

深入

雨阳隆春 等 编著

JSP

网络编程



清华大学出版社

<http://www.tup.tsinghua.edu.cn>

# 深入JSP网络编程

雨阳隆春 等 编著

清华大学出版社

(京)新登字 158 号

## 内 容 摘 要

本书较为深入、详实地讲述了怎样利用 JSP 构建完整的虚拟网站的全部技术与过程。全书共分为五个部分:第一部分是相关知识,包括 JSP 起源与构建 JSP 环境。第二部分是支持技术,包括 HTML 与 Dreamweaver。第三部分是继承者,包括指南、Java 基础、JSP 基本语法、Servlet 及其 API、内部对象、JSP Container、JSP 核心 API。第四部分是集成者,包括 JDBC、JavaBeans、Applet、XML。第五部分是综合应用。

本书语言简洁,由浅入深,既可作为广大 JSP 网络编程人员的参考指南,也可作为资深网站编程人员理想的参考用书。

版权所有,翻印必究。

本书封面贴有清华大学出版社激光防伪标签,无标签者不得销售。

## 图书在版编目(CIP)数据

深入 JSP 网络编程/雨阳隆春编著,一北京:清华大学出版社,2001.6

ISBN 7-302-04336-1

I. 深... II. 雨... III. 计算机网络—程序设计

IV. TP393.092

中国版本图书馆 CIP 数据核字(2001)第 034827 号

出版者:清华大学出版社(北京清华大学学研大厦 A 座,邮编 100084)

责任编辑:宋 韬

印刷者:北京市清华园胶印厂

发行者:新华书店总店北京科技发行所

开 本:787×1092 1/16 印张:32.75 字数:756 千字

版 次:2001 年 6 月第 1 版 2001 年 6 月第 1 次印刷

书 号:ISBN 7-302-04336-1/TP·2680

印 数:0001-5000

定 价:52.00 元

# 前 言

亲爱的读者,非常高兴您注意到本书。JSP 这种新兴的编程技术,说它新,真是当之无愧。1998 年 4 月,Sun 公司发布 JSP 0.9 规范,到现在,才有 JSP 1.2 草案。历史很短,然而成就却是巨大的。到网上看看,就知道它的影响有多大了。本书中前言部分涉及到的内容包括本书主要内容、特点、符号约定、适用对象,另外还有编写分工、致谢和参考文献。

## 主要内容

本书的书名是《深入 JSP 网络编程》,从概括书的内容、结构和特点这些角度看,我们不妨为本书拟定的另一个戏说书名——《三“心”二“意”不懂四“书”五“经”学 JSP 构建虚拟网站》,更能体现本书的主要内容、结构和特点。

那么,何谓三“心”二“意”,四“书”五“经”?

所谓三“心”就是 JSP 技术的三个核心概念,即:从整体技术角度把握,JSP 是服务端动态页面技术;从组成结构角度把握,JSP 是页面应用程序(Web Application);从语义实现角度把握,JSP 是一个容器(JSP Container)。即这三“心”的关系是从整体到局部,层层递进的关系。前面说的仅仅是概念上的三“心”,而基本语法、Servlet 和内部对象则是内容上的三“心”。二“意”是指 JSP 在内容上,既间接继承了 Java 的语法、类库,又直接继承了 Servlet 的类库、容器。事实上,JSP 与 Servlet 关系是如此的紧密,以至于可把 JSP 看成 Servlet 的特定简化实现。

四“书”是指 JSP 主要集成了四种技术:首先,JSP 是三层架构 C/S(客户/服务)技术,集成了 JDBC;其次,JSP 支持组件模型,集成了 Java Beans;再次,JSP 支持客户端动态页面技术,集成了 Applet;最后,JSP 支持开放技术,集成了 XML。说 JSP 是一个集大成者一点也不过分。五“经”是指本书共分为五个部分:第一部分是相关知识,包括 JSP 起源与构建 JSP 环境;第二部分是支持技术,包括 HTML 与 Dreamweaver;第三部分是继承者,包括指南、Java 基础、JSP 基本语法、Servlet 及其 API、内部对象、JSP Container、JSP 核心 API;第四部分是集成者,包括 JDBC、JavaBeans、Applet、XML;第五部分是综合应用。

JSP 内容众多,本书主要讲解构建网站的关键技术,不作而面俱到的讲解。重点讲了 JSP 语法、内部对象 Servlet、JDBC 和综合应用内容。

相信本书会使读者对 JSP 也有了一个比较清晰的整体印象,并学到非常实用的编程知识和技巧。

## 特点

入门要求低:阅读本书不要求一定会 HTML、Dreamweaver 和 SQL,本书用较少的篇幅系统地、结构化地介绍了上述内容。在不了解上述内容的情况下,读者会在阅读中不知不觉



已经循序渐进地掌握了上述知识。

**抓住主要矛盾:**本书抓住主要矛盾体现在两个方面。一是决不纠缠于与 JSP 不密切的知识。比如 JSP 环境构建、数据源的建立,如果在学习 JSP 之前还要求掌握 Linux 的操作,或者要求使用 Oracle 建立一个纯粹的 Java 驱动程序(JDBC),那就没有抓住主要矛盾。事实上,在哪个平台、哪个数据库支持下并不重要,重要的是怎样在该平台、数据库下使用 JSP。抓住主要矛盾的另一个方面是对主要内容绝对不惜笔墨,而对次要内容惜墨如金,如对 Servlet、JDBC 不惜一切代价争取写得更深入一些,而对 HTML 只讲述了表单、表格和框架,对 Dreamweaver、SQL 也是详略得当。这一点请读者明鉴。

**完整性:**本书基本上实现一个完整的网站所需知识,包括动态广告、时间日期服务、新闻、BBS、电子商务、留言板等等。附录 2、附录 3 包括了完整的 Servlet API 包,这对 JSP 的开发是非常有帮助的。JDBC API 也基本上做到了完整,包括所有的重要的类和界面。

**概括性:**本书每章的标题不仅对该章内容具有高度概括性,而且,内容也极具代表性和概括性。

## 符号约定

本书一共使用了三种符号,它们代表三个层次,它们是:

▼:代表顶层,有顶层目录的意味。

▽:代表中间层,有子目录的意味。

▶:代表底层,有文件的意味。

如:

▼元素(Elements)

    ▽脚本元素

        ▶声明

        ▶脚本片段

        ▶表达式

▽注释

    ▶输出注释

    ▶隐藏注释

.....

代表元素分为脚本元素、注释等几类,而脚本元素又分为声明、脚本片段和表达式三类。其余类推。另外,本书采用 Word 排版,有“注意”、“技巧”和“提示”等图文表示,并且对 JSP 语法、命令和程序代码用不同的底纹表示,请读者注意。

## 适用对象

本书不要求一定要了解 HTML、Dreamweaver 和 SQL,但最好对 Java 程序设计语言有相当的了解,熟悉其主要的类库(包)。了解 Java 是本书对读者的惟一要求,同时将发现了

解 Java,对学习、使用 JSP 是大有裨益的,用“如鱼得水,如虎添翼”来形容再恰当不过了。并且,也将看到 Java 这样的程序设计语言要在四五十页内讲清是根本不可能的。

## 编写分工

本书第 7 章、第 11 章和第 14 章聊天室部分由李云集编写,其余章节由雨阳隆春编写。全书由雨阳隆春统稿。此外,参加本书编写的还有李长传、李东阳、李义、王东、王三高、王品、张择、张阳雪、李春雨、杨新阳、张光明、赵问、赵长志、赵虚、许刚、许强、陈阵、陈加目、赵烁、陈金、陈非、孙拟、孙引元、崔飞、崔函、伍姓、伍冬夏、伍飞雪、伍欲和伍是等,在此一并致谢。

感谢博嘉科技的王松老师。首先感谢他的理解和支持,使本书能在较短的时间内以较高的质量面向读者。

感谢博嘉科技的刘清松等,是他们使本书内容更精致、版式更合理。

由于我们的水平和经验有限,书中难免有不准确、不贴切甚至错误的地方,欢迎读者批评并提出宝贵意见和建议。如有意见或建议,欢迎与我们联系:◆电话:(028) 5404228 ◆E-mail: [bojiakeji@163.net](mailto:bojiakeji@163.net)。我们的通讯地址是:四川大学西区建筑学院成都博嘉科技资讯有限公司,邮编:610065。

雨阳隆春  
2001 年 4 月

# 目 录

<b>第 1 章 JSP 起源、构建 JSP 环境 .....</b>	<b>1</b>
1.1 JSP 起源 .....	1
1.1.1 JSP 的含义 .....	1
1.1.2 JSP 起源与 Web 发展趋势 .....	1
1.1.3 JSP 的设计目标 .....	3
1.1.4 JSP 与三层结构模型关系 .....	4
1.1.5 JSP 与 ASP 的比较 .....	5
1.1.6 JSP 与 PHP 的比较 .....	8
1.2 构建 JSP 环境 .....	8
1.2.1 什么是 JSP 环境 .....	8
1.2.2 为什么构建 JSP 环境 .....	8
1.2.3 怎样构建 JSP 环境 .....	9
1.3 本章小结 .....	17
 <b>第 2 章 HTML 与 Dreamweaver .....</b>	<b>18</b>
2.1 HTML .....	18
2.1.1 TABLE(表格) .....	18
2.1.2 FORM(表单) .....	23
2.1.3 FRAMESET(框架) .....	30
2.2 Dreamweaver .....	34
2.2.1 表格 .....	34
2.2.2 表单 .....	37
2.2.3 框架 .....	39
2.2.4 用图层排版页面 .....	41
2.3 构建虚拟网站 .....	42
2.3.1 构思 JSP 虚拟网站 .....	43
2.3.2 构建 JSP 虚拟网站 .....	45
2.4 本章小结 .....	47
 <b>第 3 章 JSP 指南 .....</b>	<b>48</b>
3.1 Hello World .....	48
3.2 从组成的角度看 JSP .....	51
3.2.1 JSP 语法一览 .....	51
3.2.2 JSP 是 Web Application .....	52

---

3.3 本章小结·····	53
<b>第 4 章 Java 基础</b> ·····	54
4.1 Java 程序设计基础·····	54
4.1.1 Java 应用程序的组成·····	54
4.1.2 Java 程序设计基础·····	55
4.2 Java 面向对象程序设计·····	61
4.2.1 对象·····	62
4.2.2 类·····	62
4.2.3 打印杨辉三角·····	65
4.3 本章小结·····	67
<b>第 5 章 JSP 基本语法</b> ·····	68
5.1 元素·····	69
5.1.1 脚本元素·····	69
5.1.2 注释·····	74
5.1.3 指令·····	75
5.1.4 行为·····	87
5.2 Template Data·····	96
5.2.1 Template Data·····	96
5.2.2 Template Text·····	96
5.2.3 引用与转义·····	97
5.3 为虚拟网站加第一块砖·····	100
5.3.1 连接页头、主体和页脚·····	100
5.3.2 广告轮显·····	102
5.4 本章小结·····	105
<b>第 6 章 Servlet</b> ·····	106
6.1 Servlet 概述·····	106
6.1.1 什么是 Servlet·····	106
6.1.2 为什么要使用 Servlet·····	107
6.1.3 Servlet 与 CGI 相比有哪些优点·····	107
6.1.4 Servlet 与 JavaServer 体系结构的关系·····	108
6.2 Tutorial·····	110
6.2.1 编写 Servlet·····	110
6.2.2 编译 Servlet·····	114
6.2.3 运行 Servlet·····	114
6.2.4 Servlet 的基本执行流程·····	116

---

6.3 与客户端交互 .....	117
6.3.1 request 和 response .....	117
6.3.2 处理 GET 和 POST 请求 .....	118
6.4 Servlet 的生命周期 .....	122
6.4.1 初始化 Servlet .....	122
6.4.2 与客户端交互 .....	124
6.4.3 销毁 Servlet .....	124
6.4.4 Servlet 结束时处理 Service 线程 .....	125
6.5 存储客户端状态 .....	128
6.5.1 Session 跟踪 .....	128
6.5.2 Cookies .....	132
6.6 Servlet 的通信 .....	137
6.6.1 通过 RequestDispatcher 对象使用服务器上的其它资源 .....	137
6.6.2 在 Servlet 间共享资源 .....	140
6.6.3 从 Servlet 中调用其它 Servlet .....	142
6.7 运行 Servlet .....	143
6.7.1 在浏览器地址栏中直接键入 Servlet 的 URL .....	143
6.7.2 从 HTML 页面调用 Servlet .....	143
6.8 本章小结 .....	146
 <b>第 7 章 内部对象</b> .....	<b>147</b>
7.1 内部对象概述 .....	147
7.1.1 内部对象的功能 .....	147
7.1.2 内部对象的作用域 .....	148
7.2 JSP 内部对象详解 .....	149
7.2.1 Request 对象 .....	149
7.2.2 Response 对象 .....	158
7.2.3 Out 对象 .....	165
7.2.4 Session 对象 .....	169
7.2.5 Application 对象 .....	179
7.2.6 其它内部对象 .....	183
7.3 还想多了解点吗 .....	188
7.4 本章小结 .....	190
 <b>第 8 章 JSP Container</b> .....	<b>191</b>
8.1 编写支持实例 .....	191
8.2 命名约定 .....	200
8.3 编译 .....	201

---

8.3.1 编译 .....	201
8.3.2 预编译 .....	201
8.4 调试和错误处理 .....	202
8.4.1 调试 .....	202
8.4.2 错误处理 .....	202
8.5 翻译执行 .....	202
8.6 容器 .....	203
8.6.1 什么是 JSP 容器 .....	203
8.6.2 JSP 页面与 JSP 容器的关系 .....	203
8.6.3 JSP 页面实现类 .....	204
8.6.4 JSP 容器的行为 .....	205
8.7 留言板 .....	213
8.7.1 留言板的说明 .....	213
8.7.2 留言的处理 .....	213
8.7.3 查看留言 .....	217
8.7.4 错误处理 .....	218
8.8 本章小结 .....	220
<b>第 9 章 JSP 核心 API .....</b>	<b>221</b>
9.1 内部对象 .....	221
9.1.1 PageContext .....	221
9.1.2 JspWriter .....	227
9.1.3 一个实现实例 .....	233
9.2 Exceptions .....	234
9.2.1 JspException .....	234
9.2.2 JspTagException .....	235
9.3 JSP 页面实现对象与容器的联系 .....	236
9.3.1 JspPage .....	236
9.3.2 HttpJspPage .....	237
9.3.3 JspFactory .....	238
9.3.4 JspEngineInfo .....	239
9.4 计数器 .....	240
9.4.1 计数器的实现 .....	240
9.5 本章小结 .....	242
<b>第 10 章 JSP 对 JDBC 的集成 .....</b>	<b>243</b>
10.1 关系数据库标准语言 SQL .....	243
10.1.1 SQL 概述 .....	243

---



---

10.1.2	数据定义·····	244
10.1.3	数据查询·····	247
10.1.4	数据更新·····	252
10.2	JDBC 概述·····	255
10.2.1	JDBC 是什么·····	255
10.2.2	为什么有 JDBC·····	255
10.2.3	JDBC 与 ODBC 的比较·····	256
10.2.4	JDBC 的功能模型·····	257
10.2.5	JDBC 驱动程序的类型·····	258
10.3	Tutorial·····	259
10.3.1	建立数据源·····	259
10.3.2	加载驱动程序·····	260
10.3.3	建立连接·····	260
10.3.4	建立语句对象·····	260
10.3.5	添加数据到数据库·····	261
10.3.6	获取结果集合·····	264
10.3.7	数据处理·····	264
10.3.8	获得元数据·····	265
10.3.9	将处理结果写回数据库·····	267
10.3.10	关闭对象·····	269
10.3.11	处理异常和警告·····	270
10.4	建立数据源·····	271
10.5	Driver·····	273
10.6	DriverManager 和 DataSource·····	276
10.6.1	DriverManager 类·····	276
10.6.2	DataSource·····	278
10.7	Connection 和 PooledConnection·····	281
10.7.1	Connection·····	281
10.7.2	PooledConnection·····	284
10.8	Statement, PreparedStatement 和 CallableStatement·····	290
10.8.1	Statement·····	290
10.8.2	PreparedStatement·····	295
10.8.3	CallableStatement·····	301
10.9	ResultSet·····	308
10.10	ResultSetMetaData·····	321
10.11	SQLException 和 SQLWarning·····	323
10.11.1	SQLException·····	323
10.11.2	SQLWarning·····	325

---

---

10.12 新闻 .....	327
10.12.1 新闻显示 .....	327
10.12.2 新闻发布 .....	331
10.13 本章小结 .....	333
<b>第 11 章 JSP 对 JavaBeans 的集成 .....</b>	<b>334</b>
11.1 JavaBeans 概述 .....	334
11.1.1 JavaBeans 的属性 .....	335
11.1.2 JavaBeans 的方法 .....	335
11.1.3 JavaBeans 的事件 .....	336
11.2 在 JSP 页面中使用 JavaBeans .....	336
11.2.1 <jsp:useBean... /> 标记 .....	336
11.2.2 <jsp:setProperty> 标记 .....	337
11.2.3 <jsp:getProperty> 标记 .....	339
11.2.4 编写自己的 Bean .....	345
11.2.5 通用数据库 Bean .....	345
11.2.6 购物车 Bean .....	347
11.3 本章小结 .....	350
<b>第 12 章 JSP 对 Applet 的集成 .....</b>	<b>351</b>
12.1 <jsp:plugin> 行为 .....	351
12.2 时钟、日期 Applet 的实现 .....	353
12.2.1 时钟 Applet .....	353
12.2.2 日期 Applet .....	356
12.3 本章小结 .....	358
<b>第 13 章 JSP 对 XML 的集成 .....</b>	<b>359</b>
13.1 JSP 页面的 XML 语法 .....	359
13.1.1 XML 的几个基本概念 .....	359
13.1.2 JSP 页面的 XML 语法 .....	360
13.1.3 实例 .....	364
13.2 标记扩展 .....	365
13.2.1 taglib Directive .....	365
13.2.2 标记库描述器及其格式 .....	366
13.3 本章小结 .....	367
<b>第 14 章 网站建设 .....</b>	<b>368</b>
14.1 BBS .....	368

---

---

14.1.2	注册	370
14.1.3	登录	375
14.1.4	查询	378
14.1.5	版面显示	383
14.1.6	文章标题显示	385
14.1.7	文章显示	387
14.2	电子商务	389
14.2.1	首页	389
14.2.2	进入书屋	395
14.2.3	购物	396
14.2.4	购物车显示	406
14.2.5	退货系统——部分退回	410
14.2.6	全部退回	412
14.2.7	结帐系统	414
14.2.8	数据库更新	419
14.3	聊天室	421
14.3.1	聊客信息管理	421
14.3.2	聊天室的实现	431
14.4	本章小结	442
<b>附录 1</b>	<b>Tomcat 安装汇总</b>	<b>443</b>
1.1	资源下载	443
1.2	Tomcat 在 Windows NT 4.0, Windows 2000 下的安装	443
1.3	Tomcat 在 Redhat 下的安装并与 apache 相连	444
1.4	Tomcat 在 Unix 下的安装	445
<b>附录 2</b>	<b>Servlet API——javax.servlet 包</b>	<b>447</b>
2.1	javax.servlet Class GenericServlet	447
2.2	javax.servlet Interface RequestDispatcher	448
2.3	javax.servlet Interface Servlet	450
2.4	javax.servlet Interface ServletConfig	452
2.5	javax.servlet Interface ServletContext	453
2.6	javax.servlet Class ServletException	460
2.7	javax.servlet Class ServletInputStream	461
2.8	javax.servlet Class ServletOutputStream	462
2.9	javax.servlet Interface ServletRequest	463
2.10	javax.servlet Interface ServletResponse	469
2.11	javax.servlet Interface SingleThreadModel	473

---

---

2.12	javax.servlet Class UnavailableException .....	474
<b>附录 3</b>	<b>javax.servlet.http 包 .....</b>	<b>476</b>
3.1	javax.servlet.http Class Cookie .....	476
3.2	javax.servlet.http Class HttpServlet .....	480
3.3	javax.servlet.http Interface HttpServletRequest .....	486
3.4	javax.servlet.http Interface HttpServletResponse .....	493
3.5	javax.servlet.http Interface HttpSession .....	502
3.6	javax.servlet.http Class HttpSessionBindingEvent .....	507
3.7	javax.servlet.http Interface HttpSessionBindingListener .....	508
3.8	javax.servlet.http Interface HttpSessionContext .....	508
3.9	javax.servlet.http Class HttpUtils .....	509

# 第 1 章 JSP 起源、构建 JSP 环境

## 1.1 JSP 起源

### 1.1.1 JSP 的含义

JSP 是“Java Server Pages”的缩写,直译过来是“基于 Java 的服务端动态页面技术”。这是 Sun 公司近年来的力作。自 1990 年开始设计 Java,其后推出、应用、推广到现今不过 10 年光景,Java 已经获得极其崇高的地位,Java 被称为一种划时代的程序设计语言便是最有力的证明。事实上,Java 早已以其简单、面向对象、平台无关等特性征服了世界。在 Internet 应用上,更是如日中天。Sun 也不断推出基于 Java 的新技术巩固其地位与优势,先后推出了 Java Applet、JavaScript、Servlet 等技术及其产品。JSP 技术正是这种战略的延伸,是 Sun 保持 Java 地位与优势的需要。基于 Java 是 JSP 一切优秀品质的根本物质基础。因为 Java 是编译(在 JSP 中称翻译更恰当)解释执行的,所以 JSP 也就摆脱了 VB-Script、Perl 等脚本语言纯粹的解釋执行带来的低效率问题。尽管在 JSP 中 Java 仍被称为脚本语言。因为 Java 是平台无关的,所以 JSP 是跨平台的。基于 Java 是 JSP 区别 ASP、PHP 等服务端动态页面技术的重要特征。JSP 是服务端技术,所以 JSP 文档要先经服务端翻译、解释、执行,才能得到客户端浏览器能识别的 HTML 文档。这是当前 Web 技术的潮流,是解决客户端浏览器兼容性问题的客观需要。服务端技术是 JSP 区别于 JavaScript、VBScript 等客户端动态页面技术的重要特征。JSP 是一种页面技术,而现在 JSP 只实现了 HTTP 协议,只能应用于 Web 页面。这是 JSP 区别于 Servlet 的重要特征。JSP 是一种动态技术,这是 JSP 基于 Java 的自然延伸。我们为什么在页面中引入程序设计语言,为的就是实现动态交互,这是我们孜孜以求的目标。因此,JSP 就是基于 Java 的服务端动态页面技术。

### 1.1.2 JSP 起源与 Web 发展趋势

JSP,诞生于名门望族 Sun 家中。说起其身世,就不得不说说江湖中的一段恩怨——“嫁衣”(浏览器标准)之争。十年前,Web 诞生。当时这个小姑娘样于还是相当难看的,只有头和脚(C/S 两层架构),而且目光呆滞(静态),粗布白衣(文本)。即便是这样,她仍然获得了不少人的青睐,很快就鹤立鸡群,其他的前辈、兄长只有艳羡的份。小姑娘目光呆滞,这可是大问题。想想吧,跟她打招呼,她却不理睬你,于是就有了 CGI 脚本。还有粗布白衣,日久生厌。怎么办呢?到了 1993 年,小姑娘有了第一件漂亮的衣裳——图形

用户界面的 Web 浏览器,一下子吸引了全球人的目光,也吸引了各路英雄豪杰(IT 厂商)。小姑娘长大了,花(标准)落谁家呢?当然要看谁做的“嫁衣”功能更强大,外表(界面)更漂亮(友好)。各路英雄豪杰为争夺“嫁衣”,文争武斗,搅得江湖浩劫丛生。先是 Netscape 凭 Navigator 独领风骚两三年。Navigator 功能强大,支持更多的 HTML 扩展标记,成为当时事实上的标准。然而随着 Microsoft 的强势杀人,其 IE 支持比 Navigator 更多、更好的扩展 HTML 标记,与 Netscape 分庭抗争。此时,二者的 HTML 部分标记是不兼容的。Netscape 深感压力,联合了 Sun,Java 阵营渐成。Sun 大有来头,Java 就是其镇家之宝,当年也是春风得意。二者将 Sun 的独门武功 Java 集成于 Navigator 之中,使其支持 Java Applet。此时浏览器有了一定的交互能力,界面也更友好。Microsoft 不愧一代枭雄,独力对抗 Java 阵营亦略占上风。不久便力推 XML 以制衡 Sun 的技术优势。Microsoft 原想利用 XML 这门绝世武功将 Sun 打发出局,然而意想不到的的是这两门武功并不相生相克,反而相辅相成。二者没分出高下,自是不会善罢甘休。不久,Sun、Netscape 宣称自己的浏览器支持 JavaScript。而此时 Microsoft 也没有闲着,他想要使本门的武功发扬光大,必须发展创新。那么哪门武功能够和值得去创新呢?很快 VB 进入了 Microsoft 的视线,VB 是解释型的语言,非常适合网络上灵活多变的请求,于是全情投入 VBScript 的修炼。寒暑易节,江湖风传 JavaScript 如何厉害,让 Microsoft 大为震惊,一边加快了 VBScript 的修炼,一边找来 JavaScript 研究,这样,到 VBScript 大功告成时,JavaScript 功力亦相当深厚。二者明争暗斗,你来我往,互有攻守。一般情况下,Sun 总能在技术上快 Microsoft 半分,但 Microsoft 总能把界面做得更友好,因此,二者大比武总是不能决出胜负。这可苦了芸芸众生,由于二者不和,我等只好今天学这个,明天又学那个,谁解其中味!Microsoft 和 Sun 似乎都意识到这点,同时也看到 C/S 两层架构的弊端,二者各自根据自己的武功套路,分别演化出 ASP 和 Servlets。同样二者互有优劣,ASP 更简易,Servlet 功能强大但较复杂,Microsoft 以亲民政策略占上风。Sun 不久就意识到问题之所在,毕其功于一役,推出 JSP。JSP 支持绝大多数服务器,同时改善了人机界面,生成动态网页更简易,大有一统江湖之势,不知这次 Microsoft 以何为对?

上文“戏说”,只是想让读者了解这段发展历史,从中感受发展趋势。事实上,下面的推论也并不需要了解太多史实细节。推论如下:

(1) 当今时代,谁也不能在技术上领先对手很长时间。在 IT 业,6~18 个月是一个比较客观的平均技术代时间。谁落后对手太长时间,通常他将会被淘汰。这与几千年前是不同的,想想我们的四大发明领先其它国家多长时间。主要原因是现在与以前的资讯传播方式不一样了。

(2) 既然谁也不能领先,那么其它方面的因素,如人机界面等将会起到决定性的作用。在人机界面方面,Sun 有很大改进,但仍需加强。当您为运行 JSP、Servlet 配置环境的时候,也许就会有些感受。当您直接利用服务器调试程序的时候,您的感受可能会更深一些。

(3) 竞争从最初的浏览器之争全而升级为当今先进技术综合应用的竞争,从可视化(Visual)到组件技术(COM)再到 C/S 体系架构。当今先进技术无一不在互联网上一展身手,这也表明了互联网的勃勃生机,预示着互联网是当今最有前途的技术之一。

(4) 对同一种技术,有很多实现手段(方法)。如客户端动态页面技术,Sun 以 JavaS-



cript 实现,Microsoft 以 VBScript 实现。又如服务端动态页面技术,Sun 以 Servlets、JSP 实现,Microsoft 以 ASP 实现。当然还有其它 CGI 技术,如 PHP。

(5) 虽然有多种实现手段,但它们是有共同点的。对客户端动态页面技术,Sun 和 Microsoft 都使用了脚本语言,即解释型的语言。对服务端动态页面技术,都采用 C/S 三层体系架构,在底层都封装(Sun 使用 JDBC,Microsoft 使用 ODBC)了数据源,在中间层都使用了组件技术(Sun 使用 JavaBeans,Microsoft 使用 COM)封装了实现逻辑,以及提供事务处理环境以支持高伸缩性、分布式事务处理,等等,不一而足。“共同”就意味着“必然”。

(6) 上一点的推论,共同是有时间烙印的。时代决定了可用的资源,当前技术不能超出可用的资源。

(7) 因为有多种实现手段,所以通常互不兼容,但又兼收并蓄。例如,微软的 IE 既支持 VBScript 又支持 JavaScript(JScript)。

(8) 解决兼容性问题推动当前浏览器乃至其它技术的重要动力。

我们从更高角度看,抛开二者的商业竞争因素、技术上的个体差异,事实上,每一项新技术的应用,都有一定的目的,即都要解决一个矛盾。在 Web 产业中,有三个基本的矛盾:一是性能(功能)问题,二是界面问题(简易性),三是兼容性问题。

Web 发展至今,请容许我们大胆的把它分为三个阶段。第一阶段是二层架构服务端页面技术,目标是解决基本的交互问题,特征是使用的人比较少,界面不够友好,但有能够满足需要的交互能力;第二阶段是二层架构客户端动态页面技术,目标是解决界面不够友好的问题,特征是使用的人增多,客观上要求界面更加友好;第三阶段是三层架构服务端动态页面技术。第一、二阶段的兼容性问题暴露出来了,这时兼容性上升为主要矛盾。

### 1.1.3 JSP 的设计目标

JSP 技术让动态网页更易编写,功能更强,可移植性更好。概括起来,JSP 的设计目标主要有以下几点:

#### 1. 一次编写,处处运行

JSP 技术是完全的与平台无关的设计,包含它的动态网页与底层的服务组件设计。可以在任何平台下编写 JSP 网页并且在任何支持 JSP 的系统上执行。

也可建立自己的组件,并在 JSP 中使用。目前主要是 JavaBeans 和 Java Servlet,而它们都是跨平台的。

#### 2. 加强组件能力

JSP 技术以 Java 的组件模型 JavaBeans 加强了组件的使用能力。

这不但省去了开发的时间,而且还可以将网页页面设计和商业逻辑分开,有助于快速开发和简单维护。

#### 3. 作为 Java 企业平台的门户

JSP 高度整合了 Java 企业平台部分(Java 专注于企业应用方面的平台技术,如 JDBC、

JNDI、JINI 等)。可以利用 Java 的企业级 API 开发企业的各种需求系统,而使用 JSP 作为这些技术的前端。当需要升级应用程序时,只需升级组件与动态页面部分,而这些都存在于服务器上,所以修改服务器上的资源,所有客户端的显示都会跟着改变。

#### 4. 更容易建立动态页面

JSP 就是用标准 HTML 语法混合自身语法标记,就是如此简单,不需要有使用 Java、C++ 等程序设计语言的能力与经验。可以这么说,它的出现也正是要实现简单容易的开发页面的需要,否则它与 Java Servlet 相比没有任何存在的理由。

### 1.1.4 JSP 与三层结构模型关系

JSP 技术可以用来建立诸如时钟、计数器这样的简单页面,也可以用于面向企业的三层结构模型的中间层。三层结构模型如图 1-1 所示。JSP 是可自定义三层式应用程序接口的、服务端的页面技术,并且提供了弹性的、可升级的应用程序开发环境。

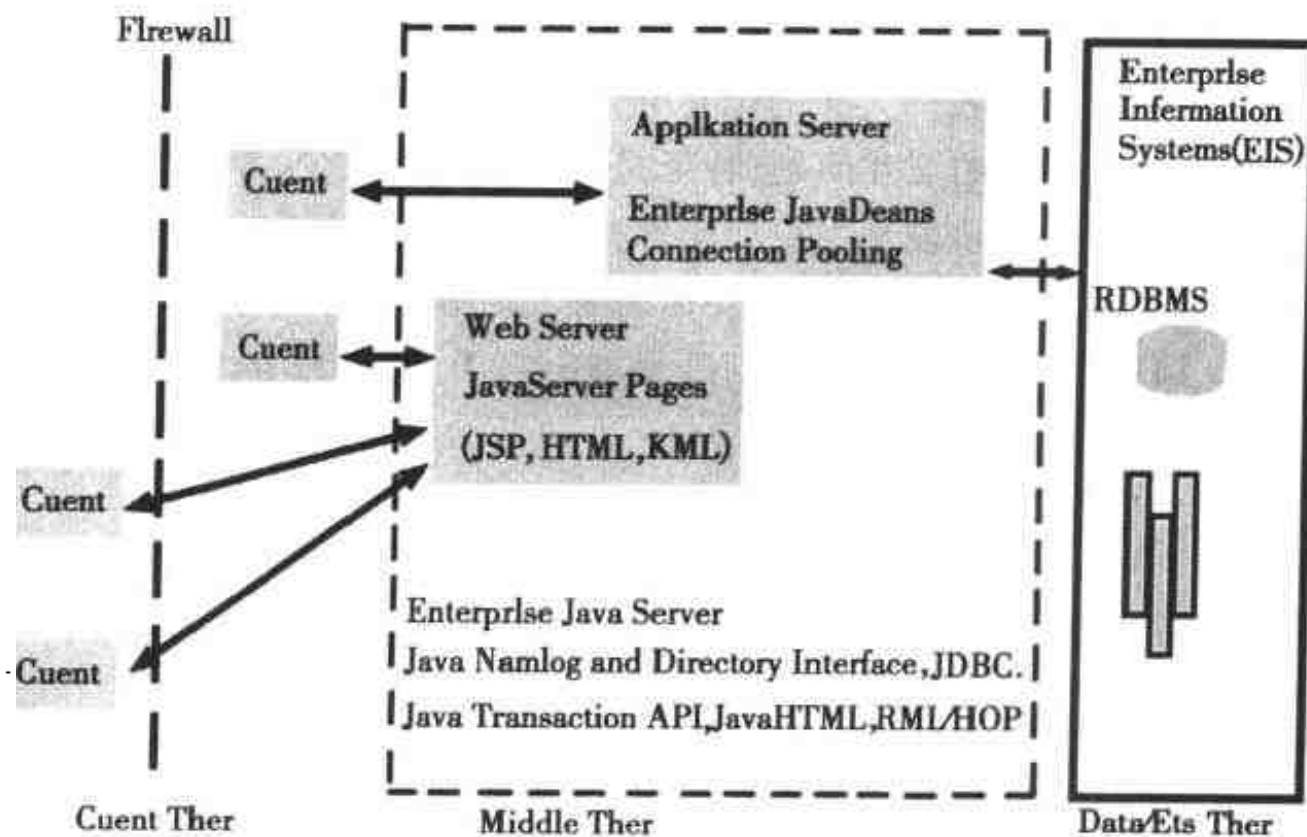


图 1-1 C/S 三层式结构模型

#### 1. 客户层

JSP 页面与基于 Web 的客户通信,包括基于 HTML 的浏览器。JSP 页面管理浏览器的交互,典型的以 HTML 格式发送和接收请求、信息。

以标准的 HTML 格式与客户端交互,使得 JSP 是跨浏览器的。这是服务端动态页面技术的一大优势。同时降低了对浏览器的要求,浏览器无需支持插件,无需支持 ActiveX。当然也降低了客户机的要求。

#### 2. 中间层

JSP 页面几乎可以置于任何网络服务端与应用程序服务端。页面拥有对 Java 应用程

序接口 API 和服务 service 等广大范围的访问权,包括实现数据库访问的 JDBC 技术、基于 Java 技术实现 e\_mail 应用的 JavaMail,实现事务处理服务的 Java 事务处理应用程序接口 Java Transaction API。

这使得 JSP 拥有极强的企业应用能力,尽管它的目标仅仅是作为 Java 企业平台的门户。

### 3. 数据/企业应用层

中间层的企业级 JavaBean 组件管理访问数据/企业应用层提供的企业资源,处理安全、授权、事务完整性、连接池和数据缓存等事务。企业级的 JavaBean 及其容器封装了需要高性能、高安全性的企业级应用程序的复杂逻辑。

## 1.1.5 JSP 与 ASP 的比较

一眼看上去,JSP 和 ASP 技术有许多的相似点。二者都将建立交互式页面作为基于 Web 的应用程序的一部分。在某种程度上,二者都能通过在页面中调用组件达到分离程序设计逻辑和页面的目的。还有很多相似点,它们不同在哪儿呢?我们从三个角度看:

### 1. JSP 技术,开放的软件设计方法

JSP 和 ASP 技术的最大不同点在于它们自身的软件设计方法。JSP 技术被广泛的工具、服务器和数据库提供商设计成与平台、服务器无关,与之相反,ASP 主要依赖于 Microsoft 的技术。

#### 1) 平台、服务器无关性

JSP 技术建立在“一次编写,处处运行”的 Java 程序设计思想体系上,避免了与单独的某个平台或提供商紧密联系。JSP 页面能运行在任意的 Web 服务器上,并被来自众多提供商的大量的不同工具支持。

因为 ASP 使用 ActiveX 控件作为它的组件,所以 ASP 技术基本上被限制在 Microsoft 的基于 Windows 的平台上,并基本上作为 Microsoft IIS 的一部分提供给用户。因为 ActiveX 对象具有平台特性,ASP 技术不可能轻易地应用在如此众多的不同 Web 服务器上。尽管通过第三方接口产品,ASP 技术能在其它平台上访问组件并与其它服务器交互,但那个平台必须有相应的 ActiveX 对象,如果没有,必须使用一个桥以支持该平台。

#### 2) 开放的开发过程,开放的源代码

1995 年,Sun 同国际 Java 组织合作开发和修订了 Java 技术与规范,对 JSP 的开发,Sun 也使用同样的方法,与 Sun 一起致力于 JSP 技术开发的有授权工具提供商(如 Macromedia),结盟公司(如 Apache 和 Netscape)、最终用户、顾问团和其他个人(公司)的技术上的合作。不仅如此,Sun 还主动将最新版本的 JSP 和 Java Servlet(JSP 1.1 和 Java Servlet 2.2)的源代码发放给 Apache,促使 JSP 和 Apache 紧密发展。

JSP 应用程序界面毫无疑问从这种开放组织的合作中获益,并将继续获益。与之相反,ASP 技术是 Microsoft 倡议的,在封闭的,专有的开发过程中发展。

## 2. 开发者的角度

JSP 和 ASP 技术都允许开发者从页面中访问组件以分离页面和商业逻辑,ASP 支持 COM 模型,JSP 技术提供基于 JavaBeans 技术和 JSP 标记的组件。

### 1) JSP 标记的可扩展性

尽管 JSP 和 ASP 都使用标记和脚本的组合来建立动态页面,然而对任何一个页面作者来说,JSP 技术使得开发者能够有效地扩展 JSP 标记,开发者可以建立自定义标记库,所以页面作者可使用 XML 样式的标记获得更多的功能而减少对脚本语言的依赖。有了自定义标记,开发者可使页面作者从建立页面的复杂性中解脱出来,而在页面作者熟悉的范围内扩展关键功能。

### 2) 跨平台重用性

开发者也非常关注可重用性。JSP 组件,包括企业级 JavaBeans、JavaBeans 或自定义 JSP 标记,都是跨平台可重用的。例如一个访问传统数据库的企业级 JavaBeans 组件可为 UNIX 和 Windows 平台上的分布式系统提供服务。并且 JSP 技术的标记扩展功能提供给开发者一个简便的、XML 风格的界面使页面设计者的功能包可在整个企业应用中共享。基于组件模型加速了应用程序的开发,因为它使开发者能够使用轻量级的组件快速构建应用程序原型,然后集成另外的功能使其变得实用。另外,它可以将一个组织中重复的工作封装为一个 JavaBeans 或企业级 JavaBeans 组件。

### 3) 继承了 Java 的优点

JSP 技术使用 Java 语言作为其脚本语言,与其对应的 Microsoft 的 ASP 使用 VBScript 或 Jscript。Java 语言是成熟的、强大的、可扩展的程序设计语言,远优于基于 BASIC 的脚本语言。例如 Java 语言提供比解释型的 VBScript 或 Jscript 好得多的执行性能。

Java 使开发人员在其它方面的工作也变得简单容易,例如,在 Windows NT 系统被怀疑可能会崩溃时,Java 能有效的防止系统崩溃;通过延迟应用程序展开(因指针错误),防止内存泄漏。Java 语言在内存管理方面对开发人员也是非常有帮助的。还有,JSP 为实时应用程序提供了健壮的异常处理机制。

### 4) 维护简单

脚本语言对小的应用程序是可行的,但是对大型的、复杂的应用程序显然是勉为其难。相比较,Java 语言是结构化的,它能轻易的建立和维护大型的、模块化的应用程序。

JSP 技术的核心在于组件模型,它使得不改变逻辑而改变内容,或改变逻辑而不改变内容变得轻而易举。

企业级 JavaBeans 体系封装了企业应用逻辑,如数据库访问、安全、完整性事务处理。并使自己与应用程序隔离,安全、维护简单是其明显的优点。

JSP 技术是一种开放的、跨平台的体系。Web 服务器、平台和其它组件都可轻松的升级或交换而不影响基于 JSP 的应用程序。这使得 JSP 非常适合需要升级而又要求保持不变的实时 Web 应用程序。

## 3. 企业应用可伸缩性

J2EE 是为开发多层企业级应用程序设计的 Java 体系。作为 J2EE 的一部分,JSP 页

面可访问所有的 J2EE 组件,包括 JavaBeans、企业级 JavaBeans 组件和 Java Servlet。JSP 页面实际上是被编译成 Servlet,所以它有服务端的、可伸缩的 Java 应用程序的所有优点。J2EE 平台容器可管理企业级应用程序的复杂逻辑,包括事务处理管理和池缓冲资源。

JSP 页面可访问所有 J2EE 的标准服务,包括:

- (1) Java 命名和目录界面 API(JNDI API)
- (2) 与关系数据库通信(JDBC API)
- (3) 支持基于 Java 的邮件和信息应用的类(JavaMail)
- (4) Java 信息服务(JMS)

通过 J2EE,JSP 有许多方法与企业级系统交互,可满足不同层次的企业应用需求。

#### 4. 支持广泛

JSP 技术是在 Java 团体开发过程中开发的,它有来自工具、Web 服务器和应用程序服务器提供商的广泛支持,这使得用户及其伙伴可选择最好的方法、工具应用于他们特殊的应用程序中以保护他们的代码和智力投资。

#### 5. ASP 与 JSP 的比较

ASP 与 JSP 的详细比较如表 1-1 所示。

表 1-1 ASP 与 JSP 的详细比较

特性	ASP 技术	JSP 技术
Web 服务器	Microsoft IIS 或 Personal Web Server	任何 Web 服务端,包括 Apache, Netscape 和 IIS
平台	Microsoft Windows	大多数流行平台,包括 Solaris 操作环境, Microsoft Windows, Mac OS, Linux 和其它 UNIX 平台实现
可重用、跨平台组件	无	JavaBeans, 企业级 JavaBeans, 自定义 JSP 标记
防止系统崩溃的安全措施	无	有
内存漏洞保护	无	有
脚本语言	VBScript, JScript	Java
自定义标记	无	有
数据库向前兼容性	有(COM)	有(JDBC API)
集成数据源能力	与任意兼容 ODBC 数据库协同工作	与任意兼容 ODBC 和兼容 JDBC 技术的数据库协同工作
组件	COM 组件	JavaBeans, 企业级 JavaBeans, 或者可扩展 JSP 标记
广泛工具支持	是	是

### 1.1.6 JSP 与 PHP 的比较

JSP 与 PHP 的简单比较如表 1-2 所示。

表 1-2 JSP 与 PHP 的简单比较

特性	PHP	JSP
内容、逻辑独立性	部分	完全
目标用户	程序员	Web 开发者, 页面作者
第三方工具提供商的工业支持	否	是
页面语言	Perl	Java
第三方组件扩展	否	是

## 1.2 构建 JSP 环境

### 1.2.1 什么是 JSP 环境

JSP 环境有两层含义:从宏观上说,在什么情况下,用基于 Java 的 JSP 语法书写的页面能在客户端浏览器上被正确地显示,也就是 JSP 技术实现的物质基础,称为 JSP 环境。没有这个环境,JSP 技术的三层架构就不能实现。例如,没有 JDBC、ODBC 等提供后台数据库处理,没有动态连接技术提供与 Web server 直接相连,那么 JSP 充其量不过是传统的 CGI 罢了。其中最重要的是一个将 JSP 文档转换为 HTML 的服务程序,通常是一个服务端应用程序。在 ASP 技术中,Microsoft 提供了 IIS、PWS。JSP 中,Sun 及其伙伴公司、第三方公司提供了 JSDK、JRun、Tomcat、WebSphere 等。

从微观上说,JSP 技术中服务端应用程序能正常运行并提供需要的服务所依赖的环境变量,也称为 JSP 环境。具体地讲,环境变量包括路径变量、类路径变量、服务名称、服务端口等等。这些环境变量保证服务端应用程序正常工作并提供我们需要的服务。例如,我们需要更改服务端口或者增加一个服务。

### 1.2.2 为什么构建 JSP 环境

为实现 JSP 这种服务端动态页面技术,需要构建 JSP 环境。为什么这么说呢?我们知道 JSP 是三层架构,而浏览器只能识别 HTML 格式的数据和一些基本的对象如按钮、文本框等,浏览器根本不能识别 JSP 文档,因此需要 JSP 环境实现转换。

JSP 是基于 J2SE 和 J2EE 平台上的,自身没有编译器,然而 JSP 的执行需要先翻译



(预编译)成 Servlet。因此需要先安装 JDK,然后在服务端定位 JDK 中的编译器,以取得其服务。翻译(预编译)还需要类库的支持,因此还需要类库路径。此外,为了保持灵活性和有效性,还可能需配置服务端口,添加服务等。

### 1.2.3 怎样构建 JSP 环境

作为本书简单性、抓住主要矛盾的第一个体现,建议学习环境是 Pwin98 + Tomcat3.1。Pwin98 简单易用,可以有效降低学习门限从而抓住主要矛盾,迅速掌握 JSP。选中 Tomcat 完全因为它是一个“升值股”。首先 Tomcat3.1 支持 JSP1.1 和 Servlet2.2 规范;其次它由著名的 Apache 开发;再次 Sun 推荐使用;最后它是免费的。另外 JSDK 也是一个相当不错的作学习之用的服务器,并且它的安装及其出现的问题具有一定的典型性。下面介绍这两种服务器的安装。为体现确有需要在其它操作系统下使用 JSP 的读者的利益,事实上,商业应用几乎不会用到 win9x 平台,因此我们以附录的形式给出 Tomcat 在其它平台下的安装与配置,参见附录 1。下面使用软件的获取也请参见附录 1。

#### 1. 构建步骤

Tomcat 和 JSDK 有相似的构建步骤:

- (1) 安装 JDK,配置 JDK 环境。
- (2) 安装 Tomcat 或 JSDK,配置相应的环境。
- (3) 启动服务器。
- (4) 确信服务器正常运行,配置服务器。

#### 2. Tomcat 的安装

##### 1) 安装 JDK,配置 JDK 环境

JDK 目前发布的是 1.3 版本,支持 Java 2,可提供完整的 Java 开发环境。然而 Tomcat 用到 JDK 的仅仅是它的编译器 javac。JDK1.3 安装过程很简单,请按提示安装。安装完后修改 Autoexec.bat 如下:

- (1) `PATH = %PATH%; 安装路径 \ bin;`
- (2) `set CLASSPATH = 安装路径 \ Lib \ tools.jar`
- (3) `set JAVA_HOME = 安装路径`

例如本书 JDK 安装路径是 C:\Sun\Jdk,因此修改 Autoexec.bat 如下:

```
PATH = %PATH%; C:\Sun\Jdk\bin;
set CLASSPATH = C:\Sun\Jdk\Lib\tools.jar
set JAVA_HOME = C:\Sun\Jdk
```

##### 2) 安装 Tomcat

安装 Tomcat 其实是一个拷贝过程,需要做的事情就是解压缩 Tomcat。如果需要,可以改名原目录,或者新建一个目录,然后将 Tomcat 拷贝或移动到其中。再次修改 Autoexec.bat 如下:

```
set TOMCAT_HOME = Tomcat 所在目录路径
```

例如本书 Tomcat 路径是 C:\Sun\Tomcat, 修改 Autoexec.bat 如下:

```
set TOMCAT_HOME=C:\Sun\Tomcat
```

### 3) 启动 Tomcat

Tomcat3.1 启动服务的文件是 startup.bat, 位于 Tomcat 安装路径 \bin 目录下。例如本书是 C:\Sun\Tomcat\bin\startup.bat, 通常是可以成功启动服务的。正常现象是应当有两个 MS-DOS 窗口出现。后出现的标志服务已启动的 MS-DOS 窗口如图 1-2 所示。

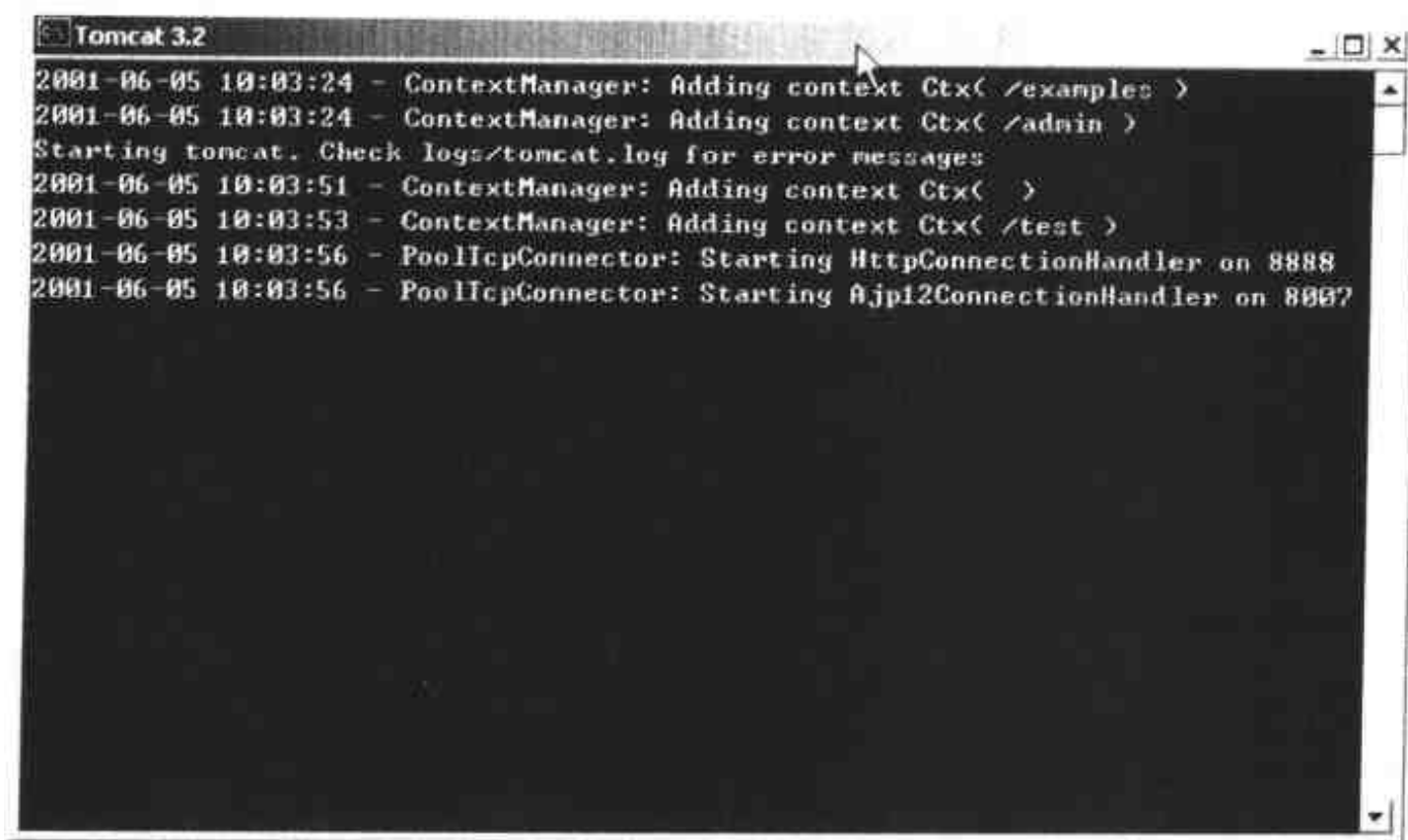


图 1-2 Tomcat 服务启动正常现象

留意一下该窗口中的信息, 包括服务路径信息、Tomcat 安装路径信息、加载类路径信息。

如果使用一台计算机既作服务器又作客户机, 那么启动的将是本地服务。通常, 获得本地服务采用如下形式: `http://127.0.0.1:port/`, 其中 port 在这儿是 8080。如果使用了专门的服务器, 需要知道该服务器的地址和端口号。本书只讲述本地服务这种情况, 在 IE 中键入地址和端口号 `http://127.0.0.1:8080/` 或者 `http://localhost:8080/`, 应该可以看到图 1-3 所示页面。至此, 您已经完全获得 Tomcat 的 JSP 服务了。

如果出现因环境变量空间太小无法启动服务, 请参见 JSDK 的安装提示。

### 4) 配置服务器

Tomcat 的配置比较简单, 其配置文件都在 Tomcat 安装路径 \conf 目录下, 例如本书是 C:\Sun\Tomcat\conf。主要配置文件是 server.xml。节选片段并整理如下:

```
<ContextManager debug="0" workDir="work" >
.....
<ContextInterceptor className="org.apache.tomcat.context.WorkDirInterceptor"/>
<ContextInterceptor className="org.apache.tomcat.context.WebXmlReader"/>
.....
<Connector className="org.apache.tomcat.service.SimpleTcpConnector">
```

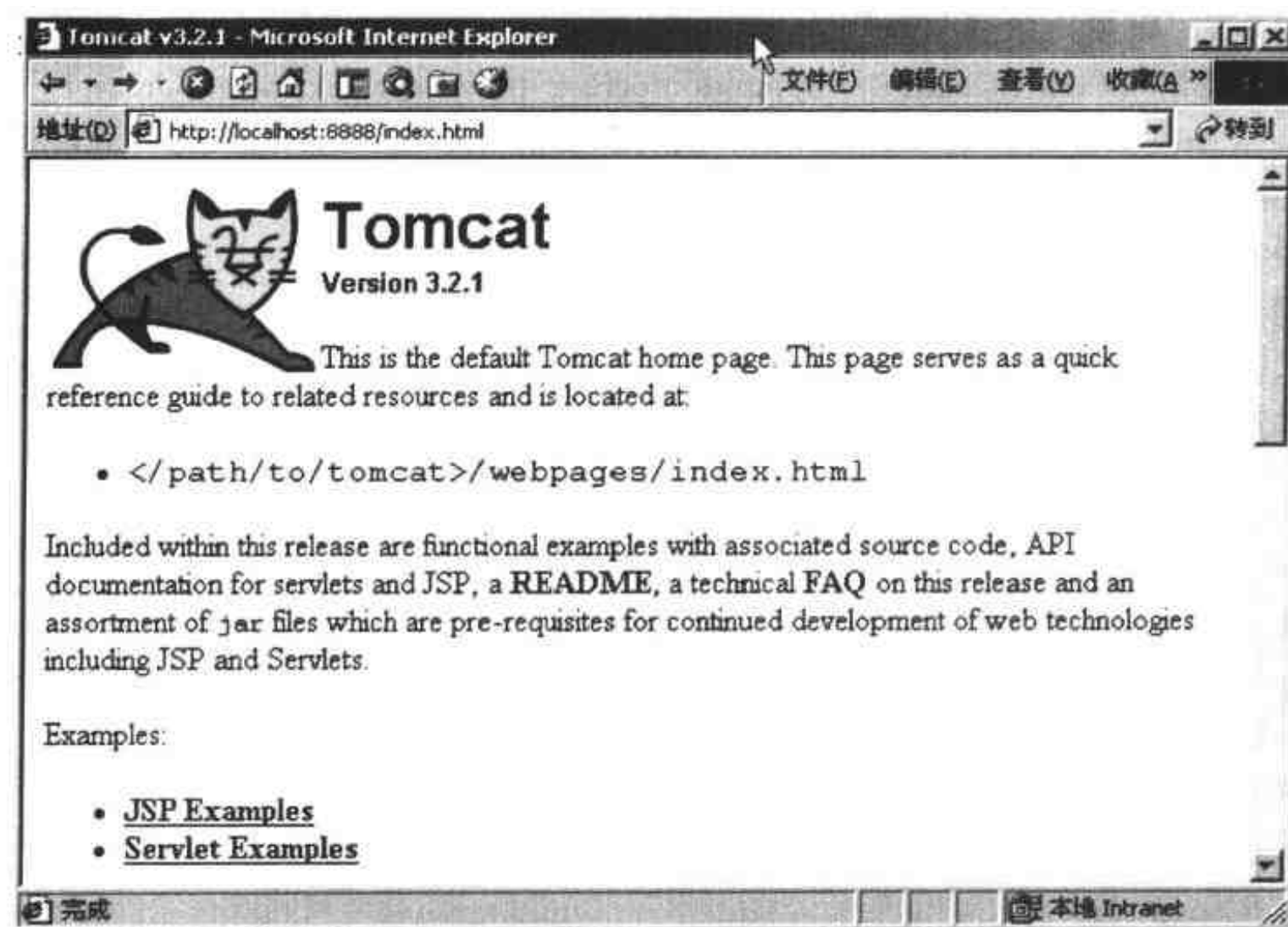


图 1-3 标志获得 Tomcat JSP 服务的页面

```

<Parameter name="handler"
value="org.apache.tomcat.service.http.HttpConnectionHandler"/>
<Parameter name="port" value="8080"/>
</Connector>
.....
<!-- example - how to override AutoSetup actions -->
<Context path="/examples" docBase="webapps/examples" debug="0"
    reloadable="true" >
</Context>
<!-- example - how to override AutoSetup actions -->
<Context path="" docBase="webapps/ROOT" debug="0" reloadable="true" >
</Context>
<Context path="/test" docBase="webapps/test" debug="0" reloadable="true">
</Context>
</ContextManager>

```

#### (1) 更改服务监听端口号

找到 `<Parameter name="port" value="8080"/>` 这一行, `value` 属性的值即是端口号, 可改变它, 比如将端口号改为 9000。



注意: 端口号一定是没被占用的, 否则将引起冲突。

## (2) 添加服务

添加类似 `<Context path="/examples" docBase="webapps/examples" debug="0" reloadable="true"> </Context>` 这样的行就可以新增一个服务,其中 path 表示该服务的 URL 路径,docBase 表示映射到文件系统的路径,debug 表示是否允许调试,reloadable 表示是否自动重新加载 Servlet。

例:

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



注意:Tomcat3.1 的 Servlet 自动重新加载功能不一定有效。

## (3) 更改 URL 根映射路径

更改 `<Context path="/" docBase="webapps/ROOT" debug="0" reloadable="true"> </Context>` 可以将 URL 的根 "/" 映射到其它文件系统路径。其中 path="/" 表示 URL 根,docBase 属性值表示 URL 根映射的实际文件系统路径。

例:

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

## 3.JSWDK 的安装

### 1) 安装 JDK

同 Tomcat 安装的第一步。

### 2) 安装 JSWDK

与 Tomcat 安装的第二步相似。无需修改 Autoexec.bat。

### 3) 启动服务

启动服务文件是安装目录下的 startserver.bat,例如本书是 C:\Sun\Jswdk\startserver.bat。通常是不能启动成功的。

问题是启动 startserver.bat 的时候,发生 Out of Environment Space 错误,原因是 Windows 提供给环境变量的空间太小。此时需要调整 startserver.bat 的属性,方法如下:

(1) 关闭 DOS 窗口(错误可能破坏了它的 CLASSPATH 变量)

(2) 打开一个新的 DOS 窗口(运行 command 命令或执行 MS-DOS 方式)

(3) 单击窗口左上角的 MS-DOS 图标,打开下拉菜单或者在标题栏单击右键,弹出快捷菜单。

(4) 选中“属性”菜单选项,打开属性对话框。

(5) 单击“内存”标签。

(6) 调整“初始环境”下拉列表“auto”为 2816 或 2816 以上。

(7) 单击 OK。

(8) 启动服务器。

此时通常还是不能启动服务,还有某个不易捕捉的错误。在 DOS 窗口下键入“java”并回车,即运行 JDK 的 Java 程序执行程序,例如本书有如下操作:

►cd C:\Sun\Jdk\bin

►java

看看 JDK 能否正常运行,应该提示注册表访问错误,键名为:

“HKEY\_LOCAL\_MACHINE\Software\JavaSoft\Java Runtime Environment”

打开注册表检查,将键名“Java 运行时环境”改回为“Java Runtime Environment”。



注意:键名以“HKEY\_LOCAL\_MACHINE”开始。

再次启动服务器,应该正常了,如果还有什么错误,检查一下 autoexec.bat 文件,通常是那儿引起的错误。正常现象是应当有两个 MS-DOS 窗口出现。后出现的标志服务已启动的 MS-DOS 窗口如图 1-4 所示。



图 1-4 JSWDK 服务启动正常现象

同样留意一下该窗口中的信息。第一行是 JSWDK WebServer 的版本号,第二行是加载的配置信息来自文件 webserver.xml,第三行是服务器的地址及端口号为 localhost:8080 或 127.0.0.1:8080。

在 IE 中键入地址端口号 <http://127.0.0.1:8080/> 或者 <http://localhost:8080/>, 应该可以看到如图 1-5 所示的页面。

看看页面的内容,如图 1-5 所示,找到这段文字:“You are viewing the default JSWDK home page which is distributed with this package. This page serves as a quick reference guide to related resources and is located at: </path/to/jsjdk>/webpages/index.html”。首先,它表明这是 JSWDK 的默认主页,定位于“JSWDK 目录/webpages/index.html”。在这儿,有两件事需要注意:一牢记根目录“/”与“webpages”目录的关系是逻辑映射对等关系。当然仅限于 JSWDK 中。二注意“index.html(index.htm)”、“index.jsp”是 JSWDK 默认打开文档。也就是说,只要当前路径下有这样的文件,JSWDK 都会自动打开,而且在浏览器上不会显示文件名。其次它还告诉我们有相关资源。接着往下看,就会找到相关的资源,其中最重要的三个资源是 Servlet、JSP 的例子及其二者的 API 文档。这些例子是有相当参考价值的。

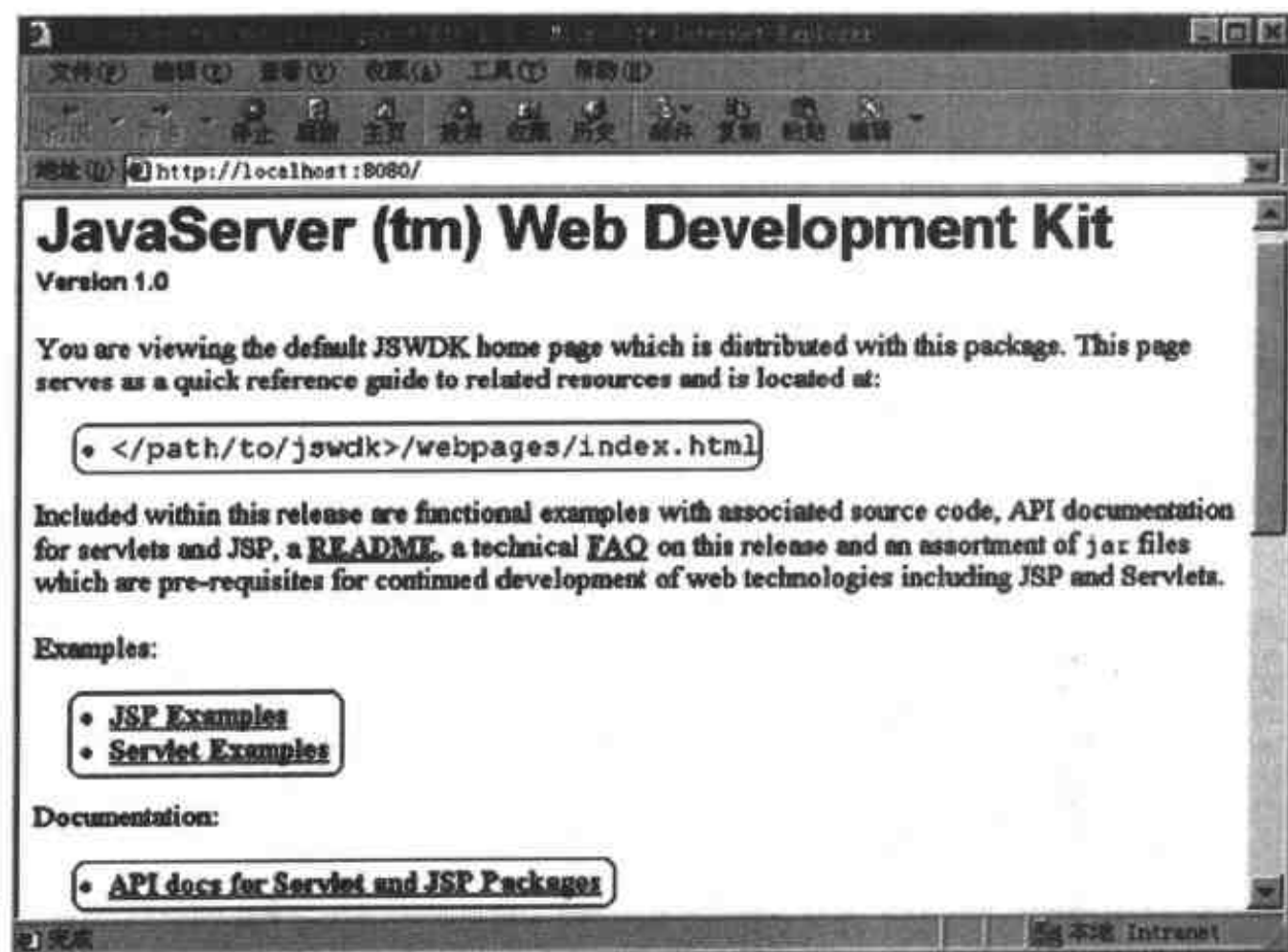



图 1-5 标志获得 JSWDK JSP 服务的页面

 提示:如果在点击了一个 JSP 的例子后出现 500 的错误而且没有别的错误信息,在浏览器 IE 上依如下次序操作工具 > Internet 选项 > 高级,把显示友好的 HTTP 错误的选项去掉。然后 IE 就会报出具体的出错信息。原因可能是没有将 tools.jar 放到 classpath 中去。

服务成功启动了,下面试验一下能否关闭服务。首先启动服务,然后执行 stopserver.bat,结果是服务并没有被关闭。事实上,JSWDK1.0.1 的 stopserver.bat 有一个小小的 bug。找到行 `java com.sun.web.shell.Shutdown %1 %2 %3 %4 %5 %6 %7 %8 %9`,在其前面添加 start。一切 OK。

技巧:如果希望在需要的时候打开服务器或关闭服务器,那么在桌面(或快速启动)建立两个快捷方式是有必要的,并调整其“属性 > 内存 > 初始环境”为上文所述。如果觉得要打开两个 MS-DOS 窗口很讨厌的话,调整其“属性 > 程序”,选中“退出时关闭”选项。这样每次启动完服务,便只剩下一个窗口。

#### 4) 配置服务器

配置服务器,一方面使它按需要提供服务,另一方面解决上面出现的不能解决的问题。JSWDK 配置文件为 webserver.xml,就在安装目录下面。节选片段如下:

```
<! DOCTYPE WebServer [

<! ELEMENT WebServer (Service+ )>
<! ATTLIST WebServer
    id ID #REQUIRED adminPort NMTOKEN "">

<! ELEMENT Service (WebApplication * )>
```



```

<! ATTLIST Service
  id ID #REQUIRED
  port NMTOKEN "8080"
  hostName NMTOKEN ""
  inet NMTOKEN ""
  docBase CDATA "webpages"
  workDir CDATA "work"
  workDirIsPersistent (false | true) "false">

<! ELEMENT WebApplication EMPTY>
<! ATTLIST WebApplication
  id ID #REQUIRED
  mapping CDATA #REQUIRED
  docBase CDATA #REQUIRED
  maxInactiveInterval NMTOKEN "30">
]>

```

这是带有 DTD 的 xml 文件,它是 JSDK 服务器的默认配置文件。下面简单的看一下服务器配置文件的选项。从顶层根开始:

**WebServer:**由唯一的 HTTP Web 服务端实例管理的至少一个 Web 服务的集合。意思是 JSDK 只能有一个服务端实例,且是 HTTP 服务,但可管理多个服务。这个实例就是 Web 服务控制台,在进程管理中的名字为 java。

**id:**一个唯一的 Web 服务器 id。

**adminPort:**Web 服务器管理端口号,它被用作外部 Web 服务器调用本机管理任务(例如非强制的关闭某个 Web 服务)的端口。



**注意:**这些被管理的服务目前并未被指定,它是可改变的。

**Service:**被管理的 Web 服务。与一个完整的 URI 相联系的一个可区分 Web 资源。

**Id:**服务的唯一 id。这个字段作为关键字是必需的。

**port:**服务被注册的端口号,即服务的监听端口号。这个字段在 WebServer 配置集合中必须是唯一的。后继 Service 实例指定重复的端口号将导致初始化失败。

**hostName:**服务所在系统主机名。可选,如果不指定将使用 localhost。

**inet:**服务所在系统 ip 地址。

**docBase:**Web 站点根目录。这是客户端发送 http 请求时,Web 服务器访问的文件系统位置。以绝对或相对路径指定而且不需要一定在 JSDK 的安装目录内。

**workDir:**服务工作目录。Web 服务器用它作缓冲,同样使用绝对或相对路径指定,而且不需要一定在 JSDK 的安装目录内。

**workDirIsPersistent:**指示器,指明服务器关闭后是否将其使用的工作目录归还给主机系统。默认值 false。

WebApplication: 被管理 Web 资源的一个关联。相关 Web 资源的集合, 它应当是一个可区分的完整 URI。

id: 唯一的 Web Application id。

mapping: 与这个 Web Application 相联系的相对于某个服务的 URI 前缀。这个字段的值必须唯一。

docBase: Web Application 文档根目录。这是 Web 服务的一个实例发送 http 请求给指定 Web Application 实例时访问的文件系统地址。

maxInactiveInterval: 最大会话时间(超时周期), 以分钟为单位。默认值是 30 分钟。

用写字板之类的工具打开 webserver.xml, 找到下面这几行。因为其它全是注释, 只有修改这几行才有用。

```
<WebServer id="webServer">
  <Service id="service0">
    <WebApplication id="examples" mapping="/examples" docBase="examples"/>
  </Service>
</WebServer>
```



注意: XML 规定只能有一个根元素, 这里只能有一个 `<WebServer ... ></WebServer>`。所以, 我们只有钻到“铁扇公主”肚里搅和, 感觉是不是很像“齐天大圣”。

#### (1) 更改端口号

如果 8080 端口被占用, 或者想用其它端口如 80, 那么修改 `<Service id="service0">` 为 `<Service id="service0" port="80">`。

#### (2) 使工作目录归还给系统

如果希望服务器关闭后能将工作目录归还给系统, 那么修改 `<Service id="service0">` 为 `<Service id="service0" workDirIsPersistent="true">`

#### (3) 添加服务

下面来高级点的, 添加一个服务。

例:

```
<Service id="service1" port="9000">
  <WebApplication id="mysample1" mapping="/mysample" docBase="../mysample"/>
</Service>
```

#### (4) 添加 Web 应用

在当前服务下添加一个 Web 应用。

例:

自己写的程序希望能单独放在某个目录里面, 且能够被执行。假设自己的程序目录路径是“c: \ sun \ mysample”, JSWDK 的安装路径为“c: \ sun \ jswdk”, 那么在 `<Service ... >` 和 `</Service>` 之间添加类似下面的行。

```
<WebApplication id="mysample" mapping="/mysample" docBase="/sun/mysample"/>
或者: <WebApplication id="mysample" mapping="/mysample" docBase="../mysam-
```

ple"/>

然后再将 examples 目录下的 Web-inf 目录拷贝到 mysample 目录。实际上,只需要四个“\*.properties”文件。

现在可以享受成果了,在没有学习 JSP 语法之前,应将光盘上 sample1\_2\_1.jsp 拷贝到新建的目录里,然后在浏览器中键入地址,这里是:“http://localhost:8080/mysample/sample1\_2\_1.jsp”,页面如图 1-6 所示。



图 1-6 欢迎进入 JSP 世界

## 1.3 本章小结

本章站在一个比较高的高度讲述了 JSP 的起源。还讲述了关于 JSP 的一个观点,JSP 是什么——从整体技术角度把握了 JSP 是基于 Java 的服务端动态页面技术。这是本书概念上关于 JSP 的第一个重要观点。最后详细讲述了 JSP 环境的建立。

# 第2章 HTML 与 Dreamweaver

## 2.1 HTML

Web 的兴起与 HTML 是密不可分的,HTML 的重要性不言自明。HTML 标记可分为三个大的类型:文档单元、页首单元和正文单元。下面的分类是有益的,可以帮助您从纷繁的标记中寻得一种线索。

▼文档单元:HTML、HEAD、BODY、FRAMESET

▼页首单元:TITLE、ISINDEX、META、LINK、BASE、SCRIPT、STYLE

▼正文单元:分为两类,字块级单元和文本级单元。

▶字块级单元:H1、H2、H3、H4、H5、H6、UL、OL、DIR、MENU、LI、DL、DT、DD、P、PRE、BLOCKQUOTE、ADDRESS、DIV、CENTER、HR、FORM、TABLE、(框架字段)FRAME、NOFRAMES (/NO-FRAME)、IFRAME

▶文本级单元:(字体风格单元)TT、I、B、U、STRIKE、BIG、SMALL、SUB、SUP  
(短语单元)EM、STRONG、DFN、CODE、SAMP、KBD、VAR、CITE(表单字段)INPUT、SELECT、OPTION、TEXTAREA(表单内容)CAPTION、TR、TH、TD

我们并不打算详细讲解每一个标记,而是依需要讲解对本书非常重要的三个标记,它们是 TABLE、FORM、FRAMESET。

### 2.1.1 TABLE(表格)

#### 1. TABLE 单元

TABLE 标记用于标记一个表格。其语法规则如表 2-1 所示。

格式中属性的含义如下:

(1) ALIGN:指定表的对齐方式,其取值为以下三种:left(左对齐)、center(居中)和 right(右对齐)。默认为 left。

(2) BACKGROUND:指定一个用作背景的图形的 URL。

(3) BGCOLOR:指定背景颜色(采用十六进制 RGB 颜色或认可的颜色名称)。

(4) BORDER:整数,指定表格边框的尺寸,以像素为单位。默认边框为 0。

表 2-1 TABLE 单元语法规则

一般形式:	<TABLE> </TABLE>
属 性:	ALIGN = left   center   right, BACKGROUND = url, BGCOLOR = # rrggbb, BORDER = n, BORDERCOLOR = # rrggbb, BORDERCOLORDARK = # rrggbb, BORDERCOLORLIGHT = # rrggbb, CELLPADDING = n, CELLSPACING = n, COLS = n, FRAME = void   above   below   hside   lhs   vside   box   border, RULES = none   groups   rows   cols   all, WIDTH = n   p%,
内容模式:	One CAPTION, TR
上 下 文:	BODY, DIV, CENTER, BLOCKQUOTE, FORM, TH, TD and DD, LI

(5) BORDERCOLOR:指定边框颜色(采用十六进制 RGB 颜色或认可的颜色名称)。



注意:它必须和 BORDER 属性一起使用。

(6) BORDERCOLORLIGHT:指定 3D 边框的高亮显示颜色(采用十六进制 RGB 颜色或认可的颜色名称)。



注意:它必须和 BORDER 属性一起使用。

(7) CELLPADDING:整数,指定单元格内的数据与单元格边框之间的水平、垂直方向上的间距,以像素为单位。

(8) CELLSPACING:整数,指定单元格之间在水平和垂直方向上的间距,以像素为单位。

(9) COLS:整数,指定表格内列的数目,该属性可以加速表处理。

(10) FRAME:指定显示那些外部边框,取值为:void, above, below, hside, lhs, vside, box, border。

(11) RULES:指定显示那些内部边框,取值为:none, groups, rows, cols, all。

(12) WIDTH:整数,指定表的宽度,以像素为单位,也可以指定为一个百分比。

## 2. CAPTION 单元

表格的标题用以说明表格的内容,但是表格并不一定要有标题。其语法规则如表2-2所示。

表 2-2 CAPTION 单元语法规则

一般形式:	<CAPTION> </CAPTION>
属 性:	ALIGN = left   center   right, VALIGN = top   middle   bottom   baseline
内容模式:	TT, I, B, U, STRIKE, BIG, SMALL, SUB, SUP, EM, STRONG, DFN, CODE, SAMP, KBD, VAR, CITE, A, APPLET, IMG, FONT, BASEFONT, BR, MAP, IN- PUT, SELECT, TEXTAREA and plaint text.
上 下 文:	TABLE

格式中属性的含义如下:

(1) ALIGN:指定标题的水平对齐方式,取值为:left,center,right。默认为 center。

(2) VALIGN:指定标题的垂直对齐方式,取值为:top,middle,bottom,baseline。默认为 bottom(底部对齐)。

### 3. TR 单元

在 HTML 中,表格的构造是按行进行的。其语法规则如表 2-3 所示。

表 2-3 TR 单元语法规则

一般形式:<TR> [</TR>]

属性:ALIGN = left | center | right, BACKGROUND = url, BGCOLOR = #rrg-gbb, BORDERCOLOR = #rrggb, BORDERCOLORDARK = #rrggb, BORDERCOLORLIGHT = #rrggb, VALIGN = top | middle | bottom | baseline

内容模式:TH, TD.

上下文:TABLE.

格式中属性的含义如下:

(1) ALIGN:指定行的对齐方式,取值为:left,center,right。默认为 left。

(2) BACKGROUND:指定一个用做表行背景的图形的 URL。

(3) BGCOLOR:指定行的背景颜色(采用十六进制 RGB 颜色或认可的颜色名称)。

(4) BORDERCOLOR:指定行的边框颜色(采用十六进制 RGB 颜色或认可的颜色名称)。



注意:只有当<TABLE> </TABLE>标记的 BORDER 属性不为 0 时,该属性才起作用。

(5) BORDERCOLORDARK:指定行的 3D 边框的阴影显示颜色(采用十六进制 RGB 颜色或认可的颜色名称)。



注意:只有当<TABLE> </TABLE>标记的 BORDER 属性不为 0 时,该属性才起作用。

(6) BORDERCOLORLIGHT:指定行的 3D 边框的高亮显示颜色(采用十六进制 RGB 颜色或认可的颜色名称)。



注意:只有当<TABLE> </TABLE>标记的 BORDER 属性不为 0 时,该属性才起作用。

(7) VALIGN:指定行中文字的垂直对齐方式。取值为:top,middle,bottom,baseline。默认为 middle。

### 4. TD 单元和 TH 单元

在 HTML 中,TD 标记和 TH 标记用来表示表行中的表元。标记内部可以嵌套表格。TD 标记表示普通表元,即表数据,表数据通常是左对齐的。TH 标记表示表头,表头通常

以粗体字居中显示。TD 单元和 TH 单元的语法规则完全相同,我们只给出 TD 单元的语法规则如表 2-4 所示。

表 2-4 TD 和 TH 单元语法规则

<p>一般规则: &lt;TD&gt; [&lt;/TD&gt;]</p> <p>属性: ALIGN = left   center   right, BACKGROUND = url, BGCOLOR = #rrggbb, BORDERCOLOR = #rrggbb, BORDERCOLORDARK = #rrggbb, BORDERCOLORLIGHT = #rrggbb, COLSPAN = n, NOWRAP, ROWSPAN = n, VALIGN = top   middle   bottom   baseline</p> <p>内容模式: H1, H2, H3, H4, H5, H6, P, UL, OL, DIR, MENU, PRE, DL, DIV, CENTER, BLOCKQUOTE, FORM, HR, TABLE, ADDRESS, as well as TT, I, B, U, STRIKE, BIG, SMALL, SUB, SUP, EM, STRONG, DFN, CODE, SAMP, KBD, VAR, CITE, A, APPLET, IMG, FONT, BASEFONT, BR, MAP, INPUT, SELECT, TEXTAREA and plain text.</p> <p>上下文: TR.</p>
---

格式中属性的含义如下:

- (1) ALIGN: 指定单元格内文本的对齐方式, 取值为: left, center, right。默认为 left。
- (2) BACKGROUND: 指定一个用做单元格背景的图形的 URL。
- (3) BGCOLOR: 指定单元格的背景颜色(采用十六进制 RGB 颜色或认可的颜色名称)。
- (4) BORDERCOLOR: 指定单元格的边框颜色(采用十六进制 RGB 颜色或认可的颜色名称)。



注意: 只有当 <TABLE> </TABLE> 标记的 BORDER 属性不为 0 时, 该属性才起作用。

- (5) BORDERCOLORDARK: 指定单元格的 3D 边框的阴影显示颜色(采用十六进制 RGB 颜色或认可的颜色名称)。



注意: 只有当 <TABLE> </TABLE> 标记的 BORDER 属性不为 0 时, 该属性才起作用。

- (6) BORDERCOLORLIGHT: 指定单元格的 3D 边框的高亮显示颜色(采用十六进制 RGB 颜色或认可的颜色名称)。



注意: 只有当 <TABLE> </TABLE> 标记的 BORDER 属性不为 0 时, 该属性才起作用。

- (7) COLSPAN: 整数, 指定单元格跨越的表的列数。
- (8) NOWRAP: 指定该属性, 可使单元格内的文本换行。
- (9) ROWSPAN: 整数, 指定单元格跨越的表的行数。
- (10) VALIGN: 指定行中文字的垂直对齐方式。取值为: top, middle, bottom, baseline。默认为 middle。

## 例 2-1

编写 HTML 文档,建立如图 2-1 所示的表格。

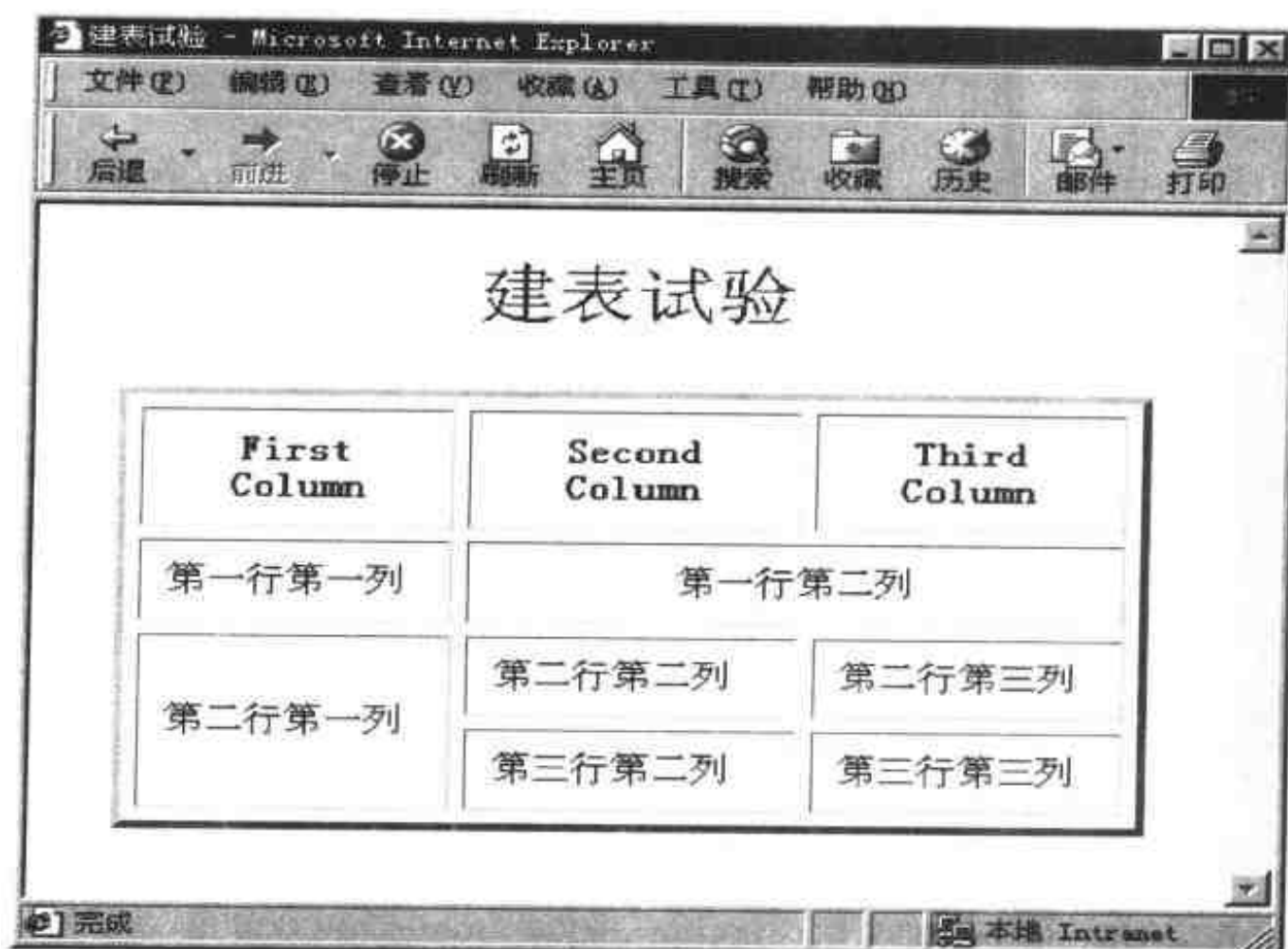


图 2-1 建表试验

源代码如下:

```
<html>
<head>
<title>建表试验</title>
</head>
<body>

<table width = 90 % border = 4 cellspacing = 6 cellpadding = 1
align = center bordercolorlight = # 99CCFF bordercolordark = # 669900>
<caption>
<font color = # CC66CC size = 6>
建表试验
</font>
</caption>
<tr>

<th>First Column</th>
<th>Second Column</th>
<th>Third Column</th>
</tr>
<tr>
```



```

        <td> 第一行第一列</td>
        <td colspan=2 align=center bgcolor= #FFFFCC>第一行第二列</td>
    </tr>
    <tr>
        <td rowspan=2>第二行第一列</td>
        <td>第二行第二列</td>
        <td> 第二行第三列 </td>
    </tr>
    <tr bgcolor= #FFFFCC>
        <td>第三行第二列</td>
        <td>第三行第三列</td>
    </tr>
</table>

</body>
</html>

```

## 2.1.2 FORM(表单)

表单实际上是一个模板,它把各种表单输入字段(INPUT 单元、SELECT 单元和 TEXTAREA 单元)组织在一起,每个输入字段单元对应一个名称/取值对,其中的名称由相应单元的 NAME 属性指定,而取值则一般由用户通过 Web 浏览器提供(当然也可以事先赋予初值或默认值)。表单中的全部名称/取值对构成了表单数据集,当表单被提交时,浏览器负责将表单数据集传送给远程 Web 服务器,由服务器上的 CGI(Perl、JavaScript、VBScript、ASP、JSP 等)脚本进行处理。表单数据集通常简称为表单数据。

### 1.FORM 单元

在 HTML 语言中,表单由 FORM 单元定义,表单中的各种输入字段由 INPUT、SELECT 和 TEXTAREA 三个单元定义,它们被封装在表单的起始标记<FORM>和结束标记</FORM>之间。同 TABLE 相区别,表单不允许嵌套,其语法规则如表 2-5 所示。

表 2-5 FORM 单元语法规则

一般形式:<FORM ACTION = URL> </FORM>
属 性:ACTION = url,METHOD = get   post,NAME = form_name,OnSubmit = 过程, TARGET = target_window
内容模式: H1, H2, H3, H4, H5, H6, P, UL, OL, DIR, MENU, PRE, DL, DIV, CENTER, BLOCKQUOTE,HR, TABLE, ADDRESS, as well as TT, I, B, U, STRIKE, BIG, SMALL, SUB, SUP, EM, STRONG, DFN, CODE, SAMP, KBD, VAR, CITE, A, APPLET, IMG, FONT, BASEFONT, BR, MAP, INPUT, SELECT, TEXTAREA and plain text.
上 下 文: BODY, DIV, CENTER, BLOCKQUOTE, TH, TD and DD, LI.

格式中属性的含义如下:

- (1) ACTION:指定将要接收表单数据的后台过程(CGI)的 URL。
- (2) METHOD:指定客户端和服务端之间数据交换的方法,取值为 get,post。
- (3) NAME:指定表单的名称,该名称在后台过程(CGI)中引用。
- (4) OnSubmit:指定提交表单时调用的事件处理程序。
- (5) TARGET:指定一个目标窗口。目标窗口决定链接(后台处理结果)加载到什么地方。

- ▶ \_blank:把链接加载到一个新的无标题窗口。
- ▶ \_parent:把链接加载到链接所在文档的父文档。
- ▶ \_self:把链接加载到链接所在的同一框架窗口。
- ▶ \_top:把链接加载到整个窗口。

表单内部的 HTML 单元,除了通常的 HTML 标记(上表中除 INPUT、SELECT、TEXTAREA)外,还有 4 个标记是专门为表单设计的,它们是 INPUT、SELECT、OPTION、TEXTAREA。这 4 个标记包含一个或多个置入浏览器的内部控件。HTML 提供了 12 个内部控件,分为 3 类:INPUT、SELECT、TEXTAREA。

## 2.INPUT 单元

INPUT 单元功能非常丰富,包含内部对象:BUTTON 控件、CHECKBOX 控件、FILE 控件、HIDDEN 控件、IMAGE 控件、PASSWORD 控件、RADIO 控件、RESET 控件、SUBMIT 控件、TEXT 控件。其语法规则如表 2-6 所示。

表 2-6 INPUT 单元语法规则

一般形式:<INPUT TYPE = x NAME = y>

属性:TYPE = button | checkbox | file | hidden | image | password | radio | reset | submit | text, NAME = string, VALUE = string, CHECKED, SIZE = n, MAXLENGTH = n, SRC = url, ALIGN = top | middle | bottom | left | Right.

内容模式:None(Empty).

上下文:DIV, CENTER, BLOCKQUOTE, FORM, TH, TD, DT, DD, LI, P, H1, H2, H3, H4, H5, H6, PRE, ADDRESS, TT, I, B, U, STRIKE, BIG, SMALL, SUB, SUP, EM, STRONG, DFN, CODE, SAMP, KBD, VAR, CITE, FONT, A, APPLET, CAPTION, but must be inside a FORM.

所有输入字段都必须有 TYPE 属性和 NAME 属性。INPUT 单元的其他属性与 TYPE 属性的取值有关,因此分类讲解如下:

### 1) BUTTON 控件

(1) 功能:在表单上显示一个多用途按钮。

(2) 属性:

NAME:字符串,BUTTON 控件的名称。

VALUE:字符串,显示在按钮上的文本。

(3) 事件:

OnClick:单击控件时激发。

OnFocus:控件接收焦点时激发。

2) CHECKBOX 控件

(1) 功能:在表单上显示一个复选框。

(2) 属性:

NAME:字符串,CHECKBOX 控件的名称。

VALUE:字符串,控件提交时的值(默认为 ON)。

(3) 事件:

OnClick:选中控件时激发。

OnFocus:控件接收焦点时激发。

3) FILE 控件

(1) 功能:在表单上显示一个文本框和一个 Browse...(浏览)按钮。

(2) 属性:

NAME:字符串,设置文件控件的名称。

4) HIDDEN 控件

(1) 功能:提供在表单上看不到的数据。

(2) 属性:

NAME:字符串,HIDDEN 控件的名称。

VALUE:字符串,控件的默认值。

5) IMAGE 控件

(1) 功能:在表单上显示一幅图像,用作提交按钮。

(2) 属性:

NAME:字符串,图像控件的名称。

SRC:图像文件的 URL,在图像控件中必须设置此属性。

ALIGN:图像在 Web 页面中的对齐方式。

(3) 事件:

OnClick:单击控件时激发。

OnFocus:控件接收焦点时激发。

6) PASSWORD 控件

(1) 功能:与 EXT 控件类似,但当在该控件中输入数据时,显示的不是实际的数据而是星号,这样可防止其他人看到屏幕上输入的数据。

(2) 属性:

NAME:字符串,PASSWORD 控件的名称。

VALUE:字符串,控件的默认值。

SIZE:整数,控件的长度,以字符为单位。

MAXLENGTH:整数,控件中所允许的最大字符数。

(3) 事件:

OnBlur: 当控件失去焦点时激发。

OnFocus: 当控件接收焦点时激发。

7) RADIO 控件

(1) 功能: 允许用户从若干个选项中选择一项。

(2) 属性:

NAME: 字符串, RADIO 空间的名称。

VALUE: 字符串, 控件提交时的值(每个单选按钮应有唯一的值)。

CHECKED: 使 RADIO 控件在默认状态下处于选中状态的选项。

(3) 事件:

OnClick: 单击控件时激发。

OnFocus: 控件接收焦点时激发。

8) RESET 控件

(1) 功能: 清除当前表单中的所有文本字段。控件以按钮形式出现在浏览器中。

(2) 属性:

NAME: 字符串, RESET 控件的名称。

VALUE: 字符串, 显示在 RESET 按钮标题中的文字。默认为“Reset”。

(3) 事件:

OnClick: 单击控件时激发。

OnFocus: 控件接收焦点时激发。

9) SUBMIT 控件

(1) 功能: 用来将表单中的所有元素传送到一个后台过程, 控件以按钮形式出现在浏览器中, 当一个表单被提交时, 输入控件中的数据以 ASCII 文本形式发送给表单 ACTION 属性表示的过程。

(2) 属性:

NAME: 字符串, SUBMIT 控件的名称。

VALUE: 字符串, 显示在 SUBMIT 按钮标题中的文字。默认值为“Submit”。

(3) 事件:

OnClick: 单击控件时激发。

OnFocus: 控件接收焦点时激发。

10) TEXT 控件

(1) 功能: 用来接收文本输入。

(2) 属性:

NAME: 字符串, TEXT 控件的名称。

VALUE: 字符串, 控件的默认值。

SIZE: 整数, 表示控件的长度, 以字符为单位。

MAXLENGTH: 整数, 控件中所允许的最大字符数。

(3) 事件:

OnBlur: 当控件失去焦点时激发。

OnFocus:控件接收焦点时激发。

OnSelect:当控件的内容被选中时激发。

OnChange:当控件改变时激发。

### 3.SELECT 单元

SELECT 标记用来在表单内生成一个选项列表,用户可以从这个列表中选择一个或多个选项。其语法规则如表 2-7 所示。

表 2-7 SELECT 单元语法规则

<p>一般形式:&lt;SELECT NAME = string&gt; &lt;/SELECT&gt;</p> <p>属性:NAME = string,SIZE = n,MULTIPLE</p> <p>内容模式:OPTION</p> <p>上下文:DIV,CENTER,BLOCKQUOTE,FORM,TH,TD,DT,DD,LI,P,H1,H2,H3,H4,H5,H6,PRE,ADDRESS,TT,I,B,U,STRIKE,BIG,SMALL,SUB,SUP,EM,STRONG,DFN,CODE,SAMP,KBD,VAR,CITE,FONT,A,APPLET,CAPTION, but must be inside a FORM.</p>
---

格式中属性的含义如下:

- (1) NAME:字符串,SELECT 控件的名称,必须设置。
- (2) SIZE:整数,列表窗口中可见的选项个数。如果 SIZE=1,得到一个下拉式列表。
- (3) MULTIPLE: 如果设置该项,则允许在列表中选择多个选择项。

### 4.OPTION 单元

SELECT 单元中的选择项由 OPTION 单元定义,OPTION 单元的语法规则如表 2-8 所示。

表 2-8 OPTION 单元语法规则

<p>一般形式:&lt;OPTION&gt; [&lt;/OPTION&gt;]</p> <p>属性:VALUE = string, SELECTED</p> <p>内容模式:Plaint text .</p> <p>上下文:SELECT.</p>
--

格式中属性的含义如下:

- (1) VALUE: 标识一个 OPTION 选项,即是一个 OPTION 选项的名称。默认取值为 OPTION 单元的内容。
- (2) SELECTED: 指定一个选项为默认的选择项。

### 5.TEXTAREA 单元

TEXTAREA 单元(控件)与 TEXT 控件相似,但允许多行输入。其语法规则如表 2-9 所示:

表 2-9 TEXTAREA 单元语法规则

一般形式: `<TEXTAREA NAME = string, ROWS = n, COLS = n></TEXTAREA>`  
 属 性: NAME = string, ROWS = n, COLS = n  
 内容模式: Plain text.  
 上 下 文: DIV, CENTER, BLOCKQUOTE, FORM, TH, TD, DT, DD, LI, P, H1, H2, H3, H4, H5, H6, PRE, ADDRESS, TT, I, B, U, STRIKE, BIG, VAR, CITE, FONT, A, APPLET, CAPTION, but must be inside a FORM.

格式中属性的含义如下:

- (1) NAME: 字符串, TEXTAREA 控件的名称。
- (2) ROWS: 整数, 控件的高度, 以行为单位。
- (3) COLS: 整数, 控件的宽度, 以字符为单位。

#### 例 2-2

编写 HTML 文档, 建立如图 2-2 所示的表单。

建立表单试验

请您如实填写下列信息 (带\*为必填项)

用户名:  \*

密 码:  \*

确认密码:  \*

性 别: ☒ 男 ☐ 女

职 业:

喜好运动: ☐ 篮球 ☐ 乒乓球 ☒ 羽毛球

座右铭:

本地 Intranet

图 2-2 建立表单试验

源代码如下:

```
<html>
<head>
<title> 表单试验</title>
```

```
</head>
<body>
<table border = 0 align = center>
<tr>
<td><p align = center>
<font size = 5 color = #CC66CC >建立表单试验</font>
</p><p>
<font size = 3 color = #993399 >请您如实填写下列信息</font>
<font size = 3 color = #FF0000 >(带 * 为必填项)</font>
</p>
<form method = post action = register.jsp >
    <p>用      户      名 :
<input type = text name = Username size = 30 maxlength = 30 >
        <font color = # FF0000 >* </font></p>
        <p>密      码 :
<input type = password name = Password size = 30 maxlength = 30 >
<font color = # FF0000 >* </font></p>
        <p>确认密码：
<input type = password name = Passwordagain size = 30 >
<font color = # FF0000 >* </font></p>.
        <p>性      别：
<input type = radio name = Radsex value = man checked>男
<input type = radio name = Radsex value = woman >女</p>
        <p>职      业：
<select name = Metier size = 1 >
            <option value = student selected>学生</option>
            <option value = businessman >商人</option>
            <option value = armyman >军人</option>
            <option value = teacher >教师</option>
            <option value = others >其他</option>
</select></p>
        <p>喜好运动：
<input type = checkbox name = Basketball value = basketball >
篮球
<input type = checkbox name = Pingpong value = pingpong >
乒乓球
<input type = checkbox name = Badminton value = badminton
checked>
羽毛球</p>
```

```

        <p>座 &nbsp;    右 &nbsp;    铭:
        <textarea name = Yourwords cols = 30 rows = 3 >
        您的名言
        </textarea></p>
        <p align = center >
            <input type = submit name = Submit value = 提交 >
            <input type = reset name = Reset value = 重置 >
        </p>
    </form>
</td>
</tr>
</table>
</body>
</html>

```

### 2.1.3 FRAMESET (框架)

TABLE 标记用于版面布局，几乎是无往不利。它使得页面内容有序而且位置相对固定。

我们看到 TABLE 标记可以将整个浏览器窗口分为多个区域（表元），这些区域之间是相关的。这里的“相关”意思是“只能对整个页面刷新，而不能对某个区域刷新”。事实上，确有需要将浏览器窗口分为多个独立的区域，或叫窗格。例如：典型的聊天室页面布局。也正是因为有这种需求，HTML4.0 正式引入 FRAMESET 标记。与 FRAMESET 配合使用的还有 FRAME、IFRAME、NOFRAME。

FRAMESET 标记是一个容器，FRAME、IFRAME、NOFRAME 包含于其中。因此，将框架分为两类（FRAMESET 单元、框架内部的 HTML 单元）是适合的。

#### 1. FRAMESET 单元

该标记用来指定一个包含 FRAME 和 NOFRAME 的容器。其语法规则如表 2-10 所示。

表 2-10 FRAMESET 单元语法规则

一般形式：	<FRAMESET COLS = width of each col   ROWS = height of each row> </FRAMESET>
属 性：	COLS = width of each col, FRAMEBORDER = 1   0, FRAMESPACING = n, ROWS = height of each row
内容模式：	FRAME, NOFRAME, IFRAME.
上 下 文：	None, But no other tags between HEAD and FRAMESET and no BODY.

格式中，此处上下文意思是 HTML 文档中，如果使用 FRAMESET 标记，那么就



不应使用 BODY 标记, 并且确保文档的 HEAD 和 FRAMESET 之间没有其它 HTML 代码。

格式中属性的含义如下:

- (1) COLS: 指定每一框架的列宽。有三种方式: 像素、百分比 (%)、相对尺寸 (\* )。COLS 和 ROWS 属性至少二选一。
- (2) FRAMEBORDER: 指定框架是否显示三维边框。如果该属性设置为 1, 则显示三维边框; 如果设置为 0, 则显示平面边框。默认值为 1。
- (3) FRAMESPACING: 整形值, 指定框架之间的间隔, 以像素为单位。默认框架之间没有间隔。
- (4) ROWS: 指定每一框架的行高。有三种方式: 像素、百分比 (%)、相对尺寸 (\* )。COLS 和 ROWS 属性至少二选一。

## 2. 框架内部的 HTML 单元

框架内部的 HTML 单元包括: FRAME、IFRAME、NOFRAMES (/NOFRAME), 分别介绍如下:

### 1) FRAME 单元

FRAME 单元必须与 FRAMESET 单元一起使用, 并位于 FRAMESET 内部。其语法规则如表 2-11 所示:

表 2-11 FRAME 单元语法规则

一般形式:	<code>&lt;FRAME NAME = name of frame SRC = name of html&gt; &lt;/FRAME&gt;</code>
属性:	<code>FRAMEBORDER = 1   0, MARGINHEIGHT = n, MARGINWIDTH = n, NAME = name of frame, NORESIZE, SCROLLING = yes   no, SRC = name of html.</code>
内容模式:	None (Empty)
上下文:	FRAMESET

格式中属性的含义如下:

- (1) FRAMEBORDER: 指定框架是否显示三维边框。当该属性值为 1 时, 显示三维边框, 如果为 0 则显示平面边框。默认值为 1。
- (2) MARGINHEIGHT: 整数, 指定框架的高度, 以像素为单位。
- (3) MARGINWIDTH: 整数, 指定框架的宽度, 以像素为单位。
- (4) NAME: 指定框架窗口的名称。
- (5) NORESIZE: 一般情况下, 用户可以调整框架的大小。如果指定了该属性, 则不能高速调整框架的大小。
- (6) SCROLLING: 指定框架是否可以滚动。如果该属性设置为 “YES”, 则框架可以滚动, 如果设置为 “NO” 则不能滚动。
- (7) SRC: URL 类型, 指定在框架内显示的 HTML 文件。

## 例 2-3

编写 HTML 文档，建立如图 2-3 所示的相似结构框架。

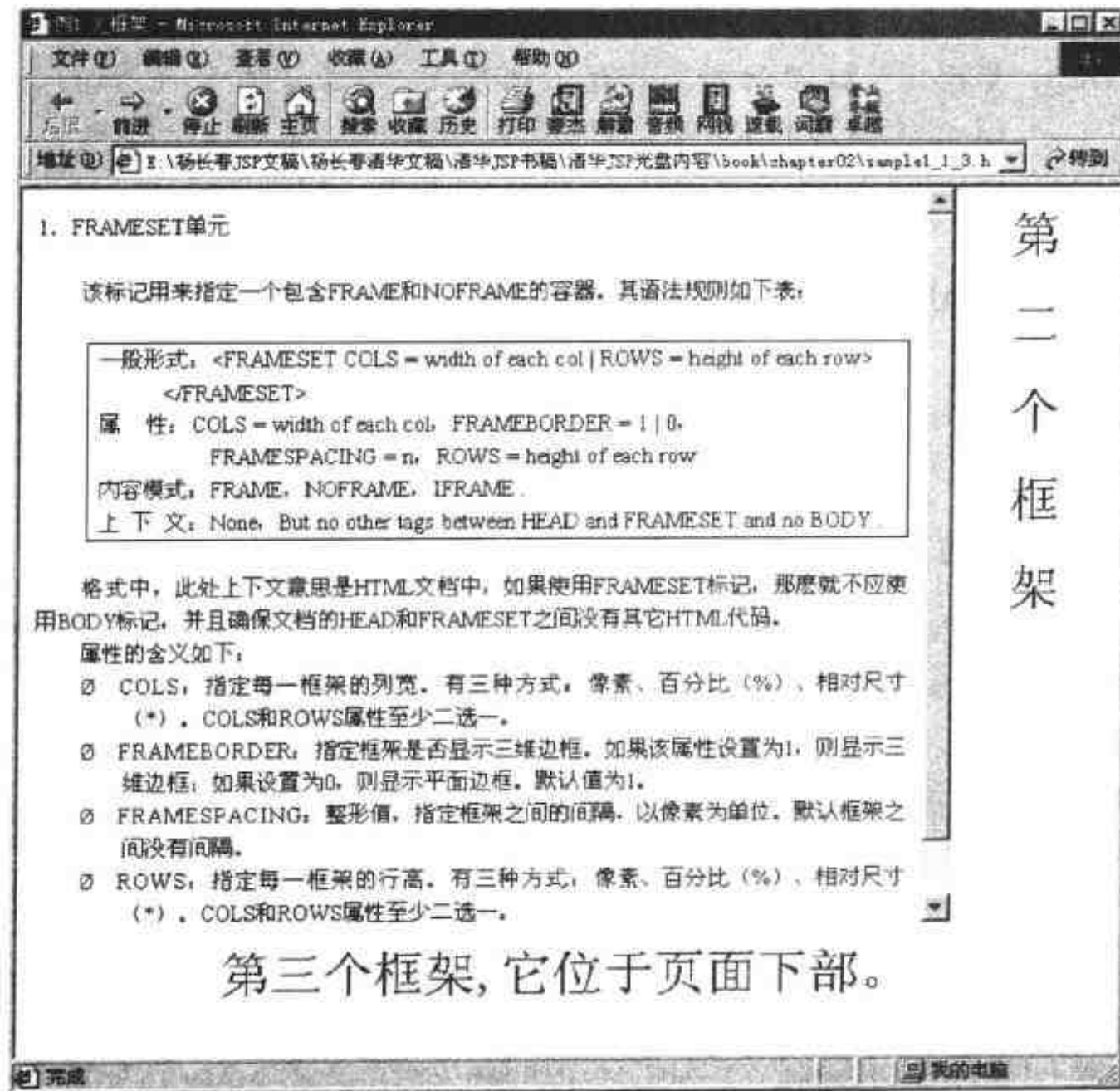


图 2-3 建立框架试验

程序如下：

```
<html>
<head>
<TITLE>例 1.3 _框架</TITLE>
</head>

<FRAMESET ROWS = 85% , * FRAMEBORDER = 0 FRAMESPACING = 0>
  <FRAMESET COLS= 85% , * FRAMEBORDER= 0 FRAMESPACING= 0>
    <FRAME NAME = main SRC = sample2 _ 31 . htm>
    <FRAME NAME = right SCROLLING = NO
NORESIZESRC = sample2 _ 32 . htm>
  </frameset>
  <FRAME NAME = bottom SCROLLING= NO NORESIZE
SRC = sample2 _ 33 . htm>
</frameset>
```

</html>

### 3. IFRAME 单元

用 FRAME 标记建立的框架其位置是相对固定的, 而用 <iframe> </iframe> 标记可以建立浮动框架, 可以将框架放在 WEB 页面的任何位置, 其语法规则如表 2-12 所示。

表 2-12 IFRAME 单元语法规则

一般形式: <IFRAME NAME = name of frame SRC = name of html> </IFRAME>
属性: ALIGN = left   right   top   center   middle   bottom, FRAMEBORDER = 1   0, HEIGHT = n, WIDTH = n, MARGINHEIGHT = n, MARGINWIDTH = n, NAME = name of frame, SCROLLING = yes   no, SRC = name of html.
内容模式: None (Empty)
上下文: FRAMESET

格式中属性的含义如下:

- (1) ALIGN: 指定浮动框架或周围文字的对齐方式, 取值为: LEFT、RIGHT、TOP、CENTER、MIDDLE、BOTTOM。其中 LEFT 或 RIGHT 指定框架的对齐方式, 其余的指定周围文字的对齐方式。
- (2) FRAMEBORDER: 指定浮动框架是否显示三维边框。当该属性值为 1 时, 显示三维边框, 如果为 0 则显示平面边框。默认值为 1。
- (3) HEIGHT: 整数, 指定浮动框架的高度, 以像素为单位, 也可用浏览器窗口的百分比来表示。
- (4) WIDTH: 整数, 指定浮动框架的宽度, 以像素为单位, 也可用浏览器窗口的百分比表示。
- (5) MARGINHEIGHT: 整数, 指定浮动框架内上边距, 以像素为单位。
- (6) MARGINWIDTH: 整数, 指定浮动框架内左边距, 以像素为单位。
- (7) NAME: 字符串, 指定浮动框架的名称。
- (8) SCROLLING: 指定浮动框架是否可以滚动。如果该属性设置为 “YES”, 则框架可以滚动, 如果设置为 “NO” 则不能滚动。
- (9) SRC: URL 类型, 指定在框架内显示的 HTML 文件。

### 4. NOFRAMES (/NOFRAME) 单元

如果浏览器不支持框架, 这时可以通过 NOFRAMES 标记显示适当的信息, 其语法规则如表 2-13 所示。

表 2-13 NOFRAMES 单元语法规则

一般形式: <NOFRAMES> </NOFRAME> 属    性: None 内容模式: All tags in the body of HTML document. 上 下 文: FRAMESET
--



注意: <NOFRAMES>的匹配是</NOFRAME>。

## 2.2 Dreamweaver

可视化的开发工具大大地提高了软件生产的效率。其所见即所得特性让我们不用再面对大堆的无趣的代码。在前一节讲 HTML 的本意只是让读者更好地使用可视化的开发工具。

说到可视化的 Web 页面开发工具,怎能不提 Macromedia,其“三剑客”早已誉满天下。“三剑客”龙头老大 Dreamweaver 与 JSP 可谓天作之和(都是基于开放基金组织的软件)。现在,让我们一睹 Dreamweaver 的风采。与前一节相对应,讲解表格、表单、框架在 Dreamweaver 中怎样实现。另外,为配合后面的内容,讲解利用层排版页面。利用层排版页面几乎每个页面设计都会涉及到。掌握上述 4 个内容,即基本掌握 Dreamweaver,至少对大多数页面和部分的页面作者是这样。

### 2.2.1 表格

在此,假定您对 Dreamweaver 有一定了解,清楚文中叙述的命令所在位置。

#### 1. 例 2-1 在 Dreamweaver 中的实现过程

##### 例 2-4

(1) 启动 Dreamweaver。通常打开一个 Document Window。标题栏通常为“Untitled Document (Untitled-1) — Dreamweaver”。否则,新建一个 Document Window。

(2) 文件 > 保存。文中存为 sample2\_2\_1.htm。

(3) 修改 > 页面设置。在打开的对话框中,将标题由“Untitled Document”改为需要的。文中为“建表试验”。

(4) 插入 > 表格。打开一个表格参数对话框,设置或不设置参数。文中选设置参数,并且参数如下: ROWS = 4; COLUMNS = 3; WIDTH = 90%; BORDER = 4; CELLSPACING = 6; CELLPADDING = 1。然后单击确定。

或者:在对象选项板中单击 Insert Table (图像)按钮(如图 2-4 所示)。打开一个相同的对话框,当然参数设置同前。

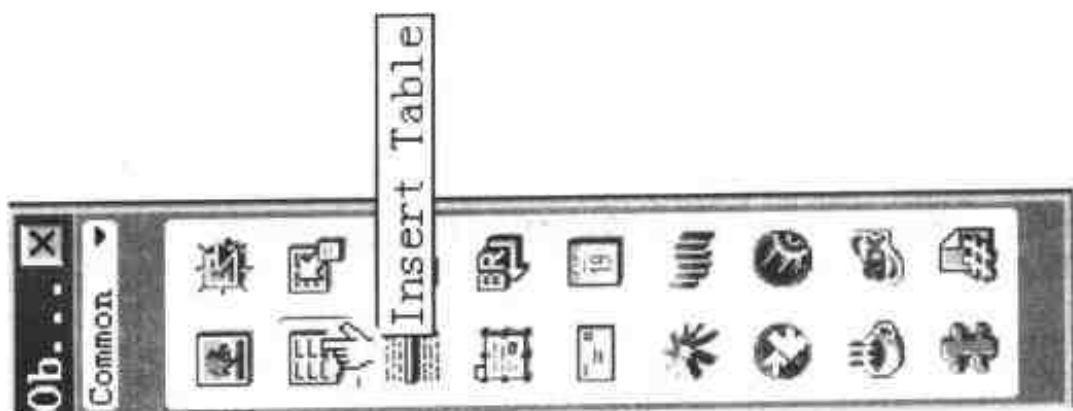


图 2-4 对象选项板

(5) 选中表格，在属性检视器（如图 2-5 所示）中进一步设置属性或在此设置属性（前面如果选不设置）。文中进一步设置参数如下：ALIGN = center; BORDERCOLOR-LIGHT = #99CCFF; BORDERCOLORDARK = #669900。

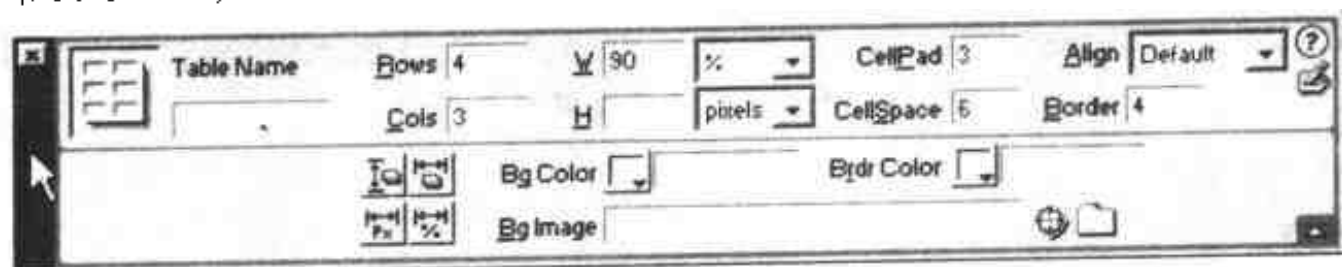


图 2-5 属性检视器

(6) 选中第 2 行第 2 列，按住 Shift 键，选中第 2 行第 3 列（如图 2-6 所示）。选择 修改 > 表格 > 合并单元格，或者单击属性检视器左下角的按钮（选中至少两个表元并悬停在按钮上，出现“merges selected cells using spans”字样的那个按钮），或使用快捷键 Ctrl + Alt + M。

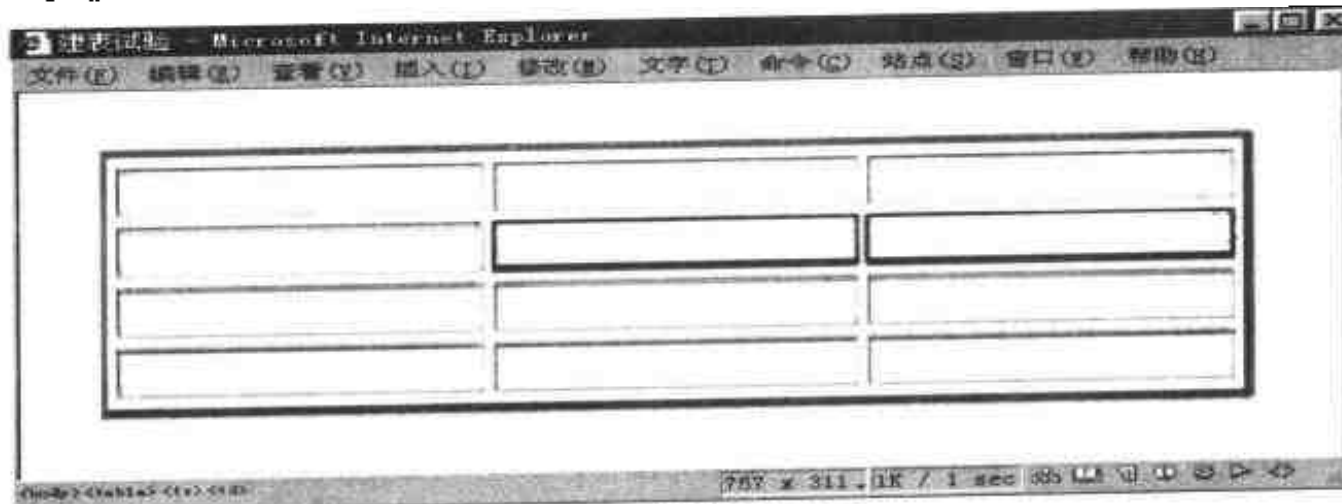


图 2-6 选择多个表元

同理，选中第 3 行第 1 列和第 4 行第 1 列，合并单元格，得到如图 2-7 所示效果。

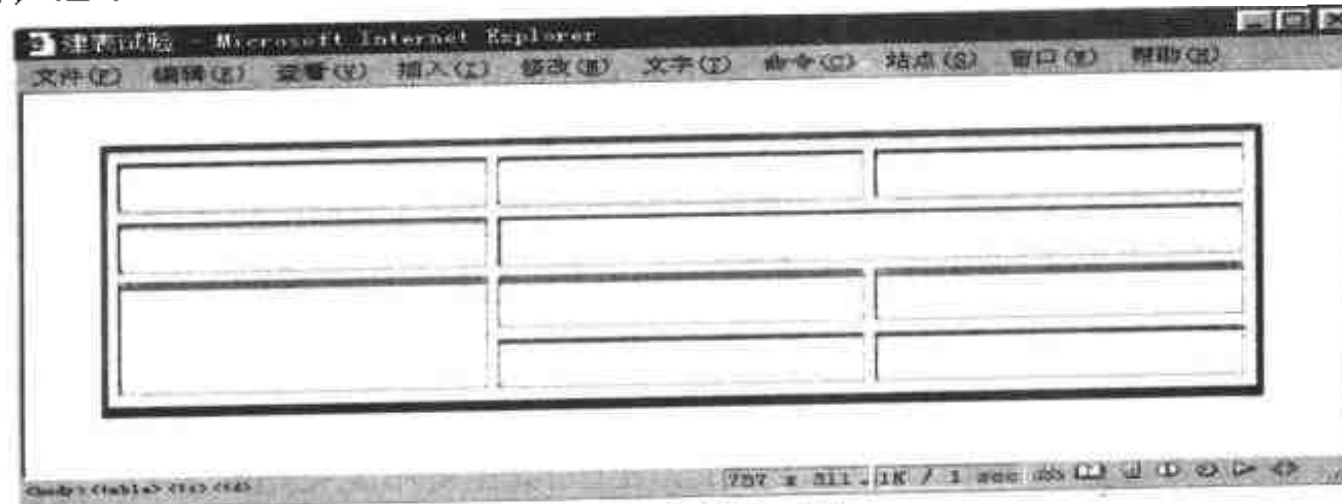


图 2-7 合并单元格

(7) 输入表元数据, 设置表元数据的属性、表元背景色。基本上, 这些操作都是选择某个对象, 然后在属性检视器中设置其属性。例如: 选择表元数据, 在属性检视器上设置字体大小、字体颜色、表元中的位置等属性。最后完成效果如图 2-8 所示。



图 2-8 完成效果图

## 2. 比较

IE 打开, 看看在浏览器中的显示效果。与图 2-1 比较异同, 二者几乎一样。由表及里, 源代码应该差不多, 事实真是这样吗? 下面, 比较一下源代码。

Dreamweaver 生成的代码如下:

```
<html>
<head>
<title>建表试验</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>

<body bgcolor="#FFFFFF">
<p align="center"><font size="6" color="#CC66CC">建表试验</font></p>
<table width="90%" border="4" cellspacing="6" cellpadding="1" align="center"
bordercolorlight="#99CCFF" bordercolordark="#669900">
<tr>
<td>
<div align="center"><b>First Column</b></div>
</td>
<td>
<div align="center"><b>Second Column</b></div>
</td>
<td>
<div align="center"><b>Third Column</b></div>
</td>
</tr>
<tr>
<td>第一行第一列</td>
<td colspan="2">第一行第二列</td>
</tr>
<tr>
<td>第二行第一列</td>
<td>第二行第二列</td>
<td>第二行第三列</td>
</tr>
<tr>
<td>第三行第一列</td>
<td>第三行第二列</td>
<td>第三行第三列</td>
</tr>
</table>
```

```
<tr>
  <td>第一行第一列</td>
    <td colspan="2" bgcolor="#FFFFCC">
      <div align="center">第一行第二列</div>
    </td>
</tr>
<tr>
  <td rowspan="2">第二行第一列</td>
  <td>第二行第二列</td>
  <td>第二行第三列</td>
</tr>
<tr>
  <td bgcolor="#FFFFCC">第三行第二列</td>
  <td bgcolor="#FFFFCC">第三行第三列</td>
</tr>
</table>
</body>
</html>
```

比较发现,二者源代码有好几处不同的地方。具体有哪些不同,请读者自己仔细看看,我们不在此多花篇幅讲解。总之,Dreamweaver生成的代码显然没有手工生成的代码简练,但是,所见即所得确实加速了页面生成速度,因此,二者结合才能使效费比最高。看来,掌握好HTML语法并不多余。

## 2.2.2 表单

### 1. 例 2-2 在 Dreamweaver 中的实现过程

#### 例 2-5

(1) 启动 Dreamweaver。通常打开一个 Document Window。标题栏通常为“Untitled Document (Untitled-1) — Dreamweaver”。否则,新建一个 Document Window。

(2) 执行文件>保存。文中存为 sample2\_2\_2.htm。

(3) 执行修改>页面设置。在打开的对话框中,将标题由“Untitled Document”改为所需要的。文中为“建立表单试验”。

(4) 执行插入>表单。

(5) 在表单中输入所需的文字并设置其属性。请按图索骥(如图 2-2 所示)。

(6) 执行插入>表单对象>文本框,插入文本框作为“用户名”输入。在属性检视器中设置属性,文中为:TextField = Username, 字符长 = 30, Max Chars = 30, 类型 = 一行,还可以设置初值。



(7) 执行插入>表单对象>文本框, 插入文本框作为“密码”输入。在属性检视器中设置属性, 文中为: TextField = Password, 字符长 = 30, Max Chars = 30, 类型 = 密码。

确认密码照此办理。TextField = Passwordagain。

(8) 执行插入>表单对象>单选按钮, 插入两个单选按钮作为“性别”输入。在属性检视器中分别设置其属性, 文中一个为: RadioButton = Radsex, 选中时的值 = men, 初始状态 = 选中; 另一个为: RadioButton = Radsex, 选中时的值 = women, 初始状态 = 未选中。

(9) 执行插入>表单对象>列表/菜单, 插入列表/菜单作为“职业”输入。在属性检视器中设置其属性, 单击属性检视器右上角按钮(按钮标题为“列表数值”)打开一个标题为“列表数值”的对话框, 如图 2-9 所示。该对话框用于添加列表数值。列表数值以“对”出现。文中列表数值对为: 学生 = student; 商人 = businessman; 军人 = armyman; 教师 = teacher; 其他 = others。属性检视器中其它参数 List/Menu = MenuItem, 类型 = 列表, 高度 = 1, 开始被选中 = 学生。还可以指定是否允许多选。

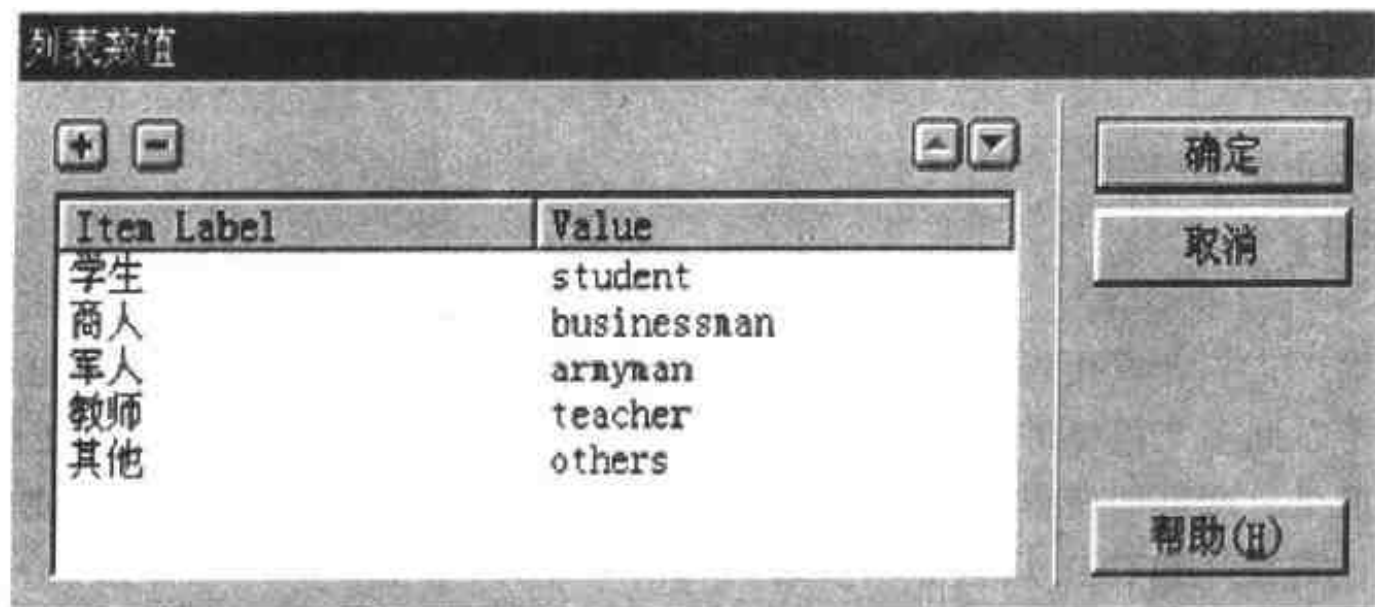


图 2-9 列表数值对话框

(10) 执行插入>表单对象>复选框, 插入三个复选框作为“喜好运动”输入, 对应三个选项依次为“篮球”、“乒乓球”、“羽毛球”。在属性检视器中设置其属性。文中各复选框属性分别如下: CheckBox 名称 = Basketball, 选中时的值 = basketball, 初始状态 = 未选中。CheckBox 名称 = Pingpong, 选中时的值 = pingpong, 初始状态 = 未选中。CheckBox 名称 = Badminton, 选中时的值 = badminton, 初始状态 = 选中。

(11) 执行插入>表单对象>文本, 插入文本作为“座右铭”输入。在属性检视器中设置属性。文中为: 首先设置类型 = 多行(属性变为 TEXTAREA 对象的属性, 否则为 TEXT 对象的属性), 然后设置 TextField = Yourwords, 字符长 = 30, Num Lines = 3, 初始值 = 你的名言。还可以设置属性包裹。

(12) 执行插入>表单对象>按钮, 插入两个按钮作为表单“提交”和“重置”输入。在属性检视器中设置属性, 文中分别为: 按钮名称 = Submit, 标签 = 提交, 动作 = 提交表单, 按钮名称 = Reset, 标签 = 重置, 动作 = 复位表单。



## 2. 比较

请读者自己比较一下。

### 2.2.3 框架

#### 1. 例 2-3 在 Dreamweaver 中的实现过程

##### 例 2-6

(1) 启动 Dreamweaver。通常打开一个 Document Window。标题栏通常为“Untitled Document (Untitled-1) — Dreamweaver”。否则，新建一个 Document Window。

(2) 插入>框架>下（左、右、上、下、左和上、左加上、上加左、split）。框架的级联菜单有括号中给出的几种框架样式可供选择。它们各自所表示的框架是什么结构呢？左表示的是整个浏览器窗口分为左右两部分（窗格），左窗格宽度为 80 像素。同理可知右、上、下的结构。左加上是整个浏览器窗口先作左右切分，左窗格 80 像素宽，然后对右窗格作上下切分，上窗格 80 像素高，切线形状“|—”，窗格总数为 3。左和上是对整个浏览器作左切分和上切分，分别为 80 像素宽、高，切线形状“—|—”，左上角有一 80×80 像素大小的窗格，窗格总数为 4。Split 的样式是对整个浏览器窗口作“+”（加号形状）切分，窗格总数为 4 且大小相等。从面的讲解可以看出：1. 样式命名以切线及切线位置为参考。2. 选用 Dreamweaver 提供的框架样式，除了 split，其余样式切分宽（高）都为 80 像素。如果 80 像素不符合您的要求可更改，方法如下：

►确信“查看>框架边框”菜单选项被选中，使框架边框可见，移动鼠标指针到切线上，鼠标指针变为双箭头。

►按下鼠标左键拖拽至适当位置后释放鼠标左键，完成更改，或者单击鼠标左键，选中整（某）个框架，在属性检视器中（如图 2-10 所示）右下角“行列的选择”按钮（两个）中选择一个，在按钮的前面有两个输入框（“数值”与“单位”输入框），可设置窗格的宽高属性。文中选“下”样式，此时整个框架被选中，可在属性检视器中设置其属性，包括是否有边框、边框颜色、边框宽度、上下窗格高度。文中上窗格高度 = 85%，下窗格高度为其余部分，即“单位”输入框选“relative”。其余为默认值。

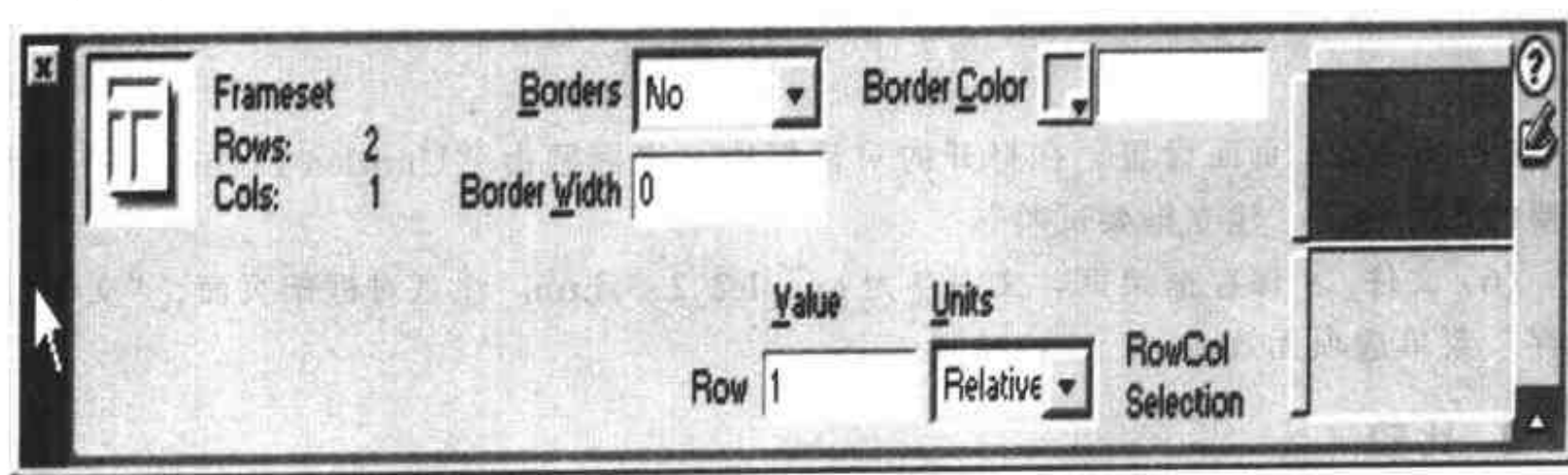


图 2-10 框架属性

(3) 单击上窗格中任意位置, 插入 > 框架 > 右, 将上窗格又分为两个窗格。同样, 可以通过选中上部框架, 在属性检视器中设置属性。选中方法如上文叙述。到这步代码生成如下:

```
<frameset rows="85%," frameborder="NO" border="0" framespacing="0">  
<frameset cols="85%," frameborder="NO" border="0" framespacing="0">  
<frame ... >  
    <frame ... >  
</frameset>  
    <frame ... >  
</frameset>
```

可见, 上述(2)、(3)步操作与 frameset 的生成相对应。具体的说, 第(2)步操作生成外层 frameset 标记, 第(3)步操作生成里层 frameset 标记。因此对此例, 单击上下切线, 选中整个框架, 单击左右切线, 选中上部框架。请读者验证一下, 并且明白调整某个框架的属性需要选择哪条切线。

(4) 确信“窗口 > 框架”菜单选项被选中, 使 Frames 选项板 (如图 2-11 所示) 可见。在 Frames 选项板中选中某个窗格, 在属性检视器中设置窗格属性。属性包括框架名称、src、滚动条、No Resize、边框、边框颜色、Margin Width、Margin Height。文中仅对 src 属性作了设置, mainFrame 的 src = sample2\_2\_31.htm, rightFrame 的 src = sample2\_2\_32.htm, bottomFrame 的 src = sample2\_2\_33.htm。其余全为默认值。

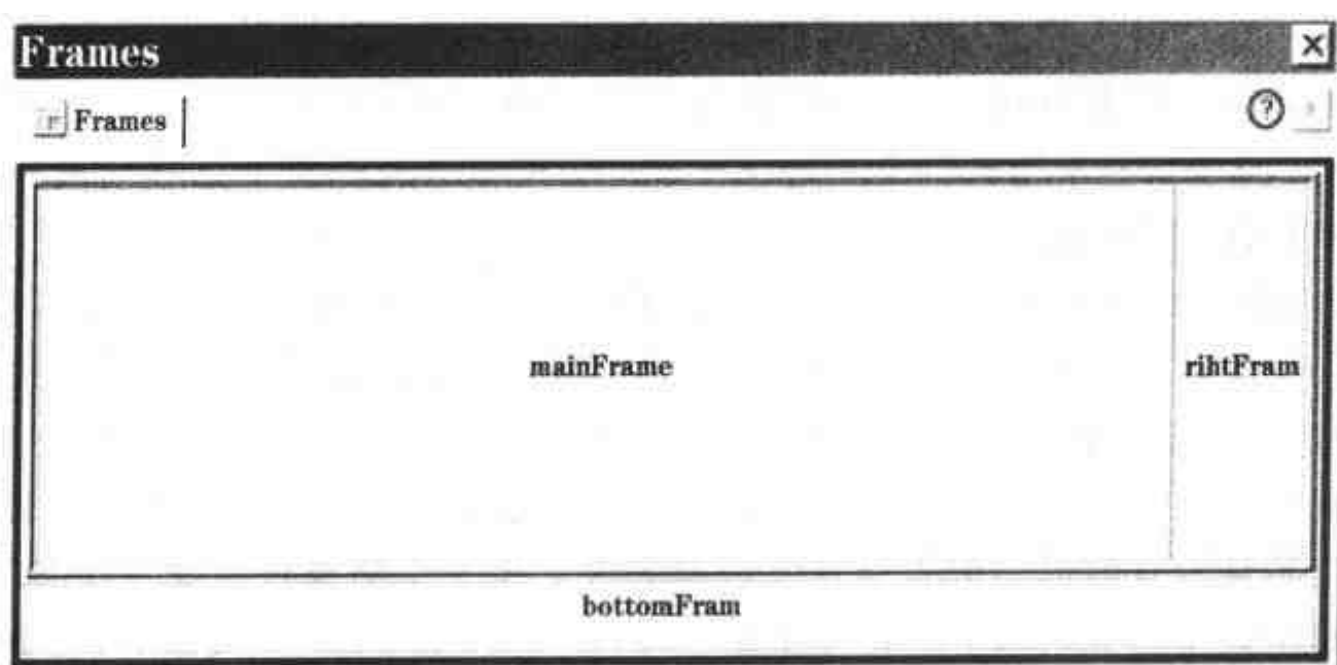


图 2-11 Frames 选项板

(5) 修改 > 页面设置。在打开的对话框中, 将标题由“Untitled Document”改为需要的。文中为“建立框架试验”。

(6) 文件 > 保存框架页。文中存为 sample2\_2\_3.htm。注意对框架页面, “文件 > 保存”菜单选项无效。

## 2. 比较

请读者自己比较一下。

## 2.2.4 用图层排版页面

在 2.2.2 节中，曾让读者比较一下例 2-2 页面与 Dreamweaver 生成页面的区别，二者最大的区别是例 2-2 中用 TABLE 标记将表单置于页面中间。利用表格排版页面是非常有效和强大的。但是要让表格定位于页面的任何位置，显然是勉为其难。这不能不说是一种遗憾，因为页面作者梦寐以求这种特性。图层是用于排版的第二种手段，其定位能力让表格望尘莫及，然而所谓“人无完人，金无足赤”，图层技术过于先进，不少浏览器不支持。Dreamweaver 实现了这种折中，它利用图层排版页面，最后转换成表格。不仅如此，它还提供将表格转换成图层，对已编辑好的页面可进行再编辑。

下面让我们用 Dreamweaver 将例 2-5 的页面效果做得与例 2-2 的一样。

### 1. 过程

(1) 打开一个 Document Window，保存。文中保存为 sample2\_2\_4.htm。从前面叙述的几个例子中，相信读者已经看出：本书例子的命名规则，第几章\_第几节\_例子序号.扩展名。以及 Dreamweaver 的操作顺序，除将要生成的页面是框架页不能先保存外，其余都是先保存，后开始工作。

(2) 修改 > 页面设置。命名页面标题。文中为“建立居中表单试验”。

(3) 插入 > 图层。将在页面上显示一个 200×115 像素大小的默认图层，或者单击对象框中的 Draw Layer 按钮，鼠标指针变为画图工具形状，此时可在 Document Window 任何位置画出一个图层。

