

Java Web 开发

盛鑫教育产品研发部研发

www.51OK.org

声明：

- 1、 本书为内部资料，仅供盛鑫教育授权单位使用；
- 2、 本书版权归盛鑫教育所有，未经本公司书面许可，任何单位和个人不得全部或部分复制本课程资料；
- 3、 如发现本书存在印刷质量问题，盛鑫教育负责免费调换；
- 4、 希望广大读者阅读本书后提出宝贵意见和建议，并将您的意见反馈至：info@51OK.org；
- 5、 本书技术支持平台网址：<http://www.51OK.org>，欢迎访问。

目 录

理 论 部 分.....	1
第一章 web运行模式.....	3
1.1 C/S和B/S 开发模式.....	4
1.2 Java web开发模式.....	4
1.3 Tomcat Web服务器.....	6
1.4 Tomcat样例程序.....	8
1.5 Eclipse web开发插件	12
小结.....	14
课后练习.....	15
第二章 Servlet概述	17
2.1 Servlet概述	18
2.2 Servlet示例	20
2.3 service方法	25
2.4 doXXX方法	26
2.5 Servlet生命周期	30
小结.....	32
课后练习.....	33
第三章 Servlet环境	35
3.1 HttpServletRequest	36
3.2 HttpServletResponse.....	43
3.3 ServletConfig	46
3.4 ServletContext.....	49
3.5 RequestDispatcher	52
3.6 二进制内容输出.....	57
3.7 web.xml文档结构.....	60
小结.....	61
课后练习.....	63
第四章 会话管理.....	65
4.1 用户验证.....	66
4.2 会话管理的意义.....	72
4.3 会话管理的分类.....	72
4.4 Cookie管理的会话	73
4.5 URL重写.....	79

小结	80
课后练习	81
第五章 线程安全	83
5.1 Servlet的容器	84
5.2 线程安全的意义	86
5.3 本地变量	87
5.4 同步	90
5.5 借助Servlet对象保证线程安全	92
小结	98
课后练习	99
第六章 JSP（一）	101
6.1 JSP概念	103
6.2 第一个JSP	103
6.3 脚本元素	105
6.4 代码段（Scriptlet）	105
6.5 声明（Declaration）	106
6.6 表达式（Expression）	107
6.7 注释	107
6.8 Page指令	108
6.9 Include指令	112
6.10 Request对象	114
6.11 Response对象	116
6.12 out对象	117
6.13 示例	117
小结	121
课后练习	122
第七章 JSP（二）.....	123
7.1 application对象.....	124
7.2 Session对象	126
7.3 pageContext对象	131
7.4 Cookie对象.....	135
7.5 config对象	137
小结	137
课后练习	138
第八章 JSP动作	139
8.1 include动作	140
8.2 forward动作.....	141
8.3 操作javaBean的动作	143

小结.....	150
课后练习.....	151
第九章 JSP2.0 特征.....	153
9.1 JSP2.0 简介.....	154
9.2 EL表达式语言.....	154
9.3 JSP2.0 中的自定义标记.....	159
9.4 使用简单标签机制.....	159
9.5 使用标签文件.....	161
9.6 JSTL.....	161
9.7 核心标签库简介.....	162
9.8 使用核心标签库.....	162
小结.....	167
课后练习.....	168
第十章 JSTL.....	169
10.1 函数标签.....	170
10.2 SQL标签.....	173
10.3 国际化支持标签.....	178
小结.....	184
课后练习.....	186
第十一章 自定义标签(一).....	187
11.1 自定义标签简介.....	188
11.2 自结束标签.....	189
11.3 标签中的属性.....	193
11.4 TLD文件.....	199
小结.....	200
课后练习.....	201
第十二章 自定义标签(二).....	203
12.1 标签中的标记体.....	204
12.2 标签中的子标记.....	209
12.3 标签处理类中的各种返回值.....	212
小结.....	212
课后练习.....	213
第十三章 MVC实现.....	215
13.1 Web开发与MVC模式.....	216
13.2 MVC模式示例.....	218
小结.....	231
课后练习.....	232
第十四章 Ajax框架.....	233

14.1 AJAX应用示例	234
14.2 AJAX框架和DWR	240
小结	243
课后练习	244
实 验 部 分	245
第一章 web运行模式	247
1.1 模拟实验	248
1.2 实战实验	267
1.3 实验后任务	267
第二章 Servlet概述	269
2.1 模拟实验	271
2.2 实战实验	275
2.3 实验后任务	277
第三章 Servlet环境	279
3.1 模拟实验	280
3.2 实战实验	289
3.3 实验后任务	289
第四章 会话管理	291
4.1 模拟实验	292
4.2 实战实验	310
4.3 实验后任务	312
第五章 线程安全	315
5.1 模拟实验	316
5.2 实战实验	322
5.3 实验后任务	323
第六章 JSP（一）	325
6.1 模拟实验	326
6.2 实战实验	339
6.3 实验后任务	339
第七章 JSP（二）	341
7.1 模拟实验	342
7.2 实战实验	350
7.3 实验后任务	350
第八章 JSP动作	353
8.1 模拟实验	354
8.2 实战实验	358
8.3 实验后任务	359
第九章 JSP2.0 特征	361

9.1 模拟实验.....	362
9.2 实战实验.....	365
9.3 实验后任务.....	365
第十章 JSTL.....	367
10.1 模拟实验.....	368
10.2 实战实验.....	374
10.3 实验后任务.....	374
第十一章 自定义标签(一).....	377
11.1 模拟实验.....	378
11.2 实战实验.....	383
第十二章 自定义标签(二).....	385
12.1 模拟实验.....	386
12.2 实战实验.....	390
第十三章 MVC实现.....	393
13.1 模拟实验.....	394
13.2 实战实验.....	400
13.3 实验后任务.....	402
第十四章 Ajax框架.....	403
14.1 模拟实验.....	404
14.2 实战实验.....	406
14.3 实验后任务.....	408
附录部分.....	409
附录一 练习答案.....	411
第一章.....	411
第二章.....	411
第三章.....	411
第四章.....	412
第五章.....	413
第六章.....	413
第七章.....	414
第八章.....	415
第九章.....	415
第十章.....	416
第十一章.....	416
第十二章.....	417
第十三章.....	417
第十四章.....	418

理 论 部 分

第一章 web 运行模式

概要

随着计算机软硬件的持续高速发展，以及计算机网络的普及。使得软件应用从以往的单机软件扩展到了基于网络的软件，并随之产生了基于 Internet 网络的 web 应用程序。Java 作为业内重要的软件开发语言，也提供了 web 的开发机制。

目标

- C/S 和 B/S 开发模式(理解)
- MVC 开发模式(理解)
- Tomcat web 服务器(掌握)
- Eclipse web 开发(掌握)

目录

- 1.1 C/S 和 B/S 开发模式
- 1.2 Java web 开发模式
- 1.3 Tomcat Web 服务器
- 1.4 Tomcat 样例程序
- 1.5 Eclipse web 开发插件

1.1 C/S 和 B/S 开发模式

在前面的课程中，我们已经学过了 java 这门面向对象的语言，以及面向对象的特征。还学会了一种重要的开发模式：**C/S**（客户机/服务器模式）。在这一模式下，一台计算机可以存储数据，并且提供操纵这些数据的访问接口，还可以包含部分的业务逻辑。这台计算机称之为服务器，专门接收其它计算机向它发出的数据请求，并对请求作相应的处理，这台计算机就是服务器；另一台计算机提供与用户进行交互的软件界面，并对用户录入的数据进行验证，包含用户所需要的业务逻辑，并根据制定的业务逻辑向服务器发出数据访问请求，这台计算机就是客户机。

狭义上的客户机/服务器建立在严格的计算机边界上，即客户机程序和服务器程序分别安装在不同的计算机上，请求将跨越计算机边界。而广义上的客户机/服务器仅将两者认为是两个程序，一个程序提出请求，另一个响应请求，是否分别安装在不同的计算机并不是它判断的依据。这样客户机/服务器可以在一台计算机上实现。

接下来我们将研究另一种开发模式：**B/S**（浏览器/服务器模式）。这种模式可以说是 C/S 的变体，或者是改进。这个模式围绕着 web 服务器来进行，web 服务器是安装了 web 服务软件的计算机，它能够接受客户端发出的 HTTP web 请求，如：HTTP://www.vista-edu.com，处理请求后，产生 HTML 脚本发回客户端；而客户端就不需要象 B/S 那样专门写一个程序，而是变为 IE 浏览器，接收服务器传回的 HTML，然后将该 HTML 显示出来，提供人机交互界面。

B/S 开发中的重点就是编写 web 程序，目前大部分流行的 web 程序，都采用 HTML 加嵌入式服务器端脚本的方式来组织。这样的 web 程序本质上就是一个纯文本文件，可以随时对其进行修改，而不需要重新编译。服务器在接受了一个 HTTP 地址后，就按照事先排定的规则（可能是服务器端的设置，或是由配置文件说明）将请求映射到一个 web 文件，然后由 web 服务器来加载这个 web 文件，并且解释执行，执行后就得到 HTML 的结果（也可以是其它的文档类型，如：doc，xls，jpg 等）。将这个 HTML 结果返回到客户端，由客户端的浏览器解释执行它。

B/S 模式的优点：

- 客户端基于统一的 WEB 浏览器，减少了投资，解决了系统维护升级的问题
- 系统功能模块化：采用模块化结构，使用户可以根据管理要求和规模对系统功能进行调整。
- 灵活性和可扩展性：系统可根据规模的不断扩大，在不影响用户日常工作的前提下，对 WEB 服务器和数据库服务器等设备进行扩展
- 简易性：操作直观、简单，培训方便，对使用人员的计算机操作水平要求不高。
- 实施成本低：充分利用现有的办公网络，避免了网络重复建设

目前流行着多种 B/S 开发语言：如 asp,php 等等，java 也有开发 web 的功能。

1.2 Java web 开发模式

Java 提供了专门的 web 组件来进行 web 开发。

- **Servlet**：Java Servlet 实质上是一种小型的、与平台无关的 Java 类，它由 web 服务器的容器管理并被编译成平台无关的字节代码，这些代码可以动态地加载到一个 web 服务器上，并

由该 web 服务器运行。通过一种由 servlet 容器实现的请求——响应模型与 Web 客户机进行交互。

- Java Server pages(jsp)提供的功能大多和 java Servlet 类似，不过实现的方式不同，servlet 全部由 java 写成并且生成 HTML；而 JSP 通常是大多数 HTML 代码中嵌入少量的 Java 代码。

随着这些技术的诞生，产生了多种多样的开发模式：

- Jsp+Jdbc

在 B/S 开发中最简单的一种开发模式是页面+逻辑处理，映射到技术上反应出来的有 Jsp+Jdbc，在基于这类的实现中在 View 层也就是 jsp 页面上负责数据的显示、逻辑处理，结合 jdbc 完成数据的访问。

- Jsp+JavaBean

Jsp+JavaBean，在这个体系中由 jsp 页面负责显示以及接收页面请求，并调用相应的 JavaBean 来完成逻辑处理，在获取其返回的处理数据后转到相应的页面进行显示。在这样的技术体系中，由于逻辑是由 JavaBean 来完成的，可以对其进行调试了，代码的重用性一定程度上也得到了提高。

- 基于 MVC Framework

在 Java B/S 开发中引入了 MVC 思想，MVC 强调 View 和 Model 的分离。Controller 负责接受页面请求，并将其请求数据进行封装，同时根据请求调用相应的 Model 进行逻辑处理，在 Model 处理后返回结果数据到 Controller，Controller 将根据此数据调用相应的 View，并将此数据传递给 View，由 View 负责将数据进行融合并最终展现。

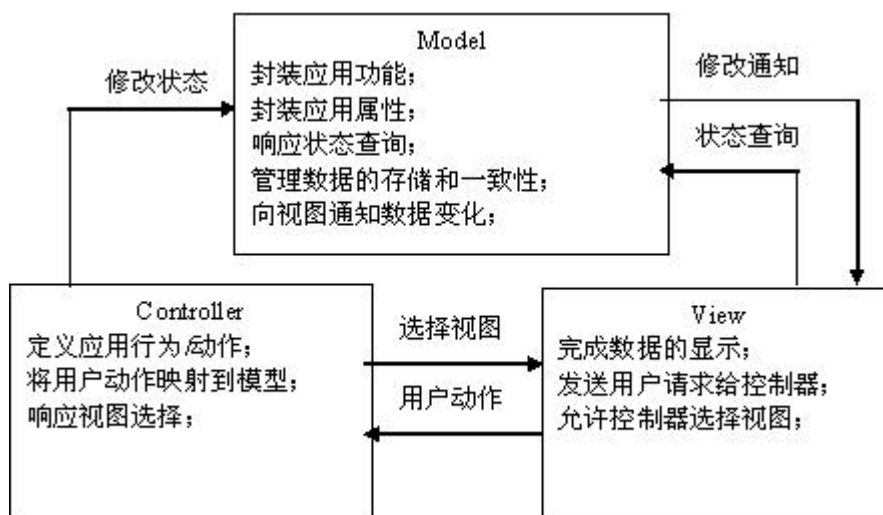


图 1-1 MVC 结构图

按照 MVC 思想，最容易想到的实现方案莫过于 jsp+servlet+java bean，在这里面 jsp 对应着 View，servlet 对应着 Controller，java bean 对应着 Model。作为 Controller 同时还需要承担根据请求调用对应的 java bean。

采用 MVC 的优点

➤ 提高代码重用率

最重要的一点是多个视图能共享一个模型，无论用户想要 Flash 界面或是 WAP 界面；用一个模型就能处理它们。由于已经将数据和业务规则从表示层分开，所以可以最大化的重用代码。

➤ 提高程序的可维护性

因为模型是自包含的，并且与控制器和视图相分离，所以很容易改变数据层和业务规则。例如，把数据库从 MySQL 移植到 Oracle，只需改变模型即可。一旦正确的实现了模型，不管数据来自哪里，视图都会正确的显示它们。MVC 架构的运用，使得程序的三个部件相互对立，大大提高了程序的可维护性。

➤ 有利于团队开发

在开发过程中，可以更好的分工，更好的协作。有利于开发出高质量的软件。良好的项目架构设计，将减少编码工作量：采用 MVC 结构 + 代码生成器，是大多数 Web 应用的理想选择。部分模型(Model)、和存储过程一般可用工具自动生成。控制器(Controller)比较稳定，一般由于架构师完成；那么整个项目需要手动编写代码的地方就只有视图(View)了。在这种模式下，个人能力不再特别重要，只要懂点语法基础的人都可以编写，无论项目成员写出什么样的代码，都在项目管理者的可控范围内。即使项目中途换人，也不会有太大问题。在个人能力参差不齐的团队开发中，采用 MVC 开发是非常理想的。

显而易见，MVC 开发模式具有极大的优点，所以目前的 java web 开发通常都采用 MVC 的模式进行开发，后续的学习也围绕着它展开。

1.3 Tomcat Web 服务器

Jakarta Tomcat 服务器是由 Apatch, Sun 和其他一些公司及个人共同开发的一个免费的开源的 Servlet 容器，Tomcat 是 jakarta 项目中的一个重要的子项目，其被 JavaWorld 杂志的编辑选为 2001 年度最具创新的 java 产品，同时它又是 sun 公司官方推荐的 servlet 和 jsp 容器，因此其越来越多的受到软件公司和开发人员的喜爱。servlet 和 jsp 的最新规范都可以在 tomcat 的新版本中得到实现。其次，Tomcat 是完全免费的软件，任何人都可以从互联网上自由地下载。

它包含 servlet/jsp 容器。负责处理客户请求，把请求传送给 Servlet 并把结果返回给客户。Servlet 包含很多的 API 函数，这些 API 函数在 Servlet 的执行过程中会被 Servlet 容器自动调用；并且 Servlet 容器还负责向 Servlet 传递一些对象，如：封装了客户端请求信息的 ServletRequest 对象，代表服务器传递回客户端的响应对象 ServletResponse。

建立B/S的核心是创建“web应用”。web应用是一个基于web的程序集合，能够使客户端通过发送http请求来访问它。更简单的理解：一个web应用就是一个网站。而在物理上，一个web应用体现为tomcat目录中webapps文件夹下的子文件夹。webapps文件夹是专门用于放置和布置web应用，在它的根目录中每一个web应用就体现为一个文件夹。比如：有一个web应用名称为demo，它在tomcat目录中就体现为在webapps文件夹中的demo文件夹，而HTTP地址默认就为http://localhost:8080/demo。

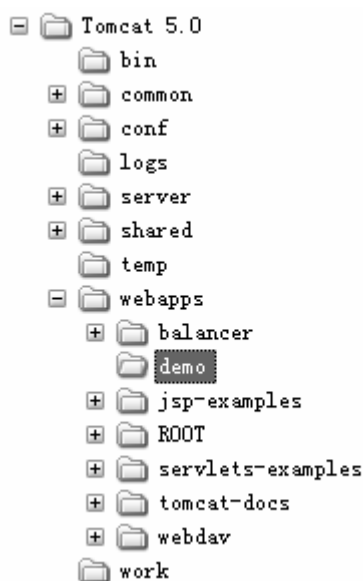


图 1-2 Tomcat 目录结构

其中：

- bin: 存放启动和关闭 tomcat 脚本；
- conf: 存放不同的配置文件（server.xml 和 web.xml）；
- doc: 存放 Tomcat 文档；
- lib/jasper/common: 存放 Tomcat 运行需要的库文件（JARS）；
- logs: 存放 Tomcat 执行时的 LOG 文件；
- src: 存放 Tomcat 的源代码；
- webapps: Tomcat 的主要 Web 发布目录（包括应用程序示例）；
- work: 存放 jsp 编译后产生的 class 文件；

在 Tomcat 中还包含一个非常重要的配置文件，可以通过这个配置文件来设置 Tomcat。该配置文件是放置在<CATALINA_HOME>/CONF 文件夹下面的 server.xml，它的主要结构包括：

```

<Server>
  <Service>
    <Connector/>
    <Engine>
      <Host>
        <Context>
        </Context>
      </Host>
    </Engine>
  </Service>
</Server>

```

- 其中我们所关注的是： <Connector/>元素
 - ◆ **Port**: 指定服务器端要创建的端口号，并在这个端口监听来自客户端的请求。默认是 8080。相应的 HTTP 请求就应为：http://localhost:8080/demo;
 - ◆ **minProcessors**: 服务器启动时创建的处理请求的线程数
 - ◆ **maxProcessors**: 最大可以创建的处理请求的线程数
 - ◆ **enableLookups**: 如果为 true，则可以通过调用 request.getRemoteHost()进行 DNS 查询来得到远程客户端的实际主机名，若为 false 则不进行 DNS 查询，而是返回其 IP 地址。
 - ◆ **acceptCount**: 指定当所有可以使用的处理请求的线程数都被使用时，可以放到处理队列中的请求数，超过这个数的请求将不予处理。
 - ◆ **connectionTimeout**: 指定超时的时间数(以毫秒为单位)。
- <Host>元素，它代表了运行在虚拟主机。
- <Context>元素，它代表了运行在虚拟主机上的单个 WEB 应用。
 - ◆ **docBase** 应用程序的路径或者是 WAR 文件存放的路径
 - ◆ **path** 表示此 web 应用程序的 url 的前缀,这样请求的 url 为 http://localhost:8080/path/****
 - ◆ **reloadable** 这个属性非常重要，如果为 true，则 tomcat 会自动检测应用程序的 /WEB-INF/lib 和 /WEB-INF/classes 目录的变化，自动装载新的应用程序，我们可以在不重启 tomcat 的情况下改变应用程序

1.4 Tomcat 样例程序

Tomcat 软件可以从 <http://tomcat.apache.org/> 网站免费下载。进入网站后选择好版本和平台，就可以下载了。下载结束后就得到了一个可执行文件，然后根据提示安装就可以了。但是要注意在安装前，要确保已经安装好了 JDK。

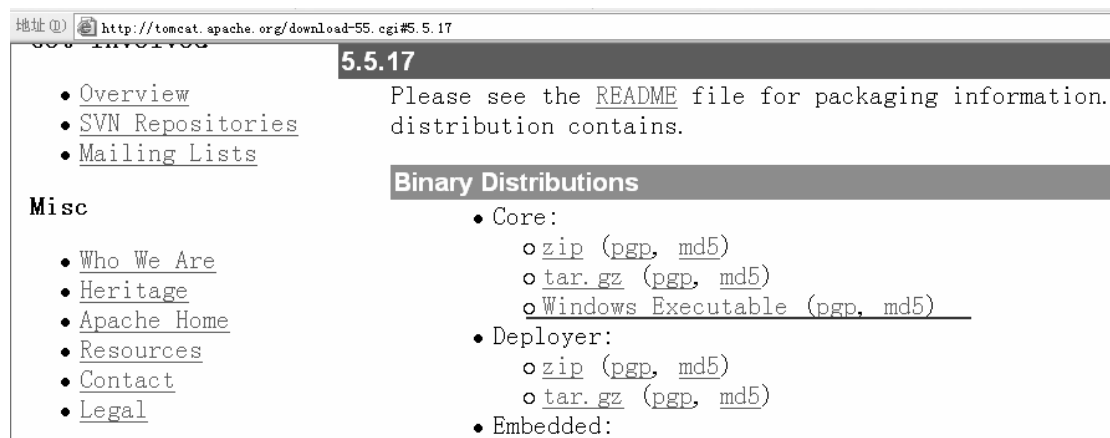


图 1-3 Tomcat 的下载页面

当成功安装完成后，可以打开 IE 浏览器，在地址栏输入 HTTP://localhost:8080 来测试 Tomcat web 服务器是否成功安装。如果成功安装，则在 IE 中将出现下述画面：

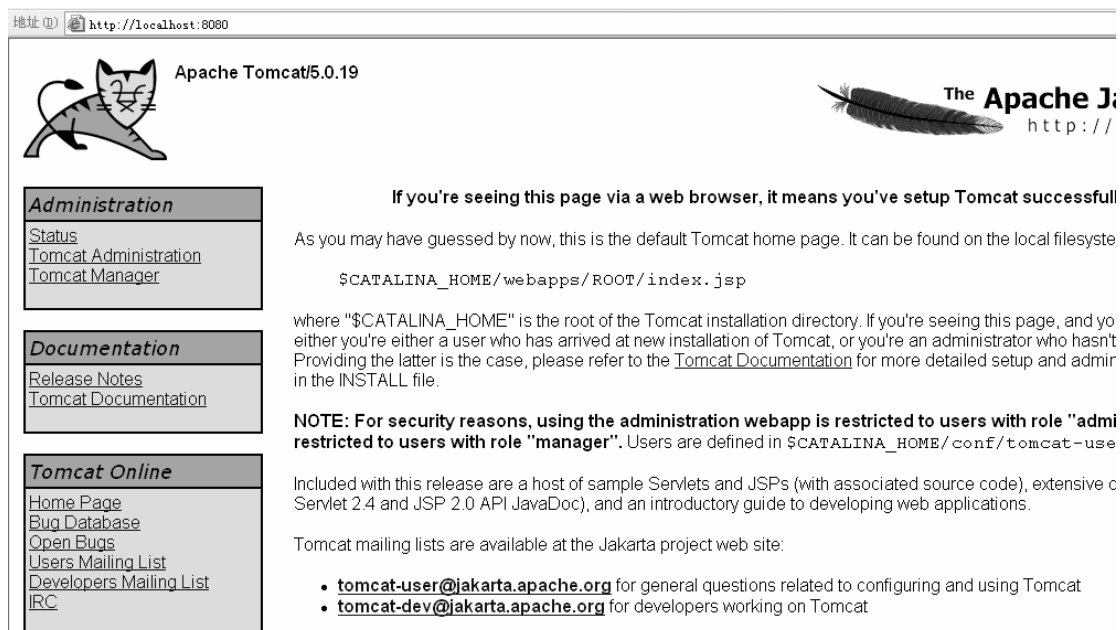


图 1-4 Tomcat 的欢迎 web 页面

Tomcat 软件中还包含了一个重要的内容，就是它的样例程序。在 Tomcat 的欢迎页面的下方有着这样一个超链接。

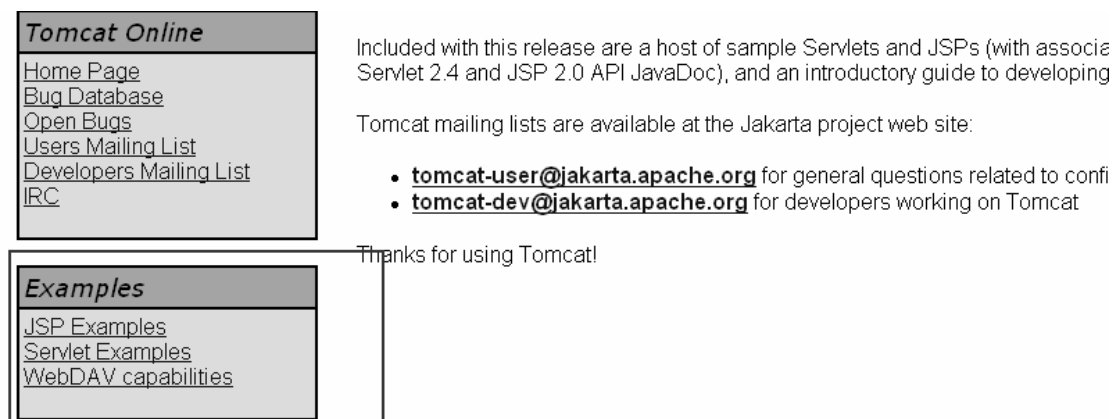


图 1-5 Tomcat 的样例超链接页面

Tomcat 提供了三种示例，第一个和第二个就是所要研究的 Jsp 和 Servlet。当点击进入时，将看到 6 个示例程序的超链接，可以直接执行示例和也可以看到源代码。

Tip: To see the cookie interactions with your browser, try turning on the "notify when setting a cookie" option in your browser preferences. ' feedback when looking at the cookie demo.


Hello World	 Execute	 Source
Request Info	 Execute	 Source
Request Headers	 Execute	 Source
Request Parameters	 Execute	 Source
Cookies	 Execute	 Source
Sessions	 Execute	 Source

图 1-6 Tomcat 的 Servlet 样例页面

点击 Hello world 的 Execute 超链接，将看到 Hello World 这个 servlet 执行的结果。



图 1-7 Tomcat 的 hello world 样例页面

而回到上一页面点击 Hello world 的 Source 超链接，将看到这个 WEB 页面的源代码，可以看到 Servlet 就是从特定类派生出来的 java 类。

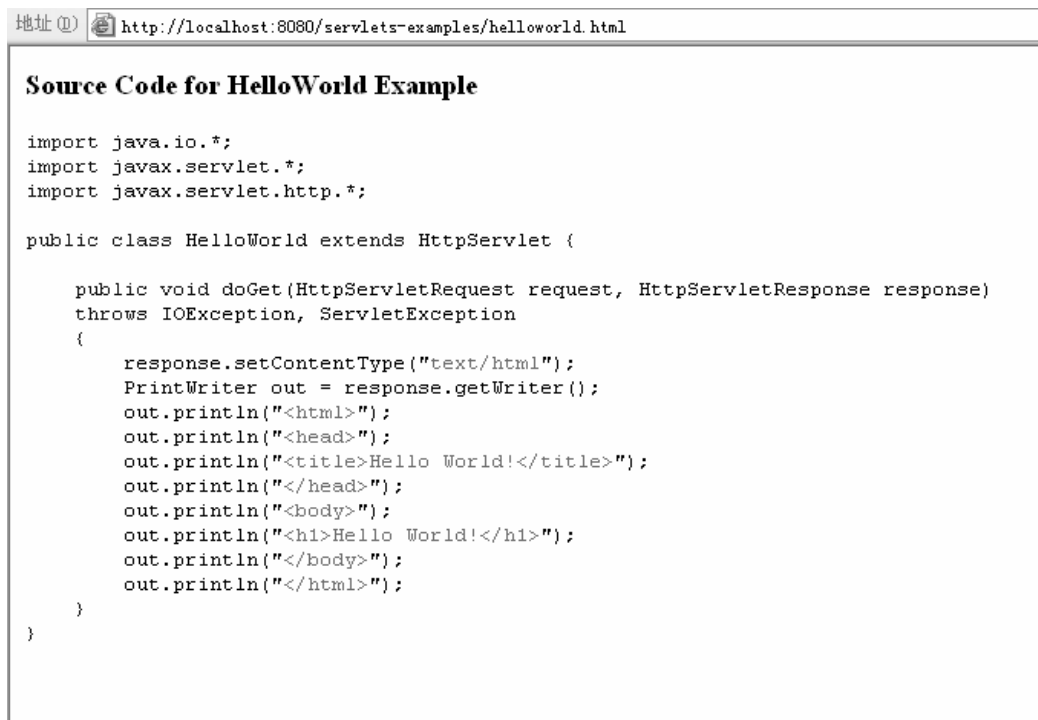


图 1-7 Tomcat 的 hello world 样例源代码页面

这些样例程序是存放在 Tomcat 安装目录的 webapps 的 servlet-Example 和 jsp-Example 两个文件夹里面的。如果需要可以直接到这两个文件夹中去查看源程序。

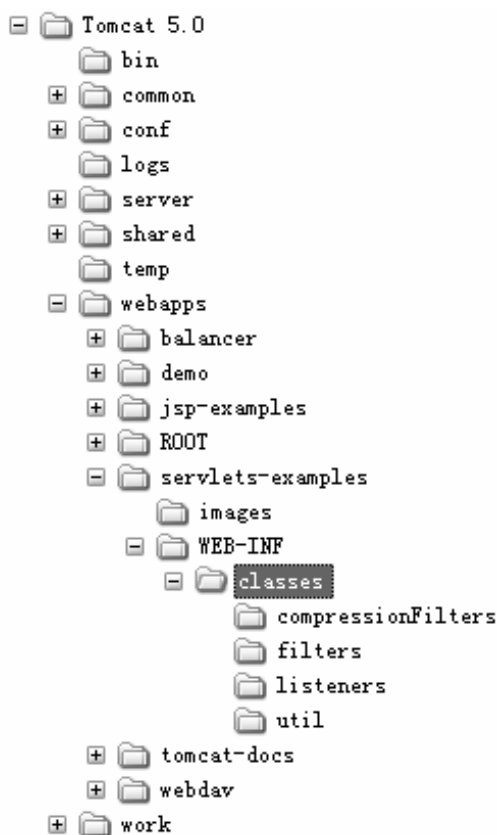


图 1-8 Tomcat 中样例所在的页面

1.5 Eclipse web 开发插件

Eclipse 是 IBM 公司出品的功能强大的 java 集成开发环境(Integrated Development Environment 简称 IDE)，由于免费，插件数量众多，功能齐全等原因，在 java 开发中日益流行起来。Eclipse 是一个开放源代码的软件开发项目，它由 Eclipse 项目、Eclipse 工具项目、Eclipse Web 工具平台项目和 Eclipse 技术项目四个项目组成。

可以通过不断地加载插件来实现同其他制品的合作。整个 Eclipse 体系结构就像一个大拼图，可以不断地向上加插件，同时，现有插件上还可以再加插件，进而实现功能的扩展。目前，Eclipse 已经开始提供 C 语言开发的功能插件。更难能可贵的，Eclipse 是一个开放源代码的项目，任何人都可以下载 Eclipse 的源代码，并且在此基础上开发自己的功能插件。也就是说未来只要有人需要，就会有建立在 Eclipse 之上的 COBOL，Perl，Python 等语言的开发插件出现。同时可以通过开发新的插件扩展现有插件的功能，比如在现有的 Java 开发环境中加入 Tomcat 服务器插件。可以无限扩展，而且有着统一的外观、操作和系统资源管理，这也正是 Eclipse 的潜力所在。

只要事先安装好 JDK，再将 eclipse 从 www.eclipse.com 网站上下载下来，无需安装直接解压缩，就可以进行 java 程序开发了。如果需要开发 j2ee，也只需要下载安装 j2ee 插件，就能得到高效的 j2ee 开发平台。目前可以使用的 j2ee 插件非常多，如：lomboz, MyEclipse 等。而 MyEclipse 所具备的集成度高，配置简单等特点使其成为 j2ee 开发的首选。

MyEclipse 可以在 <http://www.myeclipseide.com/> 网站上下载，解压缩后拷贝进 eclipse 目录中。

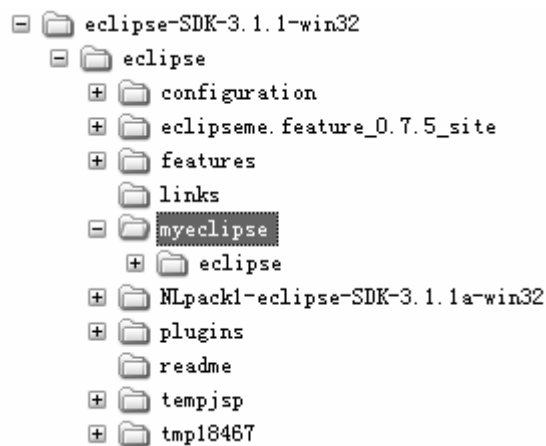


图 1-9 MyEclipse 插件

然后在 links 文件夹中建立相应的链接文件，eclipse 通过侦测该文件夹来获知目前所安装的插件。

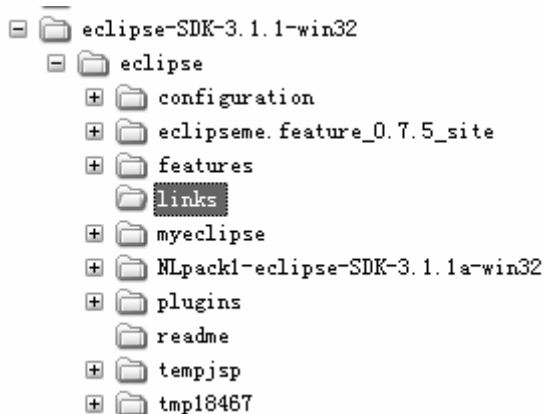


图 1-10 Links 文件夹

链接文件有着一定的命名规则。如图 1-16 中，对 MyEclipse 插件而言，相应的链接文件名就变为：myeclipse.start。

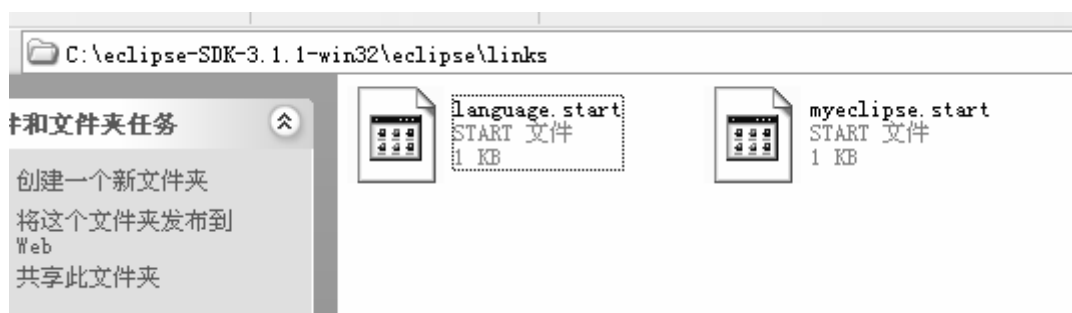


图 1-11 Links 文件

myeclipse.start 文件里需要用 path 指名 myeclipse 文件夹的路径，可以采用绝对路径，用双斜杠来表示路径分割符。



图 1-12 Links 文件

安装好后就可以创建 web 项目了。



图 1-13 创建 web 项目

小结

C/S (Client/Server) 结构，即大家熟知的客户机和服务器结构。它是软件系统体系结构，通过它可以充分利用两端硬件环境的优势，将任务合理分配到 Client 端和 Server 端来实现，降低了系统的通讯开销。

B/S 结构，即 Browser/Server (浏览器/服务器) 结构，是随着 Internet 技术的兴起，对 C/S 结构的一种变化或者改进的结构。在这种结构下，用户界面完全通过 WWW 浏览器实现，一部分事务逻辑在前端实现，但是主要事务逻辑在服务器端实现

MVC 设计模式早在面向对象语言 Smalltalk-80 中就被提出并在此后得到业界的广泛接受。它包括三类对象：(1) 模型 (Model) 对象：是应用程序的主体部分。(2) 视图 (View) 对象：是应用程序中负责生成用户界面的部分。(3) 控制器 (Controller) 对象：是根据用户的输入，控制用户界面数据显示及更新 Model 对象状态的部分

Tomcat 是 jakarta 项目中的一个重要的子项目，是 sun 公司官方推荐的 servlet 和 jsp 容器(具体可以见 <http://java.sun.com/products/jsp/tomcat/>)，因此其越来越多的受到软件公司和开发人员的喜爱。servlet 和 jsp 的最新规范都可以在 tomcat 的新版本中得到实现。

MyEclipse 是 Eclipse 平台下开发 j2ee 应用的非常优秀的工具。通过它可以快速的开发 web 应用，自动管理文件系统，支持相应得应用部署，启动 web 服务等。

课后练习

- 1、 MVC 模式中，如果想设定出货价格必须高于进货价格 80%这样的业务规则，可以将这个规则放在哪个部分来完成？
- 2、 如果想以 `http://localhost:8080/`来访问开发的 web 应用，应该将该应用放在 Tomcat 文件夹中的什么位置？

第二章 Servlet 概述

概要

Java web 开发中的核心组件有两个：Servlet、Jsp，这些组件再结合一些编程思想的指导，就可以开发出各种 Java Web 应用，学习这些组件的开发方法是开发 WEB 应用的基础。本章主要介绍 Servlet 的基础知识,包括 Servlet 的概念、编写 Servlet 程序应遵守的规则、基本 Servlet 中常用的类和方法、Servlet 的生命周期。

目标

- Servlet 概述（理解）
- Servlet 开发方法（掌握）
- Servlet 生命周期（理解）

目录

- 2.1 Servlet 概述
- 2.2 Servlet 示例
- 2.3 service 方法
- 2.4 doXXX 方法
- 2.5 Servlet 生命周期

2.1 Servlet 概述

2.1.1 Servlet 概念

随着 Internet 网络的迅速普及，许多企业逐渐利用 B/S 结构的应用程序来取代传统的 C/S 结构的应用程序，这就促进了基于分布式 Web 应用程序的发展。一个分布式 Web 应用程序的各个部分分布在网络中的不同计算机上，整个 Web 应用运行在 TCP/IP 协议的网络上。

一个 Web 应用程序的处理流程如下图所示：

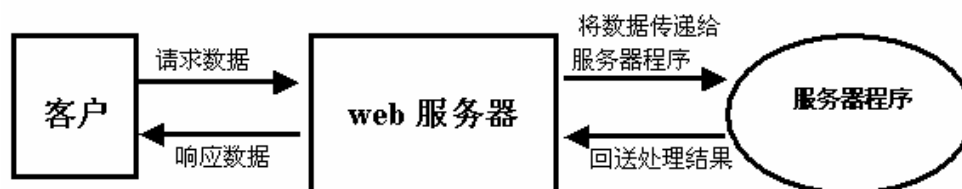


图 2-1 Web 应用处理流程

在这个过程中，前端（表示层），一般是浏览器，从用户那里收集数据后，将数据通过 HTTP 协议发送到服务器端，Web 服务器收到数据后，会运行客户端指定的服务器端程序，并向该服务器端程序传递接收到的用户数据（也称请求），运算的结果由 Web 服务器返回给浏览器，运算的结果以 html、xml 或二进制数据表示，浏览器会根据数据类型进行相应的处理和显示。

服务器端程序和 Web 服务器是紧密结合在一起的，服务器端程序具有下面几个特点：

- 程序必须能够被 WEB 服务器激活。当收到从浏览器发出的请求时，WEB 服务器应该能够定位并装入相应的运行环境执行该程序。
- 能够有一种机制把浏览的表单数据传送给该程序。
- 程序被激活后，要有一个标准的入口点来开始执行，并进行一些初始化操作。
- 程序可以把输出写入到 http 的响应中，传回给浏览器。

Java servlet 是 sun 公司提出的针对企业级 Web 应用的 JavaAPI，使用这些 API 可以开发出服务器端程序，这些程序会自动满足以上提到的服务器端程序的特征。Java Servlet 是 Java 整体解决方案的一部分，在 servlet 程序中能够访问所有其他的 JavaAPI，可以利用 JAVA 语言提供的所有功能（如：可以利用 java mail api 收发邮件、可以使用 java bean 对数据库进行访问、可以使用 HashMap 等各种集合类进行业务处理……）。以下为 Servlet 工作流程图：



图 2-2 Servlet 工作流程

使用 servlet 的基本流程如下：

- 客户端(很可能是 Web 浏览器)通过 HTTP 提出请求。

- Web 服务器接收该请求并将其发给 servlet。如果这个 servlet 尚未被加载，Web 服务器将把它加载到 Java 虚拟机并且执行它。
- servlet 将接收该 HTTP 请求并执行某种处理。
- servlet 将向 Web 服务器返回应答。
- Web 服务器将从 servlet 收到的应答发送给客户端。

另外，一定要注意：Web 浏览器并不直接和 servlet 通信，servlet 是由 Web 服务器加载和执行的。

2.1.2 servlet 体系结构

刚才分析了 servlet 在 web 应用程序中的访问流程，要运行一个 Servlet，我们要做以下工作：

需要一个支持 Servlet API 的 Web 服务器（如：Tomcat）

具备某种能够调用 servlet 的客户端应用程序（如：IE 浏览器）

开发并部署实现业务逻辑的 Servlet 类

其中程序员的核心工作就是第三个工作（开发 Servlet），其实 Servlet 的本质就是一个符合了指定规范的 JAVA 类，下面三种方式都可将一个 Java 类变为 Servlet：

1) 直接实现 javax.servlet.Servlet 接口

javax.servlet.Servlet 接口是所有 Java Servlet 的基本接口，Servlet 的基本方法由该接口定义。

该接口中包含 5 个方法：

1. public void init(ServletConfig config)throws ServletException
2. public void service(ServletRequest req,ServletResponse res)
throws ServletException,java.io.IOException
3. public java.lang.String getServletInfo()
4. public ServletConfig getServletConfig()
5. public void destroy()

2) 继承 javax.servlet.GenericServlet

3) 继承 javax.servlet.http.HttpServlet

以上 java 包是标准 java 的扩展包，没有包含在 j2sdk 中，在安装 Java Web 服务器时，会自动在你的机器上安装 servlet 包；上述三个类（接口）的关系如下：



图 2-3 Servlet 体系结构

从图 2-3 中可以看出，GenericServlet 实现了 Servlet 接口，而 HttpServlet 扩展了 GenericServlet。用户定义的类只要符合了上述三个方式，就实现了 Servlet 接口，也就是说用户类就符合了 Servlet 规范，从而成为 Servlet 类。

2.1.3 Servlet 的功能

刚才分析了 servlet 的体系结构和 servlet 的运行流程，在整个 web 应用程序中，Servlet 的整体功能体现如下：

- 1) 借助 WEB 服务器，接收客户端请求数据
- 2) 对接收到的数据进行处理，通常我们称为业务逻辑处理，如：在一个用户注册系统中，servlet 要负责接收注册信息，并且要完成以下业务逻辑：判断数据格式是否完整、数据库中是否有同名的用户存在、如果没有同名用户则进行注册操作。
- 3) 在服务器端调用其他 WEB 资源，Servlet 在服务器端通常充当“管家”的地位，在 J2EE 中称为“控制器”，在 WEB 应用中，Servlet 本身只负责整体业务流程，在与数据库打交道时，Servlet 要去调用其他类（通常是 javabean）去完成 JDBC 操作，在业务处理完成后，Servlet 还要负责选择一个合适的视图（通常是 JSP）发送到客户端浏览器，在 Servlet 中通常不直接将处理结果输出到浏览器中。

2.2 Servlet 示例

上面讲解了 Servlet 基本概念，其本质就是一个符合了 Servlet 规范的类，一旦实现了这个规范，它就可以被部署到 Web 服务器中被远程用户访问到，下面我们来开发一个简单的 Servlet 应用，以了解其开发流程。整个开发工作分为以下几步：

- 1) 编写 Servlet 源代码，并编译成 class 字节码文件；
- 2) 将 class 文件放到 web 站点指定位置；
- 3) 在站点中的 web.xml 描述符文件中配置 servlet，以使客户端可对其进行访问；
- 4) 在客户端（即浏览器中）进行访问；

2.2.1 编写、编译 Servlet 源代码

如前面 1.1.2 中所述，有三种方式可以生成 Servlet：

➤ 方式一：直接实现 javax.servlet.Servlet 接口

该接口中声明有五个方法：

- 1) void destroy()
- 2) ServletConfig getServletConfig()
- 3) String getServletInfo()
- 4) init(ServletConfig config)
- 5) service(ServletRequest req, ServletResponse res)

在类中只需覆盖全部五个方法即可，其中最重要的是 service 方法，其他方法功能可以为空，这些方法的作用在后续内容中会讲解到。下面是一个直接实现 Servlet 接口的 Servlet：

```
package test;
import javax.servlet.*;
import java.io.*;
//覆盖 Servlet 接口的五个方法
public class MyServlet implements Servlet{
    public void init(ServletConfig config)
```

```

        throws ServletException{
System.out.println("MyServlet init()");
    }
    public ServletConfig getServletConfig(){
        return null;
    }
    public java.lang.String getServletInfo(){
        return "";
    }
    public void service(ServletRequest req,
                        ServletResponse res)
                        throws ServletException, java.io.IOException{
System.out.println("MyServlet service()");

//下面语句用于设置输出的内容为简体中文编码格式
res.setContentType("text/html;charset=gb2312");

//下面语句用于得到输出流，该输出流的目标是客户浏览器
PrintWriter out=res.getWriter();

//向客户端输出文本结果，最终会显示在客户浏览器中
out.println("这是一个 Servlet,直接实现了 Servlet 接口.");
    }
    public void destroy(){
        System.out.println("MyServlet1 destroy()");
    }
}

```

➤ 方式二：继承 javax.servlet.GenericServlet

GenericServlet 类由 servlet 包提供，并且已经实现了 javax.servlet.Servlet 接口，还添加了许多新的功能，该类中有一个名为 service 的方法为抽象方法，其他方法均已实现，自定义的类继承该类后，只要覆盖这一个方法即可变为合法的 servlet，自然根据需要也可以覆盖其他方法（如：init、destroy 方法，在后续内容中会讲到）。下面是一个继承了 GenericServlet 的 Servlet：

```

package test;

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

//覆盖 GenericServlet 的 service 方法

```

```
public class MyGenericServlet extends GenericServlet
{
    public void service(ServletRequest req,ServletResponse res)
        throws ServletException,java.io.IOException{
        System.out.println("MyGenericServlet service()");
        res.setContentType("text/html;charset=gb2312");
        PrintWriter out=res.getWriter();
        out.println("这是一个 Servlet,继承了 GenericServlet.");
    }
}
```

➤ 方式三：继承 javax.servlet.http.HttpServlet

HttpServlet 是 GenericServlet 的子类，HttpServlet 即有普通 servlet 的功能，还提供了对 http 协议的支持，继承该类，通常不必实现 service 方法，因为 HttpServlet 类已经实现了 service 方法，一般实现 doGet 或 doPost 方法即可。这些方法在本章后续内容中有专门讲解。下面是一个继承了 HttpServlet 的 Servlet：

```
package test;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;

public class MyHttpServlet extends HttpServlet{

    //覆盖 GenericServlet 的 doGet 方法
    protected void doGet(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException,
        java.io.IOException{
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("这是一个 Servlet,继承了 HttpServlet.");
        out.println("现在运行的是 doGet()方法.");
    }

    //覆盖 GenericServlet 的 doPost 方法
    protected void doPost(HttpServletRequest request,
        HttpServletResponse response)
```

```

        throws ServletException,
            java.io.IOException{
    response.setContentType("text/html;charset=gb2312");
    PrintWriter out=response.getWriter();
    out.println("这是一个 Servlet,继承了 HttpServlet.");
    out.println("现在运行的是 doPost()方法.");
    }
}

```

使用上述方法的三者之一写好源代码后, 可以进行编译, 在此需要 servlet api 的支持, 即需要用到 servlet 包 (tomcat 会自动安装此包); 另外, 由于绝大多数 web 程序工作在 http 协议之上, 故第三种生成 servlet 的方式是实际开发中最常选择的。

2.2.2 将 class 文件放到 web 站点指定位置——部署 Servlet

编译出 class 字节码文件后, 还要将该文件存放到 web 站点的指定位置才能被 web 服务器正确装载, 通常存放在 站点\WEB-INF\classes 目录下:

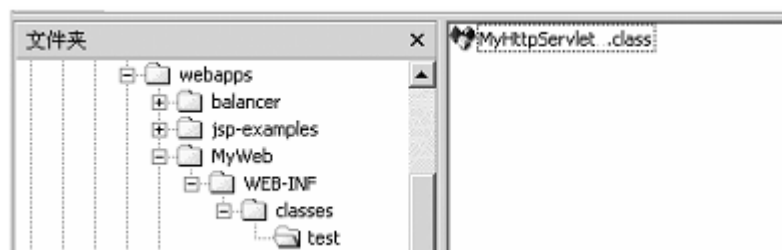


图 2-4 Servlet 的存放位置

2.2.3 在站点中的 web.xml 描述符文件中配置 servlet

客户访问服务器中的 servlet 时, 一般是通过 url (如: http://localhost:8080/MyWeb/myhttpServlet1) 进行访问的, Web 服务是如何知道要调用哪一个 servlet 呢? 这里需要一个配置文件的支持: 站点 \WEB-INF\web.xml, 在该文件中可以为每个 servlet 配置对应的 url 形式。刚才的三个 Servlet 在 web.xml 中的配置如下:

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
    version="2.4">

    <servlet>
        <servlet-name>MyServlet</servlet-name>
        <servlet-class>test.MyServlet</servlet-class>
    </servlet>

```

```
<servlet-mapping>
  <servlet-name>MyServlet</servlet-name>
  <url-pattern>/myservlet</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>MyGenericServlet</servlet-name>
  <servlet-class>test.MyGenericServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>MyGenericServlet</servlet-name>
  <url-pattern>/mygenericservlet</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>MyHttpServlet</servlet-name>
  <servlet-class>test.MyHttpServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>MyHttpServlet</servlet-name>
  <url-pattern>/myhttpservlet</url-pattern>
</servlet-mapping>

<welcome-file-list>
  <welcome-file>index.html</welcome-file>
</welcome-file-list>

</web-app>
```

解释:

➤ <servlet>用于指定要配置的 servlet:

<servlet-class>指定要配置的 servlet 类的名字 (包名要写上);

<servlet-name> 是一个逻辑名, 可以是任何有效的标识名;

➤ <servlet-mapping>用于为 servlet 映射 url 访问形式:

<servlet-name>与<servlet>内的 <servlet-name>一一对应;

<url-pattern>指定在浏览器中 url 的请求形式, 以后只要用这个形式的 url 进行访问, 对应的 servlet

会自动运行，这里的 / 是相对于当前的 web 目录的，\webapps\MyWeb;

- <welcome-file-list>用于为站点指定默认页（主页）：

经过配置的 servlet 就可以被客户端访问到了

2.2.4 在客户端（即浏览器中）进行访问

根据刚才的配置，客户在浏览器中通过“http://localhost:8080/MyWeb/myhttpervlet”，即可访问到 test.MyHttpServlet 类。



图 2-5 在浏览器中访问 servlet

访问流程如下：用户在浏览器地址栏中输入以上网址，打回车时浏览器将这个 url 作为请求发送给服务器 tomcat，tomcat 接收到请求后，会自动根据 web.xml 文件中的配置，去匹配<url-pattern>中值为 /myhttpervlet 的项，继而找到其对应的 servlet 类并装载运行；

2.3 service 方法

刚才的示例 Servlet 中，使用最多的是名为 service()的方法，这个方法的作用是用于处理客户请求，对于到达 servlet 容器(即 servlet 服务器)的客户请求，Servlet 容器创建特定于这个请求的 ServletRequest 对象和 ServletResponse 对象，然后调用 Servlet 的 service()方法，service 方法从 ServletRequest 对象获得客户请求信息并处理该请求，通过 ServletResponse 对象向客户返回响应结果。

- 该方法是在 javax.servlet.Servlet 接口中定义的，其定义如下：

```
public void service(ServletRequest req,ServletResponse res)
    throws ServletException,java.io.IOException
```

可以看出，其两个参数用于接收 Servlet 容器传递过来的请求和响应对象，servlet 程序员要做的工作就是使用这两个对象的方法去处理业务逻辑，然后将结果通过响应对象的方法返回给服务器，服务器接收到结果后会传递给客户端浏览器，所以说 Servlet 并不是直接与客户端浏览器打交道，而是由 Web 服务器这个“中间人”负责联系起来的。

- GenericServlet 实现了 Servlet 接口，但没有实现 service 方法，该方法的声明在 GenericServlet 类中与 Servlet 接口中完全一样；
- HttpServlet 类继承了 GenericServlet，并实现了 service 方法，而且还对其进行了重载，在 HttpServlet 中共有两个 service 方法，其中一个与 GenericServlet 中的声明相同，另一个 service

方法的声明如下：

```
protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, java.io.IOException
```

仔细观察，会发现其参数由 `ServletRequest` 变为了 `HttpServletRequest`，`ServletResponse` 变为了 `HttpServletResponse`，通过这两个接口，使得该 `service` 方法提供了对 `Http` 协议的支持；

那么 `HttpServlet` 类中的两个 `service` 方法是如何运行的？客户请求传递给 `Web` 服务器后，服务器还是会调用第一个版本的 `service`，而 `HttpServlet` 中的 `serviceI()` 会自动调用支持 `Http` 协议的第二个版本的 `service` 方法。从方法声明中也可以看出，支持 `Http` 的 `service` 方法被声明为 `protected`，限制了 `Web` 容器没法对其直接调用。

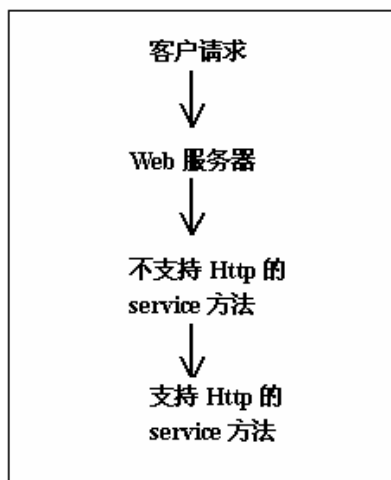


图 2-6 `HttpServlet` 的 `service` 方法

2.4 doXXX 方法

在 `HttpServlet` 中提供了第二个版本的 `service` 方法，那么开发者是不是重写这个方法，在这个方法中就可以实现自己的业务了？开发者通常不会直接重写 `HttpServlet` 的任何一个 `service` 方法，原因如下：

`HttpServlet` 中的第二个 `service` 方法提供了对 `Http` 的支持，当 `HttpServlet` 接收到 `Web` 服务器传递过来的 `HTTP` 请求时，会在 `service()` 方法内判断 `HTTP` 请求消息的方法类型 (`Get`、`Post`、`Head`、`Options`、`Delete`、`Put`、`Trace`)，然后将传入的请求信息转发给相应的处理方法 (`doGet`、`doPost`、`doHead`、`doOptions`、`doDelete`、`doPut`、`doTrace`)。每个 `doXxxx()` 方法和上面第二个 `service()` 定义具有相同的形式，都是用 `HttpServletRequest` 类和 `HttpServletResponse` 类的对象作为参数，并且都抛出 `ServletException` 和 `IOException` 异常。在具体编写扩展 `HttpServlet` 类的 `Servlet` 时，必须根据客户端发出请求的具体情况，重载相应的 `doXxxx()` 方法。如果具体的 `Servlet` 类没有重载相应的 `doXxxx()` 方法，当该 `Servlet` 接收到 `HTTP` 请求时，会返回一个“请求的资源不可用”的标准 `HTTP` 错误。

2.4.1 `Http` 请求方法

`Http` 请求由浏览器根据用户选择自动生成，请求是符合 `Http` 协议规定的字符串，被打包成 `IP` 包在网上传递。比如我们点击超链接 `xxx` 时，浏览器会

生成如下 Http 请求:

```
GET /a.jsp HTTP/1.1
Accept:image/gif,image/jpeg,*/*
Accept-Language:zh-cn
Connection:Keep-Alive
Host:localhost
User-Agent:Mozilla/4.0(compatible;MSIE 5.0.1;Windows NT 5.0)
Accept-Encoding:gzip,deflate

page=1&book=jsp
```

请求的第一行是:“方法 uri HTTP/1.1”:

GET /a.jsp HTTP/1.1

其中"GET"表示请求方法, "/a.jsp"表示 uri, "HTTP1.1"代表协议及版本。

HTTP 协议标准中, 请求可以使用多种请求方法, HTTP1.1 支持 7 种请求方法: Get、Post、Head、Options、Delete、Put、Trace, 但最常用的方法是 GET 和 POST。

2.4.2 doGet、doPost 方法

当用户使用 GET、POST 请求访问服务器时, Servlet 中的 doGet、doPost 方法会被调用。

➤ doGet

1) 通常客户使用以下操作时会产生 GET 请求:

- ◆ 在浏览器地址栏中输入 URL 并回车时
- ◆ 点击超链接时
- ◆ 提交某个表单 (该表单的 method 属性设置为 get 时)
- ◆ 在服务器端运行某些操作时 (如: 重定向等)

2) GET 请求特征:

- ◆ GET 请求只能上传少量的数据, 要上传大量的数据, 比如要上传一张图片, 此时就必须使用 POST 请求;
- ◆ GET 请求上传数据时, 数据会显示在地址栏中

➤ doPost

1) 通常客户使用以下操作时会产生 POST 请求:

- ◆ 提交某个表单 (该表单的 method 属性设置为 post 时)

2) POST 请求特征:

- ◆ POST 请求能上传大量的数据 (理论上讲没有上限)
- ◆ POST 请求上传数据时, 数据不会显示在地址栏中, 而是在幕后打包上传

2.4.3 示例

1、 doGet 示例

```
//index.html
<h4><font color="blue">doGet、doPost 方法演示</font></h4>
<a href="myhttpervlet">用 Get 方法访问 myhttpervlet</a>
<br>
<form action="myhttpervlet" method="get">
    <input type="submit" value="用 Post 方法访问 myhttpervlet"> 点击该按钮
</form>
<br>
```

```
//MyHttpServlet.java
package test;
import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;
import java.io.*;

public class MyHttpServlet extends HttpServlet{

    //覆盖 GenericServlet 的 doGet 方法
    protected void doGet(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException,
            java.io.IOException{
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("这是一个 Servlet,继承了 HttpServlet.");
        out.println("现在运行的是 doGet()方法.");
    }

    //覆盖 GenericServlet 的 doPost 方法
    protected void doPost(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException,
            java.io.IOException{
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("这是一个 Servlet,继承了 HttpServlet.");
        out.println("现在运行的是 doPost()方法.");
    }
}
```

```
}  
}
```

当用户在 index.html 中点击超链接“用 Get 方法访问 myhttpervlet”时，或在中点击表单中的第一个按钮时，MyHttpServlet 中的 doGet()方法会被 Web 服务器调用运行。

2、 doPost 示例

```
//index.html  
....  
<form action="myhttpervlet" method="post">  
    <input type="submit" value="用 Post 方法访问 myhttpervlet"> 点击该按钮  
</form>  
....
```

```
//MyHttpServlet.java  
package test;  
import javax.servlet.*;  
import javax.servlet.http.*;  
import java.util.*;  
import java.io.*;  
public class MyHttpServlet extends HttpServlet{  
    //覆盖 GenericServlet 的 doGet 方法  
    protected void doGet(HttpServletRequest request,  
                           HttpServletResponse response)  
        throws ServletException,  
        java.io.IOException{  
        response.setContentType("text/html;charset=gb2312");  
        PrintWriter out = response.getWriter();  
        out.println("这是一个 Servlet,继承了 HttpServlet.");  
        out.println("现在运行的是 doGet()方法.");  
    }  
    //覆盖 GenericServlet 的 doPost 方法  
    protected void doPost(HttpServletRequest request,  
                           HttpServletResponse response)  
        throws ServletException,  
        java.io.IOException{  
        response.setContentType("text/html;charset=gb2312");  
        PrintWriter out = response.getWriter();  
        out.println("这是一个 Servlet,继承了 HttpServlet.");  
        out.println("现在运行的是 doPost()方法.");  
    }  
}
```

```

    }
}

```

当用户在 index.html 中点击表单中的提交按钮时，MyHttpServlet 中的 doPost() 方法会被 Web 服务器调用运行，如果在该 MyHttpServlet 中没有提供 doGet 方法，所以客户如果直接在浏览器地址栏中输入 URL 进行访问时，服务器会提供错误信息。

2.5 Servlet 生命周期

在用 Java Servlet 进行实际开发时，理解 Servlet 的生命周期是十分必要的。在一个具体的 Http 请求处理响应过程中，容器环境中 Java Servlet 的基本生命周期如下：

- ◆ 当 Web 客户请求 Servlet 服务或当 Web 服务启动时，容器环境加载一个 servlet 类。
- ◆ 容器环境创建一个或多个 Servlet 对象实例，并把这些实例加入到 Servlet 实例池中。
- ◆ 容器环境调用 Servlet 的初始化方法 init() 进行 Servlet 初始化。在调用初始化时，web 服务器会给 init() 方法传入一个 ServletConfig 对象，该对象中包含了初始化参数和容器环境的信息。
- ◆ 容器环境利用一个 HttpServletRequest 和 HttpServletResponse 对象，封装从 Web 客户接收到的 HTTP 请求和由 Servlet 生成的响应。
- ◆ 容器环境把 HttpServletRequest 和 HttpServletResponse 对象传递给 HttpServlet 的 service() 方法，这样，自定义的 Servlet 内部就可以访问 Http 请求和响应对象。
- ◆ 自定义的 Servlet 运行 service 方法（在 httpServlet 中，service 会自动调用 doXxx() 方法），处理业务并输出响应。
- ◆ 当 Web 服务器关闭时，或某个 Servlet 长时间无人访问时，容器调用 Servlet 的 destroy() 方法，进行关闭前的处理。

在 Servlet 的生命期中，init()、destroy() 方法仅执行一次，service() 方法每次有客户访问时都会运行

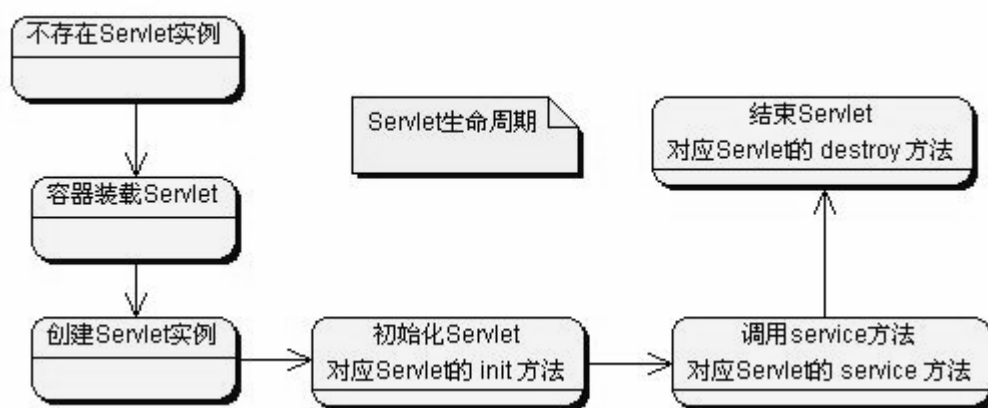


图 2-7servlet 生命周期

➤ Servlet 生命周期示例

```

package test;
import javax.servlet.*;
import java.io.*;

```

```
public class MyServlet implements Servlet{
    public void init(ServletConfig config)
        throws ServletException{
        System.out.println("MyServlet init()");
    }
    public ServletConfig getServletConfig(){
        return null;
    }
    public java.lang.String getServletInfo(){
        return "";
    }
    public void service(ServletRequest req,
                        ServletResponse res)
        throws ServletException, java.io.IOException{
        System.out.println("MyServlet service()");

        //下面语句用于设置输出的内容为简体中文编码格式
        res.setContentType("text/html;charset=gb2312");

        //下面语句用于得到输出流，该输出流的目标是客户浏览器
        PrintWriter out=res.getWriter();

        //向客户端输出文本结果，最终会显示在客户浏览器中
        out.println("这是一个 Servlet,直接实现了 Servlet 接口.");
    }
    public void destroy(){
        System.out.println("MyServlet1 destroy()");
    }
}
```

我们在浏览器中多次调用该 Servlet，通过查看 Tomcat 控制台的输出可以观察到：init()、destroy() 方法在生命周期中只执行一次，而 service 方法每次有用户访问都会运行。

运行结果如下图所示：

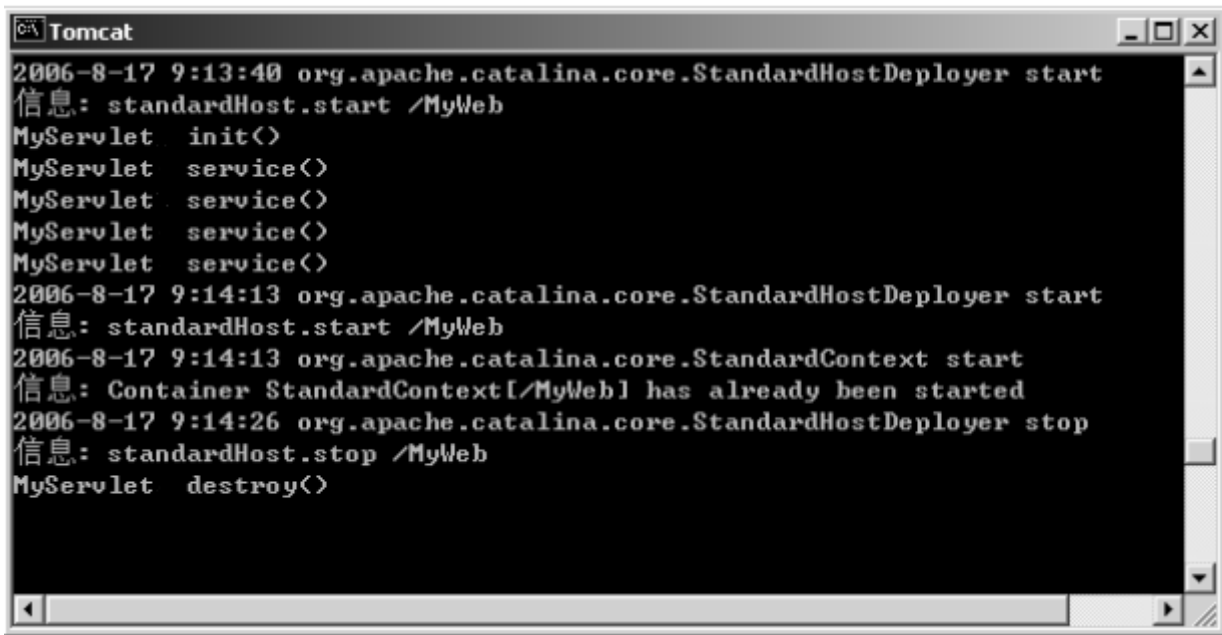


图 2-8 servlet 生命周期示例

注意：要想观察到 `destroy()` 方法的运行效果，需要到 Tomcat 管理员页面停止 web 站点的运行，在站点停止时，Servlet 会被卸载，从而运行 `destroy()` 方法。也可以关闭浏览器等待一段时间（大约 20 分钟左右），长时间无人访问时该 Servlet 也会被卸载。

小结

Servlet API 是 sun 提供的开发基于 JAVA 的 Web 服务器端程序的规范，开发 Servlet 时需要实现这些规范。

Servlet 规范中，最重要的三个类（接口）是：`javax.servlet.Servlet`、`javax.servlet.GenericServlet`、`javax.servlet.http.HttpServlet`，开发自定义 Servlet 最常用的方法是继承 `javax.servlet.http.HttpServlet`。

`service()` 方法是 Servlet 规范中最重要方法，在该方法中开发者实现业务处理，并返回结果给客户。端。`javax.servlet.Servlet`、`javax.servlet.GenericServlet` 中有一个 `service()` 方法，而 `javax.servlet.http.HttpServlet` 中有两个 `service()` 方法，新加的一个 `service()` 对 Http 提供了支持。

开发 `HttpServlet` 时，不要重写 `service()` 方法，通常重写 `doGet()`、`doPost()` 其中之一即可，或者两者全部重写。

Servlet 生命周期方法有：`init()`、`service()`、`destroy()`，在 Servlet 对象的不同生命时刻，相应的方法会自动运行。

课后练习

- 1、 开发一个 Servlet，有几种方法？分别如何实现？
- 2、 简述 HttpServlet 中以下方法的联系：service()、doGet()、doPost()？
- 3、 简述 Servlet 生命周期及各生命时刻要运行的方法是什么？

第三章 Servlet 环境

概要

Java Servlet API 为编程人员实现服务器端操作提供了丰富的支持，通过上一章的学习我们可以生成一个 Servlet，在开发一个实际的 Servlet 时还需要用到其它 Servlet API。其中包括两个最重要的用于所有 Servlet 的基本软件包:javax.servlet 和 javax.servlet.http,本章主要介绍几个 Servlet 常用 API 及 web.xml 的文档结构。

目标

- Servlet 常用接口（掌握）
- Servlet 请求转发（掌握）
- Servlet 请求包含（了解）
- Servlet 二进制内容输出（理解）
- web.xml 文档结构（掌握）

目录

- 3.1 HttpServletRequest
- 3.2 HttpServletResponse
- 3.3 ServletConfig
- 3.4 ServletContext
- 3.5 RequestDispatcher
- 3.6 二进制内容输出
- 3.7 Web.xml 文档结构

3.1 HttpServletRequest

3.1.1 Servlet API 简介

Servlet API 库的用法相当简单，但为开发人员提供了巨大的功能。Servlet 的核心包有 4 个，分别是 `javax.servlet.*`、`javax.servlet.http.*`、`javax.servlet.jsp.*` 和 `javax.servlet.jsp.tagext.*`，各包功能如下：

包名	功能
<code>javax.servlet.*</code>	基本的通用的 Servlet API 包
<code>javax.servlet.http.*</code>	基于 HTTP 的 web 应用专用扩展 Servlet API
<code>javax.servlet.jsp.*</code>	创建 JSP 的类
<code>javax.servlet.jsp.tagext.*</code>	jsp 类专用扩展

表 3-1 Servlet 核心包

以上 4 个包中，前两个包用于 Servlet 开发，后两个包主要是在 JSP 中使用，在后续内容中有讲解，本章着重讲解 `javax.http.*` 和 `javax.servlet.http.*`，`javax.servlet` 包提供整个 API 的基本结构，最常用的接口包括：

API	API 类型	功能描述
Servlet	接口	这个接口定义所有 Servlet 都要实现的方法， <code>GenericServlet</code> 类实现了该接口。
ServletRequest	接口	关于客户机请求的所有信息可以通过实现这个接口的对象访问，Servlet 引擎（通常包含在 WEB 服务器内部）负责创建 <code>ServletRequest</code> 对象。
ServletResponse	接口	Servlet 引擎要创建实现这个接口的对象并传入 Servlet 的 <code>service</code> 方法，用于向客户机输出响应信息。
ServletConfig	接口	该接口的实现类的对象保存 Servlet 初始化期间使用的信息。
ServletContext	接口	该接口的实现类的对象使用 Servlet 可以找到所运行 Servlet 容器及其环境信息。
RequestDispatcher	接口	这个接口可以将请求从当前 Servlet 转发到另一个 Servlet 或 Jsp 页面中继续处理。
SingleThreadModel	接口	这个接口不包含方法，是个标志，使 Servlet 引擎保证 Servlet 实例中一次只运行一个线程。Servlet 引擎可以限制访问一个 Servlet 实例或对每个线程创建一个实例。在第 5 章会专门讨论线程安全相关内容。
GenericServlet	类	提供最基本 Servlet 的功能。
ServletInputStream	类	读取请求中二进制数据流的类。
ServletOutputStream	类	向响应中写入二进制数据流的类。

表 3-2 javax.servlet 包中的 API

用 Servlet 创建 Web 应用时，要使用 `javax.servlet.http` 包中的类。为了充分提高 Servlet 引擎设计人员的灵活性，这个包的大多数功能在接口中定义：

API	API 类型	功能描述
<code>HttpServletRequest</code>	接口	这是 <code>ServletRequest</code> 接口的扩展，增加针对 HTTP 请求的方法。
<code>HttpServletResponse</code>	接口	这是 <code>ServletResponse</code> 接口的扩展，可以向客户端输出 HTTP 响应信息。
<code>HttpSession</code>	接口	使用这个接口的对象，开发人员可以在多个页面间存储用户信息。
<code>HttpServlet</code>	类	抽象类，继承了 <code>Servlet</code> 。
<code>Cookie</code>	类	这些对象操作服务器发送给浏览器和浏览器返回给服务器的 Cookie 信息。

表 3-3 `javax.servlet.http` 包中的 API

3.1.2 `HttpServletRequest`

该接口的声明如下：

```
javax.servlet.http
public interface HttpServletRequest extends ServletRequest
```

`HttpServletRequest` 接口的实现用于封装 HTTP 请求消息，该接口定义了获得请求信息的方法，开发人员可以通过该接口读取到客户发送到服务器的数据。所谓请求，是指由浏览器发送到服务器的所有数据的总和，这些数据大体上可以分成两个部分：一部分由用户提供，可以称为“用户数据”，比如：在一个用户注册系统中，用户提供的用户名、密码、Email 等；第二部分数据由浏览器自动在客户机上搜集得到，通常以 HTTP “协议头”形式发送给 WEB 服务器，比如：客户所用的操作系统的类型、版本、客户所在的地区信息、客户机使用的语言等。

`HttpServletRequest` 重要接口的方法列表如表所示：

方法	功能描述
<code>getCookies()</code>	返回该请求消息包含的所有 Cookie。
<code>getHeader(String name)</code>	返回由 <code>name</code> 指定名称的报头的值。如果报头不存在则返回空，报头名称大小写不敏感。
<code>getHeaderNames()</code>	返回由请求消息中所有报头名称构成的一个枚举，如果报头不存在则返回 <code>null</code> 。
<code>getMethod()</code>	返回请求方法名称。例如：“GET”、“POST”等。
<code>getPathInfo()</code>	返回请求 URI 中在 Servlet 路径之后的附加路径信息，如果没有信息，则返回 <code>null</code> 。
<code>getQueryString()</code>	返回请求 URI 中“?”后的查询参数字符串。如果没有字符串，则返回 <code>null</code> 。
<code>getRequestSessionId()</code>	返回请求中包含的会话 ID 的值。如果没有会话 ID 值，则返回 <code>null</code> 。

<code>getServletPath()</code>	返回请求 URI 中指向 Servlet 的部分。
<code>getSession()</code>	取得与当前请求相关联的会话。如果会话尚不存在，就新建一个。
<code>setAttribute(String name,Object p)</code>	用指定的键名，向请求中存储对象。
<code>getAttribute(String name)</code>	从请求中取出指定键名的对象，返回值为 Object 类型。
<code>removeAttribute(String name)</code>	从请求中删除指定键名的对象
<code>setCharacterEncoding(String env)</code>	设置请求中信息字符编码格式
<code>getParameter(java.lang.String name)</code>	读取请求中指定名字的参数，参数通常由表单中的元素提供。
<code>getParameterNames()</code>	读取请求中所有参数的名字。
<code>getParameterValues(java.lang.String name)</code>	读取同名参数的多个值，比如：用户注册时可能会选择多个爱好。

表 3-4 HttpServletRequest 接口的方法

我们通常在 HttpServlet 的 `doGet()`、`doPost()` 方法中使用这个接口，这两个方法声明如下：

```
protected void doGet(HttpServletRequest request,HttpServletResponse response)
    throws ServletException, java.io.IOException
protected void doPost(HttpServletRequest request,HttpServletResponse response)
    throws ServletException, java.io.IOException
```

现在的问题是：参数 `request` 指向的对象何时生成？由谁生成？该对象是哪个类的对象？前两个问题好回答：客户浏览器将请求发送给 WEB 服务器时，WEB 服务器会自动生成 `HttpServletRequest` 对象，然后传递到 `doXXX()` 方法中来。

但 `HttpServletRequest` 是一个接口，怎么会生成对象呢？答案肯定是生成了 `HttpServletRequest` 接口的实现类的对象，在 J2EE 规范中，这个实现类，由 WEB 服务器提供，我们可以认为这个类存在于服务器内部，也可以将它作为一个匿名类看待，总之参数 `request` 指向了 `HttpServletRequest` 接口的实现类的对象，这是 JAVA 多态性的体现，作为开发者，我们只须调用 `request` 引用中的方法即可。

下面通过实例，来学习 `HttpServletRequest` 接口的方法。

示例 1——用户注册

这个例子中，要求客户从浏览器中输入自己的注册信息，然后在 `servlet` 中接收，并将注册成功的消息输出到客户浏览器中，本例中并没有真正去连接数据库，我们关心的主要内容是 `servlet` 如何去接收请求信息。

```
//reg.html
<html>
<head>
<title>新用户注册</title>
</head>
```

```

<body>
<center>新用户注册</center>
<form action="regservlet" method="POST">
用户名:<input type="text" size="10" name="username"/><br />
密码:<input type="password" size="10" name="userpass"/><br />
爱好:
<input type="checkbox" name="hb" value="看书"/>看书
<input type="checkbox" name="hb" value="睡觉"/>睡觉
<input type="checkbox" name="hb" value="游泳"/>游泳
<input type="checkbox" name="hb" value="游戏"/>游戏

<input type="submit" value="注册" />
</form>
</body>
</html>

```

以上页面的表单中，有两个单值元素（用户名、密码），有一个多值元素（爱好，一个元素名对应多个值），提交后我们通过以下 Servlet 来接收表单参数值：

```

package demo;

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

public class RegServlet extends HttpServlet {
    /**
     * 演示以下方法的使用：
     * setCharacterEncoding、getParameter、getParameterValues
     */
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=GBK");
        PrintWriter out = response.getWriter();
        //接收信息；
        //设置按简体中文格式读取请求信息，无此语句则接收中文会出现乱码；
        request.setCharacterEncoding("gb2312");
        String username = request.getParameter("username");
    }
}

```

```
String userpass = request.getParameter("userpass");
String array1[] = request.getParameterValues("hb");

