib` 指令需要指定标签库描述文件（一个 `.tld` 文件或者 `.jar` 文件），如下例所示：

```
<%@ taglib uri="/WEB-INF/tlds/jmltags.tld" prefix="jml" %>
```

要获得 JML 编译时实现的相关信息，请参看附录 C “编译时 JML 标签支持”。

2.6 OracleJSP 运行模型

前面已经提到过，你可以在多重 Server 环境下使用 OracleJSP，OracleJSP 提供了如下两种不同的运行模型：

- 在 Oracle Servlet Engine 以外的环境中，OracleJSP 容器一般在 JSP 页面运行前按需解释页面，与大部分厂商所提供的 JSP 实现是一致的。
- Oracle Servlet Engine 环境下——JSP 页面运行在 Oracle8i 数据库里——开发者提前解释页面，将它们加载到 Oracle8i 数据库里作为可运行的 Servlets（有相应的命令行工具用来解释页面、加载页面以及发布页面以使页面可以运行，你可以在客户端也可以在服务器端来解释页面）。当最终用户请求 JSP 页面时，页面无需解释就可以直接运行。

2.6.1 按需解释模型

OracleJSP 在除了 Oracle Servlet Engine 以外的所有支持它的 Servlet 环境里都使用典型的按需解释模型。例如在 Apache/JServ Web Server 中，或者其他的 Oracle 环境中使用 OracleJSP 时。

当 JSP 页面被 Web Server 请求时，如果需要的话就解释并编译此页面（当页面的实现类不存在或者比 JSP 页面源文件的日期早时），最后运行这个 JSP 页面。

要注意一点，那就是 Web Server 首先应该正确配置，将 `*.jsp` 文件扩展名与 `JspServlet` 程序之间建立关联。关于如何在 Apache/JServ，Sun 公司的 JWSDK 以及 Tomcat 环境下完成配置过程，请参看“配置 Web Server 和 Servlet 环境运行 OracleJSP”。

2.6.2 Oracle Servlet Engine 的预解释模型

如果要在 Oracle Servlet Engine (OSE) ——一个包含在 Oracle8i 里的 Web Server 和

Servlet 容器——下面运行 JSP 页面，就要使用 OSE 的预解释模型。页面首先被预解释并发布到 Oracle8i 数据库里作为可运行的 Servlets，在请求时可以直接运行。

发布页面的步骤

要将 JSP 页面发布到 Oracle8i 数据库里，请按照下面步骤来做：

1. 预解释 JSP 页面（一般还要包括编译），由 JSP 解释器生成的页面实现类实质上就是可以运行的 Servlets 程序。
2. 将解释过的 JSP 页面加载到 Oracle8i 数据库里。
3. “热加载”生成的 JSP 页面（可选类）。
4. “发行”JSP 页面以使数据库可以访问它们并运行它们。

有相应的命令行工具可以用来解释、加载以及发行 JSP 页面，解释器首先创建页面实现类到一个 .java 文件，然后将 .java 文件编译成 .class 文件。

热加载可以通过额外的步骤来实现，它是一个有用的特性，可以允许在 JSP 页面实现类中更有效地使用语法字符串（如生成的 HTML 标签等）。

将 JSP 页面发布到 Oracle8i 时，可以在服务器端解释页面，以可以在客户端解释页面。要获得更多的信息，请参看“发布到 Oracle8i——服务器端解释”和“发布到 Oracle8i——客户端解释”。

Oracle Servlet Engine 的 JSP 容器

Oracle Servlet Engine 拥有自己的 OracleJSP 容器，它包括了标准 OracleJSP 容器的大部分，但是没有解释器（因为在 OSE 环境下运行的任何 JSP 页面都是预解释的）。

OSE 也拥有前端 JSP 处理工具，它的功能与按需解释模型中的 JspServlet 功能相似。

OSE 的前端处理工具通过 Servlet 路径（常常指向一个虚拟路径）来查找和运行 JSP 页面。当你在发行一个 JSP 页面时就必须指定一个 Servlet 路径，它被记录在 Oracle8i 的 JNDI 名字空间里。

2.7 Oracle JDeveloper 对 OracleJSP 的支持

可视化的 Java 程序开发工具也开始支持 JSP 编程了，并且 Oracle 的 JDeveloper 不仅支持 OracleJSP，并且还包括下面的特性（适用于 JDeveloper 3.1 发行版）：

- 集成了 OracleJSP 容器，支持全部应用程序开发周期——编码、调试以及运行 JSP 页面。
- 调试已经发布的 JSP 页面。
- 拥有 JDeveloper Web Beans，一套扩展的允许数据库访问及网络访问的 JavaBeans。
- 提供了 JSP 元素向导，可以方便地将预定义好的网络组件添加到页面中来。
- 支持页面中含有定制的 JavaBeans。
- 对依赖于 JDeveloper Business Components for Java(BC4J)的 JSP 应用程序提供了一个发布选项。

要获得关于 JSP 发布支持的更多信息，请参看“用 JDeveloper 发布 JSP 页面”。

在调试方面，JDeveloper 可以在 JSP 页面源代码中设置断点，并且可以跟踪对 JDeveloper 的调用。相对于传统的手工调试技术（在 JSP 页面中添加 print 语句将状态信息输出到响应流中，然后在浏览器中查看，或者通过调用隐式对象 application 的 log() 方法，然后查看服务器的日志文件）来说，这无疑为开发者带来了巨大的便利性。

第3章 基础知识

本章讨论了 JSP 编程中最基本的问题，如应用程序和会话、JSP-servlet 之间的交互、资源管理、应用程序和文档根目录等，最后介绍了一个 JSP 实例“Starter samper”来演示数据库访问功能。

本章涉及以下主题：

- 基本问题
- 应用程序根目录和文档根目录的功能
- JSP 的应用程序和会话
- JSP-Servlet 之间的交互
- JSP 资源管理
- JSP 运行时错误处理
- JSP 实例“Starter sample”

3.1 基本问题

本节讨论了在进行 JSP 开发之前应该注意到的几个问题，所涉及内容如下：

- 安装和配置概述
- 开发环境与发布环境的比较
- 客户端应考虑的问题

3.1.1 安装和配置概述

在附录 A “安装与配置”中讨论了 OracleJSP 在各种环境尤其是主流的非 Oracle 环境下的安装与配置过程，请参看它以获得相关信息。

要在支持 OracleJSP 的 Oracle 环境下安装并配置，请参看特定 Oracle 产品的相关文档。

在 Oracle8i 中，Oracle8i JServer 提供了包含有 OracleJSP 的 Oracle Servlet Engine(OSE)。

3.1.2 开发环境与发布环境的比较

如果目标平台是诸如 Apache/JServ 这样的非 Oracle 环境，JSP 开发者一般要在与目标平台相同的环境中进行开发。在这种情况下，附录 A “安装与配置”中的安装、配置指令对开发环境和发布环境都适用，尽管可能有一些配置参数对开发环境更为密切相关。

如果目标平台是 Oracle Servlet Engine 或者其他的 Oracle 环境，开发者至少有两种方案可选：

- 使用 Oracle JDeveloper 作为开发环境和发布环境
JDeveloper 包含了 OracleJSP 和 Servlet 容器，可以在开发过程中用来测试、运行 JSP 程序，它也提供了很多特性以帮助开发人员更容易地将完成的产品发布到目标平台。

- 在诸如 Apache/JServ 等非 Oracle 环境下进行开发和测试，然后发布到目标 Oracle 平台下进行最终的测试和使用。

在这种情况下，附录 A 中提供的信息可能对开发环境的设置更为适用。

在开发环境下测试后，你可以预解释 JSP 页面，将它们发布到 Oracle8i 数据库里，这些都可以通过命令工具来完成。OracleJSP 的命令行解释器跟按需解释模型里的解释工具有等价的功能，可以通过命令行参数来设置，并且它们的命令行参数有对应关系。

要获得支持 OracleJSP 的 Oracle 环境的安装和配置信息，请参看特定产品的相关文档。

3.1.3 客户端应考虑的问题

JSP 页面可以在任何支持 HTTP 1.0 或更高版本的浏览器中运行。

客户端浏览器无需任何 JDK 或者其他的 Java 环境，因为 JSP 页面中所有的 Java 代码都在网络服务器上或数据服务器上执行。

3.2 应用程序根目录和文档根目录的功能

本节对应用程序根目录（Application Roots）和文档根目录（Doc Roots）提供了一个概述，并且介绍了 Servlet 2.2 和 Servlet 2.0 环境下它们功能的差异。

3.2.1 Servlet 2.2 环境下的应用程序根目录

前面曾经提到过，Servlet 2.2 标准对每个应用程序都提供了它自己的 Servlet 环境。每个 Servlet 环境与文件系统的目录路径相关联，并将此路径作为应用程序中各个模块的相对基路径。这个基路径就是应用程序根目录，每个应用程序都有它自己的应用程序根目录。

这与 Web Server 使用文档根目录作为一个网络应用程序中各个 HTML 页面和其他文件定位的根目录是相似的。

对 Servlet 2.2 环境下的应用程序来说，在应用程序根目录（用来存放 Servlets 和 JSP 页面）和文档根目录（用来存放静态文件，如 HTML 文档等）之间存在一对一的映射关系，它们在实质上是相同的。

请注意，Servlet 一般有如下通用形式的 URL：

`http://host[:port]/contextpath/servletpath`

当一个 Servlet 环境被创建的时候，在应用程序根目录和上面的 URL 中的 contextpath 之间就会建立映射关系。

例如：假设一个应用程序的根目录是 /home/dir/mybankappdir，并且它被映射到 contextpath 是 mybank，如果应用程序有一个 Servlet，它的路径是 loginservlet，那么此 Servlet 的 URL 就是：

`http://host[:port]/mybank/dir1/abc.html`

对每个 Servlet 平台来说，一般都有个缺省的 Servlet 环境，它的 contextpath 就是 “/”，并且被映射到缺省的应用程序根目录。例如，假设应用程序的缺省根目录是

/home/mydefaultdir, 并且应用程序中有一个 Servlet, 它的路径是 myervlet, 那么此 Servlet 的 URL 就是:

```
http://host[:port]/myervlet
```

(如果在 URL 中指定的 contextpath 不存在, 那么缺省的 contextpath 就会被使用。)对 HTML 文件来说, 下面的 URL 指向文件/home/mydefaultdir/dir2/def.html:

```
http://host[:port]/dir2/def.html
```

3.2.2 Servlet 2.0 环境下的应用程序根目录

Apache/JServ 和其他的 Servlet 2.0 环境还没有应用程序根目录的概念, 因为它们只允许单个的应用程序环境, 所有 Web Server 的文档根目录就是应用程序的根目录。

对 Apache 来说, 文档根目录一般都是些形如 “.../htdocs” 的目录, 并且它还可能通过在 http.conf 配置文件设置别名来指定虚拟文档根目录。

在 Servlet 2.0 环境下, OracleJSP 对文档根目录和应用程序根目录提供如下的功能:

- 缺省情况下, OracleJSP 使用文档根目录作为应用程序根目录。
- 通过 OracleJSP 的 globals.jsa 机制, 你可以在文档根目录下设置一个目录作为任何一个应用程序的根目录, 并且在此目录下要设置一个 globals.jsa 文件来作为一个标记。通过这种机制, Servlet 2.0 就可以支持多个应用程序。

3.3 JSP 的应用程序和会话概述

本节对 OracleJSP 是如何支持 JSP 应用程序和会话提供了一个简单的概述。

3.3.1 OracleJSP 对应用程序和会话的通用支持

OracleJSP 使用 Servlet 环境的内在机制来管理应用程序和会话。对 Servlet 2.1 和 Servlet 2.2 环境来说, 这些内在的机制已经足够了, 它为每个 JSP 应用程序提供不同的 Servlet 环境以及 session 对象。

不过, 在诸如 Apache/JServ 等 Servlet 2.0 环境下, 使用 Servlet 机制就会存在很多问题。网络应用程序的概念在 Servlet 2.0 标准下没有完整的定义, 因此在每个 Servlet 容器里只有一个 Servlet 环境, 并且在每个 Servlet 容器里也有一个 session 对象。针对 Apache/JServ 和其他 Servlet 2.0 环境, OracleJSP 提供了扩展允许每个应用程序有不同的 Servlet 环境和 session 对象 (如果 Web Server 只处理一个应用程序, 那么这个扩展不是必需的)。

注意: 要获得 Apache/JServ 和其他 Servlet 2.0 环境的相关信息, 请参看的“Apache/JServ 环境应考虑的问题” 和的 “globals.jsa 功能概述”。

3.3.2 JSP 的缺省会话请求

一般来说, Servlets 在缺省情况下不会请求一个 HTTP 会话, 但是 JSP 页面实现类在缺省情况下需请求一个 HTTP 会话, 你可以通过在 page 指令中将 session 参数设成 false 来改变这一缺省设置, 如下所示:

```
<%@ page... session="false" %>
```

3.4 JSP-Servlet 之间的交互

尽管 JSP 页面编码在很多方面是很方便的，但是在某些情况下还是需要调用 Servlets，比如当你要输出二进制数据时。

因此，在某些情况下需要在应用程序的 Servlet 和 JSP 页面中来回跳转，本节讨论如何实现它们两者之间的交互，包括以下内容：

- 在 JSP 页面中调用 Servlet
- 给被调用的 Servlet 传递数据
- 在 Servlet 中调用 JSP 页面
- 在 JSP 页面和 Servlet 之间传递数据
- JSP-Servlet 交互实例

注意：本节讨论的内容只适用于 Servlet 2.2 环境，对 Apache/JServ 和其他的 Servlet 2.0 环境来说，本节引用了其他部分的内容来作合适的参考。

3.4.1 在 JSP 页面中调用 Servlet

与在 JSP 页面中调用其他 JSP 页面一样，你可以使用 `jsp:include` 和 `jsp:forward` 语句在 JSP 页面中调用一个 Servlet。下面是一个例子：

```
<jsp:include page="/servlet/MyServlet" flush="true" />
```

在页面的执行过程中，当遇到这一语句时，页面缓冲区里的内容被输出到浏览器，并且这个 Servlet 也被执行，当 Servlet 执行结束后，控制权又回到 JSP 页面中，此时 JSP 页面继续执行。这与使用 `jsp:include` 语句在一个 JSP 页面里调用另一个 JSP 页面具有相同的功能。

与在 JSP 页面中使用 `jsp:forward` 语句以转移到另外一个 JSP 页面一样，下面的语句将清空页面缓冲区，终止当前 JSP 页面的执行过程，并且执行指定的 Servlet：

```
<jsp:forward page="/servlet/MyServlet" />
```

注意：在 Apache/JServ 或者其他 Servlet 2.0 环境下，你不能包含或者前进到一个 Servlet，要实现这样的功能，只能自己写一个 JSP 封装页面来代替。要获得相关信息，请参看“Apache/JServ 下的动态包含与前进”。

3.4.2 给被调用的 Servlet 传递数据

当在 JSP 页面中动态地包含或前进到一个 Servlet 时，你可以利用 `jsp:param` 标签传递数据给 Servlet（与动态包含或前进到一个 JSP 页面相同）。

`jsp:param` 标签在 `jsp:include` 和 `jsp:forward` 标签中使用，如下例所示：

```
<jsp:include page="/servlet/MyServlet" flush="true" >
  <jsp:param name="username" value="Smith" />
  <jsp:param name="userempno" value="9876" />
</jsp:include>
```

另外，你也可以通过具有合适作用域的 `JavaBean` 或通过 HTTP 请求对象的属性来在 JSP 页面和 Servlet 之间传递数据。使用请求对象的属性传递数据将在后面 3-9 页的“在 JSP 页

面和 Servlet 之间传递数据”中讨论。

注意: jsp:param 标签是 JSP 1.1 标准所引入的。

3.4.3 在 Servlet 中调用 JSP 页面

你可以在一个 Servlet 中通过标准的 javax.Servlet.RequestDispatcher 接口所提供的功能来调用 JSP 页面。在你的 Servlet 代码中使用以下步骤以调用 JSP 页面:

1. 从 Servlet 的实例中得到 Servlet 环境的实例:
`ServletContext sc=this.getServletContext();`
2. 从 Servlet 的实例中得到 RequestDispatcher 的实例, 指定目标 JSP 页面的页面相对路径或程序相对路径来作为 get.RequestDispatcher() 方法的输入参数:
`RequestDispatcher rd=sc.get.RequestDispatcher("/jsp/mypage.jsp");`
在此步骤之前, 你可以通过 HTTP 请求对象的属性将参数传递给 JSP 页面, 要获得更多信息, 请参看下节“在 JSP 页面和 Servlet 之间传递数据”。
3. 调用 Request Dispatcher 对象的 include()或 forward()方法, 指定 HTTP 请求(request)和响应(response)对象作为这两个方法的输入参数。如下所示:
`rd.include(request, response);`
或
`rd.forward(request, response);`
这两个方法的功能与 jsp:include 和 jsp:forward 动作标签的功能是相似的。include()方法只是暂时得到程序控制权, 在执行完后返回调用的 Servlet 中。
请注意 forward()方法要将输出缓冲清空。

注意:

- HTTP 请求和响应对象必须在程序前面使用标准的 Servlet 功能得到, 如调用 javax.servlet.http.HttpServlet 类的 doGet()方法。
- 此功能是在 Servlet 2.1 标准所引入的。

3.4.4 在 JSP 页面和 Servlet 之间传递数据

在前一节“在 Servlet 中调用 JSP 页面”中曾提到在 Servlet 中通过 Request Dispatcher 调用 JSP 页面时, 你可以使用 HTTP 请求对象的属性来传递数据。

要达到这一目标, 可以通过下面所列方法中的一种来实现:

- 在调用 getRequest Dispatcher()方法时, 在目标 JSP 页面的 URL 后面加上查询字符串, 使用“?”开始, 然后是一对对的 name=value 用来指定传进去的参数值。
下面是一个例子:
`RequestDispatcher rd =
sc.getRequestDispatcher("/jsp/mypage.jsp?username=Smith");`
在目标 JSP 页面(或 Servlet)中, 你可以使用隐式对象 request 的 getParameter()方法来得到通过这种方法所设置的参数值。
- 可以直接使用 HTTP 请求对象的 SetAttribute()方法。

下面是一个例子：

```
request.setAttribute("username", "Smith");
RequestDispatcher rd = sc.getRequestDispatcher("/jsp/mypage.jsp");
在目标 JSP 页面（或 Servlet）中，你可以使用隐式对象 request 的 getParameter()
方法来得到通过这种方法所设置的参数值。
```

注意：

- 此功能是 Servlet 2.1 标准所引入的，要注意的是，Servlet 2.1 标准和 Servlet 2.2 标准在此功能上的规定是有差异的——在 Servlet 2.1 环境中一个给定的属性只能被设置一次。
- 本节讨论的方法可以用来代替 jsp:param 标签从 JSP 页面中向 Servlet 传递数据。

3.4.5 JSP-Servlet 交互实例

本节提供了一个 JSP 页面与 Servlet 交互的实例，使用了前面几节所描述的功能。在这个实例中，JSP 页面 Jsp2Servlet.jsp 包含了 Servlet 程序 MyServlet，而 MyServlet 又包含了另一个 JSP 页面 welcome.jsp。

Jsp2Servlet.jsp 的程序代码：

```
<HTML>
<HEAD> <TITLE> JSP Calling Servlet Demo </TITLE> </HEAD>
<BODY>
<!-- Forward processing to a servlet -->
<% request.setAttribute("empid", "1234"); %>
<jsp:include page="/servlet/MyServlet?user=Smith" flush="true"/>
</BODY>
</HTML>
```

MyServlet.java 的程序代码：

```
import javax.servlet.*;
import javax.servlet.http.*;
import java.io.PrintWriter;
import java.io.IOException;

public class MyServlet extends HttpServlet {

    public void doGet (HttpServletRequest request,
        HttpServletResponse response)
        throws IOException, ServletException {
        PrintWriter out= response.getWriter();
        out.println("<B><BR>User:" + request.getParameter("user"));
        out.println
            ("", Employee number:" + request.getAttribute("empid") + "</B>");
        this.getServletContext().getRequestDispatcher("/jsp/welcome.jsp").
            include(request, response);
    }
}
```

welcome.jsp 的程序代码:

```
<%-----
Copyright © 1999, Oracle Corporation. All rights reserved.
-----%>

<HTML>
<HEAD> <TITLE> The Welcome JSP </TITLE> </HEAD>
<BODY>

<H3> Welcome! </H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! </B></P>
</BODY>
</HTML>
```

3.5 JSP 资源管理

javax.servlet.http 包提供了一套标准的机制来管理会话资源, 另外 Oracle 提供了管理应用程序、会话、页面和请求等资源的扩展功能。

3.5.1 标准的会话资源管理——HttpSessionBindingListener

一个 JSP 页面必须合适地管理它在运行过程中所得到的资源, 比如 JDBC 连接、状态以及查询结果集合等。标准的 javax.servlet.http 包提供了 HttpSessionBindingListener 接口和 HttpSessionBindingEvent 类来管理具有会话作用域的资源。举例来说, 通过这个机制, 假设有一个用来查询游标, 而在 HTTP 会话终止的时候自动被关闭。

本节讨论如何使用 HttpSessionBindingListener 的 valueBound () 和 valueUnbound () 方法。

注意: JavaBean 的实例必须在 HTTP 会话对象的事件通知列表中将自己注册进去, 但是 jsp:useBean 语句已经为你自己做好了这一切。

valueBound () 和 valueUnbound () 方法

一个实现了 HttpSessionBindingListener 接口的对象可以实现 valueBound () 和 valueUnbound () 方法, 这两个方法的输入参数都是一个 HttpSessionBindingEvent 的实例, 它们被 Servlet 容器所自动调用——当对象被存储到会话中时 valueBound() 方法被调用; 当对象从会话中删除或者会话超时或者会话无效时 valueUnbound () 方法被调用。通常, 开发人员使用 valueUnbound () 方法去释放对象中所占用的资源 (在下面的例子中, 数据库连接被释放掉。)

注意: OracleJSP 提供了扩展的功能对其他的资源继续管理, 允许在 JavaBean 中编程来管理具有页面作用域、请求作用域、应用程序作用域以及会话作用域的资源。

在下一节 “JDBCQueryBean 程序代码” 中提供了一个 JavaBean 的样例, 它实现了 HttpSessionBindingListener 接口, 同时也提供了一个样例 JSP 页面来调用 JDBCQueryBean。

JDBCQueryBean 程序代码

下面是 JDBCQueryBean 的样例代码，JDBCQuerybean 实现了 HttpSessionBindingListener 接口，使用 JDBC OCI 驱动来进行数据库连接。（如果你想在自己的环境中运行这个例子的话，请使用合适的 JDBC 驱动和连接字符串。）

JDBCQueryBean 从 HTML 请求中获得一个搜索条件，然后基于这个搜索条件执行一个动态查询并将结果返回。

JDBCQueryBean 也实现了 ValueUnbound() 方法（在 HttpSessionBindingListener 接口中指定），用来在会话结束之前将数据库连接关闭。

```
package mybeans;

import java.sql.*;
import javax.servlet.http.*;

public class JDBCQueryBean implements HttpSessionBindingListener
{
    String searchCond = "";
    String result = null;

    public void JDBCQueryBean() {
    }

    public synchronized String getResult() {
        if (result != null) return result;
        else return runQuery();
    }

    public synchronized void setSearchCond(String cond) {
        result = null;
        this.searchCond = cond;
    }

    private Connection conn = null;

    private String runQuery() {
        StringBuffer sb = new StringBuffer();
        Statement stmt = null;
        ResultSet rset = null;
        try {
            if (conn == null) {
                DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
                conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                                    "scott", "tiger");
            }

            stmt = conn.createStatement();
```

```

        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
            (searchCond.equals("") ? "" : "WHERE " + searchCond ));
        result = formatResult(rset);
        return result;

    } catch (SQLException e) {
        return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    }
    finally {
        try {
            if (rset != null) rset.close();
            if (stmt != null) stmt.close();
        }
        catch (SQLException ignored) {}
    }
}

private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
        sb.append("<P> No matching rows.<P>\n");
    else {
        sb.append("<UL><B>");
        do { sb.append("<LI>" + rset.getString(1) +
            " earns $ " + rset.getInt(2) + "</LI>\n");
        } while (rset.next());
        sb.append("</B></UL>");
    }
    return sb.toString();
}

public void valueBound(HttpSessionBindingEvent event) {
    // do nothing -- the session-scoped bean is already bound
}

public synchronized void valueUnbound(HttpSessionBindingEvent
event) {
    try {
        if (conn != null) conn.close();
    }
    catch (SQLException ignored) {}
}
}

```

注意：上面的代码只是作为一个例子，在大规模的网络应用程序开发中并不建议采用它来作为处理数据库连接池的手段。

JSP 页面 UseJDBCQueryBean

在上一节中定义了 JDBCQueryBean，下面的 JSP 页面说明了如何在 JSP 中调用它。在

这个页面中，JDBCQueryBean 的作用域被设为会话作用域，并且它用来显示符合用户输入的搜索条件的所有的雇员名字。

JDBCQueryBean 通过 JSP 页面中的 `jsp:setProperty` 命令来得到搜索条件。在 JDBCQueryBean 中有一个属性叫 SearchCond，`jsp:setProperty` 首先从 HTML 表单中用户所输入的请求参数中得到 SearchCond 的值，然后赋给 JDBCQueryBean 的 SearchCond 属性。（用 HTML 的 INPUT 标签可以指定表单中用户所输入的搜索条件，这个文本框控件的名字叫 SearchCond。）

UseJDBCQueryBean 的页面源代码如下：

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session" />
<jsp:setProperty name="queryBean" property="searchCond" />

<HTML>
<HEAD> <TITLE> The UseJDBCQueryBean JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String searchCondition = request.getParameter("searchCond");
    if (searchCondition != null) { %>
        <H3> Search results for : <I> <%= searchCondition %> </I> </H3>
        <%= queryBean.getResult() %>
        <HR><BR>
    <% } %>

<B>Enter a search condition for the EMP table:</B>

<FORM METHOD="get">
<INPUT TYPE="text" NAME="searchCond" VALUE="ename LIKE 'A%'" SIZE="40">
<INPUT TYPE="submit" VALUE="Ask Oracle">
</FORM>

</BODY>
</HTML>
```

下图是此页面的输入与输出：



HttpSessionBindingListener 的优点

在上面的例子中，除了 HttpSessionBindingListener 机制另一个可以选择的方法是：在 JavaBean 的 finalize 方法中关闭数据库连接。finalize 方法将会在会话关闭后的内存碎片整理过程中被调用。不管，HttpSessionBindingListener 接口比 finalize 方法的动作更容易预测。

内存碎片整理常常依赖于应用程序的内容消耗模式，与之相对的是，HttpSessionBindingListener 接口的 valueUnbound() 方法在会话关闭的时候将会被很稳定地调用。

3.5.2 Oracle 对资源管理的扩展功能概述

Oracle 为了管理应用程序和会话资源以及页面和请求资源，提供了如下的扩展功能：

- JspScopeListener——管理应用程序、会话、页面以及请求对象的资源。
要得到相关信息，请参看“OracleJSP 事件处理——JspScopeListener”。
- globals.jsa 中提供了应用程序和会话事件——start 和 end 事件，一般用在诸如 Apache/JServ 等的 Servlet 2.0 环境下。

3.6 JSP 运行时错误处理

当 JSP 页面在执行和处理客户端请求时，有可能会发生运行时错误，运行时错误可能在页面中出现，也可能在页面外出现（如在一个被调用的 Java Bean 中），本节描述了 JSP 的错误处理机制，并且提供了一个简单的例子。

3.6.1 使用 JSP 错误页面

在 JSP 页面的运行过程中所碰到的任何运行时错误都可以使用下面两种中的任意一种标准的 Java 异常机制来处理。

- 你可以在 JSP 页面的 Java 代码段中捕捉并处理异常，使用标准的 Java 异常处理代码。
- 在 JSP 页面中所没有捕捉到的异常将导致请求信息和异常信息都被送到一个错误页面中，这种机制提供了一种很好的手段来处理 JSP 错误。

你可以通过在原始的 JSP 页面中使用 page 指令，并设置 error Page 参数的值来指定错误页面的 URL。

在 Servlet 2.2 环境下，你也可以通过以下的语句在 web.xml 文件中指定一个缺省的错误页面：

```
<error-page>
  <error-code>404</error-code>
  <location>/error404.html</location>
</error-page>
```

（请参看 Sun 公司的 Java Servlet 2.2 技术标准来获得关于缺省错误页面的更多信息。）
错误页面中必须有一个 page 指令以将 isErrorPage 参数值设为 true。

描述错误的异常对象是一个 java.lang.Exception 的实例，它可以在错误页面中通过隐式的 exception 对象来访问。

只有一个错误页面能访问隐式的 `exception` 对象。

请参看下一节“JSP 错误页面实例”以获得关于如何使用错误页面的例子。

注意：关于 JSP 机制所能处理的异常类型，在 JSP1.1 标准中存在模糊不清的地方。

OracleJSP 解释器所生成的页面实现类能处理 `java.lang.Exception` 类或子类的实例，但是不能处理除了 `Exception` 以外的 `java.lang.Throwable` 类或子类的实例。

OracleJSP 容器将 `Throwable` 的实例抛出给 Servlet 容器。

JSP1.2 标准有望解决这些模糊不清的问题，OracleJSP 在将来的发行版本中也会适当修改处理异常的动作。

3.6.2 JSP 错误页面实例

下面的例子 `nullpointer.jsp` 生成一个错误并使用了错误页面 `myerror.jsp`，在 `myerror.jsp` 页面中输出隐式对象 `exception` 的内容。

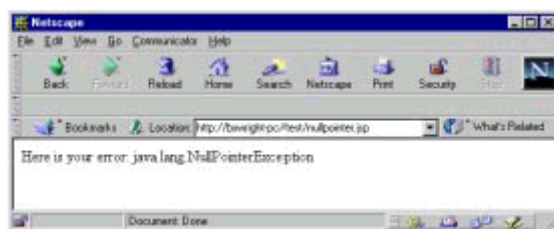
`nullpointer.jsp` 的程序代码：

```
<error-page>
  <error-code>404</error-code>
  <location>/error404.html</location>
</error-page>
```

`myerror.jsp` 的程序代码：

```
<HTML>
<BODY>
<%@ page isErrorPage="true" %>
Here is your error:
<%= exception %>
</BODY>
</HTML>
```

这个例子的输出如下图所示：



注意：当处理过程被导向错误页面时，`nullpointer.jsp` 里的“Nullpointer is generated below:”将不会被输出，这显示了“include”和“forward”功能上的差异——当使用“forward”时，导向的页面的输出内容将取代原始的页面的输出。

3.7 JSP 实例“Starter Sample”

在第一章“概述”中提供了很多简单的 JSP 例子，但是你选择 OracleJSP 的目的就是为了访问 Oracle 数据库，因此本节提供了一个更有价值的实例来演示在 JSP 页面中如何使

用标准的 JDBC 代码来执行数据库查询。

因为 JDBC API 只是一套 Java 接口，所以 JSP 技术直接支持在 Java 代码段中调用它。

注意：

- Oracle JDBC 提供了几个可供选择的驱动：1) JDBC OCI 驱动，在 Oracle 客户端安装；2) 一个 100% 的 Java JDBC Thin 驱动，在任何客户端环境（包括 applets）都可以使用；3) 一个 JDBC 服务器端 Thin 驱动，在一个 Oracle 数据库里访问另外一个 Oracle 数据库；4) 一个 JDBC 服务器端内部驱动，在 Java 代码的运行环境（如 Java 存储过程或者 Enterprise JavaBean）中访问 Oracle 数据库。
- Oracle JSP 也支持 SQLJ（Java 中的内嵌 SQL 语句）以处理静态 SQL 操作，并且提供定制的 JavaBeans 和定制的 SQL 标签以访问数据库。

下面的例子动态地创建一个查询，它首先通过一个 HTML 表单让用户在文本框中输入查询条件，然后点 Ask Oracle 按钮以执行查询。为了执行指定的查询，例子中在 JSP 声明部分定义了一个方法 runQuery()，它使用 JDBC 代码访问数据库。在声明部分还定义了一个方法 formatResult() 来产生输出结果。（例子中使用 JDBC 而不使用 SQLJ 的原因是：查询需要动态生成，而 SQLJ 只适用于静态 SQL 语句。）

HTML 的 INPUT 标签指定了表单中输入的字符串名字叫 cond，因此，cond 也是 HTTP 请求所对应的隐式对象 request 的 getParameter() 方法的输入参数，它可以得到用户所输入的字符串。并且，runQuery() 方法的输入参数也是 cond，它用来在 WHERE 语句中指定一个查询条件。

注意：

- 本例中 runQuery() 方法是使用 <%!...%> 语法来声明的，另外一种声明方法是使用 <%...%> 语法。
- 本例使用 JDBC OCI 驱动，因此需要一个 Oracle 的客户端安装，如果你自己想要运行这个例子，请选用合适的 JDBC 驱动和连接字符串。

实例代码如下：

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

<HTML>
<HEAD> <TITLE> The JDBCQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String searchCondition = request.getParameter("cond");
   if (searchCondition != null) { %>
       <H3> Search results for <I> <%= searchCondition %> </I> </H3>
       <B> <%= runQuery(searchCondition) %> </B> <HR><BR>
   <% } %>
   <B>Enter a search condition:</B>
   <FORM METHOD="get">
   <INPUT TYPE="text" NAME="cond" SIZE=30>
   <INPUT TYPE="submit" VALUE="Ask Oracle">;
   </FORM>
```

```

</BODY>
</HTML>

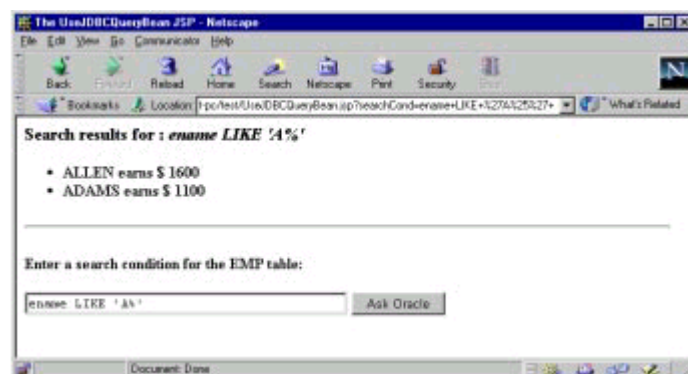
<%-- Declare and define the runQuery() method. --%>
<%! private String runQuery(String cond) throws SQLException {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rset = null;
    try {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn = DriverManager.getConnection("jdbc:oracle:oci8:@",
                                           "scott", "tiger");

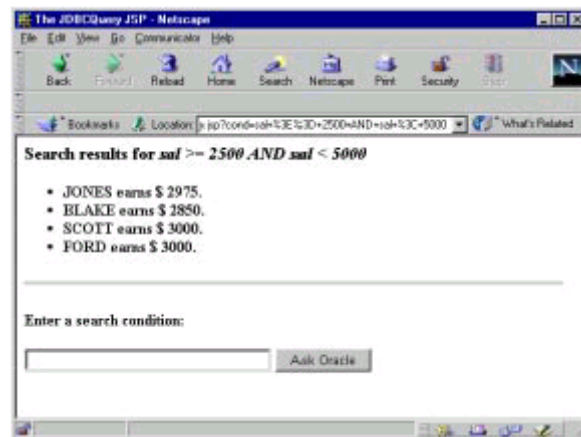
        stmt = conn.createStatement();
        // dynamic query
        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                                   (cond.equals("") ? "" : "WHERE " + cond ));
        return (formatResult(rset));
    } catch (SQLException e) {
        return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    } finally {
        if (rset!= null) rset.close();
        if (stmt!= null) stmt.close();
        if (conn!= null) conn.close();
    }
}

private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
        sb.append("<P> No matching rows.<P>\n");
    else { sb.append("<UL>");
        do { sb.append("<LI>" + rset.getString(1) +
                       " earns $ " + rset.getInt(2) + ".</LI>\n");
        } while (rset.next());
        sb.append("</UL>");
    }
    return sb.toString();
}
%>

```

本例的输入和输出如下图所示：





第4章 关键考虑

本章讨论了一些很重要的编程、配置和运行时需要考虑的问题，以及对特定的运行环境的特殊考虑。本章涉及下面一些内容：

- 通用的 JSP 编程策略、方法和技巧
- 关键的 OracleJSP 配置问题
- OracleJSP 运行时考虑（非 OSE）
- Oracle Servlet Engine 环境应考虑的问题
- Apache/Jserv Servlet 环境应考虑的问题

4.1 通用的 JSP 编程策略、方法和技巧

本节讨论通用的 JSP 页面编程时应考虑的问题，不涉及具体的目标平台的环境。

注意：为了对本节所讨论的问题有所认识，你应该对 Oracle JSP 的解释和发布过程中的问题有所认识，请参看第 6 章“JSP 的解释和发布”。

4.1.1 JavaBeans 与 Java 代码段的比较

在前面描述了 JSP 技术的一个关键优点：包含商业逻辑并且决定动态内容的 Java 代码可以与包含请求信息、外观逻辑和静态内容的 HTML 代码分隔开来，从而允许 HTML 专家专注于 JSP 页面自身的处理逻辑中，而 Java 专家专注于 JSP 页面所调用的 JavaBeans 的商业逻辑中。

一个典型的 JSP 页面往往只包含很少的 Java 代码段，用来处理请求信息或输出内容。在“JSP 实例 Starter Sample”中的样例页面只是用来演示用的，并不是一个理想的设计方案。数据库访问部分（例子中的 runQuery()方法）通常适合于放在 JavaBeans 中，而处理输出内容格式的代码（例子中的 formatResult()方法）更适合于放在 JSP 页面中。

4.1.2 在 JSP 页面中使用 Enterprise JavaBeans

要在 JSP 页面中使用 Enterprise Java Beans (EJB)，你可以选择下面两种方案中的一种：

- 使用 EJB 的封装类，然后在 JSP 页面中像调用其他的 Java Bean 一样调用 EJB。
- 直接在 JSP 页面中调用 EJB。

本节提供了从 JSP 页面中调用 EJB 的两个例子——一个是 JSP 页面在中间层环境运行，另外一个是在 JSP 页面在 Oracle Servlet Engine 环境下运行。

这些例子显示了使用 Oracle Servlet Engine 的一些显著的优点。

在中间层从 JSP 页面调用 EJB

下面的 JSP 页面从一个中间层环境（如 Oracle Internet Application Server）调用 EJB。在这个例子中，服务的 URL 被指定为 sess_iiop://localhost:2481:ORCL（你需要修改它以使

用你自己的主机名: IIOP 端口号和 OrcaI 实例名), JNDI 命名环境 (naming context) 通过 InitialContest(env)构造函数来设置, 其中 env 是一个定参数的哈希表。一旦初始的环境 (ic) 创建起来, 下面的代码使用服务 URL 和 JNDI 名字来查询 EJB Home 对象:

```
EmployeeHome home=(EmployeeHome) ic.lookup(Surl+"/test/employeeBean");
```

接着 home.create()方法被调用, 创建此 JavaBean 的一个实例, 然后这个实例的 query()方法被调用以得到满足一定条件的雇员的名字和薪水。在本例中, 查询条件是用户从 JSP 页面的 HTML 表单中输进去的雇员号码。

下面是例子的程序代码:

```
<HTML>
<%@ page import="employee.Employee, employee.EmployeeHome,
employee.EmpRecord, oracle.aurora.jndi.sess_iiop.ServiceCtx,
javax.naming.Context, javax.naming.InitialContext, java.util.Hashtable"
%>

<HEAD> <TITLE> The CalleJB JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<BR>
<% String empNum = request.getParameter("empNum");
String surl = request.getParameter("surl");
if (empNum != null) {
    try {
        Hashtable env = new Hashtable();
        env.put(Context.URL_PKG_PREFIXES, "oracle.aurora.jndi");
        env.put(Context.SECURITY_PRINCIPAL, "scott");
        env.put(Context.SECURITY_CREDENTIALS, "tiger");
        env.put(Context.SECURITY_AUTHENTICATION,
            ServiceCtx.NON_SSL_LOGIN);
        Context ic = new InitialContext (env);
        EmployeeHome home = (EmployeeHome)ic.lookup (surl +
            "/test/employeeBean");
        Employee testBean = home.create\(\);
        EmpRecord empRec = testBean.query (Integer.parseInt(empNum));

    } catch (Exception e) {
        Error occurred: <%= e %>
    }
} %>

</PRE></BIG></BLOCKQUOTE></h2>
<HR>

<P><B>Enter an employee number and EJB service URL:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="empNum" SIZE=10 value="7654">
<INPUT TYPE=text NAME="surl" SIZE=40 value="sess_iiop://localhost:2481:ORCL">
```

```

<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>

```

在 Oracle Servlet Engine 环境下从 JSP 页面中调用 EJB

如果你打算将 JSP 页面发布到 Oracle8i 的 Oracle Servlet Engine 环境下,那么 EJB 的查找和引用过程将会更简单并且是高度优化的。在下面这个例子中, EJB 查找是在 Oracle8i 的 JNDI 名字空间里完成的, 并且不需要显式地指定一个服务 URL。命名环境可以通过如下简单的调用在当前的会话中被初始化:

```
Context ic=new InitialContext();
```

注意此例中的构造函数是不需要任何参数的, 与中间层环境的例子不同, 要查找某个 JavaBean 只需通过它的 JNDI 名字即可 (不需要服务 URL):

```
EmployeeHome home=(EmployeeHome) ic.lookup("/test/employeeBean");
```

下面是例子的程序代码:

```

<HTML>
<% page import="employee.Employee, employee.EmployeeHome,
employee.EmpRecord, oracle.aurora.jndi.sess_iiop.ServiceCtx,
javax.naming.Context, javax.naming.InitialContext,
java.util.Hashtable" %>
<HEAD> <TITLE> The CalleJB JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<BR>
<% String empNum = request.getParameter("empNum");
if (empNum != null) {
    try {
        Context ic = new InitialContext();
        EmployeeHome home = (EmployeeHome)ic.lookup("/test/employeeBean");
        Employee testBean = home.create();
        EmpRecord empRec = testBean.query (Integer.parseInt(empNum));
    }
    %>
<h2><BLOCKQUOTE><BIG><PRE>
    Hello, I'm an EJB in Oracle8i.
    Employee <%= empRec.ename %> earns $ <%= empRec.sal %>
<% } catch (Exception e) { %>
    Error occurred: <%= e %>
<% }
} %>
</PRE></BIG></BLOCKQUOTE></h2>
<HR>

<P><B>Enter an employee number URL:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="empNum" SIZE=10 value="7654">
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>

```

4.1.3 使用 JDBC 的性能增强特性

可以在 JSP 应用程序中使用如下的 OracleJSP 所提供的性能增强特性：

- 缓冲数据库连接（通过 Oracle 扩展）
- 缓冲 JDBC 语句（通过 Oracle 扩展）
- 成批更新语句（通过 Oracle 扩展）
- 在查询过程中预取行（通过 Oracle 扩展）
- 缓冲行记录（通过 Sun 的扩展）

在这些性能增强特性中的大部分都可以由数据库访问 JavaBeans 来支持，比如 ConnBean 和 ConnCacheBean,但是 DBBean 并不支持这些特性。

数据库连接缓冲

创建一个新的数据库连接是很费时的操作，因此在可能的情况下要尽量避免这个操作，而使用数据库连接缓冲来代替。JSP 应用程序可以从一个预先建好的物理连接缓冲池中得到一个逻辑连接，并且在使用结束后将连接返回到缓冲池中。

你可以创建一个连接缓冲池并使其具有四种作用域——应用程序、页面、会话、请求——中的一种。一般情况下，使用最大可能的作用域是最有效的，所以在 Web Server 许可的情况下，尽量使用应用程序作用域，否则使用会话作用域。

Oracle JDBC 连接缓冲方案是建立在 JDBC 2.0 标准扩展所指定的标准的连接池之上的。它在 OracleJSP 所提供的数据库访问 JavaBean 中的 ConnCacheBean 里面被实现。ConnCacheBean 可能是绝大部分开发者所使用的连接缓冲器。

除了 ConnCacheBean 以外，也可以直接使用 Oracle JDBC 的 OracleConnectionCacheImpl 类，下面是一个例子（不过所有的 OracleConnectionCacheImpl 功能都可以由 ConnCacheBean 来实现）：

```
<jsp:useBean id="occi" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
            scope="session" />
```

OracleConnenctionCacheImpl 有着与 ConnCacheBean 相同的属性可用，它们都可以通过 jsp:setProperty 语句或者直接通过类属性设置方法来设置。

JDBC 语句缓冲

语句缓冲是 Oracle8i 8.1.7 发行版才引入的一个 Oracle JDBC 扩展，它通过缓冲在一个物理连接中被重复调用的可执行语句来提高性能，比如一个循环语句或者一个被重复调用的方法都是被缓冲的理想对象。当一个语句被缓冲后，这个语句在以后每次被执行时都不必重新被解析，语句对象不必被重新创建，参数大小定义也不必被重新计算，从而可以提高程序的性能。

Oracle JDBC 语句缓冲方案是在 OracleJSP 所提供的数据库访问 JavaBeans 中的 ConnBeans 和 ConnCacheBean 里面被实现的。这两个 bean 都有一个 StmtCacheSize 属性，它可以通过 jsp:setProperty 语句或者 bean 的 setStmtCacheSize()方法来设置。

语句缓冲也可以直接通过 Oracle JDBC 的 OracleConnection 和 OracleConnectionCacheImpl 类来实现。

注意：语句只能在单个的物理连接中被缓冲，如果在连接缓冲中被使用的同时又允许语句缓冲，那么语句可以跨多个逻辑连接对象被缓冲，不过这多个逻辑连接对象只能是来自单个的缓冲池连接对象，不能跨多个缓冲池连接对象。

成批更新

Oracle JDBC 的成批更新特性将一个每批数目值与每个待处理语句对象关联起来。在普通情况下，JDBC 驱动在每次调用待处理语句的 `executeBatch()` 方法时都要执行此语句，而通过成批更新特性，JDBC 驱动将待处理的语句都添加到一个累积执行请求中，当积累的语句条数达到了预先设定好的每批数目值时，JDBC 驱动就会将所有的操作一起传递给数据库去执行。例如，如果每批数目值被设置为 10，那么在每次积累够待处理语句后，就会有 10 个操作被送到数据库中去处理。

OracleJSP 直接支持 Oracle JDBC 的成批更新机制，它是通过数据库访问 JavaBean 中的 `ConnBean` 的 `executeBatch` 属性来实现的。你可以通过 `jsp:setProperty` 语句或者通过 `ConnBean` 的属性设置方法来设置这一属性。如果使用 `ConnCacheBean` 来替代 `ConnBean`，你可以通过创建的连接和语句对象的 Oracle JDBC 功能来允许成批更新。

要得到关于 Oracle JDBC 成批更新机制的更多信息，请参看“Oracle8i JDBC 开发人员指南与参考”。

行预取

Oracle JDBC 行预取特性允许你设置一个预取的行数，这样在每次查询过程中当结果记录集合得到时可以将多行预取记录一次传给客户端，减少信息在客户端与服务器端来回往返的次数。

OracleJSP 直接支持 Oracle JDBC 的行预取特性，它是通过数据库访问 JavaBean 中的 `ConnBean` 的 `preTetch` 属性来实现的。你可以通过 `jsp:setProperty` 语句或者 `ConnBean` 的属性设置方法来设置这一属性。如果你使用 `ConnCacheBean` 来替代 `ConnBean`，可以通过创建的连接和语句对象的 Oracle JDBC 功能来允许行预取特性。

行记录缓冲

一个缓冲过的行记录提供了一个无连接、可序列化、可滚动的数据容器，用来包含从数据库检索到的数据。对于小的、不经常改变的数据记录来说，这个特性是很有用处的，尤其是对客户端需要经常或连续访问的信息来说。与之相对的，使用一个标准的数据记录需要保持连接和其他资源等信处，因此效率要低一些。不过应该认识到的是越大的缓冲行记录就会耗费客户端越多的内存。

OracleJSP 8.1.7 发行版中，Oracle JDBC 没有提供缓冲行记录的实现，不过 Sun 公司提供了一个参与实现，可以在它的站点上下载。从 Sun 的网站上下载到文件 `rowset.jav` 以后，在你的 Web Server 类路径上包含它的路径，并且在你的代码中引入 `sun.jdbc.rowset.*` 包，然后就可以使用下面的代码去创建并安置一个缓冲行记录：

```
CachedRowSet crs = new CachedRowSet();
crs.populate(rset); // rset is a previously created JDBC ResultSet object.
```


一旦安置了这个行记录，为了得到原始的结果而使用的连接和语句对象都可以被释放了。

4.1.4 静态包含与动态包含的比较

在“指令语句”一节中，曾经描述了 `include` 指令语句，它首先制作一份被包含页面的拷贝，然后在解释过程中将它复制到 JSP 页面中。这种包含被称为静态包含（或解释时包含），并且使用如下的语法：

```
<%@include file= "/jsp/userinfopage.jsp" %>
```

在“JSP 的 Actions 和 `<jsp:include>` 标签集”中描述的 `jsp:include` 语句，在运行过程中动态地将被包含（included）页面的输出包含到包含页面（including）的输出中，这种包含被称为动态包含（或运行时包含），并且使用如下的语法：

```
<jsp:include page= "/jsp/userinfopage.jsp" flash="true"/>
```

对于那些熟悉 C 语言语法的开发者来说，静态包含就相当于 `#include` 语句，而动态包含相当于一个函数调用。它们都是很有用处的功能，不过目标有所不同。

注意：静态包含和动态包含都只能在相同的 Servlet 环境下使用。

静态包含的逻辑

静态包含将会导致包含页面所生成的代码的大小增加，因为在解释过程中被包含页面的文字被真正地拷贝进了包含页面。如果在一个包含页面中有一个页面被包含了多次，那么就有它的多份拷贝被插入包含页面。

一个被静态包含的 JSP 页面不需要是一个独立的、可解释的页面，它可以只包含一些文字内容，只要能被复制到包含页面中即可。而含有被包含文字内容的包含页面则必须是可解释的，但是在被包含页面被复制到包含页面之前，包含页面可以不必是可解释的，如果有一系列的静态的被包含页面，那么它们每个都可以是 JSP 页面的一小片段，但是可能自己不能独立作为一个页面。

动态包含的逻辑

动态包含一般不会显著增加包含页面所生成的代码大小，尽管方法调用被添加了进去。动态包含会导致运行时的处理流程从包含页面切换到被包含页面，与静态包含的直接将包含页面的文字复制到包含页面中形成了很大的对比。

动态包含增加了页面处理的负载，因为要处理额外的函数调用。

一个被动态包含的页面必须是一个独立的、可解释并能自己运行的页面，同样的，包含页面也必须是独立的，能被解释并运行的页面。

优点、缺点以及典型的用途

静态包含影响页面大小；动态包含影响页面处理的负载。静态包含避免了动态包含所必需的对请求分发器的过载。但是一旦涉及到的文件就可能导致问题。（在页面实现类的 `Service` 方法里对页面的大小有 64k 的限制。

过度使用静态包含也可能使调试 JSP 页面更困难，更难于跟踪程序的执行过程。

静态包含典型地用来包含小的文件，并且文件的内容在多个 JSP 页面中重复出现，例如：

- 商标或版权信息，要显示在应用程序每个页面的顶部或底部。
- 声明或指令语句（如 `imports` 语句用来引入 Java 类），在每个页面中都需要。
- 中心状态检查页面，在应用程序的每个页面都要使用。

动态包含对模块化编程是大有好处的。你可以编写一个页面使它某些时候只是运行自己，而某些时候可以用来生成其他页面的输出。动态被包含页面可以多次重用而不增加包含页面的大小。

4.1.5 创建和使用 JSP 标签库的时机

在某些情况下，开发团队可能要考虑创建并使用定制标签，尤其是在下列条件下：

- JSP 页面里面包含有大量 Java 逻辑
- 需要特殊的操作或者重新导向 JSP 输出

替代 Java 语法

因为开发者并不一定都有 Java 编程经验，所以它们可能并不能很理想地胜任 JSP 页面里的 Java 逻辑的编码工作（JSP 页面里需要 Java 逻辑来完成输出内容的格式及外观整理）。

在这种情况下，JSP 标签库也许是很有用的。如果有许多 JSP 页面都需要 Java 逻辑来生成输出内容，那么对 JSP 开发者来说使用标签库来代替 Java 逻辑将是一条很便利的途径。

OracleJPS 所提供的 JML 样例标签库就是一个很好的例子，在库中提供了与 Java 的循环和条件语句等价的标签。

特殊操作或重新导向 JSP 输出

另外一个使用定制标签的通用情况就是在运行时需要对响应对象的输出作特殊处理。也许想得到的功能需要一个额外的步骤或者将输出重新导向到除了浏览器以外的其他地方。

这种情况的一个例子是创建一个定制标签，然后在正文区放置一些文字，这些文字的输出生将被重新定向到一个日志文件而不是浏览器中，如下例所示：（其中 `Cust` 是标签库的前缀，`log` 是标签库的一个标签。）

```
<cust:log>
  Today is <%= new java.util.Date() %>
  Text to log.
  More text to log.
  Still more text to log.
</cust:log>
```

请参看“标签处理器”以获得关于标签正文处理的信息。

4.1.6 使用中心检查页面

为了对你的 JSP 应用程序进行一般的管理或监测，使用中心检查页面往往是很很有用的。此时你要在应用程序的每个页面都包含中心检查页面，中心检查也可以在每个页面的运行

过程中完成如下任务：

- 检查会话状态
- 检查登录状态（比如查看 cookie 以检查是否一个合法的登录已完成）
- 检查使用率情况（如果日志记录机制被允许，那么如鼠标点击率或页面访问率等信息都会被记录）

中心检查页面还能完成更多跟上面所列相似的功能。

作为一个例子，考虑有一个会话检查类 `MySessionChecker`，它实现了 `HttpSessionBindingListener` 接口（请参看“标准的会话管理——`HttpSessionBindingListener`”。）

```
public class MySessionChecker implements HttpSessionBindingListener
{
    ...

    valueBound(HttpSessionBindingEvent event)
    {...}

    valueUnbound(HttpSessionBindingEvent event)
    {...}

    ...
}
```

你可以创建一个 JSP 页面用来检查，假设此页面的名字为 `centralcheck.jsp`，包含了下面的语句：

```
<jsp:useBean id="sessioncheck" class="MySessionChecker" scope="session" />
```

在任何包含了 `centralcheck.jsp` 的页面中，Servlet 容器都将在 `sessioncheck` 对象超出作用域（即会话结束）时调用在 `MySessionChecker` 类中实现的 `valueUnbound()` 方法。你可以在应用程序的每个页面的结尾包含 `centralcheck.jsp` 以进行会话资源的管理。

4.1.7 处理大规模静态内容 JSP 页面的技巧

含有大量静态内容（一般是大量的 HTML 代码，它们在运行时不改变）的 JSP 页面可能导致解释和运行过程变慢。

对于这种情况有两个主要的解决方法（每一个解决方法都将提高解释的速度）：

- 将静态的 HTML 内容放到一个单独的文件中，然后使用动态包含语句(`jsp:include`) 将它的输出在运行时包含到 JSP 页面的输出中去。要获得关于 `jsp:include` 语句的相关信息，请参看“JSP 的 Actions 和 `<jsp:>` 标签集”。

注意：在这种情况下使用静态包含命令 `<%@include...%>` 将没有任何帮助，它导致被包含文件在解释时被复制到包含页面中去，因此不会解决任何问题。

- 将静态的 HTML 内容放到一个 Java 资源文件中。
如果你允许了 `external_resource` 配置参数，那么 OracleJSP 就会自动这样处理。要获得关于此参数的信息，请参看“OracleJSP 在非 OSE 环境下的配置参数”。
如果要发布到 Oracle8i，那么预解释工具 `ojspc` 的 `-extres` 和 `-hotload` 选项以及 `shell` 命令 `publishjsp` 的 `-hotload` 选项也提供了此项功能。

注意：将静态 HTML 内容放到资源文件中与上面所提到的使用 `jsp:include` 动态包含相比较而言要占用更大的内存，因为在加载页面实现类时必须加载这个资源文件。

对拥有大量静态内容的 JSP 页面来说还可能有另外一个问题，那就是绝大多数（但不是全部）的 Java 代码太多，尽管 `javac` 能编译它，但是 Java 虚拟机也不能运行它。依赖于 JSP 解释器的实现，这个限制有可能对 JSP 页面造成影响，因为从整个 JSP 页面源文件所生成的 Java 代码都被放到页面实现类的 `service` 方法里了。（JSP 页面里的静态 HTML 内容都将被解释成 Java 代码，而脚本段里的 Java 代码则被直接复制到页面实现类里。）

另一个有可能出现但概率很小的情况就是 JSP 页面的 Java 脚本段太大了以致于突破了 `service` 方法的 64k 大小限制。在这种情况下，最好把 JSP 页面中的大量 Java 代码转移到一个 `JavaBean` 中。

4.1.8 方法变量声明与成员变量声明的比较

在“脚本元素”一节中曾经讨论过，成员变量应该在 `<%!...%>` 中声明，而方法变量必须在 `<%...%>` 中声明。

对你的每个变量声明请注意使用合适的机制，这也取决于你如何使用变量：

- 使用 `<%!...%>` JSP 语法来声明的变量是页面生成类的成员变量，它具有类层次的生命周期，标签由 JSP 解释器所生成。
- 使用 `<%...%>` JSP 脚本语法来声明的变量是页面生成类的 `service` 方法中的局部变量。

考虑下面的一个例子——`decltest.jsp`：

```
<HTML>
<BODY>
<% double f2=0.0; %>
<%! double f1=0.0; %>
Variable declaration test.
</BODY>
</HTML>
```

这将导致页面生成类的代码，如下所示：

```
package ...;
import ...;

public class decltest extends oracle.jsp.runtime.HttpJsp {

    ...
    // ** Begin Declarations
    double f1=0.0;                // *** f1 declaration is generated here ***
    // ** End Declarations
    public void _jspService
        (HttpServletRequest request, HttpServletResponse
response)
        throws IOException, ServletException {
        ...

        try {
            out.println( "<HTML>");
            out.println( "<BODY>");
            double f2=0.0; // *** f2 declaration is generated here ***
            out.println( "");
```

```

        out.println( "");
        out.println( "Variable declaration test.");
        out.println( "</BODY>");
        out.println( "</HTML>");
        out.flush();
    }
    catch( Exception e) {
        try {
            if (out != null) out.clear();
        }
        catch( Exception clearException) {
        }
        finally {
            if (out != null) out.close();
        }
    }
}

```

注意：此处的代码知识是用来作为演示概念的目的而使用的，有许多类被删掉了，因此实际由 OracleJSP 所生成的页面实现类与实际的代码是不同的。

4.1.9 page 指令的特征

本节讨论了 page 指令的下面一些特征：

- page 指令是静态的并且在解释期生效；你不能在运行期指定参数设置。
- page 指令里的 import 设置是累积的。

page 指令是静态的

page 指令是静态的，它在解释期被处理，你不能在运行期指定动态的设置来被解释，参考下面的例子：

例子一：下面的 page 指令是有效的

```
<%@ page contentType="text/html; charset=EUCJIS" %>
```

例子二：下面的 page 指令是无效的，并且将导致一个错误出现（因为 S 是一个动态参数，要在运行时才能被动态地确定）

```
<% String s="EUCJIS"; %>
```

```
<%@ page contentType="text/html; charset=<%=s%>" %>
```

对 page 指令的一些属性的动态设置还是有可以解决的办法的，重新考虑例子二，有一个方法叫做 SetContentType() 可以用来动态地设置内容类型的属性，请参看“动态内容类型设置”。

page 指令里的 import 设置是累积的

在一个 JSP 页面里，使用 page 指令的 import 设置可以一次导入多个包，也可以多次累积设置。

在任何单个的 JSP 页面里，下面的两个例子都是等价的：

```
<%@ page language="java" %>
```

```
<%@ page import="sqlj.runtime.ref.DefaultContext, java.sql.*" %>
```

或

```
<%@ page language="java" %>
```

```
<%@ page import="sqlj.runtime.ref.DefaultContext" %>
<%@ page import="java.sql.*" %>
```

在第一个 `page` 指令的 `import` 之后，第二个 `page` 指令的 `import` 在已经导入的类的基础之上增添新的导入类（或包），也就是说 `import` 设置是可以累积的，而不是用后来导入的类覆盖掉前面导入的类。

4.1.10 JSP 对空白字符的保留以及二进制数据的使用

OracleJSP（包括其他通用的 JSP 实现）保留源代码中的空白字符，包括回车、换行符等，并将它们输出到浏览器。但实际上这些空白字符可能是开发人员并不希望输出的，并且使得 JSP 技术在生成二进制数据方面是一个很坏的选择。

空白字符实例

下面的两个 JSP 页面生成不同的 HTML 输出，只因为源代码中对回车的用法不同。

例子一：——没有回车（`nowhisp.jsp`）

下面的 JSP 页面在 `Date()` 和 `getParameter()` 方法调用后没有使用回车。（第三行和第四行实际上是由一行代码组成的。）

```
<HTML>
<BODY>
<%= new java.util.Date() %> <% String user=request.getParameter("user");
%> <%=
(user==null) ? "" : user %>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

它所对应的浏览器的 HTML 输出如下：

```
<HTML>
<BODY>
Tue May 30 20:07:04 PDT 2000
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

例子二：——使用回车（`whisp.jsp`）

下面的 JSP 页面在 `Date()` 和 `getParameter()` 方法调用后使用了回车：

```
<HTML>
<BODY>
<%= new java.util.Date() %>
<% String user=request.getParameter("user"); %>
```

```

<%= (user==null) ? "" : user %>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>

```

它所对应的浏览器的 HTML 输出如下：

```

<HTML>
<BODY>
Tue May 30 20:19:20 PDT 2000

<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>

```

请注意在日前和“Enter name:”之间的空行。在上面所举的两个例子中空白符所造成的差异并不是很重要的，因为每个例子在浏览器中的外观都是相同的。然而在有些情况下可能会造成很显著的差异，本部分的讨论只不过是从小意义上演示了保留空白字符，上面的例子所对应的浏览器输出如下图：



避免在 JSP 页面中使用二进制数据的原因

因为如下原因导致了 JSP 技术在生成二进制数据方面是一个很坏的选择，通常情况下你应该使用 Servlets 来代替它：

- JSP 技术的设计目标就不是用来处理二进制数据的——在 JspWriter 对象中没有合适的方法用来输出二进制字节。
- 在运行过程中，JSP 目前将保留字符，而空白字符在某些时候是不希望出现的，因此使得 JSP 页面在生成二进制数据并输出到浏览器（例如一个.gif 文件）时是一个很坏的选择。