(4) 选中一个图层。有两种方法：一是确信“查看 > 图层边框”菜单选项被选中，使图层边框可见，移动鼠标指针到图层边框，鼠标指针变为 4 向箭头形状，单击鼠标左键，选中该图层。二是确信“窗口 > 图层”菜单选项被选中，使图层选项板（如图 2-12 所示）可见，然后选中其中的某个图层。

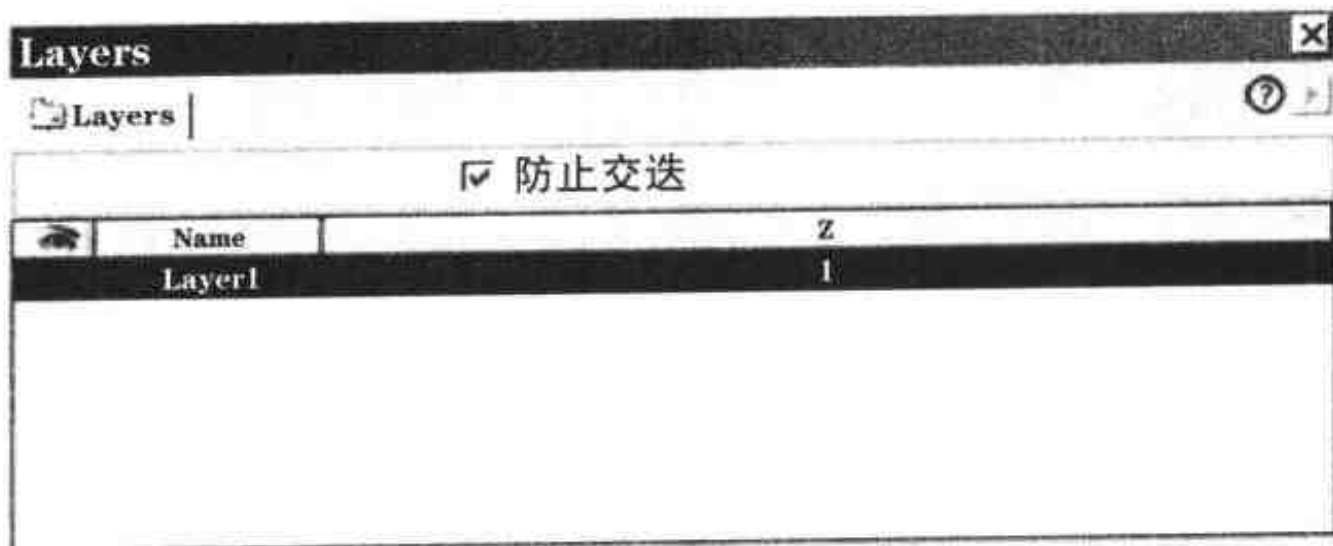


图 2-12 图层选项板

(5) 在属性检视器中设置选中图层的属性。属性包括：层识别号、左偏移量、上偏移量、宽、高、Z 轴序号、可视性、背景图案、背景颜色以及不常用的属性如标记、溢出、剪切左、右、顶、底。事实上，对不同的属性还有其它不同的设置方法。如层识别号，可在图层选项框中双击其名字，名字变为可改后更名；左偏移量、上偏移量实际上

就是指定位置，可移动图层改变其值，方法是移动鼠标指针到图层边框，指针变为 4 向箭头形状时，按下鼠标左键拖拽至适当位置。宽、高更改方法是移动鼠标指针到图层边框节点，指针变为双向箭头形状时，按下鼠标左键拖拽至适当位置。文中左偏移量为 200 像素，上偏移量为 10 像素，宽 400 像素，其余为默认值。

(6) 将例 2-2 中的表单(从 `<form...>` 至 `</form>` 结束)源代码复制到本例的 `<div...>` 和 `</div>` 之间。或重做例 2-5 的第 4 步到第 12 步操作。

(7) 修改 > 排版 > 转换图层为表格，打开一对话框(如图 2-13 所示)，有不少转换细节需要调整。此处由于只有一个图层，防止层交迭选项都可不选。但是这在有多个图层的时候，在设计阶段可防止图层交迭，使转换成功。还有一种方法，使“查看 > 防止图层交迭”菜单选项被选中，有同样效果。文中选中防止层交迭选项，其余采用默认值。

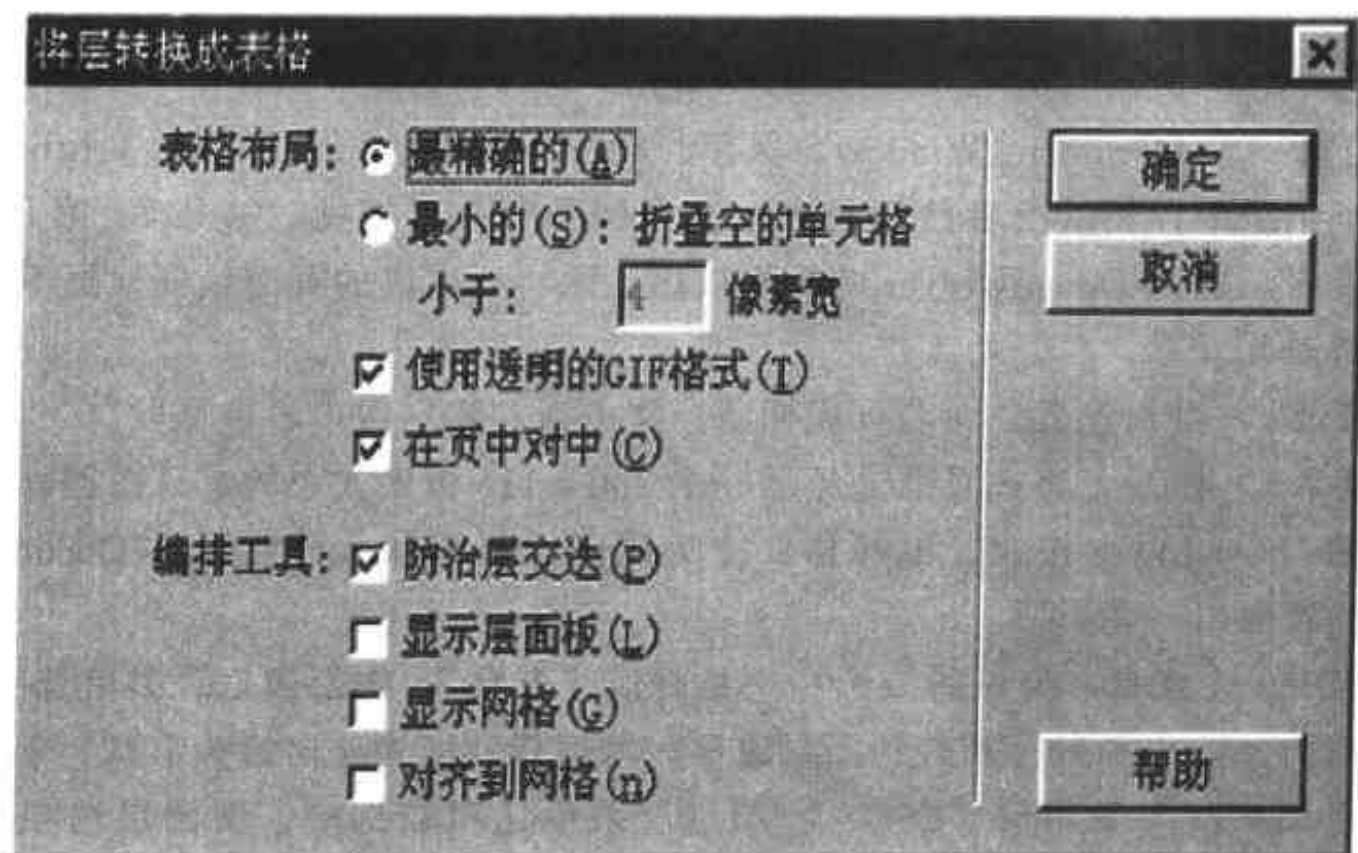


图 2-13 层转换成表格对话框

(8) 保存，结束。

## 2. 比较

请读者自己比较一下页面效果与源代码。

## 2.3 构建虚拟网站

构建虚拟网站是本章的重点，也是全书的目的之所在。我们希望通过一个渐进的过程让读者获得持续的成就感，因此，我们建立这个虚拟网站，引导读者渐入佳境，我们的程序几乎都通过渐改达到比较完善的地步。当然，我们只实现最基本的功能，但我们会在提高的总结论述中提出改进的方向和意见，希望读者能深入探讨。

那么，我们的网站主题是什么，有些什么功能呢？

### 2.3.1 构思 JSP 虚拟网站

#### 1. 主题

我们决定构建一个关于 JSP 的教学网站，其最主要的原因是基于电子文档的学习方式。我们将本书作成电子文档，例于构建为实际的网站，使读者从实际需要出发，思考建网的过程，看到设计的效果，清楚页面的实现，展望网站的发展。

#### 2. 功能

主题确定了，那么网站有些什么功能呢？想一想，网站通常要实现那些功能。一般主页都有页头，那么广告轮显是必要的，根据其实现所用知识点，将其作为第5章基本语法的例子。因为第8章我们使用了一个 JSP 文档生成器观察 JSP Container 的行为，而留言板可以使用与 JSP 文档生成器相似的技术实现，因此留言板在那时加入我们的虚拟网站，为网站添光加彩。一个小巧的计数器却有非常的价值，它体现了 JSP 最核心的 API，因此它作为第9章核心 API 的例子。通常一个网站的首页都是新闻，它作为 JDBC 典型应用之一将在第10章引入。时间、日期服务当然少不了。时间、日期是动态的，用 Java Applet 实现是最好的，因此，这是第11章很好的例子。最后，我们在网站的落成典礼中增加 JDBC 的一个主要应用——BBS，内部对象的典型应用——聊天室，综合应用——电子商务。还有网个东西——下载中心和电子邮件功能是 Sun 的组件，因此我们希望读者到网上下载并运用所学知识将其实现，应该没有任何问题。

#### 3. 版面

在实现之前，先来看看最后的效果图，如图 2-14 所示。



图 2-14 虚拟网站最终效果图——首页部分



当然这儿的首页新闻系统显示的内容并不是新闻。看了这个效果图，如果读者心中充满了期待，那么我们将感到非常高兴。

#### 4. 结构

这里的结构有两层含义，一是页面结构，或叫链接网络，在生成页面超链接时建立。二是文档存储结构，物理存储的树状结构，在文档存储时建立。这儿讨论文档存储结构，首先这种结构是非常重要的，合理的结构可以减少工作量和使逻辑清晰。其次文档（站点）管理是 Dreamweaver 非常强大的功能，Dreamweaver 站点管理器类似于 Windows 的资源管理器，可以执行打开、复制、更名等常规操作，还能自动超链接更新，既是说当某个文档执行了更名、移动、删除等操作时，能自动检查链接完整性，提示需要更新还是不更新。还有模板重用与自动更新功能。重量级的功能是实现本地与远程站点二者文件存储结构的相互映像，这保证了下载或上传某个文件（夹）能够保持相对（或绝对）路径不变，使得远程站点管理更轻松更快捷。下面我们看看怎样构建本地站点。

(1) 站点 > 新建站点，打开一个对话框，如图 2-15 所示。

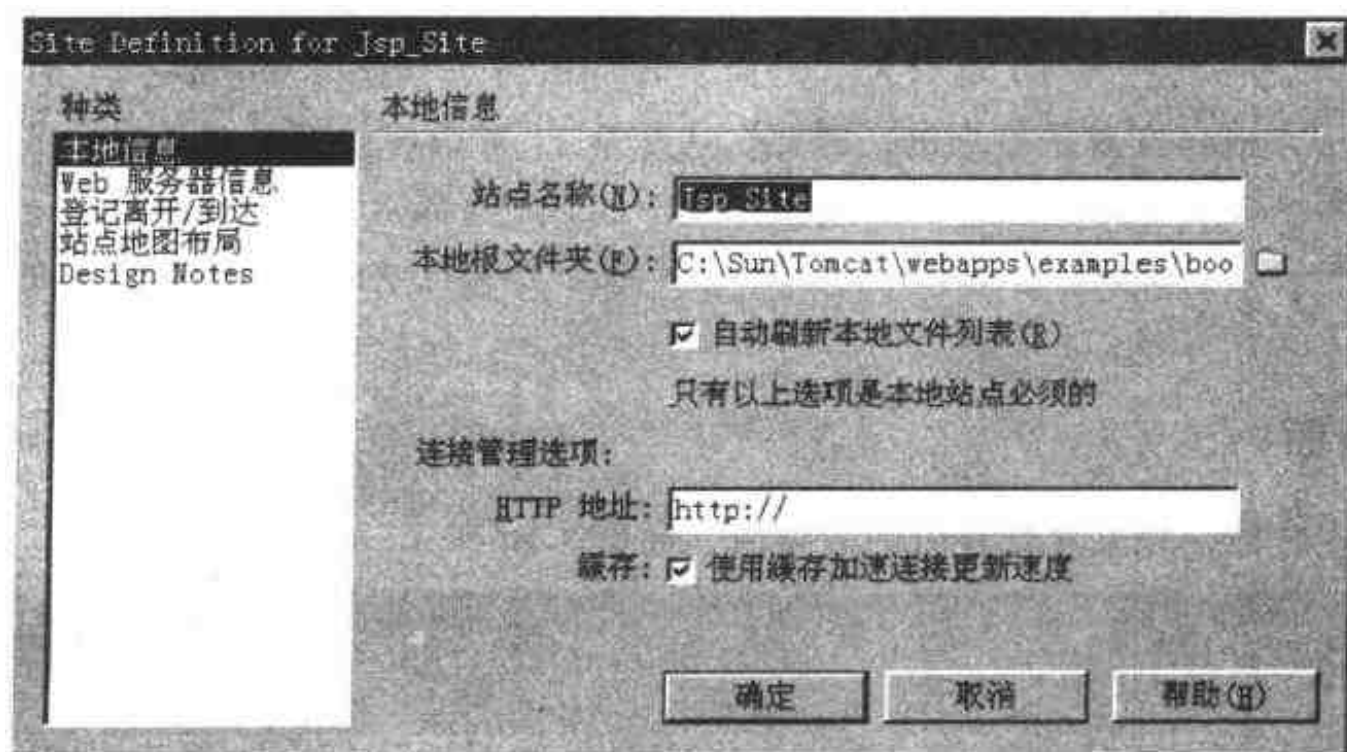


图 2-15 新建站点对话框

(2) 在左边“种类”列表中，选中本地信息，然后在右边“本地信息”的“站点名称”输入框中，输入满意的站点名称；在“本地根文件夹”输入框中，输入绝对路径或浏览的选择。还有是否自动刷新本地文件列表、远程 HTTP 地址、是否缓存。文中站点名称为 Jsp\_Site，自动刷新本地文件列表，使用缓存。

(3) 单击“确定”按钮，打开站点管理器，如图 2-16 所示。请读者注意一下通常有两个窗口，还请注意工具栏最左边的两个按钮，待会儿会用到。左边按钮叫“Site Files”，右边按钮叫“Site Map”。

(4) 定义一些文件夹。这儿我们假设每一章都是独立的，不共享任何资源如图片，由此建立第 1 章到 14 章的文件夹。第 14 章是整个网站的实现，因此它的结构又单独设计，我们将网站的所有组件都放在了 this 目录，请读者留意一下它的结构。马上会用到的是 chapter14 目录下的 template 子目录。几乎完整的结构如图 2-16 所示。

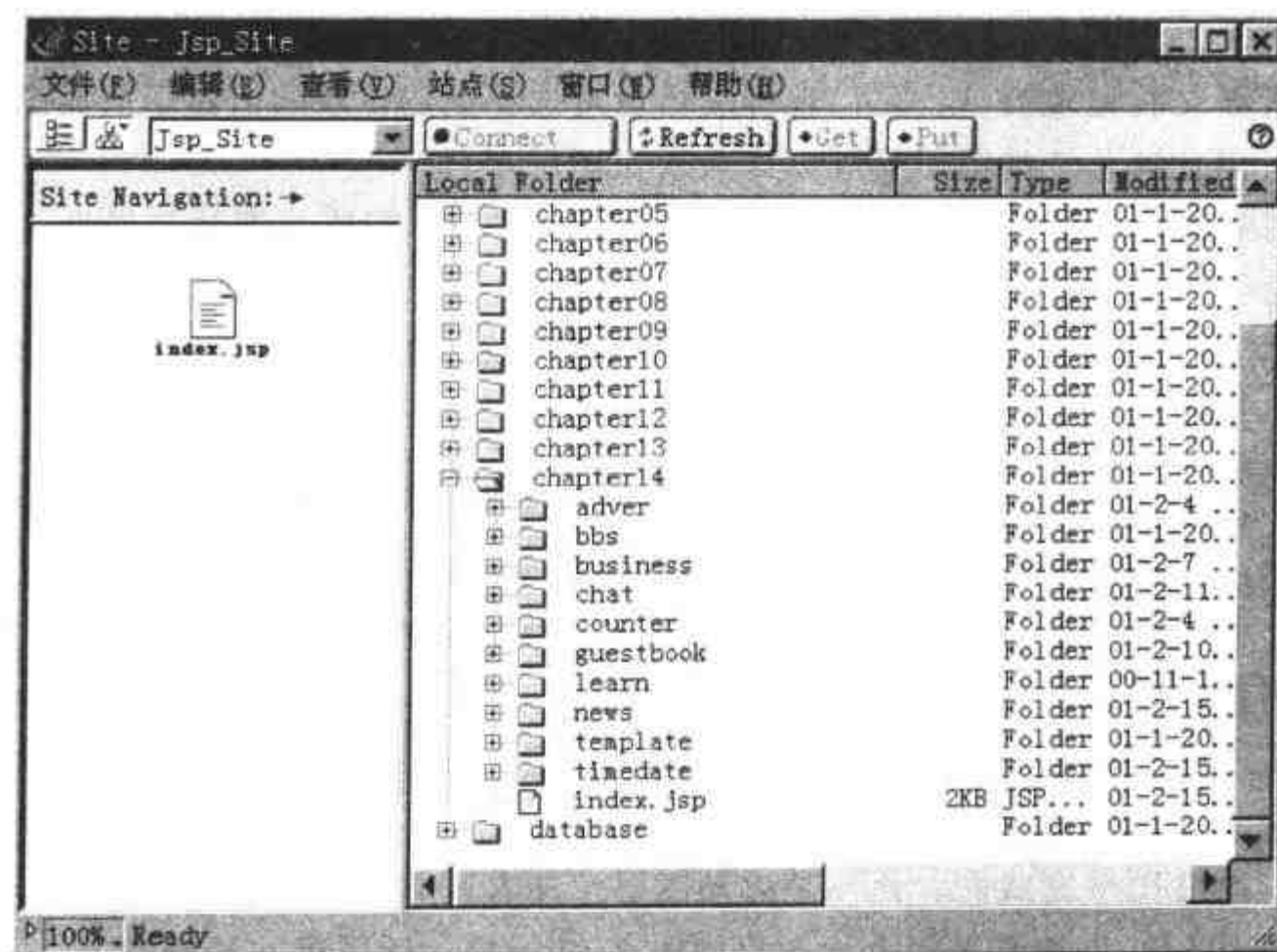


图 2-16 站点管理器及站点结构设计

### 2.3.2 构建 JSP 虚拟网站

我们开始 JSP 虚拟网站的一期工程，将网站的“心脏”——主页——规划出来。这也正是前面内容的应用提高。如同 HTML 文档结构，将主页分为页头、主体、页脚。为什么将主页分为三个部分呢？这是因为计算机中非常讲究结构与逻辑独立性，独立性意味着我们可以分别对其操作，而不影响其它部分。下面，让我们分别实现。

#### 1. 建立页头

##### 1) 目标

首先，让我们明确一下目标，这一步完成什么任务。如图 2-17 所示。

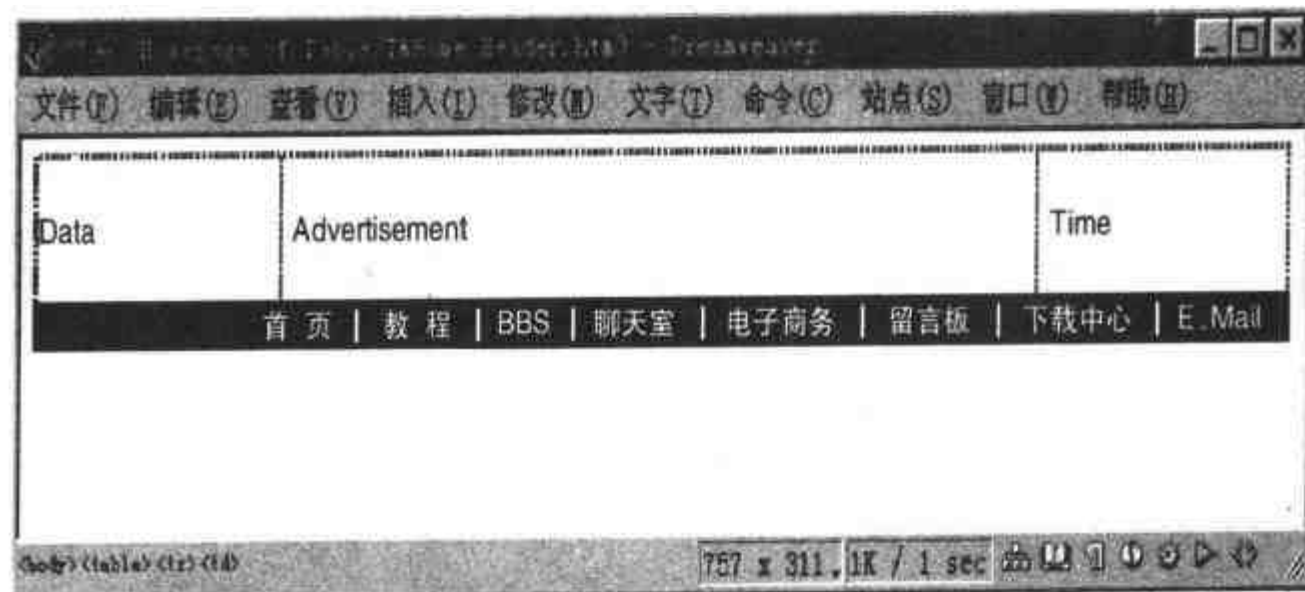


图 2-17 页头




## 2) 过程

这个页面结构比较简单, 可以选择表格也可选择图层排版。但由于前面图层排版讲得太简单, 因此这儿选择图层排版以深入认识图层。

(1) 用前面讲两种方法中的任一种建立 4 个图层。粗略的调整其属性 (包括相对位置)。

(2) 使上三个图层同高, 方法是按住 shift 键, 选中上三个图层, 选择“修改 > 图层和热点 > Make Same Height”菜单选项。

 注意: 三个图层的高度与最后选择的哪个图层的高度一致, 因此, 如果想使某个图层作为其它图层的标准, 那么这个图层一定要最后选择。

同理, 使上三个图层顶端对齐。可以看出, 操作多是“插入…”, 然后“修改…”。

(3) 确信“查看 > 防止层交迭”选项被选中, 选中上三个图层中的某一个, 移动图层使与其相邻的图层之间没有空隙。方法可用鼠标拖拽, 也可以选中图层, 按键盘方向键移动它, 一次移动 1 个像素, 如果同时按下 Shift 键, 一次移动 5 个像素。如果按下 ctrl 键, 改变图层宽或高, 而不是移动图层。还有其它组合键, 请读者试验一下。

用 (2)、(3) 步的方法使页面效果同图 2-17 一样, 除了现在还是图层。

(4) 输入图示文字, 转换为表格。

## 2. 建立主体

### 1) 目标

首先, 还是明确一下目标。如图 2-18 所示。

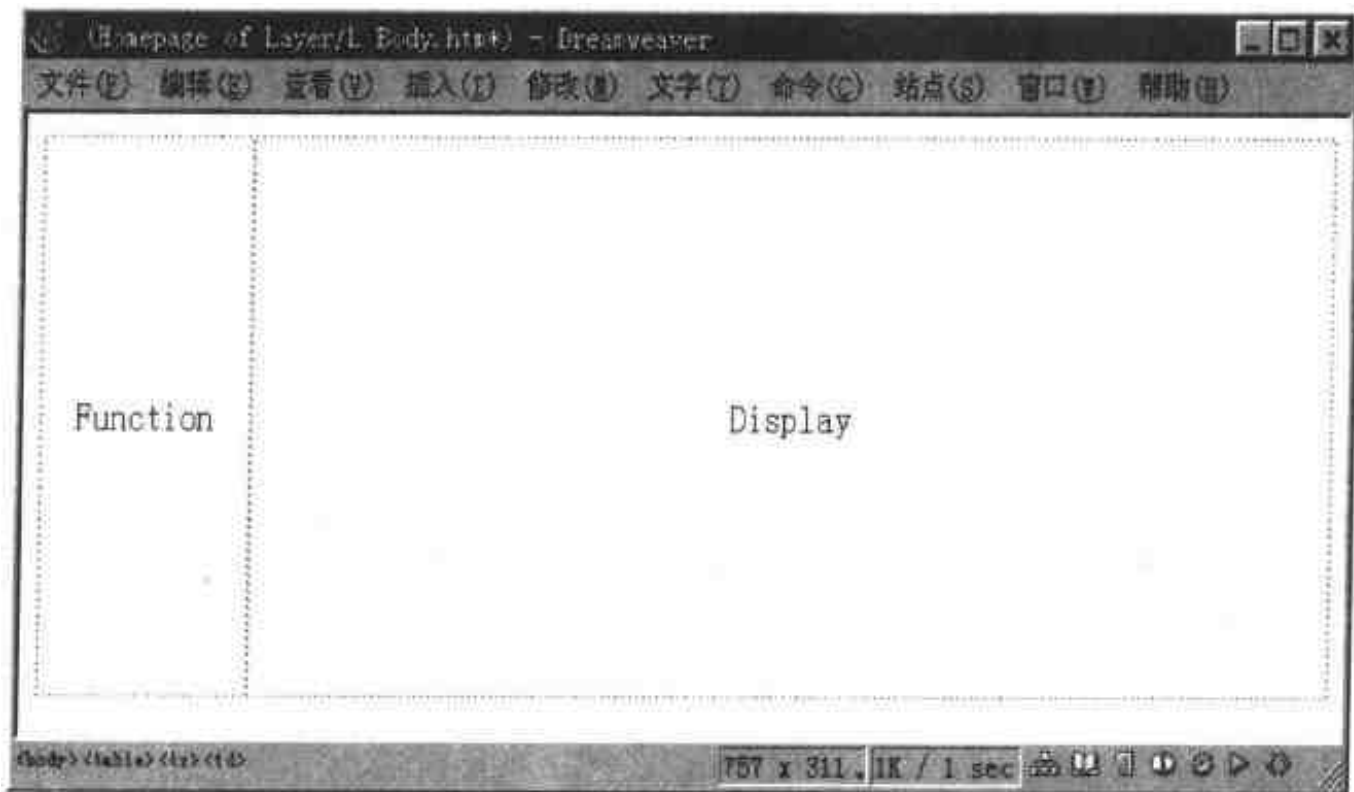


图 2-18 主体

### 2) 过程

非常简单, 请读者千万不要不屑一顾, 后面的变化是显著的。

### 3. 建立页脚

#### 1) 目标

同样，明确一下目标，如图 2-19 所示。

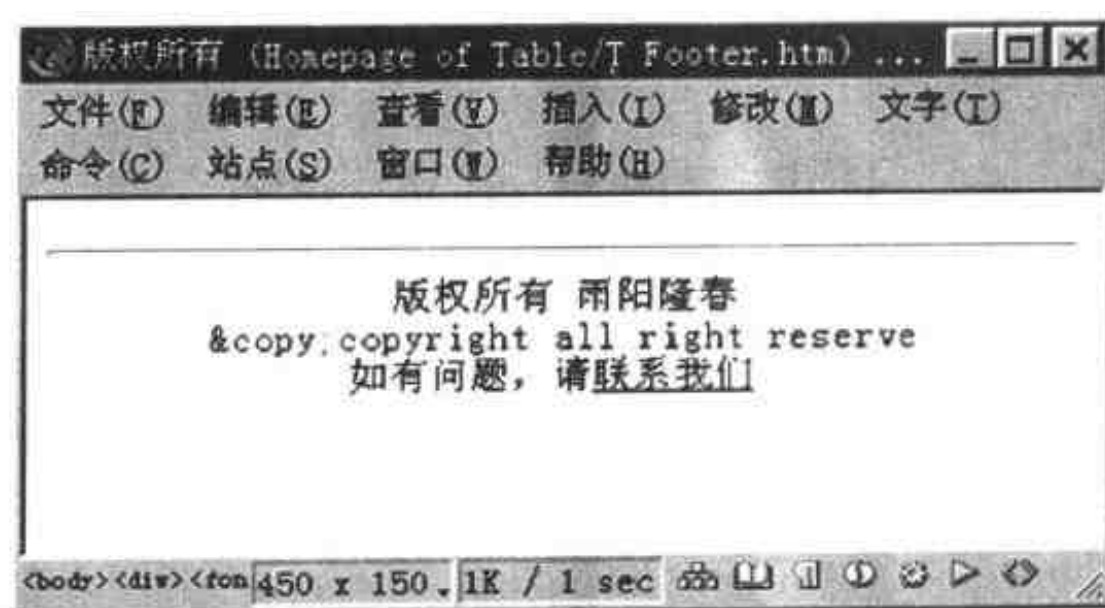


图 2-19 页脚

#### 2) 过程

这个页面也非常简单，但几乎每个页面都少不了。页面中有一个特殊字符（版权），该怎样输入呢？有两种方法：一、选择“插入 > 字符 > 版权”。二、在对象选项板（如图 2-20 所示）中选择。

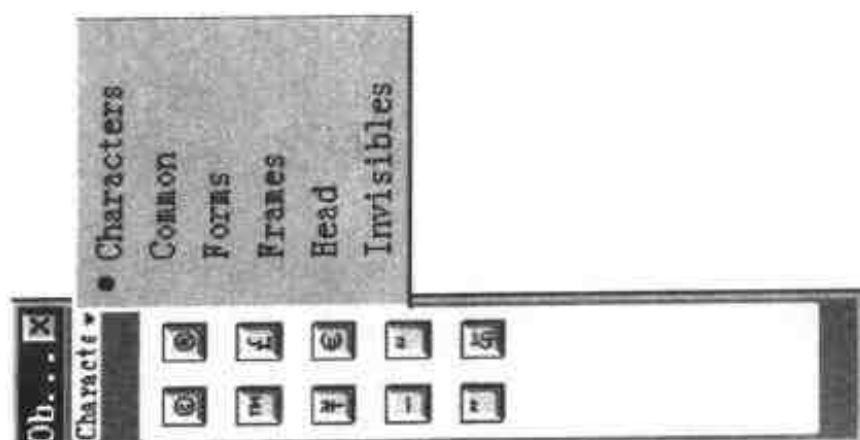


图 2-20 特殊字符选项板

## 2.4 本章小结

这里用了相当大的篇幅讲述最基本的东西。但是，我们不是泛泛而谈，而是采用一种新颖的，规范的方法论述了 HTML 的精华：表格、表单、框架。希望读者能从中巩固知识，并获得某种启示。如果真如所愿，那么我们的目的达到了。同样，Dreamweaver 的操作，我们是通过实例进行阐述的。用 Macromedia 的提法叫“Tutorial”。我们尽量做到清晰、准确。

本章讲述了与本书关系相当密切的 HTML 语言的三个标记和对应的 Dreamweaver 操作。事实上，还讲述了 Dreamweaver 其它操作，如站点管理。希望通过这些讲解，使读者在本书后固的建站过程中游刃有余。

# 第 3 章 JSP 指南

## 3.1 Hello World

一本不写 Hello World 的书算不上一本好书,Hello World 业已成为计算机编程语言书籍的永恒经典。例子绝对简单,但不简单的是它可以给读者关于这种语言的最基本的概念。下面,让我们从 Hello World 开始。

### 例 3-1

```
<HTML>
<HEAD>
<TITLE>Hello World! </TITLE>
</HEAD>
<BODY>
<CENTER>
<% for(int i=1;i<=7;i++) { %>
<FONT SIZE=<%= I %> COLOR=RED>
<% out.println("Hello World!"); %>
</FONT>
<BR>
<% } %>
</CENTER>
</BODY>
</HTML>
```

这个程序可能比想象中的 Hello World 稍微复杂一点,它实现了从 1 递增至 7 放大字体显示 Hello World,页面效果如图 3-1 所示。3 秒钟扫视完程序,发现了什么? 对了,JSP 文档有点像 Java 文档,像 HTML 文档,更像 JavaScript 文档,最像 ASP 文档。从中可以看出:

(1) JSP 文档像 HTML 文档,因为 JSP 是一种嵌入式脚本语言。嵌入式脚本语言简单、小巧,易于与 HTML 语法混合。JSP 保持了嵌入式脚本语言的特色,它继承了 Java 语言语法的简单性并且支持内部对象,使得它更简单更易于与 HTML 混合。可以这样认为,它的出现就是要实现更简便的编写动态页面。

(2) JSP 更像 JavaScript,因为二者都是基于 Java 的动态页面技术。不同的是前者是服务端的技术,而后者是客户端的技术。如果读者有编写 JavaScript 的经验,那么成为 JSP 高手需要做的仅仅是将客户端的经验移植到服务端而已。



图 3-1 Hello World!

(3) JSP 最像 ASP, 因为二者为区别于 HTML 语法, 都使用“<%”和“%>”作分隔符。在“<%”和“%>”之中的就是它们的语法。不同的是 JSP 使用了更多的语法标记, 并且语法标记是可以扩展的。

(4) JSP 有点像 Java, 除去“<%”和“%>”, JSP 程序就是 Java 程序片段, 不需要用类来封装。通过第 1 章的学习, 我们已经知道 JSP 是基于 Java 的技术, 因此很容易理解这点。事实上, 在 JSP 语法中称这样的 Java 程序片段叫脚本片段。

以上四点, 第一点是灵魂, 因为嵌入所以需要分隔符, 又因为不是纯粹的 Java, 不需要大量的、复杂的编程开发经验, 所以显得简单, 灵活。

例 3-1 给了关于 JSP 的最粗略的印象, 结合第 1 章的概念, 我们知道, JSP 就是嵌入 HTML 文档在服务端运行的 Java 小程序。与之比较的是 Java Applet, Java Applet 是嵌入 HTML 文档在客户端运行的 Java 小程序。下面再来看一个例子。

### 例 3-2

```
<! -- This is a more complex Jsp program, you'll know the basic of Jsp -- >
<% -- This is a more complex Jsp program, you'll know the basic of Jsp -- % >
<% @page language = "java" % >
<%! String morning = "Good morning, World!"; % >
<%! String afternoon = "Good afternoon, World!"; % >
<HTML>
<HEAD>
<TITLE>Good morning and Good night Word! </TITLE>
</HEAD>
<body>
<center>
<font size = 5 color = red>
```

```
<jsp:useBean id="data _ timer" scope="page" class="chapter03.JspCalendar"/>
<%if(data _ timer.getAMPM() == 1){%>
<% = afternoon%>
<% } else{ %>
<%out.println(morning);%>
<% } %>
</font>
</center>
</body>
</HTML>
```

例子根据当前系统时间,判断是上午还是下午,显示“Good Morning World!”或者“Good Afternoon World!”。请稍微仔细的看看,程序只有短短的 22 行,真正是 JSP 语言的语句只有 11 行。看上去非常简单,其实,从技术上说,几乎包含了所有的 JSP 语法。第 1、2 句是 JSP 的两种注释,第 1 句注释产生输出到页面源代码中,第 2 句不产生输出到页面源代码中,仅仅是对 JSP 文档的注释。第 3 句是一个 JSP 页面指令,其属性 language 指定了页面使用的脚本语言类型。第 4、5 句是变量声明,声明了两个字符串变量。第 13 句是 JSP 对 JavaBeans 的集成,使用的是<jsp: useBean……/>行为,其中 JspCalendar 是被封装的 JavaBeans 组件。第 14 句是一个脚本片段——if 条件判断语句。if 语句的条件表达式是对第 13 句 JavaBeans 对象属性值引用的判断,判断当前系统时间是上午还是下午。若属性值引用返回 1,则是下午,那么,执行第 15 句,使用 JSP 的表达式显示“Good afternoon, World!”否则,执行第 17 句,使用 JSP 的内部对象 out 显示“Good morning, World!”。第 15 句和第 17 句作用相等,可以互换。剩下的都是 HTML 语句。页面效果如图 3-2 所示。

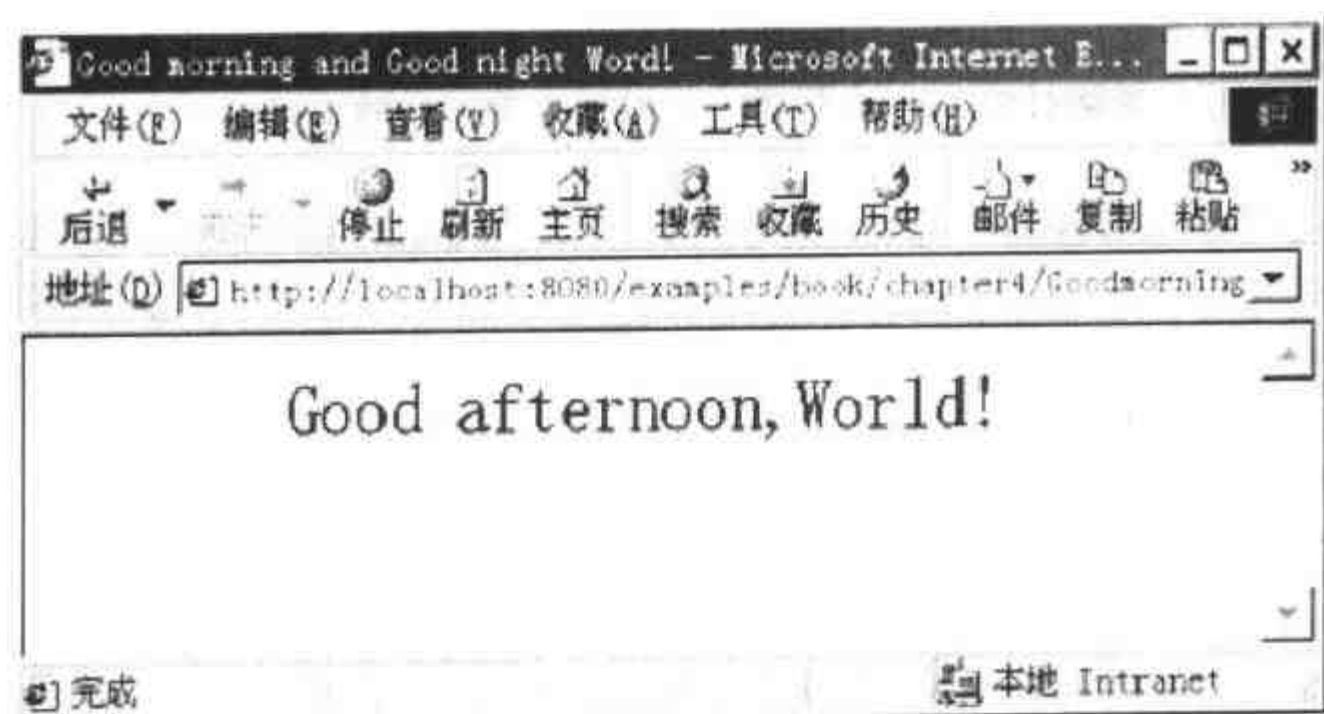


图 3-2 Good afternoon, World!

这个例子中,读者看到了几乎所有的 JSP 语法,包括注释、声明、表达式、指令、行为、内部对象。那么一个 JSP 文档到底可以包括那些语法呢,更进一步,JSP 页面作为页面应

用程序到底可以包含哪些组成部分? 请看 3.2 节。

## 3.2 从组成的角度看 JSP

### 3.2.1 JSP 语法一览

JSP 页面包括那些语法呢? 它包括:

#### 1. 注释(Comment)

- (1) 输出注释(Output Comment):产生输出到客户端页面源代码的注释。
- (2) 隐藏注释(Hidden Comment):不产生输出到客户端页面源代码的注释,即 JSP 页面注释。

#### 2. 指令(Directive)

- (1) 页面指令(Page Directive):定义用于整个 JSP 页面的属性,这些属性的值将传递给容器。
- (2) 包含指令(Include Directive):在翻译(预编译)时将静态资源包含到 JSP 源文件中。
- (3) 标记库指令(Taglib Directive):定义标记库,并加前缀于 JSP 页面的自定义标记。

#### 3. 脚本元素(Scripting)

- (1) 声明(Declaration):声明页面使用脚本语言确定的有效变量或方法。目前 JSP 规范只定义了 Java 这种脚本语言。
- (2) 表达式(Expression):包含页面使用脚本语言确定的有效表达式。目前 JSP 规范只定义了 Java 这种脚本语言。
- (3) 脚本片段(Scriptlet):包含页面使用脚本语言确定的有效代码片段。目前 JSP 规范只定义了 Java 这种脚本语言。

#### 4. 行为(Actions)

- (1) 标准行为(Standard Actions):XML 风格的语法元素,JSP 规范明确规定必须提供的行为。
- (2) 自定义行为(Custom Actions):通过标记库指令扩展的行为。

#### 5. 对象(Object)

- (1) 内部对象(Implicit Object):不用声明可以直接使用的对象,JSP 规范明确规定必须提供的对象。
- (2) 自定义对象(Custom Object):通过标记库指令扩展的对象。

### 3.2.2 JSP 是 Web Application

可以看出,JSP 的语法是相当丰富并且是开放的可扩展的,它为 JSP 胜任 C/S 三层结构模型的中间层的功能要求提供了有力的保障。然而中间层功能到底是什么呢? Microsoft 称之为 Web Computing,即页面计算。Web Computing 使得前端可以使用任何浏览器,后端可以访问任何数据库。从这点看,一个 JSP 页面就是一个 Web Application,即页面应用程序,它是若干 URL 定位的有效资源集合,由以下部分组成:

- (1) 服务端的 Java 运行时环境,这是必要的。
- (2) JSP 页面,处理请求,产生动态内容。
- (3) Servlets,处理请求,产生动态内容。
- (4) 服务端 JavaBeans 组件,封装行为与状态。
- (5) 静态 HTML、DHTML、XHTML、XML 和类似页面。
- (6) 客户端的 Java Applets、JavaBeans 组件和任意的 Java 类文件。
- (7) 客户端的 Java 运行时环境。

可见,一个 JSP 页面应用程序 = (1)|(2)(3)(4)(5)(6)(7)|'。其中,(1)、(2)、(3)、(4)、(5)、(6)、(7)分别代表上述七个组成部分,(1)是必要的,上标“+”代表包含(2)、(3)、(4)、(5)、(6)、(7)至少一个组成部分。这就是一个 JSP 页面应用程序的组成,换句话说,JSP 实现了对 Java、Servlet 的继承,对 JavaBeans、Java Applet、XML 以及 JDBC 的集成。那么 JSP 通过什么实现对上述技术的继承或集成的呢? 语法主要继承自 Java;内部对象主要继承自 Servlet;另外通过<jsp:useBean……/>行为实现了对 JavaBeans 的集成;通过<jsp:plugin……/>行为实现了对客户端 Java Applet 的集成;通过<%@ taglib……%>实现了对 XML 的集成……。从前面的叙述可以看出,JSP 实是新瓶装老酒,它不过是通过一些标记实现对现有一些优秀技术的继承和集成,尤其是基于 Java 的技术。可以这样说,JSP 就是继承者和集成者。

上而我们从组成角度把握了 JSP 是什么——一个 JSP 页面就是一个 Web Application,并得出 JSP 是继承者和集成者的结论。这是本书中非常重要的第二条线索,它决定了本书后续章节的结构:

#### ▼继承者

- ▶第 4 章——Java 基础
- ▶第 5 章——JSP 基本语法
- ▶第 6 章——Servlet
- ▶第 7 章——内部对象
- ▶第 8 章——JSP Container
- ▶第 9 章——JSP 核心 API

#### ▼集成者

- ▶第 10 章——JSP 对 JDBC 的集成
- ▶第 11 章——JSP 对 JavaBeans 的集成
- ▶第 12 章——JSP 对 Applet 的集成

---

▶第13章——JSP 对 XML 的集成

## 3.3 本章小结

本章列举了两个简单的例子,帮助我们从组成(结构)上把握 JSP 是什么。这是本书非常重要的概念和线索,它是后续章节组织的依据。



# 第 4 章 Java 基础

我们知道 Java 从 C++ 发展而来,不仅继承了 C++ 大量的语言要素,还继承了异常处理、多线程的思想。由此看来 Java 也是一个继承者。它继承了大量的优秀技术,同时开创了一种崭新的程序设计语言风格——简单、面向对象、跨平台……。从这种意义上说,Java 真的是前无古人。这些特性赋予其神奇的力量,使其首先在 Internet 上生根发芽,进而遍及应用领域。我们欣喜地看到,Sun 与其 Java 事业部 JavaSoft 为巩固 Java 的地位正忙于建立一个 Java 王国。从客户端 Applet 到服务端 Servlet,从组件技术 JavaBeans 到数据库技术 JDBC……。

## 4.1 Java 程序设计基础

### 4.1.1 Java 应用程序的组成

Java 应用程序是可以独立运行的 Java 程序,它由一个或多个类组成,并且其中必须有一个类定义了 main()方法。main()方法就像 C 语言的 main 函数一样是 Java 应用程序的起始点。还是举这个经典的例子:HelloWorld.java。

```
public class HelloWorld{  
    public static void main(String arg[]){  
        System.out.println("Hello World!");  
    }  
}
```

Java 应用程序由一个或多个编译单元组成,每个编译单元最多包含下列内容(空格和注释除外):

- (1)package 语句
- (2)import 语句
- (3)类
- (4)界面

每个 Java 的编译单元可包含多个类和界面,但最多只能有一个类或界面声明为 public。其中类必须包含类说明、类体、方法定义;可以包含方法体、类继承、实现界面。Java 应用程序中,必须有 main()方法,它是程序运行的人口。其一般格式如下:

```
public static void main(String arg[])  
    void:指明返回类型。
```

public:指定 main()方法的访问权限,它表明 main()方法可被其它对象调用。  
static:说明 main 方法是一个静态方法,即 main()方法是类方法。

## 4.1.2 Java 程序设计基础

### 1. 标识符命名

标识符是对变量、类、方法、标号和其它用户自定义对象的命名。Java 中标识符必须以字母、下划线或美元符开头,后面跟 0 个或多个由字母、下划线、美元符或数字组成的字符数字串。

例:

```
pen, _con, $(s)
```

### 2. 变量类型

Java 语言规定使用变量之前必须先定义。变量定义的一般形式为:

```
type variable _list;
```

type 是某种有效的 Java 类型,variable \_list 是变量列表。当有多个变量时,各变量之间用逗号分隔。Java 可在变量说明的同时进行初始化。

例:

```
int i,j,k;
```

```
int i = 1;
```

Java 基本数据类型如表 4-1 所示:

表 4-1 Java 基本数据类型

类型名称	声明关键字	宽度	范围
整数	byte	8	-128 ~ 128
	short	16	-32768 ~ 32767
	int	32	-2147483648 ~ 2147483647
	long	64	-9223372036854775808 ~ 9223372036854775807
浮点数	float	32	1.4E-45 ~ 3.4028235E38
	double	64	4.9E-324 ~ 1.7976931348623157E308
字符	char	16	
布尔	Boolean	1	

### 3. 常量

Java 常量包括整数、浮点数、布尔、字符和字符串五种。这里编写一个小程序,打印整

数、浮点数的范围。

#### 例 4-1

打印整数和浮点数的最大、最小值。

```
public class Sp01_MinMax{
    public static void main(String arg[]){
        System.out.print("Byte:" + Byte.MIN_VALUE + "~");
        System.out.println(Byte.MAX_VALUE);
        System.out.print("Short:" + Short.MIN_VALUE + "~");
        System.out.println(Short.MAX_VALUE);
        System.out.print("Integer:" + Integer.MIN_VALUE + "~");
        System.out.println(Integer.MAX_VALUE);
        System.out.print("Long:" + Long.MIN_VALUE + "~");
        System.out.println(Long.MAX_VALUE);
        System.out.print("Float:" + Float.MIN_VALUE + "~");
        System.out.println(Float.MAX_VALUE);
        System.out.print("Double:" + Double.MIN_VALUE + "~");
        System.out.println(Double.MAX_VALUE);
    }
}
```

结果就是表 4-1 各类型的表示范围。

## 4. 运算符与表达式

表达式是运算符、常量和变量的组合。Java 的表达式即可以是单独组成语句,也可出现在循环条件测试、变量说明、方法的调用参数等场合。运算符是组成表达式的关键,下面我们简单介绍一下 Java 语言的各种运算符。Java 语言的运算符与标准 C 基本相同,但需要注意两点:首先 Java 是强类型的语言,类型限制比 C 语言严格。其次是 Java 不支持 C 的逗号运算符、指针运算符;同时新增对象操作符“instanceof”、字符串运算符“+”和零填充右移“>>>”。

Java 所有运算符如表 4-2 所示。

表 4-2 Java 运算符一览

优先级	运算符	操作数类型	操作数个数	结合性	含义
1	++	算术	前缀	R	自增
	--	算术	前缀	R	自减
	+, -	算术	单目	R	一元加, 一元减
	~	整型	单目	R	按位取反
	!	布尔	单目	R	逻辑非
	(type)	任意	单目	R	强制类型转换

续表

优先级	运算符	操作数类型	操作数个数	结合性	含义
2	<code>*</code> , <code>/</code> , <code>%</code>	算术	双目	L	乘, 除, 余
3	<code>+</code> , <code>-</code>	算术	双目	L	加, 减
	<code>+</code>	字符串	双目	L	字符串合并
4	<code>&lt;&lt;</code>	整型	双目	L	左移
	<code>&gt;&gt;</code>	整型	双目	L	继承符号右移
	<code>&gt;&gt;&gt;</code>	整型	双目	L	零填充右移
5	<code>&lt;</code> , <code>&lt;=</code>	算术	双目	L	小于, 小于等于
	<code>&gt;</code> , <code>&gt;=</code>	算术	双目	L	大于, 大于等于
	<code>instance of</code>	对象	双目	L	类型比较
6	<code>==</code>	基本类型	双目	L	等于运算
	<code>!=</code>	基本类型	双目	L	不等运算
7	<code>&amp;</code>	整型	双目	L	按位与
	<code>&amp;</code>	布尔	双目	L	逻辑与
8	<code>^</code>	整型	双目	L	按位异或
	<code>^</code>	布尔	双目	L	逻辑异或
9	<code> </code>	整型	双目	L	按位或
	<code> </code>	布尔	双目	L	逻辑或
10	<code>&amp;&amp;</code>	布尔	双目	L	条件与
11	<code>  </code>	布尔	双目	L	条件或
12	<code>?:</code>	布尔, 任意	三目	R	三元条件运算符
13	<code>=</code>	变量, 任意	双目	R	赋值
	<code>*=</code> , <code>/=</code> , <code>%=</code> , <code>+=</code> , <code>-=</code> , <code>&lt;&lt;=</code> , <code>&gt;&gt;=</code> , <code>&gt;&gt;&gt;=</code> , <code>&amp;=</code> , <code>^=</code> , <code>=</code>	变量, 任意	双目	R	带运算符的赋值

## 5. 流程控制

### 1) 条件语句

Java 语言支持两种类型的条件语句: `if` 和 `switch`。

#### (1) `if` 语句

if 语句的一般形式为:

```
if(expression) statement;  
else statement;
```

其中 statement 可以是单独的语句,也可以是用“{}”括起来的块语句。

需要注意的是 Java 中 if 语句的条件判断表达式值必须是一个布尔类型,而不能像 C 语言那样可以是其它类型的值,否则会产生编译错误。

## (2) 多重嵌套的 if 语句

其一般形式如下:

```
if(expression) statement;  
else  
    if(expression) statement;  
    else  
        if(expression) statement;  
        .....  
        else statement;
```

## (3) switch 开关语句

在有多重条件判断的情况下,switch 语句是高效和简洁的。同大多数其它语言一样,Java 也提供了多分支选择 switch 开关语句,它根据表达式的值在多个分支语句中决定执行哪个分支语句。它与 if 语句不同,只能进行等值测试。其一般形式为:

```
switch(expression) {  
    case constant 1:  
        statement sequence  
        break;  
    case constant 2:  
        statement sequence  
        break;  
    .....  
    default:  
        statement sequence  
}
```

## 2) 循环语句

### (1) for 循环

for 循环非常有用,其一般形式为:

```
for(initialization; condition; increment) statement;
```

for 循环,条件测试在循环开始之前。它首先执行初始化部分,然后执行条件判断部分,根据判断结果,或执行 for 循环体,或推出 for 循环。若执行 for 循环,执行完循环体后执行增量部分。再判断条件……。

for 循环非常灵活,有如下几种变化:

for( ; ; ) statement; 实现无限循环。

`for(int i=0;i<somevalue;i++)`;无循环体,产生时间延迟。

`for(x=0,y=0;x+y<10;x++)statement`;多条件决定循环。

## (2) while 循环

while 循环的一般形式是:

```
while(condition) statement;
```

其中条件表达式的值必须是逻辑类型。当条件为真时,循环重复;当条件为假时,控制转入紧跟循环体的那行代码开始执行。

## (3) do-while 循环

其一般形式是:

```
do|
    statement;
}while(condition);
```

与 for 和 while 循环不一样,do-while 循环的条件检测在循环的尾部,这意味着这 do-while 循环至少会被执行一次。

## 3) 转移语句

Java 语言有三种无条件的转移语句: `return`、`break` 和 `continue`。

### (1) return 语句

其一般形式是:

```
return expression;
```

作用是使程序从方法中返回,并为方法设置一个返回值。程序一遇到 `return` 语句就立即从方法体中返回。

### (2) break 语句

其一般形式是:

```
break label;
.....
```

```
toHere:statement
```

作用是从一个封闭的语句跳出,典型的是从 `switch` 开关语句中跳出。其中 `label` 指定一个封闭语句的标号,如格式中的 `toHere`,但标号必须是合法的 Java 标识符。

### (3) continue 语句

其一般形式是:

```
continue break;
```

`continue` 语句必须用于循环结构中,在 `for` 循环语句中遇到 `continue` 后首先进行条件测试,然后执行循环增量部分。在 `while` 和 `do-while` 循环中,`continue` 语句使控制直接回到条件测试部分。

## 6. 数组与字符串

数组是 Java 的一种复合数据类型,也是一种特殊的对象。Java 的字符串也是对象,分为不变字符串和可变字符串。不变字符串是 `java.lang.String` 类的实例,可变字符串是 `java.lang.StringBuffer` 类的实例。

### 1) 数组的定义、创建与释放

Java 的数组与 C 语言是不同的,它在定义时不允许指定数组的大小,它通过 new 运算符显示创建,或通过初始化方法隐藏创建。

数组说明的一般形式:

type var \_name[]; 或 type[] var \_name;

其中 type 是数组元素的类型,它可以是任意的 Java 类型,当然也可以是数组。

在 Java 程序中,一般将数组的定义和创建连在一起使用。

例:

```
char s[] = new char[3];
```

还可以通过静态初始化创建。

例:

```
int lookup_table[] = {1,4,5,7}
```

### 2) 数组元素的访问

数组元素的访问是通过下标实现的。Java 数组的第一个元素下标是 0,最后一个元素的下标是数组大小减 1。数组的大小可通过 length 获得。

例:

```
int a[] = new int[10]
```

```
a[0] = 10;
```

```
for(int i=1;i<a.length;i++) a[i] = a[0]-i;
```

### 3) 字符串的创建

不变字符串与可变字符串都是对象,因此它们也是利用 new 运算符创建。

例:

```
StringBuffer sb = new StringBuffer();
```

```
String str = new String("abc");
```

### 4) 字符串的访问方法

#### (1) 获得长度

```
int len = str.length();
```

#### (2) 返回指定位置字符

```
char c = str.charAt(i);
```

#### (3) 确定指定字符或字符串在给定字符串中的位置

```
int i = str.indexOf("ab");
```

```
int j = str.lastIndexOf("bc");
```

#### (4) 获得子串

```
String s = str.substring(2,2);
```

substring()方法有数种形式,请读者注意一下。我们将在第 8 章综合例子留言板中使用到上述方法。

### 5) 修改可变字符串

#### (1) 后加

使用 append()方法将一个字符加到当前字符串的末尾。StringBuffer 类提供了将各



种数据类型添加到字符串末尾的方法,包括对象。

(2) 插入

insert()方法将一个字符加到可变字符串中。

例:

```
StringBuffer sb = "this is mistake sentence";
```

```
sb.insert(7, " a");
```

6) 其它重要方法

(1) 将对象转化成不变字符串。

例:

```
sb.toString();
```

(2) String 类的静态方法 valueOf(),将不同类型的变量转化成字符串。

例:

```
System.out.print(String.valueOf(Long.MAX_VALUE));
```

## 7. 异常处理

异常处理其实也是一种流程控制,期望将错误流程恢复为正常指令流。当然,许多异常是不可恢复的,如磁盘物理损坏。Java 的异常处理机制由 try/catch/finally 三个语句组成,其一般格式如下:

```
try{
    statement;
}
catch(.....){
    statement;
}
catch(.....){
    statement;
}
finally{
    statement;
}
```

其中 try 块管理包含于其中的语句,定义与之相关的异常的范围;try 块后面跟着零个或多个由 catch 语句引导的块,catch 块把异常与 try 块联系起来,它负责处理指定类型的异常;catch 块后面可以跟一个由 finally 引导的块,它包含清除程序理场的语句。不论怎样,都将执行 finally 块。

## 4.2 Java 面向对象程序设计

面向对象是 Java 语言最重要的特征之一,它是完全面向对象的程序设计语言。当

然,为了使语言简单和高效,它仍保留了一些面向过程的特征,如简单的基本变量类型。但这并不意味着它支持面向过程的程序设计。本节我们介绍 Java 的各种面向对象的特征,主要内容有对象的使用、类的定义和实现、类成员变量的说明、方法的定义和实现。

### 4.2.1 对象

类和对象是面向对象程序设计的核心。类是创建对象实例的样板,它包含所创建对象的状态描述和方法定义。对象是类的实例化,它有自己的独特的、特定的特征。如我们使用的 Java 语言,可以定义一个名为“Java 语言”的类,使用的 Java 语言的实现如 JDK1.3 便是这个类的一个实例。

#### 1. 创建对象

Java 中通过类来创建具体的对象,即对象通过实例化的类来实现。一般情况下使用 new 运算符创建新的对象。

##### 1) 说明对象

对象说明的一般形式是:

```
type name;
```

其中,type 是一个类名,说明对象所属的类,name 是对象名。对象说明只是建立了一个如同 C 语言内的指针,以使用来引用该类型的对象。

##### 2) 实例化、初始化对象

对象一般使用 new 运算符进行实例化。其一般形式是:

```
new type();
```

其中 type 是 new 的参数,它代表一个构造方法。该构造方法可能有参数。new 运算符返回新创建对象的引用,这个引用通常赋值给一个合适的变量,即将说明对象与实例化对象合并起来,一般形式是:

```
type name = new type();
```

新创建的对象通过构造方法初始化。一个类可以有多个构造方法,不同的构造方法通过不同的参数区分。

##### 3) 使用对象

引用对象的变量与引用 C 语言的结构元素相似,其方法是在对象名后使用需要引用的变量,中间用“.”将它们连接起来。一般形式是:

```
objectReference.variable
```

调用对象的方法与引用对象的变量一样是将方法名接到对象引用后,中间用“.”连接,同时在圆括号内提供所需参数。一般形式是:

```
objectReference.mothodName([argumentlist]);
```

方法调用实际上就是给指定对象发送消息。

### 4.2.2 类

Java 的类是一种复合数据类型,类是对象的模板,是一组具有共同状态和行为对象的

抽象。Java 类的实现包括两部分内容:类的说明和类体,其一般形式是:

```
ClassDeclaration{
    classBody
}
```

## 1. 类说明

类说明包括关键字 `class`、类名及类的属性。类的属性用来说明类属于哪个超类、类的访问限制、类是否为抽象类、是否为最终类等等。

类说明中必须包括关键字 `class` 和自定义类名,其形式是:

```
class ClassName{
    .....
}
```

Java 编译器将这种形式的类说明为非最终、非公有、非抽象、不实现任何界面、超类为 `Object`。

最常用的形式是:

```
public class ClassName{
    .....
}
```

完整的形式是:

```
[modifiers] class ClassName [extends SuperClassName] [implements Interface-
Name
List] {
    .....
}
```

各部分功能如下:

- (1) `modifiers`: 类修饰符, 可以是 `abstract`、`final`、`public`、`private` 及它们的有效组合。
- (2) `public`: 说明一个类可以被类所属的包之外的类或对象使用。
- (3) `private`: 说明该类只能被同一个包中的其它类使用。如果类说明中没有说明访问权限, 默认为 `private`。
- (4) `abstract`: 说明该类不能直接实例化为对象。
- (5) `final`: 说明该类不能再有子类。
- (6) `ClassName`: 类名
- (7) `extends` 子句: 说明类的超类, 其中 `SuperClassName` 是超类名。
- (8) `implements` 子句: 说明类中将要实现的界面, 其中 `InterfaceName List` 是一个逗号分隔的界面列表。

## 2. 类体

类体中一般首先说明类的成员变量, 然后提供方法的说明和实现, 其一般形式是:

```
classDeclaration{
```

```
    memberVariableDeclarations
    methodDeclarations
}
```

#### 成员变量的说明

成员变量用来表示、维持类的状态。类成员变量的说明必须在类体中,但不能包含在方法体中。其一般形式是:

```
type variableName;
```

完整形式是:

```
[VariableModifier] type variableName;
```

其中 VariableModifier(变量修饰符)类型如下:

(1)访问权限修饰符:说明变量的访问权限。

(2)public:允许所有的类访问。

(3)protected:允许类自身、子类(有一定限制)以及同一个包中的所有类访问。

(4)private protected:允许类自身及它的子类访问。

(5)private:只能被定义它的类访问。

(6)friendly:没有显式给出访问权限指示符的变量称为友好变量。

静态变量(static):由类的所有实例化对象共享,编译实现中,静态变量仅有一份拷贝,因此它也被称为类变量。它与实例变量是不同的,同一个实例变量在类的不同实例化对象中是不同的。静态变量可以由类创建的对象引用,也可通过类直接引用。

常量(final):常量标识符通常都用大写字母。

### 3. 方法定义和实现

与类的实现相似,方法的实现包括两部分:即方法说明和方法体。其一般形式是:

```
method Declaration {
    method Body
}
```

#### 1) 方法说明

最简单的方法说明形式如下:

```
returntype methodName() {
    .....
}
```

Java 要求在方法定义中必须说明返回值的类型。如果一个方法不需要返回值,则返回值类型需要说明为 void。因此,除了返回类型定义为 void 的方法外,其它方法必须包含 return 语句。

方法访问权限的说明形式、含义与成员变量的访问权限说明完全一样。

同样,类方法对整个类也是共享的,类方法仅能对类变量进行操作,在类方法中不能引用实例变量,也不能引用实例方法。

Java 中,简单数据类型的参数是传值的,调用方法时,方法接收变量传入的值,建立该变量的一个拷贝,而不是直接引用变量本身。这种特性给程序带来一定的安全性,但却限

制了灵活性。为了让方法能改变参数,必须传入一个对象,Java 中的对象也是传值的,然而对象的值是一个引用,因此实际上传递的是对象地址的引用,从而达到传址的效果。

## 2) 方法体

方法体中可以使用两个特殊的变量: `this` 和 `super`。这两个变量在使用前都不需要说明, `this` 变量指向当前的对象。当前对象是指调用当时正在执行方法的那个对象; `super` 变量用来引用超类中的变量和方法。

## 3) 构造方法、结束方法

构造方法是用来初始化新创建对象的特殊方法。构造方法不能有自己的名字;构造方法也不能从超类中继承。构造方法说明中的修饰符只能是访问控制指示符,即它是 `public`、`protected`、`private`、`private protected` 之一。



注意:构造方法不能直接通过名字引用,而必须由 `new` 运算符或 `Class` 类的 `newInstance` 方法使用。

还有在构造方法体中可以调用这个类或其超类中的另一个构造方法,但这个调用必须是方法体的第一条语句。其一般形式是:

```
{  
    [this([ArgumentList]);]  
    [super([ArgumentList]);]  
    Blockbody  
}
```

结束方法 `finalize()` 类似 C++ 的析构函数,它用来释放对象,以有效的进行垃圾收集。

## 4.2.3 打印杨辉三角

对 Java 的介绍可以说是简单的、粗略的。它的作用仅仅让您回顾一下 Java,或者帮您解决某个语法结构不清楚的问题。最初我们并不打算编写这一章,但考虑到在学习 JSP 之前回顾一下 Java 是非常有必要的,要学好 JSP,应该对 Java 有相当的认识程度。对这一点,也许在你今后的实践中能够得到印证。

下面举个简单的例子,帮助读者找找感觉。

### 例 4-2

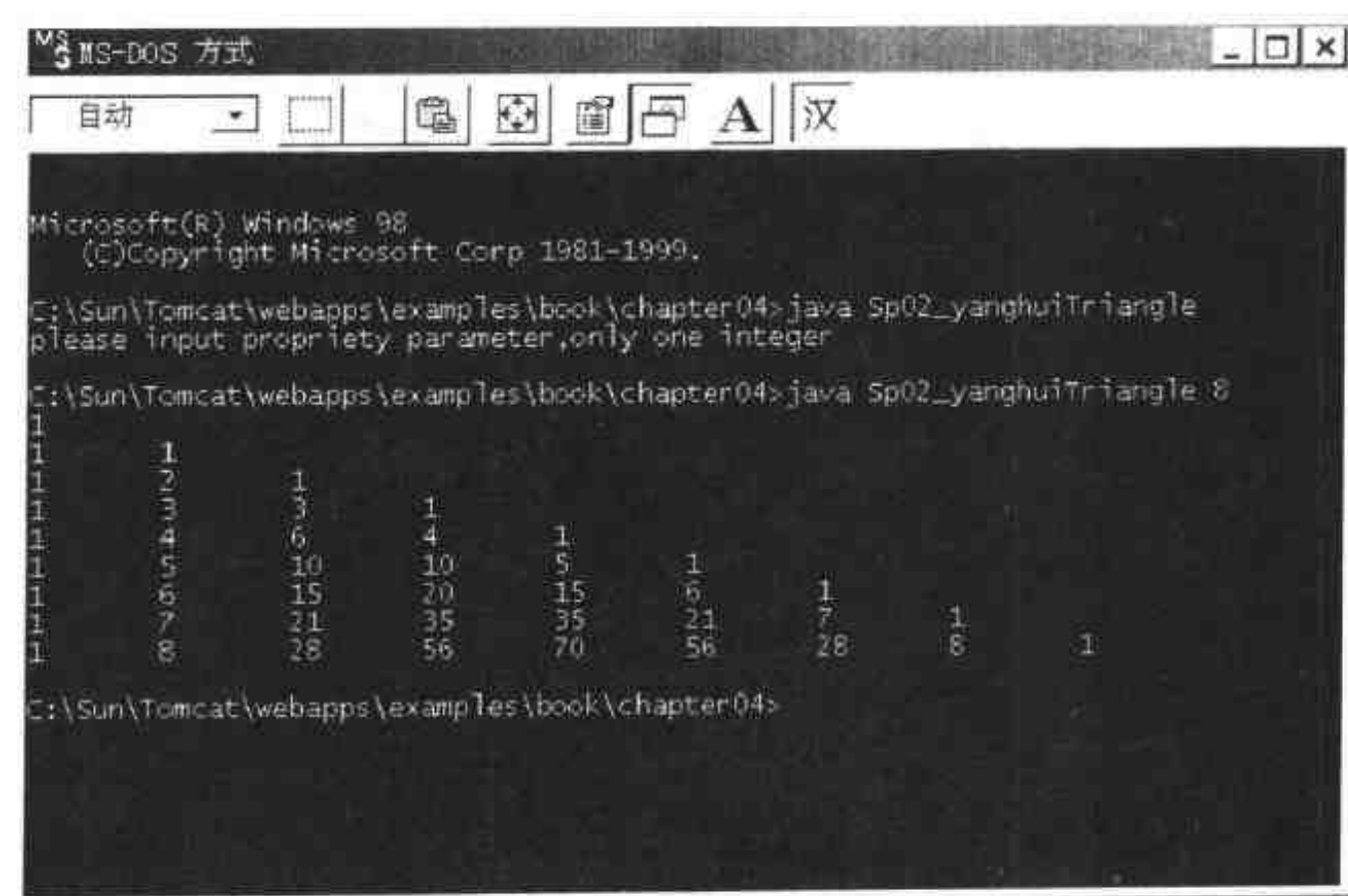
打印杨辉三角

本例采用递归算法,从思想上是比较自然的,但确实不是效率最高的算法。程序如下:

```
public class Sp02 _ yanghuiTriangle{  
    private int cycle;  
  
    public static void main(String arg[]){  
        if(arg.length == 1){
```

```
        Sp02 _ yanghuiTriangle triangle =  
        new Sp02 _ yanghuiTriangle(Integer.parseInt(arg[0]));  
        triangle.printTriangle();  
    }  
    else{  
        System.out.println("please input propriety paramctcr,only one integer");  
    }  
}  
  
public Sp02 _ yanghuiTriangle(int i){  
    cycle = i;  
}  
  
public void printTriangle(){  
    for(int i=0;i<=cycle;i++){  
        for(int j=0;j<=i;j++){  
            System.out.print((int)recursion(i,j));  
            System.out.print(" \t");  
        }  
        System.out.println();  
    }  
}  
  
private float recursion(int i,int j){  
    if(j==0){  
        return 1;  
    }  
    else{  
        fi = (float)i;  
        float fj = (float)j;  
        float each = (fi-fj+1)/fj;  
        return each * recursion(i,j-1);  
    }  
}
```

程序效果如图 4-1 所示。



```
MS-DOS 方式
自动
Microsoft(R) Windows 98
(C)Copyright Microsoft Corp. 1981-1999.
C:\Sun\Tomcat\webapps\examples\book\chapter04>java Sp02_yanghuiTriangle
please input propriety parameter,only one integer
C:\Sun\Tomcat\webapps\examples\book\chapter04>java Sp02_yanghuiTriangle 8
1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
C:\Sun\Tomcat\webapps\examples\book\chapter04>
```

图 4-1 打印杨辉三角

## 4.3 本章小结

本章非常简略地讲述了 Java 的基本知识,主要提了一下它的基本语法。还是那句话,本章目的仅仅当您忘记某个语法的时候使用一下。我们的观点是学习 JSP 之前,至少应对面向对象程序设计有所了解,最好对 Java 的类库(包)多了解一些。下面列举一些从本书出发认为比较重要的包,它们是 `java.lang`,`java.io`,`java.util`,`java.sql`。



# 第5章 JSP 基本语法

JSP 基本语法有“<%...%>”、“<%=...%>”、“<%!...%>”、“<jsp:.../>”等数种标记。纷繁的标记就风格而言有两种。一种是 XML 风格的,例如“<jsp:forward page=‘...’/>”;另一种是脚本语言风格的,例如“<%...%>”,“<%=...%>”。更奇妙的是它的标记是可扩展的。

学习本章最好拥有如下知识:

Java 程序设计基础,包括标识符命名、变量类型、常量、运算符、表达式和流程控制。  
HTML 基本知识。

第3章使我们在整体上对 JSP 语言组成有了一定的认识。本章将深入一些学习 JSP 的语法。JSP 语法标记可分为两类,详细列表如下。本章的内容结构正是分类结构。

## ▼(元素(Elements))

### ▼脚本元素

- ▶声明
- ▶脚本片段
- ▶表达式

### ▼注释

- ▶输出注释
- ▶隐藏注释

### ▼指令

- ▶页面指令
- ▶标记库指令
- ▶包含指令

### ▼行为

- ▶<jsp:forward>
- ▶<jsp:include>
- ▶<jsp:plugin>
- ▶<jsp:getProperty>
- ▶<jsp:setProperty>
- ▶<jsp:useBean>

## ▼Template Data

- ▶Template Data
- ▶Template Text

## 5.1 元素

元素(Elements)就是 JSP 容器能识别的元素类型实例。元素的类型描述了它的语法与语义。如果元素有属性值,类型也描述属性名、属性的有效类型与解释。JSP 有四种类型的元素:脚本、注释、指令与行为。

### 5.1.1 脚本元素

脚本元素一般用于操纵对象和执行计算以产生动态效果。JSP 有三种脚本元素:声明、脚本片段、表达式。

声明用来声明其它所有脚本元素都可以使用的变量和方法。脚本片段,即程序片段用来描述处理请求,响应客户的行为。脚本片段也可用在 JSP 页面中反复地或有条件地执行其它元素,即流程控制。表达式必须是完整的,响应时计算它的值,计算结果通常被转换成字符串,然后写入当前 out 流。

#### 1. 声明

##### 1) 语法

```
<%! Declaration(s) %>
```

##### 2) 语义

JSP 页面使用的某种脚本语言中的声明语法用来声明变量和类(方法)。声明应当是符合脚本语言语法规定的、完整的、有序的语句。目前,JSP 只实现了脚本语言为 Java 的版本,因此自然要遵循 Java 的语法规则了。

声明不产生任何输出到当前 out 流。声明在 JSP 页面初始化时被创建,以使其它声明、脚本片段、表达式有效。这是声明必须有序的原因。声明的同时也可以赋予其初值,即将其初始化。

所有的声明,其作用域是整个 page 范围,即整个 JSP 页面的任何一部分都可以存取它,包含所有的方法(类的方法,注意类的方法与类方法是不同的概念。类方法请参见第 4 章)以及<%@ include... %>指令包含进来的资源。同样,也可以直接使用<%@ include... %>指令包含进来的声明过的变量和方法(类的方法)。

如果需要在同一个配对分隔符“<%!”和“%>”内声明多个同类型变量,变量之间使用逗号“,”分隔。



注意:声明必须以分号“;”结尾。

例:

```
<%! int a,b,c; %>
```

```
<%! String stra = "a", strb = stra + "b", strc; %>
```

例:声明一个类。

```

<%! public class Strexpand{
    private String strexp;
    public Strexpand(String s){
        .....
    }
    public String placeStr(String oldstr,String newstr){
        .....
    }
}
%>
<%
String sall = "<font size=4>";
Strexpand sexp = new Strexpand(sall);
sall = sexp.placeStr("<","&lt;");
%>

```

## 2. 脚本片段 (scriptlets)

### 1) 语法

```
<% scriptlets %>
```

### 2) 语义

脚本片段可以包含任意正确的脚本语言代码片段。代码片段正确与否取决于脚本语言的细节。目前的 JSP 版本中,脚本片段只能是合法的 Java 程序片段。

脚本片段在请求时被执行。它们是否产生输出到 out 流取决于脚本片段中的实际代码。

脚本片段中也可声明变量,但是声明的同时必须赋予初值。

因为脚本片段中只能是合法的 Java 程序片段,所以任何文本、HTML 标记、其它类型 JSP 语法元素都不能包含在其内。



注意:同样,脚本片段也必须以分号“;”结尾。

例:打印九九乘法表

```

<%
for(int i=1;i<=9;i++){
    for(int j=1;j<=i;j++){
        out.print(i*j);
        out.print("\t");
    }
    out.println("<br>");
}
%>

```

例:

```
<%  
String indexid = request.getParameter("id");  
if(indexid == null || indexid.equals("")){  
    indexid = "1";  
}  
if(indexid.equals("1")){  
%>  
<jsp:include page = "news/newstitlelist.jsp" flush = "true" />  
<% } %>  
%>  
if(indexid.equals("11")){  
%>  
<jsp:include page = "news/newscontentdis.jsp" flush = "true" />  
<% } %>
```

### 3. 表达式

#### 1) 语法

```
<% = exprection %>
```

#### 2) 语义

JSP 中表达式元素就是脚本语言的表达式。它在 HTTP 请求时求值,其值将被强制转换成 String 类对象,然后写入当前 out 对象的适当位置。如果表达式的结果不能被强制转换成 String 类对象,那么将产生一个翻译错误,或者翻译时不进行强制类型检查,待到请求时,引发一个 ClassCastException 异常。

以某种脚本语言书写的表达式内容必须是完整的。目前,必须符合 Java 程序设计语言规范。表达式是从左到右求值的。如果一个表达式出现在多个元素的(运行时)属性中,它们仍是从左到右的求值。



注意:表达式是唯一不用分号“;”结尾的语句。

#### 例 5-1

表达式试验。(光盘 sample5 \_ 1 \_ 1.jsp)

```
<% @ page contentType = "text/html; charset = gb2312" %>  
<% @ page import = "java.util. *" %>  
<%  
    int i = 2;  
    String str = "个字符串! 现在北京时间";  
    Date date = new Date();  
%>  
<HTML>
```

```
<HEAD>
<TITLE>表达式试验</TITLE>
</HEAD>
<BODY>
<h2>表达式试验</h2>
<font size = < % = i + 2 % > color = blue >
< % = "这儿有" + i + str + date.getHours() + ":" + date.getMinutes() + ":"
+ date.getSeconds() % >
</font>
</BODY>
</HTML>
```

页面效果如图 5-1 所示。

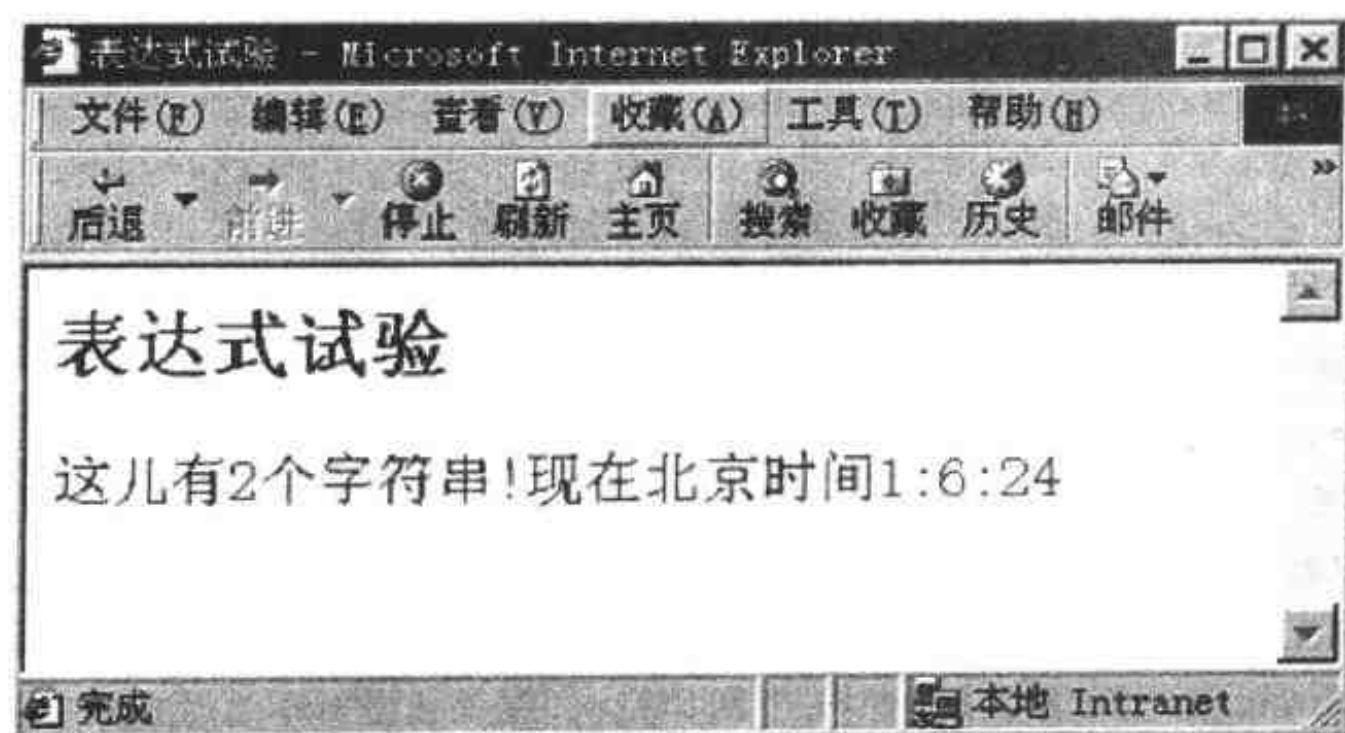


图 5-1 表达式试验

其实,JSP 引擎将表达式转换成 `out.print(表达式内容)`。上例中,表达式 `< % = i + 2 % >` 被转换成:

```
out.print(i + 2);
```

`< % = "这儿有" + i + str + date.getHours() + ":" + date.getMinutes() + ":" + date.getSeconds() % >` 被转换成:

```
out.print("这儿有" + i + str + date.getHours() + ":" + date.getMinutes() + ":" +
date.getSeconds());
```

### 例 5-2

脚本元素综合应用——杨辉三角在页面中的显示。

将第 4 章的例子——打印杨辉三角——移植到页面上,将会发现 HTML 真的是一大创举。源代码如下:

```
< %!
    public class yanghuiTriangle{
        private int cycle;
```

```
public yanghuiTriangle(int i){
    cycle = i;
}

public void setCycle(int i){
    cycle = i;
}

public int getCycle(){
    return cycle;
}

public float recursion(int i,int j){
    if(j == 0){
        return 1;
    }
    else{
        float fi = (float)i;
        float fj = (float)j;
        float each = (fi - fj + 1)/fj;
        return each * recursion(i,j - 1);
    }
}

% >
<HTML>
<HEAD>
<TITLE>脚本元素综合应用——杨辉三角</TITLE>
</HEAD>
<body>
<div align = center><b2>杨辉三角</b2>
<% ! int cycle; % >
<%
yanghuiTriangle triangle = new yanghuiTriangle(9);
cycle = triangle.getCycle();
for(int i=0;i<=cycle;i++){
    for(int j=0;j<=i;j++){
% >

        <% = (int)triangle.recursion(i,j) % >
```

```

<%
    out.print("&nbsp;");
}
out.println("<br>");
}
%>
</div>
</body>
</HTML>

```

页面效果如图 5-2 所示。

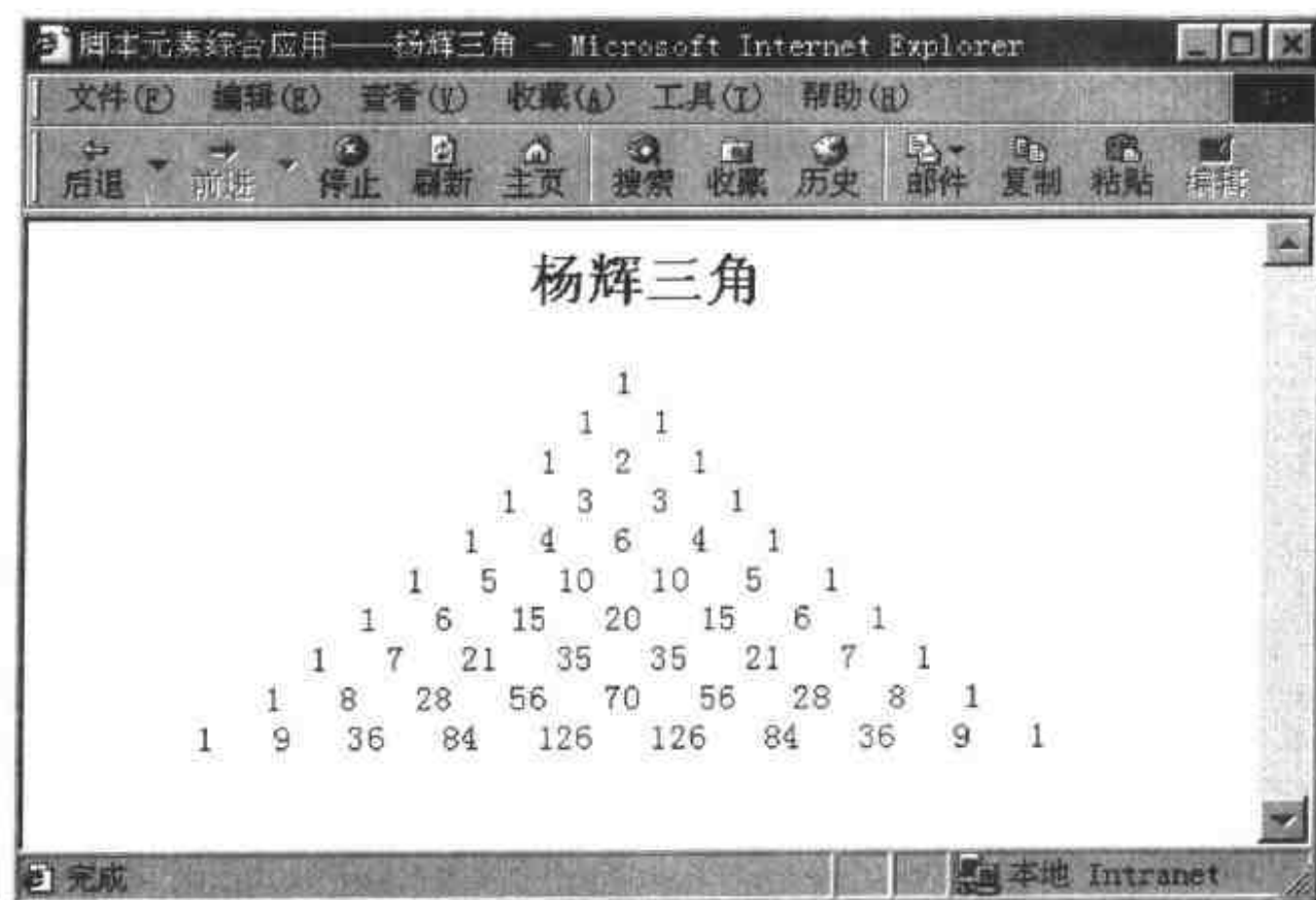


图 5-2 页面中的杨辉三角

一个配对 HTML 标记 `<div align = center>` 和 `</div>` 将杨辉三角的外观轻松搞定，而在第 4 章中几乎相同的代码，它还只是一个直角三角形。HTML 真的很神奇。

## 5.1.2 注释

### 1. Output Comment——在页面源代码中显示的注释

#### 1) 语法

```
<! -- comments... -- >
```

```
<! -- comments< % = expression % > more comments... -- >
```

#### 2) 语义

继承自 HTML、XML 的注释。产生的注释，出现在发出请求的客户端的响应输出流中。这些注释被 JSP 容器当作无解释的 template text 直接输出到客户端。



在注释中产生动态信息,动态信息由表达式给出,可以为任何内容,只要表达式合法。并且在这个注释中,表达式可有多个。

例:

```
<! -- The comment will appear in the source code of page.  
In fact, it is same as comment of HTML -- >
```

例:

```
<! -- This page was called at < % = (new java.util.Date()).toLocaleString() % > -- >  
执行它,在页面上没有显示,但是在页面源代码中有如下内容:  
<! -- This page was called at 2000-9-11 3:58:39 -- >
```

## 2. Hidden Comment——对 JSP 文档的注释

### 1) 语法

```
< % -- anything but a closing... -- % >  
< % / * * ..... * * / % >  
< % //anything but must be a line% >
```

### 2) 语义

< % -- ... % > 完全被 JSP 编译器忽略。可用来注释多行内容。注意 JSP 注释不能嵌套。

JSP 中一种替代(1)的注释方法,使用的是脚本语言的注释机制,也可注释多行。

继承 Java 中的一种注释方法,注释一行。

例:

下面是一段最常见的注释,程序代码如下:

```
< % String Author = "onlyyou"; % >  
< % String Version = "1.0"; % >  
< % String Info = "this is a share program!"; % >  
< % -- begin the comments -- % >  
<! -- Author: < % = Author % > -- >  
<! -- Version: < % = Version % > -- >  
<! -- Info: < % = Info % > -- >  
< % / * * Next comment is common and not changed. * * / % >  
<! -- Copyright, All rights reserved! -- >
```

上面程序主要展示四种注释,内容比较简单,还可以添加其它信息,如最近修改时间、联系方式等等。这样便可以构成一个实用的注释。

## 5.1.3 指令

指令提供与 JSP 页面接收到的任何特定请求无关的、有效的、全域信息给翻译步骤,不产生任何输出到当前 out 流。JSP 中有三种指令:page 指令、taglib 指令和 include 指令,它们有相同的语法形式:< % @directive {attr = "value"} \* % >。

下面将详细讲述 page 指令和 include 指令,而 taglib 指令留待第 13 章《JSP 对 XML 的集成》中讲述。

## 1. page 指令

page 指令定义了大量与页面相关的属性并将这些属性值传递给 JSP 容器。一个翻译单元(JSP 源文件和任意能用 include 指令包含的源文件称为翻译单元)可包含多个 page 指令的实例,所有的属性可应用于整个翻译单元。在一个给出的翻译单元内这个指令定义的任何“属性/取值”对只能出现一次,唯一的例外是“import”属性,这个属性可累积多次使用。其它“属性/取值”对的多次使用将导致致命的翻译错误。

未被 JSP 规范定义的属性或取值也将引发致命的翻译错误。

### 1) 语法

```
<%@page
[language="java"]
[extends="package.class"]
[import="{package.class | package.*},..."]
[session="true | false"]
[buffer="none | 8kb | sizekb"]
[autoFlush="true | false"]
[isThreadSafe="true | false"]
[info="text"]
[errorPage="relativeURL"]
[contentType="mimeType[;charset=characterSet]"| "text/html; charset=ISO-8859-1"]
[isErrorPage="true | false"]
%>
```

### 2) 属性及其属性值

#### (1) language

定义翻译单元内的脚本片断、表达式、声明使用的脚本语言。

现在,JSP 规范中,唯一定义的、必须实现的脚本语言的属性值是 Java。也就是说到目前为止,只定义了 language 属性是 Java 的语义。因此用在翻译单元之中的 Java 语言程序源码片段必须遵循 Java 程序语言规范。

JSP 规范规定可以使用其它的脚本语言实现 JSP,但所使用的脚本语言必须提供一些内部对象,必须支持 JRE(Java 运行时环境),必须可以在它的语法中嵌入 Java 组件模型(JavaBeans)。

脚本元素如表达式、脚本片段出现后,再使用 page 指令的 language 属性,将引起致命的翻译错误。因此,或者不使用该属性,默认值也只可能是“Java”;或者尽量靠前使用该属性,例如文档的第一行。

例:

```
<% -- At the beginnign of this file -- %>
<%@page language="java"%>
```

```
<% -- All the others thing is under the page language directive -- %>
```

### (2) extends

其值必须是一个完整的 Java 语言类名(包括包名和类名),这个类是 JSP 页面翻译成的那个类的超类。超类必须是实现 `javax.servlet.jsp.HttpJspPage` 接口的类。如果不设置这个属性,JSP 引擎会假定 JSP 页面继承某个实现 `HttpJspPage` 接口的类。具体继承哪个类,由 JSP 引擎的实现者决定。例如 JSWDK 实现 `HttpJspPage` 接口的类叫“`com.sun.jsp.runtime.HttpJspBase`”。Tomcat 实现 `HttpJspPage` 接口的类叫“`org.apache.jasper.runtime.HttpJspBase`”。其实在翻译出错时显示的错误信息页面中就可以找到它的踪迹,随便改个错误出来,仔细看看 tomcat 的错误信息页面中,一大堆以“at”开头的句子中的一个,即可觅得芳踪。

因为这个属性值限定 JSP Container 的能力以产生特定的超类。通过它,可以提高服务的质量,当然也能引起不可预料的后果,所以未经仔细考虑最好不要使用这个属性。

### (3) import

import 属性导入对脚本环境有用的类,其值与 Java 程序设计语言 import 声明中的一样。例如一个完整的 Java 类名列表或是一个包名后跟“\*”,说明调用包中的所有公有类。这个导入列表将在 JSP 页面执行翻译时导入,以使脚本环境有效。

在 JSP 中默认的导入列表是:

```
java.lang.*
javax.servlet.*
javax.servlet.jsp.*
java.servlet.http.*
```

这与将在第6章讲述的 Servlet 是不同的。Servlet 默认导入包仅有 `java.lang.*`,通常还需要导入 `javax.servlet.*` 和 `javax.servlet.http.*`。



注意:在 Java 语言中使用分号“;”作为导入清单的分隔字符。

例如:

```
import java.awt.*;
import java.sql.*;
```

而在 JSP 中则是使用逗号“,”作为导入清单的分隔字符。因此上面的例子在 JSP 中应有如下格式:

```
<%@page import = "java.awt.* , java.sql.*"%>
或使用两句导入:
<%@page import = "java.awt.*"%>
<%@page import = "java.sql.*"%>
```



注意:这个属性当前只定义了 language 属性值是 Java 时的情形。



注意:如果您使用的服务端是 JSWDK 1.0EA 的话,您会发觉 import 根本不管用,这是 1.0EA 的一个 bug。语法 import 没错,但是 JSWDK 的研发小组误置为 imports。因此,如果您真的使用这个版本,千万不要感到奇怪。

#### (4) session

指出页面是否需要加入一个(http 协议的)session 对客户端进行会话管理。

如果为“true”, javax.servlet.http.HttpSession 类的内部对象 session 引用页面的当前 session 或引用一个新的 session。

如果为“false”, 页面不使用 session 会话管理, 内部对象 session 无效, 为 null。这个 JSP 页面中任何对 session 的引用都是违法的, 将导致致命的翻译错误。同时也会导致页面中的 JavaBeans 的 scope 属性值只能是“page”, 若是指定 page 以外的值, 也会导致致命的翻译错误。

session 属性的默认值是“true”。

session 是非常重要的一个内部对象, 我们将在第 6 章 Servlet 和第 7 章内部对象中详细讲述它的使用。

#### (5) buffer

为初始化内部对象 out(JspWriter 的实例)配置缓冲类型。out 对象用来容纳页面输出。

如果为“none”, 输出将不使用缓冲, 所有输出将穿透 ServletResponse.PrintWriter(使用 response.getWriter()方法获得)直接写到客户端。

如果配置了缓冲大小, 那么输出将被一个不小于配置大小的缓冲空间缓存起来。即输出使用的实际缓存空间至少应为配置大小。其大小只能是 1Kbyte 的整数倍, 即后缀 KB 是强制的。

缓冲空间的内容是自动溢出(不引发异常), 还是溢出引发一个异常将取决于 autoFlush 属性的值。autoFlush 请参见下一个属性。

默认情况下, 输出将被一个不小于 8KB 的缓冲器缓存。

#### (6) autoFlush

指定当缓冲区被填满时, 缓冲输出是自动溢出(autoFlush = “true”)还是引发一个指示缓冲溢出的异常(autoFlush = “false”)。

默认值是“真”。



注意: 当 buffer = none, 设置 autoFlush = false 是违法的, 将导致一个致命编译错误。因为没有缓存时, 页面的 JspWriter 本身的行为就是自动溢出, 否则, 输出要放在何处才不会丢失。

#### 例 5.3

buffer 属性讨论

程序清单如下:(注意: 书写格式不一样, 结果也不一样。)

```
<%@page autoFlush="false" buffer="1" %>
<HTML>
<HEAD><TITLE>Buffer 溢出试验</TITLE></HEAD>
<BODY>
<%try{ %>
<%int firstloop=10; %>
```

```
<%int secondloop=10;%>
<%for(int i=0;i<firstloop;i++)|%>
<%for(int j=0;j<secondloop;j++)|%>
<%out.print("@@");%>
<%}%>
<%out.print("<br>");%>
<%}%>
<%out.print("*****");%>
<%out.print("*****");%>
<%|catch(Exception e)|%>
<%out.print("exception"+e);%>
<%|%>
</BODY>
</HTML>
```

程序非常容易看懂,两次循环共打印 200 个“@”,然后再打印 19 个“\*”。页面效果如图 5-3 所示。

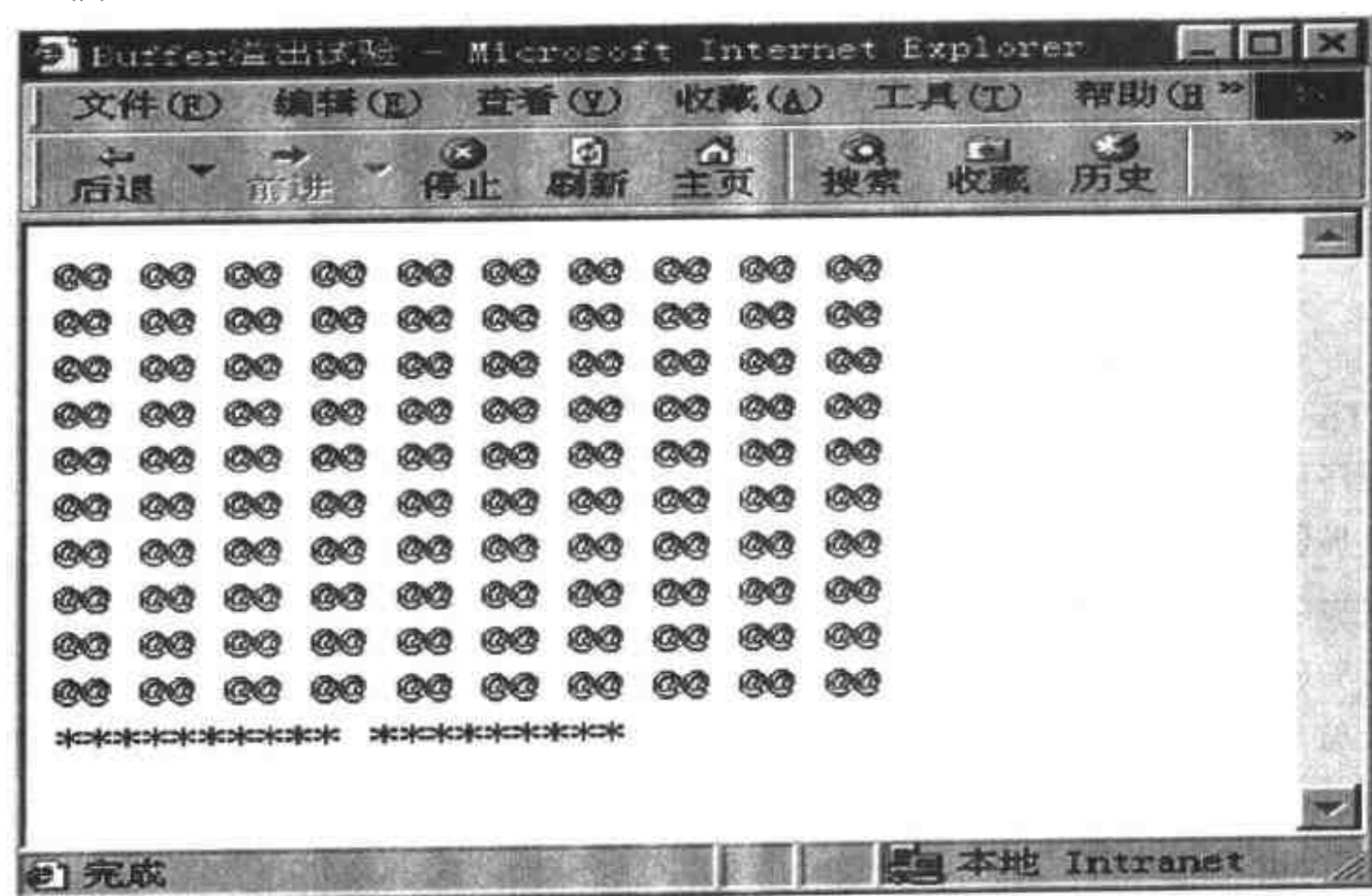


图 5-3 Buffer 溢出试验

下面我们分情况讨论一下:

autoFlush="false"buffer="1"

程序即为此处给出代码。buffer="1"是有缓冲区情况下的最小值。例子给出的情况是不会把缓冲区填满的,看看页面效果如图 5-3 所示是怎样的,看看页面源代码,如果有兴趣还可统计一下字符数(包括空格),再改变其中显示的内容,比如添加一行<%out.print("\*\*\*\*\*");%>,并执行,此时就溢出了,发生致命编译错误

——JSP Buffer overflow——JSP 缓冲区溢出。有两个结论,一是 Java 使用 unicode 字符集,是 ASCII 码两倍长。二是缓冲区大小确实不小于指定值。

AutoFlush = "false" buffer = "none"

此时发生致命编译错误——Illegal combination of buffer = "none" && autoFlush = "false"——buffer 和 autoFlush 非法组合。

AutoFlush = "true" buffer = "any legal value", buffer 为任意合法值。

此时,无论怎样改变循环体的大小,都不会发生前两种编译错误。此时还有一个比较,将循环体改得相当大,并且 buffer 一次取较大值,比如 32k,一次取较小值,比如 2k。比较二者的立即响应速度和响应持续时间。在负载相当大的时候,二者是有差异的。缓冲区较小,立即响应速度很快,但响应持续时间长。反之则是缓冲区较大的特点。这就是矛盾,解决方法还是那句老话,“具体情况,具体分析”。

#### (7) isThreadSafe

配置运行在页面内的线程的安全特性。

如果为“假”,JSP 容器将接收同时发生的对该页面的多个突发请求,并按它们的接收顺序选择其中一个进行处理。实际上是指示 JSP 页面要实现 javax.servlet.SingleThreadModel 接口。

如果为“真”,JSP 容器将选择接收同时发生的对该页面的多个突发客户端请求。

页面作者使用“真”时,必须确保它们进入临界区同步。

默认值为“真”。



注意:即使 isThreadSafe 属性为“false”,JSP 页面作者也必须确保对 ServletContext 或 HttpSession 内的任何共享对象的访问都能被正确同步。

#### 例 5-4

isThreadSafe 属性在计数器中的应用。

程序清单如下:

```
<%@page language="java"%>
<%@page import="java.io.*"%>
<%@page isThreadSafe="false"%>
<%
```

```
String temp;
```

```
int visitedNum = 0;
```

```
try{
```

```
    //read the value of last time
```

```
    RandomAccessFile raf = new
```

```
RandomAccessFile("../webapps/examples/book/chapter05/sample5_1_3.txt","rw");
```

```
    //synchronized reading and writing
```

```
    synchronized(raf){
```

```
        temp = raf.readLine();
```

```

        visitedNum = Integer.parseInt(temp);
        temp = Integer.toString(++visitedNum);
        raf.seek(0);
        raf.writeBytes(temp);
        raf.close();
    }
}
catch(FileNotFoundException e){
    out.println("file not found" + e);
}
catch(IOException e){
    out.println("io error" + e);
}
%>

<HTML>
<HEAD>
<TITLE>isThreadSafe 属性应用于简单的计数器</TITLE>
</HEAD>
<body>
<center>
<h1>isThreadSafe 属性应用于简单的计数器</h1>
<p><font size = 5 color = blue>
This page has been visited
<font size = 5 color = red>
< % = visitedNum % >
</font>
times
</font>
</p>
</center>
</body>
</HTML>

```

页面效果如图 5-4 所示。

该计数器每刷新一次页面,就要读一次文件,取得上一次的计数值,然后在其上加 1,最后将新的数值写回文件。这个计数器涉及到读写文件,在实际应用中,当多个用户同时请求该页面时,就会发生读写文件冲突,即该文件是共享资源。此时需要同步该文件的读写,因此在本程序第 3 行,使 page 指令的 isThreadSafe 属性值为“false”,指示页面实现单线程,即实现 javax.servlet.SingleThreadModel 接口。



图 5-4 isThreadSafe 属性的应用

这种方式的计数器几乎没有任何实用价值,为什么还要讲它。因为一是这个例子讲线程同步和单线程非常典型;二是这个是其它计数器的基础,不管怎样,计数器总得把计数值保留下来。

#### (8) Info

定义任意的一个字符串,该字符串与页面一起被翻译,并且稍后能通过页面的实现(类)的方法 `Servlet.getServletInfo` 取得。

#### (9) sErrroPage

指出当前的 JSP 页面是否作为另一个 JSP 页面的 `errorpage` 属性的 URL 值,即当前页面是否接受来自另一个页面的 `Throwable` 类错误并处理它。为讲述方便,称当前 JSP 页面为“目的页面”或“错误处理页面”,称另一个页面为“源页面”或“出错页面”。

如果为“真”,那么目的页面的内部变量“`exception`”将被定义,并且它的值是对源页面 `throwable` 的引用。

如果为“假”,那么目的页面的内部变量“`exception`”是无效的,任何在该 JSP 页面内对它的引用都是违法的,并将导致一个致命的翻译错误。

`sErrroPage` 属性的默认值为“false”。

#### (10) errorPage

定义一个 URL,使抛出的任何 Java 程序设计语言的 `throwable` 对象不被页面的实现捕获,而是导向 URL 指向资源,由该资源进行错误处理。这个资源,即目的页面必须是一个 JSP 页面。

如果指定了目的页面,调用目的页面的内部变量 `exception` 也包含引用源页面不可捕获的 `Throwable` 类异常。

默认 URL 是与实现相关的。





注意:由抛出页面实现使用 `setAttribute()` 方法把 `Throwable` 对象的引用存储在公共的 `ServletRequest` 对象中将其传递给错误页面实现,这种类型的 `Throwable` 对象名为 `javax.servlet.jsp.jspException`。



注意:如果 `autoFlush=true`,又如果初始化 `JspWriter` 的内容已经被刷新到 `ServletResponse` 输出流中,那么任何后来试图发送从出错页面来的不可捕捉的异常给 `errorpage` 的企图都将失败。

当一个错误处理页面在 `web.xml` 描述中也被指定时,那么 JSP 错误处理页面被先执行,然后才执行 `web.xml` 中定义的错误处理页面。

### 例 5-5

`isErrorPage`, `errorPage` 属性的应用

我们将 `sample5_1_3.jsp` 的代码重列如下,在此另存名为 `sample5_1_5.jsp`。这个程序中,我们改变其循环体大小,它就可能发生缓冲溢出异常,在 `sample5_1_3.jsp` 中,我们用 `try(catch)` 来捕获异常,但效果不好,它根本不能捕获到该异常,因为该异常其实是一个致命的翻译错误, `try(catch)` 根本没有执行的机会,谈什么捕获异常。在学了 `errorPage`, `isErrorPage` 这两个页面指令后,我们来看看它们的本领。为此编写错误处理页面 `sample5_1_5a.jsp`。

`sample5_1_5.jsp` 程序清单如下:

```
<%@page autoFlush="false" buffer="1" errorPage="sample5_1_5a.jsp"%>
<HTML>
<HEAD>
<TITLE>errorPage、isErrorPage 属性应用</TITLE>
</HEAD>
<body>
<h3>errorPage、isErrorPage 属性应用</h3>
<%
    int firstloop=30;
    int secondloop=30;
    for(int i=0;i<firstloop;i++){
        for(int j=0;j<secondloop;j++){
            out.print("@@");
        }
        out.print("<br>");
    }
    out.print("*****");
    out.print("*****");
%>
</BODY>
</HTML>
```

sample5\_1\_5a.jsp 程序清单如下:

```
<HTML>
<HEAD>
<TITLE>错误处理页面</TITLE>
</HEAD>
<body>
<h1>错误处理页面</h1>
<%@page isErrorPage="true"%>
<h2>Attention the felloing error occurs</h2>
<font size=4 color=red>
<%=exception.getMessage()%>
</font>
</body>
</HTML>
```

执行 sample5\_1\_5.jsp, 异常发生, 并被导向 sample5\_1\_5a.jsp 这个错误处理页面, 错误处理页面的内部对象 exception 引用 sample5\_1\_5.jsp 抛出的 Throwable 对象。错误处理页面效果如图 5-5 所示。可以看出, errorPage 指令功能比 try(catch) 强大得多, 当源页面不能执行, 或对其异常做特殊处理, 这个属性是非常有用的, 并且使用起来也不比 try(catch) 麻烦。

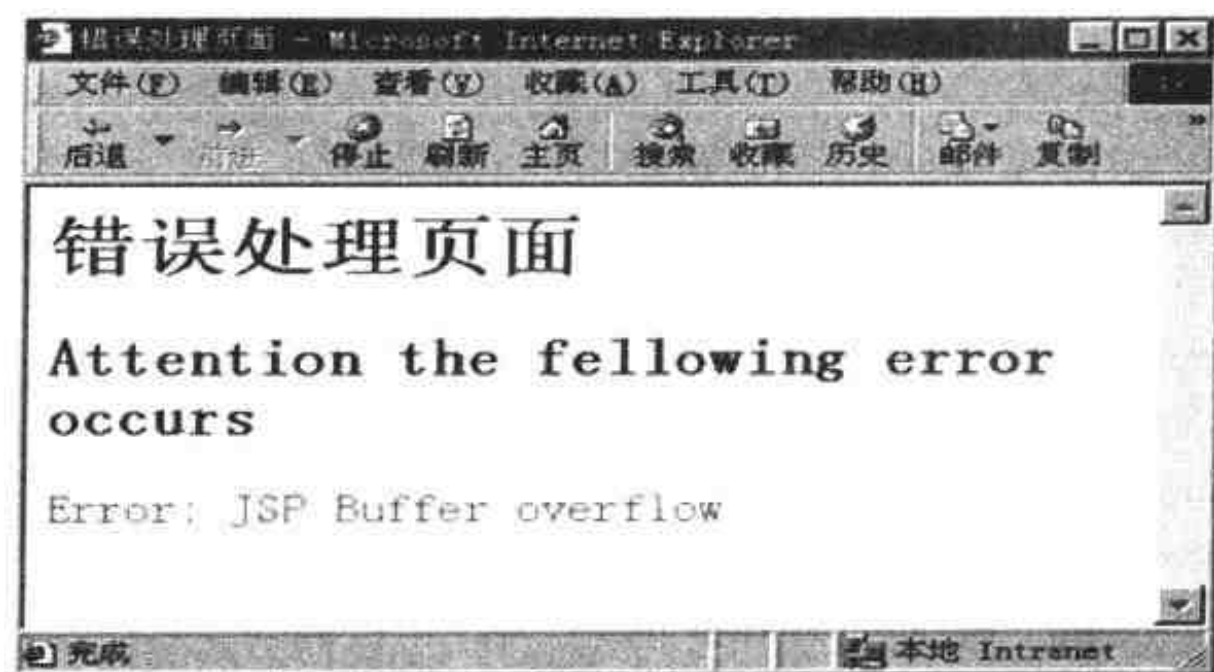


图 5-5 errorPage 和 isErrorPage 属性应用

异常发生后, 容器将异常导向目的页面, 所以源页面的异常处理根本不能获得执行权, 即实际上被这个页面给屏蔽掉了。正是这个原因, 我们在 sample5\_1\_5.jsp 的代码中去掉了原本在 sample5\_1\_3.jsp 中的 try(catch)。这也是它的缺点, 不够灵活和高效, 没办法对某个错误做特殊处理; 并且可能需要捕获所有的异常以满足各页面的需求; 另外重写 URL 也是很耗时的。

sample5\_1\_5 的变例

事实上, sample5\_1\_5.jsp 与 sample5\_1\_3.jsp 几乎一样。sample5\_1\_3.jsp 中只要再添加一个字符, 就会导致缓冲区溢出错误。然而在 5\_1\_5.jsp 中, 即使将两个循

环体都改为 18(存档为 sample5\_1\_51.jsp),也不会发生缓冲区溢出错误,页面效果如图 5-6 所示,面二者仅书写格式不一样,请比较一下,二者书写格式差异有多大。事实上,正是这点差异导致了效果的不同(请比较一下图 5-3 和图 5-6)。比较一下源代码发现 sample5\_1\_3.jsp 有很多空格,而 sample\_1\_51.jsp 则几乎没有空格。这就是说,二者代码生成是不一样的。如果要求页面布局非常精确,那么一定要注意这种差异。

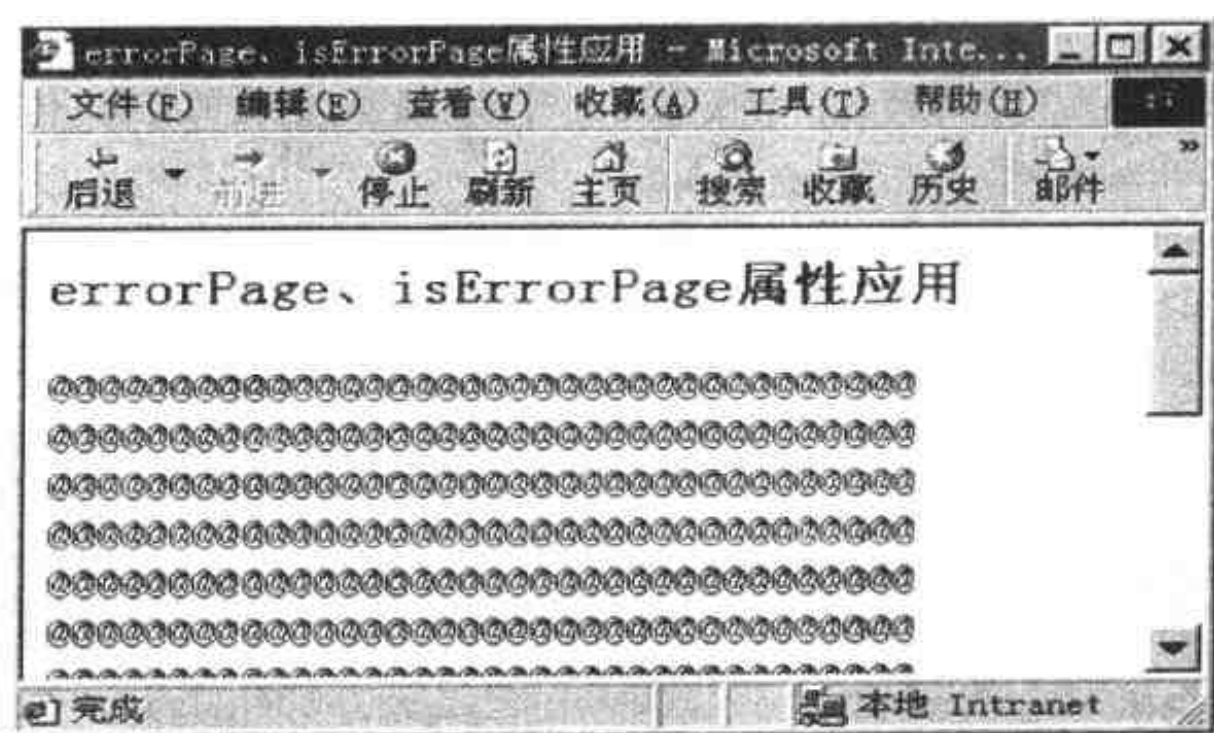


图 5-6 变例

(11) contentType

定义 JSP 页面、JSP 页面响应(response)的字符编码类型,和定义 JSP 页面 response 的 MIME 类型。值的形式为如下两种之一:

Type

Type; charset = CHARSET

如果出现 CHARSET,则字符编码类型必须为 IANA 值。TYPE 是一个 MIME 类型。默认“TYPE”类型为“text/html”;默认字符编码类型值为 ISO - 8859 - 1。

(12) pageEncoding

定义 JSP 页面的字符编码类型。

值的形式是“CHARSET”——一种字符编码类型——必须是 IANA 值。

默认值是 contentType 属性的 CHARSET。如果使用这个属性,其值为 ISO-8859-1 或其它。

## 2.Taglib Directive——标记库指令

为分类完整,仅在此列出。完整论述参见对 XML 的集成。

### 3. The include Directive——包含指令

include 指令用于 JSP 页面翻译时插入(替换)文本和(或)代码。<%@include file="relative URLspec"%>指令将指定资源的文本插入 JSP 文件。包含文件能被 JSP 容器感知,并受其访问管理支配。

如果包含文件正被改变,JSP 容器能够感知这种改变,这时 JSP 容器可以重新编译

JSP 页面。然而,JSP 规范中并没有一种方法指出包含文件曾经改变过。

1) 语法

```
<%@include file="relativeURLspec"%>
```

例:将版权信息文件如 copyright.jsp 包含进来。

```
<%@include file="copyright.jsp"%>
```

例 5-6

我们来看一个更复杂的例子。请注意包含资源的类型,包含的位置,包含的时间。

主 JSP 文件 sample5 \_ 1 \_ 6.jsp 如下:

```
<%@ page language="java" %>
<HTML>
<HEAD><TITLE>Include 指令试验</TITLE></HEAD>
<BODY>
<center>
<h1>Include Directive Demo</h1>
<%@ include file="sample5 _ 1 _ 6.txt" %>
</center>
</BODY>
</HTML>
```

包含文件 sample5 \_ 1 \_ 6.txt 如下:

```
sample5 _ 1 _ 6.txt start here!
<% for(int i=1;i<5;i++) { %>
<% out.println("<h"+i+">"); %>
<% out.println("number"); %>
<% out.println(i); %>
<% out.println("</h"+i+">"); %>
<% } %>
sample5 _ 1 _ 6.txt end here!
```

页面效果如图 5-7 所示。

下面把 sample5 \_ 1 \_ 6.txt 改一下,比如循环从“5”改到“7”,执行 sample5 \_ 1 \_ 6.jsp 看一下,页面有变化吗? 再把 sample5 \_ 1 \_ 6.jsp 改一下,比如将“<h1></h1>”改为“<h2></h2>”,再一次执行 sample5 \_ 1 \_ 6.jsp,页面有变化吗? 是不是两次改变都在这次生效了,这是为什么呢?

为叙述的方便,我们称 sample5 \_ 1 \_ 6.jsp 为包含文件,它包含其它的资源;称 sample5 \_ 1 \_ 6.txt 为被包含文件。上例的 sample5 \_ 1 \_ 6.txt 不是以 jsp 结尾,但是其内部的 JSP 语法一样有用,这是为什么呢?