//输出响应
out.println("<h3>你的信息如下:");
out.println("用户名:" + username + "<br>");
out.println("密码:" + userpass + "<br>" );
out.println("爱好:" );
for(int i = 0; i < array1.length; i++){
    out.println(array1[i] + ",");
}
out.println("</h3>");
}
}
```

运行时，先访问 reg.html，并填写表单：



图 3-1 访问 reg.html

点击“注册”后，RegServlet 响应如下：

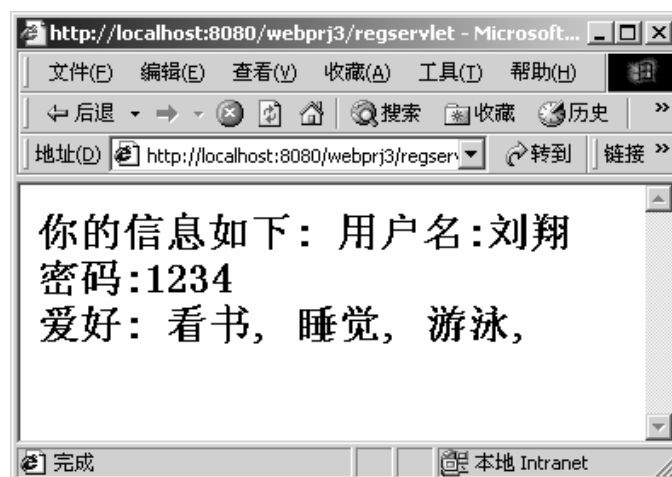


图 3-2 RegServlet 的响应

示例 2——创建一个可以接收任意表单数据的 Servlet

刚才的例子中，我们在开发 Servlet 时，必须知道要读取的表单中有几个元素，各元素名称分别是什么，但有时我们需要读取的表单中有哪些元素是不知道的（如：在 struts 框架中有 ActionServlet，这个 Servlet 可以读取任意表单中的数据），此时应该如何编码呢，Servlet API 中提供了相应的功能。

下面例子中演示如何编写可以接收任意表单数据的 Servlet：

```
//reg2.html
<html>
<head>
<title>新用户注册</title>
</head>
<body>
<center>新用户注册</center>
<form action="actionServlet" method="POST">
用户名:<input type="text" size="10" name="username"/><br />
密码:<input type="password" size="10" name="userpass"/><br />
爱好:
<input type="checkbox" name="hb" value="看书"/>看书
<input type="checkbox" name="hb" value="睡觉"/>睡觉
<input type="checkbox" name="hb" value="游泳"/>游泳
<input type="checkbox" name="hb" value="游戏"/>游戏

<input type="submit" value="注册" />
</form>
</body>
```

假设我们不知道以上表单中有多少元素，可以编写如下 Servlet 来接收未知表单的数据：

```
//ActionServlet.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class ActionServlet extends HttpServlet {
    /**
     * 演示以下方法的使用法：
     * getParameterNames、getParameterValues
     */
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=GBK");
        PrintWriter out = response.getWriter();
        //接收信息；
        //设置按简体中文格式读取请求信息，无此语句则接收中文会出现乱码；
        request.setCharacterEncoding("gb2312");

        //得到集合的枚举器(Enumeration 相当于 JDBC 中结果集的指针)
        java.util.Enumeration e = request.getParameterNames();
        while( e.hasMoreElements() ){
            String str1 = (String)e.nextElement();
            String array1[] = request.getParameterValues(str1);
            out.println("<h3>" + str1 + "：");
            for(int i = 0; i < array1.length; i++){
                out.println(array1[i] + "，");
            }
            out.println("</h3>");
        }
    }
}
```

运行 reg2.html 并点击提交按钮后，ActionServlet 显示如下：



图 3-3 能接收任意表单的 Servlet

解释：

➤ `getParameterNames()` 方法：

可以得到表单中所有元素的名字，当我们不知道表单中有多少元素时，可以使用该方法获取，该方法的返回值为 `java.util.Enumeration`，它相当于 JDBC 结果集中的指针，指向了所有元素名字所在的集合，该对象有两个常用方法：

`hasMoreElements()`：判断集合中是否还有元素；

`nextElement()`：取出指针所指位置的元素，并且指针向下移动；

➤ `getParameterValues()` 方法：

考虑为什么不使用 `getParameter()` 方法？因为我们不知道表单元素是单值元素还是多值元素，通过该方法将所有元素均当作多值元素看待，该方法返回值为一个 `String` 数组，其中包含了某个元素的所有值。

3.2 HttpServletResponse

`HttpServletResponse` 接口表示了一个 HTTP servlet 的应答，通过它可以将数据或 Http 头信息发回客户端，可以向客户端发送文本或二进制数据。该接口声明如下：

```
javax.servlet.http
public interface HttpServletResponse extends ServletResponse
```

3.2.1 HttpServletResponse 接口方法

`HttpServletRequest` 接口的重要方法列表如下：

方法	功能描述
<code>setContentType()</code>	设置输出字符编码格式
<code>getWriter()</code>	得到字符输出流，目的地是客户端浏览器
<code>getOutputStream()</code>	得到字节输出流，目的地是客户端浏览器，但可以通过它输出二进制流
<code>setContentType(String)</code>	设置将要输出到响应中的数据的 MIME 类型，如要将 excel 文件、pdf 文件输出到客户浏览器时必须先设置类型

addCookie(Cookie)	向响应中加入指定的 Cookie
encodeURL(String)	将指定的 url 编码以便在 sendRedirect 方法中使用, 通常在 url 中包含特殊字符时使用, 如包含: 空格、&字符、? 等
sendError(int)	向客户端发送一个错误状态代码, 客户端浏览器会显示相应错误信息
sendRedirect(String)	将客户端的响应重定向到指定 URL 上
setHeader(String,String)	在响应中加入指定响应头, 并赋给它一个字符串值

表 3-5 HttpServletResponse 接口的方法

3.2.2 HttpServletResponse 接口示例

示例 1——用户登录

在用户登录系统中, 往往要判断用户是否为合法用户, 此例中我们在 Servlet 中判断用户填写的登录信息是否正确(假设数据库中存在用户 admin, 密码是 admin), 而且我们不允许用户注册时提供的用户名中包含单引号, 因为包含这类信息意味着用户可能想通过“SQL 注入”绕过系统, 在发现用户名中包含单引号时我们将错误状态代码发送到客户端以保护服务器不被攻击。

```
//login.html
<html>
<head>
<title>用户登录</title>
</head>
<body>
<center>用户登录</center>
<form action="loginservlet" method="POST">
用户名:<input type="text" size="10" name="username"/><br />
密码:<input type="password" size="10" name="userpass"/><br />
<input type="submit" value="登录" />
</form>
</body>
</html>
```

```
//LoginServlet.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
public class LoginServlet extends HttpServlet {  
    /**  
     * 演示以下方法的使用法：  
     * sendRedirect、sendError  
     */  
    public void doPost(HttpServletRequest request,  
                        HttpServletResponse response)  
        throws ServletException, IOException {  
        response.setContentType("text/html; charset=GBK");  
        PrintWriter out = response.getWriter();  
        //接收信息；  
        //设置按简体中文格式读取请求信息,无此语句则接收中文会出现乱码；  
        request.setCharacterEncoding("gb2312");  
        String username = request.getParameter("username");  
        String userpass = request.getParameter("userpass");  
  
        //如果用户名中包含单引号，则发送错误状态编号  
        if( username.indexOf("'")>-1 ){  
            response.sendError(500);  
        }  
        if( username.equals("admin") && userpass.equals("admin") ){  
            out.println("<h1>欢迎,您已成功登录!</h1>");  
        }  
        else{  
            //重定向  
            response.sendRedirect("login.html");  
        }  
    }  
}
```

运行 login.html，在登录表单输入用户名：admin，密码：admin，则显示成功登录信息。如果用户名中包含单引号，则客户端得到错误状态码：

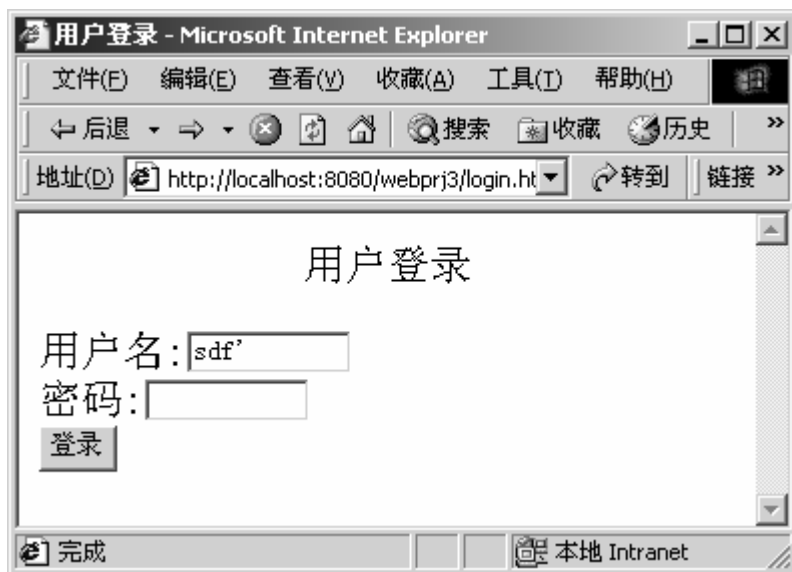


图 3-4 登录用户名中包含单引号

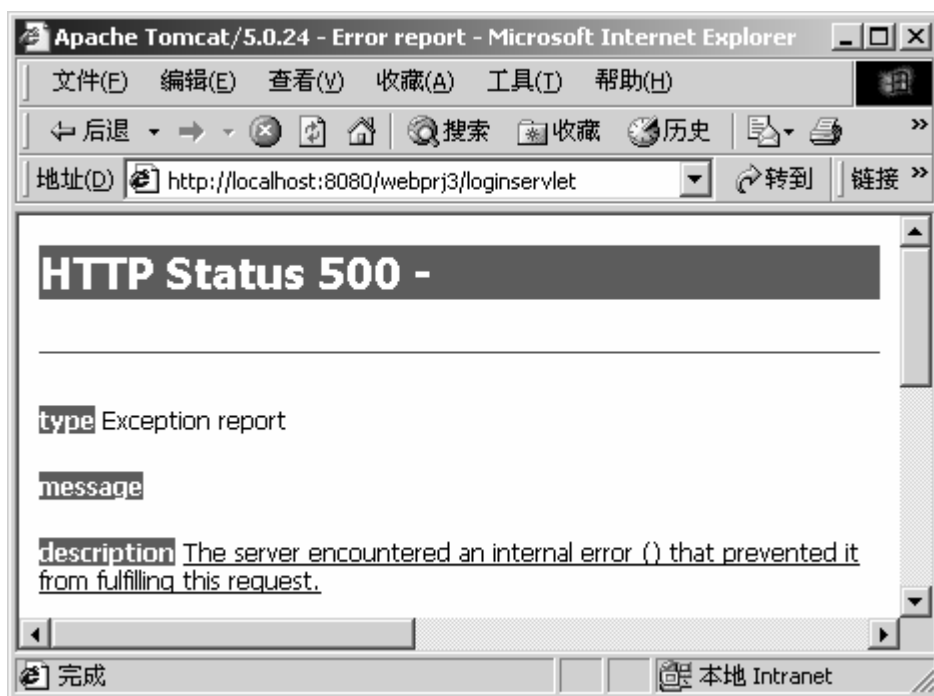


图 3-5 客户端得到错误状态码

3.3 ServletConfig

实现 ServletConfig 接口的类,可以在 Servlet 实例化时向 Servlet 传递配置信息,配置信息由 servlet 容器提供。ServletConfig 中还定义了获取 ServletContext 对象的方法。

3.3.1 ServletConfig 接口方法

ServletConfig 接口的重要方法列表如下:

方法	功能描述
getInitParameter(String)	返回指定名称的初始化参数的值，如果参数不存在，则返回 null
getInitParameterNames()	返回所有初始化参数名字的集合的枚举器，如果没有初始化参数，则返回 null
getServletContext()	返回此 Servlet 中的上下文对象，上下文对象中包含 WEB 应用中的全局性信息

表 3-6 ServletConfig 接口的重要方法

说明：以上提到的初始化参数在 web.xml 中配置 servlet 时提供，下面是一个配置的例子，在此例中，为 InitParamsServlet 传递了三个初始化参数，每个参数由<init-param>定义，三个参数名称分别为：p1、p2、p3，参数值由<param-value>指定，我们可以在 Servlet 中通过 ServletConfig 读取这些信息：

```
.....
<servlet>
    <servlet-name>InitParamsServlet</servlet-name>
    <servlet-class>demo.InitParamsServlet</servlet-class>
    <init-param>
        <param-name>p1</param-name>
        <param-value>1</param-value>
    </init-param>
    <init-param>
        <param-name>p2</param-name>
        <param-value>2</param-value>
    </init-param>
    <init-param>
        <param-name>p3</param-name>
        <param-value>3</param-value>
    </init-param>
</servlet>
.....
```

3.3.2 ServletConfig 示例

示例 1——通过 ServletConfig 读取初始化参数：

```
//InitParamsServlet .java
package demo;

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

```
public class InitParamsServlet extends HttpServlet {  
    /**  
     * 通过 ServletConfig 读取初始化参数  
     */  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        response.setContentType("text/html;charset=gb2312");  
        PrintWriter out = response.getWriter();  
        out.println("chb.servlet.InitParamsServlet 初始化参数<hr>");  
  
        ServletConfig config = getServletConfig();  
        Enumeration initParams = config.getInitParameterNames();  
        int n = 0;  
        while(initParams.hasMoreElements()){  
            String initParam = (String)initParams.nextElement();  
            out.println("<h3>" + initParam);  
            out.print(": " + config.getInitParameter(initParam));  
            out.println("</h3>");  
            n++;  
        }  
        out.println("<hr>共有参数" + n + "个,使用 ServletConfig 接口可以取得在 web.xml  
中为某个 Servlet 配置的参数.");  
    }  
}
```

运行结果如下:



图 3-6 通过 ServletConfig 读取初始化参数

3.4 ServletContext

Servlet 容器在启动时会加载 WEB 应用，并为每一个 WEB 应用创建一个唯一的 ServletContext 对象，ServletContext 接口的对象，封装了 Servlet 运行的环境信息，并且可以存储 WEB 应用的全局对象，开发者在 Servlet 中可以通过该接口获得 Servlet 容器的相关信息。由于 ServletContext 中可以存储全局信息，故该对象常用来存储几个页面或几个 Servlet 共用的信息，如聊天室系统中的公共发言内容。

3.4.1 ServletContext 常用方法

ServletContext 接口的重要方法列表如下：

方法	功能描述
setAttribute(String, Object)	将对象用指定键名存储到上下文对象中
getAttribute(String)	返回指定名字的属性值，如果不存在返回 null
removeAttribute(String)	从上下文对象中删除指定键名的对象
getAttributeNames()	返回所有属性的名称集合的枚举器
getMajorVersion()	返回当前 Servlet API 的主版本号
getMinorVersion()	返回当前 Servlet API 的次版本号
getMimeType()	返回给定文件的 MIME 类型
getRealPath(String)	返回给定虚拟路径对应的物理路径
getResourceAsStream(String)	将一个指定资源转变为 InputStream 对象
getServerInfo()	得到当前 Servlet 容器的名字
getInitParameter(String)	得到初始化参数
getInitParameterNames()	得到所有初始化参数的名字的枚举器

表 3-7 ServletConfig 接口的重要方法

说明：通过 ServletContext 中也可以获取初始化参数，这些参数与 ServletConfig 读取的参数区别如下：

- 在 web.xml 中的配置方法不同;
- ServletConfig 读取的参数属于某个特定的 Servlet, 其他 Servlet 不能读取; ServletContext 读取的参数属于整个 WEB 应用, 所有 Servlet 共享, 都可以读取到。

web.xml 中全局初始化参数配置方法如下 (注意, 这些标记不要嵌到 <Servlet> 配置内部):

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <!-- =====以下开始传递 context 参数===== -->
    <context-param>
        <param-name>dbDriverName</param-name>
        <param-value>org.gjt.mm.mysql.Driver</param-value>
    </context-param>
    <context-param>
        <param-name>dbName</param-name>
        <param-value>myWebDB</param-value>
    </context-param>
    .....
</web-app>
```

3.4.2 ServletContext 示例

示例 1——通过 ServletContext 存取全局对象:

```
//ContextServlet2.java
package demo;

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

public class ContextServlet2 extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gbk");
        PrintWriter out = response.getWriter();
    }
}
```

```

//得到服务器传递给 servlet 的一个 ServletContext 对象
ServletConfig config = this.getServletConfig();
ServletContext context = config.getServletContext();

context.setAttribute("contextparam1", "1111111");
context.setAttribute("contextparam2", "2222222");
/*String s=config.getInitParameter("dbDriverName");
out.println("s 的值是:"+s);
//获取 web.xml 文件中的变量值。*/
//得到一个 Enumeration 对象, 其中含有在 servlet 上下文中有有效的属性名。
Enumeration enum = context.getAttributeNames();
while (enum.hasMoreElements())
{
    String key = (String)enum.nextElement();
    Object value = context.getAttribute(key);
    out.println("<b>" + key + "</b> = " + value+"<br>");
}
//从 ServletContext 取得某个指定值;
out.println("<br>从 ServletContext 取得某个指定值<br>");
out.println("<br><b>contextparam1</b>="+context.getAttribute("contextparam1"));
out.println("</body></html>");
}
}

```

运行该 Servlet, 结果如下:

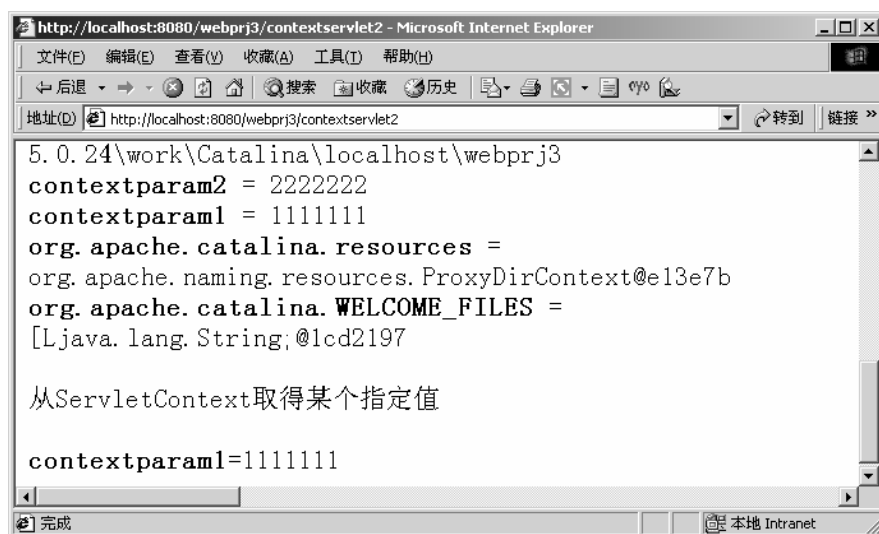


图 3-7 通过 ServletContext 存取全局对象

说明:

由于存储在 `ServletContext` 中的对象属于整个 WEB 应用所共有，所有在其他 `Servlet` 中也可以访问得到 `contextpara2`、`contextparam1` 这些参数值；另外，除了开发人员向 `ServletContext` 中添加全局数据外，WEB 服务器也向该对象中添加许多全局数据，通过上图可以看出，如键名为 `org.apache.catalina.resources` 的全局数据。

3.5 RequestDispatcher

在实际业务系统中，常常需要多个 `Servlet` 共同协作来完成一项处理，这就需要某个 `Servlet` 将自己接收到的请求数据能够通过一种方法传递给另外一个 `Servlet`，`Servlet` API 中的 `RequestDispatcher` 接口专门提供此类功能，`RequestDispatcher` 接口提供了将请求转发给另外的资源，或在一个 `Servlet` 的响应中包含另外的资源的输出。

3.5.1 RequestDispatcher 常用方法

`RequestDispatcher` 接口的重要方法列表如下：

方法	功能描述
<code>forward(ServletRequest,ServletResponse)</code>	将请求转发给服务器上的其他资源
<code>include(ServletRequest,ServletResponse)</code>	在响应中包含服务器上的其他资源的输出内容

表 3-8 `RequestDispatcher` 接口的重要方法

在 `Servlet` 中得到 `RequestDispatcher` 对象的方法如下：

```
ServletContext servletContext=this.getServletContext();
RequestDispatcher requestDispatcher =
    servletContext.getRequestDispatcher("/RequestDispatcherServlet_2");
requestDispatcher.forward( request, response ); //请求转发
requestDispatcher.include( request, response ); //请求包含，第一个 Servlet 中会包含第二个 Servlet 内容
```

3.5.2 RequestDispatcher 示例

示例 1——请求转发

所谓请求转发，就是第一个 `Servlet` 将其接收到的请求数据传递给第二个 `Servlet` 或其他服务器组件（JSP 等），在其他组件中可以继续处理请求数据；

与请求重定向的区别：通过 `HttpServletResponse.sendRedirect()` 方法可以将请求重定向到其他资源，但第一个 `Servlet` 中的请求数据在第二个资源中不能访问到。

重定向的流程：`sendRedirect()` 方法使服务器向客户端浏览器发送一个命令，浏览器收到命令后再去请求服务器上的指定页面，一个重定向过程中，浏览器其实生成了一个新的请求；

请求转发的流程：第一个 `Servlet` 直接在服务器上请求数据传递给其他 `Servlet`，客户端不知道后台发生的事情，整个访问过程中，客户端浏览器只生成了一个请求。

下面例子中演示一个注册系统，第一个 `Servlet` 负责接收请求，然后转发给第二个 `Servlet` 进行下一步处理：

```
//forward.html
<HTML><HEAD><TITLE>注册</TITLE>
<META content="text/html; charset=gb2312" http-equiv=Content-Type>
```

```

<BODY bgColor=#ffffff>
<FORM action="requestdispatcherervlet_1" method="post">
<P>&nbsp;  </P>
<TABLE align=center border=2 width="49%">
  <TBODY>
    <TR align=middle bgColor=#6633cc>
      <TD align=middle colSpan=2>
        <H4><FONT color=white
        face="Verdana, Arial, Helvetica, sans-serif">注册-请求转发
        </FONT></H4></TD></TR>
    <TR bgColor=#ffffcc>
      <TD align=middle width="43%">
        <DIV align=right><FONT
        face="Verdana, Arial, Helvetica, sans-serif">用户: </FONT></DIV></TD>
      <TD width="57%">
        <DIV align=left><FONT face="Verdana, Arial, Helvetica, sans-serif">
        <INPUT name=username> </FONT></DIV></TD></TR>
    <TR bgColor=#ccff99>
      <TD align=middle width="43%">
        <DIV align=right><FONT
        face="Verdana, Arial, Helvetica, sans-serif">密码: </FONT></DIV></TD>
      <TD width="57%">
        <DIV align=left><FONT face="Verdana, Arial, Helvetica, sans-serif"><INPUT
        name=password type=password> </FONT></DIV></TD></TR>
    <TR bgColor=#ffffcc>
      <TD align=middle width="43%">
        <DIV align=right><FONT
        face="Verdana, Arial, Helvetica, sans-serif">电子信箱:</FONT></DIV></TD>
      <TD width="57%">
        <DIV align=left><FONT face="Verdana, Arial, Helvetica, sans-serif"><INPUT
        name=Email> </FONT></DIV></TD></TR>
    <TR bgColor=#ccff99>
      <TD align=middle width="43%">
        <DIV align=right><FONT
        face="Verdana, Arial, Helvetica, sans-serif">昵称: </FONT></DIV></TD>
      <TD width="57%">
        <DIV align=left><FONT face="Verdana, Arial, Helvetica, sans-serif"><INPUT
        name=nickname> </FONT></DIV></TD></TR></TBODY></TABLE>

```

```
<P align=center>
<INPUT name=Submit2 type=submit value=进入>
<INPUT name=Reset type=reset value=重填></P>
</FORM></BODY></HTML>
```

点击以上表单提交按钮，数据被传递给下面的 Servlet:

```
//RequestDispatcherServlet_1.java
package demo;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestDispatcherServlet_1 extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request,response);
    }
    public void doPost(HttpServletRequest request,
                        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();

        //得到转发器 RequestDispatcher
        ServletContext servletContext = this.getServletContext();
        RequestDispatcher requestDispatcher =
        servletContext.getRequestDispatcher("/requestdispatcherservlet_2");
        request.setAttribute("requestAttr1", "requestAttr1_value");

        //由于要进行请求转发，下面语句输出在浏览器中是看不到的
        out.println("<h2>我是 RequestDispatcherServlet_1</h2>");
        requestDispatcher.forward( request, response );
    }
}
```

在该 servlet 中请求被 requestDispatcher.forward(request, response); 转发到到了 requestdispatcherservlet_2，下面为 requestdispatcherservlet_2 的内容:

```
// RequestDispatcherServlet_2.java
package demo;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class RequestDispatcherServlet_2 extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        String title = "表单提交的数据如下";
        request.setCharacterEncoding("gb2312");

        out.println("<BODY BGCOLOR=\\\"#FDF5E6\\\">\\n" +
            "<H1 ALIGN=\\\"CENTER\\\">" + title + "</H1>\\n" +
            "<h3>我是 RequestDispatcherServlet_2</h3>"+
            "<h3>从 RequestDispatcherServlet_1 转发过来，也带过来几个 context、"
            "request、session 级 Attribute</h3>"+
            "<UL>\\n" +
            "  <LI><B>用户</B>: "
            + request.getParameter("username") + "\\n" +
            "  <LI><B>密码</B>: "
            + request.getParameter("password") + "\\n" +
            "  <LI><B>电子信箱</B>: "
            + request.getParameter("Email") + "\\n" +
            "  <LI><B>昵称</B>: "
            + request.getParameter("nickname") + "\\n" +
            "</UL>\\n");

        out.println("<br>requestAttr1="+
            request.getAttribute("requestAttr1")+ "<br>");
        out.println("</BODY></HTML>");
    }
}
```

```
}
```

访问 forward.html，并填写表单数据：



图 3-8 填写表单数据

点击“进入”提交按钮，结果如下：

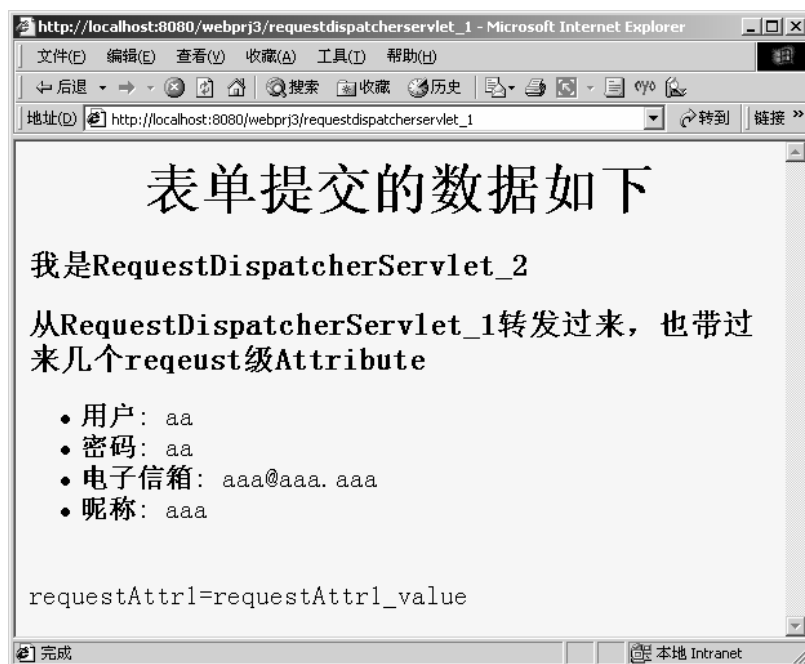


图 3-9 请求转发结果

解释：

- 表单数据直接发送给 requestdispatcherservlet_1，但在后台又转发给了 requestdispatcherservlet_2，页面上显示的是 requestdispatcherservlet_2 的输出，但地址栏中仍然显示为 requestdispatcherservlet_1，证明客户端只生成了一次请求。
- 本例中使用了 `HttpServletRequest.setAttribute()` 方法，该方法可以在请求中存储指定的对象，然后可以转发到其他 Servlet 或 jsp 中。在其他 Servlet 中使用 `HttpServletRequest.getAttribute()`

方法可以读取到。

- 如果将转发换成 `response.sendRedirect()` 方法，表单数据及用 `setAttribute()` 存储在 `request` 中的值是传递不到第二个 `Servlet` 中的。

3.6 二进制内容输出

到目前为止，我们通过 `HttpServletResponse` 可以向客户端输出文本数据和 `HTTP` 协议头数据，但许多应用中需要向客户端输出二进制数据，最典型的应用是一个下载系统中，在下载系统中通常要求用户输入一个密码，通过验证后再将文件输出到客户端，这些文件通常是二进制的，比如：`PDF` 文件、`Excel` 文件、`MP3` 文件等。

要向客户端输出二进制数据，要用到 `HttpServletResponse` 的 `getOutputStream` 方法，通过该方法可以得到一个字节输出流，输出目的地是客户端浏览器，然后我们向该流中输出二进制数据即可。

3.6.1 示例

下面示例要求客户输入密码，然后将一个 `PDF` 文件输出到客户端浏览器中。

```
//download.html
<HTML><HEAD><TITLE>文件下载</TITLE>
<BODY bgColor=#ffffff>

<FORM action="downloadervlet" method="post">
    请输入验证码:<INPUT type="text" name="password">
    <INPUT name=Submit2 type=submit value=下载>
    <INPUT name=Reset type=reset value=重填></P>
</FORM>
```

输出密码“abc”后，点击“下载”，数据发送到 `Servlet`，`Servlet` 在内部判断密码是否为“abc”，如果密码错误不让下载，否则直接将文件输出到浏览器中：

```
//DownloadServlet .java
package demo;

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

public class DownloadServlet extends HttpServlet {
    public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        String password=req.getParameter("password");
```

```
if(!password.equals("abc")){
    res.setContentType("text/html;charset=gb2312");
    PrintWriter out = res.getWriter();
    out.println("<head><title>下载信息</title></head>");
    out.println("<h1 align='center'>验证码不正确</h1><hr>");
}else{
    long totalsize = 0;
    //得到物理路径
    File f = new File( this.getServletConfig().getServletContext()
        .getRealPath("/res/a.pdf") );
    long filelength = f.length();
    byte[] b = new byte[1024];
    FileInputStream fin = new FileInputStream(f);
    DataInputStream in = new DataInputStream(fin);

    res.setContentType("application/pdf");
    String filesize = Long.toString(filelength);
    res.setHeader("Content-Length", filesize);
    /**
    * 将文件内容输出到网络流中;
    */
    ServletOutputStream servletOut = res.getOutputStream();
    while(totalsize < filelength){
        totalsize += 1024;
        if(totalsize > filelength){
            byte[] leftpart = new byte[1024-(int)(totalsize-filelength)];
            in.readFully(leftpart); //读满 leftpart 数组才结束读取操作;
            servletOut.write(leftpart);
        }
        else{
            in.readFully(b);
            servletOut.write(b);
        }
    }
    servletOut.close();
}
```

```
}

```

运行效果如下：



图 3-10 输入密码以下载文件

由于密码正确，客户端浏览器显示如下：

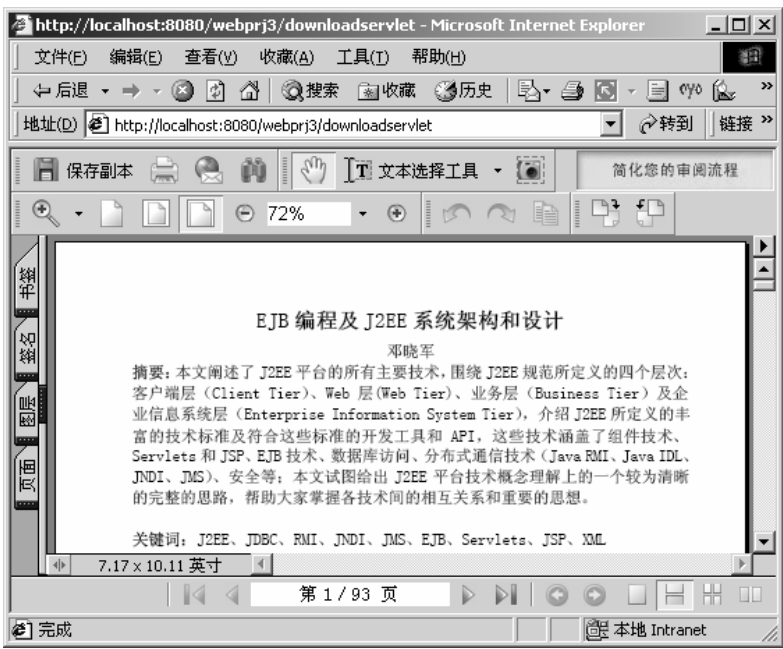


图 3-11 文件直接显示到浏览器中

解释：

- new File(this.getServletConfig().getServletContext().getRealPath("/res/a.pdf"))语句在当前站点根目录下的 res 子目录中寻找 a.pdf，然后其路径转为服务器上物理路径。
- ServletOutputStream servletOut=res.getOutputStream()语句用来得到字节输出流，输出目的地是客户端浏览器；
- res.setContentType("application/pdf")用以告诉浏览器：服务器准备发送 PDF 类型的二进制数据；
- res.setHeader("Content-Length",filesize)用以告诉浏览器：服务器准备发送多少字节的数据，以便浏览器作好接收准备；

- 要想发送其他类型的二进制数据，必须知道其 MIME 类型，可以用 `ServletContext.getMimeType()` 方法得到某文件的 MIME 类型；

3.7 web.xml 文档结构

到现在为止，我们配置 Servlet 时用到了 web.xml 文档，该文档存放在站点的 WEB-INF 目录下，在文档中除了可以配置 Servlet 外，还可以配置其他相关信息，另外该文档中的各子元素的顺序由相应 DTD 规定，顺序不能打乱，否则站点启动不了：

3.7.1 web.xml 的 DTD 规则

```
<!ELEMENT web-app (icon?, display-name?, description?, distributable?,
context-param*, filter*, filter-mapping*, listener*, servlet*,
servlet-mapping*, session-config?, mime-mapping*, welcome-file-list?,
error-page*, taglib*, resource-env-ref*, resource-ref*, security-constraint*,
login-config?, security-role*, env-entry*, ejb-ref*, ejb-local-ref*)>
```

3.7.2 web.xml 中重要配置

1、 默认(欢迎)文件的设置

```
<welcome-file-list>
<welcome-file>index.html</welcome-file>
<welcome-file>index.htm</welcome-file>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

2、 报错文件的设置

```
<error-page>
<error-code>404</error-code>
<location>/notFileFound.jsp</location>
</error-page>
<error-page>
<exception-type>java.lang.NullPointerException</exception-type>
<location>/null.jsp</location>
</error-page>
```

如果某文件资源没有找到，服务器要报 404 错误，按上述配置则会调用 `notFileFound.jsp`。如果执行的某个 JSP 文件产生 `NullPointerException`，则会调用 `null.jsp`

3、 会话超时的设置(设置 session 的过期时间，单位是分钟)

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

4、 Servlet 的设置

```
<servlet>
  <description>This is the description of my J2EE component</description>
```

```

<display-name>This is the display name of my J2EE component</display-name>
<servlet-name>InitParamsServlet</servlet-name>
<servlet-class>demo.InitParamsServlet</servlet-class>
<init-param>
    <param-name>p1</param-name>
    <param-value>1</param-value>
</init-param>
<init-param>
    <param-name>p2</param-name>
    <param-value>2</param-value>
</init-param>
<init-param>
    <param-name>p3</param-name>
    <param-value>3</param-value>
</init-param>
</servlet>

<servlet-mapping>
    <servlet-name>InitParamsServlet</servlet-name>
    <url-pattern>/initparamsservlet</url-pattern>
</servlet-mapping>

```

解释:

- <servlet>内的<servlet-name> 是一个逻辑名, 可以是任何有效的标识名;
- <init-param>是 Servlet 初始参数, 在 Servlet 的 init()方法中通过 getInitParameter("ip")取得, 返回 String 型数据;
- <servlet-mapping>内的<servlet-name>与<servlet>内的 <servlet-name>一一对应, 把客户端对/的请求对应到指定的 Servlet。
- <url-pattern>/initparamsservlet</url-pattern>指在 IE url 中的请求形式。这里的 / 是相对于当前的 web 目录的

小结

HttpServletRequest 接口的实现用于封装 HTTP 请求消息, 该接口定义了获得请求信息的方法。

HttpServletResponse 接口表示了一个 HTTP servlet 的应答, 通过它可以将数据或 Http 头信息发回客户端, 可以向客户端发送文本或二进制数据。

实现 ServletConfig 接口的类, 可以在 Servlet 实例化时向 Servlet 传递配置信息, 配置信息由 servlet 容器提供。ServletConfig 中还定义了获取 ServletContext 对象的方法。

Servlet 容器在启动时会加载 WEB 应用, 并为每一个 WEB 应用创建一个唯一的 ServletContext 对象, ServletContext 接口的对象, 封装了 Servlet 运行的环境信息, 并且可以存储 WEB 应用的全局

对象。

`RequestDispatcher` 接口提供了将请求转发给另外的资源,或在一个 `Servlet` 的响应中包含另外的资源的输出。

要向客户端输出二进制数据,要用到 `HttpServletResponse` 的 `getOutputStream` 方法,通过该方法可以得到一个字节输出流,输出目的地是客户端浏览器,然后向该流中输出二进制数据即可。

`web.xml` 文档存放在 站点的 `WEB-INF` 目录下,在文档中除了可以配置 `Servlet` 外,还可以配置其他相关信息。

课后练习

- 1、 简述 `HttpServletRequest`、`HttpServletResponse` 两个接口的功能及常用方法？
- 2、 `ServletConfig` 与 `ServletContext` 有何区别？
- 3、 请求重定向、请求转发、请求包含分别如何实现，三者有何区别？

第四章 会话管理

概要

通常，Web 环境中的会话是指用户 / 客户机的 Web 浏览器和一个特定的 Web 服务器之间的一组交互。会话从最初浏览器调用 Web 服务器的 URL 开始，到 Web 服务器结束会话，这个会话“超时”，或当用户关闭浏览器时结束。会话数据是指在永久保存之前用户提供的在多个页面上使用的信息。本章主要讨论如何使用 Servlet API 来完成各种各样的会话管理，主要内容包括：session、cookie、url 重写。

目标

- 会话管理的意义（理解）
- 使用 session（掌握）
- 使用 cookie（掌握）
- URL 重写（了解）

目录

- 4.1 用户验证
- 4.2 会话管理的意义
- 4.3 会话管理的分类
- 4.4 Cookie 管理的会话
- 4.5 URL 重写

4.1 用户验证

会话的基本概念

HTTP 是一种无状态协议，每当用户发出请求时，服务器就做出响应，客户端与服务器之间的联系是离散的、非连续的。当用户在同一网站的多个页面之间转换时，每一次请求之间都是独立的，根本无法知道是否是同一个客户。

状态管理机制克服了 HTTP 的一些限制并允许网络客户端及服务器端维护请求间的关系。在这种关系维持的期间叫做会话(session)。

在服务器上，通过为在站点上的用户创建一个会话对象保存该用户的信息。当用户第一次访问站点时，分配给用户一个会话对象和一个单独的会话 ID，这个 ID 是惟一的并且会存储到客户端。在接下来的请求中，会话 ID 标识了这个用户，会话 ID 作为请求的一部分发送给 Servlet，Servlet 能从会话对象中获取相关信息。

用户闲置了一段时间后，这个会话对象自动失效，会话对象会被删除，这段时间默认是 30 分钟，当然可以在系统管理工具中设置这个时间。也可以通过手工操作使会话失效。

会话处理流程

每当新用户请求一个 Servlet，服务器除了发回应答页面之外，它还要向浏览器发送一个特殊的数字。这个特殊的数字称为“会话标识符”，它是一个唯一的用户标识符。此后，HttpSession 对象就驻留在服务器内存之中，等待同一用户返回时再次调用它的方法。

在客户端，浏览器保存会话标识符，并在每一个后继请求中把这个会话标识符发送给服务器。会话标识符告诉 Servlet 容器当前请求不是用户发出的第一个请求，服务器以前已经为该用户创建了 HttpSession 对象。此时，Servlet 容器不再为用户创建新的 HttpSession 对象，而是寻找具有相同会话标识符的 HttpSession 对象，然后建立该 HttpSession 对象和当前请求的关联。

会话管理示例——用户验证

Servlet API 规范定义了一个简单的 HttpSession 接口，通过它我们可以方便地实现会话跟踪。HttpSession 接口提供了存储和返回数据的方法。在 HttpSession 中，数据都以“名字-值”对的形式保存。简而言之，HttpSession 接口提供了一种把对象保存到内存、在同一用户的后继请求中提取这些对象的标准办法。该接口中的主要方法如下：

方法	功能
setAttribute(String,Object)	将对象按指定键名存储到会话中
getAttribute(String)	返回会话中包含的指定名称的对象
getCreateTime()	返回该会话的创建时间
getId()	返回会话 ID
invalidate()	使会话失效
isNew()	判断是否为刚创建的新会话，即该会话还未使用过

表 4-1 HttpSession 接口中的主要方法

下面示例演示了 HttpSession 的用法，本示例中演示管理员登录后，如何在服务器 session 中存储其登录状态：

6) 管理员登录



图 4-1 管理员登录

登录成功后转到 adminervlet, 点击“退出管理系统”时, 删除后台的 session:

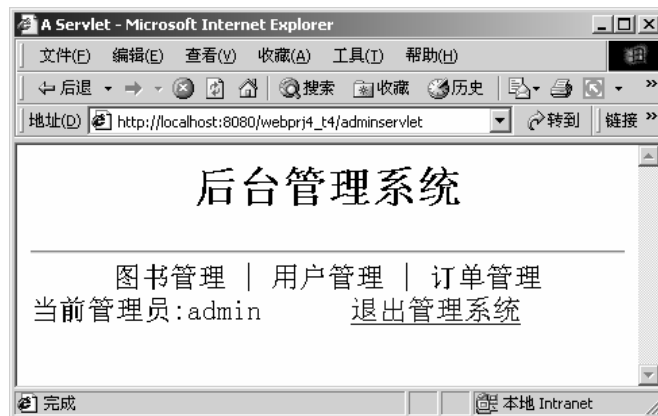


图 4-2 管理员登录成功

如果没有正常登录, 而是直接访问 adminervlet, 则会在后台判断 session 中是否存储了登录成功的标志, 如果尚未登录过, 则显示错误提示, 点击“确定”后转到登录页:



图 4-3 未登录而直接访问 adminervlet

程序源代码如下:

1) login.html

```
//login.html
<body>
管理员登录<hr>
    <form action="loginservlet" mehtod="post">
        管理员名称:<input type="text" name="username"><br>
        管理员密码:<input type="password" name="password"><br>
        <input type="submit" value=" 登录 ">
    </form>
</body>
```

2) LoginServlet.java, 负责接收登录信息, 判断不为空后转向 adminervlet 显示管理员可用功能:

```
//LoginServlet.java
package demo;

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

public class LoginServlet extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }

    /**
     *处理管理员登录业务
     */
    public void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        String username=request.getParameter("username");
        String password=request.getParameter("password");
```

```

//如果登录信息不全, 则返回登录页
if(username.equals("") || password.equals("")) {
    response.sendRedirect("login.html");
    return;
}

//将用户名保存到 session, 以便在其他管理模块中验证是否登录过
HttpSession session=request.getSession(true);
session.setAttribute("username",username);

//转到管理员页面
response.sendRedirect("adminservlet");
}

/**
 * Initialization of the servlet. <br>
 *
 * @throws ServletException if an error occure
 */
public void init() throws ServletException {
    // Put your code here
}
}

```

3) AdminServlet.java, 显示可用的管理员功能:

```

//AdminServlet.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

```

[illegible]

```

        out.close();
    }

    /**
     * The doPost method of the servlet. <br>
     */
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

        doGet(request,response);
    }
}

```

4) LogoutServlet.java, 用户选择“退出管理系统”时, 用于清除服务器中的 session 对象:

```

//LogoutServlet.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class LogoutServlet extends HttpServlet {

    /**
     * 处理退出管理系统的业务
     * 管理员退出系统时, 将 session 中的数据清除, 防止其他人借助该 session 继续使用管理系统
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        //得到 session

```

```
        HttpSession session=request.getSession(true);
        session.invalidate();
        response.sendRedirect("index.html");
    }

    /**
     * The doPost method of the servlet. <br>
     */
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

        doGet(request,response);
    }
}
```

通过 session，管理员只需一次登录，即可使用各管理模块的功能，而且通过 session 还可以判断管理员有没有登录过，只需在各模块中加上对 session 中登录数据的判断即可。

4.2 会话管理的意义

在 Web 服务器端编程中，会话状态管理是一个经常必须考虑的重要问题。通过会话管理可以跟踪浏览者在整个网站中的活动，对他们的身份进行自动的识别。

通过会话管理，网站能够为浏览者提供一系列的方便，例如在线交易过程中标识用户身份、安全要求不高的场合避免用户重复输入名字和密码、门户网站的主页定制、有针对性地投放广告，等等。

使用会话最典型的例子是网上商店，当客户在网上商店向他们的购物车中添加商品时，服务器如何才能知道购物车中已经有什么商品了呢？当客户决定结帐时，服务器如何能够确定客户创建的是哪个购物车呢？系统必须保存每一个用户购物车里商品的列表。为了实现这一功能，服务器必须能够区分不同的客户，而且还要有一种为每一个客户存储数据的方法。

还有一些网站后台管理程序，后台管理程序通常分为多个模块，通常管理员只需登录一次即可访问所有的管理模块，这就需要在服务器上存储管理员是否登录的状态。

以上这些完全可以通过会话管理来解决。

4.3 会话管理的分类

由于 Web 服务器使用的协议 HTTP 是一种无状态的协议，因此要维护一系列请求之间的状态就需要使用其它的附加手段，随着技术的发展，人们开发了各种各样的会话状态跟踪机制：

- Cookie

Cookie 是服务器发送给浏览器的体积很小的纯文本信息，用户以后访问同一个 Web 服务器时浏览器会把它们原样发送给服务器。通过让服务器读取它原先保存到客户端的信息，客户每次访问服务

器时，Cookie 自动发送给服务器，这使得它可以作为一种会话跟踪的解决方案。

使用 Cookie 的好处是它储存数据很直观。

Cookie 的缺点是它可以用于在更长时间内跟踪用户，甚至可以用来跟踪某个用户向站点发送的每一个请求，因此有人担心自己的隐私问题而关闭了 Cookie，一些老的浏览器也不支持 cookie。

● URL 重写

- 1、就是把 session id 直接附加在 URL 路径的后面。如以下的 url：`会员专区`，在 url 中包含了 sessionid 数据。
- 2、这种方法的缺点是：对于大量的数据，URL 会变得很长而失去控制；在某些环境下，URL 的字符串长度有一定的限制；数据保密问题，你可能不想让旁边的人或者可以使用同一个计算机的看到你的会话数据。

● 表单隐藏字段

这种方法目前已极少使用，该方法通过修改表单，添加一个隐藏字段，以便在表单提交时能够把 session id 传递回服务器。比如下面的表单：

```
<form action="loginServlet" method="post">
```

```
<input type="text" name="username">
```

```
</form>
```

在被传递给客户端之前将被改写成

```
<form action="loginServlet" method="post">
```

```
<input type="hidden" name="jsessionid" value="ByOKx9zW145788764">
```

```
<input type="text" name="username">
```

```
</form>
```

Session 与 Cookie 的关系：Session 要在客户端存储 sessionid，通常上借助 cookie 将 sessionid 存储到客户硬盘上。

HttpSession 接口封装了以上会话跟踪机制，我们使用 HttpSession 接口创建的会话本身是保存在服务器上的。每次向服务器发送请求时，客户端向服务器发送 cookie 中的会话 ID，如果浏览器不允许使用 cookie，服务器会自动将会话 ID 写入 URL，即 HttpSession 接口会根据具体情况自动选择合适的会话跟踪机制，使用 HttpSession 接口，Servlet 程序员可以从底层细节中解脱。

4.4 Cookie 管理的会话

Cookies 是一种能够让网站服务器把少量数据储存在客户端的硬盘或内存，或是从客户端的硬盘读取数据的一种技术。Cookies 是当客户浏览某网站时，由 Web 服务器置于你硬盘上的一个非常小的文本文件，它可以记录你的用户 ID、密码、浏览过的网页、停留的时间等信息。Cookies 只能由提供它的服务器来读取，当你再次来到该网站时，网站通过读取 Cookies，得知你的相关信息，就可以做出相应的动作，如在页面显示欢迎语，或者让客户不用输入 ID、密码就直接登录等等。

ServletAPI 中提供了创建、使用 Cookie 的专门 API——`javax.servlet.http.Cookie` 类。该类的用法如下：

要把 Cookie 发送到客户端，先要调用 `new Cookie(name,value)` 用合适的名字和值创建一个或多个 Cookie，通过 `cookie.setXXX` 设置各种属性，通过 `response.addCookie(cookie)` 把 cookie 写入客户端。

要从客户端读入 Cookie，应该调用 `request.getCookies()`，`getCookies()` 方法返回一个 Cookie 对象的数组。然后用循环访问该数组的各个元素寻找指定名字的 Cookie，然后对该 Cookie 调用 `getValue` 方法取得与指定名字关联的值。

`javax.servlet.http.Cookie` 类的主要方法如下：

方法	功能
<code>Cookie(String,String)</code>	构造 Cookie 对象，第一个参数为键名，第二个参数为键值
<code>setMaxAge(long)</code>	设置 Cookie 过期之前的时间，以秒计。如果不设置该值，则 Cookie 只在当前会话内有效，即在用户关闭浏览器之前有效，而且这些 Cookie 不会保存到磁盘上。
<code>setComment(String)</code>	设置 Cookie 的注释
<code>getName()</code>	获取 Cookie 的名字
<code>getValue</code>	获取 Cookie 的值

表 4-2 Cookie 的主要方法

示例—会员登录

该示例中，会员登录系统后，系统会将登录信息写到客户端的 Cookie 中，在以后客户再次来访时就不必再次登录了，而是直接可以进入系统：

1、 管理员登录

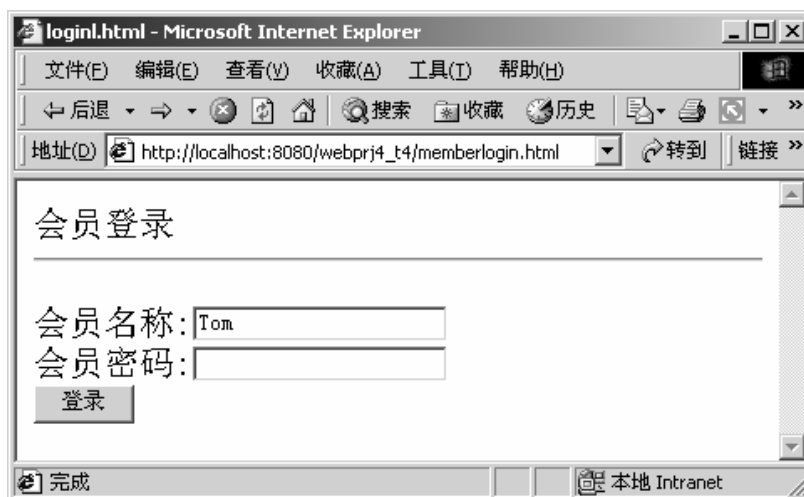


图 4-4 会员登录

2、 管理员登录



图 4-5 会员登录成功

3、以后会员可以直接进入系统，而不必登录

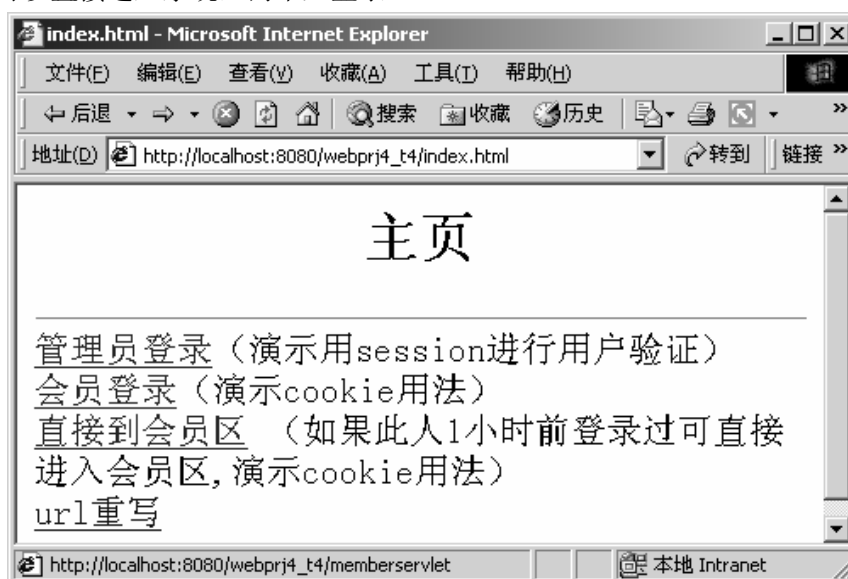


图 4-6 会员可以直接进入系统

程序源代码如下：

7) 会员登录页—memberlogin.html

```
//memberlogin.html
<body>
会员登录<hr>
  <form action="memberloginservlet" mehtod="post">
    会员名称:<input type="text" name="username"><br>
    会员密码:<input type="password" name="password"><br>
    <input type="submit" value=" 登录 ">
  </form>
</body>
```

处理会员登录的 MemberloginServlet，正确登录后转到 memberservlet 显示会员专区：

```
//MemberloginServlet.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.*;

public class MemberloginServlet extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request,response);
    }

    /**
     * 处理会员登录
     */
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {

        String username=request.getParameter("username");
        String password=request.getParameter("password");

        //如果登录信息不全，则返回登录页
        if(username.equals("") || password.equals("") ){
            response.sendRedirect("login.html");
            return;
        }

        //将用户名保存到 cookie，以便在其他管理模块中验证是否登录过
        Cookie cookie=new Cookie("username",username);
```

```
        //设置 cookie 在一个小时之内有效, 无论用户是否退出浏览器
        cookie.setMaxAge(3600); //单位是秒
        response.addCookie(cookie);

        //转到会员页面
        response.sendRedirect("memberservlet");
    }
}
```

会员页—MemberServlet

```
//MemberServlet.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.*;

public class MemberServlet extends HttpServlet {

    /**
     * 会员页面
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();

        //得到 Cookie 集合
        Cookie cookies[] = request.getCookies();
        boolean flag=false;
        String username="";
```

[illegible]

```

/**
 * The doPost method of the servlet. <br>
 */
public void doPost(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {

    doGet(request,response);
}
}

```

4.5 URL 重写

就是把 session id 直接附加在 URL 路径的后面。如以下的 url：`会员专区`，在 url 中包含了 sessionid 数据。在 Servlet 中提供了将 sessionid 附加到 url 的方法，`response.encodeURL()` 方法会根据需要在 url 中附加 sessionid，示例代码如下：

```

package demo;
import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class UrlrewriteServlet extends HttpServlet {
    /**
     * The doGet method of the servlet. <br>
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();

        //如果浏览器不支持 cookie,则下面语句会在生成的 url 中附加 sessionid;
        String url=response.encodeURL("urlrewriteservlet");
    }
}

```

```
        out.println("<HTML>");
        out.println("  <HEAD><TITLE>A Servlet</TITLE></HEAD>");
        out.println("  <BODY>");
        out.print("<h2>url 重写演示</h2>");

        out.print("<a href='" + url + "'>xxx</a><br>");
        //以下手工生成 url 重写形式
        out.print("<a href='urlrewriteservlet;jsessionid="
+ request.getSession(true).getId() + "'>xxx</a>");
        out.print(" <br>sessionid:"+ request.getSession(true).getId() );
        out.println("  </BODY>");
        out.println("</HTML>");
        out.flush();
        out.close();
    }

    /**
     * The doPost method of the servlet. <br>
     */
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        doGet(request,response);
    }
}
```

小结

HTTP 是一种无状态协议，状态管理机制克服了 HTTP 的一些限制并允许网络客户端及服务器端维护请求间的关系。在这种关系维持的期间叫做会话(session)。

Servlet API 规范定义了一个简单的 HttpSession 接口，通过它我们可以方便地实现会话跟踪。HttpSession 接口提供了存储和返回数据的方法。在 HttpSession 中，数据都以“名字-值”对的形式保存。

为保持多个请求间的状态，出现了各种各样的会话状态跟踪机制：Cookie、URL 重写、表单隐藏字段，HttpSession 接口封装了以上会话跟踪机制。

Cookie 是服务器发送给浏览器的体积很小的纯文本信息，用户以后访问同一个 Web 服务器时浏览器会把它们原样发送给服务器。ServletAPI 中提供了创建、使用 Cookie 的专门 API——javax.servlet.http.Cookie 类。

课后练习

- 1、 Session 与 Cookie 之间有何关系？
- 2、 session 中可以存储对象吗？Cookie 中呢？

第五章 线程安全

概要

Servlet 由于其以多线程方式运行而具有很高的执行效率，但同时由于 Servlet 默认是以多线程模式执行的，所以，在编写代码时需要非常细致地考虑多线程的安全性问题。本章主要介绍 Servlet 的多线程机制、引起 Servlet 线程不安全的原因、保证 Servlet 线程安全的方案。

目标

- Servlet 线程安全（理解）
- 本地变量（掌握）
- 同步（掌握）
- 三个存储对象(request/session/servletcontext)（掌握）
- Servlet 监听器

目录

- 5.1 Servlet 的容器
- 5.2 线程安全的意义
- 5.3 本地变量
- 5.4 同步
- 5.5 借助 Servlet 对象保证线程安全

5.1 Servlet 的容器

理解 Servlet 容器的工作机理对开发或维护基于 Servlet 的 Web 应用是很重要的。Servlet 体系结构是建立在 Java 多线程机制之上的，Servlet 的生命周期是由 Servlet 容器负责的。

Servlet 容器工作机制：当客户端第一次请求某个 Servlet 时，Servlet 容器将会根据 web.xml 配置文件实例化这个 Servlet 类。当有第二个以上的客户端请求该 Servlet 时，不会再实例化该 Servlet 类，即内存中只有一个该 Servlet 类的对象，Servlet 容器让该实例服务于所有的请求，对于客户端同时请求一个 servlet，他们是被并发的处理的，并不是等上一个请求处理完成再处理下一个。如果两个请求同时到达，那么他们处理完成的时间也是差不多的，即服务器中可能同时有多个线程在使用这个对象。如图 5-1 所示：

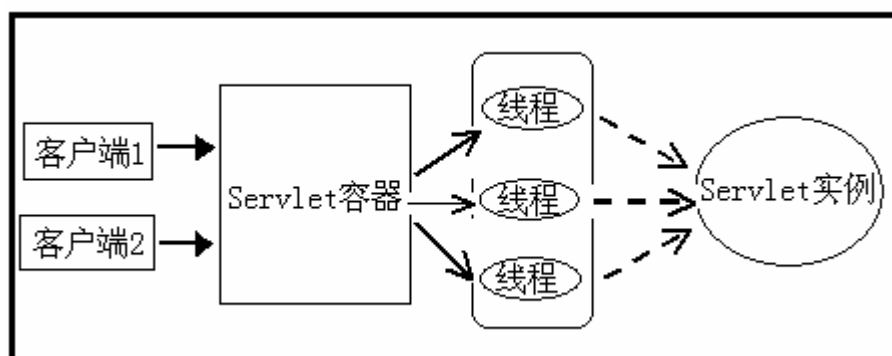


图 5-1 Servlet 线程模型

由于服务器上只为同一 Servlet 类生成一个实例，所以有效节省了服务器资源耗费，但是，当两个或多个线程同时访问同一个 Servlet 时，可能会发生多个线程同时访问同一资源的情况，数据可能会变得不一致。所以在用 Servlet 构建的 Web 应用时如果不注意线程安全的问题，会使所写的 Servlet 程序有难以发现的错误。

以下示例演示了 Servlet 的多线程运行模式，在此例中，我们在 Servlet 中声明了一个 int 型的实例变量，然后打开多个浏览器模拟多个请求进行访问，观察其运行结果：

1、 TestServlet:

```
//TestServlet.java
package demo;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 *演示：Servlet 的多线程运行模式；
 */
```

```
public class TestServlet extends HttpServlet {
    //声明实例变量
    int i;
    /**
     * The doGet method of the servlet. <br>
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }
    /**
     * 修改实例变量
     */
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gbk");
        PrintWriter out = response.getWriter();
        ++i;//增加 i
        out.println("<HTML>");
        out.println("  <HEAD><TITLE>A Servlet</TITLE></HEAD>");
        out.println("  <BODY>");
        //显示实例变量的值
        out.print("i=" + i);
        out.println("  </BODY>");
        out.println("</HTML>");
        out.flush();
        out.close();
    }
}
```

2、 访问该 Servlet:

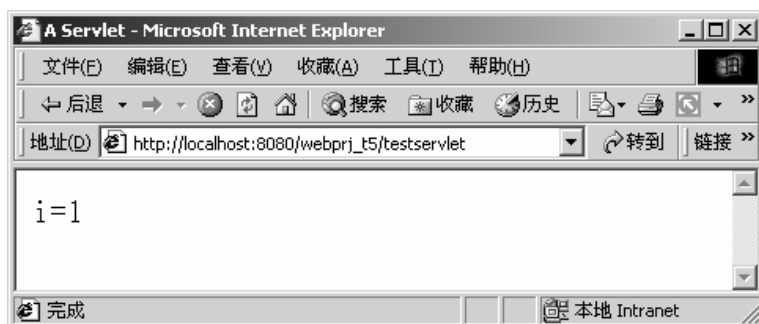


图 5-2 访问 TestServlet

3、再打开一个新的浏览器访问该 Servlet:

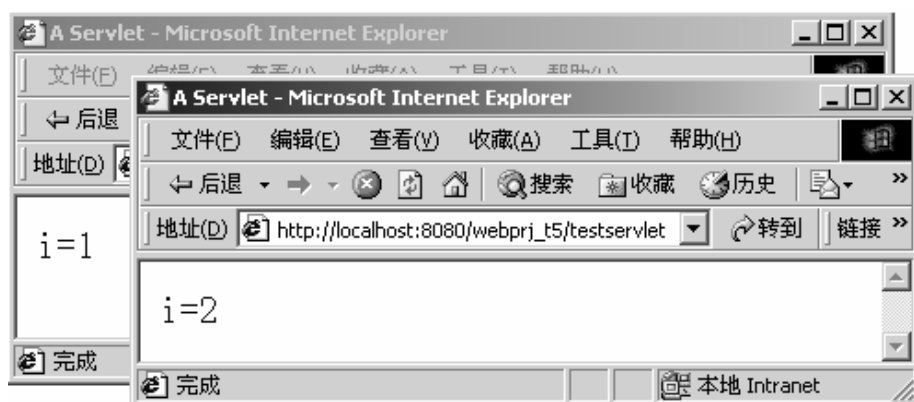


图 5-3 再次访问 TestServlet

通过结果可以观察到：两个浏览器对同一个 Servlet 进行访问，服务器内存中只生成了该 Servlet 的一个实例，两个用户是在访问同一个 Servlet 对象，所以第二个用户访问时，实例变量 *i* 的值会显示为 2。如果第一个访问者将 *i* 的值加完后还没来得及使用，第二个用户又修改了数据，则第一个用户的业务逻辑将发生严重问题。当然适当的时候，我们也可以利用 Servlet 的这种多线程特征，比如，该例中，可以统计某个 Servlet 被访问的次数。

5.2 线程安全的意义

刚才分析了 Servlet 容器默认情况下，为提高运行效率，所有 Servlet 是以多线程模式执行的，编写 Servlet/JSP 程序时如果没有注意到多线程的问题，这往往造成编写的程序在少量用户访问时没有任何问题，而在并发用户上升到一定值时，就会经常出现一些莫明其妙的问题，对于这类随机性的问题调试难度也很大。

为开发线程安全的 Servlet，可以通过以下手段来实现：

1、让 Servlet 实现 SingleThreadModel 接口

该接口中没有任何方法，是一个标识接口，实现了该接口的 Servlet 不会被以多线程模式运行，而是生成它的多个实例，每个请求对应一个实例。

多线程和单线程 servlet 具体区别：

- ◆ 多线程情况下每个线程会保存实例中的局部变量的一个副本，对局部变量的修改只会影响自己的副本不会影响别的线程，所以局部变量是安全的。但是对于实例变量，是属于所有线程的。对他的修改会影响到其他的线程。
- ◆ 而对于单线程 servlet，会有多个实例，这些实例的变量都是属于自己的，当然是线程安

全的。

◆ 两种情况下 `static` 变量都不是线程安全的。

- 1、 对于不想被其他线程共同访问的变量要写成局部变量，以保证安全。
- 2、 同步 `Servlet` 的 `service()` 方法，此时，某个请求正在访问 `service()` 方法时，其他请求要等待。

通过以上方案，可以有效保护 **WEB** 应用中的数据不被意外修改，保证业务逻辑的正常处理，以上三种方案中的第一种方案会大幅增加 `web` 服务器中 `Servlet` 实例的个数，会导致严重的性能问题。

5.3 本地变量

本地变量指作用在某个程序块内，而其它程序块中的代码不能访问的数据项。例如，定义在某个 `Java` 方法中的变量就是本地变量，不能在该方法外使用。多个线程访问同一个对象时，对象的实例变量保存在对象中，所有线程共享一份，而定义在方法内部的局部变量会在每个线程中保存一份副本。

`Servlet` 的线程安全问题主要是由于实例变量使用不当而引起的，由于线程之间无法相互直接访问局部变量，所以将变量声明为局部变量（对线程而言就是本地变量，因为线程将该局部变量副本复制到了每个线程自己的空间），可以有效避免 `Servlet` 多线程引发的问题。

以下示例，演示本地变量的用法，在本例中同时还使用了实例变量，以演示两者的区别：

- 1、 `index.html`，该页面中提供用户登录的表单：

```
//index.html
.....
测试本地变量与实例变量:<br>
<form action="loginservlet" method="post">
    用户名:<input name="username" size="10">
    <input type="submit" value="进入">
</form>
.....
```

- 2、 `LoginServlet.java`，在该 `Servlet` 中接收用户名，并将名字同时存储到该 `Servlet` 的实例变量和 `doPost()` 的本地变量中：

```
//LoginServlet.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class LoginServlet extends HttpServlet {
```

```
String username;

/**
 * The doGet method of the servlet. <br>
 */
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    doPost(request, response);
}

/**
 * The doPost method of the servlet. <br>
 */
public void doPost(HttpServletRequest request, HttpServletResponse
response)
    throws ServletException, IOException {
    String username2;
    response.setContentType("text/html;charset=gbk");
    PrintWriter out = response.getWriter();
    request.setCharacterEncoding("gbk");

    //将用户名存储到本地变量
    username2=request.getParameter("username");
    //将用户名存储到实例变量中
    this.username=username2;
    //以下语句延迟三秒钟后，给用户输出响应，以模拟服务器中访问量大的情况；
    try{Thread.sleep(3000);}catch(Exception e){}

    out.println("<HTML>");
    out.println("  <HEAD><TITLE>A Servlet</TITLE></HEAD>");
    out.println("  <BODY>");
    out.print("你好: " + username + ",欢迎光临!");
    out.print("username2: " + username2 );
    out.println("  </BODY>");
    out.println("</HTML>");
    out.flush();
    out.close();
}
```



```
}

```

3、部署该 WEB，同时打开两个浏览器，运行 index.html:

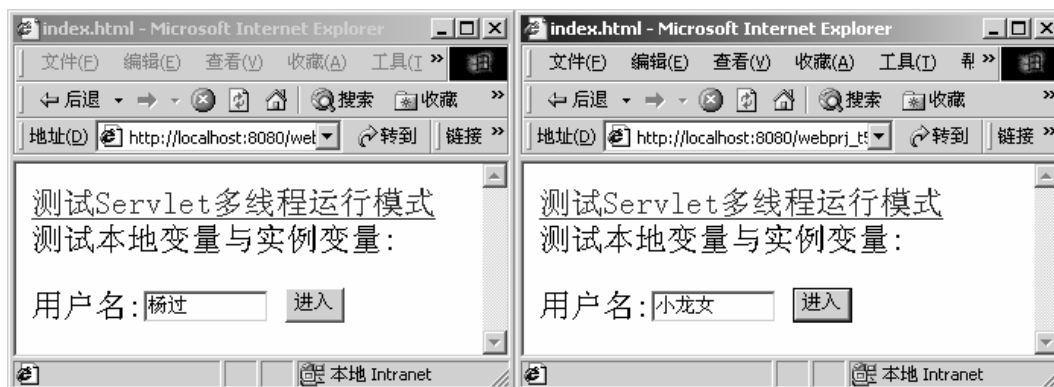


图 5-4 访问 index.html

4、点击两个浏览器中的“进入”按钮，两次点击间隙时间不要超过 3 秒钟，结果如下，会发现两个浏览器中显示的实例变量值都是“小龙女”，本地变量值正常显示:



图 5-5 两个用户登录后的结果

分析:

- ◆ 在 LoginServlet 中声明了一个实例变量 username，在 doPost()方法内部声明了一个本地变量，在接收到用户名后，同时将该名称保存到了这两个变量中。
- ◆ “杨过”注册后，由于服务器上 Thread.sleep(3000)语句导致 Servlet 三秒后才能将结果输出到客户浏览器中，而三秒钟内“小龙女”也进行注册，服务器为该客户开一个新的线程去访问同一个 Servlet 实例，此时“小龙女”的名字将覆盖实例变量 username 中的“杨过”；
- ◆ 两个线程中都保存了 doPost()方法中局部变量 username2 的副本，所以 username2 每个线程中都有一份，值不会被覆盖。
- ◆ 内存模型见图 5-6 所示:

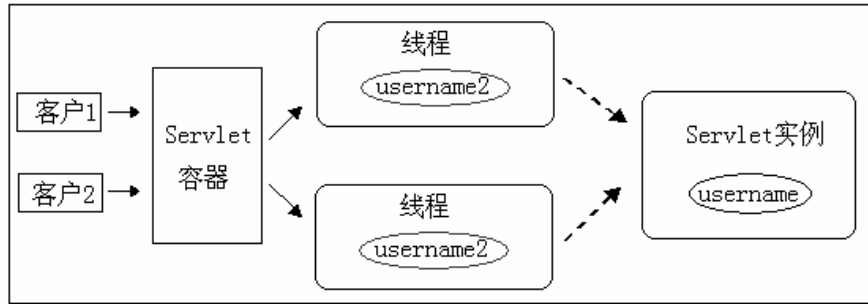


图 5-6 两个用户同时访问同一 Servlet 时本地变量与实例变量的情况

5.4 同步

通过本地变量可以使线程各自都有一份变量的副本，数据访问不会发生冲突。除此之外，通过同步操作也可以避免访问实例变量的冲突。

图 5-6 中的 Servlet 中的 `username` 为实例变量，多个线程访问时出现数据紊乱的原因是：第一个线程对实例的访问未完全结束时，第二个线程也可以访问该实例。

通过同步操作可以保证一个实例只能被一个线程访问，其他线程要访问必须等当前线程访问结束后才能取得对实例的访问权。

下面示例演示如何在 Servlet 中使用同步机制：

1) `index.html`：

```
//index.html
.....
测试同步操作:<br>
<form action="loginservlet2" method="post">
    用户名:<input name="username" size="10">
    <input type="submit" value="进入">
</form>
.....
```

2) `LoginServlet2.java`，该 Servlet 中将 `doPost()` 方法声明成了同步方法，则该方法同一时刻只能被一个线程访问，将一个方法变为同步方法很简单，只要在该方法前加关键字 `synchronized`；

```
//LoginServlet2.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```
public class LoginServlet2 extends HttpServlet {
    String username;

    /**
     * The doGet method of the servlet. <br>
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        doPost(request, response);
    }

    /**
     * doPost()方法声明成了同步方法;
     */
    synchronized public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        String username2;
        response.setContentType("text/html;charset=gbk");
        PrintWriter out = response.getWriter();
        request.setCharacterEncoding("gbk");

        //将用户名存储到本地变量
        username2=request.getParameter("username");
        //将用户名存储到实例变量中
        this.username=username2;
        //以下语句延迟三秒钟后，给用户输出响应，以模拟服务器中访问量大的情况；
        try {
            Thread.sleep(3000);
        }
        catch(Exception e) {
        }
        out.println("<HTML>");
        out.println("<HEAD><TITLE>A Servlet</TITLE></HEAD>");
        out.println("<BODY>");
        out.print("你好: " + username + ",欢迎光临!<br>");
        out.print("username2: " + username2 );
    }
}
```

```

        out.println("    </BODY>");
        out.println("</HTML>");
        out.flush();
        out.close();
    }
}

```

3) 运行结果如下:



图 5-7 访问 index.html



图 5-8 显示实例变量及本地变量(使用了同步)

分析:

- ◆ “杨过”注册后，由于服务器上 Thread.sleep(3000)语句导致 Servlet 三秒后才能将结果输出到客户浏览器中，三秒钟内“小龙女”进行注册时，服务器为该客户开一个新的线程去访问同一个 Servlet 实例，但此时该实例被“杨过”占用，“小龙女”必须等待杨过完成访问后才能进入 doPost()方法进行访问；
- ◆ 3 秒钟后“杨过”输出实例变量的值，由于同步的原因，该值没有被其他线程修改；
- ◆ 同步方法会降低 Servlet 的整体工作效率，因为服务器中同一时刻只能为一个用户服务，所以不提倡使用同步方法来解决 Servlet 线程安全问题，最佳方案还是在 Servlet 中使用本地变量。

5.5 借助 Servlet 对象保证线程安全

5.5.1 用 Servlet 对象存储传递数据

前面分析了三种保证 Servlet 线程安全的方案，这些方案都有潜在的缺陷：

- 1、本地变量的方案可以保证数据不被意外修改，但由于是局部变量，所以同时也导致了用户访问完后数据会立即丢失，而不能保存在服务器上供后续的业务处理中使用。
 - 2、同步 `doXXX()` 方法后，导致一个 `Servlet` 同一时刻只能被一个客户访问，增加了访问瓶颈、限制了网站流量；
 - 3、让 `Servlet` 实现 `SingleThreadModel` 接口会导致服务器中产生大量 `Servlet` 实例，消耗资源过多；
- 综合比较，在不需要长时间保存数据的场合可以使用以上“本地变量”的解决方案，而另两种方案通常不必考虑。

在 Web 应用中更多的是要跨多个页面、多个 `Servlet` 访问同一个数据，例如购物车系统中，在多个 `Servlet` 中都需要访问同一个用户信息。在 WEB 应用中更多的是使用以下 `Servlet` 对象来存储数据：

- 1、`HttpServletRequest`——存储在该对象中的数据可以跨两个 WEB 资源进行存取，前提是需要将当前请求转发到第二个 WEB 资源中。通过 `Request` 对象存储、传递数据的简单示例如下：

```
//servlet1
request.setAttribute("nick","张三");
RequestDispatch dispatcher=this.getServletConfig().getServletContext().
getRequestDispatcher("/servlet2");
dispatcher.forward(request,response);
```

以上代码第一行将昵称“张三”存储到 `request` 对象中，后两行代码将当前请求转发到 `servlet2`，这样在 `servlet2` 中也可以访问到昵称“张三”。

- 2、`HttpSession`——在用户一次会话中，可以在任何 WEB 资源（`servlet/jsp`）中访问存储在该对象中的数据。当需要存储用户会话级的数据时（如：聊天室中的昵称），可以使用该对象。
- 3、`ServletContext`——该对象中存储的数据，是网站的全局数据。可以被任何用户、在任何 WEB 资源中访问到。

5.5.2 Servlet 监听器

`Servlet` 规范提供了事件监听机制，可以监听会话(`Session`)、`ServletContext` 的各种事件。如：会话开始/结束、添加/删除属性等。Web 应用程序启动时，Web 容器注册它的事件监听器，并适时调用响应事件的方法。

可以将一个或多个事件监听器加入 `web.xml` 描述文件中，Web 容器装载 WEB 应用后，会查看该文件中的全部配置找出监听器定义，将其注册。配置 `Servlet` 监听器的示例如下：

```
<listener>
<listener-class>demo.MySessionListener</listener-class>
</listener>
```

要在 Web 应用中使用监听器，首先要开发监听器类，该类必须实现指定监听器接口。`Servlet` 监听器常用的监听接口：

- 1、`HttpSessionListener`

监听 `HttpSession` 的操作。当创建一个 `Session` 时，激发 `session Created(SessionEvent se)` 方法；当销毁一个 `Session` 时，激发 `sessionDestroyed (HttpSessionEvent se)` 方法。

- 2、`HttpSessionAttributeListener`

监听 `HttpSession` 中的属性的操作。当在 `Session` 增加一个属性时，激发

attributeAdded(HttpSessionBindingEvent se) 方法；当在 Session 删除一个属性时，激发 attributeRemoved(HttpSessionBindingEvent se) 方法；当在 Session 属性被重新设置时，激发 attributeReplaced(HttpSessionBindingEvent se) 方法。

3、 ServletContextListener

监听 ServletContext，当创建 ServletContext 时，激发 contextInitialized(ServletContextEvent sce) 方法；当销毁 ServletContext 时，激发 contextDestroyed(ServletContextEvent sce) 方法。

4、 ServletContextAttributeListener

监听对 ServletContext 属性的操作，比如增加/删除/修改变属性值。

下面通过示例演示 Servlet 监听器的开发和使用方法，该示例描述：在一个用户登录后，将其昵称存储在 session 中，然后设置 ServletContext 中的一个计数器值，提供了监听器对 session、servletcontext 对象进行了监听：

1) 登录页面

```
.....
<form action="counterservlet" method="post">
昵称:<input type="text" name="nick" size="10">
<input type="submit" value=" 进入 ">
</form>
.....
```

2) CounterServlet 主要代码如下：

```
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    //得到 ServletContext 对象，并设置计数器值
    ServletContext application=this.getServletConfig().getServletContext();
    application.setAttribute("online_counter","1000");

    //得到 session 对象，并将昵称存储到 session
    HttpSession session=request.getSession(true);
    String nick=request.getParameter("nick");
    session.setAttribute("nick",nick);

    response.setContentType("text/html;charset=gbk");
    PrintWriter out = response.getWriter();
    out.println("<b" + nick + "</b>登录.");
    out.println("已经将在线人数设置为: 1000");
    out.println("<br><br><a href=session servlet1>销毁 session</a>");
}
```

以上 servlet 最后一行，在网页上显示超链接，点击后运行 sessionservlet，在该 servlet 中通过代码

销毁 session 对象，以观察 session 监听的作用。

3) sessionservlet 主要代码如下：

```
/**
 * 测试 HttpSessionListener 监听器
 * 当创建一个 Session 时，激发 session Created(SessionEvent se)方法；
 * 当销毁一个 Session 时，激发 sessionDestroyed(HttpSessionEvent se)方法
 */
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    //人为销毁 session 对象；
    request.getSession().invalidate();
    response.setContentType("text/html;charset=gbk");
    PrintWriter out = response.getWriter();
    out.println("session 对象已销毁。");
}
```