参看下面的例子：

```

...
<% out.getOutputStream().write(... binary data...) %>
<% out.getOutputStream().write(... more binary data...) %>

```

在这个例子中，浏览器将收到一个并不希望出现的回车字符，取决于你的输出缓冲的机制，这个回车字符可能在二进制数据的中间出现，也可能在二进制数据的结束处出现。当然你可以在代码的行与行之间不加回车来避免这个问题，但这显然是一个很不好的编程风格。

试图在 JSP 页面中生成二进制数据极大地违背了 JSP 技术的出发点，它的主要设计目标是用来简化动态内容的编程的。

4.2 关键的 OracleJSP 配置问题

本节讨论了如何设置关键的 `page` 指令参数以及 OracleJSP 配置参数等重要问题。讨论的内容着力于 JSP 页面优化、类路径以及类加载器等问题，包括以下主题：

- JSP 运行优化
- 类路径和类加载的问题（非 OSE 环境）

4.2.1 JSP 运行优化

你可以考虑使用下面的设置来优化 JSP 运行性能：

- 禁止缓冲机制
- 不检查页面是否需要重解释（非 OSE 环境）（“developer mode”）
- 不使用 HTTP 会话

禁止缓冲机制

在缺省情况下，JSP 页面使用一片内存区域作为一个页面缓冲区。这个缓冲区的缺省大小是 8KB，用来支持动态的 NLS 内容类型设置，导向以及错误页面等概念。如果你不需要使用这些功能，可以通过下面的 `page` 指令来禁止缓冲：

```
<%@ page buffer="none" %>
```

这将在一定程度上提升程序的性能，因为它减少了内存的使用，并且节省了一个输出的步骤（输出直接送给浏览器，而不像以前那样先送到缓冲区再送到浏览器）。

不检查页面是否需要重解释（非 OSE 环境）

当 OracleJSP 运行一个 JSP 页面时，缺省情况下它会检查此页面的实现类是否存在，并将页面实现类（.class 文件）的修改日期与页面源文件（.jsp 文件）的修改日期作比较，如果.class 文件已经过时的话，OracleJSP 将会重新解释此文件。

如果日期比较并不是很必要的话（比如在一个典型的发布环境下，源代码已经固定不会改变），你可以通过禁止 OracleJSP 的 `developer_mode` 标记（设置 `developer_mode=false`）来避免日期的比较，这样会提高程序的性能。

`developer_mode` 的缺省值是 `true`，要获得关于在 Apache/Jserv，JSDK 以及 Tomcat 环境下如何设置 `developer_mode` 这个标志的信息，请参看“OracleJSP 的配置参数设置”。

不使用 HTTP 会话

如果一个页面不需要使用 HTTP 会话（也就是说，不需要存储或接收会话属性），那

么可以通过下面的 `page` 指令来避免使用会话：

```
<%@ page session="false" %>
```

这将从一定程度上提升页面的运行性能，因为它清除了创建会话以及检索会话所耗费的资源。

应该注意的是，一个 `Servlet` 在缺省情况下不使用会话，而一个 `JSP` 页面在缺省情况下将使用会话，要获得相关的背景信息，请参看“`Servlet` 会话”。

4.2.2 类路径和类加载器的问题（非 OSE 环境）

`OracleJSP` 使用它自己的类路径，而不使用 `Web Server` 的类路径，并且它拥有自己的类加载器。在缺省情况下，`OracleJSP` 使用自己的类加载器从自己的类路径中去加载类，这一机制具有明显的优点，但是也有很多缺点。

`OracleJSP` 的类路径包含了下面的元素：

- `OracleJSP` 的缺省类路径
- 通过 `OracleJSP` 类路径配置参数指定的附加的类路径

如果你希望某个类型被 `OracleJSP` 的类加载器所加载，而不是被系统的类加载器所加载，那么你可以使用 `OracleJSP` 的类路径配置参数来设置，或者将这个类放进 `OracleJSP` 缺省的类路径中。要得到相关的信息，请参看“`OracleJSP` 类加载器的优点与缺点”。

`OracleJSP` 缺省的类路径

`OracleJSP` 在 `Web Server` 的目录结构上定义了它自己缺省的类路径，所有它需要的 `.class` 文件和 `.jar` 文件都会在这个路径下查找。即使在没有任何 `Web Server` 类路径配置信息的情况下，`OracleJSP` 也能通过缺省的类路径来找到所需的文件。

`OracleJSP` 的缺省类路径如下所示，它们都是相对于应用程序根目录的相对路径：

```
/WEB-INF/classes  
/WEB-INF/lib  
/_pages
```

注意：如果你希望在 `WEB-INF` 目录下的类不是被 `OracleJSP` 的类加载器加载，而是被系统的类加载器加载，那么你可以将这些类也同时放到 `Web Server` 的类路径下。系统的类加载器具有更高的优先权——任何同时被设置在系统类路径和 `OracleJSP` 类路径下的类都将被系统的类加载器所加载。

`/WEB-INF/classes` 目录用来存放单个的 `Java` 类文件（`.class` 文件），这些类文件应该根据 `Java` 包的命令规则被存储到相应的子目录下。

例如，考虑一个名字为 `LottoBean` 的 `JavaBean`，假设它的代码在 `oracle.jsp.sample.lottery` 包中定义，那么它在 `/WEB-INF/classes` 目录下的存放目录为：

```
/WEB-INF/classes/oracle/jsp/sample/lottery/lottoBean.class
```

`/WEB-INF/lib` 目录是用来存放 `.jar` 文件的，因为 `Java` 包结构是在 `.jar` 文件的结构中指定的，因此 `.jar` 文件可以直接存储在 `/WEB-INF/lib` 的目录下（不需要子目录）。

继续考虑上面的例子，`lottoBean.class` 所对应的 `.jar` 文件是 `lottery.jar`，那么 `lottery.jar` 文件所存放的目录为：

/WEB-INF/lib/lottery.jar

请注意，上面的例子中所使用的路径都是相对于应用程序根目录的相对路径，而应用程序根目录的位置是要取决于特定的 Web Server 和 Servlet 环境，下面列出了一些常见的应用程序根目录的位置：

- 应用程序被映射到的 Web Server 上的目录
- Web Server 的文档根目录
- 包含 globals.jsa 文件的目录（适用于 Servlet 2.0 环境）

注意：

- 一些 Web Server，尤其是支持 Servlet 2.0 标准的 Web Server 模样提供完整的应用程序支持，比如 Servlet 环境功能等。在这种情况下，或者当应用程序映射没有被使用时，缺省的应用程序就是 Server 本身，并且应用程序的根目录就是 Web Server 的文档根目录。
- 对旧的 Servlet 环境来说，globals.jsa 文件是一个 Oracle 扩展，用来作为一个应用程序标记器以建立一个应用程序根目录。要获得相关信息，请参看“OracleJSP 对 Servlet 2.0 应用程序和会话的支持”。

OracleJSP 类路径配置参数

使用 OracleJSP 类路径配置参数可以将需要的类添加进 OracleJSP 的类路径中去。

要获得关于如何在 Apache/JServ、JSWDK 以及 Tomcat 环境中设置这些参数的信息，请参看“OracleJSP 的配置参数设置”。

OracleJSP 类加载器的优点与缺点

使用 OracleJSP 类加载器将导致如下的优点和缺点：

- 从其他类加载器所加载的类对从 OracleJSP 类加载器所加载的访问将会受到限制。当一个类被 OracleJSP 类加载器所加载时，它的定义只存在于 OracleJSP 类加载器里，因此导致从系统或者任何其他类加载器所加载的类对它的访问受到限制，不能调用它的方法。这一机制可能是你期望的，也可能是你所不期望的，这完全取决于你的应用条件。

- 自动类重载

缺省情况下，当一个类文件或者 JAR 文件从它上次被加载以后有所改动时，OracleJSP 的类加载器将自动地重新加载它。（例如，对一个 JSP 页面来说，当它的页面实现类的.class 文件比它的源文件（.jsp 文件）更老的话，此页面就会被动地解释。）

这一机制通常只在开发环境下是一个优点，它在一个典型的发布环境下，页面源代码，类文件以及 JAR 文件都不会改变，此时再检查并比较它们是相当低效的一种做法。

在发布环境下通常应该遵循这样一个原则：不要使用 OracleJSP 类路径。缺省情况下，类路径参数是空的。

4.3 OracleJSP 运行时考虑（非 OSE）

本节描述了运行时 OracleJSP 重解释页面、重加载页面以及重加载的条件，所涉及的内容不适用于运行在 Oracle Servlet Engine 环境下的 JSP 页面。

4.3.1 动态页面重解释

当一个网络应用程序正在运行时，OracleJSP 容器在缺省情况下会将页面源代码被修改过的 JSP 页面重新解释并加载。

OracleJSP 首先会检查页面实现类文件的上次修改事件（在 OracleJSP 内存缓冲里记录），然后检查 JSP 页面源文件的上次修改时间，如果发现页面源文件比较新的话，就会重新解释并加载这个 JSP 页面。

为了避免这种检查所带来的性能上的损失，你可以将 `developer_mode` 标记设为 `false`。在发布环境下，页面源代码文件和类文件都不会改变，此时设置 `developer_mode=false` 将会带来很大的性能提升。

注意：

- 因为使用内存里的值来记录页面实现类文件的上次修改日期，所以当从文件系统中删除一个页面实现类时，将不会导致它所对应的 JSP 页面被重新解释。OracleJSP 将只对那些文件日期改变的 JSP 页面源文件继续重新解释。
- 当内存里所记录的缓冲值丢失时，页面实现类文件将被重新生成。这通常发生在当服务器重新启动后或者此应用程序中的另外一个页面被重新解释后有一个请求到达此页面。

4.3.2 动态页面重载

在下列情况下，OracleJSP 容器将自动地重新加载一个 JSP 页面（或者用另外一个词来说，重新加载生成的页面实现类）：

- 此页面被重新解释
（请参看上一节“动态页面重解释”）
- 此页面所调用的一个 Java 类由 OracleJSP 的类加载器所加载（不是由系统的类加载器所加载），并且被修改过。
（请参看下一节“动态类重载”）
- 任何一个应用程序里的页面被重新加载
每个 JSP 页面都与一个网络应用程序相关联，它在此应用程序里运行。（即使 JSP 页面与任何一个特定的应用程序都模样关联，它也被认为是和一个“缺省应用程序”相关联。）

当一个 JSP 页面被重新加载时，同一个应用程序里的 JSP 页面都被重新加载。

注意：

- 当只有一个被解释静态包含的文件改变时，OracleJSP 不会重新加载页面。

(通过`<%@include%>`语法来静态包含的文件在解释期才被插入。)

- 页面重载和页面重解释不是同一概念，重载不一定意味着要重解释。

4.3.3 动态类重载

在缺省情况下，在 OracleJSP 会发一个将要执行由 OracleJSP 类加载器所加载的类的请求之前，它会首先检查这个类文件自从首次被加载以后是否被修改过，如果被修改过的话，OracleJSP 类加载器就会重新加载这个类。

这个机制只适用于在 OracleJSP 类路径里所指定的类，包括下面几种：

- 在 WEB-INF/lib 目录里的 JAR 文件
- 在 WEB-INF/classes 目录里的 class 文件
- 通过 OracleJSP 类路径配置参数所指定的路径下的类
- 在 _page 输出目录下生成的.class 文件

正如上一节“动态页面重载”里所提到的那样，重新加载一个类导致重新加载，所以引用这个类的 JSP 页面。

注意：

- 请注意要想实现动态重载概念。类必须放在 OracleJSP 的类路径里，而不能放在系统的类路径里。如果类也同时在系统的类路径里，那么系统的类加载器就有比较高的优先权，有可能破坏 JSP 的自动重载概念。
- 动态类重载从 CPU 使用率的角度上来讲是一个很费时的操作，你可以通过将 developer_mode 参数设置成 false 来禁止这一特性，这对发布环境来说尤其有效，因为此时类是不期望被改变的。要获得关于类路径和 developer_mode 配置参数以及如何设置它们的信息。请参看“OracleJSP 在非 OSE 环境下的配置参数”和“OracleJSP 的配置参数设置”。

4.4 Oracle Servlet Engine 环境应考虑的问题

Oracle Servlet Engine (OSE) 被集成在 Oracle8i JServer 环境里，为了在 OSE 下运行 JSP 页面，必须首先将页面加载并发布到数据库中。关于如何在 JSP 页面发布到 Oracle8i 数据库里的详细讨论，请参看第 6 章“JSP 的解释和发布”。本节只讨论在 OSE 环境下所应作的特殊的编程考虑，并且提供了 OSE 关键特征的概述。

通过使用 Oracle HTTP Server 作为前台 Web Server，OSE 能够运行 JSP 引用程序。尽管 OSE 本身也可以直接作为 Web Server 来使用，但一般建议使用 Oracle HTTP Server，要获得相关信息，请参看“Oracle 网络应用程序的数据库访问”。在安装 Oracle8i 8.1.7 发行版的过程中，Oracle HTTP Server 就自动地设置为缺省的 Web Server，如果要改变这一缺省的设置，请参考 Oracle 安装手册。

运行在 Oracle Servlet Engine 环境下的 JSP 页面一般都比认为要访问数据库，所以这里提供了有关 JServer 环境下数据库连接的背景知识。

JSP 代码一般情况下是完全可以在 OSE 和其他能运行 OracleJSP 的环境下相互移植的，

但是如果在 JServer 里通过 JDBC 服务器端内部的驱动来连接数据库，那么情况就有所不同了（这时数据库连接不需要连接字符串。），相关的讨论可以参考“JServer 连接”。

除了使用 JServer 数据库连接代码或者其他 JServer 特定的代码以外，在 OSE 环境下所写的 JSP 页面都可以移植到其他 OracleJSP 的环境下，并且仅仅当使用了 JServer 特定的功能时，JSP 代码才需要修改。

本节包含了下面的内容：

- JServer JVM 和 JDBC 服务器端内部驱动简介
- JServer 连接
- Oracle Servlet Engine 所使用的 JNDI
- OracleJSP 运行时配置参数的等价代码

注意：本节主要考虑 OSE 环境下开发时所应考虑的问题，对发布环境下应考虑的问题，包括类的热加载以及客户端解释及服务器端解释的比较等，请参看“发布过程中的逻辑与特性概述”。

4.4.1 JServer JVM 和 JDBC 服务器端内部驱动简介

每个 Oracle8i 的 JServer 数据库会话都调用它自己的 Java 虚拟机（JVM），也就是说在会话和 JVM 之间存在着一对一的对应关系，意识到这一点是很重要的。

任何可以 Oracle8i 数据库为目标平台并且运行在 JServer 下的 Java 程序一般都使用 JDBC 服务器端内容的驱动去访问本地的 SQL 引擎。内部 JDBC 驱动与 Oracle8i 数据库和 JVM 紧密地联系在一起，它与数据库运行在同一进程中，并且运行在缺省的数据库会话（与调用 JVM 相同的会话）中。

服务器端内部的驱动被专门做了优化以运行于数据库里面，并且可以对本地数据库里面的 SQL 数据和 PL/SQL 程序作直接的访问。整个 JVM 与数据库和 SQL 引擎运行在同一个地址空间中，对 SQL 引擎的访问只需通过一个本地函数调用——无需通过网络。这就增强了 JDBC 程序的性能，并且通过远程的 Net8 调用来访问数据库快得多。

4.4.2 JServer 连接

因为 JDBC 服务器端内部的驱动在一个缺省的数据库会话里运行，因此你已经隐式地连接到了一个数据库。你可以通过下面所列的两个 JDBC 方法来访问缺省的数据连接。

- 使用 Oracle 专用的 `defaultConnection()` 方法。（它是 `OracleDriver` 类的一个方法，并且每次调用都返回相同的连接对象。）
- 使用静态的 `DriverManager.getConnection()` 方法，它的输入参数为一个 URL 字符串，可以是 `jdbc:oracle:kprb` 或者 `jdbc:default:connection`。（这个方法每次被调用时返回不同的连接对象。）

一般情况下建议使用 `defaultConnection()` 方法。

另外也可以使用服务器端 Thin 驱动来作为内容的连接（连接到运行你的 Java 代码的数据库），但是这不是一种典型的情况。

注意:

- 作为可选择的一种方法,你可以使用 OracleJSP 所提供的定制 JavaBeans 来继续数据库连接。要获得相关信息,请参看“Oracle 的数据库访问 JavaBeans”。
- 使用服务器端内部驱动来建立数据库连接不需要注册 OracleDriver,但是如果注册了也不会有上述坏处。不管你使用 getConnection() 还是 defaultConnection() 方法来继续连接,这条规则都是适用的。

使用 OracleDriver 类的 defaultConnection()方法来连接

OracleDriver 类位于 oracle.jdbc.driver 包中,它的 defaultConnection()方法是一个 Oracle 扩展,可以用来继续一个内部数据库连接。使用此方法总是返回相同的连接对象。即使你多次调用这个方法,并将返回的连接对象赋给多个不同的变量,也只有一个连接对象被重复使用。

defaultConnection()方法不需要连接字符串作为参数。下面是一个例子:

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

class JDBCConnection
{
    public static Connection connect() throws SQLException
    {
        Connection conn = null;
        try {
            // connect with the server-side internal driver
            OracleDriver ora = new OracleDriver();
            conn = ora.defaultConnection();
        }
        catch (SQLException e) {...}
        return conn;
    }
}
```

请注意在这个例子中并没有调用 conn.close()方法,这是因为当 JDBC 代码在目标服务器内部运行时,数据库连接只是一个内部的数据通道,而不是一个显式的 connection 实例(从客户端连接时),所以到最后就无需关闭。

如果你调用了 close()方法,那么应该意识到下面的事实:

- 所有通过 defaultConnection()方法获得的连接实例(它们实际上都指向同一个连接对象),都被关闭并且在将来也不可用,它们的状态和资源也被清空。在此以后如果再调用 defaultConnection()方法就将导致一个新的连接对象被使用并且因此产生一个新的事务。
- 即使连接对象被关闭了,到数据库的内部的连接并没有被关闭。

使用 DriverManager.getConnection()方法来连接

通过调用 defaultConnection()方法可以建立一个内部的数据库连接,你还可以使用

`DriverManager.getConnection()`方法来代替它。`DriverManager.getConnection()`方法的输入参数是一个连接字符串，它的形式可以为下面两种之一：

```
Connection conn = DriverManager.getConnection("jdbc:oracle:kprb:");
```

或者

```
Connection conn = DriverManager.getConnection("jdbc:default:connection:");
```

在 URL 字符串里所指定的任何用户名及密码都将被忽略，因为这是连接到数据库的缺省连接。每次调用 `DriverManager.getConnection()`方法都将返回一个新的 Java 连接对象。请注意，尽管这个方法并没有创建一个新的物理连接（只有一个内部的连接被使用），但是它返回一个新的对象。

如果你正在使用被称为“类型映射”的对象映射，那么 `DriverManager.getConnection()`方法在每次被调用时都返回一个新的连接对象，这一事实是很重要的。

一个类型映射可以将 Oracle 的 SQL 对象类型映射到 Java 类上，并且它与一个指定的连接对象以及此对象的状态之间有一种关联关系。如果你在自己的程序中想要使用多个类型映射时，就可以为每个类型映射调用 `getConnection()`方法并得到一个新的连接对象。

使用服务器端 Thin 驱动来连接

一般情况下，Oracle JDBC 服务器端 Thin 驱动适合用来从一个数据库连接到另一个数据库。不过使用服务器端 Thin 驱动来进行内部的连接也是可以的，你可以像使用任何 Oracle JDBC Thin 驱动一样来指定一个连接字符串。

使用服务器端 Thin 驱动的优点是 JSP 代码可以很容易地在 Oracle Servlet Engine 和其他 Servlet 环境之间移植，然而服务器端内部驱动提供了更为高效的性能。

在服务器端内部驱动中没有自动提交机制

在服务器端内部驱动中，JDBC 自动提交特性被禁止了，因此你必须手工提交或者回滚所做的改变。

在服务器端内部驱动中没有连接池或连接缓冲机制

当使用服务器端内部驱动时，连接池和缓冲是无法应用的，因为此时只有一个内部的数据库连接。任何试图通过内部驱动使用这些特性的做法都将导致实际性能的下降。

4.4.3 Oracle Servlet Engine 所使用的 JNDI

Oracle Servlet Engine 使用 JNDI 机制去查找已经发行的 JSP 页面和 Servlets，但是这个机制在一般情况下对 JSP 开发者或用户来说是不可见的。发行一个 JSP 页面一般是在将 JSP 页面发布到 OSE 的过程中来完成的，可以通过调用 Oracle session-shell 的 `publishjsp` 命令（服务器端解释）或者 `publishservlet` 命令（客户端解释）来实现。

`publishservlet` 命令需要你指定页面实现类的虚拟路径名和 Servlet 名，虚拟路径名被用来通过一个 URL 来调用页面，或者在其他任何运行在 OSE 下的页面中包含或者导向到此页面。

`publishjsp` 命令可以通过在命令行指定虚拟路径名和 Servlet 名，也可以直接从指定的 JSP 源文件的名称以及目录路径中得到信息。

虚拟路径名和 Servlet 名都将进入到 JServer 的 JNDI 名字空间，但是 JSP 开发人员或用户只需要了解虚拟路径名就可以了。

要获得关于在 OSE 环境下发行 JSP 页面的更多信息，请参看“在 Oracle8i 中解释并发行 JSP 页面 (publishjsp)”（适用于服务器端解释的发布）或“在 Oracle8i 中发行解释后的 JSP 页面 (publishservlet)”（适用于客户端解释的发布）。

要获得关于 Oracle Servlet Engine 如何使用 JNDI 的一般信息，请参看“Oracle8i Oracle Servlet Engine 用户指南”。

4.4.4 Oracle 运行时配置参数的等价代码

一些 OracleJSP 的配置参数在解释时生效，而另外一些则在运行时生效。当你在 Oracle Servlet Engine 下将 JSP 页面发布到 Oracle8i 数据库中时，你可以通过 OracleJSP 的预解释工具的命令行选项来进行合适的解释时参数设置。

但是 Oracle Servlet Engine 不支持运行时的参数配置，最重要的运行时参数是 translate_params，它与 NLS 相关。要获得关于它的等价代码的讨论信息，请参看“translate_params 配置参数的等价代码”。

4.5 Apache/Jserv Servlet 环境应考虑的问题

在基于 Apache/Jserv 平台下运行 OracleJSP 时有一些应该特殊考虑的问题，这些平台还包括 Oracle Internet Application Server 1.0.x 发行版，因为它是一个 Servlet 2.0 环境。Servlet 2.0 标准缺乏对一些关键特性的支持，而这些特性在 Servlet 2.1 或 Servlet 2.2 环境下都是可用的。

要获得有关如何在 Apache/Jserv 环境下配置 OracleJSP 的信息，请参看下面的一些章节。

- “在 Web Server 的类路径里添加与 OracleJSP 相关的 JAR 和 ZIP 文件”。
- “在 JSP 扩展名与 OracleJSP 之间建立关联”。
- “在 Apache/Jserv 环境下设置 OracleJSP 参数”。

（如果你通过一个 Oracle 平台来使用 Apache/Jserv，那么请参看特定平台所提供的安装和配置文档。）

本节的剩余部分首先总结在 Oracle Internet Application Server 中 Apache/Jserv 的使用，然后讨论下面一些 Apache 专有的需要考虑的问题：

- Apache/Jserv 下的动态包含与前进
- Apache/Jserv 的应用程序框架
- JSP 和 Servlet 会话的共享
- 目录别名解释

4.5.1 在 Oracle Internet Application Server 使用 Apache/Jserv

在 Oracle Internet Application Server 1.0.0 和 1.0.1 发行版中，它使用了 Apache/Jserv 作为它的 Servlet 环境。

与 Apache/JServ 或者其他的 Servlet 2.0 环境一样, 使用 Oracle Internet Application Server 1.0.x 发行版运行 JSP 页面和 Servlet 时, 有一些应该特殊考虑的问题, 它们将在下面的章节中详细地讨论。

(Oracle Internet Application Server 包含 Oracle HTTP Server 作为它的 Web Server。你应该认识到, 如果使用 Oracle HTTP Servlet 的 mod_ose 来在 Oracle Servlet Engine 中运行 JSP 应用程序, 那么你正在使用的是 OSE Servlet 2.2 环境, 而不是 Internet Application Server 中的 Apache/JServ Servlet 2.0 环境。)

注意: Oracle HTTP Server 和 Oracle Internet Application Server 的未来发行版本将可能使用除 Apache/JServ 以外的别的 Servlet 环境。

4.5.2 Apache/JServ 下的动态包含与前进

JSP 的动态包含 (jsp:include) 与前进 (jsp:forward) 依赖于请求分发器的功能, 而这些功能是在 Servlet 2.1 与 2.2 环境下引入的, 在 Servlet 2.0 环境下并不具备。

然而, OracleJSP 提供了扩展的功能以允许在 Apache/JServ 和其他的 Servlet 2.0 环境下从一个 JSP 页面动态包含与前进到另一个 JSP 页面或一个静态的 HTML 页面。

但是, 这一 OracleJSP 功能不允许动态包含或前进到一个 Servlet。(Servlet 的执行是由 JServ 或者其他的 Servlet 容器所控制的, 不是由 OracleJSP 容器控制。)

如果想动态包含或者前进到一个 Servlet, 你可以对这个 Servlet 创建一个 JSP 页面来封装它。

下面的例子实现了一个 Servlet 以及一个封装了此 Servlet 的 JSP 页面。在一个 Apache/JServ 环境下, 你可以通过包含或前进到这个 JSP 封装页来首先高效地包含或前进到这个 Servlet。

Servlet 代码: 假设你想包含或前进到下面的 Servlet:

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

public class TestServlet extends HttpServlet {

    public void init(ServletConfig config) throws ServletException
    {
        super.init(config);
        System.out.println("initialized");
    }
    public void destroy()
    {
        System.out.println("destroyed");
    }

    public void service
        (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
```

```

    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML><BODY>");
        out.println("TestServlet Testing");
        out.println("<H3>The local time is: " + new java.util.Date());
        out.println("</BODY></HTML>");
    }
}

```

JSP 封装页代码: 你可以创建下面的 JSP 页面 (wrapper.jsp) 来封装上面的 Servlet:

```

<!-- wrapper.jsp--wraps TestServlet for JSP include/forward --%>
<%@ page isThreadSafe="true" import="TestServlet" %>
<%!
    TestServlet s=null;
    public void jspInit() {
        s=new TestServlet();
        try {
            s.init(this.getServletConfig());
        } catch (ServletException se)
        {
            s=null;
        }
    }
    public void jspDestroy() {
        s.destroy();
    }
%>
<% s.service(request,response); %>

```

在 Servlet 2.0 环境下, 包含或者前进到 wrapper.jsp 与在 Servlet 2.1 或 2.2 环境下直接包含或者前进到 TestServlet 有着相同的效果。

注意:

- 在 JSP 封装页里将 isThreadSafe 设置成 true 还是 false 取决于是否原始的 Servlet 具有安全线程。
- 在 Servlet 2.0 环境下, 除了使用 JSP 封装页以外, 还有一种可选择的方法: 你可以在原始的 JSP 页面 (即包含 include 或者 forward 的那个页面) 中添加 HTTP 客户代码。你可以使用标准的 java.net.URL 类的实例创建一个从原始 JSP 页面到 Servlet 的 HTTP 请求。(注意在这种情况下你不能共享会话数据或安全证书), 或者你也可以使用 Innovation GmbH 中的 HTTPClient 类来作为一个替换。Oracle8i JServer 提供了这个类的修改后的版本, 当你使用 http://来作为 URL 的时候, 它可以直接或通过代理来支持 SSL。

4.5.3 Apache/JServ 应用程序框架

Servlet 2.0 标准没有提供一个完整的 Servlet 环境框架来支持应用程序, 只是在它后续

的版本中才有了这一功能。

对 Servlet 2.0 环境（包括 Apache/JServ）来说，OracleJSP 通过文件 `globals.jsa` 提供了它自己的应用程序框架，而 `globals.jsa` 文件可以认为是一个应用程序标记器。

4.5.4 JSP 和 Servlet 会话的共享

在 Apache/JServ 环境下，要在 JSP 页面和 Servlets 之间共享 HTTP 会话的信息，就必须配置你的环境以便 `oracle.jsp.JspServlet`（OracleJSP 的前端工具）与你想要的 JSP 页面与之共享到相同会话的那个 Servlet 或 Servlets 在同一个区域（zone）中。要获得更多的信息，请参看相应的 Apache 文档。

为了验证区域是否正确设置，一些浏览器允许你打开一个对 cookies 的警告信息。在一个 Servlet 环境下，cookie 名字就包含了区域的名字。

并且，对那些使用 `globals.jsa` 文件的应用程序来说，OracleJSP 的配置参数 `session_sharing` 应该被设成 `true`（缺省值），这样 JSP 会话的数据才能被 Servlets 所访问。要获得相关信息，请参看下面一些章节：

- “OracleJSP 对 Servlet 2.0 应用程序和会话的支持”
- “OracleJSP 在非 OSE 环境下的配置参数”
- “OracleJSP 的配置参数设置”

4.5.5 目录别名解释

Apache 允许你创建一个“虚拟目录”来支持目录别名，从而使得 Web 文档可以被在缺省的文档根目录之外的地方。（对 Web Server 的文档根目录和每个别名根目录来说都有一个隐式的应用程序被创建。）

要创建“虚拟目录”，可以在 `httpd.conf` 配置文件里使用 `Alias` 命令，下面是一个例子：
`Alias /icons/ "/apache/apache139/icons/"`

这个例子将使 `icons` 成为路径 `/apache/apache139/icons/` 的别名。此时对文件 `/apache/apache139/icons/art.gif` 可以通过下面的 URL 来访问：

`http:// host[: port]/icons/art.gif`

但是，当前这个功能对 Servlet 和 JSP 页面来说工作得不太正常，这是因为 Apache/JServ 的 `getRealPath()` 方法在处理一个别名目录下的文件时将返回错误的结果。

为了解决这个问题，OracleJSP 提供了一个 Apache 专用的配置参数 `alias_translation`，当将它设置成 `true` 的时候，它将正确地处理别名目录问题。（缺省值是 `false`）

要得到关于如何在 Apache/JServ 环境下设置 OracleJSP 配置参数的信息，请参看“在 Apache/JServ 环境下设置 OracleJSP 参数”。

第 5 章 OracleJSP 的扩展功能

本章讨论了 OracleJSP 所提供的扩展功能，涵盖以下主题：

- 可移植的 OracleJSP 编程扩展
- Oracle 专有的编程扩展
- OracleJSP 对 Servlet 2.0 的应用程序和会话支持

可移植的扩展功能是通过 Oracle 的 JSP Markup Language (JML) 定制标签、JML 扩展数据类型、SQL 定制标签以及数据库访问 JavaBeans 来实现的，你可以在其他 JSP 环境下使用这些功能。

不能移植的扩展功能都需要 OracleJSP 以解释并运行。

对 Servlet 2.0 环境的应用程序和会话的扩展支持是通过 Oracle 的 `globals.jsa` 机制来实现的，它也需要 OracleJSP 来解释并运行。

5.1 可移植的 OracleJSP 编程扩展

本节所讨论的 Oracle 扩展功能都是通过 Oracle JSP Markup Language (JML) 样例标签库或者定制的 JavaBeans 来实现的，这些扩展功能可以很轻松地移植到任何标准的 JSP 环境下。本节讨论以下扩展功能：

- JML 扩展数据类型
- XML 和 XSL 支持（包括 JML 标签）
- 数据库访问 JavaBeans
- JML SQL 标签

注意：要使用 JML 定制标签所提供的功能，请参看 “JSP 标记语言 (JML) 样例标签库概述”。

5.5.1 JML 扩展数据类型

为了弥补在 JSP 页面里使用 Java 简单数据类型和 `java.lang` 里的封装数据类型时所具有的缺陷（请参看 “OracleJSP 的扩展数据类型”），OracleJSP 在 `oracle.jsp.jml` 包里提供了下面的 JavaBeans 类以作为对通用 Java 数据类型的封装：

- `JmlBoolean`，代表一个布尔类型的值
- `JmlNumber`，代表一个整型的值
- `JmlFPNumber`，代表一个浮点型的值
- `JmlString`，代表一个字符串的值

上面这些类中每一个都只有一个属性：`value`，然后是对此属性处理的一些方法，包括从多种格式的输入中得到这个值，设置这个值，测试这个值是否与另外一个指定的值相等，将这个值转化成字符串等。

另外，除了使用 `getValue()` 和 `setValue()` 方法，你也可以选择使用 `jsp:getProperty` 和

jsp:setProperty 标签，它们的用法与其他标准 JavaBeans 中的用法都相同。

下面的例子创建了一个 JmlNumber 的实例，名字叫 count，作用域为应用程序范围的作用域。

```
<jsp:useBean id="count" class="oracle.jsp.jml.JmlNumber" scope="application" />
```

然后就可以对 count 的值来进行设置，标签可以通过下面的例子来得到它的值：

```
<h3> The current count is <%=count.getValue() %> </h3>
```

下面的例子创建了一个 JmlNumber 的实例，名字叫做 maxsize，作用域为请求作用域，并且通过 jsp:setProperty 标签来设置它的值：

```
<jsp:useBean id="maxSize" class="oracle.jsp.jml.Number" scope="request" >
  <jsp:setProperty name="maxSize" property="value" value="<%= 25 %>" />
</jsp:useBean>
```

本节的剩余部分将列出这四种扩展数据类型的公有方法，然后通过一个例子来说明它们的用法。

JmlBoolean 数据类型

一个 JmlBoolean 型的对象代表着一个 Java 布尔型的值。

getValue()和 setValue()方法用来取得或者设置 value 属性的值，它们是通过布尔型的参数来传递数据的。下面是这两个方法的原型：

- boolean getValue()
- void setValue(boolean)

setTypedValue()方法有几种不同的形式可用，它可以将一个字符串（如“true”或者“false”）、一个 java.lang.Boolean 型的值、一个 Java 布尔型的值或者一个 JmlBoolean 型的值赋给 value 属性。对字符串输入来说，此方法按照与标准的 java.lang.Boolean.valueOf()方法相同的规则将输入字符串转换成布尔型的值。下面是这个方法的原型：

- void setTypedValue(String)
- void setTypedValue(Boolean)
- void setTypedValue(boolean)
- void setTypedValue(JmlBoolean)

equals()方法用来测试 value 属性的值是否与一个指定的 Java 布尔型的值（输入参数）相等，下面是它的原型：

- boolean equals(boolean)

typedEquals()方法与 setTypedValue()方法很相像，它也有几种不同的形式，用来测试 value 属性的值是否与下面几种形式的值相等：一个字符串（如“true”或者“false”）、一个 java.lang.Boolean 型的值或者一个 JmlBoolean 型的值。此方法的原型如下所示：

- boolean typedEquals(String)
- boolean typedEquals(Boolean)
- boolean typedEquals(JmlBoolean)

toString()方法将 value 属性的值转换成一个 java.lang.String 型的值（要么是“true”，要么是“false”），并且返回这个字符串。它的原型如下所示：

- String toString()

JmlNumber 数据类型

一个 JmlNumber 型的对象代表着一个 32 位的整数值，它与 Java 的 int 数据类型等价。

getValue()和 setValue()方法用来取得或者设置 value 属性的值，它们是通过 Java 整型的参数来传递数据的。下面是这两个方法的原型：

- int getValue()
- void setValue(int)

setTypedValue()方法有几种不同的形式可用，它可以将一个字符串（如“1234”）、一个 java.lang.Integer 型的值、一个 Java 整型的值或者一个 JmlNumber 型的值赋给 value 属性。对字符串输入来说，此方法按照与标准的 java.lang.Integer.decode()方法相同的规则将输入字符串转换成整型的值。下面是这个方法的原型：

- void setTypedValue(String)
- void setTypedValue(Integer)
- void setTypedValue(int)
- void setTypedValue(JmlNumber)

equals()方法用来测试 value 属性的值是否与一个指定的 Java 整型的值（输入参数）相等，下面是它的原型：

- boolean equals(int)

typedEquals()方法与 setTypedValue()方法很相像，它也有几种不同的形式，用来测试 value 属性的值是否与下面几种形式的值相等：一个字符串（如“1234”）、一个 java.lang.Integer 型的值或者一个 JmlNumber 型的值。此方法的原型如下所示：

- boolean typedEquals(String)
- boolean typedEquals(Integer)
- boolean typedEquals(JmlNumber)

toString()方法将 value 属性的值转换成一个 java.lang.String 型的值（如“1234”），并且返回这个字符串。此方法的功能与标准的 java.lang.Integer.toString()方法的功能是相同的。它的原型如下所示：

- String toString()

JmlFPNumber 数据类型

一个 JmlFPNumber 型的对象代表着一个 64 位的浮点数值，它与 Java 的 double 数据类型等价。

getValue()和 setValue()方法用来取得或者设置 value 属性的值，它们是通过 Java 的 double 型的参数来传递数据的。下面是这两个方法的原型：

- double getValue()
- void setValue(double)

setTypedValue()方法有几种不同的形式可用，它可以将一个字符串（如“3.57”）、一个 java.lang.Integer 型的值、一个 Java 整型的值、一个 java.lang.Float 型的值、一个 Java float

型的值、一个 `java.lang.Double` 型的值、一个 `Java double` 型的值或者一个 `JmlFPNumber` 型的值赋给 `value` 属性。对字符串输入来说，此方法按照与标准的 `java.lang.Double.valueOf()` 方法相同的规则将输入字符串转换成浮点类型的值。下面是这个方法的原型：

- `void setTypedValue(String)`
- `void setTypedValue(Integer)`
- `void setTypedValue(int)`
- `void setTypedValue(Float)`
- `void setTypedValue(float)`
- `void setTypedValue(Double)`
- `void setTypedValue(double)`
- `void setTypedValue(JmlFPNumber)`

`equals()` 方法用来测试 `value` 属性的值是否与一个指定的 `Java double` 型的值(输入参数)相等，下面是它的原型：

- `boolean equals(double)`

`typedEquals()` 方法与 `setTypedValue()` 方法很相像，它也有几种不同的形式，用来测试 `value` 属性的值是否与下面几种形式的值相等：一个字符串(如“3.57”)、一个 `java.lang.Integer` 型的值、一个 `Java int` 型的值、一个 `java.lang.Float` 型的值、一个 `Java float` 型的值、一个 `java.lang.Double` 型的值、一个 `Java double` 型的值或者一个 `JmlFPNumber` 型的值。此方法的原型如下所示：

- `boolean typedEquals(String)`
- `boolean typedEquals(Integer)`
- `boolean typedEquals(int)`
- `boolean typedEquals(Float)`
- `boolean typedEquals(float)`
- `boolean typedEquals(Double)`
- `boolean typedEquals(double)`
- `boolean typedEquals(JmlFPNumber)`

`toString()` 方法将 `value` 属性的值转换成一个 `java.lang.String` 型的值(如“3.57”)，并且返回这个字符串。此方法的功能与标准的 `java.lang.Double.toString()` 方法的功能是相同的。它的原型如下所示：

- `String toString()`

JmlString 数据类型

一个 `JmlString` 型的对象代表着一个 `java.lang.String` 型的字符串。

`getValue()` 和 `setValue()` 方法用来取得或者设置 `value` 属性的值，它们是通过 `java.lang.String` 型的参数来传递数据的。如果 `setValue()` 方法的输入参数为 `null`，那么它将 `value` 属性设置成一个空字符串(长度为 0 的字符串)。下面是这两个方法的原型：

- `String getValue()`
- `void setValue(String)`

toString()方法与 getValue()方法的功能是相同的，将 value 属性的值作为一个字符串返回。它的原型如下所示：

- String toString()

setTypedValue()方法用来将一个指定的 JmlString 字符串的值（输入参数）赋给 value 属性。如果输入字符串为 null，那么 value 属性的值被设为一个空字符串（长度为 0 的字符串）。下面是这个方法的原型：

- void setTypedValue(JmlString)

isEmpty()方法用来测试 value 属性是否是一个空字符串（长度是否为 0）：""，它的原型如下所示：

- boolean isEmpty()

equals()方法有两种不同的形式，用来测试 value 属性的值是否与一个指定的 java.lang.String 型的值或者与一个指定的 JmlString 型的值相等，下面是它的原型：

- boolean equals(String)
- boolean equals(JmlString)

JML 数据类型实例

下面提供了一个例子，用来演示使用 JML 数据类型 JavaBeans 来管理简单的数据类型，标签说明如何使用作用域。在这个例子中声明了四个具有会话作用域的对象——每一个代表一种不同的 JML 类型。页面首先显示一个表单，允许你对四个类型中的每一个都输入一个值，当一个新值被提交后，页面显示出这个新值和前一个设定的值。在生成输出内容的处理过程中，页面用新的表单里设定的值来更新这四个会话对象的值。

例子的程序代码如下：

```
<jsp:useBean id = "submitCount" class = "oracle.jsp.jml.JmlNumber" scope =
"session" />

<jsp:useBean id = "bool" class = "oracle.jsp.jml.JmlBoolean" scope = "session" >
    <jsp:setProperty name = "bool" property = "value" param = "fBoolean"
/>
</jsp:useBean>

<jsp:useBean id = "num" class = "oracle.jsp.jml.JmlNumber" scope = "session" >
    <jsp:setProperty name = "num" property = "value" param = "fNumber" />
</jsp:useBean>

<jsp:useBean id = "fpnum" class = "oracle.jsp.jml.JmlFPNumber" scope = "session" >
    <jsp:setProperty name = "fpnum" property = "value" param = "fFPNumber"
/>
</jsp:useBean>

<jsp:useBean id = "str" class = "oracle.jsp.jml.JmlString" scope = "session" >
    <jsp:setProperty name = "str" property = "value" param = "fString" />
</jsp:useBean>

<HTML>
```



```

<HEAD>
    <META HTTP-EQUIV="Content-Type"
CONTENT="text/html;CHARSET=iso-8859-1">
    <META NAME="GENERATOR" Content="Visual Page 1.1 for Windows">
    <TITLE>OracleJSP Extended Datatypes Sample</TITLE>

</HEAD>

<BODY BACKGROUND="images/bg.gif" BGCOLOR="#FFFFFF">
<% if (submitCount.getValue() > 1) { %>
    <h3> Last submitted values </h3>
    <ul>
        <li> bool: <%= bool.getValue() %>
        <li> num: <%= num.getValue() %>
        <li> fpnum: <%= fpnum.getValue() %>
        <li> string: <%= str.getValue() %>
    </ul>
<% }

if (submitCount.getValue() > 0) { %>

    <jsp:setProperty name = "bool" property = "value" param = "fBoolean"
/>

    <jsp:setProperty name = "num" property = "value" param = "fNumber" />
    <jsp:setProperty name = "fpnum" property = "value" param = "fFPNumber"
/>

    <jsp:setProperty name = "str" property = "value" param = "fString" />

    <h3> New submitted values </h3>
    <ul>
        <li> bool: <jsp:getProperty name="bool" property="value" />
        <li> num: <jsp:getProperty name="num" property="value" />
        <li> fpnum: <jsp:getProperty name="fpnum" property="value" />
        <li> string: <jsp:getProperty name="str" property="value" />
    </ul>
<% } %>

<jsp:setProperty name = "submitCount" property = "value" value = "<%=
submitCount.getValue() + 1%>" />

<FORM ACTION="index.jsp" METHOD="POST"
ENCTYPE="application/x-www-form-urlencoded">
<P> <pre>
boolean test: <INPUT TYPE="text" NAME="fBoolean" VALUE="<%= bool.getValue() %>" >
    number test: <INPUT TYPE="text" NAME="fNumber" VALUE="<%= num.getValue() %>" >
    fpnumber test: <INPUT TYPE="text" NAME="fFPNumber" VALUE="<%= fpnum.getValue()
%>" >
    string test: <INPUT TYPE="text" NAME="fString" VALUE=" <%= str.getValue() %>" >
</pre>

```

```
<P> <INPUT TYPE="submit">

</FORM>

</BODY>

</HTML>
```

5.1.2 OracleJSP 对 XML 和 XSL 的支持

JSP 技术可以用来生成与动态 HTML 页面一样棒的动态 XML 页面。OracleJSP 提供了下面两个方面的功能以支持在 JSP 页面中使用 XML 和 XSL 技术：

- OracleJSP 的解释器包含了扩展的功能，可以识别出标准的与 XML 可互换的 JSP 语法。
- OracleJSP 提供了相应的 JML 标签以将 XSL 样式单应用到 JSP 的输出流上。

并且，Oracle8i 提供了 oracle.xml.sql.query.OracleXMLQuery 类来作为 XML-SQL 实用工具的一部分，可以在数据库查询中使用 XML 功能。这个类需要文件 xsu12.jar(对 JDK 1.2.x 来说)或 xsu111.jar(对 JDK 1.1.x 来说)，并且需要由 Oracle8i 8.1.7 发行版所提供的 OracleJSP 数据库访问 JavaBeans 以得到 XML 功能。

请参看“XML 查询——XMLQuery.jsp”，那里提供了一个使用 OracleXMLQuery 的 JSP 例子。

要获得关于 OracleXMLQuery 类和其他 XML-SQL 实用工具的特性介绍，请参看“Oracle8i 应用程序开发人员指南——XML”。

可替换 XML 的 JSP 语法

在 JSP 的标签当中有很多是不符合标准的 XML 文档语法的，如用来标记脚本段的<%...%>标签，用来声明的变量的<%!...%>标签以及用来输出表达式的<%=...%>标签等。为了解决这个问题，Sun 公司在 JSP 1.1 技术标准中定义了一些与上面所提的不符合 XML 语法的标签相等价的 JSP 标签，但是这些标签都与 XML 语句兼容。这是通过一个标准的 DTD 来实现的。你可以在一个 XML 文档的开始部分的 jsp:root 标签中来指定它。

这一功能允许你在一个 XML 创作工具中编写基于 XML 的 JSP 页面。

不过 OracleJSP 并不使用这一机制，它不直接使用 DTD，也不需要使用 jsp:root 标签。OracleJSP 的解释器包含了扩展的功能，可以识别出在标准的 DTD 中指定的可替换 XML 的 JSP 语法。表 5-1 总结了这些语法：

表 5-1 XML 可选择语法

标准的 JSP 语法	可替换 XML 的 JSP 语法
<%@ directive ...%>	<jsp:directiv directive .../>
诸如：	诸如：
<%@ page ...%>	<jsp:directiv page .../>
<%@ include ...%>	<jsp:directiv include .../>

(续表)

标准的 JSP 语法	可替换 XML 的 JSP 语法
<%! ... %> (声明语句)	<jsp:declaration> ...声明语句放在这里... </jsp:declaration>
<%= ... %> (表达式)	<jsp:expression> ...表达式放在这里... </jsp: expression >
<% ... %> (脚本段)	<jsp:scriptlet> ..代码段放在这里... </jsp: scriptlet >

诸如 `jsp:useBean` 这样的 JSP 动作标签大部分已经使用了与 XML 相兼容的语法，不过因为调用规则不同或为了得到请求时属性表达式，一些必要的改变也是可能的。

XSL 样式单所对应的 JML 标签

许多利用 XML 和 XSL 技术来得到动态页面的应用程序，在输出结果被返回到客户端之前都需要在服务器上应用 XSL 变换。

为了简化这一过程，OracleJSP 提供了两个功能相同的 JML 标签——`transform` 标签和 `styleSheet` 标签，你可以使用它们两个当中的任何一个来得到动态页面的效果。下面是一个使用 `<jml:transform>` 标签的简单例子：

```
<jml:transform href="xslRef" >
```

```
... Tag body contains regular JSP commands and static text that  
produce the XML code that the stylesheet is to be applies to...
```

```
</jml:transform >
```

(jml:前缀是惯例用法，但是你也可以在你的 taglib 指令中指定任意的前缀名)

注意：如果你准备使用任何一个 JML 标签，请参看“JSP 标记语言 (JML) 样例标签库概述”。

关于 href 参数请注意下面一些问题：

- 它可以指向一个静态的 XSL 样式单或者一个动态生成的 XSL 样式单。例如，它可以指向一个 JSP 页面或者一个生成样式单的 Servlet。
- 它可以是一个全路径的 URL (`http://host[:port]/yourpath`)、一个相对于应用程序的 JSP 页面 (以 “/” 开始) 或者相对于页面的 JSP 页面 (不以 “/” 开始)。要获得关于相对于应用程序或者相对于页面的路径的信息，请参看“间接请求 JSP 页面”。
- 它可以动态地指定。缺省情况下，href 的值是一个静态的 Java 字符串，但是你可以使用标准的 JSP 表达式语法以提供一个动态被计算的值。

在比较典型的应用场合中，你将使用 `transform` 或者 `styleSheet` 标签对整个页面来进行变换。但是这两个标签只对它的正文区，也就是在开始和结束标签之间的区域生效，因此

你可以在一个页面里使用多个不同的 XSL 块, 每个块都用它自己的 transform 或者 styleSheet 标签来封装, 并且每个块都可以指定自己的 href 指针到一个合适的样式单文件。

使用 jml:transform 的 XSL 例子

本节提供了一个 XSL 样式单的例子以及一个 JSP 页面, 它使用 jml:transform 标签对输出内容应用 XSL 变换。(这只是一个简单的例子——页面中的 XML 是静态的, 更实用的例子往往是在应用 XSL 变换之前, 使用 JSP 页面动态地生成所有或者一部分的 XML 内容。)

样式单例子: hello.xsl

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

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

  <xsl:template match="page">
    <html>
      <head>
        <title>
          <xsl:value-of select="title"/>
        </title>
      </head>
      <body bgcolor="#ffffff">
        <xsl:apply-templates/>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="title">
    <h1 align="center">
      <xsl:apply-templates/>
    </h1>
  </xsl:template>

  <xsl:template match="paragraph">

    <p align="center">
      <i>
        <xsl:apply-templates/>
      </i>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

JSP 页面例子: hello.jsp

```
<%@ page session = "false" %>
<%@ taglib uri="/WEB-INF/jmltaglib.tld" prefix="jml" %>

<jml:transform href="style/hello.xsl" >

<page>
```

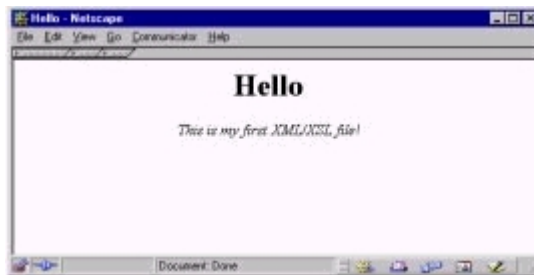
```

<title>Hello</title>
<content>
  <paragraph>This is my first XML/XSL file!</paragraph>
</content>
</page>

</jml:transform>

```

这个例子的输出如下图所示。



5.1.3 Oracle 的数据库访问 JavaBeans

Oracle JSP 提供了一套定制的 JavaBeans 来访问 Oracle 数据库。在 `oracle.jsp.dbutil` 包中包含了下面的 JavaBeans:

- `ConnBean`, 用来打开单个数据库连接。
- `ConnCacheBean`, 使用 Oracle 的连接缓冲机制来进行数据库连接(需要 JDBC 2.0)。
- `DBBean`, 用来执行一个数据库查询。
- `CursorBean`, 提供对查询的通用 DML 制裁, 包括 UPDATE, INSERT, DELETE 语句已经存储过程的调用等。

要获得使用这些 JavaBeans 的例子程序, 请参看“数据库访问 JavaBean 实例”。

所有这四个 JavaBeans 都实现了 OracleJSP 的 `JspScopeListener` 接口以提供事件通知功能。请参看“OracleJSP 事件处理——JspScopeListener”以获得相关的信息。

本节的内容假设你已经对 Oracle JDBC 有了一定程度的了解, 如果你对 Oracle JDBC 一点都不了解的话, 请先去查阅一下“Oracle8i JDBC 开发人员指南与参考”。

注意: 要正确使用 Oracle 的数据库访问 JavaBeans, 你必须首先安装文件 `ojsputil.jar` 并在类路径中包含它, 这个文件由 OracleJSP 所提供。要使用 XML 相关的功能与方法, 你也将需要文件 `xsu12.jar` (对 JDK 1.2.x) 或文件 `xsu11.jar` (对 JDK 1.1.x), 这两个文件由 Oracle8i 8.1.7 发行版提供。

数据库连接——ConnBean

你可以使用 `Oracle.jsp.dbutil.ConnBean` 来建立单个的数据库连接 (即不使用连接池或缓冲的连接。)

注意:

- 如果只是为了查询, 那么直接使用 `DBBean` 是一种最简单的办法, 它有自己

的连接机制。

- 要想实用连接缓冲，请使用 ConnCacheBean 来替代 ConnBean。
- 与你在 JSP 页面中所使用的任何别的 JavaBean 一样，你可以使用 `jsp:setProperty` 语句来设置 ConnBean 的属性；你也可以使用 ConnBean 自己所提供的方法来设置它的属性。

ConnBean 的属性如下：

- user（数据库连接所需要的用户标识）
- password（数据库连接所需要的口令）
- URL（数据库连接字符串）
- stmtCacheSize（Oracle JDBC 语句缓冲的大小）
设置 stmtCacheSize 属性的值就会打开 Oracle JDBC 语句缓冲特性。请参看“JDBC 语句缓冲”以获得关于语句缓冲的优点和局限性的简介。
- executeBatch（Oracle JDBC 成批更新的每批语句数）
设置 executeBatch 属性的值就会打开 Oracle JDBC 成批更新特性。请参看“成批更新”以获得关于成批更新的简介。
- preFetch（在 Oracle JDBC 行预取功能中，设置每次预取的语句数）
设置 preFetch 属性的值将打开 Oracle JDBC 成批更新的特性。请参看“行预取”以获得关于行预取功能的简介。

ConnBean 为上面所列的属性提供了以下的设置和读取的方法：

- void setUser (String)
- String getUser()
- void setPassword(String)
- String getPassword()
- void setURL(String)
- String getURL()
- void setStmtCacheSize(int)
- int getStmtCacheSize()
- void setExecuteBatch(int)
- int getExecuteBatch()
- void setPreFetch(int)
- int getPreFetch()

使用下面的方法可以打开、关闭一个数据库连接：

- void connect()——使用 ConnBean 的属性设置来建立一个数据库连接。
- void close()——关闭数据库连接以及所有打开的游标。

使用下面的方法打开一个游标标签返回一个 CursorBean 对象：

- CursorBean getCursorBean (int, String)

或者

- CursorBean getCursorBean (int)

输入参数的解释如下：

- 一个 int 型的常量，指定你想要使用的 JDBC 语句的类型。常量可以有以下几种取值：CursorBean.PLAIN_STMT（使用 Statement 对象），CursorBean.PREP_STMT（使用 PreparedStatement 对象），cursorBean.CALL_STMT（使用 CallableStatement 对象）。
- 一个 String 型的字符串，指定执行的 SQL 操作。（可选的参数；另外一种可供选择的方案是在 CursorBean 的执行 SQL 语句的方法里来指定 SQL 操作。）

要获得关于 CursorBean 的功能的相关信息，请参看“DML 操作与存储过程调用——CursorBean”。

连接缓冲——ConnCacheBean

使用 oracle.jsp.dbutil.ConnCacheBean 可以利用 Oracle JDBC 的连接缓冲机制（使用 JDBC 2.0 连接池）来建立数据库连接。要获得数据库连接缓冲的一个简要的概述，请参看“数据库连接缓冲”。

注意：

- 如果使用单个的连接对象（没有连接池或缓冲），请使用 ConnBean 来代替 ConnCacheBean。
- ConnCacheBean 是从 OracleConnectionCacheImpl 扩展下来的，而 OracleConnectionCacheImpl 又是从 OracleDataSource 扩展下来的。（这两个类都在 Oracle JDBC 的 oracle.jdbc.pool 包中。）请参看 Oracle8i 8.1.6 或更高版本所带的“Oracle8i JDBC 开发人员指南与参考”以获得关于 Oracle JDBC 类的更多信息。
- 与你在 JSP 页面中所使用的任何别的 JavaBean 一样，你可以使用 jsp:setProperty 语句来设置 ConnCacheBean 的属性；你也可以使用 ConnCacheBean 自己所提供的方法来设置它的属性。
- 与 ConnBean 不同，当你使用 ConnCacheBean 来建立数据库连接时，你使用标准的连接对象的功能来创建并执行语句对象。

下面是 ConnCacheBean 所拥有的属性：

- user（数据库连接所需要的用户标识）
- password（数据库连接所需要的口令）
- URL（数据库连接字符串）
- maxLimit（缓冲所允许的最大连接数）
- minLimit（缓冲所允许的最小连接数；如果你使用的连接数小于此数值，那么在缓冲里将会有空闲的连接池出现。）
- stmtCacheSize（Oracle JDBC 语句缓冲的大小）
设置 stmtCacheSize 属性的值就会打开 Oracle JDBC 语句缓冲特性。请参看“JDBC 语句缓冲”以获得关于语句缓冲的优点和局限性的简介。
- cacheScheme（缓冲的类型）

- DYNAMIC_SCHEME——连接数可以大于 maxLimit 所限制的最大连接数，但是每个连接在逻辑连接的实例不再被使用时会被自动关闭并释放掉。
- FIXED_WAIT_SCHEME——当连接数达到 maxLimit 所规定的最大连接数时，新建的连接要一直等待，直到连接池里某一个连接被释放掉它才可以进来。
- FIXED_RETURN_NULL_SCHEME——当连接数达到 maxLimit 所规定的最大连接数时，新建的连接一直失败（返回 null），直到连接对象被释放。

ConnCacheBean 类支持在 Oracle JDBC 的 OracleConnectionCacheImpl 类里所定义的方法，包括下面的对属性的读取和设置方法：

- void setUser (String)
- String getUser()
- void setPassword(String)
- String getPassword()
- void setURL(String)
- String getURL()
- void setMaxLimit(int)
- int getMaxLimit()
- void setMinLimit(int)
- int getMinLimit()
- void setStmtCacheSize(int)
- int getStmtCacheSize()
- void setCacheScheme(int)
指定 connCacheBean.DYNAMIC_SCHEME,
ConnCacheBean.FIXED_WAIT_SCHEME 或者
ConnCacheBean.FIXED_RETURN_NULL_SCHEME 来作为输入参数。
- int getCacheScheme()
返回 ConnCacheBean.DYNAMIC_SCHEME,
ConnCacheBean.FIXED_WAIT_SCHEME 或者
ConnCacheBean.FIXED_RETURN_NULL_SCHEME。

ConnCacheBean 类还有一些从 oracle.jdbc.pool.OracleDataSource 类继承下来的属性和相关的读取、设置方法，这些属性为 databaseName, dataSourceName, description, networkProtocol, portNumber, serverName 和 driverType。要获得关于这些属性和它们的读取、设置方法的参考信息，请参看“Oracle8i JDBC 开发人员指南与参考”。

使用下面的方法可以打开、关闭一个数据库连接：

- Connection getConnection()——使用 ConnCacheBean 的属性设置从连接缓冲中获得一个连接。
- void close()——关闭所有的连接以及所有打开的游标。

尽管 ConnCacheBean 类不直接支持 Oracle JDBC 的成批更新与行预取特性，但是也可以通过调用 getConnection()方法先得到一个 Connection 对象，然后再调用 Connection 对象

的 `setDefaultExecuteBatch(int)`和 `setDefaultRowPrefetch(int)`方法来允许这些特性。标签还有另外一种可选择的方案，就是先通过 `getConnection()`方法得到 `Connection` 对象，然后从 `Connection` 对象创建一个 `JDBC` 语句对象，再调用语句对象的 `setExecuteBatch(int)`和 `setRowPrefetch(int)`方法，可达到同样的目的（请注意：成批更新只在 `prepared` 语句中被支持。）。

数据库查询——DBBean

使用 `oracle.jsp.dbutil.DBBean` 可以用来进行数据库查询。

注意：

- `DBBean` 有自己的连接机制，不需要使用 `ConnBean`。
- `DBBean` 只能用来数据库查询，要执行别的 DML 操作（如 `UPDATE`、`INSERT`、`DELETE` 或调用存储过程），请使用 `CursorBean`。
- 与你在 JSP 页面中所使用的任何别的 `JavaBean` 一样，可以使用 `jsp:setProperty` 语句来设置 `DBBean` 的属性；你也可以利用 `DBBean` 自己所提供的方法来设置它的属性。

`DBBean` 具有以下属性：

- `user`（数据库连接所需要的用户标识）
- `password`（数据库连接所需要的口令）
- `URL`（数据库连接字符串）

`DBBean` 提供了下面的方法来读取和设置它的属性值：

- `void setUser (String)`
- `String getUser()`
- `void setPassword(String)`
- `String getPassword()`
- `void setURL(String)`
- `String getURL()`

使用下面的方法可以打开、关闭一个数据库连接：

- `void connect()`——使用设置好的 `DBBean` 属性值来建立一个数据库连接。
- `void close()`——关闭连接和所有打开的游标。

使用下面两种方法中的任何一种都可以执行一个查询操作：

- `String getResultAsHTMLTable (String)` ——输入参数是一个字符串，必须是一个 `SELECT` 语句。此方法返回一个字符串，在字符串中包括了必要的 `HTML` 命令以将查询结果记录集输出到一个 `HTML` 表格中，表格的列标题使用 `SQL` 的自段名（或别名）。
- `String getResultAsXMLString(String)` ——输入参数是一个字符串，必须是一个 `SELECT` 语句。此方法将查询结果记录集以 `XML` 字符串的形式返回，`XML` 标签名使用 `SQL` 字段名（或别名）。

DML 操作与存储过程调用——CursorBean

使用 `oracle.jsp.dbutil.CursorBean` 可以用来执行除查询外的其他 DML 操作，如 `SELECT`，`UPDATE`，`INSERT`，`DELETE` 操作或者存储调用等，它要使用 `ConnBean` 来进行数据库连接，因此之前必须先定义一个 `ConnBean` 对象。

你可以在 `ConnBean` 对象的 `getCursorBean` 方法中指定一个 SQL 操作，或者通过 `CursorBean` 对象的 `create()`，`execute()` 或者 `executeQuery()` 方法来得到相同的效果，这些方法将在下面详细讨论。

`CursorBean` 支持可滚动和可更新的游标、成批更新、行预取以及查询超时限制等特性。

注意：

- 如果要使用连接缓冲，请使用 `ConnCacheBean` 和标准的 `Connection` 对象的功能，不要使用 `CursorBean`。
- 与你在 JSP 页面中所使用的任何别的 `JavaBean` 一样，你可以使用 `jsp:setProperty` 语句来设置 `CursorBena` 的属性；你也可以利用 `CursorBean` 自己所提供的方法来设置它的属性。

`CursorBean` 具有一些属性：

- `executeBatch`（Oracle JDBC 成批更新的每批语句数）
设置这个属性的值将打开 Oracle JDBC 的成批更新的特性。
- `preFetch`（在 Oracle JDBC 行预取的功能中，设置每次预取的语句数）
设置这个属性的值将打开 Oracle JDBC 的行预取功能。
- `queryTimeout`（对一个语句执行时间的超时设置，单位为秒，如果一个语句的执行时间超过此设置值，驱动就会认为已经超时。）
- `resultSetType`（结果事件集的滚动能力）：
 - `TYPE_FORWARD_ONLY`（缺省值）——结果时间集只能向前滚动（使用 `next()` 方法）标签不能任意定位。
 - `TYPE_SCROLL_INSENSITIVE`——结果数据集可以向前滚动、向后滚动，标签可以任意定位，但是数据库的改变并不及时做出响应。
 - `TYPE_SCROLL_SENSITIVE`——结果数据集可以向前滚动、向后滚动并且可以任意定位，对数据库的改变可以做出及时响应。
- `resultSetConcurrency`（结果数据集的更新能力）
 - `CONCUR_READ_ONLY`——结果数据集是只读的（不能被修改）。
 - `CONCUR_UPDATABLE`——结果数据集可以被修改。

你可以使用下面所列的方法来设置这些属性的值打开 Oracle JDBC 的特有功能等：

- `void seExecuteBatch(int)`
- `int getExecuteBatch()`
- `void setPreFetch(int)`
- `int getPreFetch()`
- `void setQueryTimeout(int)`

- `int getQueryTimeout ()`
- `void setResultSetConcurrency(int)`
指定 `CursorBean.CONCUR_READ_ONLY` 或 `CONCUR_UPDATABLE` 作为输入参数。
- `int getResultSetConcurrency()`
返回 `CursorBean.CONCUR_READ_ONLY` 或 `CursorBean.CONCUR_UPDATABLE`。
- `void setResultSetType(int)`
指定 `CursorBean.TYPE_FORWARD_ONLY`, `CursorBean.TYPE_SCROLL_INSENSITIVE` 或者 `CursorBean.TYPE_SCROLL_SENSITIVE` 作为输入参数。
- `int setResultSetType()`
返回 `CursorBean.TYPE_FORWARD_ONLY`, `CursorBean.TYPE_SCROLL_INSENSITIVE` 或者 `CursorBean.TYPE_SCROLL_SENSITIVE`。

当使用 `jsp:useBean` 语句定义了 `CursorBean` 的实例以后, 你就可以使用 `CursorBean` 的方法来创建一个游标, 从而可以进行数据库查询。要创建一个游标有两种方法: 你可以使用下面所列的方法先创建游标, 然后提供一个连接, 分两步完成:

- `void create()`
- `void setConnBean(ConnBean)`

或者你也可以将这两步合并到一步中, 通过下面的方法调用来完成:

- `void create(ConnBean)`

(要获得关于如何使用 `ConnBean` 对象的信息, 请参看“数据库连接——`ConnBean`”。) 然后你就可以使用下面所列的方法来指定并执行一个查询:

- `ResultSet executeQuery(String)`

其中输入参数为一个字符串型的值, 指定了一个 `SELECT` 语句来进行查询。

如果你想要把结果数据集以 `HTML` 表格或者 `XML` 字符串的形式输出, 请使用下面两种方法中的一种来代替 `executeQuery()` 方法:

- `String getResultAsHTMLTable(String)`——输入参数是一个字符串, 必须是一个 `SELECT` 语句。此方法返回一个字符串, 在字符串中包括了必要的 `HTML` 命令已经将查询结果记录集输出到一个 `HTML` 表格中, 表格的列标题使用 `SQL` 的字段名 (或别名)。
- `String getResultAsXMLString(String)`——输入参数是一个字符串, 必须是一个 `SELECT` 语句。此方法将查询结果记录集以 `XML` 字符串的形式返回, `XML` 标签名使用 `SQL` 字段名 (或别名)。

在使用 `jsp:useBean` 语句定义了 `CursorBean` 的实例之后, 如果要执行一个 `UPDATE`、`INSERT` 或者 `DELETE` 语句, 可以使用 `CursorBean` 的方法来创建一个游标。创建游标有两种方法: 你可以使用下面所列的方法先创建游标, 然后提供一个连接, 分两步完成:

- `void create(int, String)`——其中, `int` 型的参数指定一个语句类型, `String` 型的参数指定一个 `SQL` 语句。
- `void setConnBean(ConnBean)`

或者你也可以将这两步合并到一步中, 通过下面的方法调用来完成:

- void create(ConnBean, int, String)
- （要获得关于如何使用 ConnBean 对象的信息，请参看“数据库连接——ConnBean”。）

在输入参数中，int 型的参数是下面所列的常量中的一个，它用来指定 JDBC 语句的类型：CursorBean.PLAIN_STMT（对 Statement 对象），CursorBean.PREP_STMT（使用 PreparedStatement 对象），或 cursorBean.CALL_STMT（使用 CallableStatement 对象）。String 型的参数用来指定一个 SQL 语句。

在创建游标后，就可以使用下面的方法来执行 INSERT，UPDATE 或者 DELETE 语句（你可以忽略掉 boolean 型的返回值）：

- boolean execute()

或者使用下面的方法来利用成批更新功能，int 型的返回值指示受影响的行数（请参看后面的内容以获得如果允许成批更新功能的信息）。

- int executeUpdate()

注意：execute()和 executeUpdate()方法都可以接收一个 String 型的输入参数来指定想要执行的 SQL 操作。相应的 create()方法以及 ConnBean 的 getCursorBean()方法可以不需要一个 String 型的执行期，但是不能同时在两个时期都指定 SQL 操作。

除了上面介绍的方法以外，CursorBean 还支持 Oracle JDBC 的语句和结果集功能，诸如 registerOutParameter()方法，setXXX()方法以及 getXXX()方法等。使用下面的方法可以将数据库游标关闭：

- void close()

5.1.4 OracleJSP 的 SQL 标签库

在 OracleJSP 8.1.7 发行版中提供了一套定制的标签库用来实现 SQL 功能，它与 JML 定制标签库是两个独立的库。

在 SQL 标签库中提供了一些标签：

- dbOpen——打开一个数据库连接。
- dbClose——关闭一个数据库连接。
- dbQuery——执行一个数据库查询。
- dbCloseQuery——关闭一个查询的游标。
- dbNextRow——除了结果数据集的行记录。
- dbExecute——执行一个 SQL 语句（DML 或者 DDL）。

这些标签将在本节中分别讲述，要获得 SQL 标签库的例子，请参看“SQL 标签实例”。

注意 SQL 标签库需要一些条件：

- 安装文件 ojsputil.jar 标签在类路径中包含它。此文件由 OracleJSP 安装包所提供。
- 确保标签库描述文件 sqltaglib.tld 与应用程序一起发布，标签 sqltaglib.tld 的存放路径应该与 JSP 页面中用 taglib 指令指定的路径相同，如下例所示：

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
```

要获得关于 JSP 1.1 标签库的用法以及标签库描述文件和 taglib 指令的用法等信息，请

参看“标准标签库框架”。

SQL dbOpen 标签

使用 SQL dbOpen 标签来打开一个数据库连接，下面是它的语法：

```
<sql:dbOpen
  [ connId=" connection-id" ]
  user=" username"
  password=" password"
  URL=" databaseURL" >
...