这是因为 include 指令是翻译时期的指令,也就是 JSP 容器先把资源插入 JSP 文件中,然后再翻译生成 Servlet。我们可以在服务器的工作目录下找到生成的 Servlet 类。因

此所有使用 include 指令插入的资源,其内容必须符合 JSP 语法。

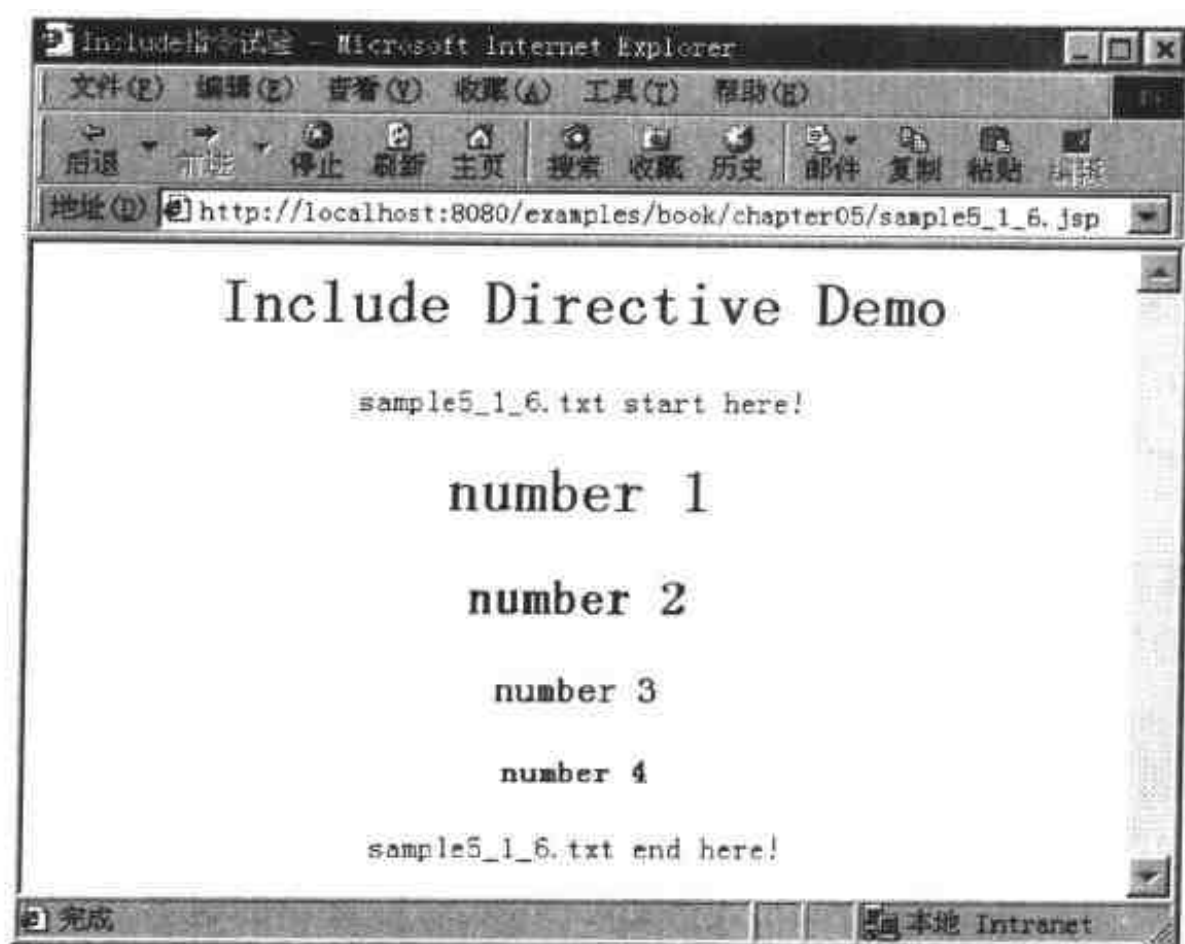


图 5-7 包含指令试验

因为 include 指令是翻译时期的指令,改变被包含文件的内容对包含文件根本没有影响。即 JSP 容器不能感知被包含文件的变化,它只能感知到包含文件的改变。所以才出现了上文叙述的情况。因此,欲使改变过的被包含文件生效,方法是把包含文件也改变一下,让 JSP 容器重新翻译一次。或者不使用 include 指令,而使用下文叙述的 include 行为。



注意:如果被包含文件中有 HTML 标记,那么在 HTML 文档中只能出现一次的标记 `<HTML>`、`</HTML>`、`<HEAD>`、`</HEAD>`、`<TITLE>`、`</TITLE>`、`<BODY>`、`</BODY>` 同时出现在包含文件中和被包含文件中可能导致错误。如果在包含文件中出现过页面指令的 `contentType` 属性,又在被包含文件中出现,将导致错误。

### 5.1.4 行为

行为的解释通常是依赖于 JSP 页面接到的特定请求细节。行为为请求处理阶段提供信息。行为可以分为两类:

- ▶ 自定义行为,通过标记扩展机制获得。详见 JSP 对 XML 的继承。
- ▶ 标准行为,由规范定义。包括 `<jsp:forward>`、`<jsp:include>`、`<jsp:plugin>`、`<jsp:getProperty>`、`<jsp:setProperty>`、`<jsp:useBean>`。本节只讲述 `<jsp:include>` 和 `<jsp:forward>` 行为。

## 1. <jsp:include>

### 1) 语法

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

或者:

```
<jsp:include page="URLspec" flush="true">
```

```
    |<jsp:param.../>{*
```

```
</jsp:include>
```

### 2) 语义

第一种语法形式仅用于请求时包含。第二种语法形式, <jsp:param.../> 为被包含文件提供额外的参数。

“page/URLspec”中 URLspec(相对全球资源定位规范)为一个请求时属性值。

flush: 可选布尔属性, 如果为“true”, 缓存立即刷新, 即将缓存内容写入 out, 并清空自身。默认值为“false”。

<jsp:include> 提供在当前页面上下文对静态、动态资源的包含。include 行为与 include 指令的最大不同在于, include 行为是请求时期的语法, 可随着插入资源的改变而改变, 即每次都要重新读取包含资源, 然后解释执行; 而 include 指令将资源读入, 然后翻译成一个执行类放在服务端 work 目录下, 只要包含文件不改变, 无论被包含文件怎样改变, 服务端 JSP 容器(Servlet 容器)都不会检测到这种变化, 当然也不会对其重新编译。只有包含文件改变了, 服务端才会在重新编译时, 又一次读取被包含资源。这时, 被包含资源的变化(如果有变化的话)才能体现出来。二者还有一个区别, include 指令包含的资源无论是什么类型(不管是什么后缀名), 都一视同仁, 即只要其中有 JSP 语法, 都会被解释执行。然而, include 行为是典型的“势利眼”, 对后缀名为 jsp 的资源, 当有对它的请求时, 才处理它, 然后将结果输出; 而对后缀名非 jsp 的资源, 原样输出, 由浏览器解释。

让我们验证一下:

#### 例 5-7

<jsp:include> 行为讨论。

将 sample5 \_ 1 \_ 6 的程序做如下改写: 将 <%@ include file="sample5 \_ 1 \_ 6.txt" %> 改为 <jsp:include page="sample5 \_ 1 \_ 7a.jsp" flush="true"/>。

sample5 \_ 1 \_ 7 程序清单如下:

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

```
<HTML>
```

```
<HEAD>
```

```
<TITLE> Include 行为试验</TITLE>
```

```
</HEAD>
```

```
<BODY>
```

```
<center>
```

```
<h1> Include 行为试验</h1>
```

```
<jsp:include page="sample5 _ 1 _ 7a.jsp" flush="true"/>
```

```
</center>  
</BODY>  
</HTML>
```

支持程序 sample5 \_ 1 \_ 7a.jsp 由 sample5 \_ 1 \_ 6.txt 改变而来。

sample5 \_ 1 \_ 7a.jsp 程序清单如下：

```
sample5 _ 1 _ 7a.<font size = 5 color = red>jsp</font> start here!
```

```
<% for(int i = 1;i < 5;i++) { %>
```

```
<% out.print("<h" + i + ">"); %>
```

```
<% out.print("number"); %>
```

```
<% out.print(i); %>
```

```
<% out.println("</h" + i + ">"); %>
```

```
<% } %>
```

```
sample5 _ 1 _ 7a.<font size = 5 color = red>jsp</font> end here!
```

页面效果如图 5-8 所示。

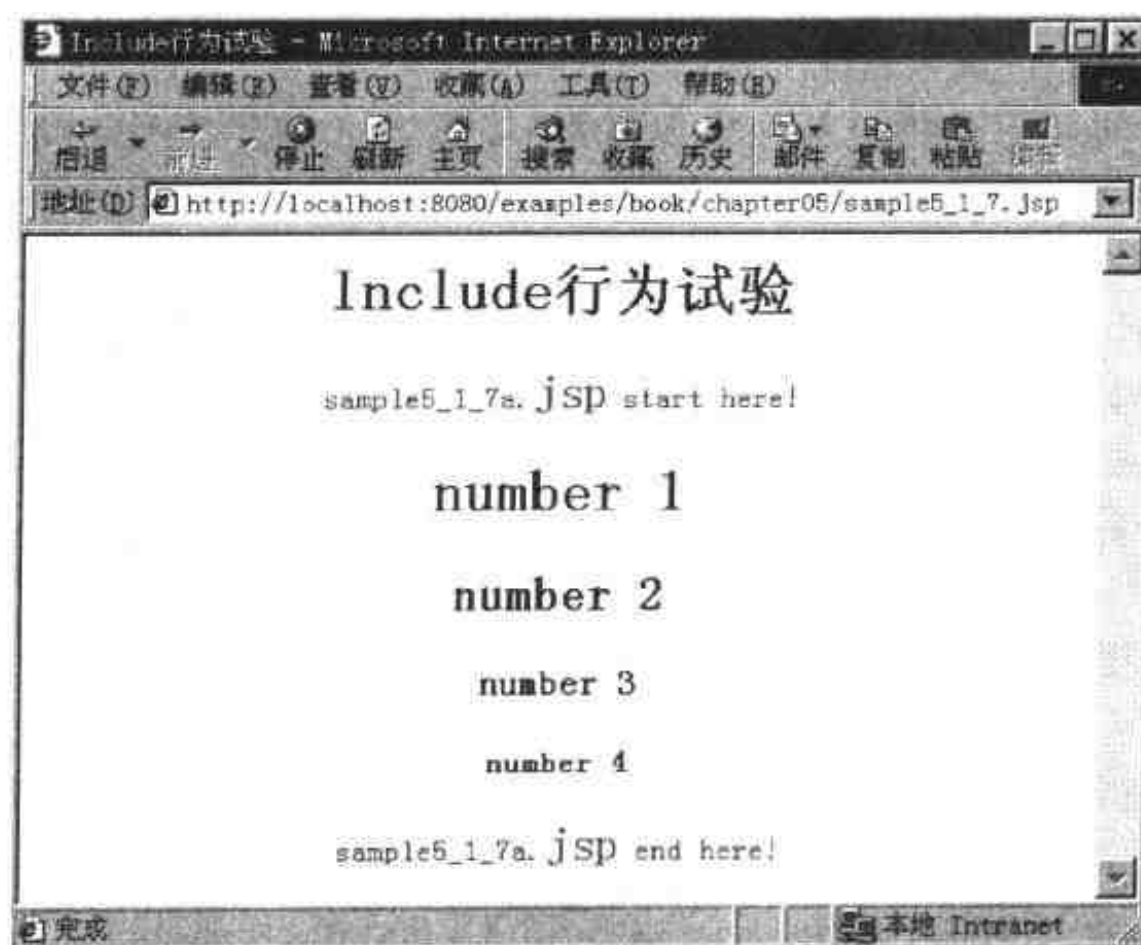


图 5-8 include 行为包含 JSP 文件

比较一下 sample5 \_ 1 \_ 7.jsp 和 sample5 \_ 1 \_ 6.jsp 的显示效果,没有多大差别。但是别忘了各自包含的资源类型不一样。

将 sample5 \_ 1 \_ 7.jsp 的支持程序 sample5 \_ 1 \_ 7a.jsp 的某个地方改变一下,例如改变循环体的大小。重新执行,页面是不是立刻有了变化。同样这儿也存在一个矛盾,include 行为灵活但费资源,include 指令省资源但不够灵活。

例 5 \_ 1 \_ 7 的变例 1(sample5 \_ 1 \_ 71.jsp 和支持程序 sample5 \_ 1 \_ 71a.html)

其实差不多,只是将包含资源换成了 html 类型文件。看看页面效果,如图 5-9 所示。





图 5-9 include 行为包含 HTML 文件

例 5\_1\_7 的变例 2(sample5\_1\_72.jsp 和支持程序 sample5\_1\_72a.txt)也是差不多的,只是将包含资源换成了 txt 类型文件。请读者自己试验一下这种非 JSP 资源的包含情况。

现在,比较一下 sample5\_1\_7.jsp、sample5\_1\_71.jsp、sample5\_1\_72.jsp 和 sample5\_1\_6.jsp 这四者,很容易得出上文所述结论。

include 行为实际上是将包含资源写进 out 的当前值。资源用相对 URL 提供。URL 由页面服务端上下文对其解释。包含文件仅仅能将包含资源写入 JspWriter 对象,而不能重置 JspWriter 头部,但是这要排除调用 setCookie()这类方法。如果这种约束不令人满意的话,一个请求时异常将发生。这种约束与调用 RequestDispatcher 类的 include()方法是等价的。RequestDispatcher 参见第 6 章第 6 节有关叙述。

<jsp: include>行为可能有<jsp: param...>子元素。<jsp: param>子元素能提供一些用于包含行为的请求参数的值。一旦包含结束,对调用 JSP 页面的请求的处理将开始。

flush 属性控制刷新,如果为“true”,又如果页面输出被缓存了,那么缓存在包含之前被刷新。

### 3) 在 JSP 页面中包含资源

在 JSP 页面中,包含资源是意义重大的任务部分。相应的,JSP 规范有两种包含机制适用于不同的任务,如表 5-1 所示。

表 5-1 JSP 的两种包含机制

语法:	类型	包含阶段	包含对象	描 述
<%@include file = ... %>	指令	翻译时	静 态	内容由 JSP 容器解析
<jsp:include page = .../>	行为	请求时	静态和动态	内容不解析,被包含在原来位置



二者的路径信息都遵循相对 URL 规范。相对 URL 由 web/application server 解析并且与 web/application server 的 URL map 有关(URL 映像有关)。

包含指令将资源如一个 JSP 页面当作静态的对象。例如 JSP 页面内的字节。

包含行为将资源如一个 JSP 页面当作动态的对象。例如被请求的对象和它的处理结果。

#### 4) 相对 URL 规范

在 JSP 中将大量使用相对 URL 规范。事实上,我们在前面已经使用过多次,例如包含指令、包含行为等。我们在多个场合提醒读者注意路径问题,例如在安装 JSWDK 时。我们将碰到较为复杂的路径问题,因此有必要在此说明。


相对 URL 规范,在 Servlet2.1 规范中被称为 URI 路径。我们先举几个例子。


- (1) "errorPage.jsp"
- (2) "/examples/book/chapter14/bbs/bbsindex.jsp"
- (3) "../bbs/bbsindex.jsp"


一个以斜杠"/"开始的路径,如例子中的(2),它被应用程序解释为:这个 JSP 页面相对于它的 ServletContext 对象提供的上下文根路径。我们称这样的路径为“上下文相对路径”。


一个不是以斜杠"/"开始的路径,如例子中的(1)和(3),它被解释成相对于当前 JSP 页面。当前页面是以“上下文相对路径”表示,所以它最终也将被 ServletContext 对象解释成“上下文相对路径”,我们称这样的路径为“页面相对路径”。

JSP 规范要求一律在运行 JSP 页面的 Web 服务器的上下文中解释这些路径。该规范贯穿整个地址映射,其语义作用于翻译时和请求时两个阶段。

 **注意:** JSP 中都是 URL 或 URI 路径,而非文件系统路径。二者最大的不同是分隔符不一样,URL 的分隔符是斜杠"/",而文件路径分隔符是反斜杠"\”。二者的作用范围不一样,URL 是全球范围可达的,后者仅限于本地。二者有区别,也有联系。在本地系统中,页面相对 URL 与文件相对路径是等价的(语义等价,形式上仍有分隔符的差异)。但是上下文相对 URL 与文件绝对路径是不等的,甚至没有任何直接联系。

 **注意:**“协议://IP(hostname): port /”,如“http://127.0.0.1:8080/”,其中根路径“/”的映射文件系统目录由服务应用程序决定,并可以更改。例如本书中的 Tomcat,安装在“C:\Sun\Tomcat”目录下,它的根路径“/”对应“C:\Sun\Tomcat\webapps\ROOT”。你可以在其配置文件 server.xml 中更改。详见第 1 章 Tomcat 的安装。

 **注意:**某个服务的 URL 路径映射到实际文件系统路径与根路径“/”映射到实际文件系统路径之间可以没有任何联系。例如你可以在服务应用程序安装目录外定义一个服务。举个例,服务 URL 路径为“/mysample”,实际路径为“D:\mysample”,而服务应用程序安装目录为“C:\Sun\Tomcat”。

 **注意:**书写的上下文相对路径始终是以某个服务 URL 路径开始的。例如本书中,

书写程序中一个可能的上下文相对路径如“/examples/book/chapter14/index.jsp”。



注意:当有多个文件使用包含行为包含同一个文件,例如包含页头,最好使用上下文相对路径,不要使用页面相对路径。



注意:如果被包含文件有超级链接,那么就要注意超级链接路径与包含文件的关系。



注意:如果还有解决不了的路径问题,请查看服务应用程序的日志。例如 Tomcat 的 logs 目录下 jasper.log 记录了每个请求的路径及其真实的文件系统路径。



注意:Tomcat 中,文件夹名不能使用中文和带有下划线的名字。

#### 例 5-8

路径问题。使用了三个文件,文件 sample5\_1\_8.jsp、文件 sample5\_1\_8a.jsp 位于文件夹 sample518a 中,文件 sample5\_1\_8b.jsp 位于文件夹 sample518b 中。

sample5\_1\_8.jsp 源代码如下:

```
<%@ page contentType="text/html; charset = gb2312" %>
<HTML>
<HEAD>
<TITLE>路径问题实验</TITLE>
</HEAD>
<body>
第一个文件<br>
<a href="sample518a/sample5_1_8a.jsp">第二个文件</a><br>
<jsp:include page="sample518a/sample5_1_8a.jsp" flush="true" /><br>
```

sample5\_1\_8a.jsp 源代码如下:

```
<%@ page contentType="text/html; charset = gb2312" %>
第二个文件<br>
<a href="/examples/book/chapter05/sample518b/sample5_1_8h.jsp">第三个文件</a><br>
<jsp:include page="/book/chapter05/sample518b/sample5_1_8b.jsp" flush="true"/>
```

sample5\_1\_8b.jsp 源代码如下:

```
<%@ page contentType="text/html; charset = gb2312" %>
第三个文件<br>
<a href="/examples/book/chapter05/sample5_1_8.jsp">第一个文件</a>
```

在使用比较多的重导(转发)行为、包含行为(指令)的情况下,超级链接最好使用绝对 URL 规范。虽然不利于移植,但可有效地防止路径错误。

## 2. <jsp:forward> (重导或转发)

### 1) 语法

<jsp:forward page="relativeURLspec"/>

或者:

<jsp:forward page="urlSpec">

  | <jsp:param.../> | \*

</jsp:forward>

这个标记允许页面作者通过指定下述属性影响当前请求的处理。

page:URL 是一个相对路径,接受一个请求时属性值(必须是一个能转换成相对路径的字符串。)

### 2) 语义

<jsp:forward page="urlSpec">元素允许运行时将当前请求分配给在同一上下文中的静态资源,JSP 页面或 Servlet 类,如同在当前页面。<jsp:forward>将有效的终止当前页的执行。urlSpec 参见前文叙述。

请求对象将根据 page 指令的属性值作相应调整。

<jsp:forward>行为可能有<jsp:param>子元素,子元素提供用于 forward 请求的参数值。

如果页面输出被缓存起来,那么 forward 之前将清除缓存内容。

如果页面输出没有被缓存起来(buffer="none"或 buffer 太小,autoFlush="true"),那么某些内容有可能已经被写出,此时试图 forward 请求将导致 IllegalStateException。

例:下述语句可在某些动态场合将请求转发给一个静态页面。

```
<% String whereTo = "/templates/" + someValue; %>
```

```
<jsp:forward page = '<% = whereTo %>'
```

其中 page = '<% = whereTo %>' 这种语法,JSP 规范书称之为 Request-time 属性值,即属性值在一次请求/响应期间,才决定它的值。

## 3. <jsp:param>

### 1) 语法

<jsp:param name="name" value="value"/>

这个行为有两个必需的属性:name 和 value。name 指定参数名字,value,可由一个请求时的表达式指定它的值。

### 2) 语义

<jsp:param>元素用来提供“属性/值”,(key/value)信息。这个元素可用在<jsp:include>,<jsp:forward>和<jsp:plugin>这三个元素中。

执行<jsp:include>或<jsp:forward>时,包含页面或转发页面首先参考原始请求对象,然后在原始参数上增加<jsp:param>引入的新参数,如果存在相同的参数,优先使用新值。新参数的作用域是<jsp:include>或<jsp:forward>的调用页。例如,<jsp:include>的新参数在包含行为发生之后就不再有效。这与 ServletRequest 的 include 和 for-

ward 方法有相同的行为。

例如,某个请求有参数  $A = \text{foo}$ ,转发又指定了参数  $A = \text{bar}$ ,那么转发请求将有  $A = \text{bar}, \text{foo}$ 。注意新值优先级更高,位置在前。

#### 例 5-9

将 `<jsp:forward />` 和 `<jsp:param />` 结合起来举一个例。例子是一个“烦琐”的注册器。这里只列出后台处理文件 `sample5_1_9.jsp` 和支持文件 `register.jsp` 的源代码。

sample5\_1\_9 源代码如下:

```
<%@ page contentType="text/html; charset = gb2312" %>
```

```
<%! String sforwardto=""; %>
```

```
<%
```

```
String sname = request.getParameter("txtName");
```

```
String spassword = request.getParameter("txtPassword");
```

```
String spassagain = request.getParameter("txtPassagain");
```

```
//清除 sforwardto 的值,避免刷新错误。
```

```
sforwardto="";
```

```
if(sname == null || sname.equals("")){
```

```
    sforwardto="errornoname.jsp";
```

```
}
```

```
else{
```

```
    if(spassword == null || spassword.equals("")){
```

```
        sforwardto="errornopassword.jsp";
```

```
    }
```

```
    else{
```

```
        if(spassagain == null || spassagain.equals("") || ! spassagain.equals(spassword))
```

```
{
```

```
            sforwardto="errorpassagain.jsp";
```

```
        }
```

```
    }
```

```
}
```

```
if(! sforwardto.equals(""))
```

```
{
```

```
    sforwardto="sample519/" + sforwardto;
```

```
%>
```

```
<jsp:forward page='<% = sforwardto %>' />
```

```
<%//注意这儿的书写格式
```

```
    }else{
```

```
        sname = new String(sname.getBytes("ISO-8859-1"));
```

```
        spassword = new String(spassword.getBytes("ISO-8859-1"));
```

```
        spassagain = new String(spassagain.getBytes("ISO-8859-1"));
```

```
//同样,避免刷新错误。
sforwardto="";
sforwardto="sample519/register.jsp";

%>

<jsp:forward page='<% = sforwardto%>'>
    <jsp:param name="Name" value='<% = sname%>' />
    <jsp:param name="Password" value='<% = spassword%>' />
</jsp:forward>

<%
}

%>
```

支持文件 register.jsp 的源代码如下：

< %@ page contentType="text/html; charset = gb2312" % >

< %

//这儿是服务器转发的请求,但是与从客户端请求获得信息的方式没有差别。

```
String sname = request.getParameter("Name");
```

```
String spassword = request.getParameter("Password");
```

% &gt;

<HTML>

&lt;HEAD&gt;&lt;TITLE&gt;&lt;/TITLE&gt;

&lt;/HEAD&gt;

&lt; body &gt;

[illegible]

---

<div align= center>

**<font size=5 color=red>这儿是处理结果**

</font>

<font size=4 color=blue>

< %

```
out.println("Welcome you " + sname + "!");
```

% >

</font>

&lt;/body&gt;

</HTML>

处理结果页面效果如图 5-10 所示：

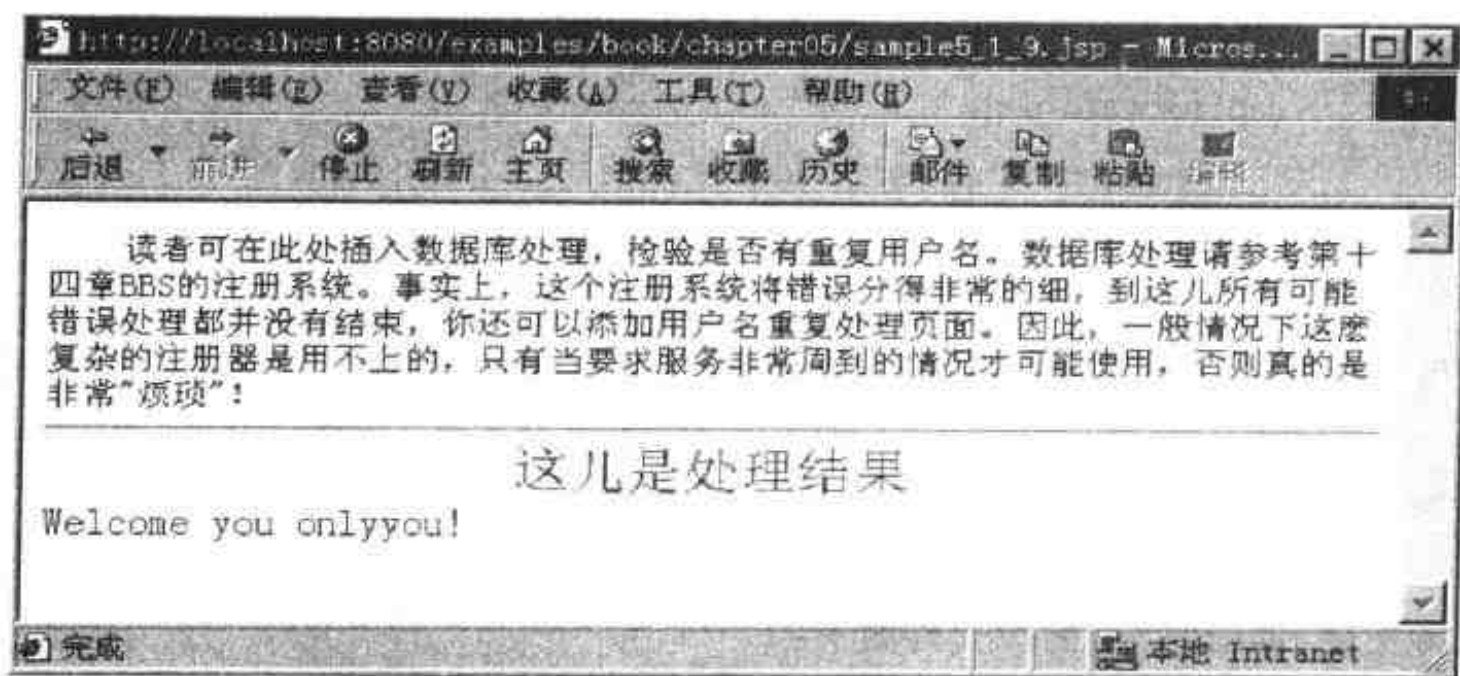


图 5-10 <jsp:forward />与<jsp:param />试验

## 5.2 Template Data

### 5.2.1 Template Data

Template Data 直译为模板数据,根据其语义也可译为无解释数据。这两种翻译都不太好,易于与其它应用程序的模板概念相混淆。因此,我们将其作为一个专有名词,不译直接引用。

通常,Template Data 与 fixed template data 有相同含义:JSP 文档中任何未被 JSP 规范描述的部分,称为 Template Data。例如:HTML 标记、XML 标记以及文本。Template Data 将被直接返回到客户端或被组件处理。注意,在这个定义中,Template Data 是相对的,有确定范围,是在 JSP 文文件中相对元素而言。如 HTML 标记,JSP 不会对其作任何处理,但浏览器却对它“情有独钟”。

### 5.2.2 Template Text

从 Template Data 的含义不难看出,Template Text 是 Template Data 的子集。用“引用与转义”规则把它处理(通常是置换)后,可以直接返回到客户端。

### 5.2.3 引用与转义

#### 1) 在脚本元素中的引用

文本“%>”是“% \>”的引用。

#### 2) 在 Template Text 中的引用

文本“<%”是“< \ %”的引用。

#### 3) 在属性中的引用

(1) 单引号“'”是“\ '”的引用。

(2) 双引号“”是“\ ””的引用。

(3) 反斜线“\”是“\ \”的引用。

(4) “%>”是“% \>”的引用。

(5) “<%”是“< \ %”的引用。

引用与转义看起来简单,事实并非如此。例如用 `out.print()` 在页面上显示脚本片段结束标记“%>”。根据上文叙述,似乎应该有如下的表述方式:`<% out.print("% \>"); %>`执行看看,会报告出错了。到底应该是怎样的呢?我们举个例子,然后再总结规则。

#### 例 5-10

##### 转义与引用

先给出页面效果图,以便有一个直观印象。页面效果如图 5-11 所示。

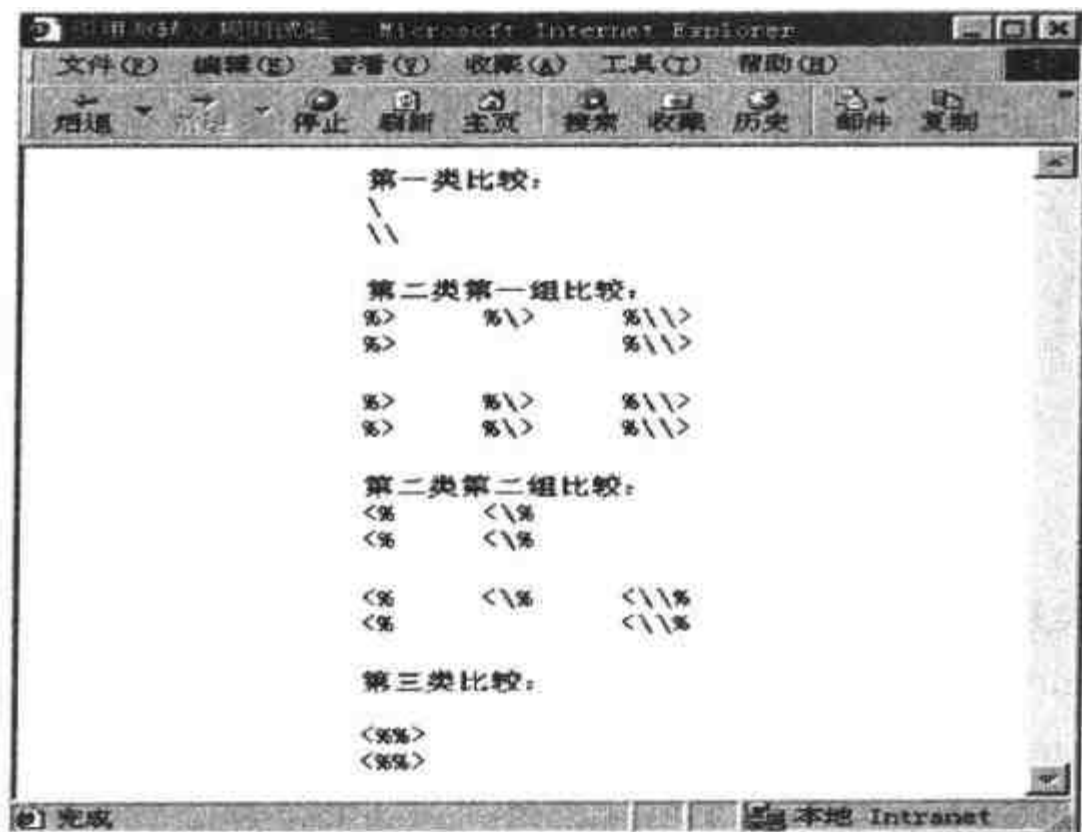


图 5-11 引用与转义规则试验

程序代码比较复杂,去掉冗余显示如下,但还是比较复杂。如果在这方面有困难或有这方面的需要(如写一个 JSP 教程的页面),那么,请仔细看看,一定会茅塞顿开。不管怎样,建议先看代码后的两点规则。代码如下:

```
<html>
```

```
<head>
<title>引用与转义规则试验</title>
</head><body>
```

第一类比较:

```
<%out.print("\\");%>
\\<br>
```

<p>第二类第一组比较:

```
<%out.print("%>");%>
<! -- out.print("% \>");% -- >
<%out.print("% \>");%>
<! -- out.print("% \>");% -- >
<%out.print("% \>");%>
<br>
<! -- %out.print("%>");% -- >
<! -- out.print("% \>");% -- >
<%out.print("% \>");%>
<! -- out.print("% \>");% -- >
<%out.print("% \>");%>
</p><p>
%>
% \>
% \>
<br>
%>
% \>
% \>
</p>
```

<p>第二类第二组比较:

```
<%out.print("<");%>
<! -- %out.print("< \");% -- >
<%out.print("< \");%>
<br>
<%out.print("<");%>
<! -- %out.print("< \");% -- >
<%out.print("< \");%>
</p><p>
< %
```



```

<lt; \ %
<lt; \\ %
<br>
<% -- <% -- %>
< \ %
< \\ %
</p>

<p>第三类比较:
<% %>
<br>
<lt;% % >
<br>
<%out.print("<lt;% % >");%>
</p>
</body>
</html>

```

通过这个例子,现将引用与转义的规则总结如下:

(1) 第一条规则:Java 字符串转义。在双引号括起来的字符串中,出现反斜线“\”,必是转义符。反之,出现在没有用双引号括起来的字符串中,不是转义符。是转义符的,必先执行转义,如果转义合法则转第二条规则执行。否则,报错。

(2) 第二条规则:JSP 转义。Java 字符串转义合法后又分三种情况再执行转义。

►脚本片段中:“% \ >”转义为“% >”。

►template text 中:“< \ %”转义为“< %”。

►属性中:“\ '”转义为单引号“'”、“\ ””转义为双引号“””、“\\ ”转义为反斜线“\ ”、“% \ >”转义为“% >”、“< \ %”转义为“< %”。

让我们分析几个典型的情况。

<%out.print(" \\ ");%>;首先执行 Java 字符串转义,“\\ ”转义为“\ ”。然后转第二条规则,没有需要再转义的了,完毕。输出“\ ”。

HTML 文本“\\ ”:非 Java 字符串,也不包含在 JSP 转义规则内。所以直接输出“\\ ”。

<%out.print("% \\ >");%>;首先执行 Java 字符串转义,“% \\ >”转义为“% \ >”。然后转第二条规则执行 JSP 转义,“% \ >”转义为“% >”,完毕。输出“% >”。

其它情况类似。总之,首先执行 Java 字符串转义,然后再执行 JSP 转义。问题的关键在于是否是 Java 字符串。

下面键入<%out.println("a \ b");%>并执行它,得到了什么。这可能是 Tomcat 的一个 bug。

## 5.3 为虚拟网站加第一块砖

学完第 5 章,看看能为虚拟网站做点什么?本书为虚拟网站添加的第一块砖是广告轮显。相信读者还有更多更好的想法。

在开始之前的几点假设:

(1) 按照第 2 章所述,将虚拟网站所有组件都放在了 chapter14 这个目录,关于它的结构请读者留意一下。当然完全可以按自己的规划去安排,并且非常支持这种行为。因为真正过手的才是自己的知识。

(2) 读者已经按第 2 章所述建立了页头、主体和页脚。

### 5.3.1 连接页头、主体和页脚

现在利用本章学到的 include 行为将页头、主体和页脚连接起来,使它成为一个完整的页面。首先需要将它们动点小手术,然后改名存档。调整分别如下所述。

将页头 header1.html 改名为 header1.jsp 并修改为类似如下所示源代码:

```
<%@ page contentType="text/html; charset=gb2312" %>
<table cellspacing="0" cellpadding="0" border="0" align="center" width="750">
<tr valign="baseline">
    <td width="135">Data</td>
    <td width="445">Advertisement</td>
    <td width="170">Time</td>
</tr>
<tr bgcolor="#006633" valign="middle">
    <td height="16" colspan="3">
        <div align="right"><font size="2" color="#FFFFFF">首 页 | 教 程 |
        BBS | 聊天室 | 电子商务 | 留言板 | 下载中心 | E_Mail</font></div>
    </td>
</tr>
</table>
```

将页脚 footer.html 改名为 footer.jsp 并修改为类似如下所示源代码:

```
<%@ page contentType="text/html; charset=gb2312" %>
<table width="750" border="0" cellspacing="0" cellpadding="0" align="center">
    <tr height="2" bgcolor="#666666">
        <td></td>
    </tr>
    <tr height="12">
```

```

        <td> </td>
    </tr>
    <tr> <td>
        <div align="center"> <font size="2">版权所有 雨阳隆春</font> </div>
    </td> </tr>
    <tr> <td>
        <div align="center"> <font size="2"> &copy;copyright all right
        reserve</font>
        </div>
    </td> </tr>
</table>

```

最后将 body.html 改名为 index.jsp 并使用 include 行为将 header1.jsp 和 footer.jsp 包含进来。得到的 index.jsp 源代码如下：

```

<HTML>
<HEAD>
<TITLE> 主页</TITLE>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</HEAD>
<body bgcolor="#FFFFCC" leftmargin="10" topmargin="8" alink="#FFFF33"
    vlink="#990099">
<% @ page contentType="text/html; charset=gb2312" %>
<jsp:include page="template/header1.jsp" flush="true"/>

<table width="750" height="280" border="0" cellspacing="0" cellpadding="0"
    align="center">
    <tr>
        <td width="100" valign="top" align="center">Function</td>
        <td width="1" valign="top" bgcolor="#006633"></td>
        <td valign="top">Display</td>
    </tr>
</table>

<jsp:include page="template/footer.jsp" flush="true"/>
</body>
</HTML>

```

这时,应该有类似图 5-12 所示的效果。

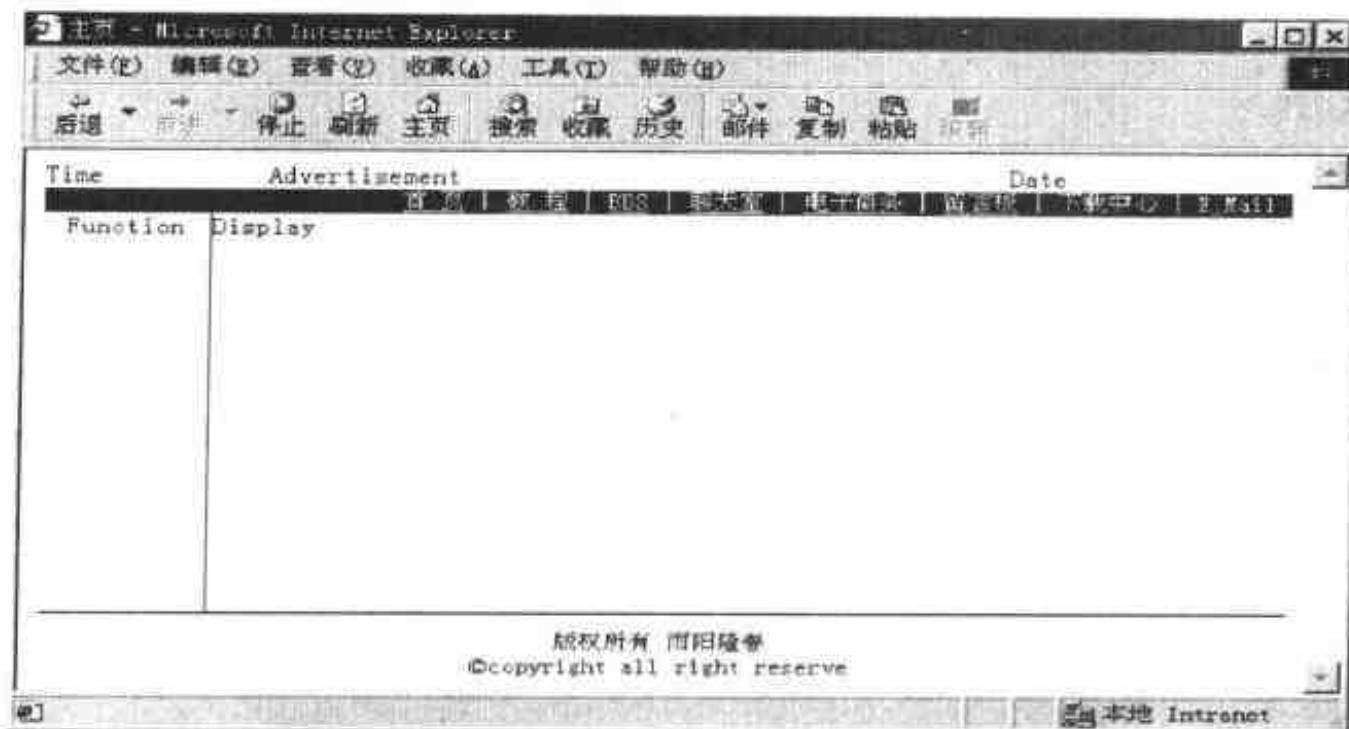


图 5-12 页头、主体和页脚连接效果图

### 5.3.2 广告轮显

用过 ASP 的读者一定对 Microsoft 的广告轮显组件记忆犹新, 因为 JSP 中没有这样的组件, 太痛苦了。现在你可以大展身手了, 实现 JSP 的广告轮显, 并且与 ASP 的一样好。ASP 中使用一个文本文件记录广告的参数, 我们也使用这种方式。先来看看页面效果。

#### 1. 页面效果

页面效果如图 5-13 所示。

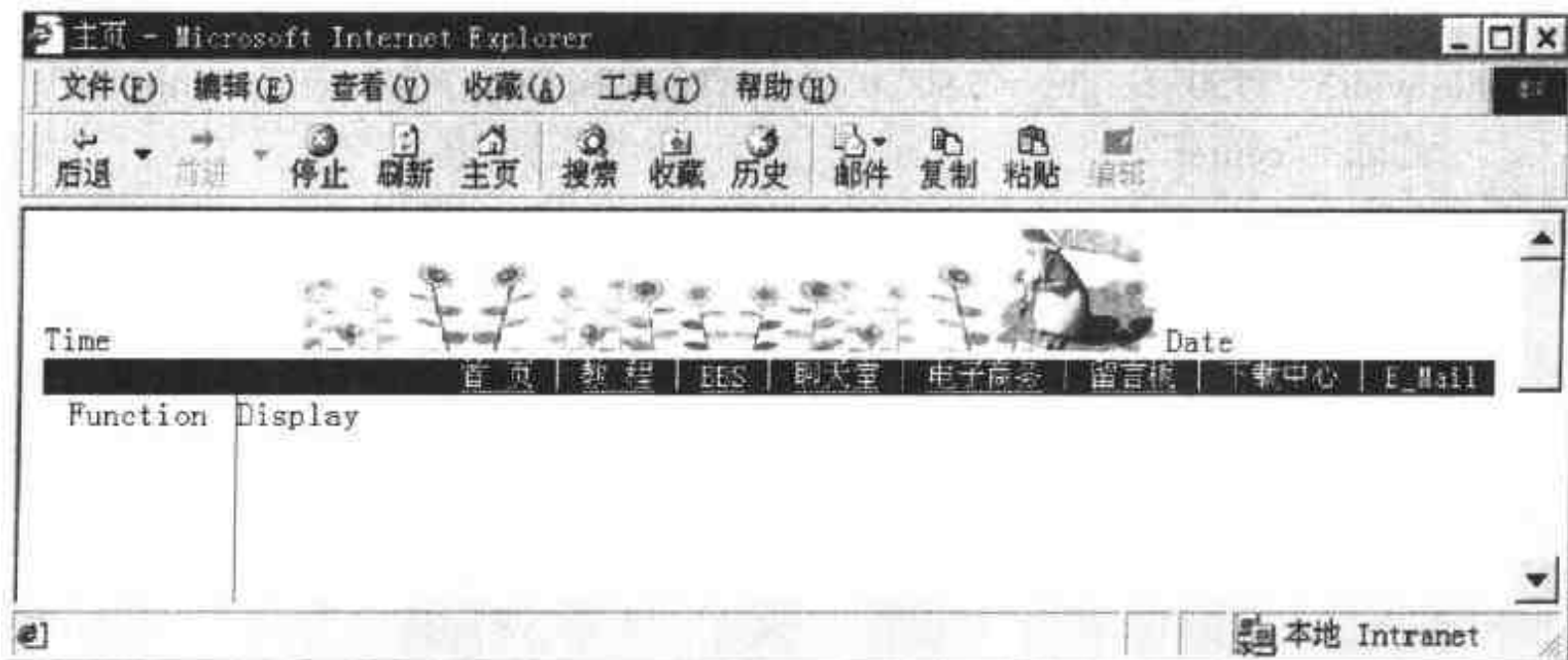


图 5-13 广告轮显效果图

#### 2. 程序源代码

程序源代码如下:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ page import="java.io.*" %>
```

```
<%@ page import = "java.util. *" %>
<%! boolean readeof = false; %>
<%! String linetemp = ""; %>
<%! String picaddress = ""; %>
<%! String urladdress = ""; %>
<%! String info = ""; %>
<%! int width,height,border,choose; %>
<%! int ratearray[] = new int[100]; %>

<%
try{
    File file = new File("../webapps/examples/book/chapter14/adver/adver.txt");
    RandomAccessFile raf = new RandomAccessFile(file,"r");
    //获得广告栏宽度
    width = Integer.parseInt(raf.readLine());
    //获得广告栏高度
    height = Integer.parseInt(raf.readLine());
    //获得广告栏边线宽
    border = Integer.parseInt(raf.readLine());
    linetemp = raf.readLine();
    if(linetemp == null){
        readeof = true;
    }
    else{
        readeof = false;
    }
    int i = 0;
    int j = 1;
    int k = 0;
    while(! readeof){
        if(j % 4 == 0){
            //记录一共有多少个广告
            i++;
            //概率累加并记入数组
            k = k + Integer.parseInt(linetemp);
            ratearray[i] = k;
        }
        j++;
        linetemp = raf.readLine();
    }
}
```

```

        if(linetemp == null){
            readeof = true;
        }
    }
    ratearray[0] = i;
    Random random = new Random();
    //产生随机数并判断落入哪个区间,即哪个广告被选中
    float f = random.nextFloat() * ratearray[ratearray[0]];
    for(int r = 1; r <= ratearray[0]; r++){
        if(f < ratearray[r]){
            choose = r;
            break;
        }
    }
    //将选中广告信息读出
    raf.seek(0);
    for(int s = 1; s < 4 * choose; s++){
        raf.readLine();
    }
    picaddress = raf.readLine();
    urladdress = raf.readLine();
    info = raf.readLine();
} catch (FileNotFoundException e){
    System.out.println("file not found:" + e);
}
catch (IOException e){
    System.out.println("io wrong" + e);
}
% >
<HTML>
<HEAD>
<TITLE>广告牌</TITLE>
</HEAD>
<body>
<a href = "< % = urladdress % >" target = "_ self" >
<img src = "< % = picaddress % >" width = "< % = width % >" height = "< % =
height % >" border = "< % = border % >" alt = "< % = info % >" >
</a>
</body>

```

</HTML>

### 3. 说明与改进建议

本广告轮显使用的是数组记录广告的总数和概率分配,然后判断产生的随机数落入哪个区间从而选中某个广告。这个区间是由数组相邻两个元素界定的。然后再从文件中读出选中广告的信息显示。这个广告轮显实现有两个问题,一是最大广告数量不能超过100个,二是两次读取文件。改进建议是将广告信息构造成一个类,比如叫 adver 类,然后采用向量记录 adver 类的对象,这样就没有最大广告数量的限制,并且只需要读一次文件。关于采用向量技术的实现请参见第14章购物车的实现,二者是相似的。

这里采用这种实现无非是想多介绍点实现方案。每件事物都有很多种实现方案,关键在于最终获得一个良好的实现方案。

## 5.4 本章小结

本章较为深入的讲述了JSP的基本语法。需要重点掌握的是声明、表达式、脚本片段以及 include 行为和 forward 行为,最后讲解了为虚拟网站添加了一个动态的广告的内容。

# 第 6 章 Servlet

本章将要说明了两个事实：

(1) Servlet 本身功能是非常强大的。Servlet 是 JavaServer 体系结构的一部分。而 JavaServer 体系结构是 Sun(JavaSoft)致力将 Java 扩展到服务器领域的框架结构,它定义了服务、服务器进程、Servlet API。JavaServer 体系结构是相当先进、合理的,使用 Servlet 扩展服务器功能。因此可以将 Servlet 认为是服务器端的 Applet,可以增加、删除或配置 Servlet,以实现特定的功能以及实现功能升级等等。一句话,Servlet 是 JavaServer 体系结构功能的最终实现者和体现者。

(2) Servlet 与 JSP 有直接血缘关系。JSP 是 Servlet 的直接继承者,甚至可以认为 JSP 是 Servlet 的简化和特定实现。事实上,目前以及在可以预见的将来,JSP 只会实现 HTTP 协议,因为 Servlet 和 JSP 有明确的功能、市场和技术层次的定位。JSP 是 Java Enterprise 平台(Java 企业级平台)的门户,提供给网站或网页设计者一种更加容易的动态网页开发手段。而 Servlet 则用来实现 Java 企业级平台的核心商业逻辑。

## 6.1 Servlet 概述

我们惊奇地发现,Java 诞生 10 年来,其触角已遍布各个领域。Java 不仅仅定义了一种计算机语言,而且还提供了一整套客户/服务解决方案。这个方案,一方面程序可以自动的下载到客户端并执行,使我们有了初步的交互能力,但伴随产生的是安全、兼容等问题。这就是大家所熟知的 Applet 和客户端图形用户界面组件(如客户端 JavaBeans)。Applet 的确是客户/服务计算环境的重要组成部分,但是它仅仅是问题的一半。另一方面,程序可在服务端被动态的加载和执行,使我们有了更强大的交互能力,并较好的解决了安全、兼容等一系列问题。这就是问题的另一半——Servlet。

### 6.1.1 什么是 Servlet

Servlet 是 Java2 中新增的一个全新功能,它是由容器管理、可以产生动态内容的页面组件。Servlet 是一个扩展模块,它扩展 request-response 这种类型的服务器,如一个能运行 Java 的 web 服务器。

Servlet 是小型的、与平台无关的 Java 类,它被编译成结构中立的字节码,由 Web 服务器动态加载和执行。Servlet 通过容器实现的 request 和 response 实例与页面客户交互。



这种 request-response 模型是基于 HTTP 协议的行为。Servlet 可以被认为是服务端的 Applet。Servlet 被 Web 服务器加载和执行,就如同 Applet 被 Web 浏览器加载和执行一样。与 Applet 不同的是 Servlet 没有图形用户界面(GUI)。Servlet 通过 Web 服务器从客户端接收请求,执行某种作业,然后返回结果,其工作流程如图 6-1 所示。例如,Servlet 可从 HTML 订购输入表单中获取数据,然后用商业上的算法来更新公司相应的订单数据库。

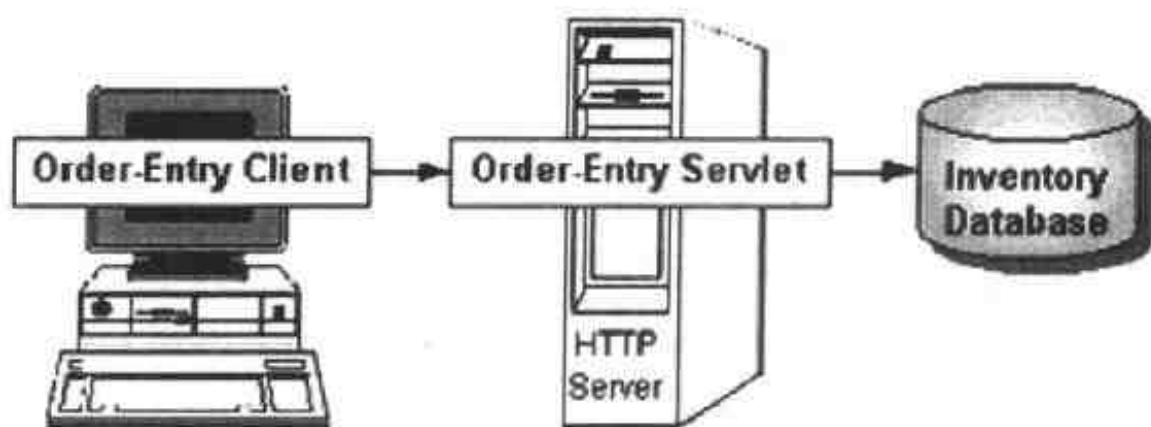


图 6-1 Servlet 工作流程

### 6.1.2 为什么要使用 Servlet

Servlet 可有效的代替 CGI 脚本。它提供了一种轻松编写动态文档,并且快速执行该文档的方法。Servlet 也定位解决只能用平台特定的 API 编写服务端程序这个问题,即解决 CGI 可移植性差的通病。CGI 脚本是用 Perl 或者 C 语言编写的,它们总是和特定的服务器平台紧密相关。Servlet 使用 Java Servlet API 开发,Java Servlet API 是 Java 的一个标准扩展,所以它们一开始就与平台无关。这样,Java 一次编写处处运行,同样可在服务器上实现。

Servlet 支持多人协作。Servlet 可并发处理多个请求,并且同步这些请求。这使得 Servlet 支持像在线会议这样的系统。

Servlet 支持转发请求以平衡负载。Servlet 可转发请求给其它的服务器和 Servlet,因此 Servlet 可用来平衡几个服务器之间的负载,并且可根据任务类型或组织结构在几个服务器上分隔出一个单独的逻辑服务。

### 6.1.3 Servlet 与 CGI 相比有哪些优点

Servlet 有许多传统 CGI 脚本语言所不具备的独特优点:

Servlet 是持久的。Servlet 只需 Web 服务器加载一次,而且可以在不同请求之间保持服务(例如一次数据库连接可以持续使用直至 Servlet 销毁它)。与之相反,CGI 脚本是短暂的、瞬态的。每一次对 CGI 脚本的请求,都会使 Web 服务器加载并执行该脚本。而这个 CGI 脚本运行结束时,又会把它从内存清除。因此 CGI 脚本的每一次使用,都会造成程序初始化过程(例如连接数据库)的重复执行。

Servlet 是快速的。与 CGI 相比,由于 Servlet 只需要被加载一次并且长驻内存(如果

必要的话)自然提供了更快的响应速度,更少的内存占用(最终结果),更佳的综合性能。

Servlet 是与平台无关的。Servlet 是用 Java 编写的,自然秉承了 Java 优秀的平台无关特性。

Servlet 是可扩展的。Servlet 基于 Java,继承了 Java 固有的一切优良特性。Java 是健壮的、可移植的、易扩展的、面向对象的程序设计语言。它很容易扩展从而适应你的需求,Servlet 自然也具备了这些特征。

Servlet 是安全的。从外界调用一个 Servlet 的唯一方法就是通过 Web 服务器。这提供了高级别的安全性保障,尤其是您的 Web 服务器有防火墙保护的时候。

Servlet 可以在多种客户机上使用。Servlet 是 Java 的继承者,所以您可以很方便地在 HTML 中使用它们,就如同使用 Java Applet 一样。

可以看出,Servlet 以及 JSP 的一切优秀特性无不来自于 Java,Java 是它们“力量”的源泉。Sun 正在做的工作就是充分利用 Java 的“能量”,反过来又巩固 Java 的地位。

### 6.1.4 Servlet 与 JavaServer 体系结构的关系

Servlet 是 JavaServer 体系结构的一部分。JavaServer 体系结构是 Sun(JavaSoft)致力于将 Java 扩展到服务器领域的框架结构,它定义了服务、服务器进程、Servlet API。下面来看看 JavaServer 体系结构及其定义的框架结构。

#### 1. 服务框架

服务框架定义了实现服务的一系列接口,通过这些接口提供的服务,使多个服务线程与客户端实现交互。服务被定义为某个协议(如 HTTP、FTP)的一个实现。注意目前 JavaServer 体系只定义了基于连接的协议。不过,不久将加入对基于数据包的协议的支持。JavaServer 体系结构提供的核心服务类包括系统管理、线程管理、连接管理、会话管理以及安全保障。

在服务初始化时,要为其分配特定的服务套接字。在取得一个服务 socket 之后,服务器将创建一个服务线程池,如图 6-2 所示。池中的每一个服务线程都等待响应连接请求。一旦接到连接请求,就会有一个服务线程在这个连接上实现协议确定的全部交互。

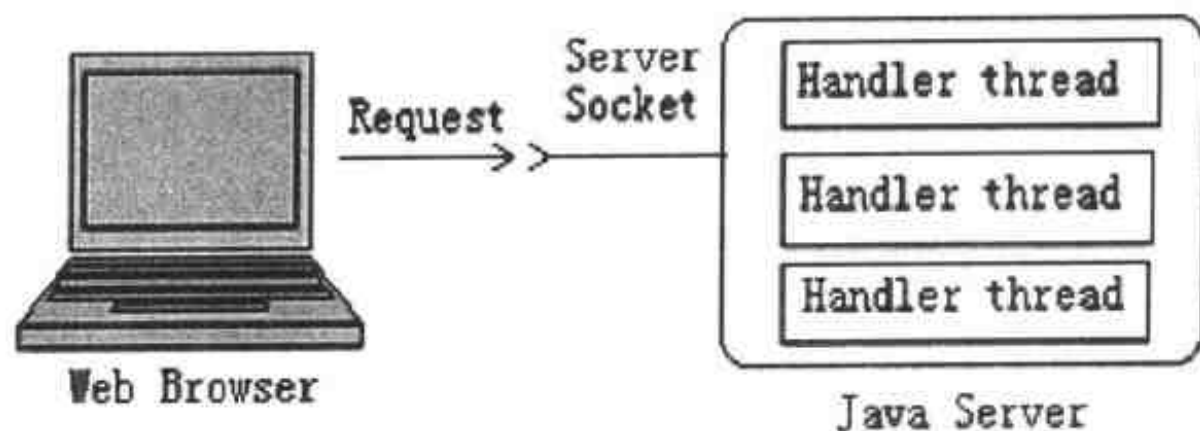


图 6-2 JavaServer 服务框架

服务线程池的大小是动态的,还有上下限的限制。

## 2. 服务器框架

服务器是 Java 虚拟机的一个实例。一个服务器可以支持多个并发的服务,这些服务在服务器进程初始化过程中启动。如图 6-3 所示。一个服务器,例如 Java Web 服务器,它一般会启动系统管理服务、HTTP 服务,很可能会启动 Web 代理服务。在服务器运行中,仍然可以增加、删除或配置服务。

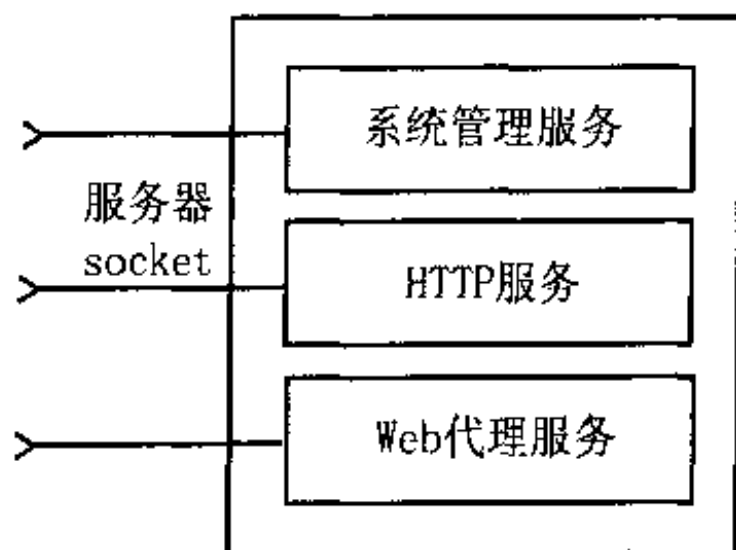


图 6-3 JavaServer 的服务器框架

## 3. Servlet 框架

Servlet 是符合 JavaServer 体系结构定义的特殊接口的 Java 对象。如图 6-4 所示。Servlet 由服务加载和调用,并且一个服务可以同时使用多个 Servlet。无论是由服务器提供的内置 Servlet,还是用户编写的扩展 Servlet,都可以将其认为是一种扩展服务器功能的简单手段。Servlet 可在服务器运行的过程中被增加、删除或配置。

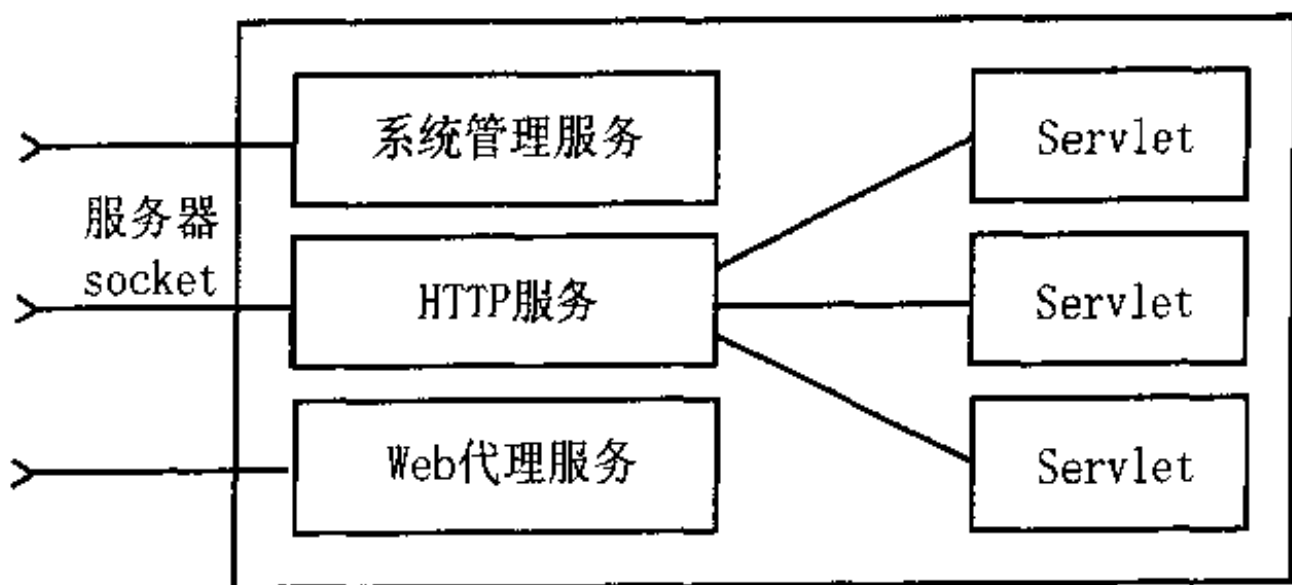


图 6-4 JavaServer 的 Servlet 框架

## 6.2 Tutorial

本节通过两个比较简单的例子讲述完整的 Servlet 页面应用程序开发过程。读者将看到 Servlet 的编写其实并不比 JSP 困难多少。

### 6.2.1 编写 Servlet

Servlet 的编写需要以下两个步骤：

(1) 创建一个扩展 `javax.servlet.http.HttpServlet` 界面的 Servlet 类。`javax.servlet.http.HttpServlet` 界面是 `javax.servlet.GenericServlet` 的扩展界面,它分析 HTTP 首部并将客户端信息打包成 `javax.servlet.http.HttpServletRequest` 类的一个实例。

(2) 重写 `doGet` 和 `doPost` 方法之一或全部。这里是 Servlet 实际完成工作的地方。假设客户端发出一个 GET 请求(默认的 HTTP 请求方法),那么 `doGet` 方法被调用,它将把某个 HTML 页面返回给客户端。

`doGet` 方法内用户的请求表示为一个 `HttpServletRequest` 对象。对用户的响应表示为一个 `HttpServletResponse` 对象。如果返回客户端的是文本数据,那么使用从 `HttpServletResponse` 对象获得的 `Writer` 对象发送应答。

#### 例 6-1

第一个 Servlet 程序,congratulate。程序源代码如下：

```
//import needful package
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class congratulate extends HttpServlet{
    public void doGet(HttpServletRequest request,HttpServletResponse response)
        throws IOException,ServletException
    {
        //set the content type of response
        response.setContentType("text/html");
        //create a PrintWriter to write the response
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>Congratulate you</TITLE>");
        out.println("</HEAD>");
    }
}
```

```
out.println("<body>");
out.println("<p><font size=7 color='red'>Congratulate You!
    </font><br></p>");
out.println("<p> This is your first Servlet, you do very well! </p>");
out.println("</body>");
out.println("</html>");
out.flush();
```

这个程序非常简单,但是仍然有几点需要注意:



注意:首先你需要导入类库。java.io.\*、javax.servlet.\* 和 javax.servlet.http.\* 是三个必不可少的包。如果不导入类库,可能不得不使用 Java 完整的类库名。如 java.io.PrintWriter。



注意:Servlet 与 Java 一样,必须封装成一个类。但是与 Java 不同的是它并不需要 main() 方法。通常 Servlet 扩展 javax.servlet.http.HttpServlet 界面,因为现在 Servlet 主要应用于 Web 领域。



注意:必须至少覆盖 doGet()、doPost()、doPut() 等方法中一个。通常覆盖 doGet() 方法,因为这是默认的 HTTP 请求方法。输出处理结果到客户端之前,必须设置 HTTP 头部数据。例如使用 setContentType() 方法设置内容类型。Servlet 沿用了传统 CGI 通过 IO 系统输出的方式。

现在,给出页面效果图,如图 6-5 所示。待我们举个稍微复杂的,可以实现基本交互功能的例子后,再一起来看看 Servlet 的编译、执行。



图 6-5 First Servlet——Congratulate you

#### 例 6-2

用 JSP、Servlet 实现简单的交互。JSP 和 Servlet 分别对应不同的应用层次。JSP 适

合产生页面,而 Servlet 适合封装核心商业逻辑。现在使用 JSP + Servlet 的模式实现一个简单的交互。例如某个人向您问好,您有多少种反应呢? 你的反应很重要,更重要的是哪个人根据您的反应能作出多少种相应的反应。反应类型越丰富,就说明这个人越会“察言观色”。在这里,称核心逻辑越强大。本例中实现核心逻辑的 Servlet 源代码如下:

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

public class eidolon extends HttpServlet{
    public void doGet(HttpServletRequest request,HttpServletResponse response)
        throws IOException,ServletException
    {
        response.setContentType("text/html;charset = gb2312");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>eidolon</title>");
        out.println("<meta http-equiv = 'Content-Type' content = 'text/html; charset = gb2312'>");
        out.println("</head>");
        out.println("<body bgcolor = ' # FFFFCC'>");
        out.println("<p><font size = 5 color = red>Servlet eidolon:</font></p>");
        String name = request.getParameter("name");
        String where = request.getParameter("where");
        boolean noname,nowhere;
        if(name.equals("")){
            name = "<font size = 4>Oh, my god! your daddy forgot give your name? </font><br>";
            out.println(name);
            noname = false;
        }
        else{
            out.println("<font size = 5>Glad to see you,<font color = red>"
                + name + "! </font></font><br>");
            noname = true;
        }
        if(where.equals("")){
            where = "<font size = 4>no hometown, but must be one that you come from
```

```

        your mum's...</font><br>";
        out.println( where);
        nowhere = false;
    }
else{
    out.println("<font size = 5>Your hometown is <font color = red>"
        + where + "</font>," + "there is beautiful place! </font><br>");
    nowhere = true;
}
if( ! (noname&nowhere)){
    out.println("<font size = 5>oh, I am ...jsut forgot input! </font>");
    out.println("<a href = '/examples/book/chapter06/aidolon.jsp' >"
        "<font size = 5>芝麻开门</font></a>");
}
out.println("</body>");
out.println("</html>");
}

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

public void init(ServletConfig cfg)
    throws ServletException
{
    super.init(cfg);
}

public void destroy()
{
    super.destroy();
}

```

这个程序编写的 Servlet 较为完整,必要的时候,才覆盖 init()和 destroy()方法,否则只需要覆盖您期望的 doXXX 方法。如果需要处理 GET 请求,那么覆盖 doGet 方法,其余类推。这里写出 init()和 destroy()方法只是表示一个比较完整的结构,并无实在意义。

这个例子可看出编写 Servlet 的一般顺序:

(1) 如果需要做准备工作,例如连接到数据库,那么在 `init()` 方法中编写初始化代码。事实上连接到数据库,是 `init()` 方法的典型应用。

(2) 然后覆盖 `doGet` 或其它 `doXXX` 方法,在其中编写核心处理过程。

(3) 设置响应头部信息,如内容类型,字符编码。

(4) 使用 `response` 的 `getWriter()` 或 `getOutputStream()` 方法获得输出流。

(5) 使用输出流写 HTML 文档头部。

(6) 使用 `request` 的 `getParameter()` 或 `getParameterNames()`、`getParameterValues()` 等方法获得请求参数及其值。

(7) 处理请求。

(8) 响应输出。

(9) 写 HTML 文档尾部。

(10) 善后工作,例如断开到数据库的连接。

## 6.2.2 编译 Servlet

Servlet 是一个 Java 类,因此运行前,必须将它编译。现在需要做以下工作:

(1) 确认安装的 JDK 支持 Servlet,并且环境变量 `PATH` 和 `CLASSPATH` 被正确设置。这一步应该在第 1 章已经解决。建议不要使用 JDK1.1.4 以前的版本。

(2) 确认安装了 Servlet API 类文件,因为编译的时候需要它。Servlet API 由于并非核心类库,因此并不是所有的基于 Java 的服务器都支持 Servlet。可以根据需要下载、安装支持 Servlet 的引擎。通常,在 Tomcat 中需要将“`C:\Sun\Tomcat\lib\webserver.jar;C:\Sun\Tomcat\lib\servlet.jar`”加入 `CLASSPATH`。JSWDK 中,将“`C:\Sun\JSWDK\webserver.jar;C:\Sun\JSWDK\lib\servlet.jar`”加入 `CLASSPATH`。

注意文件名必须与类名一致,包括大小写。最好不使用包名。

(3) 在 DOS 下或 MS-DOS 下键入:[路径] `javac` 文件名。当然可以只键入 `javac` 以查看帮助文件。

如果还有问题,根据出错信息检查上面 4 个步骤。例如,出错信息为“无法识别的类”,通常第二步有问题,找不到 Servlet API 类文件。

## 6.2.3 运行 Servlet

需要做以下工作:

(1) 确认正确安装了支持 Servlet 的 Web 服务器。

(2) 将编译生成的类文件放在 Web 服务器特定的目录下面。Tomcat 下,将 Servlet 类文件放在 `web-inf\classes` 中。Tomcat 中 `admin`、`examples`、`Root` 和 `test` 目录下都有 `web-inf\classes` 子目录,任选一个。JSWDK 下将 Servlet 放在 `examples\web-inf\servlet` 或 `webpages\web-inf\servlet` 中。

(3) 读者有两种方法运行 Servlet。

(4) 在浏览器窗口中直接键入 Servlet 的 URL。这就是执行例 6-1 `congratulate` 使用



的方法。该例在 Tomcat 下,congratulate.class 放在了 examples \ web-inf \ classes 目录中,因此在浏览器地址栏中键入: http://localhost:8080/examples/servlet/congratulate。如果在 JSDK 下,将 congratulate.class 放在 examples \ web-inf \ servlet 中,相应的在浏览器地址栏中键入: http://localhost:8080/examples/servlet/congratulate。

(5) 从 HTML 页面调用 Servlet。从一个 HTML 页面中调用一个 Servlet 仅仅是在适当的 HTML 标记中使用 Servlet 的 URL。能带 URL 的标记包括锚、表单和 Meta。

下面继续做例 6-2 未完部分,编写调用 Servlet 的页面,这里使用 JSP 页面调用 Servlet——使用表单提交调用 eidolon Servlet。

编写调用 eidolon Servlet 的 JSP 文件,源代码如下:

```
<%@ page import = "java.util. *" %>
<HTML>
<HEAD>
<TITLE>eidolon</TITLE>
<meta http-equiv = "Content-Type" content = "text/html; charset = gb2312">
</HEAD>
<body bgcolor = "#FFFFCC">
<% if (Calendar.getInstance().get(Calendar.AM_PM) == Calendar.AM) |%>
<p><font size = 5>Good Morning! </font></p>
<% | else | %>
<p><font size = 5>Good Afternoon! </font></p>
<% | %>
<P><font size = 5>I am&nbsp;
<font color = red>Servlet eidolon</font>
,come from another space.</font>
</p>
<p><font size = 5>You? </font></p>
<form method = "post" action = "/examples/servlet/eidolon">
  <p>your name:
    <input type = "text" name = "name" maxlength = "16" size = "16">
  </p>
  <p>come from:
    <input type = "text" name = "where" maxlength = "16" size = "16">
  </p>
  <p>
    <input type = "submit" name = "Submit" value = "Submit">
    <input type = "reset" name = "Reset" value = "Reset">
  </p>
</form>
</body>
```

</html>

可谓万事具备,只欠东风。JSP 文件页面效果如图 6-6 所示。

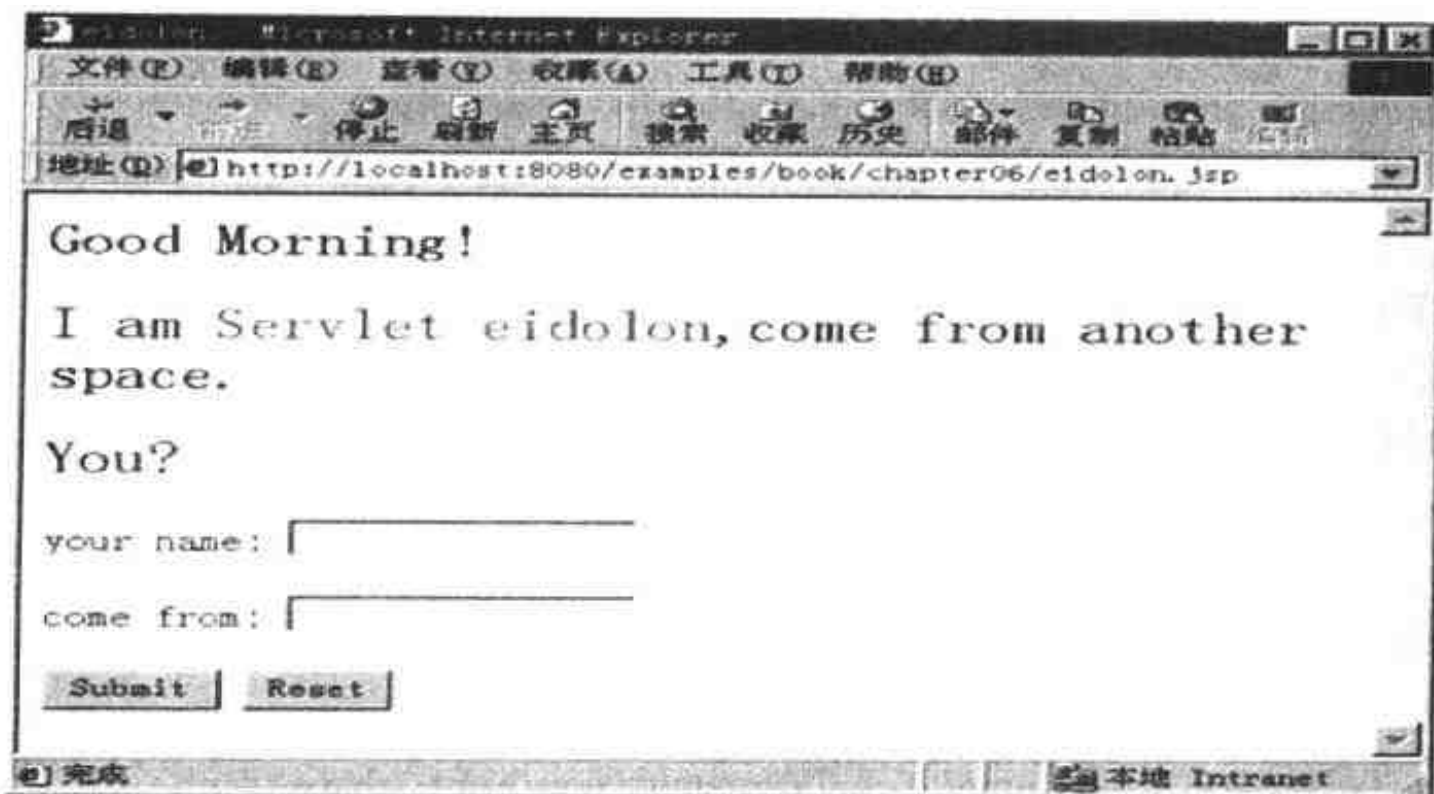


图 6-6 来自 eidolon 的问候

输入数据分三种情况:第一种情况名字、地址都输入;第二种情况只输入名字;第三种情况只输入地址,分别看一下页面效果。这里给出的是第三种情况下获得的页面效果图,如图 6-7 所示。



图 6-7 来自 eidolon 的回答

## 6.2.4 Servlet 的基本执行流程

讨论完两个例子后,我们来看看 Servlet 的基本执行流程。

先回顾一下 JavaServer 体系机构以及 HTTP 请求如何调用 Servlet。客户(Web 浏览器)向服务器发送一个加载某个 HTML 页面的请求,服务器上的 HTTP Web 服务接收该请求,并识别出这是一个读 HTML 文件的请求,然后调用文件读写 Servlet 实现具体的文

件输入/输出。该 HTML 页面将被返回给客户端并由 Web 浏览器显示。此时,事情并没有完,交互还没有实现。Web 浏览器将再次发出请求,假设发出的是一个 HTML POST 请求,该请求仍由 HTTP Web 服务接收。如果该 POST 请求要求加载一个 Servlet,那么该请求将被转发给资源调用 Servlet,由资源调用 Servlet 调用并加载被请求的特定 Servlet。之后,由被加载的 Servlet 处理请求并将处理结果通过 HTTP Web 服务返回给客户端。可以看出,文件读写 Servlet 和资源调用 Servlet 都是 HTTP Web 服务的内置 Servlet 组件,由它们实现调用其它的扩展 Servlet。可见通过 Servlet,HTTP Web 确实实现了功能扩展。

Servlet 的基本执行流程如:

(1) 加载被请求 Servlet。如果被请求 Servlet 尚未加载,资源调用 Servlet 将调用该 Servlet 并解析、加载。请注意,Servlet 既可以从本地加载,也可以从远程主机加载——你可以通过 Java Web 服务器的 HTTP Web 服务的 Servlet 控制台来控制。与传统 CGI 不同,Servlet 只加载一次,然后 Servlet 的多线程能力将处理来自多个客户的多个请求。

(2) 初始化 Servlet。Servlet 的 `init()` 方法被调用,以便 Servlet 执行诸如连接数据库之类的初始化操作。`init()` 方法只在 Servlet 加载后调用一次,而且对 Servlet 的其他任何调用都要在 `init()` 方法执行结束后才能处理。

(3) 对于 HTML POST 请求,调用 Servlet 的 `doPost()` 方法。其它类型的 HTTP 请求,调用相应的方法。

Servlet 执行某种处理,然后通过输出流返回响应。

响应最初由 HTTP Web 服务接收。Web 服务可能还会进行某种处理。

## 6.3 与客户端交互

HTTP Servlet 通过它的 `service` 方法处理客户请求。`service` 方法通过分配每个请求给相应的处理方法达到支持标准的 HTTP 客户请求的目的。

### 6.3.1 request 和 response

这部分讨论表示客户端请求(`HttpServletRequest` 对象)和 Servlet 的响应(`HttpServletResponse` 对象)的对象。这些对象提供给 `service` 方法和 `service` 方法调用处理 HTTP 请求的其它方法。

`HttpServletRequest` 类中处理客户端请求的方法都带有两个参数:

- ▶ `HttpServletRequest` 对象,封装了来自客户端的数据。
- ▶ `HttpServletResponse` 对象,封装了对客户端的响应。

#### 1. `HttpServletRequest` 对象

`HttpServletRequest` 对象可访问 HTTP 头数据,比如请求中可找到的任何 cookie 和产生该请求的 HTTP 方法。`HttpServletRequest` 对象也允许获得客户端作为请求的一部分

发送的参数。

#### 1) 获得客户端数据

`getParameter()`方法返回给出名字参数的值。如果参数值超过一个以上,用 `getParameterValues()`方法代替。`getParameterValues()`方法返回给出名字参数值的数组。(方法 `getParameterNames()`提供参数的名字。

对 HTTP GET 请求,`getQueryString()`方法以字符串形式返回来自客户端的未经处理的数据,您必须自己解析这个数据以获得参数和值。

对 HTTP POST,PUT 和 DELETE 请求:

如果期望的是文本数据,可用 `getReader()`方法返回的 `BufferedReader` 对象去读取未经处理的数据。

如果希望的是二进制数据,可用 `getInputStream()`方法返回的 `ServletInputStream` 对象去读取未经处理的数据。



注意:只能使用 `getParameter[Values]()`方法或者使用允许自己解析数据的方法中的一个,两者不能一起使用在同一个请求上。

## 2. HttpServletResponse 对象

`HttpServletResponse` 对象,提供了两种将数据返回给用户的方法。

`getWriter()`方法返回一个 `PrintWriter` 对象。

`getOutputStream()`方法返回一个 `ServletOutputStream` 对象。

`getWriter()`方法返回文本数据给用户,`getOutputStream()`方法返回二进制数据给用户。

发送响应之后关闭 `PrintWriter` 对象或 `ServletOutputStream` 对象,可以使服务器知道响应是什么时候结束的。

## 3. HTTP 头部数据

访问 `PrintWriter` 对象或 `ServletOutputStream` 对象之前,必须设置 HTTP 头部数据。`HttpServletResponse` 类提供了访问头部数据的方法。例如,`setContentType()`方法设置响应内容类型。(这个头部通常只能手工设置)

### 6.3.2 处理 GET 和 POST 请求

`service()`方法分配 HTTP 请求给出下列方法:

- (1) `doGet`:处理 GET 和 HEAD 请求。
- (2) `doPost`:处理 POST 请求。
- (3) `doPut`:处理 PUT 请求。
- (4) `doDelete`:处理 DELETE 请求。

默认情况下,这些方法返回 `BAD_REQUEST(400)`错误。你的 Servlet 应当覆盖你希望实现的方法。本小节说明怎样实现处理大多数通常的 HTTP 请求的方法:GET 和

POST。

当 Servlet 接收到 OPTIONS 请求的时候,HttpServlet 的 service()方法也调用 doOptions()方法,接收到 TRACE 请求的时候,调用 doTrace()方法。DoOptions()的默认实现自动决定支持什么 HTTP 选项并返回相应的信息。DoTrace()的默认实现引发一个 trace (跟踪)请求发送的所有头部信息的响应。这些方法典型的不需要覆盖。

## 2. 处理 GET 和 POST 请求

如果希望 Servlet 处理 HTTP 请求,那么扩展 HttpServlet 类并覆盖 Servlet 支持的那些处理 HTTP 请求的 Servlet 方法。本节举例说明怎样处理 GET 和 POST 请求。处理这些请求的方法是 doGet()和 doPost()。

### 1) 处理 GET 请求

处理 GET 请求需要覆盖 doGet()方法,下例说明怎样实现。

例:

```
public class BookDetailServlet extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ...

        // set content-type header before accessing the Writer
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // then write the response
        out.println("<html>" + "<head><title>Book Description</title></head>" + ...);

        //Get the identifier of the book to display
        String bookId = request.getParameter("bookId");
        if (bookId != null) {
            // and the information about the book and print it
            ...
        }
        out.println("</body></html>");
        out.close();
    }
    ...
}
```

这个 Servlet 扩展 HttpServlet 类并覆盖 doGet()方法。

在 doGet()方法内,getParameter()方法获得该 Servlet 期望的参数。

为响应客户端,这个例子 doGet()方法使用通过 HttpServletResponse 界面的方法获得的 Writer 对象返回文本数据给客户端。在访问这个 Writer 对象之前,例子中设置了响应内容类型头部。doGet()方法结尾处,响应发送后,这个 Writer 对象被关闭。

## 2) 处理 POST 请求

处理 POST 请求需要覆盖 doPost()方法,下例说明怎样实现。

例:

```
public class ReceiptServlet extends HttpServlet {

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ...

        // set content type header before accessing the Writer
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // then write the response
        out.println("<html>" +
            "<head><title> Receipt </title>" +
            ...);
        out.println("<h3> Thank you for purchasing your books from us " +
            request.getParameter("cardname") +
            ...);
        out.close();
    }
    ...
}
```

这个 Servlet 扩展 HttpServlet 类并覆盖 doPost()方法。

doPost()方法内,getParameter()方法获得该 Servlet 期望的参数。

为响应客户端,这个例子 doPost()方法使用通过 HttpServletResponse 界面的方法获得的 Writer 对象返回文本数据给客户端。在访问这个 Writer 对象之前,例子设置内容类型头部。doPost()方法结尾处,响应发送后,这个 Writer 对象被关闭。

## 3. 线程问题

如果 Servlet 里的方法需要访问共享资源,那么必须:

(1) 同步访问共享资源。

或者

(2) 建立一个 Servlet 一次只处理一个客户端请求。

这里说明怎样实现第二种情况。

HTTP Servlet 典型的是并发的为多个客户端服务。如果 Servlet 里的方法需要访问共享资源,那么可以通过建立一个 Servlet 一次只处理一个客户端请求来处理并发问题。

要使 Servlet 一次只处理一个客户端请求,需要实现 SingleThreadModel 界面,并扩展自 HttpServlet 类。

实现 SingleThreadModel 界面不需要写任何额外的方法。仅仅需要声明 Servlet 实现这个界面,服务器会使 Servlet 每次只运行一个 service 方法。

下例中的 Servlet 接收用户名和信用卡号码,并感谢用户的订购。如果这个 Servlet 实际更新一个数据库,例如保存存货记录,那么数据库连接可能就是一个共享资源。Servlet 同步访问该资源或实现 SingleThreadModel 界面。如果 Servlet 实现 SingleThreadModel 界面,仅需改变上例中的一行。

例:

```
public class ReceiptServlet extends HttpServlet
implements SingleThreadModel {

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

#### 4. Servlet 的描述

除了处理 HTTP 客户请求,Servlet 也可被调用提供自身的描述信息。这里说明通过覆盖 getServletInfo()方法提供描述信息。

有些应用程序,例如 Java Web Server Administration Tool,从 Servlet 获得描述信息并显示它们。Servlet 的描述是一个字符串,可描述它的目的、作者、版本号或者 Servlet 作者认为重要的任何信息。

getServletInfo()方法返回这些信息,默认它返回 null。不是必须覆盖这个方法,当然应用程序也不会提供 Servlet 的描述信息。

例:

说明 BookStoreServlet 的描述:

```
public class BookStoreServlet extends HttpServlet {
    ...
    public String getServletInfo() {
        return "The BookStore Servlet returns the " +
            "main web page for YYLC's Bookstore.";
    }
}
```

## 6.4 Servlet 的生命周期

本节讨论 Servlet 生命周期内重要事件,并且说明怎样自定义 Servlet 初始化和关闭。

每个 Servlet 有相同的生命周期:

- (1)服务器加载并初始化一个 Servlet。
- (2)该 Servlet 处理零或更多的客户请求。
- (3)服务器卸载该 Servlet(有些服务器只在它们关闭的时候才做这一步)。

Servlet 的生命周期如图 6-8 所示。

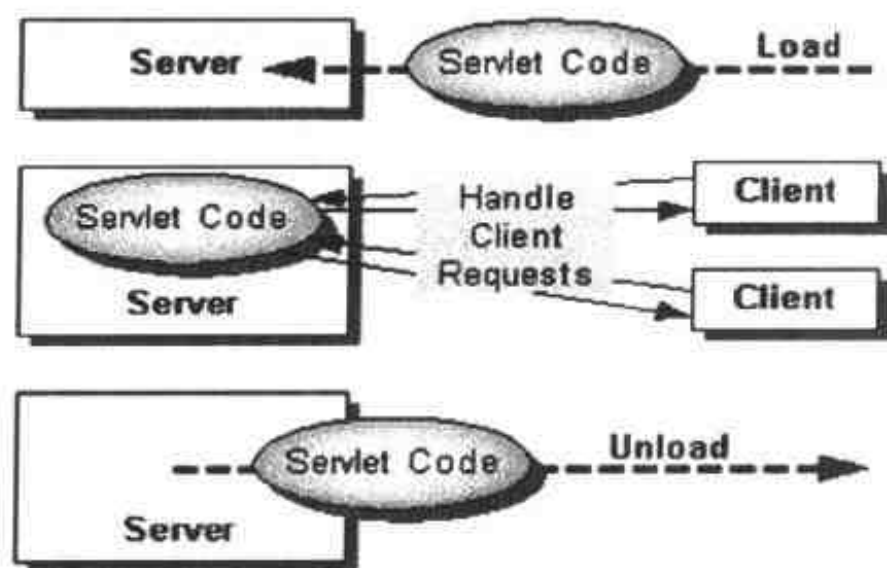


图 6-8 Servlet 生命周期

### 6.4.1 初始化 Servlet

当服务器加载 Servlet 时,服务器运行该 Servlet 的 `init()` 方法。初始化在客户请求被处理和该 Servlet 被销毁之前完成。

即使大多数 Servlet 运行在多线程服务器中,Servlet 初始化期间是没有并发问题的。当服务器加载 Servlet 时,服务器才调用 `init()` 方法一次。除非服务器重新加载该 Servlet,否则,`init()` 方法不会被再次调用。服务器不会重新加载 Servlet,直到服务器运行 `destroy()` 方法销毁该 Servlet 后。

`HttpServlet` 类的默认初始化方法初始化 Servlet 并将初始化过程做日志。为特殊初始化 Servlet,需要覆盖 Servlet 版本指定的 `init()` 方法。覆盖 `initL()` 方法时,需要遵守下列规则:

- (1) 如果初始化错误发生导致 Servlet 不能处理客户端请求,那么抛出 `UnavailableException` 异常。这种类型错误的最突出的一个问题就是不能建立一个必需的网络连接。
- (2) 不要调用 `System.exit` 方法。

例:



```

public class BookDetailServlet ... {

    public void init() throws ServletException {
        BookDBFrontEnd bookDBFrontEnd =
            (BookDBFrontEnd)getServletContext().getAttribute(
                "examples.bookstore.database.BookDBFrontEnd");

        if (bookDBFrontEnd == null) {
            getServletContext().setAttribute(
                "examples.bookstore.database.BookDBFrontEnd",
                BookDBFrontEnd.instance());
        }
        ...
    }
}

```

这个 `init()` 方法是简单明了的, 它试图获得一个属性, 如果属性还没有值, 它提供一个值。

Servlet 也会执行其它的初始化任务。如果 Servlet 使用数据库, 例如, `init()` 方法试图打开一个连接, 如果不成功那么抛出一个 `UnavailableException`。这里有一段 `init()` 方法连接数据库的示范代码:

例:

```

public class DBServlet ... {
    Connection connection = null;

    public void init() throws ServletException {
        // Open a database connection to prepare for requests
        try {
            databaseUrl = getInitParameter("databaseUrl");
            ...

            // get user and password parameters the same way
            connection = DriverManager.getConnection(databaseUrl, user, password);
        }
        catch (Exception e) {
            throw new UnavailableException (this,
                "Could not open a connection to the database");
        }
        ...
    }
}

```

## 2) 初始化参数

Servlet2.2 中 `init()` 方法可以调用 `getInitParameter()` 方法获得初始化参数。这个方法以初始化参数名作为自身的参数,返回表示初始化参数值的字符串。

初始化参数的规范是与特定服务器相关的。例如,Servlet 运行在 `servletrunner` 或 `JS-DK2.1` 服务器上时,参数由属性指定。

因为某种原因,需要获得参数名,那么请使用 `getParameterNames()` 方法。

## 6.4.2 与客户端交互

初始化之后,Servlet 就能处理客户请求了。如 6.3 节所述。

## 6.4.3 销毁 Servlet

Servlet 将一直运行直到服务器销毁它们。当服务器销毁 Servlet 时,服务器运行 Servlet 的 `destroy()` 方法。这个方法一旦运行,服务器不会再次运行 `destroy()` 方法直到服务器重新加载并重新初始化一个 Servlet 之后。

服务器调用 `destroy()` 方法时,另一个线程可能正运行一个 service 请求。本节将讨论当一个长时运行线程仍运行服务请求时怎样彻底的关闭它。

`HttpServlet` 类方法 `destroy()` 销毁 Servlet 并将销毁事件作一日志。为销毁 Servlet 占用的任意资源,那么覆盖 `destroy()` 方法。`destroy()` 可能重做一些初始化工作并且使当前内存状态与持久状态同步。

例:

这个例子是与 6.4.1 小节中 `init()` 方法示范代码相对应的 `destroy()` 方法的用法。

```
public class DBServlet ... {  
    Connection connection = null;  
  
    ... // the init method  
  
    public void destroy() {  
        // Close the connection and allow it to be garbage collected  
        connection.close();  
        connection = null;  
    }  
}
```

服务器在所有服务调用完成后,或者服务器指定的时间结束后调用 `destroy()` 方法。如果 Servlet 正处理一个长时运行操作,服务器调用 `destroy()` 方法时, `service()` 方法可能仍在运行。

当 `destroy()` 方法被调用时,前面说明的 `destroy()` 方法都认为所有的客户交互是已完成的,即 Servlet 没有长时运行操作。

### 6.4.4 Servlet 结束时处理 Service 线程

当 Servlet 被卸载时,该 Servlet 的所有 service()方法都应当是已完成的。服务器保证在所有服务请求都已返回后,或者服务器指定的跟踪时间结束后才调用 destroy()方法。如果 Servlet 有必要长时间运行,即操作运行时间比服务器跟踪时间更长,当调用 destroy()方法时,操作可能仍在运行中。必须保证任何线程仍然要将客户请求完成,本小节讲述这样的技术。

如果 Servlet 有潜在的长时运行服务请求,可使用下面的技术:跟踪有多少线程当前正运行 service()方法。为彻底关闭服务,使用 destroy()方法通知长时运行线程关闭并等待它们结束。使长时运行方法周期性的查询是否需要关闭,如果必要的话,停止工作、清理现场并返回。

#### 1. 跟踪服务请求

为跟踪服务请求,在 Servlet 类里设置一个域以统计正在运行的 service()方法的数量,可称为服务计数器。这个域应当有增加、减少和返回其值的方法。因为多个线程访问这个域,所以应当保证访问同步。Destroy()方法将等待直到这个域的值 0。这个私有域对象可叫作 lock,它是我们需要线程同步的对象。

例:

```
public ShutdownExample extends HttpServlet {
    private int serviceCounter = 0;
    private Object lock = new Object();
    ...
    //Access methods for serviceCounter
    protected void enteringServiceMethod() {
        synchronized(lock) {
            serviceCounter++;
        }
    }

    protected void leavingServiceMethod() {
        synchronized(lock) {
            serviceCounter--;
            if (serviceCounter == 0 && isShuttingDown())
                notifyAll();
        }
    }

    protected int numServices() {
```

```

        synchronized(lock) {
            return serviceCounter;
        }
    }
}

```

`service()` 方法进入时, 增加该数值, `service` 方法返回时, 减少该数值。这是为数不多的几个 `HttpServlet` 子类需要覆盖 `service` 方法中的一个。新的服务方法将调用 `super.service`, 以保持所有原始 `HttpServlet.service` 方法的功能。

例:

```

protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException
{
    enteringServiceMethod();
    try {
        super.service(req, resp);
    }
    finally {
        leavingServiceMethod();
    }
}

```

## 2. 彻底关闭

为彻底关闭。`destroy()` 方法不应该销毁所有的共享资源, 直到所有的服务请求都已完成。其中一部分工作就是检查服务计数器。另一部分工作是通知长时运行方法应该关闭。这样, 需要另一个域及其一般的访问方法。

例:

```

public ShutdownExample extends HttpServlet {
    private boolean shuttingDown;
    ...
    //Access methods for shuttingDown
    protected void setShuttingDown(boolean flag) {
        shuttingDown = flag;
    }
    protected boolean isShuttingDown() {
        return shuttingDown;
    }
}

```

例:

`destroy()` 方法使用这些域提供一个彻底关闭:

```

public void destroy() {

    synchronized(lock) {
        // Check to see whether there are still service methods running,
        //and if there are, tell them to stop.
        if (numServices() > 0) {
            setShuttingDown(true);
        }

        // Wait for the all of the service methods to stop.
        while(numServices() > 0) {
            try {
                wait();
            }
            catch (InterruptedException e) {
                ...
            }
        }
    }
}

```

### 3. 创建规范的长时运行方法

彻底关闭的最后一步是使任何长时运行方法行为规范。可能需要长时间运行的方法应当检查用来通知它们关闭的那些域的值,如果必要的话,中断自身的工作。

例:

```

public void doPost(...) {
    ...
    for(i = 0; ((i < lotsOfStuffToDo) && ! isShuttingDown()); i++) {
        try {
            partOfLongRunningOperation(i);
        }
        catch (InterruptedException e) {
            ...
        }
    }
}

```

## 6.5 存储客户端状态

Servlet API 提供了两种记录客户端状态的方法。

### 6.5.1 Session 跟踪

Session 跟踪是一段时间内 Servlet 用来维持一连串请求状态的机制,这些请求来自同一用户,即请求源自同一浏览器。

Session 在客户端访问是所有 Servlet 之间共享的,这便于应用程序由多个 Servlet 组成。例如,YYLC's BookStore 使用 Session 跟踪保持用户订购书的记录。例子中的所有 Servlet 都可访问用户的 Session。

使用 Session 跟踪步骤如:

- (1) 获得 Session(一个 HttpSession 对象)。
- (2) 存储或获得来自 HttpSession 对象的数据。
- (3) 使该 Session 失效(可选)。

#### 1. 获得一个 Session

HttpServletRequest 对象的 getSession()方法返回用户的 Session。调用该方法并且指定它的 creat 参数为“true”,如果需要的话,Servlet 实例会建立一个 Session。

为正确的维持 Session,必须在任何输出写入响应前调用 getSession()方法。如果使用 Writer 对象响应,那么必须在访问 Writer 对象前调用 getSession(),在这之前不能发送任何响应数据。

例子 YYLC's Bookstore 使用 Session 保持用户购物车内书的记录。下例是 CatalogServlet 获得用户的 Session。

例:

```
public class CatalogServlet extends HttpServlet {
    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Get the user's session and shopping cart
        HttpSession session = request.getSession(true);
        ...
        out = response.getWriter();
        ...
    }
}
```

## 2. 存储与获得数据

HttpSession 界面提供存储和获取数据的方法,可存储的数据类型如下:

(1) 标准的 Session 属性,例如一个 Session 标识符。

(2) 应用程序数据,以“名字/值”对形式存储。名字是字符串,值是 Java 程序设计语言中的对象。因为允许多个 Servlet 访问用户的 Session,所以应当采用一种命名约定以组织与应用程序数据相关的名字,这可避免多个 Servlet 之间偶然复写彼此在 Session 中的值。一种约定是 Servletname.name,其中 Servletname 是 Servlet 的完整名字,包括它的包名。例如,com.acme.WidgetServlet.state 是一个 Servletname 为 com.acme.WidgetServlet,name 为 state 的 Session。

例子 YYLC's Bookstore 使用 Session 跟踪保持用户购物车内书的记录。下例是 CatalogServlet 获得用户的 Session 标识符,并且获得和设置与该用户的 Session 相关的应用程序数据。

例:

```
public class CatalogServlet extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Get the user's session and shopping cart
        HttpSession session = request.getSession(true);
        ShoppingCart cart =
            (ShoppingCart)session.getAttribute(session.getId());

        // If the user has no cart, create a new one
        if (cart == null) {
            cart = new ShoppingCart();
            session.setAttribute(session.getId(), cart);
        }
        ...
    }
}
```

因为对象可与 Session 绑定,例子 YYLC's Bookstore 将用户已订购书的记录保存在一个对象里。对象类型是 ShoppingCart,用户订购的每一本书作为 ShoppingCartItem 对象被存储于购物车里。

例:

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

```

HttpSession session = request.getSession(true);
ShoppingCart cart = (ShoppingCart)session.getValue(session.getId());
...

// Check for pending adds to the shopping cart
String bookId = request.getParameter("Buy");

//If the user wants to add a book, add it and print the result
String bookToAdd = request.getParameter("Buy");
if (bookToAdd != null) {
    BookDetails book = database.getBookDetails(bookToAdd);
    cart.add(bookToAdd, book);
    out.println("<p><h3>" + ...);
}
}

```

最后,注意 Session 可被指定为 new。新的 Session 将使 HttpSession 界面的 isNew() 方法返回 true。例如,客户端还不知道该 Session。新的 Session 没有绑定的数据。

这里,必须处理包含新的 Session 的情况,在上例 YYLC's Bookstore 中,如果用户没有购物车(只有可以绑定到 Session 的数据),Servlet 应创建一个新的购物车。如果需要来自用户的信息(例如用户名)以创建一个新的 Session,可以重定向用户到收集信息的开始页面。

### 3. 使 Session 失效

用户可以使用手动或自动的方法使 Session 失效。这取决于 Servlet 运行的状态,例如,某个时间段内没有页面的请求到来,通常服务器自动使 Session 失效,一般默认时间是 30 分钟。使一个 Session 失效意味着从系统中删除 HttpSession 类对象和它的值。

手动使 Session 失效,使用 Session 的 invalidate() 方法。例子 YYLC's Bookstore 中,用户已经购买了书之后,可使用户的 Session 失效。这发生在 ReceiptServlet 中。

例:

```

public class ReceiptServlet extends HttpServlet {

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {

        ...

        scart = (ShoppingCart)session.getValue(session.getId());
        ...

        // Clear out shopping cart by invalidating the session
        session.invalidate();
    }
}

```



```

        // set content type header before accessing the Writer
        response.setContentType("text/html");
        out = response.getWriter();
        ...
    }
}

```

#### 4. 处理所有浏览器

默认 Session 跟踪使用 Cookie 将 Session 标识符与用户联系起来。为了使浏览器不支持 cookie 或者拒绝使用 Cookie 的用户仍能访问 Servlet, 必须使用 URL 重写技术代替。

当使用 URL 重写时, 如果需要, 可调用方法在超链接里包含 Session ID。必须为这个 Servlet 响应内的每个超链接调用这些方法。

Servlet2.1 以前, 将 Session ID 与 URL 联系起来是 `HttpServletResponse.encodeUrl()` 方法, Servlet2.1 以来是 `HttpServletResponse.encodeURL()` 方法。如果重定向用户到另外一个页面, Servlet2.1 以前中调用 `HttpServletResponse.encodeRedirectUrl()` 方法将 Session ID 与重指向的 URL 联系起来, Servlet2.1 及以后中是 `HttpServletResponse.encodeRedirectURL()`。

URL 编码和编码重定向方法决定 URL 是否需要被重写, 据此返回经改变的或未经改变的 URL。(URL 和重定向 URL 的重写规则是不一样的, 但是通常如果服务器检测到浏览器支持 Cookie, 那么就不重写 URL)

例

如果例子 YYLC's Bookstore 使用 URL 重写, 代码如下:

```

public class CatalogServlet extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Get the user's session and shopping cart, the Writer, etc.
        ...

        // then write the data of the response
        out.println("<html>" + ...);
        ...

        // Get the catalog and send it, nicely formatted
        BookDetails[] books = database.getBooksSortedByTitle();
        ...

        for(int i=0; i < numBooks; i++) {
            ...

            //Print out info on each book in its own two rows

```

```

        out.println("<tr>" + ...

        "<a href = \"\" +
            response.encodeURL("/servlet/bookdetails? bookId=" + bookId) +
            "\"> <strong>" + books[i].getTitle() +
            "</strong></a></td>" + ...

        "<a href = \"\" +
            response.encodeURL("/servlet/catalog? Buy=" + bookId)
            + "\"> Add to Cart </a></td></tr>" + ...
    }
}
}

```

CatalogServlet 对每本书返回两个链接给用户。一个链接提供有关书的细节,另一个允许添加书到购物车。两个 URL 都将被重写。因为,一旦使用 URL 重写,那么 Servlet 返回给用户的每一个链接都必须被重写。如果用户点击一个重写 URL 的链接,Servlet 识别并析取出 Session ID,然后 getSession()方法使用析取出的 Session ID 获得用户的 HttpSession 对象。

另一方面,如果浏览器不支持 Cookie,并且用户点击一个非重写 URL,Session 将丢失。通过超链接联系的 Servlet 将建立一个新的 Session,但是新的 Session 没有与先前 Session 相关联(绑定到先前 Session)的数据。因此,一旦 Servlet 丢失 Session 内的数据,对所有共享该 Session 的 Servlet 来说,数据都会丢失。如果 Servlet 要支持那些不支持或不接受 cookie 的客户端,那么应当自始至终的使用 URL 重写。

## 6.5.2 Cookies

Cookies 是 Servlet 让客户端保存少量与用户相关的状态信息的机制。Servlet 可用 Cookie 中的信息作为用户登录站点(例如,低安全级用户)的凭证,或者作为站点内的导航(例如,作为用户参数的仓库)。

Cookie 为服务器发送某些信息到客户端存储提供了一条途径,并且稍后服务器可从那个客户端重新获得该信息。Servlet 通过添加字段到 HTTP 响应头部来发送 Cookie 给客户端。

每个 HTTP 请求和响应头部都必须命名并且只能是单值。例如,一个 Cookie 可能是一个名为 BookToBuy,值为 304qty1 的头部,表明调用这个应用程序的用户想购买一本库存号为 304 的书。

多个 Cookie 可有相同的名字。例如,一个 Servlet 可发送两个头部名为 BookToBuy 的 Cookie。其中一个头部值为 304qty1,表明用户想买一本库存号为 304 的书;另一个头部值为 301qty3,表明用户想买 3 本库存号为 301 的书。

另外,名字和值都可以提供可选属性,例如注释。现在的 Web 浏览器几乎不能正确处理可选属性,所以不能依赖它们。

服务器可提供一个或更多的 Cookie 给一个客户端。客户端软件,例如一个 Web 浏览器,希望它对每个主机能支持 20 个 Cookie,每个 Cookie 至少 4KB(4kilobytes)。

当你发送 Cookie 到服务端时,标准的 HTTP/1.0 缓冲存储器不会缓存页面。现在, `javax.servlet.http.Cookie` 不支持 HTTP/1.1 缓存控制模式。

客户端为服务器存储的 Cookie 将由客户端返回给那个服务器并且只能是那个服务器。一个服务器可包含多个 Servlet。例子 YYLC's Bookstore 由多个运行在一个服务器的 Servlet 组成。本小节举例说明 `CatalogServlet` 和 `ShowCart Servlet` 使用同一个 Cookie。

为发送一个 Cookie 需要:

▶实例化一个 Cookie 对象。

▶设置任意属性。

▶发送该 Cookie。

从 Cookie 获得信息需要:

▶获得来自用户请求的所有 Cookie。

▶以标准的程序设计技巧通过名字找到你感兴趣的某个 Cookie 和几个 Cookie。

▶获得你查找到 Cookie 的值。

## 1. 建立一个 Cookie

类 `javax.servlet.http.Cookie` 的构造器以初始的名字和值建立一个 Cookie,稍后可以使用 `setValue()` 方法改变该 Cookie 的值。

Cookie 的名字必须是 HTTP/1.1 可接受的字母数字串。



注意:以美元字符("\$")开始的名字将被保留。

Cookie 的值可以是任意字符串,然而 null 值,不能保证所有浏览器以相同的方式运作。另外,如果遵循 Netscape 的原始 Cookie 规范发送 Cookie,不要使用空白和任何下列字符:

[ ] ( ) = , " / ? . : ;

如果 `CatalogServlet` 使用 Cookie 保持客户订购书的记录,可像这样建立 Cookie。

例:

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    // Check for pending adds to the shopping cart
    String bookId = request.getParameter("Buy");

    //If the user wants to add a book, remember it by adding a cookie
    if (bookId != null) {
        Cookie getBook = new Cookie("Buy", bookId);
```

```

        ...
    }

    // set content-type header before accessing the Writer
    response.setContentType("text/html");

    // now get the writer and write the data of the response
    PrintWriter out = response.getWriter();
    out.println("<html>" +
        "<head><title> Book Catalog </title></head>" + ...);
    ...
}

```

## 2. 设置 Cookie 的属性

类 `Cookie` 提供多种方法设置 `Cookie` 的值和属性。下例设置 `CatalogServlet`'s cookie 的解释字段。解释字段描述了这个 `Cookie` 的目的。

例：

```

public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    ...

    //If the user wants to add a book, remember it by adding a cookie
    if (values != null) {
        bookId = values[0];
        Cookie getBook = new Cookie("Buy", bookId);
        getBook.setComment("User wants to buy this book from the bookstore.");
    }
    ...
}

```

还可以设置 `Cookie` 的最大生命周期。这个属性是非常有用的，例如为删除一个 `Cookie`。如果 `YYLC's Bookstore` 使用 `Cookie` 保持用户的订购记录，可使用这个属性从用户订购中删除一本书。用户从购物车中删除一本书，`ShowCartServlet` 的代码如下。

例：

```

public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    ...

    /* Handle any pending deletes from the shopping cart */
    String bookId = request.getParameter("Remove");
}

```

```

...
if (bookId != null) {
    // Find the cookie that pertains to the book to remove
    ...
    // Delete the cookie by setting its maximum age to zero
    thisCookie.setMaxAge(0);
    ...
}

// also set content type header before accessing the Writer
response.setContentType("text/html");
PrintWriter out = response.getWriter();

//Print out the response
out.println("<html> <head>" + "<title> Your Shopping Cart</title>" + ...);

```

### 3. 发送 Cookie

Cookie 作为响应的头部发送给客户端,使用 `HttpServletResponse` 类的 `addCookie()` 方法把它们加入响应。如果正使用 `Writer` 对象发送文本数据给客户端,那么调用 `HttpServletResponse` 的 `getWriter()` 方法之前必须调用 `addCookie()` 方法。继续 `CatalogServlet` 这个例子,下面的代码发送一个 Cookie。

例:

```

public void doGet (HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    ...
    //If the user wants to add a book, remember it by adding a cookie
    if (values != null) {
        bookId = values[0];
        Cookie getBook = new Cookie("Buy", bookId);
        getBook.setComment("User has indicated a desire " +
            "to buy this book from the bookstore.");
        response.addCookie(getBook);
    }
    ...
}

```

### 4. 获得 Cookie

客户端将 Cookie 加入 HTTP 请求头部的一个字段返回服务端。要获得任意的 Cook-

ie, 必须使用 `HttpServletRequest` 类的 `getCookies()` 方法获得所有的 Cookie。

`GetCookies()` 方法返回 `Cookie` 对象的一个数组, 可以搜索找到想要的某个 Cookie 或几个 Cookie。(记住多个 Cookie 可以有相同的名字。为获得某个 Cookie 的名字, 使用它的 `getName()` 方法)。继续例子 `ShowCartServlet`。

例:

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    ...

    /* Handle any pending deletes from the shopping cart */
    String bookId = request.getParameter("Remove");
    ...

    if (bookId != null) {
        // Find the cookie that pertains to the book to remove
        Cookie[] cookies = request.getCookies();
        ...

        // Delete the book's cookie by setting its maximum age to zero
        thisCookie.setMaxAge(0);
    }

    // also set content type header before accessing the Writer
    response.setContentType("text/html");
    PrintWriter out = response.getWriter();

    //Print out the response
    out.println("<html> <head>" + "<title>Your Shopping Cart</title>" + ...);
}
```

## 5. 获得 Cookie 的值

为获得某个 Cookie 的值, 可使用它的 `getValue()` 方法。继续例子 `ShowCartServlet`。

例:

```
public void doGet (HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException
{
    ...

    /* Handle any pending deletes from the shopping cart */
    String bookId = request.getParameter("Remove");
    ...
}
```

```
if (bookId != null) {
    // Find the cookie that pertains to that book
    Cookie[] cookies = request.getCookies();
    for(i=0; i < cookies.length; i++) {
        Cookie thisCookie = cookies[i];
        if (thisCookie.getName().equals("Buy") &&
            thisCookie.getValue().equals(bookId)) {

            // Delete the cookie by setting its maximum age to zero
            thisCookie.setMaxAge(0);
        }
    }
}

// also set content type header before accessing the Writer
response.setContentType("text/html");
PrintWriter out = response.getWriter();

//Print out the response
out.println("<html> <head>" + "<title>Your Shopping Cart</title>" + ...);
```

## 6.6 Servlet 的通信

为满足客户端请求,Servlet 有时需要访问网络资源如其它的 Servlet,HTML 页面,同一服务器上几个 Servlet 间的共享对象,等等。

如果 Servlet 需要网络资源,例如运行在不同服务器上的一个 Servlet,那么可生成一个对其它资源的 HTTP 请求。本节将讲述如果 Servlet 需要来自自身服务器的一个有效资源,应该怎样做。

### 6.6.1 通过 RequestDispatcher 对象使用服务器上的其它资源

要使 Servlet 能访问其它资源如其它 Servlet,JSP 页面或 CGI 脚本,可以使 Servlet 生成一个 HTTP 请求(这是基本的 Java 程序设计技巧)。如果资源对运行 Servlet 的服务器是有效的,那么可使用 RequestDispatcher 对象生成对资源的请求,步骤如下:

- ① 获得绑定到某个资源的 RequestDispatcher 对象。
- ② 转发客户端请求给该资源,让它对用户的请求作出应答。
- ③ 在 Servlet 的输出中包含该资源的响应。

## 1. 获得一个 RequestDispatcher 对象

为获得一个 RequestDispatcher 对象,使用 ServletContext 对象的 getRequestDispatcher()方法。这个方法把被请求的资源 URL 作为自己的参数。这个参数的格式是一个斜线("/")后跟一个或更多的斜线分隔的目录名,最后以资源的名称结束。下列是有效 URL 的例子:

```
/servlet/myservlet
/servlet/tests/MyServlet.class
/myinfo.html
```

BookStoreServlet 获取 YYLC's Bookstore 主页的一个 RequestDispatcher 对象。

例:

```
public class BookStoreServlet extends HttpServlet {

    public void service (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Get the dispatcher; it gets the main page to the user
        RequestDispatcher dispatcher = getServletContext().getRequestDispatcher(
            "/examples/applications/bookstore/bookstore.html");
        ...
    }
}
```

URL 指向的资源对运行这个 Servlet 的服务器必须是有效的。如果资源是无效的,或服务器没有实现资源所属类型的 RequestDispatcher 对象,那么这个方法返回 null。Servlet 应准备处理这种情况。BookStoreServlet 以下述方式处理这种情况。

例:

```
public class BookStoreServlet extends HttpServlet {

    public void service (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        // Get the dispatcher; it gets the main page to the user
        RequestDispatcher dispatcher = ...;

        if (dispatcher == null) {
            // No dispatcher means the html file can not be delivered
            response.sendError(response.SC_NO_CONTENT);
        }
        ...
    }
}
```



```

}
}

```

## 2. 转发请求

一旦有了 `RequestDispatcher` 对象,可以分配给它相关的资源,让它负责响应客户的请求。转发是有用的,例如 `Servlet` 处理请求,然后产生非特殊的响应,这时请求可被继续向后传递给另一个资源。如用户订购的时候,某个 `Servlet` 可处理信用卡信息,然后将客户请求继续向后传递给另一个资源以产生一个“Thank you”的页面。例子 `BookStoreServlet` 获得用户的 `Session`,然后使请求分配器返回 `YYLC's Bookstore` 的开始页面。

例:

```

public class BookStoreServlet extends HttpServlet {

    public void service (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        // Get or start a new session for this user
        HttpSession session = request.getSession();
        // Send the user the bookstore's opening page
        dispatcher.forward(request, response);
        ...
    }
}

```

记住:`forward` 方法用来将响应用户的责任转交给另一个资源。如果已经访问 `ServletOutputStream` 或 `PrintWriter` 对象,就不能使用这个方法。它将抛出 `IllegalStateException` 异常。

如果已经开始访问 `PrintWriter` 或 `ServletOutputStream` 以响应用户,必须使用 `include` 方法代替。

## 3. 包含请求

`RequestDispatcher` 界面的 `include` 方法提供响应客户端的能力给调用的 `Servlet`,但是 `RequestDispatcher` 对象关联的资源只作响应的一部分。

`Servlet` 调用 `RequestDispatcher.include()` 方法,同时也希望能够响应客户端,`Servlet` 调用 `include` 方法之前和之后都可以使用 `PrintWriter` 和 `ServletOutputStream` 对象。必须记住被调用的资源不能设置响应的头部。如果资源试图设置头部,那么设置有可能失败。

下例说明 `ReceiptServlet` 可以看到的內容。

例:

首先是谢谢用户订购,然后在输出中包含订购一览。

```

public class ReceiptServlet extends HttpServlet {

```

```
public void doPut(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
{
    // Process a customer's order
    ...
    // Thank the customer for the order
    res.setContentType("text/html");
    PrintWriter toClient = res.getWriter();
    ...
    toClient.println("Thank you for your order!");

    // Get the request-dispatcher, to send an order-summary to the client
    RequestDispatcher summary =
        getServletContext().getRequestDispatcher("/OrderSummary");

    // Have the servlet summarize the order; skip summary on error.
    if (summary != null)
        try {
            summary.include(req, res);
        }
        catch (IOException e) {
        }
        catch (ServletException e) {
        }
        toClient.println("Come back soon!");
        toClient.println("</html>");
        toClient.close();
    }
}
```

### 6.6.2 在 Servlet 间共享资源

运行在同一服务器上的多个 Servlet 之间有时需要共享资源。特别是编译一个应用程序中获得的多个 Servlet 之间,这种需要尤为重要。如同 YYLC's Bookstore Servlet,同一服务器上的 Servlet 可使用 ServletContext 界面中操纵属性的方法(setAttribute(),getAttribute(),removeAttribute())共享资源。

## 1. 属性的命名约定

一个上下文中的所有 Servlet 可共享以 ServletContext 界面存储的属性。为避免属性名冲突,属性名使用与包命名相同的约定。例如,YYLC's Bookstore Servlet 共享一个名叫 examples.bookstore.database.BookDBFrontEnd 的属性。



注意:以前缀 java. \*, javax. \* 和 sun. \* 开始的名字被保留。

## 2. setAttribute

Servlet 使用 ServletContext.setAttribute() 方法设置属性。当有多个 Servlet 共享一个属性的时候,可能需要为每个 Servlet 初始化该属性。如果是这样,每个 Servlet 都将检查属性值,如果另外的 Servlet 还没有初始化该属性,那么只设置这个 Servlet 的属性。

例:

下例说明 CatalogServlet 的 init() 方法为 YYLC's Bookstore 设置共享属性。

```
public class CatalogServlet extends HttpServlet {

    public void init() throws ServletException {
        BookDBFrontEnd bookDBFrontEnd = ...

        if (bookDBFrontEnd == null) {
            getServletContext().setAttribute(
                "examples.bookstore.database.BookDBFrontEnd",
                BookDBFrontEnd.instance());
        }
    }
    ...
}
```

一旦一个 Servlet 设置了属性,在相同上下文中的任何 Servlet 都能获得它的值,重新设置它的值或删除该属性。

## 3. GetAttribute

获得属性的值就是调用方法 ServletContext.getAttribute() 这么简单。

例:

下例说明 CatalogServlet 在初始化时获得属性的值。

```
public class CatalogServlet extends HttpServlet {

    public void init() throws ServletException {
        BookDBFrontEnd bookDBFrontEnd =
            (BookDBFrontEnd) getServletContext().getAttribute(
```

```

        "examples.bookstore.database.BookDBFrontEnd");

    if (bookDBFrontEnd == null) {
        getServletContext().setAttribute(
            "examples.bookstore.database.BookDBFrontEnd",
            BookDBFrontEnd.instance());
    }
}
...
}

```

#### 4.removeAttribute

任何 Servlet 可从 ServletContext 对象中删除属性。然而,因为属性是共享的,所以必须小心不要删除另外的 Servlet 可能仍在使用的属性。YYLC's Bookstore 没有使用到 removeAttribute() 方法。

### 6.6.3 从 Servlet 中调用其它 Servlet

为使 Servlet 调用其它 Servlet,可以生成一个对另一个 Servlet 的 HTTP 请求,或以直接调用另一个 Servlet 的公有方法(如果两个 Servlet 运行在同一服务器上)。

本节讨论第二种情况。为直接调用另一个 Servlet 的公有类,必须知道想调用的 Servlet 的名字,获得访问哪个 Servlet(一个实例)的 Servlet(类)对象(另一个实例);调用后者(另一个 Servlet 实例)的公有方法。

为获得 Servlet 对象,使用 ServletContext 对象的 getServlet 方法,其中 ServletContext 对象又从 ServletConfig 对象获得。下例讲解当 BookDetail Servlet 调用 BookDB Servlet, BookDetail Servlet 如何获得 BookDB Servlet 的 Servlet 对象。

例:

```

public class BookDetailServlet extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ...

        BookDBServlet database = (BookDBServlet)
            getServletConfig().getServletContext().getServlet("bookdb");
        ...
    }
}

```

一旦获得 Servlet 对象,就可以调用这个 Servlet 的任何公有方法。例如,BookDetail Servlet 调用 BookDB Servlet 的 getBookDetails 方法。

例:

```
public class BookDetailServlet extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ...

        BookDBServlet database = (BookDBServlet)
            getServletConfig().getServletContext().getServlet("bookdb");
        BookDetails bd = database.getBookDetails(bookId);
        ...
    }
}
```

当调用另一个 Servlet 的方法时,必须小心。如果想调用的 Servlet 实现了 SingleThreadModel 界面,调用就会干扰被调用的 Servlet 的单线程特性。在这种情况下,Servlet 应当生成一个对其它 Servlet 的 HTTP 请求以代替直接调用其它 Servlet 的方法。

## 6.7 运行 Servlet

本节说明怎样从浏览器和 HTML 页面内调用 Servlet。

### 6.7.1 在浏览器地址栏中直接键入 Servlet 的 URL

Servlet 可通过键入它们的 URL 到浏览器的地址栏而被直接调用。这是访问例子 YYLC's Bookstore 的主页的方法。这一节说明 Servlet URL 的一般形式。

Servlet 的 URL 有如下形式,其中 Servletname 对应 Servlet 的名字。http://machine-name:port/servlet/Servletname,例如,提交 YYLC's Bookstore 主页的 Servlet 有属性 servlet.bookstore.code = BookStoreServlet。要查看例子的主页,在浏览器中键入如下 URL:

http://localhost:8080/servlet/bookstore

Servlet URL 可包含查询,如 HTTP GET 请求。例如,显示特定的某本书的详细资料的 Servlet,以书的库存号作为一个查询。Servlet 的名字是 bookdetails,Servlet 的 URL 为:

http://localhost:8080/servlet/bookdetails? bookId = 203

### 6.7.2 从 HTML 页面调用 Servlet

Servlet URL 可用在 HTML 表示 CGI-bin 或文件 URL 的标记里。本节说明 Servlet

URL 用作锚的目的地址, 表单行为对象, Meta 标记指引的是被刷新页面。

从一个 HTML 页面中调用一个 Servlet 仅仅是在适当的 HTML 标记中使用 Servlet 的 URL。能带 URL 的标记包括锚、表单和 Meta。

本节使用例子 YYLC's Bookstore 的 ShowCart, Cashier 和 Receipt Servlet。幸运的是当您查看您的购物车, 购买书的时候, Servlet 是顺序调用的。

为直接访问 ShowCart Servlet, 从 YYLC's Bookstore 的主页点击 Show Cart 超链接。

## 1. HTML 标记中的 Servlet URL

ShowCartServlet 返回的页面有大量的锚, 每个锚的目的都是一个 Servlet。下例显示这些锚中的一个。

例:

```
public class ShowCartServlet extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ...

        out.println( ... + "<a href = \" " +
            response.encodeUrl("/servlet/cashier") + "\">Check Out</a> " + ... );
        ...
    }
    ...
}
```

这段代码在 HTML 页面上结果是下面的锚:

```
<a href = "http://localhost:8080/servlet/cashier">Check Out</a>
```

如果 showcart Servlet 的页面显示在浏览器上, 查看页面源代码的话, 可以看到这个锚, 然后单击这个链接, cashier Servlet 将返回包含下一个例子的页面。cashier Servlet 显示的页面是一个询问用户名字和信用卡的表单。输出这个表单标记的代码。

例:

```
public class CashierServlet extends HttpServlet {

    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ...

        out.println( ... + "<form action = \" " +
            response.encodeUrl("/servlet/receipt") +
            "\" method = \"post\"> " +
            ...
    }
}
```

```

        "<td><input type = \"text\" name = \"cardname\" \" +
        "value = \"Gwen Canigetit\" size = \"19\"></td>" +
        ...
        "<td><input type = \"submit\" \" +
        "value = \"Submit Information\"></td>" +
        ...
        "</form>" +
        ...);
    out.close();
}
...
}

```

这段代码在页面上的结果是如下一个表单标记：

```
<form action = "http://localhost:8080/servlet/receipt" method = "post">
```

如果 cashier Servlet 的页面显示在您的浏览器上,查看这个页面的源代码,可以看到开始一个表单的标记。然后提交这个表单, receipt Servlet 将返回包含下一个例子的页面。

由 receipt Servlet 返回的页面有一个 meta 标记,它使用一个 Servlet URL 作为其 http-equiv 属性值的一部分。这个标记指引页面到 YYLC's Bookstore 的主页。下面显示产生这个标记的代码。

例：

```

public class ReceiptServlet extends HttpServlet {

    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        ...
        out.println("<html>" +
            "<head><title> Receipt </title>" +
            "<meta http-equiv = \"refresh\" content = \"4; url = \" +
            "http://\" + request.getHeader(\"Host\") +
            "/servlet/bookstore; \">" +
            "</head>" +
            ...
        }
    }
}

```

这段代码在 HTML 页面上的结果有如下代码：

```
<meta http-equiv = "refresh" content = "4; url = http://localhost:8080/servlet/bookstore;"
```

## 6.8 本章小结

本章开始就是一个 Tutorial,帮助读者尽快上手,然后详细讲述了编写 Servlet 整个过程的详细内容,它们是 Request、Response 处理客户端请求,Session、Cookie 维持客户端状态、RequestDispatcher 分配客户端请求,以及怎样运行 Servlet 等。本章还讲述了一个非常重要的知识点 Servlet 的生命周期。需要注意的是这儿给出的代码并非可以直接执行的代码,即伪代码。YYLC 书屋的具体实现请参考第 14 章,请注意这里给出的伪代码也并非与后面的实现完全一样,在命名和一些细节上是有出入的,例如 YYLC 的实际实现关于书的数据采用的是数据库保存。



# 第 7 章 内部对象

## 7.1 内部对象概述

本章讲述如何使用 JSP 提供的内部对象与客户机进行交互,开发动态的 WEB 页面。

### 7.1.1 内部对象的功能

熟悉 ASP 的读者应该对内部对象不会陌生,运用它们可以不必了解内部复杂的交互机制,就程序设计面言这是一大优点,但同时编程人员来说也是一种束缚。JSP 共提供了 9 个内部对象,它们是: Request、Response、Out、Session、Application、Config、PageContext、Page 和 Exception。在详细讲述它们之前,来了解一下可以利用它们来做的事:

(1) 使用内部 Out 对象输出响应到页面,该对象功能比表达式强得多。如:

```
<% out.println("Hello World"); %>
```

(2) 使用内部 Request 对象获取参数的值、Session、Cookie。如:

```
<% String name=request.getParameter("name"); %>
```

(3) 使用内部 Session、Cookie 对象维持客户端状态。如:

```
<% Object ob=session.getAttribute(session.getId()); %>
```

(4) 使用内部 Application 对象提供共享信息。如:

```
<% Object ob=application.getAttribute(session.getId()); %>
```

上面几个简单例子只是内部对象强大功能之冰山一角。JSP9 个内部对象功能描述如表 7-1 所示:

表 7-1 java Server page 内部对象功能描述

对象名称	功能描述
Request	一个 HttpServletRequest 对象。它封装了浏览器的请求信息,并且提供了获取 Cookie、Header 和 Session 等对象、数据的方法。
Response	一个 HttpServletResponse 对象,提供数个方法设置返回浏览器的响应(如 Cookies,Header 等)。
Out	JspWriter 的一个实例,并提供大量方法能轻松将响应输出到浏览器。
Session	提供服务器——客户端之间的一种联系,也就是通常所说的会话。



注意:不要将作用域与对象混淆了,例如 session 作用域与 Session 对象,application 作用域与 Application 对象。

JSP 各内部对象及作用域如表 7-2 所示。

表 7-2 Jsp 各内部对象的作用域

内部对象	作用域
Request	request
Response	page
PageContext	page
Session	session
Application	application
Out	page
Config	page
Page	page

## 7.2 JSP 内部对象详解

### 7.2.1 Request 对象

JSP 服务器接收到客户端请求时,可利用 Request 对象取得客户端信息。那么 Request 对象怎样取得这些信息呢?弄清这一点也就掌握了 Request 对象。Request 对象是 HttpServletRequest 接口的一个实例,因此它可使用 HttpServletRequest 接口的所有方法。在第 6 章中已涉及到 HttpServletRequest 接口的一些方法,Request 对象利用这些方法,可以很容易的从客户端取得各种信息。

#### 1. 常用方法

(1) public abstract String getAttribute(String name)

返回指定名字的属性的值,如果这个属性不存在则返回 null。与 setAttribute()方法配合使用可实现两个 jsp 文件之间传递参数。

(2) public abstract void setAttribute(String name, Object object)

如果对给定请求的属性有访问权,该方法用给出的对象来替代这个请求属性的现有值。

(3) public abstract Enumeration getAttributeNames()

返回请求中所有属性的名字的枚举。

(4) `public abstract getCharacterEncoding()`

返回请求内容的字符集编码,如果未知则返回空值。

(5) `public abstract getContentlength()`

返回客户端提交的数据缓冲区的大小,如果未知则返回空值。

(6) `public abstract getContentType()`

返回客户端提交数据的类型,如果未知则返回空值。

(7) `public abstract ServletInputStream getInputStream()`

返回一个可以用来读入客户端请求内容的输入流。

(8) `public abstract String getParameter(String name)`

返回给定参数的值,如果参数不存在则返回空值。

(9) `public abstract Enumeration getParameterNames()`

返回含有当前请求的参数名的枚举类型,如果输入流为空或者没有参数被指定,则返回空值。

(10) `public abstract String[] getParameterValues(String name)`

以字符串数组形式返回指定参数的所有值,如果这个参数不存在则返回空值。

(11) `public abstract String getProtocol()`

返回一个由客户端请求使用的协议及其版本号组成的字符串。

(12) `public abstract String getRemoteAddr()`

返回由提交请求的客户端的 IP 地址组成的字符串。

(13) `public abstract String getRemoteHost()`

返回提交请求的客户端的主机名,如果主机名不能确定,那么就返回客户端的 IP 地址。

(14) `public abstract int getServerPort()`

返回接收当前请求的端口号。

(15) `public abstract getServerName()`

返回接收当前请求的服务器的主机名,如果主机名不能确定,那么返回这个主机的 IP。

(16) `public abstract String getAuthType()`

返回这个请求所使用的 HTTP 认证方式,如果当前没有使用认证,则返回空值。

(17) `public abstract Cookie[] getCookies()`

将当前请求中所能找到的所有 Cookie 都放在一个 Cookie 对象的数组中返回。

(18) `public abstract getDateHeader(String name)`

返回指定的日期域的值,如果这个域未知,则返回空值。

(19) `public abstract String getHeader(String name)`

返回指定的域的值,如果这个域未知,则返回空值。

(20) `public abstract Enumeration getHeaderNames()`

返回头部所有域名所构成的枚举,如果服务器不能访问该头部的域名,那么返回空值。

(21) `public abstract String getMethod()`

返回客户端产生请求所使用的方法。

(22) `public abstract String getPathInfo()`

返回 URL 中跟在服务器路径后面的可选路径信息,如果没有路径信息,则返回空值。

(23) `public abstract String getPathTranslated()`

返回额外的路径信息,这个路径将被翻译成物理路径,如果没有给出这个额外路径,那么返回空值。

(24) `public abstract String getQueryString()`

返回 URL 的请求字符串部分,如果没有请求字符串,那么返回空值。

(25) `public abstract String getRemoteUser()`

返回产生请求的用户的用户名,如果该用户未知,则返回空值。

(26) `public abstract String getSessionId()`

返回当前请求所指定的会话 ID。

(27) `public abstract String getRequestURI()`

返回当前请求的 URI。

(28) `public abstract HttpSession getSession()`

该方法取得与当前请求绑定的 Session,如果当前 Session 尚不存在,那么就为这个请求创建一个新的 session。

(29) `public abstract boolean isRequestSessionIdFromCookie()`

如果这个请求的会话 ID 是从一个 Cookie 中得来的,那么返回真。

(30) `public abstract boolean isRequestSessionIdFromURL()`

如果这个请求的会话 ID 是从一个 URL 的一部分中得来的,那么返回真。

## 2. 例示

在 JSP 页面设计中用到的 Request 对象的方法主要是获取服务器各种参数的方法。如例 7-1 所示。与客户端提交的表单交互的方法,如例 7-2、例 7-3 所示。仅仅用 Request 对象开发动态 Web 页面是不够的,它必须和其它内部对象一起共同完成。下面我们就 Request 对象的一些方法作点介绍,希望能起到抛砖引玉的作用。

让我们先看看 Tomcat 的 JSP 帮助文档中的 `snoop.jsp`。细心的读者会发现在许多资料上看过,因为该例子体现了 Request 对象的许多方法,下例在此基础上加入了一些其它的方法。

### 例 7-1

```
<html>
<!--
  Copyright (c) 1999 The Apache Software Foundation. All rights
  reserved.
-->
<body bgcolor="white">
<h1> Request Information </h1>
```

```
<font size="4">
<%String name="holle";%>
JSP Request Method: <%= request.getMethod() %>
<br>
Request URI: <%= request.getRequestURI() %>
<br>
Request Protocol: <%= request.getProtocol() %>
<br>
Servlet path: <%= request.getServletPath() %>
<br>
Path info: <%= request.getPathInfo() %>
<br>
Path translated: <%= request.getPathTranslated() %>
<br>
Query string: <%= request.getQueryString() %>
<br>
Content length: <%= request.getContentLength() %>
<br>
Content type: <%= request.getContentType() %>
<br>
Server name: <%= request.getServerName() %>
<br>
Server port: <%= request.getServerPort() %>
<br>
Remote user: <%= request.getRemoteUser() %>
<br>
Remote address: <%= request.getRemoteAddr() %>
<br>
Remote host: <%= request.getRemoteHost() %>
<br>
getAttribute : <%= request.getAttribute("name") %>
<br>
getSession : <%= request.getSession() %>
<br>
Authorization scheme: <%= request.getAuthType() %>
<br>
PathTranslated: <%= request.getPathTranslated() %>
<br>
QueryString: <%= request.getQueryString() %>
```

---

<hr>

The browser you are using is <%= request.getHeader("User-Agent") %>

<hr>

</font>

</body>

</html>

运行结果如图 7-1 所示。

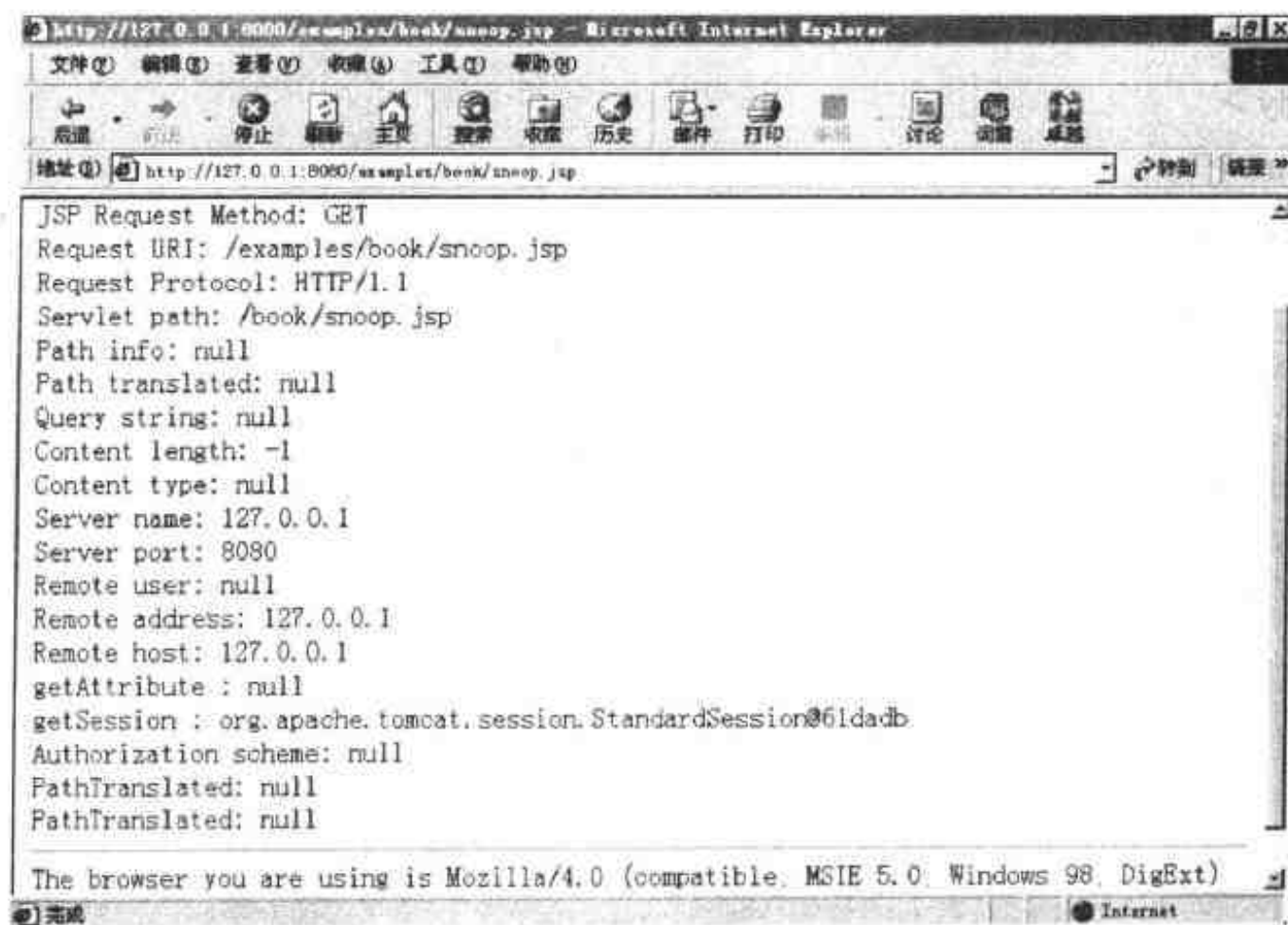


图 7-1 snoop.jsp 的输出结果

上例主要体现了 Request 对象获取服务端各种参数的方法。在帮助文件原例上加入了几个方法,如:getAttribute()方法、getSession()方法。

在 JSP 程序设计中,Request 对象经常被用来获取客户端表单数据,下面介绍 getParameter()方法。

通过下面的例子我们可以了解 Request 对象如何从文本框中获取参数。首先建立一个表单,它含有三个文本框,分别用来接受用户的姓名、昵称和密码,通过它们判断该用户是否拥有服务权限。

#### 例 7-2

<html>

<head>

<title>login</title>

</head>

<body>

<form method="post" action="login\_result.jsp">

你的姓名:

```
<input name="name" type="text" size=16 maxlength="30"><br><br>
你的昵称:
<input name="l_name" type="text" size=16 maxlength="30"><br><br>
你的密码:
<input name="psw" type="password" size=16 maxlength="30"><br><br>
<input type="submit" value="提交"><br><br>
<input type="reset" value="重置" name="reset">
</form>
<body>
</html>
```

其在浏览器上的显示效果如图 7-2 所示。

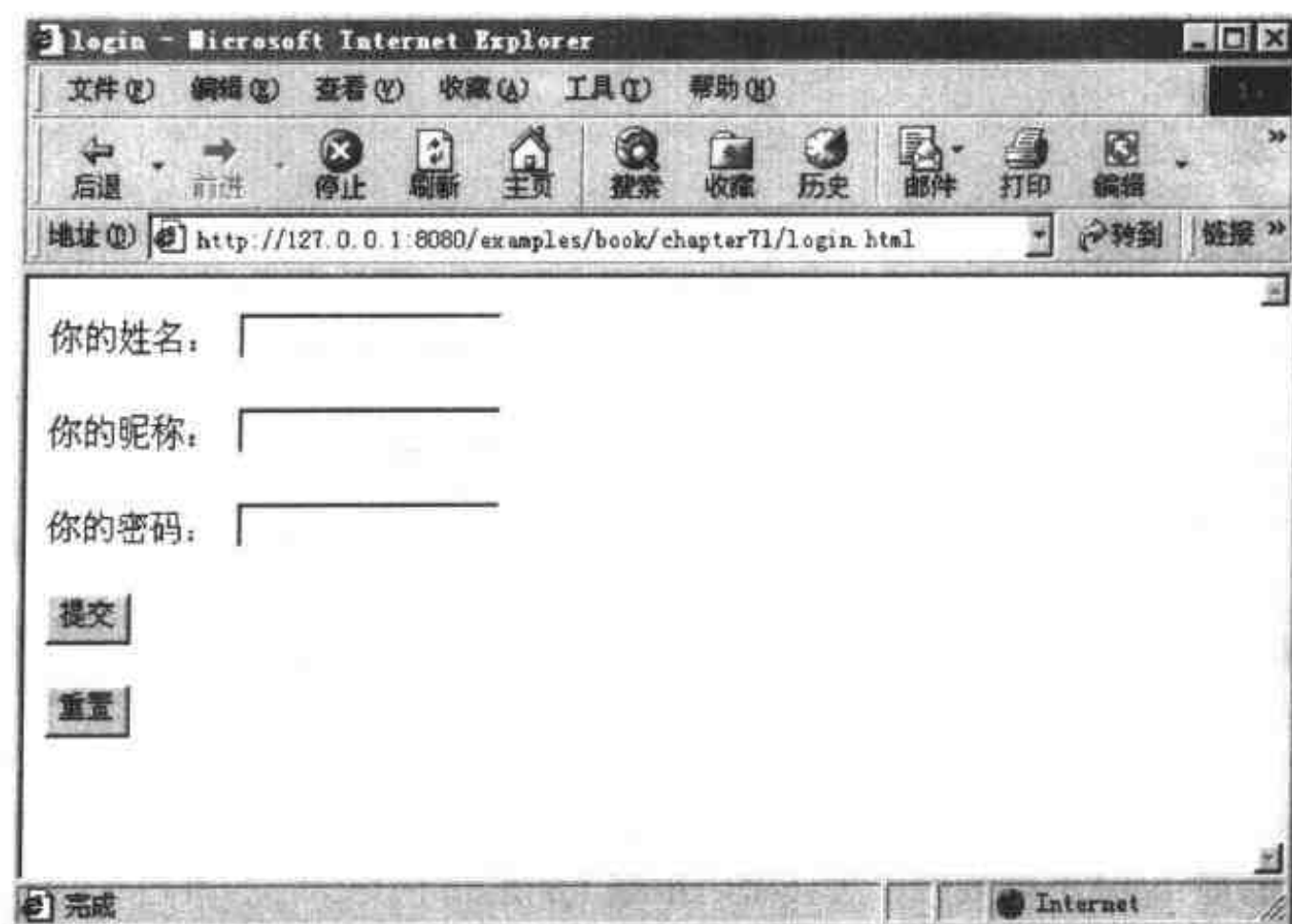


图 7-2 登陆页面效果图



注意:该 html 文件用的是 post 方法(其它方法有 get, put 等)向服务器提交信息,使服务器获得文本框的 name 属性。request.getParameter()就是通过 name 属性取得参数。

处理上述表单数据的程序源代码如下:(login\_result.jsp)

```
<html>
<head>
<title>login_result</title>
</head>
<body>
<%@page contentType="text/html; charset=GB2312"%>
<%
```

```
//定义一个字符串对象用来保存 getParameter()方法获取的表单属性名为 name 的值。  
String temp_username = request.getParameter("name");  
String temp_lname = request.getParameter("l_name");  
String tempsw = request.getParameter("psw");  
//将获得的参数进行内码转换。  
byte[] temp_busername;  
String temp_Susername = temp_username;  
temp_busername = temp_Susername.getBytes("ISO8859-1");  
temp_username = new String(temp_busername);  
  
byte[] temp_blname;  
String temp_slname = temp_lname;  
temp_blname = temp_slname.getBytes("ISO8859-1");  
temp_lname = new String(temp_blname);  
//在客户端用 out 对象的 println 方法将三个字符对象的值输出。  
out.println(temp_username);  
out.println("<br>");  
out.println(temp_lname);  
out.println("<br>");  
out.println(tempsw);  
out.println("<br>");  
%>  
</body>  
</html>
```

当用户在表单中键入姓名为黄冬,昵称为秋秋,密码为 88866 后,点击提交,运行结果如图 7-3 所示。

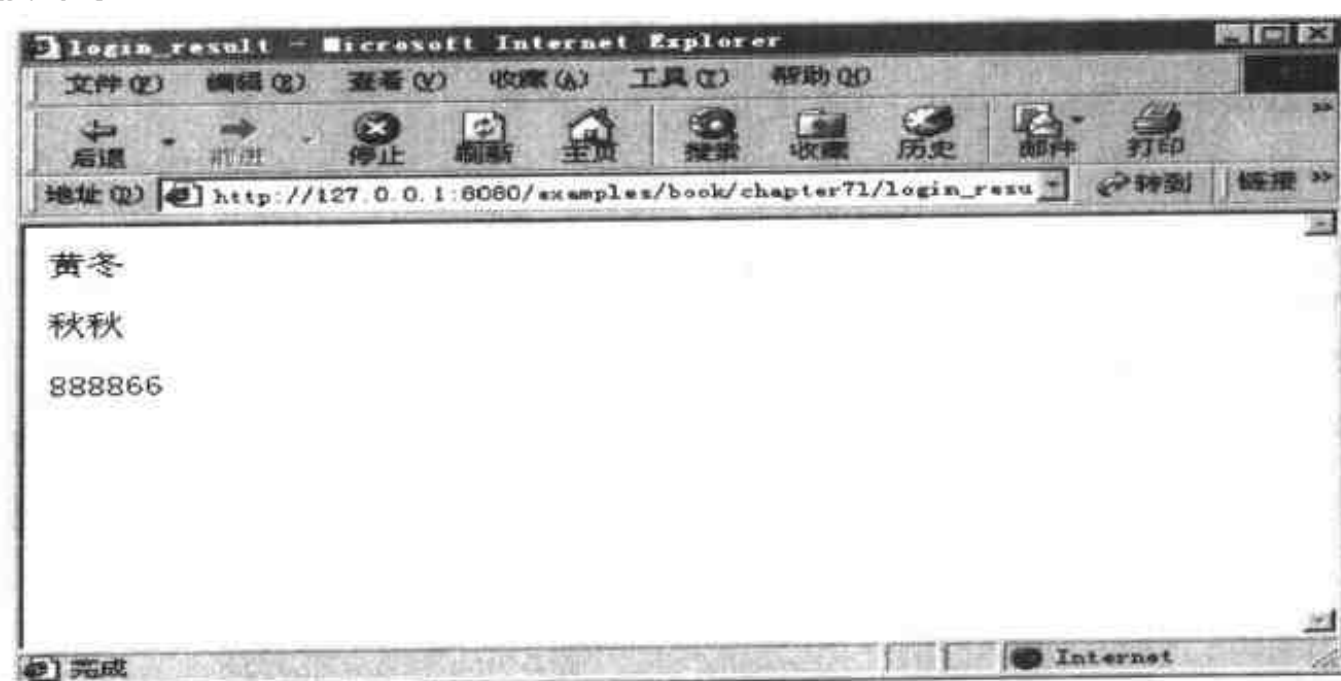


图 7-3 登陆信息处理结果效果图

细心的读者会发现在源代码中有几行代码是进行内码转换的,为什么呢?因为在使



用中文时发现无法正常显示,常以问号“?”代替,或者是乱码,所以,当 JSP 页面中出现中文乱码问题时可以用上例中的方法解决。根据经验还有其它的一些方法:

(1) 在 JSP 页面中加入一条语句:

```
<%@ page contentType="text/html; charset = gb2312"%>
```

(2) 在 Classpath 中加入 il8n.jar

(3) 在控制面板中的区域设置中将区域设为英语(美国)然后重新启动即可。

数据库的中文问题比较复杂,笔者曾遇到输入数据库中的中文字符在直接打开数据库查看表项时不会显示,但通过 JSP 程序对其查询、插入、更新时只要内码转换一致,一般不会出错。有兴趣的读者可以看一下本书所附光盘中综合实例聊天室部分的例子,相信会有帮助。

### 例 7-3

下例将介绍如何通过复选框、单选按钮获得信息。首先建立一个 HTML 表单如下。

```
<html>
<head>
<title>choise</title>
</head>
<body>
<form method="post" action="choise.jsp">
<p>您喜欢的水果(多选):
<p><input type="checkbox" name="mychoise" value="苹果"> 苹果
<p><input type="checkbox" name="mychoise" value="桔子"> 桔子
<p><input type="checkbox" name="mychoise" value="葡萄"> 葡萄
<p><input type="checkbox" name="mychoise" value="都不喜欢"> 都不喜欢
<br><br>
<p>您的性别:<P>
<input type="radio" name="sex" value="1"> 男<br>
<input type="radio" name="sex" value="0"> 女<hr>
<input type="submit" value="提交">
</form>
<body>
</html>
```

其在浏览器上显示如图 7-4 所示。

表单中有四个复选框(表示喜欢的水果,多选)两个单选按钮(表示不同性别),用户可选择自己喜欢的水果和自己的性别,也可以不作任何选择。当用户选择完提交后,由 choise.jsp 文件处理,源程序如下:

```
<html>
<head>
<title>choise</title>
</head>
```

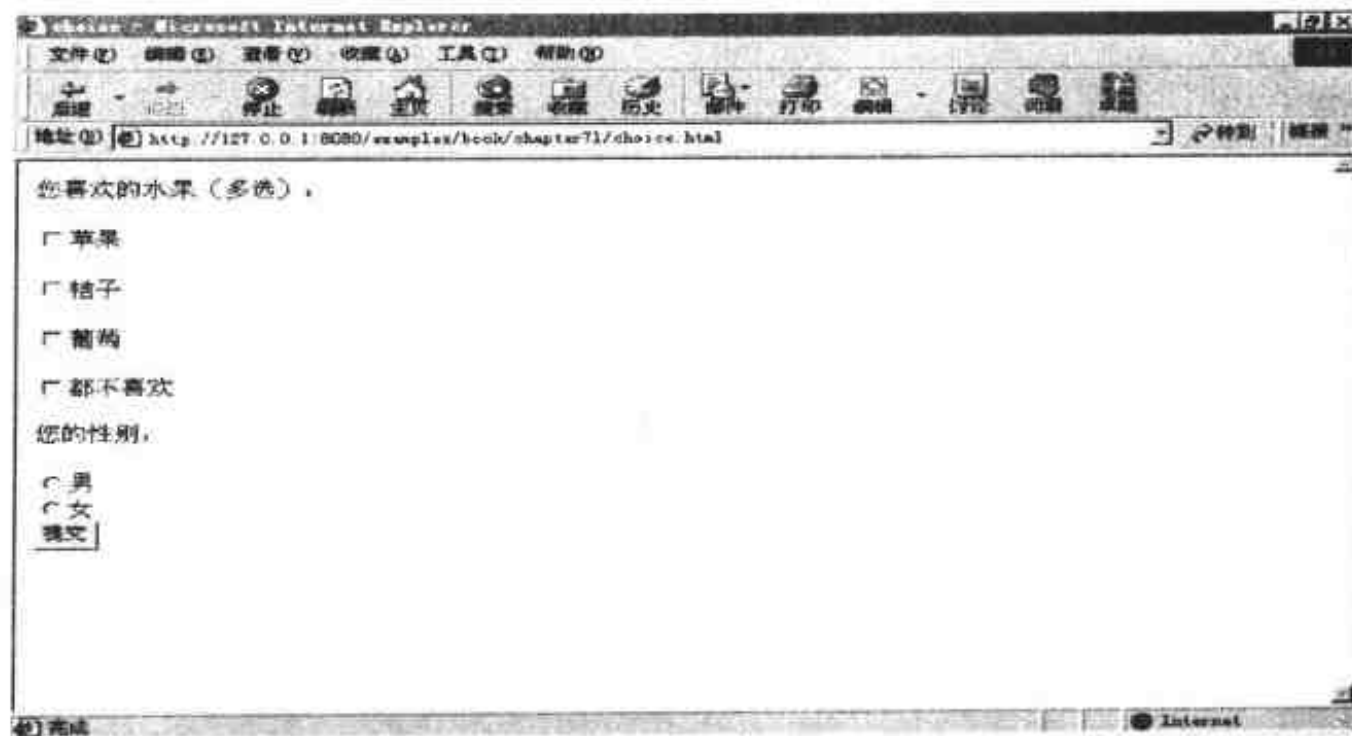


图 7-4 choise.html 运行效果图

```

<body>
<%
String msg = "";
String sex = request.getParameter("sex");
String[] item = request.getParameterValues("mychoise");
int len = item.length;
    if (sex == null){
msg = "您没有选择性别<br>";
    }
    if (sex.equals("1")){
msg = msg + "你好先生！<br>";
    }
    if
msg = msg + "你好女士！<br>";
    }
    if(len == 0){
msg = msg + "您没有选择水果";
    }
    else{
out.println("<p>感谢您的选择！</p><br>");
msg = msg + "<br>您选择了:<br>";
for(int i=0;i<len;i++){
msg = msg + item[i] + "<br><br>";
}
}
out.println(msg);

```

```
%>  
</body>  
</html>
```

从程序中可以看到使用了 `getParameter()` 方法来取得表单中属性为“sex”的值,并把值赋给变量 `sex` 保存,接着使用 `if` 语句作判断,如为 `null`,则将字符串变量 `msg` 用“您没有选择您的性别”字符串赋值,表明用户没有对单选框作选择。如为“1”,表明用户选择了“男”,则将“您好先生”字符串送到 `msg` 中,如为“0”,表明用户选择了“女性”,就将“您好女士”送到字符串变量 `msg` 中。程序对表单中复选框的内容使用 `getParameterValues()` 方法,获取后也作同样的判断,并同样对 `msg` 变量赋相应的字符串,最后将结果返回到客户端显示。如图 7-5 所示(例如用户选择了“男”、“苹果”、“桔子”、“葡萄”)。

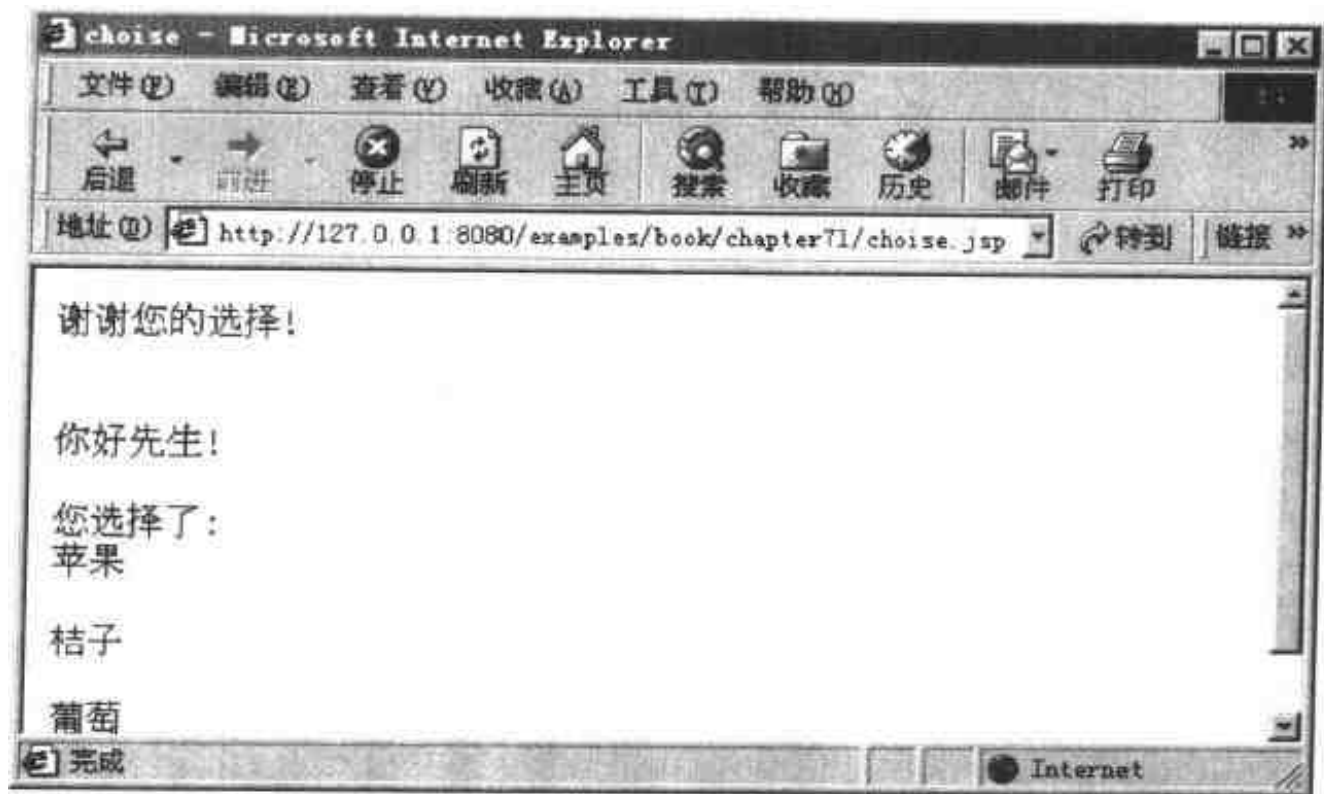


图 7-5 choose.jsp 处理用户提交信息的输出界面

## 7.2.2 Response 对象

当服务器处理完客户请求时,往往需要发送一些信息给客户端浏览器,或者需要重定向到其它页面。内部对象 `Response` 是一个 `HttpServletResponse` 对象,它提供了几个用于设置送回浏览器的响应方法(如 `Cookies` 信息)。

### 1. 常用方法

(1) `public abstract String getCharacterEncoding()`

返回响应的字符编码的 MIME 类型,如果没有指定类型,那么字符编码被默认设置为 `text/plain`。

(2) `public abstract ServletOutputStream getOutputStream()`

返回用来写入响应数据的输出流。

(3) `public abstract PrintWriter getWriter()`

返回一个打印 `writer` 来产生发回用户端的格式化的文本响应。

- (4) `public abstract void setContentLength(int length)`  
设置返回响应的数据的长度。
- (5) `public abstract void setContentType(String type)`  
设置响应的 MIME 类型。
- (6) `public abstract void addCookie(Cookie cookie)`  
将指定的 Cookie 加入响应。
- (7) `public abstract String encodeRedirectURL(String url)`  
将指定的 URL 编码,以便在 `sendRedirect()`方法中使用。
- (8) `public abstract String encodeURL()`  
将指定的 URL 和会话 ID 一起编码。
- (9) `public abstract void sendError(int code)`  
用某个状态代码向用户端发送一个发现错误代码,出错信息使用默认值。
- (10) `public abstract void sendError(int code,String message)`  
用给出的状态代码和消息向用户端发送一个发现错误响应。
- (11) `public abstract void sendRedirect(String url)`  
将对用户端的响应重定向到指定的 URL 上。
- (12) `public abstract void setDataHeader(String name,long value)`  
将指定的域加到响应首部,并赋给它一个时间值,如果这个域已经设置了值,那么它将被新设置的值代替。
- (13) `public abstract void setHeader(String name,String value)`  
将指定的域加入到响应首部,并赋给它一个字符串值,如果这个域已经设置了值,那么它将被新设置的值代替。
- (14) `public abstract void setStatus(int code)`  
设置响应的状态代码,使用默认的消息。
- (15) `public abstract void setStatus(int code,String message)`  
设置响应的状态代码及消息。

## 2. 例示

下面结合几个例子对上面的一些重要方法进行讲解,同时对前面所学的 Request 对象的方法作一下复习。

### 例 7-4

在例 7-4 中用 Request 对象的 `getParameter()`方法获取了客户端提交的表单内容,并将该内容显示到客户端。在这里仍然用此方法获取同样的内容,但并不把它直接显示到客户端,而是先判断该用户是否为合法的用户,再根据判断结果使用 Response 对象的 `encodeURL()`方法和 `sendRedirect()`方法对不同用户指定不同登录页面。登录页面效果参见图 7-2 所示。处理程序如下。