4) Session 监听器如下：

```
package demo;

import javax.servlet.*;
import javax.servlet.http.*;

public class MySessionListener
    implements HttpSessionListener, HttpSessionAttributeListener {
    public void sessionCreated(HttpSessionEvent hse) {
        //增加在线人数；
        HttpSession session=hse.getSession();
        ServletContext application=session.getServletContext();
        //得到在线人数计数器
        Object obj=application.getAttribute("online_counter");
        int count=0;
        //在线人数不存在，则设置成 1
        if(obj==null){
            application.setAttribute("online_counter","1");
        }else{
            //在线人数增 1，再存回
            String oldcount=(String)obj ;
            System.out.println("目前在线人数：" + oldcount);
            count=Integer.parseInt( oldcount ) + 1;
        }
    }
}
```

```

        application.setAttribute("online_counter",""+count);
    }
    System.out.println("新建一个 session,在线人数:" + count);
}

public void sessionDestroyed(HttpSessionEvent hse) {
    //增加在线人数;
    HttpSession session=hse.getSession();
    ServletContext application=session.getServletContext();
    Object obj=application.getAttribute("online_counter");
    //在线人数减1,再存回
    int count=Integer.parseInt( (String)obj ) - 1;
    application.setAttribute("online_counter",""+count);
    System.out.println("销毁一个 session,在线人数:" + count);
}

//调用 session 的 setAttribute() 时运行该方法
public void attributeAdded(HttpSessionBindingEvent se){
}

//调用 session 的 removeAttribute() 时运行该方法
public void attributeRemoved(HttpSessionBindingEvent se){
}

//调用 session 的 setAttribute() 修改已存在的值时运行该方法(即覆盖)
public void attributeReplaced(HttpSessionBindingEvent se){
}
}

```

5) ServletContext 监听器如下,该监听器实现了两个接口,各接口功能见前所述:

```

package demo;

import javax.servlet.*;
import javax.servlet.http.*;

public class MyContextListener
    implements ServletContextListener,ServletContextAttributeListener {
    public void contextInitialized(ServletContextEvent e) {
        System.out.println("contextInitialized.....");
    }
}

```



```

        ServletContext application=e.getServletContext();
        application.setAttribute("online_counter","0");
        System.out.println("已经将在线人数设置为: 0");
    }
    public void contextDestroyed(ServletContextEvent e) {
        System.out.println("contextDestroyed.....");
    }
    //向 context 中添加新属性时运行该方法
    public void attributeAdded(ServletContextAttributeEvent scab){
        System.out.println("context attributeAdded.....");
    }
    //属性被移除时运行该方法
    public void attributeRemoved(ServletContextAttributeEvent scab){
        System.out.println("context attributeRemoved.....");
    }
    //调用 context 的 setAttribute() 修改已存在的值时运行该方法(即覆盖)
    public void attributeReplaced(ServletContextAttributeEvent scab){
    }
}

```

6) web.xml 中对监听器的配置如下, 其中配置了两个监听器:

```

<listener>
    <listener-class>demo.MySessionListener</listener-class>
</listener>
<listener>
    <listener-class>demo.MyContextListener</listener-class>
</listener>

```

7) 运行效果如图 5-9:

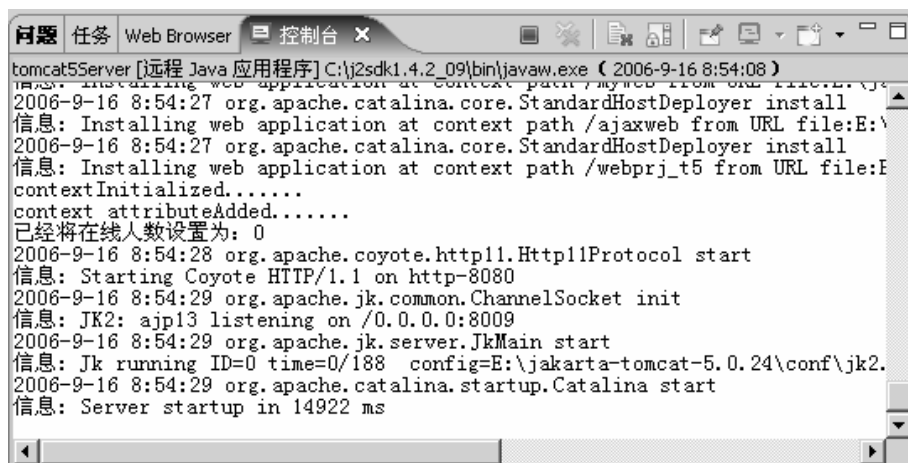


图 5-9 监听 ServletContext

图中为 tomcat 控制台的输出，从图中可以看出，启动 web 应用时，ServletContext 监听器中的 contextInitialized()、attributeAdded()两个方法自动运行，实现了监听功能。

填写昵称登录后，昵称被存储到 session 中，Session 监听器的方法自动运行，监听到了 session 的创建事件。效果如下，注意最后两行

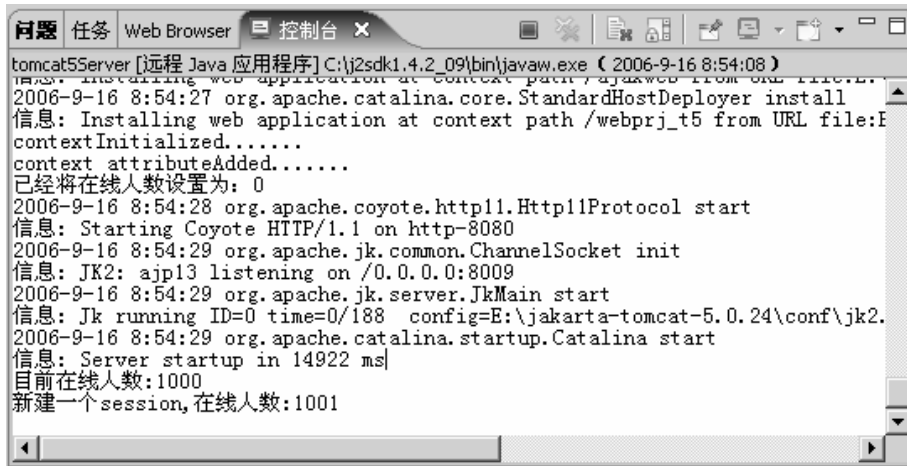


图 5-10 监听 session

运行 sessionservlet 销毁 session 对象后，tomcat 控制台输出效果如下，新加了最后一行内容：



图 5-11 监听 session

小结

服务器只为同一 Servlet 类生成一个实例，所以有效节省了服务器资源耗费，但是，当两个或多个线程同时访问同一个 Servlet 时，可能会发生多个线程同时访问同一资源的情况，数据可能会变得不一致。

要有效保护 WEB 应用中的数据不被意外修改，可以使用以下三种方案：让 Servlet 实现 SingleThreadModel 接口、使用本地变量、同步 Servlet 的方法。

本地变量指作用在某个程序块内，而其它程序块中的代码不能访问的数据项。这些变量对线程而言就是本地变量，因为线程将该局部变量副本复制到了每个线程自己的空间。

通过同步操作可以保证一个实例只能被一个线程访问，其他线程要访问必须等当前线程访问结束后才能取得对实例的访问权。

Servlet 规范提供了事件监听机制，可以监听会话(Session)、ServletContext 的各种事件。

课后练习

- 1、 简述 Servlet 容器运行一个 Servlet 的机制？
- 2、 本地变量是如何解决 Servlet 线程安全问题的？
- 3、 简述同步操作如何解决 Servlet 线程安全？
- 4、 简述三个 Servlet 对象存储和传递数据的特征。

第六章 JSP（一）

概要

JSP 是 Java Server Page 的缩写，它扩展了 Servlet，是一种基于文本的、以显示为中心的开发技术，其目的是简化动态网站的开发工作，通过 Java Servlet 与 JSP 结合的方式可以将网站的逻辑和视图分离，利于维护 WEB 应用。本章主要介绍 JSP 的基础知识。

目标

- JSP 的概念（理解）
- JSP 脚本元素（掌握）
- JSP 指令（掌握）
- Request 对象（掌握）
- Response、out 对象（掌握）

目录

- 6.1 JSP 概念
- 6.2 第一个 JSP
- 6.3 脚本元素
- 6.4 代码段（Scriptlet
- 6.5 声明（Declaration）
- 6.6 表达式（Expression）
- 6.7 注释
- 6.8 Page 指令
- 6.9 Include 指令
- 6.10 Request 对象
- 6.11 Response 对象
- 6.12 out 对象
- 6.13 示例

6.1 JSP 概念

到目前为止，我们已经掌握了 Servlet 的开发技术，通过 Servlet 可以处理业务逻辑，也可以将处理结果输出到客户端浏览器中，但使用 Servlet 进行响应输出时，输出语句 `out.println()` 显得非常笨拙，对于布局复杂的页面更是如此。

JSP 就是为了解决这些问题在 Servlet 技术之上开发的。JSP 技术由 SUN 公司提出，利用它可以很方便地在页面中生成动态的内容。JSP 的本质是一个页面文件，通过开发 JSP 页面，我们可以在 HTML 中内嵌 Java 代码段，并且可以调用其他 Java 组件（如：javabeen）。在标准 HTML 页面中可以出现的任何内容都可以在 JSP 页面中出现。

实际上，JSP 只是简单地将 Java 代码嵌到 HTML 网页中去而已。可以直接将现有的 HTML 网页将它们的扩展名由 “.html” 改为 “.jsp”，就创建了一个 JSP 页。

JSP 是一种文本页面，它本身是不能运行的，JSP 页面在运行之前要被转换成 Servlet，解释过程由 WEB 服务器自动完成，转换后的 Servlet 会被 WEB 容器调用并运行，然后 WEB 容器将结果送到客户端浏览器中。从这个角度看，JSP 的本质其实是 Servlet。

6.2 第一个 JSP

本部分通过一个简单的 JSP 页面来理解 JSP 的运行机制：

- 1、在 Tomcat 中创建一个 WEB 站点 `webprj_t6`（步骤参考第二章内容），然后在站点中创建一个文本文件，将文件名改为 `test.jsp`，站点结构如下：

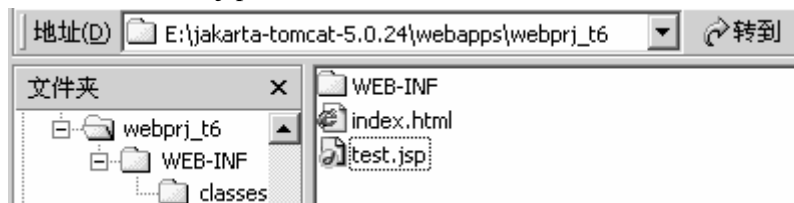


图 6-1 站点 `webprj_t6`

- 2、在 `test.jsp` 中输入以下内容：

```
<html>
<body>
This is the first jsp!!!
</body>
</html>
```

- 3、启动 Tomcat，然后访问 `test.jsp`：



图 6-2 访问 test.jsp

分析：运行结果与普通的 HTML 页面一样，在上例中将 test.jsp 更名为 test.html 效果与上图相同，当然在 JSP 中除了可以包含普通的 HTML 代码以外，还可以包含许多 JSP 特有的内容。在这里，先分析一下 test.jsp 的运行流程：

- 1、 客户访问 test.jsp，tomcat 服务器捕获请求，然后在站点中查找 test.jsp；
- 2、 tomcat 在站点中找到 test.jsp 后，tomcat 会自动为该页面生成一个相应的 Servlet JAVA 文件，该 servlet 源代码在 TOMCAT_HOMES\work 目录的相应子目录下可以找到，如图：

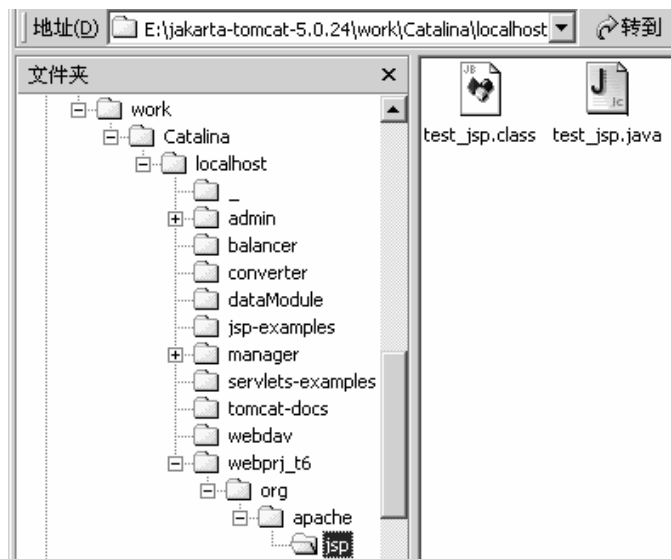


图 6-3 Tomcat 为 test.jsp 生成的 Servlet 代码

- 3、 通过上图可以看出，tomcat 服务器为 test.jsp 生成了一个 Servlet：test_jsp.java，并且已经编译成了 test_jsp.class，这两个文件都是自动产生的。
- 4、 第一次访问 test.jsp 时，会感觉响应速度很慢，是因为 tomcat 在执行转换工作，但第二次访问时速度会很快，原因是同一个 jsp 只会被转换一次，一旦 servlet 对应的 class 文件产生了，就会服务于以后的所有请求。当然，如果该 jsp 页面的内容发生变化时，服务器还是会自动重新将 JSP 页面编译成 servlet 的。
- 5、 观察一下 test_jsp.java 的部分代码：

```
try {  
    _jspxFactory = JspFactory.getDefaultFactory();  
    response.setContentType("text/html");  
    pageContext = _jspxFactory.getPageContext(this, request, response,  
        null, true, 8192, true);
```



```

    _jspx_page_context = pageContext;
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    _jspx_out = out;

    out.write("<html>\r\n");
    out.write("<body>\r\n");
    out.write("This is the first jsp!!!\r\n");
    out.write("</body>\r\n");
    out.write("</html>");
}

```

可以看出, test.jsp 页面中的所有内容均被放到了 out.write() 中, 即 jsp 的本质仍然是 servlet, 它的所有内容还是通过字符输出流输出到客户浏览器中的, 只不过这些 JAVA 代码不用 JSP 程序员编写。

6.3 脚本元素

在 1.1 中分析了 JSP 的运行过程, 及其与 Servlet 的联系, 如果只是将普通 HTML 放到 JSP 页中, 则 JSP 的存在就没有多大意义了, 在 JSP 中, 还可以包含以下内容:

➤ 指令

指令提供该页的全局信息, 例如, 字符编码格式, 错误处理, 是否支持 session 等。

➤ 声明 (Declaration)

可以声明页面范围的变量和方法。

➤ 代码段 (Scriptlet)

嵌入页面内的是一些 java 代码片断。

➤ 表达式 (Expression)

可以将表达式的结果变成 String 类型以便于包含在页面的输出中。

以上内容中“指令”只是用以设置 JSP 的环境信息, 其余三项内容可以运行以实现某些具体的功能, 这三项(声明、代码段、表达式)也统称为脚本元素。三个脚本元素的基本语法都是以一个“<%”开头, 而以一个“%>”结尾的, 三者构成了 JSP 中主要的程序部分。

6.4 代码段 (Scriptlet)

通过代码段, 可以将 Java 代码块嵌入到 JSP 中, 代码段就是 jsp 中的代码部分, 在这个部分中可以使用几乎任何 java 的语法。

语法为:

<% 代码片段 %>

代码段示例 (第一行用以设置本 JSP 可以显示中文字符):

```

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

<html>

```

```
<body>
<br>开始执行代码段:<br>
<%
    java.util.Date date=new java.util.Date();
    out.println("现在时间:" + date.toString() );
%>

<br>
</body>
</html>
```

分析：在代码段中，为了向浏览器输出内容，要使用一个称为“out”的变量，它代表一个到客户端的输出流。这个变量不需要定义。实际上，jsp 被转成 servlet 时，该变量会被自动定义的，一起被定义的还有其它变量，这些变量在后续的教程中会细说。

该 jsp 运行结果如下：

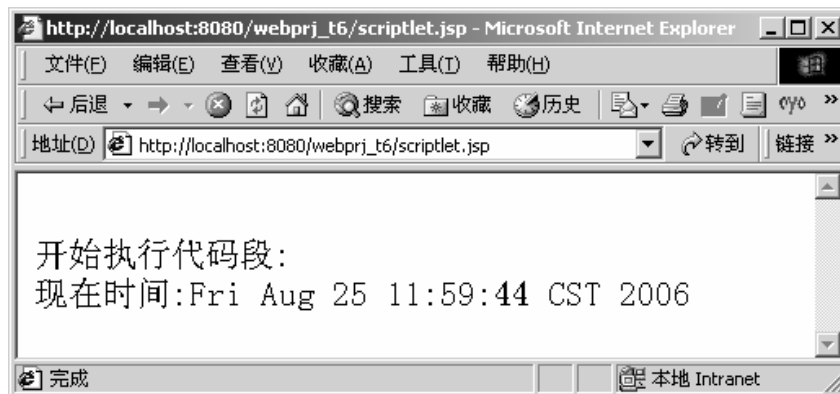


图 6-4 JSP 运行结果（使用代码段）

6.5 声明（Declaration）

JSP 中的声明用于声明一个或多个变量和方法，并不输出任何的文本到 out 输出流中。在声明元素中声明的变量和方法将在 JSP 页面初始化时初始化。

语法为：

```
<%! 变量或方法声明 %>
```

以“<%!”开头，一定要注意感叹号，如果省略感叹号，则变为了代码段。通过该元素声明的变量会变为相应 servlet 的数据成员，声明的方法会变为 servlet 的方法成员。示例如下：

```
<%@ page contentType="text/html; charset=gb2312" %>
<%!
    int age=10;
    public int getAge(){
        return this.age;
    }
}
```

```

%>
<br>开始执行代码段:<br>
<%
    int age=getAge();
    out.println("age=" + age);
%>

```

分析:

- ◆ 本 jsp 中声明了一个变量 `age`，一个方法 `getAge()`;
- ◆ 在代码段中也使用了语句 “`int age=getAge()`”，但这不是 JSP 声明，该 `age` 会作为 servlet 中 `service` 方法的局部变量来实现，而 `<%! int age %>` 中的 `age`，则会作为 servlet 的成员，可以到 tomcat 的 `work` 目录下观察生成的 Servlet JAVA 文件;

6.6 表达式 (Expression)

JSP 中的表达式可以被看作一种简单的输出形式。

语法为: `<%= 表达式 %>`，表达式可以为任意合法的 JAVA 表达式。

举例如下:

```

<%@ page contentType="text/html; charset=gb2312" %>
<h1>
100*20=<%=100*20%>
</h1>

```

运行结果如下:



图 6-5 JSP 运行结果 (使用表达式)

6.7 注释

在 JSP 中共有以下几种注释:

1、HTML 注释

`<!-- HTML 注释 -->`

2、JAVA 注释

```

<%
    //这是 JAVA 注释
    /*这是 JAVA 注释*/

```

```
/**
 * 这也是 JAVA 注释
 */
%>
```

3、 JSP 注释

<%-- JSP 注释 --%>

其中第一种注释可以在客户端浏览器中“查看源文件”中查看到，第二、三种注释在客户端看不到，但第二种注释可以在 JSP 对应的 servlet 源文件中看到，第三种注释只能在 JSP 页面中看到，而不会写入 Servlet 源文件。

6.8 Page 指令

6.8.1 指令概述

指令(Directives)主要用来提供整个 JSP 网页相关的信息，并且用来设定 JSP 网页的相关属性，例如：网页的编码方式、语法、信息等。

在 JSP 语法台指令的起始符号为：<%@，终止符号为： %>

中间部分就是一些指令及该指令一连串的属性设定，如下所示：

```
<%@ directive { attribute="value" } %>
```

当我们设定两个属性时，可以将之合二为一，如下：

```
<%@ directive attribute1="value1" %>
```

```
<%@ directive attribute2="value2" %>
```

亦可以写成：

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

在 JSP 规范中，有三种指令：page、include 和 taglib，每一种指令都有各自的属性，三种指令分别实现不同的功能，其中第三个指令在第十一章讲解自定义标签时讲解，以下讲解前两个指令。

6.8.2 Page 指令

page 指令描述了和页面相关的指示信息，设定整个 JSP 网页的属性和相关功能。在一个 jsp 页面中，page 指令可以出现多次，但是每一种属性却只能出现一次，重复的属性设置将覆盖掉先前的设置。

page 指令的基本语法格式如下：

```
<%@ 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" ]
%>
```

这些属性的含义见表 7-1 所示:

属性及可选值	功能
language="java"	language 变量告诉服务器在文件中将采用哪种语言, 默认值为 Java。
extends="package.class"	extends 定义了由 jsp 页面产生的 servlet 的父类, 通常这个属性不会用到。
import="package.*,package.class"	声明需要导入的 Java 包的列表, 这些包可用于程序段, 表达式, 以及声明。 下面的包在 JSP 编译时已经自动导入到 Servlet, 不需再次指明: java.lang.* javax.servlet.* javax.servlet.jsp.* javax.servlet.http.*
session="true false"	设定客户是否需要 HTTP Session, 如果它为 false, 在本 JSP 页中不能使用 session 对象, 缺省值是 true。
buffer="none 8kb sizekb"	决定输出流 (out 对象) 是否需要缓冲, 缺省值是 8kb。
autoFlush="true false"	设置如果 buffer 溢出, 是否需要强制输出, 如果其值被定义为 true(缺省值), 输出正常, 如果它被设置为 false, 如果这个 buffer 溢出, 就会导致异常的发生. 如果 buffer 为 none, 则不能将 autoFlush 设置为 false。
isThreadSafe="true false"	设置 Jsp 文件是否能多线程使用。缺省值是 true, 也就是说, JSP 能够同时处理多个用户的请求, 如果设置为 false, 一个 jsp 只能一次处理一个请求 (即以单线程方式运行)
info="text"	页面信息, 在程序中能够使用 Servlet.getServletInfo 方法取出。
errorPage="relativeURL"	当本 JSP 页出现异常时要转向的 URL, 要转向的目的 JSP 页中要把 isErrorPage 设成 true。
isErrorPage="true false"	标志一个页面是否为错误处理页面。如果设置为 true, 该页中可以使用 Exception 对象。
contentType="mimeType [;charset=characterSet]" "text/html; charset=ISO-8859-1"	设置 MIME 类型。缺省 MIME 类型是: text/html, 缺省字符集为 ISO-8859-1。

表 6-1 page 指令属性列表

<%@ page %>指令作用于整个 JSP 页面，可以在一个页面中多次使用<%@ page %>指令，但是除 import 属性以外，其它的属性只能用一次，如果重复使用同一属性则后面的设置会覆盖原先设置。

<%@ page %>指令可以放在 JSP 页的任何位置，但按照规范，建议放在文件顶部。

6.8.3 page 指令示例

1、 示例 1——使用 contentType、import 属性

示例中开发了一个自定义的 javabean，在 JSP 中设置字符编码为简体中文，且导入了其他 JAVA 包及自定义 bean 所在的包：

1) 自定义 bean:

```
package demo;
import java.io.Serializable;
public class User implements Serializable{
    private String username;
    private String userpass;

    public void setUsername(String username){
        this.username=username;
    }
    public String getUsername(){
        return this.username;
    }

    public void setUserpass(String userpass){
        this.userpass=userpass;
    }
    public String getUserpass(){
        return this.userpass;
    }
}
```

2) page_test_1.jsp 页内容（向 JSP 中导入了两个 JAVA 包，其中一个为自定义包）:

```
<%@ page contentType="text/html;charset=gb2312" %>
<%@ page import="java.util.*" %>
<%@ page import="demo.*" %>
使用 java.util.*包中的类:<br>
<%
    ArrayList list=new ArrayList();
    list.add( "hello" );
    list.add( "ok" );
```

```

list.add( "welcome" );
out.print("下面为集合 list 中的内容:<br>");
out.print( list );
%>
<br><br>
使用自定义 demo 包中的类:<br>
<%
    User user=new User();
    user.setUsername("张三");
    user.setUserpass("1234");
    out.println( "用户名:"+user.getUsername() );
    out.println( "<br>密码:"+user.getUserpass() );
%>

```

3) 运行结果如下:

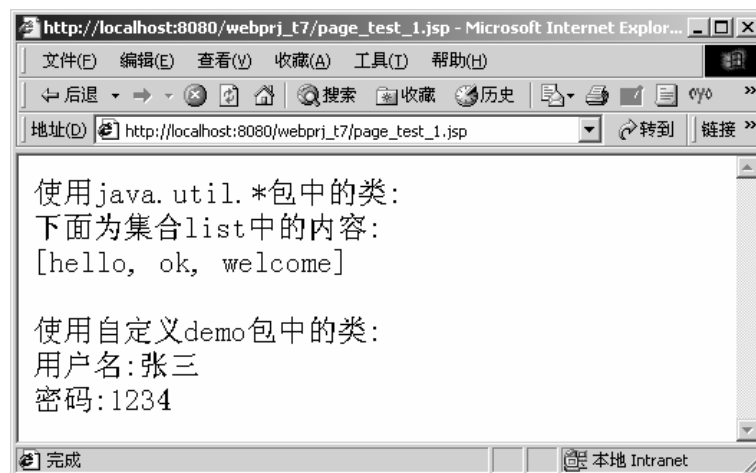


图 6-6 page_test_1.jsp 运行结果

2、 示例 2——使用 page 指令其他属性

1) 编写 page_test_2.jsp, JSP 页内容如下:

```

<%@ page contentType="text/html; charset=GBK" %> <!--默认 text/htm-->
<%@ page info="作者:XXXX"%> <!--指定页面信息如作者等-->
<%@ page language="java" %> <!--指定使用的脚本语言-->
<%@ page session="true" %> <!--说明本页面是否允许使用 session, 缺省为 true-->
<%@ page buffer="12kb" %> <!--控制页面缓冲输出是否使用 none 或 12kb 8kb(默)...-->
<%@ page autoFlush="true" %> <!--控制页面缓冲满时是否自动刷新缓冲器, 当前内容被发至
http,true(默)-->
<html>
<head>

```

```

<title>page_test_2</title>
</head>
<body bgcolor="#ffffff">
<h1>
page 指令属性使用<hr>
</h1>
</body>
</html>

```

2) 运行结果:

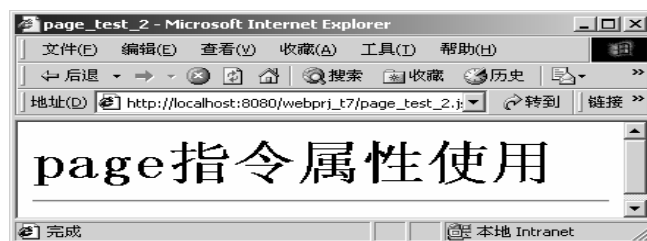


图 6-7 page_test_2.jsp 运行结果

6.9 Include 指令

include 指令的作用是在本 JSP 页中包含另一个文件。包含的过程是静态的（即包含的是文件的源代码内容），包含的文件可以是 JSP、HTML、文本。

include 指令的基本语法格式如下，其中 file 属性用来指明要包含的文件路径：

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

6.9.1 include 指令示例

1、 示例 1——包含其他 JSP 页

1) 编写 JSP 页 include1.jsp，内容如下：

```

<%@ page contentType="text/html; charset=gb2312"%>
开始包含 include2.jsp:<br>
<%@ include file="include2.jsp" %>
<br>include2.jsp 包含完毕.
<hr>
<%
    out.println("<br>调用 include2.jsp 中声明的函数: square(10)="+square(10));
%>
<br><br><br>

```

通过指令 include 包含过来的 include2.jsp 中声明的函数

可以在包含文件 include1.jsp 中使用（包含的是 include2.jsp 的源代码）

编写 JSP 页 include2.jsp，内容如下：


```

<%@ page contentType="text/html; charset=gb2312"%>
<%!
//声明求平方的方法;
public int square(int i){
    return i*i;
}
%>
<%
    out.println("in include2.jsp: square(10)="+square(10));
%>

```

访问 include1.jsp, 结果如下:

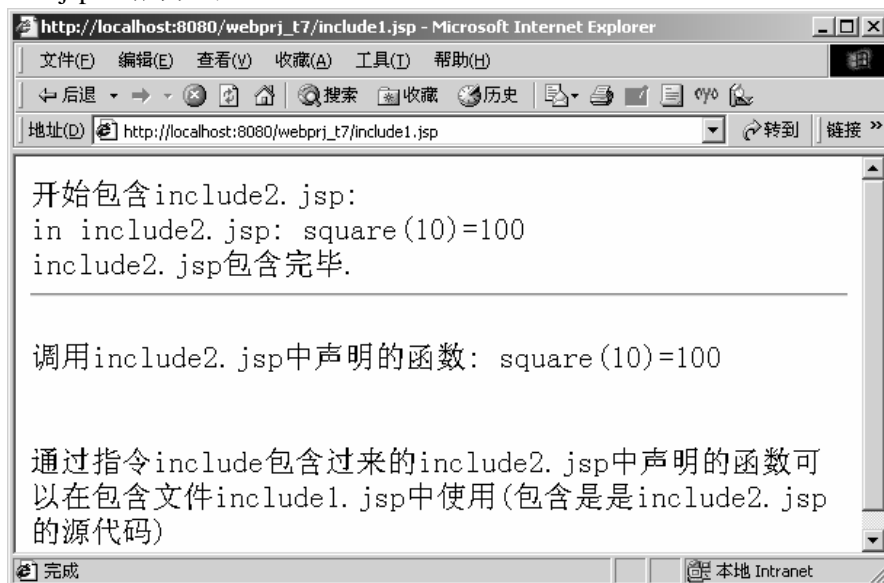


图 6-8 page_test_2.jsp 运行结果

2) 分析:

以上例子中, 在 page_test_2.jsp 中声明了一个 square () 方法, 然后将这个文件包含到了 page_test_1.jsp 中, 由于 include 指令是以源代码的方式静态包含的, 所以该方法的代码被包含到了 page_test_1.jsp 中。

提示: 如果两个文件中都声明了相同的方法, 则运行会报错。原因是: 在第一个文件中会包含两份代码完全相同的方法声明。

2、 示例 2——包含文本文件

1) 编写文本文件 include.txt:

```

aaaaaaaaaaaa
bbbbbbbbbbbb
cccccccccccc

```

2) 编写 JSP 页面 include3.jsp, 代码如下:

```
<%@ page contentType="text/html; charset=gb2312"%>
<html><body>
包含 include.txt:<br>
<pre>
<%@ include file="include.txt" %>
</pre>
</body></html>
```

访问 include3.jsp, 结果如下

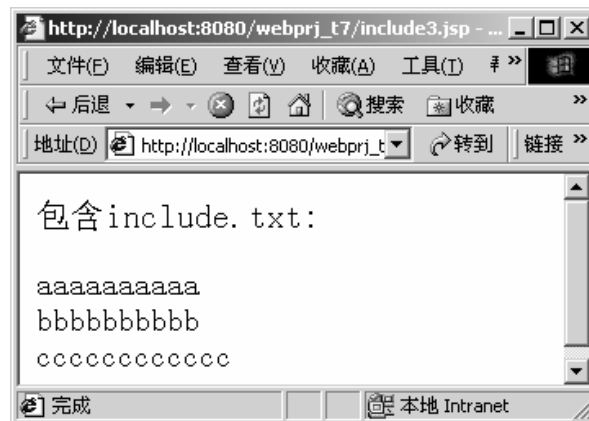


图 6-9 page_test_2.jsp 运行结果

说明：如果在程序运行过程中修改了 include.txt 文件的内容，则刷新页面时会自动反应出来。包含 JSP 页时也是如此。

6.10 Request 对象

为方便开发，JSP 环境中内置了一些对象，不需要预先声明就可以在脚本代码和表达式中随意使用，前面已经接触过的 out 就是内置对象。在 JSP 中总共提供了 9 个内置对象，分别实现不同功能。有些资料中也将内置对象称为隐含对象、预定义变量，其本质都是一样的。

6.10.1 理解内置对象

上面已经提到，内置对象是由 JSP 环境提供的，那么到底是在什么时候由谁提供的？现回忆一下 JSP 页的运行流程：用户访问 JSP 页时，服务器寻找该页，找到后将其转为 Servlet，就是在这个转换过程中为 JSP 声明了内置对象，通过下面示例可以理解这一点：

1) 编写 test.jsp, 内容如下(就一行字符，使用 isErrorPage 属性是为了产生 exception 内置对象)：

```
<%@ page isErrorPage="true" %>
aaaaaaaaaa
```

2) 运行 test.jsp, 结果如下：

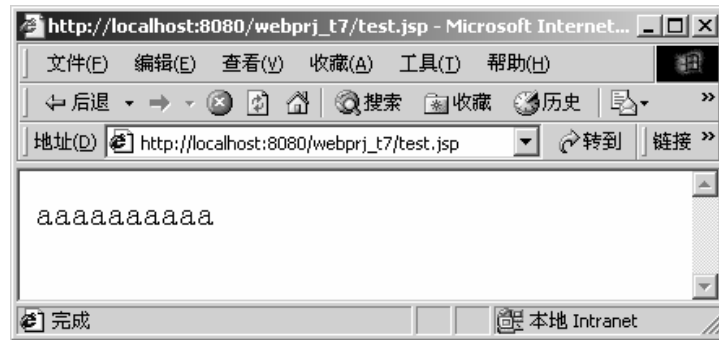


图 6-10 Servlet 的存放位置

- 3) 打开 `tomcat_home\work\Catalina\localhost\webprj_t7\org\apache\jsp` 文件夹, 在该文件夹下会发现一个 `test_jsp.java` 的源文件, 这就是 tomcat 服务器为 `test.jsp` 生成的 servlet 的源代码, 其部分代码如下:

```
public void _jspService(HttpServletRequest request, HttpServletResponse
response)
    throws java.io.IOException, ServletException {

    JspFactory _jspxFactory = null;
    PageContext pageContext = null;
    HttpSession session = null;
    Throwable exception =
org.apache.jasper.runtime.JspRuntimeLibrary.getThrowable(request);
    if (exception != null) {
        response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
    }
    ServletContext application = null;
    ServletConfig config = null;
    JspWriter out = null;
    Object page = this;
    JspWriter _jspx_out = null;
    PageContext _jspx_page_context = null;

    try {
        _jspxFactory = JspFactory.getDefaultFactory();
        response.setContentType("text/html");
        pageContext = _jspxFactory.getPageContext(this, request, response,
            null, true, 8192, true);
        _jspx_page_context = pageContext;
```

```
application = pageContext.getServletContext();
config = pageContext.getServletConfig();
session = pageContext.getSession();
out = pageContext.getOut();
_jsp_out = out;

out.write("\r\n");
out.write("aaaaaaaaaa");

.....
```

4) 分析:

_jspService() 方法相当于 Servlet 中的 Service() 的功能, 该方法的参数 request、response 是两个对象, 已经被定义好, 称为内置对象。且该方法内部还定义了几个对象: pageContext 、

session 、 application 、 config 、 out 、 page 、 exception, 共计是九个对象, 这些对象称为内置对象。也可以观察出来: 这些对象很多都是根据 ServletAPI 生成的。

6.10.2 request 对象

request 代表请求对象, 是 javax.servlet.http.HttpServletRequest 接口的对象, 主要用于接受客户端通过 HTTP 协议连接传输到服务器端的数据。其功能与 Servlet 中的 HttpServletRequest 对象的功能完全相同。该对象提供的重要方法参见下表:

方法	功能
getAttribute(String name)	返回指定属性的属性值
getContentType()	得到请求体的 MIME 类型
getParameter(String name)	返回 name 指定参数的参数值
getRemoteAddr()	设定客户是否需要 HTTP Session, 如果它为 false, 在本 JSP 页中不能使用 session 对象, 缺省值是 true.
setAttribute(String key, Object obj)	设置属性的属性值
getRmoteAddr()	得到发送此请求的客户端 IP 地址
getRealPath(String path)	返回一虚拟路径的真实路径
setCharacterEncoding(String)	设置按什么编码读取字符内容, 当客户传递过来的参数是中文时, 应该设置为 request.setCharacterEncoding("gb2312")进行读取;

表 6-2 request 对象重要方法列表

6.11 Response 对象

response 代表响应对象, 是 javax.servlet.http.HttpServletResponse 接口的对象, 主要用于向客户端发送数据。该对象提供的重要方法参见下表:

方法	功能
getWriter()	返回可以向客户端输出字符的字符流
setContentType(String type)	设置响应的 MIME 类型

<code>sendRedirect(java.lang.String location)</code>	重定向客户端的请求
--	-----------

表 6-3 response 对象重要方法列表

6.12 out 对象

out 对象代表到客户端的字符型输出流，通过该对象可以将字符数据发送到客户浏览器中。该对象提供的重要方法参见下表：

方法	功能
<code>out.print(String)</code>	输出字符数据
<code>out.println(String)</code>	输出字符数据并输出 “\n”
<code>out.flush()</code>	输出缓冲区的数据
<code>out.close()</code>	关闭输出流
<code>out.clear()</code>	清除缓冲区里的数据，但不把数据写到客户端

表 6-4 out 对象重要方法列表

response 对象与 out 对象的区别与联系：

- ◆ 两者都是用于向客户端输出数据
- ◆ out 负责输出字符型数据，且这些数据要在浏览器中显示出来
- ◆ response 常用以输出一些不在浏览器中显示的数据（如：Http 协议头、重定向指令等）

6.13 示例

1、 示例 1

该示例综合演示 request、response 对象的用法。

1) 编写 login.html，提供登录表单：

```
<html><head><title>Login</title></head>
<body>
<form action="login.jsp">
    用户名称: <input type="text" name="username"><br>
    用户密码: <input type="password" name="userpass"><br>
    <input type="submit" value="登录">
</form></body></html>
```

2) 编写 login.jsp，用来接收登录数据：

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    //如果用户名称为 admin,则显示欢迎语;
    String name=request.getParameter("username");
    if(name.equals("admin")) {
%>
<h3>欢迎您, <%=name %>!/h3>
```

```

<%
    }else{
        //用户名称不是 admin,则转到错误页;
        response.sendRedirect("loginfail.jsp");
    }
%>

```

3) 编写 loginfail.jsp, 显示登录失败的信息:

```

<%@ page language="java" contentType="text/html; charset=gb2312" %>
<h3>登录失败</h3>
<a href="login.html">重新登录</a>

```

4) 运行 login.html, 效果如下:



图 6-11 不使用 admin 登录, 登录失败



图 6-12 使用 admin 登录, 登录成功

2、 示例 2——测试 request 对象

本示例中演示 request 对象其他方法的使用。

1) 编写 request.jsp, 内容如下:

```

<%@ page language="java" contentType="text/html; charset=gb2312" %>
<html><head><title>request</title></head><body>

<b>Browser:</b> <%=request.getHeader("User-Agent") %><br>

```

```

<b>Cookies:</b> <%=request.getHeader("Cookie") %><br>

<b>Accepted MIME types:</b> <%=request.getHeader("Accept") %><br>
<b>HTTP method:</b> <%=request.getMethod() %><br>
<b>IP Address:</b> <%=request.getRemoteAddr() %><br>
<b>Country:</b> <%=request.getLocale().getDisplayCountry() %><br>
<b>Language:</b> <%=request.getLocale().getDisplayLanguage() %><br>
<b>RealPath("/")</b><%=request.getRealPath("/")%>
</body></html>

```

2) 访问 request.jsp, 结果如下:

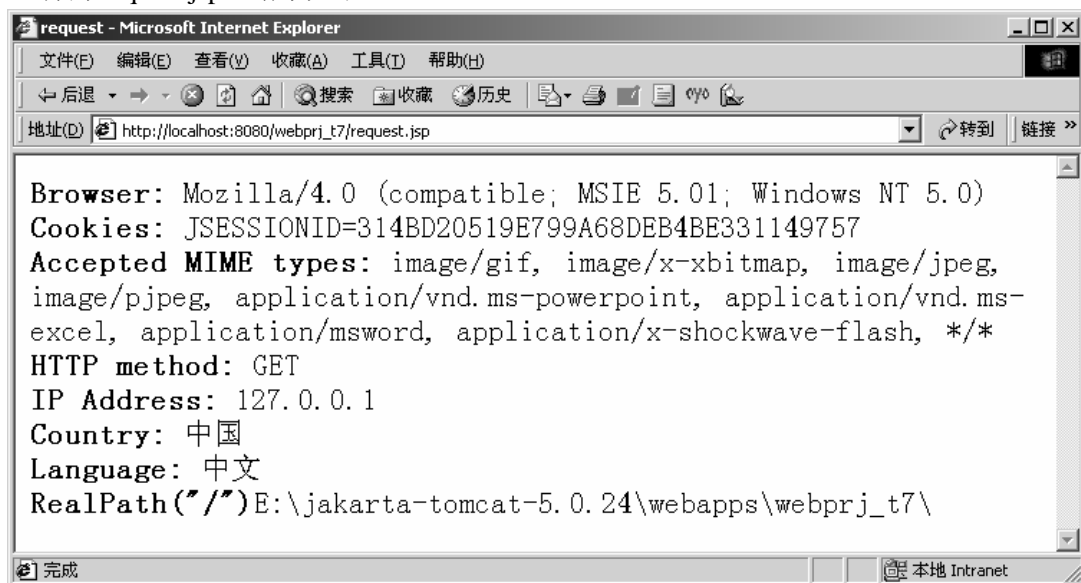


图 6-13 request 对象的其他方法

3、 示例 3——测试 response 对象

1) 编写 sendError.jsp, 内容如下:

```

<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    response.sendError(500);
%>

```

2) 说明: response.sendError(500)用以向客户端浏览器中发送错误状态码, 结果如下:

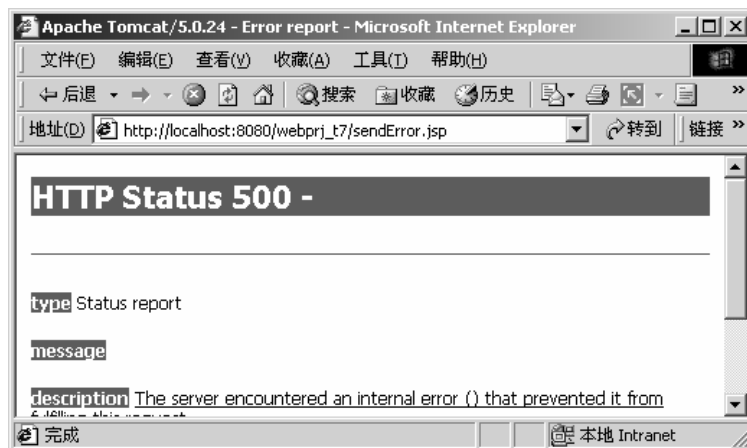


图 6-14 response 对象的 sendError()方法

4、 示例 4——使用 response 对象发送协议头控制浏览器

1) 编写 refresh.jsp, 内容如下:

```
<%@ page language="java" contentType="text/html; charset=GB2312"%>
<HTML>
<HEAD>
</HEAD>
<BODY>
    <div align="center">
        <H1>
            浏览器将在 2 秒钟后返回主页..</H1><br>
        <%
            response.setHeader("Refresh", "2;URL=index.html");
        %>
    </div>
</BODY>
</HTML>
```

2) 运行 refresh.jsp, 结果如下 (2 秒钟后会转向 index.html 页):

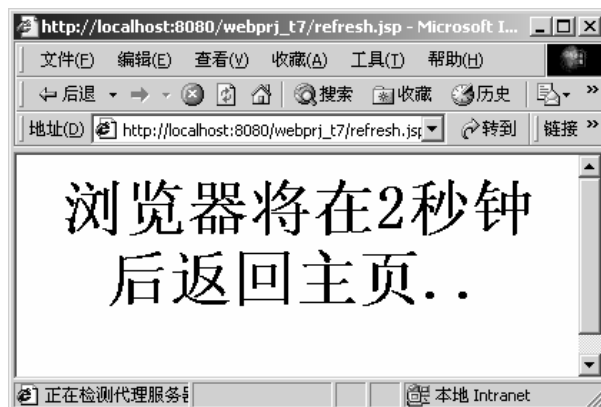


图 6-15 使用 response 对象的 setHeader 方法

在聊天室这类应用中，用户列表、公共发言区可以使用该方法进行定时自动刷新。

小结

JSP 技术由 SUN 公司提出，利用它可以很方便地在页面中生成动态的内容。

JSP 页面在运行之前要被转换成 Servlet，解释过程由 WEB 服务器自动完成，从这个角度看，JSP 的本质其实是 Servlet。

在 JSP 中除了可以包含 HTML 以外，还可以包含：指令、声明、代码段、表达式；

声明、代码段、表达式三者统称为脚本元素，构成了 JSP 中主要的程序部分。

指令(Directives)主要用来提供整个 JSP 网页相关的信息，其中 page 指令描述了和页面相关的指示信息，设定整个 JSP 网页的属性和相关功能。

include 指令的作用是在本 JSP 页中包含另一个文件。包含的过程是静态的（即包含的是文件的源代码内容），包含的文件可以是 JSP、HTML、文本。

JSP 环境中内置 9 个对象，不需要预先声明就可以在脚本代码和表达式中随意使用，。有些资料中也将内置对象称为隐含对象、预定义变量，其本质都是一样的。

request 对象用于接受客户端通过 HTTP 协议连接传输到服务器端的数据。

response 对象主要用于向客户端发送数据。

课后练习

- 1、 简述 JSP 页面的运行流程，说明 JSP 与 Servlet 的联系？
- 2、 简述声明、代码段、表达式三种语法及其作用？
- 3、 简述 page 指令中几个重要属性的功能？
- 4、 为什么说 include 指令是静态包含？使用时有什么注意事项？
- 5、 request、response 对象主要功能是什么，分别与 Servlet 中的哪些 API 对应？

第七章 JSP(二)

概要

上一章已经了解了 JSP 内置对象的概念，本章继续讲解其他几个重要的内置对象，主要介绍以下对象的用法：application、session、pageContext、config，另外还讲解了 Cookie 在 JSP 中的用法，但要清楚 Cookie 不是内置对象，它要通过内置对象 request、response 来进行操作。application、session、pageContext 经常用于存储 WEB 应用中的数据，它们存储的数据作用域大小各不相同。

目标

- application（掌握）
- session（掌握）
- pageContext（掌握）
- cookie（掌握）
- config（理解）

目录

- 7.1 application 对象
- 7.2 session 对象
- 7.3 pageContext 对象
- 7.4 cookie
- 7.5 config 对象

7.1 application 对象

7.1.1 application 对象介绍

applicaton 是 javax.servele.ServletContext 接口的对象,可以在该对象中保存数据信息,它是一个共享的内置对象,即一个容器中的多个用户共享一个 application 对象,故其保存的信息被所有用户所共享。一旦创建,除非服务器关闭,否则将一直保持下去。

该对象的方法与 javax.servele.ServletContext 接口中的方法完全相同,重要方法列表如下:

方法	功能描述
setAttribute(Stirng ,Object)	将对象用指定键名存储到 applicaton 对象中
getAttribute(String)	返回指定名字的属性值, 如果不存在返回 null
removeAttribute(String)	从 application 中移除指定数据
getRealPath(String)	返回给定虚拟路径对应的物理路径

表 7-1 applicaton 的重要方法

7.1.2 application 对象示例

1、 示例 1——计数器

本示例中统计某个页面的访问次数,只要有人访问该页,计数器就增 1,该计数值存储在 application 全局作用域中。

1) JSP 页 application_1.jsp 内容如下:

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

<%
    int n = 0;
    Object obj = application.getAttribute("counter");

    if(obj==null){
        //如果没取到计数值,则将计数值设为 1
        n = 1;
        application.setAttribute("counter", new Integer(n) );
    }
    else{
        //如果取到计数值,则将计数值增 1,再存回 application
        n = ((Integer)obj).intValue() + 1;
        application.setAttribute("counter", new Integer(n) );
    }
%>

<h2>你是第<%=n%>位访客! </h2>

<center>
    <a href="application_1.jsp">刷新本页</a> |
```

```
<a href="index.html">返回主页</a>
</center>
```

2) 访问 application_1.jsp, 并连接访问三次(也可以打开新的浏览器进行访问), 运行结果如下:



图 7-1 page_test_1.jsp 运行结果

3) 说明: application 对象被所有用户共享, 本例中每打开一个新浏览器就相当于增加了一个新用户, 但后台计数值是共享的。

2、 示例 2——清除计数器

需要从 application 作用域中清除计数器对象时, 可以使用以下代码。

JSP 页 application_2.jsp 内容如下:

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    application.removeAttribute("counter");
%>
<script language="javascript">
    window.alert("计数器已清除!");
    window.location="index.html";
</script>
```

访问 application_2.jsp, 运行结果如下:



图 7-2 用 removeAttribute()清除计数值

再次访问 application_1.jsp, 计数器会重新开始累计, 运行结果如下:



图 7-3 计数值开始重新累计

3、 示例 3——设置计数器

有时在应用中需要将 `application` 作用域中的计数值设置为指定值，可以使用以下代码。

JSP 页 `application_3.jsp` 内容如下：

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    application.setAttribute("counter",new Integer(1000) );
%>
<script language="javascript">
    window.alert("计数器已置为 1000!");
    window.location="index.html";
</script>
```

运行结果如下，再次访问计数器时，将从 1000 开始累计：

图 7-4 用 `setAttribute()` 设置计数值

7.2 Session 对象

7.2.1 Session 对象介绍

会话状态维持是 Web 应用开发者必须面对的问题。在第四章已经讲到，有多种方法可以用来解决这个问题，如使用 Cookies、隐藏的表单输入域，或直接将状态信息附加到 URL 中(即 URL 重写)。Java Servlet 提供了一个在多个请求之间持续有效的会话对象，该对象允许用户存储和提取会话状态信息。JSP 也同样支持 Servlet 中的这个概念，即通过 `session` 内置对象可以完成与 Servlet 中相同功能

的会话管理。在 session 内置对象中可以存储用户级的数据，这些数据可以在同一会话中的任何 JSP 页中访问到。

session 重要方法列表如下：

方法	功能描述
setAttribute(String, Object)	将对象用指定键名存储到 session 中
getAttribute(String)	返回指定名字的属性值，如果不存在返回 null
invalidate()	销毁 session 对象
removeAttribute(String)	从 session 作用域中移除指定数据

表 7-2 session 的重要方法

7.2.2 session 对象示例

1、 示例 1——session 计数器

本示例中针对某个用户，统计某个页面的访问次数，只要用户访问该页，计数器就增 1，该计数值存储在 session 作用域中，所以每个用户都有一个不同的计数值。

1) JSP 页 session_1.jsp 内容如下：

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    int n=0;
    Object obj=session.getAttribute("counter");
    if(obj==null){
        //如果没取到计数值，则将计数值设为 1
        n=1;
        session.setAttribute("counter", new Integer(n) );
    }else{
        //如果取到计数值，则将计数值增 1，再存回 session
        n=((Integer)obj).intValue()+1;
        session.setAttribute("counter", new Integer(n) );
    }
%>
    session 中的 counter=<%=n%><br>
    sessionID: <%=session.getId()%><br>
    session 创建时间: <%=session.getCreationTime()%><br>
    session 创建时间:
        <%= new java.util.Date(session.getCreationTime()).toString() %>
    <br>
<center>
    <a href="session_1.jsp">刷新本页</a> |
    <a href="index.html">返回主页</a>
</center>
```

访问 session_1.jsp, 运行结果如下:

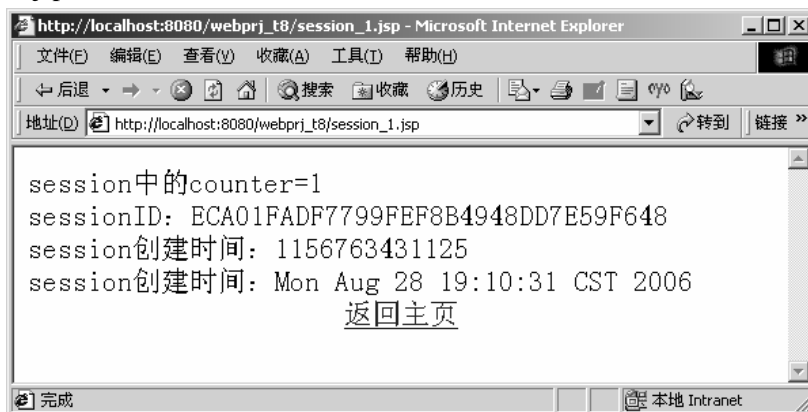


图 7-5 session 计数器

说明: session 对象是用户级的, 本例中每打开一个新浏览器就相当于增加了一个新用户, 所以各浏览器中看到的计数值都是当前用户的, 各浏览器间计数值的累计互不影响。

2、 示例 2——清除 session 中的数据

需要从 session 作用域中清除数据时, 可以使用以下代码。

1) JSP 页 session_2.jsp 内容如下:

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    session.removeAttribute("counter");
%>
<script language="javascript">
    window.alert("计数器已清除!");
    window.location="index.html";
</script>
```

2) 访问 session_2.jsp, 运行结果如下:

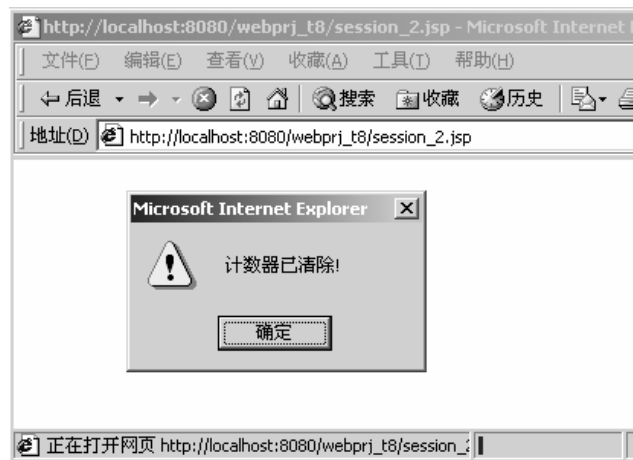


图 7-6 清除 session 中的数据

3) 再次访问 session_1.jsp, 计数器会重新开始累计, 另外从运行结果中可以看出, session 的创

建时间未改变:

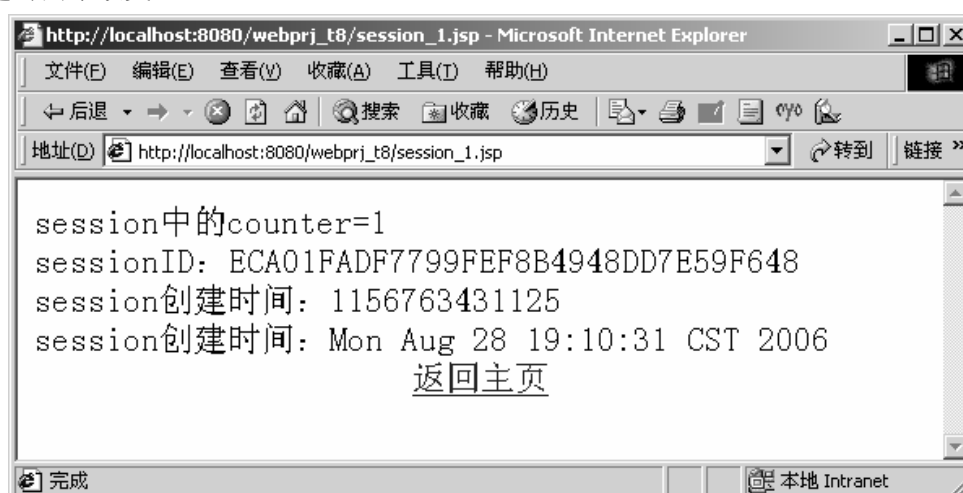


图 7-7 清除计数值后再次访问 session 计数器页面

3、 示例 3——设置计数器、销毁 session

JSP 页 session_3.jsp 内容如下:

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    session.setAttribute("counter",new Integer(1000) );
%>
<script language="javascript">
    window.alert("计数器已置为 1000!");
    window.location="index.html";
</script>
```

JSP 页 session_4.jsp 内容如下:

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    session.invalidate();
%>
<script language="javascript">
    window.alert("session 已销毁!");
    window.location="index.html";
</script>
```

访问 session_3.jsp, 运行结果如下:

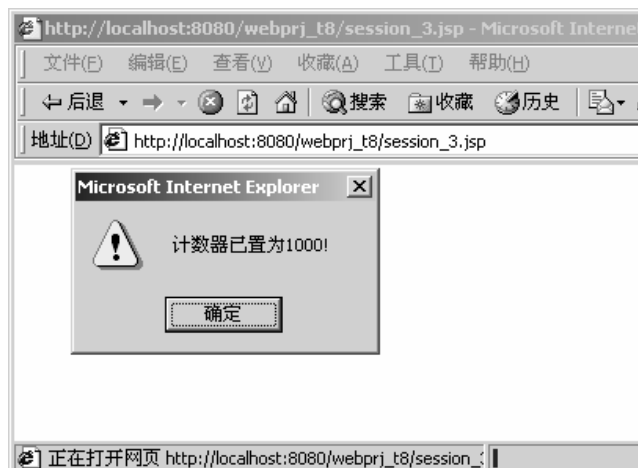


图 7-8 改变 session 中数据的值

访问 session_4.jsp, 运行结果如下:

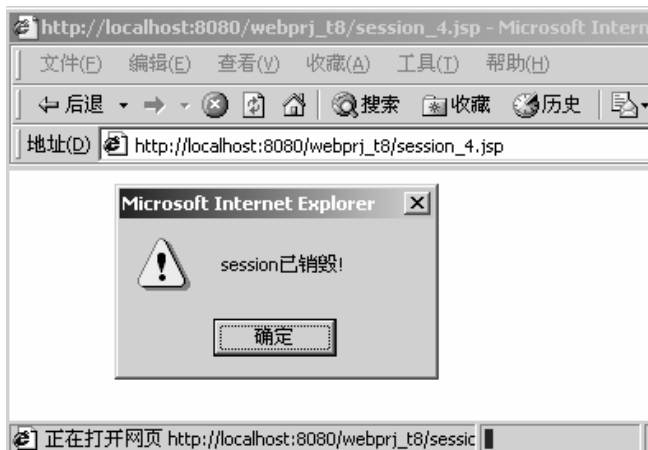


图 7-9 销毁 session

再次访问 session_1.jsp, 从运行结果中的 session 生成时间中可以看到, 此时会重新生成新的 session, 并进行计数器操作, 运行结果如下:

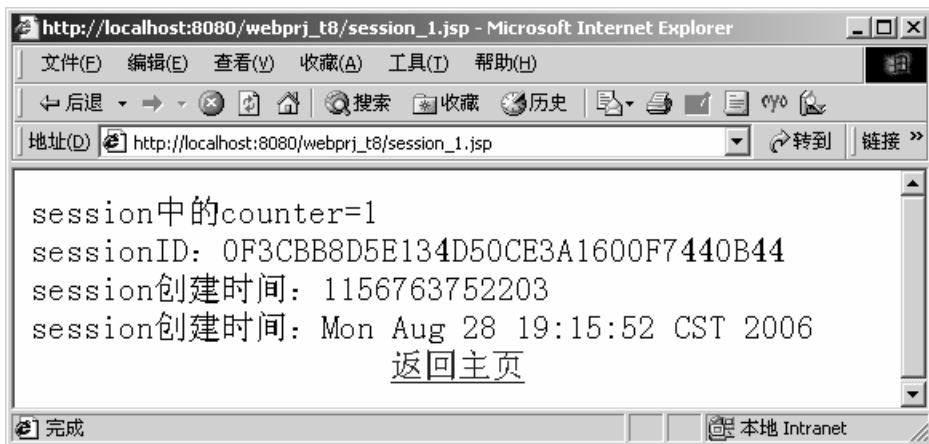


图 7-10 重新访问 session 计数器页, 会生成新的 session 对象

7.3 pageContext 对象

7.3.1 pageContext 对象介绍

pageContext 对象中也可以存储数据，但其中存储的数据作用域是页面级的，即该页运行完毕后数据就会消失。另外在 pageContext 对象中保存了其他几个内置对象的引用，故通过该对象还可以存取其他作用域中的数据：application、session、request。

pageContext 对象的重要方法列表如下：

方法	功能描述
setAttribute(String,Object)	将对象用指定键名存储到 pageContext 对象中
getAttribute(String)	返回指定名字的属性值，如果不存在返回 null
setAttribute(String,Object,int scope)	将对象用指定键名存储到指定作用域对象中，这些作用域对象可以是：application、session、request、page
getAttribute(String,Object,int scope)	从指定作用域对象中读取指定键名的数据
forward(String url)	将请求转发到指定 url
include(String url)	包含指定 url 的运行结果
findAttribute(String)	在四种作用域中搜索指定名字的数据

表 7-3 pageContext 的重要方法

7.3.2 pageContext 对象示例

1、 示例 1——页面计数器

本示例演示如何在 pageContext 中存储数据。

1) JSP 页 pageContext_1.jsp 内容如下：

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

<%
    int n=0;
    Object obj=pageContext.getAttribute("counter");

    if(obj==null){
        //如果没取到计数值，则将计数值设为 1
        n=1;
        pageContext.setAttribute("counter", new Integer(n) );
    }else{
        //如果取到计数值，则将计数值增 1，再存回 pageContext
        n=((Integer)obj).intValue()+1;
        pageContext.setAttribute("counter", new Integer(n) );
    }
%>

pageContext 中的 counter=<%=n%><br>
```

```
<center>
    <a href="pagecontext_1.jsp">刷新本页</a> |
    <a href="index.html">返回主页</a>
</center>
```

- 2) 说明: 该 JSP 页面运行时, 会保存页面级数据 `counter`, 该数据在页面运行结束时会自行消失, 所以多次刷新本页看到的结果一直是: 1。

2、 示例 2——通过 `pagecontext` 向其他作用域中存储数据

1) JSP 页 `pagecontext_2.jsp` 内容如下:

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>
通过 pageContext 对象, 可以向四个作用域中存储数据: <br>
这 4 个作用范围由大到小排列如下: <br>
application、session、request、pageContext<br>

<%
    pageContext.setAttribute("username", "tom_application",
pageContext.APPLICATION_SCOPE);
    pageContext.setAttribute("username", "tom_session", pageContext.SESSION_SCOPE);
    pageContext.setAttribute("username", "tom_request", pageContext.REQUEST_SCOPE);

    pageContext.setAttribute("username", "tom_page", pageContext.PAGE_SCOPE);
%>

<hr>
读取数据如下: <br>
application 中的 username:<%= (String) application.getAttribute("username") %><br>
session 中的 username:<%= (String) session.getAttribute("username") %><br>
request 中的 username:<%= (String) request.getAttribute("username") %><br>
pageContext 中的 username:<%= (String) pageContext.getAttribute("username") %><br>

<center>
    <a href="index.html">返回主页</a>
</center>
```

2) 访问 pageContext_2.jsp, 运行结果如下:

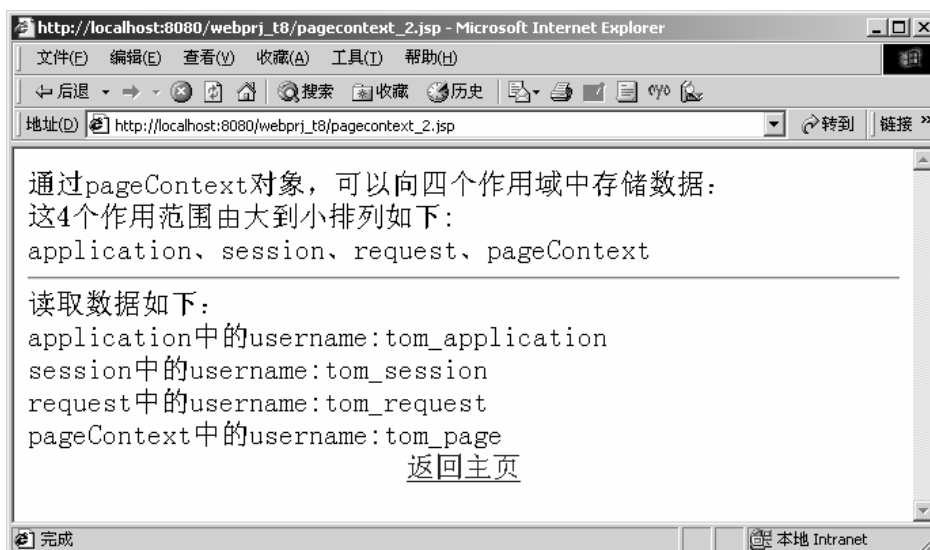


图 7-11 pagecontext_2.jsp 运行结果

3、 示例 3——通过 pageContext 读取其他作用域中的数据

1) JSP 页 pageContext_3.jsp 内容如下:

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

通过 pageContext 对象, 可以从四个作用域中读取数据: <br>

开始存储数据. <br>
<%
    pageContext.setAttribute("username", "tom_page");
    request.setAttribute("username", "tom_request");
    session.setAttribute("username", "tom_session");
    application.setAttribute("username", "tom_application");
%>

<hr>
读取数据如下: <br>

application 中的 username:
<%= (String)pageContext.getAttribute("username", pageContext.APPLICATION_SCOPE) %><br>
session 中的 username:
<%= (String)pageContext.getAttribute("username", pageContext.SESSION_SCOPE) %><br>
```

```

request 中的 username:
<%= (String)pageContext.getAttribute("username",pageContext.REQUEST_SCOPE)%><br>
pageContext 中的 username:
<%= (String)pageContext.getAttribute("username",pageContext.PAGE_SCOPE)%><br>

<center>
    <a href="index.html">返回主页</a>
</center>

```

2) 访问 pageContext_3.jsp, 运行结果如下:

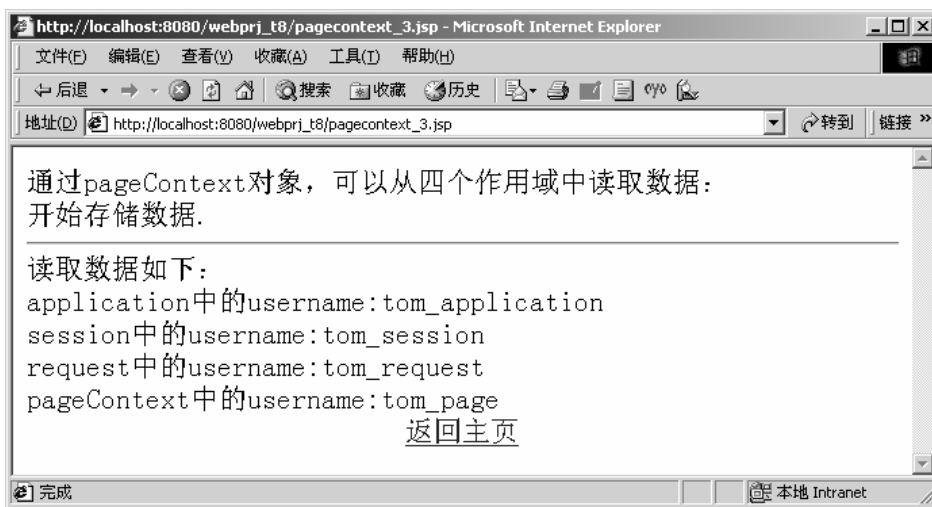


图 7-12 pageContext_3.jsp 运行结果

4、 示例 4——请求转发和请求包含

1) JSP 页 pageContext_4.jsp 内容如下, 运行本页请求将被转发到 pagecontext.jsp:

```

<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%@ page import="javax.servlet.jsp.PageContext" %>
开始转发:<br>
<%
    pageContext.setAttribute("username","tom_page");
    request.setAttribute("username","tom_request");
    session.setAttribute("username","tom_session");
    application.setAttribute("username","tom_application");
    pageContext.forward("pagecontext.jsp");//请求转发
%>

```

2) JSP 页 pagecontext.jsp 内容如下, 该页读取转发过来的数据:

```

<%@ page language="java" contentType="text/html; charset=gb2312" %>
通过 pageContext 对象，可以从四个作用域中读取数据: <br>
<hr>

```

读取数据如下:


```

application                                中                                的
username:<%= (String)application.getAttribute("username") %><br>
session 中的 username:<%= (String)session.getAttribute("username") %><br>
request 中的 username:<%= (String)request.getAttribute("username") %><br>
pageContext                                中                                的
username:<%= (String)pageContext.getAttribute("username") %><br>
<center>
    <a href="index.html">返回主页</a>
</center>

```

3) 说明:

- ◆ 在 pageContext_4.jsp 中向四个作用域 (application、session、request、page) 中分别存储了数据, 这些数据中除了 page 级数据外, 其他三项数据都能转发到目标页面 pagecontext.jsp。
- ◆ 如果要实现请求包含, 则将语句 pageContext.forward("pagecontext.jsp"); 改为 pageContext.include("pagecontext.jsp")即可。

7.4 Cookie 对象

cookie 的概念在第四章已经有过详细讲解, 在 Servlet 中可以借助 cookie 实现会话管理, 同样在 JSP 中也可以使用 Cookie, 方法与 Servlet 中相同。但 Cookie 在 JSP 中没有被实现成内置对象, 要得到 Cookie 必须通过 request 对象的 getCookies() 方法得到 Cookie 数组, 然后再循环读取。

7.4.1 Cookie 使用示例

1、 示例 1, 该示例演示如何在 JSP 中使用 Cookie

1) JSP 页 cookie_1.jsp 内容如下, 该页面实现写 Cookie 操作:

```

<%@ page language="java" contentType="text/html; charset=gb2312" %>
<%
    Cookie cookie=new Cookie("nick","tom1");
    response.addCookie(cookie);
%>
<h3>Cookie 存储完毕!</h3>
<center>
    <a href="index.html">返回主页</a>
</center>

```

2) JSP 页 cookie_2.jsp 内容如下, 该页面实现读 Cookie 操作:

```

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

<%
    Cookie cookies[] = request.getCookies();

```

```
//得到 cookie,并判断用户是否登录过
String nick="";
if(cookies!=null)
{
    for(int i=0;i<cookies.length;i++)
    {
        Cookie cookie=cookies[i];
        if(cookie.getName().equals("nick"))
        {
            nick=cookie.getValue();//得到 cookie 的值
            break;
        }
    }
}

%>
读取的 Cookie 值: <br>
nick:<%=nick%>

<center>
    <a href="index.html">返回主页</a>
</center>
```

3) 两个页面运行结果如下:

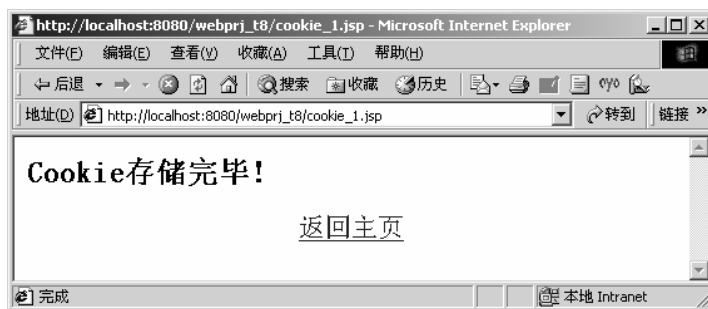


图 7-13 cookie_1.jsp 结果



图 7-14 cookie_2.jsp 结果

7.5 config 对象

config 是 `javax.servlet.ServletConfig` 接口的对象。该对象通常用于读取 servlet 的初始化参数值，在 JSP 中很少用到。该对象的方法与用法与第三章 `ServletConfig` 的用法完全相同。

小结

applicaton 是 `javax.servle.ServletContext` 接口的对象，可以在该对象中保存全局可共享的数据信息。

在 session 对象中可以存储用户级的数据，这些数据可以在同一会话中的任何 JSP 页中访问到。

pageContext 对象可以存储页面级的数据，该页运行完毕后数据就会消失。

pageContext 对象中保存了其他几个内置对象的引用，故通过该对象还可以存取其他作用域中的数据：application、session、request。

在 JSP 中也可以使用 Cookie，但 Cookie 在 JSP 中没有被实现成内置对象，要在 JSP 中使用 Cookie 必须通过 request 对象的 `getCookies()` 方法得到 Cookie 数组，然后再循环读取。

config 是 `javax.servlet.ServletConfig` 接口的对象。该对象通常用于读取 servlet 的初始化参数值。

课后练习

- 1、 JSP 中存储数据的对象有几种，分别有何种作用域？
- 2、 `pageContext` 对象中的 `findAttribute()` 方法有什么作用？

第八章 JSP 动作

概要

JSP 动作也称 JSP 动作标记，JSP 动作利用 XML 语法格式的标记可以在 Web 服务器中完成某些指定功能，如：利用 JSP 动作可以动态地插入文件、可以建立和使用 JavaBean 对象、还可以对请求进行处理。在 JSP 页中使用 JSP 动作可以简化 JSP 页面的开发与维护，有利于业务逻辑与显示的分离。本章主要介绍 JSP 标准动作的使用方法。

目标

- JSP 动作的概念（理解）
- `<jsp:include>`动作（掌握）
- `<jsp:forward>`动作（掌握）
- 操作 javaBean 的动作（掌握）

目录

- 8.1 `<jsp:include>`动作
- 8.2 `<jsp:forward>`和`<jsp:param>`动作
- 8.3 `<jsp:useBean>`动作
- 8.4 `<jsp:setProperty>`动作
- 8.5 `<jsp:getProperty>`动作

8.1 include 动作

8.1.1 JSP 动作概念

JSP 动作是一种 XML 格式的标记，与 HTML 客户端标记不同，每一个 JSP 动作标记都可以在服务器端完成某种对应的功能。通过在 JSP 页面中使用动作标记，可以将页面中更多的代码和业务逻辑隐藏到后台，从而减少 JSP 页中代码段的数量，这样页面开发者在对 JSP 页进行处理时不用担心意外修改 JSP 代码段，即业务可以与显示分离的更清晰。

JSP 动作有标准动作和自定义动作之分，自定义标记在后续章节中有专门讲解，标准动作是 Servlet 容器本身自支持的标记，这些标记以 “<jsp:” 开头，标准动作标记共有以下几种：

JSP 标准标记	功能描述
<jsp:include>	将指定 JSP 页或 Servlet 的运行结果嵌入到本页面中
<jsp:forward>	进行请求转发，与 <code>pageContext.forward()</code> 方法功能相同
<jsp:useBean>	生成或查询 javabean 对象
<jsp:setProperty>	设置 javabean 对象的属性
<jsp:getProperty>	读取 javabean 对象的属性

表 8-1 JSP 标准动作标记

8.1.2 include 动作

通过 include 动作，可以将其他 JSP 页或 Servlet 的运行结果包含到本页中来，这是一个动态包含过程，所以不能够包含静态页面。<%@ include file=“xxx” %>指令也是用于包含其他文件的，但该指令包含的是源代码，即以静态方式包含的，静态包含时有可能出现两个 JSP 页中同名变量冲突的问题，但动态包含由于包含的是其他页的运行结果，故不会出现变量名冲突的情况。include 动作的语法如下：

```
<jsp:include page="jsp/servlet" />
```

8.1.3 include 动作示例

1、 示例 1——在 include_1.jsp 中包含 include_2.jsp:

1) include_1.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%!
    //声明方法;
    public String f1(){
        return "include_1.jsp 中的 f1()";
    }
%>
调用 f1():<br>
结果:<%=f1()%>
<br><br>
下面开始包含 include_2.jsp:<br>
<hr color="red">
```

```
<jsp:include page="include_2.jsp" />
<br>
<hr color="red">
```

包含完毕。

2) include_2.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%!
//声明方法;
public String f1(){
    return "include_2.jsp 中的 f1()";
}
%>
<h4>我是 include_2.jsp</h4>
<h4>调用 f1():</h4>
<h4>结果:<%=f1()%></h4>
```

3) 访问 include_1.jsp, 结果如下:

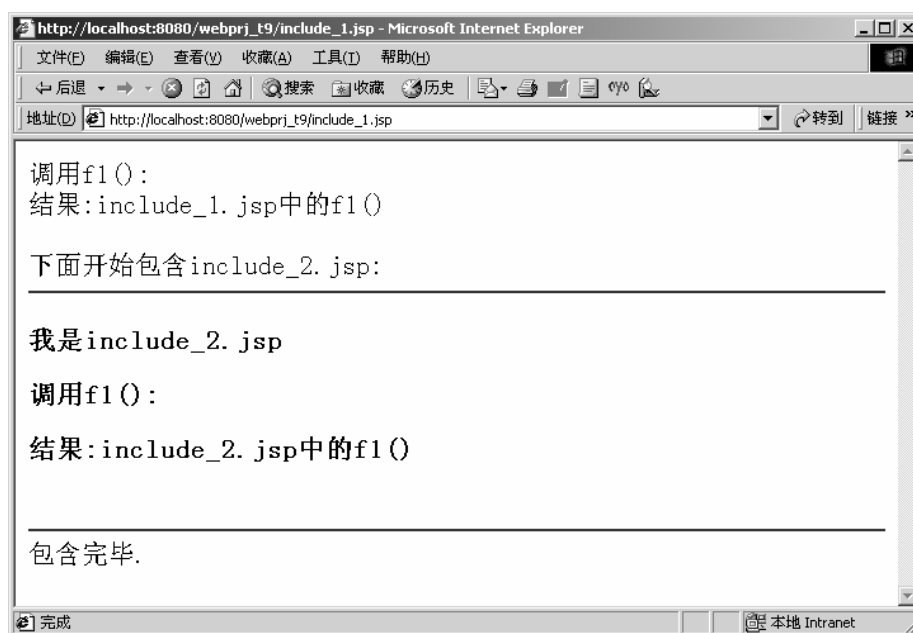


图 8-1 include_1.jsp 运行结果

8.2 forward 动作

8.2.1 forward 动作介绍

通过 forward 动作可以将请求转发到其他页面或 Servlet 中, 以前章节中已经学习过很多请求转发的方法, 但都需要编写代码来实现。如: 通过 JSP 隐含对象 `pageContext.forward()` 方法可以转发、通

过 Dispatcher 对象的 forward()方法可以转发。

通过 forward 动作进行转发可以隐藏以上请求转发时所编写的代码，forward 动作语法如下：

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

或进行带参数转发：

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

```
    <param ... />
```

```
</jsp:forward>
```

8.2.2 forward 动作示例

1、 示例 1——在请求 request 中存储数据然后转发到其他 JSP 页

1) forward_1.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%
    request.setAttribute("username", "tom1");
%>
<jsp:forward page="forward_2.jsp" />
```

2) forward_2.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
这是 forward_2.jsp<br>
由 forward_1.jsp 转发过来.
<h3>开始读取 request 中的数据</h3>
<h3>
username=<%= request.getAttribute("username") %>
</h3>
```

3) 运行 forward_1.jsp, 结果如下:

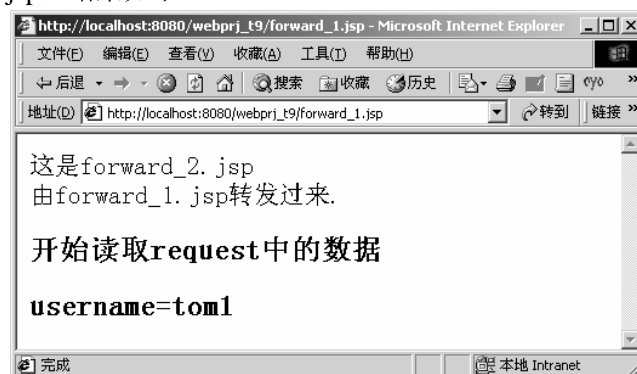


图 8-2 forward_1.jsp 运行结果

4) 说明：在 forward_1.jsp 中使用 request.setAttribute("username","tom1")语句，在请求中存储了数据，然后转发到 forward_2.jsp 中进行读取操作，请求中的数据被转发到第二个页了，如果换成请求重定向则 request 中的数据在第二个页中访问不到。

2、 示例 2——带参数的请求转发

1) include_3.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%--
<jsp:param> 相当于表单中的一个元素;
--%>
<jsp:forward page="forward_4.jsp">
    <jsp:param name="username" value="tom1" />
    <jsp:param name="userpass" value="1234" />
</jsp:forward>
```

