Broadview
www.broadview.com.cn

·轻松学开发·

图解版

# 轻松学

# Java Web开发

张昆 编著

图解学编程，Java Web原来这么简单！

本书特点

◎775幅教学插图，轻松学习技术　◎208个典型示例，熟练掌握应用

◎603分钟视频，体验全新方式　◎44个课后题目，全面测试能力

随书DVD

603分钟全程视频·本书源代码·电子课件

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

## 本书知识要点

HTML
CSS
MyEclipse
JSP
JavaBean
Servlet
EL、JSTL
Struts 2
拦截器
类型转换
输入校验
国际化
文件上传
标签库
Hibernate
Spring
控制反转
面向切面编程
框架技术整合

## 本书所针对的读者群

- ☑ Java Web的初学者
- ☑ 网站开发爱好者
- ☑ 大中专院校的学生
- ☑ 社会培训班学员
- ☑ Java Web专业开发人员

# 轻松学 Java Web 开发

张　昆　编著

電子工業出版社
Publishing House of Electronics Industry
北京・BEIJING

## 内 容 简 介

本书由浅入深，全面、系统地介绍了 Java Web 开发技术。本书最大的特色就是提供了大量的插图，一改过去编程书籍枯燥乏味的文字讲解，利用各种说明插图和运行结果示意图，生动形象地再现了 Java Web 开发需要的所有知识，使读者能够轻松地掌握学习内容。另外，作者专门为每一章编写了一些习题，以便读者对该章的学习水平进行检测。本书还录制了大量的配套教学视频，这些视频和书中的实例源代码一起收录于本书的配套光盘中。

本书共分 5 篇。第 1 篇“JSP 基础篇”，主要包括浏览器技术、JSP 基础、JSP 内置对象、JavaBean 基础、Servlet 编程以及 EL 表达式语言和 JSTL 标签等知识。第 2 篇“Struts 2 技术篇”，主要内容包括 Struts 2 框架入门、Struts 2 配置详解、Struts 2 拦截器、Struts 2 类型转换和输入校验以及国际化和文件上传、标签库等内容。第 3 篇“Hibernate 技术篇”，主要包括 Hibernate 框架入门、Hibernate 的配置和会话等技术。第 4 篇“Spring 技术篇”，主要内容包括 Spring 框架入门、控制反转和面向切面编程等方面的知识。第 5 篇“S2SH 整合篇”，本篇主要实现了 3 种重要技术的整合开发。

本书涉及面广，从基本操作到高级技术和核心原理，再到项目开发，几乎涉及 Java Web 开发的所有重要知识。本书适合所有想全面学习 Java Web 开发技术的人员阅读，也适合各种希望使用 Java Web 3 大框架进行开发的工程技术人员使用。对于经常使用 Java Web 做开发的人员，更是一本不可多得的案头必备参考书。



**图书在版编目（CIP）数据**

轻松学 Java Web 开发 / 张昆编著. —北京：电子工业出版社，2013.3
（轻松学开发）
ISBN 978-7-121-19558-7

Ⅰ. ①轻… Ⅱ. ①张… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312

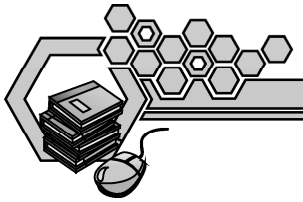
中国版本图书馆 CIP 数据核字（2013）第 026949 号

策划编辑：胡辛征
责任编辑：葛 娜 郑志宁
印 刷：北京中新伟业印刷有限公司
装 订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
北京市海淀区万寿路 173 信箱 邮编 100036
开 本：787×1092 1/16 印张：24.75 字数：609 千字
印 次：2013 年 3 月第 1 次印刷
印 数：4000 册 定价：49.00 元（含 DVD 光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。



# 前　言

随着互联网的发展，人们越来越认识到 Web 开发的重要性，越来越多的应用都是基于 Web 发展起来的，大到企业门户网站，小到网页游戏。Java Web 开发在整个 Web 开发领域一直占据着领头羊的位置，目前许多的 Web 应用将 Java Web 开发作为首选技术。目前，越来越多的企业开始注重对 Java Web 开发技术人才的吸收和培养，相信从事 Java Web 开发将是一个非常不错的选择。

笔者结合自己多年的 Java Web 开发经验和心得体会，花费了一年多的时间写作本书。希望各位读者能在本书的引领下跨入 Java Web 世界的大门，并成为一名开发高手。本书最大的特色就是结合大量的说明插图，全面、形象、系统、深入地介绍了 Java Web 开发程序，并以大量实例贯穿于全书的讲解之中，最后还详细介绍了 Struts 2、Hibernate 以及 Spring 三大框架的开发流程以及相互间的整合方法。学习完本书后，读者应该可以具备独立进行项目开发的能力。

## 本书特色

### 1．大量教学插图，读书学习不再枯燥乏味

本书最大的特点就是通篇采用图片讲解，将传统的文字讲解转换为各种形式的图形图表，最大限度的提升读者的阅读兴趣，让读者在潜移默化中掌握 Java Web 语言的开发精髓。

### 2．配有大量多媒体语音教学视频，体验全新教学课堂

作者专门录制了大量的配套多媒体语音教学视频，以便让读者更加轻松、直观地学习本书内容，提高学习效率。这些视频与本书源代码一起收录于配套光盘中。

### 3．讲解由浅入深，循序渐进，适合各个层次的读者阅读

本书从 Java Web 语言的基础开始讲解，逐步深入到 Java Web 语言的高级开发技术及应用，内容梯度从易到难，讲解由浅入深，循序渐进，适合各个层次的读者阅读，并均有所获。

### 4．贯穿大量的开发实例和技巧，迅速提升开发水平

本书在讲解知识点时贯穿了大量短小精悍的典型实例，并提供了大量的开发技巧，以便让读者更好地理解各种概念和开发技术，体验实际编程，迅速提高开发水平。

## 本书内容及体系结构

### 第 1 篇　JSP 基础篇（第 1～7 章）

本篇主要内容包括：浏览器技术、JSP 基础、JSP 内置对象、JavaBean 基础、Servlet 编程

以及 EL 表达式语言和 JSTL 标签等知识。通过本篇的学习，读者可以掌握 Java Web 开发所需要的最基本的知识。

**第 2 篇　Struts 2 技术篇（第 8～13 章）**

本篇主要内容包括：Struts 2 框架入门、Struts 2 配置详解、Struts 2 拦截器、Struts 2 类型转换和输入校验以及国际化和文件上传、标签库等内容。通过本篇的学习，读者可以掌握 Struts 2 编程的核心技术与应用。

**第 3 篇　Hibernate 技术篇（第 14～15 章）**

本篇主要内容包括：Hibernate 框架入门、Hibernate 的配置和会话等技术。通过本篇的学习，读者可以掌握 Hibernate 框架的一些基本的开发技术。

**第 4 篇　Spring 技术篇（第 16～18 章）**

本篇主要内容包括：Spring 框架入门、控制反转和面向切面编程等方面的知识。通过本篇的学习，读者可以掌握 Spring 框架的基础以及两大核心技术及应用。

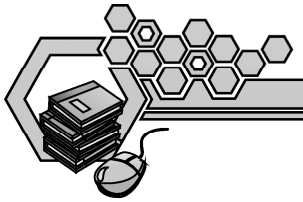
**第 5 篇　S2SH 整合篇（第 19 章）**

本篇是三大框架的整合章节，通过本章的学习，读者可以实现 3 种重要技术的整合开发，充分体验 S2SH 的强大功能和便利性。

# 本书读者对象

- 从未接触过 Java Web 的初学者；
- 了解一些 Java Web，希望进一步学习的自学者；
- 想学习一门技术，以方便找工作的求职者；
- Java Web 开发爱好者；
- 大中专院校的学生和相关授课教师；
- 社会培训班学员；
- Java Web 专业开发人员；
- 需要一本案头必备手册的程序员。

编　者



# 目　录

## JSP 基础篇

## Struts 2 技术篇

## Hibernate 技术篇

## Spring 技术篇

## S2SH 整合篇

PART 1

# JSP 基础篇

# 第 1 章　浏览器技术

如今随着互联网技术的飞速发展，人们对网络的依赖不断加深。其中，浏览器更是一个连接 Internet 的主要工具。我们熟悉的浏览器包括微软的 IE、Mozilla 的 FireFox、Opera 等。那么浏览器真正的作用是什么呢？它是如何工作的呢？官方的解释是这样定义浏览器的：万维网（Web）服务的客户端浏览程序，可向万维网（Web）服务器发送各种请求，并对从服务器发来的超文本信息和各种多媒体数据格式进行解释、显示和播放。下面我们就来详细地了解它。

## 1.1　HTTP 协议

我们通过浏览器在互联网上浏览新闻、看电影、购物等，这些行为看似是顺理成章的事，但其实，这一切的行为都是浏览器通过与远在各地的 Web 服务器进行交互而实现的。为了交互的进行，它们需要共同遵守一定的协议来控制。这就是 HTTP（Hypertext Transport Protocol），超文本传输协议，一种详细规定了浏览器和 Web 服务器之间互相通信的规则，通过互联网传送万维网文档的数据传送协议。

### 1.1.1　HTTP 协议原理

HTTP 协议是一种通信协议。它允许将 HTML（超文本标记语言）从 Web 服务器传送到 Web 浏览器，因此需要 Web 服务器和 Web 浏览器都支持该协议。它的具体请求、响应格式如图 1.1 所示。

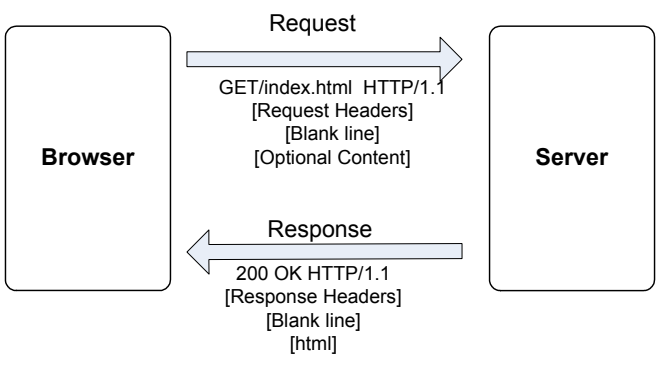



图 1.1　HTTP 协议的请求、响应格式

当浏览器向 Web 服务器发送一个请求，Web 服务器在接收到这个请求后，会返回一个响应给浏览器。这个请求包含一个请求页面的名字和请求页面的信息等。返回的响应包含被请求的页面和被请求页面的信息以及服务器的一些信息等。从图 1.1 我们也可以看到，浏览器发送这个请求的时候，依据的是 HTTP/1.1 的格式，因此在返回响应的时候，服务器也必须按照 HTTP/1.1 的协议格式来响应。

## 1.1.2　HTTP 请求格式

HTTP 协议对浏览器所发出的 Request 格式有如下三部分的规定。

- 第一部分是 Request line。它包括请求的方法、所请求资源的名字以及现在所使用的协议。
- 第二部分是 Request headers。它包含浏览器的一些信息。
- 第三部分是 Request body。其中 Request headers 与 Request body 之间有一个空行。

具体结构如图 1.2 所示。

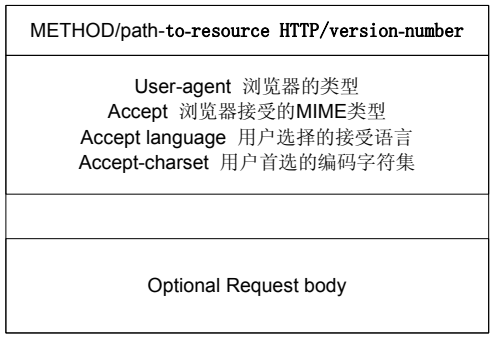



图 1.2　Request 请求的结构

其中，METHOD 表示请求的方法，如“POST”、“GET”。path-to-resource 表示请求的资源。HTTP/version-number 表示 HTTP 协议的版本号。

**【示例 1.1】**我们借用 Fiddler 工具（可以记录客户端与服务器端的 HTTP 请求与响应信息）来捕捉一个向百度首页所发出 Request 的具体格式及内容，如图 1.3 所示。

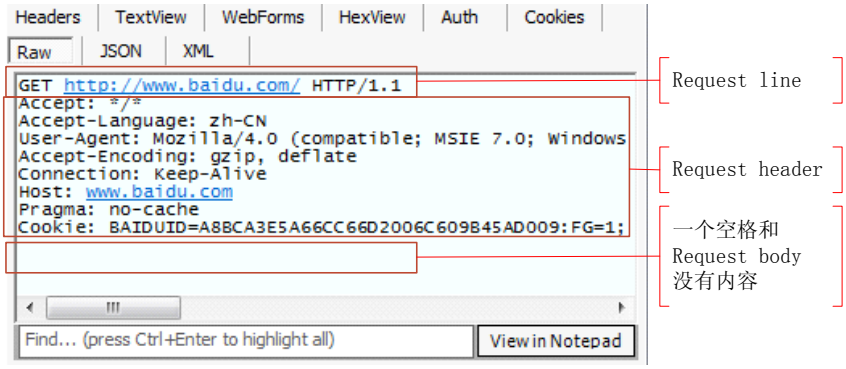



图 1.3　Fiddler 捕捉的浏览器向百度首页发出的 Request 内容

## 1.1.3　HTTP 响应格式

HTTP 协议对 Web 服务器所返回的 Response 也有具体的格式规定。和 Request 一样，

Response 也分为三部分。

- 第一部分是 Response line。它包含 HTTP 协议的版本信息，响应状态等。
- 第二部分是 Response header。它包括服务器的一些基本信息。
- 第三部分是 Response body。Response header 与 Response body 之间也有一个空行。

具体结构如图 1.4 所示。

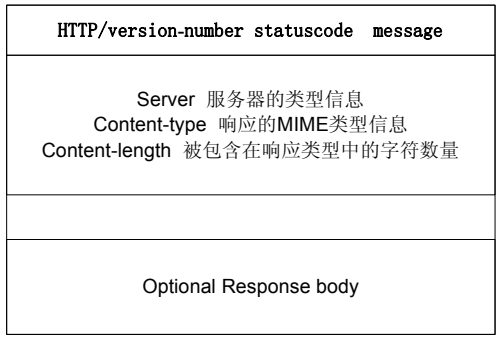



图 1.4　Response 响应的结构

其中，HTTP/version-number 表示 HTTP 协议的版本号。statuscode 表示服务器返回的状态码。message 表示服务器返回的状态消息。

**【示例 1.2】**查看通过 Fiddler 工具请求百度首页后返回的 Response 的信息格式和内容，如图 1.5 所示。

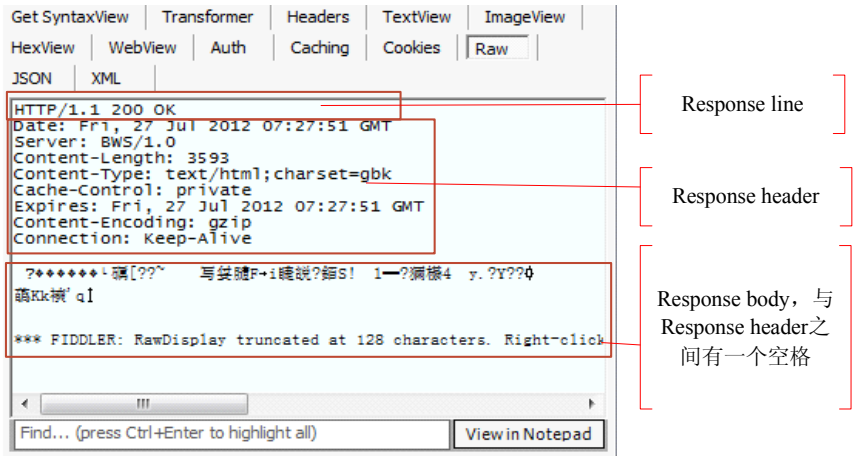



图 1.5　Fiddler 截取的百度首页 Response 的信息内容

**注意：**这里返回的状态码是 200，状态信息是 OK。表示服务器响应成功，请求被成功地完成，所请求的资源被发送到客户端。

### 1.1.4　Content type

服务器在接收到请求后，必须能识别要发送的信息类型，比如图片、txt 文本、Excel 表格或者其他的形式。还需要知道网页的编码方式是什么。因此，Content type 就是用于定义网络

文件的类型以及网页字符的编码，用于决定浏览器以什么形式、什么编码读取这个文件。

### 1．MIME 类型

MIME（Multipurpose Internet Mail Extensions），即多功能 Internet 邮件扩充服务。它是一种多用途网际邮件扩充协议，服务器会通过这种手段来告诉浏览器它所发送的这些多媒体数据是什么类型的，需要用何种程序来打开这种文件。最常用的 MIME 类型如表 1.1 所示。

表 1.1　常用的MIME类型

| 名　　称 | MIME 类型 |
|---|---|
| 超文本标记语言 | hext/html |
| 普通文本 | text/plain |
| Microsoft Word | application/msword |
| PDF 文档 | application/pdf |
| AVI 文件 | video/x-msvideo |

### 2．Content-charset

字符的编码方式有很多种。有的支持中文显示，有的支持英文显示。其中最常见的字符集编码类型如表 1.2 所示。

表 1.2　常见的charset字符集类型

| charset 类型 | 字符集编码类型 |
|---|---|
| ISO-8859-1 | 拉丁语系 1 |
| Big5 | 繁体中文 |
| UTF-8 | 通用子集转化格式（8 位） |
| ISO-2022-JP | 日本语 |
| ISO-2022-KR | 韩国语 |
| GBK | 简体中文（兼容 GB2312） |
| GB2312 | 汉字国标码 |

## 1.2　HTML

HTML（Hypertext Markup Language），即超文本标记语言，是用于描述网页文档的一种标记语言。它是一种规范，一种标准，通过标记符号来标记要显示网页的各个部分。任何动态语言都离不开 HTML 的支持。所以在学习 Web 开发之前，读者首先要掌握 HTML 的相关基础知识。

### 1.2.1　标记语言

标记语言，也称为置标语言。是将文本以及文本相关的其他信息结合起来，展现出文档的结构和数据处理细节的计算机文字编码。

标记语言根据使用范围的不同有很多类型。例如，用于描述用户网页信息的是超文本标记语言（HTML），用于描述用户计算机处理各种信息的是可扩展的标记语言（XML）等。由于类型的不同，它们各自的语法也各不相同。下面我们重点介绍的是用于搭建网页的超文本标记

语言，即 HTML。HTML 的大致结构如图 1.6 所示。

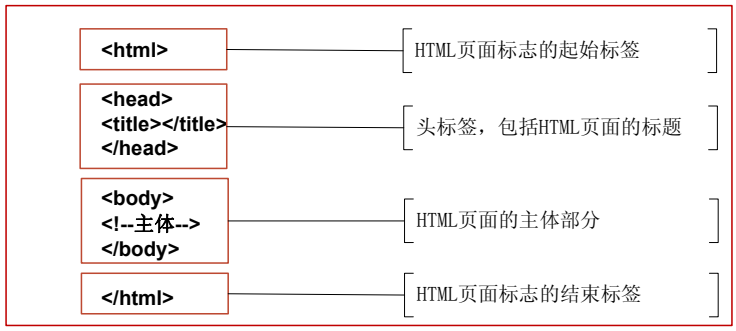



图 1.6　HTML 页面的结构

**【示例 1.3】**我们按照 HTML 文档的结构举一个最简单的使用 HTML 语言的例子 First.html，如图 1.7 所示。由于目前还没有使用开发工具，所以我们首先将代码写在 txt 文件中，然后将文件后缀名改为 html 即可在浏览器中运行。

```
<html>
  <head>
    <title>这是第一个HTML例子</title>
  </head>
  <body>
    欢迎光临！这是我的第一个HTML文档。<br/>
  </body>
</html>
```

图 1.7　第一个 HTML 文档示例

我们在地址栏中输入文件的存储地址，就可以运行这个 html 文件，运行结果如图 1.8 所示。

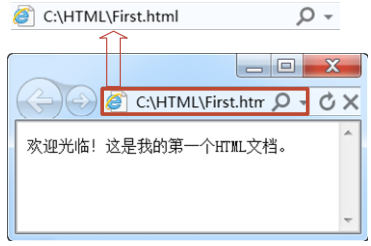



图 1.8　First.html 运行结果

**注意：**一般情况下，可以包括其他内容的 HTML 标签都是成对出现的，例如上面例子中的<title></title>这对标签，它包含了一个文字的标题信息，所以成对出现。而<br/>这样的标签仅仅是一个回车换行的作用，它不包含其他内容，所以不成对出现。

### 1.2.2　超链接

在 HTML 中实现页面的跳转的主要方式是用超链接的形式。超链接的用途非常广泛，可以是一个字、一个词，或者一组词，也可以是一幅图像。用户可以单击这些内容跳转到新的文档或者当前文档中的某个部分。例如我们打开百度首页，如图 1.9 所示。它的每个词汇单击后都可以链接到一个新的页面，甚至在单击首页中图片的时候也可以打开一个新的页面，这都是超链接的应用。


图 1.9　百度主页

在 HTML 中超链接的标签是<a></a>，它的语法格式如图 1.10 所示。

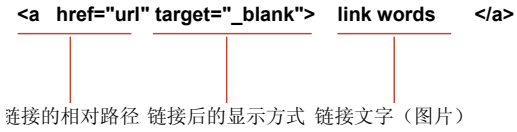



图 1.10　超链接语法格式

**【示例 1.4】**我们将这个语法格式加以应用，就可以得到 HTML 超链接页面 link.html，如图 1.11 所示。

```
<html>
 <head>
   <title>超链接示例</title>
 </head>
 <body>
   <a href="http://www.baidu.com" target="_blank">百度</a>
 </body>
</html>
```

图 1.11　link.html 示例

在浏览器地址栏中输入地址 C:\HTML\link.html，显示如图 1.12 所示的页面。

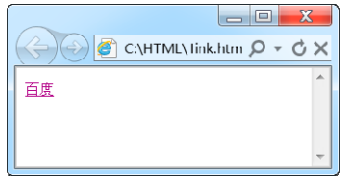



图 1.12　link.html 运行结果

单击“百度”超链接，我们就会进入百度主页，如图 1.13 所示。


图 1.13　百度主页

## 1.2.3　静态页面

当一个网页页面仅仅由 HTML 语言代码组织起来，那么这个页面就是一个静态的页面。它不会与数据库、服务器进行交互，只能通过浏览器进行显示，是一个独立存在的文件。现今我们通过浏览器浏览互联网的时候，所看到的大部分网站都不是静态网站。因为它的显示单一，功能简单，所以已经被动态网站所取代。

**【示例 1.5】**我们创建一个静态的网页 static.html，并在浏览器上显示，如图 1.14 所示。

```
<html>
  <head>
    <title>静态页面</title>
  </head>
  <body>
    <strong>这是一个静态页面，仅由HTML语言组织起来。</strong><br>这是第二行。
  </body>
</html>
```

图 1.14　static.html 示例

在浏览器中输入 HTML 文件地址，显示静态页面，如图 1.15 所示。

与静态网站相对应的是动态网站，所谓的动态网站，就是用户可以通过浏览器与服务器端进行数据交互的网站。例如国内的搜索网站百度，如图 1.16 所示。

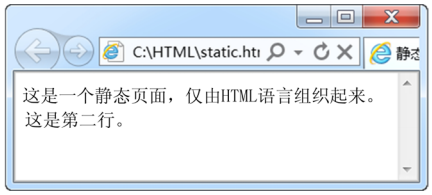



图 1.15　静态页面显示结果

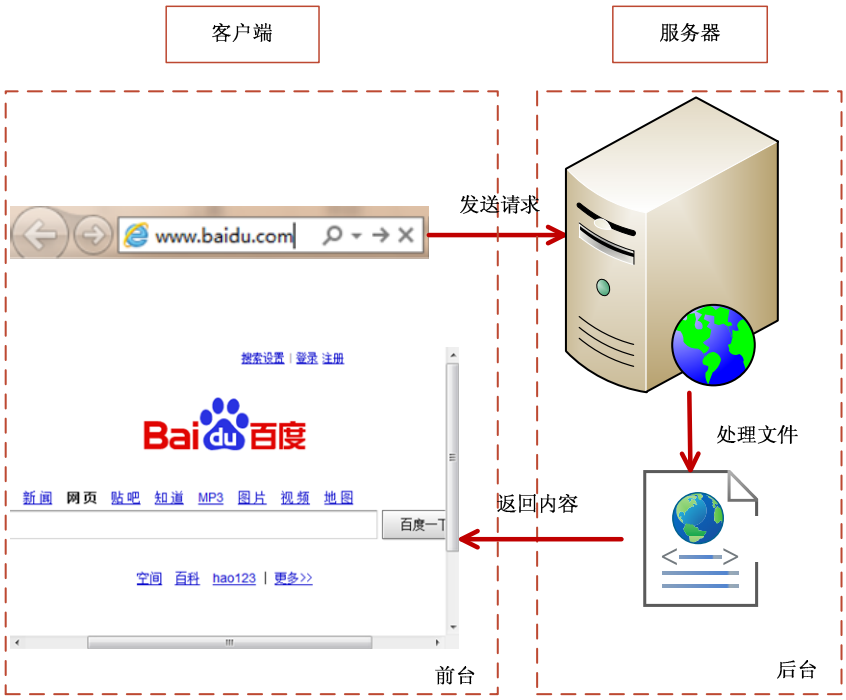



图 1.16　动态搜索网站百度的交互过程

动态网站不只是由 HTML 语言组织起来的，它还是由 HTML 语言与其他的脚本语言共同组织起来的，例如 HTML+ASP、HTML+PHP、HTML+JSP 等。HTML 语言起到一个显示的作用，至于交互的动作及过程，都是由脚本语言完成的。有关动态网站的内容我们会在后面章节中为大家讲解。

## 1.3　HTML 常用表单标签

在 HTML 的布局标签中，表单标签是使用频率最高的一个。它可以把一组信息用表格的形式表示出来。这一节我们就来看表单元素是如何实现的。

### 1.3.1　表单元素

在 HTML 的标签库中有<form></form>这样一个成对标签，它用于向一个目标地址提交一些数据。在这个标签中，我们可以设计文本框、单选框、复选框和按钮等一些元素用于获取数据，这些元素都称为表单元素。下面我们介绍几个最常用的表单元素。

### 1．文本框

在 HTML 页面中最常用的就是一个单行的文本框了。语法格式如下所示：

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

在浏览器上的显示效果如图 1.17 所示。

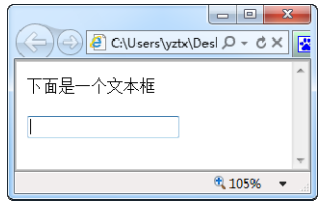



图 1.17　文本框

### 2．单选框

单选框就是在并列的几个值中只能选择一项。对于一组元素，必须保证它们的 name 属性值相同。语法格式如下所示：

```
<input type="radio" value="值 1" name="dxk">
<input type="radio" value="值 2" name="dxk">
```

在浏览器上的显示效果如图 1.18 所示。

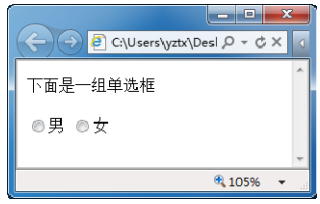



图 1.18　单选框

### 3．复选框

当用户需要从若干给定的选择中选取一个或若干选项时，就会用到复选框。对于同一组值，必须要保证它们的 name 值相同。语法格式如下所示：

```
<input type="checkbox"value="值 1"name="dxk">
<input type="checkbox"value="值 2"name="dxk">
```

在浏览器上的显示效果如图 1.19 所示。

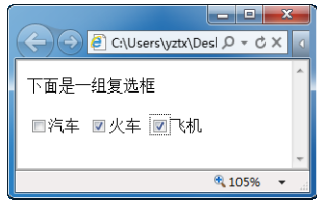



图 1.19　复选框

### 4．提交按钮

当表单要提交时，单击此按钮，表单就会将表单中所选数据提交到目的地址。语法格式如

下所示：

```
<input type="submit"value="提交">
```

在浏览器上的显示效果如图 1.20 所示。

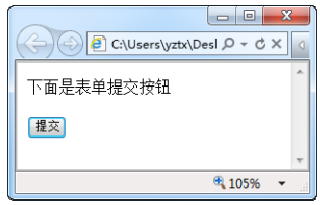



图 1.20　提交按钮

除了上述介绍的几个元素外，表单还有其他很多的元素，例如重置按钮、密码框和下拉列表等。

**【示例 1.6】**用表单元素建立一个用于用户注册的 HTML 页面 register.html，代码如图 1.21 所示。

```
<html>
  <title>注册页面
  </title>
  <body>
     <h1>用户注册</h1>
     <form>
  请输入昵称：<input type="text" name="name"><br>
  请选择性别：<input type="radio" name="radio" value="male">男
     <input type="radio" name="radio" value="female">女<br>
  请选择你所在的城市：
     <select name="city">
     <option value="bj">北京</option>
     <option value="sh">上海</option>
     <option value="dl">大连</option>
     </select><br>
     请选择你最喜欢的运动：
     <input type="checkbox" name="sport" value="swimming">游泳
     <input type="checkbox" name="sport" value="basketball">篮球
     <input type="checkbox" name="sport" value="football">足球<br>
     <input type="submit" value="提交">
     <input type="reset" value="重置">
     </form>
</body>
</html>
```

运行结果

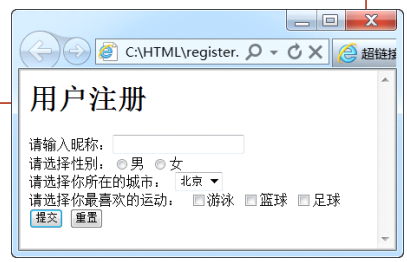



图 1.21　register.html 示例

### 1.3.2 表单元素的属性

在这些元素中，有一些共同的属性，即 type、name 和 value。那么这三个属性都有什么作用呢？

- type 属性：表示该元素的类型。有 text（文本框）、checkbox（单选框）和 button（按钮）等值。
- name 属性：表示该元素的名称，只能有唯一值。
- value 属性：设置该元素的默认值。

下面我们通过一个示例来了解一下这三个属性的作用。

**【示例 1.7】**在表单中将文本框的值通过按钮提交。新建一个 HTML 文件 tijiao.html，代码内容如图 1.22 所示。

```
<html>
  <body>
  <form>
  <input type="text" name="hobby">
  <input type="submit" value="botton" name="tijiao">
  </form>
  </body>
</html>
```

图 1.22　tijiao.html 示例

用浏览器打开后如图 1.23 所示。

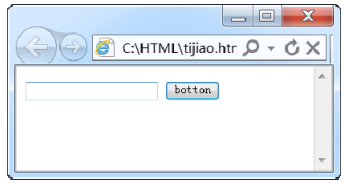



图 1.23　HTML 提交文件

我们在文本框中输入值：test，单击“botton”按钮。这时我们在浏览器的地址栏中会发现如图 1.24 所示的结果。

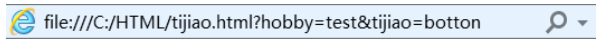



图 1.24　提交表单后浏览器地址栏

与提交表单之前相比，字符串后面多了“?hobby=test&tijiao=botton”，这表示，在我们提交表单后，表单会将表单元素中文本框和按钮属性的值以“key=value”的形式附加到地址字符串的后面，之间用符号“&”隔开。

### 1.3.3 表单中添加目的地址

在示例 1.6 中我们可以看到，在 form 表单中并没有目的地址属性。表单提交仅仅是包含着

数据但并没有提交到某个页面。在这里我们在 form 中添加 action 这个属性。这个属性的作用就是表示表单要提交的目的地址。语法格式如图 1.25 所示。

```
<form action="url"> </form>
```

表单提交地址的相对路径

图 1.25　表单提交目的地址的语法格式

**【示例 1.8】**在示例 1.7 的基础上，新建一个 HTML 页面 action.html，并且添加 action 属性，其详细代码如图 1.26 所示。

```
<html>
  <body>
    这是第二个页面。
  </body>
</html>
```

图 1.26　action.html 示例

然后我们对 tijiao.html 进行修改，修改后代码 tijiao2.html 如图 1.27 所示。

```
<html>
 <body>
 <form action="action.html">
 <input type="text" name="hobby">
 <input type="submit" value="botton" name="tijiao">
 </form>
 </body>
</html>
```

图 1.27　tijiao2.html 示例

将 tijiao2.html 与 action.html 文件放入同一级目录下。打开 tijiao2.html，单击“botton”按钮。运行后结果如图 1.28 和图 1.29 所示。

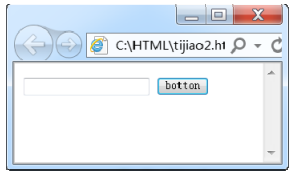



图 1.28　运行 tijiao2.html 并输入

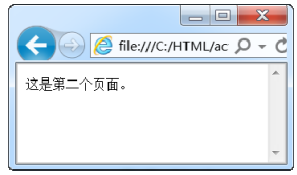



图 1.29　单击按钮后的结果

## 1.3.4　表单中添加数据的提交方式

当我们的数据可以提交到目的地址后，疑问也随之产生，就是数据是以何种方式提交的呢？是通过浏览器的地址栏中的附加到地址字符串方式提交的吗？还是有其他的方式？

form 表单中还有一个属性 method。它表示表单中数据的提交方式。它有两个可选值：POST

和 GET。

- POST：将数据打包，以隐含的方式传递。
- GET：附加到 URL 上，通过 URL 来传递数据。

示例 1.7 与 1.8 表示，虽然没有 method 属性，但是浏览器会默认的用 GET 方式进行传递。

**【示例 1.9】**将示例 1.8 的 tijiao2.html 做修改，添加 method 属性为 POST。修改后的代码如图 1.30 所示。

```
<html>
  <body>
  <form action="action.html" method="POST">
  <input type="text" name="hobby">
  <input type="submit" value="botton" name="tijiao">
  </form>
  </body>
</html>
```

图 1.30　修改后的 tijiao2.html 代码

运行后的结果如图 1.31 所示。

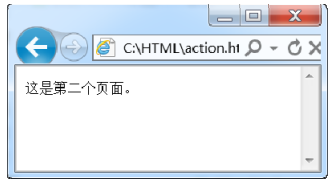



图 1.31　以 POST 方式提交表单数据

我们可以对比一下两次地址栏中的地址信息，如图 1.32 所示。


图 1.32　两种传递方式地址栏的区别

我们可以发现，浏览器的地址栏中不再有附加的字符串信息了。这表明 POST 传递方式是以不同于 GET 的方式传递数据的。

## 1.4　CSS 基础

在前面的内容中讲解了 HTML，现在我们已经基本可以编出最简单的网页了。但是仅仅这样还是不够的，一个专业的网页需要在字体、颜色、布局等方面进行各种设置，需要给用户带来视觉的冲击。这一节我们简要介绍一下页面美化技术——CSS。

### 1.4.1　CSS 属性设置

CSS（Cascading Style Sheets）即层叠样式表，也就是通常所说的样式表。CSS 是一种美化网页的技术。通过使用 CSS，可以方便、灵活地设置网页中不同元素的外观属性，通过这些设

置可以使网页在外观上达到一个更高的级别。CSS 美化网页就是通过设置页面元素的属性来实现的，下面我们就来介绍 CSS 属性设置的一些基本方法。

### 1．字体、颜色、背景等属性设置

字体、颜色、背景属性是 CSS 最基本的属性，其最常用的属性方法如表 1.3 所示。

表 1.3　CSS属性设置

| 属 性 方 法 | 方 法 作 用 |
|---|---|
| font-family | 定义使用哪种字体 |
| font-style | 定义是否使用斜体 |
| font-weight | 定义字体的粗细 |
| font-size | 定义字体的大小 |
| background-color | 定义背景颜色 |
| background-image | 定义背景图片 |
| background-attacement | 定义滚动 |
| background-position | 定义背景图片的初始位置 |

**【示例 1.10】**下面通过一个示例程序 FontColor.html 展示这些属性的设置方法，如图 1.33 所示。

```
<html>
    <head>
       <title>字体、颜色、背景属性设置示例</title>
    </head>
    <body>
     <div align="left">字体属性设置：</div>
     <span style="font-family:幼圆;font-style:italic;
     font-weight:bold;font-size:10pt;">幼圆、斜体、黑体、10pt</span><br>
     <span style="font-family:隶书;font-size:16pt;">隶书、黑体、16pt</span><br>
     <div align="left">字体属性设置：</div>
     <span style="color:red;background-color:yellow;">
      前景红色、背景黄色</span><br>
      <span style="background-image:url('image.jpg');
       background-repeat:no-repeat;">    
      图片背景、不重复</span><br>
    </body>
</html>
```

图 1.33　字体、颜色、背景属性设置方法

运行后结果如图 1.34 所示。

### 2．鼠标样式属性设置

在一些网页中，我们经常会遇到这样一种情况，当把鼠标移到不同区域，或者是在执行不同功能的时候，鼠标的形状都会发生变化。这种功能的实现其实非常简单，就是控制 CSS 中的 cursor 属性来实现的。


图 1.34　FontColor.html 运行结果

【示例 1.11】我们来看一个示例 cursor.html，通过这个页面可以改变鼠标的形状，具体代码如图 1.35 所示。

```
<html>
    <head>
       <title>设置鼠标样式</title>
    </head>
    <body>
       <div style="font-family:宋体;font-size:20pt;">
        <span style="cursor:hand">手形状</span><br>
        <span style="cursor:move">移动</span><br>
        <span style="cursor:ne-resize">反方向</span><br>
        <span style="cursor:wait">等待</span><br>
        <span style="cursor:help">求助</span><br>
        <span style="cursor:text">文本</span><br>
        <span style="cursor:crosshair">十字</span><br>
        <span style="cursor:s-resize">箭头朝下</span>
        </div>
    </body>
</html>
```

图 1.35　cursor.html 示例

运行后结果如图 1.36 所示。

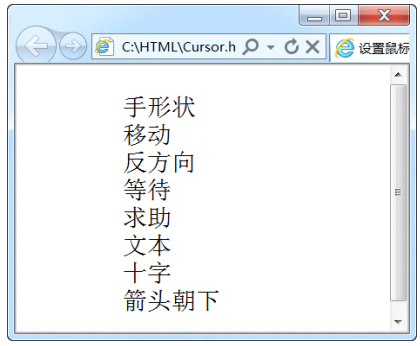



图 1.36　cursor.html 示例运行结果

上面这个网页在浏览器中打开以后，鼠标移动到不同区域就会变成不同样式。

## 1.4.2　CSS 绝对定位

在 HTML 中布局一般情况下需要使用表格，如果要定位只有通过表格的嵌套来实现，这种方法的确可以在一定程度上解决问题，但是却不能随意定位页面元素，而且对某个元素位置的改变有可能影响到整个页面的布局。在 CSS 中提供了灵活的定位方法，所以在页面布局中我们又多了一种可以选择的方案。

CSS 中常用的定位属性如表 1.4 所示。

表 1.4　CSS常用定位属性

| 属　性 | 作　用 | 属　性 | 作　用 |
|---|---|---|---|
| absolute | 绝对定位 | width | 定义宽 |
| relative | 相对定位 | height | 定义高 |
| static | 静态定位 | overflow | 定义内容超出的处理方法 |
| left | 定义横坐标 | z-index | 定义立体效果 |
| top | 定义纵坐标 | visibility | 定义可见性 |

**【示例 1.12】**我们也来举一个例子 Locate.html 来实现 CSS 绝对定位，其具体代码如图 1.37 所示。

```
<html>
 <head>
   <title>CSS定位示例</title>
 </head>
 <body>
   <div style="position:absolute;left=100;top=20;visibility=visible">
     下面这个图片是可见的：
       <img src="Image.gif"/>
  </div>
  <div style="position:absolute;left=100;top=60;visibility=hidden">
     下面这个图片是不可见的：
       <img src="Image.gif"/>
   </div>
 </body>
</html>
```

图 1.37　Locate.html 示例

运行后结果如图 1.38 所示。


图 1.38　Locate.html 示例运行结果

### 1.4.3 CSS 实现表格变色

在一些 Web 应用中经常会用表格来展示数据，当表格行数比较多的时候，就容易有看错行的情况发生，所以需要一种方法来解决这个问题。在这里我们采取这样一种措施，当鼠标移到某一行的时候，这行的背景颜色发生变化，这样当前行就会比较突出，不容易出错。

**【示例 1.13】**我们最后来举一个例子 colorTable.html，来看 CSS 是如何实现表格变色的，具体代码如图 1.39 所示。

```
<html>
   <head>
       <title>变色表格示例</title>
          <script language="javascript">
            function changeColor(row)
              { document.getElementById(row).style.backgroundColor='#CCCCFF';
                }
              function resetColor(row)
              { document.getElementById(row).style.backgroundColor='';
               }
                  </script>
   </head>
   <body>
   <table width="200" border="1" cellspacing="1" cellpadding="1">
    <tr>
       <th>学校</th>        <th >专业</th>       <th>人数</th>
    </tr>
    <tr align="center" id="row1" onMouseOver="changeColor('row1')"
         onMouseOut="resetColor('row1')">
       <td>北大</td>      <td>法律</td>    <td>2000</td>
    </tr>
   <tr align="center" id="row2" onMouseOver="changeColor('row2')"
        onMouseOut="resetColor('row2')">
      <td>清华</td>       <td>计算机</td>      <td>5000</td>
   </tr>
         <tr align="center" id="row3" onMouseOver="changeColor('row3')"
         onMouseOut="resetColor('row3')">
      <td>人大</td>       <td>经济</td>      <td>6000</td>
   </tr>
</table>
</body>
</html>
```

图 1.39　colorTable.html 示例

colorTable.html 示例运行后结果如图 1.40 所示。

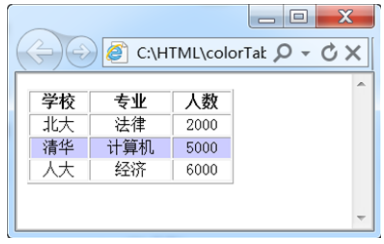



图 1.40　colorTable.html 示例运行结果

## 1.5　小结

本章主要讲解了与浏览器相关的技术，包括 HTTP 协议、HTML、HTML 中用于提交数据的表单以及 CSS 技术的基础知识。重点是在理解 HTTP 的请求与响应的交互原理以及表单中各个元素的用法。难点在于 HTML 与 CSS 技术的熟练运用。在接下来的章节中，我们将开始 JSP、Servlet 等关键技术的讲解，这也是本书的重点。

## 1.6　本章习题

1．请读者参照示例 1.1 编写一个 HTML 程序，运行后在浏览器中输出“I love Java Web!”。运行结果如图 1.41 所示。

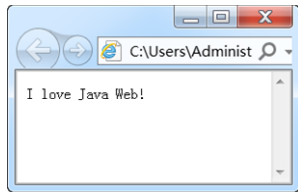



图 1.41　运行结果

【分析】本题非常简单，只是对 HTML 语言最基本的应用。我们参照示例 1.1 就可以很容易地将结果输出出来。

【核心代码】本题的关键代码如下所示。

```
<html>
 <head>
   <title>这是第一个 HTML 习题例子</title>
 </head>
 <body>
   I love Java Web!<br/>
 </body>
</html>
```

2．请读者参照示例 1.6 运用表单元素建立一个用于用户注册的 HTML 页面 register2.html，本页面中包括用户姓名，用户性别，用户注册密码以及用户地址（选择方式，包括北京、上海、广州）等选项。执行效果如图 1.42 所示。


图 1.42　运行结果

【分析】本题主要考查 HTML 常用表单标签的运用。这四种选项框分别对应不同的标签，

读者只要参考示例 1.6 对其进行适当的修改，就很容易得到需要执行的效果图。

【核心代码】本题的关键代码如下所示。

```
<html>
  <title>注册页面
  </title>
  <body>
     <h1>用户注册</h1>
     <form>
   用户姓名：<input type="text" name="name"><br>
   性别：<input type="radio" name="radio" value="male">男
      <input type="radio" name="radio" value="female">女<br>
   注册密码：<input type="text" password="password"><br>
   请选择你所在的城市：
     <select name="city">
     <option value="bj">北京</option>
     <option value="sh">上海</option>
     <option value="gz">广州</option>
     </select><br>
     <input type="submit" value="提交">
     <input type="reset" value="重置">
     </form>
</body>
</html>
```

3．使用 CSS 美化网页的技术，创建一个 HTML 页面。要求标题为绿色，文本为红色。再设置一行蓝色字体，黄色背景的文字。执行效果如图 1.43 所示。

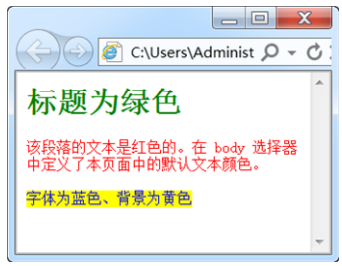



图 1.43　运行结果

【分析】本题主要考查 CSS 基础的运用。在实际操作中我们经常会遇到字体颜色的处理，通过本题读者可以参照 CSS 属性设置的方法很容易地实现页面中各部分字体颜色的设置。

【核心代码】本题的关键代码如下所示。

```
<html>
<head>
<style type="text/css">
body {color:red}
h1 {color:green}
</style>
</head>
<body>
<h1>……</h1>
<p>……</p>
<span style="color:blue;background-color:yellow;">……</span><br>
</body>
</html>
```

# 第2章 JSP基础

上一章我们主要讲解了从浏览器端用表单提交数据，数据通过 POST 或者 GET 的方式提交到服务器端。那么在服务器端是通过什么技术来获取这些数据的呢？并且对这些数据是进行怎么处理的呢？接下来在本章中，我们主要讲解的是在服务器端获取和处理数据的技术——JSP（Java Server Pages）。

## 2.1 JSP与服务器

客户端通过表单将数据提交到 action 指定的目的地址。在这个目的地址指向的页面，需要将数据提取出来。这就需要一个动作脚本来完成动态网页技术中的数据交互。这种动作脚本与 HTML 语言相结合来获取和处理表单提交的数据。在 Java Web 中，这种用于服务器端处理数据的动作脚本就是 JSP。

JSP 是 Java Server Pages 的简称，它是在传统的 HTML 文件中插入 Java 程序段和 JSP 标记，形成的 JSP（.jsp）文件。它是一种动态网页技术，遵从动态网页的技术标准。

### 2.1.1 JSP在服务器上的工作原理

JSP 文件是运行在服务器端的脚本文件，它由 HTML 语言、Java 代码和一些独特的 JSP 标记组成。由于它包含了 Java 程序段，所示它需要被服务器编译才能运行。我们知道 JSP 页面被部署在 Web 服务器或应用服务器上。整个 JSP 工作机制如图 2.1 所示。

服务器管理 JSP 页面分为两个阶段：转换阶段和执行阶段。

（1）当有一个 JSP 请求到来时，服务器会首先检验 JSP 页面的语法是否正确，将 JSP 转换成 Servlet（Servlet 就是用 Java 语言实现的 CGI 程序，后面章节将详细介绍）源文件，然后调用 javac 工具类编译 Servlet 源文件生成 .class 文件，这就是转化阶段。

（2）Servlet 容器加载转化后的 Servlet 类，实例化一个对象处理客户端的请求。在请求处理完成后，响应对象被服务器接受，服务器将 HTML 格式的响应信息发送给客户端，这一阶段便是执行阶段。

JSP 页面的第一次执行要花费一些时间，去完成 JSP 页面到 Servlet 的转换。当再次请求时，JSP 服务器就会直接执行第一次请求时产生的 Servlet，而不再进行 JSP 文件的转换。

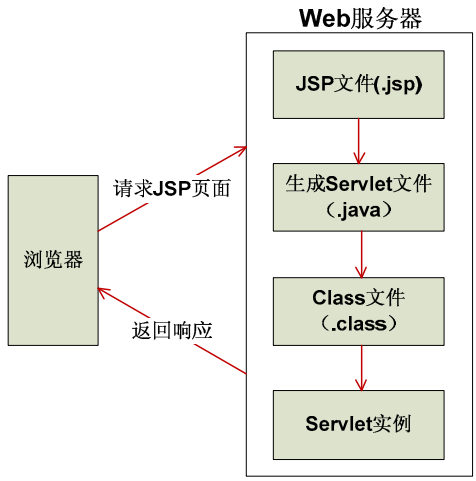



图 2.1　JSP 工作机制

## 2.1.2　Web 服务器 Tomcat 的搭建

JSP 页面必须被部署和运行于 Web 服务器中，所谓的 Web 服务器是指为特定组件提供服务的一个标准化的运行时的环境，其中封装了 JSP 运行所需要的底层 API，为组件提供事务处理，数据访问，安全性，持久性等服务。

Web 的服务器有很多种，其中 Tomcat 就是一个免费并且开源的 JSP 服务器。它是 Apache 软件基金会 Jakarta 项目中的一个核心项目。它由 Apache、Sun 和其他一些公司及个人共同开发而成。目前它是使用最广泛的 JSP 服务器。到目前为止，它的最新版本是 Tomcat 7.0。读者可以从 Apache Tomcat 官方网站（http://tomcat.apache.org/）上选择下载 Tomcat 7.0，如图 2.2 所示。

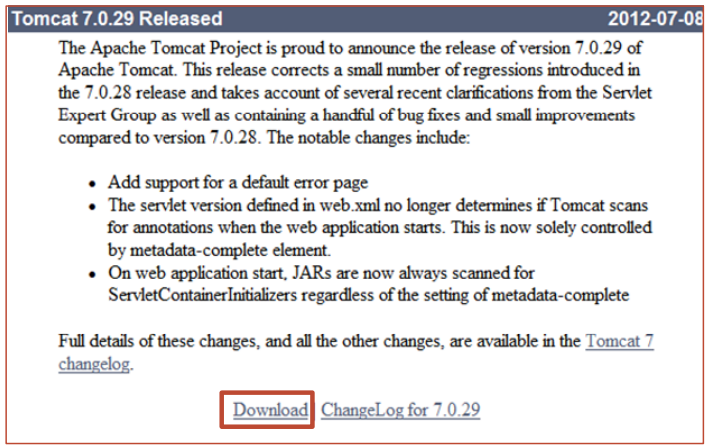



图 2.2　Tomcat 下载版本

**注意：**Tomcat 7.0.29 是笔者写作时的最新版本。读者下载时，可能是更新的版本。但是只要大版本号一致，就不会影响读者学习。

在单击 Download 之后，我们进入具体的版本选择页面，在这里选择“32-bit/64-bit Windows Service Installer”版本进行下载，如图 2.3 所示。

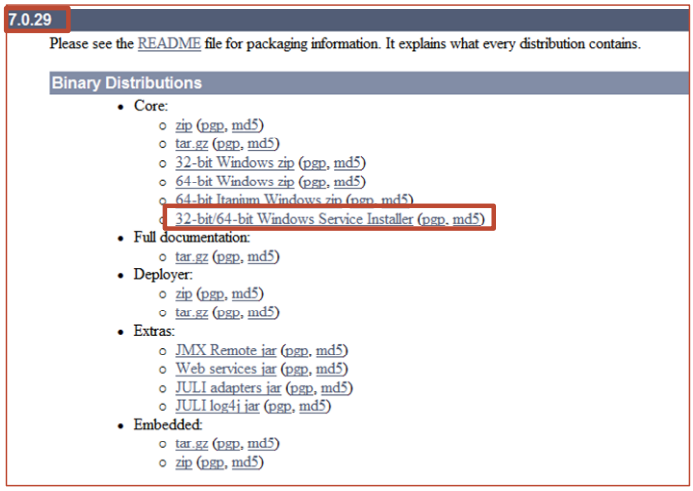



图 2.3　Tomcat 具体版本下载

下载完成后，双击下载的 Tomcat 安装文件 apache-tomcat-7.0.29.exe，弹出安装对话框。整个安装过程如图 2.4 和图 2.5 所示。

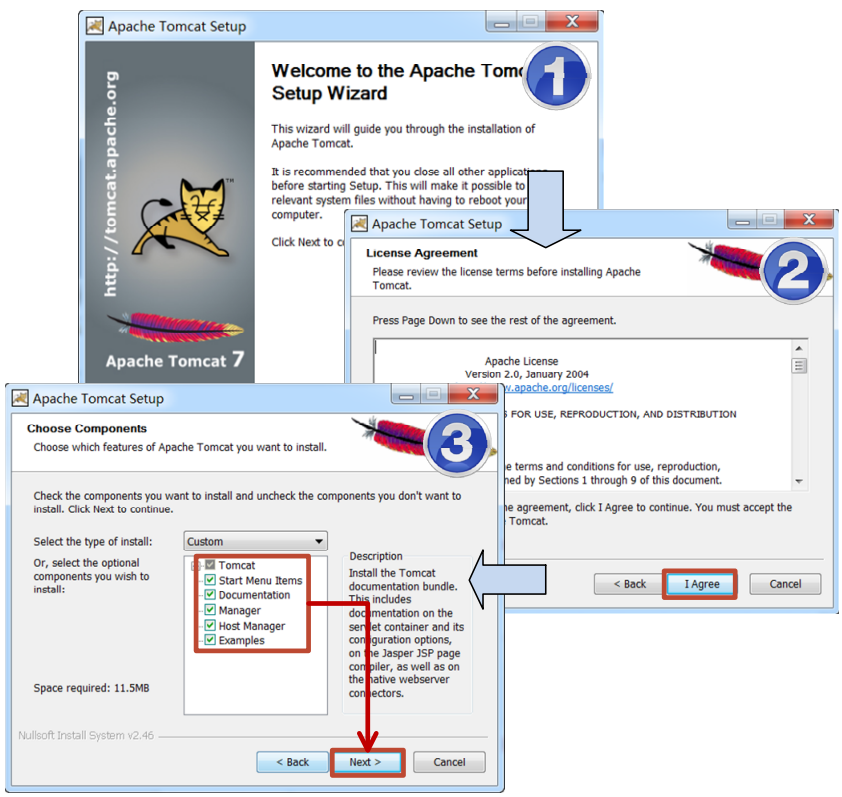



图 2.4　Tomcat 的安装过程

在单击“Next”按钮之后，我们要对 Tomcat 的路径进行配置。首先选择读者计算机中所

安装 Java 的 JRE 地址，然后设置 Tomcat 的安装路径。为了方便将来在 Tomcat 中部署应用程序，我们建议读者选择比较浅的路径进行安装，比如“D:\Tomcat7.0”，如图 2.5 所示。

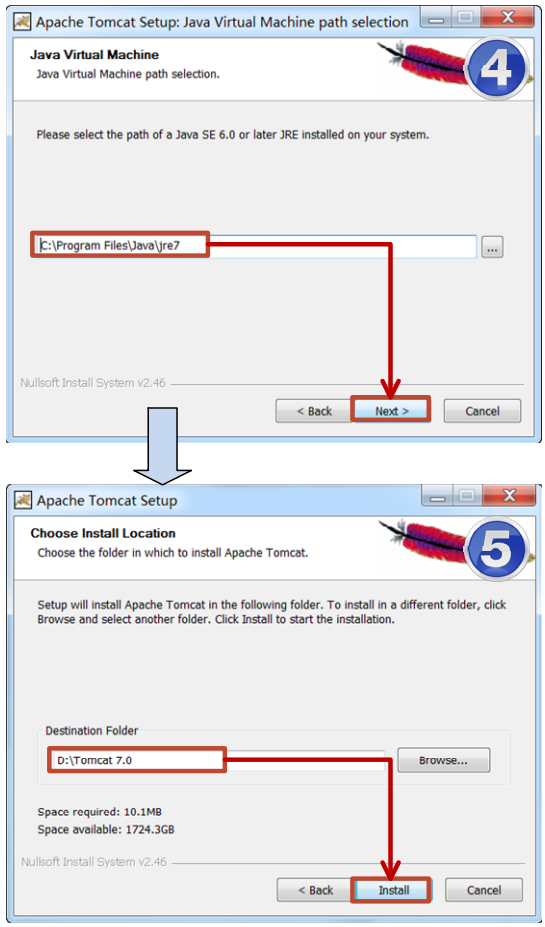



图 2.5　Tomcat 的安装路径

单击“Install”按钮，完成 Tomcat 的安装，如图 2.6 所示。

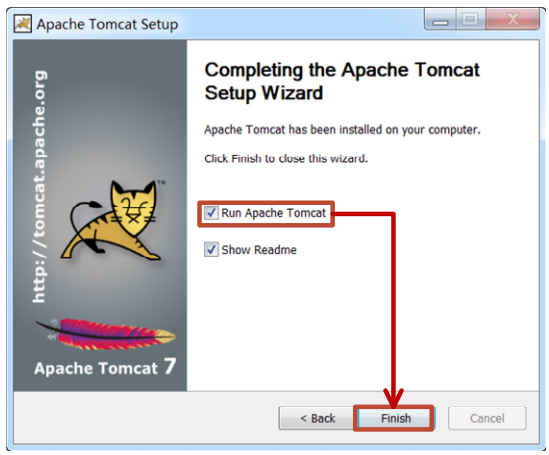



图 2.6　Tomcat 安装完成

安装完成后，会在系统栏中加载一个绿色的服务器图标，如图 2.7 所示。

图 2.7　右下角的服务器图标

打开浏览器，输入 http://localhost:8080/。会显示出一个如图 2.8 所示的界面，这表示 Web 服务器 Tomcat 已经成功地启动了。

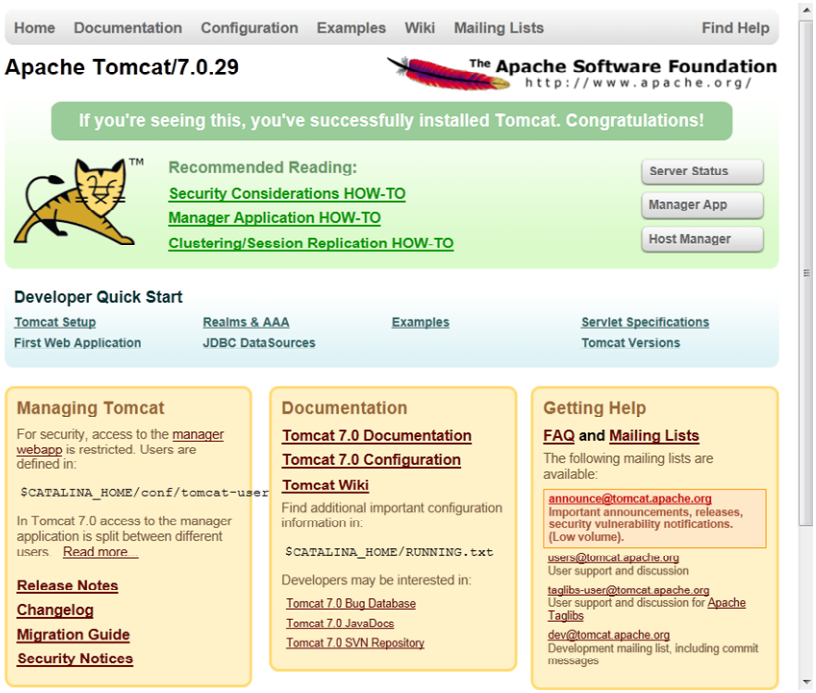



图 2.8　启动后的首页

在 Tomcat 上发布 Web 应用之前，我们首先先了解一下 Tomcat 的目录结构。打开 Tomcat 的安装路径后，如图 2.9 所示。


图 2.9　Tomcat 的目录结构

具体的每个文件夹的描述如表 2.1 所示。

表 2.1　Tomcat目录描述

| 目　　录 | 描　　述 |
| --- | --- |
| /bin | 存放在 Windows 平台以及 Linux 平台上启动和关闭 Tomcat 的脚步文件 |
| /conf | 存放 Tomcat 服务器的各种配置文件，其中最重要的配置文件是 server.xml |
| /lib | 存放 Tomcat 服务器所需的各种 JAR 文件 |
| /temp | 存放 Tomcat 产生的临时文件 |
| /logs | 存放 Tomcat 的日志文件 |
| /webapps | 当发布 Web 应用时，默认情况下把 Web 应用文件存放于此目录下 |
| /work | Tomcat 把由 JSP 生成的 Servlet 存放于此目录下 |

### 2.1.3　安装 MyEclipse

MyEclipse 是目前应用最为广泛的 Java 应用程序集成开发环境。它是由 Genuitec 公司开发的一款商业化软件。用户可以通过购买或互联网下载获得其安装包。本书所应用的为最新的 MyEclipse 10.0 版本。

首先我们双击 MyEclipse 的安装文件 myeclipse-10.0-offline-installer-windows.exe，开始 MyEclipse 的安装，安装过程如图 2.10 和图 2.11 所示。

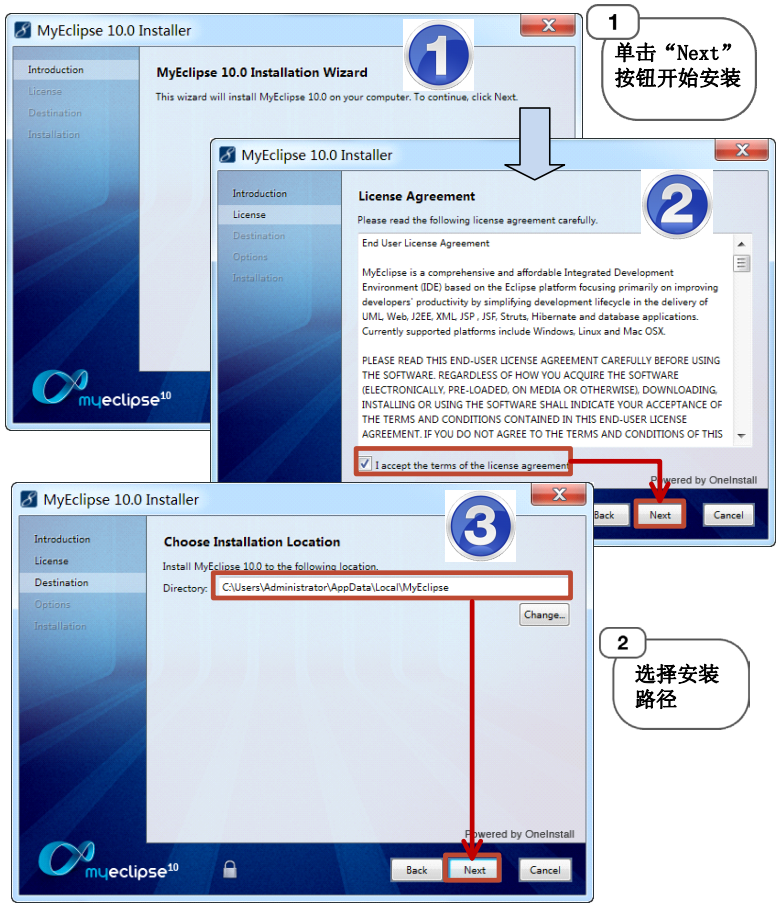



图 2.10　MyEclipse 的安装过程一

在下面的安装过程中我们需要为 MyEclipse 选择具体的安装版本，请读者根据自己计算机的系统信息进行安装，如图 2.11 所示。


图 2.11　MyEclipse 的安装过程二

单击“Finish”按钮完成 MyEclipse 的安装，系统自动显示 MyEclipse 的启动界面，如图 2.12 所示。

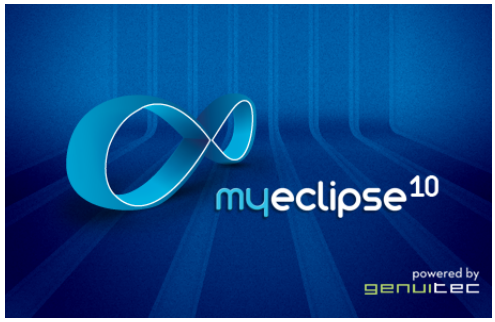



图 2.12　MyEclipse 的启动界面

接着我们就进入了 MyEclipse 的工作台窗口，其由菜单栏、工具栏等几部分构成，如图 2.13 所示。

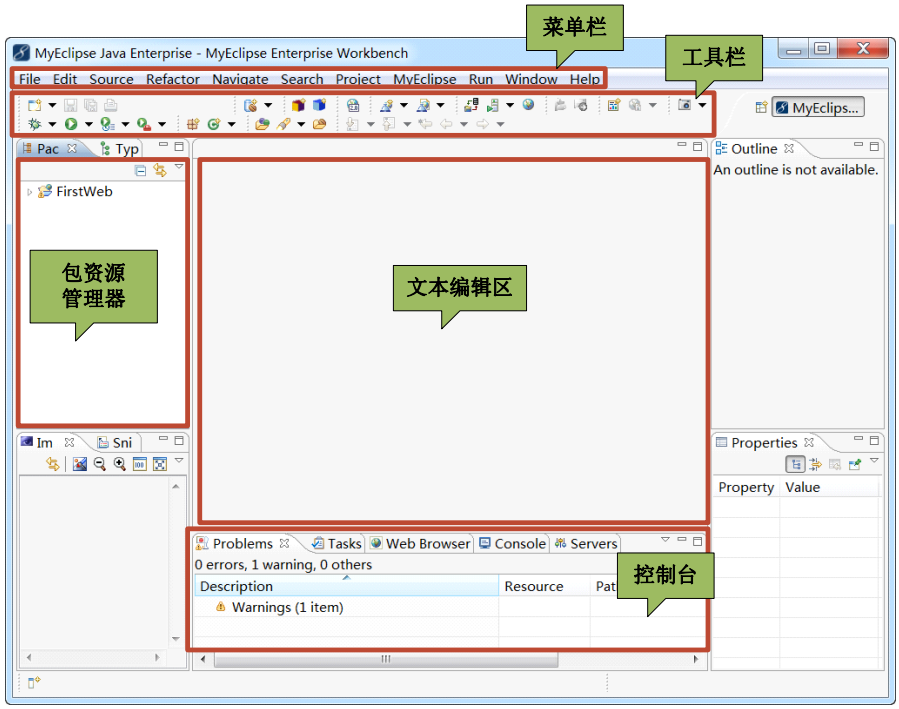



图 2.13　MyEclipse 的工作台窗口

## 2.1.4　MyEclipse 中集成 Tomcat 服务器

在 MyEclipse 中其实已经自带了一个 Tomcat 服务器，但是为了日后程序的开发、部署和运行更加方便和快捷，我们将用户安装的 Tomcat 服务器集成到 MyEclipse 中。具体的集成步骤如下所示。

① 选择 MyEclipse 菜单栏中的“Window”→“Preferences”命令，在弹出的窗口中选择“MyEclipse”→“Servers”→“Tomcat”，如图 2.14 所示。

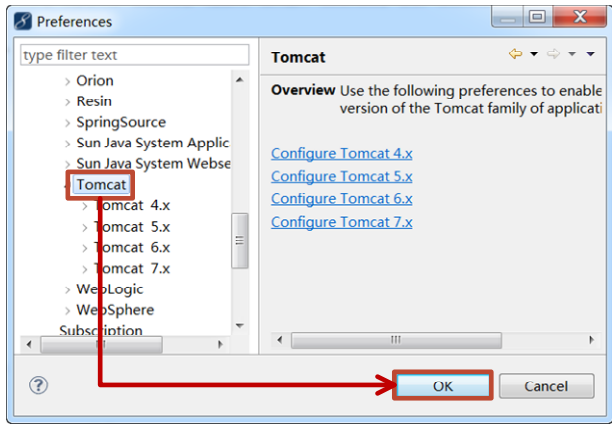



图 2.14　选择服务器配置

② 根据我们安装的 Tomcat 的版本选择 Tomcat 7.x 链接进行配置，系统出现 Tomcat 7.x 服务器配置窗口，首先我们将“Tomcat Server”选项设置为“Enable”，然后单击“Tomcat home

directory”选项后的“Browse…”按钮，选择 Tomcat 7.x 的安装目录。设置完成后如图 2.15 所示。

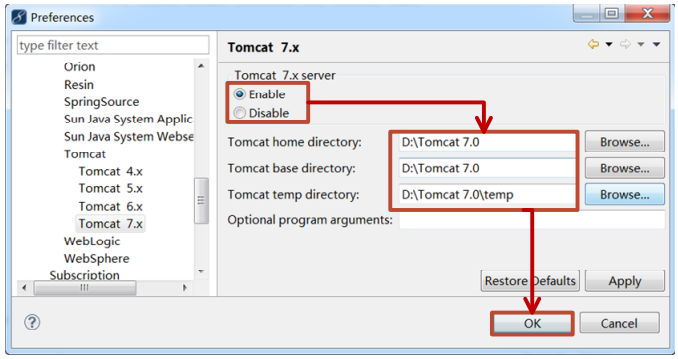



图 2.15　选择 Tomcat 7.x 的安装目录

③ 这样我们就基本完成了 MyEclipse 中 Tomcat 的集成，然后我们单击 MyEclipse 工具栏中的“Run MyEclipse Servers”按钮，将会看到 Tomcat 7.x 服务器。单击“Start”按钮，启动 Tomcat 服务器，如图 2.16 所示。

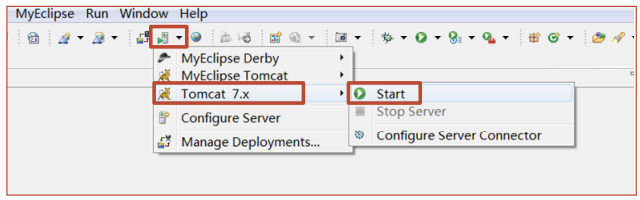



图 2.16　启动 Tomcat 7.x 服务器

打开浏览器，输入 http://localhost:8080/，若显示如图 2.17 所示的 Tomcat 服务器默认首界面，就说明我们可以在 MyEclipse 中直接控制 Tomcat 服务器了。

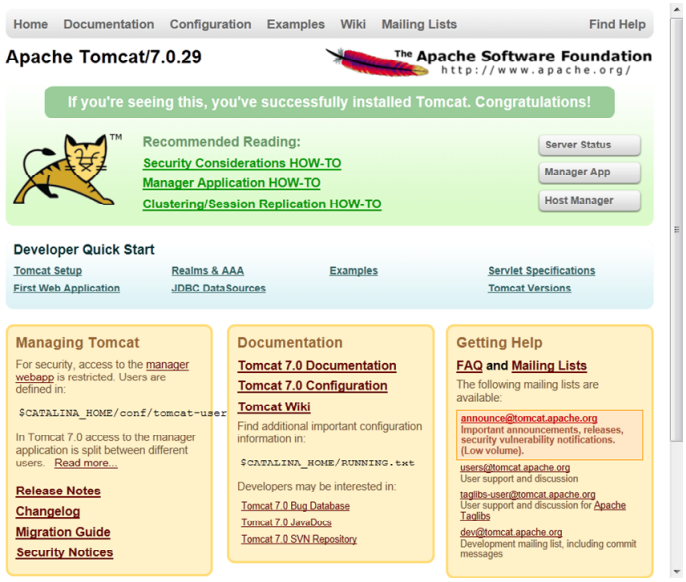



图 2.17　Tomcat 服务器默认首界面

### 2.1.5 MyEclipse 中 JSP 页面的创建

完成了各项软件的配置之后，我们一起来学习如何在 MyEclipse 中进行 JSP 页面的创建。在 MyEclipse 中，JSP 页面是以 Web 项目的形式组织起来的。所以要创建 JSP 页面之前，必须要创建一个 Web 项目，创建的具体步骤如下。

① 选择 MyEclipse 菜单栏中的“File”→“New”→“Project…”命令，将显示如图 2.18 所示的项目对话框，在其中选择“Web Project”选项。

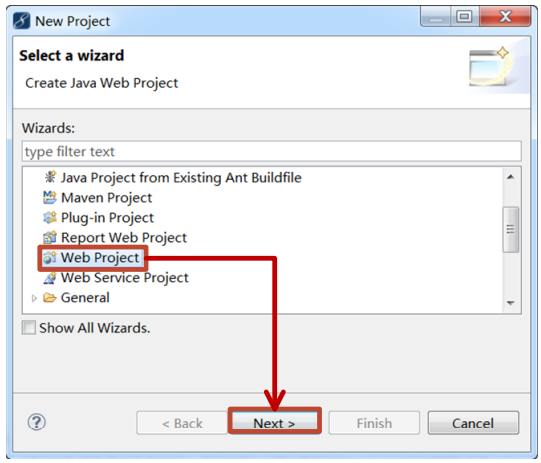



图 2.18　项目对话框

② 单击“Next”按钮，启动创建 Web 项目向导。在 Project Name 文本框中输入项目名称 FirstWeb，然后选择 J2EE Specification Level 下的 Java EE 6.0 单选按钮。最后单击“Finish”按钮完成 Web 项目的创建，如图 2.19 所示。

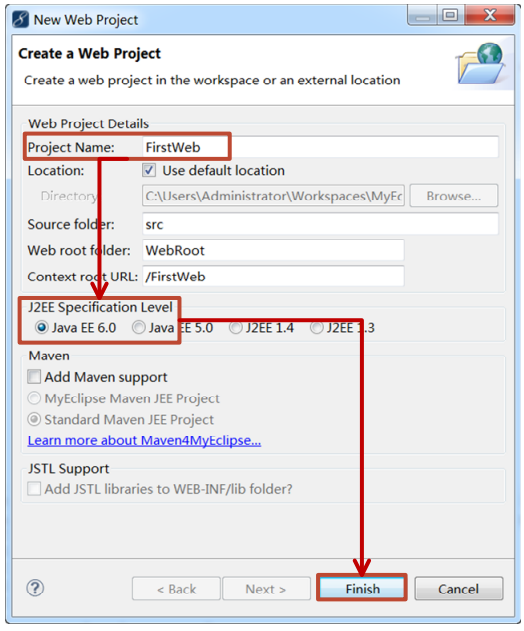



图 2.19　创建 Web 项目向导窗口

**注意：**J2EE Specification Level 选择哪个版本取决于读者所应用服务器版本。例如 Tomcat 4.x 以下版本只能选择 J2EE 1.4，而 Tomcat 6.x 服务器就可以选择 Java EE 5.0。我们应用的 Tomcat 7.x，所以建议选择 Java EE 6.0。

Web 项目创建完成后，就可以在该项目中创建 JSP 页面了，具体步骤如下。

① 选择 MyEclipse 菜单栏中的“File”→“New”→“JSP”（Advanced Templates）命令，将显示 JSP 创建窗口。我们可以在此创建窗口中确定 JSP 页面的名称和存储路径，如图 2.20 所示。

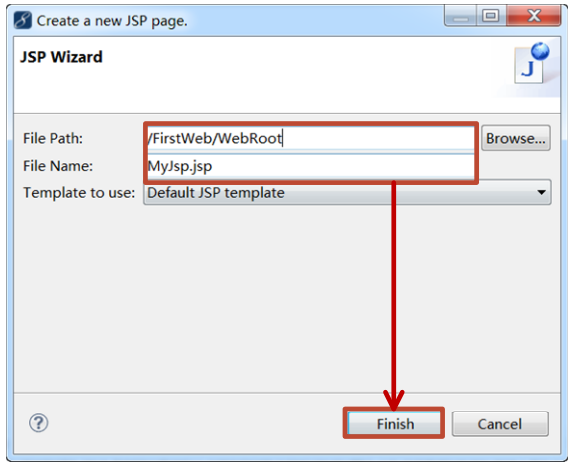



图 2.20　创建 JSP 向导窗口

② 单击“Finish”按钮，完成 JSP 页面创建，系统会自动生成如图 2.21 所示的界面。

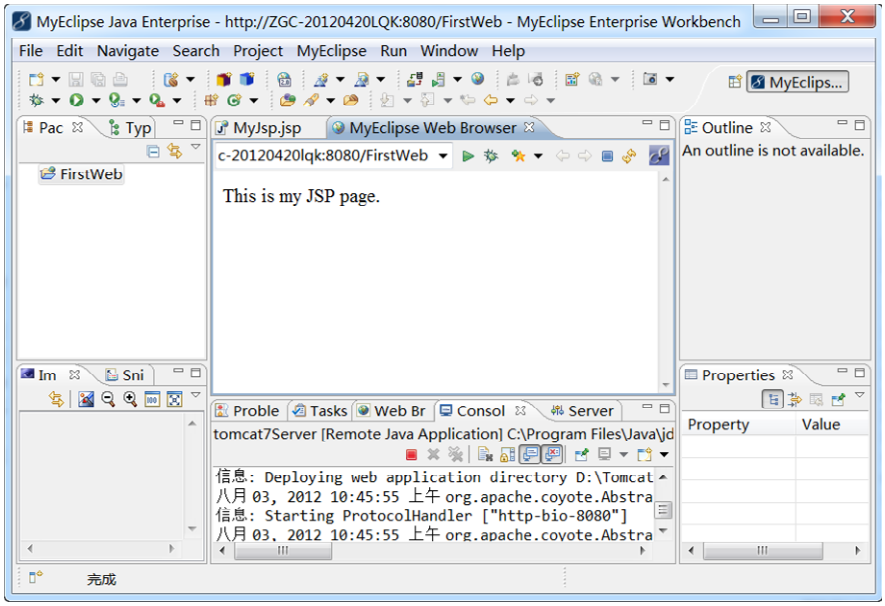



图 2.21　JSP 编辑器

### 2.1.6 MyEclipse 中 Web 项目的发布和运行

Web 项目在开发完成之后，需要发布到 Web 服务器上才能够被访问和运行。所以我们必须要掌握如何在 MyEclipse 中进行 Web 项目的发布和运行。在开发过程中 Web 项目的发布和运行的步骤如下所示。

① 在 Package Explorer 视图中，右键单击 FirstWeb 项目，在弹出的快捷菜单中选择“MyEclipse”→“Add and Remove Project Deployments…”命令，系统出现如图 2.22 所示的项目部署对话框。

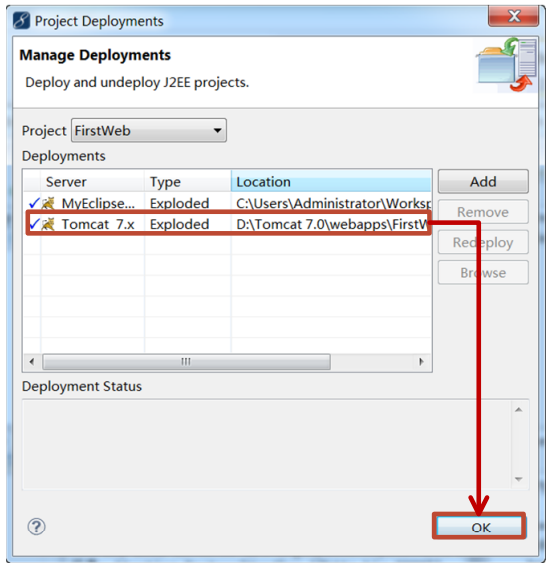



图 2.22 项目部署对话框

② 单击“Add”按钮，出现创建新部署对话框，选择 Tomcat 7.x 部署到服务器上，并在 Deploy type 中选择 Exploded Archive 开发模式，如图 2.23 所示。

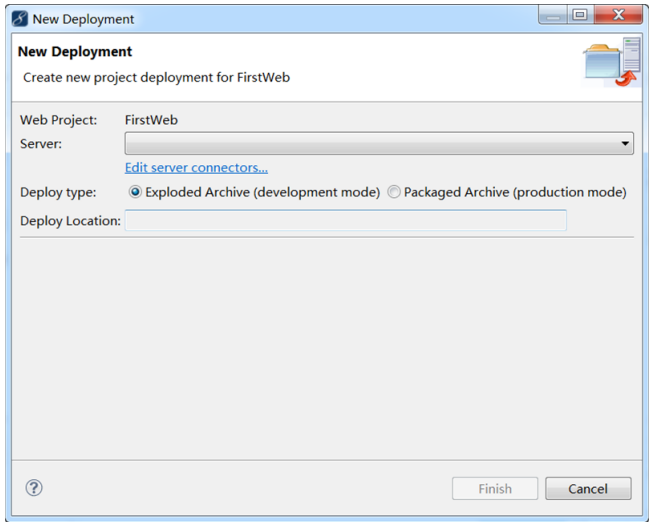



图 2.23 创建新部署

③ 单击“Finish”按钮，项目将部署到所选择的服务器中，如图 2.24 所示。

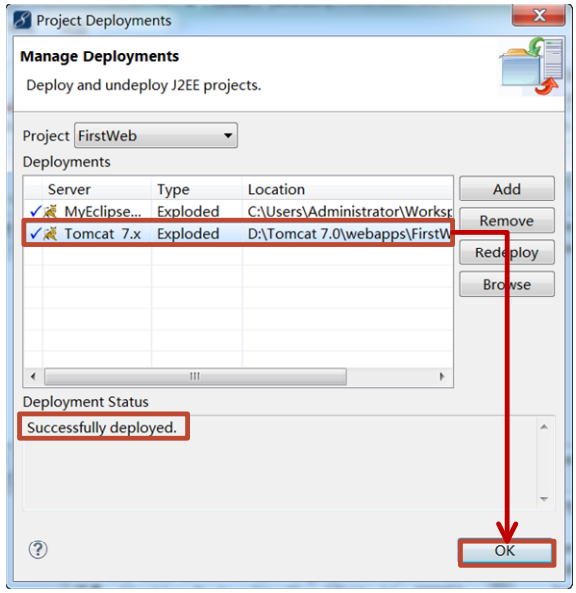



图 2.24　项目部署成功

④ 启动 Tomcat 服务器，输出的日志就会自动显示在 Console 视图中，便于读者浏览和判断服务器是否正常启动完毕，如图 2.25 所示。

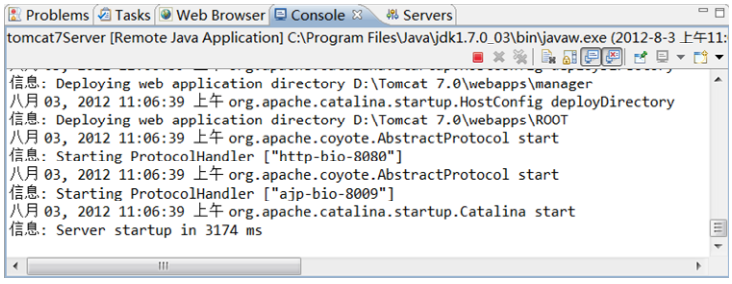



图 2.25　Tomcat 启动成功日志输出

## 2.2　JSP 的基本语法

JSP 网页主要分为脚本和网页数据两部分。网页数据就是 JSP 服务器不需要处理的部分。例如，HTML 的内容会直接送到客户端执行。脚本是必须经由 JSP 处理的部分，大部分脚本都以 XML 作为语法基础，其可以分为四种类型：JSP 脚本、编译指令、动作标签和表达式语言，如图 2.26 所示。

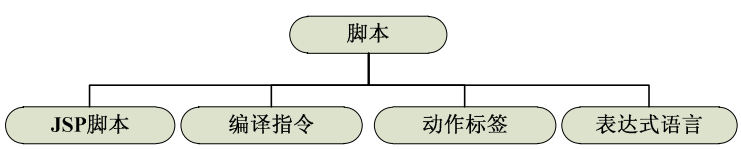



图 2.26　脚本的四种类型

本章我们会为大家讲解前面三种类型，关于表达式语言（Expression Language）会在后面的章节中为大家单独讲述。本节我们首先来看 JSP 脚本。

## 2.2.1 JSP 注释

JSP 程序中可以包含 3 种不同类型的注释，如图 2.27 所示。

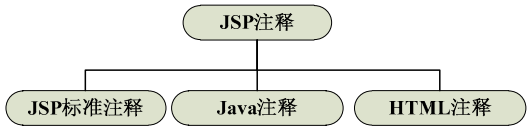



图 2.27 JSP 注释

### 1. JSP 标准注释

JSP 标准注释通常用来编写 JSP 说明文档，当 JSP 网页在服务器中编译时将被完全忽略，其语法格式如图 2.28 所示。

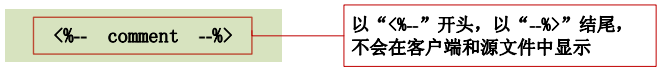



图 2.28 JSP 标准注释

这是开发程序员专用的注释，可以将开发人员希望隐藏的 JSP 程序注释起来。这些注释将不会显示在客户的浏览器中，用户也不能通过浏览器的“查看”→“源文件”操作，在源代码中查到。

### 2. Java 注释

在 JSP 的 Java 程序中，我们也可以遵循 Java 语言本身的注释规则，即在一对“<%”和“%>”中，将 Java 注释添加进去，如图 2.29 所示。

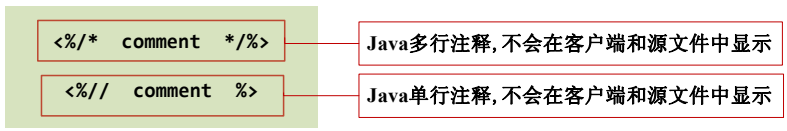



图 2.29 Java 注释

Java 注释在 JSP 页面编译时，也将会被完全忽略，同样用户也不能通过浏览器的“查看”→“源文件”操作，在源代码中查到。

### 3. HTML 注释

HTML 注释是一种能在客户端显示的注释，它的语法规则如图 2.30 所示。

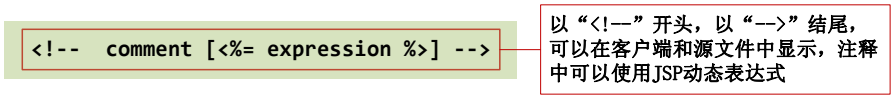



图 2.30 HTML 注释

JSP 页面中的 HTML 注释和 HTML 中的注释很相像，也可以通过浏览器的“查看”→“源文件”操作，在源代码中查到。唯一不同的是，可以在这个注释中使用 JSP 表达式，从而记录一些 JSP 页面动态运行结果。

【**示例 2.1**】我们可以创建一个 HelloWorld.jsp 示例来看 JSP 注释的用法，如图 2.31 所示。

```
<HTML>
  <BODY>
     <!--嵌入JSP代码-->
     <%
          for(int i=3;i<5;i++){
     %>
     <font size=<%=i%>><strong>世界，你好！ </strong></font>
     <!-- 嵌入JSP代码-->
     <%
          }
     %>
     <%--这里的注释不会被编译，在客户端的源文件中也不会出现 --%>
     <% //这里的注释不会被编译，在客户端的源文件中也不会出现 %>
  </BODY>
</HTML>
```

图 2.31　HelloWorld.jsp 示例

由于这是我们的第一个 JSP 文件，所以我们为大家讲解一下 JSP 文件的运行过程。首先在 MyEclipse 中建立一个名为 HelloWorld 的 JSP 文件，如图 2.32 所示。

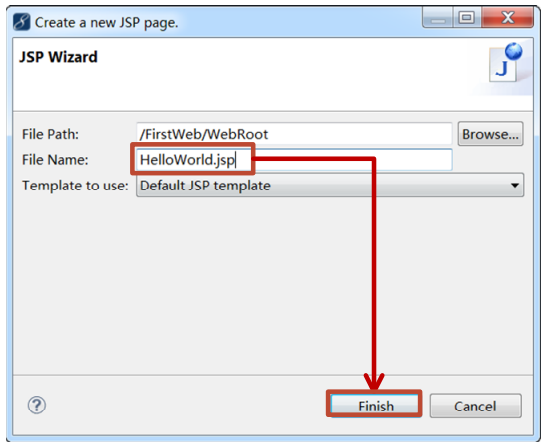



图 2.32　创建 HelloWorld.jsp 文件

然后在文本编辑区内输入如图 2.30 所示的内容，如图 2.33 所示。

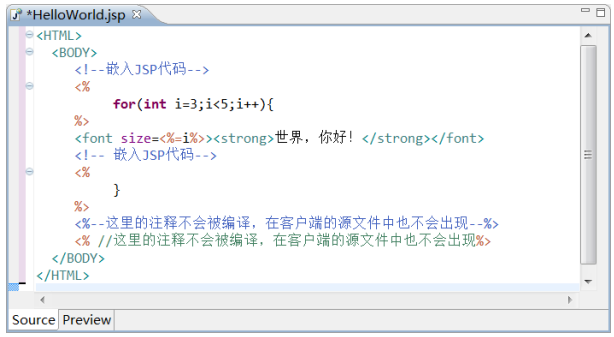



图 2.33　在文本编辑器中输入内容

接着对 HelloWorld.jsp 文件进行编译，编译方式有三种：我们可以执行菜单栏中的“Run”→“Run”命令进行编译，也可以使用快捷键“Ctrl+F11”，或者直接单击工具栏上的“编译”按钮。都可以完成编译过程，如图 2.34 所示。

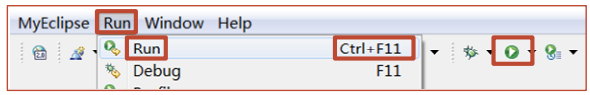



图 2.34　编译的三种方式

当控制台出现如图 2.35 所示的界面时说明编译通过，否则读者需要对自己的代码进行修改和调试。

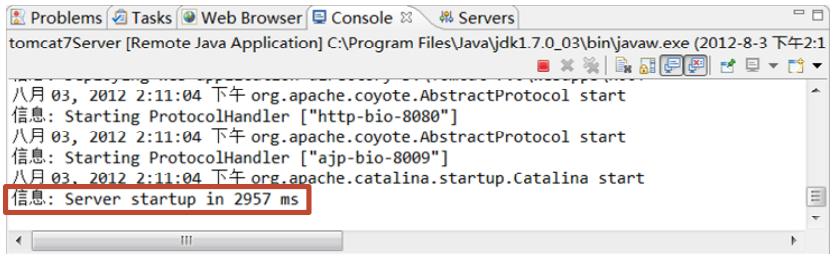



图 2.35　正确的编译结果

编译通过后，读者需要在自己的浏览器中输入文件的运行地址才能将 JSP 文件的运行结果显示出来，如图 2.36 所示。

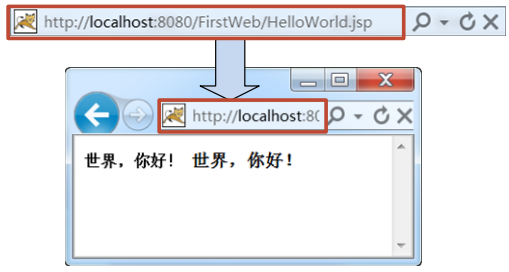



图 2.36　HelloWorld 程序运行结果

我们可以右键单击浏览器界面，在弹出的快捷菜单中单击“查看源文件”选项，就可以看出三种注释方法的区别，如图 2.37 所示。


图 2.37　三种注释方法在源文件中的显示区别

### 2.2.2　声明变量和方法

声明用于声明 JSP 程序中要用到的一个或多个变量和方法。在 JSP 中声明变量和方法，是以“<%!”开头，以“%>”结尾的，多个变量和方法以“;”分隔。JSP 声明的语法如图 2.38 所示。

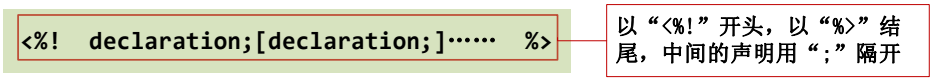



图 2.38　JSP 变量的声明

【**示例 2.2**】我们来看一个声明变量和方法的实例 jspdec.jsp，如图 2.39 所示。

```
<HTML>
    <head>
    </head>
    <BODY>
          <!--JSP中声明变量-->
          <%! int i = 0;  %>
          <%! int e,f,d ; %>
          <%! Object a= new Object();  %>
          <!--JSP中声明方法-->
          <%! public String f(int i){
                 if(i<3)
                    return "i小于3";
                 else
                    return "i大于等于3";
              }  %>
    </BODY>
</HTML>
```

图 2.39　jspdec.jsp 示例

### 2.2.3　JSP 表达式

JSP 表达式用来在 JSP 页面中输出作为运行结果的字符串或是数值变量。JSP 表达式可以被看做是一种简单的输出形式，任何在 Java 语言规范中有效的表达式都能够作为 JSP 表达式在 JSP 页面中使用。JSP 表达式语法如图 2.40 所示。

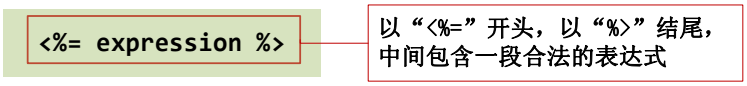



图 2.40　JSP 表达式语法

表达式具体的使用示例代码如下：

```
<font size=<%=i%>><strong>世界，你好！</strong></font>
<%=circle.getArea() %>
```

由于表达式的书写格式比较烦琐，而且完全可以由 JSP 中的内置对象 out（在后面章节中会作介绍）来替代，因此在实际开发中，JSP 表达式很少被用到。

## 2.3 JSP 编译指令

编译指令是指在 JSP 文件中包含在符号“<%@”和符号“%>”之间的部分。它不向客户端输出任何内容，是用来设置全局变量、声明类、方法和输出内容的类型指令。编译指令元素的格式如图 2.41 所示。

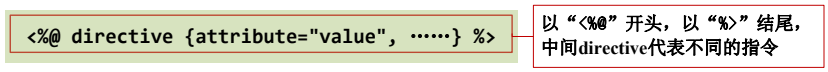



图 2.41　编译指令元素的语法格式

在 JSP 中的编译指令包括 3 种指令，如图 2.42 所示。

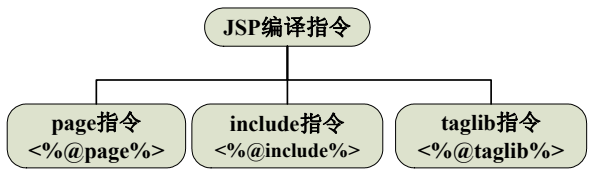



图 2.42　JSP 编译指令

### 2.3.1 page 指令

页面指令用来定义 JSP 文件中的全局属性，这些全局属性都是影响整个页面的重要属性。一个 JSP 文件中可以有多个页面指令，在 JSP 文件被解析为 Java 代码时，这些页面指令也被解析为对应的 Java 代码。page 页面指令的详细语法如图 2.43 所示。

```
<%@ page
[language="Java"]
[import="{package.class|package.*}",...]
[contentType="TYPE,charset=CHARSET"]
[session="true"|"false"]
[buffer="none|4kb|sizekb"]
[autoFlush="true"|"false"]
[info="text"]
[errorPage="true|false"]
[isErrorPage="true"|"false"]
[extends="package.class"]
[isELIgnored="true"|"false"]
[pageEncoding="painfo"]
%>
```

图 2.43　page 页面指令的详细语法

page 指令可以在一个 JSP 文件中多次、多处使用，但是其中的属性却只能使用一次（import 除外），重复的属性设置将会覆盖掉先前的设置。无论用户将 page 指令放在 JSP 程序的任何地方，它的作用范围都是整个 JSP 页面。

图 2.43 的语法列出来了大部分页面指令属性及其取值，下面通过表 2.2 详细解释每个属性的意义和使用方法。

表 2.2　page页面指令属性解释

| 属　性 | 描　述 | 默 认 值 | 例　子 |
|---|---|---|---|
| language | 指定使用的脚本语言，目前只能为“Java” | “Java” | language="Java" |
| import | 和 Java 代码中 import 的作用一样，如果有多个包，用“,”隔开 | 默认忽略该属性 | import="Java.io.*, Java.utils.*" |
| session | 这个页面是否参与一个指定的 HTTP 会话 | true | session="true" |
| buffer | 指定客户端输出流的缓冲模式，可以为一个数值或 none | 不小于 8kb | buffer="32kb" |
| autoFlash | 设置当缓冲区满时，是否实现自动刷新 | true | autoFlush="true" |
| Info | 指定关于该 JSP 文件的信息 | 默认忽略该属性 | info="第一个 JSP 页面" |
| isErrorPage | 表明当前页是否为其他页的 errorPage，由此决定当前页是否可以使用 exception 对象 | false | isErrorPage="false" |
| errorPage | 定义当前页运行出现异常时调用的页面 | 默认忽略该属性 | errorPage="error.jsp" |
| isThreadSafe | 指定该 JSP 文件是否能多线程使用，由此判断是否可以处理多个用户的请求 | true | isThreadSafe="true" |
| contentType | 定义JSP字符编码和页面响应的MIME类型 | TYPE=text/html CHARSET=iso8859-1 | contentType="text/html; charset=UTF-8" |
| pageEncoding | 指定该 JSP 文件的字符编码 | pageEncoding="iso-8859-1" | pageEncoding="UTF-8" |
| isELIgnored | 指定 EL（表达式语言）是否被忽略，如果为 true，则容器忽略“${}”表达式的计算 | Servlet2.3 之前的版本忽略该属性 | isELIgnored="true" |

一个“<%@%>”内可以出现一个或多个上述页面指令的属性。

**【示例 2.3】**我们来看一个使用了多个页面指令属性的 JSP 文件 page.jsp，如图 2.44 所示。

```
<!--下面的代码使用页面指令-->
    <%@ page language="java" import="java.util.Date"
        session="true"  buffer="12kb" autoFlush="true" info="One test page"
        errorPage="error.jsp" isErrorPage="false"
        contentType="text/html; charset=UTF-8"%>
    <%@ page errorPage="error.jsp" %>
    <%@ page isELIgnored="false"%>
<html>
   <body>
      <h1> 使用page指令的页面</h1>
           现在的时间是：
      <%=new java.util.Date().toLocaleString()%>
   </body>
</html>
```

图 2.44　page.jsp 示例

在浏览器中的运行结果，如图 2.45 所示，显示了当前的计算机的时间。


图 2.45　page.jsp 的运行结果

## 2.3.2　include 指令

include 指令用来将指定位置上的资源包含在当前 JSP 文件中。在 JSP 文件被编译为 Java 文件时，这些被包含的资源会被作为 JSP 文件的一部分被翻译为 Java 文件。所以这些资源可以看做是 JSP 文件的一部分。在 JSP 文件中 include 指令的语法形式如图 2.46 所示。

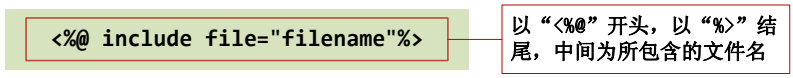



图 2.46　include 指令的语法形式

图中的 filename 指定要包含资源的文件名。如果 filename 以“/”开头，那么该文件的路径是参照 JSP 应用的上下文路径；如果 filename 是以文件名或目录名开头，那么该文件的路径就是当前 JSP 文件的路径。

include 指令的具体示例代码如下：

```
<%@ include file="HelloWorld.jsp"%>
<%@ include file="name.html"%>
<%@ include file="/FirstWeb/test.txt"%>
```