```
<html>
<head>
<title>login _ result</title>
```

```

</head>
<body>
<%@page contentType="text/html; charset = GB2312"%>
<%
String temp _ username = new String(request.getParameter("name"));
String temp _ lname = new String(request.getParameter("l _ name"));
String temp _ psw = new String(request.getParameter("psw"));
//将获得的参数进行内码转换。
byte[] temp _ busurname;
String temp _ Surname;
temp _ Surname = temp _ username;
temp _ busurname = temp _ Surname.getBytes("ISO8859-1");
temp _ username = new String(temp _ busurname);

byte[] temp _ blname;
String temp _ slname;
temp _ slname = temp _ lname;
temp _ blname = temp _ slname.getBytes("ISO8859-1");
temp _ lname = new String(temp _ blname);
//判断用户输入信息完整性。
if(temp _ username.equals("") || temp _ lname.equals("") || temp _ psw.equals("")){
//定义一个字符变量保存 login.jsp 文件的绝对路径。
String redirectURL = "/examples/book/chapter7l/login.jsp";
//用 encodeURL 方法对该路径进行编码,在用 sendRedirect 方法重定向。
response.sendRedirect(response.encodeURL(redirectURL));
}
else{
if(temp _ username.equals("黄冬") && temp _ lname.equals("秋秋") &&
temp _ psw.equals("88866")){
out.println("<h2>" + temp _ lname + "你好! 送你一个惊喜<h2>");
out.println("<td><img src='image.JPG' width='160' height='100' >
</td><br><br>");
out.println("喜欢吗?");
}
else{
String _ redirectURL = "/examples/book/chapter7l/hello.jsp";
response.sendRedirect(response.encodeURL(_ redirectURL));
}
}
}

```

```
%>  
</body>  
</html>
```

当用户未填写完就提交,处理文件经过判断,先用一个字符串变量保存 login.jsp 文件的绝对路径,用 encodeURL()方法对该路径进行编码,再用 sendRedirect()方法重定向,使用户端始终显示该页面,一直到用户输入完整的信息并提交才进行下一步判断。如果用户名为“黄冬”,昵称为“秋秋”,密码为 88866,则显示友好界面(即输出一幅图片);若不符合条件则使用上述方法进行重定向,引导用户进入注册界面。

当用户未输入完整信息时,重定向到图 7-2 所示页面。当用户正确输入后输出页面效果如图 7-6 所示。

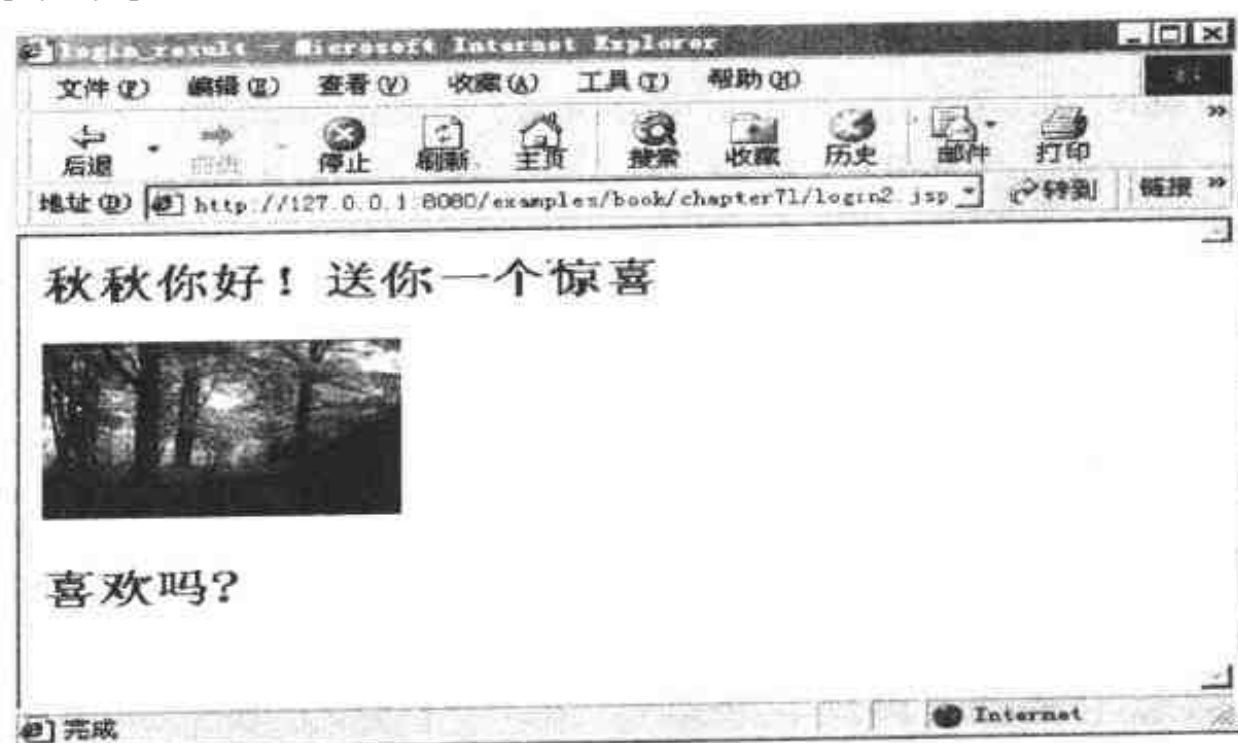


图 7-6 合法用户界面

当用户不是合法用户时,重定向到图 7-7 所示的注册界面。

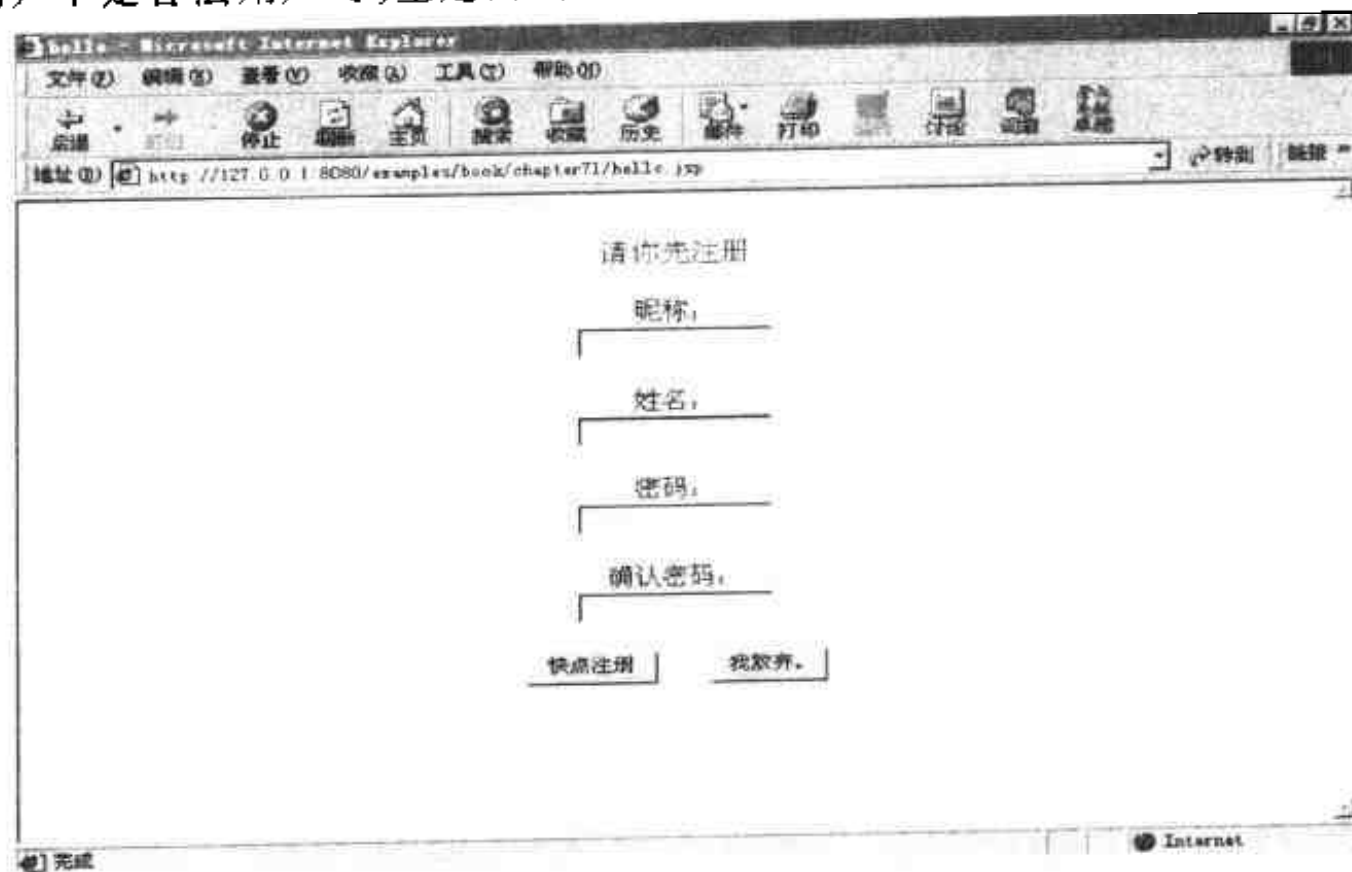


图 7-7 不合法用户的界面

Response 对象的 addCookie() 方法, 与 Request 对象的 getCookie() 方法一起可设置和获取客户端浏览器的诸多信息。目前许多人对 Cookie 技术有争议, 认为这是一种侵犯个人隐私的技术, 但是对程序员而言 Cookie 技术还是一个比较好的获取用户信息的手段。在 JSP 页面设计中, 我们创建一个带有指定信息的 Cookie 对象, 接着再用 addCookie() 方法将该 Cookie 对象写入响应。

例:

```
String username = "黄冬";
Cookie mycookie = new("name", username);
Response.addCookie(mycookie);
```

当我们要从浏览器读出 Cookie 时可使用 getCookie() 方法, 此方法将返回一个 Cookie 数组, 因此需要定义一个数组接受。代码如下:

```
Cookie[] cookie_array = request.getCookie();
If (cookie_array != null)
    { for (int i=0; i<cookie_array.length; i++)
        if (cookie_array[i].getName().equals(name))
            { temp_username = cookie_array[i].getValues(); }
    }
```

最后给出一个使用 Request 对象和 Response 对象创建和显示 Cookie 的完整例子。(该例引自 SUN 公司的技术文档示例)。

```
import java.io.*;
import java.text.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
/ **
 * Example servlet showing request headers
 *
 * @author James Duncan Davidson <duncan@eng.sun.com>
 * /
public class CookieExample extends HttpServlet {

    ResourceBundle rb = ResourceBundle.getBundle("LocalStrings");

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

```

out.println("<html>");
    out.println("<body bgcolor = \"white\">");
    out.println("<head>");
    String title = rb.getString("cookies.title");
    out.println("<title>" + title + "</title>");
    out.println("</head>");
    out.println("<body>");

    // relative links
    out.println("<a href = \".../servlets/cookies.html\">");
    out.println("<img src = \".../images/code.gif\" height = 24 \" +
"width = 24 align = right border = 0 alt = \"view code\"></a>");
    out.println("<a href = \".../servlets/index.html\">");
    out.println("<img src = \".../images/return.gif\" height = 24 \" +
"width = 24 align = right border = 0 alt = \"return\"></a>");
    out.println("<h3>" + title + "</h3>");
    Cookie[] cookies = request.getCookies();
    if (cookies.length > 0) {
        out.println(rb.getString("cookies.cookies") + "<br>");
        for (int i = 0; i < cookies.length; i++) {
            Cookie cookie = cookies[i];
            out.print("Cookie Name: " + cookie.getName() + "<hr>");
            out.println("Cookie Value: " + cookie.getValue() +
                "<br><br>");
        }
    }
    else {
out.println(rb.getString("cookies.no_cookies"));
    }
    String cookieName = request.getParameter("cookieName");
    String cookieValue = request.getParameter("cookieValue");
    if (cookieName != null && cookieValue != null) {
        Cookie cookie = new Cookie(cookieName, cookieValue);
        response.addCookie(cookie);
        out.println("<P>");
        out.println(rb.getString("cookies.set") + "<br>");
        out.print(rb.getString("cookies.name") + " " + cookieName +
            "<br>");
        out.print(rb.getString("cookies.value") + " " + cookieValue);
    }

```



```

    }
    out.println("<P>");
    out.println(rb.getString("cookies.make_cookie") + "<br>");
    out.print("<form action = \"");
    out.println("CookieExample \" method = POST>");
    out.print(rb.getString("cookies.name") + " ");
    out.println("<input type = text length = 20 name = cookiename><br>");
    out.print(rb.getString("cookies.value") + " ");
    out.println("<input type = text length = 20 name = cookievalue><br>");
    out.println("<input type = submit></form>");

    out.println("</body>");
    out.println("</html>");
}

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

```

该程序第一次运行页面效果如图 7-8 所示。

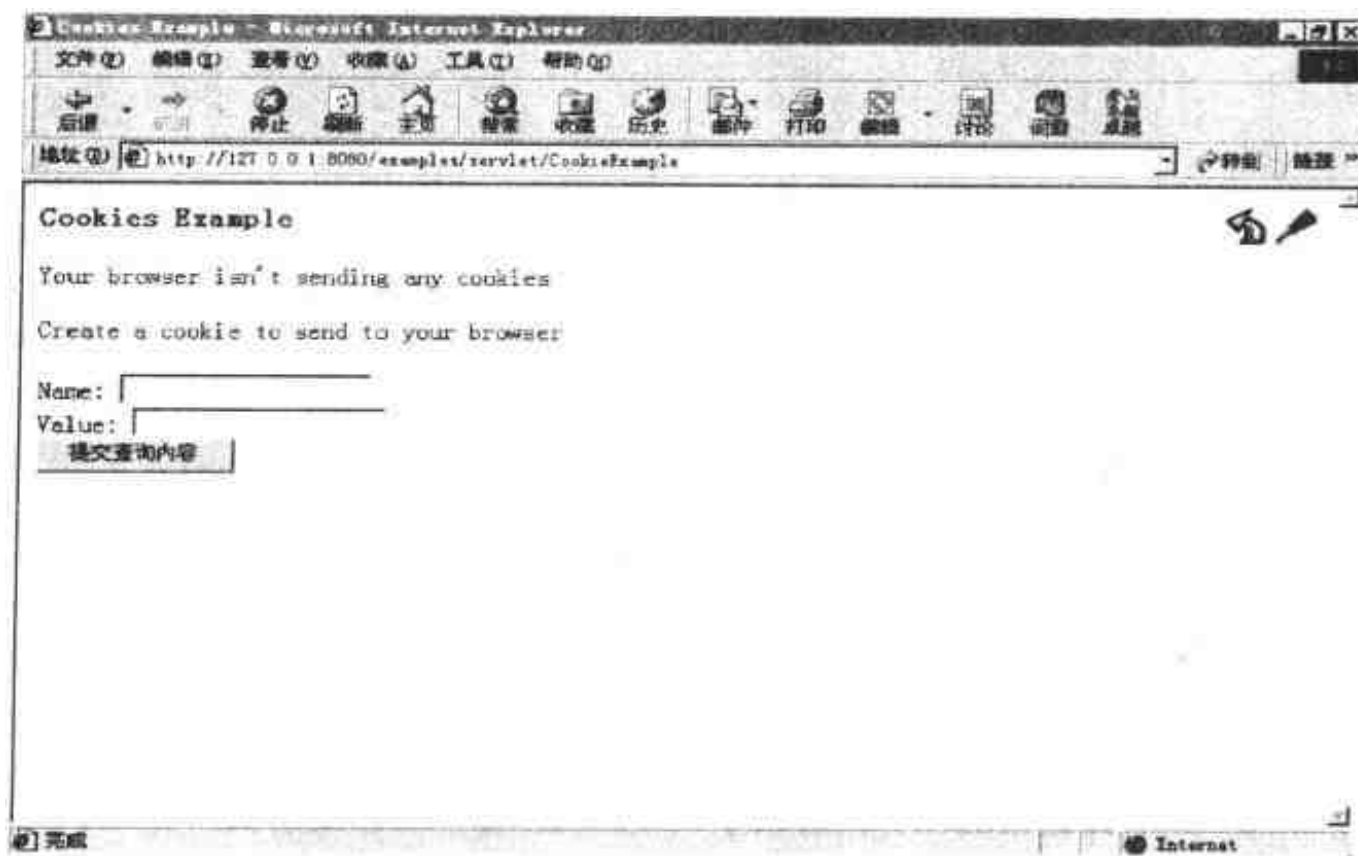


图 7-8 CookieExample

提交表单, 创建一个 Cookie 后, 页面效果如图 7-9 所示。



图 7-9 CookiesExample 运行后效果图

### 7.2.3 Out 对象

服务器是怎么把信息在用户浏览器显示的呢？这是由 Out 对象来完成的。可能有人会说，用表达式就可以嘛？确实是这样的。但是，表达式最终也是转换成 Out 对象输出的。JSP 页面中的表达式，经求值以后的结果将转换成 String 对象，随后该 String 对象被发送到 out 对象输出。

Out 对象的基类是 JspWriter。Out 对象主要的方法是：out.print() 方法和 out.println() 方法。两者区别在于 print 方法输出完后，并不结束当前行，而 println() 方法在输出完毕后，会结束当前行，这一点与 C 语言中输出语句在其后加 ln 和不加 ln 时作用是一样的。上述两种方法在 JSP 页面设计中是经常用到的，它们可以输出各种格式的数据类型，如字符型、整型、浮点型、布尔型甚至可以是一个对象，还可以是字符串与变量的混合型以及表达式。



注意：字符串一定要用引号将其括起，与变量在一起时要用“+”相连。

#### 1. 常用方法

##### (1) newLine()

输出换行符。

##### (2) close()

关闭输出流。

##### (3) flush()

输出缓冲区数据。若缓冲区未设置或无数据则输出空串，在页面中无显示。

##### (4) clear()

清除缓冲区。

##### (5) getBufferSize()

获得缓冲区大小。

## 2. 例示

### 例 7-5

```
<html>
<head>
<title>print</title>
</head>
<body>
<%
double p=3.14;
String objmy= new String("Π=");
for(int i=0;
out.println("<h1>" + objmy + p + "</h1>");
|
%>
</body>
</html>
```

页面效果,如图 7-10 所示。

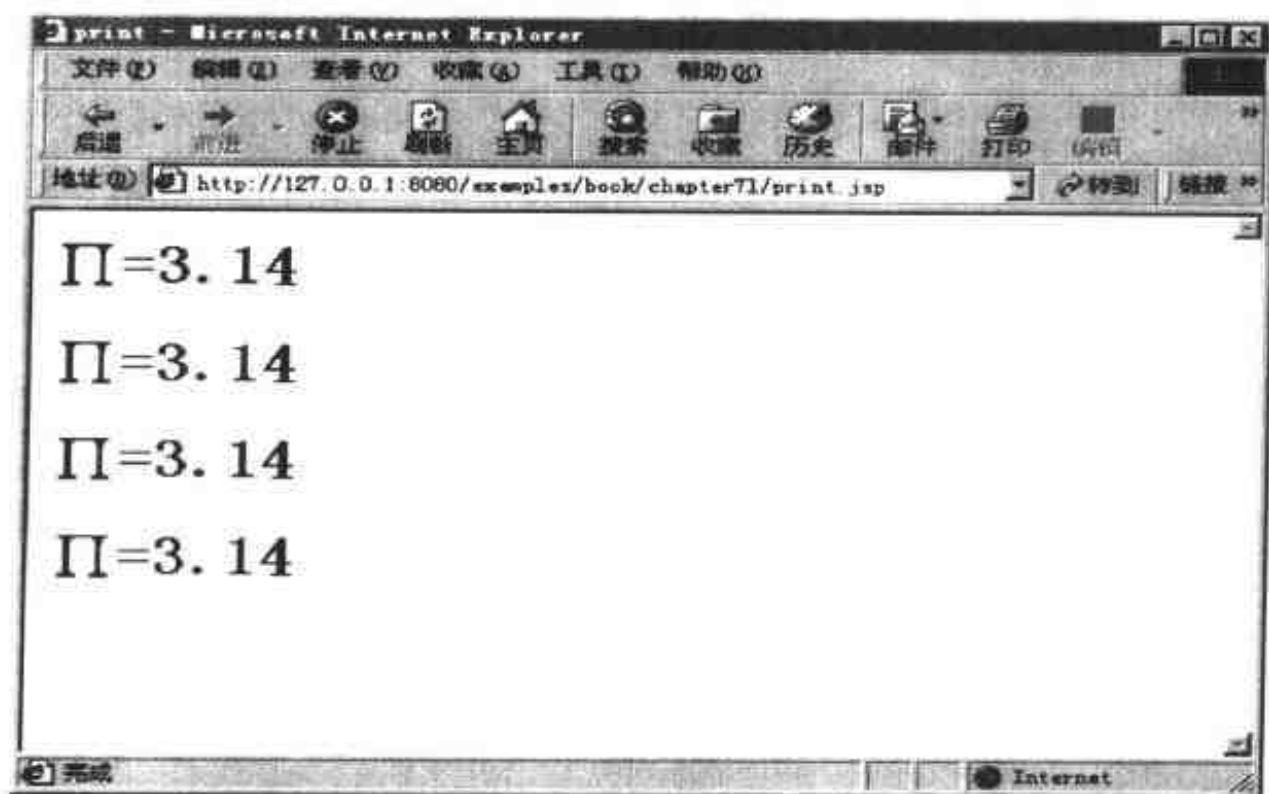


图 7-10 Print.jsp 运行效果图

### 例 7-6

```
<html>
<head>
<title>cookie</title>
</head>
```

```
<body>
<%@page contentType="text/html; charset = gb2312"%>
<%
String _name = request.getParameter("username");
String _addr = request.getParameter("addr");
if (_name! = null && _addr! = null){
//将获得的参数进行内码转换。
byte[] temp_bname;
String temp_Sname;
temp_Sname = _name;
temp_bname = temp_Sname.getBytes("ISO8859-1");
_name = new String(temp_bname);
byte[] temp_baddr;
String temp_saddr;
temp_saddr = _addr;
temp_baddr = temp_saddr.getBytes("ISO8859-1");
_addr = new String(temp_baddr);
    out.println("<h1> 姓名:" + _name + "<h1>");
out.println("<h1> 地址:" + _addr + "<h1>");
}
else{
out.println("<form method = 'POST'");
out.println("action = 'print2.jsp' >");
out.println("<table border>");
out.println("<tr>");
out.println("<th> 姓名:");
out.println("</th><td colspan = 3><input");
out.println(" name = 'username' type = 'text' size = '35' ></td>");
out.println("</tr><tr><th> 地址:");
out.println("</th><td colspan = 3>");
out.println("<input name = 'addr' type = 'text'");
out.println(" size = '35' ></td>");
out.println("</tr></table>");
out.println("<p><input type = reset value = '复位'");
out.println("><input type = submit value = '提交' ></form>");
}
%>
</body>
</html>
```

上例中我们先用 Request 对象获取表单中属性名为 name 和 addr 的值,因为在同一个 jsp 文件中处理,所以在第一次调用该文件时表单中可能无提交值,则须进行判断,若取值为空,则用 out 对象输出表单让用户添写提交。若不为空,则显示用户提交内容。上例第一次运行如图 7-11 所示。

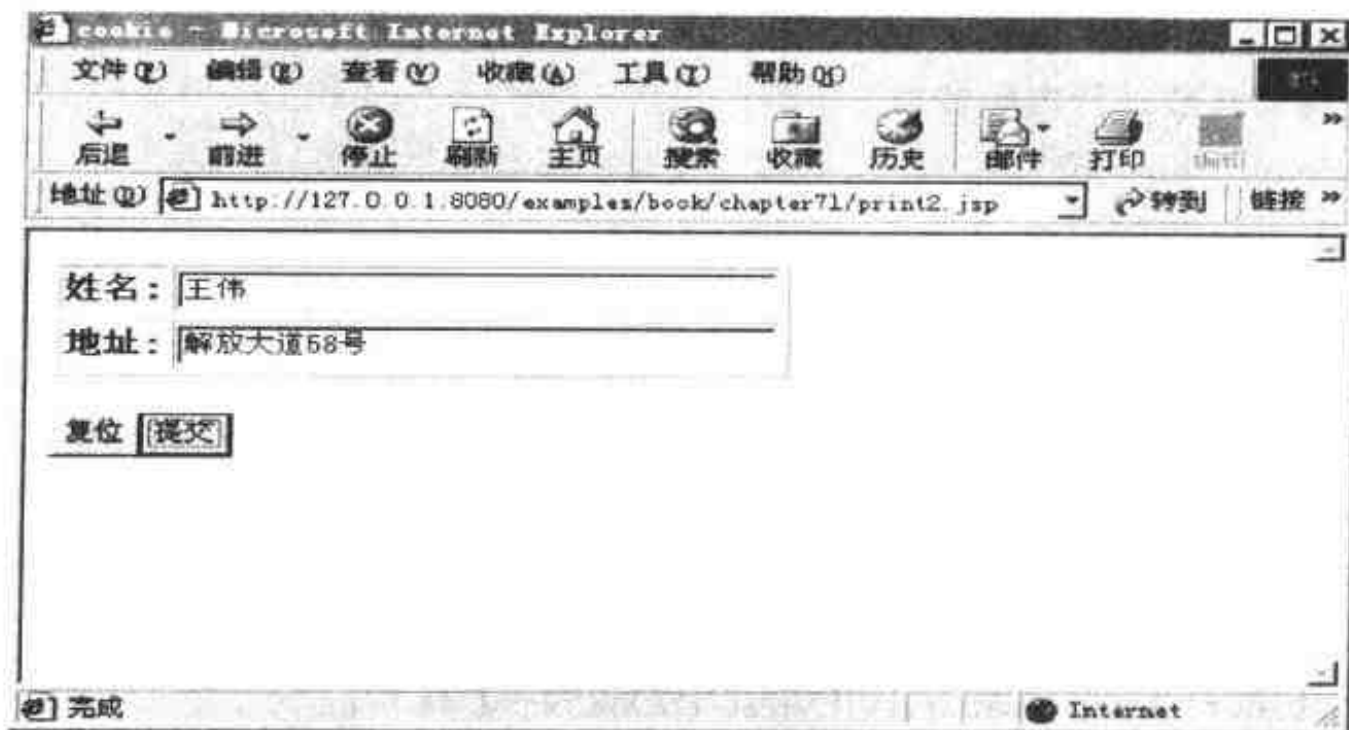


图 7-11 用户输入界面

当用户填写提交后在浏览器上显示如图 7-12 所示。



图 7-12 结果输出界面

最后,谈一下 println()方法与<% = value %>的区别:

(1) 这两种方法都可以进行输出。<% value %>中 value 可以是变量名、数值、字符串,也可以为一个对象。

(2) 两种方法在不同的情况下有各自的优点,比如在实现多语句输出时,前种方法较好;在不同地方直接显示值时,后种方法就更符合人的思维,更灵活。

(3) 两种方法也各有不足之处,前种方法不够灵活,后种方法每一次输出都必须加

上`<% = Value% >`,在输出内容多时,反而会显的复杂。

## 7.2.4 Session 对象

Session 对象提供了服务器与客户端的一种联系,也就是通常所说的会话。这种会话可以在给定的时间内保持多个连接,用来维护状态和用户标志。在 JSP 页面中,我们可以用 Session 对象来保存特定用户的会话信息,即使该用户由一个页面跳到另外一个页面,该会话信息仍然存在。Session 对象在客户端向该页面发出请求时建立,Session 到期或被终止时撤消。

换个角度来看,其实 Session 对象就像是一把个人的密匙,当任何一个用户进入某一个 JSP 页面时,系统就为他生成一个独一无二的 Session 对象来记录该用户信息,该用户就拥有了一把只供用户个人使用的密匙。你可以使用 Session 对象来储存某个特定用户信息,即使该用户由一个页面跳转到另一个页面,该 Session 对象内的信息也仍然存在。一旦用户正常退出,该密匙即作废。

### 1. 常用方法

(1) `setAttribute(String name,Java.lang.Object)`

设定指定名字的属性值,并且将它添加到 session 对象中。

(2) `getAttribute(String name)`

获得指定名字的属性,若不存在则返回 null。

(3) `putValue(String name,Java.lang.Object)`

设定 name 指定属性的值。

(4) `getValue(String name)`

取得 name 指定属性的值。

(5) `getAttributeName()`

返回 Session 对象中存储的每一个对象,结果为 Enumeration 类实例。

(6) `getId()`

每生成一个 Session 对象,服务器会给一个编号,编号无重复。此方法返回当前编号。

(7) `getCreationTime()`

退回创建时间,单位为毫秒。

### 2. 例示

#### 例 7-7

有时我们可以用 Session 对象来判断该用户是否具有访问某个页面的权限,防止非法用户直接在浏览器的 URL 栏中键入某个页面的 URL 地址,而访问到该页面,我们可以单独的编写一个名为 `check..jsp` 文件,将它用 `include` 指令包含到每一个静态页面中,这样当用户非法调用时,`check.jsp` 文件会从用户的 Session 对象中取出用户的用户名,该用户名是在通过了身份验证后写人的。所以只要读出的值为空,就可判断用户有无访问权限,这时会利用 Response 对象重定向到登陆界面进行身份验证。

代码如下：

```
<%  
//读取 session 对象中属性为“name”的值。并转换为字符型由 _name 保存。  
String _name = (String) session.getValue("name");  
//对 _name 进行判断。  
if (_name == null){  
//定义一个字符变量保存 login.jsp 文件的绝对路径。  
String _redirectURL = "/examples/book/chapter7l/login.jsp";  
//用 encodeURL 方法对该路径进行编码,在用 sendRedirect 方法重定向。  
response.sendRedirect(response.encodeURL(_redirectURL));  
}  
%>
```

比如现在有一个网上电话业务的网站,只有该网站的注册会员才可使用该业务,我们便可在该页面头部用 include 指令将 check.jsp 文件包含进去,直接在浏览器上调用该页,这时将出现登录界面,代码如下。

```
<html>  
<head>  
<title>网上电话系统</title>  
</head>  
//注意:此处将 check.jsp 文件包含  
<%@ include file = "check.jsp"%>  
<body>  
<div align="center">  
<P><font size=7><h>欢迎使用网上 IP 电话</h></font>  
<br><br>  
<font size=5><b>  
<a href="c:\tel.html">拨打电话</a>  
</hr><br>  
<a href="cxhf.jsp">查寻话费</a>  
<br><br>  
<a href="xgmm.jsp">修改密码</a>  
<br><br>  
<a href="index.html">返回主页</a>  
<br><br>  
</b></font>  
</div>  
</body>  
</html>
```

## 例 7-8

为说明 Session 对象的具体应用,接下来我们用三个页面模拟一个多页面的 Web 应用。第一个页面( p1.jsp )仅包含一个要求输入用户名字的 HTML 表单,代码如下:

```
<html>
<head>
<title>p1.jsp</title>
</head>
<body>
<form method="post" action="p2.jsp">
<p>你的姓名:
<input name="username" type="text" size="16" maxlength="10">
<br><br>
<input type="submit" value="提交">
</form>
</body>
</html>
```

运行效果,如图 7-13 所示。

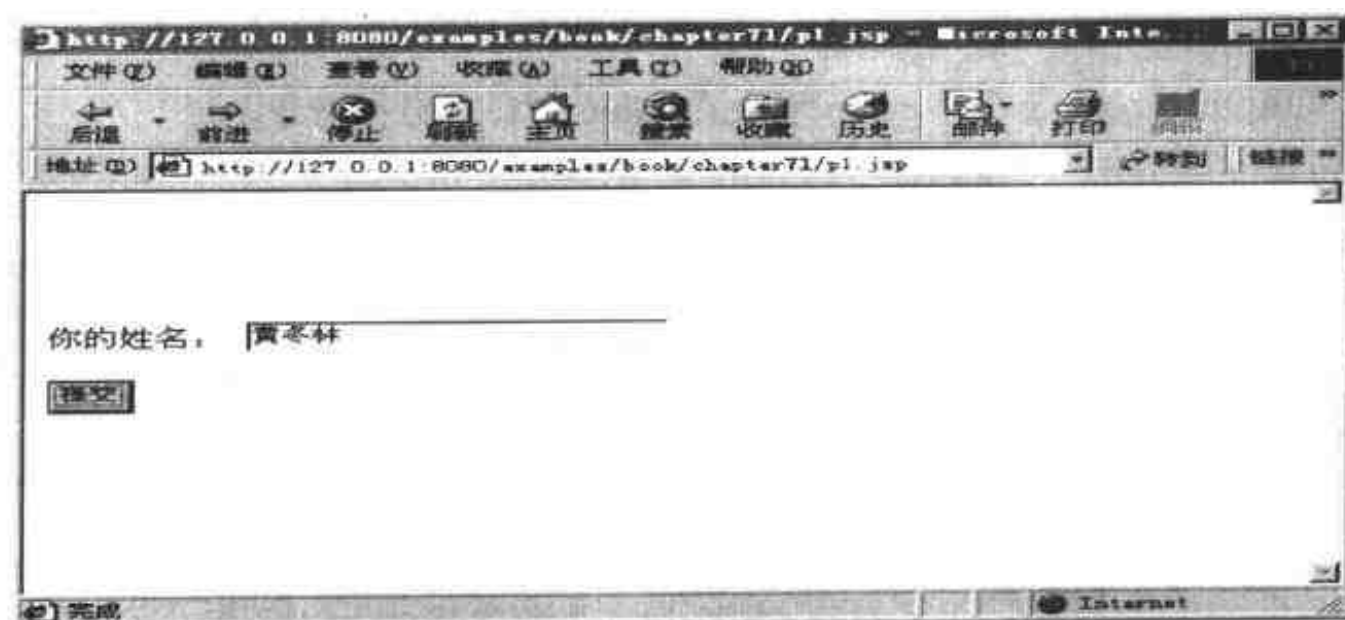


图 7-13 P1.jsp 运行效果图

第二个 JSP 页面 p2.jsp,它通过 request 对象提取 p1.jsp 表单中的 username 值,将它存储为 \_name 变量,然后将这个 name 值保存到 Session 对象中。Session 对象是一个名字/值对应的集合,在此对应中,名字为“Sname”,值即为 \_name 变量的值。由于 Session 对象在会话期间是一直有效的,因此这里保存的变量对后继的页面也有效。p2.jsp 的另外一个任务是询问第二个问题。下面是它的代码:

```
<html>
<head>
<title>p2.jsp</title>
</head>
<body>
<% @page contentType="text/html; charset = GB2312"% >
```



```

<%
String _name = new String(request.getParameter("username"));
byte[] temp _busername;
String temp _Susername;
temp _Susername = _name;
temp _busername = temp _Susername.getBytes("ISO-8859-1");
_name = new String(temp _busername);
session.putValue("Sname", _name);
%>
<p>您的姓名是:<% = _name %>
<form method=POST action="p3.jsp">
<p>您喜欢的水果(多选):
<p><input type = checkbox name="mychoise" value="Apple"> Apple
<p><input type = checkbox name="mychoise" value="Orange"> Orange
<p><input type = checkbox name="mychoise" value="Grape"> Grape
<p><input type = checkbox name="mychoise" value="Not all"> Not all
<br><br>
<p>您的性别:
<p><input type="radio" name="sex" value="1">男<br>
<input type="radio" name="sex" value="0">女<br>
<input type="submit" value="提交"><br><br>
</form>
</body>
</html>

```

运行后如图 7-14 所示。

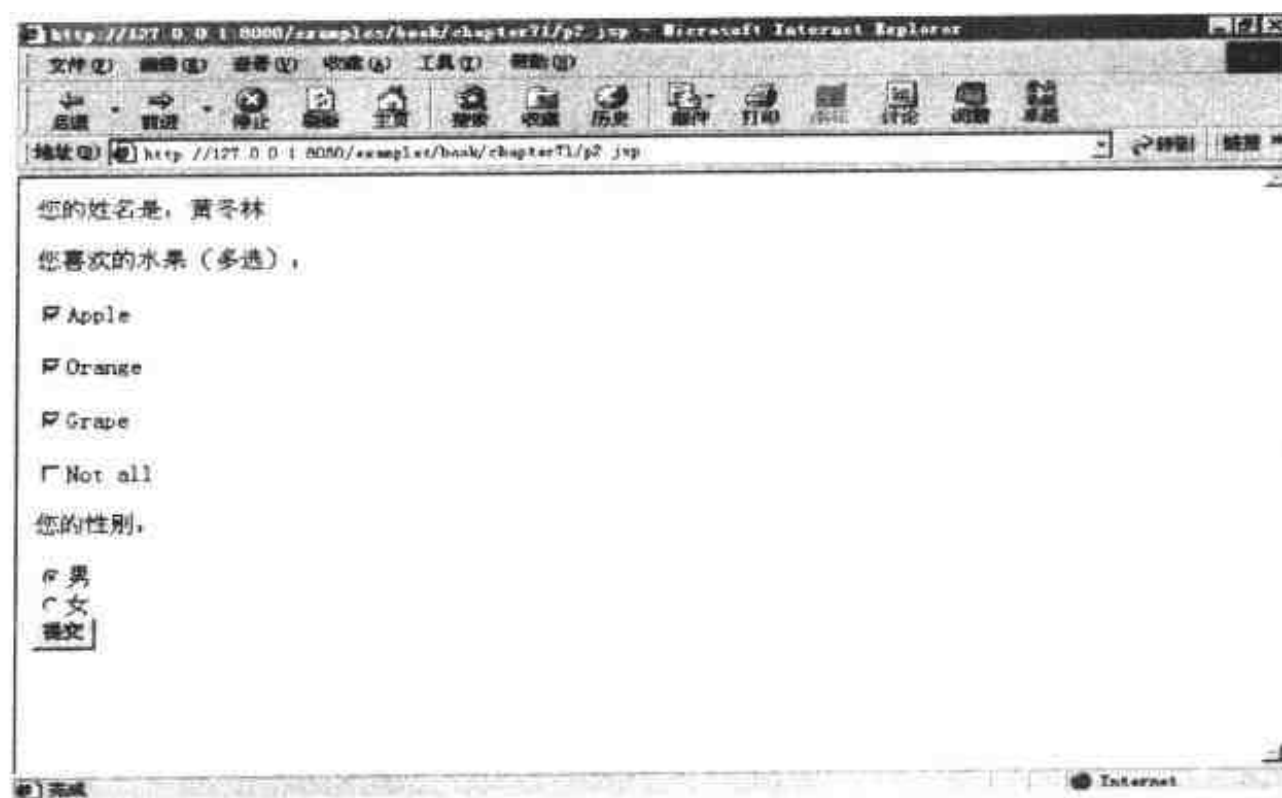


图 7-14 p2.jsp 运行后效果图

第三个页面 p3.jsp, 主要任务是显示问答结果。它从 Session 对象提取 Sname 的值并显示它, 以此证明虽然该值在第一个页面输入, 但通过 Session 对象得以保留。p3.jsp 的另外一个任务是提取在第二个页面中的用户输入并显示它:

```
<html>
<head>
<title>p3.jsp</title>
</head>
<body>
<%@page contentType="text/html;charset=gb2312"%>
<%
String msg = (String)session.getValue("Sname");
String sex = request.getParameter("sex");
String[] item = request.getParameterValues("mychoise");
int len = item.length;
if (sex == null) {
msg = msg + "没有选择性别<br>";
}
else {
if (sex.equals("1")) {
msg = msg + "先生<br>";
}
else if
msg = msg + "女士<br>";
}
}
if(len == 0) {
msg = msg + "您没有选择水果";
}
else {
out.println("<p>感谢您的选择! </p><br>");
msg = msg + "<br>您选择了:<br>";
for(int i=0;i<len;i++) {
msg = msg + item[i] + "<br><br>";
}
}
out.println(msg);
%>
</body>
</html>
```

运行后如图 7-15 所示。

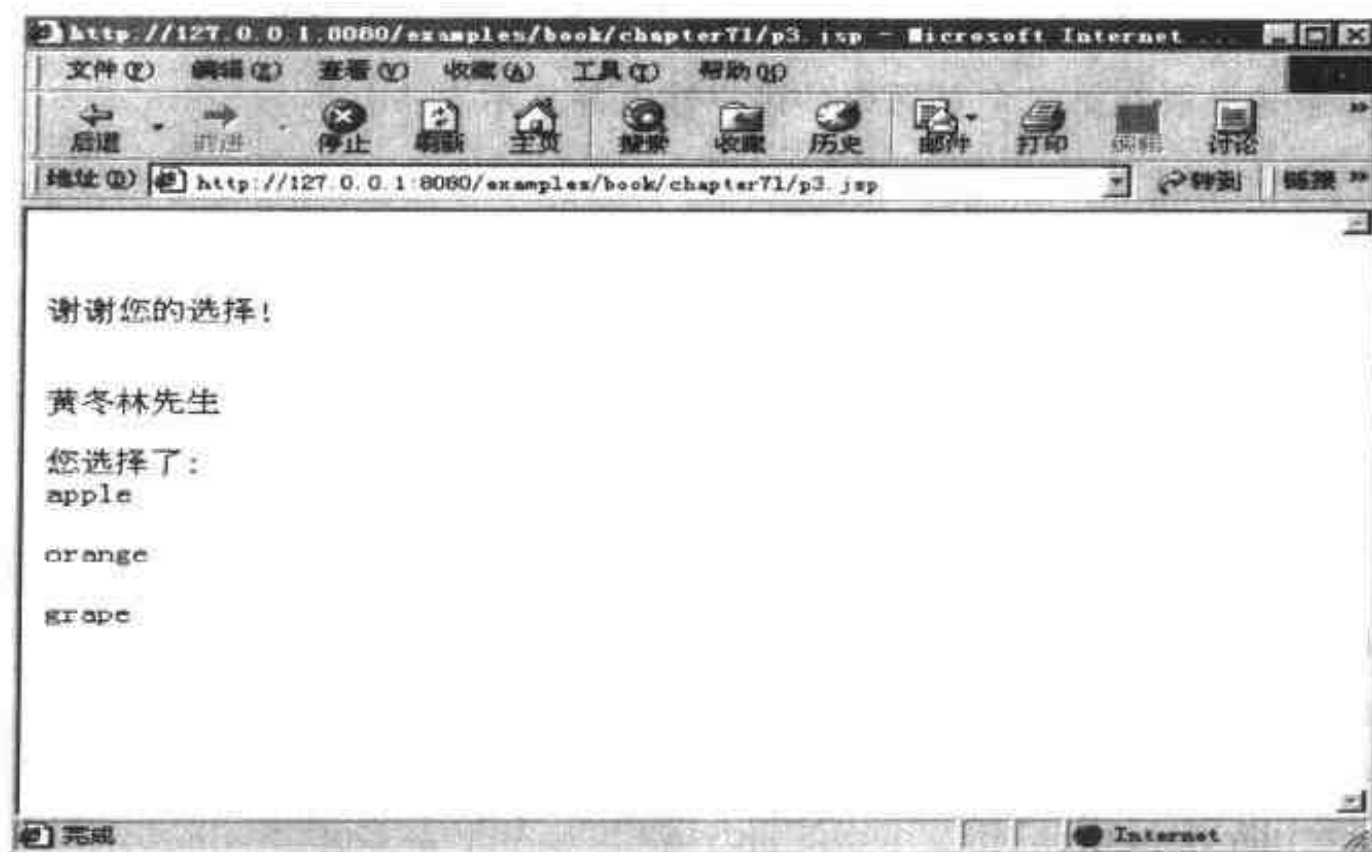


图 7-15 P3.jsp 运行后效果图

#### 例 7-9

最后给出一个用 Session 对象的 `getAttribute(String name)` 方法和 `setAttribute(String name, Java.lang.Object)` 方法实现一个简单的购物车的例子。整个处理过程由两个页面完成。shopcars.html 页面主要为用户提供物品选择, 用户可以选择物品, 然后通过点击相应的提交按钮将信息传给 shopcars.jsp 文件处理。程序中将一个向量对象通过 `setAttribute()` 方法存储在用户 Session 中形成一个用户的购物车。shopcars.htm 源代码如下(注意: 该程序有一处 bug, 请读者自行分析)。

```
<html>
<head>
<title>shopcars</title>
</head>
<body bgcolor="white">
<font size = 4 color="#CCA00">
<form type=POST action=shopcars.jsp>
<br>请你选择购买或退货
<br> <br>物品种类...
<SELECT NAME="item">
<OPTION value="0">———物品———
<OPTION value="1">音响
<OPTION value="2">VCD
<OPTION value="3">手提袋
<OPTION value="4">CD 盒
<OPTION value="5">书籍
```

```

<OPTION value="6">牙膏
<OPTION value="7">衣服
<OPTION value="8">玩具
<OPTION value="9">食品
</SELECT>
<br> <br> <br>
<INPUT TYPE=submit name="submit" value="购买">
<INPUT TYPE=submit name="submit" value="退货">
</form>
</font>
</body>
</html>

```

shopcars.jsp 处理表单提交数据,其源代码如下:

```

<html>
<head>
<title>shopcars</title>
</head>
<%@page contentType="text/html;charset=gb2312"%>
<%
//字符变量 _item 和 _sub 用于保存 request 对象获取的用户表单内容
String _item = request.getParameter("item");
String _sub = request.getParameter("submit");
//内码转换
byte[] temp _username;
String temp _suserername;
temp _suserername = _sub;
temp _username = temp _suserername.getBytes("ISO8859-1");
_sub = new String(temp _username);
//取出 session 对象中属性名为 caritem 的值(为一向量),并进行判断
if
out.println("你还没有购买物品,请购买!");
Vector newcar = new Vector();
newcar.addElement("空");
session.setAttribute("caritem", newcar);
}
else {

```

//创建一个向量对象 caritem(相当于购物车)用于接受取出 session 对象中属性名为 caritem 的值

```
Vector caritem = (Vector)session.getAttribute("caritem");
String tempcaritem = "";
//对用户所选的提交按钮进行判断
if(_sub.equals("购买")){
//用户选择了购买,将所选物品添加到 caritem 向量里,同时用 session
//对象的 setAttribute()方法存向量对象到用户 session 中,属性名为 caritem
if (_item.equals("1")){
caritem.addElement("音响");
session.setAttribute("caritem",caritem);
}
if (_item.equals("2")){
caritem.addElement("VCD");
session.setAttribute("caritem",caritem);
}
if (_item.equals("3")) {
caritem.addElement("手提袋");
session.setAttribute("caritem",caritem);
}
if (_item.equals("4")) {
caritem.addElement("CD 盒");
session.setAttribute("caritem",caritem);
}
if
caritem.addElement("书籍");
session.setAttribute("caritem",caritem);
}
if (_item.equals("6")){
caritem.addElement("牙膏");
session.setAttribute("caritem",caritem);
}
if (_item.equals("7")){
caritem.addElement("衣服");
session.setAttribute("caritem",caritem);
}
if (_item.equals("8")){
caritem.addElement("玩具");
session.setAttribute("caritem",caritem);
}
if (_item.equals("9")) {
```

```
caritem.addElement("食品");
session.setAttribute("caritem",caritem);
}
}
//若用户选择退货,先判断该退货物在用户的购物车上是否存在
if(_sub.equals("退货")){
for(int i=0;i<caritem.size();i++){
tempcaritem=caritem.elementAt(i).toString();
if(tempcaritem.equals("音响")&&_item.equals("1")){
caritem.remove(tempcaritem);
}
if(tempcaritem.equals("VCD")&&_item.equals("2")){
caritem.remove(tempcaritem);
}
if(tempcaritem.equals("手提袋")&&_item.equals("3")){
caritem.remove(tempcaritem);
}
if(tempcaritem.equals("CD 盒")&&_item.equals("4")){
caritem.remove(tempcaritem);
}
if(tempcaritem.equals("书籍")&&_item.equals("5")){
caritem.remove(tempcaritem);
}
if(tempcaritem.equals("牙膏")&&_item.equals("6")){
caritem.remove(tempcaritem);
}
if(tempcaritem.equals("衣服")&&_item.equals("7")){
caritem.remove(tempcaritem);
}
if(tempcaritem.equals("玩具")&&_item.equals("8")){
caritem.remove(tempcaritem);
}
if(tempcaritem.equals("食品")&&_item.equals("9")){
caritem.remove(tempcaritem);
}
session.setAttribute("caritem",caritem);
//重新将向量存入 session 对象中
}
}
```

```
}  
%>  
<font size = 5 color = "#CCD000">  
<br>你已经选购了下列物品:  
<%  
//取出 session 对象中的向量,若不为空将其显示到客户端  
if (session.getAttribute("caritem") == null){  
out.println("你还没有购买物品,请购买!");  
}  
else {  
Vector caritem = (Vector)session.getAttribute("caritem");  
for(int i = 0; i < caritem.size(); i++) {  
String tempcaritem = caritem.elementAt(i).toString();  
out.println("<br>" + "第" + i + "件是:" + tempcaritem);  
}  
}  
%>  
</font><br>  
//动态包含进 shopcars.html 文件。  
<jsp:include page = "shopcars.html" flush = "true" />  
</html>
```

整个运行界面如图 7-16 和图 7-17 所示。



图 7-16 shopcar.html 运行效果图

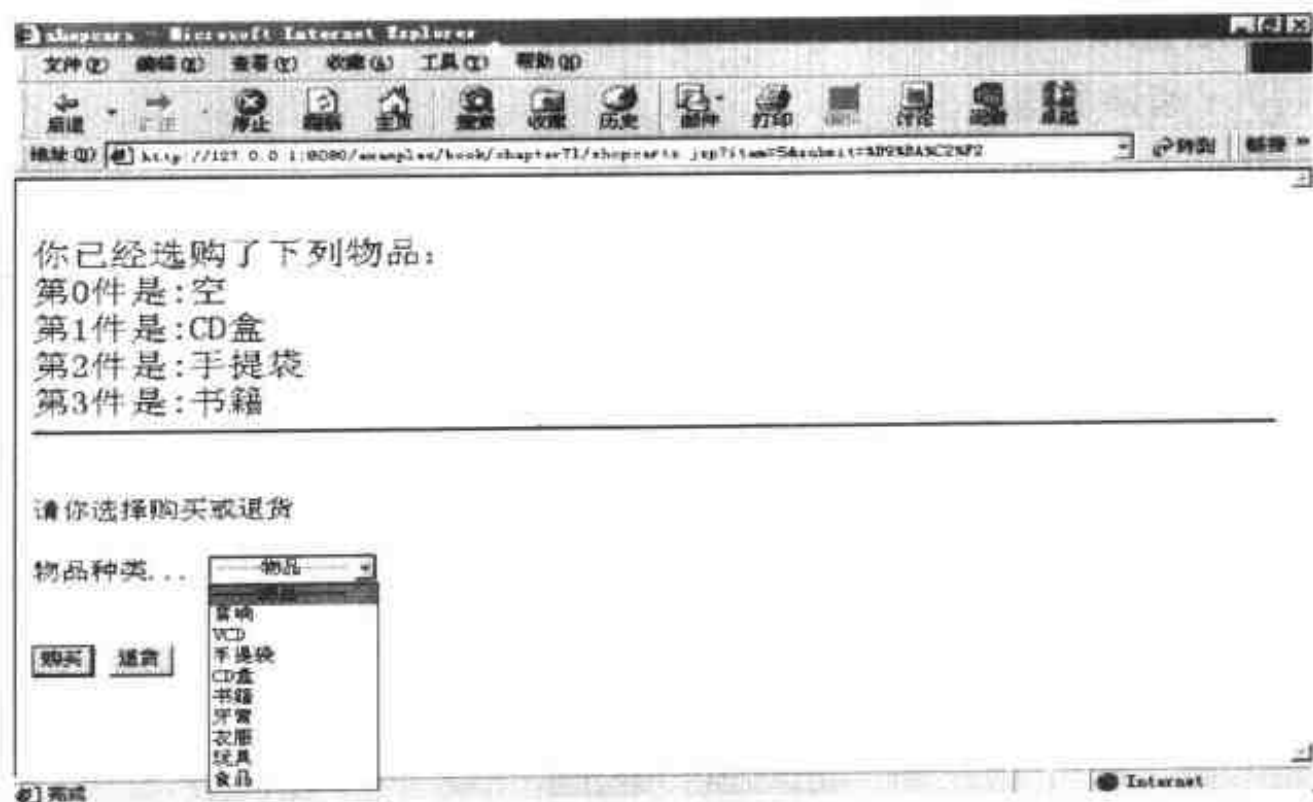


图 7-17 shopcar.jsp 运行效果图

### 7.2.5 Application 对象

如果说 Session 对象给我们提供了存储各个用户信息的手段, Application 对象则给各个用户共享信息提供了方便。该对象一旦被创建, 就会一直保持下去, 除非关闭服务器。熟悉 ASP 技术的人一定知道 global.asa 文件, 它用来记录 Session 和 Application 事件的程序或声明对象实例。在 JSP 中有些服务器也支持 global.jsa 文件, 用于完成 Application、Session 对象的构造和注销工作。

在上一节中我谈到 Session 对象时, 将它比作一把个人密匙, 与其相反 Application 则像一把共同密匙, 只要进入同一个 Web 页面内的用户都可使用这把公共的密匙。总之, Application 对象为用户共享信息提供了方便。

#### 1. 常用方法:

##### (1) `getAttribute(String name)`

返回由 name 对象指定的 Application 对象属性的值。该方法的使用与 Session 对象相同, 在 7.2.4 小节的例中讲过。

##### (2) `setAttribute(String name, Java.lang.object)`

用 object 来初始化由 name 指定的属性。

##### (3) `getAttributeName()`

返回一个枚举变量, 包含当前 Servlet 上下文中所有有效的属性名字。

##### (4) `getIntParameter(String name)`

返回由 name 指定的某个属性的初始值。

##### (5) `getInitParameterNames()`

返回一个 String 对象的枚举变量, 包含当前上下文中所有初始化参数的名字, 如果当前上下文没有初始化参数, 返回一个空的枚举变量。



**(6) getServerInfo()**

返回 Servlet 编译器信息。

**(7) getMinorVersion()**

返回当前 Servlet 容器支持的 Java Servlet API 的版本号。

**(8) getServlets()**

不赞成使用。Java Servlet API 2.0 中没有替换者。

最后要向大家说明一点,因为 Application 表示一个 javax.serve.ServletContext 对象,所以,在有些不支持 application.method(…)的服务器中,我们可以直接用 getServletContext().Method()的形式实现 Application 的功能。总之对于后一种调用方法在任何服务器上都支持,建议大家使用,可避免不必要的调试。

## 2. 例示

### 例 7-10

```
<html>
<head>
<title></title>
</head>
<body>
<%
    //声明一个字符串对象
    String welcome = new String("欢迎您们! .....");
    //声明一个向量对象
    Vector name = new Vector();
    name.addElement("黄冬林");
    name.addElement("秋秋");
    name.addElement("Tom");
    //传字符对象到 APPLICATION 对象中,属性名为"welcome"
    getServletContext().setAttribute("welcome",welcome);
    //传向量对象到 APPLICATION 对象中,属性名为" name "
    getServletContext().setAttribute("name",name);
    String _redirectURL = "/examples/book/chapter7l/file2.jsp";
    //用 encodeURL 方法对该路径进行编码,再用 sendRedirect 方法重定向
    response.sendRedirect(response.encodeURL(_redirectURL));
%>
</body>
</html>
```

在上例中我们定义了一个字符串对象和一个向量对象,用 setAttribute(String name, java.lang.object)方法将它们加到 Application 对象中,由此可见,在 Application 对象中可以保存各种各样的对象,比如字符串、向量、数组和哈希表等等。

下面的程序将上面保存在 Application 对象中的数据取出。

```
<html>
<head>
<title>welcome</title>
</head>
<body>
<%
//取出 application 对象中属性名为 welcome 的值
String _welcome = (String)getContext().getAttribute("welcome");
Vector name = new Vector();
    out.println("<h1>" + _welcome + "<h1>");
//取出 application 对象中属性名为"name"的值
name = (Vector)getContext().getAttribute("name");
for(int i = 0; i < name.size(); i++) {
String _name = name.elementAt(i).toString()
out.println(_name + "<br>");
}
%>
</body>
</html>
```

运行效果,如图 7-18 所示。



图 7-18 Application 试验运行效果图

#### 例 7-11

我们再举个用 Application 对象完成计数器的例子。计数器是一般网站必备的东西,别小看它了,每当网站维护员看着小小计数器上的数字飞速增长的时候,感觉实在是好极了。以前我们用 CGI、ASP 来写计数器,这方面的文章很多了,在这里,我们将会采用目前

比较流行的 JSP 技术演示如何做一个计数器。其代码如下：

```
<html>
<head>
<title></title>
</head>
<body>
<%! int num; %>
<%
if(getServletContext().getAttribute("numvister") == null){
getServletContext().setAttribute("numvister","0");
}
else{
num = Integer.parseInt((String)getServletContext().getAttribute("numvister"));
num = num + 1;
getServletContext().setAttribute("numvister",Integer.toString(num));
}
%>
<h1>这个 WEB 页面已经被浏览了
<% = (String)getServletContext().getAttribute("numvister") %>
次<h1>
<body>
</html>
```

在上例中,又用到了 `getAttribute()` 方法和 `setAttribute()` 方法,可见在 JSP 页面设计中 Application 对象的这两种方法用的比较普遍,希望大家能掌握。在这里将一个字符串存进 Application 中,属性名为 `numvister`。当用户访问页面时,从 Application 中取出属性名为 `numvister` 的值,并将它转换为整型,执行加 1 操作后又转化为字符串存入 Application 中。这样,每当用户访问时,`numvister` 的值就会自动加 1。

运行效果如图 7-19 所示。

这样,一个计数器就生成了。在前面已讲过 Application 对象一旦被创建,就会一直保持下去,除非服务器被关闭。那么当服务器关闭时,怎样才能保证计数的数据不丢失呢? 这里我们可以提供两种方法:

(1) 用读写文件的形式。使用 `RandoAccessFile(filename,"r")` 方法以读文件形式打开一个文本文件,接着用 `readLine()` 方法读出数据加 1 后用 `writeBytes()` 方法写回文件中,同样在使用写命令之前也必须用 `RandOAccessFile(filename,"rw")` 以写方式将文件打开。这样数据将会一直保存在文件中。

(2) 用 `gloabal.jsa` 文件形式。为了避免每次都进行文件读写操作,我们可以在 `gloabal.jsa` 文件中编写文件读和文件写的代码,分别将它们放到 `applicationInit()` 和 `applicationDestroy()` 方法中。这样只需进行一次文件读写操作,当创建 Application 对象时将读出文件中的数据,当服务器关闭或 Application 对象注销时会将数据写回,保证了数据不

会丢失。(只有部分服务器支持 global.jsa 文件)



图 7-19 计数器运行后效果图

Application 对象在 JSP 的内部对象中非常重要。请大家一定重视,在这里仅仅只谈到一部分,关于它的更多用法可参看第 14 章聊天室部分。

## 7.2.6 其它内部对象

前面几节讨论了 JSP 常用的几个内部对象,本节将对其他的几个内部对象进行介绍。

### 1.Exception 对象

Exception 对象代表了 jsp 文件运行时所产生的错误和异常的对象,它有三个方法:

(1) **getMessage()**

返回错误信息。

(2) **printStackTrace()**

输出一个错误和一个错误堆栈。

(3) **toString()**

以字符串形式返回一个错误信息。



注意:此对象不能在一般 jsp 文件中直接使用,而只能在使用了“<%@ page isErrorPage="true"%>”的 jsp 文件中使用。这是因为 jsp 文件运行时产生的错误对象被向外抛出,只能被使用了“<%@ page isErrorPage="true"%>”标记从而具有拦截错误对象功能的 jsp 文件所拦截。

Exception 对象与 page 指令结合时,可以指定某一个页面为错误处理页面,保证系统产生错误后也能正常的进行处理,同时使程序流程更加简明。

整个处理机制的描述如图 7-20 所示:

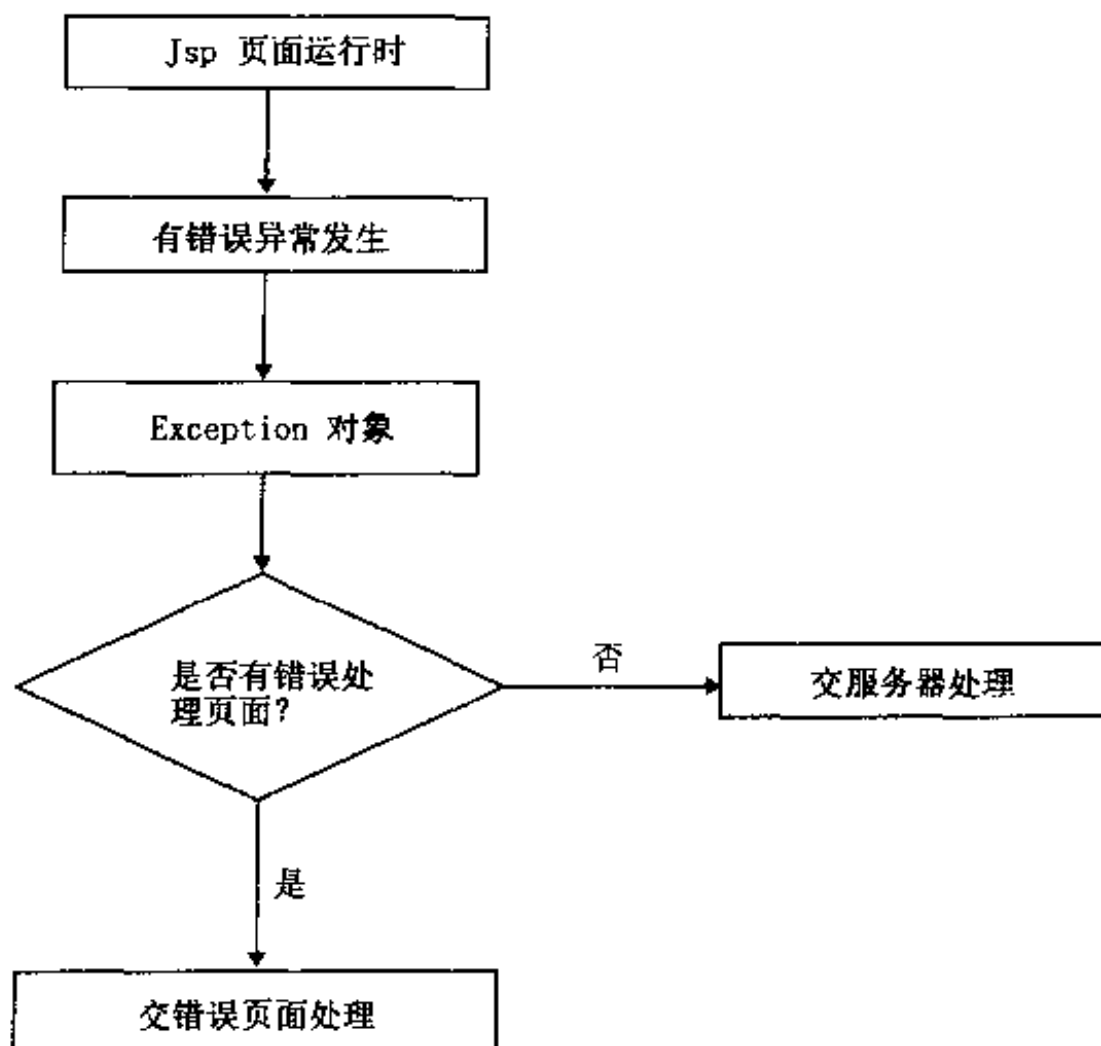


图 7-20 Exception 对象与 page 指令结合处理错误流程图

**例 7-12**

本例的两个文件组成:counter.jsp 文件以文件读写的方式实现计数器,error.jsp 文件用于处理 counter.jsp 页面中产生的错误。如果我们将文件的路径名故意改错,我们将会看到在浏览器上会给出具体的错误原因,这因为我们在 error.jsp 文件中用到了 getMessage()方法、toString()方法。counter.jsp 源程序如下:

```
<%@ page import = "java.io. *" errorPage = "error.jsp"% >
<%! int test; %>
<%
File sfile = new File("../webapps/examples/counter1/counter.txt");
//错改为:File sfile = new File("counter.txt")
RandomAccessFile sraf = new RandomAccessFile(sfile,"r");
test = Integer.parseInt(sraf.readLine());
test = test + 1;
File dfile = new File("../webapps/examples/counter1/counter.txt");
//错改为:File sfile = new File("counter.txt")
RandomAccessFile draf = new RandomAccessFile(dfile,"rw");
draf.writeBytes(Integer.toString(test));
draf.close();
%>
<html>
<head>
```

```

<title>counter</title>
</head>
<body>
<%! int i; %>
<%! String temp; %>
<% temp = Integer.toString(test);
out.println(test);
%>
This page has been visited
<% for(i=0;i<temp.length();i++) { %>
 </img>
<% } %>
times
</body>
</html>

```

错误处理程序 error.jsp 源代码如下：

```

<html>
<%@ page isErrorPage="true"%>
<head>
<title>error.jsp</title>
</head>
<body>
<%
out.println("发生了错误<br>");
out.println("exception.getMessage() = " + exception.getMessage() + "<br>");
out.println("exception.toString() = " + exception.toString() + "<br>");
out.println("exception.printStackTrace() = ");
exception.printStackTrace();
%>
</body>
</html>

```

运行后如图 7-21 所示。

## 2. PageContext 对象

PageContext 对象直译时可以称作页面上下文对象,代表的是当前页面运行的一些属性,常用的方法包括 findAttribute()、getAttribute()、getAttributeScope()和 getAttributeNamesInScope()。一般情况下“pageContext”对象用到得不是很多。下面我们给出一个 PageContext 对象的示例。

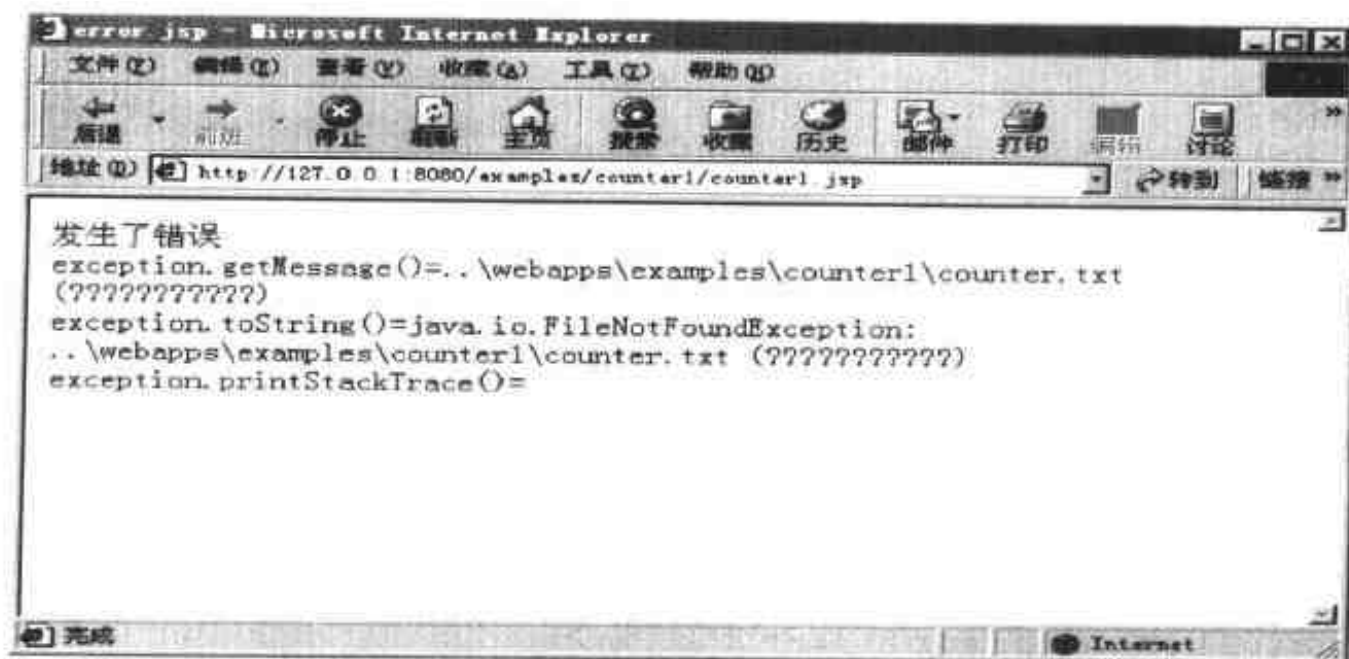


图 7-21 错误输出页面

## 例 7-13

```

<html>
<head><title></title></head>
<body>
<%
    out.println("<H2>pageContext 对象示例</H2>");
    //用 setAttribute()将属性名为"姓名"的值设为"黄东林"
    pageContext.setAttribute("姓名","黄东林");
    //用 getAttribute()取出属性名为"姓名"中的值
    out.println("getAttribute('姓名')="+pageContext.getAttribute("姓名")+"<br>");
    //按照 page、request、session (如果有效)及 application 的次序查找名为"姓名"
    //的属性,以获得相关联的属性值。
    out.println("findAttribute('姓名')="+pageContext.findAttribute("姓名")+"<br>");
    //取得属性的对象的作用范围
    out.println("getAttributesScope('姓名')="+pageContext.getAttributesScope("姓名")
    + "<br>");
    pageContext.removeAttribute("姓名");
    out.println("删除后,getAttribute('姓名')="+pageContext.getAttribute("姓名")+"<br>");
%>
</body>
</html>

```

运行效果,如图 7-22 所示。

最后说明一点,在讲 Application 对象时谈到可用 `getServletContext()` 方法得到一个 Application 对象,同样,PageContext 对象通过使用 `getRequest()`、`getSession()`、`getResponse()`、`getPage()`、`getOut()`、`getServletConfig()` 方法也可以间接的得到其它的内部对象。

### 3. Config 对象

Config 对象提供一些配置信息,常用的方法有 `getInitParameter()` 和 `getInitPa-`

parameterNames(), 以获得 Servlet 初始化时的参数。



图 7-22 pageContext 对象的示例效果图

#### (1) GetServletContext()

ServletContext 对象是服务器传给这个 Servlet 的。在 ServletConfig 接口定义中 ServletContext 对象是其中的一部分。

返回值: 一个 ServletContext 对象, 它能给出 Servlet 如何与服务器交互的信息。

#### (2) GetInitParameter()

获得命名过的初始化参数的值, 如果此参数不存在则返回空值。此初始化参数的值只是一个字符串, 因此必须进行处理。

参数: name 包含参数名称的字符串。

返回值: 一个字符串, 表示参数的值。

#### (3) GetInitParameterNames()

通过此方法来获得 Servlet 的初始化参数名称。

返回值: 一个枚举对象, 包括 Servlet 的初始化参数名称, 若无参数则返回空值。

### 4. Page 对象

Page 对象代表了正在运行的由 JSP 文件产生的类对象, 一般不建议读者使用。但通过使用, 可有助于更好的了解 JSP 服务器的运行原理。Page 对象主要方法有。

#### (1) GetClass()

获得对象的运行时类。这个 Class 对象是被这个类通过静态同步方法锁定的。

返回值: 当前对象的运行时类的类型。

#### (2) GetHashCode()

获得对应此对象的哈希码值。可以提供给诸如 java.util.Hashtable 类型的对象使用。

返回值: 对应此对象的哈希码值。

#### (3) Equals(Object obj)

用来判断其它对象是否与此对象相等。



参数:obj 将进行比较的对象引用。

返回值:布尔值,相等时返回真,否则为假。

#### (4) clone

创建并返回当前对象的拷贝。

#### (5) toString

获得表达此对象的字符串。一般情况下,此方法返回此对象的文字型描述。

返回值:一个字符串,以表达此对象。

#### 例 7-14

page 对象的使用。

```
<html>
<head><title>config</title></head>
<body>
<%
    out.println("page.getClass()=:<br>" + page.getClass() + "<br><br>");
    out.println("page.hashCode()=" + page.hashCode() + "<br><br>");
    out.println("page.toString()=" + page.toString() + "<br>");
%>
</body>
</html>
```

运行效果,如图 7-23 所示。

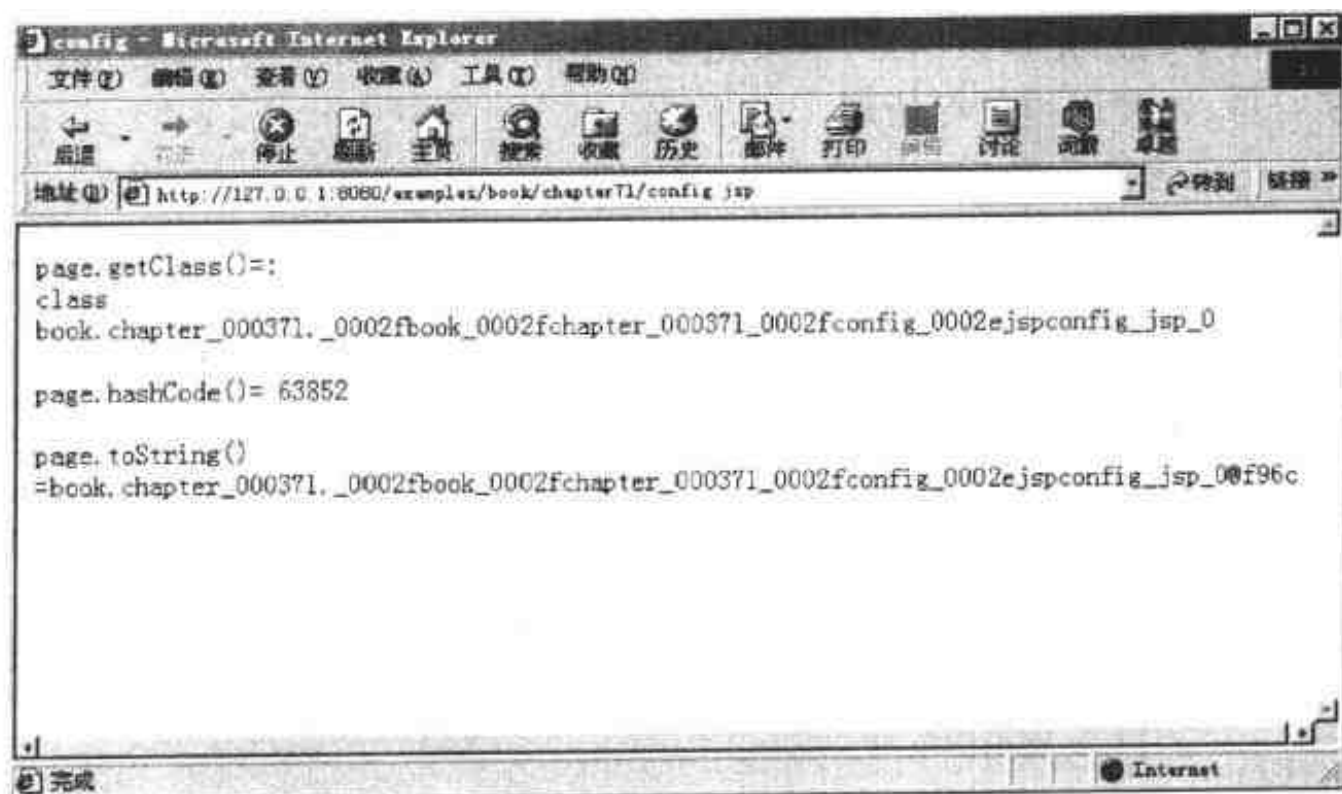


图 7-23 config.jsp 运行效果图

## 7.3 还想多了解点吗

通过本章的学习,能对 JSP 的内部对象有比较深入的了解,但是,为什么这些对象不用显式声明,也无需创建其实例就可以直接使用呢?通过这一节的学习,可以回答这个问

题。首先,请看下面代码:

```
public void _jspService(HttpServletRequest request,
    *           HttpServletResponse response)
    *           throws IOException, ServletException {
    *
    *   JspFactory factory      = JspFactory.getDefaultFactory();
    *   PageContext pageContext = factory.getPageContext(
    *       this,
    *       request,
    *       response,
    *       null, // errorPageURL
    *       false, // needsSession
    *       JspWriter.DEFAULT_BUFFER,
    *       true // autoFlush
    *   );
    *
    *   // initialize implicit variables for scripting env ...
    *
    *   HttpSession session = pageContext.getSession();
    *   JspWriter out       = pageContext.getOut();
    *   Object page        = this;
    *
    *   try {
    *       // body of translated JSP here ...
    *   } catch (Exception e) {
    *       out.clear();
    *       pageContext.handlePageException(e);
    *   } finally {
    *       out.close();
    *       factory.releasePageContext(pageContext);
    *   }
    * }
```

上面的代码定义了一个 `_jspService()` 方法, 该方法是 `HttpJspPage` 界面的一个方法。`HttpJspPage` 界面继承了 `JspPage` 界面, 主要用于 JSP 引擎产生满足 Http 协议的类, 而 `_jspService()` 方法就相当于 Jsp 页面的主体, 首先通过 `JspFactory` 类的 `getDfaultFactory()` 方法创建一个 `JspFactory` 对象, 再由该对象的 `getPageContext()` 方法创建 `PageContext` (页面上下文对象) 即 `javax.servlet.jsp.PageContext` 抽象类的一个实例。通过 `PageContext` 对象的

方法间接的获得其它的内置对象。通过 JSP 引擎可以自动的执行 `_jspService()` 方法无需 JSP 设计者亲自调用。所以,我们不用显式申明,也无需创建其实例就可以直接使用。

## 7.4 本章小结

本章全面介绍了 JSP 的 9 种内置对象,给出了每一个对象所具有的方法及相应的使用说明。熟练使用这些内置对象是开发 JSP 应用程序的基本要求,尤其是对于 Request、Response、Application、Session 和 Out 对象要熟练掌握。所谓掌握对象的使用也就是掌握对象所具有的方法的使用,每一种对象都有几种常用的方法,详见相应说明及对应的示例。

## 第 8 章 JSP Container

为什么 JSP 功能如此强大而使用却如此方便,为什么 JSP 可以集成如此众多的技术而自身却非常简单,等等。俗话说得好,无欲则刚,有容乃大!答案就落在“容”字上。容器是 JSP 容器赋予了 JSP 矛盾的特质。容器是 Sun 的专有名词,可以把它理解为提供某种服务的实体。JSP 容器继承自 Servlet,它是一个系统级实体,为 JSP 页面和 Servlet 组件提供生命周期管理和运行时支持。容器对 JSP 乃至所有基于 Java 的技术都是一个非常重要的概念,因此它是本书内容上的一个核心。

为什么 JSP 编写的程序不需要编译就能执行,而其它的 Java 技术需要编译,页面指令怎样作用在 JSP 页面上,单线程到底是怎样实现的,包含指令与包含行为到底有什么区别?下面让我们揭开谜底——Servlet 级的系统实体——JSP 容器。在讲述容器之前,还要简单的看看编写 JSP 的整个过程及其注意事项。当然本章的落脚点是看从运行角度看 JSP 技术是什么?

### 8.1 编写支持实例

要获得一个可运行的 JSP 程序,首先需要编写,其次存档,再次调试错误处理,最后才能完整的翻译执行。现在首先编写一个程序,它是本章后面比较工作的支持程序。

#### 例 8-1

首先实现的功能是自动生成 JSP 文档,然后实现提交后显示提交的 JSP 文档的页面效果及其源代码。自动生成的 JSP 文件是 sample8\_1\_11.jsp,可在页面上显示源代码的文件是 sample8\_1\_11.txt。

用前面讲述的 Dreamweaver 和 HTML 知识实现这个页面。页面很容易实现,需要注意的是表单提交数据几乎都是 JSP 的页面指令。需要仔细的考虑每个单选按钮或是下拉菜单的属性值。还有一点是导入包名是允许多重选择的,需要用列表实现。这里列出部分的参考代码如下。

```
<form method="post" action="sample8.jsp">
<table width="750" border="0" cellspacing="0" cellpadding="0" align="center">
  <tr>
    <td>
      <div align="right">session 属性:</div>
    </td>
```

```

        <td>
            <input type="radio" name="rbsession" value="true" checked>true
        </td>
        <td>
            <input type="radio" name="rbsession" value="false">false
        </td>
        <td>
            <div align="right">import 属性:</div>
        </td>
        <td rowspan="4" valign="top">
            <select name="slimport" size="6" multiple>
                <option value="java.applet.*">applet</option>
                <option value="java.beans.*">beans</option>
                <option value="java.io.*">io</option>
                <option value="java.math.*">math</option>
                <option value="java.sql.*">sql</option>
                <option value="java.util.*">util</option>
                <option value="java.text.*">text</option>
            </select>
        </td>
    </tr>
    <tr>
        <td>
            <div align="right">autoFlush 属性:</div>
        </td>
        <td>
            <input type="radio" name="rbautoFlush" value="true" checked>true
        </td>
        <td>
            <input type="radio" name="rbautoFlush" value="false">false
        </td>
    </tr>
    <tr>
        <td>
            <div align="right">isErrorPage 属性:</div>
        </td>
        <td>
            <input type="radio" name="rbisErrorPage" value="true">true

```

```

        </td>
        <td>
            <input type="radio" name="rbisErrorPage" value="false"
                checked>false
        </td>
    </tr>
    <tr>
        <td>
            <div align="right">isThreadSafe 属性:</div>
        </td>
        <td>
            <input type="radio" name="rbisThreadSafe" value="true" checked>true
        </td>
        <td>
            <input type="radio" name="rbisThreadSafe" value="false">false
        </td>
    </tr>
    <tr>
        <td colspan="3">
            <div align="right">info:
                <input type="text" name="txtinfo" size="30">
            </div>
        </td>
        <td>
            <div align="right">contentType 属性:</div>
        </td>
        <td>
            <select name="slcontentType">
                <option value="text/html;charset=ISO-8859-1">默认国际标准
            </option>
                <option value="text/html;charset=gb2312" selected>简体中文</option>
            </select>
        </td>
    </tr>
    <tr>
        <td colspan="3">
            <div align="right">errorPage:
                <input type="file" name="fileerrorPage" size="15">
            </div>
        </td>
    </tr>

```

```

</td>
<td>
    <div align="right">buffer 属性:</div>
</td>
<td>
    <select name="slbuffer">
    <option value="none">none</option>
    <option value="1">1k</option>
    <option value="8" selected>8k</option>
    <option value="16">16k</option>
    <option value="32">32k</option>
    </select>
</td>
</tr>
.....
</table>

```

页面效果如图 8-1 所示。

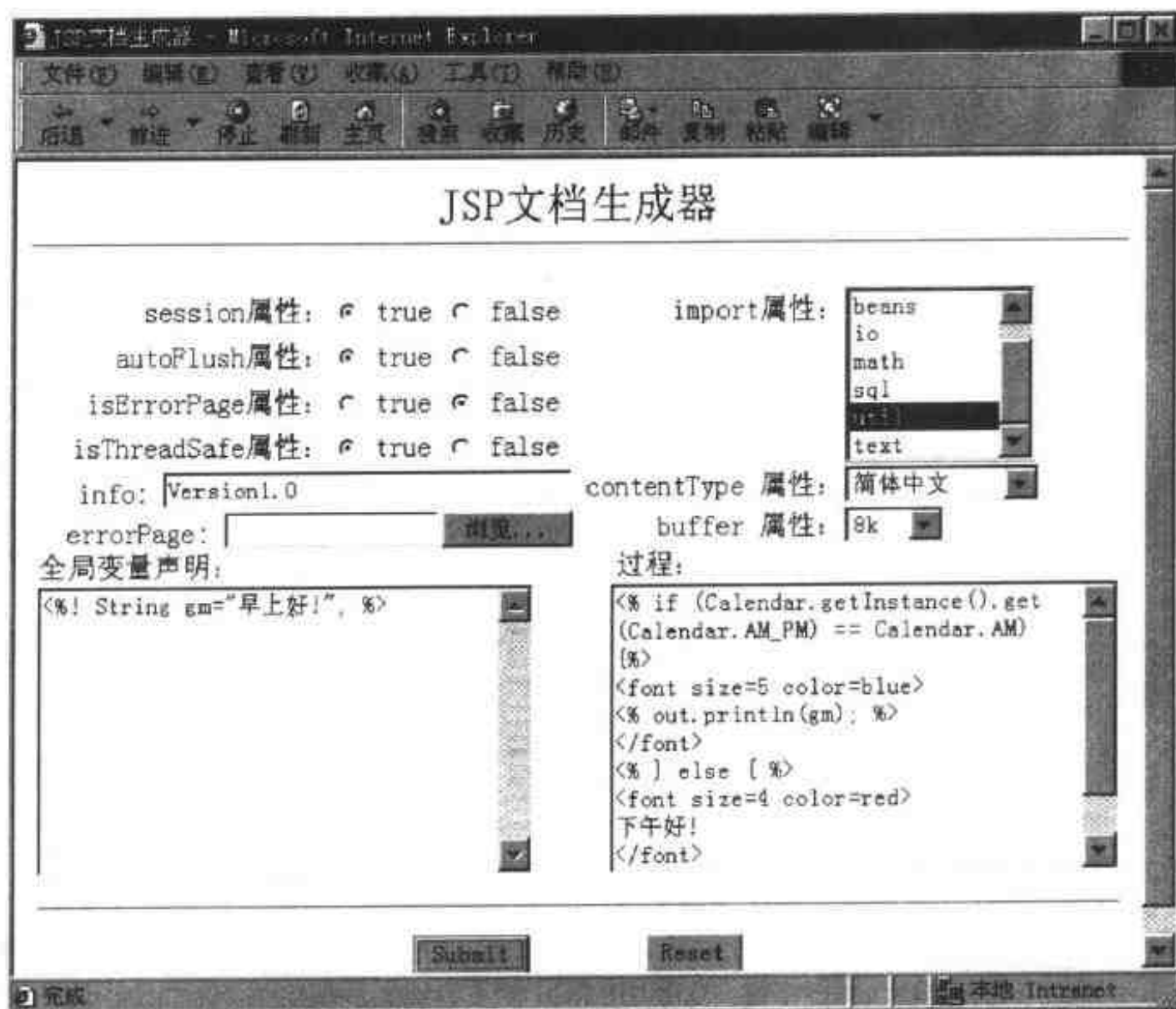


图 8-1 JSP 文档生成器

假设已经完成这个页面,下面就来实现处理这个页面的 JSP 程序。这是典型的表单提交技术,前面已经应用得很多了,下面列出源代码,稍后讲解其中的重要知识点。

```
<%@ page import = "java.io. *" %>
<%@ page contentType = "text/html; charset = gb2312" %>
<%! boolean EOF = false; %>
<%
    try{
        FileWriter fw =
            new FileWriter("../webapps/examples/book/chapter08/sample81.jsp");
        PrintWriter pw = new PrintWriter(fw);
        String myimport[] = request.getParameterValues("slimport");
        String myautoFlush = request.getParameter("rbautoFlush");
        String mybuffer = request.getParameter("slbuffer");
        String myinfo = request.getParameter("txtinfo");
        String myfileerrorPage = request.getParameter("fileerrorPage");
        String mysession = "session = \"\" + request.getParameter("rbsession") + "\"";
        String mycontentType =
            "contentType = \"\" + request.getParameter("slcontentType") + "\"";
        String myisErrorPage =
            "isErrorPage = \"\" + request.getParameter("rbisErrorPage") + "\"";
        String myisThreadSafe =
            "isThreadSafe = \"\" + request.getParameter("rbisThreadSafe") + "\"";
        String myvariable = new String(
            request.getParameter("txtvariable").getBytes("ISO-8859-1"));
        String myprocedure = new String(
            request.getParameter("txtprocedure").getBytes("ISO-8859-1"));

        if(mybuffer.equals("none") & myautoFlush.equals("false")){
            out.println("buffer 属性与 autoFlush 属性非法组合!");
        }
        else{
            String endbuffer = "buffer = \"\" + mybuffer + "\"";
            String endautoFlush = "autoFlush = \"\" + myautoFlush + "\"";
            String endbufFls =
                "< %@ page \" + endbuffer + \" \" + endautoFlush + \" % \\\\ >";
            pw.println(endbufFls);

            if(myimport! = null){
                String endimport = "< %@ page import = \"\" + myimport[0];
                for(int i = 1; i < myimport.length; i ++ ){
                    endimport = endimport + ",";
                }
            }
        }
    }
}
```



```

        endimport = endimport + myimport[i];
    }
    endimport = endimport + "\ " % \ \ >";
    pw.println(endimport);
}
String endcontentType = "< %@ page " + mycontentType + " " + "% \ \ >";
pw.println(endcontentType);
String endsessThd =
    "< %@ page " + mysession + " " + myisThreadSafe + " " % \ \ >";
pw.println(endsessThd);
String mypage = "< %@ page";
if(! myfileerrorPage.equals("")){
    mypage = mypage + " " + "errorPage = \ "" + myfileerrorPage + " \ """;
}
mypage = mypage + " " + myisErrorPage + " " % \ \ >";
pw.println(mypage);
if(! myinfo.equals("")){
    pw.println("< %@ page info = \ "" + myinfo + " \ "" + " " % \ \ >");
}
}

pw.println(myvariable);
pw.println("<HTML>");
pw.println("<HEAD>");
pw.println("<TITLE></TITLE>");
pw.println("</HEAD>");
pw.println("<body>");
pw.println(myprocedure);
pw.println("</body>");
pw.println("</HTML>");
pw.close();
fw.close();

File from = new File("../webapps/examples/book/chapter08/sample81.jsp");
FileInputStream fis = new FileInputStream(from);
File to = new File("../webapps/examples/book/chapter08/sample81.txt");
FileOutputStream fos = new FileOutputStream(to);
DataOutputStream dos = new DataOutputStream(fos);
String dosstring;
int chl;

```

```
int ch2;

while((ch1 = fis.read()) != -1){
    if((ch2 = fis.read()) != -1){
        if(ch1 != '<' && ch1 != '>' && ch2 != '<' && ch2 != '>'){
            char c = (char)((ch1 << 8) + (ch2 << 0));
            dos.writeChars(new Character(c).toString());
        }
    }
    else{
        switch(ch1){
            case '<':
                dosstring = "&lt;";
                dos.writeChars(dosstring);
                break;
            case '>':
                dosstring = "&gt;<br>";
                dos.writeChars(dosstring);
                break;
            default:
                dos.writeInt(ch1);
        }
        switch(ch2){
            case '<':
                dosstring = "&lt;";
                dos.writeChars(dosstring);
                break;
            case '>':
                dosstring = "&gt;<br>";
                dos.writeChars(dosstring);
                break;
            default:
                dos.writeInt(ch2);
        }
    }
}
else{
    switch(ch1){
        case '<':
```

```

        dosstring = "&lt;";
        dos.writeChars(dosstring);
        break;
    case '>':
        dosstring = "&gt;<br>";
        dos.writeChars(dosstring);
        break;
    default:
        dos.writeInt(ch1);
    }
}

dos.close();
fis.close();
fos.close();
}
catch(EOFException e){
    EOF = true;
}
catch(FileNotFoundException e){
    out.println("file not found" + e);
}
catch(IOException e){
    out.println("io error" + e);
}
}

% >
<HTML>
<HEAD>
<TITLE>程序结果</TITLE>
</HEAD>
<body>
<font size=5 color=red>
不想知道结果吗? 结果是:
</font>
<hr>
    <jsp:include page="sample81.jsp" flush="true" />
<p> &nbsp;</p>
<hr>

```

源代码如下:<br>

```
<jsp:include page="sample81.txt" flush="true" />
</body>
</HTML>
```

本例中有以下几个知识点:

(1) 对表单提交的数据需要分类进行处理,至少要分为三类:一是还需要处理的数据如 buffer 属性、autoFlush 属性、import 属性直接获得其值。二是不需要处理的数据直接与其相应的属性结合,生成最终需要的字符串,例如对 session 属性的处理, String mysession="session = \" + request.getParameter("rbsession") + "\"。当然也可以直接在表单提交数据中指定简化这步操作。三是需要进行代码转换的属性,如变量声明、过程等,使用下面的代码将单字节编码字符串转换成双字节编码字符串,也就保证了汉字的正确显示。

```
String myvariable=
    new String(request.getParameter("txtvariable").getBytes("ISO-8859-1"));
```

(2) 处理不允许出现的属性组合,如 buffer 与 autoFlush 属性的非法组合。只有当没有非法组合时,才允许下面的操作执行。代码如下:

```
if(mybuffer.equals("none") & myautoFlush.equals("false")){
    out.println("buffer 属性与 autoFlush 属性非法组合!");
}
else{
    .....
}
```

(3) 因为 import 属性是允许多重选择的,因此需要处理一个数组获得 import 的所有值。当然前面获得该属性值应当使用 request.getParameterValues(), 该方法获得指定参数的所有值。对 import 属性处理如下:

```
String myimport[] = request.getParameterValues("slimport");
.....
if(myimport != null){
    String endimport = "< %@ page import = \" + myimport[0];
    for(int i = 1; i < myimport.length; i++){
        endimport = endimport + ",";
        endimport = endimport + myimport[i];
    }
}
```

(4) 将上面处理过的属性字符串写入文件,这里直接假定为 sample8\_1\_1.jsp,事实上,完全可以从表单提交中获得文件名,这样处理的灵活性更大。使用 PrintWriter 对象的 println() 方法将属性字符串写入文件。例如:

```
pw.println(myvariable);
```

PrintWriter 类是非常有用的,希望读者仔细看看这个类的方法。

(5) 最后使用 `DataOutputStream` 类将 `sample8_1_1.jsp` 中的数据倒入 `sample8_1_1.txt` 中,当然数据是经过转义处理的,将所有“<”和“>”转换成“&lt;”和“&gt;”,达到显示源代码的目的。这种转换由 `while` 循环包含的代码实现。读者有兴趣的话可以看看,但是还有更好的方法实现字符串替换,请参考本章 8.7 节留言板的实现。事实上,我们一共写了三种实现方法,这种是比较低级的实现。

编写了这么多 JSP 程序,各位读者想必都有自己的心得体会了,但是应注意编写 JSP 程序的基本事项。



注意:(1)大小写区分,包括类名、文件名、变量、表单属性名等等。

(2)`import`、`package` 的位置。如果有 `package`,那么它一定是该文档第一行有效代码。`import` 位置也应尽量靠前,倒是 `import` 的个数没有限制。

(3)分号问题,除了表达式,其它语法都是以分号结尾的。

(4)路径问题,如果大量的使用包含指令或包含行为,那么一定要有足够的心理准备处理路径问题。

(5)使用 `include` 指令包含的文件和被包含文件不能同时出现 `contentType` 属性。

本书其它地方也有一些注意事项。这些都是最基本的注意事项,一不留神就容易出错。

写完程序需要存档,那么 Sun 在这方面有什么建议呢?请看下一节。

## 8.2 命名约定

在页面应用程序中,JSP 页面被打包为一个或多个文件,然后交付给一个工具,如一个 JSP 容器、一个 J2EE 容器或一个 IDE,在某些情况下,单独的一个文件也能包含一个完整的 JSP 页面;在另外一些情况下,顶层文件能够包含一个或多个文件,被包含文件或是完整的 JSP 页面,或仅仅是程序片段。

对工具来说,通常要将包含 JSP 页面的文档与其它文档相区分,也要区分顶层 JSP 文档和包含片段。举个例子,一个程序片段可能不是合法的 JSP 文档,那么它就不能正确的被编译。从文档编写与维护的角度看,确定文件的类型也是非常有用的。例如人们熟知用 `.c`、`.h` 作为 C 程序设计语言的约定。

`Servlet2.3` 规范中用扩展名 `.jsp` 表示 JSP 页面,但它并不能真正区分顶层 JSP 文档和包含片段。与 `Servlet` 规范立场一致,强制实行任何特定的约定都不可能,因此 Sun 推荐:

(1)“`.jsp`”文档与顶层 JSP 文档相联系,它可以包含其它 JSP 页面。

(2) 包含片段不采用“`.jsp`”扩展名,任何其他扩展名均可。但是 Sun 仍然推荐了“`.jspf`”、“`.jsf`”作为包含片段的扩展名。

页面是否编写成功,需要运行观看结果。传统的程序设计语言大都需要编译,JSP 也支持编译,尽管通常我们都不编译。

## 8.3 编译

### 8.3.1 编译

JSP 页面可以被编译成 JSP 页面实现类加一些展开信息。这使得用 JSP 页面开发工具和 JSP 标记库开发 Servlet 成为可能。将 JSP 编译成 Servlet 有以下两个优点：一、免除了 JSP 页面第一次接到请求以原程序方式提交的启动落后时间。二、减少了运行 JSP Container 的空间，因为不再需要 Java 编译器。

如果一个 JSP 执行类依赖于某些附加的 JSP 和 Servlet 类的支持。那么这些支持类必须包含在 WAR 中，这样，支持类就可以方便地在所有 JSP Container 移植使用。

JSP 页面在 Web Application 的上下文环境中被编译，上下文环境提供相对 URL 规范所需的资源。这些 URL 通常由 include 指令、Taglib 引用和解释时的自定义行为指定。

### 8.3.2 预编译

使用 HTTP 协议的 JSP 页面接受 HTTP 请求，所有符合 JSP 规范的 Container 必须支持一个简单的预编译协议和一些基本的保留参数名。注意预编译协议不要与将 JSP 页面编译为一个 Servlet 类的概念相混淆。

#### 1. 请求参数名

所有以 jsp 为前缀的请求参数名被 JSP 规范所保留。除非规范指明，否则任何用户及实现都不能使用。所有 JSP 页面将忽略任何以 jsp 开始的参数。

#### 2. 预编译协议

对 JSP 页面带有参数“jsp-precompile”的请求是一个预编译请求。“jsp-precompile”参数没有值或者有值“true”、“false”。所有情况下，请求都不会提交给 JSP 页面。

预编译请求的意图是提示 JSPContainer 将 JSP 页面预编译为 JSP 页面实现类。这个提示是由参数值“真”或无值传递的。但是这两种情况下的请求都可以被忽略。

例：

- (1) ? jsp-precompile
- (2) ? jsp-precompile = “true”
- (3) ? jsp-precompile = “false”
- (4) ? foobar = “foobar” & jsp-precompile = “true”
- (5) ? foobar = “foobar” & jsp-precompile = “false”
- (6) ? jsp-precompile = “foo”

1,2,4 是正确的，请求不会传递给页面。3,5 也是正确的，请求将不被改变地传递给

页面。6 是违法的,将产生 HTTP 500 错误状态码。

## 8.4 调试和错误处理

### 8.4.1 调试

大量开发环境支持原码级的 JSP 页面调试。如果不喜欢工具软件,就在服务器上调试也未尝不可。事实上,本书的例子就是这样调试出来的。

### 8.4.2 错误处理

错误可能发生在翻译时或在请求时,本小节介绍这些错误怎样由相应的实现处理。

#### 1. 翻译时错误处理

JSP Container 使用 Java 技术将 JSP 页面源代码翻译成相应的 JSP 页面实现类发生在下面两个时刻之间:一、JSP 页面初始化展开到 JSP Container 运行环境中;二、接到并处理客户请求的目标 JSP 页面。

如果翻译先于 JSP Container 接到客户端对目标 JSP 页面的请求,那么错误处理与通知是实现相关的。致命翻译错误将导致后续的客户对翻译目标的请求也因相应的错误面失致。对 HTTP 协议,错误状态码为 500。

#### 2. 请求时错误处理

在处理客户请求过程中,运行错误可能发生在 JSP 页面实现类体中或发生在 JSP 页面实现类体中调用的其它代码中(Java 或其它实现程序语言)。这样的错误使用 Java 程序设计语言中的异常机制,由页面实现识别并通知它们的调用者。在 JSP 页面实现体中这些异常可以被捕捉和处理。然而 JSP 页面实现类体中抛出的任何不可捕捉异常将导致客户请求和不可捕捉异常转发给特定的 errorpage URL,或与实现相关的默认行为。

`java.lang.Throwable` 描述发生的错误并把它存储在 `javax.servlet.ServletRequest` 实例中。如果 `errorPage` 属性指定的 URL 指向另一个 JSP 页面,并且那个 JSP 页面表明自己是一个错误处理页面(设置 `page` 指令的 `isErrorPage="true"`),那么页面脚本语言内部对象 `Exception` 是已初始化的 `Throwable` 的引用。

## 8.5 翻译执行

JSP 页面是文本组件,它们要经过两个阶段:翻译阶段和请求阶段。翻译阶段每个页面只执行一次,请求阶段发生在每次请求时。

翻译阶段, Container 建立给定页面的实现类。这个处理决定于 JSP 页面的语义, 标准指令和行为的语义, 以及使用在页面中的标记库的自定义行为的语义。标记库能随意的提供一个转换以扩展翻译阶段, 并且有确认方法保证 JSP 页面正确的使用标记库。

翻译阶段的结果是一个 Servlet 类的建立: JSP 页面实现类在请求时被实例化, JSP 页面实现对象可以处理请求建立应答。

翻译阶段提前是可能的(有时称将 JSP 页面编译为 Servlets)并且以一个 Web Application 提交(预编译结果), 很明显一个 Servlet 类与 JSP 页面的文本表示法有相同的行为。

翻译阶段也可在 JSP Container 展开时执行, 或者请求到达一个还未被翻译的 JSP 页面, 面又要立即响应时, 即原码页面的翻译可能发生在页面初始化展开为运行时环境, 和接到并处理对目标 JSP 页面的请求之间。

## 8.6 容器

### 8.6.1 什么是 JSP 容器

JSP Container 是一个系统级实体, 它为 JSP 页面和 Servlet 组件提供生命周期管理和运行时支持。请求阶段, JSP Container 将对某个 JSP 页面的请求, 提交给适当的 JSP 页面实现类。执行阶段, JSP Container 提交事件给 JSP 实现对象。Container 还负责实例化 Request、Response 对象。

JSP Container 负责两种独立行为。一是决定一个给定 JSP 页面其对应的实现类。另一个是管理这个类的一个或多个实例, 以响应请求或其它事件。

所有 JSP Container 必须支持 HTTP 协议的 Request 和 Response 对象。但是一个 Container 也可以支持额外的 Request/Response 协议。默认的 Request 和 Response 对象分别是类型 HttpServletRequest 和 HttpServletResponse 的实例。

### 8.6.2 JSP 页面与 JSP 容器的关系

JSP 页面表现为页面实现类, 它由一个 JSP Container 执行。当这个 JSP 页面实现类执行时, JSP Container 则提交对该 JSP 页面实现类的客户请求并接收 JSP 页面实现类对客户作出的响应。Container 可能会做进一步的处理。

#### 1. 从页面服务角度看协议

JSP Container 的角色是首先定位 JSP 页面实现类的适当实例, 然后根据 Servlet 协议提交对它的请求。当定位在其它地方(非本地), JSP Container 可能需要在提交对它的 Request 和 Response 对象之前根据 JSP 页面原码动态的创建一个类。因此, Servlet 定义了 JSP Container 与 JSP 页面实现类的关系。当使用 HTTP 协议时, 这种关系表现为 HttpServlet 类。大多数页面使用 HTTP 协议, 但是 JSP 规范里也允许其它协议。



## 2. 从 JSP 页面作者看协议

这个 JSP 规范也定义了 JSP Container 与 JSP 页面作者之间的关系,这种关系有利于页面作者的页面描述行为。

这种关系的主要部分是 JSP Container 根据 JSP 页面自动的产生 `_jspService()` 方法。

这种关系也描述了页面的 `init()`, `destory()` 方法执行时, JSP 页面作者怎样实现那些必须发生的行为。在 JSP 中,这些通过定义名为 `jspInit()` 和 `jspDestory()` 方法来完成。在第一次请求被提交给页面之前,如果存在 `jspInit()` 将调用以准备页面, JSP Container 可以调用一次它的 `jspDestory()` 方法(如果存在),以收回那些由 JSP 页面占用的,但不再用于 JSP 页面服务的资源。

## 3. HttpJspPage 接口

JSP 页面相应的 Servlet 类必须实现 `IHttpJspPage` 接口,这种需要实现了 JSP Container 和 JSP 页面作者之间的强制关系。图 8-2 表示了 JSP 页面与 JSP 容器之间的关系。

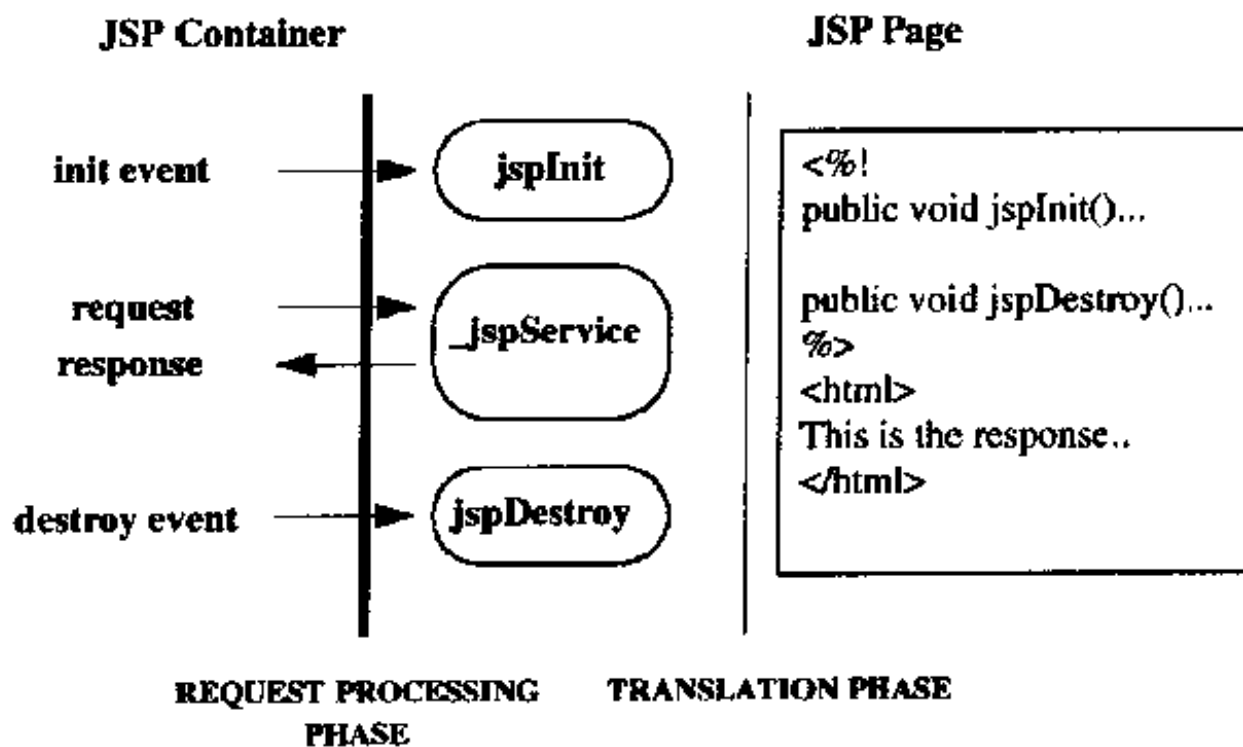


图 8-2 JSP 页面与 JSP 容器间的关系。

### 8.6.3 JSP 页面实现类

JSP Container 为每一个 JSP 页面创建一个 JSP 页面实现类, JSP 页面实现类的名字是与实现相关的,通常与使用的服务器相关。

JSP 页面实现对象属于一个与实现相关命名的包,使用的包在一个 JSP 页面和另一个 JSP 页面之间可能发生改变,所以无名包中的类没有显式的引入某个类就不应该使用。

某个 JSP 页面的实现类的建立可能仅由 JSP Container 来做,或者它可能包含一个超类,这个超类由 JSP 页面作者使用 JSP 指令的 `extends` 属性提供。对一个有经验的用户来说,这种扩展机制是有用的,但是仍然应当格外小心地使用,因为它严格限定一个 JSP Container 能采用的行为,例如提高性能。

JSP 页面实现类将实现 Servlet,并且 Servlet 协议被用来提交请求给这个类。JSP 页

面实现类可能依赖于某些支持类,这个 JSP 页面实现类应当打包为一个 WAR,支持类也将包含在 WAR 包中,此时,这个 JSP 页面实现类可以方便的在所有 Container 中使用。

JSP 页面作者编写 JSP 页面时,希望客户与服务仅使用同一种协议就能够通信。JSP Container 必须保证来自哪个页面和响应哪个页面都用相同的协议。大多数 JSP 页面使用 HTTP,它们的实现类必须实现 `HttpJspPage` 接口。

## 8.6.4 JSP 容器的行为

上面说了一大堆理论上的东西,那么 JSP 容器到底是怎样实现 JSP 页面的,JSP 容器到底做了些什么,页面指令在 JSP 容器中怎样反映出来,表达式、声明都转换成什么?

### 1. 容器对 JSP 页面的处理

表 8-1 表示了容器处理 JSP 页面的过程。

表 8-1 JSP 容器怎样处理 JSP 页面

JSP 容器调用的方法	注 释
<code>void jspInit()</code>	这个方法可能在 JSP 页面中被定义,该方法在 JSP 页面初始化时调用。调用该方法后,Servlet 中的所有方法,包括 <code>getServletConfig()</code> 都是可用的。
<code>void jspDestroy()</code>	这个方法可能在页面中被定义,调用该方法以销毁页面。
<code>void _jspService(&lt; ServletRequestSubtype &gt;, &lt; ServletResponseSubtype &gt;)</code> <code>throws IOException, ServletException</code>	这个方法不能在 JSP 页面中被定义,JSP 容器根据 JSP 页面的内容自动的产生该方法。每次客户端请求都要调用这个方法。

这个表的意思是说每个 JSP 页面实现类必然有 `_jspService()` 方法,而 `jspInit()` 和 `jspDestroy()` 是可能实现的方法。事实上真是这样的吗?让我们来看看 JSP 页面翻译生成的文件。

下面你可能需要了解你使用的服务器的工作目录。本书 Tomcat 的工作目录是 `Work`,它位于 Tomcat 的根目录下。因为本书的例子都在 `examples` 下面,因此这些 JSP 页面生成的类及 `.java` 文件都放在 `localhost_8080%2Fexamples` 目录下,并且其文件名包含了原文件名。例如例 8-1 中,文件名为 `sample8_1_1.jsp`,那么生成的类及 `.java` 文件的名为 `_0002fbook_0002fchapter_00030_00038_0002fsample_00038_0005f_00031_0005f_00031_00031_0002ejspsample8_0005f1_0005f1_jsp_0.java (class)`,其包含了“sample8”这个字符串,因此比较容易找到。

在 8.1 节例子中输入图 8-1 所示数据,实际上是 JSP 语句即判断时间,打印“早上好!”或“下午好!”点击提交,效果图 8-3 所示。

查看页面的源代码,然后找到被翻译成的 `.java` 文件,看看它到底被翻译成什么?翻译成的 `.java` 文件如下:



图 8-3 JSP 文档生成器产生的效果

```
package book.chapter_00030_00038;
```

```
import javax.servlet.*;
```

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

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

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

```
import java.io.PrintWriter;
```

```
import java.io.IOException;
```

```
import java.io.FileInputStream;
```

```
import java.io.ObjectInputStream;
```

```
import java.util.Vector;
```

```
import org.apache.jasper.runtime.*;
```

```
import java.beans.*;
```

```
import org.apache.jasper.JasperException;
```

```
import java.util.*;
```

```
public
```

```
class
```

```
_0002fbook _0002fchapter_00030_00038_0002fsample_00038_0005f_00031_0005f_00031_00031_0002ejspsample8_0005f1_0005f11_jsp_1 extends HttpJspBase {
```

```

// begin [file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(5,0);to=(5,29)]
    public String getServletInfo() {
        return "Version1.0";
    }
// end
// begin [file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(6,3);to=(6,22)]
    String gm="早上好!";
// end

static {
}

public
_0002fbook _ 0002fchapter _ 00030 _ 00038 _ 0002fsample _ 00038 _ 0005f _ 00031 _
0005f _ 00031 _ 00031 _ 0002ejspsample8 _ 0005f1 _ 0005f11 _ jsp _ 1( ) {
}

private static boolean _jspx _inited = false;

public final void _jspx _init() throws JasperException {
}

public void _jspService(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException {

    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    String _value = null;
    try {
        if (_jspx _inited == false) {
            _jspx _init();
            _jspx _inited = true;
        }
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html; charset = gb2312");
    }

```

```

        pageContext = _jspxFactory.getPageContext(this, request, response,
        "", true, 8192, true);

        application = pageContext.getServletContext();
        config = pageContext.getServletConfig();
        session = pageContext.getSession();
        out = pageContext.getOut();

        //          HTML          //          begin
[file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(0,39);to=(1,0)]
        out.write("\r\n");
        // end
        //          HTML          //          begin
[file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(1,32);to=(2,0)]
        out.write("\r\n");
        // end
        //          HTML          //          begin
\ file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(2,50);to=(3,0)]
        out.write("\r\n");
        // end
        //          HTML          //          begin
[file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(3,46);to=(4,0)]
        out.write("\r\n");
        // end
        //          HTML          //          begin
[file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(4,31);to=(5,0)]
        out.write("\r\n");
        // end
        //          HTML          //          begin
[file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(5,29);to=(6,0)]
        out.write("\r\n");
        // end
        //          HTML          //          begin
[file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(6,24);to=(13,0)]
        out.write("\r\n\r\n<HTML>\r\n<HEAD>\r\n<TITLE></TITLE>\r\n</HEAD>\r\n<body>\r\n");
        // end
        // begin
[file="C: \\ book \\ chapter08 \\ sample8 _ 1 _ 11.jsp";from=(13,2);to=(14,2)]

```

```

        if (Calendar.getInstance().get(Calendar.AM _ PM) == Calendar.AM)
        {
            // end
            //
            HTML
            //
            begin
[ file="C: \ \ book \ \ chapter08 \ \ sample8 _ 1 _ 11.jsp";from=(14,4);to=(16,0)]
            out.write("\r\n<font size=4 color=blue> \r\n");
            // end
            // begin
[ file="C: \ \ book \ \ chapter08 \ \ sample8 _ 1 _ 11.jsp";from=(16,2);to=(16,20)]
            out.println(gm);
            // end
            //
            HTML
            //
            begin
[ file="C: \ \ book \ \ chapter08 \ \ sample8 _ 1 _ 11.jsp";from=(16,22);to=(18,0)]
            out.write("\r\n</font> \r\n");
            // end
            // begin
[ file="C: \ \ book \ \ chapter08 \ \ sample8 _ 1 _ 11.jsp";from=(18,2);to=(18,12)]
            } else {
            // end
            //
            HTML
            //
            begin
[ file="C: \ \ book \ \ chapter08 \ \ sample8 _ 1 _ 11.jsp";from=(18,14);to=(22,0)]
            out.write("\r\n<font size=4 color=red> \r\n 下午好! \r\n
n</font> \r\n");
            // end
            // begin
[ file="C: \ \ book \ \ chapter08 \ \ sample8 _ 1 _ 11.jsp";from=(22,2);to=(22,5)]
            }
            // end
            //
            HTML
            //
            begin
[ file="C: \ \ book \ \ chapter08 \ \ sample8 _ 1 _ 11.jsp";from=(22,7);to=(25,0)]
            out.write("\r\n</body> \r\n</HTML> \r\n");
            // end

    } catch (Exception ex) {
        if (out.getBufferSize() != 0)
            out.clearBuffer();
        pageContext.handlePageException(ex);
    } finally {

```

```

        out.flush();
        _jspxFactory.releasePageContext(pageContext);
    }
}

```

从上面的文件中可以看到生成的 .java 文件与我们编写的 Servlet 格式上没有两样。首先打包,然后导入类库。接着就是类体,类体里有静态初始化, `_jspX_init()`、`_jspService()`。最重要的就是 `_jspService()` 了,它包含了原 JSP 文件的大部分内容,是实现其功能的主要场所。

## 2. JSP 实现类的结构

事实上,这样一个完整的类包括表 8-2 所示内容:

表 8-2 JSP 实现类的结构

import name1	可选 import 语句,通过 jsp 指令指明
class _jspXXX extends SuperClass	JSP 容器或 JSP 页面作者通过 JSP 指令选择的超类。实现相关的类(_jspXXX)的名字
{	JSP 页面实现类体的开始
// declarations ...	声明区
public void _jspService( < ServletRequestSubtype > request, < ServletResponseSubtype > response) throws ServletException, IOException {	_jspService 的开始
//code that defines and initializes request, response, page, pageContext etc.	内部对象区
// code that defines request/response mapping	主体区
}	关闭 _jspService 方法
	关闭 _jspXXX 类实现

### 1) 声明区间

声明区间对应声明元素,这种区间的内容是由所有出现在页面中的声明按顺序连接而成。

### 2) 初始化区间

它定义和初始化对 JSP 页面有用的内部对象。

### 3) 主区间(主体区)

主区间提供 request 和 response 对象之间的主要映射。

代码段的内容由脚本片断、表达式和 JSP 页面的文本决定。这些元素相继被处理,对每个元素的转换由以下决定:

(1)模板数据(Template data)被转换成代码,代码将代替 Template data 写入当前内

部变量 out 流中。所有空白被保留。

忽略引用问题和实现问题,模板数据的转换相应形式为:

```
out.print(Template data);
```

(2)脚本片断转换成 Java 语句片断。

(3)表达式被转成 Java 语句并将表达式的值转成 `Java.lang.string`,插入当前内部变量 out 流中。没有新行或空白产生。

忽略引用问题和实现问题,表达式的转换相应形式如下:

```
out.print(expression)
```

(4)定义一个或多个对象的行为被转换成对这些对象的一个或多个变量声明,与行为一起的代码用于初始化变量。这些变量的可见性被其它结构影响,如脚本片断。行为(类型)的语义决定通常由 id 属性决定变量的名称和类型。在 JSP 已知规范中,唯一的定义对象的标准行为是:`<jsp:useBean />`行为。引进的变量名是 useBean 的 id 属性,变量类型是 Bean 的 class 属性。



注意:scope 属性的值在其产生的程序中并不影响其可见性,它仅仅影响那些通过变量引用这个对象的场合。

### 3.JSP 容器行为比较

#### 1) isThreadSafe=false 与 isThreadSafe=true 的比较

清楚了 JSP 容器的作用,也就很容易知道实现线程安全(isThreadSafe=false)与没有实现线程安全有什么不同了。利用第一节提供的工具,改变 isThreadSafe=false,比较产生的代码。

isThreadSafe=true 生成代码如下:

```
public class _0002fbook_0002fchapter_00030_00038_0002fsample_00038_0005f_00031_0005f_00031_00031_0002ejspsample8_0005f1_0005f11_jsp_1 extends HttpJspBase {
    .....
}
```

isThreadSafe=false 生成代码如下:

```
public class _0002fbook_0002fchapter_00030_00038_0002fsample_00038_0005f_00031_0005f_00031_00031_0002ejspsample8_0005f1_0005f11_jsp_2 extends HttpJspBase
implements SingleThreadModel
{
    .....
}
```

可以看到,后者实现了 SingleThreadModel 界面。

#### 2) include 指令与 include 行为的比较

再来看看前面反复比较的 include 指令与 include 行为有什么不同。在 JSP 文档生成器的过程中两次分别输入`<%@ include file="sample8_1_11.txt"%>`和`<jsp: include page="sample8_1_11.txt" flush="true"/>`。这次差异相当大,比较如下:



<%@ include file="sample8 \_ 1 \_ 11.txt"%>生成代码整理如下:

```
// HTML // begin [file="C:\\book\\chapter08\\sample8 _ 1 _ 11.txt";from=(0,0);to=(13,0)]
out.write("<%@ page buffer = \"8\" autoFlush = \"true\" %><br> \r\n
<%@ page contentType = \"text/html; charset = gb2312\" %><br> \r\n
<%@ page session = \"true\" isThreadSafe = \"true\" %><br> \r\n
<%@ page isErrorPage = \"false\" %><br> \r\n\r\n
<HTML><br> \r\n
<HEAD><br> \r\n
<TITLE><br>
</TITLE><br> \r\n
</HEAD><br> \r\n
<body><br> \r\n
<%@ include file = \"sample8 _ 1 _ 11.txt\" %><br> \r\n
</body><br> \r\n
</HTML><br> \r\n");
// end
```

<jsp: include page="sample8 \_ 1 \_ 11.txt" flush="true"/>生成代码如下:

```
pageContext.include("sample8 _ 1 _ 11.txt" + _jspx _ qStr);
```

可以很容易的得出第 5 章讲述的结论,即 include 指令在翻译时包含进调用文件,与包含文件一起被翻译。而 include 行为直到请求时才被包含。

### 3) buffer 属性不同取值的比较

最后来比较一下有 buffer 和没有 buffer 的情况。

JSP Container 使用 buffer 缓存数据。从服务端发送数据到客户端时,直到第一个 flush()方法被调用,否则 Header(头部)不会被发送到客户端。因此,操作不能依赖于头部。例如,SetContentType,Redirect 或 error 方法是有效的,一旦 flush 方法被执行,头部被发出。

当缓存打开时,也可以通过调用 Response.redirect(someURL)方法在 JSP 脚本片断中使用 redirect 方法。

```
buffer = 8k
```

```
pageContext = _jspxFactory.getPageContext(this, request, response, "", true, 8192,
true);
```

```
buffer = 0k
```

```
pageContext = _jspxFactory.getPageContext(this, request, response, "", true, 0, true);
```

可以看到仅仅 getPageContext()方法使用的参数不一样。

有了这个工具,可以很轻松的比较各种页面指令不同取值之间的差异。在此,不作过多的比较。

## 8.7 留言板

本章开始给出的 JSP 文档生成器不仅仅用来方便地比较 page 指令不同属性之间的差异,而且只要稍微改造一下,它还可以是一个不使用数据库的留言板。这个留言板可以屏蔽用您想屏蔽的任意字符串,还能使最新的留言总置于最前面。

### 8.7.1 留言板的说明

这个留言板没有使用数据库技术,而直接采用读写文件的形式。首先它从表单中获得数据,然后进行代码转换,获得双字节编码,再经过一个字符串替换函数的处理,屏蔽掉不安全的,不需要出现的字符串,最后写入一个文件。而这个文件正是查看留言时使用的 JSP 文件。

### 8.7.2 留言的处理

#### 1. 页面效果及其源代码

首先来看看页面效果,见图 8-4。

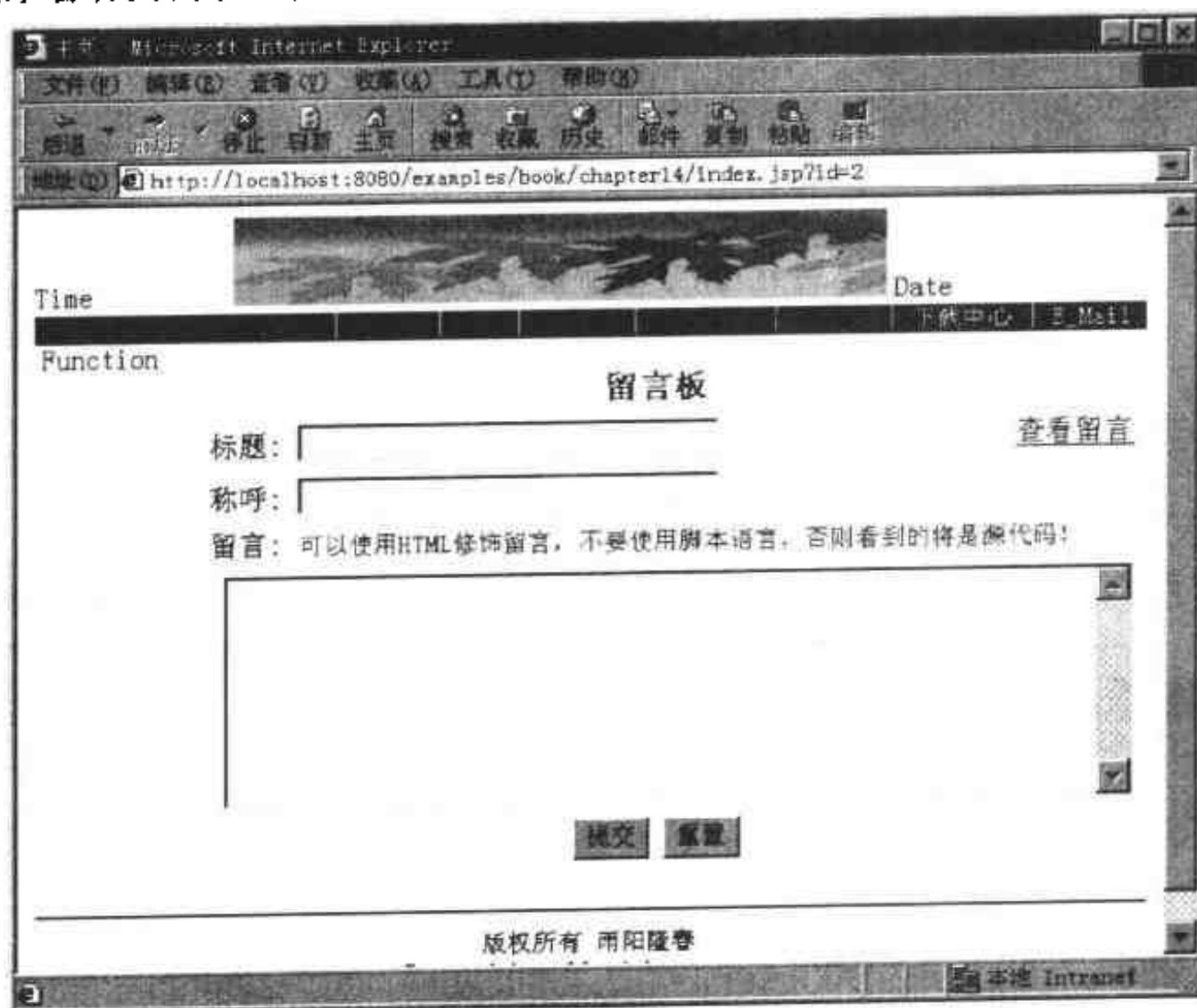


图 8-4 留言板页面效果图

< %@ page contentType="text/html; charset = gb2312" % >

	<table width="98%" border="0" cellspacing="8" cellpadding="0">
--	--

|
 <td height="40" colspan="2"><font color="#990099" size="4"> |**留言板**| <tr> |
[illegible]  |

<div align="right"><a href="/examples/book/chapter14/index.jsp? id=21">

[查看留言](#)

| <tr> |
 称呼: | |

```
<input type="text" name="txtName" size="30" maxlength="50">
```

| <tr> |
[illegible]| <tr> |
  | |

&lt;textarea name="txtArticle" cols="65" rows="8"&gt;&lt;/textarea&gt;

 $\langle \tau_I \rangle$ `<td colspan="2">`

<div align="center">

```

        </td>
    </tr>
</table>
</form>

```

## 2. 后台处理代码

这个页面的后台处理代码如下：

```

<%@ page import = "java.io. *" %>
<%@ page contentType = "text/html; charset = gb2312" %>
//这个类作用是屏蔽掉任意需要屏蔽的字符串，
//比 JSP 文档生成器的屏蔽功能强大得多。
<%! public class Strexpand{
    private String strexp;
    public Strexpand(String s){
        strexp = new String(s);
    }

    public String placeStr(String oldstr, String newstr){
        int lengthofold = oldstr.length();
        int lengthofstrexp = strexp.length();
        int p = strexp.indexOf(oldstr);
        String placedstrexp = strexp.substring(0, p);
        placedstrexp = placedstrexp + newstr;
        placedstrexp = placedstrexp + strexp.substring(p + lengthofold, lengthofstrexp);
        return placedstrexp;
    }
}

%>
<%
try{
    //编码转换
    String stitle = new String(request.getParameter("txtTitle").getBytes("ISO-8859-1"));
    String sname = new String(request.getParameter("txtName").getBytes("ISO-8859-1"));
    String sarticle = new String(request.getParameter("txtArticle").getBytes("ISO-8859-1"));
    if(stitle.equals("") || sname.equals("") || sarticle.equals("")){
        //如果任一输入未填写，转发到错误处理页面
    }
}
%>
<jsp:include page = "error.jsp" flush = "true" />
<%

```

```

    }else{
        //写入文件的代码
        String tb="<table width='100%' cellpadding='3'>";
        String t1="<tr align=center bgcolor=#EEFFCC><td><font size=2 color=red>" +
            stitle + "</font>";
        String t2="</td></tr><tr align=right><td>" + sname + "</td></tr><tr><td>" +
            sarticle + "</td></tr></table>";
        String sall = tb + t1 + t2;
        //屏蔽"< %"
        while(sall.indexOf("< %")! = -1){
            Strexpand sexp = new Strexpand(sall);
            sall = sexp.placeStr("< %", "&lt; %");
        }
        //屏蔽"% >"
        while(sall.indexOf("% \\ >")! = -1){
            Strexpand sexp = new Strexpand(sall);
            sall = sexp.placeStr("% \\ >", "% &gt;");
        }
        //屏蔽"< SCRIPT"
        while(sall.indexOf("< SCRIPT")! = -1){
            Strexpand sexp = new Strexpand(sall);
            sall = sexp.placeStr("< SCRIPT", "&lt; SCRIPT");
        }
        //读出以前的留言
        File gbookfile = new File("../webapps/examples/book/chapter14/guestbook/seegb.jsp");
        RandomAccessFile raf = new RandomAccessFile(gbookfile, "r");
        int allsize = (int)raf.length();
        byte bc[] = new byte[allsize];
        raf.readFully(bc);
        raf.close();
        String before = new String(bc);
        //屏蔽"< %@page contentType='text/html; charset = gb2312' % >"
        String ct = "< %@ page contentType = \"text/html; charset = gb2312\" % \\ >";
        while(before.indexOf(ct)! = -1){
            Strexpand sexp = new Strexpand(before);
            before = sexp.placeStr(ct, "\\ r \\ n");
        }
        sall = sall + "\\ r \\ n" + before;
        FileWriter fw = new FileWriter(gbookfile);
    }
}

```

```
PrintWriter pw=new PrintWriter(fw,true);
//将处理后的字符串写入文件
pw.println(ct);
pw.println(sall);
pw.close();
fw.close();
//转发到查看留言页面
%>
<jsp: include page="seegb.jsp" flush="true" />
<%
}
}
catch(Exception e){
    System.out.println(e.getMessage());
}
%>
```

### 3. 说明与改进建议

首先实现了一个 `Strexpend` 类,可以替换任意的字符串,然后将表单提交的数据经编码转换后使用 `Strexpend` 类的 `placeStr()` 方法替换为需要屏蔽的字符串,例如脚本语言分隔符。为了实现最新的留言总在最前面,需要将文件中的内容读出,再将当前字符串与之合并,最后将合并后的字符串写入文件,生成一个 JSP 文件,该文件就作为查看留言板的处理程序。

改进建议,替换任意字符串的算法是比较低效的,因为它总是从头到尾去匹配屏蔽字符串,也就是说曾经替换过的字符串还会多次遍历,可以考虑将已经处理过的字符串用另外的变量存储起来。另外为了保证最新的留言总在最前面,需要从文件中读出以前的内容,在文件比较大的时候,这种算法肯定是行不通的。因此要么放弃这种最新留言总在是前这种做法,要么采用数据库实现。

### 8.7.3 查看留言

查看留言使用的 JSP 文件就是上一步生成的,因此不再列出源代码,请读者参见 `chapter14/guestbook` 目录下的 `seegb.jsp`。这里仅仅给出页面效果图,如图 8-5 所示:

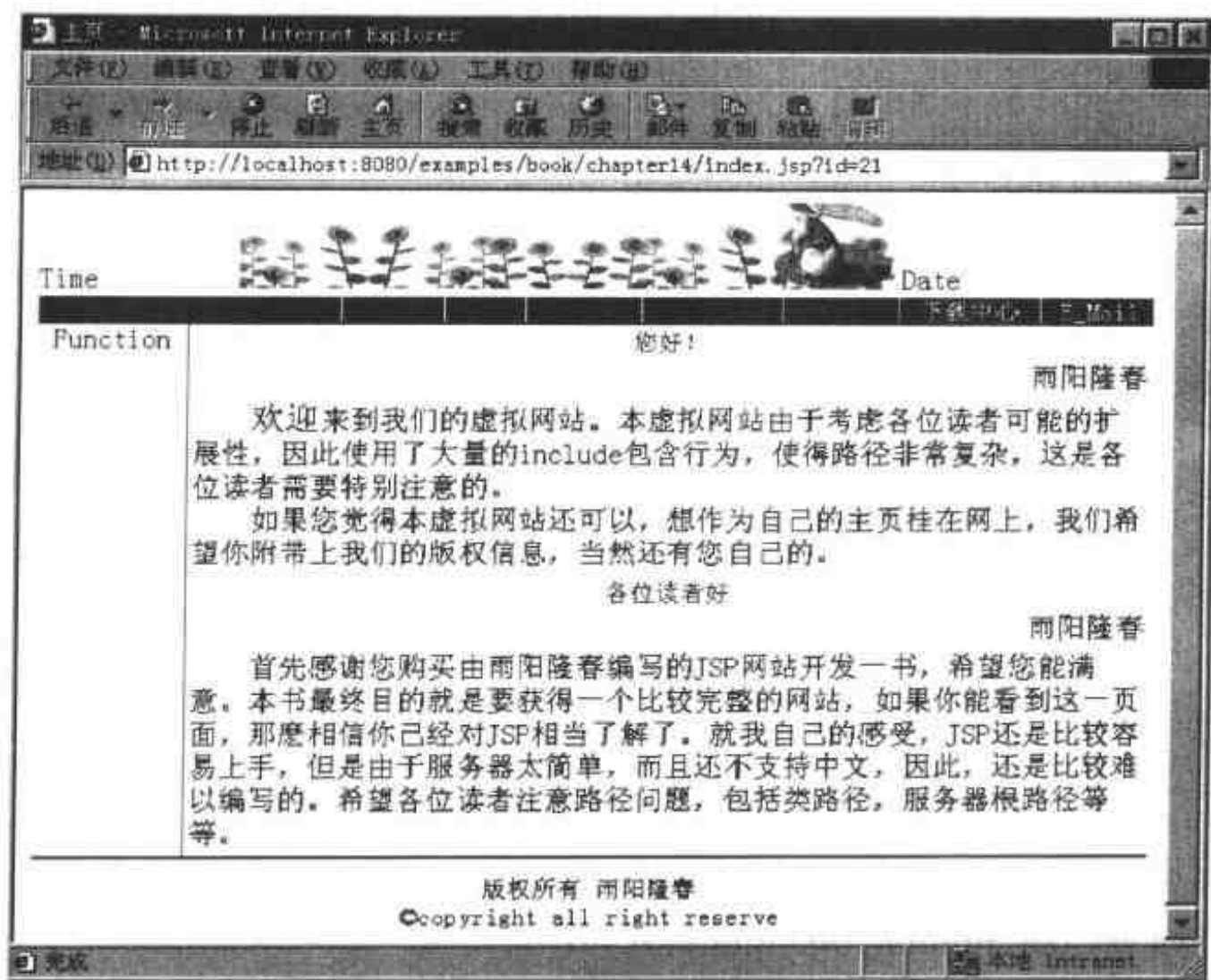


图 8-5 查看留言板效果图

## 8.7.4 错误处理

### 1. 页面效果及其源代码

页面效果如图 8-6 所示。

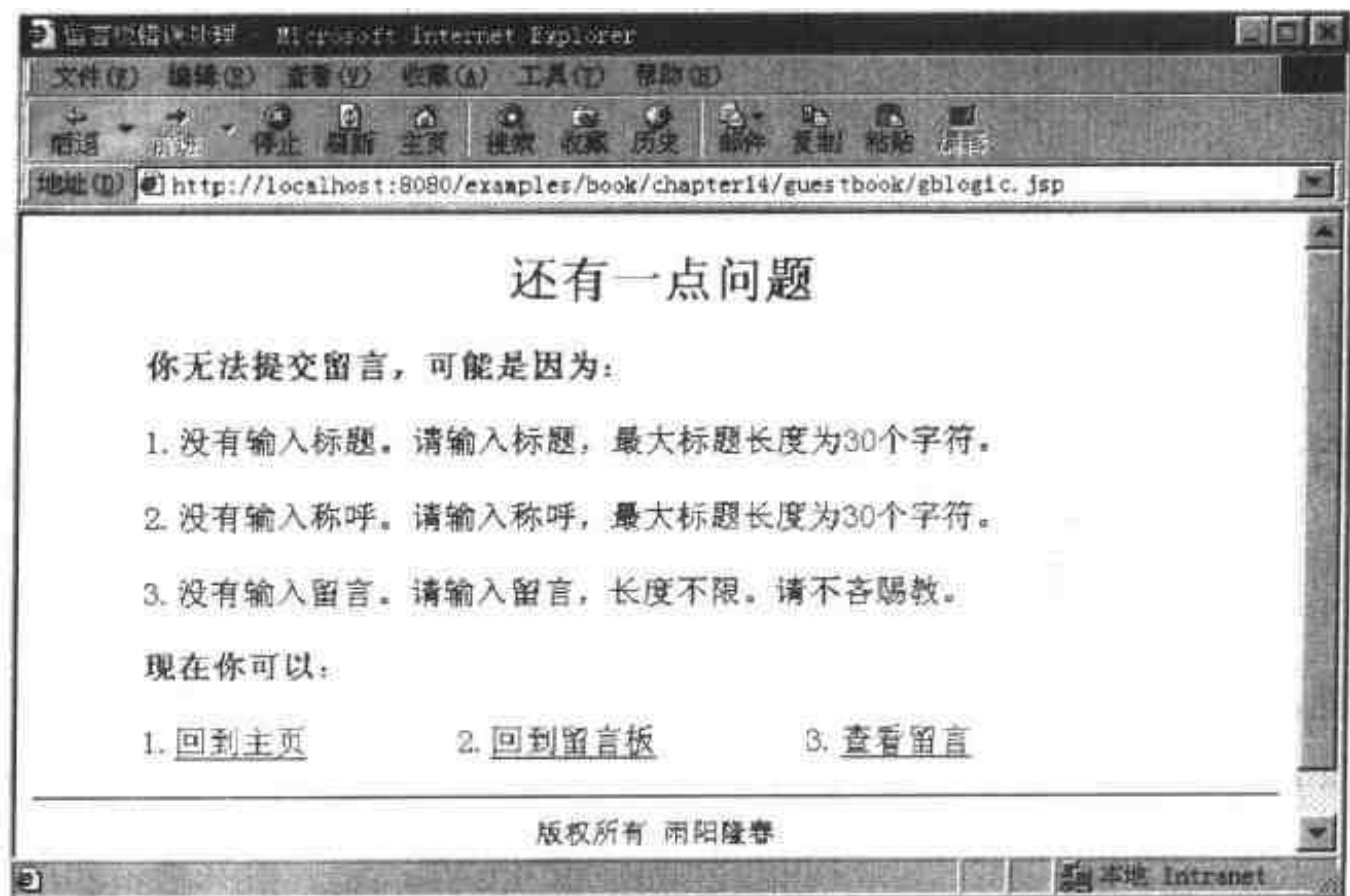


图 8-6 错误处理页面

源代码如下：

```
<%@ page contentType="text/html; charset = gb2312" %>
<html>
<head>
<title>留言板错误处理</title>
</head>
<body bgcolor="# FFFFCC">
<table cellspacing="10" cellpadding="0" border="0" align="center">
  <tr>
    <td width="620" height="320" valign="top">
      <p align="center"><font color="# 990099" size="5">
        <b>还有一点问题</b> </font></p>
      <p align="left"><font color="# 006633">
        <b>你无法提交留言,可能是因为:</b></font></p>
      <p align="left">1. 没有输入标题。请输入标题,最大标题长度为 30
        个字符。</p>
      <p align="left">2. 没有输入称呼。请输入称呼,最大标题长度为 30 个字
        符。</p>
      <p align="left">3. 没有输入留言。请输入留言,长度不限。请不吝赐教。</p>
      <p align="left"><b><font color="#006633">现在你可以:</font></b></p>
      <p align="left">
        1.<a href=" ../index.jsp? id=1">回到主页</a>
        2.<a href=" ../index.jsp? id=2">回到留言板</a>
        3.<a href=" ../index.jsp? id=21">查看留言</a>
      </p>
    </td>
  </tr>
</table>
<jsp:include page=" ../template/footer.jsp" flush="true" />
</body>
</html>
```

## 2. 说明

如果采用这种方式实现留言板,这里唯一需要注意的是采用了 URL 重写技术,如  
 <a href=" ../index.jsp? id=1">。



## 8.8 本章小结

本章讲述了 JSP 概念的第三个核心,从运行角度把握了 JSP 是一个 Container,它为 JSP 页面提供生命周期管理和运行时支持。并比较了 page 指令属性值不同时生成的 Servlet 类的不同之处。最后我们实现了一个使用文件保存留言信息的留言板,又为创建我们的虚拟网站添加了一笔。

# 第9章 JSP 核心 API

## 9.1 内部对象

本章讲述 javax.servlet.jsp 包。包中包含了大量的类和接口。这些类和接口描述和定义了 JSP 页面实现类和 JSP 容器为这些类提供的运行环境之间的关系。默认情况下, PageContext 对象和 JspWriter 是可用的内部对象。

### 9.1.1 PageContext

#### 1. 语法

```
public abstract class PageContext
```

#### 2. 描述

PageContext 实例提供了访问所有与 JSP 页面相关的名域的方法,还提供访问几个页面属性的方法。PageContext 类是一个抽象类,兼容 JSP 引擎运行环境使用它实现扩展。JSP 实现类调用方法 JspFactory.getPageContext() 获得一个 PageContext 实例,调用方法 JspFactory.releasePageContext() 释放一个 PageContext 实例。

PageContext 提供了大量的便捷方法给页面/组件作者和页面实现者,包括:一、管理不同(作用)名域的单一 API;二、大量的、方便的、访问各种公共对象的 API;三、为输出获得 JspWriter 的机制;四、管理页面使用 session 的机制;五、传递页面指令属性给脚本环境的机制;六、应用程序中,转发(重导)或包含当前 Request 给其它活动组件的机制;七、处理异常的 errorPage 机制。

##### 1) 用于容器产生代码的方法

(1) 有些方法有意只提供给容器产生代码,不能被 JSP 页面作者或标记库开发者使用。

(2) 支持生命周期的方法: initialize() 和 release()。

下述方法支持嵌套 JspWriter 流的管理以实现标记库的扩展: pushBody() 和 popBody()。

##### 2) 给页面作者使用的方法

(1) 提供统一的访问不同作用域对象的方法,实现必须使用相应于哪个作用域的底层 Servlet 机制。这样信息就可在 Servlet 和 JSP 页面之间往返传递,其方法有: setAttribute(), getAttribute(), findAttribute(), removeAttribute(), getAttributesScope() 和

getAttributeNamesInScope()。

(2) 方便地访问内部对象的方法: getOut(), get-Exception(), getPage() getRequest(), getResponse(), getSession(), getServletConfig() 和 getServletContext()。

支持转发、包含和错误处理的方法有: forward(), include(), and handlePageException()。

### 3. 域

(1) public static final java.lang.String APPLICATION

PageContext 名字表中用来存储 ServletContext 的名字。

(2) public static final int APPLICATION \_ SCOPE

Application 的作用域: ServletContext 中名字引用保持有效, 直至引用被要求归还。

(3) public static final java.lang.String CONFIG

PageContext 名字表中用来存储 ServletConfig 的名字。

(4) public static final java.lang.String EXCEPTION

ServletRequest 属性列表和 PageContext 名字表中用来存储不可捕获异常的名字。

(5) public static final java.lang.String OUT

PageContext 名字表中用来存储 JspWriter 的名字。

(6) public static final java.lang.String PAGE

当前 PageContext 的名字表中用来存储 Servlet 的名字。

(7) public static final int PAGE \_ SCOPE

页面作用范围: (这是默认的) 当前 PageContext 中名字引用保持有效, 直到从当前 Servlet.service() 方法调用中返回。

(8) public static final java.lang.String PAGECONTEXT

在自己的名字表中, 存储当前 PageContext 的名字。

(9) public static final java.lang.String REQUEST

PageContext 名字表中用来存储 ServletRequest 的名字。

(10) public static final int REQUEST \_ SCOPE

请求作用域: 从 ServletRequest 与 Servlet 发生联系到这个 request 被完成, 名字引用保持有效。

(11) public static final java.lang.String RESPONSE

PageContext 名字表中用来存储 ServletResponse 的名字。

(12) public static final java.lang.String SESSION

PageContext 名字表中用来存储 HttpSession 的名字。

(13) public static final int SESSION \_ SCOPE

Session 的作用域(当前页面使用了 session 才有效): 从 HttpSession 与 Servlet 相联系到 HttpSession 失效, 名字引用保持有效。

### 4. 构造器

public PageContext()

## 5. 方法

1) `public abstract java.lang.Object findAttribute(java.lang.String name)`

依次在 `page`, `request`, `session` (如果有效) 和 `application` 作用域中搜索给定名属性并返回相联系的值或 `null`。

返回:

相关值或 `null`。

2) `public abstract void forward(java.lang.String relativeUrlPath)`

方法用于重定向或转发当前 `ServletRequest` 和 `ServletResponse` 到应用程序中另外的活动组件。

从 JSP 页面 `_jspService()` 方法中的一个运行线程调用这个方法是唯一有效途径。

一旦这个方法调用成功,调用线程试图改变 `ServletResponse` 对象是违法的,任何这样的意图都将导致未定义的行为。调用这个方法之后,调用者立即从 `_jspService()` 方法中返回。

参数:

`relativeUrlPath`: 指定到目标资源的相对路径。

抛出:

`ServletException`, `IOException`

`IllegalArgumentException`: 如果目标资源 URL 不可解析。

`IllegalStateException`: 如果 `ServletResponse` 不在 `forward` 能处理的状态。例如, `ServletResponse` 已发出。

`SecurityException`: 如果目标资源不能被调用者访问。

3) `public abstract java.lang.Object getAttribute(java.lang.String name)`

返回 `page` 作用域中与名字相联系的对象,或未找到,返回空。

参数:

`name`: 欲获得的属性名。

抛出:

`NullPointerException`: 如果名字为空。

`IllegalArgumentException`: 如果作用域无效。

4) `public abstract java.lang.Object getAttribute(java.lang.String name, int scope)`

返回指定作用域中与名字相联系的对象,或未找到,返回空。

参数:

`name`: 欲设置的属性名。

`scope`: 与名字或对象相联系的作用域。

抛出:

`NullPointerException`: 如果名字为空。

`IllegalArgumentException`: 如果作用域无效。

5) `public abstract java.util.Enumeration getAttributeNamesInScope(int scope)`

枚举给定作用域中的所有属性。

返回：

指定作用域中所有属性名组成的一个枚举变量。

6) public abstract int getAttributesScope(java.lang.String name)

获得给定属性所定义的作用域。

返回：

与指定名字相联系的对象的作用域或 0。

7) public abstract java.lang.Exception getException()

异常对象的当前值。

返回：

一个错误页面传递给它的任何异常。

8) public abstract JspWriter getOut()

out 对象(JspWriter)的当前值。

返回：

用作客户端响应的当前 JspWriter 流。

9) public abstract java.lang.Object getPage()

页面对象的当前值(一个 Servlet)。

返回：

与这个 PageContext 相联系的页面实现类的实例。

10) public abstract javax.servlet.ServletRequest getRequest()

请求对象的当前值(一个 ServletRequest)。

返回：

对这个 PageContext 的 ServletRequest。

11) public abstract javax.servlet.ServletResponse getResponse()

响应对象的当前值(一个 ServletResponse)。

返回：

这个 PageContext 的 ServletResponse。

12) public abstract javax.servlet.ServletConfig getServletConfig()

ServletConfig 实例。

返回：

这个 PageContext 的 ServletConfig。

13) public abstract javax.servlet.ServletContext getServletContext()

ServletContext 实例。

返回：

这个 PageContext 的 ServletContext。

14) public abstract javax.servlet.http.HttpSession getSession()

session(会话)对象的当前值(一个 HttpSession)。

返回：

这个 PageContext 的 HttpSession 或 null。

15) `public abstract void handlePageException(java.lang.Exception e)`

这个方法试图通过这样的行为——重定向未经处理过的“页面”级异常到指定的错误处理页面或未指定错误处理页面而执行一些与实现相关的动作——来处理异常。

一个 JSP 实现类在调用这个方法之前清除任何本地状态,调用之后立即返回。调用这个方法后,产生任何输出到客户端或修改任何 `ServletResponse` 状态都是违法的。

因为需要向后兼容,所以这个方法 `PageContext.handlePageException( Throwable)` 被保留。最新的 JDK 版本,应当使用方法 `PageContext.handlePageException( Throwable)`。

参数:

`e`:待处理的异常。

抛出:

`ServletException`, `IOException`

`NullPointerException`:如果异常为 `null`。

`SecurityException`:如果目标资源不能被调用者访问。

参照:

`public abstract void handlePageException(java.lang.Throwable t)`

16) `public abstract void handlePageException(java.lang.Throwable t)`

这个方法与 `handlePageException(Exception)` 是等价的,除了接受参数是一个 `Throwable` 对象。这是首选的使用方法,因为它是 `errorpage` 语义的正确实现。

这个方法试图通过这样的行为——重定向未经处理过的“页面”级异常到指定的错误处理页面或未指定错误处理页面而执行一些与实现相关的动作——来处理异常。

一个 JSP 实现类在调用这个方法之前清除任何本地状态,调用之后立即返回。调用这个方法后,产生任何输出到客户端或修改任何 `ServletResponse` 状态都是违法的。

参数:

`t`:待处理的抛出类(`throwable`)对象。

抛出:

`ServletException`, `IOException`

`NullPointerException`:如果异常为 `null`。

`SecurityException`:如果目标资源不能被调用者访问。

参照:

`public abstract void handlePageException(java.lang.Exception e)`

17) `public abstract void include(java.lang.String relativeUrlPatb)`

使指定资源作为调用线程的当前 `ServletRequest` 和 `ServletResponse` 的一部分被处理。目标资源的处理输出被直接写到 `ServletResponse` 输出流。这个 JSP 页面的当前 `JspWriter` “out”作为这个调用的副作用先于 `include` 处理被刷新。

如果相对 `Url` 路径以 “/” 开头,那么这个指定的 `URL` 将相对于当前 JSP 的 `Servlet-Context` (当前 `Servlet` 上下文)的 `DOCROOT` (文档根目录)计算。如果路径不是以 “/” 开头,那么指定的 `URL` 将相对于调用者发出的请求映像的路径来计算。

这个方法只有 JSP 的 `_jsp-Service(...)` 方法内运行的某个线程调用才是有效的。

参数:

relativeUrlPath:指定(配置)被包含目标资源的相对 URL 路径。

抛出:

ServletException, IOException

IllegalArgumentException:如果目标资源 URL 不可解析。

SecurityException:如果目标资源不能被调用者访问。

18) public abstract void initialize(javax.servlet.Servlet servlet, javax.servlet.ServletRequest request, javax.servlet.ServletResponse response, java.lang.String errorPageURL, boolean needsSession, int bufferSize, boolean autoFlush)

初始化方法被调用以初始化一个还未初始化的 PageContext,所以,它可被一个 JSP 实现类用在 \_jspService()方法为即将到来的 Request 和 Response 提供服务。

这个方法的典型调用来自 JspFactory.getPageContext(),目的是初始化状态。

这个方法需要建立一个初始的 JspWriter,并且在页面作用域内使这个新建立的对象与“out”相联系。

注意这个方法不能被页面或标记库作者使用。

参数:

servlet:与这个 PageContext 相联系的 Servlet。

request:对这个 Servlet 的未定当前请求。

response:对这个 Servlet 的未定当前响应。

errorPageURL:来自页面指令属性 errorpage 的值或为 null。

needsSession:来自页面指令属性 session 的值。

bufferSize:来自页面指令属性 buffer 的值。

autoFlush:来自页面指令属性 autoflush 的值。

抛出:

IOException:在 JspWriter 的建立期间。

IllegalStateException:如果 out 未被正确初始化。

IllegalArgumentException

19) public JspWriter popBody()

返回与其配对的 pushBody()先前存储的 JspWriter“out”对象,并更新 PageContext 中名域为“页面作用域属性”下“out”属性的值。

返回:

保存的 JspWriter。

20) public BodyContent pushBody()

返回一个新的 BodyContent 对象,存储在当前“out”JspWriter,并更新 PageContext 中名域为“页面作用域属性”下“out”属性的值。

返回:

新的 BodyContent。

21) public abstract void release()

这个方法将“重置”一个 PageContext 的内部状态,释放所有的内部引用,并为今后可能调用 initialize()方法做准备。

这个方法典型地被 `JspFactory.releasePageContext()` 方法调用。

其子类将封装这个方法。注意这个方法不能被页面和标记库作者使用。

22) `public abstract void removeAttribute(java.lang.String name)`

删除与给出名字相联系的对象引用,这个方法将以作用域顺序检查所有的作用域。

参数:

`name`: 欲删除对象的名字。

23) `public abstract void removeAttribute(java.lang.String name, int scope)`

删除给出作用域内与指定名字相联系的对象引用。

参数:

`name`: 欲删除对象的名字。

`scope`: 将检查的作用域。

24) `public abstract void setAttribute(java.lang.String name, java.lang.Object attribute)`

注册名字和页面作用域语义指定的对象。

参数:

`name`: 欲设置的属性的名字。

`attribute`: 与名字相联系的对象。

抛出:

`NullPointerException`: 如果名字或对象为 `null`。

25) `public abstract void setAttribute(java.lang.String name, java.lang.Object o, int scope)`

注册名字和适当的作用域语义指定的对象。

参数:

`name`: 欲设置的属性的名字。

`o`: 与名字相联系的对象。

`scope`: 与名字对象相联系的作用域。

抛出:

`NullPointerException`: 如果名字或对象为 `null`。

`IllegalArgumentException`: 如果作用域无效。

## 9.1.2 JspWriter

### 1. 语法

```
public abstract class JspWriter extends java.io.Writer
```

### 2. 已知直接子类

`BodyContent`

### 3. 描述

JSP 页面中的行为和 `template data` 使用 `JspWriter` 对象写。内部对象 `out` 引用 `Jsp-`



Writer, out 在 PageContext 对象中被自动初始化。

这个抽象类模仿了 java.io.BufferedWriter 和 java.io.PrintWriter 类的某些功能,然而,不同的是在 print 方法中它抛出 io.IOException 异常,而 PrintWriter 不会。

最初的 JspWriter 对象是否与 ServletResponse 的 PrintWriter 对象相联系取决于页面是否被缓存。如果没有被缓存,写向当前 JspWriter 对象的输出将直接写穿 PrintWriter,有必要的话,在 response 对象上调用 getWriter() 方法可建立一个 PrintWriter。如果页面被缓存了,直到缓冲区被刷新并且像 setContentType() 之类的操作合法,PrintWriter 对象才会被建立。这种弹性简化了程序设计,对 JSP 页面,buffering 是默认值。

当缓冲区溢出时,buffering 提高了问题的难度,可采用如下两种办法:

- (1) 当缓冲区溢出不是一个致命错误的时候,仅仅刷新输出。
- (2) 当缓冲区溢出是一个致命错误的时候,引发一个异常。

两种方法都是有效的,都是 JSP 技术支持的。页面的行为是由 autoFlush 属性控制,autoFlush 默认值是 true。通常需要将正确和完整的数据发送给客户端的 JSP 页面可能设置 autoFlush 为 false,一个典型的情况是客户端为应用程序自身。另一方面,JSP 页面数据有特殊意义,即使仅仅构造了一部分也希望发出,例如当需要立即把数据显示在浏览器上的时候,这时需要设置 autoFlush 为 true。总之,每个应用程序都需要考虑其特殊的要求。

一个可选择的考虑是设置极大的缓冲区。但是,这有个缺点是失败(失控)的计算将销毁大量的资源。

JSP 实现类的内部对象“out”就是这种类型。如果页面指令选择 autoFlush = “true”,并且当前操作导致缓冲区溢出,那么这个类的所有 I/O 操作将自动的刷新缓冲区的内容。如果 autoFlush = “false”,并且当前操作导致缓冲区溢出,那么这个类的所有 I/O 操作将抛出一个 IOException。

#### 4. 参照

java.io.Writer, java.io.BufferedWriter, java.io.PrintWriter

#### 5. 域

- (1) protected boolean autoFlush

protected autoFlush 域,参见第 5 章 page 指令的 autoFlush 属性。

- (2) protected int bufferSize

protected bufferSize 域,指定值或取值为下面三个常数的一个。

- (3) public static final int DEFAULT\_BUFFER

常数指明 Writer 将被缓存并且使用默认的缓冲区大小。

- (4) public static final int NO\_BUFFER

常数指明 Writer 没有缓冲输出。

- (5) public static final int UNBOUNDED\_BUFFER

常数指明 Writer 被缓存并且有一个极大的缓冲区。这在 Body-Content 中使用。

## 6. 构造器

(1) `protected JspWriter(int bufferSize, boolean autoFlush)`

`protected` 构造器。

## 7. 方法

1) `public abstract void clear()`

清除缓冲区的内容。如果缓冲区内容已经被刷新,那么清除操作将抛出一个 `IOException` 表示实际上有部分数据已经被错误的写到客户端响应流中。

抛出:

`IOException` - 如果一个 I/O 错误发生。

2) `public abstract void clearBuffer()`

清除当前缓冲区中的内容。与 `clear()` 不同,在缓冲区已经被刷新的情况下,这个方法不会抛出 `IOException`。它只是清除当前缓冲区的内容,然后返回。

抛出:

`IOException` - 如果 I/O 错误发生。

3) `public abstract void close()`

首先刷新流,然后关闭它。

这个方法不需要显式调用,因为 JSP 容器产生的初始化的 `JspWriter` 将自动的包含对 `close()` 的调用。与 `flush()` 不同,关闭一个先前关闭的流是没有影响的。

覆盖:

`java.io.Writer.close()` in class `java.io.Writer`

抛出:

`IOException` - 如果一个 I/O 错误发生。

4) `public abstract void flush()`

刷新流。如果流已经把来自各种 `write()` 方法的任何字符存储在一个缓冲区,立即把它们写到目的地。如果目的是其它的字符或字节流,刷新它。如此一次 `flush()` 调用将刷新 `Writers` 和 `OutputStreams` 链上的所有缓冲区。

如果溢出缓冲区大小,这个方法就不能被直接调用。

一旦一个流被关闭,`writer()` 或 `flush()` 调用将导致 `IOException` 抛出。

覆盖:

`java.io.Writer.flush()` in class `java.io.Writer`

抛出:

`IOException`; 如果一个 I/O 错误发生。

5) `public int getBufferSize()`

这个方法返回 `JspWriter` 使用的缓冲区大小。

返回:

以字节为单位返回缓冲区的大小,或没有缓冲区返回 0。

6) public abstract int getRemaining()

这个方法以字节为单位返回缓冲区中未使用的大小。

返回:

缓冲区中未使用大小的字节数。

7) public boolean isAutoFlush()

这个方法指明 JspWriter 是否是 autoFlush。

返回:

如果当前 JspWriter 是 autoFlush 的,或当缓冲区溢出时抛出 IOException。

8) public abstract void newLine()

写一个行结束符。行结束符字符串 line.separator 由系统属性定义。并且新行字符“\n”不是必须的。

抛出:

IOException: 如果 I/O 错误发生。

9) public abstract void print(boolean b)

打印一个 boolean 值。java.lang.String.valueOf(boolean)产生的字符串依据平台默认字符编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写出。

参数:

b: The boolean to be printed

抛出:

java.io.IOException

10) public abstract void print(char c)

打印一个字符。字符依据平台默认字符编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写出。

参数:

c: 欲打印的字符。

抛出:

java.io.IOException

11) public abstract void print(char[] s)

打印一个字符数组。字符依据平台默认字符编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写出。

参数:

s: 欲打印的字符数组。

抛出:

NullPointerException: 如果数组为 null。

java.io.IOException

12) public abstract void print(double d)

打印一个双精度浮点数。java.lang.String.valueOf(double)产生的字符串将依据平台默认编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写

出。

参数:

d: 欲打印的双精度浮点数。

抛出:

java.io.IOException

参照:

java.lang.Double

13) public abstract void print(float f)

打印一个浮点数。java.lang.String.valueOf(float)产生的字符串将依据平台默认编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写出。

参数:

f: 欲打印的浮点数。

抛出:

java.io.IOException

参照:

java.lang.Float

14) public abstract void print(int i)

打印一个整数。java.lang.String.valueOf(int)产生的字符串将依据平台默认编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写出。

参数:

i: 欲打印的整数。

抛出:

java.io.IOException

参照:

java.lang.Integer

15) public abstract void print(long l)

打印一个长整数。java.lang.String.valueOf(long)产生的字符串将依据平台默认编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写出。

参数:

l: 欲打印的长整欲。

抛出:

java.io.IOException

参照:

java.lang.Long

16) public abstract void print(java.lang.Object obj)

打印一个对象。java.lang.String.valueOf(Object)方法产生的字符串将依据平台默认编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写出。

参数:

obj: 欲打印的对象。

抛出:

java.io.IOException

参照:

java.lang.Object.toString()

17) public abstract void print(java.lang.String s)

打印一个字符串。如果参数为 null,那么字符串“null”将被打印。否则,这个字符串的字符将依据平台默认编码被转换成字节,并且这些字节以方法 java.io.Writer.write(int) 正确的风格写出。

参数:

s:欲打印的字符串。

抛出:

java.io.IOException

18) public abstract void println()

通过写一个行结束符终止当前行。行结束符由系统属性 line.separator 定义,新行字符“\n”不是必须的。

抛出:

java.io.IOException

19) public abstract void println(boolean x)

打印一个 boolean 值,然后终止该行。这个方法的行为通过调用方法 public abstract void print(boolean b)和 public abstract void println()达到。

抛出:

java.io.IOException

20) public abstract void println(char x)

打印一个字符,然后终止该行。这个方法的行为通过调用方法 public abstract void print(char c)和 public abstract void println()达到自身行为。

抛出:

java.io.IOException

21) public abstract void println(char[] x)

打印一个字符数组,然后终止该行。这个方法的行为通过调用方法 public abstract void print(char[])和 public abstract void println()达到自身行为。

抛出:

java.io.IOException

22) public abstract void println(double x)

打印一个双精度浮点数,然后终止该行。这个方法的行为通过调用方法 public abstract void print(double d)和 public abstract void println()达到自身行为。

抛出:

java.io.IOException

23) public abstract void println(float x)

打印一个浮点数,然后终止该行。这个方法的行为通过调用方法 public abstract void

`print(float f)`和 `public abstract void println()`达到自身行为。

抛出：

`java.io.IOException`

24) `public abstract void println(int x)`

打印一个整数,然后终止该行。这个方法的行为通过调用方法 `public abstract void print(int i)`和 `public abstract void println()`达到自身行为。

抛出：

`java.io.IOException`

25) `public abstract void println(long x)`

打印一个长整数,然后终止该行。这个方法的行为通过调用方法 `public abstract void print(long l)`和 `public abstract void println()`达到自身行为。

抛出：

`java.io.IOException`

26) `public abstract void println(java.lang.Object x)`

打印一个对象,然后终止该行。这个方法的行为通过调用方法 `public abstract void print(java.lang.Object obj)`和 `public abstract void println()`达到自身行为。

抛出：

`java.io.IOException`

27) `public abstract void println(java.lang.String x)`

打印一个字符串,然后终止该行。这个方法的行为通过调用方法 `public abstract void print(java.lang.String s)`和 `public abstract void println()`达到自身行为。

抛出：

`java.io.IOException`

### 9.1.3 一个实现实例

JSP 实现类从它的 `_jspService()`方法开始,通过实现默认 `JspFactory` 可建立依赖这个抽象基类的子类实现的一个实例。下例是使用上述方法建立的一个实现实例。

```
public class foo implements Servlet {
    //...

    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException {
        JspFactory factory = JspFactory.getDefaultFactory();
        PageContext pageContext = factory.getPageContext(
            this,
            request,
            response,
            null, // errorPageURL
            false, // needsSession
```

```
JspWriter.DEFAULT_BUFFER,
true // autoFlush
);
// initialize implicit variables for scripting env ...
HttpSession session = pageContext.getSession();
JspWriter out = pageContext.getOut();
Object page = this;
try {
// body of translated JSP here ...
}
catch (Exception e) {
out.clear();
pageContext.handlePageException(e);
}
finally {
out.close();
factory.releasePageContext(pageContext);
}
}
```

## 9.2 Exceptions

类 `JspException` 是所有 JSP 异常的基类。`JSPTag-Exception` 由标记扩展机制使用。

### 9.2.1 JspException

#### 1. 语法

```
public class JspException extends java.lang.Exception
```

#### 2. 直接已知子类

`JspTagException`

#### 3. 所有实现界面

`java.io.Serializable`

#### 4. 接述

JSP 引擎已知的一个一般异常。不可捕捉 `JspExceptions` 将导致 `errorpage` 机制的调

用。

## 5. 构造器

1) `public JspException()`

构造一个 `JspException`。

2) `public JspException(java.lang.String msg)`

以指定的信息构造一个新的 JSP 异常。信息可写进服务端日志和/或显示给用户。

参数：

`msg`: 一个指定异常信息文本的字符串。

3) `public JspException(java.lang.String message, java.lang.Throwable rootCause)`

当 JSP 需要抛出一个异常时, 构造一个新的 JSP 异常并且包含有干扰正常操作的“root cause”异常的信息(包含描述信息)。

参数：

`message`: 一个包含异常信息文本的字符串。

`rootCause`: 干扰 servlet 的正常操作, 使当前 servlet 必然发生异常的一个 `Throwable` 异常。

4) `public JspException(java.lang.Throwable rootCause)`

当 JSP 需要抛出一个异常时, 构造一个新的 JSP 异常并且包含有干扰正常操作的“root cause”异常的信息(包含描述信息)。这个异常的信息是以底层异常的本地信息为基础的。这个方法在 `Throwable` 异常上调用 `getLocalizedMessage` 方法可获得本地异常信息。对于类 `JspException`, 这个方法可被覆盖以建立一个异常信息来表达特殊的现场(本地信息)。

参数：

`rootCause`: 干扰 servlet 的正常操作, 使当前 servlet 必然发生异常的一个 `Throwable` 异常。

## 6. 方法

`public java.lang.Throwable getRootCause()`

返回导致当前 JSP 发生例外(异常)的异常。

返回：

导致当前 JSP 异常的 `Throwable` 对象。

### 9.2.2 JspTagException

#### 1. 语法

`public class JspTagException extends JspException`

#### 2. 所有实现界面

`java.io.Serializable`



### 3. 描述

标记处理使用的异常以标识一些不可恢复的错误。这个错误可被 JSP 页面的顶层捕获到,它将导致一个错误页面。

### 4. Constructors

(1) `public JspTagException()`

没有信息的构造器。

(2) `public JspTagException(java.lang.String msg)`

有信息的构造器。

## 9.3 JSP 页面实现对象与容器的联系

这节描述 JSP 页面实现对象和它的容器之间的基本关系。主要关系由类 `JspPage` 和 `HttpJspPage` 定义。`JspFactory` 类描述简便实例化所有需要运行时对象的机制。`JspEngineInfo` 提供当前 JSP 容器的基本信息。

注意这儿描述的类无意让 JSP 页面作者使用。因此如果希望更深入了解 JSP,那么不要放过下述内容。

### 9.3.1 JspPage

#### 1. 语法

```
public interface JspPage extends javax.servlet.Servlet
```

#### 2. 所有已知子界面

`HttpJspPage`

#### 3. 所有超界面

`javax.servlet.Servlet`

#### 4. 描述

`JspPage` 界面描述一个 JSP 页面实现类必须实现的一般交互,使用 HTTP 协议的页面由 `HttpJSPPage` 界面描述。

`JspPage` 对象可以从 `JspFactory` 对象中获得。

#### 5.2+1 个方法

这个界面使用三个方法定义一个协议,其中只有两个:`jspInit()`和 `jspDestory()`是这

个界面的一部分。至于第三个方法：`_jspService()`的用法依赖于所使用的特定协议，并且它不是以 Java 通常方式表示的。

实现这个界面的类的任务是在适当的时候调用上面的方法。

`jspInit()`和`jspDestroy()`方法可以由 JSP 作者重新定义，但是`_jspService()`是由 JSP 处理器根据 JSP 页面的内容自动定义的。

`_jspService()`

`public void _jspService (ServletRequestSubtype request, ServletResponseSubtype response)`

`throws ServletException, IOException;`

`_jspService()`方法对应于 JSP 页面主体，这个方法由 JSP 容器自动定义而绝不会由 JSP 页面作者定义。

如果用 `extends` 属性为 JSP 页面指定了一个超类，那么这个超类可能会在调用`_jspService()`方法前或后在其自身的 `service()`方法中执行某些动作。

特殊的用法取决于 JSP 页面支持的协议。

## 6. 方法

1) `public void jspDestroy()`

当 JSP 页面将被销毁时，`jspDestroy()`方法被调用。JSP 页面可通过在声明元素里包含一个定义覆盖这个方法。JSP 页面可从 Servlet 入手重新定义 `destroy()`方法。

2) `public void jspInit()`

JSP 页面初始化时，`jspInit()`方法被调用。它的任务是 JSP 实现(如果有 `extends` 属性，还应当实现 `extends` 属性指定的类)。在这点上，使用 `getServletConfig()`方法可返回需要的值。JSP 页面可通过声明中包含一个定义覆盖这个方法。JSP 页面可从 Servlet 入手重新定义 `init()`方法。

## 9.3.2 HttpJspPage

### 1. 语法

`public interface HttpJspPage extends JspPage`

### 2. 所有超界面

`JspPage, javax.servlet.Servlet`

### 3. 描述

`HttpJspPage` 界面描述的是 JSP 页面实现类使用 HTTP 协议时，其必须满足的交互要求。

`HttpJspPage` 对象从 `JspFactory` 类中获得，其行为与 `JspPage` 一样，除了 `_jspService` 的实现。现在 `_jspService` 可以使用 Java 的类型系统表达，并且明确地被包含在界面中。

## 4. 方法

- (1) `public void _jspService(javax.servlet.http.HttpServletRequest request,`
- (2) `javax.servlet.http.HttpServletResponse response)`

`_jspService` 方法对应于 JSP 页面的主体。这个方法由 JSP 容器自动定义而绝不会由 JSP 页面作者定义。

如果用 `extends` 属性为 JSP 页面指定了一个超类,那么这个超类可能会在调用 `_jspService()` 方法前或后在其自身的 `service()` 方法中执行某些动作。

抛出:

`IOException, ServletException`

## 9.3.3 JspFactory

### 1. 语法

```
public abstract class JspFactory
```

### 2. 描述

`JspFactory` 是一个抽象类,定义了大量有效的 `Factory` 方法以建立用于支持 JSP 实现的不同界面和类的实例。

兼容 JSP 引擎的一个实现,在初始化这个类的相关子类的实例过程中,通过注册使用静态方法 `setDefaultFactory()` 建立的实例,使得它对 JSP 实现类全局(作用范围)有效。

`PageContext` 和 `JspEngineInfo` 类都是只能从 `Factory` 建立的实现相关类。



注意:`JspFactory` 对象不能被 JSP 页面作者使用。

### 3. 构造器

```
public JspFactory()
```

### 4. 方法

- 1) `public static synchronized JspFactory getDefaultFactory()`

获得当前实现的默认 `Factory` 设置或默认的预先设置。

返回:

当前实现的默认工厂设置。

- 2) `public abstract JspEngineInfo getEngineInfo()`

获得当前 JSP 引擎的特殊实现信息。

返回:

描述当前 JSP 引擎的 `JspEngineInfo` 对象。

- 3) public abstract PageContext getPageContext(javax.servlet.Servlet servlet,
- 4) javax.servlet.ServletRequest request, javax.servlet.ServletResponse response,
- 5) java.lang.String errorPageURL, boolean needsSession, int buffer, boolean autoflush)

为调用 Servlet 和当前尚未处理的 request 和 response 去获取与实现相关的 javax.servlet.jsp.PageContext 抽象类的一个实例。

这个方法的典型的应用是在处理 JSP 实现类的 \_jspService 方法之前调用以获得一个 PageContext 对象,用于尚未处理的请求。

调用这个方法将导致 PageContext.initialize() 方法被调用,如果初始化正确,返回 PageContext。调用 releasePageContext 释放所有通过这个方法获得的 PageContext 对象。

参数:

servlet:处于请求状态的 servlet。

config:处于请求状态的 servlet 的 ServletConfig。

request:尚未处理的当前 request。

response:尚未处理的当前 response。

errorPageURL:处于请求状态的 JSP 页面的错误处理页面的 URL 或 null。

needsession:如果 JSP 使用会话管理为真。

buffer:以 bytes 为单位的缓存大小。如果没有缓存,为 PageContext \_NO \_BUFFER 如果默认实现,为 PageContext.DEFAULT \_BUFFER。

autoflush:buffer 溢出时,缓存自动刷新到输出流,或者抛出一个 IOException 异常。

返回:

页面上下文。

- 6) public abstract void releasePageContext(PageContext pc)

调用以释放先前分配的 PageContext 对象,导致 PageContext.release() 方法被调用。这个方法在 JSP 实现类的 \_jspService() 方法返回之前调用。

参数:

pc:一个先前由 getContext() 获得的 PageContext。

- 7) public static synchronized void setDefaultFactory(JspFactory deflt)

设置这个实现为默认 Factory 值。任何不在 JSP 引擎运行时对这个方法的调用都是违法的。

参数:

default:默认工厂实现。

### 9.3.4 JspEngineInfo

#### 1. 语法

```
public abstract class JspEngineInfo
```

#### 2. 描述

JspEngineInfo 是一个抽象类,提供当前 JSP 引擎的信息。

### 3. 构造器

```
public JspEngineInfo()
```

### 4. 方法

```
public abstract java.lang.String getSpecificationVersion()
```

返回当前 JSP 引擎支持的 JSP 规范的版本号。

返回：

返回当前 JSP 引擎支持的 JSP 规范的版本号,如果是未知的,返回 null。

## 9.4 计数器

毫不夸张地说,实现了这个计数器后,JSP 已经学会了七成了。为什么这么说,其中一个原因是以前认为未经深思熟虑不可覆盖 `jspInit()` 和 `jspDestroy()` 方法,但是在实现了计数器后,会发觉跟编写 Servlet 没有什么两样。

### 9.4.1 计数器的实现

#### 1. 页面效果

首先来看看页面效果,见图 9-1。

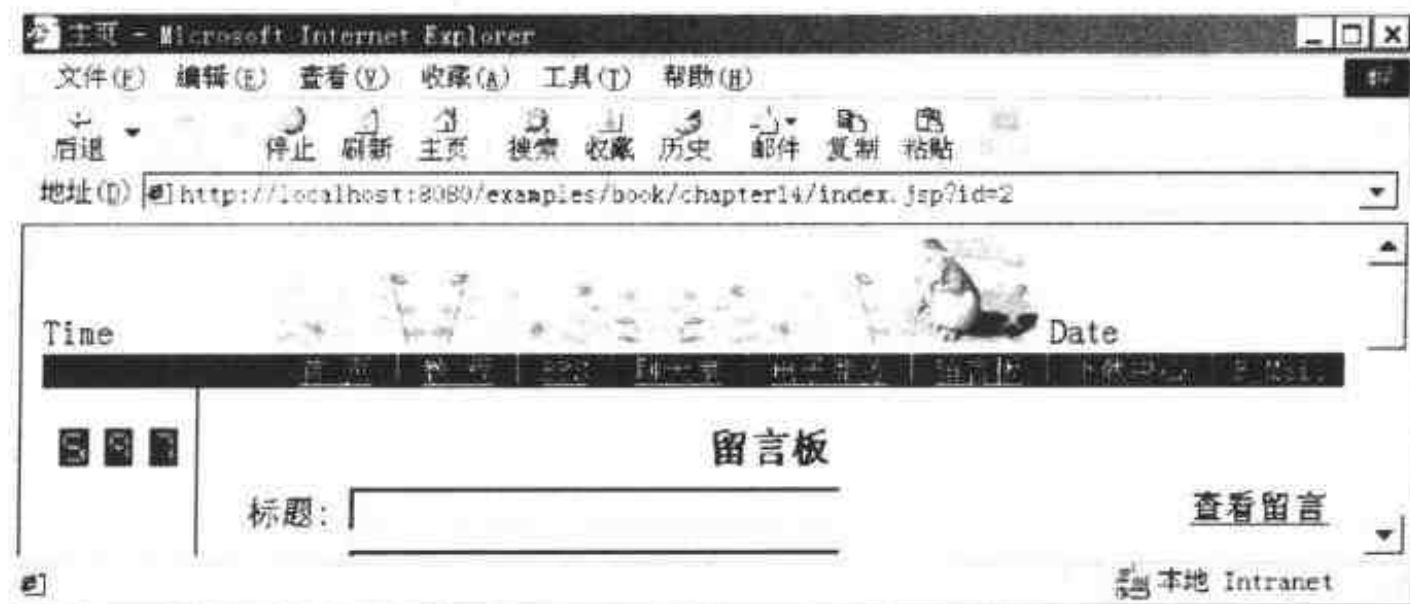


图 9-1 计数器在页面上的效果

#### 2. 页面源代码

这个计数器实现的源代码如下：

```
<%@ page import = "java.io.*" %>
<%! int test; %>
<%!
```

```

public void jspInit() {
    try {
        File file = new
        File("../webapps/examples/book/chapter14/counter/counter.txt");
        RandomAccessFile raf = new RandomAccessFile(file, "r");
        test = Integer.parseInt(raf.readLine());
    }
    catch (FileNotFoundException e) {
        System.out.println("file not found:" + e);
    }
    catch (IOException e) {
        System.out.println("io wrong" + e);
    }
}

% >
< % !
public void jspDestroy() {
    try {
        File file = new
        File("../webapps/examples/book/chapter14/counter/counter.txt");
        RandomAccessFile raf = new RandomAccessFile(file, "rw");
        raf.writeBytes(Integer.toString(test));
        raf.close();
    }
    catch (FileNotFoundException e) {
        System.out.println("file not found:" + e);
    }
    catch (IOException e) {
        System.out.println("io wrong" + e);
    }
}

% >
< % ! int i; % >
< % ! String temp; % >
< % temp = Integer.toString(++test); % >
< br >
< % for(i=0;i<temp.length();i++) | % >
< img src="/examples/book/chapter14/counter/pic/< % = temp.charAt(i) % > .gif" >
< /img >

```

<% } %>

### 3. 说明及改进建议

就是这样简单。初始化的时候将记数值读出,到服务器关闭的时候将记数值写回文件。这个实现显然就比我们在第 5 章实现的效率高多了。唯一需要注意的是在初始化的时候不能使用内部对象,因为此时内部对象还不存在。这与编写 Servlet 没有什么不同。这一点是很容易理解的,通过第 8 章的学习,可以知道其实 JSP 最终是要翻译成 Servlet 类的。

这个计数器并非完美无缺,如果将记录记数值的文件 counter.txt 删除或移动位置,那么这个程序仍然是可以运行的,但是它的起始值总是 0,因为它并没有将记数值记录下来。改进建议是初始化时如果文件未找到就退出程序运行。注意不要使用 System.exit() 方法。

## 9.5 本章小结

本章介绍了 JSP 的核心 API,有些 API 并非提供给页面作者使用。但是了解它,对理解 JSP 的实现是非常有益的。一些 API 提供给页面作者使用,需要注意的是 PageContext、JspWriter 和 JspPage 的方法。最后我们覆盖了 JspPage 的 jspInit() 和 jspDestroy() 方法实现了一个比较实用的计数器。

# 第 10 章 JSP 对 JDBC 的集成

JDBC 是 Sun 基于 X/Open SQL CLI 的数据库技术。JDBC 最大的特点是:对下,JDBC 封装了各种底层数据源之间的差异;对上,JDBC 提供标准的 SQL 界面。这使得上层应用对底层数据源的访问完全透明,大大地简化了访问底层数据源的复杂性,真正做到了无障碍沟通。

## 10.1 关系数据库标准语言 SQL

在关系数据库大行其道(非关系系统的产品大都加上了关系接口)的今天,怎能不知关系数据库标准语言 SQL。SQL(即英文 structured query language 的缩写),译成中文为结构化查询语言,是一种介于关系代数与关系演算之间的语言,其功能包括查询、操纵、定义和控制 4 个方面,是一个通用的、功能极强的关系数据库语言。目前已成为关系数据库的标准语言。

### 10.1.1 SQL 概述

SQL 语言是 1974 年由 Boyce 和 Chamberlin 提出的。1975 年至 1979 年 IBM 公司 San Jose Research Laboratory 研制的关系数据库管理系统原型系统 System R 实现了这种语言。由于它功能丰富,语言简洁,使用方法灵活,倍受计算机用户及计算机工业界欢通,被众多计算机公司和软件公司所采用。经各公司的不断修改、扩充和完善,SQL 语言最终发展成为关系数据库的标准语言。

第一个 SQL 标准是 1986 年 10 月由美国国家标准局(ANSI)公布的,所以也称该标准为 SQL-86。1987 年国际标准化组织(ISO)也通过了这一标准。此后 ANSI 不断修改和完善 SQL 标准,并于 1989 年第二次公布了 SQL 标准(SQL-99),1992 年又公布了 SQL-92 标准。目前 SQL 最新标准为 SQL-99。

SQL 语言之所以能够为用户和业界所接受,成为国际标准,是因为它是一个综合的、通用的、功能极强的,同时又简单易学的语言。由于设计巧妙,语言仅使用了 9 个动词就完成了数据查询、数据定义、数据操纵、数据控制这 4 个核心功能。9 个动词及相应功能如表 10-1 所示。



表 10-1 SQL 语言的动词及其功能

SQL 功能	动 词
数据定义	CREATE, DROP, ALTER
数据查询	SELECT
数据操纵	INSERT, UPDATE, DELETE
数据控制	GRANT, REVOTE

下面我们就按表 10-1 的分类及顺序讲述 SQL 语言。

### 10.1.2 数据定义

关系数据库由模式、外模式和内模式组成,即关系数据库的基本对象是表、视图和索引。因此 SQL 的数据定义功能包括定义表、定义视图和定义索引,如表 10-2 所示。

表 10-2 SQL 的数据定义语句

操作对象	操 作 方 式		
	创 建	删 除	修 改
表	CREATE TABLE	DROP TABLE	ALTER TABLE
视图	CREATE VIEW	DROP VIEW	
索引	CREATE INDEX	DROP INDEX	

#### 1. 定义、删除与修改基本表

##### 1) 定义基本表

建立数据库最重要的一步就是定义一些基本表。SQL 语言使用 CREATE TABLE 语句定义基本表,其一般格式如下:

```
CREATE TABLE <表名>(
    <列名> <数据类型> [列级完整性约束条件]
    [, <列名> <数据类型> [列级完整性约束条件]...]
    [, <表级完整性约束条件>]);
```

例:

建立一个“Article”表,它由唯一的不空的序号 ID、文章标题 Article、文章内容 Content、作者名 Username、所属版面 HypotaxissubID、发表日期 Data、点击次数 Clickcount、回复哪篇文章 ReplyID 组成。

```
CREATE TABLE Article(
    ID                BIGINT NOT NULL UNIQUE,
    Article            VARCHAR[50],
    Content            LONGVARCHAR[65535],
```

```

        Username          CHAR[30],
        HypotaxissubID    INT,
        ReplyID           INT,
Data          DATA,
        Clickcount        INT );

```

上例中的类型名都是 SQL 的标准类型名。常用的 SQL 类型及与 Java 类型的映射如表 10-3 所示。

表 10-3 常用 SQL 类型及与 Java 类型映射

SQL 类型	Java 类型	SQL 类型	Java 类型
CHAR	STRING	BIGINT	long
VARCHAR	String	REAL	float
LONGVARCHAR	String	FLOAT	double
NUMERIC	java.math.BigDecimal	DOUBLE	double
DECIMAL	java.math.BigDecimal	BINARY	byte[]
BIT	Boolean	VARBINARY	byte[]
TINYINT	Byte	LONGVARBINARY	byte[]
SMALLINT	Short	DATE	java.sql.Date
INTEGER	Int	TIME	java.sql.Time
		TIMESTAMP	java.sql.Timestamp

## 2) 修改基本表

包括增加新列、增加新的完整性约束条件、修改原有的列定义或删除已有的完整性约束条件等。SQL 语言使用 ALTER TABLE 语句修改基本表,其一般格式为:

```


ALTER TABLE <表名>
    [ADD <新列名> <数据类型> [完整性约束]]
    [DROP <完整性约束名>]
    [MODIFY <列名> <数据类型>];

```

其中<表名>指定需要修改的基本表,ADD 子句用于增加新列和新的完整性约束条件,DROP 子句用于删除指定的完整性约束条件,MODIFY 子句用于修改原有的列定义。例:

向 Article 表增加“作者 ID”,其数据类型为 BIGINT。

```
ALTER TABLE Article ADD UserID BIGINT;
```

 注意:SQL 并没有提供删除属性列的语句。

## 3) 删除基本表

使用 SQL 语句 DROP TABLE 进行删除。其一般格式为:

```
DROP TABLE <表名>;
```

## 2. 建立与删除索引

建立索引是加快表的查询速度的有效手段。SQL 语言支持用户在基本表上建立一

个或多个索引。

#### 1) 建立索引

在 SQL 语言中,建立索引使用 CREATE INDEX 语句,其一般格式为:

```
CREATE [UNIQUE] [CLUSTER] INDEX <索引名>  
ON <表名>(<列名> [<次序>] [,<列名> [<次序>]]...);
```

索引可建在表的一列或多列上,各列名之间用逗号分隔。每个<列名>后面还可以用<次序>指定索引值的排列次序,包括 ASC(升序)和 DESC(降序)两种,缺省值为 ASC。

UNIQUE 表示此索引的每一个索引值只对应唯一的数据记录。

CLUSTER 表示要建立的索引是聚簇索引。所谓聚簇索引是指索引项的顺序与表中记录的物理顺序一致的索引组织。

例:

为 Article 表建立索引。按序号 ID 降序和点击次数 Clickcount 降序建立唯一索引。

```
CREATE UNIQUE INDEX NewBestArticle ON Article(ID DESC,Clickcount ASC);
```

#### 2) 删除索引

在 SQL 语言中,删除索引使用 DROP INDEX 语句,其一般格式为:

```
DROP INDEX <索引名>;
```

### 3. 建立与删除视图

视图是关系数据库系统提供给用户以多角度观察数据库中数据的重要机制。视图是从一个或多个基本表(或视图)导出的表。它与基本表不同,是一个虚表。

#### 1) 建立视图

在 SQL 语言中,用 CREATE VIEW 语句建立视图,其一般格式为:

```
CREATE VIEW <视图名> [( <列名> [,<列名>]...)]  
AS <子查询>  
[WITH CHECK OPTION];
```

其中子查询可以是任意复杂的 SELECT 语句(参见 10.1.3 小节数据查询)。WITH CHECK OPTION 表示对视图进行 UPDATE,INSERT 和 DELETE 操作时要保证更新、插入或删除的行,满足视图定义中的谓词条件(即子查询中的条件表达式)。

例:

在 Article 表上建立一个名为 CommonlyInfor 的视图,包括序号 ID、文章标题 Article、文章作者 Username、发表日期 Data、点击次数 Clickcount。

```
CREATE VIEW CommonlyInfor  
AS  
SELECT ID,Article,Username,Data,Clickcount  
FROM Article
```

因为子查询可以是任意复杂的 SELECT 语句,所以视图可建立在多个基本表上,甚至视图上。

## 2) 删除视图

在 SQL 语言中,需要使用 DROP VIEW 语句显式删除视图。

DROP VIEW <视图名>

## 10.1.3 数据查询

数据查询是数据库的核心操作。SQL 语言提供了 SELECT 语句进行数据的查询,该语句具有灵活的使用方式和丰富的功能。其一般格式为:

SELECT [ALL | DISTINCT] <目标列表达式> [, <目标列表达式>]...

FROM <表名或视图名> [, <表名或视图名>]...

[WHERE <条件表达式>]

[GROUP BY <列名 1> [HAVING <条件表达式>]]

[ORDER BY <列名 2> [ASC | DESC]];

SELECT 语句相当复杂,要在这么短的篇幅内讲清并非易事。因此我们借助表格的强大功能实现归类化和序列化。如表 10-4 所示,表中阴影部分是一个查询语句必须有的部分。

表 10-4 SELECT 语句

SELECT		
目标列表达式	* COUNT([DISTINCT   ALL] *) 属性列名表达式,是属性列、集函数和常量组成的四则运算公式。其中集函数一般格式为: COUNT   SUM   AVG   MAX   MIN([DISTINCT   ALL] <列名>)	表名. 对应前面的叙述。如 表名. *, COUNT(表名. *)
FROM	单表	多表
WHERE	比 较: =, >, <, >=, <=, != 或 <> 空 值: IS NULL, IS NOT NULL 确定范围: BETWEEN AND, NOT BETWEEN AND 确定集合: IN, NOT IN 字符匹配: LIKE, NOT LIKE 多重条件: AND, OR	连接查询 嵌套查询
GROUP BY	列名	
ORDER BY	升序: ASC 降序: DESC	

整个 SELECT 语句的含义是,根据 WHERE 子句的条件表达式,从 FROM 子句指定的基本表或视图找出满足条件的元组,再按 SELECT 子句中的目标列表达式,选出元组中的属性值形成结果表。如果有 GROUP 子句,则将结果按 <列名 1> 的值进行分组,该属性列值相等的元组为一个组,每个组产生结果表中的一条记录。通常会在每组中作用集函数。如果 GROUP 子句带 HAVING 短语,则只有满足指定条件的组才予输出。如果有 ORDER 子句,则结果表还要按 <列名 2> 的值的升序或降序排序。SELECT 功能十

分强大,可完成简单的单表查询,也可完成复杂的连接查询和嵌套查询。此外还有集合查询。本书只讲述前三种查询,而尤以单表查询为重点,因为集合查询几乎不会用到。连接查询和嵌套查询基本上就是 WHERE 子句复杂的单表查询。

下面以“BBS”数据库为例说明 SELECT 语句的各种用法。选取“BBS”数据库中三个表,Article 表、Subcatalog 表、User 表。下文只讨论了表的查询,事实上对视图这些查询同样有效。

Article 表记为: Article( ID, Article, Content, UserID, Username, ReplyID, Hypotaxis-subID, Data, Clickcount)

Subcatalog 表记为: Subcatalog( ID, Subcatalog, Comment, HypotaxiscatID)

User 表记为: User( ID, Username, Password, State)

## 1. 单表查询

### 1) 选择表中的若干列

#### ●查询表的所有列

例:

查询 Subcatalog 表的详细情况,即所有子版面详细资料。

```
SELECT *  
FROM Subcatalog;
```

#### ●查询表的指定列

例:

查询 Article 表的序号 ID、标题 Article、作者名 Username、日期 Data 和点击次数 Clickcount。

```
SELECT ID, Article, Username, Data, Clickcount  
FROM Article;
```

#### ●查询经过计算的列

例:

查询 Article 表中点击次数最大值。

```
SELECT MAX(Clickcount)  
FROM Article;
```

#### ●消除值重复

例:

查询发表过文章的用户。

```
SELECT DISTINCT Username  
FROM Article;
```

### 2) 查询满足条件的元组

#### ●比较大小

例:

查询点击次数大于 100 的文章标题。

```
SELECT Article
```

```
FROM Article
WHERE Clickcount > 100;
```

#### ●涉及空值的查询

例:

Article 表中,如果某篇文章不是回复的,而是新发表的,那么 ReplyID 就为空值(当然也可设计为一个特殊的值,例如 0。查询新发表的文章。

```
SELECT Article
FROM Article
WHERE ReplyID IS NULL;
```

#### ●确定范围

例:

查询 10 月份发表的文章总数。

```
SELECT COUNT(Article)
FROM Article
WHERE Data BETWEEN 10/1/1999 AND 10/31/1999;
```

#### ●确定集合

例:

假设 BBS 系统有两级版面,查询属于子版面 ID 为 8,9 的所有文章。

```
SELECT Article
FROM Article
WHERE HypotaxissubID IN (8,9);
```

#### ●字符匹配

字符匹配在查询中用得最为频繁,我们多举几个例子。

谓词 LIKE 可以用来进行字符串的匹配。其一般语法格式如下:

[NOT] LIKE '<匹配串>' [ESCAPE '<换码字符>']

其含义是查找指定的属性列值与<匹配串>相匹配的元组。<匹配串>可以是一个完整的字符串,也可以含有通配符%和\_。其中:

%(百分号) 代表任意长度(长度可以为 0)的字符串。

\_(下划线) 代表任意单个字符。

例:

查询一个名为“onlyyou”的用户写的所有文章。

```
SELECT Article
FROM Article
WHERE Username LIKE 'onlyyou';
```

例:

查询以“onlyyou”开头的名字,及其状态(普通用户、斑竹(某网络 BBS 的主持人的称呼)、系统管理员)。

```
SELECT Username, State
FROM User
```

```
WHERE Username LIKE 'onlyyou%';
```

例:

查询以“onlyyou”开头且其后只有一个字符的名字及其状态。

```
SELECT Username, State  
FROM User  
WHERE Username LIKE 'onlyyou _';
```

例:

查询以“onlyyou \_a”开头的且其后只有一个字符的名字,及其状态。

```
SELECT Username, State  
FROM User  
WHERE Username LIKE 'onlyyou \_ a _' ESCAPE '\';
```

#### ●对查询结果分组

GROUP BY 子句可以将查询结果表的各行按一列或多列取值相等的原则进行分组。

例:

考察哪个版面最受欢迎。

```
SELECT HypotaxissubID, COUNT(Article)  
FROM Article  
GROUP BY HypotaxissubID;
```

#### ●对查询结果排序

ORDER BY 子句可以将查询结果按一个或多个属性列的升序(ASC)或降序(DESC)重新排列。

例:

将文章以时间降序排列。

```
SELECT Article, Data  
FROM Article  
ORDER BY Data;
```

## 2. 连接查询

一个数据库中的多个表之间一般都存在某种内在联系,因此,同时涉及多个表的连接查询,实际上是关系数据库中最主要的查询,主要包括等值连接查询、非等值连接查询、自身连接查询、外连接查询和复合条件连接查询。

### 1) 等值连接查询

使用比较运算符“=”进行的连接查询叫等值连接查询。

例:

考察所有用户发表文章的情况(例如包括 Article 表中的文章 ID、标题、发表日期、用户表中的用户ID、用户名)。在我们选择的目标列中有同名列,这种情况下必须在其前面加表名前缀。

```
SELECT Article. ID, Article. Article, Article. Data, User. ID, User. Username  
FROM Article, User
```

```
WHERE User.ID = Article.UserID;
```

#### 2) 非等值连接查询

使用比较运算符 >、<、>=、<=、!= 等进行的连接查询叫非等值连接查询。

例:

```
SELECT Article.ID, Article.Article, Article.Data, Article.Username  
FROM Article, User  
WHERE Article.Username != User.Username;
```

这个查询有什么意义呢? 假设一个 BBS 系统允许未注册的用户发表文章, 那么这个查询就是所有未注册用户发表文章的情况。

#### 3) 自身连接查询

一个表与其自身进行的连接查询叫自身连接查询。

例:

递归查询每篇文章的根文章(即新发的文章)。递归表达式如下, 其中 FIRST, SECOND 是表 Article 的别名。

```
SELECT FIRST.ID, FIRST.Article, SECOND.ReplyID  
FROM Article FIRST, Article SECOND  
WHERE FIRST.ReplyID = SECOND.ID
```

#### 4) 外连接查询

例:

```
SELECT Article.ID, Article.Article, Article.Data, User.ID, User.Username  
FROM Article, User  
WHERE User.ID = Article.UserID( * );
```

实际上, 本例是等值连接查询的变例, 两者有什么不同呢? 如果有用户从来没发表过文章, 那么后者就不会将这些用户的信息输出, 而前者(本例)则将这些用户有的信息如用户 ID、用户名输出, 而将没有的信息置为空。本例这样的查询叫外连接查询。

#### 5) 复合条件连接查询

WHERE 子句中有多个条件的连接查询叫复合条件连接查询。

例:

查询用户 onlyyou 在版面 7 发表的所有文章

```
SELECT ID, Article  
FROM Article  
WHERE Username = 'onlyyou' AND  
HypotaxissubID = 7;
```

### 3. 嵌套查询

在 SQL 语言中, 一个 SELECT—FROM—WHERE 语句称为一个查询块。将一个查询块嵌套在另一个查询块的 WHERE 子句或 HAVING 短语的条件中的查询称为嵌套查询或子查询。嵌套查询分为: 带有 IN 谓词的子查询、带有比较运算符的子查询、带有 ANY 或 ALL 谓词的子查询和带有 EXISTS 谓词的子查询。



## 1) 带有 IN 谓词的子查询

例:

查询某篇文章的相关文章( ReplyID 相同)。

```
SELECT ID,Article
FROM Article
WHERE ReplyID IN
(SELECT ReplyID
FROM Article
WHERE Article = 'articlename');
```

## 2) 带有比较运算符的子查询

例:

因为每篇文章至多回复一篇文章,所以上例还可以这样实现:

```
SELECT ID,Article
FROM Article
WHERE ReplyID =
(SELECT ReplyID
FROM Article
WHERE Article = 'articlename');
```

可见,条条道路通罗马,问题的求解从来就不是唯一的。当然不同的方法其效率是不一样的,重要的是每个人可以找到一种自然的求解方法,因为关系数据库本身可进行查询优化。事实上,关系数据库发展如此迅猛与其以关系代数为基础可进行关系演算,从而可实现最优查询是分不开的。

## 3) 带有 ANY 或 ALL 谓词的子查询

例:

评选最佳作者用户,即发表的某篇文章点击次数值最大。一种实现方法如下:

```
SELECT UserID,Username
FROM Article
WHERE Clickcount > = ANY
(SELECT Clickcount
FROM Article);
```

## 4) 带有 EXISTS 谓词的子查询

带有 EXISTS 谓词的子查询不返回任何实际数据,它只产生逻辑真值“true”或逻辑假值“false”。使用存在量词 EXISTS 后,若内层查询结果非空,则外层的 WHERE 子句返回真值,否则返回假值。这种查询比较简单,我们不再给出例子。

## 10.1.4 数据更新

SQL 中数据更新包括基本表和视图的更新,二者都有插入数据、修改数据和数据删除三条语句。由于视图实际上是不存在的虚表,因此对视图的更新,最终都要转换或对基本

表的更新。所以对视图的更新有一定的限制,除此之外与基本表的更新没有什么区别。因此为了抓住重点,节约篇幅,本书只讲述基本表的更新。

## 1. 插入数据

SQL 的数据插入语句 INSERT 通常有两种形式。一种是插入一个元组,另一种是插入子查询结果。后者可以一次插入多个元组。

### 1) 插入单个元组

插入单个元组的 INSERT 语句的格式为:

```
INSERT  
  INTO <表名> [(<属性列 1> [,<属性列 2>...])]  
  VALUES(<属性列 1> [,<属性列 2>]...)
```

例:

将一个新开子版面信息(9,'sayyousayme','talk about you and me',4)插入 Subcatalog 表。

```
INSERT  
  INTO Subcatalog  
  VALUES(9,'sayyousayme','talk about you and me',4);
```

### 2) 插入子查询结果

功能是批量插入,一次将子查询的结果全部插入指定表中。插入子查询结果的 INSERT 语句的格式为:

```
INSERT  
  INTO <表名> [(<属性列 1> [,<属性列 2>...])]  
  子查询;
```

例:

将 BBS 中的精华文章放入另一个表中。假定点击次数大于 1000 的为精华文章,新表为 Prime(Article,Content)。

```
INSERT  
  INTO Prime(Article,Content)  
  SELECT Article,Content  
  FROM Article  
  WHERE Clickcount > 1000;
```

## 2. 修改数据

修改数据又称为更新操作,其语句一般格式为:

```
UPDATE <表名>  
  SET <列名> = <表达式> [,<列名> = <表达式>]...  
  [WHERE <条件>];
```

其功能是修改指定表中满足 WHERE 子句条件的元组。其中 SET 子句用于指定用 <表达式> 的值取代相应的属性列值。如果省略 WHERE 子句,则表示要修改表中所有

元组。

1) 修改某一个元组的值

例:

BBS 中每点击一次某篇文章,该文章的 Clickcount 就应加 1。其实现 SQL 语句如下:

```
UPDATE Article
SET Clickcount = Clickcount + 1
WHERE ID = articleID;
```

2) 修改多个元组的值

例:

因某种原因需要删除某个子版面,但又要将部分精华文章放入另一个子版面中。假设欲删除子版面序号为 8,点击次数大于 500 的精华文章放入序号为 9 的子版面。其实现的 SQL 语句如下:

```
UPDATE Article
SET HypotaxissubID = 9
WHERE HypotaxissubID = 8 AND
Clickcount > 500;
```

上述两种类型是按修改的数量分类,还可按照 WHERE 子句是否带有子查询分类,分为“不带子查询”和“带子查询”的修改语句。其 WHERE 子句与嵌套查询的 WHERE 子句一样。这里不再讲述。

### 3. 删除数据

删除语句的一般格式为:

```
DELETE
FROM <表名>
[WHERE <条件>];
```

DELETE 语句的功能是从指定表中删除满足 WHERE 子句条件的所有元组。如果省略 WHERE 子句,表示删除表中全部元组,但表的定义仍在数据字典中。与修改分类相似,按删除元组数量分为两类。

1) 删除某一个元组的值

例:

删除某篇文章,假设删除序号为 699 的文章。

```
DELETE
FROM Article
WHERE ID = 699;
```

2) 删除多个元组的值

例:

如可以根据需要删除多篇文章或言论。

```
DELETE
FROM Article
```

```
WHERE UserID = userID;
```

也可按 WHERE 子句是否带有子查询分类,来查找并删除相关内容。

到这儿关系数据库标准语言 SQL 讲完了。我们不奢望每个读者都能在这么短的文字中学会用好 SQL。事实上,我们的目的仅仅是帮助读者温故知新,序列化和结构化相关知识,将书读薄。如果读者还有什么没有搞明白的,推荐清华大学出版社出版,王珊、陈红编著的《数据库系统原理教程》。10.1 节内容援引了该书第三章的结构和重要叙述,再结合将在后面实现的 BBS 系统数据库举了不少的例子,希望能起到承前启后的作用。在此郑重感谢原书作者。

## 10.2 JDBC 概述

JSP 是基于三层结构模型的网络应用程序,那么是谁提供底层数据库支持? JSP 与平台无关,那么又是谁赋予了它这种能力? JSP 功能是强大的,适合开发大型电子商务,那么又是谁提供强大的数据支持? 这一切的一切,归根到底都是因为有了 JDBC。事实上,JDBC 技术是 Java 技术的最重要部分之一,作为 Java 技术的集大成者 JSP 怎能不包含对 JDBC 的集成。那么 JDBC 到底是什么,JDBC 与其它数据库技术相比较,其优势在哪里?

### 10.2.1 JDBC 是什么

JDBC 就是 Java Database Connectivity。JDBC 实现了 Java 与数据库的互连,是一个定义了以下内容的一个 API 规范。

如何在 Java applet、application 或者 Servlet 中与数据源交互。

如何使用 JDBC 驱动程序。

如何编写 JDBC 驱动程序。

JDBC API 是 Java 程序设计语言中访问关系数据的接口,使 Java 能够在分布式、异种环境中与多个数据源实现交互。它是 Java 平台(包括 J2SE、J2EE)的一部分,有两个包 `java.sql` 和 `javax.sql`。现在这两个包已经成为 Java 核心框架的组成部分。

JDBC API 基于 X/Open SQL CLI (Data Management: SQL Call Level Interface (X/Open SQL CLI)) 的数据库技术。X/Open SQL CLI 也是 ODBC 的基础。JDBC 定义了一种自然的、易使用的、从 Java 程序设计语言到 X/Open SQL CLI 定义的抽象、概念以及 SQL 标准之间的影射。

JDBC API 是一种底层的 API,为工具和高层 API 提供了基础。

### 10.2.2 为什么有 JDBC

1995 年 11 月,Sun 已经认识到数据库连接性将对 Java 的成功产生巨大的影响。因为 Java 语言于 1995 年 5 月正式发布后,Java 应用日益广泛。然而在有关数据库应用程

序开发中,由于没有一个 Java 语言的数据库 API,开发人员不得不在 Java 程序中加入 C 语言的 ODBC 函数调用。这使得 Java 的很多优秀特性无法充分发挥,比如平台无关性、面向对象特性等。因此,需要一个具有连接到任意数据库功能的纯 Java API 的呼声越来越高,足以影响到 Java 的前景。

1996 年 6 月, Sun 发布了 JDBC1.0 版本。而此时 ODBC 作为一个基于 SQL 数据库引擎用 C 实现的接口,提供了一个与数据库通信和访问数据库元数据(数据库系统提供商的信息、数据被怎样存储等等)的统一接口,并已经成为数据库访问事实上的工业标准并在业界广泛使用。

事实上, ODBC 也并非完美无缺。首先, ODBC 实在是一个由非常有经验的开发人员才能使用的 API,需要有很多数据库经验。在为 ODBC 设计的上下文中,非常适合。然而与 Java 的风格完全是格格不入。

由此可见, Sun 公司研发 JDBC 是其战略上的需要,是 Sun 完善 Java 优点,尤其是完善其跨平台、简单性和面向对象的需要。事实证明, Java 联合 JDBC 提供了真正实现数据库应用程序可移植的解决方案。

### 10.2.3 JDBC 与 ODBC 的比较

到目前为止,微软的 ODBC 可能是用得最广泛的访问关系数据库的 API。它提供了连接几乎任何一种平台、任何一种数据库的能力。那么,为什么不直接在 Java 中使用 ODBC 呢?

回答是:虽然可以在 Java 中直接使用 ODBC,但最好是在 JDBC 协助下,用 JDBC—ODBC 桥实现,主要有下面几个方面的原因:

首先, ODBC 并不适合在 Java 中直接使用。ODBC 是一个 C 语言实现的 API,从 Java 程序调用本地的 C 程序会带来一系列如安全性、完整性、健壮性方面的缺点。

其次,完全精确的实现从 C 代码 ODBC 到 Java API 翻译的 ODBC 也并不令人满意。比如, Java 没有指针,而 ODBC 中大量使用了指针,包括极易出错的空指针“void \*”。

再次, ODBC 并不容易学习,它将简单特性和复杂特性混杂在一起,甚至对非常简单的查询都有复杂的选项。而 JDBC 刚好相反,它保持了简单性,但又允许复杂的特性。

最后, JDBC 对于纯 Java 方案来说是必须的。当使用 ODBC 时,人们必须在每一台客户机上安装 ODBC 驱动程序和驱动管理器。如果 JDBC 驱动程序是完全用 Java 语言实现的话,那么 JDBC 的代码就可以自动的下载和安装,并保证其安全性,而且,这将适应任何 Java 平台,从网络计算机 NC 到大型主机 Mainframe。

总而言之, JDBC API 是最能体现 SQL 基本概念的、直接的 Java 接口。它是在 ODBC 的基础上构建的, JDBC 保持了 ODBC 的基本设计特征,因此,熟悉 ODBC 的程序员将发现 JDBC 非常容易使用。

实际上,这两种接口都是基于 X/OPEN SQL 的调用级接口(CLI)。它们最大的不同是 JDBC 是基于 Java 的风格和优点,并强化了 Java 的风格和优点。

## 10.2.4 JDBC 的功能模型

### 1. 两层模型

两层模型将功能分成客户层和服务层,如图 10-1 所示。客户层包括功能层和一个或更多的 JDBC 驱动程序。对下列范围进行处理:

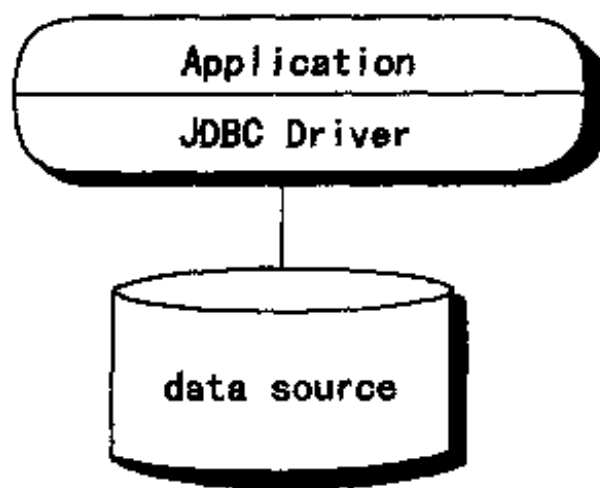


图 10-1 JDBC 两层模型

表示逻辑

商业逻辑

多状态事务处理或分布式事务处理的管理

资源管理

在这种模型中,应用程序直接与 JDBC 驱动程序交互,包括建立、管理物理连接和处理特殊的底层数据源实现的细节。应用程序可以使用数据源的特殊实现知识,以利用非标准的特性和作性能上的调整。

这个模型的缺点如下:

(1) 下部构造和系统级功能混淆了表达和商业逻辑,这阻碍了产生结构清晰、可维护的代码。

(2) 应用程序缺乏可移植性。因为它们变成一个特定数据库的实现。应用程序需要与多个数据库建立连接,必须知道不同提供商实现之间的差异。

(3) 受限的伸缩性。应用程序将控制一个或多个物理的数据库连接直至终止。限制了能支持的当前应用程序的个数。在这个模型里,性能、伸缩性、有效性问题均由 JDBC 驱动程序与相应的底层数据源控制。如果一个应用程序处理多个驱动程序,它也需要知道每个 Driver/Data Source 对的差异以解决这些问题。

### 2. 三层模型

三层模型介绍覆盖下部构造和商业逻辑的中间层服务(Middle \_ tier Server)。如图 10-2 所示。

这种结构设计,改善了商业应用的性能、伸缩性和有效性。功能分层如下:

1) Client tier

实现人们交流的表现逻辑。Java 程序、Web 浏览器都是典型的客户层实现。客户端

与中间层应用程序交互,不需要知道任何下部构造或数据源功能的信息。

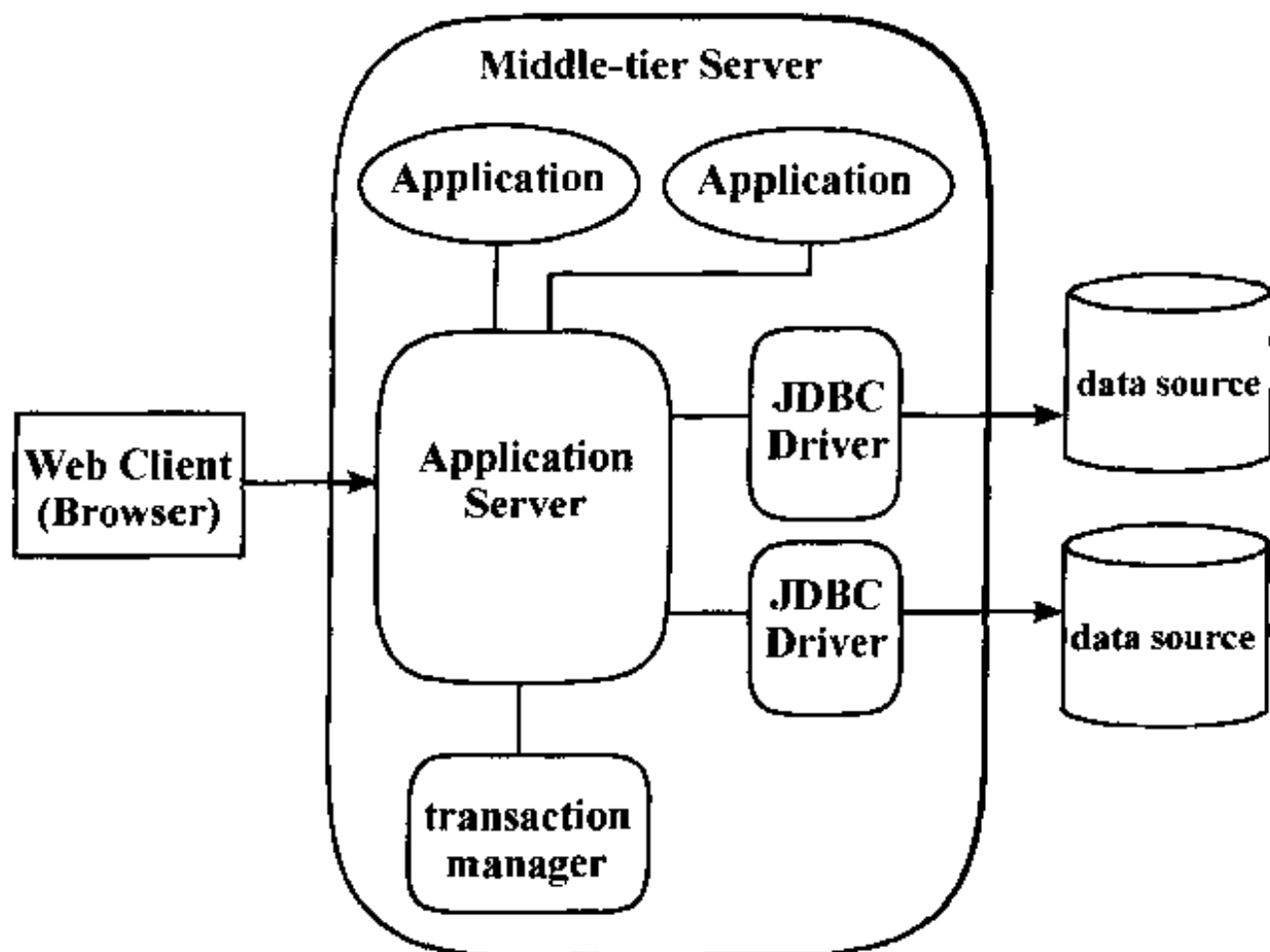


图 10-2 JDBC 三层模型

## 2) Middle\_tier Server

中间层包括:

(1) 应用程序,实现与客户端交流,实现商业逻辑。如果应用程序需要与数据源交互,它处理的是高层的抽象。例如,DataSource 对象和逻辑连接而不是低层的驱动程序 API。

(2) 应用程序服务,为广泛的应用程序提供下部构造支持。这包括物理连接池管理、事务管理和覆盖不同 JDBC 驱动程序的差异。最后一点使书写可移植的应用程序变得容易。应用程序服务角色可以由 J2EE Server 实现。应用程序服务实现了应用程序使用的高层抽象并与 JDBC 驱动程序直接交互。

(3) JDBC 驱动程序提供与底层数据源的连接,每个驱动程序实现标准的 JDBC API,而不管底层的数据源支持的特性如何。驱动程序层覆盖了标准的 SQL99 语法与数据源支持的本地方言之间的差异。如果数据源不是相关的 DBMS,驱动程序表示的是 Application Server 使用的相关层。

## 3) 底层数据源

数据所在层,可以包括关系 DBMS,传统文件系统,而向对象的 DBMS,数据仓库,电子数据表格或其它意义上的包和表示数据。唯一的要求是需要一个支持 JDBC API 的驱动程序。

## 10.2.5 JDBC 驱动程序的类型

Sun 定义了四种 JDBC 驱动程序的基本类型即对其它数据库访问 API 的映射、本地

API 半 Java 驱动程序、JDBC—NET 纯 Java 驱动程序、本地协议纯 Java 驱动程序。

1) 对其它数据库访问 API 的映射

这种类型的 JDBC 驱动程序是对其它数据库访问 API 的映射,如 ODBC。该类型的驱动程序一般与一个本地库相关。这种方式限制了它们的可移植性。JDBC—ODBC 桥驱动程序是典型的该类型的一个实例。

2) 本地 API 半 Java 驱动程序

驱动程序部分的用 Java 程序设计语言,部分的用本地代码书写。这种驱动程序对它们连接的数据源使用一个特定本地库(程序)。

3) JDBC—NET 纯 Java 驱动程序

驱动程序使用一个纯粹的 Java 客户端并且与一个使用数据库无关协议的中间服务端通信,然后中间服务端与数据源交流客户端请求。

4) 本地协议纯 Java 驱动程序

驱动程序使用纯粹的 Java 并对一个特定的数据源实现网络协议,客户端直接与数据源相连。

## 10.3 Tutorial

编写一个简单但是完整的例子将有助于抓住 JDBC 的主要特征。编写一个数据库应用程序的基本流程如下:

- (1)建立数据源(Data Source)
- (2)加载驱动程序(Register Driver)
- (3)建立连接(Connection)
- (4)建立语句对象(Statement)
- (5)如果有必要,添加数据到数据源
- (6)获取结果集合(ResultSet)
- (7)如果有必要,获取元数据(ResultSetMetaData)
- (8)处理数据
- (9)如果有必要,将处理结果写回数据源
- (10)关闭对象
- (11)处理警告、异常

不要看到有 11 步就被吓住了,其实非常简单。我们仅仅是希望能把这个流程表达得更清晰。

### 10.3.1 建立数据源

为抓住主要问题并保持简单,我们在 Win98 平台下建立一个名叫 Mybbs 的系统数据源,使用 Microsoft Access 驱动程序。同样为抓住主要矛盾,具体建立过程请参见 10.4.1 节叙述。下文假设已经建立该数据源,且登录名称为“loginname”,密码为“password”。



### 10.3.2 加载驱动程序

JDBC 的驱动程序有四种类型,这儿应该选择哪一种呢?不用说,大家都知道应该选择 JDBC—ODBC 桥驱动程序,因为我们建立的是 Win98 下的 ODBC 数据源。因此需要加载 JDBC—ODBC 桥。必须加载该驱动程序以告诉 JDBC 怎样与 ODBC 数据源联系,因此,需要类 `JdbcOdbcDriver`。

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
```

这一句的作用是在驱动程序管理器中注册 `sun.jdbc.odbc.JdbcOdbcDriver` 驱动程序。`Class.forName()` 是 Java 程序设计语言中的特殊构造方法。

### 10.3.3 建立连接

使用 URL 连接一个特定的 JDBC 数据源。对已加载的 JDBC—ODBC 桥,URL 有如下形式:

```
jdbc:odbc:data - source - name
```

本书例子为: `jdbc:odbc:Mybbs`

然后,请求建立 URL 指定的连接, `DriverManager` 选择一个适当的驱动程序见 10.3.2 小节显式指定加载的 `JdbcOdbcDriver` 驱动程序。建立连接有如下形式:

```
Connection con = DriverManager.getConnection(URL, username, password);
```

本书例子为:

```
Connection con = DriverManager.getConnection("jdbc:odbc:Mybbs", loginname, password);
```

上一句的作用是驱动程序管理器调用 `getConnection` 方法以便让 10.3.2 小节显式注册的驱动程序与指定数据源建立一个连接。

### 10.3.4 建立语句对象

与数据源建立连接后,我们就可以建立语句对象执行某些查询了,怎样建立语句对象呢?使用上一步建立的连接对象就可以获得语句对象了。建立语句对象有如下形式:

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

上一句的作用是连接对象调用 `createStatement` 方法建立一个语句对象。

到这一步,回想一下,我们做了些什么?首先建立了一个数据源,确定了操作了对象。接着,加载驱动程序告诉驱动程序管理器使用什么协议与数据源建立联系。然后由 `DriverManager` 调用方法 `getConnection` 实现上述协议的一个连接。最后由连接对象调用 `createStatement` 方法建立一个用于执行 SQL 的语句对象。在 JSP 中,你将获得类似下面的代码:

```
<%
```

```
//register driver
```

```

String driver="sun.jdbc.odbc.JdbcOdbcDriver";
Class.forName(driver).newInstance();
//create Connection
String ConnectionURL="jdbc:odbc:Mybbs";
Connection con=DriverManager.getConnection(ConnectionURL,loginname,
password);
//create Statement
Statement stmt=con.createStatement();

%>

```

通过上面步骤建立了语句对象,可以发送任何 SQL 语句执行需要的操作:查询、更新等。

### 10.3.5 添加数据到数据库

逢山开路,遇水搭桥。数据库中还没有表,因此需要建一张表。建的表就是我们在 SQL 语法叙述中使用得最多的 Article 表。在 10.1.2 小节中,建立的表就是它,应该还有印象。本书根据需要作了一点变化。下面是变化后在 JSP 中执行它的语句:

```

<%
String s="CREATE TABLE Article("
    +"ID          long NOT NULL UNIQUE,"
    +"Article      String(50),"
    +"UserID       long,"
    +"Username     String(30),"
    +"HypotaxissubID int,"
    +"ReplyID      int,"
    +"Clickcount   int);";

stmt.execute(s);

%>

```

有了表当然还要有数据,本书向 Article 表添加如下数据,如表 10-5 所示:

表 10-5 建立表所示数据库

ID	Article	UserID	Username	HypotaxissubID	ReplyID	Clickcount
1	Welcome...	9999	Admin	1	Null	80
633	Myfirst article	3689	Onlyyou	3	Null	90
756	My first too...	9973	Xixi	3	633	76

将上表数据插入数据库的 JSP 语句如下:

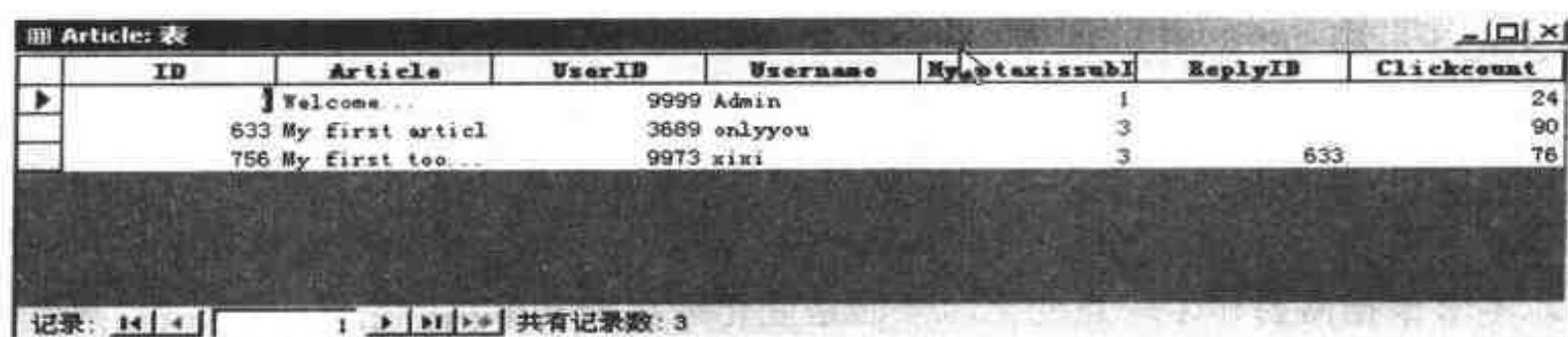
```

<%

```

```
String[] insertvalue = {
    "INSERT INTO Article VALUES(1,'Welcome...',9999,'Admin',1,null,
    80);",
    "INSERT INTO Article VALUES(633,'My first article',3689,'onlyyou',3,null,
    90);",
    "INSERT INTO Article VALUES(756,'My first too...',9973,'xixi',
    3,633,76);"};
    for(int i=0;i<insertvalue.length;i++){
        stmt.execute(insertvalue[i]);
    }
%>
```

到此,应该得到类似图 10-3 所示的数据库。



ID	Article	UserID	Username	MyotaxissubI	ReplyID	Clickcount
1	Welcome...	9999	Admin	1		24
633	My first article	3689	onlyyou	3		90
756	My first too...	9973	xixi	3	633	76

记录: 14 | 共有记录数: 3

图 10-3 建立图示数据库

现在正将前面所学的 SQL 语法用于实践。此时应注意一个重要的方法,那就是 Statement 对象的 execute 方法。

到这里是整个流程的第一个阶段。下面将完整的代码列出如下:

```
<%@page import="java.sql.*" contentType="text/html; charset=gb2312"%>
<%
    //register driver
    String driver="sun.jdbc.odbc.JdbcOdbcDriver";
    try{
        Class.forName(driver).newInstance();
    }
    catch(Exception e){
        out.print("Failed to load JDBC—ODBC Driver.");
        return;
    }
    //create Connection
    String ConnectionURL="jdbc:odbc:Mybbs";
    Connection con=null;
    Statement stmt=null;
    try{
        con=DriverManager.getConnection(ConnectionURL);
```

```
//create Statement
stmt = con.createStatement();
}
catch(Exception e){
    out.print("Problems connecting to " + ConnectionURL);
    return;
}
//create table
String createtable = "CREATE TABLE Article("
    + "ID                long NOT NULL UNIQUE,"
    + "Article            String(50),"
    + "UserID              long,"
    + "Username            String(30),"
    + "HypotaxissubID      int,"
    + "ReplyID             int,"
    + "Clickcount          int);"
try{
    stmt.execute(createtable);
}
catch(Exception e){
    out.print("Problems with SQL:" + e.getMessage());
    return;
}
out.print("You have successfully created a table!");
//insert data
String[] insertvalue = {
    "INSERT INTO Article VALUES(1,' Welcome. . .', 9999,' Admin', 1,
    null,80);",
    "INSERT INTO Article VALUES(633,' My first article',3689,'onlyyou',
    3,null,90);",
    "INSERT INTO Article VALUES(756,' My first too. . .', 9973,' xixi',
    3,633,76);"
};
try{
    for(int i=0;i<insertvalue.length;i++){
        stmt.execute(insertvalue[i]);
    }
}
catch(Exception e){
    out.print("Problems with SQL:" + e.getMessage());
}
```

```
        return;  
    }  
    out.print("Congratulation! you have got a table with data.");  
    stmt.close();  
    con.close();  
%>
```

### 10.3.6 获取结果集合

现在可以从 Mybbs 数据库中获取信息了。通过 JDBC 的 Statement.executeQuery 方法使用 SQL SELECT 语句,它将返回一个包含结果数据行的 ResultSet 对象。

现在让我们来评选 BBS 系统的最佳作者。执行这个查询的 SQL 语句很简单,曾在 SQL 语法讲述中实现了这个查询。以 SQL 的观点,获得最大值的一个方法是以 Clickcount 列的降序排列,匹配点击次数最大者的 ID 和名字将被选出,作为一个元组(或行数据)存储在 ResultSet 对象中。使用的 SQL 语句是:

```
SELECT UserID, Username, Clickcount  
FROM Article  
ORDER BY Clickcount DESC;
```

在 JSP 中,执行这条语句:

```
<%  
    String sqlstring="SELECT UserID, Username, Clickcount  
        FROM Article ORDER BY Clickcount DESC;";  
    ResultSet rs = stmt.executeQuery(sqlstring);  
%>
```

### 10.3.7 数据处理

数据处理的关键在于怎样定位行、列,从而从 ResultSet 对象中读出数据。借助游标,我们能够达到上述目的。定位行,最平常、最通用的方法是使用 ResultSet 对象的 next() 方法。

该方法使游标移动到当前行的下一行。这里只使用它移动到第一行,因为最佳作者就在第一行。

定位列最常用的方法是 getXXX(columnIndex)。其中 XXX 代表类型名, columnIndex 代表列序号。例如本例中使用 getLong(1) 获得 UserID 的值,使用 getString(2) 获得 Username 的值,使用 getInt(3) 获得 Clickcount 值。

现在,一切尽在掌握之中,可将数据进行任何处理。完成打印输出的代码如下:

```
<%  
    rs.next();  
    out.println(rs.getLong(1));
```

```

        out.println(rs.getString(2));
        out.println(rs.getInt(3));
    %>

```

### 10.3.8 获得元数据

事实上,我们还可以使用另一种方法定位列。尤其是需要获得元组每一列上的数据,而又不知道一共有多少列时,这种方法更是异常高效。不仅如此,还可以获得属性列的数据类型。方法是在 `ResultSet` 对象上进一步获得元数据对象,即 `ResultSetMetaData`。

紧接着上面的例子,假设我们不仅仅想知道最佳作者,例如评选十佳,然后把他们的完整信息打印出来。本书这个例子为评选前三名最佳作者,代码如下:

```

<%
    String sqlstring="SELECT * FROM Article ORDER BY Clickcount DESC;";
    ResultSet rs=stmt.executeQuery(sqlstring);
    ResultSetMetaData rsmd=rs.getMetaData();
    int columncount=rsmd.getColumnCount();
    for(int i=1;i<=columncount;i++){
        out.print(rsmd.getColumnLabel(i));
        out.print(rsmd.getColumnTypeName(i));
    }
    out.println("<br>");
    while(rs.next()){
        for(int j=1;j<=columncount;j++){
            out.print(rs.getString(j));
        }
        out.print("<br>");
    }
%>

```

这里,应当注意这样五个重点:一是使用 `while(rs.next())` 遍历 `ResultSet` 对象;二是使用 `for(...)` 循环定位元组的每一列;三是使用 `ResultSetMetaData` 对象的 `getColumnCount` 方法获得 `for(...)` 循环的次数;四是使用 `ResultSetMetaData` 对象的 `getColumnLabel` 方法获得属性列的名称;五是使用 `ResultSetMetaData` 对象的 `getColumnTypeName` 方法获得属性列数据类型的名称。

这是整个流程的第二个阶段。至此,一般的数据库查询,数据处理都没有问题了。完整的代码列出如下:

```

<%@page import="java.sql.*" contentType="text/html; charset = gb2312"%>
<%
    String driver="sun.jdbc.odbc.JdbcOdbcDriver";
    try{

```

```
//register driver
Class.forName(driver).newInstance();
}
catch(Exception e){
    out.print("Failed to load JDBC—ODBC Driver.");
    return;
}
String ConnectionURL="jdbc:odbc:Mybbs";
Connection con=null;
Statement stmt=null;
try{
    //create Connection
    con=DriverManager.getConnection(ConnectionURL);
    //create Statement
    stmt=con.createStatement();
}
catch(Exception e){
    out.print("Problems connecting to "+ConnectionURL);
    return;
}
String sqlstring="SELECT UserID,Username,Clickcount
FROM Article ORDER BY Clickcount DESC;";
ResultSet rs=null;
try{
    //get result set
    rs=stmt.executeQuery(sqlstring);
}
catch(Exception e){
    out.print("error:" + e.getMessage());
}
//get first data that is best author's information
rs.next();
out.print(rs.getLong(1));
out.print(rs.getString(2));
out.print(rs.getInt(3));
out.print("<br>");
//get information of authors whose article are best three article
sqlstring="SELECT * FROM Article ORDER BY Clickcount DESC;";
ResultSetMetaData rsmd=null;
```

```

try|
    rs = stmt.executeQuery(sqlstring);
    rsmd = rs.getMetaData();
|
catch(Exception e){
    out.print("error message:" + e.getMessage());
}
int columncount = rsmd.getColumnCount();
for(int i = 1; i <= columncount; i++){
    //display label of column
    out.print(rsmd.getColumnLabel(i));
    //display type name of column
    out.print(rsmd.getColumnTypeName(i));
}
out.println("<br>");
while(rs.next()){
    //display data of each column
    for(int j = 1; j <= columncount; j++){
        out.print(rs.getString(j));
    }
    out.print("<br>");
}
rs.close();
stmt.close();
con.close();

```

% >

### 10.3.9 将处理结果写回数据库

#### 1. 更新

例子中没有需要写回数据库的数据。但是假设需要更新某篇文章的点击次数,例如 Admin 发表的 ID 为 1 的文章。更新使用 Statement 对象,代码如下:

```

< %
String updatestring = "UPDATE Article SET Clickcount = Clickcount + 1
WHERE ID = 1;";
stmt.executeUpdate(updatestring);

```

% >

值得留意的仅有 Statement 对象的 executeUpdate 方法,它用来执行更新语句、插入语



句和删除语句。

## 2. 插入

同样,假设需要将一篇新文章的完整信息插入 Article 表。例如需要插入(767,'Ask...',517,'haha',1,null,23)。代码如下:

```
<%  
    String insertstring="INSERT INTO Article  
    VALUES (767,'Ask...',517,'haha',1,null,23);"  
    stmt.executeUpdate(insertstring);  
%>
```

## 3. 删除数据

假设将刚才插入的文章删除。代码如下:

```
<%  
    String delstring="DELETE FROM Article WHERE ID = 767";  
    stmt.executeUpdate(delstring);  
%>
```

这是整个流程的第三个阶段。至此,你已经可以凭这“一招半式”闯江湖了。事实上,将这“一招半式”烂熟于胸,江湖再大的风浪你也能闯过,因为这是 JDBC 最基本的东西,放之四海而皆准。现将完整的代码列出如下:

```
<%@page import="java.sql.*" contentType="text/html; charset = gb2312"%>  
<%  
    String driver="sun.jdbc.odbc.JdbcOdbcDriver";  
    try{  
        Class.forName(driver).newInstance();  
    }  
    catch(Exception e){  
        out.print("Failed to load JDBC—ODBC Driver.");  
        return;  
    }  
    String ConnectionURL="jdbc:odbc:Myhhs";  
    Connection con = null;  
    Statement stmt = null;  
    try{  
        con = DriverManager.getConnection(ConnectionURL);  
        stmt = con.createStatement();  
    }  
    catch(Exception e){  
        out.print("Problems connecting to " + ConnectionURL);
```

```

        return;
    }
    try{
        String updatestring = "UPDATE Article SET Clickcount = Clickcount + 1
        WHERE ID = 1;";
        stmt.executeUpdate(updatestring);
        out.print("Update successfully! <br>");
        String insertstring = "INSERT INTO Article
        VALUES(767, 'Ask...', 517, 'haha', 1, null, 23);";
        stmt.executeUpdate(insertstring);
        out.print("Insert successfully! <br>");
        String delstring = "DELETE FROM Article WHERE ID = 767;";
        stmt.executeUpdate(delstring);
        out.print("Delete successfully!");
    }
    catch(Exception e){
        out.print("Execute error:" + e.getMessage());
    }
}
%>

```

细心的读者一定会发现给出的完整程序代码中还有两个内容在前面并没有提到,一个是对象的关闭,另一个是异常和警告处理。

### 10.3.10 关闭对象

关闭对象很简单。需要说明的是这里的“对象”是一个复数,指代了三个实际的对象,它们是 ResultSet 对象、Statement 对象和 Connection 对象。关闭对象一般形式如下:

```

<%
    Connection con = null;
    Statement stmt = null;
    ResultSet rs = null;
    .....
    rs.close();
    stmt.close();
    con.close();
%>

```



**注意:** (1) 它们总是按这样的顺序被创建、被关闭,这由系统保证。

(2) 不一定要显式关闭它们,在程序结束时,它们会被系统自动关闭。

(3) 或者显示关闭某个对象,那么在该对象上建立的其它对象都将被关闭。

其后,使用该对象和在其上建立的对象都是非法的。

例如,显示关闭 Statement 对象 stmt,那么建立在其上的 ResultSet 对象 rs 也将被关闭。如果没有建立新的 Statement 对象并赋值给 stmt,那么使用 stmt 是非法的,使用 rs 也是非法的。



**注意:** Statement 对象执行新的查询语句(execute 或 executeXXX)将自动关闭前一个语句对象和建立在其上的 ResultSet 对象。因此,如果需要交替执行查询,那么需要至少两个语句对象。

### 10.3.11 处理异常和警告

程序难免出错,Java 提供了一个非常有用的捕获异常的机制,而且简单易用。JDBC 当然继承了这种机制,提供了三个专用的异常类:SQLException、SQLWarning 和 BatchUpdateException。本书只讲述前两者。

#### 1. 异常处理

SQLException 是所有 JDBC 异常的基本类。这里我们给出一个使用它的标准模板:

```
<%
    try{
        .....
    }
    catch(SQLException ex){
        out.println("SQLException caught");
        out.println("<br>");
        while(ex != null){
            out.println("Message:    " + ex.getMessage());
            out.println("<br>");
            out.println("SQLState:   " + ex.getSQLState());
            out.println("<br>");
            out.println("ErrorCode:  " + ex.getErrorCode());
            out.println("<br>");
            ex = ex.getNextException();
        }
    }
%>
```

#### 2. 警告处理

SQLWarning 类与 SQLException 是相似的,不同之处在于它是一个非关键的错误并且不会被抛出。必须主动从 Connection, ResultSet 和 Statement 对象中获取。我们给出 ResultSet 对象中获取警告的标准模板,其它两个与之相似。

```
<%
```

```

        ResultSet rs = null;
        .....
        SQLWarning warn = rs.getWarnings();
        if(warn != null){
            out.println("----- Warning -----");
            out.println("<br>");
            while(warn != null){
                out.println("Message:" + warn.getMessage());
                out.println("<hr>");
                out.println("SQLState:" + warn.getSQLState());
                out.println("<br>");
                out.println("Vendor error code:");
                out.println(warn.getErrorCode());
                out.println("<br>");
                warn = warn.getNextWarning();
            }
        }
    }
%>

```

这是整个流程的第四个阶段,也是最后一个阶段。现在,可以写出完整的、规范的、优美的数据库应用程序了。不要再犹豫,赶快动手写出自己第一个优美的数据库应用程序。

事实上,10.3 节的主要目的是让大家尽快上手,所以并没有纠缠于类、方法的细节,而且使用的方法都是最基本的。接下来的几节将详细讨论上述过程,内容包括上文中出现过的重要的类及其方法,还将介绍未在上文出现但非常重要的几个类及其方法,如 PreparedStatement, PooledConnection。

## 10.4 建立数据源

使用 Win98 平台下的 ODBC 数据源并且使用 Microsoft Access 驱动程序,当然性能和支持属性的完整性是比较差的。这部分内容非常简单,熟悉的读者完全可以跳过这一节。这里只给出系统数据源的建立过程,用户数据源建立和它是相似的。二者的差异主要是可见性范围不同,系统数据源对当前计算机上的所有用户可见。

下面给出较为简练的步骤:

- (1) 打开“控制面板 > ODBC 数据源(32 位)”。
- (2) 选择“系统 DSN”标签。如图 10-4 所示。
- (3) 单击“添加”按钮,打开“创建新数据源”对话框。如图 10-5 所示。

默认为 Microsoft Access Driver,否则选中该驱动程序。然后单击“完成”按钮,打开“ODBC Microsoft Access 安装”对话框。如图 10-6 所示。

在“数据源名”输入框中输入你希望的数据库。本书为“Mybbs”。建议使用这个名字。



图 10-4 ODBC 数据源管理器的系统 DSN 窗口



图 10-5 创建新数据源对话框



图 10-6 ODBC Microsoft Access 安装

字,这样无需改动范例便能执行,并看到效果图。另外,还可以在描述中输入数据库相关信息。

如果已经用 Microsoft Access 应用程序创建了一个数据库,那么请选择“选取”按钮。否则,请选择“创建”按钮。前者打开“选定数据库”对话框,后者打开“新数据库”对话框,本书选择后者,如图 10-7 所示。

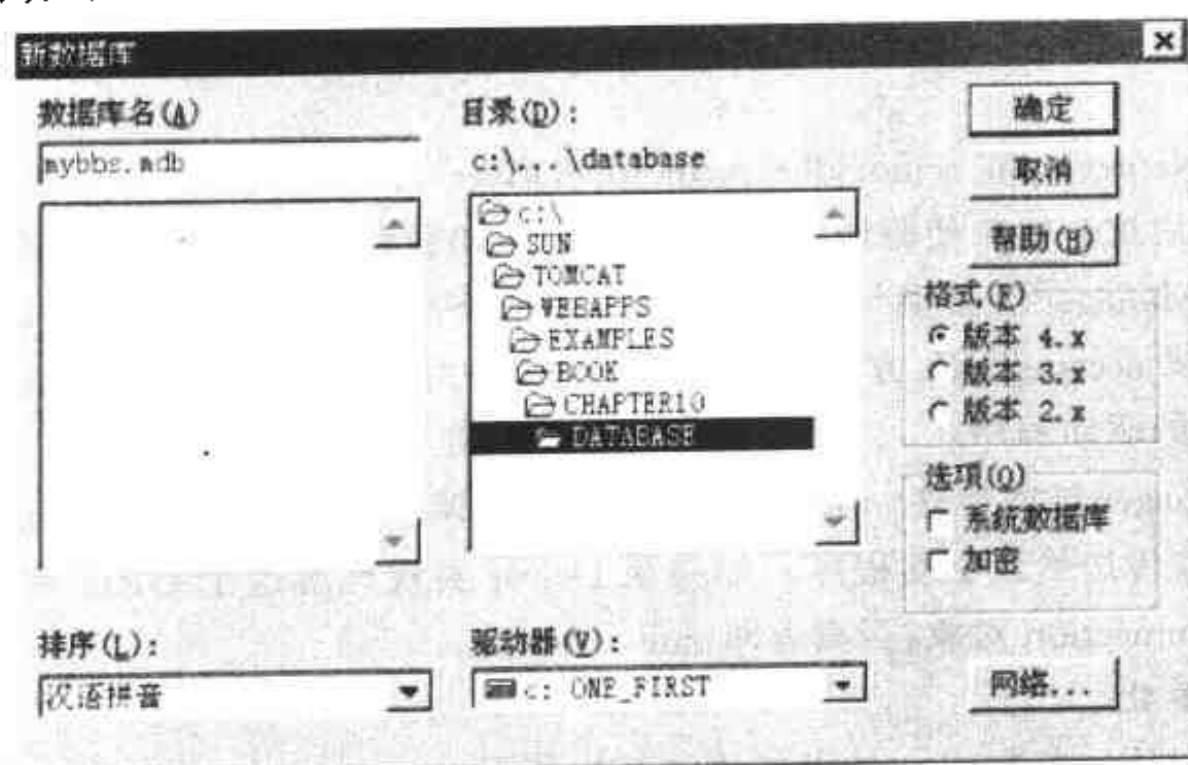


图 10-7 新数据库

在“目录”列表框中选择欲建立数据库所在目录,本书为“C:\Sun\Tomcat\webapps\examples\book\chapter10\database”。在“数据库名”输入框中输入你希望的数据库名,文中为“mybbs.mdb”。最重要的一点,一定要在“排序”下拉菜单中选择一项,例如文中为“汉语拼音”。否则,下一步发生错误。

单击“确定”按钮,回到图 10-6 所示对话框。“Mybbs”数据源配置成功。当然,还可以选择“高级”按钮,打开“设置高级选项”对话框,设置“登录名称”、“密码”等。至此,已经建立好一个数据源,可以通过外部编程接口访问它了。

## 10.5 Driver

### 1. 概述

#### 1) Driver 与 DriverManager 关系

JDBC 驱动程序必须实现 Driver 界面并且这个实现必须包含一个静态的初始化,当驱动程序被加载时,这个初始化就会被调用。初始化使用 DriverManager 注册一个自身实例。

例:

```
public class AcmeJdbcDriver implements java.sql.Driver {  
    static {  
        java.sql.DriverManager.registerDriver(new AcmeJdbcDriver());  
    }  
}
```

```

    }
    ...
}

```

上例静态初始化一个实现 `java.sql.Driver` 界面的驱动程序。

当应用程序加载一个 `Driver` 实现,静态初始化将自动注册这个驱动程序的一个实例。

例:

```
Class.forName("com.acme.jdbc.AcmeJdbcDriver");
```

为确保使用这个机制使驱动程序能被加载,驱动程序必须提供一个构造器。

当 `DriverManager` 类希望与某个注册的驱动程序交互的时候,它调用 `Driver` 的方法。`Driver` 界面提供 `acceptsURL` 方法,`DriverManager` 使用这个方法判断对给出的 URL 应使用哪个注册过的驱动程序。

`DriverManager` 试图建立一个 `Connection` 时,它调用哪个驱动程序的 `connect` 方法并且将 URL 参数传送给该驱动程序。如果该 `Driver` 实现理解这个 URL,那么这个 `Driver` 将返回一个 `Connection` 对象,否则返回 `null`。

## 2) URL 参数

界面中的 URL 参数,Sun 推荐的语法结构如下:

```
jdbc:< subprotocol >:< subname >
```

这与 World Wide Web 使用的命名系统 URL 机制是相同的。`subprotocol` 命名一种特定类型的数据库连接机制,它可能由一个或多个驱动程序支持,并且它决定 `subname` 的语法形式和内容。

如果指定网络地址为 `subname` 的一部分,Sun 推荐使用下述标准的 URL 命名规则:

```
//hostname:port / subsubname
```

`subsubname` 可有任意的本地语法。

例如最常用的,使用 JDBC-ODBC 桥访问一个数据库,可使用如下格式的 URL:

```
jdbc:odbc:fred
```

这个例子中,子协议 `subprotocol` 是“ODBC”,`subname` 定位一个名为“fred”的 ODBC 数据源。JDBC-ODBC 驱动程序可检查到 URL 有子协议“odbc”,然后以 ODBC SQL-Connect 方式使用 `subname`。

如果希望使用一般的数据库连接协议“dbnet”与数据库监听者交互,可以使用这样的 URL:

```
jdbc:dbnet:// wombat:356 / fred
```

这个例子中的 URL 表示使用“dbnet”协议连接一个名为 `wombat` 的主机,该主机监听端口为 356。`Subsubname`“fred”定位最终的数据库。

如果想使用网络服务器名字间接提供数据库的名字,Sun 推荐使用命名服务器的名字作为子协议。例如:

```
jdbc:dcenaming:accounts - payable
```

这个例子中,URL 表示的是使用 DCE 命名的服务器将名为“account - payable”的数据库解析为多个特定的名字,然后使用这些名字连接到实际的数据库。

“odbc”子协议保留为表示 ODBC 类型数据源的名字。这个子协议允许在数据源名字之后使用任意的属性值。

完整的 odbc 子协议 URL 语法如下：

`jdbc:odbc:<data-source-name>[;<attribute-name>=<attribute-value>]*`

例：

`jdbc:odbc:wombat;cacheSize=20;ExtensionCase=LOWER`

## 2. 界面说明

### 1) 界面声明

```
java.sql Interface Driver
```

```
public interface Driver
```

任何驱动程序类都必须实现这个界面。Java SQL 框架支持多种类型数据库驱动程序。

每个驱动程序必须提供一个实现 Driver 界面的类。

DriverManager 总是设法加载它能找到的尽可能多的驱动程序，然后，对给出的任何连接请求，它都将轮询每个驱动程序并设法与 URL 表示的目标相连。

建议：每个 Driver 类应当是小型的和独立的，以便 Driver 类被加载而不需要引入大量的支持代码。

当一个 Driver 类被加载时，它将建立自身的一个实例并且将自身注册到 DriverManager 中。这意味着用户能够通过调用下式加载和注册一个驱动程序。

(1) `Class.forName("foo.bah.Driver")`

还有另一种方法加载和注册一个驱动程序，如下所示。

(2) `new foo.Driver()`

这种方法与 `Class.forName("foo.bah.Driver")` 是完全等价的。

### 2) 参照

DriverManager, Connection。

### 3) 方法一览

(1) `boolean acceptsURL(java.lang.String url)`

如果驱动程序认为它能建立给出 URL 的一个连接，则返回“true”。

(2) `Connection connect(java.lang.String url, java.util.Properties info)`

建立给出 URL 的一个数据库连接。

(3) `int getMajorVersion()`

获得驱动程序的主版本号。

(4) `int getMinorVersion()`

获得驱动程序的次版本号。

(5) `DriverPropertyInfo[] getPropertyInfo(java.lang.String url, java.util.Properties info)`

获得这个驱动程序可能的有关属性信息。

(6) `boolean jdbcCompliant()`

报告这个驱动程序是否是真正兼容 JDBC 技术的驱动程序。



## 10.6 DriverManager 和 DataSource

### 10.6.1 DriverManager 类

#### 1. 概述

DriverManager 类与 Driver 界面协同工作,管理对客户有效的一组 JDBC 驱动程序,当客户端需要一个连接并且提供了一个 URL。DriverManager 的责任就是寻找能识别这个 URL,并使用这个 URL 与相应的数据源建立连接的驱动程序。

##### 1) DriverManager 的关键方法

RegisterDriver: 添加一个驱动程序到一组有效的驱动程序集中,并且驱动程序被加载时,registerDriver 被隐含调用。每个驱动程序提供的静态初始化典型调用 RegisterDriver 方法。

getConnection: JDBC 客户端调用建立一个连接的方法。这种调用包含一个 JDBC URL,URL 被 DriverManager 传递给 DriverManager 的列表中每个驱动程序,直至它找到一个驱动程序,这个驱动程序的 Driver.connect 方法能识别这个 URL,该 Driver 返回一个 Connection 对象给 DriverManager,DriverManager 依次将 Connection 传递给应用程序。

例:

下面举例说明一个 JDBC 客户端怎样使用 DriverManager 获得一个 Connection。

```
// Load the driver. This creates an instance of the driver
// and calls the registerDriver method to make acme.db.Driver
// available to clients.
Class.forName("acme.db.Driver");
// Set up arguments for the call to the getConnection method.
// The sub-protocol "odbc" in the driver URL indicates the
// use of the JDBC-ODBC bridge.
String url = "jdbc:odbc:DSN";
String user = "someUser";
String passwd = "somePwd";
// Get a Connection from the first driver in the DriverManager
// list that recognizes the URL "jdbc:odbc:DSN",
Connection con = DriverManager.getConnection(url, user, passwd);
```

##### 2) 另外两种 getConnection 方法

(1) getConnection(String url) 不需要用户名和密码建立数据源的连接。

(2) getConnection(String url, java.util.Properties prop), 允许客户端使用一组描述

用户名、密码和任何必要的额外信息建立一个连接。DriverPropertyInfo 类提供 JDBC 驱动程序能理解的属性的信息。

## 2. 类库说明

### 1) 类层次

java.lang.Object

|


+ ——— java.sql.DriverManager

java.sql Class DriverManager

### 2) 类声明

```
public class DriverManager extends java.lang.Object
```

DriverManager 类提供管理 JDBC 驱动程序的基本服务。

 注意:JDBC 2.0 API 中的新界面 DataSource,提供了另一种连接数据源的方法,DataSource 对象是连接数据源的首选手段。

作为初始化的一部分,DriverManager 类试图加载那些引用“jdbc.drivers”系统属性的驱动程序类。这使得用户可以自定义其应用程序中使用的 JDBC 驱动程序。例如:在 ~/.hotjava/properties 文档中可以指定:

```
jdbc.drivers=foo.bah.Driver:wombat.sql.Driver:bad.taste.ourDriver
```

一个应用程序也可在任意时刻显式加载 JDBC 驱动程序。例如:my.sql.Driver 用如下语句加载:Class.forName(“my.sql.Driver”);

当方法 getConnection 被调用时,DriverManager 试图在初始化加载的驱动程序和当前 applet,或应用程序使用相同的类加载器加载的驱动程序中寻得一个合适的驱动程序。

### 3) 方法一览

(1) static void deregisterDriver(Driver driver)

从 DriverManager 列表中注销一个驱动程序。

(2) static Connection getConnection(java.lang.String url)

试图建立一个对某数据库的连接,其中某数据库由数据库 URL 指定。

(3) static Connection getConnection(java.lang.String url, java.util.Properties info)

试图建立一个对某数据库的连接,其中某数据库由数据库 URL 指定。

(4) static Connection getConnection(java.lang.String url, java.lang.String user, java.lang.String password)

试图建立一个对某数据库的连接,其中某数据库由数据库 URL 指定。

(5) static Driver getDriver(java.lang.String url)

试图获得理解参数指定的 URL 值的驱动程序。

(6) static java.util.Enumeration getDrivers()

检索获得调用者能访问的当前已加载 JDBC 驱动程序的枚举。

(7) static int getLoginTimeout()

获得驱动程序登录到某个数据库的最大等待时间(秒),否则便登录失败。

(8) static java.io.PrintStream getLogStream()

不赞成使用。

(9) static java.io.PrintWriter getLogWriter()

获得日志写出流。

(10) static void println(java.lang.String message)

将一条消息写入当前 JDBC 日志流。

(11) static void registerDriver(Driver driver)

注册给出驱动程序到 DriverManager 列表中。

(12) static void setLoginTimeout(int seconds)

设置驱动程序试图连接一个数据库可能等待的最大秒数。

(13) static void setLogStream(java.io.PrintStream out)

不赞成使用。

(14) static void setLogWriter(java.io.PrintWriter out)

建立 DriverManager 和所有驱动程序用来写日志的 PrintWriter 对象。

4) 继承自 class java.lang.Object 的方法

equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## 10.6.2 DataSource

### 1. 概述

DataSource 界面是建立数据源连接的首选方法。实现 DataSource 界面的 JDBC 驱动程序返回一个 Connection 对象,与 DriverManager 使用 Driver 界面返回没有什么不同。DataSource 对象提高了应用程序的可移植性。因为它使得应用程序使用数据源的逻辑名代替必须针对特定数据源提供特定信息成为可能。使用 JNDI(Java Naming and Directory Interface)名字服务可将逻辑名映射到一个 DataSource 对象上。DataSource 对象代表了一个物理数据源并且提供对那个数据源的连接。如果数据源和与数据源相关的信息发生改变,仅仅简单改变一个 DataSource 对象的属性就可以映射出这种改变,而不必修改程序代码。

#### 1) DataSource 的服务

DataSource 界面的实现透明的提供如下服务:

(1) 通过 Connection pooling 提高性能与可伸缩性。

(2) 通过 XADataSource 界面支持分布式事务处理。

下面将进行三点讨论:一是 DataSource 基本属性。二是怎样用 JNDI API 命名逻辑名以提高应用程序可移植性及使其更加易于维护。三是怎样获得一个连接。

#### 2) DataSource 属性

JDBC API 定义了一个属性集以识别和描述一个 DataSource 实现,对一个特定的实现需要的实际属性集依赖于 DataSource 对象的类型,也就是说,是一个基本的 DataSource 对象,还是一个 ConnectionPoolDataSource 对象,或是一个 XADataSource 对象。对所有的 DataSource 实现唯一必要的属性是 description。表 10-6 描述了标准 DataSource 的属性。

表 10-6 DataSource 的属性

属 性 名	类 型	描 述
DatabaseName	String	服务端特定数据库的名字
DataSourceName	String	一个数据源的名字,当池缓冲连接建立后,用来命名一个底层 XADataSource 对象或 ConnectionPoolDataSource 对象
Description	String	描述一个数据源
NetworkProtocol	String	与服务端通信的网络协议
Password	String	数据库的密码
PortNumber	int	服务端监听请求的端口号
RoleName	String	初始 SQL 角色名
ServerName	String	数据库服务端名字
User	String	用户帐户名

DataSource 属性遵循 JavaBean1.01 规范中 JavaBean 组件属性的配置约定。为使用特殊属性,DataSource 实现可以扩展这个集,如果添加(增加)新的属性,它们的名字不能与标准属性名发生冲突。

DataSource 实现必须为它们支持的每个属性提供“setter”和“getter”方法。这些属性在 DataSource 对象展开时被初始化。

例:

设置和获取一个 DataSource 属性

```
VendorDataSource vds = new VendorDataSource();
vds.setServerName("my _ database _ server");
String name = vds.getServerName();
```

DataSource 属性有意不让 JDBC 客户端直接访问。在实现类上定义访问方法,而不是在应用程序使用的公共 DataSource 界面上定义访问方法,客户端操纵的对象仅仅是实现 DataSource 界面的一个壳,属性的“getter”和“setter”方法的细节不会暴露给客户端。

### 3) JNDI API 和应用程序可移植性

JNDI(Java Naming and Directory Interface)API 为应用程序通过网络访问远程服务提供了一种统一的方法。这里描述怎样用它来注册和访问一个 JDBC DataSource 对象。

有了 JNDI API,应用程序可以通过指定 DataSource 的逻辑名访问它。名字服务器使用 JNDI API 映射这个逻辑名到对应的数据源。这种方案大大的增强了可移植性,因为任意一个 DataSource 属性,例如 portNumber 或 ServerName 都可被改变而不影响 JDBC 客户端代码。事实上,应用程序能以完全透明的方式重新指向一个不同的底层数据源。这在三层环境中特别有用,因为应用程序服务隐藏了访问不同数据源的细节。

例:

使用基于 JNDI 的名字服务注册一个 DataSource 对象。

```
// Create a VendorDataSource object and set some properties
VendorDataSource vds = new VendorDataSource();
vds.setServerName("my _ database _ server");
```

```
vds.setDatabaseName("my _ database");  
vds.setDescription("data source for inventory and personnel");  
// Use the JNDI API to register the new VendorDataSource object.  
// Reference the root JNDI naming context and then bind the  
// logical name "jdbc/AcmeDB" to the new VendorDataSource object.  
Context ctx = new InitialContext();  
ctx.bind("jdbc/AcmeDB", vds);
```

#### 4) 使用 DataSource 对象获得 Connection 对象

一旦 Data Source 对象被基于 JNDI 的名字服务注册,应用程序就能使用它获得对它表示的物理数据源的连接。

例:

```
// Get the initial JNDI naming context  
Context ctx = new InitialContext();  
// Get the DataSource object associated with the logical name  
// "jdbc/AcmeDB" and use it to obtain a database Connection  
DataSource ds = (DataSource)ctx.lookup("jdbc/AcmeDB");  
Connection con = ds.getConnection("user", "pwd");
```

使用 DataSource 对象获得一个 Connection 对象。绑定名为"jdbc/AcmeDB"的 DataSource 实现可在不影响应用程序代码的情况下被修改或替换。

## 2. 界面说明

### 1) 界面声明

```
javax.sql Interface DataSource  
public interface DataSource
```

DataSource 对象是 Connection 对象的一个(实例)Factory 实现。实现 DataSource 界面的对象被 JNDI 服务提供者注册。由 DataSource API 访问的一个 JDBC 驱动程序不会自动使用 DriverManager 注册自身。

### 2) 方法一览

(1) java.sql.Connection getConnection()

试图建立一个数据库连接。

(2) java.sql.Connection getConnection(java.lang.String username, java.lang.String password)

试图建立一个数据库连接。

(3) int getLoginTimeout()

获得当前数据源登录到某个数据库的最大等待时间(秒),否则便登录失败。

(4) java.io.PrintWriter getLogWriter()

当前数据源获得日志写出流。

(5) void setLoginTimeout(int seconds)

设置当前数据源试图连接一个数据库可能等待的最大秒数。

(6) void setLogWriter(java.io.PrintWriter out)

当前数据源建立日志写出流。

## 10.7 Connection 和 PooledConnection

### 10.7.1 Connection

#### 1. 概述

Connection 对象表示基于通过 JDBC 技术驱动程序与连接的数据源的连接,数据源可以是一个 DBMS,一个传统文件系统,或是其它与一个 JDBC 驱动程序相联系的数据源。一个独立的应用程序使用 JDBC API 可以维持多个连接,这些连接访问多个数据源或都访问同一个数据源。

从 JDBC 驱动程序的角度看,一个 Connection 对象代表一个客户端会话,它有与之相关联的状态信息,例如用户帐号、一组 SQL 语句和在当前会话中使用的结果集合。

#### 2. 界面说明


##### 1) 界面声明

```
java.sql Interface Connection
```

```
public interface Connection
```

特定数据库的一个 Connection 在其上下文中,可以执行 SQL 语句并获得返回结果。

一个 Connection 对象指向的数据库应该有能力提供描述自身的表,所支持的 SQL 语法,所存储的过程(stored procedures),当前连接的容量等等信息。这些信息可用 getMetaData 方法获得。

 注意:默认情况下,每执行完一条语句,Connection 自动的提交变化(更新数据库)。

如果自动提交(更新数据库)被禁止,那么必须显式调用方法 commit 以更新数据库,否则,数据库的变化是会被保存的。

##### 2) 参照

DriverManager.getConnection(java.lang.String, java.util.Properties), Statement, ResultSet,

##### 3) 域一览

(1) static int TRANSACTION\_NONE

指明不支持事务处理(Transaction)。

(2) static int TRANSACTION\_READ\_COMMITTED

禁止不明读,但允许影像(phantom)读和非可重复读。

(3) static int TRANSACTION\_READ\_UNCOMMITTED

允许不明读,影像读,非可重复读。

(4) static int TRANSACTION \_ REPEATABLE \_ READ

禁止不明读、非可重复读,但允许影像读。

(5) static int TRANSACTION \_ SERIALIZABLE

禁止不明读,非可重复读,影像读。

#### 4) 方法一览

(1) void clearWarnings()

清除对当前 Connection 对象的所有警告。

(2) void close()

立即关闭当前 Connection 对象以释放其指向的数据库和 JDBC 资源,而不是等待它们自动释放。

(3) void commit()

使上一次永久性提交(commit)或回滚(rollback)后发生的所有改变生效并释放 Connection 当前拥有的任何数据库锁。

(4) Statement createStatement()

建立一个 Statement 对象以发送 SQL 语句给数据库执行。

(5) Statement createStatement(int resultSetType, int resultSetConcurrency)

建立一个 Statement 对象,当前 Statement 对象产生的 ResultSet 对象有参数指定的类型和并发性。

(6) Statement createStatement(int resultSetType, int resultSetConcurrency, int resultSetHoldability)

建立一个 Statement 对象,该 Statement 对象将产生的 ResultSet 对象有参数指定的类型、并发性 and 持有能力。

持有能力:数据库中的数据被传送到其它位置后,此数据仍然保持在数据库中的能力。

(7) boolean getAutoCommit()

获得当前的自动提交状态,是否自动提交被禁止。

(8) java.lang.String getCatalog()

返回 Connection 的当前目录名。目录名是全部数据集合索引的汇总,控制程序用它来查找含有特定数据集合所在的卷。

(9) int getHoldability()

获得当前 Connection 对象建立的 resultset 对象的持有能力(Holdability)。

(10) DatabaseMetaData getMetaData()

获得当前 Connection 指向数据库的元数据。

(11) int getTransactionIsolation()

获得当前 Connection 的当前事务处理隔离(isolation)等级。

隔离:自动数据处理系统,如数据库中的用户和资源的相关牵制关系,即用户和进程彼此分开,且和操作系统的保护控制也分开。

(12) java.util.Map getTypeMap()

获得与当前 Connection 相关的类型映射(type map)对象。

(13) `SQLWarning getWarnings()`

返回调用者报告的对当前 `Connection` 对象的第一个警告。

(14) `boolean isClosed()`

测试一个 `Connection` 是否被关闭。

(15) `boolean isReadOnly()`

测试当前 `Connection` 是否为只读(read-only)模式。

(16) `java.lang.String nativeSQL(java.lang.String sql)`

转换给出的 SQL 语句为系统本地 SQL 语法。

(17) `CallableStatement prepareCall(java.lang.String sql)`

建立一个 `CallableStatement` 对象,以调用数据库已存储的过程(stored procedures)。

(18) `CallableStatement prepareCall(java.lang.String sql, int resultSetType, int resultSetConcurrency)`

建立一个 `CallableStatement` 对象,当前 `CallableStatement` 对象产生的 `ResultSet` 对象有参数指定的类型和并发性。

(19) `CallableStatement prepareCall(java.lang.String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)`

建立一个 `CallableStatement` 对象,当前 `CallableStatement` 对象产生的 `ResultSet` 对象有参数指定的类型、并发性和持有能力。

(20) `PreparedStatement prepareStatement(java.lang.String sql)`

建立一个 `PreparedStatement` 对象,以发送带有参数的 SQL 语句给数据库。

(21) `PreparedStatement prepareStatement(java.lang.String sql, int flag)`

建立一个 `PreparedStatement` 对象,以发送带有参数的 SQL 语句给数据库。

(22) `PreparedStatement prepareStatement(java.lang.String sql, int[] columnIndexes)`

建立一个 `PreparedStatement` 对象,以发送带有参数的 SQL 语句给数据库。

(23) `PreparedStatement prepareStatement(java.lang.String sql, int resultSetType, int resultSetConcurrency)`

建立一个 `PreparedStatement` 对象,该 `PreparedStatement` 对象将产生的 `ResultSet` 对象有参数指定的类型、并发性。

(24) `PreparedStatement prepareStatement(java.lang.String sql, int resultSetType, int resultSetConcurrency, int resultSetHoldability)`

建立一个 `PreparedStatement` 对象,该 `PreparedStatement` 对象将产生的 `ResultSet` 对象有参数指定的类型、并发性和持有能力。

(25) `PreparedStatement prepareStatement(java.lang.String sql, java.lang.String[] columnNames)`

建立一个 `PreparedStatement` 对象,以发送带有参数的 SQL 语句给数据库。

(26) `void releaseSavepoint(Savepoint savepoint)`

从当前事务处理中删除一个存储点(savepoint)。

(27) `void rollback()`

撤消自上一次提交(commit)或回滚(rollback)以来发生的所有变化并释放当前 Con-



nection 当前拥有的数据库锁。

(28) void rollback(Savepoint savepoint)

撤消自设置存储点以来发生的所有变化。

(29) void setAutoCommit(boolean autoCommit)

设置当前 Connection 对象的自动提交模式。

(30) void setCatalog(java.lang.String catalog)

设置一个目录名,选择当前 Connection 对象工作在哪个数据库子域(子空间)。

(31) void setHoldability(int holdability)

改变当前 Connection 对象创建的 resultset 对象的持有能力。

(32) void setReadOnly(boolean readOnly)

设置当前 Connection 对象为只读模式,暗含使数据库优化的意思。

(33) Savepoint setSavepoint()

设置一个未命名存储点(unnamed savepoint)。

(34) Savepoint setSavepoint(java.lang.String name)

设置一个命名存储点(named savepoint)。

(35) void setTransactionIsolation(int level)

为给出值设置事务处理隔离性等级。

(36) void setTypeMap(java.util.Map map)

安装给出的类型映射作为当前连接的类型映射。

## 10.7.2 PooledConnection

### 1. 概述

#### 1) 连接池原理

对一个基本的 DataSource 实现,客户端的 Connection 对象与物理数据库连接之间是一一对应关系。当一个 Connection 对象关闭时,物理连接也断开。因此,每次客户端会话都有打开、初始化、关闭物理连接的开销。

连接池通过维持一个可重用的跨越客户端会话的物理数据库连接的缓存来解决这个问题。连接池大大的提高了连接的性能和可伸缩性,特别是在三层环境中多个客户端只能够共享少量的物理数据库连接的时候。连接池原理如图 10-8 所示。

管理连接池的法则是实现特定的应用程序服务并随之变化。应用程序服务提供一个 DataSource 界面的实现给它的客户端,使得对客户端来说,连接池是透明的。作为结果,在与以前一样使用 JNDI 和 DataSource API 的时候,客户端获得了更好的性能和可伸缩性。

接下来我们将介绍 ConnectionpoolDataSource 界面和 PooledConnection 界面,客户端使用它们在 DataSource 和 Connectin 界面下的操作。

尽管这里的许多讨论假定的是三层环境,但连接池也与两层环境相关,在两层环境中,JDBC 驱动程序实现 DataSource 和 ConnectionpoolDataSource 界面。这种实现使一个

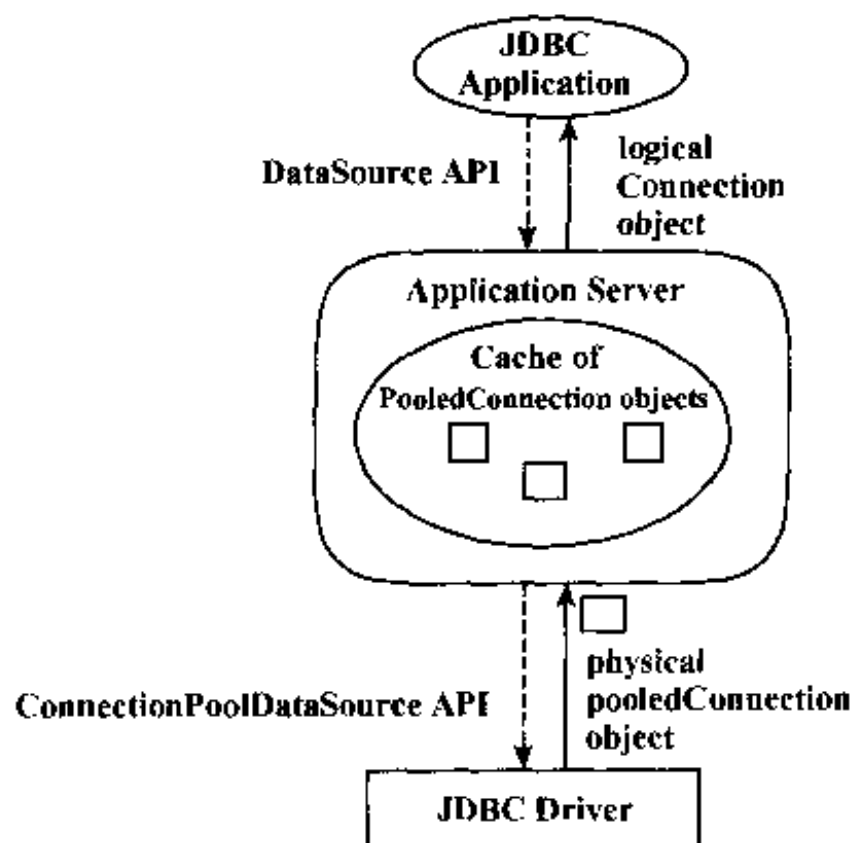


FIGURE 11-1 Connection pooling

图 10-8 连接池原理图

应用程序可以打开和关闭多个连接,并从连接池获益。

## 2) ConnectionPoolDataSource 和 PooledConnection

典型的,JDBC 驱动程序实现 ConnectionPoolDataSource 界面,然后应用程序服务端使用它以获得多个 pooledConnection 对象。ConnectionPoolDataSource 界面定义如下:

```
public interface ConnectionPoolDataSource {
    PooledConnection getPooledConnection() throws SQLException;
    PooledConnection getPooledConnection(String user,
    String password) throws SQLException;
    ...
}
```

PooledConnection 对象代表对数据源的物理连接。JDBC 驱动程序实例的 pooled Connection 封装了维持该连接的所有细节。应用程序服务端在它的 DataSource 界面的实现中缓存并且重用 PooledConnection 对象。当客户端调用 DataSource.getConnection 方法时,应用程序服务端使用物理的 PooledConnection 对象获得一个逻辑的 Connection 对象。

例:

```
public interface PooledConnection {
    Connection getConnection() throws SQLException;
    void close() throws SQLException;
    void addConnectionEventListener(
    ConnectionEventListener listener);
    void removeConnectionEventListener(
    ConnectionEventListener listener);
}
```

应用程序结束使用 Connection 时,它使用方法 Connection.close()方法关闭逻辑连接,但是不会关闭物理连接,当前物理连接被归还给缓存池。所以它是可重用的。

连接池对客户端是完全透明的,客户端获得和使用一个 PooledConnection 对象,与获得和使用一个非 PooledConnection 一样。

### 3) 连接事件

当应用程序调用方法 Connection.close 时,底层的物理连接的 Pooled Connection 对象可有效的被重用。JavaBeans 风格的事件用于通知连接池管理者(应用程序服务端)一个 PooledConnection 对象能被重新循环。为使通知事件作用在 PooledConnection 对象上,连接池管理者必须实现 ConnectionEventListener 界面,然后由 Pooled Connection 对象注册为一个监听者。ConnectionEventListener 界面定义了下面两个方法,那些符合这两种类型的事件将作用到 Pooled Connection 对象上。

ConnectionClosed:当与逻辑连接对象相联系的 PooledConnection 对象被关闭时,触发这个事件。也就是说,应用程序调用方法 Connection.close()。

ConnectionErrorOccured:当一个致命错误,如服务器崩溃,导致一个连接丢失,触发该事件。

### 4) 三层环境中的连接池

下述顺序步骤勾勒出,一个 JDBC 客户请求在实现了连接池的 DataSource 对象上建立一个连接所发生的事件:

调用 DataSource.getConnection()。

应用程序服务端提供一个 DataSource,它将检测自身的连接池看是否有一个合适的 PooledConnection 对象——一个有效的物理数据库连接。与使用其它特定实现标准一样,决定给定 PooledConnection 对象的适应性,可能包含匹配客户端用户识别信息或是应用程序类型。与管理连接池相联系的查找方法和其它方法对应用程序服务端来说是特定的。


如果连接池中没有一个合适的有效 PooledConnection 对象,应用程序服务端会调用 ConnectionpoolDataSource.getPooledConnection 方法获得一个新的物理连接,JDBC 驱动程序执行 ConnectionpoolDataSource 建立一个新的 PooledConnection 对象并将该对象返回给应用程序服务端。

不管 PooledConnection 是从连接池中获得的还是新创建的,应用程序服务端都在其内部作标记,指明该物理连接现在是可用的。

应用程序服务端调用 PooledConnection.getConnection()方法获得一个逻辑 Connection 对象,这个逻辑 Connection 对象是一个物理 PooledConnection 对象的句柄,并且这个句柄是在连接池起作用的时候由 DataSource.getConnection()方法返回。

应用程序服务端通过调用 PooledConnection().addConnectionEventListener()方法注册自身为一个连接事件监听者。如此这样之后,当物理连接可以重用时,应用程序服务端将被告知。

逻辑 Connection 对象返回给 JDBC 客户端,客户端使用 Connection API 与在基本的 DataSource 情况下是一样的。

 注意:底层的物理连接不能被重用,直到客户端调用方法 Connection.close。

连接池也能在两层环境中实现,只是两层环境中没有应用程序服务端。在这种情况下,JDBC 驱动程序提供两种 DataSource 的实现,对客户端和底层 ConnectionPoolDataSource 实现来说,DataSource 都是可见的。

抛开提高性能和可伸缩性不谈,一个 JDBC 应用程序在访问一个实现了连接池的 DataSource 对象和一个未实现连接池的 DataSource 对象之间不会看到任何差异。然而,在应用程序服务端和驱动程序级实现上还是存在一些重要的差异。

基本的 DataSource 实现,是由驱动程序提供商提供的,在一个基本的 DataSource 实现中,存在以下两种情况:

(1) DataSource.getConnection 方法建立一个新的 Connection 对象,该对象表示一个物理连接并且封装了建立和管理哪个连接的所有工作。

(2) Connection.close 方法关闭物理连接并且释放相关资源。

包含连接池的 DataSource 实现,大量地处理发生在幕后。对这样的一个实现,以下是真实的情况:

(1)DataSource 实现包含一个实现特定的连接池模块,模块管理 PooledConnection 对象的缓存空间。DataSource 对象被应用程序服务端实现,应用程序服务端作为驱动程序的 ConnectionPoolDataSource 和 PooledConnection 界面的实现的上层。

(2)DataSource.getConnection()方法调用 PooledConnection.getConnection()方法获得一个对底层物理连接的逻辑句柄。建立一个新的物理连接的开销仅仅发生在连接池中不存在有效的连接。当需要一个新的物理连接时,连接池管理器将调用 ConnectionPoolDataSource 对象的 getPooledConnection()方法建立一个,然后这个物理连接的管理工作被委托给 PooledConnection 对象。

(3)Connection.close()方法关闭逻辑句柄,但是当前物理连接仍然维持。连接池管理器被告知底层 PooledConnection 对象现在可以重用,如果应用程序试图重用逻辑句柄,Connection 实现将抛出一个 SQLException。

(4)一个物理的 PooledConnection 对象在它的生命周期中可以产生多个逻辑 Connection 对象,对一个给定的 PooledConnection 对象,只有最近产生的逻辑 Connection 对象是有效的。当相关联的 PooledConnection.getConnection()方法被调用时,先前存在的任何 Connection 对象都会被自动关闭。在这种情况下,监听者不会被告知。这给应用程序服务端提供了一种撤消客户端连接的方法,例如应用程序服务端强制关机,尽管这是一种不太可能的假设,但是非常有用。

(5)连接池管理者调用 PooledConnection.close()方法关闭一个物理连接。这个方法在特定的环境下被调用,例如应用程序服务端正执行一个顺序的关闭,或者连接缓存正被重新初始化,或者应用程序服务端收到一个事件表明某个连接发生了不可挽回的错误。

#### 5) 配置

配置一个实现连接池的 DataSource 对象需要一个客户端可见的 DataSource 对象和一个被基于 JNDI 命名服务注册的底层 ConnectionPoolDataSource 对象。

例:

配置一个 ConnectionPoolDataSource 对象

```
// ConnectionPoolDS implements the ConnectionPoolDataSource
```

```
// interface. Create an instance and set properties.  
com.acme.jdbc.ConnectionPoolDS cpds = new com.acme.jdbc.ConnectionPoolDS();  
cpds.setServerName("bookserver");  
cpds.setDatabaseName("booklist");  
cpds.setPortNumber(9040);  
cpds.setDescription("Connection pooling for bookserver");  
// Register the ConnectionPoolDS with JNDI, using the logical name  
// "jdbc/pool/bookserver _ pool"  
Context ctx = new InitialContext();  
ctx.bind("jdbc/pool/bookserver _ pool", cpds);
```

一旦上述步骤完成,对客户端可见的 DataSource 实现来说,ConnectionPoolDataSource 实现是可用的。然后配置 DataSource 使它引用 ConnectionPoolDataSource 实现。

例:

通过 ConnectionPoolDataSource 对象配置 DataSource 对象。现在 DataSource 对象可在应用程序中使用了。

```
// PooledDataSource implements the DataSource interface.  
// Create an instance and set properties.  
com.acme.appserver.PooledDataSource ds =  
new com.acme.appserver.PooledDataSource();  
ds.setDescription("datasource with Connection pooling");  
// Reference the previously registered ConnectionPoolDataSource  
ds.setDataSourceName("jdbc/pool/bookserver _ pool");  
// Register the DataSource implementation with JNDI, using the logical  
// name "jdbc/bookserver"  
Context ctx = new InitialContext();  
ctx.bind("jdbc/bookserver", ds);
```

#### (1) 池缓存语句的重用

池缓存语句允许应用程序重用 PreparedStatement 对象,与能重用 Connection 一样,在大多数地方通过缓存连接实现。

图 10-9 是 PreparedStatement 对象的缓存池怎样与一个 PooledConnection 对象连接的逻辑视图。与 PooledConnection 对象本身一样,PreparedStatement 对象能被多个逻辑连接以透明方式重用。

#### 6) 池缓存语句的使用

如果池缓存连接要重用语句对象,那么这种重用对应用程序必须是完全透明的,换句话说,从应用程序的角度来看,使用一个参与共享池缓存语句的 PreparedStatement 对象确实与使用不参与共享池缓存语句的 PreparedStatement 是一样的。应用程序根本不用改变自身代码就能重用池缓存语句。如果应用程序关闭了 PreparedStatement 对象,那么它仍能调用 Connection.prepareStatement,以从新使用 PreparedStatement。池缓存语句的唯一可见效果是性能的提高。

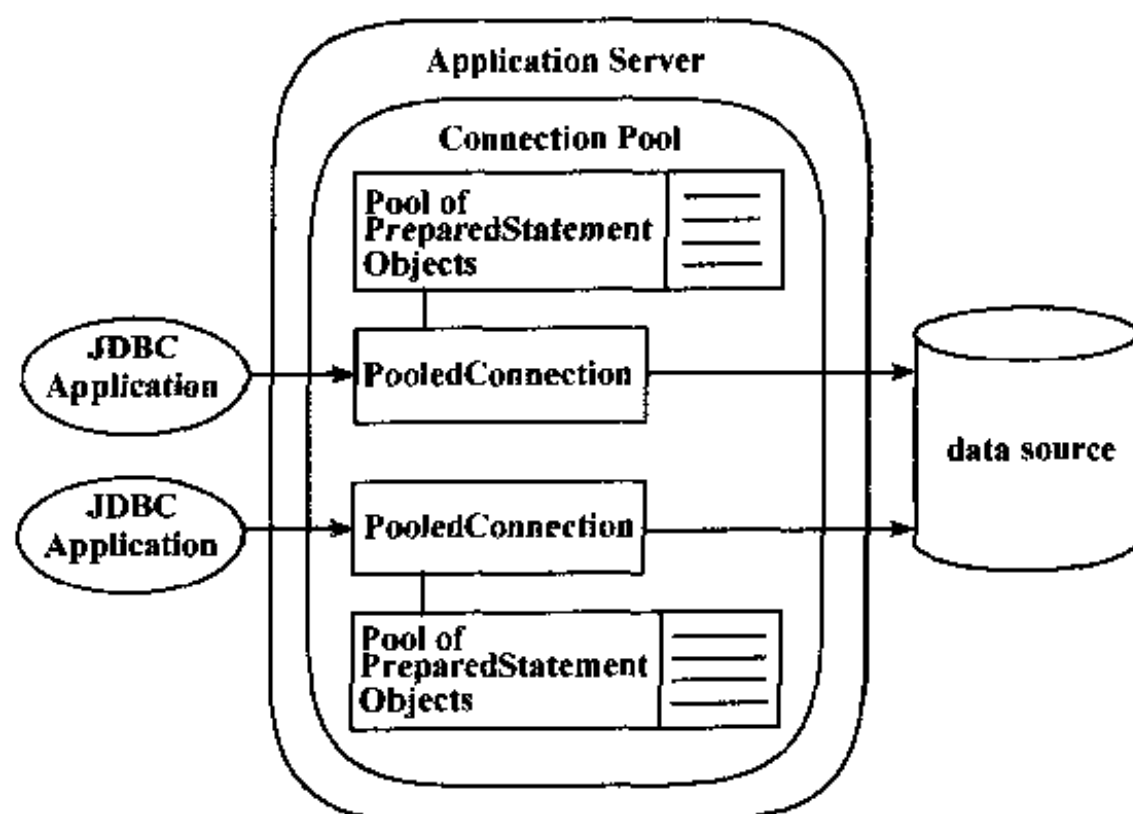


FIGURE 10-2 Logical view of prepared statement reused by pooled connections

图 10-9 PooledConnection 重用的 PreparedStatement 对象的逻辑视图

### 7) 关闭缓存池存语句

应用程序关闭一个缓存池语句与关闭一个非缓存池语句完全一样,不管缓存与否,对应用程序来说,那些已被关闭的语句不再有效,并且试图重用将导致异常抛出。

下面的方法关闭缓存池语句

(1) `Statement.close()`——由应用程序调用,如果当前语句被缓存,关闭应用程序使用的逻辑语句而不关闭已缓存的物理语句。

(2) `Connection.close()`——由应用程序调用。

►非缓存池连接。关闭由连接池建立的物理连接和所有语句。这是必需的,因为外部管理的资源被释放时,垃圾收集机制不能检测到。

►缓存池连接。关闭逻辑连接和逻辑语句,但是剩下底层的 `PooledConnection` 对象和任何相关的连接池语句处于打开状态。

(3) `PooledConnection.closeAll()`——由连接池管理器调用以关闭所有已被 `PooledConnection` 对象缓存的物理语句。应用程序不能直接关闭一个已被池缓存的物理语句,而应由连接池管理器关闭。方法 `PooledConnection.closeAll()` 关闭所有的处于打开状态的语句,它释放与这些语句相关的资源。应用程序不能直接操纵语句怎样被缓存。

## 2. 界面说明

### 1) 界面声明

```
javax.sql Interface PooledConnection
public interface PooledConnection
```

`PooledConnection` 对象是 `Connection` 对象的一个分支,具有连接池管理能力。一个 `PooledConnection` 对象表示对一个数据源的一个物理连接。

## 2) 所有已知子界面

XAConnection

## 3) 方法一览

(1) void addConnectionEventListener(ConnectionEventListener listener)

添加一个事件监听者。

(2) void close()

关闭物理连接。

(3) void closeAll()

关闭由当前 PooledConnection 对象打开的所有 Statement 对象。

(4) java.sql.Connection getConnection()

建立当前物理连接(ConnectionCreate)的一个对象句柄。

(5) void removeConnectionEventListener(ConnectionEventListener listener)

删除一个事件监听者。

## 10.8 Statement, PreparedStatement 和 CallableStatement

Statement 界面定义了一系列执行不带参数 SQL 语句的方法; PreparedStatement 界面增加了设置 SQL 输入参数的方法; CallableStatement 增加了从存储程序获得输出参数值的方法。

### 10.8.1 Statement

#### 1. 概述

## 1) 建立语句对象

例:

建立一个语句对象

```
Connection conn = dataSource.getConnection(user, passwd);
```

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

例:

在一个 Connection 对象建立多个被程序同时使用的 Statement 对象。

```
// get a Connection from the DataSource object ds
```

```
Connection conn = ds.getConnection(user, passwd);
```

```
// create two instances of Statement
```

```
Statement stmt1 = conn.createStatement();
```

```
Statement stmt2 = conn.createStatement();
```

## 2) 设置 ResultSet 特性

构造器可用来设置 Statement 产生的任一 ResultSets 的类型的并发性、持有能力。

例:

建立一个可滚动的,不敏感的,可更新的,当 commit 方法被调用后,仍保持打开状态的 ResultSet 对象。

```
Connection conn = ds.getConnection(user, passwd);
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATABLE,
    ResultSet.HOLD_CURSOR_OVER_COMMIT);
```

## 3) 执行语句

执行 Statement 对象的方法依赖于被执行的 SQL 语句的类型,如果 Statement 对象代表一个返回值是 ResultSet 对象的 SQL 查询语句,应当使用方法 executeQuery。如果 SQL 是一个 DDL 语句或一个 DML 语句,返回一个更新集,应当使用方法 executeUpdate。如果 SQL 语句的类型不知道,则应当使用方法 execute。

例:

执行语句对象,返回一个 ResultSet 对象。

```
Statement stmt = conn.createStatement();
ResultSet rs = stmt.executeQuery("select TITLE, AUTHOR, ISBN " +
    "from BOOKLIST");
while (rs.next()) {
    ...
}
```

如果执行 SQL 语句后,不能返回一个 ResultSet 对象,方法 executeQuery 将抛出一个 SQLException 异常。

例:

执行一个 Statement 对象返回更新数量。

```
Statement stmt = conn.createStatement();
int rows = stmt.executeUpdate("update STOCK set ORDER = 'Y' +
    "where SUPPLY = 0");
if (rows > 0) {
    ...
}
```

如果执行 SQL 语句后,不能返回一个更新数量,方法 executeUpdate 将抛出一个 SQLException 异常。

## 4) 使用方法 execute

方法 execute 用在执行 SQL 语句返回值,或是一个更新数量,或是一个 Result set 对象。当结果是一个 Resultset 对象时,execute 方法返回“真”,当结果是一个更新数量时,execute 方法返回“假”。



例:

执行一个语句对象返回一个更新数量或返回一个 ResultSet 对象。

```
String sql;
...
Statement stmt = conn.createStatement();
boolean b = stmt.execute(sql);
if (b == true) {
    // b is true if a ResultSet is returned
    ResultSet rs;
    rs = stmt.getResultSet();
    while (rs.next()) {
        ...
    }
} else {
    // b is false if an update count is returned
    int rows = stmt.getUpdateCount();
    if (rows > 0) {
        ...
    }
}
```

执行 SQL 语句返回一个 ResultSet 对象, getUpdateCount() 方法返回 -1, 如果执行 SQL 语句返回一个更新数量, getResultSet() 方法返回 null。

#### 5) 关闭 Statement 对象

应用程序调用 Statement.close() 方法表明它已经完成处理一条语句。当建立 Statement 对象的连接被关闭时, 所有建立在其上的 Statement 对象都将被关闭。然而, 对应用程序来说, 一旦这些 Statement 对象完成处理语句就关闭它们是一个好的编程习惯, 这使得 Statement 使用的任何外部资源都将立即被释放。关闭一个 Statement 对象将关闭并使该 Statement 对象产生的任何 ResultSet 实例都无效。ResultSet 对象所拥有的资源不会被释放, 直到垃圾收集器再次运行。所以一个好的习惯是当 ResultSet 对象不再需要时就显式关闭它们。

关于关闭 Statement 对象的这些解释同样适用于 PreparedStatement 与 CallableStatement 对象。

## 2. 界面说明

### 1) 界面声明

```
java.sql Interface Statement
```

```
public interface Statement
```

用来执行一条静态的 SQL 语句并获得产生的结果。

无论何时何地由一个 Statement 对象产生的多个 ResultSet 对象只能有一个处于打开

状态,就是说一个 Statement 对象一次产生一个 ResultSet 对象。因此,如果一个 ResultSet 对象的读取与另一个 ResultSet 对象的读取是交替进行的,那么每个 ResultSet 对象必须由不同的语句对象产生。如果存在一个 ResultSet 对象处于打开状态,那么 Statement 对象的所有执行方法隐含关闭当前 ResultSet 对象。

## 2) 所有已知子界面

CallableStatement, PreparedStatement

## 3) 参照

Connection.createStatement(), ResultSet

## 4) 域一览

### (1) static int CLOSE \_ ALL \_ RESULTS

指明调用 getMoreResults 时,先前保持打开的所有 ResultSet 对象都将被关闭。

### (2) static int CLOSE \_ CURRENT \_ RESULT

指明调用 getMoreResults 时,当前 ResultSet 对象将被关闭。

### (3) static int EXECUTE \_ FAILED

指明执行一个批语句时发生了一个错误。批语句:语句的聚集,用来处理大量具有相似特征的数据。

### (4) static int KEEP \_ CURRENT \_ RESULT

指明调用 getMoreResults 时,当前 ResultSet 对象不会被关闭。

### (5) static int SUCCESS \_ NO \_ INFO

指明一个批语句执行成功,但该批语句影响行的数量——count 是无效的。

## 5) 方法一览

### (1) void addBatch(java.lang.String sql)

添加一条 SQL 命令到当前 Statement 对象的当前命令组中。

### (2) void cancel()

如果 DBMS 和驱动程序都支持终止一条 SQL 语句,那么取消当前 Statement 对象。

### (3) void clearBatch()

使当前批命令中的命令集合为空。

### (4) void clearWarnings()

清除对当前 Statement 对象的所有警告。

### (5) void close()

立即释放当前 Statement 对象的数据库和 JDBC 资源,而不是等待当前 Statement 自动关闭时释放这些资源。

### (6) boolean execute(java.lang.String sql)

执行一条可能会返回多个结果的 SQL 语句。

### (7) boolean execute(java.lang.String sql, int flag)

执行一条可能会返回多个结果的 SQL 语句。

### (8) boolean execute(java.lang.String sql, int[] columnIndexes)

执行一条可能会返回多个结果的 SQL 语句。

### (9) boolean execute(java.lang.String sql, java.lang.String[] columnNames)

执行一条可能会返回多个结果的 SQL 语句。

(10) `int[] executeBatch()`

提交一组命令给数据库执行并且如果所有命令成功执行,返回一个更新数量的数组。

(11) `ResultSet executeQuery(java.lang.String sql)`

执行一条返回单个 `ResultSet` 对象的 SQL 语句。

(12) `int executeUpdate(java.lang.String sql)`

执行一条 SQL INSERT, UPDATE 或 DELETE 语句。

(13) `int executeUpdate(java.lang.String sql, int flag)`

执行一条 SQL INSERT, UPDATE 或 DELETE 语句。

(14) `int executeUpdate(java.lang.String sql, int[] columnIndexes)`

执行一条 SQL INSERT, UPDATE 或 DELETE 语句。

(15) `int executeUpdate(java.lang.String sql, java.lang.String[] columnNames)`

执行一条 SQL INSERT, UPDATE 或 DELETE 语句。

(16) `Connection getConnection()`

返回产生当前 `Statement` 对象的 `Connection` 对象。

(17) `int getFetchDirection()`

获得从数据库取出行的取出方向,对当前 `Statement` 对象产生的结果集合来说,方向是默认的。

(18) `int getFetchSize()`

获得结果集合行的数量,对当前 `Statement` 对象产生的结果集合来说,数量是默认的取出大小。

(19) `ResultSet getGeneratedKeys()`

从当前语句获得任何自动产生的关键字。

(20) `int getMaxFieldSize()`

返回任意列的值所允许的最大尺寸,以字节为单位。

(21) `int getMaxRows()`

返回一个 `ResultSet` 对象能容纳的最大行数。

(22) `boolean getMoreResults()`

移动到 `Statement` 对象的下一个结果。

(23) `boolean getMoreResults(int current)`

移动到 `Statement` 对象的下一个结果。

(24) `int getQueryTimeout()`

获得驱动程序等待 `Statement` 对象执行完成的最长时间,以秒为单位。

(25) `ResultSet getResultSet()`

以 `ResultSet` 对象形式返回当前结果。

(26) `int getResultSetConcurrency()`

获得当前 `Statement` 对象产生的 `ResultSet` 对象的并发性。

(27) `int getResultSetHoldability()`

获得当前 `Statement` 对象产生的 `ResultSet` 对象的持有能力。

(28) int getResultSetType()

获得当前 Statement 对象产生的 ResultSet 对象的类型。

(29) int getUpdateCount()

把当前结果当作一个更新数量返回,如果结果是一个 ResultSet 对象或没有更多结果,返回 -1。

(30) SQLWarning getWarnings()

获得调用者报告的对当前 Statement 对象的第一个警告。

(31) void setCursorName(java.lang.String name)

定义后继 Statement 对象执行方法使用的 SQL 指针名。

(32) void setEscapeProcessing(boolean enable)

设置转义处理为“开”或“关”。

(33) void setFetchDirection(int direction)

设置结果集中行的处理方向。

(34) void setFetchSize(int rows)

提供一个值给 JDBC 驱动程序,当需要更多行时,这个值就作为从数据库读取的行数的值。

(35) void setMaxFieldSize(int max)

设置一列的最大字节数限制为给出的数量。

(36) void setMaxRows(int max)

设置任意 ResultSet 对象能包含的最大行数限制为给出的数量。

(37) void setQueryTimeout(int seconds)

设置驱动程序等待一个 Statement 对象执行完成的秒数为给出的数量。

## 10.8.2 PreparedStatement

### 1. 概述

PreparedStatement 扩展自 Statement,增加了对包含在语句中的参数记录器设值的能力。PreparedStatement 对象代表已准备好的或预编译的 SQL 语句,因而一旦执行一次,就可多次重用。SQL 语句中的参数标识符用“?”表示,执行前,可使用指定输入值去改变它。

#### 1) 建立

除了 SQL 命令不同外,PreparedStatement 对象与 Statement 对象有相同的建立方式。  
例:

建立一个 PreparedStatement 对象。

```
Connection conn = ds.getConnection(user, passwd);
```

```
PreparedStatement ps = conn.prepareStatement("INSERT INTO BOOKLIST" +  
"(AUTHOR, TITLE, ISBN) VALUES (?, ?, ?)");
```

## 2) 设置 ResultSet 属性

与 `createStatement()` 方法一样, `PreparedStatement()` 方法也定义了一个构造器, 使用它可以设置 `PreparedStatement` 产生的 `ResultSet` 对象的特性。

例:

建立一个 `PreparedStatement` 对象, 使它能返回一个只能向前移动, 可更新的 `ResultSet` 对象。

```
Connection conn = ds.getConnection(user, passwd);
PreparedStatement ps = conn.prepareStatement(
    "SELECT AUTHOR, TITLE FROM BOOKLIST WHERE ISBN = ?",
    ResultSet.TYPE_FORWARD_ONLY,
    ResultSet.CONCUR_UPDATABLE);
```

## 3) 设置参数

`PreparedStatement` 界面定义了 `setter` 方法, 使用它替换预编译 SQL 语句中的每个参数的值, 方法的名字遵循下面这个模式: “set<Type>”。

例如 `setString()` 方法用来为期望是字符串的参数配置值。每个 `setter` 方法至少有两个参数, 第一个参数始终是整数, 等于要设置参数的序号。第二个参数是分配给参数的值。

例:

设置 `PreparedStatement` 对象中的参数

```
PreparedStatement ps = conn.prepareStatement("INSERT INTO BOOKLIST" +
    "(AUTHOR, TITLE, ISBN) VALUES (?, ?, ?)");
ps.setString(1, "Zamiatin, Evgenii");
ps.setString(2, "We");
ps.setLong(3, 0140185852);
```

在 `PreparedStatement` 对象运行之前, 必须为每个参数提供一个值, 如果一个参数不匹配参数标识符, 那么执行 `PreparedStatement` 对象的方法 (`executeQuery`, `executeUpdate` and `execute`) 将抛出一个 `SQLException` 异常。

当 `PreparedStatement` 对象正执行时, 它的参数不能被重新设置, 方法 `clearParameters` 可显式调用以清除曾经设定的值。另外用一个不同的值去设置参数, 将取代上次的值。

方法 `setNull` 用来设置任一参数 JDBC Null。它有两个参数, 参数标识符的序号和参数的 JDBC 类型。

例:

设置一个 String 参数为 JDBC Null。

```
ps.setNull(2, java.sql.Types.VARCHAR);
```

## 4) 获得列信息

`PreparedStatement.getMetaData()` 获得一个包含列描述信息的 `ResultSetMetaData` 对象。`ResultSetMetaData` 是返回列的集合。`ResultSetMetaData` 界面提供了返回列的数量和每行特征的方法。

例:

建立一个 ResultSetMetaData 对象并获得列信息。

```
PreparedStatement pstmt = conn.prepareStatement(
    "SELECT * FROM CATALOG");
ResultSetMetaData rsmd = pstmt.getMetaData();
int colCount = rsmd.getColumnCount();
int colType;
String colLabel;
for (int i = 1; i <= colCount; i++) {
    colType = rsmd.getColumnType(i);
    colLabel = rsmd.getColumnLabel(i);
    ...
}
```

方法 PreparedStatement.getParameterMetaData 返回一个描述 PreparedStatement 对象中参数标识符的 ParameterMetaData 对象。ParameterMetaData 界面提供了获取有关参数个数及其属性信息的方法。

例：

建立一个 ParameterMetaData 对象并获取参数信息。

```
PreparedStatement pstmt = conn.prepareStatement(
    "SELECT * FROM BOOKLIST WHERE ISBN = ?");
...
ParameterMetaData pmd = pstmt.getParameterMetaData();
int colType = pmd.getParameterType(1);
...
```

PreparedStatement 对象的执行与 Statement 对象一样, PreparedStatement 对象使用哪个方法取决于将要执行的 SQL 语句的类型。如果 PreparedStatement 对象是一个将返回 ResultSet 对象的查询, 应当使用方法 executeQuery。如果是一个将返回行数目的 DML 语句, 应当使用 executeUpdate。方法 execute 应当仅用在语句返回的类型是未知的时候。

PreparedStatement 的任何一个执行方法作为某个 SQL 语句子查询参数被调用时, 将抛出一个 SQLException 异常。

#### 5) 返回一个 ResultSet 对象

例：

预处理并执行一条语句返回一个结果集合。

```
PreparedStatement pstmt = conn.prepareStatement("SELECT AUTHOR, " +
    "TITLE FROM BOOKLIST WHERE SECTION = ?");
for (int i = 1; i <= maxSectionNumber; i++) {
    pstmt.setInt(1, i);
    ResultSet rs = pstmt.executeQuery();
    while (rs.next()) {
        // process the record
    }
}
```

```

    }
    rs.close();
}
pstmt.close();

```

如果执行该语句不能返回一个 `ResultSet` 对象, `executeQuery` 方法将抛出一个 `SQLException` 异常。

#### 6) 返回一个行数量

如果预处理和执行的语句是一个 DML 或 DDL 操作, 使用 `executeUpdate()` 方法执行。该方法返回语句影响行的数量。

例:

预处理和执行一条语句返回更新数量。

```

PreparedStatement pstmt = conn.prepareStatement(
    "update stock set reorder = 'Y' where stock < ?");
pstmt.setInt(1, 5);
int num = pstmt.executeUpdate();

```

如果执行的语句返回一个 `ResultSet` 对象, 将抛出一个 `SQLException` 异常。

#### 7) 使用 `execute` 方法

如果 `PreparedStatement` 对象的返回类型是未知的, 应当使用 `execute()` 方法执行。此时可使用 `getResultSet` 和 `getUpdateCount` 方法获得可能的结果。

例:

预处理并执行一个语句, 返回一个 `ResultSet` 对象或一个更新数量。

```

PreparedStatement pstmt = conn.prepareStatement(sqlStatement);
// set any parameters the user passes
...
boolean b = pstmt.execute();
if (b == true) {
    ResultSet rs = pstmt.getResultSet();
    // process a ResultSet
    ...
}
else {
    int rowCount = pstmt.getUpdateCount();
    // process row count
    ...
}

```

## 2. 界面说明

### 1) 界面声明

```
java.sql Interface PreparedStatement
```

```
public interface PreparedStatement extends Statement
```

表示一条已预编译过的 SQL 语句对象。

一条 SQL 语句被预编译并存储在一个 PreparedStatement 对象中。然后该对象可多次高效率的执行这条语句。

如果需要任意的参数类型转换,目标 SQL 类型应使用 setObject 方法。

例:


设置参数。con 是活动的连接 Connection。

```
PreparedStatement pstmt = con.prepareStatement("UPDATE EMPLOYEES
```

```
SET SALARY = ? WHERE ID = ?");
```

```
pstmt.setBigDecimal(1, 153833.00)
```

```
pstmt.setInt(2, 110592)
```

 注意:设置 IN 参数值的 setXXX 方法指定类型必须与输入参数的已定义 SQL 类型一致(兼容)。例如,如果 INC 参数为 SQL 类型 Integer,那么应当使用 setInt 方法。

## 2) 所有超界面

Statement

## 3) 所有已知子界面

CallableStatement

## 4) 参照

Connection.prepareStatement(java.lang.String),

## 5) 继承自界面 java.sql.Statement 的域

CLOSE \_ ALL \_ RESULTS, CLOSE \_ CURRENT \_ RESULT, EXECUTE \_  
FAILED, KEEP \_ CURRENT \_ RESULT, SUCCESS \_ NO \_ INFO

## 6) 方法一览

### (1) void addBatch()

添加一组参数到当前 PreparedStatement 对象的命令批中。

### (2) void clearParameters()

立即清除当前参数值。

### (3) boolean execute()

执行任何一种 SQL 语句。

### (4) ResultSet executeQuery()

当前 PreparedStatement 对象执行 SQL 查询并返回 SQL 查询产生的结果集合。

### (5) int executeUpdate()

当前 PreparedStatement 对象执行 SQL INSERT, UPDATE 或 DELETE 语句。

### (6) ResultSetMetaData getMetaData()

获得一个 ResultSet 对象的列的数量、类型和属性。

### (7) ParameterMetaData getParameterMetaData()

获得一个 PreparedStatement 对象的参数的数量、类型和属性。

### (8) void setArray(int i, Array x)



将指定参数设置为给出的 Array 对象。

(9) void setAsciiStream(int parameterIndex, java.io.InputStream x, int length)

将指定参数设置为给出的输入流,其长为指定值,以字节为单位。

(10) void setBigDecimal(int parameterIndex, java.math.BigDecimal x)

将指定参数设置为一个 java.math.BigDecimal 类型的值。

(11) void setBinaryStream(int parameterIndex, java.io.InputStream x, int length)

将指定参数设置为给出的输入流,其长为指定值,以字节为单位。

(12) void setBlob(int i, Blob x)

将指定参数设置为一个给出的 Blob 对象。

(13) void setBoolean(int parameterIndex, boolean x)

将指定参数设置为一个 Java boolean 类型的值。

(14) void setByte(int parameterIndex, byte x)

将指定参数设置为一个 Java byte 类型的值。

(15) void setBytes(int parameterIndex, byte[] x)

将指定参数设置为一个 Java 字节数组类型值。

(16) void setCharacterStream(int parameterIndex, java.io.Reader reader, int length)

将指定参数设置为给出的 Reader 对象,其长为给出的数值,以字符为单位。

(17) void setClob(int i, Clob x)

将指定参数设置为给出的 Clob 对象。

(18) void setDate(int parameterIndex, Date x)

将指定参数设置为一个 java.sql.Date 类型的值。

(19) void setDate(int parameterIndex, Date x, java.util.Calendar cal)

用给出的 Calendar 对象,将指定参数设置为一个 java.sql.Date 类型的值。

(20) void setDouble(int parameterIndex, double x)

将指定参数设置为一个 Java double 类型的值。

(21) void setFloat(int parameterIndex, float x)

将指定参数设置为一个 Java float 类型的值。

(22) void setInt(int parameterIndex, int x)

将指定参数设置为一个 Java int 类型的值。

(23) void setLong(int parameterIndex, long x)

将指定参数设置为一个 Java long 类型的值。

(24) void setNull(int parameterIndex, int sqlType)

将指定参数设置为 SQL Null。

(25) void setNull(int paramIndex, int sqlType, java.lang.String typeName)

将指定参数设置为 SQL Null。

(26) void setObject(int parameterIndex, java.lang.Object x)

用给出的对象设置指定参数的值。

(27) void setObject(int parameterIndex, java.lang.Object x, int targetSqlType)

用给出的对象设置指定参数的值。

(28) void setObject(int parameterIndex, java.lang.Object x, int targetSqlType, int scale)

用给出的对象设置指定参数的值。

(29) void setRef(int i, Ref x)

将指定参数设置为给出的 Ref 类型的值。

(30) void setShort(int parameterIndex, short x)

将指定参数设置为一个 Java short 类型的值。

(31) void setString(int parameterIndex, java.lang.String x)

将指定参数设置为一个 Java String 类型的值。

(32) void setTime(int parameterIndex, Time x)

将指定参数设置为一个 java.sql.Time 类型的值。

(33) void setTime(int parameterIndex, Time x, java.util.Calendar cal)

使用给出的 Calendar 对象,将指定参数设置为一个 java.sql.Time 类型的值。

(34) void setTimestamp(int parameterIndex, Timestamp x)

将指定参数设置为一个 java.sql.Timestamp 类型的值。

(35) void setTimestamp(int parameterIndex, Timestamp x, java.util.Calendar cal)

使用给出的 Calendar 对象,将指定参数设置为一个 java.sql.Timestamp 类型的值。

(36) void setUnicodeStream(int parameterIndex, java.io.InputStream x, int length)

不赞成使用。

(37) void setURL(int parameterIndex, java.net.URL x)

将指定参数设置为一个 java.net.URL 类型的值。

7) 继承自界面 java.sql.Statement 的方法

addBatch, cancel, clearBatch, clearWarnings, close, execute, execute, execute, execute, executeBatch, executeQuery, executeUpdate, executeUpdate, executeUpdate, executeUpdate, getConnection, getFetchDirection, getFetchSize, getGeneratedKeys, getMaxFieldSize, getMaxRows, getMoreResults, getMoreResults, getQueryTimeout, getResultSet, getResultSetConcurrency, getResultSetHoldability, getResultSetType, getUpdateCount, getWarnings, setCursorName, setEscapeProcessing, setFetchDirection, setFetchSize, setMaxFieldSize, setMaxRows, setQueryTimeout

### 10.8.3 CallableStatement

#### 1. 概述

CallableStatement 界面扩展自 PreparedStatement,提供执行和从存储程序中获得结果的方法。

##### 1) 建立

与 Statement 和 PreparedStatement 对象一样,CallableStatement 对象由 Connection 对象建立。下例建立一个 CallableStatement 以调用一个存储过程 'validate'。CallableState-

ment 对象有一个返回参数和其它两个参数。

例：

建立一个 CallableStatement 对象。

```
CallableStatement cstmt = conn.prepareCall("(? = call validate(?, ?))");
```

## 2) 设置参数

CallableStatement 对象可以带三种类型的参数:IN,OUT,INOUT。这些参数或作为序号参数或作为名字参数被指定。语句中的每个参数标识符都必须设置一个值。

存储过程参数的数量、类型和属性可使用 DatabaseMetaData 的 getProcedureColumns 方法获得。

顺序参数,可以通过相应的 setter 方法传递给它。它是对语句中的参数占位符的引用,以 1 开始,参数占位符不随着语句中字面意义上的参数个数的增加而增加。

例：

指定顺序参数。

```
CallableStatement cstmt = con.prepareCall("{CALL PROC(?, 'Literal _ Value', ?)}");
```

```
cstmt.setString(1, "First");
```

```
cstmt.setString(2, "Third");
```

名字参数用来指定特殊的参数。这在过程有许多默认参数值时是非常有用的。对应 COLUMN \_ NAME 域的参数名字可使用 DatabaseMetaData.getProcedureColumns 方法获得。

例：

设置存储过程的两个输入参数。

```
CallableStatement cstmt = con.prepareCall("{CALL COMPLEX _ PROC(?, ?)}");
```

```
cstmt.setString("PARAM _ 1", "Price");
```

```
cstmt.setFloat("PARAM _ 5", 150.25);
```

## (1) IN 参数

使用 setter 方法分配值给 IN 参数。详细内容见 PreparedStatement 设置参数。

例：

设置 IN 参数。

```
cstmt.setString(1, "October");
```

```
cstmt.setDate(2, date);
```

## (2) OUT 参数

方法 registerOutParameter 必须在 CallableStatement 执行前调用以设置每个 OUT 参数的类型。

OUT 参数的值可用相应的 getter 方法获得。

例：

注册和获得 OUT 参数。例子是有两个 OUT 参数的存储过程的执行,一个参数是 String,另一个是 Float。

```
CallableStatement cstmt = conn.prepareCall(
```

```

        "(CALL GET _ NAME _ AND _ NUMBER(?, ?)|)";
cstmt.registerOutParameter(1, java.sql.Types.STRING);
cstmt.registerOutParameter(2, java.sql.Types.FLOAT);
cstmt.execute();
// Retrieve OUT parameters
String name = cstmt.getString(1);
float number = cstmt.getFloat(2);

```

## 2. 界面说明

### 1) 界面声明

java.sql Interface CallableStatement

public interface CallableStatement extends PreparedStatement

这个界面用来执行 SQL 存储的过程。JDBC 提供一个存储过程的 SQL 转义语法,转义语法允许存储过程以一种标准方式被所有的 RDBMS(关系数据库)调用。转义语法有包含一个结果参数的形式和不包含一个结果参数的形式。如果使用它,这个结果参数必须注册为一个 OUT 参数。其它的参数用作输入,输出或二者兼俱。参数被顺序引用,用数字表示,第一个参数就是 1。

```

{? = call <procedure—name> [<arg1>, <arg2>, ...]}
{call <procedure—name> [<arg1>, <arg2>, ...]}

```

用来设置 IN 参数值的 set 方法继承自 PreparedStatement。所有 OUT 参数的类型必须先于执行存储过程被注册;它们的值在执行后通过这里提供的 get 方法获得。

一个 CallableStatement 返回的一个或多个 ResultSet 对象都可用继承自 Statement 的操作来处理。

为实现最大的可移植性,调用者的 ResultSet 对象和更新计数应当先于获得 output 参数数值被处理。

### 2) 所有超界面

PreparedStatement, Statement

### 3) 参照

Connection.prepareCall(java.lang.String), ResultSet

### 4) 继承自界面 java.sql.Statement 的域

CLOSE \_ ALL \_ RESULTS, CLOSE \_ CURRENT \_ RESULT, EXECUTE \_  
FAILED, KEEP \_ CURRENT \_ RESULT, SUCCESS \_ NO \_ INFO

### 5) 方法一览

#### (1) Array getArray(int i)

以 Java 语言的 Array 对象形式返回一个 JDBC ARRAY 参数的值。

#### (2) Array getArray(java.lang.String parameterName)

以 Java 语言的 Array 对象形式返回一个 JDBC ARRAY 参数的值。

#### (3) java.math.BigDecimal getBigDecimal(int parameterIndex)

以 Java 语言的 java.math.BigDecimal 对象形式返回一个 JDBC NUMERIC 参数的值。

(4) `java.math.BigDecimal getBigDecimal(int parameterIndex, int scale)`

不赞成使用

(5) `java.math.BigDecimal getBigDecimal(java.lang.String parameterName)`

以 Java 语言的 `java.math.BigDecimal` 对象形式返回一个 JDBC NUMERIC 参数的值。

(6) `Blob getBlob(int i)`

以 Java 语言的 `Blob` 对象形式返回一个 JDBC BLOB 参数的值。

(7) `Blob getBlob(java.lang.String parameterName)`

以 Java 语言的 `Blob` 对象形式返回一个 JDBC BLOB 参数的值。

(8) `boolean getBoolean(int parameterIndex)`

以 Java 语言的 `boolean` 形式返回一个 JDBC BIT 参数的值。

(9) `boolean getBoolean(java.lang.String parameterName)`

以 Java 语言的 `boolean` 形式返回一个 JDBC BIT 参数的值。

(10) `byte getByte(int parameterIndex)`

以 Java 语言的 `byte` 形式返回一个 JDBC TINYINT 参数的值。

(11) `byte getByte(java.lang.String parameterName)`

以 Java 语言的 `byte` 形式返回一个 JDBC TINYINT 参数的值。

(12) `byte[] getBytes(int parameterIndex)`

以 Java 语言的 `byte` 数组形式返回一个 JDBC BINARY 或 VARBINARY 参数的值。

(13) `byte[] getBytes(java.lang.String parameterName)`

以 Java 语言的 `byte` 数组形式返回一个 JDBC BINARY 或 VARBINARY 参数的值。

(14) `Clob getClob(int i)`

以 Java 语言的 `Clob` 对象形式返回一个 JDBC CLOB 参数的值。

(15) `Clob getClob(java.lang.String parameterName)`

以 Java 语言的 `Clob` 对象形式返回一个 JDBC CLOB 参数的值。

(16) `Date getDate(int parameterIndex)`

以 `java.sql.Date` 对象形式返回一个 JDBC DATE 参数的值。

(17) `Date getDate(int parameterIndex, java.util.Calendar cal)`

以 `java.sql.Date` 对象形式返回一个 JDBC DATE 参数的值,返回的 `date` 由给出的 `Calendar` 构造。

(18) `Date getDate(java.lang.String parameterName)`

以 `java.sql.Date` 对象形式返回一个 JDBC DATE 参数的值。

(19) `Date getDate(java.lang.String parameterName, java.util.Calendar cal)`

以 `java.sql.Date` 对象形式返回一个 JDBC DATE 参数的值,返回的 `date` 由给出的 `Calendar` 构造。

(20) `double getDouble(int parameterIndex)`

以 Java 语言的 `double` 形式返回一个 JDBC DOUBLE 参数的值。

(21) `double getDouble(java.lang.String parameterName)`

以 Java 语言的 `double` 形式返回一个 JDBC DOUBLE 参数的值。

(22) `float getFloat(int parameterIndex)`

以 Java 语言的 float 形式返回一个 JDBC FLOAT 参数的值。

(23) float getFloat(java.lang.String parameterName)

以 Java 语言的 float 形式返回一个 JDBC FLOAT 参数的值。

(24) int getInt(int parameterIndex)

以 Java 语言的 int 形式返回一个 JDBC INTEGER 参数的值。

(25) int getInt(java.lang.String parameterName)

以 Java 语言的 int 形式返回一个 JDBC INTEGER 参数的值。

(26) long getLong(int parameterIndex)

以 Java 语言的 long 形式返回一个 JDBC LONG 参数的值。

(27) long getLong(java.lang.String parameterString)

以 Java 语言的 long 形式返回一个 JDBC LONG 参数的值。

(28) java.lang.Object getObject(int parameterIndex)

以 Java 语言的 Object 返回一个参数的值。

(29) java.lang.Object getObject(int i, java.util.Map map)

返回一个表示 OUT 参数 i 值的对象并使用自定义映射得到参数的值。

(30) java.lang.Object getObject(java.lang.String parameterName)

以 Java 语言的 Object 返回一个参数的值。

(31) java.lang.Object getObject(java.lang.String parameterName, java.util.Map map)

返回一个表示 OUT 参数 i 值的对象并使用自定义映射得到参数的值。

(32) Ref getRef(int i)

以 Java 语言的 Ref 对象形式返回一个 JDBC REF 参数的值。

(33) Ref getRef(java.lang.String ParameterName)

以 Java 语言的 Ref 对象形式返回一个 JDBC REF 参数的值。

(34) short getShort(int parameterIndex)

以 Java 语言的 short 形式返回一个 JDBC SMALLINT 参数的值。

(35) short getShort(java.lang.String parameterName)

以 Java 语言的 short 形式返回一个 JDBC SMALLINT 参数的值。

(36) java.lang.String getString(int parameterIndex)

以 Java 语言的 String 形式返回一个 JDBC CHAR, VARCHAR 或 LONGVARCHAR 参数的值。

(37) java.lang.String getString(java.lang.String parameterName)

以 Java 语言的 String 形式返回一个 JDBC CHAR, VARCHAR 或 LONGVARCHAR 参数的值。

(38) Time getTime(int parameterIndex)

以 Java 语言的 java.sql.Time 对象形式返回一个 JDBC TIME 的值。

(39) Time getTime(int parameterIndex, java.util.Calendar cal)

以 Java 语言的 java.sql.Time 对象形式返回一个 JDBC TIME 的值,返回的 time 是由给出的 Calendar 对象构造的。

(40) Time getTime(java.lang.String parameterName)

以 Java 语言的 `java.sql.Time` 对象形式返回一个 JDBC TIME 的值。

(41) `Time getTime(java.lang.String parameterName, java.util.Calendar cal)`

以 Java 语言的 `java.sql.Time` 对象形式返回一个 JDBC TIME 的值,返回的 time 是由给出的 `Calendar` 对象构造的。

(42) `Timestamp getTimestamp(int parameterIndex)`

以 Java 语言的 `java.sql.Timestamp`(时间戳)对象形式返回一个 JDBC TIMESTAMP 的值。

(43) `Timestamp getTimestamp(int parameterIndex, java.util.Calendar cal)`

以 Java 语言的 `java.sql.Timestamp` 对象形式返回一个 JDBC TIMESTAMP 的值,返回的 `Timestamp` 对象是由给出的 `Calendar` 对象构造的。

(44) `Timestamp getTimestamp(java.lang.String parameterName)`

以 Java 语言的 `java.sql.Timestamp` 对象形式返回一个 JDBC TIMESTAMP 的值。

(45) `Timestamp getTimestamp(java.lang.String parameterName, java.util.Calendar cal)`

以 Java 语言的 `java.sql.Timestamp` 对象形式返回一个 JDBC TIMESTAMP 的值,返回的 `Timestamp` 对象是由给出的 `Calendar` 对象构造的。

(46) `java.net.URL getURL(int parameterIndex)`

以 `java.net.URL` 对象形式返回一个 JDBC DATALINK 的值。

(47) `void registerOutParameter(int parameterIndex, int sqlType)`

注册位置参数指定的 OUT 参数为 JDBC 类型 `sqlType`。

(48) `void registerOutParameter(int parameterIndex, int sqlType, int scale)`

注册位置参数指定的参数为 JDBC `sqlType` 中的某种类型。

(49) `void registerOutParameter(int paramIndex, int sqlType, java.lang.String typeName)`

注册指定的输出参数。

(50) `void registerOutParameter(java.lang.String parameterName, int sqlType)`

注册 `parameterName` 命名的 OUT 参数为 JDBC 类型 `sqlType`。

(51) `void registerOutParameter(java.lang.String parameterName, int sqlType, int scale)`

注册 `parameterName` 命名的参数为 JDBC 类型 `sqlType`。

(52) `void registerOutParameter(java.lang.String parameterName, int sqlType, java.lang.String typeName)`

注册指定的输出参数。

(53) `void setAsciiStream(java.lang.String parameterName, java.io.InputStream x, int length)`

将指定参数设置为给出的输入流,其长为指定值,以字节为单位。

(54) `void setBigDecimal(java.lang.String parameterName, java.math.BigDecimal x)`

将指定参数设置为一个 `java.math.BigDecimal` 类型的值。

(55) `void setBinaryStream(java.lang.String parameterName, java.io.InputStream x, int length)`

将指定参数设置为给出的输入流,其长为指定值,以字节为单位。

(56) void setBoolean(java.lang.String parameterName, boolean x)

将指定参数设置为一个 Java boolean 类型的值。

(57) void setByte(java.lang.String parameterName, byte x)

将指定参数设置为一个 Java byte 类型的值。

(58) void setBytes(java.lang.String parameterName, byte[] x)

将指定参数设置为一个 Java 字节数组(array of bytes)的值。

(59) void setCharacterStream(java.lang.String parameterName, java.io.Reader reader, int length)

将指定参数设置为给出的 Reader 对象,其长为给出的数值,以字符为单位。

(60) void setDate(java.lang.String parameterName, Date x)

将指定参数设置为一个 java.sql.Date 类型的值。

(61) void setDate(java.lang.String parameterName, Date x, java.util.Calendar cal)

用给出的 Calendar 对象,将指定参数设置为一个 java.sql.Date 类型的值。

(62) void setDouble(java.lang.String parameterName, double x)

将指定参数设置为一个 Java double 类型的值。

(63) void setFloat(java.lang.String parameterName, float x)

将指定参数设置为一个 Java float 类型的值。

(64) void setInt(java.lang.String parameterName, int x)

将指定参数设置为一个 Java int 类型的值。

(65) void setLong(java.lang.String parameterName, long x)

将指定参数设置为一个 Java long 类型的值。

(66) void setNull(int paramIndex, int sqlType, java.lang.String typeName)

将指定参数设置为 SQL NULL。

(67) void setNull(java.lang.String parameterName, int sqlType)

将指定参数设置为 SQL NULL。

(68) void setObject(java.lang.String parameterName, java.lang.Object x)

用给出的对象设置指定参数的值。

(69) void setObject(java.lang.String parameterName, java.lang.Object x, int targetSqlType)

用给出的对象设置指定参数的值。

(70) void setObject(java.lang.String parameterName, java.lang.Object x, int targetSqlType, int scale)

用给出的对象设置指定参数的值。

(71) void setShort(java.lang.String parameterName, short x)

将指定参数设置为一个 Java short 类型的值。

(72) void setString(java.lang.String parameterName, java.lang.String x)

将指定参数设置为一个 Java String 类型的值。

(73) void setTime(java.lang.String parameterName, Time x)



将指定参数设置为一个 `java.sql.Time` 类型的值。

(74) `void setTime(java.lang.String parameterName, Time x, java.util.Calendar cal)`

使用给出的 `Calendar` 对象,将指定参数设置为一个 `java.sql.Time` 类型的值。

(75) `void setTimestamp(java.lang.String parameterName, Timestamp x)`

将指定参数设置为一个 `java.sql.Timestamp` 类型的值。

(76) `void setTimestamp(java.lang.String parameterName, Timestamp x, java.util.Calendar cal)`

使用给出的 `Calendar` 对象,将指定参数设置为一个 `java.sql.Timestamp` 类型的值。

(77) `void setURL(java.lang.String parameterName, java.net.URL val)`

将指定参数设置为一个 `java.net.URL` 类型的值。

(78) `boolean wasNull()`

指明最后的 OUT 参数的读出值是否为 SQL Null。

6) 继承自界面 `java.sql.PreparedStatement` 的方法

`addBatch`, `clearParameters`, `execute`, `executeQuery`, `executeUpdate`, `getMetaData`, `getParameterMetaData`, `setArray`, `setAsciiStream`, `setBigDecimal`, `setBinaryStream`, `setBlob`, `setBoolean`, `setByte`, `setBytes`, `setCharacterStream`, `setClob`, `setDate`, `setDate`, `setDouble`, `setFloat`, `setInt`, `setLong`, `setNull`, `setObject`, `setObject`, `setObject`, `setRef`, `setShort`, `setString`, `setTime`, `setTime`, `setTimestamp`, `setTimestamp`, `setUnicodeStream`, `setURL`

7) 继承自界面 `java.sql.Statement` 的方法;

`addBatch`, `cancel`, `clearBatch`, `clearWarnings`, `close`, `execute`, `execute`, `execute`, `execute`, `executeBatch`, `executeQuery`, `executeUpdate`, `executeUpdate`, `executeUpdate`, `executeUpdate`, `getConnection`, `getFetchDirection`, `getFetchSize`, `getGeneratedKeys`, `getMaxFieldSize`, `getMaxRows`, `getMoreResults`, `getMoreResults`, `getQueryTimeout`, `getResultSet`, `getResultSetConcurrency`, `getResultSetHoldability`, `getResultSetType`, `getUpdateCount`, `getWarnings`, `setCursorName`, `setEscapeProcessing`, `setFetchDirection`, `setFetchSize`, `setMaxFieldSize`, `setMaxRows`, `setQueryTimeout`

## 10.9 ResultSet

### 1. 概述

`ResultSet` 对象的属性包括三个方面:类型、并发性、持有能力。

1) `ResultSet` 对象的类型

`ResultSet` 对象的类型取决于它在两个方面的功能层次:一是指针操作方式;二是 `ResultSet` 对象怎样映射底层数据源的并发性改变,这叫做 `ResultSet` 对象的敏感性。

(1) `TYPE_FORWARD_ONLY`

结果集合不能滚动,它的指针只能向前移动,从第一行之前到最后一行之后。

结果集中的行依赖于底层数据源怎样实现结果。它包含的行要满足一定的时间,或者就是查询获得的行。

### (2) TYPE\_SCROLL\_INSENSITIVE

结果集合可滚动,它的指针可向当前位置的前和后移动,也可移动到一个绝对位置。结果集合状态为打开时,对作用在底层数据源的变化不敏感。

### (3) TYPE\_SCROLL\_SENSITIVE

结果集合可滚动,它的指针可向当前位置的前和后移动,也可移动到一个绝对位置。结果集合状态保持打开时,结果集合映射作用在底层数据源的变化。

默认类型是 TYPE\_FORWARD\_ONLY。

如果指定的类型被驱动程序支持,那么方法 `DataBaseMetaData.supportsResultSetType` 返回真,否则返回假。

## 2) ResultSet 并发性

ResultSet 对象的并发性取决于更新功能支持的层次。

### (1) CONCUR\_READ\_ONLY

ResultSet 对象使用 ResultSet 界面不能被更新。

### (2) CONCUR\_UPDATABLE

ResultSet 对象使用 ResultSet 界面能被更新。

默认的 ResultSet 并发性是 CONCUR\_READ\_ONLY。

如果指定的并发性级别被驱动程序支持,方法 `DataBaseMetaData.supportsResultSetConcurrency` 返回真,否则返回假。

应用程序调用方法 `ResultSet.getConcurrency` 能获得 ResultSet 对象的并发性级别。

## 3) ResultSet 游标持有能力

调用方法 `Connection.commit` 关闭当前事务处理中建立的 ResultSet 对象,然而,在某些情况下,这是不期望的行为。ResultSet 属性 `holdability` 使应用程序能够在 `commit` 被调用时控制 ResultSet 对象(指针)是否被关闭。

### (1) HOLD\_CURSORS\_OVER\_COMMIT

当方法 `commit` 被调用时,ResultSet 对象不会被关闭。

### (2) CLOSE\_CURSORS\_AT\_COMMIT

ResultSet 对象会被关闭。对某些应用程序在提交时,关闭指针可以获得较好的性能。

默认的持有能力与实现有关。调用 `DataBaseMetaData` 的方法 `getResultSetHoldability` 可以获得由底层数据源返回的 ResultSet 的默认 Holdability。

例:

```
Connection conn = ds.getConnection(user, passwd);
Statement stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY,ResultSet.CLOSE_CURSORS_AT_COMMIT);
```

#### 4) 指针移动

ResultSet 对象需要维持一个指针,指针指向 Result 对象的当前数据行。当 Result 对象刚被建立时,指针指向第一行之前。下列方法用于指针移动:

(1) next()—移动指针向前一行,如果指针现在指在一行之上,返回真,如果指针位于最后一行之后,返回假。

(2) previous()—移动指针向后一行,如果指针现在指在某行之上,返回真,如果指针位于第一行之前,返回假。

(3) first()—ResultSet 对象中,移动指针到第一行。如果指针现在位于第一行返回真,如果 ResultSet 对象不包含任何行,返回假。

(4) last()—ResultSet 对象中,移动指针到最后一行。如果指针现在位于最后一行返回真,如果 ResultSet 对象不包含任何行,返回假。

(5) beforeFirst()—定位指针在 ResultSet 对象的开始处,即第一行之前。如果 ResultSet 对象不包含任何的行,这个方法没有效果。

(6) afterLast()—定位指针在 ResultSet 对象的结束处,最后一行之后。如果 ResultSet 对象不包含任何的行,这个方法没有效果。

(7) relative(int rows)—相对当前位置移动指针。如果参数 rows 为 0,指针不变;如果 rows 为正,指针向前移动 rows 行,如果从最后一行到指针所在当前行的行数小于 rows 参数指定的行数,指针将定位于最后一行之后。如果参数 rows 为负,指针向后移动 rows 行,如果从第一行到指针所在当前行的行数小于 rows 参数指定的行数,指针将定位于第一行之前。如果指针定位在一有效行上,方法 relative 返回真,否则为假。如果 rows 为 1, relative() 与方法 next() 等价。如果 rows = -1, 则与 previous() 等价。

(8) absolute(int row)—定位指针在 ResultSet 对象的第 row 行。调用 absolute(0) 移动指针到第一行之前。

ResultSet 界面提供方法获得指针所在当前行的值,列可以是一列,也可以是多列。

对每个 JDBC 类型存在两种 getter 方法。一种方法是使用索引作为它的一个参数,另一个方法使用列名(列标签)作为参数。

#### 5) ResultSet 元数据

在 ResultSet 对象上使用方法 getMetaData 将返回描述那个 ResultSet 对象的所有列对象信息的 ResultSetMetaData。直到运行时,SQL 语句还是未知的,这种情况下,可以使用 ResultSetMetaData 对象决定使用那一种方法去获得数据。

例:

```
ResultSet rs = stmt.executeQuery(sqlString);
ResultSetMetaData rsmd = rs.getMetaData();
int colType[] = new int[rsmd.getColumnCount()];
for (int idx = 0, int col = 1; idx < colType.length; idx++, col++)
    colType[idx] = rsmd.getColumnType(col);
```

#### 6) 更新一行

支持并发性 CONCUR \_ UPDATABLE 的 ResultSet 对象能使用 ResultSet 的方法更新。

一个 ResultSet 对象更新其中一行是一个二步处理过程。首先,设置需要更新的列的

新值,然后将变化作用到行上。直到第二步完成,否则底层数据源的行是不会更新的。  
例:

```
Statement stmt = conn.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_UPDATABLE);
ResultSet rs = stmt.executeQuery("select author from booklist " +
"where isbn = 140185852");
rs.next();
rs.updateString("author", "Zamyatin, Evgenii Ivanovich");
rs.updateRow();
```

#### 7) 删除一行

例:

```
rs.absolute(4);
rs.deleteRow();
```

#### 8) 插入一行

移动游标到插入行,使用 ResultSet 界面的更新方法设置列的值。插入新行到 ResultSet 对象。

例:

```
// select all the columns from the table booklist
ResultSet rs = stmt.executeQuery("select author, title, isbn " + "from booklist");
rs.moveToInsertRow();
// set values for each column
rs.updateString(1, "Huxley, Aldous");
rs.updateString(2, "Doors of Perception and Heaven and Hell");
rs.updateLong(3, 60900075);
// insert the row
rs.insertRow();
// move the cursor back to its position in the result set
rs.moveToCurrentRow();
```

#### 9) 定位更新和删除

例:

```
Statement stmt1 = conn.createStatement();
stmt1.setCursorName("CURSOR1");
ResultSet rs = stmt1.executeQuery("select author, title, isbn " +
"from booklist for update of author");
// move to the row we want to update
while ( ... ) {
    rs.next()
}
String cursorName = rs.getCursorName();
```

```
Statement stmt2 = conn.createStatement();  
// now update the row  
int updateCount = stmt2.executeUpdate("update booklist "  
"set author = "Zamyatin, Evgenii Ivanovich?" +  
"where current of ? + cursorName);
```

#### 10) 关闭一个 ResultSet 对象

当产生它的语句对象被关闭时,当前 ResultSet 对象自动被关闭。也可显式调用 close 方法关闭一个 ResultSet 对象,释放任何外部资源并立即使垃圾收集器有效。

## 2. 界面说明

### 1) 界面声明

```
java.sql Interface ResultSet  
public interface ResultSet
```

通常执行语句查询数据库产生的数据表被表示为一个结果集合。

ResultSet 对象维持一个指针指向其数据的当前行。最初指针定位在第一行前。next 方法移动指针到下一行,并且当 ResultSet 对象中没有更多的行时,将返回“false”,所以 next 方法可用来遍历整个结果集合。

默认 ResultSet 对象不能被更新并且指针只能向前移动。这样,只能从第一行到最后--行进行一次遍历。JDBC 2.0 API 中的新方法可产生滚动和(或)更新的 ResultSet 对象。下面的代码片断中,con 是有效的 Connection 对象,举例说明怎样使一个结果集合可滚动,对其他用户的更新不敏感和可更新。其它选项请看 ResultSet 的域。

```
Statement stmt = con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,  
                                         ResultSet.CONCUR_UPDATABLE);  
ResultSet rs = stmt.executeQuery("SELECT a, b FROM TABLE2");  
// rs 将是可滚动的,不显示其它用户做的改变。  
// 并且可被更新。
```

ResultSet 界面提供 getXXX 方法从当前行中获得列值。值也可使用列的索引或列的名字获得。通常使用列的索引会更有效率。列从 1 开始计数。为取得最大的可移植性,每行内的结果集合各列从“左”向“右”读,并且每列只允许读一次。对 getXXX 方法,JDBC 驱动程序试图将底层数据转换成 getXXX 的 XXX 部分指定的 Java 类型并返回适当的 Java 值。

列的名字作 getXXX 方法的参数是非大小写敏感的。当一个 getXXX 方法以列名作参数被调用并且几个列有相同的名字时,将返回第一个匹配列的值。列名用在 SQL 查询中产生结果集合是一种可选设计。因为查询过程中不能明确指定列名,因此最好使用列数。如果使用列名,不能保证实际的引用正是程序设计员所期望的。

一系列的 updateXXX 方法在发布 JDBC 2.0 API 时加入该界面。关于 getXXX 方法的参数的解释也可应用到 updateXXX 方法的参数。

updateXXX 方法可有两种使用方式。

(1) 更新当前行的一个列值。对一个可滚动的 ResultSet 对象,指针可向前和向后移

动到一个绝对位置或移动到当前行的相对位置。下面的代码片断更新 ResultSet 对象 rs 第五行中名为“NAME”的列,然后使用方法 updateRow 更新导出 rs 的数据源的表。

```
rs.absolute(5); // 移动指针到 rs 的第五行。  
rs.updateString("NAME", "AINSWORTH"); // 更新第五行 NAME 列的值为“AIN-  
SWORTH”。  
rs.updateRow(); // 更新数据源的行。
```

(2) 插入列值到某个插入行。一个可更新的 ResultSet 对象有一个与之相关的特殊行,特殊行为建造插入列提供区域数据移动服务。下面的代码片断移动指针到插入行,建造一个有三列的行,并且使用 insertRow 方法将其插入 rs 和数据源的表中。

```
rs.moveToInsertRow(); // 移动指针到插入行。  
rs.updateString(1, "AINSWORTH"); // 更新插入行的第一列为“AINSWORTH”。  
rs.updateInt(2, 35); // 更新第二列为“35”。  
rs.updateBoolean(3, true); // 更新第三列为“true”。  
rs.insertRow();  
rs.moveToCurrentRow();
```

当产生 ResultSet 对象的 Statement 对象关闭时,ResultSet 对象自动关闭。只有重新执行或从多个结果序列中重新获得下一个结果。

一个 ResultSet 对象列的数量、类型和属性可由 ResultSet.getMetaData 方法返回的 ResultSetMetaData 对象提供。

## 2) 参照

Statement.executeQuery(java.lang.String), Statement.getResultSet(), ResultSetMetaData

## 3) 域一览

(1) static int CLOSE\_CURSORS\_AT\_COMMIT

指明当 commit 被调用时,ResultSet 对象应被关闭。

(2) static int CONCUR\_READ\_ONLY

指明 ResultSet 对象的并发性为不可更新模式。

(3) static int CONCUR\_UPDATABLE

指明 ResultSet 对象的并发性为可更新模式。

(4) static int FETCH\_FORWARD

指明结果集中的行以向前方式处理即第一行到最后一行。

(5) static int FETCH\_REVERSE

指明结果集中的行以倒退方式处理即最后一行到第一行。

(6) static int FETCH\_UNKNOWN

指明结果集中的行的处理顺序是未知的。

(7) static int HOLD\_CURSORS\_OVER\_COMMIT

指明 commit 被调用时,ResultSet 对象不会被关闭。

(8) static int TYPE\_FORWARD\_ONLY

指明 ResultSet 对象的指针类型为只能向前移动。

(9) static int TYPE\_SCROLL\_INSENSITIVE

指明 ResultSet 对象的指针类型为可滚动,但是通常对其他用户所做的改变不敏感。

(10) static int TYPE\_SCROLL\_SENSITIVE

指明一个 ResultSet 对象的指针类型为可滚动,并且通常对其他用户所做的改变敏感。

#### 4) 方法一览

(1) boolean absolute(int row)

在当前 ResultSet 对象中移动指针到参数 row 给出的数值处。

(2) void afterLast()

移动指针到当前 ResultSet 对象的结束处即最后一行之后。

(3) void beforeFirst()

移动指针到当前 ResultSet 对象的开始处即第一行之前。

(4) void cancelRowUpdates()

在当前 ResultSet 对象中取消对当前行所做的更新。

(5) void clearWarnings()

清除对当前 ResultSet 对象的所有警告。

(6) void close()

立即释放当前 ResultSet 对象的数据库和 JDBC 资源,而不是等待其自动关闭并释放这些资源。

(7) void deleteRow()

从当前 ResultSet 对象和底层数据库删除当前行。

(8) int findColumn(java.lang.String columnName)

将给出 ResultSet 对象列名映射到 ResultSet 的列索引。

(9) boolean first()

移动指针到当前 ResultSet 对象的第一行。

(10) Array getArray(int i)

以 Java 语言的数组对象形式返回当前 ResultSet 对象当前行中指定列的值。

(11) Array getArray(java.lang.String colName)

以 Java 语言的数组对象形式返回当前 ResultSet 对象当前行中指定列的值。

(12) java.io.InputStream getAsciiStream(int columnIndex)

以 ASCII 字符流形式返回当前 ResultSet 对象当前行中指定列的值。

(13) java.io.InputStream getAsciiStream(java.lang.String columnName)

以 ASCII 字符流形式返回当前 ResultSet 对象当前行中指定列的值。

(14) java.math.BigDecimal getBigDecimal(int columnIndex)

以完整精确的 java.math.BigDecimal 形式返回当前 ResultSet 对象当前行指定列的值。

(15) java.math.BigDecimal getBigDecimal(int columnIndex, int scale)

不赞成使用。

(16) java.math.BigDecimal getBigDecimal(java.lang.String columnName)

以完整精确的 java.math.BigDecimal 形式返回当前 ResultSet 对象当前行指定列的值。

(17) java.math.BigDecimal getBigDecimal(java.lang.String columnName, int scale)

不赞成使用。

(18) `java.io.InputStream getBinaryStream(int columnIndex)`

以不经解释的二进制字节流形式返回当前 `ResultSet` 对象当前行中指定列的值。

(19) `java.io.InputStream getBinaryStream(java.lang.String columnName)`

以不经解释的字节流形式返回当前 `ResultSet` 对象当前行中指定列的值。

(20) `Blob getBlob(int i)`

以 Java 语言的 `Blob` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(21) `Blob getBlob(java.lang.String colName)`

以 Java 语言的 `Blob` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(22) `boolean getBoolean(int columnIndex)`

以 Java 语言的 `boolean` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(23) `boolean getBoolean(java.lang.String columnName)`

以 Java 语言的 `boolean` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(24) `byte getByte(int columnIndex)`

以 Java 语言的字节(`byte`)形式返回当前 `ResultSet` 对象当前行中指定列的值。

(25) `byte getByte(java.lang.String columnName)`

以 Java 语言的字节(`byte`)形式返回当前 `ResultSet` 对象当前行中指定列的值。

(26) `byte[] getBytes(int columnIndex)`

以 Java 语言的字节数组(`byte array`)形式返回当前 `ResultSet` 对象当前行指定列的值。

(27) `byte[] getBytes(java.lang.String columnName)`

以 Java 语言的字节数组(`byte array`)形式返回当前 `ResultSet` 对象当前行指定列的值。

(28) `java.io.Reader getCharacterStream(int columnIndex)`

以 `java.io.Reader` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(29) `java.io.Reader getCharacterStream(java.lang.String columnName)`

以 `java.io.Reader` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(30) `Clob getClob(int i)`

以 Java 语言的 `Clob` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(31) `Clob getClob(java.lang.String colName)`

以 Java 语言的 `Clob` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(32) `int getConcurrency()`

返回当前 `ResultSet` 对象的并发性模式。

(33) `java.lang.String getCursorName()`

获得当前 `ResultSet` 对象使用的 SQL 指针的名字。

(34) `Date getDate(int columnIndex)`

以 Java 语言的 `java.sql.Date` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(35) `Date getDate(int columnIndex, java.util.Calendar cal)`

以 Java 语言的 `java.sql.Date` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(36) `Date getDate(java.lang.String columnName)`



以 Java 语言的 `java.sql.Data` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(37) `Date getDate(java.lang.String columnName, java.util.Calendar cal)`

以 Java 语言的 `java.sql.Data` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(38) `double getDouble(int columnIndex)`

以 Java 语言的 `double` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(39) `double getDouble(java.lang.String columnName)`

以 Java 语言的 `double` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(40) `int getFetchDirection()`

返回当前 `ResultSet` 对象处理数据的方向。

(41) `int getFetchSize()`

返回当前 `ResultSet` 对象处理数据的大小。

(42) `float getFloat(int columnIndex)`

以 Java 语言的 `float` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(43) `float getFloat(java.lang.String columnName)`

以 Java 语言的 `float` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(44) `int getInt(int columnIndex)`

以 Java 语言的 `int` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(45) `int getInt(java.lang.String columnName)`

以 Java 语言的 `int` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(46) `long getLong(int columnIndex)`

以 Java 语言的 `long` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(47) `long getLong(java.lang.String columnName)`

以 Java 语言的 `long` 形式返回当前 `ResultSet` 对象当前行中指定列的值。

(48) `ResultSetMetaData getMetaData()`

获得当前 `ResultSet` 对象列的数量、类型和属性。

(49) `java.lang.Object getObject(int columnIndex)`

以 Java 语言的对象(`Object`)形式返回当前 `ResultSet` 对象当前行中指定列的值。

(50) `java.lang.Object getObject(int i, java.util.Map map)`

以 Java 语言的对象(`Object`)形式返回当前 `ResultSet` 对象当前行中指定列的值。

(51) `java.lang.Object getObject(java.lang.String columnName)`

以 Java 语言的对象(`Object`)形式返回当前 `ResultSet` 对象当前行中指定列的值。

(52) `java.lang.Object getObject(java.lang.String colName, java.util.Map map)`

以 Java 语言的对象(`Object`)形式返回当前 `ResultSet` 对象当前行中指定列的值。

(53) `Ref getRef(int i)`

以 Java 语言的 `Ref` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(54) `Ref getRef(java.lang.String colName)`

以 Java 语言的 `Ref` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(55) `int getRow()`

获得当前行数。

(56) short getShort(int columnIndex)

以 Java 语言的 short 形式返回当前 ResultSet 对象当前行中指定列的值。

(57) short getShort(java.lang.String columnName)

以 Java 语言的 short 形式返回当前 ResultSet 对象当前行中指定列的值。

(58) Statement getStatement()

返回产生当前 ResultSet 对象的语句(Statement)对象。

(59) java.lang.String getString(int columnIndex)

以 Java 语言的 String 形式返回当前 ResultSet 对象当前行中指定列的值。

(60) java.lang.String getString(java.lang.String columnName)

以 Java 语言的 String 形式返回当前 ResultSet 对象当前行中指定列的值。

(61) Time getTime(int columnIndex)

以 Java 语言的 java.sql.Time 对象形式返回当前 ResultSet 对象当前行中指定列的值。

(62) Time getTime(int columnIndex, java.util.Calendar cal)

以 Java 语言的 java.sql.Time 对象形式返回当前 ResultSet 对象当前行中指定列的值。

(63) Time getTime(java.lang.String columnName)

以 Java 语言的 java.sql.Time 对象形式返回当前 ResultSet 对象当前行中指定列的值。

(64) Time getTime(java.lang.String columnName, java.util.Calendar cal)

以 Java 语言的 java.sql.Time 对象形式返回当前 ResultSet 对象当前行中指定列的值。

(65) Timestamp getTimestamp(int columnIndex)

以 Java 语言的 java.sql.Timestamp 对象形式返回当前 ResultSet 对象当前行指定列的值。

(66) Timestamp getTimestamp(int columnIndex, java.util.Calendar cal)

以 Java 语言的 java.sql.Timestamp 对象形式返回当前 ResultSet 对象当前行指定列的值。

(67) Timestamp getTimestamp(java.lang.String columnName)

以 Java 语言的 java.sql.Timestamp 对象形式返回当前 ResultSet 对象当前行指定列的值。

(68) Timestamp getTimestamp(java.lang.String columnName, java.util.Calendar cal)

以 Java 语言的 java.sql.Timestamp 对象形式返回当前 ResultSet 对象当前行指定列的值。

(69) int getType()

返回当前 ResultSet 对象的类型。

(70) java.io.InputStream getUnicodeStream(int columnIndex)

不赞成使用。使用 getCharacterStream 代替 getUnicodeStream。

(71) java.io.InputStream getUnicodeStream(java.lang.String columnName)

不赞成使用。使用 `getCharacterStream` 代替 `getUnicodeStream`。

(72) `java.net.URL getURL(int columnIndex)`

以 Java 语言的 `java.net.URL` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(73) `java.net.URL getURL(java.lang.String columnName)`

以 Java 语言的 `java.net.URL` 对象形式返回当前 `ResultSet` 对象当前行中指定列的值。

(74) `SQLWarning getWarnings()`

返回调用者产生的当前 `ResultSet` 对象的第一个警告。

(75) `void insertRow()`

将待插入行的内容插入 `ResultSet` 对象和数据库。

(76) `boolean isAfterLast()`

指示指针是否在当前 `ResultSet` 对象的最后一行之后。

(77) `boolean isBeforeFirst()`

指示指针是否在当前 `ResultSet` 对象的第一行之前。

(78) `boolean isFirst()`

指示指针是否指向当前 `ResultSet` 对象的第一行。

(79) `boolean isLast()`

指示指针是否指向当前 `ResultSet` 对象的最后一行。

(80) `boolean last()`

移动指针指向(到)当前 `ResultSet` 对象的最后一行。

(81) `void moveToCurrentRow()`

移动指针到曾作过记录的指针位置,通常是当前行。

(82) `void moveToInsertRow()`

移动指针到插入行。

(83) `boolean next()`

移动指针到当前行的下一行。

(84) `boolean previous()`

在当前 `ResultSet` 对象中移动指针到前一行。

(85) `void refreshRow()`

刷新当前行使其是数据库中的最新值。

(86) `boolean relative(int rows)`

移动指针一个相对行数,相对行数或为正或为负。

(87) `boolean rowDeleted()`

指示一行是否已被删除。

(88) `boolean rowInserted()`

指示当前行是否已被插入。

(89) `boolean rowUpdated()`

指示当前行是否已被更新。

(90) void setFetchDirection(int direction)

设置当前 ResultSet 对象行的处理方向。

(91) void setFetchSize(int rows)

提供一个数值给 JDBC 驱动程序,当 ResultSet 对象需要更多行时,该数值即为应当从数据库取出的行数。

(92) void updateArray(int columnIndex, Array x)

以 java.sql.Array 形式的值更新指定列。

(93) void updateArray(java.lang.String columnName, Array x)

以 java.sql.Array 形式的值更新指定列。

(94) void updateAsciiStream(int columnIndex, java.io.InputStream x, int length)

以 ASCII 流形式的值更新指定列。

(95) void updateAsciiStream(java.lang.String columnName, java.io.InputStream x, int length)

以 ASCII 流形式的值更新指定列。

(96) void updateBigDecimal(int columnIndex, java.math.BigDecimal x)

以 java.math.BigDecimal 形式的值更新指定列。

(97) void updateBigDecimal(java.lang.String columnName, java.math.BigDecimal x)

以 java.math.BigDecimal 形式的值更新指定列。

(98) void updateBinaryStream(int columnIndex, java.io.InputStream x, int length)

以二进制流形式的值更新指定列。

(99) void updateBinaryStream(java.lang.String columnName, java.io.InputStream x, int length)

以二进制流形式的值更新指定列。

(100) void updateBlob(int columnIndex, Blob x)

以 java.sql.Blob 形式的值更新指定列。

(101) void updateBlob(java.lang.String columnName, Blob x)

以 java.sql.Blob 形式的值更新指定列。

(102) void updateBoolean(int columnIndex, boolean x)

以 boolean 形式的值更新指定列。

(103) void updateBoolean(java.lang.String columnName, boolean x)

以 boolean 形式的值更新指定列。

(104) void updateByte(int columnIndex, byte x)

以 byte 形式的值更新指定列。

(105) void updateByte(java.lang.String columnName, byte x)

以 byte 形式的值更新指定列。

(106) void updateBytes(int columnIndex, byte[] x)

以 byte 数组形式的值更新指定列。

(107) void updateBytes(java.lang.String columnName, byte[] x)

以 byte 数组形式的值更新指定列。

(108) void updateCharacterStream(int columnIndex, java.io.Reader x, int length)

以字符流形式的值更新指定列。

(109) void updateCharacterStream(java.lang.String columnName, java.io.Reader reader, int length)

以字符流形式的值更新指定列。

(110) void updateClob(int columnIndex, Clob x)

以 java.sql.Clob 形式的值更新指定列。

(111) void updateClob(java.lang.String columnName, Clob x)

以 java.sql.Clob 形式的值更新指定列。

(112) void updateDate(int columnIndex, Date x)

以 java.sql.Date 形式的值更新指定列。

(113) void updateDate(java.lang.String columnName, Date x)

以 java.sql.Date 形式的值更新指定列。

(114) void updateDouble(int columnIndex, double x)

以 double 形式的值更新指定列。

(115) void updateDouble(java.lang.String columnName, double x)

以 double 形式的值更新指定列。

(116) void updateFloat(int columnIndex, float x)

以 float 形式的值更新指定列。

(117) void updateFloat(java.lang.String columnName, float x)

以 float 形式的值更新指定列。

(118) void updateInt(int columnIndex, int x)

以 int 形式的值更新指定列。

(119) void updateInt(java.lang.String columnName, int x)

以 int 形式的值更新指定列。

(120) void updateLong(int columnIndex, long x)

以 long 形式的值更新指定列。

(121) void updateLong(java.lang.String columnName, long x)

以 long 形式的值更新指定列。

(122) void updateNull(int columnIndex)

置可取空值的列为 Null。

(123) void updateNull(java.lang.String columnName)

以 Null 值更新指定列。

(124) void updateObject(int columnIndex, java.lang.Object x)

以 Object 形式的值更新指定列。

(125) void updateObject(int columnIndex, java.lang.Object x, int scale)

以 Object 形式的值更新指定列。

(126) void updateObject(java.lang.String columnName, java.lang.Object x)

以 Object 形式的值更新指定列。

(127) void updateObject(java.lang.String columnName, java.lang.Object x, int scale)

以 Object 形式的值更新指定列。

(128) void updateRef(int columnIndex, Ref x)

以 java.sql.Ref 形式的值更新指定列。

(129) void updateRef(java.lang.String columnName, Ref x)

以 java.sql.Ref 形式的值更新指定列。

(130) void updateRow()

以当前 ResultSet 对象的当前行的内容更新底层数据库。

(131) void updateShort(int columnIndex, short x)

以 short 形式的值更新指定列。

(132) void updateShort(java.lang.String columnName, short x)

以 short 形式的值更新指定列。

(133) void updateString(int columnIndex, java.lang.String x)

以 String 形式的值更新指定列。

(134) void updateString(java.lang.String columnName, java.lang.String x)

以 String 形式的值更新指定列。

(135) void updateTime(int columnIndex, Time x)

以 java.sql.Time 形式的值更新指定列。

(136) void updateTime(java.lang.String columnName, Time x)

以 java.sql.Time 形式的值更新指定列。

(137) void updateTimestamp(int columnIndex, Timestamp x)

以 java.sql.Timestamp 形式的值更新指定列。

(138) void updateTimestamp(java.lang.String columnName, Timestamp x)

以 java.sql.Timestamp 形式的值更新指定列。

(139) boolean wasNull()

判断最后一行的读出值是否为 SQL Null。

## 10.10 ResultSetMetaData

### 1. 概述

ResultSetMetaData 是用来获得有关 ResultSet 对象列的类型和属性信息的一个对象。下面的代码片段建立一个 ResultSet 对象 rs, 建立一个 ResultSetMetaData 对象 rsmd, 并且使用 rsmd 获取 rs 有多少列, rs 中的第一列是否可用在一个 WHERE 子句中。

```
ResultSet rs = stmt.executeQuery("SELECT a, b, c FROM TABLE2");
ResultSetMetaData rsmd = rs.getMetaData();
int numberOfColumns = rsmd.getColumnCount();
boolean b = rsmd.isSearchable(1);
```

## 2. 界面说明

### 1) 界面声明

java.sql Interface ResultSetMetaData

public interface ResultSetMetaData

### 2) 域一览

(1) static int columnNoNulls

指明不允许列为 NULL 值。

(2) static int columnNullable

当前常数指明允许列为 NULL 值。

(3) static int columnNullableUnknown

当前常数指明是否允许列为 NULL 值是未知的。

### 3) 方法一览

(1) java.lang.String getCatalogName(int column)

获得指定列所在表的目录名。

(2) java.lang.String getColumnClassName(int column)

返回完整的 Java 类名。如果调用 ResultSet.getObject 方法获取列值,将返回该类型的实例。

(3) int getColumnCount()

返回当前 ResultSet 对象的列数。

(4) int getColumnDisplaySize(int column)

获得指定列的最大宽度,以字符为单位。

(5) java.lang.String getColumnLabel(int column)

获得指定列用于打印输出和显示的建议标题。

(6) java.lang.String getColumnName(int column)

获得指定列的名字。

(7) int getColumnType(int column)

获得指定列的 SQL 类型。

(8) java.lang.String getColumnName(int column)

获得指定列的数据库特定的类型名字。

(9) int getPrecision(int column)

获得指定列的小数位数字。

(10) int getScale(int column)

获得指定列小数点右边的数字位数。

(11) java.lang.String getSchemaName(int column)

获得指定列所在表的模式。

(12) java.lang.String getTableName(int column)

获得指定列所在的表名。

(13) boolean isAutoIncrement(int column)

表明指定列是否为自增类型。如果是,那么也是只读的。

(14) `boolean isCaseSensitive(int column)`

表明指定列是否对状态(变化)敏感。

(15) `boolean isCurrency(int column)`

表明指定列是否是当前值。

(16) `boolean isDefinitelyWritable(int column)`

表明指定列是否是明确可写的。

(17) `int isNullable(int column)`

表明指定列的值是否允许为 null。

(18) `boolean isReadOnly(int column)`

表明指定列是否是明确不可写的。

(19) `boolean isSearchable(int column)`

表明指定列是否可用在一个 WHERE 子句中。

(20) `boolean isSigned(int column)`

表明指定列的值是否有符号数。

(21) `boolean isWritable(int column)`

表明指定列是否是可读可写的。

注释:

`getCatalogName()` 中, `catalog` (目录): 全部数据集合索引的汇总, 程序用它来查找含有特定数据集合所在的卷。

`getColumnClassName()` 中, 完整 Java 类名: 包名. 类名。如 `java.lang.String`。

`getSchemaName()` 中, `schema` (模式): 一组以数据定义语言来表达的语句集合, 该语句集合完整的描述了整个数据库的结构。

`isCaseSensitive()` 中, 是否对状态(变化)敏感取决于使用 `Statement` 对象建立 `ResultSet` 对象时使用的参数。若是状态(变化)敏感的, 将映射对数据库做的修改; 反之, 不映射对数据库做的修改。

## 10.11 SQLException 和 SQLWarning

### 10.11.1 SQLException

#### 1. 概述

`SQLException` 类及其子类提供访问数据源时发生的错误和警告。与数据源交互发生错误的时候, 一个 `SQLException` 实例将被抛出。异常包括如下信息:

- (1) 错误的文字描述。调用方法 `SQLException.getMessage` 可获得该描述信息字符串。
- (2) SQL 状态。调用方法 `SQLException.getSQLState` 可获得包含 SQL 状态的字符



串。SQL 状态字符串的值取决于底层数据源设置的值。

(3) 错误码。整型值表示导致 `SQLException` 抛出的错误。其值和意义是与特定实现相关的,也可能是底层数据源返回的实际错误码。错误码可使用 `SQLException.getErrorCode` 方法获得。

(4) 对任何连锁异常的引用。如果多于一个错误发生或导致异常抛出的事件可描述为一串事件。异常就是对当前链的引用。连锁异常可使用 `SQLException.getNextException` 方法获得。如果没有更多的连锁异常,该方法返回 `null`。

## 2. 类库说明

### 1) 类层次

```

java.lang.Object
|
+ ——— java.lang.Throwable
      |
      + ——— java.lang.Exception
            |
            + ——— java.sql.SQLException
  
```

### 2) 类声明

java.sql Class `SQLException`

public class `SQLException` extends `java.lang.Exception`

一个提供有关数据库的访问错误或其它错误信息的异常。

每个 `SQLException` 提供以下几种信息:

一个描述错误的字符串。与 Java 异常信息一样使用,通过方法 `getMessage` 获得。

一个 `SQLState`(SQL 状态)的字符串,它遵循 XOPEN `SQLState` 约定。`SQLState` 字符串的值由 XOPEN SQL 规范描述。

一个整数错误码,这是提供商特定的。通常这是底层数据库返回的实际错误码。

一个链,链接下一个异常。这可用来提供额外的错误信息。

### 3) 所有实现的界面

java.io.Serializable

### 4) 已知直接子类

`BatchUpdateException`, `SQLWarning`

### 5) 参照

Serialized Form

### 6) 构造器一览

#### (1) `SQLException()`

构造一个 `SQLException` 对象, `reason` 默认为 `null`, `SQLState` 默认为 `null` 和 `vendorCode` 默认为 0。

#### (2) `SQLException(java.lang.String reason)`

构造一个带有 `reason` 的 `SQLException` 对象, `SQLState` 默认为 `null` 和 `vendorCode` 默认为 0。

(3) `SQLException(java.lang.String reason, java.lang.String SQLState)`

构造一个带有 `reason` 和 `SQLState` 的 `SQLException` 对象, `vendorCode` 默认为 0。

(4) `SQLException(java.lang.String reason, java.lang.String SQLState, int vendorCode)`

构造一个完整的 `SQLException` 对象。

#### 7) 方法一览

(1) `int GETERRORCODE()`

获得当前 `SQLException` 对象提供商特殊的异常代码。

(2) `SQLException getNextException()`

获得链接到当前 `SQLException` 对象上的异常。该方法返回链上的下一个 `SQLException`, 如果链上没有异常了, 返回 `null`。

(3) `java.lang.String getSQLState()`

获得当前 `SQLException` 对象的 `SQLState`。

(4) `void setNextException(SQLException ex)`

添加一个 `SQLException` 对象到异常链的末尾。

8) 继承自类 `java.lang.Throwable` 的方法

`fillInStackTrace`, `getLocalizedMessage`, `getMessage`, `printStackTrace`, `printStackTrace`, `printStackTrace`, `toString`

9) 继承自类 `java.lang.Object` 的方法

`equals`, `getClass`, `hashCode`, `notify`, `notifyAll`, `wait`, `wait`, `wait`

## 10.11.2 SQLWarning

### 1. 概述

下列界面的方法将产生一个 `SQLWarning` 对象, 如果它们导致一个数据库访问警告。

(1) `Connection`。

(2) `Statement` 及其子类, `PreparedStatement` 和 `CallableStatement`。

(3) `ResultSet`。

当一个方法产生一个 `SQLWarning` 对象, 调用者不会被通知已经发生了一个数据访问警告。方法 `getWarnings` 必须由适当的对象调用以获得 `SQLWarning` 对象。

如果多个数据访问警告发生, 他们被链接到第一个警告后, 可调用方法 `SQLWarning.getNextWarning` 获得。如果链上没有更多的警告, 方法将返回 `null`。

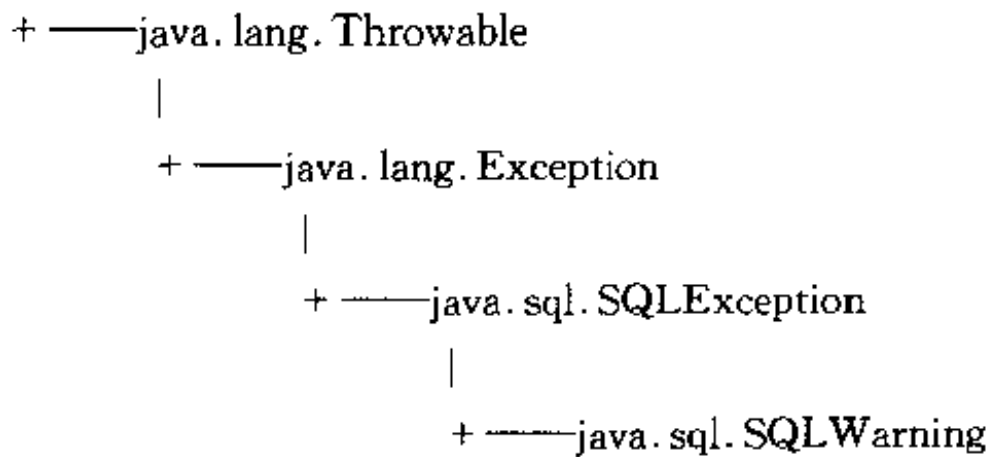
后继 `SQLWarning` 对象将继续添加到链上, 直到下一个语句被执行, 或者在 `ResultSet` 对象中, 指针被重新定位, 那么链上的所有 `SQLWarning` 对象将被删除。

### 2. 类库说明

#### 1) 类层次

`java.lang.Object`

|



## 2) 类声明

java.sql Class SQLWarning

public class SQLWarning extends SQLException

这是一个提供数据库访问警告信息的异常。警告被链接成一个对象,直到调用它的方法才报告警告的存在。

警告可发生在 Connection, Statement 和 ResultSet 对象上。试图在一个已经关闭的 Connection 上获取警告并将导致一个异常抛出。相似的,在已关闭的 Statement 和 ResultSet 上获取警告也将导致一个异常抛出。注意,关闭一个 Statement 对象也将关闭由它产生的 ResultSet 对象。

## 3) 所有实现界面

java.io.Serializable

## 4) 已知直接子类

DataTruncation

## 5) 参照

Connection. getWarnings(), Statement. getWarnings(), ResultSet. getWarnings(), Serialized Form

## 6) 构造器一览

### (1) SQLWarning()

构造一个默认的 SQLWarning 对象。

### (2) SQLWarning(java.lang.String reason)

以给出的 reason 值构造一个 SQLWarning 对象,SQLState 默认为 null, vendorCode 默认为 0。

### (3) SQLWarning(java.lang.String reason, java.lang.String SQLstate)

以给出的 reason 和 SQLState 构造一个 SQLWarning 对象, vendorCode 默认为 0。

### (4) SQLWarning(java.lang.String reason, java.lang.String SQLstate, int vendorCode)

构造一个完整的 SQLWarning 对象,并以给出的值初始化。

## 7) 方法一览

### (1) SQLWarning GETNEXTWARNING()

获得链接到当前 SQLWarning 对象上的警告。

### (2) void setNextWarning(SQLWarning w)

添加一个 SQLWarning 对象到链的末尾。

8) 继承自类 java.sql.SQLException 的方法

getErrorCode, getNextException, getSQLState, setNextException

9) 继承自类 java.lang.Throwable 的方法

fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

10) 继承自类 java.lang.Object 的方法

equals, getClass, hashCode, notify, notifyAll, wait, wait, wait

## 10.12 新闻

学了数据库后,可以做的事情就很多了。可以说,数据库是网站的根基,没有了它,就如同鱼没了水。这是我们花了这么大的代价讲述它的唯一原因。

在这里我们选择了新闻作为数据库的综合应用。本新闻系统采用了 PreparedStatement 对象并实现了分页显示。下面,来实现新闻显示与新闻发布系统。

### 10.12.1 新闻显示

#### 1. 新闻标题显示

##### 1) 建立数据源

在开始之前,假设已经建立了图 10-10 所示的数据库并建立了数据源。数据源的建立参见 10.4 节。数据库的详细资料参见光盘 database 目录下的 News.mdb。



NewsID	Title	Content	Stine
17	水瓶座	待雪草、梅花、蟹	2001. 二月. 15
18	双鱼座	野玫瑰、水仙、雉	2001. 二月. 15
19	白羊座	紫罗兰、山茶花、	2001. 二月. 15
20	金牛座	忘忧草、樱花、	2001. 二月. 15
21	双子座	铃兰、甘桔、风信	2001. 二月. 15
22	巨蟹座	玫瑰、紫阳花、	2001. 二月. 15
23	狮子座	百合、向日葵、	2001. 二月. 15
24	处女座	樱粟花、波斯菊、	2001. 二月. 15
25	天秤座	牵牛花、海芋、	2001. 二月. 15

图 10-10 新闻的数据库

##### 2) 页面效果

首先来看看页面效果,见图 10-11。

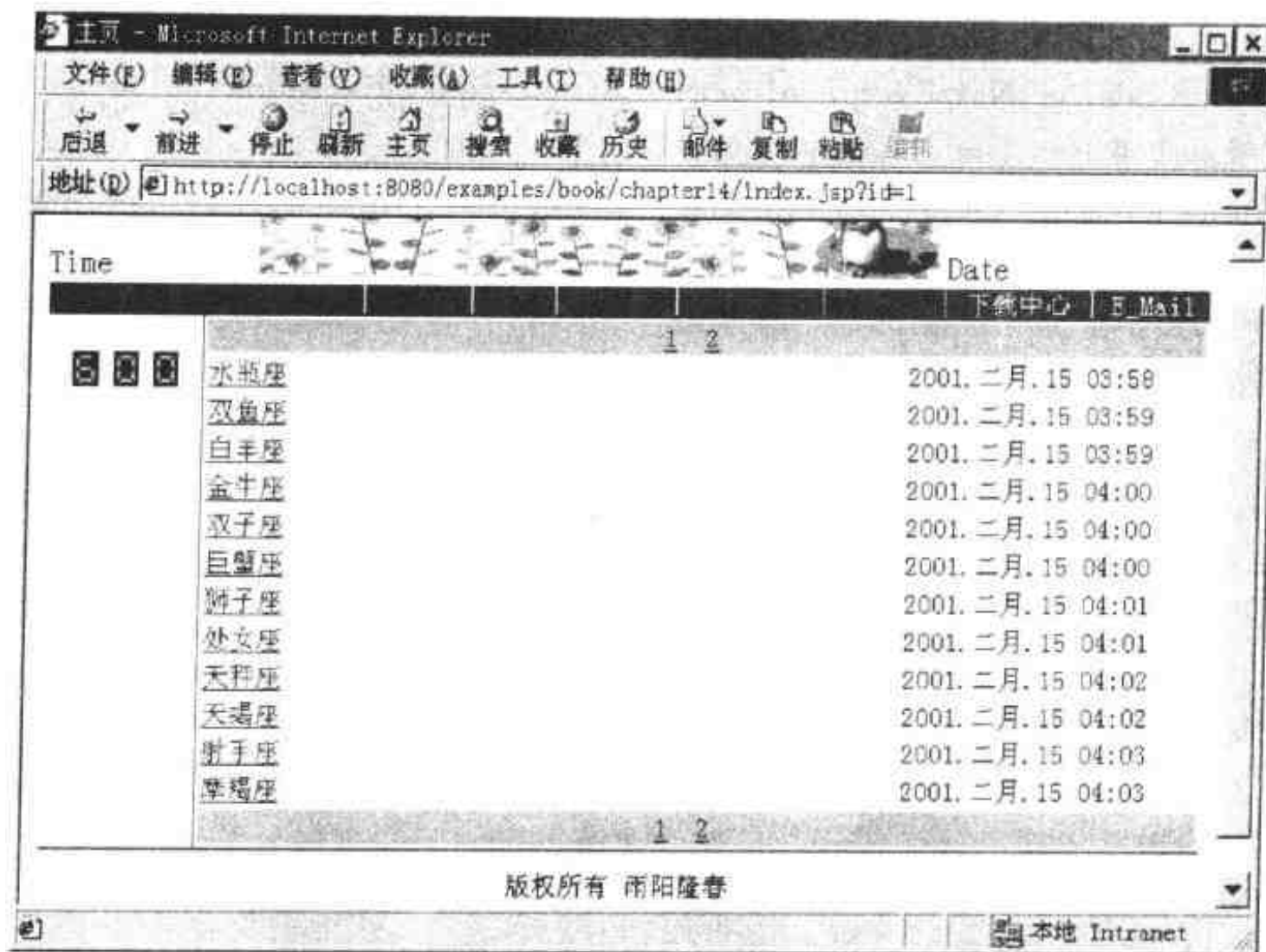


图 10-11 新闻页面效果

## 3) 页面源代码

显示图 10-11 的源代码如下：

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ page import="java.sql. *" %>
//页面显示新闻最大条数
<%! final int PAGESIZE = 12; %>
//总页数,页序号,开始显示新闻的序号
<%! int pagecount, pageorder, startdisplay; %>

<%
    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    Class.forName(driver).newInstance();
    String connectionURL = "jdbc:odbc:News";
    Connection con = DriverManager.getConnection(connectionURL);
    String sqlstring = "SELECT NewsID, Title, Stime FROM Culture";
    //建立一个 PreparedStatement 对象
    PreparedStatement pstmt = con.prepareStatement(
        sqlstring, ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
    ResultSet rs = pstmt.executeQuery();
```

```
rs.last();
//获得查询到的所有行数
int rowcount = rs.getRow();
//计算页数
pagecount = (rowcount + PAGESIZE - 1) / PAGESIZE;
out.println("<table width = 100 % border = 0 >");
out.println("<tr align = center bgcolor = #99cc66 > <td height = 10 colspan = 2 >");
for(int i = 0; i < pagecount; i++) {
    //建立分页序号超链接
    out.print("<a href = index.jsp? id = 1 & pageorder = " + (i + 1) + ">");
    out.print("<font size = 2 >");
    out.print(i + 1);
    out.print("</a >");
    out.print("&nbsp;&nbsp;&nbsp;");
    out.print("</font >");
}
out.println("</td > </tr >");
//获得请求显示的页序号
String strpageorder = request.getParameter("pageorder");
if(strpageorder == null || strpageorder.equals("")) {
    pageorder = 1;
}
else {
    pageorder = Integer.parseInt(strpageorder);
}
if(pageorder > pagecount) {
    pageorder = pagecount;
}
//计算得到开始显示的新闻序号
startdisplay = (pageorder - 1) * PAGESIZE + 1;
//定位到需要显示的第一条新闻
rs.absolute(startdisplay);
int j = 0;
//显示新闻
while(j < PAGESIZE && ! rs.isAfterLast()) {
    if(j % 2 == 0) {
        out.println("<tr > <td >");
    }
    else {
```



## 2. 新闻内容显示

与新闻标题显示类似,这里仅列出源代码。

```
<%@ page contentType="text/html; charset = gb2312" %>
<%@ page import="java.sql. *" %>

<%
    String driver="sun.jdbc.odbc.JdbcOdbcDriver";
    Class.forName(driver).newInstance();
    String connectionURL="jdbc:odbc:News";
    Connection con=DriverManager.getConnection(connectionURL);
    String sqlstring="SELECT Content FROM Culture WHERE NewsID=?";
    PreparedStatement pstmt=con.prepareStatement(sqlstring);

    String strnewsid=request.getParameter("newsid");
    if(strnewsid!=null&&!strnewsid.equals("")){
        pstmt.setString(1,strnewsid);
        ResultSet rs=pstmt.executeQuery();
        out.println("< table width=100% border=0>");
        out.println("< tr>< td>");
        rs.next();
        out.println(rs.getString(1));
        out.println("</td></tr>");
        out.println("</table>");
    }
%>
```

## 10.12.2 新闻发布

### 1. 新闻发布

#### 1) 页面效果及页面源代码

页面效果如图 10-12 所示。

这是一个非常简单的页面,请读者根据页面效果使用 Dreamweaver 实现这个页面,我们不再给出源代码。

#### 2) 后台处理源代码

后台处理源代码如下:

```
<%@ page contentType="text/html; charset = gb2312" %>
<%@ page import="java.util. *, java.sql. *, java.text. *" %>
```



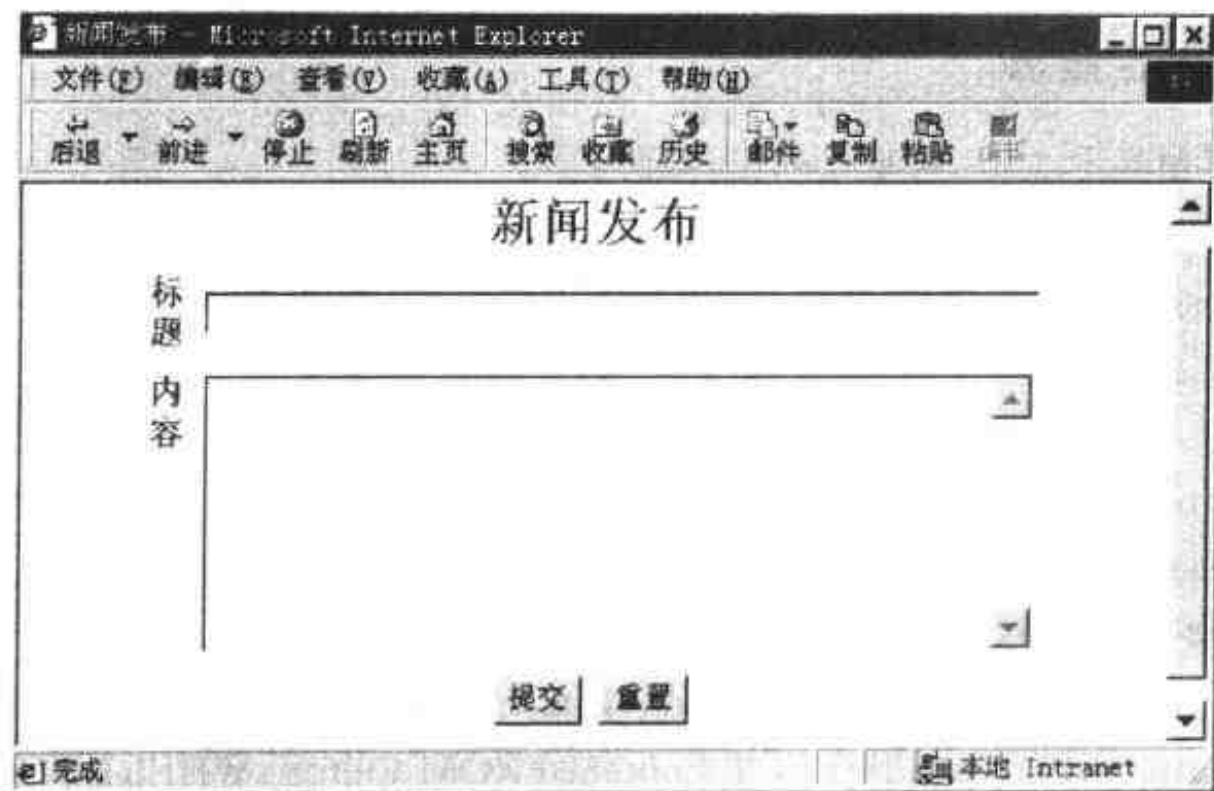


图 10-12 新闻发布页面

&lt; %

```

String strtitle = request.getParameter("txtTitle");
String strcontent = request.getParameter("txtContent");
if(strtitle! = null && ! strtitle.equals("") && strcontent! = null && ! strcon-
tent.equals("")){
    String charcodetitle = new String(strtitle.getBytes("ISO-8859-1"));
    String charcodecontent = new String(strcontent.getBytes("ISO-8859-1"));
    SimpleDateFormat df = new SimpleDateFormat(
        "yyyy.MM.dd hh:mm");
    String strtime = df.format(Calendar.getInstance().getTime());
    String insertvalue = "INSERT INTO Culture(Title,Content,Stime)
        VALUES(?,?,?)";
    String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    Class.forName(driver).newInstance();
    String connectionURL = "jdbc:odbc:News";
    Connection con = DriverManager.getConnection(connectionURL);
    PreparedStatement pstmt = con.prepareStatement(insertvalue);
    pstmt.setString(1,charcodetitle);
    pstmt.setString(2,charcodecontent);
    pstmt.setString(3,strtime);
    pstmt.execute();
    out.println("新闻发布成功");
}
else{

```

```
out.println("请输入合适的数据");
```

```
%>
```

### 3) 说明

有如下两点值得注意：

- (1) `SimpleDateFormat df = new SimpleDateFormat("yyyy.MM.dd hh:mm");`  
`String strtime = df.format(Calendar.getInstance().getTime());` 获得需要的时间格式。这一点还将在第 12 章 JSP 对 Applet 的集成中使用到。
- (2) 更新、插入操作应使用 `PreparedStatement.execute()` 方法。

## 10.13 本章小结

本章在本书中占据了非常重要的地位,从其占有的篇幅可见一斑,希望读者也能意识到这点。本章详细地讲述了使用 JDBC 操纵数据库的全过程,又着重叙述的 `Connection`、`Statement`、`PreparedStatement`、`ResultSet` 以及 `ResultSetMetaData` 的方法的使用。最后综合应用实现了新闻发布。

# 第 11 章 JSP 对 JavaBeans 的集成

用程序设计的观点来看,我们把 Java Beans 理解为组件的封装技术。JavaBeans 常用来封装商业的核心逻辑,它需要支持自检、属性等等,一句话,支持用户直接使用而不用关心内部是如何实现的。这大大增强了安全和快速原型能力。JSP 集成了 JavaBeans,使自身分离页面与逻辑的手段又多了一种。这是 JSP 一个极其重要的优点。

JavaBeans 是基于 Java 的组件技术,它提供了创建和使用以组件形式出现的 Java 类的方法。JavaBeans 通过封装属性和方法成为具有某种功能或者处理某个业务的对象,除了可以把我们以前写在 Scriptlet 中的代码放在 JavaBeans 的方法之中外,通过 JavaBeans 的方法调用可以处理几乎所有通过 Java 编程可以完成的工作,这本身也是 JSP 的极大优势所在。这种方式大大拓展了编程人员的可操纵范围,不用再像以往其它类似语言一样,因为找不到实现所需功能的组件而一筹莫展。可以说,JavaBeans 组件是目前具有功能强大和开发简单双重特点的最好的组件方式。由于 JavaBean 技术牵涉到很多方面的知识,本章并不打算介绍 JavaBeans 的方方面面(事实上也不可能),有兴趣的读者可以查阅相关资料,我们将把重点放在如何在 JSP 页面中使用 JavaBeans 组件。

## 11.1 JavaBeans 概述

JavaBeans 技术是 Sun 公司对 Java 技术的一项补充,弥补了 Java 缺少组件技术的缺点,JavaBeans 组件技术和其它组件技术一样,是一种围绕代码实施的高度可重用的软件技术,另一方面,由于它是基于 Java 的组件技术,因此 JavaBeans 也是跨平台的。

也许有的读者会问,JavaBeans 究竟是什么东西?实际上,JavaBeans 就是一个普通的 Java 类,如果读者了解 ActiveX 控件,可以先认为两者的功能是类似的。只是它具有以下一些特征:

(1)支持自检。所谓的自检就是提供一种方法来获得该组件的内部信息。JavaBean API 提供了很多用于自检的工具。通过自检,应用程序可以查询一个组件的功能,然后与这个组件进行相应的交互。自检是组件模型的一个关键,应为这样才能描述一个组件如何向应用程序以及其它组件描述自己的功能。

(2)支持定制。各个用户对组件的使用不同,要允许用户通过某些工具定制组件的外观和行为。

(3)支持持久性。组件经定制后要能保存状态。

(4)支持属性(properties)。各个组件要有自己的状态。Bean 的属性就是对象的属性,但提供了属性读取和设置 API 的支持。每个属性通常遵守简单的方法命名规则。这样,开发工具和最终用户可以按照这种特定的命名方式找出 Bean 所提供的属性,然后就可以查询属性值或改变属性值,对 Bean 进行操作。Bean 应该对属性值的改变作出及时

的反应。

(5) 支持事件。各个组件要从外界获得信息,也要向外界发送信息,这一点和对象之间通过消息通信类似。

(6) 支持分布式计算。Internet 的流行,使组件模型中的支持分布式计算变得更加重要了。

最后,我们给 JavaBeans 下一个明确的定义:JavaBeans 是客户用的平台独立的软件组件,开发则可以在软件构造器或服务器中直接进行可视化操作。换句话说,应用程序开发者可以通过支持 JavaBeans 的开发工具,直接使用现成的 JavaBeans,也可以在开发工具容器中,对组件进行必要的修改、测试而不必编写和编译程序。在 Java 模型中,组件可以修改或与其它组件组合以生成新组件或完整的应用程序。

### 11.1.1 JavaBeans 的属性

从面向对象的观点看,JavaBeans 的属性就是对象的属性,但 JavaBeans 提供了属性读取和设置 API 的支持。例如一个 book Bean 可以有 name、artist 和 price 属性,car Bean 可以有 speak 和 color 属性。属性是与 bean 的内部状态有关的命名的性质。这样,开发工具和最终用户可以按照这种特定的命名方式找出 Bean 所提供的属性,然后就可以查询属性值或改变属性值,对 JavaBeans 进行操作。另外,JavaBeans 应该对属性值的改变作出及时的反应。例如,如果改变 car 的 speed 属性,则 car 会加速;改变 car 的 color 属性,则 car 的颜色会改变。

JavaBeans 的属性按作用不同可分为四类:

(1) Simple 属性:Simple 属性表示一个有一对 get/set 方法的变量。如 setX() 和 getX() 方法。

(2) Bound 属性:Bound 属性表示当一个属性的值发生变化时,会与其它对象进行通讯。这时该属性会激活 PropertyChange 事件,用于告知其它对象该属性名,属性的原值,属性的新值。

(3) Constrained 属性:Constrained 属性表示了该属性值发生变化时,与该属性相关的对象是否允许这种变化。对象会抛出 PropertyVetoException 异常来阻止改变。

(4) Indexed 属性:Indexed 属性表示一个数组值。要取得数组的中值可以用 set/get 方法。

### 11.1.2 JavaBeans 的方法

JavaBeans 是 Java 对象,调用这个对象的方法是与其交互作用的唯一途径。JavaBeans 严格遵守面向对象的类设计逻辑,外部对象不允许访问其内部任何实例字段。也就是说,方法调用是和 JavaBeans 进行交流的唯一途径。但是有些 JavaBeans 采用调用实例方法的低级机制并不是操作和使用 JavaBeans 的主要途径。公开的 JavaBeans 方法在 JavaBeans 操作中降为次要地位,因为其它两个高级 JavaBeans 特性即属性和事件是与 JavaBeans 交互作用的更好的方式。因此 JavaBeans 可以提供要让客户使用的 public 方

法,但应当认识到,JavaBeans 设计规则希望看到绝大部分 JavaBeans 的功能反映在属性和事件中,而不是调用各个方法。

### 11.1.3 JavaBeans 的事件

JavaBeans 与其它软件组件交换信息的主要方式是发送和接收事件。这一点和对象之间通过消息通信类似。事件为 JavaBeans 提供了一种发送通知给其它组件的方法。就好比集成电路中的输入输出引脚。

JavaBeans 的事件是 `JavaEventObject` 扩展类的实例,用对象进行传递,常用的事件有:

```
java.util.EventObject
java.awt.AWTEvent
    java.awt.event.ActionEvent
        java.awt.event.AdjustmentEvent
            java.awt.event.InputMethodEvent
                java.awt.event.InvocationEvent
                    java.awt.event.ItemEvent
                        java.awt.event.TextEvent
```

## 11.2 在 JSP 页面中使用 JavaBeans

### 11.2.1 `<jsp:useBean... />` 标记

JSP 网页吸引人的地方之一就是能结合 JavaBeans 技术来扩充网页中程序的功能。在 JSP 页面中引入 JavaBeans 用 `<jsp:useBean>` 标记,它的语法如下:

```
<jsp:useBean id="name" scope=page | request | session | application
             class="className" | beanName="beanName" type="typeName">
```

id:为该 Bean 创建的实例的名字,以后可通过这个名字使用这个 Bean。

scope:该对象的作用范围,其默认为 page。

class:Bean 的实现文件,源文件经编译后的 .class 文件。

beanName:一个 bean 的名字,类似于 class 属性,该属性可以接收一个请求时的属性值作为它的值。

type:该对象的类型,可以是类本身的类型,或者是它的超类类型,也可以是类实现的接口的类型。

对于 `<jsp:useBean>` 标记,有以下几点说明:

- (1) id 属性是必须的。
- (2) scope 是可选的。
- (3) class 和 type 属性必具其一。

(4) class 和 beanName 属性只能有一个,或者一个没有。

我们举几个例子来说明该标记的用法。

例:

创建了一个类型为 MyBean.doSomething 的 JavaBeans 对象,并命名为 firstBean。该对象具有默认的 page 作用域。

```
<jsp:useBean id="firstBean" class="MyBean.doSomething"/>
```

例:

同样一个 Bean,而这时用 scope="application"告诉 Bean 整个应用程序保留信息。

```
<jsp:useBean id="firstBean" class="MyBean.doSomething" scope="application"/>
```

还有一种使用<jsp:useBean>标记的语法,即在<jsp:useBean>标记和</jsp:useBean>之间加入主体,在其中完成一些 bean 初始化,带主体的<jsp:useBean>标记语法如下:

```
<jsp:useBean 属性列>
```

```
body
```

```
</jsp:useBean>
```

主体一般完成对象的初始化工作,常常包括一些 Java 脚本和<jsp:setProperty>标记

例:

创建了一个名为 fourBean 的对象,该对象在当前 session 内有效,同时给 userName 属性赋值为"jim"。

```
<jsp:useBean id="fourBean" scope="session" class="MyBean.user">
```

```
<jsp:setProperty name="firstBean" property="username" value="jim">
```

```
</jsp:useBean>
```

## 11.2.2 <jsp:setProperty>标记

用<jsp:useBean>标记定位到 bean,是为了利用这个 bean 为我们做一些事情,通常的想法是首先利用某种方法把 JSP 页面中的数据传送到 bean 里,由 bean 提供的方法来处理数据,形成我们需要的结果,再用某种方法把结果传送回 JSP 页面,由页面决定把结果发送到何处。JSP 也是这么做的,它通过<jsp:setProperty>标记来设置 bean 的属性,在需要的时候用<jsp:getProperty>读出该属性。本小节讲解<jsp:setProperty>的语法。

<jsp:setProperty>语法如下:

```
<jsp:setProperty name="beanName" propertyExpression>
```

propertyExpression 有几种形式

(1) property="\*"

(2) property="propertyName"

(3) property="propertyName" param="parameterName"

(4) property="propertyName" value=string|expression,

其中 name 属性就是<jsp:useBean>标记中的 id 属性值,也就是我们用<jsp:useBean>创建的 Javabeans 实例名,该 Javabeans 实例必须包含我们想设置值的属性,就是

property 属性指定的名字,如果 property="\*",那么属性不会按 html 的表单顺序排序。

#### 例 11-1

```
package test;

public class helloWorld {
    public String name = "My first bean";
    public String getHi(){
        return "Hello from " + name;
    }
}
```

helloWorld.java 编辑好后,用 JDK 的 javac 命令编译 helloWorld.java。

编译成功就表示建立了一个 JavaBean,可将它放在任意的类路径下。下面看如何在 JSP 中使用这个 JavaBean。用文本编辑器创建 hi-bean.jsp,源代码如下。

```
<html>
<head>
<title>JavaBean 试验</title>
</head>
<body>
<jsp:useBean id="helloBean" scope="session" class="test.helloWorld" />
<%= helloBean.getHi() %>
<hr>
<%=
    helloBean.name = "JSP";
    out.print(helloBean.getHi());
%>
</body>
</html>
```

运行后如图 11-1 所示。

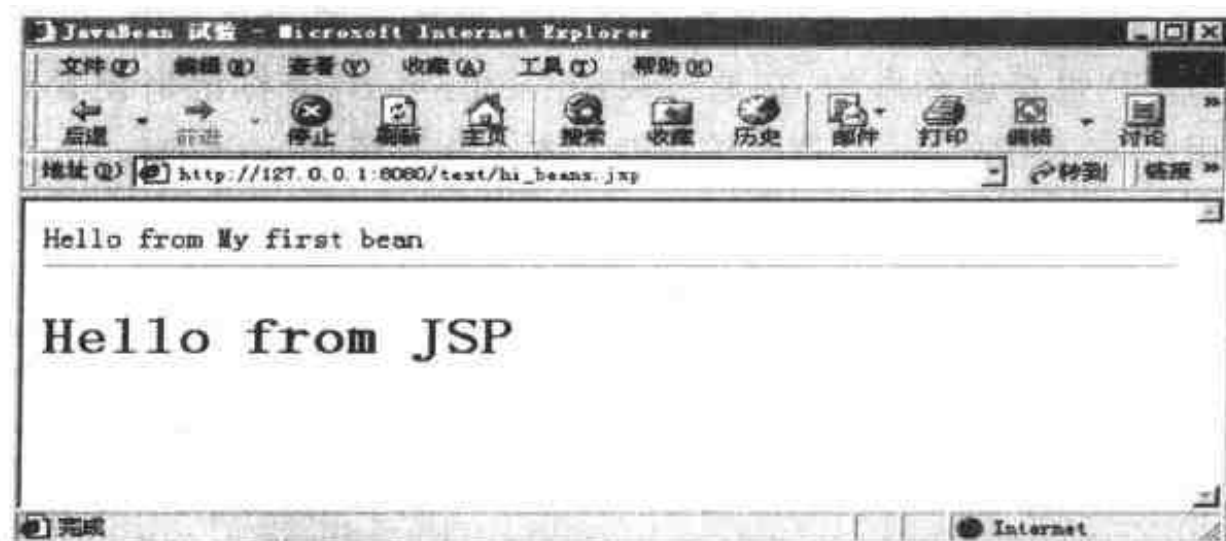


图 11-1 hi-bean.jsp 运行结果

读者可从这个简单的例子中看出如何设置、获取 JavaBean 属性,以及怎样调用 JavaBean 方法。在 JSP 页面中,通过<jsp:useBean ... />语法使用 JavaBeans,并且必须给





“hello.htm”执行结果如图 11-2 所示。

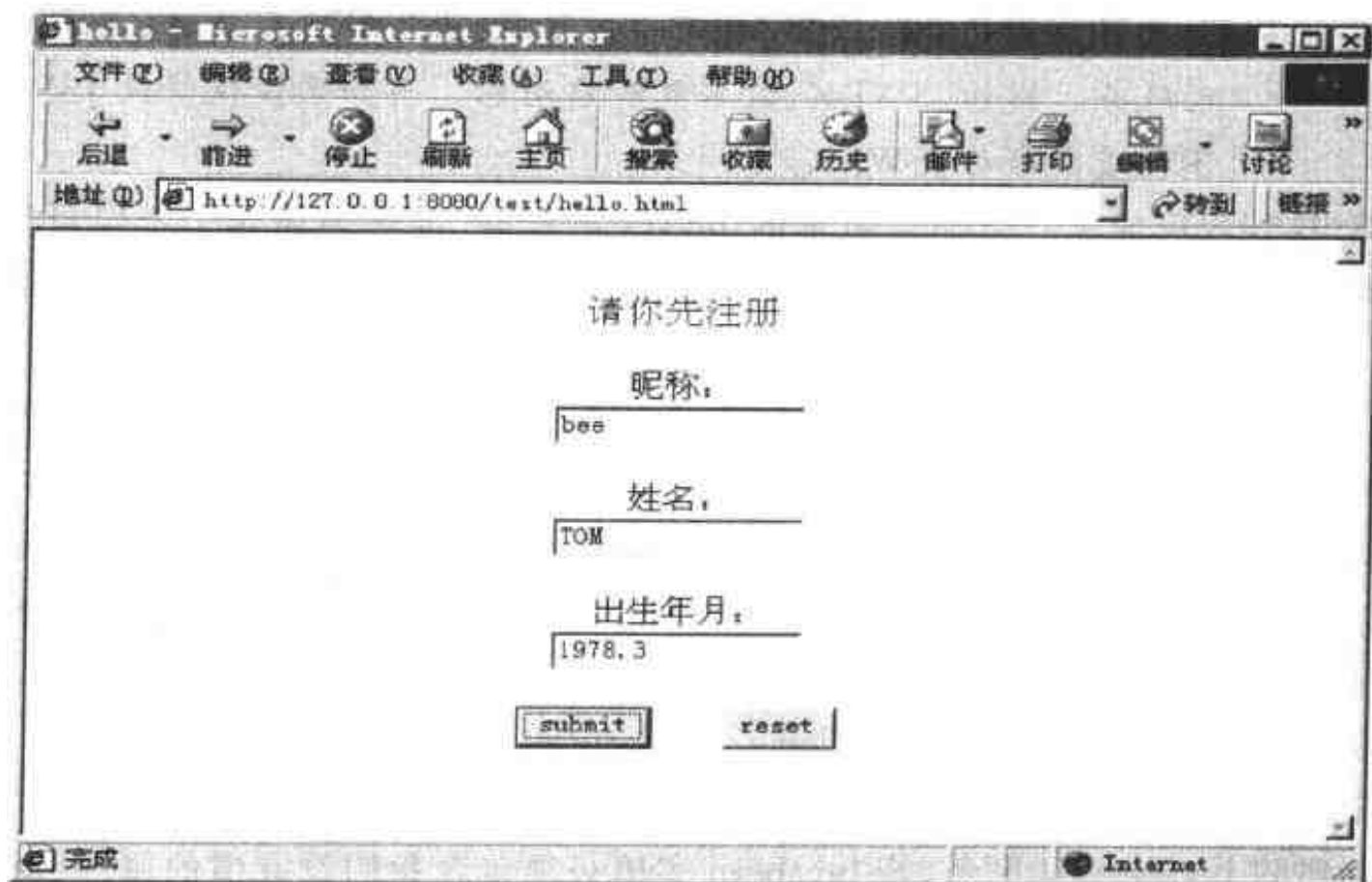


图 11-2 hello.html 界面图

通过这个 hello.html 表单共提交了 4 个参数,参数名及参数值分别为“rname = 'Tom'”、“birth = '1978.3'”、“fname = 'bee'”和“submit = 'submit'”。接下来我们定义一个具有这 4 个属性的 Bean,源文件为“getPara.java”,如下所示:

```
package test;

public class getPara{
    String rname;
    String fname;
    String birth;
    String submit;
    public void getPara(){
    }

    public void setRname(String _ rname){
        rname = _ rname;
    }

    public String getRname(){
        return rname;
    }

    public void setFname(String _ fname){
        fname = _ fname;
```

```
    }  
    public String getFname() {  
        return fname;  
    }  
  
    public void setBirth(String _ birth) {  
        birth = _ birth;  
    }  
  
    public String getBirth() {  
        return birth;  
    }  
  
    public void setSubmit(String _ submit) {  
        submit = _ submit;  
    }  
  
    public String getSubmit() {  
        return submit;  
    }  
}
```

编译通过后再写一个使用此 Bean 的 JSP 文件“welcome.jsp”，如下所示：

```
<html>  
<%@page contentType="text/html; charset = gb2312"%>  
<jsp:useBean id="getp" class="test.getPara"/>  
<head>  
<title>welcome</title>  
</head>  
<body>  
<h2>欢迎您！</h2><br>  
<jsp:setProperty name="getp" property="*" />  
姓名：  
<jsp:getProperty name="getp" property="rname"/><br>  
昵称：  
<jsp:getProperty name="getp" property="fname"/><br>  
出生年月：  
<jsp:getProperty name="getp" property="birth"/><br>  
submit：
```

```
<jsp:getProperty name="getp" property="submit"/> <br>  
</body>  
</html>
```

当 hello.html 提交后, welcome.jsp 文件将对提交的信息进行处理, 处理结果如图 11-3 所示。

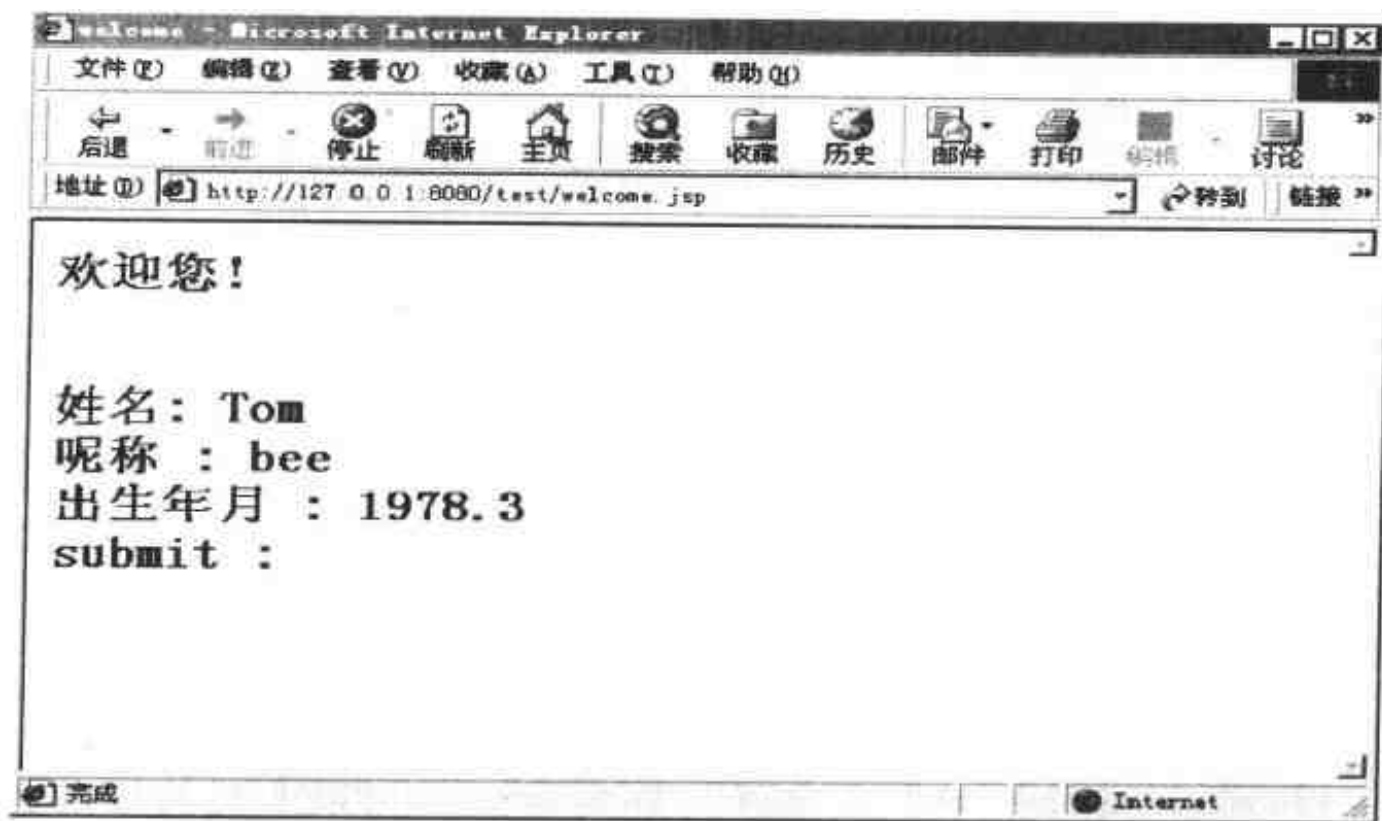


图 11-3 welcome.jsp 处理结果

### 例 11-3

本例中创建了一个名为 bookBean 的 JavaBeans。它有 4 个属性, 即 book(书名)、price(价格)、num(数量)和 countprice(总价)。4 个 set 方法分别用来设置这 4 个属性, 4 个 get 方法则用于提取这 4 个属性。在实际应用中, 这种 JavaBeans 一般应当与购物车相联系, 此处我们简化了这个过程。下面是这个 JavaBeans 的代码清单。

```
package test;  
  
public class bookBean{  
    String book;  
    int num;  
    double price;  
    double countprice;  
    public bookBean(){  
        book = "jsp";  
        num = 1;  
        countprice = 50;  
        price = 50;  
    }  
    public void setbook(String bookName){  
        book = bookName;  
    }  
}
```

```
public String getbook(){
    return(book);
}

public void setnum(int booknum){
    num = booknum;
}

public int getnum(){
    return(num);
}

public void setprice(double pricevalue){
    price = pricevalue;
}

public double getprice(){
    return(price);
}

public void setcountprice(){
    countprice = price * num;
}

public double getcountprice(){
    return(countprice);
}
}
```

前面讲过在 JSP 页面中应用 JavaBeans 要用到 `<jsp:useBean>` 标记。由于具体使用的 JSP 引擎的不同,在何处配置以及如何配置 JavaBeans 的方法也可能略有不同。本文将这个 JavaBeans 的 .class 文件放在 `\jswdk-1.0.1\webpages\WEB-INF\jsp\beans\` 目录下,这里的 test 是一个专门存放 Bean 的目录。应用上述 Bean 的 bookbean.jsp 源代码如下。

```
<html>
<body>
<%@page contentType="text/html; charset = gb2312"%>
<jsp:useBean id="bookbean" scope="application" class="test.bookBean"/>
<head>
<title>bookbean</title>
```

```
</head>
<%
    bookbean.setbook("php");
    bookbean.setprice(40);
    bookbean.setnum(3);
    bookbean.setcountprice();
%>
<h3>方法 1:</h3>
书名:<% = bookbean.getbook() %><br>
数量:<% = bookbean.getnum() %><br>
价格:<% = bookbean.getprice() %><br>
总价:<% = bookbean.getcountprice() %><br>
<%
    bookbean.setbook("Delphi");
    bookbean.setprice(40);
    bookbean.setnum(5);
    bookbean.setcountprice();
%>
<h3>方法 2:</h3>
书名:<jsp:getProperty name="bookbean" property="book"/><br>
数量:<jsp:getProperty name="bookbean" property="num"/><br>
价格:<jsp:getProperty name="bookbean" property="price"/><br>
总价:<jsp:getProperty name="bookbean" property="countprice"/><br>
</body>
</html>
```

运行效果如图 11-4 所示。

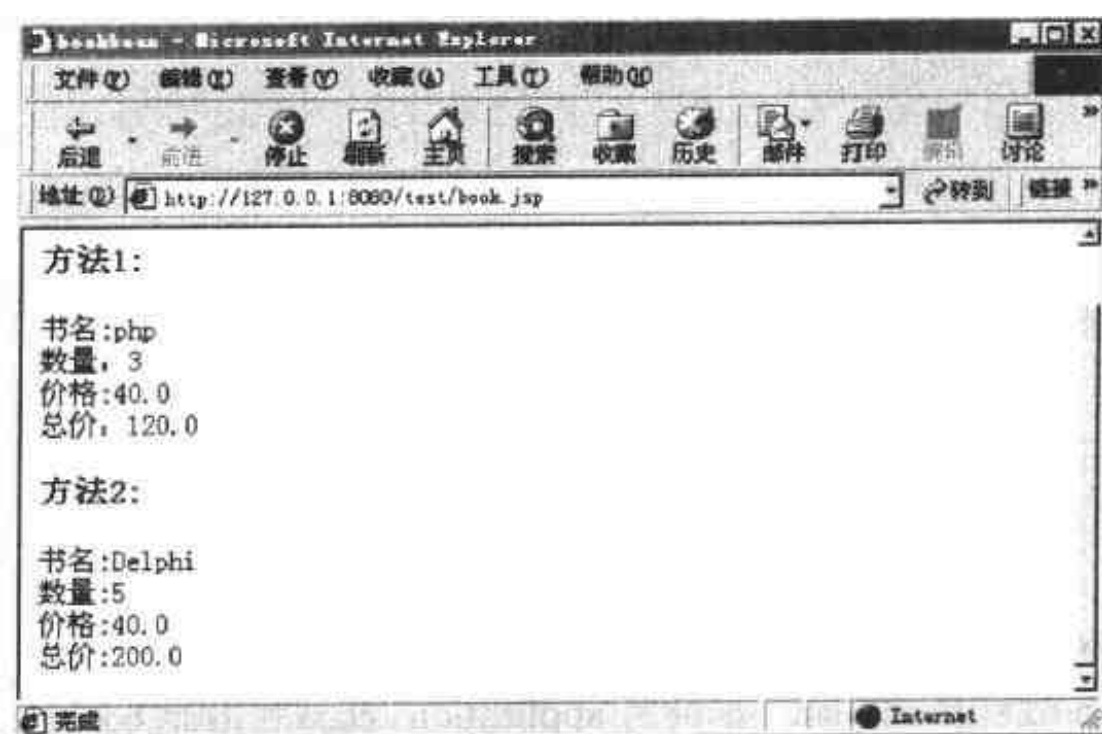


图 11-4 book.jsp 处理结果图

在<jsp:useBean>标记内定义了几个属性,其中 id 是整个 JSP 页面内该 JavaBeans 的标识,scope 属性定义了该 JavaBeans 的生命周期为 application,class 属性说明了该 JavaBeans 的类文件(从包名开始)这个 JSP 页面不仅使用了 JavaBeans 的 set 和 get 方法设置和提取属性值,还用到了提取 JavaBeans 属性值的第二种方法,即使用<jsp:getProperty>标记。<jsp:getProperty>中的 name 属性即为<jsp:useBean>中定义的 JavaBeans 的 id,它的 property 属性指定的是目标属性的名字。

### 11.2.4 编写自己的 Bean

前面谈到使用 Bean 并非只是利用它来保存和提供参数,使用 Bean 的根本目的还在于利用它来实现一些独立的功能,正如前几章在讲 servlet 和内部对象时,提到的数据库操作和购物车都可以通过编写相应的 Bean 来实现。在 JSP 文件中除了使用 Bean 方法的语句外,不需要其它的代码。因此 JSP 文件的功能完全取决于 Bean 的方法执行了什么操作。除了可以把我们以前写在脚本代码放在 Bean 的方法之中外,通过 Bean 的方法调用可以处理几乎所有通过 Java 编程可以完成的工作,这本身也是 JSP 的极大优势所在。所以,充分利用 Bean 的优势是开发 JSP 应用程序是应该始终关注的焦点。下面介绍几个应用实例。

### 11.2.5 通用数据库 Bean

#### 例 11-4

在前一章讲述了如在 jsp 页面中进行数据库操作,在这一章我们可以将数据库的一些常用的功能如:连接数据库,打开数据库,执行 sql 语句以及关闭数据库等用 Bean 来实现。

```
package test;
import java.sql. * ;
public class mysql{
    String driver="sun.jdbc.odbc.JdbcOdbcDriver";
    String connectionURL="jdbc:odbc:char";
    Connection con=null;
    ResultSet rs=null;
    Statement stmt=null;
    public mysql() {
        try{
            Class.forName(driver);
        }
        catch(java.lang.ClassNotFoundException e){
            System.err.println("mysql(): " + e.getMessage());
        }
    }
}
```

```
    }

    public void Insert(String sqlstring) {
        try {
            con = DriverManager.getConnection(connectionURL);
            stmt = con.createStatement();
            stmt.executeUpdate(sqlstring);
        }
        catch (SQLException ex) {
            System.err.println("mysql.executeUpdate:" + ex.getMessage());
        }
    }

    public void Delete(String sqlstring) {
        try {
            con = DriverManager.getConnection(connectionURL);
            stmt = con.createStatement();
            stmt.executeUpdate(sqlstring);
        }
        catch (SQLException ex) {
            System.err.println("sql _ data.Delete:" + ex.getMessage());
        }
    }

    public ResultSet executeQuery(String sqlstring) {
        try {
            con = DriverManager.getConnection(connectionURL);
            stmt = con.createStatement();
            rs = stmt.executeQuery(sqlstring);
        }
        catch (SQLException ex) {
            System.err.println("sql _ data.execQuery:" + ex.getMessage());
        }
        return rs;
    }
}
```

在 mysql Bean 中我们定义对数据库的查询、插入、删除操作。这样只需要在 jsp 页面中声明该 Bean 就可以直接使用了。用 Bean 来实现数据库的操作另一个好处在于保证了数据的安全性。

### 11.2.6 购物车 Bean

在第 6 章中用 Session 对象实现了一个购物车,在这里将用以一个 Bean 来完成该功能,具体方法如下:

首先,定义一个 shopCart 类,具体代码如下:

```
package test;

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

public class shopCart{
    Hashtable h = new Hashtable();
    Vector v = new Vector();
    String item = null;
    Integer num = new Integer(1);
    String submit = null;
    int sortnum = 0;

    private void addItem(String name, Integer num) {
        h.put(name, num);
        v.addElement(name);
    }

    private void removeItem(String name){
        h.remove(name);
        v.removeElement(name);
    }

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

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

    public String[] getItems(){
```



```
        String[] s = new String[v.size()];
        v.copyInto(s);
        return s;
    }

    public String[] getNum(){
        String[] n = new String[h.size()];
        for (int i = 0; i < h.size(); i++) {
            n[i] = h.get(v.elementAt(i).toString()).toString();
        }
        return n;
    }

    public void processRequest(HttpServletRequest request) {
        if (submit == null)
            addItem(item, num);
        if (submit.equals("add")) {
            if (h.containsKey(item)) {
                Integer n_temnum = (Integer)h.get(item);
                int temnum = n_temnum.intValue();
                temnum = temnum + 1;
                n_temnum = new Integer(temnum);
                h.put(item, n_temnum);
            }
            else
                addItem(item, num);
        }
        else if (submit.equals("remove")) {
            if (h.containsKey(item)) {
                Integer n_temnum = (Integer)h.get(item);
                int temnum = n_temnum.intValue();
                temnum = temnum - 1;
                if (temnum <= 0) removeItem(item);
                n_temnum = new Integer(temnum);
                h.put(item, n_temnum);
            }
        }
        reset();
    }
}
```

```

private void reset() {
    submit = null;
    item = null;
    num = new Integer(1);
}
}