2) forward_4.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
<h3>forward_4.jsp</h3>
<h3>开始读取转发过来的参数值:</h3>
username:<%=request.getParameter("username") %> <br>
userpass:<%=request.getParameter("userpass") %> <br>
```

3) 访问 forward_3.jsp, 运行结果如下:

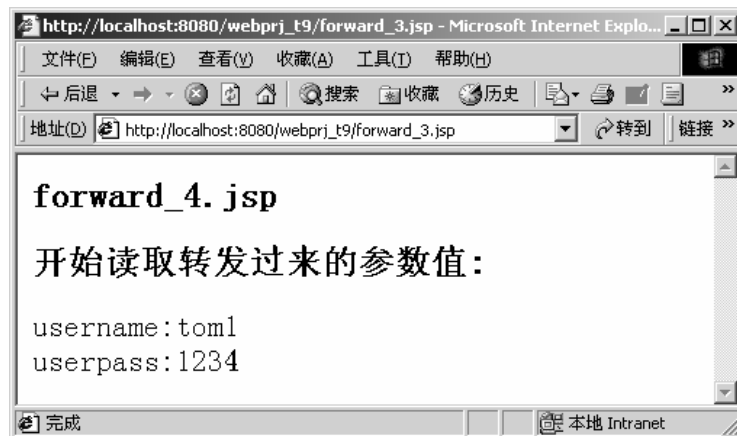


图 8-3 forward_3.jsp 运行结果

- 4) 说明: 在<jsp:forward>标记中间可以嵌入<jsp:param>标记进行带参数的请求转发, 可以将<param>中的子元素当作表单中的元素看待, 则这些参数在第二个页中使用“request.getParameter()”读取到。

8.3 操作 javabeen 的动作

8.3.1 bean 相关动作介绍

JSP 要完成某些业务逻辑操作时, 通常是通过 javabeen 来实现, 生成 javabeen 的对象以及读写 javabeen 的属性, 可以在 JSP 的代码段中通过代码来完成。也可以通过动作标记来代替这些代码。

bean 相关动作标记共有三个, 语法分别如下:

- 1) <jsp:useBean> 创建一个 Bean 实例并指定它的名字和作用范围。

语法:

```
<jsp:useBean
id="beanInstanceName"
scope="page | request | session | application"
class="package.class" |
type="package.class" |
/>
</jsp:useBean>
```

说明: id——javabean 的名字; scope——javabean 要存储的范围; class——javabean 所属的类; type——id 的类型, 即引用的类型, 它可以是 javabean 的任何父类或本类型。

<jsp:setProperty>动作用于向一个 javabean 的属性赋值, 需要注意的是, 在这个动作中将会使用到的 name 属性的值将是一个前面已经使用<jsp:useBean>动作引入的 javabean 的名字。

语法: <jsp:setProperty name="id" property="propertyname" />

<jsp:getProperty>动作用于从一个 javabean 中得到某个属性的值, 无论原先这个属性是什么类型的, 都将被转换为一个 String 类型的值。

语法: <jsp:getProperty name="id" property="propertyname" />, 不论属性是什么类型, 该标记都会将属性值转成 String 类型。

8.3.2 bean 相关动作示例

1、 示例 1

本示例演示如何使用动作生成 javabean 对象, 及如何为其属性赋值。

1) javabean 内容如下:

```
package demo;

public class User implements java.io.Serializable{
    private String username;
    private String userpass;

    public User(){
        username=" ";
        userpass=" ";
    }

    public void setUsername(String username){
        this.username=username;
    }
    public void setUserpass(String userpass){
        this.userpass=userpass;
    }
    public String getUsername(){
```



```

        return this.username;
    }
    public String getUserpass(){
        return this.userpass;
    }
}

```

2) bean_1.jsp 如下，本页中实例化了以上 javabean: User 的对象:

```

<%@ page contentType="text/html; charset=gb2312" %>
<jsp:useBean id="user1" class="demo.User" />
<h3>user1 对象实例化完成</h3>
<h3>开始设置 user1 对象的属性</h3>
<jsp:setProperty name="user1" property="username" value="张三"/>
<jsp:setProperty name="user1" property="userpass" value="1234"/>

<h3>开始读取 user1 对象的属性</h3>
用户名:<jsp:getProperty name="user1" property="username" /><br>
用户密码:<jsp:getProperty name="user1" property="userpass" /><br>

```

3) 访问 bean_1.jsp, 结果如下:

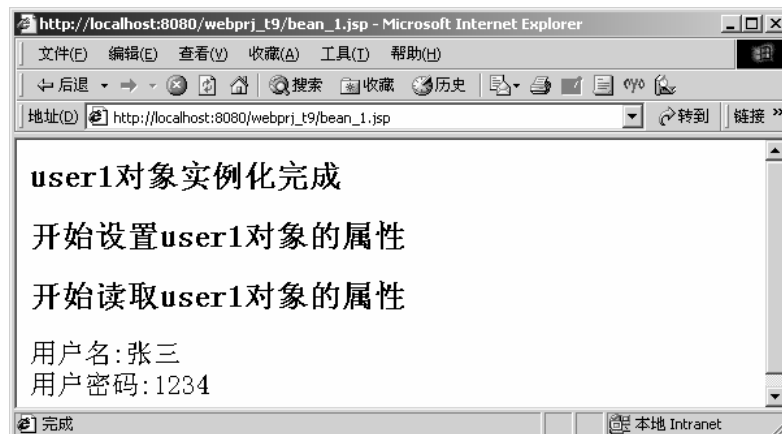


图 8-4 bean_1.jsp 运行结果

2、 示例 2——通过表单为 bean 属性赋值

1) bean_2.html 中提供了表单:

```

<h3>用户注册</h3>

<form action="bean_2.jsp" method="post">
    用户名:<input size="10" name="username"><br>

```

```
    用户密码:<input size="10" name="userpass"><br>
    <input type="submit" value="注册">
</form>
```

2) 在 bean_2.jsp 中接收以上表单数据, 并赋给 javabean 对象:

```
<%@ page contentType="text/html; charset=gb2312" %>
<jsp:useBean id="user1" class="demo.User" />
<h3>user1 对象实例化完成</h3>
<%
    request.setCharacterEncoding("gb2312");
%>

<h3>开始用表单元素值设置 user1 对象的属性</h3>
<jsp:setProperty name="user1" property="username" />
<jsp:setProperty name="user1" property="userpass" />

<h3>开始读取 user1 对象的属性</h3>
用户名:<jsp:getProperty name="user1" property="username" /><br>
用户密码:<jsp:getProperty name="user1" property="userpass" /><br>
```

3) 运行 bean_2.html, 结果如下

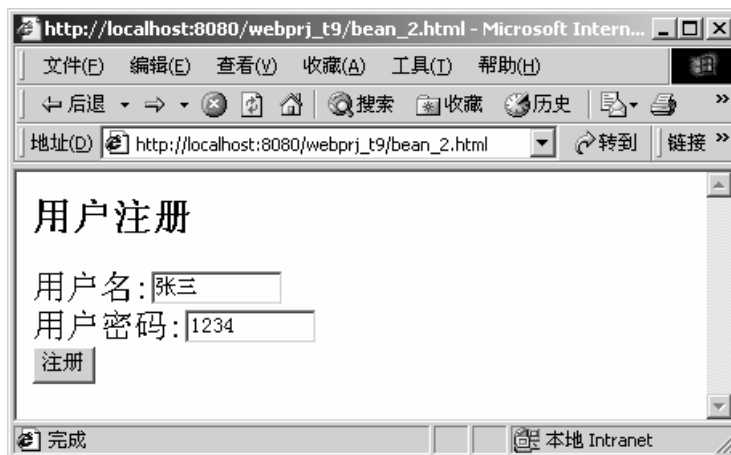


图 8-5 bean_2.html

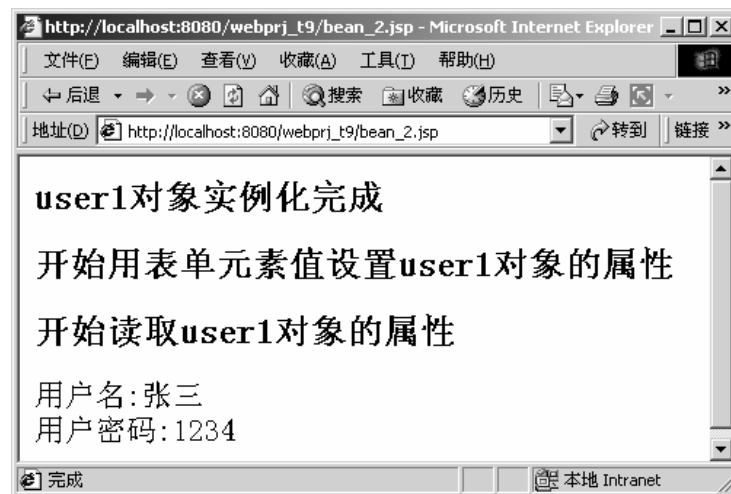


图 8-6 bean_2.jsp

说明:

- ◆ 在 JSP 页面中使用语句 `<jsp:setProperty name="user1" property="username" />` 可以接收表单中的元素值。但要求表单中的元素与 jsp 动作标记中指定的属性同名。
- ◆ 在 bean_2.html 中两个元素的名称分别为: username、userpass, 与 javabean 属性同名, 会被 `<jsp:setProperty>` 自动识别并接收。

3、 示例 3——通过表单为 javabean 赋值方法 2

1) 表单如下:

```
<h3>用户注册</h3>
<form action="bean_3.jsp" method="post">
    用户名:<input size="10" name="username"><br>
    用户密码:<input size="10" name="userpass"><br>
    <input type="submit" value="注册">
</form>
```

2) bean_3.jsp 页如下, 本页运行效果与图 8-6 完全相同:

```
<%@ page contentType="text/html; charset=gb2312" %>
<jsp:useBean id="user1" class="demo.User" />
<h3>user1 对象实例化完成</h3>
<%
    request.setCharacterEncoding("gb2312");
%>
<h3>开始用表单元素值设置 user1 对象的属性</h3>
<jsp:setProperty name="user1" property="*" />
<h3>开始读取 user1 对象的属性</h3>
用户名:<jsp:getProperty name="user1" property="username" /><br>
用户密码:<jsp:getProperty name="user1" property="userpass" /><br>
```

3) 说明:

- ◆ 在 JSP 页面中使用语句<jsp:setProperty name="user1" property="*" />可以接收表单中的所有元素值,然后 jsp 动作会自动将与 javabeen 属性同名的元素赋给 javabeen 的相应属性。
- ◆ 适用范围: 表单中元素与 javabeen 属性同名时,使用该方法可以自动进行赋值操作,比较方便,但是当表单中各元素名称与 javabeen 属性不同名时则不可用。

4、 示例 4——通过表单为 javabeen 赋值方法 3

1) 表单如下:

```
<h3>用户注册</h3>
<form action="bean_4.jsp" method="post">
  用户名:<input size="10" name="name"><br>
  用户密码:<input size="10" name="pass"><br>
  <input type="submit" value="注册">
</form>
```

2) bean_4.jsp 如下, 程序运行效果同图 8-6:

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

<jsp:useBean id="user1" class="demo.User" />
<h3>user1 对象实例化完成</h3>
<%
    request.setCharacterEncoding("gb2312");
%>

<h3>开始用表单元值设置 user1 对象的属性</h3>
<jsp:setProperty name="user1" property="username" param="name"/>
<jsp:setProperty name="user1" property="userpass" param="pass"/>

<h3>开始读取 user1 对象的属性</h3>
用户名:<jsp:getProperty name="user1" property="username" /><br>
用户密码:<jsp:getProperty name="user1" property="userpass" /><br>
```

3) 说明:

- ◆ 表单中两个元素名称分别为: name、pass, 而 javabeen 的两个属分别为: username、userpass;
- ◆ 语句: <jsp:setProperty name="user1" property="username" param="name"/>中, property 用于指定要将值赋给 JAVABEAN 的哪个属性, param 指定表单中的元素, 该元素的值

会被赋给 property 指定的 javabean 属性。

5、 示例 5——控制 bean 的存储范围

本示例中通过 JSP 动作生成 JAVABEAN 的对象，并通过动作的 scope 属性将该对象存储到 SESSION 范围内，在其他页面中可以从 SESSION 中读取到该 BEAN 对象。

1) bean_scope_1.jsp, 实例化 bean 并将其存储到 session:

```
<%@ page contentType="text/html; charset=gb2312" %>
<jsp:useBean id="user1" class="demo.User" scope="session" />
<jsp:setProperty name="user1" property="username" value="张三"/>
<jsp:setProperty name="user1" property="userpass" value="1234"/>
<h3>user1 已经存储到 session 中</h3>
<a href="bean_scope_2.jsp">
bean_scope_2.jsp(到其他页中搜索 session 中的 user1)
</a>
```

2) bean_scope_2.jsp, 读取 session 中的 bean:

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

<h3>在 session 中搜索 user1 对象</h3>
<jsp:useBean id="user1" class="demo.User" scope="session"/>

<h3>开始读取 user1 对象的属性</h3>
用户名:<jsp:getProperty name="user1" property="username" /><br>
用户密码:<jsp:getProperty name="user1" property="userpass" /><br>
```

3) 运行 bean_scope_1.jsp:

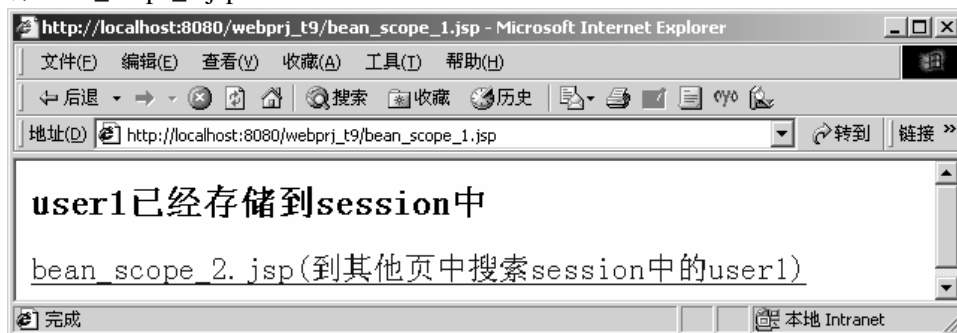


图 8-7 bean_scope_1.jsp

点击浏览器中的超链接，转到 bean_scope_2.jsp:



图 8-8 bean_scope_2.jsp

4) 说明:

- ◆ `<jsp:userBean>` 的 `scope` 属性可以指明一个作用域 (`page`、`session`、`request`、`application`)，该动作先到指定作用域中搜索指定名字的 `javabeen` 实例，如果找到则取出，如果找不到则生成新的实例存储到指定作用域中。

小结

通过在 JSP 页面中使用动作标记，可以将页面中更多的代码和业务逻辑隐藏到后台，从而将业务与显示分离的更清晰。

`<jsp:include>` 动作，可以将其他 JSP 页或 Servlet 的运行结果包含到本页中来。

`<jsp:forward>` 动作可以将请求转发到其他页面或 Servlet 中。

`<jsp:userBean>` 动作用于生成 `javabeen` 的对象，`<jsp:setProperty>` 和 `<jsp:getProperty>` 用于读写 `javabeen` 的属性值。

课后练习

- 1、 举例说明何时应该使用 `jsp` 动作标记？
- 2、 简单说明操作 `javaBean` 的三个动作标记的用法？

第九章 JSP2.0 特征

概要

JSP2.0 版是对 JSP1.2 的升级，JSP2.0 最主要的新特性包括：JSP 表达式语言 EL、简单标记类 SimpleTag。JSP 标准标记库（JSTL）是一个由 apache 的 jakarta 小组不断完善的开源代码的 JSP 标签库，使用 JSTL 可以简化 JSP 页面的开发，在 JSP2.0 中使用 JSTL 更方便。本章主要介绍 JSP2.0 中的新特征及 JSTL 中的核心标签的用法。

目标

- JSP2.0 简介（了解）
- EL 表达式语言（掌握）
- JSP2.0 中的自定义标记（了解）
- JSTL 核心标签库（掌握）

目录

- 9.1 JSP2.0 简介
- 9.2 EL 表达式语言
- 9.3 JSP2.0 中的自定义标记
- 9.4 JSTL 核心标签库

9.1 JSP2.0 简介

JSP2.0 版是对 JSP1.2 的升级，增加了一些新特性。JSP2.0 的目标是：

- 使动态网页的设计更加容易
- 简化 JSP 页面，使得 JSP 页面容易维护
- 使 WEB 应用程序前后台应用更清晰

具体来说，JSP2.0 引入的最主要的新特性包括：

- 2) 一种简单的表达式语言（EL），能够用来容易地从 JSP 页面访问数据，使用这种表达式语言在 JSP 中可以替代 Java 代码段或者 Java 表达，从而减少页面中的代码；

更简单的开发自定义标记的方法：继承 SimpleTag、开发.tag 的标签文件；

在使用 JSP2.0 时，必须注意所选择的 WEB 服务器是否支持该 JSP 版本，Tomcat5.0 以上版本都是支持 JSP2.0 的。

9.2 EL 表达式语言

EL(表达式语言)，全名为 Expression Language，是一种数据访问语言，可以方便地访问和处理应用程序数据，而无需使用 JSP 代码段或 JSP 表达式进行编程(即不需要使用<% 和%>来获得数据)，EL 使 JSP 页面编写人员摆脱了 java 语言。

9.2.1 EL 语法

EL 主要的语法结构：\${10+1}

所有 EL 都是以 \${ 为起始、以 } 为结尾的。以上表达式的意思是计算 10+1 的值。

- . 点运算符、[] 运算符
- 点运算符和[]用以读取 javabean 的属性值，假设有以下代码：

```
User user=new User();
user.setUsername(username);
user.setUserpass(userpass);
session.setAttribute("user",user);
```

则，以下两行代码都是读取 session 中的 user 对象属性中的值：

```
${sessionScope.user.username}
${sessionScope.user["username"]}
```

- 关于[]运算符

当要存取的属性名称中包含一些特殊字符，如 . 或 - 等并非字母或数字的符号，就一定要使用 []，例如：

`${user.e-mail}` 是不正确的方式，应当改为：`${user["e-mail"]}`

- EL 变量

EL 存取变量数据的方法很简单，例如：`${username}`。它的意思是取出某一范围中名称为 username 的变量。因为我们并没有指定哪一个范围的 username，所以它的默认值会先从 Page 范围找，假如找不到，再依序到 Request、Session、Application 范围搜索，如找到直接返回，全部的范围都没有找到时，就返回 null。

- 也可以指定要取出哪一个范围的变量，例如：

- `${pageScope.username}` —— 取出 page 范围的 username 变量
- `${requestScope.username}` —— 取出 request 范围的 username 变量
- `${sessionScope.username}` —— 取出 session 范围的 username 变量
- `${applicationScope.username}` —— 取出 application 范围的 username 变量

➤ EL 保留字

在为变量命名时要避免使用 EL 保留字，EL 保留字如下：

And、eq、gt、true、Or、ne、le、false、No、lt、ge、null、instanceof、empty、div、mod

➤ EL 隐含对象

EL 隐含对象是指可以在 EL 表达式中直接使用的对象，共有 11 个，分为以下几类：

与范围有关的隐含对象

名称	含义
applicationScope	取得 Application 范围的属性名称所对应的值
sessionScope	取得 session 范围的属性名称所对应的值
requestScope	取得 request 范围的属性名称所对应的值
pageScope	取得 page 范围的属性名称所对应的值

表 9.1 与范围有关的 EL 隐含对象

与输入有关的隐含对象

名称	含义
param	功能同 <code>ServletRequest.getParameter(Stringname)</code>
paramValues	功能同 <code>ServletRequest.getParameterValues(Stringname)</code>

表 9.2 与输入有关的 EL 隐含对象

其他隐含对象

名称	含义
cookie	功能同 <code>HttpServletRequest.getCookies()</code>
header	功能同 <code>ServletRequest.getHeader(Stringname)</code>
iParam	功能同 <code>ServletContext.getInitParameter(Stringname)</code>
pageContext	表示本 JSP 的 pageContext
headerValues	功能同 <code>ServletRequest.getHeaders(Stringname)</code>

表 9.3 其他 EL 隐含对象

EL 隐含对象的使用方法如下：

在 session 中储存了一个属性，它的名称为 username，在 JSP 中使用 `session.getAttribute("username")` 可以取得 username 的值，但是在 EL 中则可用 `${sessionScope.username}` 来取得。同理，有以下用法：

- `${param.username}` 代表表单中的 username 元素的值
- `${pageContext.username}` 代表 page 范围中的 username 属性的值

- `${pageContext.request.method}` 取得 HTTP 的方法(GET、POST)
- `${pageContext.request.remoteAddr}` 取得用户的 IP 地址

- EL 运算符，如下面表格所示

算术运算符	范 例	结 果
+	<code>\${ 5 + 5 }</code>	10
-	<code>\${ 10 - 5 }</code>	5
*	<code>\${ 5 * 5 }</code>	25
/或 div	<code>\${ 10 / 5 }</code> 或 <code>\${ 10 div 5 }</code>	2
%或 mod	<code>\${ 8 % 5 }</code> 或 <code>\${ 8 mod 5 }</code>	3

表 9.4 EL 算术运算符

关系运算符	含义	逻辑运算符	含义
<code>==</code> 或 <code>eq</code>	等于	<code>&&</code> 或 <code>and</code>	与
<code>!=</code> 或 <code>ne</code>	不等于	<code> </code> 或 <code>or</code>	或
<code><</code> 或 <code>lt</code>	小于	<code>!</code> 或 <code>not</code>	非
<code>></code> 或 <code>gt</code>	大于		
<code><=</code> 或 <code>le</code>	小于等于		
<code>>=</code> 或 <code>ge</code>	大于等于		

表 9.5 EL 关系及逻辑运算符

9.2.2 EL 示例

1) el_3.jsp

```

<%@ page contentType="text/html; charset=gb2312" %>
<%
    request.setAttribute("nick", "tom_request");
    pageContext.setAttribute("nick", "tom_page");
%>
作用域范围中的数据: <br>
\${requestScope.nick} : ${requestScope.nick} <br>
\${pageScope.nick} : ${pageScope.nick} <br>
<br>
请求参数中的数据: <br>
\${param.nick} : ${param.nick} <br>

```

运行效果如下:

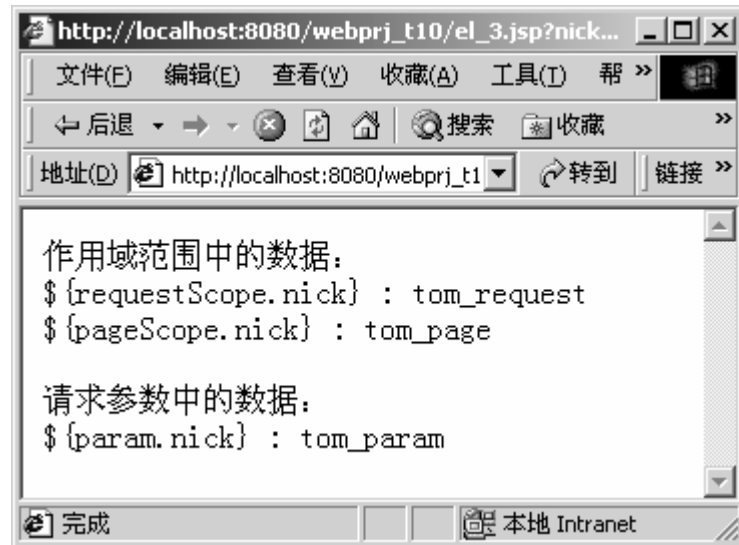


图 9-1 使用 EL 表达式

2) el_1.jsp

```

<tr>
    <td>\${1 + 2}</td>
    <td>${1 + 2}</td>
</tr>
<tr>
    <td>\${1.2 + 2.3}</td>
    <td>${1.2 + 2.3}</td>
</tr>
<%
    request.setAttribute("username", "tom");
%>
<TR>
    <TD>empty username</TD>
    <TD>${empty username}</TD>
</TR>

```

说明: \\${1+2} 代表原样显示 “\${1+2}”, 而不是作为表达式运行。

3) 一个综合例子, 在 JSP 中使用 EL 接收表单数据:

表单内容:

```

<form method="post" action="reg.jsp">
用户名:<input type="text" name="username" size="15" /><br>
密码:<input type="password" name="password" size="15" /><br>
性别:<input type="radio" name="sex" value="1" checked/>男

```

```

<input type="radio" name="sex" value="0" />女<br>
年龄:<select name="old">
    <option value="10">10 - 20</option>
    <option value="20" selected>20 - 30</option>
    <option value="30">30 - 40</option>
    <option value="40">40 - 50</option>
</select><br>
爱好:
<input type="checkbox" name="habit" value="睡觉"/>睡觉
<input type="checkbox" name="habit" value="阅读"/>阅读
<input type="checkbox" name="habit" value="跑步"/>跑步<br>
<input type="submit" value="提交"/>
</form>

```

接收表单数据的 JSP 内容:

```

<%@ page contentType="text/html; charset=gb2312" %>
<%
    request.setCharacterEncoding("gb2312");
%>
用户名:${param.username}</br>
密码:${param.password}</br>
性别:${param.sex}</br>
年龄:${param.old}</br>
爱好:${paramValues.habit[0]}

```

运行效果如下:



图 9-2 填写表单



图 9-3 用 EL 接收表单数据

9.3 JSP2.0 中的自定义标记

JSP 2.0 中提供了两种新的开发自定义标记的方法:

1) 简单标签机制 SimpleTag

JSP 2.0 中加入了新的创建自定义标记的 API: `javax.servlet.jsp.tagext.SimpleTag`, 该 API 定义了用来实现简单标记的接口。和 JSP 1.2 中的已有接口不同的是, `SimpleTag` 接口不使用 `doStartTag()` 和 `doEndTag()` 方法, 而提供了一个简单的 `doTag()` 方法。这个方法在调用该标记时只被使用一次。一个自定义标记中实现的所有逻辑都在这个方法中实现。相对 JSP1.2 中自定义标记机制, `SimpleTag` 的方法和处理周期要简单得多。

2) 标签文件

标签文件允许 JSP 网页作者使用 JSP 语法创建可复用的标签库。标签文件的扩展名必须是 `.tag`。

9.4 使用简单标签机制

与 JSP1.2 相似, 开发自定义标签要遵循“开发标记类---配置 TLD 文件----在 JSP 中使用”的过程, 示例如下:

1) 标记处理类 AddTag.java

```
package demo;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class AddTag extends SimpleTagSupport {
    private int num1 = 0;
    private int num2 = 0;
    public void setNum1(int num1) {
        this.num1 = num1;
    }
    public void setNum2(int num2) {
        this.num2 = num2;
    }
}
```

```

    }

    public void doTag() throws JspException, IOException {
        JspContext ctx = getJspContext();
        JspWriter out = ctx.getOut();
        int sum = num1 + num2;
        out.println(num1 + " + " + num2 + " = " + sum);
    }
}

```

2) 描述符文件 test.tld:

```

.....
<tag>
  <description>Add Tag</description>
  <name>add</name>
  <tag-class>demo.AddTag</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>num1</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
  <attribute>
    <name>num2</name>
    <required>true</required>
    <rtexprvalue>true</rtexprvalue>
  </attribute>
</tag>
.....

```

3) 在 JSP 中使用标记:

```

<%@ page contentType="text/html;charset=gb2312" %>
<%@ taglib uri="/WEB-INF/test.tld" prefix="test" %>
SimpleTag 测试:
<h1><test:add num1="3" num2="6" /></h1>

```

4) 运行效果如下:

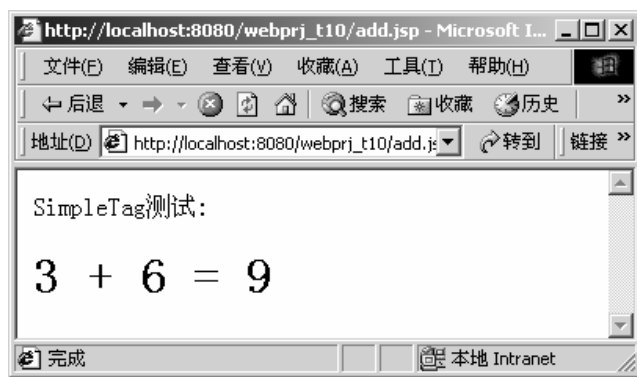


图 9-4 使用简单自定义标记

9.5 使用标签文件

通过标签文件实际上可以将一个 JSP 文件的内容作为标签处理程序，但该文件扩展名必须是.tag，示例如下：

- 1) 标记文件 hello.tag，该文件存放在 WEB-INF\tags 目录下

```
hello.tag.<br>
IP:<%= request.getRemoteAddr() %>
```

- 2) 在 JSP 中使用 tag 文件

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="test" tagdir="/WEB-INF/tags/" %>
<h2>Tag File 测试</h2>
<test:hello/>
```

- 3) 运行效果如下：



图 9-5 使用 Tag 文件作自定义标签

9.6 JSTL

JSTL 全称为 Java Server Pages Standard Tag Library。JSTL 是一个已经被标准化的标记库集合，支持迭代、条件、XML 文档的解析，国际化，和利用 SQL 与数据库交互等功能。

新版本的 JSTL 包括 5 组标记库：core 核心标签库、XML 处理标签库、国际化支持标签、SQL 访问标签、函数标签。这些标签使用简单但功能强大。JSTL 中支持 EL 语言，两者配合使用可以极大

简化在 JSP 中对应用数据的访问和操作。

9.7 核心标签库简介

JSTL 核心标签库(Core)用来实现一些通用的操作,主要有 4 组操作:基本输入输出、流程控制、迭代操作和 URL 操作,每种操作都有相应的标签相对应,核心标签如下表所示:

功能分组	标签
变量定义及输出操作	out、set、remove、catch
流程控制	if、choose、when、otherwise
循环操作	forEach、forEachTokens、Core
URL 操作	import、param、url、param、redirect、param

表 9.6 JSTL 核心标签库

9.8 使用核心标签库

以上 JSTL 核心标签,每个标签有自己的属性,下面通过示例演示各标签的使用方法:

1) 使用变量定义及输出标签:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

输出常量:<c:out value="这是一个常量字符串."/><br>
输出 http 协议头数据:
<c:out value="\${header['User-Agent']}" /><br>
输出自定义变量:
<c:set var="n" value="100"/>
<c:out value="\${n}" />
<br>

<c:set scope="page" var="num1" value="200"/>
<c:set scope="page" var="num">
    <c:out value="\${3+3}" />
</c:set>
<c:set scope="request" var="num">
    <%=30%>
</c:set>
<c:set scope="session" var="num">
4
</c:set>

开始输出定义的变量: <br>
pageScope.num1:<c:out value="\${pageScope.num1}" default="No" /><br>
```

```

pageScope.num:<c:out value="\${pageScope.num}" default="No" /><br>
requestScope.num:<c:out value="\${requestScope.num}" default="No" /><br>
sessionScope.num:<c:out value="\${sessionScope.num}" default="No" /><br>

<br>删除 page 中的 num1:<br>
<c:remove var="num1" scope="page" />
pageScope.num1:<c:out value="\${pageScope.num1}" default="No" /><br><br>

删除 num:<br>
<c:remove var="num"/>
pageScope.num1:<c:out value="\${pageScope.num1}" default="No" /><br>
pageScope.num:<c:out value="\${pageScope.num}" default="No" /><br>
requestScope.num:<c:out value="\${requestScope.num}" default="No" /><br>
sessionScope.num:<c:out value="\${sessionScope.num}" default="No" /><br>
<br>
catch:<br>
<c:catch var="message">
<%
    String str1 = "aa";
    int n = Integer.parseInt(str1);
%>
</c:catch>
message:\${message}

```

运行效果如下:

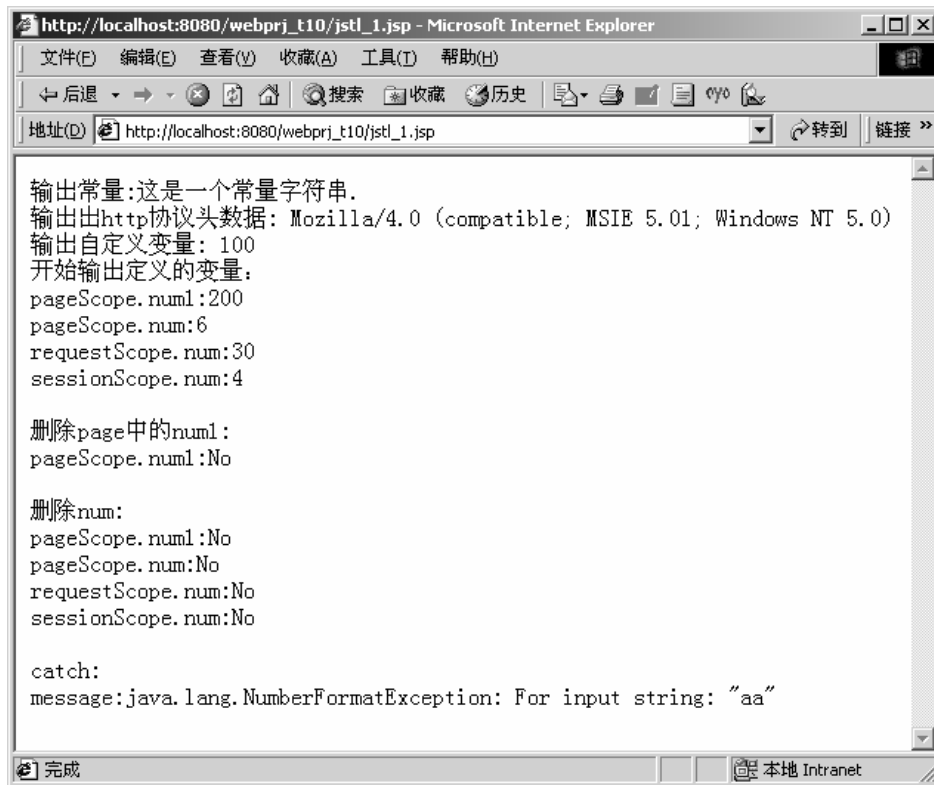


图 9-6 使用核心标签

2) 使用流程控制标签:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%
    request.setAttribute("username", "tom_request");
%>
if 测试:<br>
<c:if test="${requestScope.username == 'tom_request'}" var="result"
scope="page">
用户名是:tom_request!.
</c:if>
</br>
result:${result}<br>
<br><br>
choose、when、otherwise 测试:<br>
<c:set var="n" value="70"/>
<c:choose>
<c:when test="${n<60}">
你的成绩不及格!
</c:when>
```

```

<c:when test="\${n}>60 && n<70}">
你的成绩:中等
</c:when>
<c:when test="\${n}>=70 && n<80}">
你的成绩:良
</c:when>
<c:otherwise>
你的成绩:优
</c:otherwise>
</c:choose>

```

运行效果如下:

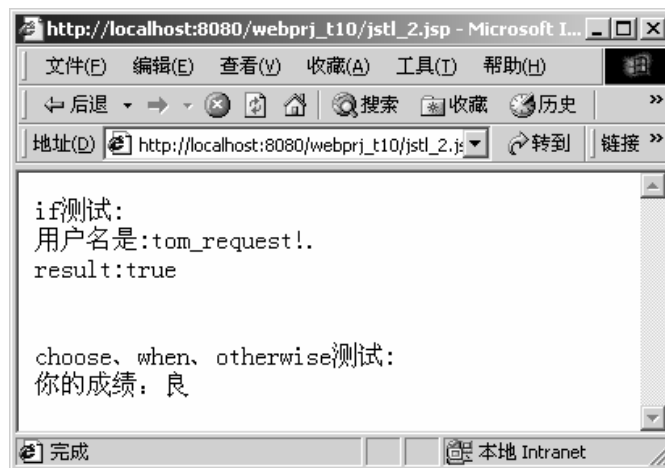


图 9-7 使用流程控制标签

3) 使用循环控制标签:

```

<%@ page contentType="text/html;charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

forEach 测试 1:<br>
<c:forEach begin="0" end="10" var="i" step="2">
count=<c:out value="\${i}"/><br>
</c:forEach>

<br><br>
forEach 测试 2:<br>
<%
    String names[] = new String [3];
    names[0]="张飞";

```

```
names[1]="刘备";
names[2]="关羽";
request.setAttribute("names", names);
%>
<c:forEach items="${names}" var="item" >
${item}</br>
</c:forEach>

<br><br>
forTokens 测试 1:<br>
<c:forTokens items="hello-world-how-are-you" delims="-" var="token">
<c:out value="${token}" />
</c:forTokens>
```

运行效果如下:

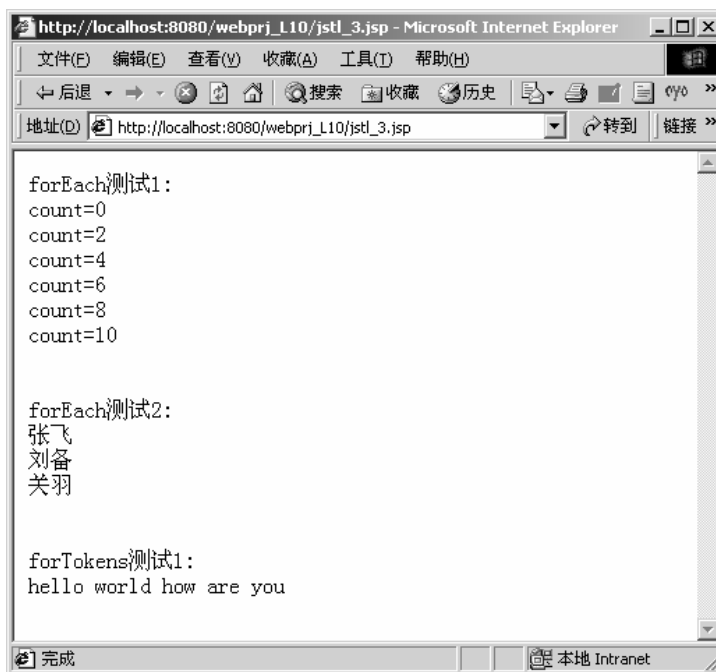


图 9-8 使用循环控制标签

4) 使用 URL 操作标签实现文件包含:

```
<c:import url="test1.jsp">
    <c:param name="nick" value="nick1" />
</c:import>
```

解释：以上代码可以将 test1.jsp 包含到本 JSP 中，名为 nick 的参数可以在 test1.jsp 中访问：

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<h3>test1.jsp</h3>
param.nick:<c:out value="${param.nick}" />
```

5) 使用 URL 操作标签实现重定向：

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:redirect url="test1.jsp">
    <c:param name="nick" value="nick1" />
</c:redirect>
```

小结

JSP2.0 版是对 JSP1.2 的升级，引入的最主要的新特性包括：表达式语言（EL）、更简单的开发自定义标记的方法。

EL(表达式语言)，全名为 Expression Language，是一种数据访问语言，可以方便地访问和处理应用程序数据，而无需使用 JSP 代码段或 JSP 表达式进行编程。

JSTL 全称为 Java Server Pages Standard Tag Library。JSTL 是一个已经被标准化的标记库集合，新版本的 JSTL 包括 5 组标记库：core 核心标签库、XML 处理标签库、国际化支持标签、SQL 访问标签、函数标签。

JSTL 中支持 EL 语言，两者配合使用可以极大简化在 JSP 中对应用数据的访问和操作。

课后练习

- 1、 简述 EL 表达式语言的功能及主要语法？
- 2、 简述 JSTL 核心标签库的四类功能及分别用什么标签来实现？
- 3、 有以下代码：`session.setAttribute("user",user)`，要在 JSP 中显示 user 的 username 属性值，使用 JSTL 如何实现？

第十章 JSTL

概要

JSTL 包括 5 组标记库：`core` 核心标签库、XML 处理标签库、国际化支持标签、SQL 访问标签、函数标签。上一章介绍了核心标签库的使用方法，本章着重讲解其它 4 类标签的使用。

目标

- 使用 JSTL 函数标签（掌握）
- 使用 JSTL-SQL 标签（掌握）
- 使用 JSTL 国际化支持标签（掌握）
- 使用 XML 处理标签（了解）

目录

- 10.1 函数标签
- 10.2 SQL 标签
- 10.3 国际化支持标签
- 10.4 XML 处理标签

10.1 函数标签

JSTL 的函数标签提供了常用的函数功能，使用该类标签时通常使用“fn”为标签前缀，函数标签中主要包含的函数功能如下表所示：

名称	含义
fn:substring	取子字符串
fn:substringAfter	取指定子字符串后边的内容
fn:substringBefore	取指定子字符串后边的内容
fn:contains	判断是否包含指定字符串
fn:containsIgnoreCase	判断是否包含指定字符串，不区分大小写
fn:startsWith	判断字符串是否以指定值开头
fn:endsWith	判断字符串是否以指定值结尾
fn:indexOf	搜索子字符串的位置
fn:join	连接字符串
fn:escapeXml	过滤字符串格式

表 10.1 函数标签

以上各标签分别有自己的属性，通过为属性赋值可以决定要完成的功能。下面通过实例的形式来研究各标签的用法：

10.1.1 子字符串操作

本示例演示如何通过函数标签取子字符串、判断是否存在指定子串。代码如下：

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<h3>函数标签测试</h3>
[取子字符串]<br>
<c:set var="tel" value="010-88886666"/>
<b>fn:substring:</b>${fn:substring(tel, 4, 7)}<br>
<b>fn:substring:</b>${fn:substring(tel, 4, -1)}<br>
<b>fn:substring:</b>${fn:substring("helloworld!", 5, -1)}<br>
<br>
<b>fn:substringAfter:</b>${fn:substringAfter(tel, "-")}<br>
<b>fn:substringAfter:</b>${fn:substringAfter("hello world!", "hello")}<br>
<b>fn:substringAfter:</b>${fn:substringAfter("hello world!", "")}<br>
<br>
<b>fn:substringBefore:</b>${fn:substringBefore(tel, "-")}<br>
<b>fn:substringBefore:</b>${fn:substringBefore(tel, "")}<br>
<br>
[判断是否存在子串]<br>
```

```

<c:set var="str1" value="public static void main()."/>
<b>fn:contains:</b>${fn:contains(str1,"void")}<br>
<b>fn:contains:</b>${fn:contains(str1,"")}<br>
<b>fn:contains:</b>${fn:contains(str1,"Void")}<br>
<b>fn:containsIgnoreCase:</b>${fn:containsIgnoreCase(str1,"Void")}<br>

```

解释:

- <%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>, 用于导入函数标签库, 声明标签前缀为 “fn”
- fn:substring(tel, 4, 7), 是在字符串 tel 中, 从下标为 4 的字符开始取, 直到下标为 7 的字符的前一个字符。
- fn:substringAfter(tel, "-"), 是在字符串 tel 中, 取字符 “-” 后边的字符串。
- fn:contains(str1,"void"), 判断字符串 str1 中是否包含子字符串 “void”。

运行效果如下:

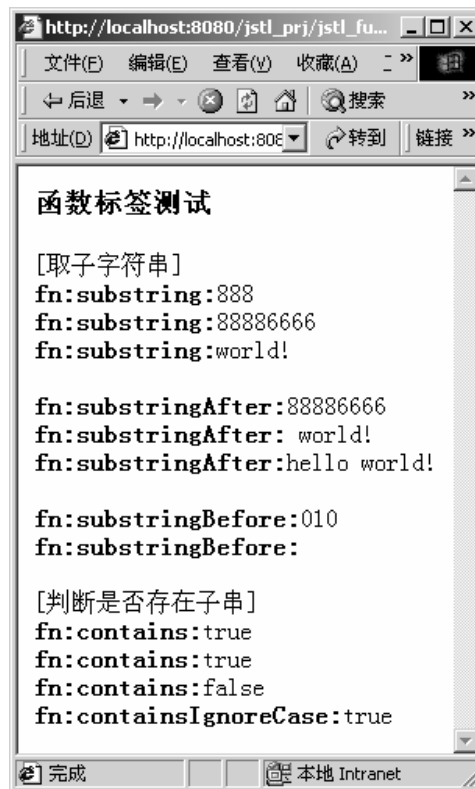


图 10-1 使用函数标签

10.1.2 判断字符串开头和结尾

此部分演示如何通过函数标签判断一个字符串是否以某子字符串开头/结尾, 代码如下:

```

<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<h3>函数标签测试</h3>
[判断字符串开头和结尾]<br>

```

```

<c:set var="tel" value="010-88886666"/>
<c:set var="str1" value="public static void main()."/>
<b>fn:startsWith:</b>${fn:startsWith(str1,"public")}<br>
<b>fn:startsWith:</b>${fn:startsWith(str1,"private")}<br>
<b>fn:startsWith:</b>${fn:startsWith(str1,"")}<br>
<br>
<b>fn:endsWith:</b>${fn:endsWith(str1,".")}<br>
<b>fn:endsWith:</b>${fn:endsWith(str1,";")}<br>
<br>
[indexof]<br>
<b>fn:indexOf:</b>${fn:indexOf(str1,"void")}<br>

```

解释:

- `fn:startsWith(str1,"public")`, 判断字符串 `str1` 是否以 “public” 开头。
- `fn:endsWith(str1,".")`, 判断字符串 `str1` 是否以点 “.” 结尾。
- `fn:indexOf(str1,"void")`, 用于计算子串 “void” 在 `str1` 中的起始位置。

运行效果如下:

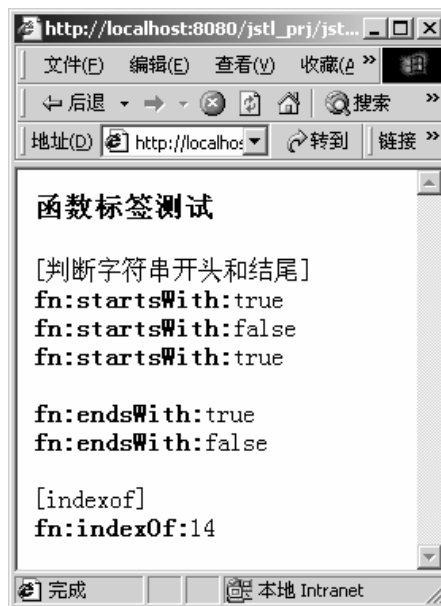


图 10-2 使用函数标签

10.1.3 拆分和连接字符串

本部分演示如何按指定分隔符将一个字符串拆分成一个数组、如何将字符串数组中各元素用指定字符进行连接以生成一个字符串。代码如下:

```

<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<h3>函数标签测试</h3>
[join 和 split]<br>

```

```

<c:set var="str1" value="public static void main()"/>
<c:set var="array1" value='${fn:split(str1, " ")}' />
str1:${str1}<br>
array1:${fn:join(array1, "-")}<br>
<br><br>
<c:set var="str2" value="<b>Test escapeXml</b>"/>
escapeXml:${fn:escapeXml(str2)}<br>
str2:${str2}<br>

```

解释:

- `<c:set var="array1" value='${fn:split(str1, " ")}' />`, 用空格符将 `str1` 拆分成字符串数组 `array1`。
- `fn:join(array1, "-")`, 用减号符将数组 `array1` 中的各元素连接成一个字符串。
- `fn:escapeXml(str2)`, 用于将 `str2` 中的内容原样显示到浏览器中, 即 `` 不作为 `html` 指令, 而代码最后一行中的 `${str2}`, 没有过滤字符串格式, 所以 `` 是作为 `html` 指令起作用的。

运行效果如下:



图 10-3 使用函数标签

10.2 SQL 标签

SQL 标签库用于访问各种关系数据库, 它提供的各种标签可用于在 JSP 页面直接访问数据库。但在大中型应用中通常都是基于 MVC 模型设计的, 不提倡直接在 JSP 页中访问数据库。SQL 标签是为基于 WEB 的小型应用程序而设计的。SQL 标签库提供的功能有: 传递各种数据库查询、访问查询结果、数据库修改、执行各种数据库事务。

SQL 标签库中提供的标签如下表所示:

名称	含义
<code><sql:setDataSource></code>	为数据库设置数据源
<code><sql:query></code>	执行查询并返回结果集
<code><sql:update></code>	执行 insert,update,delete 语句
<code><sql:transaction></code>	建立事务处理

<sql:param>	为 SQL 语句中的参数设置值
-------------	-----------------

表 10.2 SQL 标签

1) <sql:setDataSource>

用于为数据库设置数据源。他是一个空标签，允许用户为数据库设置数据源信息。其语法为：

```
<sql:setDataSource      DataSource="datasource"      |      url="jdbcurl"
driver="driverclassdriver"      user="username"      password="userpwd"
var="varname"  scope="page/request/session/application"  />
```

其中：

DataSource 可以是 JAVA 命名和目录接口 (JNDI) 资源的路径或 JDBC 参数字符串。
url 是与数据库关联的 URL（使用了 DataSource 就不能使用 url）
driver 是一个 JDBC 参数，其值为驱动程序的类名
user 是数据库的用户名
password 是用户的密码
var 是指定数据源的导出范围变量的名称
scope 指定范围

2) <sql:query>

搜索数据库并返回结果集。此标签可以是空标签或容器标签。其语法为：

```
<sql:query  var="varname"  datasource="  datasource"  scope="{page | request |
session | application}"  maxRows="  maxRows"  startRow="  startRow"  />
sql  statement
<sql:param/>
</sql:query>
```

其中：

sql 指定 sql 查询语句
var 为查询结果指定导出的范围变量的名称
scope 指定变量的范围
datasource 指定与要查询的数据库关联的数据源
maxRows 指定结果中所包含的数据的最大行数
startRow 指定从指定索引开始的数据行

3) <sql:update>

执行 insert,update,delete 语句。如果所有数据行都没有受到插入，更新或删除操作的影响，则会返回 0，标签的语法为：

```
<sql:update      sql="sqlupdate"      datasource="  datasource"      var="varname"
scope="{page | request | session | application}"  />
sql  statement
```

```
<sql:param value="value">
</sql:update>
```

其中：

update 是 SQL 的 UPDATE 语句

param 为查询的参数

4) <sql:transaction>

用于为<sql:query>标签和<sql:update>标签建立事务处理上下文。其语法是：

```
<sql: transaction  datasource=" datasource"  isolation=isolationLevel >
<sql:update>or <sql:query>statements
</sql: transaction>
```

其中：

datasource 设置 SQL 的数据源，他可以是字符串或一个 datasource 对象

isolation 设置事务处理的隔离级别。隔离级别可以是 read_committed, read_uncommitted, repeatable_read 或 serializable

5) <sql:param>

为 SQL 语句中的参数设置值。他充当<sql:query>和<sql:update>的子标签。其语法是：

```
<sql:param value="value">
```

其中：

value 设置参数的值

下面通过实例演示各 SQL 标签的使用。

10.2.1 执行查询

该示例演示如何使用 SQL 标签查询数据库表中的内容，并显示出来，代码如下：

```
<%@ page contentType="text/html;charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="sql" uri="http://java.sun.com/jstl/sql_rt" %>

<h3>SQL 标签测试</h3>
<sql:setDataSource driver="sun.jdbc.odbc.JdbcOdbcDriver"
url="jdbc:odbc:dbdsn"
user="sa" password="" var="conn1"/>

<sql:query var="users" dataSource="${conn1}">
select * from users
</sql:query>
<table border=1>
<tr>
<td>用户名</td>
```

```

<td>密码</td>
</tr>
<c:forEach var="row" items="${users.rows}">
<tr>
<td><c:out value="${row.username}"/></td>
<td><c:out value="${row.userpass}"/></td>
</tr>
</c:forEach>
</table>

```

解释:

- `<sql:query var="users" dataSource="${conn1}">`, 用于执行查询, 并将查询结果保存到 `users` 变量中。
- `<c:forEach var="row" items="${users.rows}">`, 用于循环显示 `users` 中的所有用户数据。

10.2.2 执行更新

本示例演示了如何通过 `SQL` 标签执行更新语句、建立事务、及向 `SQL` 语句传递参数, 代码如下:

```

<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@taglib prefix="sql" uri="http://java.sun.com/jstl/sql_rt" %>

<h3>SQL 标签测试</h3>
<sql:setDataSource driver="sun.jdbc.odbc.JdbcOdbcDriver"
url="jdbc:odbc:dbdsn"
user="sa" password="" var="conn1"/>

<sql:update var="newuser" dataSource="${conn1}">
insert into users (username,userpass) values ('张三','1111')
</sql:update>

<c:set var="name1" value="李四"/>
<c:set var="pass" value="1111"/>
<sql:transaction dataSource="${conn1}">
    <sql:update>
        insert into users (username,userpass) values (?,?)
        <sql:param value="${name1}"/>
        <sql:param value="${pass}"/>
    </sql:update>
</sql:transaction>

```


新用户添加完成,

`查看所有用户`

解释:

- `<sql:param value="${name1}"/>`用于向语句 “insert into users (username,userpass) values (?,?)” 的参数传递值。
- 本代码运行后, 会在数据库表 `users` 中添加以下两行数据:
张三, '1111'
李四, '1111'

10.2.3 分页显示

本示例演示了如何通过 SQL 标签实现 JSP 的分页显示功能, 代码如下:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@taglib prefix="sql" uri="http://java.sun.com/jstl/sql_rt" %>
<h3>SQL 标签测试</h3>
<sql:setDataSource driver="sun.jdbc.odbc.JdbcOdbcDriver"
url="jdbc:odbc:dbdsn"
user="sa" password="" var="conn1"/>

<c:set var="size" value="2" />
<c:if test="${users == null}">
    <sql:query var="users"
        scope="session"
        sql="SELECT * FROM users ORDER BY id"
        dataSource="${conn1}"/>
</c:if>

<c:choose>
    <c:when test="${users.rowCount == 0}">
        没有用户...
    </c:when>
    <c:otherwise>
        <b>以下是用户列表: </b>
        <p>
            <table border="1">
                <th>用户名</th>
                <th>密码</th>

                <c:forEach items="${users.rows}"
```

```
        var="row"
        begin="{param.start}" end="{param.start + size - 1}">
    <tr>
        <td><c:out value="{row.username}" /></td>
        <td><c:out value="{row.userpass}" /></td>
    </tr>
</c:forEach>

</table>
</c:otherwise>
</c:choose>
<p>
<c:choose>
    <c:when test="{param.start > 0}">
        <a href="jstl_sql_3.jsp?start=<c:out value="{param.start - size}"
/>">    上一页</a>
    </c:when>
</c:choose>
<c:choose>
    <c:when test="{param.start + size < users.rowCount}">
        <a href="jstl_sql_3.jsp?start=<c:out value="{param.start + size}"
/>">    下一页</a>
    </c:when>
</c:choose>
```

解释:

- <c:set var="size" value="2" />, 声明每页显示 2 条记录。
- <c:when test="{users.rowCount == 0}">, 用于判断结果集 users 中的记录数据是否为 0。

10.3 国际化支持标签

国际化(I18N)与格式化标签库可用于创建国际化的 WEB 应用程序，他们对数字和日期-时间的输出进行了标准化。使用此类标签可以使 WEB 应用快速本地化。国际化 (I18N) 与格式化标签库中的标签如下表所示:

名称	功能
<fmt:setLocale>	用于设置客户端的区域设置
<fmt:bundle>	指定消息资源使用的文件
<fmt:setBundle>	设置消息资源文件
<fmt:message>	用于输出给定资源包的值

表 10.3 国际化及格式化标签库

1) <fmt:setLocale>

用于设置客户端指定的区域设置。他将区域设置存储在 `javax.servlet.jsp.jstl.fmt` 配置变量中。

SetLocale 是一个空标签。其语法为：

```
<fmt:setLocale          value="setting"          variant="variant"
scope="page/request/session/application" />
```

其中：

- `value` 包含一个含有两个小写字母的语言代码和一个含有两个大写字母的国家代码。语言和国家代码应该用连字符或下划线分隔。
- `Variant` 指定特定于浏览器的变量（可选）
- `Scope` 指定配置变量的范围

2) <fmt:bundle>

指定消息资源使用的文件，创建一个 I18N 本地化上下文，并将他的资源包加载到其中。语法如下：

```
<fmt:bundle  basename="basename">
body  content
</fmt:bundle>
```

其中：

- `basename` 指定资源包的名称

3) <fmt:setBundle>

设置消息资源文件，创建一个 I18N 本地化上下文，并将他存储在范围变量中。该标签是一个空标签。其语法为：

```
<fmt:setBundle  basename=" basename"  var="varname"  scope="page/request/session/application"
/>
```

其中：

- `basename`: 指定资源包的名称
- `var`: 指定导出的范围变量的名称，他存储 I18N 本地化上下文
- `scope` 指定 `var` 的范围

4) <fmt:message>

用于输出给定资源包的值。语法为：

```
<fmt:message  key="messagekey">
```

其中：

- `key` 指定消息的关键字

10.3.1 国际化格式化标签示例

本示例演示如何使用不同国家的格式显示指定日期，如何使页面自动显示为本地区的语言形式。代码如下：

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

```

<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<fmt:setBundle basename="messages" scope="session"/>

<title>
<fmt:message key="title"/>
</title>

<h2><fmt:message key="hello"/></h2>
<hr>
<h3>国际化标签测试</h3>
<%
    Date now = new Date();
    request.setAttribute("date",now);
%>
美国格式:<fmt:setLocale value="us"/>
<fmt:formatDate value="{date}"/><br>
中国格式:<fmt:setLocale value="zh_CN"/>
<fmt:formatDate value="{date}"/></br>

```

解释:

- <fmt:setLocale value="us"/>, 将地区设置为美国, 日期按美国格式显示。
- <fmt:setLocale value="zh_CN"/>, 将地区设置为中国, 日期按中国格式显示。
- <fmt:setBundle basename="messages" scope="session"/>, 会自动装载 WEB-INF\classes 目录下的 *.properties 文件, basename="messages" 用于指定资源文件的主文件名。
- 本例中 classes 目录下有以下文件: messages_en.properties、messages_zh.properties, 当地区信息为英美国家时, 自动装载 messages_en.properties, 地区为中国时, 自动装载 messages_zh.properties 文件。
- <fmt:message key="title"/>, 用于在装载过的资源文件中查找键名为 “title” 的消息, 然后显示。

messages_en.properties 文件内容如下:

```

title=computer
hello=Hello

```

messages_zh.properties 文件内容如下:

```

title=\u8ba1\u7b97\u673a
hello=\u4f60\u597d

```

该文件中的内容是经过编码的中文, 这样做是为了保证平台无关性。要将中文编码成 unicode 编码格式, 可以使用 JDK\bin 中自带的命令: native2ascii, 如: 要将 a.txt 的内容编码后存储到 b.txt 中, 可用如下指令: native2ascii a.txt b.txt。

代码运行效果如下（由于浏览器是在中文环境下运行，所以装的是 messages_zh.properties 文件中的资源，如果环境变成英美地区，则 WEB 容器会自动装载 messages_en.properties 文件中的消息，这样页面会自动显示消息文件中的英文提示信息）：



图 10-4 使用国际化及格式化标签

10.3.2 XML 处理标签

XML 标签库提供一组标记来管理 XML 数据，通过这些标签可以解析一个 XML 文档，然后将其值读取出来。通常解析 xml 文件的工作应该在 javabeen 使用 java 的 XML 解析 API 来完成，在此通过示例了解使用 JSTL 中的 XML 标签解析 XML 的方法：

示例 1——解析 XML 片段，代码如下：

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ page import = "java.util.Date" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<h3>xml 标签测试</h3>
<c:set var="xmlcontent">
    <user>
        <username>张三</username>
        <userpass>1111</userpass>
    </user>
</c:set>

<x:parse var="xml1" xml="{xmlcontent}" />
<x:out select="$xml1/user/username" />
<x:out select="$xml1/user/userpass" />
<br>
```

```
<x:out select="$xml1/user/userpass" escapeXml="true"/>
```

解释:

- `<c:set var="xmlcontent">`, 用于将 XML 片段装入变量 `xmlcontent` 中。
- `<x:parse var="xml1" xml="{xmlcontent}" />`, 用于解析 XML 文档片段, 分解后的结果存储在 `xml1` 变量中。
- `<x:out select="$xml1/user/userpass" />`, 用于根据 Xpath 路径显示相应值。

运行效果如下:



图 10-5 使用 XML 标签

示例 2——使用 XML 标签中的循环、判断标签:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ page import = "java.util.Date" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml" %>
<h3>xml 标签测试</h3>
<c:set var="xmlcontent">
<users>
    <user id="001">
        <username>张三</username>
        <userpass>1111</userpass>
    </user>
    <user id="002">
        <username>李四</username>
        <userpass>2222</userpass>
    </user>
</users>
</c:set>
<x:parse var="doc" xml="{xmlcontent}" />
<x:forEach select="$doc/users">
```

```

    <x:out select="." /><br>
</x:forEach>

<hr>
<x:forEach select="$doc//user">
    <x:if select="./username">
        <x:out select="./username" /><br>
    </x:if>
</x:forEach>
<hr>
<x:forEach select="$doc/users">
    <x:choose>
        <x:when select='$doc//user[@id="001"]'>
            <x:out select="$doc//username" /><br>
        </x:when>
        <x:otherwise>
            没有 001<br>
        </x:otherwise>
    </x:choose>
</x:forEach>

```

以上语法非常类似于 xsl 对 xml 的操作，<x:forEach select="\$doc/users">用于对指定 xpath 下的元素进行遍历，<x:if select="./username">用于判断当前节点是否为指定结点。该代码运行效果如下：



图 10-6 使用 XML 标签

示例 3——装载一个 XML 文档进行解析：

.....

```
<c:import var="xmlcontent" url="a.xml"/>
<x:parse var="doc" xml="{xmlcontent}" />

<x:forEach select="$doc/users">
    <x:out select="." /><br>
</x:forEach>
.....
```

解释:

- `<c:import var="xmlcontent" url="a.xml"/>`, 用于装载当前目录下的 `a.xml` 然后存储在 `xmlcontent` 变量中。

xml 文档内容如下:

```
<?xml version="1.0" encoding="utf-8"?>
<users>
    <user id="001">
        <username>tom1</username>
        <userpass>1111</userpass>
    </user>
    <user id="002">
        <username>tom2</username>
        <userpass>2222</userpass>
    </user>
</users>
```

运行后, 解析效果如下图所示:



图 10-7 使用 XML 标签

小结

JSTL 的函数标签提供了常用的函数功能。

SQL 标签库用于访问各种关系数据库, 它提供的各种标签可用于在 JSP 页面直接访问数据库。

大中型应用中通常都是基于 MVC 模型设计的,不提倡直接在 JSP 页中访问数据库,SQL 标签是为基于 WEB 的小型应用程序而设计的。

国际化(I18N)与格式化标签库可用于创建国际化的 WEB 应用程序,他们对数字和日期-时间的输出进行了标准化。

XML 标签库提供一组标记来管理 XML 数据,通过这些标签可以解析一个 XML 文档,然后将其值读取出来。

课后练习

- 1、 SQL 标签库中的哪个动作用于从 JSP 页面将数据插入数据库？
- 2、 通过 JSTL 使项目中的文本实现国际化的步骤，假设该项目要求自动在中英两种语言间切换？

第十一章 自定义标签(一)

概要

JSP 自定义标记 是用户定义的标记，当 **servlet** 容器处理自定义标记时，会调用一个或者多个指定的 **Java** 类文件处理它。**JSP** 自定义标记为在 **JSP** 页中将表示与业务逻辑分离提供了一种标准化的机制，使页面设计者可以将注意力放到表示上，而应用程序开发人员编写标记处理程序类以处理标记并处理所有需要的 **Java** 代码和数据操作。本章主要介绍自定义标签的开发过程及其使用方法。

目标

- 理解自定义标签概念
- 掌握自定义标签开发过程

目录

- 11.1 自定义标签简介
- 11.2 自结束标签
- 11.3 标签中的属性
- 11.4 TLD 文件

11.1 自定义标签简介

11.1.1 自定义标签概念

顾名思义,JSP 标签就是在 JSP 文件中使用的标记。它类似于 html 语法中的标记,像 `body`、`table`。通过在 JSP 文件中引用它(就像使用 html 标记那样),可以更方便的实现对于 Java 代码模块的重用。JSP 标签分为标准 JSP 环境自带的标签(即前面章节中学习过的 JSP 动作标签)和 JSP 自定义标签。

JSP 自定义标签是用户定义的标记,它遵循 XML 语法。当 servlet 容器处理自定义标记时,会自动调用一个 Java 类文件完成相对应的功能。

Java 开发人员编写标记处理程序类以处理标记并处理所有需要的 Java 代码和数据操作。对于 Web 页面设计者来说,自定义标记与标准 HTML 标记使用起来没什么区别,但 HTML 标记只能完成前台显示的功能,而自定义标记可以在后台完成某些操作。

正确编写自定义标记可以让 Web 设计者创建、查询和操作数据而无需编写一行 Java 代码。正确使用自定义标记使 Java 开发人员不必再在编码过程中考虑表示层。这样应用程序开发小组的每一位成员都可以关注于他或者她最擅长的事物。

所以说,JSP 自定义标记为在动态 Web 页中将表示与业务逻辑分离提供了一种标准化的机制,使页面设计者可以将注意力放到表示上,而应用程序开发人员编写后端的代码。

11.1.2 标签相关概念

JSP 自定义标签的使用语法与普通 HTML 标签相同,与自定义标签相关的基本术语简单说明如下,这些术语在开发 JSP 自定义标签时要用到:

1) 自结束标签——没有标记体的标签

示例: `<test: myhrtag />`

说明: 假设 `myhrtag` 是一个自定义标签

2) 属性

示例: `<test: myhrtag color="red" />`

说明: 以上标签中包含了 `color` 属性,值为 `red`

3) 带标记体的标签

示例: `<test: myhrtag > xxxxx </test: myhrtag>`

说明: 以上标签中间的 `xxxxx` 即为标记体

4) 子标记

示例: `<test: myhrtag >`
`<test:mytag2/>`
`</test: myhrtag>`

说明: 以上 `myhrtag` 标签中间的 `mytag2` 即为子标记

11.1.3 如何创建自定义标签

先从一个基础概念入手,一个 java 类想成为 Servlet 则该类必须实现某些规范,在 JAVA 中实现某些规范通常是指继承某些特定类或实现某些特定的接口。自定义标签功能的实现要求在后台必须有一个相关的 JAVA 类的支持,但并不是任意编写一个 JAVA 类就能处理 JSP 标签,这个类也必须实现指定的规范才能用于支持 JSP 标签,这些规范表现形式也是接口和类,它们在 `javax.servlet.jsp.tagext`

包中声明，主要接口/类的描述如下：

- `javax.servlet.jsp.tagext.Tag` 接口，所有处理 JSP 标签的类必须实现该接口。该接口中声明了 6 个方法，如果直接从该接口生成类则必须实现所有的 6 个方法，通常不会直接通过该接口生成标签的处理类。
- `javax.servlet.jsp.tagext.TagSupport` 类，该类实现了 `Tag` 接口，用于创建不带标记体的自结束标签，这些标签中可以带属性。
- `javax.servlet.jsp.tagext.BodyTagSupport` 类，该类继承了 `TagSupport`，用于创建带标记体的标签。

通常我们自定义的标签，编写处理程序时使用 `TagSupport` 和 `BodyTagSupport` 即可，以下是开发和用一个 JSP 自定义标签的全过程：

- 1) 开发标记处理类，编译生成 class 文件，该类要继承 `TagSupport` 或 `BodyTagSupport`；
- 2) 创建标记库描述符文件*.tld，在该文件中为标记处理类指定标签名、声明标签属性；
- 3) 在 JSP 中引用标签库；
- 4) 在 JSP 中使用标 JSP 标签

11.2 自结束标签

11.2.1 自结束标签简介

这是一种不带标记体的标签，所以该类标签的处理类直接继承 `javax.servlet.jsp.tagext.TagSupport` 即可。`TagSupport` 的主要方法如下：

- `public int doStartTag() throws JspException`
在 WEB 容器遇到标签开始时，该方法会运行。
- `public int doEndTag() throws JspException`
在 WEB 容器遇到标签结束时，该方法会运行。

`TagSupport` 类中有一个重要的成员：`pageContext`，该成员的功能与 JSP 的内置对象 `pageContext` 完全相同。通过该对象可以得到其他几个 JSP 对象的引用。这样，我们就可以在 JAVA 类中与 JSP 进行交互了。如：`JspWriter out=pageContext.getOut();`这一语句可以得到 JSP 内置对象 `out` 的引用，通过 `out` 我们就可以向客户端浏览器中输出内容了。要使用其他几个 JSP 对象原理与此相同。

11.2.2 自结束标签开发示例

以下通过一个实例演示自结束标签的开发步骤，在该示例中开发了一个名为<test: MyHr>的标签，在 JSP 中使用后可以在浏览器中显示 5 条水平线，其中输出 5 条水平线的功能是在 JAVA 类中完成的，步骤如下：

第一步：编写标记处理类

```
//MyHrTag.java
package demo;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
import javax.servlet.http.*;
```

```
import java.text.*;
import java.util.*;

public class MyHrTag extends TagSupport {
    public int doStartTag() throws JspException {
        try {
            //得到网络输出流,pageContext 是从父类继承过来的成员;
            JspWriter out=pageContext.getOut();
            //向网页输出内容;
            out.println("<h4>开始执行 doStartTag().....</h4>");
            //输出 5 条水平线;
            for(int i=1; i<=5; i++)
                out.println("<hr>");
        }
        catch(Exception e) {
        }
        //return EVAL_BODY_INCLUDE; //处理标记体
        return Tag.SKIP_BODY; //跳过标记体;
    }
    public int doEndTag() throws JspException {
        try {
            JspWriter out=pageContext.getOut();
            out.println("<h3>开始执行 doEndTag().....</h3>.");
        }
        catch(IOException ioe) {
            throw new JspException(ioe.getMessage());
        }
        //return Tag.SKIP_PAGE; //返回这个值,则终止页面执行;
        return EVAL_PAGE;
    }
}
```

解释:

- 在自定义的类中,重写了父类 `TagSupport` 的两个方法: `doStartTag()`、`doEndTag()`,在容器遇到标记开始时运行 `doStartTag()`,遇到标记结束时运行 `doEndTag()` 方法;
- `doStartTag()` 方法的返回值:通常可以取两个值:
`EVAL_BODY_INCLUDE`——包含标记体,本例中要编写自结束标记所以不使用该值;
`SKIP_BODY`——跳过标记体,即不处理标记体,开发自结束标记应该使用该值。
- `doEndTag()` 方法的返回值:通常可以取两个值:

SKIP_PAGE——返回这个值，则终止页面执行；

EVAL_PAGE——返回该值则处理完当前标记后，JSP 页面中止运行。

➤ 本例中，doStartTag()中输出 5 条水平线，doEndTag()中输出一些提示信息。

步骤二：创建标记库描述符文件 myhr.tld，该文件要存放在 WEB-INF 目录下：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>myhr</shortname>
    <tag>
        <name>MyHr</name>
        <tagclass>demo.MyHrTag</tagclass>
        <bodycontent>EMPTY</bodycontent>
    </tag>
</taglib>
```

解释：

- 在 tld 文件中，映射了标记名和处理程序类；
- <tallib>元素，代表开始一个标记库的描述
- <tligversion>元素，代表标记库的版本
- <jspversion>元素，代表标记所支持的 JSP 的版本
- <shortname>为标记库起别名，相当于注释，无实际用途
- <tag>元素，代表开始描述一个标记，其下子元素如下：

<name>——为标记处理类起的标记名

<tagclass>——指定标记处理类的全名(即带包的名字)

<bodycontent>——标记体的类型，该示例中不需要标记体，所有设置为 EMPTY，该值的其他取值在后续内容中讲解

步骤三：在 JSP 中引用标记库描述文件

引用标记库有两种方式，分别称为静态引用和动态引用。

方式一：动态引用（即直接在 JSP 页面中使用 TLD 文件）：

```
<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/myhr.tld" prefix="test" %>
```

解释：

➤ JSP 指令<%@ taglib... %>用于引用标记库，该指令的两个属性作用如下：

uri——指明要引用的标记库，在静态引用中就是 TLD 文件的路径

prefix——为标记起的前缀名，可以防止多个标记重名的情况出现

方式二：静态引用

首先在 web.xml 中为 TLD 文件声明别名：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<Web-app>
    .....
    <taglib>
        <taglib-uri>myhr</taglib-uri>
        <taglib-location>/WEB-INF/myhr.tld</taglib-location>
    </taglib>
    .....
</Web-app>
```

然后在 JSP 中通过别名引用 TLD 文件：

```
<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="myhr" prefix="test" %>
```

使用静态引用方式时，当 TLD 文件名变更时，不必修改使用该标记的 JSP 文件，只要在 web.xml 中进行配置即可，但使用较麻烦。在调试程序时，通常使用第一种方式，即动态引用方式。

步骤四：在 JSP 中使用标记

```
<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/myhr.tld" prefix="test" %>

<test:MyHr/>
```

到此为止，自结束标签开发完毕，其中主要工作有两个：开发标记处理类、配置 TLD 文件。访问 JSP，运行结果如下：

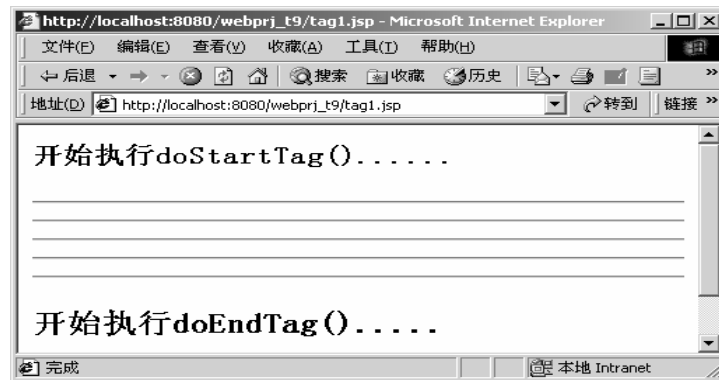


图 11-1 使用自定义标签

11.3 标签中的属性

11.3.1 为自定义标签添加属性

以上的示例中开发了一个简单的 JSP 标签，但一个实用的标签通常还要由属性来制定标签的特定行为，以下示例演示为自定义标签添加属性。该示例在 1.2 示例基础上，为标签添加了 color 和 loop 两个自定义属性，以控制水平线的颜色和输出的水平线的数量。

步骤一：编写标记处理类

```
package demo;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
import javax.servlet.http.*;
import java.text.*;
import java.util.*;

public class MyHrTag2 extends TagSupport {
    //声明属性;
    private String color = "black";
    private String loop = "1";

    public int doStartTag() throws JspException {
        try {
            //得到网络输出流;
            JspWriter out = pageContext.getOut();
            //向网页输出内容;
            out.println("<h4>开始执行 doStartTag().....</h4>");
            //输出 5 条水平线;
            int n=Integer.parseInt(loop);
```

```

        for(int i = 1; i <= n; i++)
            out.println("<hr color=" + this.color + ">");
    }
    catch(Exception e) {
    }
    //return EVAL_BODY_INCLUDE; //处理标记体
    return Tag.SKIP_BODY; //跳过标记体;
}

//声明属性方法;
public void setColor(String color){
    this.color = color;
}
public void setLoop(String loop){
    this.loop = loop;
}
}

```

解释:

该类中添加了两个属性，属性的声明与 javabeen 语法完全相同，即属性本身是 `private` 类型，但要求提供 `public` 的 `get`、`set` 方法。

步骤二：创建 TLD 文件，本例中是在 1.2 中的 `myhr.tld` 文件中进行修改得到：

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>myhr</shortname>
    <tag>
        <name>MyHr</name>
        <tagclass>demo.MyHrTag</tagclass>
        <bodycontent>EMPTY</bodycontent>
    </tag>
    <tag>
        <name>MyHr2</name>
        <tagclass>demo.MyHrTag2</tagclass>
    </tag>

```

```

        <bodycontent>EMPTY</bodycontent>
        <attribute>
            <name>color</name>
            <required>>false</required>
        </attribute>
        <attribute>
            <name>loop</name>
            <required>>false</required>
        </attribute>
    </tag>
</taglib>

```

解释:

➤ <tag>中的子元素<attribute>用于为标签声明属性，其子元素如下:

<name>——用于指定属性名称

<required>——用于声明该属性是否为必需的，本例中声明 color、loop 两个属性都不是必需的。

在 JSP 中引用 TLD，并使用标签

```

<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/myhr.tld" prefix="test" %>

```

第一次测试(未赋属性值):


```
<test:MyHr2/>
```

第二次测试(使用两个属性):


```
<test:MyHr2 color="red" loop="3"/>
```

第三次测试(使用一个属性):


```
<test:MyHr2 color="green"/>
```

至此，开发完毕，运行 JSP，结果如下:

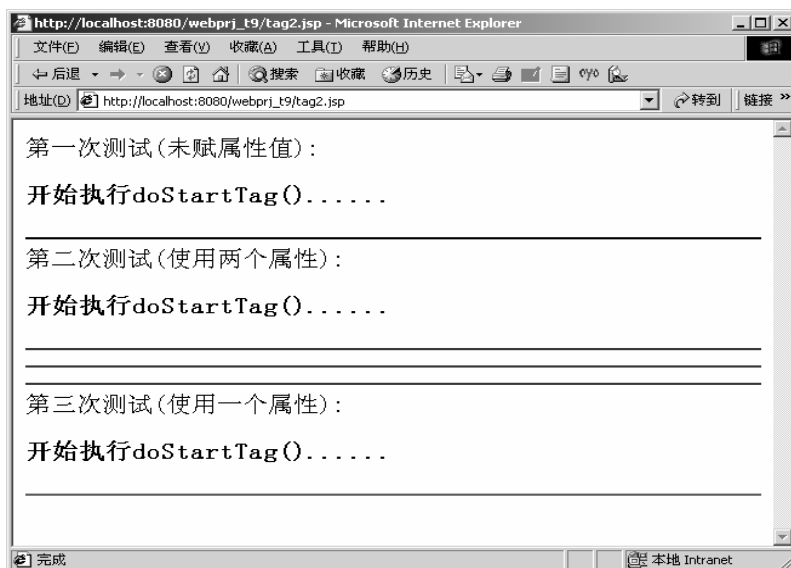


图 11-2 使用带属性的 JSP 自定义标签

开发带属性的自定义标签时，主要工作有两个：在标记处理类中声明属性和 `get/set` 方法、在 TLD 文件中声明属性及其必需性。

11.3.2 带属性的标签综合示例

通过以上两个简单示例，演示了创建一个自定义标签的步骤。在实际开发过程中，许多场合都可以使用自定义标签。如：判断用户是否登录、显示购物车信息等。

接下来，本实例中演示了自定义标签在实际开发中的一个应用，该示例中创建了一个 JSP 标签，用于在 JSP 中判断用户是否登录过，如果没登录过则自动转到登录页，这样就可以避免在 JSP 中使用代码段进行业务判断了。

login.html，显示登录表单