其中“/FirstWeb/test.txt”表示所要包含的文件 test.txt 在当前使用的 JSP 文件同级路径中的 FirstWeb 目录中。

值得注意的是，被包含 JSP 文件中不能还有 page 指令，以免同 JSP 文件中已有的同样指令发生冲突。

**【示例 2.4】**我们来看一个 include 指令的应用示例 include.jsp，如图 2.47 所示。

**include.jsp:**

```
<%@  page contentType="text/html;charset=GB2312"%>
<html>
  <head>
  include编译指令测试<br>
  <%@ include file = "include1.jsp" %>
  <%--使用include指令包含include1.jsp页面 --%>
  </body>
</html>
```

**include1.jsp:**

```
<%! String str="Here is include's context!";%>
<% out.println(str+"<br>"); %>
```

图 2.47　include 指令的应用示例 include.jsp

在浏览器中输入地址，显示运行结果，如图 2.48 所示。


图 2.48　include.jsp 的运行结果

## 2.3.3　taglib 指令

taglib 指令是当 JSP 页面中引用了用户自定义标签时，用来声明这些用户自定义标签的。JSP 引擎使用 taglib 指令以确定在 JSP 中遇到用户自定义标签时应该怎样去做。taglib 指令的语法形式如图 2.49 所示。


图 2.49　taglib 指令的语法形式

**【示例 2.5】**我们举一个例子 Mytaglib.jsp 运用 taglib 指令说明如何使用 JSP 标准标签库（JSTL）中的 c 标签，如图 2.50 所示。

```
<!--uri声明了使用的标签库的地址，c为标签的前缀-->
<%@ page contentType="text/html; charset=UTF-8"  %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
   <head>
       <title>一个用户使用JSTL标签的例子</title>
   </head>
   <body>
     <h3>一个用户使用JSTL标签的例子：</h3>
     <% session.setAttribute("Name","Smith");%>
     <h4>Name:</h4>
       <c:if test = "${sessionScope.Name == 'Smith'}">
       <!--if标签为定义好的jstl标签-->
     ${sessionScope.Name}<br>
     </c:if>
   </body>
</html>
```

uri声明了使用的标签库和描述文件的地址，prefix指定了前缀名

图 2.50　taglib 指令的使用示例

“<c:if test = "${sessionScope.Name == 'Smith'}">”用来判断 session 对象中存放的 Name 属性的值是否为“Smith”。如果是，则显示该值。在浏览器中输入地址，显示运行结果，如图 2.51 所示。

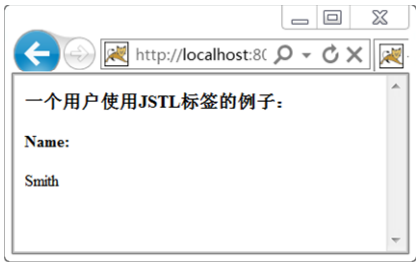



图 2.51　Mytaglib.jsp 的运行结果

## 2.4　JSP 动作指令

JSP 动作是一种特殊的标签。利用 XML 语法格式的标签来控制 JSP 引擎的行为，影响 JSP 运行时的功能，并返回客户端的响应。JSP 动作都以“<jsp:”开头，相对应地则以“/>”结束。

JSP 主要包括的动作如图 2.52 所示。

| | |
|---|---|
| <jsp:include> | 用于在客户端请求时间内把静态或者动态的资源包含在JSP页面内 |
| <jsp:forward> | 用于将请求转发到其他的JSP页面、Servlet或者静态资源文件 |
| <jsp:param> | 用于给其他的标签提供参数 |
| <jsp:useBean> | 用于在JSP文件中使用一个JavaBean的实例，并声明该实例的名字以及作用范围 |
| <jsp:setProperty> | 用于给JavaBean设置属性，该标签会调用JavaBean的setXXX()方法去完成一个或者多个属性的设置 |
| <jsp:getProperty> | 用于获取JavaBean中属性的值。它将JavaBean的值转换为String类型，然后发送到输出流中 |

图 2.52　JSP 中主要的动作指令

本节将首先介绍前 3 个动作指令，在后面章节中我们将结合 JavaBean 详细介绍与 JavaBean 相关的动作指令。

### 2.4.1　<jsp:include>动作指令

<jsp:include>动作元素用于在客户端请求时间内把静态或者动态的资源包含在 JSP 页面内。包含的静态或动态的资源在 page 属性中用 URL 的形式指定。<jsp:include>指令的格式如图 2.53 所示。

include 动作指令可以在 JSP 页面中动态包含一个文件，这与 include 指令不同，前者可以动态包含一个文件，文件的内容可以是静态的文件也可以是动态的脚本，而且当包含的动态文件被修改时，JSP 引擎可以动态对其进行编译更新。而 include 指令仅仅是把一个文件简单的包含在一个 JSP 页面中，从而组合成一个文件，仅仅是简单的组合作用。

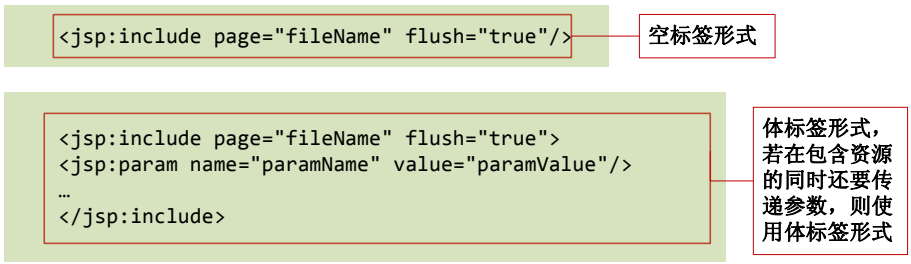



图 2.53　<jsp:include>指令使用格式

【**示例 2.6**】我们来看一个具体的示例 IncludeAction.jsp 来看 include 动作指令的使用方法，如图 2.54 所示。

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

<html>

 <head>

   <title>include动作指令使用示例程序</title>

</head>

 <body>

 <font size="2">

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

  </font>

 </body>

</html>
```

<jsp:include>指令的使用格式

图 2.54　IncludeAction.jsp 示例

在程序中使用了 include 指令包含了一个动态的 JSP 文件 footer.jsp，其内容如图 2.55 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=UTF-8"%>

<html>

 <body>

   <%

      out.print("这里用include动作指令包含一个动态的JSP页面!");

    %>

 </body>

</html>
```

图 2.55　footer.jsp 示例

在浏览器中输入地址 http://localhost:8080/FirstWeb/IncludeAction.jsp，显示运行结果，如图 2.56 所示。


图 2.56　IncludeAction.jsp 的运行结果

## 2.4.2　<jsp:forward>动作指令

forward 动作指令可以用来控制网页的重定向。即将请求转发到其他的 JSP 页面、HTML 页面或者 Servlet 类。<jsp:forward>动作指令的语法格式如图 2.57 所示。

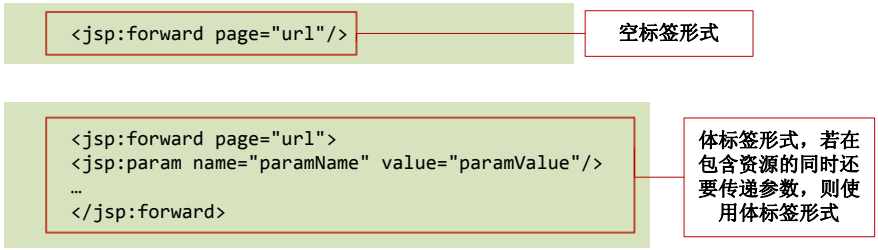



图 2.57　<jsp:forward>动作指令语法格式

只要指明 page 的值，当 JSP 执行到这行代码时就可以直接跳转到对应的页面。

**【示例 2.7】**forward 动作指令的具体用法可以参考如图 2.58 所示的示例 Forward.jsp。

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

<html>

 <head>

    <title>forward动作指令使用示例</title>

 </head>

 <body>

  <jsp:forward page="http://www.baidu.com"/>

 </body>

</html>
```

<jsp:Forward>指令使用格式

图 2.58　Forward.jsp 示例

在浏览器中输入地址 http://localhost:8080/FirstWeb/Forward.jsp，显示运行结果，如图 2.59 所示。


图 2.59　Forward.jsp 运行结果

## 2.4.3　<jsp:param>动作指令

<jsp:param>动作指令主要用于传递参数，为其他动作指令提供附加信息，它必须和<jsp:include>、<jsp:forward>等动作指令一起使用，可以将<jsp:param>动作指令中的值传递到其他动作指令所要加载的文件中去，从而实现向所加载的页面动态传递参数的功能。<jsp:param>动作指令的语法格式如图 2.60 所示。

```
<jsp:param name="paramName" value="paramValue"/>
```

name表示传递参数名，
value表示传递参数值

图 2.60　<jsp:param>动作指令语法格式

通过<jsp:param/>可以给请求的页面传输一个或多个参数。如果要传输多个参数，可以使用多个<jsp:param/>标签。当给一个页面传输参数时，该页面肯定是动态页面。

**【示例 2.8】**下面通过一个示例 PassParam.jsp 说明<jsp:param/>传递参数方法，如图 2.61 所示。

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

<html>

 <head>

   <title>页面跳转并且传递参数示例</title>

 </head>

 <body>

      从这个页面传递参数：

     <jsp:forward page="GetParam.jsp">

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

     </jsp:forward>

 </body>

</html>
```

<jsp:param>动作指令参数的传递

图 2.61　PassParam.jsp 示例

在上面这段程序中，把页面跳转到 GetParam.jsp 这个页面，同时向 GetParam.jsp 这个页面传递了一个名称为 param 的参数，这个参数的值为 JavaWeb。GetParam.jsp 示例如图 2.62 所示。

```
<html>
 <head>
   <title>页面跳转并且传递参数示例</title>
 </head>
 <body>
 <font size="2">
  这个页面接受传递过来的参数：<br>
 前一个页面传递过来的参数为：<%out.print(request.getParameter("param")); %>
  </font>
 </body>
</html>
```

接收参数值

图 2.62　GetParam.jsp 示例

在浏览器中输入地址 http://localhost:8080/FirstWeb/PassParam.jsp，显示运行结果，如图 2.63 所示。

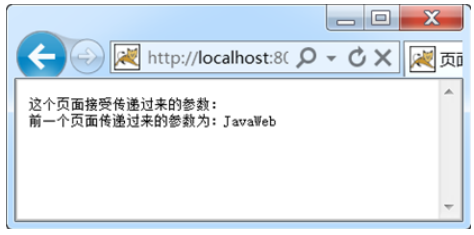



图 2.63　PassParam.jsp 运行结果

## 2.5　小结

本章主要介绍了 Tomcat 服务器和 MyEclipse 开发软件的安装过程，以及和 JSP 有关的基本语法等方面的内容。并通过一个 HelloWorld 程序对 MyEclipse 中 Java Web 应用开发的基本步骤和流程有了一个大体的认识。本章的重点是讲解 Java Web 程序运行环境的搭建和 JSP 基本语法，难点是运用 JSP 编译指令和动作指令的运用。希望读者多加练习，以便为后面章节中 JSP 学习的深入打好基础。

## 2.6　本章习题

1．请读者参照示例 2.4 运用 include 指令建立一个网页，并在此网页中输出另一个 JSP 文件中的内容。执行效果如图 2.64 所示。


图 2.64　运行结果

【分析】本题主要考查 JSP 编译指令中 include 指令的运用。include 指令用来将指定位置上的资源包含在当前 JSP 文件中，所以我们只要在第一个文件中指定好引用文件的路径就可以了。

【核心代码】本题的关键代码如下所示。

include.jsp:

```
<%@ page contentType="text/html;charset=GB2312"%>
<html>
  <head>
  include 编译指令习题测试<br>
  <%@ include file = "include1.jsp" %>
  </body>
</html>
```

include1.jsp:

```
<%! String str="I love Java Web!";%>
<% out.println(str+"<br>"); %>
```

2．请读者参照示例 2.8 运用<jsp:param>动作指令建立一个动态网页，并在此网页中输出另一个 JSP 文件传递的参数内容。执行效果如图 2.65 所示。

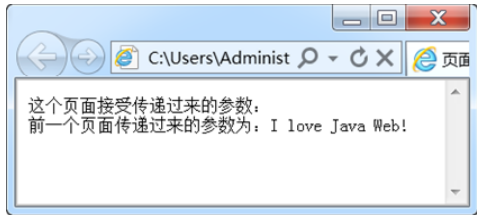



图 2.65　运行结果

【分析】本题主要考查 JSP 编译指令中<jsp:param>动作指令的运用。<jsp:param>动作指令主要用于传递参数，为其他动作指令提供附加信息，所以我们需要在第二个文件中添加参数信息。

【核心代码】本题的关键代码如下所示。

PassParam.jsp：

```
    <%@ page language="java" import="java.util.*" contentType="text/html;charset=
gb2312"%>
    <html>
     <head>
       <title>页面跳转并且传递参数示例</title>
     </head>
     <body>
         从这个页面传递参数：
         <jsp:forward page="GetParam.jsp">
```

```
        <jsp:param name="param" value="I love Java Web!"/>
    </jsp:forward>
 </body>
</html>
```

GetParam.jsp：

```
<html>
 <head>
   <title>页面跳转并且传递参数示例</title>
 </head>
 <body>
 <font size="2">
  这个页面接受传递过来的参数：<br>
  前一个页面传递过来的参数为：<%out.print(request.getParameter("param")); %>
  </font>
 </body>
</html>
```

# 第3章 JSP 内置对象

JSP 内置对象是为了简化 JSP 页面开发而建立的一些内部对象。这些对象不需要声明，可以在程序中直接使用。它们是 JSP 语言的精髓，掌握常见内建对象的使用技巧是进行 Java Web 开发必不可少的。正确地掌握和灵活地使用 JSP 内置对象是学习 JSP 开发的重中之重。本章就来为大家讲解九大内置对象，如表 3.1 所示。

表 3.1 JSP内置对象及作用

| 内 置 对 象 | 主 要 作 用 |
| --- | --- |
| request | 包含客户端请求信息 |
| response | 页面传回给用户端的相应信息 |
| out | 用来向客户端浏览器输出信息的数据流 |
| session | 为发送请求的客户建立会话 |
| application | 保存整个应用程序的共享信息 |
| pageContext | 保存当前 JSP 页面的共享信息 |
| config | 读取初始化参数 |
| page | 代表 JSP 网页本身 |
| exception | 获取运行时的异常 |

由于我们在实际开发中经常会遇到中文乱码问题，所以在本章的结尾我们会为大家单独用一小节来讨论如何解决中文乱码的问题。

## 3.1 request 内置对象

request 对象用来接收客户端提交的各种信息。如果要实现与用户的互动，必须要知道用户的需求，然后根据这个需求生成用户期望看到的结果。这样才能实现与用户的互动。在 Web 应用中，用户的需求就抽象成一个 request 对象，这个对象中间包括用户所有的请求数据，例如通过表单提交的表单数据等方式传递的参数，这些就是用户的需求。表 3.2 中列举了 request 对象中的常用方法及方法描述。

表 3.2 request对象常用方法

| 方 法 | 方 法 描 述 |
| --- | --- |
| getParameter(Sting name) | 获取客户端传给服务器的参数值，name 指定表单中参数的名字 |
| getParameterNames() | 获取客户端传给服务器的所有参数的名字，返回的结果是一个枚举实例 |

续表

| 方　法 | 方 法 描 述 |
| --- | --- |
| getParameterValues(Sting name) | 获得某一个参数的所有的值，name 指定参数名字 |
| getAttribute(Sting name) | 获得 request 对象中某一个属性的值，name 为属性名，如果该属性不存在，则返回 null |
| setAttribute(Sting name,Java.long.Object.objt) | 给 request 对象设置一个名字为 name 的属性值，该值由 objt 设置 |
| removeAttribute(String name) | 移除 request 对象中名字为 name 的属性 |
| getAttributeNames() | 返回 request 对象中所有属性的名字，结果是一个枚举类型 |
| getCookies() | 返回客户端所有的 Cookie 对象，结果是一个 Cookie 数组 |
| getCharacterEncoding() | 返回客户端请求中字符的编码方式 |
| getContentLength | 返回客户端请求的 body 的长度 |
| getMethod() | 返回客户端向服务器传输数据的方法，如 get、post、header、trace 等 |
| getRequestURL() | 获取发送请求的客户端地址 |
| getRemoteAddr() | 获取客户端的 IP 地址 |
| getServerName() | 获取服务器的名字 |
| getServerPort() | 获取服务器的端口号 |
| getServletPath() | 获取客户端所请求的脚本文件的文件路径 |

### 3.1.1　获取用户提交的表单信息

request 对象最主要的一个作用就是用来封装用户提交的表单信息，然后通过如下两个方法来获取用户提交的表单信息。

- getParameter(Sting name)：获取客户端传给服务器的参数值。
- getParameterValues(Sting name)：获得某一个参数的所有的值。

**【示例 3.1】**下面是一个简单的表单示例 Form.jsp。在这里为了方便说明，我们在把表单提交给这个页面自身，并在这个页面中取出提交的表单的数据，如图 3.1 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
 <head>
   <title>request获取表单数据示例</title>
 </head>
 <body>
 <font size="2">
 下面是表单内容:
 <form action="Form.jsp" method="post">
      用户名:<input type="text" name="userName" size="10"/>
      密　码:<input type="password" name="password" size="10"/>
    <input type="submit" value="提交">
  </form>
  下面是表单提交以后用request取到的表单数据:<br>
  <%
out.println("表单输入userName的值:"+request.getParameter("userName")+"<br>");
out.println("表单输入password的值:"+request.getParameter("password")+"<br>");
   %>
  </font>
 </body>
</html>
```

图 3.1　Form.jsp 示例

在浏览器中输入地址 http://localhost:8080/JSPWeb/Form.jsp，显示运行结果，如图 3.2 所示。

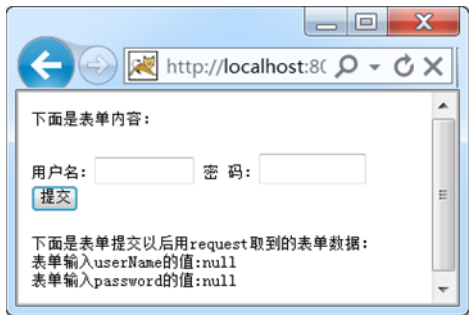



图 3.2　Form.jsp 运行结果

然后我们在表单中填入内容，例如用户名为“lester”，密码为 123，运行结果如图 3.3 所示。


图 3.3　填入值后运行结果

这是一个典型的获取用户表单的示例，它体现了 JavaWeb 中数据和提交的大致实现过程，如图 3.4 所示。

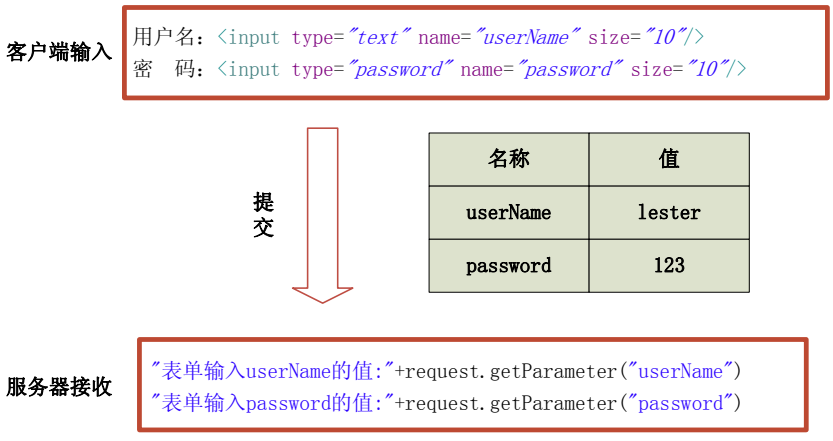



图 3.4　JavaWeb 中数据的提交和接收

## 3.1.2　获取服务器端和客户端信息

使用 request 对象可以获取提交请求的客户端信息及接收请求的服务器端信息，这些获取信息的方法如图 3.5 所示。

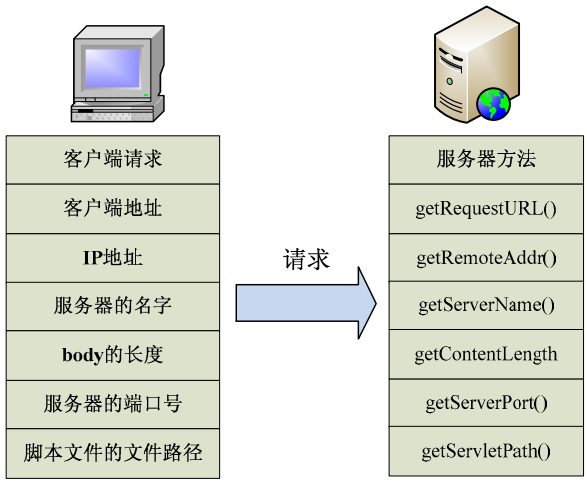



图 3.5　获取服务器端和客户端信息

【**示例 3.2**】我们举一个示例 request.jsp 来获取客户端和服务器的信息，如图 3.6 所示。

```
<%@ page import="java.io.*"%>
   request对象的方法:
   <hr>

   <%
      out.println("<br>getAttributeNames:");

      java.util.Enumeration e=request.getAttributeNames();

      while(e.hasMoreElements())
      out.println(e.nextElement());

      out.println("<br>getMethod:");
      out.println(request.getMethod());

      out.println("<br>getParameter:");
      out.println(request.getParameter("name"));

      out.println("<br>getCharacterEncoding:");
      out.println(request.getCharacterEncoding());

      out.println("<br>getContentLength: ");
      out.println(request.getContentLength());

      out.println("<br>getRemoteAddr:");
      out.println(request.getRemoteAddr());

      out.println("<br>getServerName:");
      out.println(request.getServerName());

      out.println("<br>getServerPort:");
      out.println(request.getServerPort());

    %>
```

图 3.6　request.jsp 示例

在浏览器中输入地址，显示运行结果，如图 3.7 所示。

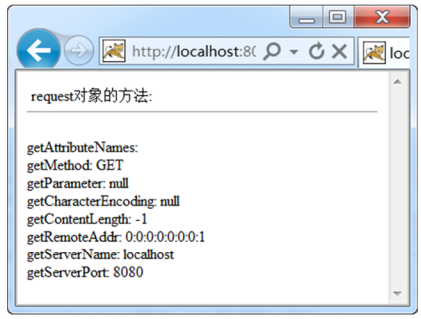



图 3.7　request.jsp 的运行结果

### 3.1.3　request 中保存和读取共享数据

request 对象不仅能够封装请求信息，而且可以保存和读取某一范围内的共享数据。request 对象定义了一对方法 getAttribute(Sting name)和 setAttribute(Sting name,Java.long.Object.objt)，用来在 request 对象中读取和保存数据。

**【示例 3.3】**我们看一个 Attribute.jsp 的示例，在这个示例中实现共享数据的保存和读取，如图 3.8 所示。

```
<%@ page contentType="test/html;charset=GB2312" %>
<html>
  <body>
   <%
   request.setAttribute("name","JavaWeb");
   %>
  <jsp:forward page="readAttribute.jsp"/>
  </font>
  </body>
</html>
```

将共享信息存储在request对象中

图 3.8　Attribute.jsp 示例

setAttribute()方法将共享数据保存到 request 对象中，然后再使用<jsp:forward/>动作指令转向 readAttribute.jsp 页面。readAttribute.jsp 示例如图 3.9 所示。

```
<%@ page contentType="test/html;charset=GB2312" %>
<html>
  <body>
      从request中读取的共享信息是:
   <% out.println(request.getAttribute("name")); %>
  </font>
  </body>
</html>
```

获取共享信息并显示

图 3.9　readAttribute.jsp 示例

在浏览器中输入地址，显示运行结果，如图 3.10 所示。


图 3.10　Attribute.jsp 的运行结果

## 3.2　response 内置对象

response 对象是服务器端向客户端返回的数据，从这个对象中间可以取出一部分与服务器互动的数据和信息。Response 对象的主要方法及方法描述如表 3.3 所示。

表 3.3　response 对象的主要方法

| 方　法 | 方 法 描 述 |
|---|---|
| addCookie(Cookie cookie) | 添加一个 Cookie 对象，用来保存客户端信息 |
| addHeader(Sting name, String value) | 添加 HTTP 文件头信息，如果已有同名的 Header，则覆盖它 |
| containHeader(String name) | 判断名字为 name 的 HTTP 文件头是否已存在 |
| flushBuffer() | 强制将当前缓冲区的内容发送到客户端 |
| getBufferSize() | 返回缓冲区的大小 |
| getOutputStream() | 获取到客户端的输出流对象 |
| sendError(int) | 向客户端发送错误信息 |
| sendRedirect(String location) | 将响应发送到另一个位置去处理 |
| setContentType(Sting contentType) | 设置响应的 MIME 类型 |
| setHeader(String name, String value) | 设置名字为 name 的 HTTP 文件头的值，新设置的值可以覆盖旧值 |

### 3.2.1　response 实现页面转向

使用 response 对象的 sendRedirect(String location)方法可以实现页面的转向。在上一章的动作指令中，我们学过<jsp:forward>动作指令也能够实现页面的转向，那么这两种转向有什么不同呢？它们的区别如图 3.11 所示。

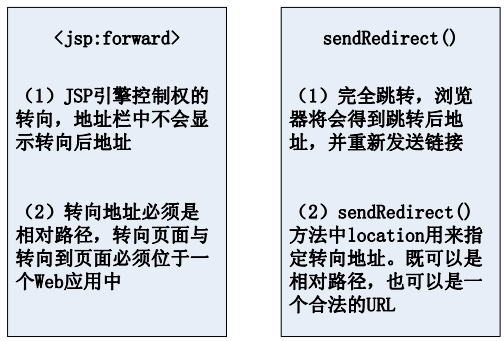



图 3.11　<jsp:forward>动作指令与 sendRedirect()方法的对比

【示例 3.4】我们举一个例子 sendRedirect.jsp 使用 response 实现页面的转向，如图 3.12 所示。

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

<html>

  <body>

   include 编译指令测试<br>

     <%  response.sendRedirect("include.jsp");  %>

  </body>

</html>
```

sendRedirect()方法

图 3.12　sendRedirect.jsp 示例

include.jsp 的代码如图 3.13 所示。

```
<%@ page language="java"  contentType="text/html;charset=ISO-8859-1"%>

<%! String str="Here is a JavaWeb!";%>

<% out.println(str+"<br>"); %>
```

图 3.13　include.jsp 示例

在浏览器中输入地址 http://localhost:8080/JSPWeb/sendRedirect.jsp，显示运行结果，如图 3.14 所示。

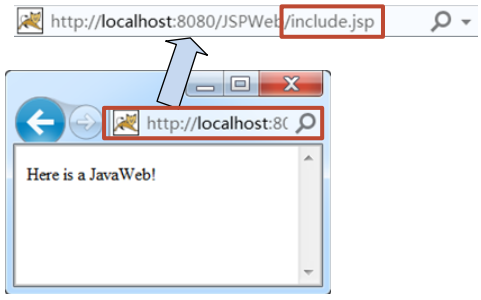



图 3.14　sendRedirect.jsp 运行结果

## 3.2.2　动态设置页面返回的 MIME 类型

在 JSP 中可以使用 page 编译指令来设置页面的 MIME（Multipurpose Internet Mail Extensions 多功能 Internet 邮件扩充服务，即文件的类型）返回类型，但是在这里设置是页面的编译阶段，以计算机设置完成，且在运行阶段是不可更改的。而使用 response 对象中的 setContentType(String type)方法可以来动态设置页面的返回类型。

【示例 3.5】我们举一个示例 setContentType.jsp 设置页面的返回类型。在该示例中包含一个文本文件，然后在运行时由用户选择页面的返回类型，如图 3.15 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<html>
  <body><font size=5>
   <p>使用什么方式显示成绩?
     <form action="show.jsp"  method="post"  name=form>
        <input type ="submit" value="word" name="submit1">
        <input type ="submit" value="excel" name="submit2">
     </form>
  </font></body>
</html>
```

图 3.15　setContentType.jsp 示例

创建一个文本文件 score.txt，在其中输入如图 3.16 所示的内容。

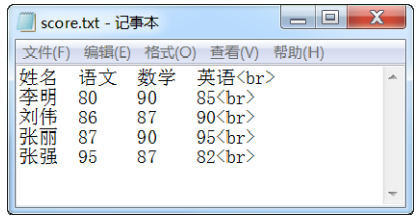



图 3.16　score.txt 示例

在 show.jsp 页面中包含该文本文件，该页面将根据获取的提交按钮的 value 值来判断用户到底是单击了哪一个按钮，并返回相应的页面类型。show.jsp 页面的具体代码定义如图 3.17 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<html>
  <body>
  <%  String str1=request.getParameter("submit1");
      String str2=request.getParameter("submit2");
      if(str1==null){
      str1="";}
      if(str2==null){
      str2="";}
      if(str1.startsWith("word")){
      response.setContentType("appplication/msword;charset=gb2312");
      }
      if(str2.startsWith("excel")){
      response.setContentType("appplication/x-msexcel;charset=gb2312");
      }
   %>
   <jsp:include page="score.txt"/>
   </body>
</html>
```

图 3.17　show.jsp 示例

在浏览器中输入地址 http://localhost:8080/JSPWeb/setContentType.jsp，显示运行结果，如图 3.18 所示。

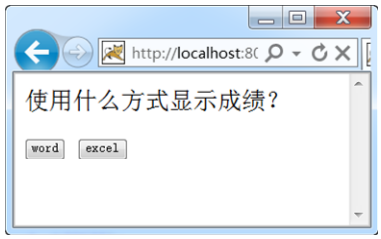



图 3.18　setContentType.jsp 运行结果

单击“word”或者“excel”按钮，JSP 页面就会以不同的文档形式返回，如图 3.19 所示。

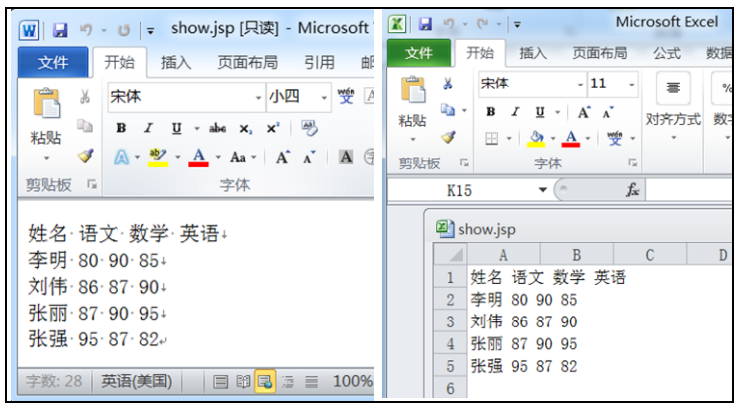



图 3.19　返回页面形式

## 3.3　out 内置对象

out 内置对象是在 Web 应用开发过程中使用最多的一个对象，其功能就是动态的向 JSP 页面输出字符流，从而把动态的内容转化成 HTML 形式来展示。这个对象在任何 JSP 页面中都可以任意访问。out 对象的方法主要用于输出各种各样格式的数据，如表 3.4 所示。

表 3.4　out内置对象常用方法

| 方　法 | 方 法 描 述 |
|---|---|
| clear() | 清除缓冲区的数据，但是仅仅是清除，并不向用户输出 |
| clearBuffer() | 清除缓冲区的数据，同时把这些数据向用户输出 |
| close() | 关闭 out 输出流 |
| flush() | 输出缓冲区的内容 |
| isAutoFlush() | 判断是否为自动刷新 |
| print(String str) | 输出带 HTML 格式的各种类型的数据，下一个输出语句不换行 |
| println(String str) | 输出带 HTML 格式的各种类型的数据，下一个输出语句换行 |

在 out 对象方法中，最常用的就是 print()和 println()方法。我们可以运用这两种方法实现各种类型数据的输出。

【示例 3.6】我们可以举一个示例 out.jsp，实现数据的输出，如图 3.20 所示。

```
<%@ page contentType="text/html; charset=UTF-8" language="Java" %>

<%
response.setContentType("text/html");
out.println("out对象的使用方法：<br><hr>");

out.println("<br>out.println(boolean):");
out.println(true);      //输出boolean类型的值

out.println("<br>out.println(char):");
out.println('A');      //输出一个字符

out.println("<br>out.println(char[]):");
out.println(new char[]{'J','a','v','a','W','e','b'}); //输出一个字符数组

out.println("<br>out.println(double):");
out.println(3.1415926d);   //输出一个double类型的数值

out.println("<br>out.println(float):");
out.println(3.15f);     //输出一个float类型的数值

out.println("<br>out.println(int):");
out.println(55);         //输出一个int类型的数值

out.println("<br>out.println(long):");
out.println(12345678L);      //输出一个long类型的数值

out.println("<br>out.println(object):");
out.println(new java.util.Date());    //输出一个日期类型的值

out.println("<br>out.println(string):");
out.println("I love JavaWeb! ");     //输出一个字符串

%>
```

图 3.20　out.jsp 示例

在浏览器中输入地址 http://localhost:8080/JSPWeb/out.jsp，显示运行结果，如图 3.21 所示。

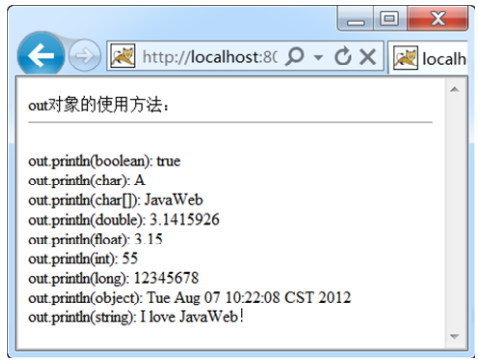



图 3.21　out.jsp 运行结果

## 3.4　session 内置对象

session 对象用来保存每个用户的信息。例如，登录名、密码、上次访问时间等，从而可以跟踪每个用户的操作状态。一般情况下，当用户首次登录系统时，Web 容器就会给该用户创建

一个唯一用来标识该用户会话的 session ID。为了跟踪用户的操作状态，在多个页面之间保存共享信息，JSP 中提供了 session 对象。当该用户退出系统时，这个 session 自动消失。session 对象的主要方法如表 3.5 所示。

表 3.5　session对象的主要方法

| 方　法 | 方 法 描 述 |
|---|---|
| getAttribute(String name) | 从 session 中获取名字为 name 的属性 |
| getAttributeNames() | 返回存储在 session 对象中的所有属性的名字，结果为一个枚举类型 |
| removeAttribute(String name) | 删除名字为 name 的属性 |
| setAttribute(String name, Java.lang.Object value) | 设置一个名字为 name 的属性，其值为 value |
| getCreationTimes() | 返回该 session 被创建的时间 |
| getId() | 返回唯一标识该 session 的 ID |
| getLastAccessedTime() | 返回与该 session 相关的客户端最后发送请求的时间 |

下面我们通过几个 session 常用实例的介绍，来详细说明这些方法的具体用法。

### 3.4.1　获取 session 的 ID

session 对象的 ID 是用来唯一识别 session 的标识。该 ID 由一个 32 位的十六进制字符串组成，可以保证服务器中所创建的所有 session 对象都不相同。

**【示例 3.7】**我们举一个例子 session1.jsp 来获取 session 的 ID。该实例通过获取一个 session 的 ID 来证明一个会话过程中的多个页面之间进行跳转时使用的是同一个 session 对象。session1.jsp 的具体代码如图 3.22 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<html>
  <body>
   <p>
     <%
     String s=session.getId();
      %>
      <p>您的session对象的ID是：
      <br>
      <%=s%>
      <p>输入用户姓名连接到session2.jsp
      <form action="session2.jsp" method=post name= form>
      <input type ="text" name ="name">
      <input type ="submit" value ="提交" name =submit>
      </form>
 </body>
</html>
```

图 3.22　session1.jsp 示例

程序通过 getId()方法获取当前页面的 session ID。在单击“提交”按钮后，将通过<form>表单跳转到 session2.jsp 页面上。session2.jsp 程序代码如图 3.23 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<html>
  <body>
   <p>session2.jsp页面
     <%
     String s=session.getId();
      %>
      <p>在session2.jsp页面中的session对象的ID是：
      <%=s%>
      <p>单击超链接，连接到session1.jsp页面
      <A href="session1.jsp">
      <br>进入session1.jsp
      </A>
 </body>
</html>
```

图 3.23　session2.jsp 程序

在浏览器中输入地址 http://localhost:8080/JSPWeb/session1.jsp，显示运行结果，如图 3.24 所示。

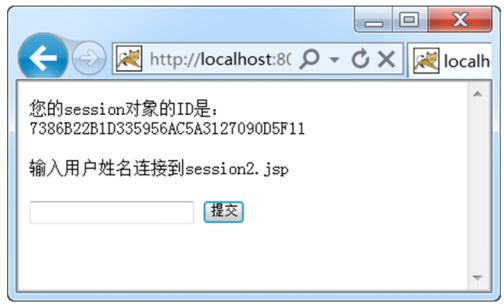



图 3.24　session1.jsp 运行结果

单击“提交”按钮，进入 session2.jsp 页面，如图 3.25 所示。

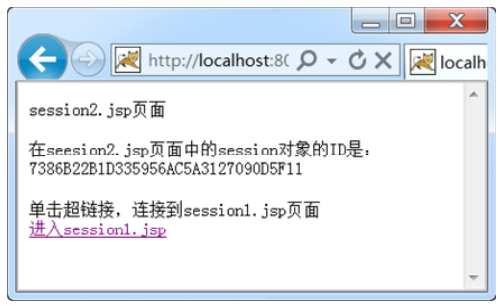



图 3.25　session2.jsp 页面

我们通过运行结果可以很容易地看出 session 对象的 ID 相同，这就说明了在一个会话过程中的多个页面之间进行跳转时使用的是同一个 session 对象。

### 3.4.2　session 中保存和读取共享数据

与 request 对象一样，session 对象也有一对 setAttribute()和 getAttribute()方法，用来存储或

者读取 session 中的共享信息。而两种对象的两个方法的区别在于共享信息的范围不同，session 对象中保存的共享信息的范围是整个会话过程，而 request 对象中保存共享信息的范围则是提交和被提交的页面。

**【示例 3.8】**我们举一个例子 sessionSetAttribute.jsp 来看 session 对象共享信息的范围，其具体代码如图 3.26 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<html>
  <body>
     <%
     session.setAttribute("name","lester");
      %>
      <A href="sessionReadAttribute.jsp">读取共享信息
      </A>
 </body>
</html>
```

图 3.26　sessionSetAttribute.jsp 示例

然后我们使用超链接转向到读取共享信息的页面 sessionReadAttribute.jsp，其具体代码如图 3.27 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<html>
  <body><font>
  读取的共享信息是：
     <%
     out.println(session.getAttribute("name"));
      %>
      </font>
 </body>
</html>
```

图 3.27　sessionReadAttribute.jsp 示例

在浏览器中输入地址 http://localhost:8080/JSPWeb/sessionSetAttribute.jsp，显示运行结果，单击超链接，转向 sessionReadAttribute.jsp 页面，如图 3.28 所示。

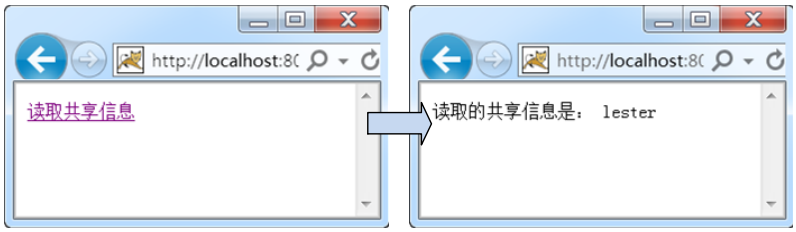



图 3.28　读取并显示 session 对象中的数据

而如果我们将上述代码中的 session 对象换成 request 对象，就会得到如图 3.29 所示的结果。


图 3.29　session 对象换成 request 对象后运行结果

通过运行结果可知，request 对象只能读取提交页面中保存的共享信息，而 session 对象则可以读取会话中存储的共享信息。

### 3.4.3　session 对象的生命周期

session 对象的创建是由服务器完成的，当客户端第一次请求服务器时由服务器创建。如果会话过程一直存在，则 session 对象也将一直存在下去。只有当 session 过期、客户端关闭浏览器或者服务器端调用了 session 的 invalIDate()方法时 session 对象才被释放掉，结束其生命周期。

**【示例 3.9】**我们来看一个查看 session 对象生命周期的例子 session.jsp，其代码如图 3.30 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<html>
<head>
<title>session生命周期</title>
</head>
  <body><h2>session生命周期</h2>
     <%
    if(session.isNew()){
    session.setMaxInactiveInterval(10);
    session.setAttribute("expire","10");//10秒内没有活动，则session过期
    out.print("设定session若10秒内没有活动则使session过期");
    }
    else{
    String expiretime =(String)session.getAttribute("expire");
    long createtime = session.getCreationTime();//取得session创建时间
    long accesstime = session.getLastAccessedTime();//取得session最后访问时间
    long currenttime = System.currentTimeMillis();//获取当前系统时间
    long existtime = (currenttime - createtime )/1000;//计算session存在时间
    out.println("session已存在"+existtime +"秒");//输出session存在时间
    if(existtime >= 30){
        out.println("session 事件已到期，自动失效");//使session失效
        session.invalidate();
        }
    }
      %>
 </body>
</html>
```

图 3.30　session.jsp 示例

在浏览器中输入地址 http://localhost:8080/JSPWeb/session.jsp，显示运行结果，该页面在不同时间段内的运行结果如图 3.31 至图 3.33 所示。

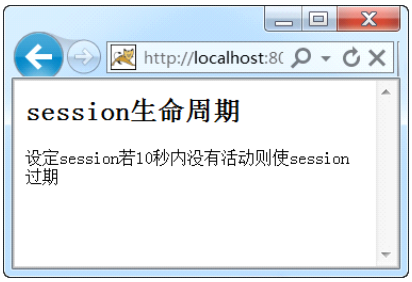



图 3.31　新建 session 对象时显示的页面信息

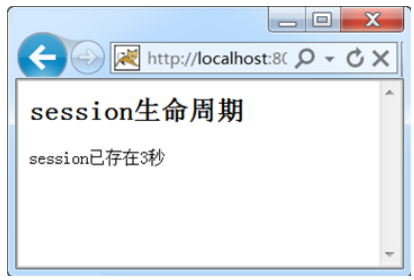



图 3.32　30 秒内显示的页面信息

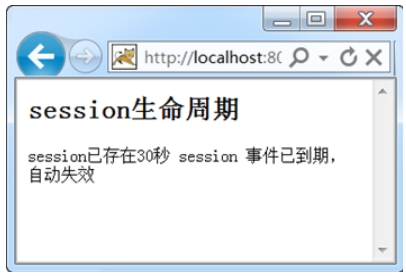



图 3.33　session 对象失效后显示的页面信息

## 3.5　application 内置对象

application 对象保存着整个 Web 应用运行期间的全局数据和信息。从 Web 应用运行开始，这个对象就会被创建。在整个 Web 应用运行期间可以在任何 JSP 页面中访问这个对象。所以如果要保存在整个 Web 应用运行期间都可以访问的数据，这时候就要用到 application 对象。

Application 对象的常用方法及方法描述如表 3.6 所示。

表 3.6　application对象常用方法

| 方　法 | 方 法 描 述 |
|---|---|
| getAttribute(String name) | 返回 application 对象中名字为 name 的属性的值 |
| getAttributeNames() | 返回 application 对象中所有属性的名字，结果为一个枚举类型 |
| getInitParameter(String name) | 返回 application 对象中名字为 name 的属性的初始值 |
| getServletInfo() | 返回 Servlet 编译器的当前版本的信息 |
| setAttribute(String name,Object object) | 在 application 对象中设置一个名字为 name 的属性，其值为 object |

application 对象最常用的方法是 getAttribute()和 set Attribute()方法。

**【示例 3.10】**我们就举一个使用 application 对象实现网站访问计数器功能的例子 application.jsp，其代码如图 3.34 所示。

在浏览器中输入地址 http://localhost:8080/JSPWeb/application.jsp，显示运行结果，如图 3.35 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
 <head>
   <title>利用application对象实现的计数器示例</title>
 </head>
 <body>
     <font size="5">
         <%
             int count=0;
             if(application.getAttribute("count")==null)
             {
                 count = count +1;
                 application.setAttribute("count",count);
             } else {
count = Integer.parseInt(application.getAttribute("count").toString());
                 count = count + 1;
                 application.setAttribute("count",count);
              }
              out.println("您是本网页的第"+count+"访问者！ ");
          %>
     </font>
 </body>
</html>
```

图 3.34　application.jsp 示例

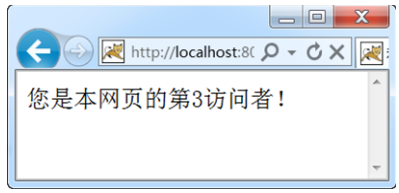



图 3.35　application.jsp 运行结果

## 3.6　其他内置对象

前面讲解的五种内置对象是在 JSP 中最为常用的对象，需要读者熟悉并运用。还有四种 JSP 内置对象使用几率较小，我们只介绍其基本用法。

### 3.6.1　pageContext 内置对象

pageContext 对象又称为 JSP 作用域通信对象。该对象提供了访问其他内置对象的统一入口，使用户可以方便地访问页面作用域中定义的所有内置对象。PageContext 对象的主要方法及方法描述如表 3.7 所示。

表 3.7　pageContext对象的主要方法

| 方　法 | 方 法 描 述 |
|---|---|
| getRequest() | 返回当前页面的 request 对象 |
| getResponse() | 返回当前页面的 response 对象 |
| getServletConfig() | 返回当前页面的 servletConfig 对象 |
| getServletContext() | 返回当前页面的 ServletContext 对象，这个对象是所有页面共享的 |
| getSession() | 返回当前页面的 session 对象 |
| setAttribute() | 设置默认页面范围或特定对象范围之中的对象 |
| removeAttribute() | 删除默认页面对象或特定对象范围之中的已命名对象 |

## 3.6.2　config 内置对象

config 对象代表当前 JSP 页面的配置信息。但 JSP 页面通常无须预先进行配置，也就不存在配置信息了。因此该对象在 JSP 页面中比较少用，但在 Servlet 中则用处相对较大，因为 Servlet 需要在 web.xml 文件中进行配置，从而设置初始化配置参数。config 对象的常用方法及方法描述如表 3.8 所示。

表 3.8　config对象的常用方法

| 方　法 | 方 法 描 述 |
|---|---|
| getInitParameter(String name) | 返回 String 类型的初始化参数 |
| getInitParameterNames(String name) | 返回所有初始化参数的名称 |
| getServletName() | 获得当前 JSP 页面名称 |
| getServletContext() | 获得当前 JSP 页面的服务器上下文环境 |

## 3.6.3　exception 内置对象

exception 对象用来封装运行时出现的异常信息。该对象只能被处理错误的页面使用，一般用来处理错误的页面会在其页面指令中声明“isErrorPage=true”。exception 对象常用的方法和方法描述如表 3.9 所示。

表 3.9　exception对象的常用方法

| 方　法 | 方 法 描 述 |
|---|---|
| getMessage() | 返回描述异常的消息 |
| toString() | 返回关于异常的简短描述消息 |
| printStackTrace() | 显示异常及其栈中的跟踪信息 |

## 3.6.4　page 内置对象

page 内置对象指向当前 JSP 页面本身，有点类似于类中的 this 指针，它表示当前 JSP 页面转换后生成的 Servlet 类的实例。page 对象的常用方法及方法描述如表 3.10 所示。

表 3.10　page对象的常用方法

| 方　法 | 方 法 描 述 |
|---|---|
| getClass() | 返回当前 Object 的类 |
| toString() | 返回当前 Object 对象的字符串 |

续表

| 方　法 | 方 法 描 述 |
| --- | --- |
| hashCode() | 返回当前 Object 的哈希代码 |
| equals(Object o) | 比较当前对象与给定的对象是否相等 |
| copy(Object o) | 把当前对象赋值到给定的对象中去 |
| clone() | 对当前对象进行克隆操作 |

## 3.7　JSP 中的中文乱码问题

在 Java 开发中，中文乱码是一个最让人头疼的问题，如果不对中文做特殊的编码处理，这些中文字符就会变成乱码或者是问号。而在不同情况下对这些乱码的处理方法又各不相同，这就导致很多初学者对中文乱码问题束手无策。其实造成这种问题的根本原因是 Java 中默认的编码方式是 Unicode，而中文的编码方式一般情况是 GB2312,因为编码格式的不同，导致在中文不能正常显示。

本节我们将对 JSP 开发过程中的中文乱码常见问题进行介绍，并提供对应的解决方法。UTF-8、GBK、GB2312 是三种支持中文显示的编码方案，在本节我们统一采用 GB2312 的编码格式。

### 3.7.1　JSP 页面中文乱码

在 JSP 页面中，中文显示乱码有两种情况：一种是 HTML 中的中文乱码，另一种是在 JSP 中动态输出的中文乱码。

**【示例 3.11】**我们先来看一个最简单的 JSP 程序 PageCharset.jsp，看会出现什么显示结果，如图 3.36 所示。

```
<%@ page language="java" import="java.util.*"%>
<html>
  <head>
    <title>中文显示示例</title>
  </head>
  <body>
    这是一个中文显示示例：
  <%
      out.print("这里是用JSP输出的中文。");
   %>
  </body>
</html>
```

运行结果

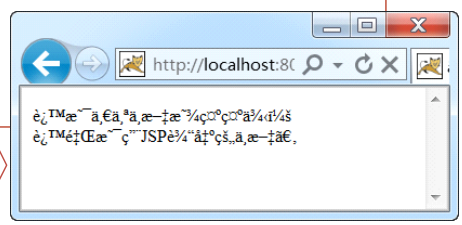



图 3.36　PageCharset.jsp 示例

显然，图 3.37 所示并非我们预期的效果，在 JSP 源代码中清清楚楚看到的是中文，在这里为什么就成了乱码，造成这种原因的可能就是出在浏览器端的字符显示设置上，我们需要对其进行如图 3.39 所示的改进。

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

↓

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

图 3.37　设置支持中文显示的编码方式

改进后，重新输入地址运行，就会正常显示运行结果，如图 3.38 所示。


图 3.38　JSP 页面正常运行效果

## 3.7.2　表单提交中文乱码

对于表单中提交的数据，可以使用 request.getParameter("")的方法获取。但是当表单中如果出现中文数据的时候就会出现乱码。这是我们在 JavaWeb 开发中经常会遇到的情况。我们可以先看如下一个例子。

**【示例 3.12】**我们采用前面提到过的用户登录程序，来看如何避免出现乱码情况，首先来看一个乱码的实例 FormCharset.jsp，如图 3.39 所示。

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

<html>

 <head>

   <title>Form中文处理示例</title>

 </head>

 <body>

 <font size="4">

 下面是表单内容：

 <form action="AcceptFormCharset.jsp" method="post">

     用户名：<input type="text" name="userName" size="10"/>

     密　码：<input type="password" name="password" size="10"/>

    <input type="submit" value="提交">

  </form>

  </font>

 </body>

</html>
```

图 3.39　FormCharset.jsp 示例

在上面这个表单中，向 AcceptFormCharset.jsp 这个页面提交了两项数据，AcceptFormCharset.jsp 的代码内容如图 3.40 所示。

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

<html>

 <head>

   <title>Form中文处理示例</title>

 </head>

 <body>

 <font size="4">

  下面是表单提交以后用request取到的表单数据：<br>

  <%

out.println("表单输入userName的值:"+request.getParameter("userName")+"<br>");

out.println("表单输入password的值:"+request.getParameter("password")+"<br>");

   %>

  </font>

 </body>

</html>
```

图 3.40　AcceptFormCharset.jsp 示例

在浏览器中输入地址 http://localhost:8080/JSPWeb/FormCharset.jsp，显示输入页面，我们在用户名中输入中文用户名和密码，单击“提交”按钮，进入响应页面如图 3.41 所示。

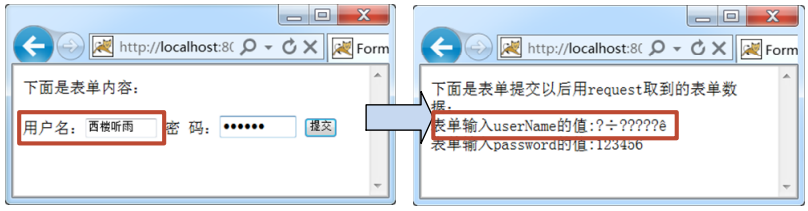



图 3.41　FormCharset.jsp 示例运行结果

我们可以从图3.41 中清楚地看到输入的中文用户名在用 request 取出以后全部变成了乱码，造成这个问题的原因是：在 Tomcat 中，对于以 POST 方法提交的表单采用的默认编码为 ISO-8859-1，而这种编码格式不支持中文字符。要解决这个问题，我们可以采用转换编码格式的方法，转换方法如图 3.42 所示。

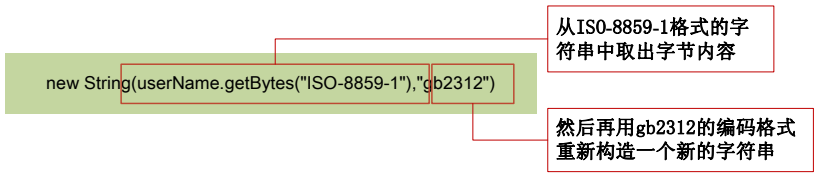



图 3.42　编码格式转化方法

我们按照这个方法来对 AcceptFormCharset.jsp 进行如图 3.43 所示的改造。

```
<%
  out.println("表单输入userName的值:"+request.getParameter("userName")+"<br>");

  out.println("表单输入password的值:"+request.getParameter("password")+"<br>");
%>
```

↓

```
<%
  String userName = request.getParameter("userName");

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

  out.println("表单输入userName的值:"+new String(userName.getBytes("ISO-8859-1"),"gb2312")+"<br>");

  out.println("表单输入password的值:"+new String(password.getBytes("ISO-8859-1"),"gb2312")+"<br>");
%>
```

图 3.43　转换编码格式

改进以后的程序运行结果如图 3.44 所示。


图 3.44　JSP 页面正常运行效果

## 3.7.3　URL 传递参数中文乱码

在一般情况下，我们可以采用类似 http://localhost:8080/JSPWeb/URLCharset.jsp?param='中文'这种形式来传递参数。但是这种传递方式仍然有可能会发生乱码问题。下面就是采用 URL 传递参数的一个示例程序。

**【示例 3.13】**我们通过 URLCharset.jsp 示例页面中的一个超链接向页面中输入“中文”字样，如图 3.45 所示。

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

<html>

 <head>

   <title>URL传递参数中文处理示例</title>

 </head>

 <%
      String param = request.getParameter("param");
  %>

 <body><font size="4">

     <a href="URLCharset.jsp?param='中文'">请单击这个链接</a><br>

     你提交的参数为：<%=param%>

  </font></body>

</html>
```

图 3.45　URLCharset.jsp 示例

在浏览器中输入地址 http://localhost:8080/JSPWeb/URLCharset.jsp，显示输入页面，单击页面中的超链接，就会出现如图 3.46 所示的情况。

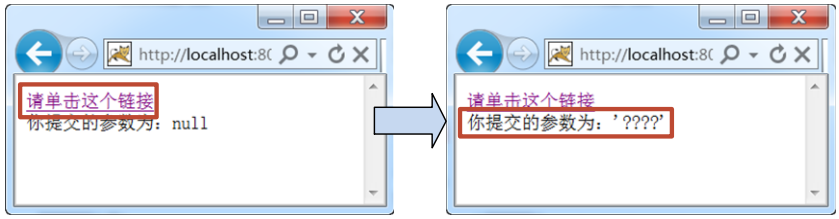



图 3.46　URLCharset.jsp 示例乱码的运行结果

对于 URL 传递中文参数乱码这个问题，其处理方法比较独特，仅仅转换这个中文字符串的编码，或者设置 JSP 页面显示编码都是不能解决问题的。在这里需要多 Tomcat 服务器的配置文件进行修改才可以解决问题，需要修改 Tomcat 的 conf 目录下的 server.xml 配置文件。修改方法是在 port="8080"后面添加 URI 编码设置 URIEncoding="gb2312"即可，如图 3.47 所示。

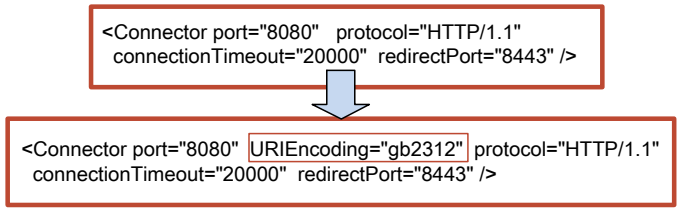



图 3.47　对 server.xml 配置文件的修改

修改完成后，重新启动 Tomcat 服务器。输入 URLCharset.jsp 地址，单击超链接查看系统运行结果，如图 3.48 所示。

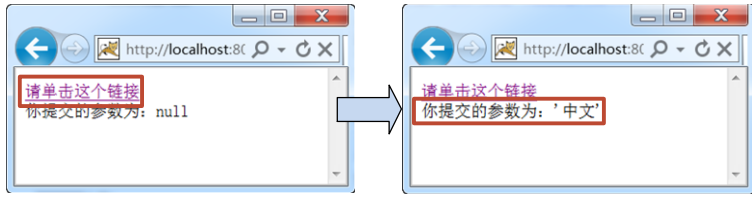



图 3.48　JSP 页面正常运行效果

### 3.7.4　MyEclipse 开发工具中文 JSP 文件的保存

在 Eclipse 中，JSP 文件默认的编码格式为 ISO-8859-1，所以在 JSP 代码中间如果出现中文就不能保存。

**【示例 3.14】**比如我们可以举一个示例 zhongwen.jsp 来看包含中文字样的 JSP 文件该如何保存，如图 3.49 所示。

对与这个问题，只要在 JSP 页面中指明页面编码即可。pageEncoding="gb2312"指明了 JSP 页面编码采用 gb2312，这样就可以正常保存 JSP 的源文件。修改方式如图 3.50 所示。

```
<%@ page language="java" import="java.util.*" %>
<html>
 <head>
   <title>中文显示示例</title>
 </head>
 <body>
    这是一个中文显示示例：
 </body>
</html>
```

运行结果

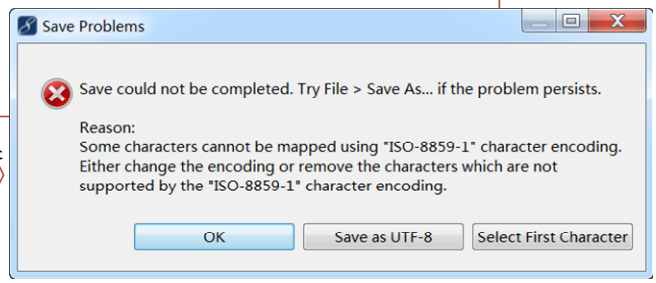



图 3.49　JSP 页面中文不能保存提示信息

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

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
```

图 3.50　指明页面编码

## 3.8　小结

本章主要讲解 JSP 内置对象的内容，包括了 JSP 九大对象，其中重点讲解了前 5 种类型。由于在 JSP 编码中读者经常会遇到中文乱码问题，所以我们又增加了一节对于这个问题解决方法的讲解。本章的重点是对 request、response、session、out 这些重要内置对象的理解，难点是能在理解的基础上实现对这些对象的熟练运用。希望读者多加练习，在今后的 Web 开发中正确地掌握和灵活地使用 JSP 内置对象。

## 3.9　本章习题

1．请读者参照示例 3.1 运用 request 对象建立一个网页，用来接收客户端提交的注册表信息。执行效果如图 3.51 所示。

【分析】本题主要考查 request 内置对象的运用。request 对象就是用来封装用户提交的表单信息的，我们也把表单提交给这个页面自身，并在这个页面中取出提交的表单数据。

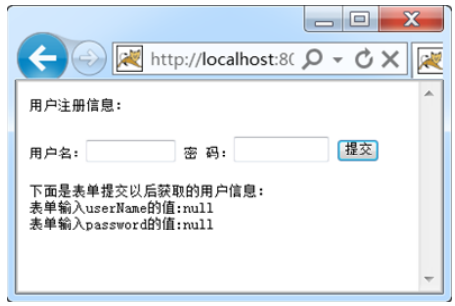



图 3.51　运行结果

【核心代码】本题的关键代码如下所示。

```
<%@ page language="java" import="java.util.*" contentType="text/html;charset=
gb2312"%>
<html>
 <body>
 <font size="2">
 用户注册信息：
 <form action="Form.jsp" method="post">
     用户名：<input type="text" name="userName" size="10"/>
     密　码：<input type="password" name="password" size="10"/>
    <input type="submit" value="提交">
 </form>
 下面是表单提交以后获取的用户信息：<br>
 <%
out.println("表单输入 userName 的值:"+request.getParameter("userName")+"<br>");
out.println("表单输入 password 的值:"+request.getParameter("password")+"<br>");
  %>
 </font>
 </body>
</html>
```

2．response 的用法很多，在这里我们请读者用 response 来实现向新浪网（http://www.sina.com）的重定向，执行效果如图 3.52 所示。


图 3.52　运行结果

【分析】本题主要考查 response 内置对象中 sendRedirect 方法的运用。我们可以参考示例 3.4 来编写代码，实现向新浪网的跳转。

【核心代码】本题的关键代码如下所示。

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

```
<html>
 <body>
    <%
       response.sendRedirect("http://www.sina.com");
     %>
 </body>
</html>
```

3．请读者通过在 application 对象中存放一个 count 属性，来统计一个页面被访问的次数并输出被访问信息，执行效果如图 3.53 所示。


图 3.53　运行结果

【分析】本题主要考查 application 内置对象的运用。我们可以参考示例 3.10 来编写代码，实现对网页浏览次数的统计。

【核心代码】本题的关键代码如下所示。

```
<html>
 <body>
    <font size="2">
       <%
          int count=0;
          if(application.getAttribute("count")==null)
          {
            count = count +1;
            application.setAttribute("count",count);
          }else
           {
             count=Integer.parseInt(application.getAttribute("count").toString());
             count = count + 1;
             application.setAttribute("count",count);
           }
          out.println("本网页已经被访问"+count+"次！");
        %>
    </font>
 </body>
</html>
```

# 第4章 JavaBean 基础

JavaBean 是一种 Java 语言写成的可重用组件，JSP 可以方便地支持 JavaBean 组件的使用。用户将常用的功能写入 JavaBean，当用户需要使用这些功能时，直接在 JSP 页面调用对应的 JavaBean 即可。实现了一次编写，任何地方调用。本章将详细讲解如何编写 JavaBean，以及 JSP 如何调用 JavaBean，最后我们会为大家展示 JavaBean 在 Web 领域的具体应用。

## 4.1 创建 JavaBean

Sun 公司对 JavaBean 的定义为可以重复利用的软件组件，它在遵循 JavaBean 技术规范的基础上提供特定的功能，这些功能模块可以组合成更大规模的应用系统。JavaBean 其实本质上就是一个封装了一系列属性和方法的类。其中属性和方法封装需要遵循特定的规范。本节将讲解如何创建 JavaBean。

### 4.1.1 JavaBean 类

首先我们要创建一个 JavaBean 类。JavaBean 类创建的语法格式如图 4.1 所示。

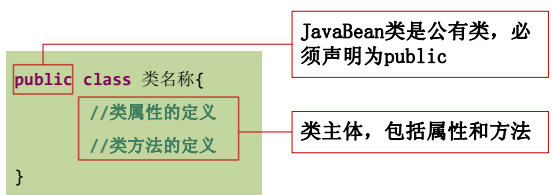



图 4.1 JavaBean 类语法格式

一个标准的 JavaBean 类有以下几个特性。

- 它是一个公开的（public）类。
- 它有一个默认的构造方法，也就是不带参数的构造方法（在实例化 JavaBean 对象时，需要调用默认的构造方法）。
- 它提供 getXXX()和 setXXX()方法来让外部程序设置和获取 JavaBean 的属性。

一般来说，符合上述条件的类，我们都可以将其看做 JavaBean 类。

明白了如何创建 JavaBean 类后，我们再来看如何创建 JavaBean 的属性和方法。

### 4.1.2 JavaBean 属性和方法

JavaBean 的属性用于表示其内部状态。在 Java Web 开发中，其属性主要用来存储中间数

据。JavaBean 属性定义如图 4.2 所示。

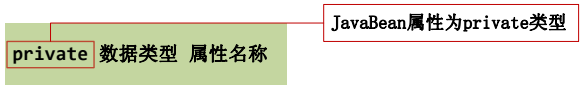



图 4.2　JavaBean 属性定义格式

对于我们在 JavaBean 中生命的属性，在类中必须定义用来获取或更改属性值的两个方法——getXXX()和 setXXX()方法。

JSP 文件就运用 JavaBean 方法在需要时从 JavaBean 中把这些属性取出，然后在客户端将其显示出来。根据 JavaBean 类特定的接口格式要求我们可以将其属性分为简单方法和索引方法。

### 1．简单方法

简单的方法是指一个拥有 get 或者 set 方法的方法。我们在 Java Web 开发中使用的 JavaBean 属性一般都是读/写类型，必须采用标识命名约定来定义 getXXX()和 setXXX()方法。对于布尔类型的值还可以采用 is()属性来获取属性值。简单方法的使用语法如图 4.3 所示。

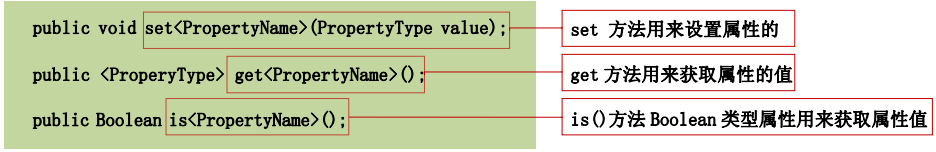



图 4.3　简单方法使用语法

**【示例 4.1】**我们下面通过一个例子来看 JavaBean 程序中简单方法是如何运用的，例如我们编写一个 ProductBean.java，代码如图 4.4 所示。

```
public class ProductBean
{
        //产品的名字name，类型是String
        private String name;
        //产品是否以生产好done，类型为boolean
        private boolean done=false;
        //getXXX方法，返回这个属性的值
        public String getName()
        {
                return this.name;
        }
        //setXXX方法，设置这个属性的值
        public void setName(String name)
        {
                this.name=name;
        }
        //对于boolean类型的属性，可以使用isXXX方法来获得属性
        public boolean isDone()
        {
                return this.done;
        }
        //设置boolean类型的属性
        public void setDone(boolean done)
        {
                this.done=done;
        }
}
```

图 4.4　ProductBean.java 示例

### 2．索引方法

索引方法是指一个有 get/set 方法的数组方法。get 和 set 方法的作用同简单类型的方法一样，即用来获取和设置属性值。但是索引方法不只有一个 get 或者 set 方法，可能有两个 get 方法，但是参数不一样。索引方法的语法格式如图 4.5 所示。

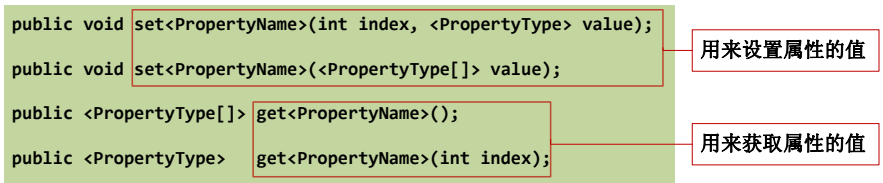



图 4.5　索引方法的语法格式

【示例 4.2】我们再来看 JavaBean 程序中索引方法是如何运用的，例如我们可以编写一个 CategoryBean.java，代码如图 4.6 所示。

```
public class CategoryBean
{
        // product为属性的名字，类型是String[]
        private String[] product=
        new String[]{"product1","product2","product3","product4"};
        //getXXX方法，返回这个属性的值
        public String[] getProduct()
        {
                return this.product;
        }
        //setXXX方法，设置这个属性的值
        public void setProduct(String[] product)
        {
                this.product=product;
        }
        //另外的设置属性和获得属性值的方法
        public void setProduct(int index,String value)
        {
                product[index]=value;
        }
        public String getProduct(int index)
        {
                return product[index];
        }
}
```

图 4.6　CategoryBean.java 示例

## 4.2　JSP 与 JavaBean 交互的动作指令

在 JSP 中专门提供了 3 个动作指令来与 JavaBean 进行交互，分别为<jsp:useBean>动作指令、<jsp:setProperty>动作指令和<jsp:getProperty>动作指令。

### 4.2.1　<jsp:useBean>动作指令

<jsp:useBean>动作指令用来在 JSP 页面中获取或创建一个 JavaBean 组件的实例并指定它

的名字和作用范围，<jsp:useBean>动作指令的语法形式如图 4.7 所示。

```
<jsp:useBean id="name" scope="page|request|session|application" class="className"/>
```

**或者**

```
<jsp:useBean id="name" scope="page|request|session|application" class="className">
 body  //执行语句
</jsp:useBean>
```

图 4.7　<jsp:useBean>动作指令语法形式

该动作指令表示的含义是在页面中引用一个已经存在或创建一个新的由 class 属性指定的 Java 类的实例，然后将其绑定到名字由 id 属性给出的变量上，并且该变量只在 scope 属性所指定的范围内有效。对于第二种形式，只有当第一次实例化 JavaBean 时，才执行 body（JSP 语句）部分，如果是获取现有的 JavaBean 实例，则不执行 body 部分。然后我们看一下这几个属性的作用，如表 4.1 所示。

表 4.1　<jsp:useBean>动作指令属性的作用

| 属 性 名 | 属 性 作 用 |
|---|---|
| id 属性 | 在定义范围内确认 JavaBean 实例变量，也可以用该变量名引用 JavaBean 实例 |
| class 属性 | 引用的 JavaBean 的完整类名。JSP2.0 规范要求 JavaBean 必须要有包名 |
| scope 属性 | JavaBean 存在范围及 id 变量名有效范围。范围由小到大依次为：page、request、session 和 application。默认为 page |

**【示例 4.3】**下面我们通过一个完整的示例来看 JavaBean 中的<jsp:useBean>动作指令是如何使用的。首先我们先来定义一个 JavaBean 类，用 Student.java 来定义学生的姓名和年龄属性，具体代码如图 4.8 所示。

```
package com.yzty.xue;
public class Student {
        private String name;
        private int age;
        public String getName(){
                return name;
        }
        public void setName(String name){
                this.name=name;
        }
        public int getAge(){
                return age;
        }
        public void setAge(int age){
                this.age=age;
        }
}
```

声明变量属性

name属性的get和set方法

age属性的get和set方法

图 4.8　Student.java 示例

然后编写一个引入 JavaBean 的 JSP 页面 useBean.jsp，具体代码及运行结果如图 4.9 所示。


图 4.9　useBean.jsp 示例及运行结果

## 4.2.2　<jsp:getProperty>动作指令

在 JSP 页面中我们可以通过<jsp:getProperty>和<jsp:setProperty>动作指令来代替一般的 get 和 set 方法。<jsp:getProperty>动作指令用来获取 JavaBean 中指定的属性值并将其转化为一个字符串，然后将其输出到页面中。即其作用相当于前面提到的 getXXX()方法。

<jsp:getProperty>动作指令的语法格式如图 4.10 所示。

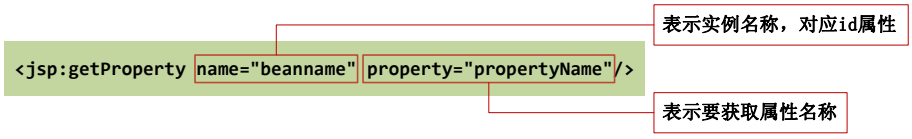



图 4.10　<jsp:getProperty>动作指令语法格式

值得一提的是，在使用<jsp:getProperty>动作指令之前，必须使用<jsp:useBean>动作指令来获取或者创建 JavaBean 实例。

**【示例 4.4】**我们把示例 4.3 中的 useBean.jsp 示例加以修改得到 useBean2.jsp，进行<jsp:getProperty>动作指令的测试，如图 4.11 所示。

## 4.2.3　<jsp:setProperty>动作指令

<jsp:setProperty>动作指令用来设置已经实例化的 JavaBean 对象的属性值。实际上，该动作指令作用即相当于获取属性值的 setXXX()方法。<jsp:setProperty>动作指令有 3 种不同的语法形式。

### 1．通过表达式或字符串常量设置属性

这种形式的具体语法格式如图 4.12 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312" %>
<html>
 <body><font size=4>
  <%--引入JavaBean--%>
   <jsp:useBean id="student" class="com.yzty.xue.Student" scope="page" />
    <%student.setAge(20);
      student.setName("李默"); %>
    <p>名字是：<jsp:getProperty name="student" property="name"/> </p>
    <p>年龄是：<jsp:getProperty name="student" property="age"/> </p>
  </font>
 </body>
</html>
```

引入JavaBean的属性值

运行结果


图 4.11　useBean2.jsp 示例

```
< jsp:setProperty name="beanname"
  property="propertyName"
  value="<%=expression%>|字符串"/>
```

表示实例名称，对应id属性

表示要获取属性名称

表示设置属性值表达式或字符串常量

图 4.12　通过表达式或字符串常量设置属性

【示例 4.5】我们将 useBean.jsp 示例加以修改得到 useBean3.jsp，进行<jsp:setProperty>动作指令的测试，如图 4.13 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312" %>
<html>
 <body><font size=4>
  <%--引入JavaBean--%>
   <jsp:useBean  id="student" class="com.yzty.xue.Student" scope="page" />
   <jsp:setProperty  name="student" property="name" value="李默"/>
   <jsp:setProperty  name="student" property="age" value="20"/>
   <p>名字是：<jsp:getProperty name="student" property="name"/> </p>
   <p>年龄是：<jsp:getProperty name="student" property="age"/> </p>
  </font>
 </body>
</html>
```

运行结果


图 4.13　useBean3.jsp 示例

### 2．通过内置对象 request 传递的参数值设置属性

在实际应用中，直接使用表达式或字符串常量设置值的情况很少，往往都是通过接收用户

请求中传递的参数值来设置 JavaBean 属性的。该形式的具体语法如图 4.14 所示。

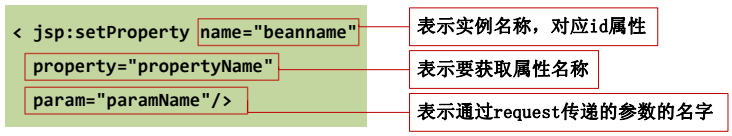



图 4.14　通过内置对象 request 传递的参数值设置属性

**【示例 4.6】**我们通过一个完整的示例来看<jsp:setProperty>动作指令的用法。我们创建一个添加输入学生信息的页面 studentInfo.jsp，具体代码如图 4.15 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312" %>
<html>
 <head>
   <title>输入学生信息</title>
 </head>
 <body><font size=4>
   <form action="useBean4.jsp" Method="post">
   <p>
   输入学生的姓名：<input type=text name="name">
   <p>
   输入学生的年龄：<input type=text name="age">
   <input type =submit value="提交">
   </form>
 </body>
</html>
```

跳转到useBean4.jsp页面

输入学生信息

图 4.15　studentInfo.jsp 示例

我们对 useBean3.jsp 示例加以修改，使其通过内置对象 request 传递的参数值设置属性 useBean4.jsp，如图 4.16 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312" %>
<html>
 <body><font size=4>
  <%--引入JavaBean--%>
   <jsp:useBean  id="student" class="com.yzty.xue.Student" scope="page" />
   <jsp:setProperty  name="student" property="name" param="name"/>
   <jsp:setProperty  name="student" property="age" param="age"/>
   <p>名字是：<jsp:getProperty name="student" property="name"/> </p>
   <p>年龄是：<jsp:getProperty name="student" property="age"/> </p>
  </font>
 </body>
</html>
```

运行结果

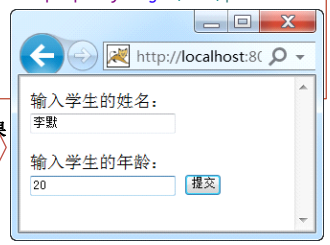



图 4.16　useBean4.jsp 示例

我们单击“提交”按钮，就会显示如图 4.17 所示的运行界面。


图 4.17　studentInfo.jsp 示例运行结果

### 3．通过表单的提交参数设置属性

这种形式的具体语法形式如图 4.18 所示。

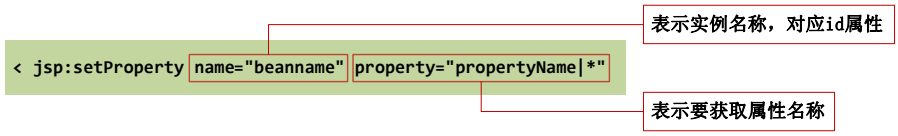



图 4.18　通过表单的提交参数设置属性

这种形式省略了第二种方式中的 param 属性。但要求表单中参数名字必须与 JavaBean 中的名字一致。

**【示例 4.7】**我们将 useBean4.jsp 中的<jsp:setProperty>动作指令进行修改，得到 useBean5.jsp。将 studentInfo.jsp 示例中跳转页面进行修改后执行。其具体代码及运行结果如图 4.19 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312" %>
<html>
 <body><font size=4>
  <%--引入JavaBean--%>
   <jsp:useBean  id="student" class="com.yzty.xue.Student" scope="page" />
   <jsp:setProperty  name="student" property="name"/>
   <jsp:setProperty  name="student" property="age" />
   <p>名字是：<jsp:getProperty name="student" property="name"/> </p>
   <p>年龄是：<jsp:getProperty name="student" property="age"/> </p>
  </font>
 </body>
</html>
```


图 4.19　useBean5.jsp 示例

## 4.3　JavaBean 的应用

在 Web 应用中，我们经常要用到 JavaBean，其中最常用的有两种——计数器和数据库应用。有关数据库的具体知识我们将在后面单独一章为大家介绍。本节先来介绍这两种功能的简

单应用。

### 4.3.1 计数器 JavaBean

对于一个 Web 应用来说，计数器的功能几乎是必不可少的。接下来我们就为大家介绍如何应用 JavaBean 实现一个简单的计数器。

**【示例 4.8】**我们选用在 application 中存储计数器的值，使用 JavaBean 来实现 Counter.java。具体代码如图 4.20 所示。

```
package com.yzty.xue;
public class Counter {
    //定义计数器变量
private long counter;
    //取出计数器的值
    public long getCounter() {
              return counter;
    }
    //对计数器赋值，每次加1
    public void setCounter(long counter) {
              this.counter = counter + 1;
    }
}
```

图 4.20　Counter.java 示例

这个 JavaBean 的功能是定义一个计数器变量，并且给出这个变量的取值和赋值的方法。我们再创建一个 JSP 文件用 Counter.jsp 来调用它，具体调用方法参考如图 4.21 所示的程序代码。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<html>
 <body>
  <jsp:useBean id="counter" class="com.yzty.xue.Counter"
      scope="application"></jsp:useBean>
   <%
      if(session.isNew())
         {
            long temp = counter.getCounter();
            counter.setCounter(temp);
         }
   %>
   <font size=4><strong> 本页面展示的功能是利用JavaBean实现的计数器：</strong><br>
    欢迎光临！你是本网站的第
   <jsp:getProperty name="counter" property="counter" /> 访客<br>
  </font>
 </body>
</html>
```

判断当前用户是否为新的会话，如果是调用计数器的赋值方法

先取出上次计数器的值
调用计数器的赋值方法
对计数器进行加1操作

运行结果


图 4.21　Counter.jsp 示例及运行结果

**注意：**在这个计数器中，刷新页面不会改变计数器的值，只有新打开一个浏览器窗口这时候才会使计数器的值增加，而且因为这个 JavaBean 的作用范围是 application，所以只要服务器在运行，这个计数器的值都会保存在服务器中，当服务器关闭时这个值会被置零。

### 4.3.2　数据库应用

JavaBean 同样可以使用到数据库开发中，从而简化开发过程，提高代码的可重用性。接下来将要介绍的内容就是利用 JavaBean 封装数据库操作。我们首先通过一个例子来说明如何实现 JavaBean 操作数据库。

**【示例 4.9】**这个例子创建一个用户注册系统，JavaBean 用来处理业务逻辑，以及用来接收表单数据。由于这个例子要用到一个 userinfo 表来存储用户的信息，所以，首先在 mysqltest 数据库下创建该表，创建过程如图 4.22 所示。

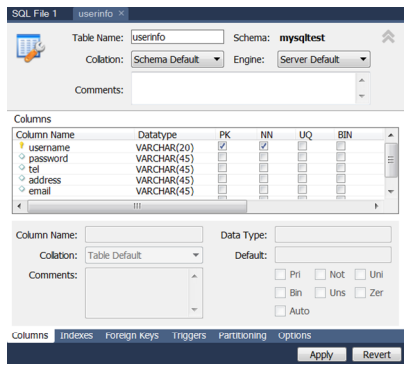



图 4.22　在 MySQL 数据库中创建表

然后我们创建一个 userInfo.java，用于接收用户输入的信息，具体代码如图 4.23 所示。

```
public class UserInfo {
        private String username;
        private String password;
        private String tel;
        private String address;
        private String email;

        public String getAddress() {
                return address;
        }
        public void setAddress(String address) {
                this.address = address;
        }
        …
        public String getEmail() {
                return email;
        }
        public void setEmail(String email) {
                this.email = email;
        }
        …
}
```

userInfo的所有属性

各属性的set()和get() 方法

图 4.23　userInfo.java 示例

我们可以创建一个 JSP 文件 userRegist.jsp 来接收用户输入的表单数据，然后把这个对象传递给 AddUser 类去向数据库插入数据。首先来看 addUser.java，它用来完成用户注册的业务处理，addUser.java 代码如图 4.24 所示。

```
package com.yzty.xue;
import java.sql.*;
public class addUser {
    private Connection con;
    private userInfo userInfo;
    public addUser(){
            String CLASSFORNAME="com.mysql.jdbc.Driver";
            String SERVANDDB="jdbc:mysql://127.0.0.1:3306/mysqltest";
            String USER="root";
            String PWD= "123456";
        }
        public void setUserInfo(userInfo userInfo){
                this.userInfo = userInfo;
        }
        public void regist(){
                String reg="insert into userinfo values(?,?,?,?,?,?,?)";
                try{
                    PreparedStatement pstmt = con.prepareStatement(reg);
                    pstmt.setString(1, userInfo.getusername());
                    pstmt.setString(2, userInfo.getpassword());
                    pstmt.setString(3, userInfo.gettel());
                    pstmt.setString(4, userInfo.getaddress());
                    pstmt.setString(5, userInfo.getemail());
                    pstmt.executeUpdate();
                } catch(Exception e){
                        e.printStackTrace();
                }
        }
}
```

连接MySQL数据库

给各参数赋值

图 4.24　addUser.java 示例

首先，创建一个 register.jsp 用于显示用户注册的页面，这个文件没有什么特殊的地方，当单击“提交”按钮时，它把请求转发给 userRegist.jsp，userRegist.jsp 页面负责控制程序的流程，它接收请求、操作数据、调用 AddUser 去操作数据库，其代码如图 4.25 所示。

```
<%@ page contentType="text/html;charset=gb2312" language="Java"
import="com.yzty.xue.*"%>
<jsp:useBean id="userInfo" class="com.yzty.xue.userInfo" scope="page">
<jsp:setProperty name="userInfo" property="*"/>
</jsp:useBean>
<jsp:useBean id="addUser" class="com.yzty.xue.addUser" scope="page"/>
<%      addUser.setUserInfo(userInfo);
        addUser.regist();
        out.println("注册成功！");
%>
```

图 4.25　userRegist.jsp 示例

在 Web 浏览器中输入：http://localhost:8080/JavaBean/register.jsp，在显示的注册页面中输入以下信息，单击“提交”按钮，进行注册，运行结果如图 4.26 所示。然后打开数据库，看 usreinfo 表中是否添加了一条记录。

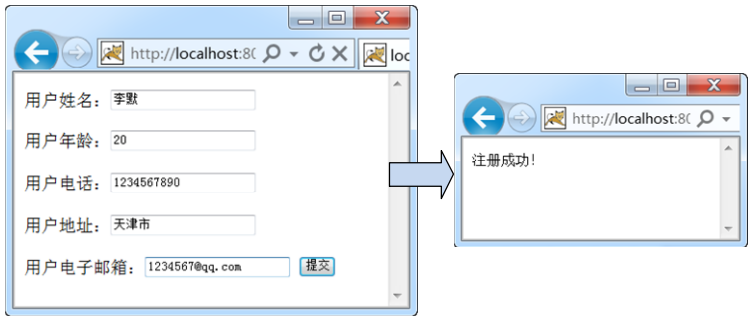



图 4.26　register.jsp 示例运行结果

## 4.4　小结

本章主要介绍了 JavaBean 的属性和方法，并在此基础上介绍了 JSP 中与 JavaBean 交互的 3 个动作指令的具体用法。最后通过实例讲解了 JavaBean 作为计数器和在数据库中的应用。本章的重点是了解 3 个动作指令的用法，难点是能够熟练掌握 JavaBean 在 Web 中，尤其是数据库中的应用。熟练掌握并运用 JSP+JavaBean 模式进行 Web 应用的开发，是目前 JSP 技术的基本要求，所以读者要多加练习，以打好 JSP 编程的基础。

## 4.5　本章习题

1．请读者创建一个 AddTeacher.jsp 页面，在本页面中实现对教师姓名、性别、年龄以及职称的输入。执行效果如图 4.27 所示。

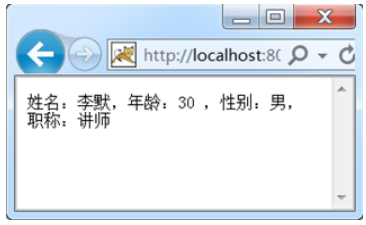



图 4.27　运行结果

【分析】本题主要考查读者运用 JSP 中 3 个动作指令来与 JavaBean 进行交互的能力。在这个题目中，我们主要要清楚如何综合应用 3 个动作指令来实现各自的功能，并通过设置其实例化对象的属性来获得属性值。

【核心代码】本题的关键代码如下所示。

Teacher.java：

```
package bean;
public class Teacher {
    private String name;            // 姓名
```

```
    private int age;                // 年龄
    private boolean sex;            // 性别
    private String professional;    // 职称
    /**
     * 省略各属性的 setter 和 getter 方法
     */
}
```

AddTeacher.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<html>
    <body>
        <%-- 通过 useBean 动作指令调用 JavaBean --%>
        <jsp:useBean id="teacher" scope="page" class="bean.Teacher"></jsp:useBean>
        <%
            teacher.setName("李默");  //设置姓名
            teacher.setAge(30);      //设置年龄
            teacher.setSex(true);    //设置为男性
            teacher.setProfessional("讲师"); //设置职称
         %>
         <%
        //输出各属性
         %>
    </body>
</html>
```

2．请读者运用 JSP 中专门提供的 3 个动作指令来与 JavaBean 进行交互，对第一题进行修改，在页面中实现对教师姓名、性别、年龄以及职称的输入。执行效果如图 4.28 所示。

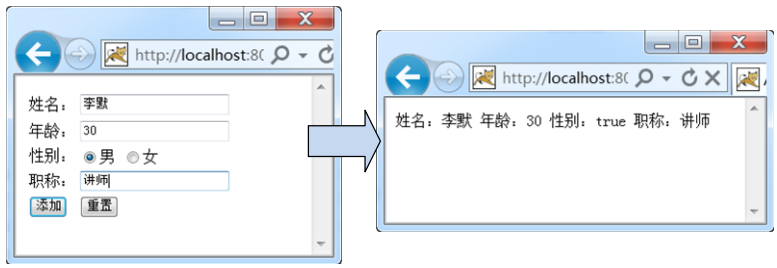



图 4.28　运行结果

【分析】本题主要考查读者在 JSP 中使用 JavaBean 的能力。在这个题目中，我们主要要清楚如何在 JSP 中调用 JavaBean，并通过设置其实例化对象的属性来获得属性值。

【核心代码】本题的关键代码如下所示。

AddTeacher1.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
    <body>
        <%
            request.setCharacterEncoding("gb2312");     //设置参数编码格式
         %>
        <jsp:useBean id="teacher" scope="page" class="bean.Teacher"></jsp:useBean>
        <jsp:setProperty name="teacher" property="*"/><%-- 设置 JavaBean 实例化对象的属性--%>
         姓名：<jsp:getProperty name="teacher" property="name"/>
        ……
```

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

TeacherForm.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
    <form action="AddTeacher1.jsp" method="post">
        <table>
            <tr>
                <td>姓名：</td>
                <td><input type="text" name="name"/></td>
            </tr>
            ……
            <tr>
                <td><input type="submit" value="添加"></td>
                <td><input type="reset" value="重置"></td>
            </tr>
        </table>
    </form>
  </body>
</html>
```

# 第5章 Servlet 编程

Servlet 是 Java Web 程序的核心。JSP 和几乎所有的 Java Web 框架（如 Struts、Webwork）在底层的实现都会看到 Servlet 的影子。因此，充分了解 Servlet 的原理和使用方法，对于以后学习 Struts 等 Web 框架将起到非常大的帮助。本章我们将为大家介绍 Servlet 的基础知识，并通过具体的示例介绍 Servlet 的强大功能。

## 5.1 Servlet 基础

Servlet 在本质上就是 Java 类。编写 Servlet 需要遵循 Java 的基本语法，但是与一般 Java 类有所不同的是，Servlet 是只能运行在服务器端的 Java 类，而且必须遵循特殊的规范，在运行的过程中有自己的生命周期。

### 5.1.1 什么是 Servlet

Servlet 是运行于服务器端的、按照其自身规范编写的 Java 应用程序。我们可以用图 5.1 来解释这个概念。

| | |
|---|---|
| Java类 | Servlet是用Java语言编写的，遵守所有Java语言的语法规则的Java类 |
| 服务器端运行 | Servlet是在服务器端运行的。它编译后的“.class”文件被服务器端调用和执行 |
| 必须调用Java Servlet API | Servlet必须调用Java Servlet API，必须是对特定类或接口的继承或实现。并且，它必须重写特定的方法去处理客户端请求 |

图 5.1 Servlet 概念的 3 大特点

Servlet 的主要功能是用来接受、处理客户端请求，并把处理结果返回到客户端显示。其作用过程如图 5.2 所示。

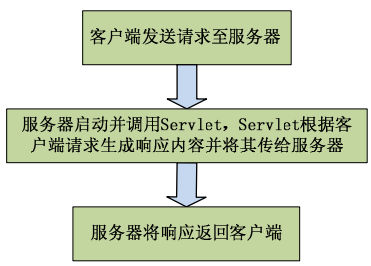



图 5.2 Servlet 的工作模式

### 5.1.2　Servlet 的生命周期

Servlet 需要在特定的容器中才能运行，这里所说的容器即 Servlet 运行时所需的运行环境。一般情况下，市面上常见的 Java Web Server 都可以支持 Servlet，例如 Tomcat、Resin、Weblogic、WebSphere 等，在本书中采用 Tomcat 作为 Servlet 的容器，由 Tomcat 为 Servlet 提供基本的运行环境。

Servlet 的生命周期指的是 Servlet 从被 Web 服务器加载到它被销毁的整个生命过程。这个过程如图 5.3 所示。

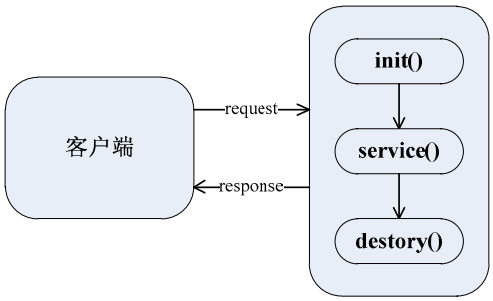



图 5.3　Servlet 生命周期的执行过程

从图 5.3 中我们可以看出，Servlet 生命周期的执行大致分为 4 个步骤，如图 5.4 所示。

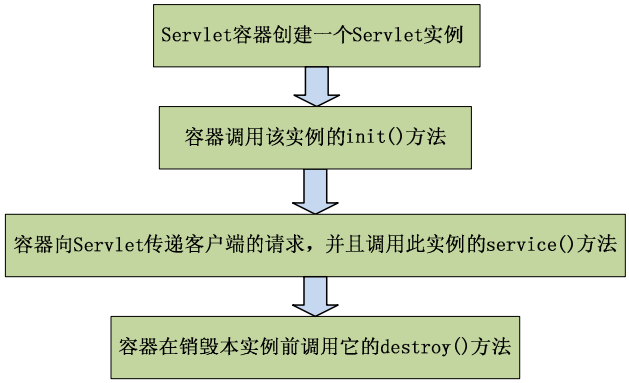



图 5.4　Servlet 生命周期的执行步骤

在以上几个阶段中，Servlet 对外提供服务阶段是最重要的。service()方法是编程人员真正要关心的方法。因为它才是 Servlet 真正开始响应客户端请求，并且处理业务逻辑的方法。service()接收到客户端请求后，再调用该 Servlet 相应的方法去处理请求。所以程序员在编写自己的 Servlet 时，一般只需要重写方法。在该方法中去处理客户端请求，并把处理结果返回。

## 5.2　简单 Servlet 开发配置示例

在这一节中我们主要应用 Servlet 编写一个 HelloWorld 程序，实现向客户端浏览器中输出“HelloWorld”信息，具体的步骤如下所示。

首先我们启动 MyEclipse 软件，创建一个 Web 工程 Servlet。然后我们右击这个工程，选

择“New”→“Servlet”命令，打开 Create a new Servlet 对话框。在 Package 文本框中输入包名“servlets”，在 Name 文本框中输入 Servlet 名称“HelloWorld”，并选择 doGet 等复选框，如图 5.5 所示。

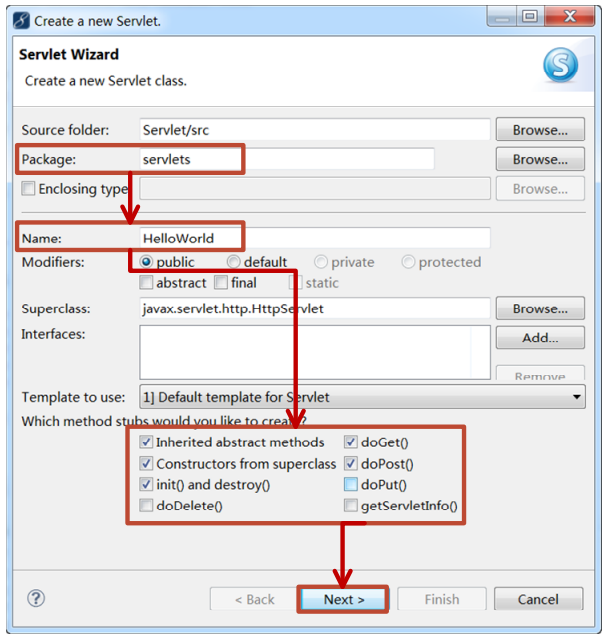



图 5.5　创建 Servlet

单击“Next”按钮进入下一步。并在显示的对话框中完成相关 Servlet 文件的配置，如图 5.6 所示。

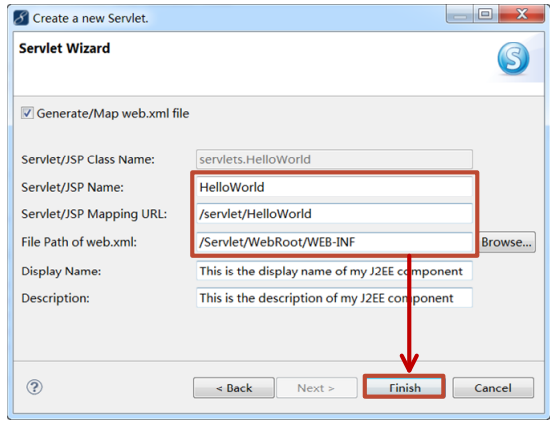



图 5.6　完成 Servlet 的建立

单击“Finish”按钮完成 Servlet 的建立。

**【示例 5.1】**Servlet 建立完成后，系统会自动弹出 HelloWorld.java 文件，我们将其中的代码修改为如图 5.7 所示的形式。

单击 MyEclipse 工具栏上的 run 按钮，在弹出的下拉菜单中选择“Run on Tomcat 7.x”选项。系统会完成对 HelloWorld.java 示例的编译。然后我们在浏览器地址栏中输入

http://localhost:8080/Servlet/servlet/HelloWorld 来测试这个程序，显示结果如图 5.8 所示。

