



华章科技

全球知名Java技术专家（《How Tomcat Works》作者）亲自执笔，全面解读Servlet 和 JSP 最新技术

重点阐述Java Web开发的重要编程概念和设计模型，系统学习Servlet和JSP的必读著作



Servlet & JSP: A Tutorial

华章程序员书库

# Servlet和JSP学习指南

（加）Budi Kurniawan 著

崔毅 俞哲皆 俞黎敏 译



机械工业出版社  
China Machine Press

更多资源请访问稀酷客([www.ckook.com](http://www.ckook.com))

Servlet和JSP是开发Java Web应用程序的基础技术，为了有效地使用诸如JavaServer Faces、Struts 2、Spring MVC等框架，任何程序员都必须掌握Servlet和JSP。本书基于最新的Servlet 3.0和JSP 2.2技术，详细介绍Java Web开发的重要编程概念和设计模型，以及Servlet和JSP的最新版本中的相关技术及新功能。本书内容全面，包含大量可操作性极强的实例，是系统学习Servlet和JSP技术不可多得的参考指南！

## 本书主要内容：

- Servlet API和几个简单的Servlet
- Session追踪以及保持状态的4种技术
- JSP语法
- JSTL中最重要的类库
- 标签的具体编写方法以及标签文件
- Servlet中的事件驱动编程
- 过滤器和Model 2架构
- 如何利用Servlet 3的文件上传特性，以及如何在客户端改善用户体验
- 如何通过编程方式将资源发送到浏览器
- 如何利用Decorator模式以及类来改变Servlet请求和响应的行为
- Servlet 3中的新增功能
- 如何通过声明和编程方式来保护Java的Web应用程序
- 讨论Servlet/JSP应用程序的部署过程
- MVC框架的Struts 2



客服热线：(010) 88378991, 88361066  
购书热线：(010) 68326294, 88379649, 68995259  
投稿热线：(010) 88379604

读者信箱：hzjsj@hzbook.com  
华章网站：[www.hzbook.com](http://www.hzbook.com)  
网上购书：[www.china-pub.com](http://www.china-pub.com)



上架指导：计算机/程序设计/Java

ISBN 978-7-111-41861-0



9 787111 418610 >

定价：59.00元

013032829

TP312JA

1477

序员书库

Servlet & JSP: A Tutorial

# Servlet和JSP学习指南

(加) Budi Kurniawan 著

崔毅 俞哲皆 俞黎敏 译



北航

C1640837

TP312JA  
1477



机械工业出版社  
China Machine Press

7388800319

## 图书在版编目(CIP)数据

Servlet 和 JSP 学习指南 / (加) 克尼亞万 (Kurniawan, B.) 著; 崔毅, 俞哲皆, 俞黎敏译. —北京: 机械工业出版社, 2013.4  
(华章程序员书库)  
书名原文: Servlet & JSP: A Tutorial

ISBN 978-7-111-41861-0

I. S... II. ① 克… ② 崔… ③ 俞… ④ 俞… III. ① JAVA 语言 - 程序设计 - 指南 ② JAVA 语言 - 网页制作工具 - 程序设计 - 指南 IV. ① TP312-62 ② TP393.092-62

中国版本图书馆 CIP 数据核字 (2013) 第 053366 号

### 版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2012-4858

本书是系统学习 Servlet 和 JSP 的必读之作。由全球知名的 Java 技术专家 (《How Tomcat Works》作者) 亲自执笔, 不仅全面解读 Servlet 和 JSP 的最新技术, 重点阐述 Java Web 开发的重要编程概念和设计模型, 而且包含大量可操作性极强的案例。

本书共 18 章: 第 1 章介绍 Servlet API 和几个简单的 Servlet; 第 2 章讨论 Session 追踪, 以及保持状态的 4 种技术; 第 3 章和第 4 章系统讲解 JSP 的语法以及 JSP 中的重要特性之一: Expression Language; 第 5~7 章分别阐述 JSTL 中最重要的类库、标签的具体编写方法和标签文件; 第 8~10 章讨论 Servlet 中的事件驱动编程、过滤器, 以及 Model 2 架构; 第 11 章展示如何利用 Servlet 3 的文件上传特性, 以及如何在客户端改善用户的体验; 第 12 章解释如何通过编程方式将资源发送到浏览器; 第 13 章介绍如何利用 Decorator 模式以及类来改变 Servlet 请求和响应的行为; 第 14 章讨论 Servlet 3 中的一项新特性, 用来处理异步的操作; 第 15 章阐述如何通过声明和编程方式来保护 Java 的 Web 应用程序; 第 16 章讨论 Servlet/JSP 应用程序的部署过程, 以及部署描述符中的元素; 第 17 章阐述 Servlet 3 中的两项新特性; 第 18 章介绍 Struts 2 的用法。

Authorized translation from the English language edition entitled by Servlet & JSP: A Tutorial

Budi Kurniawan, published by Brainy Software , Inc, Copyright © 2012 .

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanic, including photocopying, recording, or by any information storage retrieval system, without permission of Brainy Software, Inc.

Chinese simplified language edition published by China Machine Press.

Copyright © 2013 by China Machine Press.

本书中文简体字版由 Brainy Software 授权机械工业出版社独家出版。未经出版者书面许可, 不得以任何方式复制或抄袭本书内容。

机械工业出版社(北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 谢晓芳

北京市荣盛彩色印刷有限公司印刷

2013 年 4 月第 1 版第 1 次印刷

186mm×240mm • 21.25 印张

标准书号: ISBN 978-7-111-41861-0

定 价: 59.00 元

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991 88361066

投稿热线: (010) 88379604

购书热线: (010) 68326294 88379649 68995259

读者信箱: hzjsj@hzbook.com

# 译者序

Sun 公司于 1996 年发布了 Java Servlet 技术，与 CGI (Common Gateway Interface，公共网关接口) 形成竞争，之后，它成为在 Web 中生成动态内容的标准。自从 Servlet 面世以来，也开发出了许多基于 Java 的 Web 框架，以帮助程序员更迅速地编写 Web 应用程序。目前全世界拥有了众多的 Java Web 开发人员，也是最热门的编程技术。作者 Budi Kurniawan 是 BrainySoftware.com 的高级架构师，也是《How Tomcat Works》、《Java for the Web with Servlets, JSP and EJB》以及《Struts 2 Design and Programming》、《Java 7: A Beginner's Tutorial》的作者。他已经发表了上百篇技术文章，并编写过授权给全球著名的大公司使用的软件。

## 本书读者对象

本书是针对有 Java 编程语言基础的 Web 开发者的，Java Web 应用程序开发是一种很成熟并且很热门的编程技术。同时，它也汇集各种技术于一身，经常令初学者不知道从何入手。如果你也有同感，那么本书就很适合你，因为它就是一本特意为初学者量身定制的教程。作为一套面对初学者的教程，本书不是要教会你每一种 Servlet/JSP 技术。如果你是一名有经验的 Web 应用开发者，对 Servlet 3.0 的新特性感兴趣，那么本书涵盖新特性的章节正是你所需要阅读并实践的内容。

## 章节简介

本书介绍了最重要的编程概念，并教你如何编写 Servlet/JSP，以及如何使用 Servlet 核心类库。对于编写真实的 Web 应用程序提供了很好的指导。本书是你所能找到的最全面的 Java Web 应用程序开发之入门教程，其主题有：

- Java Web 编程的核心技术 Servlet
- 4 种 Session 管理技术
- JSP
- 表达式语言
- JSTL 以及定制标签
- 定制标签文件
- Servlet 事件驱动编程之监听器
- 拦截请求的 Web 对象过滤器

- Model 2 架构
- 利用 Servlet 3.0 新特性进行文件上传以改善用户的体验
- 通过编程方式将资源发送到浏览器
- 利用 Decorator 模式等来改变 Servlet 请求和响应的行为
- Servlet 3.0 新特性之异步处理
- 通过声明和编程方式来保护 Java 的 Web 应用程序
- 应用程序的部署及描述符说明
- Servlet 3.0 新特性之动态注册和 Servlet 容器初始化

## 技术范围

本书旨在使你更全面、更专业地掌握 Java Web 应用程序开发技术，但是它只是一本比较基础的书籍，如果你已经有多年的 Java Web 应用程序开发经验，或许可以通过快速的阅读或者直接找到自己所需要的新技术点。当阅读本书时，你会遇到许多需要动手进行验证的实例，可以利用本书附带的示例程序进行练习与实践。示例与答案下载地址为：<http://books.brainysoftware.com/download/servletjsp.zip>。

正如在翻译过程中发现原著的错误一样，虽然我们在翻译过程中竭力以求信、达、雅，但限于自身水平，必定仍会有诸多不足，还望各位读者不吝指正。大家可以通过访问我的博客 <http://YuLimin.ItEye.com> 或者发送电子邮件到 YuLimin@163.com 进行互动。

关于术语的翻译，仍然沿用翻译《Effective Java 中文（第 2 版）》时采用的术语表以及满江红开放技术研究组织翻译术语，请见 <http://yulimin.iteye.com/blog/272088>。

感谢崔毅 (<http://cuiyi.javaeye.com/>) 对我在翻译中碰到的问题进行的深入讨论，并对本书翻译时所采用的术语进行了认真的磋商；感谢“满江红开放技术研究组织”的翻译同仁们在术语表讨论中提出许多中肯的建议；感谢满江红开源组织的曹晓钢提供的一些翻译注意事项和热情的帮助；感谢机械工业出版社的编辑认真仔细地审稿，辛苦了，谢谢！

本书由我组织翻译，崔毅负责翻译第 1 ~ 8 章，俞哲皆负责翻译第 9 ~ 14 章，我负责翻译前言、第 15 ~ 18 章、附录并对全书所有章节进行全面审校，还负责对原文中的错误与作者进行沟通并加以修正。参与翻译与审校的还有：杨春花、崔毅、张琬滢、蒋凌峰、魏伟、万国辉等，在此再次深表感谢。

本书章节安排合理，内容承上启下，但是需要边看书边动手做实验，才能充分理解并掌握 Java Web 应用程序开发技术及新特性。快乐分享，实践出真知，最后，祝大家能够像我一样在阅读中享受本书带来的乐趣！

Read a bit and take it out, then come back read some more.

俞黎敏

# 前　　言

欢迎你阅读本书，其内容涵盖了 Servlet 3.0 和 JSP 2.2 方面的技术。

Java Servlet 技术，或简称 Servlet，是 Java 中用于开发 Web 应用程序的基本技术。Sun 公司于 1996 年发布了 Java Servlet 技术，与 CGI（Common Gateway Interface，公共网关接口）形成竞争。之后，它成为在 Web 中生成动态内容的标准。CGI 的主要问题在于，它为每一个 HTTP 请求都创建一个新的进程。因为创建进程需要花费大量的 CPU 周期，这使得编写可扩展的 CGI 程序变得极为困难。另一方面，Servlet 程序也比 CGI 程序运行得更快，这是因为 Servlet 执行完它的第一个请求之后，就会驻留在内存中，等待后续的请求。

自从 Servlet 面世以来，也开发出了许多基于 Java 的 Web 框架，以帮助程序员更迅速地编写 Web 应用程序。这些框架可以使我们只关注业务逻辑，而不在编写样板代码（boilerplate code）上花费太多的时间。但你还是需要了解 Servlet 的基本知识。后来，JavaServer Pages（JSP）发布了，这使得编写 Servlet 变得更加轻松了。或许你正在使用一种很好的框架，如 Struts 2、Spring MVC，或者 JavaServer Faces。但是，如果没有充分理解 Servlet 和 JSP 方面的知识，你将无法进行高效的编码。顺便说一下，Servlets 是指在 Servlet 容器中运行的 Java 类。Servlet 容器或者 Servlet 引擎，就像是一个 Web 服务器，但它能够生成动态的内容，而不只是提供静态的资源。

目前的 Servlet 3.0 是在 JSR（Java Specification Request，Java 规范请求）315 中定义的 (<http://jcp.org/en/jsr/detail?id=315>)。它需要用 Java Standard Edition 6 或者更新的版本。JSP 2.2 则是在 JSR 245 中定义的 (<http://jcp.org/en/jsr/detail?id=245>)。本书假设你事先已经了解 Java 和面向对象编程。如果你是刚刚接触 Java，那么建议你先看一下笔者的另一本书：《Java 7: A Beginner's Tutorial》<sup>⊖</sup>，ISBN 978-0-9808396-1-6。

下一节将讨论 Servlet/JSP 应用程序架构（Application Architecture）、HTTP 协议，以及书中各章节的主要内容。

## Servlet/JSP 应用程序架构

Servlet 是一个 Java 程序。一个 Servlet 应用程序经常包含一个或者多个 Servlet。JSP 页面要被翻译成 Servlet，并进行编译。

Servlet 应用程序是在 Servlet 容器中运行的，它不能自动运行。Servlet 容器将用户的请

---

<sup>⊖</sup> 中文版书名：《Java 7 程序设计》，由机械工业出版社引进并出版。——编辑注

求传给 Servlet 应用程序，并将 Servlet 应用程序的响应回传给用户。大多数 Servlet 应用程序至少都包含几个 JSP 页面。因此，将 Java 的 Web 应用程序称为“Servlet/JSP 应用程序”比“Servlet 应用程序”更为恰当一些。

Web 用户可以利用如 Internet Explorer、Mozilla Firefox 或者 Google Chrome 这类 Web 浏览器来访问 Servlet 应用程序。一个 Web 浏览器相当于一个 Web 客户端。图 1 中展示了一个 Servlet/JSP 应用程序的架构。

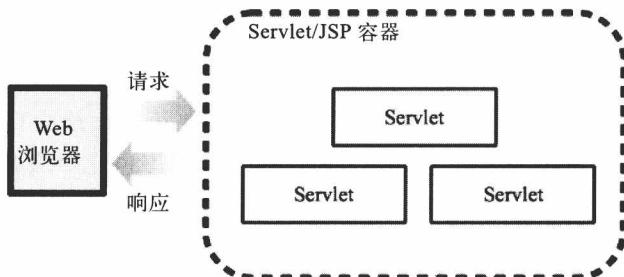


图 1 Servlet/JSP 应用程序架构

Web 服务器与 Web 客户端之间通过双方都熟悉的一种语言进行通信，即超文本转移协议（HyperText Transfer Protocol, HTTP）。为此，Web 服务器也称作 HTTP 服务器。关于 HTTP 的更多内容请参见下一节。

Servlet/JSP 容器是一种特殊的 Web 服务器，它可以处理 Servlet，并提供静态内容。在过去，人们更习惯于将 Servlet/JSP 容器作为 HTTP 服务器的一个模块来运行，如 Apache HTTP Server，因为他们认为 HTTP 服务器比 Servlet/JSP 容器更健壮。在这种情况下，Servlet/JSP 容器的任务是生成动态内容，HTTP 服务器则负责提供静态资源。如今，人们认为 Servlet/JSP 容器已经成熟，它们不需要 HTTP 服务器就可以进行广泛的部署。Apache Tomcat 和 Jetty 是其中最为盛行的两种 Servlet/JSP 容器，并且它们都是免费、开源的。可以分别从以下两个网站下载二者：<http://tomcat.apache.org> 和 <http://jetty.codehaus.org>。

Servlet 和 JSP 是 Java Enterprise Edition (EE) 定义的众多技术当中的两种。其他的 Java EE 技术还包括 Java Message Service (JMS)、Enterprise JavaBeans (EJB)、JavaServer Faces (JSF)，以及 Java Persistence。Java EE 6 (当前的最新版本) 的完整技术列表，可以在以下网站找到：

<http://www.oracle.com/technetwork/java/javaee/tech/index.html>

运行 Java EE 应用程序时，需要一个 Java EE 容器，如 GlassFish、JBoss、Oracle WebLogic，以及 IBM WebSphere。也可以将 Servlet/JSP 应用程序部署在 Java EE 容器中，只不过用 Servlet/JSP 容器已经绰绰有余，它还比 Java EE 容器更轻量化。Tomcat 和 Jetty 则不属于

Java EE 容器，因此它们不能运行 EJB 或者 JMS。

## 超文本转移协议 (HTTP)

HTTP 协议使得 Web 服务器与浏览器之间可以通过互联网 (Internet) 或者企业内部网 (Intranet) 来交换数据。万维网联盟 (World Wide Web Consortium, W3C) 是一个开发标准的国际化社区，它负责修订和维护这个协议。HTTP 的第一个版本是 HTTP 0.9，之后是 HTTP 1.0。再后来，HTTP 1.0 又被目前的 HTTP 1.1 取代。HTTP 1.1 是在 W3C 的 RFC (Request for Comments) 2616 中定义的，它可以通过以下网址下载到：<http://www.w3.org/Protocols/HTTP/1.1/rfc2616.pdf>。

Web 服务器每天 24 小时，每周 7 天不间断地运行，随时等待 HTTP 客户端（一般是 Web 浏览器）的连接和资源请求。在 HTTP 中，总是由客户端发起连接，服务器从不主动联系客户端。在查找某一个资源时，互联网用户是通过单击一个包含 URL (Uniform Resource Locator, 统一资源定位器) 的链接，或在他 / 她的浏览器地址栏中输入一个 URL 进行的。下面举 URL 的两个例子：

```
http://google.com/index.html  
http://facebook.com/index.html
```

URL 的第一部分是 **http**，它是协议的标识。并非所有的 URL 都用 HTTP。例如，下面这两个 URL 虽然没有使用 HTTP，但它们也都是有效的：

```
mailto:joe@example.com  
ftp://marketing@ftp.example.org
```

一般来说，URL 的格式是这样的：

```
protocol://[host.]domain[:port] [/context] [/resource] [?query string]
```

或者如下所示：

```
protocol://IP address[:port] [/context] [/resource] [?query string]
```

以上方括号中的内容都是可选的，因此 URL 也可以像 <http://yahoo.ca> 或者 <http://192.168.1.9> 这么简单。顺便说一下，IP 地址其实是分配给某一台计算机或者另一种设备的一个数字标签。一台计算机可以有多个域名，因此，几个域名也可以共用一个 IP 地址。换句话说，如果不用 <http://google.com>，也可以用它的 IP 地址：<http://209.85.143.99>。为了查出某个域名的 IP 地址，可以在计算机的控制台或命令行中使用 ping 命令，如：

```
ping google.com
```

IP 地址很难记住，因此人们更喜欢使用域名。但你知道吗？像 example.com 和 example.org 这样的域名是无法购买到的，因为它们是留着备用的，比如编写文档时作为示例用。

主机名 host 部分可以有，但与没有主机名 host 的地址相比，它表示的则是互联网或者内部网中另一个完全不同的位置。例如，`http://yahoo.com`（没有主机名 host）与 `http://mail.yahoo.com`（有主机名，为 mail）表示的是两个完全不同的位置。在过去，www 是最盛行的主机名，因此它变成是默认的。一般来说，`http://www.domainName` 就是指 `http://domainName`。

80 是 HTTP 的默认端口。因此，如果 Web 服务器是在 80 端口上运行，那么不需要端口号也能到达服务器。但是，有的时候，如果 Web 服务器不是在端口 80 上运行的，那么就需要输入端口号。例如，Tomcat 默认是在端口 8080 上运行，因此需要提供端口号：

```
http://localhost:8080
```

`localhost` 是一个保留名称，一般用来表示本机，即正在运行 Web 浏览器的这台计算机。

URL 中的 context 部分是指应用程序的名称，但这个也是可选的。Web 服务器可以运行多个 context 或多个应用程序，其中一个可以设置为默认的 context。如果要请求的是默认 context 中的资源，则 URL 中的 context 部分就可以忽略。

最后，一个 context 可以有一个或者多个默认资源（一般为 `index.html`、`index.htm` 或者 `default.htm`）。一个没有资源名称的 URL，通常被当作是默认资源。当然，如果一个 context 中存在多个默认资源，那么当客户端没有指定资源名称时，将总是返回优先级最高的那一个。

在资源名称之后，一般是一个或者多个查询字符串。查询字符串是指可以传到服务器进行处理的一个键 / 值对。后面的章节会介绍到更多关于查询字符串的内容。

接下来的几个小节将详细介绍 HTTP 请求和响应。

## HTTP 请求

一个 HTTP 请求中通常包含三个部分：

- 方法 / 统一资源标识符 (Uniform Resource Identifier, URI) / 协议 / 版本
- 请求标头
- 实体主体

下面是一个 HTTP 请求的示例：

```
POST /examples/default.jsp HTTP/1.1
Accept: text/plain; text/html
Accept-Language: en-gb
Connection: Keep-Alive
Host: localhost
```

```
User-Agent: Mozilla/5.0 (Macintosh; U; Intel Mac OS X 10.5; en-US;
  rv:1.9.2.6) Gecko/20100625 Firefox/3.6.6
Content-Length: 30
Content-Type: application/x-www-form-urlencoded
Accept-Encoding: gzip, deflate

lastName=Blanks&firstName=Mike
```

方法 /URI/ 协议版本号，放在请求的第一行。如：

```
POST /examples/default.jsp HTTP/1.1
```

这里的 POST 是请求方法，/examples/default.jsp 是 URI，HTTP/1.1 是协议 / 版本号部分。

HTTP 请求可以利用 HTTP 标准中定义的其中一个请求方法。HTTP 1.1 支持 7 种请求类型：GET、POST、HEAD、OPTIONS、PUT、DELETE 以及 TRACE。GET 和 POST 是互联网应用程序中最常用的。

URI 用于指定一个互联网资源，它通常解读为是相对于服务器的根目录。因此，它始终应该以一个正斜线 (/) 开头。统一资源定位器 (URL) 实际上也是一种 URI (详情请查看 <http://www.ietf.org/rfc/rfc2396.txt>)。

在一个 HTTP 请求中，请求标头包含关于客户端环境和请求实体主体的有用信息。例如，它可以包含为浏览器设置的语言、实体主体的长度等。各标头之间用一个回车换行 (Carriage Return/LineFeed, CRLF) 序列符隔开。

在标头和实体主体之间是一个空行 (CRLF)，它对于 HTTP 请求格式是很重要的。CRLF 告诉 HTTP 服务器，实体主体从这里开始。在有些互联网编程书籍中，它们把这个 CRLF 当作是 HTTP 请求的第四个部分。

在前一个 HTTP 请求中，实体主体只有下面这一行：

```
lastName=Blanks&firstName=Mike
```

实体主体常常比一个典型的 HTTP 请求更长。

## HTTP 响应

与 HTTP 请求类似，一个 HTTP 响应中通常也包含三个部分：

- 协议 / 状态码 / 描述
- 响应标头
- 实体主体

下面是一个 HTTP 响应的示例：

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
```

```
Date: Thu, 5 Jan 2012 13:13:33 GMT
Content-Type: text/html
Last-Modified: Wed, 4 Jan 2012 13:13:12 GMT
Content-Length: 112
```

```
<html>
<head>
<title>HTTP Response Example</title>
</head>
<body>
Welcome to Brainy Software
</body>
</html>
```

响应标头的第一行与请求标头的第一行类似。它在告诉我们，所使用的协议版本是 HTTP 1.1，并且请求成功（200 是成功状态码）。

响应标头也包含与 HTTP 请求中的标头类似的有用信息。响应的实体主体是响应本身的 HTML 内容。标头和实体主体之间用一系列的回车换行符（CRLF）隔开。

当且仅当 Web 服务器能够找到所请求的资源时，才会发出状态码 200。如果无法找到某个资源，或者无法理解请求，服务器就会发出一个不同的请求代码。例如，401 是未授权访问的状态码，405 表示未经允许的 HTTP 方法。关于 HTTP 状态码的完整列表，请查看以下在线文档：

<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

## 关于本书

下面介绍一下各章节的主要内容。

第 1 章介绍 Servlet API，还示范了几个简单的 Servlet。该章主要关注 Servlet API 4 个 Java 包当中的两个：javax.servlet 包和 javax.servlet.http 包。

第 2 章讨论 Session 追踪，或称作 Session 管理。由于 HTTP 的无状态性，因此这是 Web 应用程序开发中一个非常重要的主题。该章探讨保持状态的 4 种技术：改写 URL、隐藏域、cookie 以及 HttpSession 对象。

JavaServer Pages (JSP) 是 Servlet 的一种补充技术。第 3 章介绍 JSP 语法，包括它的指令、脚本元素以及动作。

第 4 章阐述 JSP 2.0 中增加的最重要特性之一：Expression Language (EL)。EL 旨在能够使我们设计出无脚本的 JSP 页面，并且能够协助你写出更简短、更高效的 JSP 页面。在该章中，你还会学到如何利用 EL 来访问 JavaBeans 和内置对象 (Scoped Object)。

第 5 章阐述 JavaServer Pages Standard Tag Library (JSTL) 中最重要的类库，这是一个

定制标签类库的集合，用于解决像迭代映射或集合、条件测试、XML 处理以及数据库访问和数据操作这类常见的问题。

通常情况下，是利用 JSTL 访问内置对象，以及完成 JSP 页面中的其他任务。但是，对于更为具体的任务，可能需要你自己编写定制标签。第 6 章就教你具体的编写方法。

第 7 章讨论标签文件，这是 JSP 2.0 中的一项新特性，它使得编写定制动作变得更加简单。该章讨论只利用标签文件来编写定制标签的几个方面。

第 8 章讨论 Servlet 中的事件驱动编程。其中讨论了 Servlet API 中的事件类和监听器，并介绍了如何编写监听器，以及如何在 Servlet/JSP 应用程序中使用它们。

第 9 章讲解过滤器，它们是拦截请求的 Web 对象。该章内容涵盖了 Filter API，其中包括 Filter、FilterConfig 及 FilterChain 接口，并介绍如何通过实现 Filter 接口来编写过滤器。

第 10 章解释 Model 2 架构，这是针对除最简单的 Java Web 应用程序之外的建议架构。该章还提供了几个范例，展示了 Model 2 应用程序中的不同组件。

第 11 章展示如何利用 Servlet 3 的文件上传特性，以及如何在客户端改善用户的体验。

第 12 章解释如何通过编程方式将一个资源发送到浏览器。

Servlet API 提供了用于包装 Servlet 请求和响应的类。在第 13 章中，将会学到如何利用 Decorator 模式及这些类来改变 Servlet 请求和响应的行为。

第 14 章讨论 Servlet 3.0 中的一项新特性，其用来处理异步的操作。如果 Servlet/JSP 应用程序中有一个或多个长时间运行的操作，那么这项特性就特别有帮助。该特性的原理是将那些操作分配给一个新的线程，从而将处理线程的请求释放回到池中，准备为另一个请求提供服务。

第 15 章阐述如何通过声明和编程方式来保护 Java 的 Web 应用程序。这里讨论了 4 个主要的安全性主题：验证、授权、保密性及数据完整性。

第 16 章讨论 Servlet/JSP 应用程序的部署过程，以及部署描述符中的元素。

第 17 章阐述 Servlet 3 中的两项新特性。动态注册是指不需要重启应用程序就可以动态地注册 Web 对象。框架开发者自然会喜欢 Servlet 容器初始化。

第 18 章介绍作为 MVC 框架的 Struts 2。该章阐述了 Struts 2 的基础组件和配置，并展示了一个简单的应用程序。

附录 A 阐述如何安装和配置 Tomcat，以及如何在各种操作系统中运行它。

附录 B 列出可以用来配置 Web 对象的所有注解，如 Servlet、监听器及过滤器。这些注解是 Servlet 3 的新特性，它们使部署描述符变成是可选的。

附录 C 阐述如何利用 KeyTool 程序生成公 / 私钥对，以及如何请一家可信任的证书颁发机构，将公钥变成一份数字证书。

## 下载应用程序范例

从以下网站可以下载到本书配套应用程序范例的压缩档：

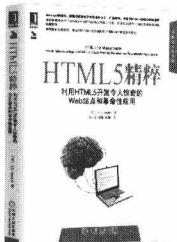
<http://books.brainysoftware.com/download>

## 选择框架

当你学完本书，并掌握了 Servlet 和 JSP 之后，最好至少选择学习一种 Web 框架。目前有许多优秀的框架可以免费使用，例如 Struts 2、JavaServer Faces、Spring MVC 及 Google Web Toolkit 等都不错。框架可以解决 Servlet/JSP 开发中常见的问题，大大缩短开发时间。

你可以根据工作需要，选择目前最需要的那些框架。

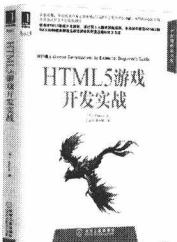
# 推荐阅读



## HTML 5精粹

Amazon畅销书，被翻译成西班牙语等多种文字，广受好评，被誉为HTML 5领域的经典著作

详尽地讲解和分析了HTML 5中的所有新特性和核心技术，能为有一定HTML基础的读者迅速提升HTML 5开发技能提供绝佳指导



## HTML 5游戏开发实战

清晰而全面地展示了如何使用最新的HTML 5和CSS 3标准来构建纸牌、绘图、物理等各种常见类型的游戏

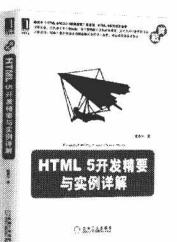
以实例为导向，系统介绍网络游戏开发技术，结合具体示例的操作步骤讲解，浅显易懂，适合网络游戏开发人员、管理人员阅读



## HTML 5游戏开发实践指南

来自美国硅谷的资深专家亲自执笔，包含大量实例，可操作性极强

全面讲解HTML 5游戏开发所需要掌握的各种最新技术、工具和框架（包含Canvas、SVG、WebGL等），以及开发的思维和方法



## HTML 5开发精要与实例详解

畅销书《HTML 5与CSS 3权威指南》姊妹篇，HTML 5实战进阶必备

注重实战：包含28个中大型案例，每个案例既有直观的效果图，又有详尽的源代码分析

讲解透彻：对每个案例所涉及的理论精要进行深入剖析，理论与实践完美结合



## HTML 5实战

依据HTML 5标准的最新草案编写，对HTML 5进行了系统、全面、透彻的讲解

106个精心设计的经典案例对各个知识点进行补充和阐释，理论与实践完美结合

# 推荐阅读



## 标签：标记系统设计实践

理论完备，深入阐释了标记系统的价值和原理，实践性强，全面讲解了标记系统的架构设计方法和实现细节。

## jQuery Mobile权威指南

由资深专家根据jQuery Mobile最新版本撰写，对jQuery Mobile的所有功能、特性、使用方法和开发技巧进行了全面而透彻的讲解，是系统学习jQuery Mobile的权威参考书。

## PhoneGap实战

来自腾讯的资深专家团队撰写，Adobe中国区专家和



北航

C1640837

## Ext JS权威指南

Ext JS领域的集大成之作，全面系统地讲解Ext JS的开发方法与技巧，包含大量案例和最佳实践，系统学习和开发参考必备。

## Sencha Touch权威指南

资深HTML 5专家根据Sencha Touch最新版撰写，对Sencha Touch的所有功能、特性、使用方法和开发技巧进行了全面而深入的讲解，是系统学习Sencha Touch的权威参考书。

## 优秀网站设计（第3版）

经典畅销书全新升级，10余年长销不衰，被翻译为近10种文字在全球范围内出版，广受好评。

# 目 录

译者序

前 言

## 第 1 章 Servlet / 1

- 1.1 Servlet API 概述 / 1
- 1.2 Servlet / 2
- 1.3 编写基础的 Servlet 应用程序 / 3
- 1.4 ServletRequest / 6
- 1.5 ServletResponse / 7
- 1.6 ServletConfig / 8
- 1.7 ServletContext / 10
- 1.8 GenericServlet / 11
- 1.9 HTTP Servlet / 13
- 1.10 处理 HTML 表单 / 15
- 1.11 使用部署描述符 / 21
- 1.12 小结 / 23

## 第 2 章 Session 管理 / 24

- 2.1 网址重写 / 24
- 2.2 隐藏域 / 29
- 2.3 cookie / 34
- 2.4 HttpSession 对象 / 42
- 2.5 小结 / 51

## 第 3 章 JSP / 52

- 3.1 JSP 概述 / 52
- 3.2 备注 / 57
- 3.3 隐式对象 / 57

3.4 指令 / 60

3.5 脚本元素 / 63

3.6 动作 / 68

3.7 小结 / 71

## 第 4 章 EL / 72

- 4.1 EL 语法 / 72
- 4.2 访问 JavaBean / 74
- 4.3 EL 隐式对象 / 75
- 4.4 使用其他 EL 运算符 / 78
- 4.5 使用 EL / 80
- 4.6 在 JSP 2.0 及更高版本中配置 EL / 83
- 4.7 小结 / 85

## 第 5 章 JSTL / 86

- 5.1 下载 JSTL / 86
- 5.2 JSTL 类库 / 86
- 5.3 通用动作指令 / 87
- 5.4 条件式动作指令 / 91
- 5.5 iterator 动作指令 / 93
- 5.6 格式化动作指令 / 102
- 5.7 函数 / 109
- 5.8 小结 / 114

## 第 6 章 编写定制标签 / 115

- 6.1 定制标签概述 / 115
- 6.2 简单的标签处理器 / 116

6.3 SimpleTag 范例 / 116	10.2 Model 2 概述 / 170
6.4 处理属性 / 119	10.3 基于 Servlet Controller 的 Model 2 / 172
6.5 管理标签主体 / 122	10.4 基于 Filter Dispatcher 的 Model 2 / 181
6.6 编写 EL 函数 / 125	10.5 验证器 / 184
6.7 发布定制标签 / 126	10.6 数据库访问 / 189
6.8 小结 / 128	10.7 依赖注入 / 199
<b>第 7 章 标签文件 / 129</b>	10.8 小结 / 208
7.1 标签文件简介 / 129	<b>第 11 章 文件上传 / 209</b>
7.2 我们的第一个标签文件 / 130	11.1 客户端编程 / 209
7.3 标签文件指令 / 131	11.2 服务器端编程 / 210
7.4 doBody / 139	11.3 上传 Servlet 范例 / 212
7.5 invoke / 141	11.4 多文件上传 / 214
7.6 小结 / 143	11.5 上传客户端 / 217
<b>第 8 章 监听器 / 144</b>	11.6 小结 / 223
8.1 监听器接口和注册 / 144	<b>第 12 章 文件下载 / 224</b>
8.2 Servlet Context 监听器 / 145	12.1 文件下载概述 / 224
8.3 Session 监听器 / 148	12.2 范例 1：隐藏资源 / 225
8.4 ServletRequest 监听器 / 153	12.3 范例 2：防止跨站引用 / 230
8.5 小结 / 155	12.4 小结 / 232
<b>第 9 章 过滤器 / 156</b>	<b>第 13 章 请求和响应的装饰 / 233</b>
9.1 Filter API / 156	13.1 Decorator 模式 / 233
9.2 过滤器的配置 / 157	13.2 Servlet Wrapper 类 / 234
9.3 范例 1：日志过滤器 / 159	13.3 范例：AutoCorrect 过滤器 / 235
9.4 范例 2：图片保护过滤器 / 163	13.4 小结 / 242
9.5 范例 3：下载计数过滤器 / 164	
9.6 过滤器的顺序 / 168	
9.7 小结 / 169	
<b>第 10 章 应用程序设计 / 170</b>	<b>第 14 章 异步处理 / 243</b>
10.1 Model 1 概述 / 170	14.1 概述 / 243

- 14.2 编写异步的 Servlet 和 Filter / 243
- 14.3 编写异步的 Servlet / 244
- 14.4 异步监听器 / 249
- 14.5 小结 / 251

## 第 15 章 安全性 / 252

- 15.1 验证和授权 / 252
- 15.2 验证方法 / 256
- 15.3 SSL / 263
- 15.4 通过编程确保安全性 / 268
- 15.5 小结 / 271

## 第 16 章 部署 / 272

- 16.1 部署描述符概述 / 272
- 16.2 部署 / 284
- 16.3 Web Fragment / 285
- 16.4 小结 / 287

## 第 17 章 动态注册和 Servlet 容器初始化 / 288

- 17.1 动态注册 / 288
- 17.2 Servlet 容器初始化 / 291
- 17.3 小结 / 293

## 第 18 章 Struts 2 简介 / 294

- 18.1 Struts 2 的优势 / 294
- 18.2 Struts 2 工作原理 / 295
- 18.3 拦截器 / 297
- 18.4 Struts 2 的配置文件 / 299
- 18.5 简单的 Struts 应用程序 / 307
- 18.6 小结 / 311

## 附录 A Tomcat / 312

## 附录 B Web 注解 / 316

## 附录 C SSL 证书 / 320

# 第 1 章 Servlet

Servlet 是开发 Servlets 应用程序的主要技术。掌握 Servlet API 是成为一名技术高超的 Java Web 开发者的基础。你必须非常熟悉 Servlet API 中所定义的 70 多种类型。这个数字听起来似乎不少，但是如果你每次学一种，就不会觉得困难了。

本章将介绍 Servlet API，并教你编写第一个 Servlet 应用程序。

## 1.1 Servlet API 概述

Servlet API 中有 4 个 Java 包，包括：

- `javax.servlet`。包含定义 Servlet 与 Servlet 容器之间契约的类和接口。
  - `javax.servlet.http`。包含定义 HTTP Servlet 与 Servlet 容器之间契约的类和接口。
  - `javax.servlet.annotation`。包含对 Servlet、Filter 和 Listener 进行标注的注解。它还为标注元件指定元数据。
  - `javax.servlet.descriptor`。包含为 Web 应用程序的配置信息提供编程式访问的类型。
- 本章主要关注 `javax.servlet` 和 `javax.servlet.http` 包中的成员。

### javax.servlet 包

图 1-1 展示了 `javax.servlet` 中的主要类型。

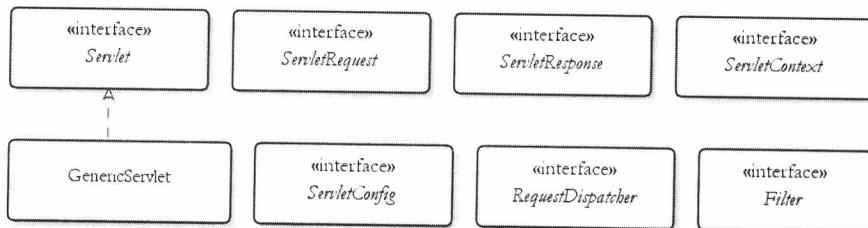


图 1-1 `javax.servlet` 包中的主要成员

Servlet 技术的核心是 `Servlet` 接口，这是所有 Servlet 类都必须直接或者间接实现的一个接口。当编写实现 `Servlet` 接口的 Servlet 类时，直接实现它；当扩展一个实现这个接口的类时，则间接实现它。

`Servlet` 接口定义了 Servlet 与 Servlet 容器之间的一个契约。这个契约归结起来是说，Servlet 容器会把 Servlet 类加载到内存中，并在 Servlet 实例中调用特定的方法。在一个应用

程序中，每个 Servlet 类型只能有一个实例。

用户的请求会引发 Servlet 容器调用一个 Servlet 的 service 方法，并给这个方法传入一个 ServletRequest 实例和一个 ServletResponse 实例。ServletRequest 封装当前的 HTTP 请求，以便 Servlet 的开发者不必解析和操作原始的 HTTP 数据。ServletResponse 表示当前用户的 HTTP 响应，它的作用是使得将响应回传给用户更容易。

Servlet 容器还为每个应用程序创建一个 ServletContext 实例。这个对象封装 context(应用程序) 的环境细节。每个 context 只有一个 ServletContext。每个 Servlet 实例还有一个封装 Servlet 配置信息的 ServletConfig。

接下来我们先看一下 Servlet 接口。上面提到过的其他接口将在本章的其他小节中讨论。

## 1.2 Servlet

Servlet 接口定义了以下 5 个方法。

```
void init(ServletConfig config) throws ServletException
void service(ServletRequest request, ServletResponse response)
    throws ServletException, java.io.IOException
void destroy()
java.lang.String getServletInfo()
ServletConfig getServletConfig()
```

注意，编写 Java 方法签名的规则是：与包含该方法的类型不在同一个包中的类型，要使用全类名。如 service 方法 javax.servlet.ServletException 的签名，由于它与 Servlet 接口处在同一个包中，因此编写的时候不需要包信息，用 ServletException 即可；但是由于 java.io.Exception 类不在同一个包中，因此需要用全类名才行，即 java.io.Exception。<sup>⊖</sup>

init、service 和 destroy 方法属于 Servlet 生命周期方法。Servlet 容器将根据以下原则调用这三个方法：

- **init**。第一次请求 Servlet 时，Servlet 容器就会调用这个方法。在后续的请求中，将不再调用该方法。可以利用这个方法来编写一些应用程序初始化相关的代码。在调用这个方法时，Servlet 容器会传递一个 ServletConfig。一般来说，会将 ServletConfig 赋给一个类级变量，以便 Servlet 类中的其他方法也可以使用这个对象。
- **service**。每次请求 Servlet 时，Servlet 容器都会调用这个方法。必须在这里编写要 Servlet 完成的相应代码。第一次请求 Servlet 时，Servlet 容器会调用 init 方法和

---

<sup>⊖</sup> 如果在类的 import 部分进行了 import java.io.Exception 声明，那么这里就不需要使用全类名了。——译者注

`service` 方法。对于后续的请求，则只调用 `service` 方法。

- `destroy`。要销毁 Servlet 时，Servlet 容器就会调用这个方法。它通常发生在卸载应用程序，或者关闭 Servlet 容器的时候。一般来说，可以在这个方法中编写一些资源清理相关的代码。

Servlet 中的另外两个方法是非生命周期方法：`getServletInfo` 和 `getServletConfig`。

- `getServletInfo`。该方法返回 Servlet 的描述。可以返回可能有用的任意字符串，甚至是 `null`。

- `getServletConfig`。该方法返回由 Servlet 容器传给 `init` 方法的 `ServletConfig`。但是，为了让 `getServletConfig` 返回非 `null` 值，你肯定已经为传给 `init` 方法的 `ServletConfig` 赋给了一个类级变量。`ServletConfig` 将在 1.6 节讨论。

必须注意的一点是线程安全性。一个应用程序中的所有用户将共用一个 Servlet 实例，因此不建议使用类级变量，除非它们是只读的，或者是 `java.util.concurrent.atomic` 包中的成员。

下一节将介绍如何编写 Servlet 实现。

## 1.3 编写基础的 Servlet 应用程序

Servlet 应用程序编写起来非常简单，只需要创建一个目录结构，并将 Servlet 类放在某一个目录下即可。在本节中，将学习如何编写一个简单的 Servlet 应用程序，将它命名为 `app01a`。最初它只包含一个 Servlet： `MyServlet`，其会给用户发送一条问候信息。

需要用一个 Servlet 容器来运行 Servlet。Tomcat 是一个开源的 Servlet 容器，可以免费获得，它也可以在能够使用 Java 的任何操作系统平台上运行。如果你还没有安装 Tomcat，现在应该先去阅读附录 A，并安装 Tomcat。

### 1.3.1 编写和编译 Servlet 类

确定你的机器上已经有了 Servlet 容器之后，下一步就是编写和编译 Servlet 类。这个例子中的 Servlet 类是指 `MyServlet`，如代码清单 1-1 所示。按照规范，Servlet 类的名称要以 `Servlet` 作为后缀。

代码清单 1-1 MyServlet 类

---

```
package app01a;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.Servlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
```

```

import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebServlet;

@WebServlet(name = "MyServlet", urlPatterns = { "/my" })
public class MyServlet implements Servlet {

    private transient ServletConfig servletConfig;

    @Override
    public void init(ServletConfig servletConfig)
        throws ServletException {
        this.servletConfig = servletConfig;
    }

    @Override
    public ServletConfig getServletConfig() {
        return servletConfig;
    }

    @Override
    public String getServletInfo() {
        return "My Servlet";
    }

    @Override
    public void service(ServletRequest request,
                        ServletResponse response) throws ServletException,
                        IOException {
        String servletName = servletConfig.getServletName();
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.print("<html><head></head>" +
                    "<body>Hello from " + servletName +
                    "</body></html>");
    }

    @Override
    public void destroy() {
    }
}

```

---

在阅读代码清单 1-1 中的代码时，可能首先会注意到下面这个注解：

```
@WebServlet(name = "MyServlet", urlPatterns = { "/my" })
```

用 `WebServlet` 注解类型来声明一个 `Servlet`。在声明 `Servlet` 的同时，还可以告诉容器是哪个 URL 调用这个 `Servlet`。`name` 属性是可选的，如果有，一般是用来提供 `Servlet` 类的名称。关键是 `urlPatterns` 属性，它也是可选的，但是几乎都会用到它。在 `MyServlet` 中，`urlPattern` 告诉容器，`/my` 模式应该调用这个 `Servlet`。

注意，URL 模式必须以一条正斜线开头。

Servlet 的 init 方法调用一次，并将 private transient 变量 servletConfig 设置成传给该方法的 ServletConfig 对象。