```
.....  
<FORM action="loginservlet" method="post">  
用户名:<INPUT name=username>  
用户密码:<INPUT name=password type=password>  
<INPUT name=Submit2 type=submit value=登录>  
<INPUT name=Reset type=reset value=清除></P>  
</FORM>  
.....
```

LoginServlet.java，判断用户名和密码是否为空，如果非空则认为登录成功：

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

```

public class LoginServlet extends HttpServlet {
    /**
     * The doGet method of the servlet. <br>
     */
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }

    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=gb2312");
        PrintWriter out = response.getWriter();
        String username = request.getParameter("username");
        String userpass = request.getParameter("userpass");
        if (!username.equals("") && !userpass.equals("")) {
            // 登录成功则在 session 中存储登录成功的标记, 然后转到管理员页;
            HttpSession session = request.getSession(true);
            session.setAttribute("login", "yes");
            response.sendRedirect("admin.jsp");
        }
        else {
            response.sendRedirect("login.html");
        }
    }
}

```

LoginTag.java, 为标记处理程序, 从 session 中取出登录标志进行判断用户是否登录过:

```

package demo;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
import javax.servlet.http.*;
import java.text.*;

```

```
import java.util.*;

public class LoginTag extends TagSupport {
    public int doStartTag() throws JspException {
        try {
            HttpSession session = pageContext.getSession();
            Object obj = session.getAttribute("login");
            //判断是否从未登录过，如果没登录过则转到登录页；
            if(obj == null) {
                ((HttpServletResponse)pageContext.getResponse())
                    .sendRedirect("login.html");
                return SKIP_BODY;
            }
            String login=(String)obj;
            //判断是否与登录成功后存储的值相同；
            if(!login.equals("yes")){
                ((HttpServletResponse)pageContext.getResponse())
                    .sendRedirect("login.html");
                return SKIP_BODY;
            }
        }
        catch(Exception e) {
        }
        return Tag.SKIP_BODY;
    }
}
```

标记描述符文件 myhr.tld，为标记处理类起标记名：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
.....
    <tag>
        <name>islogin</name>
        <tagclass>demo.LoginTag</tagclass>
```

```

        <bodycontent>EMPTY</bodycontent>

    </tag>

.....

</taglib>

```

admin.jsp，在该页面开始利用自定义标记判断用户是否登录过：

```

<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/myhr.tld" prefix="test" %>

<test:islogin/>

<h3>欢迎来到管理员页面</h3>

```

运行该示例，如果用户没有登录过而直接去访问 admin.jsp 则自定义标记会将页面转到 login.html，显示登录页，这样可以有效限制未授权用户绕过登录直接访问 admin.jsp。通过在 JSP 中使用自定义标签，减少了 JSP 页中的代码段和业务处理代码，实现了显示与逻辑的分离。

11.4 TLD 文件

以上示例中创建了标记描述符文件*.TLD，现总结该文件的用法如下：

- 该文件必须放在 WEB-INF 目录下；
- 该文件是 XML 格式的文件，各元素及功能说明如下：

元素	说明
<taglib>	代表开始一个标记库的描述
<tlibversion>	代表标记库的版本，是自定义的
<jspversion>	代表标记所支持的 JSP 的版本
<shortname>	为标记库起别名，相当于注释，无实际用途
<tag>	代表开始描述一个标记

表 11-1 标记描述符 TLD 文件元素说明

其中<tag>元素中又包含若干子元素，说明如下：

元素	说明
<name>	为标记处理类起的标记名
<tagclass>	指定标记处理类的全名(即带包的名字)
<bodycontent>	标记体的内容类型，如果为 EMPTY 代表无标记体
<attribute>	用于为标签声明属性

表 11-2 <tag>元素的子元素

<attribute>元素为标签声明属性时，需要两个子元素：

元素	说明
<name>	用于指定属性名称
<required>	用于声明该属性是否为必需的

表 11-3 <attribute>元素的子元素

小结

JSP 自定义标记为在动态 Web 页中将表示与业务逻辑分离提供了一种标准化的机制。

编写标记处理程序时，通常要继承 TagSupport 类或 BodyTagSupport 类。

自结束标签可以带属性。

标记库描述文件*.tld 用以为标记处理类指定标签名、声明标签属性。

在 JSP 页面中引用 TLD 时有两种方式：静态引用、动态引用。

课后练习

- 1、 简述开发一个自结束自定义标签的步骤？
- 2、 简述在 JSP 页面中引用 TLD 文件的两种方式？

第十二章 自定义标签(二)

概要

上一章介绍了简单的 JSP 自定义标签的开发方法，在简单标签中只有属性，没有标记体，也不能嵌套其他的标记。本章介绍如何创建带标记体的标签和可嵌套标签。

目标

- 开发带标记体的标记（掌握）
- 嵌套标记（理解）

目录

- 12.1 标签中的标记体
- 12.2 标签中的子标记
- 12.3 标签处理类中的各种返回值

12.1 标签中的标记体

12.1.1 标记体简介

标签的标记体是 JSP 页中出现在自定义标签的开始和结束标签之间的数据，标记体也称正文。操纵其正文的标签称为带标记体的标签（也称为正文标签）。

可以编写标签处理程序对标签的标记体进行操作。要编写标记体标签处理程序，必须实现 `BodyTag` 接口。`BodyTag` 继承了 `Tag` 的所有方法，而且还实现了另外两个处理正文内容的方法，见下表：

方法	说明
<code>setBodyContent(BodyContent b)</code>	<code>bodyContent</code> 属性的 Setter 方法
<code>doInitBody()</code>	对正文内容进行初始化操作

表 12-1 `BodyTag` 接口的方法

为方便开发，在 JSP 类库中为 `BodyTag` 接口提供了实现类：`javax.servlet.jsp.tagext.BodyTagSupport`。该类继承了 `TagSupport` 并实现了 `BodyTag` 接口。因此，标记体标签处理程序只需要覆盖要使用的方法。`BodyTagSupport` 类中定义了一个 `protected bodyContent` 成员变量及 `get/setBodyContent()` 方法，`bodyContent` 是一个缓冲区，用以保存标记体正文内容。

在一个标签处理类中，`BodyTag` 的处理流程如下：

- 当容器创建一个新的标签实例后，通过 `setPageContext` 来设置标签的页面上下文。
- 使用 `setParent` 方法设置这个标签的上一级标签，如果没有上一级嵌套，设置为 `null`。
- 设置标签的属性，如果没有定义属性，就不调用此类方法。
- 调用 `doStartTag` 方法，这个方法可以返回以下三者之一：`EVAL_BODY_INCLUDE`、`EVAL_BODY_BUFFERED`、`SKIP_BODY`，当返回 `EVAL_BODY_INCLUDE` 时，就将标记体直接写到输出流中，如果返回 `SKIP_BODY`，就不再计算标签的正文，如果返回 `EVAL_BODY_BUFFERED`，就将标记体的正文包含到 `bodyContent` 成员中。
- 调用 `setBodyContent` 设置当前的 `BodyContent`。
- 调用 `doInitBody`，可以在该方法中对 `BodyContent` 进行一些初始化操作。
- 每次计算完 `Body` 后调用 `doAfterBody`，如果返回 `EVAL_BODY_AGAIN`，表示继续处理一次标记体，直到返回 `SKIP_BODY` 才继续往下执行。
- 调用 `doEndTag` 方法，结束标签处理。

12.1.2 一个简单的带标记体的标签

了解了标签处理的流程，下面通过实例了解这类标签的开发过程，本示例中创建了一个标签用于在浏览器中输出其标记体内容，且输出的次数由标签的属性决定：

步骤一：编写标记处理类

```
package demo;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
```

```

public class BodyTag extends BodyTagSupport {
    private int loop;
    public void setLoop(int loop){
        this.loop = loop;
    }
    public int doStartTag() throws JspTagException {
        if (loop > 0) {
            return EVAL_BODY_INCLUDE; //自动将标记体包含到输出流中;
        }
        else{
            return SKIP_BODY; //跳过标记体，不进行处理;
        }
    }
    public int doAfterBody() throws JspTagException {
        if(loop > 1) {
            loop--;
            return EVAL_BODY_AGAIN;
        }
        else {
            return SKIP_BODY;
        }
    }
}

```

解释:

- doStartTag()中的返回值 EVAL_BODY_INCLUDE, 可以直接将标签的正文内容输出到浏览器中。
- doAfterBody()在处理完一次正文后会自动执行, 该方法如果返回 EVAL_BODY_AGAIN, 则代表再处理一遍正文 (即输出到浏览器), 返回 SKIP_BODY 代表正文处理到此结束。本例中循环向浏览器中输出标记体的正文, 直到属性 loop 的值小于 1。

步骤二: 创建标记库描述符文件 myhr.tld, 该文件要存放在 WEB-INF 目录下:

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>

```

```
<tlibversion>1.0</tlibversion>
<jspversion>1.1</jspversion>
<shortname>test</shortname>
<tag>
  <name>bodytag</name>
  <tagclass>demo.BodyTag</tagclass>
  <bodycontent>tagdependent</bodycontent>
  <attribute>
    <name>loop</name>
    <required>true</required>
  </attribute>
</tag>
</taglib>
```

步骤三：在 JSP 中使用该标记

```
<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/test.tld" prefix="test" %>

测试带标记体的自定义标记:<br>
<test:bodytag loop="3">
  这是标记体<br>
</test:bodytag>
```

访问以上 JSP，结果如下：

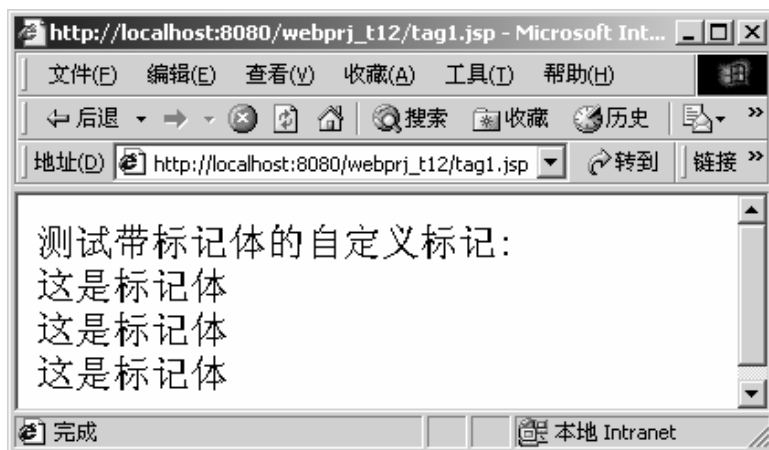


图 12-1 使用带标记体的自定义标签

12.1.3 一个简单的带标记体的标签

上面示例中通过在 `doStartTag()` 方法中返回 `EVAL_BODY_INCLUDE`，简单地将标记体的内容直接输出到了浏览器中，并未对内容进行任何处理，在一些业务中有时需要对标记体正文进行处理，以

下示例演示了如何读取标记体内容并进行处理，本示例读取了标记体内容，然后将其变为粗斜体输出到浏览器中：

1) JSP 文件内容如下：

```
<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/test.tld" prefix="test" %>
测试带标记体的自定义标记:<br>
<%
    session.setAttribute("username","tom");
%>
<test:equal name="username" value="tom">
    session 中的用户名是:tom<br>
</test:equal>
```

解释：

- 该 JSP 中使用了自定义标签，标签属性 name 指定 session 中所存储数据的键名，value 属性用以指定一个常量，在标签处理类中会比较两者的值，如果相等则输出标记体内容到浏览器。

2) 标签处理类如下：

```
package demo;
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class EqualTag extends BodyTagSupport {
    private String name;
    private String value;

    public void setName(String name){
        this.name = name;
    }
    public void setValue(String value){
        this.value = value;
    }
    public int doStartTag() throws JspTagException {
        System.out.println("doStartTag().....");
        return EVAL_BODY_BUFFERED;
    }
    public void setBodyContent(BodyContent bodyContent) {
        System.out.println("setBodyContent().....");
        this.bodyContent = bodyContent;
    }
}
```

```

    }
    //初始化标记体
    public void doInitBody() throws JspTagException {
        System.out.println("doInitBody().....");
    }
    public int doAfterBody() throws JspTagException {
        System.out.println("doAfterBody().....");
        return SKIP_BODY; //停止包含
    }
    public int doEndTag() throws JspException {
        System.out.println("doEndTag().....");
        String username =
        (String)pageContext.getSession().getAttribute(name);
        if (username.equals(value)) {
            String str = bodyContent.getString();
            str = "<b><i>" + str + "</i></b>";
            try{
                //将结果写回到输出流中
                JspWriter out=pageContext.getOut();
                out.println( str );
                this.bodyContent=null;
            }catch (IOException e){
                e.printStackTrace();
            }
        }
        return EVAL_PAGE;
    }
}

```

解释:

- doStratTag()中的返回值 EVAL_BODY_BUFFERED 代表——不直接将标记体内容写到输出流中，而是缓存到成员变量 bodyContent 中（该成员从 BodyTagSupport 继承过来）。
- doEndTage()中，bodyContent.getString()语句用于将缓存的标记体内容读取出来转成 String。

3) 标记库描述符文件

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

```



```

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>test</shortname>
  <tag>
    <name>equal</name>
    <tagclass>demo.EqualTag</tagclass>
    <bodycontent>tagdependent</bodycontent>
    <attribute>
      <name>name</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>value</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>

```

4) 运行 JSP, 结果如下:



图 12-2 处理标记体内容

12.2 标签中的子标记

一个标签中可以再包含其他的子标记, 这种标签称为嵌套标记。创建嵌套标签时, 标记处理类与普通标签相似, 但在 `doStartTag()` 方法中必须返回 `EVAL_BODY_INCLUDE`, JSP 容器才会处理嵌套的子标记。本部分通过示例, 演示如何创建嵌套的 JSP 自定义标签。

12.2.1 嵌套标签示例

1) 先看一下 JSP 文件中是如何使用该标记的:

```

<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/test.tld" prefix="test" %>

```

测试嵌套的自定义标记:


```
<%
    //向 session 中存储测试用的数据
    session.setAttribute("username","tom2");
%>
<test:nest name="username" value="tom2">
    <test:bodytag loop="3">
        session 中的用户名是:tom2<br>
    </test:bodytag>
</test:nest>
```

解释:

- 子标记<test:bodytag>是在第一个例子中创建的标记,用于将标记体内容输出 loop 遍。
- 外层标记<test:nest>是本例中要创建的标记,该标记根据属性值判断 session 中键名为 username 的值是否为 tom2,如果是则运行子标记。

2) 标记处理类如下:

```
package demo;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class NestTag extends BodyTagSupport {
    private String name;
    private String value;

    public void setName(String name){
        this.name = name;
    }
    public void setValue(String value){
        this.value = value;
    }
    public int doStartTag() throws JspTagException {
        String username=(String)pageContext.getSession().getAttribute(name);
        if (username.equals(value)) {
            return EVAL_BODY_INCLUDE;//自动将标记体包含到输出流中;
        }
        else{
```

```

        return SKIP_BODY; //跳过标记体，不进行处理；
    }
}
}

```

3) 标记库描述符文件:

```

.....
<tag>
    <name>nest</name>
    <tagclass>demo.NestTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
        <name>name</name>
        <required>true</required>
    </attribute>
    <attribute>
        <name>value</name>
        <required>true</required>
    </attribute>
</tag>
.....

```

解释:

- `<bodycontent>JSP</bodycontent>`, 用以声明标记体内容和其他标记, 也可以是 JSP 标准标记。

4) 访问 JSP, 运行结果如下:

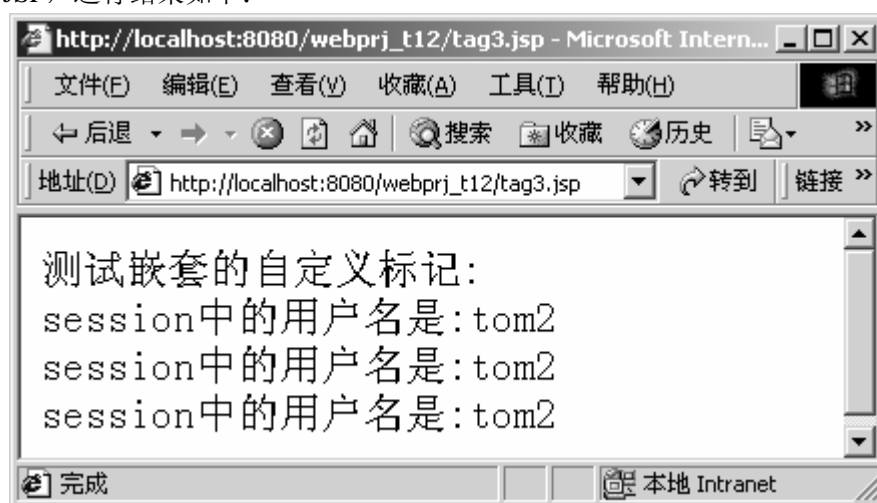


图 12-3 使用嵌套标签

12.3 标签处理类中的各种返回值

在 `doStartTag()`、`doEndTag()` 等方法中有各种各样的返回值，现各返回值的取值和功能总结如下：

返回值	调用者	说明
<code>EVAL_BODY_INCLUDE</code>	<code>doStartTag()</code>	将正文的内容传递给服务器，把这些内容送入输出流
<code>EVAL_BODY_BUFFERED</code>	<code>doStartTag()</code>	不直接处理标记体，而是将标记体内容保存到缓冲区 <code>bodyContent</code> 成员中
<code>SKIP_BODY</code>	<code>doStartTag()</code> <code>doAfterBody()</code>	告诉服务器不要处理正文内容
<code>EVAL_PAGE</code>	<code>doEndTag()</code>	继续执行页面剩余的内容
<code>SKIP_PAGE</code>	<code>doEndTag()</code>	不要处理剩余的页面，中止页面的运行
<code>EVAL_BODY_AGAIN</code>	<code>doAfterBody()</code>	再次处理标记体内容

表 12-1 返回值说明

小结

JSP 自定义标签中可以包含标记体，要处理标记体，标记处理类需要继承 `BodyTagSupport` 类。

JSP 自定义标签中可以嵌套子标记，`doStartTag()` 方法中必须返回 `EVAL_BODY_INCLUDE`，JSP 容器才会处理嵌套的子标记。

课后练习

- 1、 要将标记体内容输出到浏览器有几种方法，分别如何实现？
- 2、 如何开发嵌套 JSP 标记？

第十三章 MVC 实现

概要

MVC 是一种流行的软件设计模式，它可以将一个 WEB 应用分为 3 个层次：模型层、视图层、控制器层，各模块各负其责且相互联系，协作完成一个完整的业务，使用 MVC 设计模式可以使 WEB 应用逻辑更清晰，有利于开发大型的 WEB 应用，本章介绍如何在 WEB 应用中使用 MVC 模式。

目标

- JSP 设计模式（理解）
- MVC 模式（掌握）

目录

- 13.1 Web 开发与 MVC 模式
- 13.2 MVC 模式示例

13.1 Web 开发与 MVC 模式

13.1.1 设计模式简介

所谓设计模式，简单言之就是解决某种问题通常要遵循的一种方法和解决方案，这种解决方案是经过大量实践检验证明是行之有效的。任何行业的工作都有一套成型的模式，比如建筑行业，要建一座住宅楼，需要对该楼的结构、施工方案进行规划设计。住宅楼的设计师不必从头开始自己探索，因为建筑行业已有几千年的发展史，建筑物如何设计抗震性更强、采光面更广、与周边环境配合更自然，这些方面都有前人总结出来许多成熟的经验，作为一个新的设计师在具备建筑学、环境学的基础知识后，再学习并利用这些成功的模式才能设计一个完整的建设方案。如果他自己从头开始自己全新的设计，会遇到许多难题，自己经过努力解决后会发现他发现的解决方案绝大多数已经包含在前人总结的经验中。

软件开发行业也是如此，在开发一个完整的项目时，如何设计软件的整体架构是非常重要的，设计不合理会导致开发人员间配合不协调、开发速度缓慢等许多问题。软件行业经过多年发展，也形成了许多成熟的模式。我们本章关心的是开发 JavaWeb 应用方面的模式，也称为 JSP 设计模式。

13.1.2 JSP 设计模式

基于 java 的 Web 应用的核心技术是 jsp、servlet、javaBean、自定义标签，在一个 JAVA WEB 应用中，我们到底应该将什么样的代码写入 JSP、什么样的代码写入 Servlet 中？JSP 设计模式告诉 JAVA WEB 开发人员，应该如何设计整个 JSP 网站，应该将哪些代码写入哪些组件中。JSP 设计模式主要包含两个：

1、 JSP Model1——JSP+JavaBean

该模式的体系结构如图 13-1 所示：

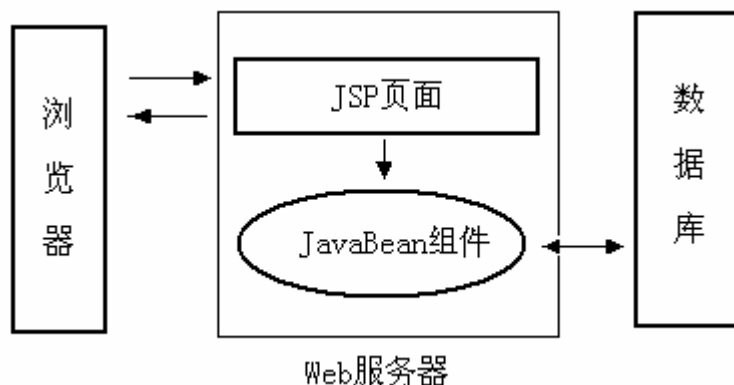


图 13-1 JSP Model1

在 JSP 设计模式 1 中，没有用到 Servlet 组件，JSP 负责接收和处理用户传递过来的数据，JSP 借助 JavaBean 组件来完成在封装数据或与数据库打交道的操作，这种模式下 JSP 通常既要负责显示信息，还要负责处理业务逻辑。

为便于理解，以“用户注册”为例，使用模式 1 的处理流程如下：用户在浏览器中填写 HTML 表单——>提交表单——>JSP 接收用户名、密码——>JSP 验证信息完整性——>JSP 通过 javaBean 验证数据库中是否已存在该用户——>JSP 根据 javaBean 的返回结果显示注册成功/失败的信息。

2、 JSP Model2——JSP+Servlet+JavaBean

该模式的体系结构如图 13-2 所示：

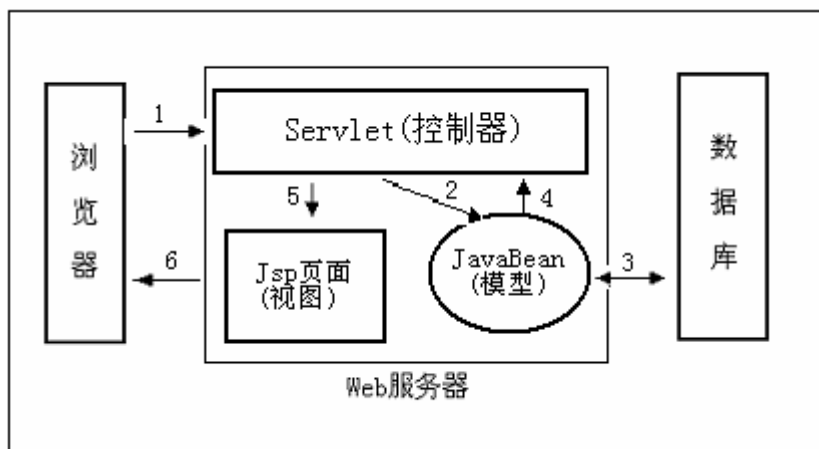


图 13-2 JSP Model2

在 JSP 设计模式 2 中，用到了三种 WEB 组件：jsp、servlet、javabean，servlet 负责接收请求数据并处理整体业务流程，在处理业务过程中需要封装数据或与数据库打交道则调用 javabean，然后根据 javabean 的结果或 servlet 自己的处理结果选择合适的 JSP 页面传递给客户端浏览器。

这样，就将 Model-1 中 JSP 页中的业务逻辑处理代码分离到了 Servlet 中（即 Servlet 负责判断信息完整性、用户名是否已存在这些情况），整体逻辑更清晰，利于开发人员协作分工，也利于维护。这种模式简称 MVC，MVC 将 WEB 应用分成三个核心模块：模型、视图、控制器，三个模块各负其责、相互协作：

➤ 视图

视图是用户直接看到并与之交互的界面。视图用于接收用户的输入数据和向用户显示数据（显示的数据是由控制器存储到某些作用域中转发到视图的），但视图中不进行任何业务处理。

➤ 模型

模型用于封装业务数据和业务逻辑，模型的主要作用是数据库打交道以处理业务逻辑、向视图层传递数据。

➤ 控制器

控制器用以接收用户请求，并调用模型处理业务，然后根据模型的处理结果选择相应的视图显示到客户端。

将应用分为三个模块后，非常利于应用的维护，如果视图要变更，模型和控制器不受影响，同理要改变其他部分，也不会导致整个网站的整体修改。

13.1.3 两种 JSP 设计模式对比

以上介绍了两种 JSP 设计模式，利用这两种设计模式都可以开发 JavaWeb 应用，它们分别用于不同的场合：

如果要开发一个较简单的规模较小的 WEB 应用，可以使用 JSP Model-1 来实现，这样在项目中引入的技术较少，开发难度小，比如：在 2006 年世界杯开赛赛前，某足彩网站要开发一个简单的投票系统来统计球队投票信息，该投票系统就使用一个月的时间，在世界杯赛结束后就不需要了，这种系统很显然可以使用简单的 jsp+javabean 来完成。

如果要开发的 WEB 应用规模很大，需要很多开发人员配合完成，则应该选择 MVC 模式来设计整个应用，这样将整个应用分为不同的组件，利于分工协作，也利于维护。

13.2 MVC 模式示例

以上介绍了两种 JSP 设计模式，本部分通过示例演示如何在 WEB 应用中使用 JSP Model-2，即 MVC 模式，然后会简单演示相同的功能使用 Jsp Model-1 如何实现。

13.2.1 示例分析

本部分示例以“用户注册”为例，通过这个简单的业务可以清晰的理解 MVC 模式，本例中用户通过表单提交的注册信息由 Servlet 接收并处理，处理过程中借助 javabean 进行与数据库有关的业务处理，以下是该“用户注册”的体系图：

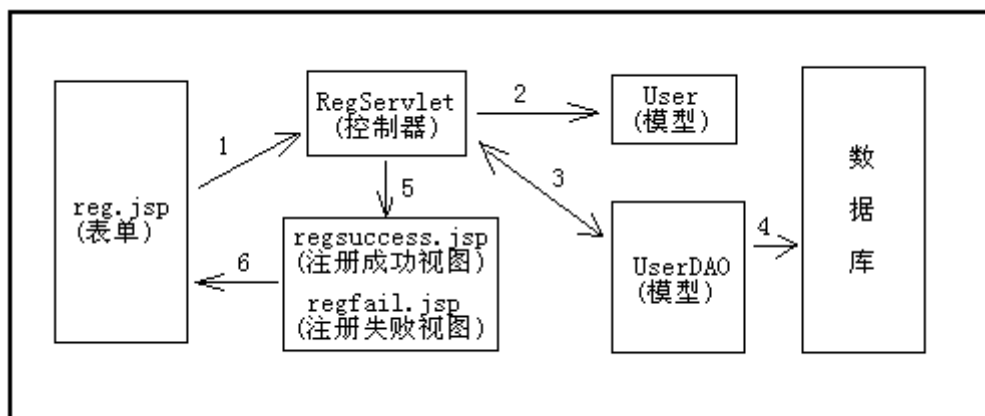


图 13-3 “用户注册”结构图

解释：

- 1) 客户通过浏览器访问得到 reg.jsp，填写表单然后提交。
- 2) 用户提交的注册信息由 RegServlet 接收，在 RegServlet 内部先验证用户名、密码是否为空、密码长度是否足够 8 位，如果验证失败则转到 regfail.jsp，同时传递错误信息给视图 regfail.jsp。如果表单验证通过，RegServlet 将注册信息封装到 User 对象中（User 是一个 javabean，专门用于封装用户信息）。
- 3) RegServlet 实例化 UserDAO 对象，该对象删除包含与用户有关的业务逻辑方法（如：判断用户是否已存在、将用户插入到数据库），RegServlet 将上一步中生成的 User 对象传递给 UserDAO，由 DAO 对象负责处理与数据库有关的业务。
- 4) UserDAO 对象与数据库打交道，然后将处理结果返回给 RegServlet。
- 5) RegServlet 根据 UserDAO 对象的返回值，判断注册是否成功，然后选择相应的视图传递给客户端浏览器。

该示例中 MVC 各模块组件如下：

MVC	组件
M（模型）	User.java ——封装用户信息 UserDAO.java——封装数据库访问逻辑
V（视图）	reg.jsp——注册表单 regsuccess.jsp——注册成功的信息 regfail.jsp——注册失败的信息 ErrorsTag.java —— 自定义 JSP 标签，用于显示注册出错时的信息 SuccessTag.java —— 自定义 JSP 标签，用于显示注册成功时的信息
C（控制器）	RegServlet.java —— 处理注册流程

表 13-1 “用户注册”模块分解

另外，示例中将访问数据库的代码封装到了一个专门的类（DBConnection.java）中，该类根据配置文件 db.properties 连接数据库，并提供了访问数据的常用方法（executeQuery()、executeUpdate()），然后 UserDAO 类继承了 DBConnection 类，如图所示：

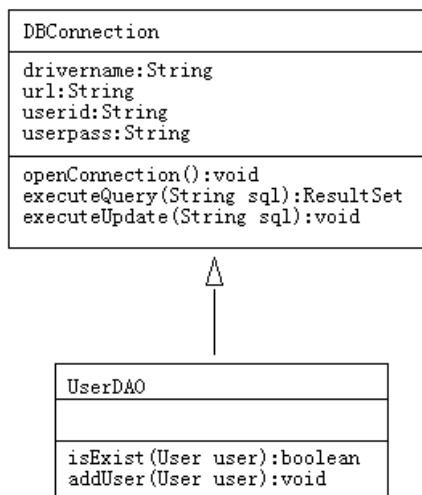


图 13-4 模型 UserDAO

13.2.2 开发 MVC 模式的注册系统

步骤一：创建注册表单 reg.jsp

```

<%@ page contentType="text/html;charset=gb2312" %>
<html>
<head>
<title>用户注册</title>
</head>
<body>
<form action="regservlet" method="post">
    用户名称:<input type="text" size="10" name="username"><br>
    用户密码:<input type="password" size="10" name="userpass"><br>
    <input type="submit" value=" 确定 ">
  
```

```
</form>
</body>
</html>
```

步骤二：创建数据库配置文件 db.properties

```
drivername=sun.jdbc.odbc.JdbcOdbcDriver
url=jdbc:odbc:dbdsn
userid=
userpass=
```

步骤三：创建封装数据库底层操作的类 DBConnection.java

```
package demo;
import java.sql.*;
import java.util.Properties;
import java.io.*;

//封装数据库底层操作(连接、执行查询、执行更新);
public class DBConnection {
    private String drivername;
    private String url;
    private String userid;
    private String userpass;
    private static Connection conn = null;
    private static Statement stmt = null;

    public DBConnection(){
        init();
    }
    //读取配置文件，将连接数据库的配置信息读到数据成员；
    private void init(){
        try{
            Properties p = new Properties();
            //将文件内容装载到集合；
            p.load( this.getClass().getResourceAsStream("db.properties") );
            //从 p 中读取信息；
            drivername = p.getProperty("drivername", "");
            url = p.getProperty("url", "");
        }
    }
}
```

```
        userid = p.getProperty("userid", "");
        userpass = p.getProperty("userpass", "");
    }catch(Exception e){
        e.printStackTrace();
    }
}
//连接数据库;
public void openConnection(){
    try{
        Class.forName( drivername );
        conn = DriverManager.getConnection(url,userid,userpass);
        stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);

    }catch(Exception e){
        e.printStackTrace();
    }
}
//执行查询;
public ResultSet executeQuery(String sql) {
    ResultSet rs = null;
    try{
        if (conn == null)
            this.openConnection();
        rs = stmt.executeQuery(sql);
    }catch(Exception e){
        e.printStackTrace();
    }
    return rs;
}
//执行更新;
public void executeUpdate(String sql){
    try{
        if (conn == null) this.openConnection();
        stmt.executeUpdate(sql);
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

```
//关闭资源;
public void close(){
    try{
        stmt.close();
    }catch(Exception e){
    }
    try{
        conn.close();
    }catch(Exception e){
    }
}
}
```

步骤四：创建模型中的实体类 User.java，用于封装用户信息，这是一个简单的 javabean

```
package demo;
import java.io.Serializable;
public class User implements Serializable{
    private String username;
    private String userpass;

    public void setUsername(String username){
        this.username = username;
    }
    public void setUserpass(String userpass){
        this.userpass = userpass;
    }
    public String getUsername(){
        return this.username;
    }
    public String getUserpass(){
        return this.userpass;
    }
}
```

步骤五：创建模型中与数据库打交道的类 UserDao.java，其中包含业务方法

```
package demo;
import java.util.*;
```

```

import java.sql.*;

/**
 * 本类用于封装与 User 有关的业务逻辑方法;
 */
public class UserDao extends DBConnection{
    /**
     * 判断用户是否已占用
     */
    public boolean isExist(User user){
        boolean isexist=false;
        String username=user.getUsername();
        String sql="select * from users where username='" + username + "'";
        try{
            ResultSet rs = this.executeQuery(sql);
            if(rs.next()) isexist=true;
            rs.close();
        }catch(Exception e){
            e.printStackTrace();
        }
        return isexist;
    }
    /**
     * 将用户信息插入数据库
     */
    public void addUser(User user){
        String username=user.getUsername();
        String userpass=user.getUserpass();
        String sql="insert into users (username,userpass) values ('"
            + username + "','" + userpass + "')";
        this.executeUpdate(sql);
    }
}

```

步骤六：创建 RegServlet.java，处理整体注册流程

```

package demo;

import java.io.IOException;

```

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

public class RegServlet extends HttpServlet {
    /**
     * 处理用户注册的流程
     * 1. 先进行表单验证, 有错误则定位到错误页显示错误消息.
     * 2. 判断用户名是否被占用, 被占用则转到错误页显示错误消息.
     * 3. 进行注册.
     * 4. 转到注册成功的页面.
     */
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        // 读取表单信息
        String username = request.getParameter("username");
        String userpass = request.getParameter("userpass");

        // 得到请求转发器
        RequestDispatcher dispatcher1 = this.getServletConfig()
            .getServletContext().getRequestDispatcher("/regfail.jsp");
        // 判断信息是否为空, 为空则转到错误页
        if( username.equals("") || userpass.equals("") ){
            request.setAttribute("msg", "用户名或密码为空!");
            dispatcher1.forward(request, response);
            return;
        }
        // 如果密码长度小于 8, 转到错误页
        if( userpass.length() < 8 ){
            request.setAttribute("msg", "密码小于 8 位!");
            dispatcher1.forward(request, response);
            return;
        }
        // 将信息封装到实体
```



```

        User user = new User();
        user.setUsername(username);
        user.setUserpass(userpass);

        //通过 JAVABEAN 判断, 该用户名是否被占用.
        UserDao dao = new UserDao();
        if( dao.isExist(user) ){//如果要注册的用户名已被占用
            request.setAttribute("msg", "用户名已存在, 请重新起名字!");
            dispatcher1.forward(request, response);
            return;
        }
        //要注册的名字可用, 进行注册操作
        dao.addUser(user);
        //将用户存储到请求, 然后转发到页面显示
        request.setAttribute("user", user);
        RequestDispatcher dispatcher2 = this.getServletConfig()
            .getServletContext().getRequestDispatcher("/regsuccess.jsp");
        dispatcher2.forward(request, response);
    }
}

```

步骤七: 创建自定义 JSP 标签类——ErrorsTag.java, 用以在 JSP 中显示注册错误的消息

```

package demo;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
import javax.servlet.http.*;
import javax.servlet.*;

public class ErrorsTag extends TagSupport{
    public int doStartTag() throws JspException {
        try {
            //得到网络输出流;
            JspWriter out = pageContext.getOut();

            //读取请求中的错误消息
            ServletRequest request = pageContext.getRequest();
            String msg = (String)request.getAttribute("msg");

```

```
        //向网页输出错误消息;
        out.println("错误:" + msg );
    }
    catch(Exception e) {
    }
    return Tag.SKIP_BODY; //跳过标记体;
}
}
```

步骤八：创建自定义 JSP 标签类——SuccessTag.java，用以在 JSP 中显示注册成功的消息

```
package demo;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
import javax.servlet.http.*;
import javax.servlet.*;

public class SuccessTag extends TagSupport{
    public int doStartTag() throws JspException {
        try {
            //得到网络输出流;
            JspWriter out = pageContext.getOut();
            //读取请求中的 user
            ServletRequest request = pageContext.getRequest();
            User user = (User)request.getAttribute("user");
            //向网页输出息;
            out.println("用户名:" + user.getUsername() + "<br>" );
            out.println("密码:" + user.getUserpass() + "<br>" );
        }
        catch(Exception e) {
        }
        return Tag.SKIP_BODY; //跳过标记体;
    }
}
```

步骤九：创建自定义标签的标签库描述文件——test.tld

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
```

```

<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>test</shortname>
    <tag>
        <name>errors</name>
        <tagclass>demo.ErrorsTag</tagclass>
        <bodycontent>EMPTY</bodycontent>
    </tag>
    <tag>
        <name>success</name>
        <tagclass>demo.SuccessTag</tagclass>
        <bodycontent>EMPTY</bodycontent>
    </tag>
</taglib>

```

步骤十：创建注册失败的信息显示页面——regfail.jsp，本页中使用了一张图片 fail.jpg

```

<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/test.tld" prefix="html" %>
<img src=fail.jpg>
<html:errors/>

```

步骤十一：创建注册成功的信息显示页面——regsuccess.jsp，本页中使用了一张图片 success.jpg

```

<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/test.tld" prefix="html" %>

    注册成功，您的信息如下：<br>
<html:success/>

```

现在可以部署该 WEB 应用，在运行之前还要配置数据库：

1、 创建数据库 test，在库中创建表 users，并输入测试数据

	列名	数据类型	长度	允许空
	id	int	4	
	username	nvarchar	50	✓
	userpass	nvarchar	50	✓

图 13-5 users 表

	id	username	userpass
	1	admin	admin
	2	b	b
	3	c	c

图 13-6 users 表的测试数据

2、 创建 ODBC 数据源 dbdsn，指向以上 test 数据库。

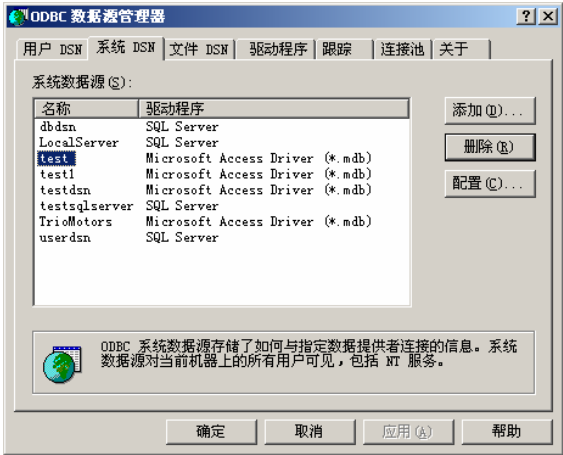


图 13-7 创建 ODBC 数据源

3、 运行效果如下：

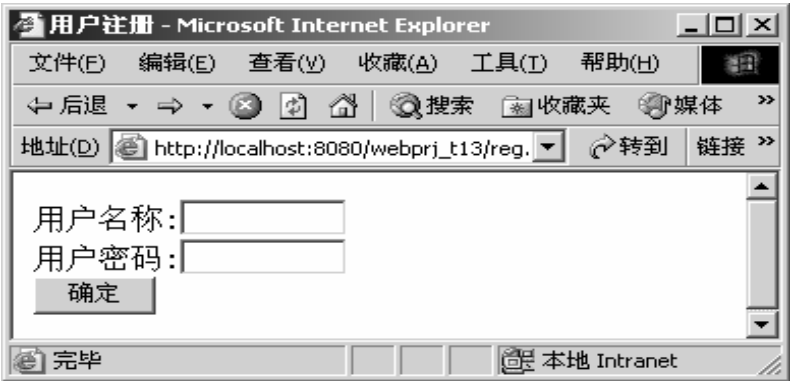


图 13-8 用户注册表单视图

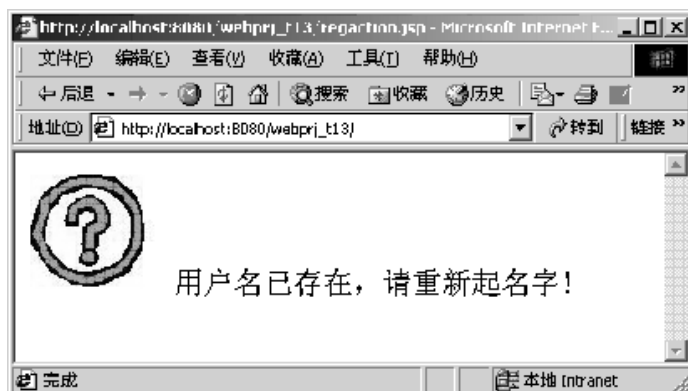


图 13-9 注册失败的视图

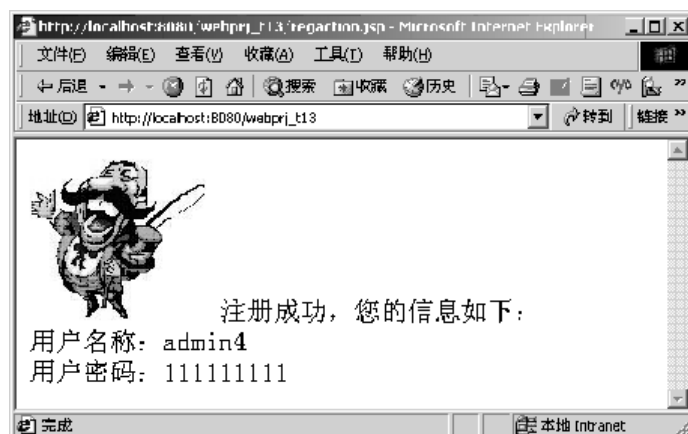


图 13-10 注册成功的视图

解释:

- 1) 在本例中为演示完整的 MVC 模式, 使用了 JSP 自定义标签技术, 在标签中封装了读取 request 中的错误消息的语句。这样可以避免在 JSP 中使用代码段进行读取操作。
- 2) 在实际开发中可以借助一些现成的框架技术 (如: Struts), 这些框架中会提供许多现成的标签用于完成常见的视图功能 (如: 显示消息、循环显示集合数据、显示 javabeans 的数据等)。

13.2.3 简单了解 JSP Model-1

以上示例中使用 JSP Model-2 实现了“用户注册”系统, 本部分简单了解一下如何使用 JSP Model-1 来实现相同的功能, 在模式 1 中没有了 Servlet, 为此可以添加一个负责处理业务流程的 JSP 页——regaction.jsp, 替代原来的 RegServlet 完成注册流程控制, 用户注册表单提交后提交至 regaction.jsp 即可:

regaction.jsp 代码如下

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ page import="demo.*" %>
<%
    //读取表单信息
    String username=request.getParameter("username");
    String userpass=request.getParameter("userpass");
    //判断信息是否为空, 为空则转到错误页
```

```

    if( username.equals("") || userpass.equals("") ){
        out.println("<img src=fail.jpg>");
        out.println("用户名或密码为空!");
        return;
    }
    //如果密码长度小于 8，转到错误页
    if( userpass.length()<8 ){
        out.println("<img src=fail.jpg>");
        out.println("密码小于 8 位!");
        return;
    }
    //将信息封装到实体
    User user=new User();
    user.setUsername(username);
    user.setUserpass(userpass);

    //通过 JAVABEAN 判断，该用户名是否被占用。
    UserDao dao=new UserDao();
    if( dao.isExist(user) ){//如果要注册的用户名已被占用
        out.println("<img src=fail.jpg>");
        out.println("用户名已存在，请重新起名字!");
        return;
    }
    //要注册的名字可用，进行注册操作
    dao.addUser(user);
%>

注册成功，您的信息如下: <br>
用户名称: <%=user.getUsername()%><br>
用户密码: <%=user.getUserpass()%><br>

```

解释:

- 1、在 `regaction.jsp` 中即有业务流程控制的代码，又有显示信息的 **HTML** 语句，这样很不利于美工对页面进行美工设计，而且 **JAVA** 程序员在这样的页面中编码也很不方便。
- 2、如果业务流程有变化，要对该 **JSP** 页进行修改，因为业务和显示代码在一起，所以这有可能会无意中修改到视图部分，而且美工与程序员不能同时工作。不利于分工协作和程序的维护。
- 3、但是在较简单的系统中还是可以使用 **JSP Model-1** 的。

小结

设计模式是解决某种问题通常要遵循的一种方法和解决方案，这种解决方案是经过大量实践检验证明是行之有效的。

JSP 设计模式告诉 JAVA WEB 开发人员，应该如何设计整个 JSP 网站，应该将哪些代码写入哪些组件中。JSP 设计模式主要包含两个：jsp+javaBean、MVC。

MVC 将 WEB 应用分成三个核心模块：模型、视图、控制器，三个模块各负其责、相互协作,共同完成 WEB 功能。

使用 MVC 模式，利于开发组的协作分工及程序维护，开发大型 WEB 应用时应该使用 MVC 模式。

课后练习

- 1、 简述两种 **JSP** 设计模式的区别？
- 2、 简述 **MVC** 设计模式各组成部分的功能？
- 3、 在 **MVC** 模式中，视图与模型之间是如何交换数据的？

第十四章 Ajax 框架

概要

AJAX (Asynchronous JavaScript and XML) 是多种技术的综合, 包括 Javascript、XHTML 和 CSS、DOM、XML 和 XSTL、XMLHttpRequest。AJAX 与 Web 服务器异步地交换和处理数据, 大大减少了网络交互过程中的等待时间。手工编写 AJAX 代码的过程非常繁琐, Ajax 框架可以大大减化 AJAX 的开发过程。本章主要介绍 Ajax 框架的概念及 DWR 框架的使用方法。

目标

- AJAX 框架的概念 (理解)
- 使用 AJAX 框架 (掌握)

目录

- 14.1 AJAX 应用示例
- 14.2 AJAX 框架和 DWR
- 14.3 DWR 框架程序示例

14.1 AJAX 应用示例

在以前阶段的学习中,我们已经接触过 Ajax 的知识, Ajax 的工作原理相当于在用户和服务器之间加了一个中间层,使用户操作与服务器响应异步化。并不是所有的用户请求都提交给服务器,像一些数据验证和数据处理等都交给 Ajax 引擎自己来做,只有确定需要从服务器读取新数据时再由 Ajax 引擎代为向服务器提交请求。使用 Ajax 后用户会感觉所有的操作都会很快响应,而没有页面重新装载的等待。

Ajax 的核心构成技术有 JavaScript、XMLHttpRequest 和 DOM 等,本部分通过几个示例演示如何使用 Ajax 技术得到服务器中的数据,访问静态数据以前阶段学习中已经接触过,在此主要关注如何访问 JSP、Servlet 生成的动态数据。

14.1.1 访问静态数据

本示例演示如何通过 Ajax 访问得到 WEB 中的静态数据,在 Ajax 中静态数据主要包含普通文本和 xml 文本两种形式,以下分别演示了如何得到这两种形式的数据:

1、 得到文本数据

假设有 hello.html, 内容如下:

```
<h1>hello!</h1>
```

编写 Ajax 网页 ajax1.html 得到 hello.html 的内容,代码如下:

```
<h1>ajax1.html</h1>
<hr color="red">

<script type="text/javascript" language="javascript">
    var http_request = false;
    function makeRequest(url){
        http_request = new ActiveXObject("Microsoft.XMLHTTP");
        if (!http_request) {
            alert('Giving up :( Cannot create an XMLHTTP instance');
            return false;
        }
        http_request.onreadystatechange = alertContents;
        http_request.open('GET', url, true);
        http_request.send(null);
    }
    function alertContents() {
        if (http_request.readyState == 4) {
            if (http_request.status == 200) {
                //alert(http_request.responseText);
                span1.innerHTML=http_request.responseText;
            } else {
```

```

        alert('request 出现问题!');
    }
}
}
</script>
<span style="cursor: hand; text-decoration: underline"
    onclick="makeRequest('hello.html')">
    请求 hello.html
</span>
<br><br>
<span id="span1" style="BACKGROUND-COLOR: lavender; FONT-SIZE:
larger"></span>

```

解释:

- “请求 hello.html” 放在中, 且的 onclick 事件中关联了 javascript 函数, 当用户点击该文字时会通过 XMLHttpRequest 对象向服务器发送请求;
- span1.innerHTML=http_request.responseText;该语句将服务器返回的结果输出到 id 为”span1”的中。

运行效果如下:



图 14-1 通过 Ajax 得到文本数据

2、 得到 xml 数据

1) 假设有 hello.xml, 内容如下:

```

<?xml version="1.0" ?>
<root>hello.</root>

```

2) 编写 Ajax 网页 ajax1.html 得到 hello.xml 的内容, 并将内容显示在文本区中, 代码如下:

```

<h1>ajax2.html</h1>
<hr color="red">

```

```
<script type="text/javascript" language="javascript">
    var http_request = false;

    function makeRequest(url){
        http_request = new ActiveXObject("Microsoft.XMLHTTP");
        if (!http_request) {
            alert('Giving up :( Cannot create an XMLHTTP instance');
            return false;
        }
        http_request.onreadystatechange = alertContents;
        http_request.open('GET', url, true);
        http_request.send(null);
    }

    function alertContents() {
        if (http_request.readyState == 4) {
            if (http_request.status == 200) {
                var xmlDoc = http_request.responseXML;
                var root_node = xmlDoc.getElementsByTagName('root').item(0);
                //alert(root_node.firstChild.data);
                result.value=root_node.firstChild.data;
            } else {
                alert('request 出现问题!');
            }
        }
    }
}
</script>
<span style="cursor: hand; text-decoration: underline"
    onclick="makeRequest('hello.xml')">
    请求 hello.xml
</span>
<br>
<textarea id="result" rows="5" cols="60"></textarea>
```

3) 运行效果如下:



图 14-2 通过 Ajax 得到 xml 数据

14.1.2 访问动态数据

此部分阐述如何通过 Ajax 访问 WEB 服务器中由 JSP、Servlet 生成的动态数据，通过 XMLHttpRequest 对象发送请求的方法与访问静态数据相同，下面示例演示了如何访问动态数据：

1、 示例 1——得到 jsp 结果

1) 假设有 add.jsp，内容如下：

```
<%
    out.println("<h2>add.jsp</h2>");
%>
time:<%=new java.util.Date().toString()%><br>
```

2) 通过 Ajax 得到 add.jsp 的输出，代码如下：

```
<h1>ajax3.html</h1>
<hr color="red">
<script type="text/javascript" language="javascript">
    var http_request = false;
    function makeRequest(url){
        http_request = new ActiveXObject("Microsoft.XMLHTTP");
        if (!http_request) {
            alert('Giving up :( Cannot create an XMLHTTP instance');
            return false;
        }
        http_request.onreadystatechange = alertContents;
        http_request.open('GET', url, true);
        http_request.send(null);
    }
    function alertContents() {
```

```

        if (http_request.readyState == 4) {
            if (http_request.status == 200) {
                result.innerHTML=http_request.responseText;
            } else {
                alert('request 出现问题!');
            }
        }
    }
}
</script>
<span style="cursor: hand; text-decoration: underline"
    onclick="makeRequest('add.jsp')">
    请求 add.jsp
</span>
<br><br><br>
<span id="result" style="FONT-SIZE: larger"></span>

```

解释:

- 语句 `result.innerHTML=http_request.responseText;`，将结果显示到 `` 中，且输出字符中的 HTML 格式保持有效。

3) 运行效果如下:



图 14-3 使用 Ajax 访问 JSP

2、 示例 2——访问 Servlet

本示例向 servlet 传递两个加数，servlet 将两数的和返回给浏览器；

AddServlet.java 核心代码如下：

```
//接收两个整数
```

```
String num1=request.getParameter("num1");
String num2=request.getParameter("num2");
int n1=Integer.parseInt(num1);
int n2=Integer.parseInt(num2);
//得到输出流
PrintWriter out=response.getWriter();
out.println(num1 + "+" + num2 + "=" + (n1+n2) );
```

通过 Ajax 向 servlet 传递参数并得到输出结果，代码如下：

```
<h1>ajax4.html</h1>
<hr color="red">
<script type="text/javascript" language="javascript">
    var http_request = false;
    function makeRequest(url){
        http_request = new XMLHttpRequest("Microsoft.XMLHTTP");
        if (!http_request) {
            alert('Giving up :( Cannot create an XMLHTTP instance');
            return false;
        }
        http_request.onreadystatechange = alertContents;
        http_request.open('POST', url, true);
        http_request.setRequestHeader("Content-Type",
            "application/x-www-form-urlencoded");
        http_request.send("num1=10&num2=20");
    }
    function alertContents() {
        if (http_request.readyState == 4) {
            if (http_request.status == 200) {
                result.innerHTML=http_request.responseText;
            } else {
                alert('request 出现问题!');
            }
        }
    }
</script>
<span style="cursor: hand; text-decoration: underline"
    onclick="makeRequest('addservlet')">
    请求 addservlet 求两数之和.
```

```

</span>
<br><br><br>
<span id="result" style="FONT-SIZE: larger"></span>

```

解释:

- 语句 `http_request.setRequestHeader("Content-Type","application/x-www-form-urlencoded");`用于设置发送的请求中的数据类型不是普通文本类型。
- 语句 `http_request.send("num1=10&num2=20");`向 WEB 服务器发送两个参数。

通过 XMLHttpRequest 将请求发送给 servlet 后, 浏览器得到结果并显示, 结果如下:



图 14-4 访问 Servlet

14.2 AJAX 框架和 DWR

14.2.1 框架的意义

通过以上 4 个示例可以看出: 手工编写 AJAX 代码必须处理许多潜在的问题:

1、 编写复杂、容易出错

javascript 是基于对象的语言, 而不是面向对象的, 对 OOP 的支持很少, 这就限制了 javascript 代码的可重用可封装等特性, 而且 JavaScript 没有专门的 Debug 软件, 导致在编码和调试 Javascript 代码上面耗费过多的时间。

2、 浏览器标准不统一

各种浏览器之间存在差异, 支持的 css 不一样, 支持的客户端脚本不一样。这就需要开发者编写代码处理浏览器兼容性的问题。

AJAX 框架为开发者提供一种简单的方式使用 Ajax 和 XMLHttpRequest, 框架中包含了许多现成的通用的代码, 通过在开发过程中使用框架, 开发者可以重用框架代码, 大大减少 javascript 的编码量, 从而减少编码出错率, 提高开发效率。

14.2.2 DWR 框架

DWR(Direct Web Remoting)是一个 WEB 远程调用框架.利用这个框架可以让 AJAX 开发变得很简单.利用 DWR 可以在客户端利用 JavaScript 直接调用服务端的 Java 方法并返回值给 JavaScript 就好像直接本地客户端调用一样(DWR 根据 Java 类来动态生成 JavaScript 代码)。DWR 在客户端 javascript 和

服务器端 java 类中间起了一个协调作用。DWR 可以声明哪些类可以供 javascript 调用。

DWR 可以与 java web 应用无缝配合，对缺少 DHTML 编程经验的开发者来说，DWR 提供了一个 JavaScript 库包含了常用的 DHTML 代码。

14.2.3 DWR 框架程序示例

要使用 DWR 框架，必须有该框架的资源（java 包、javascript 库），这些资源可以从网上下载到（官方网址为：<https://dwr.dev.java.net>）。下面通过示例演示在 Java WEB 应用中使用 DWR 框架的步骤。

- 1) 创建 WEB 站点；
- 2) 将 DWR 支持库复制到网站 WEB-INF\lib 目录下，DWR 库的通常为 dwr.jar、dwr-1.0RC2.jar、dwr-2.0.jar 等名字。在此使用 dwr-1.0RC2.jar，其内部集成了 javascript 库。
- 3) 在 WEB.XML 中配置 DWRServlet，该 servlet 可以将 Ajax 发送的请求接收到框架进行处理，配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Ajax Examples</display-name>
  <servlet>
    <servlet-name>dwr-invoker</servlet-name>
    <display-name>DWR Servlet</display-name>
    <description>Direct Web Remoter Servlet</description>
    <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
    <init-param>
      <param-name>debug</param-name>
      <param-value>true</param-value>
    </init-param>
  </servlet>
  <servlet-mapping>
    <servlet-name>dwr-invoker</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>
</web-app>
```

- 4) 开发一个 JAVA 类，其中包含要被 Ajax 调用的方法，代码如下：

```
package com.test;

public class Demo {
    public String hello(){
```

```
        return "你好,我来自 com.test.Demo 类.";
    }
}
```

5) 在 WEB-INF 目录下创建名为 dwr.xml 的配置文件,在该文件中可以配置要将哪些类共享出去让 Ajax 访问,配置如下:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 0.4//EN"
    "http://www.getahead.ltd.uk/dwr/dwr.dtd">
<dwr>
    <allow>
        <create creator="new" javascript="Test" scope="application">
            <param name="class" value="com.test.Demo" />
        </create>
    </allow>
</dwr>
```

解释:

- 每个<create>声明一个可供 javascript 调用的类, javascript="Test"表明:在 javascript 中可以用 Test 调用类 Test 的方法,creator="new"表明每个请求都新建一个 Test 实例。

6) 编写页面 dwr.jsp(也可以是普通的 html 页),通过 Javascript 调用 Java 类:

```
<html>
<head>
<title>DWR 示例</title>
<script type='text/javascript' src='dwr/engine.js'> </script>
<script type='text/javascript' src='dwr/interface/Test.js'></script>
<script type='text/javascript' src='dwr/util.js'> </script>
<script type="text/javascript">
function callBack(data){
    result.value=data;
}
</script>
</head>
<body>
<input type="Button" name="button3" value="测试"
    onclick="Test.hello(callBack)" />
<br>
```

```
<textarea id="result" rows="5" cols="60"></textarea>
</body>
</html>
```

解释:

- onclick="Test.hello(callBack)", 指明在点击按钮时, 调用服务器端的 Test 类的 hello()方法。
- Test.hello(callBack)中的参数 callBack 指明了由名为“callBack”的 javascript 函数来接收服务器端的回应。

7) 配置完成, 启动 Web 应用, 输入 <http://localhost:8080/ajax-demo/dwr.jsp>: 效果如下(服务器的回应显示在了文本区中):

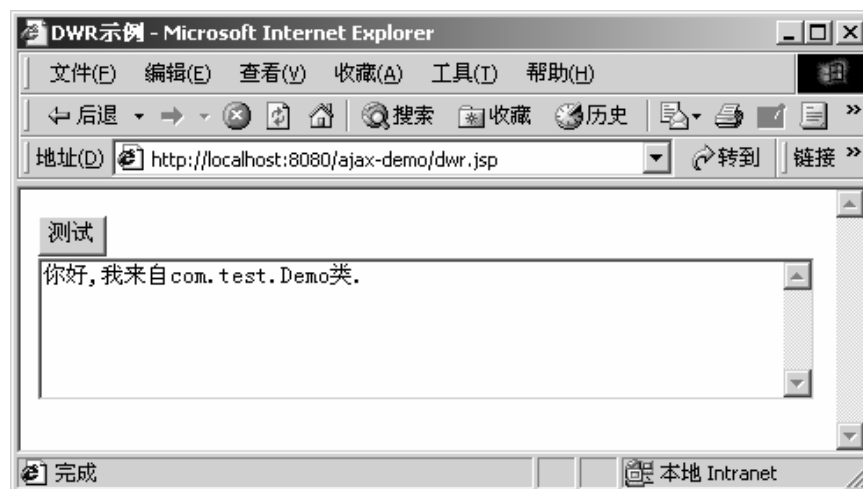


图 14-5 使用 DWR 框架

小结

通过 Ajax 可以访问服务器中的静态数据, 也可以访问由 JSP、Servlet 动态生成的数据。

使用 Ajax 框架开发 WEB 应用可以大大减少编码量、减少错误率、加快开发速度。

DWR 框架可以在客户端远程利用 JavaScript 直接调用服务端的 Java 方法并返回值给 JavaScript。

课后练习

- 1、 DWR 框架解决的主要问题是什么？
- 2、 简述在 Java WEB 应用中使用使用 DWR 框架的步骤？



第一章 web 运行模式

学习目标

- 安装使用 tomcat 服务
- 配置 MyEclipse
- 创建第一个 web 应用

1.1 模拟实验

安装 tomcat

在开始安装之前,先准备 J2SDK 和 TOMCAT 两个软件,如果已经安装了 J2SDK,就只需 TOMCAT 即可。

运行 jakarta-tomcat-5.0.28.exe 按照提示安装,这里选择了 Service,就是作为 Windows 服务来运行。

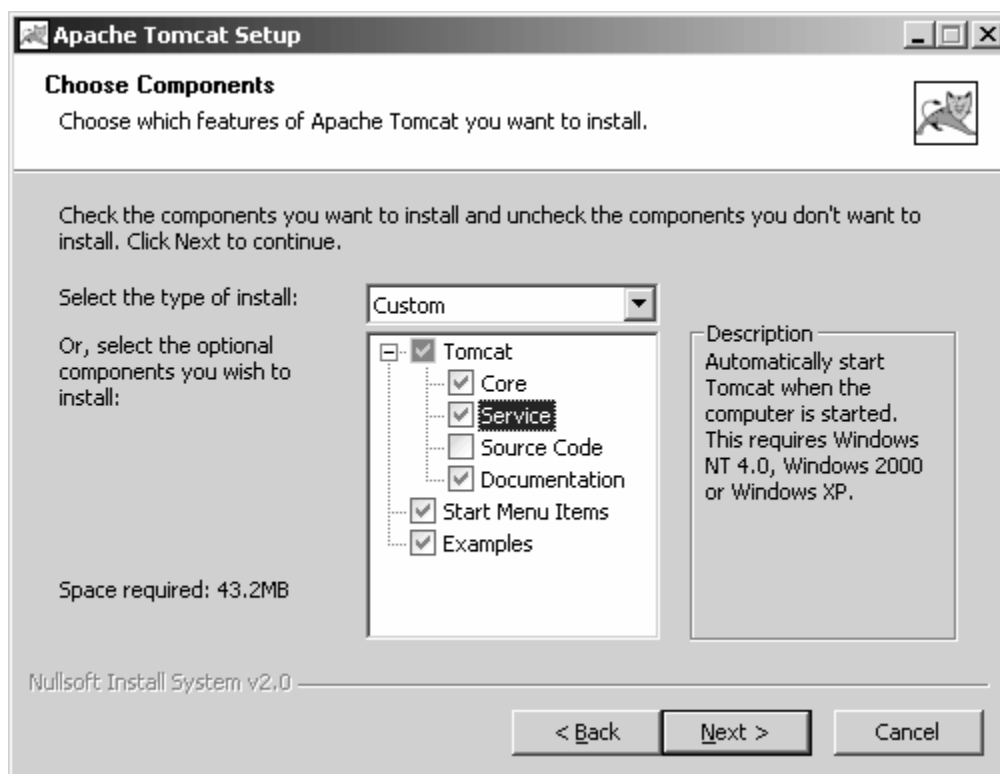


图 1-1 tomcat 安装组件选择界面

如果要改变安装路径,可以在这个步骤操作。

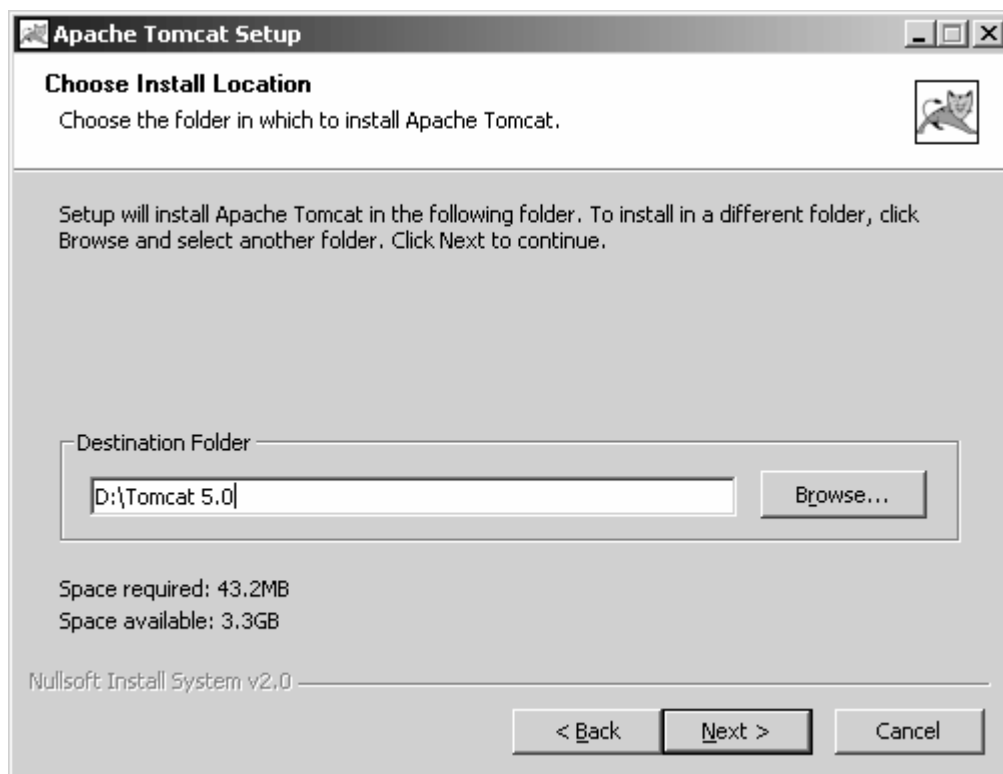


图 1-2 tomcat 安装路径界面

在这里设置 TOMCAT 使用的端口以及 WEB 管理界面用户名和密码，请确保该端口未被其他程序占用

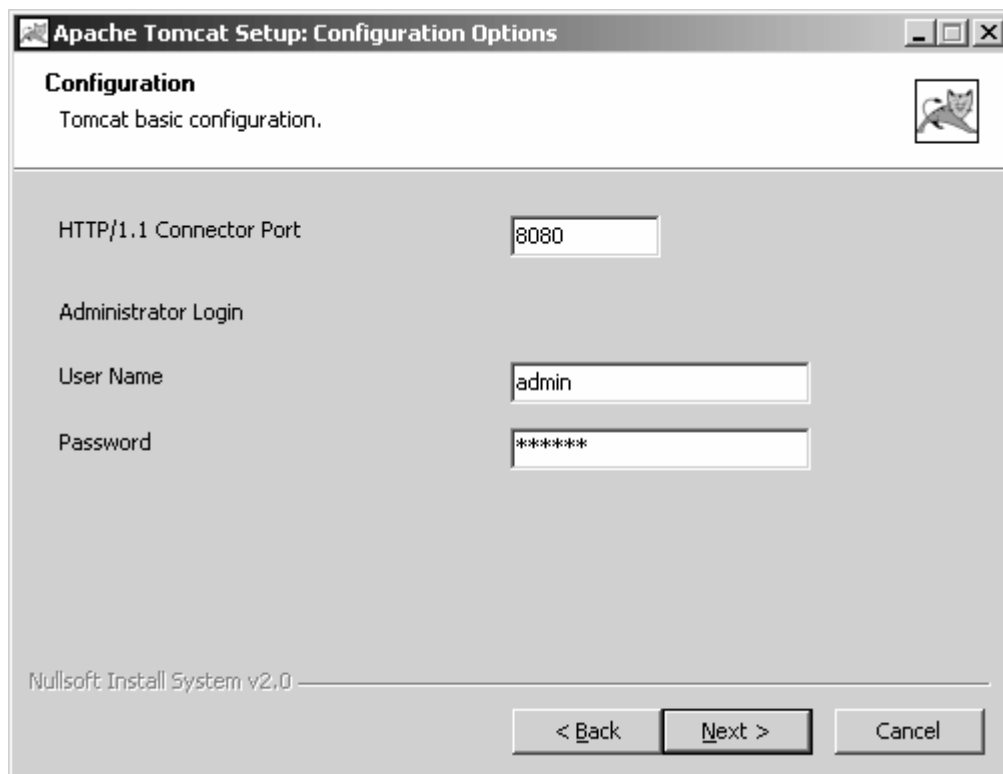


图 1-3 tomcat 安装端口，密码设置界面

选择 J2SDK 的安装路径，安装程序会自动搜索，如果没有正确显示，则可以手工修改，这里改

为 d:\j2sdk1.4.2_04

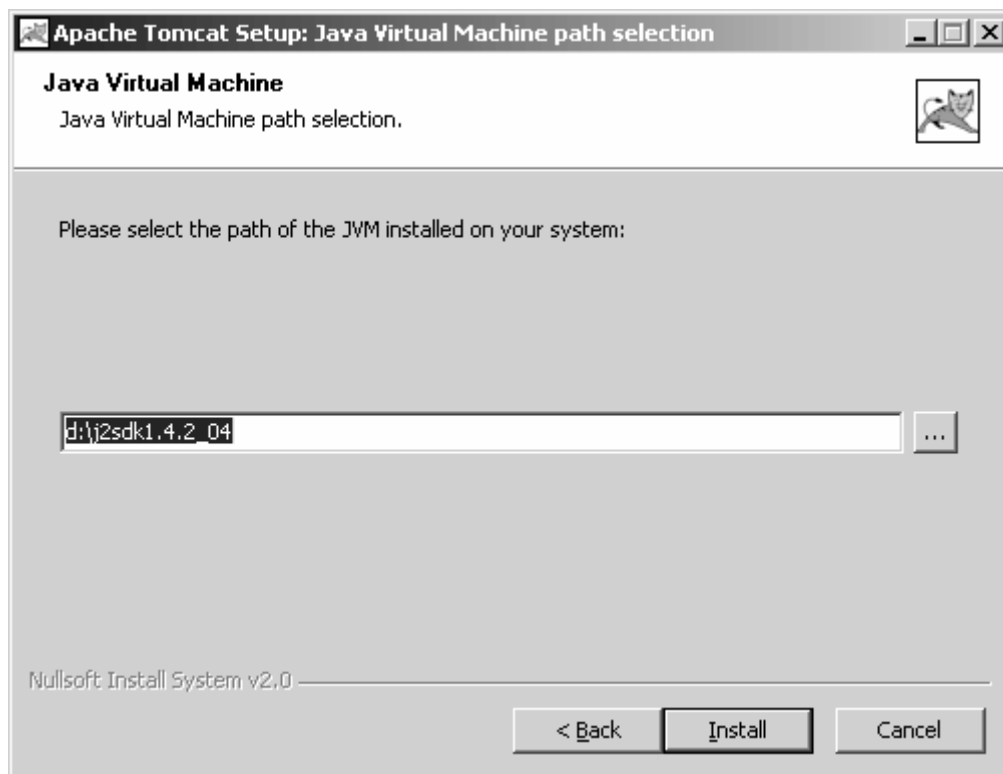


图 1-4 tomcat 安装 JDK 路径选择界面

接下来就开始拷贝文件了，成功安装后，程序会提示启动 tomcat 并查看 readme 文档。Tomcat 正常启动后会在系统栏加载图标。



图 1-5 tomcat 服务界面

至此安装与配置都已完成，打开浏览器输入：<http://localhost:8080> 即可看到 TOMCAT 的欢迎页面。

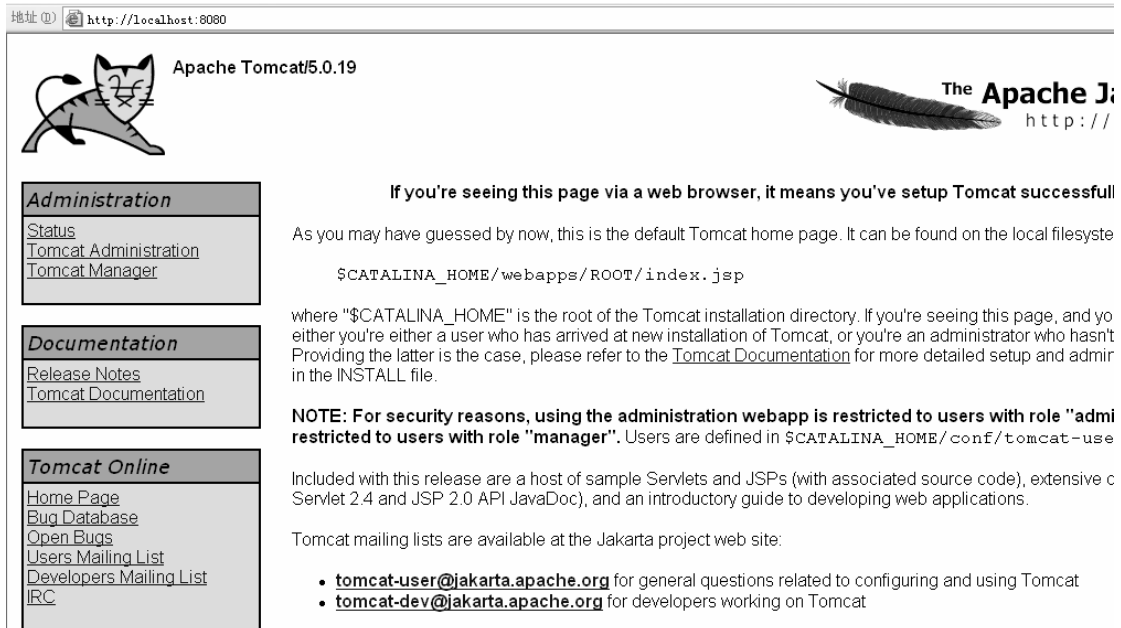


图 1-6 tomcat 欢迎界面

安装 MyEclipse

MyEclipse可以在 <http://www.myeclipseide.com/> 网站上下载，然后就可以安装了。

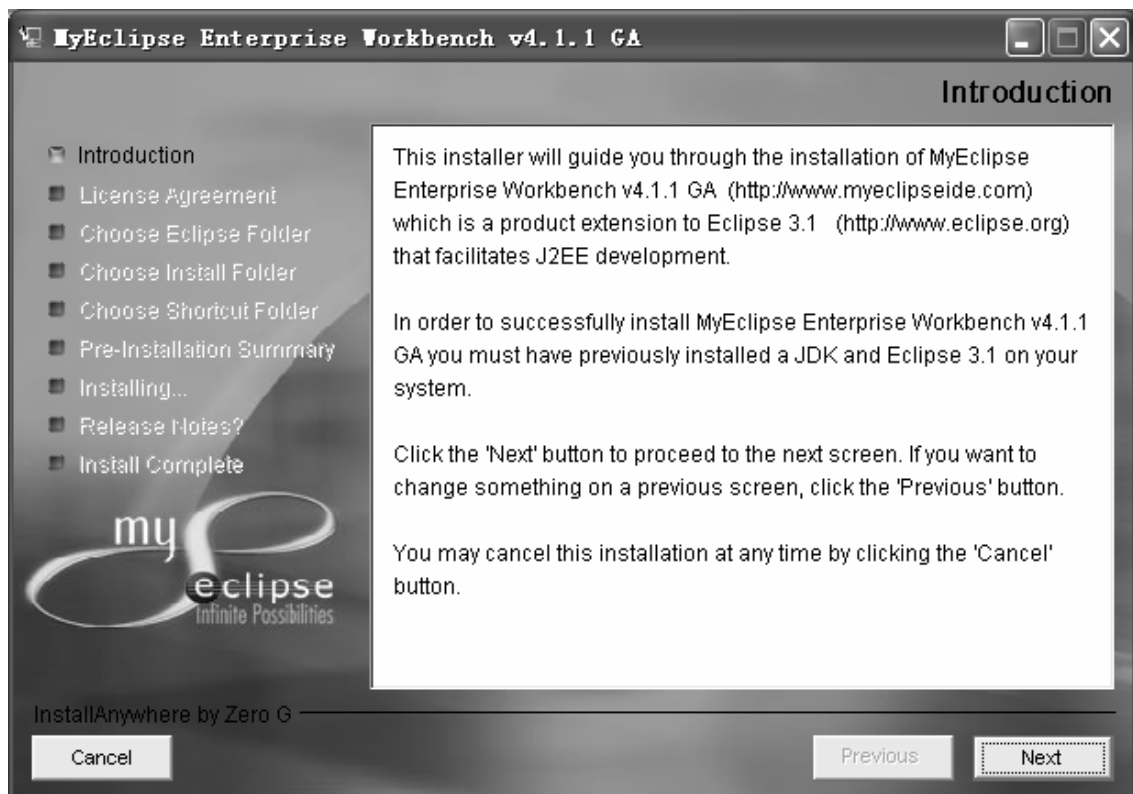


图 1-7 myEclipse 安装界面

选择相应的 eclipse 路径。

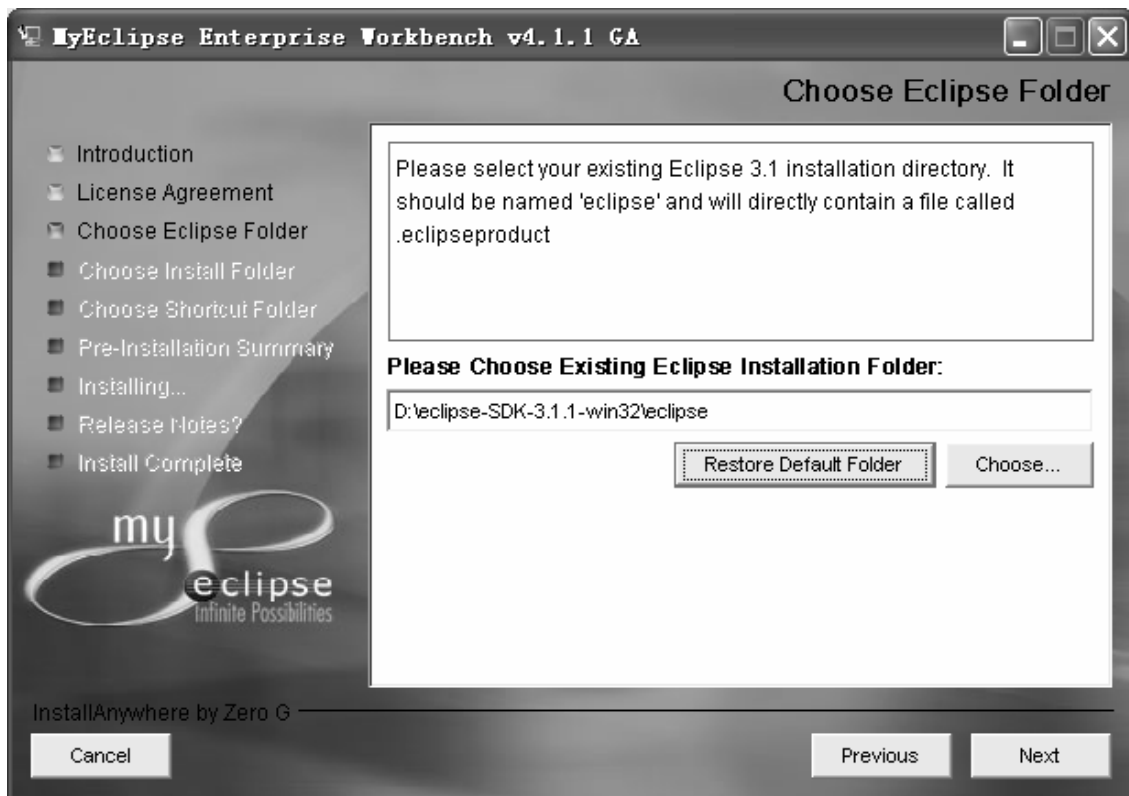


图 1-8 选择 eclipse 目录

选择 MyEclipse 的安装路径，建议就安装在 eclipse 文件夹内。

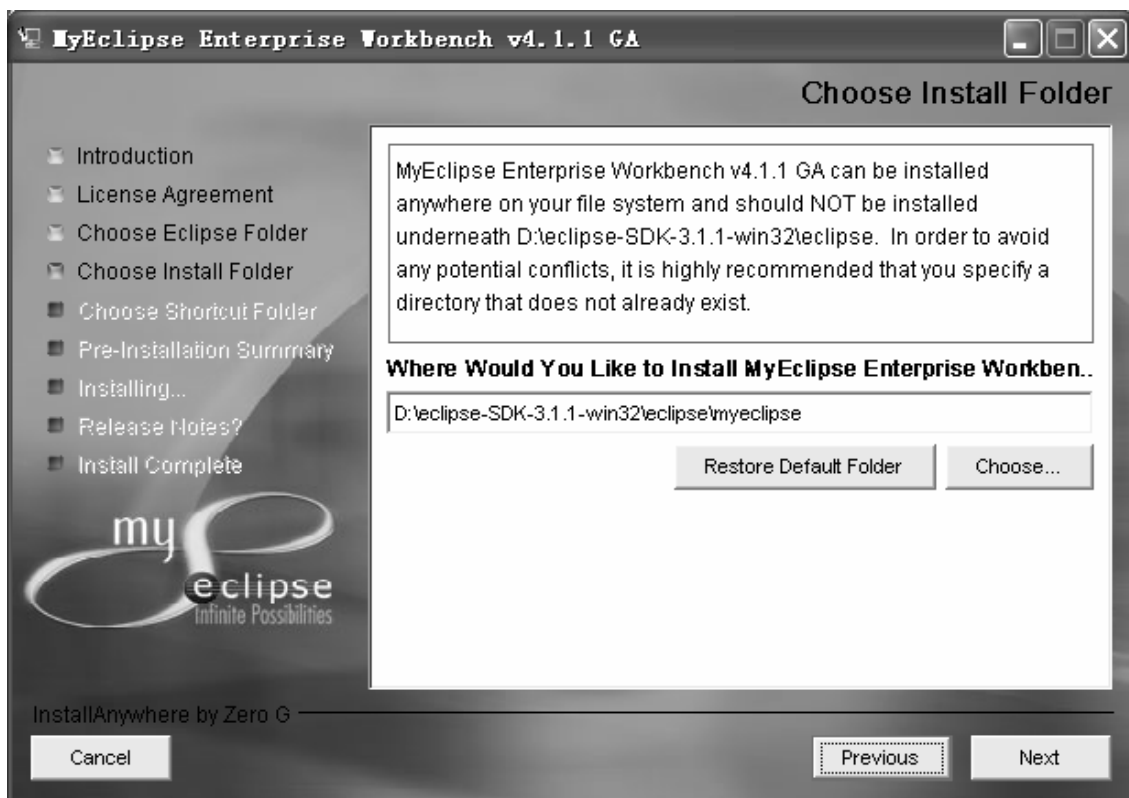


图 1-9 选择 MyEclipse 的安装目录

安装完成后，再运行 Eclipse，会看到在菜单栏中多出了 MyEclipse 选项，工具栏中也多出了一些

按钮。为了使 MyEclipse 能够支持 Tomcat 服务器，还需要作一些配置工作。选择菜单中的“窗体”选项，在展开的菜单项中选择“首选项”，进入配置窗体。

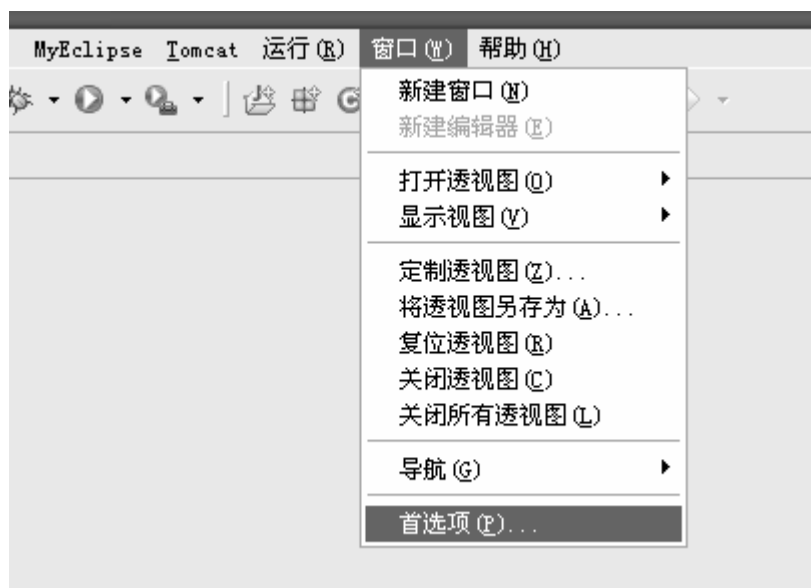


图 1-10 进入 MyEclipse 的配置界面

在左边树型目录中，展开“MyEclipse”选项，在展开其“Application servers”应用服务器选项。找到 Tomcat5，按照图 1-11 配置 Tomcat 的路径，并将其激活，这样就可以在 MyEclipse 中使用 Tomcat 服务了。