```

在该 Bean 中我们用一个向量来保存货物名,用一个哈希表来保存与该货物名相对应的数量,因为在哈希表中具有一一对应的关系。具体地说明见源程序中的注释。下面将定义一个使用该 Bean 的 jsp 页面。

```

<html>
<jsp:useBean id="cart" scope="session" class="test.shopCart"/>
<jsp:setProperty name="cart" property=" * "/>
<%
    cart.processRequest(request);
%>
<FONT size = 5 COLOR = "# CC0000">
<br> You have the following items in your cart:<p>
<%
    String[] items = cart.getItems();
    String[] nums = cart.getNum();
    for(int i=0; i<items.length;i++) {
%>
<p><font color='black'>Goodsname:</font><%= items[i] %>
<font color='black'>Number:</font><%= nums[i] %><br>
<%= } %>
</font><hr>
<%= @ include file = "shopcart.html" %>
</html>

```

shopcart.html 如下:

```

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

```

```
<SELECT NAME="item">
<OPTION>Beavis & Butt-head Video collection
<OPTION>X-files movie
<OPTION>Twin peaks tapes
<OPTION>NIN CD
<OPTION>JSP Book
<OPTION>Concert tickets
<OPTION>Love life
<OPTION>Switch blade
<OPTION>Rex, Rugs & Rock n' Roll
</SELECT>
<br><br>
<INPUT TYPE=submit name="submit" size="5" value="add">
<INPUT TYPE=submit name="submit" size="5" value="remove">
</form>
</font>
</body>
</html>
```

运行效果如图 11-5 所示。

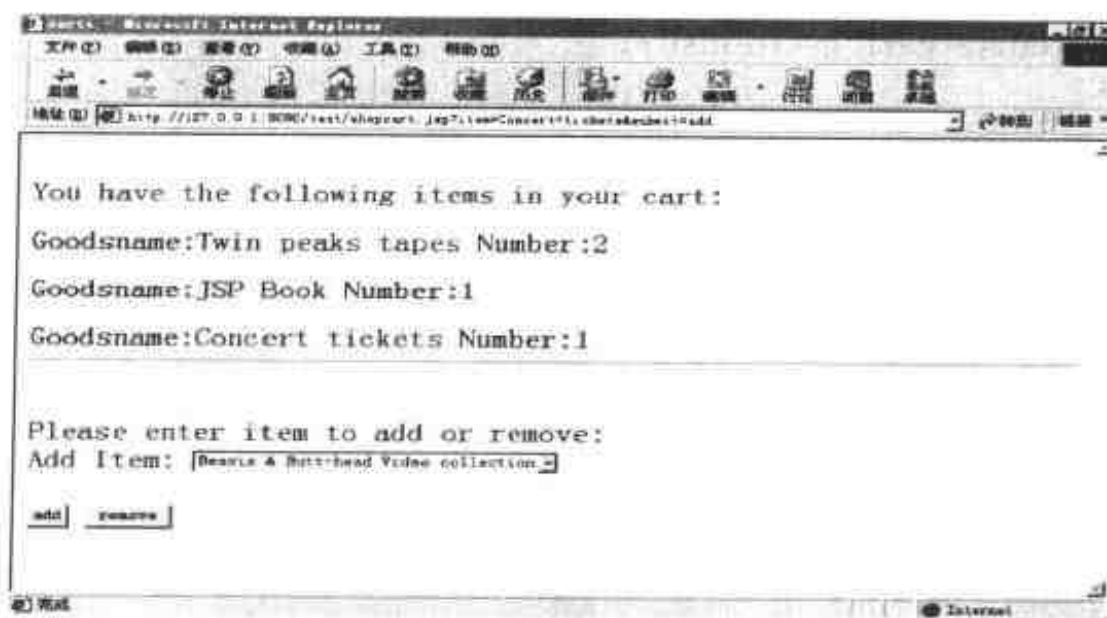


图 11-5 购物车示例图

## 11.3 本章小结

本章向读者介绍了 JavaBeans 的基本概念,着重讲述了如何在 JSP 页面中使用 JavaBeans。在最后给出了常用的 Bean 开发实例。通过讲解与示例,希望大家能熟练掌握 JavaBeans 使用方法。事实证明,Java Servlet 是一种开发 Web 应用的理想构架。JSP 以 Servlet 技术为基础,又在许多方面作了改进。利用跨平台运行的 JavaBean 组件,一次编写,到处运行。

# 第 12 章 JSP 对 Applet 的集成

Applet 是 Java 向 Internet 进军的第一个产物,正是因为 Applet 的巨大成功,才有了 Java 今日之崇高地位,才有了基于 Java 技术的繁荣昌盛,才有了“Java 就是计算机网络语言”的广泛流传。可以这么说,Applet 为打造今日今时之 Java 盛世立下了汗马功劳。Applet 确曾是重要的客户端交互手段,即使在服务端技术雄图霸业的今天,Applet 仍有用武之地。因为服务端技术与客户端技术是相辅相承的,而 Applet 是客户端技术的重要一员,自然不甘寂寞,而且在某些领域非其莫属,例如掌上环境等。

这里讲述 Applet 这种客户端技术。JSP 通过 `<jsp:plugin>` 标记集成了 Applet,Applet 仍然被下载到客户端运行。`<jsp:plugin>` 标记还集成了客户端 JavaBeans 组件。鉴于 Applet 技术的普及,本章主要讲述 `<jsp:plugin>` 标记的使用。

学习本章最好拥有如下知识:

理解“多线程”概念。

Applet 的基本知识。如 Applet 的生命周期、java.awt 包中的 API 等。

## 12.1 `<jsp:plugin>` 行为

plugin 行为使 JSP 页面作者产生的 HTML 文档能够包含依赖于客户端浏览器的结构,如 OBJECT 或 EMBED,这将导致 Java Plugin 软件(Applet 或 JavaBeans)被下载到客户端并且在客户端并发执行。

为适合不同的浏览器,`<jsp:plugin>` 标记被写入响应输出流的时候,可被 `<object>` 或 `<embed>` 标记代替,`<jsp:plugin>` 标记的属性为元素的表示提供配置数据。

`<jsp:param>` 元素指定 Applet 或 JavaBeans 组件的参数。参见第 5 章的叙述。

如果插件不能执行(因为浏览器不支持 OBJECT 和 EMBED,或者因其它问题),此时,`<jsp:fallback>` 元素指定客户端浏览器可使用的內容。如果插件可以执行,但是 Applet 或 JavaBeans 组件找不到,那么将显示一个插件的特殊的消息给用户,大多数弹出一个窗口报告 `ClassNotFoundException` 异常。

尽管有些提供商为他们客户的利益选择了包含插件,但是实际插件代码不需要绑定到 JSP 容器,可使用 Sun 的插件地址代替。

语法:

```
<jsp:plugin type = "bean|applet"
code = "objectCode"
codebase = "objectCodebase"
{ align = "alignment" }
{ archive = "archiveList" }
```

```

{ height="height" |
{ hspace="hspace" |
{ jreversion="jreversion" |
{ name="componentName" |
{ width="width" |
{ nspluginurl="url" |
{ iepluginurl="url" | >
{ <jsp:params>
{ <jsp:param name="paramName" value="paramValue" /> | +
</jsp:params> |
{ <jsp:fallback> arbitrary_text </jsp:fallback> |
</jsp:plugin>

```

type:指定组件的类型,JavaBeans 或 Applet。

code:将要执行的 class 文件名称,这个文件必须存在于 codebase 指定的目录中。

codebase:将要执行的 class 文件所在的目录。

align:Bean 或 Applet 实例显示时的对齐方式,例如“left”为左对齐。

archive:预装 class 的路径,多个路径使用逗号分开。

height:Bean 或 Applet 实例显示的高度。

hspace:左页边距,单位为 Pixels。

jreversion:Bean 或 Applet 运行需要的 JRE 版本号,默认是 1.2。

name:Bean 或 Applet 实例的名称。

vspace:顶页边距,单位为 Pixels。

title:Bean 或 Applet 实例的标题。

width:Bean 或 Applet 实例显示的宽度。

nspluginurl:可以下载 Netscape Navigator 的 JRE 插件的 URL。默认由实现定义。

iepluginurl:可以下载 IE 的 JRE 插件的 URL,默认由实现定义。

例:

```

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

```

## 12.2 时钟、日期 Applet 的实现

首先让我们来实现一个简单的时钟 Applet。

### 12.1.1 时钟 Applet

#### 1. 页面效果

页面效果如图 12-1 所示。

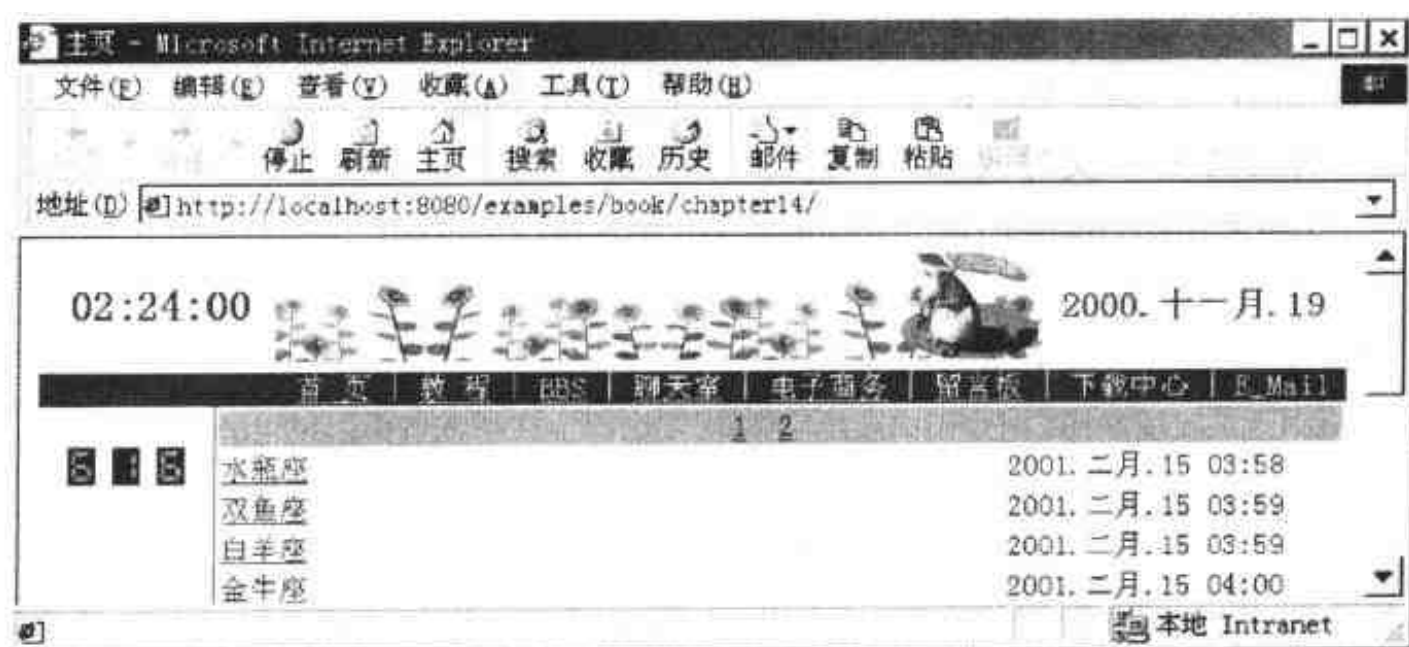


图 12-1 简单的时钟、日期 Applet

#### 2. 时钟源代码

这是一个 .java 文件, 本书中名为 Clock.java (光盘 chapter14/timedate/class 目录下), 需要编译成类文件。源代码如下:

```
import java.util. * ;  
import java.awt. * ;  
import java.applet. * ;  
import java.text. * ;
```

```
public class Clock extends Applet implements Runnable {  
    //显示时钟的线程  
    Thread timer;  
    //格式化时间显示  
    SimpleDateFormat formatter;  
    //时间显示字体  
    Font timeStrFont;  
    //时间显示颜色
```

```
Color timeStrColor;
//当前完整格式的日期时间
Date currentDate;
//当前时间(HH:mm:ss 格式)
String currentTime;
//上一次时间(HH:mm:ss 格式)
String lastTime;

public void init() {
    //获得 SimpleDateFormat 对象,用于格式时间
    formatter = new SimpleDateFormat("HH:mm:ss");
    //获得当前系统时间
    currentDate = Calendar.getInstance().getTime();
    //设置上一次时间
    lastTime = formatter.format(currentDate);
    //获得字体对象
    timeStrFont = new Font("Arial", Font.BOLD, 24);
try {
    //设置背景
    setBackground(new Color(Integer.parseInt(getParameter("bgcolor"), 16)));
    |
catch(Exception E) { }
try {
    //获得用于显示时间的颜色
    timeStrColor = new Color(Integer.parseInt(getParameter("fgcolor"), 16));
    |
catch(Exception E) { |
|
public void paint(Graphics g) {
    formatter = new SimpleDateFormat("HH:mm:ss");
    currentDate = Calendar.getInstance().getTime();
    currentTime = formatter.format(currentDate);
    //设置字体
    g.setFont(timeStrFont);
    //清除上一次显示
    g.setColor(getBackground());
    g.drawString(lastTime, 17, 38);
    //显示当前时间
    g.setColor(timeStrColor);
```

```

        g.drawString(currentTime, 17, 38);
        //置上一次时间为当前值
        lastTime = currentTime;
        currentDate = null;
    }

    public void start() {
        timer = new Thread(this);
        timer.start();
    }

    public void stop() {
        timer = null;
    }

    public void run() {
        Thread me = Thread.currentThread();
        while (timer == me) {
            try {
                Thread.currentThread().sleep(100);
            }
            catch (InterruptedException e) {
            }
            repaint();
        }
    }

    public void update(Graphics g) {
        paint(g);
    }
}

```

### 3. 调用文件源代码

如何调用文件源代码,并传递参数给它,请看下面的源代码:

```

<applet code="Clock.class" width=130 height=62
codebase="/examples/book/chapter14/timedate/class/">
<param name=bgcolor value="ffffcc">
<param name=fgcolor value="ff0000">
</applet>

```



需要注意的是使用<jsp:plugin……>标记未必有效。如果无效可以考虑以<applet></applet>标记代替。

## 12.2.2 日期 Applet

### 1. 页面效果

页面效果如图 12-1 所示。

### 2. 源代码

这也是一个 .java 文件,本书中名为 Dater.java(光盘 chapter14/timedate/class 目录下),需要编译成类文件。源代码如下:

```
import java.util. * ;
import java.awt. * ;
import java.applet. * ;
import java.text. * ;

public class Dater extends Applet implements Runnable {
    //显示日期的线程
    Thread dater;
    //格式化日期显示
    SimpleDateFormat formatter;
    //日期显示字体
    Font timeStrFont;
    //日期显示颜色
    Color timeStrColor;
    //当前完整格式的日期时间
    Date currentDate;
    //当前日期(yyyyy.MMMMM.dd 格式)
    String currentDateFormat;
    //上一次日期(yyyyy.MMMMM.dd 格式)
    String lastDate;

    public void init() {
        //获得 SimpleDateFormat 对象,用于格式日期
        formatter = new SimpleDateFormat ("yyyyy.MMMMM.dd");
        //获得当前系统时间
        currentDate = Calendar.getInstance().getTime();
        //设置上一次日期
```

```
        lastDate = formatter.format(currentDate);
        //获得字体对象
        timeStrFont = new Font("Arial", Font.BOLD, 20);
    try{
        //设置背景
        setBackground(new Color(Integer.parseInt(getParameter("bgcolor"),16)));
    }
    catch(Exception E){ }
    try{
        //获得用于显示日期的颜色
        timeStrColor = new Color(Integer.parseInt(getParameter("fgcolor"),16));
    }
    catch(Exception E){ }
}

public void paint(Graphics g) {
    formatter = new SimpleDateFormat("yyyy.MM.dd");
    currentDate = Calendar.getInstance().getTime();
    currentDateFormat = formatter.format(currentDate);
    //设置字体
    g.setFont(timeStrFont);
    //清除上一次显示
    g.setColor(getBackground());
    g.drawString(lastDate,7,38);
    //显示当前日期
    g.setColor(timeStrColor);
    g.drawString(currentDateFormat, 7, 38);
    //置上一次日期为当前值
    lastDate = currentDateFormat;
    currentDate = null;
}

public void start(){
    dater = new Thread(this);
    dater.start();
}

public void stop(){
    dater = null;
}
```

```
public void run(){
    Thread me = Thread.currentThread();
    while (dater == me){
        try{
            Thread.currentThread().sleep(100);
        }
        catch(InterruptedException e){
        }
        repaint();
    }
}

public void update(Graphics g){
    paint(g);
}

}
```

### 3. 调用文件源代码

调用文件源代码与调用时钟源代码是相似的,因此不再给出源代码。

## 12.3 本章小结

本章主要讲述了<jsp:plugin……/>标记的使用。然后用 Applet 技术实现了时钟和日期服务。为我们的网站增加一些活泼的色彩。

# 第 13 章 JSP 对 XML 的集成

本章讲解 JSP 集成 XML 方面的问题。

学习本章最好对 XML 语言有全面了解,包括一般性概念,文档类型定义 DTD。虽然我们会给出几个基本的概念,但仍需要你了解得更多一些。

## 13.1 JSP 页面的 XML 语法

所有 JSP 页面都有一个等价的 XML 文档,也可直接书写为与它等价的 XML 文档形式。在介绍 JSP 文档的 XML 等价物前,先了解几个最基本的概念,也许对大多数读者是有必要的。

### 13.1.1 XML 的几个基本概念

XML 为“Extensible Markup Language”的缩写,即“可扩展标记语言”。它不像 HTML 那样有固定格式,它使得 SGML 在 Web 上应用自如。XML 在许多方面增强了 Java,它们是完美的一对。XML 为 Java 提供数据,Java 为 XML 提供易于使用的代码。彼此相辅相承。

SGML 是“Standard Generalized Markup Language”的缩写,即“通用标识语言标准”。它是国际上定义电子文件结构和内容描述的标准。HTML 是 SGML 在网络上的一个特殊应用。XML 是 SGML 的一个子集,它省略了一些 SGML 中复杂和不常用的部分。

DTD 是“Document Type Declaration”的缩写,即“文档类型定义”。DTD 为 XML 文档定义应该包含或者可以包含的元素。

例:

下面是一个有效的 XML 文档。如果一个 XML 文档与一个文档类型定义(DTD)相关联,并且该 XML 文档符合该 DTD 定义的各种规格,该 XML 文档就是有效的。

```
<? xml version = "1.0"? >
<! DOCTYPE DOCUMENT[
<! ELEMENT CUSTOMER(NAME, ORDERS)>
<! ELEMENT NAME(LASTNAME, FIRSTNAME)>
<! ELEMENT LASTNAME(#PCDATA)>
<! ELEMENT FIRSTNAME(#PCDATA)>
<! ELEMENT ORDERS(ITEM)* >
<! ELEMENT ITEM(PRODUCT, NUMBER, PRICE)>
<! ELEMENT PRODUCT(#PCDATA)>
```

```
<! ELEMENT NUMBER( # PCDATA)>
<! ELEMENT PRICE( # PCDATA)>
]>
<DOCUMENT>
  <CUSTOMER>
    <NAME>
      <LASTNAME>only</LASTNAME>
      <FIRSTNAME>you</FIRSTNAME>
    </NAME>
  <ORDERS>
    <ITEM>
      <PRODUCT>Banana</PRODUCT>
      <NUMBER>3</NUMBER>
      <PRICE>1.6</PRICE>
    </ITEM>
  </ORDERS>
</CUSTOMER>
</DOCUMENT>
```

### 13.1.2 JSP 页面的 XML 语法

所有 JSP 页面都有一个等价的 XML 文档。这个等价的 XML 文档是 JSP 页面在翻译阶段的视图。JSP 页面也可直接书写为与它等价的 XML 文档形式。JSP1.0 和 JSP1.1 规定,XML 文档自身不能提交给 JSP 容器处理,但 JSP1.2 可以。

JSP 页面(以 JSP 或 XML 任意一种语法形式表达)可通过指令包含其它的以任意一种语法形式书写的 JSP 页面。

本小节给 JSP 页面定义 XML 语法和这种语法书写页面的语义。本小节也定义 JSP 语法书写的 JSP 页面和 XML 语法书写的等价 JSP 页面之间的映射。

用 XML 语法书写 JSP 页面的方法如下:

- (1)确认 JSP 页面。
- (2)使用 XML 工具操纵 JSP 页面。
- (3)用 XML 转换并从其它的文本表示生成页面,例如 XSLT。
- (4)由序列化对象产生页面。

JSP1.2 中通过标记库验证器与标记库相连的机制直接支持验证。验证器作用在 XML 语法书写的 JSP 页面上。以 XML 语法书写的 JSP 页面能提交给 JSP 容器处理。

无论以那种语法书写的 JSP 页面通过指令可以包含任意一种语法形式的 JSP 页面。例如一个单元里可使用一种语法,但是不能在同一源文件中混合“标准语法”和 XML 语法。

本节也使用术语“JSP 文档”指代 XML 语法形式的 JSP 页面。

## 1. JSP 页面的 XML 语法

XML 语法形式的 JSP 页面是一个名域确定的 XML 文档。名域用来标识核心语法, 也用来描述页面中使用的标记库。所有 JSP 相关的名域在 XML 文档的 root 中引入。XML 语法形式的 JSP 页面可使用下列元素:

Jsp:root 元素为所有页面中自定义标记引入名域。

directive 元素

scripting 元素

标准 action 元素

自定义 action 元素

jsp:cdata 元素对应 template data

其它 XML 片段

XML 语法形式的 JSP 页面语义模型与 JSP 语法形式的 JSP 页面语义模型是相同的, 以产生字符的响应流的观点来说, 它们仍定义自己的语义。

### 1) jsp:root 元素

JSP 文档用 jsp:root 作为自身的根元素类型。root 是 taglib 插入它们名域属性的地方。顶层元素有 xmlns 属性, 因此可使用 JSP1.2 规范中定义的标准元素。

所有 JSP 文档中使用的标记库都表示为根元素的附加 xmlns 属性。这个元素中没有其它属性。

```
<jsp:root
  xmlns:jsp="http://java.sun.com/jsp_1_2"
  xmlns:prefix1="URI-for-taglib1"
  xmlns:prefix2="URI-for-taglib2"... >
  JSP page
</jsp:root>
```

自定义标记库的 xmlns 属性的形式 xml:prefix='uri' 通过 uri 值表示标记库, uri 有形式 "urn:jsp:tld:path"。描述怎样为 JSP 语法形式的 JSP 页面中的 taglib 指令定位标记库描述符。

### 2) jsp:directive.page 元素

jsp:directive.page 元素定义了大量页面相关属性并传递给 JSP 容器。这个元素必须是 root 元素的孩子且必须出现在 JSP 文件的开始。它的语法是:

```
<jsp:directive.page page_directive_attr_list />
page_directive_attr_list 如第 5 章所述。
```

Jsp:directive.page 元素的解释第 5 章所述, 在这儿, 它的作用域是 JSP 文档和通过包含指令包含进来的任何片段。

### 3) jsp:directive.include 元素

jsp:directive.include 元素在翻译时, 用来代替文本和/或代码。这个元素可出现在 JSP 文档的任何地方。它的语法是:

```
<jsp:directive.include file="relativeURLspec? flush="true|false" />
```

jsp:directive.include 的解释第 5 章所述。

#### 4) jsp:declaration 元素

jsp:declaration 元素被用来声明对所有其它脚本元素有用的脚本语言构造器。jsp:declaration 元素没有属性并且它的体是声明自身。它的语法是:

```
<jsp:declaration> declaration goes here </jsp:declaration>
```

jsp:declaration 的解释第 5 章所述。

#### 5) jsp:scriptlet 元素

jsp:scriptlet 元素用来描述响应中执行的行为。脚本片段是程序片段。Jsp:scriptlet 元素没有属性并且它的体是由脚本片段组成的程序片段。它的语法是:

```
<jsp:scriptlet> code fragment goes here </jsp:scriptlet>
```

jsp:scriptlet 的解释第 5 章所述。

#### 6) jsp:expression 元素

jsp:expression 元素以脚本语言描述完整的表达式并在响应时求值。jsp:expression 元素没有属性并且它的体是表达式。它的语法是:

```
<jsp:expression> expression goes here </jsp:expression>
```

jsp:expression 的解释第 5 章所述。

#### 7) 标准和自定义 action 元素

JSP 文档可使用如前所述的标准行为,因为这些行为的语法本来就是基于 XML 的,所以前面的描述已经足够了。为了内容的完整性,行为元素如下:

jsp:useBean

jsp:setProperty

jsp:getProperty

jsp:include

jsp:forward

jsp:param

jsp:params

jsp:plugin

标记的嵌套限制与前面保持一致,除了下面的 jsp:cdata。

这些元素的解释也与前面描述的相同,唯一不同的是 jsp:cdata 元素。

#### 8) 请求时属性

可接受请求时属性的行为元素为形如“% = text”的属性接收参数(text 两边没有空白并且注意缺‘<’和‘>’)。这个 text 其它 XML 文档引用后,将作为表达式求值。

#### 9) jsp:cdata 元素

jsp:cdata 元素用来封装 XML 表示中的 template data。jsp:cdata 元素没有属性值并且能出现在 template data 出现的任何地方。template data 只能出现在属性或 jsp:cdata 元素中。它的语法是:

```
<jsp:cdata> template data </jsp:cdata>
```

jsp:cdata 元素的解释是把它的内容传递给 out 的当前值。



注意:因为定义了 JSP 页面的 XML 语法,所以 template data 可以不经 jsp:cdata 元素封装而直接出现。

## 2. 映射

这部分描述以 JSP 语法书写的 JSP 页面和它对应的以 XML 语法书写的 JSP 页面。JSP 页面的标准 XML 文档由 JSP 页面的翻译器定义。

增加<jsp:root>作为根,有 xmlns:jsp 属性。

转换所有的<%元素为有效的 XML 元素和下述部分。

转换 JSP 引用为 XML 引用。

转换 taglib 指令为<jsp:root>元素的 xmlns:attributes。

为 JSP 页面的所有片段建立<jsp:cdata>元素,不用与 JSP 元素一一对应。

如表 13-1 所示:

表 13-1 指令和脚本元素的标准 XML 标记

JSP 页面元素	XML 等价物
<%@ page ... %>	<jsp:directive.page ... />
<%@ taglib ... %>	jsp:root element is annotated with namespace information.
<%@ include ... %>	<jsp:directive.include ... />
<%! ... %>	<jsp:declaration> .... </jsp:declaration>
<% ... %>	<jsp:scriptlet> .... </jsp:scriptlet>
<% = .... %>	<jsp:expression> .... </jsp:expression>

### 1) 页面指令

页面指令的形式:

<%@ page {attr="value"} \* %>

被转换成下述形式的元素:

<jsp:directive.page {attr="value"} \* />

### 2) taglib 指令

taglib 指令的形式:

<%@ taglib uri="uriValue" prefix="prefix" %>

被转换成 JSP 文档根下的 xmlns:prefix 属性,其值由 uriValue 决定。如果 uriValue 是相对路径,那么使用的值是“urn:jsptld:uriValue”;否则,直接使用 uriValue。

### 3) include 指令

include 指令的形式:

<%@ include file="value" %>

被转换成下述形式的元素:

<jsp:directive.include file="value"/>

### 4) Declarations

例:



```
<%! public String f(int i) { if (i<3) return("...");... } %>
```

被转换成使用一个 CDATA 语句以避免必须在 jsp:declaration 中引用“<”

```
<jsp:declaration> <![CDATA[ public String f(int i) { if (i<3)
return("..."); } ]]> </jsp:declaration>
```

#### 5) Scriptlets

在 JSP 页面相应的 XML 文档中,directives 使用如下语法表示:

```
<jsp:scriptlet> code fragment goes here </jsp:scriptlet>
```

DTD Fragment

```
<! ELEMENT jsp:scriptlet ( #PCDATA) >
```

#### 6) Expressions

在 JSP 页面相应的 XML 文档中,directives 使用如下语法表示:

```
<jsp:expression> expression goes here </jsp:expression>
```

```
<! ELEMENT jsp:expression ( #PCDATA) >
```

#### 7) 标准和自定义 Actions

标准和自定义 Actions 元素的语法都是基于 XML 的,唯一需要转换的是引用规则和请求时属性表达式的语法。

#### 8) 请求时属性表达式

请求时属性表达式形式如“<% = expression %>”。尽管这个语法与使用在 JSP 页面其它地方的语法是相容的,但是它不是一个合法的 XML 语法。表达式的 XML 映射是转换成形式“% = expression”的值。这里 JSP 规范的引用规则已经被转换成 XML 引用规则。

#### 9) Template Text 和 XML 片段

所有 JSP 翻译器不解释的文本都将转换成<jsp:cdata>元素的体。结果是没有 XML 片段会出现在经过翻译获得的 JSP 文档中。

### 3. 确认 JSP 文档

JSP 文档是一个名域确定的文档。然而除了在相当简单的场合,否则它不能使用 DTD 确认。当然,DTD 描述文档的目的是有用的。确定名域的机制,如 Xschema,相当适合描述 JSP 文档。

## 13.1.3 实例

这一小节展示两个 JSP 文档的例子。第一个展示 JSP 语法形式的 JSP 页面和它的 XML 语法形式的映射。第二个例子展示 XML 语法形式的 JSP 文档包含 XML 片段。

### 1.JSP 页面和它对应的 JSP 文档

#### 1) JSP 和 XML 语法之间映射的例子

```
<title> positiveTagLib</title>
```

```
<body>
```

```

<%@ taglib uri="http://java.apache.org/tomcat/examples-taglib" prefix="eg" %>
<%@ taglib uri="/tomcat/taglib" prefix="test" %>
<%@ taglib uri="WEB-INF/tlds/my.tld" prefix="temp" %>
<eg:test toBrowser="true" att1="Working">
Positive Test taglib directive </eg:test>
</body>
</html>

```

## 2) XML 语法形式

```

<jsp:root xmlns:jsp="http://java.sun.com/jsp_1_2"
xmlns:eg="http://java.apache.org/tomcat/examples-taglib"
xmlns:test="urn:jsptld:/tomcat/taglib"
xmlns:temp="urn:jsptld:/WEB-INF/tlds/my.tld">
<jsp:cdata><![CDATA[<html>
<title>positiveTagLig</title>
<body>
]]></jsp:cdata>
<eg:test toBrowser="true" att1="Working">
<jsp:cdata>Positive test taglib directive</jsp:cdata>
</eg:test>
<jsp:cdata><![CDATA[
</body>
</html>
]]></jsp:cdata>
</jsp:root>

```

## 2. 有 XML 片段的 JSP 文档

```

<jsp:root xmlns:jsp="http://java.sun.com/jsp_1_2"
xmlns:mytags="prefix1-URL">
<mytags:iterator count="4">
<foo> </foo>
</mytags:iterator>
</jsp:root>

```

## 13.2 标记扩展

### 13.2.1 taglib Directive

JSP 容器可解释的一组有意义的标记能通过“tag library”扩展。JSP 页面中 taglib 指

令声明页面要使用标记库。使用 URI 唯一的标识一个标记库并且加上一个前缀标记以区别库中的行为。

如果 JSP 容器实现不能查找到标记库的描述,那么将产生一个致命翻译错误。

如果 taglib 指令出现在使用它引入前缀的行为后面,也将产生一个致命翻译错误。

标记库可能包含一个能保证 JSP 页面正确使用标记库功能的确认方法。

#### 1) 语法

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix"%>
```

#### 2) 属性:

uri: 一个绝对 URI 或一个相对 URI, 唯一标识与这个前缀相联系的标记库描述符。

tagPrefix: 定义在 <prefix> : <tagname> 中的 prefix 字符串, 用来区别自定义行为, 例如: <myPrefix : myTag>。

前缀“jsp:”, “jspx:”, “java:”, “jax:”, “servlet:”, “sun:”, “sunw:”被保留。在 JSP1.2 中, 空前缀是非法的。

如果 JSP 页面翻译者遇到使用 taglib 指令引入的前缀名为 prefix:Name 的标记, 但是 Name 未被相应的标记库验证, 此时将产生一个致命翻译错误。

例:

下面例中, 标记库被引入, 使这个页面可以使用“super”前缀, 当然不能再引入一个标记库也使用这个前缀。在这种特定情况下, 我们假设标记库包含一个“doMagic”元素类型, 那么, 在页面内应按如下使用。

```
<%@ taglib uri="http://www.mycorp/supertags" prefix="super"/>
...
<super:doMagic>
...
</super:doMagic>
```

## 13.2.2 标记库描述器及其格式

### TLD

TLD 是“Tag Library Descriptor”, 它是描述标记库的 XML 文档。标记库的 TLD 由 JSP 容器使用解释标记库指令。

#### 1) TLD 格式

标记库描述器的 DTD 如下:

#### 2) 根元素

taglib 元素是文档根元素。taglib 有两个属性。

```
<! ATTLIST taglib
id
```

ID

# IMPLIED

```
xmlns
  CDATA
  #FIXED
  "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd"
>
```

### 3) 子元素

taglib 元素也定义了几个子元素:

tlib-version: 一个标记库实现的版本号。

jsp-version: 标记库支持的 JSP 规范版本号。

short-name: JSP 页面开发工具建立记忆名使用的简单的默认短名字。例如,它可能在 taglib 指令中被优先使用。

uri: 唯一标识这个标记库的 uri。

display-name: 包含将由工具显示的短名字。

small-icon: 可被工具使用的可选小图标。

large-icon: 可被工具使用的可选大图标。

description: 描述这个标记库使用的字符串。

validator: 可选标记库确认器信息。

listener: 可选事件监听者规范。

<! ELEMENT taglib

(tlib-version, jsp-version?,

short-name, uri?, display-name?, small-icon?, large-icon?

description?, validator?, listener\*, tag+)>

## 13.3 本章小结

本章主要讲述了 JSP 文档的另一种表达形式——XML 形式,介绍了各语法的 XML 等价物。还介绍了 <%@ taglib ..... %> 的使用。

# 第 14 章 网站建设

本章讲授网站建设。在此之前,我们在第 5 章,实现了广告轮显,结束了网站“没头没脑”的日子;在第 8 章,借“JSP 文档生成器”的东风,实现了留言板,使网站变成了一个“谦谦君子”;在第 9 章,实现了每个网站必备的东西——计数器,成为网站跳动的“心脏”;在第 10 章,实现了非常重要的“新闻”,使网站“耳聪目明”起来;在第 12 章,我们引入了时间和日期 Applet 使网站活泼起来,变得明眸善睐。至此,我们的虚拟网站应该有图 14-1 所示效果。

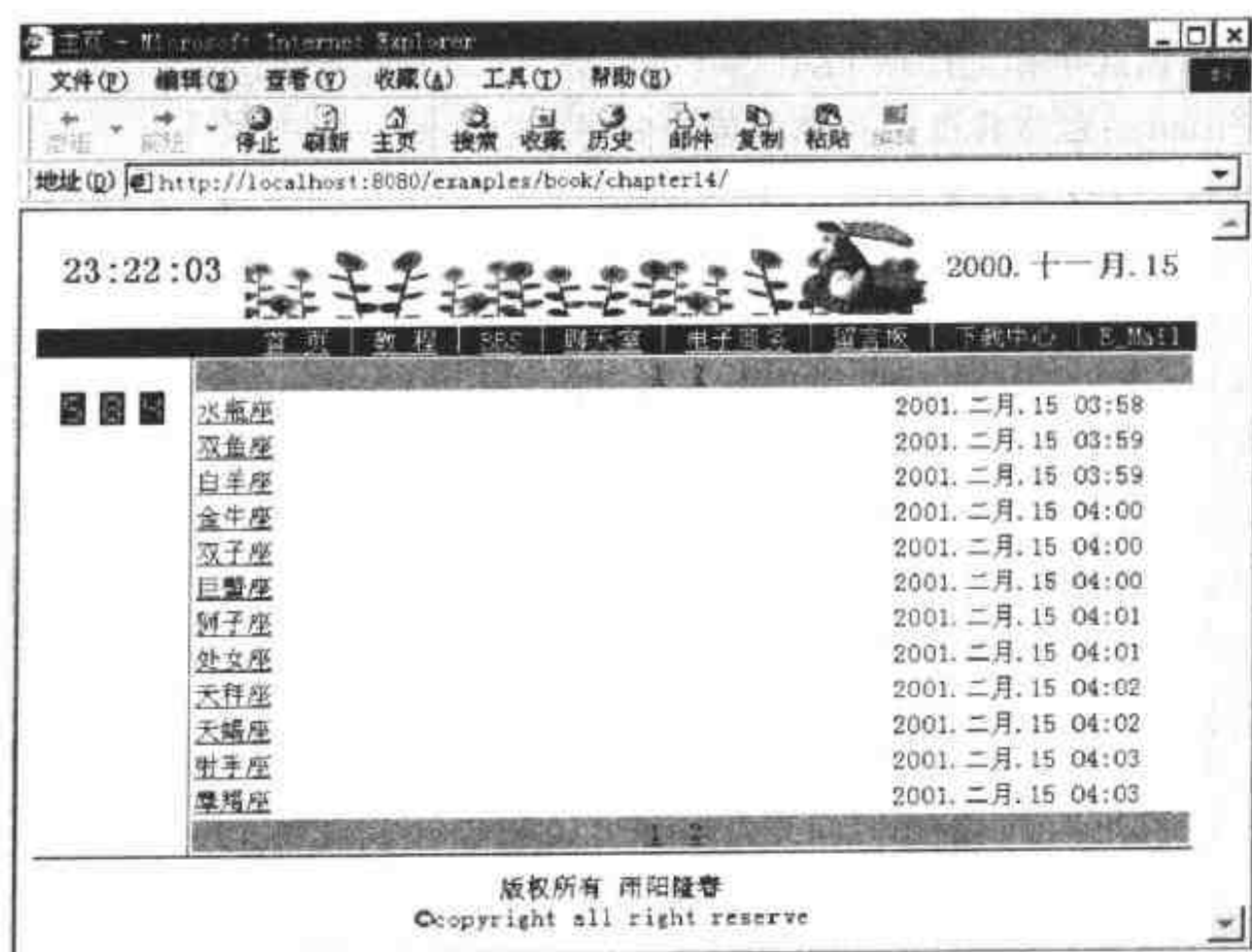


图 14-1 网站前期完成效果图

本章将着重介绍 BBS、电子商务、聊天室等重要内容。

## 14.1 BBS

BBS 就是数据库那点内容吗? 对, 没错! 大家非常清楚 BBS 的实现技术, 好, 让我们“八仙过海, 各显神通”实现自己的 BBS。实现的 BBS(未完)效果如图 14-2 所示。

相信每个读者都应该比以上效果实现得更强。读者可根据自己的情况, 适当参考。不过, BBS 还是相当庞杂的, 实现起来并不是十分容易。

通常 BBS 采用二级版面形式, 如同文件目录一样, 目录下面有子目录, 二级版面的意思就是有两层目录的意思。这里实现的就是二级版面, 图 14-2 表现得很清楚, 这是进入

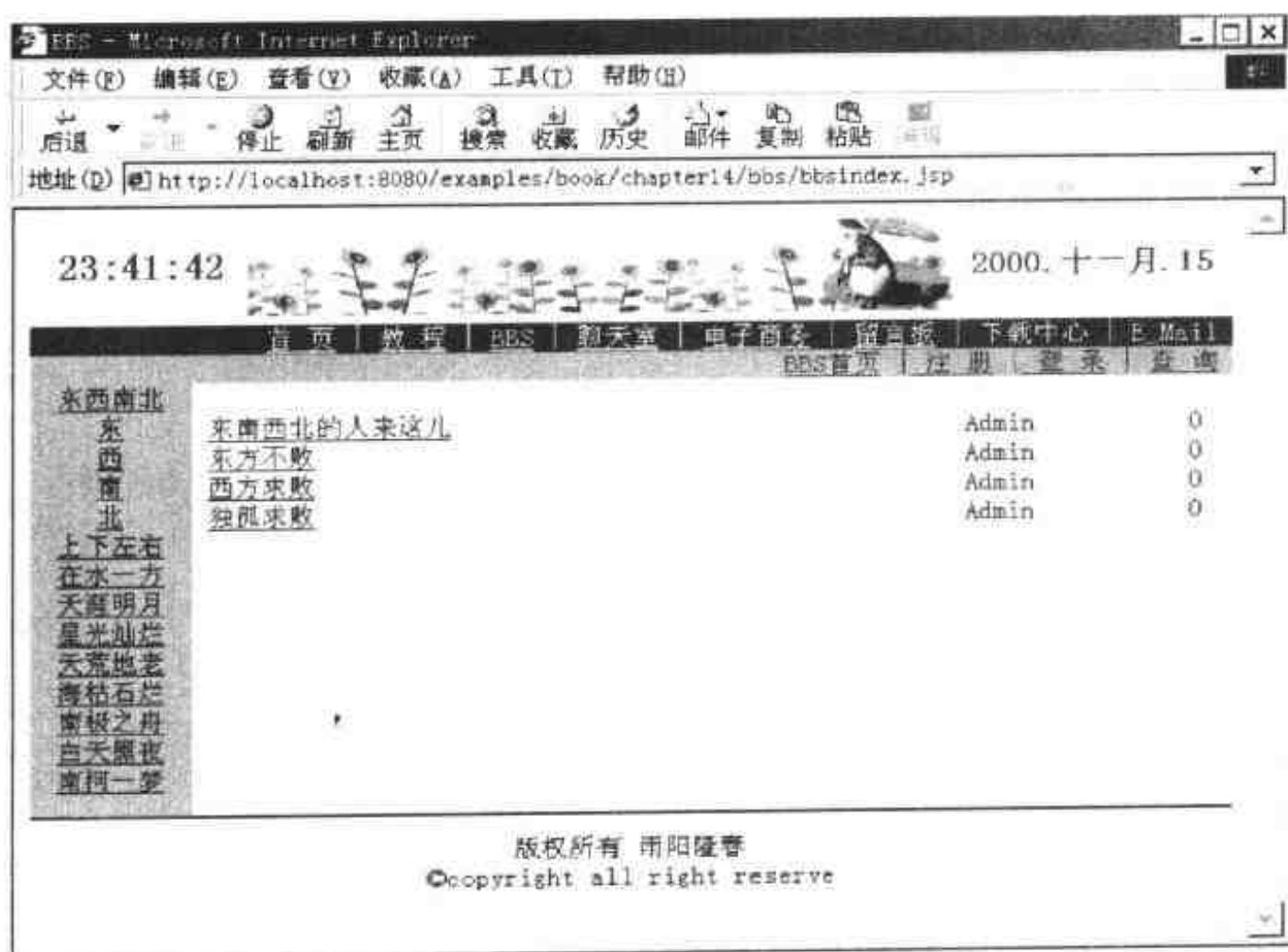


图 14-2 BBS 效果图

BBS 默认打开的版面和子版面情况。对二级版面的显示分两种情况，一是完全显示，就是说全部的版面及其子版面都列出，一眼就能看到有多少个版面，每个版面有多少子版面。还有一种就是不显示所有的版面，只显示用户请求版面下的子版面，其它子版面信息隐藏。这与 Windows 的资源管理器目录树相似。这两种实现形式各有优劣，前者实现起来比较容易，对用户了解全面信息非常有帮助；后者对服务器负载较小。这里实现的是第二种形式。如果读者没有明白，请比较图 14-3 和图 14-2 的效果，应该不难理解。

实际使用的 BBS 基本上都采用“超链接标题”形式，即在子版面下显示文章标题，文章标题超链接到文章的详细内容，如图 14-2 和图 14-3 所示。

这就是通常的 BBS 版面组织形式，他们的共同目标是实现 BBS 最基本的功能也是最重要的功能——分类组织文章和显示文章内容。通常这些都集成在 BBS 的首页，即进入 BBS 或点击 BBS 首页通常显示该 BBS 的版面组织和文章标题。

那么版面与子版面，子版面与文章是怎样联系起来的呢？为什么点击某个版面，其子版面就显示出来了呢？一般由两个机制保证这样的行为实现，一是数据库表之间建立某种关联，二是使用 URL 重写技术。这将在后面的例子中多次用到，不再详细叙述。

通常版面组织和文章显示集成在首页。那么首页怎样排版布局呢？通常采用框架组织。一般将页面分为三个框架，分别显示页头、版面、文章标题。这样组织的最大好处是服务器负载小，因为不用对整个页面刷新，也就不需要大量的数据库操作。本书没有采用通常的方式，而是采用一个页面的形式，好处是页面整体感强，可用空间大。坏处就是需要大量数据库操作，响应速度慢。

通过上面的叙述，想来你已经勾画出自己的 BBS 组织结构蓝图了，那么，建立数据库也就不成问题了。通常数据库的结构就是与版面组织结构对应。如还有问题，请参见光盘上给出的简单的示范性数据库。









```

Class.forName(_driver).newInstance();
String _connectionURL = "jdbc:odbc:Bbs";
Connection _con = null;
_con = DriverManager.getConnection(_connectionURL);
Statement _stmt = null;
_stmt = _con.createStatement();
ResultSet _rs = null;
_rs = _stmt.executeQuery(_sqlquerystring);
//以输入信息查询数据库,如数据库中已经存在这样的信息,重导回注册页面
if(_rs.next()){
    _rs.close();
    _stmt.close();
    _con.close();
    String _redirectURL = "/examples/book/chapter14/bbs/register.jsp";
    response.sendRedirect(response.encodeURL(_redirectURL));
}
else{
    //否则,以输入信息更新数据库
    _stmt = _con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                                   ResultSet.CONCUR_UPDATABLE);

    _sqlquerystring = "SELECT * FROM User";
    _rs = _stmt.executeQuery(_sqlquerystring);
    _rs.last();
    _rs.updateString(2, _username);
    _rs.updateString(3, _password);
    _rs.updateInt(4, Math.round(6663 * (float) Math.random()));
    _rs.updateString(5, _e_mail);
    //直到下一句执行成功,数据库更新操作才完成。否则,不做任何动作。_rs.insertRow();

    //一个完整的警告处理格式
    SQLWarning warn = _rs.getWarnings();
    if(warn != null){
        out.println("----- Warning -----");
        out.println("<br>");
        while(warn != null){
            out.println("Message:" + warn.getMessage());
            out.println("<br>");
            out.println("SQLState:" + warn.getSQLState());
            out.println("<br>");
        }
    }
}

```

```

        out.println("Vendor error code:");
        out.println(warn.getErrorCode());
        out.println("<br>");
        warn = warn.getNextWarning();
    }
}

    _rs.close();
    _stmt.close();
    _con.close();
    //注册成功,重导向登录页面
    String _redirectURL = "/examples/book/chapter14/bbs/login.jsp";
    response.sendRedirect(response.encodeURL(_redirectURL));
}
}

//一个完整的 SQL 异常处理格式
catch(SQLException ex){
    out.println("SQLException caught");
    out.println("<br>");
    while(ex != null){
        out.println("Message: " + ex.getMessage());
        out.println("<br>");
        out.println("SQLState: " + ex.getSQLState());
        out.println("<br>");
        out.println("ErrorCode: " + ex.getErrorCode());
        out.println("<br>");
        ex = ex.getNextException();
    }
}

%>
</body>
</html>

```

### 3. 说明

在注册处理代码中,必填输入项如果未填或数据库中已有同名记录,都重导回注册页面。如果注册成功,则转发至登录页面,页面效果如图 14-5 所示。

#### 4. 改进建议

增加一个注册不成功处理页面。凡是注册不成功都导向这个页面,使用 `request.setAttribute()` 方法将错误情况转发给该处理页面,这个处理页面使用 `request.getAttribute()` 方法获得错误情况,然后分别处理。详细论述参见后面的 14.2 节电子商务有关例子的论述。

增加一个注册成功标志页面,以便给用户一个明确信息。获取更详细的资料,实际应用的页面远远比这里的复杂。对同名注册不成功情况,可以更进一步地给出用户输入字符串的相似字符串,使用户获得更多的参考信息。

### 14.1.3 登录

## 1. 页面效果图及其源代码

页面效果如图 14-5 所示,代码如下:

[illegible]

```

        </p>
    </form>
    <p> &nbsp; </p>
</td>
</tr>
</table>
</body>
</html>

```

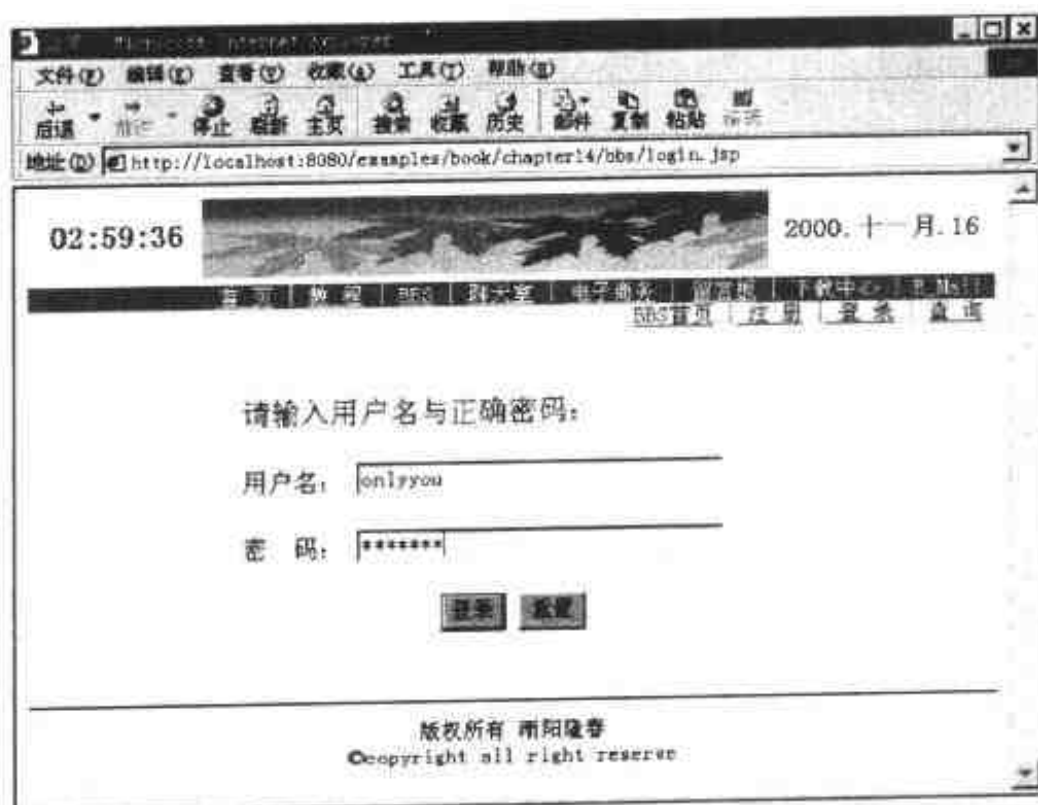


图 14-5 登录页面效果图

## 2. 页面逻辑处理代码

```
<%@page import="java.sql.*"%>
<%@page import="javax.servlet.*"%>
<%@page contentType="text/html; charset=gb2312"%>
```

```
<html>
<head>
<title></title>
</head>
<body>
```

```
< %
String _driver = "sun.jdbc.odbc.JdbcOdbcDriver";
Class.forName(_driver).newInstance();
String _connectionURL = "jdbc:odbc:Bbs";
Connection _con = null;
```

```
_con = DriverManager.getConnection(_connectionURL);
Statement _stmt = null;
_stmt = _con.createStatement();
String _username = request.getParameter("Username");
String _password = request.getParameter("Password");
//输入为空,重导回这个页面
if(_username.equals("") || _password.length() == 0){
    String _redirectURL = "/examples/book/chapter14/bbs/login.jsp";
    response.sendRedirect(response.encodeURL(_redirectURL));
}
else{
    String _sqlstring = "SELECT Username, Password FROM User";
    _sqlstring + = " ";
    _sqlstring + = "WHERE Username LIKE '" + _username + "'";
    ResultSet _rs = null;
    _rs = _stmt.executeQuery(_sqlstring);
    if(_rs.next()){
        if(_rs.getString(2).equals(_password)){
            _rs.close();
            _stmt.close();
            _con.close();
            //如果数据库中存在该用户名,并且密码正确,重导至 BBS 首页
            String _redirectURL = "/examples/book/chapter14/bbs/bbsindex.jsp";
            response.sendRedirect(response.encodeURL(_redirectURL));
        }
        else{
            _rs.close();
            _stmt.close();
            _con.close();
            //存在该用户名,但密码不正确,重导回登录页面
            String _redirectURL = "/examples/book/chapter14/bbs/login.jsp";
            response.sendRedirect(response.encodeURL(_redirectURL));
        }
    }
    else{
        _rs.close();
        _stmt.close();
        _con.close();
        //不存在该用户名,重导至注册页面
```

```
String _redirectURL = "/examples/book/chapter14/bbs/register.jsp";  
response.sendRedirect(response.encodeURL(_redirectURL));  
}
```

%>

</body>

</html>

### 3. 说明

非常有用的两个方法 `response.encodeURL(URL)` 和 `response.sendRedirect()`。

### 4. 改进建议

同样增加错误处理页面。

## 14.1.4 查询

### 1. 页面效果图及其源代码

页面效果如图 14-6 所示。代码如下：



图 14-6 查询效果图

<html>

<head>

<title>用户查询</title>

<meta http-equiv="Content-Type" content="text/html; charset=gb2312">

</head>





```

<body bgcolor = " # FFFFCC" leftmargin = "10" topmargin = "8" alink = " # FFFF33"
link = " # FFFFFFF" vlink = " # 990099" >
<% @page import = "java.sql. * "% >
<% @page import = "javax.servlet. * "% >
<% @page contentType = "text/html; charset = gb2312"% >

<jsp:include page = "/book/chapter14/template/header2.jsp" flush = "true"/>

<%
    String _username = request.getParameter("Username");
    String _usernameDisplay = "";
    if(_username.equals("")){
        _usernameDisplay = "所有用户";
    }
    else{
        _usernameDisplay = _username;
    }
    String _area = request.getParameter("Area");
    String _sqlstring = "SELECT Username, State, E _ Mail FROM User";
    _sqlstring + = " ";
    _sqlstring + = "WHERE Username LIKE ' % " + _username + " % '";
    String _driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    Class.forName(_driver).newInstance();
    String _connectionURL = "jdbc:odbc:Bbs";
    Connection _con = null;
    _con = DriverManager.getConnection(_connectionURL);
    Statement _stmt = null;
    _stmt = _con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
    ResultSet _rs = null;
    _rs = _stmt.executeQuery(_sqlstring);
    ResultSetMetaData _rsmd = _rs.getMetaData();
    int _columnCount = _rsmd.getColumnCount();
% >
<table align = "center" border = "0" width = "756" height = "280" cellpadding = "0" cell-
spacing = "0">
    <tr height = "40" valign = "bottom">
        <td> &nbsp; </td>

```

```

        <td colspan="5"> <font color="#990099"><i>查询<%=_usernameisplay%>结果</i></font>
    </td>
</tr>
<tr bgcolor="#99CC66">
    <% for(int i=0;i<_columnCount;i++){ %>
        <% String _line=_rsmd.getColumnLabel(i+1); %>
        <td width="6%" height="20">&nbsp;</td>
        <td width="24%" height="20">
            <font size="3" color="#FFFFFF">
                <% out.println(_line); %>
            </font>
        </td>
    <% } %>
</tr>
<% while(_rs.next()){ %>
    <tr>
        <% for(int i=0;i<_columnCount;i++){ %>
            <% String _line=_rs.getString(i+1); %>
            <td width="6%" height="16">&nbsp;</td>
            <td width="24%" height="16">
                <font size="2">
                    <% out.println(_line); %>
                </font>
            </td>
        <% } %>
    </tr>
    <% if(_rs.next()){ %>
        <tr bgcolor="#eeffcc">
            <% for(int i=0;i<_columnCount;i++){ %>
                <% String _line=_rs.getString(i+1); %>
                <td width="6%" height="16">&nbsp;</td>
                <td width="24%" height="16">
                    <font size="2">
                        <% out.println(_line); %>
                    </font>
                </td>
            <% } %>
        </tr>
    }
}

```

```
<% | %>
<% | %>
<% _rs.close(); %>
<% _stmt.close(); %>
<% _con.close(); %>
<tr>
<td></td>
</tr>
</table>
```

```
<jsp:include page = "/book/chapter14/template/footer.jsp" flush = "true"/>
```

```
</body>
```

```
</html>
```

### 3. 说明

这里只实现了在所有注册用户范围内的查询。

查询结果如图 14-7 所示。



图 14-7 查询结果图

### 4. 改进建议

如果要在实现在线用户范围的查询,需要在数据库中添加一个数据项,标识用户是否在线,或者使用一个向量,动态的维持在线用户状态。当然,判断是否在线还得使用 session。

### 14.1.5 版面显示

#### 1. 页面效果图

页面效果如图 14-2 和图 14-3 所示。

#### 2. 页面处理代码

```
<%@page import="java.sql.* ,javax.servlet.*"
    contentType="text/html; charset = gb2312"
%>
<%
    String _catalogidreq = request.getParameter("catalogid");
    if(_catalogidreq == null){
        _catalogidreq = "1";
    }
    String _sqlstring = "SELECT ID,Catalog FROM Catalog";
    String _driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    Class.forName(_driver).newInstance();
    String _connectionURL = "jdbc:odbc:Bbs";
    Connection _con = null;
    _con = DriverManager.getConnection(_connectionURL);
    Statement _stmt = null;
    _stmt = _con.createStatement();
    ResultSet _rs = null;
    _rs = _stmt.executeQuery(_sqlstring);
    ResultSetMetaData _rsmd = _rs.getMetaData();
    int _columnCount = _rsmd.getColumnCount();
%>
<table width="100%" border="0" cellspacing="0" cellpadding="0" align="center"
bgcolor="#99cc66">
<%while(_rs.next()){%>
<tr align="center">
<%for(int i=0;i<_columnCount;){%>
<td>
<%
    String _catalogid = _rs.getString(++i);
    String _subcatalogid = "1";
    String _sqlstring1 = "SELECT ID,Subcatalog FROM Subcatalog";
```

```

    _sqlstring1 = _sqlstring1 + " " + "WHERE Hypotaxis=" + _catalogid;
    _sqlstring1 = _sqlstring1 + " " + "ORDER BY ID ASC";
    Statement _stmt1 = null;
    _stmt1 = _con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                    ResultSet.CONCUR_READ_ONLY);

    ResultSet _rs1 = null;
    _rs1 = _stmt1.executeQuery(_sqlstring1);
    if(_rs1.next()){
        _subcatalogid = _rs1.getString(1);
    }
    _rs1.beforeFirst();
%>

<font size="2">
<a
href="bbsindex.jsp? catalogid = < % = _catalogid % > &subcatalogid = < % = _subcata-
logid % >">
    < % = _rs.getString( ++ i) % > </a>
</font>
</td>
</tr>

< % if(_catalogid.equals(_catalogidreq)) { % >
<tr><td>
    <table width="100 %" border="0" cellspacing="0" cellpadding="0" align="center"
bgcolor="#99cc66">
        < % while(_rs1.next()) { % >
        <tr align="center">
        <td>
            <font size="2">
            <a
href="bbsindex.jsp? catalogid = < % = _catalogid % > &subcatalogid = < % = _rs1.get-
String(1) % >">
                < % = _rs1.getString(2) % > </a>
            </font>
        </td>
        </tr>
        < % } % >
    </table>
    < % _rs1.close(); % >

```

```

<%_stmt1.close();%>
</td></tr>
<|%>

```

```

<%
    }
}
_rs.close();
_stmt.close();
_con.close();
%>
</table>

```

### 3. 说明

如果需要交替执行查询,必须使用两个 Statement 对象。

### 4. 改进建议

在数据库版面表中,如增加一个数据项,记录它的一个子版面,缺点是易使数据处于不一致状态。

将显示版面、文章标题分为两步处理,降低了复杂性与大量数据库操作,但缺点是数据更新不同步。

## 14.1.6 文章标题显示

### 1. 页面效果图

页面效果如图 14-2 和图 14-3 所示。

### 2. 页面处理代码

代码如下:

```

<%@page import="java.sql.*",javax.servlet.*"
    contentType="text/html; charset = gb2312"
%>
<%
    String _catalogidreq = request.getParameter("catalogid");
    if(_catalogidreq == null){
        _catalogidreq = "1";
    }
    String _subcatalogidreq = request.getParameter("subcatalogid");

```

```

        if(_subcatalogidreq == null){
            _subcatalogidreq = "1";
        }
        String _sqlstring = "SELECT ID,Article,Username,Clickcount FROM Article";
        _sqlstring = _sqlstring + " " + "WHERE HypotaxisID=" + _subcatalogidreq;
        String _driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        Class.forName(_driver).newInstance();
        String _connectionURL = "jdbc:odbc:Bbs";
        Connection _con = null;
        _con = DriverManager.getConnection(_connectionURL);
        Statement _stmt = null;
        _stmt = _con.createStatement();
        ResultSet _rs = null;
        _rs = _stmt.executeQuery(_sqlstring);
        ResultSetMetaData _rsmd = _rs.getMetaData();
        int _columnCount = _rsmd.getColumnCount();
    % >
    <table width="100%" border="0" cellspacing="0" cellpadding="0">
        <% while(_rs.next()){ % >
        <tr height="16">
            <td><a
href="bbsindex.jsp? catalogid= < % = _catalogidreq % > &subcatalogid= < % = _subcat-
alogidreq % > &articleid= < % = _rs.getString(1) % >"><font size="2">< % = _rs.
getString(2) % >
                </font> </td>
            <td> <font size="2"> < % = _rs.getString(3) % > </font> </td>
            <td> <font size="2"> < % = _rs.getString(4) % > </font> </td>

        <%
out.println("</tr>");
        }
        _rs.close();
        _stmt.close();
        _con.close();
    % >
    <tr>
        <td colspan="3"></td>
    </tr>

```

</table>

### 3. 说明

使用 URL 重写技术,如:

<ahref="bbsindex.jsp? catalogid= < % = \_ catalogidreq % > &subcatalogid= < % = \_ subcatalogidreq % > &articleid= < % = \_ rs.getString(1) % >">使用了三个参数,显得非常的复杂。

rs.getString()是最常用的获取数据的方法。同时也要注意其它 getXXX 方法。

### 4. 改进建议

将其改为框架结构,减少 URL 重写使用的参数,从而减少了判断,达到简化逻辑和代码的目的。

使用连接池技术优化性能。或者使用 JavaBeans 封装数据库连接操作,简化连接过程。

## 14.1.7 文章显示

### 1. 页面效果图

页面效果如图 14-8 所示。

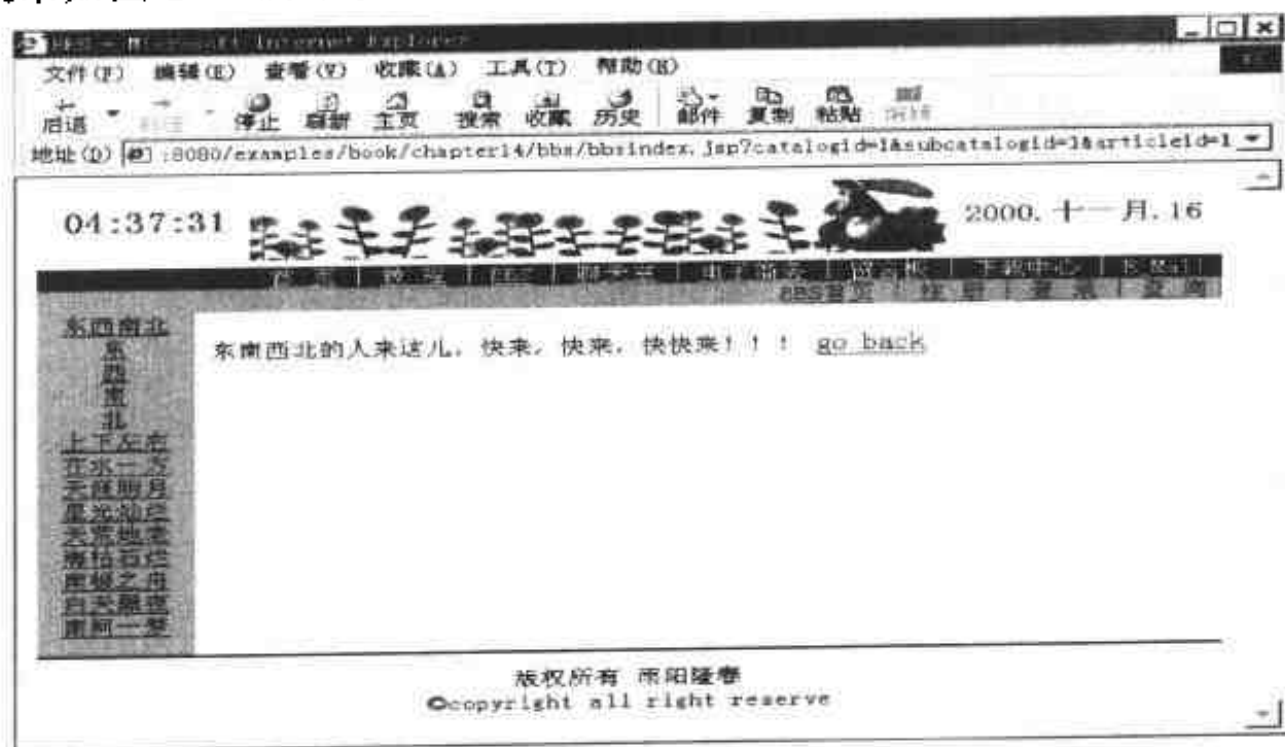


图 14-8 文章内容显示

### 2. 页面处理代码

代码如下:

```
< % @page import = "java.sql. * ,javax.servlet. * "
    contentType = "text/html; charset = gb2312"
% >
< %
```



```

String _catalogidreq = request.getParameter("catalogid");
if(_catalogidreq == null){
    _catalogidreq = "1";
}

String _subcatalogidreq = request.getParameter("subcatalogid");
if(_subcatalogidreq == null){
    _subcatalogidreq = "1";
}

String _articleidreq = request.getParameter("articleid");
if(_articleidreq == null){

% >
    <jsp:include page = "articlelist.jsp" flush = "true"/>
< %
}
else{
    String _sqlstring = "SELECT content FROM Article";
    _sqlstring = _sqlstring + " " + "WHERE ID = " + _articleidreq;
    String _driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    Class.forName(_driver).newInstance();
    String _connectionURL = "jdbc:odbc:Bbs";
    Connection _con = null;
    _con = DriverManager.getConnection(_connectionURL);
    Statement _stmt = null;
    _stmt = _con.createStatement();
    ResultSet _rs = null;
    _rs = _stmt.executeQuery(_sqlstring);
    ResultSetMetaData _rsmd = _rs.getMetaData();
    int _columnCount = _rsmd.getColumnCount();
    while(_rs.next()){
        out.println("<font size = 2>");
        out.println(_rs.getString(1));
        out.println("</font>");
    }

String href = "<a href = \"bbsindex.jsp? catalogid = ";
href = href + _catalogidreq + "&subcatalogid = " + _subcatalogidreq + "\">";
out.println(href);
out.println("go back");
out.println("</a>");
}

% >

```

至此,完全实现了 BBS 最基本的功能。值得注意的是本书的 BBS 实现了大量采用 include 包含行为,一方面提高了可扩展性,另一方面却带来了路径的烦琐。它作为一本书的例子,提供给大家实践还是比较适合的。这种模块化的功能,可以代换某个模块面几乎不改动其它模块。但是对于其它特定的实现,这种方法是不可取的,一是性能低下,二是细分模块的同时也使逻辑变得复杂。

## 14.2 电子商务

电子商务在 Web 上的应用比例越来越大,可以预见在未来社会里,电子商务将起着举足轻重的作用。因此,虚拟网站当然要实现电子商务,没有电子商务的网站可以被认为是一个不完善的网站。下面让我们走进电子商务,感受未来的电子货币时代。

本书讲解的电子商务使用 Servlet + JSP 的形式架构,也就是 Sun 及各软件公司包括 Microsoft 极力主张的将表示层和商业层分开的架构。现在,让我们一起进入“雨阳隆春在线书屋”。

### 14.2.1 首页

#### 1. 页面效果图及其源代码

首页是一个 JSP 文件,当然它与一个 HTML 文件几乎没有差别。页面效果如图 14-9 所示。

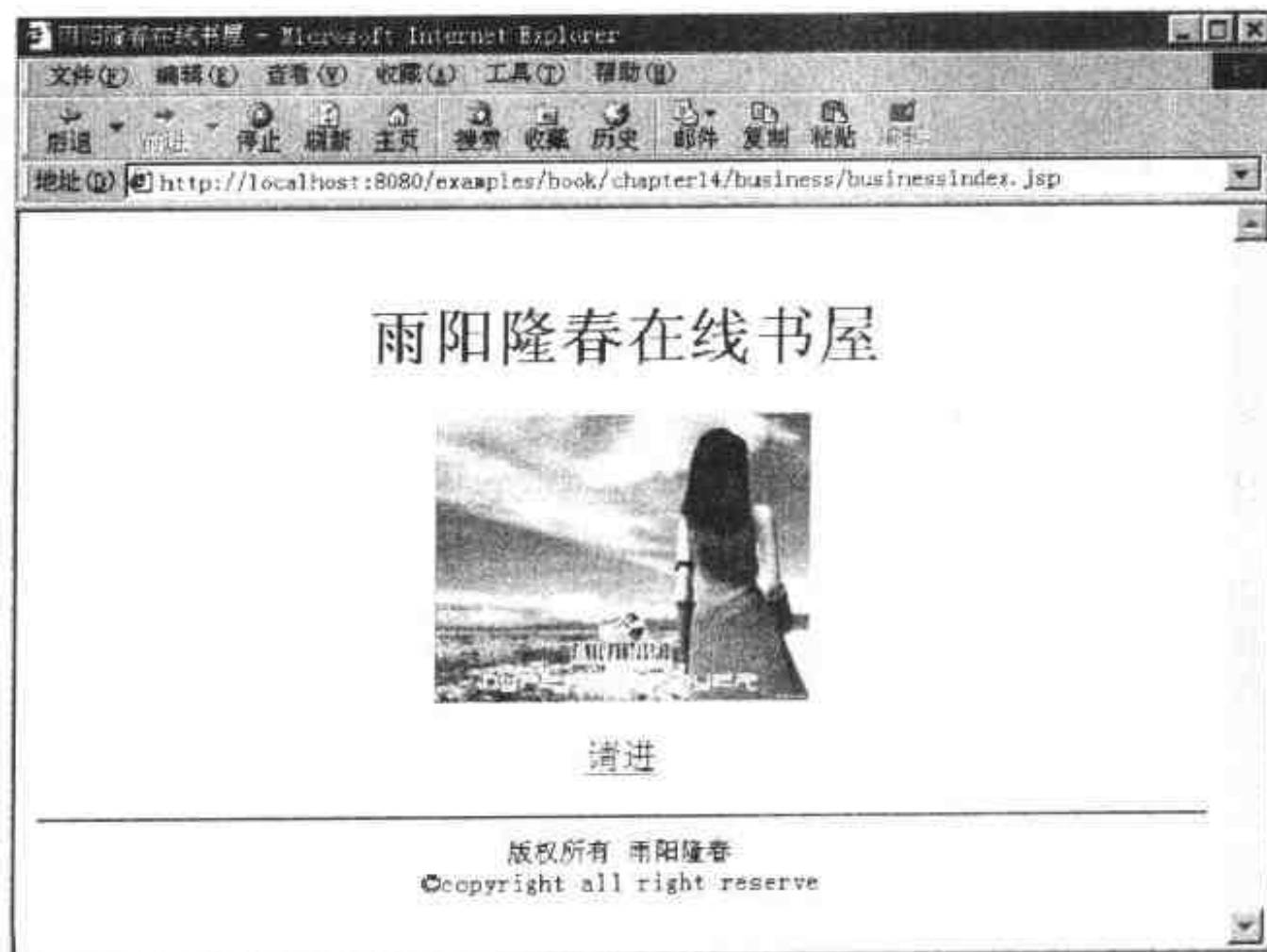


图 14-9 电子商务首页

页面源代码如下：

```
<%@ page contentType="text/html; charset = gb2312" %>
<html>
<head>
<title>雨阳隆春在线书屋</title>
<meta http-equiv="Content-Type" content="text/html; charset = gb2312">
</head>

<body bgcolor="#FFFFCC">
<p> &nbsp;   </p>
<p align="center"><font size="6" color="#990099"><b>雨阳隆春在线书屋
</b></font></p>
<p align="center"></p>
<p align="center"><a href="/examples/servlet/LoginBusiness"><font size="4">
请进</font></a></p>
<jsp:include page="../template/footer.jsp" flush="true" />
</body>
</html>
```

注意到 `<a href="/examples/servlet/LoginBusiness">`，这就是处理这个页面的 Servlet，将进入书屋的用户与一个 session 绑定。

## 2. 页面逻辑处理代码

代码如下：

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

public class LoginBusiness extends
    public void doGet (HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        try{
            //购物车的一个实例
            ShoppingCart cart = new ShoppingCart();
            //将用户与一个 session 绑定
            HttpSession session = request.getSession(true);
            //将购物车绑定到 session, 即用户获得一个具有唯一标识的购物车
            //用户与购物车具有一一对应关系
```

```

        session.setAttribute(session.getId(), cart);
        //转发请求
        RequestDispatcher
dispatcher = getServletContext().getRequestDispatcher("/book/chapter14/business/informa-
tiondesk.jsp");
        //捕获转发异常
        if(dispatcher == null){
            response.sendError(response.SC_NO_CONTENT);
        }
        dispatcher.forward(request, response);
    }
    catch(Exception e){
        System.out.println("error: " + e.getMessage());
    }
}
}
}

```

这里出现了 ShoppingCart 的类实例, ShoppingCart 类是怎样定义的呢? 请看代码:

```
import java.util. * ;
```

```

public class ShoppingCart{
    Vector items = null;

    public ShoppingCart(){
        items = new Vector();
    }

    public void addItem(BookDetails bookdetails){
        items.addElement(bookdetails);
    }

    public void removeItem(BookDetails bookdetails){
        items.removeElement(bookdetails);
    }

    public void removeItem(long bookId){
        for(int i=0; i<items.size(); i++){
            BookDetails bookdetails = (BookDetails)items.elementAt(i);

```

```
        if(bookdetails.getBookId() == bookId){
            items.removeElementAt(i);
            break;
        }
    }

    public void removeAllItem(){
        items.removeAllElements();
    }

    public int getNumberOfItems(){
        return items.size();
    }

    public BookDetails getItem(BookDetails bookdetails){
        return (BookDetails)items.elementAt(items.indexOf(bookdetails));
    }

    public BookDetails getItem(long bookId){
        int i = 0;
        boolean find = false;
        if(items.size() > 0){
            for(i = 0; i < items.size(); i++){
                BookDetails bookdetails = (BookDetails)items.elementAt(i);
                if(bookdetails.getBookId() == bookId){
                    find = true;
                    break;
                }
            }
            if(find){
                return (BookDetails)items.elementAt(i);
            }
            else{
                return null;
            }
        }
        else{
            return null;
        }
    }
```

```

    }
}

public BookDetails getItem(int i) {
    return (BookDetails)items.elementAt(i);
}

public void setItem(BookDetails bookdetails,int index) {
    items.setElementAt(bookdetails,index);
}

public int indexOf(BookDetails bookdetails) {
    return items.indexOf(bookdetails,0);
}

public void updateItem(long bookid,long amount) {
    BookDetails bookdetails = this.getItem(bookid);
    if(bookdetails != null) {
        int i = items.indexOf(bookdetails);
        long bid = bookdetails.getBookId();
        String btitle = bookdetails.getTitle();
        String author = bookdetails.getAuthor();
        float price = bookdetails.getPrice();
        String description = bookdetails.getDescription();
        bookdetails = new BookDetails(bid,btitle,author,price,amount,description);
        this.setItem(bookdetails,i);
    }
}
}

```

使用一个向量保存用户选购的书的详细信息。定义了数个重要的方法：

- (1) addItem: 添加一本书的详细资料到购物车。
- (2) removeItem: 从购物车中删除一本书。
- (3) removeAllItem(): 删除购物车中所有书的资料。
- (4) getNumberOfItems: 获得购物车中书的数量。
- (5) getItem: 获得某本书的详细资料。
- (6) setItem: 设置某本书的详细资料。
- (7) indexOf: 获得某本书在购物车中的位置。

(8) updateItem:更新某本书的详细资料。

这里又定义了一个 BookDetails 类。它是关于书的详细资料的类,定义如下:

```
import java.util. * ;
```

```
public class BookDetails{
```

```
    private long bookId = 0;
```

```
    private String title = null;
```

```
    private String author = null;
```

```
    private float price = 0;
```

```
    private long amount = 0;
```

```
    private String description = null;
```

```
    public BookDetails ( long bookId, String title, String author, float price, long  
amount, String description) {
```

```
        this.bookId = bookId;
```

```
        this.title = title;
```

```
        this.author = author;
```

```
        this.price = price;
```

```
        this.amount = amount;
```

```
        this.description = description;
```

```
    }
```

```
    public long getBookId() {
```

```
        return bookId;
```

```
    }
```

```
    public String getTitle() {
```

```
        return title;
```

```
    }
```

```
    public String getAuthor() {
```

```
        return author;
```

```
    }
```

```
    public float getPrice() {
```

```
        return price;
```

```
    }
```

```

    public long getAmount(){
        return amount;
    }

    public String getDescription(){
        return description;
    }
}

```

这里首先它定义了六个私有变量用于描述书的信息,然后提供了一个构造器和六个获取信息的方法,非常简单,不再细述。

### 14.2.2 进入书屋

首页为每个用户分配了一个购物车,并将购物车、用户绑定到一个唯一标识的 session 中。现在就进入了书屋的服务台,其效果如图 14-10 所示。

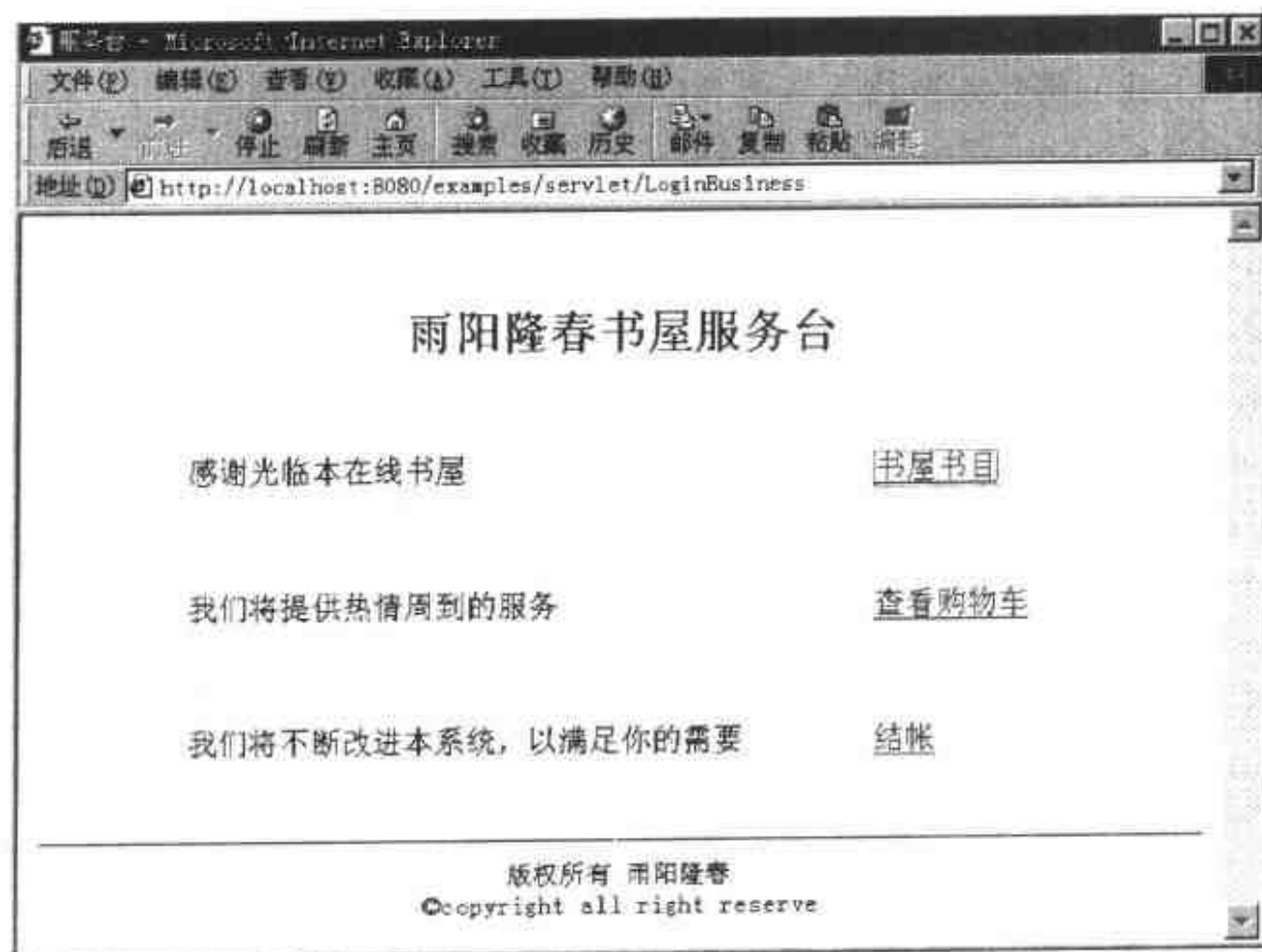


图 14-10 服务台

代码如下:

```

<%@ page contentType="text/html; charset = gb2312" %>
<html>
<head>
<title>服务台</title>
<meta http-equiv="Content-Type" content="text/html; charset = gb2312">

```





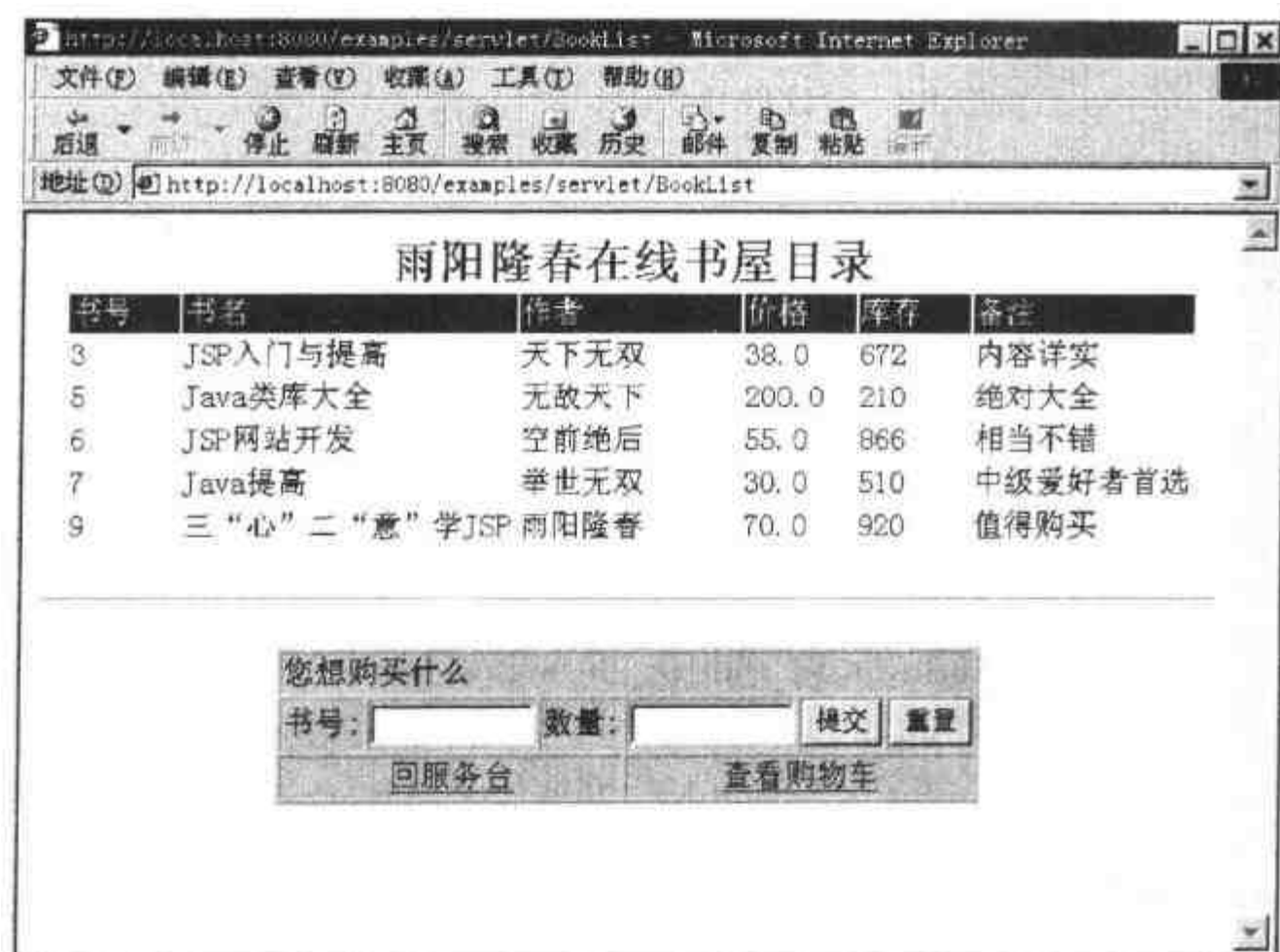


图 14-11 书屋的书目效果图

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

```
public class BookList extends HttpServlet{
    private Connection con = null;
    private Statement stmt = null;
    private ResultSet rs = null;
    private ResultSetMetaData rsmd = null;
    int columnCount = 0;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException{
        try{
            stmt = con.createStatement();
            String sqlstring = "SELECT * FROM BookDetails";
            rs = stmt.executeQuery(sqlstring);
            rsmd = rs.getMetaData();
            columnCount = rsmd.getColumnCount();
            response.setContentType("text/html;charset = gb2312");
            PrintWriter out = response.getWriter();
            out.println("< HTML>");
            out.println("< HEAD>");
            out.println("< TITLE></TITLE>");
```

```
out.println("</HEAD>");
out.println("</HTML>");
out.println("<div align = 'center'>");
out.println("<font size = 5 color = ' # 990099' ><b>雨阳隆春在线  
书屋目录</b></font>");
out.println("</div align = 'center'>");
out.println("<table align = 'center'>");
out.println("<tr bgcolor = ' # 006633' valign = 'middle'>");
out.println("<td width = '10%'><font color = 'white'>书号</font>  
</td>");
out.println("<td width = '30%'><font color = 'white'>书名</font>  
</td>");
out.println("<td width = '20%'><font color = 'white'>作者</font>  
</td>");
out.println("<td width = '10%'><font color = 'white'>价格</font>  
</td>");
out.println("<td width = '10%'><font color = 'white'>库存</font>  
</td>");
out.println("<td width = '20%'><font color = 'white'>备注</font>  
</td>");
out.println("</tr>");
while(rs.next()){
    out.println("<tr bgcolor = ' # FFFFCC'>");
    out.println("<td width = '10%'>");
    out.println(rs.getLong(1));
    out.println("</td>");
    out.println("<td width = '30%'>");
    out.println(rs.getString(2));
    out.println("</td>");
    out.println("<td width = '20%'>");
    out.println(rs.getString(3));
    out.println("</td>");
    out.println("<td width = '10%'>");
    out.println(rs.getFloat(4));
    out.println("</td>");
    out.println("<td width = '10%'>");
    out.println(rs.getLong(5));
    out.println("</td>");
    out.println("<td width = '20%'>");
```

```

        out.println(rs.getString(6));
        out.println("</td>");
        out.println("</tr>");
    }
    out.println("</table><p>");
    out.println("<hr width='750' align='center'>");
    out.println("<form method='post' action='/servlet/ReceiptData'>");
    out.println("<table bgcolor='#99CC66' border='1' align='center'>");
    out.println("<tr>");
    out.println("<td colspan='6'>您想购买什么</td>");
    out.println("</tr>");
    out.println("<tr>");
    out.println("<td>书号:</td>");
    out.println("<td>");
    out.println("<input type='text' name='BookID' size='10' maxlength="
= '16'>");
    out.println("</td>");
    out.println("<td>数量:</td>");
    out.println("<td>");
    out.println("<input type='text' name='Amount' size='10' max-
length='16'>");
    out.println("</td>");
    out.println("<td>");
    out.println("<input type='submit' name='Submit' value='提交'>");
    out.println("</td>");
    out.println("<td>");
    out.println("<input type='reset' name='Reset' value='重置'>");
    out.println("</td>");
    out.println("</tr>");
    out.println("<tr><td align='center' colspan='3'>
<a href='/examples/book/chapter14/business/informationdesk.jsp'>
回服务台</a></td>");
    out.println("<td align='center' colspan='3'>
<a href='/servlet/ShowCart'>查看购物车</a></td></tr>");
    out.println("</table>");
    out.println("</form>");
    out.println("</body>");
    out.println("</html>");
}

```

```
        catch(Exception e){
            System.out.println(e.getMessage());
        }
    }

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

    public void init() throws ServletException{
        try{
            String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
            Class.forName(driver).newInstance();
            String connectionURL = "jdbc:odbc:Business";
            con = DriverManager.getConnection(connectionURL);
        }
        catch(Exception e){
            System.out.println("error: " + e.getMessage());
        }
    }

    public void destroy(){
        try{
            con.close();
        }
        catch(Exception e){
            System.out.println("error: " + e.getMessage());
        }
    }
}
```

这里覆盖(重载)了 `init()` 方法,在其中建立了对数据库的连接。请注意,这是 `init()` 方法的典型应用。

## 2. 后台处理代码

选购一本书,观察有没有响应。这个页面的后台处理代码如下:

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

public class ReceiptData extends HttpServlet{
    private Connection con = null;
    private Statement stmt = null;
    private ResultSet rs = null;

    public void init() throws ServletException
    {
        try{
            String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
            Class.forName(driver).newInstance();
            String connectionURL = "jdbc:odbc:Business";
            con = DriverManager.getConnection(connectionURL);
        }
        catch(Exception e){
            System.out.println("error: " + e.getMessage());
        }
    }

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        try{
            HttpSession session = request.getSession(true);
            ShoppingCart cart = (ShoppingCart)session.getAttribute(session.getId());
            if(cart == null){
                cart = new ShoppingCart();
                session.setAttribute(session.getId(), cart);
            }

            String bookidStr = request.getParameter("BookID");
            String amountStr = request.getParameter("Amount");
            long bookidL = Long.parseLong(bookidStr);
            long amountL = Long.parseLong(amountStr);
```

```
stmt = con.createStatement();
String sqlstring = "SELECT * FROM BookDetails";
sqlstring = sqlstring + " " + "WHERE BookID=" + bookidStr + " " + "AND
Amount >=" + amountL;
rs = stmt.executeQuery(sqlstring);
rs.next();
long tpbookid = rs.getLong(1);
String tptitle = rs.getString(2);
String tpauthor = rs.getString(3);
float tpprice = rs.getFloat(4);
long tpamount = rs.getLong(5);
String tpdescription = rs.getString(6);

BookDetails bookdetails = cart.getItem(tpbookid);
if(bookdetails != null){
    amountL = amountL + bookdetails.getAmount();
    cart.removeItem(tpbookid);
}
bookdetails = new
BookDetails(tpbookid, tptitle, tpauthor, tpprice, amountL, tpdescription);
cart.addItem(bookdetails);
session.setAttribute(session.getId(), cart);

RequestDispatcher
dispatcher = getServletContext().getRequestDispatcher("/servlet/ShowCart");
if(dispatcher == null){
    response.sendError(response.SC_NO_CONTENT);
    System.out.println("error: " + response.SC_NO_CONTENT + "转
发异常, 请检查你的目的路径是否正确或是否存在");
}
dispatcher.forward(request, response);

}
catch(Exception e){
    request.setAttribute("error", e);
    RequestDispatcher
dispatcher = getServletContext().getRequestDispatcher("/examples/book/chapter14/busi-
ness/error.jsp");
    if(dispatcher == null){
```

```

        response.sendError(response.SC_NO_CONTENT);
        System.out.println("error: " + response.SC_NO_CONTENT + "转
发异常,请检查你的目的路径是否正确或是否存在");
    }
    dispatcher.forward(request,response);
    System.out.println(e.getMessage());
}

}

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

public void destroy(){
    try{
        con.close();
    }
    catch(Exception e){
        System.out.println("error: " + e.getMessage());
    }
}

}

```

这个购物车可以识别同名书籍,为此只增加了下面几行代码:

```

BookDetails bookdetails= cart.getItem(tpbookid);
if(bookdetails!= null){
    amountL= amountL+ bookdetails.getAmount();
    cart.removeItem(tpbookid);
}

```

效果却大不一样,请参见后面的购物车显示系统。

还请注意这里的错误处理代码:

```

catch(Exception e){
    request.setAttribute("error",e);
    RequestDispatcher

```

```

dispatcher = getServletContext ( ). getRequestDispatcher ("/examples/book/chapter14/busi-

```



```
ness/error.jsp");  
    if(dispatcher == null){  
        response.sendError(response.SC_NO_CONTENT);  
        System.out.println("error: " + response.SC_NO_CONTENT + "转  
发异常,请检查你的目的路径是否正确或是否存在");  
    }  
    dispatcher.forward(request,response);  
    System.out.println(e.getMessage());  
}
```

这是一种相当不错的方式,将各种异常转发到一个专门进行错误处理的页面中去。假设我们没有选购任何东西时提交表单,那么将出现一个错误处理页面。

### 3. 后台错误处理

效果如图 14-12 所示。

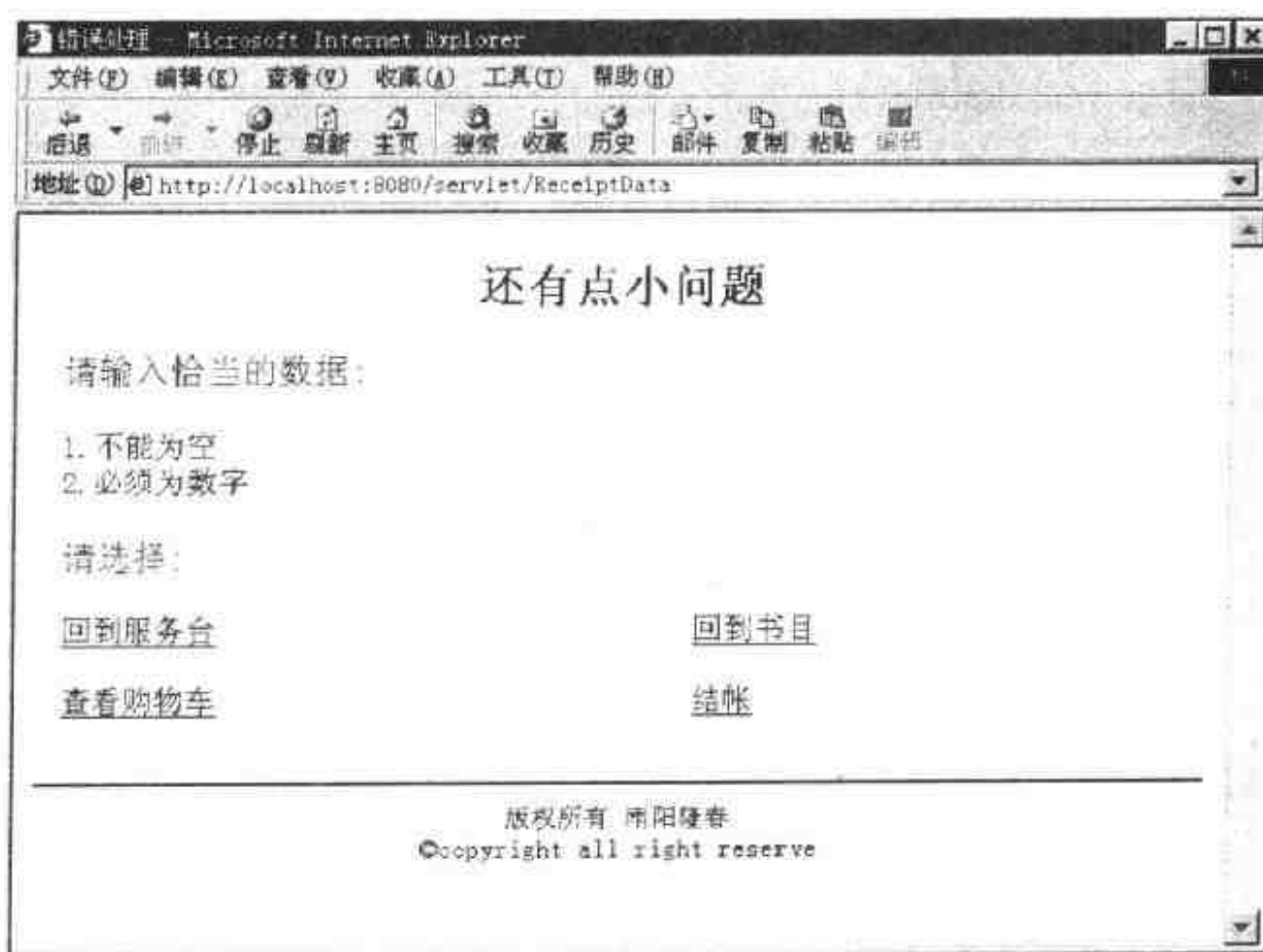


图 14-12 错误处理页面

这是一个 JSP 文件,它根据不同的错误,进行不同的处理。代码如下:

```
<%@ page contentType="text/html; charset=gb2312" isErrorPage="true"%>  
<HTML>  
<HEAD>  
<TITLE>错误处理</TITLE>  
</HEAD>  
<body bgcolor="#FFFFCC">
```

```

<table width="750" border="0" cellspacing="10" align="center" cellpadding="6"
height="280">
  <tr>
    <td colspan="2">
      <div align="center"><font size="5" color="#990099"><b>还有点小问
题</b></font></div>
      <%
      try{
        Object error=request.getAttribute("error");
        if(error.toString().startsWith("java.lang.NumberFormatException")){
          out.println("<p><font size='4' color='#FF9900'>请输入恰当的
数据:</font></p>");
          out.println("<font color='#006633'>1. 不能为空</font><br>");
          out.println("<font color='#006633'>2. 必须为数字</font><br>");
        }
        else{
          if(error.toString().startsWith("java.sql.SQLException")){
            out.println("<p><font size='4' color='#FF9900'>找不到满足
您条件的书:</font></p>");
            out.println("<font color='#006633'>1. 书号不存在</font><br>");
            out.println("<font color='#006633'>2. 你需要的量大于库存
</font><br>");
          }
          else{
            out.println("<p><font size='4' color='#FF9900'>还有其他错
误,请仔细核对,稍后再提交</font><br>");
          }
        }
      }
      catch(Exception e){
        out.println(e.toString());
      }
      %> <br>
      <p><font color="#FF9900" size="4">请选择:</font></p>
    </td>
  </tr>
  <tr>
    <td><a href="/examples/book/chapter14/business/informationdesk.jsp">回到

```

```

服务台</a></td>
    <td><a href="/servlet/BookList">回到书目</a></td>
</tr>
<tr>
    <td><a href="/servlet/ShowCart">查看购物车</a></td>
    <td><a href="/servlet/CheckOut">结帐</a></td>
</tr>
</table>
<br>
<table width="750" border="0" cellspacing="0" cellpadding="0" align="center">
    <tr height="2" bgcolor="# 666666">
        <td></td>
    </tr>
    <tr height="12">
        <td></td>
    </tr>
    <tr>
        <td>
            <div align="center"><font size="2">版权所有 雨阳隆春</font></div>
        </td>
    </tr>
    <tr>
        <td>
            <div align="center"><font size="2"> &copy; copyright all right reserve
</font></div>
        </td>
    </tr>
</table>

</body>
</HTML>

```

上面,基本上将书屋的购物系统走了一遍,下面看看购物车显示。

## 14.2.4 购物车显示

### 1. 页面效果图

假设选择了某本书,例如 9 号书,并提交了表单,那么系统将自动重导到购物车显示

系统。请看类似图 14-13 的页面。

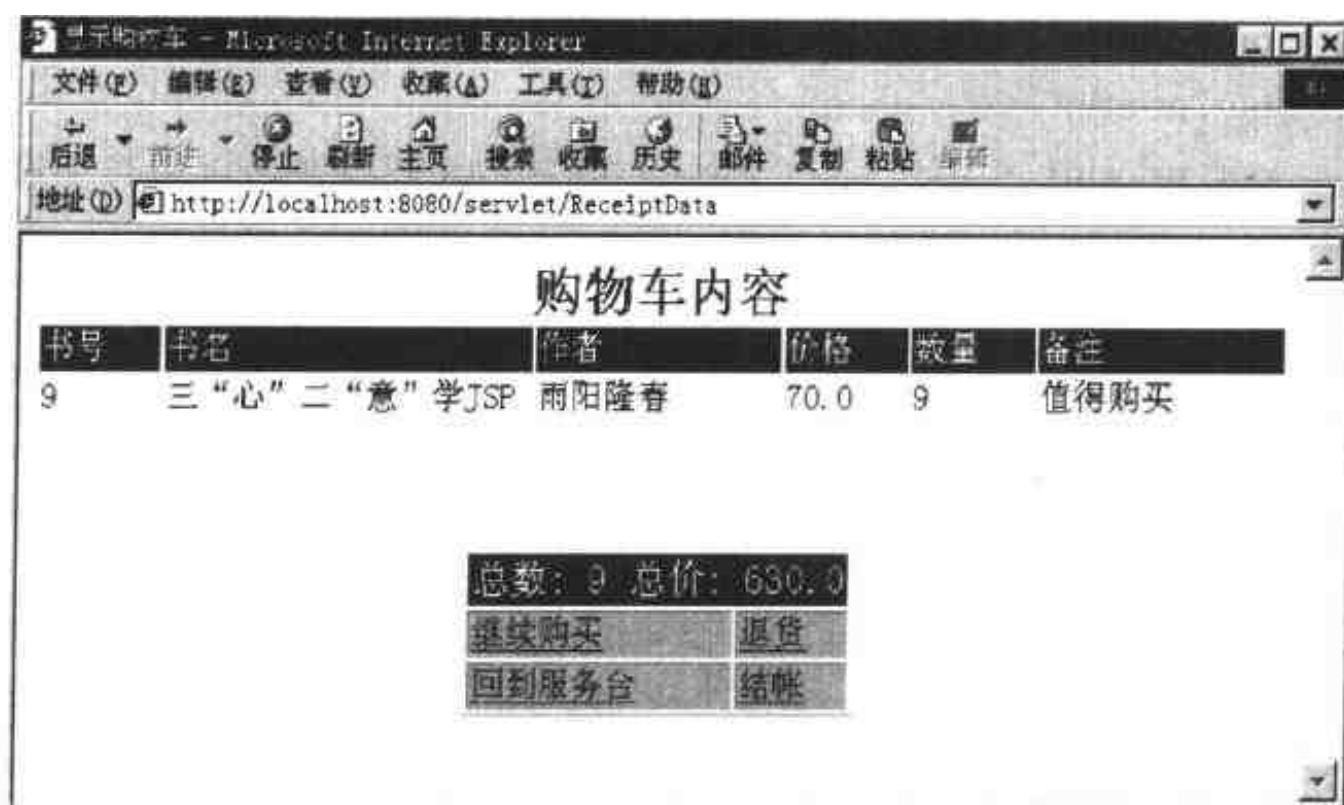


图 14-13 购物车显示

## 2. 源代码如下

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

public class ShowCart extends HttpServlet{

    public void doGet(HttpServletRequest request,HttpServletResponse response)
        throws IOException, ServletException
    {
        HttpSession session = request.getSession(true);
        ShoppingCart cart = (ShoppingCart)session.getAttribute(session.getId());
        if(cart == null){
            cart = new ShoppingCart();
            session.setAttribute(session.getId(),cart);
        }

        response.setContentType("text/html;charset = gb2312");
        PrintWriter out = response.getWriter();

        out.println("<HTML>");
        out.println("<HEAD>");
        out.println("<TITLE>显示购物车</TITLE>");
```

```

        out.println("</HEAD>");
        out.println("</HTML>");
        out.println("<div align = 'center' >");
        out.println("< font size = 5 color = ' # 990099' > < b > 购物车内容 </b >
</font>");
        out.println("</div align = 'center' >");
        out.println("< table width = '750' align = 'center' >");
        out.println("< tr bgcolor = ' # 006633' valign = 'middle' >");
        out.println("< td width = '10%' > < font color = 'white' > 书号 </font> </td>");
        out.println("< td width = '30%' > < font color = 'white' > 书名 </font> </td>");
        out.println("< td width = '20%' > < font color = 'white' > 作者 </font> </td>");
        out.println("< td width = '10%' > < font color = 'white' > 价格 </font> </td>");
        out.println("< td width = '10%' > < font color = 'white' > 数量 </font> </td>");
        out.println("< td width = '20%' > < font color = 'white' > 备注 </font> </td>");
        out.println("</tr>");
        BookDetails bookdetails;
        float totalmoney = 0;
        long totalamount = 0;
        for(int i = 0; i < cart.getNumberofItems(); i++ ){
            bookdetails = cart.getItem(i);
            totalamount = totalamount + bookdetails.getAmount();
            totalmoney = totalmoney + bookdetails.getPrice() * bookdetails.getA-
mount();

            out.println("< tr bgcolor = ' # FFFFCC' >");
            out.println("< td width = '10%' >");
            out.println(bookdetails.getBookId());
            out.println("</td>");
            out.println("< td width = '30%' >");
            out.println(bookdetails.getTitle());
            out.println("</td>");
            out.println("< td width = '20%' >");
            out.println(bookdetails.getAuthor());
            out.println("</td>");
            out.println("< td width = '10%' >");
            out.println(bookdetails.getPrice());
            out.println("</td>");
            out.println("< td width = '10%' >");
            out.println(bookdetails.getAmount());
            out.println("</td>");

```

```

        out.println("<td width='20%'>");
        out.println(bookdetails.getDescription());
        out.println("</td>");
        out.println("</tr>");
    }

    out.println("</table>");
    out.println("<p> &nbsp;   </p>");

    out.println("<table border='1' align='center'>");
    out.println("<tr bgcolor=' #006633'><td colspan='6'>");
    out.println("<font size='4' color='white'>");
    out.println("总数:");
    out.println(totalamount);
    out.println("总价:");
    out.println(totalmoney);
    out.println("</font></td></tr>");
    out.println("<tr bgcolor=' #99CC66'>");
    out.println("<td><a href='/servlet/BookList'>继续购买</a></td>");
    out.println("<td><a href='/servlet/SendBackList'>退货</a></td>");
    out.println("</tr>");
    out.println("<tr bgcolor=' #99CC66'>");
    out.println("<td><a href='/examples/book/chapter14/business/informationdesk.jsp'>回到服务台</a></td>");
    out.println("<td><a href='/servlet/CheckOut'>结帐</a></td>");
    out.println("</tr>");
    out.println("</table>");

}

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

}

```

购物车显示其实非常简单,一言以蔽之,就是如何从 session 中获得数据,然后显示出

来即可。在前面提到这个购物车可以识别同名书籍,现在可以看到效果了。假设在图 14-13 基础上,我们再次购买 9 号书,将其购买数量由 9 改为 16,结果如图 14-14 所示。

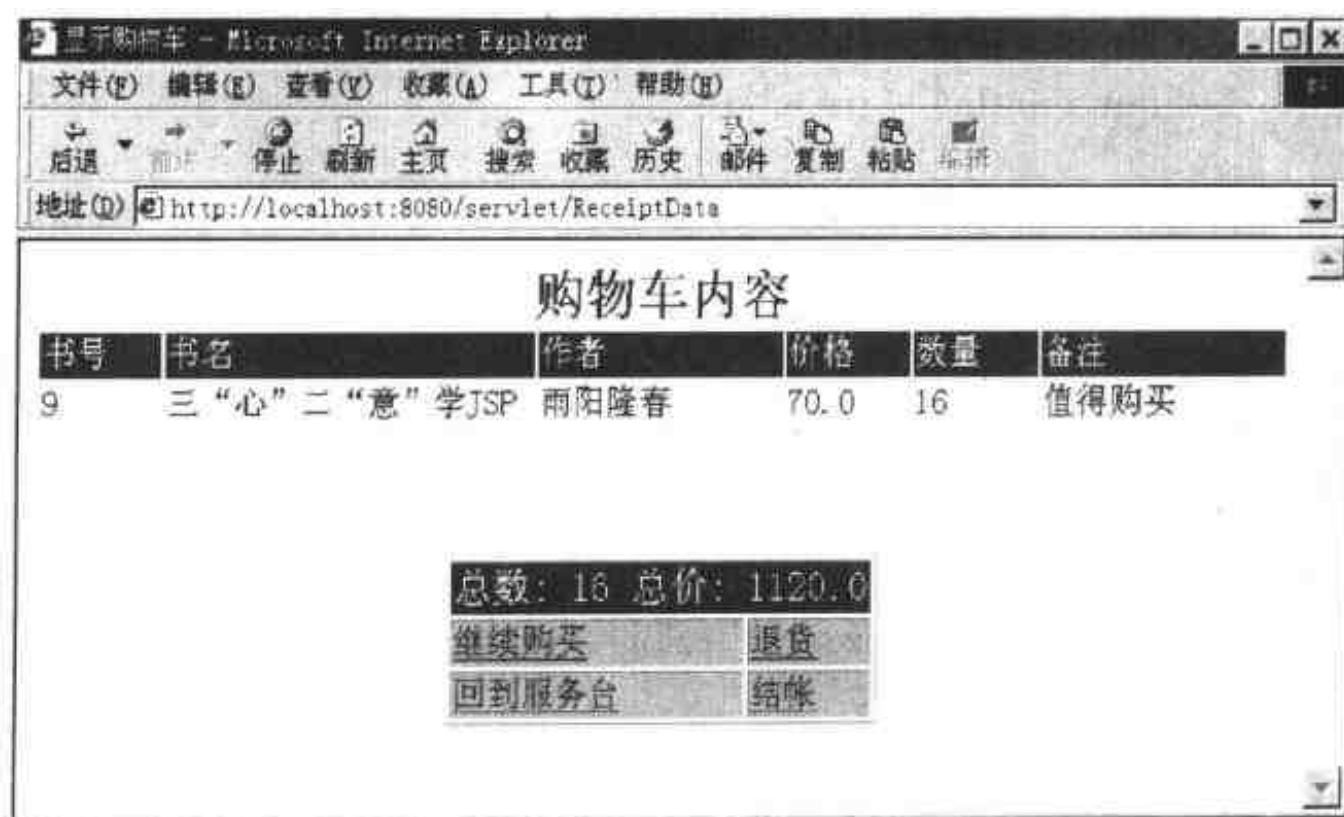


图 14-14 同名书籍的处理

## 14.2.5 退货系统——部分退回

### 1. 页面效果图

在“购物车内容”提供的菜单中选择“退货”,效果如图 14-15 所示。

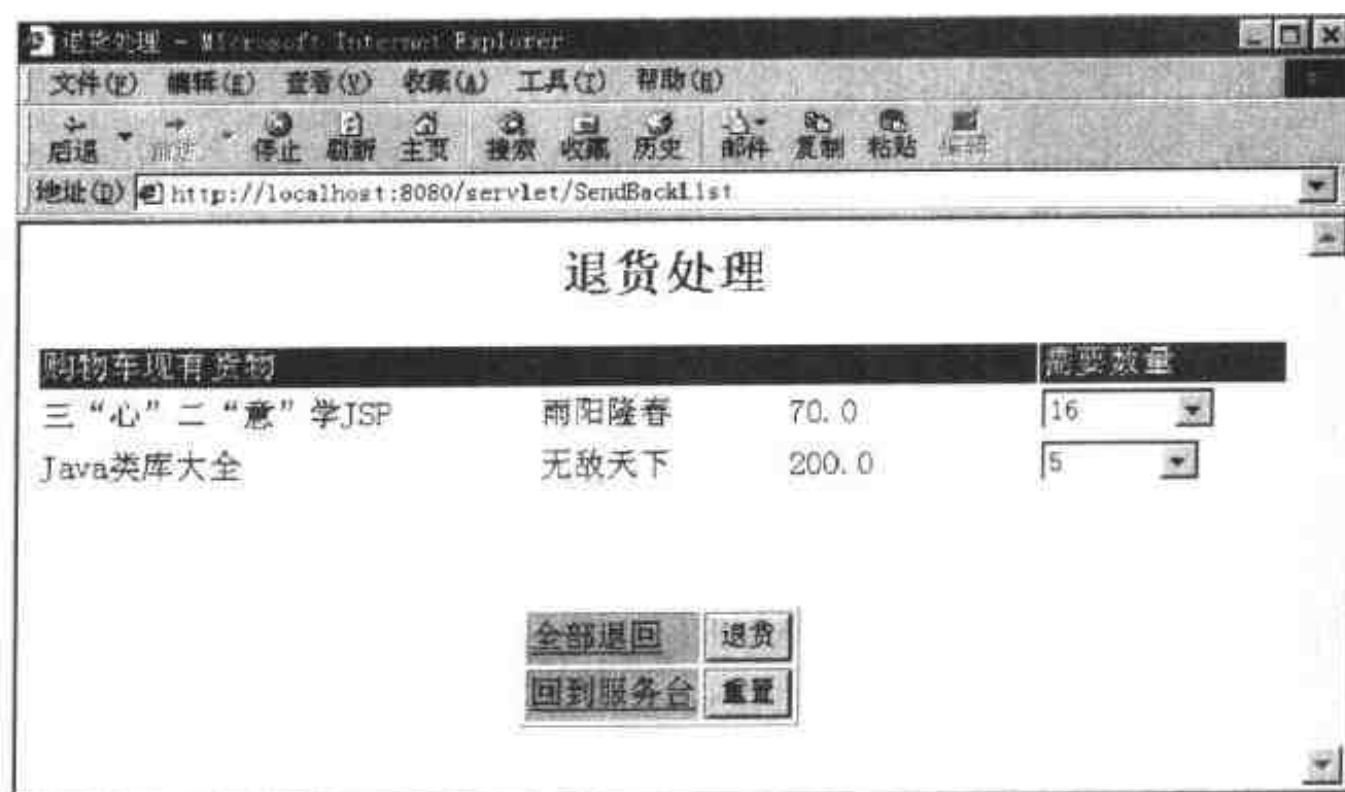


图 14-15 退货系统

退货提供了两种手段:全部退回和部分退回,下面介绍部分退回。

### 2. 部分退回处理代码

```
import java.io.*;
```

```

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

public class SendBackSome extends HttpServlet{

    public void doGet(HttpServletRequest request,HttpServletResponse response)
        throws IOException,ServletException
    {
        HttpSession session = request.getSession(true);
        ShoppingCart cart = (ShoppingCart)session.getAttribute(session.getId());
        if(cart! = null){
            Enumeration e = request.getParameterNames();
            while(e.hasMoreElements()){
                String bookidstr = e.nextElement().toString();
                String amountstr = request.getParameter(bookidstr);
                if(! bookidstr.equals("Submit")){
                    long bookidlong = Long.parseLong(bookidstr);
                    long amountlong = Long.parseLong(amountstr);
                    if(amountlong = = 0){
                        cart.removeItem(bookidlong);
                    }
                    else{
                        cart.updateItem(bookidlong,amountlong);
                    }
                }
            }
        }
        session.setAttribute(session.getId(),cart);

        RequestDispatcher
dispatcher = getServletContext().getRequestDispatcher("/servlet/ShowCart");
        if(dispatcher == null){
            response.sendError(response.SC_NO_CONTENT);
            System.out.println("error: " + response.SC_NO_CONTENT + "转
发异常,请检查你的目的路径是否正确或是否存在");
        }
        dispatcher.forward(request,response);
    }
}

```



```

    }

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

```

### 3. 退回效果图

部分退回页面效果如图 14-16 所示。

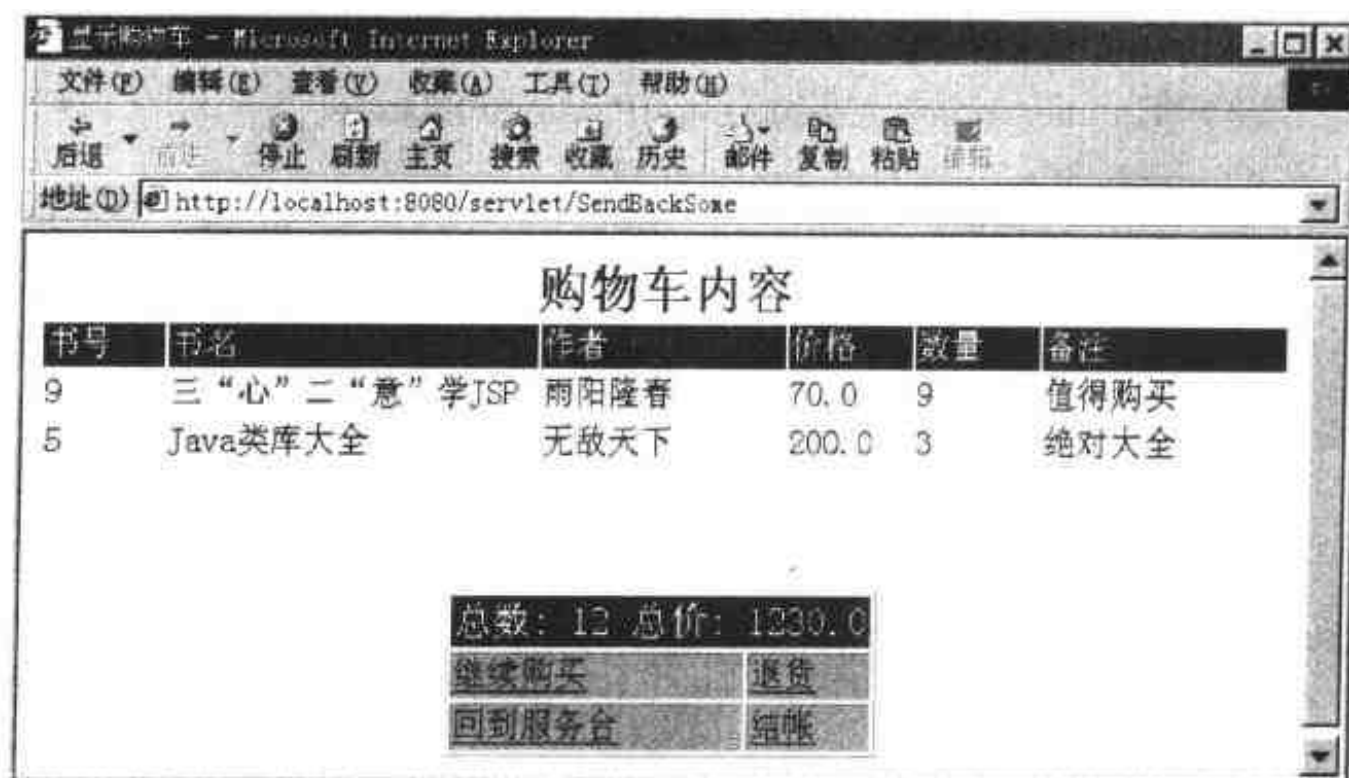


图 14-16 部分退货效果图

## 14.2.6 全部退回

### 1. 效果图

全部退回页面效果如图 14-17 所示。

### 2. 后台处理代码

代码如下：

```

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

public class SendBackAll extends HttpServlet{

```



图 14-17 退回全部书籍的效果图

```

public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
{
    HttpSession session = request.getSession(true);
    ShoppingCart cart = (ShoppingCart)session.getAttribute(session.getId());
    if(cart != null){
        cart.removeAllItem();
        session.setAttribute(session.getId(), cart);
    }

    RequestDispatcher
dispatcher = getServletContext().getRequestDispatcher("/servlet/ShowCart");
    if(dispatcher == null){
        response.sendError(response.SC_NO_CONTENT);
        System.out.println("error: " + response.SC_NO_CONTENT + "转
发异常, 请检查你的目的路径是否正确或是否存在");
    }
    dispatcher.forward(request, response);
}

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

```

}

## 14.2.7 结帐系统

### 1. 页面效果图

页面效果如图 14-18 所示。



图 14-18 结帐系统效果图

### 2. 后台处理代码

代码如下：

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

```
public class CheckOut extends HttpServlet{
```

```
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        HttpSession session = request.getSession(true);
        ShoppingCart cart = (ShoppingCart)session.getAttribute(session.getId());
        if(cart == null){
            cart = new ShoppingCart();
            session.setAttribute(session.getId(), cart);
        }
    }
```

```
if(cart.getItemCount() != 0){
    response.setContentType("text/html;charset = gb2312");
    PrintWriter out = response.getWriter();

    out.println("<HTML>");
    out.println("<HEAD>");
    out.println("<TITLE>结帐</TITLE>");
    out.println("</HEAD>");
    out.println("</HTML>");
    out.println("<div align = 'center'>");
    out.println("<font size = 5 color = ' # 990099'><b>您的帐单</b>");
    out.println("</font>");

    out.println("</div align = 'center'>");
    out.println("<table width = '750' align = 'center'>");
    out.println("<tr bgcolor = ' # 006633' valign = 'middle'>");
    out.println("<td width = '40%'><font color = 'white'>书名</font>");
    out.println("</td>");

    out.println("<td width = '20%'><font color = 'white'>单价</font>");
    out.println("</td>");

    out.println("<td width = '10%'><font color = 'white'>数量</font>");
    out.println("</td>");

    out.println("<td width = '30%'><font color = 'white'>小计</font>");
    out.println("</td>");

    out.println("</tr>");
    BookDetails bookdetails;
    float subtotalmoney = 0;
    float totalmoney = 0;
    long totalamount = 0;

    for(int i = 0; i < cart.getItemCount(); i++){
        bookdetails = cart.getItem(i);
        totalamount = totalamount + bookdetails.getAmount();
        subtotalmoney = bookdetails.getPrice() * bookdetails.getAmount();
        totalmoney = totalmoney + subtotalmoney;
        out.println("<tr bgcolor = ' # FFFFCC'>");
        out.println("<td width = '40%'>");
        out.println(bookdetails.getTitle());
        out.println("</td>");
        out.println("<td width = '20%'>");
```



```

dispatcher = getServletContext ( ). getRequestDispatcher ("/examples/book/chapter14/busi-
ness/welcomeagain.jsp");
    if(dispatcher == null){
        response.sendError(response.SC_NO_CONTENT);
        System.out.println("error: " + response.SC_NO_CONTENT + "
转发异常,请检查你的目的路径是否正确或是否存在");
    }
    dispatcher.forward(request,response);
}

}

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

}

```

输入信用卡号,当然在本例中输入任意的字符串均可。接下来,将看到这样的欢送页面,如图 14-19 所示。

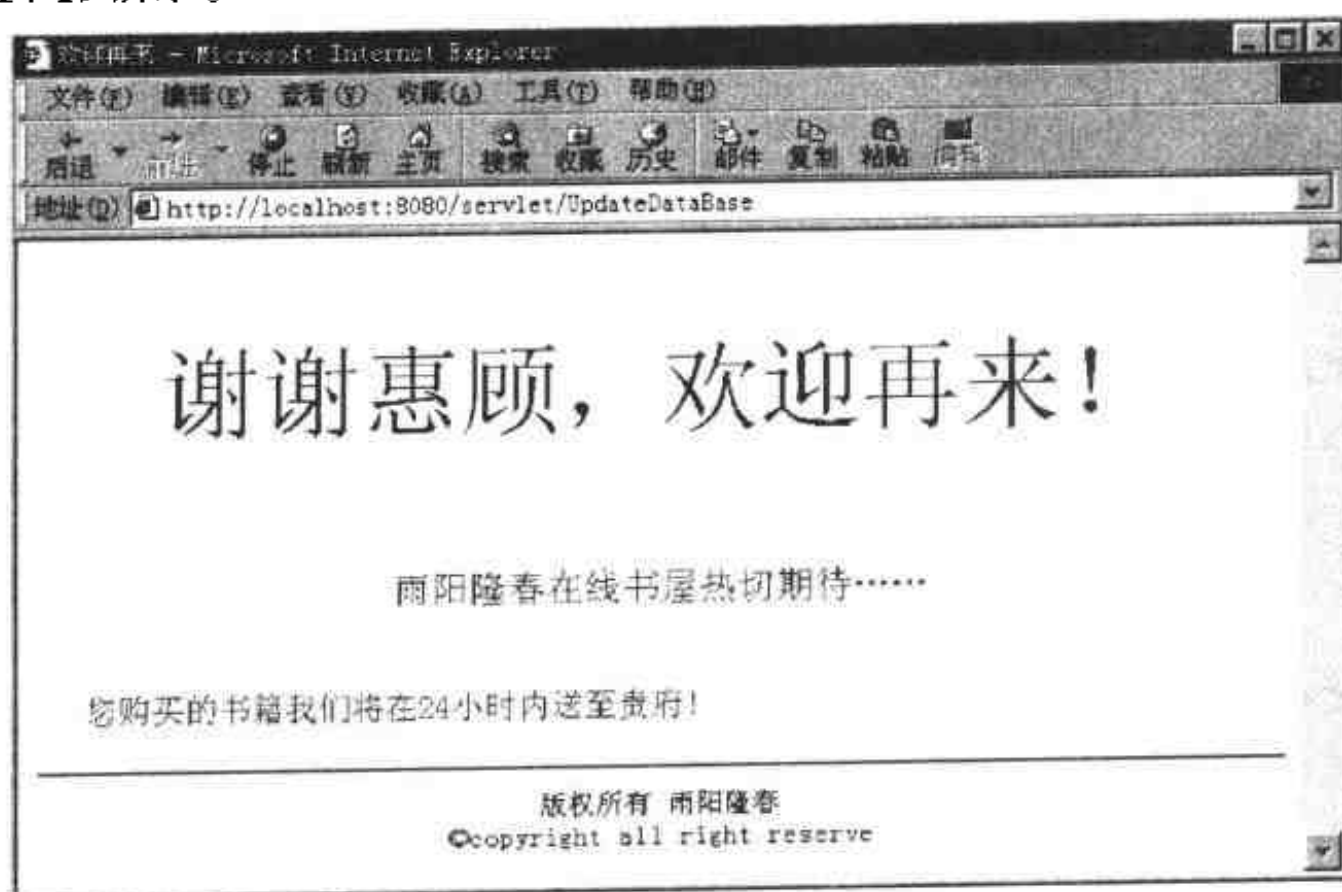


图 14-19 退出系统的欢送词

这是一个 JSP 文件,代码如下:

```

<%@ page contentType="text/html; charset = gb2312" %>
<html><title>欢迎再来</title>

```



```
</body>
</html>
```

### 14.2.8 数据库更新

用户购物之后,还应进行数据库更新。数据库更新的代码如下:

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

public class UpdateDataBase extends HttpServlet{
    private Connection con = null;
    private Statement stmt = null;

    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        try{
            stmt = con.createStatement();
            HttpSession session = request.getSession(true);
            ShoppingCart cart = (ShoppingCart)session.getAttribute(session.getId());
            if(cart != null){
                for(int i = 0; i < cart.getItemCount(); i++){
                    BookDetails bookdetails = cart.getItem(i);
                    long bookid = bookdetails.getBookId();
                    long amount = bookdetails.getAmount();
                    String updatestring = null;
                    updatestring = "UPDATE BookDetails SET Amount = Amount
- " + amount + " " + "WHERE BookID = " + bookid;
                    stmt.executeUpdate(updatestring);
                    System.out.println("DataBase update successfully");
                }
                request.setAttribute("hasBook", "true");
            }
            session.invalidate();
            stmt.close();
            RequestDispatcher
dispatcher = getServletContext().getRequestDispatcher("/examples/book/chapter14/busi-
```



```
ness/welcomeagain.jsp");
    if(dispatcher == null){
        response.sendError(response.SC_NO_CONTENT);
        System.out.println("error: " + response.SC_NO_CONTENT + "
转发异常, 请检查你的目的路径是否正确或是否存在");
    }
    dispatcher.forward(request, response);

}
catch(Exception e){
    System.out.println(e.getMessage());
}
}

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

public void init() throws ServletException{
    try{
        String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
        Class.forName(driver).newInstance();
        String connectionURL = "jdbc:odbc:Business";
        con = DriverManager.getConnection(connectionURL);
    }
    catch(Exception e){
        System.out.println("error: " + e.getMessage());
    }
}

public void destroy(){
    try{
        con.close();
    }
    catch(Exception e){
        System.out.println("error: " + e.getMessage());
    }
}
```

当再次进入系统时,数据库已经更新。

至此,我们将书屋完全实现。应该说还是不错的,希望读者朋友都有很大的收获。

## 14.3 聊天室

聊天室是每一个网站都具有的功能之一,本部分我们用数据库实现聊客登陆信息管理,用 application 对象和 session 对象来实现聊天室程序。总体上可分为两大部分:一、聊客的信息管理,二、聊天室的实现。下面是其功能的具体实现。

### 14.3.1 聊客信息管理

#### 1. 用户登录程序

```
login_init.jsp
<html>
<head>
<title>登录聊天室</title>
<body bgcolor="ffffcc">
<%@page contentType="text/html;charset=gb2312"%>
<jsp:include page="../template/header1.jsp" flush="true"/>
<p align="center"><font color="#000000"><h><font color="#666666" size
="4">登录聊天室</font></b>
</font></p>
<form method="POST" action="login.jsp">
  <div align="center"><center>
    <table border="0" width="350" height="230" cellpadding="2" cellspacing="
"3" bgcolor="007733">
      <tr>
        <td width="70">
          <div align="center"><font color="ffffff">昵称</font></div>
        </td>
        <td width="200">
          <input type="text" name="username" size="20"><font color="red">
* </font>
```



```
<title>登录聊天室
</title>
</head>
<body bgcolor = 'ffffcc' >

<%@ page import = "java.util. *" %>
<%@ include file = "obj.jsp"%>
//指定页面为中文字符
<%@page contentType="text/html; charset = gb2312"%>
<jsp:include page = "../template/header1.jsp" flush = "true"/>

<%
String username = new String(request.getParameter("username"));
String temchater = new String(request.getParameter("chatername"));
String psw = request.getParameter("psw");
boolean login = true;
%>
<%
    //进行内码转换
    byte[] temp _ busername;
    String temp _ suserername;
    temp _ suserername = username;
    temp _ busername = temp _ suserername.getBytes("ISO - 8859 - 1");
    username = new String(temp _ busername);

    byte[] temp _ hchatername;
    String temp _ schatername;
    temp _ schatername = temchater;
    temp _ bchatername = temp _ suserername.getBytes("ISO - 8859 - 1");
    temchater = new String(temp _ busername);

String tempuser;
Vector temp = new Vector();
int cn = 0;
String erromsg = "你填写时:<br>";

java.sql.Connection sqlconn = null;
java.sql.PreparedStatement mysql = null;
java.sql.ResultSet myset = null;
```

```
String _sqlquerystring = "SELECT * FROM logintable where 昵称 = ? and 名字 = ?
and 密码 = ?";
```

```
String _driver = "sun.jdbc.odbc.JdbcOdbcDriver";
Class.forName(_driver).newInstance();
sqlconn = java.sql.DriverManager.getConnection("jdbc:odbc:cbat");
mysql = sqlconn.prepareStatement(_sqlquerystring);
mysql.setBytes(1, username.getBytes("GBK"));
mysql.setBytes(2, temchater.getBytes("GBK"));
mysql.setBytes(3, psw.getBytes("GBK"));
// 用于处理数据库的中文问题
myset = mysql.executeQuery();
if (! myset.next())
{
    login = false;
    cn = cn + 1;
    errmsg = errmsg + cn + "." + "没有该纪录, 请先注册<br>";
    errmsg = errmsg + "<a href = 'register.jsp'>现在就注册</a><br><br>";
    myset.close();
    mysql.close();
    sqlconn.close();
}
```

```
if(getServletContext().getAttribute(user) != null){
    temp = (Vector)getServletContext().getAttribute(user);

    for(int i=0;i<temp.size();i++){
tempuser = temp.elementAt(i).toString();
        if(username.equals(tempuser)){
            login = false;

cn = cn + 1;
errmsg = errmsg + cn + "." + "昵称相同<br><br>";
    }
    }
}
```

```
if (username.equals("") || temchater.equals("") || psw.equals("")){
    login = false;
cn = cn + 1;
errmsg = errmsg + cn + "." + "未输入完整<br><br>";
```

```

|
if (username.equals("大家") || username.equals("每一个人") || username.equals
("all")){

    cn = cn + 1;
    errmsg = errmsg + cn + "." + "该昵称禁用<br><br>";
    |
    if(login == true){
        temp.addElement(username);
        getServletContext().setAttribute(user,temp);
        //在用户向量中加入新的用户
        session.putValue("username",username);
        //设置 session 记录用户名,添加欢迎信息到消息向量里
        Vector tempmsg = new Vector();
        tempmsg.addElement("<font size = '3' color = 'red'>" + username + "自
[ " + request.getRemoteAddr() + "],欢迎你的到来! </font><br>");
        getServletContext().setAttribute(chatmsg,tempmsg);
        //添加信息到信息向量中
        Vector top = new Vector();
        top.addElement("足球");
        //添加主题
        getServletContext().setAttribute(chatobj,top);
        // 送出欢迎页面
        out.println("<p align = 'center'><font color = ' # 666666' size = '3'>大话西游
</font></p>");
        out.println("<hr width = '90%' color = ' # 666666'>");
        out.println("<div align = 'center'><center>");
        out.println("<table border = '0' width = '86%' height = '190'>");
        out.println("<tr>");
        out.println("<td width = '4%' height = '13'></td>");
        out.println("<td width = '100%' height = '23' bgcolor = ' # 006633'></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td width = '4%' height = '13'></td>");
        out.println("<td width = '96%' height = '13' bgcolor = ' # 007833'></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td width = '4%' height = '13'></td>");
        out.println("<td width = '96%' height = '11' bgcolor = ' # 008a33'></td>");
    }
}

```

```

        out.println("</tr>");
        out.println("<tr>");
        out.println("<td width = '4%' height = '7'></td>");
        out.println("<td width = '96%' height = '7' bgcolor = '#008a33'></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td width = '4%' height = '13'></td>");
        out.println("<td width = '96%' height = '19' bgcolor = '#009c33'></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td width = '4%' height = '13'></td>");
        out.println("<td width = '96%' height = '20' bgcolor = '#00ae33'></td>");
        out.println("</tr>");
        out.println("<tr>");
        out.println("<td width = '4%' height = '13'></td>");
        out.println("<td width = '96%' height = '22' bgcolor = '#00bf33'></td>");
        out.println("</tr>");
        out.println("</table>");
        out.println("</center></div>");
        out.println("<p align = 'center'><a href = 'main.html' target = '_top'>点  

击进入...</a></p>");
    }
    else
    {
        out.println("<br><br>");
        out.println("<hr><br>");
        out.println("<center><font color = '006633' size = '3'>" + errmsg);
        out.println("<br><br>");
        out.println("<a href = 'login _ init.jsp'>返回</a></font></center>");
    }
}
%>
<%
    if (login == true){
//用一个哈希表来记录用户在聊天室呆的时间
        Hashtable userLife = new Hashtable();

//用一个哈希表记录在聊用户的 IP 地址
        Hashtable userIP = new Hashtable();

//取得用户登录时间

```

```

        Date loginTime = new Date();

        //记录登录时间
        userLife.put((String)session.getValue("username"), new
Long(loginTime.getTime()));

        //记录用户 IP 地址
        userIP.put((String)session.getValue("username"), request.getRemoteAddr());

        //将登录时间和 IP 地址添加到哈希表
        getServletContext().setAttribute(usetime, userLife);
getServletContext().setAttribute(userid, userIP);

        % >
    </body>
</html>

```

在 login.jsp 程序中对用户提交的信息处理时,用到了数据库,对用户填写的信息进行判断,为避免在数据库中出现乱码,在处理数据的中文问题时,用 prepareStatement 方法得到 PreparedStatement 对象。PreparedStatement 对象用于发送带有一个或多个参数的 sql 语句,它继承了 Statement 对象的常用的方法。用此对象可以先将用户提交参数进行内码转换后,再到库中进行查询。如果有该用户信息,就定义一个临时向量存放用户信息,用哈希表存放用户相对应的 IP 地址和时间。用 application 和 session 对象的 setAttribute 方法和 getValue 方法将其保存,同时送出欢迎界面。当库中没有用户信息时,将给出错误信息引导用户进入注册界面。

## 2. 用户注册程序

```

register.jsp
<html>
    <meta http _ equiv = "conten _ type" content = "text/html";
    charset = "GB2312">
<head>
    <title>注册</title>
</head>
    <body bgcolor = 'ffffcc'>
<% @page contentType = "text/html; charset = gb2312" %>
<jsp:include page = "../template/header1.jsp" flush = "true"/>

    <div align = "center">
    <br>

```



该程序同样是为用户填写信息时提供输入界面,其结果将提交给 register\_result 程序处理。

register	result
----------	--------

© 2006 The Authors  
Journal compilation © 2006 Blackwell Publishing Ltd

```
String _password = request.getParameter("psw");
String _passwordagain = request.getParameter("pswagain");

//获取用户提交的参数
byte[] temp _busername;
String temp _suserername;
temp _suserername = _username;
temp _busername = temp _suserername.getBytes("ISO-8859-1");
_username = new String(temp _busername);

byte[] temp _bchatername;
String temp _schatername;
temp _schatername = _chatername;
temp _bchatername = temp _suserername.getBytes("ISO-8859-1");
_chatername = new String(temp _bchatername);

if(_username.equals("") || _chatername.equals("") || _passwordagain.equals("") || !_password.equals(_passwordagain))
    String _redirectURL = "/examples/book/chapter14/chat/register.jsp";
    response.sendRedirect(response.encodeURL(_redirectURL));
}
else{
    String _sqlquerystring = "SELECT * FROM logintable where 昵称 = ?";
    String _driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    Class.forName(_driver).newInstance();
    String _connectionURL = "jdbc:odbc:chat";
    Connection _con = null;
    _con = DriverManager.getConnection(_connectionURL);
    PreparedStatement P _stmt = null;
    P _stmt = _con.prepareStatement(_sqlquerystring);
    P _stmt.setBytes(1, _username.getBytes("GBK"));
    ResultSet _rs = null;
    _rs = P _stmt.executeQuery();

    if(_rs.next()){
        _rs.close();
        P _stmt.close();
        _con.close();
        String _redirectURL = "/examples/book/chapter14/chat/register.jsp";
```

```

        response.sendRedirect(response.encodeURL(_redirectURL));
    }
    else{
        //处理数据库中文问题
        _sqlquerystring = "INSERT INTO logintable (名字,昵称,密码)
values(?,?,?)";

        P_stmt = _con.prepareStatement(_sqlquerystring);
        P_stmt.setBytes(1,_chatername.getBytes("GBK"));
        P_stmt.setBytes(2,_username.getBytes("GBK"));
        P_stmt.setBytes(3,_password.getBytes("GBK"));

        P_stmt.executeUpdate();
        SQLWarning warn = _rs.getWarnings();
        if(warn != null){
            out.println("----- Warning -----");
            out.println("<br>");
            while(warn != null){
                out.println("Message:" + warn.getMessage());
                out.println("<br>");
                out.println("SQLState:" + warn.getSQLState());
                out.println("<br>");
                out.println("Vendor error code:");
                out.println(warn.getErrorCode());
                out.println("<br>");
                warn = warn.getNextWarning();
            }
        }

        _rs.close();
        P_stmt.close();
        _con.close();
        String _redirectURL = "/examples/book/chapter14/chat/login_init.jsp";
        response.sendRedirect(response.encodeURL(_redirectURL));
    }
}

catch(SQLException ex){
    out.println("SQLException caught");
}

```

```

        out.println("<br>");
        while(ex! = null){
            out.println("Message: " + ex.getMessage());
            out.println("<br>");
            out.println("SQLState: " + ex.getSQLState());
            out.println("<br>");
            out.println("ErrorCode: " + ex.getErrorCode());
            out.println("<br>");
            ex = ex.getNextException();
        }
    }
% >
</body>
</html>

```

对注册信息的处理同样会遇到数据库的中文问题,相应的处理方法已在登录处理部分讲过,在此再次提醒一下大家。该程序用 try ,catch 结构来捕获程序运行时的错误,作为数据库开发者应养成这一习惯。整个程序的流程是先读取数据库,再通过查询结果的空或不空来判断是否可以入库,如果注册成功,将用 response.sendRedirect 方法重定位到登录界面。如果未成功同样用 response.sendRedirect 方法重定位到注册页面。

### 14.3.2 聊天室的实现

当用户登录成功后将进入聊天室主页面,该页面可分为四帧:页头,发送聊天信息、聊天室状态和显示聊天信息,具体代码如下:

```

Main.html
<html>
<head>
<title>
</title>
</head>
<frameset cols="90%,1*" border="0" framespacing="0" frameborder="NO">
    <frameset rows="8%,1*" border="0" framespacing="1" frameborder="no">
        <frame src="header1.jsp" marginwidth="10" marginheight="10" scrolling="no"
frameborder="0">
        <frameset rows="80%,100*" cols="*">
            <frame src="showchat.jsp" marginwidth="10" marginheight="10" scrolling="
"auto" frameborder="0">
            <frame src="sendmsg_init.jsp" marginwidth="10" marginheight="10" scroll-
ing="no" frameborder="0">

```

```

</frameset>
</frameset>
< frame src = "chatroom.jsp" marginwidth = "10" marginheight = "10" scrolling = "
no" frameborder = "0">
</frameset>
</html>

```

程序中用 frame 标记将 header1.jsp(页头)、showchat.jsp(显示聊天信息程序)、sendmsg.jsp(发送聊天信息程序)、chatroom.jsp(显示聊天室信息程序)四部分组合在一起。整个聊天室都是基于 application 对象实现的。在第 7 章我们谈到 application 对象,该对象在整个应用程序中有效,并可以在其中保存任何类型的数据。因此,我们利用 sendmsg.jsp 程序将用户想发送的信息存储在向量中,用 application 对象的 setAttribute()方法将向量添加到 application 中保存;利用 showchat.jsp 程序的 getAttribute()方法将向量取出并显示其中的信息。

下面分别是发送聊天信息程序、显示聊天信息程序、显示聊天室信息程序和离开聊天室程序等具体的代码:

### 1. 发送聊天信息程序

```

sendmsg.jsp
<html>
<meta http-equiv="refresh" content="100">
<title>
</title>
<head>

<script language="JavaScript" type="text/javascript">
function setCookie()
{

document.cookie = form1.color.selectedIndex + "*" + form1.whisperto.selectedIndex
+ "*" + form1.action.selectedIndex + "*" + form1.whisper.checked;

}
function setFocus()
{

form1.message.focus();
var cookieValue = document.cookie;
var formValue = cookieValue.split("*");
form1.color.selectedIndex = formValue[ ];
form1.whisperto.selectedIndex = formValue[1];

```

```
form1.action.selectedIndex = formValue[2];

var s_ formValue = formValue[3].split(";");

if (s_ formValue[0] == "false")
{
    form1.whisper.checked = false;
}
if (s_ formValue[0] == "true")
{
    form1.whisper.checked = true;
}

}
</script>

</head>
<body bgcolor = "#99cc66" onload = "setFocus()">
<%@ page import = "java.util.*" %>
<%@ page contentType = "text/html; charset = gb2312" %>
<%!
public void clearVector()
{
    Vector Msg = (Vector)getContext().getAttribute("chatmessage");

    if (Msg.size() > 40)
    {
        Msg.removeAllElements();
        Msg.addElement("<font color = 'orange' size = '2'>" + "清除了消息,大家慢聊" + "</font><br>");
    }
}
%>
<%@include file = "obj.jsp"%>
<%
boolean ifsed = true;
Vector tempChat1Msg = (Vector)getContext().getAttribute("chatmsg");
String cUserName = (String)session.getValue("username");
```

```

        Date userTime = new Date();
        String
timestamp = "[" + userTime.getHours() + ":" + userTime.getMinutes() + ":" + userTime.
getSeconds() + "]";

% >

< %
byte[] temp_b;
String temp_s;
temp_s = request.getParameter("message");
temp_b = temp_s.getBytes("ISO-8859-1");
String temp = new String(temp_b);

byte[] object_b;
String object_s;
object_s = request.getParameter("whisperto");
object_b = object_s.getBytes("ISO-8859-1");
String object = new String(object_b);

String color = request.getParameter("color");
String action = request.getParameter("action");
String whisper = request.getParameter("whisper");
String iswhisper = "on";
% >

< %

if(temp.startsWith("^"))
{
    temp = "< font color = 'red' size = '2' > 请不要使用^标记" + timestamp +
"</font><br>";
    tempChat1Msg.addElement(temp);
    ifsed = false;
    getServletContext().setAttribute(chatmsg, tempChat1Msg);
}

if(temp.startsWith("<"))
{

```

```
temp = "<font color = 'red' size = '2'>请不要使用 html 标记" + timestamp + "  
</font><br>";  
tempChat1Msg.addElement(temp);  
ifsed = false;  
getServletContext().setAttribute(chatmsg, tempChat1Msg);  
}  
  
if (temp.endsWith("/>"))  
{  
temp = "<font color = 'red' size = '2'>请不要使用 html 标记" + timestamp + "  
</font><br>";  
tempChat1Msg.addElement(temp);  
ifsed = false;  
getServletContext().setAttribute(chatmsg, tempChat1Msg);  
}  
  
if (wbisper == null)  
{  
whisper = "off";  
  
}  
else  
{  
whisper = "on";  
  
}  
  
String speaker = "self";  
if (wbisper == "on")  
{  
speaker = "^" + cUserName + "^" + object + "^";  
}  
  
if (object.equals("all"))  
{  
object = "每一个人";  
}
```



```

% >

< %
//String cUserName = (String)session.getValue("username");
Vector _tempuser = (Vector)getContext().getAttribute("user");
if (_tempuser.contains(cUserName) == false)
{
    ifsed = false;
}

cUserName = "<font color='red' size='3'>" + cUserName + "</font>";
object = "<font color=' # ee6666' size='3'>" + object + "</font>";

    if(ifsed == true)
    {

        Hashtable userLife = (Hashtable)getContext().getAttribute("usertime");
        userLife.put((String)session.getValue("username"), new
Long(userTime.getTime()));
        getContext().setAttribute("usertime", userLife);

    }
    if(ifsed == true)
    {
        if(whisper.equals("off"))
        {

            if(action.equals("1"))
            {
                clearVector();
                action = "对";
                tempChat1Msg.addElement("<font color=" + color + " size='2'>" +
cUserName + action + object + "说:" + temp + "</font><br>");
                getContext().setAttribute("chatmsg", tempChat1Msg);

            }
            if (action.equals("2"))
            {
                clearVector();

```

```

        action = "微微笑着对";
        tempChat1Msg.addElement("<font color = " + color + " size = '2' ">" +
cUserName + action + object + "说:" + temp + "</font><br>");
        getServletContext().setAttribute(chatmsg, tempChat1Msg);
    }

    .
    .
    .