```
package servlets;
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 HelloWorld extends HttpServlet
{
   public void doGet(HttpServletRequest request, HttpServletResponse response)
               throws ServletException, IOException
        {
                  response.setContentType("text/html");
                  PrintWriter out = response.getWriter();
                  out.println("<b>Hello World</b>");
        }
}
```

引入相应的架包

运用doGet()方法并捕获相应异常

图 5.7　HelloWorld.java 示例

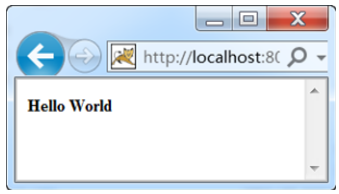



图 5.8　HelloWorld.java 示例运行结果

在建立 Servlet 的过程中，MyEclipse 还会自动在 web.xml 文件（这个文件可以在 WebRoot\WEB-INF 目录中找到）中添加 Servlet 的配置代码，如在本例中自动生成的 web.xml 文件的代码如图 5.9 所示。

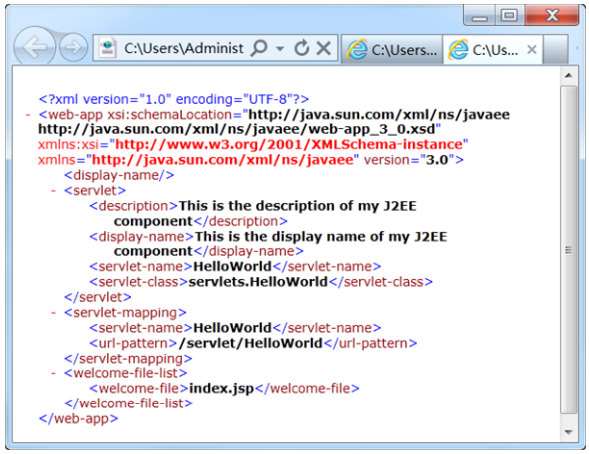



图 5.9　HelloWorld.java 的 web.xml 配置文件

在系统中创建的第一个 Servlet 程序系统会为我们自动生成 web.xml 配置文件，但是以后的 Servlet 程序就需要我们自己来配置了。即一般情况下都需要在当前应用项目的 web.xml 配置文件中对各个 Servlet 进行配置，其中 web.xml 文件的位置在当前项目应用的 WEB-INF 文件夹下。我们就结合图 5.9 的示例来讲解如何对 Servlet 进行配置，如图 5.10 所示。

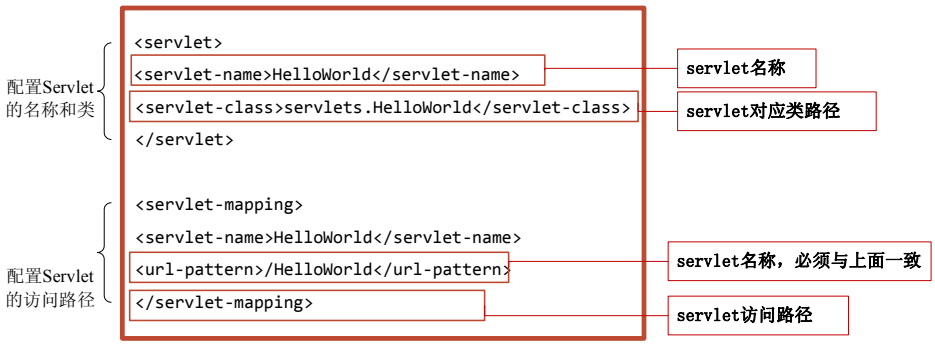



图 5.10　Servlet 的配置

总之，编写一个 Servlet 要经过以下三个步骤。

（1）编写 Servlet 的功能代码，即实现功能的代码类。

（2）把编译成功的 Servlet 功能代码类文件拷贝到当前应用项目的 WEB-INF/classes 目录下。

（3）在当前应用项目的 web.xml 文件中对 Servlet 进行配置，即在 web.xml 中添加配置信息。

经过这样三个步骤我们就可以通过浏览器访问这个 Servlet 了。

## 5.3　使用 HttpServlet 处理客户端请求

HttpServlet 是使用 HTTP 协议的 Web 服务器的 Servlet 类，这个类已经被系统定义好。该类的一些方法，如 doGet()方法、doPost()方法等，提供了处理客户端请求的接口。在实际编程中，程序员需要继承这个类，并重写上述方法，去编写自己的 Servlet。使用重写后的方法，就可以完成对客户端请求的处理。

### 5.3.1　处理 Get 请求 doGet

doGet()方法是 HttpServlet 类中用来处理 Get 请求的方法。用户通过继承 HttpServlet，重写 doGet()方法，实现对客户端的 Get 请求进行处理。要调用 doGet()方法，必须在客户端的表单里指定请求的类型为 Get。doGet()方法的语法格式如图 5.11 所示。


图 5.11　doGet()方法的语法格式

**【示例 5.2】**例如我们来看一个用户提交表单的实例 userform.jsp，通过这个例子来看 doGet()方法的用法，userform.jsp 的具体代码如图 5.12 所示。

```jsp
<%@ page language="java" contentType="text/html;charset=gb2312"%>
<html>
<head>
   <title>用户表单</title>
</head>
<body>
   <form action="doGetDemo" method="get">
      用户名:<input type="text" name="username"/><br>
      密码:<input type="password" name="password"/><br>
      <input type="submit" value="提交"/>
      <input type="reset" value="重置"/>
   </form>
</body>
</html>
```

指明跳转Servlet文件
指定请求的类型为Get

图 5.12　userform.jsp 示例

在上面这个表单中，指明表单的处理程序是 doGetDemo 这个 Servlet，doGetDemo.java 的具体代码如图 5.13 所示。

```java
package servlets;
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 doGetDemo extends HttpServlet {
    public void doGet(HttpServletRequest request,HttpServletResponse response)
                    throws ServletException, IOException {
            response.setContentType("text/html;charset=gb2312");
            PrintWriter out = response.getWriter();
            request.setCharacterEncoding("gb2312");
            String username = request.getParameter("username");
            String password = request.getParameter("password");
            out.println("<html>");
            out.println("<body>");
            out.println("用户名:" + username + "<br>");
            out.println("密码: " + password);
            out.println("</body>");
            out.println("</html>");
      }
}
```

doGet()方法

图 5.13　doGetDemo.java 示例

在前面 HelloWorld 的例子中已经介绍过，每个 Servlet 必须在 web.xml 中进行配置，然后

Servelt 引擎才能通过访问路径找到对应的 Servlet 类文件，上面这个 Servlet 的配置信息如图 5.14 所示。

```
<servlet>
    <servlet-name>doGetDemo</servlet-name>
    <servlet-class>servlets.doGetDemo</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>doGetDemo</servlet-name>
    <url-pattern>/doGetDemo</url-pattern>
</servlet-mapping>
```

图 5.14　doGetDemo.java 的 web.xml 配置

然后我们在浏览器地址栏中输入 http://localhost:8080/Servlet/userform.jsp 来测试这个程序，显示结果如图 5.15 所示。

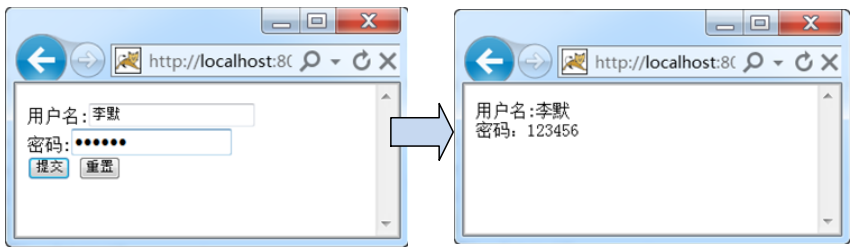



图 5.15　userform.jsp 示例运行结果

## 5.3.2　处理 Post 请求 doPost

doPost()是 HttpServlet 中用于处理 Post 请求的方法。如果要调用 doPost()方法，必须在表单中指定 Post 请求。doPost()方法与 doGet()方法的用法一般来说没什么区别，doGet()方法用于处理 http get 请求，doPost()方法用于处理 http post 请求。至于它们的不同，简单地说，get 是通过 http header 来传输数据，有字数限制，而 post 则是通过 http body 来传输数据，没有字数的限制。doPost()方法的语法格式如图 5.16 所示。


图 5.16　doPost()方法的语法格式

**【示例 5.3】**我们来看一个使用 doPost()方法的用户提交表单实例 userform2.jsp，其具体代码如图 5.17 所示。

这个表单中指明表单的处理程序是 doPostDemo 这个 Servlet，doPostDemo.java 的具体代码如图 5.18 所示。

```jsp
<%@ page language="java" contentType="text/html;charset=gb2312"%>
<html>
<head>
        <title>用户表单</title>
</head>
<body>
        <form action="doPostDemo" method="post">
                用户名:<input type="text" name="username"/><br>
                密码:<input type="password" name="password"/><br>
                <input type="submit" value="提交"/>
                <input type="reset" value="重置"/>
        </form>
</body>
</html>
```

指明跳转Servlet文件
指定请求的类型为post

图 5.17　userform2.jsp 示例

```java
package servlets;
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 doPostDemo extends HttpServlet {
    public void doPost(HttpServletRequest request,HttpServletResponse response)
                    throws ServletException, IOException {
        response.setContentType("text/html;charset=gb2312");
        PrintWriter out = response.getWriter();
        request.setCharacterEncoding("gb2312");
        String username = request.getParameter("username");
        String password = request.getParameter("password");
        out.println("<html>");
        out.println("<body>");
        out.println("用户名:" + username + "<br>");
        out.println("密码: " + password);
        out.println("</body>");
        out.println("</html>");
        }
}
```

doPost()方法

图 5.18　doPostDemo.java 示例

这个 Servlet 的配置信息如图 5.19 所示。

```
<servlet>
    <servlet-name>doPostDemo</servlet-name>
    <servlet-class>servlets.doPostDemo</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>doPostDemo</servlet-name>
    <url-pattern>/doPostDemo</url-pattern>
</servlet-mapping>
```

图 5.19　doPostDemo.java 示例的 web.xml 配置

在浏览器地址栏中输入 http://localhost:8080/Servlet/userform2.jsp 来测试这个程序，显示结果如图 5.20 所示。

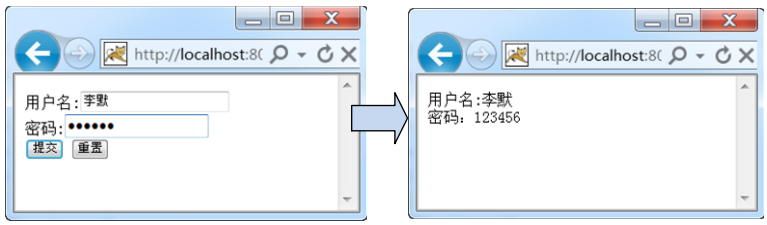



图 5.20　userform2.jsp 运行结果

下面我们来为大家介绍一下 javax.servlet.http 提供的 HTTP Servlet 应用编程接口。创建 Servlet，需要扩展 HttpServlet 类，HttpServlet 类包含 init()、destroy()、service()等方法，其中 init()和 destroy()方法是继承的，具体的方法及方法描述如表 5.1 所示。

表 5.1　HttpServlet类方法及方法描述

| 方 法 名 | 方 法 描 述 |
|---|---|
| init()方法 | 服务器装入 Servlet 时执行。可以配置服务器，在启动服务器或客户机首次访问 Servlet 时装入 Servlet |
| service()方法 | Servlet 的核心。每当一个客户请求一个 HttpServlet 对象，该对象的 service()方法就要被调用，而且传递给这个方法一个"请求"(ServletRequest)对象和一个"响应"(ServletResponse)对象作为参数 |
| destroy()方法 | 在服务器停止且卸装 Servlet 时执行该方法。可以将 Servlet 作为服务器进程的一部分来关闭 |
| GetServletConfig()方法 | GetServletConfig()方法返回一个 ServletConfig 对象，该对象用来返回初始化参数和 ServletContext。ServletContext 接口提供有关 servlet 的环境信息 |
| GetServletInfo()方法 | GetServletInfo()方法是一个可选的方法，它提供有关 servlet 的信息，如作者、版本、版权等 |

当服务器调用 Servlet 的 Service()、doGet()和 doPost()这三个方法时，均需要“请求”和“响应”对象作为参数。“请求”对象提供有关请求的信息，而“响应”对象提供了一个将响应信息返回给浏览器的一个路径。

## 5.4　JSP 页面调用 Servlet

在上面 HelloWorld 的示例程序中，我们直接在浏览器中输入具体的地址进行访问。在实际的应用中，不可能让用户在浏览器中直接输入 Servlet 的地址进行访问。一般情况下，可以通过调用 Servlet 进行访问，在这里介绍通过提交表单和超链接两种方式调用 Servlet。

### 5.4.1　通过表单提交调用 Servlet

在通过提交表单调用 Servlet 时，只需要把表单的 action 指向对应的 Servlet 即可。

**【示例 5.4】**下面是一个简单的表单 form.jsp，通过这个表单可以调用指定的 Servlet，具体代码如图 5.21 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
 <head>
   <title>Servlet接收表单示例</title>
 </head>
 <body>
    <font size="4">
       <form action="AcceptForm" method="post">
            姓名: <input type="text" name="name"/><br>
            省份: <input type="text" name="province"><br>
           <input type="submit" value="提交">
       </form>
    </font>
 </body>
</html>
```

把表单的action指向对应的Servlet

图 5.21　form.jsp 示例

在上面这个表单中，指明表单的处理程序是 AcceptForm 这个 Servlet，AcceptForm 的具体代码如图 5.22 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class AcceptForm extends HttpServlet {
   public void doGet(HttpServletRequest request,HttpServletResponse response)
                    throws IOException,ServletException {
         response.setContentType("text/html");
         response.setCharacterEncoding("gb2312");
         PrintWriter out = response.getWriter();
         String name = request.getParameter("name");
         String province = request.getParameter("province");
         out.println("<font size='4'>");
         out.print("提交的表单内容为:<br>");
         out.println("姓名:"+new String(name.getBytes
                 ("ISO-8859- 1"),"gb2312")+"<br>");
         out.println("省份:"+new String(province.getBytes
                 ("ISO-8859-1"),"gb2312")+"<br>");
         out.print("</font>");    }
    public void doPost(HttpServletRequest request,HttpServletResponse response)
                    throws IOException,ServletException {
         doGet(request, response);
       }
}
```

doGet()方法

对doGet()方法进行重写，在doPost()方法中直接调用doGet()方法的具体内容

doPost()方法

图 5.22　AcceptForm.java 示例

这个 Servlet 的配置信息如图 5.23 所示。

```
<servlet>
     <servlet-name>AcceptForm</servlet-name>
     <servlet-class>servlets.AcceptForm</servlet-class>
 </servlet>
 <servlet-mapping>
      <servlet-name>AcceptForm</servlet-name>
      <url-pattern>/AcceptForm</url-pattern>
  </servlet-mapping>
```

图 5.23　AcceptForm.java 的 web.xml 配置

然后我们在浏览器地址栏中输入 http://localhost:8080/Servlet/form.jsp 来测试这个程序，显示结果如图 5.24 所示。

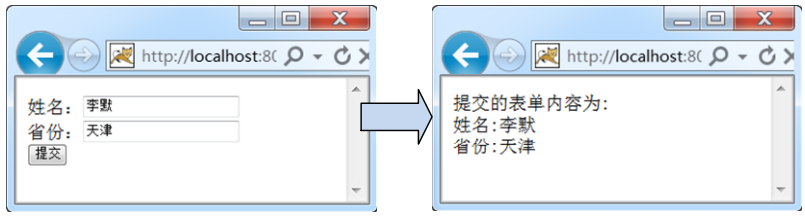



图 5.24　form.jsp 示例运行结果

## 5.4.2　通过超链接调用 Servlet

当用户有输入的内容需要提交给服务器时，我们可以用表单来调用 Servlet。如果在没有输入的数据内容需要提交的情况下，我们可以直接通过超链接的方式来调用 Servlet，并对其传递参数。

**【示例 5.5】**下面的例子 link.jsp 就是使用超链接的方式调用 Servlet，并且向这个 Servlet 传递一个参数，link.jsp 示例代码如图 5.25 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
 <head>
   <title>Servlet接收链接传递参数示例</title>
 </head>
 <body>
    <font size="4">
      单击下面的链接：<br>
      <a href="AcceptLink?name=limo">调用Servlet,并传递参数</a>
    </font>
 </body>
</html>
```

AcceptLink就是这个链接调用的Servlet

图 5.25　link.jsp 示例

我们通过链接向 AcceptLink 这个 Servlet 传递了一个名为 name 的参数，这个参数的值为中文“李默”，AcceptLink.java 的具体代码如图 5.26 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class AcceptLink extends HttpServlet {
        public void doGet(HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException {
            response.setContentType("text/html");
            response.setCharacterEncoding("gb2312");
            PrintWriter out = response.getWriter();
            String name = request.getParameter("name");
            out.println("<font size='4'>");
            out.println("链接传递过来的参数为：<br>");
            out.println("参数名称：姓名<br>");
            out.println("参数值：" + name + "<br>");
            out.print("</font>");
        }
        public void doPost(HttpServletRequest request, HttpServletResponse
response)throws IOException, ServletException {
            doGet(request, response);
        }
}
```

doGet()方法

对doGet()方法进行重写，在doPost()方法中直接调用doGet()方法的具体内容

doPost()方法

图 5.26　AcceptLink.java 示例

对 web.xml 进行配置，Servlet 的配置信息如图 5.27 所示。

```
<servlet>
    <servlet-name>AcceptLink</servlet-name>
    <servlet-class>servlets.AcceptLink</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>AcceptLink</servlet-name>
    <url-pattern>/AcceptLink</url-pattern>
</servlet-mapping>
```

图 5.27　AcceptLink.java 的 web.xml 配置

然后我们在浏览器地址栏中输入 http://localhost:8080/Servlet/link.jsp 来测试这个程序，显示结果如图 5.28 所示。

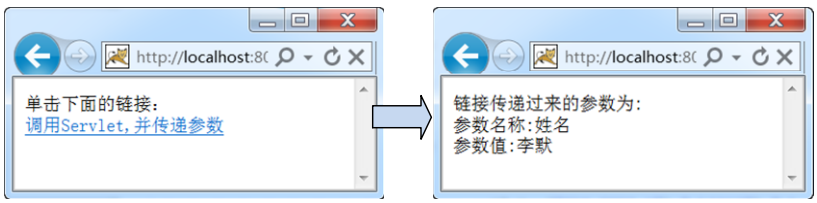



图 5.28　link.jsp 示例运行结果

## 5.5 Servlet 文件操作

在 JSP 的开发过程中，我们常常把相关内容存储为文件。在 Servlet 中我们可以使用输入输出流实现对文件的读写。同时，使用 Servlet 还可以很方便的实现文件的下载操作。这一节我们就来学习如何实现 Servlet 的文件操作。

### 5.5.1 Servlet 读取文件

**【示例 5.6】**我们举一个实例 FileRead.java 来读取一个文本文件 content.txt 的内容，并且在页面上打印文件的内容，这个 Servlet 的代码如图 5.29 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FileRead extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException
   {
       response.setContentType("text/html");
       response.setCharacterEncoding("gb2312");
       PrintWriter out = response.getWriter();
       String fileName =  "content.txt";
       String realPath = request.getRealPath(fileName);
       File file = new File(realPath);
   if(file.exists())
   {
       FileReader reader = new FileReader(file);
       BufferedReader bufferReader = new BufferedReader(reader);
       String line = null;
       while((line = bufferReader.readLine())!=null)
          {
             out.print("<font size='4'>"+line+"</font><br>");
          }
   }else
   {
            out.print("文件不存在！");
   }
}
   public void doPost(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException
   {
       doGet(request, response);
   }
}
```

设置服务器的响应内容类型，并从文件中读取内容

取得文件绝对存储路径

读取文件内容

图 5.29　FileRead.java 示例

这个 Servlet 的配置信息如图 5.30 所示。

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/FileRead 来测试这个程序，显示结果如图 5.31 所示。

```
<servlet>
    <servlet-name>FileRead</servlet-name>
    <servlet-class>servlets.FileRead</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FileRead</servlet-name>
    <url-pattern>/FileRead</url-pattern>
</servlet-mapping>
```

图 5.30　FileRead.java 的 web.xml 配置

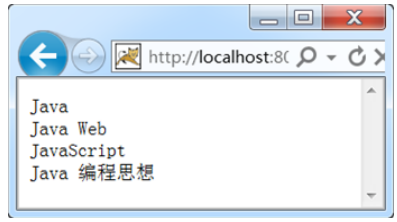



图 5.31　FileRead.java 示例运行结果

## 5.5.2　Servlet 写文件

Servlet 写文件的处理方法和读取文件的处理方法非常类似，即把文件输入流换成文件输出流。我们也可以来看一个写文件示例。

**【示例 5.7】**我们举一个实例 FileWrite.java 来向一个文件 new.txt 中写入内容。这个 Servlet 的代码如图 5.32 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class FileWrite extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException
   {
       response.setCharacterEncoding("gb2312");
       PrintWriter out = response.getWriter();
       String fileName =  "new.txt";
       String realPath = request.getRealPath(fileName);
       File file = new File(realPath);
       FileWriter writer = new FileWriter(file);
       BufferedWriter bufferWriter = new BufferedWriter(writer);
       bufferWriter.write("计算机网络");
       bufferWriter.newLine();
       bufferWriter.write("计算机组成原理");
       bufferWriter.flush();
       bufferWriter.close();
       writer.close();
   }
   public void doPost(HttpServletRequest request, HttpServletResponse
response)
   throws IOException, ServletException
   {
       doGet(request, response);
   }
}
```

设置服务器的响应内容类型，并创建一个文件

在文件中写入内容

图 5.32　FileWrite.java 示例

这个 Servlet 的配置信息如图 5.33 所示。

```
<servlet>
   <servlet-name>FileWrite</servlet-name>
   <servlet-class>servlets.FileWrite</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>FileWrite</servlet-name>
   <url-pattern>/FileWrite</url-pattern>
</servlet-mapping>
```

图 5.33　FileWrite.java 的 web.xml 配置

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/FileWrite 来测试这个程序，显示结果如图 5.34 所示。

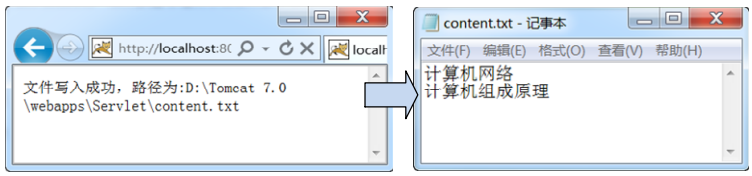



图 5.34　FileWrite.java 运行结果

## 5.5.3　Servlet 下载文件

利用 Servlet 可以很方便地实现文件的下载，我们只需要对服务器的响应对象 response 进行简单的设置即可。

**【示例 5.8】**下面的就是一个文件下载的示例程序 FileDownload.java，这个程序将从当前应用项目的根目录下载一个名为 test.xls 的 Excel 文档，具体代码如图 5.35 所示。

```
package servlets;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class FileDownload extends HttpServlet {
    public void doGet(HttpServletRequest request,HttpServletResponse response){
      try {
         String fname = "test.xls";
         response.setCharacterEncoding("UTF-8");
         fname = java.net.URLEncoder.encode(fname, "UTF-8");
         response.setHeader("Content-Disposition","attachment;filename="+fname);
         response.setContentType("application/msexcel");
         } catch (Exception e) {
         System.out.println(e);
         }
     }
}
```

字符编码的设置，用来支持中文的文件名

指明文件位置及文件类型

图 5.35　FileDownload.java 示例

同样我们需要对这个 Servlet 进行配置，配置信息如图 5.36 所示。

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/FileDownload，显示结果如图 5.37 所示。

```
<servlet>
    <servlet-name>FileDownload</servlet-name>
    <servlet-class>servlets.FileDownload</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FileDownload</servlet-name>
    <url-pattern>/FileDownload</url-pattern>
</servlet-mapping>
```

图 5.36　FileDownload.java 的 web.xml 配置


图 5.37　文件下载对话框

## 5.6　Servlet 的应用

Servlet 是与 HTTP 协议紧密结合的，使用 Servlet 几乎可以处理 HTTP 协议各个方面的内容，在本节的几个示例程序中，将集中展示 Servlet 在 HTTP 方面的具体应用。

### 5.6.1　获取请求信息头部内容

当用户访问一个页面时，会提交一个 HTTP 请求给服务器的 Servlet 引擎，在这个请求中包含了 HTTP 文件的详细属性信息。我们可以应用 request.getHeaderNames()方法来获取请求信息头部内容。

**【示例 5.9】**在下面这个 Servlet 示例 RequestHeader.java 中将取出 HTTP 头部内容，并在页面打印，其具体代码如图 5.38 所示。

```
package servlets;
import java.io.*;
import java.util.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class RequestHeader extends HttpServlet {
   public void doGet(HttpServletRequest request,HttpServletResponse response)
   throws IOException, ServletException
   {
       response.setContentType("text/html");
       PrintWriter out = response.getWriter();
       Enumeration e = request.getHeaderNames();
       while (e.hasMoreElements()) {
           String name = (String)e.nextElement();
           String value = request.getHeader(name);
           out.println(name + " = " + value+"<br>");
       }
   }
}
```

首先使用方法取出HTTP头信息的名称，然后循环取出对应的头信息的值

图 5.38　RequestHeader.java 示例

这个 Servlet 的配置信息如图 5.39 所示。

```
<servlet>
   <servlet-name>RequestHeader</servlet-name>
   <servlet-class>servlets.RequestHeader</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>RequestHeader</servlet-name>
   <url-pattern>/RequestHeader</url-pattern>
</servlet-mapping>
```

图 5.39　RequestHeader.java 的 web.xml 配置

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/RequestHeader，显示结果如图 5.40 所示。

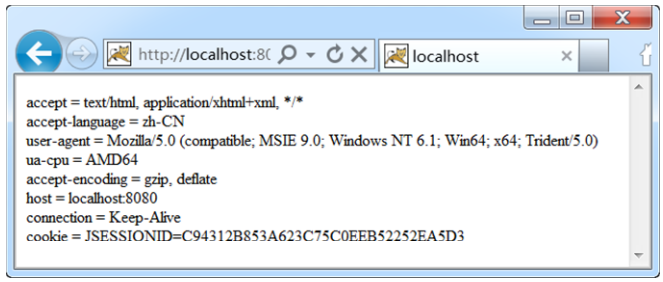



图 5.40　RequestHeader.java 的运行结果

如图 5.40 所示，等号左边的是 HTTP 头信息项的名称，等号右边是这项的值。而且具体 HTTP 头文件信息的内容不尽相同，在不同的浏览器中会有所不同。

## 5.6.2　获取请求信息

在上面的 Servlet 示例中，我们取出了 HTTP 文件头信息，在 Servlet 中还可以很方便取出用户发出请求对象自身的信息。这些信息是和用户的请求密切相关的，例如用户提交请求所使用的协议，客户提交表单的方法是 POST 还是 GET 等。

表 5.2　获取请求信息的方法

| 方 法 名 | 方 法 描 述 |
|---|---|
| request.getMethod() | 取出表单提交的方法，可以是 POST 或者是 GET |
| request.getRequestURI() | 取出这个客户请求的 URI，即相对的访问路径 |
| request.getProtocol() | 取出客户发出请求时使用的协议 |
| request.getRemoteAddr() | 取出客户的 IP 地址 |

**【示例 5.10】**在下面这个示例程序 RequestInfo.java 中我们将看到这些获取请求信息方法的用法，其具体代码如图 5.41 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class RequestInfo extends HttpServlet {
   public void doGet(HttpServletRequest request,HttpServletResponse response)
   throws IOException, ServletException
   {
       response.setContentType("text/html");
       response.setCharacterEncoding("gb2312");
       PrintWriter out = response.getWriter();
       out.println("<html>");out.println("<body>");out.println("<head>");
       out.println("<title>请求信息示例</title>");
       out.println("</head>");
       out.println("<body><font size='2'");
       out.println("<b>请求信息示例</b><br>");
       out.println("Method: " + request.getMethod()+"<br>");
       out.println("Request URI: " + request.getRequestURI()+"<br>");
       out.println("Protocol: " + request.getProtocol()+"<br>");
       out.println("Remote Address: " + request.getRemoteAddr()+"<br>");
       out.println("</font></body>");out.println("</html>");
   }
   public void doPost(HttpServletRequest request,HttpServletResponse response)
   throws IOException, ServletException
   {
       doGet(request, response);
   }
}
```

获取请求信息方法的用法

图 5.41　RequestInfo.java 示例

这个 Servlet 的配置信息如图 5.42 所示。

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/RequestInfo，显示结果如图 5.43 所示。

```
<servlet>
   <servlet-name>RequestInfo</servlet-name>
   <servlet-class>servlets.RequestInfo</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>RequestInfo</servlet-name>
   <url-pattern>/RequestInfo</url-pattern>
</servlet-mapping>
```

图 5.42　RequestInfo.java 示例的 web.xml 配置

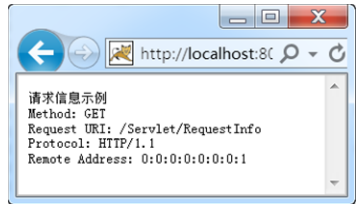



图 5.43　RequestInfo.java 示例的运行结果

## 5.6.3　获取参数信息

有关用户请求的参数信息，也可以通过 Servlet 来获取。这种参数既包括以 POST 方法或者是 GET 方法提交的表单，也包括直接使用超链接传递的参数。Servlet 都可以使用 request.getParameter()方法取得这些参数信息并且加以处理。

**【示例 5.11】**在下面的例子 RequestParam.java 中将具体展示 Servlet 获取各种参数的方法。首先我们要建立一个前面多次使用到的简单表单 paramForm.jsp，在这个例子中我们会分别使

用 POST 方法和 GET 方法提交用户的请求信息，paramForm.jsp 示例具体代码如图 5.44 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
 <head>
   <title>Servlet接收表单示例</title>
 </head>
 <body>
    <font size="4">
       <form action="RequestParam" method="post">
            姓名：<input type="text" name="name"/><br>
            省份：<input type="text" name="province"><br>
           <input type="submit" value="提交">
       </form>
    </font>
 </body>
</html>
```

指明跳转Servlet文件
指定请求的类型为post

图 5.44　paramForm.jsp 示例

接收这个表单请求的 Servlet 即 RequestParam.java 的具体代码如图 5.45 所示。

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

public class RequestParam extends HttpServlet {
   public void doGet(HttpServletRequest request,HttpServletResponse response)
   throws IOException, ServletException
   {
      response.setContentType("text/html");
      response.setCharacterEncoding("gb2312");
      Enumeration e = request.getParameterNames();
      PrintWriter out = response.getWriter ();
      out.print("<font size='4'>");
      out.print("下面是用GET方法传递过来的参数:<br>");
      while (e.hasMoreElements()) {
        String name = (String)e.nextElement();
        String value = request.getParameter(name);
        out.println(name + " = " + value+"<br>");
        }
        out.print("</font>");
   }

   public void doPost(HttpServletRequest request,HttpServletResponse response)
   throws IOException, ServletException
   {
       response.setContentType("text/html");
       response.setCharacterEncoding("gb2312");
     Enumeration e = request.getParameterNames();
     PrintWriter out = response.getWriter ();
     out.print("<font size='4'>");
     out.print("下面是用POST方法传递过来的参数:<br>");
       while (e.hasMoreElements()) {
         String name = (String)e.nextElement();
         String value = request.getParameter(name);
         out.println(new String(name.getBytes("ISO-8859-1"),"gb2312")+" = "
           + new String(value.getBytes("ISO-8859-1"),"gb2312")+"<br>");
       }
       out.print("</font>");
   }
}
```

doGet()方法

doPost()方法

图 5.45　RequestParam.java 示例

在这个 Servlet 中可以看到，我们分别处理 doGet()和 doPost()方法，因为要处理不同方法提交的参数，需要有不同的处理方法，所以在这里分别实现 doGet()和 doPost()方法。

这个 Servlet 的配置信息如图 5.46 所示。

```
<servlet>
    <servlet-name>RequestParam</servlet-name>
    <servlet-class>servlets.RequestParam</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>RequestParam</servlet-name>
    <url-pattern>/RequestParam</url-pattern>
  </servlet-mapping>
```

图 5.46　RequestParam.java 示例的 web.xml 配置

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/RequestParam，显示结果如图 5.47 所示。


图 5.47　POST 方法传递结果

我们把表单 paramForm.jsp 的提交方法改为 GET 时，这个 Servlet 的运行效果如图 5.48 所示。

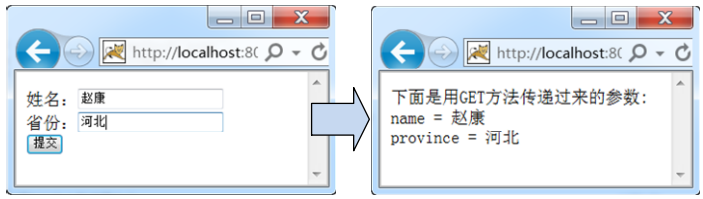



图 5.48　GET 方法传递结果

## 5.6.4　Cookie 操作

Cookie 是一种在客户端保存信息的技术。读者在浏览网页时可能会注意到这样的现象，如在打开某个登录网页时，在第一次打开时，用户名文本框是空的，当输入一个用户名，并成功登录后。在第二次打开这个登录网页时，在第一次输入的用户名会被自动填入这个用户名文本框，就算重启计算机后，仍然如此，其实这就是 Cookie 所起的作用。

在 Servlet 中，使用 java.servlet.http.Cookie 类来封装一个 Cookie 消息，在 HttpServletResponse 接口中定义了一个 addCookie 方法来向浏览器发送 Cookie 消息（也就是 Cookie 对象），在 HttpServletRequest 接口中定义了一个 getCookies 方法来读取浏览器发送的 Web 服务器的所有 Cookie 消息。Cookie 类中定义了生成和提取 Cookie 消息的各个属性的方法。Cookie 类只有一个构造方法，它的语法结构如图 5.49 所示。

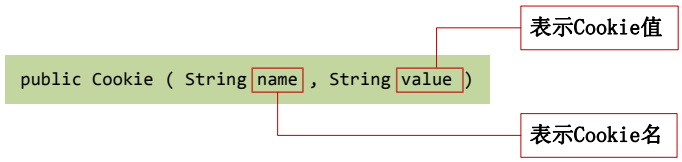



图 5.49　Cookie 方法的语法结构

Cookie 类中的其他常用方法如表 5.3 所示。

表 5.3　Cookie类中方法及方法描述

| 方 法 名 | 方 法 描 述 |
|---|---|
| getName 方法 | 用于获得 Cookie 的名称 |
| setValue 和 getValue 方法 | 分别用于设置和获得 Cookie 的值 |
| setMaxAge 和 getMaxAge 方法 | 分别用于设置和获得 Cookie 在客户机的有效时间，也就是在客户机上的有效秒数 |
| setPath 和 getPath 方法 | 分别用于设置和获得当前 Cookie 的有效 Web 路径 |
| setDomain 和 getDomain 方法 | 分别用于设置和获得当前 Cookie 的有效域 |
| setComment 和 getComment 方法 | 分别用于设置和返回当前 Cookie 的注释部分 |
| setVersion 与 getVersion 方法 | 分别用于设置和获得当前 Cookie 的协议版本 |
| setSecure 和 getSecure 方法 | 分别用于设置和获得当前 Cookie 是否只能使用安全的协议传输 Cookie |

**【示例 5.12】**现在我们给出一个完整的实例 SaveCookie.java 来演示如何在 Servlet 中使用 Cookie。其中 SaveCookie 类负责向客户端浏览器写入三种 Cookie：永久 Cookie、临时 Cookie 和有效时间为 0 的 Cookie，其代码如图 5.50 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class SaveCookie extends HttpServlet
{
        protected void service(HttpServletRequest request,
        HttpServletResponse response)throws ServletException,IOException
        {
            response.setContentType("text/html; charset=UTF-8");
            PrintWriter out = response.getWriter();
            Cookie tempCookie = new Cookie("temp", "87654321");
            tempCookie.setMaxAge(-1);
            response.addCookie(tempCookie);
            Cookie cookie = new Cookie("cookie", "6666");
            cookie.setMaxAge(0);
            response.addCookie(cookie);
            String user = request.getParameter("user");
                  if (user != null)
                  {
                       Cookie userCookie = new Cookie("user", user);
                       userCookie.setMaxAge(60 * 60 * 24);
                       userCookie.setPath("/");
                       response.addCookie(userCookie);
                  }
            RequestDispatcher readCookie=
               getServletContext().getRequestDispatcher("/servlet/ReadCookie");
            readCookie.include(request, response);
        }
}
```



图 5.50　SaveCookie.java 示例

我们在 SaveCookie 中使用了一个 ReadCookie 类，这个类负责读取被保存的 Cookie 值，ReadCookie.java 的代码如图 5.51 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class ReadCookie extends HttpServlet
{
   protected Cookie getCookieValue(Cookie[] cookies, String name)
     { if (cookies != null)
         { for (Cookie c : cookies)
             {    if (c.getName().equals(name))
                  return c;
              }
         }
        return null;
     }
    public void service(HttpServletRequest request,HttpServletResponse response)
                          throws ServletException, IOException
        {
            response.setContentType("text/html; charset=UTF-8");
            PrintWriter out = response.getWriter();
            Cookie tempCookie = getCookieValue(request.getCookies(), "temp");
            if (tempCookie != null)
               out.println("临时Cookie值: " + tempCookie.getValue() + "<br/>");
            else
               out.println("临时Cookie未设置! <br/>");
            Cookie cookie = getCookieValue(request.getCookies(), "cookie");
            if (cookie != null)
               out.println("cookie: " + cookie.getValue() + "<br/>");
            else
               out.println("cookie已经被删除!<br/>");
            Cookie userCookie = getCookieValue(request.getCookies(), "user");
            if (userCookie != null)
               out.println("user: " + userCookie.getValue());
            else
               out.println("user未设置! ");
        }
}
```

通过一个Cookie名获得Cookie对象

获得临时Cookie

获取超时时间为0的Cookie

获得永久Cookie

图 5.51　ReadCookie.java 示例

这个 Servlet 的配置信息如图 5.52 所示。

```
<servlet>
    <servlet-name>SaveCookie</servlet-name>
    <servlet-class>servlets.SaveCookie</servlet-class>
</servlet>
<servlet>
    <servlet-name>ReadCookie</servlet-name>
    <servlet-class>servlets.ReadCookie</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>SaveCookie</servlet-name>
    <url-pattern>/SaveCookie</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>ReadCookie</servlet-name>
    <url-pattern>/ReadCookie</url-pattern>
</servlet-mapping>
```

图 5.52　SaveCookie.java 和 ReadCookie.java 的 web.xml 配置

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/ReadCookie，显示结果如图 5.53 所示。


图 5.53　读取 Cookie 值

**【示例 5.13】**我们再举一个例子，来展示 Servlet 操作 Cookie 的具体方法。首先我们向浏览器中发送一个 Cookie 消息，这个 Servlet（Cookies.java）的具体代码如图 5.54 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Cookies extends HttpServlet {
   public void doGet(HttpServletRequest request,HttpServletResponse response)
   throws IOException, ServletException
   {
       response.setContentType("text/html");
       response.setCharacterEncoding("gb2312");
       PrintWriter out = response.getWriter();
       Cookie cookie=
       new Cookie("name",java.net.URLEncoder.encode("李默","gb2312"));
       response.addCookie(cookie);
   }
}
```

设置并发送一个cookie

图 5.54　Cookies.java 示例

然后我们在创建一个 Cookies1.java 来读取我们所写入的 Cookie 消息，其具体代码如图 5.55 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class Cookies1 extends HttpServlet {
   public void doGet(HttpServletRequest request,HttpServletResponse response)
   throws IOException, ServletException
   {
       response.setContentType("text/html");
       response.setCharacterEncoding("gb2312");
       Cookie[] cookies = request.getCookies();
       PrintWriter out = response.getWriter();
       out.print("<font size='4'>");
       out.print("下面是Cookie的内容:<br>");
       for (int i = 0; i < cookies.length; i++) {
          Cookie c = cookies[i];
          String name = c.getName();
          String value = c.getValue();
          out.println(name+" = "
           + java.net.URLDecoder.decode(value)+"<br>");
       }
       out.print("</font>");
   }
}
```

取出用户机器中的Cookie

图 5.55　Cookies1.java 示例

这个 Servlet 的配置信息如图 5.56 所示。

我们在浏览器地址栏中依次输入 http://localhost:8080/Servlet/Cookies 和 http://localhost:8080/Servlet/Cookies1，显示结果如图 5.57 所示。

```
<servlet>
    <servlet-name>Cookies</servlet-name>
    <servlet-class>servlets.Cookies</servlet-class>
</servlet>
<servlet>
<servlet-name>Cookies1</servlet-name>
<servlet-class>servlets.Cookies1</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Cookies</servlet-name>
    <url-pattern>/Cookies</url-pattern>
</servlet-mapping>
<servlet-mapping>
    <servlet-name>Cookies1</servlet-name>
    <url-pattern>/Cookies1</url-pattern>
</servlet-mapping>
```

图 5.56　Cookies.java 和 Cookies1.java 的 web.xml 配置


图 5.57　Cookie 操作示例程序运行效果

## 5.7　Session 技术

Session 对象用来保存每个用户的用户信息和会话状态。Session 对象由服务器端自动创建，可以跟踪每个用户的操作状态。用户首次登录系统时，服务器会自动给用户分配唯一标识的 Session ID，可以用来区分开其他用户。相对于 Cookie，Session 是存储在服务器端的会话，相对安全，而且其存储长度限制也比 Cookie 的存储长度限制扩大了。

### 5.7.1　HttpSession 接口方法

在 Servlet 中使用 HttpSession 对象来描述 Session。一个 HttpSession 对象就是一个 Session。使用 HttpServletRequest 接口的 getSession 方法来获得一个 HttpSession 对象。

HttpSession 接口中的主要方法如表 5.4 所示。

表 5.4　HttpSession接口主要方法

| 方 法 名 | 方 法 描 述 |
|---|---|
| getId 方法 | 用于返回当前 HttpSession 对象的 Session ID |
| getCreationTime 方法 | 用于返回当前的 HttpSession 对象的创建时间 |
| getLastAccessedTime 方法 | 用于返回当前 HttpSession 对象的上一次被访问的时间 |
| setMaxInactiveInterval 和 getMaxInactiveInterval 方法 | 分别用来设置和返回当前 HttpSession 对象的可空闲的最长时间（单位：秒），这个时间也就是当前会话的有效间隔 |
| isNew 方法 | 用于判断当前的 HttpSession 对象是否是新创建的，如果是则返回 true，否则返回 false |
| isvalidate 方法 | 用于强制当前的 HttpSession 对象失效，这样 Web 服务器可以立即释放该 HttpSession 对象 |
| getServletContext 方法 | 用于返回当前 HttpSession 对象所属的 Web 应用程序的 ServletContext 对象 |

续表

| 方 法 名 | 方 法 描 述 |
| --- | --- |
| setAttribute 方法 | 用于将一个 String 类型的 ID 和一个对象相关联，并将其保存在当前的 HttpSession 对象中 |
| getAttribute 方法 | 用于返回一个和 String 类型的 ID 相关联的对象 |
| remoteAttribute 方法 | 用于删除与一个 String 类型的 ID 相关联的对象 |

getSession 是 HttpServletRequest 接口的方法，这个方法用于返回与当前请求相关的 HttpSession 对象，该方法有两种重载形式，它们的定义语法如图 5.58 所示。

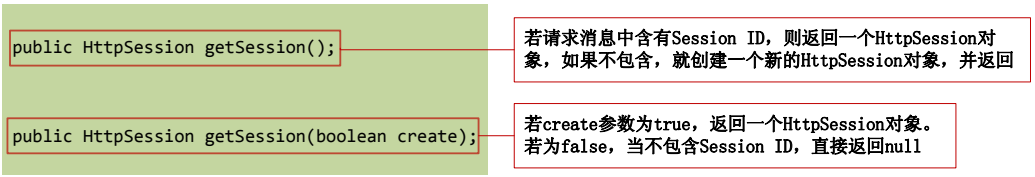



图 5.58　getSession 方法的定义语法

## 5.7.2　通过 Cookie 跟踪 Session

客户端必须通过一个 Session ID 才能找到以前在服务端创建的某一个 HttpSession 对象。通过 Session ID 找 HttpSession 对象的过程也叫做 Session 跟踪。一般客户端的 Session ID 通过 HTTP 请求消息头的 Cookie 字段发送给服务端，然后服务端通过 getSession 方法读取 Cookie 字段的值，以确定是否需要新建一个 HttpSession 对象，还是获得一个已经存在的 HttpSession 对象，或是什么都不做，直接返回 null。

**【示例 5.14】**下面的 SessionServlet 类演示了使用 Cookie 跟踪 Session 的过程，其具体代码如图 5.59 所示。

```
import javax.servlet.http.*;
public class SessionServlet extends HttpServlet
{
    protected void service(HttpServletRequest request,
          HttpServletResponse response) throws ServletException, IOException
        {
            response.setContentType("text/html; charset=UTF-8");
            PrintWriter out = response.getWriter();
            HttpSession session = request.getSession();
            session.setMaxInactiveInterval(60 * 60 * 24);
            if (session.isNew())
               {
                  session.setAttribute("session", "Servlet");
                  out.println("新会话已经建立! ");
               }
            else
               out.println("会话属性值: " + session.getAttribute("session"));
        }
}
```



图 5.59　SessionServlet.java 示例

当 HttpSession 对象是第一次创建时，向这个对象中写一个字符串值。如果 HttpSession 对象不是第一次创建，那么就将保存在 HttpSession 对象中的字符串值输出到客户端。

我们对其 web.xml 文件进行配置，如图 5.60 所示。

```
<servlet>
  <servlet-name>SessionServlet</servlet-name>
  <servlet-class>servlets.SessionServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>SessionServlet</servlet-name>
  <url-pattern>/SessionServlet</url-pattern>
</servlet-mapping>
```

图 5.60　SessionServlet.java 的 web.xml 配置

我们在浏览器地址栏中依次输入 http://localhost:8080/Servlet/SessionServlet，在显示结果后再刷新一次，页面将发生变化，如图 5.61 所示。


图 5.61　SessionServlet.java 示例运行结果

## 5.7.3　通过重写 URL 跟踪 Session

如果客户端浏览器不支持 Cookie 或是将 Cookie 功能关闭，那么就无法使用 Cookie 来传递 Session ID。为了在这种情况下仍然可以使用 Session，Servlet 规范提供了一种补充会话管理机制。这种管理机制允许在 Cookie 无法工作的情况下使用 URL 参数来传递 Session ID。

要想通过 URL 来发送 Session ID，必须要重写 URL。HttpServletResponse 提供了两个方法用于重写 URL，如图 5.62 所示。

| | |
|---|---|
| encodeURL方法 | 用于对所有内嵌在Servlet中的URL进行重写 |
| encodeRedirectURL方法 | 用于对sendRedirect方法所使用的URL进行重写 |

图 5.62　重写 URL 方法

**【示例 5.15】**下面我们举一个完整的通过重写 URL 来跟踪 Session 的例子 NewSessionServlet.java，它包含了 SessionServlet 类，并调用了 encodeURL 来重写 URL。

首先我们需要关闭 Cookie 功能，在 IE 中选择“工具”→“Internet”选项命令，打开 Internet 选项对话框，单击隐私标签，单击“高级”按钮，打开高级隐私设置对话框。勾选“替代自动 cookie 处理”复选框，选取两个“阻止”单选项，如图 5.63 所示。


图 5.63　关闭 Cookie 功能

然后我们建立一个 NewSessionServlet 类，其具体代码如图 5.64 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class NewSessionServlet extends HttpServlet
{
   protected void service(HttpServletRequest request,
       HttpServletResponse response) throws ServletException,IOException
   {
       response.setContentType("text/html; charset=UTF-8");
       PrintWriter out = response.getWriter();
       RequestDispatcher sessionServlet = getServletContext()
       .getRequestDispatcher("/SessionServlet");
       sessionServlet.include(request, response);
       out.println("<br><a href='" +
       response.encodeURL("SessionServlet") + "'>SessionServlet</a>");
   }
}
```

获得指向SessionServlet的RequestDispatcher对象

向客户端输出被重写的URL

图 5.64　NewSessionServlet.java 示例

我们对其 web.xml 文件进行配置，如图 5.65 所示。

```
  <servlet>
    <servlet-name>SessionServlet</servlet-name>
    <servlet-class>servlets.SessionServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>NewSessionServlet</servlet-name>
    <servlet-class>servlets.NewSessionServlet</servlet-class>
  </servlet>
 <servlet-mapping>
    <servlet-name>SessionServlet</servlet-name>
    <url-pattern>/SessionServlet</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>NewSessionServlet</servlet-name>
    <url-pattern>/NewSessionServlet</url-pattern>
  </servlet-mapping>
```

图 5.65　NewSessionServlet.java 的 web.xml 配置

在 IE 的地址栏中输入如下的 URL：http://localhost:8080/Servlet/NewSessionServlet，在运行页面上再单击访问链接，就会得到如图 5.66 所示的输出结果。

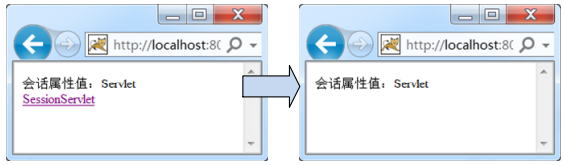



图 5.66　NewSessionServlet.java 的运行结果

在任何一个浏览器（如 FireFox）中输入上面的 URL，都会显示同样的内容。

## 5.8　Servlet 过滤器

过滤器是小型的 Web 组件，它负责拦截请求和响应，以便查看、提取或以某种方式操作正在客户机和服务器之间交换的数据。Servlet 过滤器应用非常广泛，有拦截的地方一般都可以用到过滤器。当前 Web 应用中过滤器已经是不可或缺的一部分了。

### 5.8.1　过滤器的方法和配置

与过滤器相关的 Servlet 共包含 3 个简单的接口，分别是 Filter、FilterChain 及 FilterConfig。要实现过滤器功能，必须先实现 Filter 接口。Filter 接口定义了 3 个方法，如图 5.67 所示。

| | |
|---|---|
| init()方法 | 在容器实例化过滤器时使用，使过滤器为后面的处理操作做好准备 |
| doFilter()方法 | 用于处理请求和响应，当请求与过滤器相关联Web资源时，进行调用 |
| destroy()方法 | Servlet在销毁过滤器实例时调用该方法 |

图 5.67　Filter 接口的 3 个方法

Servlets 过滤器是一个 Web 应用组件，和 Servlet 类似，也需要在 Web 应用配置文件（即 web.xml）中进行配置部署，如图 5.68 所示。

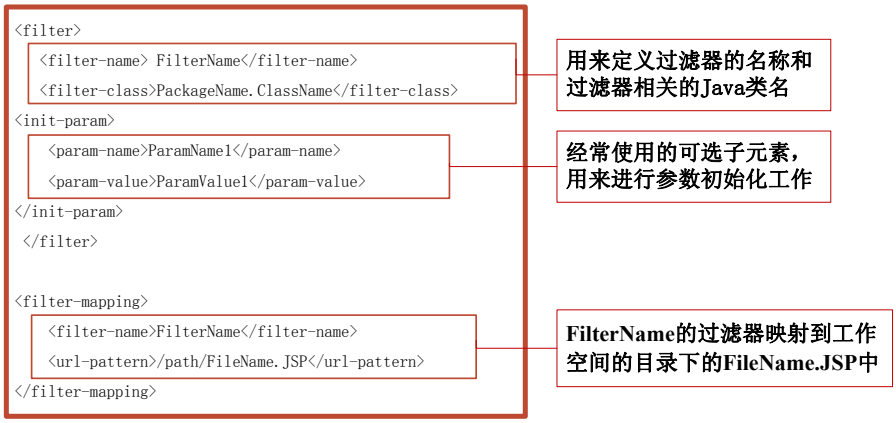



图 5.68　过滤器的配置

对于过滤器的映射配置，可以将过滤器映射到一个或多个 Servlet 和 JSP 文件中。以 Servlet 为例，我们来看其映射配置，如图 5.69 所示。

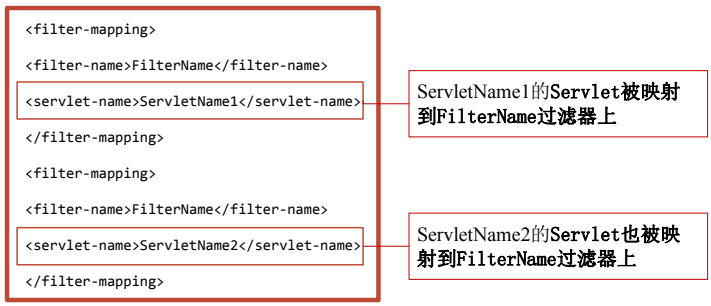



图 5.69　过滤器的映射配置

**注意**：在 web.xml 中配置 Servlet 和 Servlet 过滤器，应该先声明过滤器元素，再声明 Servlet 元素。

## 5.8.2　过滤器应用实例——禁止未授权的 IP 访问站点

在实际的应用中，可能会遇到这样的情况，需要对某些 IP 进行访问限制，不让非法的 IP 访问应用系统，这时就需要用到过滤器进行限制，当一个用户发出访问请求时，首先通过过滤器进行判断，如果用户的 IP 地址被限制，就禁止访问，只有合法的 IP 才可以继续访问。

**【示例 5.16】**下面我们就来举一个实例 FilterIP.java 来看如何用过滤器实现禁止未授权的 IP 访问站点，其具体代码如图 5.70 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
public class FilterIP implements Filter{
  protected FilterConfig filterconfig;
  protected String FilteredIP;
  public void init(FilterConfig conf) throws ServletException{
      this.filterconfig =conf;
      FilteredIP=conf.getInitParameter("FilteredIP");
      if(FilteredIP==null){
      FilteredIP="";
      }
  }
  public void doFilter(
      ServletRequest request,ServletResponse response,FilterChain chain)
           throws IOException, ServletException {
      response.setContentType("text/html");
      response.setCharacterEncoding("gb2312");
      RequestDispatcher dispatcher =
      request.getRequestDispatcher("ErrorInfo.jsp");
      String remoteIP = request.getRemoteAddr();
      if(remoteIP.equals(FilteredIP)){
          dispatcher.forward(request,response);
          return;
      }
      else{
           chain.doFilter(request,response);
      }
  }
  public void destroy(){
      this.filterconfig = null;
  }
}
```

过滤器初始化，获取被过滤IP

定义错误转向页面

读出本地IP，将其与要过滤掉的IP比较，如果相同，就转移到错误处理页面

图 5.70　FilterIP.java 示例

然后我们分别新建一个成功访问页面的 JSP 文件 Succeed.jsp 和访问发生错误的响应页面 ErrorInfo.jsp，其具体代码如图 5.71 和图 5.72 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
<head>
 <title>欢迎登录</title>
</head>
<body>
  <center><font size=4 >欢迎登录JavaWeb服务器</font></center>
 </body>
</html>
```

成功访问输出语句

图 5.71　Succeed.jsp 文件示例

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
<head>
   <title>错误报告</title>
</head>
<body>
<%      out.println("<center><font size=4 >
        对不起,您的IP不能登录本站点!</font></center>");   %>
 </body>
</html>
```

被拒绝服务IP登录显示信息

图 5.72　ErrorInfo.jsp 文件示例

本实例过滤器的 web.xml 配置如图 5.73 所示。

```
<filter>
    <filter-name>FilterIP</filter-name>
    <filter-class>servlets.FilterIP</filter-class>
    <init-param>
      <param-name>FilteredIP</param-name>
      <param-value>0:0:0:0:0:0:0:1</param-value>
    </init-param>
</filter>
<filter-mapping>
<filter-name>FilterIP</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

图 5.73　FilterIP.java 的 web.xml 配置

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/Succeed.jsp 来测试这个程序，由于本机地址是被拒绝服务的 IP，所以运行结果如图 5.74 所示。

如果我们修改 web.xml 配置的站点属性，在浏览器地址栏中再次输入 http://localhost:8080/Servlet/Succeed.jsp，就会出现正常登录界面，如图 5.75 所示。


图 5.74　被过滤 IP 的计算机所返回的页面

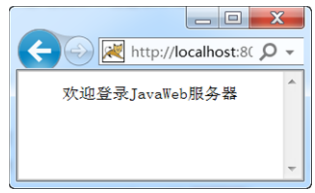



图 5.75　正常的登录返回界面

### 5.8.3　过滤器应用实例——版权过滤器

现在的网页都会在尾部加上版权标志，对于这一操作，我们可以运用过滤器很方便地实现它。

**【示例 5.17】**下面我们就来举一个实例 CopyrightFilter.java 来看如何用过滤器实现在网页尾部加上版权标志的功能，其具体代码如图 5.76 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
public class CopyrightFilter implements Filter{
      protected FilterConfig config;
      protected String date;
      public void init(FilterConfig filterconfig) throws ServletException{
       this.config=filterconfig;
       date=filterconfig.getInitParameter("date");
    }
   public void doFilter(ServletRequest request,ServletResponse response,
          FilterChain chain) throws IOException,ServletException{
       chain.doFilter(request,response);
       PrintWriter out=response.getWriter();
       out.println("<br><center><font size=4>
       版权所有：北京XXXX公司</font></center>" );
       if(date!=null){
       out.println("<br><center><font size=4>"+date+"</font></center>");
       }
       out.flush();
    }
   public void destroy(){
       this.config =null;
    }
}
```

过滤器初始化，读取日期参数

打印版权信息及日期

图 5.76　CopyrightFilter.java 示例

本实例过滤器的 web.xml 配置如图 5.77 所示。

```
<filter>
  <filter-name>CopyrightFilter</filter-name>
  <filter-class>servlets.CopyrightFilter</filter-class>
  <init-param>
       <param-name>date</param-name>
       <param-value>2012.9</param-value>
  </init-param>
</filter>
<filter-mapping>
   <filter-name>CopyrightFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>
```

图 5.77　CopyrightFilter.java 的 web.xml 配置

最后我们创建一个 HelloFilter.jsp 文件来测试过滤器是否已经产生作用了，其具体代码如图 5.78 所示。

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/HelloFilter.jsp 来测试这个程序，运行结果如图 5.79 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
 <head>
   <title>欢迎使用过滤器!</title>
 </head>
 <body>
   <%    for(int i=0;i<5;i++)
       out.println("Hello,Filter!"+   "<br>");  %>
 </body>
</html>
```

循环打印输出结果

图 5.78　HelloFilter.jsp 文件示例

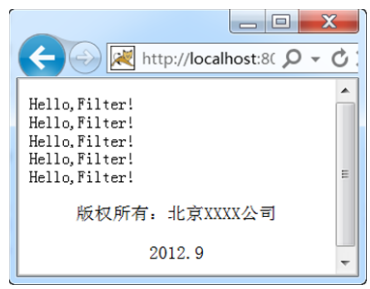



图 5.79　HelloFilter.jsp 运行结果

## 5.9　Servlet 监听器

Servlet 监听器是当今 Web 应用开发的一个重要组成部分。Servlet 监听器主要用来对 Web 应用进行监听和控制，极大地增强了 Web 应用的事件处理能力。一般来说，Servlet 监听就是指一些特殊的 Servlet 类，这些类可以监听 Web 应用的上下文信息、Servlet 会话信息、Servlet 请求信息。在实际操作中，程序员需要继承或实现一些已定义好的类或接口，从而编写出自己用于监听的类。这些类对特定的信息进行监听。一旦被监听的事件发生，这些类会自动调用相应的方法去执行指定的操作。

### 5.9.1　监听 Servlet 上下文信息

Servlet 上下文信息主要是指关于 ServletContext 接口的一些信息，比如 ServletContext 的创建和删除，Servlet 属性的增加、删除和修改等。这样就可以实现对 Servlet 上下文信息的跟踪和记录。为了实现这样的功能，程序员需要实现 ServletContextListener 和 ServletContextAttribute Listener 接口，从而编写出自己的 Servlet 类。ServletContext 接口的主要方法如表 5.5 所示。

表 5.5　ServletContext接口的主要方法

| 方 法 名 称 | 方 法 描 述 |
|---|---|
| getAttribute(String name) | 返回 Servlet 环境对象中指定的属性对象。如果该属性对象不存在，返回空值 |
| getAttributeNames() | 返回一个 Servlet 环境对象中可用的属性名的列表 |
| getContext(String uripath) | 返回一个 Servlet 环境对象，这个对象包括了特定 URI 路径的 Servlets 和资源，如果该路径不存在，则返回一个空值 |
| getRealPath(String path) | 返回与一个符合该格式的虚拟路径相对应的真实路径的 String |
| getResource(String uripath) | 返回一个 URL 对象，该对象反映位于给定的 URL 地址的 Servlet 环境对象已知的资源 |
| getServerInfo() | 返回一个 String 对象，该对象至少包括 Servlet 引擎的名字和版本号 |
| void log(String msg,Throwable t) | 写指定的信息到一个 Servlet 环境对象的 log 文件中 |
| setAttribute(String name,Object o) | 给予 Servlet 环境对象中你所指定的对象的一个名称 |
| removeAttribute(String name) | 从指定的 Servlet 环境对象中删除一个属性 |

**【示例 5.18】**下面我们编写一个实例 MyServletContextListener.java，使它能够对 ServletContext 以及属性进行监听。由以上介绍可知，该类需要实现 ServletContextAttribute Listener 和 ServletContextListener 接口类，其详细代码如图 5.80 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
public class MyServletContextListener implements
             ServletContextListener,ServletContextAttributeListener{
    private ServletContext context = null;
    public void contextInitialized(ServletContextEvent s) {
          this.context = s.getServletContext();
          print("ServletContext初始化......");
    }
    public void contextDestroyed(ServletContextEvent s) {
          this.context = null;
          print("ServletContext被释放......");
    }
    public void attributeAdded(ServletContextAttributeEvent sa) {
          print("增加ServletContext对象的一个属性:
               attributeAdded('"+sa.getName()+"',' "+sa.getValue()+"')");
    }
    public void attributeRemoved(ServletContextAttributeEvent sa) {
          print("删除ServletContext对象的某一个属性:
               attributeRemoved(' "+sa.getName()+"','");
    }
    public void attributeReplaced(ServletContextAttributeEvent sa) {
          print("更改ServletContext对象的某一个属性:
               attributeReplaced(' "+sa.getName()+"','"+sa.getValue()+"','");
    }
    private void print(String message){
          PrintWriter out = null;
          try{
            out = new PrintWriter(new FileOutputStream("D:\\output.txt",true));
            out.println(new java.util.Date().toLocaleString()
               +"ContextListener: "+message);
            out.close();
          }
          catch(Exception e)
          {
           e.printStackTrace();
          }
    }
}
```

实现ServletContextListener接口定义，对ServletContext进行初始化

用于释放ServletContext对象

当上下文属性进行添加、删除、更新时，将调用这些方法

图 5.80　MyServletContextListener.java 示例

在使用这个监听器之前还需要对 Web 模块中的 web.xml 配置文件进行配置，配置代码如图 5.81 所示。

```
<listener>
  <listener-class>servlets.MyServletContextListener</listener-class>
</listener>
```

listener对应类路径

图 5.81　MyServletContextListener.java 的 web.xml 配置

然后我们就可以编写一个 JSP 程序 testListener.jsp 来操作 ServletContext 的属性，看监听器程序做出什么反应，该 JSP 的代码如图 5.82 所示。

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/testListener.jsp 来测试这个程序，运行结果如图 5.83 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
  <head>
     <title>测试监听器</title>
  </head>
 <body>
  <%
   out.println("Test ServletContextListener"+ "<br>");
   application.setAttribute("userid","1234"); //添加一个属性
   application.setAttribute("userid","12345"); //替换掉已经添加的属性
   application.removeAttribute("userid"); //删除该属性
   out.println("监听器已做出反应，详情见D:\output.txt文件");
  %>
 </body>
</html>
```

图 5.82　testListener.jsp 示例


图 5.83　testListener.jsp 示例运行结果

我们可以打开 D 盘查看 output.txt 文件，会发现里面记录了监听器所做的动作，如图 5.84 所示。

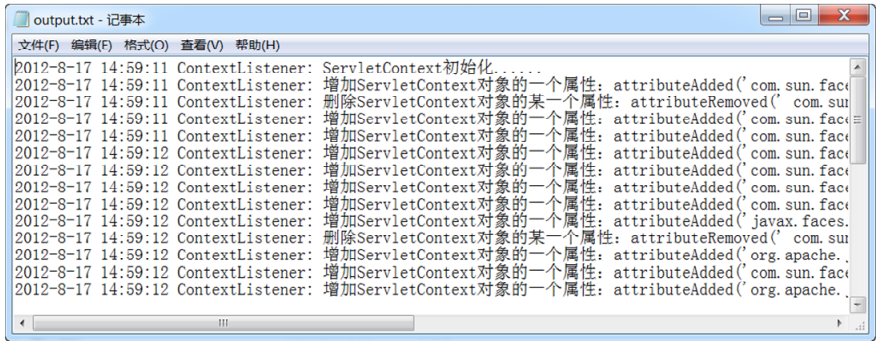



图 5.84　output.txt 文件

## 5.9.2　监听 HTTP 会话信息

HTTP 会话信息指的是 Session 对象的创建和销毁、会话中属性的设置请求、会话的状态和会话的绑定信息等。通过对 HTTP 会话信息的监听，可以进行一些很有用的操作，比如，统

计当前会话的数目、设置某个对话的属性、了解某个对话的状态等。与 ServletContext 监听的实现方法类似，对 HTTP 会话的监听也是通过实现特定的接口来完成的。监听 HTTP 会话信息需要使用到三个接口类：HttpSessionListener、HttpSessionActivationListener 和 HttpSessionAttributeListener 接口。这些接口的主要方法如表 5.6 所示。

表 5.6　监听HTTP会话信息的主要方法

| 方 法 名 | 方 法 描 述 |
|---|---|
| sessionCreated(HttpSessionEvent arg0)方法 | 进行 Http 会话创建的监听，如果 Http 会话被创建将调用该方法 |
| sessionDestroyed(HttpSessionEvent arg0)方法 | 对 Http 会话销毁进行监听，如果某个 Http 会话被释放将调用该方法 |
| sessionDidActivate(HttpSessionEvent arg0)方法 | 对 Http 会话处于 active 情况进行监听 |
| sessionWillPassivate(HttpSessionEvent arg0)方法 | 对 Http 会话处于 passivate 情况进行监听 |
| attributeAdded(HttpSessionBindingEvent arg0)方法 | 对 Http 会话中属性添加进行监听 |
| attributeReplaced(HttpSessionBindingEvent arg0)方法 | 对 Http 会话中属性修改进行监听 |
| attributeRemoved(HttpSessionBindingEvent arg0)方法 | 对 Http 会话中属性删除进行监听 |

**【示例 5.19】**下面我们编写一个实例 MySessionListener.java，使它能够对 HTTP 会话信息进行监听。该监听器实现了在线会话人数的统计，当一个会话创建时，users 变量将加一；当销毁一个会话对象时，users 变量将减一。具体代码如图 5.85 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class MySessionListener implements
   HttpSessionAttributeListener,HttpSessionListener{
   ServletContext context = null;
        int users = 0;
        public void attributeAdded(HttpSessionBindingEvent arg0) {
            print("attributeAdded('"+arg0.getSession().getId()+
            "','"+arg0.getName()+"','"+arg0.getValue()+"')");
        }
        public void attributeRemoved(HttpSessionBindingEvent arg0) {
            print("attributeRemoved('"+arg0.getSession().getId()+
            "','"+arg0.getName()+"','"+arg0.getValue()+"')");
        }
        public void attributeReplaced(HttpSessionBindingEvent arg0) {
            print("attributeReplaced('"+arg0.getSession().getId()+
            "','"+arg0.getName()+"','"+arg0.getValue()+"')");
        }
        public void sessionCreated(HttpSessionEvent arg0) {
           users++;
            print("sessionCreated('"+arg0.getSession().getId()+"'),"+
            "目前拥有"+users+"个用户");
            context.setAttribute("users",new Integer(users));
        }
        public void sessionDestroyed(HttpSessionEvent arg0){
            users--;
            print("sessionDestroyed('"+arg0.getSession().getId()+"'),"+
            "目前拥有"+users+"个用户");
            context.setAttribute("users",new Integer(users));
        }
        private void print(String message){
           PrintWriter out = null;
           try{
           out = new PrintWriter(new FileOutputStream("d:\\output.txt",true));
           out.println(new java.util.Date().toLocaleString()+
                       " SessionListener: "+message);
           out.close();
                  }
                  catch(Exception e)
                  {
                           e.printStackTrace();
                  }
        }
}
```

监听Http会话中的属性添加、删除、更新

Http会话的创建、释放监听

图 5.85　MySessionListener.java 示例

然后对 Web 模块中的 web.xml 配置文件进行配置，配置代码如图 5.86 所示。

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

图 5.86　MySessionListener.java 的 web.xml 配置

然后我们就可以编写一个 JSP 程序 testSessionListener.jsp 来看监听器程序做出什么反应，该 JSP 的代码如图 5.87 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
  <head>
     <title>测试监听器</title>
  </head>
 <body>
<%
 out.println("Test SessionListener");
 session.setAttribute("username","abc");//在Http会话中设置一个用户username属性
 session.setAttribute("username","abcd");//修改之上添加的username属性
 session.removeAttribute("username");//删除创建的username属性
 session.invalidate();//passivate Http会话
 out.println("监听器已做出反应，详情见D:\output.txt文件");
%>
 </body>
</html>
```

图 5.87　testSessionListener.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/testSessionListener.jsp，执行效果可以查看 output.txt 文档内容，如图 5.88 所示。

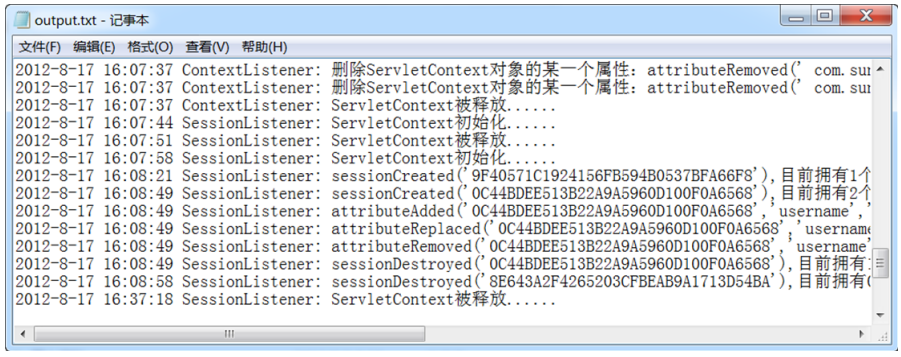



图 5.88　output.txt 文本文件

## 5.9.3　对客户端请求进行监听

客户端请求信息是指请求对象的创建、销毁以及其属性的添加、更改和删除。一旦可以对

客户端发向服务器的请求进行监听，就可以对它们进行识别，然后统一处理。对客户端请求信息的监听的实现方法与上面两种类似，可以通过实现 ServletRequestListener 和 ServletRequestAttributeListener 接口来完成。这些接口的主要方法如表 5.7 所示。

表 5.7　对客户端请求进行监听的主要方法

| 方 法 名 | 方 法 描 述 |
|---|---|
| ServletRequestListener()方法 | 监听客户端请求的创建和销毁 |
| attributeAdded(HttpSessionBindingEvent arg0)方法 | 对 Http 会话中属性添加进行监听 |
| attributeReplaced(HttpSessionBindingEvent arg0)方法 | 对 Http 会话中属性修改进行监听 |
| attributeRemoved(HttpSessionBindingEvent arg0)方法 | 对 Http 会话中属性删除进行监听 |

**【示例 5.20】**下面我们编写一个实例 MyRequestListener.java，使它能够实现客户端请求监听。例如现在系统要求通过监听客户端的属性信息，判断客户端请求是否是本机请求，如果是，不用登录，反之则要求其登录，其具体的实现代码如图 5.89 所示。

```
package servlets;
import java.io.*;
import javax.servlet.*;
public class MyRequestListener implements
            ServletRequestListener,ServletRequestAttributeListener{
    public void requestDestroyed(ServletRequestEvent arg0) {
       print("reqeust destroyed");
    }
    public void requestInitialized(ServletRequestEvent arg0) {
       print("Request init");
       ServletRequest sr = arg0.getServletRequest();
       print(sr.getRemoteAddr());
    }
    public void attributeAdded(ServletRequestAttributeEvent arg0) {
       print("attributeAdded('"+arg0.getName()+"','"+arg0.getValue()+"')");
    }
    public void attributeRemoved(ServletRequestAttributeEvent arg0) {
       print("attributeRemoved('"+arg0.getName()+"','"+arg0.getValue()+"')");
    }
    public void attributeReplaced(ServletRequestAttributeEvent arg0) {
       print("attributeReplaced('"+arg0.getName()+"','"+arg0.getValue()+"')");
    }
    private void print(String message){
       PrintWriter out = null;
       try{
         out = new PrintWriter(new FileOutputStream("d:\\output.txt",true));
         out.println(new java.util.Date().toLocaleString()
              +" RequestListener: "+message);
         out.close();
        }
       catch(Exception e)
         {  e.printStackTrace();  }
    }
}
```

对销毁和实现客户端请求进行监听

对属性的添加、删除、更新进行监听

图 5.89　MyRequestListener.java 示例

然后对 Web 模块中的 web.xml 配置文件进行配置，配置代码如图 5.90 所示。

```
<listener>
 <listener-class>servlets.MyRequestListener</listener-class>
</listener>
```

图 5.90　MyRequestListener.java 的 web.xml 配置

然后我们编写一个 JSP 程序 testRequestListener.jsp 来看监听器程序做出的反应，该 JSP 的代码如图 5.91 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<html>
  <head>
    <title>测试监听器</title>
  </head>
 <body>
 <%
   out.println("Test RequestListener"+  "<br>");
   request.setAttribute("username","abcd"); //在请求中设置一个用户username属性
   request.setAttribute("username","abcde"); //修改之上添加的username属性
   request.removeAttribute("username"); //删除创建的username属性
   out.println("监听器已做出反应，详情见D:\output.txt文件");
 %>
 </body>
</html>
```

图 5.91　testRequestListener.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/Servlet/testListener.jsp 来测试这个程序，运行结果如图 5.92 所示。

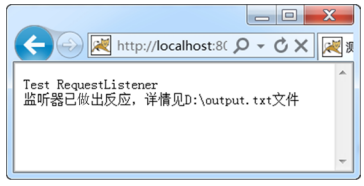



图 5.92　testRequestListener.jsp 运行结果

我们可以打开 D 盘中的 output.txt 文件，来查看其发生了什么变化，如图 5.93 所示。

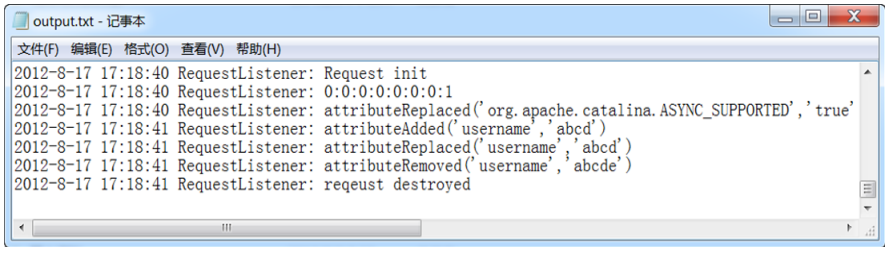



图 5.93　output.txt 文件

## 5.10 小结

本章首先介绍了 Servlet 编程方面的基础知识，然后在此基础上介绍了 Servlet 的配置和处理方法，接着我们为大家讲解了如何利用 JSP 页面调用 Servlet 和有关 Servlet 的文件操作，最后我们通过实例讲解了 Servlet 的具体应用和过滤器、监听器的知识。本章的重点是 Servlet 的文件操作以及 Servlet 应用方面的知识，难点是 Session 技术以及 Servlet 过滤器、监听器知识的理解和应用。熟练掌握 Servlet 是学好 Java Web 技术的基本要求，所以读者要多加练习，以打好基础。

## 5.11 本章习题

1．请读者编写一个 Servlet，通过该 Servlet 来接受一个学生表单中输入的信息，并最终将学生信息在网页中打印输出，执行结果如图 5.94 所示。

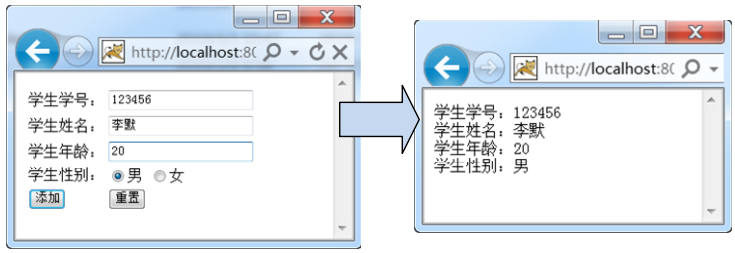



图 5.94 运行结果

【分析】本题主要考查读者通过提交表单调用 Servlet 的能力。在这个题目中，我们要清楚把表单的 action 指向对应的 Servlet。

【核心代码】本题的关键代码如下所示。

PrintStudentServlet.java：

```
public class PrintStudentServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        request.setCharacterEncoding("gb2312");
        response.setCharacterEncoding("gb2312");
        String stuNO = request.getParameter("stuNO");
    ......
        PrintWriter out = response.getWriter();
        out.println("<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.01 Transitional
//EN\">");
    ......
        out.flush();
        out.close();
    }
    public void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        doGet(request, response);
    }
}
```

StudentForm.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <body>
     <form action="servlet/PrintStudentServlet" method="post">
         <table>
             <tr>
                 <td>学生学号：</td>
                 <td><input type="text" name="stuNO"/></td>
             </tr>
             ……
             <tr>
                 <td><input type="submit" value="添加"></td>
                 <td><input type="reset" value="重置"></td>
             </tr>
         </table>
     </form>
  </body>
</html>
```

2．请读者参照【示例 5.7】来向一个文件 new2.txt 中写入字符串“I love Java Web!”。这个 Servlet 的执行结果如图 5.95 所示。

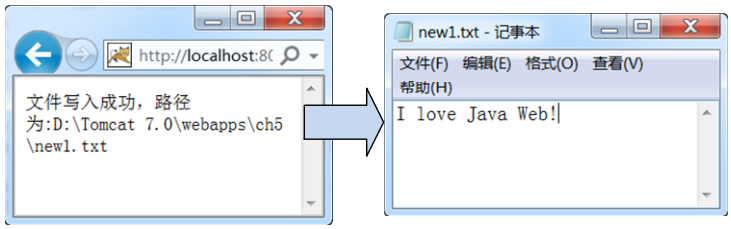



图 5.95　运行结果

【分析】本题主要考查读者通过 Servlet 写文件的能力。在 Servlet 中我们可以使用输入输出流实现对文件的读写。在这个题目中，我们要清楚通过 Servlet 写文件的方法。

【核心代码】本题的关键代码如下所示。

FileWrite.java:

```
public class FileWrite extends HttpServlet {
   public void doGet(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException
   {
      response.setCharacterEncoding("gb2312");
      PrintWriter out = response.getWriter();
      String fileName =  "new1.txt";
      String realPath = request.getRealPath(fileName);
      File file = new File(realPath);
      FileWriter writer = new FileWriter(file);
      BufferedWriter bufferWriter = new BufferedWriter(writer);
      bufferWriter.write("I love Java Web!");
   ……
      out.print("<font size='4'>文件写入成功，路径为:"+file.getAbsolutePath()+"</font>");
   }
   public void doPost(HttpServletRequest request, HttpServletResponse response)
   throws IOException, ServletException
  ……
}
```

web.xml 文件配置：

```
<servlet>
    <servlet-name>FileWrite</servlet-name>
    <servlet-class>servlet.FileWrite</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FileWrite</servlet-name>
    <url-pattern>/FileWrite</url-pattern>
</servlet-mapping>
```

3．请读者使用 Filter 实现一个非法文字过滤器。其中用户发言表单中如果出现“臭”这个字符，就弹出提示页面，提示“发言失败，含有非法文字”信息，执行结果如图 5.96 所示。

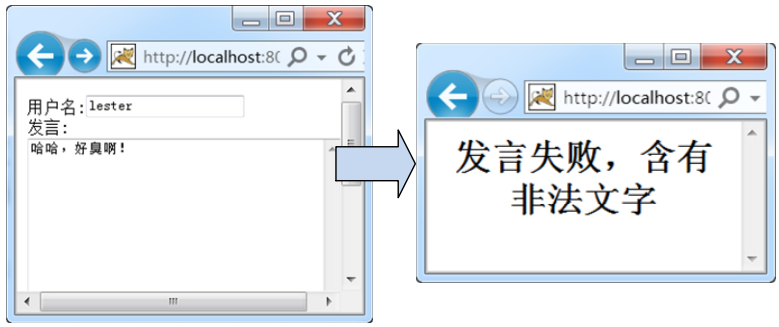



图 5.96　运行结果

【分析】本题主要考查读者运用过滤器的方法和配置的能力。我们可以使用过滤器实现特定字符的屏蔽措施。读者可以参考【示例 5.16】和【示例 5.17】实现过滤器的操作。

【核心代码】本题的关键代码如下所示。

CharFilter.java：

```
public class CharFilter implements Filter{
    public void init(FilterConfig filterConfig) throws ServletException {
        System.out.println("非法文字过滤器初始化");
    }
    public void doFilter(ServletRequest req, ServletResponse res,
            FilterChain chain) throws IOException, ServletException {
        HttpServletRequest request = (HttpServletRequest)req;
        String charContent = request.getParameter("charContent");
         charContent = new String(charContent.getBytes("ISO-8859-1"),"gb2312");
        if(charContent != null) {
            if(charContent.indexOf("臭")== 1) {
                chain.doFilter(req, res);
            } else {
                request.getRequestDispatcher("SendFailure.jsp").forward(req, res);
            }
        } else {
            chain.doFilter(req, res);
        }
    }
    public void destroy() {
        System.out.println("非法文字过滤器销毁");
    }
}
```

CharForm.jsp：

```
<%@ page language="java" contentType="text/html;charset=gb2312"%>
<html>
<body>
    <%--表单提交方式为 post，提交到 DoGetDemo--%>
    <form action="ShowContent.jsp" method="post">
        用户名:<input type="text" name="username"/><br>
        发言:<br>
        <textarea name="charContent" rows="20" cols="40"></textarea><br>
        <input type="submit" value="发送"/>
        <input type="reset" value="重置"/> 
    </form>
</body>
</html>
```

# 第 6 章　用户自定义标签

JSP 自定义标签是用户定义的 JSP 语言元素，可以看成是一种通过标签处理器生成基于 XML 脚本的方法。自定义标签在功能上和逻辑上都与 JavaBean 类似，都是一组可重用的组件代码。相较于 JavaBean，自定义标签可以使 Web 开发者可以完全从 Java 编程中脱离开来，专注于页面显示和格式上面去，所以具有广阔的发展前景。本章我们就为大家来讲解有关用户自定义标签的知识。

## 6.1　自定义标签概述

在第 4 章中我们讲解了 JavaBean，知道 JSP 专门提供了 3 个动作指令来调用 JavaBean 组件，简化 JSP 页面的开发和维护。但是，这远远不能满足实际开发的需要。因此，在 JSP 技术中提供了一种新的封装动态功能的机制，这就是用户自定义标签。

### 6.1.1　自定义标签的构成

一个自定义标签一般由 JavaBean、标签库描述、标签处理器、web.xml 文件配置、标签库声明等元素构成，它们的作用如图 6.1 所示。

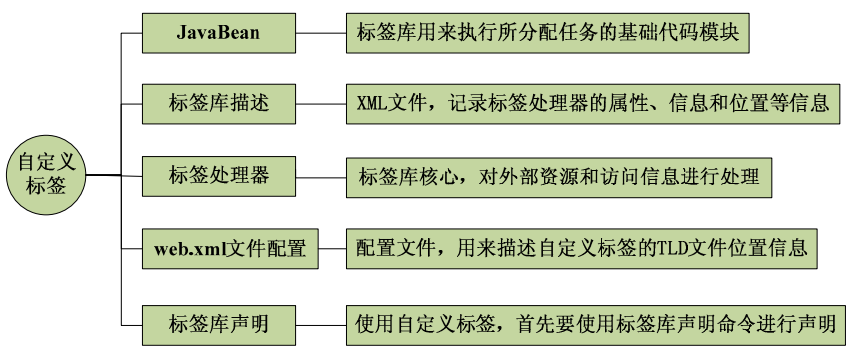



图 6.1　自定义标签的构成

JavaBean、web.xml 文件配置比较简单。以下仅对标签库声明、标签库描述和标签处理器进行简要介绍。

### 6.1.2　自定义标签声明

我们在第 2 章中介绍过 taglib 指令。该指令就是当 JSP 页面中引用自定义标签时，用来在页面上对自定义标签进行声明的。taglib 编译指令的作用主要是定义一个标签库路径及其前缀。taglib 指令的语法格式如图 6.2 所示。

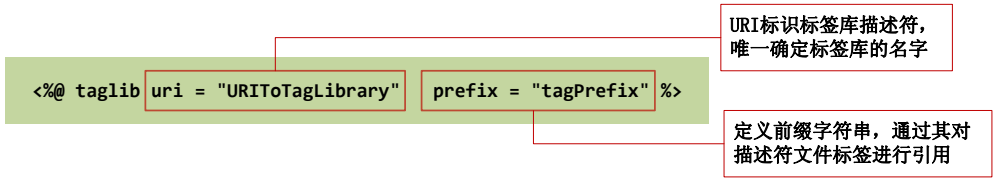



图 6.2　taglib 指令的语法格式

**注意：**无论 JSP 页面中的自定义标签出现在什么位置，taglib 指令都必须出现在页面的前端位置，如图 6.3 所示。

```
<%@ page contentType="text/html;charset=GBK"%>
<%@ taglib prefix="mytag" uri="/mytag" %>
<html>
    <head><title>简单标签实例</title></head>
  <body>
     <h3>……</h3>
     调用标签：<mytag:print />
  </body>
</html>
```

taglib指令位置

图 6.3　taglib 指令的位置

### 6.1.3　标签库描述符文件

标签库描述符（TLD 文件）是一个描述标签库的 XML 文档。TLD 包含有关整个库以及库中包含的每一个标签的信息。它把自定义标签与对应的处理程序关联起来。TLD 文件名称的扩展名必须为.tld。TLD 文件存储在 Web 模块的 WEB-INF 目录下或者子目录下，并且一个标签库要对应一个标签库描述文件，而在一个描述文件中可以包含多个自定义标签的声明。

标签库描述符文件的根元素是<taglib>，该元素下包含如表 6.1 所示的子元素。

表 6.1　<taglib>子元素

| 元　素 | 说　明 | 元　素 | 说　明 |
|---|---|---|---|
| <tlib-version> | 用于设置标签库版本 | <small-icon> | 用于设置标签库的可选小图标 |
| <jsp-version> | 用于设置标签库要求的 JSP 规范版本 | <large-icon> | 用于设置标签库的可选大图标 |
| <short-name> | 用于设置该标签库的助记名 | <description> | 用于设置标签库的描述信息 |
| <uri> | 唯一标识该标签库的 URI | <listener> | 用于设置标签库的监听器类 |
| <display-name> | 用于设置标签库显示的可选名 | <tag> | 用于设置标签库的具体标签 |

通过表 6.1 我们可以看出，<taglib>元素中大部分子元素都是对标签库的一些基本属性或者显示的名称或图表的设定，并不具备实际意义。真正用来查找标签库中具体标签的是<tag>元素。<tag>元素也包括子元素，其具体说明如表 6.2 所示。

表 6.2　<tag>子元素

| 元　素 | 说　明 | 元　素 | 说　明 |
|---|---|---|---|
| <name> | 用于设置标签的唯一名称 | <small-icon> | 用于设置标签的可选小图标 |
| <tag-class> | 用于设置标签处理器的完全限定名 | <large-icon> | 用于设置标签的可选大图标 |
| <tei-class> | 用于设置脚本变量信息的子类名称 | <description> | 用于设置标签的描述信息 |
| <body-content> | 用于设置标签的正文内容类型 | <variable> | 用于设置标签的脚本变量信息 |
| <display-name> | 用于设置标签显示的可选名 | <attribute> | 用于设置标签的属性信息 |

**【示例 6.1】**根据对 TLD 文件中标签的介绍，我们可以举一个例子来展示标签描述符文件的典型格式，如图 6.4 所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
   xmlns:xsi="http://www.w3.org/2001/-instance"
   xsi:="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-_2_0.xsd" version="2.0">

 <description>mytag 1.1 print library</description> <!-- 对标签库进行描述 -->
 <display-name>mytag标签库</display-name> <!--  可由工具显示的选择名 -->
 <tlib-version>1.0</tlib-version> <!-- 此标签库的版本 -->
 <jsp-version>1.2></jsp-version> <!-- 这个标签库需要的JSP规范版本 -->
 <short-name>mytag</short-name> <!-- JSP页面编写工具用来创建助记名的可选名字 -->
 <uri>http://www.tag.com/mytag</uri> <!-- 唯一标识该标签库的URI -->
 <tag>
   <description>打印页面 taglib</description> <!-- 此标签的描述信息 -->
   <name>print</name> <!-- 唯一标签名 -->
   <tag-class>cn.com.zzb.tag.</tag-class> <!-- 定义这个标签对应的处理类文件 -->
   <body-content>JSP</body-content> <!-- 体中正文内容类型 -->
   <attribute>
     <name>value</name> <!-- 开发标签中包括的属性 -->
     <required>true</required> <!-- 是否为必选属性 -->
   </attribute>
   <attribute>
     <name>type</name>
     <required>false</required>
   </attribute>
 </tag>
</taglib>
```

描述性文件头部

tag元素

图 6.4　描述性文件典型格式

### 6.1.4　标签处理器

把自定义标签的主体和属性转变为 HTML 代码的实际工作，是由标签处理器来完成的。标签处理器也叫标签处理类，它是一个 Java 类。当 JSP 容器编译自定义标签时，就会需要使用标签处理器类的实例。

标签处理器虽然是一个 Java 类，但不仅仅是一个普通的 Java 类，在定义时需要满足特殊的要求。开发的标签处理类必须实现 Tag 或者 BodyTag 接口类（它们的包为 javax.servlet.jsp.tagext）。BodyTag 接口是继承了 Tag 接口的子接口。如果创建的自定义标签不带体式，可以实现 Tag 接口，但是如果创建的自定义标签带体，则需要实现 BodyTag 接口。

Tag 接口类中所定义的方法如表 6.3 所示。

表 6.3　Tag接口的方法及方法描述

| 方 法 名 | 方 法 描 述 |
|---|---|
| setPageContext(PageContext pc) | 设置当前页面的上下文 |
| setParent(Tag t) | 设置这个标签处理类的父类 |
| getParent() | 获得父类 |
| doStartTag() | 处理这个实例中的开发标签 |
| doEndTag() | 处理这个实例中的结束标签 |
| release() | 由标签处理类引起，来释放状态 |

BodyTag 子接口类又重新定义了两个新方法，如表 6.4 所示。

表 6.4　BodyTag接口的方法及方法描述

| 方 法 名 | 方 法 描 述 |
|---|---|
| setBodyContent(BodyContent b) | 为体中代码做初始化 |
| doInitBody() | 为标签体中的内容设置属性 |

在标签处理器中定义了标签处理方法 doStartTag()和 doEndTag()，这两个方法分别在标签开始和结束时执行处理和输出动作。这两个方法都要求分别返回一个状态码，通知 JSP 容器自定义标签的处理结果及整个 JSP 页面的运行状态。状态码一共有四种，具体作用如图 6.5 所示。

| | |
|---|---|
| EVAL_BODY_INCLUDE | 当doStartTag()返回时，指明Servlet应对标签体进行评估 |
| SKIP_BODY | 当doStartTag()返回时，指明Servlet应忽视标签体 |
| EVAL_PAGE | 当doEndTag()返回时，指明页面其余部分应被评估 |
| SKIP_PAGE | 当doEndTag()返回时，指明页面其余部分应被跳过 |

图 6.5　四种返回值

标签处理器也有其生命周期，其大致可以分为 5 个阶段，如图 6.6 所示。

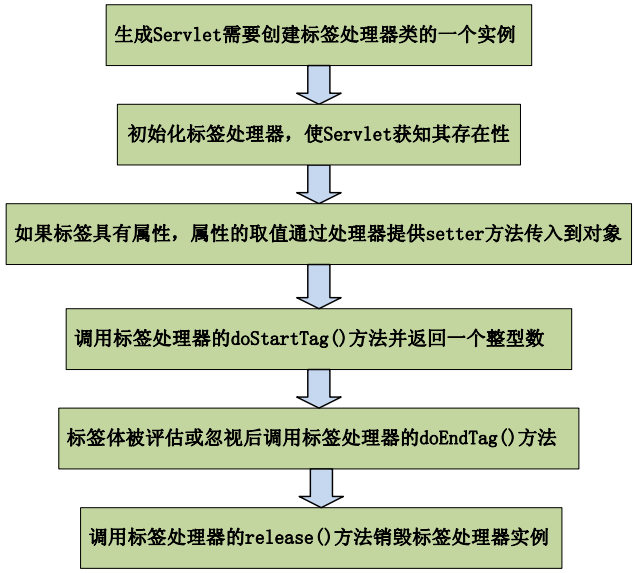



图 6.6　标签处理器生命周期

下面我们就在自定义标签的基本概念描述的基础上，列举一系列的自定义标签开发的实例，来教会读者如何开发各类自定义标签。

## 6.2　简单格式的标签开发

简单格式的标签没有属性和体，它必须实现 Tag 接口中的 doStartTag()和 doEndTag()方法。当 Web 容器遇到开始标签时会自动调用 doStartTag()方法。由于简单格式的标签没有体，所以这个方法会直接返回一个 SKIP_BODY。在遇到结束标签时会调用 doEndTag()方法。如果还需要页面中的其他部分进行判断，则 doEndTag()方法会返回 EVAL_PAGE，否则，会返回 SKIP_PAGE。

**【示例 6.2】**我们举一个例子创建一个简单格式的标签<mytag:print />。这个标签没有属性，也没有体的部分，直接打印一个“Hello”字符串。首先我们先创建一个标签处理类：MytagPrint.java，具体代码如图 6.7 所示。

```
package tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;

public class MytagPrint  extends TagSupport{
    public int doStartTag() throws JspException{
       try {
            pageContext.getOut().print("Hello Tag!");
           }catch (Exception e) {
         throw new JspTagException("SimpleTag: " + e.getMessage());}
         return SKIP_BODY;
        }
   public int doEndTag() {
         return EVAL_PAGE;
        }
}
```

打印HelloTag!字符串

需要对页面其他部分进行判断，否则返回SKIP_PAGE

图 6.7　MytagPrint.java 示例

我们在 6.1.3 小节中已经介绍过了描述性文件 TLD 的编写格式，一个 TLD 文件对应着一个自定义的标签库，在这个文件中可以申明一组自定义的标签。首先在 Web 模块的 WEB-INF 目录下创建一个 mytag.tld 描述性文件，文件内容如图 6.8 所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee" xmlns:xsi="http://www.w3.org/2001/-
instance"  xsi:="http://java.sun.com/xml/ns/j2ee  http://java.sun.com/xml/ns/j2ee/
web-_2_0.xsd"  version="2.0">
        <!--上面是描述性文件的头部 -->
 <description>mytag 1.1 print library</description>
 <display-name>mytag标签库</display-name>
 <tlib-version>1.0</tlib-version>
 <short-name>mytag</short-name>
 <uri>/mytag</uri>

 <tag>
   <name>print</name>
   <tag-class>tags.</tag-class>
   <body-content>empty</body-content>
 </tag>
</taglib>
```

tag元素属性设置

图 6.8　mytag.tld 文件示例

然后再对 web.xml 文件进行配置，即在 web.xml 配置文件的</web-app>前面加上如图 6.9 所示的一段代码。

```
<taglib>
<taglib-uri>/mytag</taglib-uri>
<taglib-location>/WEB-INF/mytag.tld</taglib-location>
</taglib>
```

图 6.9　web.xml 配置文件配置方法

最后我们要做的就是在 JSP 文件中声明和使用自定义标签了，要在 JSP 中使用自定义的标签，必须要使用 taglib 指令来声明引用的标签库。例如，创建一个 mytag_print.jsp 文件，源代码如图 6.10 所示。

```
<%@ page contentType="text/html;charset=GBK"%>
<%@ taglib prefix="mytag" uri="/mytag" %>
<html>
    <head><title>简单标签实例</title></head>
  <body>
     <h3>调用mytag标签库中的print标签</h3>
         调用print标签的结果：<mytag:print />
  </body>
</html>
```

图 6.10　mytag_print.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/Tag/mytag_print.jsp 来测试这个示例，显示结果如图 6.11 所示。

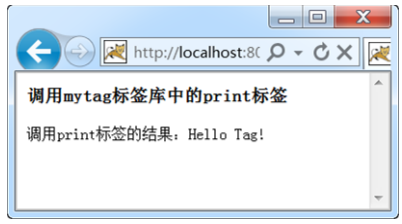



图 6.11　mytag_print.jsp 示例运行结果

## 6.3　自定义带有属性的标签

自定义标签可以有自己的属性。属性一般在开始标记中定义，语法为 attr="value"。而且对于每个 value 属性，还需要在这个标签相对应的处理类中定义一个属性的 set()和 get()方法。

**【示例 6.3】**我们下面通过一个例子来说明如何定义带有属性的标签。首先，要在 Tag 项目的包 tags 中创建一个 MytagPrinta.java 的标签处理类，其源代码如图 6.12 所示。

```
package tags;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.*;
public class MytagPrinta extends TagSupport {
        protected String value = null;
        protected String type = "0";
        public void setValue(String value){
                this.value = value; }
        public String getValue(){
                return value; }
        public void setType(String type){
                this.type = type; }
        public String getType(){
                return type; }
        public int doStartTag() throws JspException{
            try {
                if(type.equals("0"))
                pageContext.getOut().print("Hello Tag!"+value);
                else
                pageContext.getOut().print ("Hello Tag! "+value+"<br>");
                }catch (Exception ex) {
                throw new JspTagException ("attrTag: " + ex.getMessage());}
                return SKIP_BODY;
        }
   public int doEndTag() {
                return EVAL_PAGE;
        }
}
```

定义一个value和type的set()和get()方法

字符串输出后是否换行

图 6.12　MytagPrinta.java 示例

mytag 标签库的描述性文件已经在上一小节中创建了，这里只需要把创建的新标签 printa

在 mytag.tld 文件中进行定义。即把如图 6.13 所示的一段代码插入到 mytag.tld 文件的<taglib>元素中。

```
<tag>
  <description>带有type 和 value属性的printa标签</description>
  <name>printa</name>
  <tag-class>tags.</tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>type</name>
    <required>false</required>
    <>true</>
  </attribute>
  <attribute>
    <name>value</name>
    <required>true</required>
    <>true</>
  </attribute>
</tag>
```

指定属性的名称，区分大小写

规定该属性是否必须指定，true表示必须，false表示可选

图 6.13　mytag.tld 文件

由于 mytag 标签库已经在 web.xml 配置文件中进行了声明，所以这里就跳过这一步骤。我们直接创建一个 mytag_printa.jsp 文件，用来调用新创建的<mytag:printa>标签，其源代码如图 6.14 所示。