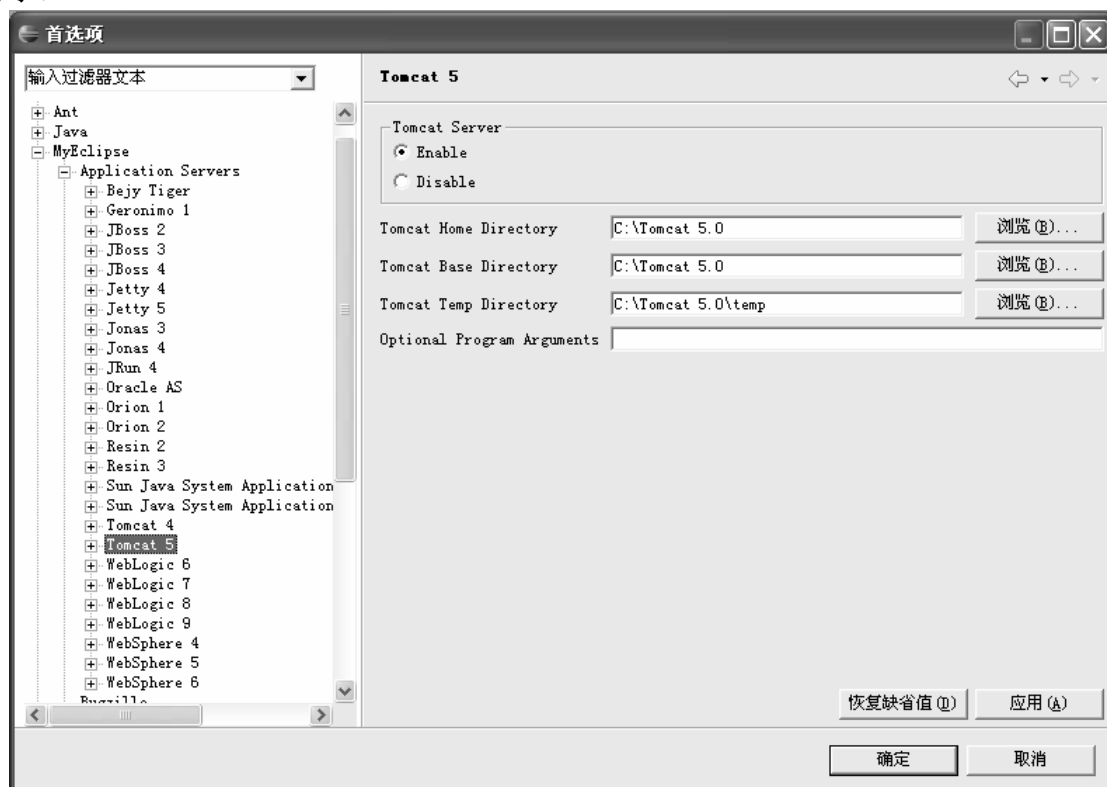


图 1-11 tomcat 的配置页面

Tomcat 配置好后可以创建 web 项目了。新建一个 java 项目。



图 1-12 新建 Tomcat 工程

选择 MyEclipse 下面的 web 工程，利用 MyEclipse 插件来新建 web 项目。



图 1-13 创建 web 项目

参照图 1-14 指定 web 工程的工程名，也可以更改工程中的其它设置，如源程序的存放位置等。

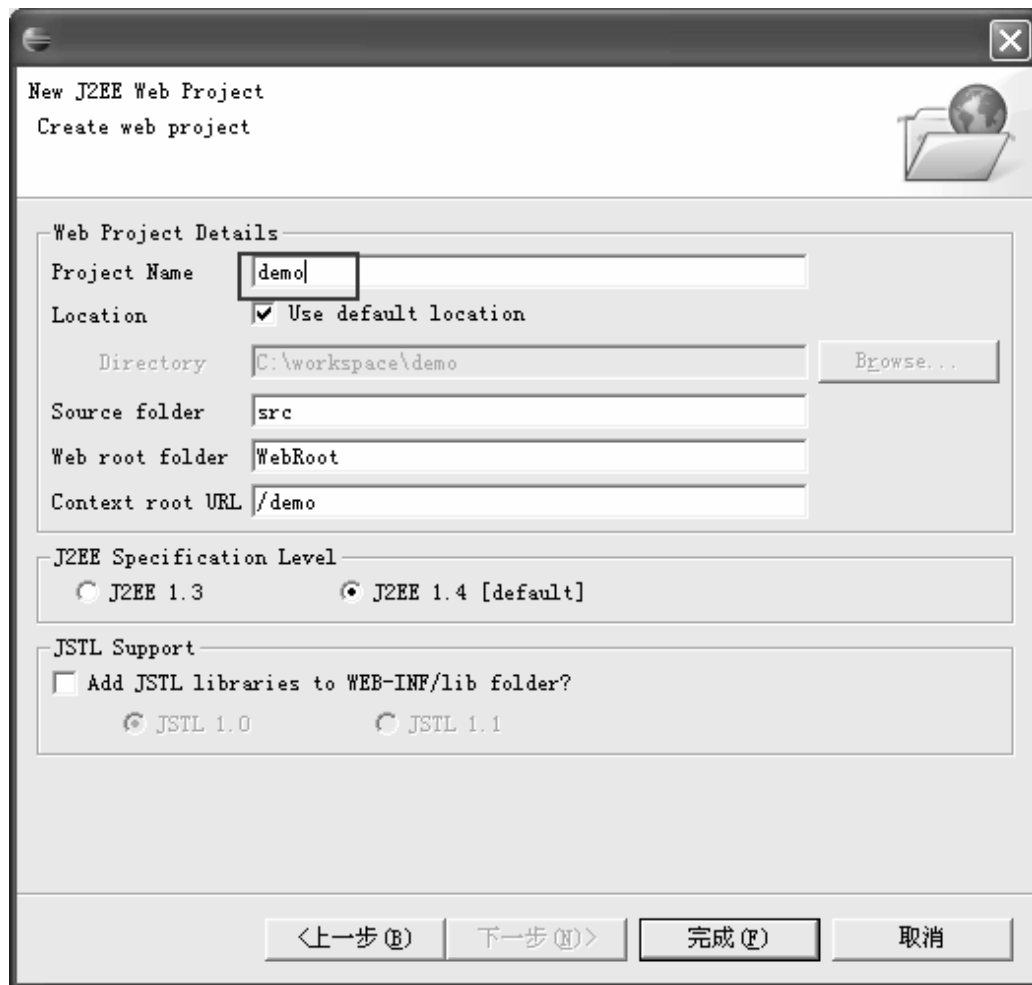


图 1-14 指定工程名

创建好的空 web 工程，包含必要的库和相应的文件夹。

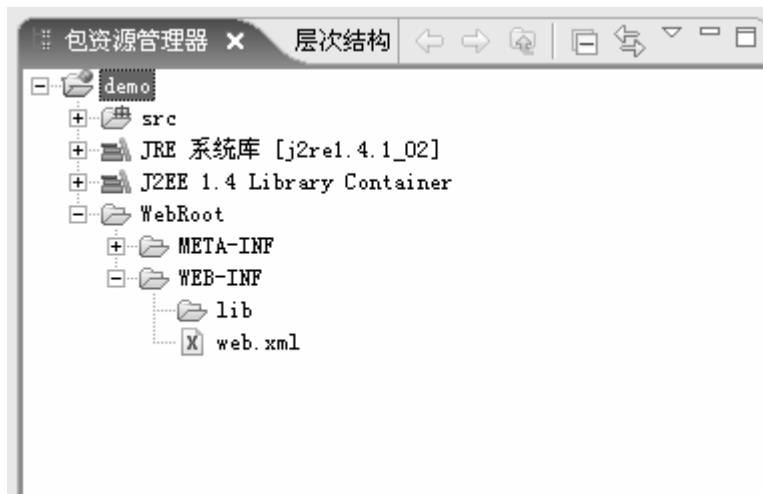


图 1-15 空工程

创建 web 组件 Servlet 类，作为 web 应用的主程序。

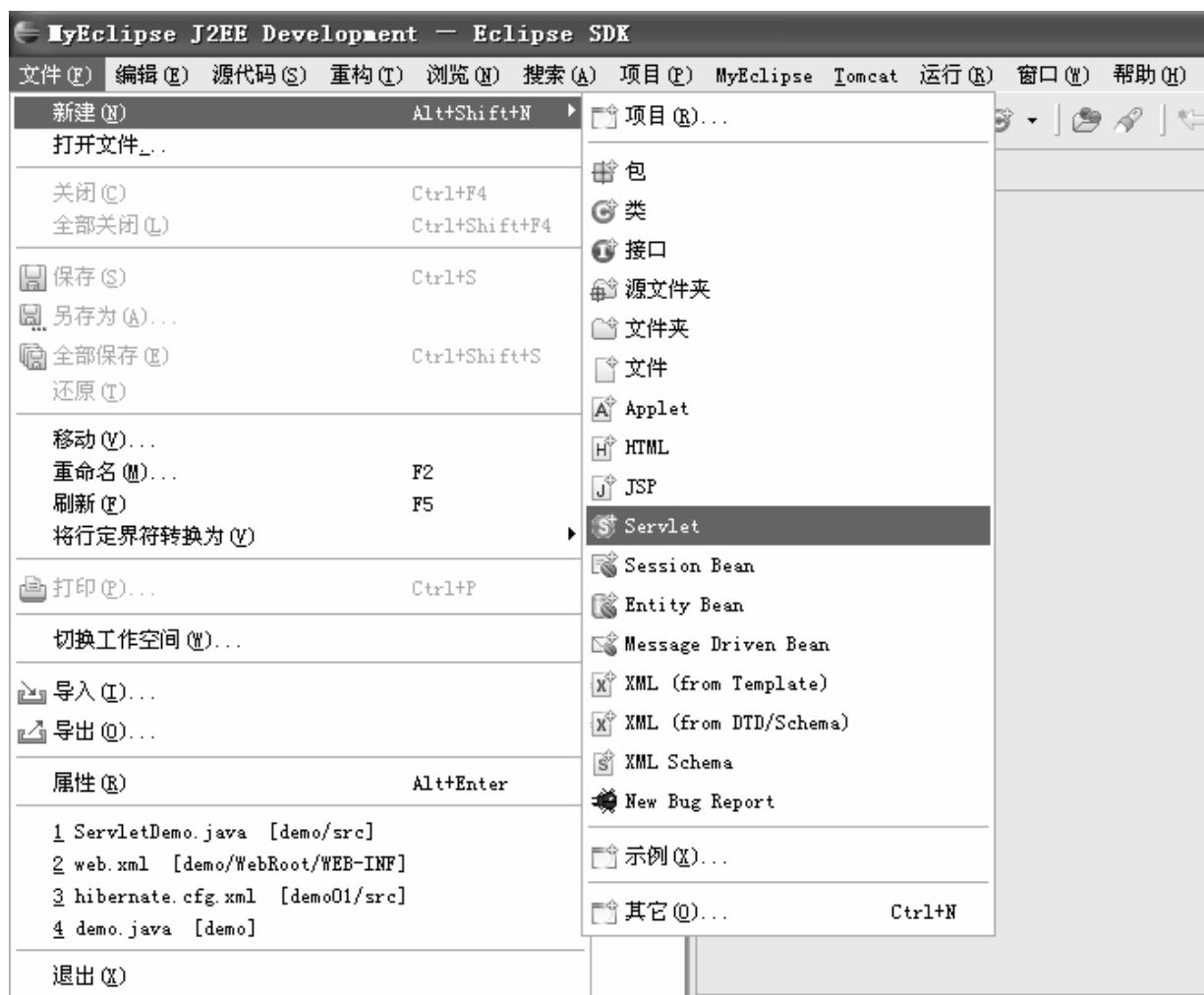


图 1-16 创建 Servlet 类

设置 Servlet 类名和所在包的名字，也可以在这个页面设置 Servlet 类中包含的成员。



图 1-17 Servlet 类的设置页面

上述内容设置成功后，就可以设置该 Servlet 类的 web 访问路径了。比如 Servlet 的映射路径，即它的 HTTP 路径。这里可以采用默认值，如果要修改的话，也可以随后在 web 应用的配置文件中更改。

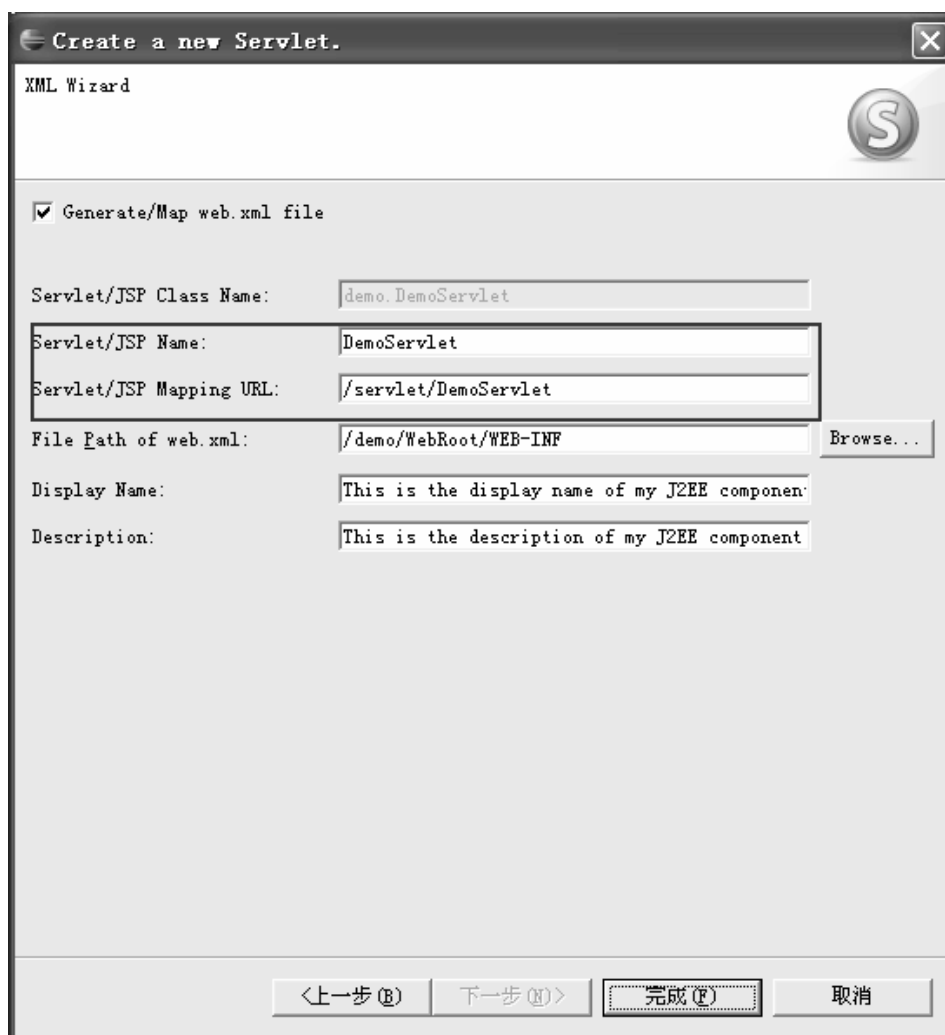


图 1-18 Servlet 的映射路径

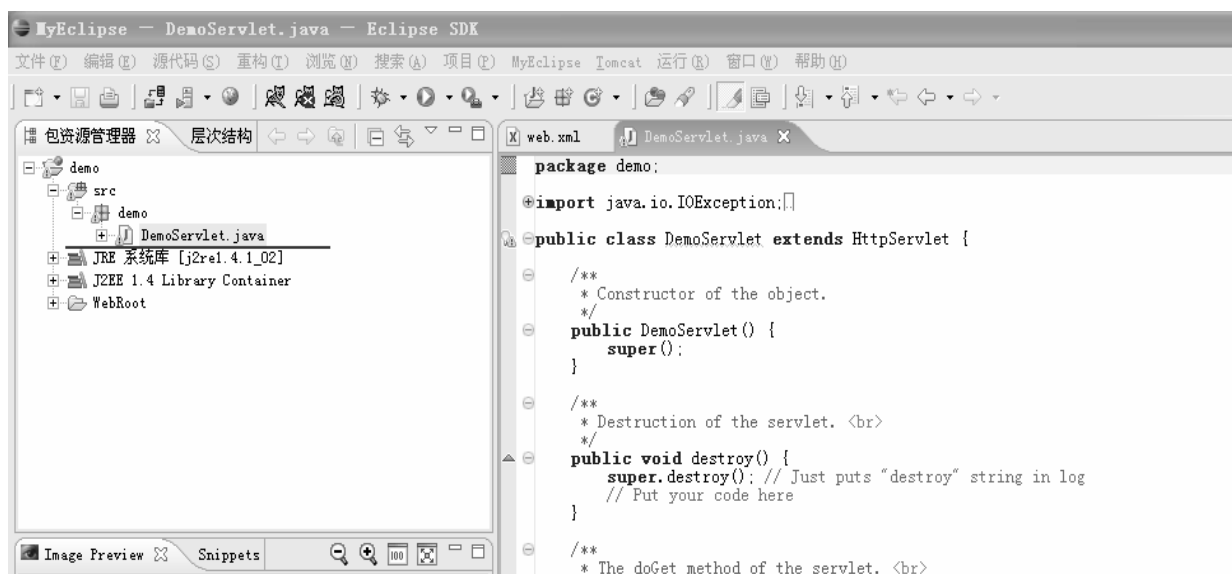


图 1-19 创建好的 Servlet 类

MyEclipse 会自动创建 Servlet 代码，包含之前选择的成员。这个类不用作任何修改，就可以部署在 web 服务器上。

```
package demo;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class DemoServlet extends HttpServlet {

    /**
     * Constructor of the object.
     */
    public DemoServlet() {
        super();
    }

    /**
     * Destruction of the servlet. <br>
     */
    public void destroy() {
        super.destroy(); // Just puts "destroy" string in log
        // Put your code here
    }

    /**
     * The doGet method of the servlet. <br>
     *
     * This method is called when a form has its tag value method equals to get.
     *
     * @param request the request send by the client to the server
     * @param response the response send by the server to the client
     * @throws ServletException if an error occurred
     * @throws IOException if an error occurred
     */
}
```

```

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

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out
            .println("<!DOCTYPE    HTML    PUBLIC    \"/>

```

```

Transitional//EN">");

    out.println("<HTML>");

    out.println("  <HEAD><TITLE>A Servlet</TITLE></HEAD>");

    out.println("  <BODY>");

    out.print("    This is ");

    out.print(this.getClass());

    out.println(", using the POST method");

    out.println("  </BODY>");

    out.println("</HTML>");

    out.flush();

    out.close();

}

/**
 * Initialization of the servlet. <br>
 *
 * @throws ServletException if an error occure
 */
public void init() throws ServletException {
    // Put your code here
}

}

```

Web 应用创建好后需要部署在 web 服务器上，利用 MyEclipse 提供的功能，点击图 1-20 中的部署按钮进行部署。

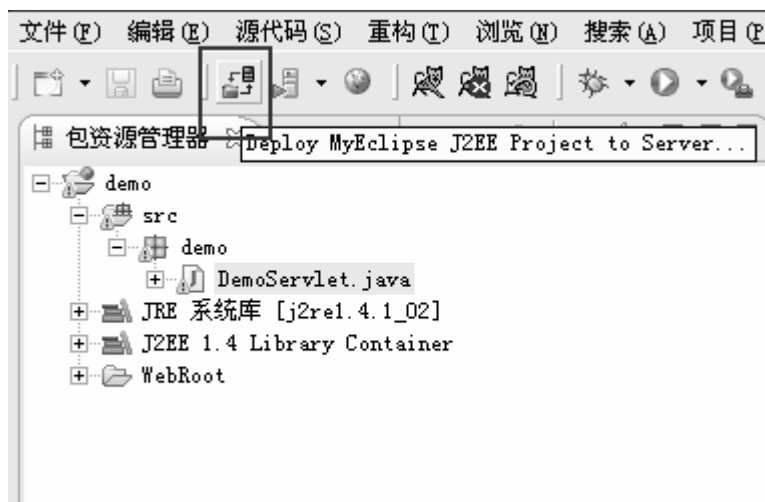


图 1-20 部署按钮

从下拉列表框中选择要部署的 web 应用，点击 add 按钮，进入 web 服务器选择页面。

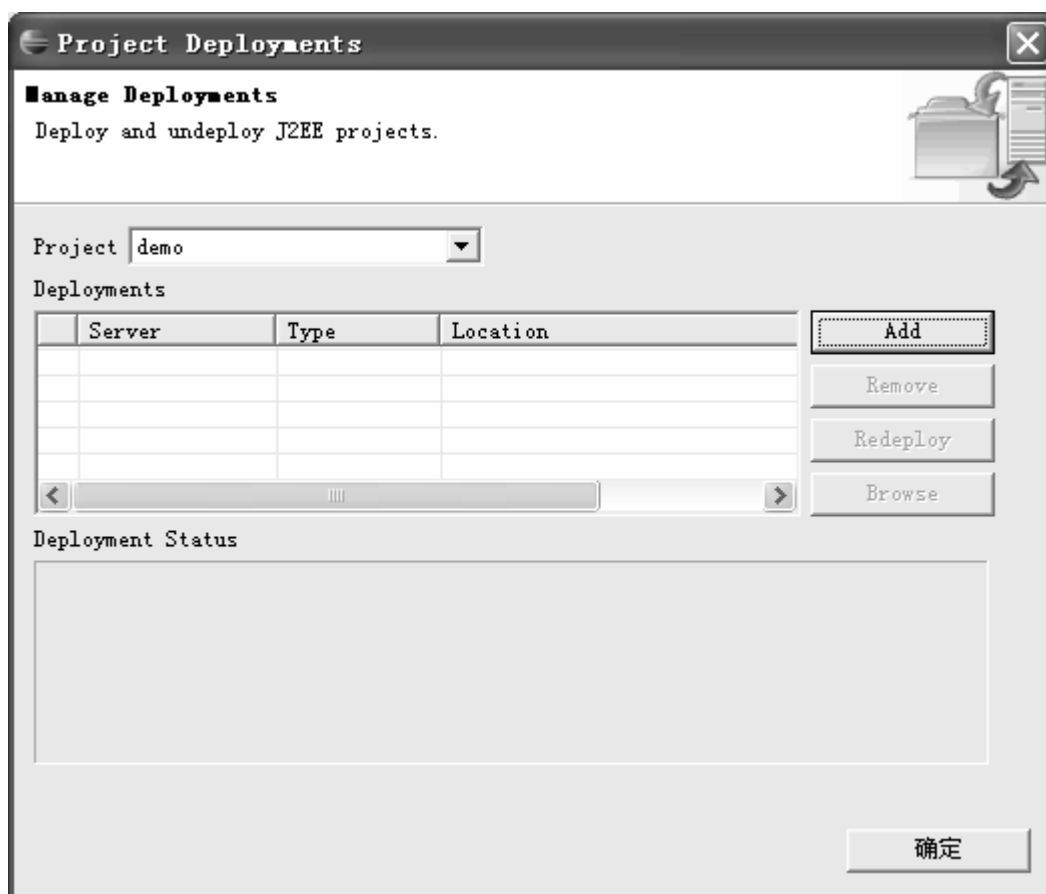


图 1-21 web 应用部署页面

选择目标 web 服务器，这里使用之前配置好的 Tomcat 作为这个应用的 web 服务器。

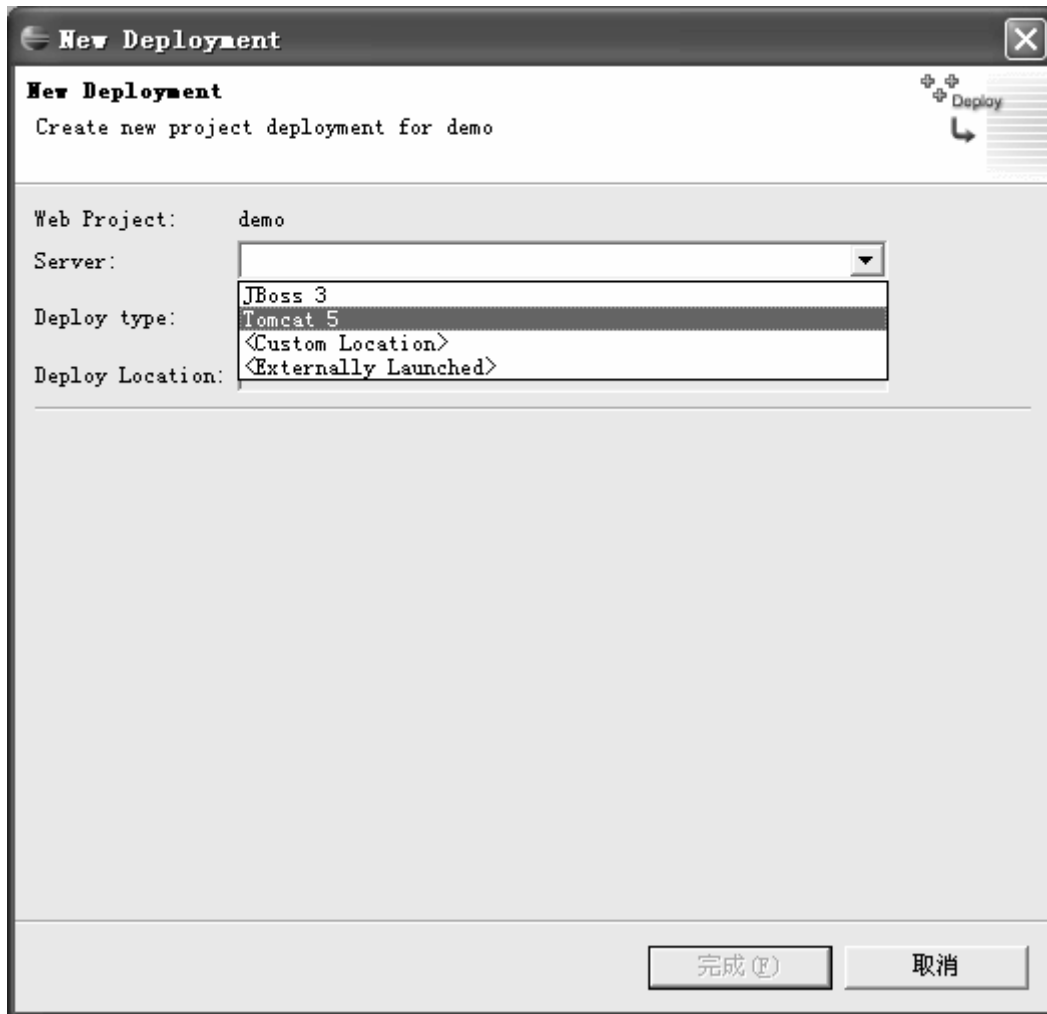


图 1-22 web 应用的部署

点击完成按钮，MyEclipse 会把当前 web 应用的内容拷贝到 web 服务器上。

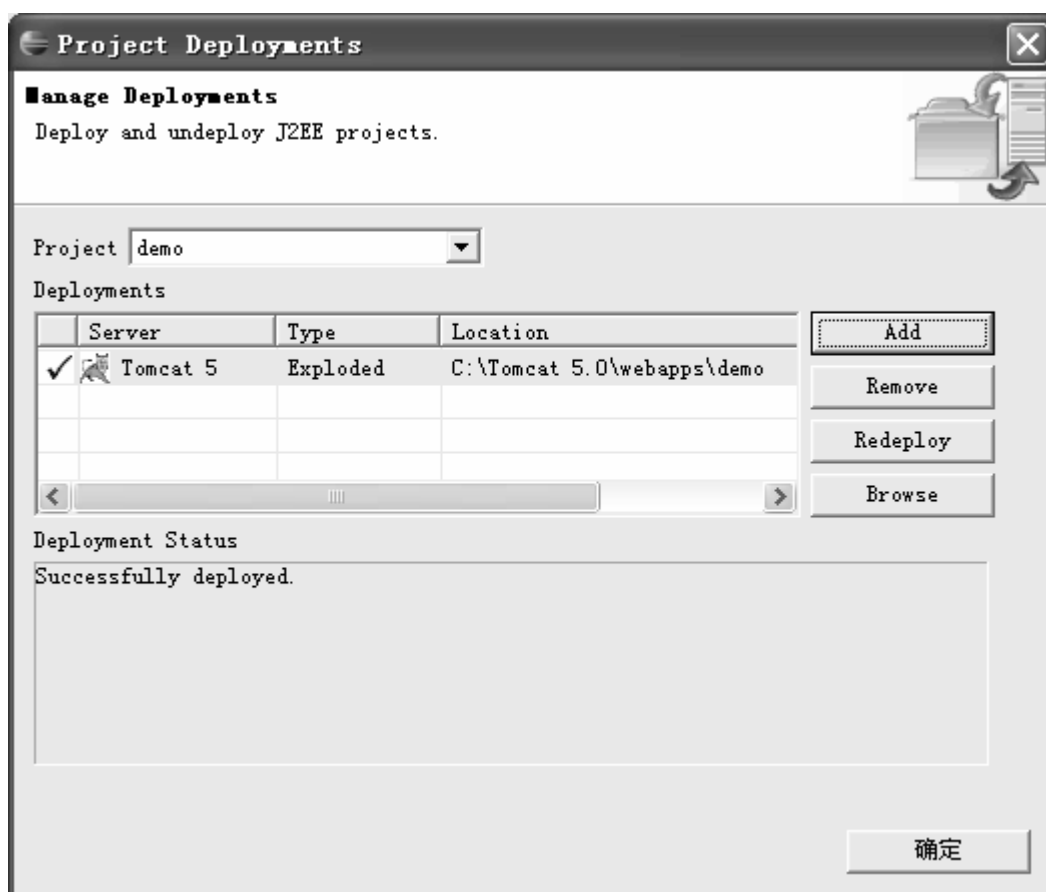


图 1-23 web 应用部署

然后需要启动 Tomcat web 服务器，开启 web 服务，接受客户端请求。可以直接在 Eclipse 中启动 Tomcat。

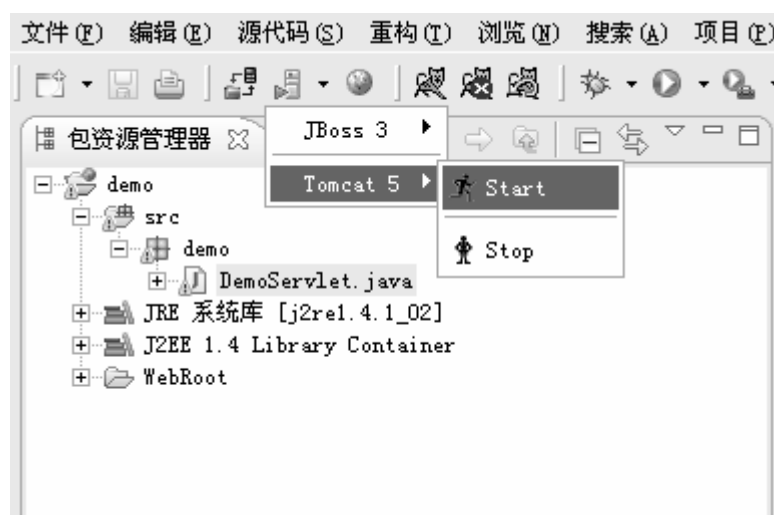


图 1-24 通过工具栏启动 tomcat

Tomcat 成功启动，注意提示性文字的最后一行：Server startup in 12345ms，指明 tomcat 服务已启动成功，数字代表启动时间。

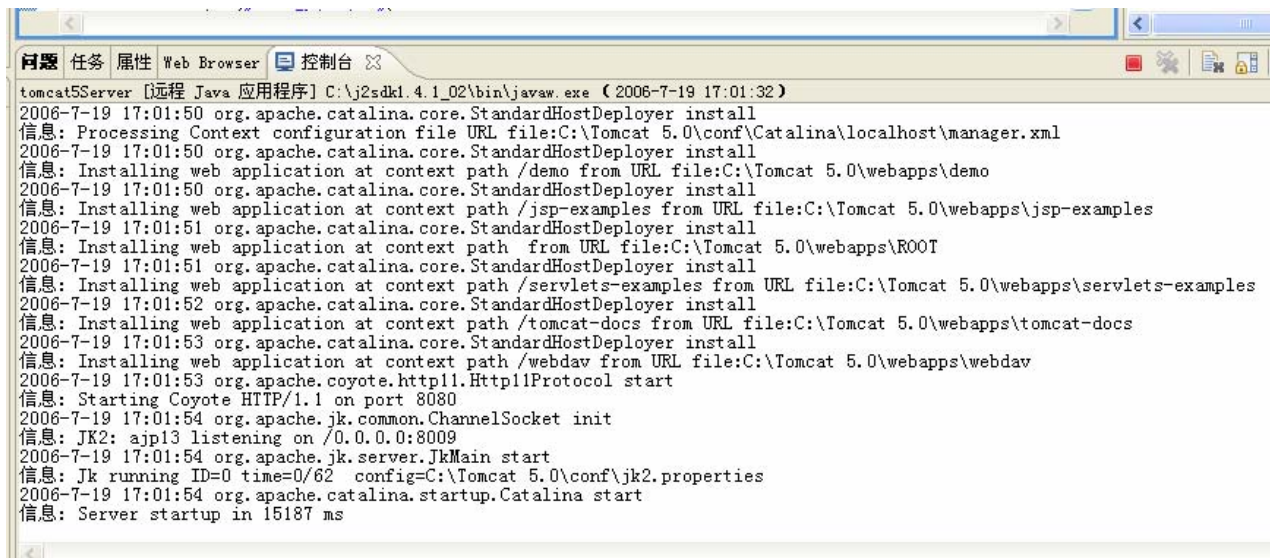


图 1-25 Tomcat 启动成功

启动成功后,就可以通过 IE 浏览器访问 web 应用了。在 HTTP 地址栏中写入 HTTP://localhost:8080/ 加上 web 应用的工程名,再加上在 图 1-18 中设置的 Servlet 的映射名,就构成完整的 HTTP 地址,通过这一地址就可以访问当前的 web 应用。

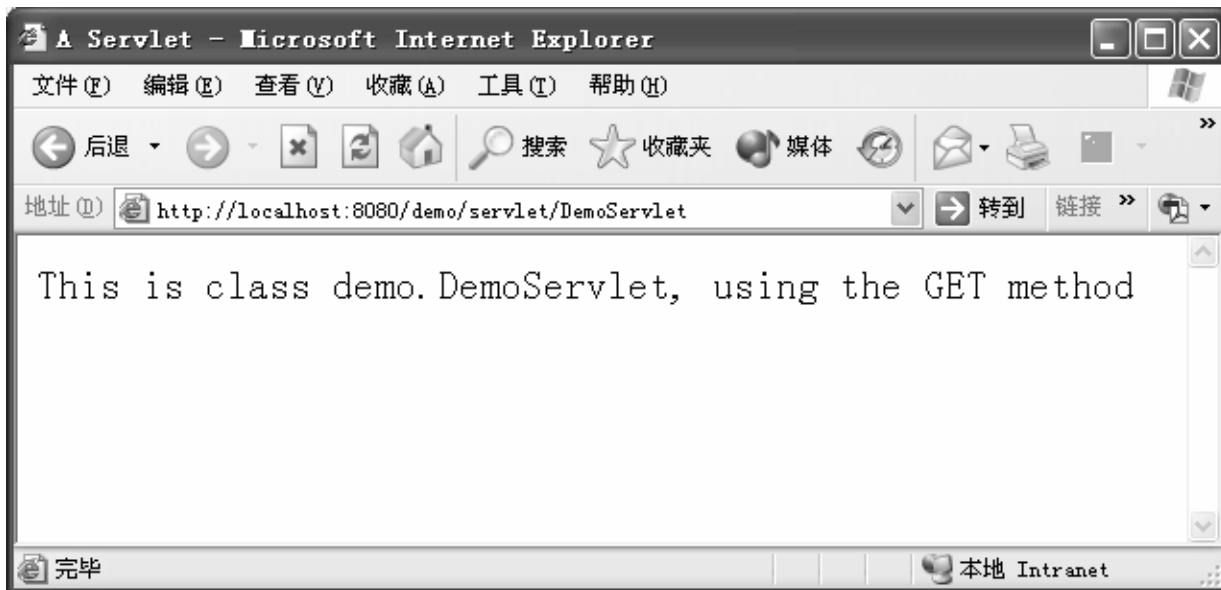


图 1-26 Servlet 的响应页面

1.2 实战实验

- 1、 打开 Eclipse 创建一个 web 工程，名为 HelloWorld。
- 2、 在工程中创建一个 Servlet 类，放在包 demo 下，类名为 HelloWorld
- 3、 参照图 1-18 中更改 Servlet 的映射名为 Hello
- 4、 然后找到这部分代码

```
out.print(" This is ");  
out.print(this.getClass());  
out.println(", using the GET method");
```

- 5、 替换成

```
out.println("Welcome to java web world");
```

- 6、 保存，部署 web 应用
- 7、 打开 IE 浏览器，键入 HTTP://localhost:8080/HelloWorld/Hello 访问 web 应用。

1.3 实验后任务

- 1、 在家里的电脑上安装 tomcat 和 myEclipse。

第二章 Servlet 概述

学习目标

- 理解 Servlet 开发部署过程
- 使用 MyEclipse 创建 Servlet
- 发布 web 应用
- 运行 web 应用

2.1 模拟实验

创建 WEB 站点和 Servlet

本部分使用手工方式创建一个 web 站点，以便于理解。

- 1、在 TOMCAT_HOME\$\webapps 下创建文件夹 MyWeb，并在 MyWeb 下创建子文件夹 WEB-INF，在 WEB-INF 文件夹下创建一个空的 web.xml，在 WEB-INF 文件夹下创建一个空的 classes 子文件夹，该文件夹用于存放 Servlet 类。

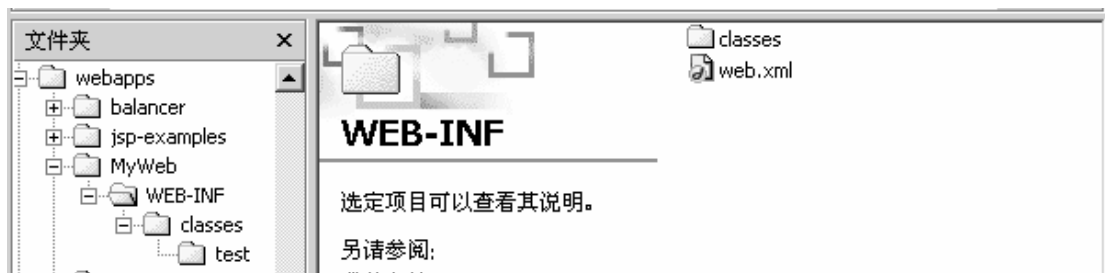


图 2-1 站点目录结构

- 2、编写、编译 Servlet

- 1) 将 ServletAPI 包放在 CLASSPATH 类路径中

该包在 TOMCAT_HOME\$\COMMON\lib 下，如图 2-2 所示：

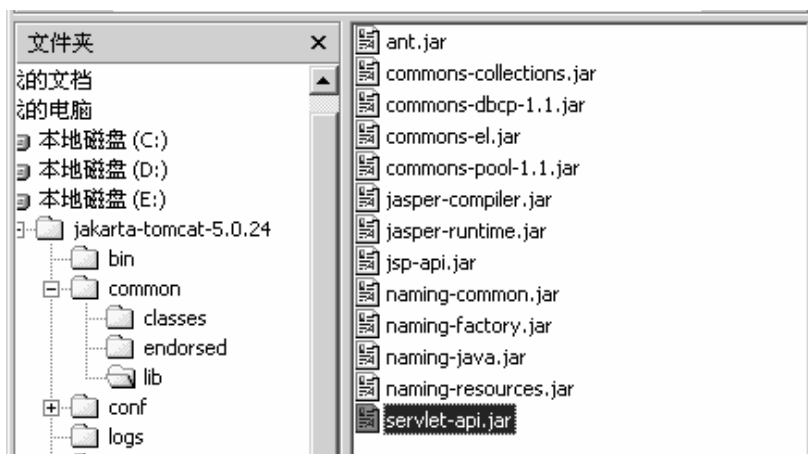


图 2-2 tomcat 中的 servlet 包

在命令行窗口运行：set CLASSPATH=%classpath%;tomcat 安装目录\common\lib\servlet-api.jar，可以将 ServletAPI 包放在 CLASSPATH 类路径中，如图：

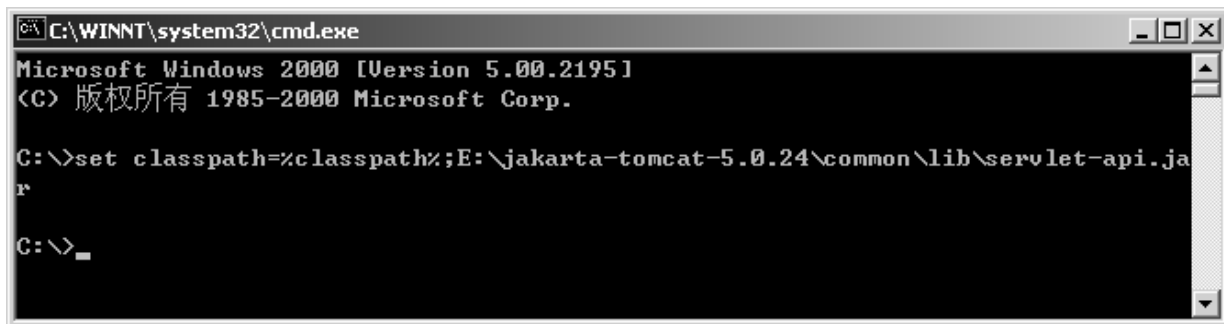


图 2-3 设置 CLASSPATH 环境变量

2) 编写 servlet 源文件, 该文件可以放在任何地方:

```
//MyHttpServlet.java
package test;

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

public class MyHttpServlet extends HttpServlet{
    protected void doGet(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException, java.io.IOException{
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("这是一个 Servlet,继承了 HttpServlet.");
        out.println("现在运行的是 doGet()方法.");
    }
    protected void doPost(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException, java.io.IOException{
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("这是一个 Servlet,继承了 HttpServlet.");
        out.println("现在运行的是 doPost()方法.");
    }
}
```

3) 编译 Servlet, 注意使用 javac -d 参数以生成包:

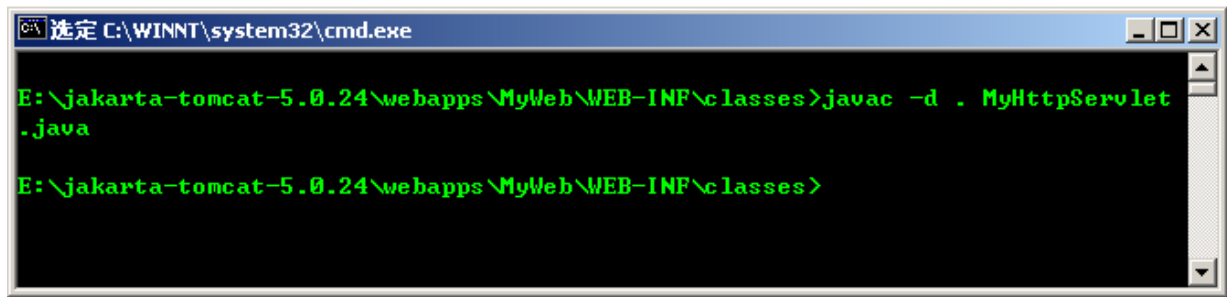


图 2-4 编译 servlet

- 3、将编译完的 Servlet 放到 站点\WEB-INF\classes 文件夹，注意要将编译生成的包一块拷到 classes 下
- 4、在 web.xml 中配置 Servlet，在 web.xml 中书写以下内容：

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <servlet>
    <servlet-name>MyHttpServlet</servlet-name>
    <servlet-class>test.MyHttpServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>MyHttpServlet</servlet-name>
    <url-pattern>/myhttpervlet</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.html</welcome-file>
  </welcome-file-list>

</web-app>
```

以上 web.xml 中，配置了 MyHttpServlet 的 url 访问形式为 /myhttpervlet，配置站点主页为 index.html。

- 5、创建站点主页

在 MyWeb 文件夹下，创建 index.html 文件，书写以下内容：

```
<h1>主页</h1>
```

```
<hr>  
<a href="myhttpServlet">myhttpServlet</a>  
<br>
```

6、启动 Tomcat

双击 TOMCAT_HOME\$bin\startup.bat 即可

7、访问 Servlet

打开浏览器，在地址栏中输入: <http://localhost:8080/MyWeb/>，会显示主页：

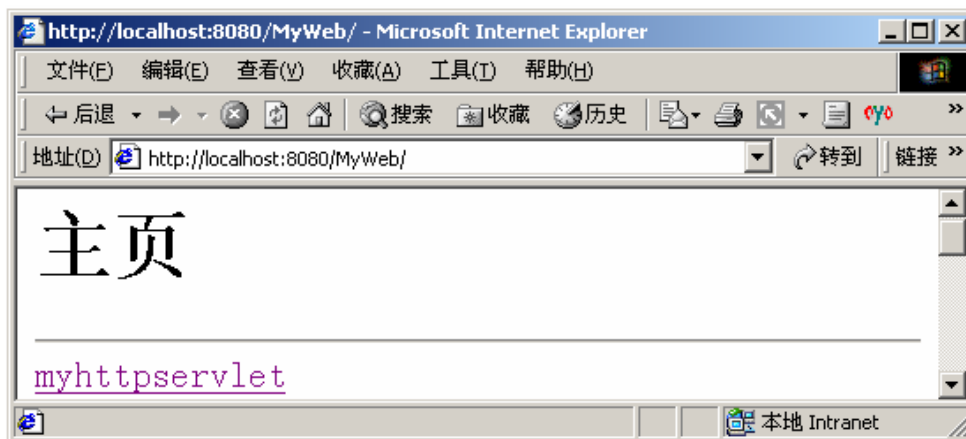


图 2-5 站点主页

点击超链接，显示：

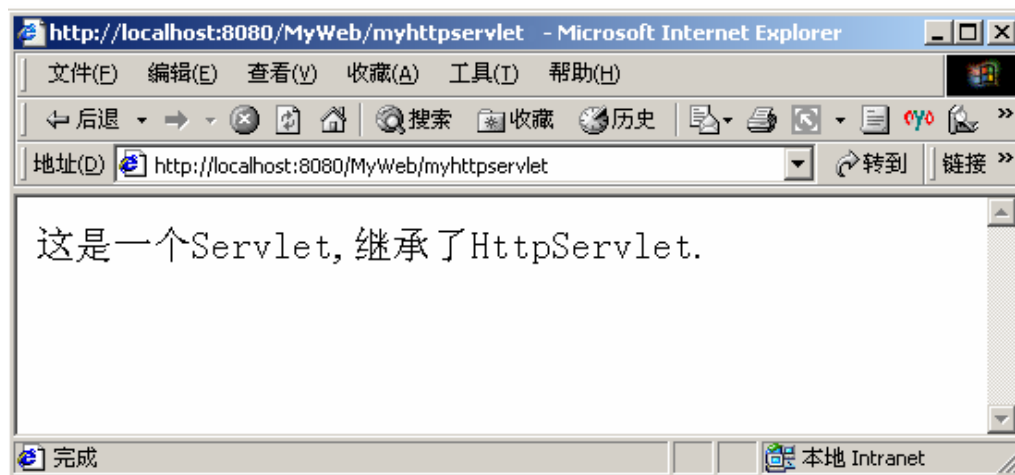


图 2-6 MyHttpServlet 显示结果

使用 MyEclipse 创建 Servlet

1、 利用第一阶段的方法配置好 Tomcat、MyEclipse 后，在 MyEclipse 中新建一个 java 项目，再新建一个 Servlet，Servlet 名为 MyHttpServlet，步骤参照第一阶段。

2、 Servlet 源代码如下：

```
//MyHttpServlet.java  
package test;
```

```

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

public class MyHttpServlet extends HttpServlet{

    protected void doGet(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException,
            java.io.IOException{
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("这是一个 Servlet,继承了 HttpServlet.");
    }

    protected void doPost(HttpServletRequest request,
                           HttpServletResponse response)
        throws ServletException,
            java.io.IOException{
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();
        out.println("这是一个 Servlet,继承了 HttpServlet.");
    }
}

```

- 3、 在 MyEclipse 中发布、运行 web 应用，步骤参照第一阶段。
- 4、 访问 Servlet，显示结果与图 2-5、图 2-6 相同。

2.2 实战实验

- 1、 打开 Eclipse 创建一个 web 工程，名为 MyWeb。
- 2、 在工程中创建一个 Servlet 类，放在包 demo 下，类名为 MyHttpServlet2
- 3、 参照图 1-18 中更改 Servlet 的映射名为 myhttpservlet2
- 4、 在 Servlet 中添加 init()、destroy()、doGet()方法，分别写上不同的输出语句,其中 init()、destroy()在 tomcat 控制台上输出，doGet()在客户浏览器中输出
- 5、 保存，部署 web 应用
- 6、 打开 IE 浏览器，键入 HTTP://localhost:8080/MyWeb/myhttpservlet2 访问 web 应用，并查看 tomcat 控制台的输出，再进行多次访问，以观察生命周期方法 init()、destroy()的运行方式，在此由于 httpservlet 的 service 方法已经实现，不必对其进行测试。

7、注意：要观察 destroy()的运行，可按以下步骤进行操作：

- 配置 Tomcat，添加 tomcat 管理员帐号

在 Tomcat 安装目录\conf\tomcat-users.xml 中，输入以下内容即可：

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
  <role rolename="tomcat"/>
  <role rolename="role1"/>
  <role rolename="manager"/>
  <role rolename="admin"/>
  <user username="tomcat" password="tomcat" roles="tomcat"/>
  <user username="both" password="tomcat" roles="tomcat,role1"/>
  <user username="role1" password="tomcat" roles="role1"/>
  <user username="admin" password="" roles="admin,manager"/>
</tomcat-users>
```

- 登录进 tomcat 管理员界面

通过 <http://localhost:8080> 进入tomcat默认站点主页：

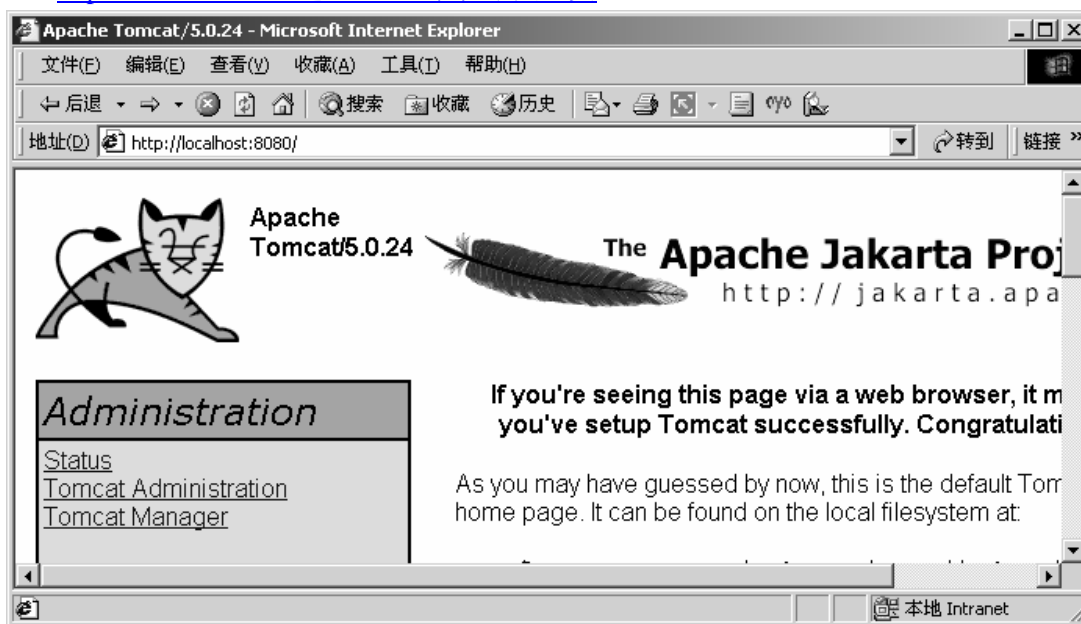


图 2-7 tomcat 默认站点主页

点击左侧的 Tomcat Manager 链接，登录时要求输入管理员密码，输出在 tomcat-users.xml 中配置的密码即可进入管理员页面：

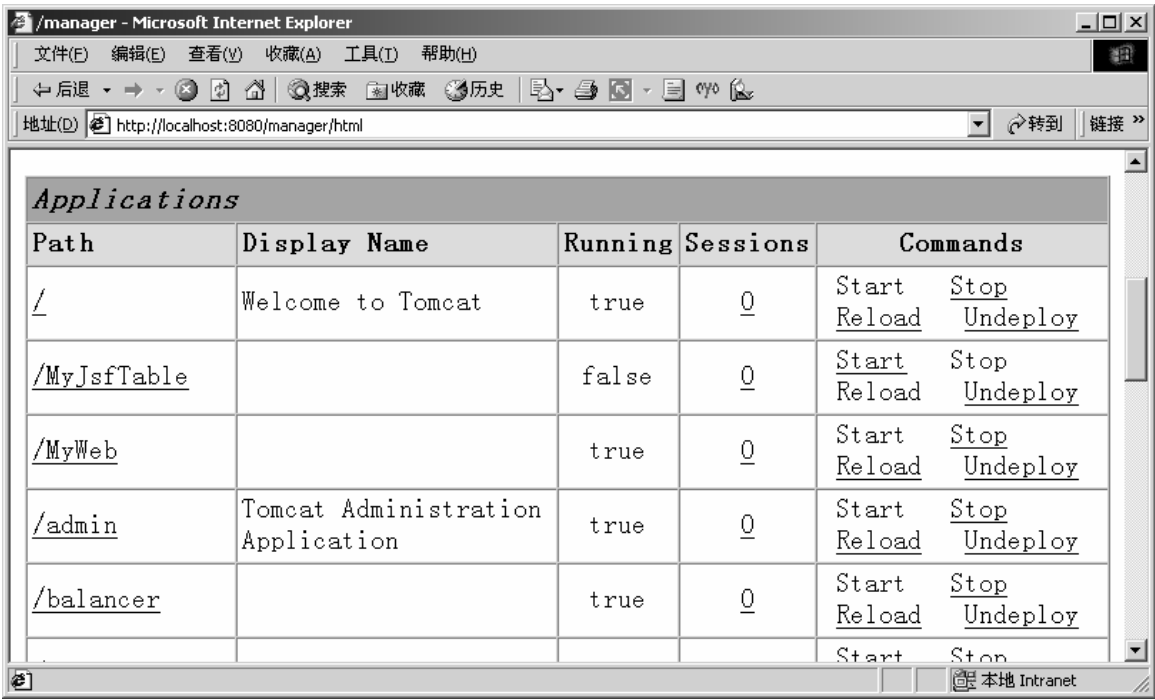


图 2-8 tomcat 管理员页

- 停止站点运行，此时 Servlet 的 destroy()方法会运行
- 在管理员页中可以显示当前 Tomcat 服务器中所有的 Web 应用，并且可以随时停止某个应用。点击应用对应的行中的“Stop”即可，再点击“Start”可以重启该应用。

2.3 实验后任务

- 1、 开发一个 HttpServlet，练习 doPost()方法的使用，要求实现如下功能：在页面中提供“用户注册”的表单，点击提交后，在 Servlet 中接收这些内容，并将注册信息显示给客户端用户。
表单如下所示：

用户注册	
用户：	<input type="text" value="注册"/>
密码：	<input type="text"/>
电子信箱：	<input type="text" value="124"/>
昵称：	<input type="text" value="1414"/>
<input type="button" value="进入"/> <input type="button" value="重填"/>	

图 2-9 用户注册表单

提示:

- 接收客户端表单中的数据，可按以下语句进行:

```
protected void doPost(HttpServletRequest request,
                        HttpServletResponse response)
    throws ServletException, java.io.IOException{
    //接收表单中名为 username 的元素的值
    String nick = request.getParameter("username");
}
```

- 另外 Servlet 在接收英文时没问题，但在接收中文数据时默认情况下会出现乱码，这个问题不要求解决，在第三章内容中会专门讲解。

第三章 Servlet 环境

学习目标

- 理解并掌握 `HttpServletRequest`、`HttpServletResponse`
- 掌握 `web.xml` 文件配置
- 理解 `ServletContext`、`ServletConfig`
- 理解 `RequestDispatcher`

3.1 模拟实验

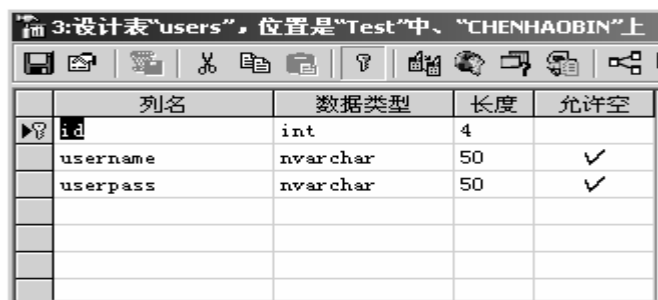
用户验证模块

本部分使用 Servlet 对登录用户的信息进行验证，整体流程如下：

- 1) 提交登录表单
- 2) Servlet 接收登录信息
- 3) Servlet 委托 java bean 判断数据库中是否存在此人
- 4) java bean 连接数据库进行判断，并将结果返回给 Servlet
- 5) Servlet 根据 java bean 返回的结果向客户端发送登录成功/失败的信息

具体步骤如下：

- 1、在 sqlserver 中创建数据库 Test，并创建 users 表



列名	数据类型	长度	允许空
id	int	4	
username	nvarchar	50	✓
userpass	nvarchar	50	✓

图 3-1 users 表结构

- 2、在 users 表中输入测试数据：



id	username	userpass
1	a	a
2	b	b
3	c	c
4	d	d

图 3-2 users 表示例数据

- 3、为 Test 数据库创建 DSN 数据源 dbDSN（在此使用 JDBC 第一类驱动）

- 4、在 Eclipse 中创建 WEB 工程 webprj_l3，步骤参见第一阶段练习；

- 5、向工程中添加 index.html、login.html，两文件内容如下：

```
//index.html
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>index.html</title>
    <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
    <meta http-equiv="description" content="this is my page">
  </head>
</html>
```



```

<meta http-equiv="content-type" content="text/html; charset=UTF-8">

<!--<link rel="stylesheet" type="text/css" href="./styles.css">-->
</head>
<body>
  <a href=login.html">用户登录</a><br>
</body>
</html>

```

```

//login.html
<HTML><HEAD><TITLE>登录</TITLE>
<META content="text/html; charset=gb2312" http-equiv=Content-Type>
<BODY bgColor=#ffffff>

<FORM action="loginServlet" method="post">
<P>&nbsp;  </P>
<TABLE align=center border=2 width="49%">
  <TBODY>
    <TR align=middle bgColor=#6633cc>
      <TD align=middle colSpan=2>
        <H4><FONT color=white
          face="Verdana, Arial, Helvetica, sans-serif">登录
        </FONT></H4></TD></TR>
    <TR bgColor=#ffffcc>
      <TD align=middle width="43%">
        <DIV align=right><FONT
          face="Verdana, Arial, Helvetica, sans-serif">用户: </FONT></DIV></TD>
      <TD width="57%">
        <DIV align=left><FONT face="Verdana, Arial, Helvetica, sans-serif">
          <INPUT name=username> </FONT></DIV></TD></TR>
    <TR bgColor=#ccff99>
      <TD align=middle width="43%">
        <DIV align=right><FONT
          face="Verdana, Arial, Helvetica, sans-serif">密码: </FONT></DIV></TD>
      <TD width="57%">
        <DIV align=left><FONT face="Verdana, Arial, Helvetica, sans-serif"><INPUT
          name=password type=password> </FONT></DIV></TD></TR>
  </TBODY></TABLE>

```

```
<P align=center>
<INPUT name=Submit2 type=submit value=进入>
<INPUT name=Reset type=reset value=重填></P>
</FORM></BODY></HTML>
```

6、添加一个普通类 ConnectionBean，实现数据库底层连接的操作：

```
//ConnectionBean.java
package demo;

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

public class ConnectionBean implements java.io.Serializable {
    Connection conn = null;
    private Statement stmt = null;
    private ResultSet rs = null;

    public ConnectionBean() {
        Properties prop = new Properties();
        try{
            InputStream file_in =
                getClass().getResourceAsStream("db.properties");
            prop.load(file_in);
            if(file_in != null) file_in.close();
        }catch(Exception e){
            System.out.println("打开设置文件发生错误!" + e.getMessage());
            e.printStackTrace();
        }

        String DsnName = prop.getProperty("SysDSN","dbdsn");
        String UserName = prop.getProperty("UserName","");
        String PassWord = prop.getProperty("PassWord","");
        System.out.println(DsnName);
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            Conn = DriverManager.getConnection("jdbc:odbc:" +
                                                DsnName, UserName, PassWord);
        }
```

```

    }catch(Exception e){
        System.out.println("打开数据库发生错误:"+e.getMessage());
    }
}
//打开数据库的连接
public Connection getConnection() {
    Properties prop = new Properties();
    try{
        InputStream file_in =
            getClass().getResourceAsStream("db.properties");
        prop.load(file_in);
        if(file_in != null) file_in.close();
    }catch(Exception e){
        System.out.println("打开设置文件发生错误!" + e.getMessage());
        e.printStackTrace();
    }
    String DsnName = prop.getProperty("SysDSN", "bookldsn");
    String UserName = prop.getProperty("UserName", "");
    String PassWord = prop.getProperty("PassWord", "");
    System.out.println(DsnName);
    try{
        Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
        Conn = DriverManager.getConnection("jdbc:odbc:" +
            DsnName, UserName, PassWord);
    }catch(Exception e){
        System.out.println("打开数据库发生错误:" + e.getMessage());
    }finally{
        return conn;
    }
}
//执行查询结果, 并返回记录集
public ResultSet excuteQuery(String SqlStr){
    try{
        stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY);
        rs = stmt.executeQuery(SqlStr);
    }
    catch(Exception e){

```

```
System.out.println("ConnectionBean 执行查询时发生错误:" +
                    e.getMessage());
    }
    finally{
        return rs;
    }
}
//执行数据库更新操作
public void excuteUpDate(String SqlStr) {
    try {
        stmt = conn.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                     ResultSet.CONCUR_READ_ONLY);
        stmt.executeUpdate(SqlStr);
    } catch (Exception e) {
        System.out.println("执行数据库更新操作发生错误:"+e.getMessage());
    }
}
public void close() throws SQLException {
    if(rs != null)
        rs.close();
    if(conn!=null)
        conn.close();
    if(stmt!=null)
        stmt.close();
}
protected void finalize() throws Throwable{
    this.close();
}
}
```

说明:

- 本类中封装了数据库底层连接的操作
- 构造方法中使用 `Properties` 类读取数据库配置信息文件 `db.properties`, 该文件要求放在与该文件的 `.class` 文件同目录下

7、添加一个 `UserLoginBean` 类, 将其变为 `javabean`, 实现与数据库打交道的操作:

```
//UserLoginBean.java
package demo;
```

```
import java.util.*;
import java.sql.*;
import java.io.*;

public class UserLoginBean extends ConnectionBean implements
java.io.Serializable {
    private String username = null;
    private String password = null;

    public UserLoginBean() {
    }

    public boolean isReguser(){
        boolean flag = false;

        String str_select = "select * from UserLogin where username = '" + username
+
            "' and password='" + password + "'";
        ResultSet rs = this.excuteQuery( str_select );
        System.out.println( str_select );
        try{
            if(rs != null && rs.next()) {
                flag = true;
                rs.close();
            }
        }catch(Exception e){
            System.out.println("UserLoginBean:"+e.getMessage());
        }finally{
            return flag;
        }
    }

    public void setUsername(String username){
        this.username = username;
    }
    public String getUsername(){
        return this.username;
    }
}
```

```
public void setPassword(String password){
    this.password = password;
}
public String getPassword(){
    return this.password;
}
}
```

8、添加一个 Servlet: LoginServlet, 用以接收登录信息并处理登录业务:

```
//LoginServlet.java
package demo;

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

public class LoginServlet extends HttpServlet {
    UserLoginBean userLoginBean = null;

    public void init(ServletConfig config)throws ServletException{
        userLoginBean = new UserLoginBean();
    }

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

    public void doPost(HttpServletRequest request,HttpServletResponse response)
        throws ServletException, IOException{
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();

        userLoginBean.setUsername( getStr(request.getParameter("username")) );
        userLoginBean.setPassword(getStr(request.getParameter("password")));

        if(userLoginBean.isReguser())
            out.println("登录成功!!!");
        else
            out.println("登录失败!!!");
    }
}
```

```

    }
    private String getStr(String str){
        try{
            byte[] temp = str.getBytes("ISO8859_1");
            String values2 = new String(temp);
            return values2;
        }catch(Exception e){
            e.printStackTrace();
        }
        return "null";
    }
}

```

说明:

- 本类中的 `getStr()`方法用以将接收到的表单参数转换成中文，也可以在接收数据之前调用语句 `request.setCharacterEncoding("gbk")`来实现自动转换。
- servlet 中访问数据库的部分委托给了 `UserLoginBean`, 实现业务与模型的分离, `UserLoginBean` 充当模型层。

9、在 Eclipse 中部署该 WEB 应用，步骤参见第一阶段练习

10、启动 Tomcat 服务器

11、访问主页：http://localhost:8080/webprj_l3

12、用用户名 a 、密码 a 进行登录，结果如下：

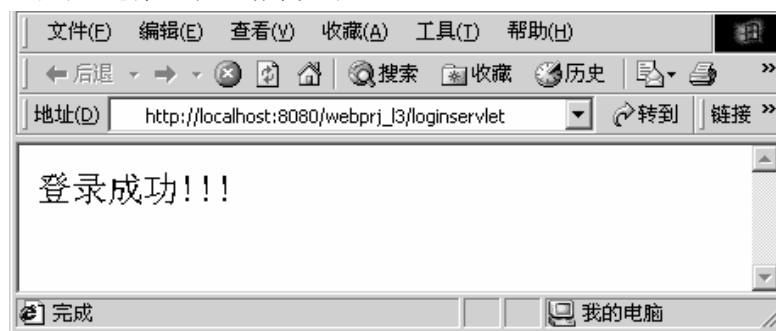


图 3-3 登录成功

页面跳转

1、完善 1.1 中开发的 `LoginServlet`, 通常在 `Servlet` 只处理业务逻辑, 不负责输出响应, 一般是 `Servlet` 根据 `javabean` 访问数据库后返回的结果, 选择不同的页面送到客户端, 这样, 显示与逻辑就分离开了, 程序更具扩展性, 也好维护。

2、修改 1.1 中的 `LoginServlet`:

```

//LoginServlet.java
.....