```
private transient ServletConfig servletConfig;

@Override
public void init(ServletConfig servletConfig)
    throws ServletException {
    this.servletConfig = servletConfig;
}
```

如果你想从 Servlet 内部使用 ServletConfig，只须将传进来的 ServletConfig 赋给一个类变量即可。

service 方法将字符串“Hello from MyServlet”传给浏览器。每当有针对该 Servlet 的 HTTP 请求进来时，都会调用 service 方法。

要编译 Servlet，必须将包含 Servlet API 的类型放在类路径（Class Path）中。Tomcat 的 servlet-api.jar 文件中打包有 javax.servlet 和 javax.servlet.http 包的成员。这个 jar 文件就放在 Tomcat 安装目录的 lib 目录下。

### 1.3.2 应用程序的目录结构

Servlet 应用程序必须以特定的目录结构进行部署。图 1-2 展示了这个应用程序的目录结构。

这个目录结构最上方的 app01a 就是应用程序目录。应用程序目录下方是一个 WEB-INF 目录。它有两个子目录：

❑ classes。Servlet 类和其他的 Java 类都必须放在这里。

类下方的目录反映了类的包结构。在图 1-2 中，已经部署了一个类：app01a.MyServlet。

❑ lib。在这里部署 Servlet 应用程序所需的 jar 文件。

Servlet API jar 文件则不需要部署在这里，因为 Servlet 容器已经包含这些 Servlet API 了。在这个应用程序中，lib 目录是空的。空的 lib 目录也可以删除。

Servlet/JSP 应用程序一般会有 JSP 页面、HTML 文件、图像文件以及其他资源。这些都应该放在应用程序的目录下，并且经常放在子目录下。例如，所有的图像文件可以放在一个 image 目录下，所有的 JSP 页面可以放在一个 jsp 目录下，以此类推。

放在应用程序目录下的任何资源，用户都可以通过输入该资源的 URL 而直接进行访问。如果你希望某个资源可以被 Servlet 访问，但是不能被用户访问，那么应该把它放在 WEB-INF 目录下面。

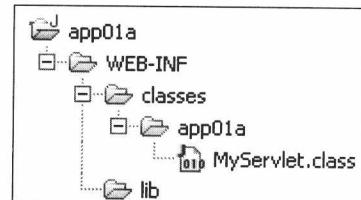


图 1-2 应用程序的目录结构

现在，把这个应用程序部署到 Tomcat 中。在 Tomcat 中，一种部署方法是将应用程序目录直接复制到 Tomcat 安装目录下方的 **webapps** 目录下。也可以通过在 Tomcat 的 **conf** 目录下编辑 **server.xml** 文件来部署应用程序；或者为了不用编辑 **server.xml**，而单独部署一个 XML 文件到 **conf\Catalina\localhost** 目录下。其他 Servlet 容器可能会有不同的部署规则。关于如何将一个 Servlet/JSP 应用程序部署到 Tomcat 的详细信息，可查看附录 A。

这里建议部署 Servlet/JSP 应用程序的方法是将它打包成一个 war 文件来进行部署。war 文件是指以 **war** 作为扩展名的 jar 文件。可以利用 JDK 提供的 jar 程序或者像 WinZip 这类工具来创建 war 文件。然后，将 war 文件复制到 Tomcat 的 **webapps** 目录下。当启动或者重启 Tomcat 时，Tomcat 会自动解压 war 文件。以 war 文件的方式部署，这在所有的 Servlet 容器中都适用。在第 16 章中，你还会学到更多关于部署的知识。

### 1.3.3 调用 Servlet

要测试你的第一个 Servlet 应用程序，需先启动或者重启 Tomcat，并将浏览器切换到以下 URL（假设将 Tomcat 配置成在它的默认端口 8080 上进行监听）：

`http://localhost:8080/app01a/my`

其输出应该如图 1-3 所示。



图 1-3 MyServlet 的响应

恭喜！你的第一个 Servlet 应用程序已经写好了！

### 1.4 ServletRequest

对于每一个 HTTP 请求，Servlet 容器都会创建一个 **ServletRequest** 实例，并将它传给 Servlet 的 **service** 方法。**ServletRequest** 封装有关请求的信息。

下面是 **ServletRequest** 接口中的部分方法。

```
public int getContentLength()
```

返回请求主体中的字节数。如果不知道字节的长度，该方法将返回 -1。

```
public java.lang.String getContentType()
```

返回请求主体的 MIME 类型，如果不知道类型，则返回 null。

```
public java.lang.String getParameter(java.lang.String name)
```

返回指定请求参数的值。

```
public java.lang.String getProtocol()
```

返回这个 HTTP 请求的协议名称和版本号。

**getParameter** 是 **ServletRequest** 中最常用的方法。该方法通常用来返回一个 HTML 表单域的值。1.10 节将介绍如何获取表单值。

**getParameter** 也可以用来获取查询字符串的值。例如，如果利用下面这个 URI 调用一个 Servlet：

```
http://domain/context/servletName?id=123
```

将可以在 Servlet 中利用下面这个语句来获取 id 的值：

```
String id = request.getParameter("id");
```

注意，如果该参数不存在，那么 **getParameter** 将返回 null。

除了 **getParameter** 之外，还可以利用 **getParameterNames**、**getParameterMap** 和 **getParameterValues** 来获取表单域的名称和值，以及查询字符串。关于如何使用这些方法的范例参见 1.9 节。

## 1.5 ServletResponse

**javax.servlet.ServletResponse** 接口表示一个 Servlet 响应。在调用一个 Servlet 的 **service** 方法之前，Servlet 容器会先创建一个 **ServletResponse**，并将它作为第二个参数传给 **service** 方法。**ServletResponse** 隐藏了将响应发给浏览器的复杂性。

**ServletResponse** 中定义的其中一个方法是 **getWriter** 方法，它返回可以将文本传给客户端的 **java.io.PrintWriter**。在默认情况下，**PrintWriter** 对象采用 ISO-8859-1 编码。

在将响应发送给客户端时，通常将它作为 HTML 发送。因此，你对 HTML 一定要非常熟悉。

**提示** 还有一个方法可以用来将输出传给浏览器：`getOutputStream`。但是，这个方法是用来传输二进制数据的，因此，在大多数时候，需要使用`getWriter`，而不是`getOutputStream`。关于如何传输二进制内容的说明，可查看第12章中关于文件下载的内容。

在发送任何HTML标签之前，应该先通过调用`setContentType`方法来设置响应的内容类型，比如，将`text/html`作为参数传递，这是在告诉浏览器内容类型为HTML。如果没有设置内容类型，那么大多数浏览器将会默认以HTML的形式显示响应的内容。但是，如果没有设置响应的内容类型，有些浏览器则会将HTML标签显示为普通文本。

代码清单1-1的`MyServlet`中使用了`ServletResponse`。在本章及后续章节的其他应用程序中，你还会看到它的用法。

## 1.6 ServletConfig

在Servlet容器初始化Servlet时，Servlet容器将`ServletConfig`传给Servlet的`init`方法。`ServletConfig`封装可以通过`@WebServlet`或者部署描述符传给一个Servlet的配置信息。以这种方式传递的每一条信息都称作初始参数。初始参数有两个组成部分：键和值。

为了从一个Servlet内部获取某个初始参数的值，应该在由Servlet容器传给Servlet的`init`方法的`ServletConfig`中调用`getInitParameter`方法。`getInitParameter`方法的签名如下：

```
java.lang.String getInitParameter(java.lang.String name)
```

此外，`getInitParameterNames`方法则是返回所有初始参数名称的一个`Enumeration`：

```
java.util.Enumeration<java.lang.String> getInitParameterNames()
```

例如，要获取`contactName`参数值，利用这个：

```
String contactName = servletConfig.getInitParameter("contactName");
```

除了`getInitParameter`和`getInitParameterNames`之外，`ServletConfig`还提供了另一个很有用的方法：`getServletContext`。可以利用这个方法从Servlet内部获取`ServletContext`。关于这个对象的讨论，参见1.7节。

举个`ServletConfig`的例子。在app01a中添加一个`ServletConfigDemoServlet`，这个新的Servlet如代码清单1-2所示。

代码清单1-2 `ServletConfigDemoServlet`类

---

```
package app01a;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.Servlet;
```

```
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;

@WebServlet(name = "ServletConfigDemoServlet",
    urlPatterns = { "/servletConfigDemo" },
    initParams = {
        @WebInitParam(name="admin", value="Harry Taciak"),
        @WebInitParam(name="email", value="admin@example.com")
    }
)
public class ServletConfigDemoServlet implements Servlet {
    private transient ServletConfig servletConfig;

    @Override
    public ServletConfig getServletConfig() {
        return servletConfig;
    }

    @Override
    public void init(ServletConfig servletConfig)
        throws ServletException {
        this.servletConfig = servletConfig;
    }

    @Override
    public void service(ServletRequest request,
        ServletResponse response)
        throws ServletException, IOException {
        ServletConfig servletConfig = getServletConfig();
        String admin = servletConfig.getInitParameter("admin");
        String email = servletConfig.getInitParameter("email");
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.print("<html><head></head><body>" +
            "Admin:" + admin +
            "<br/>Email:" + email +
            "</body></html>");
    }

    @Override
    public String getServletInfo() {
        return "ServletConfig demo";
    }

    @Override
    public void destroy() {
    }
}
```

如代码清单 1-2 所示，在 @WebServlet 的 initParams 属性中给 Servlet 传递了两个参数（admin 和 email）：

```
@WebServlet(name = "ServletConfigDemoServlet",
    urlPatterns = { "/servletConfigDemo" },
    initParams = {
        @WebInitParam(name="admin", value="Harry Taciak"),
        @WebInitParam(name="email", value="admin@example.com")
    }
)
```

可以利用下面这个 URL 调用 ServletConfigDemoServlet：

`http://localhost:8080/app01a/servletConfigDemo`

其结果应该如图 1-4 所示。

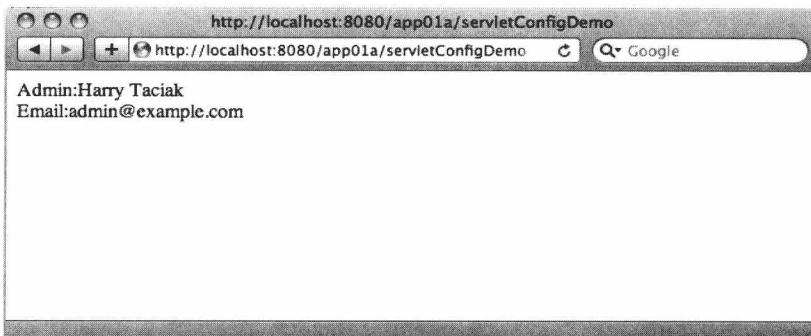


图 1-4 ServletConfigDemoServlet 示例

还可以在部署描述符中传递初始参数。利用部署描述符来完成这项工作，比用 @WebServlet 更容易些，因为部署描述符是一个文本文件，不需要重新编译 Servlet 类就可以进行编辑。

关于部署描述符的讨论，可查看 1.11 节，以及 16.2 节。

## 1.7 ServletContext

ServletContext 表示 Servlet 应用程序。每个 Web 应用程序只有一个 context。在分布式环境中，一个应用程序同时部署到多个容器中，并且每台 Java 虚拟机都有一个 ServletContext 对象。

在 ServletConfig 中调用 getServletContext 方法可以获得 ServletContext。

有了 ServletContext 之后，就可以共享能通过应用程序的所有资源访问的信息，促进 Web 对象的动态注册。前者是通过将一个内部 Map 中的对象保存在 ServletContext 中来实

现的。保存在 **ServletContext** 中的对象称作属性 (attribute)。

**ServletContext** 中的下列方法是用于处理属性的：

```
java.lang.Object getAttribute(java.lang.String name)
java.util.Enumeration<java.lang.String> getAttributeNames()
void setAttribute(java.lang.String name, java.lang.Object object)
void removeAttribute(java.lang.String name)
```

关于这些方法的范例可查看第 8 章的内容。利用 **ServletContext** 动态注册 Web 对象的内容，可查看第 17 章的内容。

## 1.8 GenericServlet

前面的例子展示了如何通过实现 **Servlet** 接口来编写 **Servlet**。但你是否注意到，必须为 **Servlet** 中的所有方法都提供实现，即使它们中有些不含代码的方法也是一样？此外，还需要将 **ServletConfig** 对象保存在一个类级变量中。

值得庆幸的是，我们有了 **GenericServlet** 抽象类。它本着在面向对象编程中易于编写代码的精神，实现了 **Servlet** 和 **ServletConfig**，并完成以下工作：

- 将 **init** 方法中的 **ServletConfig** 赋给一个类级变量，使它可以通过调用 **getServletConfig** 来获取。
- 为 **Servlet** 接口中的所有方法提供默认实现。
- 提供方法来包装 **ServletConfig** 中的方法。

**GenericServlet** 通过在 **init** 方法中将 **ServletConfig** 对象赋给一个类级变量 **servletConfig**，实现对 **ServletConfig** 的保存。下面就是 **init** 方法在 **GenericServlet** 中的实现。

```
public void init(ServletConfig servletConfig)
    throws ServletException {
    this.servletConfig = servletConfig;
    this.init();
}
```

但是，如果在类中覆盖了这个方法，则调用 **Servlet** 中的 **init** 方法，并且必须调用 **super.init(servletConfig)** 来保存 **ServletConfig**。为了避免这么做，**GenericServlet** 又另外提供了一个 **init** 方法，它不带参数。当把 **ServletConfig** 赋给 **servletConfig** 之后，这个方法就会被第一个 **init** 方法调用：

```
public void init(ServletConfig servletConfig)
    throws ServletException {
    this.servletConfig = servletConfig;
    this.init();
}
```

这意味着，可以通过覆盖无参的 init 方法来编写初始化代码，ServletConfig 仍然由 GenericServlet 实例保存。

代码清单 1-3 中的 GenericServletDemoServlet 类是对代码清单 1-2 中 ServletConfig-DemoServlet 的改写。注意，这个新的 Servlet 继承了 GenericServlet 类，而不是实现 Servlet 接口。

代码清单 1-3 GenericServletDemoServlet 类

---

```

package app01a;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.GenericServlet;
import javax.servlet.ServletConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.annotation.WebServlet;

@WebServlet(name = "GenericServletDemoServlet",
    urlPatterns = { "/generic" },
    initParams = {
        @WebInitParam(name="admin", value="Harry Taciak"),
        @WebInitParam(name="email", value="admin@example.com")
    }
)
public class GenericServletDemoServlet extends GenericServlet {

    private static final long serialVersionUID = 62500890L;

    @Override
    public void service(ServletRequest request,
        ServletResponse response)
        throws ServletException, IOException {
        ServletConfig servletConfig = getServletConfig();
        String admin = servletConfig.getInitParameter("admin");
        String email = servletConfig.getInitParameter("email");
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.print("<html><head></head><body>" +
                    "Admin:" + admin +
                    "<br/>Email:" + email +
                    "</body></html>");
    }
}

```

---

可以看出，通过继承 GenericServlet，就不需要覆盖你没计划要修改的那些方法了。这样，代码就会变得更加清晰。在代码清单 1-3 中，唯一被覆盖的方法是 service 方法，而且，

不必亲自保存 `ServletConfig`。

利用下面这个 URL 调用 Servlet，其结果应该与 `ServletConfigDemoServlet` 的一样。

```
http://localhost:8080/app01a/generic
```

虽然 `GenericServlet` 是 `Servlet` 的增强版本，但前者毕竟不常用，因为它不如 `HttpServlet` 高级。在现实的应用程序中，真正使用的还是 `HttpServlet`。详情查看下一节的内容。

## 1.9 HTTP Servlet

我们所编写的 Servlet 应用程序，尽管不能说全部，但其中大多数要用到 HTTP。这意味着，可以利用 HTTP 提供的特性。`javax.servlet.http` 包是 Servlet API 中的第二个包，其包含了编写 Servlet 应用程序的类和接口。`javax.servlet.http` 中的许多类型覆盖了 `javax.servlet` 中的类型。

图 1-5 展示了 `javax.servlet.http` 中的主要类型。

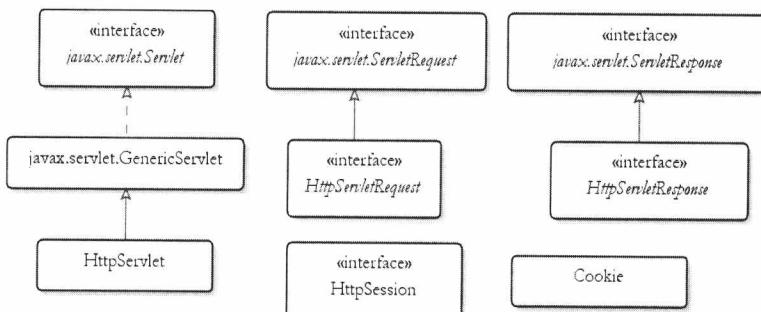


图 1-5 `javax.servlet.http` 中比较重要的成员

### 1.9.1 HttpServlet

`HttpServlet` 类覆盖 `javax.servlet.GenericServlet` 类。在使用 `HttpServlet` 时，还要使用 `HttpServletRequest` 和 `HttpServletResponse` 对象，它们分别表示 Servlet 请求和 Servlet 响应。`HttpServletRequest` 接口继承 `javax.servlet.ServletRequest`，`HttpServletResponse` 继承 `javax.servlet.ServletResponse`。

`HttpServlet` 覆盖 `GenericServlet` 中的 `service` 方法，并用以下签名添加了另一个 `service` 方法：

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

这个新的 `service` 方法与 `javax.servlet.Servlet` 中的区别在于，前者接受的是 `HttpServletRequest` 和 `HttpServletResponse`，而不是 `ServletRequest` 和 `ServletResponse`。

与往常一样，Servlet 容器调用 `javax.servlet.Servlet` 中原始的 `service` 方法，`HttpServletRequest` 中的 `service` 方法要如下这么写：

```
public void service(ServletRequest req, ServletResponse res)
    throws ServletException, IOException {
    HttpServletRequest request;
    HttpServletResponse response;
    try {
        request = (HttpServletRequest) req;
        response = (HttpServletResponse) res;
    } catch (ClassCastException e) {
        throw new ServletException("non-HTTP request or response");
    }
    service(request, response);
}
```

原始的 `service` 方法将请求和响应对象进行向下转换，分别从 Servlet 容器转换成 `HttpServletRequest` 和 `HttpServletResponse`，并调用新的 `service` 方法。向下转换总是会成功，因为在调用一个 Servlet 的 `service` 方法时，Servlet 容器总会预计使用 HTTP，所以传递一个 `HttpServletRequest` 和一个 `HttpServletResponse`。即使正在实现 `javax.servlet.Servlet` 接口或者继承 `javax.servlet.GenericServlet`，也可以将传给 `service` 方法的 Servlet 请求和 Servlet 响应，分别向下转换成 `HttpServletRequest` 和 `HttpServletResponse`。

之后，`HttpServletRequest` 中新的 `service` 方法会查看通常用来发送请求（通过调用 `request.getMethod()`）的 HTTP 方法，并调用以下某个方法（`doGet`、`doPost`、`doHead`、`doPut`、`doTrace`、`doOptions` 和 `doDelete`）。这 7 个方法各自表示一个 HTTP 方法。其中，`doGet` 和 `doPost` 是最常用的。因此，通常不需要再覆盖 `service` 方法，而是覆盖 `doGet` 或者 `doPost`，或者将 `doGet` 和 `doPost` 都覆盖掉。

总之，`HttpServletRequest` 中有两项特性是 `GenericServlet` 所没有的：

- 不覆盖 `service` 方法，而是覆盖 `doGet`、`doPost`，或者两者都覆盖掉。在极少数情况下，还要覆盖以下某个方法：`doHead`、`doPut`、`doTrace`、`doOptions` 或 `doDelete`。
- 将用 `HttpServletRequest` 和 `HttpServletResponse` 代替 `ServletRequest` 和 `ServletResponse`。

## 1.9.2 HttpServletRequest

`HttpServletRequest` 表示 HTTP 环境中的 Servlet 请求。它继承 `javax.servlet.ServletRequest` 接口，并增加了几个方法，例如：

```
java.lang.String getContextPath()
```

返回表示请求 context 的请求 URI 部分。

`Cookie[] getCookies()`

返回一个 `Cookie` 对象数组。

`java.lang.String getHeader(java.lang.String name)`

返回指定 HTTP 标头的值。

`java.lang.String getMethod()`

返回发出这条请求的 HTTP 方法的名称。

`java.lang.String getQueryString()`

返回请求 URL 中的查询字符串。

`HttpSession getSession()`

返回与这个请求有关的 session 对象。如果没有找到，则创建新的 session 对象。

`HttpSession getSession(boolean create)`

返回与这个请求有关的 session 对象。如果没有找到，并且 `create` 参数为 `true`，那么将创建新的 session 对象。

在接下来的章节中，将学习如何使用这些方法。

### 1.9.3 HttpServletResponse

`HttpServletResponse` 表示 HTTP 环境下的 Servlet 响应。下面是其中定义的部分方法：

`void addCookie(Cookie cookie)`

给这个响应对象添加 cookie。

`void addHeader(java.lang.String name, java.lang.String value)`

给这个响应对象添加标头。

`void sendRedirect(java.lang.String location)`

发送响应代号，将浏览器重定向到指定的位置。

在接下来的章节中将进一步学习这些方法。

## 1.10 处理 HTML 表单

每个 Web 应用程序中几乎都会包含一个或者多个 HTML 表单，用来接收用户输入。你可以轻松地将一个 HTML 表单从 Servlet 发送到浏览器。当用户提交表单时，在表单元素中

输入的值会被当作请求参数发送到服务器。

HTML 输入域（文本域、隐藏域或密码域）或者文本域的值被当作一个字符串发送到服务器。对于空白的输入域或者文本域将发送一条空白的字符串。因此，带有一个输入域名称的 `ServletRequest.getParameter` 将永远不会返回 null。

HTML 的 select 元素还会给标头发送一个字符串。如果没有选择 select 元素中的任何选项，那么将会发送所显示的选项值。

对于一个带有多个值的选择元素（即允许多个选项的 select 元素，用 `<select multiple>` 表示）是发送一个字符串数组，并且必须由 `ServletRequest.getParameterValues` 进行处理。

复选框比较特别一些。被选中的复选框将字符串“on”发送到服务器。没有被选中的复选框则不发送任何内容到服务器，并且 `ServletRequest.getParameter(fieldname)` 返回 null。

单选按钮将被选按钮的值发送到服务器。如果没有选中任何按钮，则不发送任何内容到服务器，并且 `ServletRequest.getParameter(fieldname)` 返回 null。

如果一个表单包含多个同名的输入元素，那么所有的值都会提交，必须用 `ServletRequest.getParameterValues` 来获取它们。`ServletRequest.getParameter` 将只返回最后一个值。

代码清单 1-4 中的 FormServlet 类示范了处理 HTML 表单的方法。它的 `doGet` 方法发送了一个订单表单给浏览器。它的 `doPost` 方法获取输入的值，并输出它们。这个 Servlet 是 app01b 应用程序的一部分。

代码清单 1-4 FormServlet 类

---

```

package app01b;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Enumeration;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "FormServlet", urlPatterns = { "/form" })
public class FormServlet extends HttpServlet {
    private static final long serialVersionUID = 54L;
    private static final String TITLE = "Order Form";

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.println("<html>");
        writer.println("<head>");
        writer.println("<title>" + TITLE + "</title></head>");
    }
}

```

```
writer.println("<body><h1>" + TITLE + "</h1>");  
writer.println("<form method='post'>");  
writer.println("<table>");  
writer.println("<tr>");  
writer.println("<td>Name:</td>");  
writer.println("<td><input name='name' /></td>");  
writer.println("</tr>");  
writer.println("<tr>");  
writer.println("<td>Address:</td>");  
writer.println("<td><textarea name='address' " +  
    + "cols='40' rows='5'></textarea></td>");  
writer.println("</tr>");  
writer.println("<tr>");  
writer.println("<td>Country:</td>");  
writer.println("<td><select name='country'>");  
writer.println("<option>United States</option>");  
writer.println("<option>Canada</option>");  
writer.println("</select></td>");  
writer.println("</tr>");  
writer.println("<tr>");  
writer.println("<td>Delivery Method:</td>");  
writer.println("<td><input type='radio' " +  
    + "name='deliveryMethod'" +  
    + " value='First Class'>First Class");  
writer.println("<input type='radio' " +  
    + "name='deliveryMethod'" +  
    + "value='Second Class'>Second Class</td>");  
writer.println("</tr>");  
writer.println("<tr>");  
writer.println("<td>Shipping Instructions:</td>");  
writer.println("<td><textarea name='instruction' " +  
    + "cols='40' rows='5'></textarea></td>");  
writer.println("</tr>");  
writer.println("<tr>");  
writer.println("<td>&nbsp;</td>");  
writer.println("<td><textarea name='instruction' " +  
    + "cols='40' rows='5'></textarea></td>");  
writer.println("</tr>");  
writer.println("<tr>");  
writer.println("<td>Please send me the latest " +  
    + "product catalog:</td>");  
writer.println("<td><input type='checkbox' " +  
    + "name='catalogRequest' /></td>");  
writer.println("</tr>");  
writer.println("<tr>");  
writer.println("<td>&nbsp;</td>");  
writer.println("<td><input type='reset' />" +  
    + "<input type='submit' /></td>");  
writer.println("</tr>");  
writer.println("</table>");
```

```
        writer.println("</form>");  
        writer.println("</body>");  
        writer.println("</html>");  
    }  
  
    @Override  
    public void doPost(HttpServletRequest request,  
                        HttpServletResponse response)  
            throws ServletException, IOException {  
        response.setContentType("text/html");  
        PrintWriter writer = response.getWriter();  
        writer.println("<html>");  
        writer.println("<head>");  
        writer.println("<title>" + TITLE + "</title></head>");  
        writer.println("</head>");  
        writer.println("<body><h1>" + TITLE + "</h1>");  
        writer.println("<table>");  
        writer.println("<tr>");  
        writer.println("<td>Name:</td>");  
        writer.println("<td>" + request.getParameter("name")  
                      + "</td>");  
        writer.println("</tr>");  
        writer.println("<tr>");  
        writer.println("<td>Address:</td>");  
        writer.println("<td>" + request.getParameter("address")  
                      + "</td>");  
        writer.println("</tr>");  
        writer.println("<tr>");  
        writer.println("<td>Country:</td>");  
        writer.println("<td>" + request.getParameter("country")  
                      + "</td>");  
        writer.println("</tr>");  
        writer.println("<tr>");  
        writer.println("<td>Shipping Instructions:</td>");  
        writer.println("<td>");  
        String[] instructions = request  
                .getParameterValues("instruction");  
        if (instructions != null) {  
            for (String instruction : instructions) {  
                writer.println(instruction + "<br/>");  
            }  
        }  
        writer.println("</td>");  
        writer.println("</tr>");  
        writer.println("<tr>");  
        writer.println("<td>Delivery Method:</td>");  
        writer.println("<td>"  
                      + request.getParameter("deliveryMethod")  
                      + "</td>");  
        writer.println("</tr>");  
        writer.println("<tr>");
```

```

writer.println("<td>Catalog Request:</td>");
writer.println("<td>");
if (request.getParameter("catalogRequest") == null) {
    writer.println("No");
} else {
    writer.println("Yes");
}
writer.println("</td>");
writer.println("</tr>");
writer.println("</table>");
writer.println("<div style='border:1px solid #ddd;" +
    "margin-top:40px;font-size:90%'>");

writer.println("Debug Info<br/>");
Enumeration<String> parameterNames = request
    .getParameterNames();
while (parameterNames.hasMoreElements()) {
    String paramName = parameterNames.nextElement();
    writer.println(paramName + ": ");
    String[] paramValues = request
        .getParameterValues(paramName);
    for (String paramValue : paramValues) {
        writer.println(paramValue + "<br/>");
    }
}
writer.println("</div>");
writer.println("</body>");
writer.println("</html>");
}
}

```

---

可以利用下面这个 URL 调用 FormServlet:

<http://localhost:8080/app01b/form>

被调用的 DoGet 方法将这个 HTML 表单发送给浏览器。

```

<form method='post'>
<input name='name' />
<textarea name='address' cols='40' rows='5'></textarea>
<select name='country'>;
    <option>United States</option>
    <option>Canada</option>
</select>
<input type='radio' name='deliveryMethod' value='First Class' />
<input type='radio' name='deliveryMethod' value='Second Class' />
<textarea name='instruction' cols='40' rows='5'></textarea>
<textarea name='instruction' cols='40' rows='5'></textarea>
<input type='checkbox' name='catalogRequest' />
<input type='reset' />
<input type='submit' />
</form>

```

表单的方法被设置为 post，确保当用户提交表单时，采用的是 HTTP POST 方法。它的 action 属性缺失，表示表单会提交给发出请求的那个 URL。

图 1-6 展示了一个空白的订单表单。

The screenshot shows a web browser window titled "Order Form". The address bar displays the URL "http://localhost:8080/app01b/form". The main content area is titled "Order Form". It contains several input fields and controls:

- Name: A text input field.
- Address: A large text input field.
- Country: A dropdown menu set to "United States".
- Delivery Method: Two radio buttons labeled "First Class" and "Second Class".
- Shipping Instructions: Two large text input fields stacked vertically.
- Please send me the latest product catalog: A checkbox.
- Buttons: "Reset" and "Submit".

图 1-6 空白的订单表单

现在，填写表单，并单击 Submit（提交）按钮。在表单中输入的值将会通过 HTTP POST 方法发送到服务器，这样就会调用 Servlet 的 doPost 方法。因此，将会看到如图 1-7 所示的值。

The screenshot shows the same "Order Form" window after inputting values. The fields now contain the following data:

Name:	Ted Mosby
Address:	123 XYZ Street Markham ON L1L L3L
Country:	Canada
Shipping Instructions:	Please leave at door Don't disturb the dog
Delivery Method:	First Class
Catalog Request:	Yes

Below the form, under "Debug Info", the submitted parameters are listed:

```
instruction: Please leave at door
Don't disturb the dog
address: 123 XYZ Street Markham ON L1L L3L
deliveryMethod: First Class
name: Ted Mosby
catalogRequest: on
country: Canada
```

图 1-7 在订单表单中输入的值

## 1.11 使用部署描述符

在前面的范例中已经学过，编写和部署 Servlet 应用程序是很容易的。部署的一个方面是给 Servlet 映射配置一条路径。在那些例子中，利用 `WebServlet` 注解类型给一个 Servlet 映射了一条路径。

使用部署描述符是配置 Servlet 应用程序的另一种方法，关于部署描述符的详细讨论，可查看第 16 章的内容。部署描述符总是命名为 `web.xml`，并放在 `WEB-INF` 目录下。本章将介绍如何创建一个名为 `app01c` 的 Servlet 应用程序，并为它编写一个 `web.xml` 文件。

`app01c` 有两个 Servlet：`SimpleServlet` 和 `WelcomeServlet`，以及一个映射 Servlet 的部署描述符。代码清单 1-5 和代码清单 1-6 分别展示了 `SimpleServlet` 和 `WelcomeServlet`。注意，Servlet 类没有用 `@WebServlet` 进行标注。部署描述符如代码清单 1-7 所示。

代码清单 1-5 未标注的 `SimpleServlet` 类

---

```
package app01c;
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 SimpleServlet extends HttpServlet {
    private static final long serialVersionUID = 8946L;

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.print("<html><head></head>" +
                    "<body>Simple Servlet</body></html>");
    }
}
```

---

代码清单 1-6 未标注的 `WelcomeServlet` 类

---

```
package app01c;
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 WelcomeServlet extends HttpServlet {
```

---

```

private static final long serialVersionUID = 27126L;

@Override
public void doGet(HttpServletRequest request,
                   HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.print("<html><head></head>" +
                "<body>Welcome</body></html>");
}
}

```

---

代码清单 1-7 部署描述符

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
          version="3.0">

    <servlet>
        <servlet-name>SimpleServlet</servlet-name>
        <servlet-class>app01c.SimpleServlet</servlet-class>
        <load-on-startup>10</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>SimpleServlet</servlet-name>
        <url-pattern>/simple</url-pattern>
    </servlet-mapping>

    <servlet>
        <servlet-name>WelcomeServlet</servlet-name>
        <servlet-class>app01c>WelcomeServlet</servlet-class>
        <load-on-startup>20</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>WelcomeServlet</servlet-name>
        <url-pattern>/welcome</url-pattern>
    </servlet-mapping>
</web-app>

```

---

使用部署描述符有许多好处。第一，可以包含 @WebServlet 中没有的元素，如 load-

`on-startup` 元素。这个元素在应用程序启动时加载 Servlet，而不是第一次调用 Servlet 时加载。使用 `load-on-startup` 意味着，Servlet 的第一次调用并不比后续的调用更占用时间。如果 Servlet 的 `init` 方法比较费时的话，这个元素就特别有帮助。

使用部署描述符的另一个好处是，如果需要修改配置值，如 Servlet 路径，就不需要重新编译 Servlet 类。

此外，可以将初始参数传给一个 Servlet，并且不需要重新编译 Servlet 类就可以对它们进行编辑。

部署描述符还允许覆盖 Servlet 注解中指定的值。Servlet 中的 `WebServlet` 注解，如果没有在部署描述符中进行声明，那么它将是无效的。但是，一个带有部署描述符的应用程序，如果对于不在其部署描述符中的 Servlet 进行标注，则其仍然有效。这意味着，可以在标注完 Servlet 之后，又在同一个应用程序的部署描述符中声明 Servlet。

图 1-8 展示了 `app01c` 的目录结构。这个目录结构与 `app01a` 的并没有太大的区别。唯一的不同在于，`app01c` 在 `WEB-INF` 中有一个 `web.xml` 文件。

在部署描述符中声明了 `SimpleServlet` 和 `WelcomeServlet` 之后，就可以利用下面这些 URL 来访问它们了：

```
http://localhost:8080/app01c/simple
http://localhost:8080/app01c/welcome
```

关于部署和部署描述符的更多信息，可查看第 16 章的内容。

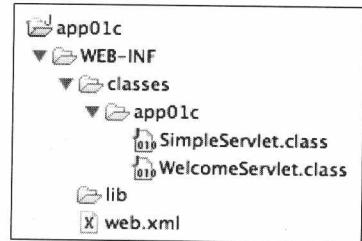


图 1-8 带有部署描述符 `app01c` 目录结构

## 1.12 小结

Servlet 技术是 Java EE 技术的组成部分。Servlet 容器中运行的所有 Servlet，以及容器与 Servlet 之间的契约，都采用了 `javax.servlet.Servlet` 接口的形式。`javax.servlet` 包也提供了实现 `Servlet` 接口的 `GenericServlet` 抽象类。这是一个便利类，可以通过扩展它来创建 Servlet。但是，大多数现代的 Servlet 都在 HTTP 环境中处理请求。因此，将 `javax.servlet.http.HttpServlet` 类子类化会更有意义。`HttpServlet` 类本身也是 `GenericServlet` 的一个子类。

## 第 2 章 Session 管理

Session 管理（或 Session 追踪）是 Web 应用程序开发中一个非常重要的主题。这是因为 Web 语言 HTTP 是无状态的。在默认情况下，Web 服务器不知道一个 HTTP 请求是来自初次用户，还是来自之前已经访问过的用户。

例如，webmail 应用程序要求其用户在查看邮件之前要先登录。但是，一旦用户输入正确的用户名和密码，用户在访问应用程序的其他部分时，就不应该再次提示他们登录。应用程序需要记住哪些用户已经登录成功。换句话说，它必须能够管理用户 Session。

本章讲解可以用于保持状态的 4 种方法：网址重写（URL rewriting）、隐藏域、cookie 及 HttpSession 对象。本章展示的范例都是 app02a 应用程序中的内容。

### 2.1 网址重写

网址重写是一种 Session 追踪技术，需要将一个或多个 token 作为一个查询字符串添加到一个 URL 中。token 的格式一般是键 = 值：

*url?key-1=value-1&key-2=value-2 ... &key-n=value-n*

注意，URL 和 token 之间要用一个问号（?）隔开，两个 token 之间则是用一个 & 符号隔开。

如果 token 不必在过多的 URL 中四处携带，那么网址重写就比较合适。采用网址重写的缺点如下：

- 在有些 Web 浏览器中，URL 限制为 2000 个字符。
- 仅当有链接要插入值时，值才能转换成后面的资源。此外，要把值添加到静态页面的链接中，可不是一件容易的事情。
- 网址重写必须在服务器端有效。所有的链接都必须带有值，这样可能出现一个问题，即一个页面中可能会有许多个链接。
- 某些字符，例如空格、& 符号及问号都必须进行编码。
- 添加到 URL 中的信息是明显可见的，这种情况有时可不是我们所期待的。

由于上述局限性，网址重写只适用于那些既需要保持，却又不跨越太多页面，并且又不太重要的信息。

举个例子，代码清单 2-1 中的 Top10Servlet 类是一个 Servlet，它展示了伦敦和巴黎这两座城市中十大最受人喜爱的旅游胜地。信息分两页显示。第一页显示所选城市中前五大最

受人喜爱的旅游胜地，第二个显示接下来的五个。Servlet 利用网址重写追踪被选城市和页码。它继承 HttpServlet，并利用 /top10 URL 模式调用。

#### 代码清单 2-1 Top10Servlet 类

```
package app02a.urlrewriting;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "Top10Servlet", urlPatterns = { "/top10" })
public class Top10Servlet extends HttpServlet {
    private static final long serialVersionUID = 987654321L;

    private List<String> londonAttractions;
    private List<String> parisAttractions;
    @Override
    public void init() throws ServletException {
        londonAttractions = new ArrayList<String>(10);
        londonAttractions.add("Buckingham Palace");
        londonAttractions.add("London Eye");
        londonAttractions.add("British Museum");
        londonAttractions.add("National Gallery");
        londonAttractions.add("Big Ben");
        londonAttractions.add("Tower of London");
        londonAttractions.add("Natural History Museum");
        londonAttractions.add("Canary Wharf");
        londonAttractions.add("2012 Olympic Park");
        londonAttractions.add("St Paul's Cathedral");

        parisAttractions = new ArrayList<String>(10);
        parisAttractions.add("Eiffel Tower");
        parisAttractions.add("Notre Dame");
        parisAttractions.add("The Louvre");
        parisAttractions.add("Champs Elysees");
        parisAttractions.add("Arc de Triomphe");
        parisAttractions.add("Sainte Chapelle Church");
        parisAttractions.add("Les Invalides");
        parisAttractions.add("Musee d'Orsay");
        parisAttractions.add("Montmartre");
        parisAttractions.add("Sacre Couer Basilica");
    }

    @Override
    public void doGet(HttpServletRequest request,
```

```

HttpServletResponse response) throws ServletException,
IOException {

    String city = request.getParameter("city");
    if (city != null &
        (city.equals("london") || city.equals("paris"))) {
        // show attractions
        showAttractions(request, response, city);
    } else {
        // show main page
        showMainPage(request, response);
    }
}

private void showMainPage(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
IOException {
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.print("<html><head>" +
                "<title>Top 10 Tourist Attractions</title>" +
                "</head><body>" +
                "Please select a city:" +
                "<br/><a href='?city=london'>London</a>" +
                "<br/><a href='?city=paris'>Paris</a>" +
                "</body></html>");
}

private void showAttractions(HttpServletRequest request,
    HttpServletResponse response, String city)
throws ServletException, IOException {

    int page = 1;
    String pageParameter = request.getParameter("page");
    if (pageParameter != null) {
        try {
            page = Integer.parseInt(pageParameter);
        } catch (NumberFormatException e) {
            // do nothing and retain default value for page
        }
        if (page > 2) {
            page = 1;
        }
    }
    List<String> attractions = null;
    if (city.equals("london")) {
        attractions = londonAttractions;
    } else if (city.equals("paris")) {
        attractions = parisAttractions;
    }
    response.setContentType("text/html");
}

```

```

PrintWriter writer = response.getWriter();
writer.println("<html><head>" +
    "<title>Top 10 Tourist Attractions</title>" +
    "</head><body>");
writer.println("<a href='top10'>Select City</a> ");
writer.println("<hr/>Page " + page + "<hr/>");

int start = page * 5 - 5;
for (int i = start; i < start + 5; i++) {
    writer.println(attractions.get(i) + "<br/>");
}
writer.print("<hr style='color:blue' />" +
    "<a href='?city=" + city +
    "&page=1'>Page 1</a>");
writer.println("&nbsp; <a href='?city=" + city +
    "&page=2'>Page 2</a>");
writer.println("</body></html>");
}
}

```

---

当第一个用户请求 Servlet 时，调用 `init` 方法，并填充两个类级 `List`：`londonAttractions` 和 `parisAttractions`，它们每一个都包含十个旅游胜地。

每一次请求都会调用 `doGet` 方法，查看 URL 是否包含请求参数 `city`，以及它的值是否为“`london`”或者“`paris`”。方法会根据这个参数的值调用 `showAttractions` 或者 `show MainPage`。

```

String city = request.getParameter("city");
if (city != null &&
    (city.equals("london") || city.equals("paris"))) {
    // show attractions
    showAttractions(request, response, city);
} else {
    // show main page
    showMainPage(request, response);
}

```

最初，用户会不带请求参数调用 Servlet，并且还会调用 `show MainPage`。这是一个简单的方法，它给浏览器发送了两个超链接，每个链接中都嵌有一个 token：`city=cityName`。用户会看到如图 2-1 所示的屏幕。现在，用户可以选择其中一个城市了。

如果你打开页面源代码，就会看到主体标签中有以下 HTML 标签：

```

Please select a city:<br/>
<a href='?city=london'>London</a><br/>
<a href='?city=paris'>Paris</a>

```

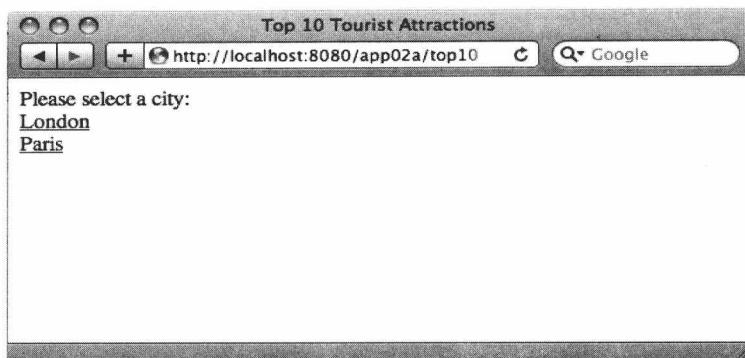


图 2-1 Top10Servlet 的初始页

最值得关注的是 `a` 标签的 `href` 属性值，它包含一个问号，接着是 `token: city=london` 或者 `city=paris`。任何相对的 URL（没有协议部分的 URL）都会被当作是相对于当前页面的 URL。换句话说，如果单击了页面中的某一个超链接，那么：

```
http://localhost:8080/app02a/top10?city=london
```

或者

```
http://localhost:8080/app02a/top10?city=paris
```

就会被发送到服务器。

一旦用户单击其中一个超链接，`doGet` 方法就会找到请求参数 `city`，并将控制权传给 `showAttractions` 方法，然后查看 URL，看它里面是否包含请求参数 `page`。如果没有请求参数 `page`，或者它的值不能转换成数字，就会假设这个值为 1，该方法就会发送被选中城市的前五个旅游胜地。图 2-2 展示了选中伦敦时的情况。

除了城市中的旅游胜地之外，`showAttractions` 还会发送三个超链接：`Select City`、`Page 1` 及 `Page 2`。`Select City` 会调用没有请求参数的 Servlet。`Page 1` 和 `Page 2` 包含两个 `token: city` 和 `page`。

```
http://localhost:8080/app02a/top10?city=cityName&page=pageNumber
```

如果选择了伦敦，并单击 `Page 2`，它就会将这个 URL 发送到服务器，并在后面添加两个键 / 值 `token`：

```
http://localhost:8080/app02a/top10?city=london&page=2
```