```
<%@ page contentType="text/html;charset=GBK"%>
<%@ taglib prefix="mytag" uri="/mytag" %>
<html>
<head><title>带属性标签实例</title></head>
<body>
        <h3>调用mytag标签库中的printa标签</h3>
        打印后不换行：<mytag:print value="print标签" type="0" />
        打印后换行：   <mytag:printa value="printa标签" type="1" />
</body>
</html>
```

图 6.14　mytag_printa.jsp 文件示例

我们在浏览器地址栏中输入 http://localhost:8080/Tag/mytag_printa.jsp，显示结果如图 6.15 所示。

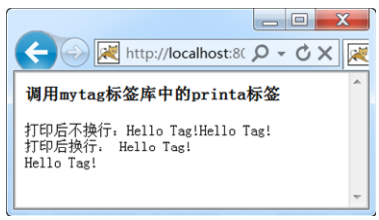



图 6.15　mytag_printa.jsp 示例运行结果

## 6.4 自定义带有体的标签

之前我们定义的标签都是不带体的，接下来我们看如何创建一个带体的标签。一个自定义标签可以包含其他自定义标签、脚本变量、HTML 标记或其他内容。而且其必须继承 javax.servlet.jsp.tagext.BodyTagSupport 类，实现其中的 doInitBody()和 doAfterBody()方法。这两种方法的说明如表 6.5 所示。

表 6.5 BodyTagSupport类的两种方法说明

| 方 法 名 | 方 法 描 述 |
| --- | --- |
| doInitBody()方法 | 用这个方法执行所有依赖于正文内容的初始化，对正文内容进行判断之前调用 |
| doAfterBody()方法 | 返回指明是否继续判断体中正文内容的指示，在判断了正文内容之后调用 |

下面还是通过实例来讲解如何创建一个带体的自定义标签。

**【示例 6.4】**首先，要在 Tag 项目的包 tags 中创建一个 LowerCaseTag.java 的标签处理类，其源代码如图 6.16 所示。

```
package tags;
import java.io.*;
import javax.servlet.jsp.*;
import javax.servlet.jsp.tagext.BodyContent;
import javax.servlet.jsp.tagext.BodyTagSupport;
public class LowerCaseTag extends BodyTagSupport{
   public int doAfterBody()throws JspException{        遇到标签体执行该方法
      try{
        BodyContent body=getBodyContent();
        JspWriter writer=body.getEnclosingWriter();    获取向页面的输入流对象
        String bodyString= body.getString();
        if(bodyString!=null){
            writer.print(bodyString.toLowerCase());}   对标签体进行大小写转换
            } catch (IOException ioe) {
            throw new JspException(
            "Error:IOException while writing to the user");
                  }
            return SKIP_BODY;
            }
}
```

图 6.16 LowerCaseTag.java 示例

然后，在项目中的 WEB-INF 目录下创建一个名为 bodytags 的标签描述符文件，其具体代码如图 6.17 所示。

接下来，在项目的 web.xml 的配置文件中添加标签描述符文件的位置指定信息，其具体代码如图 6.18 所示。

```
<taglib>
 <tlib-version>1.0</tlib-version>
 <jsp-version>2.0</jsp-version>
 <short-name>BodyTag library</short-name>
 <tag>
   <name>lowercase</name>
   <tag-class>tags.</tag-class>
   <body-content>JSP</body-content>
<description>Put body in lowercase</description>
 </tag>
</taglib>
```

图 6.17　bodytags.tld 示例

```
<taglib>
<taglib-uri>/lowercase</taglib-uri>
<taglib-location>/WEB-INF/bodytag.tld</taglib-location>
</taglib>
```

图 6.18　web.xml 配置文件信息

最后，在项目中创建使用自定义标签的页面 bodytags.jsp，其具体代码如图 6.19 所示。

```
<%@ page contentType="text/html;charset=GBK"%>
<%@ taglib prefix="jspx" uri="/lowercase" %>
<html>
<head><title>带体标签实例</title></head>
<body>
        <h3>调用标签库中的lowercase标签</h3>
        <jspx.lowercase>I love Java Web!</jspx.lowercase>
</body>
</html>
```

使用标签输出字符串

图 6.19　bodytags.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/Tag/bodytags.jsp，显示结果如图 6.20 所示。


图 6.20　bodytags.jsp 示例运行结果

## 6.5　自定义嵌套标签

到目前为止，我们创建的标签都是单个的标签，也是被单独的应用在 JSP 页面中的。而在

实际开发中，往往需要通过多个标签来实现特定的功能，这样的标签就存在嵌套关系。存在嵌套关系的标签也可以被称为父子标签，一个父标签可以嵌套多个子标签、HTML 和 Java 片段代码。

**【示例 6.5】**我们通过一个示例来看如何进行自定义嵌套标签的开发和使用。本实例实现了一对嵌套标签，其中父标签用来进行条件判断，子标签用来将属性输出到页面上面。

首先，在项目中创建名称为 IfTag 的父标签的标签处理类，其具体代码如图 6.21 所示。

```
package tags;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.BodyTagSupport;
public class IfTag extends BodyTagSupport
{
    private boolean value;
    public void setValue(boolean value)
      {
            this.value=value;
      }
    public int doStartTag() throws JspTagException
      {
        if(value)
      {
          System.out.println("value is true");
          return EVAL_BODY_INCLUDE;
        }
        else
        {
          System.out.println("value is false");
          return SKIP_BODY;
        }
     }
   public int doEndTag() throws JspTagException
   {
       try
       {
          if(bodyContent != null)
           {
            bodyContent.writeOut(bodyContent.getEnclosingWriter());
           }
       }
       catch(java.io.IOException e)
       {
          throw new JspTagException("IO Error: " + e.getMessage());
       }
       return EVAL_PAGE;
   }
}
```

doStartTag方法，如果value为true，就计算tagbody值，否则不计算body值

覆盖doEndTag方法

图 6.21　IfTag.java 文件示例

然后，我们在项目中再创建一个名为 OutTag 的子标签的标签处理类，其具体代码如图 6.22 所示。

接下来，在项目中的 WEB-INF 目录下创建名为 mytag1.tld 的标签描述符文件，其具体代码如图 6.23 所示。

```
package tags;
import java.io.IOException;
import javax.servlet.jsp.JspTagException;
import javax.servlet.jsp.tagext.TagSupport;
public class OutTag extends TagSupport
{
    private Object value;
    public void setValue(Object value)
    {
      this.value=value;
    }
    public int doStartTag() throws JspTagException {
      return EVAL_BODY_INCLUDE;
    }
    public int doEndTag()throws JspTagException
     {
       try
         {
           System.out.println(value);
           pageContext.getOut().write(value.toString());
         }
       catch(IOException ex)
         {
           throw new JspTagException(
            "Fatal error:hello tag conld not write to JSP out");
         }
           return EVAL_PAGE;
        }
}
```

定义起始标签的方法，表示继续执行标签体

定义结束标签的方法，将属性内容输出到页面

图 6.22　OutTag.java 文件示例

```
<tag>
    <name>if</name>
      <tag-class>tags.IfTag</tag-class>
      <body-content>jsp</body-content>
    <attribute>
        <name>value</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
```

父标签定义

```
<tag>
    <name>out</name>
<tag-class>tags.OutTag</tag-class>
       <body-content>jsp</body-content>
    <attribute>
          <name>value</name>
          <required>true</required>
          <rtexprvalue>true</rtexprvalue>
    </attribute>
</tag>
```

子标签定义

图 6.23　mytag1.tld 示例

接下来，在项目的 web.xml 的配置文件中添加标签描述符文件的位置指定信息，其具体代

码如图 6.24 所示。

```
<taglib>
<taglib-uri>/mytag</taglib-uri>
<taglib-location>/WEB-INF/mytag1.tld</taglib-location>
</taglib>
```

图 6.24　web.xml 配置文件配置方法

最后，在项目中创建使用自定义标签的页面 vcorworktag.jsp，其具体代码如图 6.25 所示。

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ taglib uri="/demotag" prefix="vt" %>
<html>
<head>
<title>嵌套标签实例</title>
<meta http-equiv="Content-Type" content="text/html; charset=gb2312">
</head>
<body>
<p>嵌套标签实例</p>
<HR><br>
<%
        String outValue="故宫、天坛、天安门、长城……";
%>
<vt:if value="true">
 <vt:out value="<%=outValue%>">
 北京的名胜古迹:<br>
 </vt:out>
</vt:if>
<HR>
<vt:if value="false">
<vt:out value="<%=outValue%>">
 这些内容会显示在客户端
 </vt:out>
</vt:if>
<br>
</BODY>
</HTML>
```

输出内容

图 6.25　vcorworktag.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/Tag/vcorworktag.jsp 来测试这个示例，显示结果如图 6.26 所示。

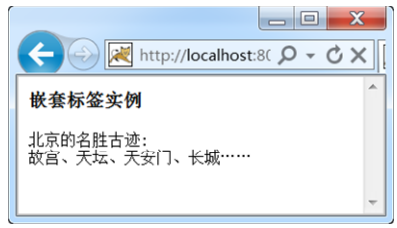



图 6.26　vcorworktag.jsp 示例运行结果

## 6.6　小结

本章主要讲解了 JSP 自定义标签的基本开发流程，并且通过具体实例详细介绍了各种类型自定义标签在具体实现时的技术细节和应用场合。本章的重点和难点都是熟练掌握包括带有属性、带有体以及含有嵌套的标签的定义和使用方法。通过本章的学习，读者应对 JSP 技术有了更深层次的理解，并为后面学习 JSTL 标签库及各种 Java Web 框架中的标签库打下扎实的基础。

## 6.7　本章习题

1．请读者实现一个简单的自定义标签，这个标签的功能只是简单地输出当前的系统时间。执行效果如图 6.27 所示。

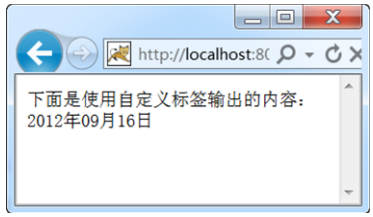



图 6.27　运行结果

【分析】本题主要考查读者对于自定义标签的掌握程度。开发自定义标签，首先需要开发标签所对应的功能类，其次要编写标签的描述文件 tld，并把这个文件放在项目的 WEB-INF/ 目录下，然后才可以在 JSP 页面上调用自定义的标签。

【核心代码】本题的核心代码如下所示。

DateTag.java:

```
package taglibs;
import java.io.IOException;
……;
public class DateTag implements Tag {
  private PageContext pageContext;
  private Tag tag;
  public int doEndTag() throws JspException {
     try {
        Date date = new Date();
        SimpleDateFormat dateFormater = new SimpleDateFormat("yyyy年MM月dd日");
         pageContext.getOut().print(dateFormater.format(date));
     } catch (IOException e) {
         e.printStackTrace();
     }
     return Tag.EVAL_PAGE;
  }
  public int doStartTag() throws JspException {
     return Tag.SKIP_BODY;
  }
  ……
}
```

datetag.tld：

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE taglib  PUBLIC "-//Sun Microsystems, Inc.//DTD JSP Tag Library 1.2//EN"
   "http://java.sun.com/dtd/web-jsptaglibrary_1_2.dtd">
<taglib>
 ……
   <short-name>dateTagExample</short-name>
   <uri>/mytags</uri>
   <tag>
      <name>date</name>
      <tag-class>taglibs.DateTag</tag-class>
      <body-content>empty</body-content>
   </tag>
</taglib>
```

dateTag.jsp：

```
<%@ page contentType="text/html; charset=gb2312" %>
<%@ taglib uri="/WEB-INF/datetag.tld" prefix="mytags" %>
<html>
……
<body>
<font size="4">
   下面是使用自定义标签输出的内容：<br>
   <mytags:date/>
</font>
</body>
</html>
```

# 第 7 章　EL 与 JSTL

（JSP Standard Tag Library，JSP 标准标签库）是一个不断完善的开放源代码的 JSP 标签库，是由 apache 的 jakarta 小组来维护的。JSTL 标签库的出现，不仅简化了 JSP 和 WEB 应用程序的开发，而且在应用程序服务器之间提供了一致的接口，最大程度地提高了 WEB 应用在各应用服务器之间的移植。而由 JSTL 1.0 发展而来的 EL 语言更加简化了 JSP 页面中对变量和对象的访问操作。本章我们就来带领大家一起学习 EL 表达式语言和 JSTL 标签库的相关知识。

## 7.1　EL 简介

表达式语言是 JSP 2.0 的一个新特性，全名为 Expression Language，简称 EL。EL 能实现对 pageContext 对象、session 对象、request 对象等存储对象的简化访问，能够简洁地访问请求参数、Cookie 和其他请求数据，即 EL 可以很方便地访问大多数 JSP 内置的隐含对象，从而简化编程。此外，EL 还可以简化对 JavaBean 属性和集合元素的访问。

EL 语法很简单，使用非常方便。其语法格式如图 7.1 所示。

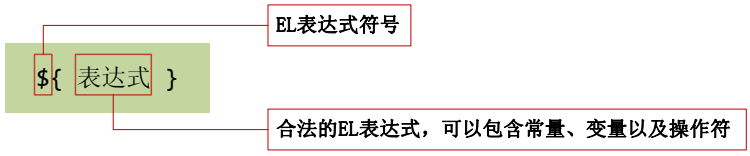



图 7.1　EL 语法格式

我们先来看几个使用 EL 表达式的示例：

```
${ 1+2+3 }   //计算 1+2+3 的值并将结果返回
${ username }   //查找并返回 username 的值
${ user.name }   //访问 JavaBean 对象 user 的属性 name
```

用 EL 语言中操作对象时，可以非常简单地使用各种算术、关系、逻辑或空值测试运算符，来简化操作运算。如果要针对不同情况和条件进行不同信息的输出，将不再需要采用 Java 语言编程，就可以轻松地实现条件化输出，从而消除大部分不必要的类型转换，同时也省略掉很多将字符串解析成数字的代码，实现了自动类型转换。例如，实现将用户输出的参数加 5 后的和输出到页面上这一功能，使用 JSP 传统语法的具体代码如图 7.2 所示。

```
<%
  String str_count = request.getParameter("count");
  int count = Integer.parseInt(str_count);
  count = count+5;
  out.println("count:"+count);
%>
```

图 7.2　JSP 传统示例

而使用 EL 实现同样的功能只需要如下简单的一行代码：

```
count: ${ param.count + 5}
```

大多数 Java Web 服务器都是默认支持 EL 的。对于单个 JSP 页面，可以使用 page 指令来设置 JSP 页面是否支持 EL。JSP 页面默认支持 EL，如果不支持的话，我们可以通过设置 page 指令的 isELIgnored 属性为 false，来实现对 EL 的支持，其具体格式如图 7.3 所示。

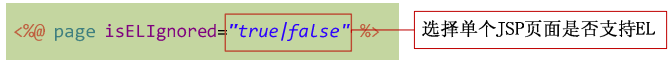



图 7.3　设置 JSP 页面是否支持 EL

**注意**：isELIgnored 属性表示是否忽略 EL。由于我们需要使用 EL，所以我们将其设置为 false。

而对于整个 JSP 应用，要修改 Web 应用的 web.xml 配置文件来设置是否支持 EL。如果要使整个 JSP 应用都支持 EL，则设置<jsp-property-group>元素的子元素<el-ignored>的值为 false，具体格式如图 7.4 所示。

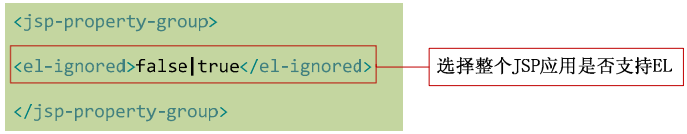



图 7.4　设置 JSP 应用是否支持 EL

**注意**：el-ignored 属性表示整个 JSP 应用是否支持 EL。由于我们需要使用 EL，所以我们也将其设置为 false。

## 7.2　EL 应用

在 JSP 中使用表达式语言，可以大大简化 JSP 开发的工作量。下面我们就从 EL 在运算符求值、访问作用域变量以及 EL 内置对象和函数几方面的应用来看 EL 是如何实现代码简化的。

### 7.2.1　EL 运算符求值

EL 中的运算符包括算术运算符（+、−、*、/）、关系运算符（>、<）和逻辑运算符（&&、||、!），还有 empty 运算符，它是用来判断值是否为空的，EL 中的运算符如表 7.1 所示。

表 7.1　EL中的运算符

| 运 算 符 | 代 表 运 算 | 运 算 符 | 代 表 运 算 |
|---|---|---|---|
| + | 加（算术） | >、gt | 大于（比较） |
| - | 减（算术） | <、lt | 小于（比较） |
| * | 乘（算术） | <=、le | 小于等于（比较） |
| /、div | 除（算术） | >=、ge | 大于等于（比较） |
| %、mod | 取模（算术） | ==、= | 等于（比较） |
| &&、and | 与（逻辑） | !=、ne | 不等于（比较） |
| \|\|、or | 或（逻辑） | x?y:z | 条件求值 |
| !、not | 非（逻辑） | empty | 检查是否为空 |

**【示例 7.1】**我们举一个例子 Compare.jsp 来看如何运用 EL 运算符进行比较，具体代码如图 7.5 所示。

```
<%@ page contentType="text/html; charset=GB2312" %>
<html>
 <head><center>
   <title>JSP 2.0 表达式 - 运算符比较</title>
 </head>
 <body>
   <h2>JSP 2.0 表达式 - 运算符比较</h2>
     <u><b>数字比较</b></u>
     <code>
       <table border="1">
         <thead>
            <td><b>  EL 表 达 式  </b></td>
            <td><b>  比 较 结 果  </b></td>
          </thead>
    <tr>  <td>\${1 &lt; 2}</td>  <td>${1 < 2}</td>  </tr>
    <tr>  <td>\${1 &gt; 2}</td>  <td>${1 > 2}</td>  </tr>
    <tr>  <td>\${1 &gt; (4/2)}</td>  <td>${1 > (4/2)}</td>  </tr>
    <tr>  <td>\${1 &lt; (4/2)}</td>  <td>${1 < (4/2)}</td>  </tr>
    <tr>  <td>\${4.0 &gt; 3}</td>  <td>${4.0 > 3}</td>  </tr>
    <tr>  <td>\${4.0 &ge; 3}</td>  <td>${4.0 >= 3}</td>  </tr>
    <tr>  <td>\${4 &lt; 3}</td>  <td>${4 < 3}</td>  </tr>
    <tr>  <td>\${4 &le; 3}</td>  <td>${4 <= 3}</td> </tr>
    <tr>  <td>\${100.0 == 100}</td>  <td>${100.0 == 100}</td> </tr>
    <tr>  <td>\${(10*10)  != 100}</td> <td>${(10*10)  != 100}</td> </tr>
       </table>
     </code>
    <br>
 </body>
</html>
```

运算符的两种表示方法

图 7.5　Compare.jsp 示例

在浏览器地址栏中输入 http://localhost:8080/JSTL/Compare.jsp 来测试这个程序，显示结果如图 7.6 所示。

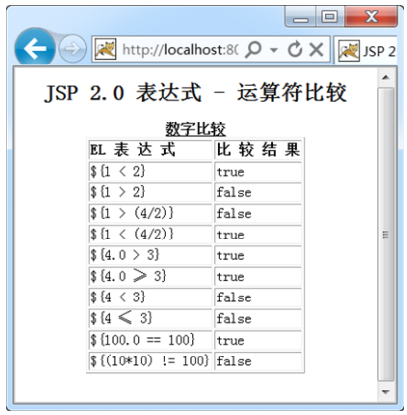



图 7.6　Compare.jsp 运行结果

## 7.2.2　访问作用域变量

可以用 EL 表达式语言按照 pageContext、HttpServletRequest、HttpSession 和 ServletContext 的顺序访问作用域，其一般格式如图 7.7 所示。

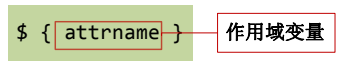



图 7.7　EL 访问作用域变量语法格式

它的作用相当于如下代码：

```
${attrname}
<%=pageContext.findAttribute(attrname)%>
<jsp:useBean id="attrname" type="Package.Class" scope=="...">
<%=attrname%>
```

**【示例 7.2】**下面我们通过一个 Servlet 例子 Scope.java 说明如何运用访问作用域变量，具体代码如图 7.8 所示。

```
package EL;
import java.io.IOException;
import javax.servlet.*;
import javax.servlet.http.*;
public class Scope extends HttpServlet {
   public void doGet(HttpServletRequest request,
   HttpServletResponse response)throws ServletException, IOException {
     //属性1的作用域：requestScope
     request.setAttribute("attr1", "泰山");
     //属性2的作用域：sessionScope
     HttpSession session = request.getSession();
     session.setAttribute("attr2", "华山");
     //属性3的作用域：sessionScope
     session.setAttribute("attr3", "衡山");
     //属性4的作用域：sessionScope
     session.setAttribute("attr4", "恒山");
     //属性5的作用域：applicationScope
     ServletContext application = getServletContext();
     application.setAttribute("attr5",new java.util.Date());
      RequestDispatcher dispatcher =request.getRequestDispatcher("Scope.jsp");
      dispatcher.forward(request, response);
         }
}
```

设置作用域变量

图 7.8　Scope.java 示例

新建一个 Scope.jsp 文件，来对 Scope.java 文件进行调用，其具体代码如图 7.9 所示。

```
<%@ page contentType="text/html; charset=GB2312" %>
<html>
   <head>
       <Title>访问作用域变量</Title>
   </head>
<body>
    <center> 访问作用域变量 </center>
<UL>
        属性1: ${attr1} <br>
        属性2: ${attr2} <br>
        属性3: ${attr3} <br>
        属性4: ${attr4} <br>
        属性5: ${attr5} <br>
</UL>
</body>
</html>
```

访问作用域变量

图 7.9　Scope.jsp 示例

这个 Servlet 的配置信息如图 7.10 所示。

```
<servlet>
    <servlet-name>Scope</servlet-name>
    <servlet-class>EL.Scope</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Scope</servlet-name>
    <url-pattern>/Scope</url-pattern>
</servlet-mapping>
```

图 7.10　Scope.java 示例的 web.xml 配置

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/Scope.jsp，显示结果如图 7.11 所示。


图 7.11　Scope.jsp 示例运行结果

## 7.2.3　EL 内置对象

我们在第 3 章曾经介绍过 JSP 的 9 个内置对象，在 EL 中也有自己的内置对象。它们共有 11 个，按功能可以大致分为 3 类。这 11 个内置对象的名称和具体说明如表 7.2 所示。

表 7.2　EL中的内置对象

| | | | |
|---|---|---|---|
| 第一类：pageContext 对象。可以用来访问 JSP 其他 8 个内置对象 | pageContext | javax.servlet.ServletContext | 可以用于访问 JSP 的隐含对象 |

续表

| | | | |
|---|---|---|---|
| 第二类：用于访问环境信息的对象 | cookie | java.util.Map | 映射 cookie 名到单个 cookie 对象 |
| | initParam | java.util.Map | 映射上下文初始化参数名称到单个值 |
| | header | java.util.Map | 映射请求头名称到单个字符串数值 |
| | param | java.util.Map | 映射请求参数名到单个字符串参数值 |
| | headerValues | java.util.Map | 映射请求头名称到字符串数组 |
| | paramValues | java.util.Map | 映射请求参数名到字符串数组 |
| 第三类：用于访问作用域范围的内置对象 | applicationScope | java.util.Map | 映射应用程序范围的变量名到其值 |
| | sessionScope | java.util.Map | 映射会话范围的变量名到其值 |
| | requestScope | java.util.Map | 映射请求范围的变量名到其值 |
| | pageScope | java.util.Map | 映射页面范围的变量名到其值 |

**【示例 7.3】**我们来举一个简单的例子 Implicit-object.jsp 来看这些内置对象的用法，我们随机调用其中的几个对象来实现对其的访问，具体代码如图 7.12 所示。

```
<%@ page contentType="text/html; charset=GB2312" %>
<html>
  <head>
    <title>访问隐含对象</title>
  </head>
  <body>
    页面请求参数: ${param.argc} <br>
    上下文路径:  ${pageContext.request.contextPath}  <br>
    User-Agent Header: ${header["User-Agent"]} <br>
    Cookie JSESSIONID Value: ${cookie.JSESSIONID.value} <br>
    Web服务器信息: ${pageContext.servletContext.serverInfo} <br>
  </body>
</html>
```

EL内置对象的用法

图 7.12　Implicit-object.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/Implicit-object.jsp?argc=Hello, World!，来测试这个示例，显示结果如图 7.13 所示。


图 7.13　Implicit-object.jsp 示例运行结果

### 7.2.4　EL 函数

表达式语言允许用户自定义函数。此函数必须采用 public 类中的 public static 方法编写并映射到 TLD 标签库文件中。EL 函数的定义语法格式如图 7.14 所示。

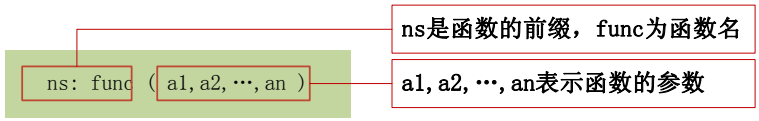



图 7.14　EL 函数的定义语法格式

**【示例 7.4】**下面我们通过一个例子 ELFunction.java 简单地说明 EL 函数的用法。该函数的目的是比较 2 个整数值并获取其中的较大值，具体代码如图 7.15 所示。

```
package EL;
public class ELFunction {
    public static int max(int a1,int a2){
       if(a1>=a2)
              return a1;
       else
              return a2;
       }
}
```

图 7.15　ELFunction.java 示例

然后在标签库描述文件 Web-INF/tlds/ELFunction.tld 中对函数进行声明，具体内容如图 7.16 所示。

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE taglib
PUBLIC "-//SUN Microsystems, Inc.//DTD JSP Tag Library 2.0//EN"
"http://java.sun.com/j2ee/dtds/web-jsptaglibrary_2_0.dtd">

<taglib>
 <tlib-version>1.0</tlib-version>
 <uri>/WEB-INF/ELFunction</uri>

 <function>
    <name>max</name>
    <function-class>EL.ELFunction</function-class>
    <function-signature>int max(int,int)</function-signature>
 </function>

</taglib>
```

函数配置说明

图 7.16　ELFunction.tld 文件内容

接下来我们可以编写一个测试页面 ELFunction.jsp，具体代码如图 7.17 所示。

最后我们对 web.xml 文件进行配置，以使系统可以根据 uri 的值，到 web.xml 文件中找到相对应的 TLD 文件，其配置方法如图 7.18 所示。

```
<%@ page contentType="text/html; charset=GB2312" %>
<%@ page taglib prefix="myfn" uri="/ELFunction" %>
<html>
  <head>
    <title>函数示范</title>
  </head>
  <body>
     <h2>EL函数示范</h2>
     比较25和35，其中较大的值是${myfn:max(25,35)}
  </body>
</html>
```

图 7.17　ELFunction.jsp 示例

```
<jsp-comfig>
 <taglib>
   <taglib-uri>/ELFunction</taglib-uri
   <taglib-location>/web-INF/tlds/ELFunction.tld</taglb-location>
 </taglib>
</jsp-config>
```

图 7.18　ELFunction.java 的 web.xml 配置

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/ELFunction.jsp 来测试这个示例，显示结果如图 7.19 所示。


图 7.19　ELFunction.jsp 运行结果

## 7.3　JSTL 简介

JSTL 的全称是 JavaServer Pages Standard Tag Library。它是一个 JSP 标签的集合，用于简化 JSP 程序的开发，并使 JSP 程序更易于维护和管理。JSTL 的具体目标是简化 JSP 页面的设计。对于页面设计人员来说，实用程序语言来操作动态数据是比较困难的，而采用标签和表达式语言则相对容易一些。可以说，JSTL 的使用为页面设计人员和程序开发人员的分工协作提供了便利。

JSTL 实际上是由 5 个不同的标签库组成的，这样也减少了不同类动作之间的名字冲突。在 JSP 规范中，为这 5 个标签库默认指定了默认 URI 和推荐前缀，如表 7.3 所示。

表 7.3　JSTL常用标签库

| 标 签 名 | URI | 前　缀 | 范　例 |
|---|---|---|---|
| Core（核心） | http://java.sun.com/jsp/jstl/core | c | <c:tagname> |
| XML | http://java.sun.com/jsp/jstl/xml | x | <x:tagname> |
| Internationalization（国际化） | http://java.sun.com/jsp/jstl/fmt | fmt | <fmt:tagname> |
| Functions（函数） | http://java.sun.com/jsp/jstl/functions | fn | <fn:tagname> |
| SQL（关系型数据库） | http://java.sun.com/jsp/jstl/sql | sql | <sql:tagname> |

JSTL 主要是由 Apache 组织的 Jakarta 小组负责实现的，读者可以从 Apache 网站上下载 JSTL 安装包，下载地址为 http://tomcat.apache.org/taglibs/standard/，文件名为 jakarta-taglibs-standard-1.1.2.zip，具体的下载过程如图 7.20 所示。

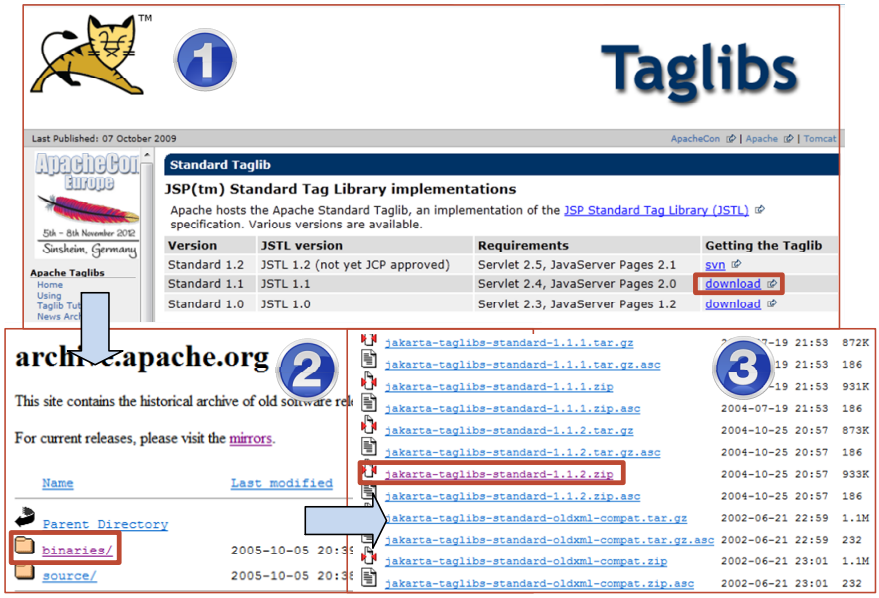



图 7.20　JSTL 文件的下载过程

下载完后解压缩，可以发现其中的内容如图 7.21 所示。

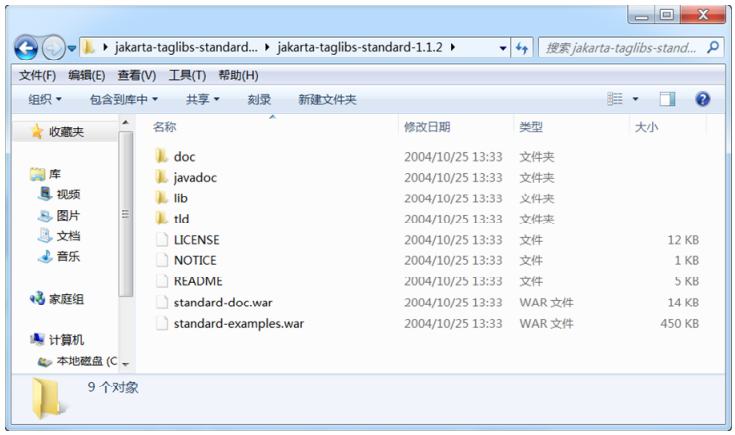



图 7.21　JSTL 包的内容

在 lib 目录下有两个 JAR 包：jstl.jar 和 standard.jar，这两个文件即为所需要的 JSTL 标签库。将上述两个文件复制到网站根目录下的/WEB-INF/lib/目录下，例如本书中该目录的路径为 C:\Users\Administrator\Workspaces\MyEclipse 10\JSTL\WebRoot\WEB-INF\lib。重新启动 Tomcat，测试 JSTL 配置是否成功。

在解压出的文件夹中，有一个 standard-examples.war 文件，这是 Jakarta 所提供的 JSTL 的范例程序。我们将其复制到 Tomcat 的 webapps 目录下，然后重新启动 Tomcat，在浏览器中输入网址 http://localhost:8080/standard-examples/，如果看到如图 7.22 所示的界面，说明 JSTL 配置成功。

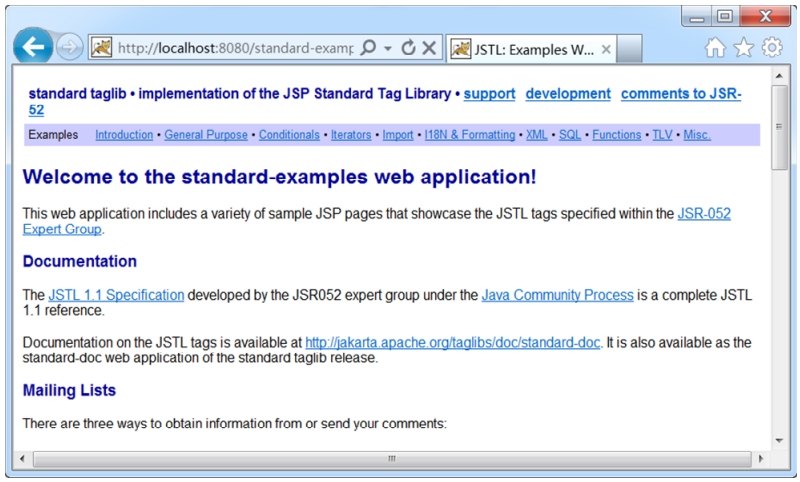



图 7.22　JSTL 范例应用首页

下面我们就来详细介绍 JSTL 标签库。

## 7.4　Core 标签库（核心标签库）

JSTL 核心标签库中包含很多标签，根据其功能大致可以分为四类，如图 7.23 所示。

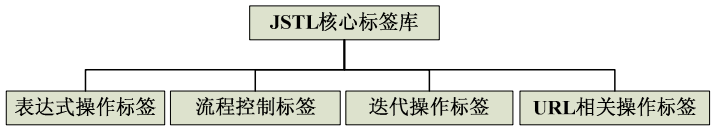



图 7.23　JSTL 核心标签库

如果要在 JSP 页面中使用核心库的标签，需要用 taglib 指令指明这个标签库的路径为如图 7.24 所示的形式。

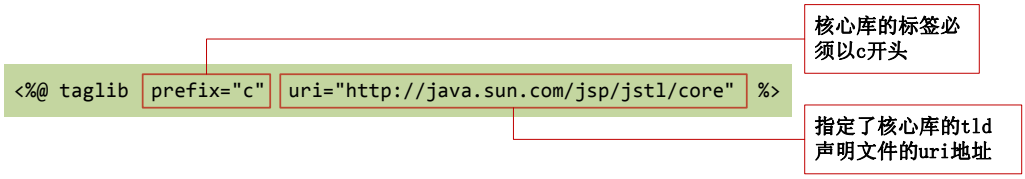



图 7.24　使用核心库标签

### 7.4.1　表达式操作标签

表达式操作标签中包含 4 个标签，分别是<c:out>、<c:set>、<c:remove>和<c:catch>。

#### 1．<c:out>标签

核心标签库中最为基本的标签就是<c:out>。它可以在页面中显示一个字符串或者一个 EL 表达式的值。它的功能与 JSP 传统的<%=表达式%>相类似。<c:out>标签的使用格式如图 7.25 所示。

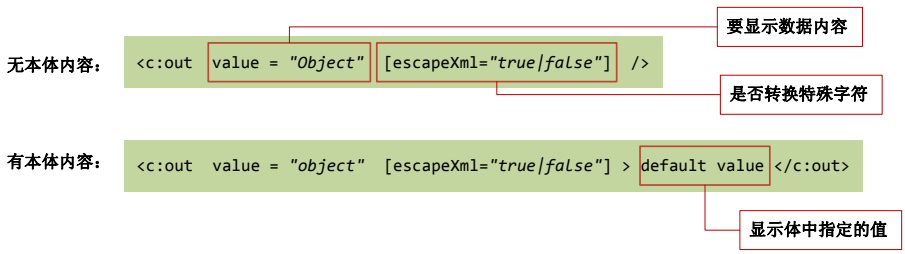



图 7.25　<c:out>标签的使用格式

<c:out>标签的各个属性的具体描述如表 7.4 所示。

表 7.4　<c:out>标签属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| value | Object | 输出信息，可为常量值或表达式 |
| escapeXml | Boolean | 设置< > & ‘ “这些特殊字符在输出字符串中是否要转换成对应的代码，默认取值为 true |
| default | Object | 当 value 属性为空(null)时，要显示的值 |

**【示例 7.5】**下面通过一个<c:out>的简单实例，来示范以上各属性设置的效果，创建的 c_out.jsp 页面代码如图 7.26 所示。

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_out.jsp 来测试这个示例，显示结果如图 7.27 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
  <h4><c:out value="核心标签库中的out标签的简单实例"/></h4>
  <c:out value="${3+7}"/><br>
  <c:out value="<p>有特殊字符</p>" /><br>
  <c:out value="<p>有特殊字符</p>" escapeXml="false" />
```

是否进行特殊字符的转换


图 7.26　c_out.jsp 示例

图 7.27　c_out.jsp 示例运行结果

#### 2．<c:set>标签

<c:set>标签是用来在某个范围（request、session 或者 application）内设值，或者设置某个对象的属性值的标签。<c:set>标签的使用格式如图 7.28 所示。

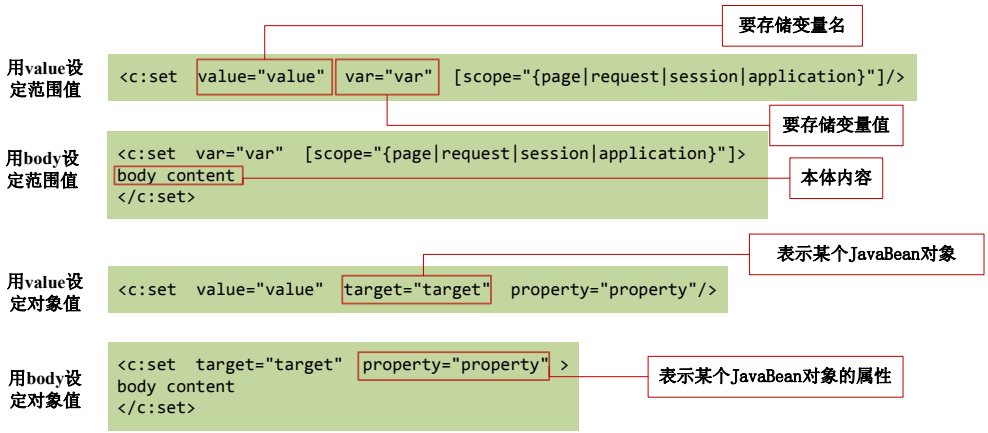



图 7.28　<c:set>标签的使用格式

<c:set>标签的各个属性的具体描述如表 7.5 所示。

表 7.5　<c:set>标签属性描述

| 名　　称 | 类　　型 | 属 性 描 述 |
|---|---|---|
| value | Object | 需设定的范围变量或对象属性的值 |
| var | String | 要设定的范围变量名 |
| scope | String | var 指定变量的范围，默认值为 page |
| target | Object | 需设定属性的对象 |
| property | String | 需设定值的 target 对象的属性名称 |

**【示例 7.6】**接下来给出一个<c:set>标签的应用实例，创建的 c_set.jsp 页面代码如图 7.29 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
 <title>JSTL: -- Set标签实例</title>
</head>
<body>
 <h3>set标签实例</h3>
<hr>
 <c:set var="num1" scope="request" value="${3+5}"/>
 <c:set var="num2" scope="session" >
  ${3+6}
 </c:set>
 <c:set var="num3" scope="session" >
  10
 </c:set><br>
   num1变量值为: <c:out value="${num1}" /><br>
   num2变量值为: <c:out value="${num2}" /><br>
   num3变量值为: <c:out value="${num3}" />
</body>
</html>
```

通过set标签赋值

通过out标签输出变量值

图 7.29　c_set.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_set.jsp 来测试这个示例，显示结果如图 7.30 所示。


图 7.30　c_set.jsp 示例运行结果

### 3．<c:remove>标签

<c:remove>标签一般和<c:set>标签配套使用，两者是相对应的。<c:remove>标签用于删除某个变量或者属性。<c:remove>标签的使用格式如图 7.31 所示。

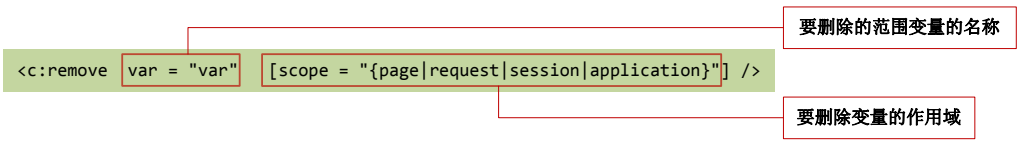



图 7.31　<c:remove>标签的使用格式

**【示例 7.7】**接下来还是通过一个实例来了解<c:remove>标签的使用，创建的 c_remove.jsp 源代码如图 7.32 所示。

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_remove.jsp 来测试这个示例，显示结果如图 7.33 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head><title> JSTL: -- remove标签的使用</title></head>
<body>
<h3>remove标签实例</h3> <hr>
<c:set  value="Lily"  var="name"  scope="page"/>
<c:set  value="Lucy"  var="name"  scope="session"/>

<c:remove  var="name"  scope="page"/>
<c:out  value="${name}"/>

<c:remove  var="name"  scope="session"/>
<c:out  value="${name}"/>
</body>
</html>
```

删除作用域为page的范围变量name，JSP页面输出结果Lucy

删除作用域为session的范围变量name，输出结果为空

图 7.32　c_remove.jsp 示例

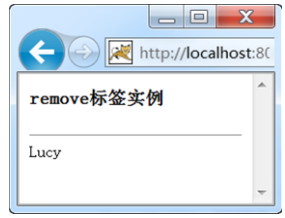



图 7.33　c_remove.jsp 示例运行结果

### 4．<c:catch>标签

<c:catch>标签的功能和 Java 程序中 try{}catch{}语句功能很类似，它用于捕获嵌入到它中

间语句抛出的异常。<c:catch>标签的使用格式如图 7.34 所示。

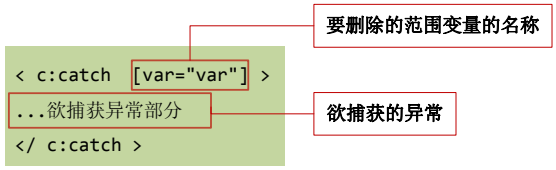



图 7.34　<c:catch>标签的使用格式

**【示例 7.8】**下面还是通过一个实例来具体了解<c:catch>标签的使用，创建的 c_catch.jsp 页面代码如图 7.35 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
 <title>JSTL: -- catch标签实例</title>
</head>
<body>
<h4>catch标签实例</h4>
<hr>
<c:catch var="errors">
<%
    String num = request.getParameter("num");
    int i = Integer.parseInt(num);
    out.println("the number is "+i);
%>
</c:catch>
<c:out value="${errors}"/>
</body>
</html>
```

接收整数值并输出

输出异常

图 7.35　c_catch.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_catch.jsp?num=5 来测试这个示例，显示结果如图 7.36 所示。

如果我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_catch.jsp?num=abc，就会发生异常，显示结果如图 7.37 所示。


图 7.36　c_catch.jsp 示例正确显示结果

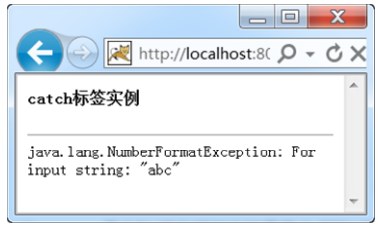



图 7.37　发生异常的 c_catch.jsp 界面

## 7.4.2　流程控制标签

流程控制分类标签中也包含 4 个标签，分别是<c:if>、<c:choose>、<c:when>和<c:otherwise>。

### 1.<c:if>标签

<c:if>标签的作用和 Java 程序中的 if 语句作用相同，用于判断条件语句。<c:if>标签使用的格式如图 7.38 所示。

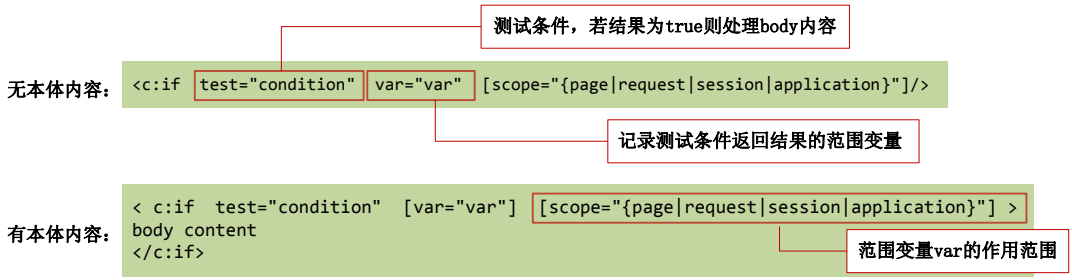



图 7.38　<c:if>标签使用的格式

**【示例 7.9】**下面我们通过一个实例来具体了解<c:if>标签的使用，创建的 c_if.jsp 页面代码如图 7.39 所示。

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_if.jsp 来测试这个示例，显示结果如图 7.40 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
 <title>JSTL: -- if标签实例</title>
</head>
<body>
<h4>if标签实例</h4>
<hr>
<c:set var="num" scope="page" value="admin" />
<c:if test="${num == 'admin'}" var="condition" scope="page">
        你好 admin!
</c:if>
<c:out value="这里判断结果: ${condition}"/>
<br>
<c:if test="${num == 'guest'}" var="condition" scope="page">
        你好 admin!
</c:if>
<c:out value="这里判断结果: ${condition}"/>
</body>
</html>
```

设定value值为admin

当num值相等时的情况

当num值不相等时的情况

图 7.39　c_if.jsp 示例


图 7.40　c_if.jsp 示例运行结果

### 2．<c:choose>、<c:when>和<c:otherwise>标签

<c:choose>、<c:when>和<c:otherwise>标签一般是组合起来一起使用的，就相当于 Java 程序中的 switch 条件语句。在<c:choose>标签体中包括<c:when>和<c:otherwise>子标签。<c:when>子标签代表<c:choose>的一个条件分支。这三个标签的使用格式如图 7.41 所示。

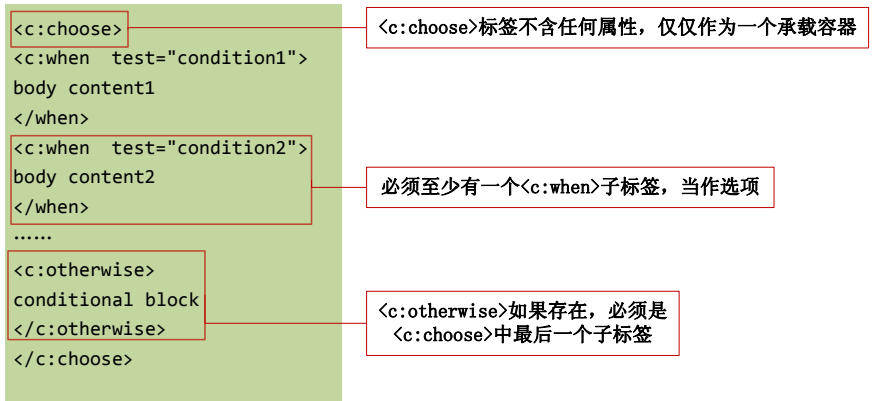



图 7.41　<c:choose>、<c:when>和<c:otherwise>标签的使用格式

**【示例 7.10】**下面我们通过一个综合实例来了解这些标签的具体使用方法，创建一个 c_choose.jsp 文件代码如图 7.42 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
 <title>JSTL: -- choose、when、otherwise标签实例</title>
</head>
<body>
<h4>choose、when、otherwise标签实例</h4>
<hr>
<c:set var="num" scope="page" value="admin" />
<c:choose>
 <c:when test="${num == 'guest'}">
                  <c:out value="guest" />
 </c:when>
 <c:when test="${num == 'admin'}">
      <c:out value="admin" />
 </c:when>
 <c:otherwise>
      <c:out value="other" />
 </c:otherwise>
</c:choose>
</body>
</html>
```

设定value值为admin

当num值为guest的情况

当num值为admin的情况

当num值为其他时的情况

图 7.42　c_choose.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_choose.jsp，显示结果如图 7.43 所示。

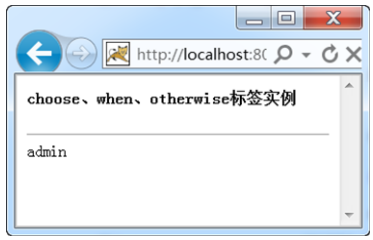



图 7.43　c_choose.jsp 示例运行结果

## 7.4.3　迭代操作标签

迭代分类标签中包含 2 个标签，分别是<c:forEach>标签和<c:forTokens>标签。使用迭代标签可以遍历集合、数组和字符串等，还可以用来指定语句的执行次数。

### 1．<c:forEach>标签

<c:forEach>标签允许用户对一个对象集合执行相同的操作，相对于 Iterator 迭代器而言，JSTL 在很大程度上简化了迭代操作的使用。<c:forEach>标签与 Java 语言中的 for 循环有异曲同工之妙，可以用于枚举一个集合对象中的元素，或者用来循环指定的次数，<c:forEach>标签使用格式如图 7.44 所示。

**遍历集合对象**

```
<c:forEach [var="varName"]  items="collection"
        [varStatus="varStatusName"][begin="begin"] [end="end"] [step="step"]>
   循环体中要执行的内容
</c:forEach>
```

**执行特定次数的迭代**

```
<c:forEach [var="varName"]   items="collection"
          [varStatus="varStatusName"] begin="begin"end="end" [step="step"]>
循环体中要执行的内容
</c:forEach>
```

图 7.44　<c:forEach>标签使用格式

图中<c:forEach>标签属性的相关描述信息如表 7.6 所示。

表 7.6　<c:forEach>标签的属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| var | String | 表示当前处理对象的变量 |
| items | 支持多种类型，主要包括：Array、Collection、Iterator、Enumeration、Map、string 类型 | 进行循环的项目 |
| varStatus | String | 记录循环状态的变量 |
| begin | int | 循环的起始点 |
| end | int | 循环的结束位置，end 的取值不能小于 begin |
| step | int | 循环步长 |

**【示例 7.11】**下面我们通过一个实例来了解<c:forEach>标签的具体使用方法，创建一个 c_forEach.jsp 文件代码如图 7.45 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<body>
<h4><c:out value="forEach实例"/></h4>
<hr>
<%
String items[] = new String[5];
items[0] = "核心标签库";items[1] = "国际化标签库";
items[2] = "SQL标签库";items[3] = "XML标签库";
items[4] = "函数标签库";
request.setAttribute("items",items);
%>
<B><c:out value="不指定begin和end的迭代：" /></B><br>
<c:forEach var="item" items="${items}">
  <c:out value="${item}"/><br>
</c:forEach>
<B><c:out value="指定begin和end的迭代：" /></B><br>
<c:forEach var="item" items="${items}" begin="1" end="3" step="1">
         <c:out value="${item}" /><br>
</c:forEach>
<B><c:out value="输出整个迭代的信息：" /></B><br>
<c:forEach var="item" items="${items}" begin="3" end="4" step="1"
varStatus="s">
         <c:out value="${item}" />的四种属性：<br>
          所在位置，即索引：<c:out value="${s.index}" /><br>
          总共已迭代的次数：<c:out value="${s.count}" /><br>
          是否为第一个位置：<c:out value="${s.first}" /><br>
          是否为最后一个位置：<c:out value="${s.last}" /><br>
</c:forEach>
</body>
</html>
```

设定数组值

对数组值进行遍历

指定循环次数

图 7.45　c_forEach.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_forEach.jsp，显示结果如图 7.46 所示。


图 7.46　c_forEach.jsp 示例运行结果

### 2．<c:forTokens>标签

<c:forTokens>标签是 JSTL 种的另一个迭代循环标签，它可以用来对一个字符串进行迭代循环，这个字符串是用符号分开的，<c:forTokens>标签的使用语法如图 7.47 所示。

```
<c:forTokens  items="stringtokens"  delims="delims"  [var="var"]
                [varStatus="varstatus"] [begin="begin"]  [end=" end"]
[step="step"]>

body content

</c:forTokens>
```

分隔字符集合

图 7.47　<c:forTokens>标签使用语法

<c:forTokens>标签属性的相关描述信息如表 7.7 所示。

表 7.7　<c:forTokens>标签的属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| items | String | 要处理的一系列以特定符号隔开的字符串 |
| delims | String | 分隔字符集合 |
| var | Sring | 记录当前处理项目的变量 |
| varStatus | String | 记录循环状态的变量 |
| begin | int | 循环的起始点 |
| end | int | 循环的结束位置 |
| step | int | 循环步长 |

**【示例 7.12】**下面我们通过一个实例来了解<c:forTokens>标签的具体使用方法，创建一个 c_forTokens.jsp 文件代码如图 7.48 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
 <title>JSTL: -- forTokens标签实例</title>
</head>
<body>
<h4><c:out value="forTokens实例"/></h4>
<hr>
<h4>使用 '|' 作为分隔符</h4>
<c:forTokens var="token" items="red,blue,yellow|red,blue|yellow,green"
delims="|">
   <c:out value="${token}"/>
</c:forTokens>
<h4>使用 '|' 和 ',' 一起做分隔符</h4>
<c:forTokens var="token" items="red,blue,yellow|red,blue|yellow,green"
delims="|,">
   <c:out value="${token}"/>
</c:forTokens>
</body>
</html>
```

图 7.48　c_forTokens.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_forTokens.jsp，显示结果如图 7.49 所示。

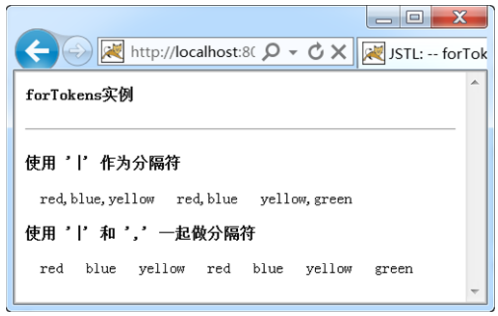



图 7.49　c_forTokens.jsp 示例运行结果

## 7.4.4　URL 相关的标签

核心标签库中还有一些是用来进行与 URL 相关的操作，主要包括 4 个标签，分别是<c:import>、<c:redirect>、<c:url>和<c:param>标签。

### 1．<c:import>标签

<c:import>标签用来引入某个 URL 指向的网页内容到当前 JSP 网页中。它和传统的 JSP 标记<jsp:include>相类似，但有所不同是<jsp:include>标签只能用来包含该应用中的其他文件，而<c:import>则还可以包含外部站点中的静态文件，所以它的功能更加强大。<c:import>标签的使用格式如图 7.50 所示。

网页内容以String对象的形式输出

```
<c:import  url="url"  [context="context"][var="var"]
   [scope="{page|request|session|application}"][charEncoding="charEncoding"]>
...
</c:import>
```

网页内容以Reader对象的形式输出

```
<c:import  url="url"  [context="context"]  varReader="varreader"
     [charEncoding="charencoding"]>
...
</c:import>
```

图 7.50　<c:import>标签的使用格式

<c:import>标签属性的相关描述信息如表 7.8 所示。

表 7.8　<c:import>标签的属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| url | String | 需要导入的网页 URL |
| context | String | url 指定为其他网站的某个 URL 时，本地网站的路径 |
| var | String | 存储导入文件内容的变量 |
| scope | String | 变量 var 的作用域 |
| charEncoding | String | 导入文件的字符集 |
| varReader | String | 用来读取导入文件内容的 Reader 对象 |

【示例 7.13】下面我们通过一个实例来了解<c:import>标签的具体使用方法，创建一个 import_URL.jsp 文件代码如图 7.51 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
  <head><title>在当前JSP网页中导入其他网页</title></head>
 <body>
   <c:catch var="exception">
   <c:import url="welcome.jsp"/>
   </c:catch>
   <c:if test="${not empty exception}">
    ${exception.message}
   </c:if>
 </body>
</html>
```

引入另一个JSP内容

图 7.51　import_URL.jsp 示例

被引入的 welcome.jsp 文件的具体代码如图 7.52 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<html>
  <body>
  <center>
       这是被引入的另一个JSP文件中的内容。
  </center>
  </body>
</html>
```

图 7.52　welcome.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/import_URL.jsp 来测试这个示例，显示结果如图 7.53 所示。


图 7.53　import_URL.jsp 示例运行结果

### 2．<c:redirect>和<c:param>标签

<c:redirect>标签可以把用户的请求从一个页面转向另一个页面，同 JSP 中 response 内置对象的跳转功能类似。<c:param>标签是和<c:redirect>标签一起使用的，它用来进行参数值的传递。<c:redirect>标签的使用格式如图 7.54 所示。

【示例 7.14】下面我们通过一个实例来了解<c:redirect>标签的具体使用方法，创建一个

c_redirect.jsp 文件代码如图 7.55 所示。

无参数传递

```
<c:redirect  url="url"  [context="context"]/>
```

需要重定向到的网页URL位置

url指定为其他网站的某个URL时，本地网站的路径

有参数传递

```
<c:redirect  url="url"  [context="context"]>
<c:param  name="name"  value="value"/>
</c:redirect>
```

图 7.54　<c:redirect>标签的使用格式

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<html>
<head>
 <title>JSTL: -- redirect标签实例</title>
</head>
<body>
<c:redirect url="c_redirect2.jsp">
<c:param name="userName" value="admin" />
</c:redirect>
</body>
</html>
```

重定向到c_redirect2.jsp页面

图 7.55　c_redirect.jsp 示例

重定向的 c_redirect2.jsp 页面的具体代码如图 7.56 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<h4>重定向的页面</h4>
<hr>
userName=<c:out value="${param.userName}" />
```

获取传递值

图 7.56　c_redirect2.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/c_redirect.jsp 来测试这个示例，显示结果如图 7.57 所示。


图 7.57　c_redirect.jsp 示例运行结果

### 3．<c:url>标签

<c:url>标签主要是用来重写 URL 地址。<c:url>标签也有两种使用语法，如图 7.58 所示。

无参数传递

```
<c:url value="value" [context="context"] [var="varName"]
                [scope="{page|request|session|application}"] />
```

存储构造后URL的范围变量

需要构造的URL

有参数传递

```
<c:redirect  url="url"  [context="context"]>
<c:param  name="name"  value="value"/>
</c:redirect>
```

图 7.58　<c:url>标签的使用格式

【示例 7.15】我们举一个例子 c_url.jsp 来看如何运用<c:url>标签，具体代码如图 7.59 所示。

```
<%@ page contentType="text/html;charset=GBK" %>
<%@ page isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<c:url var="url" value="c_url2.jsp" scope="session">
 <c:param name="userName" value="admin"/>
 <c:param name="password" value="123456"/>
</c:url>
<c:out value="${url}" />
<a href='<c:out value="${url}"/>'>Link to other page </a>
```

重写URL地址

图 7.59　c_url.jsp 示例

在浏览器地址栏中输入 http://localhost:8080/JSTL/c_url.jsp，显示结果如图 7.60 所示。

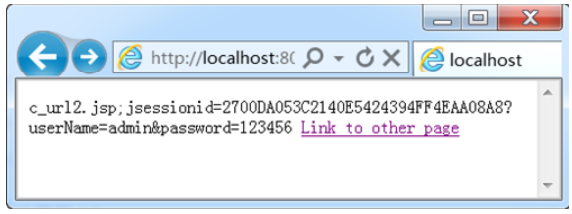



图 7.60　c_url.jsp 示例运行结果

## 7.5　XML 操作标签库

在 Java 中可以使用 SAX 或者 DOM 等 API 接口来操作 XML 文档，尽管这种操作功能非常强大而且灵活，但是要很快熟练掌握是有相当大难度的，在 JSTL 中提供了一组专门处理 XML 文档的标签，这些标签所提供的功能尽管非常有限，但是已经可以满足基本的 XML 文档处理需要，而且这些标签学习起来明显比掌握复杂的 API 接口要容易。接下来的章节中将介绍 JSTL 中用来处理 XML 文档的标签。

JSTL XML 为操作标签库中的标签，根据其功能大致可以分为三类，如图 7.61 所示。


图 7.61　JSTL XML 操作标签库

如果要在 JSP 页面中使用 XML 操作标签库的标签，需要用 taglib 指令指明这个标签库的路径为如图 7.62 所示的形式。


图 7.62　使用 XML 操作标签库

## 7.5.1　核心操作标签

XML 核心操作分类标签包含<x:out>、<x:set>和<x:parse>标签。其功能与 JSTL 中核心标签库中对应的功能基本相同。

### 1．<x:parse>和<x:out>标签

<x:parse>标签可以用来解析一个 XML 文档。<x:parse>标签有两种使用语法，如图 7.63 所示。

以String或Reader对象指定XML

```
<x:parse  {doc="xmldocument"|xml="xmldocument"}
{var="var"  [scope="scope"]|varDom="var"  [scopeDom="scopedom"]}
[systemId="systemid"]
[filter="filter"]/>
```

以body内容指定XML

```
<x:parse {var="var"  [scope="scope"]|varDom="var" [scopeDom="scope"]}
[systemId="systemid"][filter="filter"]>
XML document to parse
</x:parse>
```

图 7.63　<x:parse>标签使用语法

<x:parse>标签的属性描述如表 7.9 所示。

表 7.9　<x:parse>标签的属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| doc | String or Reader | 需要解析的 XML 文件 |
| xml | String or Reader | 同 doc 属性，现已不再使用 |
| var | String | 用于存储解析后 XML 文件的范围变量名 |
| scope | String | 范围变量 var 的作用域 |
| varDom | String | 用于存储解析后 XML 文件的范围变量名 |
| scopeDom | String | 范围变量 varDom 的作用域 |
| systemId | String | 用于解析 XML 文件的路径 |
| filter | org.xml.sax.XMLFilter | 解析 XML 文件时使用在 XML 文件的过滤器 |

使用<x:parse>把一个 XML 文档解析以后，就可以使用<x:out>标签输出 XML 节点的值。

<x:out>标签用于计算 XPath 表达式，然后把查找到的元素进行输出。XPath 标准是用来对 XML 文档各部分数据进行定位的语言。<x:out>标签的使用格式如图 7.64 所示。

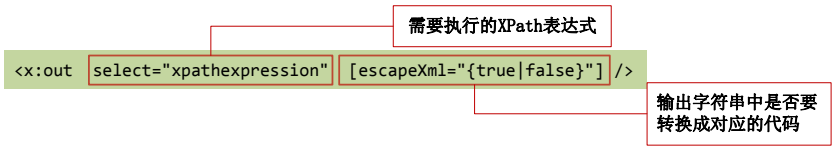



图 7.64　<x:out>标签的使用格式

**【示例 7.16】**我们举一个例子 x_parse.jsp 来看如何运用<x:parse>和<x:out>标签，具体代码如图 7.65 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x"%>
<html>
 <head>
   <title><c:out value="标签示例"/></title>
 </head>
 <body>
 <font size="2">
    <c:out value="标签示例"/><br>
    <c:import var="doc" url="Students.xml" charEncoding="gb2312"/>
    <x:parse var="studentDoc" doc="${doc}"/>
    <c:out value="学生姓名："/>
    <x:out select="$studentDoc/students/student/name"/><br>
    <c:out value="学生年龄："/>
    <x:out select="$studentDoc/students/student/age"/><br>
    <c:out value="学生性别："/>
    <x:out select="$studentDoc/students/student/sex"/><br>
  </font>
 </body>
</html>
```

需要解析的XML文件

输出XML文档各部分数据

图 7.65　x_parse.jsp 应用示例

被调用的 XML 文件 Students.xml 的具体代码如图 7.66 所示。

```
<?xml version="1.0" encoding="gb2312"?>
<students>
  <student>
    <name>Lester</name>
    <age>23</age>
    <sex>男</sex>
  </student>
</students>
```

Student各部分数据

图 7.66　Students.xml 示例

在浏览器地址栏中输入 http://localhost:8080/JSTL/x_parse.jsp，显示结果如图 7.67 所示。

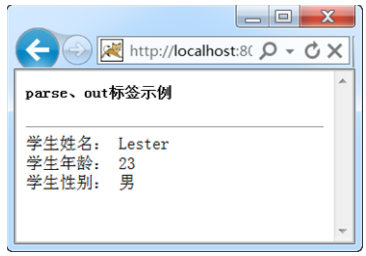



图 7.67　x_parse.jsp 示例运行结果

### 2．<x:set>标签

<x:set>标签用来把 XPath 表达式处理的结果存储到某个范围变量中。与<c:set>标签不同之处在于<x:set>标签是把 XML 中某个节点的内容赋值到一个变量中。<x:set>标签的使用语法如图 7.68 所示。

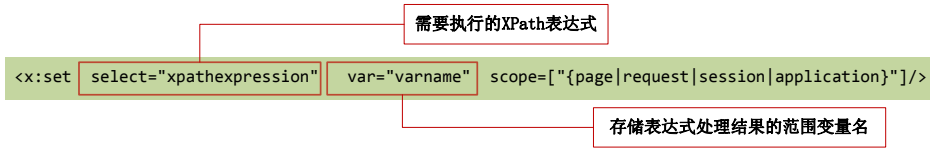



图 7.68　<x:set>标签使用语法

**【示例 7.17】**我们举一个例子 x_set.jsp 来看如何运用<x:set>标签，具体代码如图 7.69 所示。

```
<%@ page  contentType="text/html;charset=gb2312" %>
<%@ taglib  prefix="c"  uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib  prefix="x"  uri="http://java.sun.com/jsp/jstl/xml" %>
<html>
<head><title>xml文件处理结果存储标签使用例程</title></head>
<body>
 <center>
<c:import var="doc" url="Students.xml" charEncoding="gb2312"/>
<x:parse var="studentDoc" doc="${doc}"/>
<x:set select="$studentDoc/students/student/name"var="name1"/><br>
<x:out select="$name1/name"/><br>
</center>
</body>
</html>
```

将name值存到name1中

图 7.69　x_set.jsp 示例

在浏览器地址栏中输入 http://localhost:8080/JSTL/x_set.jsp，显示结果如图 7.70 所示。

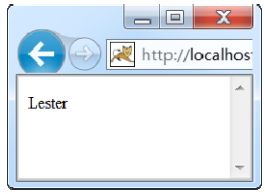



图 7.70　x_set.jsp 运行结果

### 7.5.2　流程控制标签

XML 流程控制标签包含<x:if>、<x:choose>、<x:when>、<x:otherwise>及<x:forEach>标签。这几个标签的功能与用法和核心标签库对应的标签非常类似，所以我们在此就不再赘述了，读者可以对照核心标签库的内容来体会流程控制标签的用法。

### 7.5.3　转换操作标签

XML 转换操作标签包含<x:transform>标签和<x:param>标签。<x:param>标签的作用和用法也与<c:param>标签类似，我们不再赘述，下面我们来看<x:transform>标签。

在网络应用程序中，经常需要通过 XSL 样式表对一篇 XML 文件执行转换，本节所要介绍的<x:tranform>标签就是为了实现这一功能设置的，<x:transform>标签的使用语法如图 7.71 所示。

```
<x:transform xml="expression" xslt="expression"  var="name"
   scope="scope"  xmlSystemId="expression"
xsltSystemId="expression">
<x:param name="expression" value="expression"/>
```

图 7.71　<x:transform>标签使用语法

<x:transform>标签的属性描述如表 7.10 所示。

表 7.10　<x:transform>标签的属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| doc | 包含 XML 文件、接收 XML 文件的 Reader 等 | 需要执行转换的 XML 文件 |
| xslt | String、Reader、javax.xml.transform.Source 实例 | 用来执行转换的 XSL 样式表 |
| docSystemId | String | 用于解析 doc 属性所设定 XML 文件的路径 |
| xsltSystemId | String | 用于解析 xslt 属性规定的 xsltstylesheet 的路径 |
| var | String | 用来存储转换后 XML 文件的范围变量名 |
| scope | String | 范围变量 var 的作用域 |

## 7.6　JSTL 格式化标签库

格式化标签库又被称为 I18N 格式标签库，I18N 是 Internationalization（国际化）的缩写。在不同的国家和地区，对数字和货币等的表示是有所不同的。JSTL 提供了格式化标签库来根据不同的语言请求做出不同的响应。

格式化标签库中的标签，根据其功能大致可以分为三类，如图 7.72 所示。

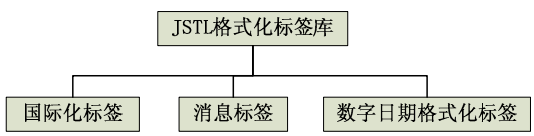



图 7.72　格式化标签库

如果要在 JSP 页面中使用格式化的标签，需要用 taglib 指令指明这个标签库的路径为如图 7.73 所示的形式。


图 7.73　使用格式化（国际化）标签库

## 7.6.1　国际化标签

国际化分类标签包含 2 个标签，一个是设定语言地区的<fmt:setLocale>标签，另一个是设定请求的字符编码集合的<fmt:requestEncoding>标签。

### 1．<fmt:setLocale>标签

<fmt:setLocale>标签的功能是用来设定用户的语言地区代码，即设置格式化或解析日期/时间、数值时所使用的语言环境，<fmt:setLocale>标签的使用格式如图 7.74 所示。

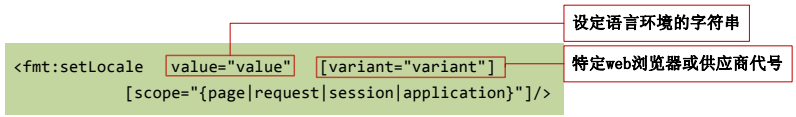



图 7.74　<fmt:setLocale>标签使用格式

**注意：** value 属性表示语言地区代码，至于具体的代码是多少，读者可以从 http://www.w3.org/查询语言代码，在 http://unicode.org 找到国家和地区代码。例如 en 表示英文、en_US 表示英文（美国）、zh_CN 表示中文（中国）。

**【示例 7.18】**我们举一个例子 fmt_setLocale.jsp 来看如何运用<fmt:setLocale>标签，具体代码如图 7.75 所示。

```
<%@ page contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
<head><title>语言环境设置例程</title></head>
<body>
<font size="4">
<center>
中国汉语语言环境下：
<fmt:setLocale value="zh_CN" scope="application"/>
<fmt:formatNumber value="6.21" type="currency" var="format_number"/>
<c:out value="${format_number}"/><br>
美式英语语言环境下：
<fmt:setLocale value="en_US" scope="application"/>
<fmt:formatNumber value="6.21" type="currency" var="format_number"/>
<c:out value="${format_number}"/>
</center>
</body>
</html>
```

输出人民币符号

输出美元符号

图 7.75　fmt_setLocale.jsp 示例

在浏览器地址栏中输入 http://localhost:8080/JSTL/fmt_setLocale.jsp，显示结果如图 7.76 所示。

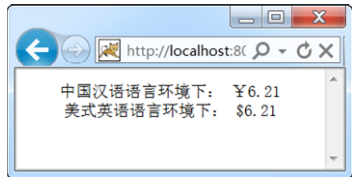



图 7.76　fmt_setLocale.jsp 示例运行结果

### 2．<fmt:requestEncoding>标签

<fmt:requestEncoding>标签用于向 JSP 容器指定请求（request）的字符编码，其语法格式如图 7.77 所示。

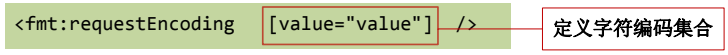



图 7.77　<fmt:requestEncoding>标签语法格式

<fmt:requestEncoding>标签就相当于 JSP 内置对象中的 setCharacterEncoding()方法，我们就不再为大家举例了。

## 7.6.2　消息标签

消息分类标签包含 4 个标签，分别是<fmt:message>、<fmt:param>、<fmt:bundle>和<fmt:setBundle>。这些标签的主要功能为获取系统设定的语言资源，从而可以轻易地做到多国化信息。由于这些标签常常是一起使用的，所以我们在对其使用格式进行说明后再统一举例说明它们的用法。

### 1．<fmt:message>标签

<fmt:message>标签用来根据本地化环境从资源包检索文本信息，从而实现文本的本地化。<fmt:message>标签的使用语法如图 7.78 所示。

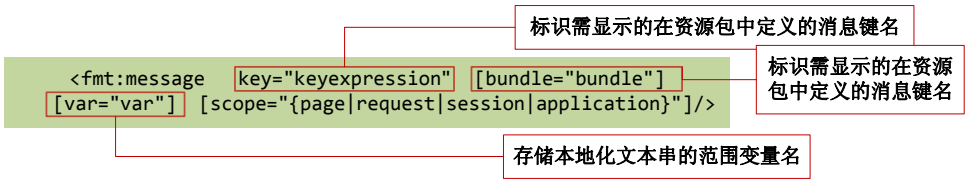



图 7.78　<fmt:message>标签使用语法

### 2．<fmt:bundle>标签

<fmt:bundle>标签用来根据本地化环境来选择所需的资源包，<fmt:bundle>标签的使用语法如图 7.79 所示。

### 3．<fmt:setBundle>标签

<fmt:setBundle>标签用来为本地化环境设置一个默认的资源包，在<fmt:message>标签的特定作用域内起作用，<fmt:setBundle>标签的使用语法如图 7.80 所示。

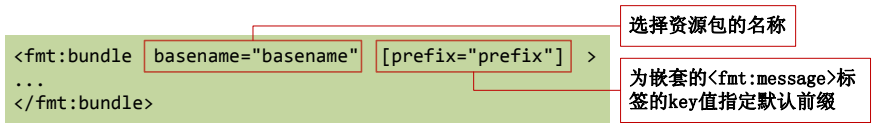



图 7.79　<fmt:bundle>标签使用语法

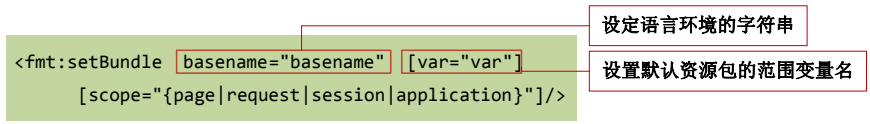



图 7.80　<fmt:setBundle>标签使用语法

### 4．<fmt:param>标签

<fmt:param>标签用来为<fmt:message>标签指定文本消息参数值，动态的设定参数。<fmt:param>标签的使用语法如图 7.81 所示。

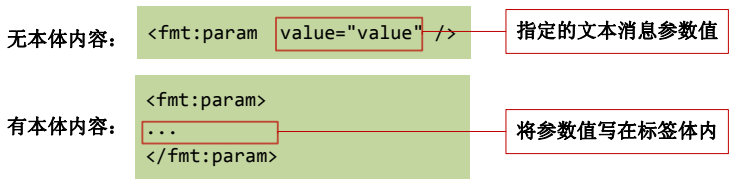



图 7.81　<fmt:param>标签使用语法

在使用消息标签之前，必须首先创建资源文件，其扩展名为 properties，资源文件内容必须按照“key=value”的格式，也就是一个索引对应一个值。资源文件需要放置在应用程序的 WEB-INF/classes 目录下。

**【示例 7.19】**我们举一个例子 fmt_message.jsp 来看如何运用消息标签。首先，我们在项目中新建一个文本文件，将其命名为 Resource.properties，并将其存放在项目的 src 目录下，这样就创建了消息资源文件，创建对话框如图 7.82 所示。

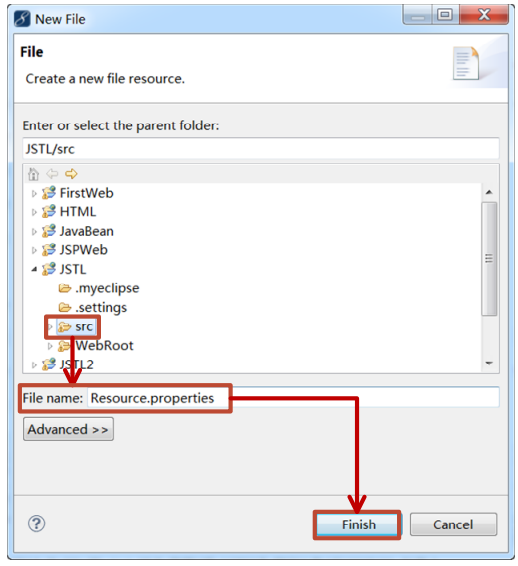



图 7.82　创建消息资源文件对话框

单击“Finish”按钮，在打开的资源文件编辑器中输入以下代码：

```
str = Hi! {0} </br>  today is {1 , date , long} </br> time is {2 , time , full }
</br> number is {3 , number , #,# }
```

其中，str 是 key，在页面中就是通过它来获取等号右边的值的。

然后我们在项目中创建使用 JSTL 格式化标签库中消息标签的页面 fmt_message.jsp，其具体代码如图 7.83 所示。

```
<%@ page import="java.util.Date"%>
<%@ page import="java.lang.Double"%>
<%@ page contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
<head>
  <title>I18N</title>
</head>
 <body><%
    request.setAttribute("now", new Date());
    request.setAttribute("num", new Double(1987.621));
  %>
  <fmt:bundle basename="Resource">
    <fmt:message key="str">
      <fmt:param value="sun"/>
      <fmt:param value="${now}"/>
      <fmt:param value="${now}"/>
      <fmt:param value="${num}"/>
    </fmt:message>
  </fmt:bundle>
 </body>
</html>
```

bundle标签设置消息资源文件

message标签设置要显示的消息的key

param标签传递参数

图 7.83　fmt_message.jsp 示例

在浏览器地址栏中输入 http://localhost:8080/JSTL/fmt_message.jsp，显示结果如图 7.84 所示。


图 7.84　fmt_message.jsp 示例运行结果

### 7.6.3 数字日期格式化标签

数字日期格式化分类标签包含 6 个标签，分别是<ftm:setTimeZone>、<ftm:TimeZone>、<ftm:formatNumber>、<ftm:formatDate>、<ftm:parseDate>和<ftm:parseNumber>。该类标签的主要功能是对页面的数字和日期进行格式化定制。由于篇幅原因，我们在对其使用格式进行说明后再统一举例说明它们的用法。

#### 1．<fmt:timeZone>标签

<fmt:timeZone>标签用来设置时区，其使用语法如图 7.85 所示。

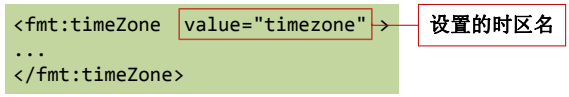



图 7.85 <fmt:timeZone>标签使用语法

#### 2．<fmt:setTimeZone>标签

<fmt:setTimeZone>标签用来将设定了的时区存储在某个范围变量 var 中，其使用语法如图 7.86 所示。

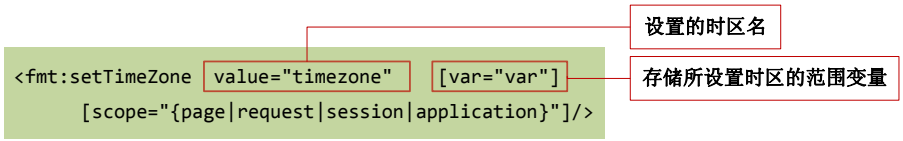



图 7.86 <fmt:setTimeZone>标签使用语法

#### 3．<fmt:formatDate>标签

<fmt:formatDate>标签用来设定日期和时间的格式并按照设置的格式给予输出，其语法如图 7.87 所示。

```
<fmt:formatDate  value="date/time"  [type="type"]  [dateStyle="datestyle"]
[timeStyle="timestyle"]  [pattern="patternexpression"]  [timeZone="timezone"]
[var="varname"]  [scope="{page|request|session|application}"]  />
```

图 7.87 <fmt:formatDate>标签使用语法

<fmt:formatDate>标签的各个属性的具体描述如表 7.11 所示。

表 7.11 <fmt:formatDate>标签属性

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| value | java.util.Date 类 | 指定要进行格式化的日期/时间 |
| type | String | 指定要设置格式的 Date 实例部分，可为 time/date/both |
| dateStyle | String | 设置使用于特定语言环境的日期格式，可为 default/short/medium/long/full |
| timeStyle | String | 设置使用于特定语言环境的时间格式，可为 default/short/medium/long/full |
| pattern | String | 用户自定义日期/时间格式 |

续表

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| timeZone | String 或 java.util.TimeZone | 需格式化的时间所在时区 |
| var | String | 用来存储格式化后结果的范围变量名 |
| scope | String | 范围变量 var 的作用域 |

### 4.<fmt:parseDate>标签

<fmt:parseDate>标签用于解析日期和时间值，其使用语法如图 7.88 所示。

**无本体内容：**

```
<fmt:parseDate  value="data/time"[type="type"]  [dateStyle="datestyle"]
[timeStyle="timestyle"]  [pattern="patternexpression"][timeZone="timezone"]
[parseLocale="parselocale"][var="var"][scope="{page|request|session|application}"]/>
```

**有本体内容：**

```
<fmt:parseDate [type="field"]  [dateStyle="datestyle"] [timeStyle="timestyle"]
[pattern="patternexpression"][timeZone="timezone"]  [parseLocale="parselocale"]
[var="var"][scope="{page|request|session|application}"]>

body content

</fmt:parseDate>
```

图 7.88　<fmt:parseDate>标签使用语法

<fmt:parseDate>标签的各个属性的具体描述如表 7.12 所示。

表 7.12　<fmt:parseDate>标签属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| value | java.util.Date 类 | 指定要进行解析的日期/时间 |
| type | String | 指定需要解析的 Date 实例部分，可为 time/date/both |
| dateStyle | String | 设置 Date 实例日期部分解析方式，可为 default/short/medium/long/full |
| timeStyle | String | 设置 Date 实例时间部分解析方式，可为 default/short/medium/long/full |
| pattern | String | 用户自定义日期/时间的格式以确定解析方式 |
| timeZone | String 或 java.util.TimeZone | 需解析的时间所在时区 |
| parseLocale | String 或为 java.util.Locale 类 | 指定一种语言环境，根据这种语言环境来解析日期/时间值 |
| var | String | 用来存储解析结果的范围变量名 |
| scope | String | 范围变量 var 的作用域 |

### 5．<fmt:formatNumber>标签

<fmt:formatNumber>标签用于格式化数值，即设置特定语言环境下的数值的输出方式，其使用语法如图 7.89 所示。

```
<fmt:formatNumber  value="number"[type="type"] [pattern="patternexpression"]
[currencyCode="currencycode"][currencySymbol="currencysymbol"]
[maxintegerDigits="maxintegerdigits"][minintegerDigits="minintegerdigits"]
[maxFractionDigits=" maxfractiondigits"][minFractionDigits=" minfractiondigits"]
[groupingUsed="groupinUsed"][var="var"]  [scope="scope"]/ >
```

图 7.89　<fmt:formatNumber>标签使用语法

<fmt:formatNumber>标签的属性描述如表 7.13 所示。

表 7.13　<fmt:formatNumber>标签属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| value | String 或为 Number | 指定需格式化的数值 |
| type | String | 指定格式化方式，其取值有 number、currency、percent |
| pattern | String | 用户自定义格式化方式 |
| currencyCode | String | 设置所显示数值的货币单位 |
| currencySymbol | String | 设置所显示数值的货币符号 |
| maxintegerDigits | int | 设置格式化后数值的最大整数位数 |
| minintegerDigits | int | 设置格式化后数值的最小整数位数 |
| maxFractionDigits | int | 设置格式化后数值的最大小数位数 |
| minFractionDigits | int | 设置格式化后数值的最小小数位数 |
| groupingUsed | Boolean | 指定格式化后是否要对小数点前面的数字分组 |
| var | String | 用来存储格式化后数值的范围变量名称 |
| scope | String | 范围变量 var 的作用域 |

### 6．<fmt:parseNumber>标签

<fmt:parseNumber>标签用来解析数值字符串，其使用语法如图 7.90 所示。

```
<fmt:parseNumber  value="numberstring"[type="type"]
[pattern="patternexpression"][parseLocale="parseLocale"]
[integerOnly="integerOnly"][var="var"]  [scope="scope"]/>
```

图 7.90　<fmt:parseNumber>标签使用语法

<fmt:parseNumber>标签的属性描述如表 7.14 所示。

表 7.14　<fmt:parseNumber>标签属性描述

| 名　称 | 类　型 | 属 性 描 述 |
|---|---|---|
| value | String | 需解析的数值字符串 |
| type | String | 指定数值字符串，其取值有 number、currency、percentage |
| pattern | String | 用户自定义数值字符串的格式以确定解析方式 |
| parseLocale | String 或为 java.util.Locale 类 | 指定一种语言环境，根据这种语言环境来解析数值字符串 |
| integerOnly | Boolean | 设置要解析的部分是否仅为数值字符串的整数部分 |
| var | String | 存储数值解析结果的范围变量名 |
| scope | String | 范围变量 var 的作用域 |

**【示例 7.20】**下面我们还是通过一个实例来具体了解数字日期格式化标签的使用方法，创建的 fmt_numberDate.jsp 页面代码如图 7.91 所示。

在浏览器地址栏中输入 http://localhost:8080/JSTL/fmt_numberDate.jsp，显示结果如图 7.92 所示。

```
<%@ page language="java" import="java.util.*" contentType="text/
html;charset=gb2312"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<html>
 <head>
   <title>数字格式标签示例</title>
 </head>
 <body>
 <font size=4>
   <fmt:setLocale value="en-US" />
    美国：<br>
    数字格式：<fmt:formatNumber value="123456789" /><br>
    百分数格式：<fmt:formatNumber type="percent">123456789</fmt:formatNumber><br>
    货币格式<fmt:formatNumber value="12345.67" type="currency" /><br>

   <c:set var="date" value="<%=new java.util.Date()%>"/>
      <fmt:formatDate value="${date}"
          pattern="yyyy年MM月dd日" type="date" dateStyle="full"/><br>
      <fmt:formatDate value="${date}"
          pattern="HH:mm:ss" type="time"  timeStyle="full"/><br>
      <fmt:formatDate value="${date}"
          pattern="yyyy年MM月dd日 HH:mm:ss"
          type="both" dateStyle="full" timeStyle="full"/><br>
      <fmt:formatDate value="${date}" type="both"
          timeStyle="full" var="formatted"/>
      <fmt:parseDate value="${formatted}" type="both"
          timeStyle="full"  timeZone="GMT" var="parsed"/>
      <c:out value="${parsed}"/><br>
  </font>
 </body>
</html>
```

数字格式标签

日期格式标签

图 7.91　fmt_numberDate.jsp 示例

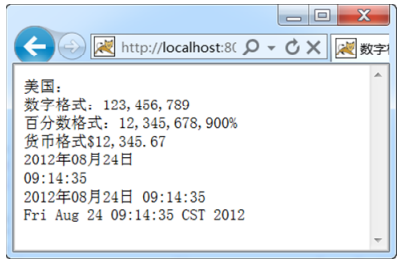



图 7.92　fmt_numberDate.jsp 示例运行结果

## 7.7　JSTL 数据库标签库

数据库开发在 JSP 中占有非常重要的地位，JSTL 也提供了对数据库操作的支持，通过使用 JSTL，数据库操作可以简化为简单的几个标签，大大提高了数据库开发的效率和程序的可

维护性。JSTL 数据库操作标签库中的标签，根据其功能大致可以分为两类，如图 7.93 所示。

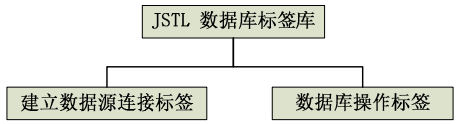



图 7.93　数据库标签库

如果要在 JSP 页面中使用数据库标签库的标签，需要用 taglib 指令指明这个标签库的路径为如图 7.94 所示的形式。

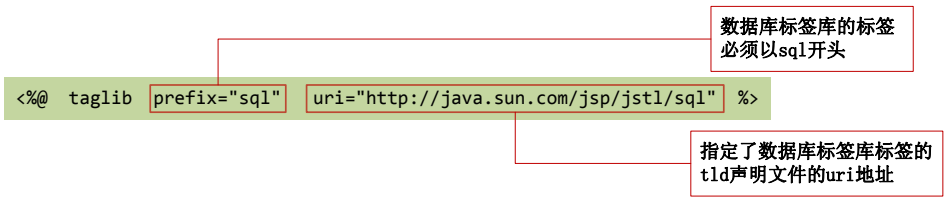



图 7.94　使用数据库标签库

## 7.7.1　建立数据源连接标签

建立数据源连接分类标签只包含一个标签<sql:setDataSource>，该标签用来建立数据库连接，<sql:setDataSource>标签的使用语法如图 7.95 所示。

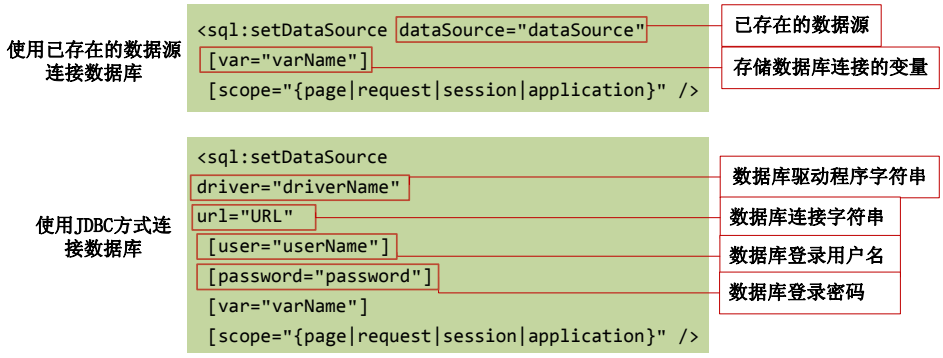



图 7.95　<sql:setDataSource>标签使用语法

**【示例 7.21】**下面我们通过一个实例来具体了解<sql:setDataSource>标签的使用，我们还是使用 MySQL 数据库，创建的连接数据库代码如图 7.96 所示。

```
<sql:setDataSource
 var="example"
 driver="com.mysql.jdbc.Driver"
 url="jdbc:mysql://127.0.0.1:3306/mysqltest"
 user="root"
 password="123456"
/>
```

图 7.96　<sql:setDataSource>标签连接数据库代码

### 7.7.2　数据库操作标签

数据库操作分类标签包含 5 个标签，分别是<sql:query>、<sql:update>、<sql:param>、<sql:dataParam>和<sql:transaction>。

#### 1．<sql:query>标签

<sql:query>标签的功能是执行数据库中的查询操作，<sql:query>标签的使用语法如图 7.97 所示。

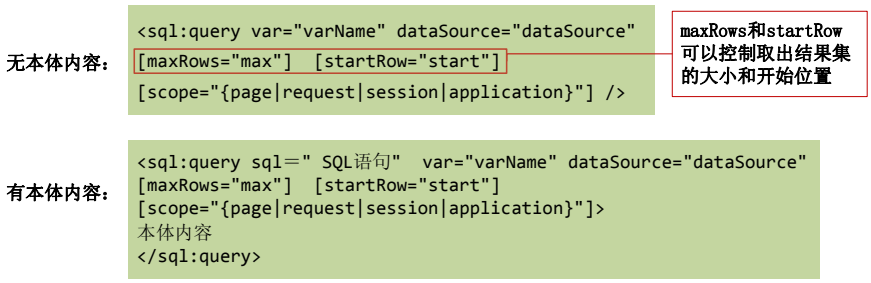



图 7.97　<sql:query>标签使用语法

#### 2．<sql:param>标签

<sql:param>标签的功能就是向<sql:query>标签的 SQL 语句中传递参数，该标签的使用语法如图 7.98 所示。

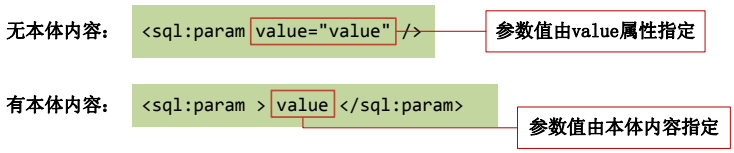



图 7.98　<sql:param>标签的使用语法

#### 3．<sql:update>标签

<sql:update>标签的功能是对数据库进行插入、更新和删除操作，<sql:update>标签的使用语法如图 7.99 所示。

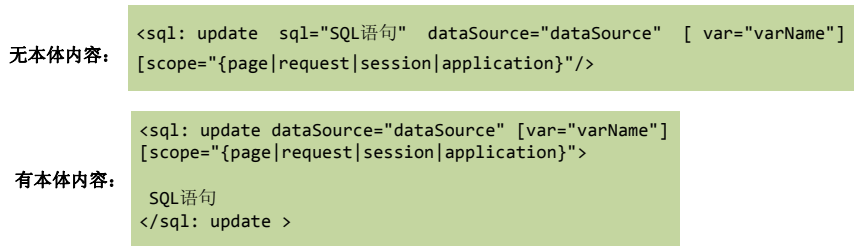



图 7.99　<sql:update>标签的使用语法

#### 4．<sql:dateParam>标签

<sql:dateParam>标签和<sql:param>标签的功能和用法完全相同，不同之处是<sql:dateParam>标签是用来设置日期格式的参数，其使用语法如图 7.100 所示。

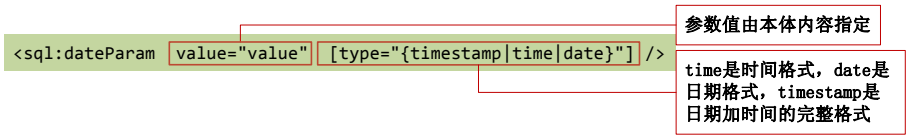



图 7.100　<sql:dateParam>标签使用语法

### 5．<sql:transaction>标签

在 JSTL 中，同样支持数据库操作的事务处理，在 JSTL 中是采用<sql:transaction>标签来实现这个功能的，<sql:transaction>标签的使用语法如图 7.101 所示。

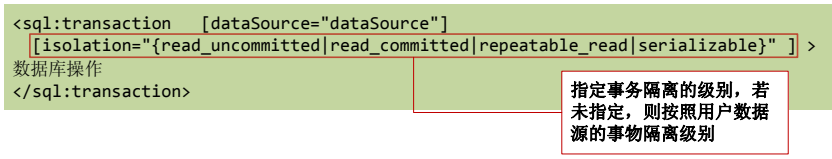



图 7.101　<sql:transaction>标签的使用语法

**【示例 7.22】**下面我们通过一个实例 sql_sql.jsp 来具体了解数据库标签的使用方法，我们还是使用 MySQL 数据库，具体代码如图 7.102 所示。

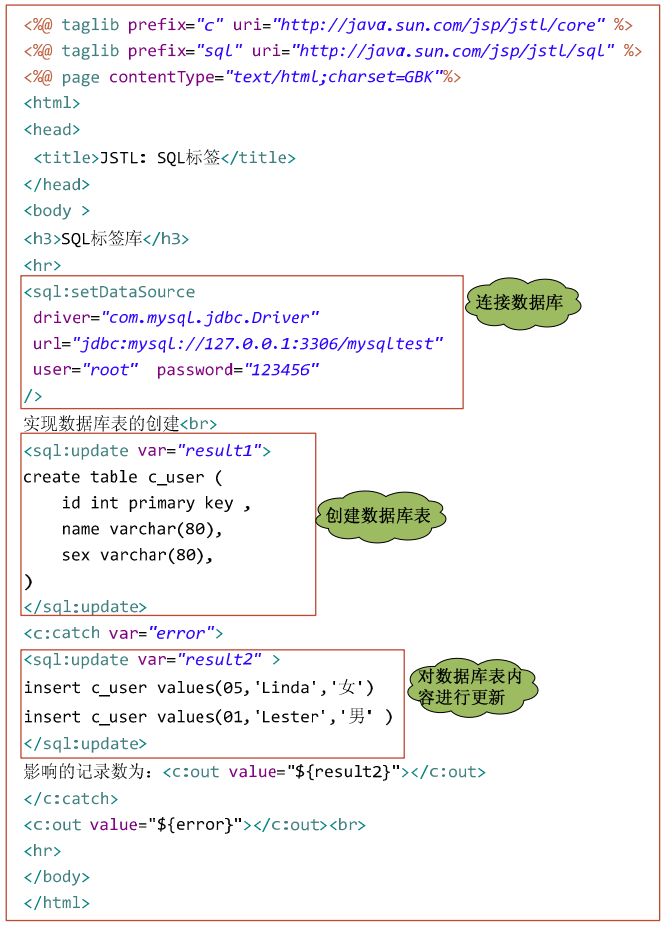



图 7.102　sql_sql.jsp 示例

## 7.8　JSTL 函数标签库

JSTL 函数标签库就是一些常用的函数，在 JSTL 中把这些常用的函数封装成标签的形式，然后可以在 JSP 页面上进行方便的调用。如果要在 JSP 页面中使用函数标签库的标签，需要用 taglib 指令指明这个标签库的路径为如图 7.103 所示的形式。

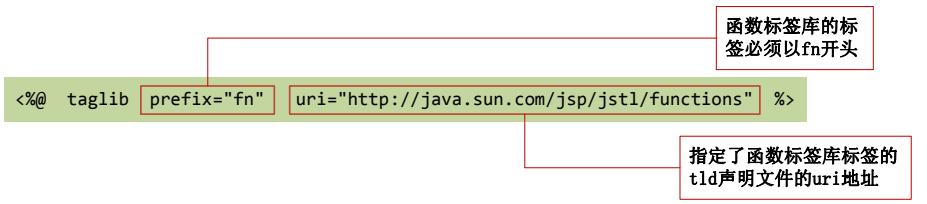



图 7.103　使用函数标签库

限于篇幅，本节以表格的方式列出每个函数标签的语法和功能，而不再对每一个函数的参数作具体的讲解，如表 7.15 所示。有兴趣的读者可参阅有关书籍。

表 7.15　JSTL函数库常用标签

| 函　　数 | 描　　述 |
|---|---|
| fn:contains(string, substring) | 如果参数 string 中包含参数 substring，返回 true |
| fn:containsIgnoreCase(string,substring) | 如果参数 string 中包含参数 substring（忽略大小写），返回 true |
| fn:endsWith(string, suffix) | 如果参数 string 以参数 suffix 结尾，返回 true |
| fn:escapeXml(string) | 将有特殊意义的 XML(HTML)转换为对应的 XML character entity code，并返回 |
| fn:indexOf(string, substring) | 返回参数 substring 在参数 string 中第一次出现的位置 |
| fn:join(array, separator) | 将一个给定的数组 array 用给定的间隔符 separator 串在一起，组成一个新的字符串并返回 |
| fn:length(item) | 返回参数 item 中包含元素的数量。参数 Item 类型是数组、collection 或者 String。如果是 String 类型,返回值是 String 中的字符数 |
| fn:replace(string, before, after) | 返回一个 String 对象。用参数 after 字符串替换参数 string 中所有出现参数 before 字符串的地方，并返回替换后的结果 |
| fn:split(string, separator) | 返回一个数组，以参数 separator 为分割符分割参数 string，分割后的每一部分就是数组的一个元素 |
| fn:startsWith(string, prefix) | 如果参数 string 以参数 prefix 开头，返回 true |
| fn:substring(string, begin, end) | 返回参数 string 部分字符串，从参数 begin 开始到参数 end 位置，包括 end 位置的字符 |
| fn:substringAfter(string, substring) | 返回参数 substring 在参数 string 中后面的那一部分字符串 |
| fn:substringBefore(string, substring) | 返回参数 substring 在参数 string 中前面的那一部分字符串 |
| fn:toLowerCase(string) | 将参数 string 所有的字符变为小写，并将其返回 |
| fn:toUpperCase(string) | 将参数 string 所有的字符变为大写，并将其返回 |

**【示例 7.23】**下面以一个简单的实例做示范，其他函数的使用方法与此相类似。创建一个 fn_length.jsp 文件，其源代码如图 7.104 所示。

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ page contentType="text/html;charset=GBK"%>
<%@ page isELIgnored="false" %>
<html>
<head>
 <title>JSTL: Funciton -- fn:length</title>
</head>
<body bgcolor="#FFFFFF">
<h3>fn:length()</h3>
<hr>
<c:set var="s1" value="I love Java Web!"/>
<%
String[] customers = {"Lester","Johnson","Jake"};
session.setAttribute("customers",customers);
%>
<table cellpadding="5" border="1">
 <tr>
   <th align="left">Input String</th> <th>Result</th>
 </tr>
 <tr>
   <td>${s1}</td> <td>${fn:length(s1)}</td>
 </tr>
 <tr>
   <td>${customers}</td> <td>${fn:length(customers)}</td>
 </tr>
 <tr>
   <td>null</td> <td>${fn:length(undefined)}</td>
 </tr>
 <tr>
   <td>empty string</td> <td>${fn:length("")}</td>
 </tr>
</table>
</body>
</html>
```