    </td>
    </tr>
    </table>
    </td>
    </tr>
    </table>
</form>
</body>
</html>

```

发送聊天信息程序比较复杂,程序开头定义了两个 javascript 函数用于在页面刷新后能保证用户选择内容保持不变,在处理私聊时用 '^' 号将说话者与对象分开,便于在显示聊天部分进行处理。最后,对用户选择信息与输出内容进行匹配,存入到向量中。

## 2. 显聊天信息程序

showchat.jsp

```

<html>
<meta http-equiv="refresh" content="10">
<head>
</head>
<%@ page contentType="text/html; charset = gb2312" %>
<%@ page import="java.util. *" %>
<%@include file="obj.jsp"%>
<body bgcolor="#FFFFCC">
<%
    String s_c_n;
    String o_c_n;
    int first;
    int last;
    String temp_msg;
    String ccn = (String)session.getValue("username");

```

```

        boolean iswhisper = false;
        int j = 0;

        Vector Msg = (Vector)getContext().getAttribute(chatmsg);
        for (int i = 0; i < Msg.size(); i++) {

            temp_msg = Msg.elementAt(Msg.size() - i - 1).toString();

            if (temp_msg.startsWith("^")) {

                iswhisper = true;
            }

            //处理私聊和普通聊天的信息显示。
            if (iswhisper == true) {
                first = 1;
                last = temp_msg.indexOf("^", first);
                s_c_n = temp_msg.substring(first, last);
                first = last + 1;
                last = temp_msg.indexOf("^", first);
                o_c_n = temp_msg.substring(first, last);
                first = last + 1;
                temp_msg = temp_msg.substring(first, temp_msg.length());
                if (o_c_n.equals(ccn)) {
                    temp_msg = "<img src='home.jpg'>" + temp_msg;
                    out.println(temp_msg);
                }
                if (s_c_n.equals(ccn)) {
                    temp_msg = "<img src='home.jpg'>" + temp_msg;
                    out.println(temp_msg);
                }
                iswhisper = false;
            }
            else {
                out.println(temp_msg);
            }
        }
    }
    %>
</body>

```

```
</html>
```

该程序主要用于提取向量内容,对私聊部分进行处理,同时将信息输出。在发送信息程序中将私聊信息加上了^符号进行分隔,在此处先遍历向量,查找是否有以^号开头的信息,来判断是否是悄悄话。如果不是就直接显示出来,若是就将^号分隔的内容分离出来,同时判断是否是当前用户,若满足条件就显示,并在信息前加上图标表示是悄悄话。这样,其他聊天者就无法看到。

### 3. 显示聊天室信息程序

```
chatroom.jsp
<html>
<meta http-equiv="refresh" content="120">
<title>
</title>
<head>
</head>
<body bgcolor="#999966">
<%@ page contentType="text/html; charset = gb2312"%>
<%@ page import="java.util.*"%>
<%@ include file="obj.jsp"%>
<%
    Hashtable chaterlife = (Hashtable)getContext().getAttribute("usetime");

    Hashtable cbaterip = (Hashtable)getContext().getAttribute("userid");

    Vector chater = (Vector)getContext().getAttribute("user");

    Date nowDate = new Date();

    Long _cbaterTime = (Long)chaterlife.get((String)session.getValue("username"));
    long chatTime = _cbaterTime.longValue();
    long nowTime = nowDate.getTime();
    long stoptime = nowTime - chatTime;

    if (stoptime > (long)240000)//4 minutes{

        boolean timeEsc = chaterlife.containsKey((String)session.getValue("username"));
        boolean ipEsc = cbaterip.containsKey((String)session.getValue("username"));
        boolean chaterEsc = chater.contains((String)session.getValue("username"));
        if (timeEsc == true){
```

```
tempchaterTime = (Long)chaterlife.remove((String)session.getValue("username"));
    }
    if (ipEsc == true){
```

```
tempchaterip = (String)chaterip.remove((String)session.getValue("username"));

    }
    if (chaterEsc == true)
    {
        boolean
        booltempchater = chater.remove((String)session.getValue("username"));
        {
            {
                getServletContext().setAttribute(usetime, chaterlife);
                getServletContext().setAttribute(userid, chaterip);
                getServletContext().setAttribute(user, chater);
            }
        }
    }
}
```

<p><font family="宋体"; size=3>聊天室</font></p>

<code>&lt;table cellpadding="2" cellspacing="2" border="0" width="80" align="center"&gt;</code>
---

<td><font family="宋体"; size=2>帮助文件</font></td>
--

 $\langle \text{tr} \rangle$  <td> <font family="宋体"; size=2><a href="listuser.jsp" target="\_new"> | $\langle \text{tr} \rangle$

```

        <td> <font family="宋体"; size=2><a href="" target="_new">新建聊天室
</a></font></td>
    </tr>
    <tr>
        <td> <font family="宋体"; size=2><a href="leave.jsp" target="_top">
回到主页</a></font></td>
    </tr>

    <tr>
        <td> <font family="宋体"; size=2><br>
聊天论题:<br>
        <% out.println(getServletContext().getAttribute(chatobj)); %><br></font>
></td>
    </tr>
    <tr>
        <td> <font family="宋体"; size=2><br>
当前在聊:<br>
        <%
Vector temp=(Vector)getServletContext().getAttribute(user);
int i=temp.size();
String num=Integer.toString(i);
for(i=0;i<num.length();i++)
%>

        <img src='pic/% = num.charAt(i) %>.gif'></img>
        <% | %>
人<br></font></td>
    </tr>
</table>
</body>
</html>

```

#### 4. 离开聊天室程序

```

<html>
<head>
<title></title>
</head>
<body bgcolor="#006633">
<% @ page import="java.util. *" %>

```

```
<%@ page contentType="text/html; charset = gb2312" %>
<jsp:include page = "../template/header2.jsp" flush = "true"/>
<%@include file = "obj.jsp"%>
<%
String temuser = (String)session.getValue("username");
Vector temp = (Vector)getContext().getAttribute(user);
temp.removeElement(temuser);
getContext().setAttribute(user, temp);
%>
<%
temp = (Vector)getContext().getAttribute(chatmsg);
temp.addElement("<font color = 'red' size = '2'>" + "告诉大家:" + temuser + "离开了." <br></font>");
getContext().setAttribute(chatmsg, temp);
%>
<br>
<center> 欢迎你常来! </center>
</body>
</html>
```

该程序主要是当用户点击离开链接时,将该用户向量中的用户信息删除并发送信息到消息向量中。这样就将用户信息注销掉了。

该例子仅仅只是聊天室的一个雏形,安全性不高,感兴趣的读者可以在此基础上加以改进,比如将聊天部分也挂接数据库,添加管理程序等等。

## 14.4 本章小结

在本章中详细介绍了网站建设中具有代性的有机组成部分。通过本章的学习读者可以达到利用 JSP 技术架设一个完整的站点。当然,书中介绍的内容仅仅是实际网站建设的部分内容,希望大家在实践中不断地摸索,完善 JSP 网站开发技术。

# 附录 1 Tomcat 安装汇总

## 1.1 资源下载

### 1.Sun JDK 1.3

可在 <http://java.sun.com/j2se/1.3/> 下载 JDK (Java™ 2 SDK, Standard Edition, v 1.3, 其中 linux 和 solaris 版本为 Beta 版)。

### 2.Sun JSWDK1.0.1 (最新,735K)

<http://java.sun.com/products/jsp/>

### 3.Jakarta Tomcat3.1(最新)

<http://jakarta.apache.org>

### 4.Sun Java Web Server2.0 for NT beta2 trial (最新,9M)

试用 30 日。

<http://www.sun.com/software/jwebserver/try/index.html>

下载之前还需要填一些你的注册信息。

Tomcat 是 apache 上实现 jsp + javabeen 环境的接口程序,集成了 jsp1.1 和 servlet 2.2。它提供一个单独的 mod-jserv.so 模块,利用 apache 的 DSO 动态载入,与以前的 Jserver 不同, Tomcat 必须作为一个单独的程序运行,所有提交给 apache 的 java 请求将通过 Tomcat 自带的 jserv 模块提交给 Tomcat 进一步处理。这就是说,必须先运行 Tomcat,再运行 apache 才能解释 jsp/servlet 程序。

## 1.2 Tomcat 在 Windows NT 4.0, Windows 2000 下的安装

### 1.JDK 的安装

(1) 双击文件进行安装,使用默认缺省配置进行安装,JDK 的默认安装目录为 C:\jdk1.3,JRE 的默认安装目录为 C:\Program Files\JavaSoft\JRE\1.2。

(2) 重启计算机。



(3) 更新下列环境变量:把 C:\jdk1.2.2\bin 目录追加到 PATH 中;把 C:\jdk1.3\lib\tools.jar;C:\jdk1.3\lib\dt.jar 加入到 CLASSPATH 中。更新方法:控制面板→系统→环境→系统变量。

## 2. Tomcat 的安装

(1) 用 Winzip 等解压缩软件把 tomcat.zip 解压缩到一个目录下,如把它解压缩到 C:\,它会自动创建 tomcat 子目录,这样在 C 盘就多了一个目录 C:/tomcat,使用 \$TOMCAT 标识。

(2) 修改 Tomcat 运行的端口号,注意 Tomcat 自己有一个独立的 HTTP 服务器,它必须使用一个还未被使用的端口号,这里的 8081 还未被占用,即将 \$TOMCAT\_PORT 标识,分配给 Tomcat:

►(打开 \$TOMCAT/server.xml;

►(修改 ContextManager;

(3) 双击 \$TOMCAT 目录下的 startup.bat 来启动 Tomcat,在浏览器上输入 http://localhost:\$TOMCAT\_PORT/,能看到 Tomcat Version 3.0 这一页就表示 Tomcat 安装成功了。

(4) 点击 Servlet Example 进入 Servlet 界面,应该能执行 Servlet。

(5) 点击 JSP Example 进入 JSP 界面,执行 JSP,若不能,修改 \$TOMCATconfstart.bat;加上 SET JAVA\_HOME=C:\jdk1.3 即可。

# 1.3 Tomcat 在 Redhat 下的安装并与 apache 相连

## 1. 安装

(1) 设置环境变量,在登录用户目录HOME/.bash\_profile 设置环境变量如下:

```
JAVA_HOME=/usr/local/jdk
```

```
TOMCAT_HOME=/usr/local/tomcat
```

```
CLASSPATH=$JAVA_HOME/lib/tools.jar:/usr/local/apache/classes/classes11
```

(2) zip:.

(3) 将 jakarta-tomcat.tarexport JAVA\_HOME TOMCAT\_HOME CLASSPATH

(4) cd /usr/local;

```
tar zxvf jdk1_2_2-linux-i386.tar.gz;
```

```
ln -s jdk1.2.2 jdk;
```

(5) cd /usr/local;

```
tar zxvf jakarta-tomcat.tar;
```

```
ln -s jakarta-tomcat tomcat;
```

(6) 可以启动 Tomcat 自带的调试环境,端口 8080

启动: /usr/local/tomcat/bin/tomcat.sh start

访问: http://localhost:8080/examples

## 2. 与 Apache 建立连接

(1) cp mod\_jserv.so /usr/lib/apache

(2) 在/etc/httpd/conf/httpd.conf 文件的最后加入

Include /usr/local/tomcat/conf/tomcat - apache.conf

(3) 更改/usr/local/tomcat/conf/tomcat - apache.conf 第一行为:

(4) LoadModule

jserv\_module /usr/lib/apache/mod\_jserv.so /usr/local/tomcat/bin/tomcat.sh stop;

(5) /usr/local/tomcat/bin/tomcat.sh start;

(6) /etc/rc.d/init.d/httpd restart 现在可以通过 http://localhost/examples 访问 jsp/servlet

## 3. 配置

加入一个新的映射目录(如:通过 http://localhost/chinapic 访问, /chinapic 映射到系统目录/home/httpd/chinapic)。

(1) stop Tomcat 和 Apache

(2) vi /usr/local/tomcat/conf/server.xml, 加入:

```
<Context path="/chinapic" docBase="/home/httpd/chinapic"
    debug="0" reloadable="true">
</Context>
```

(3) /usr/local/tomcat/bin/tomcat.sh start

(4) cp /usr/local/tomcat/conf/tomcat - apache.conf /

usr/local/tomcat/tomcat - apache - cig.conf

(5) 修改/usr/local/tomcat/tomcat - apache - cig.conf, 将 chinapic 定义部分的目录映射改为/home/httpd/chinapic

(6) cp /usr/local/tomcat/webapps/examples /home/httpd/chinapic/ -R

(7) 重启 Apache, 即可通过 http://localhost/chinapic 访问 jsp/servlet.

# 1.4 Tomcat 在 Unix 下的安装

## 1. 安装

(1) 解压缩此文件到某目录(如:foo)将会生成一个子目录,名为“tomcat”。

(2) 转换到“tomcat”目录设置一新的环境变量(TOMCAT\_HOME)指向安装的 tomcat 的目录。

如是 bash/sh 环境, 键入: “TOMCAT\_HOME = foo/tomcat; export TOMCAT\_

HOME”

如是 tcsh 环境，键入：“setenv TOMCAT\_HOME foo/tomcat”

(3) 设置环境变量 JAVA\_HOME 指向 JDK 的目录，然后添加 JAVA 解释器到 PATH 环境变量。

现在可以运行 TOMCAT 并作为一个独立的 Servlet 容器。

## 2. 启动与关闭 Tomcat

使用“bin”目录中的脚本启动与关闭 Tomcat。

启动：

unix: bin/startup.sh

关闭：

unix: bin/shutdown.sh

# 附录 2 Servlet API

## —javax.servlet 包

### 2.1 javax.servlet Class GenericServlet

#### 1. 类层次

```
java.lang.Object
|
+-- javax.servlet.GenericServlet
```

#### 2. 直接已知子类

HttpServlet

#### 3. 说明

```
public abstract class GenericServlet extends java.lang.Object
    implements Servlet, ServletConfig, java.io.Serializable
```

定义一个一般的,与协议无关的 servlet。当书写用于 Web 的 HTTP servlet 时,尽量使其扩展自 HttpServlet,而非扩展自 GenericServlet。

GenericServlet 实现了 Servlet 和 ServletConfig 界面。尽管一般都扩展实现特定协议的子类,但 GenericServlet 仍可被一个 servlet 直接扩展,

GenericServlet 使书写 servlet 更容易。它提供简单版本的生命周期方法 init 和 destroy 以及 ServletConfig 界面中的方法。GenericServlet 也实现了在 ServletContext 界面中声明的 log 方法,

书写一个一般的 servlet,只需要覆盖抽象方法 service。

#### 4. 构造器一览

GenericServlet()	什么也不做。
------------------	--------

#### 5. 方法一览

void	destroy()	Servlet 容器调用,将当前 servlet 移出服务(栈),即销毁它。
java.lang.String	getInitParameter( java.lang.String name)	返回一个字符串,包含给出名字的初始化参数的值,或如果该参数不存在返回 null。

续表

java.util.Enumeration	getInitParameterNames()	返回一个 String 对象的枚举变量,包含当前 servlet 所有的初始化参数,或如果当前 servlet 没有初始化参数返回一个空的枚举变量。
ServletConfig	getServletConfig()	返回当前 servlet 的 ServletConfig 对象。
ServletContext	getServletContext()	返回对当前 servlet 运行的上下文的一个引用,即对当前 ServletContext 的一个引用。
java.lang.String	getServletInfo()	返回当前 servlet 的有关信息,例如作者、版本和著作权。
java.lang.String	getServletName()	返回当前 servlet 实例的名字。
void	init()	一个有用的方法,可被覆盖,因此没有必要调用 super.init(config)。
void	init(ServletConfig config)	Servlet 容器调用,指示当前 servlet 将被放入服务(栈)。
void	log(java.lang.String msg)	将指定信息写入当前 servlet 的日志文件中。
void	log(java.lang.String message, java.lang.Throwable t)	将解释性信息和给出异常的堆栈跟踪信息写入当前 servlet 的日志文件中。
abstract void	service(ServletRequest req, ServletResponse res)	Servlet 容器调用,允许当前 servlet 响应请求。

## 6. 继承自 class java.lang.Object 的方法

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## 2.2 javax.servlet Interface RequestDispatcher

### 1. 说明

```
public interface RequestDispatcher
```

定义一个对象,接收来自客户端的请求并把这些请求发送给服务端的任意资源(例如 servlet,HTML 文件或 JSP 文件)。Servlet 容器创建 RequestDispatcher 对象,该对象封装了特定路径或由特定名字给出的资源。

这个界面打算封装 servlet,但是 servlet 容器能创建 RequestDispatcher 对象封装任何类型的资源。

## 2. 方法一览

Void	forward(ServletRequest request, ServletResponse response)	转发来自一个 servlet 的一个请求到服务端的另一个资源(servlet, JSP 文件或 HTML 文件)
void	include(ServletRequest request, ServletResponse response)	在响应中包含一个资源(servlet, JSP 文件, HTML 文件)的内容。

## 3. 方法详解

### 1) forward

```
public void forward(ServletRequest request, ServletResponse response)
    throws ServletException, java.io.IOException
```

转发来自一个 servlet 的一个请求到服务端的另一个资源(servlet, JSP 文件或 HTML 文件)。这个方法使得一个 servlet 预先处理一个请求,然后由另一个资源产生响应。

通过 getRequestDispatcher() 方法获得一个 RequestDispatcher,对它来说,ServletRequest 对象中有其路径成分和匹配目标资源路径的参数。

Forward 应在响应已提交给客户端之前调用,即在响应体输出被刷新之前。如果响应已被提交,这个方法抛出一个 IllegalStateException。

Uncommitted output in the response buffer is automatically cleared before the forward. 请求和响应参数必须与传递给 servlet 的 service 方法的对象是相同的。

#### 参数

request: 一个 ServletRequest 对象,表示客户端对 servlet 的请求。

Response: 一个 ServletResponse 对象,表示 servlet 返回给客户端的响应。

#### 抛出

ServletException——如果目标资源抛出这个异常。

java.io.IOException——如果目标资源抛出这个异常。

java.lang.IllegalStateException——如果响应已被提交。

### 2) include

```
public void include(ServletRequest request, ServletResponse response)
    throws ServletException, java.io.IOException
```

在响应中包含一个资源(servlet, JSP 文件, HTML 文件)的内容。本质上,这个方法实现了程序级服务端包含。

ServletResponse 对象有 include 的路径成分和来自调用者的保持不变的参数。被包含的 servlet 不能改变响应的状态码或设置头部,任何作修改的意图都将被忽略。

请求和响应参数必须与传递给 servlet 的 service 方法的对象是相同的。

#### 参数

request: 一个 ServletRequest 对象,包含客户端的请求。

Response: 一个 ServletResponse 对象,包含 servlet 返回给客户端的响应。

抛出

ServletException——如果被包含资源抛出这个异常。

java.io.IOException——如果被包含资源抛出这个异常。

## 2.3 javax.servlet Interface Servlet

### 1. 所有已知子界面

HttpJspPage, JspPage

### 2. 所有已知实现类

GenericServlet

### 3. 说明

public interface **Servlet**

定义所有的 servlet 必须实现的方法。

一个 servlet 是一个运行在 Web 服务端的小的 Java 程序。Servlet 接收和响应来自 Web 客户端的请求,通常通过 HTTP 协议。

为实现这个界面,你可以写一个扩展自 javax.servlet.GenericServlet 的一般 servlet 或者一个扩展自 javax.servlet.http.HttpServlet 的 HTTP servlet。

这个界面定义了初始化 servlet,提供服务给请求,从服务端删除 servlet 的方法。这就是生命周期方法,以如下顺序被调用。

构造 servlet,然后用 init 方法初始化该 servlet。

客户端的任何请求由 service 方法处理。

该 servlet 被移出服务栈,然后由 destroy 方法销毁,最后作为垃圾被收集并结束。

除生命周期方法之外,这个界面提供 getServletConfig 方法,servlet 使用它可获得任何启动信息,还有 getServletInfo 方法,允许 servlet 返回关于自身的基本信息,例如作者、版本、著作权。

### 4. 方法一览

Void	destroy()	Servlet 容器调用,指示当前 servlet 将被移出服务(栈)。
ServletConfig	getServletConfig()	返回一个 ServletConfig 对象,包含当前 servlet 的初始化和启动信息。
java.lang.String	getServletInfo()	返回有关当前 servlet 的信息,例如作者、版本、著作权。
void	init(ServletConfig config)	Servlet 容器调用,指示当前 servlet 将被放入服务(栈)。
void	service(ServletRequest req, ServletResponse res)	Servlet 容器调用,允许当前 servlet 响应请求。

## 5. 方法详解

### 1) init

public void init(ServletConfig config) throws ServletException

Servlet 容器调用,指示当前 servlet 将被放入服务(栈)。一旦当前 servlet 实例化, servlet 容器就调用 init 方法。当前 servlet 能接收任何请求之前,init 方法必须成功执行完成。

servlet 容器不能将当前 servlet 放入服务栈,如果 init 方法:

抛出一个 ServletException 异常。

没能在 Web 服务端指定的时间段之内返回。

参数

config: 一个 ServletConfig 对象,包含当前 servlet 的配置和初始化参数。

抛出

ServletException——如果一个已发生的异常干扰了当前 servlet 的正常操作。

参照

UnavailableException, getServletConfig()

### 2) getServletConfig

public ServletConfig getServletConfig()

返回一个 ServletConfig 对象,包含当前 servlet 的初始化和启动信息。返回的 ServletConfig 对象将传递给 init 方法。

这个界面的实现负责存储 ServletConfig 对象,因此这个方法才能返回该 ServletConfig 对象。

GenericServlet 类,实现了这个界面,所以可以使用这些方法。

返回

用于初始化当前 servlet 的 ServletConfig 对象。

参照

init(javax.servlet.ServletConfig)

### 3) service

public void service(ServletRequest req, ServletResponse res)  
throws ServletException, java.io.IOException

Servlet 容器调用,允许当前 servlet 响应请求。

这个方法只能在 servlet 初始化方法 init()成功完成后被调用。

Servlet 典型的运行在多线程环境中,即 servlet 容器能同时处理多个请求。开发者必须清楚同步访问任何共享资源,例如文件,网络连接以及 servlet 类和变量实例。

参数

req: ServletRequest 对象,包含客户端的请求。

res: ServletResponse 对象,包含 servlet 的响应。

抛出

ServletException——如果一个已发生的异常干扰了当前 servlet 的正常操作。



java.io.IOException——如果一个输入或输出异常发生。

#### 4) getServletInfo

```
public java.lang.String getServletInfo()
```

返回有关当前 servlet 的信息,例如作者、版本、著作权。

这个方法返回的字符串应当是普通文本,不应该是任何类型的标记文本(例如 HTML、XML 等)。

返回

一个包含 servlet 信息的字符串。

#### 5) destroy

```
public void destroy()
```

Servlet 容器调用,指示当前 servlet 将被移出服务(栈)。

当前 servlet 的 service 方法内的所有线程都已退出或经过一个超时时间片后(未被处理),才能调用这个方法。Servlet 容器调用这个方法之后,它就不再对当前 servlet 调用 service 方法。

这个方法使 servlet 能清除它拥有的任何资源(例如内存、句柄、线程),也使 servlet 确定任何持久性状态与它自身在内存中的状态是同步。

## 2.4 javax.servlet Interface ServletConfig

### 1. 所有已知子界面

GenericServlet

### 2. 说明

```
public interface ServletConfig
```

一个 servlet 配置对象,在初始化期间,servlet 容器使用它传递信息给一个 servlet。

配置信息 包含一组“名字/值”对形式的初始化参数和一个 ServletContext 对象,配置信息提供有关服务端的信息给 servlet。

### 3. 方法一览

java.lang.String	getInitParameter( java.lang.String name)	返回一个字符串,包含给出名字的初始化参数的值,或如果该参数不存在返回 null。
java.util.Enumeration	getInitParameterNames()	以 String 对象的枚举形式返回当前 servlet 的所有初始化参数的名字,或当前 servlet 没有初始化参数返回一个空的枚举变量。
ServletContext	getServletContext()	返回对当前 ServletContext 的引用,当前 ServletContext 就是当前 servlet 正执行在的哪个上下文。
java.lang.String	getServletName()	返回当前 servlet 实例的名字。

## 2.5 javax.servlet Interface ServletContext

### 1. 说明

public interface ServletContext

定义一组方法, servlet 使用它们与自身的 servlet 容器通信, 例如, 获得一个文件的 MIME 类型, 分配请求, 或写日志文件。

每个 Java 虚拟机, 每个页面应用程序都有一个上下文。(页面应用程序是 servlet 和内容的集合, 内容可能由 .war 文件安装在服务端 URL 名域的特定子集下)

一个页面应用程序在它的调度描述符中标记为“可分布的”, 在这种情况下, 对每个虚拟机都有一个上下文实例。这时候, 上下文不能用来定位共享的全局信息(因为这种信息不是真的全局信息)。使用外部资源如数据库代替。

当前 ServletContext 对象包含在当前 ServletConfig 对象中, 当前 servlet 初始化时, Web 服务端把它提供给该 servlet。

### 2. 方法一览

java.lang.Object	getAttribute( java.lang.String name)	返回当前 servlet 容器中有参数给出名字的属性对象, 或如果参数给出名字指定的属性不存在, 返回 null。
java.util.Enumeration	getAttributeNames()	返回一个枚举变量, 包含当前 servlet 上下文中所有有效的属性名字。
ServletContext	getContext( java.lang.String uripath)	返回服务端的一个与指定 URL 相符合的 ServletContext 对象。
java.lang.String	getInitParameter( java.lang.String name)	返回一个字符串, 包含当前上下文中给出名字的初始化参数的值, 或如果该参数不存在, 返回 null。
java.util.Enumeration	getInitParameterNames()	返回一个 String 对象的枚举变量, 包含当前上下文中所有初始化参数的名字, 或如果当前上下文没有初始化参数, 返回一个空的枚举变量。
int	getMajorVersion()	返回当前 servlet 容器支持的 Java Servlet API 的主版本号。
java.lang.String	getMimeType( java.lang.String file)	返回指定文件的 MIME 类型, 或如果 MIME 类型未知, 返回 null。
int	getMinorVersion()	返回当前 servlet 容器支持的 Java Servlet API 的次版本号。
RequestDispatcher	getNamedDispatcher( java.lang.String name)	返回一个 RequestDispatche 对象, 它封装了给出名字的 servlet。

续表

java.lang.String	getRealPath( java.lang.String path)	返回一个字符串,包含给出虚拟路径的真实路径。
RequestDispatcher	getRequestDispatcher( java.lang.String path)	返回一个 RequestDispatcher 对象,它封装了给出路径定位的资源。
java.net.URL	getResource( java.lang.String path)	返回与指定路径相映射的资源的 URL。
java.io.InputStream	getResourceAsStream( java.lang.String path)	以 InputStream 对象形式返回给出路径定位的资源。
java.lang.String	getServerInfo()	返回当前 servlet 正运行在其上的 servlet 容器的名字和版本号,即当前容器的名字和版本号。
Servlet	getServlet( java.lang.String name)	不赞成使用。Java Servlet API 2.1 中没有直接的替换者。
java.util.Enumeration	getServletNames()	不赞成使用。Java Servlet API 2.1 中没有替换者。
java.util.Enumeration	getServlets()	不赞成使用。Java Servlet API 2.1 中没有替换者。
void	log(java.lang.Exception exception, java.lang.String msg)	不赞成使用。Java Servlet API 2.1 中使用 log(String message, Throwable throwable)代替。
void	log(java.lang.String msg)	写指定的信息到 servlet 日志文件,通常是事件日志。
void	log(java.lang.String message, java.lang.Throwable throwable)	写解释信息和给出的可抛出异常的一个堆栈跟踪信息到 servlet 日志文件。
void	removeAttribute( java.lang.String name)	从 servlet 上下文中删除给出名字的属性。
void	setAttribute(java.lang. String name, java.lang.Object object)	绑定一个对象到当前 servlet 上下文的给出属性名中

### 3. 方法详解

#### 1) getContext

public ServletContext getContext(java.lang.String uripath)

返回服务端的一个与指定 URL 相关联的(绑定的)ServletContext 对象。

这个方法使得 servlet 可访问服务端的不同部分的上下文。给出的路径必须是绝对路径(以“/”开始),它被解释成以服务端的文档根目录为基础的。

在一个安全环境下,对给出的 URL,servlet 容器可能返回 null。

参数

uripath:一个字符串,指明资源在服务端的绝对 URL。

参照

RequestDispatcher

2) getMajorVersion

```
public int getMajorVersion()
```

返回当前 servlet 容器支持的 Java Servlet API 的主版本号。所有兼容 2.2 版本的实现必须有这个方法返回整数 2。

返回

2

3) getMinorVersion

```
public int getMinorVersion()
```

返回当前 servlet 容器支持的 Java Servlet API 的次版本号。所有兼容 2.2 版本的实现必须有这个方法返回整数 2。

返回

2

4) getMimeType

```
public java.lang.String getMimeType(java.lang.String file)
```

返回指定文件的 MIME 类型,或如果 MIME 类型未知,返回 null。MIME 类型由 servlet 容器的配置决定,也可在一个页面应用程序开发环境中指定。一般 MIME 类型是“text/html”和“image/gif”。

参数

file:一个指定文件名的字符串。

返回

一个指定该文件 MIME 类型的字符串。

5) getResource

```
public java.net.URL getResource(java.lang.String path)
```

throws java.net.MalformedURLException

返回与指定路径相映射的资源的 URL。路径必须以“/”开始,它被解释为对当前上下文的根的相对路径。

这个方法允许 servlet 容器使一个资源对任何 servlet 都有效。资源可定位在本地或远程文件系统,数据库或“.war”文件。

Servlet 容器必须实现 URL 处理者和 URL 连接对象,这是访问资源所必须的。

如果指定路径上没有资源相映射,返回 null。

有些容器可能允许使用 URL 类中的方法写这个方法返回的 URL。

资源内容被直接返回,注意对 .jsp 页面的请求返回 JSP 源代码。使用 RequestDispatcher 代替,以包含执行结果。

与 java.lang.Class.getResource 相比,这个方法有不同的目的。java.lang.Class.getResource 方法基于类加载器查看资源,这个方法不使用类加载器。

参数

path: 指定资源路径的字符串。

返回

指定路径定位的资源,或在指定路径上没有资源,返回 null。

抛出

java.net.MalformedURLException——如果路径名没有以正确的形式给出。

#### 6) getResourceAsStream

```
public java.io.InputStream getResourceAsStream(java.lang.String path)
```

以 InputStream 对象形式返回给出路径定位的资源。

InputStream 中的数据可是任意的类型和长度。路径必须根据在 getResource 中给出的规则指定。如果在指定的路径上不存在资源,这个方法返回 null。

通过 getResource 方法可获得元信息,例如内容长度和内容类型,但是使用这个方法,这些元信息都将丢弃。

Servlet 容器必须实现 URL 处理者和 URL 连接对象,这是访问资源所必须的。

这个方法与 java.lang.Class.getResourceAsStream 方法使用一个类加载器是不同的,这个方法允许 servlet 容器使一个资源对任何位置的 servlet 都有效,而不使用类加载器。

参数

name: 一个指定资源路径的字符串。

返回

InputStream,或如果指定的路径上不存在资源,返回 null。

#### 7) getRequestDispatcher

```
public RequestDispatcher getRequestDispatcher(java.lang.String path)
```

返回一个 RequestDispatcher 对象,它封装了给出路径定位的资源。RequestDispatcher 对象可用作转发请求给该资源或将该资源包含在响应中。资源可是动态的,也可是静态的。

路径必须以“/”开始,它被解释为对当前上下文的根的相对路径。使用 getContext 可获得对其它上下文的资源的一个 RequestDispatcher。如果 ServletContext 不能返回一个 RequestDispatcher,这个方法返回 null。

参数

path: 一个指定资源路径的字符串。

返回

一个 RequestDispatcher 对象,它封装了给出路径定位的资源。

参照

RequestDispatcher, getContext(java.lang.String)

#### 8) getNamedDispatcher

```
public RequestDispatcher getNamedDispatcher(java.lang.String name)
```

返回一个 RequestDispatcher 对象,它封装了给出名字的 servlet。

Servlet(和 JSP 页面)可通过服务端管理员或页面应用程序开发环境赋予一个名字。servlet 实例可使用 ServletConfig.getServletName()方法确定自身的名字。

如果 ServletContext 因为某种原因不能返回 RequestDispatcher,这个方法返回 null。

**参数**

name: 一个字符串,指定欲封装的 servlet 的名字。

**返回**

一个 RequestDispatcher 对象,封装了给出名字的 servlet。

**参照**

RequestDispatcher, getContext(java.lang.String), ServletConfig.getServletName()

**9) getServlet**

```
public Servlet getServlet(java.lang.String name)
```

throws ServletException

不赞成使用。Java Servlet API 2.1 中没有直接的替换者。

这个方法原本定义从 ServletContext 中获得一个 servlet。在本版本中,这个方法始终返回 null,保留它只是保持二成分(getServlet()方法和 getServlets()方法)对称兼容。这个方法将在 Java Servlet API 的今后版本中永久地删除。

使用这个方法的情况,使用 ServletContext 类,几个 servlet 之间可共享信息并且通过调用公共的非 servlet 类执行共享的商业逻辑。

**10) getServlets**

```
public java.util.Enumeration getServlets()
```

不赞成使用。Java Servlet API 2.0 中没有替换者。

这个方法原本定义从 ServletContext 中获得当前 servlet 上文中所有已知 servlet 的一个枚举变量。在本版本中,这个方法始终返回一个空的枚举变量,保留它只是保持二成分(getServlet()方法和 getServlets()方法)对称兼容。这个方法将在 Java Servlet API 的今后版本中永久地删除。

**11) getServletNames**

```
public java.util.Enumeration getServletNames()
```

不赞成使用。Java Servlet API 2.1 中没有替换者。

这个方法原本定义从 ServletContext 中获得当前 servlet 上文中所有已知 servlet 名字的一个枚举变量。在本版本中,这个方法始终返回一个空的枚举变量,保留它只是保持二成分(getServlet()方法和 getServlets()方法)对称兼容。这个方法将在 Java Servlet API 的今后版本中永久地删除。

**12) log**

```
public void log(java.lang.String msg)
```

写指定的信息到 servlet 日志文件,通常是事件日志。Servlet 日志文件的名称和类型对该 servlet 的容器来说是特定的。

**参数**

msg: 一个字符串,指定将写入日志文件的信息。

**13) log**

```
public void log(java.lang.Exception exception, java.lang.String msg)
```

不赞成使用。Java Servlet API 2.1 中使用 log(String message, Throwable throwable) 代替。

这个方法原本定义将错误的解释性信息和异常的堆栈跟踪信息写入 servlet 日志文件。

#### 14) log

```
public void log(java.lang.String message, java.lang.Throwable throwable)
```

写解释信息和给出的可抛出异常的一个堆栈跟踪信息到 servlet 日志文件。通常是事件日志。Servlet 日志文件的名称和类型对该 servlet 的容器来说是特定的。

##### 参数

message: 一个描述错误或异常的字符串。

Throwable; 可抛出的错误或异常。

#### 15) getRealPath

```
public java.lang.String getRealPath(java.lang.String path)
```

返回一个字符串, 包含给出虚拟路径的真实路径。例如, 虚拟路径“/index.html”在服务端文件系统的真实路径由对“/index.html”的请求提供服务。

返回的真实路径的形式对当前 servlet 容器运行的计算机和操作系统是适当的。如果 servlet 容器因为某种原因不能将虚拟路径转换成真实路径, 这个方法返回 null。

##### 参数

path: 一个字符串, 指定一个虚拟路径。

##### 返回

一个字符串, 指定一个真实路径, 或如果转换不能被执行, 返回 null。

#### 16) getServerInfo

```
public java.lang.String getServerInfo()
```

返回当前 servlet 正运行在其上的 servlet 容器的名字和版本号, 即当前容器的名字和版本号。返回字符串的形式是“servername/versionnumber”。例如, JavaServer Web Development Kit 可能返回字符串“JavaServer Web Dev Kit/1.0”。

servlet 容器可能在基本信息之后以圆括号形式返回其它的可选信息, 例如, JavaServer Web Dev Kit/1.0 (JDK 1.1.6; Windows NT 4.0 x86)。

##### 返回

一个字符串, 至少包含 servlet 容器的名字和版本号。

#### 17) getInitParameter

```
public java.lang.String getInitParameter(java.lang.String name)
```

返回一个字符串, 包含当前上下文中给出名字的初始化参数的值, 或如果该参数不存在, 返回 null。

这个方法可使有效的配置信息对整个“页面应用程序”有用。例如, 它可提供一个网管的 email 地址或一个拥有临界数据系统的名字。

##### 参数

name: 一个字符串, 包含被请求参数的名字。

##### 返回

一个字符串, 至少包含 servlet 容器名字和版本号。

##### 参照

ServletConfig.getInitParameter(java.lang.String)

18) getInitParameterNames

public java.util.Enumeration getInitParameterNames()

返回一个 String 对象的枚举变量,包含当前上下文中所有初始化参数的名字,或如果当前上下文没有初始化参数,返回一个空的枚举变量。

返回

一个 String 对象的枚举变量,包含当前上下文中所有初始化参数的名字。

参照

ServletConfig.getInitParameter(java.lang.String)

19) getAttribute

public java.lang.Object getAttribute(java.lang.String name)

返回当前 servlet 容器中有参数给出名字的属性对象,或如果参数给出名字指定的属性不存在,返回 null。属性使得 servlet 容器可提供这个界面原本没有提供的附加信息给当前 servlet。查看服务端有关它的属性的文档。使用 getAttributeNames 方法可获得支持属性的一个列表。

属性以 java.lang.Object 或某些子类形式返回。属性名遵循包名同样的规则。Java Servlet API 规范保留匹配“java.\*”,“javax.\*”, and “sun.\*”的名字。

参数

name: 一个指定属性名字的字符串。

返回

一个对象,包含属性的值,或如果给出名字的属性不存在,返回 null。

参照

getAttributeNames()

20) getAttributeNames

public java.util.Enumeration getAttributeNames()

返回一个枚举变量,包含当前 servlet 上下文中所有有效的属性名字。使用 getAttribute(java.lang.String) 方法获得一个属性的值。

返回

属性名的一个枚举变量。

参照

getAttribute(java.lang.String)

21) setAttribute

public void setAttribute(java.lang.String name, java.lang.Object object)

绑定一个对象到当前 servlet 上下文的给出属性名中。如果指定的名字已被一个属性使用,这个方法将删除旧的属性,并把该名字与新的属性绑定在一起。

属性名遵循包名同样的规则。Java Servlet API 规范保留匹配“java.\*”,“javax.\*”, and “sun.\*”的名字。

参数



name: 一个指定属性名的字符串。

object: 一个对象, 将绑定到某属性上。

## 22) removeAttribute

```
public void removeAttribute(java.lang.String name)
```

从 servlet 上下文中删除给出名字的属性。删除之后, 后继调用 `getAttribute(java.lang.String)` 方法获取属性值将返回 `null`。

### 参数

name: 一个字符串, 指定将要删除属性的名字。

## 2.6 javax.servlet Class ServletException

### 1. 类层次

```
java.lang.Object
```

```
|
```

```
+-- java.lang.Throwable
```

```
|
```

```
+-- java.lang.Exception
```

```
|
```

```
+-- javax.servlet.ServletException
```

### 2. 已知直接子类

```
UnavailableException
```

### 3. 说明

```
public class ServletException
```

```
extends java.lang.Exception
```

定义一个 servlet 在遇到困难时抛出的普通异常。

### 4. 构造器一览

<code>ServletException()</code>	构造一个新的 servlet 异常。
<code>ServletException(java.lang.String message)</code>	以指定信息构造一个新的 servlet 异常。
<code>ServletException(java.lang.String message, java.lang.Throwable rootCause)</code>	当前 servlet 需要抛出一个异常时, 构造一个新的 servlet 异常并且包含干扰正常操作的“root cause”异常信息和指定的描述信息。
<code>ServletException( java.lang.Throwable rootCause)</code>	当前 servlet 需要抛出一个异常时, 构造一个新的 servlet 异常并且包含干扰正常操作的“root cause”异常信息。

## 5. 方法一览

java.lang.Throwable	getRootCause()	返回导致当前 servlet 发生异常的那个异常。
---------------------	----------------	---------------------------

## 6. 继承自 class java.lang.Throwable 的方法

fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

## 7. 继承自 class java.lang.Object 的方法

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

# 2.7 javax.servlet Class ServletInputStream

## 1. 类层次

```

java.lang.Object
|
+-- java.io.InputStream
|
+-- javax.servlet.ServletInputStream

```

## 2. 说明

public abstract class ServletInputStream extends java.io.InputStream

提供一个输入流以读取客户端请求中的二进制数据, 包括一个非常有用的方法——readLine 方法, 一次读取一行数据。对有些协议, 例如 HTTP POST 和 PUT, ServletInputStream 对象可读取来自客户端的数据。

ServletInputStream 对象通常通过 ServletRequest.getInputStream() 方法获得。

这是一个 servlet 容器实现的抽象类。这个类的子类必须实现 java.io.InputStream.read() 方法。

## 3. 构造器一览

protected ServletInputStream()	什么也不做, 因为这是一个抽象类。
--------------------------------	-------------------

## 4. 方法一览

int	readLine(byte[] b, int off, int len)	读取输入流, 一次一行。
-----	--------------------------------------	--------------

### 5. 继承自 class java.io.InputStream 的方法

available, close, mark, markSupported, read, read, read, reset, skip

### 6. 继承自 class java.lang.Object 的方法

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## 2.8 javax.servlet Class ServletOutputStream

### 1. 类层次

```

java.lang.Object
|
+ -- java.io.OutputStream
|
+ -- javax.servlet.ServletOutputStream
  
```

### 2. 说明

public abstract class ServletOutputStream

extends java.io.OutputStream 提供一个输出流以发送二进制数据到客户端。ServletOutputStream 对象通常通过 ServletResponse.getOutputStream() 方法获得。

这是 servlet 容器实现的抽象类。这个类的子类必须实现 java.io.OutputStream.write(int) 方法。

### 3. 构造器一览

protected	ServletOutputStream()	什么也不做,因为这是一个抽象类。
-----------	-----------------------	------------------

### 4. 方法一览

void	print(boolean b)	写一个 boolean 值到客户端,结尾没有回车换行符。
void	print(char c)	写一个字符到客户端,结尾没有回车换行符。
void	print(double d)	写一个 double 值到客户端,结尾没有回车换行符。
void	print(float f)	写一个 float 值到客户端,结尾没有回车换行符。
void	print(int i)	写一个 int 值到客户端,结尾没有回车换行符。
void	print(long l)	写一个 long 值到客户端,结尾没有回车换行符。

续表

void	print(java.lang.String s)	写一个字符串到客户端,结尾没有回车换行符。
void	println()	写一个回车换行符到客户端。
void	println(boolean b)	写一个 boolean 值到客户端,后跟回车换行符。
void	println(char c)	写一个字符到客户端,后跟回车换行符。
void	println(double d)	写一个 double 值到客户端,后跟回车换行符。
void	println(float f)	写一个 float 值到客户端,后跟回车换行符。
void	println(int i)	写一个 int 值到客户端,后跟回车换行符。
void	println(long l)	写一个 long 值到客户端,后跟回车换行符。
void	println(java.lang.String s)	写一个字符串到客户端,后跟回车换行符。

### 5. 继承自 class java.io.OutputStream 的方法

close, flush, write, write, write

### 6. 继承自 class java.lang.Object 的方法

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## 2.9 javax.servlet Interface ServletRequest

### 1. 所有已知子界面

HttpServletRequest

### 2. 说明

public interface **ServletRequest**

定义一个提供客户端的请求信息给 servlet 的对象。Servlet 容器创建一个 ServletRequest 对象并把它作为该 servlet 的 service 方法的一个参数。

ServletRequest 对象提供这些数据,包括参数名和值、属性、输入流。扩展自 ServletRequest 的界面能提供另外的特殊协议的数据(例如,由 HttpServletRequest 提供 HTTP 数据)。

### 3. 方法一览

Java.lang.Object	getAttribute( java.lang.String name)	以对象形式返回给出名字指定的属性的值,或如果这样的属性不存在,返回 null。
java.util. Enumeration	getAttributeNames()	返回一个枚举变量,包含当前请求的所有有效属性的名字。
Java.lang.String	getCharacterEncoding()	返回当前请求体中的字符编码方式的名字。

续表

Int	getLength()	返回请求体的有效长度,以字节为单位,或长度未知,返回 -1。
Java.lang.String	getContentType()	返回请求体的 MIME 类型,或如果类型未知,返回 null。
ServletInputStream	getInputStream()	使用 ServletInputStream 对象获得二进制形式的请求体。
Java.util.Locale	getLocale()	客户端首选的地域对象。
java.util. Enumeration	getLocales()	客户端首选地域对象的枚举。
Java.lang.String	getParameter( java.lang.String name)	以 String 形式返回由 name 指定的请求参数的值,或如果参数不存在,返回 null。
java.util. Enumeration	getParameterNames()	返回一个 String 对象的枚举变量,包含当前请求中所有参数的名字。
Java.lang.String[]	getParameterValues( java.lang.String name)	返回一个 String 对象的数组,包含由 name 指定的请求参数所有的值。
Java.lang.String	getProtocol()	以“协议/主版本号.次版本号”形式返回当前请求使用协议的名字和版本,例如,HTTP/1.1。
java.io. BufferedReader	getReader()	使用 BufferedReader 对象获得字符数据形式的请求体。
Java.lang.String	getRealPath( java.lang.String path)	不赞成使用。 Java Servlet API 2.1 版本中,使用 ServletContext.getRealPath(java.lang.String)代替。
Java.lang.String	getRemoteAddr()	返回发送当前请求的客户端的 IP 地址。
Java.lang.String	getRemoteHost()	返回发送当前请求的客户端的全名,或如果名字不能确定,返回 IP 地址。
RequestDispatcher	getRequestDispatcher( java.lang.String path)	返回一个 RequestDispatcher 对象,该对象封装了已给出路径定位资源。
java.lang.String	getScheme()	返回生成当前请求的方案的名字,例如,http,https,或者 ftp。
java.lang.String	getServerName()	返回接收当前请求的服务端的主机名。
int	getServerPort()	返回接收当前请求的端口号。
boolean	isSecure()	返回一个 boolean 变量,指明当前请求是否由安全通信方式生成,例如,HTTPS。
void	removeAttribute( java.lang.String name)	从当前请求中删除一个属性。
Void	setAttribute(java.lang. String name, java.lang. Object o)	存储一个属性在当前请求中。

## 4. 方法详解

### 1) getAttribute

`public java.lang.Object getAttribute(java.lang.String name)`

以对象形式返回由参数给出名指定的属性的值,或如果这样的属性不存在,返回 `null`。

有两种方法设置属性。Servlet 容器可设置属性使一个请求的自定义信息有效。例如,使用 HTTPS 生成的请求,属性 `javax.servlet.request.X509Certificate` 用来获得客户端的认证信息。属性也可使用 `setAttribute(java.lang.String, java.lang.Object)` 方法设置。这使得信息可以在 `RequestDispatcher` 方法调用前嵌入一个请求。

属性名遵循与包名相同的约定。这个规范保留匹配“`java.*`”,“`javax.*`”,“`sun.*`”的名字。

参数

`name`: 一个指定属性名字的字符串。

返回

一个对象,包含指定属性的值,或如果指定属性不存在,返回 `null`。

### 2) getAttributeNames

`public java.util.Enumeration getAttributeNames()`

返回一个枚举变量,包含当前请求的所有有效属性的名字。如果当前请求中没有有效的属性,这个方法返回一个空的枚举变量。

返回

一个 `String` 对象的枚举变量,包含当前请求中所有属性的名字。

### 3) getCharacterEncoding

`public java.lang.String getCharacterEncoding()`

返回当前请求体中的字符编码方式的名字。如果当前请求没有指定字符编码方式,返回 `null`。

返回

一个字符串,包含字符编码方式的名字,或如果当前请求没有指定字符编码方式,返回 `null`。

### 4) getContentLength

`public int getContentLength()`

返回请求体的有效长度,以字节为单位,或长度未知,返回 `-1`。对 HTTP servlet 来说,与 CGI 变量 `CONTENT_LENGTH` 有相同的值。

返回

一个整数,包含当前请求体的长度,或如果长度未知,返回 `-1`。

### 5) getContentType

`public java.lang.String getContentType()`

返回请求体的 MIME 类型,或如果类型未知,返回 `null`。对 HTTP servlet 来说,与 CGI 变量 `CONTENT_TYPE` 有相同的值。

返回

一个字符串,包含当前请求的 MIME 类型,或如果类型未知,返回 -1。

#### 6) getInputStream

public ServletInputStream **getInputStream()** throws java.io.IOException

使用 ServletInputStream 对象获得二进制形式的请求体。可调用这个方法或者 getReader() 方法读取请求体。

返回

一个 ServletInputStream 对象,包含当前请求的请求体。

抛出

java.lang.IllegalStateException——如果当前请求已经调用 getReader() 方法。

java.io.IOException——如果一个输入或输出异常发生。

#### 7) getParameter

public java.lang.String **getParameter**(java.lang.String name)

以 String 形式返回由 name 指定的请求参数的值,或如果参数不存在返回 null。请求参数是当前请求发送的额外信息。对 HTTP servlet,参数包含在查询串或提交表单数据中。

当确定参数只有一个值时,应当使用这个方法。如果参数可能有几个值,应使用 getParameterValues(java.lang.String) 方法。

如果对一个多值参数使用这个方法,返回值等于使用 getParameterValues() 方法返回的数组的第一个值。

如果在请求体中发送参数数据,例如出现在一个 HTTP POST 请求中,然后通过 getInputStream() 或 getReader() 方法直接读取参数体将干扰这个方法的执行。

参数

name: 一个指定参数名字的字符串。

返回

一个字符串,表示参数的(一个)值。

参照

getParameterValues(java.lang.String)

#### 8) getParameterNames

public java.util.Enumeration **getParameterNames**()

返回一个 String 对象的枚举变量,包含当前请求中所有参数的名字。如果当前请求中没有参数,这个方法返回一个空的枚举变量。

返回

一个 String 对象的枚举变量,每个 String 包含一个请求参数的名字,或如果当前请求中没有参数,返回一个空的枚举变量。

#### 9) getParameterValues

public java.lang.String[] **getParameterValues**(java.lang.String name)

返回一个 String 对象的数组,包含由 name 指定的请求参数所有的值,或如果指定参数不存在,返回 null。

如果指定参数只有一个值,数组长度为1。

返回

一个 String 对象的数组,包含指定参数所有的值。

参照

`getParameter(java.lang.String)`

10) `getProtocol`

`public java.lang.String getProtocol()`

以“协议/主版本号.次版本号”形式返回当前请求使用协议的名字和版本,例如,HTTP/1.1。对 HTTP servlet,返回值与 CGI 变量 `SERVER_PROTOCOL` 的值相同。

返回

一个字符串,包含协议名字和版本号。

11) `getScheme`

`public java.lang.String getScheme()`

返回生成当前请求的模式,例如,http,https,或者 ftp。不同的方案有不同的 URLs 构造规则。

返回

一个字符串,包含用于生成当前请求的方案的名字。

12) `getServerName`

`public java.lang.String getServerName()`

返回接收当前请求的服务端的主机名。对 HTTP servlet,与 CGI 变量 `SERVER_NAME` 有相同的值。

返回

一个字符串,包含接收当前请求的服务端的主机名。

13) `getServerPort`

`public int getServerPort()`

返回接收当前请求的端口号。对 HTTP servlet,与 CGI 变量 `SERVER_PORT` 的值相同。

返回

一个整数,指定端口号。

14) `getReader`

`public java.io.BufferedReader getReader() throws java.io.IOException`

使用 `BufferedReader` 对象获得字符数据形式的请求体。读取器(reader)根据请求体使用的字符编码方式翻译字符数据。可调用这个方法或 `getReader()` 读取请求体。

返回

一个 `BufferedReader` 对象,包含当前请求体。

抛出

`java.io.UnsupportedEncodingException`——如果不支持使用的字符集编码方式并且文本不能被解码。

`java.lang.IllegalStateException`——如果当前请求已调用 `getInputStream()` 方法。



java.io.IOException——如果一个输入或输出异常发生。

参照

getInputStream()

15) getRemoteAddr

public java.lang.String **getRemoteAddr()**

返回发送当前请求的客户端的 IP 地址。对 HTTP servlet 与 CGI 变量 REMOTE\_ADDR 的值相同。

返回

一个字符串,包含发送当前请求的客户端的 IP 地址。

16) getRemoteHost

public java.lang.String **getRemoteHost()**

返回发送当前请求的客户端的全名,或如果名字不能确定,返回 IP 地址。对 HTTP servlet,与 CGI 变量 REMOTE\_HOST 的值相同。

返回

一个字符串,包含客户端的全名。

17) setAttribute

public void **setAttribute**(java.lang.String name, java.lang.Object o)

存储一个属性在当前请求中。属性在两个请求之间设置。这个方法经常与 RequestDispatcher 联合使用。

属性名遵循与包名相同的约定。这个规范保留匹配“java.\* ,javax.\* ,sun.\*”的名字。

参数

name: 一个指定属性名字的字符串。

o: 欲存储的对象。

18) removeAttribute

public void **removeAttribute**(java.lang.String name)

从当前请求中删除一个属性。这个方法通常没有必要等到这个请求被处理完。

属性名遵循与包名相同的约定。这个规范保留匹配“java.\* ,javax.\* ,sun.\*”的名字。

参数

name: 一个指定将要删除属性名字的字符串。

19) getLocale

public java.util.Locale **getLocale()**

返回根据可接受语言头信息得出的用户首选地域,以该地域发送的内容能为客户端接受。如果客户端请求没有包含可接受语言头信息,这个方法返回服务器默认的地域。

返回

客户端首选的地域。

地域对象表示一个特定的地理、政治或文化区域。一个需要地域信息才能执行的操作叫地域相关操作,例如显示数字是一个地域相关操作——数字将依据用户或用户所在国家、区域或文化的相关约定格式化。

## 20) getLocales

```
public java.util.Enumeration getLocales()
```

以枚举形式返回以首选地域开始并做递减顺序排列的地域信息。客户端可接受的地域基于可接受语言头信息,如果客户端请求没有包含可接受语言头信息,这个方法返回包含服务器默认的地域的枚举。

返回

客户端首选地域对象的枚举。

## 21) isSecure

```
public boolean isSecure()
```

返回一个 boolean 变量,指明当前请求是否由安全通信方式生成,例如,HTTPS。

返回

一个 boolean 变量,指明当前请求是否由安全通信方式生成。

## 22) getRequestDispatcher

```
public RequestDispatcher getRequestDispatcher(java.lang.String path)
```

返回一个 RequestDispatcher 对象,该对象封装了指定路径上的资源。RequestDispatcher 对象可将请求转发给资源或在响应中包含资源。资源可是动态的或静态的。

指定的路径名可是相对的,尽管它不能在当前 servlet 上下文之外扩展。如果路径以“/”开始,那么它被解释为相对当前上下文根路径的相对路径。如果 servlet 容器不能返回一个 RequestDispatcher,这个方法返回 null。

这个方法与 ServletContext.getRequestDispatcher(java.lang.String)方法不同的是这个方法可接受相对路径。

参数

path: 一个指定资源路径的字符串。

返回

一个 RequestDispatcher 对象,该对象封装了指定路径上的资源。

参照

RequestDispatcher, ServletContext.getRequestDispatcher(java.lang.String)

## 23) getRealPath

```
public java.lang.String getRealPath(java.lang.String path)
```

不赞成使用, Java Servlet API 2.1 版本中,使用 ServletContext.getRealPath(java.lang.String)代替。

## 2.10 javax.servlet Interface ServletResponse

### 1. 所有已知子界面

HttpServletResponse

## 2. 说明

**public interface ServletResponse**

定义一个使 servlet 可将响应发送给客户端的对象。servlet 容器创建一个 ServletResponse 对象并把它作为该 servlet 的 service 方法的一个参数。

发送 MIME 体响应中的二进制数据,使用 `getOutputStream()` 方法返回的 `ServletOutputStream` 对象;发送字符数据,使用 `getWriter` 返回的 `PrintWriter` 对象。发送二进制和文本的混合数据,例如建立一个有几个部分的响应,使用 `ServletOutputStream` 对象手工管理类型为字符的这部分数据。

MIME 体响应的字符集可用 `setContentType(java.lang.String)` 方法指定。例如,“text/html; charset = Shift\_JIS”。字符集可用 `setLocale(java.util.Locale)` 方法替换。如果没有指定字符集,将用 ISO-8859-1。`SetContentType` 或 `setLocale` 方法必须在 `getWriter` 构造 `writer` 之前被调用。

参考 Internet 上的 RFCs 例如 RFC 2045 了解更多 MIME 的信息。如 SMTP 和 HTTP 定义了 MIME 的某个方面,这些协议仍在演化中。

## 3. 方法一览

void	<code>flushBuffer()</code>	迫使缓冲区中的任何内容都写到客户端。
int	<code>getBufferSize()</code>	返回当前响应的实际缓冲区大小。
java.lang.String	<code>getCharacterEncoding()</code>	返回当前响应的 MIME 体使用的字符集的名字。
java.util.Locale	<code>getLocale()</code>	返回当前响应使用的地域(信息)对象。
ServletOutputStream	<code>getOutputStream()</code>	返回一个适合在响应中写二进制数据的 <code>ServletOutputStream</code> 对象
java.io.PrintWriter	<code>getWriter()</code>	返回一个 <code>PrintWriter</code> 对象,以发送字符文本到客户端。
boolean	<code>isCommitted()</code>	返回一个 boolean 变量指明当前响应是否已被提交。
void	<code>reset()</code>	清除缓冲区中的任何数据,如状态码和头部。
void	<code>setBufferSize(int size)</code>	设置响应体优先使用的缓冲区大小。
void	<code>setContentLength(int len)</code>	设置 HTTP servlets 响应的内容体长度,即设置 HTTP Content-Length(内容长度)头部。
void	<code>setContentType( java.lang.String type)</code>	设置发送到客户端的响应的内容类型。
void	<code>setLocale( java.util.Locale loc)</code>	设置当前响应的地域,即设置适当的头部(包括 Content-Type 的字符集)。

## 4. 方法详解

### 1) getCharacterEncoding

public java.lang.String **getCharacterEncoding()**

返回当前响应的 MIME 体使用的字符集的名字。如果字符集还没有分配,隐含设置为 ISO-8859-1 (Latin-1)。

参考 RFC2047 (<http://ds.internic.net/rfc/rfc2045.txt>) 了解更多有关字符编码和 MIME 的信息。

返回

一个指定字符集名字的字符串。例如 ISO-8859-1。

### 2) getOutputStream

public ServletOutputStream **getOutputStream()**

throws java.io.IOException

返回一个适合在响应中写二进制数据的 ServletOutputStream 对象。servlet 容器不会编码二进制数据。调用这个方法或 `getWriter()` 方法写响应体。

返回

一个写二进制的 ServletOutputStream 对象。

抛出

java.lang.IllegalStateException——如果当前响应已调用 `getWriter` 方法。

java.io.IOException——如果一个输入或输出异常发生。

参照

`getWriter()`

### 3) getWriter

public java.io.PrintWriter **getWriter()**

throws java.io.IOException

返回一个 PrintWriter 对象以发进字符文本到客户端。使用的字符编码由 `setContentType(java.lang.String)` 方法中的“charset=”属性指定,`setContentType(java.lang.String)` 方法必须在使字符集生效(调用这个方法)之前被调用。

如果必要,响应的 MIME 类型可被改变以映射使用的字符编码。

调用这个方法或 `getOutputStream()` 方法写响应体。

返回

一个可将字符文本发送到客户端的 PrintWriter 对象。

抛出

java.io.UnsupportedEncodingException——如果由 `setContentType()` 方法指定的字符集不可用。

java.lang.IllegalStateException——如果当前响应已调用 `getOutputStream` 方法。

java.io.IOException——如果一个输入或输出异常发生。

参照

`getOutputStream()`, `setContentType(java.lang.String)`

## 4) setContentLength

```
public void setContentLength(int len)
```

设置 HTTP servlets 响应的内容体长度,即设置 HTTP Content - Length(内容长度)头部。

## 参数

len:一个整数,指定返回客户端内容的长度,设置 Content - Length 头部。

## 5) setContentType

```
public void setContentType(java.lang.String type)
```

设置发送到客户端的响应的内容类型。内容类型可以包括使用的字符编码类型。例如,text/html; charset = ISO - 8859 - 4。

## 参数

type:一个指定内容 MIME 类型的字符串。

## 参照

```
getOutputStream(),getWriter()
```

## 6) setBufferSize

```
public void setBufferSize(int size)
```

设置响应体优先使用的缓冲区大小。servlet 容器将使用一个至少与请求大小相同的缓冲区。使用 getBufferSize 方法可获得实际使用的缓冲区大小。

在内容被实际发送之前,更大的缓冲区可容纳更多的内容,因而应提供更多时间给 servlet 以设置适当的状态码和头部。较小的缓冲区减小服务端内存负载并且使客户端更快的接收数据。

这个方法必须在写任何响应体内容之前被调用,如果内容已被写,这个方法抛出一个 IllegalStateException 异常。

## 参数

size:优先使用的缓冲区大小。

## 抛出

java.lang.IllegalStateException——如果这个方法在内容已被写出之后调用。

## 参照

```
getBufferSize(),flushBuffer(),isCommitted(),reset()。
```

## 7) getBufferSize

```
public int getBufferSize()
```

返回当前响应的实际缓冲区大小。如果没使用缓冲区,这个方法返回 0。

## 返回

实际使用的缓冲区大小。

## 参照

```
setBufferSize(),flushBuffer(),isCommitted(),reset()。
```

## 8) flushBuffer

```
public void flushBuffer() throws java.io.IOException
```

迫使缓冲区中的任何内容都写到客户端。调用这个方法,自动提交响应,意味着

写状态码和头部。

参照

`setBufferSize()`, `getBufferSize()`, `isCommitted()`, `reset()`。

9) `isCommitted`

`public boolean isCommitted()`

返回一个 `boolean` 变量指明当前响应是否已被提交。一个被提交的响应,其状态码和头部已经写好。

返回

一个 `boolean` 指明响应是否已被提交。

参照

`setBufferSize()`, `getBufferSize()`, `flushBuffer()`, `reset()`。

10) `reset`

`public void reset()`

清除缓冲区中的任何数据如状态码和头部。如果响应已被提交,这个方法抛出一 `IllegalStateException` 异常。

抛出

`java.lang.IllegalStateException` — 如果响应已被提交。

参照

`setBufferSize()`, `getBufferSize()`, `flushBuffer()`, `isCommitted()`。

11) `setLocale`

`public void setLocale(java.util.Locale loc)`

设置当前响应使用的地域,设置适当的头部(包括 `Content-Type` 的字符集)。这个方法应在 `getWriter()` 方法之前被调用。默认,响应地域是服务端的默认地域。

参数

`loc`:响应的地域(信息)。

参照

`getLocale()`

12) `getLocale`

`public java.util.Locale getLocale()`

返回当前响应使用的地域。

参照

`setLocale(java.util.Locale)`

## 2.11 javax.servlet Interface SingleThreadModel

### 1. 说明

`public interface SingleThreadModel`

使 servlet 一次只处理一个请求。这个界面没有方法。

如果一个 servlet 实现这个界面,必须保证该 servlet 的 service 方法中不会有两个及以上线程并发执行。通过同步访问该 servlet 的唯一实例或通过维持该 servlet 实例的一个缓冲池并把每个新的请求分配给一个空闲的 servlet,该 servlet 容器可保证不会有两个及以上线程并发执行。

如果一个 servlet 实现这个界面,该 servlet 将是线程安全的。然而,这个界面不能防止由 servlet 访问共享资源(例如静态类变量或该 servlet 作用域之外的类)产生的同步问题。

## 2.12 javax.servlet Class UnavailableException

### 1. 类层次

```
java.lang.Object
|
+-- java.lang.Throwable
    |
    +-- java.lang.Exception
        |
        +-- javax.servlet.ServletException
            |
            +-- javax.servlet.UnavailableException
```

### 2. 说明

```
public class UnavailableException extends ServletException
```

定义 servlet 抛出的一个异常,指明该 servlet 是永久的还是暂时的不可获得。

当一个 servlet 是永久的不可获得时,该 servlet 发生错误并且不能处理请求,直到某些动作发生。例如,该 servlet 可能被配置错误,或它的状态被破坏。Servlet 应将错误和需要的纠正动作都记入日志。

当一个 servlet 是暂时的不可获得时,如果它不能处理请求,那么立刻可归,而为某些系统范围的问题。例如,第三层服务不可进入,或内存,磁盘空间不足。系统管理员可能需要采取纠正动作。

Servlet 容器可安全的以同样的方式处理这两种类型的异常。然而,有效的处理暂时不可获得异常可使 servlet 容器的性能更加稳定。servlet 容器应该封锁对该 servlet 的请求一段时间,这比拒绝它们直至 servlet 容器重启要好得多。

### 3. 构造器一览

UnavailableException(int seconds, Servlet servlet, java.lang.String msg)	不赞成使用。Java Servlet API 2.2 中使用 UnavailableException(String, int)代替。
UnavailableException(Servlet servlet, java.lang.String msg)	不赞成使用。Java Servlet API 2.2 中使用 UnavailableException(String)代替。
UnavailableException(java.lang.String msg)	以指明 servlet 是永久不可获得的描述信息构造一个新的异常。
UnavailableException(java.lang.String msg, int seconds)	以指明 servlet 是暂时不可获得的描述信息和一个多长时间不可获得的估计值构造一个新的异常。

### 4. 方法一览

Servlet	getServlet()	不赞成使用。Java Servlet API 2.2 中没有替换者。返回 servlet, 它被报告认为是不可获得的。
int	getUnavailableSeconds()	返回当前 servlet 预期的暂时不可获得时间, 以秒为单位。
boolean	isPermanent()	返回一个 boolean, 指明当前 servlet 是否是永久不可获得。

### 5. 继承自 class javax.servlet.ServletException 的方法

getRootCause

### 6. 继承自 class java.lang.Throwable 的方法

fillInStackTrace, getLocalizedMessage, getMessage, printStackTrace, printStackTrace, printStackTrace, toString

### 7. 继承自 class java.lang.Object 的方法

clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait



## 附录 3 javax.servlet.http 包

### 3.1 javax.servlet.http Class Cookie

#### 1. 类层次

java.lang.Object

| 继承

实现

+——javax.servlet.http.Cookie———java.lang.Cloneable

#### 2. 说明

```
package javax.servlet.http;
```

```
public class Cookie extends Object
```

```
    implements Cloneable
```

Cookie 类用来表示在 HTTP 协议和 HTTPS 协议中进行会话管理的数据。Cookie 是客户端(特别是 Web 浏览器)用来在客户会话中保存状态信息的少量数据。

Cookie 是有名字的,而且每一个 Cookie 具有唯一的值。然而由于目前的 Web 浏览器在使用 Cookie 上各有不同,所以写 servlet 的时候不能过多依赖于它们。

#### 3. 方法一览

java.lang.Object	clone()	覆盖标准的 java.lang.Object.clone 方法,返回当前 Cookie 的一个拷贝。
java.lang.String	getComment()	返回描述当前 Cookie 意图的注释,如果没有就返回空(null)。
java.lang.String	getDomain()	返回给当前 Cookie 设置的域名。
int	getMaxAge()	返回当前 Cookie 的最大寿命,默认以“秒”为单位。 -1表明当前 Cookie 持续到浏览器关闭。
java.lang.String	getName()	返回当前 Cookie 的名字。
java.lang.String	getPath()	返回浏览器发送回来的当前 Cookie 在服务端的路径。
boolean	getSecure()	如果浏览器只用安全协议发送 Cookie,返回“true”,如果浏览器使用任何协议发送 Cookie,返回“false”。

续表

java.lang.String	getValue()	返回当前 Cookie 的值。
int	getVersion()	返回当前 Cookie 遵循协议的版本号。
void	setComment(String)	设置描述当前 Cookie 意图的注释。
void	setDomain(String)	设置当前 Cookie 应当出现在其内的域。
void	setMaxAge(int)	设置当前 Cookie 以“秒”为单位的最大寿命。
void	setPath(String)	为当 Cookie 设置客户端将其发送回来的路径。
void	setSecure(boolean)	设置浏览器是否只使用安全协议发送当前 Cookie, 例如 HTTPS 或 SSL。
void	setValue(String)	当前 Cookie 被创建后, 给其赋个新值。
void	setVersion(int)	设置当前 Cookie 遵循协议的版本号。

#### 4. 继承自 class java.lang.Object 的方法

equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait。

#### 5. 构造函数详解

Cookie

```
public Cookie(java.lang.String name, java.lang.String value)
```

以指定名字和值构造一个 Cookie。

名字必须遵循 RFC 2109(Request For Comments, 请求注解, Internet 标准)。这意味着它只能包含 ASCII 字符而不能包含逗号、分号或空白; 以及不能以“\$”字符打头。Cookie 创建后, 其名字不能改变。

值可以是服务端选择发送的任意字符串, 可能只有服务端对这个值感兴趣。Cookie 创建后, 其值可用 setValue 方法改变。

默认 Cookie 依照 Netscape Cookie 规范创建。版本号可用 setVersion 方法改变。

参数

name: 指定 Cookie 名字的字符串。

value: 指定 Cookie 值的字符串。

抛出

java.lang.IllegalArgumentException: 如果 Cookie 的名字包含非法的字符或者它是 Cookie 协议的保留字。

#### 6. 方法详解

##### 1) setComment

```
public void setComment(java.lang.String purpose)
```

设置描述当前 Cookie 意图的注释。如果浏览器需要把这个 Cookie 显示给用户, 那么注释是有用的。

### 参数

purpose: 一个字符串,指定显示给用户的注释。

#### 2) getComment

```
public java.lang.String getComment()
```

返回描述当前 Cookie 意图的注释,如果没有就返回空(null)。

### 返回

包含注释的一个字符串,如果没有;注释为空(null)。

#### 3) SetDomain

```
public void setDomain(java.lang.String pattern)
```

设置当前 Cookie 应当出现在其内的域。

域名的形式由 RFC 2109 指定。一个域名以一个点开始(.foo.com),意味着在一个特定的域名系统(DNS)中这个 Cookie 对服务端是可见的(例如,www.foo.com,但是不是a.b.foo.com)。默认 Cookies 只能返回发送它们的服务端。

### 参数

pattern: 包含域名的一个字符串,在该域名内,当前 Cookie 是可见的,形式依照 RFC 2109。

#### 4) getDomain

```
public java.lang.String getDomain()
```

返回给当前 Cookie 设置的域名。域名的形式由 RFC 2109 设置。

返回: 包含域名的一个字符串。

#### 5) SetMaxAge

```
public void setMaxAge(int expiry)
```

设置当前 Cookie 以“秒”为单位的最大寿命。

正值指明经过那么多秒之后,当前 Cookie 将终止。注意:当前 Cookie 终止时,其值是大寿命值,而不是该 Cookie 的当前寿命。

负值意味着当前 Cookie 不会被长期存储,当 Web 浏览器退出时,它将被删除。零值导致当前 Cookie 被删除。

### 参数

expiry: 一个整数,以“秒”为单位指定当前 Cookie 的最大寿命。如果为负,意味着当前 Cookie 不会被存储,如果为零,删除当前 Cookie。

#### 6) getMaxAge

```
public int getMaxAge()
```

返回当前 Cookie 的最大寿命,默认以“秒”为单位。-1 表明当前 Cookie 持续到浏览器关闭。

### 返回

一个整数,当前 Cookie 以“秒”为单位的最大寿命值。如果为负,意味着当前 Cookie 持续到浏览器关闭。

#### 7) SetPath

```
public void setPath(java.lang.String uri)
```

为当前 Cookie 设置客户端将其发送回来的路径。

当前 Cookie 对指定的目录中的所有页面都是可见的,并且对那个目录的子目录中的所有页面也是可见的。一个 Cookie 的路径必须包含设置它的 servlet,例如:/ catalog,使 Cookie 对服务端/catalog 下的所有目录都是可见的。

参数

uri:指定路径的一个字符串。

#### 8) getPath

public java.lang.String **getPath()**

返回浏览器发送回来的当前 Cookie 在服务端的路径。当前 Cookie 对服务端所有子路径都是可见的。

返回

一个字符串,包含一个 servlet 名的路径。例如:/ catalog。

#### 9) setSecure

public void **setSecure**(boolean flag)

设置浏览器是否只使用安全协议发送当前 Cookie,例如 HTTPS 或 SSL。

默认值是“false”。

参数

flag:如果为“true”,只使用安全协议从浏览器发送 Cookie 到服务端。如果为“false”,使用任意协议发送。

#### 10) getSecure

public boolean **getSecure**()

如果浏览器只用安全协议发送 Cookie,返回“true”,如果浏览器使用任何协议发送 Cookie,返回“false”。

返回

如果浏览器能使用任何标准协议,返回“true”,否则返回“false”。

#### 11) getName

public java.lang.String **getName**()

返回当前 Cookie 的名字。Cookie 创建后,其名字不能改变。

返回

指定当前 Cookie 名字的一个字符串。

#### 12) setValue

public void **setValue**(java.lang.String newValue)

当前 Cookie 被创建后,给其赋个新值。如果使用二进制值,就需要使用 BASE64 编码。

0 版本中的 Cookies,其值不能包含空白、方括号、圆括号、等号、逗号、双引号、斜杠、问号、冒号和分号。对所有浏览器,空值可能没有相同的行为。

参数

newValue: 指定新值的一个字符串。

13) `getValue`

```
public java.lang.String getValue()
```

返回当前 Cookie 的值。

返回

包含当前 Cookie 的当前值的一个字符串。

14) `getVersion`

```
public int getVersion()
```

返回当前 Cookie 遵循协议的版本号。版本 1 遵循 RFC2109, 版本 0 遵循 Netscape 起草的原始 Cookie 规范。浏览器提供的 Cookies 使用并且识别这个浏览器的 Cookie 版本。

返回

如果当前 Cookie 遵循原始的 Netscape 规范, 返回 0, 如果遵循 RFC2109, 返回 1。

15) `setVersion`

```
public void setVersion(int v)
```

设置当前 Cookie 遵循协议的版本号。版本 0 遵循原始的 Netscape 规范, 版本 1 遵循 RFC2109。

因为 RFC 2109 仍然有些新, 考虑到版本 1 的实验性, 在生成站点时, 不要使用它。

参数

v: 如果当前 Cookie 遵循原始的 Netscape 规范为 0, 如果遵循 RFC2109 为 1。

16) `clone`

```
public java.lang.Object clone()
```

覆盖标准的 `java.lang.Object.clone` 方法, 返回当前 Cookie 的一个拷贝。

覆盖

class `java.lang.Object` 中 `clone` 方法。

## 3.2 javax.servlet.http Class HttpServlet

### 1. 类层次

```
java.lang.Object
```

```
|
```

```
+-- javax.servlet.GenericServlet
```

```
|
```

```
+-- javax.servlet.http.HttpServlet
```

### 2. 说明

```
public abstract class HttpServlet
```

```
extends GenericServlet
```

```
implements java.io.Serializable
```

---

提供一个可扩展(子类)的抽象类以创建适合 Web 站点的 HTTP servlet。

HttpServlet 的子类必须覆盖至少一个方法,通常是下面中的一个:

- (1) doGet, 如果 servlet 支持 HTTP GET 请求。
- (2) doPost, for HTTP POST requests
- (3) doPut, for HTTP PUT requests
- (4) doDelete, for HTTP DELETE requests
- (5) init and destroy, 管理 servlet 生命周期内拥有的资源。
- (6) getServletInfo, servlet 用它提供关于自身的信息。

几乎没有理由覆盖 service 方法。Service 通过为每个 HTTP 请求类型(上文列出的 doXXX 方法)分配相应的处理方法来处理标准的 HTTP 请求。

同样的,几乎没有理由覆盖 doOptions 和 doTrace 方法。

典型的, servlets 在多线程服务端(器)上运行,所以应意识到 servlet 必须处理并发请求,并且小心对共享资源的同步访问。共享资源包括在内存中的数据如实例、类变量和外部对象如文件、数据库连接、网络连接。

### 3. 构造器一览

HttpServlet()	什么也不做,因为这是一个抽象类。
---------------	------------------

### 4. 方法一览

protected void	doDelete(HttpServletRequest req, HttpServletResponse resp)	由服务端(经 service 方法)调用,允许 servlet 处理 DELETE 请求。
protected void	doGet(HttpServletRequest req, HttpServletResponse resp)	由服务端(经 service 方法)调用,允许 servlet 处理 GET 请求。
protected void	doOptions(HttpServletRequest req, HttpServletResponse resp)	由服务端(经 service 方法)调用,允许 servlet 处理 OPTIONS 请求。
protected void	doPost(HttpServletRequest req, HttpServletResponse resp)	由服务端(经 service 方法)调用,允许 servlet 处理 POST 请求。
protected void	doPut(HttpServletRequest req, HttpServletResponse resp)	由服务端(经 service 方法)调用,允许 servlet 处理 PUT 请求。
protected void	doTrace(HttpServletRequest req, HttpServletResponse resp)	由服务端(经 service 方法)调用,允许 servlet 处理 TRACE 请求。
protected long	getLastModified(HttpServletRequest req)	返回 HttpServletRequest 对象最近被修改的时间,用自 1/1/1970 GMT 午夜以来的毫秒值表示。
protected void	service(HttpServletRequest req, HttpServletResponse resp)	从 public service 方法接收标准 HTTP 请求并且把它们分配给这个类中定义的 doXXX 方法。
void	service(ServletRequest req, ServletResponse res)	把客户端的请求分配给 protected service 方法。

## 5. 继承自 class javax.servlet.GenericServlet 的方法

destroy, getInitParameter, getInitParameterNames, getServletConfig, getServletContext, getServletInfo, getServletName, init, init, log, log

## 6. 继承自 class java.lang.Object 的方法

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

## 7. 构造器详解

```
public HttpServlet()
```

什么也不做,因为这是一个抽象类。

## 8. 方法详解

### 1) doGet

```
protected void doGet(HttpServletRequest req, HttpServletResponse resp)  
    throws ServletException, java.io.IOException
```

由服务端(经 service 方法)调用,允许 servlet 处理 GET 请求。覆盖这个方法以支持 GET 请求,也自动支持 HTTP HEAD 请求。HEAD 请求是这样一个 GET 请求——对该 GET 请求的响应没有响应体,只有请求头部各域(字段)。

覆盖这个方法时,读取请求数据,写响应头部,获得响应的 writer 或输出流对象,最后,写响应数据,最好包括内容类型和编码。使用一个 PrintWriter 对象返回响应时,在访问该 PrintWriter 对象前设置内容类型。

提交响应前,servlet 容器必须写头部,因为在 HTTP 中,头部必须在响应体前被发送。

在可能的地方,设置内容长度(Content-Length)(使用 ServletResponse.setContentLength(int) 方法),允许 servlet 容器使用一个持久连接返回对客户端的响应,以提高性能。如果全部响应都装入了响应缓冲中,那么内容长度可被自动设置。

GET 方法应当是安全的,即没有任何可由用户控制的副作用。例如,大多数表单查询没有副作用。如果一个客户端请求有意改变存储的数据,请求应该使用其它的 HTTP 方法。

GET 方法也应当是幂等的,意思是它可被安全的重复。有时使一个方法安全也就是使它是幂等的。例如,重复查询是安全的也是幂等的,但是在线购物或修改数据,既不安全也不幂等。

如果请求格式不正确,doGet 返回一个 HTTP“Bad Request”信息。

### 参数

req: 一个 HttpServletRequest 对象,包含客户端已产生的对 servlet 的请求。

req - an HttpServletRequest object that contains the request the client has made of the servlet

resp: 一个 HttpServletResponse 对象, 包含 servlet 发送给客户端的响应。

resp - an HttpServletResponse object that contains the response the servlet sends to the client

抛出

java.io.IOException —— 当 servlet 处理 GET 请求时, 如果检测到输入或输出错误。

ServletException —— 如果 GET 请求不能被处理。

参照

ServletResponse.setContentType(java.lang.String)

2) getLastModified

protected long **getLastModified**(HttpServletRequest req)

返回 HttpServletRequest 对象最近被修改的时间, 用自 1/1/1970 GMT 午夜以来的毫秒值表示。如果时间是未知的, 这个方法返回一个负值(默认)。

那些支持 HTTP GET 请求并且能迅速确定其最近更改时间的 servlet 应当覆盖这个方法。这使得浏览器和代理的缓存机制更加有效的工作, 减少了服务端的负载和对网络资源的占用。

参数

req: 发送到 servlet 的 HttpServletRequest 对象。

返回

一个 long 类型整数, 指定 HttpServletRequest 对象最近被修改的时间, 用自 1/1/1970 GMT 午夜以来的毫秒值表示。或如果时间是未知的, 返回 -1。

3) doPost

protected void **doPost**(HttpServletRequest req, HttpServletResponse resp)

throws ServletException, java.io.IOException

由服务端(经 service 方法)调用, 允许 servlet 处理 POST 请求。HTTP POST 方法允许客户端一次发送一个长度不限的数据到 Web 服务端, 当提交如信用卡号码这样的信息时是有用的。覆盖这个方法时, 读取请求数据, 写响应头部, 获得响应的 writer 或输出流对象, 最后, 写响应数据, 最好包括内容类型和编码。使用一个 PrintWriter 对象返回响应时, 在访问该 PrintWriter 对象前设置内容类型。

提交响应前, servlet 容器必须写头部, 因为在 HTTP 中, 头部必须在响应体前被发送。在可能的地方, 设置内容长度(Content-Length)(使用 ServletResponse.setContentLength(int) 方法), 允许 servlet 容器使用一个持久连接返回对客户端的响应, 以提高性能。如果全部响应都装入了响应缓冲中, 那么内容长度可被自动设置。

使用 HTTP 1.1 分块译码(意思是响应有一个转移译码头部), 不要设置内容长度头部。

这个方法既不需要安全也不需要等幂。通过 POST 提交的请求操作可能有由用户控制的副作用, 例如, 更新存储的数据或在线购物。

如果 HTTP POST 请求格式不正确, doPost 返回一个 HTTP“Bad Request”信息。

参数

req: 一个 HttpServletRequest 对象, 包含客户端已产生的对 servlet 的请求。



resp: 一个 `HttpServletResponse` 对象, 包含 servlet 发送给客户端的响应。

抛出

`java.io.IOException`——当 servlet 处理请求时, 如果检测到输入或输出错误。

`ServletException`——如果 POST 请求不能被处理。

参照

`ServletOutputStream`, `ServletResponse.setContentType(java.lang.String)`

#### 4) doPut

protected void **doPut**(`HttpServletRequest req`, `HttpServletResponse resp`)

throws `ServletException`, `java.io.IOException`

由服务端(经 `service` 方法)调用, 允许 servlet 处理 PUT 请求。PUT 操作允许客户端在服务端放置一个文件, 这与使用 FTP 发送一个文件相类似。

覆盖这个方法时, 请求头部保留完整的内容头部(包括 `Content-Length`, `Content-Type`, `Content-Transfer-Encoding`, `Content-Encoding`, `Content-Base`, `Content-Language`, `Content-Location`, `Content-MD5`, and `Content-Range`)。如果你的方法不能处理一个内容头部, 那么它必须给出一个错误信息(HTTP 501——没有实现)并且丢弃该请求。更多有关 HTTP 1.1 的信息, 请参考 RFC2068。

这个方法既不需要安全也不需要等幂。DoPut 执行的操作可有由用户控制的副作用, 使用这个方法在临时存储区保存受影响 URL 的一个拷贝是有用的。

如果 HTTP POST 请求格式不正确, doPost 返回一个 HTTP“Bad Request”信息。

参数

req: 一个 `HttpServletRequest` 对象, 包含客户端已产生的对 servlet 的请求。

resp: 一个 `HttpServletResponse` 对象, 包含 servlet 发送给客户端的响应。

抛出

`java.io.IOException`——当 servlet 处理 PUT 请求时, 如果检测到输入或输出错误。

`ServletException`——如果 PUT 请求不能被处理。

#### 5) doDelete

protected void **doDelete**(`HttpServletRequest req`, `HttpServletResponse resp`)

throws `ServletException`, `java.io.IOException`

由服务端(经 `service` 方法)调用, 允许 servlet 处理 DELETE 请求。DELETE 操作允许客户端删除服务端的文件或 Web 页面。

这个方法既不需要安全也不需要等幂。通过 DELETE 提交的请求操作可有由用户控制的副作用, 使用这个方法从临时存储区删除受影响 URL 的一个拷贝是有用的。

如果 HTTP POST 请求格式不正确, doPost 返回一个 HTTP“Bad Request”信息。

参数

req: 一个 `HttpServletRequest` 对象, 包含客户端已产生的对 servlet 的请求。

resp: 一个 `HttpServletResponse` 对象, 包含 servlet 发送给客户端的响应。

抛出

`java.io.IOException`——当 servlet 处理 DELETE 请求时, 如果检测到输入或输出错误。

ServletException——如果 DELETE 请求不能被处理。

#### 6) doOptions

```
protected void doOptions(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, java.io.IOException
```

由服务端(经 service 方法)调用,允许 servlet 处理 OPTIONS 请求。OPTIONS 请求确定服务端支持那些 HTTP 方法并且返回一个适当的头部。例如,如果 servlet 覆盖 doGet,这个方法返回下述头部:

GET, HEAD, TRACE, OPTIONS。

不需要覆盖这个方法,除非 servlet 实现 HTTP 1.1 已实现的那些方法之外的新 HTTP 方法。

#### 参数

req: 一个 HttpServletRequest 对象,包含客户端已产生的对 servlet 的请求。

resp: 一个 HttpServletResponse 对象,包含 servlet 发送给客户端的响应。

#### 抛出

java.io.IOException——servlet 处理 OPTIONS 请求时,如果检测到输入或输出错误。

ServletException——如果 OPTIONS 请求不能被处理。

#### 7) doTrace

```
protected void doTrace(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, java.io.IOException
```

由服务端(经 service 方法)调用,允许 servlet 处理 TRACE 请求。TRACE 将 TRACE 请求发送的头部返回给客户端,因此,它们可用于调试。不需要(没有必要)覆盖这个方法。

#### 参数

req: 一个 HttpServletRequest 对象,包含客户端已产生的对 servlet 的请求。

resp: 一个 HttpServletResponse 对象,包含 servlet 发送给客户端的响应。

#### 抛出

java.io.IOException——当 servlet 处理 TRACE 请求时,如果检测到输入或输出错误。

ServletException——如果 TRACE 请求不能被处理。

#### 8) service

```
protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, java.io.IOException
```

从 public service 方法接收标准 HTTP 请求并且把它们分配给这个类中定义的 doXXX 方法。这个方法是 Servlet.service(javax.servlet.ServletRequest, javax.servlet.ServletResponse)方法的一个特殊的 HTTP 版本。不需要覆盖这个方法。

#### 参数

req: 一个 HttpServletRequest 对象,包含客户端已产生的对 servlet 的请求。

resp: 一个 HttpServletResponse 对象,包含 servlet 发送给客户端的响应。

抛出

java.io.IOException——当 servlet 处理 TRACE 请求时,如果检测到输入或输出错误。

ServletException——如果 TRACE 请求不能被处理。

参照

Servlet.service(javax.servlet.ServletRequest, javax.servlet.ServletResponse)

9) service

public void **service**(ServletRequest req, ServletResponse res)

throws ServletException, java.io.IOException

把客户端的请求分配给 protected service 方法。

覆盖

class GenericServlet 中的 service 方法。

参数

req: 一个 HttpServletRequest 对象,包含客户端已产生的对 servlet 的请求。

resp: 一个 HttpServletResponse 对象,包含 servlet 发送给客户端的响应。

抛出

java.io.IOException——当 servlet 处理 TRACE 请求时,如果检测到输入或输出错误。

ServletException——如果 TRACE 请求不能被处理。

参照

Servlet.service(javax.servlet.ServletRequest, javax.servlet.ServletResponse)

### 3.3 javax.servlet.http Interface HttpServletRequest

#### 1. 界面层次

javax.servlet.ServletRequest

| 继承

+—— javax.servlet.http.HttpServletRequest

#### 2. 说明

package javax.servlet.http;

public interface HttpServletRequest

extends javax.servlet.ServletRequest

扩展自 javax.servlet.ServletRequest 界面,提供请求信息给 HTTP servlets。

servlet 容器创建一个 HttpServletRequest 对象并且把它当做一个参数传递给 servlet 的方法 service(doGet, doPost, etc)。

## 3. 方法一览

java.lang.String	getAuthType()	返回用于保护当前 servlet 的身份验证方案的名字, 或者, 如果 servlet 未受到保护, 返回空(null)。
java.lang.String	getContextPath()	返回 request URI 的一部分, 指明当前 request 上下文。
Cookie[]	getCookies()	返回一个数组, 包含与当前 request 一起由客户端发送的所有 Cookie 对象。
long	getDateHeader(java.lang.String name)	以 long 形式返回当前 request 中指定头部的值。
java.lang.String	getHeader(java.lang.String name)	以 String 对象形式返回当前 request 中指定头部的值。
java.util.Enumeration	getHeaderNames()	返回一个当前 request 包含的所有头部名字的枚举。
java.util.Enumeration	getHeaders()	以 String 对象的枚举形式返回当前 request 中指定头部的所有值。
int	getIntHeader(java.lang.String name)	以 int 形式返回当前 request 中指定头部的值。
java.lang.String	getMethod()	返回与当前请求一起产生的 HTTP 方法的名字。
java.lang.String	getPathInfo()	返回任何与客户端产生当前 request 时发送的 URL 相关的额外路径信息。
java.lang.String	getPathTranslated()	返回任何在 servlet 名之后但在查询字符串之前的额外路径名, 并把它翻译成真实路径。
java.lang.String	getQueryString()	返回当前 request URL 中位于路径之后的查询字符串。
java.lang.String	getRemoteUser()	如果产生当前 request 的用户已经过身份验证, 返回用户的注册名, 否则返回 null。
java.lang.String	getRequestedSessionId()	返回客户端指定的 session ID。

续表

java.lang.String	getRequestURI()	返回 HTTP request 中当前 request 的 URL 从协议名到查询字符串的那部分。
java.lang.String	getServletPath()	返回当前 request 的 URL 中调用某个 servlet 的那部分。
HttpSession	getSession()	返回与当前 request 相联系的当前 session, 或者如果当前 request 没有 session, 那么新创建一个。
HttpSession	getSession(boolean create)	返回与当前 request 相联系的当前 session, 或者如果没有当前 session 并且参数 create 为“true”, 返回一个新的 session。
java.security.Principal	getUserPrincipal()	返回一个包含当前身份验证用户名字的 java.security.Principal 对象。
boolean	isRequestedSessionIdFromCookie()	检查被请求的 session ID 是否以一个 cookie 到达。
boolean	isRequestedSessionIdFromUrl()	不赞成使用。
Boolean	isRequestedSessionIdFromURL()	检查被请求的 session ID 是否以当前 request URL 的部分到达。
boolean	isRequestedSessionIdValid()	检查被请求的 session ID 是否仍然有效。
boolean	isUserInRole(java.lang.String role)	返回一个 boolean, 指明身份验证用户是否包含在指定的逻辑“角色”中。

#### 4. 方法详解

##### 1) getAuthType

public java.lang.String **getAuthType()**

返回用于保护当前 servlet 的身份验证方案的名字。例如:“BASIC”或“SSL”。如果 servlet 未受到保护,返回空(null)。

与 CGI 变量 AUTH\_TYPE 的值相同。

返回

指定身份验证方案名字的一个字符串,或如果请求不能被验证,返回 null。

##### 2) getCookies

public Cookie[] **getCookies()**

返回一个数组,包含与当前 request 一起由客户端发送的所有 Cookie 对象。如果没

有 Cookies 被发送,返回 null。

返回

一个包含当前 request 中所有 Cookies 的数组,如果当前 request 中没有 Cookies,为 null。

### 3) getDateHeader

public long **getDateHeader**(java.lang.String name)

以 long 形式返回当前 request 中指定头部的值,指定头部表示一个日期对象。包含日期的头部可使用这个方法,例如 If - Modified - Since。

如果当前 request 中没有这个指定名字的头部(header)方法返回 -1。如果这个头部不能转换成一个日期,方法抛出一个 IllegalArgumentException。

参数

name: 一个指定头部名字的字符串。

返回

表示头部中指定日期的一个 long 值,表达形式为自 1 / 1 / 1970 GMT 以来毫秒值,或者如果指定名字的头部未包含在当前 request 中,为 -1。

抛出

java.lang.IllegalArgumentException——如果头部值不能转换成日期。

### 4) getHeader

public java.lang.String **getHeader**(java.lang.String name)

以 String 对象形式返回当前 request 中指定头部的值。如果当前 request 不包含指定名字的头部,该方法返回 null。

参数

name: 一个指定头部名字的字符串。

返回

一个包含被请求头部值的字符串,或如果当前 request 中没有哪个名字的头部,为 null。

### 5) getHeaders

public java.util.Enumeration **getHeaders**(java.lang.String name)

以 String 对象的枚举形式返回当前 request 中指定头部的所有值。

有些头部,例如 Accept - Language,宁愿以几个头部,每个头部有不同的值,而不愿以一个头部,每个值由逗号分隔的形式被客户端发送。

如果当前 request 不包含任何指定名字的头部,该方法返回一个空的枚举变量。

参数

name: 一个指定头部名字的字符串。

返回

一个包含被请求头部值的枚举变量,或如果当前 request 没有那个名字的任何头部,为 null。

### 6) getHeaderNames

public java.util.Enumeration **getHeaderNames**()

返回一个当前 request 包含的所有头部名字的枚举。如果当前 request 没有头部,该方法返回一个空的枚举变量。

有些 servlet 容器不允许 servlets 使用这个方法访问头部,这种情况,该方法返回 null。

返回

一个与当前 request 一起发送的所有头部名字的枚举变量。如果当前 request 没有头部,返回一个空的枚举变量,如果 servlet 容器不允许使用这个方法,返回 null。

#### 7) getIntHeader

```
public int getIntHeader(java.lang.String name)
```

以 int 形式返回当前 request 中指定头部的值。如果当前 request 不包含指定名字的头部,该方法返回 -1。如果指定头部不能转换成一个整数,该方法抛出 NumberFormatException。

参数

name: 一个指定请求头部名字的字符串。

返回

一个表示指定请求头部值的整数,或如果当前 request 没有这个名字的头部,返回 -1。

抛出

java.lang.NumberFormatException——如果指定头部值不能转换成一个 int。

#### 8) getMethod

```
public java.lang.String getMethod()
```

返回与当前请求一起产生的 HTTP 方法的名字。例如,GET,POST 或 PUT。与 CGI 变量 REQUEST\_METHOD 的值相同。

返回

一个指定与当前请求一起产生的 HTTP 方法的名字的字符串。

#### 9) getPathInfo

```
public java.lang.String getPathInfo()
```

返回任何与客户端产生当前 request 时发送的 URL 相关的额外路径信息。额外路径信息跟在 servlet 路径之后,但在查询字符串之前。如果没有额外路径信息,这个方法返回 null。

与 CGI 变量 PATH\_INFO 的值相同。

返回

一个字符串,指定当前 request URL 中紧接在 servlet 路径之后,但在查询字符串之前的额外路径信息;如果当前 request URL 没有任何额外路径信息,返回 null。

#### 10) getPathTranslated

```
public java.lang.String getPathTranslated()
```

返回任何在 servlet 名之后但在查询字符串之前的额外路径名,并把它翻译成真实路径。

与 CGI 变量 PATH\_TRANSLATED 相同。

如果当前 request URL 没有任何额外路径信息,该方法返回 null。

返回

一个指定真实路径的字符串,或如果当前 request URL 没有任何额外路径信息,返回 null。

#### 11) getContextPath

```
public java.lang.String getContextPath()
```

返回 request URI 的一部分,其指明当前 request 上下文。一个 request URI 总是以上下文路径开始。路径以一个“/”开始但不能以一个“/”结束。在默认(root)上下文中,对 servlets 来说,这个方法返回“”(空字符串)。

返回

一个字符串,指定 request URI 的一部分——指明当前 request 上下文。

#### 12) getQueryString

```
public java.lang.String getQueryString()
```

返回当前 request URL 中位于路径之后的查询字符串。如果当前 request URL 中没有查询字符串,该方法返回 null。

与 CGI 变量 QUERY\_STRING 的值相同。

返回

一个字符串,其包含查询字符串。如果当前 request URL 中不包含查询字符串,返回 null。

#### 13) getRemoteUser

```
public java.lang.String getRemoteUser()
```

如果产生当前 request 的用户已经过身份验证,返回用户的注册名,否则返回 null。后继的 request 是否发送用户名字取决于浏览器和身份验证类型。

与 CGI 变量 REMOTE\_USER 相同。

返回

一个指定当前 request 产生的用户注册名的字符串,或返回 null。

#### 14) isUserInRole

```
public boolean isUserInRole(java.lang.String role)
```

返回一个 boolean,指明身份验证用户是否包含在指定的逻辑“角色”中。角色和角色成员可使用调度描述符定义。如果用户还没有被验证,该方法返回“false”。

参数

role: 一个指定角色名字的字符串。

返回

一个指明身份验证用户是否包含在指定的逻辑“角色”中的 boolean 变量;如果用户还没有被验证,返回“false”。

#### 15) getUserPrincipal

```
public java.security.Principal getUserPrincipal()
```

返回一个包含当前身份验证用户名字的 java.security.Principal 对象。如果用户还没有被验证,该方法返回“false”。

返回



一个包含当前 request 产生的用户名字的 java.security.Principal 对象。如果用户还没有被验证,返回“null”。

#### 16) getRequestedSessionId

```
public java.lang.String getRequestedSessionId()
```

返回客户端指定的 sessionId。这可能与实际使用的 session 的 ID 不一样。例如,如果 request 指定一个旧的(已终止的)session ID 并且服务端已经开始一个新的 session,这个方法获得一个新 session,其拥有新的 ID。如果 request 不指定 session ID,该方法返回 null。

返回

一个指定 session ID 的字符串,如果 request 不指定 session ID,返回 null。

#### 17) getRequestURI

```
public java.lang.String getRequestURI()
```

返回 HTTP request 第一行中当前 request 的 URL 从协议名到查询字符串的那部分。

例如:

HTTP request 的第一行	返回值
POST/ some/ path.html HTTP/ 1.1	/ some/ path.html
GET http: // foo.bar/ a.html HTTP/ 1.0	http: // foo.bar/ a.html
HEAD / xyz? a = b HTTP/ 1.1	/ xyz

因为某种意图或主机原因而重构一个 URL,使用 use

HttpUtils.getRequestURL(javax.servlet.http.HttpServletRequest)。

返回

一个字符串,包含 URL 从协议名到查询字符串的那部分。

#### 18) getServletPath

```
public java.lang.String getServletPath()
```

返回当前 request 的 URL 中调用某个 servlet 的那部分。这或者包含 servlet 的名字,或者包含 servlet 的路径,但是不包含任何额外路径信息或查询字符串。

与 CGI 变量 SCRIPT\_NAME 相同。

返回

一个包含被调用 servlet 的名字或路径的字符串。

#### 19) getSession

```
public HttpSession getSession(boolean create)
```

返回与当前 request 相联系的当前 session,或者如果没有当前 session 并且参数 create 为“true”,返回一个新的 session。

如果 create 为“false”并且当前 request 没有有效的 HttpSession,这个方法返回 null。

为确认 session 被适当的维持着,在 response 提交前,必须调用这个方法。

参数

create:true,如果需要,为当前 request 创建一个新的 session:false 如果没有当前 session,返回 null。

返回

与当前 request 相联系的当前 HttpSession, 或者如果 create 为“false”并且当前 request 没有有效的 session, 返回 null。

20) getSession

public HttpSession getSession()

返回与当前 request 相联系的当前 session, 或者如果当前 request 没有 session, 那么新创建一个。

返回

与当前 request 相联系的 HttpSession。

21) isRequestedSessionIdValid

public boolean isRequestedSessionIdValid()

检查被请求的 session ID 是否仍然有效。

返回

如果在当前 session 上下文中当前 request 有一个有效 session 的 ID, 返回“true”, 否则, 返回“false”。

22) isRequestedSessionIdFromCookie

public boolean isRequestedSessionIdFromCookie()

检查被请求的 session ID 是否以一个 Cookie 到达。

返回

如果 session ID 以一个 Cookie 到达, 返回“true”; 否则, 返回“false”。

23) isRequestedSessionIdFromURL

public boolean isRequestedSessionIdFromURL()

检查被请求的 session ID 是否以当前 request URL 的部分到达。

返回

如果 session ID 以 URL 的一部分到达, 返回“true”; 否则, 返回“false”。

24) isRequestedSessionIdFromUrl

public boolean isRequestedSessionIdFromUrl()

不赞成使用。Java Servlet API 2.1 中, 使用 isRequestedSessionIdFromURL() 代替。

## 3.4 javax.servlet.http Interface HttpServletResponse

### 1. 界面层次

javax.servlet.ServletResponse

| 继承

+—— javax.servlet.http.HttpServletResponse

### 2. 说明

package javax.servlet.http;

```
public interface HttpServletResponse
extends javax.servlet.ServletResponse
```

### 3. 扩展自 **ServletResponse** 界面

在发送响应方面,提供 HTTP 特殊功能。例如:它拥有访问 HTTP 头部和 cookies 的方法。

servlet 容器创建一个 **HttpServletResponse** 对象并且把它当做一个参数传递给 servlet 的方法 `service(doGet,doPost,etc)`。

### 4. 域一览

static int	SC_ACCEPTED	状态码(202)指明请求已被接受,但还未被处理完。
static int	SC_BAD_GATEWAY	状态码(502)指明 HTTP 服务端从代理或网关接收到一个无效的响应。
static int	SC_BAD_REQUEST	状态码(400)指明客户端发送的请求有语法错误。
static int	SC_CONFLICT	状态码(409)指明因为与某个资源的当前状态冲突,所以请求不能被完成。
static int	SC_CONTINUE	状态码(100)指明客户端可以继续。
static int	SC_CREATED	状态码(201)指明请求成功并在服务端创建了一个新的资源。
static int	SC_EXPECTATION_FAILED	状态码(417)指明服务端不能满足期望请求的头部给出的期望值。
static int	SC_FORBIDDEN	状态码(403)指明服务端理解请求,但拒绝实现它。
static int	SC_GATEWAY_TIMEOUT	状态码(504)指明服务端不能从代理或网关接收到一个及时的响应。
static int	SC_GONE	状态码(410)指明服务端的资源不再有效并且前驱(转发)地址未知。
static int	SC_HTTP_VERSION_NOT_SUPPORTED	状态码(505)指明服务端不支持或拒绝支持请求信息使用的 HTTP 协议版本。

续表

static int	SC_INTERNAL_SERVER_ERROR	状态码(500)指明 HTTP 服务端发生了内部错误,请求不能被实现。
static int	SC_LENGTH_REQUIRED	状态码(411)指明请求因为没有确定的内容长度(Content Length),不能被处理。
static int	SC_METHOD_NOT_ALLOWED	状态码(405)指明请求 URI 指定资源不支持请求行(Request Line)指定方法。
static int	SC_MOVED_PERMANENTLY	状态码(301)指明资源已经被永久的移动到一个新的位置,并且以后的引用应该使用新的 URI。
static int	SC_MOVED_TEMPORARILY	状态码(302)指明资源被临时的移动到其它位置,但是以后的引用仍使用原来的 URI。
static int	SC_MULTIPLE_CHOICES	状态码(300)指明被请求的资源符合表达式集合中的任意表达式,每个都有其特殊的位置。
static int	SC_NO_CONTENT	状态码(204)指明请求成功,但是没有新的信息返回。
static int	SC_NON_AUTHORITATIVE_INFORMATION	状态码(203)指明客户端给出的元信息不是发源于服务端的。
static int	SC_NOT_ACCEPTABLE	状态码(406)指明由请求确定的资源产生的响应实体,依照请求头部的要求,其内容特性是不可接受的。
static int	SC_NOT_FOUND	状态码(400)指明被请求的资源是无效的。
static int	SC_NOT_IMPLEMENTED	状态码(501)指明 HTTP 服务端不支持实现请求所必须的功能。
static int	SC_NOT_MODIFIED	状态码(304)指明一个条件 GET 操作发现资源是有效的,但是也是不可修改的。
static int	SC_OK	状态码(200)指明请求正常并取得成功。
static int	SC_PARTIAL_CONTENT	状态码(206)指明服务端已实现对资源的部分 GET 请求。
static int	SC_PAYMENT_REQUIRED	状态码(402)保留以后使用。

续表

static int	SC _ PRECONDITION _ FAILED	状态码(412)指明服务端对请求头部的一个或更多的域(字段)给出的前提条件进行测试,结果是前提条件错误。
static int	SC _ PROXY _ AUTHENTICATION _ REQUIRED	状态码(407)指明客户端必须首先通过代理对其进行的身份验证。
static int	SC _ REQUEST _ ENTITY _ TOO _ LARGE	状态码(413)指明服务端拒绝处理某个请求,因为该请求实体比服务端愿意或能处理的大。
static int	SC _ REQUEST _ TIMEOUT	状态码(408)指明在服务端有意等待的时间之内,客户端不能产生请求。既产生的请求因为得不到及时处理而失效。
static int	SC _ REQUEST _ URI _ TOO _ LONG	状态码(414)指明服务端拒绝为请求提供服务,因为请求 URI 比服务端愿意解释的长。
static int	SC _ REQUESTED _ RANGE _ NOT _ SATISFI- ABLE	状态码(416)指明服务端不能提供服务给被请求的字节范围。
static int	SC _ RESET _ CONTENT	状态码(205)指明代理应当重置那些导致请求发出的文档视图。
static int	SC _ SEE _ OTHER	状态码(303)指明对某个请求的响应可在另一个 URI 中找到。
static int	SC _ SERVICE _ UNAVAILABLE	状态码(503)指明 HTTP 服务端临时超载,不能处理请求。
static int	SC _ SWITCHING _ PROTOCOLS	状态码(101)指明服务端正根据后继头部转换协议。
static int	SC _ UNAUTHORIZED	状态码(401)指明请求需要 HTTP 验证。
static int	SC _ UNSUPPORTED _ MEDIA _ TYPE	状态码(415)指明服务端拒绝为请求提供服务,因为对被请求的方法来说,被请求的资源不支持请求实体的格式。
static int	SC _ USE _ PROXY	状态码(305)指明被请求的资源必须通过 Location 域(字段)给出的代理来访问。

## 5. 方法一览

void	addCookie(Cookie Cookie)	将指定的 Cookie 加入响应。
void	addDateHeader(java.lang.String name, long date)	将给出的名字(name)和日期值(data _ value)加入响应的一个头部。

续表

void	<code>addHeader(java.lang.String name, java.lang.String value)</code>	将给出的名字 (name) 和值 (value) 加入响应的一个头部。
void	<code>addIntHeader(java.lang.String name, int value)</code>	将给出的名字 (name) 和整型值 (integer value) 加入响应的一个头部。
boolean	<code>containsHeader(java.lang.String name)</code>	返回一个 boolean 变量指明参数 name 指定的响应头部是否已被设置。
java.lang.String	<code>encodeRedirectUrl(java.lang.String url)</code>	不赞成使用。2.1 版本中, 使用 <code>encodeRedirectURL (String url)</code> 代替。
java.lang.String	<code>encodeRedirectURL(java.lang.String url)</code>	编码 <code>sendRedirect</code> 方法中使用的指定 URL, 或如果不需编码, 返回未改变的 URL。
java.lang.String	<code>encodeUrl(java.lang.String url)</code>	不赞成使用。2.1 版本中, 使用 <code>encodeURL (String url)</code> 代替。
java.lang.String	<code>encodeURL(java.lang.String url)</code>	编码指定的 URL, 包括其中的 session ID 部分, 或如果不需编码, 返回未改变的 URL。
void	<code>sendError(int sc)</code>	使用指定的状态码发送一个错误响应到客户端。
void	<code>sendError(int sc, java.lang.String msg)</code>	使用指定的状态码和描述信息发送一个错误响应到客户端。
void	<code>sendRedirect(java.lang.String location)</code>	使用指定的重定向位置 URL 发送一个临时的重定向响应到客户端。
void	<code>setDateHeader(java.lang.String name, long date)</code>	以给出的名字 (name) 和日期值 (date _ value) 设置响应的一个头部。
void	<code>setHeader(java.lang.String name, java.lang.String value)</code>	以给出的名字 (name) 和值 (value) 设置响应的一个头部。
void	<code>setIntHeader (java.lang.String name, int value)</code>	以给出的名字 (name) 和整型值 (integer value) 设置响应的一个头部。
void	<code>setStatus(int sc)</code>	给当前响应设置状态码。
void	<code>setStatus(int sc, java.lang.String sm)</code>	不赞成使用。给当前响应设置状态码和信息。

## 6. 方法详解

### 1) addCookie

```
public void addCookie(Cookie Cookie)
```

将指定的 Cookie 加入响应。这个方法可被多次调用以设置多个 Cookie。

参数

Cookie——返回到客户端的 Cookie。

2) containsHeader

public boolean **containsHeader**(java.lang.String name)

返回一个 boolean 变量指明参数 name 指定的响应头部是否已被设置。

参数

name——头部的名字。

返回

如果参数 name 指定的响应头部已被设置,返回“true”,否则返回“false”。

3) encodeURL

public java.lang.String **encodeURL**(java.lang.String url)

编码指定的 URL,包括其中的 session ID 部分,或如果不需编码,返回未改变的 URL。这个方法的实现包括决定 URL 中的 session ID 是否需要编码的逻辑。例如,如果浏览器支持 cookies,或 session 跟踪被关闭,则不需要 URL 编码。

为健全(健壮、完善)session 跟踪,由一个 servlet 发出的所有 URL 都应当能通过这个方法运行。另外,不支持 cookies 的浏览器不能使用 URL 重写。

参数

url——将被编码的 url。

返回

如果需要编码,返回经过编码的 URL,否则返回未改变的 URL。

4) encodeRedirectURL

public java.lang.String **encodeRedirectURL**(java.lang.String url)

编码 sendRedirect 方法中使用指定的 URL,或如果不需编码,返回未改变的 URL。这个方法的实现包括决定 URL 中的 session ID 是否需要编码的逻辑。因为这种决定的规则与决定是否编码一个标准链接下使用的规则是不一样的,所以这个方法与 encodeURL 方法是分离的。发送给 HttpServletResponse.sendRedirect 方法的所有 URL 都应当能通过这个方法运行。另外,不支持 cookies 的浏览器不能使用 URL 重写。

参数

url——将被编码的 url。

返回

如果需要编码,返回经过编码的 URL,否则返回未改变的 URL。

参照

sendRedirect(java.lang.String), encodeUrl(java.lang.String)

5) encodeUrl

public java.lang.String **encodeUrl**(java.lang.String url)

不赞成使用。2.1 版本中,使用 encodeURL(String url) 代替。

参数

url——将被编码的 url。

返回

如果需要编码,返回经过编码的 URL,否则返回未改变的 URL。

#### 6) encodeRedirectUrl

```
public java.lang.String encodeRedirectUrl(java.lang.String url)
```

不赞成使用。2.1 版本中,使用 `encodeRedirectURL(String url)` 代替。

参数

url——将被编码的 url。

返回

如果需要编码,返回经过编码的 URL,否则返回未改变的 URL。

#### 7) sendError

```
public void sendError(int sc, java.lang.String msg)
```

```
throws java.io.IOException
```

使用指定的状态码和描述信息发送一个错误响应到客户端,服务端建立的该错误响应,通常看起来像一个标准的服务端错误页面。

如果该错误响应在这个方法调用前被提交,这个方法抛出一个 `IllegalStateException`。使用这个方法之后,就应该认为该错误响应已被提交,因而不可写入。

参数

sc:错误状态码。

msg:描述信息。

抛出

`java.io.IOException`——如果一个输入或输出异常发生。

`java.lang.IllegalStateException`——如果响应在这个方法调用之前被提交。

#### 8) sendError

```
public void sendError(int sc)
```

```
throws java.io.IOException
```

使用指定的状态码和描述信息发送一个错误响应到客户端,服务端建立的该错误响应,通常看起来像一个标准的服务端错误页面。

如果该错误响应在这个方法调用前被提交,这个方法抛出一个 `IllegalStateException`。使用这个方法之后,就应该认为该错误响应已被提交,因而不可写入。

参数

sc:错误状态码

抛出

`java.io.IOException`——如果一个输入或输出异常发生。

`java.lang.IllegalStateException`——如果响应在这个方法调用之前被提交。

#### 9) sendRedirect

```
public void sendRedirect(java.lang.String location)
```

```
throws java.io.IOException
```

使用指定的重定向位置 URL 发送一个临时的重定向响应到客户端。这个方法可接受相对 URL;在发送该响应到客户端之前,servlet 容器(container)将把相对 URL 转换成



绝对 URL。

如果该响应在这个方法调用之前被提交,这个方法抛出一个 `IllegalStateException`。使用这个方法之后,就应该认为该错误响应已被提交,因而不可写入。

参数

location: 重定向位置 URL。

抛出

`java.io.IOException`——如果一个输入或输出异常发生。

`java.lang.IllegalStateException`——如果响应在这个方法调用前被提交。

10) `setDateHeader`

`public void setDateHeader(java.lang.String name, long date)`

以给出的名字(name)和日期值(data \_ value)设置响应的一个头部。这个日期由自新纪元始的毫秒值指定。如果头部已被设置,新值覆盖先前的值。设置头部的值之前,可用 `containsHeader` 方法检测该头部是否存在。

参数

name: 欲设置头部的名字。

value: 赋予的日期值。

参照:

`containsHeader(java.lang.String)`, `addDateHeader(java.lang.String, long)`

11) `addDateHeader`

`public void addDateHeader(java.lang.String name, long date)`

将给出的名字(name)和日期值(data \_ value)加入响应的一个头部。这个日期由自新纪元始的毫秒值指定。这个方法允许响应头部有多个值。

参数

name: 欲设置头部的名字。

value: 额外(另外)的日期值。

参照

`setDateHeader(java.lang.String, long)`

12) `setHeader`

`public void setHeader(java.lang.String name, java.lang.String value)`

以给出的名字(name)和值(value)设置响应的一个头部。如果头部已被设置,新值覆盖先前的值。设置头部的值之前,可用 `containsHeader` 方法检测该头部是否存在。

参数

name: 欲设置头部的名字。

value: 头部的值。

参照

`containsHeader(java.lang.String)`, `addHeader(java.lang.String, java.lang.String)`

13) `addHeader`

`public void addHeader(java.lang.String name, java.lang.String value)`

将给出的名字(name)和日期值(value)加入响应的一个头部。这个方法允许响应头

部有多个值。

参数

name: 欲设置头部的名字。

value: 额外(另外)的日期值。

参照

setHeader(java.lang.String, java.lang.String)

14) setIntHeader

public void **setIntHeader**(java.lang.String name, int value)

以给出的名字(name)和整数值(integer value)设置响应的一个头部。如果头部已被设置,新值覆盖先前的值。设置头部的值之前,可用 containsHeader 方法检测该头部是否存在。

参数

name: 欲设置头部的名字。

value: 赋予的整数值。

参照

containsHeader(java.lang.String), addIntHeader(java.lang.String, int)

15) addIntHeader

public void **addIntHeader**(java.lang.String name, int value)

将给出的名字(name)和整数值(integer value)加入响应的一个头部。这个方法允许响应头部有多个值。

参数

name: 欲设置头部的名字。

value: 赋予的整数值。

参照

setIntHeader(java.lang.String, int)

16) setStatus

public void **setStatus**(int sc)

给当前响应设置状态码。当没有错误的时候(例如,对状态码 SC\_OK 或者 SC\_MOVED\_TEMPORARILY),这个方法用来设置返回的状态码。如果有错误,用 sendError 方法代替。

参数

sc: 状态码。

参照

sendError(int, java.lang.String)

17) setStatus

public void **setStatus**(int sc, java.lang.String sm)

不赞成使用,因为信息参数的意义不明确。设置状态码用 setStatus(int)方法,发送有描述信息的错误使用 sendError(int, String)。给当前响应设置状态码和信息。

参数

sc:状态码。

sm:状态信息。

## 3.5 javax.servlet.http Interface HttpSession

### 1. 说明

public interface HttpSession

提供一种方法,从页面的多个请求或访问到整个 Web 站点这样广的范围内识别一个用户,并且存储这个用户的有关信息。

Servlet 容器使用该界面在 HTTP 客户端和 HTTP 服务端之间创建一个 session。Session 维持指定长时间,可以跨越多个连接或用户页面请求。一个 session 通常只与一个用户通信,该用户可能多次访问站点。服务端有多种方法维持 session,如 cookies 或 URLs 重写。

这个界面允许 servlet 查看和操纵 session 的有关信息,如 session 识别器在创建时,最近访问时间。

绑定对象到 session,允许用户信息维持在多个用户连接之间。

当应用程序存储一个对象到 session 或从 session 中删除一个对象,该 session 检查对象是否实现了 HttpSessionBindingListener。如果实现了,servlet 通知对象,它已经被绑定到该 session 或者已与该 session 解除绑定。

Servlet 应能处理客户端禁止使用 session(选择不加入 session)的情形,如有意将 cookies 关闭。直到客户端加入 session,isNew 返回 true。如果客户端选择不加入 session,对每个请求,getSession 将返回不同的 session,并且 isNew 将总是返回 true。

Session 信息(可理解为变量)的作用域是当前 web 应用程序(ServletContext),所以存储在一个 context(上下文)中的信息不能被另一个 context 中的信息直接所见。

参照

HttpSessionBindingListener, HttpSessionContext

### 2. 方法一览

java.lang.Object	getAttribute(java.lang.String name)	返回绑定在当前 session 中指定名字之下的对象。或如果没有对象绑定在指定名字之下,返回 null。
java.util.Enumeration	getAttributeNames()	返回一个 String 对象的枚举变量,包含绑定在当前 session 中的所有对象的名字。

续表

long	getCreationTime()	返回当前 session 被创建时的时间,以自 1/1/1970 GMT 午夜以来的毫秒值表示。
java.lang.String	getId()	返回一个字符串,包含分配给当前 session 的唯一标识符。
long	getLastAccessedTime()	返回最近一次客户端发送与当前 session 相关联的请求的时间,以自 1/1/1970 GMT 午夜以来的毫秒值表示。
int	getMaxInactiveInterval()	返回以秒表示的当前 session 处于活动状态的最大时间间隔,即在这个时间间隔内, servlet 容器将保持当前 session 对客户端访问处于打开状态。
HttpSessionContext	getSessionContext()	不赞成使用。2.1 版本中没有代替者,将在今后的 Java Servlet API 版本中被删除。
java.lang.Object	getValue(java.lang.String name)	不赞成使用。2.2 版本中,使用 getAttribute(java.lang.String)代替。
java.lang.String[]	getValueNames()	不赞成使用。2.2 版本中,使用 getAttributeNames()代替。
void	invalidate()	使当前 session 失效并且将绑定到其中的任何对象与之解除绑定。
boolean	isNew()	如果客户端还不知道当前 session 或如果客户端选择不加入当前 session,返回 true。
void	putValue(java.lang.String name, java.lang.Object value)	不赞成使用。2.2 版本中使用 setAttribute(java.lang.String, java.lang.Object)代替。
void	removeAttribute(java.lang.String name)	从当前 session 中删除绑定到指定名下的对象。
void	removeValue(java.lang.String name)	不赞成使用。2.2 版本中,使用 removeAttribute(java.lang.String name)代替。
void	setAttribute(java.lang.String name, java.lang.Object value)	使用指定的名字绑定一个对象到当前 session。
void	setMaxInactiveInterval(int interval)	指定以秒表示的时间间隔,在这个间隔之内, servlet 使当前 session 处于打开状态,之外,由 servlet 容器使当前 session 失效。

### 3. 方法详解

#### 1) getCreationTime

`public long getCreationTime()`

返回当前 session 被创建时的时间,以自 1/1/1970 GMT 午夜以来的毫秒值表示。

返回

一个 long 类型变量,表示当前 session 被创建的时间,以自 1/1/1970 GMT 午夜以来的毫秒值度量。

抛出

`java.lang.IllegalStateException`——如果该方法调用作用在一个无效的 session 上。

#### 2) getId

`public java.lang.String getId()`

返回一个字符串,包含分配给当前 session 的唯一标识符。该标识符由 servlet 容器分配并且与实现无关。

返回

一个 string 类型变量,表示分配给该 session 的标识符。

#### 3) getLastAccessedTime

`public long getLastAccessedTime()`

返回最近一次客户端发送与当前 session 相关联的请求的时间,以自 1/1/1970 GMT 午夜以来的毫秒值表示。

注意应用程序获得或设置 session 的值,不会影响(改变)访问时间。

返回

一个 long 类型变量,表示最近一次客户端发送与当前 session 相关联的请求的时间,以自 1/1/1970 GMT 午夜以来的毫秒值表示。

#### 4) setMaxInactiveInterval

`public void setMaxInactiveInterval(int interval)`

指定以秒表示的时间间隔,在这个间隔之内,servlet 使当前 session 处于打开状态,之外,servlet 容器使当前 session 失效。负数时间指明 session 永不超时,则永远不失效。

参数

interval——一个整数,指定以秒表示的时间段。

#### 5) getMaxInactiveInterval

`public int getMaxInactiveInterval()`

返回以秒表示的当前 session 处于活动状态的最大时间间隔,既是在这个时间间隔内,servlet 容器将保持当前 session 对客户端访问处于打开状态。最大时间间隔可用 `setMaxInactiveInterval` 方法设置。负数时间指明 session 永不超时,既永远不失效。

返回

一个 integer 类型变量,表示客户端请求期间当前 session 保持打开状态的秒数。

参照

`setMaxInactiveInterval(int)`

## 6) getSessionContext

```
public HttpSessionContext getSessionContext()
```

不赞成使用。2.1 版本中没有代替者,将在今后的 Java Servlet API 版本中被删除。

## 7) getAttribute

```
public java.lang.Object getAttribute(java.lang.String name)
```

返回绑定在当前 session 中指定名字之下的对象。或如果没有对象绑定在指定名字之下,返回 null。

参数

name—— 一个字符串,指定对象的名字。

返回

有指定名字的对象。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

## 8) getValue

```
public java.lang.Object getValue(java.lang.String name)
```

不赞成使用。2.2 版本中,使用 getAttribute(java.lang.String)代替。

参数

name—— 一个字符串,指定对象的名字。

返回

有指定名字的对象。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

## 9) getAttributeNames

```
public java.util.Enumeration getAttributeNames()
```

返回一个 String 对象的枚举变量,包含绑定在当前 session 中的所有对象的名字。

返回

一个 String 对象的枚举变量,表示绑定在当前 session 中的所有对象的名字。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

## 10) getValueNames

```
public java.lang.String[] getValueNames()
```

不赞成使用。2.2 版本中,使用 getAttributeNames()代替。

返回

一个 String 对象的数组,表示绑定在当前 session 中的所有对象的名字。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

## 11) setAttribute

```
public void setAttribute(java.lang.String name, java.lang.Object value)
```

使用指定的名字绑定一个对象到当前 session。如果已有一个同名的对象绑定到当前

session, 替换掉已有同名对象。

这个方法执行后, 并且如果对象实现了 HttpSessionBindingListener, 容器将调用 HttpSessionBindingListener.valueBound。

参数

name: 对象将被绑定到这个名字之下。不能为 null。

value: 欲绑定对象。不能为 null。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

12) putValue

public void **putValue**(java.lang.String name, java.lang.Object value)

不赞成使用。2.2 版本中使用 setAttribute(java.lang.String, java.lang.Object) 方法代替。

参数

name: 对象将被绑定到这个名字之下。不能为 null。

value: 欲绑定对象。不能为 null。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

13) removeAttribute

public void **removeAttribute**(java.lang.String name)

从当前 session 中删除以指定名绑定的对象。如果当前 session 中没有以指定名绑定的对象, 这个方法什么也不做。

这个方法执行后, 并且如果对象实现了 HttpSessionBindingListener, 容器将调用 HttpSessionBindingListener.valueUnbound。

参数

name: 从当前 session 删除对象的名字。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

14) removeValue

public void **removeValue**(java.lang.String name)

不赞成使用。2.2 版本中, 使用 removeAttribute(java.lang.String name) 代替。

参数

name: 从当前 session 删除对象的名字。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

15) invalidate

public void **invalidate**()

使当前 session 失效, 并且解除绑定到其中的任何对象的绑定。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

16) isNew

public boolean **isNew()**

如果客户端还不知道当前 session 或如果客户端选择不加入当前 session, 返回 true。例如, 如果服务端只使用基于 cookie 的 session, 并且客户端禁止使用 cookie, 那么对每个请求, session 都是新的。

返回

如果服务端已经创建了一个 session, 但是客户端还没有将该 session 加入(写入自身)。

抛出

java.lang.IllegalStateException——如果该方法调用作用在一个无效的 session 上。

## 3.6 javax.servlet.http Class HttpSessionBindingEvent

### 1. 类层次

java.lang.Object

|

+-- java.util.EventObject

|

+-- javax.servlet.http.HttpSessionBindingEvent

### 2. 说明

public class HttpSessionBindingEvent

extends java.util.EventObject

当对象被绑定到 session 或与 session 解除绑定时, 发送一个事件给该对象, 该对象实现了 HttpSessionBindingListener 界面。

session 调用 HttpSession.putValue() 方法与一个对象绑定, 调用 HttpSession.removeValue() 方法与一个对象解除绑定。

### 3. 构造器一览

HttpSessionBindingEvent(HttpSession session, java.lang.String name)

构造一个事件, 该事件通知一个对象——它已被绑定到一个 session 或与一个 session 解除绑定。

### 4. 方法一览

java.lang.String	getName()	返回被绑定到 session 或与 session 解除绑定的对象的名字。
HttpSession	getSession()	返回对象被绑定到或与之解除绑定的 session。



## 3.7 javax.servlet.http Interface HttpSessionBindingListener

### 1. 说明

public interface **HttpSessionBindingListener**

extends java.util.EventListener

当一个对象被绑定到一个 session 或与一个 session 解除绑定时,导致这个对象被通知。这个对象被 HttpSessionBindingEvent 对象通知。

### 2. 方法一览

void	valueBound(HttpSessionBindingEvent event)	通知对象它已被绑定到一个 session 并标识该 session。
void	valueUnbound(HttpSessionBindingEvent event)	通知对象它已与一个 session 解除绑定并标识该 session。

## 3.8 javax.servlet.http Interface HttpSessionContext

### 1. 说明

不赞成使用。因 Java(tm) Servlet API 2.1 的安全原因,没有替代者。这个界面将在这个 API 的今后版本中删除。

public interface HttpSessionContext

### 2. 方法一览

java.util.Enumeration	getIds()	不赞成使用。Java Servlet API 2.1 中没有替换者。这个方法必然返回一个空的枚举变量,将在今后的版本中删除。
HttpSession	getSession (java.lang.String sessionId)	不赞成使用。Java Servlet API 2.1 中没有替换者。这个方法必然返回 null,将在今后的版本中删除。

## 3.9 javax.servlet.http Class HttpUtils

### 1. 类层次

java.lang.Object

|

+-- javax.servlet.http.HttpUtils

### 2. 说明

public class **HttpUtils**

extends java.lang.Object

提供一个对书写 HTTP servlet 有用的方法的集合。

### 3. 构造器一览

HttpUtils()

构造一个空的 HttpUtils 对象。

### 4. 方法一览

static java.lang.StringBuffer	getRequestURL( HttpServletRequest req)	使用 HttpServletRequest 对象中的信息重构该请求的 URL。
static java.util.Hashtable	parsePostData(int len, ServletInputStream in)	解析客户端使用 HTTP POST 方法以 application/x-www-form-urlencoded MIME 类型发送到服务端的表单数据。
static java.util.Hashtable	parseQueryString( java.lang.String s)	解析客户端传递给服务端的查询串,并以“关键字——值”对的形式建立一个哈希表。

[ G e n e r a l   I n f o r m a t i o n ]

书名=深入JSP网络编程

作者=

页数=509

SS号=10455305

出版日期=

www.mycodes.net