这时就会看到伦敦前十大旅游胜地的后五个，如图 2-3 所示。

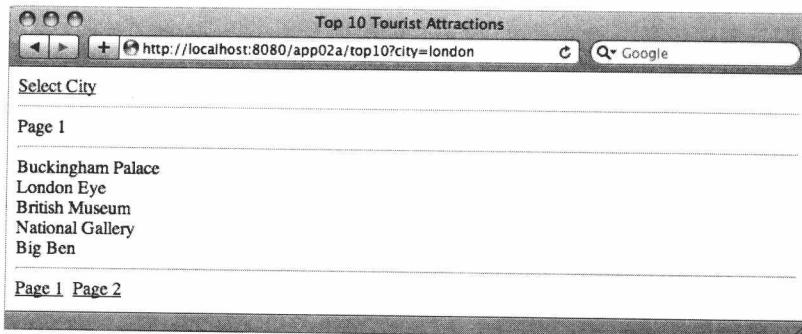


图 2-2 伦敦前十大旅游胜地中的第一页

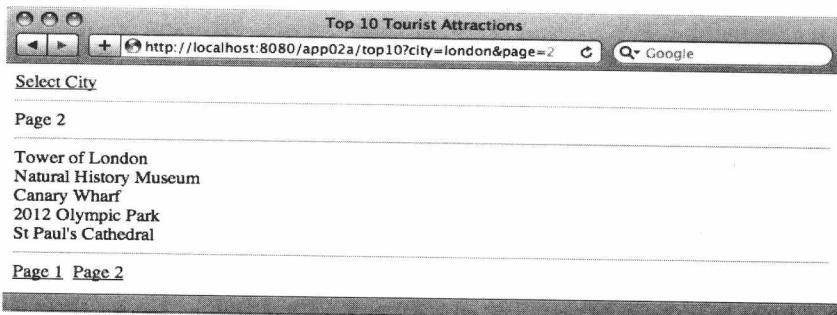


图 2-3 伦敦十大旅游胜地的第二页

这个范例展示了如何利用网址重写来嵌入一个城市，以便服务器知道要在第二页中显示什么内容。

## 2.2 隐藏域

利用隐藏域来保持状态，与采用网址重写技术类似。但它不是将值添加到 URL 后面，而是将它们放在 HTML 表单的隐藏域中。当用户提交表单时，隐藏域中的值也传送到服务器。只有当页面包含表单，或者可以在页面中添加表单时，才适合使用隐藏域。这种技术胜过网址重写技术的地方在于，可以将更多的字符传到服务器，并且不需要进行字符编码。但是像网址重写一样，也只有当要传递的信息不需要跨越多个页面时，才适合使用这种技术。

**Customer** 类构建了一个客户模型，如代码清单 2-2 所示。代码清单 2-3 中的 Servlet 展示了如何利用隐藏域来更新客户信息。

### 代码清单 2-2 Customer 类

---

```
package app02a.hiddenfields;
public class Customer {
```

```

private int id;
private String name;
private String city;

public int getId() {
    return id;
}
public void setId(int id) {
    this.id = id;
}
public String getName() {
    return name;
}
public void setName(String name) {
    this.name = name;
}
public String getCity() {
    return city;
}
public void setCity(String city) {
    this.city = city;
}
}

```

---

代码清单 2-3 CustomerServlet 类

```

package app02a.hiddenfields;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/*
 * Not thread-safe. For illustration purpose only
 */
@WebServlet(name = "CustomerServlet", urlPatterns = {
    "/customer", "/editCustomer", "/updateCustomer"})
public class CustomerServlet extends HttpServlet {
    private static final long serialVersionUID = -20L;

    private List<Customer> customers = new ArrayList<Customer>();

    @Override
    public void init() throws ServletException {
        Customer customer1 = new Customer();
        customer1.setId(1);
    }
}

```

```
customer1.setName("Donald D.");
customer1.setCity("Miami");
customers.add(customer1);

Customer customer2 = new Customer();
customer2.setId(2);
customer2.setName("Mickey M.");
customer2.setCity("Orlando");
customers.add(customer2);
}

private void sendCustomerList(HttpServletRequest response)
throws IOException {
response.setContentType("text/html");
PrintWriter writer = response.getWriter();
writer.println("<html><head><title>Customers</title></head>" +
+ "<body><h2>Customers </h2>");
writer.println("<ul>");
for (Customer customer : customers) {
writer.println("<li>" + customer.getName() +
+ "(" + customer.getCity() + ") (" +
+ "<a href='editCustomer?id=" + customer.getId() +
+ "'>edit</a>)");

}
writer.println("</ul>");
writer.println("</body></html>");

}

private Customer getCustomer(int customerId) {
for (Customer customer : customers) {
if (customer.getId() == customerId) {
return customer;
}
}
return null;
}

private void sendEditCustomerForm(HttpServletRequest request,
HttpServletResponse response) throws IOException {
response.setContentType("text/html");
PrintWriter writer = response.getWriter();
int customerId = 0;
try {
customerId =
Integer.parseInt(request.getParameter("id"));
} catch (NumberFormatException e) {
}
Customer customer = getCustomer(customerId);
```

```

        if (customer != null) {
            writer.println("<html><head>" +
                + "<title>Edit Customer</title></head>" +
                + "<body><h2>Edit Customer</h2>" +
                + "<form method='post' " +
                + "action='updateCustomer'>");
            writer.println("<input type='hidden' name='id' value='"
                + customerId + "'/>");
            writer.println("<table>");
            writer.println("<tr><td>Name:</td><td>" +
                + "<input name='name' value='"
                + customer.getName().replaceAll("'", "\")" +
                + "'/></td></tr>");
            writer.println("<tr><td>City:</td><td>" +
                + "<input name='city' value='"
                + customer.getCity().replaceAll("'", "\")" +
                + "'/></td></tr>");
            writer.println("<tr>" +
                + "<td colspan='2' style='text-align:right'>" +
                + "<input type='submit' value='Update' /></td>" +
                + "</tr>");
            writer.println("<tr><td colspan='2'>" +
                + "<a href='customer'>Customer List</a>" +
                + "</td></tr>");
            writer.println("</table>");
            writer.println("</form></body>");
        } else {
            writer.println("No customer found");
        }
    }

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String uri = request.getRequestURI();
        if (uri.endsWith("/customer")) {
            sendCustomerList(response);
        } else if (uri.endsWith("/editCustomer")) {
            sendEditCustomerForm(request, response);
        }
    }

    @Override
    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        // update customer
        int customerId = 0;
        try {
            customerId =

```

```

        Integer.parseInt(request.getParameter("id"));
    } catch (NumberFormatException e) {
    }
    Customer customer = getCustomer(customerId);
    if (customer != null) {
        customer.setName(request.getParameter("name"));
        customer.setCity(request.getParameter("city"));
    }
    sendCustomerList(response);
}
}

```

CustomerServlet 类继承 HttpServlet，并映射到三个 URL 模式：/customer、/editCustomer 及 /updateCustomer。前两个模式会调用 Servlet 的 doGet 方法，/updateCustomer 调用 doPost 方法。

/customer 是这个小应用程序的入口点。它列出了 init 方法填入的类级 customers List 的客户。(在现实的应用程序中，可能是从数据库中获得客户信息)。详情见图 2-4。

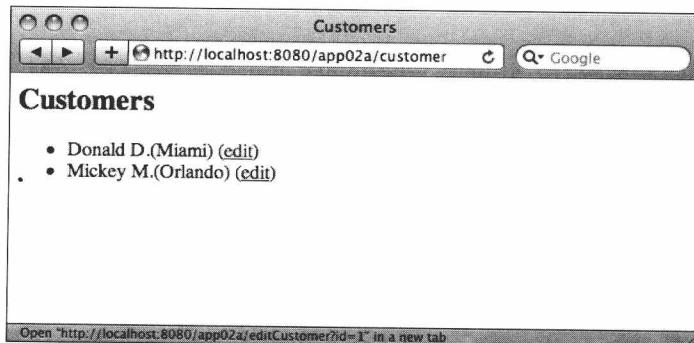


图 2-4 客户列表

如图 2-4 所示，每个客户都有一个 edit 链接。/editCustomer?id=customerId 指向每个链接的 href 属性。一旦收到 /editCustomer，Servlet 就会发送一个如图 2-5 所示的客户编辑表单。

图 2-5 Edit Customer 表单

如果单击第一个客户，Servlet 就会发送这个包含了隐藏域的 form 标签：

```
<form method='post' action='updateCustomer'>
<input type='hidden' name='id' value='1' />
<table>
    <tr><td>Name:</td>
        <td><input name='name' value='Donald DC.' /></td>
    </tr>
    <tr>
        <td>City:</td><td><input name='city' value='Miami' /></td>
    </tr>
    <tr>
        <td colspan='2' style='text-align:right'>
            <input type='submit' value='Update' />
        </td>
    </tr>
    <tr>
        <td colspan='2'><a href='customer'>Customer List</a></td>
    </tr>
</table>
</form>
```

注意表单中的隐藏域没有？它包含了 customer id，因此当提交表单时，服务器就会知道正在编辑哪个客户的信息。

值得一提的是，这个表单使用了 post 方法，因此当提交它时，浏览器就会采用 HTTP POST 方法，并调用 Servlet 的 doPost 方法。

## 2.3 cookie

网址重写和隐藏域都只适用于保持那些不需要跨越许多页面的信息。如果这些信息需要跨越很多页面，这两种技术就变得很难实现，因为你必须管理每一个页面的信息。值得庆幸的是，cookie 能够解决网址重写和隐藏域都无法解决的问题。

cookie 是自动地在 Web 服务器和浏览器之间来回传递的一小块信息。cookie 适用于那些需要跨越许多页面的信息。由于 cookie 是作为 HTTP 标头嵌入的，因此传输它的过程由 HTTP 协议处理。除此之外，还可以根据自己的需要设置 cookie 的有效期限。对于 Web 浏览器而言，每台 Web 服务器最多可以支持 20 个 cookie。

cookie 的不足之处在于，用户可以通过修改他 / 她的浏览器设置来拒绝接受 cookie。

要使用 cookie，必须熟悉 javax.servlet.http.Cookie 类，以及 HttpServletRequest 和 HttpServletResponse 接口中的几个方法。

要创建 cookie，传递一个名称和一个值给 Cookie 类的构造器：

```
Cookie cookie = new Cookie(name, value);
```

例如，要创建一个选择语言的 cookie，可以这么写：

```
Cookie languageSelectionCookie = new Cookie("language", "Italian");
```

创建 **Cookie** 之后，可以设置它的 **domain**、**path** 及 **maxAge** 属性。尤其值得关注的是 **maxAge** 属性，因为它决定 cookie 的有效期限。

为了将一个 cookie 发送到浏览器，需在 **HttpServletResponse** 上调用 **add** 方法：

```
httpServletResponse.addCookie(cookie);
```

当浏览器再次发出对同一个资源或者对同一台服务器中的不同资源的请求时，它会同时把从 Web 浏览器处收到的 cookie 再传回去。

cookie 也可以利用 JavaScript 在客户端进行创建和删除，但是这超出了本书的内容范围。

要访问浏览器发出的 cookie，可以在 **HttpServletRequest** 中使用 **getCookies** 方法。该方法将返回一个 **Cookie** 数组，如果请求中没有 cookie，将返回 null。为了找到某个名称的 cookie，需要迭代数组。下面举个例子，看看如何读取一个名为 **maxRecords** 的 cookie。

```
Cookie[] cookies = request.getCookies();
Cookie maxRecordsCookie = null;
if (cookies != null) {
    for (Cookie cookie : cookies) {
        if (cookie.getName().equals("maxRecords")) {
            maxRecordsCookie = cookie;
            break;
        }
    }
}
```

令人遗憾的是，没有 **getCookieByName** 方法可以使获取 cookie 变得更简单一些。更令人难过的是，也没有方法可以直接删除 cookie。为了删除 cookie，需要创建一个同名的 cookie，将它的 **maxAge** 属性设置为 0，并在 **HttpServletResponse** 中添加一个新的 cookie。看看下面是如何删除一个名为 **userName** 的 cookie 的：

```
Cookie cookie = new Cookie("userName", "");
cookie.setMaxAge(0);
response.addCookie(cookie);
```

为了举例说明如何在 Session 管理中使用 cookie，可参见代码清单 2-4 中的 **PreferenceServlet** 类。这个 Servlet 允许用户修改 4 个 cookie 的值，从而在同一个应用程序中设定其他 Servlet 的显示设置。

#### 代码清单 2-4 PreferenceServlet 类

---

```
package app02a.cookie;
import java.io.IOException;
```

```
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "PreferenceServlet",
            urlPatterns = { "/preference" })
public class PreferenceServlet extends HttpServlet {
    private static final long serialVersionUID = 888L;

    public static final String MENU =
        "<div style='background:#e8e8e8;'"
        + "padding:15px'>"
        + "<a href='cookieClass'>Cookie Class</a>&nbsp;&nbsp;"
        + "<a href='cookieInfo'>Cookie Info</a>&nbsp;&nbsp;"
        + "<a href='preference'>Preference</a>" + "</div>";

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) throws ServletException,
                      IOException {
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();
        writer.print("<html><head>" + "<title>Preference</title>" +
                    + "<style>table {" + "font-size:small;" +
                    + "background:NavajoWhite }</style>" +
                    + "</head><body>" +
                    + MENU +
                    + "Please select the values below:" +
                    + "<form method='post'>" +
                    + "<table>" +
                    + "<tr><td>Title Font Size: </td>" +
                    + "<td><select name='titleFontSize'>" +
                    + "<option>large</option>" +
                    + "<option>x-large</option>" +
                    + "<option>xx-large</option>" +
                    + "</select></td>" +
                    + "</tr>" +
                    + "<tr><td>Title Style & Weight: </td>" +
                    + "<td><select name='titleStyleAndWeight' multiple>" +
                    + "<option>italic</option>" +
                    + "<option>bold</option>" +
                    + "</select></td>" +
                    + "</tr>" +
                    + "<tr><td>Max. Records in Table: </td>" +
                    + "<td><select name='maxRecords'>" +
                    + "<option>5</option>" +
                    + "<option>10</option>"
```

```

        + "</select></td>" +
        + "</tr>" +
        + "<tr><td rowspan='2'>" +
        + "<input type='submit' value='Set' /></td>" +
        + "</tr>" +
        + "</table>" + "</form>" + "</body></html>") ;
    }

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

    String maxRecords = request.getParameter("maxRecords");
    String[] titleStyleAndWeight = request
        .getParameterValues("titleStyleAndWeight");
    String titleFontSize =
        request.getParameter("titleFontSize");
    response.addCookie(new Cookie("maxRecords", maxRecords));
    response.addCookie(new Cookie("titleFontSize",
        titleFontSize));

    // delete titleFontWeight and titleFontStyle cookies first
    // Delete cookie by adding a cookie with the maxAge = 0;
    Cookie cookie = new Cookie("titleFontWeight", "");
    cookie.setMaxAge(0);
    response.addCookie(cookie);

    cookie = new Cookie("titleFontStyle", "");
    cookie.setMaxAge(0);
    response.addCookie(cookie);

    if (titleStyleAndWeight != null) {
        for (String style : titleStyleAndWeight) {
            if (style.equals("bold")) {
                response.addCookie(new
                    Cookie("titleFontWeight", "bold"));
            } else if (style.equals("italic")) {
                response.addCookie(new Cookie("titleFontStyle",
                    "italic"));
            }
        }
    }

    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.println("<html><head>" + "<title>Preference</title>" +
        + "</head><body>" + MENU
        + "Your preference has been set."
        + "<br/><br/>Max. Records in Table: " + maxRecords
}

```

```

        + "<br/>Title Font Size: " + titleFontSize
        + "<br/>Title Font Style & Weight: ");

    // titleStyleAndWeight will be null if none of the options
    // was selected
    if (titleStyleAndWeight != null) {
        writer.println("<ul>");
        for (String style : titleStyleAndWeight) {
            writer.print("<li>" + style + "</li>");
        }
        writer.println("</ul>");
    }
    writer.println("</body></html>");
}
}

```

PreferenceServlet 的 doGet 方法发送了一个表单，其中包含几个输入域，如图 2-6 所示。

表单的上方有三个链接（Cookie Class、Cookie Info 及 Preference），可以导航到同一个应用程序的其他 Servlet。稍后会简单介绍一下 Cookie Class 和 Cookie Info。

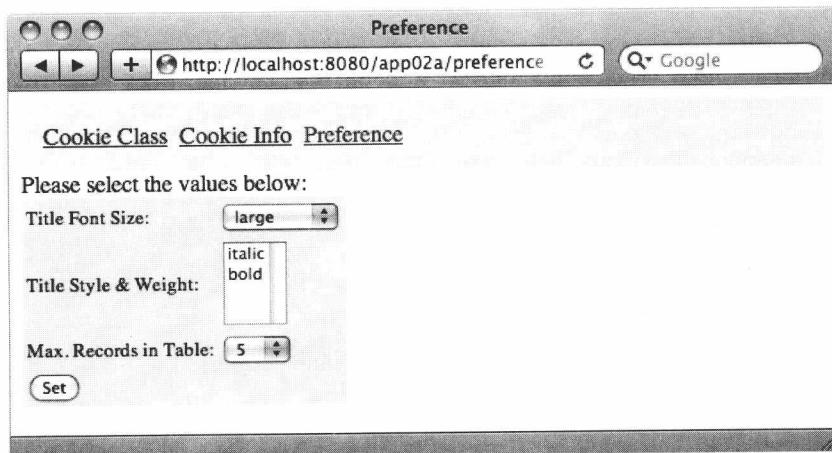


图 2-6 用 cookie 管理用户偏好

当用户提交表单时，会调用 PreferenceServlet 的 doPost 方法。doPost 方法会创建这些 cookie：maxRecords、titleFontSize、titleFontStyle 及 titleFontWeight，并将同一个 cookie 的所有之前值都覆盖掉。之后，它会将用户选择的值发送给浏览器。

可以利用下面这个 URL 来调用 PreferenceServlet：

`http://localhost:8080/app02a/preference`

代码清单 2-5 中的 CookieClassServlet 类和代码清单 2-6 中的 CookieInfoServlet 类，

是利用 **PreferenceServlet** 设置的 cookie 来格式化它们的内容的。CookieClassServlet Servlet 将 Cookie 类的属性写在一个 HTML 列表中。列表中的项编号由 **maxRecords** cookie 的值决定，而这个值则可以由用户利用 **PreferenceServlet** 进行设置。

代码清单 2-5 CookieClassServlet 类

---

```

package app02a.cookie;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "CookieClassServlet",
           urlPatterns = { "/cookieClass" })
public class CookieClassServlet extends HttpServlet {
    private static final long serialVersionUID = 837369L;

    private String[] methods = {
        "clone", "getComment", "getDomain",
        "getMaxAge", "getName", "getPath",
        "getSecure", "getValue", "getVersion",
        "isHttpOnly", "setComment", "setDomain",
        "setHttpOnly", "setMaxAge", "setPath",
        "setSecure", "setValue", "setVersion"
    };

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

        Cookie[] cookies = request.getCookies();
        Cookie maxRecordsCookie = null;
        if (cookies != null) {
            for (Cookie cookie : cookies) {
                if (cookie.getName().equals("maxRecords")) {
                    maxRecordsCookie = cookie;
                    break;
                }
            }
        }

        int maxRecords = 5; // default
        if (maxRecordsCookie != null) {
            try {
                maxRecords = Integer.parseInt(

```

```

        maxRecordsCookie.getValue());
    } catch (NumberFormatException e) {
        // do nothing, use maxRecords default value
    }
}

response.setContentType("text/html");
PrintWriter writer = response.getWriter();
writer.print("<html><head>" + "<title>Cookie Class</title>" +
    "</head><body>" +
    PreferenceServlet.MENU
    + "<div>Here are some of the methods in " +
    "javax.servlet.http.Cookie");
writer.print("<ul>");
for (int i = 0; i < maxRecords; i++) {
    writer.print("<li>" + methods[i] + "</li>");
}
writer.print("</ul>");
writer.print("</div></body></html>");
}
}

```

---

**CookieInfoServlet** 类读取 titleFontSize、titleFontWeight 和 titleFontStyle cookie 的值，将以下 CSS 样式写到浏览器中，这里的 x、y 和 z 是指上述 cookie 的值。

```

.title {
    font-size: x;
    font-weight: y;
    font-style: z;
}

```

一个 div 元素用这个样式来格式化文本 “Session Management with Cookies.”。

代码清单 2-6 CookieInfoServlet 类

---

```

package app02a.cookie;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "CookieInfoServlet", urlPatterns =
    { "/cookieInfo" })
public class CookieInfoServlet extends HttpServlet {
    private static final long serialVersionUID = 3829L;

```

```

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

    Cookie[] cookies = request.getCookies();
    StringBuilder styles = new StringBuilder();
    styles.append(".title {");
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            String name = cookie.getName();
            String value = cookie.getValue();
            if (name.equals("titleFontSize")) {
                styles.append("font-size:" + value + ";");
            } else if (name.equals("titleFontWeight")) {
                styles.append("font-weight:" + value + ";");
            } else if (name.equals("titleFontStyle")) {
                styles.append("font-style:" + value + ";");
            }
        }
    }
    styles.append("}");

    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.print("<html><head>" + "<title>Cookie Info</title>" +
        "<style>" + styles.toString() + "</style>" +
        "</head><body>" + PreferenceServlet.MENU +
        "<div class='title'>" +
        "Session Management with Cookies:</div>");
    writer.print("<div>");

    // cookies will be null if there's no cookie
    if (cookies == null) {
        writer.print("No cookie in this HTTP response.");
    } else {
        writer.println("<br/>Cookies in this HTTP response:");
        for (Cookie cookie : cookies) {
            writer.println("<br/>" + cookie.getName() + ":" +
                cookie.getValue());
        }
    }
    writer.print("</div>");
    writer.print("</body></html>");
}
}

```

可以利用下面这个 URL 调用 CookieClassServlet:

<http://localhost:8080/app02a/cookieClass>

可以通过将浏览器跳转到下面这个 URL，来调用 CookieInfoServlet servlet：

`http://localhost:8080/app02a/cookieInfo`

图 2-7 和图 2-8 分别展示了 CookieClassServlet 和 CookieInfoServlet 的结果。

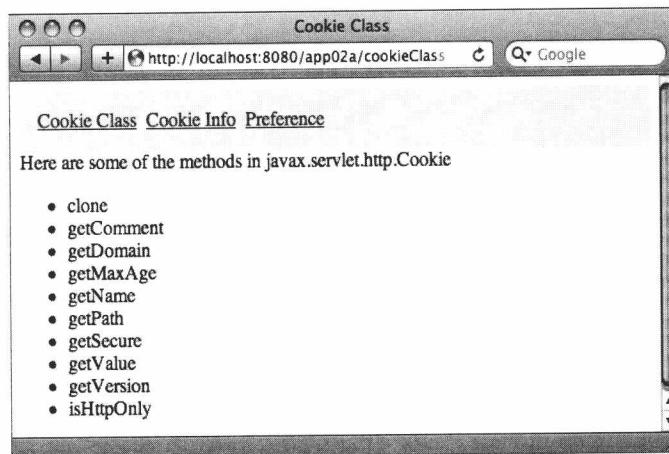


图 2-7 CookieClassServlet 的输出结果

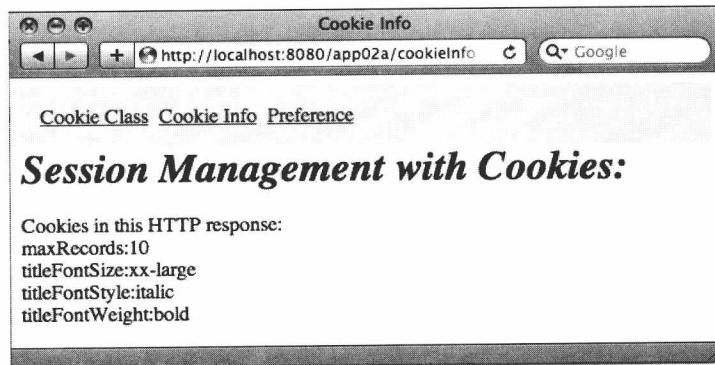


图 2-8 CookieInfoServlet 的输出结果

## 2.4 HttpSession 对象

在所有的 Session 追踪技术中，`HttpSession` 对象是最强大的，也是功能最多的。用户可以没有或者有一个 `HttpSession`，并且只能访问他 / 她自己的 `HttpSession`。

`HttpSession` 是当一个用户第一次访问某个网站时自动创建的。通过在 `HttpServlet`-

**Request** 中调用 `getSession` 方法，可以获取用户的 `HttpSession`。`getSession` 有两个重载方法：

```
 HttpSession getSession()
 HttpSession getSession(boolean create)
```

无参的 `getSession` 方法返回当前的 `HttpSession`，如果当前没有，则创建一个并返回。`getSession(false)` 方法返回当前的 `HttpSession`（若有），如果没有，则返回 `null`。`getSession(true)` 方法返回当前的 `HttpSession`（若有），如果没有，则新建一个并返回。`getSession(true)` 和 `getSession()` 是一样的。

`HttpSession` 的 `setAttribute` 方法将一个值放在 `HttpSession` 中，其方法签名如下：

```
 void setAttribute(java.lang.String name, java.lang.Object value)
```

注意，与网址重写、隐藏域和 cookie 不同的地方在于，放在 `HttpSession` 中的值是保存在内存中的。因此，你只能将尽可能小的对象放在里面，并且数量不能太多。即使现代的 Servlet 容器可以在内存将满时将 `HttpSession` 中的对象移到辅助存储设备中，但是这样会影响性能。因此，对于保存在 `HttpSession` 里面的内容一定要慎重。

添加到 `HttpSession` 中的值不一定是 `String`，可以为任意 Java 对象，只要它的类实现了 `java.io.Serializable` 接口即可，以便当 Servlet 容器认为有必要的时候，保存的对象可以序列化成一个文件或者保存到数据库中，例如，当容器的内存快要用完的时候。仍然可以将非序列化的对象保存在 `HttpSession` 中，但是如果 Servlet 容器试图将它们序列化，将会以失败告终，并抛出异常。

`setAttribute` 方法要求不同的对象要有不同的名称。如果传递一个之前用过的属性名称，那么该名称将与旧值无关联，而与新值相关联了。

通过在 `HttpSession` 中调用 `getAttribute` 方法，同时传递一个属性名称，可以获取 `HttpSession` 中保存的对象。这个方法的签名如下：

```
 java.lang.Object getAttribute(java.lang.String name)
```

`HttpSession` 中另一个有用的方法是 `getAttributeNames`，它返回一个 `Enumeration`，迭代一个 `HttpSession` 中的所有属性：

```
 java.util.Enumeration<java.lang.String> getAttributeNames()
```

注意，`HttpSession` 中保存的值不发送到客户端，这与其他的 Session 管理方法不同。而是 Servlet 容器为它所创建的每一个 `HttpSession` 生成一个唯一标识符，并将这个标识符作为一个 `token` 发送给浏览器，一般是作为一个名为 `JSESSIONID` 的 cookie，或者作为一个 `jsessionid` 参数添加到 URL 后面。在后续的请求中，浏览器会将这个 `token` 发送回服务器，

使服务器能够知道是哪个用户在发出请求。无论 Servlet 容器选择用哪一种方式传输 session 标识符，那都是在后台自动完成的，不需要你去做额外的处理工作。

通过在 `HttpSession` 中调用 `getId` 方法，可以获取 `HttpSession` 的标识符。

```
java.lang.String getId()
```

`HttpSession` 中还定义了一个 `invalidate` 方法。这个方法强制 Session 过期，并将绑定到它的所有对象都解除绑定。在默认情况下，`HttpSession` 是在用户静默一定时间之后过期。可以在部署描述符的 `session-timeout` 元素中将 session 的期限设置为整个应用程序（详情查看第 16 章的内容）。例如，将这个值设为 30，使所有 session 对象在用户最后一次访问之后 30 分钟过期。如果没有配置这个元素，这个期限将由 Servlet 容器决定。

很多时候，还需要销毁未过期却又没用的 `HttpSession` 实例，以便释放一些内存空间。

可以调用 `getMaxInactiveInterval` 方法，以了解一个 `HttpSession` 在用户最后一次访问之后还可以维持多久。这个方法返回用户离开的秒数。`setMaxInactiveInterval` 方法可以帮助你为个别 `HttpSession` 的 Session 期限设置一个不同的值。

```
void setMaxInactiveInterval(int seconds)
```

如果向这个方法传递 0，那么 `HttpSession` 将永远不会过期。一般来说，这不是一种好办法，因为 `HttpSession` 占用的堆（heap）空间将永远不会释放，直到应用程序卸载或 Servlet 容器关闭为止。

举个例子，请看代码清单 2-9 中的 `ShoppingCartServlet` 类。这个 Servlet 实现了一个小型的在线商店，里面有 4 种商品。它允许用户将商品添加到购物车中，并浏览它的内容。Servlet 利用代码清单 2-7 中的 `Product` 类和代码清单 2-8 中的 `ShoppingItem` 类。`Product` 定义了 4 个属性 (`id`、`name`、`description` 及 `price`)，`ShoppingItem` 则包含一个 `quantity` 和一个 `Product`。

代码清单 2-7 Product 类

---

```
package app02a.httpssession;
public class Product {
    private int id;
    private String name;
    private String description;
    private float price;

    public Product(int id, String name, String description, float
        price) {
        this.id = id;
        this.name = name;
        this.description = description;
        this.price = price;
    }
}
```

```
// get and set methods not shown to save space
}
```

---

#### 代码清单 2-8 ShoppingItem 类

```
package app02a.httpsession;
public class ShoppingItem {
    private Product product;
    private int quantity;

    public ShoppingItem(Product product, int quantity) {
        this.product = product;
        this.quantity = quantity;
    }

    // get and set methods not shown to save space
}
```

---

#### 代码清单 2-9 ShoppingCartServlet 类

```
package app02a.httpsession;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.NumberFormat;
import java.util.ArrayList;
import java.util.List;
import java.util.Locale;

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

@WebServlet(name = "ShoppingCartServlet", urlPatterns = {
    "/products", "/viewProductDetails",
    "/addToCart", "/viewCart" })
public class ShoppingCartServlet extends HttpServlet {
    private static final long serialVersionUID = -20L;
    private static final String CART_ATTRIBUTE = "cart";

    private List<Product> products = new ArrayList<Product>();
    private NumberFormat currencyFormat = NumberFormat
        .getCurrencyInstance(Locale.US);

    @Override
    public void init() throws ServletException {
        products.add(new Product(1, "Bravo 32' HDTV",
            "Low-cost HDTV from renowned TV manufacturer",
            159.95F));
        products.add(new Product(2, "Bravo BluRay Player",
            "A high-quality BluRay player with built-in Wi-Fi",
            399.99F));
    }
}
```

```

        "High quality stylish BluRay player", 99.95F));
products.add(new Product(3, "Bravo Stereo System",
    "5 speaker hifi system with iPod player",
    129.95F));
products.add(new Product(4, "Bravo iPod player",
    "An iPod plug-in that can play multiple formats",
    39.95F));
}

@Override
public void doGet(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
String uri = request.getRequestURI();
if (uri.endsWith("/products")) {
    sendProductList(response);
} else if (uri.endsWith("/viewProductDetails")) {
    sendProductDetails(request, response);
} else if (uri.endsWith("viewCart")) {
    showCart(request, response);
}
}

@Override
public void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
// add to cart
int productId = 0;
int quantity = 0;
try {
    productId = Integer.parseInt(
        request.getParameter("id"));
    quantity = Integer.parseInt(request
        .getParameter("quantity"));
} catch (NumberFormatException e) {
}

Product product = getProduct(productId);
if (product != null && quantity >= 0) {
    ShoppingItem shoppingItem = new ShoppingItem(product,
        quantity);
    HttpSession session = request.getSession();
    List<ShoppingItem> cart = (List<ShoppingItem>) session
        .getAttribute(CART_ATTRIBUTE);
    if (cart == null) {
        cart = new ArrayList<ShoppingItem>();
        session.setAttribute(CART_ATTRIBUTE, cart);
    }
    cart.add(shoppingItem);
}
}

```

```

        sendProductList(response);
    }

private void sendProductList(HttpServletRequest response)
    throws IOException {
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.println("<html><head><title>Products</title>" +
        "</head><body><h2>Products</h2>");
    writer.println("<ul>");
    for (Product product : products) {
        writer.println("<li>" + product.getName() + "("
            + currencyFormat.format(product.getPrice())
            + ")" + "<a href='viewProductDetails?id="
            + product.getId() + "'>Details</a>)");
    }
    writer.println("</ul>");
    writer.println("<a href='viewCart'>View Cart</a>");
    writer.println("</body></html>");

}

private Product getProduct(int productId) {
    for (Product product : products) {
        if (product.getId() == productId) {
            return product;
        }
    }
    return null;
}

private void sendProductDetails(HttpServletRequest request,
    HttpServletResponse response) throws IOException {
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    int productId = 0;
    try {
        productId = Integer.parseInt(
            request.getParameter("id"));
    } catch (NumberFormatException e) {
    }
    Product product = getProduct(productId);

    if (product != null) {
        writer.println("<html><head>" +
            "<title>Product Details</title></head>" +
            "<body><h2>Product Details</h2>" +
            "<form method='post' action='addToCart'>" +
            writer.println("<input type='hidden' name='id' "
                + "value='" + productId + "'/>");

        writer.println("<table>");
```

```

        writer.println("<tr><td>Name:</td><td>" +
                      + product.getName() + "</td></tr>");
        writer.println("<tr><td>Description:</td><td>" +
                      + product.getDescription() + "</td></tr>");
        writer.println("<tr>" + "<tr>" +
                      + "<td><input name='quantity' /></td>" +
                      + "<td><input type='submit' value='Buy' />" +
                      + "</td>" +
                      + "</tr>\"");
        writer.println("<tr><td colspan='2'>" +
                      + "<a href='products'>Product List</a>" +
                      + "</td></tr>\"");
        writer.println("</table>");
        writer.println("</form></body>");

    } else {
        writer.println("No product found");
    }
}

private void showCart(HttpServletRequest request,
                      HttpServletResponse response) throws IOException {
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.println("<html><head><title>Shopping Cart</title>" +
                  + "</head>\"");
    writer.println("<body><a href='products'>" +
                  "Product List</a>\"");
    HttpSession session = request.getSession();
    List<ShoppingItem> cart = (List<ShoppingItem>) session
        .getAttribute(CART_ATTRIBUTE);
    if (cart != null) {
        writer.println("<table>");
        writer.println("<tr><td style='width:150px'>Quantity" +
                      + "</td>" +
                      + "<td style='width:150px'>Product</td>" +
                      + "<td style='width:150px'>Price</td>" +
                      + "<td>Amount</td></tr>\"");
        double total = 0.0;
        for (ShoppingItem shoppingItem : cart) {
            Product product = shoppingItem.getProduct();
            int quantity = shoppingItem.getQuantity();
            if (quantity != 0) {
                float price = product.getPrice();
                writer.println("<tr>");
                writer.println("<td>" + quantity + "</td>");
                writer.println("<td>" + product.getName() +
                              + "</td>\"");
                writer.println("<td>" +
                              + currencyFormat.format(price) +
                              + "</td>\"");
                double subtotal = price * quantity;

```

```

        writer.println("<td>" +
                      + currencyFormat.format(subtotal) +
                      + "</td>");
        total += subtotal;
        writer.println("</tr>");
    }
}
writer.println("<tr><td colspan='4' " +
              + "style='text-align:right'" +
              + "Total:" +
              + currencyFormat.format(total) +
              + "</td></tr>");
writer.println("</table>");
}
writer.println("</table></body></html>");

}
}

```

---

ShoppingCartServlet Servlet 映射为以下这些 URL 模式:

- /products。展示所有商品。
- /viewProductDetails。展示某件商品的详细描述。
- /addToCart。将一件商品添加到购物车中。
- /viewCart。展示购物车的内容。

除 /addToCart 之外的所有 URL 都要调用 ShoppingCartServlet 的 doGet 方法。doGet 首先查看请求的 URI，并生成相应的内容:

```

String uri = request.getRequestURI();
if (uri.endsWith("/products")) {
    sendProductList(response);
} else if (uri.endsWith("/viewProductDetails")) {
    sendProductDetails(request, response);
} else if (uri.endsWith("viewCart")) {
    showCart(request, response);
}

```

下列 URL 是调用应用程序的主页面:

<http://localhost:8080/app02a/products>

访问这个 URL 将会执行 doGet 方法，然后将一个商品清单发送到浏览器（如图 2-9 所示）。

如果单击 Details 链接，doGet 就会显示出被选商品的详细说明，如图 2-10 所示。注意输入域和 Buy 按钮了吗？要添加商品时，可以在输入域中输入一个数字，并单击 Buy（立即购买）按钮。

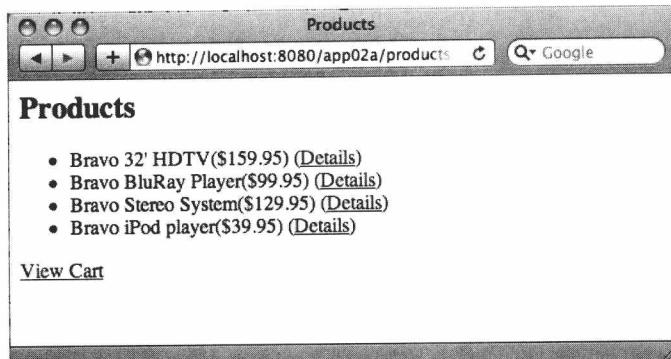


图 2-9 商品清单

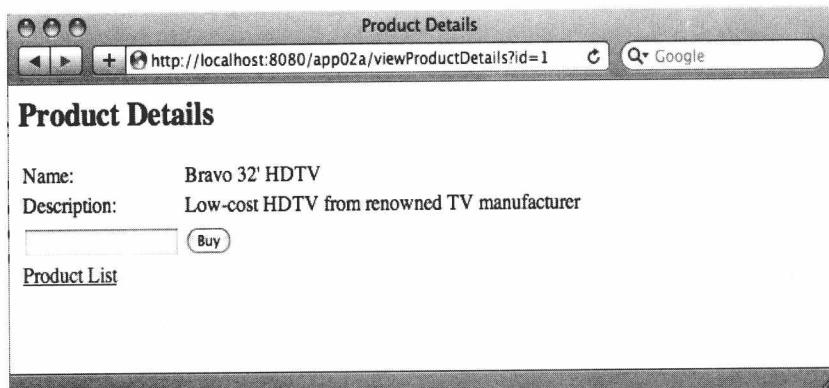


图 2-10 Product Details 页面

在 Product Details 页面中提交 Buy 表单，将会调用 ShoppingCartServlet 的 doPost 方法。正是这个时候往用户的 HttpSession 中添加了一件商品。

doPost 方法首先根据用户输入的数量和被选商品的标识符构造一个 ShoppingItem：

```
ShoppingItem shoppingItem = new ShoppingItem(product,
    quantity);
```

之后，它获取当前用户的 HttpSession，查看其中是否已经包含一个与属性名称“cart”相关的 List：

```
HttpSession session = request.getSession();
List<ShoppingItem> cart = (List<ShoppingItem>) session
    .getAttribute(CART_ATTRIBUTE);
```

如果找到 List，将会用它添加 ShoppingItem。如果没有找到 List，则会创建一个，再添

加到 HttpSession 中。

```
if (cart == null) {
    cart = new ArrayList<ShoppingItem>();
    session.setAttribute(CART_ATTRIBUTE, cart);
}
```

最终，把 ShoppingItem 添加到了列表中。

```
cart.add(shoppingItem);
```

当用户单击 View Cart 链接查看购物车的内容时，会再次调用 doGet 方法，并调用 showCart 方法。后者获取当前用户的 HttpSession，并调用它的 getAttribute 方法，获得购物清单：

```
HttpSession session = request.getSession();
List<ShoppingItem> cart = (List<ShoppingItem>) session
    .getAttribute(CART_ATTRIBUTE);
```

之后，它迭代 List，并将每件商品的内容发送给浏览器：

```
if (cart != null) {
    for (ShoppingItem shoppingItem : cart) {
        Product product = shoppingItem.getProduct();
        int quantity = shoppingItem.getQuantity();
        ...
    }
}
```

## 2.5 小结

本章学习了 Session 管理和 4 种 Session 管理技术。网址重写和隐藏域主要针对“轻量化”的 Session 追踪技术，它适用于不需要跨越许多页面的信息。另两种技术：cookie 和 HttpSession 对象则比较灵活，但绝非没有局限性。在使用 HttpSession 对象时要特别小心，因为每个对象都会消耗服务器的内存。

# 第3章 JSP

在第1章中我们已经知道，Servlet有两个缺陷无法克服。第一，在Servlet中编写的所有HTML标签都必须包在Java字符串中，这使得发送HTTP响应变成一项十分繁琐的工作。第二，所有文本和HTML标签都必须进行硬编码；因此，即使只对表示层做极其微小的修改，例如，修改背景颜色，也需要重新编译。

JavaServer Pages (JSP) 解决了Servlet中的这两个问题。但是，JSP并没有取代Servlet，而是对它进行了补充。现代的Java Web应用程序都使用Servlet和JSP页面。在编写本书之时，JSP规范的最新版本是2.2。

本章将先对JSP进行概述，再详细讨论JSP页面中的注释（comment）、隐式对象（implicit object），以及三个句法元素，分别是指令（directive）、脚本元素（scripting element）及动作（action）。本章结尾处将讨论错误处理。

JSP可以用标准语法或者XML语法进行编写。用XML语法编写的JSP页面称作JSP文档。用XML语法编写的JSP非常少见，这里不做阐述。本章将介绍用标准语法编写的JSP。

## 3.1 JSP概述

JSP页面其实是一个Servlet。但是，使用JSP页面则比Servlet要容易得多，这有两个原因：第一，不需要编译JSP页面；第二，JSP页面一般是扩展名为.jsp的文本文件，可以利用任何文本编辑器来编写。

JSP页面是在JSP容器中运行的。Servlet容器一般也是JSP容器。例如，Tomcat就是一个Servlet/JSP容器。

第一次请求一个JSP页面时，Servlet/JSP容器要做两件事情：

1. 将JSP页面转换成一个JSP页面实现类，这是一个实现javax.servlet.jsp.JspPage接口或其子接口javax.servlet.jsp.HttpjspPage的Java类。JspPage是javax.servlet.Servlet的子接口，这样会使每个JSP页面都成为一个Servlet。所生成Servlet的类名取决于Servlet/JSP容器。这一点不必操心，因为不需要你直接处理。如果有转换错误，错误消息将会发送到客户端。

2. 如果转换成功，Servlet/JSP容器将会编译Servlet类。之后，容器加载和实例化Java字节码，并执行它通常对Servlet所做的生命周期操作。

对于同一个JSP页面的后续请求，Servlet/JSP容器会查看这个JSP页面自从最后一次转换以来是否修改过。如果修改过，就会重新转换、重新编译，并执行。如果没有，则执行内

存中已经存在的 JSP Servlet。这样，第一次调用 JSP 页面的时间总是会比后续请求的更长，因为它需要转换和编译。为了解决这个问题，可以采取以下任意一种措施：

- 配置应用程序，以便在应用程序启动之时，调用所有的 JSP 页面（实际上是指转换和编译），而不是在初始请求时才调用。
  - 预先编译 JSP 页面，并将它们以 Servlet 的方式进行部署。
- JSP 中有一个 API，其中包含 4 个包：
- `javax.servlet.jsp`。包含核心类和接口，Servlet/JSP 容器用它们将 JSP 页面转换成 Servlet。`JspPage` 和 `HttpJspPage` 接口是这个包中的重要成员。所有 JSP 页面实现类都必须实现 `JspPage` 或 `HttpJspPage`。在 HTTP 环境下，显然是选择 `HttpJspPage`。
  - `javax.servlet.jsp.tagext`。包含用于开发定制标签的类型（详情查看第 6 章的内容）。
  - `javax.el`。为 Unified Expression Language 提供 API。详情查看第 4 章的内容。
  - `javax.servlet.jsp.el`。提供 Servlet/JSP 容器必须支持的类，以便支持 JSP 中的 Expression Language。

除了 `javax.servlet.jsp.tagext` 之外，很少需要直接用到 JSP API。事实上，在编写 JSP 页面时，相对于 JSP API 本身，你会更关注 Servlet API。当然，你还需要掌握 JSP 语法，这个在本章中会讲到。在开发 JSP 容器或者 JSP 编译器的时候，就要大量使用 JSP API。