返回字符串s1中的字符个数

返回customers数组的维数

当字符串变量为NULL或者空时，返回0

图 7.104　fn_length.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/JSTL/fn_length.jsp 来测试这个示例，显示结果如图 7.105 所示。

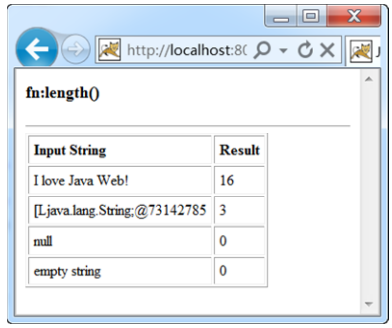



图 7.105　fn_length.jsp 示例运行结果

## 7.9　小结

本章介绍了 EL 表达式语言和 JSTL 标准标签库的基本知识，包括 EL 简介和 EL 应用，以

及 JSTL 核心标签库、XML 标签库、国际化标签库和数据库操作标签库等。本章的重点和难点都是能否熟练使用 JSTL 标准标签库，尤其是核心标签库进行编程和开发。读者在实际的开发过程中可以参考本章的示例程序，在学习本章知识的基础上熟练运用 JSTL 提高开发速度和质量，并在自己的开发中尝试 JSTL 标签库来简化开发的代码量。

## 7.10　本章习题

1．请读者参考示例 7.2，创建一个旅游地门票显示的例子，运行结果如图 7.106 所示。


图 7.106　运行结果

【分析】本题主要考查读者对于 EL 表达式的应用。这种表达式不易掌握，读者可以参照示例 7.2 来做相应处理，从而实现题目的要求。

【核心代码】本题的核心代码如下所示。

SetAccess.java:

```
import java.io.IOException;
……
public class SetAccess extends HttpServlet {
      public void doGet(HttpServletRequest request,HttpServletResponse response)
                              throws ServletException, IOException {
               String[] name = { "1-兵马俑", "2-黄帝陵", "3-法门寺" };
               ArrayList city = new ArrayList();
               city.add("西安");
               city.add("高陵");
               city.add("宝鸡");
……
               request.setAttribute("name", name);
               request.setAttribute("city", city);
               request.setAttribute("price", price);
               RequestDispatcher dispatcher =
               request.getRequestDispatcher("SetAccess.jsp");//转向 SetAccess.jsp
               dispatcher.forward(request, response);
      }
}
```

SetAccess.jsp：

```
<%@ page contentType="text/html; charset=GB2312" %>
<!DOCTYPE HTML PUBLIC "-//w3c//dtd html 4.0 transitional//en">
<html>
<body bgcolor="#FFFFFF">
<center>访问集合中的元素</center>
<TABLE ALIGN="CENTER" border=1>
<TR><TD>名称</TD> <TD> 城市</TD> <TD>票价 </TD></TR>
<TR><TD>${name[0]}</TD> <TD> ${city[0]}</TD> <TD> ${price[0]}</TD></TR>
……
```

```
</TABLE>
</body>
</html>
```

web.xml：

```
<servlet>
<display-name>SetAccess</display-name>
<servlet-name>SetAccess</servlet-name>
<servlet-class>SetAccess</servlet-class>
</servlet>
<servlet-mapping>
<servlet-name>SetAccess</servlet-name>
<url-pattern>/SetAccess</url-pattern>
</servlet-mapping>
```

2．请读者参照示例 7.9，使用 JSTL 核心标签库中的<c:if/>标签，判断我们输入的值是否为 2。执行结果如图 7.107 所示。

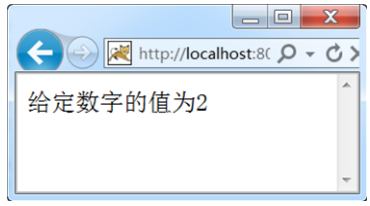



图 7.107　运行结果

【分析】本题主要考查读者对于核心标签库知识的运用。

【核心代码】本题的核心代码如下所示。

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<html>
<body>
 <font size="2">
   <c:set var="number" value="2" scope="request"/>
   <c:if test="${number==1}">
     <c:out value="给定数字的值为 1"></c:out>
   </c:if>
……
  </font>
 </body>
</html>
```

3．请读者参照示例 7.18 使用国际化标签来用美国和中国不同的格式来表示数字，可以用百分数或者是货币，执行结果如图 7.108 所示。

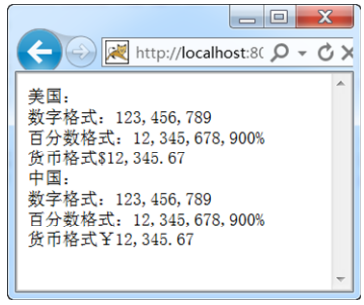



图 7.108　运行结果

【分析】本题主要考查读者对于格式化标签库中国际化标签知识的运用。

【核心代码】本题的核心代码如下所示。

```
    <%@ page language="java" import="java.util.*" contentType="text/html;charset=
gb2312"%>
    <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
    <%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
    <html>
     <body>
     <font size="4">
        <fmt:setLocale value="en-US" />
          美国：<br>
          数字格式：<fmt:formatNumber value="123456789" /><br>
          百分数格式：<fmt:formatNumber type="percent">123456789</fmt:formatNumber><br>
          货币格式<fmt:formatNumber value="12345.67" type="currency" /><br>
         <fmt:setLocale value="zh_CN" />
          中国：<br>
    ……
      </font>
     </body>
    </html>
```

PART 2

# Struts 2 技术篇

# 第 8 章　Struts 2 框架入门

Struts 这个名字来源于在建筑和旧式飞机中使用的支持金属架。它是第一个实现了 Web 层 MVC 架构的开源框架。本章我们在简要介绍 MVC 模式和 Struts 2 框架安装基础上，实现我们第一个 HelloWorld 程序的配置与实现。

## 8.1　Struts 2 概述

### 8.1.1　Struts 2 的由来

Struts 2 是 Struts 的下一代产品，是在 Struts 和 WebWork 的技术基础上进行了合并的全新的 Struts 2 框架，如图 8.1 所示。


图 8.1　Struts 2 的由来

但是 Struts 2 的体系结构与 Struts 1 的体系结构的差别很大。Struts 2 是以 WebWork 为核心的，所以 Struts 2 可以理解为 WebWork 的更新产品。但是由于 Struts 1 名声较大的缘故，所以合并之后 Apache 基金会将其命名为 Struts 2。

Struts 2 是一个基于 J2EE 平台的 MVC 框架，它主要是采用 Servlet 和 JSP 技术来实现的。下面我们就带领大家一起来进入 Struts 2 框架的世界。

### 8.1.2　MVC 模式

MVC 是一种设计模式，最早是由 Xerox（施乐）公司在 20 世纪 80 年代提出的。随后，它成为了一种著名的用户界面设计架构，如图 8.2 所示。

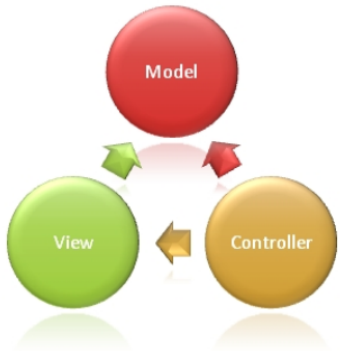



图 8.2　MVC 模式

MVC 英文全称为 Model-View-Controller，即把一个应用程序的输入层、业务处理层、控制流程层按照 View、Model、Controller 的方式实现了分离，并分别承担不同的任务。图 8.3 显示了这三个模块各自的功能。

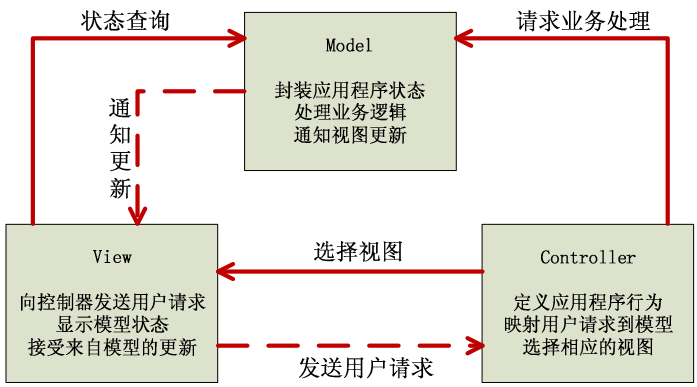



图 8.3　MVC 三个模块的功能

MVC 模式体现了分层设计的思想，它有以下几点好处：

- 从视图方面来说，由于多种视图可共享一个后台模型，这就为实现多种用户界面提供了便利。
- 从模型方面来说，由于其实现与界面独立，因此模型只需提供接口供上层调用，很好地体现了面向对象设计的信息封装和隐藏的原则。
- 从控制器方面来说，控制器作为介于视图和后台模型间的控制组件，可更好的维护程序流程，选择业务模型，选择用户视图，使程序的调用规则更加清晰，很大程度上优化了系统结构。

正是由于 MVC 的优势，使它成为软件设计的典范，目前大多数系统都采用了 MVC 模式来进行系统架构与实现。

### 8.1.3　Java Web 的实现模型

在 Java Web 领域存在着两种经典模型，也可以称为实现模式，分别是 Model 1 和 Model 2。这两种模型都是由 Sun 公司提出的，它们都可被看做是 MVC 的具体实现形式。现在我们就来

比较一下这两种模型。首先来看 Model 1，如图 8.4 所示。


图 8.4　Model 1 模型

在这种模型中，JSP 充当着控制器与视图的双重角色，JavaBean 扮演了模型的角色。JSP 直接调用后台模型进行业务处理，同时，再由 JSP 返回用户结果界面，如图 8.5 所示。

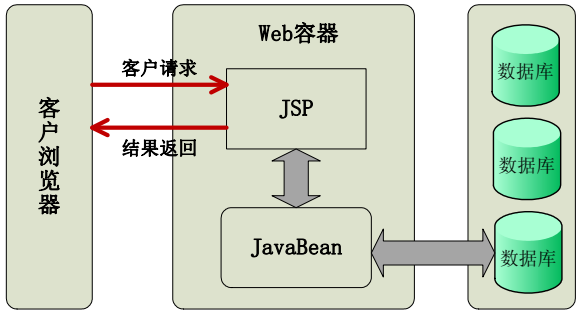



图 8.5　Model 1 原理

这种模型对于一些小型的程序还是可以满足的，但对于大规模的系统就显得有些力不从心。因为倘若将 JSP 既当成控制器又当成视图，那么在页面代码里就会有大量的 HTML 标记与 Java 语言的混合物，这对程序的维护是非常不利的，而且对于页面开发人员和程序设计人员的分工将造成太多约束，无法使它们并行工作，开发效率也就被大大降低。所以 Sun 公司在 Model 1 基础上开发出了 Model 2 模型，如图 8.6 所示。

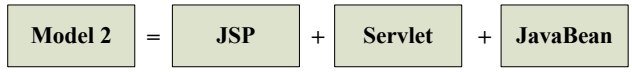



图 8.6　Model 2 模型

在 Model 2 模型中，JSP 既作为视图又作为控制器的局面不再存在了，而是使用了 Servlet 作为控制器，JSP 则单纯的只负责显示逻辑（还包括很少量的 Java 代码），如图 8.7 所示。

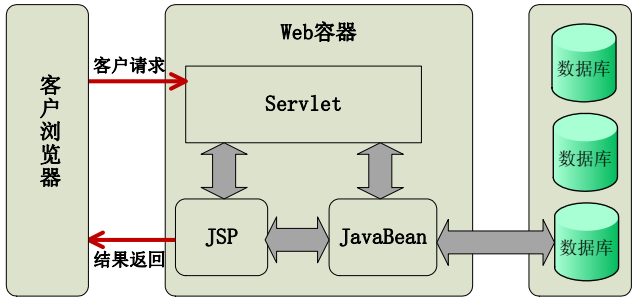



图 8.7　Model 2 原理

Model 2 清楚的划分了表达、控制、模型这三层结构，很好的实现了 MVC 设计思想。因此，对于大型系统的设计与开发 Model 2 提供了很大的帮助。

### 8.1.4　为什么要使用 Struts 2

Struts 2 是目前最为成功的 J2EE 框架之一，在众多的 MVC 框架之中脱颖而出，受到了绝

大部分程序员的青睐。究其原因，是因为 Struts 2 具备了其他框架无法比拟的优势，如图 8.8 所示。

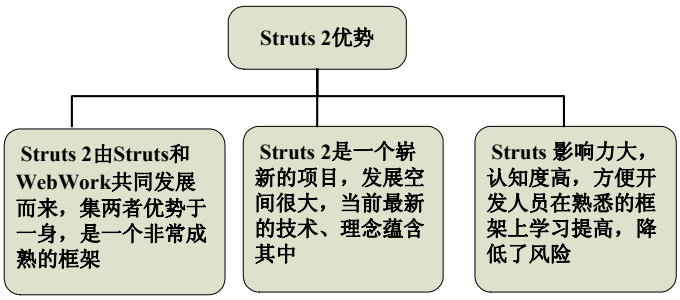



图 8.8　Struts 2 的优势

Struts 2 有着如此大的优势，相信大家已经跃跃欲试了，下面我们就来介绍一下如何在自己的计算机中完成 Struts 2 的安装和配置。

## 8.2　Struts 2 的下载与安装

本节我们为大家介绍如何下载和安装 Struts 2，以及 Struts 2 中包含文件的作用，然后通过 Struts 2 自带的实例验证 Struts 2 安装是否成功。

### 8.2.1　Struts 2 的下载过程

Apache 官方网站提供最新版本的 Struts 2 下载，所以建议读者到官方网站下载。下面将详细讲解 Struts 2 的下载过程。

在浏览器地址栏中输入 Apache struts 官方网站网址 http://struts.apache.org/。页面更新后单击 Recent Release 模块中的 Struts 2.3.4 版本进行下载，如图 8.9 所示。


图 8.9　Apache struts 官方网站网址

**注意**：Struts 1 与 Struts 2 仍然在同一个网页中，没有明显区别，下载时读者要特别注意版本号。即版本号要以 2 开头。

在进入的下载页面中，我们选择 Full Releases 版本集合进行下载。单击 Struts 2.3.4 中的完全发布版（Full Distribution）struts-2.3.4-all.zip 版本完成 Struts 2 的下载，如图 8.10 所示。

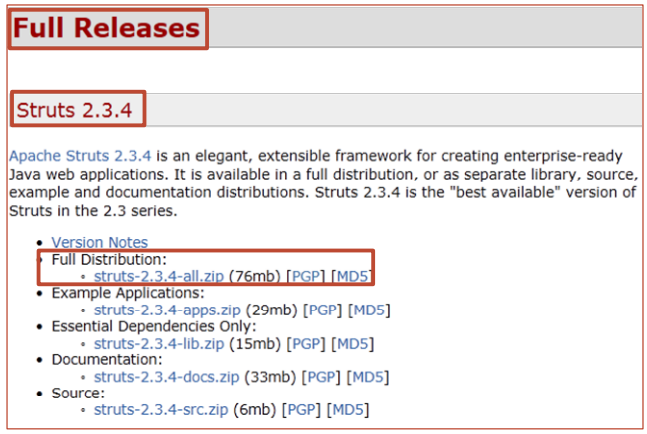



图 8.10　Struts 2 下载页面

下载完成后，我们得到一个 zip 文件，将其进行解压，可以看到该文件夹中包括 4 个目录。它们各自具有不同的作用，如图 8.11 所示。

| | |
|---|---|
| src源代码目录 | 存放*.java文件，Struts 2是一个开源项目，可在此存放所有的代码 |
| doc文档目录 | 存放文档文件 |
| lib库目录 | 存放提供给开发人员的jar文件，开发过程中需要将其加入到CLASSPATH中 |
| apps为例子目录 | 存放Struts 2给出的实例，都是*.war文件 |

图 8.11　安装文件中各子文件的作用

## 8.2.2　Struts 2 安装过程

首先我们在 MyEclipse 软件中新建一个 Struts 项目工程，在 Struts 工程的右键菜单上选择“Properties”命令，打开“Properties for Struts2”对话框，在 Struts 2 解压后的 lib 目录中选择如下 7 个文件：

- struts2-core-2.3.4.jar
- xwork-core-2.3.4.jar
- ognl-3.0.5.jar
- freemarker-2.3.19.jar
- commons-logging-1.1.1.jar
- commons-fileupload-1.2.2.jar

- commons-lang3-3.1.jar

添加过程如图 8.12 所示。

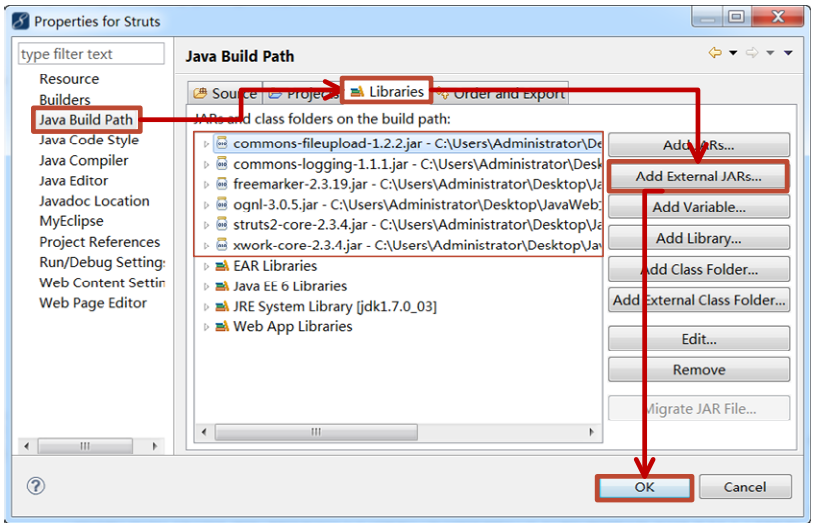



图 8.12　Struts 2 依赖的 jar 包

把 apps 目录下文件名 struts2-blank.war 的文件复制到 Tomcat 的 webapps 下。重新启动 Tomcat，访问 http://localhost:8080/struts2-blank/来测试这个实例，如果出现图 8.13 所示的界面，说明 Struts 2 安装成功了。

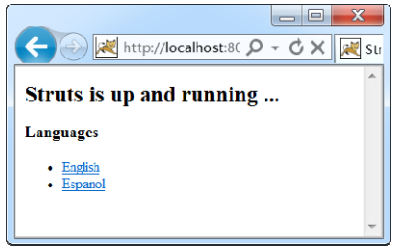



图 8.13　Struts 2 实例

## 8.3　使用 Struts 2 实现第一个程序

这一节我们为大家介绍如何利用 Struts 2 进行 Web 应用开发。首先我们学习一下 Struts 2 的工作流程。

### 8.3.1　Struts 2 的工作流程

Struts 2 与 WebWork 的工作方式类似，Struts 2 同样使用了拦截器作为其处理用户请求的控制器。在 Struts 2 中有一个核心控制器 FilterDispatcher，它负责处理用户的所有请求，如果遇到以.action 结尾的请求 URL，就会交给 Struts 2 框架来处理，Struts 2 的工作流程我们可以用图 8.14 来表示。

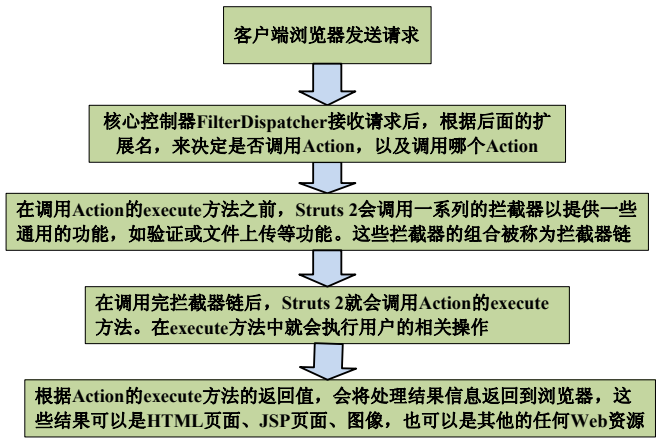



图 8.14　Struts 2 的工作流程

我们还可以对图 8.14 的内容简化为如图 8.15 所示的样式。

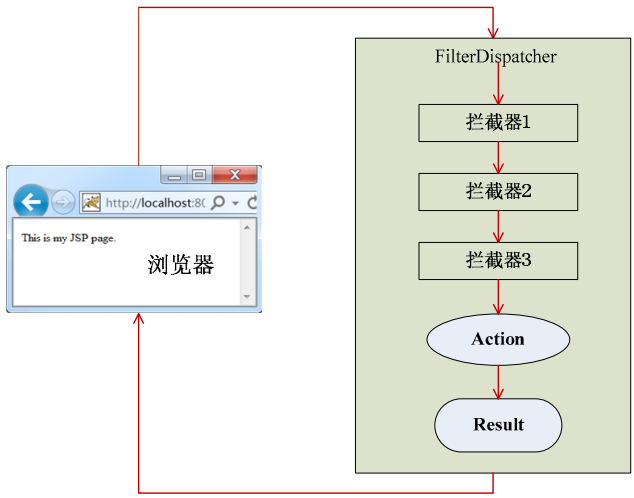



图 8.15　Struts 2 的工作流程简图

## 8.3.2　开发一个 Struts 2 框架程序的步骤

对应于 Struts 2 的工作流程，我们来为大家讲解开发一个 Struts 2 框架程序的步骤，其步骤大致如图 8.16 所示。

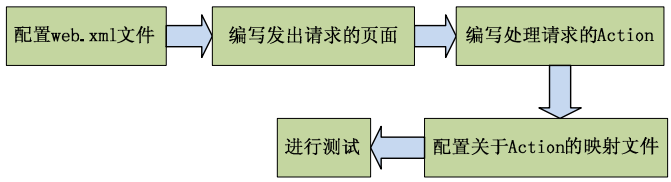



图 8.16　开发 Struts 2 框架程序的步骤

知道了 Struts 2 框架程序的开发步骤，我们就可以开始 Struts 2 程序的开发了，同样，我们来看看如何用 Struts 2 输出最经典的“HelloWorld!”语句。

### 8.3.3　配置 web.xml

Struts 2 的 web.xml 文件配置方法非常简单，即在 web.xml 中配置 Struts 2 提供的过滤器，并设置为所有的请求（/*）都要通过这个过滤器，如图 8.17 所示。

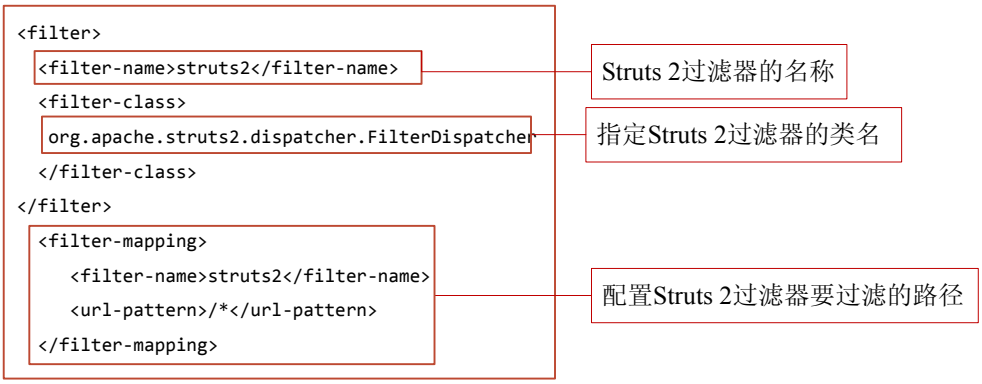



图 8.17　Struts 2 中 web.xml 文件的配置

### 8.3.4　编写 JSP 界面

接下来我们编写一个页面文件。在这个例子中使用了 Struts 2 标签库提供的“property”标签用来显示 message 的属性值。

**【示例 8.1】**在使用 Struts 2 提供的标签库之前，需要在 JSP 中引入这个标签库，前缀定义为“s”。这个 helloWorld.jsp 文件的具体代码如图 8.18 所示。

```
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
	<head>
		<title>Hello World!</title>
	</head>
	<body>
		<h2>
			<s:property value="message" />
		</h2>
	</body>
</html>
```

使用property标签来获取Action中的属性

图 8.18　helloWorld.jsp 文件示例

### 8.3.5　编写 Action

Action 类是最基本的逻辑处理单元，在 MVC 模式中分发器分发给不同的 Action 类来处理请求。在 Struts 2 中 Action 类不必再实现 Action 接口，可以是任何类。但是一般还要继承 ActionSupport 类，因为其提供了大量的基本功能，如错误信息处理等。

**【示例 8.2】**我们来创建一个实现业务逻辑的 Action：HelloWorld.java，具体代码如图 8.19 所示。

```
package struts2;
import com.opensymphony.xwork2.ActionSupport;
public class HelloWorld extends ActionSupport {
    public static final String MESSAGE = "Hello World! I'm from Struts 2";
    public String execute() throws Exception {
        setMessage(MESSAGE);
        return SUCCESS;
    }
    private String message;
    public void setMessage(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
}
```

继承ActionSupport类

执行方法

设置属性的getter和setter方法

图 8.19　HelloWorld.java 示例

### 8.3.6　配置文件中增加映射

Struts 2 的配置文件是 struts.xml，所有请求和分发以及其他配置都在这个文件中定义，struts.xml 文件应该放在 WEB-INF 目录下的 classes 文件中。如示例 8.3 所示，配置了一个名称为 HelloWorld 的 Action，处理类是 struts2.HelloWorld，处理后的结果转到 helloWorld.jsp 页面上。

**【示例 8.3】**关于 Struts 2 框架的配置文件：struts.xml，具体配置方式如图 8.20 所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
          "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
   <constant name="struts.devMode" value="false" />
   <package name="struts2" extends="struts-default">
      <action name="HelloWorld"  class="struts2.HelloWorld">
        <result>/helloWorld.jsp</result>
      </action>
   </package>
</struts>
```

配置类HelloWorld，并设置转向页面

图 8.20　struts.xml 文件的配置

最后我们重新启动 Tomcat，并在浏览器中输入 http://localhost:8080/Struts2/HelloWorld.action 进行运行，系统就会出现如图 8.21 所示的运行结果。

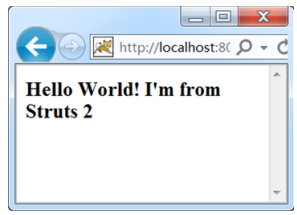



图 8.21　Struts 2 示例结果页面

## 8.4　小结

本章是 Struts 2 的入门章节，主要介绍了 Struts 2 的一些基础知识及下载安装过程，在章节时最后我们还通过一个 HelloWorld 程序展示了 Struts 2 框架程序的开发步骤。虽然本章内容不多，但是是后面所有 Struts 2 开发框架学习的基础。读者应多加练习，熟练掌握 Struts 2 程序的开发步骤和配置方法。

## 8.5　本章习题

1．请读者基于 Struts 2 开发一个用户注册模块。使用户在注册页面中填写用户信息，包括用户名、用户密码等信息。填写完成后提交注册表单，表单提交给 Strust 2 的业务控制器 Action，控制器处理提交的参数并决定跳转页面。页面跳转到用户信息显示页面，在该显示页面显示用户信息。执行结果如图 8.22 所示。

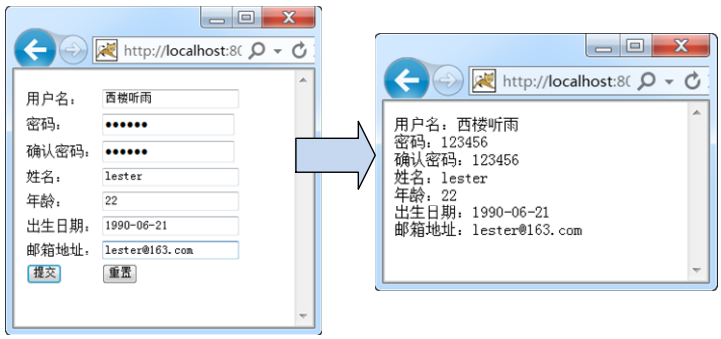



图 8.22　运行结果

【分析】要实现题目要求，我们可以定义如下的实现文件来实现用户注册模块。

表 8.1　用户注册模块需求的实现文件描述

| 文　件 | 功　能 |
|---|---|
| Register.jsp | 填写用户信息，提交用户表单信息 |
| RegisterAction.java | 接收表单参数，对参数进行处理，决定跳转页面 |
| ShowUserInfo.jsp | 显示用户的注册信息 |

【核心代码】本题所用到的各文件的核心代码如下所示。

Register.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
    <body>
        <form action="register.action" method="post">
            <table>
                <tr><td colspan="2"><s:actionerror/></td></tr>
                <tr><td>用户名：</td>
                    <td><input type="text" name="username"></td></tr>
……
```

```
……
                    <tr><td><input type="submit" value="提交"></td>
                        <td><input type="reset" value="重置"></td></tr>
                </table>
            </form>
        </body>
</html>
```

RegisterAction.java：

```
public class RegisterAction  extends ActionSupport{
    private String username;  //用户名信息
    ……
    public String getUsername() {  //获得用户名
        return username;
    }
    public void setUsername(String username) {//设置用户名
        this.username = username;
    }
    ……
……
    }
    public void validate() {
        if(upassword == null || "".equals(upassword)){
            this.addActionError("密码必须输入");
        ……

    }
    public String execute() throws Exception {//执行方法
        return "success";
    }
}
```

ShowUserInfo.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<html>
    ……
    <body>
        用户名：${username}<br>     <!-- 显示用户名信息 -->
        密码：${upassword}<br>      <!-- 显示密码信息 -->
        ……
    </body>
</html>
```

struts.xml：

```
<struts><!-- 根节点 -->
    <constant name="struts.i18n.encoding" value="gb2312"></constant>
    <package name="struts2" extends="struts-default">
        <!-- 定义 register 的 Action，其实现类为 action.RegisterAction-->
        <action name="register" class="action.RegisterAction">
            <!-- 定义处理结果与视图资源之间的关系-->
            <result name="success">/ShowUserInfo.jsp</result>
            <result name="input">/Register.jsp</result>
        </action>
    </package>
</struts>
```

# 第 9 章　Struts 2 配置详解

在 Struts 2 框架中，常用的配置文件包括 web.xml、struts.xml、struts.properties 和 struts-default.xml 等。其中 struts.xml 是 Struts 2 框架的核心配置文件，负责管理 Struts 2 框架的业务控制器 Action 和拦截器等。本章将详细介绍 Struts 2 应用中的各种配置文件，掌握这些配置文件后才能更好地使用和扩展 Struts 2 框架的功能。

## 9.1　Struts 2 配置文件

Struts 2 框架的配置文件可以分为两类：内部配置文件和开发人员使用的配置文件，其具体内容如图 9.1 所示。

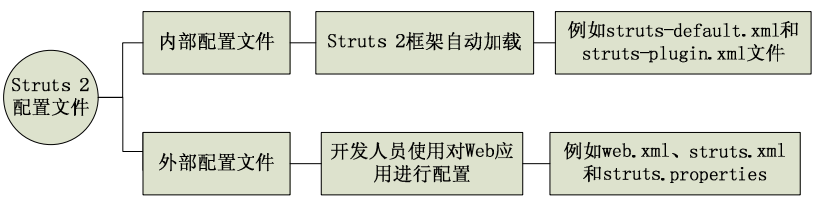



图 9.1　Struts 2 配置文件

Struts 2 框架的主要配置文件如表 9.1 所示。

表 9.1　Struts 2 框架的主要配置文件

| 配置文件 | 必选 | 位置（相对于 webapp） | 说　明 |
|---|---|---|---|
| web.xml | 是 | /WEB-INF/ | Web 部署描述文件，包括所有的必需框架组件 |
| struts.xml | 否 | /WEB-INF/classes | Struts 2 主要配置文件 |
| struts.properties | 否 | /WEB-INF/classes | Struts 2 框架的属性配置文件 |
| struts-default.xml | 否 | /WEB-INF/lib/struts2-core-x.x.x.jar | Struts 2 框架提供的默认配置 |
| struts-plugin.xml | 否 | /WEB-INF/lib/struts2-xxx-plugin.jar | Struts 2 框架的插件配置文件 |

这几个配置文件的相互关系如图 9.2 所示。

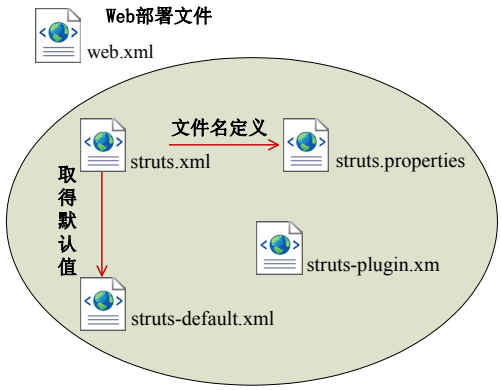



图 9.2　配置文件关系图

## 9.1.1　web.xml 文件

Web 应用都需要一个配置文件 web.xml，该文件用来对整个应用程序进行配置。在不同的 Web 框架中，web.xml 文件是不同的。在 Struts 2 框架中，web.xml 文件需要配置一个前端过滤器：StrutsPrepareAndExecuteFilter，用于对 Struts 2 框架进行初始化以及处理所有的请求。

**【示例 9.1】**下面是一个 web.xml 文件，在该文件中配置 StrutsPrepareAndExecuteFilter，文件的内容如图 9.3 所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app id="WebApp_9" version="2.4" 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">

  <display-name>Struts Blank</display-name>

  <filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-class>
  </filter>

  <filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>

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

</web-app>
```

配置Struts 2核心Filter的名称和实现类

使用配置的Fiter名称拦截所有的URL请求

默认访问的资源文件

图 9.3　web.xml 文件

其实，StrutsPrepareAndExecuteFilter 过滤器还可以进行进一步的配置，如图 9.4 所示。StrutsPrepareAndExecuteFilter 类的初始化参数有 3 个，具体描述如表 9.2 所示。

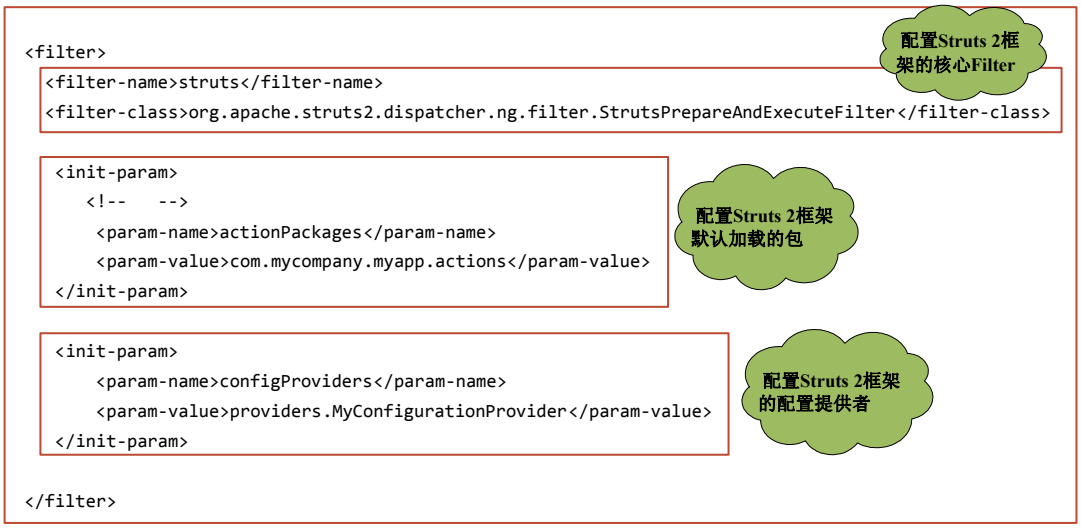



图 9.4　StrutsPrepareAndExecuteFilter 过滤器的配置

表 9.2　StrutsPrepareAndExecuteFilter类初始化参数

| 参数名称 | 参 数 描 述 |
|---|---|
| config | 该参数表示 Struts 2 框架自动加载的系列配置文件。如果有多个配置文件，中间用英文逗号（,）分隔 |
| actionPackages | 该参数表示 Struts 2 框架要扫描的包。如果有多个 ConfigurationProvider 类，中间用英文逗号（,）分隔 |
| configProviders | 该参数表示自定义的 ConfigurationProvider 类，用户可以提供一个或多个实现了 ConfigurationProvider 接口的类，并将这些类名设置成 configProviders 属性值。多个类名间用英文逗号（,）分隔 |

本书所使用的 Struts 2 为 struts-2.3.4.1，所以使用的过滤器为 StrutsPrepareAndExecuteFilter，读者可能会在其他书籍中发现有的书使用的是 org.apache.struts2.dispatcher.FilterDispatcher 过滤器，这是因为 FilterDispatcher 是 struts2.0.x 到 2.1.2 版本的核心过滤器，而 StrutsPrepareAndExecuteFilter 是自 2.1.3 版本后就替代 FilterDispatcher 了。由于 StrutsPrepareAndExecuteFilter 比 FilterDispatcher 功能更加齐全，我们建议读者使用新版本的过滤器。

## 9.1.2　struts.xml 文件

struts.xml 是 Struts 2 框架的核心配置文件。struts.xml 文件具有重要作用，因为应用中的所有常量、Action 和拦截器等几乎都配置在该文件中。

**【示例 9.2】**在默认情况下，Struts 2 会自动加载 WEB-INF\classes 目录中的 struts.xml 文件，struts.xml 文件的大体结构如图 9.5 所示。

struts.xml 配置文件中涉及到了 Struts 2 的 DTD（Document Type Definition，文档类型定义）信息。这些存在于 struts.xml 文件中的 DTD 是必须的。因为 Struts 2 在装载 struts.xml 文件时根据这些 DTD 信息来核对 struts.xml 文件中的标签设置是否合法。DTD 信息除了有核查功能外，MyEclipse 还会根据 DTD 信息自动列出当前标签允许设置的子标签以及标签的所有属性。

在我们下载的 struts-2.3.4.1 文件夹的 lib 文件夹中找到 struts2-core-2.3.4.1.jar，将其解压。我们就可以找到其中的 struts-2.3.dtd 文件，该文件就是 struts.xml 和 struts-default.xml 文件的 DTD。struts-2.3.dtd 文件的部分代码如图 9.6 所示，其他的限于篇幅读者可以自己理解。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
      "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
      "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
  <constant name="struts.enable.DynamicMethodInvocation" value="false" />
  <constant name="struts.devMode" value="false" />

  <package name="default" namespace="/" extends="struts-default">
    <default-action-ref name="index" />
    <global-results>
      <result name="error">/error.jsp</result>
    </global-results>
      <global-exception-mappings>
      <exception-mapping exception="java.lang.Exception" result="error"/>
    </global-exception-mappings>

    <action name="index">
      <result type="redirectAction">
         <param name="actionName">HelloWorld</param>
         <param name="namespace">/example</param>
      </result>
    </action>
  </package>

  <include file="example.xml"/>
  <!-- Add packages here -->
</struts>
```

配置常量

配置result

配置Action

图 9.5　struts.xml 文件

```
<!ELEMENT package (result-types?, interceptors?, default-interceptor-ref?, default-action-ref?,
default-class-ref?, global-results?, global-exception-mappings?, action*)>
<!ATTLIST package
    name CDATA #REQUIRED
    extends CDATA #IMPLIED
    namespace CDATA #IMPLIED
    abstract CDATA #IMPLIED
    strict-method-invocation CDATA #IMPLIED
    externalReferenceResolver NMTOKEN #IMPLIED
>
```

对package元素的定义

```
<!ELEMENT interceptors (interceptor|interceptor-stack)+>

<!ELEMENT interceptor (param*)>
<!ATTLIST interceptor
    name CDATA #REQUIRED
    class CDATA #REQUIRED
>
```

对 interceptor元素的定义

```
<!ELEMENT action ((param|result|interceptor-ref|exception-mapping)*,allowed-methods?)>
<!ATTLIST action
    name CDATA #REQUIRED
    class CDATA #IMPLIED
    method CDATA #IMPLIED
    converter CDATA #IMPLIED
>
```

对action元素的定义

```
<!ELEMENT result (#PCDATA|param)*>
<!ATTLIST result
    name CDATA #IMPLIED
    type CDATA #IMPLIED
>
```

对result元素的定义

图 9.6　DTD 文件示例

### 9.1.3　struts-default.xml 和 struts.properties 文件

struts-default.xml 文件是 Struts 2 框架的基础配置文件，为框架提供默认配置。struts-default.xml 文件是 struts2 框架默认加载的配置文件。它定义 struts2 一些核心的 bean 和拦截器。struts-default.xml 文件也是包含在 struts2-core-2.3.4.1.jar 文件中的。

Struts 2 框架除了 struts.xml 配置文件外，还有另外一个核心配置文件，这就是 struts.properties。struts.properties 文件用于配置 Struts 2 中所需的大量属性。

struts.properties 文件是一个标准的属性文件，该文件包含了大量的 key-value 对，每个 key 就是一个 Struts 2 属性，该 key 对应的 value 就是 Struts 2 的一个属性值，我们可以举一个例子如图 9.7 所示。

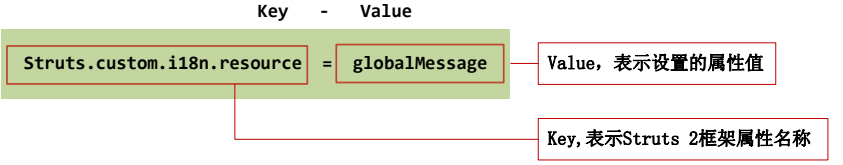



图 9.7　struts.properties 键值对形式

struts.properties 文件通常放在 Web 应用的 WEB-INF/classes 文件夹下，Struts 2 框架可以自动加载该文件。struts.properties 文件中常用属性及含义如表 9.3 所示。

表 9.3　struts.properties文件中常用属性

| 属 性 名 称 | 含　义 |
|---|---|
| struts.configuration | 指定加载 Struts 2 配置文件的配置管理器。默认值 org.apache.struts2.config.DefaultConfiguration |
| struts.locale | 指定了 Web 应用程序默认的 locate 和 encoding scheme，默认值是 en_US |
| struts.i18n.encoding | 指定了 Web 应用程序的默认编码集，正确地设置该属性可以解决客户端请求的中文编码问题 |
| struts.objectFactory | 指定了 Struts 2 默认的 ObjectFactory Bean，默认值是 spring |
| struts.custom.properties | 指定了 Struts 2 加载的用户自定义属性文件 |
| struts.velocity.configfile | 该属性指定了 Velocity 框架所使用的 velocity.properties 文件的位置 |
| struts.custom.i18n.resources | 该属性指定了 Struts 2 所使用的国际化文件，如果有多个资源文件，中间用逗号（,）分隔 |

**【示例 9.3】**除了在 struts.properties 文件中配置 Struts 2 属性外，还可以通过 struts.xml 配置文件的常量来配置 Struts 2 属性，比如我们可以举一个例子如图 9.8 所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
   "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
   "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
   <constant name="struts.i18n.encoding"  value="GBK"  />
</struts>
```

通过constant标签来配置Struts 2的属性

图 9.8　通过 struts.xml 配置 Struts 2 属性

**注意：**Struts 2 提供了两种设置 Struts 2 属性的方式：通过 struts.properties 以 key-value 方式来设置 Struts 2 属性，也可以在 struts.xml 文件中通过<constant>标签来设置 Struts 2 属性。

## 9.2 struts.xml 文件配置详解

在 Struts 2 框架的配置文件 struts.xml 文件中，可以将配置内容分为三大类，其中的每种元素可以包含不同的配置内容，如图 9.9 所示。

| 类别名称 | 包含的配置 |
| --- | --- |
| 管理元素 | Bean配置、常量配置、包配置、命名空间配置、包含配置 |
| 用户请求处理元素 | 拦截器配置、Action配置、Result配置 |
| 错误处理元素 | 异常配置 |

图 9.9　struts.xml 文件配置内容分类

### 9.2.1 Bean 配置

在 Struts 2 框架的 struts-default.xml 文件中，定义了大量的核心组件。这些组件不是直接以硬编码的形式写在代码中，而是以自己的依赖注入容器来装配。Struts 2 框架可以通过可配置的方式很方便地管理 Struts 2 的核心组件。

读者可以打开 struts2-core-2.3.4.1.jar 中的 struts-default.xml 文件，在这个文件中会看到大量的 Bean 定义，如图 9.10 所示的代码片段配置了 Struts 2 的类型检测器、上传文件处理器和模板引擎管理器。

```
<bean type="com.opensymphony.xwork2.util.ObjectTypeDeterminer" name="tiger"
class="com.opensymphony.xwork2.util.GenericsObjectTypeDeterminer"/>
<bean type="com.opensymphony.xwork2.util.ObjectTypeDeterminer" name="notiger"
class="com.opensymphony.xwork2.util.DefaultObjectTypeDeterminer"/>
<bean type="com.opensymphony.xwork2.util.ObjectTypeDeterminer" name="struts"
class="com.opensymphony.xwork2.util.DefaultObjectTypeDeterminer"/>
... ...
```

配置Struts 2的三个类型检测器Bean

```
<bean type="org.apache.struts2.dispatcher.multipart.MultiPartRequest" name="struts"
class="org.apache.struts2.dispatcher.multipart.JakartaMultiPartRequest" scope="default" optional="true"/
>
   <bean type="org.apache.struts2.dispatcher.multipart.MultiPartRequest" name="jakarta"
class="org.apache.struts2.dispatcher.multipart.JakartaMultiPartRequest" scope="default"
optional="true" />
... ...
```

配置Struts 2的两个文件上传处理器Bean

```
<bean type="org.apache.struts2.components.template.TemplateEngine" name="ftl"
class="org.apache.struts2.components.template.FreemarkerTemplateEngine" />
<bean type="org.apache.struts2.components.template.TemplateEngine" name="vm"
class="org.apache.struts2.components.template.VelocityTemplateEngine" />
<bean type="org.apache.struts2.components.template.TemplateEngine" name="jsp"
class="org.apache.struts2.components.template.JspTemplateEngine" />
```

配置Struts 2的三个模板引擎管理器Bean

图 9.10　struts-default.xml 文件代码片段

在 struts.xml 文件中定义的 Bean 主要有如下两个作用。

- Struts 2 创建该 Bean 的实例，并作为框架的内部对象使用。
- 对 Bean 包含的静态方法进行值注入。

**【示例 9.4】**如果用户需要加入自己的类作为 Struts 2 的内部对象使用，往往需要实现 Struts 2 提供的一些接口。如要加入一个新的文件上传处理器 Bean，可以使用如图 9.11 所示的配置代码。

```
<struts>
   <bean type="org.apache.struts2.dispatcher.multipart.MultiPartRequest" name="myUpload"
    class="com.company.UploadMultiPartRequest" scope="default" optional="true"/>
   ... ...
</struts>
```

配置Struts 2的三个模板引擎管理器Bean

图 9.11　Bean 配置示例

struts.xml 文件中<bean>元素的属性如表 9.4 所示。

表 9.4　<bean>元素的属性

| 属 性 名 称 | 必选 | 说　　明 |
|---|---|---|
| class | 是 | 指定 Bean 的类名 |
| name | 否 | 指定 Bean 实例的名字，对于相同类型的多个 Bean 来说，它们的 name 属性值必须唯一 |
| type | 否 | 指定 Bean 实现的接口 |
| scope | 否 | 指定 Bean 实例的作用域，这个属性值必须是 default、singleton、request、session 或 thread 其中之一 |
| optional | 否 | 指定 Bean 是否为一个可选 Bean |
| static | 否 | 指定 Bean 是否使用静态方法注入。当指定 type 属性时，该属性值不应该为 true |

## 9.2.2　常量配置

在 Struts.xml 文件中，通过<constant>元素配置常量，可以作为指定 Struts 2 属性的一种方式。而 Struts 2 的常量既可以在 struts.xml 文件中配置，也可以在 struts.properties 文件中配置，实际上也可以在其他的一些文件中实现配置。

Struts 2 框架将按如图 9.12 所示的搜索顺序加载 Struts 2 常量。

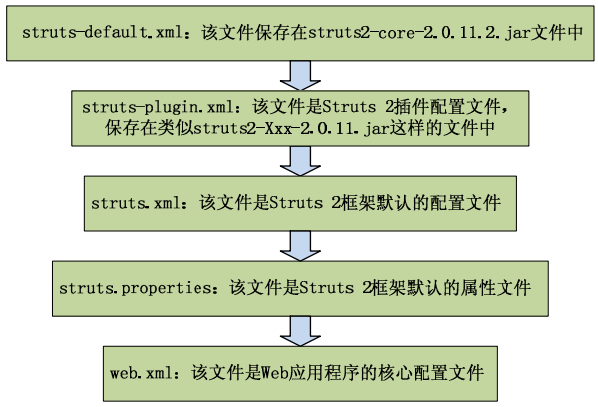



图 9.12　Struts 2 框架加载 Struts 2 常量的顺序

使用<costant>元素配置常量时，需要指定以下两个必填属性。

- name：该属性指定了常量名。
- value：该属性指定了常量值。

如果在 sturts.xml 中通过 devMode 属性来设置 Struts 2 的工作模式，可以按照如图 9.13 所示的代码来设置。

```
<struts>
   <constant name="struts.devMode" value="true" />
   ... ...
</struts>
```

设置了Struts 2的工作模式为开发模式

图 9.13　在 sturts.xml 中配置常量

在 struts.properties 文件中的属性和属性值是 key-value 对，其中 key 对应于 Struts 2 常量的 name，而 value 对应于 Struts 2 常量的 value，配置代码如图 9.14 所示。

```
struts.devMode = true
```

图 9.14　在 struts.properties 文件中配置常量

**【示例 9.5】**在 web.xml 文件中配置 Struts 2 常量，可通过<filter>标签的<init-param>子标签来指定，每个<init-param>标签配置了一个 Struts 2 常量。图 9.15 通过 web.xml 文件来配置了 devMode 属性。

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

   <filter>
       <filter-name>struts</filter-name>
       <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>

       <init-param>
          <param-name>struts.devMode</param-name>
          <param-value>true</param-value>
       </init-param>
   </filter>
   ... ...
</web-app>
```

指定了Struts 2的核心过滤器

通过init-param元素配置Struts 2常量

图 9.15　在 web.xml 文件中配置 Struts 2 常量

**注意：**在实际的开发中，最好不要在 struts.properties 和 web.xml 文件中配置常量，因为会使代码量明显增加。而 Struts 2 推荐在 struts.xml 中配置常量，便于集中管理。

## 9.2.3　包配置

在 Struts 2 框架中，其核心组件是 Action 和拦截器等，该框架使用包来管理这些组件。在

包中可以配置多个 Action、多个拦截器或者是多个拦截器引用的集合等。使用<package>元素配置包时，可以指定 4 个属性，如表 9.5 所示。

表 9.5　<package>元素的属性

| 属 性 名 | 必　选 | 说　明 |
|---|---|---|
| name | 是 | 该属性指定了包的名字。这个名字也是其他包引用的 key |
| extends | 否 | 该属性指定了该包继承的其他包的名字 |
| namespace | 否 | 该属性指定了包的命名空间 |
| abstract | 否 | 该属性指定当前包是否为一个抽象包。抽象包不能包含 Action |

**【示例 9.6】**例如，我们在 struts.xml 文件中配置两个包，配置方式如图 9.16 所示。

```
<struts>
    <package name="default" extends="struts-default">
        <interceptors>
            <interceptor-stack name="crudStack">
                <interceptor-ref name="checkbox" />
                <interceptor-ref name="static-params" />
            </interceptor-stack>
        </interceptors>
        <action name="showcase">
            <result>showcase.jsp</result>
        </action>
    </package>

    <package name="skill" extends="default" namespace="/skill">
        <!--  定义默认的拦截器引用  -->
        <default-interceptor-ref name="crudStack"/>
        <action name="list" class="org.company.action.SkillAction" method="list">
            <result>/manager/listSkills.jsp</result>
            <interceptor-ref name="basicStack"/>
        </action>
    </package>
</struts>
```

定义了一个拦截器栈

配置第一个包，该包名为default，这个包继承于struts-default

定义了一个Action

配置第二个包，该包名为skill，这个包继承于default

定义名为list的Action，将结果直接指向了listSkills.jsp

图 9.16　在 struts.xml 文件中配置包

## 9.2.4　命名空间配置

命名空间用来解决在同一个 Web 应用中 Action 重名的问题。如果在一个 Action 类中有多个业务处理方法，而客户端请求需要指向不同的方法，这时 Struts 2 以命名空间（Namespace）的方式来管理 Action。

**【示例 9.7】**图 9.17 是 struts.xml 中的配置代码片段，在这段代码中使用了<package>标签的 namespace 属性来指定当前包的命名空间。

在为包指定命名空间后，在访问包中的 Action 时就要在 Action 的 name 前加上命名空间的名字。如访问 edit 包中的 test 动作的 URL 如下：

```
http://localhost:8080/web/edit/test.action
```

如果将包 edit 的 namespace 属性去掉，则可以使用下面的 URL 来访问 test 动作：

```
http://localhost:8080/web/test.action
```


图 9.17　struts.xml 中的 namespace 属性

在使用命名空间时要注意以下三点：

- 命名空间名必须以斜线（/）开头，否则 Struts 2 不识别该命名空间。
- 根命名空间和默认命名空间是不同的。如使用 namespace="/"设置了根命名空间后，如果请求为/register.action，系统会先在根命名空间中查找 register 动作，如果 register 动作不存在，则会到默认命名空间中去查找这个 Action。
- Struts 2 不会对命名空间分层查找。也就是说，如果请求为/edit/dummy/test.action，当命名空间/edit/dummy 下没有 test 动作时，系统会直接到默认命名空间去查找 test 动作，而不会再到命名空间/edit 去查找 test 动作。

## 9.2.5　包含配置

**【示例 9.8】**在一个 JSP 文件中，可以使用<jsp:include>指令，经其他文件包含到该文件中。同样的道理，在 struts.xml 文件中，也可以使用包含元素<include>包含其他配置文件，具体代码如图 9.18 所示。

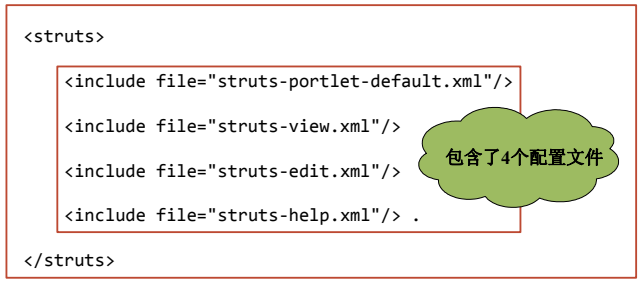



图 9.18　包含配置

**注意：**被包含的 struts-view.xml、struts-edit.xml 等配置文件必须是完整的 Struts 2 配置文件，也就是说，必须包含 DTD 信息、配置文件的<struts>标签等信息。一般情况下，将被包含的配置文件和 struts.xml 都放到 WEB-INF\classes 目录中。

### 9.2.6　拦截器配置

拦截器在前面已经多次提到了。拦截器的作用就是在执行 Action 处理用户请求之前或之后，执行拦截器来进行某些拦截操作。例如，用户请求删除 Action 时，拦截器判断用户是否具有删除权限，如果没有，则不执行删除操作。

**【示例 9.9】**Struts 2 为了使用拦截器更方便，还增加拦截器栈的概念（使用<interceptor-stack>标签配置）。实际上，这个拦截器栈就是使用不同的拦截器形成的一个拦截器组合。只要引用了拦截器栈，Struts 2 就会引用拦截器栈中的所有拦截器，图 9.19 是定义拦截器的代码片段示例。

```
<interceptors>
    <interceptor name="autowiring"
        class="com.opensymphony.xwork2.spring.interceptor.ActionAutowiringInterceptor"/>
    <interceptor name="chain" class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>
    <interceptor name="conversionError"
        class="org.apache.struts2.interceptor.StrutsConversionErrorInterceptor"/>

    <interceptor-stack name="validationWorkflowStack">
        <interceptor-ref name="basicStack"/>
        <interceptor-ref name="validation"/>
        <interceptor-ref name="workflow"/>
    </interceptor-stack>
</interceptors>
```

定义了三个拦截器

定义了一个拦截器栈

图 9.19　定义拦截器的代码片段

关于拦截器更详细的用法，我们会在后面的章节中为大家继续讲解。

## 9.3　配置 Action

Action 是 Struts 2 的核心。在本节将讲解 Action 的一些常用技术，如 Action 接口、Action 接口的默认实现类 ActionSupport、在 Action 中访问 Servlet API，处理多个提交动作和使用通配符等技术。通过本节的学习，读者可以基本了解 Action 中的常用技术和操作。

### 9.3.1　Action 实现类

Struts 2 中的 Action 其实就是一个普通的 Java 类，在该类中包含一个普通的 execute()方法，该方法没有任何参数，返回一个字符串类型的值。

**【示例 9.10】**但是这么简单的一个 Action 类，将如何获得 HTTP 请求参数呢？下面通过一个 Action 类来说明，代码如图 9.20 所示。

在 LoginAction.java 中，我们定义了 userName 属性，用于获取用户提交的表单中的参数值，然后我们可以生成 setXXX()和 getXXX()方法，分别用来设置和获取属性的值。最后 execute()方法返回一个逻辑字符串，例如 SUCCESS。

为了使 Action 类更加规范，Struts 2 中提供了一个 Action 接口。在这个接口中定义了一些

常用结果常量，而且在 Action 接口中还有一个 execute 方法。如果要实现 Action 接口，就必须要实现 execute 方法，Action 接口的代码如图 9.21 所示。

```
public class LoginAction extends ActionSupport{
        private String userName;

        public String getUserName() {
                return userName;
        }
        public void setUserName(String userName) {
                this.userName = userName;
        }

        public String execute() throws Exception{
                if(userName.equals("admin"));
                  return SUCCESS;
        }else{
                return INPUT;
        }
}
```

定义类和属性

userName属性的setXXX()和getXXX()方法

execute()方法内的业务处理

图 9.20　LoginAction.java 示例

```
package com.opensymphony.xwork2;
public interface Action
{
    public static final String SUCCESS = "success";
    public static final String NONE = "none";
    public static final String ERROR = "error";
    public static final String INPUT = "input";
    public static final String LOGIN = "login";

    public String execute() throws Exception;
}
```

在Action接口中定义的结果常量

定义处理用户请求的execute方法

图 9.21　Action 接口

在上述 Action 接口中，定义了 SUCCESS、NONE、ERROR、INPUT 和 LOGIN 这 5 个常量。这 5 个常量所表示内容如图 9.22 所示。

| 常量名称 | 表示内容 |
| --- | --- |
| SUCCESS | 表示动作执行成功，并应把相应的结果视图显示给用户 |
| NONE | 表示动作执行，但不应该把任何结果视图显示给用户 |
| ERROR | 表示动作执行不成功，并应把相应的报错视图显示给用户 |
| INPUT | 表示输入验证失败，并应把用户输入的表单重新显示给用户 |
| LOGIN | 表示动作没有执行，并应把登录视图显示给用户 |

图 9.22　Action 接口返回结果字符集

Action 类通常不会实现该接口，而是继承该接口的实现类 ActionSupport。下面是 Struts 2 中 ActionSupport 类的部分内容，如图 9.23 所示。

```
package com.opensymphony.xwork2;
public class ActionSupport implements Action, Validateable, ValidationAware,
TextProvider, LocaleProvider, Serializable
{
   ... ...
   public void setActionErrors(Collection errorMessages)
   {
       validationAware.setActionErrors(errorMessages);
   }

   public Collection getActionErrors()
   {
       return validationAware.getActionErrors();
   }

   public String getText(String aTextName, String defaultValue, String obj)
   {
       return textProvider.getText(aTextName, defaultValue, obj);
   }

   public boolean hasActionErrors() {
       return validationAware.hasActionErrors();
   }
   ... ...
}
```

设置校验错误的方法

获得校验错误的方法

返回国际化信息的方法

判断是否有Action错误的方法

图 9.23　ActionSupport 类

从图 9.23 可以看出，ActionSupport 实现了非常多的接口，如 Validateable、ValidationAware 等，并提供了这些接口中方法的默认实现。如果 Action 类从 ActionSupport 类继承，将会大大简化 Action 的开发。

## 9.3.2　间接访问 Servlet API

在 Struts 2 框架中，Action 与 Servlet API 相分离，这种低耦合性给开发者提供了便利。但是 Action 不访问 Servlet API，就不能实现业务逻辑处理。

Struts 2 框架认识到了这一点，于是提供了名称为 ActionContext 的类，在 Action 中可以通过该类获得 Servlet API。创建 ActionContext 类对象的语法格式如图 9.24 所示。

```
ActionContext actionContext = ActionContext.getContext();
```

getContext()方法

对象名

图 9.24　ActionContext 类对象语法格式

在 ActionContext 类中有一些常用的方法，如表 9.6 所示。

表 9.6　ActionContext类的常用方法

| 方 法 名 称 | 方 法 描 述 |
|---|---|
| Object get(Object key) | 通过参数 key 来查找当前 ActionContext 中的值，相当于 getAttribute(String name)方法 |
| Map getApplication() | 返回一个 Map 类型的 ServetContext 对象 |
| Map getParameters() | 返回一个包含所有 HttpServletRequest 参数信息的 Map 对象 |
| Map getSession() | 返回一个 Map 类型的 HttpSession 对象 |
| static ActionContext getContext() | 返回当前线程的 ServetContext 对象 |
| void put(Object key, Object value) | 向当前 ActionContext 对象中存入键值对信息 |
| void setSession(Map session) | 设置一个 Map 类型的 session 值 |

**【示例 9.11】**接下来我们举一个模拟添加图书的示例，使用上述 ActionContext 类和该类的一些方法，实现对 Servlet API 的访问。

首先我们在 MyEclipse 中创建 Web 应用 ch9，配置 Struts 2 开发环境，然后在 action 包中创建一个 Action 类 AddBook.java，该类实现对图书信息的处理，具体代码如图 9.25 所示。

```
package action;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;

public class AddBook extends ActionSupport {
    private String bookName;
    private double bookPrice;
    private String bookPress;

    public String getBookName() {
        return bookName;
    }
    public void setBookName(String bookName) {
        this.bookName = bookName;
    }
    public double getBookPrice() {
        return bookPrice;
    }
    public void setBookPrice(double bookPrice) {
        this.bookPrice = bookPrice;
    }
    public String getBookPress() {
        return bookPress;
    }
    public void setBookPress(String bookPress) {
        this.bookPress = bookPress;
    }

    public String execute() {
        ActionContext actionContext = ActionContext.getContext();
        actionContext.getSession().put("bookName",this.bookName);
        actionContext.getSession().put("bookPrice",this.bookPrice);
        actionContext.getSession().put("bookPress",this.bookPress);
        return SUCCESS;
    }
}
```

定义3个属性

属性的setXXX()和getXXX方法

execute()方法，获取输入值

图 9.25　AddBook.java 添加图书 Action

然后在 src 文件中新建 struts.xml 文件，在该文件中对 Action 进行配置。struts.xml 文件的内容如图 9.26 所示。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
        "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
        "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
    <constant name="struts.i18n.encoding" value="gb2312" />
    <package name="default" extends="struts-default">
        <action name="addBook" class="action.AddBook">
            <result name="success">/addSuccess.jsp</result>
        </action>
    </package>
</struts>
```

struts核心配置

图 9.26　Action 的配置文件 Struts.xml

然后我们创建一个图书添加页面 addBook.jsp，文件内容如图 9.27 所示。

```
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head>
    <title>添加图书</title>
  </head>
  <body>
   <center>
      <s:form action="addBook" method="post">
          <s:textfield name="bookName" label="书名"/>
          <s:textfield name="bookPrice" label="价格"/>
          <s:textfield name="bookPress" label="出版社"/>
          <s:submit value="提交"/>
      </s:form>
   </center>
  </body>
</html>
```

设置输入标签

图 9.27　添加图书页面 addBook.jsp

最后我们添加成功页面 addSuccess.jsp，将输入的内容显示出来，其具体代码如图 9.28 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <title>添加成功</title>
  </head>
  <body>
   读取session中的内容，您添加的图书信息为：
   <br><br>
   书名：<s:property value="#session.bookName"/> <br><br>
   价格：<s:property value="#session.bookPrice"/> <br><br>
   出版社：<s:property value="#session.bookPress"/>
  </body>
</html>
```

输出输入信息

图 9.28　添加成功页面 addSuccess.jsp

为了测试程序，我们在浏览器地址栏中输入 http://localhost:8080/ch9/addBook.jsp，显示运行结果，如图 9.29 所示。


图 9.29　运行结果显示界面

## 9.3.3　直接访问 Servlet API

在 9.3.2 小节中，通过使用 ActionContext 类，实现了 Action 间接访问 Servlet API。接下来我们将为大家介绍如何使用 Action 直接访问 Servlet API，直接访问方式可以分为 IoC 方式和非 IoC 方式。IoC（Inversion of Control，控制反转）就是将设计好的类交给系统去控制，而不是在自己的内部控制。

### 1．IoC 方式

在 Struts 2 框架中，通过 IoC 方式访问 Servlet API，就必须在 Action 中实现相应的接口。这些接口的名称和说明如表 9.7 所示。

表 9.7　访问IoC需要实现的接口名称及说明

| 接 口 名 称 | 说　　明 |
| --- | --- |
| ServletContextAware | 实现该接口的 Action 可以直接访问 ServletContext 对象。Action 必须实现该接口的 void setServletContext()方法 |
| ServletRequestAware | 实现该接口的 Action 可以直接访问 HttpServletRequest 对象。Action 必须实现该接口的 void setServletRequest()方法 |
| ServletResponseAware | 实现该接口的 Action 可以直接访问 HttpServletResponse 对象。Action 必须实现该接口的 void setServletResponse()方法 |
| SessionAware | 实现该接口的 Action 可以直接访问 HttpSession 对象。Action 必须实现该接口的 void setSession()方法 |

**【示例 9.12】**我们仍然以前面的添加图书为例，在 Action 类中使用 Ioc 方式实现 ServletRequestAware 接口，实现对 Servlet API 的访问，其具体实现方法如图 9.30 所示。

### 2．非 Ioc 方式

在非 Ioc 方式中，Struts 2 提供 ServletActionContext 类来获得 Servlet API。在 ServletActionContext 类中也提供了 getPageContext()、getRequest()、getResponse()等方法来实现直接访问功能。

```
package action;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpSession;
import org.apache.struts2.interceptor.ServletRequestAware;
import com.opensymphony.xwork2.ActionSupport;

public class IoCAction extends ActionSupport implements ServletRequestAware {
	private String bookName;
	private double bookPrice;
	private String bookPress;
	//省略bookName、bookPrice和bookPress属性的setXXX()和getXXX()方法

	private HttpServletRequest request;
	public void setServletRequest(HttpServletRequest request) {
		this.request = request;
	}

	public String execute() {
		HttpSession session = request.getSession();
		session.setAttribute("bookName",this.bookName);
		session.setAttribute("bookPrice",this.bookPrice);
		session.setAttribute("bookPress",this.bookPress);
		return SUCCESS;
	}
}
```

需要继承的接口

定义request对象

图 9.30　Ioc 方式示例 IoCAction.java

【示例 9.13】还是以添加图书为例，使用非 Ioc 方式时，Action 类的内容如图 9.31 所示。

```
import com.opensymphony.xwork2.ActionSupport;
public class NoIoCAction extends ActionSupport {
	private String bookName;
	private double bookPrice;
	private String bookPress;
	//省略属性的setXXX()和getXXX()方法
	public String execute() {
		HttpServletRequest request = ServletActionContext.getRequest();
		HttpSession session = request.getSession();
		session.setAttribute("bookName",this.bookName);
		session.setAttribute("bookPrice",this.bookPrice);
		session.setAttribute("bookPress",this.bookPress);
		return SUCCESS;
	}
}
```

实现getRequest()方法

图 9.31　非 Ioc 方式示例 NoIoCAction.java

**注意**：对于直接访问 Servlet API，我们推荐大家使用非 Ioc 方式，因为 Ioc 方式实现起来比较麻烦，并且 Servlet API 耦合大，而非 Ioc 方式实现简单，代码量少又能满足要求。

### 9.3.4 动态方法调用

在业务处理 Action 中，可以包含一个或者多个逻辑处理方法。例如，JSP 文件中的同一个 form 表单有多个用来提交表单的按钮，当用户通过不同的按钮提交表单时，需要调用 Action 中的不同处理方法，这时就可以使用动态方法调用。使用动态调用时，form 表单的 action 属性值必须符合如图 9.32 所示的格式。

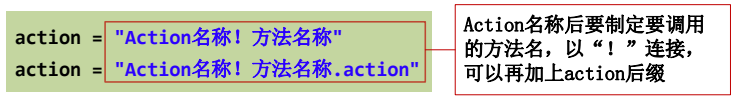



图 9.32 action 属性值设置

**【示例 9.14】**接下来我们在示例中实现动态方法的调用。

首先在 Web 应用 ch9 中创建 JSP 文件 loginRegister.jsp，在该文件的 form 表单中将定义两个按钮，loginRegister.jsp 文件的具体代码如图 9.33 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head>
    <title>登录或者注册</title>
  </head>
  <SCRIPT type="text/javascript">
  function reg(){
            targetForm = document.forms[0];
            targetForm.action = "loginRegister!register.action";
            //targetForm.action = "Register.action";
            targetForm.submit();
   }
  </SCRIPT>
  <body>
   <center>
   用户注册/登录
   <s:form action="loginRegister!execute.action" method="post" theme="simple">
   <ul><li>账号：<s:textfield name="userName"/>      </li>
            <li>密码：<s:password name="userPassword"/></li>
            <li><input type="button" value="注册" onclick="reg()"/>
            <s:submit value="登录"/>
            </li>
       </ul>
       </s:form>
   </center>
  </body>
</html>
```

设置表单属性，调用Action中的register()方法

图 9.33 loginRegister.jsp 文件

在包 action 中创建一个 Action 类 LoginRegister.java，该类的内容如图 9.34 所示。

然后在 struts.xml 文件中，对 LoginRegister.java 类进行配置，配置内容如图 9.35 所示。

接着我们就要创建一个 JSP 文件 success.jsp 来向页面输出信息了，主要内容如图 9.36 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
public class LoginRegister extends ActionSupport {
      private String userName;
      private String userPassword;
      private String tip;
      public String getUserName() {
            return userName;
      }
      public void setUserName(String userName) {
            this.userName = userName;
      }
      public String getUserPassword() {
            return userPassword;
      }
      public void setUserPassword(String userPassword) {
            this.userPassword = userPassword;
      }
      public String getTip() {
            return tip;
      }
      public void setTip(String tip) {
            this.tip = tip;
      }
      public String register() throws Exception{
       setTip("您单击了【注册】按钮！"+"您使用账号"+userName+"注册成功！");
       return SUCCESS;
      }
       public String execute() {
            setTip("您单击了【登录】按钮！"+"您使用账号"+userName+"登录成功！");
            return SUCCESS;
       }
}
```

定义属性，并生成对应的get和set方法

定义了register()和execute()方法

图 9.34　Action 类 LoginRegister.java

```
<package name="default" extends="struts-default">
                <action name="LoginRegister" class="action.LoginRegister" method="execute">
                        <result name="success">/success.jsp</result>
                </action>
</package>
```

图 9.35　配置 Action

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head>
    <title>登录或者注册</title>
  </head>
  <body>
根据您刚才选择的按钮，输出相应的信息：
<br>
<s:property value="tip"/>
  </body>
</html>
```

使用标签，输出信息

图 9.36　success.jsp 文件

我们在浏览器地址栏中输入 http://localhost:8080/ch9/loginRegister.jsp，显示运行结果，如图 9.37 所示。

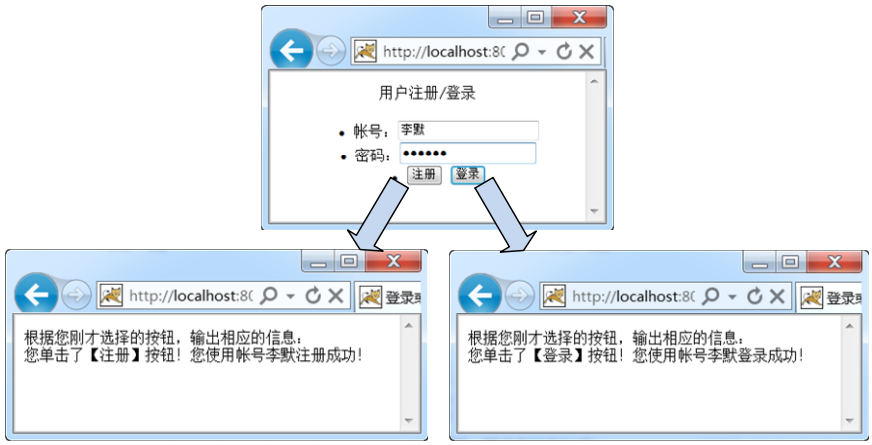



图 9.37　运行结果示意图

## 9.3.5　使用 method 属性和通配符映射

除了 name 属性和 class 属性以外，配置 Action 时还可以使用 method 属性，其基本格式如图 9.38 所示。

```
<action name="Action名称" class="包名.Action类名" method="方法名称">
        <result >视图URL</result>
</action>
```

图 9.38　使用 method 属性

**【示例 9.15】**通过使用 method 属性，也可以实现将不同的请求对应到不同的处理方法，例如，对于 9.3.4 小节中的注册/登录示例，我们可以通过改变 struts.xml 文件来实现相同的功能，修改后的代码如图 9.39 所示。

```
<action name="loginRegister" class="action.LoginRegister" method="execute">
                    <result name="success">/success.jsp</result>
</action>
<action name="Register" class="action.LoginRegister" method="register">
                    <result name="success">/success.jsp</result>
</action>
```

使用method分开配置

图 9.39　添加 method 属性后的 struts.xml 文件

这时我们运行程序，就可以得到与前面示例一样的效果。

在使用 method 属性时，由于在 Action 类中有多个业务逻辑处理方法，那么在配置 Action 时，就需要使用多个<action>元素。但是这样也会让 struts.xml 文件过于庞大而不易管理，这时我们就可以使用通配符映射。

**【示例 9.16】**首先我们来看一个例子，如图 9.40 所示，我们看通配符是如何使用的。

上述代码中定义了 3 个<action>元素，每个元素中的 name 属性值都不相同。当用户请求

某个 Action 时，Struts 2 框架将按照图 9.41 所示的规则检索 Action。

```
<package name="default" extends="struts-default">
        <action name="book_*" class="action.BookAction" >
                <result name="success">/success.jsp</result>
        </action>
        <action name="book_add" class="action.BookAction" >
                <result name="success">/success.jsp</result>
        </action>
        <action name="*" class="action.BookAction" >
                <result name="success">/success.jsp</result>
        </action>
</package>
```

图 9.40　通配符使用示例

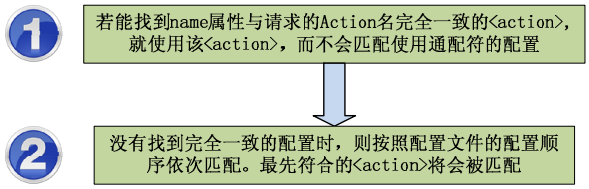



图 9.41　Struts 2 框架检索 Action 顺序

例如，如果用户请求 book_add.action，Struts 2 能够找到完全符合的<action>，就会使用这个<action>元素。如果用户请求 bookupdate.action，由于无法找到 name="bookupdate.action"的<action>，框架就会按照配置的先后顺序来匹配，最终选择<action name="*">，并进行匹配。

**注意：**在配置中使用通配符时，尽量将<action name="*">的形式放在最后，以防这种形式最先被匹配。

## 9.3.6　默认 Action

在 Struts 2 框架中，允许用户定义一个默认的 Action，当用户请求找不到对应的<action>元素值时，系统将调用默认 Action 来接收用户请求信息。这个 Action 也在 struts.xml 文件中配置，使用的配置元素为<default-action-ref>。

**【示例 9.17】**我们可以举一个默认 Action 配置的例子，如图 9.42 所示。

```
<package name="default" extends="struts-default">
<default-action-ref name="NoAction"/>
        <action name="addBook" class="action.BookAction" >
                <result name="success">/success.jsp</result>
        </action>
        <action name="NoAction" class="action.NoAction" >
                <result name="success">/ok.jsp</result>
        </action>
</package>
```

将NoAction指定为默认Action

配置NoAction

图 9.42　默认 Action 的配置

## 9.4 配置 Result

业务控制器 Action 负责处理用户请求，但它不能提供对用户的直接响应，当处理完请求信息后，需要根据 Result 结果配置，将 Action 的处理结果对应到相应的视图。本节就来详细介绍 Result 配置的相关知识。

### 9.4.1 结果映射

在前面配置 Action 时都使用了<result>元素，<result>元素的值可以是 JSP 页面文件，也可以是一个 Action 的 name 值。通过这些配置告诉 Struts 2 框架，在执行 Action 后，将响应一个 JSP 文件或者将执行另一个 Action。我们先来看一个最典型的 Result，如图 9.43 所示。

```
<action name="register" class="action.RegisterAction">
    <result name="success" type="dispatcher">/success.jsp</result>
</action>
```

通过type指定结果类型

图 9.43 典型的 Result 示例

<result>元素可以配置在<action>元素中，也可以配置在<action>元素外，根据这两种形式可以将 Result 的配置分为两类：局部 Result 和全局 Result。

#### 1. 局部 Result

**【示例 9.18】**局部 Result 定义<action>元素中，作用范围是这个<action>，这时<result>元素是<action>元素的子元素，具体实例如图 9.44 所示。

```
<package name="default" extends="struts-default">
        <action name="addBook" class="action.BookAction" >
                <result name="success">/success.jsp</result>
        </action>
</package>
```

局部Result

图 9.44 局部 Result

这个实例返回结果只对 addBook 这个 Action 起作用。

#### 2. 全局 Result

**【示例 9.19】**全局 Result 定义在<package>的<global-results>元素下，作用范围是这个包，这时<result>元素是<global-results>元素的子元素，具体实例如图 9.45 所示。

```
<package name="default" extends="struts-default">
    <global-results>
            <result name="error">/error.jsp</result>
    </global-results>
        <action name="addBook" class="action.BookAction" >
           <result>/success.jsp</result>
        </action>
</package>
```

全局Result

图 9.45 全局 Result

全局 Result，名称为 error。如果 default 包下的任何一个 Action 返回字符串 ERROR，那么都可以调用这个 result，页面将返回 error.jsp。

**注意：**当一个 Action 的局部 Result 与全局 Result 重名时，那么对于该 Action 的返回视图来说，局部 Result 会覆盖全局 Result。

在<result>配置的标准形式中，使用了 type 属性定义结果映射的类型。默认情况下表示使用 JSP 视图技术。Struts 2 默认支持的视图技术和 result 类型如表 9.8 所示。

表 9.8　Struts 2 默认支持的视图技术和result类型

| 类　　型 | 说　　明 |
| --- | --- |
| chain | 用于 Action 链式处理 |
| dispatcher | 用来整合 JSP，是<result>元素默认类型 |
| redirect | 用来重定向到其他文件 |
| velocity | 用来整合 velocity |
| xslt | 用来整合 XML/XSLT |
| redirectAction | 用来重定向到其他 Action |

下面我们对其中较为重要的几种类型进行讲解。

## 9.4.2　dispatcher 结果类型

dispatcher 结果类型用来表示“转发”到指定结果资源，它是 Struts 2 的默认结果类型。

接下来以添加联系人信息为例，介绍 dispatcher 类型的使用，认识 dispatcher 结果类型下的输出结果。

**【示例 9.20】**首先我们在 Web 应用 ch9 中，首先创建 JSP 文件 test.jsp，其具体代码如图 9.46 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head><title>添加联系人</title></head>
  <body>
   <center>
   添加联系人
      <s:form action="test" method="post">
            <s:textfield name="personName" label="姓名"/>
            <s:textfield name="personTelephone" label="电话"/>
            <s:textfield name="personAddress" label="地址"/>
            <s:submit value="提交"/>
      </s:form>
   </center>
  </body>
</html>
```

建立表单信息

图 9.46　输入信息页面 test.jsp

然后在 action 文件夹下创建 Action 类 Test.java，该类的内容如图 9.47 所示。

```
package action;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class Test extends ActionSupport {
        private String personName;
        private String personTelephone;
        private String personAddress;

    //省略属性的get和set方法

        public String execute() {
                return SUCCESS;
        }
}
```

定义属性及方法

execute()方法

图 9.47　Action 类 Test.java

在 struts.xml 文件中对上述 Action 类进行配置，配置内容如图 9.48 所示。

```
<package name="default" extends="struts-default">
        <action name="test" class="action.Test">
                <result type="dispatcher">/testSuccess.jsp</
result>
        </action>
</package>
```

图 9.48　配置 Action 类 struts.xml

然后创建一个返回结果的 JSP 页面 testSuccess.jsp，其主要内容如图 9.49 所示。

```
<html>
  <head><title>添加成功</title></head>
  <body>
  您添加的联系人信息为：
   <hr>
   联系人名称：<s:property value="personName"/> <br><br>
   联系人电话：<s:property value="personTelephone"/> <br><br>
   联系人地址：<s:property value="personAddress"/>
  </body>
</html>
```

输出页面信息

图 9.49　输出信息页面 testSuccess.jsp

运行程序，我们在浏览器地址栏中输入 http://localhost:8080/ch9/test.jsp，显示运行结果，如图 9.50 所示。

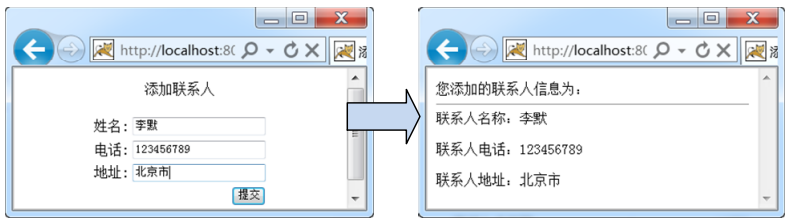



图 9.50　使用 dispatcher 结果类型

### 9.4.3　redirect 结果类型

redirect 结果类型用来“重定向”到指定的结果资源，该资源可以是 JSP 文件，也可以是 Action 类。

**【示例 9.21】**例如，我们把 9.4.3 小节中的 struts.xml 文件中的<result>元素使用 redirect 结果类型，如图 9.51 所示。

在<result>中使用 type="redirect"，表示当 Action 处理请求后，重新生成一个请求。我们重新运行 http://localhost:8080/ch9/test.jsp，就会出现图 9.52 所示的结果。

```
<package name="default" extends="struts-default">
        <action name="test" class="action.Test">
                <result type="redirect">/testSuccess.jsp</
result>
        </action>
</package>
```

图 9.51　使用 redirect 配置 struts.xml

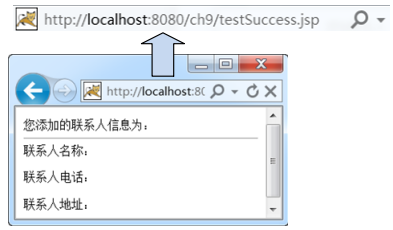



图 9.52　使用 redirect 结果类型

从图中地址可以发现，请求地址显示 testSuccess.jsp，而不是 test.action。使用 redirect 重定位到其他资源，原来的请求内容将全部丢失。

### 9.4.4　redirectAction 结果类型

redirectAction 结果类型和 redirect 结果类型相似，也是重定向到其他资源，重新生成一个新的请求。但是在 redirectAction 结果类型中，使用 ActionMapperFactory 类的 ActionMapper 实现重定向。使用 redirectAction 可以简化对带有命名空间的 Action 的访问。

**【示例 9.22】**例如，我们在 struts.xml 文件中使用 redirectAction 结果类型，具体代码如图 9.53 所示。

```
<package name="default" extends="struts-default">
        <action name="test" class="action.Test">
            <result type="redirectAction">
                    <param name="actionName">test2.action</param>
                    <param name="namespace">/temp</param>
            </result>
        </action>
</package>
<package name="secure" extends="struts-default" namespace="/temp">
          <action name="test2" class="action.Test">
                    <result name="success">/testSuccess.jsp</result>
          </action>
</package>
```

配置Action，指定返回结果类型

图 9.53　使用 redirectAction 结果类型

由于 redirectAction 结果类型表示重定向，所以和 redirect 结果类型一样，将会丢失请求信息。运行程序并进行同样的操作，运行效果如图 9.54 所示。

从图中地址可以发现，请求地址显示 test2.action，而不是 test.action 或者 testSuccess.jsp。使用 redirectAction 重定位到其他 action 请求资源，原来的请求内容将全部丢失。

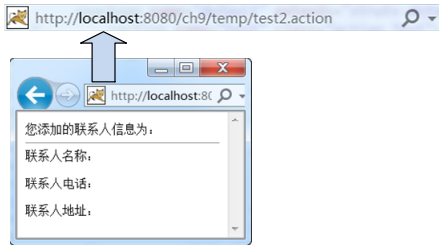



图 9.54　使用 redirectAction 结果类型

## 9.4.5　使用通配符动态配置 result

在配置 Action 时，可以使用通配符对 Action 进行动态映射。同样，对<result>元素也可以采用通配符进行动态配置。

**【示例 9.23】**具体的通配符示例如图 9.55 所示。

```
<package name="default" extends="struts-default">
    <action name="book_*" class="action.AddBook" method="{1}">
        <result name="success">/{1}.jsp</result>
    </action>
</package>
```

使用通配符

图 9.55　采用通配符动态配置 struts.xml

这里的<result>元素值不是固定的、唯一的，而是使用{1}.jsp 来表示，可以对应多个文件。如果客户端发送 book_add.action 请求，<result>元素的值将被设置为 add.jsp。如果发送 book_update.action 请求，<result>元素的值将被设置为 update.jsp。通过使用通配符，从而实现对请求结果的动态映射。

## 9.4.6　使用 OGNL 动态配置 result

在上面使用通配符动态配置 result 时，<result>元素值为{1}.jsp，这是根据 URL 参数来匹配的。如果根据 Action 中的属性名进行动态配置，那么就需要使用 OGNL 表达式。OGNL（bject-Graph Navigation Language）是一种功能强大的表达式语言，通过它简单一致的表达式语法，可以存取对象的任意属性，调用对象的方法，实现字段类型转化等功能。

**【示例 9.24】**例如，我们在 ch9 应用程序下创建一个 JSP 文件 OGNLTest.jsp 以及 Action 类 OGNLTest.java，OGNLTest.jsp 的具体代码如图 9.56 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head><title>使用OGNL动态配置result</title></head>
  <body>
   <center>添加联系人
      <s:form action="ognlTest" method="post">
            <s:textfield name="personName" label="姓名"/>
            <s:textfield name="personTelephone" label="电话"/>
            <s:textfield name="personAddress" label="地址"/>
            <s:submit value="提交"/>
      </s:form> </center>
  </body>
</html>
```

图 9.56　OGNLTest.jsp 示例

Action 类 OGNLTest.java 的具体代码如图 9.57 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
public class OGNLTest extends ActionSupport {
        private String personName;
        private String personTelephone;
        private String personAddress;
        //省略属性的get和set方法
        public String execute() {
         return SUCCESS;
        }
}
```

图 9.57　OGNLTest.java 示例

接下来在 struts.xml 文件中对 OGNLTest.java 这个 Action 类进行配置，配置方法如图 9.58 所示。

```
<action name="ognlTest" class="action.OGNLTest">
        <result name="success">/${personName}.jsp</result>
</action>
```

使用OGNL表达式

图 9.58　OGNLTest.java 的 struts.xml 文件配置

在这种配置中，<result>元素值使用 OGNL 表达式实现。用户输入的联系人姓名为 limo，那么这里的<result>元素值就为 limo.jsp；如果用户输入的联系人姓名为 lester，那么这里的<result>元素值就为 lester.jsp。所以为了正确显示页面，我们输入的用户名必须要与创建的 JSP 文件名保持一致。

现在我们创建一个 limo.jsp，具体代码内容如图 9.59 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head><title>添加成功</title></head>
  <body>
  下面将显示limo的信息：
   <hr>
  联系人名称：<s:property value="personName"/> <br><br>
  联系人电话：<s:property value="personTelephone"/> <br><br>
  联系人地址：<s:property value="personAddress"/>
  </body>
</html>
```

获取输入内容

图 9.59　limo.jsp 示例

我们在浏览器地址栏中输入 http://localhost:8080/ch9/OGNLTest.jsp.jsp，并在用户名中输入 limo，显示运行结果，如图 9.60 所示。

如果我们输入的不是 limo，就会生成错误页面，因为我们没有设置相应的响应页面，如图 9.61 所示。


图 9.60　正确的显示结果

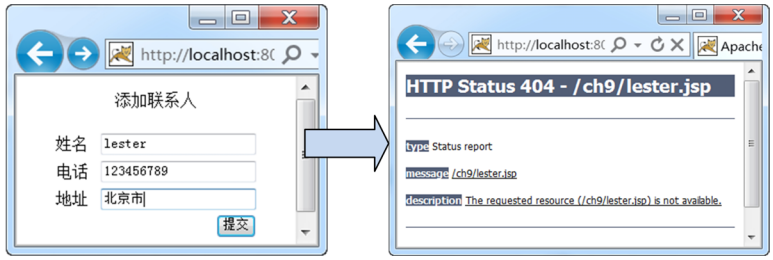



图 9.61　错误的显示结果

## 9.5　小结

本章是 Struts 2 的配置章节，是整个 Struts 2 框架最基础也是最重要的知识。只有正确地理解和熟练地掌握了本章所讲述的知识，我们才能在后面章节的具体应用中轻松地使用 Struts 2。本章的重点内容是 web.xml 和 struts.xml 文件的配置、Action 文件的配置和 Result 文件的配置。难点内容是能熟练掌握 Action 文件和 Result 文件在项目中的应用。希望大家一定要动手多加练习，熟练掌握本章所讲解的内容。

## 9.6　本章习题

1．新建一个火车票录入页面 trainTicketAdd.jsp，在该页面中录入所有的火车票信息，包括火车车次、起点站名称、终点站名称、车票价格、发车时间，执行结果如图 9.62 所示。

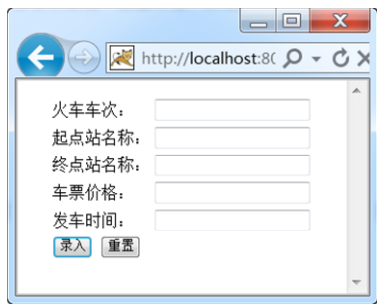



图 9.62　运行结果

【分析】本题主要考查读者建立 JSP 页面的基础能力。

【核心代码】本题的核心代码如下所示。

```
<%@page contentType="text/html;charset=gb2312"%>
<html>
……
<body>
<center>
    <form action="TrainTicketAdd.action" method="post">
    <table>
        <tr>
            <td>火车车次：</td>
            <td><input type="text" name="no"></td>
        </tr>
……
        <tr>
            <td colspan="2">
            <input type="submit" value="录入">
            <input type="reset" value="重置">
            </td>
        </tr>
    </table>
    </form>
</center>
</body>
</html>
```

2．新建一个火车票录入控制器 TrainTicketAddAction，该控制器接收 trainTicketAdd.jsp 中输入的火车票信息。并在 TrainTicketAddAction 控制器中调用 DAO 组件代码，将火车票信息添加到数据库中。

【分析】本题主要考查读者对于 Action 功能的配置能力。Action 是 Struts 2 的核心，读者应该了解 Action 中的常用技术和操作。

【核心代码】本题的核心代码如下所示。

```
package action;
import java.util.Date;
……
public class TrainTicketAddAction extends ActionSupport {
    private String no;
……
    public String getNo() {
        return no;
    }
    public void setNo(String no) {
        this.no = no;
    }
……
    public String execute() throws Exception {
        Train train = new Train();
        train.setNo(no);
        ……
        TrainDAO trainDAO = new TrainDAOImpl();
        trainDAO.addTrain(train);
        return SUCCESS;
    }
}
```

3．新建一个火车票查询页面 queryTrainTicket.jsp。在该页面中包含一个文本框，用来输入火车车次信息，如图 9.63 所示。然后再创建一个火车票查询控制器 TrainTicketQueryAction，在该控制器中接受 queryTrainTicket.jsp 中输入的火车车次信息，并通过调用 DAO 组件，实现查询该车次的火车票信息。

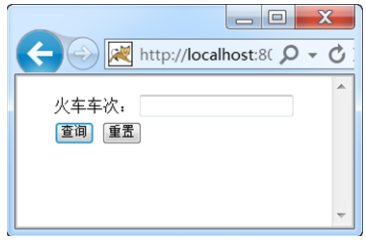



图 9.63　运行结果

【分析】本题主要考查读者对于 Action 以及 Result 功能的配置能力。业务控制器 Action 负责处理用户请求，但它不能提供对用户的直接响应，当处理完请求信息后，需要根据 Result 结果配置，将 Action 的处理结果对应到相应的视图。

【核心代码】本题的核心代码如下所示。

queryTrainTicket.jsp：

```
<%@page contentType="text/html;charset=gb2312"%>
<html>
……
<body>
<center>
     <form action="QueryTrainTicket.action" method="post">
     <table>
          <tr>
               <td>火车车次：</td>
               <td><input type="text" name="no"></td>
          </tr>
          <tr>
               <td colspan="2">
               <input type="submit" value="查询">
               <input type="reset" value="重置">
               </td>
          </tr>
     </table>
     </form>
</center>
</body>
</html>
```

TrainTicketQueryAction.java：

```
package action;
import java.util.Date;
……
public class TrainTicketQueryAction extends ActionSupport {
     private String no;
     public String getNo() {
          return no;
     }
     public void setNo(String no) {
          this.no = no;
     }
     public String execute() throws Exception {
          TrainDAO trainDAO = new TrainDAOImpl();
          Train train = trainDAO.findTrainByNo(no);
          ServletActionContext.getRequest().setAttribute("train", train);
          return SUCCESS;
     }
}
```

# 第 10 章　Struts 2 拦截器

拦截器（Interceptor）是 Struts 2 框架的核心组成部分。Struts 2 的很多功能都是构建在拦截器基础上的，如文件的上传下载、国际化、类型转换等。本章首先对拦截的实现原理和意义进行介绍，然后介绍 Struts 2 拦截器、如何自定义拦截器，最后通过一个简单的权限拦截器示例具体介绍一下拦截器的应用过程。

## 10.1　拦截器的实现原理

拦截器顾名思义就是能够拦截用户操作。拦截器的功能就是在进行一个操作（如调用方法）时，它会在用户执行操作前进行一系列操作，同样在用户操作完成后，进行一系列操作。

### 10.1.1　拦截器简介

到底什么是拦截器呢？我们可以通过一个实际例子来说明。比如现在你要乘坐电梯，在你进入电梯之前，电梯门首先要打开你才能进入。当你进入电梯后，电梯门会自动关闭。这时电梯就可以看做是一个拦截器，在你进入之前和之后都进行了一系列操作，如图 10.1 所示。

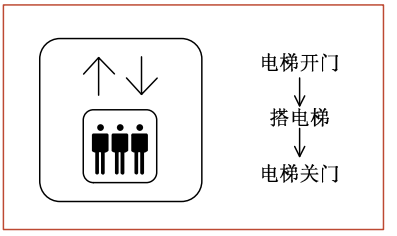



图 10.1　拦截器示意图

实际上，拦截器应用了软件开发中的一个重要思想——AOP（Aspect Oriented Programming，面向切面程序设计）。AOP 可以看做是 OOP（Object Oriented Programming，面向对象程序设计）的补充和完善，它可以将影响多个类的公共行为封装到一个可重用模块上。

### 10.1.2　拦截器实现原理

电梯是自动开门和关门的，那么拦截器是否也有这种功能呢？答案是肯定的。读者可能会

有疑问，方法可以在不调用的情况下自动执行吗？其实我们这里所指的自动执行是指通过代码驱动来实现方法的执行。

所谓的拦截其实就是动态地生成一个代理对象，这个对象中包含了拦截器方法的调用。当调用这个对象时，就会同时调用拦截器的方法，从而实现动态调用拦截器方法的目的。实现过程如图 10.2 所示。

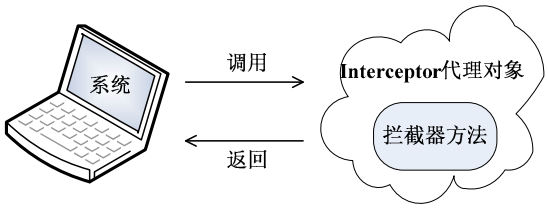



图 10.2　拦截器实现原理

## 10.2　Struts 2 拦截器

Struts 2 拦截器动态拦截 Action 调用的对象。它提供了一种机制，使开发者可以定义一个特定的功能模块，这个模块可以在 Action 执行之前或者之后运行，也可以在一个 Action 执行之前阻止 Action 执行。

### 10.2.1　Struts 2 拦截器原理

通常情况下，拦截器都是通过代理的方式调用。当请求到达 Struts 2 的 ServletDispatcher 时，Struts 2 会查找配置文件，并根据配置实例化相应的拦截器对象，然后将这些对象组成一个列表，最后逐个调用列表中的拦截器，如图 10.3 所示。

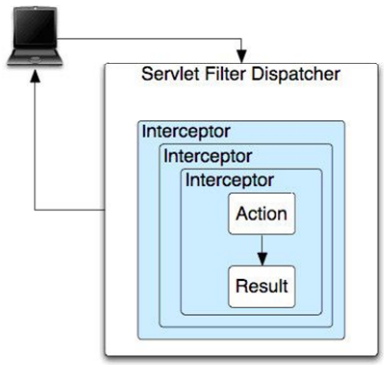



图 10.3　Struts 2 拦截器

每个 Action 请求都包装在一系列的拦截器内部。拦截器可以在 Action 执行之前做准备工作，也可以在 Action 之后做回收工作，拦截器的工作时序图如图 10.4 所示。

### 10.2.2　配置拦截器

在 struts.xml 文件中配置一个拦截器非常简单，只需要使用<interceptor>元素指定拦截器类

和拦截器名即可，配置拦截器的语法格式如图 10.5 所示。

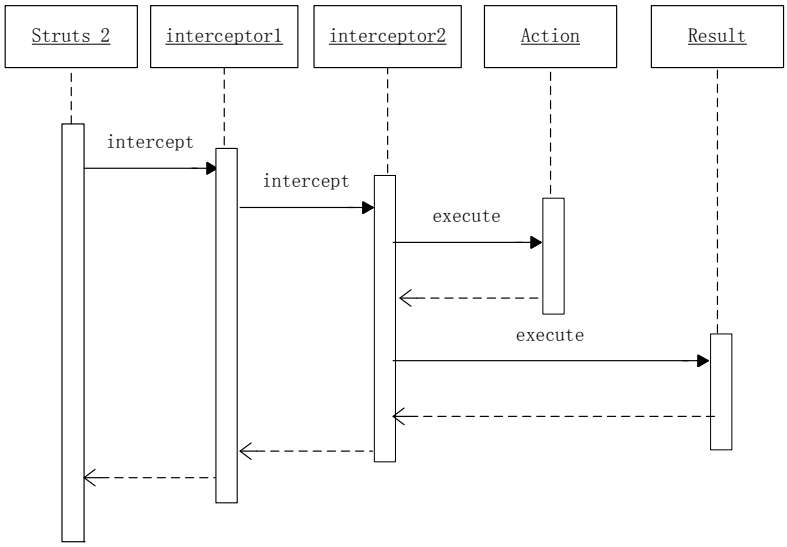



图 10.4　拦截器时序图