</sql:dbOpen>
```

在 dbOpen 开始和结束标签之间，可以放上你通过这个连接所希望执行的操作代码。（请参看“SQL 标签实例”。）如果你使用了 connId 参数来设置这个连接的标识符，那么通过这个连接来执行的代码就可以引用连接的标识符，并且不需放在 dbOpen 的开始和结束标签之间。（连接标识符可以是任意的字符串。）

另外，在 JSP 页面中最好不要使用静态的 password（即直接给 password 赋一个值），这很容易导致安全问题。作为替代的方法，你可以从请求对象中来得到 password 参数以及其他一些参数，如下例所示：

```
<sql:dbOpen connId="conn1" user=<%=request.getParameter("user")%>
  password=<%=request.getParameter("password")%> URL="url" />
```

（在这个例子中，你不需要标签的正文来编写使用这个连接的代码；如果你想使用这个连接来进行一些操作，那么在代码中你可以通过 connId 的值 conn1 来引用这个连接）

如果你设置了连接标识符，那么在你显式地使用 dbClose 标签来关闭连接之前，此连接一直有效。如果没有设置连接标识符，那么当碰到</sql:dbOpen>结束标签时，连接就会被自动关闭。

dbOpen 标签使用一个 ConnBean 对象来进行数据库连接，因此你可以设置 ConnBean 的 stmtCacheSize、preFetch 以及 batchSize 等属性来允许这些 Oracle JDBC 的特性。要获得更多的相关信息，请参看“数据库连接——ConnBean”。

SQL dbClose 标签

使用 dbClose 标签来关闭使用 dbOpen 标签打开的数据库连接，它使用 ConnId 来判断应该关闭哪一个连接。如果在 dbOpen 标签中没有使用 ConnId 参数，那么在执行到 dbOpen 结束标签时此连接就会自动关闭，无需使用 dbClose 标签。dbClose 标签的语法如下：

```
<sql:dbClose connId=" connection-id" />
```

注意：在 OracleJSP 环境下，你可以通过 Oracle 的 JspScopeListener 机制使用事件处理以使连接在会话结束的时候被自动关闭。要获得相关的信息，请参看“OracleJSP 事件处理——JspScopeListener”。

SQL dbQuery 标签

使用 dbQuery 标签来执行一个数据库查询，结果的输出形式可以是一个 JDBC 记录集

合、一个 HTML 表格或者一个 XML 字符串。在此标签的正文区中，也就是在<sql:dbQuery>开始标签和</sql:dbQuery>结束标签之间可以插入一条 SELECT 语句（只允许一条）来执行查询。它的语法如下：

```
<sql:dbQuery
  [ queryId=" query-id" ]
  [ connId=" connection-id" ]
  [ output="HTML|XML|JDBC" ] >
  ... SELECT statement (one only) ...
</sql:dbQuery>
```

注意：在 OracleJSP 8.1.7 (1.1.0.0.0) 发行版中，SELECT 语句并不以分号 (;) 终止，这将导致一个语法错误。

dbQuery 标签的所有参数都是任选的，根据用法不同，你可以设置不同的参数，这将在下面分别讲述。

如果希望使用 dbNextRow 标签来处理结果记录集，那么你必须使用 queryId 参数来设置一个查询标识符。queryId 可以被设置成任意的字符串。

另外，如果设置了 queryId 参数，那么数据库游标不会被自动关闭，你只有显式地使用 dbCloseQuery 标签来关闭它。如果没有设置 queryId 参数，那么在遇到<sql:dbQuery>结束标签时，数据库游标被自动关闭。

如果 connId 参数没有被指定，那么 dbQuery 标签必须出现在 dbOpen 标签的正文区，标签使用由 dbOpen 标签所打开的那个数据库连接。

对输出类型来讲：

- HTML——将结果记录集以 HTML 表格的形式输出（缺省值）。
- XML——将结果记录集以 XML 字符串的形式输出。
- JDBC——将结果记录集放到一个 JDBC ResultSet 对象中，在后面可以使用 dbNextRow 标签来循环整个记录集并做一些处理工作。

dbQuery 标签使用一个 CursorBean 对象来处理游标。

SQL dbCloseQuery 标签

使用 dbCloseQuery 标签来关闭用 dbQuery 标签所打开的数据库游标，它用 queryId 参数来判断应该关闭哪一个游标。如果在 dbQuery 标签中没有使用 queryId 参数，那么在执行到 dbQuery 结束标签时此游标会被自动关闭，无需使用 dbCloseQuery 标签。dbCloseQuery 标签的语法如下：

```
<sql:dbCloseQuery queryId=" query-id" />
```

注意：在 OracleJSP 环境下，你可以通过 Oracle 的 JspScopeListener 机制，使用事件处理以使数据库游标在会话结束的时候被自动关闭。要获得相关的信息，请参看“OracleJSP 事件处理——JspScopeListener”。

SQL dbNextRow 标签

使用 dbNextRow 标签用来对通过 dbQuery 标签查询得到的结果记录集的每一行进行处

理，并且它与一个指定的 `queryId` 参数相关联。处理代码应该放在 `dbNextRow` 标签的正文区，也就是`<sql:dbNextRow>`开始标签和`</sql:dbNextRow>`结束标签之间，正文区的内容对记录集里的每一行都要被调用。

如果你使用 `dbNextRow` 标签，那么在 `dbQuery` 标签中必须指定 `output=JDBC`，并且必须指定 `queryId` 参数以便 `dbNextRow` 能引用。

`dbNextRow` 标签的语法如下所示：

```
<sql:dbNextRow queryId=" query-id" >
```

```
... Row processing...
```

```
</sql:dbNextRow >
```

结果记录集对象在 `dbQuery` 标签的 `tag-extra-info` 类的实例中被创建。

SQL `dbExecute` 标签

使用 `dbExecute` 标签来执行任意一条 DML 或 DDL 语句（只允许一条）。将要执行的语句放在 `dbExecute` 标签的正文区，也就是`<sql: dbExecute 标签>`开始标签和`</sql: dbExecute 标签>`结束标签之间。`dbExecute` 标签的语法如下：

```
<sql:dbExecute
  [connId=" connection-id"]
  [output="yes|no"] >
  ... DML or DDL statement (one only)...
</sql:dbExecute >
```

注意：在 OracleJSP 8.1.7 发行版中，DML 或 DDL 语句并不以分号（;）终止，这将导致一个语法错误。

如果你没有指定 `connId` 参数，那么 `dbExecute` 标签必须出现在 `dbOpen` 标签的正文区，并且使用由 `dbOpen` 标签所打开的那个数据库连接。

如果设置了 `output=yes`，对 DML 语句来，HTML 字符串 “number row[s] affected” 将被输出到浏览器以通知用户此次操作影响到了多少行数据；对 DDL 语句来说，语句的执行状态也会被输出到屏幕上。`output` 参数的缺省设置是 `no`。

`dbExecute` 标签使用一个 `CursorBean` 对象来处理游标。

SQL 标签实例

下面的例子演示了如何使用 OracleJSP 的 SQL 标签。（如果你想在自己的环境下运行它们，必须适当地设置 URL，user name 以及 password 等属性。）

例一：使用连接标识符来查询

```
<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
  <HEAD>
    <TITLE>A simple example with open, query, and close tags</TITLE>
  </HEAD>
  <BODY BGCOLOR="#FFFFFF">
    <HR>
```

```

        <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
            user="scott" password="tiger" connId="con1">
        </sql:dbOpen>
        <sql:dbQuery connId="con1">
            select * from EMP
        </sql:dbQuery>
        <sql:dbClose connId="con1" />
        <HR>
    </BODY>
</HTML>

```

例二：在 dbOpen 标签中使用查询代码

```

<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
    <HEAD>
        <TITLE>Nested Tag with Query inside Open </TITLE>
    </HEAD>
    <BODY BGCOLOR="#FFFFFF">
        <HR>
        <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
            user="scott" password="tiger">
            <sql:dbQuery>
                select * from EMP
            </sql:dbQuery>
        </sql:dbOpen>
        <HR>
    </BODY>
</HTML>

```

例三：带有 XML 输出的查询

```

<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
    <HEAD>
        <TITLE>A simple tagLib with XML output</TITLE>
    </HEAD>
    <BODY BGCOLOR="#FFFFFF">
        <HR>
        <sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
            user="scott" password="tiger">
            <sql:dbQuery output="xml">
                select * from EMP
            </sql:dbQuery>
        </sql:dbOpen>
        <HR>
    </BODY>
</HTML>

```

例四：循环处理结果记录集

```

<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
    <HEAD>
        <TITLE>Result Set Iteration Sample </TITLE>

```



```

</HEAD>
<BODY BGCOLOR="#FFFFFF">
    <HR>
    <sql:dbOpen connId="con1"
URL="jdbc:oracle:thin:@dlsun991:1521:816"
        user="scott" password="tiger">
    </sql:dbOpen>
    <sql:dbQuery connId="con1" output="jdbc" queryId="myquery">
        select * from EMP
    </sql:dbQuery>
    <sql:dbNextRow queryId="myquery">
        <%= myquery.getString(1) %>
    </sql:dbNextRow>
    <sql:dbCloseQuery queryId="myquery" />
    <sql:dbClose connId="con1" />
    <HR>
</BODY>
</HTML>

```

例五：DDL 与 DML 语句

本例中使用了一个 HTML 表单，允许用户指定哪种类型的 DML 或 DDL 语句被执行。

```

<%@ taglib uri="/WEB-INF/sqltaglib.tld" prefix="sql" %>
<HTML>
<HEAD><TITLE>DML Sample</TITLE></HEAD>
<FORM METHOD=get>
<INPUT TYPE="submit" name="drop" VALUE="drop table test_table"><br>
<INPUT TYPE="submit" name="create"
        VALUE="create table test_table (col1 NUMBER)"><br>
<INPUT TYPE="submit" name="insert"
        VALUE="insert into test_table values (1234)"><br>
<INPUT TYPE="submit" name="select" VALUE="select * from test_table"><br>
</FORM>
<BODY BGCOLOR="#FFFFFF">
Result:

```

```

<HR>
<sql:dbOpen URL="jdbc:oracle:thin:@dlsun991:1521:816"
user="scott" password="tiger">
    <% if (request.getParameter("drop")!=null) { %>
    <sql:dbExecute output="yes">
        drop table test_table
    </sql:dbExecute>
    <% } %>
    <% if (request.getParameter("create")!=null) { %>
    <sql:dbExecute output="yes">
        create table test_table (col1 NUMBER)
    </sql:dbExecute>
    <% } %>
    <% if (request.getParameter("insert")!=null) { %>
    <sql:dbExecute output="yes">
        insert into test_table values (1234)
    </sql:dbExecute>

```

```

        <% } %>
        <% if (request.getParameter("select")!=null) { %>
        <sql:dbQuery>
            select * from test_table
        </sql:dbQuery>
        <% } %>
    </sql:dbOpen>
    <HR>
</BODY>
</HTML>

```

5.2 Oracle 专用的编程扩展

本节讨论的 OracleJSP 扩展功能不能移植到别的 JSP 环境中，包含以下内容：

- 通过 Oracle JspScopeListener 机制来实现的事件处理
- 对 SQLJ（一个标准的语法，允许直接将 SQL 语句嵌入到 Java 代码中）的支持
- JDBC 的性能增强特性

注意：

- 对 Servlet 2.0 环境来说，OracleJSP 提供了一个不可移植的扩展功能，允许通过 globals.jsa 机制来支持网络应用程序框架。
- OracleJSP 也提供了扩展的 NLS 支持（不可移植）。

5.2.1 OracleJSP 事件处理——JspScopeListener

在标准的 Servlet 和 JSP 技术中，仅仅支持基于会话的事件。OracleJSP 通过 oracle.jsp.event 包中的 JspScopeListener 接口和 JspScopeEvent 类扩展了这一支持。OracleJSP 提供的机制支持下面四种标准的 JSP 作用域的事件处理，并且适用于 JSP 应用程序中的所有 Java 对象：

- 页面（page）作用域
- 请求（request）作用域
- 会话（session）作用域
- 应用程序（application）作用域

对于应用程序中所用到的 Java 对象，首先在适当的类中实现 JspScopeListener 接口，然后将这个类的对象分配一个 JSP 作用域，如使用 jsp:useBean 语句来设定作用域。

当作用域到达终止范围时，实现了 JspScopeListener 接口并且分配了此作用域的对象将会收到一个事件消息。OracleJSP 容器通过在 JspScopeListener 接口中指定的 outOfScope() 方法向这些满足条件的对象发送一个 JspScopeEvent 的实例，从而实现事件的通告。

JspScopeEvent 对象包括下面一些属性：

- 正在终止的作用域（是下面几个常量的一个：PAGE_SCOPE, REQUEST_SCOPE, SESSION_SCOPE 或者 APPLICATION_SCOPE）。
- 容器对象，即包括了这个作用域里所有对象的一个容器（是下面几个隐式对象中的一个：page, request, session 或者 applicaton）。

- 被通告对象的名字（即实现了 JspScopeListener 接口的类的实例的名字）。
- JSP 隐式对象 application。

OracleJSP 事件监听——通告机制极大地方便了开发人员，可以使他们无需担心资源的释放问题，也不必在他们的页面里使用大量的 Java try/catch/finally 代码块来处理错误。相反地，他们只需在事件处理函数里面将一些必要的资源释放掉即可，并且这些适用于所有的四种作用域。

5.2.2 OracleJSP 对 Oracle SQLJ 的支持

SQLJ 是一套标准的语法，用来将静态的 SQL 语句直接嵌入 Java 代码中，极大地简化了编程工作量。OracleJSP 以及 OracleJSP 解释器支持 Oracle SQLJ，允许你在 JSP 脚本段中直接使用 SQLJ 语法。SQLJ 语句又用 #sql 来标识。

SQLJ 与 JSP 混合编程的例子

下面是一个使用了 SQLJ 的样例 JSP 页面。（此页面中使用 pageimport 指令所包含的类，都是一些典型的 SQLJ 所需要的类。）

页面的代码如下：‘

```
<%@ page language="sqlj"
import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle"
%>

<HTML>
<HEAD> <TITLE> The SQLJQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">
<% String empno = request.getParameter("empno");
if (empno != null) { %>
<H3> Employee # <%=empno %> Details: </H3>
<%= runQuery(empno) %>
<HR><BR>
<% } %>
<B>Enter an employee number:</B>
<FORM METHOD="get">
<INPUT TYPE="text" NAME="empno" SIZE=10>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%!

private String runQuery(String empno) throws java.sql.SQLException {
    DefaultContext dctx = null;
    String ename = null; double sal = 0.0; String hireDate = null;
    StringBuffer sb = new StringBuffer();
    try {
        dctx = Oracle.getConnection("jdbc:oracle:oci8:@", "scott",
"tiger");
        #sql [dctx] {
            select ename, sal, TO_CHAR(hiredate,'DD-MON-YYYY')
            INTO :ename, :sal, :hireDate
            FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
        };
        sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
        sb.append("Name : " + ename + "\n");
        sb.append("Salary : " + sal + "\n");
    }
}
```

```

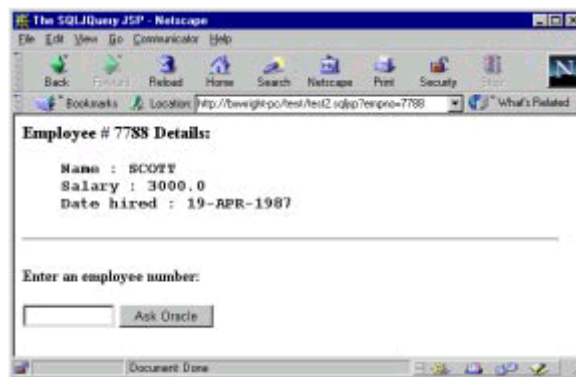
sb.append("Date hired : " + hireDate);
sb.append("</PRE></B></BIG></BLOCKQUOTE>");
} catch (java.sql.SQLException e) {
    sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
} finally {
    if (dctx!= null) dctx.close();
}
return sb.toString();
}

```

%>

这个例子使用了 JDBC OCI 驱动，因此需要一个 Oracle 客户端安装。在例子中所使用的用来得到数据库连接的 Oracle 类，由 Oracle SQLJ 提供。

如果在雇员编号输入框中输入 7788，那么本例的输出页面如下图所示。



注意：

- 在一个 Java 虚拟机中多次调用同一个 JSP 页面时，建议你总是使用一个显式的连接环境来代替缺省的连接环境，如在本例中使用的 dctx。（请注意，dctx 是一个局部的方法变量。）
- OracleJSP 需要 Oracle SQLJ 8.1.6.1 或更高的发行版本。
- 在将来的版本中，OracleJSP 将在 page 指令中支持 language = “sqlj” 这些的设置以便在 JSP 解释过程中能触发 SQLJ 解释器。为了向前兼容，建议你总是在 page 指令中使用该条设置，并且把这作为一个良好的习惯。

要获得在 JSP 页面中使用 SQLJ 的更复杂一些的例子，请参看“SQLJ 查询——SQLJSelectInto.sqljsp 和 SQLJIterator.sqljsp”。

触发 SQLJ 解释器

通过将 JSP 源文件的扩展名改为.sqljsp，也可以触发 OracleJSP 的解释器调用 Oracle SQLJ 解释器来处理源文件。

在这种情况下，OracleJSP 解释器将生成一个.sqlj 文件，而不是通常的.java 文件，通过调用 Oracle SQLJ 解释器将.sqlj 文件解释成.java 文件。

使用 SQLJ 导致了额外的输出文件。

注意:

- 要使用 Oracle SQLJ, 你必须正确安装合适的 SQLJ ZIP 文件 (取决于你自己的环境) 并且将它们添加进你自己的类路径中去。
- 在同一个应用程序里, .jsp 文件和 .sqljsp 文件不要使用相同的文件名, 尽管它们的扩展名不同, 但是它们的生成类文件 (.java 文件) 将出现同名问题。

设置 Oracle SQLJ 选项

当执行或者预解释一个 SQLJ JSP 页面时, 你可以设置自己希望的 SQLJ 选项参数, 并且这对按需解释场合和预解释场合都适用, 如下所示:

- 对按需解释场合来说, 请使用 OracleJSP 的 sqljcmd 配置参数。这个参数允许你设置 SQLJ 的命令行选项, 并且允许你指定除了标准的 SQLJ 解释器以外的 SQLJ 解释器。(sqljcmd 配置参数在 OracleJSP 1.1.0.0.0 发行版之前是不可用的。)
- 对使用 ojspc 预解释工具的预解释场合来说, 请使用 ojspc-S 选项。这一选项允许你设置 SQLJ 的命令行选项。

5.3 OracleJSP 对 Servlet 2.0 应用程序和会话的支持

OracleJSP 定义了一个叫做 globals.jsa 的文件, 通过它在 Servlet 2.0 环境下实现了完整的 JSP 标准所规定的功能。因为 Servlet 2.0 标准并没有完整定义网络应用程序和 Servlet 环境, 所以 OracleJSP 才使用 globals.jsa 这一机制来弥补这一缺陷。

本节讨论了 globals.jsa 机制, 包括以下几部分内容:

- globals.jsa 的功能概述
- globals.jsa 的语法和语义概述
- globals.jsa 的事件处理器
- 全局的声明和指令语句

注意: 对 globals.jsa 文件名的拼写来说, 最好全部使用小写字母。混合大小写的写法可以在大小写不敏感的环境下通过, 但是在大小写敏感的环境下就会出问题。

5.3.1 globals.jsa 功能概述

在任何单个的 Java 虚拟机中, 你可以对每个应用程序使用一个 globals.jsa 文件 (在这里, 应用程序与 Servlet 环境是等价的)。globals.jsa 文件适用于下面所列的情况:

- 应用程序的发布——在这种情况下, globals.jsa 文件作为一个应用程序位置标记器, 用来定义一个应用程序的根目录。
- 不同的应用程序和会话——在这种情况下, OracleJSP 通过使用 globals.jsa 文件来为每个应用程序提供不同的 Servlet 环境和会话环境。
- 应用程序生命周期管理——在这种情况下, 通过会话和应用程序的 Star 和 end 事件来管理应用程序的生命周期。

globals.jsa 文件也为全局的 Java 声明和 JSP 指令语句提供了存放的空间, 它们可以在

应用程序里所有的 JSP 页面中访问。

通过 globals.jsa 发布应用程序

要发布一个不包含 Servlets 的 OracleJSP 应用程序，首先将应用程序的目录结构复制到 Web Server 上，然后在应用程序的根目录下创建一个叫做 globals.jsa 的文件。

globals.jsa 文件的长度可以是零字节。OracleJSP 容器将会寻找 globals.jsa 文件，并将它所在的目录作为应用程序的根目录。

OracleJSP 也定义了一些缺省的路径用来查找 JSP 应用程序资源。例如，WEB-INF/classes 和 WEB-INF/lib 这两个目录下的 Java 类和 JavaBeans 会由 OracleJSP 的类加载器自动加载，并不需要特别的配置。

注意：对于一个不包含 Servlets 的应用程序，尤其是在早于 Servlet 2.2 标准的 Servlet 环境下，要正确地发布 Servlets 就必须通过手工配置来完成。对于 Servlet 2.2 环境下的 Servlet 来说，你可以在标准的 web.xml 发布描述文件中包含必要的配置信息。

通过 globals.jsa 得到不同的应用程序和会话

Servlet 2.0 标准对网络应用程序的概念并没有一个清晰的定义，并且也没有定义 Servlet 环境和应用程序之间的关系，而在以后版本的 Servlet 标准中则对这些概念作了定义。因此在诸如 Apache/JServ 这样的 Servlet 2.0 环境下，每个 Java 虚拟机只有一个 Servlet 环境对象，并且一个 Servlet 2.0 环境只有一个会话对象。

不过 globals.jsa 文件提供了对一个 Web Server 下的多个应用程序和多个会话的支持，这对 Servlet 2.0 环境尤其意义重大。

在 Servlet 2.0 环境下，每个应用程序不允许不同的 Servlet 环境，不但每个应用程序可以拥有一个 globals.jsa 文件，这一机制允许 OracleJSP 容器为每个应用程序提供不同的 ServletContext 对象。并且对那些只允许有一个会话对象的环境，globals.jsa 文件的使用允许 OracleJSP 容器对应用程序提供一个代理 HttpSession 对象。（这就阻止了拥有程序间的会话变量名字冲突的问题，但不幸的是，它还不能保护应用程序的数据不被其他应用程序所查看或修改。这是因为 HttpSession 对象必须依赖于 Servlet 环境的一些功能。）

通过 globals.jsa 管理应用程序和会话生命周期

当一些重要的状态改变发生时，应用程序应该得到 tonggao 信息。例如，应用程序常常希望能在一个 HTTP 会话开始时得到一些资源并在会话结束时释放这些资源，或者在应用程序本身启动时恢复一些永久性数据，在应用程序关闭时保存这些数据。

然而，在标准的 JSP 和 Servlet 技术中，仅仅基于会话的事件得到支持。

对于使用 globals.jsa 文件的应用程序，OracleJSP 扩展了对事件处理的支持，它可以处理以下四种事件：

- session_OnStart
- session_OnEnd

- application_OnStart
- application_OnEnd

你可以在 globals.jsa 文件中针对以上的 4 个事件分别编写事件处理代码。

session_OnStart 事件在一个 HTTP 会话开始时被触发, 而 session_OnEnd 事件在会话终止时被触发。

application_OnStart 事件在当一个运行在 Java 虚拟机里的应用程序第一次被请求运行时触发, 而 application_OnEnd 事件在 OracleJSP 容器卸载一个应用程序时触发。

5.3.2 globals.jsa 的语法和语义概述

本节对 globals.jsa 文件的语法和语义作了一般性的概述。

在 globals.jsa 文件里的每一个事件处理代码块——session_OnStart 块, session_OnEnd 块, application_OnStart 块或者 application_OnEnd 块——都有一个开始标签, 一个事件结束标签, 还有一个正文区 (在开始标签和结束标签之间的部分) 用来包含事件处理的代码。

下面的例子表明了这一基本语法:

```
<event:session_OnStart>
  <% This scriptlet contains the implementation of the event handler %>
</event:session_OnStart>
```

事件处理代码块的正文部分可以包含任何有效的 JSP 标签——标准的标签以及在定制标签库中定义的标签。

在事件处理代码块中的 JSP 标签的作用域只限于这个代码块, 例如, 如果在一个代码块中用 jsp:useBean 标签声明了一个 bean, 那么在另外一个代码块中如果想使用它的话就必需重新声明。不过你可以通过 globals.jsa 文件的全局声明机制来避免这一限制。

要得到关于这 4 个事件的事件处理器的详细内容, 请参看 “globals.jsa 事件处理器”。

注意: 在普通的 JSP 页面中可以使用的静态文本只能出现在 session_OnStart 的事件处理代码块中。对 session_OnEnd, application_OnStart 以及 application_OnEnd 的事件处理代码块来说, 它们的正文部分只运行出现 Java 脚本代码。

在 globals.jsa 文件中, 事件处理代码块所允许访问的 JSP 隐式对象如下所示:

- 对 application_OnStart 事件处理代码块来说, 可以访问 application 对象。
- 对 application_OnEnd 事件处理代码块来说, 可以访问 application 对象。
- 对 session_OnStart 事件处理代码块来说, 可以访问 application、session、request、response、page 以及 out 对象。
- 对 session_OnEnd 事件处理代码块来说, 可以访问 application 和 session 对象。

一个完整的 globals.jsa 文件的例子: 下面这个例子演示了一个完整的 globals.jsa 文件, 并且使用了所有的 4 个事件处理器。

例子的程序代码如下所示:

```
<event:application_OnStart>
  <%-- Initializes counts to zero --%>
  <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application"
/>
  <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application"
```

```

/>
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application"
/>

</event:application_OnStart>

<event:application_OnEnd>

    <%-- Acquire beans --%>
    <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application"
/>
    <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
    <% application.log("The number of client sessions were: " + sessionCount.getValue() );
%>

</event:application_OnEnd>

<event:session_OnStart>

    <%-- Acquire beans --%>
    <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application"
/>
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application"
/>
    <%
        sessionCount.setValue(sessionCount.getValue() + 1);
        activeSessions.setValue(activeSessions.getValue() + 1);
    %>
    <br>
    Starting session #: <%=sessionCount.getValue() %> <br>
    There are currently <b> <%= activeSessions.getValue() %> </b> active sessions <p>

</event:session_OnStart>

<event:session_OnEnd>

    <%-- Acquire beans --%>
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application"
/>
    <%
        activeSessions.setValue(activeSessions.getValue() - 1);
    %>

</event:session_OnEnd>

```

5.3.3 globals.jsa 事件处理器

本节将对 globals.jsa 的 4 个事件处理器作详细的介绍:

application_OnStart

application_OnStart 事件处理器的通用语法如下所示:

```
<event:application_OnStart>
    <% This scriptlet contains the implementation of the event handler %>
</event:application_OnStart>
```

application_OnStart 事件处理器的正文区在 OracleJSP 容器加载应用程序中的第一个 JSP 页面时被执行, 这通常发生在第一个客户端 HTTP 请求到达应用程序的页面时。应用程序一般使用这个事件来初始化一些应用程序所需要的全局资源, 比如一个数据库连接池或者从一个永久性存储介质中读取数据到应用程序的对象当中。

在此事件处理器的正文区当中只能包含了 JSP 标签 (包括定制标签) 和空白字符——要记住, 它不能含有静态文本内容。

如果此事件处理器中有错误发生, 并且事件处理器的代码也没有处理这个错误, 那么 OracleJSP 容器就会自动跟踪这个错误并且使用应用程序的 Servlet 环境来记录错误信息。事件处理代码就好象没有任何错误发生一样继续往下执行。

例子: application_OnStart 下面的 application_OnStart 例子是从 “Application 事件 global.jsa 实例——lotto.jsp” 抽出来的, 在这个例子中, 对一个特定用户所生成的六合彩号码在缓存里要保留一整天, 如果这个用户重新选择的话, 他得到的将是相同的号码。

缓存每天被清理一次, 因此在新的一天每个用户就可以有一套新的选择。为了实现这些功能, lotto 这个应用程序必须在它被关闭之前将缓存里的内容保存在一个永久性介质 (如硬盘) 上, 然后在它被重新激活时必须刷新缓存。

这个例子中, application_OnStart 事件处理器是从 lotto.che 文件中读取缓存内容的, 程序代码如下所示:

```
<event:application_OnStart>

<%

    Calendar today = Calendar.getInstance();
    application.setAttribute("today", today);
    try {
        FileInputStream fis = new FileInputStream

(application.getRealPath("/") + File.separator + "lotto.che");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Calendar cacheDay = (Calendar) ois.readObject();
        if (cacheDay.get(Calendar.DAY_OF_YEAR) == today.get(Calendar.DAY_OF_YEAR)) {
            cachedNumbers = (Hashtable) ois.readObject();
            application.setAttribute("cachedNumbers", cachedNumbers);
        }
        ois.close();
    } catch (Exception theE) {
        // catch all -- can't use persistent data
    }

%>
```

```
</event:application_OnStart>
```

application_OnEnd

application_OnEnd 事件处理器的通用语法如下所示:

```
<event:application_OnEnd>
    <% This scriptlet contains the implementation of the event handler %>
</event:application_OnEnd>
```

application_OnEnd 事件处理器的正文区在 OracleJSP 容器卸载 JSP 应用程序时被执行,这通常发生在两种情况下:在按需解释模型下,一个已经被加载的页面又被动态地重新解释后,这时 JSP 应用程序要被卸载(除非 OracleJSP 的配置参数中 unsafe_reload 被允许);或者当 OracleJSP 容器本身就是一个 Servlet,它被 Servlet 容器用来调用 destroy()方法终止时,此时 JSP 应用程序也被卸载。应用程序一般使用 application_OnEnd 事件来释放应用程序级的资源或者将应用程序的状态写到一个永久性介质中。

在此事件处理器的正文区当中只能包含了 JSP 标签(包括定制标签)和空白字符——要记住,它不能含有静态文本内容。

如果此事件处理器中有错误发生,并且事件处理器的代码也没有处理这个错误,那么 OracleJSP 容器就会自动跟踪这个错误并且使用应用程序的 Servlet 环境来记录错误信息。事件处理代码就好象没有任何错误发生一样继续往下执行。

例子: application_OnEnd 下面的 application_OnEnd 例子是从“Application 事件 globals.jsa 实例——lotto.jsp”中抽出来的。在这个事件处理器中,缓存的内容在应用程序结束之前被写到 lotto.che 文件中。例子的程序代码如下:

```
<event:application_OnEnd>

<%
    Calendar now = Calendar.getInstance();
    Calendar today = (Calendar) application.getAttribute("today");
    if (cachedNumbers.isEmpty() ||
        now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
        File f = new File(application.getRealPath("/") + File.separator + "lotto.che");
        if (f.exists()) f.delete();
        return;
    }
    try {
        FileOutputStream fos = new FileOutputStream
            (application.getRealPath("/") + File.separator + "lotto.che");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(today);
        oos.writeObject(cachedNumbers);
        oos.close();
    } catch (Exception theE) {
        // catch all -- can't use persistent data
    }
%>

</event:application_OnEnd>
```

session_OnStart

session_OnStart 事件处理器的通用语法如下所示:

```
<event:session_OnStart>
    <% This scriptlet contains the implementation of the event handler %>
    Optional static text...
</event:session_OnStart>
```

session_OnStart 事件处理器的正文区在 OracleJSP 容器创建一个新的会话时被执行。一个会话往往对应着一个 JSP 页面请求, 这个请求是跟客户端相关的, 它发生在这个特定客户端的第一个请求被应用程序的 JSP 页面所收到时。

应用程序可以使用 session_OnStart 事件来达到以下目的:

- 初始化与某一个特定客户端相关联的资源
- 控制应用程序客户端的起始页面

因为在 session_OnStart 事件中可以利用隐式的 out 对象, 所以它成了 globals.jsa 的 4 个事件处理器中唯一一个能够同时包含 JSP 标签和静态文本内容的事件处理器。

session_OnStart 事件处理器在 JSP 页面代码执行之前被调用, 因此从 session_OnStart 中得到的输出也在任何页面输出之前。

session_OnStart 事件处理器和触发此事件的那个 JSP 页面共享相同的输出流, 输出流的缓冲大小是由 JSP 页面的缓冲大小所决定的。session_OnStart 事件处理器不会自动将输出流刷新到浏览器——输出流是按照一般的 JSP 规则来刷新的。头信息仍然可以被写入到 JSP 页面中以触发 session_OnStart 事件。

如果此事件处理器中有错误发生, 并且事件处理器的代码也没有处理这个错误, 那么 OracleJSP 容器就会自动跟踪这个错误并且使用应用程序的 Servlet 环境来记录错误信息。事件处理代码就好象没有任何错误发生一样继续往下执行。

例子: session_OnStart: 下面的例子可以确保每个新的会话都开始于应用程序的初始页面 (index.jsp)。例子的程序代码如下:

```
<event:session_OnStart>

    <% if (!page.equals("index.jsp")) { %>
        <jsp:forward page="index.jsp" />
    <% } %>

</event:session_OnStart>
```

session_OnEnd

session_OnEnd 事件处理器的通用语法如下所示:

```
<event:session_OnEnd>
    <% This scriptlet contains the implementation of the event handler %>
</event:session_OnEnd>
```

session_OnEnd 事件处理器的正文区在 OracleJSP 容器使一个会话无效时被执行, 这通常发生在下面两种情况下:

- 应用程序通过调用 session.invalidate()方法来使会话无效
- 会话过期 (超时)

应用程序可以使用 `session_OnEnd` 事件来释放客户端的资源。

在此事件处理器的正文区当中只能包含了 JSP 标签（包括定制标签）和空白字符——要记住，它不能含有静态文本内容。

如果此事件处理器中有错误发生，并且事件处理器的代码也没有处理这个错误，那么 OracleJSP 容器就会自动跟踪这个错误并且使用应用程序的 Servlet 环境来记录错误信息。事件处理代码就好象没有任何错误发生一样继续往下执行。

例子：session_OnEnd 下面的例子演示了如何在一个会话结束时将激活的会话数目减少一个，这在统计访问数量时相当有用。例子的程序代码如下：

```
<event:session_OnEnd>

  <%-- Acquire beans --%>
  <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope="application"
  />

  <%
    activeSessions.setValue(activeSessions.getValue() - 1);
  %>

</event:session_OnEnd>
```

5.3.4 全局的声明和指令语句

除了可以保存事件处理器代码，`globals.jsa` 文件也可以用来为 JSP 应用程序声明全局指令语句以及对象变量等。你可以在 `globals.jsa` 文件中包含 JSP 指令、JSP 声明、JSP 注释以及含有作用域参数的 JSP 标签（如 `jsp:useBean`）等。

本节涵盖了以下几部分内容：

- 全局 JSP 指令
- `globals.jsa` 的声明
- 全局的 JavaBeans
- `globals.jsa` 的结构
- 全局的声明和指令语句例子

全局 JSP 指令

在一个 `globals.jsa` 文件中使用的指令语句都有如下双重作用：

- 它们所声明的信息可以被 `globals.jsa` 文件本身所利用
- 它们所声明的信息可以被后面的 JSP 页面所利用

在 `globals.jsa` 文件中的指令语句变成了应用程序中所有 JSP 页面的隐式指令语句，但是在任何一个特定的 JSP 页面中都可以覆盖掉这些指令语句。

如果 JSP 页面覆盖了 `globals.jsa` 的指令语句，那么这种覆盖的单位是属性，也就是说只覆盖指令语句中相同的属性。例如，假设一个 `globals.jsa` 文件中有下面一条指令：

```
<%@ page import="java.util.*" bufferSize="10kb" %>
```

并且一个 JSP 页面中有下面的指令：

```
<%@page bufferSize="20kb" %>
```

那么最终的结果 JSP 页面相当于含有下面的指令：

```
<%@ page import="java.util.*" bufferSize="20kb" %>
```

globals.jsa 的声明

如果你想要 globals.jsa 的所有事件处理器都能共享一个方法或者数据成员，那么你可以在 global.jsa 文件里使用<%!...%>来声明这些方法和数据成员。

请注意应用程序中的 JSP 页面是无法访问这些声明的，因此你不能利用这个机制去实现一个应用程序库。对声明的支持是仅仅由 globals.jsa 提供用来在所有的事件处理器中共享通用的函数。

全局的 JavaBeans

在 globals.jsa 文件中最通用的元素声明就是全局的对象声明，在 globals.jsa 文件中声明的对象自动变成 globals.jsa 事件处理器以及应用程序中所有 JSP 页面的隐式对象，因此可以跨文件使用这些全局对象。

在 globals.jsa 文件中声明的对象（如使用 jsp:useBean 语句声明）不需要在应用程序中任何单个 JSP 页面中重新声明。

你可以使用任何 JSP 标签或者扩展来声明一个全局的对象，标签还可以带有作用域参数（如使用 jsp:useBean 或 jml:useVariable）。全局声明的对象应该具有会话或者应用程序作用域之一（不能具有页面或者请求作用域）。

在 globals.jsa 文件中还支持嵌套标签，因此在 jsp:useBean 声明语句中可以嵌套使用 jsp:setProperty 命令。（如果 jsp:setProperty 在 jsp:useBean 声明语句外使用，那么一个解释器就会发生。）

globals.jsa 的结构

当在 globals.jsa 的事件处理器中使用一个全局对象时，这个全局对象声明的位置是很重要的，只有那些在某个特定的事件处理器之前被声明的对象才作为隐式对象加入这个事件处理器。由于这个原因，建议开发人员按如下顺序来组织 globals.jsa 文件的结构：

- 1、全局指令语句
- 2、全局对象
- 3、事件处理器
- 4、globals.jsa 声明

全局声明和指令语句例子

下面举出了一个 globals.jsa 文件的例子，它完成了以下功能：

- 定义了 JML 标签库（在本例中是编译时实现），可以在 globals.jsa 文件中使用，也可以在后面所有的 JSP 页面中使用。
通过在 globals.jsa 文件中包含 taglib 指令语句，在应用程序中其他任何一个 JSP 页面中就不需要重新包含这个指令了。
- 声明三个应用程序变量，可以在所有的页面中使用（在例子中是通过 jsp:useBean 语句完成的）。

要获得使用 `globals.jsa` 文件来进行全局声明的其他的例子，请参看 “全局声明 `global.jsa` 实例——`index2.jsp`）。

例子的程序代码如下所示：

```
<%-- Directives at the top --%>

    <%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>

<%-- Declare global objects here --%>

<%-- Initializes counts to zero --%>
    <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope = "application" />
    <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope = "application"
/>
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope = "application"
/>

<%-- Application lifecycle event handlers go here --%>

    <event:application_OnStart>
        <% This scriptlet contains the implementation of the event handler %>
    </event:application_OnStart>

    <event:application_OnEnd>
        <% This scriptlet contains the implementation of the event handler %>
    </event:application_OnEnd>

    <event:session_OnStart>
        <% This scriptlet contains the implementation of the event handler %>
    </event:session_OnStart>

    <event:session_OnEnd>
        <% This scriptlet contains the implementation of the event handler %>
    </event:session_OnEnd>

<%-- Declarations used by the event handlers go here --%>
```

第 6 章 JSP 的解释和发布

本章主要讨论在 Oracle Servlet Engine 环境下将 JSP 应用程序发布到 Oracle8i 数据库中的过程和应考虑的问题，同时也描述了 OracleJSP 的通用解释功能。另外，本章还简要讨论了其他环境下，尤其是 Oracle Internet Application Server 所使用的 Apache/JServ 环境下的 JSP 应用程序的发布问题。

本章的主要内容包括：

- OracleJSP 解释器的功能
- 发布到 Oracle8i 中的逻辑和特性概述
- 解释并发布到 Oracle8i 所使用的工具和命令
- 发布到 Oracle8i 服务器端解释
- 发布到 Oracle8i 客户端解释

6.1 OracleJSP 解释器的功能

JSP 解释器的标准功能就是对一个 JSP 页面生成它对应的页面实现类，也就是标准的 Java 代码。页面实现类在本质上是一个 Servlet 类，它封装了 JSP 功能与特性。

本节讨论 OracleJSP 解释器的通用功能，主要着重于对按需解释环境（如 Apache/JServ 或者 Oracle Internet Application Server）的介绍，包括以下几部分的内容：

- 生成代码的特性
- 生成包和类的命名（按需解释）
- 生成的文件和位置（按需解释）
- 样例页面实现类源代码

注意：本节所讨论的关于包和类的命名、输出文件的定位以及生成的代码等功能的实现细节，只是出于演示目的。这些细节只适用于符合 JSP 1.1 标准的 OracleJSP 8.1.7(1.1.0.0.0)发行版，并且在未来的发行版中，这些细节将可能被改变。

如果你的目标平台是 Oracle Servlet Engine，那么你必须首先预解释 JSP 页面，这可以通过两种方法来完成：一种是运行会话 shell 的 `publishjsp` 命令（适用于服务器端解释的发布）；另外一种是直接运行 `ojspc` 预解释工具（适用于客户端解释的发布）。这两种方法的任何一种，都与本节所讨论的功能有些差异，如输出文件的存放位置等，请参看“在 Oracle8i 中解释并发行 JSP 页面（`publishjsp`）”以及“预解释工具 `ojspc`”。

6.1.1 生成代码的特性

本小节讨论页面而生成类代码的通用特性，这些代码是由 OracleJSP 解释器在解释 JSP 源文件（`.jsp` 和 `.sqljsp` 文件）时生成的。

页面实现类代码的特性

当 OracleJSP 解释器生成页面实现类的 Servlet 代码时，它会自动处理一些标准的编程问题，为开发人员减轻负担。对按需解释模型和预解释模型来说，生成的代码都自动地含有下列的特性：

- 它继承了类 `oracle.jsp.runtime.HttpJsp`，这个封装类是由 OracleJSP 容器所提供的，并且实现了标准的 `javax.servlet.jsp.JspPage`，又继承了标准的 `javax.servlet.Servlet` 接口）。
- 它实现了由 `HttpJspPage` 接口所指定的 `_jspService()` 方法，这个方法通常被广泛地叫做“Service”方法，是页面实现类的最重要的方法。JSP 页面中的任何 Java 脚本段和表达式代码都被包含在 Service 方法的实现中。
- 除非你的 JSP 源代码特意设置了 `session=false`（可以通过 `page` 指令来设置），它包含了必要的代码段去请求一个 HTTP 会话。

要获得关于重要的 JSP 和 Servlet 类与接口的参考信息，请参看附录 B “JSP 和 Servlet 的技术背景”。

静态文本的内部类

页面实现类的 `_jspService()` 方法包括打印命令——对隐式对象的 `out.print()` 方法的调用——来打印 JSP 页面中的任何静态文本。不过，OracleJSP 解释器将静态文本内容放在了页面实现类的一个内部类中，使得 `_jspService()` 方法中的 `out.print()` 语句引用内部类的属性来打印静态文本。

这个内部类的实现导致当页面解释和编译时产生一个额外的 `.class` 文件。在一个客户端预解释的应用场合下（通常用来发布到 Oracle8i），应该意识到这意味着一个额外的 `.class` 文件必须被发布。

内部类文件的名称总是与 `.jsp` 文件或者 `.jspx` 文件的基名（即除了扩展以外的部分）相同。例如，对 `mypage.jsp` 文件来说，它的内部类（`.class` 文件）的名称将总是包括“`mypage`”。

注意：OracleJSP 解释器可以任选地将静态文本内容放到一个 Java 资源文件中，这在页面里含有大量静态文本时是很有用的。要得到这一功能，你必须将 OracleJSP 的 `external_resource` 配置参数设为 `true`（在按需解释模型下）或者使用预解释工具 `ojspc` 的 `-extres` 选项参数（在预解释模型下）。如果允许了热加载（发布到 Oracle8i），那么也将导致静态文本内容被放到了一个资源文件中，内部类仍然要被生成，并且它的 `.class` 文件也必须被发布（这只在客户端预解释的应用场合下才值得使用）。

6.1.2 生成包和类的命名（按需解释）

尽管 Sun 公司的 JSP 1.1 技术标准中定义了一个统一的过程用来解析和解释 JSP 文本，但是它没有描述生成的类应如何被命名——命名机制仍然取决于各个特定的 JSP 实现。

本小节描述 OracleJSP 是如何对解释过程中生成的代码中的包和类来命名的。

注意：在 OracleJSP 8.1.7(1.1.0.0.0)发行版中，URL 路径目录和.jsp 文件名（用来生成包和类的名字）被限制必须是有效的 Java 包和类标识符。例如，一个路径目录或者一个.jsp 文件名不能以数字开头，并且不能使用 Java 保留字，如 for, class 等（class.jsp 就不是一个有效的文件名）。这一限制在将来的发行版本中将有可能会更改。

包命名

在一个按需解释的应用场合下，用户通过请求一个 JSP 页面来指定的 URL 路径决定了生成的页面实现类中包的名字。需要注意的是，这个路径是相对于文档根目录或者应用程序根目录的路径。URL 路径中的每一个目录都代表了包的分层名字中的一层。

另外需要特别注意的一点就是：生成的包总是由小写字母组成的，与 URL 路径中的大小写无关。

考虑下面的 URL 路径：

`http://host[:port]/HR/expenses/login.jsp`

在 OracleJSP 8.1.7(1.1.0.0.0)发行版中，上面的 URL 将导致在页面生成类的代码中出现以下的 package 指令（实现细节在未来的发行版本可能要改变）：

```
package hr.expenses;
```

如果 JSP 页面处于文档根目录或应用程序根目录下的话，那么包的名字将不会被生成，如下面的 URL 所示：

`http://host[:port]/login.jsp`

类命名

.jsp 文件（或者.sqljsp 文件）的基名决定了生成代码中类的名字，考虑下面的 URL 例子：

`http://host[:port]/HR/expense/userLogin.jsp`

在 OracleJSP 8.1.7(1.1.0.0.0)发行版中，上面的 URL 将导致在生成的代码中出现以下类名（实现细节在未来的发行版中可能要改变）：

```
public class userLogin extends...
```

应该注意的是，最终用户在 URL 中所键入字母的大小写应该和实际的.jsp 或者.sqljsp 文件名中字母的大小写完全匹配。例如，如果实际的文件名是 UserLogin.jsp 时，那么用户可以在 URL 指定 UserLogin.jsp；如果实际的文件名是 userlogin.jsp 时，那么用户可以在 URL 中指定 userlogin.jsp；如果实际的文件名是 UserLogin.jsp 时，那么用户在 URL 中指定 userlogin.jsp 就会出错。

在 OracleJSP 8.1.7(1.1.0.0.0)发行版中，解释器根据文件名字母的大小写来决定类名字的大小写。例如：

- UserLogin.jsp 对应类名 UserLogin
- Userlogin.jsp 对应类名 Userlogin
- userlogin.jsp 对应类名 userlogin

如果你对类名的大小写很在意的话，那么就必须适当地命名.jsp 文件与.sqljsp 文件的名字。不过由于页面实现类对最终用户是不可见的，因此这些通常可以不作考虑。

6.1.3 生成的文件和位置（按需解释）

本小节描述 OracleJSP 解释器所生成的文件以及它们被存放的位置。对预解释应用场合来说，ojspc 对文件的存放位置是不同的，并且有它自己的一套相关选项参数——请参看“ojspc 的参数小结表”以获得相关信息。

下面的几个小节提到了一些 OracleJSP 的配置参数。要获得关于它们的更多信息，请参看“OracleJSP 在非 OSE 环境下的配置参数”以及“OracleJSP 的配置参数设置”。

OracleJSP 生成的文件

本小节考虑了由 OracleJSP 解释器所生成的列表文件中的规则的 JSP 页面（.jsp 文件）和 SQLJ JSP 页面（.sqljsp 文件）。为了提供文件名的例子，下面假设要解释的文件名是 Foo.jsp 或者 Foo.sqljsp。

由 OracleJSP 所生成的文件有：

源文件：

- 如果页面是一个 SQLJ JSP 页面，那么 OracleJSP 解释器会生成一个 .sqlj 文件（例如，Foo.sqlj）。
- 对页面实现类和内部类生成一个 .java 文件（例如，Foo.java）。此文件要么是由 OracleJSP 解释器直接生成的，要么是由 SQLJ 解释器从 .sqlj 文件生成的。（缺省情况下使用当前安装的 Oracle SQLJ 解释器，但是你可以指定一个其他的 SQLJ 解释器或者使用不同版本的 Oracle SQLJ 解释器，这是通过配置 sqljcmd 参数来实现的。）

二进制文件：

- 如果页面是一个 SQLJ 页面，那么在 SQLJ 的解释过程中将会有有一个或多个二进制文件被生成。缺省情况下，这些文件的后缀名是 .ser，它们是 Java 资源文件，但是如果你允许了 SQLJ -ser2class 选项参数（通过使用 OracleJSP 的 sqljcmd 配置参数），那么这些文件将是 .class 文件资源文件或者 .class 文件都将“Foo”值作为名字的一部分。
- 对页面实现类 Java 编译器要生成一个 .class 文件（缺省情况下，Java 编译器是标准的 javac，但是你可以指定一个其他的 Java 编译器来代替 javac，这是通过 OracleJSP 的配置参数 javacmd 来实现的）。
- 对页面实现类的内部类生成一个额外的 .class 文件，此文件以“Foo”作为名字的一部分。
- 如果 OracleJSP 的配置参数 external_resource 被设置为 true 时，对静态页面内容，一个 .res（Java 资源文件）可以任选地被生成（例如 Foo.res）。

注意：在 OracleJSP 的未来发行版本中，对页面实现类的生成文件的名字有可能被更改，但是将仍然保留通用的形式。名字将总是包括文件的基名（如这些例子中的“Foo”），但是可能包含一些细小的改变，如 _Foo.java 或者 _Foo.class。

OracleJSP 解释器的输出文件定位

OracleJSP 使用 Web Server 的文档目录来生成或者加载 JSP 页面。

缺省情况下，根目录就是 Web Server 的文档根目录（对 Apache/JServ 来说）或者页面属于的应用程序的 Servlet 环境根目录。

你可以通过配置 OracleJSP 的 `page_repository_root` 参数来指定另外一个根目录来替代缺省的根目录。

在 OracleJSP 8.1.7(1.1.0.0.0)发行版中，生成的文件的定位如下所示（实现细节可能在未来的版本中改变）：

- 如果 .jsp（或者 .sqljsp）文件在根目录下，那么 OracleJSP 将把生成的文件存放到根目录下的 `_pages` 子目录下。
- 如果 .jsp（或者 .sqljsp）文件在根目录下的某一个子目录中，那么在根目录下的 `_pages` 子目录下将会有对应的子目录被创建，并且生成的文件被存放到这个新建的子目录下。

作为一个例子，考虑在 Apache/JServ 环境下有一个根目录为 `htdocs`，如果一个 .jsp 文件在下面的目录中：

`htdocs/subdir/test`

那么生成的文件将会被放到如下的目录中：

`htdocs/_pages/subdir/test`

6.1.4 页面实现类的实例源代码

本节使用一个例子来演示前面几节所讲述的内容，假设具有以下条件：

- JSP 页面代码位于文件 `hello.jsp` 中
- 页面在 Apache/JServ 环境中执行
- `hello.jsp` 文件的位置位于以下目录：

`htdocs/test`

注意：这里所讨论的代码生成的细节实现是根据 Oracle 的 JSP 1.1 技术标准来的，这些细节在未来的版本中可能改变，因为要么标准本身会改变，要么 Oracle 的特定实现会改变。

页面代码实例：`hello.jsp`

下面是 `hello.jsp` 文件的 JSP 代码：

```
<HTML>
<HEAD><TITLE>The Hello User JSP</TITLE></HEAD>
<BODY>
<% String user=request.getParameter("user"); %>
<H3>Welcome <%= (user==null) ? "" : user %>!</H3>
<P><B> Today is <%= new java.util.Date() %>. Have a nice day! :-)</B></P>
<B>Enter name:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="user" SIZE=15>
<INPUT TYPE="submit" VALUE="Submit name">
</FORM>
</BODY>
</HTML>
```

实例：生成的包和类

因为 `hello.jsp` 在根目录（`htdocs`）下的 `test` 子目录中，因此 OracleJSP 8.1.7(1.1.0.0.0) 发行版在页面实现类代码中生成下面的包名：

```
package name;
```

生成的 Java 类的名字与 `.jsp` 文件（大小写敏感）的基名是相同的，因此在页面实现类代码中将会生成以下的类定义代码：

```
public class hello extends oracle.jsp.runtime.HttpJsp
{
    ...
}
```

（因为页面实现类对最终用户是不可见的，所以页面实现类的名字不遵守 Java 的大小写命名规则，这一事实并不会成为一个问题。）

实例：生成的文件

因为 `hello.jsp` 文件的存放路径如下：

```
htdocs/test/hello.jsp
```

所以 OracleJSP 8.1.7(1.1.0.0.0) 发行版将会生成以下的输出文件（页面实现类 `.java` 文件和 `.class` 文件，以及内部类的 `.class` 文件）：

```
htdocs/_pages/test/hello.java
htdocs/_pages/test/hello.class
htdocs/_pages/test/hello$_jsp_staticText.class
```

注意：这些文件名是基于 OracleJSP 1.1.0.0.0 标准的，精确的细节在未来的发行版本中可能要改变，不过有一点不会改变，那就是所有的文件名都必须含有“hello”。

页面实现代码实例：`hello.java`

下面是 OracleJSP 8.1.7(1.1.0.0.0) 发行版所生成的页面实现类的 Java 代码（`hello.java`）：

```
package test;

import oracle.jsp.runtime.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
import java.io.*;
import java.util.*;

import java.lang.reflect.*;
import java.beans.*;

public class hello extends oracle.jsp.runtime.HttpJsp {

    public final String _globalsClassName = null;
    // ** Begin Declarations

    // ** End Declarations
```

```

    public void _jspService(HttpServletRequest request, HttpServletResponse
response) throws IOException, ServletException {

        /* set up the intrinsic variables using the pageContext goober:
        ** session = HttpSession
        ** application = ServletContext
        ** out = JspWriter
        ** page = this
        ** config = ServletConfig
        ** all session/app beans declared in globals.jsa
        */
        JspFactory factory = JspFactory.getDefaultFactory();
        PageContext pageContext = factory.getPageContext( this, request,
response,null, true, JspWriter.DEFAULT_BUFFER, true);
        // Note: this is not emitted if the session directive == false
        HttpSession session = pageContext.getSession();
        if (pageContext.getAttribute(OracleJspRuntime.JSP_REQUEST_REDIRECTED,
PageContext.REQUEST_SCOPE) != null) {
            pageContext.setAttribute(OracleJspRuntime.JSP_PAGE_DONTNOTIFY,
"true",PageContext.PAGE_SCOPE);
            factory.releasePageContext(pageContext);
            return;
        }

        ServletContext application = pageContext.getServletContext();
        JspWriter out = pageContext.getOut();
        hello page = this;
        ServletConfig config = pageContext.getServletConfig();

        try {
            // global beans
            // end global beans
            out.print(__jsp_StaticText.text[0]);
            String user=request.getParameter("user");
            out.print(__jsp_StaticText.text[1]);
            out.print( (user==null) ? "" : user );
            out.print(__jsp_StaticText.text[2]);
            out.print( new java.util.Date() );
            out.print(__jsp_StaticText.text[3]);
            out.flush();
        }
        catch( Exception e) {
            try {
                if (out != null) out.clear();
            }
            catch( Exception clearException) {
            }
            pageContext.handlePageException( e);
        }
        finally {
            if (out != null) out.close();
        }
    }

```