在下列网站可以查看到 JSP API：

[http://download.oracle.com/docs/cd/E17802\\_01/products/products/jsp/2.1/docs/jsp-2\\_1-pfd2/index.html](http://download.oracle.com/docs/cd/E17802_01/products/products/jsp/2.1/docs/jsp-2_1-pfd2/index.html)

JSP 页面可以包含模板数据和句法元素。对于 JSP 转换器而言，元素具有特别的含义。例如，`<%` 是一个元素，因为它在 JSP 页面中表示一个 Java 代码块的开始。`%>` 也是一个元素，因为它表示一个 Java 代码块的结束。不属于元素的其他内容都是模板数据。模板数据也发送到浏览器。例如，JSP 页面中的 HTML 标签和文本都是模板数据。

代码清单 3-1 给出了一个名为 `welcome.jsp` 的 JSP 页面。这是一个简单的页面，它只是给客户端发送了一条问候消息。你注意到了吗？与具有相同功能的 Servlet 相比，JSP 页面是多么简单啊！

代码清单 3-1 `welcome.jsp` 页面

---

```
<html>
<head><title>Welcome</title></head>
<body>
Welcome
</body>
</html>
```

---

在 Tomcat 中，第一次调用完 `welcome.jsp` 页面之后，它被转换成一个 `welcome.jsp` Servlet。你可以在 Tomcat 的 `work` 目录的子目录下看到所生成的 Servlet 文件。这个

Servlet 继承了 org.apache.jasper.runtime.HttpJspBase，这是一个继承 javax.servlet.http.HttpServlet 并实现 javax.servlet.jsp.HttpJspPage 的抽象类。

下面是为 welcome.jsp 生成的 Servlet 文件。如果你现在觉得它很神秘，不必担心。就算你目前不理解，也可以继续往下学。不过如果你能理解，当然就更好了。

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class welcome_jsp extends
    org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    private static final javax.servlet.jsp.JspFactory _jspxFactory =
        javax.servlet.jsp.JspFactory.getDefaultFactory();

    private static java.util.Map<java.lang.String,java.lang.Long>
        _jspx_dependants;

    private javax.el.ExpressionFactory _el_expressionfactory;
    private org.apache.tomcat.InstanceManager _jsp_instancemanager;

    public java.util.Map<java.lang.String,java.lang.Long>
        getDependants() {
        return _jspx_dependants;
    }

    public void _jspInit() {
        _el_expressionfactory =
            _jspxFactory.getJspApplicationContext(
                getServletConfig().getServletContext())
            .getExpressionFactory();
        _jsp_instancemanager =
            org.apache.jasper.runtime.InstanceManagerFactory
            .getInstanceManager(getServletConfig());
    }

    public void _jspDestroy() {
    }

    public void _jspService(final
        javax.servlet.http.HttpServletRequest request, final
        javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {

        final javax.servlet.jsp.PageContext pageContext;
        javax.servlet.http.HttpSession session = null;
        final javax.servlet.ServletContext application;
        final javax.servlet.ServletConfig config;
```

```
javax.servlet.jsp.JspWriter out = null;
final java.lang.Object page = this;
javax.servlet.jsp.JspWriter _jspx_out = null;
javax.servlet.jsp.PageContext _jspx_page_context = null;

try {
    response.setContentType("text/html");
    pageContext = _jspxFactory.getPageContext(this, request,
        response, null, true, 8192, true);
    _jspx_page_context = pageContext;
    application = pageContext.getServletContext();
    config = pageContext.getServletConfig();
    session = pageContext.getSession();
    out = pageContext.getOut();
    _jspx_out = out;

    out.write("<html>\n");
    out.write("<head><title>Welcome</title></head>\n");
    out.write("<body>\n");
    out.write("Welcome\n");
    out.write("</body>\n");
    out.write("</html>");
} catch (java.lang.Throwable t) {
    if (!(t instanceof
        javax.servlet.jsp.SkipPageException)) {
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            try {
                out.clearBuffer();
            } catch (java.io.IOException e) {
            }
        if (_jspx_page_context != null)
            _jspx_page_context.handlePageException(t);
    }
} finally {
    _jspxFactory.releasePageContext(_jspx_page_context);
}
}
```

从上述代码可以看出，JSP 页面的主体被转换成一个 `_jspService` 方法。这个方法在 `HttpJspPage` 中定义，并且通过 `HttpJspBase` 的 `service` 方法实现调用。下面是来自 `HttpJspBase` 类的内容：

```
public final void service(HttpServletRequest request,
    HttpServletResponse response) throws ServletException,
    IOException {
    _jspService(request, response);
}
```

为了覆盖 init 和 destroy 方法，可以根据本章稍后的 3.5 节的内容来声明方法。

JSP 页面与 Servlet 不同的另一个方面是，前者不需要在部署描述符中进行标注，或映射成一个 URL。应用程序目录下的每一个 JSP 页面都可以通过在浏览器中输入页面的路径来实现直接的调用。图 3-1 展示了 app03a 的目录结构，这是本章配套提供的一个 JSP 应用程序范例。

因为只有一个 JSP 页面，因此 app03a 应用程序的结构非常简单，只包含一个空的 WEB-INF 目录和一个 welcome.jsp 页面。

利用下面这个 URL 可以调用 welcome.jsp 页面：

```
http://localhost:8080/app03a/welcome.jsp
```

**提示** 添加完一个新的 JSP 页面之后，不需要重启 Tomcat。

代码清单 3-2 展示了如何在 JSP 中利用 Java 代码来生成动态的页面。代码清单 3-2 中的 todaysDate.jsp 页面展示了当天的日期。

代码清单 3-2 todaysDate.jsp 页面

```
<%@page import="java.util.Date"%>
<%@page import="java.text.DateFormat"%>
<html>
<head><title>Today's date</title></head>
<body>
<%
    DateFormat dateFormat =
        DateFormat.getDateInstance(DateFormat.LONG);
    String s = dateFormat.format(new Date());
    out.println("Today is " + s);
%>
</body>
</html>
```

todaysDate.jsp 页面会将几个 HTML 标签和字符串 “Today is” 加当天的日期发送到浏览器。

有两件事情需要注意。第一，Java 代码要用 <% 和 %> 包起来，并且可以放在 JSP 页面中的任何位置；第二，为了导入一个 JSP 页面中要用到的 Java 类型，可以利用 page 指令的 import 属性。如果没有导入类型，那么在代码中必须编写 Java 类型的全类名。

<%...%> 块称作 **scriptlet**，在本章稍后的 3.5 节还会进一步讨论它。page 指令将在本章稍后的 3.4 节中做详细的探讨。

利用下面这个 URL 可以调用 todaysDate.jsp 页面：

```
http://localhost:8080/app03a/todaysDate.jsp
```



图 3-1 app03a 的应用  
程序目录结构

## 3.2 备注

给 JSP 页面添加备注是一种良好的编程习惯。JSP 页面中可以使用两种备注：

1. JSP 备注。注明该页面的作用。
2. HTML/XHTML 备注。这些将被发送到浏览器。

JSP 备注以 `<%--` 开头，以 `--%>` 结束。例如，下面就是一个 JSP 备注：

```
<%-- retrieve products to display --%>
```

JSP 备注不会被发送到浏览器，也不能进行嵌套。

HTML/XHTML 备注的语法如下：

```
<!-- [comments here] -->
```

HTML/XHTML 备注不是由容器进行处理，而是被发送到浏览器。HTML/XHTML 备注的用途之一是标识 JSP 页面：

```
<!-- this is /jsp/store/displayProducts.jspf -->
```

在处理带有许多 JSP 片断 (fragment) 的应用程序时，这个特别有用。开发者通过查看浏览器的 HTML 源代码，可以轻松查出某个 HTML 代码部分生成了哪个 JSP 页面或者哪个片断。

## 3.3 隐式对象

Servlet 容器将几个对象传给它所运行的 Servlet。例如，在 Servlet 的 `service` 方法中获得 `HttpServletRequest` 和 `HttpServletResponse`，并在 `init` 方法中获得 `ServletConfig`。此外，还可以通过在 `HttpServletRequest` 对象中调用 `getSession` 获得一个 `HttpSession`。

在 JSP 中，可以通过使用隐式对象来获取那些对象。表 3-1 列出了隐式对象。

例如，隐式对象 `request` 表示由 Servlet/JSP 容器传给 Servlet 的 `service` 方法的 `HttpServletRequest`。可以像使用 `HttpServletRequest` 的变量引用一样使用 `request`。例如，以下代码是从 `HttpServletRequest` 对象处获取 `userName` 参数。

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

`pageContext` 是指为页面创建的 `javax.servlet.jsp.PageContext`。它提供了有用的 `context` 信息，并通过一些名如其义的方法来访问与 Servlet 有关的各种对象，例如，`getRequest`、`getResponse`、`getServletContext`、`getServletConfig` 及 `getSession`。这些方法在 Scriptlet 中作用不大，因为它们所返回的对象可以通过隐式对象 `request`、`response`、

`session` 及 `application` 更直接地访问到。但是，如第 4 章所述，`PageContext` 允许利用 Expression Language 访问那些对象。

表 3-1 JSP 隐式对象

对    象	类    型
<code>request</code>	<code>javax.servlet.http.HttpServletRequest</code>
<code>response</code>	<code>javax.servlet.http.HttpServletResponse</code>
<code>out</code>	<code>javax.servlet.jsp.JspWriter</code>
<code>session</code>	<code>javax.servlet.http.HttpSession</code>
<code>application</code>	<code>javax.servlet.ServletContext</code>
<code>config</code>	<code>javax.servlet.ServletConfig</code>
<code>pageContext</code>	<code>javax.servlet.jsp.PageContext</code>
<code>page</code>	<code>javax.servlet.jsp.HttpJspPage</code>
<code>exception</code>	<code>java.lang.Throwable</code>

`PageContext` 提供的另一些重要的方法是那些存取属性，如 `getAttribute` 和 `setAttribute` 方法。属性可以保存在以下 4 种范围中：`page`、`request`、`session` 及 `application`。`page` 范围最窄，保存在这里的属性只能在同一个 JSP 页面中使用。`request` 范围是指当前的 `ServletRequest`，`session` 范围是指当前的 `HttpSession`，`application` 范围是指 `ServletContext`。

`PageContext` 中的 `setAttribute` 方法具有以下签名：

```
public abstract void setAttribute(java.lang.String name,
                                   java.lang.Object value, int scope)
```

`scope` 值可以是以下任意一个 `PageContext` 中的 static final int 值：`PAGE_SCOPE`、`REQUEST_SCOPE`、`SESSION_SCOPE` 及 `APPLICATION_SCOPE`。

另外，属性也可以保存在 `page` 范围中，可以利用下面这个 `setAttribute` 方法重载：

```
public abstract void setAttribute(java.lang.String name,
                                   java.lang.Object value)
```

例如，下面这个 Scriptlet 在 `ServletRequest` 中保存了一个属性。

```
<%
    // product is a Java object
    pageContext.setAttribute("product", product,
                           PageContext.REQUEST_SCOPE);
%>
```

上述代码与下面这行代码的效果是一样的：

```
<%
    request.setAttribute("product", product);
%>
```

隐式对象 `out` 引用 `javax.servlet.jsp.JspWriter`, 它类似于在 `HttpServletResponse` 中调用 `getWriter()` 之后得到的 `java.io.PrintWriter`。也可以调用它的 `print` 方法重载 `PrintWriter`, 将消息发送到浏览器。例如:

```
out.println("Welcome");
```

代码清单 3-3 中的 `implicitObjects.jsp` 页面示范了部分隐式对象的用法。

代码清单 3-3 `implicitObjects.jsp` 页面

---

```
<%@page import="java.util.Enumeration"%>
<html>
<head><title>JSP Implicit Objects</title></head>
<body>
<b>Http headers:</b><br/>
<%
    for (Enumeration<String> e = request.getHeaderNames() ;
        e.hasMoreElements(); ) {
        String header = e.nextElement();
        out.println(header + ": " + request.getHeader(header) +
                    "<br/>");
    }
%>
<hr/>
<%
    out.println("Buffer size: " + response.getBufferSize() +
                "<br/>");
    out.println("Session id: " + session.getId() + "<br/>");
    out.println("Servlet name: " + config.getServletName() +
                "<br/>");
    out.println("Server info: " + application.getServerInfo());
%>
</body>
</html>
```

---

可以利用下面这个 URL 调用 `implicitObjects.jsp` 页面:

`http://localhost:8080/app03a/implicitObjects.jsp`

这个页面在浏览器中产生了以下文本:

```
Http headers:
host: localhost:8080
user-agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_5_8)
           AppleWebKit/534.50.2 (KHTML, like Gecko) Version/5.0.6
```

```

Safari/533.22.3
accept:text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=
0.8
accept-language: en-us
accept-encoding: gzip, deflate
connection: keep-alive

Buffer size: 8192
Session id: 561DDD085ADD99FC03F70BDEE87AAF4D
Servlet name: jsp
Server info: Apache Tomcat/7.0.14

```

你在浏览器中看到的具体内容取决于你使用什么样的浏览器，及其所处的环境。

注意，在默认情况下，JSP 编译器是将 JSP 页面的内容类型设为 `text/html`。如果你发送了另一种不同的类型，就必须通过调用 `response.setContentType()` 或者利用 `page` 指令（详情查看 3.4 节）来设置内容类型。例如，下面的代码就是将内容类型设为 `text/json`：

```
response.setContentType("text/json");
```

还要注意，隐式对象 `page` 表示当前的 JSP 页面，一般不为 JSP 页面的设计者所用。

## 3.4 指令

指令（Directive）是第一种 JSP 句法元素，其指示 JSP 转换器应该如何将某个 JSP 页面转换成 Servlet 的命令。JSP 2.2 中定义了几个指令，但最重要的是这两个：`page` 和 `include`，这些在本章中都会讨论到。其他章节中还会讲到的其他指令有：`taglib`、`tag`、`attribute` 及 `variable`。

### 3.4.1 page 指令

利用 `page` 指令可以就当前 JSP 页面的某些方面对 JSP 转换器提出指示。例如，可以告诉 JSP 转换器隐式对象 `out` 应该使用多大容量的缓存区，要使用哪种内容类型，要导入哪些 Java 类型等。

`page` 指令的语法如下：

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

`@` 和 `page` 之间的空格是可选的，`attribute1`、`attribute2` 等都是 `page` 指令的属性。下面是 `page` 指令的属性列表：

- ❑ `import`。指定要导入的一种或多种 Java 类型，供本页的 Java 代码所用。例如，`import="java.util.List"` 导入 `List` 接口。利用通配符 \* 还可以导入整个包，如 `import=`

"java.util.\*"。导入多种类型时，两种类型之间要用一个逗号隔开，如 `import="java.util.ArrayList,java.util.Calendar,java.io.PrintWriter"`。下面这些包中的所有类型都是隐式导入的：`java.lang`、`javax.servlet`、`javax.servlet.http`、`javax.servlet.jsp`。

- ❑ `session`。值为 `true` 时，表示这个页面参与 Session 管理；值为 `false` 时，表示不参与 Session 管理。默认值为 `true`，意味着如果之前还没有 `javax.servlet.http.HttpSession` 实例，那么调用 JSP 页面将始终会创建一个。
- ❑ `buffer`。指定隐式对象 `out` 的缓冲区大小，以千字节为单位。强制以 `kb` 作为后缀。缓冲区的默认容量大于或等于 8 KB，具体取决于 JSP 容器。这个属性值还可以为 `none`，表示不使用缓存，但这样会导致输出的内容直接被写入相应的 `PrintWriter`。
- ❑ `autoFlush`。默认值为 `true`，表示当缓冲区满时，被缓存的输出应该自动刷新。值为 `false` 时，表示只有在调用隐式对象 `response` 的 `flush` 方法时，才进行刷新缓冲区。因此，当缓冲区溢出时会抛出一个异常。
- ❑ `isThreadSafe`。表示页面中实现的线程安全级别。建议 JSP 的作者不要使用这个属性，因为它会产生一个包含不建议使用的代码的 `Servlet`。
- ❑ `info`。指定所生成 `Servlet` 的 `getServletInfo` 方法返回值。
- ❑ `errorCode`。表示负责处理该页面可能出现的错误的页面。
- ❑ `isErrorPage`。表明这个页面是否负责处理错误。
- ❑ `contentType`。指定该页面隐式对象 `response` 的内容类型，其默认值为 `text/html`。
- ❑ `pageEncoding`。指定该页面的字符编码，其默认值为 `ISO-8859-1`。
- ❑ `isELIgnored`。表明是否忽略 EL 表达式。EL 是 Expression Language 的缩写，将在第 4 章中讨论。
- ❑ `language`。指定该页面使用的脚本语言，其默认值为 `java`，这是 JSP 2.2 中唯一有效的值。
- ❑ `extends`。指定这个 JSP 页面的实现类必须扩展的超类。该属性很少使用，使用时应该特别小心。
- ❑ `deferredSyntaxAllowedAsLiteral`。指明是否允许用字符序列 `#{}`  作为该页面和编译单元的 `String` 字面值，其默认值为 `false`。`#{}`  很重要，因为它在 Expression Language 中是一个特殊的字符序列（详情查看第 4 章的相关内容）。
- ❑ `trimDirectiveWhitespaces`。表明是否从输出内容中删除只包含空格的模板文本，其默认值为 `false`，也就是说，不删除空格。

`page` 指令可以出现在页面中的任何位置。只是当它包含 `contentType` 或者 `pageEncoding` 属性时，它就必须放在所有的模板数据之前，并且是在利用 Java 代码发送任何内容之前。这是因为，必须在发送任何内容之前设置内容类型和字符编码。

`page` 指令也可以多次出现。但是，在多个 `page` 指令中多次出现的同一个属性，它的值必须一致，只有 `import` 属性例外。放在多个 `page` 指令中的 `import` 属性，其效果可以累积。

例如，下面的 `page` 指令将会同时导入 `java.util.ArrayList` 和 `java.io.File`。

```
<%@page import="java.util.ArrayList"%>
<%@page import="java.util.Date"%>
```

其结果与下面这行代码的相同：

```
<%@page import="java.util.ArrayList, java.util.Date"%>
```

下面再举个例子。这个 `page` 指令将 `session` 属性值设为 `false`，并将页面的缓冲区容量设为 16 KB：

```
<%@page session="false" buffer="16kb"%>
```

### 3.4.2 include 指令

利用 `include` 指令可以将另一个文件的内容放到当前的 JSP 页面中。在一个 JSP 页面中可以使用多个 `include` 指令。如果某部分特殊的内容需要被其他页面所用，或者被处于不同位置的某个页面所用，那么将这部分内容做成一个 `include` 文件是很有帮助的。

`include` 指令的语法如下：

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

此处 `@` 和 `include` 之间的空格是可选的，并且 `url` 是表示一个 `include` 文件的相对路径。如果 `url` 以一个正斜线 (/) 开头，那么其在服务器中就会被解读成是一条绝对路径。如果不是以正斜线开头，则被解读为是相对于当前 JSP 页面的路径。

JSP 转换器转换 `include` 指令时，用 `include` 文件的内容替换指令。换句话说，如果已经写好如代码清单 3-4 所示 `copyright.jspf` 文件。

---

代码清单 3-4 include 文件 `copyright.jspf`

---

```
<hr/>
&copy;2012 BrainySoftware
<hr/>
```

---

同时，又有如代码清单 3-5 所示的 `main.jsp` 页面。

---

代码清单 3-5 `main.jsp` 页面

---

```
<html>
<head><title>Including a file</title></head>
<body>
This is the included content: <hr/>
<%@ include file="copyright.jspf"%>
</body>
</html>
```

---

那么，在 main.jsp 页面中使用 `include` 指令时，其结果与编写以下 JSP 页面将是一样的。

```
<html>
<head><title>Including a file</title></head>
<body>
This is the included content: <hr/>
<hr/>
&copy;2012 BrainySoftware
<hr/>
</body>
</html>
```

为了让上述 `include` 指令生效，`copyright.jspf` 文件必须与所包含的页面放在同一个目录下。

按照规范，`include` 文件的扩展名应为 `jspf`，表示为 JSP fragment。如今，JSP fragment 也称作 JSP segment，只是为了保持一致，仍然用 `jspf` 作为扩展名。

注意，也可以包含静态的 HTML 文件。

本章稍后将讲到的 `include action`，与 `include` 指令类似。这两者之间的细微差别将在 3.6 节中解释，要理解这两者之间的差别，这一点很重要。

## 3.5 脚本元素

第二种 JSP 句法元素是脚本元素，它将 Java 代码合并成一个 JSP 页面。脚本元素有 3 种类型：Scriptlet、声明及表达式，这些都将在接下来的小节中讨论。

### 3.5.1 Scriptlet

Scriptlet 是一个 Java 代码块，它以 `<%` 开头，以 `%>` 结束。例如，代码清单 3-6 中的 `scriptletTest.jsp` 页面就是使用了 Scriptlet。

代码清单 3-6 使用 scriptlet (`scriptletTest.jsp`)

---

```
<%@page import="java.util.Enumeration"%>
<html>
<head><title>Scriptlet example</title></head>
<body>
<b>Http headers:</b><br/>
<%-- first scriptlet --%>
<%
    for (Enumeration<String> e = request.getHeaderNames();
         e.hasMoreElements(); ) {
        String header = e.nextElement();
        out.println(header + ": " + request.getHeader(header) +
                    "<br/>");
```

```

    }
    String message = "Thank you.";
%>
<hr/>
<%-- second scriptlet --%>
<%
    out.println(message);
%>
</body>
</html>

```

---

上述 JSP 页面中有两个 Scriptlet。注意，在一个 Scriptlet 中定义的变量，对于它后面的其他 Scriptlet 是可见的。

一个 Scriptlet 中的首行代码可以与 `<%` 标签放在同一行，`%>` 标签可以与最后一行代码放在同一行。但是这样会破坏页面的可读性。

### 3.5.2 表达式

表达式 (Expression) 的运算结果会被填入隐式对象 `out` 的 `print` 方法中。表达式以 `<%` 开头，并以 `%>` 结束。例如，以下粗体部分就是一个表达式：

```
Today is <%=java.util.Calendar.getInstance().getTime()%>
```

注意，表达式的后面不需要用分号。

对于这个表达式，JSP 容器会先运算 `java.util.Calendar.getInstance().getTime()`，然后将结果传给 `out.print()`。它与下面这个 Scriptlet 的结果是一样的：

```
Today is
<%
    out.print(java.util.Calendar.getInstance().getTime());
%>
```

### 3.5.3 声明

可以声明 (declaration) 能够在 JSP 页面中使用的变量和方法。声明要用 `<%!` 和 `%>` 包起来。如代码清单 3-7 中 `declarationTst.jsp` 页面展示的 JSP 页面，就声明了一个名为 `getTodaysDate` 的方法。

代码清单 3-7 声明的用法范例 (`declarationTest.jsp`)

---

```

<%!
    public String getTodaysDate() {
        return new java.util.Date();
    }
%>
<html>
<head><title>Declarations</title></head>

```

---

```
<body>
Today is <%=getTodaysDate()%>
</body>
</html>
```

声明可以放在 JSP 页面中的任何位置，并且同一个页面中可以有多个声明。

我们可以利用声明覆盖实现类中的 `init` 和 `destroy` 方法。覆盖 `init`，要声明一个 `jsplInit` 方法；覆盖 `destroy`，要声明一个 `jspDestroy` 方法。这两个方法详解如下：

- `jsplInit`。该方法与 `javax.servlet.Servlet` 中的 `init` 方法类似。JSP 页面被初始化时，就会调用 `jsplInit`。与 `init` 方法不同的是，`jsplInit` 不带参数。你仍然可以通过隐式对象 `config` 来获得 `ServletConfig` 对象。
- `jspDestroy`。该方法与 `Servlet` 中的 `destroy` 方法相似，当 JSP 页面要被销毁时，就会调用它。

代码清单 3-8 中的 `lifeCycle.jsp` 页面，示范了覆盖 `jsplInit` 和 `jspDestroy` 的方法。

代码清单 3-8 lifeCycle.jsp 页面

---

```
<%!
    public void jsplInit() {
        System.out.println("jsplInit ...");
    }
    public void jspDestroy() {
        System.out.println("jspDestroy ...");
    }
%>
<html>
<head><title>jspInit and jspDestroy</title></head>
<body>
Overriding jsplInit and jspDestroy
</body>
</html>
```

`lifecycle.jsp` 页面将被转换成如下这样的 Servlet：

```
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;

public final class lifeCycle_jsp extends
    org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {

    public void jsplInit() {
        System.out.println("jsplInit ...");
    }
}
```

```

public void jspDestroy() {
    System.out.println("jspDestroy ...");
}

private static final javax.servlet.jsp.JspFactory _jspxFactory =
    javax.servlet.jsp.JspFactory.getDefaultFactory();

private static java.util.Map<java.lang.String,java.lang.Long>
    _jspx_dependants;

private javax.el.ExpressionFactory _el_expressionfactory;
private org.apache.tomcat.InstanceManager _jsp_instancemanager;

public java.util.Map<java.lang.String,java.lang.Long>
    getDependants() {
    return _jspx_dependants;
}

public void _jspInit() {
    _el_expressionfactory =
        _jspxFactory.getJspApplicationContext(
            getServletConfig().getServletContext())
        .getExpressionFactory();
    _jsp_instancemanager =
        org.apache.jasper.runtime.InstanceManagerFactory
        .getInstanceManager(getServletConfig());
}

public void _jspDestroy() {
}

public void _jspService(final
    javax.servlet.http.HttpServletRequest request, final
    javax.servlet.http.HttpServletResponse response)
    throws java.io.IOException,
    javax.servlet.ServletException {

    final javax.servlet.jsp.PageContext pageContext;
    javax.servlet.http.HttpSession session = null;
    final javax.servlet.ServletContext application;
    final javax.servlet.ServletConfig config;
    javax.servlet.jsp.JspWriter out = null;
    final java.lang.Object page = this;
    javax.servlet.jsp.JspWriter _jspx_out = null;
    javax.servlet.jsp.PageContext _jspx_page_context = null;

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

```

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

out.write("\n");
out.write("<html>\n");
out.write("<head><title>jspInit and jspDestroy" +
        "</title></head>\n");
out.write("<body>\n");
out.write("Overriding jspInit and jspDestroy\n");
out.write("</body>\n");
out.write("</html>");
} catch (java.lang.Throwable t) {
    if (!(t instanceof
        javax.servlet.jsp.SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            try {
                out.clearBuffer();
            } catch (java.io.IOException e) {
            }
        if (_jspx_page_context != null)
            _jspx_page_context.handlePageException(t);
    }
} finally {
    _jspxFactory.releasePageContext(_jspx_page_context);
}
}
```

注意到上述 Servlet 中的 `jspInit` 和 `jspDestroy` 方法了吗？

利用以下 URL 可以调用 `lifeCycle.jsp`:

`http://localhost:8080/app03a/lifeCycle.jsp`

当第一次调用这个 JSP 页面时，会在控制台上看到“`jspInit...`”，当关闭 Servlet/JSP 容器时，则会看到“`jspDestroy...`”。

### 3.5.4 关闭脚本元素

随着 JSP 2.0 中 Expression Language 的发展，建议做法是利用 EL 来访问服务器端的对象，而不是在 JSP 页面中编写 Java 代码。为此，原本开启的 JSP 2.0 脚本元素，可以通过在部署描述符的 `<jsp-property-group>` 中定义一个 `scripting-invalid` 元素，将它关闭，如下所示：

```
<jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
</jsp-property-group>
```

## 3.6 动作

第三种句法元素是动作 (Action)，它们被编译成执行某个操作的 Java 代码，例如访问某个 Java 对象，或者调用某个方法。本节讨论必须能被所有 JSP 容器支持的标准动作。除标准动作之外，还可以创建定制的标签，用来执行某些操作。定制标签将在第 6 章中讨论。

下面列举部分标准动作。`doBody` 和 `invoke` 这两个标准动作将在第 7 章中讨论。

### 3.6.1 useBean

这个动作将创建一个与某个 Java 对象相关的脚本变量。它是将表现逻辑与业务逻辑分隔开来最容易的方法之一。但是有了像定制标签和 Expression Language 这类技术之后，现在已经很少使用 `useBean` 了。

举个例子。代码清单 3-9 中的 `useBeanTest.jsp` 页面创建了一个 `java.util.Date` 实例，并将它与脚本变量 `today` 关联起来，之后将它用在一个表达式中。

代码清单 3-9 `useBeanTest.jsp` 页面

---

```
<html>
<head>
    <title>useBean</title>
</head>
<body>
<jsp:useBean id="today" class="java.util.Date"/>
<%=today%>
</body>
</html>
```

---

在 Tomcat 中，这个动作会被编译成下面的代码：

```
java.util.Date today = null;
today = (java.util.Date) _jspx_page_context.getAttribute("today",
    javax.servlet.jsp.PageContext.REQUEST_SCOPE);
if (today == null) {
    today = new java.util.Date();
    _jspx_page_context.setAttribute("today", today,
        javax.servlet.jsp.PageContext.REQUEST_SCOPE);
}
```

运行这个页面之后，将会在浏览器中输出当前的日期和时间。

### 3.6.2 setProperty 和 getProperty

`setProperty` 动作是在一个 Java 对象中保存一个属性, `getProperty` 则是获取一个 Java 对象的属性。举个例子, 代码清单 3-11 中的 `get SetPropertyTest.jsp` 页面保存和输出 `Employee` 类实例的 `firstName` 属性, 具体如代码清单 3-10 所示。

代码清单 3-10 Employee 类

---

```
package app03a;
public class Employee {
    private String id;
    private String firstName;
    private String lastName;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

---

代码清单 3-11 getPropertyTest.jsp

---

```
<html>
<head>
<title>getProperty and setProperty</title>
</head>
<body>
<jsp:useBean id="employee" class="app03a.Employee"/>
<jsp:setProperty name="employee" property="firstName"
    value="Abigail"/>
First Name: <jsp:getProperty name="employee" property="firstName"/>
</body>
</html>
```

---

### 3.6.3 include

**include** 动作用于动态地包含另一个资源，它可以包含另一个 JSP 页面、一个 Servlet 或者一个静态的 HTML 页面。例如，代码清单 3-12 中的 `jsplIncludeTest.jsp` 页面就是利用 `include` 动作来包含 `menu.jsp` 页面的。

代码清单 3-12 `jsplIncludeTest.jsp` 页面

---

```
<html>
<head>
<title>Include action</title>
</head>
<body>
<jsp:include page="jspf/menu.jsp">
    <jsp:param name="text" value="How are you?" />
</jsp:include>
</body>
</html>
```

---

理解 `include` 指令和 `include` 动作之间的区别是很重要的。使用 `include` 指令时，这种包含是发生在页面转换的时候，例如 JSP 容器将页面转换成一个生成的 Servlet 的时候。使用 `include` 动作时，这种包含则是发生在请求的时候。因此，可以利用 `include` 动作传递参数，而不是利用 `include` 指令。

第二个区别在于，使用 `include` 指令时，被包含资源的文件扩展名并不重要。而使用 `include` 动作时，文件扩展名则必须为 `jsp`，以便它能够作为一个 JSP 页面进行处理。例如，在 `include` 动作中用 `jspx` 作为扩展名时，将会使得这个 JSP segment 被当作静态文件进行处理。

### 3.6.4 forward

`forward` 动作是将当前页面跳转到另一个不同的资源。例如，下面的 `forward` 动作就是将当前页面跳转到 `login.jsp` 页面。

```
<jsp:forward page="jspf/login.jsp">
    <jsp:param name="text" value="Please login" />
</jsp:forward>
```

### 3.6.5 错误处理

在 JSP 中错误处理支持得很好。你可以利用 `try` 语句处理 Java 代码，也可以指定一个页面，让它在应用程序遇到未捕捉的异常时显示出来。那么，一旦发生异常，用户将会看到一张经过精心设计的页面，解释目前发生了什么状况，而不是用一条错误消息打发用户，让他们皱眉不已。

利用 `page` 指令的 `isErrorPage` 属性，就可以把一个 JSP 页面变成一个错误处理页面，

该属性值必须为 true。代码清单 3-13 展示了这样一个错误处理程序。

代码清单 3-13 errorHandler.jsp 页面

---

```
<%@page isErrorPage="true"%>
<html>
<head><title>Error</title></head>
<body>
An error has occurred. <br/>
Error message:
<%
    out.println(exception.toString());
%>
</body>
</html>
```

---

要防止未捕捉异常的其他页面则必须使用 page 指令的 `errorPage` 属性，将路径引向属性值指定的错误处理页面。例如，代码清单 3-14 中的 `buggy.jsp` 页面就是利用了代码清单 3-13 的错误处理程序。

代码清单 3-14 buggy.jsp 页面

---

```
<%@page errorPage="errorHandler.jsp"%>
Deliberately throw an exception
<%
    Integer.parseInt("Throw me");
%>
```

---

如果运行这个 `buggy.jsp` 页面，它就会抛出一个异常。但你不会看到 Servlet/JSP 容器产生的错误消息，而是会看到 `errorHandler.jsp` 页面显示的内容。

## 3.7 小结

JSP 是在 Java 中构建 Web 应用程序的第二种技术，作为 Servlet 技术的补充，而不是要取代它。设计得好的 Java Web 应用程序一般都会用到 Servlet 和 JSP。

本章学习了 JSP 的工作原理，以及如何编写 JSP 页面。现在，你应该已经了解了关于隐式对象的相关内容，并且能够利用可以在 JSP 页面中使用的 3 种句法元素：指令、脚本元素及动作（action）。

# 第 4 章 EL

JSP 2.0 中最重要的特性之一是 Expression Language (EL)，JSP 设计者可以用它访问应用程序数据。受 ECMAScript 和 XPath 表达式语言的启发，EL 被设计成能够轻松地编写无脚本或不包含 Java 代码的 JSP 页面，即不使用 JSP 声明、表达式或者 Scriptlet 的页面。(第 10 章将会解释为什么无脚本的 JSP 页面会被当作是一种良好的编程习惯。)

EL 最早应用于 JSP Standard Tag Library (JSTL) 1.0 规范中首次出现的 JSP 2.0 中。JSP 1.2 的程序员可以通过将标准的类库导入到他们的应用程序当中，来使用这种语言。JSP 2.0 及以上版本的开发者则不需要 JSTL 就可以使用 EL，不过有许多应用程序仍然需要 JSTL，因为它还包含了与 EL 无关的其他标签。

JSP 2.1 和 JSP 2.2 中的 EL，尝试将 JSP 2.0 中的 EL 和 JavaServer Faces (JSF) 1.0 中定义的 EL 统一起来。JSF 是建立在 JSP 1.2 基础之上，用 Java 快速构建 Web 应用程序的一种框架。由于 JSP 1.2 缺乏一种集成的 EL，而 JSP 2.0 又不能满足 JSF 的所有需求，因此为 JSF 1.0 开发了一种 EL 变体。这两种语言变体后来合二为一。本章只关注针对非 JSF 开发者的 EL。

## 4.1 EL 语法

EL 表达式是以 \${ 开头，以 } 结束。一个 EL 表达式的构造如下：

`${expression}`

例如，编写表达式 `x+y` 时，要使用下面的结构：

`${x+y}`

两个表达式相连接，这种也很常见。表达式的运算顺序是从左到右，结果的类型强制为 String，然后连在一起。例如，如果 `a+b` 等于 8，`c+d` 等于 10，那么下面这两个表达式的结果将是 810：

`${a+b}${c+d}`

如  `${a+b}and${c+d}` 的结果将是 8and10。

如果 EL 表达式用在某个定制标签的属性值中，那么将会运算这个表达式，并强制结果字符串为该属性想要的类型：

```
<my:tag someAttribute=" ${expression}" />
```

字符 \${ 表示某一个 EL 表达式开始了。如果你想要发送字符 \${，则必须转换掉第一个

字符，如：\\${}。

### 4.1.1 保留字

下面是一些保留字，不能用作标识符：

```
and   eq   gt   true   instanceof
or    ne   le   false  empty
not   lt   ge   null   div   mod
```

### 4.1.2 [] 和 . 运算符

EL 表达式可以返回任何类型。如果某个 EL 表达式的结果对象中有一个属性，那么你就可以利用 [] 或者 . 运算符来访问该属性。[] 和 . 运算符的功能相似；[] 比较常用，但是 . 比较简洁。

要访问某个对象的属性，可以使用以下任意一种形式：

```
${object["propertyName"]}
${object.propertyName}
```

但是，如果 *propertyName* 不是有效的 Java 变量名称，则只能使用 [] 运算符。例如，下面这两个 EL 表达式可以用来访问隐式对象标头中的 HTTP 标头 host：

```
${header["host"]}
${header.host}
```

但是，要访问 accept-language 标头，则只能使用 [] 运算符，因为 accept-language 不是一个合法的 Java 变量名称。如果用 . 运算符来访问它，将会导致抛出一个异常。

如果一个对象的属性碰巧返回另一个带有属性的对象，那么就可以利用 [] 或者 . 来访问第二个对象的属性。例如，隐式对象 pageContext 表示当前 JSP 的 PageContext 对象，它带有 request 属性，表示 HttpServletRequest。HttpServletRequest 又带有 servletPath 属性。下面这些表达式是相等的，都将在 pageContext 中生成 servletPath 属性值：

```
${pageContext["request"]["servletPath"]}
${pageContext.request["servletPath"]}
${pageContext.request.servletPath}
${pageContext["request"].servletPath}
```

要访问 HttpSession，可使用下面这个语法：

```
${pageContext.session}
```

例如，下面这个表达式将会输出 session 标识符：

```
 ${pageContext.session.id}
```

### 4.1.3 运算规则

EL 表达式的运算顺序是从左到右。对于像 *expr-a[expr-b]* 这种表达式，其 EL 表达式的运算规则是：

1. 运算 *expr-a*，得到 *value-a*。
2. 如果 *value-a* 为 null，则返回 null。
3. 运算 *expr-b*，得到 *value-b*。
4. 如果 *value-b* 为 null，则返回 null。
5. 如果 *value-a* 的类型为 `java.util.Map`，就要查看 *value-b* 是否为 Map 中的一个键。如果是，返回 *value-a.get(value-b)*；如果不是，则返回 null。
6. 如果 *value-a* 的类型为 `java.util.List`，或者是一个数组，那么就要：
  - a. 强制 *value-b* 的类型为 `int`。如果强制失败，将抛出一个异常。
  - b. 如果 *value-a.get(value-b)* 抛出一个 `IndexOutOfBoundsException`，或者假设 `Array.get(value-a,value-b)` 抛出一个 `ArrayIndexOutOfBoundsException`，则返回 null。
  - c. 否则，如果 *value-a* 为 `List`，就返回 *value-a.get(value-b)*，如果 *value-a* 为数组，则返回 `Array.get(value-a,value-b)`。
7. 如果 *value-a* 不是 `Map`、`List` 或数组，*value-a* 就必须是一个 JavaBean。在这种情况下，就要强制 *value-b* 为 `String`。如果 *value-b* 是 *value-a* 的一个可读属性，那么将调用该属性的 `getter` 方法，并返回来自 `getter` 方法的值。如果 `getter` 方法抛出异常，则表示该表达式无效。否则，有效。

## 4.2 访问 JavaBean

我们可以利用 . 或 [] 运算符来访问一个 Bean 的属性，其构造如下：

```
 ${beanName["propertyName"]}
 ${beanName.propertyName}
```

例如，要访问 `myBean` 中的 `secret` 属性，要使用下面这个表达式：

```
 ${myBean.secret}
```

如果该属性也是一个带有属性的对象，那么仍然可以利用 . 或者 [] 运算符来访问第二个对象的属性。或者，假如该属性是一个 `Map`、`List` 或者数组，那么就可以利用前一节讲过的规则来访问 `Map` 值、`List` 成员或数组元素。

## 4.3 EL 隐式对象

在一个 JSP 页面中，可以利用 JSP 脚本访问 JSP 隐式对象。但是，在一个无脚本的 JSP 页面中，则不可能访问这些隐式对象。EL 通过提供一组它自己的隐式对象，可以帮助你访问各种对象。EL 隐式对象如表 4-1 所示。

下面将分别讲解这些隐式对象。

表 4-1 EL 隐式对象

对 象	描 述
pageContext	当前 JSP 页面的 javax.servlet.jsp.PageContext
initParam	包含所有 context 初始化参数并以参数名称作为键的 Map
param	包含所有请求参数并以参数名称作为键的 Map。每个键的值就是指定名称的第一个参数值。因此，如果有两个同名的请求参数，将只有第一个参数可以利用 param 对象获取到。要想获取所有同名参数的值，则要使用 params 对象
paramValues	包含所有请求参数并以参数名称作为键的 Map。每个键的值就是包含所有指定名称值的一个字符串数组。如果该参数只有一个值，它仍然会返回一个只有一个元素的数组
header	包含所有请求标头并以标头名称作为键的 Map。每个键的值就是指定标头名称的第一个标头。换句话说，如果某个标头具有多个值，将只返回第一个值。要想获得多值标头，则要使用 headerValues 对象
headerValues	包含所有请求标头并以标头名称作为键的 Map。每个键的值就是包含指定标头名称所有值的一个字符串数组。如果标头只有一个值，将返回只有一个元素的数组
cookie	包含当前请求对象中所有 Cookie 对象的 Map。Cookie 的名称就是 Map 的键，每一个键都映射到一个 Cookie 对象
applicationScope	包含 ServletContext 对象中所有属性并以属性名称作为键的 Map
sessionScope	包含 HttpSession 对象中所有属性并以属性名称作为键的 Map

### 4.3.1 pageContext

pageContext 对象表示当前 JSP 页面的 javax.servlet.jsp.PageContext。它包含所有其他的 JSP 隐式对象，如表 4-2 所示。

表 4-2 JSP 隐式对象

对 象	EL 中的类型
request	javax.servlet.http.HttpServletRequest
response	javax.servlet.http.HttpServletResponse
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
pageContext	javax.servlet.jsp.PageContext

(续)

对    象	EL 中的类型
page	javax.servlet.jsp.HttpJspPage
exception	java.lang.Throwable

例如，可以利用以下任意一个表达式获得当前的 `ServletRequest`:

```
 ${pageContext.request}
 ${pageContext["request"]}
```

请求方法则可以利用以下任意一个表达式获得:

```
 ${pageContext["request"]["method"]}
 ${pageContext["request"].method}
 ${pageContext.request["method"]}
 ${pageContext.request.method}
```

比起其他隐式对象，请求参数更经常被访问到，因此，提供了 `param` 和 `paramValues` 这两个隐式对象。隐式对象 `param` 和 `paramValues` 将分别在下面讨论。

### 4.3.2 initParam

隐式对象 `initParam` 用于获取一个 `context` 参数值。例如，要想获得 `context` 参数 `password`，需使用下面的表达式:

```
 ${initParam.password}
```

或者

```
 ${initParam["password"]}
```

### 4.3.3 param

隐式对象 `param` 用于获取一个请求参数。这个对象表示一个包含所有请求参数的 `Map`。例如，为了获取参数 `userName`，需使用下面任意一个表达式:

```
 ${param.userName}
 ${param["userName"]}
```

### 4.3.4 paramValues

利用隐式对象 `paramValues` 可以获取一个请求参数的多个值。这个对象表示一个包含所有请求参数并以参数名称作为键的 `Map`。每个键的值都是一个字符串数组，其中包含指定参数名称的所有值。如果该参数只有一个值，也仍然返回只有一个元素的一个数组。例如，要获得 `selectedOptions` 参数的第一个和第二个值，可以使用如下表达式:

```

${paramValues.selectedOptions[0]}
${paramValues.selectedOptions[1]}

```

### 4.3.5 header

隐式对象 `header` 表示一个包含所有请求标头的 `Map`。要获取一个标头值，需用该标头名称作为键。例如，要获取 `accept-language` 标头的值，需用下面的表达式：

```

${header["accept-language"]}

```

如果该标头名称是一个有效的 Java 变量名，如 `connection`，那么也可以使用 `.` 运算符：

```

${header.connection}

```

### 4.3.6 headerValues

隐式对象 `headerValues` 表示一个包含所有请求标头并以标头名称作为键的 `Map`。但与 `header` 不同的是，隐式对象 `headerValues` 返回的 `Map` 返回一个字符串数组。例如，要获得 `accept-language` 标头的第一个值，需使用下面这个表达式：

```

${headerValues["accept-language"]}[0]

```

### 4.3.7 cookie

利用隐式对象 `cookie` 可以获取一个 `cookie`。这个对象表示包含当前 `HttpServletRequest` 中所有 `cookie` 的 `Map`。例如，要获取一个名为 `jsessionid` 的 `cookie` 值，需使用下面这个表达式：