```

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException{
    response.setContentType("text/html;charset=gb2312");
    PrintWriter out = response.getWriter();
    userLoginBean.setUsername( getStr(request.getParameter("username")) );
    userLoginBean.setPassword(getStr(request.getParameter("password")));
    if(userLoginBean.isReguser())
        response.sendRedirect("loginsuccess.html");
    else
        response.sendRedirect("loginfail.html");
}
.....
```

3、 在工程中添加 loginsuccess.html、 loginfail.html:

```
//loginsuccess.html
<html>
<head>
<title>loginsuccess.html</title>
<meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
<meta http-equiv="description" content="this is my page">
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
<!--<link rel="stylesheet" type="text/css" href="./styles.css">-->
</head>
<body>
    登录成功!!!br>
</body>
</html>
```

```
//loginfail.html
<html>
<head>
<title>loginsuccess.html</title>
<meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
<meta http-equiv="description" content="this is my page">
<meta http-equiv="content-type" content="text/html; charset=UTF-8">
<!--<link rel="stylesheet" type="text/css" href="./styles.css">-->
</head>
<body>
```



```
登录失败!!!br>  
</body>  
</html>
```

- 4、部署 WEB 站点，访问 login.html，效果与 1.1 相同，只是将 Servlet 中与视图有关的部分交给了 html 页面去显示。

3.2 实战实验

- 1、修改 1.2 中的应用，以演示 RequestDispatcher 的用法。
- 2、在工程中添加一个新的 Servlet: ActionServlet，该 Servlet 负责接收登录信息，然后进行以下处理：
 - 1) 判断用户名、密码是否为空，为空则重定向回 login.html；
 - 2) 判断密码是否够 8 位长度，不足 8 位则重定向回 login.html；
 - 3) 判断用户名中是否包含单引号，如果包含则向客户端输入 HTTP 错误编码 500，防止输入非法用户名；
 - 4) 以上信息判断正确，将请求转发给 LoginServlet 进行登录处理，也就是说 ActionServlet 负责表单数据完整性验证，LoginServlet 负责登录业务；
- 3、保存 ActionServlet，并重新访问 login.html，以验证效果。

3.3 实验后任务

- 1、开发一个简单的聊天室，运行效果如下：
 - 1) 登录聊天室：

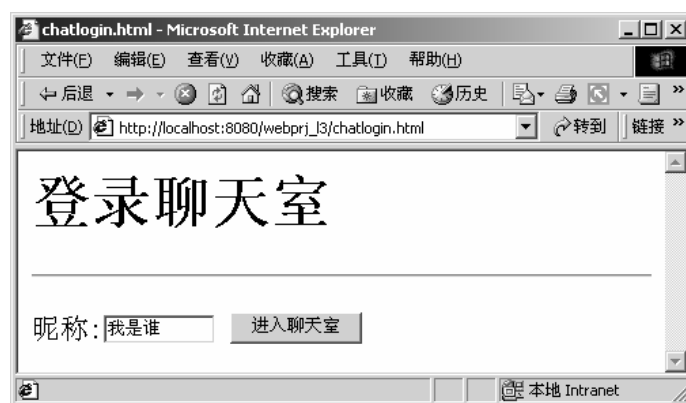


图 3-4 登录

- 2) 登录聊天室：

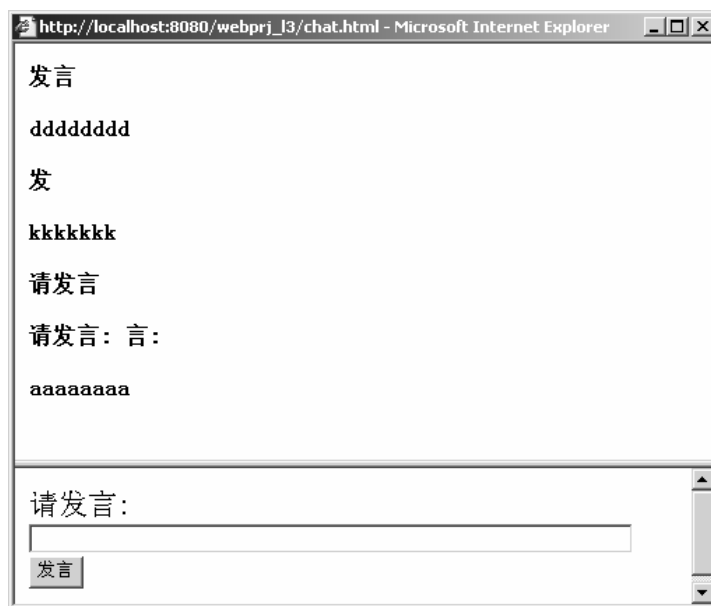


图 3-5 聊天界面

提示步骤:

- 编写登录表单页 chatlogin.html
- 表单提交给 ChatloginServlet, 在该 servlet 中判断昵称是否为空, 如果为空则转回登录页, 否则转向聊天框架页 chat.html, 该页中显示如图 3-5 的框架;
- 在 chat.html 中框架上方显示 DisplayServlet, 该 Servlet 定时刷新显示聊天内容, 聊天内容来自于 ServletContext 中存储的一个聊天内容集合;
- 在 chat.html 中框架下方显示发言页 talk.html;
- 在发言页中点击“发言”按钮时, 将发言传递给 TalkServlet, 该 Servlet 负责接收发言并将发言存储到集合中, 再将集合存储到 ServletContext 中;
- 在*.html 中为保证能正常显示中文, 在开头应加入:
<meta http-equiv="content-type" content="text/html; charset=gb2312">

第四章 会话管理

学习目标

- 理解并掌握 session 的用法
- 理解并掌握 cookie 的用法

4.1 模拟实验

产品列表

本部分使用综合使用 Servlet、javabean 从数据库提取产品信息，然后显示在客户端浏览器中，为后续购物车做准备。

1、创建数据库表 product，并创建 ODBC 数据源 dbdsn:


	列名	数据类型	长度	允许空
	id	int	4	
	productName	nvarchar	50	✓
	productPrice	money	8	✓
	productNum	int	4	✓

图 4-1

在表中填写以下数据:

	id	productName	productPrice	productNum
	1	asp	20	500
	2	vc++开发	100	500
	3	java	30	400
	4	j2ee	40	500
				

图 4-2

2、在 Eclipse 中创建 web 工程 webprj_14，然后在工程中创建 index.html:

```
//index.html
<html>
  <head>
    <title>index.html</title>
    <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
    <meta http-equiv="description" content="this is my page">
    <meta http-equiv="content-type" content="text/html; charset=gb2312">

    <!--<link rel="stylesheet" type="text/css" href="./styles.css">-->
  </head>

  <body>
    <a href="getproductsservlet">产品列表</a><br>
  </body>
</html>
```

3、开发用于操作数据库的 bean: DBConnection:

```
//DBConnection.java
package demo;

import java.sql.*;
import java.util.Properties;
import java.io.*;

//封装数据库底层操作(连接、执行查询、执行更新);
public class DBConnection {
    private String drivename;
    private String url;
    private String userid;
    private String userpass;
    private static Connection conn=null;
    private static Statement stmt=null;

    public DBConnection(){
        init();
    }

    //读取配置文件，将连接数据库的配置信息
    //读到数据成员;
    private void init(){
        try{
            Properties p=new Properties();
            //将文件内容装载到集合;

p.load( this.getClass().getResourceAsStream("db.properties") );
            //从p 中读取信息;
            drivename= p.getProperty("drivename","");
            url=p.getProperty("url","");
            userid=p.getProperty("userid","");
            userpass=p.getProperty("userpass","");
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

```
//连接数据库;
public void openConnection(){
    try{
        Class.forName( drivename );
        conn=DriverManager.getConnection(url,userid,userpass);
        stmt=conn.createStatement(
            ResultSet.TYPE_SCROLL_INSENSITIVE,
            ResultSet.CONCUR_READ_ONLY
        );
    }catch(Exception e){
        e.printStackTrace();
    }
}

//执行查询;
public ResultSet executeQuery(String sql) {
    ResultSet rs =null;
    try{
        if (conn == null)
            this.openConnection();
        rs = stmt.executeQuery(sql);
    }catch(Exception e){
        e.printStackTrace();
    }
    return rs;
}

//执行更新;
public void executeUpdate(String sql){
    try{
        if (conn == null) this.openConnection();
        stmt.executeUpdate(sql);
    }catch(Exception e){
        e.printStackTrace();
    }
}
```

```

//关闭资源;
public void close(){
    try{ stmt.close(); }catch(Exception e){}
    try{ conn.close(); }catch(Exception e){}
}
}

```

- 4、创建 db.properties 文件，放在上面编译出来的 DBConnecton.class 文件同级目录中（即 WEB-INF\classes\demo 目录下），该文件提供数据库配置信息：

```

# 注释
drivername=sun.jdbc.odbc.JdbcOdbcDriver
url=jdbc:odbc:dbdsn
userid=
userpass=

```

- 5、编写、编译实体类—Product，用于存储产品信息：

```

//Product.java
package demo;
import java.io.Serializable;
/**
 * 本类用于封装一个产品的信息，每个 Product 对象可以封装一条产品记录；
 */
public class Product extends DBConnection implements Serializable {
    private int id;
    private String productName;
    private double productPrice;
    private int productNum;

    public Product() {
    }

    public void setId(int id) {
        this.id = id;
    }

    public void setProductName(String productName) {
        this.productName = productName;
    }
}

```

```
public void setProductPrice(double productPrice) {
    this.productPrice = productPrice;
}

public void setProductNum(int productNum) {
    this.productNum = productNum;
}

public int getId() {
    return id;
}

public String getProductName() {
    return productName;
}

public double getProductPrice() {
    return productPrice;
}

public int getProductNum() {
    return productNum;
}
}
```

6、编写、编译 DAO 类：ProductDAO.java，用于提供对产品数据库的操作

```
//ProductDAO.java
package demo;

import java.util.*;
import java.sql.*;

/**
 * 本类用于封装与 product 有关的业务逻辑方法；
 */
public class ProductDAO extends DBConnection {

    //根据 id 得到指定商品；
```



```
public Product getProductById(String id){
    String sql="select * from product where id=" + id;
    Product p = new Product();
    try{
        ResultSet rs = this.executeQuery(sql);
        rs.next();
        p.setId(rs.getInt("id"));
        p.setProductName(rs.getString("productName"));
        p.setProductPrice(rs.getDouble("productPrice"));
        p.setProductNum(rs.getInt("productNum"));
        rs.close();
    }catch(Exception e){
        e.printStackTrace();
    }
    return p;
}

//得到所有商品;
public Collection getAllProducts(){
    ResultSet rs=executeQuery("select * from product");
    ArrayList list=new ArrayList();
    try{
        while (rs.next()) {
            Product p = new Product();
            p.setId(rs.getInt("id"));
            p.setProductName(rs.getString("productName"));
            p.setProductPrice(rs.getDouble("productPrice"));
            p.setProductNum(rs.getInt("productNum"));
            list.add(p);
        }
        rs.close();
    }catch(Exception e){
        e.printStackTrace();
    }
    return list;
}
}
```

7、编译 GetProductsServlet.java，用于提取并显示产品信息：

```
//GetProductsServlet.java
package demo;

import javax.servlet.*;
import javax.servlet.http.*;

import java.io.*;
import java.util.*;

/**
 * 到数据库中抽取所有商品信息，并存放于集合，然后显示到客户端；
 */
public class GetProductsServlet extends HttpServlet {
    /**
     * 到数据库中抽取所有商品信息，并存放于集合，然后显示到客户端
     * 访问产品表的业务逻辑由 ProductDAO 完成；
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {

        //1、实例化 ProductDAO 对象，以处理与产品有关的业务逻辑；
        ProductDAO pdao=new ProductDAO();

        //2、得到所有商品；
        Collection c=pdao.getAllProducts();

        //3、显示产品信息；
        Iterator it=c.iterator();
        Product p = null;

        response.setContentType("text/html;charset=gb2312");
        PrintWriter out=response.getWriter();

        out.println( "<table width=70% align=center border=1>");
        out.println("<tr><td colspan=4 align=center>产品列表</td></tr>");
        out.println("<tr>");
```

```
        out.println("<td>编号</td><td>品名</td><td>单价</td><td>数量</td>");
        out.println("</tr>");

        //循环显示产品信息;
        while (it.hasNext()) {
            p = (Product) it.next();
            out.println("<tr>");
            out.println("<td>" + p.getId() + "</td>");
            out.println("<td>" + p.getProductName() + "</td>");
            out.println("<td>" + p.getProductPrice() + "</td>");
            out.println("<td>" + p.getProductNum() + "</td>");
            out.println("</tr>");
        }

        out.println("</table>");
    }

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

8、部署、运行 web 应用

打开浏览器，在地址栏中输入: http://localhost:8080/webprj_14/，会显示主页：



图 4-3 站点主页

点击超链接，显示：

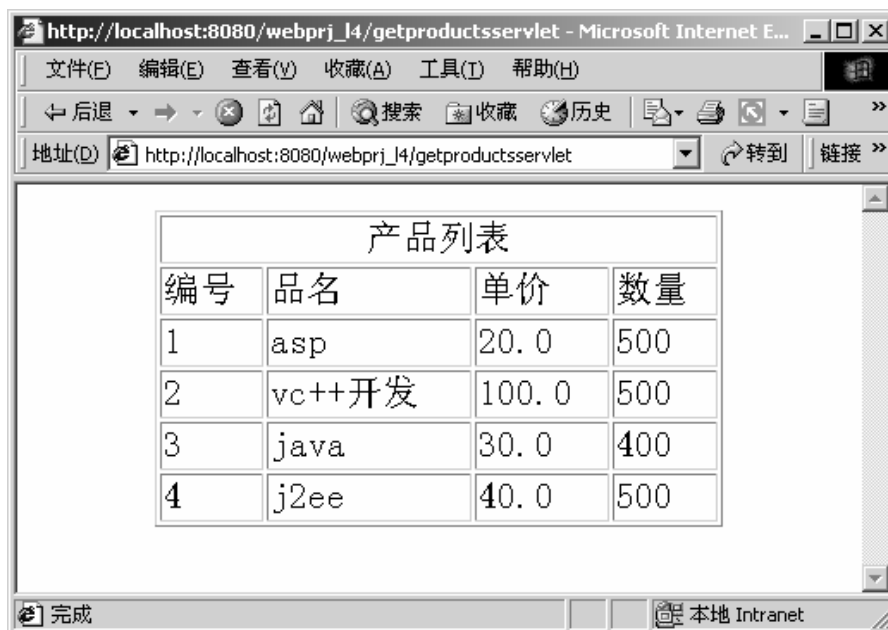


图 4-4 显示产品列表

说明：本示例主要演示在 Servlet 中如何使用 `javaBean` 去提取数据库中的信息，通常 Servlet 访问数据库时要调用相应的 DAO 类（也称数据访问对象），由该类完成对数据库的操作，而该类在完成操作时需要将得到的数据封装到 `Product` 实体类中，以供显示用。

购物车

此部分在 1.1 的基础上，逐步完成一个简单的购物车系统。

1、 修改 index.html：

```
//index.html
<html>
  <head>
    <title>index.html</title>
    <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
    <meta http-equiv="description" content="this is my page">
```

```
<meta http-equiv="content-type" content="text/html; charset=gb2312">

<!--<link rel="stylesheet" type="text/css" href="./styles.css">-->
</head>

<body>
  <h1>购物车系统登录</h1>
  <form action="loginServlet" method="post">
    用户名:<input size="10" name="username">
    <input type="submit" value="进入购物系统">
  </form>
</body>

</html>
```

2、添加 LoginServlet，负责接收登录用户信息：

```
//LoginServlet.java
package demo;

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

public class LoginServlet extends HttpServlet {

    /**
     * 处理登录
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }

    /**
     * 处理登录
     */
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
```

```
request.setCharacterEncoding("gbk");
String username=request.getParameter("username");
//如果没有填写用户名，则转回主页；
if(username.equals("")){
    response.sendRedirect("index.html");
    return;
}
//如果提供了用户名，则将用户名保存到 session;
HttpSession session=request.getSession(true);
session.setAttribute("username",username);

//将请求转发到产品列表 servlet;
RequestDispatcher dispatcher=
    this.getServletConfig().getServletContext().getRequestDispatcher
("/getproductsservlet");
dispatcher.forward(request,response);
}
}
```

3、修改 GetProductsServlet，添加“购买”超链接：

```
//GetProductsServlet.java
package demo;

import javax.servlet.*;
import javax.servlet.http.*;

import java.io.*;
import java.util.*;

/**
 * 到数据库中抽取所有商品信息，并存放到集合，然后显示到客户端；
 */
public class GetProductsServlet extends HttpServlet {

    /**
     * 到数据库中抽取所有商品信息，并存放到集合，然后显示到客户端
```

```

* 访问产品表的业务逻辑由 ProductDAO 完成;
*/
public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {

    //1、实例化 ProductDAO 对象, 以处理与产品有关的业务逻辑;
    ProductDAO pdao=new ProductDAO();

    //2、得到所有商品;
    Collection c=pdao.getAllProducts();

    //3、显示产品信息;
    Iterator it=c.iterator();
    Product p = null;

    response.setContentType("text/html;charset=gb2312");
    PrintWriter out=response.getWriter();

    //从 session 取出当前用户的名称
    HttpSession session=request.getSession(true);
    String username=(String)session.getAttribute("username");
    out.println("当前用户:" + username + "<br>");

    out.println(" <table width=70% align=center border=1>");
    out.println("<tr><td colspan=4 align=center>产品列表</td></tr>");
    out.println("<tr>");
    out.println("<td>编号</td><td>品名</td><td>单价</td><td>&nbsp;</td>");
    out.println("</tr>");

    //循环显示产品信息;
    while (it.hasNext()) {
        p = (Product) it.next();
        out.println("<tr>");
        out.println("<td>" + p.getId() + "</td>");
        out.println("<td>" + p.getProductName() + "</td>");
        out.println("<td>" + p.getProductPrice() + "</td>");
        out.println("<td><a href='cartervlet?id=" + p.getId()+ "'>购买");
    }
    out.println("</td>");
}

```

```
        out.println("</tr>");
    }

    out.println("</table>");
}

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

4、添加购物车类 Cart.java，负责存储购物信息：

```
//Cart.java
package demo;

import java.util.*;

public class Cart {
    private HashMap map;

    public Cart() {
        map=new HashMap();
    }

    //添加商品；
    public void addProduct(Product p){
        String key=""+p.getId();
        if( map.containsKey(key) ){//如果该编号的产品已经购买过；
            Product temp=(Product)map.get(key);//取出该产品的信息；
            temp.setProductNum( temp.getProductNum()+1 );//购买量增 1；
            map.put( ""+p.getId(),temp );//覆盖原产品信息；
        }else{
            map.put(key,p);//如果没有购买过该产品，则直接加入购物车中
        }
    }
}
```



```
}  
}  
  
//删除商品;  
public void delProduct(Product p){  
    map.remove( ""+p.getId());  
}  
  
//得到所有商品;  
public Collection getAllProduct(){  
    return map.values();  
}  
}
```

5、添加 CartServlet 接收选购信息，并将选购商品加入购物车：

```
//CartServlet.java  
package demo;  
  
import javax.servlet.*;  
import javax.servlet.http.*;  
  
import java.io.*;  
import java.util.*;  
  
/**  
 * 接收选购信息，并将选购商品加入购物车，然后显示购物车清单；  
 */  
public class CartServlet extends HttpServlet {  
  
    /**  
     * The doGet method of the servlet. <br>  
     */  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        doPost(request,response);  
    }  
  
    /**  
     * The doPost method of the servlet. <br>
```

```
    */
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        //1、接收选购的商品 ID;
        String id=request.getParameter("id");

        //2、得到购物车
        //先到 session 中取购物车，如果取不到，则生成新的购物车
        Cart cart=null;
        HttpSession session=request.getSession(true);
        Object obj=session.getAttribute("cart");
        if(obj!=null) cart=(Cart)obj;
        else cart=new Cart();

        //3、将选购商品加入购物车;
        ProductDAO pdao=new ProductDAO();
        //通过 DAO 类得到 ID 对应的产品信息;
        Product temp=pdao.getProductById( id );
        //将产品加入购物车;
        temp.setProductNum(1); //此时,Product 的 productnum 用于保存购买量;
        cart.addProduct(temp);

        //4、将购物车存入 session;
        request.getSession(true).setAttribute("cart",cart);

        //5、转到显示购物车的 Servlet;
        response.sendRedirect("showcartervlet");
    }
}
```

6、添加 ShowcartServlet，显示购物信息：

```
//ShowcartServlet.java
package demo;
```

```
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Iterator;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpSession;

public class ShowcartServlet extends HttpServlet {

    /**
     * The doGet method of the servlet. <br>
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }

    /**
     * 显示购物车信息;
     */
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        //从 session 取出购物车
        HttpSession session=request.getSession(true);
        Cart cart=(Cart)session.getAttribute("cart");

        //以下显示购物车信息
        String username=(String)session.getAttribute("username");
        out.println("当前用户:" + username + "<br>");
        out.println(" <table width=70% align=center border=1>");
        out.println("<tr><td colspan=5 align=center>购物车清单</td></tr>");
        out.println("<tr>");
```

```
        out.println("<td> 编 号 </td><td> 品 名 </td><td> 单 价 </td><td> 数 量  
</td><td>&nbsp;</td>");  
        out.println("</tr>");  
  
        //循环显示购物车信息;  
        Product p = null;  
        Iterator it=cart.getAllProduct().iterator();  
        while (it.hasNext()) {  
            p = (Product) it.next();  
            out.println("<tr>");  
            out.println("<td>" + p.getId() + "</td>");  
            out.println("<td>" + p.getProductName() + "</td>");  
            out.println("<td>" + p.getProductPrice() + "</td>");  
            out.println("<td>" + p.getProductNum() + "</td>");  
  
            out.println("</tr>");  
        }  
        out.println("</table>");  
  
        out.println("<br><center>");  
        out.println("<a href='getproductsservlet'>继续购物</a>");  
  
        out.println("<center>");  
    }  
}
```

7、编译、部署工程，运行 index.html:



图 4-5 登录购物系统

8、登录后，显示如下，每点击一次“购买”，则在购物车中增加一件该类产品：

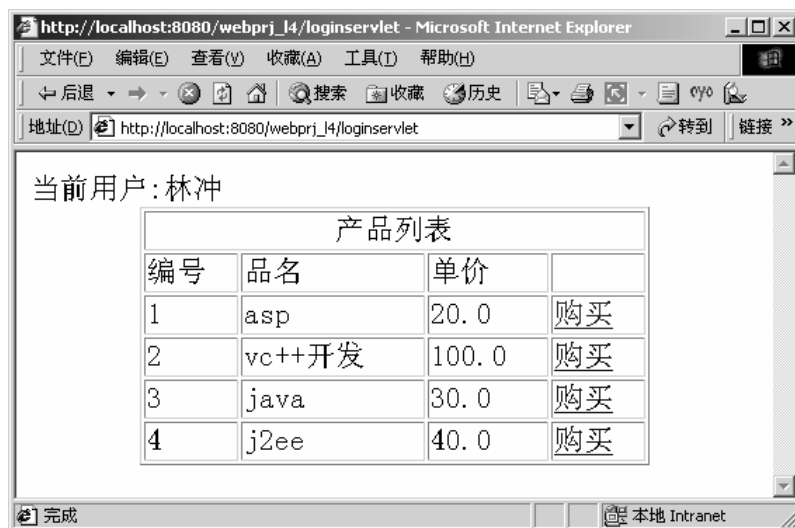


图 4-6 显示产品列表

9、选购商品后显示如下：

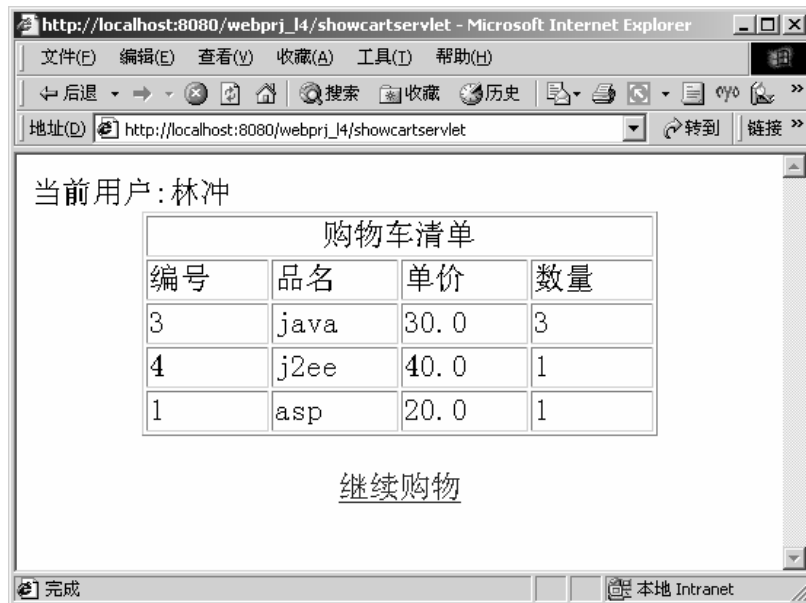


图 4-7 选购后显示购物车

10、点击“继续购物”回到图 4-6;

4.2 实战实验

继续完善购物车，在以上基础之上，添加到下功能：

- 1、添加“删除商品”的功能，可以删除指定商品
- 2、添加“清空购物车”功能
- 3、添加“结帐”功能，点击后显示购物信息及金额
- 4、参考界面如下：

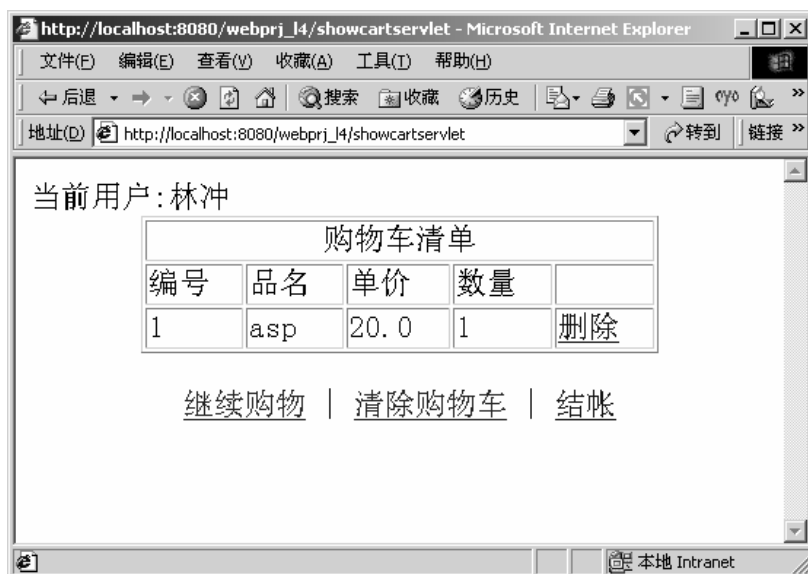


图 4-8 选择商品后显示购物车



图 4-9 删除某件商品

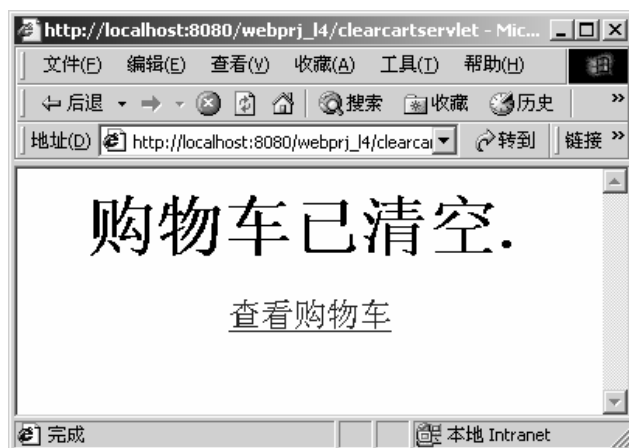


图 4-10 清空购物车



图 4-11 结帐界面

4.3 实验后任务

- 1、 某网上商城出售各类商品，现要求开发一个模块，该模块能够根据用户以往的购买习惯，在用户登录后显示用户以往最爱购买的商品列表，如果用户是首次登录，则显示表单询问用户最喜欢购买哪类商品，然后记录到客户端 Cookie 中，为以后了解客户购物习惯作准备。
 - 1) 用户首次登录：

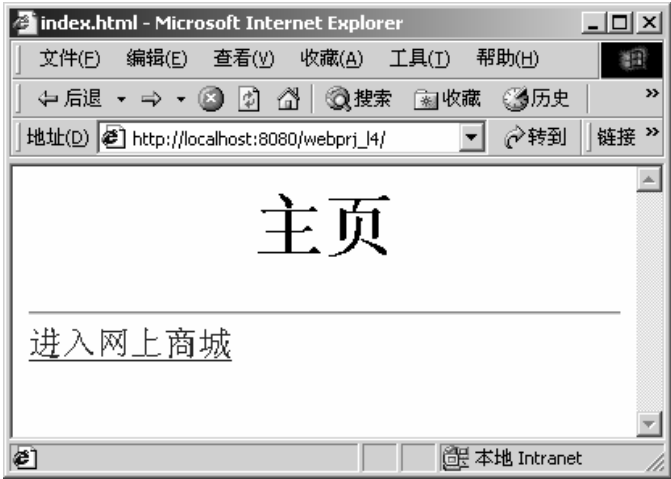


图 4-12 主页

- 2) 登录：

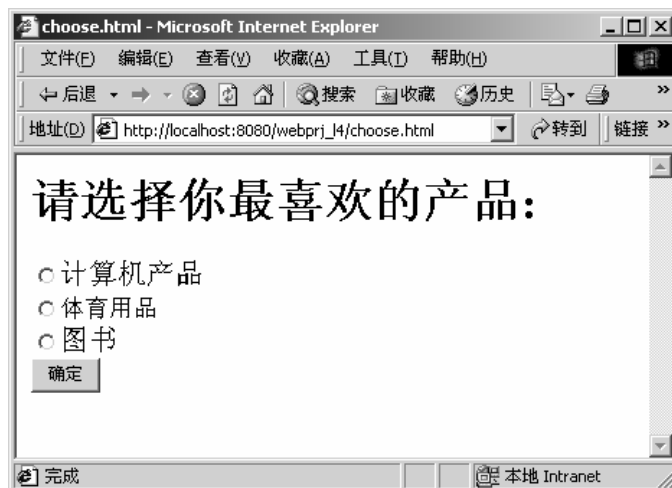


图 4-13 选择产品类型

3) 选择“体育用品”后，将用户选择写到 cookie，并显示：



图 4-14 产品信息

4) 再打开一个新浏览器，打开图 4-12 的主页，点击进入商城，则不需用户重新选择商品类型，而是直接显示图 4-14；

提示：

- ◆ 用户首次登录时，将客户选择的喜爱的商品类型保存到 cookie 中；
- ◆ 每次用户进入商城时，自动读取 cookie 值，判断有没有读到用户喜爱的商品类型，如果找到该 cookie 值则直接显示某类商品，否则显示图 4-11。

第五章 线程安全

学习目标

- 理解本地变量与实例变量
- 掌握同步化操作发布 web 应用

5.1 模拟实验

本地变量

本部分通过示例测试本地变量，与实例变量的访问方式作对比，以理解多线程 Servlet 中的实例变量安全问题。

- 1、 编写 TestServlet1, 该 Servlet 中声明了一个实例变量 count, 一个局部变量(即本地变量)localcount, 每次访问时这两个变量的值会递增 1, 多个用户访问该 Servlet 时, 实例变量是共享的, 本地变量每个线程有自己的本地副本, 源代码如下:

```
//TestServlet1.java
package demo;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class TestServlet1 extends HttpServlet {
    int count;
    /**
     * 测试本地变量，与实例变量作对比
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        int localcount = 0;
        response.setContentType("text/html;charset=gbk");
        PrintWriter out = response.getWriter();
        out.println("<HTML>");
        out.println("  <HEAD><TITLE>A Servlet</TITLE></HEAD>");
        out.println("  <BODY>");
        out.print("++localcount<br>");
        out.print("++count<br>");
        out.print("本地变量 localcount:" + (++localcount) + "<br>");
        out.print("实例变量 count:" + (++count) + "<br>");
        out.println("</BODY>");
        out.println("</HTML>");
        out.flush();
        out.close();
    }
}
```

```

    }
    /**
     * The doPost method of the servlet. <br>
     */
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
                       throws ServletException, IOException {
        doGet(request, response);
    }
}

```

2、打开两个浏览器访问该 Servlet:

打开第一个浏览器，访问 testervlet1，多刷新几次,如图 2-2 所示;

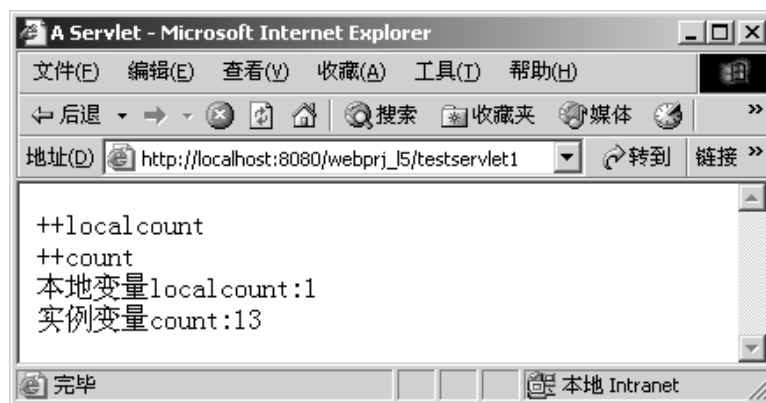


图 5-1 在第一个浏览器中访问 testervlet1

打开第二个浏览器，访问 testervlet1：如图：

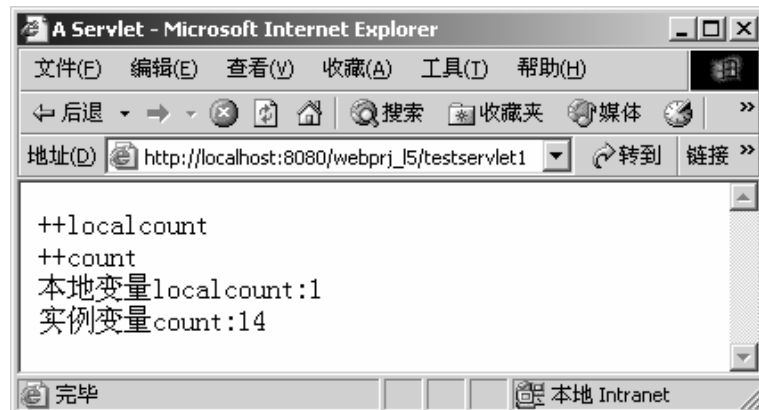


图 5-2 在第二个浏览器中访问 testervlet1

结果表明：实例变量 count 的值是在以往客户访问的基础上变化的，即实例变量是所有客户所共享的，在开发 Servlet 时要十分注意此类问题，以防程序出现逻辑问题。

同步

通过同步操作可以保证一个实例只能被一个线程访问，其他线程要访问必须等当前线程访问结束后才能取得对实例的访问权，这样可避免多个用户同时访问共享资源，下面示例演示如何在 Servlet 中使用同步：

本示例假设要统计某个论坛的访问人次，为此生成计数器存储在 `ServletContext` 全局作用域中，当有人登录论坛后就将计数值增 1，由于计数器在 `ServletContext` 全局作用域中，所以要防止多个用户同时修改计数值的情况。

1、论坛登录页 index.html:

```
//index.html
测试 Servlet 同步(统计某论坛访问人次):<br>
<form action="loginservlet" method="post">
用户名:<input name="username" size="10">
<input type="submit" value="进入论坛">
</form>
```

2、计数器 bean，用以保存计数值:

```
//CounterBean.java
package demo;
import java.io.Serializable;

public class CounterBean implements Serializable{
    private int count;
    public CounterBean(){
    }
    public void increament(){
        this.count++;
    }
    public int getCount(){
        return this.count;
    }
}
```

3、处理登录的 LoginServlet:

```
//LoginServlet.java
package demo;

import java.io.IOException;
```

```
import java.io.PrintWriter;
import javax.servlet.*;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * 假设该 servlet 负责处理论坛登录，在此 servlet 中要求统计论坛访问人次，
 * 该人次保存在 CounterBean 中，该 bean 保存在 ServletContext 中，
 * 这样当该 Servlet 在一较长时间段内无人访问而被销毁时，计数值还存在。
 */
public class LoginServlet extends HttpServlet {
    //该引用用以保存 ServletContext 对象；
    ServletContext application;

    /**
     * 在初始化时得到 ServletContext 对象
     */
    public void init(ServletConfig config) throws ServletException {
        this.application = config.getServletContext();
    }

    /**
     * The doGet method of the servlet. <br>
     */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }

    /**
     * The doPost method of the servlet. <br>
     */
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException, IOException {
        String username;
        response.setContentType("text/html;charset=gbk");
        PrintWriter out = response.getWriter();
        request.setCharacterEncoding("gbk");
        username=request.getParameter("username");
    }
}
```

```
//同步化 application, 即 ServletContext 对象;
//这样, 同一时刻只能有一个人访问 ServletContext 对象;
//可以避免同时多个人对计数器进行修改;
CounterBean bean=null;
synchronized(application){
    //到 ServletContext 中得到计数器 bean;
    Object obj=this.application.getAttribute("counterbean");
    if(obj==null){//如果尚不存在计数器对象, 则生成新计数器
        bean=new CounterBean();
    }else{//如果 servletcontext 中存在计数器, 则取出
        bean=(CounterBean)obj;
    }
    //计数值增 1
    bean.increament();
    //将计数完毕的计数器存回 servletcontext
    this.application.setAttribute("counterbean",bean);
}
out.println("<HTML>");
out.println("  <HEAD><TITLE>A Servlet</TITLE></HEAD>");
out.println("  <BODY>");
out.print("你好: " + username + ",欢迎光临!您是第" +
        bean.getCount() + "位访客!<br>");
out.println("  </BODY>");
out.println("</HTML>");
out.flush();
out.close();
}
}
```

4、 运行并访问 index.html:



图 5-3 登录



图 5-4 第一个用户登录后



图 5-5 第二个用户登录



图 5-6 第二个用户登录后

分析:

- 要将对象 `application` 进行同步，语法为 `synchronized(application){代码}`，大括号中间的代码也称为同步块，该块中的代码同一时刻只能被一个线程运行；
- 本例中如果没有使用同步块，在用户少的时候也发现不了问题，但可以假想：在用户访问量很大时，碰巧有两个用户同一时刻访问计数器，要将计数器增 1，假设当前计数器值为 10，运行流程如下图所示：

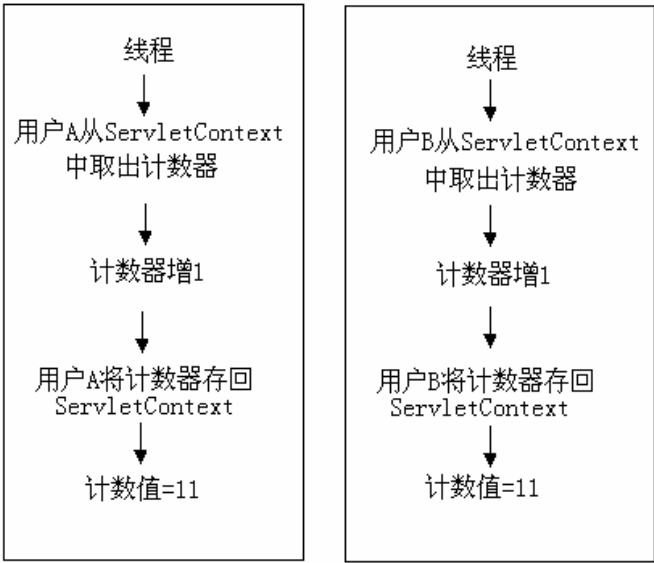


图 5-7 不使用同步的运行流程

- 从上图可以看出，多个用户同时访问计数器，导致计数器的值只增了一次，在其他各种各样的 `Servlet` 处理的业务中，如果不使用同步，也会出现类似情况；

5.2 实战实验

完善第四阶段创建的购物车，将商品列表存储到集合中，然后将集合存储到 `ServletContext` 中，这样第二个以后的客户购物时，就不必到数据库查询数据了，直接取出 `ServletContext` 全局作用域中的商品列表即可；

注意：第一个用户访问时要生成商品列表集合，并将集合存储到 `ServletContext` 中，此时应该使用同步，否则可能发生多个用户同时向 `ServletContext` 中存储数据的情况。

- 1、 在 Eclipse 中打开第四阶段创建的购物车工程；
- 2、 修改工程中的 `GetProductsServlet`，业务如下：
 - 1) 用户进入该 `servlet` 后，先在 `ServletContext` 中找产品列表集合，如果找到则不必查询数据库，直接显示该列表中的商品。
 - 2) 如果在 `ServletContext` 中未找到产品列表集合，则通过 `ProductDAO` 查询数据库得到产品集合，将集合存储到 `ServletContext` 中。
 - 3) 在将产品集合存储到 `ServletContext` 中时，要同步化 `application` 对象。
- 3、 运行购物车应用，进行测试，结果与第四阶段相同，请参见第四阶段实战实验结果。

5.3 实验后任务

- 1、 完善第三阶段作业——聊天室：
 - 1) 公共发言存储在 `ServletContext` 中，现要求将该部分实现同步化操作，防止多个聊天用户同时去修改聊天内容集合。
 - 2) 修改聊天框架页 `chat.html`，提供在线聊天用户列表的子框架，显示在线人名，可以在用户登录时，将用户昵称存入一个集合，将集合存储到 `ServletContext` 中，再开发一个 `Servlet` 负责显示在线用户。
 - 3) 修改用户列表集合的操作要实现同步。
 - 4) 运行效果如下：



图 5-8 登录聊天室



图 5-9 登录聊天室成功

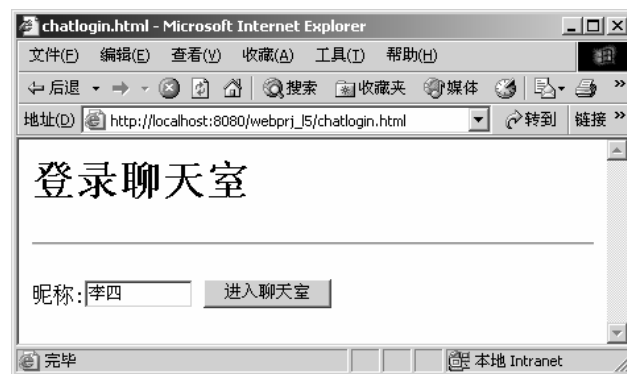


图 5-10 第二个用户登录聊天室

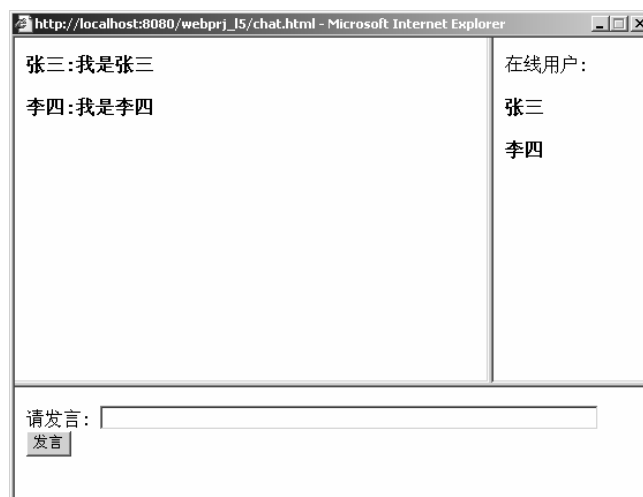


图 5-11 第二个用户登录聊天室后

提示:

显示用户列表的 Servlet 应该定时自动刷新, 可在 Servlet 中添加以下语句来实现:
`response.setHeader("refresh","2");` //添加协议头, 每 2 秒刷新一次;

第六章 JSP（一）

学习目标

- 使用 MyEclipse 创建 JSP
- 发布运行 web 应用
- 练习 JSP 声明、表达式、代码段的用法

6.1 模拟实验

使用 MyEclipse 创建 JSP

1、在 Eclipse 中创建一个 Web project:

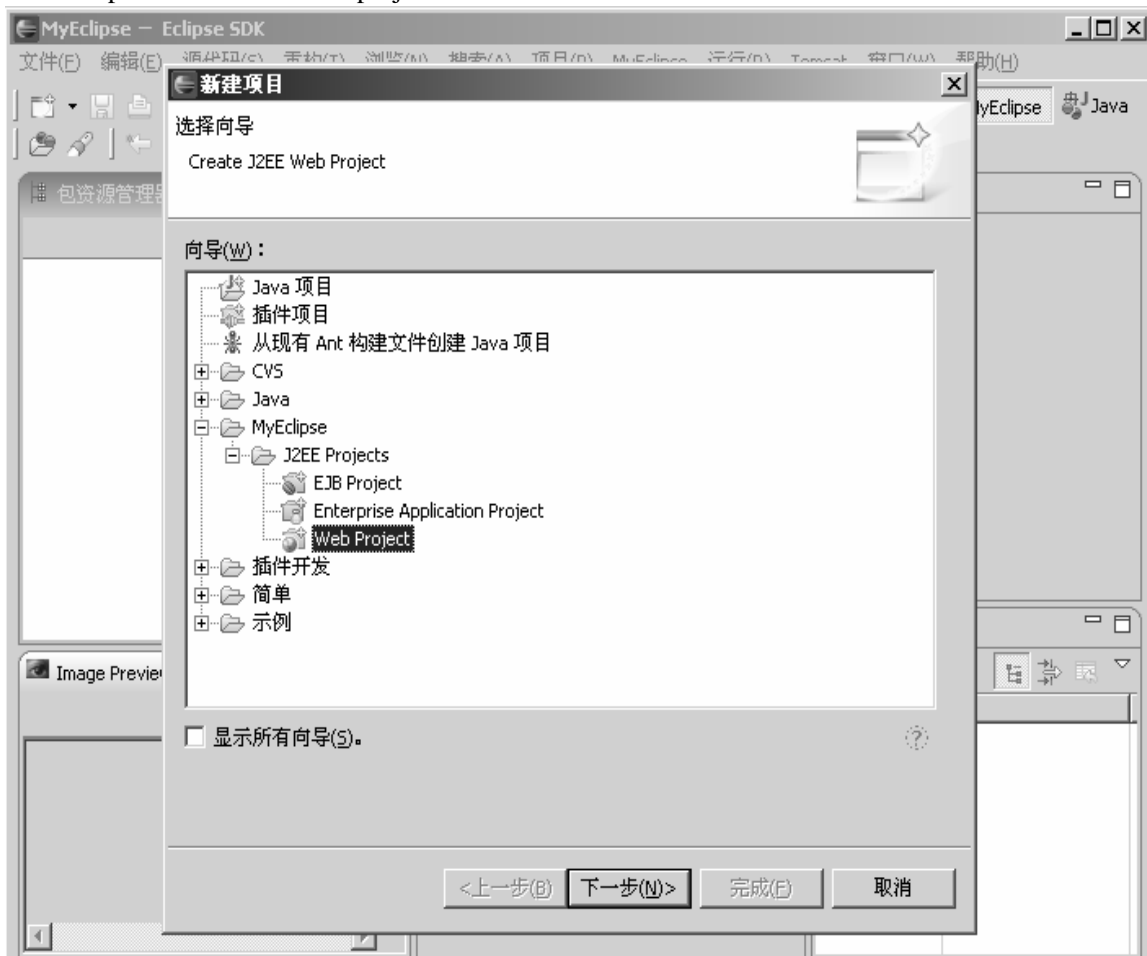


图 6-1 创建 web 工程

2、在向导中，将工程命名为 webprj_l6:



图 6-2 为 web 工程命名为 webprj_l6

3、创建 JSP 页面:



图 6-3 在工程中创建 JSP

4、将 JSP 改名为 index.jsp:



图 6-4 将 JSP 更名为 index.jsp

5、index.jsp 创建完毕，默认代码如下:

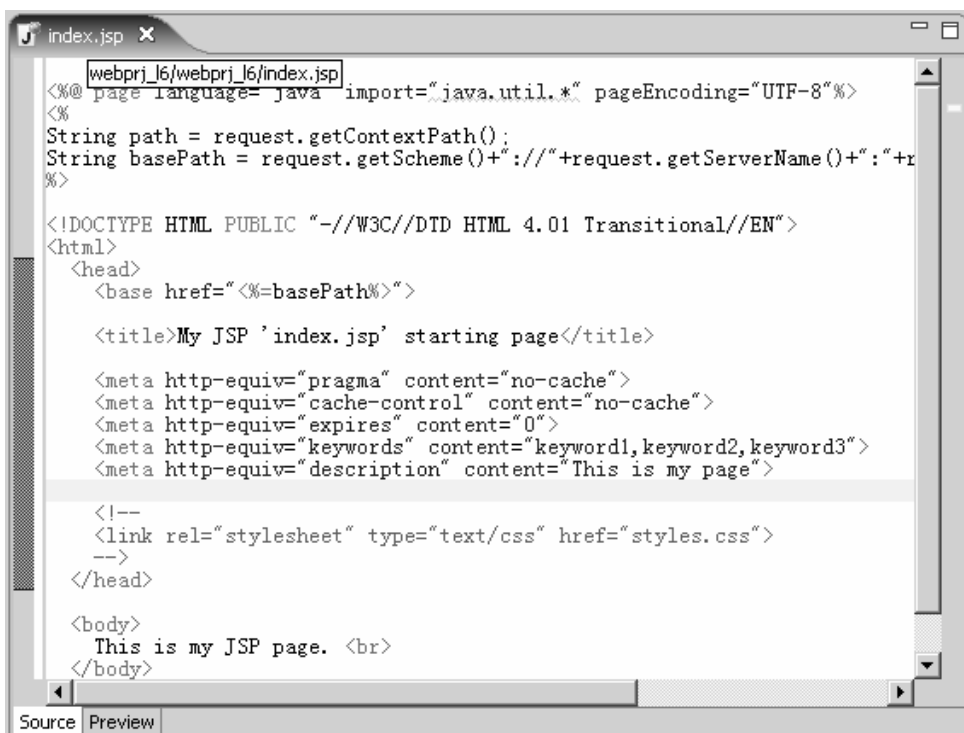


图 6-5 index.jsp 创建完毕

6、修改 index.jsp 内容，如下:

```
<%@ page contentType="text/html; charset=gb2312" %>
<html>
<head>
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
```



```

<meta http-equiv="expires" content="0">
<meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
<meta http-equiv="description" content="This is my page">
</head>
<body>
<center>
<h1>主页</h1>
</center>
</body>
</html>

```

说明: <%@ page contentType="text/html;charset=gb2312" %>用以设置 JSP 中字符是简体中文编码的, 否则中文显示为乱码。

7、启动 Tomcat (如果 Tomcat 尚未配置到 Eclipse 中来, 请参考第一阶段的配置步骤):

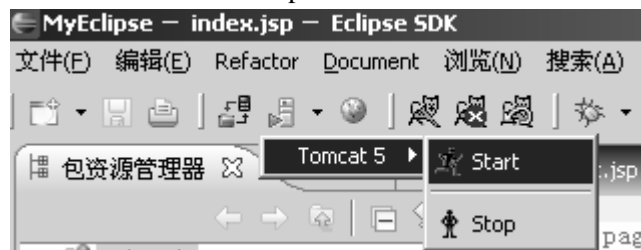


图 6-6 启动 Tomcat

8、部署站点到 tomcat 中:

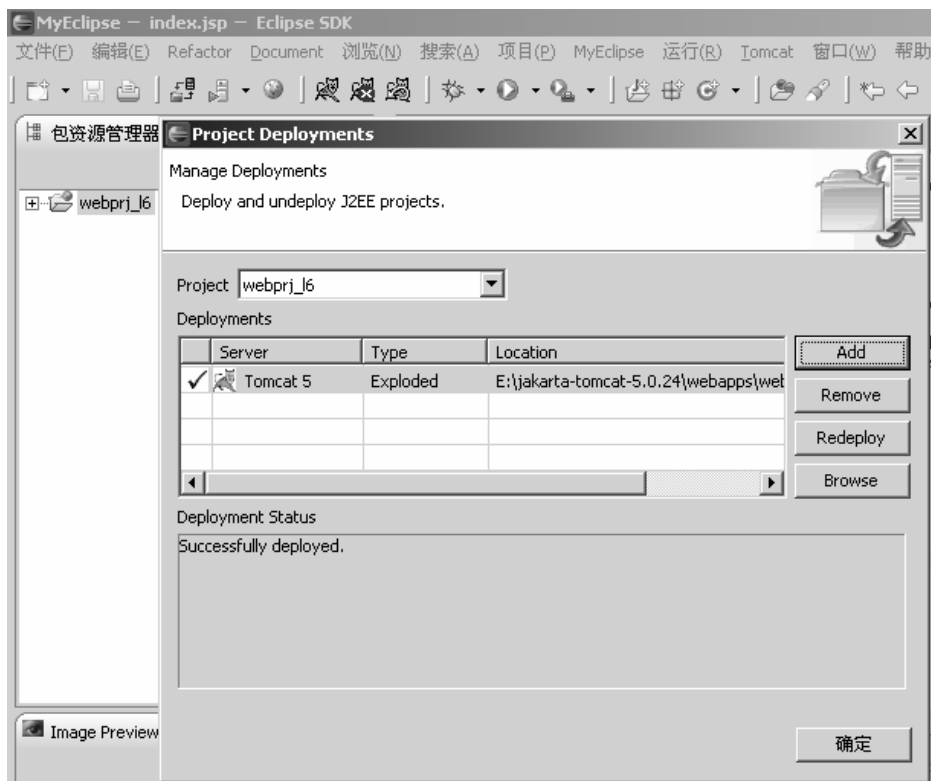


图 6-7 部署站点

9、访问 index.jsp:

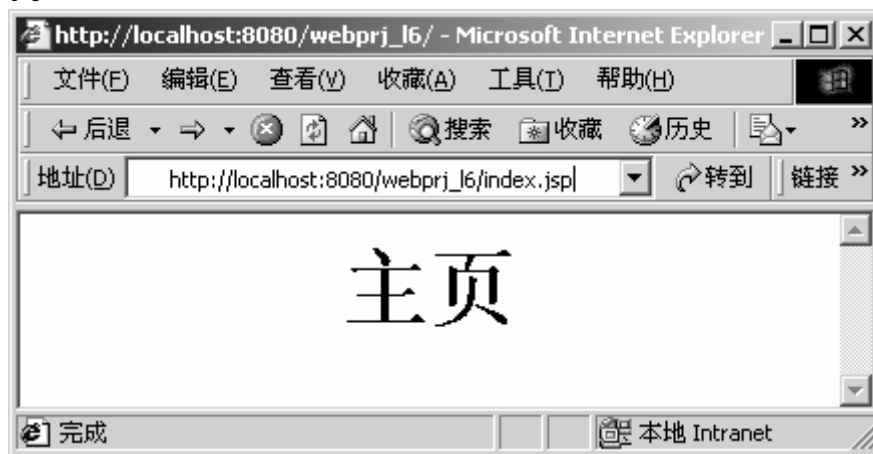


图 6-8 访问 index.jsp

主框架

本部分主要练习 include、page 指令的用法。

- 1、在 Eclipse 中创建 WebProject，名字为 webprj_l7。
- 2、编写 index.jsp，内容如下：

```
<%@ page contentType="text/html; charset=gb2312" %>
<html>
  <head>
    <style>
      <!--
      A:link { text-decoration:none; }
      A:visited { text-decoration:none;}
      A:active { text-decoration:none; color:FF0000}
      A{text-transform:none;text-decoration:none;}
      a:hover {text-decoration:underline}
      .s1{ font-size: 12px; word-spacing: 1; padding-top: 2px; padding-bottom:
-2px}
      .s2{ font-size: 12px; word-spacing: 1}
      -->
    </style>
  </head>
  <body bgcolor="#006ED2" class=s2>
    <table border="0" width="60%" align="center" cellspacing="0"
cellpadding="0" class="s1">
      <tr>
        <td width="100%">
```

```

        <font color="#FFFFFF" class=s2> ◇ <a href="index.jsp?mode=new"
style="color: #FFFFFF">新用户注册</a>
        ◇ <a style="color: #FFFFFF" href="index.jsp">匿名登录</a>
        ◇ <a href="index.jsp?mode=login" style="color: #FFFFFF">用户登录</a>
        ◇ <a href="index.jsp?mode=admin" style="color: #FFFFFF">管理员登录
</a>

        ◇</font>
    </td>
</tr>
</table>
<%
String mode=request.getParameter("mode");
%>
<br>

<table border="1" width="60%" height="220" cellspacing="0"
cellpadding="0" bordercolordark="#0084FB" bordercolorlight="#000080"
class="s2" align="center">
<tr>
    <td width="100%">
        <%if(mode==null || mode.equals("")){%>
            <%@ include file="noneuser.jsp" %>
        <%}%>
        <%if(mode!=null && mode.equals("new")){%>
            <%@ include file="addnew.jsp" %>
        <%}%>
        <%if(mode!=null && mode.equals("login")){%>
            <%@ include file="userlogin.jsp" %>
        <%}%>
        <%if(mode!=null && mode.equals("admin")){%>
            <%@ include file="adminlogin.jsp" %>
        <%}%>
    </td>
</tr>
</table>
</body>
</html>

```

分析: index.jsp 中显示了四个超链接: “新用户注册”、“匿名登录”、“用户登录”、“管理员登录”, 用户点击不同链接时, 会以 “?modi=xxx” 的方式将选择传递给 index.jsp, 然后 index.jsp 页面中显示相应表单, 显示不同表单时使用了 include 指令来实现。

3、 添加四个 JSP 页, 以提供以上 4 种表单:

1) 新用户注册 addnew.jsp:

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

<div align="center">
<center>
<table border="0" width="90%" cellpadding="0" cellspacing="0" class="s1">
<tr>
<td width="100%"><font color="#FFFFFF" class=s2>注册新用户: </font></td>
</tr>
</table>
</center>
</div>

<div align="center">
<center>
<table border="0" width="90%" cellpadding="0" cellspacing="0">
<tr>
<td width="100%">
<hr size="1" color="yellow" noshade>
</td>
</tr>
</table>
</center>
</div>

<form method="POST" action="index.jsp">
<div align="center">
<center>
<table border="0" width="80%" cellpadding="0" cellspacing="0" class="s1">
<tr>
<td width="100%"><font class=s2>用户名称: <input type="text" name="username"
size="9" maxlength="10"> 10 英文字符宽度! </font></td>
</tr>
<tr>
<td width="100%"><font class=s2>用户密码: <input type="password" name="userpwd"
```

[illegible]

2) 匿名登录 noneuser.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
<p align="center"><font size="5" face="Arial" color="yellow"><b>欢迎访本系统
</b></font></p>
<p align="center"><font size="4" color="#FFFFFF" class=s2>您可以匿名访问本站
</font></p>
<p align="center"><font size="4" color="#FFFFFF" class=s2>但只有注册用户才可使用
高级功能</font>
<form><center>
<p align="center">
</center></form>
```

3) 用户登录表单 userlogin.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
    <div align="center">
        <center>
            <table border="0" width="90%" cellpadding="0"
class="s1">
                <tr>
                    <td width="100%"><font color="#FFFFFF" class=s2> 用 户 登 录 :
</font></td>
                </tr>
            </table>
        </center>
    </div>
    <div align="center">
        <center>
            <table border="0" width="90%" cellpadding="0">
                <tr>
                    <td width="100%">
                        <hr size="1" color="#FFFFFF" noshade>
                    </td>
                </tr>
            </table>
        </center>
    </div>

    <form method="POST" action="userloginaction.jsp">
    <div align="center">
        <center>
            <table border="0" width="80%" cellpadding="0"
class="s1">
                <tr>
                    <td width="100%" align="center"><font class=s2>用户名称: <input
type="text" name="username" size="12" maxlength="10">&nbsp;</font></td>
                </tr>
                <tr>
                    <td width="100%" align="center"><font class=s2>用户密码: <input
type="password" name="userpass" size="12" maxlength="10">&nbsp;</font></td>
                </tr>
            </table>
        </center>
    </div>
    </form>
```

4) 管理员登录表单 adminlogin.jsp:

```

<div align="center">
    <center>
        <table border="0" width="80%" cellpadding="0" cellspacing="0"
class="s1">
            <tr>
                <td width="100%" align="center"><font class=s2>管理员用户: <input
type="text" name="username" size="12" maxlength="10">&nbsp;</font></td>
            </tr>
            <tr>
                <td width="100%" align="center"><font class=s2>管理员密码: <input
type="password" name="userpwd" size="12" maxlength="10">&nbsp;</font></td>
            </tr>
            <tr>
                <td width="100%" align="center" height="20"></td>
            </tr>
            <tr>
                <td width="100%" align="center">
                    <p align="center"><input type="submit" value="确认提交" name="B1"
class="s1">&nbsp;</p>
                    <input type="reset" value="全部重写" name="B2" class="s1"></td>
                </tr>
            </table>
        </center>
    </div>
</form>

```

4、部署站点，运行 index.jsp，点击不同超链接，index.jsp 页面中间会显示不同表单：



图 6-9 index.jsp

用户验证模块

本部分在第 1.1 的基础上继续完善“用户登录”功能，提供用户验证的逻辑，主要练习 request、response 对象的用法。

1、 在 1.2 工程中添加页面 userloginaction.jsp，在本页中接收登录信息，并进行验证：

```
<%@ page contentType="text/html; charset=gb2312" %>
<html>
  <body >
    <%
      request.setCharacterEncoding("gb2312");
      String username=request.getParameter("username");
      String userpass=request.getParameter("userpass");
      //如果用户名或密码为空，则转回登录页；
      if(username.equals("") || userpass.equals("")){
        response.sendRedirect("index.jsp?mode=login");
      }
      //判断用户名是否为"张三"，密码是否为"1234"
      if(username.equals("张三") && userpass.equals("1234")){
        out.println("登录成功<br>");
        out.println("用户名:" + username + "<br>");
        out.println("用户密码:" + userpass + "<br>");
        return;
      }else{
        out.println("登录失败，用户名或密码有误! <br>");
      }
    %>
    <a href="index.jsp?mode=login">重新登录</a>

  </body>
</html>
```

2、 运行并点击“用户登录”，运行结果如下：



图 6-10 用户登录



图 6-11 用户登录成功



图 6-12 用户登录失败（如果用户不是张三）

6.2 实战实验

继续完善 1.3 中的用户登录验证模块，要求验证逻辑放在 javabeen 中完成，JSP 中接收登录信息后交给 javabeen 完成验证的过程：

- 1、 在工程中创建一个类，名为 User，放在包 demo 下；
- 2、 在 User 类中提供 username、userpass 属性及 getter、setter 方法；
- 3、 在 User 类中提供判断用户是否存在的方法 isExist()，返回类型为 boolean，在方法内部判断如果用户名是“张三”，且密码是“1234”则认为用户存在；
- 4、 在 userloginaction.jsp 中使用 page 指令的 import 属性将 User 类引入到 JSP 页；
- 5、 在 userloginaction.jsp 中将接收到的用户登录信息传递给 User 对象进行判断，通过 User 对象的 isExist 方法得到验证结果；
- 6、 根据验证结果显示登录成功/失败的信息；

6.3 实验后任务

- 1、 开发购物车系统，为购物车系统提供：删除某件商品、清空购物车、结帐三部分功能，其中大部内容可以在第四阶段的基础上修改。

要求：与显示有关的部分要用 JSP 来实现（显示结帐清单、删除链接的显示），参考页面如下：



图 6-13 用户注册表单



图 6-14 用 JSP 显示结帐清单

第七章 JSP（二）

学习目标

- 使用 application
- 使用 session

7.1 模拟实验

商品列表

1、 创建数据库表 product，并创建 ODBC 数据源 dbdsn:

	列名	数据类型	长度	允许空
	id	int	4	
	productName	nvarchar	50	✓
	productPrice	money	8	✓
	productNum	int	4	✓

图 7-1

在表中填写以下数据:



	id	productName	productPrice	productNum
	1	asp	20	500
	2	vc++开发	100	500
	3	java	30	400
	4	j2ee	40	500
				

图 7-2

2、 在 eclipse 中创建 webproject: webprj_18, 然后创建主页 index.html:

```
.....
<body>
  <center>
    <h1>主页</h1>
  </center>
  <a href="listproducts.jsp">商品列表</a><br>
</body>
.....
```

3、 开发用于操作数据库的 DBConnection.java，实现数据库底层操作，代码参见第 6 阶段模拟实验。

4、 创建 db.properties 文件，创建在工程中的 src\demo 目录，这样在部署工程时该文件会自动部署到 DBConnecton.class 文件同级目录中（即 WEB-INF\classes\demo 目录下），该文件提供数据库配置信息：

```
# 注释
drivername=sun.jdbc.odbc.JdbcOdbcDriver
url=jdbc:odbc:dbdsn
userid=
userpass=
```

5、 编写、编译实体类—Product，用于存储产品信息。

6、 编写、编译 DAO 类: ProductDAO.java, 用于提供对产品数据库的操作。

7、 创建 listproducts.jsp, 用以显示商品列表:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ page language="java" import="java.util.*,demo.*"%>
<html>
  <head>
    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="cache-control" content="no-cache">
    <meta http-equiv="expires" content="0">
    <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
    <meta http-equiv="description" content="This is my page">
  </head>

  <body>
    <%
      Collection c = null;
      Object obj = application.getAttribute("productlist");

      //先到 application 中读取商品列表, 如果找不到再连接数据库查询商品;
      if(obj == null){
        //实例化 ProductDAO 对象, 以处理与产品有关的业务逻辑;
        ProductDAO pdao = new ProductDAO();
        //得到所有商品
        c = pdao.getAllProducts();
        application.setAttribute("productlist",c);
      }else{
        //如果在 application 中找到了商品列表
        c =(Collection)obj;
      }
      Iterator it = c.iterator();
      Product p = null;
    %>
    <table width=70% align = center border=1>
      <tr><td colspan=4 align=center>商品列表</td></tr>
      <tr><td>编号</td><td>品名</td><td>单价</td><td>购买</td></tr>
      <%
        //循环显示产品信息
        while (it.hasNext()) {
```

```

        p = (Product) it.next();
    %>
    <tr>
        <td><%=p.getId()%></td>
        <td><%=p.getProductName()%></td>
        <td><%=p.getProductPrice()%></td>
        <td>购买</td>
    </tr>
    <%
    }
    %>
</table>
</body>
</html>

```

说明：商品列表保存到了 `application` 对象中，以后其他用户或本用户再次访问商品列表时，就不用再次查询数据库了。

8、 部署工程，访问 `listproducts.jsp`：

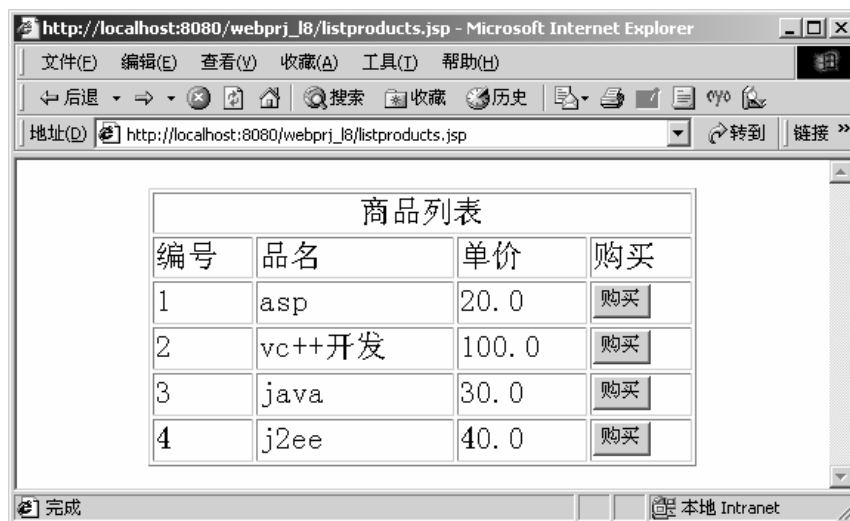


图 7-3 产品列表

购物车

此部分在 1.1 的基础上，逐步完成购物车系统。

1、 修改 `listproducts.jsp`，点击“购买”时转向 `buy.jsp`：

```

.....

<table width=70% align=center border=1>
    <tr><td colspan=4 align=center>商品列表</td></tr>
    <td>编号</td><td>品名</td><td>单价</td><td>购买</td></tr>
    <%

```



```

//循环显示产品信息
while (it.hasNext()) {
    p = (Product) it.next();
    %>
    <tr>
        <form action="buy.jsp" method="post" name="form1">
            <td><%=p.getId()%></td>
            <td><%=p.getProductName()%></td>
            <td><%=p.getProductPrice()%></td>
            <input type="hidden" name="id" value="<%=p.getId()%>">
            <input type="hidden" name="productname"
                value="<%=p.getProductName()%>">
            <input type="hidden" name="productprice"
                value="<%=p.getProductPrice()%>">
            <td>
                <input type="button" onclick="javascript:this.form.submit()"
                    value="购买">
            </td>
        </form>
    </tr>
    <%
    }
    %>
</table>
.....

```

2、 添加 buy.jsp，显示商品信息表单，让用户填写购买数量：

```

<%@ page contentType="text/html; charset=gb2312" %>
<%@ page language="java" import="java.util.*,demo.*"%>

<%
    request.setCharacterEncoding("gb2312");
    String id=request.getParameter("id");
    String productname=request.getParameter("productname");
    String productprice=request.getParameter("productprice");
    %>

<form action="cartServlet" method="post">
    <table width=50% align="center" border="1">

```

```

<tr>
    <td>品名:</td>
    <td><%=productname%></td>
    <input type="hidden" name="id" value="<%=id%>">
    <input type="hidden" name="productname" value="<%=productname%>">
</tr>
<tr>
    <td>单价:</td>
    <td><%=productprice%></td>
    <input type="hidden" name="productprice" value="<%=productprice%>">
</tr>
<tr>
    <td>数量:</td>
    <td><input type="text" name="num" size="10"></td>
</tr>
<tr>
    <td colspan="2" align="center">
        <input type="submit" value="确定购买">
    </td>
</tr>
</table>
</form>

```

3、 添加购物车类 Cart.java, 用以保存购买的商品, 该类代码参见第 4 阶段模拟实验 2。

4、 添加 CartServlet.java, 处理购物业务流程, 将选购物的商品加入购物车:

```

//CartServlet.java
package demo;

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

/**
 * 接收选购信息, 并将选购商品加入购物车, 然后显示购物车清单;
 */
public class CartServlet extends HttpServlet {
    /**

```

```

    * The doGet method of the servlet. <br>
    */
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        doPost(request, response);
    }

    /**
    * The doPost method of the servlet. <br>
    */
    public void doPost(HttpServletRequest request, HttpServletResponse
response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        //1、接收选购的商品信息;
        String id=request.getParameter("id");
        String productname=request.getParameter("productname");
        String productprice=request.getParameter("productprice");
        String num=request.getParameter("num");

        //2、得到购物车
        //先到 session 中取购物车, 如果取不到, 则生成新的购物车
        Cart cart=null;
        HttpSession session=request.getSession(true);
        Object obj=session.getAttribute("cart");
        if(obj!=null) cart=(Cart)obj;
        else cart=new Cart();

        //3、将选购商品加入购物车;
        Product temp=new Product();
        temp.setId( Integer.parseInt(id) );
        temp.setProductName(productname);
        temp.setProductPrice(Double.parseDouble(productprice));
        temp.setProductNum( Integer.parseInt(num) );//此时,Product 的 productnum
用于保存购买量;
        cart.addProduct(temp);

```

```
//4、将购物车存入 session;
request.getSession(true).setAttribute("cart",cart);

//5、转到显示购物车的 jsp;
response.sendRedirect("showcart.jsp");
}

}
```

5、 添加 showcart.jsp, 显示购物车内容:

```
<%@ page contentType="text/html;charset=gb2312" %>
<%@ page language="java" import="java.util.*,demo.*"%>
<html>
<head>
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
</head>

<body>
<%
//以下语句从 session 取出购物车, session 对象以后会讲到;
Cart cart=(Cart)session.getAttribute("cart");
%>
<table width=70% align=center border=1>
<tr><td colspan=5 align=center>购物车清单</td></tr>
<tr>
<td>编号</td><td>品名</td><td>单价</td><td>数量</td>
</tr>
<%
//循环显示购物车信息
Product p = null;
Iterator it=cart.getAllProduct().iterator();
while (it.hasNext()) {
p = (Product) it.next();
%>
<tr>
```

```

        <td><%= p.getId()%></td>
        <td><%= p.getProductName()%></td>
        <td><%= p.getProductPrice()%></td>
        <td><%= p.getProductNum()%></td>
    </tr>
<%
}
%>
</table>
<br><center>
<a href='listproducts.jsp'>继续购物</a>
<center>
</body>
</html>

```

6、 部署并运行，选购物一件商品后结果如下：



图 7-4 选购物一件商品后可以填写数量

7、 填写数量提交后，转到 showcart.jsp 显示购物车信息：

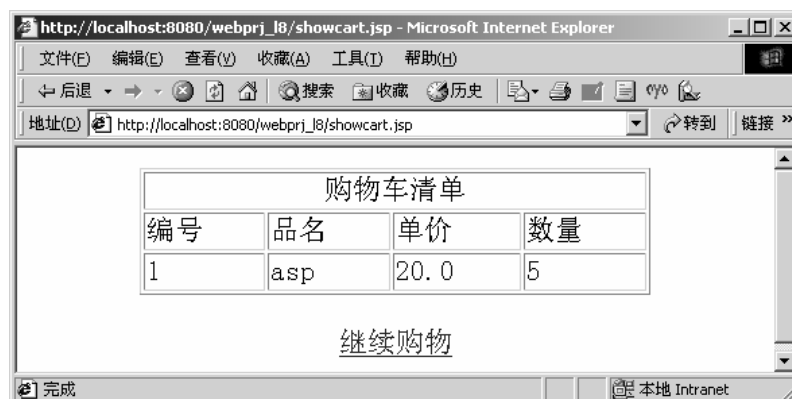


图 7-5 显示购物车信息

7.2 实战实验

为以上购物车系统添加：删除商品的功能。

- 1、 在 showcart.jsp 显示的购物车信息中添加一列“删除”，如图：



图 7-6 添加“删除”列

- 2、 点击删除后运行 clearcartervlet，并执行删除操作，然后显示如下界面：



图 7-7 删除某件商品

7.3 实验后任务

- 1、 完善本阶段中开发的购物车系统，为购物车系统提供：清空购物车、结帐功能，主要练习 session 对象的使用。

要求：参考页面如下：

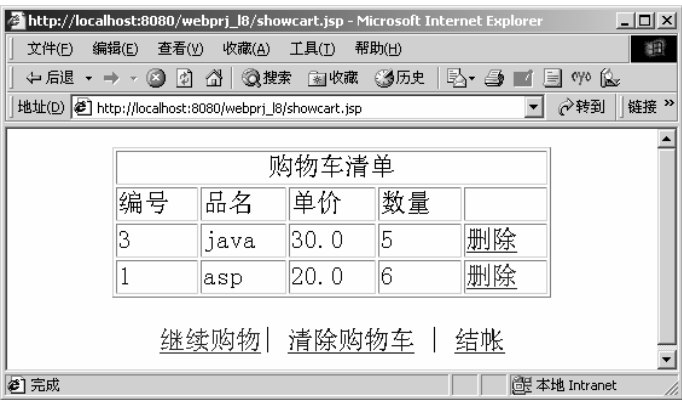


图 7-8 提供清除和结帐链接

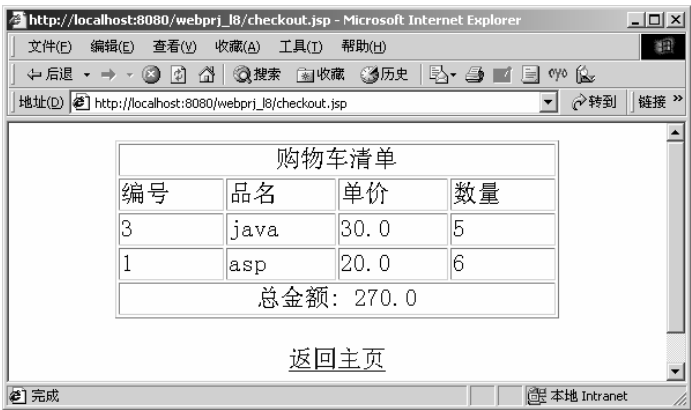


图 7-9 结帐清单

2、 使用 `pageContext` 对象完成第 6 阶段模拟实验 1——“主框架”的功能，运行结果如下：



图 7-10 主框架页面

提示：使用 `pageContext.include("url")` 可以包含指定页面。

第八章 JSP 动作

学习目标

- 练习使用 JSP 动作

8.1 模拟实验

商品列表

本实验在上一章“商品列表”的基础上，进行修改，将显示商品列表时使用的 javabean，改为用 jsp 动作的形式实现。

1、 在 Eclipse 中打开上一章的“商品列表”工程；

2、 修改工程中的 listproducts.jsp:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ page language="java" import="java.util.*,demo.*"%>
<jsp:useBean id="pdao" class="demo.ProductDAO"/>
<jsp:useBean id="p" class="demo.Product"/>
<html>
<head>
</head>
<body>
<%
    Collection c = null;
    Object obj = application.getAttribute("productlist");

    //先到 application 中读取商品列表，如果找不到再连接数据库查询商品；
    if(obj == null){
        c=pdao.getAllProducts();
        application.setAttribute("productlist",c);
    }else{
        //如果在 application 中找到了商品列表
        c = (Collection)obj;
    }
    Iterator it = c.iterator();
    // Product p = null;
%>
<table width=70% align=center border=1>
    <tr><td colspan=4 align=center>商品列表</td></tr>
    <tr><td>编号</td><td>品名</td><td>单价</td><td>购买</td></tr>
    <%
        //循环显示产品信息
        while (it.hasNext()) {
            Product temp = (Product) it.next();
            %>
            <jsp:setProperty name="p" property="id"
```

```

        value="<%=temp.getId()%>" />
        <jsp:setProperty name="p" property="productName"
            value="<%=temp.getProductName()%>" />
        <jsp:setProperty name="p" property="productPrice"
            value="<%=temp.getProductPrice()%>" />
    <tr>
        <td><jsp:getProperty name="p" property="id" /></td>
        <td><jsp:getProperty name="p" property="productName" /></td>
        <td><jsp:getProperty name="p" property="productPrice" /></td>
        <td>购买</td>
    </tr>
<%
}
%>
</table>
</body>
</html>

```

3、 部署工程，访问 listproducts.jsp:

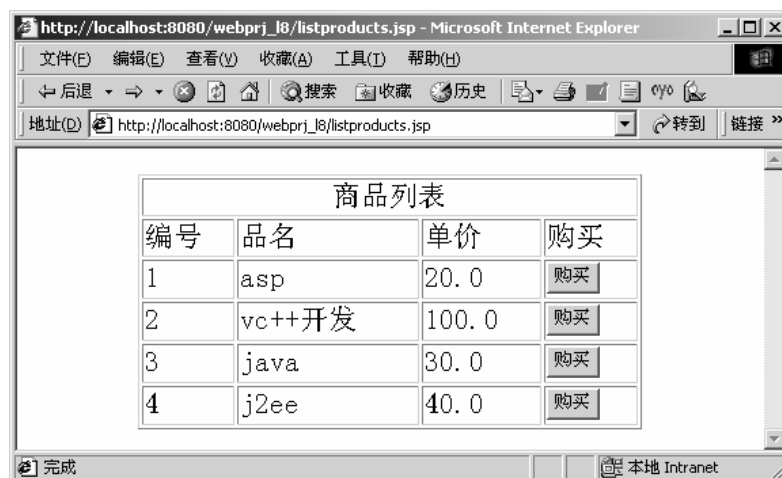


图 8-1 产品列表

购物车

此部分在上一章“购物车”的基础上，使用 JSP 动作完成购物车系统。

1、 修改 listproducts.jsp，点击“购买”时转向 buy.jsp:

```

.....