```

        factory.releasePageContext (pageContext);
    }
}
private static class __jsp_StaticText {
    private static final char text[][]=new char[4][];
    static {
        text[0] =
            "<HTML>\r\n<HEAD><TITLE>The Welcome User
JSP</TITLE></HEAD>\r\n<BODY>\r\n".toCharArray();
        text[1] =
            "\r\n<H3>Welcome ".toCharArray();
        text[2] =
            "!</H3>\r\n<P><B> Today is ".toCharArray();
        text[3] =
            ". Have a nice day! :-)</B></P>\r\n<B>Enter name:</B>\r\n<FORM
METHOD=get>\r\n<INPUT TYPE=\"text\" NAME=\"user\" SIZE=15>\r\n<INPUT
TYPE=\"submit\" VALUE=\"Submit
name\">\r\n</FORM>\r\n</BODY>\r\n</HTML>".toCharArray();
    }
}
}
}

```

6.2 发布过程中的逻辑与特性概述

本节简要概述了在 Oracle Servlet Engine 下如何将 JSP 应用程序发布到 Oracle8i 数据库中，包括其中的逻辑以及应该考虑的问题等，涵盖了以下几部分的内容：

- 数据库的 Java Schema 对象
- 用 Oracle HTTP Server 作为前端 Web Server
- Oracle Servlet Engine 中的 URLs
- Oracle Servlet Engine 中 JSP 应用程序的静态文件
- 服务器端解释与客户端解释的比较
- Oracle8i 中的热加载功能概述

6.2.1 数据库的 Java Schema 对象

在 Oracle Servlet Engine 环境下执行的 Java 代码都使用数据库内部的一个 Oracle8i Java 虚拟机，并且 Java 代码必须作为一个或者多个 Schema 对象被加载到一个特定的数据库 Schema 中。

对 Java 来说有以下三种 Schema 对象：

- 源 Schema 对象（对应 Java 源文件）
- 类 Schema 对象（对应 Java 类文件）
- 资源 Schema 对象（对应 Java 资源文件）

每个 Schema 对象在数据库中都是一个独立的库单元，当你查询 Schema 的 ALL_OBJECTS 表时，你会看到 Java Schema 对象有以下三种类型：JAVA SOURCE，JAVA CLASS 和 JAVA RESOURCE。

加载 Java 文件来创建 Schema 对象

JServer 的 loadjava 工具是用来将 Java 文件以 Schema 对象的形式加载到数据库中。

当你在客户端编译并直接加载.class 文件时，loadjava 工具将.class 文件作为一个类 Schema 对象存储在数据库中。

当你加载一个.java（或者.sqlj）源文件时，loadjava 工具将源文件作为一个源 schema 对象存储在数据库中，并且作为任选的步骤，它还在数据库内部编译源文件并且创建一个或多个类 schema 对象。

当你加载一个资源文件（如静态 JSP 内容所对应的.res 文件或者 SQLJ 所对应的.ser 文件）时，loadjava 工具将资源文件作为一个资源 Schema 对象存储在数据库中。

当你加载一个.jsp 或者.sqljsp 页面源文件时（对服务器端解释），loadjava 工具将页面源文件作为一个资源 Schema 对象存储在数据库中。在服务器端解释过程中（通过 JServer 的 shell 命令 publishjsp），服务器端的 loadjava 将被自动调用，并且在解释和编译过程中它将创建源 Schema 对象、类 Schema 对象以及资源 Schema 对象。

Schema 对象的全名和短名

在 Oracle8i 中有两种形式的 Schema 对象命名机制：全名和短名。对应一个 Schema 对象来说，只要在可能的情况下都使用全名，不过如果全名包含了多于 31 个字符，或者包含了无效的字符，或者包含的字符中有一部分不能被转化成数据库字符集中的字符，在这些情况下 Oracle8i 将把全名转换成一个短名并且作为 Schema 对象的名字。如果全名包含了 31 个以内的字符，并且没有无效或者不可转换的字符，那么全名就将用来作为 Schema 对象的名字。

要获得关于这两种命名以及其他文件命名的相关信息，包括可以从一个短名中得到全名或者从一个全名中得到短名的 DBMS_JAVA 过程的信息，请参看“Oracle8i Java 开发人员指南”。

在加载过程中决定 Java Schema 对象的包

在把 Java 文件加载到数据库的过程中，loadjava 工具将使用以下的逻辑来决定它所创建的 Schema 对象的包：

- 对源 Schema 对象（从.java 和.sqlj 文件创建）和类 Schema 对象（从.class 文件创建或者通过编译.java 文件创建）来说，Java 代码中的包信息决定了 Schema 对象的包。

例如，有一个类 Foo 并且指定了包 dir1.dir2，如果它被加载到一个名字叫做 SCOTT 的 Schema 对象中，那么 Foo 类将在 Schema 对象中以如下形式储存：

SCOTT: dir1/dir2/Foo

注意：如果使用 ojspc 工具来预解释一个 JSP 页面（适用于在客户端解释并发布到 Oracle8i 中），你可以通过 ojspc -packageName 参数来指定生成的.java 文件的包。

- 对源 Schema 对象（从.res 和.ser 等 Java 资源文件创建）来说，loadjava 命令中的路径信息决定了 Schema 对象的包（如果 Java 资源文件被直接加载）或者由 JAR

文件的路径信息来决定 Schema 对象的包（如果 Java 资源文件作为 JAR 文件的一部分被加载）。

例如，有一个资源文件/dir3/dir4/abcd.res，如果它被加载到一个名字叫做 SCOTT 的 Schema 对象中，那么它将在 Schema 对象中以如下形式存储：

SCOTT: dir3/dir4/abcd.res

发行 Schema 对象

在 Oracle Servlet Engine 下运行的任何 JSP 页面（或者 Servlet）都必须被“发行”。所谓发行，就是一个处理过程，通过这个处理过程，使得 JSP 页面的可执行 Java 代码（类 Schema 对象）通过 JServer 的 JNDI 命令空间入口可以被访问。

发行一个 JSP 页面时，要把它的页面实现类的 Schema 对象与一个 Servlet 路径（以及可选的一个非缺省的 Servlet 环境路径）链接起来。这个 Servlet 路径（或者 Servlet 环境路径）将是最终用户请求这个页面时所输入的 URL 的一部分。

发行一个 JSP 页面，可以使用 Oracle8i 的 shell 命令 publishjsp（适用于在服务器端解释的发布应用场合）或者 shell 命令 publishservlet（适用于在客户端解释的发布应用场合）。

6.2.2 用 Oracle HTTP Server 作为前端 Web Server

尽管可以使用 Oracle Servlet Engine 自身来作为 Web Server，但对于运行在 OSE 下的 JSP 页面和 Servlets 来说，最典型的应用还是通过 Oracle HTTP Server 和它的 mod_ose 模块来访问它们。

要获得关于 Oracle HTTP Server 与 mod_ose 的功能的更多信息，请参看“Oracle HTTP Server 所扮演的角色”。

6.2.3 Oracle Servlet Engine 中的 URLs

在 Oracle Servlet Engine 环境下，最终用户请求一个 JSP 页面所输入的 URLs（对 Servlet 的 URL 也适用）由以下两部分所组成（除去 hostname 和 port）：

- 在 OSE 中 Servlet 环境的路径，它在 Servlet 环境被创建时确定。
- 在 OSE 中 JSP 页面的 Servlet 路径（常常指向一个“虚拟路径”），它在 JSP 页面被发行时确定。

OSE 缺省的 Servlet 路径是/webdomains/contexts/default，可以简写为：/

对于适用 Oracle8i 的 shell 命令 createcontext 所创建的其他 Servlet 环境路径来说，它就是你在 createcontext-virtualpath 选项参数中所指定的路径。（指定环境路径与环境名字相同只是一个惯例，不是必需的）。

注意：无论什么时候使用 createcontext，都必须指定-virtualpath 参数。

对 Servlet 路径（即 JSP 页面的“虚拟路径”）来说，它是用你发行 JSP 页面的方式确定的，如下所示：

- 如果你使用 shell 命令 publishjsp（适用于服务器端解释），那么 Servlet 路径是由 publishjsp -virtualpath 参数所指定的路径决定的，或者缺省地与指定的 Schema 路

径相同。

- 如果你使用 shell 命令 `publishservlet` (适用于客户端解释), 那么 Servlet 路径是由 `publishservlet -virtualpath` 参数所指定的路径决定的。(当你使用 `publishservlet` 来发行一个 JSP 页面时, 你必须指定 `-virtualpath` 参数。)

例一: 作为一个例子, 考虑一个 JSP 页面被发行到缺省的 OSE 环境路径下, 此页面具有如下的 `servlet` 路径:

```
mydir/mypage.jsp
```

那么这个页面的 URL 就有如下的形式:

```
http://host[:port]/mydir/mypage.jsp
```

你可以从应用程序中其他的页面 (比如 `mydir/mypage2.jsp`) 中通过下面两种方式中的一种来访问上面所讨论的 JSP 页面 (第一种使用页面相对路径, 第二种使用应用程序相对路径):

```
<jsp:include page="mypage.jsp" flash="true" />
<jsp:include page="/mydir/mypage.jsp" flash="true" />
```

例二: 现在考虑一个 Servlet 环境, 它由下面的命令来创建 (\$ 是 session shell 的命令提示符):

```
$ createcontext -virtualpath mycontext /webdomains mycontext
```

这条命令有以下作用:

- 它创建了 Servlet 环境 `/webdomains/contexts/mycontext` (在 `/webdomains` 域中的所有 Servlet 环境都在子目录 `/webdomains/context` 下。)
- 它指定了环境路径与环境名字 (`mycontext`) 相同。

如果 `mydir/mypage.jsp` 被发行到了 Servlet 环境 `mycontext` 中, 那么它的 URL 就有如下的形式:

```
http://host[:port]/mycontext/mydir/mypage.jsp
```

你可以从应用程序中其他的页面 (比如 `mydir/mypage2.jsp`) 通过下面两种方式中的一种来访问 `mypage.jsp` (第一种使用页面相对路径, 第二种使用应用程序相对路径):

```
<jsp:include page="mypage.jsp" flash="true" />
<jsp:include page="/mydir/mypage.jsp" flash="true" />
```

对动态包含语句 `jsp:include` 来说, 语法与例一中所使用的相同。

即使使用了一个不同的 Servlet 环境, 但是对这个 Servlet 环境的页面相对路径仍然是不变的。

例三: 现在考虑一个 Servlet 环境, 它由下面的命令来创建 (\$ 是 session shell 的命令提示符):

```
$ createcontext -virtualpath my /webdomains mycontext
```

这条命令有以下作用:

- 它创建了 Servlet 环境 `/webdomains/contexts/mycontext`, 与例二相同。
- 它定义了一个环境路径 `mywebapp`, 与环境名字不同。在 URL 中使用的是环境路径, 而不是环境名字。

在本例中, 如果 `mydir/mypage.jsp` 被发行到了 Servlet 环境 `mycontext` 中, 那么它的 URL 就有如下形式:

```
http://host[:port]/mywebapp/mydir/mypage.jsp
```

你可以从应用程序中其他的页面（比如 mydir/mypage2.jsp）通过下面两种方式中的一种来访问 mypage.jsp（第一种使用页面相对路径，第二种使用应用程序相对路径）：

```
<jsp:include page="mypage.jsp" flash="true" />
<jsp:include page="/mydir/mypage.jsp" flash="true" />
```

6.2.4 OSE 环境下 JSP 应用程序中所使用的静态文件

本小节描述了在 OSE 环境下运行的 JSP 应用程序中所使用的静态文件（如 HTML 文件），以及为了正常运行它们应被放置的路径。

本小节所提供的信息是与使用什么 Web Server 无关的，也就是说无论是使用 Oracle HTTP Server 作为前端 Web Server 还是直接使用 OSE 自身，本节的内容都适用。

动态包含和导向的文件

在 Oracle Servlet Engine 环境下运行的 JSP 应用程序中，所有被动态包含或导向（适用 `jsp:include` 或 `jsp:forward` 语句）的静态文件都需要被手工移动或复制到对应于应用程序 Servlet 环境的 OSE 文档根目录下。当你创建一个 OSE Servlet 环境时（使用 shell 命令 `createcontext`），你通过 `createcontext -docroot` 选项参数来指定了一个文档根目录，每个 OSE 文档根目录都被链接到 JServer 的 JNDI 命名空间中。

OSE 文档根目录是位于数据库外面的，而 JNDI 对静态文件的查找机制是数据库所在服务器的文件系统的前端工具。

对 OSE 缺省的 Servlet 环境/webdomains/contexts/default 来说，对应的文档根目录如下：

```
$ ORACLE_HOME/jis/public_html
```

当你使用 shell 命令 `createcontext` 命令来创建一个额外的 Servlet 环境时，你总是能使用 `createcontext -docroot` 选项参数来指定一个文档根目录。

注意：如果你正在将自己的 JSP 应用程序从 Apache 移植到 OSE，那么建议你吧 Apache 文档根目录下的静态文件复制到 OSE 的 Servlet 环境文档根目录下，最好不要使用把 OSE 的 Servlet 环境文档根目录映射到 Apache 的文档根目录这一方法，因为它最终会导致混淆。

静态包含的文件

被一个 JSP 页面所静态包含（使用 `include` 指令语句）的任何文件，不管它是另外一个 JSP 页面或者是一个静态文件（如 HTML 文件），都必须在解释过程中可以被 OracleJSP 解释器访问到。

如果一个 JSP 应用程序的目标平台是 OSE，那么有以下两种解释方法：

- 服务器端解释

在这种应用场合下，你首先将一个 .jsp 文件作为一个 Java 资源加载到数据库中，接着使用 `publishjsp` 命令去调用服务器端的 OracleJSP 解释器。

并且，在这种情况下你必须预先将静态文件作为资源 Schema 对象加载到数据库中，这可以通过使用 `loadjava` 工具来完成。

- 客户端解释

在这种应用场合下，你首先在客户端使用 `ojspc` 工具解释 JSP 文件，然后将生成的文件加载到数据库中即可。

并且，在这种情况下，静态文件可以不必在服务器端，它们只要在客户端解释的过程中能由 `ojspc` 访问到就行了。

6.2.5 服务器端解释与客户端解释的比较

要在 Oracle Servlet Engine 环境下将 JSP 页面发布到 Oracle8i 数据库中，开发人员有两种选择：他们既可以在客户端解释页面，也可以在服务器端解释页面。

在服务器端解释的发布过程可以分为如下两个步骤：

- 1、运行 `loadjava` 工具，将 JSP 页面源文件（.jsp 或者 .sqljsp 文件）作为一个资源 Schema 对象加载到 Oracle8i 数据库中。（同时你也必须将任何需要的 Java 类或者其他需要的 JSP 页面加载进数据库。）
- 2、运行 shell 命令 `publishjsp`，它将自动完成以下功能：
 - 将 JSP 页面源文件（.jsp 或者 .sqljsp 文件）解释成页面实现类的 Java 代码（如果源文件是一个 SQLJ JSP 页面文件（.sqljsp 文件），那么首先要生成 SQLJ 源文件，然后再调用 SQLJ 解释器）。
 - 将 Java 代码编译成一个或多个类文件。
 - 热加载页面实现类（这一功能是任选的，只有你指定了 `publishjsp -hotload` 参数时此功能才启用）。
 - 将页面实现类发行到数据库中以待以后运行。此步骤也同时生成源 Schema 对象、类 Schema 对象以及资源 Schema 对象，这些 Schema 对象分别对应生成的 .java 文件（对 .sqljsp 页面来说是 .sqljsp 文件）、.class 文件和资源文件。

在客户端解释的发布过程需要如下三个（或者任选的四个）步骤：

1. 运行 OracleJSP 的预解释工具 `ojspc`，它将自动完成以下功能：
 - 将 JSP 页面源文件（.jsp 或者 .sqljsp 文件）解释成页面实现类的 Java 代码（如果源文件是一个 SQLJ JSP 页面文件（.sqljsp 文件），那么首先要生成 SQLJ 源文件，然后调用 SQLJ 解释器来生成 Java 代码）。
 - 依赖于 `ojspc -extres` 和 `-hotload` 两个参数的设置，任选地将静态文本内容转换成 Java 资源文件。
 - 将 Java 代码编译成一个或者多个类文件。
2. 运行 Oracle8i 的 `loadjava` 工具，将类文件和所有资源文件以类 Schema 对象和资源 Schema 对象的形式加载到 Oracle8i 数据库中。
3. 使用 Oracle8i 的 `session`、`shell` 命令 `java` 来执行页面实现类的 `main()` 方法，从而热加载类（这一步骤是任选的，只有在编译过程中指定了 `ojspc -hotload` 选项参数时此步骤才启用）。
4. 运行 `session shell` 命令 `publishservlet`，将页面实现类发行到数据库中以备以后运行。

如果使用 Oracle 的 JDeveloper 开发工具，你将会发现一个便利的发布途径：首先在客户端使用 JDeveloper 提供的 OracleJSP 解释器去解释页面，然后按照步骤 2、3、4 将输出

的类文件和资源文件发布。

不过，如果你不使用 JDeveloper，那么在服务器端解释将是一个比较好的选择，因为 `publishjsp` 命令可以将解释、热加载（任选的）以及发行组合到一个步骤中去。

另外，在下面所列的两种情况中的任意一种，都需要在服务器端解释：

- 如果所需的库在客户端是不可用的
- 如果你想编译在运行时使用的一套类

6.2.6 Oracle8i 中的热加载功能概述

Oracle8i JServer 提供了一个特有功能——热加载类，用来更有效率地使用静态的最终变量（即常量），特别是在多个用户并发地同时访问页面时，热加载类这个功能就显得尤为重要了。

每个 JServer 数据库会话都需要调用一个独立的 Java 虚拟机。标准情况下，每个会话从自己的会话空间获得它的所有静态最终变量的拷贝，不过在使用原义字符串（`literal strings`）的情况下，它从共享内存中的一个哈希表（被称为内部表）中获得静态最终变量。在内部表中使用原义字符串可以跨越会话空间。

原义字符串的处理过程在本质上是与 JSP 页面相关的。缺省情况下（不使用热加载），JSP 页面中的静态文本最终要被表达为原义字符串。

注意：本节运用了以下的工具：OracleJSP 的预解释工具 `ojspc`、Oracle 的 session shell 工具 `sess_sh` 以及 session shell 的 `publishjsp` 命令。

允许并使用热加载功能

要想允许热加载 JSP 页面的功能，可以通过 `ojspc -hotload` 选项（对客户端解释）或者 `publishjsp -hotload` 选项（对服务器端解释）来实现。

允许 `-hotload` 选项将导致 OracleJSP 解释器完成以下动作：

- 在页面实现类中生成相应的代码以允许热加载功能，这是通过创建一个 `hotloading` 方法，并且在 `main()` 方法中调用 `hotloading` 方法来达到目的的。
- 将静态文本内容写至一个 Java 资源文件中。（否则，静态文件内容将被写至页面实现类的一个内容类中）

热加载功能是通过以下步骤来完成的：

- 对客户端解释的发布过程来说，你必须将热加载作为一个额外的发布资源。
首先使用 `ojspc -hotload` 选项允许热加载功能并解释页面，然后将页面加载到数据库中。在这个步骤之后，同时在发行此页面之前，你必须使用 session shell 命令 `java` 来调用页面实现类的 `main()` 方法。
- 对服务器端解释的发布过程来说，当你允许 `publishjsp -hotload` 选项后，热加载会作为 `publishjsp` 功能的一部分被自动完成。

不管是直接通过 shell 命令 `java` 还是间接通过 `publishjsp` 命令，热加载一个页面实现类的作用实际上就是使用内部类中的静态文本内容在数据库的多个 Java 虚拟机中可以共享。

热加载功能的特征与优点

热加载类这一功能具有如下的特征与优点：

- 解释器生成代码并用这些代码在静态初始化函数中去读包含有静态文本的 Java 资源，初始化存放静态文本的字符数组。
- 在热加载的过程中，每个被热加载的内部类仅仅被初始化一次，并且静态 JSP 文本被转换成静态 Java 字符数组的过程也只有一次。

这些字符数组并没有被存储在同步化的内部表中，相反它们被存储在一个全局的内存区域中，这个内存区域可以被所有的会话所共享并且不需要同步（这一方案是相当可行的，因为预先已经指定此区域中所有的变量都是不变的）。

热加载避免了同步操作以及其他一些代价很高的操作，因此它能在很大程度上提升运行在 Oracle Servlet Engine 环境下的 JSP 原义程序的性能，并且能改善它们的适用性。进一步来说，当一个热加载的类被引用时，这个类的初始化函数并不返回，会话可以立即访问原义字符串和其他的静态最终变量。

除了允许使单个的 JSP 页面获得更好的性能，热加载功能还降低了服务器 CPU 的占用率。

注意：对应那些不存在多用户同时并发访问这种情况的 JSP 页面，或者只具有很少的原义字符串的小型 JSP 页面来说，热加载所带来的性能上的提升很小甚至没有。

6.3 解释并发布到 Oracle8i 所使用的工具和命令

为了解释 JSP 页面并且将它们发布到 Oracle8i 数据库中，Oracle 提供了下面所列的几个工具以供使用。这些工具具体是怎样实现的要取决于你自己所用的操作系统（例如，对 Solaris 操作系统来说，这些工具可能是用 shell 的脚本语言所实现的，而对于 Windows NT 操作系统来说，它们可能是通过批处理 .bat 文件来实现的）。

- ojspc（OracleJSP 的预解释工具）
- loadjava（将 JSP 页面或者 Java 文件加载到数据库的工具）
- sess_sh（Oracle8i 的 session shell 工具）

在客户端解释的发布过程需要所有的上面所列的工具。在客户端预解释 JSP 页面需要使用 ojspc，将解释后的页面加载到 Oracle8i 数据库中需要使用 loadjava，并且最后的发行过程需要适用 session shell 的 publishervlet 命令。

在服务器端解释的发布过程不需要 ojspc。将没有解释过的 JSP 页面加载到 Oracle8i 数据库中需要使用 loadjava，解释和发行这些页面需要使用 session shell 的 publishjsp 命令。

loadjava 和 sess_sh 工具都是 Oracle8i JServer 环境所提供的通用工具，而 ojspc 是专门用于 JSP 页面的专用工具。

注意：

- 如果你想要将应用程序编译成本机代码（native code）并且在 Oracle8i 数据库中运行，那么有另外一个工具——JServer Accelerator 可供使用。这个

工具的调用形式是 ncomp。

- 本节所讨论的工具可以在[ORACLE_HOME]/bin 目录下找到。

6.3.1 预解释工具 ojspc

在客户端解释的应用场合下，将 JSP 应用程序发布到 Oracle8i 数据库中的第一步就是运行 OracleJSP 预解释工具——ojspc。

然后你可以使用下一节将要介绍的 loadjava 工具将第一步生成的.class 文件以及资源文件（如果有的话）分别以类 schema 对象和资源 Schema 对象加载到数据库中。

本节主要讨论以下几部分的内容：

- ojspc 的功能概述
- ojspc 的参数小结表
- ojspc 的命令行语法
- ojspc 的选项说明
- ojspc 的输出文件、存放位置以及相关选项小结

注意：在其他场合下（如在一个中间层环境）也有可能使用 ojspc 来预解释 JSP 页面，要获得相关信息，请参看“在非 OSE 环境下使用预解释工具 ojspc”。

ojspc 的功能

对一个简单的 JSP 页面（不适合于 SQLJ JSP 页面）来说，ojspc 的缺省功能如下所示：

- 它需要一个.jsp 文件作为输入参数。
- 它调用 OracleJSP 解释器，将.jsp 文件解释成 Java 页面实现类代码，并且生成一个.java 文件。页面实现类包括一个内部类，它含有静态文本内容。
- 它调用 Java 的编译器，将.java 文件编译并生成两个.class 文件（一个是对应于页面实现类自身，另外一个是对应于内部类的）。

而对于一个 SQLJ JSP 页面来说，ojspc 的缺省功能如下所示：

- 它需要一个.sqljsp 文件作为输入参数。
- 它调用 OracleJSP 解释器，将.sqljsp 文件解释成页面实现类（包括内部类）对应的.sqlj 文件。
- 它调用 Oracle SQLJ 解释器，将.sqlj 文件解释并生成两个文件，一个文件是页面实现类（包括内部类）所对应的.java 文件，另外一个.ser Java 资源文件，它实际上是一个 SQLJ 的“profile”文件。
- 它调用 Java 编译器，将.java 文件编译并生成两个.class 文件（一个是对应于页面实现类自身，另外一个对应于内部类）。

在某些条件下（参看下一节将要介绍的-hotload 和-extres 选项参数），ojspc 的选项参数将控制 OracleJSP 对静态页面内容生成一个.res Java 资源文件，而不是把静态内容放到页面实现类的内部类中。不过无论怎样，内部类的.class 文件始终都是要被创建的，并且这个文件也必须随着页面实现类一块发布。

注意: ojspc 命令行工具是一个前端实用工具, 它调用了 oracle.jsp.tool.Jspc 类来完成相应的功能。

ojspc 的参数小结表

表 6-1 描述了预解释工具 ojspc 所支持的选项参数, 关于这些选项参数的进一步讨论可以参看“ojspc 的选项说明”。

表中的第二列举出了在按需解释环境(如 Apache/JServ)下的 ojspc 的参数所对应或相关的 OracleJSP 配置参数。

注意: 要允许一个布尔(boolean)型的 ojspc 选项参数, 只需敲入此参数的名字即可, 不要将它设置成 true。如果将它设置成 true, 将会导致一个错误发生。

表 6-1 ojspc 预解释工具的选项参数

选项	相关的 OracleJSP 配置参数	描述	缺省值
-addclasspath	classpath (相关, 但具有不同的功能)	附加的类路径入口	空(没有其他路径入口)
-appRoot	n/a	应用程序根目录, 用于页面中的静态包含语句(include)所指定的应用程序相对路径	当前目录
-debug	emit_debuginfo	布尔型; 设置为 true 可以控制 ojspc 对原始的.jsp 文件生成行号映射, 用来调试	false
-d	page_repository_root	指定用来存放生成的二进制文件(.class 文件和资源文件)的目录	当前目录
-extend	n/a	指定一个 Java 类, 页面实现类要从这个指定的类上继承	空
-extres	external_resource	布尔型; 设置为 true 将控制 ojspc 对.jsp 文件中的静态内容生成一个外部的资源文件	false
-hotload (仅适用于 OSE)	n/a	布尔型; 设置为 true 将使得 ojspc 在页面实现类中添加必要的代码, 从而允许热加载功能	false
-implement	n/a	指定一个 Java 接口, 页面实现类要实现这个指定的接口	空
-noCompile	javacmd	布尔型; 设置为 true 将控制 ojspc 对生成的页面实现类不进行编译	false
-packageName	n/a	生成的页面实现类的包名	空(对每个.jsp 文件的位置生成一个包)

(续表)

选项	相关的 OracleJSP 配置参数	描述	缺省值
-S-<sqlj 选项>	Sqljcmd	-S 前缀后面跟一个 Oracle SQLJ 选项 (对.sqljsp 文件适用), 用来向 SQLJ 传递 选项参数	当前目录
-srcdir	page_repository_root	指定用来存放生成的源文件 (.java 文件 和.sqlj 文件) 的目录	空
-verbose	n/a	布尔型; 设置为 true 用来控制 ojspc 在运 行时输出详细的状态信息	False
-version	n/a	布尔型; 设置为 true 用来控制 ojspc 显示 OracleJSP 的版本号	false

ojspc 的命令行语法

通用的 ojspc 命令行语法如下所示: (其中%是 UNIX 的命令提示符)

```
% ojspc [option_settings] file_list
```

filelist 代表着一个文件列表, 它可以包含一个.jsp 文件或者一个.sqljsp 文件。

在这个命令行语法中, 有以下几点需要注意:

- 如果有多个.jsp 文件需要解释, 它们必须都具有相同的字符集 (要么使用缺省的字符集, 要么使用 page 指令的 contentType 属性来设置)。
- 文件列表中使用空格来分隔文件名。
- 在选项名和选项的取值之间用空格来分隔。
- 选项名是不区分大小写的, 但是选项值 (诸如包名、目录路径、类名以及接口名字等) 一般是要区分大小写的。
- 要允许那些缺省值是 false 的布尔型选项, 只需输入选项的名字即可, 例如要允许热加载这一选项, 你必须输入 ojspc -hotload, 而不能是 ojspc -hotload true。

下面是一个使用 ojspc 命令行语法的例子:

```
%ojspc -d /myapp/mybindir/ -srcdir /myapp/mysrcdir -hotload MyPage.sqljsp  
MyPage2.jsp
```

ojspc 的选项说明

本小节提供了更详细的关于 ojspc 选项参数的描述, 如下所示:

-addclasspath (全路径; ojspc 缺省值: 空)

使用此选项用来指定一个附加的类路径, javac 在编译生成的页面实现类源文件 (.java 文件) 时可以使用这个指定的类路径。如果不使用此选项, 那么 Javac 只使用系统的类路径:

(SQLJ 解释器在解释 SQLJ JSP 页面时也要使用-addclasspath 的设置)

注意: 在一个按需解释的应用场合下, OracleJSP 的 classpath 设置参数提供了与 -addclasspath 相类似 (但是有差异) 的功能。

-appRoot（全路径；ojspc 缺省值：当前目录）

使用此选项用来指定一个应用程序根目录，缺省值是 ojspc 正在运行的当前目录。

指定的应用程序根目录路径主要在以下几种情况中使用：

- 它被用在待解释的 JSP 页面中的静态包含（include）语句中。在 include 语句中的任何应用程序相对路径（或者环境相对路径）都要预先考虑此选项所指定的应用程序根目录。
- 它被用来决定页面实现类的包的命名。包的命名是基于正在被解释的文件相对于应用程序根目录的位置，并且包又决定了输出文件的存放位置。

如果你从其他的目录运行 ojspc 并且仍然想要正确定位 include 语句中所包含的文件，那么本选项是必须指定的。

考虑下面的例子：

- 你想要解释下面的文件：
/abc/def/ghi/test.jsp
- 你可以从当前的目录/abc 中运行 ojspc，命令如下所示（其中%是 UNIX 的命令行提示符）：
% cd /abc
% ojspc def/ghi/test.jsp
- 如果 test.jsp 页面中有如下的 include 指令语句：
<%@ include file="/test2.jsp" %>
- 并且 test2.jsp 页面是在/abc 目录下，如下所示：
/abc/test2.jsp

在这种情况下不需要-appRoot 设置，因为缺省的应用程序根目录就是当前目录，对于本例来说，就是/abc 目录。页面中的 include 指令语句使用了应用程序相对路径/test2.jsp（注意到是以“/”开始的），因此被包含的页面将正确无误地在/abc/test2.jsp 中被找到。

在本例中包的名字为 def.ghi，因为 test.jsp 的路径是/abc/def/ghi/test.jsp，而 ojspc 所运行的当前目录是/abc，因此应用程序根目录也是/abc。由于包的命名是基于 test.jsp 文件相对于应用程序根目录的位置，因此可以推断出包应该为 def.ghi。输出文件也根据包的名字被存放在相应的位置。

继续上面的例子，如果在其他的目录下运行 ojspc，这里假定为/home/mydir，那么你必须正确地设置-appRoot 选项，如下所示：

```
% cd /home/mydir  
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

在此例中，包的名字仍然是 def.ghi，因为指定的应用程序根目录仍然是/abc，所有 test.jsp 相对于应用程序根目录的路径仍然没有改变。

注意：一般情况下，最后的指定应用程序根目录正好是被解释的 JSP 页面的某一层次的父目录，这样处理和理解起来更容易。

-d(全路径；ojspc 缺省值：当前目录)

使用此选项来指定一个目录以保存生成的二进制文件——.class 文件和 Java 资源文件。

（由静态文本内容通过-extres 和-hotload 选项所生成的.res 文件是 Java 资源文件，而由 SQLJ

解释器通过处理 SQLJ JSP 页面所生成的.ser 文件也是 Java 资源文件。)

此选项所指定的路径只是一个文件系统路径（不是应用程序相对路径，也不是页面相对路径）。

在诸如 Windows NT 这样的环境中，允许在目录名中出现空格，这时要用引号来把指定的目录名引起来。

在指定的目录下的子目录会在合适的时候被自动创建，它取决于包的命名。要获得更多的信息，请参看“ojspc 的输出文件、存放位置以及相关选项”。

此选项的缺省值是使用当前目录（即你调用 ojspc 的那个目录）。

建议你最好使用这个选项来把生成的二进制文件存放在一个干净的目录下，这样你就能很容易地指定哪些文件被生成了。

注意：在一个按需解释的应用场合下，OracleJSP 的 page_repository_root 配置参数提供了相似的功能，请参看“OracleJSP 在非 OSE 环境下的配置参数”以获得相关信息。

-debug（布尔型；ojspc 缺省值：false）

如果将此标记设为 true，那么 ojspc 将对原始的.jsp 文件生成行映射信息，这些信息可以用来调试。如果使用缺省设置（此标记为 false），那么只有页面实现类才有对应的映射信息。

这个标记在诸如 Oracle JDeveloper 这样的纯开发环境下相当有用，它可以允许源代码级的 JSP 调试。

注意：在一个按需解释的应用场合下，OracleJSP 的 emit_debuginfo 配置参数提供了相同的功能，请参看“OracleJSP 在非 OSE 环境下的配置参数”以获得相关信息。

-extend（Java 类名；ojspc 缺省值：空）

使用此选项可以指定一个 Java 类，页面实现要从这个指定的类上继承。

-extres（布尔型；ojspc 缺省值：false）

如果将此标记设为 true，那么 ojspc 将会把生成的静态内容（如 Java 的 print 命令所输出的静态 HTML 代码）放置在一个 Java 资源文件中。而如果使用缺省设置（此标记为 false），那么 ojspc 将会把生成的静态内容放到页面实现类的一个内部类中。

资源文件的名称是基于 JSP 页面的名称的。对 OracleJSP 8.1.7 发行版来说，资源文件的名称与 JSP 页面的名称相同，只不过后缀名是.res（例如，要解释 MyPage.jsp 这个页面，那么除了标准的输出文件以外，还会生成一个资源文件 MyPage.res）。不过，这些具体的实现细节在以后的发行版本中将可能改变。

资源文件与.class 文件存放在同一个目录中。

如果在一个页面中有大量的具体内容，那么使用资源文件这一技术将提高解释速度，并且可以提高页面的执行速度。

注意：

- 内部类文件仍然被创建并且必须被发布。
- 在一个按需解释的应用场合下，OracleJSP 的 external_resource 配置参数提

供了相同的功能。请参看“OracleJSP 在非 OSE 环境下的配置参数”以获得相关信息。

-hotload (布尔型; ojspc 缺省值: false) (只适用于 OSE)

如果将此标记设为 true, 那么将允许热加载功能。这仅仅适用于在 Oracle Servlet Engine 环境下将解释后的页面加载到 Oracle8i 数据库中的过程。

允许-hotload 选项将导致 ojspc 做以下工作:

- 1、 执行-extres 选项的功能, 将静态内容写到一个 Java 资源文件中 (请参看上面所描述的-extres 选项)。
- 2、 在生成的页面实现类中创建一个 main() 方法以及一个 hotloading 方法, 从而允许热加载功能。

要获得对热加载功能的概述, 请参看“Oracle8i 中的热加载功能概述”, 要获得怎样完成热加载的步骤 (一旦热加载功能被允许), 请参看“在 Oracle8i 中热加载页面实现类”。

注意: 如果想在不允许热加载功能的前提下将静态内容写到一个资源文件中, 那么请用-extres 选项。

-implement (Java 接口名; ojspc 缺省值: 空)

使用此选项用来指定一个接口, 页面实现类将实现此接口。

-noCompile (布尔型; ojspc 缺省值: false)

如果将此标记设为 true, 那么 ojspc 将不编译生成的页面实现类的源文件 (.java 文件), 从而允许你以后使用别的 Java 编译器来编译它。

注意:

- 在一个按需解释的应用场合下, OracleJSP 的 javaccmd 配置参数提供了相似的功能, 允许你直接指定一个可替换的 Java 编译器。请参看“OracleJSP 在非 OSE 环境下的配置参数”以获得相关信息。
- 对一个 SQLJ JSP 页面来说, 允许-noCompile 选项仅仅阻止 Java 的编译过程, 并不阻止 SQLJ 的解释过程。

-packageName (全路径包名; ojspc 缺省值: 每个.jsp 文件的位置)

使用此选项用来指定生成的页面实现类所对应的包名, 包名使用 Java 的 “.” 语法作为分隔。

如果不设置此选项, 那么包的名字是根据.jsp 文件相对于当前的目录 (也就是运行 ojspc 的目录) 的位置来决定的。

考虑下面一个例子, 假设你从/myapproot 目录下运行了 ojspc, 而要解释的.jsp 文件处在/myapproot/src/jspsrc 目录下, 如下所示: (假设%是 UNIX 的命令提示符)

```
% cd /myapproot
```

```
% ojspc -packageName myroot.mypackage src/jspsrc/Foo.jsp
```

上面的命令导致页面实现类的包被命名为 myroot.mypackage。

如果这个例子中不使用-packageName 选项, OracleJSP 8.1.7(1.1.0.0.0)发行版将缺省地使用 src.jspsrc 作为包的名字。因为 Foo.jsp 相对于当前目录(/myapproot)的路径是 src/jspsrc。

(请意识到这种实现细节在以后的发行版本中可能会改变。)

-s-<sqlj option><value>(-S 和后面的 SQLJ 选项设置; ojspc 缺省值: 空)

对 SQLJ JSP 页面来说,使用 ojspc -S 选项可以将 Oracle SQLJ 的选项参数传递给 SQLJ 解释器。

不过与你直接运行 SQLJ 解释器所不同的是,在这里 SQLJ 的选项参数与它的值之间要用空格来分隔(这是为了与其他的 ojspc 选项保持一致)。

下面是一个例子(其中%是 UNIX 的命令提示符):

```
% ojspc -s-default-customizes mypkg.MyCust -d /myapproot/mybindir
MyPage.jsp
```

这个例子中调用 Oracle SQLJ 的-default-customizer 选项去选择一个其他的可更改的 profile——customizer, 并且设置了 ojspc -d 选项参数。

对应特定的 SQLJ 选项来说, 请注意以下几点:

- 不要使用 SQLJ -encoding 选项; 不过, 你可以使用 JSP 页面中的 page 指令语句的 contentType 参数来代替它。
- 如果你使用了 ojspc -addclasspath 选项, 那么不要使用 SQLJ -classpath 选项。
- 如果你使用了 ojspc -noCompile 选项, 那么不要使用 SQLJ -compile 选项。
- 如果你使用了 ojspc -srcdir 选项, 那么不要使用 SQLJ -dir 选项。

-srcdir (全路径; ojspc 缺省值: 当前目录)

使用此选项可以指定一个目录路径, ojspc 使用此路径来保存生成的源文件——.sqlj 文件(对 SQLJ JSP 页面来说)和 .java 文件。

此选项所指定的路径只是一个文件系统路径(不是应用程序相对路径, 也不是页面相对路径)。

在诸如 Windows NT 这样的环境中, 允许在目录名中出现空格, 这时要用引号来把指定的目录名引起来。

在指定的目录下的子目录会在合适的时候被自动创建, 它取决于包的命名。要获得更多的信息, 请参看“ojspc 的输出文件、存放位置以及相关选项”。

此选项的缺省值是使用当前目录(即你调用 ojspc 的那个目录)。

建议你最好使用这个选项来把生成的源文件存放到一个干净的目录下, 这样你就能很容易地知道哪些文件被生成了。

注意: 在一个按需解释的应用场合下, OracleJSP 的 page_repository_root 配置参数提供了相似的功能, 请参看“OracleJSP 在非 OSE 环境下的配置参数”以获得相关信息。

-verbose (布尔型; ojspc 缺省值: false)

如果将此标记设为 true, 那么 ojspc 在执行时就会报告它的解释步骤并输出。

下面的例子演示了使用 -verbose 选项后解释 myerror.jsp 文件时的输出内容(在本例中, ojspc 在 myerror.jsp 文件所在的目录中被运行, 也就是说当前目录与 myerror.jsp 所在的目录相同; 并且假设下面的%是 UNIX 的命令提示符)。

```
% ojspc -verbose myerror.jsp
Translating file: myerror.jsp
```

```
1 JSP files translated successfully.  
Compiling Java file: ./myerror.java
```

-version（布尔型；ojspc 缺省值：false）

如果将此标记设为 true，那么 ojspc 在运行时将显示 OracleJSP 的版本号，并且显示后马上退出。

ojspc 的输出文件、存放位置以及相关选项小结

缺省情况下，ojspc 所生成的文件与按需解释应用场合下由 OracleJSP 解释器所生成的文件是相同的，并且 ojspc 将生成的文件存放在当前目录（调用 ojspc 的那个目录）下。

下面是 ojspc 所生成的文件：

- 一个.sqlj 源文件（仅仅对 SQLJ JSP 页面适用）。
- 一个.java 源文件。
- 一个.class 文件，代表页面生成类。
- 一个.class 文件，代表页面生成类的内部类。
- 一个 Java 资源文件，或者一个.class 文件（任选的），代表 SQLJ 的 profile（仅仅对 SQLJ JSP 页面适用）。
- 一个 Java 资源文件，代表页面中的静态文本内容（任选的）。

要获得关于 OracleJSP 解释器所生成的文件及相关信息，请参考“生成的文件和位置（按需解释）”。

下面总结了在“ojspc 的选项说明”中所描述的比较通用的 ojspc 选项，这些选项都能影响到文件的生成和存放位置：

- -appRoot，用来指定一个应用程序根目录。
- -srcdir，用来指定一个存放源文件的目录。
- -d，用来指定一个存放二进制文件（.class 文件和 Java 资源文件）的目录。
- -noCompile，用来指定 ojspc 不要编译生成的页面实现类源文件（因此，此选项导致了没有任何.class 文件被生成）。
- extres，用来指定 ojspc 将静态文本放到一个 Java 资源文件中。
- -hotload，用来指定 ojspc 将静态文本放到一个 Java 资源文件中，并且允许热加载功能（此选项仅仅适用于目标平台为 Oracle Servlet Engine 的 JSP 应用程序）。
- -S-ser2class（实际上就是 SQLJ-ser2class 选项，仅仅适用于 SQLJ JSP 页面），用来指定 ojspc 生成一个.class 文件作为 SQLJ 的 profile，而不是缺省地用一个.ser Java 资源文件作为 profile。

对输出文件的存放位置来说，它们一般被存放在当前目录（或者-d、-srcdir 等参数指定的路径）下，并且有相应的目录结构。这些目录结构的组织要取决于包的命名，而包的命名又是由当前要被解释的.jsp 文件相对于应用程序根目录的路径来决定的，其中应用程序根目录要么是当前目录，要么是由-appRoot 选项所指定的目录。

下面提供了一个例子，假设你用以下的命令来允许 ojspc（其中%是 UNIX 的命令提示符）：

```
% cd /abc  
% ojspc def/ghi/test.jsp
```

从这个例子可以判断出包的名字应该是 `def.ghi`，并且 `ojspc` 所生成的文件应该被存放在目录 `/abc/def/ghi` 中。

如果你用 `-d` 和 `-srcdir` 选项指定了输出文件的存放位置，那么在你指定的那个目录下，`def/ghi` 子目录结构将被创建以保存输出文件。

现在考虑 `ojspc` 从其他的目录中被运行，如下所示：

```
% cd /home/mydir
% ojspc -appRoot /abc /abc/def/ghi/test.jsp
```

此时包的名字仍然是 `def.ghi`，因为应用程序的根目录仍然是 `/abc`（用 `-appRoot` 指定），所以 `test.jsp` 相对于 `/abc` 的路径仍然是 `def/ghi`。不过输出文件将被存放到 `/home/mydir/def/ghi` 目录下，如果在 `ojspc` 中使用了 `-d` 或 `-srcdir` 选项，输出文件将被保存在指定的路径下的 `def/ghi` 子目录中。

注意：建议你对自己的 JSP 应用程序中的每个目录都运行一次 `ojspc`，这样可以使不同目录中的文件得到合适的不同的包名。

6.3.2 loadjava 工具概述

`loadjava` 命令行工具是由 Oracle8i 提供的，它用来从 Java 文件创建 Schema 对象并且将它们分别加载到数据库中。

一般而言（不是专门针对 JSP 应用程序），一个 Java 开发人员可以在客户端编译 Java 源文件，然后加载生成的 `.class` 文件，或者也可以直接加载 Java 源文件，并且让服务器端的编译器在 Oracle8i 数据库中自动地编译这些源文件。在第一种情况下，只有类 Schema 对象被创建，而在第二种情况下，源 Schema 对象和类 Schema 对象都被创建，并且不管在这两种情况中的哪一种情况下，开发人员都可以加载 Java 资源文件并且创建资源 Schema 对象。

`loadjava` 工具可以在命令行接受源文件、类文件、资源文件、JAR 文件以及 ZIP 文件等，不过，源文件和类文件不能同时被加载。一个 JAR 文件、ZIP 文件或者 `loadjava` 的命令行中都可以包含有源文件或者类文件，但是不能同时包含两种文件。（它们可以同时包含有源文件和资源文件，或者类文件和资源文件。）

命令行中的 JAR 文件或者 ZIP 文件将被解压并处理，它们中所包含的每一个文件都可以创建出一个或者多个 Schema 对象。

对 OracleJSP 来说，使用 `loadjava` 工具可以完成以下工作：

- 在客户端解释的应用场合下，假设你已经用 `ojspc` 将 JSP 页面进行解释，并且缺省情况下，`ojspc` 也将编译解释后的 Java 代码。这时可以使用 `loadjava` 工具来加载生成的 `.class` 文件和所有的资源文件（通过 `ojspc -hotload` 等选项参数可以生成资源文件），一般情况下，所有的这些文件都被捆绑到了一个 JAR 文件中。
作为一种可替换的方案，你也可以加载解释后的 `.java` 文件来替代加载编译后的 `.class` 文件，这时应该让服务器端的编译器编译加载后的 `.java` 文件。
- 在服务器端解释的应用场合下，使用 `loadjava` 工具直接加载没有经过解释的 `.jsp` 文件（一般它们被绑定在一个 JAR 文件中），并且将它们以资源 Schema 对象的形式

放入数据库中。(在这之后，它们将在服务器端被解释并发行，这是通过 shell 命令 **publish** 来实现的。)

下面列出了完整的 **loadjava** 命令行语法，其中花括号{...}并不是语法的一部分，它们只是用来将输入选项参数所允许的两种可能的格式括起来：

```
loadjava {-user | -u} user/password[@ database] [ options]
file.java | file.class | file.jar | file.zip | file.sqlj | resourcefile
[-debug]
[-d | -definer]
[{-e | -encoding} encoding_scheme]
[-f | -force]
[{-g | -grant} user [, user]...]
[-o | -oci8]
[ -order ]
[-noverify]
[-r | -resolve]
[{-R | -resolver} " resolver_spec"]
[{-S | -schema} schema]
[ -stdout ]
[-s | -synonym]
[-t | -thin]
[-v | -verbose]
```

在上面所列的选项参数中，比较重要的有 **-user** 和 **-resolve** 两个选项（它们可以分别缩写为 **-u** 和 **-r**）。使用 **-user** 选项可以用来指定 Schema 对象的名称和密码，而使用 **-resolve** 选项用来指定：当 **loadjava** 命令行里所有的类都被加载后，**loadjava** 是否编译并解析你所加载的类里所使用的外部引用。

如果你正在加载一个 **.java** 源文件并且希望它在加载的过程中由服务器端的编译器自动编译，那么你必须允许 **-reslove** 选项。

下面举出了一个例子，它是用在客户端解释的场合，这时 JSP 页面已经被解释并编译过了，并且生成了两个 **.class** 文件——**HelloWorld.class** 文件对应着页面实现类，还有另外一个 **.class** 文件对应着页面实现类的内部类（它的名字以 “**HelloWorld**” 起始，所以 **HelloWorld*.class** 就可以包括这两个文件了）。在下面的例子中，**%** 是 UNIX 的命令提示符：

```
% loadjava -u scott/tiger -r HelloWorld*.class
```

或者你也可以将这两个文件打包成一个 JAR 文件，使用下面的命令：

```
% loadjava -v -u scott/tiger -r HelloWorld.jar
```

其中，**loadjava -v** (**-verbose**) 选项提供了加载过程中详细的状态报告信息，它在加载很多个文件或者在服务器端编译时特别有用。

下面是另外一个例子，它也是用在客户端解释的场合（**HelloWorld.java** 是 JSP 解释器的输出文件），不过你在客户端已经略过了编译过程（使用 **-noCompile** 选项），并且希望服务器端的编译器来完成编译过程：

```
% loadjava -v -u scott/tiger -r HelloWorld.java
```

下面提供了一个应用于服务器端解释场合的例子：

```
% loadjava -u scott/tiger -r HelloWorld.jsp
```

6.3.3 Session Shell 工具概述——sess_sh

sess_sh 工具是由 Oracle8i 所提供的，它作为一个交互式接口可以用来访问数据库实例的会话密码空间。当你运行 sess_sh 工具时你必须指定一个数据库连接作为输入参数，然后 sess_sh 就会显示出 \$ 命令提示符用来指示它已经准备好接收命令了。

你可以从 \$ 提示符下运行很多的顶层命令，并且每个命令都有一套自己的选项参数。不过对 OracleJSP 开发人员来说，通常最感兴趣的只有以下几个命令：publishservlet 和 unpublishservlet 命令（用在客户端解释的发布场合下）、publishjsp 和 unpublishjsp 命令（用在服务器端解释的发布场合下）以及 createcontext 命令（用来创建 OSE 的 servlet 环境）。

下面列出了运行 sess_sh 工具所必需知道的几个关键的语法元素：

```
sess_sh -user user -password password -service serviceURL
```

- -user 指定 Schema 对象的用户名。
- -password 指定这个用户的密码。
- -service 指定了一个数据库的 URL，而这个数据库的会话命名空间要被 sess_sh 所打开。其中 serviceURL 参数必须具有以下三种形式中的一种：

```
sess_iiop://host:port:sid
```

```
jdbc:oracle:type:spec
```

```
http://host[:port]
```

下面提供了一些通用的例子：

```
sess_iiop://localhost:2481:orcl
```

```
jdbc:oracle:thin:@myhost:1521:orcl
```

```
http://localhost:8000
```

下面是一个完整的使用 sess_sh 命令行的例子：

```
% sess_sh -user SCOTT -password TIGER -service
```

```
jdbc:oracle:thin@myhost:5521:orcl
```

在启动了 sess_sh 工具以后，你将会看到以下的命令提示符：

```
$
```

除了诸如 publishservlet 和 publishjsp 等用来发行对象的命令，session shell 工具也提供了一些通用的 shell 命令，使得你对会话命名空间的操作感觉起来就像在一个 UNIX 的 shell（如 C shell）下操作 UNIX 文件系统一样。例如，下面的 sess_sh 命令显示了在 /alpha/bega/gamma 发行环境（publishing context）下所有的已经发行的对象和发行环境（这里的发行环境指的是会话命名空间中的一个节点，类似于文件系统中的目录）：

```
$ ls /alpha/bega/gamma
```

前面已经提到过，对 OracleJSP 开发人员来说，关键的 sess_sh 命令包括以下五个：

```
$ publishjsp...
```

```
$ unpublish...
```

```
$ publishservlet...
```

```
$ un publishservlet...
```

```
$ createcontext...
```

要获得关于 publishservlet 和 unpublishservlet 目录的相关信息，请参看“在 Oracle8i 中发行解释后的 JSP 页面（publishservlet）”。要获得关于 publish.jsp 和 unpublishjsp 命令的相关信息，请参看“在 Oracle8i 中解释并发行 JSP 页面（publishjsp）”。

每个 session shell 命令都有一个 -describe 选项来描述它自己的操作，一个 -help 选项来

显示它的语法信息，以及一个-version 选项来显示它自己的版本号。

注意：本部分内容只是提供了对 sess_sh 语法和选项的简单讨论，只涉及了这个工具的最简单的调用和使用方法。

sess_sh 工具还有一些比较高级的用法，例如在 sess_sh 命令行上可以直接用引号将命令引起来并执行，而无需在\$提示符下执行这些命令，不过这些内容已经超出了本书的范围，这里就不作讨论了。

另外，还有一些顶层的选项可以用来指定连接到 plain IIOP，从而代替连接到缺省的会话 IIOP；可以用来指定一个角色；可以使用服务器端的 SSL 认证来连接到数据库；并且可以使用一个 service 的名字来代替数据库 URL 中的 SID。

6.4 发布到 Oracle8i——服务器端解释

本节讨论在服务器端解释的场合下发布到 Oracle8i 所需要的步骤。

这些所需的步骤如下所示：

- 1、使用 loadjava 工具将没有经过解释的 JSP 页面和 SQLJ JSP 页面源文件加载到 Oracle8i 中。
- 2、使用 publishjsp 命令解释并发行这些页面。

在第 2 步中，publishjsp 命令将自动地处理解释、编译、热加载（如果允许的话）以及发行等过程。

6.4.1 加载没有解释的 JSP 页面（loadjava）

作为在服务器端解释的发布过程的第一步，使用 Oracle 的 loadjava 工具将没有经过解释的.jsp 或者.sqljsp 文件以 Java 资源文件的形式加载进 Oracle8i 数据库中。

如果你希望一次加载多个文件，那么建议你最好将这些文件都放在一个单一的 JAR 文件中，这样有利于加载过程。

loadjava 工具是由 Oracle8i 所提供的，它作为一个通用工具主要用途是将 Java 文件加载到服务器的数据库中。要获得关于 loadjava 工具的概述信息，请参看“loadjava 工具概述”；要获得更进一步的信息，请参看“Oracle8i Java 工具参考”。

下面是一个例子，演示了如何使用 loading 工具来加载一个没有解释过的 JSP 页面：

```
% loadjava -u scott/tiger Foo.jsp
```

这条命令还将 Foo.jsp 加载到名字为 SCOTT 的 Schema 对象（密码为 TIGER）中并且作为一个 Java 资源对象来存放，这里无需指定 loadjava -resolve (-r) 选项。

并且这条命令还将导致在数据库里以下的资源 Schema 对象被创建：

- SCOTT:Foo.jsp

请注意，不管是在 JAR 文件中，或者是在 loadjava 命令行中，它所指定的.jsp 文件的路径信息决定了资源 Schema 对象的存放位置。考虑下面一个例子，它对上一个例子作了修改，在命令行中指定了路径信息：

```
% loadjava -u scott/tiger xxx/yyy/Foo.jsp
```

这个例子将导致下面的资源 Schema 对象被创建到数据库中；

- SCOTT: xxx/yyy/Foo.jsp

要获得关于 loadjava 是如何命名它所生成的 Schema 对象的相关信息，请参看“数据库的 Java Schema 对象”。

你也可以加载一个.sqllisp 文件，如下例所示：

```
% loadjava -u scott/tiger Foo.sqllisp
```

这条命令将 Foo.sqllisp 文件加载到名字叫做 SCOTT 的 Schema 对象中，并且将导致在数据库中创建以下的资源 Schema 对象：

- SCOTT: Foo.sqllisp

如果想加载多个.jsp（或者.sqllisp）文件，可以使用通配符（具体的通配符字符取决于特定的操作系统）。下面举出一个例子，其中%是 UNIX 的命令提示符：

```
% loadjava -u scott/tiger *.jsp
```

或者假如你已经将所有的.jsp 文件放到了一个 JAR 文件中，那么使用以下命令：

```
% loadjava -u scott/tiger myjspapp.jar
```

6.4.2 在 Oracle8i 中解释并发行 JSP 页面

如果你使用了 JServer 所提供的 session shell 命令 publishjsp，那么解释、编译、热加载（如果被允许）以及发行等步骤都可以自动被处理，publishjsp 只能应用于服务器端解释的发布过程。

publishjsp 要在你将.jsp（或者.sqllisp）文件作为一个资源 Schema 对象加载到 Oracle8i 数据库之前运行。（本节单独讨论了对.sqllisp 文件运行 publishjsp 的内容，因为它与.jsp 文件所得到的结果在逻辑上有差异。）

注意：使用 publishjsp 命令所发行的 JSP 页面可以被“取消发行”（即从 JServer 的 JNDI 命名空间中删除），这时通过 session shell 命令 unpublishjsp 来实现的。要获得关于 unpublishjsp 命令的相关信息，请参看“使用 unpublishjsp 取消对 JSP 页面的发行”。

publishjsp 的语法和选项概述

启动 sess_sh 工具可以建立一个数据库连接。一旦 sess_sh 被启动，你就可以在 session shell 的命令提示符\$下运行 publishjsp 命令了。

publishjsp 命令的通用语法如下所示：

```
$ publishjsp [options] path/name.jsp
```

其中，options 可以是下面所列的形式之一：

```
[-schema schemaname] [-virtualpath path] [-servletName name] [-packageName name]
[-context context] [-hotload] [-stateless] [-verbose] [-resolver resolver]
[-extend class] [-implement interface]
```

注意：

- 要想允许某一个布尔型的选项，如-hotload，只需在命令行上录入选项的名字即可，无需将它设为 true。
- 对于那些需要指定一个值的选项来说，值不必用引号括起来。

在 `publishjsp` 的命令行语法中, `name.jsp` (对于一个 SQLJ JSP 页面来说是 `name.sqljsp`) 是唯一一个必需的参数, 它指定了你使用 `loadjava` 所加载的 JSP 页面资源 Schema 对象, 并且可以指定相对路径信息。

缺省情况下, 如果没有指定 `-virtualpath` 选项, 那么 `path/name.jsp` 就成为 `servlet` 路径。例如, 如果运行 `publishjsp dir1/foo.jsp` 这条命令, 它将导致 `dir1/foo.jsp` 成为 `servlet` 路径。

并且, 缺省情况下, 如果没有指定 `-context` 选项, 那么将使用 OSE 缺省的 `servlet` 环境, “/” 将成为环境路径。

环境路径和 `servlet` 路径两个合在一起(包括主机名和端口号)决定了调用页面的 URL。

下面的信息型选项对 `publishjsp` 也是可用的:

- 使用 `-showVersion` 可以显示 OracleJSP 的版本号并立刻退出。
- 使用 `-useage` 可以显示 `publishjsp` 的选项列表并立刻退出。

下面提供了对每个选项功能的描述:

- `-schema schemaname`

如果你通过 `sess_sh` 所登录进来的 Schema 对象不包含你想要发行的 JSP 页面所对应的资源 Schema 对象, 那么使用此选项可以指定 JSP 页面资源 Schema 对象的位置。

所指定的这个 Schema 对象必须在你登录进来的 Schema 对象里那个访问得到, `publishjsp` 命令并不提供连接其他 Schema (指定密码) 的手段。

- `-virtualpath path`

你可以通过此选项对 JSP 页面指定一个其他的 `servlet` 路径; 如果不指定此选项的话, `servlet` 路径就只是简单地由指定的 `.jsp` 文件名和任何指定的 Schema 路径所组合成。

例如, 下面是一个使用 `-virtualpath` 的例子:

```
-virtualpath altpath/Foo.jsp
```

- `-servletName name`

你可以使用此选项对 JSP 页面来指定一个其他的 `servlet` 名字(在 OSE `name_servlets` 中); 不过, 这个 `servlet` 名字与其他页面是如何被调用的并没有太大的关系, 因此这个选项是很少需要被指定的。

缺省情况下, `servlet` 名字是由 `.jsp` 文件的基名 (除去扩展名后的文件名) 与任何你所指定的路径所组合而成的。例如, 在 OracleJSP 8.1.7(1.1.0.0.0)发行版中如果运行命令 `publishjsp SCOTT:dir1/Foo.jsp`, 那么 `servlet` 名字将会是 `dir1.Foo`。(请注意, 这样的实现细节在未来的发行版本中可能被改变。)

- `-packageName name`

你可以使用此选项对生成的页面实现类指定一个包名; 如果不指定此选项的话, 包名由你在运行 `publishjsp` 时命令行上所指定的 `.jsp` 文件的路径信息来决定。例如, 如果运行命令 `publishjsp SCOTT:dir1/Foo.jsp`, 那么页面实现类的包名就将是 `dir1`。`-packageName` 选项影响到 Schema 对象在 Schema 中的存放位置, 但是不影响 JSP 页面的 `servlet` 路径。

- `-context context`

你可以使用此选项在 Oracle Servlet Engine 环境下指定一个 servlet 环境路径。这个指定的环境路径将成为 URL 的一部分去调用 JSP 页面。

如果你没有指定此选项，那么 JSP 页面将位于 OSE 的缺省环境 /webdomains/contexts/default 中，它的环境路径就是 “/”。

在此选项中任意一个被指定的环境都将在 /webdomains/contexts 下，如：
/webdomains/contexts/mycontext。

注意：在访问 JSP 页面的 URL 中所使用的是 servlet 环境的环境路径，而不是环境名字自身。

当你使用 session shell 命令 createcontext 在 OSE 环境下创建一个 servlet 环境时，环境路径（通过 createcontext -virtualpath 选项指定）和环境名字都必须给出。一般情况下，将环境名字和环境路径指定为相同的，这仅仅是一种比较方便的方法，但不是必需的。

- **-hotload**

将此选项设置为 true 将允许并执行热加载功能，这将导致 publishjsp 命令执行以下的步骤：

- 1、把静态内容写到一个资源 Schema 对象中，而缺省情况下是把静态内容写到页面实现类的 Schema 对象中。
- 2、在生成的页面实现类中实现一个 main() 和 hotloading 方法，从而允许热加载功能。
- 3、调用 main() 方法以执行热加载。

要使用 -hotload 选项，你必须被许可使用 JServer 的热加载器。这可通过以下命令来实现：

```
dbms_java.grant_permission('SCOTT', 'SYS:oracle.aurora.security.JServerPermission',  
'HotLoader', null);
```

要获得对热加载功能的概述信息，请参看 “Oracle8i 中的热加载功能概述”。

- **-stateless**

这是一个布尔型的选项，它用来向 Oracle Servlet Engine 通报当前的 JSP 页面是无状态的——也就是说，这个 JSP 页面在运行过程中不能访问 HttpSession 对象。

此选项主要是用来对 mod_ose 进行优化的。要获得关于 Apache mod_ose 模块的相关信息，请参看 “Oracle8i Oracle Servlet Engine 用户指南”。