```

${cookie.jsessionid.value}

```

要获得 `jsessionid` `cookie` 的路径值，则使用这个表达式：

```

${cookie.jsessionid.path}

```

### 4.3.8 applicationScope、sessionScope、requestScope 及 pageScope

利用隐式对象 `applicationScope` 获得一个 `application` 范围的变量值。例如，如果你有一个 `application` 范围的变量 `myVar`，就可以利用下面这个表达式来访问该属性：

```

${applicationScope.myVar}

```

注意，在 `Servlet/JSP` 编程中，有作用范围的对象是指放在以下这些对象中作为属性的对象：`PageContext`、`ServletRequest`、`HttpSession` 或者 `ServletContext`。隐式对象 `sessionScope`、`requestScope` 及 `pageScope` 与 `applicationScope` 相似。但是其范围分别是 `session`、`request` 及 `page`。

有作用范围的对象也可以用一个没有指定范围的 EL 表达式进行访问。在这种情况下，JSP 容器将会返回在 `PageContext`、`ServletRequest`、`HttpSession` 或者 `ServletContext` 中第一次识别到的指定对象。搜索顺序从最小范围（`PageContext`）开始，到最大范围（`ServletContext`）。例如，下面的表达式将返回任意范围的 `today` 所引用的对象：

```
 ${today}
```

## 4.4 使用其他 EL 运算符

除了 `.` 和 `[]` 运算符外，EL 还提供了几个其他的运算符：算术运算符、关系运算符、逻辑运算符、条件运算符及 `empty` 运算符。使用这些运算符，可以完成各种运算。但是，由于 EL 的目的是便于设计无脚本的 JSP 页面，因此这些 EL 运算符中，除条件运算符之外，其他几个的用处都十分有限。

接下来介绍这些 EL 运算符。

### 4.4.1 算术运算符

算术运算符有 5 个：

- 加法（`+`）
- 减法（`-`）
- 乘法（`*`）
- 除法（`/` 和 `div`）
- 余数 / 模（`%` 和 `mod`）

除法和余数运算符有两种形式，与 XPath 和 ECMPScript 相一致。

注意，EL 表达式的运算顺序是从最高优先级到最低优先级，并且从左到右进行。以下是按递减优先级顺序排列的算术运算符：

**`* / div % mod`**  
**`+ -`**

这表示 `*`、`/`、`div`、`%` 及 `mod` 运算符的优先级是一样的，`+` 与 `-` 优先级一样，但是其优先级比第一组低。因此，表达式：

```
 ${1+2*3}
```

结果是 7，而不是 6。

## 4.4.2 关系运算符

下面是关系运算符列表:

- 等于 (`==` 和 `eq`)
- 不等于 (`!=` 和 `ne`)
- 大于 (`>` 和 `gt`)
- 大于或等于 (`>=` 和 `ge`)
- 小于 (`<` 和 `lt`)
- 小于或等于 (`<=` 和 `le`)

例如，表达式  `${3==4}` 将返回 `false`， `${"b" < "d"}` 将返回 `true`。

## 4.4.3 逻辑运算符

下面是逻辑运算符列表:

- AND (`&&` 和 `and`)
- OR (`||` 和 `or`)
- NOT (`!` 和 `not`)

## 4.4.4 条件运算符

EL 条件运算符具有下列语法:

```
 ${statement? A:B}
```

如果 `statement` 运算结果为 `true`，则这个表达式的结果就是 `A`，否则为 `B`。

例如，利用下面的 EL 表达式可以测试 `HttpSession` 中是否包含 `loggedIn` 属性。如果找到该属性，将显示“`You have logged in`”，否则显示“`You have not logged in`”。

```
 ${(sessionScope.loggedIn==null)? "You have not logged in" :  
      "You have logged in"}
```

## 4.4.5 empty 运算符

`empty` 运算符用来检验一个值是否为 `null` 或者为空。下面举一个使用 `empty` 运算符的例子:

```
 ${empty X}
```

如果 `X` 为 `null`，或者如果 `X` 是一个长度为 0 的字符串，那么该表达式将返回 `true`。如果 `X` 是一个空的 `Map`、空数组或者空集合，它也是返回 `true`，否则返回 `false`。

## 4.5 使用 EL

例如，范例应用程序 app04a 中的 JSP 页面，就是利用 EL 来打印处于另一个 JavaBean（代码清单 4-2 中的 Employee）中的一个 JavaBean（代码清单 4-1 中的 Address）属性、Map 的内容、HTTP 标头及 session 标识符的。代码清单 4-3 中的 EmployeeServlet servlet 创建了必要的对象，并将它们放在 ServletRequest 中。之后，Servlet 利用一个 RequestDispatcher 跳转到 employee.jsp 页面。

代码清单 4-1 Address 类

---

```

package app04a.model;
public class Address {
    private String streetName;
    private String streetNumber;
    private String city;
    private String state;
    private String zipCode;
    private String country;

    public String getStreetName() {
        return streetName;
    }
    public void setStreetName(String streetName) {
        this.streetName = streetName;
    }
    public String getStreetNumber() {
        return streetNumber;
    }
    public void setStreetNumber(String streetNumber) {
        this.streetNumber = streetNumber;
    }
    public String getCity() {
        return city;
    }
    public void setCity(String city) {
        this.city = city;
    }
    public String getState() {
        return state;
    }
    public void setState(String state) {
        this.state = state;
    }
    public String getZipCode() {
        return zipCode;
    }
    public void setZipCode(String zipCode) {
        this.zipCode = zipCode;
    }
}
```

```

    }
    public String getCountry() {
        return country;
    }
    public void setCountry(String country) {
        this.country = country;
    }
}

```

---

代码清单 4-2 Employee 类

```

package app04a.model;
public class Employee {
    private int id;
    private String name;
    private Address address;

    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public Address getAddress() {
        return address;
    }
    public void setAddress(Address address) {
        this.address = address;
    }
}

```

---

代码清单 4-3 EmployeeServlet 类

```

package app04a.servlet;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.Dispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import app04a.model.Address;
import app04a.model.Employee;

```

```

@WebServlet(urlPatterns = {" /employee"})
public class EmployeeServlet extends HttpServlet {
    private static final int serialVersionUID = -5392874;
    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        Address address = new Address();
        address.setStreetName("Rue D'Anjou");
        address.setStreetNumber("5090B");
        address.setCity("Brossard");
        address.setState("Quebec");
        address.setZipCode("A1A B2B");
        address.setCountry("Canada");

        Employee employee = new Employee();
        employee.setId(1099);
        employee.setName("Charles Unjeye");
        employee.setAddress(address);
        request.setAttribute("employee", employee);
        Map<String, String> capitals = new HashMap<String,
            String>();
        capitals.put("China", "Beijing");
        capitals.put("Austria", "Vienna");
        capitals.put("Australia", "Canberra");
        capitals.put("Canada", "Ottawa");
        request.setAttribute("capitals", capitals);

        RequestDispatcher rd =
            request.getRequestDispatcher("/employee.jsp");
        rd.forward(request, response);
    }
}

```

代码清单 4-4 employee.jsp 页面

---

```

<html>
<head>
<title>Employee</title>
</head>
<body>
accept-language: ${header['accept-language']}
<br/>
session id: ${pageContext.session.id}
<br/>
employee: ${requestScope.employee.name}, ${employee.address.city}
<br/>
capital: ${capitals["Canada"]}
</body>
</html>

```

---

注意，app04a 中用一个 Servlet 包含 Java 代码，并用一个 JSP 页面显示 JavaBean 属性以及其他值，这正符合第 10 章中建议的现代 Web 应用程序设计方法。

要特别注意 JSP 页面中的 EL 表达式。request 范围的对象 `employee` 可以通过隐式对象 `requestScope` 访问，也可以不通过它进行访问。

```
employee: ${requestScope.employee.name}, ${employee.address.city}
```

利用下面这个 URL 调用 EmployeeServlet 可以对应用程序进行测试：

```
http://localhost:8080/app04a/employee
```

## 4.6 在 JSP 2.0 及更高版本中配置 EL

有了 EL、JavaBean 和定制标签，现在可以编写无脚本的 JSP 页面了。在 JSP 2.0 及更高的版本中，甚至可以选择关闭所有 JSP 页面中的脚本。现在，软件架构师可以强制编写无脚本的 JSP 页面了。

另一方面，在某些情况下，你可能需要关闭应用程序中的 EL。例如，假设你正在使用一个与 JSP 2.0 兼容的容器，但又尚未准备好升级到 JSP 2.0，此时，就要关闭 EL 表达式的运算。

本节讨论如何强制编写无脚本的 JSP 页面，以及如何在 JSP 2.0 及更高的版本中关闭 EL。

### 4.6.1 实现无脚本的 JSP 页面

要关闭 JSP 页面中的脚本元素，必须利用带有两个子元素（`url-pattern` 和 `scripting-invalid`）的 `jsp-property-group` 元素。`url-pattern` 元素定义要关闭脚本的 URL 模式。下面示范如何在应用程序中关闭所有 JSP 页面的脚本：

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <scripting-invalid>true</scripting-invalid>
    </jsp-property-group>
</jsp-config>
```

---

**提示** 部署描述符中只能有一个 `jsp-config` 元素。如果已经指定一个 `jsp-property-group` 关闭 EL，就必须编写 `jsp-property-group`，用它关闭同一个 `jsp-config` 元素下的脚本。

---

### 4.6.2 关闭 EL 运算

在某些情况下，如当你需要在一个 JSP 2.0 或者更新版本的容器中部署 JSP 1.2 应用程序

时，就会需要关闭 JSP 页面中的 EL 运算。这样，遇到 EL 结构时，就不会将它当成一个 EL 表达式进行运算了。有两种方式可以关闭 JSP 中的 EL 运算。

第一，可以将 page 指令的 isELIgnored 属性值设为 true，如：

```
<%@ page isELIgnored="true" %>
```

isELIgnored 属性值默认是 false。如果你想关闭一个或者多个 JSP 页面中的 EL 运算，建议你还是使用 isELIgnored 属性。

第二，可以在部署描述符中使用 `jsp-property-group` 元素。`jsp-property-group` 元素是 `jsp-config` 元素的一个子元素。利用 `jsp-property-group`，将某些设置应用到应用程序的一组 JSP 页面中。

为了利用 `jsp-property-group` 元素关闭 EL 运算，必须有两个子元素：`url-pattern` 和 `el-ignored`。`url-pattern` 元素定义要关闭 EL 的 URL 模式。`el-ignored` 元素必须设为 true。

举个例子。下面示范如何关闭 JSP 页面 `noEl.jsp` 的 EL 运算。

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>/noEl.jsp</url-pattern>
        <el-ignored>true</el-ignored>
    </jsp-property-group>
</jsp-config>
```

还可以通过给 `url-pattern` 元素赋值为 `*.jsp`，关闭一个应用程序中所有 JSP 页面的 EL 运算，如：

```
<jsp-config>
    <jsp-property-group>
        <url-pattern>*.jsp</url-pattern>
        <el-ignored>true</el-ignored>
    </jsp-property-group>
</jsp-config>
```

如果一个 JSP 页面中 page 指令的 isELIgnored 属性值设为 true，或者它的 URL 符合 `jsp-property-group` 元素中的模式，并且其子元素 `el-ignored` 设为 true，那么这个 JSP 页面的 EL 运算就会被关闭。例如，如果将 JSP 页面中 page 指令的 isELIgnored 属性值设为 false，但其 URL 符合在部署描述符中必须关闭 EL 运算的 JSP 页面模式，那么该页面的 EL 运算就会被关闭。

此外，如果使用与 Servlet 2.3 或更低版本兼容的部署描述符，那么即使你现在用的是 JSP 2.0 或者更新版本的容器，其 EL 运算也将已经默认为关闭。

## 4.7 小结

EL 是 JSP 2.0 及更新版本中最重要的特性之一。它可以帮助你编写出更简短、更高效的 JSP 页面，以及帮助你编写无脚本的页面。在本章中，学习了如何利用 EL 访问 JavaBean 和隐式对象。此外，还学习了如何使用 EL 运算符。在本章的最后，还学习了如何在 JSP 2.0 及更新版本的容器中利用与 EL 有关的应用程序设置。

# 第 5 章 JSTL

JavaServer Pages Standard Tag Library (JSTL) 是一个定制标签类库的集合，用于解决一些常见的问题，例如迭代一个映射或者集合、条件测试、XML 处理，甚至数据库访问和数据操作等。

本章主要讨论 JSTL 中最重要的标签，尤其是访问限域对象（Scoped Variable）、迭代集合及格式化数字和日期的那些标签。如果你有兴趣了解更多的相关信息，则可以在 JSTL 规范文档中找到所有 JSTL 标签的详细讲解。

## 5.1 下载 JSTL

JSTL 目前的版本为 1.2，由 Java Community Process (JCP，详情查看 [www.jcp.org](http://www.jcp.org)) 下的 JSR-52 专家组定义。从 [java.net](http://java.net) 网站上可以下载到其实现：

<http://jstl.java.net>

你需要下载两套软件：JSTL API 和 JSTL 实现。JSTL API 中包含 `javax.servlet.jsp.jstl` 包，由 JSTL 规范中定义的类型组成。JSTL 实现则包含了相关的实现类。必须将两个 jar 文件都复制到使用了 JSTL 的每个应用程序的 WEB-INF/lib 目录下。

## 5.2 JSTL 类库

JSTL 是指标准标签类库，但它是通过多个标签类库来显露其动作指令（标签指令）的。JSTL 1.2 中的标签可以被归纳成五大类，如表 5-1 所示。

表 5-1 JSTL 标签类库

类 别	下 属 功 能	U R I	前 缀
Core	变量支持	<a href="http://java.sun.com/jsp/jstl/core">http://java.sun.com/jsp/jstl/core</a>	c
	流向控制		
	URL 管理		
	杂项		
XML	Core	<a href="http://java.sun.com/jsp/jstl/xml">http://java.sun.com/jsp/jstl/xml</a>	x
	流向控制		
	转换		

(续)

类 别	下 属 功 能	U R I	前 缀
I18n	语 言 环 境	http://java.sun.com/jsp/jstl/fmt	fmt
	消 息 格 式 化		
	数 字 和 期 期 格 式 化		
数 据 库	S Q L	http://java.sun.com/jsp/jstl/sql	s q l
功 能	集 合 长 度	http://java.sun.com/jsp/jstl/functions	fn
	字 符 串 操 作		

为了在 JSP 页面中使用 JSTL 类库，必须以下列格式使用 taglib 指令：

```
<%@ taglib uri="uri" prefix="prefix" %>
```

例如，为了使用 Core 类库，必须在 JSP 页面开头处这样声明：

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

这个前缀 `prefix` 可以是任意内容。但是，遵循规范可以使团队中由不同人员编写的代码更加相似。正因为如此，才建议使用事先设计好的前缀。

---

**提示** 本章讨论的每一个标签都有对应的小节，每个标签的属性都会列成一张表格。属性名称后面加星号 (\*) 的表示该属性是必需的，加号 (+) 的表示该属性的 `rteprvalue` 元素为 `true`，这意味着该属性值可以为静态字符串或者动态值（Java 表达式、EL 表达式，或者用 `<jsp:attribute>` 设置的值）。`rteprvalue` 值为 `false` 表示该属性的值只能为静态字符串。

---



---

**提示** JSTL 标签的主体内容可以为 `empty`、`JSP` 或者 `tagdependent`<sup>⊖</sup>。

---

## 5.3 通用动作指令

接下来要讨论 Core 类库中用来操作限域变量的 3 个通用动作指令：`out`、`set` 和 `remove`。

### 5.3.1 `out` 标签

`out` 标签对表达式进行运算，并将结果输出到当前的 `JspWriter`。`out` 的语法有两种形式，即有主体内容和没有主体内容：

---

⊖ `empty`——空标记，即起始标记和结束标记之间内容为空。JSP——接受所有 JSP 语法，如定制的或内部的 Tag、Scripts、静态 HTML、脚本元素、JSP 指令和动作指令。`tagdependent`——标签体内容直接被写入 `BodyContent`，由自定义标签类来进行处理，而不被 JSP 容器解析。——译者注

```
<c:out value="value" [escapeXml="{true|false}"]
       [default="defaultValue"]/>
<c:out value="value" [escapeXml="{true|false}"]>
    default value
</c:out>
```

**提示** 在标签的语法中，[] 表示是可选的属性。加了下划线的那个值，则表示那是默认值。

**out** 的主体内容是 JSP。该标签的属性列表如表 5-2 所示。

表 5-2 out 标签的属性

属性	类型	描述
value*+	Object	要运算的表达式
escapeXml+	boolean	表明结果中的字符 <、>、&、' 和 " 将被转换成相应的字符实体代码，如 < 转换为 &lt;；等
default+	Object	默认值

例如，下面的 **out** 标签将输出限域变量 **x** 的值：

```
<c:out value="${x}" />
```

在默认情况下，**out** 标签会将特殊字符 <、>、'、" 和 & 分别转换成它们对应的字符实体代码 &lt;、&gt;、&#039;、&#034; 和 &amp;；

在 JSP 2.0 之前，**out** 标签是输出限域对象值的最容易方法。在 JSP 2.0 或者更新的版本中，除非你需要将某个值进行字符转换，否则可以放心地使用 EL 表达式：

```
${x}
```

**警告** 如果某个字符串中包含一个或多个特殊字符串，其结果字符不能进行转换，那么它的值将无法在浏览器上正确地显示出来。此外，未经转换的特殊字符也会使网站容易受到跨网站的脚本攻击，例如有人往网站发一个会自动执行的 JavaScript 函数 / 表达式等。

**out** 中的 **default** 属性可以设置一个默认值，当赋予其 **value** 属性的 EL 表达式返回 **null** 时，就会显示该默认值。**default** 属性可被赋予动态值。如果这个动态值返回 **null**，**out** 标签就会显示一个空字符串。

例如，在下面的 **out** 标签中，如果在 **HttpSession** 中没有找到 **myVar** 变量，那么就会显示范围为 **application** 的 **myVar** 变量。如果连后者也找不到，就会输出一个空字符串。

```
<c:out value="${sessionScope.myVar}"
       default="${applicationScope.myVar}" />
```

### 5.3.2 set 标签

利用 **set** 标签可以完成以下工作：

1. 创建一个字符串和引用该字符串的一个限域变量。
2. 创建一个限域变量，并引用已经存在的某一个限域变量。
3. 设置限域对象的属性。

如果利用 **set** 创建限域变量，那么处于该标签之后的整个 JSP 页面将都可以使用该变量。

**set** 标签的语法有 4 种形式。第一种形式用来创建一个限域变量，在该变量的 **value** 属性中可以指定要创建的字符串，或者现有的限域对象。

```
<c:set value="value" var="varName"
       [scope="{page|request|session|application}"]/>
```

这里的 **scope** 属性指定了限域变量的范围。

例如，下面的 **set** 标签创建了字符串 “The wisest fool”，并将它赋给新建的变量 **foo**：

```
<c:set var="foo" value="The wisest fool"/>
```

下面的 **set** 标签创建了一个限域变量 **job**，它引用 **request** 范围的对象 **position**。变量 **job** 的范围为 **page**：

```
<c:set var="job" value="${requestScope.position}" scope="page"/>
```

**提示** 最后这个例子可能有点费解，因为它创建了一个范围为 **page** 的变量，同时它又引用一个范围为 **request** 的对象。如果你记住限域对象本身并不是真的处在 **HttpServletRequest** “里面”，而是有一个引用（名为 **position**）在引用该对象，那么应该就容易理解了。最后这个例子中有了 **set** 标签，就只要再创建一个也引用该对象的限域变量（**job**）即可。

第二种形式与第一种相似，只不过要创建的字符串或者要引用的限域对象是作为主体内容进行传递的：

```
<c:set var="varName" [scope="{page|request|session|application}"]>
    body content
</c:set>
```

第二种形式允许在主体内容中有 JSP 代码。

第三种形式是设置限域对象的属性值。**target** 属性指定限域对象，**property** 属性指定限域对象的属性。要赋给该属性的值通过 **value** 属性进行设置：

```
<c:set target="target" property="propertyName" value="value"/>
```

例如，下面的 **set** 标签是将字符串 “Tokyo” 赋给限域对象 **address** 的 **city** 属性：

```
<c:set target="${address}" property="city" value="Tokyo"/>
```

注意，必须在 **target** 属性中用一个 EL 表达式来引用限域对象。

第四种形式与第三种相似，但它的值是作为主体内容传递的：

```
<c:set target="target" property="propertyName">
    body content
</c:set>
```

例如，下面的 **set** 标签是将字符串 “Beijing” 赋给限域对象 **address** 的 **city** 属性。

```
<c:set target="${address}" property="city">Beijing</c:set>
```

**set** 标签的属性列表如表 5-3 所示。

表 5-3 **set** 标签的属性

属性	类型	描述
value+	Object	要创建的字符串，或要引用的限域对象，或新的属性值
var	String	要创建的限域变量
scope	String	新建限域对象的范围
target+	Object	其属性要赋新值的限域对象；必须是 JavaBeans 实例或者 java.util.Map 对象
property+	String	要赋新值的属性名称

### 5.3.3 remove 标签

利用 **remove** 标签删除限域变量，其语法如下：

```
<c:remove var="varName"
    [scope="{page|request|session|application}"]/>
```

注意，限域变量引用的对象并没有被删除。因此，如果另一个限域对象也在引用这个对象，那么仍然可以通过另外这个限域对象来访问该对象。

**remove** 标签的属性列表如表 5-4 所示。

表 5-4 **remove** 标签的属性

属性	类型	描述
var	String	要删除的限域变量名称
scope	String	要删除的限域变量范围

举个例子。下面的 **remove** 标签删除了范围为 **page** 的 **job** 变量：

```
<c:remove var="job" scope="page"/>
```

## 5.4 条件式动作指令

条件式动作指令用于处理页面的输出结果依赖于某些输入值的情况，在 Java 中是利用 if、if...else 和 switch 语句来进行处理的。

在 JSTL 中有 4 个标签可以执行条件式动作指令：if、choose、when 和 otherwise。接下来逐个讨论。

### 5.4.1 if 标签

if 标签先对某个条件进行测试，如果该条件运算结果为 true，则处理它的主体内容。测试结果保存在一个 Boolean 对象中，并创建一个限域变量来引用 Boolean 对象。可以利用 var 属性设置限域变量名，利用 scope 属性来指定其作用范围。

if 的语法有两种形式，第一种形式没有主体内容：

```
<c:if test="testCondition" var="varName"
      [scope="{page|request|session|application}"]/>
```

在这种情况下，var 指定的限域变量一般由同一个 JSP 页面中更后面的另一个标签的测试所决定。

第二种形式有主体内容：

```
<c:if test="testCondition [var="varName"]
      [scope="{page|request|session|application}"]>
    body content
</c:if>
```

主体内容为 JSP，如果测试条件结果为 true，它就会得到处理。if 标签的属性列表如表 5-5 所示。

表 5-5 if 标签的属性

属性	类型	描述
test+	Boolean	决定是否应该处理某些现有主体内容的测试条件
var	String	引用测试条件值的限域变量名称；var 的类型为 Boolean
scope	String	用 var 设置的限域变量的范围

例如，假设有一个 request 参数名为 user，它的值为 ken；并且有一个 request 参数名为 password，它的值为 blackcomb，那么下面的 if 标签将会显示“You logged in successfully”：

```
<c:if test="${param.user=='ken' && param.password=='blackcomb'}">
  You logged in successfully.
</c:if>
```

为了模拟 else 的情景，需使用两个 if 标签，并用两个相反的条件。例如，如果 user

和 `password` 参数分别为 `ken` 和 `blackcomb`，那么以下代码片段将会显示“`You logged in successfully`”，否则显示“`Login failed`”。

```
<c:if test="${param.user=='ken' && param.password=='blackcomb'}">
    You logged in successfully.
</c:if>
<c:if test="${!(param.user=='ken' && param.password=='blackcomb')}">
    Login failed.
</c:if>
```

以下 `if` 标签测试 `user` 和 `password` 参数是否分别为 `ken` 和 `blackcomb`，并将结果保存在范围为 `page` 的 `loggedIn` 变量中。之后，使用一个 EL 表达式，如果 `loggedIn` 变量为 `true`，就显示“`You logged in successfully`”，如果 `loggedIn` 变量为 `false`，则显示“`Login failed`”。

```
<c:if var="loggedIn"
       test="${param.user=='ken' && param.password=='blackcomb'}"/>
...
${(loggedIn) ? "You logged in successfully" : "Login failed"}
```

## 5.4.2 choose、when 和 otherwise 标签

`choose` 和 `when` 标签的作用与 Java 中的 `switch` 和 `case` 关键字相似，也就是说，它们为互相排斥的条件式执行提供相关内容。`choose` 标签内部必须嵌有一个或多个 `when` 标签，每个 `when` 标签代表可以进行运算和处理的一种情况。`otherwise` 标签用于默认的条件代码块，如果所有 `when` 标签的测试条件运算结果都不为 `true`，就会执行该代码块。如果有 `otherwise` 标签，它必须放在最后一个 `when` 标签之后。

`choose` 和 `otherwise` 标签没有属性。`when` 标签则必须用 `test` 属性设定一个条件，用于确定是否处理主体内容。

举个例子。以下代码测试 `status` 参数的值。如果 `status` 的值为 `full`，就会显示“`You are a full member`”；如果值为 `student`，就会显示“`You are a student member`”；如果没有 `status` 参数，或者它的值既不是 `full`，也不是 `student`，那么这段代码将不显示任何内容。

```
<c:choose>
    <c:when test="${param.status=='full'}">
        You are a full member
    </c:when>
    <c:when test="${param.status=='student'}">
        You are a student member
    </c:when>
</c:choose>
```

下面这个例子与前一个相似，只是它使用的是 `otherwise` 标签，如果 `status` 参数不存在，或者它的值不为 `full` 或 `student`，那么它将显示“`Please register`”：

```
<c:choose>
    <c:when test="${param.status=='full'}">
        You are a full member
    </c:when>
    <c:when test="${param.status=='student'}">
        You are a student member
    </c:when>
    <c:otherwise>
        Please register
    </c:otherwise>
</c:choose>
```

## 5.5 iterator 动作指令

当需要迭代多次，或者需要迭代一个对象集合时，iterator 动作指令就非常好用。JSTL 中提供了两个可以执行 iterator 动作指令的标签：`forEach` 和 `forTokens`，两者都将在下面讨论。

### 5.5.1 forEach 标签

`forEach` 是将一个主体内容迭代多次，或者迭代一个对象集合。可以迭代的对象包括所有的 `java.util.Collection` 和 `java.util.Map` 接口的实现，以及对象或者基本类型的数组。它还可以迭代 `java.util.Iterator` 和 `java.util Enumeration`，但不能在多个动作指令中使用 `Iterator` 或者 `Enumeration`，因为 `Iterator` 或 `Enumeration` 都不能重置（`reset`）。

`forEach` 的语法有两种形式。第一种是将 body 内容重复一定的次数：

```
<c:forEach [var="varName"] begin="begin" end="end" step="step">
    body content
</c:forEach>
```

第二种形式用于迭代一个对象集合：

```
<c:forEach items="collection" [var="varName"]
    [varStatus="varStatusName"] [begin="begin"] [end="end"]
    [step="step"]>
    body content
</c:forEach>
```

主体内容为 JSP。`forEach` 标签的属性如表 5-6 所示。

例如，下面的 `forEach` 标签将显示“1, 2, 3, 4, 5”：

```
<c:forEach var="x" begin="1" end="5">
    <c:out value="${x}" />,
</c:forEach>
```

下面的 `forEach` 标签迭代限域变量 `address` 的 `phones` 属性。

表 5-6 forEach 标签的属性

属性	类型	描述
var	String	引用当前迭代项目的限域变量名称
items+	支持的任何类型	要迭代的对象集合
varStatus	String	保存迭代状态的限域变量名称，它的值类型为 javax.servlet.jsp.jstl.core.LoopTagStatus
begin+	int	如果指定了 items，那么迭代将从处于指定索引的项开始，该集合中的第一个项索引为 0。如果没有指定 items，迭代将从该值设定的索引开始。如有指定，begin 的值必须大于或者等于 0
end+	int	如果指定了 items，那么迭代将结束于处于指定索引的项（含）；如果没有指定 items，那么当索引到达指定值时，迭代结束
step+	int	步长，迭代会从集合的第一个 step 项开始，根据 step 步长逐个地进行。如果有 step 属性，那么它的值必须大于或者等于 1

```
<c:forEach var="phone" items="${address.phones}">
    ${phone}<br/>
</c:forEach>
```

对于每一次迭代，**forEach** 标签都会创建一个限域变量，其名称通过 **var** 属性设置。在上述例子中，限域变量命名为 **phone**。**forEach** 标签中的 EL 表达式用来显示 **phone** 的值。限域变量只能从打开和闭合的 **forEach** 标签内部进行访问，并且只能在闭合的 **forEach** 标签之前删除。

**forEach** 标签有一个 **varStatus** 变量，它的类型为 **javax.servlet.jsp.jstl.core.LoopTagStatus**。**LoopTagStatus** 接口利用 **count** 属性返回当前迭代的次数。第一次迭代，**status.count** 的值为 1；第二次迭代，其值为 2；依此类推。通过测试 **status.count%2** 的余数，可以得知该标签正在处理的是一个奇数还是偶数的元素。

举个例子。请看 **app05a** 应用程序中的 **BooksServlet** Servlet 和 **books.jsp** 页面。代码清单 5-1 中的 **BooksServlet** 类在其 **doGet** 方法中创建了三个 **Book** 对象，并将 **Book** 对象放在一个 **List** 中，之后将这个列表保存为一个 **ServletRequest** 属性。**Book** 类如代码清单 5-2 所示。在 **doGet** 方法的结尾处，Servlet 跳转到 **books.jsp** 页面，利用 **forEach** 标签迭代 **book** 集合。

代码清单 5-1 BooksServlet 类

---

```
package app05a.servlet;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import app05a.model.Book;
```

```

@WebServlet(urlPatterns = {" /books"})
public class BooksServlet extends HttpServlet {
    private static final int serialVersionUID = -234237;
    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) throws ServletException,
                      IOException {
        List<Book> books = new ArrayList<Book>();
        Book book1 = new Book("978-0980839616",
                              "Java 7: A Beginner's Tutorial", 45.00);
        Book book2 = new Book("978-0980331608",
                              "Struts 2 Design and Programming: A Tutorial",
                              49.95);
        Book book3 = new Book("978-0975212820",
                              "Dimensional Data Warehousing with MySQL: A "
                              + "Tutorial", 39.95);
        books.add(book1);
        books.add(book2);
        books.add(book3);
        request.setAttribute("books", books);
        RequestDispatcher rd =
            request.getRequestDispatcher("/books.jsp");
        rd.forward(request, response);
    }
}

```

---

#### 代码清单 5-2 Book 类

---

```

package app05a.model;
public class Book {
    private String isbn;
    private String title;
    private double price;

    public Book(String isbn, String title, double price) {
        this.isbn = isbn;
        this.title = title;
        this.price = price;
    }

    public String getIsbn() {
        return isbn;
    }
    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    public String getTitle() {
        return title;
    }
    public void setTitle(String title) {
        this.title = title;
    }
}

```

```

    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }
}

```

---

代码清单 5-3 books.jsp 页面

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Book List</title>
<style>
table, tr, td {
    border: 1px solid brown;
}
</style>
</head>
<body>
Books in Simple Table
<table>
    <tr>
        <td>ISBN</td>
        <td>Title</td>
    </tr>
    <c:forEach items="${requestScope.books}" var="book">
    <tr>
        <td>${book.isbn}</td>
        <td>${book.title}</td>
    </tr>
    </c:forEach>
</table>
<br/>
Books in Styled Table
<table>
    <tr style="background:#ababff">
        <td>ISBN</td>
        <td>Title</td>
    </tr>
    <c:forEach items="${requestScope.books}" var="book"
        varStatus="status">
        <c:if test="${status.count%2 == 0}">
            <tr style="background:#eeeeff">
        </c:if>
        <c:if test="${status.count%2 != 0}">
            <tr style="background:#dedeff">
        </c:if>
        <td>${book.isbn}</td>
    
```

```

        <td>${book.title}</td>
    </tr>
    </c:forEach>
</table>

<br/>
ISBNs only:
<c:forEach items="${requestScope.books}" var="book"
    varStatus="status">
    ${book.isbn}<c:if test="${!status.last}">,</c:if>
</c:forEach>
</body>
</html>

```

---

注意，books.jsp 页面会将那些书籍显示三次，第一次是利用 **forEach**，不需要 **varStatus** 属性。

```

<table>
    <tr>
        <td>ISBN</td>
        <td>Title</td>
    </tr>
    <c:forEach items="${requestScope.books}" var="book">
        <tr>
            <td>${book.isbn}</td>
            <td>${book.title}</td>
        </tr>
    </c:forEach>
</table>

```

第二次是利用 **forEach** 加 **varStatus** 属性实现显示，根据行数为单行或双行，在表格中显示不同的颜色：

```

<table>
    <tr style="background:#ababff">
        <td>ISBN</td>
        <td>Title</td>
    </tr>
    <c:forEach items="${requestScope.books}" var="book"
        varStatus="status">
        <c:if test="${status.count%2 == 0}">
            <tr style="background:#eeeeff">
        </c:if>
        <c:if test="${status.count%2 != 0}">
            <tr style="background:#dedeff">
        </c:if>
        <td>${book.isbn}</td>
        <td>${book.title}</td>
    </tr>
    </c:forEach>
</table>

```

第三次是利用 `forEach` 显示 ISBN，以逗号分隔。使用 `status.last` 是为了确保到了最后一个元素之后不会再显示逗号：

```
<c:forEach items="${requestScope.books}" var="book"
    varStatus="status">
    ${book.isbn}<c:if test="${!status.last}">,</c:if>
</c:forEach>
```

利用下面这个 URL 可以对上述例子进行测试：

<http://localhost:8080/app05/books>

其结果应该如图 5-1 所示。

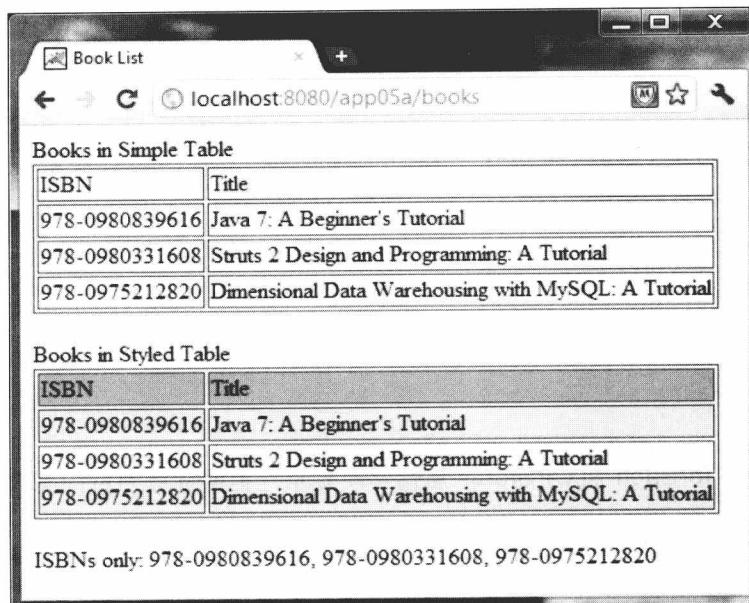


图 5-1 用 `forEach` 显示 List

还可以利用 `forEach` 迭代一个 Map，即分别利用 `key` 和 `value` 属性引用一个 Map 键和一个 Map 值。迭代 Map 的伪代码如下：

```
<c:forEach var="mapItem" items="map">
    ${mapItem.key} : ${mapItem.value}
</c:forEach>
```

下一个例子将示范如何利用 `forEach` 来处理 Map。代码清单 5-4 中的 `BigCitiesServlet` 实例化两个 Map，并给它们填入键 / 值对。第一个 Map 中的每个元素都是一个 `String/String` 对，第二个 Map 中的每个元素都是一个 `String/String[]` 对。

### 代码清单 5-4 BigCitiesServlet 类

```

package app05a.servlet;
import java.io.IOException;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = {"bigCities"})
public class BigCitiesServlet extends HttpServlet {
    private static final int serialVersionUID = 112233;
    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) throws ServletException,
    IOException {
        Map<String, String> capitals =
            new HashMap<String, String>();
        capitals.put("Indonesia", "Jakarta");
        capitals.put("Malaysia", "Kuala Lumpur");
        capitals.put("Thailand", "Bangkok");
        request.setAttribute("capitals", capitals);

        Map<String, String[]> bigCities =
            new HashMap<String, String[]>();
        bigCities.put("Australia", new String[] {"Sydney",
                                              "Melbourne", "Perth"});
        bigCities.put("New Zealand", new String[] {"Auckland",
                                                "Christchurch", "Wellington"});
        bigCities.put("Indonesia", new String[] {"Jakarta",
                                              "Surabaya", "Medan"});

        request.setAttribute("capitals", capitals);
        request.setAttribute("bigCities", bigCities);
        RequestDispatcher rd =
            request.getRequestDispatcher("/bigCities.jsp");
        rd.forward(request, response);
    }
}

```

在 `doGet` 方法的结尾处，Servlet 跳转到 `bigCities.jsp` 页面，它利用 `forEach` 迭代 `Map`。`bigCities.jsp` 如代码清单 5-5 所示。

## 代码清单 5-5 bigCities.jsp 页面

---

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Big Cities</title>
<style>
table, tr, td {
    border: 1px solid #aaeee7;
    padding: 3px;
}
</style>
</head>
<body>
Capitals
<table>
    <tr style="background:#448755;color:white;font-weight:bold">
        <td>Country</td>
        <td>Capital</td>
    </tr>
    <c:forEach items="${requestScope.capitals}" var="mapItem">
        <tr>
            <td>${mapItem.key}</td>
            <td>${mapItem.value}</td>
        </tr>
    </c:forEach>
</table>
<br/>
Big Cities
<table>
    <tr style="background:#448755;color:white;font-weight:bold">
        <td>Country</td>
        <td>Cities</td>
    </tr>
    <c:forEach items="${requestScope.bigCities}" var="mapItem">
        <tr>
            <td>${mapItem.key}</td>
            <td>
                <c:forEach items="${mapItem.value}" var="city"
                    varStatus="status">
                    ${city}<c:if test="${!status.last}">,</c:if>
                </c:forEach>
            </td>
        </tr>
    </c:forEach>
</table>
</body>
</html>

```

---

第二个 `forEach` 特别重要，它里面还嵌套了另一个 `forEach`:

```
<c:forEach items="${requestScope.bigCities}" var="mapItem">
    <c:forEach items="${mapItem.value}" var="city"
        varStatus="status">
        ${city}<c:if test="${!status.last}">,</c:if>
    </c:forEach>
</c:forEach>
```

这里的第二个 `forEach` 迭代 Map 的元素值，它是一个 String 数组。

在浏览器中打开下面这个地址，便可以对上述例子进行测试：

<http://localhost:8080/app05a/bigCities>

你的浏览器应该会用 HTML 表格显示一些国家和首都及大城市，如图 5-2 所示。

The screenshot shows a browser window titled "Big Cities". The address bar displays "localhost:8080/app05a/bigCities". The page content consists of two tables:

Capitals	
Country	Capital
Thailand	Bangkok
Malaysia	Kuala Lumpur
Indonesia	Jakarta

Big Cities	
Country	Cities
Indonesia	Jakarta, Surabaya, Medan
Australia	Sydney, Melbourne, Perth
New Zealand	Auckland, Christchurch, Wellington

图 5-2 `forEach` 和 Map

### 5.5.2 `forTokens` 标签

`forTokens` 标签可用于迭代以特定分界符分隔的 token，这个动作指令的语法如下：

```
<c:forTokens items="stringOfTokens" delims="delimiters"
    [var="varName"] [varStatus="varStatusName"]
    [begin="begin"] [end="end"] [step="step"]
>
    body content
</c:forTokens>
```

主体内容为 JSP。`forTokens` 标签的属性列表如表 5-7 所示。

表 5-7 forTokens 标签的属性

属性	类 型	描 述
var	String	引用当前迭代项的限域变量名称
items+	String	要迭代的 token 字符串
varStatus	String	保存迭代状态的限域变量名称，它的值类型为 javax.servlet.jsp.jstl.core. LoopTagStatus
begin+	int	迭代的起始索引，这里的索引是从 0 开始的。如果有 begin 属性，那么它的值必须大于或者等于 0
end+	int	迭代的结束索引，这里的索引是从 0 开始的
step+	int	迭代会从集合的第一个 step 项开始，逐个地进行。如果有 step 属性，那么它的值必须大于或者等于 1
delims+	String	一组分界符

下面是一个 forTokens 范例：

```
<c:forTokens var="item" items="Argentina,Brazil,Chile" delims=",">
    <c:out value="${item}" /><br/>
</c:forTokens>
```

当它放在 JSP 中时，上述 forTokens 将产生以下结果：

```
Argentina
Brazil
Chile
```

## 5.6 格式化动作指令

JSTL 提供了格式化和解析数字和日期的标签，有：formatNumber、formatDate、timeZone、setTimeZone、parseNumber 及 parseDate。接下来讨论这些标签。

### 5.6.1 formatNumber 标签

formatNumber 标签用来格式化数字。这个标签允许你根据自己的需要灵活地利用它的各种属性进行格式化。formatNumber 的语法有两种形式。第一种没有主体内容：

```
<fmt:formatNumber value="numericValue"
    [type="{number|currency|percent}"]
    [pattern="customPattern"]
    [currencyCode="currencyCode"]
    [currencySymbol="currencySymbol"]
    [groupingUsed="{true|false}"]
    [maxIntegerDigits="maxIntegerDigits"]
    [minIntegerDigits="minIntegerDigits"]
    [maxFractionDigits="maxFractionDigits"]
    [minFractionDigits="minFractionDigits"]
    [var="varName"]
```

```
[scope="{page|request|session|application}"]  
/>
```

第二种形式中有主体内容：

```
<fmt:formatNumber [type="{number|currency|percent}"]  
[pattern="customPattern"]  
[currencyCode="currencyCode"]  
[currencySymbol="currencySymbol"]  
[groupingUsed="{true|false}"]  
[maxIntegerDigits="maxIntegerDigits"]  
[minIntegerDigits="minIntegerDigits"]  
[maxFractionDigits="maxFractionDigits"]  
[minFractionDigits="minFractionDigits"]  
[var="varName"]  
[scope="{page|request|session|application}"]>  
    numeric value to be formatted  
</fmt:formatNumber>
```

主体内容为 JSP。formatNumber 标签的属性如表 5-8 所示。

表 5-8 formatNumber 标签的属性

属性	类型	描述
value+	String 或者 Number	要格式化的数值
type+	String	表明将该值格式化成数字、货币还是百分比。该属性值为以下三者之一： number、 currency 或 percent
pattern+	String	定制格式化模式
currencyCode+	String	ISO 4217 码，详情请查看表 5-9
currencySymbol+	String	货币符号
groupingUsed+	Boolean	表明输出中是否包含组分隔符
maxIntegerDigits+	int	输出的整数部分中最大的数字
minIntegerDigits+	int	输出的整数部分中最小的数字
maxFractionDigits+	int	输出的小数部分中最大的数字
minFractionDigits+	int	输出的小数部分中最小的数字
var	String	将输出保存为 String 的限域变量名称
scope	String	var 的范围。如果有 scope 属性，则必须指定 var 属性

formatNumber 的其中一个用处是将数字格式化成货币。为此，可以利用 currencyCode 属性指定一个 ISO 4217 货币代码。部分货币代码如表 5-9 所示。

表 5-9 ISO 4217 货币代码

货币类别	ISO 4217 码	最大单位	最小单位
加拿大元	CAD	dollar	cent
人民币	CNY	yuan	jiao
欧元	EUR	euro	euro-cent

(续)

货币类别	ISO 4217 码	最大单位	最小单位
日元	JRP	yen	sen
英镑	GBP	pound	pence
美元	USD	dollar	cent

表 5-10 示范 formatNumber 的用法。

表 5-10 使用 formatNumber 标签

action	结 果
<fmt:formatNumber value="12" type="number"/>	12
<fmt:formatNumber value="12" type="number" minIntegerDigits="3"/>	012
<fmt:formatNumber value="12" type="number" minFractionDigits="2"/>	12.00
<fmt:formatNumber value="123456.78" pattern=".000"/>	123456.780
<fmt:formatNumber value="123456.78" pattern="#,#00.0#"/>	123,456.78
<fmt:formatNumber value="12" type="currency"/>	\$12.00
<fmt:formatNumber value="12" type="currency" currencyCode="GBP"/>	GBP 12.00
<fmt:formatNumber value="0.12" type="percent"/>	12%
<fmt:formatNumber value="0.125" type="percent" minFractionDigits="2"/>	12.50%

注意，在格式化货币时，如果没有指定 currencyCode 属性，则使用浏览器所在地的货币类别。

## 5.6.2 formatDate 标签

利用 formatDate 标签格式化日期，其语法如下：