<table width=70% align=center border=1>
    <tr><td colspan=4 align=center>商品列表</td></tr>

```

```

        <td>编号</td><td>品名</td><td>单价</td><td>购买</td></tr>
    <%
    //循环显示产品信息
    while (it.hasNext()) {
        Product temp = (Product) it.next();
    %>
        <form action="buy.jsp" method="post" name="form1">
        <jsp:setProperty name="p" property="id"
            value="<%=temp.getId()%>" />
        <jsp:setProperty name="p" property="productName"
            value="<%=temp.getProductName()%>" />
        <jsp:setProperty name="p" property="productPrice"
            value="<%=temp.getProductPrice()%>" />
        <tr>
            <td><jsp:getProperty name="p" property="id" /></td>
            <td><jsp:getProperty name="p" property="productName" /></td>
            <td><jsp:getProperty name="p" property="productPrice" /></td>
            <td>
                <input type="hidden" name="id" value="<%=p.getId()%>">
                <input type="hidden" name="productname"
                    value="<%=p.getProductName()%>">
                <input type="hidden" name="productprice"
                    value="<%=p.getProductPrice()%>">
                <input type="button" onclick="javascript:this.form.submit()"
                    value="购买">
            </td>
        </form>
    </tr>
    <%
    }
    %>
</table>
.....

```

2、 修改 showcart.jsp, 显示购物车时使用 JSP 动作:

```

<%@ page contentType="text/html; charset=gb2312" %>
<%@ page language="java" import="java.util.*, demo.*" %>
<jsp:useBean id="p" class="demo.Product" />

```

```

<html>
<head>
  <meta http-equiv="pragma" content="no-cache">
  <meta http-equiv="cache-control" content="no-cache">
  <meta http-equiv="expires" content="0">
  <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">
</head>

<body>
<%
  //以下语句从 session 取出购物车, session 对象以后会讲到;
  Cart cart=(Cart)session.getAttribute("cart");
%>
<table width=70% align=center border=1>
<tr><td colspan=5 align=center>购物车清单</td></tr>
<tr>
  <td>编号</td><td>品名</td><td>单价</td><td>数量</td>
</tr>
<%
  //循环显示购物车信息
  Iterator it=cart.getAllProduct().iterator();
  while (it.hasNext()) {
    Product temp = (Product) it.next();
  %>
    <jsp:setProperty name="p" property="id" value="<%=temp.getId()%>"/>
    <jsp:setProperty          name="p"          property="productName"
value="<%=temp.getProductName()%>"/>
    <jsp:setProperty          name="p"          property="productPrice"
value="<%=temp.getProductPrice()%>"/>
    <jsp:setProperty          name="p"          property="productNum"
value="<%=temp.getProductNum()%>"/>
    <tr>
      <td><jsp:getProperty name="p" property="id"/></td>
      <td><jsp:getProperty name="p" property="productName"/></td>
      <td><jsp:getProperty name="p" property="productPrice"/></td>
      <td><jsp:getProperty name="p" property="productNum"/></td>
    </tr>
  
```

```
<%  
    }  
%>  
</table>  
  
<br><center>  
<a href='listproducts.jsp'>继续购物</a>  
<center>  
</body>  
</html>
```

3、 其他部分保持不变，选择商品后，转到 buy.jsp，运行效果如下：



图 8-2 选购物一件商品后可以填写数量

4、 填写数量提交后，转到 showcart.jsp 显示购物车信息：

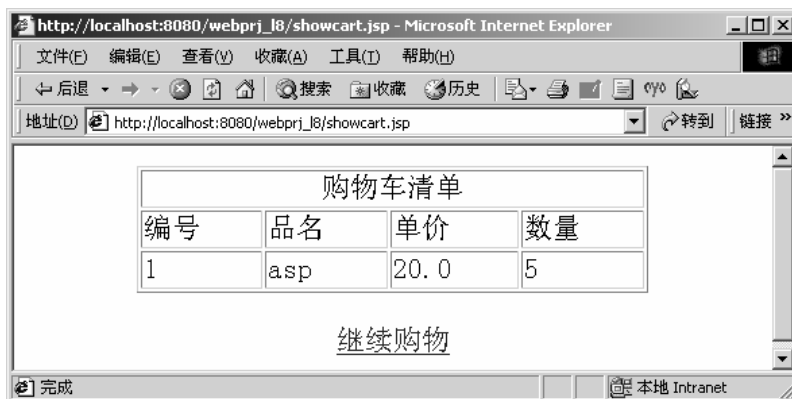


图 8-3 显示购物车信息

8.2 实战实验

编写一个简单的网上投票系统，要求使用 JSP 动作，参考步骤如下：

1、 编写 vote.jsp，显示投票界面及投票结果：

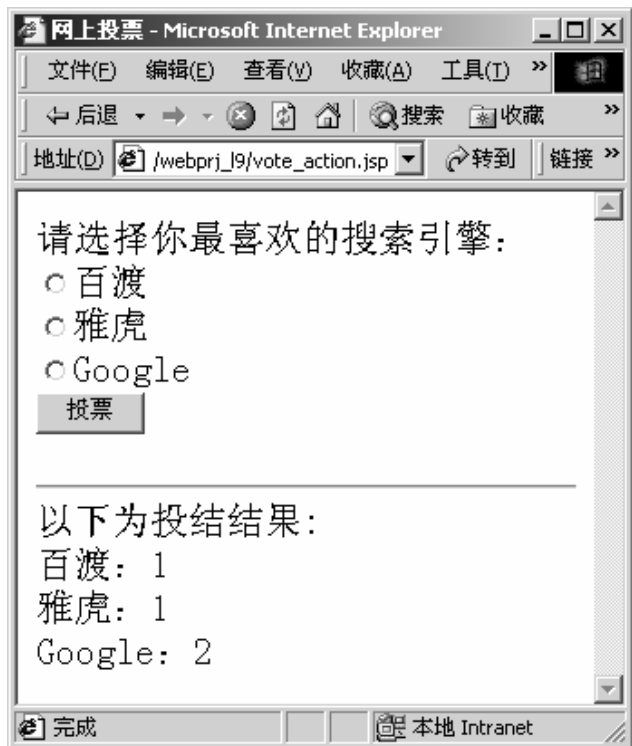


图 8-4 投票界面 vote.jsp

- 2、 选择并点击“投票”后，提交给另一个 jsp 页，该 jsp 页中通过 javabean 进行票数统计，对 javabean 的操作可以通过 JSP 动作完成，部分示例代码如下：

```
.....  
<jsp:useBean id="vote" class="demo.Vote" scope="application"/>  
.....  
    <!--以下语句设置投票 bean 的 site 属性，对应的票数会递增-->  
    <jsp:setProperty name="vote" property="site"/>  
.....  
    <!--转发到投票页-->  
    <jsp:forward page="vote.jsp"/>  
.....
```

- 3、 注意：通过 JSP 动作生成的 javabean 对象要存储在 application 域中，以便所有用户都能访问。

8.3 实验后任务

- 1、 开发一个简单的留言本系统，用户可以留言，提交后转到另一个 JSP 页，显示所有留言的列表。参考页面如下：



图 8-5 留言界面

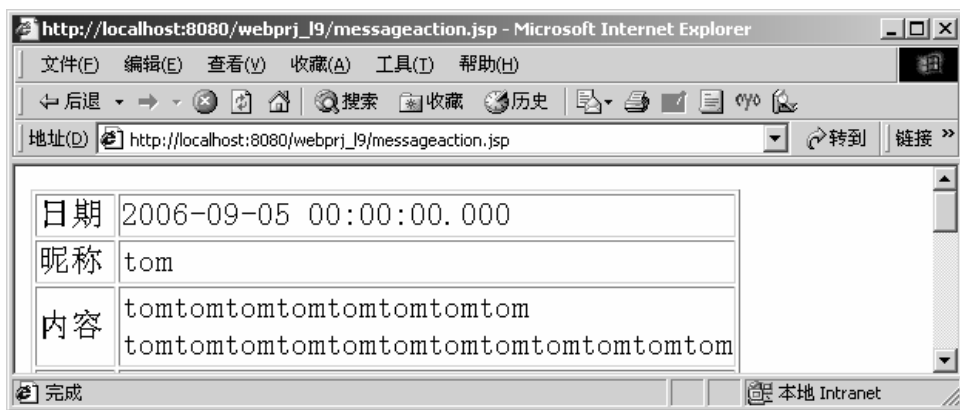


图 8-6 显示留言

提示：

可以使用<jsp:setProperty>动作自动接收表单参数值。

提交留言后转到其他 JSP 页可以使用动作<jsp:forward>来完成。

第九章 JSP2.0 特征

学习目标

- 使用 EL
- 使用 JSTL 核心标签

9.1 模拟实验

使用 EL 表达式语言

1、 使用 EL 中的算术运算符——el_1.jsp

```
<%@ page contentType="text/html; charset=gb2312" %>
算术及逻辑运算:<br>
<%
    request.setAttribute("username","tom");
%>
1.2 + 2.3==${1.2 + 2.3}<br>
3/4=${3/4}<br>
10%4=${10%4}<br>
10 mod 4=${10 mod 4}<br>
条件表达式:${(1==2) ? 3 : 4}<br>
判断用户名是否为空:${empty username}<br>
```

2、 使用 EL 中的比较及逻辑运算符——el_2.jsp

```
<%@ page contentType="text/html; charset=gb2312" %>
比较运算:<br>
1 < 2:${1 lt 2}<br>
4.0 >= 3:${4.0 ge 3}<br>
4 <= 3:${4 le 3}<br>
100.0 == 100:${100.0 == 100}<br>
100.0 eq 100:${100.0 eq 100}<br>
(10*10) != 100:${(10*10) != 100}<br>
(10*10) ne 100:${(10*10) ne 100}<br>
```

3、 使用 EL 读取作用域中的数据——el_3.jsp

```
<%@ page contentType="text/html; charset=gb2312" %>
<%
    request.setAttribute("nick","tom_request");
    pageContext.setAttribute("nick","tom_page");
%>
作用域范围中的数据:<br>
request 中的 nick : ${requestScope.nick} <br>
page 中的 nick : : ${pageScope.nick} <br>
```

使用 JSTL 核心标签

要使用 JSTL，需要在 Eclipse 工程中导入 JSTL 的支持类库，导入方法如下图所示，在菜单中选择“Add JSTL Libraries...”后会自动将 JSTL 的 jar 文件、tld 文件加载到工程的 WEB 模块中：

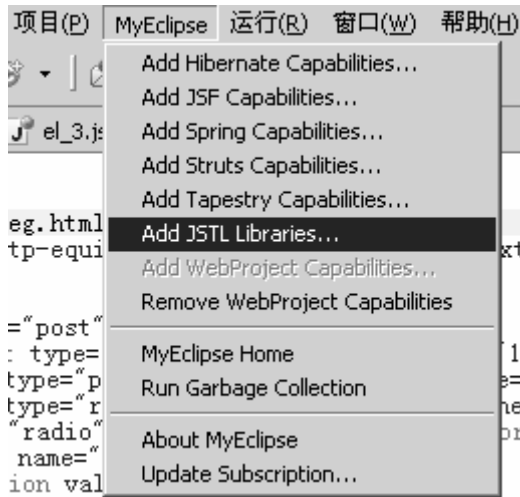


图 9-1 导入 JSTL 类库支持

1、 使用 JSTL 核心标签中的——变量定义及输出标签,在工程中添加 jstl_1.jsp 代码如下:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

输出自定义变量:
<c:set var="n" value="100"/>
<c:out value="{n}"/>
<br>

<c:set scope="page" var="num1" value="200"/>

开始输出定义的变量: <br>
pageScope.num1:<c:out value="{pageScope.num1}" default="No" /><br>

<br>删除 page 中的 num1:<br>
<c:remove var="num1" scope="page" />

pageScope.num1:<c:out value="{pageScope.num1}" default="No" /><br><br>
```

2、 使用流程控制标签,在工程中添加 jstl_2.jsp 代码如下:

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

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<%
    request.setAttribute("username","tom_request");
%>
if 测试:<br>
<c:if test="${requestScope.username == 'tom_request'}" var="result"
scope="page">
用户名是:tom_request!.
</c:if>
</br>
result:${result}<br>
<br><br>
choose、when、otherwise 测试:<br>
<c:set var="n" value="70"/>
<c:choose>
    <c:when test="${n==60}">
        n 等于 60
    </c:when>
    <c:when test="${n==70}">
        n 等于 70
    </c:when>
    <c:otherwise>
        n 不等于 60、70.
    </c:otherwise>
</c:choose>
```

3、 使用循环标签,在工程中添加 jstl_2.jsp 代码如下:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

forEach 测试 1:<br>
<c:forEach begin="0" end="10" var="i" step="2">
count=<c:out value="${i}"/><br>
</c:forEach>

<br><br>
forEach 测试 2:<br>
```

```
<%  
    String names[] = new String [3];  
    names[0]="张飞";  
    names[1]="刘备";  
    names[2]="关羽";  
    request.setAttribute("names", names);  
%>  
<c:forEach items="${names}" var="item" >  
    ${item}</br>  
</c:forEach>  
  
<br><br>  
forTokens 测试 1:<br>  
<c:forTokens items="hello-world-how-are-you" delims="-" var="token">  
<c:out value="${token}"/>  
</c:forTokens>
```

9.2 实战实验

结合 EL 表达式编写一个简单的用户注册系统，参考步骤如下：

- 1、 创建注册表单页 reg.html；
- 2、 创建 reg.jsp，在 JSP 文件中显示注册信息，要求使用 EL 表达式接收和显示表单信息；

9.3 实验后任务

结合使用 JSTL 编写一个用户注册系统，参考步骤如下：

- 1、 创建注册表单页 reg.html，至少包含两个元素：username、userpass；
- 2、 创建实体类——User.java，该类中包含两个属性：username、userpass；
- 3、 创建 servlet——RegServlet.java，在 servlet 中接收注册信息并将信息封装到实体 User 对象中；
- 4、 在 Servlet 中将 User 保存到 request 中，然后将请求转发到 regsuccess.jsp；
- 5、 创建 regsuccess.jsp，显示转发过来的 User 中的信息；

参考界面：

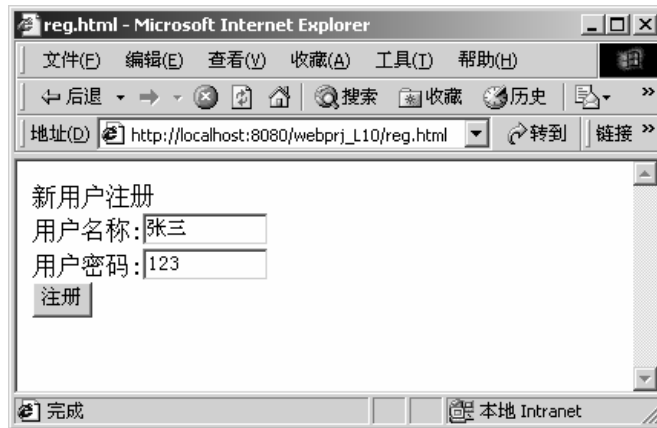


图 9-2 参考界面



图 9-3 参考界面

提示:

显示 request 中的 user 对象的属性可以使用以下代码:

```
<c:out value="${user.username}" default=""/>
```

第十章 JSTL

学习目标

- 使用函数标签
- 使用 SQL 标签
- 使用国际化标签
- 使用 XML 标签

10.1 模拟实验

使用函数标签

1、 创建工程，引入 JSTL 的类库

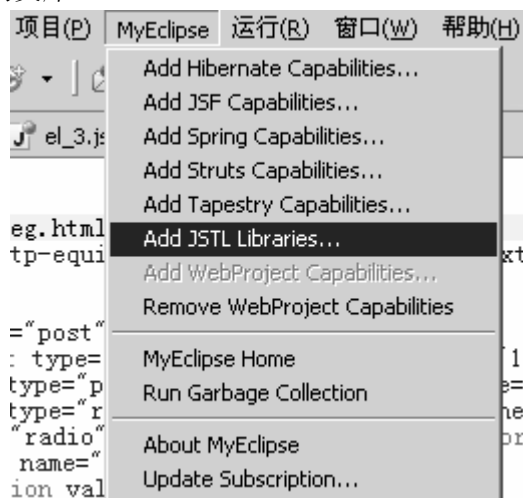


图 10-1 导入 JSTL 类库支持

2、 在工程中创建 jstl_function.jsp，编写以下代码：

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

<h3>函数标签测试</h3>

<c:set var="tel" value="010-88886666"/>
<c:set var="str1" value="www.google.com"/>

[取子字符串]<br>
<b>fn:substring:</b>${fn:substring(tel, 4, 7)}<br>
<b>fn:substringAfter:</b>${fn:substringAfter(tel, "-")}<br>
<b>fn:substringAfter:</b>${fn:substringAfter("hello world!", "hello")}<br>

<b>fn:substringBefore:</b>${fn:substringBefore(tel, "-")}<br>
<br>

[判断是否存在子串]<br>
<b>fn:contains:</b>${fn:contains(str1, "google")}<br>
<b>fn:contains:</b>${fn:contains(str1, "Google")}<br>
<b>fn:containsIgnoreCase:</b>${fn:containsIgnoreCase(str1, "Google")}<br>
```


[判断字符串开头和结尾]


```
<b>fn:startsWith:</b>${fn:startsWith(str1,"www")}<br>
```

```
<b>fn:endsWith:</b>${fn:endsWith(str1,"com")}<br>
```

```
<b>fn:indexOf:</b>${fn:indexOf(str1,"google")}<br>
```

[字符串拆分和连接]


```
<c:set var="array1" value='${fn:split(str1, ".")}' />
```

```
str1:${str1}<br>
```

```
array1:${fn:join(array1, "-")}<br>
```

[字符串过滤]


```
<c:set var="str2" value="请问,<hr>是什么意思?" />
```

```
escapeXml:${fn:escapeXml(str2)}<br>
```

3、 部署工程，访问 jstl_function.jsp,效果如下：



图 10-2 使用函数标签

创建国际化 web 应用

本部分创建一个用户注册的表单，要求该表单能自动适应两种国家的语言：美国、中国，即：美国用户访问该表单自动用英语显示表单界面，中国用户访问时则用中文显示表单界面，效果如下：



图 10-3 中国用户访问“注册表单”



图 10-4 美国用户访问“注册表单”

- 1、 在 1.1 创建的工程中添加两个资源文件：messages_en.properties、messages_zh.properties，这两个文件中分别存放两种语言形式的提示消息：

messages_en.properties 内容如下：

```
ip =new user reg.  
username=Your username:  
userpass=Your userpass:  
submit=Submit  
reset=Reset
```

- 2、 messages_zh.properties 内容如下：

```
tip=新用户注册
```

username=用户名称:

userpass=用户密码:

submit=提交

reset=清除

- 3、 注意：创建以上两个资源文件时，要存放在工程的 `src` 目录下，这样在部署时会自动部署到 `WEB-INF\classes` 目录下，如果创建时选择存放在 `web-inf\classes` 目录下则在部署时会被自动清除。

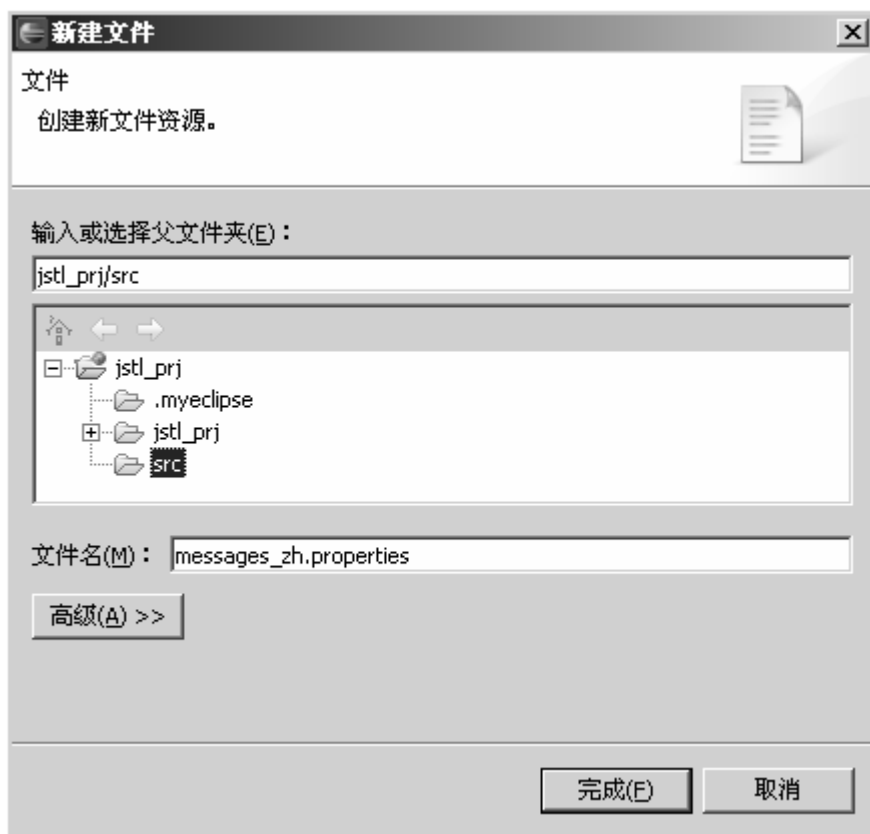


图 10-5 消息文件要存放在 `src` 目录下

- 4、 将 `messages_zh.properties` 的内容进行编码：
- 5、 先将文件内容复制到一个文件 `a.txt` 中，然后使用以下指令：`native2ascii a.txt b.txt`，这样 `b.txt` 中就存放了编码后的数据。`native2ascii.exe` 在 `JAVA_HOMES\bin` 目录下。

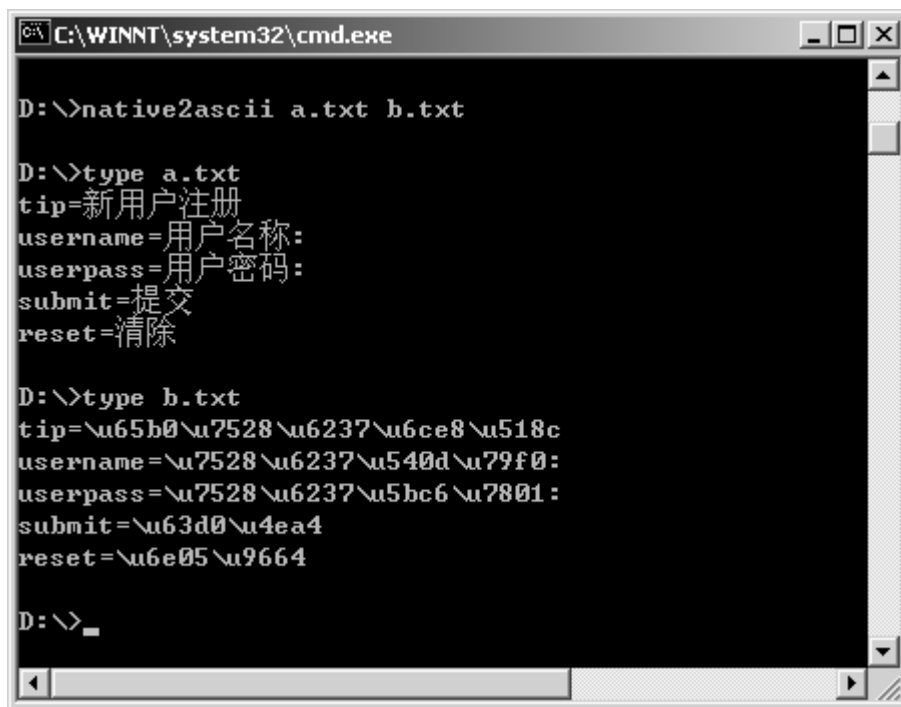


图 10-6 将中文数据编码成 Unicode 码

6、 创建 reg.jsp, 编写以下代码:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ page import = "java.util.Date" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<fmt:setBundle basename="messages" scope="session"/>

<title>
<fmt:message key="tip"/>
</title>
<h2><fmt:message key="tip"/></h2>
<hr>
<form action="regAction.jsp" method="post">
    <fmt:message key="username"/>:
    <input type="text" name="username" size=10><br>

    <fmt:message key="userpass"/>:
    <input type="text" name="userpass" size=10><br>

    <input type="submit" value="<fmt:message key="submit"/>">
    <input type="reset" value="<fmt:message key="reset"/>">
</form>
```

解释：

- ◆ `<fmt:setBundle basename="messages" scope="session"/>`用于装载主文件名为 `messages` 的资源文件；
- ◆ `<fmt:message key="tip"/>`，自动判断用户区域，并从相应的资源文件中提取消息进行显示。

- 7、要想模拟美国用户访问表单，可以将浏览器“语言首选项”中进行设置，设置完新的语言后，刷新浏览器会将客户浏览器模拟成该国家的用户，操作步骤如下：



图 10-7 点击“语言(L)...”

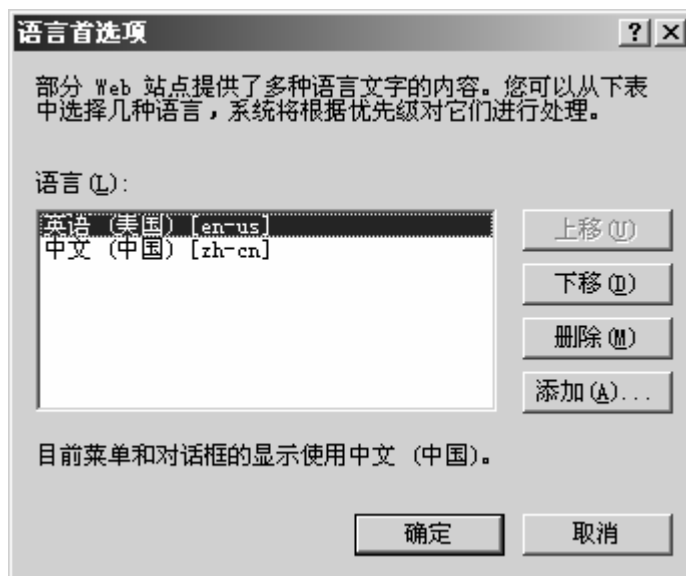


图 10-8 添加新语言并设置成默认值

10.2 实战实验

在 1.2 注册表单的基础上，利用 SQL 标签完成用户注册的功能，参考步骤如下：

- 1、 创建数据库表 users；
- 2、 创建 ODBC 数据源——dbdsn，也可以使用第四类驱动直接连接数据库；
- 3、 创建 regAction.jsp，在该 jsp 中利用 SQL 标签完成向数据库中插入数据的操作。

10.3 实验后任务

使用 xml 标签，显示站点下一个 xml 文件的内容，该 xml 内容如下：

```
<?xml version="1.0" encoding="utf-8"?>
<books>
  <book>
    <bookname>java</bookname>
    <bookprice>30</bookprice>
  </book>
  <book>
    <bookname>jsp</bookname>
    <bookprice>20</bookprice>
  </book>
  <book>
    <bookname>xml</bookname>
    <bookprice>20</bookprice>
  </book>
</books>
```

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

参考界面：

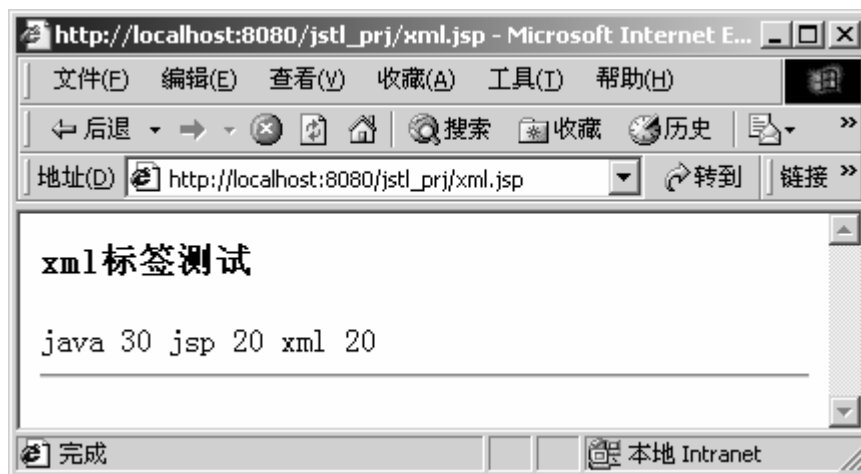


图 10-9 参考界面

提示：

遍历 xml 文档内容可以参考以下代码：

```
<x:forEach select="$doc/books">  
    <x:out select="." /><br>  
</x:forEach>
```


第十一章 自定义标签(一)

学习目标

- 开发自定义的 JSP 标签

11.1 模拟实验

自定义标签

本实验在以前章“商品列表”的基础上，进行修改，在 JSP 页中使用自定义标签完成显示商品列表的操作。

- 1、 在 Eclipse 中创建一个 WEB 工程；
- 2、 创建数据库、DSN 数据源，步骤参见以前的“购物车”系统；
- 3、 向工程中添加文件：db.properties、DBConnection.java、Product.java、ProductDAO.java，用以完成数据库连接、产品信息封装的功能；
- 4、 开发标签处理类 ListProductTag.java，在 doStartTag()方法中读取 ServletContext 作用域中的产品集合然后显示到浏览器中：

```
package demo;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
import javax.servlet.http.*;
import javax.servlet.*;
import java.text.*;
import java.util.*;

public class ListProductTag extends TagSupport {
    public int doStartTag() throws JspException {
        try {
            //得到网络输出流,pageContext 是从父类继承过来的成员;
            JspWriter out = pageContext.getOut();

            ServletContext application = pageContext.getServletContext();
            Collection c = null;
            Object obj = application.getAttribute("productlist");

            //先到 application 中读取商品列表，如果找不到再连接数据库查询商品；
            if(obj == null){
                //实例化 ProductDAO 对象，以处理与产品有关的业务逻辑；
                ProductDAO pdao = new ProductDAO();
                //得到所有商品
                c = pdao.getAllProducts();
                application.setAttribute("productlist",c);
            }
            else{
```

```

        //如果在 application 中找到了商品列表
        c = (Collection)obj;
    }
    Iterator it = c.iterator();
    Product p = null;
    out.println("<table width=70% align=center border=1>");
    out.println("<tr><td colspan=4 align=center>商品列表</td></tr>");
    out.println("<td> 编 号 </td><td> 品 名 </td><td> 单 价 </td><td> 购 买  
</td></tr>");
    //循环显示产品信息
    while (it.hasNext()) {
        p = (Product) it.next();
        out.println("<tr>");
        out.println("<td>" + p.getId() + "</td>");
        out.println("<td>" + p.getProductName() + "</td>");
        out.println("<td>" + p.getProductPrice() + "</td>");
        out.println("<td>购买</td>");
        out.println("</tr>");
    }
    out.println("</table>");
}
catch(Exception e) {
}
return Tag.SKIP_BODY; //跳过标记体;
}
}

```

5、 创建文件 test.tld, 存放在 WEB-INF 目录下:

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>myhr</shortname>
    <tag>

```

```
<name>listproduct</name>

<tagclass>demo.ListProductTag</tagclass>

<bodycontent>EMPTY</bodycontent>

</tag>

</taglib>
```

6、 添加 JSP 页 listproducts.jsp, 使用自定义标签显示商品列表:

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib uri="/WEB-INF/test.tld" prefix="test" %>

<test:listproduct/>
```

7、 运行 WEB 应用, 访问 listproducts.jsp 结果如下:

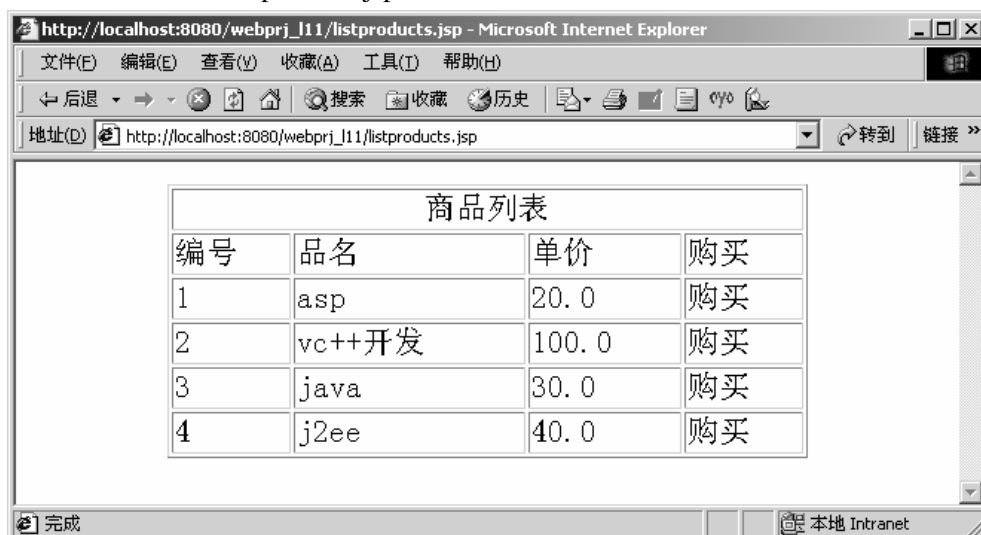


图 11-1 用自定义标记显示产品列表

标签中的属性

示例 1.1 中, 语句 `application.getAttribute("productlist")` 用以从 `application` 中读取商品列表, 但读取数据时键名固定为 `productlist`, 这样很不灵活, 本示例中我们为标签设置一个属性, 用以指明商品集合在 `application` 中存储时使用的键名, 这样我们的标签就可以读取任意键名存储的商品集合了。

1、 修改 ListProductTag.java, 添加属性 `keyname`:

```
package demo;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
import javax.servlet.http.*;
import javax.servlet.*;
```

```

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

public class ListProductTag extends TagSupport {
    private String keyname="productlist";

    public int doStartTag() throws JspException {
        try {
            //得到网络输出流,pageContext 是从父类继承过来的成员;
            JspWriter out=pageContext.getOut();
            ServletContext application=pageContext.getServletContext();
            Collection c=null;
            Object obj=application.getAttribute( this.keyname );

            //先到 application 中读取商品列表, 如果找不到再连接数据库查询商品;
            if(obj==null){
                //实例化 ProductDAO 对象, 以处理与产品有关的业务逻辑;
                ProductDAO pdao=new ProductDAO();
                //得到所有商品
                c=pdao.getAllProducts();
                application.setAttribute("productlist",c);
            }else{
                //如果在 application 中找到了商品列表
                c=(Collection)obj;
            }
            Iterator it=c.iterator();
            Product p = null;
            out.println("<table width=70% align=center border=1>");
            out.println("<tr><td colspan=4 align=center>商品列表</td></tr>");
            out.println("<td> 编 号 </td><td> 品 名 </td><td> 单 价 </td><td> 购 买  
</td></tr>");
            //循环显示产品信息
            while (it.hasNext()) {
                p = (Product) it.next();
                out.println("<tr>");
                out.println("<td>" + p.getId() + "</td>");
                out.println("<td>" + p.getProductName() + "</td>");
                out.println("<td>" + p.getProductPrice() + "</td>");
            }
        }
    }
}

```

```
        out.println("<td>购买</td>");
        out.println("</tr>");
    }
    out.println("</table>");
}
catch(Exception e) {
}
return Tag.SKIP_BODY; //跳过标记体;
}

public void setKeyname(String keyname){
    this.keyname=keyname;
}
}
```

2、 修改 test.tld 文件，在标签中声明属性：

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>myhr</shortname>
    <tag>
        <name>listproduct</name>
        <tagclass>demo.ListProductTag</tagclass>
        <bodycontent>EMPTY</bodycontent>

        <attribute>
            <name>keyname</name>
            <required>false</required>
        </attribute>
    </tag>
</taglib>
```

3、 修改 listproduct.jsp，为标签提供属性：

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

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

```
<test:listproduct keyname="productlist"/>
```

4、访问 listproduct.jsp，运行效果同图 11-1。

11.2 实战实验

编写一个自定义标签，实现两数相加的操作，如 JSP 页中：`<test:add num1="100" num2="60"/>`，则网页输出：“100+60=160”；

1、运行结果参考：



图 11-2 两数相加的自定义标签

提示：在自定义标签类中添加两个属性分别用以保存两个加数。

11.3 实验后任务

1、开发一个自定义 JSP 标记，判断 session 是否存在指定键名的数据，如果存在则在网页上显示“存在”，否则显示“不存在”。

JSP 中使用自定义标签语句示例：

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

```
<%@ taglib uri="/WEB-INF/test.tld" prefix="logic" %>
```

判断 session 中指定键名的数据是否存在:


```
<logic:isPresent name="msg"/>
```

参考页面如下：

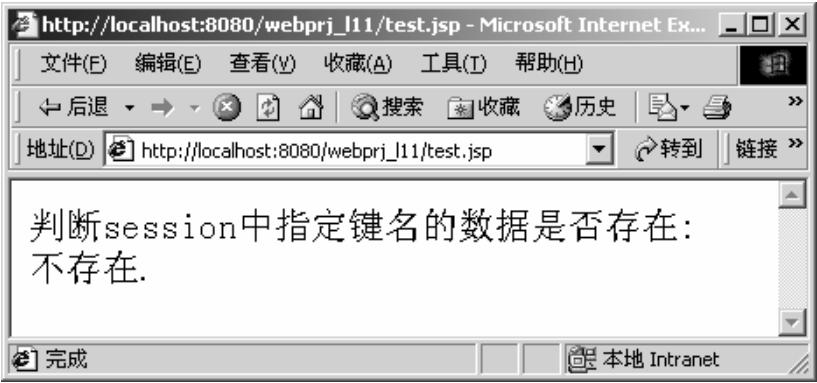


图 11-3 参考结果

第十二章 自定义标签(二)

学习目标

- 开发带标记体的自定义标签
- 开发嵌套自定义标签

12.1 模拟实验

带标记体的自定义标记

在许多应用中需要过滤字符串中的 HTML 标记，使之原样显示到浏览器中。如：在聊天室中应该将用户发言中的“<”、“>”这些字符过滤掉，否则会形成 HTML 指令显示到浏览器中。假设：用户发言时输入以下一句话：“请问在 html 中,<hr>是什么意思?”，如何使之原样显示到浏览器中，而不是将<hr>显示成水平线。下面开发一个自定义标记来实现此功能。

1、开发标记处理类——HTMLEncodeTag.java，代码如下：

```
package demo;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class HTMLEncodeTag extends BodyTagSupport {

    public int doStartTag() throws JspTagException {
        //以下返回值代表将标记体内容缓存到 bodyContent 成员中;
        return EVAL_BODY_BUFFERED;
    }

    public int doEndTag() throws JspTagException {
        BodyContent body = this.getBodyContent();
        //得到缓冲中的标记体正文
        String content = body.getString();
        //从缓冲中清除标记体正文
        body.clearBody();
        String result = htmlEncode(content);
        try{
            pageContext.getOut().write( result );
        }catch(IOException e){
            e.printStackTrace();
        }
        return SKIP_BODY;
    }

    private String htmlEncode(String source){
        String temp1 = source.replaceAll("<","&lt;");
        String temp2 = temp1.replaceAll(">","&gt;");
        return temp2;
    }
}
```

```
}
}
```

2、 创建 TLD 文件:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>test</shortname>
    <tag>
        <name>htmlencodetag</name>
        <tagclass>demo.HTMLEncodeTag</tagclass>
    </tag>
</taglib>
```

3、 创建 JSP——tag1.jsp, 使用以上标签

```
<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/test.tld" prefix="test" %>
测试带标记体的自定义标记:<br>
<test:htmlencodetag>
    请问在html中,<hr>是什么意思?
</test:htmlencodetag>
```

4、 部署工程, 访问 tag1.jsp, 结果如下:

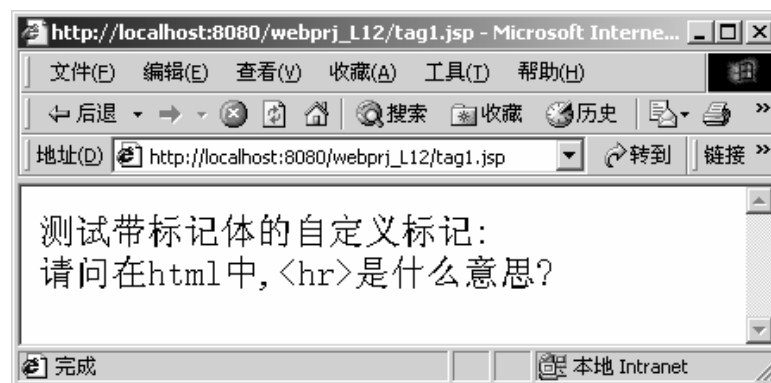


图 12-1 带标记体的 Tag

标记中的子标记

本部分创建一个可嵌套的自定义标记，在本标记内部嵌入了一个标准的 JSP 标记。本标记判断 session 中存储的用户名是否为指定值，如果不是指定值则调用标准 JSP 标记<jsp:forward>转到另外一个页面。

1、 创建标记处理类——NotEqualTag.java

```
package demo;

import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class NotEqualTag extends BodyTagSupport {
    private String name;
    private String value;

    public void setName(String name){
        this.name = name;
    }
    public void setValue(String value){
        this.value = value;
    }
    public int doStartTag() throws JspTagException {
        String username=(String)pageContext.getSession().getAttribute(name);
        //如果 session 中键名为 name 的数据不等于 value,则运行子标记;
        if (!username.equals(value)) {
            return EVAL_BODY_INCLUDE;//自动将标记体包含到输出流中;
        }
        else{
            return SKIP_BODY; //跳过标记体，不进行处理;
        }
    }
}
```

2、 在 TLD 文件中配置自定义标记:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">
```

```

<taglib>
  <tlibversion>1.0</tlibversion>
  <jspversion>1.1</jspversion>
  <shortname>test</shortname>
  <tag>
    <name>notequal</name>
    <tagclass>demo.NotEqualTag</tagclass>
    <bodycontent>JSP</bodycontent>
    <attribute>
      <name>name</name>
      <required>true</required>
    </attribute>
    <attribute>
      <name>value</name>
      <required>true</required>
    </attribute>
  </tag>
</taglib>

```

3、 创建 JSP——tag2.jsp，使该标签：

```

<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/test.tld" prefix="test" %>
测试嵌套的自定义标记:<br>
<%
    session.setAttribute("username","tom2");
    //如果 username 不等于 tom2，则定向到主页 index.html
%>
<test:notequal name="username" value="tom2">
  <jsp:forward page="index.html" />
</test:notequal>
<h2>username 等于 tom2!</h2>

```

4、 部署工程到 tomcat，当 username 的值不为 tom 时，tag2.jsp 会自动转发到 index.html 中，可以自行修改 session.setAttribute("username","tom2")中的值进行测试。

12.2 实战实验

开发一个标签，该标签标记体是一条 SQL 查询指令（select），在标签内部读取该 SQL 语句的内容，并查询数据库，然后将查询到的数据显示到浏览器中。

提示：

- 1、 连接数据库的操作可以使用以前章创建过的 `DBConnection` 类；
- 2、 由于标签中 SQL 指令查询的表不确定，所以结果集中的字段个数也不定，可以使用以下代码得到结果集中的字段数量：

```
ResultSet rs = db.executeQuery( sql );
ResultSetMetaData meta = rs.getMetaData();
//计算结果集的列数；
int count = meta.getColumnCount();
```

- 3、 JSP 示例代码如下：

```
<%@ page language="java" contentType="text/html; charset=GB2312"%>
<%@ taglib uri="/WEB-INF/test.tld" prefix="test" %>
第一次测试:<br>
<test:sqltag>
    select * from users where username='admin'
</test:sqltag>
<hr>
第二次测试:<br>
<test:sqltag>
    select * from users
</test:sqltag>
```

- 4、 运行以上 JSP 时的参考界面：

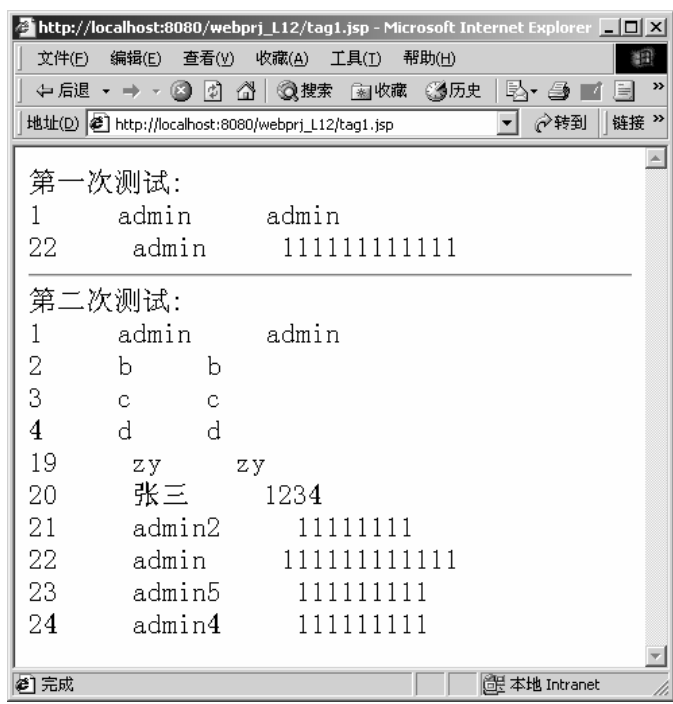


图 12-2 参考界面

12.3 实验后任务

- 1、 为实战实验中的自定义标签添加属性：**border**、**bgcolor**，将查询出来的数据显示在表格中，两个属性分别用以确定表格的边框粗细、表格背景色。参考界面如下：



图 12-3 作业参考界面

第十三章 MVC 实现

学习目标

- 使用 MVC 模式开发 Java Web 应用

13.1 模拟实验

网上投票

假设某网站要开发一个网上投票系统，且本网站准备长期重复使用该投票系统，为保证系统的可维护性，使用 MVC 模式进行开发，步骤如下：

- 1、 创建数据库及数据库表，并为表输入初始数据，创建 ODBC 数据源 dbdsn:

	列名	数据类型	长度	允许空
🔑	id	int	4	
	site	nvarchar	50	✓
▶	vote	int	4	✓

图 13-1 数据库表 vote

	id	site	vote
▶	1	baidu	0
	2	yahoo	0
	3	google	0
*			

图 13-2 数据库表 vote 的初始数据

- 2、 创建模型层中的——DBConnection.java、db.properties，其中前者提供数据库底层操作功能，后者提供数据库配置信息，两者代码请参考前面章。
- 3、 创建模型层中的——Vote.java，用于封装得票信息

```
package demo;
import java.io.Serializable;
import java.util.*;

public class Vote implements Serializable{
    private String site;//站点名
    private int vote;//票量

    public Vote(){
    }
    public void setSite(String site){
        this.site=site;
    }
    public void setVote(int vote){
        this.vote=vote;
    }
}
```

```

    }
    public String getSite(){
        return this.site;
    }
    public int getVote(){
        return this.vote;
    }
}

```

4、 创建模型层中的——VoteDAO.java, 用于提供与得票有关的业务方法

```

package demo;

import java.util.*;
import java.sql.*;

/**
 * 本类用于封装与 vote 表有关的业务逻辑方法;
 */
public class VoteDAO extends DBConnection{

    //计票
    public void count(Vote vote){
        //得到投票的站点名
        String site = vote.getSite();
        //计票
        String sql;
        sql = "update vote set vote=vote+1 where site='" + site + "'";
        super.executeUpdate(sql);
    }
    //得到所有得票总量;
    public int getVoteSum(){
        int sum = 0;
        ResultSet rs = executeQuery("select sum(vote) from vote");
        try{
            rs.next();
            sum = rs.getInt(1);
        }catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

```
    }  
    return sum;  
}  
public ArrayList getVotes(){  
    ArrayList list = new ArrayList();  
    ResultSet rs = executeQuery("select * from vote");  
    try{  
        while (rs.next()) {  
            Vote vote = new Vote();  
            vote.setSite( rs.getString("site") );  
            vote.setVote( rs.getInt("vote") );  
            list.add(vote);  
        }  
        rs.close();  
    }catch(Exception e){  
        e.printStackTrace();  
    }  
    return list;  
}  
}
```

5、 创建视图层中的——vote.jsp，显示投票表单

```
<%@ page language="java" contentType="text/html; charset=gb2312" %>  
<%@ taglib uri="/WEB-INF/test.tld" prefix="html" %>  
<html>  
    <head>  
        <title>网上投票</title>  
    </head>  
    <body>  
        <form action="voteservlet" method="post">  
            请选择你最喜欢的搜索引擎: <br>  
            <INPUT type="radio" name="site" value="baidu">百度<br>  
            <INPUT type="radio" name="site" value="yahoo">雅虎<br>  
            <INPUT type="radio" name="site" value="google">Google<br>  
            <input type="submit" value=" 投票 " >  
        </form>  
        <hr>  
        以下为投结结果:<br>
```

```

<html:showvote/>

</body>

</html>

```

6、 创建视图层中的——ShowvoteTag.java，这是一个自定义标记，用以在 JSP 中显示投票信息

```

package demo;

import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
import java.io.IOException;
import javax.servlet.http.*;
import javax.servlet.*;
import java.util.*;

public class ShowvoteTag extends TagSupport{
    public int doStartTag() throws JspException {
        try {
            //得到网络输出流;
            JspWriter out = pageContext.getOut();

            VoteDAO dao = new VoteDAO();
            Vote temp;
            int sum = dao.getVoteSum();
            Iterator it = dao.getVotes().iterator();

            //向网页输出消息;
            out.println( "<table align=left border=0>" );
            out.println( "    <tr><td> 网 站 </td><td width=80> 得 票 量"
            </td><td>&nbsp;</td></tr>" );
            while(it.hasNext()){
                temp = (Vote)it.next();
                String site = temp.getSite();//站点名
                int vote = temp.getVote();//得票量

                //计算各站票量所占总票的百分比,按百分比计算条形图长度;
                double width = ( (double)vote/(double)sum ) * 400;
                long width2 = Math.round( width );
            }
        }
    }
}

```

```

        out.println( "<tr><td>" + site + "</td><td width=80>" + vote
            + "</td><td><img src=bar.gif width="
            + width2 + "></td></tr>" );
    }
    out.println( "</table>" );
}
catch(Exception e) {
}
return Tag.SKIP_BODY; //跳过标记体;
}
}

```

7、 在 WEB-INF 目录创建 test.tld，配置自定义标记：

```

<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE taglib
    PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.1//EN"
    "http://java.sun.com/j2ee/dtds/web-jsptaglibrary_1_1.dtd">

<taglib>
    <tlibversion>1.0</tlibversion>
    <jspversion>1.1</jspversion>
    <shortname>test</shortname>
    <tag>
        <name>showvote</name>
        <tagclass>demo.ShowvoteTag</tagclass>
        <bodycontent>EMPTY</bodycontent>
    </tag>
</taglib>

```

8、 创建控制器——VoteServlet.java，处理投票逻辑：

```

package demo;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;

```

```
import javax.servlet.http.HttpServletResponse;

public class VoteServlet extends HttpServlet {
    /**
     * The doPost method of the servlet. <br>
     */
    public void doPost(HttpServletRequest request,
                       HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        //1、接收投票信息;
        String site = request.getParameter("site");
        //2、生成 Vote 实体，将信息封装到实体中
        Vote v = new Vote();
        v.setSite(site);
        //3、调用 DAO 进行计票，更新数据库;
        VoteDAO dao = new VoteDAO();
        dao.count(v);          //4、转到显示投票结果页面;
        response.sendRedirect("vote.jsp");
    }
}
```

9、 部署工程到 tomcat，运行效果如下：

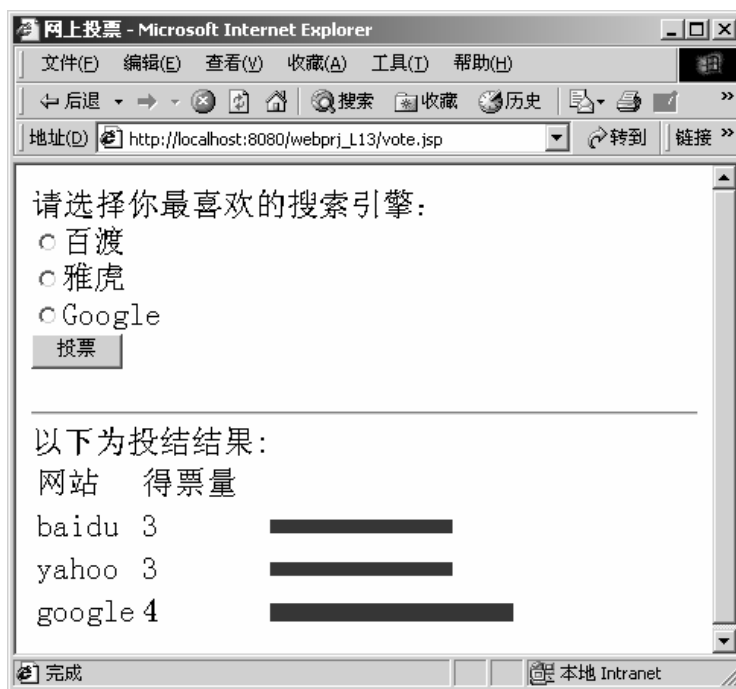


图 13-3 网上投票

13.2 实战实验

使用 MVC 模式开发一个简单的系统：在表单中输入两个加数，然后在浏览器中输出两数相加的和。参考步骤如下：

- 1、创建表单 JSP 页面 `add.jsp`，其中包含两个用于接收加数的文本框；
- 2、创建模型 `javabean`——`Add.java` 用以封装两个加数及求和的方法；
- 3、创建控制器 `Servlet`——`AddServlet.java`，接收表单数据进行表单验证并将数据封装到 `javabean`、再求和，将和存储到 `request` 作用域，然后转发到显示结果的 JSP 页面；另外表单验证失败时要传递错误信息到 JSP 页面；
- 4、创建显示错误信息和结果的 JSP 页面——`result.jsp`，
- 5、可选：创建自定义标签，用于显示求和的结果和错误提示信息。

➤ 参考界面：

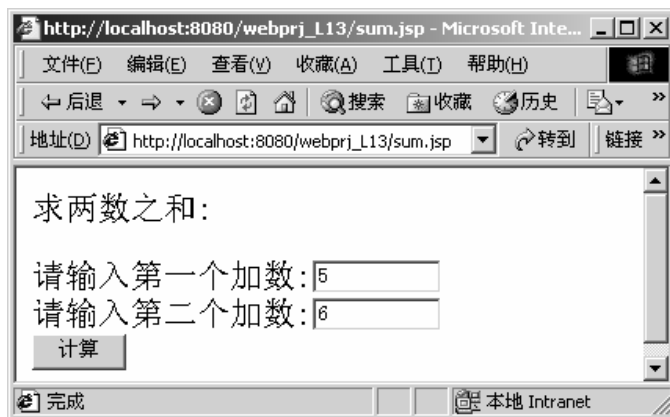


图 13-4 参考界面

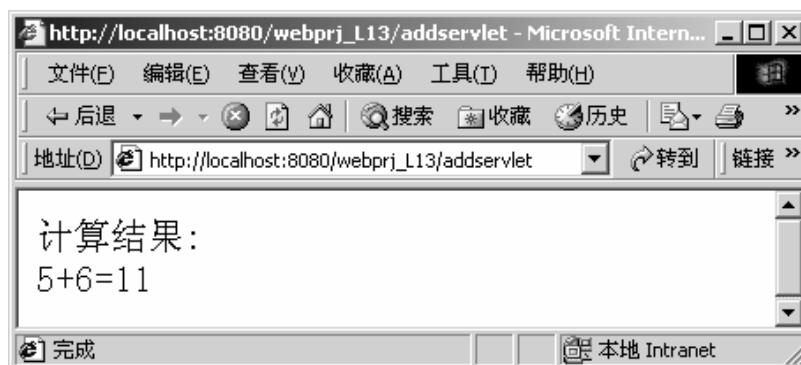


图 13-5 参考界面

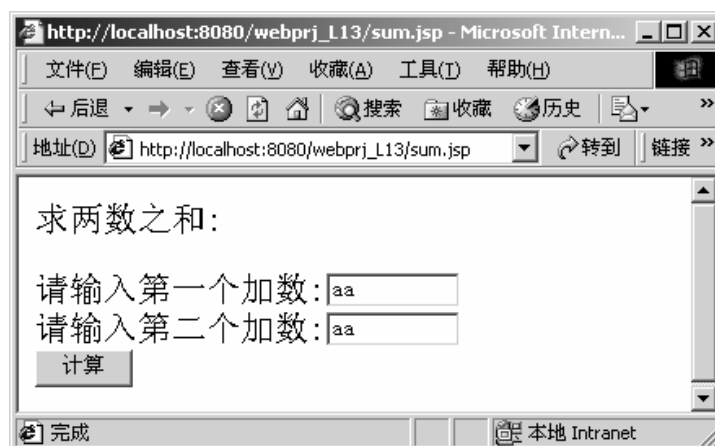


图 13-6 参考界面

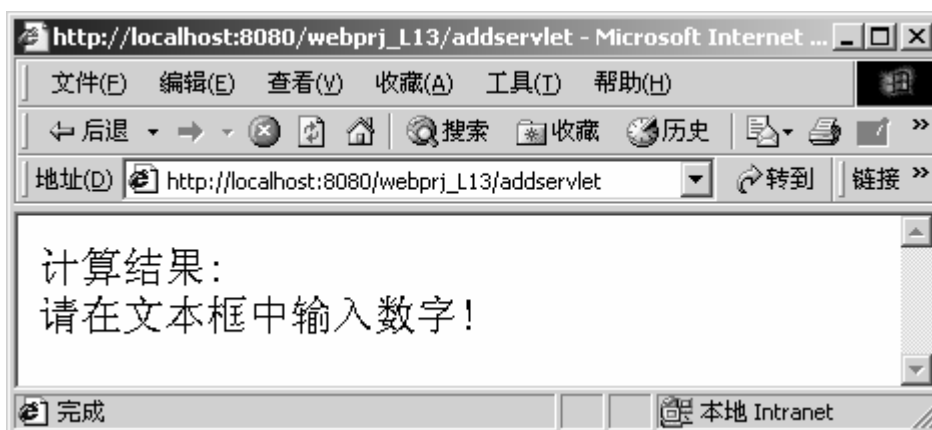


图 13-7 参考界面

13.3 实验后任务

1、 使用 MVC 模式开发一个用户登录系统，参考界面如下：



图 13-8 登录表单

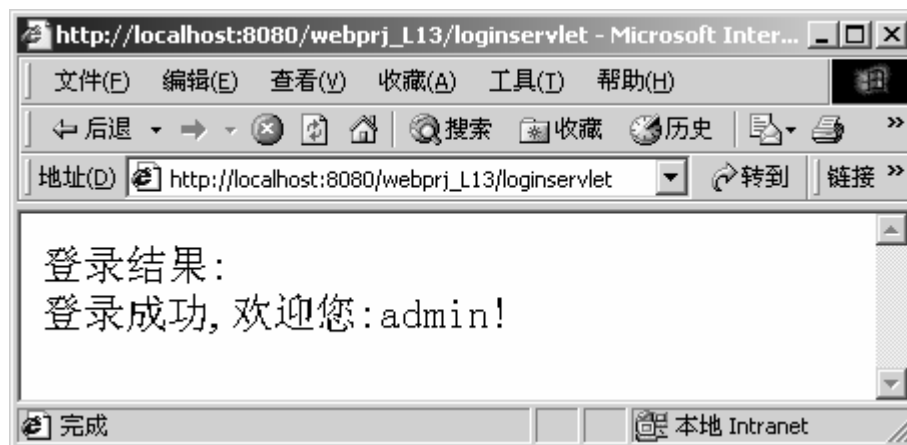


图 13-9 登录成功

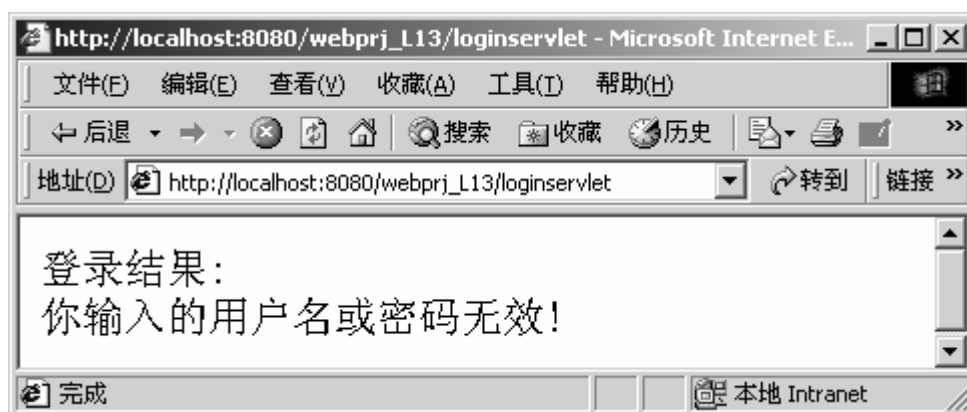


图 13-10 登录失败

要求:

登录成功/失败的信息用自定义标签显示。

第十四章 Ajax 框架

学习目标

- 使用 Ajax 框架 DWR

14.1 模拟实验

使用 DWR 框架

本部分要完成的功能：在 html 网页中，向服务器端的 JAVA 类传递两个加数，然后得到两数之和显示在网页中的文本区中。

考虑：

如果不用 Ajax，如何完成相同功能？可以在 servlet 中去计算，将结果保存到作用域中，要显示结果只能在 JSP 页中去搜索某个作用域中的数据，也即不能在静态网页*.html 中得到运算结果。但通过 Ajax 结合 DWR 框架，可以很方便的实现此功能。

以下为实现步骤：

- 1、 创建 WEB 站点 ajax-demo;
- 2、 将 DWR 支持库复制到网站 WEB-INF\lib 目录下。
- 3、 在 WEB.XML 中配置 DWRServlet:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application
2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <display-name>Ajax Examples</display-name>
    <servlet>
        <servlet-name>dwr-invoker</servlet-name>
        <display-name>DWR Servlet</display-name>
        <description>Direct Web Remoter Servlet</description>
        <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
        <init-param>
            <param-name>debug</param-name>
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>dwr-invoker</servlet-name>
        <url-pattern>/dwr/*</url-pattern>
    </servlet-mapping>
</web-app>
```

- 4、 开发一个 JAVA 类，编译到 WEB-INF\classes 目录下:

```
package com.test;

public class Demo {
```

```

    public int add(int x,int y){
        return x+y;
    }
}

```

- 5、 在 WEB-INF 目录下创建名为 dwr.xml 的配置文件，在该文件中可以配置要将哪些类共享出去让 Ajax 访问，配置如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE dwr PUBLIC
    "-//GetAhead Limited//DTD Direct Web Remoting 0.4//EN"
    "http://www.getahead.ltd.uk/dwr/dwr.dtd">
<dwr>
    <allow>
        <create creator="new" javascript="Test" scope="application">
            <param name="class" value="com.test.Demo" />
        </create>
    </allow>
</dwr>

```

- 6、 编写页面 dwr_1.html，通过 Javascript 调用 Java 类：

```

<html>
<head>
<title>DWR 示例</title>
<script type='text/javascript' src='dwr/engine.js'> </script>
<script type='text/javascript' src='dwr/interface/Test.js'></script>
<script type='text/javascript' src='dwr/util.js'> </script>
<script type="text/javascript">
function callBack(data){
    result.value="2+3="+data;
}
</script>
</head>
<body>
<input type="Button" name="button3" value=" 测 试 "
onclick="Test.add(callBack,2,3)" />
<br>
<textarea id="result" rows="5" cols="60"></textarea>
</body>

```

```
</html>
```

解释:

- `onclick="Test.add(callBack,2,3)`, 代表在点击按钮时, 调用服务器端的 `Test` 类的 `add()` 方法, 并将参数 2、3 传递过去, 即调用: `add(2,3)`;
- `onclick="Test.add(callBack,2,3)` 中的 `callBack` 用于指明哪个 javascript 函数接收服务器端返回的数据。

7、配置完成, 启动 Web 应用, 输入 `http://localhost:8080/ajax-demo/dwr_1.html`, 效果如下 (服务器的回应显示在了文本区中):

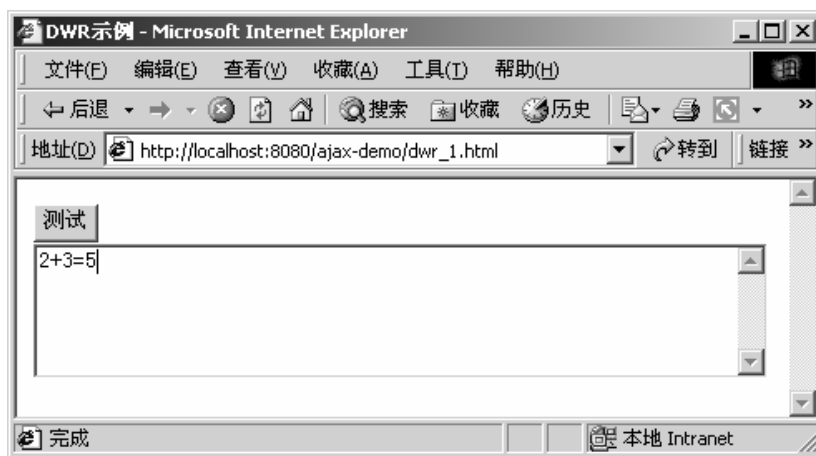


图 14-1 使用 DWR 框架

14.2 实战实验

使用 DWR 框架, 完成用户查询功能: 要求在文本框中输入用户名, JAVA 类从数据库中查询该用户对应的密码并返回, 如果查不到此人则返回“查无此人。”, 返回的密码显示在另一个文本框中。

参考步骤:

- 1、创建数据库表 `users`, 填写测试数据;
- 2、创建 ODBC 数据源——`dbdsn`, 也可以使用第四类驱动直接连接数据库;
- 3、创建 `query.html` 页, 其中包含两个文本框、一个“查询”按钮。
- 4、编写服务器端 JAVA 类;
- 5、配置 `dwr.xml`;
- 6、在 `query.html` 页中编写 javascript, 实现对服务器端类的调用。

参考界面:



图 14-2 参考界面

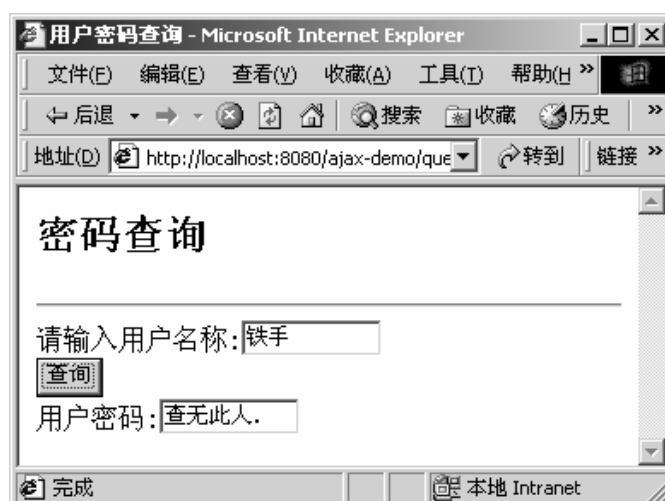


图 14-3 参考界面

14.3 实验后任务

使用 DWR 框架，开发如下功能：用户在文本框中输入单词的首字母时，自动在文本区中显示以该字母开头的单词（单词数量自己定）。

参考界面：



图 14-4 根据首字母自动显示匹配单词

提示：

- 文本框事件可用：onKeyUp，在该事件中调用 javascript 访问服务器端 JAVA 类。
- 单词可事先存储在 JAVA 类中。



附录一 练习答案

第一章

- 1、 MVC 模式中，如果想设定出货价格必须高于进货价格 80%这样的业务规则，可以将这个规则放在哪个部分来完成？

Model 部分

- 2、 如果想以 `http://localhost:8080/`来访问开发的 web 应用，应该将该应用放在 tomcat 文件夹中的什么位置？

webapps 目录下

第二章

- 1、 开发一个 Servlet，有几种方法？分别如何实现？

共三种方法：直接实现 Servlet 接口、继承 GenericServlet、继承 HttpServlet。

- 2、 简述 HttpServlet 中以下方法的联系：service()、doGet()、doPost()？

service()方法会根据客户发送请求的类型自动调用 doGet()、doPost()方法，所以开发 HttpServlet 时，覆盖 doXXX()方法即可，不必覆盖 service()。

- 3、 简述 Servlet 生命周期及各生命时刻要运行的方法是什么？

第一个客户来访时：

- ◆ Servlet 类的实例被构造出来
- ◆ 运行 init()方法进行初始化操作
- ◆ 运行 service()方法为客户提供服务

第二个以后的客户来访时：

- ◆ 直接运行 service()方法为客户提供服务

如果长时间无人来访，或容器被停止：

- ◆ 此时 Servlet 会被销毁，运行 destroy()方法处理收尾工作

第三章

- 1、 简述 HttpServletRequest、HttpServletResponse 两个接口的功能及常用方法？

1) HttpServletRequest 接口

用于封装 HTTP 请求消息，该接口定义了获得请求信息的方法，开发人员可以通过该接口读取到客户发送到服务器的数据；

常用方法：

- `getParameter()` 读取指定表单元素的数据
- `setCharacterEncoding()` 设置读取数据时用的字符编码

2) `HttpServletResponse` 接口

将数据或 `Http` 头信息发回客户端，可以向客户端发送文本或二进制数据；

常用方法：

- `getWriter()` 得到字符输出流
- `setContentType()` 设置输出内容格式及字符编码

2、`ServletConfig` 与 `ServletContext` 有何区别？

`ServletConfig` 存储特定 `servlet` 的信息，每一个 `servlet` 都有一个不同的 `ServletConfig` 对象

`ServletContext` 存储 `WEB` 应用的全局信息，所有 `servlet` 共享一个 `ServletContext` 对象

3、请求重定向、请求转发、请求包含分别如何实现，三者有何区别？

- 请求重定向

`response.sendRedirect(url)`

- 请求转发

```
ServletContext servletContext=this.getServletContext();
RequestDispatcher requestDispatcher =
    servletContext.getRequestDispatcher("/RequestDispatcherServlet_2");
requestDispatcher.forward( request, response ); //请求转发
```

- 请求包含

```
ServletContext servletContext=this.getServletContext();
RequestDispatcher requestDispatcher =
    servletContext.getRequestDispatcher("/RequestDispatcherServlet_2");
requestDispatcher.include( request, response ); //请求转发
```

三者区别：

- 请求重定向其实是客户端进行了第二次请求，原来的请求已经丢失；
- 请求转发是将第一个请求直接传递给了第二个 `servlet`，客户端只生成一次请求，该请求被两个 `servlet` 处理，但客户端只能看到第二个 `servlet` 的结果
- 请求包含是将第一个请求直接传递给了第二个 `servlet`，客户端只生成一次请求，该请求被两个 `servlet` 处理，第二个 `servlet` 的结果被包含到第一个 `servlet` 结果中，客户端可以看到两个 `servlet` 的结果

第四章

1、`Session` 与 `Cookie` 之间有何关系？

`Session` 要借助 `Cookie` 将 `sessionid` 存储到客户端硬盘上，`session` 通常在底层要依赖于 `Cookie`，当客户不支持或禁止了 `Cookie` 时，`HttpSession` 对象会自动使用其他会话跟踪技术（`URL` 重写、隐藏表单元素）将 `sessionid` 写入客户端。

2、`session` 中可以存储对象吗？`Cookie` 中呢？

session 中可以存储对象，而 Cookie 中只能存储简单的文本数据。

第五章

1、 简述 Servlet 容器运行一个 Servlet 的机制？

Servlet 容器只为同一 Servlet 类生成一个实例，该实例服务于所有的请求，对于客户端同时请求一个 servlet，他们是被并发的处理的，并不是等上一个请求处理完成再处理下一个。如果两个请求同时到达，那么他们处理完成的时间也是差不多的，即服务器中可能同时有多个线程在使用这个对象。

在用 Servlet 构建的 Web 应用时如果不注意线程安全的问题，会使所写的 Servlet 程序有难以发现的错误。

2、 本地变量是如何解决 Servlet 线程安全问题的？

本地变量指作用在某个程序块内，而其它程序块中的代码不能访问的数据项，而定义在方法内部的局部变量会在每个线程中保存一份副本。

线程之间无法相互直接访问局部变量，所以将变量声明为局部变量（对线程而言就是本地变量，因为线程将该局部变量副本复制到了每个线程自己的空间），可以有效避免 Servlet 多线程引发的问题。

3、 简述同步操作如何解决 Servlet 线程安全？

通过同步操作可以保证一个实例只能被一个线程访问，其他线程要访问必须等当前线程访问结束后才能取得对实例的访问权。

同步方法会降低 Servlet 的整体工作效率，因为服务器中同一时刻只能为一个用户服务，所以不提倡使用同步方法来解决 Servlet 线程安全问题，最佳方案还是在 Servlet 中使用本地变量。

4、 简述三个 Servlet 对象存储和传递数据的特征。

HttpServletRequest——存储在该对象中的数据可以跨两个 WEB 资源进行存取，前提是需要将当前请求转发到第二个 WEB 资源中。

HttpSession——在用户一次会话中，可以在任何 WEB 资源（servlet/jsp）中访问存储在该对象中的数据。当需要存储用户会话级的数据时（如：聊天室中的昵称），可以使用该对象。

ServletContext——该对象中存储的数据，是网站的全局数据。可以被任何用户、在任何 WEB 资源中访问到。

第六章

1、 简述 JSP 页面的运行流程，说明 JSP 与 Servlet 的联系？

客户请求 JSP 页，如果是第一次请求则由 WEB 服务器为 JSP 页生成相应的 servlet，并编译成 class 文件，然后执行该 servlet，运行结果由 WEB 服务器输出到客户端。JSP 归根到底要被转成 Servlet 才能运行。

2、 简述声明、代码段、表达式三种语法及其作用？

➤ 声明（Declaration）

可以声明页面范围的变量和方法。

语法为：<%! 变量或方法声明 %>

➤ 代码段 (Scriptlet)

嵌入页面内的是一些 java 代码片断。

语法为: `<% 代码片段 %>`

➤ 表达式 (Expression)

可以将表达式的结果变成 String 类型以便于包含在页面的输出中。

语法为: `<%= 表达式 %>`, 表达式可以为任意合法的 JAVA 表达式。

3、简述 page 指令中几个重要属性的功能?

➤ `import="package.*,package.class"`

声明需要导入的 Java 包的列表, 这些包可用于程序段, 表达式, 以及声明。

➤ `contentType="xxx"`

设置 MIME 类型。缺省 MIME 类型是: text/html, 缺省字符集为 ISO-8859-1.

4、为什么说 include 指令是静态包含? 使用时有什么注意事项?

➤ 因为 include 指令包含其他 jsp 时, 包含的是 jsp 的源代码, 而不是其他 jsp 的运行结果。

➤ 既然是以源代码的方式进行包含, 则两个 JSP 页中出现相同声明 (数据成员或方法成员) 时, 会出现定义冲突。

5、request、response 对象主要功能是什么, 分别与 Servlet 中的哪些 API 对应?

➤ request 对象用于接受客户端通过 HTTP 协议连接传输到服务器端的数据, 是 javax.servlet.http.HttpServletRequest 接口的对象。

➤ response 对象主要用于向客户端发送数据, 是 javax.servlet.http.HttpServletResponse 接口的对象。

第七章

1、JSP 中存储数据的对象有几种, 分别有何种作用域?

有 4 种作用域对象, 其作用域由小到大分别为:

➤ pageContext: 存储页面级的数据, 页面运行结束时数据会消失;

➤ request: 请求级数据, 请求被转发到其他 JSP 或 Servlet 时这些数据可以被转发过去;

➤ session: 会话级数据, 在一个会话中的所有 JSP 页或 Servlet 中都可以访问到 session 中的数据;

➤ application: 全局级数据, 任何用户在任何 JSP 页或 Servlet 中都可以访问到 application 中的数据。

2、pageContext 对象中的 findAttribute()方法, 有什么作用?

findAttribute()方法的参数是一个 String 类型的键名, 该方法可以自动在 4 种作用域中搜索指定键名的数据, 搜索顺序为: pageContext、request、session、application, 如果在低级别作用域中找到数据则会停止搜索。

第八章

1、 举例说明何时应该使用 jsp 动作标记？

通过 jsp 标记可以将 JSP 页面中的代码段数量减少到最少，从而实现显示与业务逻辑的分离。所以在 JSP 中要使用代码段时可以考虑有无相应的动作标记可以来代替代码。JSP 标准标记中提供了三种功能：文件包含、请求转发、javabean 操作，在 JSP 页中实现这三部分功能时可以使用标准 JSP 标记。

要在 JSP 页中完成其他功能，需要自定义的 JSP 标记来完成。

2、 简单说明操作 javabean 的三个动作标记的用法？

bean 相关动作标记共有三个，语法分别如下：

`<jsp:useBean>` 创建一个 Bean 实例并指定它的名字和作用范围。

语法：

```
<jsp:useBean id="beanInstanceName" scope="page | request | session | application"
class="package.class" | type="package.class" /> </jsp:useBean>
```

说明：id——javabean 的名字；scope——javabean 要存储的范围；class——javabean 所属的类；type——id 的类型，即引用的类型，它可以是 javabean 的任何父类或本类型。

`<jsp:setProperty>` 动作用于向一个 javabean 的属性赋值，需要注意的是，在这个动作中将会

使用到的 name 属性的值将是一个前面已经使用 `<jsp:useBean>` 动作引入的 javabean 的名字。

语法：`<jsp:setProperty name=" id" property=" propertyname" />`

`<jsp:getProperty>` 动作用于从一个 javabean 中得到某个属性的值，无论原先这个属性是什么类型的，都将被转换为一个 String 类型的值。

语法：`<jsp:getProperty name=" id" property=" propertyname" />`，不论属性是什么类型，该标记都会将属性值转成 String 类型。

第九章

1、 简述 EL 表达式语言的功能及主要语法？

使用 EL 可以方便地访问和处理应用程序数据，而无需使用 JSP 代码段或 JSP 表达式进行编程(即不需要使用 `<% 和 %>` 来获得数据)。

EL 主要的语法结构：`${10+1}`，所有 EL 都是以 `${` 为起始、以 `}` 为结尾的。

2、 简述 JSTL 核心标签库的四类功能及分别用什么标签来实现？

JSTL 核心标签库(Core)用来实现一些通用的操作，主要有 4 组操作：基本输入输出、流程控制、迭代操作和 URL 操作。

各组功能使用的标签：

变量定义及输出操作：`out`、`set`、`remove`、`catch`

流程控制：`if`、`choose`、`when`、`otherwise`

循环操作：`forEach`、`forTokens`、`Core`

URL 操作：`import`、`param`、`url`、`param`、`redirect`、`param`

3、 有以下代码：`session.setAttribute("user",user)`，要在 JSP 中显示 user 的 username 属性值，使用 JSTL

如何实现？

用以下代码实现：

```
<c:out value="${user.username}" default="" />
```

或

```
<c:out value="${sessionScope.user['username']}" />
```

或

```
<c:out value="${sessionScope.user.username}" />
```

第十章

- 1、 SQL 标签库中的哪个动作用于从 JSP 页面将数据插入数据库？

`<sql:update>`，用于执行 insert,update,delete 语句。

- 2、 通过 JSTL 使项目中的文本实现国际化的步骤，假设该项目要求自动在中英两种语言间切换？

步骤如下：

创建两个消息资源文件：messages_en.properties、messages_zh.properties；

messages_en.properties 中存放英文消息；

messages_zh.properties 中存放中文消息，且中文要通过 native2ascii 编码，即文件中存放的是 unicode 编码；

装载资源文件：<fmt:setBundle basename="messages" scope="session"/>;

读取并显示消息：<fmt:message key="title"/>;

第十一章

- 1、 简述开发一个自结束自定义标签的步骤？

步骤如下：

- 1) 开发标记处理类，编译生成 class 文件，该类要继承 TagSupport 或 BodyTagSupport；
- 2) 创建标记库描述符文件*.tld，在该文件中为标记处理类指定标签名、声明标签属性；
- 3) 在 JSP 中引用标签库；
- 4) 在 JSP 中使用标 JSP 标签。

- 2、 简述在 JSP 页面中引用 TLD 文件的两种方式？

➤ 方式一：动态引用（即直接在 JSP 页面中使用 TLD 文件），语法如下：

```
<%@ taglib uri="/WEB-INF/myhr.tld" prefix="test" %>
```

➤ 方式二：静态引用

a. 首先在 web.xml 中为 TLD 文件声明别名：

```
<taglib>
    <taglib-uri>myhr</taglib-uri>
    <taglib-location>/WEB-INF/myhr.tld</taglib-location>
</taglib>
```

在 JSP 中通过别名引用 TLD 文件：

```
<%@ taglib uri="myhr" prefix="test" %>
```


第十二章

1、 要将标记体内容输出到浏览器有几种方法，分别如何实现？

有两种方法：

- 在 `doStartTag()` 中返回 `EVAL_BODY_INCLUDE`，标记体内容会直接输出到浏览器中；
- 在 `doStartTag()` 中返回 `EVAL_BODY_BUFFERED` 将标记体内容保存到缓存中，在 `doEndTag()` 中调用 `bodyContent` 成员的 `getString()` 方法，得到缓存中的标记体内容；

2、 如何开发嵌套 JSP 标记？

创建嵌套标签时，标记处理类与普通标签相似，但在 `doStartTag()` 方法中必须返回 `EVAL_BODY_INCLUDE`，JSP 容器才会处理嵌套的子标记。

第十三章

1、 简述两种 JSP 设计模式的区别？

- JSP Model1——JSP+JavaBean，在 JSP 设计模式 1 中，没有用到 Servlet 组件，JSP 负责接收和处理用户传递过来的数据，JSP 借助 JavaBean 组件来完成在封装数据或与数据库打交道的操作，这种模式下 JSP 通常既要负责显示信息，还要负责处理业务逻辑。
- 在 JSP 设计模式 2 中，用到了三种 WEB 组件：jsp、servlet、javabeen，servlet 负责接收请求数据并处理整体业务流程，在处理业务过程中需要封装数据或与数据库打交道则调用 javabeen，然后根据 javabeen 的结果或 servlet 自己的处理结果选择合适的 JSP 页面传递给客户端浏览器。

2、 简述 MVC 设计模式各组成部分的功能？

MVC 将 WEB 应用分成三个核心模块：模型、视图、控制器，三个模块各负其责、相互协作：

➤ 视图

视图是用户直接看到并与之交互的界面。视图用于接收用户的输入数据和向用户显示数据，但视图中不进行任何业务处理。

➤ 模型

模型用于封装业务数据和业务逻辑，模型的主要作用是与数据库打交道以处理业务逻辑、向视图层传递数据。

➤ 控制器

控制器用以接收用户请求，并调用模型处理业务，然后根据模型的处理结果选择相应的视图显示到客户端。

3、 在 MVC 模式中，视图与模型之间是如何交换数据的？

控制器处理完业务后，如果有需要，可以将数据存储在某些作用域（request、session、application）内，然后转发到视图，视图 JSP 可以从这些作用域中读取数据以供显示。

在 JSP 中读取数据时必然要用到代码段，可以考虑将这些代码段封装到 JSP 自定义标签中，然后在 JSP 页中使用标签完成读取和显示操作。

第十四章

1、 DWR 框架解决的主要问题是什么？

DWR(Direct Web Remoting)是一个 WEB 远程调用框架，利用 DWR 可以在客户端利用 JavaScript 直接调用服务端的 Java 方法并返回值给 JavaScript，而不必由开发者自己编写 javascript 来操作 XMLHttpRequest 对象。即简少了开发者的编码量、减少了出错率。

2、 简述在 Java WEB 应用中使用使用 DWR 框架的步骤？

步骤如下：

- 1) 创建 WEB 站点
- 2) 将 DWR 支持库复制到网站 WEB-INF\lib 目录下
- 3) 在 WEB.XML 中配置 DWRServlet
- 4) 开发一个 JAVA 类，其中包含要被 Ajax 调用的方法
- 5) 在 WEB-INF 目录下创建名为 dwr.xml 的配置文件，在该文件中配置要共享出去的类
- 6) 编写页面通过 Javascript 调用 Java 类