```
<interceptor name="拦截器名" class="拦截器对应的Java类" />
```

图 10.5　拦截器的配置语法

为了能在多个动作中方便地引用一个或多个拦截器，可以使用拦截器栈将这些拦截器作为一个整体来使用。当拦截器栈被配置到一个 Action 时，要想执行该 Action，必须先执行拦截器栈中的每一个拦截器，拦截器栈的语法格式如图 10.6 所示。

```
<interceptors>
        <interceptor-stack name="拦截器栈名">
                <interceptor-ref name="拦截器名1"/>
                <interceptor-ref name="拦截器名2" />
        </interceptor-stack>
</interceptors>
```

<interceptor-ref >元素指定一个拦截器

图 10.6　拦截器栈的语法格式

**注意：**在前面提到的 struts-default.xml 文件中，系统配置了许多拦截器栈，有兴趣的读者可以去查看。

**【示例 10.1】**拦截器栈的使用方法和使用拦截器一样，如图 10.7 所示。

一个拦截器栈被定义之后，就可以将这个拦截器栈当成一个普通的拦截器来使用，只是在功能上是多个拦截器的有机组合。

**【示例 10.2】**允许开发者将某个拦截器定义为默认拦截器。自定义拦截器需要使用<default-interceptor-ref>元素，配置后，该拦截器将成为所在包中的默认拦截器。

**【示例 10.3】**下面是一个自定义默认拦截器配置的示例，具体代码如图 10.8 所示。

```
<interceptors>
        <interceptor-stack name="mystack">
                <interceptor-ref name="timmer"/>
                <interceptor-ref name="logger"/>
        </interceptor-stack>
</interceptors>
<action name="first" class="action.FirstAction">
          <param name="who">limo</param>
          <interceptor-ref name="mystack"/>
</action>
```

定义拦截器栈

指定拦截器栈

图 10.7　拦截器的使用

```
<struts>
  <package name="testInterceptor"  extends="struts-default">
     <interceptors>
            <interceptor name="拦截器a" class="对应的Java类" />
     </interceptors>
     <default-interceptor-ref name="拦截器a" />
     <action name="success">
        <result name="success">/succsee.jsp</result>
     </action>
  </package>
</struts>
```

定义拦截器

将其设置为默认拦截器

图 10.8　配置默认拦截器

## 10.3　自定义拦截器

作为一个框架，可扩展性是不可缺少的优点之一。Struts 2 框架虽然提供了丰富的内置拦截器，但是仍然允许开发者自定义拦截器。本节就来介绍如何创建和使用自定义拦截器。

### 10.3.1　自定义拦截器类

一般来说，在定义一个拦截器类时，需要将该类实现 Interceptor 接口，或者继承抽象拦截器类 AbstractInterceptor。下面我们介绍如何通过继承 AbstractInterceptor 实现一个自定义拦截器类。

**【示例 10.4】** 我们创建一个拦截器实现类 MyInterceptor，实现对 AbstractInterceptor 的继承，代码如图 10.9 所示。

```
package interceptor;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
public class MyInterceptor extends AbstractInterceptor{
       private String userName;
       public void setUserName(String userName){
              this.userName=userName;
       }
       public String intercept(ActionInvocation ai) throws Exception {
              LoginAction action = (LoginAction)ai.getAction();
              System.out.println("执行拦截器...");
              String result=ai.invoke();
              System.out.println("拦截器执行完毕...");
              return result;
       }
}
```

拦截器执行方法

图 10.9　拦截器实现类

### 10.3.2　使用自定义拦截器

在实现自定义拦截器之后，如果要使用该拦截器，需要实现如图 10.10 所示的两个步骤。

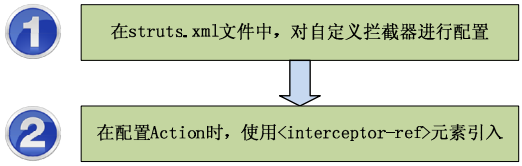



图 10.10　使用自定义拦截器的步骤

**【示例 10.5】**接下来通过一个示例介绍自定义拦截器的使用。例如，使用自定义拦截器实现文字过滤，对用户请求信息进行过滤，以“*”来代替需要过滤的文字。

首先我们在 Web 应用 ch10 中，创建一个 JSP 文件 register.jsp，文件的代码内容如图 10.11 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<HTML>
<HEAD><TITLE>欢迎注册XXX网站</TITLE></HEAD>
<center>
        <strong>注册</strong>
        <s:form action="reg" method="post">
        <s:textfield name="userName" label="注册姓名" "></s:textfield>
        <s:password name="userPassword" label="注册密码" "> </s:password>
        <s:textarea name="userInfo" label="注册说明" ></s:textarea>
        <s:submit value="提交"></s:submit>
        </s:form>
        </div>
</center>
</BODY>
</HTML>
```

注册表单

图 10.11　注册文件 register.jsp

然后在 action 包中定义用于处理请求的 Action 类 RegisterAction.java，在该 Action 中接收用户提交信息，具体代码如图 10.12 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
public class RegisterAction extends ActionSupport{
        private String userName;
        private String userPassword;
        private String userInfo;
        //省略属性的get和set方法

        public String execute(){
                return SUCCESS;
        }
}
```

execute()方法

图 10.12　处理请求 Action 类 RegisterAction.java

如果 Action 返回 SUCCESS 逻辑视图，那么页面将响应 myRegister.jsp 页面，并显示用户注册信息，主要代码如图 10.13 所示。

```
<strong>我的注册信息</strong><br>
        注册姓名：<s:property value="userName"/><br>
        注册密码：<s:property value="userPassword"/><br>
        注册说明：<s:property value="userInfo"/>
<br/><br/>
```

输出用户信息

图 10.13　显示页面 myRegister.jsp

运行程序，请求 register.jsp 文件，显示注册页面。在注册页面中输入注册信息，如图 10.14 所示。


图 10.14　填写注册信息

接下来，我们创建实现过滤文字的拦截器。在 interceptor 文件夹下创建拦截器类 RegisterInterceptor.java，其代码如图 10.15 所示。

```
package interceptor;
import action.RegisterAction;
import com.opensymphony.xwork2.Action;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
public class RegisterInterceptor extends AbstractInterceptor {
    public String intercept(ActionInvocation ai) throws Exception {
    Object object = ai.getAction();
    if (object != null) {
        if (object instanceof RegisterAction) {
            RegisterAction action = (RegisterAction) object;
            String userInfo = action.getUserInfo();
            if (userInfo.contains("程")) {
                userInfo = userInfo.replaceAll("程", "*");
                action.setUserInfo(userInfo);
            }
            return ai.invoke();
        } else {
            return Action.LOGIN;
        }
    } else {
        return Action.LOGIN;
    }
  }
}
```

获取用户注册信息

判断是否有要过滤的文字

图 10.15　拦截器类 RegisterInterceptor.java

最后在 struts.xml 文件中配置使用拦截器，配置方法如图 10.16 所示。

```
<package name="default" extends="struts-default">
        <interceptors>
                <interceptor name="replace" class="interceptor.RegisterInterceptor" />
        </interceptors>
        <action name="reg" class="action.RegisterAction">
                <result name="success">/myRegister.jsp</result>
                <result name="input">/register.jsp</result>
                <interceptor-ref name="defaultStack"/>
                <interceptor-ref name="replace"/>
        </action>
</package>
```

拦截器配置

配置响应页面及使用拦截器

图 10.16　拦截器的配置文件 struts.xml

我们重新运行程序，输入注册信息并提交，会发现显示页面中文字“程”被“*”所代替，如图 10.17 所示。


图 10.17　实现过滤文字

## 10.4　Struts 2 系统拦截器

Struts 2 框架的大部分工作都是通过系统拦截器来实现的，例如，解析请求参数、参数类型转换等。本节将为大家简要介绍系统拦截器的相关知识。

### 10.4.1　系统拦截器

Struts 2 提供了大量的系统拦截器，这些拦截器都是以键值对的形式配置在 struts-default.xml 文件中的。我们可以从中截取一部分代码如图 10.18 所示。

```
<interceptor name="alias" class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
<interceptor name="chain" class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>
<interceptor name="cookie" class="org.apache.struts2.interceptor.CookieInterceptor"/>
<interceptor name="clearSession" class="org.apache.struts2.interceptor.ClearSessionInterceptor"/>
<interceptor name="createSession" class="org.apache.struts2.interceptor.CreateSessionInterceptor"/>
<interceptor name="debugging" class="org.apache.struts2.interceptor.debugging.DebuggingInterceptor"/>
```

图 10.18　系统拦截器配置

如果开发者定义的 package 继承 Struts 2 框架的默认包，则可以自由使用上面定义的拦截器，否则只有自己来定义这些拦截器。

Struts 2 还利用这些拦截器组合了一系列的拦截器栈，我们截取一部分拦截器栈配置如图 10.19 所示。


图 10.19　拦截器栈配置

struts-default.xml 文件中包括了 Struts 2 应用所需要的大部分拦截器栈，很多时候只需要使用系统的默认拦截器栈 defaultStack 就能实现我们所期望的大部分功能。

## 10.4.2　timer 拦截器实例

在 Struts 2 系统拦截器中，timer 拦截器可以实现输出 Action 的执行时间，所以也可以称 timer 拦截器为耗时拦截器。

**【示例 10.6】**下面我们就来在示例中使用该拦截器，从而观察 Action 的执行时间。首先我们在 action 文件夹中，创建一个 TimerInterceptorAction.java 示例，在这个 Action 的 execute() 方法中，使线程休眠一段时间，来模拟程序运行消耗的时间，具体代码如图 10.20 所示。

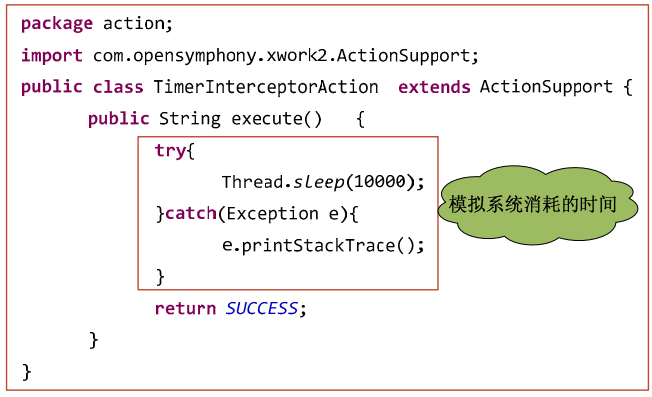



图 10.20　Action 类 TimerInterceptorAction.java

在 struts.xml 文件中配置上述 Action，并为 Action 指定 timer 拦截器，这样就可以在控制台看到 Action 执行时间的信息，配置内容如图 10.21 所示。

```
<action name="timerInterceptor" class="action.TimerInterceptorAction">
        <result name="success">/success.jsp</result>
        <interceptor-ref name="timer" />
</action>
```

图 10.21　耗时拦截器配置

运行程序，请求 timerInterceptor.action 时，控制台输出 1009 毫秒，因为第一次运行 Action 需要经历初始化阶段，该阶段也需要消耗一段时间。再次请求 Action 时，会发现执行时间明显缩短，如图 10.22 所示。

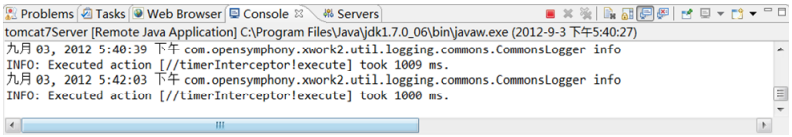



图 10.22　耗时拦截器休眠时间

## 10.5　权限拦截器实例

在实际应用中，如果用户没有登录系统，则该用户的某些操作将受到限制。如果用户强制进行操作，例如，在浏览器地址栏中请求登录后才可以访问 URL 地址，这时拦截器将执行拦截，重新返回到登录页面。接下来以此为例创建程序，实现用户的登录拦截。

### 10.5.1　权限拦截器

权限拦截器实际上就是一个实现权限控制的 Java 类。这个类只需要继承 Struts 2 框架提供的 AbstractInterceptor 类，并通过 intercept()方法，完成拦截器的权限检查功能。本章主要是说明拦截器的设计思路和实现过程，因此权限检查功能实现的比较简单。

首先是用户登录拦截器的实现类。在 interceptor 文件夹下，创建拦截器类 LoginInterceptor.java，如图 10.23 所示。

```
package interceptor;
import java.util.Map;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts2.StrutsStatics;
import com.opensymphony.xwork2.*;
public class LoginInterceptor extends AbstractInterceptor {
  public String intercept(ActionInvocation ai) throws Exception {
        ActionContext actionContext = ai.getInvocationContext();
        HttpServletRequest request =
          (HttpServletRequest)actionContext.get(StrutsStatics.HTTP_REQUEST);
        Map session = ai.getInvocationContext().getSession();
        String username = (String) session.get("user");
   if (username != null && username.length() > 0) {
        return ai.invoke();
   } else {
            ActionContext ac=ai.getInvocationContext();
            ac.put("popedom", "您还没有登录！");
            return Action.LOGIN;
   }
  }
}
```

获得session中的用户名

返回LOGIN字符串

图 10.23　登录拦截器类 LoginInterceptor.java

这个类主要是实现用户的登录拦截。如果 Session 中没有登录的用户名，则返回 Action.LOGIN；如果用户已经登录，则执行 return ai.invoke()，表示继续执行其他操作。

### 10.5.2 配置拦截器

在创建了登录拦截器后，我们还要在 struts.xml 文件中进行配置。

在 struts.xml 文件中要配置的内容有对拦截器的配置和对 Action 类的配置，如图 10.24 所示。

```
<package name="testInterceptor" extends="struts-default">
    <interceptors>
        <interceptor name="checkLogin" class="interceptor.LoginInterceptor"/>
    </interceptors>
        <action name="loginAction" class="action.LoginAction">
            <result name="success">/success.jsp</result>
            <result name="login">/login.jsp</result>
            <interceptor-ref name="defaultStack" />
            <interceptor-ref name="checkLogin" />
        </action>
</package>
```

配置Action

使用拦截器

图 10.24 拦截器配置

### 10.5.3 业务控制器 Action

上述配置文件中，已经指出该实例所需要的 Action 类为 LoginAction，该类继承 ActionSupport 类。在 action 文件夹下，创建 LoginAction.java 文件，代码如图 10.25 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport{
        public String execute(){
                return SUCCESS;
        }
}
```

实现execute()方法

图 10.25 Action 类 LoginAction.java

这个类比较简单，只是继承 ActionSupport 类，并实现了 execute()方法。

运行程序，在浏览器中输入 http://localhost:8080/ch10/loginAction.action。请求登录 Action，拦截器起作用，判断 session 中是否存在用户名信息。如果没有用户名信息就返回登录界面，如图 10.26 所示。


图 10.26 登录拦截

## 10.6　小结

本章主要讲述了 Struts 2 拦截器的相关知识。拦截器是 Struts 2 的核心组成部分，Struts 2 的大部分功能都是通过拦截器来完成的。本章的重点内容是理解拦截器的实现原理以及自定义拦截器的应用。难点是理解系统拦截器和权限拦截器的运用。希望读者多加练习，争取掌握。

## 10.7　本章习题

1．在一个完善的项目中，都要有用户权限控制，从而来区分不同用户的不同权限。例如，普通用户只能回复主题，高级用户可以发布主题等。我们请读者设置一个拦截器，来实现这种权限控制的功能，执行结果如图 10.27 所示。

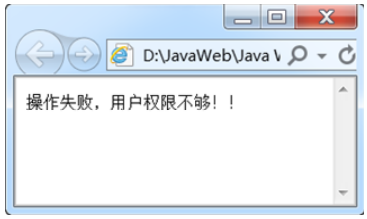



图 10.27　运行结果

【分析】要实现题目要求，我们可以定义如下的实现文件来实现用户权限控制的功能，如表 10.1 所示。

表 10.1　用户注册模块需求的实现文件描述

| 文　件 | 功　能 |
|---|---|
| UserPopedom.java | 从 session 中获得权限，对用户权限进行判断，负责页面跳转 |
| OperError.jsp | 操作失败页面，用来提示用户权限不够 |
| struts.xml | 配置拦截器，在 Action 中使用拦截器 |

【核心代码】本题所用到的各文件的核心代码如下所示。

UserPopedom.java：

```
package intercepter;
import java.util.Map;
……
public class UserPopedom extends AbstractInterceptor{
    public String intercept(ActionInvocation invocation) throws Exception {
        ActionContext ctx = invocation.getInvocationContext();
                                                    //取得 ActionContext 实例
        Map session = ctx.getSession();             //取得 session 对象
        String login = (String)session.get("popedom");//取得 session 中 popedom 属性值
        String  result = null;                                  //定义返回结果
        if(login != null && login.equals("admin")) {            //如果为管理员
            result = invocation.invoke();                       //进行下一步操作
        }
        else {                                                  //如果为非法用户
            result = Action.ERROR;                              //跳转到操作失败页面
        }
```

```
            return result;
        }
    }
```

OperError.jsp：

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<html>
……
<body>
    操作失败，用户权限不够!!
</body>
</html>
```

struts.xml：

```
……
    <constant name="struts.i18n.encoding" value="gb2312"></constant>
    <package name="struts2" extends="struts-default">
        <interceptors>
            <interceptor  name="userPopedom"  class="interceptor.UserPopedom">
</interceptor>
        </interceptors>
        <action name="register" class="action.RegisterAction">
            <result name="success">/ShowUserInfo.jsp</result>
            <result name="input">/Register.jsp</result>
            <result name="error">/OperError.jsp</result>
                <interceptor-ref name="userPopedom"></interceptor-ref>
                <interceptor-ref name="myInterceptorStack">
                    <param name="myInterceptor3.interceptorName">使用拦截器栈
</param>
                </interceptor-ref>
        </action>
    </package>
</struts>
```

# 第 11 章　Struts 2 类型转换和输入校验

类型转换和输入校验是 Struts 2 的一个非常重要的部分。通过类型转换能够将表单参数转换成 Java 类中的各种类型，而通过输入校验能够对表单参数进行合法性判断，从而过滤掉“不合法”数据。本章将详细介绍 Struts 2 的内建类型转换器和自定义类型转换器，还将介绍如何手动添加校验代码来完成校验和通过校验框架来完成校验。

## 11.1　Struts 2 类型转换基础

我们都知道在 Java 中有类型转换，同样在 Struts 2 中也有类型转换这个概念。不过它不是作用在 Java 语言内部，而是在表单参数和 Java 类中的数据中进行的。在所有的基于 Web 的 Java 开发框架中，Struts 2 拥有最优秀的类型转换能力。

### 11.1.1　为什么需要类型转换

在 Web 世界中输入/输出是没有数据类型的概念的，任何数据都被当做字符串或字符串数组来传递。而在 Web 应用的对象中，往往使用了多种不同的类型，如整数（int）、浮点数（float）、日期（Date）或者是自定义数据类型等。因此，在服务器端必须将字符串转换成合适的数据类型；服务器端完成处理后，又要将其他数据类型转换为 String，然后传递给客户端进行显示，具体流程如图 11.1 所示。

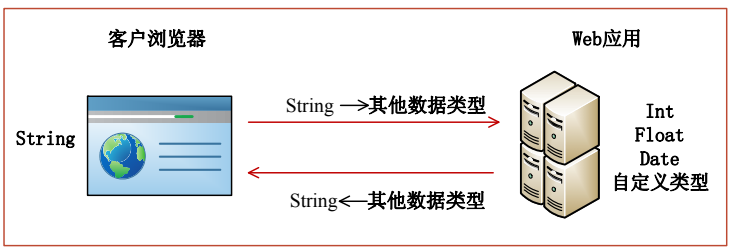



图 11.1　类型转换示意图

Struts 2 提供了强大的类型转换功能，对于常用的数据类型提供了内建的数据类型转换器。同样，开发者也可以自行设计类型转换器来进行类型转换。

### 11.1.2 自定义类型转换器

本小节我们先通过一个实现用户登录的自定义类型转换器例子来了解一下 Struts 2 的类型转换处理机制。

**【示例 11.1】**首先我们在 MyEclipse 开发工具中创建 Web 应用 ch11，配置好 Struts 2 开发环境。首先在 entity 包中创建一个 User.java，该类有两个属性：userName 和 userPassword，其具体代码如图 11.2 所示。

```
package entity;
public class User {
        private String userName;
        private String userPassword;
        public String getuserName(){
                return userName;
        }
        public void setuserName(String userName){
                this.userName=userName;
        }
        public String getuserPassword(){
                return userPassword;
        }
        public void setuserPassword(String userPassword){
                this.userPassword=userPassword;
        }
}
```

定义属性的getXXX()和setXXX()方法

图 11.2 User 类 User.java

然后我们在 action 包中创建一个 Action 类 UserAction.java。在这个 Action 中，我们将 userName 和 userPassword 属性封装到一个 User 对象中，UserAction.java 的具体代码如图 11.3 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
import entity.User;
public class UserAction extends ActionSupport{
        private String message;
        private User user;
        //省略属性的get和set方法

        public String execute(){
                if(getuser() !=null)
        message="您输入的信息为：用户名："+user.getuserName()
              +"，密码："+user.getuserPassword();
                        return SUCCESS;}
                else{
                        return INPUT;
                }
        }
}
```

定义属性的getXXX()和setXXX()方法

为message属性赋值

图 11.3 Action 类 UserAction.java

接着我们要在 struts.xml 文件中对 Action 进行配置，配置内容如图 11.4 所示。

```
<package name="default" extends="struts-default">
        <action name="userAction" class="action.UserAction">
                <result name="success">/loginSuccess.jsp</result>
                <result name="input">/login.jsp</result>
        </action>
</package>
```

图 11.4　配置 Action 在 struts.xml 文件中

如果 Action 返回字符串 SUCCESS，页面将响应 loginSuccess.jsp；如果 Action 返回字符串 INPUT，页面将响应 login.jsp。

为了实现类型转换，还需要用户建立类型转换器。转换器类需要继承 DefaultTypeConverter 类，并重写该类中的 convertValue()方法。我们在 util 文件夹下创建自定义转换器文件 UserConverter.java，该文件的内容如图 11.5 所示。

```
package util;
import java.util.Map;
import entity.User;
import ognl.DefaultTypeConverter;
public class UserConverter extends DefaultTypeConverter {
    public Object convertValue(Map context, Object value, Class toType){
      if (toType == User.class ){          从字符串类型转换为复合类型
            String[] params = (String[])value;
            User user = new User();
            String[] userValues = params[0].split(",");     将输入内容使用“,”分隔
            user.setUserName(userValues[0]);
            user.setUserPassword(userValues[1]);
            return user;
      }else if (toType == String.class ){          从复合类型转换为字符串类型
            User user = (User) value;
            return "<" + user.getUserName() + ","
            + user.getUserPassword() + ">";
       }
       return null ;
   }
}
```

图 11.5　自定义类型转换器类 UserConverter.java

UserConverter 继承自 DefaultTypeConverter，并重写了 convertValue()方法，并判断参数 toType 是 User 类型还是 String 类型，并执行不同的操作。但是如何让系统自动去完成两种类型的相互转换呢？答案是通过相应的配置文件，将这个转换类和 User 类联系起来。

在 Action 类所在目录（即 src/action）下创建资源文件 UserAction-conversion.properties，该文件的内容如图 11.6 所示。

最后我们只要创建两个视图资源文件就可以了，分别是用户登录页面 login.jsp 和登录成功页面 loginSuccess.jsp。其中用户登录页面 login.jsp 的具体代码如图 11.7 所示。

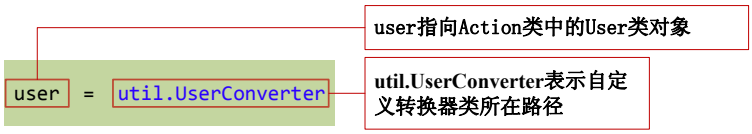



图 11.6　资源文件 UserAction-conversion.properties

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head><title>类型转换</title></head>
  <body>
  <s:form action="userAction" method="post" >
      <s:textfield name="user" label="登录"></s:textfield>
            <s:submit value="提交"></s:submit>
      </s:form>
  </body>
</html>
```

登录表单

图 11.7　用户登录页面 login.jsp

如果用户成功登录，页面将转向 loginSuccess.jsp，在该 JSP 文件中将用户输入的信息输出显示，该文件的具体代码如图 11.8 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="GB2312"%>
<%@ taglib prefix="s"  uri="/struts-tags"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head><title>类型转换</title></head>
 <body>
          <s:property value="message"/>
  </body>
</html>
```

输出message内容

图 11.8　登录成功页面 loginSuccess.jsp

运行程序，在浏览器地址栏中输入 http://localhost:8080/ch11/login.jsp，在页面的输入框中输入“lester,abc”并提交，显示运行结果，如图 11.9 所示。


图 11.9　运行结果示意图

**注意**：在页面中输入信息使用的分隔符必须与在 Action 中定义的分隔符完全一致，包括大小写和中英文状态等。

## 11.2　使用 Struts 2 的类型转换

将字符串请求参数转换为相应的数据类型，是所有 MVC 框架应该提供的功能。Struts 2 提供了相当丰富的类型转换功能，从而使开发者一般不再需要建立自定义类型转换器。

### 11.2.1　内建类型转换器

Struts 2 为常用的数据类型提供了内建的类型转换器，所以根本不需要使用自定义转换器。对于内建的转换器，Struts 2 在遇到这些类型时，会自动去调用相应的转换器进行类型转换。Struts 2 提供的主要内建类型转换器如表 11.1 所示。

表 11.1　Struts 2 内建类型转换器

| 数据类型及封装类 | 转 换 说 明 |
|---|---|
| boolean / Boolean | 完成字符串类型和布尔类型之间的转换 |
| char / Character | 完成字符串类型和布尔类型之间的转换 |
| int / Integer | 完成字符串类型和整数类型之间的转换 |
| float / Float | 完成字符串类型和单精度浮点类型之间的转换 |
| long / Long | 完成字符串类型和长整数类型之间的转换 |
| double / Double | 完成字符串类型和双精度浮点类型之间的转换 |
| Date | 完成字符串类型和日期类型之间的转换，日期格式使用当前请求的 Locate 的 SHORT 格式 |

**【示例 11.2】**我们通过一个实例来理解内建类型转换器的用法。例如，我们以用户注册为例，实现 Struts 2 类型转换。

首先我们创建一个用户注册文件 register.jsp，该 JSP 文件的内容如图 11.10 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="GB2312"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head><title>用户注册</title></head>
  <body>
  <center>
    <s:form action="registerAction" method="post">
      <s:textfield name="userName" label="姓名"/>
      <s:password name="userPassword" label="密码"/>
      <s:textfield name="userAge" label="年龄"/>
      <s:textfield name="userAddress" label="住址"/>
      <s:submit value="提交"/>
    </s:form>
    </center>
  </body>
</html>
```

用户提交表单内容

图 11.10　用户注册文件 register.jsp

在这个注册表单中包括了姓名、密码、年龄和住址等。然后我们要在 action 文件夹下创建一个 Action 类 RegisterAction.java，该文件的代码内容如图 11.11 所示。

```
package action;
import java.util.Date;
import com.opensymphony.xwork2.ActionSupport;
public class RegisterAction extends ActionSupport{
    private String userName;
    private String userPassword;
    private int userAge;
    private String userAddress;
    //省略属性的get和set方法

    public String execute(){
    return SUCCESS;
    }
}
```

定义属性及方法

execute()方法

图 11.11　Action 类 RegisterAction.java

在 struts.xml 文件中配置 RegisterAction.java，配置内容如图 11.12 所示。

```
<action name="registerAction" class="action.RegisterAction">
                <result name="success">/registerSuccess.jsp</result>
                <result name="input">/register.jsp</result>
</action>
```

图 11.12　配置 Action struts.xml

然后我们创建注册成功文件 registerSuccess.jsp，该文件的具体代码如图 11.13 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head> <title>用户注册</title> </head>
  <body >
    用户姓名：<s:property value="userName"/><br/>
    用户密码：<s:property value="userPassword"/><br/>
    用户年龄：<s:property value="userAge"/><br/>
    用户住址：<s:property value="userAddress"/><br/>
  </body>
</html>
```

将用户信息输出

图 11.13　注册成功文件 registerSuccess.jsp

运行程序，在浏览器地址栏中输入 http://localhost:8080/ch11/register.jsp，在页面的输入框中输入用户信息并提交，显示运行结果，如图 11.14 所示。

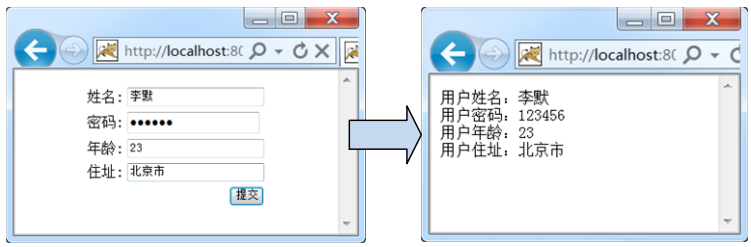



图 11.14　运行结果示意图

## 11.2.2　使用集合类型属性

在使用内建类型转换器实现类型转换时，将用户请求参数转换为复合类型的实例对象，但是只实现了添加一条注册信息的效果。如果需要实现同时处理多条注册数据，还需要使用集合类型来实现。

**【示例 11.3】**接下来我们在一个示例中使用 List 集合类型。首先我们要创建一个多注册文件 registerList.jsp，其具体代码如图 11.15 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="GB2312"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head><title>List用户注册</title></head>
  <body>
    <center>
    <s:fielderror></s:fielderror>
    <ul class="regul"><li class="regli">姓名</li><li class="regli">密码</li>
            <li class="regli">年龄</li><li class="regli">住址</li></ul>
   <s:form action="registerListAction" theme="simple">
    <s:iterator value="new int[2]" status="st">
            <ul class="regul">
            <li class="regli">
            <s:textfield name="%{'registers['+#st.index+'].userName'}"/>
            </li>
            <li class="regli">
            <s:password name="%{'registers['+#st.index+'].userPassword'}"/>
            </li>
            <li class="regli">
            <s:textfield name="%{'registers['+#st.index+'].userAge'}"/>
            </li>
            <li class="regli">
            <s:textfield name="%{'registers['+#st.index+'].userAddress'}"/>
            </li></ul>
      </s:iterator><br>
      <ul><li style="text-align: left;">
            <s:submit value="提交"></s:submit>
            </li></ul>
    </s:form>
   </center>
  </body>
</html>
```

需要注册的属性

使用iterator标签
循环输出注册信息

图 11.15　JSP 文件 registerList.jsp

接着在 action 文件夹中创建 Action 类 RegisterListAction.java，在该 Action 中将 Register 类封装在一个 List 对象 registers 中，具体代码如图 11.16 所示。

上述类中定义 List 集合属性 registers，并为该属性添加泛型为 Register 类型。我们在 struts.xml 文件中对这个 Action 类进行配置，配置代码如图 11.17 所示。

然后我们在 action 文件中创建资源文件 RegisterListAction-conversion.properties，该资源文件的具体内容如图 11.18 所示。

```
package action;
import java.util.List;
import com.opensymphony.xwork2.ActionSupport;
import entity.Register;
public class RegisterListAction extends ActionSupport{
        private List<Register> registers;
        public List<Register> getRegisters() {
                return registers;
        }
        public void setRegisters(List<Register> registers) {
                this.registers = registers;
        }
        public String execute(){
                return SUCCESS;
        }
}
```

register对象的属性和方法

图 11.16　Action 类 RegisterListAction.java

```
<action name="registerListAction" class="action.RegisterListAction">
        <result name="success">/registerListSuccess.jsp</result>
        <result name="input">/registerList.jsp</result>
</action>
```

图 11.17　配置 Action 在 struts.xml 文件中

```
Element_registers=entity.Register
```

图 11.18　资源文件 RegisterListAction-conversion.properties

最后我们只需要创建注册成功的输出文件 registerListSuccess.jsp，文件的内容是将 registers 这个集合中的数据输出，其具体代码如图 11.19 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="GB2312"%>
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
  <head><title>List用户注册</title></head>
  <body >
   <ul class="regul"><li class="regli">姓名</li><li class="regli">密码</li>
                <li class="regli">年龄</li><li class="regli">住址</li></ul>
    <s:iterator value="registers" status="st">
      <ul class="regul">
            <li class="regli">
                  <s:property value="userName"/></li>
            <li class="regli">
                  <s:property value="userPassword"/></li>
            <li class="regli">
                  <s:property value="userAge"/></li>
            <li class="regli">
                  <s:property value="userAddress"/></li>
      </ul>
    </s:iterator>
  </body>
</html>
```

使用iterator标签循环输出注册信息

图 11.19　注册成功输出文件 registerListSuccess.jsp

运行程序，在浏览器地址栏中输入 http://localhost:8080/ch11/registerList.jsp，在页面的输入框中输入用户信息并提交，显示运行结果，如图 11.20 所示。

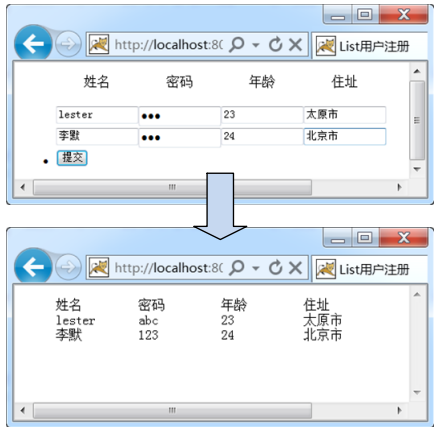



图 11.20　运行结果示意图

## 11.3　Struts 2 输入校验

输入校验是所有 Web 应用必须拥有的功能。因为 Web 应用的开放性导致了用户的多样性，因此，用户的输入信息不符合 Web 系统的要求的可能性非常大，所以就要求 Web 系统必须具有对用户输入信息的校验功能。Struts 2 在一切输入校验之前进行类型转换。类型转化成功后，即对请求信息进行数据校验。

### 11.3.1　使用 validate 方法完成输入校验

在 Struts 2 框架中，validate()方法是专门用来校验数据的。具体实现时，可以通过继承 ActionSupport 类，并重写 validate()方法来完成输入校验。Struts 2 框架执行 Action 类时，会在调用 execute()方法之前调用该 Action 类的 validate()方法。

**【示例 11.4】**接下来我们举一个实例，实现对添加的联系人信息进行输入校验。首先在应用中创建 JSP 文件 addUser.jsp，在该文件中创建添加联系人的 form 表单。addUser.jsp 的具体代码如图 11.21 所示。

```
<%@ page language="java" contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head><title>手动校验数据</title></head>
  <body>
<center>
<h4>添加联系人</h4><p/>
<s:fielderror />
<s:form action="addUser" method="post">
      <s:textfield label="联系人姓名" name="userName"></s:textfield>
      <s:textfield label="联系人电话" name="userTelephone"></s:textfield>
      <s:textfield label="联系人地址" name="userAddress"></s:textfield>
   <s:submit value="提交"/>
</s:form></center>
</body>
</html>
```

用户表单内容

图 11.21　添加联系人页面 addUser.jsp

然后在 action 文件夹下，创建一个 Action 类 AddUserAction.java，在该类中重写 ActionSupport 类的 validate()方法，实现对用户输入的数据进行校验，如图 11.22 所示。

```
package action;
import java.util.regex.Pattern;
import com.opensymphony.xwork2.ActionSupport;
public class AddUserAction extends ActionSupport {
      private String userName;
      private String userTelephone;
      private String userAddress;
      //省略属性的get和set方法
      public String execute() {
            return SUCCESS;
      }
      public void validate(){
            if(userName == null||getUserName().length()<4
                        ||getUserName().length()>20){
                  addFieldError(userName, "姓名的长度不符合要求!");
            }
            if(userTelephone == null||getUserTelephone().length()<7){
                  addFieldError(userTelephone, "电话的长度不符合要求!");
            }
            if(userAddress == null||getUserAddress().length()<4){
                  addFieldError(userAddress, "地址的长度不符合要求!");
            }
      }
}
```

数据校验 validate()方法

图 11.22　Action 类 AddUserAction.java

在 Action 中，重写 validate()方法，并将对用户提交信息的判断都放在该方法中，使得 execute()方法专注于逻辑处理。接下来对 struts.xml 文件进行配置，具体内容如图 11.23 所示。

```
<action name="addUser" class="action.AddUserAction">
        <result name="input">/addUser.jsp</result>
        <result name="success">/success.jsp</result>
 </action>
```

图 11.23　配置 Action 在 struts.xml 文件中

最后我们创建添加信息成功后的返回视图 success.jsp，success.jsp 文件的具体内容如图 11.24 所示。

```
<%@ page language="java" contentType="text/html;charset=gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head> <title>校验成功</title> </head>
 <body >
 <b>校验通过</b><br>
    联系人姓名：<s:property value="userName"/><br>
    联系人电话：<s:property value="userTelephone"/><br>
    联系人地址：<s:property value="userAddress"/><br>
 </body>
</html>
```

输出校验信息

图 11.24　添加信息返回视图 success.jsp

运行程序，在浏览器地址栏中输入 http://localhost:8080/ch11/addUser.jsp，在页面的输入框中输入联系人信息并提交，显示运行结果，如图 11.25 所示。


图 11.25　正确的输出结果示意图

如果我们输入的联系人信息不符合标准，就会返回输入页面并提示错误信息，如图 11.26 所示。

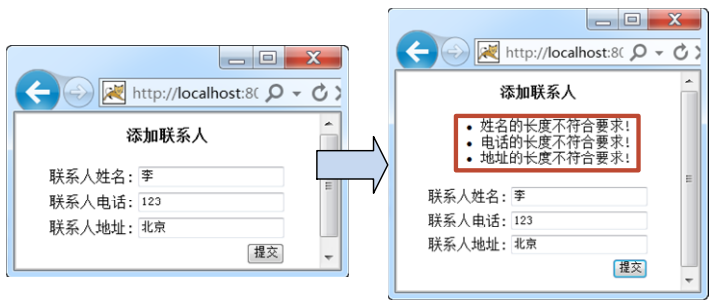



图 11.26　未通过校验示意图

使用 validate 方法完成输入校验的这个工作流程可以用图 11.27 来展示。

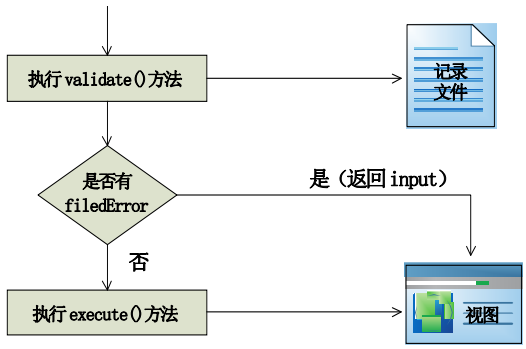



图 11.27　validate 方法校验流程

## 11.3.2　Struts 2 内置校验框架

Struts 2 框架提供了一套内置校验框架，可以实现应用中的大部分校验。通过校验框架能够非常简单和快速地完成输入校验。

解压 xwork-core-2.3.4.1.jar 文件，在 xwork-core-2.3.4.1\com\opensymphony\xwork2\validator\validators 目录下可以找到 default.xml 文件，该 XML 文件定义了 Struts 2 框架的内建校验器。我们可以截取一部分内容如图 11.28 所示。

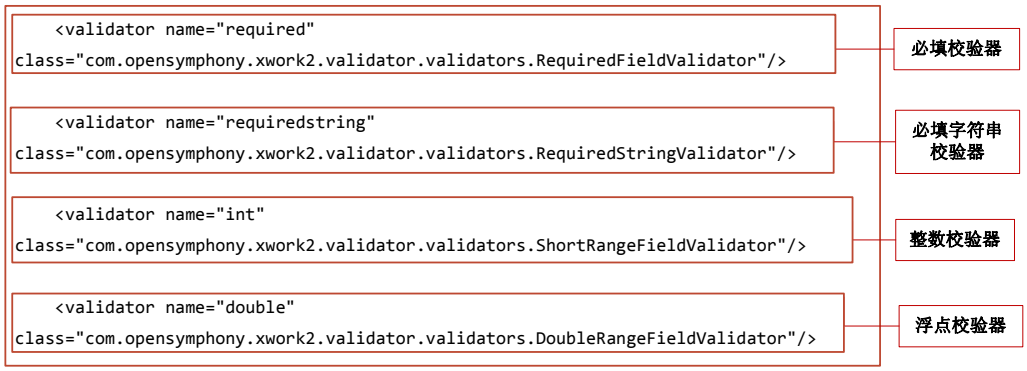



图 11.28　默认配置文件 default.xml

Struts 2 内建的校验器很多，最常用的有必填校验器、整数校验器、日期校验器和字符串长度校验器等。我们在这里就不一一为大家介绍了，只选取一个必填校验器作为示例来为大家讲解如何使用内建校验器。必填校验器一般用于校验必须输入的属性，要求属性必须有值（非null），其参数 fieldName 指定校验器的字段名称，如果是字段校验，则不用指定该参数。

**【示例 11.5】**例如在示例 11.4 中联系人姓名不可以为空，我们就可以为这个属性添加必填校验器。首先在 action 文件中创建一个 Action 类 Required.java，其代码内容如图 11.29 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
public class Required extends ActionSupport{
	private String userName;
	public void setUserName(String userName){
		this.userName=userName;
	}
	public String getUserName(){
		return this.userName;
	}
	public String execute(){
		return SUCCESS;
	}
}
```

图 11.29　Action 类 Required.java

这个 Action 只有一个属性 userName。我们只需要为它来配置必填校验器，即在 action 文件中创建一个 Required-validation.xml 文件，其代码内容如图 11.30 所示。

```
<?xml version="1.0" encoding="gb2312" ?>
<validators>
    <field name="userName">
        <field-validator type="required">
            <message>没有姓名信息，userName为null!</message>
        </field-validator>
    </field>
</validators>
```

设置必填校验器

图 11.30　配置文件 Required-validation.xml

当然我们还要在 struts.xml 文件中对 Action 类 Required.java 进行配置，配置方法如图 11.31 所示。

```
<action name="required" class="action.Required">
        <result name="input">/required.jsp</result>
        <result name="success">/success.jsp</result>
</action>
```

图 11.31　配置文件 struts.xml

我们还要创建一个 required.jsp 文件，这个文件的内容与前面的 addUser.jsp 完全相同，只是将表单提交到 required.java 这个 Action 进行处理。

运行程序，在浏览器地址栏中输入 http://localhost:8080/ch11/required.jsp，在页面的输入框中输入联系人信息并提交，显示运行结果，如图 11.32 所示。

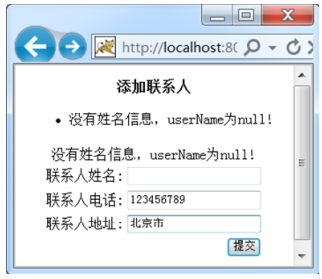



图 11.32　必填校验运行结果

## 11.4　小结

本章主要讲述了 Struts 2 框架关于类型转化和输入校验方面的知识。类型转化和输入校验是 Struts 2 框架的重要内容，集中体现了 Struts 2 框架的优势所在。本章的重点是掌握内建类型的转换器和使用内建校验类型框架对其进行校验。难点内容是理解集合类型属性的类型转换以及使用 validate 方法完成输入校验的方法。希望读者多加练习，争取掌握。

## 11.5　本章习题

1．新建一个类 Point3D，该类包含 3 个属性，分别为 x、y 和 z。自定义一个类型转换器，通过该类型转换器能够将表单中的 point3D 参数转换成 Point3D 类型的实例对象，执行效果如图 11.33 所示。

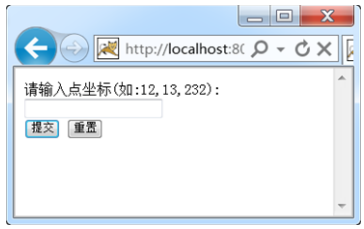



图 11.33　运行结果

【分析】本题主要考查读者对于类型转换的掌握程度。读者应熟悉 Struts 2 的类型转换处理机制。

【核心代码】本题的核心代码如下所示。

Point3D.java：

```
package bean;
public class Point3D {
    private int x;
……
    public int getX() {
        return x;
    }
    public void setX(int x) {
        this.x = x;
……
}
```

Point3DAction.java：

```
package action;
import com.opensymphony.xwork2.ActionSupport;
import bean.Point3D;
public class Point3DAction extends ActionSupport {
    private Point3D point3D;                //point 属性
    public Point3D getPoint3D() {           //获得 point 属性值
        return point3D;
    }
……
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

Point3DAction-conversion.properties：

```
point3D=converter.Point3DConverter
```

Point3DConverter.java：

```
package converter;
import java.util.Map;
……
public class Point3DConverter  extends StrutsTypeConverter{
    //从字符串类型转换成自定义数据类型
    public Object convertFromString(Map context, String[] values, Class toClass) {
        String[] pointArray = values[0].split(",");//通过逗号分隔字符串
        Point3D point = new Point3D();//新建一个 Point3D 实例
        point.setX(Integer.parseInt(pointArray[0]));//设置 x 坐标
        point.setY(Integer.parseInt(pointArray[1]));//设置 y 坐标
        point.setZ(Integer.parseInt(pointArray[2]));//设置 z 坐标
        return point;//返回转换后的 Point 实例
    }
    //从自定义数据类型转换成字符串类型
    public String convertToString(Map context, Object o) {
    ……
    }
}
```

Point3DInput.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
```

```
<html>
    <body>
        <form action="point3D.action" method="post">
            请输入点坐标(如:12,13,232): <br>
            <s:fielderror name="point3D"/>
            <input type="text" name="point3D"><br>
            <input type="submit" value="提交">
            <input type="reset" value="重置">
        </form>
    </body>
</html>
```

Point3DOutput.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
……
    <body>
        <s:property value="point3D"/>
    </body>
</html>
```

2．通过输入校验来检验用户的出生日期和其年龄是否统一，也就是说通过出生日期算出其年龄和用户输入的年龄是相同的。

【分析】本题主要考查读者对于类型转换的掌握程度。读者应熟悉 Struts 2 的类型转换处理机制。

【核心代码】本题的核心代码如下所示。

AgeAction.java：

```
package action;
import com.opensymphony.xwork2.ActionSupport;
public class AgeAction extends ActionSupport {
    private int age;          //年龄属性
……
    public void validate() {
        if(age < 0 || age > 100) {//判断年龄是否合法
            this.addActionError("请输入正确的年龄");//校验错误信息
        }
    }
    public String execute() throws Exception {
        return this.SUCCESS;
    }
}
```

AgeInput.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
    <body>
        <form action="age.action" method="post"><!--表单，提交到 age.action -->
            <s:actionerror/><!--输出 ActionError 错误信息 -->
            年龄：<input type="text" name="age" value="${param.age}"><br>
            <input type="submit" value="提交">
            <input type="reset" value="重置">
        </form>
    </body>
</html>
```

AgeOutput.jsp：

```
<%@page language="java" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
……
    <body>
        <s:property value="age"/>
    </body>
</html>
```

# 第12章　国际化和文件上传

随着网络的高速发展，大部分的 Web 站点已经走出国门，迈向世界了。而站点的访问者也不再仅仅是本地的浏览者，而是来自于全世界的访客，这时程序国际化成为了 Web 站点不可或缺的部分。而文件的上传也是 Web 应用中经常要使用的功能。基于 Struts 2 的 Web 应用使国际化开发和文件上传功能都变得非常简单易用，下面我们就来一起学习这两部分知识。

## 12.1　JSP 页面国际化

要实现 JSP 页面国际化，首先要添加并配置相应的国际化资源文件，然后在 JSP 页面中通过 Text 标签指定 name 参数为相应的 Key 值，从而实现国际化。JSP 页面国际化主要有两种方式，如图 12.1 所示。

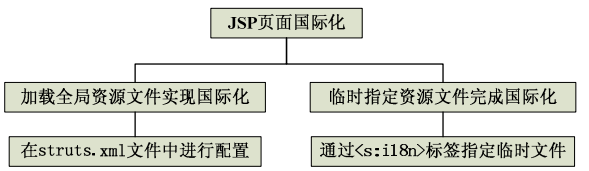



图 12.1　JSP 页面国际化

### 12.1.1　加载全局资源文件实现国际化

一般情况下，可以将国际化信息放到国际化资源文件中，然后在 struts.xml 文件中配置该文件为全局资源文件。这样就可以在 JSP 文件中很方便地访问到该资源文件，从而实现国际化。

**【示例 12.1】**下面我们举一个例子看如何在浏览器中实现页面内容的国际化。首先我们创建一个 Web 应用项目 ch12，在 src 根目录下添加两个资源文件 jspInterResource_zh_CN.properties 和 jspInterResource_en_US.properties，它们的代码内容如图 12.2 所示。

jspInterResource_zh_CN.properties:

```
welcomeMessage=\uFF0C\u4F60\u597D\uFF01
```

jspInterResource_en_US.properties:

```
welcomeMessage=Hello\!
```

图 12.2　全局资源文件

接着我们要在 struts.xml 文件中进行国际化配置，即在 struts 根节点下添加一个常量节点 constant 用来配置全局资源文件，具体代码如图 12.3 所示。

配置完国际化资源文件，就可以在 JSP 文件中通过 Text 标签来完成国际化了。我们创建一个 JSP 文件 welcomeMessage.jsp 来实现信息的输出，具体代码如图 12.4 所示。

运行程序，在浏览器中输入 http://localhost:8080/ch12/WelcomeMessage.jsp，打开欢迎页面如图 12.5 所示。

```
<struts>
<constant name="struts.custom.i18n.resources" value="jspInterResource"></constant>
</struts>
```

图 12.3　配置文件 struts.xml

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
  <head><title>欢迎信息</title></head>
  <body>
      <s:text name="welcomeMessage"></s:text>
  </body>
</html>
```

输出国际化资源文件

图 12.4　输出文件 welcomeMessage.jsp

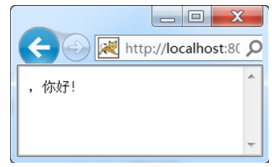



图 12.5　中文显示页面

因为我们使用的是中文操作系统，所以 IE 浏览器默认的语言环境为中文。我们可以通过 IE 属性配置来更改 IE 的默认语言环境。选择菜单栏的“工具”→“Internet 选项”配置窗口，将默认语言环境设置为英文，具体操作过程如图 12.6 所示。

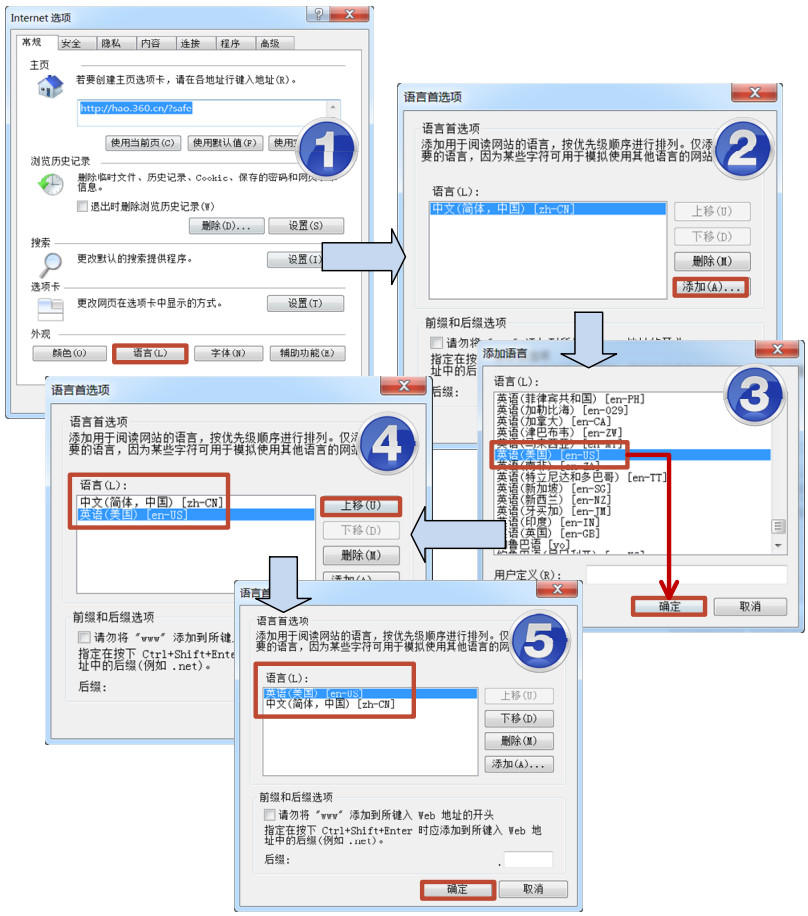



图 12.6　添加及设置英文页面

在将页面设置为英文后，我们刷新网页，页面就会发生变化，显示效果如图 12.7 所示。

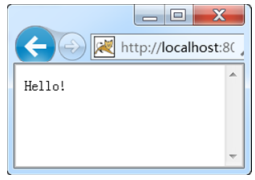



图 12.7　英文显示页面

## 12.1.2　临时指定资源文件完成国际化

前面示例中通过在 struts.xml 文件中进行配置全局资源文件，这样的资源文件在每个 JSP 文件中都可以使用。那如何才能使得资源文件不通过 struts.xml 配置，同样能被 JSP 页面调用呢？

我们可以在需要调用国际化资源文件的 JSP 页面中通过<s:i18n>标签来指定临时资源文件，这样就可以通过指定 key 值来找到获得指令的国际化信息了。

**【示例 12.2】**我们可以再定义一个 JSP 页面 WelcomeMessage2.jsp 使用<s:i18n>标签输出国际化信息，具体代码如图 12.8 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>欢迎信息</title></head>
<body>
    <s:i18n name="jspInterResource">
        <s:text name="welcomeMessage"></s:text>
    </s:i18n>
</body>
</html>
```

指定临时资源文件名

输出国际化资源文件

图 12.8　JSP 页面 WelcomeMessage2.jsp

打开 IE 浏览器，在浏览器地址栏中输入 http://localhost:8080/ch12/WelcomeMessage2.jsp，会发现页面效果与示例 12.1 相同，如图 12.9 所示。

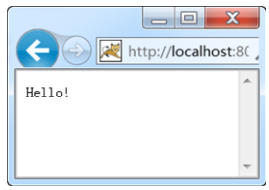



图 12.9　WelcomeMessage2.jsp 运行结果

**注意：**为了避免全局资源文件的影响，读者可以先把 struts.xml 文件中的配置全局资源文件的代码删除，这样就可以排除全局资源文件的影响了。

## 12.1.3　为资源文件传递参数

我们可以在 Text 标签中添加一个<s:param>标签来传递参数给资源文件。资源文件可以通

过占位符的形式来接收参数值。修改前面的资源文件，为资源文件添加占位符。

首先我们修改中文资源文件 jspInterResource_zh_CN.properties，其代码如图 12.10 所示。

```
welcomeMessage = {0}\uFF0C\u4F60\u597D\uFF01
```

设置占位符

图 12.10　修改中文资源文件 jspInterResource_zh_CN.properties

然后再对英文资源文件 jspInterResource_en_US.properties 进行修改，具体代码如图 12.11 所示。

```
welcomeMessage = Hello, {0} \!
```

设置占位符

图 12.11　修改英文资源文件 jspInterResource_en_US.properties

**【示例 12.3】**我们创建一个 WelcomeMessage3.jsp 文件为大家展示如何通过 Struts 2 中的 param 标签来传递参数，具体代码如图 12.12 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>欢迎信息</title></head>
<body>
    <s:i18n name="jspInterResource">
        <s:text name="welcomeMessage">
            <s:param>lester</s:param>
        </s:text>
    </s:i18n>
</body>
</html>
```

指定临时资源文件名

输出国际化资源文件

传递参数

图 12.12　JSP 页面 WelcomeMessage3.jsp

打开 IE 浏览器，在浏览器地址栏中输入 http://localhost:8080/ch12/WelcomeMessage3.jsp，显示运行结果如图 12.13 所示。

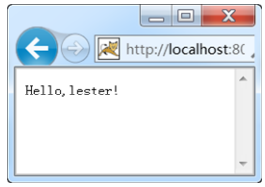



图 12.13　传递参数运行结果

## 12.2　Action 国际化

Struts 2 中的 Action 实现国际化的方式主要有 3 种，如图 12.14 所示。

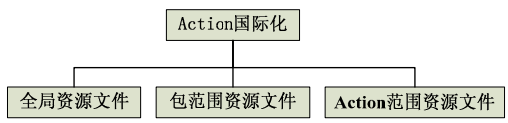



图 12.14　Action 国际化实现方式

下面我们就来介绍这 3 种加载方式，并对这 3 中加载方式的执行顺序加以说明。

## 12.2.1　加载全局资源文件完成国际化

Action 可以通过加载全局资源文件完成国际化。全局文件必须存放在项目的 WEB-INF/classes 目录下。我们首先在 src 文件夹中完成中文和英文资源文件 GlobalInterResource_zh_CN.properties 和 GlobalInterResource_en_US.properties 的添加工作，具体代码如图 12.15 所示。

GlobalInterResource_zh_CN.properties

```
ActionMessage=\u5168\u5C40\u8303\u56F4-\u6B22\u8FCE\u4FE1\u606F
```

GlobalInterResource_en_US.properties

```
ActionMessage=Global-WelcomeMessage
```

图 12.15　添加中文和英文资源文件

然后打开 struts.xml 配置文件，在 struts 根节点下添加一个常量节点 constant 用来配置全局资源文件，具体代码如图 12.16 所示。

```
<struts>
<constant name="struts.custom.i18n.resources" value="GlobalInterResource"></constant>
</struts>
```

图 12.16　配置文件 struts.xml

**【示例 12.4】**完成了相关的配置之后，我们就可以通过 ActionSupport 类中的 getText 方法获得全局资源文件的国际化信息了。我们创建一个名为 ShowActionMessage.java 的 Action 类，具体代码如图 12.17 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
public class ShowActionMessage extends ActionSupport {
        private String actionMessage;
        public String getActionMessage() {
                return actionMessage;
        }
        public void setActionMessage(String actionMessage) {
                this.actionMessage = actionMessage;
        }
        public String execute() throws Exception {
                actionMessage = this.getText("ActionMessage");
                return this.SUCCESS;
        }
}
```

取得国际化信息

图 12.17　Action 类 ShowActionMessage.java

添加完 Action，我们要在 struts.xml 文件中完成对其的配置，具体代码如图 12.18 所示。

```
<struts>
   <constant name="struts.custom.i18n.resources" value="GlobalInterResource"></constant>
   <package name="struts2" extends="struts-default">
         <action name="showActionMessage" class="action.ShowActionMessage">
         <result name="success">/ShowActionMessage.jsp</result>
         </action>
   </package>
</struts>
```

图 12.18　配置文件 struts.xml

最后我们只要再创建一个输出页面 ShowActionMessage.jsp 就可以了，其具体代码如图 12.19 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
   <head><title>国际化信息</title></head>
   <body>
      ${actionMessage}
   </body>
</html>
```

取得actionMessage属性值

图 12.19　页面输出文件 ShowActionMessage.jsp

打开 IE 浏览器，在浏览器地址栏中输入 http://localhost:8080/ch12/showActionMessage.action，分别在中英文页面上显示运行结果，如图 12.20 所示。

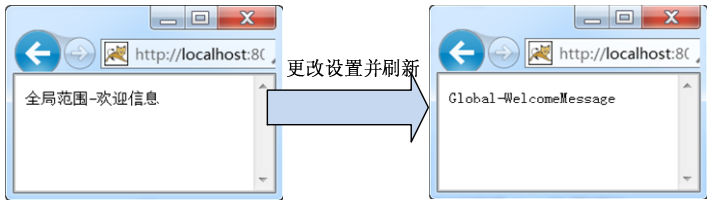



图 12.20　加载全局资源文件完成国际化

## 12.2.2　加载包范围资源文件完成国际化

下面我们来介绍包范围资源文件。包范围资源文件的好处在于不需要在 struts.xml 文件中配置。而且不同包下的 Action 使用不同的包文件，能够很好的将资源文件进行归类。包范围资源文件也可以只能被该包下的 Action 访问。

包范围资源文件的命名规则如图 12.21 所示。


图 12.21　包范围资源文件的命名规则

然后我们在 action 包中添加中英文的包范围资源文件，具体代码如图 12.22 所示。

package_zh_CN.properties

```
ActionMessage=\u5305\u8303\u56F4-\u6B22\u8FCE\u4FE1\u606F
```

package_en_US.properties

```
ActionMessage=Package-WelcomeMessage
```

图 12.22　添加中英文的包范围资源文件

打开 IE 浏览器，在浏览器地址栏中输入 http://localhost:8080/ch12/showActionMessage.action，分别在中英文页面上显示运行结果，如图 12.23 所示。

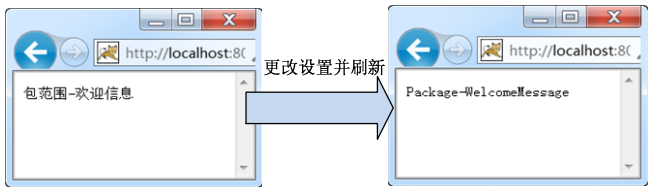



图 12.23　加载包范围属性

## 12.2.3　加载 Action 范围资源文件完成国际化

最后我们介绍 Action 范围资源文件，Action 范围资源文件同样不需要在 struts.xml 文件中配置。Action 范围资源文件只能被所对应的 Action 访问，其他 Action 无法访问。

Action 范围资源文件的命名规则如图 12.24 所示。


图 12.24　Action 范围资源文件的命名规则

Action 范围资源文件必须和对应的 Action 类保存在同目录下。我们在 action 包中添加中英文的 Action 范围资源文件，具体代码如图 12.25 所示。

ShowActionMessage_zh_CN.properties

```
ActionMessage=Action\u8303\u56F4-\u6B22\u8FCE\u4FE1\u606F
```

ShowActionMessage_en_US.properties

```
ActionMessage=Action-WelcomeMessage
```

图 12.25　添加中英文的 Action 范围资源文件

打开 IE 浏览器，在浏览器地址栏中输入 http://localhost:8080/ch12/showActionMessage.action，分别在中英文页面上显示运行结果，如图 12.26 所示。

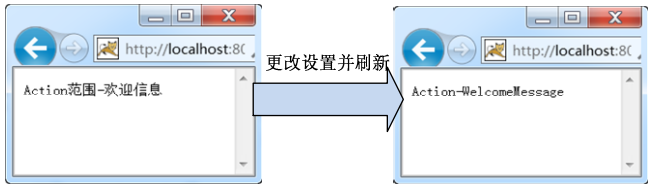



图 12.26　加载 Action 范围属性

### 12.2.4 资源文件加载顺序

下面我们对这 3 种资源文件的加载顺序进行说明，如图 12.27 所示。

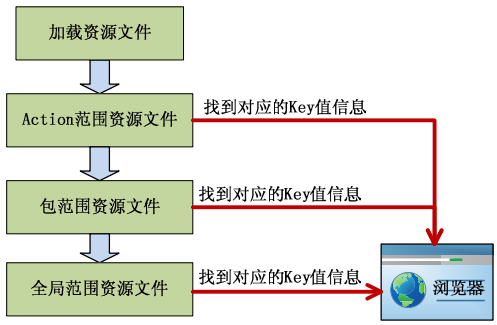



图 12.27 资源文件的加载顺序

## 12.3 基于 Struts 2 完成文件上传

Struts 2 框架中并没有提供文件上传，而是通过 Common-FileUpload 框架或 COS 框架来实现。Struts 2 在原有上传框架的基础上进行了进一步的封装，从而大大简化了文件上传的开发应用。

### 12.3.1 下载并安装 Common-FileUpload 框架

Common-FileUpload 框架是一个非常出色的上传框架，由 Apache 开源组织的 jakarta 项目组负责组织维护和更新。为了下载 Common-FileUpload 框架，我们首先要访问 Apache 官方站点 http://www.apache.org/，在屏幕下方找到“Commons”链接，并在其众多子项目中找到 FileUpload 和 IO 两个子项目，如图 12.28 所示。

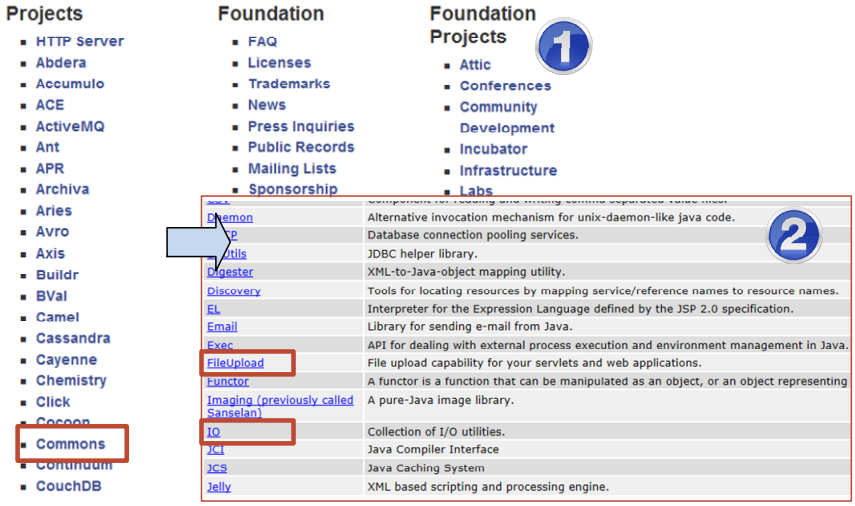



图 12.28 Commons 项目下载

FileUpload 和 IO 两个子项目的下载过程如图 12.29 所示。

下载完成后，得到两个压缩文件包，分别为 commons-fileupload-1.2.2-bin.zip 和 commons-io-2.4-bin.zip。要安装 Common-FileUpload 框架，只需将 commons-fileupload-1.2.2-bin\

bin 路径下的 commons-fileupload-1.2.2.jar 库和 commons-io-2.4-bin 目录下的 commons-io-2.4.jar 文件复制到 Web 应用中的 WEB-INF\lib 目录中就可以了。

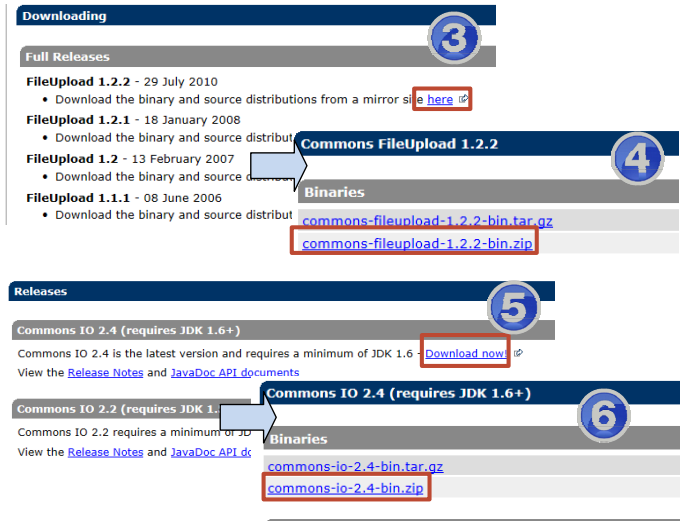



图 12.29　FileUpload 和 IO 两个子项目的下载

## 12.3.2　实现文件上传控制器

要实现文件上传，需要修改设置表单的 enctype 属性。默认情况下，这个值为 application/x-www-form-urlencoded，这时只能用来提交普通文本，不能用于文件上传；只有设置为 multipart/form-data，才能完整地传递文件数据并完成文件上传操作。

**【示例 12.5】**例如我们举一个例子使用 Action 类完成设置文件上传属性的功能，这个 Action 类 FileUploadAction.java 的具体代码如图 12.30 所示。

```
package action;
import java.io.*;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class FileUploadAction extends ActionSupport {
      private File uploadFile;
      private String uploadFileContentType;
      private String uploadFileFileName;
      //省略属性的get和set方法

      public String execute() throws Exception {
       InputStream is = new FileInputStream(uploadFile);
       String uploadPath = ServletActionContext.getServletContext()
                           .getRealPath("/upload");
       File toFile = new File(uploadPath, this.getUploadFileFileName());
       OutputStream os = new FileOutputStream(toFile);
       byte[] buffer = new byte[1024];
       int length = 0;
       while ((length = is.read(buffer)) > 0) {
            os.write(buffer, 0, length);
       }
       is.close();
       os.close();
       return SUCCESS;
      }
}
```

设置上传文件的属性

设置上传文件的目录

图 12.30　Action 类 FileUploadAction.java

### 12.3.3 完成文件上传

文件上传控制器创建完成后，需要在 struts.xml 文件中进行配置，在配置之前首先要填上上传表单页 fileUploadPage.jsp 和上传结果页 fileUploadResultPage.jsp。

上传表单页包含一个文件域和两个按钮，其表单提交到上传文件控制器，代码如图 12.31 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>文件上传</title></head>
<body>
      <form action="fileUpload.action" method="post"
                  enctype="multipart/form-data">
            上传文件:<input type="file" name="uploadFile"><br>
            <input type="submit" value="上传">
            <input type="reset">
      </form>
</body>
</html>
```

上传文件设置

图 12.31　上传表单页 fileUploadPage.jsp

上传结果页，用来显示文件上传结果，包括上传文件的名称以及类型，代码如图 12.32 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>文件上传成功</title></head>
<body>
      上传文件名：${uploadFileFileName}<br>
      上传文件类型：${uploadFileContentType}
</body>
</html>
```

取得上传结果属性

图 12.32　上传结果页 fileUploadResultPage.jsp

然后我们对 struts.xml 文件进行配置，配置方法如图 12.33 所示。

```
<action name="fileUpload" class="action.FileUploadAction">
        <result name="success">/fileUploadResultPage.jsp</result>
        <result name="input">/fileUploadPage.jsp</result>
</action>
```

图 12.33　配置文件 struts.xml

配置完成后，就可以开始上传文件了。打开 IE 浏览器，在浏览器地址栏中输入 http://localhost:8080/ch12/fileUploadPage.jsp，显示页面如图 12.34 所示。

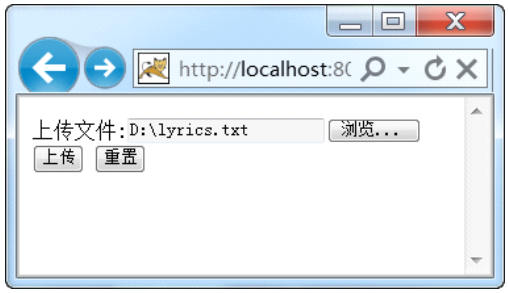



图 12.34　上传表单

单击上传按钮完成上传，我们就可以在页面和上传目录中查看文件信息了，如图 12.35 所示。

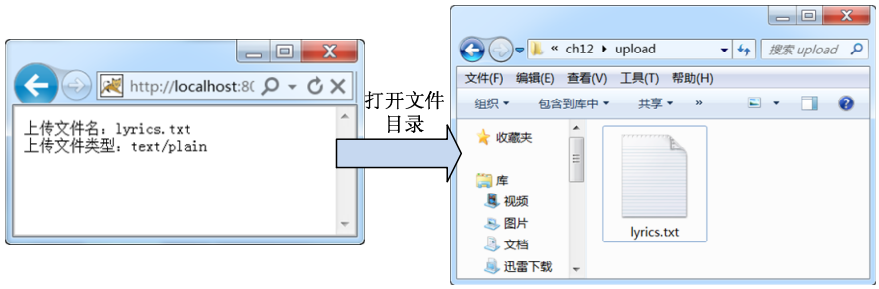



图 12.35　上传结果页面及上传目录

**注意**：在文件上传之前，要先在 WebRoot 文件夹下创建好上传目录，而且文件上传后并不会保存到 MyEclipse 项目中的 upload 目录下，而是上传到项目的发布目录中，以及 Tomcat 文件夹下的 upload 目录中。

## 12.4　多文件上传

文件上传在很多项目中都需要用到，比如博客系统、论坛系统以及邮件系统等。这些系统通常需要上传多个文件，单个文件上传已经不能满足要求了。所以我们就一起来看 Struts 2 是如何完成多文件上传的。

### 12.4.1　实现多文件上传控制器

**【示例 12.6】**首先我们在项目中的 action 文件夹下创建一个多文件上传控制器 SomeFileUploadAction.java，该控制器负责封装所有上传文件、文件名和文件类型，其具体代码如图 12.36 所示。

```
package action;
import java.io.*;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class SomeFileUploadAction extends ActionSupport {
      private File[] uploadFiles;
      private String[] uploadFilesContentType;
      private String[] uploadFilesFileName;
      private String savePath;
      //省略属性的get和set方法
      public String execute() throws Exception {
        File[] files = getUploadFiles();
        for (int i = 0; i < files.length; i++) {
            InputStream is = new FileInputStream(files[i]);
            String uploadPath = ServletActionContext.getServletContext()
                                .getRealPath(getSavePath());
            File toFile = new File(uploadPath, getUploadFilesFileName()[i]);
            OutputStream os = new FileOutputStream(toFile);
            byte[] buffer = new byte[1024];
            int length = 0;
            while ((length = is.read(buffer)) > 0) {
                  os.write(buffer, 0, length);
            }
            is.close();  os.close();
        }
        return SUCCESS;
      }
}
```

设置上传文件的属性

循环每个上传的文件

图 12.36　多文件上传控制器 SomeFileUploadAction.java

## 12.4.2　完成多文件上传

同样，在多文件上传控制器创建完成后，需要在 struts.xml 文件中进行配置，在配置之前首先要填上上传表单页 SomeFileUploadPage.jsp 和上传结果页 SomeFileUploadResultPage.jsp。

上传表单页包含三个文件域和两个按钮，其表单提交到多上传文件控制器，代码如图 12.37 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>多文件上传</title></head>
<body>
      <form action="someFileUpload.action" method="post"
                  enctype="multipart/form-data">
            多文件上传:<br>
            <input type="file" name="uploadFiles"><br>
            <input type="file" name="uploadFiles"><br>
            <input type="file" name="uploadFiles"><br>
            <input type="submit" value="上传">
            <input type="reset">
      </form>
</body>
</html>
```

包含三个文件域

图 12.37　上传表单页 SomeFileUploadPage.jsp

上传结果页，用来显示文件上传结果，包括上传文件的名称以及类型，代码如图 12.38 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head><title>文件上传成功</title></head>
<body>
    <c:forEach items="${uploadFilesFileName}"
            var="filename" varStatus="stat">
    文件名：${filename}，文件类型：${uploadFilesContentType[stat.index]}<br>
    </c:forEach><br>
</body>
</html>
```

循环所有的文件名

图 12.38　上传结果页 SomeFileUploadResultPage.jsp

然后我们对 struts.xml 文件进行配置，配置方法如图 12.39 所示。

```
<action name="someFileUpload"   class="action.SomeFileUploadAction">
        <param name="savePath">/upload</param>
        <result name="success">/SomeFileUploadResultPage.jsp</result>
        <result name="input">/SomeFileUploadPage.jsp</result>
</action>
```

图 12.39　配置文件 struts.xml

配置完成后，就可以开始上传文件了。打开 IE 浏览器，在浏览器地址栏中输入 http://localhost:8080/ch12/SomeFileUploadPage.jsp，显示页面如图 12.40 所示。


图 12.40　上传表单

单击上传按钮完成上传，我们就可以在页面和上传目录中查看文件信息了，如图 12.41 所示。


图 12.41　上传结果页面及上传目录

**注意**：在上传表单中也可以选择上传一个或两个文件，但是如果一个都不传，就会抛出空指针异常。

## 12.5 小结

本章主要讲述了 Struts 2 框架关于国际化和文件上传方面的知识。国际化是 Struts 2 框架的重要内容，集中体现了 Struts 2 框架的普遍应用性，而文件上传又是一个非常实用的功能。本章的重点是掌握 Action 实现国际化的 3 种方式以及完成文件上传的方法。难点内容是学会并掌握多文件上传的应用。希望读者多加练习，争取掌握。

## 12.6 本章习题

1．完成登录表单页面的国际化，能够在中文简体和美国英文中进行切换，其中中文简体登录界面如图 12.42 所示，美国英文登录界面如图 12.43 所示。


图 12.42 中文简体登录页面

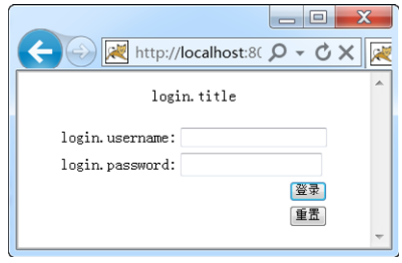



图 12.43 美国英文登录页面

【分析】本题考查读者运用国际化的能力，国际化的关键是要配置好相应的资源文件，只要我们配置好中英文转换资源文件就可以实现题目的要求了。

【核心代码】本题的核心代码如下所示。

messageResource_en_US.properties：

```
login.title    = login
login.password = password
login.username = username
password.required = password is requested
username.required = username is requested
form.choose = Please choose your language
form.option1 = English
form.option2 = Simplified_Chinese
```

messageResource_zh_CN.properties：

```
login.password = \u5BC6\u7801
login.title    = \u767B\u5F55
login.username = \u7528\u6237\u540D
password.required = \u5BC6\u7801\u5FC5\u987B\u8F93\u5165
username.required = \u7528\u6237\u540D\u5FC5\u987B\u8F93\u5165
form.choose = \u8BF7\u9009\u62E9\u8BED\u8A00\u73AF\u5883
form.option1 = \u7F8E\u56FD\u82F1\u8BED
form.option2 = \u7B80\u4F53\u4E2D\u6587
```

login.jsp：

```
<%@page contentType="text/html;charset=gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<body>
<center>
    <s:text name="login.title"></s:text>
    <%--登录表单--%>
    <s:form action="login">
        <s:textfield name="uname" key="login.username"></s:textfield>
        <s:password name="upassword" key="login.password"></s:password>
        <s:submit value="登录"></s:submit>
        <s:reset value="重置"></s:reset>
    </s:form>
</center>
</body>
</html>
```

2．在上传文件表单页面中，添加一个文本框，用来输入用户名。这样在上传文件时，同时可以知道这个文件是由谁上传的，其表单页面如图 12.44 所示。

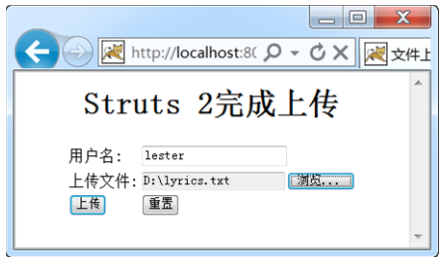



图 12.44　运行结果

【分析】本题考查读者运用文件上传知识的能力，本题的要求是在我们前面讲解的基础上，增加一个用户名来确认文件是由谁上传的。所以我们只要对 action 中的文件进行配置就不难实现题目所有要求的功能。

【核心代码】本题的核心代码如下所示。

upload.jsp：

```
<%@ page language="java" pageEncoding="gb2312"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
    <center>
    <h1>Struts 2 完成上传</h1>
      <s:fielderror/>
      <form action="upload.action" method="post" enctype="multipart/form-data">
         <table>
             <tr>
                 <td>用户名:</td>
                 <td><input type="text" name="username" ></td>
……
……
                 <td><input type="submit" value="上传"></td>
                 <td><input type="reset"></td>
             </tr>
         </table>
      </form>
  </center>
```

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

UploadAction.java：

```
package action;
import java.io.File;
……
public class UploadAction extends ActionSupport {
     private String username;
     private File myFile;
……
     public String getSavePath() {
          return savePath;
     }
     public void setSavePath(String savePath) {
          this.savePath = savePath;
     }

……
     public String execute() throws Exception {
          InputStream is = new FileInputStream(myFile);
          // 设置上传文件目录
          String uploadPath =  ServletActionContext.getServletContext().getRealPath
(getSavePath());
          //设置目标文件
          File toFile = new File(uploadPath,this.getMyFileFileName());
          //输出流
          OutputStream os = new FileOutputStream(toFile);
          byte[] buffer = new byte[1024];
          int length = 0;
          while((length = is.read(buffer)) > 0) {
               os.write(buffer, 0, length);
          }
          is.close();
          os.close();
          return SUCCESS;
     }
}
```

# 第13章 Struts 2 标签库

Struts 2 同 Struts 1 一样，为页面开发提供了大量的标签，但是相比而言 Struts 2 的标签库更为强大。因为 Struts 2 不仅整合了 Dojo 技术，能够生成大量的页面效果，而且它支持 OGNL 表达式，不再依赖任何表现层技术。借助于 Struts 2 标签来开发页面，可以使页面更加整洁而且容易维护，同样可以减少代码量以及开发时间。

## 13.1 Struts 2 标签库概述

Struts 2 标签库是一个比较完善且功能强大的标签库。该标签库大大简化了视图页面代码，提高了视图页面的维护效率。Struts 2 并没有严格地对标签进行分类，不过我们可以大体将其分成三大类，如图 13.1 所示。

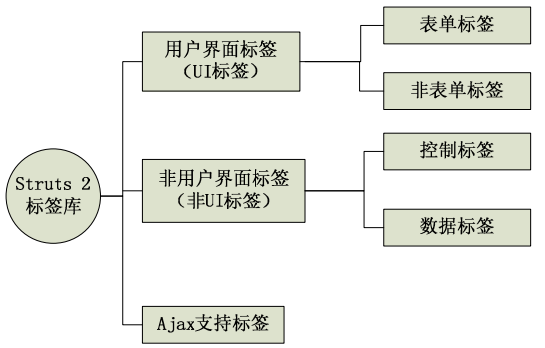



图 13.1 Struts 2 标签库分类

其中，UI 标签主要用于生成 HTML 页面元素；非 UI 标签主要用于数据逻辑输出和数据访问等；Ajax 标签主要用于 Ajax 技术。本章我们主要对非 UI 标签和 UI 标签加以讨论。

非 UI 标签主要用于数据访问和逻辑控制，按其功能又可以分为两类：一类是控制标签，一类是数据标签。表 13.1 中列出了所有的非 UI 标签及其功能。

表 13.1 非UI标签及其功能

| 标 签 分 类 | 标 签 名 | 功 能 |
| --- | --- | --- |
| 逻辑控制标签 | if | 条件判断 |
| | elseif | 条件判断 |

续表

| 标 签 分 类 | 标 签 名 | 功　能 |
|---|---|---|
| 逻辑控制标签 | else | 条件判断 |
| | append | 将集合以追加的方式合并成新集合 |
| | generator | 将字符串分割成多个子串 |
| | iterator | 迭代数组或者集合 |
| | merge | 将集合以交替方式合并成新集合 |
| | sort | 对集合中元素进行排序 |
| | subset | 获得集合的子集合 |
| 数据访问标签 | action | 在页面中调用 Action |
| | bean | 实例化 JavaBean |
| | data | 格式化时间和日期 |
| | debug | 显示调试信息 |
| | i18n | 指定国际化资源文件的文件名 |
| | include | 包含 JSP 等 Web 资源 |
| | param | 传递参数 |
| | property | 输出指定值 |
| | push | 将指定值放入 ValueStack 的栈顶 |
| | set | 创建一个新变量，并保存在指定范围 |
| | text | 输出国际化信息 |
| | url | 生成 url 地址 |

UI 标签主要用来生成 HTML 元素，按其功能也可分为两类：一类是表单标签，另一类是非表单标签。表 13.2 列出了所有的 UI 标签及其功能。

表 13.2　UI标签及其功能

| 标 签 分 类 | 标 签 名 | 功　能 |
|---|---|---|
| 表单标签 | checkbox | 用来生成单个复选框 |
| | checkboxlist | 用来生成多个复选框 |
| | combobox | 用来组合单行文本框和下拉列表框 |
| | doubleselect | 用来生成级联列表框 |
| | head | 用来生成页面文件的 HEAD 部分 |
| | file | 用来生成文件域 |
| | form | 用来生成表单 |
| | hidden | 用来生成隐藏域 |
| | label | 用来生成标签 |
| | optiontransferselect | 用来生成可交互的两个列表框 |
| | optgroup | 用来生成下拉列表框选项组 |
| | password | 用来生成密码输入框 |
| | radio | 用来生成多个单选按钮 |
| | reset | 用来生成重置按钮 |
| | select | 用来生成列表框 |

续表

| 标 签 分 类 | 标 签 名 | 功　　能 |
|---|---|---|
| 表单标签 | submit | 用来生成提交按钮 |
| | textarea | 用来生成多行文本框 |
| | textfield | 用来生成单行文本框 |
| | token | 防止多次提交表单 |
| | updownselect | 用来生成高级列表框 |
| 非表单标签 | actionerror | 显示 ActionError 中的错误信息 |
| | actionmessage | 显示 ActionMessage 中的普通信息 |
| | component | 使用模板 |
| | div | 使用层 |
| | fielderror | 显示 ActionError 中的错误信息 |

要使用 Struts 2 的标签必须在 JSP 页面中使用 taglib 指令来导入 Struts 2 的标签，引入方式如图 13.2 所示。

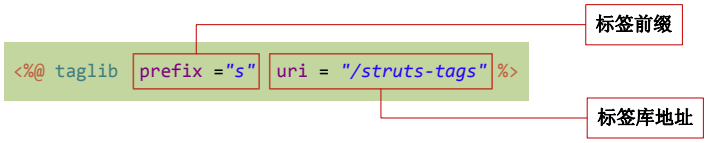



图 13.2　Struts 2 标签引入方式

通过上面的配置就可以在 JSP 页面中使用 Struts 2 提供的标签了。使用 Struts 2 标签的语法格式分为两种，如图 13.3 所示。

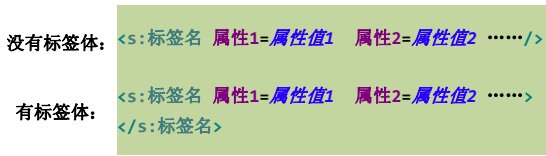



图 13.3　使用 Struts 2 标签的语法格式

## 13.2　控制标签

控制标签属于非 UI 标签，它主要用来完成条件逻辑、循环逻辑的控制，也可以用来对集合进行合并、排序操作，还可以对字符串进行分割操作。控制标签一共包含 9 个标签，我们选取其中常用的 6 个标签为大家介绍其用法。

### 13.2.1　if/elseif/else 标签

if/elseif/else 标签是 3 个标签的组合，这 3 个标签都用于进行条件逻辑控制。其中 if 标签、elseif 标签中提供了一个 test 属性用来进行判断，该属性返回一个布尔值，通过判断该布尔值来决定是否执行标签体的内容。

**【示例 13.1】**下面我们就通过一个范例 TestIF.jsp 来演示如何使用 if/elseif/else 标签组合。在这个示例中，我们使用该标签来判断学生成绩，并根据分值判定等级，具体代码如图 13.4 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>if/elseif/else标签组合使用</title></head>
<body>
    <s:if test="#parameters.score[0]>= 90"> 优秀
    </s:if>
    <s:elseif test="#parameters.score[0] >= 80"> 良好
    </s:elseif>
    <s:elseif test="#parameters.score[0] >= 70"> 中等
    </s:elseif>
    <s:elseif test="#parameters.score[0] >= 60"> 及格
    </s:elseif>
    <s:else>        不及格
    </s:else>
</body>
</html>
```

按照分数划分5个不同的等级

图 13.4　JSP 页面 TestIF.jsp

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestIF.jsp?score=85，打开页面如图 13.5 所示。

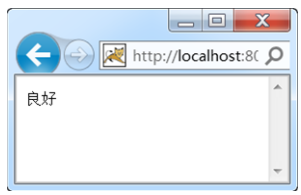



图 13.5　输出学生成绩等级

从 URL 地址可以看出我们传递的参数 score 值为 85，所以输出学生成绩等级为良好。

**注意**：if 标签可以单独使用，也可以与 else 标签、elseif 标签组合使用。但是 else 标签和 elseif 标签不能单独使用，必须和 if 标签组合使用。

## 13.2.2　iterator 标签

iterator 标签一般用来对集合进行遍历，这里所指的集合包括数组、List、Set 以及 Map 对象。iterator 标签中有 3 个属性，如表 13.3 所示。

表 13.3　iterator标签属性列表

| 属　性　名 | 必　　选 | 默　认　值 | 属性值类型 | 说　　明 |
|---|---|---|---|---|
| id | 否 | 无 | String | 指定引用当前便利元素的 ID |
| status | 否 | false | Boolean | 指定遍历时 IteratorStatus 实例 |
| value | 否 | 无 | String | 指定被遍历的集合 |

iterator 标签一般用来遍历 Action 类的集合属性，这里为了演示方便，将在 JSP 页面中创

建该集合用来模拟从 Action 获得的集合。可以使用 OGNL 表达式中的“{e1,e2,e3……}”来生成一个 List 类型集合。

**【示例 13.2】**在下面这个示例 TestIterator.jsp 中，我们主要演示如何通过 iterator 标签遍历 List 类型集合，输出集合中的全部内容，具体代码如图 13.6 所示。

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestIterator.jsp，打开页面如图 13.7 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>iterator标签使用</title></head>
<body>
      <s:iterator value=
            "{'床前明月光','疑是地上霜','举头望明月','低头思故乡'}"
             id="poesy">
       <s:property value="poesy"/><br>
      </s:iterator>
</body>
</html>
```

遍历List,将遍历的值放入poesy变量中


图 13.6　JSP 页面 TestIterator.jsp

图 13.7　iterator 标签运行结果

## 13.2.3　append 标签

append 标签用于将多个集合拼接组合成一个新集合。该标签只包含 id 一个属性。该属性用来指定新集合的名字。在 append 标签中可以指定多个 param 子标签。每一个 param 标签指定一个需要被组合的集合。

**【示例 13.3】**在这个示例中，我们主要演示如何通过 append 标签组合两个 List 集合对象，并输出新集合中的所有元素值，TestAppend.jsp 示例的具体代码如图 13.8 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>append标签使用</title></head>
<body>
      <s:append id="poesy">
            <s:param value="{'大江东去','浪淘尽','千古风流人物'}"></s:param>
            <s:param value="{'故垒西边','人道是','三国周郎赤壁'}"></s:param>
      </s:append>
      <s:iterator value="#poesy" id="sentence">
            <s:property value="sentence"/><br>
      </s:iterator>
</body>
</html>
```

使用append标签来组合多个List对象

图 13.8　JSP 页面 TestAppend.jsp

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestAppend.jsp，打开页面如图 13.9 所示。

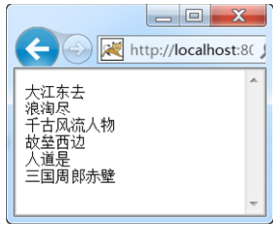



图 13.9　Append 标签运行结果

## 12.2.4　generator 标签

generator 标签用来将指定的字符串按照指定的分隔符分隔成多个子字符串，并将这些子字符串放置到一个集合对象中。转换后的集合对象可以使用 iterator 标签来迭代输出。generator 标签包含 5 个属性，如表 13.4 所示。

表 13.4　generator标签属性列表

| 属　性　名 | 必　　选 | 默　认　值 | 属性值类型 | 说　　明 |
|---|---|---|---|---|
| converter | 否 | 无 | org.apache.struts2.util.Converter | 用来将集合中的字符串转换成对象 |
| count | 否 | 无 | Integer | 指定生成集合的元素数量 |
| id | 否 | 无 | String | 指定级和存储于 page 范围的变量名 |
| separator | 否 | 无 | String | 指定分隔符 |
| val | 否 | 无 | String | 指定被解析的字符串 |

**【示例 13.4】**在下面的示例 TestGenerator.jsp 中，我们将演示如何使用 generator 标签分隔电话号码字符串，并遍历转换后的集合对象，具体代码如图 13.10 所示。

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestGenerator.jsp，打开页面如图 13.11 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>Generator标签使用</title></head>
<body>
    <s:generator separator="-" val="'010-12345678-123-123'"
                 id="phoneNum"></s:generator>
    分隔后子字符串：<br>
    <s:iterator value="#attr.phoneNum" id="number">
        <s:property value="number"/><br>
    </s:iterator>
</body>
</html>
```

分隔指定字符串

图 13.10　JSP 页面 TestGenerator.jsp


图 13.11　generator 标签运行结果

## 13.3　数据标签

数据标签主要用来提供数据访问相关的功能，如通过 action 标签可以显示 Action 中的属性，通过 bean 标签允许直接在 JSP 页面中创建 JavaBean 示例，使用 date 标签可格式化日期和时间等。

### 13.3.1　action 标签

action 标签允许在 JSP 中直接访问并调用 Action，还可以通过 executeResult 属性选择是否将处理结果包含在当前页面中，action 标签包含的属性如表 13.5 所示。

表 13.5　action标签属性列表

| 属 性 名 | 必　选 | 默 认 值 | 属性值类型 | 说　明 |
|---|---|---|---|---|
| executeResult | 否 | false | Boolean | 指定是否将 Action 处理结果对应的视图资源包含到 JSP 页面 |
| flush | 否 | true | Boolean | 指定是否刷新。此操作将写入所有已缓冲的输出字符 |
| id | 否 | 无 | String | 指定引用该 Action 的 ID |
| ignoreContextParams | 否 | false | Boolean | 指定是否传入参数，如为 false，将参数传入调用的 Action |
| name | 是 | 无 | String | 指定调用的 Action |
| namespace | 否 | 当前 namespace | String | 指定调用 Action 对应的 namespace |

**【示例 13.5】**下面的 Action 类 TransferAction.java 将在 JSP 页面中直接被访问并调用。这个类很简单，只有一个属性及一个逻辑方法，具体代码如图 13.12 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
public class TransferAction extends ActionSupport {
        private String param;
        public String getParam() {
                return param;
        }
        public void setParam(String param) {
                this.param = param;
        }
        public String execute() throws Exception {
                return this.SUCCESS;
        }
}
```

param属性，用来封装param参数

图 13.12　Action 类 TransferAction.java

创建完 TransferAction 类后，我们在 struts.xml 文件中配置该 Action，具体代码如图 13.13 所示。

```
<action name="transfer" class="action.TransferAction">
        <result name="success">/ShowParam.jsp</result>
</action>
```

图 13.13　配置文件 struts.xml

然后我们创建一个显示页面 ShowParam.jsp，在该页面中显示 param 属性值，具体代码如图 13.14 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>调用成功页面</title></head>
<body>
    <center><h2>调用成功，参数为<s:property value="param"/></h2></center>
</body>
</html>
```

输出param参数

图 13.14　显示页面 ShowParam.jsp

最后我们创建一个输入页面 TestAction.jsp 来展示如何使用 action 标签调用 Action 类，并考虑忽略和不忽略参数两种情况，具体代码如图 13.15 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>action标签使用</title></head>
<body>
    <h4>将TransferAction返回结果包含到本页面,不忽略参数</h4>
    <s:action name="transfer" executeResult="true"
        ignoreContextParams="false"></s:action>
    <h4>将TransferAction返回结果包含到本页面,忽略参数</h4>
    <s:action name="transfer" executeResult="true"
        ignoreContextParams="true"></s:action>
    <h4>直接访问Action中属性值</h4>
    <s:action name="transfer" id="mytransfer"
        executeResult="false"></s:action>
    TransferAction中param属性值：<s:property value="#mytransfer.param"/>
</body>
</html>
```

考虑忽略和不忽略参数两种情况

图 13.15　输入页面 TestAction.jsp

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestAction.jsp，打开页面如图 13.16 所示。

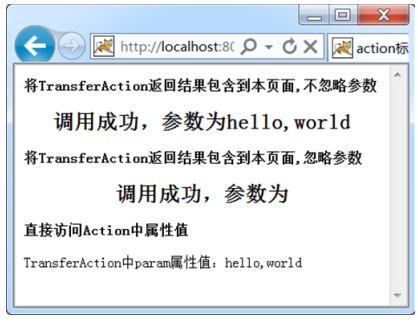



图 13.16　使用 action 标签调用 Action

### 13.3.2　bean 标签

bean 标签允许直接在 JSP 页面中创建 JavaBean 实例。在创建 JavaBean 对象时，如果需要设置 JavaBean 的属性，可以在标签体内使用 param 标签传递参数，bean 标签属性如表 13.6 所示。

表 13.6　bean标签属性列表

| 属　性　名 | 必　　选 | 默　认　值 | 属性值类型 | 说　　明 |
|---|---|---|---|---|
| id | 否 | 无 | String | 指定 JavaBean 实例存储的变量名 |
| name | 是 | 无 | String | 指定 JavaBean 对应的完整类名 |

**【示例 13.6】**接下来我们在包 bean 中创建一个 Action 类 Teacher.java。它只包含了属性及其实现方法，可以被直接访问，其具体代码如图 13.17 所示。