```
<fmt:formatDate value="date"
    [type="{time|date|both}"]
    [dateStyle="{default|short|medium|long|full}"]
    [timeStyle="{default|short|medium|long|full}"]
    [pattern="customPattern"]
    [timeZone="timeZone"]
    [var="varName"]
    [scope="{page|request|session|application}"]
/>
```

主体内容为 JSP。formatDate 标签的属性如表 5-11 所示。

表 5-11 formatDate 标签的属性

属性	类型	描述
value+	java.util.Date	要格式化的日期和 / 或时间
type+	String	表明是否格式化时间、日期，或日期和时间元件的组合
dateStyle+	String	为遵循 java.text.DateFormat 中所定义语义的日期预设格式化的样式
timeStyle+	String	为遵循 java.text.DateFormat 中所定义语义的时间预设格式化的样式
pattern+	String	为格式化定制模式
timezone+	String 或 java.util.TimeZone	表示时间的时区
var	String	将结果保存为字符串的限域变量名称
scope	String	var 的范围

关于 `timeZone` 属性的可能值，请查看“`timeZone` 标签”一节。

以下代码利用 `formatDate` 标签对限域对象 `now` 所引用的 `java.util.Date` 对象进行格式化：

```

Default: <fmt:formatDate value="${now}" />
Short: <fmt:formatDate value="${now}" dateStyle="short" />
Medium: <fmt:formatDate value="${now}" dateStyle="medium" />
Long: <fmt:formatDate value="${now}" dateStyle="long" />
Full: <fmt:formatDate value="${now}" dateStyle="full" />
```

下面的 `formatDate` 标签用来格式化时间：

```

Default: <fmt:formatDate type="time" value="${now}" />
Short: <fmt:formatDate type="time" value="${now}"
    timeStyle="short" />
Medium: <fmt:formatDate type="time" value="${now}"
    timeStyle="medium" />
Long: <fmt:formatDate type="time" value="${now}" timeStyle="long" />
Full: <fmt:formatDate type="time" value="${now}" timeStyle="full" />
```

下面的 `formatDate` 标签用来格式化日期和时间：

```

Default: <fmt:formatDate type="both" value="${now}" />
Short date short time: <fmt:formatDate type="both"
    value="${now}" dateStyle="short" timeStyle="short" />
Long date long time format: <fmt:formatDate type="both"
    value="${now}" dateStyle="long" timeStyle="long" />
```

下面的 `formatDate` 标签用不同的时区来格式化时间：

```

Time zone CT: <fmt:formatDate type="time" value="${now}"
    timeZone="CT" /><br/>
Time zone HST: <fmt:formatDate type="time" value="${now}"
    timeZone="HST" /><br/>
```

下面的 `formatDate` 标签利用定制模式来格式化日期和时间：

```
<fmt:formatDate type="both" value="${now}" pattern="dd.MM.yy"/>
<fmt:formatDate type="both" value="${now}" pattern="dd.MM.yyyy"/>
```

### 5.6.3 timeZone 标签

`timeZone` 标签用来指定时区，以便格式化或解析其主体内容中的时间信息，其语法如下：

```
<fmt:timeZone value="timeZone">
    body content
</fmt:timeZone>
```

主体内容为 JSP，属性值可以是类型为 `String` 或 `java.util.TimeZone` 的动态值。美国和加拿大的时区值如表 5-12 所示。

如果 `value` 属性为 `null` 或者为空，则使用 GMT（格林威治标准时间）时区。

下面的范例利用 `timeZone` 标签和时区来格式化日期：

```
<fmt:timeZone value="GMT+1:00">
    <fmt:formatDate value="${now}" type="both"
        dateStyle="full" timeStyle="full"/>
</fmt:timeZone>
<fmt:timeZone value="HST">
    <fmt:formatDate value="${now}" type="both"
        dateStyle="full" timeStyle="full"/>
</fmt:timeZone>
<fmt:timeZone value="CST">
    <fmt:formatDate value="${now}" type="both"
        dateStyle="full" timeStyle="full"/>
</fmt:timeZone>
```

表 5-12 美国和加拿大时区

缩写	全称	时区
NST	Newfoundland Standard Time (纽芬兰标准时间)	UTC-3:30 小时
NDT	Newfoundland Daylight Time (纽芬兰夏时制)	UTC-2:30 小时
AST	Atlantic Standard Time (大西洋标准时间)	UTC-4 小时
ADT	Atlantic Daylight Time (大西洋夏时制)	UTC-3 小时
EST	Eastern Standard Time (东部标准时间)	UTC-5 小时
EDT	Eastern Daylight Saving Time (东部夏时制)	UTC-4 小时
ET	Eastern Time, as EST or EDT (东部时间，与 EST 或 EDT 一样)	*
CST	Central Standard Time (中部标准时间)	UTC-6 小时
CDT	Central Daylight Saving Time (中部夏时制)	UTC-5 小时
CT	Central Time, as either CST or CDT (中部时间，与 CST 或 CDT 一样)	*
MST	Mountain Standard Time (山地标准时间)	UTC-7 小时

(续)

缩写	全称	时区
MDT	Mountain Daylight Saving Time (山地夏时制)	UTC-6 小时
MT	Mountain Time, as either MST or MDT (山地时间, 与 MST 或 MDT 一样)	*
PST	Pacific Standard Time (太平洋标准时间)	UTC-8 小时
PDT	Pacific Daylight Saving Time (太平洋夏时制)	UTC-7 小时
PT	Pacific Time, as either PST or PDT (太平洋时间, 与 PST 或 PDT 一样)	*
AKST	Alaska Standard Time (阿拉斯加标准时间)	UTC-9 小时
AKDT	Alaska Standard Daylight Saving Time (阿拉斯加白昼时间)	UTC-8 小时
HST	Hawaiian Standard Time (夏威夷标准时间)	UTC-10 小时

## 5.6.4 setTimeZone 标签

利用 `setTimeZone` 标签将指定的时区保存在某个限域变量或者时间配置变量中。`setTimeZone` 的语法如下：

```
<fmt:setTimeZone value="timeZone" [var="varName"]
                  [scope="{page|request|session|application}"]/>
/>
```

表 5-13 展示了 `setTimeZone` 标签的属性。

表 5-13 `setTimeZone` 标签的属性

属性	类型	描述
value+	String 或者 <code>java.util.TimeZone</code>	时区
var	String	限域变量的名称, 用于保存类型为 <code>java.util.TimeZone</code> 的时间
scope	String	<code>var</code> 的范围, 或者时区配置变量的范围

## 5.6.5 parseNumber 标签

利用 `parseNumber` 标签可以将数字、货币或百分比的字符串表示法解析成指定语言环境的数字。其语法有两种形式，第一种形式没有主体内容：

```
<fmt:parseNumber value="numericValue"
                  [type="{number|currency|percent}"]
                  [pattern="customPattern"]
                  [parseLocale="parseLocale"]
                  [integerOnly="{true|false}"]
                  [var="varName"]
                  [scope="{page|request|session|application}"]
/>
```

第二种形式使用了主体内容：

```

<fmt:parseNumber [type="number|currency|percent"]>
    [pattern="customPattern"]
    [parseLocale="parseLocale"]
    [integerOnly="true|false"]
    [var="varName"]
    [scope="page|request|session|application"]>
        numeric value to be parsed
</fmt:parseNumber>

```

主体内容为 JSP。parseNumber 标签的属性如表 5-14 所示。

举个例子。下面的 parseNumber 标签解析限域变量 quantity 所引用的值，并将结果保存在限域变量 formattedNumber 中。

```
<fmt:parseNumber var="formattedNumber" type="number"
    value="${quantity}"/>
```

表 5-14 parseNumber 标签的属性

属性	类型	描述
value+	String	要解析的字符串
type+	String	表明要将字符串解析成数字、货币还是百分比
pattern+	String	定制格式化模式，以确定如何解析 value 属性中的字符串
parseLocale+	String 或 java.util.Locale	解析操作期间使用的默认格式化模式所属的区域，或将通过 pattern 属性指定的模式应用到的指定区域
integerOnly+	Boolean	表明是否只解析给定值的整数部分
var	String	用于保存结果的限域变量名称
scope	String	var 的范围

## 5.6.6 parseDate 标签

parseDate 标签为指定区域解析日期和时间的字符串表示法，其语法有两种形式，第一种形式中没有使用主体内容：

```

<fmt:parseDate value="dateString"
    [type="time|date|both"]
    [dateStyle="default|short|medium|long|full"]
    [timeStyle="default|short|medium|long|full"]
    [pattern="customPattern"]
    [timeZone="timeZone"]
    [parseLocale="parseLocale"]
    [var="varName"]
    [scope="page|request|session|application"]>
/>

```

第二种形式使用了主体内容：

```
<fmt:parseDate [type="time|date|both"]
    [dateStyle="default|short|medium|long|full"]>
```

```
[timeStyle="default|short|medium|long|full"]
[pattern="customPattern"]
[timeZone="timeZone"]
[parseLocale="parseLocale"]
[var="varName"]
[scope="{page|request|session|application}"]
    date value to be parsed
</fmt:parseDate>
```

主体内容为 JSP。表 5-15 列出了 **parseDate** 标签的属性。

表 5-15 parseDate 标签的属性

属性	类型	描述
value+	String	要解析的字符串
type+	String	表明要将字符串解析成包含数字、时间，还是两者都要
dateStyle+	String	日期的格式化样式
timeStyle+	String	时间的格式化样式
pattern+	String	定制格式化模式，以确定如何解析字符串
timeZone+	String 或 java.util.TimeZone	将日期字符串中的所有时间信息都转换成指定时区的时间
parseLocale+	String 或 java.util.Locale	解析操作期间使用的默认格式化模式所属的区域，或者要将通过 pattern 属性指定的模式应用到的指定区域
var	String	用于保存结果的限域变量名称
scope	String	var 的范围

举个例子。下面的 **parseDate** 标签解析限域变量 **myDate** 引用的日期，并将得到的 **java.util.Date** 保存在限域变量 **formattedDate** 中。

```
<c:set var="myDate" value="12/12/2005"/>
<fmt:parseDate var="formattedDate" type="date"
    dateStyle="short" value="${myDate}"/>
```

## 5.7 函数

除了定制动作指令之外，JSTL 1.1 和 1.2 中还定义了一组标准函数，可以用在 EL 表达式中。这些函数分组放在函数标签类库中。当使用这些函数时，必须在 JSP 上使用 **taglib** 指令：

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions"
    prefix="fn" %>
```

调用函数时，要以下面这种格式来使用 EL：

```
${fn:functionName}
```

这里的 *functionName* 是函数名称。

这里的大多数函数都是用于字符串操作的。例如，`length` 函数适用于字符串和集合，并返回集合或数组中的项目数，或者字符串中的字符数。

接下来就详细讲解这些函数。

### 5.7.1 contains 函数

`contains` 函数用于检测某个字符串中是否包含指定的子字符串。如果该字符串中包含指定的子字符串，则返回值为 `true`；否则为 `false`。其语法如下：

```
contains(string, substring)
```

例如，下面这些 EL 表达式都返回 `true`：

```
<c:set var="myString" value="Hello World"/>
${fn:contains(myString, "Hello")}

${fn:contains("Stella Cadente", "Cadente")}
```

### 5.7.2 containsIgnoreCase 函数

`containsIgnoreCase` 函数与 `contains` 函数相似，但其检测是区分大小写的，具体语法如下：

```
containsIgnoreCase(string, substring)
```

例如，以下 EL 表达式将返回 `true`：

```
${fn:containsIgnoreCase("Stella Cadente", "CADENTE")}
```

### 5.7.3 endsWith 函数

`endsWith` 函数检测某个字符串是否以指定的后缀结尾。其返回值是一个 `Boolean` 值，具体语法如下：

```
endsWith(string, suffix)
```

例如，以下 EL 表达式将返回 `true`：

```
${fn:endsWith("Hello World", "World")}
```

### 5.7.4 escapeXml 函数

该函数用于给 `String` 进行编码。它的转换方法与 `out` 标签将其 `escapeXml` 属性设为 `true` 时相似，具体语法如下：

```
escapeXml(string)
```

例如，EL 表达式

```
 ${fn:escapeXml("Use <br/> to change lines")}
```

显示为：

```
Use &lt;br/&gt; to change lines
```

### 5.7.5 indexOf 函数

**indexOf** 函数返回在某个字符串中第一次出现指定子字符串时的索引。如果没有找到指定的子字符串，它将返回 -1，具体语法如下：

```
 indexOf(string, substring)
```

例如，以下 EL 表达式将返回 7：

```
 ${fn:indexOf("Stella Cadente", "Cadente")}
```

### 5.7.6 join 函数

**join** 函数将一个 **String** 数组中的所有元素都合并成一个字符串，中间用指定的分隔符隔开。其语法如下：

```
 join(array, separator)
```

如果数组为 **null**，将返回一个空的字符串。

例如，如果 **myArray** 是一个具有两个元素“my”和“world”的 **String** 数组，那么 EL 表达式：

```
 ${fn:join(myArray, ",")}
```

将返回“my,world”。

### 5.7.7 length 函数

**length** 函数返回一个集合中的项目数，或者一个字符串中的字符数。其语法如下：

```
 length(input)
```

举个例子，下面的 EL 表达式将返回 14：

```
 ${fn:length("Stella Cadente", "Cadente")}
```

### 5.7.8 replace 函数

**replace** 函数用 *afterString* 替换某一个字符串中所有的 *beforeString*，并返回结果。其语法如下：

```
replace(string, beforeSubstring, afterSubstring)
```

例如，EL 表达式

```
 ${fn:replace("Stella Cadente", "e", "E")}
```

将返回 “StEllA CaDEntE”。

### 5.7.9 split 函数

**split** 函数将一个字符串分解成一组子字符串，它的作用与 **join** 相反。例如，以下代码分解字符串 “my,world”，并将结果保存在限域变量 **split** 中。然后，它利用 **forEach** 标签将 **split** 格式化成一个 HTML 表格：

```
<c:set var="split" value='${fn:split("my,world","","")}'/>
<table>
<c:forEach var="substring" items="${split}">
    <tr><td>${substring}</td></tr>
</c:forEach>
</table>
```

其结果为：

```
<table>
    <tr><td>my</td></tr>
    <tr><td>world</td></tr>
</table>
```

### 5.7.10 startsWith 函数

**startsWith** 函数测试某个字符串是否以指定的前缀开头，其语法如下：

```
startsWith(string, prefix)
```

例如，以下 EL 表达式将返回 true：

```
 ${fn:startsWith("Stella Cadente", "St")}
```

### 5.7.11 substring 函数

**substring** 函数返回一个从指定的起始索引（从 0 开始，含起始索引）至指定的结束索引之间的子字符串，其语法如下：

```
substring(string, beginIndex, endIndex)
```

例如，以下 EL 表达式将返回 “Stel”。

```
 ${fn:substring("Stella Cadente", 0, 4)}
```

### 5.7.12 substringAfter 函数

**substringAfter** 函数返回第一次出现指定子字符串之后的字符串部分，其语法如下：

```
substringAfter(string, substring)
```

例如，以下 EL 表达式

```
${fn:substringAfter("Stella Cadente", "e")}
```

将返回 “lla Cadente”。

### 5.7.13 substringBefore 函数

**substringBefore** 函数返回第一次出现子字符串之前的字符串部分，其语法如下：

```
substringBefore(string, substring)
```

例如，以下 EL 表达式将返回 “St”。

```
${fn:substringBefore("Stella Cadente", "e")}
```

### 5.7.14 toLowerCase 函数

**toLowerCase** 函数将字符串转换成全部小写字母，其语法如下：

```
toLowerCase(string)
```

例如，以下 EL 表达式将返回 “stella cadente”：

```
${fn:toLowerCase("Stella Cadente")}
```

### 5.7.15 toUpperCase 函数

**toUpperCase** 函数将字符串转换成全部大写字母，其语法如下：

```
toUpperCase(string)
```

例如，以下 EL 表达式将返回 “STELLA CADENTE”：

```
${fn:toUpperCase("Stella Cadente")}
```

### 5.7.16 trim 函数

**trim** 函数去掉字符串前后的空格，其语法如下：

```
trim(string)
```

例如，以下 EL 表达式将返回 “Stella Cadente”：

```
 ${fn:trim(" Stella Cadente ")}
```

## 5.8 小结

利用 JSTL 可以完成普通的任务（例如迭代、集合及条件测试），也可以处理 XML 文档、格式化文本、访问数据库和操作数据等。本章主要讨论了比较重要的一些标签，例如用于操作限域对象（`out`、`set`、`remove`）、进行条件测试（`if`、`choose`、`when`、`otherwise`）、迭代集合或者 token（`forEach`、`forTokens`）、解析和格式化日期和数字（`parseNumber`、`formatNumber`、`parseDate`、`formatDate` 等）的标签，还介绍了可以通过 EL 表达式使用的 JSTL 1.2 函数。

# 第 6 章 编写定制标签

在第 5 章中，我们学习了如何在 JSTL 中使用定制标签。JSTL 中的类库提供了用于解决常见问题的标签，如果你的问题比较特殊，那么就需要通过扩展 `javax.servlet.jsp.tagext` 包的成员，来编写自己的定制标签。下面我们学习如何编写定制标签。

## 6.1 定制标签概述

利用 JSP 标准动作指令访问和操作 JavaBeans，是首次尝试将表现代码（HTML）和业务逻辑实现（Java 代码）分离。但是，标准动作指令的功能不够强大，单独使用时，开发者经常要使用 JSP 页面中的 Java 代码。例如，标准动作指令无法像 JSTL 的 `forEach` 标签那样迭代集合。

认识到了用 JavaBeans 分离表现逻辑和业务逻辑的不足之处之后，JSP 1.1 就定义了定制标签。定制标签具有 JavaBeans 所没有的优势。例如，定制标签可以访问 JSP 隐式对象，可以带有属性等。

虽然利用定制标签有助于你编写无脚本的 JSP 页面，但是众所周知，在 JSP 1.1 和 1.2 中编写定制标签（也称作典型的定制标签，Classic Custom Tag）很困难。JSP 2.0 中添加了两项新特性，使得编写定制标签变得比较容易一些。第一项特性是一个新的接口，称作 `SimpleTag`，这个将在本章中讨论。第二项特性是一种使编写定制标签就像编写标签文件一样的机制。标签文件的详情可查看第 7 章的内容。

定制标签的实现称为标签处理器（Tag Handler），简单的标签处理器是指实现 `SimpleTag` 接口的标签处理器。在本章中，我们将学习定制标签的工作原理，以及如何编写标签处理器。之所以只讲解简单的标签处理器，是因为现在已经不再编写典型的标签处理器（Classic Tag Handler）了。

与典型的标签处理器相比，简单标签处理器除了更容易编写之外，它也不像典型的标签处理器那样，可以由 JSP 容器进行缓存。但是，这并不意味着简单的标签处理器就比典型的标签处理器更慢。JSP 规范制定者在规范的 JSP.7.1.5 小节中写道：最初的性能规则显示，缓存标签处理器实例并不能提高性能，这种缓存反而使得编写轻便的标签处理器变得更加困难，并且使得标签处理器更加容易出错。

## 6.2 简单的标签处理器

JSP 2.0 的设计者意识到了在 JSP 1.1 和 JSP 1.2 中编写定制标签和标签处理器是多么地复杂。因此，在 JSP 2.0 中，他们在 `javax.servlet.jsp.tagext` 包中添加了新的接口：`SimpleTag`。实现 `SimpleTag` 接口的标签处理器称作简单的标签处理器，实现 `Tag`、`IterationTag` 或 `BodyTag` 接口的标签处理器称作典型的标签处理器。

简单的标签处理器的生命周期更加简单，并且编写起来也比典型的标签处理器更容易。`SimpleTag` 接口中只有一个方法：`doTag`，并且在标签调用时只执行一次。业务逻辑、迭代及主体操作代码都要在这里编写。简单的标签处理器中的主体是用一个 `JspFragment` 类实例表示的。`JspFragment` 将在稍后的小节中讨论。

一个简单的标签处理器的生命周期如下：

1. JSP 容器通过调用其无参构造器，创建一个简单标签处理器实例。因此，简单的标签处理器必须有一个无参构造器。
2. JSP 容器调用 `setJspContext` 方法，同时传递一个 `JspContext` 对象。`JspContext` 最重要的方法是 `getOut`，它返回一个 `JspWriter`，用于将响应发送到客户端。`setJspContext` 方法的签名如下：

```
public void setJspContext(JspContext jspContext)
```

大多数时候，会需要将传进的 `JspContext` 赋给一个类变量，以便供后续使用。

3. 如果表示标签处理器的定制标签是嵌套在另一个标签中的，JSP 容器就会调用 `setParent` 方法。该方法具有如下签名：

```
public void setParent(JspTag parent)
```

4. JSP 容器为给该标签定义的每个属性都调用设置方法（Setter）。
5. 如果标签中有主体内容，JSP 将调用 `SimpleTag` 接口的 `setJspBody` 方法，将主体内容作为 `JspFragment` 传递。如果没有主体内容，JSP 容器则不会调用这个方法。

6. JSP 容器调用 `doTag` 方法。所有变量在 `doTag` 方法返回时进行同步。

`javax.servlet.jsp.tagext` 包中也含有 `SimpleTag` 接口的一个支持类：`SimpleTagSupport`。`SimpleTagSupport` 为 `SimpleTag` 接口的所有方法都提供了默认实现，并且作为一个便利类，你可以继承它，来编写一个简单的标签处理器。当 JSP 容器调用 `SimpleTag` 接口的 `setJspContext` 方法时，`SimpleTagSupport` 类中的 `getJspContext` 方法将返回 JSP 容器传递的 `JspContext` 实例。

## 6.3 SimpleTag 范例

本节介绍 `app06a` 应用程序，这是一个简单标签处理器的范例。创建一个定制标签需要

两个步骤：编写标签处理器和注册标签。接下来介绍这两个步骤。

注意，为了编译标签处理器，需要在构建路径中包含 Servlet API 和 JSP API 类包。如果你使用的是 Tomcat，将可以在 Tomcat 的 lib 目录下找到包含这两个 API 的 jar 文件（servlet-api.jar 文件和 jsp-api.jar 文件）。

app06a 的应用程序目录结构如图 6-1 所示。定制标签包含一个标签处理器（放在 WEB-INF/classes 目录下）和一个描述符（放在 WEB-INF 目录下的 mytags.tld 文件）。图 6-1 中还包含一个测试定制标签的 JSP 文件。



图 6-1 app06a 范例应用程序的目录结构

### 6.3.1 编写标签处理器

代码清单 6-1 中展示了 MyFirstTag 类，这是 SimpleTag 接口的一个实现。

代码清单 6-1 MyFirstTag 类

---

```

package customtag;
import java.io.IOException;
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.JspFragment;
import javax.servlet.jsp.tagext.JspTag;
import javax.servlet.jsp.tagext.SimpleTag;

public class MyFirstTag implements SimpleTag {
    JspContext jspContext;

    public void doTag() throws IOException, JspException {
        System.out.println("doTag");
        jspContext.getOut().print("This is my first tag.");
    }

    public void setParent(JspTag parent) {
        System.out.println("setParent");
    }

    public JspTag getParent() {
        System.out.println("getParent");
        return null;
    }

    public void setJspContext(JspContext jspContext) {
        System.out.println("setJspContext");
        this.jspContext = jspContext;
    }

    public void setJspBody(JspFragment body) {
        System.out.println("setJspBody");
    }
}

```

---

MySimpleTag 类有一个类型为 JspContext 的 `jspContext` 变量。`setJspContext` 方法将它从 JSP 容器处接收到的 `JspContext` 赋给这个变量。`doTag` 方法利用 `JspContext` 获得一个 `JspWriter`。然后调用 `JspWriter` 的 `print` 方法，输出 `String`: “This is my first tag.”

### 6.3.2 注册标签

在 JSP 页面中使用标签处理器之前，必须先在一个标签类库描述符中对它进行注册，那是一个以 `tld` 为扩展名的 XML 文件。这个范例的标签类型描述符为 `mytags.tld`，如代码清单 6-2 所示。该文件必须放在 `WEB-INF` 目录下。

代码清单 6-2 标签类库描述符（`mytags.tld` 文件）

---

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib 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
➥ web-jsptaglibrary_2_1.xsd"
         version="2.1">

    <description>
        Simple tag examples
    </description>
    <tlib-version>1.0</tlib-version>
    <short-name>My First Taglib Example</short-name>
    <tag>
        <name>firstTag</name>
        <tag-class>customtag.MyFirstTag</tag-class>
        <body-content>empty</body-content>
    </tag>
</taglib>

```

---

标签类库描述符中的主要元素是 `tag`，它用来描述标签，其中包含一个 `name` 元素和一个 `tag-class` 元素。`name` 用于给该标签命名。`tag-class` 用于指定标签处理器的全类名。一个标签类库描述符中可以包含多个 `tag` 元素。

此外，标签类库描述符中还可以有其他元素。`description` 元素用于对该描述符中所描述的标签提供一条描述。`tlib-version` 元素设置定制标签的版本，`short-name` 元素为这些标签提供一个简称。

### 6.3.3 使用标签

使用定制标签时，要利用 `taglib` 指令。`taglib` 指令的 `uri` 属性可以引用相对路径，也可以引用绝对路径。在本例中使用的是相对路径。然而，如果使用的是打包在 `jar` 文件中的标签类库，就要使用绝对路径。本章稍后将介绍如何打包定制标签类库，才能更便于发布。

为了对定制标签 `firstTag` 进行测试，我们在代码清单 6-3 中使用了 `firstTagTest.jsp` 页面。

代码清单 6-3 firstTagTest.jsp

---

```
<%@ taglib uri="/WEB-INF/mytags.tld" prefix="easy"%>
<html>
<head>
    <title>Testing my first tag</title>
</head>
<body>
Hello!!!!
<br/>
<easy:firstTag></easy:firstTag>
</body>
</html>
```

---

利用下面这个 URL 可以调用 firstTagTest.jsp 页面：

`http://localhost:8080/app06a/firstTagTest.jsp`

在调用 firstTagTest.jsp 页面时，JSP 容器会调用标签处理器的 `setJspContext` 方法。由于 firstTagTest.jsp 没有主体内容，JSP 容器不会在调用 `doTag` 方法之前调用 `setJspBody` 方法。在控制台上，你会看到以下内容：

```
setJspContext
doTag
```

注意，JSP 容器也没有调用标签处理器的 `setParent` 方法，因为这个简单的标签没有嵌在另一个标签里面。

## 6.4 处理属性

实现 `SimpleTag` 接口或者继承 `SimpleTagSupport` 的标签处理器可以带有属性。代码清单 6-4 中展示了一个标签处理器 `DataFormatterTag`，它将一系列以逗号分隔的项格式化成一个 HTML 表格。给这个标签设置两个属性：`header` 和 `items`。`header` 属性值将成为表格的标题。例如，如果用“`Cities`”作为 `header` 的属性值，用“`London,Montreal`”作为 `items` 的属性值，那么得到的输出内容将如下所示：

```
<table style="border:1px solid green">
<tr><td><b>Cities</b></td></tr>
<tr><td>London</td></tr>
<tr><td>Montreal</td></tr>
</table>
```

代码清单 6-4 DataFormatterTag 类

---

```
package customtag;
import java.io.IOException;
```

```

import java.util.StringTokenizer;
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.SimpleTagSupport;
public class DataFormatterTag extends SimpleTagSupport {
    private String header;
    private String items;

    public void setHeader(String header) {
        this.header = header;
    }

    public void setItems(String items) {
        this.items = items;
    }

    public void doTag() throws IOException, JspException {
        JspContext jspContext = getJspContext();
        JspWriter out = jspContext.getOut();

        out.print("<table style='border:1px solid green'>\n"
                  + "<tr><td><span style='font-weight:bold'>"
                  + header + "</span></td></tr>\n");
        StringTokenizer tokenizer = new StringTokenizer(items,
                ",");
        while (tokenizer.hasMoreTokens()) {
            String token = tokenizer.nextToken();
            out.print("<tr><td>" + token + "</td></tr>\n");
        }
        out.print("</table>");
    }
}

```

---

**DataFormatterTag** 类提供了两个设置方法用来接收属性：**setHeader** 和 **setItems**。其余的任务就交给 **doTag** 了。

**doTag** 方法首先通过调用 **getJspContext** 方法获得 JSP 容器传来的 **JspContext**:

```
JspContext jspContext = getJspContext();
```

然后，它在 **JspContext** 实例中调用 **getOut** 方法，获得一个 **JspWriter**，用它编写给客户端的响应：

```
JspWriter out = jspContext.getOut();
```

接下来，**doTag** 方法利用 **StringTokenizer** 解析 **items** 属性，并将每个项设置成一个表格行：

```
out.print("<table style='border:1px solid green'>\n"
          + "<tr><td><span style='font-weight:bold'>"
```

```

        + header + "</span></td></tr>\n");
StringTokenizer tokenizer = new StringTokenizer(items, ",");
while (tokenizer.hasMoreTokens()) {
    String token = tokenizer.nextToken();
    out.print("<tr><td>" + token + "</td></tr>\n");
}
out.print("</table>");

```

为了使用 **DataFormatterTag** 标签处理器，必须利用代码清单 6-5 中的 **tag** 元素对它进行注册。只要将它添加到前一个例子中用过的 **mytags.tld** 文件中即可。

代码清单 6-5 注册 dataFormatter 标签

---

```

<tag>
    <name>dataFormatter</name>
    <tag-class>customtag.DataFormatterTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>header</name>
        <required>true</required>
    </attribute>
    <attribute>
        <name>items</name>
        <required>true</required>
    </attribute>
</tag>

```

---

然后，可以利用代码清单 6-6 中的 **dataFormatterTagTest.jsp** 页面来对标签处理器进行测试。

代码清单 6-6 dataFormatterTagTest.jsp 页面

---

```

<%@ taglib uri="/WEB-INF/mytags.tld" prefix="easy"%>
<html>
<head>
    <title>Testing DataFormatterTag</title>
</head>
<body>
<easy:dataFormatter header="States"
    items="Alabama,Alaska,Georgia,Florida"
/>
<br/>
<easy:dataFormatter header="Countries">
    <jsp:attribute name="items">
        US,UK,Canada,Korea
    </jsp:attribute>
</easy:dataFormatter>
</body>
</html>

```

---

注意，代码清单 6-6 中的 JSP 页面使用了两次 `dataFormatter` 标签，并且以两种不同的方式传递属性，一种是用一个标签属性，另一种是利用了 `attribute` 标准的动作指令。利用下面这个 URL 可以调用 `dataFormatterTagTest.jsp`：

`http://localhost:8080/app06a/dataFormatterTagTest.jsp`

图 6-2 展示了调用 `dataFormatterTagTest.jsp` 之后的结果。

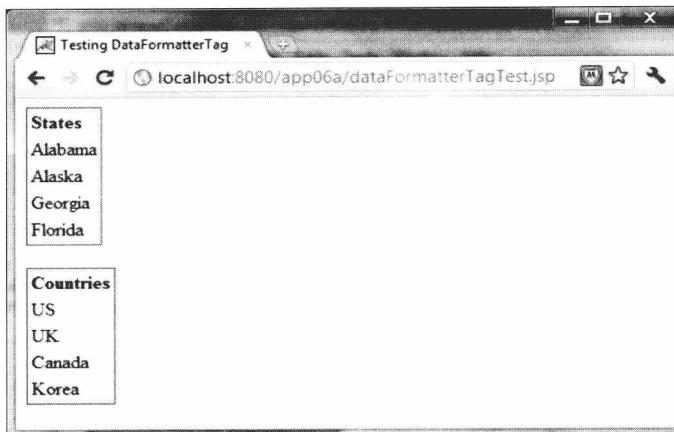


图 6-2 在 SimpleTag 中使用属性

## 6.5 管理标签主体

有了 `SimpleTag`，就可以通过 JSP 传来的 `JspFragment` 管理标签主体了。`JspFragment` 类表示一段 JSP 代码，可以不调用，也可以调用多次。JSP 片断的定义中不能包含 `Scriptlet` 或者 `Scriptlet 表达式`，它只能包含模板文本和 JSP 动作指令元素。

`JspFragment` 类有两个方法：`getJspContext` 和 `invoke`，其方法签名如下：

```
public abstract JspContext getJspContext()
public abstract void invoke(java.io.Writer writer)
    throws JspException, java.io.IOException
```

`getJspContext` 方法返回与这个 `JspFragment` 相关的 `JspContext`。我们可以调用 `invoke` 方法来执行片断（标签主体），并将所有输出内容导到指定的 `Writer`。如果传给 `invoke` 方法的值为 `null`，那么输出的结果将会被导到与该片断相关的 `JspContext` 的 `getOut` 方法所返回的 `JspWriter`。

请看代码清单 6-7 中的 `SelectElementTag` 类。利用这个标签处理器以下列格式发送一个 HTML 的 `select` 元素：

```
<select>
<option value="value-1">text-1</option>
<option value="value-2">text-2</option>
...
<option value="value-n">text-n</option>
</select>
```

在这个例子中，它的值是指 String 数组 countries 中的国家名称。

代码清单 6-7 SelectElementTag

---

```
package customtag;
import java.io.IOException;
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class SelectElementTag extends SimpleTagSupport {
    private String[] countries = {"Australia", "Brazil", "China" };

    public void doTag() throws IOException, JspException {
        JspContext jspContext = getJspContext();
        JspWriter out = jspContext.getOut();
        out.print("<select>\n");
        for (int i=0; i<3; i++) {
            getJspContext().setAttribute("value", countries[i]);
            getJspContext().setAttribute("text", countries[i]);
            getJspBody().invoke(null);
        }
        out.print("</select>\n");
    }
}
```

---

代码清单 6-8 中展示了用来注册 SelectElementTag 的 tag 元素，并将它映射到 select 标签。同样，也要将这个元素添加到前面例子中用过的 mytags.tld 文件中。

代码清单 6-8 注册 SelectElementTag

---

```
<tag>
    <name>select</name>
    <tag-class>customtag.SelectElementTag</tag-class>
    <body-content>scriptless</body-content>
</tag>
```

---

代码清单 6-9 中展示了一个使用了 SelectElementTag 的 JSP 页面（selectElementTagTest.jsp）。

## 代码清单 6-9 selectElementTagTest.jsp 页面

---

```
<%@ taglib uri="/WEB-INF/mytags.tld" prefix="easy"%>
<html>
<head>
    <title>Testing SelectElementFormatterTag</title>
</head>
<body>
<easy:select>
    <option value="\$\{value\}">\$\{text\}</option>
</easy:select>
</body>
</html>
```

---

注意，使用 `select` 标签要通过以下格式传递一个标签主体：

```
<option value="\$\{value\}">\$\{text\}</option>
```

通过在 `SelectElementTag` 标签处理器 `doTag` 方法中对 `JspFragment` 进行每一次的调用中，`value` 和 `text` 属性都获得了值：

```
for (int i=0; i<3; i++) {
    getJspContext().setAttribute("value", countries[i]);
    getJspContext().setAttribute("text", countries[i]);
    getJspBody().invoke(null);
}
```

利用下面这个 URL 可以调用 `selectElementTagTest.jsp`：

`http://localhost:8080/app06a/selectElementTagTest.jsp`

图 6-3 显示了调用结果。

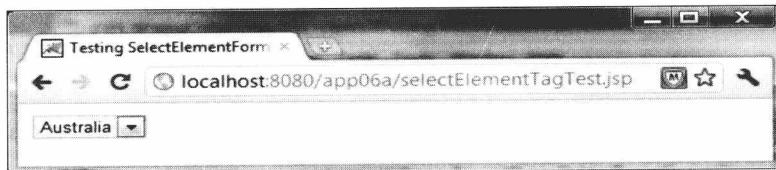


图 6-3 使用 JspFragment

如果在 Web 浏览器中查看源代码，将会看到以下内容：

```
<select>
    <option value="Australia">Australia</option>
    <option value="Brazil">Brazil</option>
    <option value="China">China</option>
</select>
```

## 6.6 编写 EL 函数

第 4 章讨论了 JSP Expression Language (EL)，并讲过可以利用 EL 表达式来编写能够调用的函数。编写 EL 函数的方法放在本章中讨论，因为涉及标签类库描述符的使用。

一般来说，编写一个 EL 函数要遵循以下两个步骤：

1. 创建一个包含静态方法的 public 类。每个静态方法表示一个函数。这个类不需要实现接口或者继承类。你可以根据需要，像对待其他任何类一样部署这个类。这个类必须保存到 WEB-INF/classes 目录或其下面的某个目录中。

2. 利用 **function** 元素在标签类库描述符中注册函数。

**function** 元素必须直接放在 **taglib** 元素下，并且可以带有以下子元素：

- ❑ **description**。这是一条特定于标签的可选信息。

- ❑ **display-name**。XML 工具显示的简称。

- ❑ **icon**。XML 工具可以使用的可选图标元素。

- ❑ **name**。该函数独特的名称。

- ❑ **function-class**。实现该函数的 Java 类的全类名。

- ❑ **function-signature**。表示该函数的静态 Java 方法签名。

- ❑ **example**。使用该函数的一个 **example** 的可选信息描述。

- ❑ **function-extension**。通过 XML 工具使用，没有扩展名，或者有多个扩展名，提供关于该函数的其他信息。

使用函数时，需利用 **taglib** 指令及其 **uri** 属性，它指向标签类库描述符，以及要使用的前缀。然后，在 JSP 页面中利用以下语法调用函数：

```
 ${prefix:functionName(parameterList) }
```

举个例子。请看本书配套的 app06b 范例应用程序。代码清单 6-10 中展示的 **MyFunctions** 类，封装着静态方法 **reverseString**。

代码清单 6-10 MyFunctions 类中的 reverseString 方法

---

```
 package function;
 public class StringFunctions {
     public static String reverseString(String s) {
         return new StringBuffer(s).reverse().toString();
     }
 }
```

---

代码清单 6-11 中展示的 **functiontags.tld** 描述符中，包含了一个描述 **reverseString** 函数的 **function** 元素。这个 TLD 必须保存在使用该函数的应用程序的 WEB-INF 目录下。

代码清单 6-11 functiontags.tld 文件

---

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib 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
    ↵web-jsptaglibrary_2_1.xsd"
    version="2.1">

    <description>
        Function tag examples
    </description>
    <tlib-version>1.0</tlib-version>
    <function>
        <description>Reverses a String</description>
        <name>reverseString</name>
        <function-class>function.StringFunction</function-class>
        <function-signature>
            java.lang.String reverseString(java.lang.String)
        </function-signature>
    </function>
</taglib>

```

---

代码清单 6-12 展示了用来测试 EL 函数的 reverseStringFunctionTest.jsp 页面。

代码清单 6-12 使用 EL 函数

---

```

<%@ taglib uri="/WEB-INF/functiontags.tld" prefix="f"%>
<html>
<head>
    <title>Testing reverseString function</title>
</head>
<body>
    ${f:reverseString("Hello World")}
</body>
</html>

```

---

利用下面这个 URL 调用 useELFunctionTest.jsp:

<http://localhost:8080/app06b/reverseStringFunctionTest.jsp>

调用了 JSP 页面之后，就可以看到“Hello World”了。

## 6.7 发布定制标签

我们可以将定制标签处理器和标签类库描述符打包成一个 jar 文件，以便发给其他人使用，像 JSTL 一样。在这种情况下，就需要包含所有的标签处理器，以及描述它们的 tld 文件。此外，还需要在描述符的 uri 元素中指定一个绝对的 URL。

例如，本书配套的 app06c 范例应用程序就是将 app06b 中的标签和描述符打包成一个 mytags.jar 文件。jar 文件的内容如图 6-4 所示。

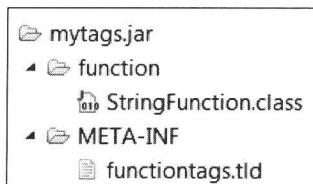


图 6-4 mytags.jar 文件

代码清单 6-13 展示了 functiontags.tld 文件。注意，描述符中已经添加了一个 uri 元素，元素值为 `http://example.com/taglib/function`。

#### 代码清单 6-13 打包定制标签中的 functiontags.tld 文件

---

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib 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
                             web-jsptaglibrary_2_1.xsd"
         version="2.1">
    <description>
        Function tag examples
    </description>
    <tlib-version>1.0</tlib-version>

    <uri>http://example.com/taglib/function</uri>
    <function>
        <description>Reverses a String</description>
        <name>reverseString</name>
        <function-class>function.StringFunction</function-class>
        <function-signature>
            java.lang.String reverseString(java.lang.String)
        </function-signature>
    </function>
</taglib>
  
```

---

在应用程序中使用类库时，必须将 jar 文件复制到应用程序的 WEB-INF/lib 下。此外，使用定制标签的所有 JSP 页面都必须指定与标签类库描述符中定义的相同 URI。

代码清单 6-14 展示了一个使用了定制标签的 JSP 页面。

#### 代码清单 6-14 app06c 中的 reverseStringFunction.jsp 文件

---

```

<%@ taglib uri="http://example.com/taglib/function" prefix="f"%>
<html>
<head>
    <title>Testing reverseString function</title>
  
```

---

```
</head>
<body>
${f:reverseString("Welcome")}
</body>
</html>
```

---

你可以在浏览器中打开下面这个网址，对上述例子进行验证：

<http://localhost:8080/app06c/reverseStringFunction.jsp>

## 6.8 小结

从本章可以看出，对于分离表现逻辑和业务逻辑而言，用定制标签比用 JavaBeans 要好得多。编写定制标签时，需要创建一个标签处理器，并在标签类库描述符中注册标签。

到 JSP 2.2 为止，有两种标签处理器：典型的标签处理器和简单的标签处理器。前者实现 Tag、IterationTag 或 BodyTag 接口，或者继承以下这两个支持类中的一个：TagSupport 和 BodyTagSupport。另一方面，简单的标签处理器则是实现 SimpleTag 接口，或者继承 SimpleTagSupport。简单的标签处理器更容易编写，其生命周期也比典型的标签处理器更为简单。建议使用简单的标签处理器。本章还举了几个简单标签处理器的例子。还可以将定制标签类库打包成一个 jar 文件供其他人使用。

# 第7章 标签文件

我们在第6章中已经讲过，定制标签使我们得以编写无脚本的JSP页面，从而可将工作进行分离，这意味着网页设计师和Java程序员可以同时工作。但是你也知道，编写定制标签是一项繁杂的工作，其中包括编写和编译标签处理器，以及在标签类库描述符中定义标签。

从JSP 2.0开始，我们就可以编写标签文件了，它是指没有标签处理器和标签类库描述符的定制动作指令。标签文件必须不用编译，必须没有标签类库描述符。

本章将详细讨论标签文件。首先简单介绍一下标签文件，随后介绍只利用标签文件编写定制标签的几个方面。在最后两个小节中还将讨论标准动作指令：`doBody` 和 `invoke`。

## 7.1 标签文件简介

标签文件从两个方面简化了编写定制标签的过程。第一，标签文件不需要像第一次被调用时那样进行编译。另外，只用JSP语法就可以编写标签文件的标签扩展。这意味着不懂Java的人也可以编写标签扩展！

第二，不需要标签类库描述符。标签类库描述符中的`tag`元素描述了要在JSP页面中用来引用定制动作指令的名称。使用标签文件时，定制动作指令的名称与表示该动作指令的标签文件相同，因此不需要标签类库描述符了。

JSP容器可以选择将标签文件编译成Java标签处理器或者解读标签文件。例如，Tomcat将标签文件转换成实现`javax.servlet.jsp.tagext.SimpleTag`接口的简单标签处理器。

标签文件看起来就像一个JSP页面，它可以有指令、Script、EL表达式、标准动作指令及定制标签。标签文件有一个`tag`或者`tagx`扩展名，也可以包括含有某个公共资源的其他文件。标签文件的`include`文件扩展名为`tagf`。

为了能够正常工作，标签文件必须放在应用程序目录的`WEB-INF/tags`目录下，或者放在它下面的某个子目录下。像标签处理器一样，标签文件也可以打包成jar文件。

从一个标签文件内部可以访问到很多隐式对象。可以通过Script或者EL表达式来访问这些对象。表7-1中列出了可以通过标签文件访问到的隐式对象。这些隐式对象与JSP隐式对象（详情请查看第3章的内容）类似。

表7-1 标签文件中的隐式对象

对 象	类 型
request	<code>javax.servlet.http.HttpServletRequest</code>
response	<code>javax.servlet.http.HttpServletResponse</code>

(续)

对 象	类 型
out	javax.servlet.jsp.JspWriter
session	javax.servlet.http.HttpSession
application	javax.servlet.ServletContext
config	javax.servlet.ServletConfig
jspContext	javax.servlet.jsp.JspContext

## 7.2 我们的第一个标签文件

本节的目的是告诉我们，编写和使用标签文件是一件多么容易的事情！范例由一个标签文件和使用该标签文件的一个 JSP 页面组成。应用程序的目录结构如图 7-1 所示。

这个例子中的标签文件名为 `firstTag.tag`，如代码清单 7-1 所示。



代码清单 7-1 firstTag.tag 文件

---