- **-verbose**

如果将此布尔型的选项设置为 true，publishjsp 在运行的时候就会报告先解释的步骤等信息。

- **-resolver**

使用此选项可以指定一个其他的 Java 类解析器。类解析器是用来编译和解析 Java 源代码的，并且还要定位 JSP 页面中所使用的类。

缺省的解析器是 ((*user) (*PUBLIC))。对名字叫 SCOTT 的 Schema 来说，解析器如下所示：

```
( (*SCOTT) (*PUBLIC) )
```

对-resolver 选项来说，你必须将此选项的值用引号引起来，如下例所示：

```
$ publishjsp ... -resolver "((*BILL) (*SCOTT) (*PUBLIC))"...
```

- -extend

使用此选项可以指定一个 Java 类，并且页面实现类将从这个指定的类继承。

- -implement

使用此选项可以指定一个 Java 接口，并且页面实现类必须要实现这个接口。

使用 publishjsp 发行 JSP 页面的实例

本小节提供了使用 publishjsp 命令解释并将.jsp 页面发行到 Oracle8i 数据库中的例子，其中这些.jsp 页面已经以资源 Schema 对象的形式被加载到数据库某个特定的 Schema 中，如 SCOTT:Foo.jsp。

要想获得关于 servlet 路径和环境路径是如何组合在一块以形成可以访问 JSP 页面的 URL 的相关信息，请参看“Oracle Servlet Engine 中的 URLs”。

注意：

- 下面的例子使用了名字叫 SCOTT 的 Schema, SCOTT 要么是在启动 sess_sh 时所指定的 Schema，要么是在这个指定的 Schema 中可以访问的 Schema，除此之外 SCOTT 均不可访问。
- 每个例子都列出了它所创建的 Schema 对象，尽管它们是次要的。要调用 JSP 页面所必需的 URL 只跟 servlet 路径和环境路径相关，页面实现类的 Schema 对象在 publishjsp 的发行步骤中被自动地映射。
- 在 Oracle8i 中的动态 jsp:include 和 jsp:forward 语句所使用的应用程序相对路径和页面相对路径语法与在其他任何 JSP 环境中所使用的语法都是相同的。相对路径是根据 JSP 页面的发行步骤来确定的（在下面的例子中将有说明）。
- 在未来的发行版本中，生成的 Schema 对象的精确的名字可能要被改变，但是它将仍然具有相同的通用形式。Schema 对象名字将总是包含文件的基名（在这些例子中是“Foo”），但是可能有一些轻微的改变，比如用_Foo 来代替 Foo 等。
- \$ 是 sess_sh 的命令提示符。

例一：

```
$ publishjsp -schema SCOTT dir1/Foo.jsp
```

这个例子使用缺省的 servlet 环境，它的环境路径就是“/”。

缺省的 servlet 路径是 dir1/Foo.jsp。

在这条命令行以后，Foo.jsp 可以通过如下的 URL 来访问：

```
http://host[:port]/dir1/Foo.jsp
```

如果要在应用程序中其他的 JSP 页面中动态地访问 Foo.jsp，这里假定在 dir1/Bar.jsp 中，那么可以使用以下的语句来完成（先使用页面相对路径，然后使用应用程序相对路径）：

```
<jsp:include page="Foo.jsp" flush="true" />
```

或者

```
<jsp:include page="/dir1/Foo.jsp" flush="true" />
```

缺省情况下，页面实现类和内部类所对应的 Java 包就是 `dir1`（根据对 SCOTT 所指定的路径来推断）。

此例子创建了下面的 Schema 对象：

- SCOTT:dir1/Foo 源 Schema 对象
- SCOTT:dir1/Foo 类 Schema 对象
- 在 `dir1` 目录下的一个类 Schema 对象，它对应着包含静态文本的内部类（它的名字中含有“Foo”，如 SCOTT:dir1/Foo\$__jsp__staticText）。

例二：

```
$ publishjsp -schema SCOTT -context /webdomains/contexts/mycontext Foo.jsp
```

其中，`mycontext` 是由以下命令所创建的：

```
$ createcontext -virtualpath mycontext /webdomains mycontext
```

在这个例子中，`publishjsp` 命令将页面发行到 `mycontext` servlet 环境中，并且在创建 `mycontext` 的过程中，已经指定了 `mycontext` 也就是环境路径。

缺省的 servlet 路径是 `Foo.jsp`。

在这条命令运行以后，`Foo.jsp` 可以通过如下的 URL 来访问：

```
http://host[:port]/mycontext/Foo.jsp
```

如果要在应用程序中其他的 JSP 页面中动态地访问 `Foo.jsp`，这里假定在 `Bar.jsp` 中，那么可以使用以下的语句来完成（先使用页面相对路径，然后使用应用程序相对路径）：

```
<jsp:include page="Foo.jsp" flush="true" />
```

或者

```
<jsp:include page="/Foo.jsp" flush="true" />
```

即便这个例子中指定了非缺省的 servlet 环境，但这与动态的 `jsp:include` 或 `jsp:forward` 语句并不相关。与它相关的只有页面的发行路径，在这个例子中它的页面相对路径很简单，就是 `/Foo.jsp`。

缺省情况下，这个例子不产生页面实现类和内部类（因为在 SCOTT Schema 并没有指定路径）。

此例子创建了下面的 Schema 对象：

- SCOTT:Foo 源 Schema 对象
- SCOTT:Foo 类 Schema 对象
- 一个 Schema 对象，它对应着包含静态文本的内部类（它的名字中含有“Foo”比如 SCOTT:Foo\$__jsp__StaticText）。

例三：

```
$ publishjsp -schema SCOTT -context /webdomains/contexts/mycontext  
dir1/Foo.jsp
```

其中，`mycontext` 是由以下命令所创建的：

```
$ createcontext -virtualpath mywebapp /webdomains mycontext
```

在这个例子中，`publishjsp` 命令将页面发行到 `mycontext` servlet 环境中，并且在创建 `mycontext` 的过程中已经指定了 `mywebapp` 来作为环境路径。

缺省的 servlet 环境是 `dir1Foo.jsp`。

在这条命令行运行以后，Foo.jsp 可以通过如下的 URL 来访问：

`http://host[:port]/mywebapp/dir1/Foo.jsp`

如果要在应用程序中其他的 JSP 页面中动态地访问 Foo.jsp，这里假定在 dir1/Bar.jsp 中，那么可以使用以下的语句来完成（先使用页面相对路径，然后使用应用程序相对路径）：

```
<jsp:include page="Foo.jsp" flush="true" />
```

或者

```
<jsp:include page="/dir1/Foo.jsp" flush="true" />
```

例一和例三使用不同的 servlet 环境，但是它们的动态 jsp:include 或者 jsp:forward 语句却是相同的，这说明 servlet 环境与它们无关，与 servlet 环境有关的是页面的发行路径，在这个例子中它的页面相对路径变成了/dir1/Foo.jsp。

缺省情况下，页面实现类和内部类所对应的 Java 包是 dir1。

此例子创建了以下的 Schema 对象：

- SCOTT:dir1/Foo 源 Schema 对象
- SCOTT:dir1/Foo 类 Schema 对象
- 一个 Schema 对象，它对应着包含静态文本的内部类（它的名字中含有“Foo”，比如 SCOTT:Foo\$__jsp__StaticText）。

例四：

```
$ publishjsp -schema SCOTT -hotload -packageName mypkg dir1/Foo.jsp
```

这个例子执行热加载功能、使用缺省的 servlet 环境并且覆盖了缺省的 dir1 包名。

环境路径是“/”。

-packageName 选项并不影响 servlet 路径，它仍然是缺省值 dir1/Foo.jsp。

在这条命令运行以后，Foo.jsp 可以通过如下的 URL 来访问：

`http://host[:port]/dir1/Foo.jsp`

如果要在应用程序中其他的 JSP 页面中动态地访问 Foo.jsp，这里假定在 dir1/Bar.jsp 中，那么可以使用以下的语句来完成（先使用页面相对路径，后使用应用程序相对路径）：

```
<jsp:include page="Foo.jsp" flush="true" />
```

或者

```
<jsp:include page="/dir1/Foo.jsp" flush="true" />
```

此例子创建了以下的 Schema 对象：

- SCOTT:mypkg/Foo 源 Schema 对象
- SCOTT:mypkg/Foo 类 Schema 对象
- 在 mypkg 目录下的一个类 Schema 对象，它对应着内部类（它的名字中包含有“Foo”，比如 SCOTT:mypkg/Foo\$__jsp__StaticText）。
- SCOTT:mypkg/Foo.res 资源 Schema 对象，包含着静态文本内容，而这些静态文本内容在缺省情况下是放在内部类中的（资源也可以被热加载，并且这是 publishjsp 功能的一部分）。

使用 publishjsp 发行 SQLJ JSP 页面

本小节提供了一个使用 publishjsp 命令解释并将.sqljsp 页面发行到 Oracle8i 数据库中的例子，其中这些.sqljsp 页面已经以资源 Schema 对象的形式被加载到数据库中某个特定的

Schema 中，如 SCOTT:Foo.sqljsp。

对于.sqljsp 页面应该意识到以下几点：

- 相对于发行一个.jsp 页面所创建的文件来说，发行一个.sqljsp 页面要多创建一个 Schema 对象——代表 SQLJ Profile 的资源 Schema 对象。这个对象总是一个.ser 资源 Schema 对象，而不是一个类 Schema 对象，因为在服务器端解释时并没有 SQLJ -ser2class 选项。
- 生成的源 Schema 对象对应着 SQLJ 源文件而不是 Java 源文件。
- SQLJ 在服务器端只有很有限的选项可以得到支持。

服务器端的 SQLJ 选项 客户端的 SQLJ 选项在服务器端的解释过程中是不可用的（这是一种通用的情况，而不是专门针对 JSP 页面的）。为了解决这一问题，通过标准的 Oracle8i JAVA\$OPTIONS 表提供了一小套可用的参数，这些参数可以用 dbms_java.set_compiler_option() 存储过程来设置（例如，使用 SQL*Plus）。在这些参数中只有下面的一些被 JSP 页面所支持：

- online

这是一个布尔型的选项，用来允许在线的语法检查。语法检查是通过缺省的 oracle.sqlj.checker.OracleChecker front_end 来实现的。

用 publishjsp 发行 SQLJ JSP 页面的例子 下面提供了一个例子，用来说明在发行一个.sqljsp 页面时 publishjsp 的用法。

注意：

- 下面的例子使用了名字叫 SCOTT 的 Schema，SCOTT 要么是在启动 sess_sh 时所指定的 Schema，要么是在这个指定的 Schema 中可以访问的 Schema，除此之外 SCOTT 均不可访问。
- 每个例子都列出了它所创建的 Schema 对象，尽管它们是次要的。要调用 JSP 页面所必需的 URL 只跟 servlet 路径和环境路径相关，页面实现类的 Schema 对象在 publishjsp 的发行步骤中被自动地映射。
- 在未来的发行版本中，生成的 Schema 对象的精确的名字可能要被改变，但是它将仍然具有相同的通用形式。Schema 对象名字将总是包含文件的基名（在这些例子中是“Foo”），但是可能有一些轻微的改变，比如用 _Foo 来代替 Foo 等。

```
$ publishjsp -schema SCOTT dir1/Foo.sqljsp
```

在这个例子中使用了缺省的 OSE servlet 环境，它的环境路径就是“/”。

缺省的 servlet 路径是 dir1/Foo.sqljsp。

在这条命令运行后，Foo.sqljsp 就可以通过如下的 URL 来访问：

```
http://host[:port]/dir1/Foo.sqljsp
```

如果要在应用程序中其他的 JSP 页面中动态地访问 Foo.sqljsp，这里假定在 dir1/Bar.jsp 中，那么可以使用以下的语句来完成（先使用页面相对路径，后使用应用程序相对路径）：

```
<jsp:include page="Foo.sqljsp" flush="true" />
```

或者

```
<jsp:include page="/dir1/Foo.sqljsp" flush="true" />
```


缺省情况下，页面实现类和内部类所对应的 Java 包就是 `dir1`（根据在 SCOTT 所指定的路径来推断）。

此例子创建了以下的 Schema 对象：

- SCOTT:dir1/Foo 源 Schema 对象
- SCOTT:dir 1/Foo 类 Schema 对象
- 在 `dir1` 目录下的一个资源 Schema 对象，它对应着 SQLJ 的 profile。（它的名字中含有“Foo”，比如 SCOTT:dir1/Foo_SJProfile0.ser。）

使用 `unpublishjsp` 取消对 JSP 页面的发行

`sess_sh` 工具还提供了一个工具 `unpublishjsp`，它可以用来从 JServer 的 JNDI 命名空间中删除一个 JSP 页面，不过这一过程并不删除页面实现类的 Schema 对象。

与 `unpublishservlet` 命令不同的是，你在 `unpublishjsp` 命令中不需要指定 `servlet` 名字（除非你在运行 `publishjsp` 时指定了一个 `servlet` 名字）。通常情况下，这个命令只需要将 `servlet` 路径作为输入即可（有时 `servlet` 路径指的是虚拟路径）。

下面是 `unpublishjsp` 命令的通用语法：

```
$ unpublishjsp [-servletName name] [-context context] [-showVersion] [-usage]
[-verbose] servletpath
```

其中，`-servletName`、`-context`、`-showVersion`、`-usage` 和 `-verbose` 选项的功能与 `publishjsp` 中对应选项的功能是相同的。

在使用 `unpublishjsp` 时，应该在 `-servletName` 和 `-context` 选项中指定与 `publishjsp` 所指定的值相同的值。

下面提供了一个例子，它用来取消发行前面一节所提供的例四中发行的页面：

```
$ unpublishjsp dir1/Foo.jsp
```

（请注意，在例四中 `-packageName` 选项指定的包名对于 `servlet` 路径是没有效果的。）

6.5 发布到 Oracle8i——客户端解释

本节讨论在客户端解释的场合下发布到 Oracle8i 所需要的步骤。

这些所需的步骤如下所示：

- 1、在客户端使用 `ojspc` 预解释 JSP 页面或 SQLJ JSP 页面。
- 2、使用 `loadjava` 将这些文件加载到 Oracle8i 中——.class 文件（任选地，.java 或者 .sqlj 文件）以及在页面解释过程中所生成的所有 Java 资源文件。
- 3、（可选地）将页面热加载到 Oracle8i 数据库中（如果在解释过程中热加载功能被允许的话）。要获得关于热加载功能的背景知识，请参看“Oracle8i 中的热加载功能概述”。
- 4、使用 `session shell` 命令 `publishservlet` 命令发行这些页面。

注意：为了简单和方便起见，一般建议使用服务器端解释的发布过程，请参看“发布到 Oracle8i——服务器端解释”。

6.5.1 预解释 JSP 页面 (ojspc)

为了在客户端预解释 JSP 页面(一般情况下是对那些运行在 Oracle Servlet Engine 环境下的页面), 可以使用 ojspc 命令行工具以调用 OracleJSP 解释器。

要获得关于 ojspc 的一般性介绍以及它的选项功能描述等信息, 请参看“预解释工具 ojspc”。

本小节的剩余部分将涵盖以下几部分内容:

- 最简单的 ojspc 用法
- 对 SQLJ JSP 页面使用 ojspc
- 允许 ojspc 的热加载功能
- 其他的关键 ojspc 特性与选项
- ojspc 例子

注意: 在未来的发行版本中, 生成文件的精确的名字可能要被改变, 但是它们仍然具有相同的通用形式。文件名将总是包含基名(如在这些例子中是“Foo”), 但是可能有一些轻微的改变, 比如会变成-Foo.java 或者-Foo.class 等。

最简单的 ojspc 用法

下面的例子演示了 ojspc 的最简单的用法:

```
% ojspc Foo.jsp
```

在这个最简单的例子中, 生成了下面几个文件:

- Foo.java
- Foo.class
- 一个.class 文件, 对应着包含有静态文本的内部类(它的名字含有“Foo”)

缺省情况下, 所有的输出文件都存放在当前目录(也就是运行 ojspc 的那个目录)下。

对 SQLJ JSP 页面使用 ojspc

ojspc 工具除了可以接受.jsp 文件作为输入参数外, 还可以接受.sqljsp 文件并处理含有 SQLJ 代码的 JSP 页面。下面是一个例子:

```
% ojspc Foo.sqljsp
```

对一个.sqljsp 文件来说, ojspc 将自动调用 SQLJ 解释器来处理它。

对于上面的例子来说, 将有以下几个文件被生成:

- Foo.sqlj (由 JSP 解释器从 Foo.sqljsp 文件所生成)
- Foo.java (由 JSP 解释器从 Foo.sqlj 文件所生成)
- Foo.class
- 一个.class 文件, 对应着包含有静态文本的内部类(它的名字含有“Foo”)
- 一个 Java 资源文件(.ser)或者类文件(.class), 对应着 SQLJ 的 profile(它的名字中含有“Foo”)。至于到底是.ser 文件还是.class 文件, 要取决于 SQLJ-ser2class 选项的设置。

缺省情况下, 所有的输出文件都存放在当前目录(也就是运行 ojspc 的那个目录)下。

允许 `ojspc` 的热加载功能

使用 `ojspc -hotload` 选项将允许热加载功能，此时静态页面放置到一个 Java 资源文件中，而不是缺省情况下放到页面实现类的内部类中。

下面的例子解释 `Foo.jsp` 页面并且控制 OracleJSP 解释器允许热加载功能：

```
% ojspc -hotload Foo.jsp
```

在这条命令运行以后，解释器将生成以下的输出文件：

- `Foo.java`（与缺省情况一样）。
- `Foo.class`（与缺省情况一样）。
- `Foo.res`，一个包含页面静态内容的 Java 资源文件。
- 一个 `.class` 文件，对应着页面实现类的内部类（与缺省情况一样，它的名字中包含“`Foo`”，不过此时页面静态内容被包含在 `Foo.res` 中，而不是在这个文件中）。

应该注意的是，`ojspc -hotload` 选项只是允许热加载功能，它并没有实际地热加载页面。

其他的关键 `ojspc` 特性与信息

下面所列的 `ojspc` 选项在发布过程中是特别有用的，要获得它们的详细描述信息，请参看“`ojspc` 的选项说明”。

- `-appRoot`——设置一个应用程序根目录用来代替缺省的应用程序根目录（当前目录，即 `ojspc` 正在运行的那个目录）。
- `-noCompile`——将这个标记设为 `true`，从而允许方在解释过程中不进行编译。使用此选项的一个应用场合就是将解释过的页面以 `.java` 文件的形式加载进数据库中，然后用服务器端的编译来执行编译过程。
- `-d`——指定一个目录用来存放生成的二进制文件（`.class` 文件和 Java 资源文件）。这一选项使得你可以更容易地指定在编译过程中生成了哪些文件，从而决定将哪些文件加载到 Oracle8i 中。
- `-srcdir`——指定一个目录用来存放生成的源文件。此选项的一个应用场合是和 `-noCompile` 联合使用，可以用来取代 `-d` 选项，并且将解释过的页面以 `.java` 源文件的形式加载进 Oracle8i 中。
- `-extres`——控制 OracleJSP 解释器将静态的页面内容放到一个 Java 资源文件中，以取代缺省情况下将它仍放到页面实现类的内部类中。
- `-hotload`——控制 OracleJSP 解释器将静态的页面内容放到一个 Java 资源文件中，以取代缺省情况下将它仍放到页面实现类的内部类中，并且在页面实现类中生成代码以允许热加载功能。设置传递给 Oracle SQLJ 解释器来处理。
- `-s`——对 SQLJ JSP 页面来说，使用 `-s` 前缀将指定 Oracle SQLJ 的选项参数；`ojspc` 把这些选项设置传递给 Oracle SQLJ 解释器来处理。

`ojspc` 例子

下面的一些例子演示了关键 `ojspc` 选项的用法。

例一：

```
% ojspc -appRoot /myroot/pagesrc -d /myroot/bin -hotload
```

/myroot/pagesrc/Foo.jsp

这个例子可以完成以下的功能：

- 指定了一个应用程序根目录，它影响到被解释的页面中静态 `include` 指令所指定的应用程序相对路径。
- 允许了热加载功能，并且对页面中的静态内容生成一个 Java 资源文件 `Foo.res`。
- 缺省地将 `Foo.java` 文件存放在当前的目录下。在这个例子中没有包，因为 `Foo.jsp` 处于指定的应用程序根目录下。
- 将 `Foo.class`、`Foo.res` 以及对应着内部类的那个 `.class` 文件都存放在 `/myroot/bin` 目录下。

例二：

```
% -appRoot /myroot/pagesrc -srcdir /myroot/gensrc -noCompile -extres  
/myroot/pagesrc/Foo.jsp
```

这个例子可以完成以下功能：

- 指定了一个应用程序根目录，它影响到被解释的页面中静态 `include` 指令所指定的应用程序相对路径。
- 对页面中的静态内容生成了一个 Java 资源文件 `Foo.res`（没有允许热加载）。
- 将 `Foo.java` 文件存放在 `/myroot/gensrc` 目录下。在这个例子中没有包，因为 `Foo.jsp` 处于指定的应用程序根目录下。
- 不编译 `Foo.java` 文件（因此没有 `.class` 文件生成）。
- 缺省地将 `Foo.res` 文件存放在当前的目录下。

例三：

```
% -appRoot /myroot/pagesrc -d /myroot/bin -extres -s -ser2class true/  
/myroot/pagesrc/Foo.jsp
```

这个例子可以完成以下功能：

- 指定了一个应用程序根目录，它影响到被解释的页面中静态 `include` 指令所指定的应用程序相对路径。
- 对页面中的静态内容生成了一个 Java 资源文件 `Foo.res`（没有允许热加载）。
- 缺省地将 `Foo.sqlj` 和 `Foo.res` 文件存放在当前的目录下。在这个例子中没有包，因为 `Foo.jsp` 处于指定的应用程序根目录之下。
- 将 `Foo.class`、`Foo.res`、对应着内部类的那个 `.class` 文件以及对应着 SQLJ profile 的 `.class` 文件都存放在 `/myroot/bin` 目录下。（如果没有指定 SQLJ `-ser2class` 选项的话，profile 将以 `.ser` Java 资源文件的形式被生成，而不是以 `.class` 文件的形式被生成。）

6.5.2 加载解释后的 JSP 页面（loadjava）

在客户端预解释步骤完成以后，可以使用 `loadjava` 关键将生成的文件加载到 Oracle8i 数据库中。

使用 `loadjava` 的应用程序场合有以下两种：

- 加载 `.class` 文件和 Java 资源文件（如果有的话）。
- 如果在前一步骤中使用了 `ojspc -noCompile` 选项，那么加载解释后的 `.java` 文件和

Java 资源文件（如果有的话）。.java 文件可以在加载的过程中由 Oracle8i 服务器端编译器来完成编译过程。

不管在这两种情况中的哪一种，如果你要加载多个文件，建议你最好是先将所有的文件放到一个 JAR 文件中，然后再加载这个 JAR 文件。

loadjava 是由 Oracle8i 提供用来作为通用目的的关键，它的主要用途是将 Java 文件加载进数据库中。

注意：在下面的两节中（“使用 loadjava 类文件”以及“使用 loadjava 加载源文件”），应该意识到下面一些重要的问题：

- 即使你通过允许-extres 或者-hotload 选项将页面静态内容放到一个资源文件中，但是页面实现类的内部类仍然被生成并且必须被加载。
- 和一个 Java 编译器相似，loadjava 也将解析对类的引用，但是不解析对资源的引用；为了确保正确地加载类文件所需的资源文件，你必须将这些资源文件和类文件所对应的.java 文件放置在一个包里。

使用 loadjava 加载类文件

假设你已经使用 ojspc -extres 或者 ojspc -hotload 对应一个 JSP 页面——Foo.jsp 进行了解释，并且生成了以下的文件：

- Foo.java
- Foo.class
- Foo\$_jsp_StaticText.class
- Foo.res

注意：这里所使用的生成文件的名字只是作为一个例子，在未来发行版本中这样的命名实现细节可能要被改变，但是文件的基名（如这里的“Foo”）将总是生成文件名字的一部分。

你可以忽略掉 Foo.java 文件，但是二进制文件(.class 文件和.res 文件)必须被加载到 Oracle8i 数据库中。一般情况下，你最好将 Foo.class、Foo\$_jsp_StaticText.class 以及 Foo.res 文件放到一个 JAR 文件中，这里假定为 Foo.jar，然后将此 JAR 文件集资到数据库中。如下所示（这里%是 UNIX 的命令提示符）：

```
%loadjava -v -u scott/tiger -r Foo.jar
```

其中-u(-user)选项指定数据库 Schema 的用户名和密码；-r(-resolve)选项在加载类的同时要解析它们；-v(-verbose)选项用来输出详细的状态信息。

作为另外一种可选择的方案，你也可以独立地加载各个文件，如下所示（具体的语法取决于你的操作系统。在这些例子中，%均为 UNIX 的命令提示符）：

```
% loadjava -v -u scott/tiger -r Foo*.class Foo.res
```

或者

```
% loadjava -v -u scott/tiger -r Foo*.*
```

所有的这些例子都将导致在数据库里创建以下的 Schema 对象（一般情况下，你只需要知道页面实现类所对应的 Schema 对象的名字就行了）：

- SCOTT:Foo 页面实现类 Schema 对象
对 Foo 类 Schema 对象来说，还可能有一个额外的包名称，这个包名称要么根据 `ojspc -packageName` 选项来确定，要么根据 .jsp 文件相对于当前运行 `ojspc` 的目录的位置来确定。例如，如果在 `ojspc` 的命令行使用了 `-packageName "abc.def"` 选项设置，此时将导致一个 SCOTT:abc/def/Foo 类 Schema 对象被创建。
- SCOTT:Foo\$__jsp__StaticText 类 Schema 对象
与页面实现类有相同的包名称。
- SCOTT:Foo.res 资源 Schema 对象
根据指定的路径信息有一个包名称，路径信息是在文件被加载时要么包含在 JAR 文件中，要么由 `loadjava` 的命令行指定。

注意：如果你正在加载一个预解释过的 SQLJ JSP 页面，你必须也同时加载生成的 profile——要么是一个 .ser Java 资源文件，要么是一个 .class 文件，取决于 SQLJ `-ser2class` 选项的设置。如果它是一个 .ser 文件，它所对应的 Schema 对象的命名是与 .res Java 资源文件的 Schema 对象命名相同的；如果它是一个 .class 文件，它所对应的 Schema 对象的命名是与其他的 .class 文件的 Schema 对象命名相同的。

使用 `loadjava` 加载源文件

假设你已经使用 `ojspc -noCompile` 并且允许了 `-extres` 选项来解释一个 JSP 页面——Foo.jsp，并且生成了以下的文件：

- Foo.java（你希望此文件以源文件的形式被加载到 Oracle8i 中并且由服务器端的编译器所编译。）
- Foo.res

一般情况下，你最好将 Foo.java 和 Foo.res 文件放到一个 JAR 文件中，这里假定为 Foo.jar，然后将此 JAR 文件加载到数据库中，如下所示（这里 % 是 UNIX 的命令提示符）：

```
% loadjava -v -u scott/tiger -r Foo.jar
```

当你允许了 `loadjava -r(-resolve)` 选项时，将导致源文件由服务器端的编译器自动编译，并且生成类 Schema 对象；`-u(-user)` 选项指定数据库 Schema 的用户名和密码；`-v(-verbose)` 选项用来输出详细的状态信息。

作为另外一种可供选择的方案，你也可以独立地加载各个文件，如下所示：

```
% loadjava -v -u scott/tiger -r Foo.java Foo.res
```

或者使用通配符来加载它们：

```
% loadjava -v -u scott/tiger -r Foo*
```

所有的这些例子都将导致在数据库里创建以下的 Schema 对象（一般情况下，你只需要知道页面实现类所对应的 Schema 对象的名字就行了）：

- SCOTT:Foo 源 Schema 对象
当你把一个源文件加载到 Oracle8i 数据库中时，源文件以一个源 Schema 对象的形式被单独存储，并且服务器端的编译器将生成类 Schema 对象。
- SCOTT:Foo 页面实现类 Schema 对象

对 Foo 类 Schema 对象以及源 Schema 对象来说，还可能有一个额外的包名称，这个包名称要么根据 `ojspc -packageName` 选项来确定，要么根据 .jsp 文件相对于当前运行 `ojspc` 的目录的位置来确定。例如，如果在 `ojspc` 的命令行使用了 `-packageName "abc.def"` 选项设置，此时将导致一个 SCOTT:abc/def/Foo 类 Schema 对象被创建。

- SCOTT:Foo\$_jsp_StaticText 类 Schema 对象

与页面实现类有相同的包名称。

- SCOTT:Foo.res 资源 Schema 对象

根据指定的路径信息有一个包名称，路径信息是在文件被加载时要么包含在 JAR 文件中，要么由 `loadjava` 的命令行指定。

要获得关于 `loadjava` 是如何对它所生成的 Schema 对象命名的概述性信息，请参看“数据库的 Java Schema 对象”。

注意：

- 这里所使用的生成文件的名称只是作为一个例子，在未来发行版本中这样的命名实现细节可能要被改变，但是文件的基名（如这里的“Foo”）将总是生成文件名字的一部分。
- 如果你正在为一个 SQLJ JSP 页面加载解释后的源文件（.java 文件），你必须也同时加载生成的 profile——要么是一个 .ser Java 资源文件，要么是一个 .class 文件，取决于 `SQLJ -ser2class` 选项的设置。如果它是一个 .ser 文件，它所对应的 Schema 对象的命名是与 .res Java 资源文件的 Schema 对象命名相同的；如果它是一个 .class 文件，它所对应的 Schema 对象的命名是与其他 .class 文件的 Schema 对象命名相同的（请注意，`ojspc -noCompile` 选项只能阻止 Java 的编译过程，并不能阻止 SQLJ 的编译过程）。

6.5.3 在 Oracle8i 中热加载页面实现类

为了把解释后的 JSP 页面热加载到 Oracle8i 中，可以使用 `session shell` 命令 `java` 来调用页面实现类 Schema 对象的 `main()` 方法。要得到关于如何启动 `session shell` 工具并连接到数据库等信息，请参看“Session Shell 工具概述——`sess_sh`”。

你必须在前面的步骤中已经用 `ojspc -hotload` 选项允许了热加载功能。`-hotload` 选项将导致在页面实现类中实现一个 `main()` 方法以及 `hotloading` 方法，调用 `main()` 方法时将调用 `hotloading` 方法并且热加载页面实现类。

下面举出了一个例子（\$ 是 `sess_sh` 的命令提示符）：

```
$ java SCOTT:Foo
```

其中假设 Foo 是一个类，并且它是由 `ojspc -hotload` 所解释得到的，然后又使用 `loadjava` 被加载到数据库里的 SCOTT Schema 中，与前面几节所提供的例子相似。那么上面这条 `java` 命令将把 Foo 的页面实现类热加载到数据库中。

要获得关于热加载功能的概述信息，请参看“Oracle8i 中的热加载功能概述”。

6.5.4 在 Oracle8i 中发行解释后的 JSP 页面 (publishservlet)

在客户端解释的发布过程中，要想发行解释后的 JSP 页面，可以使用 session shell 命令 publishservlet 来完成。要获得关于如何启动 session shell 工具并连接到数据库等信息，请参看“Session Shell 工具概述——sess_sh”。

publishservlet 命令是一个通用工具，用来发行任何运行在 OSE 下的 servlet。不过，publishservlet 也适用于 JSP 页面实现类（因为它在本质上就是一个 servlet）。

注意：使用 publishservlet 命令所发行的 servlet 和 JSP 页面可以被“取消发行”（即从 JServer 的 JNDI 命名空间中删除），这是通过 session shell 命令 unpublishservlet 来实现的。

publishservlet 的语法和选项概述

首先启动 sess_sh 建立一个数据库连接，一旦 sess_sh 被启动，你就可以在 session shell 的命令提示符下运行 publishservlet 命令了。

publishservlet 命令的通用语法如下所示：

```
$ publishservlet context servletName className -virtualpath path [-stateless]
[-reuse] [-properties props]
```

要使用 publishservlet 命令，你必须指定以下的参数：

1、一个 servlet 环境（用命令行中的 context 指定）

这个参数是 publishservlet 所需要的。你可以使用 Oracle Servlet Engine 的缺省 servlet 环境：

```
/webdomains/contexts/default
```

这里你指定了一个 servlet 环境，那么这些 servlet 环境的环境路径将被使用。

例如，如果你指定了一个 servlet 环境，名字为 mycontext，并且是有下面的命令所创建的：

```
$ createcontext -virtualpath mywebapp /webdomains mycontext
```

那么，在发行 JSP 页面时所用的环境路径就是 mywebapp。

2、一个 servlet 名字（用命令行的 servletName 指定）

这个参数用来指定在 named_servlets 目录下 JSP 页面的名字，publishservlet 需要这个参数，但是对于 JSP 开发人员或者不使用 unpublishservlet 的用户来说，这个参数并没有多少实际意义。这个参数的取值可以是任意的字符串。

3、一个类名字（用目录行的 className 来指定它）

这个参数是被发行的页面实现类所对应的 Schema 对象的名字。

4、一个 servlet 路径（用命令行中的 virtualpath 来指定）

这个参数在发行一个 JSP 页面时是需要的，但是在发行通用的 servlets 时是可以任选的。要指定这个参数可以用 -virtualpath 选项。

环境路径和 servlet 路径两个合在一起（包括主机名和端口号）决定了调用页面的 URL。

注意：

- servlet 环境、servlet 名字和类名字之前不允许出现任何语法，因此在命令

行这三个参数所出现的相对顺序应该如上所示（也就是 context 参数应该在 servletName 和 className 参数之前，而 servletName 参数应该在 className 参数之前；不过这三个参数可以和其他任意的 publishservlet 选项参数混合使用。）。

- 要想允许一个布尔型的选项，如-stateless 只需在命令行上录入选项的名字即可，无需将它设为 true。

除了上面所列必需的参数外，你还可以指定下面的选项：

- -stateless
这是一个布尔型的选项，它用来向 Oracle Servlet Engine 通报当前的 JSP 页面是无状态的——也就是说，这个 JSP 页面在允许过程中不能访问 HttpSession 对象。
- -reuse
这是一个布尔型的选项，用来为一个 JSP 页面指定一个新的 servlet 路径（即虚拟路径）。如果你允许了此选项，那么指定的 servlet 路径将被链接到 JNDI 命名空间中指定的 servlet 名字上，无需 publishservlet 完成整个发行过程。
当你允许一个-reuse 选项时，还要同时指定一个新的 servlet 路径、一个 servlet 环境以及一个前面已经发行过的 servlet 名字。
- -properties props
使用此选项用来指定传递给 JSP 页面作为运行时初始化参数的属性。

使用 publishservlet 发行 JSP 页面的实例

下面的例子将发行一个 JSP 页面——Foo.jsp，假设它已经被加载到 Oracle8i 数据库中了（\$是 sess_sh 的命令提示符）：

```
$ publishservlet /webdomains/contexts/default -virtualpath Foo.jsp
FooServlet SCOTT:Foo
```

为了简单起见，本例使用了 OSE 的缺省 servlet 环境，并且因此导致环境路径就是“/”。
servlet 路径是 Foo.jsp。（你可以对 servlet 路径指定任何你所希望的名字，但是根据原始的源文件来命名是一个良好的编程习惯。）

在 OSE 的 named_servlets 目录下对应的 servlet 名字是 FooServlet，不过除了在取消发行时，这个名字将很少被用到。

SCOTT:Foo 是被发行的页面实现类所对应的 Schema 对象。

在上面的 publishservlet 命令运行以后，最终用户可以使用以下的 URL 来调用 Foo.jsp 页面：

```
http://host[:port]/Foo.jsp
```

如果要在应用程序中其他的 JSP 页面动态地访问 Foo.jsp，这里假定在 Bar.jsp 中，那么可以使用以下的语句来完成（先使用页面相对路径，后使用应用程序相对路径）：

```
<jsp:include page="Foo.jsp" flush="true" />
```

或者

```
<jsp:include page="/Foo.jsp" flush="true" />
```

注意：在 publishservlet 命令中指定的 servlet 路径和 servlet 名字都将被保存到 JNDI 的命名空间里，不过对于 JSP 用户来说只有 servlet 路径是比较有用的。OSE 使用

JNDI 去查找任何已经发行的 JSP 页面或 servlet。

使用 `unpublishservlet` 取消对 JSP 页面的发行

`sess_sh` 工具还提供了一个命令 `unpublishservlet`，它可以用来从 JServer 的 JNDI 命名空间中删除一个 servlet 或者 JSP 页面，不过这一过程并不删除数据库中的 Schema 类 Schema 对象或者页面实现类的 Schema 对象。

要使用 `unpublishservlet` 命令，需要指定环境路径、servlet 路径（在命令行中指的是虚拟路径）以及 servlet 名字。下面列出了 `unpublishservlet` 命令的通用语法：

```
$ unpublishservlet -virtualpath path context servletName
```

这里提供了一个例子，用来将前面一节所发行的页面 `Foo.jsp` 取消发行：

```
$ unpublishservlet -virtualpath Foo.jsp /webdomains/contexts/default
Fooservlet
```

6.6 其他的 JSP 发布问题

本章的绝大部分内容都是针对目标平台为 Oracle Servlet Engine 的解释和发布过程，这是因为在数据库内部运行是一种很特殊的情况，需要特殊的考虑和特殊的逻辑。

本节所讨论的内容是对其他多种环境下的发布应考虑的问题，这些环境的大部分都不是以 Oracle Servlet Engine 为目标平台的。

本节包含了以下几部分的内容：

- Oracle Internet Application Server 与 Oracle Servlet Engine 的文档根目录之比较
- 在非 OSE 环境下使用预解释工具 `ojspc`
- 通用的 JSP 预解释（无需运行）
- 仅发布二进制文件
- WAR 发布
- 用 JDeveloper 发布 JSP 页面

6.6.1 Oracle Internet Application Server 与 Oracle Servlet Engine 的文档根目录之比较

Oracle Servlet Engine 和 Oracle Internet Application Server 都使用了 Oracle HTTP Server（本质上是一个 Apache 环境）作为处理 HTTP 请求的 Web Server。不过，它们两个都使用自己的文档根目录。

在 Oracle Servlet Engine 环境下运行的 JSP 页面和 servlets 是通过 Oracle 提供的 Apache `mod_ose` 模块被定位的，它们使用相应的 servlet 环境的 OSE 文档根目录。OSE 文档根目录位于文件系统中，但是被链接到 Oracle8i 的 JNDI 命名空间中。

请注意，对于运行在 OSE 环境下的 JSP 页面，只有静态文件在文档根本中或文档根目录下被定位，JSP 页面都位于数据库中。

OSE 文档根目录要么是缺省的文档根目录——`$ORACLE_HOME/jis/public_html`，要么是私有 session shell 命令 `createcontext -docroot` 在创建 servlet 环境时所指定的文档根目录。

在 Oracle Internet Application Server（1.0.x 发行版）的 Apache/JServ 环境下运行的 JSP

页面和 servlets 是通过 JServ 提供的 Apache mod_jserv 模块被定位的，它们使用 Apache 的文档根目录。文档根目录（一般情况下即为 htdocs）是在 Apache 的 httpd.conf 配置文件中用 documentRoot 命令来设置的。

对应于 JServ 环境下运行的 JSP 页面来说，静态文件和 JSP 页面都在文档根目录或文档根目录中被定位。

如果你希望在 Apache/JServ 环境和 OSE 环境之间移植 JSP 应用程序，复制或者移动静态文件到合适的文档根目录就行了。

注意：要获得关于 Oracle HTTP Server 的作用，它的 mod_ose 和 mod_jserv 模块等相关信息，请参看“Oracle HTTP Server 所扮演的角色”。

6.6.1 在非 OSE 环境下使用预解释工具 ojspc

Oracle 所提供的 ojspc 工具典型情况下是用于在客户端解释并发布到 Oracle8i 的场合下，这些在“预解释工具 ojspc”中提供了详细描述。不过模拟可以在任何环境下使用 ojspc 去预解释 JSP 页面，这在某些情况下是相当有用的，比如预先解释 JSP 页面可以节省最终用户在第一次请求一个页面时所等待的时间。

如果要在其他非 OSE 目标环境下使用 ojspc 来预解释页面，你必须指定 ojspc -d 选项以设置一个合适的基目录来放置生成的二进制文件。

作为一个例子，考虑在 Apache/JServ 环境下有下列的 JSP 源文件：

```
htdocs/test/foo.jsp
```

而最终用户将使用以下的 URL 来访问此页面：

```
http://host[:port]/test/foo.jsp
```

在页面运行时的按需解释过程中，OracleJSP 解释器将使用一个基目录 htdocs/_pages 来存放生成的二进制文件。

因此，如果需要预解释 foo.jsp 页面时，你必须设置 htdocs/_pages 作为存放生成的二进制文件的基目录，命令如下所示（其中 % 是 UNIX 的命令提示符）：

```
% cd htdocs
% ojspc -d _pages test/foo.jsp
```

在上面所提到的 URL 中使用了 foo.jsp 的应用程序相对路径 test/foo.jsp，因此在运行页面时 OracleJSP 容器将在 htdocs/_pages 目录下的 test 子目录中查找二进制文件。而上面所使用的 ojspc 命令会自动创建 test 这个子目录，在运行页面时，OracleJSP 容器将从这个目录下发现预解释好的二进制文件，因此就不必执行解释过程了（假设源文件在预解释后没有被修改过。缺省情况下，假定源文件是可用的并且配置参数 bypass_source 没有被允许，如果源文件的最好修改日期比二进制文件的修改日期晚的话，页面将被解释）。

6.6.2 通用的 JSP 预解释（无需运行）

在一个按需解释的环境下，可以仅仅在指定对 JSP 页面解析预编译，而无需运行它。这可以通过当最终用户从浏览器中请求一个 JSP 页面时，运行 jsp_precompile 请求参数来实现。

下面是一个例子：

`http://host[:port]/foo.jsp?jsp_precompile`

要获得更多的信息，请参看 Sun 公司的 JSP 技术标准。

6.6.3 仅发布二进制文件

如果你的 JSP 源代码是不公开的，那么你可以只发布二进制文件，而保留 JSP 源文件。这时你应该先预解释 JSP 页面，然后将解释和编译所得到的二进制文件发布。预解释 JSP 页面可以通过两种途径来完成：一种是在按需解释场合下，对所有的 JSP 页面先运行一遍；另外一种是使用 `ojspc` 预解释工具。预解释后的页面可以被发布到任何支持 OracleJSP 容器的环境下，不过有以下两个问题应该注意：

- 你必须合适地发布这些二进制文件。
- 在目标环境下，OracleJSP 必须被合适地配置，以便能在 .jsp（或者 .sqljsp）源文件不可用的情况下正确地运行页面。

发布二进制文件

在解释完 JSP 页面以后，可以得到二进制文件输出目录下的目录结构以及内容，此时把所有的目录结构和内容复制到目标环境下的合适的目录中就可完成发布过程。下面列出了一些应该注意的问题：

- 如果你使用 `ojspc` 进行预解释，你应该用 `ojspc -d` 选项来指定一个存放有效二进制输出文件的目录，然后就可以在这个指定的目录下得到所有的目录结构和内容。
- 如果在 Apache/JServ（按需解释）环境下，先通过运行所有的 JSP 页面以得到二进制文件输出的话，你可以从 `htdocs/_pages` 目录下得到二进制文件的目录结构和内容。

在目标环境下，应该在合适的目录下恢复所得到的二进制文件的目录结构和内容。例如，如果目标环境是 Apache/JServ 的话，应该将得到的二进制文件的目录结构和内容全部复制在 `htdocs/_pages` 目录下。

在仅有二进制文件的情况下配置 OracleJSP

当 .jsp 或者 .sqljsp 源文件不可用时，你应该按如下所示来设置 OracleJSP 的配置参数，以便能正常运行 JSP 页面。

- `bypass_source=true`
- `developer_mode=false`

如果没有设置这些参数，OracleJSP 将总是去查找 .jsp 或者 .sqljsp 文件以检查自从页面实现类生成以后它是否被修改过，如果找不到对应的 .jsp 或者 .sqljsp 文件，那么 OracleJSP 就会输出一个 “file not found” 错误并终止程序运行。

如果正确地设置了这些参数，那么最终用户可以使用与源文件存在时相同的 URL 来访问这个页面，而感觉不出中间的差异。作为一个例子，考虑在 Apache/JServ 环境下，如果 `foo.jsp` 解释后生成的二进制文件被存放在 `htdocs/_pages/test` 目录下，那么在 `foo.jsp` 文件不存在的情况下，最终用户可以使用下面的 URL 来访问此页面：

`http://host[:port]/test/foo.jsp`

要获得如何设置 OracleJSP 的配置参数等信息，请参看“OracleJSP 的配置参数设置”。

6.6.4 WAR 发布

Sun 公司的 JSP 1.1 技术标准支持对 Web 应用程序（包括 JSP 页面）的打包和发布，WAR 文件是由 JAR 工具所创建的。JSP 页面可以以源文件的形式被传递并且可以与任何必需的支持类和静态 HTML 文件一起被发布。

按照 Servlet 2.2 标准，一个 Web 应用程序包括一个发布描述文件——web.xml，它包含了关于应用程序中的 JSP 页面和其他的组成部分的信息。web.xml 文件必须被包含在 WAR 文件中。

Servlet 2.2 标准也为发布描述文件 web.xml 定义了一个 XML DTD，并且精确地指定了一个 servlet 容器应该如何发布一个 Web 应用程序才能符合发布描述文件所规定的规则。

通过这些逻辑，使用 WAR 文件是一种最好的手段来确保一个 Web 应用程序完全按照开发人员所希望的方式被发布到任何标准的 servlet 环境。

在发布描述文件 web.xml 中的发布配置包括在 servlet 路径、JSP 页面和将被调用的 servlets 之间建立映射关系，以及许多其他的特性，如对应用程序目录超时值的设置，把文件扩展名映射到 MIME 类型，以及把错误代码映射到 JSP 错误页面上等。

这里总结了 WAR 文件所包含的内容，如下所示：

- web.xml 发布描述文件
- JSP 页面
- 所需的 JavaBeans 和其他的支持类
- 所需的静态 HTML 文件

要获得更多的信息，请参看 Sun 公司的 Java Servlet 2.2 标准。

注意：在 Oracle 8.1.7 发行以后，OracleJSP WAR 文件的实现细节以及进一步的文档就可以通过 OracleJSP Technology Network 来得到。

在 8.1.7 发行版中，OracleJSP 只在很有限的场合下使用 web.xml 文件，比如仅在 JSP 标签库描述文件和 servlet 的 URL 快捷方式中使用它。

6.6.5 用 JDeveloper 发布 JSP 页面

OracleJSP JDeveloper 3.1 发行版中包含了一个发布选项——“Web Application to Web Server”，它是专门用来处理 JSP 应用程序而添加进来的。

这个选项生成了一个发布记录文件，在文件里指定了以下内容：

- 一个 JAR 文件，包含 JSP 应用程序所需要的 Business Component for Java (BC4J) 类。
- JSP 应用程序所需的静态 HTML 文件。
- 指向 Web Server 的路径。

开发人员可以依据此记录文件立刻发布应用程序，也可以将此文件保存下来以备以后使用。

第 7 章 JSP 标签库和 Oracle JML 标签

本章主要讨论定制标签库，包含基本框架，销售商可用来规划他们各自的库并且利用 OracleJSP 作为例子所提供的库把 JML 标签库进行归档。讨论的主题包括如下几项：

- 标准标签库框架
- JSP 标记语言（JML）实例标签库概述
- JSP 标记语言（JML）标签描述

7.1 标准标签库框架

标准 JSP 页面技术允许厂商创建定制的 JSP 标签库。

一个标签库定义了一系列定制动作的连接，标签能够通过开发者手工对 JSP 页面几项编码或者自动借助于 Java 开发工具被直接运用。一个标签库必须可在不同的 JSP 容器实现类中进行移植。

要获得有关标签库以及标准 JSP 标签框架的详细信息，可参考以下源文件：

- Sun 公司 JSP 标准 1.1 版本
- Sun 公司关于 javax.servlet.jsp.tagext 包的 Javadoc，可在下面的站点中找到：

<http://java.sun.com/j2ee/j2sdkee/techdocs/api/javax/servlet/jsp/tagext/package-summary.html>

注意：如果你正在运用定制标签时，不要运用 Tomcat 3.1 beta 版 servlet/JSP 实现中的 servlet.jar 文件。构造函数签名就会由于 javax.servlet.jsp.tagext.TagAttributeInfo 类而改变，这将导致编译错误。因此建议运用 OracleJSP 或者 Tomcat 3.1 版本的产品中的 servlet.jar 文件。

7.1.1 定制标签库实现概述

通过运用下面通用的格式一个定制标签库就被导入到一个 JSP 页面中：

```
<%@ taglib uri="URI" prefix="prefix"%>
```

注意以下几个方面：

- 一个库的标签是在标签库文件中定义的，这一点在“标签库描述文件”中有相关描述。
- taglib 文件中的 URI 命令指明了在何处可找到标签库的描述文件，这一点在“taglib 指令”中有相关描述。有关可能运用 URI 快捷键请参见“web.xml 标签库的用法”。
- taglib 指令中的前缀是在你的 JSP 页面中运用库中标签所选的字符串。

假定 taglib 指令指定了一个 oracust 前缀：

```
<%@ taglib uri="URI" prefix="oracust" %>
```

再假定库中有一个标签 mytag，你可以以下面的方式来使用 mytag：

```
<oracust:mytag attr1="...", attr2="..." />
```

运用 oracust 前缀通知 JSP 解释器能够在上面 taglib 所指定的 URI 指令中找到已在

标签库定义的 `mytag`。

- 在标签库描述文件中的标签入口处提供了有关标签用法的细节，包括标签的属性（如 `mytag`）以及那些属性的名字。
- 标签语义——作为运用标签结果所发生的动作，在标签处理器类里有定义，这一点在“标签处理器”中有相关描述，每一个标签都有自己的标签处理器类，类名指定在标签描述文件中。
- 标签库描述文件显示出是否一个标签在使用正文。

如上所示，没有正文的标签运用在下面的例子中：

```
<oracust:mytag attr1="...", attr2="..." />
```

相反，有正文的标签在下例中运用：

```
<oracust:mytag attr1="...", attr2="..." />
...body...
</oracust:mytag>
```

- 一个定制标签动作能创建一个或多个 `server-side` 对象，它们为标签自身或其他 JSP scripting 元素所运用，比如 `scriptlets` 等。这些对象也可作为 scripting 变量被引用。关于定制标签运用的 scripting 变量的详细情况在 `tag-extra-info` 类中有定义。

标签能够以下面例子中的语法创建 scripting 变量，创建的对象以 `myobj` 表示：

```
<oracust:mytag id="myobj" attr1="...", attr2="..." />
```

- 一个嵌套标签的标签处理器能够访问一个外部标签的标签处理器，在这种情况下需要这个嵌套标签的任何一项处理过程或者状态管理。

本节后续部分将提供关于这些主题的详细信息。

7.1.2 标签处理器

标签处理器描述了由于运用定制标签而导致的动作结果的语义，它是一种 Java 类的一个实例，依赖于这个标签是否处理起始标签与结尾标签之间的正文的语句，能够实现一、两个标准 Java 接口。

一个标签库的标签库描述（TLD）文件指明了每个标签库中的标签处理器类的名字。

一个标签处理器实例是用在请求时的一个 `server-side` 对象，它具有被 JSP 容器设置的属性，其中包括运用定制标签的 JSP 页面中的页面环境对象，如果这个定制标签在外部的定制标签中被嵌套使用的话，还包括一个父标签处理器对象。

注意：Sun 公司的 JSP 1.1 技术标准没有明确说明：在一个 JSP 页面中同一定制标签的多重使用是运用同一个标签处理器实例还是运用不同标签处理器实例，实现这一点的细节部分就留给 JSP 开发商来决定。OracleJSP 为每一个标签的使用都运用了一个分隔标签处理器实例。

定制标签正文过程

定制标签，如标准 JSP 标签可能有也可能没有正文。在这个定制标签例子中，即便有正文，它也可能不需要标签处理器进行特定的处理。

下面有三种情况：

- 没有正文

在这种情况下只有一个单独的标签，没有 `start` 标签和 `end` 标签，下面是一个通用例子：

```
<oracust:abcdef attr1="...", attr2="..." />
```

- 有正文但不需要通过标签处理器进行特别处理

在这种情况下有一个 `start` 标签和 `end` 标签，两者中间有正文的一些语句，但是标签处理器不必处理正文，正文语句只通行于正常的 JSP 处理过程。下面是一个通用例子：

```
<foo:if cond="<%=...%>
...body executed if cond is true, but not processed by tag handler...
</foo:if>
```

- 正文需要通过标签处理器进行特别处理

在这种情况下有一个 `start` 标签和 `end` 标签，两者中间有正文的一些语句，可是，标签处理器必须进行特别处理。

```
<oracust:ghijkl attr1="...", attr2="...">
...body processed by tag handler...
</oracust:ghijkl>
```

正文处理过程中的整数常量

在下面章节中描述的标签处理接口指定了一种 `doStartTag()` 的方法（下面会进一步描述），借助于这种情况，你必须返回一个合适的整数常量。可能返回的值如下：

- `SKIP_BODY` 要么没有正文，要么跳过正文的赋值和执行。
- `EVAL_BODY_INCLUDE` 有正文但不需要经过标签处理器进行特别处理。
- `EVAL_BODY_TAG` 有正文且需要经过标签处理器进行特别处理。

不需要处理正文的标签处理器

对于一个没有正文或者有正文但不需要经过标签处理器进行特别处理的定制标签来说，标签处理器类实现如下标准接口：

- `javax.servlet.jsp.tagext.Tag`

下面标准支持类用来实现 `Tag` 接口，并能被用作基类：

- `javax.servlet.jsp.tagext.TagSupport`

`Tag` 接口指定了一种 `doStartTag()` 方法和一种 `doEndTag()` 方法，标签开发人员需要在适当的时候为标签处理器类中的这些方法编写代码，当碰到 `start` 与 `end` 这两个标签时它们会独立地调用各自的方法。动作处理过程能在 `doStartTag()` 方法中实现，而 `doEndTag()` 方法将实现任何合适的预处理过程。在没有正文的标签实例中，本质上在这两种方法中什么也未执行。

`doStartTag()` 方法返回一个整数数值，对于一个实现 `Tag` 接口的标签处理器类来说（不管是直接还是间接），这个数值要么是 `SKIP_BODY`，要么是 `EVAL_BODY_INCLUDE`。`EVAL_BODY_TAG` 对于实现 `Tag` 接口的标签处理器类来说是不合法的。

处理正文的标签处理器

对于一个有正文且需要经过标签处理器进行特别处理的定制标签来说，标签处理器类

实现下面的标准接口：

- `javax.servlet.jsp.tagext.BodyTag`

下面标准支持类用来实现 `BodyTag` 接口，并能被用作基类：

- `javax.servlet.jsp.tagext.BodyTagSupport`

`BodyTag` 接口在除 `Tag` 接口中为 `doStartTag()` 和 `doEndTag()` 指定的方法之外，还指定一种 `doInitBody()` 方法和一种 `doAfterBody()` 方法。

正如用标签处理器实现 `Tag` 接口（前面章节“不需要处理正文的标签处理器”中有描述）那样，标签开发人员要借助于标签实现动作处理过程的 `doStartTag()` 方法和任何预处理过程的 `doEndTag()` 方法。

`doStartTag()` 方法返回一个整数数值，对于一个实现 `BodyTag` 接口的标签处理器类来说（不管是直接还是间接），这个数值要么是 `SKIP_BODY`，要么是 `EVAL_BODY_INCLUDE`。`EVAL_BODY_TAG` 对于实现 `BodyTag` 接口的标签处理器类来说是不合法的。

除了要实现 `doStartTag()` 和 `doEndTag()` 的方法之外，标签开发人员需要在适当的时候为在正文被计算之前调用的 `doInitBody()` 编写代码，同样要为在每一次正文计算之后所调用的 `doAfterBody()` 方法编写代码。（正文可能被计算多次，例如在每一次重复循环之后。）

在 `doStartTag()` 方法执行完毕之后，如果返回值为 `EVAL_BODY_TAG`，那么 `doInitBody()` 和 `doAfterBody()` 方法就被执行。

当碰到 `end` 标签时，`doEndTag()` 方法在每一次正文处理之后执行。

对于必须处理正文的定制标签来说，`javax.servlet.jsp.tagext.BodyContent` 类是可用的。它是 `javax.servlet.jsp.JspWriter` 的一个子类，能被用来处理正文的计算，以至于后来能被重新提取。

`BodyTag` 接口包含一种 `setBodyContent()` 方法，能被 JSP 容器用来提供给 `BodyContent` 一个标签处理器实例。

7.1.3 Scripting 变量和 Tag-Extra-Info 类

一个定制标签动作能创建一个或多个 server-side 对象，这在作为 scripting 变量时已经谈到，它们为标签自身或其他 JSP scripting 元素所运用，比如 `scriptlets` 或者其他标签等。关于一个定制标签所定义的 scripting 变量的细节必须在标准 `javax.servlet.jsp.tagext.TagExtraInfo` 抽象类的一个子类中被指定，本书中把这样一个子类称作 `tag-extra-info` 类。

JSP 容器在解释期间要用到 `tag-extra-info` 类。（在把库导入到一个 JSP 页面的 `taglib` 指令中指定的标签库描述文件指定了 `tag-extra-info` 类的用法，如果适当的话，对于任何给定的标签都适用。）

`tag-extra-info` 类提供了一种 `getVariableInfo()` 的方法来检索将在 HTTP 请求过程中被分配的 scripting 变量的名字和类型，JSP 解释器在解释过程中调用该方法，并把这个标准 `javax.servlet.jsp.tagext.TagData` 类的一个实例传递给它。`TagData` 实例指定了用在定制标签中 JSP 语句的属性值。

本节包括以下主题：

- 定义 Scripting 变量

- Scripting 变量作用域
- Tag-Extra-Info 类和 getVariableInfo()方法

定义脚本变量

在一个定制标签中被显式定义的对象通过页面 context 对象能被其他动作所访问,运用对象 ID 作为一个句柄。请参考下面的例子:

```
<oracust:foo id="myobj" attr1="..." attr2="..." />
```

这条语句产生的直接结果就是对象 myobj 能够被标签于页面结束之间的任何脚本所运用, id 的属性就是一个解释时的属性。标签开发人员提供的 tag-extra-info 类将被 JSP 容器所使用。在其他情况下, tag-extra-info 类指定了哪一个类作为 myobj 对象的实例。

JSP 容器把 myobj 输入到页面 context 对象中,在这里它能被其他标签或者脚本元素以下面的句法形式所获取:

```
<oracust:bar ref="myobj"/>
```

myobj 对象通过标签处理器实例就传递给了 foo 和 bar,所需要的一切就是对象 myobj 名字的知识。

注意: 请注意这里 id 和 ref 仅仅是实例属性的名字,但对于这些属性并没有预先定义具体语义,直到标签处理器定义了属性的名字和创建、检索页面 context 中的对象。

脚本变量作用域

指定创建变量标签 tag-extra-info 类中脚本变量的作用域,它可能是下面整数常量中的某一种:

- NESTED——如果这个脚本变量能够在定义它的动作中的 start 标签与 end 标签中被运用。
- AT_BEGIN——如果这个脚本变量能够从 start 标签到页面末尾被运用。
- AT_END——如果这个脚本变量能够从 end 标签到页面末尾被运用。

Tag-Extra-Info 类和 getVariableInfo()方法

你必须为任何一个创建脚本变量的定制标签创建一个 tag-extra-info 类,这个类描述这个脚本变量而且一定是标准 javax.servlet.jsp.tagext.TagExtraInfo 抽象类的一个子类。

TagExtraInfo 类的主要方法是 getVariableInfo(),它能被 JSP 解释器调用并返回标准 javax.servlet.jsp.tagext.VariableInfo 类的实例的一个数组(一个实例数组是属于标签创建的每一个脚本变量的)。

tag-extra-info 类是以下面的脚本变量信息组建的每一个 VariableInfo 实例:

- 它的名字
- 它的 Java 类型
- 它是否是一个新声明变量的布尔值
- 它的作用域

注意: getVariableInfo()方法必须返回一个符合标准的类名,比如 JML 数据类型或者脚

本变量的 Java 类型。(注意不支持原类型)

7.1.4 访问外部标签处理器实例

在定制标签被嵌套的地方，内嵌标签的标签处理器实例就要访问外部标签的标签处理器实例，这一点对内嵌标签完成任何一项处理过程和状态管理将非常有用。

这项功能通过 `javax.servlet.jsp.tagext.TagSupport` 类的静态 `findAncestorWithClass()` 方法被支持。尽管外部标签处理器实例在页面 `context` 对象中没有命名，但因为它是给定标签处理器类中最近的封装实例，它仍然能被访问。

请参考下面的 JSP 代码实例：

```
<foo:bar1 attr="abc">
<foo:bar2/>
</foo:bar1>
```

在 `bar2` 标签处理器类的代码中（按惯例来说即 `Bar2Tag`），你能拥有一条下面的语句：

```
Tag bar1tag=TagSupport.findAncestorWithClass(this,Bar1Tag.class)
```

`findAncestorWithClass()` 方法把下面的作为输入：

- 通过类处理器实例 `this` 对象 `findAncestorWithClass()` 方法能被调用（例子中的 `Bar2Tag` 实例）。
- 标签处理器类 `bar1` 的名字（假定为例子中的 `Bar1Tag`），作为一个 `java.lang.Class` 实例。

`findAncestorWithClass()` 方法返回一个适当的类处理器的实例，即例中的 `Bar1Tag`，作为一个 `javax.servlet.jsp.tagext.Tag` 实例。

对于 `Bar2Tag` 实例来说，在 `Bar2Tag` 需要 `bar1` 标签属性值时或是需要调用 `Bar1Tag` 实例中的方法时，访问外部 `Bar1Tag` 实例是非常有用的。

7.1.5 标签库描述文件

一个标签库描述（TLD）文件是一个既包含一个标签库又包含库中各个标签信息的 XML 文档，TLD 文件的扩展名为 `.tld`。

当碰到库中的一个标签时，JSP 容器使用一个 TLD 文件来决定哪个动作发生。

一个 TLD 文件中标签的入口包括以下几项：

- 定制标签的名字
- 相应标签处理器类的名字
- 相应 `tag-extra-info` 类的名字（如果适用的话）
- 标签正文如何被处理的信息
- 标签属性的信息（这里属性指的是无论什么时候在使用定制标签时你所指定的属性）