```
package bean;
public class Teacher {
        private String name;
        private int age;
        private boolean sex;
        private String professional;
        /**
         * 各属性的setter和getter方法
         */
}
```

图 13.17　Action 类 Teacher.java

然后我们来演示如何使用 bean 标签实例化 JavaBean，我们要创建的 JSP 页面 TestBean.jsp 的具体代码如图 13.18 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>subset标签使用</title></head>
<body>
<s:bean name="bean.Teacher" id="teacher">
        <s:param name="name">李默</s:param>
        <s:param name="age">28</s:param>
        <s:param name="sex">true</s:param>
        <s:param name="professional">副教授</s:param>
</s:bean>
姓名：<s:property value="#teacher.name"/><br>
年龄：<s:property value="#teacher.age"/><br>
性别：<s:property value="#teacher.sex ? '男':'女'"/><br>
职称：<s:property value="#teacher.professional"/>
</body>
</html>
```

使用bean标签实例化Teacher类

图 13.18　JSP 页面 TestBean.jsp

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestBean.jsp，打开页面如图 13.19 所示。


图 13.19　实例化 JavaBean

### 13.3.3　date 标签

date 标签用来格式化输出指定的时间或者日期。除此之外，date 标签还可以输出指定日期到当前时刻的时间差，data 标签属性如表 13.7 所示。

表 13.7　data标签属性列表

| 属　性　名 | 必　　选 | 默　认　值 | 属性值类型 | 说　　明 |
|---|---|---|---|---|
| format | 否 | 无 | String | 指定格式化样式 |
| id | 否 | 无 | String | 指定引用该元素的 ID |
| name | 是 | 无 | String | 指定引用该元素的名称 |
| nice | 否 | false | Boolean | 指定是否输出指定日期到当前时刻的时间差 |

**【示例 13.7】**我们举一个例子 TestDate.jsp 来演示如何使用 date 标签格式化当前时间，并计算当前时间和新中国成立时间的时间差，具体代码如图 13.20 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>date标签使用</title></head>
<body>
        <%
                Calendar cal = Calendar.getInstance();
                cal.set(1949,10,1);
                pageContext.setAttribute("guoqing",cal.getTime());
        %>
        <s:bean id = "now" name="java.util.Date"/>>
        <h4>使用property输出当前时间：</h4>
        <s:property value="#attr.now"/><br>
        <h4>格式化输出当前时间,格式为yy/MM/dd：</h4>
        <s:date name="#attr.now" format="yy/MM/dd"/><br>
        <h4>格式化输出当前时间,格式为yy-MM-dd HH:mm:ss</h4>
        <s:date name="#attr.now" format="yy-MM-dd HH:mm:ss"/><br>
        <h4>中华人民共和国成立距今：</h4>
        <s:date name="#attr.guoqing" format="yy/MM/dd" nice="true"/><br>
</body>
</html>
```

设置起始时间，page范围

格式化当前时间
并输出时间差

图 13.20　JSP 页面 TestDate.jsp

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestDate.jsp，打开页面如图 13.21 所示。


图 13.21　格式化输出日期时间

## 13.4　表单标签

表单标签用来向服务器提交用户输入信息，绝大部分表单标签都有相应的 HTML 标签与其对应，如<s:textfield>标签同<input type="text"…/>标签，<s:password>标签同<input type="password"…/>标签。通过表单标签可以简化表单开发，还可以实现 HTML 中难以实现的功能，如日期选择器等。

### 13.4.1　简单表单标签

下面我们来看一组，这些标签都可以在 HTML 中找到其对应的标签，标签列表如表 13.8 所示。

表 13.8　简单表单标签列表

| 标　签 | HTML 对应标签 | 说　明 |
|---|---|---|
| <s:form> | <form> | 表单标签 |
| <s:texfield> | <input type = "text"> | 单行文本框 |
| <s:textarea> | < textarea > | 文本域 |
| <s:submit> | <input type = "submit"> | 提交按钮 |
| <s:select> | < select > | 下拉列表框 |
| <s:reset> | <input type = "reset"> | 重置按钮 |
| <s:radio> | <input type = "radio"> | 单选按钮 |
| <s:password> | <input type = "password"> | 密码输入框 |
| <s:checkbox> | <input type = "checkbox"> | 复选按钮 |

**【示例 13.8】**下面我们通过这 9 种简单标签开发一个员工登记表，用来输入员工的姓名、口令、学历、性别等信息，具体代码如图 13.22 所示。

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/SimpleForm.jsp，打开页面如图 13.23 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>员工登记表</title></head>
<body>
   <center>
   <h2>员工登记表</h2>
   <s:form action="regist" method="post"><!--表单标签-->
      <s:textfield name="eName" label="姓名"></s:textfield>
      <s:password name="ePassword" label="口令"></s:password>
      <s:select name="eDegree" label="学历"
            list="{'高中及以下','大专','本科','研究生及以上'}"></s:select>
      <s:radio name="sex" label="性别" list="{'男','女'}"></s:radio>
      <s:textarea name="register" label="登记协议" value="登记协议省略">
        </s:textarea>
      <s:checkbox name="love" label="同意员工登记协议"></s:checkbox>
      <s:submit value="提交"></s:submit>
      <s:reset value="重置"></s:reset>
   </s:form>
   </center>
</body>
</html>
```

9种标签的综合运用

图 13.22　员工登记表 SimpleForm.jsp

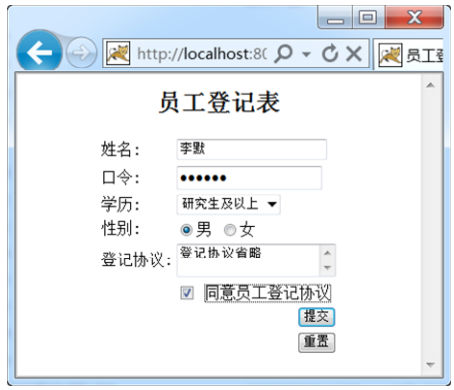



图 13.23　员工登记表示意图

## 13.4.2　combobox 标签

combobox 标签用来生成一个单行文本框和下拉列表框的组合，而且这两个元素对应同一个参数。combobox 标签多用于提示信息或者密码保护问题询问等情况。combobox 标签包括的常用属性如表 13.9 所示。

表 13.9　combobox标签属性列表

| 属　性　名 | 必　　选 | 默　认　值 | 属性值类型 | 说　　明 |
|---|---|---|---|---|
| name | 否 | 无 | String | 指定组合框名称 |
| label | 否 | 无 | String | 指定组合框前显示文本 |
| list | 是 | 无 | String | 指定组合框选项集合 |

**【示例 13.9】**在下面这个例子 TestCombobox.jsp 中，我们将演示如何使用 combobox 标签来生成单行文本框和下拉列表框的组合，用来输入密码保护问题，其具体代码如图 13.24 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>combobox标签使用</title></head>
<body>
   <s:form>
      <s:combobox label="密码保护问题"
        list="{'我的名字叫什么','我的出生地在哪里','我多大了'}"
        name="pwSafeQue">
      </s:combobox>
   </s:form>
</body>
</html>
```

单行文本框和下拉列表组合

图 13.24　JSP 页面 TestCombobox.jsp

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestCombobox.jsp，打开页面如图 13.25 所示。

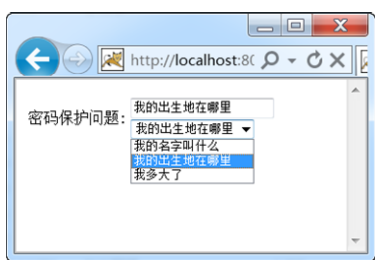



图 13.25　输入密码保护问题

## 13.4.3　datetimepicker 标签

datetimepicker 标签用来生成一个文本框和日期、时间选择器的组合。在选择器中选择完某个日期或者时间时，会自动将被选择的日期或者时间输入文本框中。datetimepicker 标签包括的常用属性如表 13.10 所示。

表 13.10　datetimepicker标签属性列表

| 属 性 名 | 必　　选 | 默 认 值 | 属性值类型 | 说　　明 |
|---|---|---|---|---|
| displayFormat | 否 | 无 | String | 指定日期显示格式 |
| displayWeeks | 否 | 6 | Integer | 指定显示的星期数 |

续表

| 属 性 名 | 必　选 | 默 认 值 | 属性值类型 | 说　明 |
|---|---|---|---|---|
| endDate | 否 | 2941-10-12 | Date | 指定最后的日期 |
| formatLength | 否 | short | String | 指定日期格式 |
| label | 否 | 无 | String | 指定日期选择器前显示文本 |
| name | 否 | 无 | String | 指定日期选择器名称 |
| startDate | 否 | 1492-10-12 | Date | 指定最开始日期 |
| type | 否 | date | String | 指定日期选择类型 |
| value | 否 | 无 | String | 指定默认初始化时间 |

**【示例 13.10】**我们创建一个实例 TestDatetimepicker.jsp，在这个示例中演示如何使用 datetimepicker 标签来生成一个日期选择器，用来输出用户的出生日期，其具体代码如图 13.26 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<%@ taglib prefix="sd" uri="/struts-dojo-tags" %>
<html>
<head><title>datetimepicker标签使用</title></head>
<s:head theme="xhtml"/>
<sd:head parseContent="true"/>
<body>
   <s:form>
       <sd:datetimepicker label="出生日期"  name="birth" value="today">
        </sd:datetimepicker>
   </s:form>
</body>
</html>
```

使用datetimepicker标签

图 13.26　JSP 页面 TestDatetimepicker.jsp

打开浏览器，在浏览器地址栏中输入 http://localhost:8080/ch13/TestDatetimepicker.jsp，打开页面如图 13.27 所示。

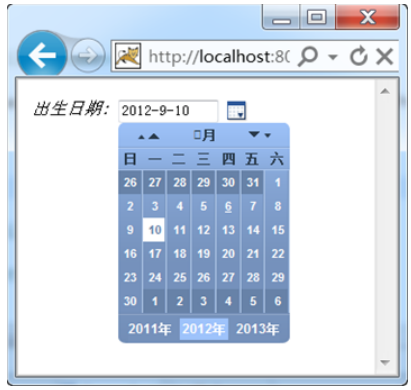



图 13.27　选择出生日期

## 13.5　小结

本章主要讲述了 Struts 2 标签库方面的知识。标签库是 Struts 2 框架的核心内容，集中体现了 Struts 2 框架的优越性。通过标签库不仅可以减少代码量以及开发时间，而且能够生成大量的页面效果。本章的重点和难点都是能否熟练掌握控制标签、数据标签以及表单标签的用法。运用 Struts 2 标签库可以给我们的编程工作带来极大的便利，希望读者多加练习，争取掌握。

## 13.6　本章习题

1．请读者参照示例 13.1 使用 if/elseif/else 标签判断用户年龄，如果年龄小于 18，则显示未成年人；如果年龄大于等于 18 并小于 30，则显示年轻人；如果年龄大于等于 30 并小于 60，则显示中年人；如果年龄大于等于 60，则显示老年人，示例输出页面结果如图 13.28 所示。

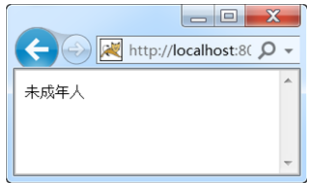



图 13.28　运行结果

【分析】本题考查读者运用控制标签的能力，if/elseif/else 标签很简单，只要我们参照示例 13.1 将成绩修改为年龄就可以实现题目的要求了。

【核心代码】本题的核心代码如下所示。

practice01.jsp：

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head><title>判断用户年龄段</title></head>
<body>
     <s:set name="age" value="#parameters.age[0]" scope="page"/>
     <s:if test="#attr.age < 18">
          未成年人
.....
</body>
</html>
```

2．请读者参照示例 13.4 使用 generator 标签分隔字符串“123-456^789”，示例输出页面结果如图 13.29 所示。

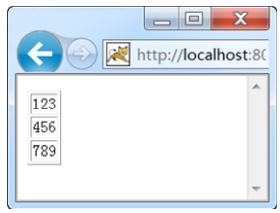



图 13.29　运行结果

【分析】本题考查读者运用控制标签的能力，generator 标签也很简单，只要我们参照示例 13.4 对其进行适当就可以实现题目的要求了。

【核心代码】本题的核心代码如下所示。

practice02.jsp：

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
……
<body>
    <table border="1">
    <s:generator separator="-,^" val="'123-456^789'">
        <s:iterator status="sta">
            <tr>
                <td><s:property/></td>
            </tr>
        </s:iterator>
    </s:generator>
    </table>
</body>
</html>
```

3．请读者参照示例 13.7 使用 date 标签输出当前时间，并计算出香港回归的时间，示例输出页面结果如图 13.30 所示。

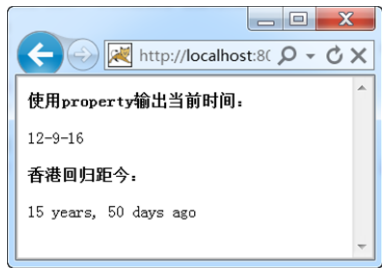



图 13.30　运行结果

【分析】本题考查读者运用数据标签的能力，date 标签也很简单，只要我们参照示例 13.7 对其进行适当就可以实现题目的要求了。

【核心代码】本题的核心代码如下所示。

practice03.jsp：

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
……
<body>
    <%
        Calendar cal = Calendar.getInstance();
        cal.set(1997,7,1);
        pageContext.setAttribute("xianggang",cal.getTime());
    %>
    <s:bean id = "now" name="java.util.Date"/>
    <h4>使用 property 输出当前时间：</h4>
    <s:property value="#attr.now"/><br>
    <h4>香港回归距今：</h4>
    <s:date name="#attr.xianggang" format="yy/MM/dd" nice="true"/><br>
</body>
```

```
</html>
```

4．请读者参照示例 13.10 使用 datetimepicker 标签生成时间选择器，用来设置时间，示例输出页面结果如图 13.31 所示。

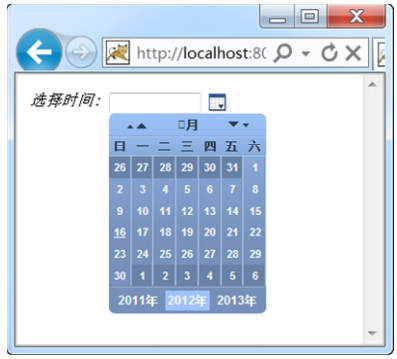



图 13.31　运行结果

【分析】本题考查读者运用表单标签的能力，datetimepicker 标签也很简单，只要我们参照示例 13.10 对其进行适当就可以实现题目的要求了。

【核心代码】本题的核心代码如下所示。

practice04.jsp：

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<%@ taglib prefix="sd" uri="/struts-dojo-tags" %>
<html>
……
<s:head theme="xhtml"/>
<sd:head parseContent="true"/>
<body>
  <s:form>
    <sd:datetimepicker label="选择时间"  name="birth" value="today">
     </sd:datetimepicker>
  </s:form>
</body>
</html>
```

PART 3

# Hibernate 技术篇

# 第14章 Hibernate 框架入门

Hibernate 是目前最流行的持久层框架，专注于数据库操作。使用 Hibernate 框架能够使开发人员从烦琐的 SQL 语句和复杂的 JDBC 中解脱出来。本章将首先详细介绍什么是 ORM 以及其优势，然后会为大家演示如何为项目中添加 Hibernate 支持，最后通过一个实际项目介绍开发 Hibernate 程序的基本步骤和开发技巧。

## 14.1 Hibernate 概述

Hibernate 如图 14.1 所示，是一个开放源代码的对象关系映射框架。它对 JDBC 进行了非常轻量级的对象封装，使得 Java 程序员可以随心所欲地使用对象编程思维来操纵数据库。Hibernate 可以应用在任何使用 JDBC 的场合，既可以在 Java 的客户端程序使用，也可以在 Servlet/JSP 的 Web 应用中使用。Hibernate 的对象关系映射是非常强大并高性能的，其目标是使开发人员从 95%的数据持久化工作中解脱出来。


图 14.1 Hibernate 框架标识

### 14.1.1 什么是 ORM

在了解 ORM 之前，我们先来了解一下什么是持久化技术。持久化技术，就是把数据（如内存中的对象）保存到可永久保存的存储设备中（如磁盘）。持久化的主要应用是将内存中的对象存储在关系型的数据库中，当然也可以存储在磁盘文件中、XML 数据文件中等，如图 14.2 所示。

持久化主要是和数据库打交道的层次，在数据库中对数据的增加、删除、查找和修改都是通过持久化来完成的。

ORM（Object/Relational Mapping，对象/关系映射）是一种常用的持久化技术。我们所说的对象是指使用的编程语言是面向对象的，关系是指使用的数据库是关系型数据库。ORM 的

映射关系可以用图 14.3 来表示。

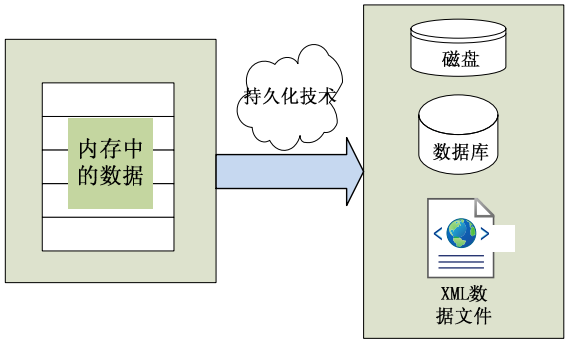



图 14.2　持久化技术

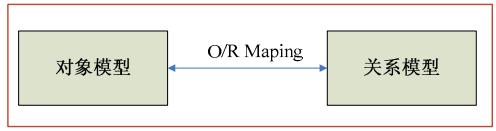



图 14.3　ORM 映射关系

使用 ORM 之后不再需要与复杂的 SQL 语句打交道。通过创建一个持久化类来映射数据库的一个数据库表，如图 14.4 所示。其中持久化的属性则映射到数据库表中的字段。当使用面向对象的方式来操作持久化对象时，ORM 框架能自动将这些操作转化成 SQL 语句，从而完成对数据库的操作。

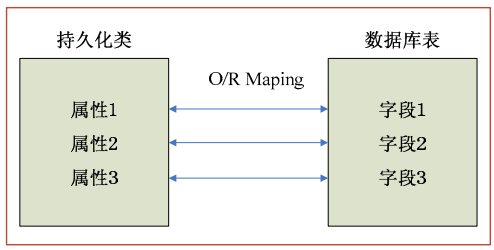



图 14.4　持久化类与数据库表映射关系

## 14.1.2　为什么要使用 ORM

现在知道了什么是 ORM，但是读者可能会有疑问，我已经能够熟练使用 JDBC 编程了，为什么还要使用 ORM 呢？使用 ORM 是整个软件业发展的趋势，下面从代码、架构及性能 3 方面来分析为什么要使用 ORM，以及使用它的好处究竟有哪些。

（1）大大降低了代码量，具体说明如图 14.5 所示。

（2）实现数据库底层透明化，具体说明如图 14.6 所示。

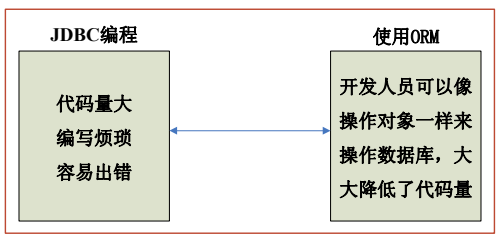



图 14.5　使用 ORM 降低了代码量

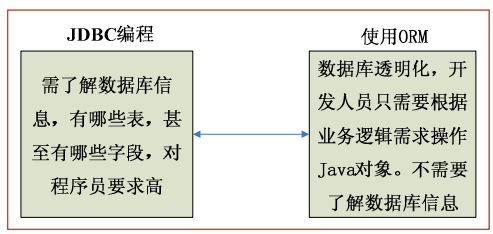



图 14.6　使用 ORM 实现数据库的透明化

（3）性能大大优化，具体说明如图 14.7 所示。

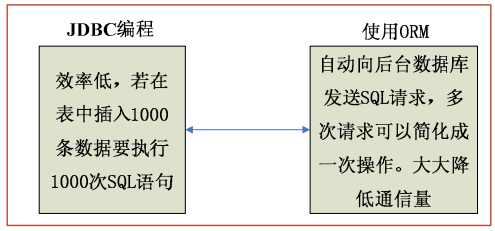



图 14.7　使用 ORM 使性能大大优化

### 14.1.3　使用 Hibernate 的优势

目前比较流行的 ORM 框架主要有 Hibernate、iBATIS 以及最新的 EJB3 版本。iBATIS 框架并没有真正实现 ORM 框架，而 EJB3 是重量级开发框架，不适合轻量级开发。

Hibernate 框架是一个完整的持久层解决方案，通过 Hibernate 的支持，可以使用面向对象方式进行各种数据库操作，从而取代传统的 JDBC 数据库操作。有关 Hibernate 的优势，我们可以用图 14.8 来表示。

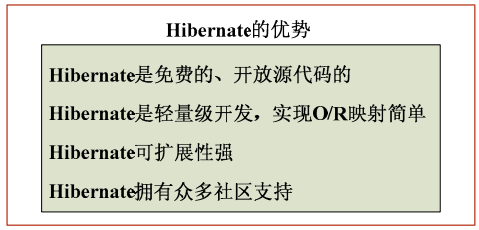



图 14.8　使用 Hibernate 的优势

## 14.2　在程序中使用 Hibernate

在应用程序中使用 Hibernate 框架非常简单。只要在 CLASSPATH 环境变量中指定 Hibernate 框架的 jar 包，就可以在程序中像使用其他的 jar 包一样使用 Hibernate。但要想使用 Hibernate 框架，需要进行一些配置。如果系统比较大，将会产生非常大的工作量。因此，要想更好的使用 Hibernate，就需要一个支持 Hibernate 的 IDE，如 MyEclipse。由于在程序中经常要使用数据库，我们先来安装一个 MySQL 数据库。

### 14.2.1　安装 MySQL 数据库

MySQL 数据库是一个小型的关系型数据库。它体积小、速度快，而且是免费、开源的。对于一般的个人使用者来说，MySQL 数据库提供的功能和性能已经绰绰有余，尤其适合初学者学习使用。本书与数据库相关的实例都是基于 MySQL 数据库的。

为了安全起见，建议大家从官网下载 MySQL。MySQL 数据库下载的官方网址是 http://dev.mysql.com/downloads/，其下载方法如图 14.9 所示。

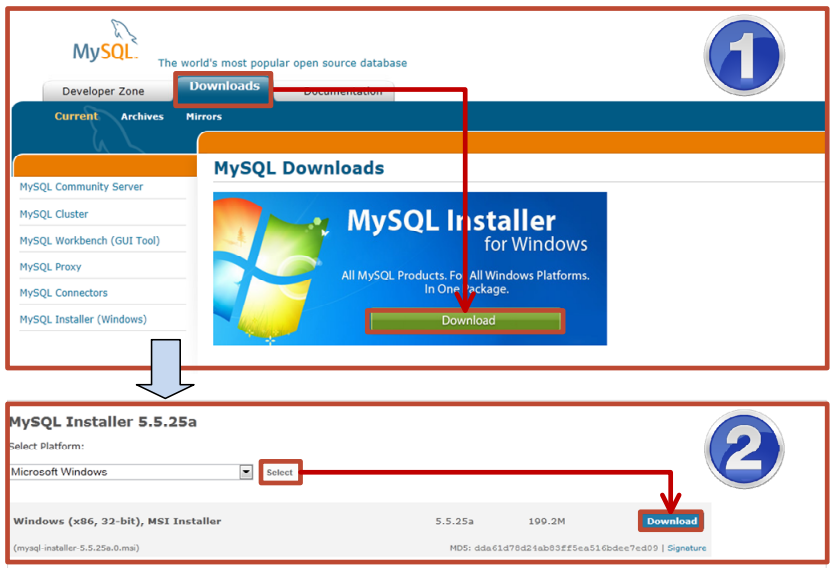



图 14.9　下载 MySQL 方法

由于 MySQL 安装软件包相对较大，所以下载需要一定时间，请读者耐心等待。下载完成后，我们继续来看 MySQL 的安装方法，如图 14.10 所示。


图 14.10　MySQL 的安装过程一

然后单击“Next”按钮，进入 Configuration，接着按图 14.11 所示的步骤进行操作。

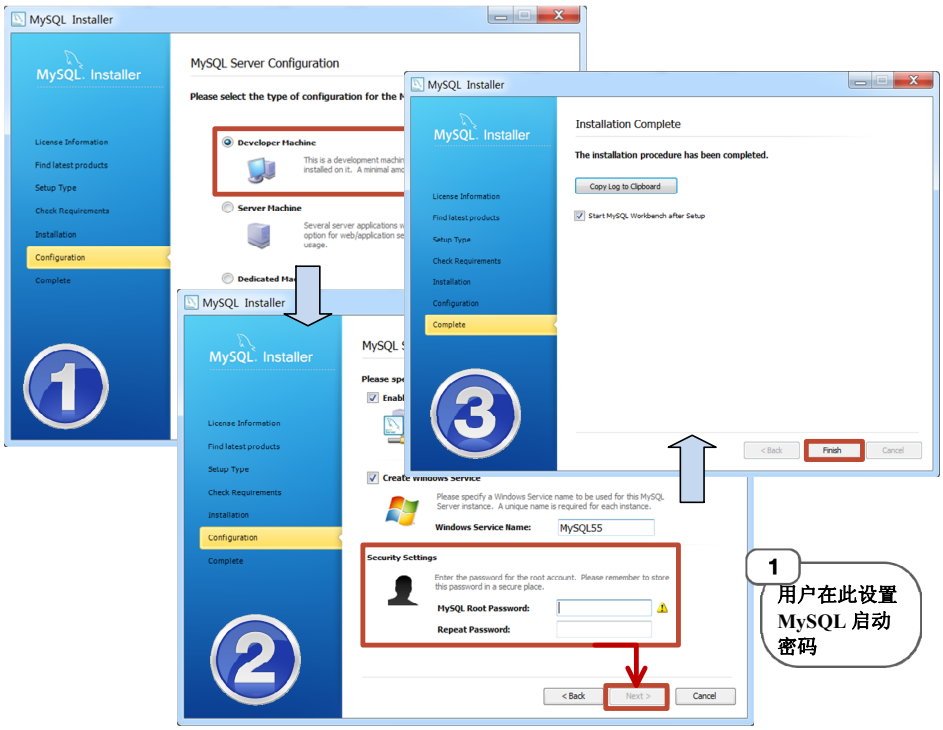



图 14.11　MySQL 的安装过程二

单击“Finish”按钮，完成 MySQL 的安装。系统自动弹出启动界面，如图 14.12 所示。

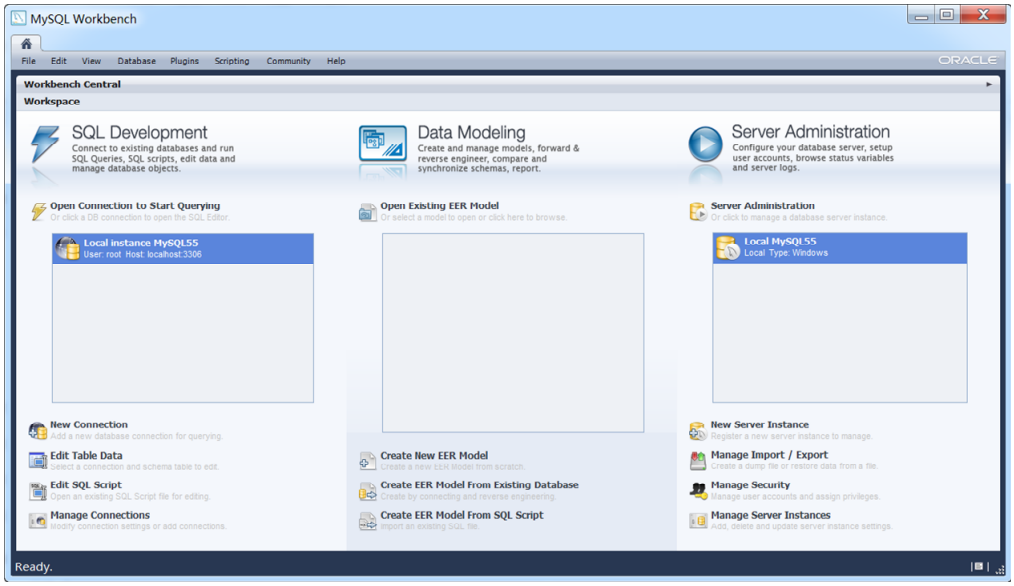



图 14.12　MySQL 的启动界面

### 14.2.2　MyEclipse 对 Hibernate 的支持

对 MyEclipse 建立的工程在默认情况下是不支持 Hibernate 的，需要按照下面的步骤进行操作，才能为工程添加 Hibernate 支持。

由于读者所使用的 MyEclipse 可能不支持 MySQL 数据库，我们先将 MySQL 数据库应用添加到 MyEclipse 开发工具中去。启动 MyEclipse，选择“Windows”→“Open Perspective”→“MyEclipse Database Explorer”命令。在弹出的 DB Browser 区域内，右击选择菜单项“New”，建立一个新连接，建立过程如图 14.13 所示。

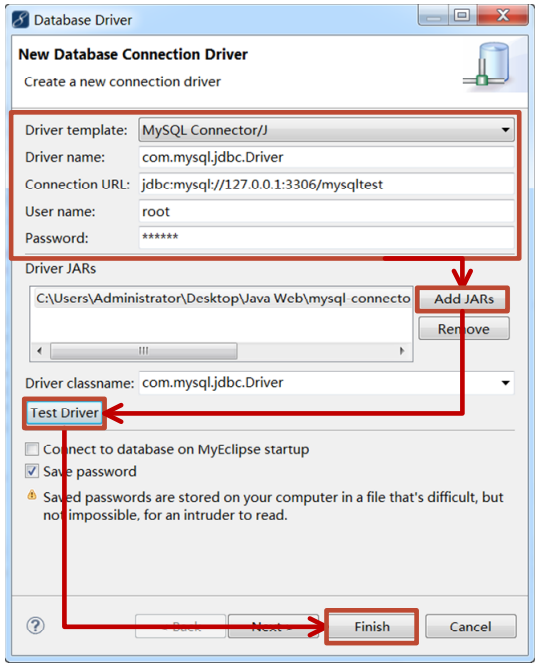



图 14.13　创建 MySQL 数据库连接

图中需要在 Driver JARs 中单击 Add JARs 添加数据库驱动。MySQL 的数据库驱动可以在 http://www.mysql.com/downloads/中下载，下载过程如图 14.14 所示。

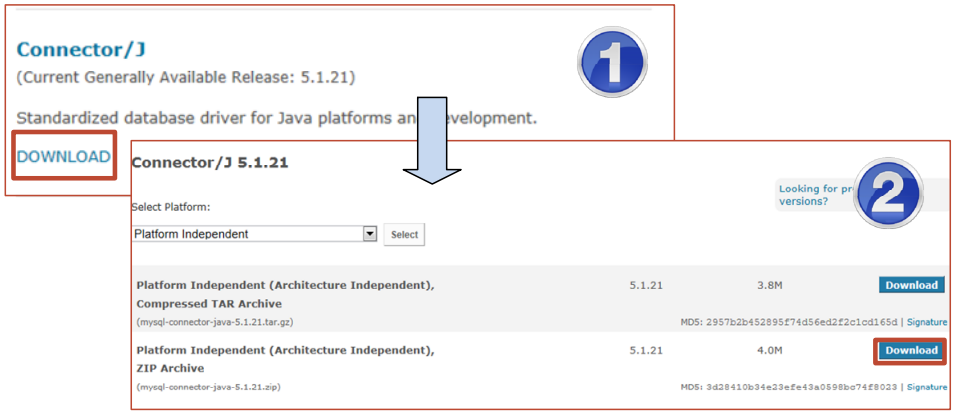



图 14.14　MySQL 的数据库驱动下载过程

引入驱动之后，单击“TestDriver”按钮，如果显示如图 14.15 所示的页面说明安装成功，单击“Finish”按钮结束配置过程。

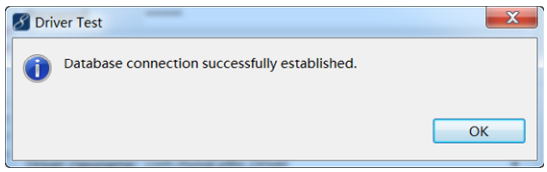



图 14.15　正确安装 MySQL 数据库驱动

然后再创建一个 Java Project（注意不是 Web Project）ch14，单击鼠标右键选择“MyEclipse”→“Add Hibernate Capabilities”选项，后面的具体操作如图 14.16 所示。

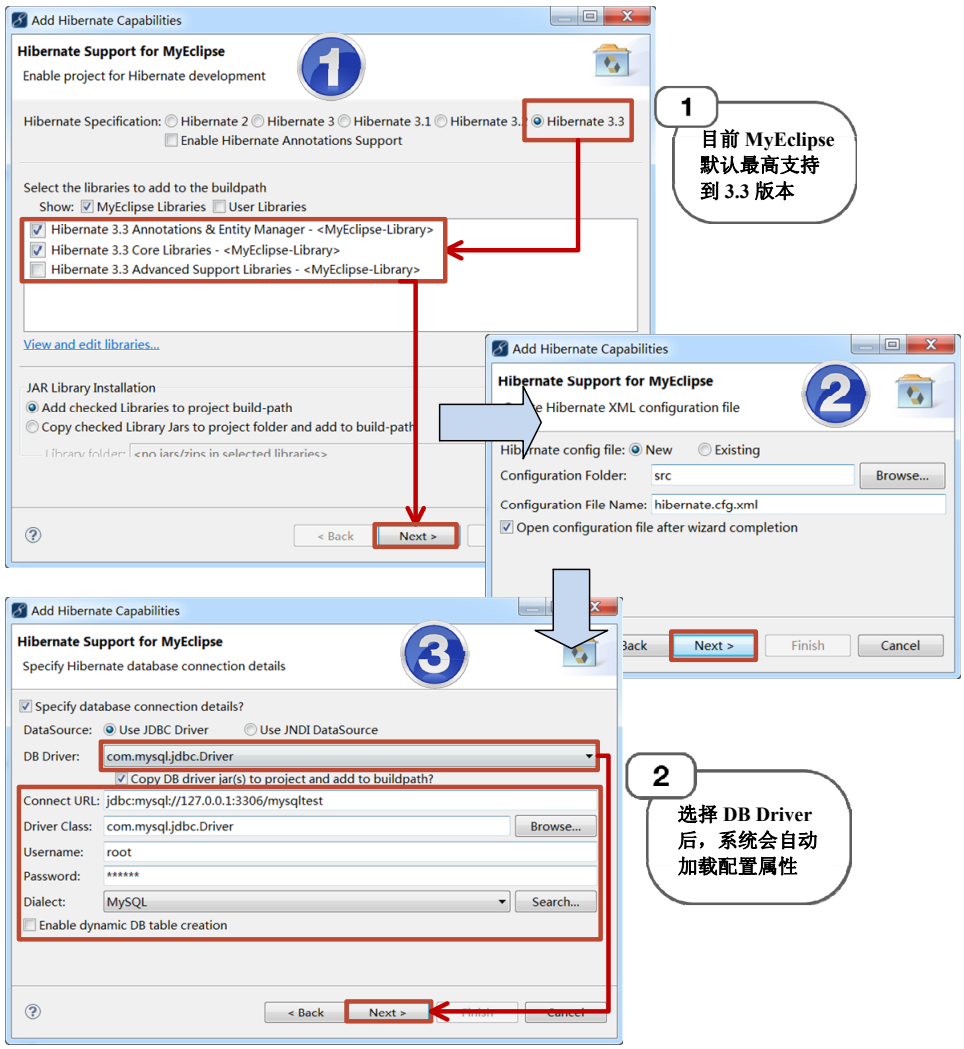



图 14.16　配置数据库连接源

然后，还要自行设置 SessionFactory 配置页面，如图 14.17 所示。

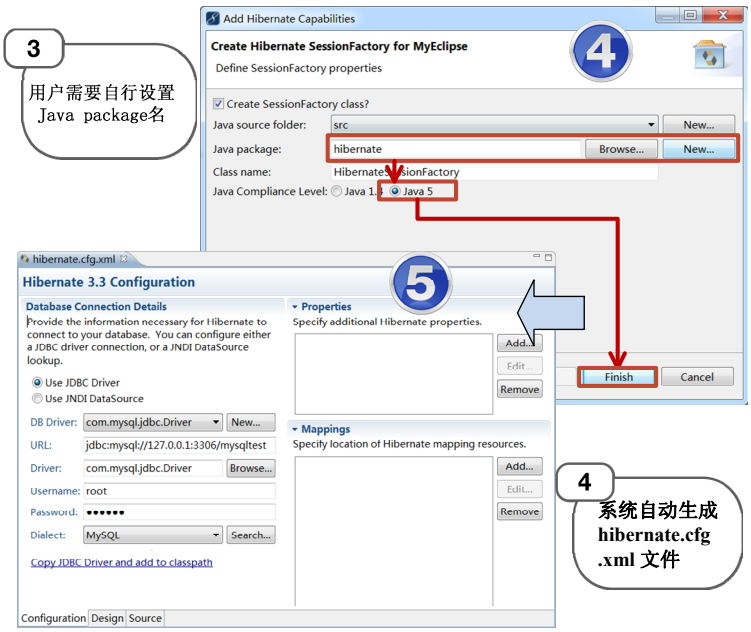



图 14.17　配置 SessionFactory

这样我们就完成了 MyEclipse 对 Hibernate 的支持配置，可以使用 MyEclipse 来完成使用 Hibernate 的程序开发了。

## 14.3　第一个 Hibernate 程序

在完成了 MyEclipse 对 Hibernate 的支持配置之后，本节通过一个 Java 应用程序来演示如何使用 Hibernate 完成持久化操作。通过该项目，读者可以了解开发 Hibernate 程序的基本步骤，感受到 SQL 操作数据库的魅力所在。

### 14.3.1　开发 Hibernate 程序的基本步骤

在开始实际项目开发之前，首先来了解开发 Hibernate 程序的基本步骤，其具体内容及相关说明可以用图 14.18 来表示。

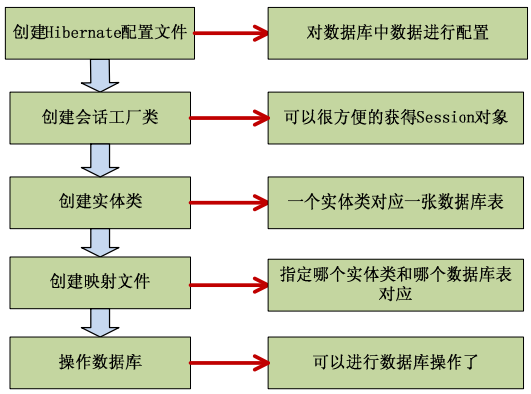



图 14.18　开发 Hibernate 程序的基本步骤

### 14.3.2 创建数据库

我们在 MySQL 数据库中创建一个数据库表 student，用来保存学生信息。student 表一共包含 4 个字段，分别用来保存学生的学号、姓名、科目和成绩，创建 student 表的 SQL 语句如图 14.19 所示。

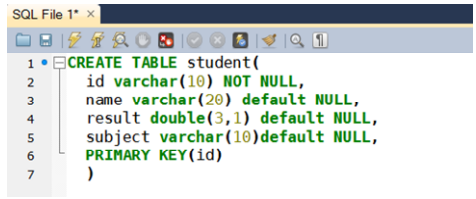



图 14.19 创建 student 表

### 14.3.3 创建 Hibernate 配置文件

Hibernate 从其配置文件中读取和数据库有关的信息。前面我们通过 MyEclipse 自动创建了一个 XML 格式的配置文件，文件名为 hibernate.cfg.xml。该文件中配置了数据库连接 URL、数据库连接驱动、数据库用户名以及用户密码。这里还配置一个属性 dialect，用来指定数据库产品类型。如果需要调试 Hibernate，可以将 show_sql 属性设为 true，这样 Hibernate 在进行持久化时会将相应的 SQL 语句输出到控制台。

**【示例 14.1】**我们来看 hibernate.cfg.xml 文件的配置，它是由系统自动生成的。我们只是在文件中增加了 show_sql 属性，具体代码内容如图 14.20 所示。

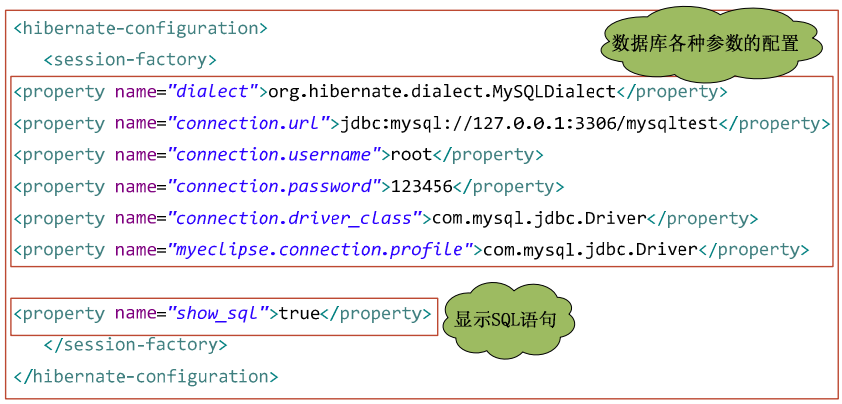



图 14.20 hibernate.cfg.xml 配置文件

### 14.3.4 创建会话工厂类

在使用 Hibernate 操作数据类型时，需要得到一个 Session 对象。通过调用 Session 对象的方法，才能完成数据库操作，如添加记录、删除记录等。这个 HibernateSessionFactory.java 也是由系统自动生成的。它存放在 14.2.2 小节中我们创建的包中，在这里就不为大家展示了。

### 14.3.5 创建实体类

实体类用来映射数据库中的数据库表，实体类中的属性与数据库表中的字段相对应。持久

化类是一个 POJO 类（简单的 Java 对象），不用继承和实现任何类或接口。

**【示例 14.2】**下面我们创建 student 表对应的实体类 Student.java，该实体类包含 4 个属性，分别用来保存学生学号、姓名、科目及成绩，具体代码如图 14.21 所示。

```
package po;
import javax.persistence.Entity;
public class Student {
        private String id;
        private String name;
        private String subject;
        private double result;

        public String getId() {
                return id;
        }
        public void setId(String id) {
                this.id = id;
        }
        //省略其他属性的get和set方法
}
```

定义四种属性

属性对应的get和set方法

图 14.21　实体类 Student.java

## 14.3.6　创建对象关系映射文件

关系映射文件用来映射持久化类和数据库表，从而将持久化类的属性和数据库表中的字段联系起来。其中 id 元素用来定义主键标识，property 元素用来定义其他属性。映射文件的文件名一般为持久化类名加上“.hbm.xml”，文件保存在持久化类的同一目录下。

**【示例 14.3】**下面的实例 Student.hbm.xml 为关系映射文件，用来映射持久化类和学生数据库表，具体代码如图 14.22 所示。

```
<hibernate-mapping>
        <class name="po.Student">
                <id name="id"> <generator class="assigned"></generator></id>

                <property name="name"></property>
                <property name="subject"></property>
                <property name="result"></property>
        </class>
</hibernate-mapping>
```

定义主键标识及生成策略

配置其他属性

图 14.22　关系映射文件 Student.hbm.xml

在添加完配置文件后，需要在 hibernate.cfg.xml 文件中进行配置，即将上述代码添加到<session-factory>与</session-factory>标签之间，具体代码如图 14.23 所示。

```
<mapping resource="po/Student.hbm.xml" />
```

关系映射文件

图 14.23　在 hibernate.cfg.xml 文件中配置映射文件

### 14.3.7 完成插入数据

如果需要使用 Hibernate 来插入数据，需要调用 Session 对象的 save 方法。save 方法接受一个 Object 类型的参数，该参数为一个封装了所有插入数据的对象。

**【示例 14.4】**使用 InsertTest.java 往数据库表中插入一条数据。在插入数据之前先开启一个事务，如果代码出现异常，则进行回滚操作，本实例具体代码如图 14.24 所示。

```
package test;
import javax.persistence.Entity;
import org.hibernate.Session;
import org.hibernate.Transaction;
import hibernate.HibernateSessionFactory;
import po.Student;
public class InsertTest {
        public static void main(String[] args) {
        Student student = new Student(); student.setId("123456");
        student.setName("李默");   student.setSubject("大学物理");
        student.setResult(90);
          Session session = HibernateSessionFactory.getSession();
        Transaction  transaction = null;
        try{
                transaction = session.beginTransaction();
                session.save(student);
                transaction.commit();
                System.out.println("添加记录成功!");
        }catch(Exception ex) {
                ex.printStackTrace();
                transaction.rollback();
        }
        HibernateSessionFactory.closeSession();
    }
}
```

设置Student属性值

开启事务
保存学生信息
提交事务

图 14.24　插入测试类 InsertTest.java

运行该程序，在控制台输出如图 14.25 所示的信息。

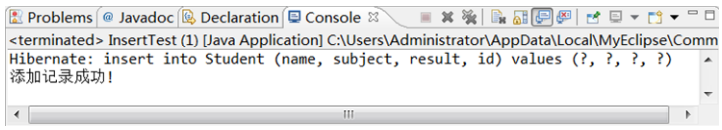



图 14.25　控制台输出信息

打开 MySQL 查看工具，查看 student 表的相关信息，会发现里面已经被插入了一条数据，如图 14.26 所示。

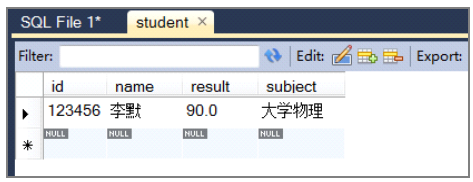



图 14.26　student 表中的数据

### 14.3.8　查询学生列表

首先可以按照同样的方法往数据库中多增加几条数据信息，然后可以实现按条件对数据库表中的信息进行查询。

**【示例 14.5】**新建一个 QueryTest.java 文件，在该文件中通过 Session 对象来查询 student 表中的数据，具体代码如图 14.27 所示。

```
public class QueryTest {
  public static void main(String[] args) {
      Session session = HibernateSessionFactory.
      Query query = session.createQuery
                ("from Student as stu where stu.result >= 85");
      List list = query.list();
      Iterator iter = list.iterator();
      Student stu = null;
      System.out.println("学号\t\t姓名\t科目\t成绩");
      while(iter.hasNext()) {
            stu = (Student) iter.next();
            System.out.println(stu.getId() + "\t" + stu.getName() + "\t"
                        + stu.getSubject() + "\t" + stu.getResult());
      }
      HibernateSessionFactory.closeSession();
  }
}
```

获得Session对象，查询student表

遍历list表

图 14.27　查询测试类 QueryTest.java

最后运行 QueryTest 类，这时在控制台上打印出如图 14.28 所示的信息。

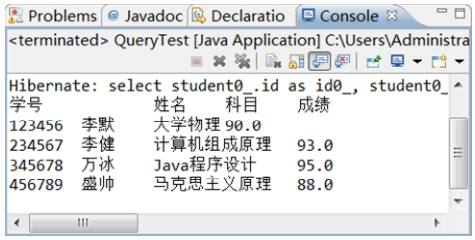



图 14.28　控制台输出信息

这样，完成了第一个 Hibernate 程序的开发设计。

## 14.4　小结

本章是 Hibernate 的入门章节，首先介绍了什么是 ORM，以及为什么要使用 Hibernate，然后介绍了 MySQL 数据库的安装过程以及如何在 MyEclipse 开发工具中使用 Hibernate，最后又通过一个完整的实例，展示了运用 Hibernate 对数据库进行操作的过程。本章的重点和难点都是理解并争取掌握 Hibernate 的开发过程，力争为后面章节的学习打下坚实的基础。

## 14.5 本章习题

1．创建一个数据库表 flight，该表中包含飞机航班号、起飞城市、降落城市、最大承载人数、票价以及折扣等6个字段，其中航班号为主键。根据 flight 表创建与之相对应的实体类 Flight。

【分析】本题考查读者建立 Hibernate 程序的能力，建立数据库表和与之相对应的实体类是我们开发 Hibernate 程序的第一步，只要参照 14.3 所述的内容进行修改就可以实现题目的要求了。

【核心代码】本题的核心代码如下所示。

flight.sql：

```
CREATE TABLE `flight` (
  `no` varchar(11) NOT NULL DEFAULT '',
  `flycity` varchar(20) DEFAULT NULL,
  `landingcity` varchar(20) DEFAULT NULL,
  `personNum` int(11) DEFAULT NULL,
  `price` double(6,2) DEFAULT NULL,
  `discount` double(3,2) DEFAULT NULL,
  PRIMARY KEY (`no`)
) ENGINE=InnoDB DEFAULT CHARSET=gb2312;
```

Flight.java：

```
package po;
public class Flight {
     private String no;
     private String flycity;
……
     public String getNo() {
          return no;
     }
     public void setNo(String no) {
          this.no = no;
     }
……
}
```

2．创建对象关系映射文件 Flight.hbm.xml，完成数据库表 flight 与实体类 Flight 之间的映射，并编写程序，用来查询所有的航班信息，执行效果如图 14.29 所示。


图 14.29 运行结果

【分析】本题考查读者建立对象关系映射文件以及查询数据库的能力，建立的对象关系映射文件首先一定不要忘记在 hibernate.cfg.xml 文件中进行映射说明，然后查询所有表中数据信息就可以实现题目中的要求了。

【核心代码】本题的核心代码如下所示。

Flight.hbm.xml：

```
<?xml version="1.0" encoding="utf-8"?>
……
<hibernate-mapping>
    <class name="po.Flight"><!-- 每个 class 对应一个持久化对象 -->
        <id name="no"><!-- id 元素用来定义主键标识,并指定主键生成策略 -->
            <generator class="assigned"></generator>
        </id>
        <property name="flycity"></property>
……
    </class>
</hibernate-mapping>
```

3．QueryFlight.java：

```
package test;
import java.util.Iterator;
……
public class QueryFlight {
    public static void main(String[] args) {
        Session session = HibernateSessionFactory.getSession();//获得 Session 对象
        Query query = session.createQuery("from Flight");
        List list = query.list();//查询结果保存到 list 中
        Iterator iter = list.iterator();//获得 list 的内部迭代器
        Flight flight = null;
        System.out.println("飞机航班号\t 起飞城市\t 降落城市\t 最大承载人数\t 票价\t 折扣");
        while(iter.hasNext()) {        //遍历 list
            flight = (Flight) iter.next();
            System.out.println(flight.getNo() + "\t\t" +
……
            );
        }
        HibernateSessionFactory.closeSession();//关闭 Session 对象
    }
}
```

# 第 15 章　Hibernate 配置和会话

Hibernate 的配置包括两个重要部分，一个是 Hibernate 的配置文件 hibernate.cfg.xml，一个是实体类的映射文件。本章将对这两个文件的配置进行详细介绍以及如何使用 Annotations 配置映射，最后还将介绍 Hibernate 的 3 种对象状态及 Session 的各种方法及应用。

## 15.1　传统方式配置 Hibernate

Hibernate 的配置主要有两种方式：传统的配置方法以及使用 Annotations 进行配置的方法。我们首先来学习传统的配置方式。

### 15.1.1　配置 Hibernate

在使用 Hibernate 进行持久化之前，必须对 Hibernate 进行一系列配置，如数据库连接 URL、数据库用户名和密码以及映射文件路径等。对于 Hibernate 的配置，最常用的就是采用 XML 格式的方法进行配置。Hibernate 默认的配置文件为 hibernate.cfg.xml，其包含的配置属性如表 15.1 所示。

表 15.1　Hibernate配置属性表

| 属 性 名 | 描　　述 |
| --- | --- |
| hibernate.dialect | Hibernate 方言所对应的类名 |
| hibernate.show_sql | 设置是否在控制台输出 SQL 语句 |
| hibernate.connection.url | 设置数据库连接 URL |
| hibernate.connection.username | 设置数据库用户名 |
| hibernate.connection.password | 设置数据库密码 |
| hibernate.connection.driver_class | 设置数据库连接驱动类 |
| hibernate.default_schema | 生成 SQL 时，schema/tablespace 的全限定名 |

**【示例 15.1】**我们为大家提供一个 Hibernate 配置文件的模板文件 myHibernate.cfg.xml。在配置 Hibernate 时，只需要在模板文件中配置相应的属性值就可以了，具体代码如图 15.1 所示。

**注意：**使用 XML 文件配置 Hibernate 时，可以将属性名简写，如将 hibernate.show_sql 直接写为 show_sql。

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
          "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
      <session-factory>
            <property name="connection.url">数据库连接URL</property>
            <property name="connection.username">数据库用户名</property>
            <property name="connection.password">数据库密码</property>
            <property name="connection.driver_class">数据库连接驱动类</property>
            <property name="dialect">数据库方言类</property>
            <property name="show_sql">是否显示SQL语句</property>
            <mapping resource="映射文件路径1" />
            <mapping resource="映射文件路径2" />
      </session-factory>
</hibernate-configuration>
```

图 15.1　Hibernate 配置模板

## 15.1.2　配置映射文件

映射文件是持久化操作中的一个重点，它是数据库表和实体类之间的连接枢纽。通过映射文件，Hibernate 能知道实体类和哪个数据库表相对应。映射文件也是采用 XML 文档规范，这样设计可以使其非常易读，而且容易修改。下面对其包含的各种元素分别做一简要介绍。

### 1．<hibernate-mapping>元素

映射文件的根节点为<hibernate-mapping>，该节点包含一系列的可选属性，如 schema 和 catalog 属性。schema 属性指定了数据库表所在的 schema 名称。

表 15.2　<hibernate-mapping>属性列表

| 属 性 名 | 必　选 | 默 认 值 | 描　述 |
|---|---|---|---|
| schema | 否 | 无 | 指定数据库 schema 的名称 |
| catalog | 否 | 无 | 指定数据库 catalog 的名称 |
| default-cascade | 否 | none | 指定默认的级联风格 |
| default-access | 否 | property | 指定访问所有属性的策略 |
| default-lazy | 否 | true | 指定默认加载风格 |
| auto-import | 否 | true | 指定是否可以查询非全限定的类名 |
| package | 否 | 无 | 指定包前缀，若没有指定全限定的类名，将使用这个作为包名 |

**【示例 15.2】**根据表 15.2 中的属性，可以创建一份完整的<hibernate-mapping>配置代码，如图 15.2 所示。

```
<hibernate-mapping
    schema="schemaName"
    catalog="catalogName"
    default-cascade="cascade_style"
    default-access="field|property|ClassName"
    default-lazy="true|false"
    auto-import="true|false"
    package="package.name"
/>
```



图 15.2　<hibernate-mapping>元素属性配置

**注意：**一个映射文件中只允许有一个<hibernate-mapping>元素。

### 2．<class>元素

<class>元素用来配置一个实体类与一个数据库表的关联。其中 name 属性用来指定实体类的名称，table 属性用来指定数据库表的名称。<class>元素的常用属性如表 15.3 所示。

表 15.3　<class>元素常用属性

| 属 性 名 | 必　选 | 默 认 值 | 描　述 |
|---|---|---|---|
| name | 否 | 无 | 指定完全路径类名 |
| table | 否 | 无 | 指定数据库表名 |
| mutable | 否 | true | 指定类的实例是否可变 |
| proxy | 否 | 无 | 指定代理类接口，为延迟加载提供支持 |
| lazy | 否 | true | 指定是否使用延迟加载 |
| dynamic-update | 否 | false | 指定生成 Update SQL 时是否仅包含发生变动的字段 |
| dynamic-insert | 否 | false | 指定生成 Insert SQL 时，是否仅包含非空字段 |

**【示例 15.3】**根据表 15.3 中的属性，可以创建一份完整的<class>元素配置代码，如图 15.3 所示。

```
<class
    name="ClassName"
    table="tableName"
    mutable="true|false"
    proxy="proxyInterface"
    lazy="true|false"
    dynamic-update="true|false"
    dynamic-insert="true|false"
/>
```

<class>元素配置代码

图 15.3　<class>元素属性配置

**注意：**<hibernate-mapping>元素下可以有多个<class>元素，但是一般推荐只添加一个，即一个实体类对应一个映射文件。

### 3．<id>元素

每一个实体类中都包含一个唯一的标识，<id>元素能够定义该属性和数据库表中的主键字段的映射。<id>元素包括的常用属性如表 15.4 所示。

表 15.4　<id>元素常用属性

| 属 性 名 | 必　选 | 默 认 值 | 描　述 |
|---|---|---|---|
| name | 否 | 无 | 指定标识属性的名称，如果不指定，表示这个类没有标识属性 |
| type | 否 | 无 | 指定标识属性的 Hibernate 类型 |

续表

| 属 性 名 | 必　选 | 默 认 值 | 描　述 |
|---|---|---|---|
| column | 否 | 无 | 指定数据库表中主键字段的名称 |
| unsaved-value | 否 | 无 | 指定该实例是刚创建的，尚未进行保存 |
| access | 否 | property | 指定 Hibernate 用来访问属性值的策略 |

**【示例 15.4】**根据表 15.4 中的属性，可以创建一份完整的<id>元素配置代码，如图 15.4 所示。

```
<id
    name="PropertyName"
    type="typeName"
    column="column_name"
    unsaved-value="null|any|none|undefined|id_value"
    access="field|property|ClassName"
    <generator class="generatorClass"/>
</id>
```

<id>元素配置代码

图 15.4　<id>元素属性配置

### 4．<property>元素

实体类的标识和数据库表的主键映射完成后，还需要为实体类的其他属性和数据库的其他字段进行映射，这个时候需要使用到<property>元素。<property>元素的常用属性如表 15.5 所示。

表 15.5　<property>元素常用属性

| 属　性　名 | 必　选 | 默 认 值 | 描　述 |
|---|---|---|---|
| name | 否 | 无 | 指定标识属性的名称，如果不指定，表示这个类没有标识属性 |
| type | 否 | 无 | 指定标识属性的 Hibernate 类型 |
| column | 否 | 无 | 指定数据库表中主键字段的名称 |
| access | 否 | property | 指定 Hibernate 用来访问属性值的策略 |
| not-null | 否 | true | 指定属性是否允许为空 |
| generated | 否 | never | 指定属性值是否由数据库生成 |

**【示例 15.5】**根据表 15.5 中的属性，可以创建一份完整的<property>元素配置代码，如图 15.5 所示。

```
<property
    name="PropertyName"
    type="typeName"
    column="column_name"
    access="field|property|ClassName"
    not-null="true|false"
    generated="never|insert|always"
/>
```

<property>元素配置代码

图 15.5　<property>元素属性配置

【示例 15.6】下面看一个完整运用这些元素的例子。例如，创建一个公司职员映射配置文件 Employee.hbm.xm。其中，employeeID 为 Employee 类的标识，配置生成策略为 assigned，同时使用<property>元素配置 Employee 类的其他属性。具体代码如图 15.6 所示。

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping
                    PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
      <class name="com.sanqing.po.Employee" table="tb_employee">
            <id name="employeeID" type="string">
                  <generator class="assigned"></generator>
            </id>
            <property name="employeeName"></property>
            <property name="employeeSex"></property>
            <property name="employeeBirth"></property>
            <property name="employeePhone"></property>
            <property name="employeePlace"></property>
            <property name="joinTime"></property>
      </class>
</hibernate-mapping>
```

完整运用映射文件元素

图 15.6　公司职员映射配置文件

## 15.2　使用 Annotations 配置映射

在 JDK 5.0 之后出现了一种新的注释技术 Annotations，而 Hibernate 也在其 3.0 之后的版本中添加了对 Annotations 的支持。通过在实体类中添加 Annotations 注释，可以达到替代映射文件的效果。

### 15.2.1　使用@Entity 注释实体类

@Entity 注释用来将一个普通的 JavaBean 标注为实体类。@Entity 注释由一个可选的 name 属性，用来设置属性名。并不是所有的 JavaBean 都能被标注为实体类，必须要满足如图 15.7 所示的 3 个条件。

| | |
|---|---|
| 标注为实体类的3个条件 | JavaBean类的访问权限只能是public |
| | JavaBean类不能是抽象类 |
| | JavaBean中必须有一个访问权限为public的无参的构造方法 |

图 15.7　JavaBean 被标注为实体类的条件

【示例 15.7】我们举一个实例来演示如何使用@Entity 注释实体类，并通过 name 属性设置该实体名为 Employee，具体代码如图 15.8 所示。

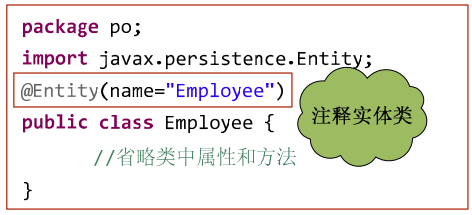



图 15.8　使用@Entity 注释实体类

**注意：**使用@Entity 注释实体类，一定要使用 import 语句引入 javax.persistence.Entity 类，该类为@Entity 注释依赖类。

## 15.2.2　使用@Table 注释实体类

@Table 注释用来对实体类进行进一步注释，用来配置实体类到数据库表映射的更详细的信息，@Table 注释包含的属性信息如表 15.6 所示。

表 15.6　@Table注释包含的属性信息

| 属 性 名 | 属 性 描 述 |
|---|---|
| catalog | 用来设置数据库名 |
| name | 用来设置数据库表名 |
| schema | 用来设置数据库表的所有者名称 |
| uniqueConstraints | 用来设置数据库表的约束 |

**【示例 15.8】**我们也举一个实例来演示如何使用@Table 注释实体类，并通过 name 属性设置数据库表名，catalog 属性设置其数据库名，具体代码如图 15.9 所示。


图 15.9　使用@Table 注释实体类

**注意：**使用@Table 注释实体类，一定要使用 import 语句引入 javax.persistence.Table 类，该类为@Table 注释依赖类。

## 15.2.3　使用@Id 注释实体类标识

@Id 注释用来对实体类的标识进行配置。一个实体类一般只有一个标识，所以一个实体类

中只出现一个@Id 注释。

【示例 15.9】我们也举一个实例来演示如何使用@Id 注释实体类标识，该标识为 Employee 的 employeeID 属性，用来表示雇员编号，具体代码如图 15.10 所示。

```
package po;
import javax.persistence.Entity;
import javax.persistence.Id;
import javax.persistence.Table;
@Entity
@Table(name = "tb_employee", catalog = "mysqltest")
public class Employee {
    @Id
    private String employeeID;
    public String getEmployeeID() {
        return employeeID;
    }
    public void setEmployeeID(String employeeID)
{
        this.employeeID = employeeID;
    }
}
```



图 15.10　使用@Id 注释实体类标识

**注意：** 使用@Id 注释实体类标识，一定要使用 import 语句引入 javax.persistence.Id 类，该类为@Id 注释依赖类。

### 15.2.4　使用@GenerateValue 注释覆盖标识的默认访问策略

使用@Id 注释实体类标识时将采用 Hibernate 的默认访问策略，这时可以使用@GenerateValue 注释覆盖标识的默认访问策略。@GenerateValue 注释包括两个属性，即使用 generator 属性指定标识生成器名，使用 strategy 属性指定标识生成策略。strategy 属性的属性值为一个枚举类型，其中包含了 4 个枚举值，如表 15.7 所示。

表 15.7　strategy属性的枚举值

| 枚 举 值 | 描 述 说 明 |
| --- | --- |
| javax.persistence.GeneratedType.AUTO | strategy 属性默认值，表示自动确定表示的类型 |
| javax.persistence.GeneratedType.IDENTITY | 用来表示由数据库自动设置标识的值，如自动递增字段 |
| javax.persistence.GeneratedType.SEQUENTITY | 用来表示标识为 SEQUENTITY 类型 |
| javax.persistence.GeneratedType.TABLE | 用来保证另一个使用该标识的表记录的唯一性 |

【示例 15.10】我们也举一个实例来演示如何使用@GenerateValue 注释来指定标识的生成策略，设置其生成策略为 javax.persistence.GeneratedType.IDENTITY，具体代码如图 15.11 所示。

```
public class Employee {
        @Id
        @GeneratedValue(strategy="javax.persistence.GeneratedType.IDENTITY")
        private String employeeID;
        public String getEmployeeID() {
                return employeeID;
        }
        public void setEmployeeID(String employeeID) {
                this.employeeID = employeeID;
        }
}
```

覆盖标识的默认访问策略

图 15.11　使用@GenerateValue 注释覆盖标识的默认访问策略

## 15.2.5　使用@GenericGenerator 注释生成标识生成器

前面我们介绍了如何使用@GenerateValue 注释的 strategy 属性来指定生成策略，但是这些生成策略明显不能满足。这时可以使用@GenericGenerator 注释产生标识生成器，然后通过@GenerateValue 注释的 generator 属性来制定生成器的 name 属性，这样就可以采用指定的生成器生成标识。

@GenericGenerator 注释包含 3 个属性，其属性说明如表 15.8 所示。

表 15.8　@GenericGenerator注释包含的属性信息

| 属 性 名 | 属 性 描 述 |
|---|---|
| name | 用来设置标识生成器名 |
| parameters | 用来设置标识生成器所需的参数 |
| strategy | 用来设置 Hibernate 内置的生成策略 |

**【示例 15.11】**我们也举一个实例来演示如何使用@GenericGenerator 注释来生成一个标识生成器，该生成器采用 UUID 生成策略，具体代码如图 15.12 所示。

```
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name = "tb_employee", catalog = "mysqltest")
public class Employee {
        @Id
        @GenericGenerator(name="myuuid",strategy="uuid")
        @GeneratedValue(generator="myuuid")
        private String employeeID;
        public String getEmployeeID() {
                return employeeID;
        }
        public void setEmployeeID(String employeeID) {
                this.employeeID = employeeID;
        }
}
```

生成标识生成器，确定UUID生成策略

图 15.12　使用@GenericGenerator 注释生成标识生成器

### 15.2.6 使用@Column 注释实体类非标识属性

一个实体类除了有标识，一般还会有许多其他属性，这时可以使用@Column 注释这些属性。@Column 注释最常用的属性为 name 属性，该属性用来设置数据库表中的字段名。

**【示例 15.12】**我们也举一个实例来演示如何使用@Column 注释来注释实体类中除标识外的其他属性，如 employeeName、employeeSex 等，具体代码如图 15.13 所示。

```
import javax.persistence.Column;
public class Employee {
        @Id
        @GenericGenerator(name="myuuid",strategy="uuid")
        @GeneratedValue(generator="myuuid")
        private String employeeID;
        @Column(name="employeeName")
        private String employeeName;
        @Column(name="employeeSex")
        private boolean employeeSex;
        @Column(name="employeeBirth")
        private Date employeeBirth;
        @Column(name="employeePhone")
        private String employeePhone;
        @Column(name="employeePlace")
        private String employeePlace;
        @Column(name="joinTime")
        private Date joinTime;
        //省略属性的get和set方法
}
```

使用@Column注释实体类非标识属性

图 15.13 使用@Column 注释实体类非标识属性

### 15.2.7 自定义 AnnotationSessionFactory 类获得 Session 对象

前面我们介绍了如何使用 Annotation 注释来完成实体类到数据库表的映射，这时还有一点需要特别注意。在以前获得 SessionFactory 对象是通过调用 Configuration 对象来实现的，但是这种方式不支持 Annotations 注释映射。要支持 Annotations 必须使用 AnnotationConfiguration 类。

**【示例 15.13】**下面的示例 AnnotationSessionFactory.java 自定义了一个 AnnotationSessionFactory 类，通过该类可以加载 Annotations 注释方式的映射配置，并返回 SessionFactory 以及 Session 对象。具体代码如图 15.14 所示。

```
package hibernate;
import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.cfg.AnnotationConfiguration;
public class AnnotationSessionFactory {
    private static String CONFIG_FILE_LOCATION = "/hibernate.cfg.xml";
    private static final ThreadLocal<Session> threadLocal
                                = new ThreadLocal<Session>();
    private  static AnnotationConfiguration configuration
                            = new AnnotationConfiguration();
    private static org.hibernate.SessionFactory sessionFactory;
    private static String configFile = CONFIG_FILE_LOCATION;
    //以下方法内容与 HibernateSessionFactory方法相同
}
```

定义Configuration和SessionFactory对象

图 15.14　自定义 AnnotationSessionFactory 类获得 Session 对象

## 15.2.8　测试 Annotations 注释是否成功完成映射

通过 AnnotationSessionFactory 可以加载 Annotations 注释方式的映射。通过调用 Session 对象的各种方法可以完成各类数据库操作，如查询记录、添加记录等。在创建测试类之前，首先同样需要在 Hibernate 的配置文件 hibernate.cfg.xml 中添加映射信息。同映射文件配置映射不同，这里需要指定 class 属性为需要映射的实体类，具体配置方法如图 15.15 所示。

```
<mapping class="po.Employee" />
```

图 15.15　配置 hibernate.cfg.xml

完成如上步骤后，映射信息已经全部完成，下面来创建一个测试类测试使用 Annotations 注释是否能成功完成映射。

首先打开 MySQL 数据库并在名为 mysqltest 的数据库中建立一个 tb_employee 表，并在表中按照前面所述的属性建立一条数据信息，具体过程如图 15.16 所示。

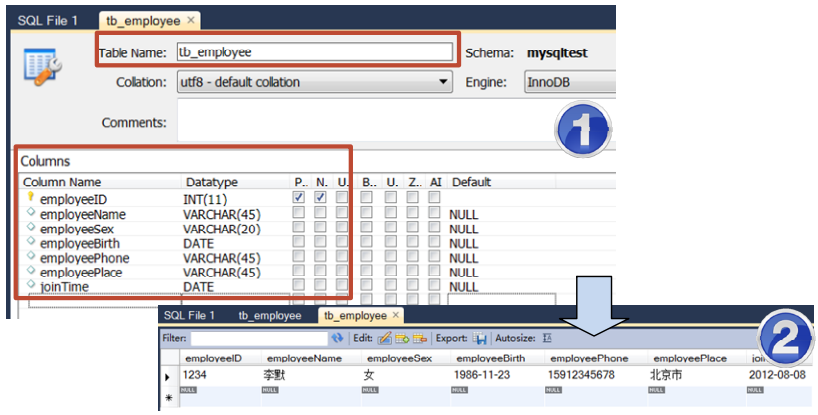



图 15.16　在 MySQL 数据库中增加数据

【示例 15.14】下面的示例 TestAnnotation.java 通过一个简单的记录查询，测试使用 Annotations 注释完成实体类到数据库表的映射是否成功，具体代码如图 15.17 所示。

```
package test;
import java.util.Iterator;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import hibernate.AnnotationSessionFactory;
import hibernate.HibernateSessionFactory;
import po.Employee;
import po.Student;
public class TestAnnotation {
       public static void main(String[] args) {
              Session session = AnnotationSessionFactory.getSession();
              Query query = session.createQuery("from Employee");
              List list = query.list();
              Iterator iter = list.iterator();
              Employee employee = null;
              System.out.println("职员编号\t职员姓名\t职员性别\t职员出生日期\t
              \t职员手机号码" +"\t职员居住地址\t职员进公司时间");
              while(iter.hasNext()) {
                 employee = (Employee) iter.next();
                 System.out.println(employee.getEmployeeID()+"\t"
                     + employee.getEmployeeName()+"\t"+
                     (employee.isEmployeeSex()?"男":"女")+"\t"
                           + employee.getEmployeeBirth()+"\t" +
                             employee.getEmployeePhone()+"\t"
                           + employee.getEmployeePlace()+"\t\t"+
                             employee.getJoinTime()
                     );
              }
              AnnotationSessionFactory.closeSession();
       }
}
```

引入各种包及资源文件

遍历List查询雇员信息

图 15.17　记录查询文件 TestAnnotation.java

运行该 Java 程序，在控制台输出如图 15.18 所示的结果。


图 15.18　控制台输出结果

## 15.3　会话（Session）的应用

Hibernate 提供了一个会话类 Session，可以通过 SessionFactory 获得 Session 实例对象。通过调用 Session 对象的方法即可完成数据库操作，如通过 save 方法插入记录、通过 load 方法按标识取出记录、通过 delete 方法删除记录等。

### 15.3.1　Hibernate 对象状态

一个实体类的实例可能处于 3 种不同的状态中的一种。这 3 种状态分别为瞬时状态、持久状态和托管状态。下面来看这 3 种状态的详细说明，如图 15.19 所示。

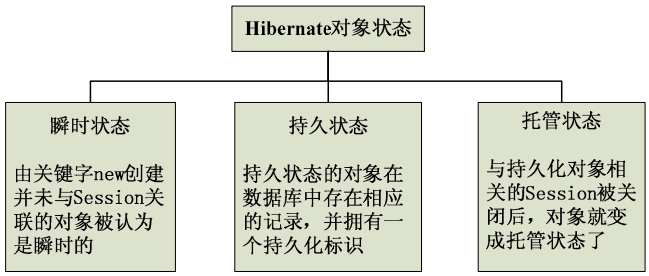



图 15.19　Hibernate 对象的 3 种状态

总结这 3 种状态，再结合各状态的转换方法可以得到其状态转换图，如图 15.20 所示。

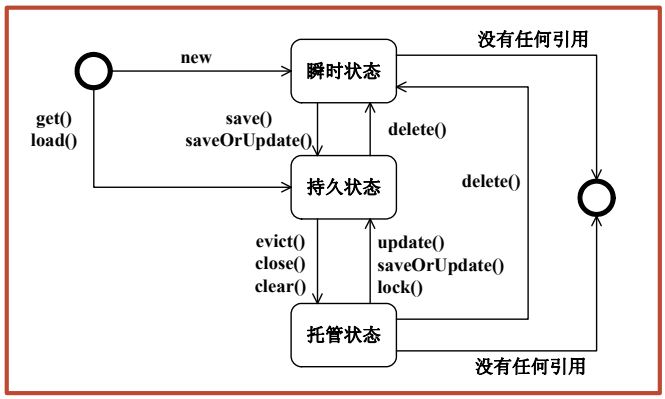



图 15.20　Hibernate 各对象状态转换图

### 15.3.2　使用 save 方法持久化对象

使用 new 关键字创建的对象并没有保存到数据库中，这时的对象为瞬时状态。通过 Session 对象的 save 方法能够将其转换成持久状态，并同时在数据库表中添加相应记录。save 方法有两种重载方式，如图 15.21 所示。

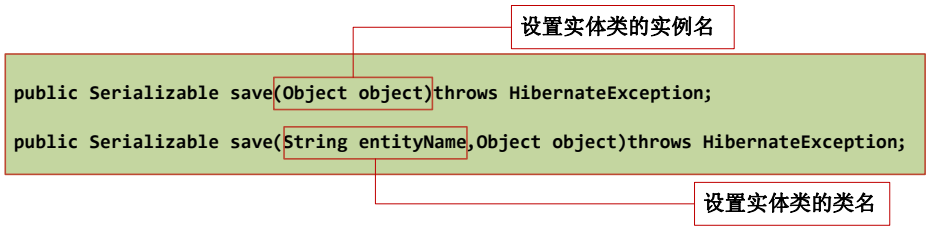



图 15.21　save 方法的重载方式

**【示例 15.15】**通过一个实例 TestSave.java 演示如何使用 Session 对象的 save 方法将瞬时状态的对象转换成持久状态，同时完成记录的添加，具体代码如图 15.22 所示。

运行该程序，记录成功添加到数据库表中，使用 MySQL 查看结果，如图 15.23 所示。

```
public class TestSave {
    public static void main(String[] args) throws ParseException {
        Session session = AnnotationSessionFactory.getSession();
        Employee employee = new Employee();
        employee.setEmployeeName("李健");
        employee.setEmployeeSex("男");
        employee.setEmployeeBirth(
          new SimpleDateFormat("yyyy-MM-dd").parse("1987-01-13"));
        employee.setEmployeePhone("13512345678");
        employee.setEmployeePlace("中国河北保定");
        employee.setJoinTime(new Date());
        Transaction transaction = null;
        transaction=session.beginTransaction();
        session.save(employee);
        transaction.commit();
        AnnotationSessionFactory.closeSession();
    }
}
```

建立一条新记录

对事务进行处理

图 15.22　TestSave.java 示例

SQL File 1　tb_employee

Filter:　Export:　Autosize:

| employeeID | employeeName | employeeSex | employeeBirth | employeePhone | employeePlace |
|---|---|---|---|---|---|
| 1234 | 李默 | 男 | 1986-11-23 | 15912345678 | 北京市 |
| 402882e539aa60e90139aa60ea510001 | 李健 | 男 | 1987-01-13 | 13512345678 | 中国河北保定 |

图 15.23　查看雇员信息

## 15.3.3　使用 load 方法装载对象

如果知道某个对象的持久化标识，可以使用 Session 对象的 load 方法从数据库中装载数据。使用 load 方法装载的对象是持久状态的。

**【示例 15.16】**通过一个实例 TestLoad.java 演示如何使用 Session 对象的 load 方法从数据库中装载数据，需要指定标识值和实体类所对应的 Class，具体代码如图 15.24 所示。

```
public class TestLoad {
    public static void main(String[] args) throws ParseException {
        Session session = AnnotationSessionFactory.getSession();
        Object obj = session.load(
              po.Employee.class,
              new String("402882e539aa60e90139aa60ea510001"));
        Employee employee = (Employee) obj;
        System.out.println("职员编号: " + employee.getEmployeeID());
        System.out.println("职员姓名: " + employee.getEmployeeName());
        System.out.println("职员性别: " + employee.getEmployeeSex());
        System.out.println("职员出生日期: " + employee.getEmployeeBirth());
        System.out.println("职员手机号码: " + employee.getEmployeePhone());
        System.out.println("职员居住地址: " + employee.getEmployeePlace());
        System.out.println("职员进公司时间: " + employee.getJoinTime());
        AnnotationSessionFactory.closeSession();//关闭Session对象
    }
}
```

获得Session对象并进行装载

图 15.24　TestLoad.java 示例

运行该 Java 程序，在控制台输出代码如图 15.25 所示。

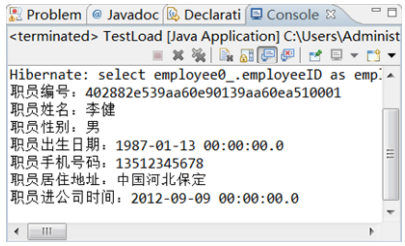



图 15.25　查看装载信息

## 15.3.4　使用 refresh 方法刷新对象

使用 refresh 方法能够根据数据库中的数据来刷新持久对象中的属性值。refresh 方法有两种重载方法，如图 15.26 所示。

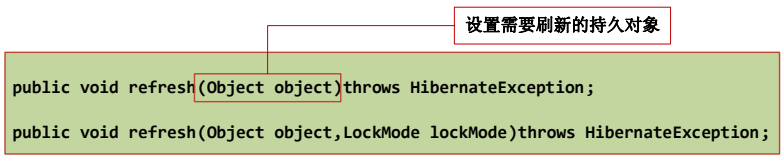



图 15.26　refresh 方法的重载方法

**【示例 15.17】**通过一个实例 TestRefresh.java 演示如何使用 Session 对象的 refresh 方法来刷新持久对象中的属性值，从而实现恢复数据，具体代码如图 15.27 所示。

```
public class TestRefresh {
    public static void main(String[] args) throws ParseException {
        Session session = AnnotationSessionFactory.getSession();
        Object obj = session.load(
                po.Employee.class,
                new String("402882e539aa60e90139aa60ea510001"));
        Employee employee = (Employee) obj;
        System.out.println("职员姓名(装载后)：" + employee.getEmployeeName());
        employee.setEmployeeName("李健儿");
        System.out.println("职员姓名(修改后)：" + employee.getEmployeeName());
        session.refresh(employee);
        System.out.println("职员姓名(刷新后)：" + employee.getEmployeeName());
        AnnotationSessionFactory.closeSession();
    }
}
```

获得Session对象并进行装载

图 15.27　TestRefresh.java 示例

运行该 Java 程序，在控制台输出代码如图 15.28 所示。

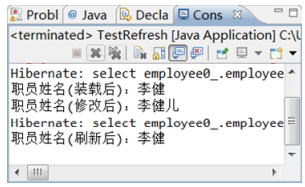



图 15.28　使用 refresh 方法刷新对象

### 15.3.5 使用 delete 方法删除对象

可以使用 Session 对象的 delete 方法来删除数据库中的记录。delete 方法有两个重载方式，如图 15.29 所示。

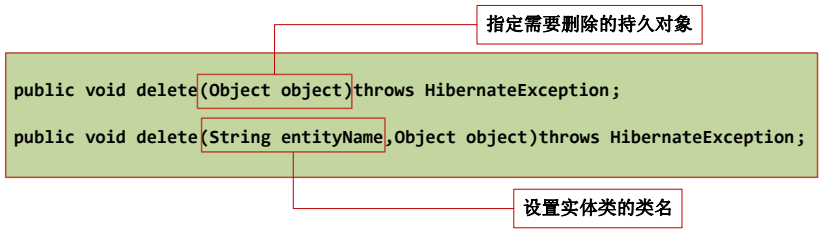



图 15.29 delete 方法的重载方式

**【示例 15.18】**通过一个实例 TestDelete.java 演示如何使用 Session 对象的 delete 方法来删除记录，并判断对象在删除前后是否为空，具体代码如图 15.30 所示。

```
public class TestDelete {
    public static void main(String[] args) throws ParseException {
        Session session = AnnotationSessionFactory.getSession();
        Object obj1 = session.get(
                po.Employee.class,
                new String("402882e539aa60e90139aa60ea510001"));
        System.out.println("持久对象是否为空: " + (obj1 == null));
        Transaction transaction = null;
        transaction=session.beginTransaction();
        session.delete(obj1);
        transaction.commit();
        Object obj2 = session.get(
                po.Employee.class,
                new String("402882e539aa60e90139aa60ea510001"));
        System.out.println("持久对象是否为空: " + (obj2 == null));
        AnnotationSessionFactory.closeSession();
    }
}
```



图 15.30 TestDelete.java 示例

运行该 Java 程序，在控制台输出代码如图 15.31 所示。

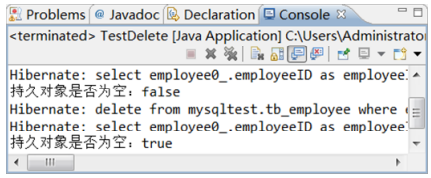



图 15.31 使用 delete 方法删除对象

## 15.4 小结

本章主要讲述了 Hibernate 的配置和会话方面的知识。熟练掌握 Hibernate 的配置和会话是

使用 Hibernate 操纵数据库的基础，也是学习高级 Hibernate 技术的基础。本章的重点内容是掌握使用 Annotations 配置映射的方法，难点内容是理解会话（Session）的应用的具体方法。希望读者在学习时多加练习，争取掌握。

## 15.5　本章习题

1．使用 XML 文件来配置 Hibernate，配置其数据库连接 URL 为“jdbc:mysql://localhost:3306/myDB”，数据库用户名为“root”，密码为“tiger”，数据库方言为“MySQL”，并设置不显示 SQL 语句。

【分析】本题意在考查读者配置 Hibernate 的能力。题目中已经给出了配置的方案，只要参考 15.1 所讲述的内容，不难将其配置出来。

【核心代码】本题的核心代码如下所示。

```
<?xml version='1.0' encoding='UTF-8'?>
……
<hibernate-configuration>
    <session-factory>
        <property name="dialect">
            org.hibernate.dialect.MySQLDialect<!-- 数据库方言 -->
        </property>
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/myDB<!-- 数据库连接 URL -->
        </property>
        <property name="connection.username">root</property>
        <!-- 数据库用户名 -->
        <property name="connection.password">tiger</property>
        <!-- 数据库用户密码 -->
        <property name="connection.driver_class">
            com.mysql.jdbc.Driver<!-- 数据库驱动类 -->
        </property>
        <property name="show_sql">false</property><!--显示 SQL 语句-->
    </session-factory>
</hibernate-configuration>
```

2．创建一个个人支出表 payout，该类包含 4 个字段，分别为支出编号、支出名称、支出额、支出时间，其中支出编号为主键，其生成策略为自动递增。并建立与之相对应的实体类 Payout。使用 Session 的 save 方法来添加一条记录，其中支出名称为“外出旅游”，支出金额为“2000”，支出时间为“2012-10-1”。执行效果如图 15.32 所示。

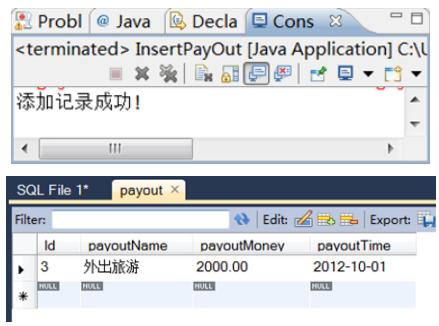



图 15.32　运行结果

【分析】本题意在考查读者使用 Hibernate 以及会话应用的能力。题目中已经给出了增加的内容，只要参考 15.1 和 15.3.2 所讲述的内容，不难将其运行出来。

【核心代码】本题的核心代码如下所示。

payout.sql：

```
CREATE TABLE `payout` (
  `Id` int(11) NOT NULL AUTO_INCREMENT,
  `payoutName` varchar(20) DEFAULT NULL,
  `payoutMoney` double(7,2) DEFAULT NULL,
  `payoutTime` date DEFAULT NULL,
  PRIMARY KEY (`Id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT CHARSET=gbk;
```

PayOut.java：

```
package com.sanqing.po;
import java.util.Date;
public class PayOut {
    private int Id;
……
    public int getId() {
        return Id;
    }
    public void setId(int id) {
        Id = id;
    }
    ……
}
```

PayOut.hbm.xml：

```
<?xml version="1.0" encoding="utf-8"?>
……
<hibernate-mapping>
    <class name="com.sanqing.po.PayOut"><!-- 每个 class 对应一个持久化对象 -->
        <id name="id" type="int">
            <generator class="native"></generator>
        </id>
        <property name="payoutName"></property>
……
    </class>
</hibernate-mapping>
```

InsertPayOut.java：

```
package test;
import java.text.ParseException;
……
public class InsertPayOut {
    public static void main(String[] args) {
        PayOut payout = new PayOut();
        payout.setPayoutName("外出旅游");
        payout.setPayoutMoney(2000);
        try {
            payout.setPayoutTime(new SimpleDateFormat("yyyy-MM-dd").parse("2012-
10-1"));
        } catch (ParseException e) {
            e.printStackTrace();
        }
        Session session =
            HibernateSessionFactory.getSession();//获得 Session 对象
        Transaction  transaction = null;//声明一个事务对象
    ……
```

```
            HibernateSessionFactory.closeSession();
        }
    }
```

3．使用 Annotations 配置映射的方式改写题目 2，通过添加一个记录来测试 Annotations 配置映射是否成功。

【分析】本题意在考查读者使用 Annotations 配置映射的能力。使用 Annotations 配置映射，是另一种不同的配置映射的方式。

【核心代码】本题的核心代码如下所示。

```
package po;
import java.util.Date;
……
import org.hibernate.annotations.GenericGenerator;
@Entity
@Table(name = "payout", catalog = "mysqltest")
public class PayOutAnnotation {
    @Id
    @GeneratedValue(strategy=javax.persistence.GenerationType.IDENTITY)
    private int Id;
    ……
    public int getId() {
        return Id;
    }
    public void setId(int id) {
        Id = id;
    }
    ……
}
```

PART 4

# Spring 技术篇

# 第16章 Spring 框架入门

SSH 框架是目前最为流行的软件开发技术，它是由 3 种技术组成的，除了我们前面讲解的 Struts 和 Hibernate 外，还包括从本章开始讲解的 Spring，如图 16.1 所示。Spring 是一种非常完善的开源的框架，通过它可以大大降低企业应用程序的复杂性。我们在开发中通常使用 Spring 开发业务逻辑层。


图 16.1　Spring 框架标识

## 16.1　Spring 概述

如果读者在学习 Spring 之前，学习过 EJB 技术，就知道开发企业级项目是一件非常复杂的工程。随着 Spring 的出现，会大大降低 J2EE 企业级开发的复杂度。作为一种开源技术，Spring 几乎替代了 EJB 技术。并且 Spring 不仅仅是替代品，其技术范围比 EJB 更广、更实用。

### 16.1.1　Spring 技术介绍

Spring 是一种非常完整的技术，即使只使用 Spring 技术也能实现项目的开发。但是在实际开发中，我们只是让 Spring 做业务逻辑层，因为 Spring 的业务处理能力是非常强大的。简单来说，Spring 就是一个轻量级的控制反转（IoC）和面向切面（AOP）的容器框架，如图 16.2 所示。

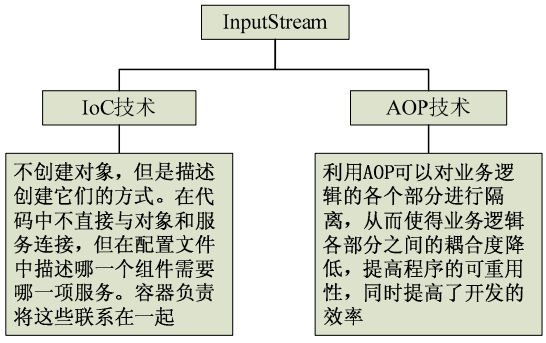



图 16.2　Spring 技术

### 16.1.2　为什么使用 Spring

在没有使用 Spring 之前，如果在业务逻辑层中访问数据访问层，首先需要在业务逻辑层中创建数据库访问层的对象，然后使用该对象调用 DAO 方法。

**【示例 16.1】**我们举一个这样的例子如图 16.3 所示。

```
public class UserManagerImpl implements UserManager {
        private UserDao userDao = new UserDaoImpl();
        public void save(String username,String password){
                this.userDao.save(username,password);
        }
}
```

业务层和数据层耦合非常紧

图 16.3　未使用 Spring 访问数据层

使用这种方式访问数据访问层，当数据访问层程序发生改动时，还需要改动业务访问层的程序，加大了程序员的工作量。

当 Spring 出现以后，这种问题得到了解决。业务逻辑层和数据访问层之间是注入的关系，在业务逻辑层中并不需要创建数据访问层的对象，如图 16.4 所示。

```
public class UserManagerImpl implements UserManager {
        private UserDao userDao;
        public UserManagerImpl(UserDao userDao){
                this UserDao = userDao;
        }
        public void save(String username,String password){
                this.userDao.save(username,password);
        }
}
```

业务层和数据层彻底分离

图 16.4　使用 Spring 访问数据层

## 16.2　Spring 开发环境的搭建

在 MyEclipse 中集成了 Spring 项目开发，通过它可以非常容易的搭建 Spring 开发环境。其搭建步骤大致如下：

（1）首先创建一个 Java 项目 ch16，选中该项目单击右键，在弹出的菜单栏选择“MyEclipse”→“Add Spring Capabilities”命令，然后选择版本号及所要添加的包。我们可以选择导入所有的 jar 包，如图 16.5 所示。

（2）单击“Next”按钮，进入 Spring 设置页面。在如图 16.6 所示的页面中，可以设置 Spring 的配置文件名和保存目录。在默认情况下，Spring 配置文件名为 applicationContext.xml，建议读者不要修改这个文件名。可以单击“Finish”按钮完成配置。

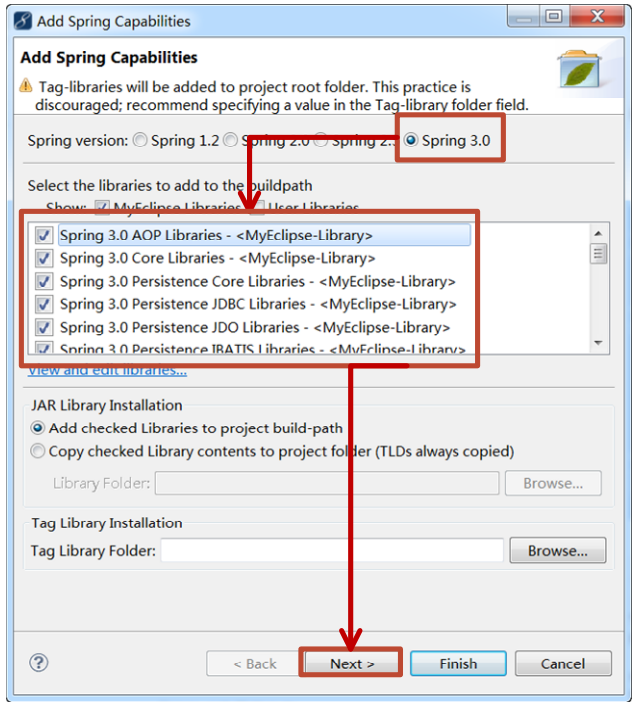



图 16.5　设置 Spring 版本和所使用的库

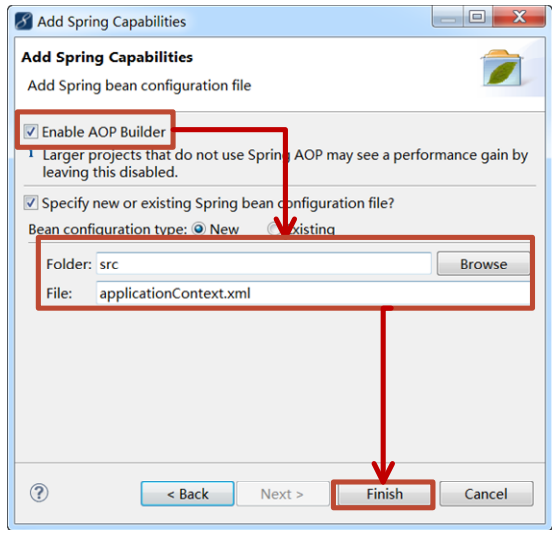



图 16.6　设置 Spring 的配置文件名和保存目录

## 16.3　开发 Spring 的 HelloWorld 程序

在前面的学习中，我们已经对 Spring 有了一个初步的了解，本节将通过一个非常简单的 HelloWorld 程序为大家介绍如何使用 Spring 开发环境进行程序的开发。Spring 有两个非常重要的应用，那就是 IoC 控制反转和 AOP 面向切面编程。这里先以 IoC 技术为代表进行讲解。

### 16.3.1　开发 Spring 程序的步骤

Spring 开发是有严格步骤的，无论项目简单和复杂，都要按照这个步骤进行操作。Spring 程序的开发步骤如图 16.7 所示。

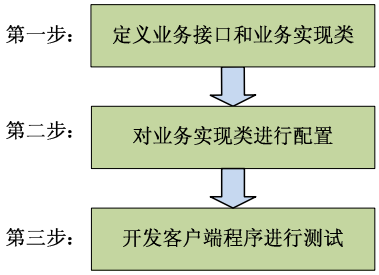



图 16.7　Spring 程序的开发步骤

### 16.3.2　编写业务接口

首先来开发业务接口，在该业务接口中定义了 SayHello 方法。通过该方法创建一个接收传递信息，然后返回问候语句的功能。

**【示例 16.2】**在这个程序 HelloService.java 中，我们主要学习如何编写 Spring 项目中的业务接口，通过该接口定义一个 SayHello 抽象方法来让业务类实现，具体代码如图 16.8 所示。


图 16.8　编写业务接口

### 16.3.3　编写业务实现类

开发完业务接口后，继续编写业务实现类。业务实现类要实现业务接口，从而实现业务接口中的抽象方法。

**【示例 16.3】**通过这个程序 HeloServiceImpl.java，我们学习如何编写 Spring 项目中的业务实现类。通过该类实现 SayHello 业务接口，从而让客户端调用该方法，具体代码如图 16.9 所示。

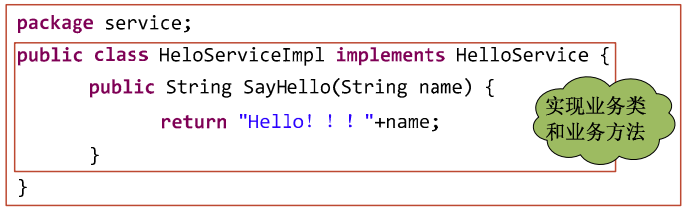



图 16.9　编写业务实现类

### 16.3.4 配置业务实现类

本节主要讲解通过 Spring 的方式来创建业务实现类对象，该步骤被称为配置业务实现类。配置业务实现类，需要在项目的 src 目录下创建一个 Spring 的配置文件，通常命名为 applicationContext.xml。

**【示例 16.4】**通过这个程序 applicationContext.xml，我们学习如何编写 Spring 项目中的配置文件。通过该配置文件来对 Spring 中的业务实现类进行配置，具体代码如图 16.10 所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="hello" class="service.HeloServiceImpl"></bean>

</beans>
```

配置业务实现类

图 16.10 applicationContext.xml 配置文件

### 16.3.5 编写客户端进行测试

到目前为止，Spring 的程序已经开发完毕，本节通过一个客户端程序来对 Spring 的程序进行测试。通过该客户端程序调用业务实现类中的业务方法。

**【示例 16.5】**通过这个程序 SpringClient.java，我们学习如何编写 Spring 项目中的客户端程序。通过该程序来对前面开发的 Spring 程序进行测试，具体代码如图 16.11 所示。