```

<%@ tag import="java.util.Date" import="java.text.DateFormat" %>
<%
    DateFormat dateFormat =
        DateFormat.getDateInstance(DateFormat.LONG);
    Date now = new Date(System.currentTimeMillis());
    out.println(dateFormat.format(now));
%>
  
```

---

如代码清单 7-1 所示，标签文件就像一个 JSP 页面。`firstTag.tag` 文件中包含了一个 `tag` 指令，它带有两个 `import` 属性和一个 `scriptlet`。这个标签文件的输出结果为 long 格式的当前日期。要想用这个标签文件作为标签扩展，只需将它保存在应用程序的 `WEB-INF/tags` 目录下即可。标签文件名很重要，因为它也表明了标签的名称。`firstTag.tag` 文件的标签名称为 `firstTag`。

代码清单 7-2 展示了一个使用 `firstTag.tag` 文件的 `firstTagTest.jsp` 页面。

代码清单 7-2 firstTagTest.jsp 页面

---

```

<%@ taglib prefix="easy" tagdir="/WEB-INF/tags" %>
Today is <easy:firstTag/>
  
```

---

利用下面这个 URL 调用 `firstTagTest.jsp` 页面：

`http://localhost:8080/app07a/firstTagTest.jsp`

## 7.3 标签文件指令

像 JSP 页面一样，标签文件可以利用指令来控制 JSP 容器如何编译和转换标签文件。标签文件指令的语法与 JSP 指令的语法一样：

```
<%@ directive (attribute="value")* %>
```

星号 (\*) 表示括号中的内容可以不重复，也可以重复多次。这个语法也可以用一种比较随意的方式进行改写：

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

其中的属性必须用单引号或者双引号括起来，起始符号 <%@ 后面的空格和结束符号 %> 前面的空格是可选的，不过使用空格可以提升可读性。

除了 page 之外，所有 JSP 指令在标签文件中都可以使用。这里不用 page，而是用 tag 指令。而且，在标签文件中，还有另外两个指令可以使用：attribute 和 variable。表 7-2 列出了可以在标签文件中使用的所有指令。

表 7-2 标签文件指令

指令	描述
tag	该指令与 JSP 页面的 page 指令相似
include	用这个指令包含来自标签文件的其他资源
taglib	用这个指令从标签文件内部使用定制标签类库
attribute	用这个指令在标签文件中声明一个属性
variable	用这个指令定义一个可以暴露给在调用 JSP 页面的变量

下面将分别介绍这些指令。

### 7.3.1 tag 指令

tag 指令与 JSP 页面中使用的 page 指令相似，下面是 tag 指令的语法：

```
<%@ tag (attribute="value")* %>
```

以上语法用非正式的方式可以表示为：

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

tag 指令的属性如表 7-3 所示。这些属性全部都是可选的。

表 7-3 标签的属性

属性	描述
display-name	通过 XML 工具显示的简称，默认值为标签文件名，没有 tag 扩展名
body-content	关于这个标签主体内容的信息，它的值可以为 empty、tagdependent 或 scriptless（默认）

(续)

属性	描述
dynamic-attributes	表明对动态属性的支持。它的值表示一个放置 Map 的有界属性，其中包含了这个调用期间传递的动态属性名称和值
small-icon	XML 工具要用到的小图片文件相对于 context 的路径，或者相对于标签资源文件的路径。我们通常不使用这个属性
large-icon	包含 XML 工具要用到的大图标图片文件相对于 context 的路径，或者相对于标签资源文件的路径。我们通常也不使用这个属性
description	描述该标签的一个字符串
example	该动作指令用法范例的一个非正式描述
language	标签文件中使用的脚本语言。该属性值在当前 JSP 版本中必须为 “java”
import	用于导入 Java 类型，与 page 指令中的 import 属性相同
pageEncoding	描述该标签文件的字符编码，这个值的形式为 “CHARSET”，必须是字符编码的 IANA 名称。该属性与 page 指令的 pageEncoding 属性相同
isELIgnored	表明忽略还是运算 EL 表达式，该属性的默认值为 “false”，意为运算 EL 表达式。该属性与 page 指令的 isELIgnored 属性相同

除了 import 属性之外，所有其他属性在同一个标签文件的一个 tag 指令或者多个 tag 指令中都只能出现一次。例如，以下 tag 文件就是无效的，因为 body-content 属性不止一次地出现在多个 tag 指令中：

```
<%@ tag display-name="first tag file" body-content="scriptless" %>
<%@ tag body-content="empty" %>
```

下面则是一个有效的 tag 指令，即使 import 属性出现了两次。这是因为 import 可以根据需要出现任意次数。

```
<%@ tag import="java.util.ArrayList" import="java.util.Iterator" %>
```

下面的指令也是有效的：

```
<%@ tag body-content="empty" import="java.util.Enumeration" %>
<%@ tag import="java.sql.*" %>
```

### 7.3.2 include 指令

标签文件的 include 指令与 JSP 页面的 include 指令相同。利用这个指令可以将其他文件的内容包含在当前的标签文件中。如果有一个公共资源要在多个标签文件中使用，这个指令就非常有用。所包含的资源可以是静态的（如 HTML 文件），也可以是动态的（如另一个标签文件）。

举个例子，在代码清单 7-3 的 includeDemoTag.tag 页面中，展示了一个包含一个静态资源（included.html）和一个动态资源（included.tagf）的标签文件。

---

代码清单 7-3 includeDemoTag.tag 文件

---

```
This tag file shows the use of the include directive.  
The first include directive demonstrates how you can include  
a static resource called included.html.  
<br/>  
Here is the content of included.html:  
<%@ include file="included.html" %>  
<br/>  
<br/>  
The second include directive includes another dynamic resource:  
included.tagf.  
<br/>  
<%@ include file="included.tagf" %>
```

---

included.html 和 included.tagf 文件分别放在代码清单 7-4 和代码清单 7-5 中，这两个文件都保存在与标签文件相同的目录下。

注意，标签文件的 segment 建议使用扩展名 tagf。

---

代码清单 7-4 included.html 文件

---

```
<table>  
<tr>  
    <td><b>Menu</b></td>  
</tr>  
<tr>  
    <td>CDs</td>  
</tr>  
<tr>  
    <td>DVDs</td>  
</tr>  
<tr>  
    <td>Others</td>  
</tr>  
</table>
```

---



---

代码清单 7-5 included.tagf 文件

---

```
<%  
    out.print("Hello from included.tagf");  
%>
```

---

要想测试 includeDemoTag.tag 文件，需用代码清单 7-6 中的 includeDemoTagTest.jsp 页面。

---

代码清单 7-6 includeDemoTagTest.jsp 页面

---

```
<%@ taglib prefix="easy" tagdir="/WEB-INF/tags" %>  
<easy:includeDemoTag/>
```

---

利用下面的 URL 可以调用 includeDemoTagTest.jsp 页面：

`http://localhost:8080/app07a/includeDemoTagTest.jsp`

调用结果如图 7-2 所示。

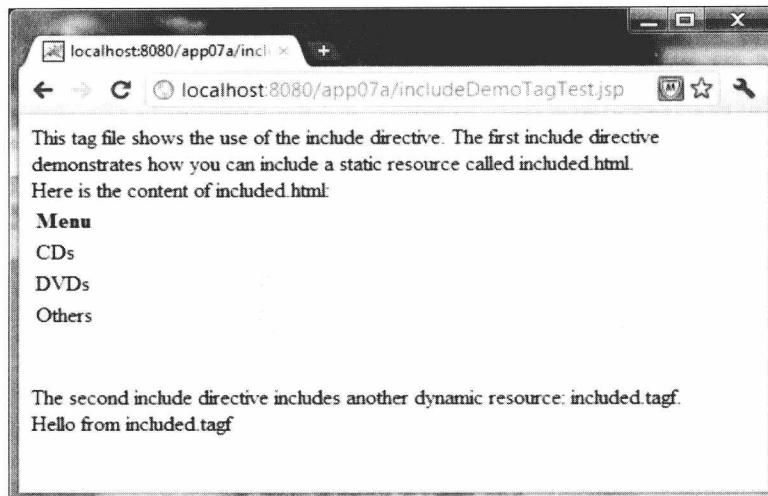


图 7-2 包含来自另一个标签文件的其他资源

关于 `include` 指令的更多信息，请查看第 3 章的相关内容。

### 7.3.3 taglib 指令

利用 `taglib` 指令可以使用标签文件中的定制标签。`taglib` 指令的语法如下：

```
<%@ taglib uri="tagLibraryURI" prefix="tagPrefix" %>
```

`uri` 属性指定一个绝对或相对的 URI，用来唯一标识与这个前缀相关的标签类库描述符。

`prefix` 属性定义一个字符串，它将成为用于区分某个定制动作指令的前缀。

有了 `taglib` 指令，就可以通过以下格式使用不带内容主体的定制标签：

```
<prefix:tagName/>
```

或者通过以下格式使用带有内容主体的定制标签：

```
<prefix:tagName>body</prefix:tagName>
```

标签文件中的 `taglib` 指令与 JSP 页面中的 `taglib` 指令相同。

举个例子，如代码清单 7-7 中的 `taglibDemo.tag` 文件。

**代码清单 7-7 taglibDemo.tag 文件**


---

```
<%@ taglib prefix="simple" tagdir="/WEB-INF/tags" %>
The server's date: <simple:firstTag/>
```

---

它利用代码清单 7-1 中的 `firstTag.tag` 文件来显示服务器的日期。`taglibDemo.tag` 文件被用在代码清单 7-8 中的 `taglibDemoTest.jsp` 页面中。

**代码清单 7-8 taglibDemoTest.jsp 页面**


---

```
<%@ taglib prefix="easy" tagdir="/WEB-INF/tags" %>
<easy:taglibDemo/>
```

---

利用下面这个 URL 可以调用这个 JSP 页面：

`http://localhost:8080/app07a/taglibDemoTest.jsp`

### 7.3.4 attribute 指令

`attribute` 指令支持在标签文件中使用属性，它相当于标签类库描述符中的 `attribute` 元素。`attribute` 指令的语法如下：

```
<%@ attribute (attribute="value")* %>
```

上述语法也可以用一种非正式的形式进行表达，如：

```
<%@ attribute attributel="value1" attribute2="value2" ... %>
```

`attribute` 指令的属性如表 7-4 所示，其中只有 `name` 属性是必需的。

例如，代码清单 7-9 中的 `encode.tag` 文件可以用于对某个字符串进行 HTML 编码。这个 `encode` 标签定义了一个 `input` 属性，其类型为 `java.lang.String`。

**表 7-4 attribute 的属性**

属性	描述
<code>name</code>	该标签文件能够接受的属性名称， <code>name</code> 属性值必须在整个当前标签文件中都是唯一的
<code>required</code>	表明该属性是否是必需的，它的值可以是 <code>true</code> 或 <code>false</code> （默认值）
<code>fragment</code>	表明该属性是一个要通过标签处理器进行运算的部分，或在传给标签处理器之前通过容器进行运算的一个普通属性。它的值为 <code>true</code> 或者 <code>false</code> （默认值）。如果该属性要通过标签处理器进行运算，则值为 <code>true</code>
<code>rtexprvalue</code>	指定该属性值是否可以在运行时通过一个 scriptlet 表达式进行动态计算。这个值为 <code>true</code> （默认值）或者 <code>false</code>
<code>type</code>	属性值的类型，默认为 <code>java.lang.String</code>
<code>description</code>	这个属性的描述

**代码清单 7-9 encode.tag 文件**


---

```
<%@ attribute name="input" required="true" %>
<%!
```

---

```

private String encodeHtmlTag(String tag) {
    if (tag==null) {
        return null;
    }
    int length = tag.length();
    StringBuilder encodedTag = new StringBuilder(2 * length);
    for (int i=0; i<length; i++) {
        char c = tag.charAt(i);
        if (c=='<') {
            encodedTag.append("&lt;");
        } else if (c=='>') {
            encodedTag.append("&gt;");
        } else if (c=='&') {
            encodedTag.append("&amp;");
        } else if (c=='"') {
            encodedTag.append("&quot;");
        } else if (c==' ') {
            encodedTag.append("&nbsp;");
        } else {
            encodedTag.append(c);
        }
    }
    return encodedTag.toString();
}
%>
<%=encodeHtmlTag(input)%>

```

要测试 `encode.tag` 文件，可以使用代码清单 7-10 中的 `encodeTagTest.jsp` 文件。

#### 代码清单 7-10 encodeTagTest.jsp 页面

```

<%@ taglib prefix="easy" tagdir="/WEB-INF/tags" %>
<easy:encode input="  
 means changing line"/>

```

利用以下 URL 可以调用 `encodeTagTest.jsp` 页面：

`http://localhost:8080/app07a/encodeTagTest.jsp`

调用结果如图 7-3 所示。

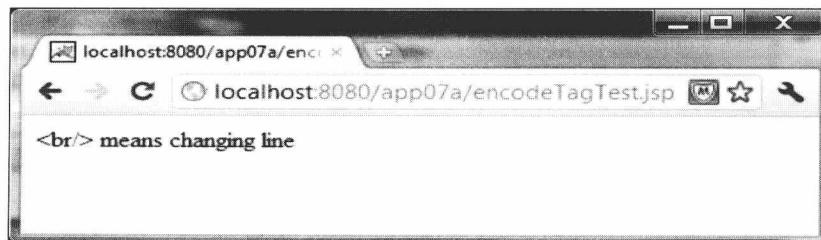


图 7-3 在标签文件中使用属性

### 7.3.5 variable 指令

有时候，将标签文件中的值暴露给在调用的 JSP 页面很有帮助。这可以通过在标签文件中使用 **variable** 指令来实现。在标签文件中，**variable** 指令类似于标签类库描述符中的 **variable** 元素，并且定义了标签处理器中可以通过在调用 JSP 页面访问到的某个变量的细节。由于一个标签文件可以有多个 **variable** 指令，因此可以给在调用的 JSP 页面提供多个值。下面将 **variable** 与用来从 JSP 页面传递值给标签文件的 **attribute** 指令做个比较。

**variable** 指令的语法如下：

```
<%@ variable (attribute="value")* %>
```

非正式形式的表达语法如下：

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

**variable** 指令的属性如表 7-5 所示。

表 7-5 **variable** 的属性

属性	描述
name-given	将要在调用 JSP 页面的脚本语言或者 EL 表达式中使用的变量名称。如果使用 <b>name-from-attribute</b> ，就不能再用 <b>name-given</b> 属性，反之亦然。 <b>name-given</b> 的值不能与该标签文件中的任何属性值相同
name-from-attribute	该属性与 <b>name-given</b> 类似，但是它的值应该是一个属性的名称，并且它在开始进行标签调用之时提供变量名称。如果同时指定 <b>name-given</b> 和 <b>name-from-attribute</b> 属性，或者这两者都没有指定，那么将会产生编译错误
alias	这是一个局部范围的属性，用于存放该变量的值
variable-class	该变量的类型，默认为 <code>java.lang.String</code>
declare	表明是在调用页面中声明该变量，还是在调用之后在标签文件中声明。默认值为 <code>true</code>
scope	所定义脚本变量的范围，其可能值为 <code>AT_BEGIN</code> 、 <code>AT_END</code> 和 <code>NESTED</code> （默认）
description	该变量的描述

你可能会问，如果可以将处理结果直接输出到在调用 JSP 页面的 `JspWriter`，那么为什么还需要 **variable** 指令呢？这是因为，只将 `String` 发送到 `JspWriter`，会导致在调用的 JSP 页面丧失如何使用该结果的灵活性。举个例子，如代码清单 7-1 中的 `firstTag.tag` 文件，它将服务器的当前日期以 `long` 格式输出。如果你还想以 `short` 格式提供服务器的当前日期，那么就必须另外编写一个标签文件。有两个标签文件来完成类似功能时，会增加不必要的维护问题。替代做法是，在标签文件中暴露两个变量：`longDate` 和 `shortDate`。

代码清单 7-11 中的标签文件以两种格式提供了服务器的当前日期：`long` 和 `short`。它有两个变量：`longDate` 和 `shortDate`。

代码清单 7-11 varDemo.tag

---

```
<%@ tag import="java.util.Date" import="java.text.DateFormat"%>
<%@ variable name-given="longDate" %>
<%@ variable name-given="shortDate" %>
<%
```

```

Date now = new Date(System.currentTimeMillis());
DateFormat longFormat =
    DateFormat.getDateInstance(DateFormat.LONG);
DateFormat shortFormat =
    DateFormat.getDateInstance(DateFormat.SHORT);
jspContext.setAttribute("longDate", longFormat.format(now));
jspContext.setAttribute("shortDate", shortFormat.format(now));
%>
<jsp:doBody/>

```

注意，我们在标签的 `JspContext` 中利用 `setAttribute` 方法设置了一个变量。隐式变量 `jspContext` 表示这个对象（详情请见本章稍后的内容）。JSTL（JSP Standard Tag Library）的 `set` 标签提供了这项功能。如果你很熟悉 JSTL，就可以用这个标签代替 `setAttribute` 方法。JSTL 已经在第 5 章做过详细的讨论。

还要注意的是，必须利用标准动作指令 `doBody` 来调用标签主体。更多详情，请查看本章稍后的内容。

我们利用代码清单 7-12 中的 `varDemoTest.jsp` 页面对 `varDemo.tag` 文件做个测试。

代码清单 7-12 varDemoTest.jsp 页面

```

<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
Today's date:
<br/>
<tags:varDemo>
In long format: ${longDate}
<br/>
In short format: ${shortDate}
</tags:varDemo>

```

利用以下 URL 可以调用 `varDemoTest.jsp` 页面：

`http://localhost:8080/app07a/varDemoTest.jsp`

调用结果如图 7-4 所示。

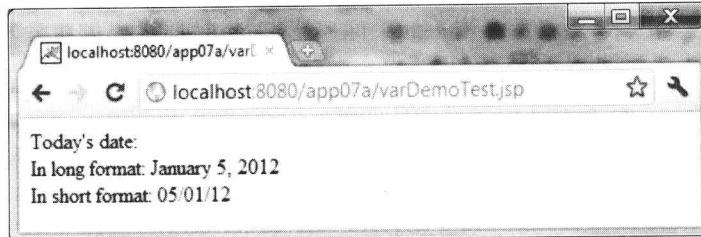


图 7-4 varDemoTest.jsp 的调用结果

有时候，会需要用到变量。再举个例子。假设你需要编写一个定制动作指令，用来从

数据库中获取某个指定产品标识符的产品详细信息。为此，可以用一个 attribute 来设置产品标识符，并用一个变量来保存每一条信息。因此，最终你会有这些变量：name、price、description、imageUrl 等。

## 7.4 doBody

标准动作指令 doBody 只能通过标签文件内部进行使用。我们可以用它调用标签的主体，在代码清单 7-11 的标签文件中已经见识过 doBody 的用法。在本节中，我们要更详细地介绍 doBody。

doBody 动作指令可以带有属性。如果需要将标签调用的结果导入到某个变量中，就可以使用这些属性。如果使用 doBody 时不带属性，那么标准动作指令 doBody 就会将输出结果写入在所调用 JSP 页面的 JspWriter 中。

标准动作指令 doBody 的属性如表 7-6 所示，这些属性全部都是可选的。

表 7-6 doBody 的属性

属性	描述
var	这是一个限域属性名称，用来保存标签主体调用的输出。它的值保存为 java.lang.String。var 和 varReader 二者不能同时存在
varReader	这是一个限域属性名称，用来保存标签主体调用的输出。它的值保存为 java.io.Reader。var 和 varReader 二者不能同时存在
scope	结果变量的范围

下面的例子中展示了如何利用 doBody 调用标签主体，以及如何将输出结果保存在范围为 session 的变量 referer 中。假设你有一个卖玩具的网站，并在众多搜索引擎上给你的网站做了大量的广告。你当然就想知道哪一个搜索引擎重定向了最终达成交易的绝大多数访问流量。为此，你可以记录 Web 应用程序首页的 referer 标头。你可以用一个标签文件将 referer 标头的值保存为 session 属性。之后，如果用户决定购买某件商品，你就可以获得该 session 属性值，并将它插入到一个数据库表中。

这个例子中包含了一个 HTML 文件（searchEngine.html）、两个 JSP 页面（main.jsp 和 viewReferer.jsp），还有一个标签文件（doBodyDemo.tag）。main.jsp 页面是这个网站的首页。它利用 doBodyDemo 定制标签来保存 referer 标头。要想预览 referer 标头值，可以利用 viewReferer.jsp 页面。如果直接通过输入其 URL 来调用 main.jsp 页面，那么 referer 标头将会为 null。因此，必须利用 searchEngine.html 文件进入到 main.jsp 页面。

doBodyDemo.tag 文件如代码清单 7-13 所示。

代码清单 7-13 doBodyDemo.tag

---

```
<jsp:doBody var="referer" scope="session"/>
```

---

没错, doBodyDemo.tag 文件中只有一行代码, 即一个 doBody 标准动作指令, 其作用是调用标签主体, 并将输出结果保存在一个名为 referer 的 session 属性中。

main.jsp 页面如代码清单 7-14 所示。

代码清单 7-14 main.jsp 页面

---

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
Your referer header: ${header.referer}
<br/>
<tags:doBodyDemo>
    ${header.referer}
</tags:doBodyDemo>
<a href="viewReferer.jsp">View</a> the referer as a Session
attribute.
```

---

main.jsp 页面利用文本和一个 EL 表达式输出 referer 标头的值:

```
Your referer header: ${header.referer}
<br/>
```

之后, 它利用 doBodyDemo 标签, 将 referer 标头作为主体传递:

```
<tags:doBodyDemo>
    ${header.referer}
</tags:doBodyDemo>
```

接下来, 为了让你更方便一些, 它还打印出了 viewReferer.jsp 页面的链接:

```
<a href="viewReferer.jsp">View</a> the referer as a Session
attribute.
```

viewReferer.jsp 页面如代码清单 7-15 所示。

代码清单 7-15 viewReferer.jsp 页面

---

```
The referer header of the previous page is ${sessionScope.referer}
```

---

viewReferer.jsp 页面利用一个 EL 表达式打印出了 session 属性 referer 的值。

最后, searchEngine.html 如代码清单 7-16 所示。

代码清单 7-16 searchEnginer.html 文件

---

```
Please click <a href="main.jsp">here</a>
```

---

检验这个例子时, 首先要利用下面这个 URL 调用 searchEngine.html 文件:

<http://localhost:8080/app07a/searchEngine.html>

你将会看到如图 7-5 所示的 searchEngine.html 页面。

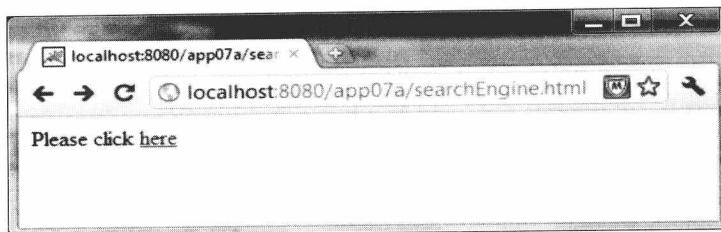


图 7-5 searchEngine.html 页面

现在，单击链接进入 main.jsp 页面，main.jsp 页面的 `referer` 标头就是 `searchEngine.html` 的 URL。main.jsp 页面的内容如图 7-6 所示。

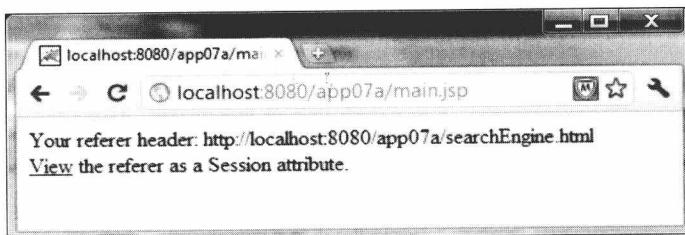


图 7-6 main.jsp 页面

main.jsp 页面调用 `doBodyDemo` 定制动作指令，以保存 session 属性 `referer`。现在，单击 main.jsp 页面中的 `view` 链接，查看一下 session 属性值，你将会看到如图 7-7 所示的内容。

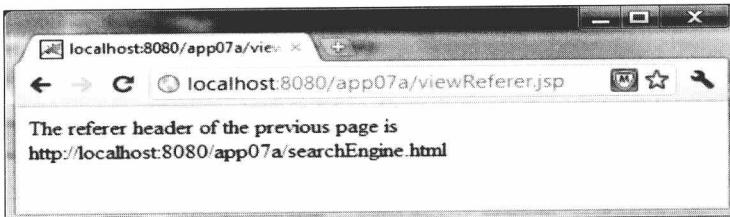


图 7-7 viewReferer.jsp 页面

## 7.5 invoke

`invoke` 标准动作指令与 `doBody` 相似，可以在标签文件中用来调用一个 `fragment` 属性。前面说过，属性可以带有一个 `fragment` 属性，它的值为 `true` 或 `false`。如果 `fragment` 属性值为 `true`，那么该属性就是一个 `fragment` 属性，你可以根据需要通过标签文件对它进行多次调用。`invoke` 也可以带有属性。`invoke` 的属性如表 7-7 所示。注意，只有 `fragment` 属性是必需的。

表 7-7 invoke 的属性

属性	描述
fragment	在这个标签调用期间用于标识该 fragment 的名称
var	这是一个限域属性名称，用来保存标签主体调用的输出。它的值保存为 java.lang.String。 var 或 varReader 属性二者不能同时存在
varReader	这是一个限域属性名称，用来保存标签主体调用的输出。它的值保存为 java.io.Reader。var 或 varReader 属性二者不能同时存在
scope	结果变量的范围

举个例子，请看代码清单 7-17 中的 invokeDemo.tag 文件。

代码清单 7-17 invokeDemo.tag 文件

```
<%@ attribute name="productDetails" fragment="true" %>
<%@ variable name-given="productName" %>
<%@ variable name-given="description" %>
<%@ variable name-given="price" %>
<%
    jspContext.setAttribute("productName", "Pelesonic DVD Player");
    jspContext.setAttribute("description",
        "Dolby Digital output through coaxial digital-audio jack," +
        " 500 lines horizontal resolution-image digest viewing");
    jspContext.setAttribute("price", "65");
%>
<jsp:invoke fragment="productDetails"/>
```

invokeDemo.tag 文件利用了 attribute 指令，它的 fragment 属性值设为 true。它还定义了三个变量，并为这些变量设定了值。标签文件的最后一行调用 fragment 属性 productDetails。由于 invoke 标准动作指令中没有 var 或 varReader 属性，因此标签调用的结果将被定向到所调用 JSP 页面的 JspWriter。

测试这个标签文件时，可以利用代码清单 7-18 中的 invokeTest.jsp 页面。

代码清单 7-18 invokeTest.jsp 页面

```
<%@ taglib prefix="easy" tagdir="/WEB-INF/tags" %>
<html>
<head>
<title>Product Details</title>
</head>
<body>
<easy:invokeDemo>
    <jsp:attribute name="productDetails">
        <table width="220" border="1">
            <tr>
                <td><b>Product Name</b></td>
                <td>${productName}</td>
            </tr>
            <tr>
                <td><b>Description</b></td>
```

```
<td>${description}</td>
</tr>
<tr>
    <td><b>Price</b></td>
    <td>${price}</td>
</tr>
</table>
</jsp:attribute>
</easy:invokeDemo>
</body>
</html>
```

我们可以利用下面的 URL 来调用 invokeTest.jsp 页面。

<http://localhost:8080/app07a/invokeTest.jsp>

其结果如图 7-8 所示。

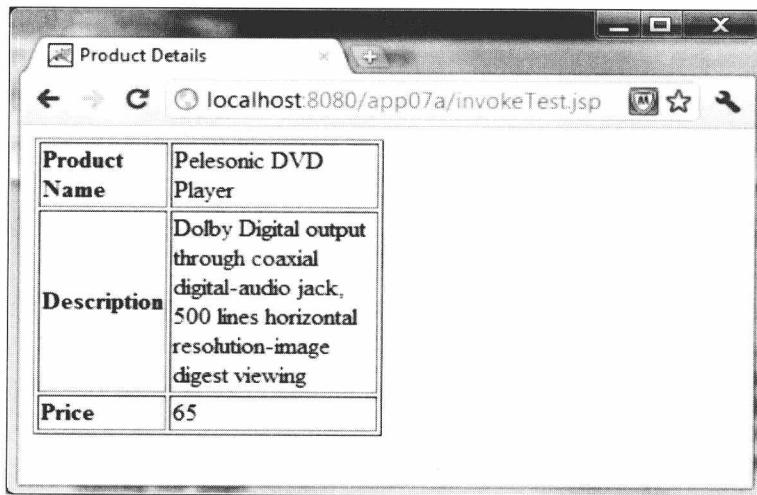


图 7-8 使用 fragment 属性

## 7.6 小结

本章讲解了标签文件，以及它们如何使编写标签扩展变得更加简单。有了标签文件，就不需要标签类库描述符，甚至不需要编译标签处理器。本章还介绍了如何使用 invoke 和 doBody 标准动作指令。

# 第 8 章 监 听 器

为了能够在 Servlet/JSP 应用程序中进行事件驱动编程 (Event-Driven Programming), Servlet API 提供了一整套事件类和监听器接口。所有事件类均源自 `java.util.Event`, 并且监听器在以下三个不同级别中均可用: `ServletContext`、`HttpSession` 及 `ServletRequest`。

本章将示范如何在 Servlet/JSP 应用程序中编写和使用监听器。Servlet 3.0 中新增了一个监听器接口: `javax.servlet.AsyncListener`, 详情将在第 14 章中讨论。

## 8.1 监听器接口和注册

创建监听器的监听器接口属于 `javax.servlet` 和 `javax.servlet.http` 包的一部分, 详情如下:

- `javax.servlet.ServletContextListener`。这是对 Servlet Context 生命周期事件做出响应的监听器。创建好 Servlet Context 时马上会调用它的其中一个方法, 并在关闭 Servlet Context 之前调用它的另一个方法。
- `javax.servlet.ServletContextAttributeListener`。这是在添加、删除或替换某个 Servlet Context 属性时采取相应动作的监听器。
- `javax.servlet.http.HttpSessionAttributeListener`。这是在创建、移除或替换 Servlet 上下文属性时响应的监听器。
- `javax.servlet.http.HttpSessionAttributeListener`。这是在添加、删除或替换某个 session 属性时被调用的监听器。
- `javax.servlet.httpSessionActivationListener`。这是在打开和关闭某个 HttpSession 时被调用的监听器。
- `javax.servlet.http.HttpSessionBindingListener`。这是一个类, 其实例将被保存为可以实现这个接口的 HttpSession 属性。当它在 HttpSession 中被添加或者删除时, 实现 HttpSessionBindingListener 的类实例会收到通知。
- `javax.servlet.ServletRequestListener`。这是对 ServletRequest 的创建和删除做出响应的监听器。
- `javax.servlet.ServletRequestAttributeListener`。当 ServletRequest 中添加、删除或替换掉某个属性时, 会调用该监听器的方法。
- `javax.servlet.AsyncListener`。用于异步操作的监听器, 详情请查看第 14 章的相关内容。

创建监听器时，只要创建一个实现相关接口的 Java 类即可。在 Servlet 3.0 中，注册监听器有两种方法，以便 Servlet 容器能够认出来。第一种方法是像下面这样使用 `WebListener` 注解：

```
@WebListener
public class ListenerClass implements ListenerInterface {
}
```

注册监听器的第二种方法是在部署描述符中使用一个 `listener` 元素：

```
</listener>
<listener-class>fully-qualified listener class</listener-class>
</listener>
```

在应用程序中可以想要多少个监听器就可以有多少个监听器。注意，对监听器方法的调用是同步进行的。

## 8.2 Servlet Context 监听器

在 `ServletContext` 级别上有两个监听器接口：`ServletContextListener` 和 `ServletContextAttributeListener`，接下来要对它们做详细的讲解。

### 8.2.1 ServletContextListener

`ServletContextListener` 会对 `ServletContext` 的初始化和解构做出响应。`ServletContext` 被初始化时，Servlet 容器会在所有已注册的 `ServletContextListener` 中调用 `contextInitialized` 方法，其方法签名如下：

```
void contextInitialized(ServletContextEvent event)
```

当 `ServletContext` 要被解构和销毁时，Servlet 容器会在所有已注册的 `ServletContextListener` 中调用 `contextDestroyed` 方法，以下是 `contextDestroyed` 的方法签名：

```
void contextDestroyed(ServletContextEvent event)
```

`contextInitialized` 和 `contextDestroyed` 都会收到一个来自 Servlet 容器的 `ServletContextEvent`。`java.util.EventObject` 类的一个派生类：`javax.servlet.ServletContextEvent`，定义了一个返回 `ServletContext` 的 `getServletContext` 方法：

```
ServletContext getServletContext()
```

这个方法很重要，因为这是访问 `ServletContext` 的唯一简便方法。它有许多 Servlet-

`ContextListener`, 可以将属性保存在 `ServletContext` 中。

举个例子, 请看本书配套的 `app08a` 范例应用程序。代码清单 8-1 中的 `AppListener` 类是一个 `ServletContextListener`, 一旦 `ServletContext` 被初始化, 它会立即创建一个包含国家代码和名称的 `Map`, 并作为 `ServletContext` 属性。

代码清单 8-1 `AppListener` 类

---

```
package app08a.listener;
import java.util.HashMap;
import java.util.Map;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;

@WebListener
public class AppListener implements ServletContextListener {
    @Override
    public void contextDestroyed(ServletContextEvent sce) {
    }

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext servletContext = sce.getServletContext();

        Map<String, String> countries =
            new HashMap<String, String>();
        countries.put("ca", "Canada");
        countries.put("us", "United States");
        servletContext.setAttribute("countries", countries);
    }
}
```

---

注意代码清单 8-1 中的 `contextInitialized` 方法实现。它首先在 Servlet 容器传递的 `ServletContextEvent` 中调用 `getServletContext` 方法, 随后创建一个 `Map`, 并填入两个国家, 用 `Map` 作为 `ServletContext` 属性。在现实的应用程序中, 保存在 `ServletContext` 中的数据则可能来自某一个数据库。

测试这个监听器时, 可以利用代码清单 8-2 中的 `countries.jsp` 页面。

代码清单 8-2 `countries.jsp` 页面

---

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
<title>Country List</title>
</head>
<body>
We operate in these countries:
```

```

<ul>
    <c:forEach items="${countries}" var="country">
        <li>${country.value}</li>
    </c:forEach>
</ul>
</body>
</html>

```

`countries.jsp` 页面利用 JSTL 的 `forEach` 标签迭代 `countries` map。注意，要想让这个范例应用程序正常运行，在 `app08a` 应用程序的 `WEB-INF/lib` 目录下必须要有 JSTL 类库。

当你将 JSTL 类库复制到 `lib` 目录下之后，重启你的 Servlet 容器，并在浏览器中打开下面这个 URL：

`http://localhost:8080/app08a/countries`

你将会看到如图 8-1 所示的画面。

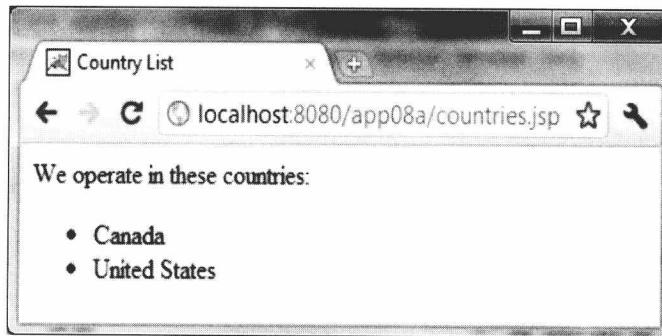


图 8-1 利用 `ServletContextListener` 加载初始值

### 8.2.2 `ServletContextAttributeListener`

每当 `ServletContext` 中添加、删除或替换了某个属性时，`ServletContextAttributeListener` 的实现都会收到通知。以下就是在这个监听器接口中定义的三个方法：

```

void attributeAdded(ServletContextAttributeEvent event)
void attributeRemoved(ServletContextAttributeEvent event)
void attributeReplaced(ServletContextAttributeEvent event)

```

每当 `ServletContext` 中添加了某个属性时，Servlet 容器就会调用 `attributeAdded` 方法。每当 `ServletContext` 中删除了某个属性时，则是调用 `attributeRemoved` 方法。每当 `ServletContext` 被新的代替时，则是调用 `attributeReplaced` 方法。所有的监听器方法都会从你获取属性名称和属性值的方法中收到一个 `ServletContextAttributeEvent` 实例。

`ServletContextAttributeEvent` 类派生于 `ServletContextAttribute`，并添加了下面这两个方法，分别用来获取属性名称和属性值：

```
java.lang.String getName()
java.lang.Object getValue()
```

## 8.3 Session 监听器

与 `HttpSession` 有关的监听器接口有 4 个：`HttpSessionListener`、`HttpSessionActivationListener`、`HttpSessionAttributeListener` 和 `HttpSessionBindingListener`。这些接口都是 `javax.servlet.http` 包的成员，下面将进行详细的讲解。

### 8.3.1 HttpSessionListener

当有 `HttpSession` 被创建或者销毁时，Servlet 容器就会调用所有已注册的 `HttpSessionListener`。`HttpSessionListener` 中定义的两个方法是 `sessionCreated` 和 `sessionDestroyed`：

```
void sessionCreated(HttpSessionEvent event)
void sessionDestroyed(HttpSessionEvent event)
```

这两个方法都收到一个 `HttpSessionEvent` 实例，它是 `java.util.Event` 的派生类。我们可以在 `HttpSessionEvent` 中调用 `getSession` 方法以获得所创建或销毁的 `HttpSession`。`getSession` 方法签名如下：

```
HttpSession getSession()
```

举个例子，请看 `app08a` 中的 `SessionListener` 类，如代码清单 8-3 所示。这个监听器提供了应用程序中 `HttpSession` 的数量。它用了一个 `AtomicInteger` 作为计数器，并保存为 `ServletContext` 属性。每当创建 `HttpSession` 时，这个计数器就会递增。每当销毁 `HttpSession` 时，计数器就会递减。因此，它能够提供在读取计数器时具有有效 session 的用户数量快照。用 `AtomicInteger` 代替 `Integer`，是为了确保递增和递减操作的原子性。

代码清单 8-3 SessionListener 类

---

```
package app08a.listener;
import java.util.concurrent.atomic.AtomicInteger;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.annotation.WebListener;
import javax.servlet.http.HttpSession;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;
```

```

@WebListener
public class SessionListener implements HttpSessionListener,
    ServletContextListener {

    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext servletContext = sce.getServletContext();
        servletContext.setAttribute("userCounter",
            new AtomicInteger());
    }

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
    }

    @Override
    public void sessionCreated(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        ServletContext servletContext = session.getServletContext();
        AtomicInteger userCounter = (AtomicInteger) servletContext
            .getAttribute("userCounter");
        int userCount = userCounter.incrementAndGet();
        System.out.println("userCount incremented to :" +
            userCount);
    }
    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        ServletContext servletContext = session.getServletContext();
        AtomicInteger userCounter = (AtomicInteger) servletContext
            .getAttribute("userCounter");
        int userCount = userCounter.decrementAndGet();
        System.out.println("----- userCount decremented to :" +
            userCount);
    }
}

```

如代码清单 8-3 所示，SessionListener 类实现了 ServletContextListener 接口和 HttpSessionListener 接口，因此，这两个接口的方法都要实现。

contextInitialized 方法继承自 ServletContextListener 接口，它创建了一个 AtomicInteger，并将它保存在 ServletContext 中。AtomicInteger 的初始值为 0，表示应用程序刚启动时的用户数为 0。ServletContext 属性的名称为 userCounter。

```

public void contextInitialized(ServletContextEvent sce) {
    ServletContext servletContext = sce.getServletContext();
    servletContext.setAttribute("userCounter",
        new AtomicInteger());
}

```

每当创建 HttpSession 时，都会调用 `sessionCreated` 方法。该方法的任务就是获取所创建的 HttpSession，并从 ServletContext 中获得 `userCounter` 属性。然后在 `userCounter` `AtomicInteger` 中调用 `incrementAndGet` 方法。下面把值打印出来，让你更容易看出监听器是如何工作的：

```
public void sessionCreated(HttpSessionEvent se) {
    HttpSession session = se.getSession();
    ServletContext servletContext = session.getServletContext();
    AtomicInteger userCounter = (AtomicInteger) servletContext
        .getAttribute("userCounter");
    int userCount = userCounter.incrementAndGet();
    System.out.println("userCount incremented to :" +
        userCount);
}
```

销毁 HttpSession 之前调用 `sessionDestroyed` 方法。这个方法实现与 `sessionCreated` 的类似，只不过它是递减而不是递增 `userCounter` 的值。

```
public void sessionDestroyed(HttpSessionEvent se) {
    HttpSession session = se.getSession();
    ServletContext servletContext = session.getServletContext();
    AtomicInteger userCounter = (AtomicInteger) servletContext
        .getAttribute("userCounter");
    int userCount = userCounter.decrementAndGet();
    System.out.println("----- userCount decremented to :"
        + userCount);
}
```

测试监听器时，可以利用不同的浏览器再次请求 `countries.jsp` 页面，看看你的控制台上输出了什么内容。下面是调用 `countries.jsp` 的 URL：

`http://localhost:8080/app08a/countries.jsp`

初次调用将会在控制台上输出以下内容：

`userCount incremented to :1`

从同一个浏览器再次发出请求时，`userCounter` 的值不会发生变化，因为它与同一个 HttpSession 关联。但是，用不同的浏览器调用该页面时，`userCounter` 的值将会增加。

如果你有时间等到 HttpSession 过期，则可以留意一下控制台上再次打印出什么内容。

### 8.3.2 HttpSessionAttributeListener

`HttpSessionAttributeListener` 就像 `ServletContextAttributeListener` 一样，只不过当 HttpSession 中有添加、删除或者替换属性的时候它才会被调用。下面是 `HttpSessionAttributeListener` 接口中定义的方法。

```

void attributeAdded(HttpSessionBindingEvent event)
void attributeRemoved( HttpSessionBindingEvent event)
void attributeReplaced( HttpSessionBindingEvent event)

```

当 HttpSession 中添加某个属性时，由 Servlet 容器调用 attributeAdded 方法。当 HttpSession 中删除了某个属性时，调用 attributeRemoved 方法，当 HttpSession 属性被新属性代替时，则调用 attributeReplaced 方法。所有的监听器方法都会收到一个 HttpSessionBindingEvent 实例，你可以从中获得相应的 HttpSession 和属性值及名称。

```

java.lang.String getName()
java.lang.Object getValue()

```

由于 HttpSessionBindingEvent 是 HttpSessionEvent 的一个子类，因此还可以在 HttpSessionAttributeListener 类中查到受影响的 HttpSession。

### 8.3.3 HttpSessionActivationListener

在分布式环境中，多个 Servlet 容器会配置成可伸缩的，为了节省内存，Servlet 容器可以对 session 属性进行迁移或者序列化。一般来说，当内存比较低时，相对较少访问的对象可以序列化到备用存储设备中，这样，Servlet 容器就能够注意到哪些 session 属性的类实现了 HttpSessionActivationListener 接口。

这个接口中定义了两个方法：sessionDidActivate 和 sessionWillPassivate：