下面是标签 `myaction` 在 TLD 文件入口的例子：

```
<tag>
  <name>myaction</name>
  <tagclass>examples.MyactionTag</tagclass>
  <teiclass>examples.MyactionTagExtraInfo</teiclass>
```

```

<bodycontent>JSP</bodycontent>
<info>
    Perform a server-side action (one mandatory attr; one optional)
</info>
<attribute>
    <name>attr1</name>
    <required>true</required>
</attribute>
<attribute>
    <name>attr2</name>
    <required>false</required>
</attribute>
</tag>

```

根据这个入口可知：标签处理器类是 `MyactionTag` 而 `tag-extra-info` 类是 `MyactionTagExtraInfo`。attr1 的属性是需要的，而 attr2 的属性是可选的。

正文内容参数显示出标签正文将被如何处理，下面有三种有效数值：

- 空值表明标签没有运用正文。
- 一个 JSP 值表明标签正文将被当作 JSP 源文件处理或解释。
- 一个 `tagdependent` 值表明标签正文将不被解释，正文中的任何一个文本被看作是静态文本。

注意：在 Tomcat 3.1 servlet/JSP 实现类中，如果标签自身（在 JSP 页面中）没有正文的话，对于给定标签的 TLD 文件正文内容参数是不被读取的。因此，在你的 TLD 文中有一个无效的值（比如用 `none` 代替 `empty`）就可能无法实现。在另一种 JSP 环境中运用此文件将导致出错。

7.1.6 标签库中 web.xml 的用法

Sun 公司 Java Servlet 标准 2.2 版本中描述了一个标准的 servlets 描述符——`web.xml` 文件。JSP 页面在指定一个 JSP 标签库描述文件的定位时能运用这个文件。

对于 JSP 标签库来说，`web.xml` 文件包含一个 `taglib` 元素和两个子元素：

- `taglib`
- `taglib-location`

`taglib-location` 子元素指出了标签库描述文件的相对应用程序的路径（以 “/” 开始。）。

`taglib-uri` 子元素在 JSP 页面的 `taglib` 指令中使用，用来指定一个 URI 的快捷方式，此 URI 被映射到与 `taglib-uri` 协同使用的 `taglib-location` 语句所指定的 TLD 文件的路径上。（术语 URI——统一资源指示器，在某种意义上等价于术语 URL——统一资源定位器，但更通用。）

注意：当一个 JSP 应用程序运用一个 `web.xml` 文件时，你必须用应用程序来扩展 `web.xml`，把它看作是一个 Java 资源文件。

下面是一个为标签库描述文件设计的 `web.xml` 入口实例：

```
<taglib>
```

```
<taglib-uri>/oracustomtags</taglib-uri>
```

```
<taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

这使得在你的 JSP 页面的 `taglib` 指令中 `/oracustomtags` 等价于 `/WEB-INF/oracustomtags/tlds/MyTLD.tld`。请参见“为 TLD 文件运用一个快捷键”中的例子。

请参见 Sun 公司 Java Servlet 标准 2.2 版本和 Sun JSP 标准 1.1 版本以获得有关 `web.xml` 扩展描述符以及它在标签库描述文件中的用法的更多信息。

注意：

- 不要运用来自 Tomcat 3.1 servlet/JSP 实现中的 `web.xml` 实例，它介绍一些新的元素，这些元素将不能通过标准 DTD XML 的检验。
- 不要在 `web.xml` 文件中用术语“urn”代替“uri”。某种 JSP 实现允许使用（比如 Tomcat 3.1），但是运用“urn”将通不过标准 DTD XML 的检验。

7.1.7 taglib 指令

以下面的格式用一条 `taglib` 指令导入一个定制库到一个 JSP 页面中：

```
<%@ taglib uri="URI" prefix="prefix"%>
```

对于 URI，你有下面的选项：

- 指定一个快捷键 URI，定义在一个 `web.xml` 文件中（请参见“标签库中 `web.xml` 文件的用法”）。
- 完整指定标签库描述（TLD）文件名与路径。

为 TLD 文件运用一个快捷键 URI

假定下面对于标签库的 `web.xml` 入口已在标签库描述文件 `MyTLD.tld` 中定义：

```
<taglib>
  <taglib-uri>/oracustomtags</taglib-uri>
```

```
<taglib-location>/WEB-INF/oracustomtags/tlds/MyTLD.tld</taglib-location>
</taglib>
```

给定这个实例，下面在你的 JSP 页面中的指令会导致 JSP 容器在 `web.xml` 找到 `oracustomtags` URI，并且找到标签库描述文件 `MyTLD.tld` 伴随的名字和路径。

```
<%@ taglib uri="/oracustomtags" prefix="oracust"%>
```

这条语句允许你在一个 JSP 页面中运用定制标签库中的任何标签。

完整指定 TLD 文件的名称和路径

如果你不想在运用一个标签库时依赖于 `web.xml` 文件，`taglib` 指令能完整指定 TLD 文件的名称和路径，请看：

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.tld" prefix="oracust"%>
```

路径是作为一个相对路径而被指定的（在本例中以“/”开始）。请参见“请求一个 JSP 页面”中的相关讨论。另外，你可以在 `taglib` 指令中指定一个 `.jar` 文件代替一个 `.tld` 文件，这里 `.jar` 文件包含一个标签库描述文件。当你创建一个 JAR 文件时，标签库描述文件必须

被指定路径和命名（如下）：

```
META-INF/taglib.tld
```

然后 `taglib` 指令可能是下面的形式：

```
<%@ taglib uri="/WEB-INF/oracustomtags/tlds/MyTLD.tld" prefix="oracust"%>
```

7.1.8 定义和运用一个定制标签：End-to-End 实例

本节提供了一个 `end-to-end` 实例，它定义和使用一个定制标签 `loop`，它被用来根据给定的次数来循环迭代标签正文。

本例中包含：

- 一个运用标签的 JSP 源文件页面
- 标签处理器的源代码
- `tag-extra-info` 类的源代码
- 标签库描述文件

JSP 页面实例：exampletag.jsp

下面的例子是一个运用循环标签的 JSP 页面实例，其中外部循环将执行 5 次，内部循环执行 3 次：

```
exampletag.jsp
<%@ taglib prefix="foo" uri="/WEB-INF/exampletag.tld" %>
<% int num=5; %>
<br>
<pre>
<foo:loop index="i" count="<%=num%>">
body1here: i expr: <%=i%> i property: <jsp:getProperty name="i" property="value"
/>

    <foo:loop index="j" count="3">
body2here: j expr: <%=j%>
    i property: <jsp:getProperty name="i" property="value" />
    j property: <jsp:getProperty name="j" property="value" />
</foo:loop>
</foo:loop>
</pre>
```

标签处理器类实例：ExampleLoopTag.java

下面的例子是标签处理器类 `ExampleLoopTag` 的源代码，请看：

- `doStartTag()` 方法返回整数变量 `EVAL_BODY_TAG`，因此标签正文（主要部分，循环）已被处理。
- 在每一次循环执行完毕后，`doAfterBody()` 方法就给计数器里的值增加 1，如果没有其他重复语句的话就返回 `EVAL_BODY_TAG`，并且在最后一次循环后返回 `SKIP_BODY`。

例子包：

```
package examples;
```

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

```

import javax.servlet.jsp.tagext.*;
import java.util.Hashtable;
import java.io.Writer;
import java.io.IOException;
import oracle.jsp.jml.JmlNumber;

public class ExampleLoopTag
    extends BodyTagSupport
{

    String index;
    int count;
    int i=0;
    JmlNumber ib=new JmlNumber();

    public void setIndex(String index)
    {
        this.index=index;
    }

    public void setCount(String count)
    {
        this.count=Integer.parseInt(count);
    }

    public int doStartTag() throws JspException {
        return EVAL_BODY_TAG;
    }

    public void doInitBody() throws JspException {
        pageContext.setAttribute(index, ib);
        i++;
        ib.setValue(i);
    }

    public int doAfterBody() throws JspException {
        try {
            if (i >= count) {
                bodyContent.writeOut(bodyContent.getEnclosingWriter());
                return SKIP_BODY;
            } else
                pageContext.setAttribute(index, ib);
            i++;
            ib.setValue(i);
            return EVAL_BODY_TAG;
        } catch (IOException ex) {
            throw new JspTagException(ex.toString());
        }
    }
}

```

Tag-Extra-Info 类实例: ExampleLoopTagTEI.java

下面是描述运用循环标签脚本变量的 tag-extra-info 类的源代码。

一个 VariableInfo 实例被构造, 同时指定了变量的下面几个方面:

- 变量名是根据索引属性所获得的
- 变量是 oracle.jsp.jml.JmlNumber 类型 (必须被指定为完全符合标准的类名)
- 变量是最新声明的
- 变量作用域是 NESTED

而且, tag-extra-info 类有一个 isValid()方法能判定出这个数的属性是否有效 (必须为整数)

例子包:

```
package examples;

import javax.servlet.jsp.tagext.*;

public class ExampleLoopTagTEI extends TagExtraInfo {
    public VariableInfo[] getVariableInfo(TagData data) {
        return new VariableInfo[]
        {
            new VariableInfo(data.getAttributeString("index"),
                             "oracle.jsp.jml.JmlNumber",
                             true,
                             VariableInfo.NESTED)
        };
    }
    public boolean isValid(TagData data)
    {
        String countStr=data.getAttributeString("count");
        if (countStr!=null) // for request time case
        {
            try {
                int count=Integer.parseInt(countStr);
            }
            catch (NumberFormatException e)
            {
                return false;
            }
        }
        return true;
    }
}
```

标签库描述文件实例: exampletag.tld

下面的例子是标签库的标签库说明 (TLD) 文件, 在本例中, 库只有一个标签构成: 循环。

这个 TLD 文件指定了这个循环标签中的以下几项:

- examples.ExampleLoopTag 是标签处理器类。
- examples.ExampleLoopTagTEI 是 tag-extra-info 类。
- 正文内容指令是 JSP，意思是 JSP 解释器将处理并解释正文代码。
- 这里有两种属性：索引和计数，且都是强制性的。计数属性可能是一个请求时 JSP 的表达式。

```
<?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 tab 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>
<!--
  there should be no <urn></urn> here
-->
  <info>
    A simple tab library for the examples
  </info>

  <!-- example tag -->
  <!-- for loop -->
  <tag>
    <name>loop</name>
    <tagclass>examples.ExampleLoopTag</tagclass>
    <teiclass>examples.ExampleLoopTagTEI</teiclass>
    <bodycontent>JSP</bodycontent>
    <info>for loop</info>
    <attribute>
      <name>index</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>count</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
</taglib>
```

7.2 JSP 标记语言 (JML) 样例标签库概述

OracleJSP 提供 JSP 标记语言 (JML) 样例标签库, 它被移植到任何一种标准的 JSP 环境中。JML 标签和任何一个标准标签库中的那些标签一样, 能够完全与规则的 JSP 脚本兼容并且能被用在任何一个 JSP 页面中。

大部分 JML 标签企图为那些对 Java 并不十分精通的 JSP 程序员简化代码语法, 也有对应的标签来实现 XML 变换 (第 5 章中有相关描述)、bean 绑定以及通用工具。

下面的主题涵盖:

- JML 标签库设计理念
- JML 标签种类
- JML 标签库描述文件和 taglib 指令

注意以下运用 JML 标签的几项需求:

- 安装文件 osjputil.jar 并把它加载到你的类路径中, 这个文件提供了 OracleJSP 的安装情况。
- 确定标签库描述文件 jml.tld 是运用应用程序展开的, 且定位在你的 JSP 页面中 taglib 指令所指定的位置。

注意:

- OracleJSP 也为 SQL 功能提供了一个标签库, 这在 “OracleJSP 的 SQL 标签库” 中有描述。
- 在 OracleJSP 1.1.0.0.0 和 JSP1.1 指令标准发布之前, OracleJSP 所支持的 JML 标签只能作为 Oracle 表达式。(标签库框架直到 JSP1.1 发布后才添加到 JavaServer 页面指令中。) 对这些发行版来说, Oracle 指定的 JML 标签过程已内嵌到 OracleJSP 解释器中, 它被称为 “JML 支持的编译时”, 在附录 C “编译时 JML 标签支持” 中有描述。

7.2.1 JML 标签库设计理念

JavaServer 页面计数是为两种分离的开发人员团体所设计的, 这两种团体一种主要技能是 Java 编程, 另一种是主要技能是在设计静态内容, 特别是 HTML 上, 可能在脚本经验上有些欠缺。

JML 标签库设计了允许大多数 Web 开发人员在具有很少量或者根本没有一点 Java 知识的情况下, 运用程序流控框架的特征来搭建 JSP 应用程序。

这个模型假定商业逻辑包含在一个 Java 开发人员独立开发的 JavaBean 中。

7.2.2 JML 标签种类

JML 标签库涵盖的功能设置比较宽, 主要的功能种类总结如表 7-1 所示。

表 7-1 JML 标签功能种类

标签种类	标签	功能
bean 绑定标签	useVariable	这些标签是在指定 JSP 作用域中声明或者未声明
	useForm	
	useCookie	
	remove	
逻辑/流控标签	if	这些标签提供了简化的语法来定义代码流，比如重复循环或者条件分支
	choose...when...otherwise	
	foreach	
	return	
	flush	
XML 转换标签	transform	这些标签简化了为所有的或者部分 JSP 页面申请一个 XSL StyleSheet 的过程
	styleSheet	

7.2.3 JML 标签库描述文件和 taglib 指令

与任何一个紧跟着 JSP 1.1 指令的标签库一样，JML 库中的标签也都有一个指定的 XML 型的标签库描述（TLD）文件。

这个 TLD 文件提供了 OracleJSP 样例应用程序，它必须被运用 JML 标签的任何 JSP 应用程序所扩展，并且被指定在运用 JML 标签的任何页面中的一个 taglib 指令中。

JML taglib 指令

一个运用 JML 标签的 JSP 页面必须在 taglib 指令中指定一个 TLD 文件，且这个指令提供一个标准的统一资源显式器（URI）来定位这个文件。URI 语法典型情况下使用应用程序相对路径，请看下面的例子：

```
<%@ taglib uri="/WEB-INF/jml.tld" prefix="jml" %>
```

另一种选择，代替 TLD 文件的全路径，在本例中，你可以在 web.xml 文件中指定一个 URI 快捷键然后在你的 taglib 指令中运用这个快捷键。

JML TLD 文件列表

本节列出了 JML 标签库的全部 TLD 文件，这在 OracleJSP 发行版 1.1.0.0.0 中支持。

```
<?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 tab library descriptor -->

<taglib>
    <!-- after this the default space is
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd"
```

```

-->

<tlibversion>1.0</tlibversion>
<jspversion>1.1</jspversion>
<shortname>jml</shortname>
<info>
  Oracle's jml tag library. Not all of the jml
  tag's available in the Oracle JSP environment
  are provided in this library. No jsp: tags are
  duplicated, some tags are unavailable, and some tags
  have stricter syntax. No bean expressions are supported.

  The differences are:
    *-jml:call - not available
    * jml:choose - works as documented
    * jml:flush - works as documented
    * jml:for - works as documented
    * jml:foreach - the type attribute is required, otherwise,
    as documented
    *!jml:forward - use jsp:forward
    *!jml:getProperty - use jsp:getProperty
    * jml:if - works as documented
    *!jml:include - use jsp:include
    *-jml:lock - not available
    *!jml:plugin - use jsp:plugin
    * jml:print - the expression to print must be supplied as
    an attribute. i.e. the tag cannot have a body
    * jml:remove - works as documented
    * jml:return - works as documented
    *-jml:set - not available
    *!jml:setProperty - use jsp:setProperty
    * jml:styleSheet - works as documented
    * jml:transform - works as documented
    *!jml:useBean - use jsp:useBean
    * jml:useCookie - works as documented
    * jml:useForm - works as documented
    * jml:useVariable - works as documented
</info>

  <!-- The choose tag -->
<tag>
  <name>choose</name>
  <tagclass>oracle.jsp.jml.tagext.JmlChoose</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    The outer tag of a multiple choice logic block,
    choose
    when condition1
    when condition2
    otherwise

```

```

    end choose
</info>
</tag>

<!-- The flush tag -->
<tag>
  <name>flush</name>
  <tagclass>oracle.jsp.jml.tagext.JmlFlush</tagclass>
  <bodycontent>empty</bodycontent>
  <info>
    Flush the current JspWriter
  </info>
</tag>

<!-- The for tag -->
<tag>
  <name>for</name>
  <tagclass>oracle.jsp.jml.tagext.JmlFor</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlForTEI</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
    A simple for loop
  </info>

  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>from</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>to</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

<!-- The foreach tag -->
<tag>
  <name>foreach</name>
  <tagclass>oracle.jsp.jml.tagext.JmlForeach</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlForeachTEI</teiclass>
  <bodycontent>JSP</bodycontent>
  <info>
    A foreach loop for iterating arrays, enumerations,
    and vector's.
  </info>

```

```

<attribute>
  <name>id</name>
  <required>true</required>
</attribute>
<attribute>
  <name>in</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
<attribute>
  <name>type</name>
  <required>true</required>
</attribute>
<attribute>
  <name>limit</name>
  <required>false</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>

<!-- The if tag -->
<tag>
  <name>if</name>
  <tagclass>oracle.jsp.jml.tagext.JmlIf</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    A classic if
  </info>

  <attribute>
    <name>condition</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

<!-- The otherwise tag -->
<tag>
  <name>otherwise</name>
  <tagclass>oracle.jsp.jml.tagext.JmlOtherwise</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    (optional) final part of a choose block
  </info>
</tag>

<!-- The print tag -->
<tag>
  <name>print</name>

```

```

<tagclass>oracle.jsp.jml.tagext.JmlPrint</tagclass>
<bodycontent>empty</bodycontent>
<info>
  print the expression specified in the eval attribute
</info>
<attribute>
  <name>eval</name>
  <required>true</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>

  <!-- The remove tag -->
<tag>
  <name>remove</name>
  <tagclass>oracle.jsp.jml.tagext.JmlRemove</tagclass>
  <bodycontent>empty</bodycontent>
  <info>
    remove the specified object from the pageContext
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>false</required>
  </attribute>
</tag>

  <!-- The return tag -->
<tag>
  <name>return</name>
  <tagclass>oracle.jsp.jml.tagext.JmlReturn</tagclass>
  <bodycontent>empty</bodycontent>
  <info>
    Skip the rest of the page
  </info>
</tag>

  <!-- The styleSheet tag -->
<tag>
  <name>styleSheet</name>
  <tagclass>oracle.jsp.jml.tagext.JmlStyleSheet</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    Transform the body of the tag using a stylesheet
  </info>
  <attribute>
    <name>href</name>

```

```

    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
</attribute>
</tag>

<!-- The transform tag -->
<tag>
  <name>transform</name>
  <tagclass>oracle.jsp.jml.tagext.JmlStyleSheet</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    Transform the body of the tag using a stylesheet
  </info>
  <attribute>
    <name>href</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

<!-- The useCookie tag -->
<tag>
  <name>useCookie</name>
  <tagclass>oracle.jsp.jml.tagext.JmlUseCookie</tagclass>
  <teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
  <bodycontent>empty</bodycontent>
  <info>
    create a jml variable and initialize it to a cookie value
  </info>
  <attribute>
    <name>id</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>scope</name>
    <required>false</required>
  </attribute>
  <attribute>
    <name>type</name>
    <required>true</required>
  </attribute>
  <attribute>
    <name>cookie</name>
    <required>true</required>
  </attribute>
</tag>

<!-- The useForm tag -->
<tag>
  <name>useForm</name>

```



```

<tagclass>oracle.jsp.jml.tagext.JmlUseForm</tagclass>
<teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
<bodycontent>empty</bodycontent>
<info>
  create a jml variable and initialize it to a parameter value
</info>
<attribute>
  <name>id</name>
  <required>true</required>
</attribute>
<attribute>
  <name>scope</name>
  <required>false</required>
</attribute>
<attribute>
  <name>type</name>
  <required>true</required>
</attribute>
<attribute>
  <name>param</name>
  <required>true</required>
</attribute>
</tag>

<!-- The useVariable tag -->
<tag>
<name>useVariable</name>
<tagclass>oracle.jsp.jml.tagext.JmlUseVariable</tagclass>
<teiclass>oracle.jsp.jml.tagext.JmlUseTEI</teiclass>
<bodycontent>empty</bodycontent>
<info>
  create a jml variable and initialize it to a parameter value
</info>
<attribute>
  <name>id</name>
  <required>true</required>
</attribute>
<attribute>
  <name>scope</name>
  <required>false</required>
</attribute>
<attribute>
  <name>type</name>
  <required>true</required>
</attribute>
<attribute>
  <name>value</name>
  <required>false</required>
  <rtexprvalue>true</rtexprvalue>
</attribute>

```

```

</tag>

<!-- The when tag -->
<tag>
  <name>when</name>
  <tagclass>oracle.jsp.jml.tagext.JmlWhen</tagclass>
  <bodycontent>JSP</bodycontent>
  <info>
    one part of a choose block, see choose
  </info>
  <attribute>
    <name>condition</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>

</taglib>

```

7.3 JSP 标记语言（JML）标签描述

本节讲解了在 OracleJSP 1.1.0.0.0 运行时实现中支持的 JML 标签,它遵循 JSP 1.1 标准,它们可被分为以下几类:

- Bean 绑定标签描述
- 逻辑和流控标签描述

7.3.1 语法符号与注解

对于标签描述中的语法文件,需注意以下几个方面:

- 斜体表明你必须指定一个数值或字符串
- 可选的属性要标注在方括号中[...]
- 可选属性的缺省值用黑体标注
- 在如果指定一个属性的选择上用竖直线分开|
- 这里运用了前缀“jml:”,这是惯例,但并不必要。你可以在你的 taglib 指令中指定任何前缀。

7.3.2 Bean 绑定标签描述

- JML useVariable 标签
- JML useForm 标签
- JML useCookie 标签
- JML remove 标签

JMLuseVariable 标签

这个标签在声明简单变量时提供了一个比较方便的 jsp:useBean 标签:

语法:

```
<jml:useVariable id = " beanInstanceName"
                [scope = "page | request | session | application"]
                type = "string | boolean | number | fpnumber"
                [value = " stringLiteral | <%= jspExpression %>"] />
```

属性:

- **id**——指定变量被声明，这个属性是必需的。
- **scope**——定义变量的范围或者作用域（例如一个 `jsp:useBean` 标签）这个属性是可选的，缺省作用域为整个页面。
- **type**——指定变量的类型（类型标准指的是 `JmlString`、`JmlBoolean`、`JmlNumber` 或 `JmlFPNumber`）。这个属性是必需的。
- **value**——允许变量在声明中被直接赋值，要么是字符串要么是一个标注在 `<%=...%>` 语句中 JSP 表达式。这个属性是可选的。如果没被指定，这个值就与它最后一次被赋值保持一致（如果它已存在），或者以缺省值来初始化。如果被指定的话，这个值就一直保持着，而不管这个声明是否实例化了这个对象或者仅仅只是从命名作用域中得到它。

参看下面的例子:

```
<jml:useVariable id = "isValidUser" type = "boolean" value = "<%= dbConn.isValid()
%>" scope = "session" />
```

它等价于下面的语句:

```
<jsp:useBean id = "isValidUser" class = "oracle.jsp.jml.JmlBoolean" scope =
"session" />
<jsp:setProperty name="isValidUser" property="value" value = "<%=
dbConn.isValid() %>" />
```

JML useForm 标签

这个标签为声明变量提供了一个方便的语法，并从请求中设置它们传递的数值。

语法:

```
<jml:useForm id = " beanInstanceName"
            [scope = "page | request | session | application"]
            [type = "string | boolean | number | fpnumber"]
            param = " requestParameterName" />
```

属性:

- **id**——指定变量被声明或引用，这个属性是必需的。
- **scope**——定义变量的范围或者作用域（例如一个 `jsp:useBean` 标签）这个属性是可选的，缺省作用域为整个页面。
- **type**——指定变量的类型（类型标准指的是 `JmlString`、`JmlBoolean`、`JmlNumber` 或 `JmlFPNumber`）。这个属性是可选的，缺省值设置为字符串。
- **param**——指定了请求参数的名字，它的值是被用来设置变量的。这个属性是必需的。如果请求参数存在，那么变量的值在不断更新，不管这个声明是否让这个变量产生。如果请求参数不存在，那么变量的值保持不变。

例子: 下面这个例子是把名为 `user` 的请求参数的值赋给了一个字符串类型名字为 `user`

的会话变量:

```
<jml:useForm id="user" type="string" param="user" scope="session"/>
```

它等价于下面的语句:

```
<jsp:useBean id="user" class="oracle.jsp.jml.JmlString" scope="session"/>
<jsp:setProperty name="user" property="value" param="user"/>
```

JML useCookie 标签

这个标签为声明变量提供了一条简便的语句, 并且把它们值设置成已经包含在 cookie 中的值。

语法:

```
<jml:useCookie id = " beanInstanceName"
               [scope = "page | request | session | application"]
               [type = "string | boolean | number | fpnumber"]
               cookie = " cookieName" />
```

属性:

- **id**——指定变量被声明或引用, 这个属性是必需的。
- **scope**——定义变量的范围或者作用域(例如一个 `jsp:useBean` 标签) 这个属性是可选的, 缺省作用域为整个页面。
- **type**——鉴别变量的类型(类型标准指的是 `JmlString`、`JmlBoolean`、`JmlNumber` 或 `JmlFPNumber`)。这个属性是可选的, 缺省值设置为字符串。
- **cookie**——指定了 `cookie` 的名字, 它的值是被用来设置变量的。这个属性是必需的。如果 `cookie` 存在, 那么变量的值在不断更新, 不管这个声明是否让这个变量产生。如果 `cookie` 不存在, 那么变量的值保持不变。

例子: 下面这个例子是把名为 `user` 的 `cookie` 的值赋给了一个字符串类型名字为 `user` 的请求变量:

```
<jml:useCookie id="user" type="string" cookie="user" scope="request"/>
```

它等价于下面的语句:

```
<jml:useCookie id = "user" type = "string" cookie = "user" scope = "request" />
```

它等价于下面的语句:

```
<jsp:useBean id = "user" class = "oracle.jsp.jml.JmlString" scope = "request"
/>
```

```
<%
    Cookies [] cookies = request.getCookies();
    for (int i = 0; i < cookies.length; i++) {
        if (cookies[i].getName().equals("user")) {
            user.setValue(cookies[i].getValue());
            break;
        }
    }
%>
```

JML remove 标签

这个标签把一个对象从它的作用域中去掉。

语法:

```
<jml:remove id="beanInstanceName"
           [scope="page" | request | session | application"] />
```

属性:

- **id**——指定被去掉的 bean 的名字, 这个属性是必需的。
- **scope**——这个属性是可选的。如果没被指定, 那么作用域就以顺序 1)page, 2)request, 3)session, 4)application 进行搜索。名字与 id 匹配的第一个对象就被去掉。

例子: 下例是去掉会话 user 对象:

```
<jml:remove id="user" scope="session" />
```

它等价于下面的语句:

```
<% session.removeValue("user"); %>
```

7.3.3 逻辑与流控标签描述

本节讲解了下面的 JML 标签, 它们被逻辑与流控所使用:

- JML if 标签
- JML choose...when...[otherwise]标签
- JML for 标签
- JML foreach 标签
- JML return 标签
- JML flush 标签

这些标签是为那些没有大量 Java 编程经验的程序员所设计, 它们能被用来代替 Java 逻辑与流控语句, 比如循环、条件分支等。

JML if 标签

这个标签是求一个单个条件的值, 如果条件为 true, 那么 if 标签中的正文就被执行。

语句:

```
<jml:if condition = "<%= jspExpression %>" >
    ... body of if tag (executed if the condition is true)...
</jml:if>
```

属性:

- **condition**——指定将被执行的条件表达式。这个属性是必需的。

例子: 下面的电子商务例子显示了一个用户的购物推车中的信息, 代码来核对保存当前 T 恤订单的变量是否为空, 如果不为空的话, 那么订单信息就会被显示出来。假定 currTS 是 JmlString 类型。

```
<jml:if condition = "<%= !currTS.isEmpty() %>" >
    <S>(size: <%= currTS.getValue().toUpperCase() %>)</S>&nbsp;
</jml:if>
```

JML choose...when...[otherwise]标签

choose 标签与 (when 与 otherwise 标签联合使用) 提供了一条有多种条件的语句。

`choose` 标签的正文中包含了一个或多个 `when` 标签, 每一个 `when` 标签代表着一个条件。哪个 `when` 标签的值首先为 `true`, 那么这个 `when` 标签中的正文就被执行。(一个 `when` 正文的最大值被执行。)

如果 `when` 条件中语句没有一个为 `true`, 且可选的 `otherwise` 标签被指定的话, 那么 `otherwise` 标签中的正文就被执行。

语法:

```
<jml:choose>
  <jml:when condition = "<%= jspExpression %>" >
    ...body of 1st when tag (executed if the condition is true)...
  </jml:when>
  ...
  [... optional additional when tags...]
  [ <jml:otherwise>
    ... body of otherwise tag (executed if all when conditions false)...
  </jml:otherwise> ]
</jml:choose>
```

属性: `when` 标签使用下面的属性 (`choose` 和 `otherwise` 标签没有属性):

- **condition**——指定被计算的条件表达式。这个属性是必需的。

例子: 下面的电子商务例子显示了一个用户的购物推车中的信息, 代码来核对是否有礼品被订购, 如果有的话, 那么订单信息就会被显示出来, 否则用户需要重新订购。假定 `orderedItem` 是 `JmlBoolean` 类型。

```
<jml:choose>
  <jml:when condition = "<%= orderedItem.getValue() %>" >
    You have changed your order:
    -- output the current order --
  </jml:when>
  <jml:otherwise>
    Are you sure we can't interest you in something, cheapskate?
  </jml:otherwise>
</jml:choose>
```

JML for 标签

这个标签具有循环迭代的能力, 正如 `Java` 中的 `for` 循环。

`id` 属性是一个具有 `java.lang.Integer` 类型的局部循环变量, 它包含着当前范围元素的值。这个范围是从 `from` 属性所代表的值开始, 每执行一次正文循环, 值就增加 1, 直到执行到 `to` 属性所代表的值为止。

一旦超出范围就跳出循环, 控制就执行紧跟着 `for` 语句的第一条 `end` 标签。

注意: 不允许出现降序循环, 即 `from` 值必须小于或等于 `to` 值。

语法:

```
<jml:for id = " loopVariable"
  from = "<%= jspExpression %>"
  to = "<%= jspExpression %>" >
  ... body of for tag (executed once at each value of range, inclusive)...
</jml:for>
```

属性:

- **id**——指定循环变量的名字，它保存着范围中当前的值。这是一个 `java.lang.Integer` 值，只能被用在标签正文里。这个属性是必需的。
- **from**——指定了范围的起点，它是一个表达式但必须被计算成一个 Java 整型值。这个属性是必需的。
- **to**——指定了范围的结束点，它是一个表达式但必须被计算成一个 Java 整型值。这个属性是必需的。

例子：下面的例子重复输出 “Hello World”，输出的标题越来越小。

```
<jml:for id="i" from="<%= 1 %>" to="<%= 5 %>" >
    <H<%=i%>>
        Hello World!
    </H<%=i%>>
</jml:for>
```

JML foreach 标签

这个标签具有循环查找一个集合中所有数值的能力。

标签正文将按照集合中的每个元素都执行一次。（如果集合为空，正文就不执行。）

id 属性是一个局部循环变量，它包含着当前集合元素的值。它的类型在 **type** 属性中被指定。（指定的类型必须与集合中元素的类型相匹配，比如 `application` 等。）

这个标签当前支持以下几种数据结构的类型：

- Java 数组
- `Java.util.Enumeration`
- `java.util.Vector`

语法:

```
<jml:foreach id = " loopVariable"
    in = "<%= jspExpression %>"
    limit = "<%= jspExpression %>"
    type = " package.class" >
    ... body of foreach tag (executes once for each element in data
structure)...
</jml:foreach>
```

属性:

- **id**——指定循环变量，它保存着当前元素在每一步循环中的值，它只能被用在标签正文中。它的类型与 **type** 属性指定的类型一致。这个属性是必需的。
- **in**——指定一个 JSP 表达式，它来计算一个 Java 数组、枚举对象、相量对象的值。这个属性是必需的。
- **limit**——指定一个 JSP 表达式，它来计算一个定义循环中的最大值的 Java 整型数值，不管这个元素在集合中的数值如何。这个属性是必需的。
- **type**——指定循环变量的类型。它需要与集合中元素的类型相匹配。这个属性是必需的。

例子：下例重复执行请求参数：

```
<jml:foreach id="name" in="<%= request.getParameterNames() %>"
```

```

type="java.lang.String" >
    Parameter: <%= name %>
    Value: <%= request.getParameter(name) %> <br>
</jml:foreach>
或者，你想用多个数值来处理请求参数：
<jml:foreach id="name" in="<%= request.getParameterNames() %>"
type="java.lang.String" >
    Parameter: <%= name %>
    Value: <jml:foreach id="val"
in="<%=request.getParameterValues(name)%>"
        type="java.lang.String" >
            <%= val %> :
        </jml:foreach>
    <br>
</jml:foreach>

```

JML return 标签

当执行到这个标签时，将从页面返回而没有进一步的过程了。

语法：

```
<jml:return />
```

属性：

(无)

例子：下面即是一个计时器终止后不再继续执行页面而返回的例子。

```

<jml:if condition="<%= timer.isExpired() %>" >
    You did not complete in time!
    <jml:return />
</jml:if>

```

JML flush 标签

这个标签把页面缓冲区内当前的内容写回到客户端浏览器。

这就要求页面必须有缓冲区，否则将没有结果输出。

语法：

```
<jml:flush />
```

属性：(无)

例子：下面的例子在执行一个费时的操作之前刷新了当前的页面内容：

```

<jml:flush />
<% myBean.expensiveOperation(out); %>

```


第 8 章 OracleJSP 对 NLS 的支持

根据 Sun 公司 JSP 标准 1.1 版本的规定, OracleJSP 提供标准的本国语言支持 (NLS), 并且为不支持多字节参数编码的 servlet 环境提供扩展支持。

标准 Java 借助于统一内部文本表示法中 Unicode 2.0 的用法支持本地化的内容。Unicode 用来把一个基本的字符集变换成另一种字符集。

本章描述的主要方面是关于 OracleJSP 是怎样支持 NLS 的, 主要涵盖以下几个主题(其他主题将会在以后发布):

- 页面指令中的内容类型设置
- 动态内容类型设置
- OracleJSP 对多字节参数编码的扩展支持

8.1 页面指令中的内容类型设置

你可以运用页面指令中的 `contentType` 参数设置 MIME 类型, 也可随意地为一个 JSP 页面进行编码设置。MIME 类型在运行时可运用于 HTTP 的响应中。字符编码如果设置的话, 就既能运用于解释过程中的页面文本又能用于 HTTP 的响应中。

把下面的语法运用到页面指令中:

```
<%@ page...contentType="TYPE; charset=character_set"...%>
```

或者, 当使用缺省字符集时设置 MIME 的类型:

```
<%@ page...contentType="TYPE"; ...%>
```

TYPE 是一个 IANA (Internet Assigned Numbers Authority) MIME, 而 `character_set` 是一个 IANA 字符集。(当指定一个字符集时, 分号后的空格属于可选项。)

例如:

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
```

或者表示为:

```
<%@ page language="java" contentType="text/html"; %>
```

缺省 MIME 类型为 `text/html`。IANA 在下面的站点中维护有 MIME 类型的记录:

<ftp://venera.isi.edu/in-notes/iana/assignments/media-type/media-types>

(只要你使用的字符集是 Java 或者 Web 浏览器支持, 就不必考虑 JSP 需不需要使用字符集, 但是 IANA 站点列出了大部分普通字符集。推荐使用名字 MIME。)

一个页面指令中的参数是静态的。如果一个页面在执行过程中发现一个不同的设置对于响应是必要的, 它能够完成下面当中的一项:

- 在执行过程中, 使用 servlet 响应对象 API 来设置内容类型, 可参考“动态内容类型设置”中的描述。
- 转向另一个 JSP 页面或者一个 servlet 中的前一个请求。

注意:

- 设置 `contentType` 的页面指令要尽可能地在 JSP 页面中早出现。

- 不同于 ISO-8859-1，用一个字符集写成的 JSP 页面必须在一个页面指令中设置适当的字符集。它不能设置成动态的，因为在解释过程中页面能觉察到。动态设置仅用在运行时。
- JSP 1.1 标准假定了写一个 JSP 页面和它输出内容所用的字符集是相同的。
- 本书为简单起见，假定页面文本、请求参数以及响应参数全部使用同一种编码方式（虽然其他场景在技术上是可能实现的）。尽管 Netscape 浏览器和 IE 接受你为响应参数指定的设置，但请求参数编码要受到浏览器的控制。

8.2 动态内容类型设置

如果 HTTP 响应对象的内容类型在运行前是未知的话，那么你就在 JSP 页面中动态地设置它。标准 `javax.servlet.ServletResponse` 接口指出了有关这个方法：

```
public void setContenttype (java.lang.String contenttype)
```

一个 JSP 页面暗含的响应对象是一个 `javax.servlet.http.HttpServletResponse` 例子，在这里 `HttpServletResponse` 接口扩展了 `ServletResponse` 接口。

`setContenttype()` 方法输出，如同 `contentType` 在一个页面指令中的设置一样，仅包含一个 MIME 类型或者是一个字符集和一个 MIME 类型。

例如：

```
response.setContentType("text/html; charset=UTF-8");
```

或者：

```
response.setContentType("text/html");
```

作为伴随着一个页面指令，MIME 类型的缺省值为 `text/html`，缺省字符编码为 ISO-8859-1。

这种方法对于在解释过程中解释 JSP 页面文本来说不起作用。如果在解释过程中需要一个特殊的字符集，那么它就必须在一个页面指令中先被指定。

请注意下面这些比较重要的用法：

- 如果你正在使用 `setContenttype()` 方法，那么 JSP 页面就必须进行缓冲。（缺省时具有缓冲，千万不要在一个页面指令中设置 `buffer="none"`。）
- `setContenttype()` 调用必须在页面的前端出现，即在任何输出到浏览器或者任何 `jsp:include` 命令执行前（先对 JSP 输出到浏览器的 `buffer` 进行刷新）。
- 在 `servlet 2.2` 环境中，响应对象有一个 `setLocale()` 方法来设置基于指定场所的缺省字符集，而不考虑任何先前的字符集。

例如：下面的调用结果是一个 `shift_JIS` 字符集：

```
response.setLocale(new Locale("ja", "JP"));
```

8.3 OracleJSP 对多字节参数编码的扩展支持

有关请求参数的字符编码在 HTTP 标准中没有明确定义。大多数 `servlet` 容器在使用 `servlet` 缺省编码 ISO-8859-1 时必须中断它们。

在这种环境下，`servlet` 容器不能对多字节请求参数和 `bean` 属性设置进行编码，

OracleJSP 通过 `translate_params` 配置参数提供了扩展支持。

也可能在 JSP 页面中使用等价的代码，这一点在 Oracle Servlet 环境中是很有必要的。

注意：不要使 `translate_params` 标志在下面的环境中以 `true` 值存在：

- 当 servlet 容器恰好在处理多字节参数自身编码时，在这种状况下设置 `translate_params` 为 `true` 将引起不正确的结果产生。写到这里，让我们想起 Apache/JServ, JSWDK, 以及 Tomcat 都不能正确处理多字节参数编码的情况。
- 当请求参数使用一种不同于 JSP 页面响应所指定的或 `setContentType()` 方法的编码时。
- 当 JSP 页面中与 `translate_params` 得到的功能等价的代码已经存在时。

8.3.1 `translate_params` 在不考虑非多字节 Servlet 容器中的影响

设置 `translate_params` 为 `true` 以忽略不能对多字节请求参数以及 bean 属性设置的 servlet 容器。

当这个标志被允许时，OracleJSP 就对基于响应对象的请求参数和 bean 属性设置进行编码，在 `response.getCharacterEncoding()` 方法中有显示。

特殊地，`translate_params` 标志对参数名字和数值有影响：

- request 对象 `getParameter()` 方法输出
- request 对象 `getParameterValues()` 方法输出
- request 对象 `getParameterNames()` 方法输出
- 为 bean 属性设置的 `jsp:setProperty` 设置

8.3.2 `translate_params` 配置参数的等价代码

`translate_params` 配置参数是一个运行时参数，它不能在 Oracle Servlet Engine 环境中被设置。（通过 `ojspc` 命令选项，解释时配置能为 OSE 环境所设置。这里没有等价于运行时的参数。）

基于这种原因，也可能是其他原因，通过 JSP 页面中的 scriptlet 代码对于能被实现的等价功能是非常有用的。例如：

```
<%@ page contentType="text/html; charset=EUC-JP" %>
...
String paramName="XXYYZZ";      // where XXYYZZ is a multibyte string
paramName =
    new String(paramName.getBytes(response.getCharacterEncoding()),
"ISO8859_1");
String paramValue = request.getParameter(paramName);
paramValue= new String(paramValue.getBytes("ISO8859_1"), "EUC-JP");
...
```

代码能实现以下功能：

- 设置 `XXYYZZ` 作为参数名字来搜索。（假定 `XX`, `YY`, `ZZ` 是三种日文字符。）
- 把参数名字编成 ISO-8859-1 码，servlet 容器字符集，以致于 servlet 容器能够解释。

（运用请求对象的字符编码，首先为参数名字创建一个字节数组。）

- 通过从请求对象中搜索匹配于参数名字来获得参数值。（能够找到一个匹配的数，因为参数名字在请求对象中也是用 ISO-8859-1 编码。）
- 为进一步处理或者输出到浏览器，把参数值编码成 EUC-JP。

参考下面两节中借助于 `translate_params` 被设置为允许时的一个 NLS 实例和一个不借助于 `translate_params` 设置而包含等价代码的实例。

8.3.3 借助于 `translate_params` 的 NLS 实例

下面的例子接受了一个日文字符的用户名字并把它正确地输出到浏览器中。在 `servlet` 环境中不能对多字节请求参数进行编码，本例则借助于 OracleJSP 配置设置 `translate_params=true`。

假定 `XXYY` 是参数名字（等价于日文中的“用户名字”），`AABB` 是缺省值（也用日文表示）。

（参考下一节中具有代码等价于 `translate_params` 功能的一个实例，因此并没有借助于 `translate_params` 设置。）

```
<%@ page contentType="text/html; charset=EUC-JP" %>

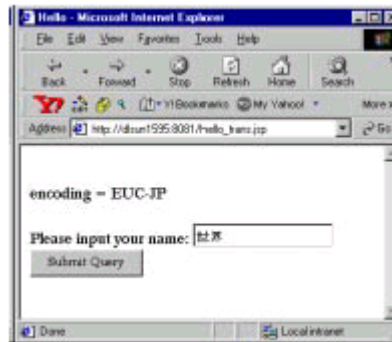
<HTML>
<HEAD>
<TITLE>Hello</TITLE></HEAD>
<BODY>
<%
    //charset is as specified in page directive (EUC-JP)
    String charset = response.getCharacterEncoding();
%>
    <BR> encoding = <%= charset %> <BR>

<%

String paramValue = request.getParameter("XXYY");

if (paramValue == null || paramValue.length() == 0) { %>
    <FORM METHOD="GET">
    Please input your name: <INPUT TYPE="TEXT" NAME="XXYY" value="AABB" size=20>
<BR>
    <INPUT TYPE="SUBMIT">
    </FORM>
<% }
else
{ %>
    <H1> Hello, <%= paramValue %> </H1>
<% } %>
</BODY>
</HTML>
```

下面是例子的输入图：



例子的输出图:



8.3.4 不借助于 translate_params 的 NLS 实例

下面的例子作为处理过程中的实例，接受了一个日文字符的用户名字并把它正确地输出到浏览器中。然而本例中具有等价于 `translate_params` 功能的代码，因此并没有借助于 `translate_params` 设置。

注意：如果你使用了 `translate_params` 的等价代码，就不要把 `translate_params` 标志设置为允许，否则将会导致错误的结果。（这不是在 OSE 环境中关心的事，这里不支持 `translate_params` 标志。）

假定 `XXYY` 是参数名字（等价于日文中的“用户名字”），`AABB` 是缺省值（也用日文表示）。

关于本例中关键性代码的解释，请参见“`translate_params` 配置参数的等价代码”的内容。

```
<%@ page contentType="text/html; charset=EUC-JP" %>

<HTML>
<HEAD>
<TITLE>Hello</TITLE></HEAD>
<BODY>
<%
    //charset is as specified in page directive (EUC-JP)
    String charset = response.getCharacterEncoding();
%>
    <BR> encoding = <%= charset %> <BR>
```

```
<%
String paramName = "XXYY";

paramName = new String(paramName.getBytes(charset), "ISO8859_1");

String paramValue = request.getParameter(paramName);

if (paramValue == null || paramValue.length() == 0) { %>
    <FORM METHOD="GET">
        Please input your name: <INPUT TYPE="TEXT" NAME="XXYY" value="AABB" size=20>
    <BR>
        <INPUT TYPE="SUBMIT">
    </FORM>
<% }
else
{
    paramValue= new String(paramValue.getBytes("ISO8859_1"), "EUC-JP"); %>
    <H1> Hello, <%= paramValue %> </H1>
<% } %>
</BODY>
</HTML>
```

第9章 程序实例

本章列举了大量有关 JSP 页面以及 JavaBeans 的实例代码，它们常应用在下列的类中：

- JDBC 实例
- JavaBean 数据库访问实例
- 定制标签实例
- 特定 Oracle 扩展程序设计实例
- 在 Servlet 2.0 环境中运用 globals.jsa 实例

9.1 基本实例

本节列举了较多相对来说比较基本而又可用作 Oracle JML 数据类型的 JSP 实例，包括一个基本的“hello”例子、一个运用 Java Bean 的例子和一个中等难度的购物车的例子。下面是所列举的实例：

- Hello 页面对应 hellouser.jsp
- Usebean 页面对应 usebean.jsp
- 购物推车页面对应 cart.jsp

这些实例可运用标准数据类型代替，但是 JML 数据类型具有很多优点，这一点在“JML 扩展数据类型”中有描述。JML 数据类型也可移植到其他 JSP 环境中。

9.1.1 Hello 页面—hellouser.jsp

这个实例是一个基本 JSP “hello”页面的例子。用户可在浏览器出现的表格中键入自己的名字，键入后，JSP 页面表格的顶端就会显示出键入的名字。

```
<%-----
Copyright (c) 1999, Oracle Corporation. All rights reserved.
-----%>
<%page session="false" %>
<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope="request" >
    <jsp:setProperty name="name" property="value" param="newName" />
</jsp:useBean>

<HTML>
<HEAD>
<TITLE>
Hello User
</TITLE>
</HEAD>

<BODY>

<% if (!name.isEmpty()) { %>
<H3>Welcome <%= name.getValue() %></H3>
```

```

<% } %>
<P>
Enter your Name:
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>

```

9.1.2 UseBean 页面——Usebean.jsp

这个页面运用一个简单的 Java Bean（名字为 NameBean）来演示 jsp:useBean 标签的用法，而且还提供了 bean 的源代码。

usebean.jsp 源代码：

```

<%-----
Copyright (c) 1999, Oracle Corporation. All rights reserved.
-----%>

<%@ page import="beans.NameBean" %>

<jsp:useBean id="pageBean" class="beans.NameBean" scope="page" />
<jsp:setProperty name="pageBean" property="*" />

<jsp:useBean id="sessionBean" class="beans.NameBean" scope="session" />
<jsp:setProperty name="sessionBean" property="*" />

<HTML>
<HEAD> <TITLE> The UseBean JSP </TITLE> </HEAD>
<BODY BGCOLOR=white>

<H3> Welcome to the UseBean JSP </H3>
<P><B>Page bean: </B>
<% if (pageBean.getNewName().equals("")) { %>
    I don't know you.
<% } else { %>
    Hello <%= pageBean.getNewName() %> !
<% } %>

<P><B>Session bean: </B>
<% if (sessionBean.getNewName().equals("")) { %>
    I don't know you either.
<% } else {
    if ((request.getParameter("newName") == null) ||
        (request.getParameter("newName").equals(""))) { %>
        Aha, I remember you.
<%     } %>
    You are <%= sessionBean.getNewName() %>.
<% } %>

```



```

<P>May we have your name?
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>
</BODY>
</HTML>

```

NameBean. java 源代码:

```

package beans;

public class NameBean {

    String newName="";

    public void NameBean() { }

    public String getNewName() {
        return newName;
    }
    public void setNewName(String newName) {
        this.newName = newName;
    }
}

```

9.1.3 购物车页面—cart. jsp

这个实例解释了如何运用对话状态来维护购物车，比如用户选择一件 T 恤或汗衫来订购，订单随后就能显示出来，如果继续订购，订单也会跟着改变，页面将刷新订单内容，适当时也可进行删除操作。

cart.jsp 是主要的源代码文件，它引用了 index.jsp，下面是页面源代码：

cart. jsp 源代码:

```

<%-----
    Copyright (c) 1999-2000, Oracle Corporation. All rights reserved.
    -----%>
<jsp:useBean id="currSS" scope="session" class="oracle.jsp.jml.JmlString"
/>
<jsp:useBean id="currTS" scope="session" class="oracle.jsp.jml.JmlString"
/>

<HTML>

<HEAD>
    <TITLE>Java Store</TITLE>
</HEAD>

<BODY BACKGROUND=images/bg.gif BGCOLOR=#FFFFFF>

<jsp:useBean id="sweatShirtSize" scope="page" class="oracle.jsp.jml.JmlString" >

```

```

        <jsp:setProperty name="sweatShirtSize" property="value" param="SS" />
    </jsp:useBean>
    <jsp:useBean id="tshirtSize" scope="page" class="oracle.jsp.jml.JmlString"
>
        <jsp:setProperty name="tshirtSize" property="value" param="TS" />
    </jsp:useBean>

    <jsp:useBean id="orderedSweatshirt" scope="page"
    class="oracle.jsp.jml.JmlBoolean" >
        <jsp:setProperty name="orderedSweatshirt" property="value"
            value= '<%= !(sweatShirtSize.isEmpty() ||
sweatShirtSize.getValue().equals("none")) %>' />
    </jsp:useBean>

    <jsp:useBean id="orderedTShirt" scope="page" class="oracle.jsp.jml.JmlBoolean" >
        <jsp:setProperty name="orderedTShirt" property="value"
            value= '<%= !(tshirtSize.isEmpty() || tshirtSize.getValue().equals("none"))
%>' />
    </jsp:useBean>

    <P>
    <TABLE BORDER=0 CELLPADDING=0 CELLSPACING=0 WIDTH=100% HEIGHT=553>
        <TR>
            <TD WIDTH=33% HEIGHT=61>&nbsp;</TD>
            <TD WIDTH=67% HEIGHT=61>&nbsp;</TD>
        </TR>
        <TR>
            <TD WIDTH=33% HEIGHT=246>&nbsp;</TD>
            <TD WIDTH=67% HEIGHT=246 VALIGN=TOP BGCOLOR=#FFFFFF>
    <% if (orderedSweatshirt.getValue() || orderedTShirt.getValue()) { %>
        Thank you for selecting our fine JSP Wearables!<P>

    <% if (!currSS.isEmpty() || !currTS.isEmpty()) { %>
        You have changed your order:
        <UL>
            <% if (orderedSweatshirt.getValue()) { %>
                <LI>1 Sweatshirt
                <% if (!currSS.isEmpty()) { %>
                    <S>(size: <%= currSS.getValue().toUpperCase() %>)</S>&nbsp;  
                <% } %>
                (size: <%= sweatShirtSize.getValue().toUpperCase() %> )
            <% } else if (!currSS.isEmpty()) { %>
                <LI><S>1 Sweatshirt (size: <%= currSS.getValue().toUpperCase()
%>)</S>
            <% } %>

            <% if (orderedTShirt.getValue()) { %>
                <LI>1 Tshirt
                <% if (!currTS.isEmpty()) { %>
                    <S>(size: <%= currTS.getValue().toUpperCase() %>)</S>&nbsp;  

```

```

        <% } %>
        (size: <%= tshirtSize.getValue().toUpperCase() %>)
    <% } else if (!currTS.isEmpty()) { %>
        <LI><S>1 Tshirt (size: <%= currTS.getValue().toUpperCase()
        %>)</S>
        <% } %>
    </UL>
    <% } else { %>
    You have selected:
    <UL>
        <% if (orderedSweatshirt.getValue()) { %>
            <LI>1 Sweatshirt (size: <%=
sweatShirtSize.getValue().toUpperCase()
        %>)

            <% } %>

            <% if (orderedTShirt.getValue()) { %>
                <LI>1 Tshirt (size: <%= tshirtSize.getValue().toUpperCase() %>)
            <% } %>

        </UL>
        <% } %>
    <% } else { %>
    Are you sure we can't interest you in something?
    <% } %>

    <CENTER>
        <FORM ACTION="index.jsp" METHOD="GET"
            ENCTYPE="application/x-www-form-urlencoded">
            <INPUT TYPE="IMAGE" SRC="images/shop_again.gif" WIDTH="91"
HEIGHT="30"
                BORDER="0">
        </FORM>
    </CENTER>
    </TD></TR>
</TABLE>

</BODY>

</HTML>

<%
if (orderedSweatshirt.getValue()) {
    currSS.setValue(sweatShirtSize.getValue());
} else {
    currSS.setValue("");
}

if (orderedTShirt.getValue()) {

```

```

        currTS.setValue(tshirtSize.getValue());
    } else {
        currTS.setValue("");
    }
}%>
index.jsp 源代码:
<%-----
    Copyright (c) 1999-2000, Oracle Corporation. All rights reserved.
-----%>
<jsp:useBean id="currSS" scope="session" class="oracle.jsp.jml.JmlString"
/>
<jsp:useBean id="currTS" scope="session" class="oracle.jsp.jml.JmlString"
/>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">
<HTML>

<HEAD>
    <TITLE>untitled</TITLE>
</HEAD>
<BODY BACKGROUND="images/bg.gif" BGCOLOR="#FFFFFF">

<FORM ACTION="cart.jsp" METHOD="POST"
ENCTYPE="application/x-www-form-urlencoded">
<P>
<TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0" WIDTH="100%"
HEIGHT="553">
    <TR>
        <TD WIDTH="33%" HEIGHT="61">&nbsp;</TD>
        <TD WIDTH="67%" HEIGHT="61">&nbsp;</TD>
    </TR>
    <TR>
        <TD WIDTH="33%" HEIGHT="246">&nbsp;</TD>
        <TD WIDTH="67%" HEIGHT="246" VALIGN="TOP" BGCOLOR="#FFFFFF">
            <TABLE BORDER="0" CELLPADDING="0" CELLSPACING="0" WIDTH="81%">
                <TR>
                    <TD WIDTH="100%" BGCOLOR="#CCFFFF">
                        <H4>JSP Wearables
                    </TD>
                </TR>
            </TABLE>
            <TR>
                <TD WIDTH="100%" BGCOLOR="#FFFFFF">

                    <BLOCKQUOTE>
                        Sweatshirt
                        <SPACER TYPE="HORIZONTAL" SIZE="10">($24.95)<BR>
                        <SPACER TYPE="HORIZONTAL" SIZE="30">
                            <INPUT TYPE="RADIO" NAME="SS" VALUE="xl"
                                <%= currSS.getValue().equals("xl") ? "CHECKED" : "" %> >XL
                        <SPACER TYPE="HORIZONTAL" SIZE="10">

```

```

        <INPUT TYPE="RADIO" NAME="SS" VALUE="l" <%= currSS.getValue().equals("l")
            ? "CHECKED" : "" %> >L
    <SPACER TYPE="HORIZONTAL" SIZE="10">
        <INPUT TYPE="RADIO" NAME="SS" VALUE="m" <%= currSS.getValue().equals("m")
            ? "CHECKED" : "" %> >M
    <SPACER TYPE="HORIZONTAL" SIZE="10">
        <INPUT TYPE="RADIO" NAME="SS" VALUE="s" <%= currSS.getValue().equals("s")
            ? "CHECKED" : "" %> >S
    <SPACER TYPE="HORIZONTAL" SIZE="10">
        <INPUT TYPE="RADIO" NAME="SS" VALUE="xs"
            <%= currSS.getValue().equals("xs") ? "CHECKED" : "" %> >XS
    <SPACER TYPE="HORIZONTAL" SIZE="10">
        <INPUT TYPE="RADIO" NAME="SS" VALUE="none"
            <%= currSS.getValue().equals("none") || currSS.isEmpty() ?
                "CHECKED" : "" %> >NONE
    <BR>
    <BR>
    T-Shirt<SPACER TYPE="HORIZONTAL" SIZE="10"> (14.95)<BR>
    <SPACER TYPE="HORIZONTAL" SIZE="30">
        <INPUT TYPE="RADIO" NAME="TS" VALUE="xl"
            <%= currTS.getValue().equals("xl") ? "CHECKED" : "" %> >XL
    <SPACER TYPE="HORIZONTAL" SIZE="10">
        <INPUT TYPE="RADIO" NAME="TS" VALUE="l" <%= currTS.getValue().equals("l")
            ? "CHECKED" : "" %> >L
    <SPACER TYPE="HORIZONTAL" SIZE="10">
        <INPUT TYPE="RADIO" NAME="TS" VALUE="m" <%=
currTS.getValue().equals("m")
            ? "CHECKED" : "" %> >M
    <SPACER TYPE="HORIZONTAL" SIZE="10">
        <INPUT TYPE="RADIO" NAME="TS" VALUE="s" <%= currTS.getValue().equals("s")
            ? "CHECKED" : "" %> >S
    <SPACER TYPE="HORIZONTAL" SIZE="10">
        <INPUT TYPE="RADIO" NAME="TS" VALUE="xs"
            <%= currTS.getValue().equals("xs") ? "CHECKED" : "" %> >XS
        <SPACER TYPE="HORIZONTAL" SIZE="10">
            <INPUT TYPE="RADIO" NAME="TS" VALUE="none"
                <%= currTS.getValue().equals("none") || currTS.isEmpty() ?
                    "CHECKED" : "" %> >NONE
        </BLOCKQUOTE>
    </TD>
</TR>
<TR>
<TD WIDTH="100%">
    <DIV ALIGN="RIGHT">
        <P><INPUT TYPE="IMAGE" SRC="images/addtobkt.gif" WIDTH="103"
HEIGHT="22"
            ALIGN="BOTTOM" BORDER="0">
    </DIV>
</TD>
</TR>

```

```

        </TABLE>
    </TD>
</TR>
</TABLE>

</FORM>

</BODY>

</HTML>

```

9.2 JDBC 实例

本节的实例是用 JDBC 来查询一个数据库，尽管连接缓冲实例是使用 Oracle 的特定连接缓冲来实现的，但大部分还是使用标准的 JDBC 功能。下面是所列举的实例：

- 简单查询对应 SimpleQuery.jsp
- 用户指定查询对应 JDBCQuery.jsp
- 运用 Query Bean 查询对应 UseHtmlQueryBean.jsp
- 连接缓冲对应 ConnCache3.jsp 和 ConnCache1.jsp

9.2.1 简单查询——SimpleQuery.jsp

这个页面执行的是一个关于 scott.emp 表的简单查询，在一个 HTML 表中列出了员工的名字以及它们的薪水情况：

```

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

<!-------
* This is a basic JavaServer Page that does a JDBC query on the
* emp table in schema scott and outputs the result in an html table.
*
-----!>
<HTML>
  <HEAD>
    <TITLE>
      SimpleQuery JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=EOFFFO>
    <H1> Hello
    <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "")
%>
    ! I am SimpleQuery JSP.
  </H1>
  <HR>
  <B> I will do a basic JDBC query to get employee data
    from EMP table in schema SCOTT..
  </B>
  <P>

```

```

<%
    try {
        // Use the following 2 files when running inside Oracle 8i
        // Connection conn = new oracle.jdbc.driver.OracleDriver().
        //                                     defaultConnection ();
        Connection conn =

DriverManager.getConnection((String)session.getValue("connStr"),
                                "scott", "tiger");

        Statement stmt = conn.createStatement ();
        ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                                "FROM scott.emp ORDER BY ename");

        if (rset.next()) {
%>
            <TABLE BORDER=1 BGCOLOR="COC0C0">
            <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
            <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
            <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
                <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
            </TR>
<% while (rset.next()) {
%>

            <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
                <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
            </TR>

<% }
%>

            </TABLE>

<% }
%>
            else {
%>
                <P> Sorry, the query returned no rows! </P>

<%
            }
            rset.close();
            stmt.close();
        } catch (SQLException e) {
            out.println("<P>" + "There was an error doing the query:");
            out.println ("<PRE>" + e + "</PRE> \n <P>");
        }
%>

    </BODY>
</HTML>

```

9.2.2 用户指定查询—JDBCQuery.jsp

这个页面是根据用户指定的条件来查询 `scott.emp` 表，并输入结果：

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

<HTML>
<HEAD> <TITLE> The JDBCQuery JSP </TITLE> </HEAD>
<BODY BGCOLOR=white>

<% String searchCondition = request.getParameter("cond");
   if (searchCondition != null) { %>
       <H3> Search results for : <I> <%= searchCondition %> </I> </H3>
       <%= runQuery(searchCondition) %>
       <HR><BR>
   <% } %>

<B>Enter a search condition:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="cond" SIZE=30>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%!
private String runQuery(String cond) throws SQLException {
    Connection conn = null;
    Statement stmt = null;
    ResultSet rset = null;
    try {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn=DriverManager.getConnection((String)session.getValue("connStr"),
                                         "scott", "tiger");

        stmt = conn.createStatement();
        rset = stmt.executeQuery ("SELECT ename, sal FROM scott.emp "+
                                  (cond.equals("") ? "" : "WHERE " + cond));
        return (formatResult(rset));
    } catch (SQLException e) {
        return ("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    } finally {
        if (rset!= null) rset.close();
        if (stmt!= null) stmt.close();
        if (conn!= null) conn.close();
    }
}

private String formatResult(ResultSet rset) throws SQLException {
    StringBuffer sb = new StringBuffer();
    if (!rset.next())
        sb.append("<P> No matching rows.<P>\n");
    else { sb.append("<UL><B>");
        do { sb.append("<LI>" + rset.getString(1) +
```



```

        " earns $ " + rset.getInt(2) + "</LI>\n");
    } while (rset.next());
    sb.append("</B></UL>");
}
return sb.toString();
}
%>

```

9.2.3 运用 Query Bean 查询——UseHtmlQueryBean.jsp

这个页面根据用户指定条件运用一个名为 HtmlQueryBean 的 JavaBean 来查询 scott.emp 表, HtmlQueryBean 依次运用 HtmlTable 类来定义输出到 HTML 表的格式。实例中包含了 JSP 页面 HtmlQueryBean 和 HtmlTable 的源代码:

UseHtmlQueryBean.jsp 源代码:

```

<jsp:useBean id="htmlQueryBean" class="beans.HtmlQueryBean" scope="session" />
<jsp:setProperty name="htmlQueryBean" property="searchCondition" />

<HTML>
<HEAD> <TITLE> The UseHtmlQueryBean JSP </TITLE> </HEAD>
<BODY BGCOLOR="white">

<% String searchCondition = request.getParameter("searchCondition");
   if (searchCondition != null) { %>
       <H3>Search Results for : <I> <%= searchCondition %> </I> </H3>
       <%= htmlQueryBean.getResult() %>
       <BR> <HR>
   <% } %>

<P><B>Enter a search condition:</B></P>
<FORM METHOD=get>
<INPUT TYPE=text NAME="searchCondition" SIZE=30>
<INPUT TYPE=submit VALUE="Ask Oracle">
</FORM>
</BODY>
</HTML>

```

HtmlQueryBean.java 源代码:

```

package beans;
import java.sql.*;

public class HtmlQueryBean {

    private String searchCondition = "";
    private String connStr = null;

    public String getResult() throws SQLException {
        return runQuery();
    }

    public void setSearchCondition(String searchCondition) {

```

```

        this.searchCondition = searchCondition;
    }

    public void setConnStr(String connStr) {
        this.connStr = connStr;
    }

    private String runQuery() {
        Connection conn = null;
        Statement stmt = null;
        ResultSet rset = null;
        try {
            if (conn == null) {
                DriverManager.registerDriver(new
oracle.jdbc.driver.OracleDriver());
                conn = DriverManager.getConnection(connStr,
                                                    "scott","tiger");
            }
            stmt = conn.createStatement();
            rset = stmt.executeQuery ("SELECT ename as \"Name\", \" +
                                     \"empno as \"Employee Id\", \" +
                                     \"sal as \"Salary\", \" +
                                     \"TO_CHAR(hiredate, 'DD-MON-YYYY') as \"Date Hired\"\"
+
                                     \"FROM scott.emp \" + (searchCondition.equals(\"\") ? \"\" :
                                     \"WHERE \" + searchCondition ));
            return format(rset);
        } catch (SQLException e) {
            return (<P> SQL error: <PRE> \" + e + \" </PRE> </P>\n");
        }
        finally {
            try {
                if (rset != null) rset.close();
                if (stmt != null) stmt.close();
                if (conn != null) conn.close();
            } catch (SQLException ignored) {}
        }
    }

    public static String format(ResultSet rs) throws SQLException {
        StringBuffer sb = new StringBuffer();
        if (rs == null || !rs.next())
            sb.append(<P> No matching rows.<P>\n");
        else {
            sb.append(<TABLE BORDER>\n");
            ResultSetMetaData md = rs.getMetaData();
            int numCols = md.getColumnCount();
            for (int i=1; i<= numCols; i++) {
                sb.append(<TH><I>\" + md.getColumnLabel(i) + </I></TH>");
            }
        }
    }

```

```

        do {
            sb.append("<TR>\n");
            for (int i = 1; i <= numCols; i++) {
                sb.append("<TD>");
                Object obj = rs.getObject(i);
                if (obj != null) sb.append(obj.toString());
                sb.append("</TD>");
            }
            sb.append("</TR>");
        } while (rs.next());
        sb.append("</TABLE>");
    }
    return sb.toString();
}
}

```

HtmlTable.java 源代码:

```

import java.sql.*;

public class HtmlTable {

    public static String format(ResultSet rs) throws SQLException {
        StringBuffer sb = new StringBuffer();
        if (rs == null || !rs.next())
            sb.append("<P> No matching rows.<P>\n");
        else {
            sb.append("<TABLE BORDER>\n");
            ResultSetMetaData md = rs.getMetaData();
            int numCols = md.getColumnCount();
            for (int i=1; i<= numCols; i++) {
                sb.append("<TH><I>" + md.getColumnLabel(i) + "</I></TH>");
            }
            do {
                sb.append("<TR>\n");

                for (int i = 1; i <= numCols; i++) {
                    sb.append("<TD>");
                    Object obj = rs.getObject(i);
                    if (obj != null) sb.append(obj.toString());
                    sb.append("</TD>");
                }
                sb.append("</TR>");
            } while (rs.next());
            sb.append("</TABLE>");
        }
        return sb.toString();
    }
}

```

9.2.4 连接缓冲—ConnCache3.jsp 和 ConnCache1.jsp

本节列举了两个运用 Oracle 缓冲实现连接缓冲的实例，是利用 Oracle JDBC OracleConnectionCacheImpl 类实现的。

实例一，ConnCache3.jsp 完成自身缓冲安装。

实例二，ConnCache1.jsp 运用一个独立的页面 setupcache.jsp 完成安装。

下面是三个页面的程序源代码：

注意：作为一个更加便利的方法，你能运用 OracleJSP 提供的 JavaBean。

ConnCache3.jsp 源代码（有关缓存安装）

这个实例页面完成的是自身连接缓存安装。

```
<%@ page import="java.sql.*, javax.sql.*, oracle.jdbc.pool.*" %>

<!-------
* This is a JavaServer Page that uses Connection Caching at Session
* scope.
-----!>

<jsp:useBean id="ods" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
scope="session" />

<HTML>
  <HEAD>
    <TITLE>
      ConnCache 3 JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=EOFFFO>
    <H1> Hello
      <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "")
%>
    ! I am Connection Caching JSP.
  </H1>
  <HR>
  <B> Session Level Connection Caching.
  </B>

  <P>
  <%
    try {
      ods.setURL((String)session.getValue("connStr"));
      ods.setUser("scott");
      ods.setPassword("tiger");
      Connection conn = ods.getConnection ();
      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
        "FROM scott.emp ORDER BY ename");
```

```

        if (rset.next()) {

%>
        <TABLE BORDER=1 BGCOLOR="C0C0C0">
        <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
        <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
        <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
                <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
        </TR>

<% while (rset.next()) {
%>
        <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
                <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
        </TR>
<% }
%>
        </TABLE>
<% }
else {
%>
        <P> Sorry, the query returned no rows! </P>
<%
    }
    rset.close();
    stmt.close();
    conn.close(); // Put the Connection Back into the Pool

    } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

</BODY>
</HTML>

```

ConnCache1.jsp 和 setupcache.jsp 的源代码

这个实例页面静态包含另一个关于自身连接缓冲安装的页面 setupcache.jsp，两个页面程序源代码如下：

```

<%@ include file="setupcache.jsp" %>
<%@ page import="java.sql.*, javax.sql.*, oracle.jdbc.pool.*" %>

<!-------
* This is a JavaServer Page that uses Connection Caching over application
* scope. The Cache is created in an application scope in setupcache.jsp
* Connection is obtained from the Cache and recycled back once done.
-----!>

```

```

<HTML>
  <HEAD>
    <TITLE>
      ConnCache1 JSP
    </TITLE>
  </HEAD>
<BODY BGCOLOR=EOFFFO>
  <H1> Hello
    <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "")
%>
    ! I am Connection Caching JSP.
  </H1>
  <HR>
  <B> I get the Connection from the Cache and recycle it back.
  </B>

  <P>
  <%
    try {
      Connection conn = cods.getConnection();

      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
                                           "FROM scott.emp ORDER BY ename");
      if (rset.next()) {
%>
        <TABLE BORDER=1 BGCOLOR="C0C0C0">
          <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
          <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
          <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
            <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
          </TR>

          <%      while (rset.next()) {
          %>

            <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
              <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
            </TR>

          <% }
          %>
          </TABLE>
          <% }
          else {
          %>
            <P> Sorry, the query returned no rows! </P>

          <%
            }

```

```

        rset.close();
        stmt.close();
        conn.close(); // Put the Connection Back into the Pool
    } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

</BODY>
</HTML>

setupcache.jsp
<jsp:useBean id="cods" class="oracle.jdbc.pool.OracleConnectionCacheImpl"
    scope="application">
<%
    cods.setURL((String)session.getValue("connStr"));
    cods.setUser("scott");
    cods.setPassword("tiger");
    cods.setStmtCache (5);
%>
</jsp:useBean>

```

9.3 数据库访问 JavaBean 实例

本节列举了运用 Oracle 数据库访问的 JavaBeans 实例，这些 JavaBeans 是由 OracleJSP 提供的，通常也可移植到其他 JSP 环境中。但是连接缓冲要依赖于 Oracle JDBC 连接缓存的实现。

DBBean 由于只支持自身的连接功能和查询而属于这些 JavaBeans 中最简单的、要进行更复杂的操作，需要运用 ConnBean（简单连接）、ConnCacheBean（连接缓冲）、以及 CursorBean（通用 SQL DML 操作）的适当结合。

下面是所包含的实例：

- 运用 DBBean 页面对应 DBBeanDemo.jsp
- 运用 ConnBean 页面对应 ConnBeanDemo.jsp
- 运用 CursorBean 页面对应 CursorBeanDemo.jsp
- 运用 ConnCacheBean 页面对应 ConnCacheBeanDemo.jsp

注意：Oracle 也提供一些关于 SQL 功能后台运用 JavaBeans 的定制标签。有关这些标签的运用实例，请参见“SQL 标签实例”。

9.3.1 运用 DBBean 页面——DBBeanDemo.jsp

这个页面运用一个 DBBean 对象连接到数据库，执行查询，然后以 HTML 表的格式输出结果。

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

```
<!-------
```

```

* This is a basic JavaServer Page that uses a DB Access Bean and queries
* dept and emp tables in schema scott and outputs the result in an html table.
*
-----!>

<jsp:useBean id="dbbean" class="oracle.jsp.dutil.DBBean" scope="session">
  <jsp:setProperty name="dbbean" property="User" value="scott"/>
  <jsp:setProperty name="dbbean" property="Password" value="tiger"/>
  <jsp:setProperty name="dbbean" property="URL" value=
    "<%= (String)session.getValue(\"connStr\") %>" />
</jsp:useBean>

<HTML>
  <HEAD>
    <TITLE>
      DBBeanDemo JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=EOFFFO>
    <H1> Hello
      <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
    ! I am DBBeanDemo JSP.
    </H1>
    <HR>
    <B> I'm using DBBean and querying DEPT & EMP tables in schema SCOTT.....
      I get all employees who work in the Research department.
    </B>

    <P>
    <%
      try {
        String sql_string = " select ENAME from EMP,DEPT " +
          " where DEPT.DNAME = 'RESEARCH' " +
          " and DEPT.DEPTNO = EMP.DEPTNO";

        // Make the Connection
        dbbean.connect();

        // Execute the SQL and get a HTML table
        out.println(dbbean.getResultAsHTMLTable(sql_string));

        // Close the Bean to close the connection
        dbbean.close();
      } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
      }
    %>

  </BODY>

```



```
</HTML>
```

9.3.2 运用 ConnBean 页面——ConnBeanDemo.jsp

这个页面运用一个 ConnBean 对象（一个简单的连接）来重新获得一个 CursorBean 对象，然后运用这个 CursorBean 对象以 HTML 表的格式输出结果。

```
<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>

<!-------
* This is a basic JavaServer Page that uses a Connection Bean and queries
* emp table in schema scott and outputs the result in an html table.
*
-----!>

<jsp:useBean id="cbean" class="oracle.jsp.dbutil.ConnBean"
scope="session">
    <jsp:setProperty name="cbean" property="User" value="scott"/>
    <jsp:setProperty name="cbean" property="Password" value="tiger"/>
    <jsp:setProperty name="cbean" property="URL" value=
        "<%= (String)session.getValue(\"connStr\") %>"/>
    <jsp:setProperty name="cbean" property="PreFetch" value="5"/>
    <jsp:setProperty name="cbean" property="StmtCacheSize" value="2"/>
</jsp:useBean>

<HTML>
    <HEAD>
        <TITLE>
            Connection Bean Demo JSP
        </TITLE>
    </HEAD>
    <BODY BGCOLOR=EOFFFO>
    <H1> Hello
        <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "") %>
        ! I am Connection Bean Demo JSP.
    </H1>
    <HR>
    <B> I'm using connection and a query bean and querying employee names
        and salaries from EMP table in schema SCOTT..
    </B>

    <P>
    <%
        try {

            // Make the Connection
            cbean.connect();
            String sql = "SELECT ename, sal FROM scott.emp ORDER BY ename";

            // get a Cursor Bean
            CursorBean cb = cbean.getCursorBean (CursorBean.PREP_STMT, sql);
```

```

        out.println(cb.getResultAsHTMLTable());

        // Close the cursor bean
        cb.close();
        // Close the Bean to close the connection
        cbean.close();
    } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

</BODY>
</HTML>

```

9.3.3 运用 CursorBean 页面——CursorBeanDemo.jsp

这个页面运用一个 ConnBean 对象（一个简单的连接）和一个 CursorBean 对象来执行一条 PL/SQL 语句，得到 REF CURSOR 后，再把结果翻译成 HTML 表。

```

<%@ page import="java.sql.* , oracle.jsp.dbutil.*" %>

<!-------
* This is a basic JavaServer Page that uses a Cursor and Conn Beans and queries
* dept table in schema scott and outputs the result in an html table.
*
-----!>

<jsp:useBean id="connbean" class="oracle.jsp.dbutil.ConnBean" scope="session">
    <jsp:setProperty name="connbean" property="User" value="scott"/>
    <jsp:setProperty name="connbean" property="Password" value="tiger"/>
    <jsp:setProperty name="connbean" property="URL" value=
        "<%= (String)session.getValue(\"connStr\") %>" />
</jsp:useBean>
<jsp:useBean id="cbean" class="oracle.jsp.dbutil.CursorBean" scope="session">
    <jsp:setProperty name="cbean" property="PreFetch" value="10"/>
    <jsp:setProperty name="cbean" property="ExecuteBatch" value="2"/>
</jsp:useBean>

<HTML>
    <HEAD>
        <TITLE>
            CursorBean Demo JSP
        </TITLE>
    </HEAD>
    <BODY BGCOLOR=EOFFFO>
    <H1> Hello
        <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "")
%>

! I am Cursor Bean JSP.

```

```

</H1>
<HR>
<B> I'm using cbean and i'm quering department names from DEPT table
      in schema SCOTT..
</B>

<P>
<%
    try {

        // Make the Connection
        connbean.connect();

        String sql = "BEGIN OPEN ? FOR SELECT DNAME FROM DEPT; END;";

        // Create a Callable Statement
        cbean.create ( connbean, CursorBean.CALL_STMT, sql);

cbean.registerOutParameter(1,oracle.jdbc.driver.OracleTypes.CURSOR);

        // Execute the PLSQL
        cbean.executeUpdate ();

        // Get the Ref Cursor
        ResultSet rset = cbean.getCursor(1);

        out.println(oracle.jsp.dbutil.BeanUtil.translateToHTMLTable
(rset));

        // Close the RefCursor
        rset.close();

        // Close the Bean
        cbean.close();

        // Close the connection
        connbean.close();

    } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

</BODY>
</HTML>

```

9.3.4 运用 ConnCacheBean 页面——ConnCacheBeanDemo. jsp

这个页面运用一个 ConnCacheBean 对象从一个连接缓冲中获得一个连接，然后运用标

准 JDBC 功能执行查询，结果以 HTML 表表示。

```
<%@ page import="java.sql.*, javax.sql.*, oracle.jsp.dbutil.ConnCacheBean"
%>

<!-------
* This is a basic JavaServer Page that does a JDBC query on the
* emp table in schema scott and outputs the result in an html table.
* Uses Connection Cache Bean.
-----!>

<jsp:useBean id="ccbean" class="oracle.jsp.dbutil.ConnCacheBean"
             scope="session">
  <jsp:setProperty name="ccbean" property="user" value="scott"/>
  <jsp:setProperty name="ccbean" property="password" value="tiger"/>
  <jsp:setProperty name="ccbean" property="URL" value=
    "<%= (String)session.getValue(\"connStr\") %>" />
  <jsp:setProperty name="ccbean" property="MaxLimit" value="5" />
  <jsp:setProperty name="ccbean" property="CacheScheme" value=
    "<%= ConnCacheBean.FIXED_RETURN_NULL_SCHEME %>" />
</jsp:useBean>
<HTML>
  <HEAD>
    <TITLE>
      SimpleQuery JSP
    </TITLE>
  </HEAD>
  <BODY BGCOLOR=EOFFFO>
    <H1> Hello
    <%= (request.getRemoteUser() != null? ", " + request.getRemoteUser() : "")
%>
    ! I am Connection Cache Demo Bean
  </H1>
  <HR>
  <B> I will do a basic JDBC query to get employee data
    from EMP table in schema SCOTT. The connection is obtained from
    the Connection Cache.
  </B>

  <P>
  <%
    try {
      Connection conn = ccbean.getConnection();

      Statement stmt = conn.createStatement ();
      ResultSet rset = stmt.executeQuery ("SELECT ename, sal " +
        "FROM scott.emp ORDER BY ename");
      if (rset.next()) {
%>
        <TABLE BORDER=1 BGCOLOR="C0C0C0">
        <TH WIDTH=200 BGCOLOR="white"> <I>Employee Name</I> </TH>
```

```

        <TH WIDTH=100 BGCOLOR="white"> <I>Salary</I> </TH>
        <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
            <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
        </TR>

<% while (rset.next()) {
%>

        <TR> <TD ALIGN=CENTER> <%= rset.getString(1) %> </TD>
            <TD ALIGN=CENTER> $<%= rset.getDouble(2) %> </TD>
        </TR>

<% }
%>

</TABLE>

<% }
%>
        else {

<%>
            <P> Sorry, the query returned no rows! </P>

<%
        }
        rset.close();
        stmt.close();
        conn.close();
        ccbean.close();
    } catch (SQLException e) {
        out.println("<P>" + "There was an error doing the query:");
        out.println ("<PRE>" + e + "</PRE> \n <P>");
    }
%>

</BODY>
</HTML>

```

9.4 定制标签实例

本节包含下列内容：

- 运用一些 Oracle JSP 标记语言（JML）定制标签的实例
- 推荐一些文档中其他地方运用定制标签的实例

9.4.1 JML 标签实例—hellouser_jml.jsp

本节列举了一个运用 Oracle JML 定制标签的基础实例。这是本章前面关于 hellouser.jsp 实例的修正版本，便于对比，这里把 JML 代码和源代码同时列举出来。

注意关于 JML 标签库运行时的实现可移植到其他 JSP 环境当中，有关运行时间的实现概述，请参照“JSP 标记语言（JML）样例标签库概述”。有关编译时（不可移植）的信息，

请参照附录 C “编译时 JML 标签支持”。

hellouser_jml.jsp 程序代码（运用 JML 标签）

```
<%-----
Copyright (c) 1999, Oracle Corporation. All rights reserved.
-----%>

<%@page session="false" %>
<%@ taglib uri="WEB-INF/jml.tld" prefix="jml" %>

<jml:useForm id="name" param="newName" scope="request" />

<HTML>
<HEAD>
<TITLE>
Hello User
</TITLE>
</HEAD>

<BODY>

<jml:if condition="!name.isEmpty()" >
<H3>Welcome <jml:print eval="name.getValue()" /></H3>
</jml:if>

<P>

Enter your Name:
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>
```

hellouser_jml.jsp 程序代码（不使用 JML 标签）

```
<%-----
Copyright (c) 1999, Oracle Corporation. All rights reserved.
-----%>

<%@page session="false" %>

<jsp:useBean id="name" class="oracle.jsp.jml.JmlString" scope="request" >
    <jsp:setProperty name="name" property="value" param="newName" />
</jsp:useBean>

<HTML>
<HEAD>
```

```

<TITLE>
Hello User
</TITLE>
</HEAD>

<BODY>

<% if (!name.isEmpty()) { %>
<H3>Welcome <%= name.getValue() %></H3>
<% } %>
<P>
Enter your Name:
<FORM METHOD=get>
<INPUT TYPE=TEXT name=newName size = 20><br>
<INPUT TYPE=SUBMIT VALUE="Submit name">
</FORM>

</BODY>
</HTML>

```

9.4.2 用指针来定制其他标签的实例

本文档中其他地方列举的一些其他定制标签的实例：

- 定义一个完整的实例和运用一个标准的 JSP 1.1 兼容的定制标签。
- 运用包含 SQL 功能的 Oracle 定制标签实例。

9.5 特定 Oracle 程序扩展实例

本节列举了许多运用特定 Oracle 程序扩展的实例，主要包括：

- 使用 JspScopeListenre 的页面对应 scope.jsp
- XML 查询对应 XMLQuery.jsp
- SQLJ 查询对应 SQLJSelectInto.sqljsp 和 SQLJIterator.sqljsp

9.5.1 使用 JspScopeListenre 的页面——scope.jsp

这个实例解释了 JspScopeListener 实现的用法，JspScopeListener 允许 JSP 对象所附加的作用域在对象超出作用域范围时予以通知，实例中执行一个以调用超出作用域范围的通知的类属监视程序来记录对象或方法。在运用这个监视程序过程中，scope.jsp 能够模拟页面事件处理器对请求和页面超出作用域的通知事件进行处理。

这个实例创建并且选用一个关于请求和页面范围的监视程序对象，它注册局部通过处理前面超出作用域通知的方法，为解释这一点，实例保留两个计数器，第一个是页面计数器，第二个是包含文件数量的计数器。

当页面超出作用域范围时，当前页面计数器就会记录；当请求超出作用域范围时，包含页面的计数器就会记录，这个例子包括它本身继续执行了 5 次。

这个例子共输出六条信息，显示页面计数器为 1，后面跟着一条简单的信息显示 5

条 jsp:include 操作完成。

监视程序的实现——PageEventDispatcher

PageEventDispatcher 是用来实现 JspCcopeListener 接口的一个 JavaBean。此接口定义了 outOfScope() 事件的方法，把 JspScopeEvent 对象作为输入，当与作用域（应用程序、会话、页面、请求）关联的对象结束时，有关 PageEventDispatcher 对象的 outOfScope() 方法就被调用。

在本例中，PageEventDispatcher 对象充当 JSP 页面的再次调度程序，它允许 JSP 页面在页面与请求事件上与 globals.jsa 在连续功能上等价。JSP 页面为每一个作用域创建一个 PageEventDispatcher 对象，这是因为每一个作用域都需要提供一个事件处理器。当超出作用域范围时，PageEventDispatcher 对象就被通知，它就调用已注册的页面的连续方法。

```
package oracle.jsp.sample.event;

import java.lang.reflect.*;
import oracle.jsp.event.*;

public class PageEventDispatcher extends Object implements JspScopeListener {

    private Object page;
    private String methodName;
    private Method method;

    public PageEventDispatcher() {
    }

    public Object getPage() {
        return page;
    }

    public void setPage(Object page) {
        this.page = page;
    }

    public String getMethodName() {
        return methodName;
    }

    public void setMethodName(String m)
        throws NoSuchMethodException, ClassNotFoundException {
        method = verifyMethod(m);
        methodName = m;
    }

    public void outOfScope(JspScopeEvent ae) {
        int scope = ae.getScope();
```



```

        if ((scope == javax.servlet.jsp.PageContext.REQUEST_SCOPE ||
            scope == javax.servlet.jsp.PageContext.PAGE_SCOPE) &&
            method != null) {
            try {
                Object args[] = {ae.getApplication(), ae.getContainer()};
                method.invoke(page, args);
            } catch (Exception e) {
                // catch all and continue
            }
        }
    }

    private Method verifyMethod(String m)
        throws NoSuchMethodException, ClassNotFoundException {
        if (page == null) throw new NoSuchMethodException
            ("A page hasn't been set yet.");

        /* Don't know whether this is a request or page handler so try one
then
        the other
        */
        Class c = page.getClass();
        Class pTypes[] = {Class.forName("javax.servlet.ServletContext"),
            Class.forName("javax.servlet.jsp.PageContext")};

        try {
            return c.getDeclaredMethod(m, pTypes);
        } catch (NoSuchMethodException nsme) {
            // fall through and try the request signature
        }

        pTypes[1] = Class.forName("javax.servlet.http.HttpServletRequest");
        return c.getDeclaredMethod(m, pTypes);
    }
}

```

scope.jsp 源代码

这个 JSP 页面运用先前的 PageEventDispatcher 类（用来实现 JspScopeListener 接口）来跟踪页面或请求作用域的事件。

```

<!-- declare request and page scoped beans here -->

<jsp:useBean id = "includeCount" class = "oracle.jsp.jml.JmlNumber" scope = "request" />
<jsp:useBean id = "pageCount" class = "oracle.jsp.jml.JmlNumber" scope = "page" >
    <jsp:setProperty name = "pageCount"
        property = "value" value = "<%= pageCount.getValue() + 1 %>" />
</jsp:useBean>
<!-- declare the event dispatchers -->
<jsp:useBean id = "requestDispatcher" class = "oracle.jsp.sample.event.PageEventDispatcher"
    scope = "request" >

```

```

        <jsp:setProperty name = "requestDispatcher" property = "page" value = "<%= this %>"
    />
    <jsp:setProperty name = "requestDispatcher" property = "methodName"
        value = "request_OnEnd" />
</jsp:useBean>

<jsp:useBean id = "pageDispatcher" class = "oracle.jsp.sample.event.PageEventDispatcher"
    scope = "page" >
    <jsp:setProperty name = "pageDispatcher" property = "page" value = "<%= this %>" />
    <jsp:setProperty name = "pageDispatcher" property = "methodName" value = "page_OnEnd"
/>
</jsp:useBean>

<%!
    // request_OnEnd Event Handler
    public void request_OnEnd(ServletContext application, HttpServletRequest request)
    {
        // acquire beans
        oracle.jsp.jml.JmlNumber includeCount =
            (oracle.jsp.jml.JmlNumber) request.getAttribute("includeCount");

        // now cleanup the bean
        if (includeCount != null) application.log
            ("request_OnEnd: Include count = " + includeCount.getValue());
    }

    // page_OnEnd Event Handler
    public void page_OnEnd(ServletContext application, PageContext page) {
        // acquire beans
        oracle.jsp.jml.JmlNumber pageCount =
            (oracle.jsp.jml.JmlNumber) page.getAttribute("pageCount");

        // now cleanup the bean -- uncomment code for real bean
        if (pageCount != null) application.log
            ("page_OnEnd: Page count = " + pageCount.getValue());
    }
%>

<!-- Page implementation goes here -->

<jsp:setProperty name = "includeCount" property = "value"
    value = '<%=
(request.getAttribute("javax.servlet.include.request_uri")
    != null) ? includeCount.getValue() + 1 : 0 %>' />

<h2> Hello World </h2>
Included: <%= request.getAttribute("javax.servlet.include.request_uri") %>
Count: <%= includeCount.getValue() %> <br>

<% if (includeCount.getValue() < 5) { %>

```

```

        <jsp:include page="scope.jsp" flush = "true" />
<% } %>

```

9.5.2 XML 查询——XMLQuery.jsp

这个实例连接到一个数据库,执行一个查询,运用 oracle.xml.sql.query.OracleXMLQuery 类的功能以 XML 字符串格式输出结果。

这是特定 Oracle 的功能,由 Oracle8i 提供的 OracleXMLQuery 类作为 XML-SQL 功能的一部分。有关 XML 和 XSL 在 JSP 页面中的详细用法,请参见“OracleJSP 对 XML 和 XSL 的支持”。

```

<%-----
Copyright (c) 1999, Oracle Corporation. All rights reserved.
-----%>

<%@ page import = "java.sql.*,oracle.xml.sql.query.OracleXMLQuery" %>
<html>
<head><title> The XMLQuery Demo </title></head>
<body>
<h1> XMLQuery Demo </h1>
<h2> Employee List in XML </h2>
<b>(View Page Source in your browser to see XML output)</b>
<% Connection conn = null;
    Statement stmt = null;
    ResultSet rset = null;
    try {

        // determine JDBC driver name from session value
        // if null, use JDBC kprb driver if in JServer, JDBC oci otherwise
        String dbURL = (String)session.getValue("connStr");
        if (dbURL == null)
            dbURL = (System.getProperty("oracle.jserver.version") == null?
                "jdbc:oracle:oci8:@" : "jdbc:oracle:kprb:@");
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        conn = DriverManager.getConnection(dbURL, "scott", "tiger");
        stmt = conn.createStatement ();
        rset = stmt.executeQuery ("SELECT ename, sal " +
                                "FROM scott.emp ORDER BY ename");
        OracleXMLQuery xq = new OracleXMLQuery(conn, rset); %>
        <PRE> <%= xq.getXMLString() %> </PRE>
<% } catch (java.sql.SQLException e) { %>
    <P> SQL error: <PRE> <%= e %> </PRE> </P>
<% } finally {
    if (stmt != null) stmt.close();
    if (rset != null) rset.close();
    if (conn != null) conn.close();
} %>
</body>
</html>

```

9.5.3 SQLJ 查询——SQLJSelectInto.sqljsp 和 SQLJIterator.sqljsp

本节列举了在 JSP 页面中运用 SQLJ 查询一个数据库的实例。

例一 SQLJSelectInto.sqljsp 运用 SQLJ SELECT INTO 语法选择单行。

例二 SQLJIterator.sqljsp 选择多行到 SQLJ 迭代器中，它与 JDBC 结果集比较相似。

要获得有关 SQLJ 在 JSP 页面中运用的信息，请参见“OracleJSP 支持下的 Oracle SQLJ”。

SQLJSelectInto.sqljsp 的源代码（选择单行）

这个例子运用 SQLJ SELECT INTO 语法从数据库中选择单行。

```
<%@ page import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>

<HTML>
<HEAD> <TITLE> The SQLJSelectInto JSP </TITLE> </HEAD>
<BODY BGCOLOR=white>

<%
    String connStr=request.getParameter("connStr");
    if (connStr==null) {
        connStr=(String)session.getValue("connStr");
    } else {
        session.putValue("connStr",connStr);
    }
    if (connStr==null) { %>
<jsp:forward page="../setconn.jsp" />
<%
    }
%>

<%
    String empno = request.getParameter("empno");
    if (empno != null) { %>
        <H3> Employee # <%=empno %> Details: </H3>
        <%= runQuery(connStr,empno) %>
        <HR><BR>
    } %>

<B>Enter an employee number:</B>
<FORM METHOD=get>
<INPUT TYPE="text" NAME="empno" SIZE=10>
<INPUT TYPE="submit" VALUE="Ask Oracle");
</FORM>
</BODY>
</HTML>
<%!
    private String runQuery(String connStr, String empno) throws
java.sql.SQLException {
    DefaultContext dctx = null;
    String ename = null; double sal = 0.0; String hireDate = null;
    StringBuffer sb = new StringBuffer();
```

```

try {
    dctx = Oracle.getConnection(connStr, "scott", "tiger");
    #sql [dctx] { SELECT ename, sal, TO_CHAR(hiredate, 'DD-MON-YYYY')
                    INTO :ename, :sal, :hireDate
                    FROM scott.emp WHERE UPPER(empno) = UPPER(:empno)
    };
    sb.append("<BLOCKQUOTE><BIG><B><PRE>\n");
    sb.append("Name : " + ename + "\n");
    sb.append("Salary : " + sal + "\n");
    sb.append("Date hired : " + hireDate);
    sb.append("</PRE></B></BIG></BLOCKQUOTE>");

    } catch (java.sql.SQLException e) {
        sb.append("<P> SQL error: <PRE> " + e + " </PRE> </P>\n");
    } finally {
        if (dctx!= null) dctx.close();
    }
    return sb.toString();
}
%>

```

SQLJIterator.sqljsp 的源代码（选择多行）

这个例子运用 SQLJ 迭代器从数据库中选择多行。

```

<%@ page import="java.sql.*" %>
<%@ page import="sqlj.runtime.ref.DefaultContext,oracle.sqlj.runtime.Oracle" %>

<!-------
* This is a SQLJ JavaServer Page that does a SQLJ query on the
* emp table in schema scott and outputs the result in an html table.
*
-----!>

<%! #sql iterator Empiter(String ename, double sal, java.sql.Date hiredate) %>

<%
    String connStr=request.getParameter("connStr");
    if (connStr==null) {
        connStr=(String)session.getValue("connStr");
    } else {
        session.putValue("connStr",connStr);
    }
    if (connStr==null) { %>
<jsp:forward page="../setconn.jsp" />
<%
    }
%>

<%
    DefaultContext dctx = null;
    dctx = Oracle.getConnection(connStr, "scott", "tiger");

```

```

%>

<HTML>
<HEAD> <TITLE> The SqljIterator SQLJSP </TITLE> </HEAD>
<BODY BGCOLOR="E0FFF0">
<% String user;
    #sql [dctx] {SELECT user INTO :user FROM dual};
%>

<H1> Hello, <%= user %>! </H1>
<HR>
<B> I will use a SQLJ iterator to get employee data
    from EMP table in schema SCOTT..
</B>
<P>
<%
    Empiter emps;
    try {
        #sql [dctx] emps = { SELECT ename, sal, hiredate
                            FROM scott.emp ORDER BY ename};
        if (emps.next()) {
%>
            <TABLE BORDER=1 BGCOLOR="C0C0C0">
            <TH WIDTH=200 BGCOLOR=white> Employee Name </TH>
            <TH WIDTH=100 BGCOLOR=white> Salary </TH>
            <TR> <TD> <%= emps.ename() %> </TD>
                <TD> <%= emps.sal() %> </TD>
            </TR>

            <% while (emps.next()) {
%>
                <TR> <TD> <%= emps.ename() %> </TD>
                    <TD> <%= emps.sal() %> </TD>
                </TR>
            <% } %>
            </TABLE>
            <% } else { %>
                <P> Sorry, the query returned no rows! </P>
            <% }
                emps.close();
            } catch (SQLException e) { %>
                <P>There was an error doing the query:<PRE> <%= e %> </PRE> <P>
            <% } %>
        </BODY>
    </HTML>

```

9.6 在 Servlet 2.0 环境下运用 globals.jsa 的实例

本节列举的例子是关于如何在 Servlet 2.0 环境下运用 Oracle globals.jsa 的技巧来创建

一个应用程序框架以及基准 application 和基准 session 事件的处理。下面是所列举的实例：

- Application 事件 global.jsa 实例对应 lotto.jsp
- Application 与 Session 事件 global.jsa 实例对应 index1.jsp
- 全局声明 global.jsa 实例对应 index2.jsp

注意：本节中的实例是基于它们在应用程序关闭时的功能，许多服务器不允许手工关闭一个应用程序，在这种情况下，globals.jsa 不能作为应用程序的一个标记工作。然而，你却能够通过升级 lotto.jsp 源代码或者 globals.jsa 文件促使应用程序自动关闭和重新启动（假定 developer_mode=false 时）。（OracleJSP 容器在重新解释和重新载入一个激活的页面之前总是终止一个正在运行的应用程序。）

9.6.1 Application 事件 global.jsa 实例—lotto.jsp

这个例子解释了 OracleJSP globals.jsa 事件在通过 application_onStart 和 application_OnEnd 事件处理器的处理过程。在本例中，每个人在当天中起始的编号被寄存下来，作为结果，对于一幅给定的彩图只有一套编号是某个用户曾经显示的。本例中，用户是靠他们各自的 IP 地址进行身份识别。

为 application_onStart 和 application_OnEnd 写下的代码目的是通过应用程序的关闭仍然使高速缓存中的数据保持持久性。本例是当程序正在终止时将缓存中的数据写到文件中，而当程序被重新启动时从文件中读出数据（假定数据被写入缓存中的同一天服务器被重新启动）。

globals.jsa 文件 lotto.jsp

```
<%@ page import="java.util.*, oracle.jsp.jml.*" %>

<jsp:useBean id = "cachedNumbers" class = "java.util.Hashtable" scope = "application" />

<event:application_OnStart>

<%
    Calendar today = Calendar.getInstance();
    application.setAttribute("today", today);
    try {
        FileInputStream fis = new FileInputStream

(application.getRealPath("/") + File.separator + "lotto.che");
        ObjectInputStream ois = new ObjectInputStream(fis);
        Calendar cacheDay = (Calendar) ois.readObject();
        if (cacheDay.get(Calendar.DAY_OF_YEAR) == today.get(Calendar.DAY_OF_YEAR))
        {
            cachedNumbers = (Hashtable) ois.readObject();
            application.setAttribute("cachedNumbers",
cachedNumbers);
        }
        ois.close();
    } catch (Exception theE) {
```

```

        // catch all -- can't use persistent data
    }
%>

</event:application_OnStart>

<event:application_OnEnd>

<%
    Calendar now = Calendar.getInstance();
    Calendar today = (Calendar) application.getAttribute("today");
    if (cachedNumbers.isEmpty() ||
        now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
        File f = new
File(application.getRealPath("/") + File.separator + "lotto.che");
        if (f.exists()) f.delete();
        return;
    }

    try {
        FileOutputStream fos = new FileOutputStream
            (application.getRealPath("/") + File.separator + "lotto.che");
        ObjectOutputStream oos = new ObjectOutputStream(fos);
        oos.writeObject(today);
        oos.writeObject(cachedNumbers);
        oos.close();
    } catch (Exception theE) {
        // catch all -- can't use persistent data
    }
%>

```

lott1.jsp 源代码

```

<%@ page session = "false" %>
<jsp:useBean id = "picker" class = "oracle.jsp.sample.lottery.LottoPicker" scope = "page" />

<HTML>
<HEAD><TITLE>Lotto Number Generator</TITLE></HEAD>
<BODY BACKGROUND="images/cream.jpg" BGCOLOR="#FFFFFF">
<H1 ALIGN="CENTER"></H1>

<BR>

<!-- <H1 ALIGN="CENTER"> IP: <%= request.getRemoteAddr() %> <BR> -->
<H1 ALIGN="CENTER">Your Specially Picked</H1>
<P ALIGN="CENTER"><IMG SRC="images/winningnumbers.gif" WIDTH="450" HEIGHT="69"
ALIGN="BOTTOM"
BORDER="0"></P>
<P>

<P ALIGN="CENTER">
<TABLE ALIGN="CENTER" BORDER="0" CELLPADDING="0" CELLSPACING="0">

```



```

<TR>
<%

    int[] picks;
    String identity = request.getRemoteAddr();

    // Make sure its not tomorrow
    Calendar now = Calendar.getInstance();
    Calendar today = (Calendar) application.getAttribute("today");
    if (now.get(Calendar.DAY_OF_YEAR) > today.get(Calendar.DAY_OF_YEAR)) {
        System.out.println("New day....");
        cachedNumbers.clear();
        today = now;
        application.setAttribute("today", today);
    }

    synchronized (cachedNumbers) {
        if ((picks = (int []) cachedNumbers.get(identity)) == null) {
            picks = picker.getPicks();
            cachedNumbers.put(identity, picks);
        }
        for (int i = 0; i < picks.length; i++) {
%>
            <TD>
            <IMG SRC="images/ball<%= picks[i] %>.gif" WIDTH="68" HEIGHT="76"
                ALIGN="BOTTOM" BORDER="0">
            </TD>

        <%
        }

    %>
</TR>
</TABLE>

</P>

<P ALIGN="CENTER"><BR>
<BR>
<IMG SRC="images/playrespon.gif" WIDTH="120" HEIGHT="73" ALIGN="BOTTOM"
BORDER="0">

</BODY>
</HTML>

```

9.6.2 Application 与 Session 事件 global.jsa 实例——index1.jsp

本例是运用一个 global.jsa 文件来处理 application 与 session 的生命周期事件，它统计激活的会话数，会话的总数以及应用程序页面被点击次数的总数。这些数值中的每一个都

被维持在应用程序作用域中，每当有一个请求应用程序页面（index1.jsp）就刷新一次页面点击数。这个 globals.jsa session_OnStart 事件处理器同时把激活的会话数和总的会话数都进行递增，而 globals.jsa session_OnEnd 事件处理器对于激活的会话数按 1 递增。

这个页面输出比较简单，当有一个新会话启动时，会话计数器就输出，每当有一个请求时，页面计数器就输出。每一个数值的最后一个数就记录在 globals.jsa application_OnEnd 事件处理器中。

请注意下面的例子：

- 随着这些数值被维持在应用程序作用域中，当计数器变量改变时，存取必须是同步进行。
- 计数器中的数值要使用 OracleJSP oracle.jsp.jml.JmlNumber 的扩展数据类型，这是一个内嵌的 bean，它简化了数据值在应用程序作用域中的用法。（有关 JML 扩展数据类型的信息，请参见“JML 扩展数据类型”。）

globals.jsa 文件 index1.jsp

```
<%@ taglib uri="oracle.jsp.parse.OpenJspRegisterLib" prefix="jml" %>

<event:application_OnStart>

    <!-- Initializes counts to zero -->
    <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope =
"application" />
    <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope
= "application" />
    <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber"
scope = "application" />
    <!-- Consider storing pageCount persistently -- If you do read it here
-->

</event:application_OnStart>

<event:application_OnEnd>

    <!-- Acquire beans -->
    <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope =
"application" />
    <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope
= "application" />
    <% application.log("The number of page hits were: " +
pageCount.getValue() ); %>
    <% application.log("The number of client sessions were: " +
sessionCount.getValue() ); %>
    <!-- Consider storing pageCount persistently -- If you do write it here
-->

</event:application_OnEnd>

<event:session_OnStart>
```

```

        <%-- Acquire beans --%>
        <jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope =
"application" />
        <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope =
"application" />

        <%
            synchronized (sessionCount) {
                sessionCount.setValue(sessionCount.getValue() + 1);
            }
            <br>
            Starting session #: <%= sessionCount.getValue() %> <br>
        <%
        }
        %>

        <%
            synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() + 1);
            }
            There are currently <b> <%= activeSessions.getValue() %> </b> active
            sessions <p>
        <%
        }
        %>

</event:session_OnStart>

<event:session_OnEnd>

        <%-- Acquire beans --%>
        <jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope =
"application" />

        <%
            synchronized (activeSessions) {
                activeSessions.setValue(activeSessions.getValue() - 1);
            }
        %>

</event:session_OnEnd>

index1.jsp 源代码
        <%-- Acquire beans --%>
        <jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope =
"application" />

        <%
            synchronized (pageCount) {
                pageCount.setValue(pageCount.getValue() + 1);
            }
        %>

```

```
    }
    %>
```

```
This page has been accessed <b> <%= pageCount.getValue() %> </b> times.
<p>
```

9.6.3 全局声明 global.jsa 实例——index2.jsp

这个例子运用一个 global.jsa 文件来声明全局变量，它是基于“Application 与 Session 事件 global.jsa 实例——index1.jsp”实例中的事件处理器，但是不同于那三个应用程序计数器变量。（在源事件处理器实例中，为进行对比，每一个事件处理器和 JSP 页面本身必须提供 jsp:useBean 语句在局部声明它们正在访问的 bean。）

在所有的事件处理器和 JSP 页面中声明全局的 bean 结果导致隐含声明。

globals.jsa 文件 index2.jsp

```
<!-- globally declares variables and initializes them to zero --%>

<jsp:useBean id="pageCount" class="oracle.jsp.jml.JmlNumber" scope =
"application" />
<jsp:useBean id="sessionCount" class="oracle.jsp.jml.JmlNumber" scope =
"application" />
<jsp:useBean id="activeSessions" class="oracle.jsp.jml.JmlNumber" scope =
"application" />

<event:application_OnStart>

    <!-- Consider storing pageCount persistently -- If you do read it here --%>

</event:application_OnStart>

<event:application_OnEnd>

    <% application.log("The number of page hits were: " + pageCount.getValue() ); %>
    <% application.log("The number of client sessions were: " +
sessionCount.getValue() ); %>

    <!-- Consider storing pageCount persistently -- If you do write it here --%>

</event:application_OnEnd>

<event:session_OnStart>

    <%
        synchronized (sessionCount) {
            sessionCount.setValue(sessionCount.getValue() + 1);
        %>

        <br>
        Starting session #: <%= sessionCount.getValue() %> <br>

    <%
        }
    %>
    <%
```

```

        synchronized (activeSessions) {
            activeSessions.setValue(activeSessions.getValue() + 1);
        }
    }
    There are currently <b> <%= activeSessions.getValue() %> </b>
active sessions <p>
    <%
    }
    %>

</event:session_OnStart>

<event:session_OnEnd>

    <%
    synchronized (activeSessions) {
        activeSessions.setValue(activeSessions.getValue() - 1);
    }
    %>

</event:session_OnEnd>

index2.jsp 源代码
<!-- pageCount declared in globals.jsa so active in all pages -->

<%
    synchronized(pageCount) {
        pageCount.setValue(pageCount.getValue() + 1);
    }
    %>

This page has been accessed <b> <%= pageCount.getValue() %> </b> times.

<p>

```

附录 A 安装与配置

本附录提供了安装 OracleJSP、配置 Web Server 运行 OracleJSP 以及配置 OracleJSP 的必要信息，这些技术信息主要适用于现在比较通用的 Web Server 和 Servlet 环境，包括 Apache/JServ，Sun 公司的 JavaServer Web Developer's Kit(JSWDK)以及 Tomcat（Apache 与 Sun 公司合作开发）。对 Oracle 环境下的安装与配置，请参考相应产品的安装和配置说明文档。

对 Oracle Servlet Engine（OSE）环境来说，解释时的配置是通过 OracleJSP 的预解释工具的不同选项来处理的，详细信息请参见“预解释工具 ojspc”。

本附录涵盖了以下主题：

- 系统需求
- OracleJSP 的安装和 Web Server 的配置
- OracleJSP 的配置

A.1 系统需求

OracleJSP 是一个纯 Java 环境，要想正确安装它，系统必须满足以下最小需求：

操作系统：	凡支持 JDK 1.1.x 或更高版本的操作系统均可
Java 版本：	JDK 1.1.x 或 1.2.x（Oracle 建议你的平台使用最新的 JDK 版本，最好是 1.1.8 或者更高版本。）
Java 编译器：	使用 JDK 提供的标准的编译器 <code>javac</code> 即可（当然，你也可以使用别的 Java 编译器。）
Web Server：	任何支持服务器端小程序（Servlet）的 Web Server
Servlet 环境：	任何实现了 Servlet 2.0 或更高版本的指令的 Servlet 环境

注意：Servlet 环境的函数库必须安装在你的系统里，并且在 Web Server 的配置文件的类路径里必须包含这些库的路径信息。这些库包含了 `javax.servlet.*` 这个包。

例如，在 Apache/JServ 环境下（包括 Oracle Internet Application Server 环境），为了正确运行 servlet 程序，你必须有 `jsdk.jar` 这个文件，此文件是由 Sun 公司的 JSDK 2.0 开发包提供的；在 Sun 公司的 JSWDK 环境下，为了正确运行 servlet 程序，你必须有 `servlet.jar`（servlet 2.1 版本）这个文件，此文件由 JSWDK1.0 开发包提供；在 Tomcat 环境下，为了正确运行 servlet 程序，你必须有 `servlet.jar`（servlet 2.2 版本）这个文件，此文件由 Tomcat 3.1 提供。需要注意的是，不要把 JSDK（Java 小程序开发工具包 Java Servlet Developer's Kit）和 JSWDK（Java 服务器端网络开发工具包 JavaServer Web Developer's Kit）这两个术语混淆了。

A.2 OracleJSP 的安装和 Web Server 的配置

本节讨论在各种各样的 JSP 环境下如何安装 OracleJSP 和配置相应的 Web Server 选项，涉及到的 JSP 环境主要有如下几种：

- Apache/JServ
- Sun 公司的 JSWDK
- Tomcat
- Oracle Servlet Engine (OSE)
- 其他的 Oracle 环境

本节所讨论的内容假设你的目标平台要么是个开发环境，要么是个发行环境，并且至少满足在“系统需求”中所提到的最小需求。还有，至少你已经在这个平台上验证过了它能干以下事情：

- 运行 Java
- 运行 Java 编译器（一般是标准的 javac）
- 运行基于 HTTP 的 Java 小程序

注意：

- 本节的例子主要是针对 Unix 环境的，但是基本的信息（诸如目录名和文件名）也适用于其他的环境。
- Web Server 的配置信息主要是针对流行的非 Oracle 环境，比如 Apache/JServ 等。对 Oracle 环境，请参考特定产品的相关文档（如 Oracle Internet Application Server 或者 Web-to-go 等）。

A.2.1 OracleJSP 必需的文件和可选的文件

本小节总结了运行 OracleJSP 所必需的 JAR 和 ZIP 文件，以及为了使用 Oracle JDBC 和 SQLJ 函数，JML 或者 SQL 的定制标签，以及定制的数据访问 JavaBeans 组件等可选功能而需要的 JAR 和 ZIP 文件。接下来讨论了如何在非 Oracle 环境下安装 OracleJSP 的文件，对于已经提供了 OracleJSP 支持的 Oracle 环境，也一并作了讨论。

运行 OracleJSP 所必需的文件一定要加进系统配置文件的类路径里。

文件小结

下面所列文件是 OracleJSP 所提供的文件，它们必须安装在系统里面：

- ojsp.jar（OracleJSP 的主文件）
- xmlparserv2.jar（用来解析 XML，web.xml 发行描述符和任何标签库描述符都需要这个文件）
- servlet.jar（标准的 Servlet 库，Servlet 2.2 版本）

另外，如果 JSP 页面要使用 Oracle 的 JSP 标记语言（JSP Markup Language，简称 JML）标签、SQL 标签或者是定制的数据访问 JavaBeans 组件，你还需要下面几个文件：

- ojsputil.jar

- xsu12.jar (JDK 1.2.x) 或者 xsu11.jar (JDK 1.1.x) (在 OSE 环境下使用, 或者在客户端使用 XML 功能时使用)

要在 OSE 环境下运行 OracleJSP, xsu12.jar 或者 xsu11.jar, 必须在 ojsputil.jar 之前安装或者同时安装。(Oracle8i 在标准安装的时候, 会自动处理这个问题。)然而, 在客户端环境下运行的时候, 只是在数据访问 JavaBeans 组件当中使用 XML 功能时才需要 xsu12.jar 或者 xsu11.jar 文件 (比如得到一个结果并赋值给一个 XML 字符串)。Oracle8i 8.1.7 发行版中包含有 xsu12.jar 和 xsu11.jar 这两个文件。

Servlet 库注解: OracleJSP 需要并且提供了 2.2 版本的 Servlet 库, 它包含标准的 javax.servlet.* 这个包。然而你的 Web Server 环境可能需要并提供了不同版本的 Servlet 库, 这时在配置类路径的时候就必须注意, 一定要先写 Web Server 所使用的 Servlet 的路径, 然后再写 OracleJSP 使用的 Servlet 的路径。

表 A-1 总结了 Servlet 库的各个版本, 需要注意的是, 不要把 JSWDK (Java 服务器端网络开发工具包 JavaServer Web Developer's Kit) 和 JSDK (Java 小程序开发工具包 Java Servlet Developer's Kit) 这两个术语混淆了。

表 A-1 Servlet 库的各个版本

Servlet 库版本	库文件名	提供者
servlet 2.2	servlet.jar	OracleJSP, Tomcat 3.1
servlet 2.1	servlet.jar	Sun JSWDK 1.0
servlet 2.0	jsdk.jar	Sun JSDK 2.0, Apache/JServ

(对 Apache/JServ 环境, jsdk.jar 必须单独下载。)

本节的剩余部分将讨论只有在使用某些可选的或者扩展的功能时才需要使用到的文件。

JDBC 所需要的文件 (可选): 下面是在使用 Oracle JDBC 的时候需要的文件。(你要同时意识到, SQLJ 也使用了 Oracle JDBC。)

- classes12.zip (在 JDK1.2.x 环境下使用)
- 或者
- classes11.zip (在 JDK1.1.x 环境下使用)

SQLJ 所需要的文件 (可选): 下面是 JSP 页面使用 Oracle SQLJ 来访问 Oracle8i 的时候需要的文件。

- translator.zip (SQLJ 的解释器, 在 JDK1.2.x 或者 1.1.x 环境下使用)
- 以及合适的 SQLJ 运行时文件。
- runtime12.zip (在 JDK1.2.x 和 Oracle JDBC 8.1.7 环境下使用)
- 或者
- runtime12ee.zip (在 JDK1.2.x 企业版和 Oracle JDBC 8.1.7 环境下使用)
- 或者
- runtime11.zip (在 JDK1.1.x 和 Oracle JDBC 8.1.7 环境下使用)
- 或者

- runtime.zip（通用版：适用于 JDK1.2.x 或 1.1.x 以及任意版本的 Oracle JDBC）
（JDK 1.2.x 企业版提供了数据源支持，并且与 SQLJ ISO 标准相兼容。）

非 Oracle 环境下的安装

为了在诸如 Apache/JServ, Sun 公司的 JSWDK 或者 Tomcat 这些非 Oracle 环境中运行 OracleJSP, 你必须从 Oracle 技术网络 (Oracle Technology Network, 简称 OTN) 上下载 OracleJSP, OTN 的网址如下:

<http://technet.oracle.com/tech/java/servlets/index.htm>

在页面上方的按钮条上点击 “Software” 按钮, 这时你需要一个 OTN 的账户, 不过注册成为会员是免费的。如果你还没有一个账户, 那么在页面上方的按钮条上点击 “Membership” 按钮, 免费注册一个账户。

从 OTN 可以下载到 ojsp.zip, 这个文件包含了本节中提到的运行 OracleJSP 所需要的文件以及将在后面讨论的配置文件, 还包括发行声明、文档文件和实例程序。

OracleJSP 的安装信息和配置指令也被包括进了 ojsp.zip 文件中, 你可以参看 install.htm 文件以得到顶层的信息和链接。不过, 你可以从本附录得到更详细的信息, 学会如何配置当前流行的非 Oracle 的 Web Server 环境——诸如 Apache/JServ, Sun 公司的 JSWDK, Tomcat 等, 使它们可以使用 OracleJSP。

只要愿意, 你可以选择任何根目录去存放 OracleJSP, 只要你选择的路径在 Web Server 的类路径里正确设置 (这方面的内容在 “在 Web Server 的类路径里添加与 OracleJSP 相关的 JAR 和 ZIP 文件” 将会详细讨论)。

Oracle JDBC 和 SQLJ 也可以从 OTN 上单独下载, 网址如下:

<http://technet.oracle.com/tech/java/sqlj-jdbc/index.htm>

在页面上方的按钮条上点击 “Software” 按钮以下载这两个软件。

注意: Oracle Internet Application Server 使用 Apache/JServ 环境, 但是需要注意, 你应该使用本附录中所讲的 Application Server 的安装和配置指令, 而不能使用 Apache/JServ 的安装和配置指令。

提供 OracleJSP 的 Oracle 环境

下面这些 Oracle 环境提供 OracleJSP 并且还提供 Web Server 或者 Web Listener:

- Oracle Servlet Engine(OSE) 8.1.7
- Oracle Internet Application Server 1.0.0
- Oracle Application Server 4.0.8.2
- Oracle Web-to-go 1.3 (可以在 Oracle8i Lite 版中使用)
- Oracle JDeveloper 3.0

在上面这些环境当中, 每一个产品的安装都包含了 OracleJSP 组件。

如果你的目标平台是 OSE, 那么你还需要一个客户端的开发和测试环境——也许是 Oracle JDeveloper, 也许是非 Oracle 的开发工具。当你在开发环境中完成了初步的开发和调试工作后, 你就可以将 JSP 页面发行到 Oracle8i 数据库中, 有关这部分的详细内容请参看第六章。

A. 2. 2 配置 Web Server 和 Servlet 环境运行 OracleJSP

为了正常运行 OracleJSP，你可以按照下面的步骤来配置 Web Server：

1. 把与 OracleJSP 相关的 JAR 和 ZIP 文件添加到 Web Server 的类路径中。
2. 配置 Web Server，将 JSP 扩展名（.jsp，.JSP 以及可选的.sqljsp 和.SQLJSP）与 Oracle JspServlet 关联起来，JspServlet 是 OracleJSP 的一个前端工具。

上述步骤适用于任何 Web Server 环境，但是这些信息主要是针对当前最流行的非 Oracle 环境——Apache/JServ，Sun 公司和 JSWDK 以及 Tomcat。

Oracle8i JServer 环境所提供的 Oracle Servlet Engine (OSE)在安装过程中会自动配置正常运行 OracleJSP 所需的所有选项。对于其他的 Oracle 环境，因为产品很多，请参考相应产品的说明文档（大部分安装和配置过程都是自动的）。

在 Web Server 的类路径里添加与 OracleJSP 相关的 JAR 和 ZIP 文件

你必须更新 Web Server 的类路径，将 OracleJSP 运行所需要的 JAR 和 ZIP 文件按正确的次序添加到搜索路径中（需要特别注意的一点是在添加 Servlet 2.2 版本的 Servlet.jsp 时的顺序问题，这将在下面详细讨论）。需要添加的文件如下：

- Ojsp.jar
- xmlparservzz.jar
- servlet.jar(Servlet 2.2 版本)
(注：OracleJSP 所提供的 Servlet.jar 和 Tomcat 3.1 提供的 Servlet.jar 是完全相同的)
- ojsputil.jar（可选的，提供 JML 标签，SQL 标签以及数据访问 JavaBeans 组件）
- xsul2.jar（适用于 JDK1.2.x 版本）或者 xsul11.jar（适用于 JDK1.1.X）（可选的，提供 JML 标签，SQL 标签和数据库 JavaBeans 组件）。
- 其他的适用于 JDBC 和 SQLJ 的可选的 ZIP 和 JSP 文件。

注意：除了上面所提到的以外，你还必须确保 javac（或其他使用 javacmd 配置参数设定的 Java 编译器）在 OracleJSP 的搜索路径下。对 javac 来说，在 JDK1.1.x 环境下，JDK 和 classes.zip 文件必须在 Web Server 的类搜索路径下；在 JDK1.2.x 环境下，JDK 的 tools.jar 文件必须在 Web Server 类的搜索路径下。

Apache/JServ 环境：在 Apache/JServ 环境下，首先找到 JServ conf 目录下的 jserv.properties 文件，然后在里面添加合适的 wrapper.classpath 命令，需注意 jsdk.jar 文件应该始终在路径下，它是由 Sun 公司的 JSDK 2.0 提供的，主要功能是提供 javax.Servlet.*这个包，而这个包是 Apache/JServ 所必需的。另外，JDK 的所有文件也必须在类的搜索路径下。

下面的例子（使用的是 UNIX 环境下的路径）解释了如何将 OracleJSP、JDBC 和 SQLJ 添加到类路径（将例子中的[Oracle_Home]替换成自己的 Oracle 安装目录）。

```
# servlet 2.0 APIs (required by Apache/JServ, from Sun JSDK 2.0):
wrapper.classpath=jsdk2.0/lib/jsdk.jar
#
# servlet 2.2 APIs (required and provided by OracleJSP):
wrapper.classpath=[Oracle_Home]/ojsp/lib/servlet.jar
```

```
# OracleJSP packages:
wrapper.classpath=[Oracle_Home]/ojsp/lib/ojsp.jar
wrapper.classpath=[Oracle_Home]/ojsp/lib/ojsputil.jar
# XML parser (used for servlet 2.2 web deployment descriptor):
wrapper.classpath=[Oracle_Home]/ojsp/lib/xmlparserv2.jar
# JDBC libraries for Oracle database access (JDK 1.2.x environment):
wrapper.classpath=[Oracle_Home]/ojsp/lib/classes12.zip
# SQLJ translator (optional):
wrapper.classpath=[Oracle_Home]/ojsp/lib/translator.zip
# SQLJ runtime (optional) (for JDK 1.2.x enterprise edition):
wrapper.classpath=[Oracle_Home]/ojsp/lib/runtime12.zip
```

注意：在 Apache/JServ 环境下，类路径 jsdk.jar 的出现顺序必须早于 Servlet.jar 的出现顺序。

现在考虑下面的例子，其中使用到了 UseBean 命令：

```
<jsp:useBean id="queryBean" class="mybeans.JDBCQueryBean" scope="session"
/>
```

你可以向 jserv.properties 文件中添加如下的 wrapper.classpath 命令（本例子使用的是 Windows NT 是环境）。

```
wrapper.classpath=D:\Apache\Apache1.3.9\beans\
```

设置好以后，系统就会在以下路径中寻找 JDBCQuery.class 这个类了。

```
D:\Apache\Apache1.3.9\beans\mybeans\JDBCQueryBean.class
```

JSWDK 环境：在 JSWDK 环境下，首先找到 jswdk-1.0 根目录下的 startserver 脚本，然后将 OracleJSP 的文件路径添加到环境变量 jspJars 中。在添加时需将文件路径追加到 jspJars 的环境变量所列出的最后一个 .jar 文件之后，并且需要使用自己的操作系统所需求的正确的目录语法和分隔字符（UNIX 操作系统下是冒号（:），在 Windows 操作系统下是分号（;））。下面是一个实例：

```
jspJars=./lib/jspengine.jar:./lib/ojsp.jar:./lib/ojsputil.jar
```

这个例子（使用 UNIX 语法）假设所需的 JAR 文件在 jswdk-1.0 根目录下的 lib 子目录中。

与上面类似，你可以修改 startserver 脚本，将其他可选的 JAR 文件添加到 miscJars 环境变量中，如下例所示：

```
miscJars=./lib/xml.jar:./lib/xmlparserv2.jar:./lib/servlet.jar
```

这个例子（使用 UNIX 语法）也假设所需的 JAR 文件在 jswdk-1.0 根目录下的 lib 子目录中。

注意：在 JSWDK 环境下，类路径里 2.1 版本的 servlet.jar（由 Sun JSWDK 1.0 提供）的出现顺序必须早于 2.2 版本的 Servlet.Jar（由 OracleJSP 提供）的出现顺序。一般情况下，Servlet 2.1 版本的文件路径是由环境变量 jsdkJars 来设置的。整体的类路径的设置是由若干条 xxxJars 的环境变量的设置来组合形成的，这些环境变量包括：jsdkJars、jspJars 和 miscJars。要注意查看 startserver 脚本以确保 miscJars 命令出现在 jsdkJars 之后。

Tomcat 环境：在 Tomcat 环境下，将文件添加到类路径的过程中与其他 Servlet 环境

相比更依赖于操作系统。

对 UNIX 操作系统来说，将 OracleJSP 的 JAR 和 ZIP 文件复制到你的 [TOMCAT_HOME]/lib 目录即可。这个目录会自动地被添加到 Tomcat 的类路径中去。

对 Windows NT 操作系统来说，需要修改 [TOMCAT_HOME]\bin 目录下的 tomcat.bat 文件，单独地将每个 OracleJSP 文件添加到 CLASSPATH 环境变量中。下面的例子假设你已经将 OracleJSP 的文件复制到了 [TOMCAT_HOME]\lib 目录下：

```
set CLASSPATH=%CLASSPATH%;%TOMCAT_HOME%\lib\ojjsp.jar;%TOMCAT_HOME%\lib\ojsputil.jar
```