```
package client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.HelloService;
public class SpringClient {
   public static void main(String[] args) {
       BeanFactory factory =
           new ClassPathXmlApplicationContext("applicationContext.xml");
       HelloService helloService=(HelloService)factory.getBean("hello");
       String name="world";
       System.out.println(helloService.SayHello(name));
   }
}
```

获得业务实现类，调用业务方法

图 16.11 SpringClient.java 示例

运行上述代码，在控制台输出运行结果如图 16.12 所示。

学习到这里，读者可能并没有感受到 Spring 的开发优势，这是因为我们还没有进行实际开发。在 Web 项目开发中，最重要的一点就是进行分层开发，而 Spring 起到了这个作用。当我

们的程序需要改动时，只需要改动 Spring 的配置文件，这在以前的方式中是不可能做到的。

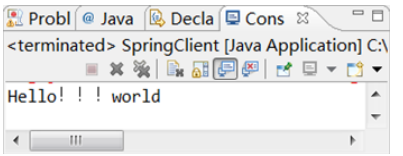



图 16.12　运行结果示意图

## 16.4　小结

本章是 Spring 的入门章节，首先为大家介绍 Spring 技术，以及为什么要使用 Spring，然后介绍了 Spring 开发环境的搭建，最后又通过一个完整的实例，向大家展示了开发 Spring 程序的步骤。本章的重点和难点都是理解并争取掌握 Spring 的开发过程，力争为后面章节的学习打下坚实的基础。

## 16.5　本章习题

1．开发一个计算两个参数平均值的业务实现类，并通过 Spring 的方法使用。

【分析】本题主要考查读者编写业务接口以及编写业务实现类的能力。编写接口是为了业务实现类能够继承并实现特定的功能。

【核心代码】本题的核心代码如下所示。

PJService.java：

```
package service;
public interface PJService {
    public int getPJ(int a,int b);
}
```

PJServiceImpl.java：

```
package service;
public class PJServiceImpl implements PJService {
    @Override
    public int getPJ(int a, int b) {
        return (a+b)/2;
    }
}
```

2．开发一个业务接口，在其中定义增加、删除、更新和查询用户的方法。开发一个实现类实现该接口，并且重写业务接口中对用户的操作方法，在其中仅适用显示语句表示运行该方法。

【分析】本题主要考查读者编写业务接口以及编写业务实现类的能力。编写接口是为了业务实现类能够继承并实现特定的功能。

【核心代码】本题的核心代码如下所示。

UserService.java：

```
package service;
import java.util.List;
public interface UserService {
```

```
    public boolean addUser(String name,String password);
……
}
```

UserServiceImpl.java：

```
package service;
import java.util.List;
public class UserServiceImpl implements UserService {
    @Override
    public boolean addUser(String name, String password) {
……
    @Override
    public boolean delectUser(String name) {
        boolean b=false;
……
    @Override
    public List findAllUser() {
        System.out.println("findall");
        return null;
    }
    @Override
    public boolean updateUser(String name, String password) {
        boolean b=false;
……
}
```

3．编写客户端测试程序，在其中定义用户测试的用户名和密码，从而定义业务方法，对用户业务模块进行测试。

【分析】本题主要考查读者编写客户端测试程序的能力。客户端测试程序是为了对创建的业务实现类进行测试，查看程序是否能正确运行。

【核心代码】本题的核心代码如下所示。

UserClient.java：

```
package client;
import org.springframework.beans.factory.BeanFactory;
……
public class UserClient {
    public static void main(String[] args) {
        // TODO Auto-generated method stub
       BeanFactory factory = new ClassPathXmlApplicationContext("applicationContext.
xml");
        UserService userService=(UserServiceImpl)factory.getBean("userManager");
        userService.addUser("Tom", "123");
    }
}
```

# 第17章 控制反转

控制反转是 Spring 的核心技术之一，英文名称为“Inversion of Control”，所以一般将其称为 IoC。Spring 的很多功能都是基于控制反转技术的。上一章中的 HelloWorld 程序就是基于控制反转技术的。本章我们将继续学习这一核心技术。

## 17.1 IoC 容器

容器的字面意思可以理解为用于存放其他事物的物品。在 Spring 中，IoC 容器除了具有该含义以外，它还可以对其中的事物进行管理。在 Spring 中，IoC 容器中的事物被称为 Bean，它是由 IoC 容器初始化、装配和管理的对象。

### 17.1.1 Bean 工厂接口

Bean 工厂接口的全称为“org.springframework.beans.factory.BeanFactory”。在 Spring 程序中，BeanFactory 实际上是 IoC 的代表者，通过使用该接口对其中的 Bean 进行管理。在 Spring 中，定义了多种 Bean 工厂接口的实现类，其中最常用的就是 XmlBeanFactory，使用该类可以以 XML 文件的形式来描述对象与对象之间的关系。Bean 工厂接口的引入方式如图 17.1 所示。

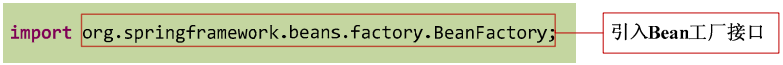



图 17.1 Bean 工厂接口

### 17.1.2 实例化容器

要想使用 IoC 容器，需要对容器进行实例化操作。通过实例化后的容器对象可以访问其中的 Bean 对象。实例化容器有多种方式，如图 17.2 所示。

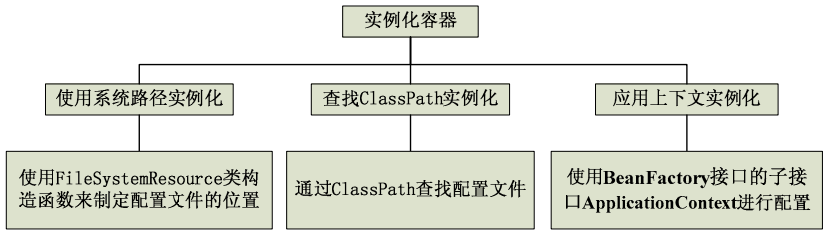



图 17.2 实例化容器的方式

【示例 17.1】通过一个 SpringClient.java 程序，我们来学习如何实例化容器。这里我们采用 3 种方式实现，并且都调用 SayHello 业务方法，具体代码如图 17.3 所示。

```
package client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.HelloService;
public class SpringClient {
    public static void main(String[] args) {
        /*Resource resource=new FileSystemResource("src\\applicationContext.xml");
        BeanFactory factory=new XmlBeanFactory(resource);*/
        /*ClassPathResource resource=new ClassPathResource("applicationContext.xml");
        BeanFactory factory=new XmlBeanFactory(resource);*/
        ApplicationContext context=new
                ClassPathXmlApplicationContext("applicationContext.xml");
        BeanFactory factory=context;
        HelloService helloService=(HelloService)factory.getBean("hello");
        String name="world";
        System.out.println(helloService.SayHello(name));
    }
}
```

使用系统路径实例化

查找ClassPath实例化

应用上下文实例化

图 17.3　SpringClient.java 示例

## 17.1.3　多配置文件的使用

当开发的项目较大时，使用一个 Spring 配置文件是比较复杂的，其中可能具有几千行的配置信息，在其中查找某一个 Bean 对象的配置是非常困难的。所以在 Spring 中可以定义多个配置文件，将一类配置放在一个文件中。

【示例 17.2】我们可以使用 ApplicationContext 实现类的重载构造函数，将所有配置文件的路径组成字符串数据作为参数，具体代码如图 17.4 所示。

```
ApplicationContext context = new ClassPathXmlApplicationContext(new String[]{
                                        "applicationContext.xml"
                                    "applicationContext-part2.xml"});
BeanFactory factory = context;
```

重载构造函数

图 17.4　多配置文件的使用

## 17.1.4　使用容器实例化 Bean

前面我们已经见过如何配置 Bean。配置 Bean 是通过使用<bean>标记对其进行配置的，其中 class 属性指定 Bean 的完整路径。每一个 Bean 中都有一个或者多个 id 属性，id 属性是 Bean 的标识符，在当前 IoC 容器中必须是唯一的。

对容器实例化后，使用容器对象调用 getBean 方法，就可以实例化指定的 Bean 了，具体示例如图 17.5 所示。

```
HelloService helloService=(HelloService)factory.getBean("hello");
```

图 17.5　使用容器实例化 Bean

## 17.2　依赖注入

在开发 HelloWorld 程序时，我们只是将业务实现类配置到 Spring 配置文件中。但是在实际开发中并不是这样的，除了业务实现类外，还有 DAO 实现类和控制层类，在项目中它们要结合使用，这时候就要使用到 Spring 中的依赖注入技术。依赖注入有两种常用的方式，分别是 Setter 方法注入和构造函数注入。

### 17.2.1　Setter 方法注入

使用 Setter 方法进行注入是依赖注入的一种方式。在使用 Bean 中定义需要注入属性的 Setter 方法，然后在 Spring 配置文件中给出该属性的值，最后在客户端程序中可以访问这些属性值。我们先来看一个完整的程序。

**【示例 17.3】**我们开发一个学生类 Student.java，在其中定义名称和年龄属性，并且通过 Setter 方法设置属性值，具体代码如图 17.6 所示。

```
package service;
public class Student
{
        String name;
        int age;
        public void setName(String name) {
                this.name = name;
        }
        public void setAge(int age) {
                this.age = age;
        }
        public String introduction(){
                return "我的名称为："+name+",年龄为:"+age;
        }
}
```

使用Setter设置姓名和年龄

图 17.6　Student.java 示例

然后对这个学生类在 Spring 配置文件 applicationContext.xml 中进行配置，并通过 Setter 方法为其中的属性注入值，具体代码如图 17.7 所示。

```
<bean id="student" class="service.Student">
        <property name="name">
                <value>Lester</value>
        </property>
        <property name="age">
                <value>24</value>
        </property>
</bean>
```

注入属性值

图 17.7　applicationContext.xml 配置文件

最后通过一个客户端程序 SetterClient.java，测试通过 Setter 方法的方式是否能够将值注入到 Bean 中，具体代码如图 17.8 所示。

```
package client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.Student;
public class SetterClient {
	public static void main(String[] args) {
		BeanFactory factory = new
		ClassPathXmlApplicationContext("applicationContext.xml");
		Student student=(Student)factory.getBean("student");
		System.out.println(student.introduction());
	}
}
```

获取学生类，调用业务方法

图 17.8　SetterClient.java 示例

运行这个 Java 程序，在控制台输出运行结果如图 17.9 所示。

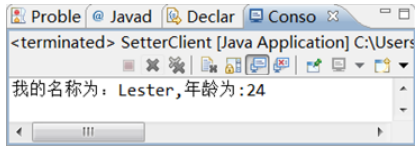



图 17.9　Setter 方法运行结果

## 17.2.2　构造函数注入

虽然在 Spring 中提倡使用 Setter 方法的方式进行注入，但是使用构造函数同样是依赖注入非常重要的方式之一。有些程序员更倾向于使用构造函数的方式进行注入，是因为使用这种方式可以将所有的属性一次性注入。我们通过一个具体的实例，看如何实现构造函数的注入。

**【示例 17.4】**通过这个程序，我们首先开发一个老师类 Teacher.java，在其中和学生类一样定义名称和年龄属性，但是这里是通过构造函数的方式注入属性值的，具体代码如图 17.10 所示。

```
package service;
public class Teacher
{
	String name;
	int age;
	public Teacher() {
	}
	public Teacher(String name, int age) {
		this.name = name;
		this.age = age;
	}
	public String introduction(){
		return "我的名称为："+name+",年龄为:"+age;
	}
}
```

定义有参构造函数设置姓名年龄属性

图 17.10　Teacher.java 示例

然后对这个老师类在 Spring 配置文件 applicationContext.xml 中进行配置，并通过构造函数方法为其中的属性注入值，具体代码如图 17.11 所示。

```
<bean id="teacher" class="service.Teacher">
        <constructor-arg index="0">
                <value>李默</value>
        </constructor-arg>
        <constructor-arg index="1">
                <value>35</value>
        </constructor-arg>
</bean>
```

注入属性值

图 17.11　applicationContext.xml 配置文件

最后通过一个客户端程序 ConstructorClient.java，测试通过构造函数的方式是否能够将值注入到 Bean 中，具体代码如图 17.12 所示。

```
package client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.Teacher;
public class ConstructorClient {
      public static void main(String[] args) {
            BeanFactory factory = new
               ClassPathXmlApplicationContext("applicationContext.xml");
            Teacher teacher=(Teacher)factory.getBean("teacher");
            System.out.println(teacher.introduction());
      }
}
```

获取学生类，调用业务方法

图 17.12　ConstructorClient.java 示例

运行这个 Java 程序，在控制台输出运行结果如图 17.13 所示。

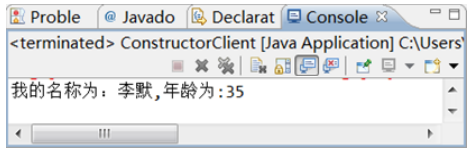



图 17.13　构造函数注入运行结果

## 17.2.3　注入其他 Bean

我们还可以将 Bean 中指定属性的值设置为对容器中的另外一个 bean 的引用的方式，即可以在一个 Bean 中注入其他 Bean。该操作是通过<ref>标记对完成，将其放在<constructor-arg>标记对或者<property>标记对中的。

**【示例 17.5】**在实际开发中注入其他 Bean 使用的最多的地方是将 DAO 注入到业务实现类中，下面举一个这样的例子为大家进行讲解。

首先，建立一个数据访问层的程序 UserDAOImpl.java，在其中定义了访问数据库的方法。

为了简单起见，我们使用了固定字符来表示。该程序也要注入 Bean，具体代码如图 17.14 所示。

```
package dao;
public interface UserDAO {
       public boolean findUser(String name,String password);
}
```

定义UserDAO类

```
package dao;
public class UserDAOImpl implements UserDAO {
      public boolean findUser(String name, String password) {
            if("Lester".equals(name) && "123456".equals(password)){
                  return true;
            }
            else{
                  return false;
            }
      }
}
```

判断是否为指定内容

图 17.14　UserDAOImpl.java 程序示例

然后创建一个业务层的实现类程序 UserServiceImpl.java，也是当前操作的 Bean 程序。在其中使用 Setter 方法的方式将 DAO 类注入进来，就可以指定查找数据库的方法了，具体代码如图 17.15 所示。

```
package service;
public interface UserService {
      public boolean Login(String name,String password);
}
```

定义UserService类

```
package service;
import dao.UserDAO;
public class UserServiceImpl implements UserService {
      private UserDAO userDAO;
      public void setUserDAO(UserDAO userDAO) {
            this.userDAO = userDAO;
      }
      public boolean Login(String name, String password) {
            boolean b=userDAO.findUser(name, password);
            return b;
      }
}
```

定义登录业务方法并调用DAO方法

图 17.15　UserServiceImpl.java 程序示例

同样也要在 Spring 配置文件 applicationContext.xml 中对这些类进行配置，并在其中将 DAO 程序注入到对用户操作的业务实现类中，具体代码如图 17.16 所示。

```
<bean id="UserDAOImpl" class="dao.UserDAOImpl">
        </bean>
<bean id="userService" class="service.UserServiceImpl">
        <property name="userDAO">
                <ref bean="UserDAOImpl"/>
        </property>
 </bean>
```

图 17.16　applicationContext.xml 配置文件

最后通过一个客户端程序 OtherBeanClient.java 测试注入其他 Bean 是否成功，在其中给定一个用户名和密码判断是否能够登录，具体代码如图 17.17 所示。

```
package client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.UserService;
public class OtherBeanClient {
    public static void main(String[] args) {
       BeanFactory factory = new
            ClassPathXmlApplicationContext("applicationContext.xml");
       UserService userService=(UserService)factory.getBean("userService");
       String name="Lester";
        String password="123456";
       System.out.println(name+"是否能够正常登录："
             +userService.Login(name, password));
    }
}
```

调用业务方法测试用户名和密码

图 17.17　OtherBeanClient.java 代码示例

运行这个 Java 程序，在控制台输出运行结果如图 17.18 所示。

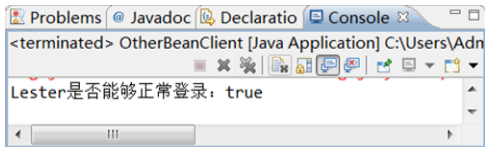
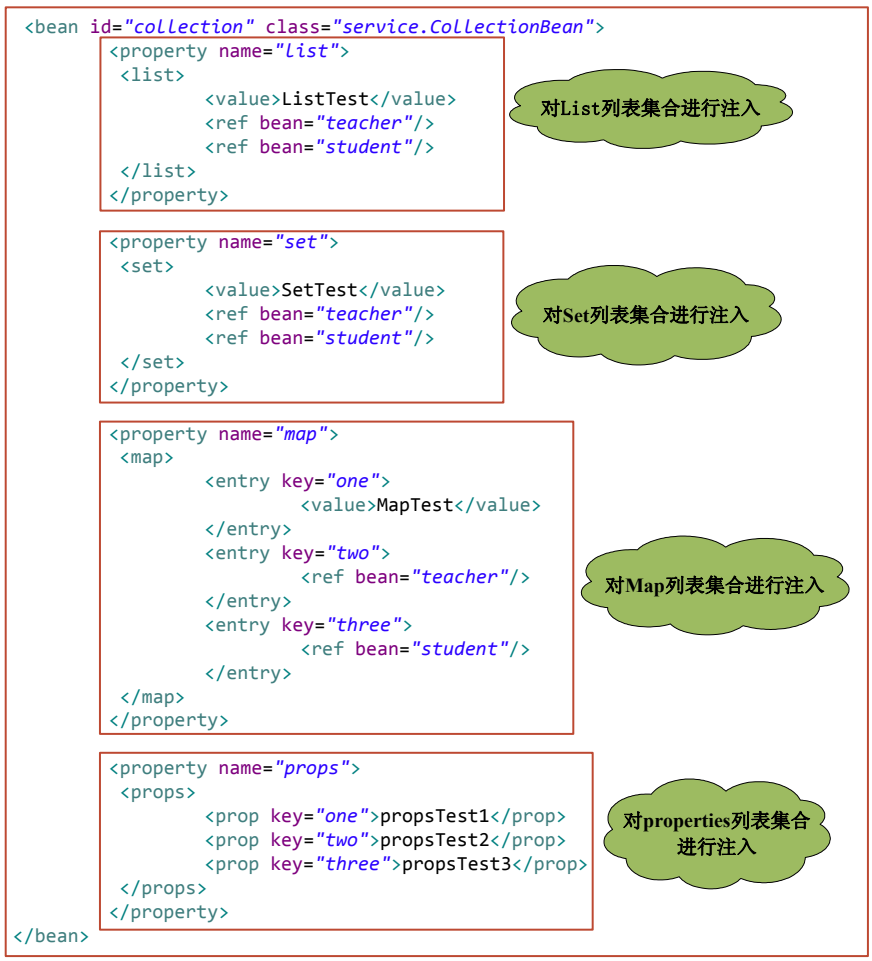



图 17.18　注入其他 Bean 运行结果示意图

## 17.2.4　注入集合

在 Java 中，包括 3 种集合 List、Set 和 Map。Spring 除了能够对这 3 种集合进行注入操作外，还包括 Properties 文件。在 Spring 配置文件中，使用<list/>、<set/>、<map/>和<props/>标记来对应集合。

**【示例 17.6】**我们举一个例子来看如何对 Spring 配置文件进行配置，才能将 List、Set、Map 和 Properties 注入到 Bean 中，具体代码如图 17.19 所示。




```
<bean id="collection" class="service.CollectionBean">
        <property name="list">
         <list>
                <value>ListTest</value>
                <ref bean="teacher"/>
                <ref bean="student"/>
         </list>
        </property>
        <property name="set">
         <set>
                <value>SetTest</value>
                <ref bean="teacher"/>
                <ref bean="student"/>
         </set>
        </property>
        <property name="map">
         <map>
                <entry key="one">
                        <value>MapTest</value>
                </entry>
                <entry key="two">
                        <ref bean="teacher"/>
                </entry>
                <entry key="three">
                        <ref bean="student"/>
                </entry>
         </map>
        </property>
        <property name="props">
         <props>
                <prop key="one">propsTest1</prop>
                <prop key="two">propsTest2</prop>
                <prop key="three">propsTest3</prop>
         </props>
        </property>
</bean>
```

图 17.19　applicationContext.xml 集合的配置

## 17.3　Bean 作用域

在 Spring 配置文件中创建 Bean 时，还要设置 Bean 的作用域。Bean 的作用域就是指一个 Bean 对象的使用方式和使用范围。Spring 中的这种设计使定义 Bean 的作用域不用在类程序中进行，而是在配置文件中进行，大大提高了灵活性。在 Spring 中，支持的 5 种内置 Bean 作用域如表 17.1 所示。

表 17.1　Bean作用域及其描述

| 作用域名称 | 作用域说明 |
| --- | --- |
| singleton | Bean 默认作用域，每个 IoC 容器中，一个 Bean 定义对应一个实例对象 |
| prototype | 每个 IoC 容器中，一个 Bean 定义对应多个实例对象 |
| request | 在一次 HTTP 请求中，一个 Bean 定义对应一个实例对象 |
| session | 在一个 HTTP Session 中，一个 Bean 定义对应一个实例对象 |
| global session | 在一个全局的 HTTP Session 中，一个 Bean 定义对应一个实例对象 |

其中 request、session 和 global session 的作用域只在 Web 的项目中有效。本节将对这几种作用域进行详细讲解。

### 17.3.1　singleton 作用域

singleton 作用域是 Bean 的默认作用域。当一个 Bean 的作用域为 singleton 时，在 Spring IoC 容器中只会存在一个共享的 Bean 实例。也就是说，所有对 Bean 的请求，只要 id 与该 Bean 定义相匹配，就会返回 Bean 的同一实例。singleton 作用域的定义方式如图 17.20 所示。

```
<bean id="teacher" class="service.teacher" scope="singleton" />
<bean id="teacher" class="service.teacher" singleton="true" />
```

图 17.20　singleton 作用域的定义方式

### 17.3.2　prototype 作用域

prototype 作用域是和 singleton 作用域相对的。当一个 Bean 设置为 prototype 作用域时，则该 Bean 每被请求或者引用一次，都会创建一个 Bean 实例对象。prototype 作用域的定义方式如图 17.21 所示。

```
<bean id="teacher" class="service.teacher" scope="prototype" />
<bean id="teacher" class="service.teacher" singleton="false" />
```

图 17.21　prototype 作用域的定义方式

### 17.3.3　request 作用域

request、session 和 global session 的作用域和前面两种作用域有很大不同，它们只能应用在 Web 的项目中。Web 中容器实例化和前面学过的实例化也是有所不同的，它要使用 XmlWebApplicationContext 来查找 Spring 配置文件。在使用 Bean 时，要对 web.xml 文件进行如图 17.22 所示的配置。

```
<web-app>
   <listener>
       <listener-class>org.springframework.web.context.request.RequestContextListener
       </listener-class>
   </listener>
 </web-app>
```

图 17.22　web.xml 文件的配置

在其中配置了监听器，使用 RequestContextListener 监听器可以将 HTTP request 对象绑定到该请求提供服务的 Thread。

在 Web 中，每次 HTTP 请求对应一个 request 对象。将 Bean 的作用域设定为 request，表示该 Bean 只在当前请求中有效。request 作用域的 Bean 的配置如图 17.23 所示。

```
<bean id="UserDAO" class="dao.UserDAOImpl" scope="request" />
```

图 17.23　request 作用域的 Bean 的配置

### 17.3.4　Session 作用域

在 Web 中，每次 HTTP 会话对应一个 Session 对象。将 Bean 的作用域设定为 session，表示该 Bean 只在当前会话中有效。当一次会话结束时，session 作用域的 Bean 的配置如图 17.24 所示。

```
<bean id="UserDAO" class="dao.UserDAOImpl" scope="session" />
```

图 17.24　session 作用域的 Bean 的配置

### 17.3.5　global session 作用域

global session 作用域和 Session 作用域是非常相似的，不过它在开发中非常少用。在 global session 作用域的 bean 中被限定在全局 portlet Session 的生命范围内。global session 作用域的 Bean 配置如图 17.25 所示。

```
<bean id="UserDAO" class="dao.UserDAOImpl" scope="global session" />
```

图 17.25　global session 作用域的 Bean 配置

## 17.4　小结

本章讲解了 Spring 的第一个重要部分控制反转，首先介绍了 IoC 容器的知识，然后介绍了依赖注入技术，最后说明了 Bean 作用域的各种配置。本章的重点是理解并争取掌握依赖注入的方法，难点是依赖注入关于注入其他 Bean 部分知识的理解。希望读者多加练习，力争为 Spring 技术的运用和学习打下坚实的基础。

## 17.5　本章习题

1．开发一个在线考试系统的试题业务类，在其中定义录入试题、查找试题的业务方法，并在 Spring 的配置文件中对试题业务类进行配置。定义 Bean 的 id 名称为“subjectSevice”。

【分析】本题意在考查读者创建试题业务类，以及对业务类进行配置的能力。

【核心代码】本题的核心代码如下所示。

SubjectServiceImpl.java:

```
package service;
import dao.SubjectDAO;
public class SubjectServiceImpl {
    SubjectDAO subjectDAO;
    public void setSubjectDAO(SubjectDAO subjectDAO) {
        this.subjectDAO = subjectDAO;
    }
.....
}
```

applicationContext.xml:

```
<bean id="subjectDAOImpl" class="dao.SubjectDAOImpl"></bean>
<bean id="subjectSevice" class="service.SubjectServiceImpl">
    <property name="subjectDAO">
        <ref bean="subjectDAOImpl"/>
    </property>
</bean>
```

2．开发一个客户端程序，首先在其中实例化容器和实体业务 Bean，调用录入试题和查找试题的业务方法，然后使用 Setter 方法的方式将操作试题的 DAO 接口注入到试题实现类中。

【分析】本题意在考查读者客户端程序，以及使用 Setter 方法进行依赖注入的能力。

【核心代码】本题的核心代码如下所示。

SubjcetClient.java:

```
package client;
import org.springframework.beans.factory.BeanFactory;
……
public class SubjcetClient {
    public static void main(String[] args) {
        BeanFactory factory = new ClassPathXmlApplicationContext("application-
Context.xml");
        SubjectServiceImpl subjectServiceImpl=(SubjectServiceImpl)factory.
getBean("subjectSevice");
        subjectServiceImpl.addSubjcet("no1");
        subjectServiceImpl.findSubject(1);
    }
}
```

applicationContext.xml:

```
<bean id="subjectDAOImpl" class="dao.SubjectDAOImpl"></bean>
    <bean id="subjectSevice" class="service.SubjectServiceImpl">
        <property name="subjectDAO">
            <ref bean="subjectDAOImpl"/>
        </property>
    </bean>
```

3．将操作试题 DAO 实现类的 Bean 设置为 Propotype 作用域，从而使在试题业务实现类中每次引用的对象不同。

【分析】本题意在考查读者设置 Bean 作用域、客户端程序以及使用 Setter 方法进行依赖注入的能力。

【核心代码】本题的核心代码如下所示。

applicationContext.xml:

```
<bean id="subjectSevice" class="service.SubjectServiceImpl">
    <property name="subjectDAO">
        <bean class="dao.SubjectDAOImpl"></bean>
    </property>
</bean>

<bean id="subjectDAO" class="dao.SubjectDAOImpl" scope="Prototype"></bean>
```

# 第18章　面向切面编程

本章继续学习Spring的第二大功能，即面向切面编程（Aspect Oriented Programming，AOP）。在前面学习 Java 时，我们知道 Java 是一门面向对象的语言。面向切面编程在一定程度上弥补了面向对象编程的不足。面向对象编程是对父类、子类这种纵向关系编程，而面向切面编程是在方法的前后进行横向关系编程。这一章我们就来对面向切面编程进行详细讲解。

## 18.1　面向切面编程简介

对于初学者而言，面向切面编程是一个全新的编程思想。这里提出了切面的概念，通过切面对关注点进行模块化，其中最常见的切入点就是方法。使用面向对象编程，在指定方法的前面执行另一个方法是不难实现的，但是如果要在同一个包下所有类的方法前面都执行统一方法就是一件比较复杂的工作了。在这种情况下，如果使用面向切面编程就很容易实现。

### 18.1.1　面向切面编程的概念

在面向切面编程中，经常会看到很多抽象的概念，就像我们刚开始学习 Java 时，也会遇到类、对象、多态等这些难懂的对象一样。我们先来对 AOP 中经常遇到的一些概念进行讲解。

首先是切面，切面就像是一把刀，可以将一个事物一分为二。Spring 中的切面可以将一个程序分为两部分，在中间加入自己想做的事。而且可能不仅仅是对一个程序，也许是一个包下所有的程序，如图 18.1 所示。

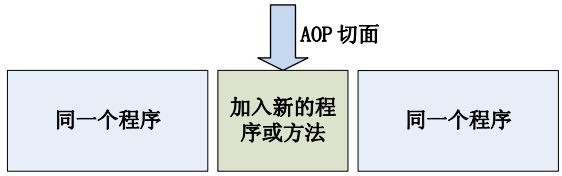



图 18.1　切面的概念

在一个程序中使用切面，通常是对程序中的方法进行操作，方法调用和处理异常等待时间段称为连接点。在面向切面编程中，一个连接点就代表一个方法的执行。在 Spring 中，执行切面编程要用到拦截器的概念，当运行到某一切入点时，会有拦截器响应，这就是通知的概念。

### 18.1.2　面向切面编程的功能

Spring AOP 是使用纯 Java 语言编写的，所以我们可以很容易的完成对它的扩展。在目前的 Spring 面向切面编程中，仅支持以方法作为连接点。之所以会这样，是因为在实际开发中以方法作为连接点是应用最多的，对于成员变量的操作是非常少的。下面我们就来重点学习 Spring 中使用方法做连接点的面向切面编程。

## 18.2　使用注解方式进行 AOP 开发

注解是 Java 5.0 版本中提出的新特性技术，通过注解可以使一个普通的 Java 程序完成特定的功能。在 Spring 中，要使用注解方法进行 AOP 开发，需要使用到 AspectJ 组件技术，通过使用 AspectJ 的库可以完成对切点的解析和匹配。

### 18.2.1　启动 AspectJ 的支持

要想在 Spring 的面向切面开发中使用注解方式，就需要使用到 AspectJ 组件技术。要使用 AspectJ 技术首先要导入 AspectJ 相关的两个 jar 包：aspectjweaver.jar 和 aspectjrt.jar，它们位于 lib/aspectj 目录下。

AspectJ 可以在其官方网站 http://www.eclipse.org/aspectj/进行下载，其具体下载过程如图 18.2 所示。

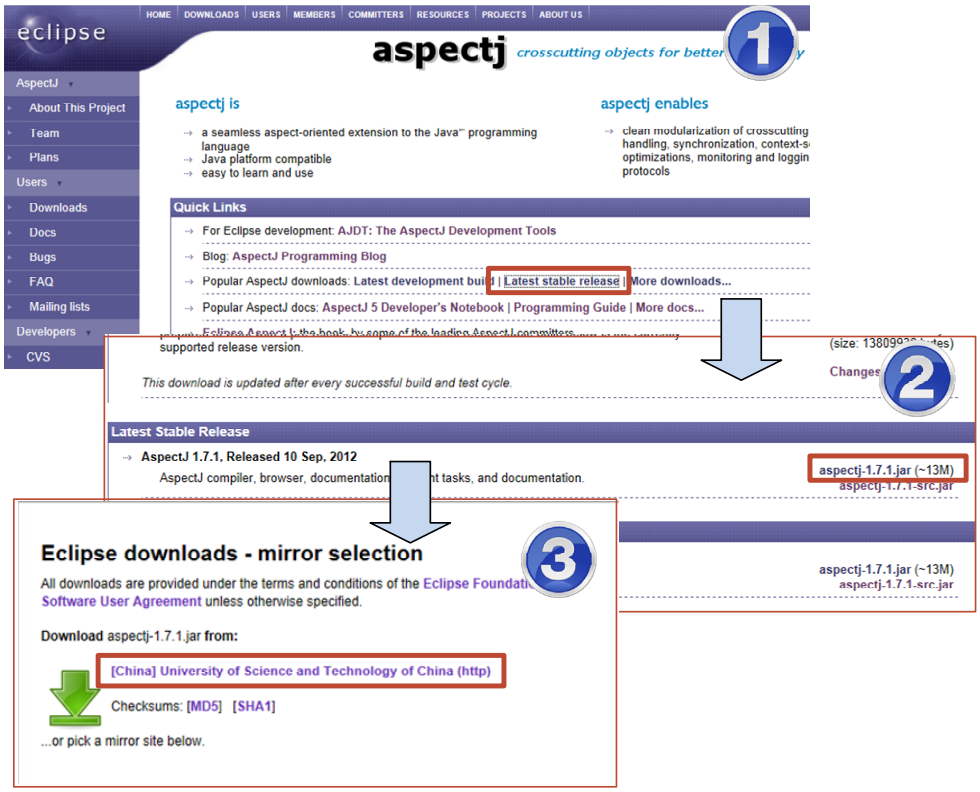



图 18.2　AspectJ 的下载过程

解压 aspectj-1.7.1 就可以得到需要的 jar 包，我们只要将其拷贝到创建的项目文件夹下就可以了。导入了 AspectJ 相关的 jar 包之外，我们还需要在 Spring 的 applicationContext.xml 配置文件中，加入如图 18.3 所示的配置。

```
<aop:aspectj-autoproxy/>
```

图 18.3　AspectJ 的配置

## 18.2.2　声明切面

在启动 AspectJ 的支持之后，我们在 Spring 中开发 Bean 程序时，需要在 Bean 类的前面加入@AspectJ，从而标明该类是 Spring 的切面，我们举一个示例如图 18.4 所示。

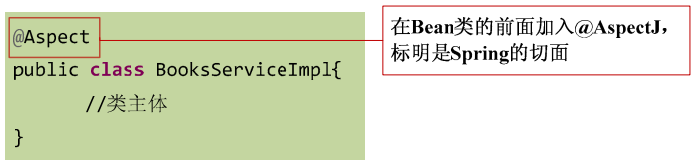



图 18.4　声明切面

同样，开发完 Bean 之后，我们也要在 Spring 的配置文件中对其进行配置，配置方法如图 18.5 所示。

```
<bean id="booksServiceImpl" class="BooksServiceImpl">
</bean>
```

图 18.5　对声明切面的 Bean 进行配置

## 18.2.3　声明切入点

切入点决定了连接点关注的内容，使得我们可以控制通知什么时候执行。我们已经知道面向切面编程仅对方法执行，所以切入点也仅仅是判断哪些方法需要进行面向切面编程。

声明切入点需要使用@Pointcut 注解，并在后面给出切入点表达式，定义关注哪些方法的执行。最后还要给出一个切入点名称，它通过一个没有返回值的普通方法构成。我们可以给出一个例子，如图 18.6 所示。

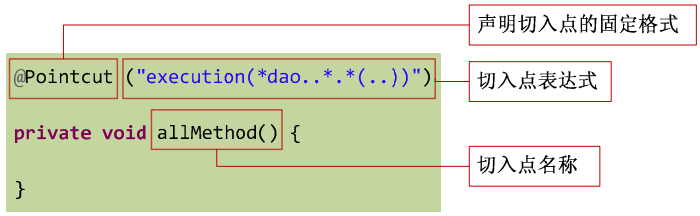



图 18.6　声明切入点

## 18.2.4　声明通知

声明通知的作用是告诉程序应当在切入点的什么时候执行下面的方法。通知有四种类型，如图 18.7 所示。

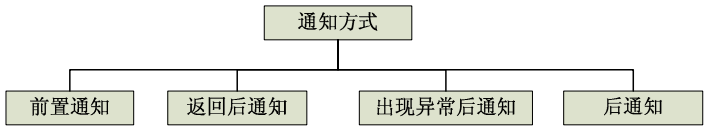



图 18.7　通知的类型

其中最常用的前置通知是指在切入点方法运行之前进行通知，从而执行下面的方法。前置通知使用的是@Before 注解。我们可以给出一个例子，如图 18.8 所示。

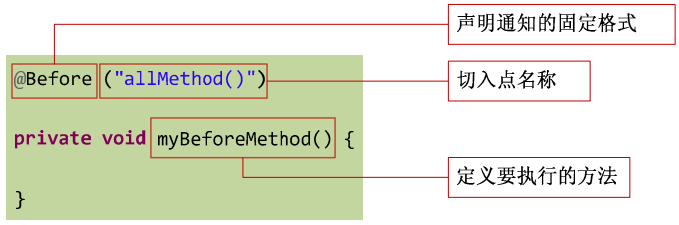



图 18.8　声明通知

关于切入点和通知，我们会在后面为大家再单独介绍。

## 18.3　使用注解对数据访问层进行管理

我们对如何使用注解方式进行 AOP 开发已经有了一定的了解，现在我们就通过一个实际的例子来看怎样使用注解对数据访问层进行管理。

在实际开发中，数据访问层是不可缺少的一部分。在数据访问层中通常要具有增、删、改、查等一系列方法。在开发过程中，有这样一个需求，不管进行任何操作时，都要通知程序员。如果我们不使用面向切面技术，就只能在每一个操作数据库方法前加入一段代码，这是非常重复的。我们现在就通过注解方式使用切面编程解决这样一个需求。

完成这样一个功能首先要定义数据访问层代码，然后定义一个使用注解的程序，在其中定义切面、切入点和通知，其中实现代码文件如表 18.1 所示。

表 18.1　进行管理功能文件描述

| 文 件 名 | 文 件 功 能 |
|---|---|
| UserDAOImpl.java | 数据访问层实现类，在其中定义了增删改查功能 |
| AspectJAnnotation.java | 面向切面编程类，在其中定义了进行数据库操作前执行的代码 |
| applicationContext.xml | Spring 的配置文件，在其中将数据访问层类和面向切面编程类进行配置 |
| UserClient.java | 用户功能测试程序，通过该程序来进行面向切面功能的测试 |

**【示例 18.1】**我们先来建立一个数据访问层实现类 UserDAOImpl.java，在其中定义了增加、删除、更新和根据 ID 查询、根据信息查询 5 种功能。为了简单起见，我们这里仅仅以一行字表示运行了该方法，具体代码内容如图 18.9 所示。

然后我们再通过“@Aspect”注解将一个类 AspectJAnnotation.java 声明一个切面，其中还包括了切入点和通知的描述，具体代码如图 18.10 所示。

```
package dao;
public interface UserDAO {
        public void addUser(String name,String password);
        public void deleteUser(int id);
        public void updateUser(int id,String name,String password);
        public void findUserById(int id);
        public void findUserByUser(String name,String password);
}
```

定义UserDAO类

```
package dao;
public class UserDAOImpl implements UserDAO {
        public void addUser(String name, String password) {
                System.out.println("执行增加用户操作");
        }
        public void deleteUser(int id) {
        System.out.println("执行删除用户操作");
        }
        public void updateUser(int id, String name, String password) {
        System.out.println("执行更新用户操作");
        }
        public void findUserById(int id) {
        System.out.println("执行根据用户ID查询用户操作");
        }
        public void findUserByUser(String name, String password) {
        System.out.println("执行根据用户信息查询用户操作");
        }
}
```

实现对数据库的各种操作

图 18.9　UserDAOImpl.java 示例

```
package service;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class AspectJAnnotation {
        @Pointcut("execution(* dao.*.*(..))")
        public void allMothod(){}
        @Before("allMothod()")
        public void myBeforeMethod(){
                System.out.println("开始进行数据库操作");
        }
}
```

声明切面

声明切入点并定义切入点名称

声明通知并定义切面方法

图 18.10　AspectJAnnotation.java 示例

接着我们还要对 Spring 的配置文件 applicationContext.xml 进行配置。不过该配置文件和前面的配置文件有所不同，需要在其中加入如图 18.11 所示的几行代码，这是 XML 文件的约束文件。

applicationContext.xml 文件的具体配置如图 18.12 所示。

```
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
```

图 18.11　XML 文件的约束文件

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
          xmlns="http://www.springframework.org/schema/beans"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xmlns:aop="http://www.springframework.org/schema/aop"
          xsi:schemaLocation="http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
           http://www.springframework.org/schema/aop
           http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
    <aop:aspectj-autoproxy/>
         <bean id="userDAOImpl" class="dao.UserDAOImpl">
         </bean>
         <bean id="aspectJAnnotation" class="service.AspectJAnnotation">
         </bean>
</beans>
```

对数据访问层进行配置

图 18.12　applicationContext.xml 配置文件

最后我们通过一个客户端程序 UserClient.java，来测试是否能够使用注解对数据访问层进行管理，具体代码如图 18.13 所示。

运行这个 Java 程序，在控制台输出运行结果如图 18.14 所示。

```
package client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import dao.UserDAO;
public class UserClient {
      public static void main(String[] args) {
            BeanFactory factory = new
                ClassPathXmlApplicationContext("applicationContext.xml");
            UserDAO userDAO=(UserDAO)factory.getBean("userDAOImpl");
            userDAO.addUser("Lester", "123456");
            userDAO.findUserById(5);
      }
}
```

获取数据访问接口，根据ID查询用户操作

图 18.13　UserClient.java 示例

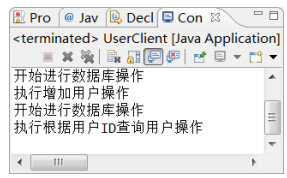



图 18.14　使用注解对数据访问层进行管理

从运行结果中可以看到，每执行一种数据库操作都会在执行前显示“开始执行数据库操作”，从而告诉程序员相关操作数据库信息。

## 18.4　切入点

在前面讲解注解方法进行 AOP 开发中，我们已经提到了切入点的使用。本节再来深入对

其进行讲解，包括切入点指定者、合并切入点和切入点表达式等知识点。

### 18.4.1 切入点指定者

在进行 Spring 面向切面编程时，支持在切入点表达式中使用多种 AspectJ 切入点指定者。其中，最常用的就是前面使用到的 execution，通过它来匹配方法执行的连接点。其他较为常用的切入点指定者如表 18.2 所示。

表 18.2 常用切入点指定者

| 切入点指定者 | 连接点说明 |
| --- | --- |
| within | 通过限定匹配特定类型确定连接点 |
| this | 通过指定类型的实例确定连接点 |
| target | 通过目标对象确定连接点 |
| args | 通过参数确定连接点 |

### 18.4.2 合并连接点

在 Java 中有“&&”、“||”和“!”3 种逻辑运算符。在切入点表达式中也可以使用这 3 种逻辑运算符，其中最常用的就是“||”，它表示两边的方法都可以声明切入点。我们可以举一个例子如图 18.15 所示。

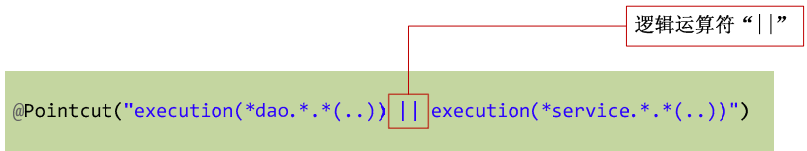



图 18.15 合并连接点

上面切入点表达式表示不管是 dao 包下的接口还是 service 包下的接口内的所有方法都会定义成切面的切入点，从而使它们都能够进行面向切面开发。

### 18.4.3 切入点表达式

读者可能对切入点表达式有很大的疑惑。本节就来通过 execution 切入点指定者来讲解切入点表达式，它的语法格式如图 18.16 所示。

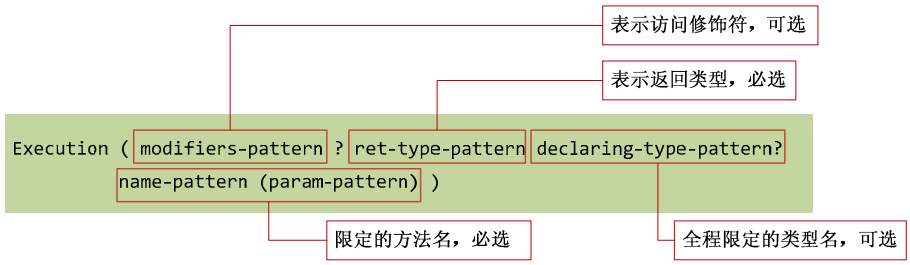



图 18.16 切入点表达式的语法格式

我们再来列举几个简单的例子来帮助大家更好地理解切入点表达式，如图 18.17 所示。

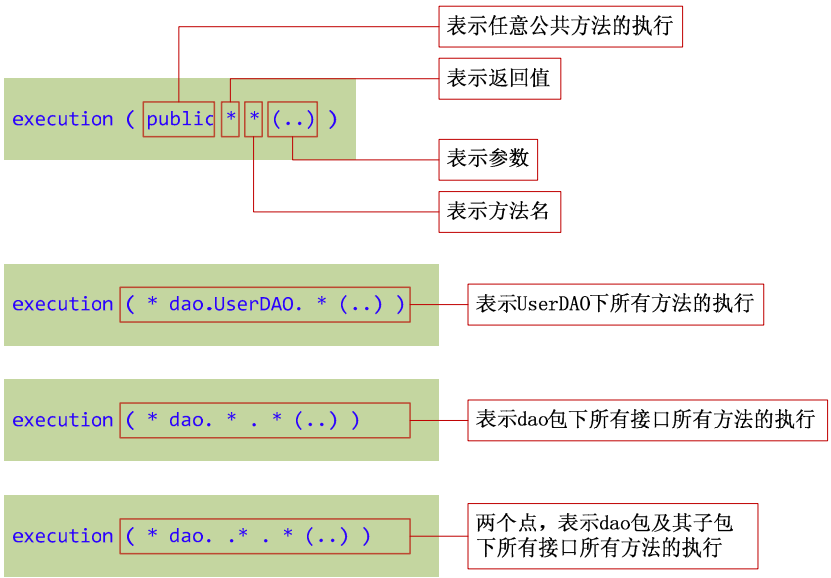



图 18.17　切入点表达式应用示例

## 18.5　通知

在前面学习开发步骤时，我们也对通知有了一个简单的了解。通知是跟一个切入点表达式关联起来的，并且在切入点匹配的方法的某一运行时刻进行运行。除了前置通知外，我们对其他几种通知类型也进行进一步讲解。

### 18.5.1　返回后通知

返回后通知是指当匹配的方法执行 return 语句返回值时进行通知，它使用@AfterReturnning 注解进行声明，其基本形式如图 18.18 所示。

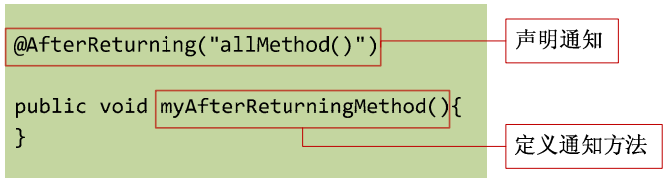



图 18.18　返回后通知的基本形式

**【示例 18.2】**我们通过一个例子 AfterReturningTest.java 学习返回后通知的使用，通过该程序获取匹配方法的返回值，并将返回值显示出来，具体代码如图 18.19 所示。

**注意：**returning 子句也是一种对匹配方法的限制，匹配方法的返回值类型必须要能转换为通知方法中的参数类型。

```
package advice;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class AfterReturningTest {
    @Pointcut("execution(* dao.*.*(..))")
    public void allMothod(){}
    @AfterReturning(
                pointcut="allMothod()",
                returning="retVal")
    public void myAfterReturningMethod(Object retVal){
        System.out.println("方法的返回值为"+retVal);
    }
}
```

声明切面

声明切入点并定义切入点名称

声明通知并定义切面方法

图 18.19　AfterReturningTest.java 示例

## 18.5.2　出错后通知

出错后通知是指当执行匹配方法时出现异常后进行通知。它使用@AfterThrowing 注解进行声明，其基本形式如图 18.20 所示。

```
@AfterThrowing("allMethod()")
public void myAfterThrowingMethod(){
}
```

声明通知

定义通知方法

图 18.20　出错后通知的基本形式

**【示例 18.3】**我们通过一个例子 AfterThrowingTest.java 学习出错后通知的使用。通过该程序获取匹配方法的出现异常类型，并将异常信息显示出来，具体代码如图 18.21 所示。

```
package advice;
import java.io.IOException;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;
@Aspect
public class AfterThrowingTest {
    @Pointcut("execution(* dao.*.*(..))")
    public void allMothod(){}
    @AfterThrowing(
                pointcut="allMothod()",
                throwing="ioex")
    public void myAfterThrowingMethod(IOException ioex){
        System.out.println("方法的抛出异常信息为：" + ioex.getMessage());
    }
}
```

声明切面

声明切入点并定义切入点名称

声明通知并定义切面方法

图 18.21　AfterThrowingTest.java 示例

### 18.5.3　后通知

后通知是指在匹配的方法结束后通知，它使用@After 注解进行声明。需要注意的是，方法的结束有很多情况，例如，正常运行结束、返回之后结束和发生异常，不管是哪种方式的结束都会进行后通知，后通知的基本形式如图 18.22 所示。

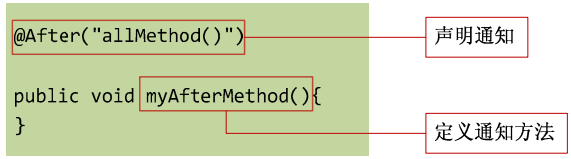



图 18.22　后通知的基本形式

### 18.5.4　环绕通知

环绕通知是前置通知和后通知的结合体，使用环绕通知可以使一个方法的前后都执行通知方法。并且通过环绕通知可以决定什么方法什么时候执行、如何执行和是否执行。

环绕通知是使用@Around 注解声明的，在其中指定切入点名称，环绕通知的基本形式如图 18.23 所示。

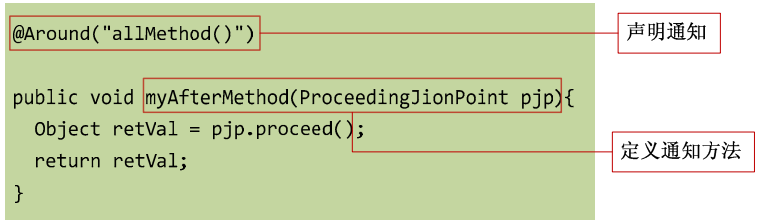



图 18.23　环绕通知的基本形式

**注意**：环绕通知并没有想象中那么好，它会使程序变得非常复杂。当只需要完成某一方面操作时，例如只是在方法运行前通知，最好还是使用前置通知。

## 18.6　在 Spring 中进行 JDBC 编程

在 Spring 中可以整合多种数据库操作框架，例如整合 Hibernate。本节我们先来讲解在 Spring 中如何进行最原始的 JDBC 编程。在 JDBC 编程中有很多重复的代码，如连接和关闭数据库连接等，在 Spring 中可以对这些重复代码进行封装，从而使程序员关注于业务的开发。

### 18.6.1　Spring 中的数据库封装操作类和数据源接口

Spring 中的数据库封装操作类是 JdbcTemplate，它位于 org.springframework.jdbc.core 包下。使用该类可以完成数据库的连接和关闭，从而简化了 JDBC 操作。

数据源接口也就是 DataSource 接口，通过使用该接口可以创建一个数据库连接。在 Spring 中有多种创建数据源的方式，本书主要通过 Spring 配置的方式建立数据源。我们知道，要想连

接数据库最少要知道数据库的驱动、url、用户名和密码，所以在配置 DataSource 接口时需要给出这些信息。首先我们来加载 MySQL 数据库驱动，添加方法如图 18.24 所示。

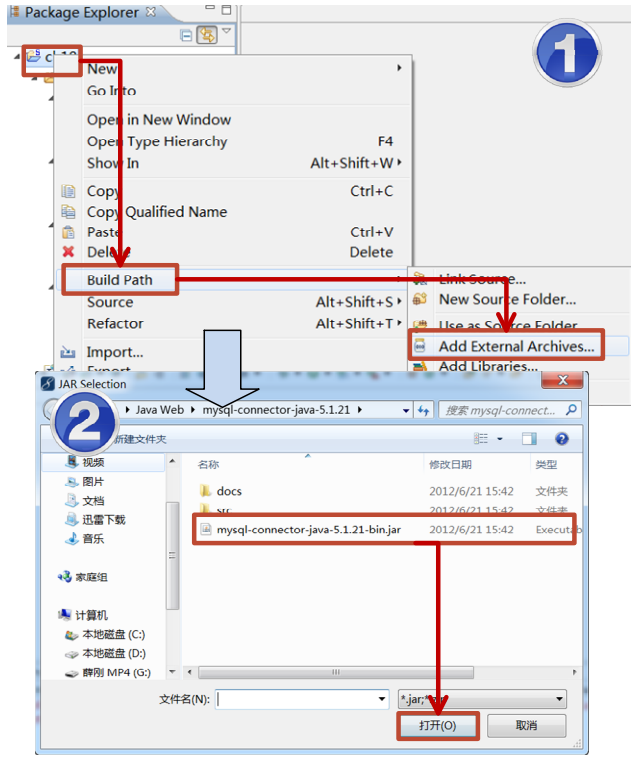



图 18.24　加载 MySQL 数据库驱动

applicationContext.xml 文件中的数据库配置代码如图 18.25 所示。

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName">
            <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
            <value>jdbc:mysql://localhost:3306/spring</value>
        </property>
        <property name="username">
            <value>root</value>
        </property>
        <property name="password">
            <value>123456</value>
        </property>
</bean>
```

图 18.25　applicationContext.xml 配置文件

## 18.6.2　创建数据库表操作

学习完 Spring 中的数据库封装操作类和数据源接口，我们就可以使用其进行数据库开发

了。我们先来看一个进行创建数据库开发的例子。

【示例 18.4】通过程序 ExecuteManager.java 我们执行一条创建数据库表的 SQL 语句，从而完成创建数据库表的功能，具体代码如图 18.26 所示。

```
package service;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
public class ExecuteManager {
    private JdbcTemplate jt;
    private DataSource dataSource;
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public void doExecute() {
        jt = new JdbcTemplate(dataSource);
        jt.execute("create table student (id integer, name varchar(100))");
    }
}
```

封装类引用数据源接口引用

执行SQL语句的业务方法

图 18.26　ExecuteManager.java 示例

完成执行 SQL 语句的业务类，还需要在 Spring 的配置文件中进行如图 18.27 所示的配置。

```
<bean id="execute" class="service.ExecuteManager">
        <property name="dataSource">
                <ref bean="dataSource"/>
        </property>
</bean>
```

图 18.27　applicationContext.xml 配置文件

配置完成后，我们就可以在客户端编写程序代码对数据库进行操作了。我们创建程序 ExecuteClient.java，通过实现 MethodBeforeAdvice 前置通知接口的方式监听数据库的操作次数，具体代码如图 18.28 所示。

```
package client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.ExecuteManager;
public class ExecuteClient {
    public static void main(String[] args) {
        BeanFactory factory = new
              ClassPathXmlApplicationContext("applicationContext.xml");
        ExecuteManager execute=(ExecuteManager)factory.getBean("execute");
        execute.doExecute();
      }
}
```

获取执行实例执行SQL语句

图 15.28　ExecuteClient.java 示例

运行这个 Java 程序，打开 MySQL 数据库查看运行结果如图 18.29 所示。

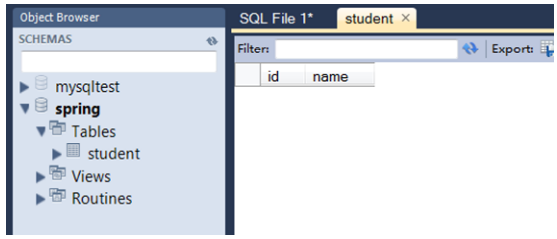



图 18.29　创建数据库表运行结果

### 18.6.3　更新数据库操作

更新数据库操作包括插入、修改和删除 3 种，这 3 种操作都是通过 update 方法完成的。在 update 方法中给出 SQL 语句参数，就可以完成相应的数据库操作。

**【示例 18.5】**通过程序 UpdateManager.java 我们执行一条向数据库表插入数据的 SQL 语句，从而完成插入信息的功能，具体代码如图 18.30 所示。

```
package service;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
public class UpdateManager {
    private JdbcTemplate jt;
    private DataSource dataSource;
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public void doUpdate() {
        jt = new JdbcTemplate(dataSource);
        jt.update("insert into user values (1,'李默')");
    }
}
```

封装类引用
数据源接口引用

执行SQL语句的
业务方法

图 18.30　UpdateManager.java 示例

完成执行 SQL 语句的业务类，还需要在 Spring 的配置文件中进行如图 18.31 所示的配置。

```
<bean id="update" class="service.UpdateManager">
        <property name="dataSource">
                <ref bean="dataSource"/>
        </property>
</bean>
```

图 18.31　applicationContext.xml 配置文件

配置完成后，我们就可以在客户端编写程序代码对数据库进行插入操作了。我们编写一个 UpdateClient 类，具体代码如图 18.32 所示。

运行这个 Java 程序，打开 MySQL 数据库查看运行结果如图 18.33 所示。

```
package client;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.ExecuteManager;
import service.UpdateManager;
public class UpdateClient {
    public static void main(String[] args) {
        BeanFactory factory = new
            ClassPathXmlApplicationContext("applicationContext.xml");
        UpdateManager update=(UpdateManager)factory.getBean("update");
        update.doUpdate();
    }
}
```

获取执行实例
执行SQL语句

图 18.32　UpdateClient.java 示例


图 18.33　更新数据库操作运行结果

## 18.6.4　查询数据库操作

创建数据库表，并且向其中插入数据后，就可以通过查询操作将数据库表中的数据查询出来。在 Spring 的封装数据库中查询数据的方法有很多，这里我们使用其中的两个，分别是查询一个数据和查询所有数据。

**【示例 18.6】**通过程序 SelectManager.java 我们执行两种查询数据库表的 SQL 语句，从而获取指定用户的名称和所有用户的名称，具体代码如图 18.34 所示。

```
package service;
import java.util.List;
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;
public class SelectManager {
    private JdbcTemplate jt;
    private DataSource dataSource;
    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
    public String doGetName() {
        jt = new JdbcTemplate(dataSource);
        String name=(String)jt.queryForObject
                ("select name from student where id=1", String.class);
        return name;
    }
    public List doGetNames(){
       jt = new JdbcTemplate(dataSource);
       List list=jt.queryForList("select name from student");
       return list;
    }
}
```

封装类引用
数据源接口引用

查询指定用户的名称

获取所有用户的名称

图 18.34　SelectManager.java 示例

完成执行 SQL 语句的业务类，还需要在 Spring 的配置文件中进行如图 18.35 所示的配置。

```
<bean id="select" class="service.SelectManager">
        <property name="dataSource">
            <ref bean="dataSource"/>
        </property>
</bean>
```

图 18.35　applicationContext.xml 配置文件

配置完成后，我们就可以在客户端编写程序代码对数据库进行查询操作了。我们编写一个 SelectClient 类，具体代码如图 18.36 所示。

```
package client;
import java.util.List;
import java.util.Map;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import service.SelectManager;
public class SelectClient {
    public static void main(String[] args) {
        BeanFactory factory = new
            ClassPathXmlApplicationContext("applicationContext.xml");
        SelectManager select=(SelectManager)factory.getBean("select");
        System.out.println("特定查找姓名："+select.doGetName());
        List list=select.doGetNames();
        for(int i=0;i<list.size();i++)
        {
            Map name=(Map)list.get(i);
            System.out.println("姓名："+name);
        }
    }
}
```

获取数据访问接口

图 18.36　SelectClient.java 示例

运行这个 Java 程序，在控制台输出运行结果如图 18.37 所示。

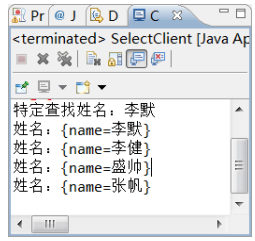



图 18.37　查询数据库操作运行结果

## 18.7　小结

本章讲解了 Spring 的第二个重要部分面向切面编程。首先我们为大家介绍了面向切面编程的相关知识，然后介绍了使用注解方式进行 AOP 开发的相关步骤，接着又为大家举例说明了

如何使用注解对数据访问层进行管理，最后为大家讲解了在 Spring 中进行 JDBC 编程的操作。本章的重点是理解并争取掌握使用注解方式进行 AOP 开发和在 Spring 中进行 JDBC 编程的方法，难点是对切入点和通知等知识点的理解。希望读者多加练习，力争为 Spring 技术的运用和学习打下坚实的基础。

## 18.8　本章习题

1．使用注解的方式将用户操作业务实现类 UserServiceImpl 声明成切面，开发用户操作 DAO，在业务实现类中声明切入点，当执行用户操作后，执行通知方法。

【分析】本题考查读者对于切面和切入点知识的理解。

【核心代码】本题的核心代码如下所示。

```
package service;
import org.aspectj.lang.annotation.AfterReturning;
……
@Aspect
public class UserServiceImpl {
    //@Pointcut("execution(* dao..*.*(..) ")
    @Pointcut("execution(* dao.UserDAOImpl.delectUser(..)) || execution(* dao.
UserDAOImpl.addUser(..))")
    private void method(){}

    /*@AfterReturning(pointcut="allMethod ()",returning="retVal")
    private void myBeforeMethod(Object retVal) {
        if(retVal instanceof java.lang.String){

        }else if(retVal instanceof java.lang.Integer){

        }
    }*/
    @AfterThrowing(pointcut="allMethod ()",throwing="e")
    private void myBeforeMethod(Exception e) {
        e.printStackTrace();
    }
}
```

2．使用 Spring 进行 JDBC 数据库编程，创建一个保存用户信息的数据表，其中包括用户 ID、用户名和密码等信息。

【分析】本题考查读者在 Spring 中进行 JDBC 编程的能力。首先我们要掌握创建数据库表的操作。

【核心代码】本题的核心代码如下所示。

```
package service;
……
public class ExecuteManager {
    private JdbcTemplate jt;            //封装类引用
   private DataSource dataSource;       //数据源接口引用
   public void setDataSource(DataSource dataSource) {    //定义注入的 Setter 方法
      this.dataSource = dataSource;
   }
   public void doExecute() {            //执行 SQL 语句的业务方法
      jt = new JdbcTemplate(dataSource);
      jt.execute("create table user (id integer, name varchar(100),password
varchar(100))");
   }
}
```

3．接收注册页面提交的用户名和密码信息，并将这些信息通过 Spring 进行 JDBC 编程的方式保存到用户表中。

【分析】本题考查读者在 Spring 中进行 JDBC 编程的能力。这里我们要掌握更新数据库表的操作。

【核心代码】本题的核心代码如下所示。

```
package service;
……
public class UpdateManager {
     private JdbcTemplate jt;                    //封装类引用
   private DataSource dataSource;                //数据源接口引用
   public void setDataSource(DataSource dataSource) {    //定义注入的 Setter 方法
      this.dataSource = dataSource;
   }
   public void doUpdate() {                      //定义更新业务方法
      jt = new JdbcTemplate(dataSource);         //创建封装类对象
      jt.update("insert into user values (1,'Tom','123456')");
   }
}
```

PART 5

# S2SH 整合篇

第 19 章　框架技术整合开发

# 第 19 章　框架技术整合开发

在前面的章节中，我们分别学习了 Struts 2、Hibernate 以及 Spring 三个框架。但是每个框架都是单独介绍的，并没有涉及整合的内容。在本章中我们将重点介绍 3 个框架的整合开发，包括 Struts 2 和 Hibernate 框架的整合开发、Struts 2 和 Spring 框架的整合开发以及 Hibernate 和 Spring 框架的整合开发。通过本章的学习，读者应能够领会各框架的整合思想，从而迈进企业级开发的殿堂。

## 19.1　Struts 2 和 Hibernate 框架的整合开发

一个 Web 应用最重要的两部分，就是与用户交互的表现层和与数据库交互的数据访问层（DAO 层）。Struts 2 和 Hibernate 框架整合正好可以完美地实现这两部分的搭配，如图 19.1 所示。本节我们就来详细地介绍如何整合 Struts 2 和 Hibernate 框架。

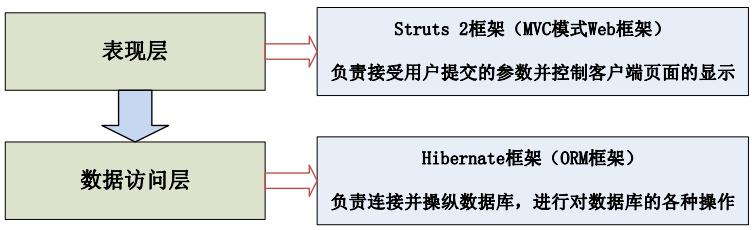



图 19.1　Struts 2 和 Hibernate 框架的整合

### 19.1.1　整合策略

在整合 Struts 2 和 Hibernate 开发之前，首先我们必须清楚开发分层的思想。一个好的软件项目都是采用多层设计的，这就好比一个软件公司分为多个部门一样，每个层次负责不同的功能。在软件开发中使用分层的思想，就可以确定每层的工作任务，从而提高代码的内聚。

在 Web 开发中，一般采用 5 层架构，分别为表现层、业务逻辑层、数据访问层、持久层以及数据库层，它们各层的作用如图 19.2 所示。

**注意：**在实际开发中，我们有时候并不将数据库层放入 Web 开发中，而是将持久层作为底层。

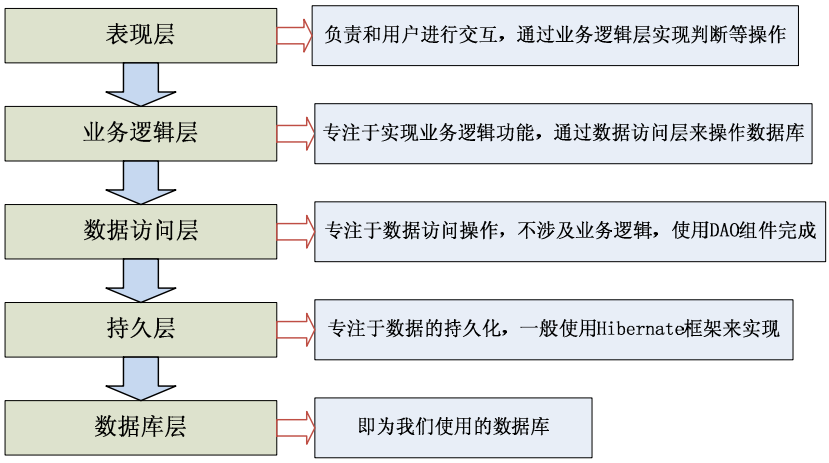



图 19.2　5 层架构模型图

下面我们就通过一个网上书店的应用来演示 Struts 2 和 Hibernate 框架的整合策略。

## 19.1.2　数据库层开发

数据库层用来接收数据访问层提交的数据，也供数据访问层获取数据。数据库层一般使用数据库管理系统，如 MySQL 数据库。在一个网上书店中要有一个用来存放书籍信息的数据库表，其中书籍信息包括书籍编号、书籍名称、书籍 ISBN 号以及书籍价格。

**【示例 19.1】**我们在 MySQL 数据库中创建一个 s2sh 数据库，并设置为默认，然后通过如图 19.3 所示的 SQL 语句，创建一个包含 4 个字段的数据库表。

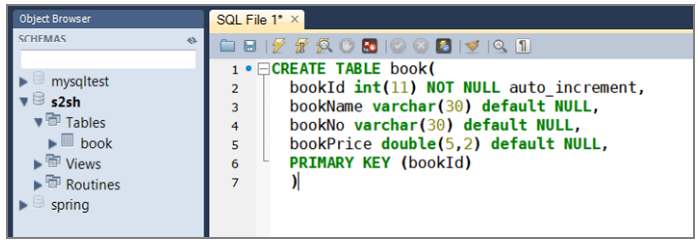



图 19.3　创建数据库表的 SQL 语句

## 19.1.3　持久层开发

持久层开发主要包含两个部分，一个是 Hibernate 配置文件 hibernate.cfg.xml 的开发，一个是实体类和映射文件的开发。关于 hibernate.cfg.xml 的开发我们不再赘述，请读者参考前面的相关章节。这里我们主要来开发实体类以及映射文件。

创建了 book 表，就应该添加一个实体类 Book 来与之相对应。同时还需要添加实体类的映射文件 Book.hbm.xml，从而完成 Book 类的对象关系映射。

**【示例 19.2】**我们现在创建一个 Book 实体类，在该类中包含书籍编号、书籍名称、书籍 ISBN 以及书籍价格 4 个属性，具体代码如图 19.4 所示。

添加完实体类后，需要添加一个映射文件 Book.hbm.xml，从而完成 Book 类的对象关系映射，Book.hbm.xml 文件的具体代码如图 19.5 所示。

```
package po;
public class Book {
        private int bookId;
        private String bookName;
        private String bookNo;
        private double bookPrice;
        public int getBookId() {
                return bookId;
        }
        public void setBookId(int bookId) {
                this.bookId = bookId;
        }
        //省略其他属性的get和set方法
}
```

定义书籍的4个属性

图 19.4　实体类 Book.java 示例

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping  PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
        <class name="po.Book" table="book">
                <id name="bookId">
                        <generator class="identity"></generator>
                </id>
                <property name="bookName"></property>
                <property name="bookNo"></property>
                <property name="bookPrice"></property>
        </class>
</hibernate-mapping>
```

配置标识及生成策略

配置属性

图 19.5　配置文件 Book.hbm.xml

**注意：**配置完 Book.hbm.xml 后一定不要忘记在 hibernate.cfg.xml 添加进行相应的配置信息。即增加<mapping resource="po/Book.hbm.xml" />语句。

### 19.1.4　数据访问层开发

数据访问层又称为 DAO 层，在该层中包含了所有的操作数据的方法，如保存数据、删除数据、修改数据和查询数据等。数据访问层包括 3 个组成部分，如图 19.6 所示。


图 19.6　数据访问层开发

首先我们来定义一个 DAO 接口 BookDAO，在该接口中定义 3 个方法，分别用来添加书籍、根据 ISBN 号查询书籍以及查询所有书籍。

【示例 19.3】以下代码为一个 DAO 接口，在该接口中我们定义了 3 个操作数据库的方法，具体代码如图 19.7 所示。

```
package dao;
import java.util.List;
import po.Book;
public interface BookDAO {
        public void saveBook(Book book);
        public Book findByBookNo(String bookNo);
        public List<Book> findAllBook();
}
```

操作数据库的3种方法

图 19.7　BookDAO.java 示例

然后我们再来添加 DAO 接口的实现类 BookDAOImpl，在该类中实现 BookDAO 中声明的 3 个方法，并通过调用 Hibernate 中的相应方法来完成数据访问操作，具体代码如图 19.8 所示。

```
package dao;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import hibernate.HibernateSessionFactory;
import po.Book;
public class BookDAOImpl implements BookDAO{
        public List<Book> findAllBook() {
                Session session = HibernateSessionFactory.getSession();
                String hql = "from Book";
                Query query = session.createQuery(hql);
                List<Book> books = query.list();
                HibernateSessionFactory.closeSession();
                return books;
        }
        public Book findByBookNo(String bookNo) {
                Session session = HibernateSessionFactory.getSession();
                String hql = "from Book as book " +
                                        "where book.bookNo = :bookNo";
                Query query = session.createQuery(hql);
                query.setString("bookNo", bookNo);
                List<Book> books = query.list();
                HibernateSessionFactory.closeSession();
                if(books.size() > 0) {
                        return books.get(0);
                }else {
                        return null;
                }
        }
        public void saveBook(Book book) {
                Session session = HibernateSessionFactory.getSession();
                Transaction transaction = session.beginTransaction();
                session.save(book);
                transaction.commit();
                HibernateSessionFactory.closeSession();
        }
}
```

获得一个Session对象，并通过HQL创建一个Query对象查询所有的记录

根据ISBN号查询书籍

数据库相关事务管理

图 19.8　BookDAOImpl.java 示例

下面我们再来看 DAO 的工厂类，通过调用该工厂类的一个静态方法，能返回一个 DAO 接口类型的 DAO 实现类实例对象，这个工厂类 BookDAOFactory.java 的具体代码如图 19.9 所示。

```
package dao;
public class BookDAOFactory {
        public static BookDAO getBookDAOInstance(){
                return new BookDAOImpl();
        }
}
```

返回DAO实现类实例对象

图 19.9　BookDAOFactory.java 示例

**注意**：数据访问层是不包含任何业务逻辑的，所以在包含书籍时并不会查询该书籍是否存在。

### 19.1.5　业务逻辑层开发

业务逻辑层的开发和数据访问层基本类似，不同的是数据访问层是通过 Hibernate 来完成数据操作，而业务逻辑层重点实现的是业务逻辑。业务逻辑层中的数据操作都是通过调用数据访问层来实现的。业务逻辑层包含 3 个部分，如图 19.10 所示。


图 19.10　业务逻辑层开发

**【示例 19.4】**下面我们首先来定义一个业务逻辑组件接口，并在该接口中声明两个业务逻辑方法，这个接口类 BookService 的具体代码如图 19.11 所示。

```
package service;
import java.util.List;
import po.Book;
public interface BookService {
        public boolean inputBook(Book book);
        public List<Book> showAllBook();
}
```

声明录入书籍和查看书籍两业务逻辑方法

图 19.11　BookService.java 示例

接着我们再来添加业务逻辑组件实现类，在该实现类中实现了业务逻辑组件接口中定义的所有方法，并通过书籍访问代码来完成数据操作，这个实现类 BookServiceImpl 的具体代码如图 19.12 所示。

```
package service;
import java.util.List;
import dao.BookDAO;
import dao.BookDAOFactory;
import po.Book;
public class BookServiceImpl implements BookService{
       private BookDAO bookDAO =
                              BookDAOFactory.getBookDAOInstance();
       public boolean inputBook(Book book) {
              Book oldBook = bookDAO.
                                    findByBookNo(book.getBookNo());
              if(oldBook != null) {
                     return false;
              }else {
                     bookDAO.saveBook(book);
                     return true;
              }
       }
       public List<Book> showAllBook() {
              return bookDAO.findAllBook();
       }
}
```

按指定的ISBN查询书籍

返回所有书籍

图 19.12　BookServiceImpl.java 示例

然后我们再来添加业务逻辑组件工厂类，在工厂类中定义了一个静态方法，通过该方法能返回一个业务逻辑组件实现类的实例对象，这个工厂类 BookServiceFactory 的具体代码如图 19.13 所示。

```
package service;
public class BookServiceFactory {
       public static BookService getBookServiceInstance(){
              return new BookServiceImpl();
       }
}
```

返回业务逻辑实现类的实例对象

图 19.13　BookServiceFactory.java 示例

**注意**：业务逻辑层是这个 Web 架构中最重要的一部分，它是连接表现层和数据访问层的桥梁。

## 19.1.6　完成书籍的录入

在 5 层架构模型的最上层是表现层，表现层一般使用 MVC 框架来充当，比如 Struts 2 框架。要完成书籍的录入，首先要添加一个书籍录入表单，用来接收书籍信息，然后需要添加一个书籍录入控制器，并调用业务逻辑层来完成书籍的录入。

【示例 19.5】首先我们来添加一个书籍录入表单 bookInput.jsp，在该表单中包含 3 个文本框，分别用来输入书籍名称、书籍 ISBN 号以及书籍价格，具体代码如图 19.14 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
    <head><title>录入图书</title></head>
<body>
    <center>
    <form action="bookInput.action" method="post">
    <table>
        <tr><td colspan="2">录入图书</td></tr>
        <tr><td colspan="2"><s:actionmessage/></td></tr>
        <tr>
            <td>书籍名称:</td>
            <td><input type="text" name="bookName"/></td>
        </tr>
        <tr>
            <td>书籍ISBN号:</td>
            <td><input type="text" name="bookNo"/></td>
        </tr>
        <tr>
            <td>书籍价格:</td>
            <td><input type="text" name="bookPrice"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="录入" />
                <input type="reset" value="重置" />
                <a href="showAllBook.action">查询所有图书</a>
            </td>
        </tr>
    </table>
    </form>
    </center>
</body>
</html>
```

指定响应action页面

3个输入文本框

图 19.14　bookInput.jsp 示例

接着我们添加一个书籍录入控制器，该控制器用来接收表单提交的所有书籍信息，并完成书籍的录入，这个响应类 BookInputAction 的具体代码如图 19.15 所示。

创建完 BookInputAction 类，我们还需要在 struts.xml 文件中配置该 Action，配置代码如图 19.16 所示。

打开浏览器，在地址栏中输入 http://localhost:8080/ch19/bookInput.jsp。打开图书录入表单，填入书籍信息并提交，运行结果如图 19.17 所示。

```
package action;
import com.opensymphony.xwork2.ActionSupport;
import po.Book;
import service.BookService;
import service.BookServiceFactory;
public class BookInputAction extends ActionSupport {
       private String bookName;
       private String bookNo;
       private double bookPrice;
       //省略个属性的get和set方法
       public String execute() throws Exception {
              Book book = new Book();
              book.setBookName(bookName);
              book.setBookNo(bookNo);
              book.setBookPrice(bookPrice);
              BookService bookService =
                     BookServiceFactory.getBookServiceInstance();
              if(bookService.inputBook(book)) {
                     this.addActionMessage("录入图书成功!");
                     return SUCCESS;
              }else {
                     this.addActionMessage("录入图书失败!");
                     return INPUT;
              }
       }
}
```

创建一个Book类示例对象

判断书籍录入是否成功

图 19.15　BookInputAction.java 代码

```
<constant name="struts.i18n.encoding" value="gb2312"></constant>
<package name="struts2" extends="struts-default">
        <action name="bookInput" class="action.BookInputAction">
                <result name="success">/bookInput.jsp</result>
                <result name="input">/bookInput.jsp</result>
        </action>
</package>
```

图 19.16　struts.xml 配置文件


图 19.17　完成书籍的录入

我们也可以打开 MySQL 数据库查看信息是否添加成功，如图 19.18 所示。

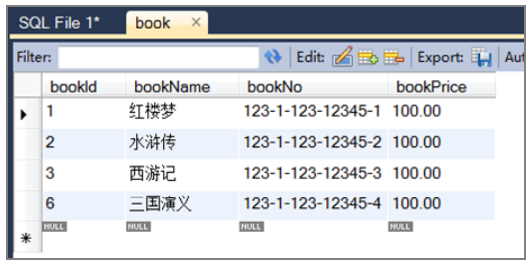



图 19.18　MySQL 数据库表信息查看结果

### 19.1.7　完成所有图书的显示

要完成所有图书信息的显示，首先需要添加一个业务控制器来获得所有的图书，然后再添加一个图书列表显示页面，在该页面中循环显示所有的图书。

**【示例 19.6】**首先我们来添加一个业务控制器 ShowAllBookAction，在该控制器中调用业务逻辑层代码，从而获得所有的图书信息，这个控制器的具体代码如图 19.19 所示。

```
package action;
import java.util.List;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
import po.Book;
import service.BookService;
import service.BookServiceFactory;
public class ShowAllBookAction extends ActionSupport {
    public String execute() throws Exception {
        BookService bookService =
                BookServiceFactory.getBookServiceInstance();
        List<Book> bookList = bookService.showAllBook();
        HttpServletRequest request = ServletActionContext.getRequest();
        request.setAttribute("bookList", bookList);
        return SUCCESS;
    }
}
```

获得组件实例
查询所有图书

图 19.19　ShowAllBookAction.java 示例

接着我们对这个 Action 类在 struts.xml 文件中进行配置，其配置代码如图 19.20 所示。

```
<action name="showAllBook" class="action.ShowAllBookAction">
        <result name="success">/showAllBook.jsp</result>
</action>
```

图 19.20　struts.xml 配置文件

然后我们再来添加一个书籍列表显示页面 showAllBook.jsp，在该页面中循环显示书籍列表中的所有图书信息，其具体代码如图 19.21 所示。

```
<%@ page language="java" import="java.util.*" pageEncoding="gb2312"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
      <head><title>显示所有图书</title></head>
<body>
<center>
<h1>查看所有书籍列表</h1>
<table border="1">
      <tr>
            <td>书籍编号</td><td>书籍名称</td>
            <td>书籍ISBN号</td><td>书籍价格</td>
      </tr>
      <s:iterator value="#request.bookList" var="book">
       <tr>
            <td><s:property value="#book.bookId"/></td>
            <td><s:property value="#book.bookName"/></td>
            <td><s:property value="#book.bookNo"/></td>
            <td><s:property value="#book.bookPrice"/></td>
      </tr>
      </s:iterator>
</table>
</center>
</body>
</html>
```

遍历循环

输出书籍信息

图 19.21　showAllBook.jsp 显示页面

打开浏览器，在地址栏中输入 http://localhost:8080/ch19/showAllBook.action。打开图书显示页面，运行结果如图 19.22 所示。


图 19.22　完成所有图书的显示

## 19.2　Struts 2 和 Spring 整合开发

Spring 本身提供了一个 MVC 框架，但是因为这套框架大量应用了映射策略，使得开发起来非常烦琐。Struts 2 框架是一个非常优秀的 MVC 框架，这时可以通过 Struts 2 和 Spring 的整合，充分利用 Spring 的 IoC 特性，从而大大降低系统各层之间的耦合度。

### 19.2.1 整合策略

在使用 Spring 框架之前，各层之间通过使用工厂类来创建组件实例，工厂模式顺序如图 19.23 所示。

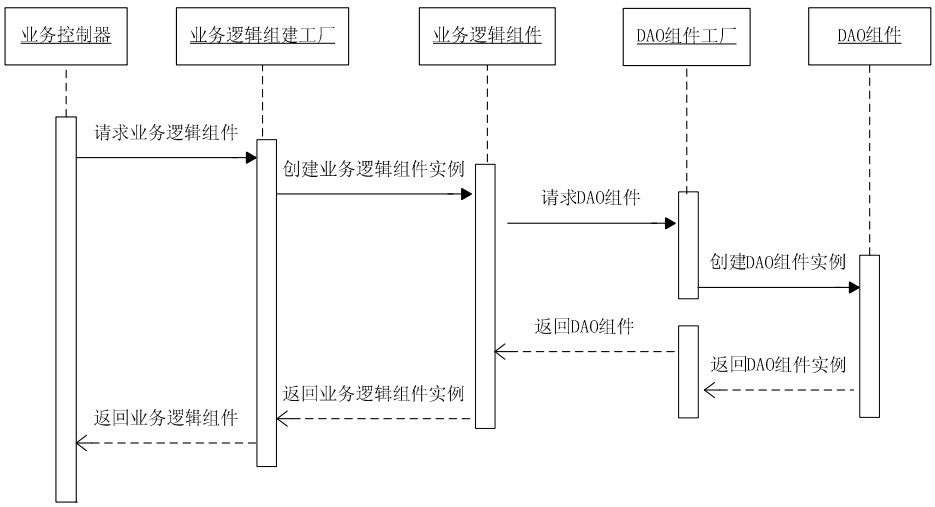



图 19.23 工厂模式顺序图

使用工厂模式确实可以大大降低各层之间的耦合度，但是同样也带来了代码编写的巨大困难。Spring 框架的出现，很好地解决了这个难题。在项目中整合 Spring 框架，可以使用 Spring 的 IoC 容器来管理控制器，并通过依赖注入的方式为控制器注入业务逻辑组件实例，依赖注入顺序如图 19.24 所示。

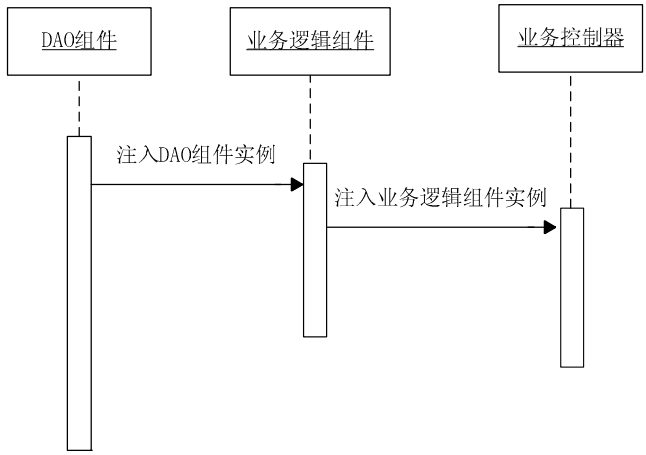



图 19.24 依赖注入顺序

### 19.2.2 安装 Spring 插件完成整合

要整合 Struts 2 和 Spring 框架，需要两个条件，第一个是为项目添加 Spring 框架支持，第二个是安装 Spring 插件。

首先我们来为项目添加 Spring 框架支持。第一步要在 Web 应用的 WEB-INF\lib 目录下添加 Spring 所需的库文件 spring-core-3.2.0.M2。所以我们要先在 Spring 的官方下载网站（http://www.springsource.org/download/）下载 Spring，具体下载方法如图 19.25 所示。在解压的 libs 包中找到 spring-core-3.2.0.M2 文件将其添加到我们所建项目的 WEB-INF\lib 文件夹下。


图 19.25　下载 Spring

第二步是修改 Web 的配置文件 web.xml，我们在其中添加一个 ContextLoaderListener 监听器，具体代码如图 19.26 所示。

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

图 19.26　web.xml 配置文件

通过添加该配置器，使得 Web 应用启动时会自动查找 WEB-INF 目录下的 applicationContext.xml 配置文件，并根据该配置文件来创建 Spring 容器。

为项目添加完 Spring 支持后，还需要为 Struts 2 安装 Spring 插件，从而完成 Struts 2 和 Spring 的整合。安装 Spring 插件非常简单，只需将 struts-2.3.4.1/lib 目录下的 struts2-spring-plugin-2.3.4.1.jar 复制到 WEB-INF/lib 目录下即可。

### 19.2.3　装配数据访问层

为项目添加了 Spring 框架支持后，就可以通过 Spring 来装配数据访问层，从而使得数据访问层由 Spring 的 IoC 容器来管理。

**【示例 19.7】**在项目的 WEB-INF 目录下，添加一个 Spring 的配置文件 applicationContext.xml，通过在该文件中配置数据访问层，从而完成数据访问层的装配，其具体代码如图 19.27 所示。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
       xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
    <bean id="bookDAO" class="dao.BookDAOImpl">
        </bean>
</beans>
```

装配数据访问层

图 19.27　配置文件 applicationContext.xml

## 19.2.4　装配业务逻辑层

**【示例 19.8】**在装配业务逻辑层之前，首先需要修改业务逻辑组件实现类，在该类添加一个数据访问层接口的类型，并为属性添加 get 和 set 方法。这样在装配业务逻辑层时，就可以在业务逻辑层中注入数据访问层，具体的修改方法如图 19.28 所示。

```
public class BookServiceImpl implements BookService{
        private BookDAO bookDAO;
        public BookDAO getBookDAO() {
                return bookDAO;
        }
        public void setBookDAO(BookDAO bookDAO) {
                this.bookDAO = bookDAO;
        }
        //省略业务逻辑层的其他方法
}
```

获得并设置数据访问层

图 19.28　对 BookServiceImpl.java 进行修改

修改完业务逻辑层代码，下面就可以装配该业务逻辑层，并为该业务逻辑层注入数据访问层实例对象，我们用图 19.29 表示如何装配业务逻辑层。

```
<bean id="bookService" class="service.BookServiceImpl">
        <property name="bookDAO" ref="bookDAO"></property>
</bean>
```

注入数据访问层

图 19.29　装配业务逻辑层

通过前面这两个步骤，这样在业务逻辑层中就可以直接使用数据访问层中的方法了，在这里就看不到工厂类的影子了。

## 19.2.5　装配业务控制器

业务控制是用来直接跟用户进行交互的，这时需要调用业务逻辑层的方法从而完成特定的业务逻辑。业务控制器同样需要在 Spring 中进行装配，并为其注入业务逻辑层实例对象。

**【示例 19.9】**在装配控制器之前，首先需要修改业务控制器类，在该类中添加一个业务逻辑层接口类型的属性，并为属性添加 get 和 set 方法。这样在装配业务控制器时，就可以为其

注入业务逻辑层，修改业务控制器类的方法如图 19.30 所示。

```
public class BookInputAction extends ActionSupport {
        private BookService bookService;
        private String bookName;
        private String bookNo;
        private double bookPrice;
        public BookService getBookService() {
                return bookService;
        }
        public void setBookService(BookService bookService) {
                this.bookService = bookService;
        }
        //省略其他属性的get和set方法
}
```

获得并设置业务逻辑层

图 19.30　修改业务控制器类

修改完业务控制器代码，下面就可以装配业务控制器，并为该业务控制器注入数据逻辑层实例对象，具体代码如图 19.31 所示。

```
<bean id="bookInputAction" class="action.BookInputAction">
        <property name="bookService" ref="bookService"></property>
</bean>
```

注入业务逻辑层

图 19.31　装配业务控制器

经过如上步骤，就完成了业务控制器 BookServiceAction 的装配和配置，读者可以参考这些步骤完成对 ShowAllBookAction 业务控制器的装配和配置。

## 19.3　Hibernate 和 Spring 整合开发

前面已经讲了 Struts 2 和 Hibernate、Spring 这两个框架的整合，同样 Hibernate 和 Spring 框架也能进行整合开发。整合后就能通过 Spring 来管理 Hibernate 连接数据库的数据源，还能管理 SessionFactory。在 Spring 框架中还提供了 HibernateTemplate 类和 HibernateDaoSupport 类来更方便地进行 Hibernate 操作。

### 19.3.1　使用 Spring 管理数据源

在整合前，我们需要在 Hibernate 的配置文件 hibernate.cfg.xml 中对数据源进行配置。然后在整合后，再在 Spring 的配置文件 applicationContext.xml 中进行配置，我们就可以使用 Spring 容器统一管理数据源了。

**【示例 19.10】**关于 applicationContext.xml 文件的配置，我们可以用图 19.32 来表示。

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName">
                <value>com.mysql.jdbc.Driver</value>
        </property>
        <property name="url">
                <value>jdbc:mysql://localhost/s2sh</value>
        </property>
        <property name="username">
                <value>root</value>
        </property>
        <property name="password">
                <value>123456</value>
        </property>
</bean>
```

设置数据库的各属性

图 19.32　applicationContext.xml 配置文件

## 19.3.2　使用 Spring 管理 SessionFactory

整合前我们都是使用 HibernateSessionFactory 来负责创建 SessionFactory，整合后就可以通过 Spring 来配置并管理 SessionFactory 了。通过装配的 SessionFactory，就能够为其他 DAO 组件的持久操作提供支持。

**【示例 19.11】**我们用图 19.33 所示的代码来在 applicationContext.xml 配置文件中配置 SessionFactory，在配置时需要指定数据源以及关系映射文件。

```
<bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource">
                <ref bean="dataSource"/>
        </property>
        <property name="hibernateProperties">
                <props>
                        <prop key="hibernate.dialect">
                          org.hibernate.dialect.MySQLDialect</prop>
                        <prop key="hibernate.show_sql">false</prop>
                        </props>
        </property>
        <property name="mappingResources">
                <list>
                        <value>po/Book.hbm.xml</value>
                </list>
        </property>
</bean>
```

Hibernate配置属性

关系映射文件

图 19.33　applicationContext.xml 配置文件

## 19.3.3　使用 HibernateTemplate 类

在 Spring 中提供了一个用于简化 Hibernate 操作的模板类 HibernateTemplate。HibernateTemplate 类中提供了一些简单的注入 find、load 或 delete 操作的方法，以及可选择的快捷函数来替换回调的实现。

【示例 19.12】要使用 HibernateTemplate 类，首先要在 Spring 中配置一个 HibernateTemplate Bean，并为其注入 SessionFactory 实例对象，具体的配置内容如图 19.34 所示。

```
<bean id="hibernateTemplate" class="org.springframework.orm.hibernate3.HibernateTemplate">
        <property name="sessionFactory">
                <ref bean="sessionFactory"/>
        </property>
</bean>
```

注入SessionFactory实例对象

图 19.34　applicationContext.xml 配置文件

## 19.3.4　使用 HibernateDaoSupport 类

在 Spring 中提供了一个用于简化 DAO 开发的类 HibernateDaoSupport。该类提供了一个 setSessionFactory 方法来接收一个 SessionFactory 对象，从而完成 SessionFactory 的注入。还提供了一个 setHibernateTemplate 方法来接收一个 HibernateTemplate 对象，从而完成 HibernateTemplate 的注入。同样，HibernateDaoSupport 类也提供了 getSessionFactory 和 getHibernateTemplate 方法来获得 SessionFactory 对象和 HibernateTemplate 对象。

【示例 19.13】我们通过一个实例来修改 DAO 实现类代码，使用该 DAO 实现类继承 HibernateDaoSupport。同时通过调用 HibernateTemplate 的方法来进行持久化访问操作，具体代码如图 19.35 所示。

```
package dao;
import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.springframework.dao.support.DaoSupport;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import hibernate.HibernateSessionFactory;
import po.Book;
public class BookDAOImpl extends HibernateDaoSupport implements BookDAO{
      public List<Book> findAllBook() {
            String hql = "from Book";
            List<Book> books = getHibernateTemplate().find(hql);
            return books;
      }
      public Book findByBookNo(String bookNo) {
            String hql = "from Book as book " +
                                    "where book.bookNo = ?";
            List<Book> books = getHibernateTemplate().find(hql,bookNo);
            if(books.size() > 0) {
                  return books.get(0);
            }else {
                  return null;
            }
      }
      public void saveBook(Book book) {
            getHibernateTemplate().save(book);
      }
}
```

查询所有的书籍

根据ISBN号查询书籍

图 19.35　BookDAOImpl.java 示例

在 BookDAOImpl 类中并没有创建 HibernateTemplate 实例对象，而是通过 Spring 为其注入

HibernateTemplate 实例对象。

【示例 19.14】我们使用如图 19.36 所示的代码来配置 BookDAOImpl，并为其注入 HibernateTemplate 实例对象，如图 19.36 所示。

```
<bean id="bookDAO" class="dao.BookDAOImpl">
        <property name="hibernateTemplate">
                <ref bean="hibernateTemplate"/>
        </property>
</bean>
```

注入HibernateTemplate实例对象

图 19.36　applicationContext.xml 配置文件

## 19.3.5　使用 Spring 管理事务管理器

在 TransactionTemplate 中必须包含一个 PlatformTransactionManager 实例对象。该实例可以通过代码来创建，也可以使用 Spring 来进行依赖注入。不管怎样，只有获得了 PlatformTransactionManager 的引用，TransactionTemplate 才能完成事务操作。

【示例 19.15】我们用图 19.37 来创建一个 HibernateTransactionManager 实例对象，并为其注入 SessionFactory 实例对象。

```
<bean id="hTManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory">
                <ref bean="sessionFactory"/>
        </property>
</bean>
```

注入SessionFactory实例对象

图 19.37　applicationContext.xml 配置文件

## 19.3.6　为业务逻辑层注入事务管理器

在 Spring 中创建完成事务管理器后，需要在业务逻辑层中添加一个属性，通过该属性来接收事务管理器。

【示例 19.16】我们通过如图 19.38 所示的代码为业务逻辑层添加 PlatformTransaction-Manager 类型的属性，从而能在业务逻辑层中使用事务管理器。

```
public class BookServiceImpl implements BookService{
    private BookDAO bookDAO;
    private PlatformTransactionManager transactionManager;
    public PlatformTransactionManager getTransactionManager() {
        return transactionManager;
    }
    public void setTransactionManager
        (PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
    public BookDAO getBookDAO() {
        return bookDAO;
    }
    public void setBookDAO(BookDAO bookDAO) {
        this.bookDAO = bookDAO;
    }
```

获得事务管理器

设置事务管理器

图 19.38　BookServiceImpl.java 示例

修改完业务逻辑层后，就可以为业务逻辑层注入事务管理器了，注入时只需要为业务逻辑组件 Bean 添加一个 transactionManager 属性配置，即可完成注入。

**【示例 19.17】**我们使用如图 19.39 所示的代码演示如何为业务逻辑层注入事务管理器。

```
<bean id="bookService" class="service.BookServiceImpl">
        <property name="transactionManager">
                <ref bean="hTManager"/>
        </property>
        <property name="bookDAO" ref="bookDAO"></property>
</bean>
```

注入事务管理器

注入数据访问层

图 19.39　applicationContext.xml 配置文件

## 19.3.7　使用 TransactionTemplate 进行事务管理

完成事务管理器的注入后，就可以通过该事务管理器来实例化一个 TransactionTemplate。他通过调用 TransactionTemplate 的 execute 方法就可以进行事务操作。

**【示例 19.18】**用来为业务逻辑层中的方法添加事务处理的代码如图 19.40 所示。

```
public boolean inputBook(final Book book) {
        Book oldBook = bookDAO.
                        findByBookNo(book.getBookNo());
        if(oldBook != null) {
                return false;
        }else {
          TransactionTemplate transactionTemplate =
                 new TransactionTemplate(this.transactionManager);
          transactionTemplate.execute(
                 new TransactionCallbackWithoutResult() {
  public void doInTransactionWithoutResult(TransactionStatus status) {
                        bookDAO.saveBook(book);
                    }
                }
          );
                return true;
        }
}
        public List<Book> showAllBook() {
                return bookDAO.findAllBook();
        }
}
```

按指定ISBN查询书籍

通过DAO保存该书籍

图 19.40　BookServiceImpl.java 示例

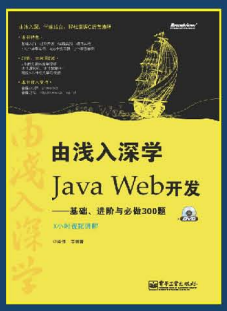
至此我们就完成了 Hibernate 和 Spring 整合开发了。

# 19.4　小结

本章是 Struts 2、Hibernate 以及 Spring 三个框架的整合开发章节，我们依次实现了 Struts 2 和 Hibernate 框架的整合开发、Struts 2 和 Spring 整合开发以及 Hibernate 和 Spring 整合开发。本章的重点和难点都是能否完成整合框架的开发配置，并实现不同框架的整合。通过本章的学习，希望读者能够领会各框架的整合思想，以便今后能够更好地实现具体的企业级开发。

## 好书分享

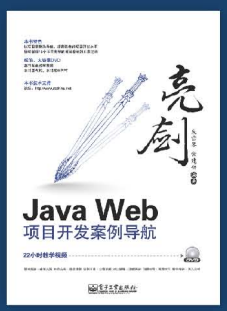

由浅入深学
Java
——基础、进阶与必做260题


由浅入深学
Java Web开发
——基础、进阶与必做300题


Java程序员
职场全攻略
——从小工到专家


从零
开始学
Java


亮剑
Java
项目开发案例导航


亮剑
Java Web
项目开发案例导航


搞定
J2EE
Struts+Spring+Hibernate
整合详解与典型案例


搞定
J2EE
核心技术与企业应用:
Ajax，JSP，Struts 2，Spring，Hibernate

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

·轻松学开发·


本书技术支持 www.rzchina.net

上架建议：程序开发>Java Web

ISBN 978-7-121-19558-7
9 787121 195587 >
定价：49.00元（含DVD光盘1张）



策划编辑：胡辛征
责任编辑：葛　娜　郑志宁
封面设计：侯士卿