Tomcat 已经包括了 2.2 版本的 Servlet.jar 文件（与 OracleJSP 所提供的变量相同，因此不需要考虑 Servlet.jar 的版本问题。

在 JSP 扩展名与 OracleJSP 之间建立关联

你必须配置 Web Server，使得它能够：

- 辨认出哪些文件是 JSP 页面，必须映射后缀名为 .jsp 和 .JSP 的文件。为了能使用 Oracle SQLJ，也要映射 .sqljsp 和 .SQLJSP 文件。
- 找到并运行 OracleJSP 中处理 JSP 页面的程序 oracle.jsp.JspServlet，你可以认为这个程序是 OracleJSP 的前端工具。

注意：通过上面的配置，OracleJSP 就会支持页面间的引用，引用可以通过 .jsp 文件扩展名来实现，也可以通过 .JSP 文件扩展名来实现，但是需要注意：在引用的时候一定要注意大小写。如果文件名是 file.jsp，你可以使用 file.jsp 来正确地引用它，但是使用 file.jsp 就会导致错误的结果。如果文件名是 file.JSP，你可以使用 file.JSP 来引用它，但是不能使用 file.jsp（对 .sqljsp 和 .SQLJSP 来说，以上规则也同样使用）。

Apache/JSer 环境：在 Apache/Jser 环境下，在 JSP 文件的扩展名和 Oracle JspServlet 之间建立关联只需要一个步骤。找到 JServ conf 目录下的配置文件 jserv.conf 或者 mod_jserv.conf，在文件中添加 ApJServAction 命令就可执行文件关联操作。

（在旧版本中，你必须修改 Apache conf 目录下的 httpd.conf 文件，而在新版本中，httpd.conf 在执行过程中已经包括了 jserv.conf 或者 mod_jserv.conf 文件——请查看 httpd.conf 文件以了解哪些文件被包括在里面。）

下面是一个实例（使用 UNIX 语法）：

```
# Map file name extensions (.sqljsp and .SQLJSP are optional).
ApJServAction .jsp /servlets/oracle.jsp.JspServlet
ApJServAction .JSP /servlets/oracle.jsp.JspServlet
ApJServAction .sqljsp /servlets/oracle.jsp.JspServlet
ApJServAction .SQLJSP /servlets/oracle.jsp.JspServlet
```

oracle.jsp.JspServlet 的路径并不是和文件系统中的目录路径完全相同的，存储区依赖于 Apache/Jser 的 Servlet 配置——即 Servlet 存储区是如何被映射的，存储区属性文件的名字以及 Servlet 区实际所在的文件系统目录。（“Servlet 存储区”是 Apache/Jser 的一个术语，它的含义与“Servlet”环境相同），请参考 Apache/Jser 文档以获得更多信息。

JSWDK 环境：在 JSWDK 环境下，在 JSP 文件扩展名与 Oracle JspServlet 之间建立关

联需要两个步骤:

第一步需要修改 WEB-INF 目录下的 mappings.properties 文件,将每个 servlet 环境都定义成 JSP 文件的扩展名。下面是一个实例:

```
# Map JSP file name extensions (.sqljsp and .SQLJSP are optional).
.jsp=jsp
.JSP=jsp
.sqljsp=jsp
.SQLJSP=jsp
```

第二步需要修改 WEB-INF 目录下的 servlet.properties 文件,将每个 servlet 环境都定义成 Oracle JspServlet, 用来处理 JSP 页面内容,并且要确保注释掉以前为了实现 JSP 引用而定义的关联,如下例所示:

```
#jsp.code=com.sun.jsp.runtime.JspServlet (replacing this with Oracle)
jsp.code=oracle.jsp.JspServlet
```

Tomcat 环境: 在 Tomcat 环境下,在 JSP 文件扩展名 Oracle JspServlet 之间建立关联只需一个步骤。修改 Web.xml 文件中的 servletmapping 一小节中的内容如下所示。

注意: 在[TOMCAT_HOME]/conf 目录下,有一个全局的 web.xml 文件,如果你想要在特定的应用中重载这些设定,只需修改特定应用程序根目录下的 WEB-INF 子目录中的 web.xml 文件即可。

```
# Map file name extensions (.sqljsp and .SQLJSP are optional).
```

```
<servlet-mapping>
  <servlet-name>
    oracle.jsp.JspServlet
  </servlet-name>
  <url-pattern>
    *.jsp
  </url-pattern>
</servlet-mapping>
```

```
<servlet-mapping>
  <servlet-name>
    oracle.jsp.JspServlet
  </servlet-name>
  <url-pattern>
    *.JSP
  </url-pattern>
</servlet-mapping>
```

```
<servlet-mapping>
  <servlet-name>
    oracle.jsp.JspServlet
  </servlet-name>
  <url-pattern>
    *.sqljsp
  </url-pattern>
</servlet-mapping>
```

```

<servlet-mapping>
  <servlet-name>
    oracle.jsp.JspServlet
  </servlet-name>
  <url-pattern>
    *.SQLJSP
  </url-pattern>
</servlet-mapping>

```

你也可以为 Oracle.jsp.JspServlet 设置一个别名，如下所示：

```

<servlet>
  <servlet-name>
    ojsp
  </servlet-name>
  <servlet-class>
    oracle.jsp.JspServlet
  </servlet-class>
  ...
</servlet>

```

设置完别名后，你就可以在其他地方的设置中使用“ojsp”来代替 oracle.jsp.JspServlet，如下所示：

```

<servlet-mapping>
  <servlet-name>
    ojsp
  </servlet-name>
  <url-pattern>
    *.jsp
  </url-pattern>
</servlet-mapping>

```

A.3 OracleJSP 的配置

OracleJSP 的前端工具 JspServlet 支持很多配置参数，再通过这些参数来控制 OracleJSP 的操作。这些参数是传给 JspServlet 的初始化参数，如何定义这些参数依赖于具体的 WebServer 和 Servlet 环境。

本节讨论 OracleJSP 的配置参数以及如何在当前最流行的 WebServer 和 Servlet 环境中设置它们。

提供 OracleJSP 的 Oracle 产品只对有限数目的参数感兴趣，并且各厂产品的设置参数的方法也不尽相同，如果需在 Oracle 环境下设置这些参数的话，请参考特定产品的说明文档。

Oracle Servlet Engine(OSE)所使用的配置设定在 OracleJSP 的预解释工具 ojspd 中都能得到相同的支持，并且 OSE 并不使用 Oracle JspServlet 作为解释和运行 JSP 页面的工具。

A.3.1 OracleJSP 在非 OSE 环境下的配置参数

本节描述了在 Apache/JServ，Sun 公司的 JSWDK 或 Tomcat 环境下 Oracle JspServlet

所支持的配置参数（需要注意的是，Oracle Internet Application Server 也使用 Apache/JServ 环境）。

对 Oracle Servlet Engine 环境来说，一些等价的配置功能可以在 ojspc 的等价选项中得到很好的支持。

配置参数列表

表 A-2 总结了 Oracle JspServlet（OracleJSP 的前端工具）所支持的配置参数，对每个参数，表中都有下列几项：

- 此参数是在页面解释时使用还是页面执行时使用。
- 此参数是在开发环境下使用还是在发行环境下使用。
- 在 ojspc 中所对应的选项，因为在 OSE 平台下并不使用 JspServlet，而是使用 ojspc。OSE 不支持运行时配置参数，还应注意下面几点：
- 在 OracleJSP 1.1.0.0.0 发行版之前的版本不支持这些参数：bypass_source，emit_debuginfo，external_resource，javaccmd 和 sqljcmd。
- 参数 alias_translation 只能在 Apache/JServ 环境中使用。
- 参数 session_sharing 只能在 globals.jsa 中使用（假设在 Apache/JServ 这样的 Servlet 2.0 环境下）。

注意：

- 参看“预解释工具 ojspc”以得到更进一步有关 ojspc 选项的信息。
- 在实时的解释环境下，运行时和解释时之间的差异并不重要但是在 OSE 环境下它们的差别就很重要了。

表 A-2 OracleJSP 配置参数

参数	相关的 ojspc 选项	描述	缺省值	用于 JSP 解释过程还是运行过程	用于开发环境还是发布环境
alias_translation	n/a	布尔型；设置为 true 将允许 OracleJSP 在一定的限制条件下使用 Apache/JServ 的目录别名服务	false	运行	开发与发布
Bypass_source	n/a	布尔型；设置为 true 将导致 OracleJSP 去处理并执行页面的实现类，而不管页面的源文件是否存在	false	运行	发布（也为 JDeveloper 所用）
classpath	-addclasspath（相关，但是有不同的功能）	指定 OracleJSP 类加载时的额外的类路径入口	null	解释或者运行	开发与发布

(续表)

参数	相关的 ojspc 选项	描述	缺省值	用于 JSP 解释过程还是运行过程	用于开发环境还是发布环境
developer_mode	n/a	布尔型；设置为 false 将导致当页面被请求时，OracleJSP 不会每次都检查页面实现类的时间标记与.jsp 源文件的时间标记	true	运行	开发与发布
emit_debuginfo	-debug	布尔型；设置为 true 将导致 OracleJSP 对源文件(.jsp 文件)生成行号信息用来调试	false	解释	开发
external_resource	-extres	布尔型；设置为 true 将导致 OracleJSP 解释器把已生成的静态内容放到一个 Java 资源文件里	false	解释	开发与发布
javacmd	-noCompile	指定 Java 编译器命令行——javac 选项，或者指定运行在单独的 Java 虚拟机中的另外一个 Java 编译器（null 意味着使用 JDK 的 javac 的缺省选项）	null	解释	开发与发布
page_repository_root	-srcdir -d	指定可选择的另外一个根目录（使用全路径），OracleJSP 使用此路径来生成并加载 JSP 页面	null(使用缺省根目录)	解释或运行	开发与发布
session_sharing(用在 globals.jsa 中)	n/a	布尔型；对于使用 globals.jsa 文件的应用程序来说，将此参数设置为 true，将使会话的数据可以共享	true	运行	开发与发布
sqljcmd	n/a	指定 SQLJ 命令行——sqlj 选项，或者指定运行在单独的 Java 虚拟机中的另外一个 SQLJ 编译器（null 意味着使用 OracleJSP 所提供的 SQLJ，并且使用缺省选项）	null	解释	开发与发布

(续表)

参数	相关的 ojspc 选项	描述	缺省值	用于 JSP 解释过程还是运行过程	用于开发环境还是发布环境
translate_params	n/a	布尔型；设置为 true 将控制 servlet 容器不要执行多字节编码	false	运行	开发与发布
unsafe_reload	n/a	布尔型；设置为 true 将允许在一个 JSP 页面被重新解释并重新加载时，不必重新启动应用程序和会话	false	运行	开发

配置参数说明

本节将更详细地讲述 OracleJSP 的配置参数。

alias_translation (布尔型；OracleJSP 缺省值：false)(Apache 专用)：此参数允许 OracleJSP 在一定的限制条件下使用 Apache/JServ 的目录别名服务，要得到当前限制条件的更进一步的信息，请参看“目录别名解释”。

为了在 Apache/JServ servlet 环境下正确地使用 httpd.conf 中的目录别名命令（如下例所示），必须将 alias_translation 设为 true。

```
Alias /icons/ "/apache/apache139/icons/"
```

bypass_source (布尔型；OracleJSP 缺省值：false)：正常情况下，当被请求的 JSP 页面的源文件没有找到时，OracleJSP 就会触发一个 FileNotFound 异常信息，它并不会去检查此页面的实现类存不存在。（这是因为缺省状态下，OracleJSP 将去检查页面源文件以查看它以页面的实现类生成以后是否被修改过。）

将此参数设成 true 将导致 OracleJSP 去处理并执行页面的实现类，而不管页面的源文件是否存在。

如果打开了 bypass_source，OracleJSP 仍将在必要的时候检查页面源文件是否需要重新解释（决定页面是否需要重新解释的一个因子是 developer_mode 参数的设置）。

注意：

- bypass_source 参数在发行环境下特别有用，它允许只存在生成的实现类，而没有源文件。
- Oracle JDeveloper 允许 bypass_source，从而使得你能够在将 JSP 页面保存到文件之前解释并运行它。

classpath(文件路径；OracleJSP 缺省值：null)：使用此参数可以向 OracleJSP 的缺省类路径里添加别的类路径入口信息，类路径信息在 JSP 页面的解释、编译和运行期都需要用到。

此命令具体的语法取决于 Web Server 环境和操作系统，“OracleJSP 的配置参数设置”一节列举了一些例子可供参考。

总体来说，OracleJSP 从自己的类路径（包括在 classpath 参数里设置的路径），系统的

类路径，Web Server 的类路径，页面的保存路径以及预先定义的相对于 JSP 应用程序根目录的路径下查找并装载类。

还要注意到通过 `classpath` 所指定的路径装载的类是由于 JSP 类载入器加载的，而不是由系统的类载入器所加载的类，也不能访问由别的类载入器所加载的类。

注意：

- OracleJSP 的运行时自动类重载仅适用于在 OracleJSP 的类路径里指定的类，它包括使用 `classpath` 参数所指定的类（请参看“动态类重载”以获得有关这方面的更详细的内容）。
- 在 Oracle Servlet Engine 环境下，如果正在使用预解释的页面，那么 `ojspc-addclasspath` 选项提供了相近（但是有差别）的功能，详细信息请参看“`ojspc` 的选项说明”。

developer_mode(布尔型; OracleJSP 缺省值: true): 如果将此参数设成 `false`，当页面被请求时，OracleJSP 就不会每次都检查页面实现类的时间标记与 `.jsp` 源文件的时间标记。在一般情况下，OracleJSP 每次都需要检查页面源文件以查看它从页面的实现类生成以后是否被修改过，如果被修改过的话，就要重新解释此页面，如果设置 `developer_mode=false`，OracleJSP 只会在第一次请求这个页面时检查，以后就简单地重复执行生成的页面实现类。

Oracle 建议将 `developer_mode` 设成 `false`，尤其是在发行环境下，此时源代码已经不可能更改，并且性能是最重要的问题。

emit_debuginfo(布尔型; OracleJSP 缺省值: false) (仅适用于开发者): 如果将此参数设为 `true`，OracleJSP 就会对源文件（`.jsp` 文件）生成行号信息用来调试。缺省情况下，此参数被设成 `false`，此时行号信息是相对于已生成的页面实现类。

注意：

- Oracle JDeveloper 允许 `emit_debuginfo` 参数。
- 在 Oracle Servlet Engine 环境下，如果正在使用预解释的页面，那么 `ojspc-debug` 选项提供了相同的功能。

external_resource(布尔型; OracleJSP 缺省值: false): 如果将此参数设为 `true`，OracleJSP 的解释器就会把已生成的静态内容（Java 的 `print` 命令输出静态的 HTML 代码）放到一个 Java 资源文件里，否则静态内容将被放到页面实现类的 `service` 方法里。

资源文件名是基于 JSP 页面的文件名的。在 8.1.7 发行版中，资源文件名与页面文件同名，不过后缀名是 `.res`。（例如解释 `MyPage.jsp` 文件，将得到 `MyPage.res` 文件以及标准的输出，然而在未来的版本中现在的实现有可能被改变）

资源文件与生成的类文件保存在同一目录下面。

如果一个页面中有许多静态内容，这一技术将加速解释速度，并且可以提高运行速度。在极端情况下，它甚至能解决因 Java 虚拟机（JVM）的限制而使 `service` 方法只能使用 64K 空间这一问题。

注意：在 Oracle Servlet Engine 环境下，如果正在使用预解释的页面，那么 `ojspc-extres` 选项提供了相同的功能。

Ojspc-hotload 选项也提供了相关的功能，它首先执行-extres 功能，然后将内容马上加载进 Oracle8i 中。

javaccmd(可执行的编译器; OracleJSP 缺省值: null):

此参数在下列情况中非常有用:

- 如果你想设置 javac 命令行选项 (尽管缺省的设置已经足够使用了)
- 如果你想使用 javac 以外的编译器 (可以自己设定命令行参数)

如果指定了别的一个编译器, OracleJSP 将会在单独的 Java 虚拟机中产生一个单独的进程以供这个编译器使用; 相反, 如果使用 javac 编译器, 则只能在 OracleJSP 自己的 Java 虚拟机中运行。在指定编译器时, 可以提供可执行文件的完全路径, 也可以只提供可执行文件名, 让 OracleJSP 在系统设定的路径中搜索。

下面的 javaccmd 命令设置了 -verbose 命令行选项。

```
javaccmd=javac -verbose
```

此命令的实际语法取决于特定的 Servlet 环境, 请参看 “OracleJSP 的配置参数设置” 以得到更多的信息。

注意:

- 指定的 Java 编译器必须在类路径里设置好, 任何前端工具也必须在系统路径里设置好。
- 在 Oracle Servlet Engine 环境下, 如果正在使用预解释的页面, 那么 ojspc -noCompile 选项提供了相似的功能, 这一选项导致页面内容不使用 javac 来编译, 因此你能使用自己希望的编译器来手工编译解释过的类。

page_repository_root(目录的全路径; OracleJSP 缺省值: null): OracleJSP 使用 Web Server 的文档仓库生成并加载解释后的 JSP 页面。缺省情况下, 在一个按需解释的应用中, 根目录就是 Web Server 的文档根目录 (适用于 Apache/JServ), 或者词页面所属应用程序的根目录。JSP 页面的源文件在应用程序的根目录下或者某些子目录中, 生成的文件存放在 _pages 子目录或别的对应目录中。

可以使用 page_repository_root 选项来改变缺省的根目录, 要想得到文件相对于根目录的路径信息以及 _pages 子目录的信息, 请参看 “OracleJSP 解释器的输出文件定位”。

注意:

- 如果指定的目录、_pages 子目录以及这些目录下的任意子目录不存在的话, OracleJSP 会自动创建它们。
- 在 Oracle Servlet Engine 环境下, 如果正在使用预解释的页面, 那么 ojspc -srcdir 和 -d 选项提供了相关的功能。

session_sharing(布尔型; OracleJSP 缺省值: true)(仅在 globals.jsa 中使用): 假设在 Servlet 2.0 环境下, 当一个应用程序使用 globals.jsa 文件时, 每个 JSP 页面所使用的 Session 对象都附属一个全局的 Servlet Session 对象, 此对象由 Servlet 容器所提供。

在这种情况下, 如果设置 session_sharing 为 true(缺省值), 就会使 JSP Session 的数据共享这唯一的 Servlet Session 对象, 从而此应用程序的 Servlets 访问 JSP 页面的 Session 数

据。

如果应用程序中设有使用 `globals.jsa` 这个文件，此参数无意义，要想得到 `globals.jsa` 文件的相关信息，请参看“`globals.jsa` 功能概述”。

sqljcmd(可执行的 SQLJ 解释器；OracleJSP 缺省值：null)：

此参数适用于下列情况：

- 如果你想设置 SQLJ 命令行选项。
- 如果你想使用与 OracleJSP 提供的解释器不同的 SQLJ 解释器（或者不同版本的解释器）。
- 如果你想在单独的进程中运行 SQLJ。

如果指定了别的 SQLJ 解释器，OracleJSP 将会在单独的 Java 虚拟机中产生一个单独的进程以供这个解释器使用；相反如果使用缺省的 SQLJ 解释器，则只能在 OracleJSP 自己的 Java 虚拟机中运行。在指定 SQLJ 解释器时，可以提供可执行文件的完全路径，也可以只提供可执行文件名，让 OracleJSP 在系统设定的路径中搜索。

下面是一个设置 `sqljcmd` 的例子。

```
sqljcmd=sqlj -user=scott/tiger -ser2class
```

注意：

- 指定的 SQLJ 解释器必须在类路径里放置好，任何前端工具（诸如 `sqlj`）也必须在系统路径里放置好。（对 Oracle SQLJ 来说，`translator.zip` 和必要的 SQLJ 运行时，ZIP 文件必须在类路径里。）
- 即使大部分的 OracleJSP 开发者愿意在他们的 JSP 页面的 SQLJ 代码开发中使用 Oracle SQLJ（与之相反的是一些别的 SQLJ 产品），这个选项还是相当有用的，你可以通过它使用一个不同版本的 Oracle SQLJ（例如，你可能倾向于使用 Oracle JDBC 8.0.x/7.3.x 来代替 Oracle8i），或者设置一些 SQLJ 命令行选项。

translate_params(布尔型；OracleJSP 缺省值：false)：在此参数的缺省设置（false）下，Servlet 容器对多字节的请求参数或 JavaBean 的属性设置不进行编码。如果将此参数设为 true，OracleJSP 就会将请求参数和 JavaBean 的属性设置进行编码。

因为 Oracle Servlet Engine 不支持运行时参数配置，所以在 OSE 环境下不能设置 `translate_params`。要想得到可用的解决方法，请参看“`translate_params` 配置参数的等价代码”。

为了得到更多关于 `translate_params` 的使用和功能的信息，包括什么时候不能使用此参数，请参看“OracleJSP 对多字节参数编码的扩展支持”。

unsafe_reload 参数（布尔型；OracleJSP 缺省值：false）（仅适用于开发者）：缺省情况下，当一个 JSP 页面被动态地重新解释并加载时（一般 `j` 在 JSP 的解释器发现 `.jsp` 源文件的修改日期比对应的页面实现类的修改日期晚的情况下），OracleJSP 将重新启动整个应用以及所有会话。

将此参数设成 true 将控制 OracleJSP 在发现页面动态重新解释并加载时，不要重启整个应用，这避免重启时现存的会话对象变成无效对象这一问题。

对一个给定的 JSP 页面来说，如果 `developer_mode` 设成 `false`，在此页面第一次被请求后，`unsafe_reload` 参数的设置就没有任何效果了（在这种情况下，OracleJSP 在页面第一次被请求后，就再也不用重新解释它了）。

注意：此参数只是为了提供给开发者使用，不建议在发行环境中使用。

A.3.2 OracleJSP 的配置参数设置

在上一节中讨论了如何设置 JSP 配置参数，它取决于特定的 Web Server 环境和 Servlet 环境。非 Oracle 环境通过属性文件来支持配置参数的设置；Oracle8i JServer 环境所提供的 Oracle Servlet Engine 不直接支持 OracleJSP 配置参数（因为它不使用 JspServlet），但是它的一些解释参数的设置和 OracleJSP 解释器的选项是等价的。这些选项在“配置参数列表”中曾经提到过，可以参看这部分内容以获得更多信息。

其他支持 OracleJSP 的 Oracle 产品有自己的设置参数的机制，请参看相应产品的说明文档。

本节剩余部分主要讨论如何在 Apache/JServ，Sun 公司的 JSWDK 以及 Tomcat 环境中设置配置参数。

在 Apache/JServ 环境下设置 OracleJSP 参数

在 Apache/JServ 环境中，每个网络应用程序都有自己的属性文件，也叫区域属性文件，在 Apache/JServ 的术语当中，区域（Zone）是和 Servlet 环境同义的。

区域属性文件的名字取决于你如何映射这个区。（请参看 Apache/JServ 文档以得到关于区域和映射（mounting）的详细信息。）

要在 Apache/JServ 环境下设置 OracleJSP 配置参数，首先在应用程序的区域属性文件中设置 JspServlet 的 `initArgs` 属性，如下例所示（使用 UNIX 语法）：

```
servlet.oracle.jsp.JspServlet.initArgs=developer_mode=false,  
sqljcmd=sqlj -user=scott/tiger -ser2class=true,classpath=/mydir/myapp.jar  
（上面是一个单行的命令，被断成了两行）。
```

`Servlet.oracle.jsp.JspServlet` 的路径取决于你是如何映射区域的，它并不是一个一成不变的路径。

注意：因为 `initArgs` 的参数是用逗号来分隔的，所以在一个参数设置里不能出现逗号，不过空格和其他特殊字符（如上例中的“=”）不会造成任何问题。

在 JSWDK 环境下设置 OracleJSP 参数

要在 JSWDK 环境下设置 OracleJSP 配置参数，首先在应用程序根目录下的 `WEB-INF` 目录中找到 `servlet.properties` 文件，然后设置 `jsp.initparams` 属性，如下例所示（使用 UNIX 语法）：

```
jsp.initparams=developer_mode=false,classpath=/mydir/myapp.jar
```

注意：因为 `initparams` 的参数是用逗号来分隔的，所以在一个参数设置里不能出现逗号，不过空格和其他特殊字符不会造成任何问题。

在 Tomcat 环境下设置 OracleJSP 参数

在 Tomcat 环境下设置 OracleJSP 参数只需将 init-param 小节加进 web.xml 文件。如下例所示：

注意：在[TOMCAT_HOME]/conf目录下，有一个全局的 web.xml 文件，如果你想要在特定的应用中重载这些设定，只需修改特定应用程序根目录下的 WEB-INF 子目录中的 web.xml 文件即可。

```
<servlet>
  <init-param>
    <param-name>
      developer_mode
    </param-name>
    <param-value>
      true
    </param-value>
  </init-param>
  <init-param>
    <param-name>
      external_resource
    </param-name>
    <param-value>
      true
    </param-value>
  </init-param>
  <init-param>
    <param-name>
      javaccmd
    </param-name>
    <param-value>
      javac -verbose
    </param-value>
  </init-param>
</servlet>
```

在 OSE 环境下的 JSP 配置

因为 OSE 不使用 OracleJSP JspServlet，所以它需要其他的机制来设置 OracleJSP 配置参数。

OSE 环境下使用 JSP 页面的预解释工具 ojspc，通过它的命令行选项，大部分解释时的参数配置可以得到等价的实现。OracleJSP 的配置参数和 ojspc 的选项之间的关系可以参看“配置参数列表”。

然而对运行时的参数配置来说，就没有如此等价的匹配了。其中最重要的是 translate_params，用来对多字节的请求参数编码，OSE 需要此功能，但是这需要开发者自己写等价的 JSP 代码来实现。要获得详细的信息，请参看“translate_params 配置参数的等价代码”。

附录 B JSP 和 Servlet 的技术背景

本附录中提供了有关 servlets 和 JSP 的技术背景。尽管本书是写给那些 servlet 技术背景知识比较不错的用户，但是这里的 servlet 信息可当做是复习内容。

标准 JSP 接口，被生成的 JSP 页面实现类所实现，在这里也作为主要讨论内容。然而大多数实际上不需要这些知识。

讨论的内容主要涵盖：

- Servlets 背景
- Web 应用程序层
- 标准 JSP 接口与方法

B.1 Servlets 背景

因为 JSP 页面被翻译成 Java servlets，因此有关 servlet 技术主要内容的复习是很有益的。关于这里要讨论的基本概念的更详细的信息，请参看 Sun 公司 Java Servlet 技术标准 2.2 版本。

要获得本节讨论方法的更多信息，请参看 Sun 公司 Javadoc 在下面网站中的内容：

<http://java.sun.com/products/servlet/2.2/javadoc/index.html>

B.1.1 Servlet 技术回顾

在近几年来，servlet 技术在通过动态 HTML 页面扩展 Web server 上呈现出一种强有力的方法。一个 servlet 就是一个运行在 Web server 中的 Java 程序（相对于 applet 来说，它是运行在客户端浏览器的 Java 程序。）。servlet 从浏览器中获取一个 HTTP 请求，生成动态内容（例如查询一个数据库），并把 HTTP 的响应返回给浏览器。

在 servlets 之前，CGI（Common Gateway Interface）技术被用在动态内容中，它的程序是用 Perl 语句书写的，并通过 Web server 被一个 Web 应用程序所调用。然而，由于它的结构以及可升级性的限制，CGI 最后被证明为是不太理想的解决方案。

Servlet 技术，在可升级性上有了很大的改善，它提供了公认的 Java 在对象定位、平台独立、安全性以及强壮性等方面的优点。

Servlets 能使用所有的标准 Java APIs，包括 JDBC API（为 Java 数据库连接性所写的接口技术，对于数据库开发者来说特别感兴趣）。

在 Java 领域中，servlet 技术为密集型 server 应用程序（比如访问一个数据库）提供了比 applet 技术更多的优点。优点之一就是 servlet 运行在服务器端，服务器端具有多种资源且属于一个相对强壮的机器，用此占用客户端的资源相当少。而 applet 则相反，它需要下载到客户端浏览器并且在客户端运行。另外一个优点就是 servlet 在访问数据时更加直接。因为运行 servlet 的 Web 服务器或者数据服务器在数据被访问时是与网络防火墙在一端，而 applet 运行在客户端的机器上，且在防火墙的外面，不同于它所下载到的本地机器，applet 在访问任何一个服务器时需要特定的措施（比如说被分配的 applets）。

B.1.2 Servlet 接口

一个 Java servlet，根据定义，实现了标准的 `javax.servlet.Servlet` 接口。这个接口指定一些方法来初始化一个 servlet，过程请求，获得配置和其他关于 servlet 的基本信息，并且终止一个 servlet 实例。

对于 Web 应用程序来说，Servlet 接口通过继承标准的 `javax.servlet.http.HttpServlet` 抽象类而间接实现。`HttpServlet` 类包含以下方法：

- `init(...)`和 `destroy(...)`——来初始化和终止 servlet，独立操作。
- `doGet(...)`——针对 HTTP GET 请求。
- `doPost(...)`——针对 HTTP POST 请求。
- `doPut(...)`——针对 HTTP PUT 请求。
- `doDelete(...)`——针对 HTTP DELETE 请求。
- `service(...)`——来接收 HTTP 请求，在缺省状态把它们分发给适当的 `doXXX()`方法。
- `getServletInfo(...)`——servlet 利用它提供有关自身的信息。

一个继承了 `HttpServlet` 的 servlet 类必须在适当时实现这些方法中的一部分，每个方法都有一个标准的 `javax.servlet.http.HttpServletRequest` 实例和一个标准的 `javax.servlet.http.HttpServletResponse` 实例作为输入参数。

`HttpServletRequest` 实例要为 servlet 提供关于 HTTP 请求的信息，比如请求参数名和值，提出请求的远程主机名和接收到请求的服务器名。`HttpServletResponse` 实例在发送响应信息时提供特定的 HTTP 功能，比如指定内容长度和 MIME 类型以及提供输出流等。

B.1.3 Servlet 容器

Servlet 容器，有时被称作是 servlet 引擎，用来执行和管理 servlets。一个 Servlet 容器通常是用 Java 写成的，它要么作为 Web Server（如果 Web Server 也是用 Java 写成的话）的一部分，要么被一个 Web server 所使用。

当一个 servlet 被调用时（比如当一个 servlet 被 URL 所指定时），Web Server 就传递 HTTP 请求给 servlet 容器。容器依次把请求传递给 servlet。在管理一个 servlet 的过程中，一个简单的容器要完成下列操作：

- 创建一个 servlet 实例并且调用它的 `init()`方法来初始化它。
- 调用 servlet 的 `service()`方法。
- 调用 servlet 的 `destroy()`方法并在适当时抛掉，以便把无用的资源回收。

由于性能原因，有一个很典型的情况是，对于一个 servlet 容器在内存被重新利用时，保存一个 servlet 实例比破坏它更容易。它只是在非常罕见的事件中遭到破坏，比如 Web Server 突然关闭。

当一个 servlet 正在运行时，如果有一个额外的 servlet 请求，servlet 容器的动作要取决于当前的 servlet 是使用一个单线程模型还是一个多线程模型。如果是在单线程情况的下，servlet 就阻止多个同时的 `service()`的调用被分发到单个的 servlet 实例中——相反，它可能产生多个分离的 servlet 实例。如果是在多线程模型中，servlet 容器就能通过为每一个调用分离线程而使多个同时对 `service()`的调用分发到单个的 servlet 实例中，但是 servlet 开发者

需要为同步管理而负责。

B.1.4 Servlet 会话

Servlets 利用 HTTP 会话来保持对来自某个用户每一个 HTTP 请求的跟踪,以致于来自单个用户的一组请求能被全状态管理。Servlet 能够设置和获取在 HttpSession 对象中会话的信息,但它必须具有与应用程序同级的作用域。

Http Session 接口

在标准的 Servlet API 中,每个用户是由一个实现了标准的 javax.servlet.http.HttpSession 接口的对象来表示的。Servlets 可以从 Http Session 对象中得到并且设置关于会话的信息,但是 HttpSession 对象必须有应用程序范围的作用域。

servlet 运用一个 HttpServletRequest 对象的 getSession()方法来为用户检索或者创建一个 HttpSession 对象。该方法采用一个布尔参数来指定是否一个新的会话对象将为用户(如果用户还不存在的话)创建。

HttpSession 接口为获取和设置信息指定了下面的方法:

- public void setAttribute (字符串名, 对象值)
这将为会话在特定的名字下绑定特定的对象。
- public void getAttribute (字符串名)
这是用来搜索在特定的名字下绑定会话的对象(如果不匹配的话则为空)

注意: 旧的 servlet 实现使用 putValue()和 getValue()代替 setAttribute()和 getAttribute(),它们具有同样的功能。

借助于对 servlet 容器的实现和 servlet 本身,会话能在一段时间后自动终止或者被 servlet 明确视为无效。Servlets 能够在被 HttpSession 接口指定的情况下,借助于下面的方法管理会话的生命周期:

- public boolean invalidate()
该方法立即使会话无效并且解除绑定它的所有对象。
- public boolean setMaxInactiveInterval(int interval)
该方法以秒作为超时时间间隔,要求是整数。
- public boolean isNew()
该方法在请求创建了会话内容返回 true, 否则返回 false。
- public boolean getCreationTime()
该方法返回在会话对象被创建时的时间,从 1970 年一月一日凌晨起以毫秒为单位。
- public boolean getLastAccessedTime()
该方法返回最后一次请求客户端的时间,从 1970 年一月一日凌晨起以毫秒为单位。

会话跟踪

HttpSession 接口支持两种机制来跟踪会话。每一种都涉及某种方法来分配一个 session ID。一个 session ID 就是一个被 servlet 容器所使用和分配的中间句柄。如果合适的话,同

一用户能够共享同一个 session ID 来管理多个会话。

下面就是所支持的会话跟踪机制：

- cookies

servlet 容器发送一个 cookie 给客户，客户在每一个 HTTP 请求中再把 cookie 返回到服务器，这把请求和被 cookie 指示的会话 ID 联合起来，JSESSIONID 就一定是 cookie 的名字。

这是最频繁使用的机制，它被任何一个符合 servlet 2.2 标准的 servlet 容器所支持。

- URL rewriting

servlet 容器对 URL 路径追加一个会话 ID，路径参数的名字必须是 jsessionid，请参考下面的例子：

```
http://host[:port]/myapp/index.html;jsessionid=6789
```

这是最频繁使用的机制，而在这里客户不接受 cookie。

- SSL Session

SSL (Secure Socket Layer, 运用在 HTTPS 协议中) 包含一个从一个客户获取多个请求的机制，并把它们定义为属于单个会话。某些 servlet 容器也为它们自身的会话跟踪使用 SSL 机制。

B.1.5 Servlet 环境

Servlet 环境被用来维护在任何单个 Java 虚拟机里的一个 Web 应用程序中的所有实例的全状态信息（也就是说，对于部分 Web 应用程序中所有的 servlet 和 JSP 页面实例）。这与在服务器端一个会话维护单个客户的全状态信息的方式类似。然而，servlet 环境并没有专门指定给任何单个用户，而是能够处理多个客户。这里通常有一个为每个运行在给定 Java 虚拟机里的 Web 应用程序所设的 servlet 环境。你可以把一个 servlet 环境想象成一个“应用程序容器”。

任何一个 servlet 环境都是一个实现标准 `javax.servlet.ServletContext` 接口的类的实例，这个类被任何一个支持 servlet 的 Web Server 所提供。

`ServletContext` 对象提供有关 servlet 环境的信息（比如服务器）并且在任何单个 JVM 中允许共享在同一组中两个会话的资源。（因为 servlet 容器支持多个同时的 JVMs，资源共享的实现是变化的。）

一个 servlet 环境是用来维护正在运行应用程序的用户的会话对象，并且为正在运行的应用程序实例提供一个作用域。通过这个机制，每一个应用程序都是从一个独特的类加载器中加载，而它的运行时对象也与其他任何一个应用程序的对象截然不同。特别地，`ServletContext` 对象对于应用程序来说是不同的，正如 `HttpSession` 对象对于每个应用程序的用户不同一样。

依照 Sun 公司 JSP 2.2 版本，大多数实现能够在单个主机上提供多个 servlet 环境，这就允许每个 Web 应用程序拥有各自的 servlet 环境。（以前的实现通常在任何给定的主机上只提供一种 servlet 环境。）

`ServletContext` 接口指定了允许 servlet 与运行它的 servlet 容器通信的方法，这是 servlet 搜索同级应用程序和全状态信息方法中的一种。

注意：在早期的 servlet 标准版本中，servlet 环境的概念没有被充分定义。从 2.1(b)版本开始，这个概念被进一步阐明了，它被定义为一个不能跨多个 servlet 环境对象而存在的 HTTP 会话对象。

B.1.6 通过事件监听器来管理应用程序的生命周期

Java Servlet 标准 2.1（或更高）版本通过标准 Java 事件监听器机制提供了有限的应用程序的生命周期管理。HTTP 会话对象能够运用事件监听器使得存储在会话对象中的对象得知他们在被添加或移动时的情况。由于在一个会话对象移动对象的典型原因是会话变为无效，这个机制允许开发者管理基于会话的资源。

不幸的是，servlet 环境对象不支持这类通告。标准的 servlet 应用程序支持不提供一种管理基于应用程序资源的方法。

B.1.7 Servlet 调用

一个 servlet 跟一个 HTML 页面一样，是通过一个 URL 调用的。servlet 是根据 servlets 如何映射到 Web Server 实现中的 URLs 而启动的。下面是可能用到的方法：

- 一个特定的 URL 能够被映射到一个指定的 servlet 类中。
- 一个全路径能够被映射以致于路径中的任何一个类被作为 servlet 而执行。例如，特定的/servlet 路径能够被映射以致于格式 URL/servlet/<servlet_name>中的任何类执行一个 servlet。
- 一个文件扩展名能够被映射，以致于指向一个含有扩展名的文件的任何 URL 执行一个 servlet。

这个映射将被指定为 Web Server 配置中的一部分。

B.2 Web 应用程序分层

与一个 Web 应用程序（由 servlets 和 JSP 页面的一些联合组成）相关联的实体不遵从一个简单的分层关系，但能以下的顺序考虑：

1、servlet 对象（包含页面实现对象）

这里有一个有关在一个正在运行的应用程序中（可能不只一个对象，取决于使用的是单线程还是多线程模型）的每一个 servlet 和每一个 JSP 页面实现的 servlet 类。一个 servlet 对象处理来自一个客户的请求对象并把响应对象送回给客户。一个 JSP 页面和它的 servlet 代码指定了如何创建响应对象。

在某些情况下，你可以想象多个 servlet 对象处在单个的请求对象中，比如当一个页面或者 servlet 包含或导向到另外一个页面或者 servlet 中时。

一般情况下，用户将在会话进行的过程中访问多个 servlet 对象，此时在 servlet 对象和会话对象之间有了一种关联关系。

servlet 对象和页面实现类对象间接地实现了标准的 javax.servlet.Servlet 接口。对于一个 Web 应用程序中的 servlets 来说，这是通过继承标准的 javax.servlet.http.HttpServlet 抽象类来实现的。对于 JSP 页面来说，这是通过实现

标准的 `javax.servlet.jsp.HttpJspPage` 接口来实现的。

2、request 和 response 对象

这两个对象分别代表着 HTTP 请求和响应信息，这些信息在一个用户运行应用程序的时候生成。

在一个会话进行的过程中，一个用户一般都要生成多个请求信息标签得到多个响应信息。request 和 response 对象并不包含在会话里，但是它们与会话相关联。

当一个请求信息从客户端到达后，它被映射到合适的 servlet 环境对象中（一般是客户端正在使用的应用程序相关联的那一个），这个映射过程是根据 URL 中所指定的虚拟路径来进行的，虚拟路径包括了应用程序的根路径。

一个 request 对象实现了标准的 `java.servlet.http.HttpServletRequest` 接口。

一个 response 对象实现了标准的 `java.servlet.http.HttpServletResponse` 接口。

3、session 对象

session 对象存储了某个特定的会话的用户信息，标签提供了一种手段可以从多个页面请求信息中辨认出某个特定的用户。对每个用户来说都有一个 session 对象。在任一个给定的时间，可能同时存在多个用户同时访问一个 servlet 或 JSP 页面，每个用户都由他们自动的 session 对象来代表。然而，所有的这些 session 对象是由整个应用程序所对应的 servlet 环境来维护的。事实上，你可以认为每个 session 对象都代表了一个与通用 servlet 环境相关联的 Web 应用程序的实例。

一般情况下，一个 session 对象将顺序地利用多个 request 对象、response 对象以及 page 或 servlet 对象，并且别的 session 对象不会使用与之相同的这些对象；不过应该注意，session 对象并不包含这些对象。

对一个给定的用户来说，会话的生命周期从此用户第一次请求页面时开始，到用户会话终止时（比如用户退出了应用程序）或者超时的时候结束。

HTTP session 对象实现了标准的 `javax.servlet.http.HttpSession` 接口。

注意：在 servlet 2.1(b)标准之前，一个 session 对象可以跨越多个 servlet 环境对象。

4、servlet 环境对象

一个 servlet 环境对象与服务器端一个特定的路径相关联。这个路径是与 servlet 环境相关联的应用程序模块的基路径，也叫做应用程序根目录。

在任何一个给定的 Java 虚拟机中，对应用程序的所有会话来说只有一个 servlet 环境，它从服务器端为组成应用程序的 servlet 和 JSP 页面提供信息。servlet 环境对象也允许应用程序的会话在一个与其他应用程序相隔离的安全的环境下共享数据。

servlet 容器提供了一个类实现了标准的 `javax.servlet.ServletContext` 接口，当一个用户第一次请求这个应用程序时，这个类将被实例化，并且与包含着应用程序位置的路径信息一起被提供。

一般情况下，servlet 环境对象有一个 session 对象池用来代表多个并发的应用程序用户。

一个 servlet 环境的生命周期由对应用程序的第一次请求（可以是来自任何用户）开始，而它的生命周期仅仅在服务器关闭或者终止的时候才结束。

5、 servlet 配置对象

servlet 容器使用一个 servlet 配置对象在一个 servlet 被初始化时给它传递信息——servlet 接口的 `init()` 方法需要一个 servlet 配置对象作为输入参数。

servlet 容器提供了一个类实现了标准的 `javax.servlet.ServletConfig` 接口并且在必要的时候实例化它。在 servlet 配置对象中包含的是一个 servlet 环境对象（它也由 servlet 容器实例）。

B.3 标准的 JSP 接口和方法

在 `javax.servlet.jsp` 包中有两个标准的接口，它们可以在 JSP 解释器所生成的代码中被实现，这两个接口是：

- `JspPage`
- `HttpJspPage`

`JspPage` 是一个通用的接口，它并不是专门用于某一个特定的协议的，它是由 `javax.servlet.Servlet` 接口继承下来的。

`HttpJspPage` 是一个专用的接口，它专门用于使用 HTTP 协议的 JSP 页面。它是从 `JspPage` 继承下来的，并且一般情况下，在 JSP 解释器所生成的任何 servlet 类中都被自动地直接实现。

`JspPage` 指定了以下的方法，它们可以被用来初始化和终止生成类的实例：

- `jspInit()`
- `jspDestroy()`

对这两个方法所写的任何代码都必须被包含在你的 JSP 页面的脚本段中，如下面的例子所示：

```
<%!
    void jspInit()
    {
        ... your implementation code...
    }
%>
```

（本章的后面将描述 JSP 语法，请参看“脚本元素”以获得相关信息。）

- `_jspService()`

这个方法的代码一般由解释器自动生成，并且它包含了 JSP 页面脚本段中的代码、任何 JSP 指令所导致的代码以及页面的任何静态内容等。（JSP 指令语句是用来为页面提供信息的，比如指定脚本语言用 Java 语言，指定导入的包等。要获得相关信息，请参看“指令语句”）

与在“Servlet 接口”中所讨论的 servlet 方法一样，`_jspService()` 方法也以一個 `HttpServletRequest` 实例和一个 `HttpServletResponse` 实例作为输入参数。

`JspPage` 和 `HttpJspPage` 接口从 `Servlet` 接口中继承了以下的方法：

- `init()`
- `destroy()`

- `service()`
- `getServletConfig()`
- `getServletInfo()`

要获得关于 Servlet 接口和它的关键方法的讨论，请参看“Servlet 接口”。

附录 C 编译时 JML 标签支持

Oracle 1.0.0.6.x 发行版，因为它们属于 JSP 1.0 实现，所以只能通过特定 Oracle 扩展来支持 JML 标签。（标签库框架是直到 JSP 1.1 版本才被加入到 JSP 标准中。）对于这些发行版，JML 标签处理是嵌入到 OracleJSP 解释器中的，这些内容可参考“编译时 JML 支持”。

1.1.0.0.0 发行版继续支持编译时 JML 实现，然而，它通常被用在运行时实现，这在第 7 章中有讲解。

本附录主要讨论编译时实现的特征，它们不同于运行时实现。主要包含以下主题：

- JML 编译时与运行时的考虑与逻辑比较
- JML 编译时/1.0.0.6.x 语法支持
- JML 编译时/1.0.0.6.x 标签支持

C.1 JML 编译时与运行时的考虑与逻辑比较

本节主要讨论了编译时标签库与运行时标签库相比较的两个方面：

- 在当可能对运用编译时标签库实现有利时的这方面的一般考虑（适用于任何库，而不仅仅是 JML）。
- 在特殊情况下，编译时 JML 实现所需求的 taglib 指令。

C.1.1 通用的编译时与运行时的考虑

Sun 公司 JSP 标准 1.1 版本中描述了对于定制标签库的运行时支持技巧。这个技巧是运用一个 XML-style 的标签库描述文件来指定标签，这包含在“标准标签库框架”中。

创建和运用一个遵循这个模型的标签库可确保这个库将能被移植到任何标准 JSP 环境中。

然而，在考虑编译时实现时有几点原因：

- 编译时实现能产生更有效的代码。
- 编译时实现允许开发人员在解释与编译时捕捉错误，从而代替了最终用户在运行时看到它们。

将来，Oracle 可能会利用编译时标签实现来为创建定制标签库提供一个通用的框架。这些实现将会依赖于 OracleJSP 解释器，因此将不能被移植到其他 JSP 环境中。

有关编译时的常用优缺点将都运用于 Oracle JML 标签库中，在某些情况下，运用编译时 JML 实现作为最初引入到 OracleJSP 的旧版本中是有利的。同时，也有一点附加的标签在那个实现中，并且支持某种附加的表达式语法。（请参见“JML 编译时/1.0.0.6.x 语法支持”和“JML 编译时/1.0.0.6.x 标签支持”中的内容。）

然而，通常情况下建议使用遵循 JSP 1.1 标准中的 JML 运行时实现。

C.1.2 编译时 JML 支持的 taglib 指令

OracleJSP1.0.0.6.x 编译时支持实现运用了 Oracle 提供的定制类 OpenJspRegisterLib 来实现 JML 标签支持。

在一个 JSP 页面中使用带有编译时实现的 JML 标签，taglib 指令就必须指定这个类中全局限定的名字（不同于指定一个 TLD 文件作为在标准 JSP 1.1 标签库的用法）。

下面是一个例子：

```
<%@ taglib uri="oracle.jsp.parse. OpenJspRegisterLib" prefix="jml" %>
```

要获得有关 taglib 指令在 JML 运行时实现中的用法的信息，请参见“taglib 指令”中的内容。

C.2 JML 编译时的语法支持（1.0.0.6.x 发行版）

本节描述了编译时 JML 实现支持的特定 Oracle bean 参考语法与表达式语法，用来指定标签属性的值。主要涵盖以下主题：

- JML Bean 引用与表达式、编译时实现
- 用 JML 表达式进行属性设置

这个功能不能被移植到其他 JSP 环境中。

C.2.1 JML Bean 引用与表达式（编译时实现）

一般来讲，一个 bean 引用是一个 JavaBean 实例的任何一个引用，这个 JavaBean 能访问这个 bean 的属性或方法。这包含引用一个 bean 的属性或方法，在这里这个 bean 本身就是另外一个 bean 的一个属性。

这就变得比较累赘，因为标准 JavaBean 语法要求这些属性要被调用它们的存取程序访问而不是被直接引用。例如，参考下面的直接引用：

```
a.b.c.d.doIt()
```

这必须用下面标准的 JavaBeans 语法来表达：

```
a.getB().getC().getD().doIt()
```

但是，Oracle 的编译时 JML 实现提供了缩写语法。

JML Bean 引用

被编译时 JML 实现所支持的特定 Oracle 语法允许 bean 在引用时直接用“.”（点）符号来表达。注意：标准 bean 属性存取程序方法中的语法也仍然有效。

参考下面标准 JavaBean 的引用：

```
customer.getName()
```

在 JML bean 引用语法中，你可以用下面另外一种表达方式：

```
customer.getName()
```

或者

```
customer.name()
```

JavaBeans 有一个可选的缺省属性，它的引用是在假定没有引用被明确确定的前提条件下。缺省属性的名字在 JML bean 引用中可被省略掉。在上面的例子中，如果 name 是缺省

属性，下面的就也是有效的 JML bean 引用：

```
customer.getName()
```

或者

```
customer.name()
```

或者更简单的：

```
customer
```

大多数 JavaBean 没有定义一个缺省属性。在那些有定义的 JavaBean 中，最重要的是“JML 扩展数据类型”中描述的 JML 数据类型 JavaBeans。

JML 表达式

被编译时 JML 实现所支持的 JML 表达式语法是一个标准 JSP 表达式的超集，添加了对 JML bean 引用语法的支持（在前面的章节中有描述）。

一个 JML bean 引用出现在一个 JML 表达式中必须是下面的封装形式：

```
$( JML_bean_reference )
```

C.2.2 用 JML 表达式来进行属性设置

在“JSP 标记语言（JML）标签描述”的下面注解中讲到：标签属性文件是可移植的。如那里所讲述的，你可以为运行时或者编译时 JML 实现甚至是非 OracleJSP 环境设置属性。

如果你打算只运用特定 Oracle 编译时实现的话，那么你能通过使用 JML bean 引用和 JML 表达式语法来设置属性，这在“JML Bean 引用与表达式、编译时实现”中有讲解。

注意下面几点：

- 无论在第 7 章中任何地方讲解的有关接受一串字符文字或者一个表达式的属性，你都能够标准 JSP<%=...%>语法中运用一个\$[...]形式的 JML 表达式。

参考一个运用 JML useVariable 标签的例子，你将运用下面的语法到运行时实现中：

```
<jml:useVariable id="isVariableUser" type="boolean" value="<%=  
dbConn.isValid()%> scope="session" />
```

你也可以运用下面的另一种语法到编译时实现中（数值的属性要么是一串字符文字要么是一个表达式）：

```
<jml:useVariable id="isVariableUser" type="boolean" value="<%=  
$(dbConn.isValid()%> scope="session" />
```

- 无论在第 7 章中任何地方讲解的仅接受一个表达式的属性时，你就能够运用一个\$[...]形式的 JML 表达式而不必嵌套在<%=...%>语法中。

参考一个运用 JML choose...when 标签的例子，你将运用下面的语法到运行时实现中（假定 orderedItem 是一个 JmlBoolean 例子）：

```
<jml:choose>  
  <jml:when condition = "<%= orderedItem.getValue() %>" >  
    You have changed your order:  
    -- outputs the current order --  
  </jml:when>  
  <jml:otherwise>  
    Are you sure we can't interest you in something?  
  </jml:otherwise>  
</jml:choose>
```

你也可以运用下面的另一种语法到编译时实现中（数值的属性要么是一串字符文字要么是一个表达式）：

```
<jml:choose>
    <jml:when condition = "$[orderedItem]" >
        You have changed your order:
        -- outputs the current order --
    </jml:when>
    <jml:otherwise>
        Are you sure we can't interest you in something?
    </jml:otherwise>
</jml:choose>
```

C.3 JML 编译时的标签支持（1.0.0.6.x 发行版）

本节主要描述以下内容：

- 所有编译时标签的汇总，注意它们在运行时实现中不支持。
- 被编译时实现所支持而不被运行时实现所支持的标签的描述（因为这类标签在“JSP 标记语言（JML）标签描述”中没有讲解）。

注意：在大多数情况下，不被运行时实现所支持的 JML 标签有标准的 JSP 当量。然而编译时中的某些标签不被支持是因为它们所具有的功能在遵循 JSP 1.1 标准时很难被实现。

C.3.1 JML 标签小结（1.0.0.6.x/编译时与 1.1.0.0.0/运行时比较）

大多数 JML 标签在运行时模型与编译时模型中均可获得，但也有例外，汇总如表 C-1。

表 C-1 JML 标签支持：编译时模型与运行时模型比较

标签	是否在编译时实现中支持	是否在运行时实现中支持
Bean 绑定标签：		
useBean	yes	no;使用 jsp:useBean
useVariable	yes	yes
useForm	yes	yes
useCookie	yes	yes
remove	yes	yes
Bean 操作标签：		
getProperty	yes	no;使用 jsp:getProperty
setProperty	yes	no;使用 jsp:setProperty
set	yes	no
call	yes	no
lock	yes	no
控制流标签：		

(续表)

标签	是否在编译时实现中支持	是否在运行时实现中支持
if	yes	yes
choose	yes	yes
for	yes	yes
foreach	yes;类型属性可选	yes;类型属性可选
return	yes	yes
flush	yes	yes
include	yes	no;使用 jsp:include
forward	yes	no;使用 jsp:forward
XML 标签:		
transform	yes	yes
styleSheet	yes	yes
工具标签:		
print	yes; 使用双引号来指定一串字符 文字	no; 使用 JSP 表达式
plugin	yes	no; 使用 jsp:plugin

C.3.2 其他 JML 标签描述（编译时实现）

本节详细地提供了被 JML 编译时实现支持，但不被运行时实现支持的 JML 标签的描述，这些标签在“JSP 标记语言（JML）标签描述”中没有讲解。

总结起来，共包括以下的标签：

- JML sueBean 标签
- JML getProperty 标签
- JML setProperty 标签
- JML set 标签
- JML call 标签
- JML lock 标签
- JML include 标签
- JML forward 标签
- JML print 标签
- JML plugin 标签

对于在标签描述中的语法文件，注意以下几个方面：

- 斜体表明你必须指定一个数值或字符串
- 可选的属性要标注在方括号中[...]
- 可选属性的缺省值用黑体标注
- 在如何指定一个属性的选择上用竖直线分开|
- 这里运用了前缀“jml:”，这是惯例，但并不必要。你可以在你的 taglib 指令中指

定任何前缀。

JML useBean 标签

这个标签声明一个被运用在页面中的对象，根据名字（如果存在）定位先前例子的对象在指定的作用域中。如果不存在，该标签将生成一个适当的类的新的例子并根据名字把它归属到指定的作用域中。

语法和语义对于标准 `jsp:useBean` 标签来说是一样的，除了一个 JSP 表达式在 `jsp:useBean` 用法中是有效的，或者说要么一个 JML 表达式要么一个 JSP 表达式在 JML `useBean` 用法中有效。

语法：

```
<jml:useBean id = " beanInstanceName"
            scope ="page | request | session | application"
            class = " package.class" | type = " package.class" |
            class = " package.class" type = " package.class" |
            beanName = " package.class | <%= jmlExpression %>"
            type = " package.class" />
```

另外，你可以运用其他嵌套的标签比如 `setProperty` 标签，用一个 `</jml:useBean>` 结束标签。

属性：

参考 Sun 公司 JSP 标准 1.1 版本中有关属性和它的语法的信息。

例子：

```
<jml:useBean id= "isValidUser" class= "oracle.jsp.jml.JmlBoolean" =scope=
"session" />
```

JML getProperty 标签

这个标签和标准 `jsp:getProperty` 标签在功能上是完全相同的，它把 bean 属性的值输出到响应中。

要获得关于 `getProperty` 用法的详细信息，请参见 Sun 公司 JSP 标准 1.1 版本。

语法：

```
<jml: getProperty name= "beanInstanceName" property= "propertyName" />
```

属性：

- **name**——这是正在被检索其属性的 bean 的名字。这个属性是必需的。
- **property**——这是正在被检索属性的名字。这个属性是必需的。

例子：下面的例子输出了当前工资属性的值：

```
<jml:getProperty name= "salary" property= "value" />
```

等价于下面的语句：

```
<% salary.getValue() %>
```

JML setProperty 标签

这个标签涵盖了被标准 `jsp:setProperty` 标签所支持的功能，但是也增加了支持 JML 表达式的功能。特别是，你可以使用 JML bean 引用。

要获得关于 `setProperty` 用法的详细信息，请参考 Sun 公司 JSP 标准 1.1 版本。

语法:

```
<jml:setProperty name = " beanInstanceName"
    property = " * " |
    property = " propertyName" [ param = " parameterName" ] |
    property = " propertyName"
    value = " stringLiteral | <%= jmlExpression %>" />
```

属性:

- **name**——这是正在被设置其属性的 **bean** 的名字。这个属性是必需的。
- **property**——这是正在被设置属性的名字。这个属性是必需的。
- **value**——这个参数是可选的, 它让你直接设置数值而不再从一个请求参数中设置。

JML **setProperty** 标签支持除标准 JSP 表达式之外的 JML 表达式来指定这个数值。

例子: 下面的例子是提升了 6% 后的更新的工资情况。(假定 **salary** 是类型 **JmlNumber**。)

```
<jml:setProperty name= "salary" property= "value" value= "<%=
${salary}*1.06%>" />
```

JML set 标签

这个标签提供了设置一个 **bean** 属性的另一种方法, 使用语法要比用 **setProperty** 标签更方便。

语法:

```
<jml:set name= "beanInstanceName.propertyName"
    value= "stringLiteral | <% jmlExpression % >" />
```

属性:

- **name**——这是一个对于被设置的 **bean** 属性的直接引用 (JML **bean** 引用)。这个属性是必需的。
- **value**——这是一个新的属性。它被表达的方式要么是用一串字符文, 要么是一个 JML 表达式, 或者是标准的 JSP 表达式。这个属性是必需的。

例子: 下面的每一个例子都是提升了 6% 后的更新的工资情况。(假定 **salary** 是类型 **JmlNumber**。)

```
<jml:set name= "salary.value" value= "<%= salary.getValue()*1.06%>" />
```

或者:

```
<jml:set name= "salary.value" value= "<%= ${salary.value}*1.06%>" />
```

或者:

```
<jml:set name= "salary" value= "<%= ${salary}*1.06%>" />
```

这些都等价于下面的语法:

```
<% salary.setValur(salary.getValue()*1.06); %>
```

JML call 标签

这个标签提供了一个关于 **bean** 方法返回空的技巧。

语法:

```
<jml:call method = "beanInstanceName.methodName(parameters)" ?>
```

属性:

- **method**——这是一个当你正在用 `scriptlet` 写方法时的方法调用，除了语句中 `beanInstanceName.methodName` 部分能被写作一个 JML bean 引用，当然这需要它 是用 JML 表达式 `[$...]` 语法封装的。这个属性是必需的。

例子：下面的例子使用户转向不同的页面：

```
<jml:call name='response.sendRedirect("http://www.oracle.com/")' />
```

它等价于下面的语法：

```
<% response.sendRedirect("http://www.oracle.com/")' %/>
```

JML lock 标签

这个标签允许被控制，同步访问以指定的对象正文中的任何代码。

一般来说，JSP 开发者无需关注并行问题，然而，由于应用程序作用域中的对象对于所有正在运行应用程序的用户来说是共享的，因此访问重要的数据必须是受控的而且是整理过的。

你可以运用 JML lock 标签来阻止其他不同的用户同时更新数据。

语法:

```
<jml:lock name = " beanInstanceName" >
... body...
</jml:lock>
```

属性:

- **name**——这是在执行 lock 标签正文代码过程中将被锁定的对象的名字。这个属性是必需的。

例如：在下面的例子中，`pageCount` 是一个应用程序作用域中 `JmlNumber` 的值。变量被锁定的目的是阻止在代码获得当前值与设置新的值过程中被其他用户更新该值。

```
<jml:lock name="pageCount" >
    <jml:set name="pageCount.value" value="<%= pageCount.getValue() + 1 %>"
/>
</jml:lock>
```

它等价于下面的程序：

```
<% synchronized(pageCount)
{
    pageCount.setValue(pageCount.getValue() + 1);
}
%>
```

JML include 标签

这个标签包含了这个页面在响应时（页面正在被调用）另外的一个 JSP 页面、一个 `servlet` 或者一个 HTML 页面的输出。它提供了与标准 `jsp:include` 标签同样的功能，不过页面属性也能被表达为一个 JML 表达式这一条需除外。

语法:

```
<jml:include page ="relativeURL | <%= jmlExpression %>" flush =true" />
```

属性:

要获得有关包含属性和用法的更全面的信息, 请参看 Sun 公司 JSP 技术标准 1.1 版本。

例子: 下面的例子包含了 `table.jsp` 的输出, 即一个 HTML 表格的外观表达, 其中内容是基于查询字符串和请求属性中的数据。

```
<jml:include page="table.jsp?maxRows=10" flush="true" />
```

JML forward 标签

这个标签把请求导向另外的一个 JSP 页面, 一个 servlet 或者一个 HTML 页面。它提供了与标准 `jsp:forward` 同样的功能, 不过页面属性也能被表达为一个 JML 表达式这一条需除外。

语法:

```
<jml:forward page="relativeURL | <%= jmlExpression %>" />
```

属性:

要获得有关 `forward` 属性与用法更全面的信息, 请参看 Sun 公司 JSP 技术标准 1.1 版本。

例子:

```
<jml:forward page="altpage.jsp" />
```

JML print 标签

这个标签主要提供了与一个标准 JSP 表达式相同的功能: `<%=expr %>`。如果一个指定 JML 表达式或者字符串文字被计算了, 那么结果将输出到响应中。运用该标签时, JML 表达式就不必一定要嵌入到 `<%=...%>` 语法中, 但是一个字符串文字则必须用双引号引起来。

语法:

```
<jml:print eval = ' "stringLiteral" ' | "jmlExpression" />
```

属性:

`eval`——指定将被计算并输出的字符串或表达式。此属性是必需的。

例子: 下面的两个例子都输出了当前工资的值, 类型为 `JmlNumber`。

```
<jml:print eval="salary.getValue()" />
```

或者:

```
<jml:print eval="salary.getValue()" />
```

下面的例子输出一串字符串文字:

```
<jml:print eval=' "Your string here" ' />
```

JML plugin 标签

这个标签具有与标准 `jsp:plugin` 标签同样的功能。

要获得有关 `plugin` 属性、用法以及例子的更全面的信息, 请参看 Sun 公司 JSP 技术标准 1.1 版本。