```

void sessionDidActivate(HttpSessionEvent event)
void sessionWillPassivate(HttpSessionEvent event)

```

包含这个对象的 HttpSession 一旦被激活（Activate），就会调用 sessionDidActivate 方法。Servlet 容器传给该方法的 HttpSessionEvent 能够帮你获取到刚刚激活的 HttpSession。

当包含该监听器的 HttpSession 即将被钝化（Passivate）时，会调用 sessionWillPassivate 方法。像 sessionDidActivate 一样，Servlet 容器也会传递一个 HttpSessionEvent 给该方法，以便 session 属性能够在 HttpSession 中使用。

### 8.3.4 HttpSessionBindingListener

当 HttpSessionBindingListener 绑定到 HttpSession，或者取消绑定时，都会收到通知。如果一个类想要知道什么时候绑定或取消绑定到 HttpSession 上，那么这个类要实现 HttpSessionBindingListener 接口，然后将它的实例保存为 session 属性。例如，有个对象的类实现了这个接口，当它保存为 HttpSession 属性时，它就可以自动更新。再比如，一旦 HttpSessionBindingListener 与 HttpSession 取消绑定，它的实现就可以释放占用的资源。

请看代码清单 8-4 中实现 HttpSessionBindingListener 的 Product 类。

代码清单 8-4 实现 HttpSessionBindingListener 的类

```
package app08a.model;
import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionBindingListener;

public class Product implements HttpSessionBindingListener {
    private String id;
    private String name;
    private double price;

    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        this.price = price;
    }

    @Override
    public void valueBound(HttpSessionBindingEvent event) {
        String attributeName = event.getName();
        System.out.println(attributeName + " valueBound");
    }

    @Override
    public void valueUnbound(HttpSessionBindingEvent event) {
        String attributeName = event.getName();
        System.out.println(attributeName + " valueUnbound");
    }
}
```

当监听器与 HttpSession 绑定或取消绑定时，它所做的只是将内容输出到控制台即可。

## 8.4 ServletRequest 监听器

在 `ServletRequest` 级别上有 3 个监听器接口：`ServletRequestListener`、`ServletRequestAttributeListener` 和 `AsyncListener`。前两个接口在本节中讨论，`AsyncListener` 将在第 14 章中讲解。

### 8.4.1 ServletRequestListener

`ServletRequestListener` 对 `ServletRequest` 的创建和销毁做出响应。在 `Servlet` 容器中是通过池来重用 `ServletRequest` 的，创建 `ServletRequest` 花费的时间相当于从池中获取它的时间，`ServletRequest` 销毁时间则相当于它返回到池中的时间。

`ServletRequestListener` 接口定义了两个方法：`requestInitialized` 和 `requestDestroyed`，它们的方法签名如下：

```
void requestInitialized(ServletRequestEvent event)
void requestDestroyed(ServletRequestEvent event)
```

创建（或从池中取出）`ServletRequest` 时会调用 `requestInitialized` 方法，`ServletRequest` 被销毁（或者返回池中）时会调用 `requestDestroyed` 方法。这两个方法都会收到一个 `ServletRequestEvent`，通过调用 `getServletRequest` 方法，可以从中获取到相应的 `ServletRequest` 实例。

```
ServletRequest getServletRequest()
```

此外，`ServletRequestEvent` 接口还定义了返回 `ServletContext` 的 `getServletContext` 方法，其方法签名如下：

```
ServletContext getServletContext()
```

举个例子。请看 `app08a` 中的 `PerfStatListener` 类。这个监听器用来测算从创建 `ServletRequest` 到销毁之间的时间差，从而准确得出执行请求所花费的时间。

代码清单 8-5 中的 `PerfStatListener` 类利用 `ServletRequestListener` 接口来计算完成一个 HTTP 请求要花费的时间。它的实现，依赖于 `Servlet` 容器在请求开始时在 `ServletRequestListener` 中调用 `requestInitialized` 方法，并在处理完之后调用 `requestDestroyed` 方法。通过读取这两个事件的起始时间并进行比较之后，可以得知完成一个 HTTP 请求大约需要花费多少时间。

代码清单 8-5 PerfStatListener

---

```
package app08a.listener;
import javax.servlet.ServletRequest;
```

```

import javax.servlet.ServletRequestEvent;
import javax.servlet.ServletRequestListener;
import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpServletRequest;

@WebListener
public class PerfStatListener implements ServletRequestListener {
    @Override
    public void requestInitialized(ServletRequestEvent sre) {
        ServletRequest servletRequest = sre.getServletRequest();
        servletRequest.setAttribute("start", System.nanoTime());
    }

    @Override
    public void requestDestroyed(ServletRequestEvent sre) {
        ServletRequest servletRequest = sre.getServletRequest();
        Long start = (Long) servletRequest.getAttribute("start");
        Long end = System.nanoTime();
        HttpServletRequest httpServletRequest =
            (HttpServletRequest) servletRequest;
        String uri = httpServletRequest.getRequestURI();
        System.out.println("time taken to execute " + uri +
                           ":" + ((end - start) / 1000) + "microseconds");
    }
}

```

---

代码清单 8-5 中的 `requestInitialized` 方法调用 `System.nanoTime()`, 并将返回值（将被装箱成 `Long`）当作 `ServletRequest` 属性。

```

public void requestInitialized(ServletRequestEvent sre) {
    ServletRequest servletRequest = sre.getServletRequest();
    servletRequest.setAttribute("start", System.nanoTime());
}

```

`nanoTime` 方法返回一个表示任意时间的 `long`。这个返回值与任何系统概念或者壁钟的时间无关，但是在同一台 JVM 中得到的两个返回值，最适合用来计算第一个 `nanoTime` 调用和第二个调用之间过了多长时间。

或许你也猜到了，没错，正是 `requestDestroyed` 方法再次调用 `nanoTime` 方法，并将它的值与第一个值进行比较。

```

public void requestDestroyed(ServletRequestEvent sre) {
    ServletRequest servletRequest = sre.getServletRequest();
    Long start = (Long) servletRequest.getAttribute("start");
    Long end = System.nanoTime();
    HttpServletRequest httpServletRequest =
        (HttpServletRequest) servletRequest;
    String uri = httpServletRequest.getRequestURI();
    System.out.println("time taken to execute " + uri +

```

```
    ":" + ((end - start) / 1000) + "microseconds");  
}
```

测试 PerfStatListener 时，要再次调用 app08a 中的 countries.jsp 页面。

#### 8.4.2 ServletRequestAttributeListener

当在 `ServletRequest` 中添加、删除或者替换某个属性时，会调用 `ServletRequestAttributeListener`。`ServletRequestAttributeListener` 接口中定义了 3 个方法：`attributeAdded`、`attributeReplaced` 和 `attributeRemoved`，这些方法的签名如下：

```
void attributeAdded(ServletRequestAttributeEvent event)  
void attributeRemoved(ServletRequestAttributeEvent event)  
void attributeReplaced(ServletRequestAttributeEvent event)
```

所有方法都会收到一个 `ServletRequestAttributeEvent`（这是 `ServletRequestEvent` 的一个子类）实例。`ServletRequestAttributeEvent` 类通过 `getName` 和 `getValue` 方法暴露有关的属性：

```
java.lang.String getName()  
java.lang.Object getValue()
```

### 8.5 小结

本章学习了 `Servlet API` 中的几种监听器类型。这些监听器属于以下三种范围之一：`application`、`session` 或 `request`。让监听器生效也很简单，只需要一个简单的注册过程。注册监听器时，可以利用 `@WebListener` 标注实现类，也可以在部署描述符中使用 `listener` 元素。

`Servlet 3.0` 中还新增了一个监听器接口：`javax.servlet.AsyncListener`，这个将在第 14 章中讲解。

# 第9章 过滤器

过滤器（filter）是指拦截请求，并对传给被请求资源的 `ServletRequest` 或 `ServletResponse` 进行处理的一个对象。过滤器可以用于登录、加密和解密、会话检查、图片转换，等等。过滤器可以配置为拦截一个或多个资源。

过滤器配置可以通过注解或者部署描述符来完成。如果同一个资源或同一组资源中应用了多个过滤器，则其调用顺序有时候显得比较重要，这时候就需要用到部署描述符。

本章将介绍如何编写和注册过滤器。下面举几个例子。

## 9.1 Filter API

接下来讲解过滤器中使用的接口，包括 `Filter`、`FilterConfig` 和 `FilterChain`。

过滤器类必须实现 `javax.servlet.Filter` 接口。这个接口为每个过滤器暴露 3 个生命周期方法：`init`、`doFilter` 和 `destroy`。

当过滤器启动服务时，比如应用程序启动时，Servlet 容器就会调用 `init` 方法。换句话说，不用等到调用与被过滤器相关的资源之后，才调用 `init` 方法。这个方法只调用一次，并且应该包含该过滤器的初始化代码。`init` 方法的签名如下：

```
void init(FilterConfig filterConfig)
```

注意，Servlet 容器给 `init` 方法传递了一个 `FilterConfig`。下面介绍 `FilterConfig` 接口。

每次调用与过滤器相关的资源时，Servlet 容器都会调用 `Filter` 实例的 `doFilter` 方法。该方法会收到一个 `ServletRequest`、`ServletResponse` 和 `FilterChain`。

下面是 `doFilter` 方法的签名：

```
void doFilter(ServletRequest request, ServletResponse response,
              FilterChain filterChain)
```

如你所见，`doFilter` 的实现可以访问 `ServletRequest` 和 `ServletResponse`。因此，可以在 `ServletRequest` 中添加属性，或者在 `ServletResponse` 中添加一个标头。甚至可以对 `ServletRequest` 或者 `ServletResponse` 进行修饰，改变它们的行为，详情请查看第 13 章的内容。

`doFilter` 方法实现中的最后一行代码应该是调用 `FilterChain`（作为 `doFilter` 方法的第三个参数）中的 `doChain` 方法：

```
filterChain.doFilter(request, response)
```

一个资源可以与多个过滤器（更专业地说，是一条过滤器链）关联，`FilterChain.doFilter()`通常会引发调用链中的下一个过滤器被调用。在链中的最后一个过滤器中调用`FilterChain.doFilter()`会引发资源本身被调用。

如果你没有在`Filter.doFilter()`方法实现代码的最后调用`FilterChain.doFilter()`方法，那么程序的处理将会在这个地方停止，并且不会调用请求。

注意，`doFilter`方法是`FilterChain`接口中唯一的方法。它与`Filter`中的`doFilter`方法稍有不同。在`FilterChain`中，`doFilter`只有两个参数，而不是三个。

`Filter`中的最后一个生命周期方法是`destroy`，其方法签名如下：

```
void destroy()
```

这个方法在过滤器即将终止服务之前，由`Servlet`容器调用，一般发生在应用程序停止的时候。

除非一个过滤器类在部署描述符的多个`filter`元素中进行了声明，否则`Servlet`容器将只给每一类过滤器创建一个实例。由于`Servlet/JSP`应用程序通常是多用户的应用程序，因此可以同时通过多个线程访问一个过滤器实例，但你必须谨慎处理好多线程的问题。关于如何处理线程安全的范例，请查看本章稍后的例子。

## 9.2 过滤器的配置

编写好过滤器类之后，还需要对它进行配置。配置过滤器的目标如下：

- 确定过滤器要拦截哪些资源。
- 要传给过滤器`init`方法的启动初始值。
- 给过滤器起个名字。给过滤器起名在大多数时候似乎没有什么用处，但是有时候很有帮助。例如，可以记录过滤器启动的时间，或者假如同一个应用程序中有多个过滤器，它还可以帮助你查看过滤器的调用顺序。

`FilterConfig`接口允许通过其`getServletContext`方法访问`ServletContext`：

```
ServletContext getServletContext()
```

如果过滤器有了名字，`FilterConfig`中的`getFilterName`方法就会返回它的名称。下面是`getFilterName`的方法签名。

```
java.lang.String getFilterName()
```

但是，你最关注的应该是获取初始参数，这是开发者或者部署者传给过滤器的初始值。处理初始参数有两个方法，其中一个是`getParameterNames`：

```
java.util.Enumeration<java.lang.String> getInitParameterNames()
```

该方法返回过滤器参数名称的一个 Enumeration。如果过滤器没有参数，那么该方法将会返回一个空的 Enumeration。

处理初始参数的第二个方法是 getParameter：

```
java.lang.String getInitParameter(java.lang.String parameterName)
```

配置过滤器有两种方法。一种是利用 WebFilter 注解类型，或者通过在部署描述符中注册它，来配置过滤器。使用 @WebFilter 很容易，因为只需对过滤器类进行标注，并且不需要部署描述符。但是修改配置设置时，就必须重新编译过滤器类。另一方面，在部署描述符中配置过滤器意味着，要修改配置值就必须编辑文本文件。

要使用 @WebFilter，需要熟悉它的属性。表 9-1 中列出了可以在 WebFilter 注解类型中出现的属性。所有属性都是可选的。

表 9-1 WebFilter 属性

属性	描述
asyncSupported	指定过滤器是否支持异步操作模式
description	过滤器的描述
dispatcherTypes	应用过滤器的 dispatcher 类型
displayName	过滤器的显示名称
filterName	过滤器的名称
initParams	初始参数
largeIcon	过滤器的大图标名称
servletNames	适用于过滤器的 Servlets 名称
smallIcon	过滤器的小图标名称
urlPatterns	应用过滤器的 URL 模式
value	应用过滤器的 URL 模式

例如，以下 @WebFilter 注解指定过滤器的名称为 DataCompressionFilter，它适用于所有资源。

```
@WebFilter(filterName="DataCompressionFilter", urlPatterns={"/"})
```

它相当于在部署描述符中声明这些 filter 和 filter-mapping 元素。

```
<filter>
    <filter-name>DataCompressionFilter</filter-name>
    <filter-class>
        the fully-qualified name of the filter class
    </filter-class>
</filter>
```

```
<filter-mapping>
    <filter-name>DataCompresionFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

再举个例子。下面的过滤器中设置了两个初始参数：

```
@WebFilter(filterName = "Security Filter", urlPatterns = { "/" },
    initParams = {
        @WebInitParam(name = "frequency", value = "1909"),
        @WebInitParam(name = "resolution", value = "1024")
    }
)
```

在部署描述符中使用 filter 和 filter-mapping 元素，其配置设置如下：

```
<filter>
    <filter-name>Security Filter</filter-name>
    <filter-class>filterClass</filter-class>
    <init-param>
        <param-name>frequency</param-name>
        <param-value>1909</param-value>
    </init-param>
    <init-param>
        <param-name>resolution</param-name>
        <param-value>1024</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>DataCompresionFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

关于部署描述符的详情请查看第 16 章的内容。

### 9.3 范例 1：日志过滤器

第一个例子，先看一下 app09a 应用程序，其中有一个简单的过滤器，用于在一个文本文件中记录请求 URI。文本文件的名称可以通过一个初始参数进行配置。此外，日志中的每个入口前面都可以添加一个也是初始参数的预设字符串。从日志中可以推断出一些有价值的信息，例如应用程序中的哪一项资源最受欢迎，或者网站每天哪个时间段的访问量最大等。

这里的过滤器类为 LoggingFilter，如代码清单 9-1 所示。按照规范，过滤器类的名称必须以 Filter 结束。

## 代码清单 9-1 LoggingFilter 类

```
package filter;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Date;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.http.HttpServletRequest;

@WebFilter(filterName = "LoggingFilter", urlPatterns = { "/" },
    initParams = {
        @WebInitParam(name = "logFileName",
            value = "log.txt"),
        @WebInitParam(name = "prefix", value = "URI: ") })
public class LoggingFilter implements Filter {

    private PrintWriter logger;
    private String prefix;

    @Override
    public void init(FilterConfig filterConfig)
        throws ServletException {
        prefix = filterConfig.getInitParameter("prefix");
        String logFileName = filterConfig
            .getInitParameter("logFileName");
        String appPath = filterConfig.getServletContext()
            .getRealPath("/");
        // without path info in logFileName, the log file will be
        // created in $TOMCAT_HOME/bin

        System.out.println("logFileName:" + logFileName);
        try {
            logger = new PrintWriter(new File(appPath,
                logFileName));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            throw new ServletException(e.getMessage());
        }
    }

    @Override
    public void destroy() {
```

```

        System.out.println("destroying filter");
        if (logger != null) {
            logger.close();
        }
    }

    @Override
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain filterChain)
        throws IOException, ServletException {
        System.out.println("LoggingFilter.doFilter");
        HttpServletRequest httpServletRequest =
            (HttpServletRequest) request;
        logger.println(new Date() + " " + prefix
            + httpServletRequest.getRequestURI());
        logger.flush();
        filterChain.doFilter(request, response);
    }
}

```

---

让我们仔细分析一下这个过滤器类吧。

首先，这个过滤器类实现了 **Filter** 接口，并声明了两个变量：称作 **logger** 的 **PrintWriter**，和称作 **prefix** 的 **String**：

```

private PrintWriter logger;
private String prefix;

```

**PrintWriter** 将用来编写文本文件。**String prefix** 则作为每个日志条目的前缀。

过滤器类要用 **@WebFilter** 进行标注，并给过滤器传递两个参数 (**logFileName** 和 **prefix**)：

```

@WebFilter(filterName = "LoggingFilter", urlPatterns = { "/" })
    initParams = {
        @WebInitParam(name = "logFileName",
                      value = "log.txt"),
        @WebInitParam(name = "prefix", value = "URI: ")
    }
)

```

**init** 方法在传入的 **FilterConfig** 中调用 **getInitParameter** 方法，获取 **prefix** 和 **logFileName** 初始值。**prefix** 参数值赋给类级变量 **prefix**，并用 **logFileName** 创建一个 **PrintWriter**。

```

prefix = filterConfig.getInitParameter("prefix");
String logFileName = filterConfig
    .getInitParameter("logFileName");

```

由于 **Servlet/JSP** 应用程序是由 **Servlet/JSP** 容器启动的，因此当前工作路径就是调用 **java** 执行文件时的位置。在 **Tomcat** 中，这个路径应为 **Tomcat** 安装目录下的 **bin** 目

录。为了在应用程序目录中创建一个日志文件，需要有一个它的绝对路径。我们可以利用 `ServletContext.getRealPath` 方法获取，也就是将应用程序路径和 `logFileName` 初始参数合并，得到：

```
String appPath = filterConfig.getServletContext()
    .getRealPath("/");
// without path info in logFileName, the log file will be
// created in $TOMCAT_HOME/bin

try {
    logger = new PrintWriter(new File(appPath,
        logFileName));
} catch (FileNotFoundException e) {
    e.printStackTrace();
    throw new ServletException(e.getMessage());
}
```

执行 `init` 方法时就会创建一个日志文件。如果应用程序目录下已经有同名文件存在，该文件将会被新文件覆盖。

当应用程序关闭时，必须关闭 `PrintWriter`。因此，在过滤器的 `destroy` 方法中要这样编写：

```
if (logger != null) {
    logger.close();
}
```

`doFilter` 方法通过将 `ServletRequest` 向下转换成 `HttpServletRequest`，并调用其 `getRequestURI` 方法，记录下所有请求的日志。然后，将 `getRequestURI` 的结果填入 `PrintWriter` 的 `println` 方法中：

```
HttpServletRequest httpServletRequest =
    (HttpServletRequest) request;
logger.println(new Date() + " " + prefix
    + httpServletRequest.getRequestURI());
```

每一个日志条目都会有一个时间戳和一个前缀，以便识别。之后，`doFilter` 方法接着刷新 `PrintWriter`，并调用 `FilterChain.doFilter` 来调用资源。

```
logger.flush();
filterChain.doFilter(request, response);
```

当你运行 Tomcat 时，不用等待第一个请求进来，过滤器就会进入服务状态。在控制台上应该能够看到 `logFileName` 参数的值。

如果想要测试一下这个过滤器，则可以利用下面这个 URL 在 `app09a` 应用程序中调用 `test.jsp` 页面：

<http://localhost:8080/app09a/test.jsp>

通过查看日志文件的内容，可以验证过滤器是否正常工作。

## 9.4 范例 2：图片保护过滤器

本例中的 Image Protector Filter 防止通过在浏览器的地址栏中直接输入图片 URL 来下载图片。只有在页面中单击图片的链接时，才会显示应用程序中的图片。过滤器通过查看 HTTP 标头 `referer` 的值进行工作。值为空表示当前请求没有相当的引用页，换句话说，该资源是直接输入其 URL 进行请求的。标头 `referer` 值非空的资源，将以原始页面作为引用页。注意标头名称 `referer` 的拼写，它在第二个 e 和第三个 e 之间只有一个 r。

过滤器类 `ImageProtectorFilter` 如代码清单 9-2 所示。从 `WebFilter` 注解中可以看出，该过滤器适用于扩展名为 png、jpg 或 gif 的所有资源。

代码清单 9-2 `ImageProtectorFilter` 类

---

```
package filter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

@WebFilter(filterName = "ImageProtectorFilter", urlPatterns = {
    "*.png", "*.jpg", "*.gif" })
public class ImageProtectorFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig)
        throws ServletException {
    }

    @Override
    public void destroy() {
    }

    @Override
    public void doFilter(ServletRequest request,
        ServletResponse response, FilterChain filterChain)
        throws IOException, ServletException {
        System.out.println("ImageProtectorFilter");
        HttpServletRequest httpServletRequest =

```

```

        (HttpServletRequest) request;
        String referrer = httpServletRequest.getHeader("referer");
        System.out.println("referrer:" + referrer);
        if (referrer != null) {
            filterChain.doFilter(request, response);
        } else {
            throw new ServletException("Image not available");
        }
    }
}

```

---

**init** 和 **destroy** 方法为空。**doFilter** 方法读取 **referer** 标头的值，并调用资源或者抛出异常：

```

String referrer = httpServletRequest.getHeader("referer");
System.out.println("referrer:" + referrer);
if (referrer != null) {
    filterChain.doFilter(request, response);
} else {
    throw new ServletException("Image not available");
}

```

测试过滤器时，试着在浏览器地址栏中直接输入下面这个 URL 来打开 **logo.png** 图片：

<http://localhost:8080/app09a/image/logo.png>

你将会收到一条“Image not available”的错误消息。

现在，调用 **image.jsp** 页面：

<http://localhost:8080/app09a/image.jsp>

此时应该可以看到图片了。为什么可以这样做？这是因为 **image.jsp** 页面中包含了这个链接，可以引导浏览器去下载图片：

```
<img src='image/logo.png' />
```

当浏览器通过链接请求图片时，它也以 **referer** 标头值的形式，将页面的 URL 发送给了服务器（在本例中是指 <http://localhost:8080/app09a/image.jsp>）。

## 9.5 范例 3：下载计数过滤器

本例中的下载计数过滤器可以计算某一个资源被下载了多少次。当你想要知道你的文档或者视频的受欢迎程度时，这个就很有帮助。为了简便起见，这些数字会被保存在一个属性文件中，而不是保存在一个数据库中。资源 URI 作为属性文件的属性键。

由于我们是将值保存在属性文件中，并且多个线程可以同时访问一个过滤器，因此就有

一个线程安全性的问题需要解决。用户请求资源，过滤器就需要读取相应的属性值，将它递增一个，并存回新值。如果在第一个线程完成任务之前，又有第二个用户请求同一个资源，该怎么办？那么，这时候的计数将是不准确的。同步读写值的代码似乎不是个好办法，因为这会涉及可扩展性的问题。

这个例子展示了如何利用 `Queue` 和 `Executor` 解决这个线程安全性问题。如果你对这两个 Java 类型不熟悉，请参阅第 18 章或者笔者的另一本书《Java 7 程序设计》(Java: A Beginner's Tutorial) 的相关内容。

简而言之，所有进来的请求都在一个线程 `Executor` 的队列中放置一项任务。放置任务比较快，因为这是一个异步的操作，你不需要等待任务完成。`Executor` 每次从队列中取用一个项目，并递增正确的属性。由于 `Executor` 只使用一个线程，因此消除了多线程访问该属性文件的可能。

`DownloadCounterFilter` 类如代码清单 9-3 所示。

代码清单 9-3 `DownloadCounterFilter`

---

```

package filter;
import java.io.File;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Properties;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

@WebFilter(filterName = "DownloadCounterFilter",
           urlPatterns = { "/" })
public class DownloadCounterFilter implements Filter {

    ExecutorService executorService = Executors
        .newSingleThreadExecutor();
    Properties downloadLog;
    File logFile;

    @Override
    public void init(FilterConfig filterConfig)
        throws ServletException {
        System.out.println("DownloadCounterFilter");
}

```

```

String appPath = filterConfig.getServletContext()
    .getRealPath("/");
logFile = new File(appPath, "downloadLog.txt");
if (!logFile.exists()) {
    try {
        logFile.createNewFile();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
downloadLog = new Properties();
try {
    downloadLog.load(new FileReader(logFile));
} catch (IOException e) {
    e.printStackTrace();
}
}

@Override
public void destroy() {
    executorService.shutdown();
}

@Override
public void doFilter(ServletRequest request,
    ServletResponse response, FilterChain filterChain)
    throws IOException, ServletException {
    HttpServletRequest httpServletRequest = (HttpServletRequest)
request;

    final String uri = httpServletRequest.getRequestURI();
    executorService.execute(new Runnable() {
        @Override
        public void run() {
            String property = downloadLog.getProperty(uri);
            if (property == null) {
                downloadLog.setProperty(uri, "1");
            } else {
                int count = 0;
                try {
                    count = Integer.parseInt(property);
                } catch (NumberFormatException e) {
                    // silent
                }
                count++;
                downloadLog.setProperty(uri,
                    Integer.toString(count));
            }
            try {
                downloadLog
                    .store(new FileWriter(logFile), "");
            } catch (IOException e) {

```

```

        }
    }
}
filterChain.doFilter(request, response);
}
}

```

---

如果应用程序目录下还没有 downloadLog.txt 文件，那么过滤器的 init 方法就会创建一个。

```

String appPath = filterConfig.getServletContext()
    .getRealPath("/");
logFile = new File(appPath, "downloadLog.txt");
if (!logFile.exists()) {
    try {
        logFile.createNewFile();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

之后，创建一个 Properties 对象，并加载文件：

```

downloadLog = new Properties();
try {
    downloadLog.load(new FileReader(logFile));
} catch (IOException e) {
    e.printStackTrace();
}

```

注意，过滤器类用一个 ExecutorService (Executor 的一个子类) 作为类级对象引用。

```

ExecutorService executorService = Executors
    .newSingleThreadExecutor();

```

当过滤器即将被销毁时，过滤器就会关闭 ExecutorService。

```

public void destroy() {
    executorService.shutdown();
}

```

doFilter 方法做了大量的工作。它取得每个请求的 URI，调用 ExecutorService 的 execute 方法，并调用 FilterChain.doFilter()。传给 execute 方法的任务很容易理解。它一般用 URI 作为一个属性键，并从 Properties 对象中获取属性值，将它递增一个数，并将值刷新到底层的日志文件中去。

```

@Override
public void run() {
    String property = downloadLog.getProperty(uri);
}

```

```

if (property == null) {
    downloadLog.setProperty(uri, "1");
} else {
    int count = 0;
    try {
        count = Integer.parseInt(property);
    } catch (NumberFormatException e) {
        // silent
    }
    count++;
    downloadLog.setProperty(uri,
                           Integer.toString(count));
}
try {
    downloadLog
        .store(new FileWriter(logFile), "");
} catch (IOException e) {
}
}

```

过滤器适用于任何资源，但是如果你愿意，也可以轻松地将它限制为只适用于 PDF 或 AVI 文件。

## 9.6 过滤器的顺序

如果多个过滤器应用于同一个资源，那么调用顺序就很重要，必须用部署描述符管理应该先调用哪一个过滤器。假如 Filter1 必须在 Filter2 之前调用，那么在部署描述符中，Filter1 的声明就要放在 Filter2 的声明之前。

```

<filter>
    <filter-name>Filter1</filter-name>
    <filter-class>
        the fully-qualified name of the filter class
    </filter-class>
</filter>
<filter>
    <filter-name>Filter2</filter-name>
    <filter-class>
        the fully-qualified name of the filter class
    </filter-class>
</filter>

```

如果没有部署描述符，是不可能管理过滤器调用顺序的。关于部署描述符的详情请查看第 16 章的相关内容。

## 9.7 小结

本章学习了 Filter API，包括 Filter 接口、FilterConfig 接口和 FilterChain 接口。还学习了如何通过实现 Filter 接口来编写过滤器，以及如何利用 @WebFilter 注解类，并在部署描述符中为它进行注册。

每一种过滤器都只有一个实例。因此，如果需要保持和改变过滤器类中的状态，那么线程安全可能会是一个问题。最后一个过滤器范例中讲解了如何处理这个问题。

# 第 10 章 应用程序设计

Java Web 应用程序设计中使用了两个模型，简称 Model 1 和 Model 2。Model 1 以页面为中心，只适用于非常小型的应用程序。除了最简单的 Java Web 应用程序之外，我们建议对所有应用程序都使用 Model 2。

本章将详细介绍 Model 2，并提供了 5 个 Model 2 的范例应用程序。第一个范例介绍了一个基础的 Model 2 应用程序，并用一个 Servlet 作为控制器。第二个范例也是一个简单的 Model 2 应用程序，但它用一个过滤器作为控制器。第三个范例介绍了一个验证器组件，用来验证用户的输入。第四个范例介绍如何利用 DAO（Data Access Object）模式，在 Model 2 应用程序中促进数据库访问和数据操作。最后一个范例介绍了一个支持依赖注入（Dependency Injection）的 Model 2 应用程序。

## 10.1 Model 1 概述

初学 JSP 时，范例应用程序一般是通过单击另一个页面的可单击链接，将浏览器从一个页面导航到另一个页面的。虽然这种导航方法很简单，但是对于中等或是带有大量页面的大型应用程序，这种方法就会带来维护的问题。例如，要修改某一个 JSP 页面的名字，就不得不修改该页面在其他许多页面中的链接名称。因此，不建议使用 Model 1，除非你的应用程序中只有两三个页面。

## 10.2 Model 2 概述

Model 2 基于 MVC（Model-View-Controller）设计模式，这也是 Smalltalk-80 用户界面背后的主要概念。因为当时还没有“设计模式”的概念，因此称作“MVC 范例”。

实现 MVC 模式应用程序由三个模块组成：模型 Model、视图 View 和控制器 Controller。View 负责应用程序的显示，Model 负责封装应用程序的数据和业务逻辑，Controller 则负责接收用户的输入，并命令 Model 和 / 或 View 做出相应的修改。

---

**提示** 由 Steve Burbeck 博士所著的论文“Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller(MVC)”中全面阐述了 MVC 模式。在以下网站可以查找到这篇文章：<http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>。

---

在 Model 2 中，是用一个 Servlet 或者过滤器充当控制器 Controller。所有现代的 Web 框架都是 Model 2 实现。像 Struts 1 和 Spring MVC 这类框架是在它们的 MVC 架构中使用一个 Servlet Controller，而另一个流行的框架：Struts 2，则是使用过滤器。尽管它也支持其他的 View 技术，但一般来说，它用 JSP 页面作为应用程序的 View。至于 Model，则是使用 POJO（POJO 是 Plain Old Java Object 的缩写，意思是简单 Java 对象）。POJO 是普通的对象，与 EJB（Enterprise JavaBeans）或者其他特殊对象不同。许多人选择用 JavaBean（普通的 JavaBean，而不是 EJB）保存模型对象的状态，并将业务逻辑转移到一个 Action 类中。JavaBean 必须有一个无参的构造器，以及用于访问属性的 get/set 方法。此外，它还必须是可以被序列化的。

图 10-1 展示了 Model 2 应用程序的架构。

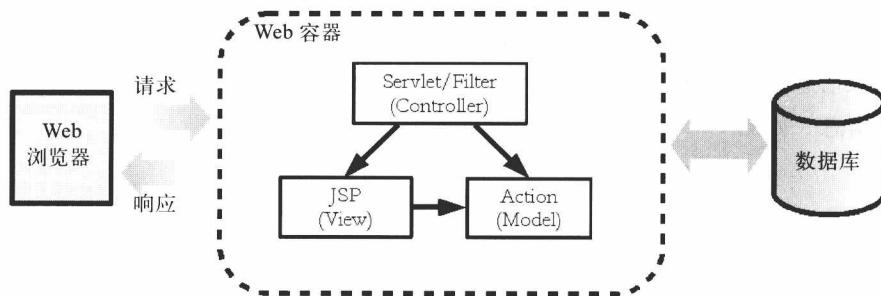


图 10-1 Model 2 架构

在 Model 2 应用程序中，每一个 HTTP 请求都必须被定向到控制器中。请求的 URI (Uniform Request Identifier，统一请求标识符) 告诉控制器要调用哪一个 Action。术语“Action”是指应用程序能够执行的一项操作。与一个 Action 相关的 Java 对象称作 Action 对象。一个 Action 类可以用来服务几个不同的 Action (像在 Struts 2 和 Spring MVC 中那样) 或单个 Action (像在 Struts 1 中那样)。

一项看似很小的操作可能会执行多个 Action。例如，往数据库中添加一项产品需要两个 Action：

1. 显示一个 Add Product 表单，供用户输入产品信息。
2. 在数据库中保存产品信息。

如上所述，我们是用 URI 告诉控制器要调用哪一项 Action。例如，想让应用程序发送一个 Add Product 表单，要使用一个像下面这样的 URI：

`http://domain/appName/product_input`

想让应用程序保存一项产品时，URI 应该为：

`http://domain/appName/product_save`

控制器会仔细查看 URI，决定要调用哪一项 Action。它还会将模型对象保存在一个可以通过 View 访问到的地方，以便服务器端的值可以在浏览器中显示出来。最后，控制器用一个 RequestDispatcher 跳转到一个 View (JSP 页面)。在 JSP 页面中，我们利用 Expression Language 表达式和定制标签来显示值。

注意，调用 RequestDispatcher.forward() 并没有阻止执行它后面的代码。因此，除非是在方法的最后一行调用，否则都需要显式地返回。

### 10.3 基于 Servlet Controller 的 Model 2

本节介绍一个简单的 Model 2 应用程序，让你对什么是 Model 2 应用程序有一个整体的了解。在现实生活中，Model 2 应用程序远比这个要复杂得多。

这个应用程序可以用来输入产品信息，我们把它命名为 app10a。用户在如图 10-2 所示的表单中输入信息，并提交。之后，应用程序就会给用户发送一个确认页面，并显示所保存产品的详细信息（如图 10-3 所示）。

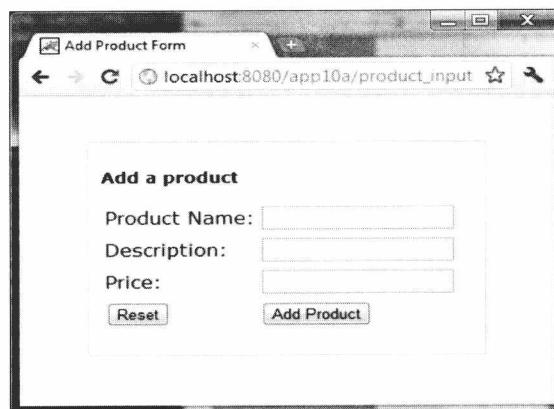


图 10-2 产品表单

这个应用程序能够执行下面这两个 Action：

1. 显示 Add Product 表单。这个 Action 将图 10-2 中的登录表单发送到浏览器。调用这个 Action 的 URI 必须包含字符串 product\_input。
2. 保存产品，并返回如图 10-3 所示的确认页面。调用这个 Action 的 URI 必须包含字符串 product\_save。

这个应用程序包含以下组件：

1. Product 类是模型对象的模板。这个类的实例中包含的是产品信息。
2. ProductForm 类，它封装用于输入产品的 HTML 表单域。ProductForm 的属性用来填充 Product。

3. ControllerServlet 类，它是这个 Model 2 应用程序的 Controller。
  4. Action 类：SaveProductAction。
  5. 作为 View 的两个 JSP 页面（ProductForm.jsp 和 ProductDetails.jsp）。
  6. 定义视图样式的一个 CSS 文件，这是一个静态资源。
- 这个应用程序的目录结构如图 10-4 所示。

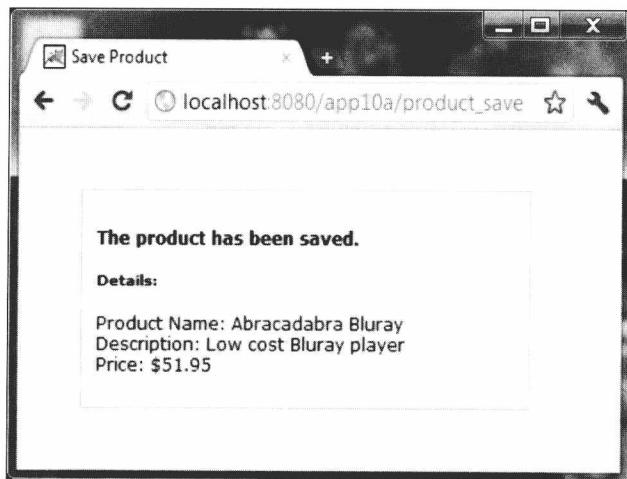


图 10-3 产品描述页面

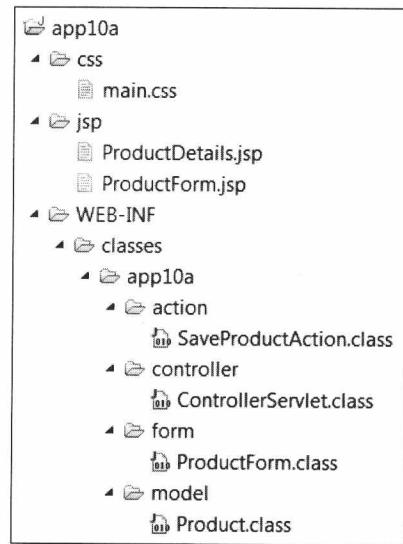


图 10-4 app10a 的目录结构

我们来看一下 app10a 中的各个组件。

### 10.3.1 Product 类

Product 实例是一个封装产品信息的 JavaBean。Product 类（如代码清单 10-1 所示）中有 3 个属性：productName、description 和 price。

代码清单 10-1 Product 类

---

```

package app10a.model;
import java.io.Serializable;

public class Product implements Serializable {
    private static final long serialVersionUID = 748392348L;
    private String name;
    private String description;
    ...
}

```

```

private float price;

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getDescription() {
    return description;
}

public void setDescription(String description) {
    this.description = description;
}

public float getPrice() {
    return price;
}

public void setPrice(float price) {
    this.price = price;
}
}

```

---

Product 类实现了 `java.io.Serializable` 接口，以便其实例可以安全地保存在 `HttpSession` 对象中。作为 `Serializable` 接口的一个实现，Product 类应该要有一个 `serialVersionUID` 域。

### 10.3.2 ProductForm 类

Form 类被映射到一个 HTML 表单。它是 HTML 表单在服务器中的表示法。如代码清单 10-2 所示，`ProductForm` 类中包含了产品的 `String` 值。乍看之下，`ProductForm` 类与 `Product` 类相似，那么你可能会问，为什么还需要 `ProductForm` 类呢？如本章稍后将要讲述的，Form 对象不需要将 `ServletRequest` 传给其他组件，比如验证器。`ServletRequest` 是一个针对 Servlet 的类型，不应该暴露给应用程序的其他层。

Form 对象的第二个用途是保存用户的输入，并于验证失败时，在它的原始表单中重新显示出来。在本章稍后会学到具体的做法。

注意，Form 类通常并不需要实现 `Serializable` 接口，因为 Form 对象通常不会保存在 `HttpSession` 中。

代码清单 10-2 ProductForm 类

```

package app10a.form;
public class ProductForm {
    private String name;
    private String description;
    private String price;

    public String getName() {

```

```

        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public String getPrice() {
        return price;
    }
    public void setPrice(String price) {
        this.price = price;
    }
}

```

### 10.3.3 ControllerServlet 类

ControllerServlet 类（如代码清单 10-3 所示）继承了 javax.servlet.http.HttpServlet 类。它的 doGet 和 doPost 方法都调用 process 方法，这是 Servlet 控制器的核心。

我把这个 Servlet Controller 命名为 ControllerServlet 时也忍不住皱了几次眉头，但是按照规范，所有的 Servlet 类都必须以 Servlet 作为后缀啊。

代码清单 10-3 ControllerServlet 类

```

package app10a.controller;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import app10a.action.SaveProductAction;
import app10a.form.ProductForm;
import app10a.model.Product;

@WebServlet(name = "ControllerServlet", urlPatterns = {
    "/product_input", "/product_save" })
public class ControllerServlet extends HttpServlet {
    private static final long serialVersionUID = 1579L;

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

```

```
}

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

private void process(HttpServletRequest request,
    HttpServletResponse response)
    throws IOException, ServletException {

    String uri = request.getRequestURI();
    /*
     * uri is in this form: /contextName/resourceName,
     * for example: /app10a/product_input.
     * However, in the event of a default context, the
     * context name is empty, and uri has this form
     * /resourceName, e.g.: /product_input
     */
    int lastIndex = uri.lastIndexOf("/");
    String action = uri.substring(lastIndex + 1);
    // execute an action
    if (action.equals("product_input")) {
        // no action class, there is nothing to be done
    } else if (action.equals("product_save")) {
        // create form
        ProductForm productForm = new ProductForm();
        // populate action properties
        productForm.setName(request.getParameter("name"));
        productForm.setDescription(
            request.getParameter("description"));
        productForm.setPrice(request.getParameter("price"));

        // create model
        Product product = new Product();
        product.setName(productForm.getName());
        product.setDescription(productForm.getDescription());
        try {
            product.setPrice(Float.parseFloat(
                productForm.getPrice()));
        } catch (NumberFormatException e) {
        }
        // execute action method
        SaveProductAction saveProductAction =
            new SaveProductAction();
        saveProductAction.save(product);

        // store model in a scope variable for the view
        request.setAttribute("product", product);
    }
}
```

```

        }

        // forward to a view
        String dispatchUrl = null;
        if (action.equals("product_input")) {
            dispatchUrl = "/jsp/ProductForm.jsp";
        } else if (action.equals("product_save")) {
            dispatchUrl = "/jsp/ProductDetails.jsp";
        }
        if (dispatchUrl != null) {
            RequestDispatcher rd =
                request.getRequestDispatcher(dispatchUrl);
            rd.forward(request, response);
        }
    }
}

```

---

**ControllerServlet** 类中的 **process** 方法负责处理所有进来的请求。首先，它要获得请求 URI 和 Action 名称。

```

String uri = request.getRequestURI();
int lastIndex = uri.lastIndexOf("/");
String action = uri.substring(lastIndex + 1);

```

这个应用程序中的 **action** 值可以是 **product\_input**，也可以是 **product\_save**。

之后，**process** 方法继续执行以下步骤：

- 如果有相关的 Action 类，则将它实例化。

- 如果有 Action 对象，则创建一个 Form 对象，并为它填入请求参数。**product\_save** Action 中有 3 个属性：**name**、**description** 和 **price**。接下来，创建一个方法对象，并通过 Form 对象填入它的属性值。

- 如果有 Action 对象，则调用 Action 方法。

- 将请求转到一个 View (JSP 页面) 中。

决定要执行哪个 Action 的 **process** 方法部分放在下面这个 if 块中：

```

// execute an action
if (action.equals("product_input")) {
    // there is nothing to be done
} else if (action.equals("product_save")) {
    // instantiate action class
    ...
}

```

**product\_input** Action 没有什么 Action 类需要实例化。对于 **product\_save**，**process** 方法会创建一个 **ProductForm** 和一个 **Product**，并将值从前者复制到后者中去。此时，并不能确保所有的非字符串属性（例如 **price**）都可以复制成功，但是我们在稍后会解决这个问题。

接着，`process` 方法实例化 `SaveProductAction` 类，并调用它的 `save` 方法。

```
// create form
ProductForm productForm = new ProductForm();
// populate action properties
productForm.setName(request.getParameter("name"));
productForm.setDescription(
    request.getParameter("description"));
productForm.setPrice(request.getParameter("price"));

// create model
Product product = new Product();
product.setName(productForm.getName());
product.setDescription(product.getDescription());
try {
    product.setPrice(Float.parseFloat(
        productForm.getPrice()));
} catch (NumberFormatException e) {
}
// execute action method
SaveProductAction saveProductAction =
    new SaveProductAction();
saveProductAction.save(product);

// store model in a scope variable for the view
request.setAttribute("product", product);
```

接着，将 `Product` 保存在 `HttpServletRequest` 中，以便 View 能够访问到它。

```
// store action in a scope variable for the view
request.setAttribute("product", product);
```

`process` 方法通过运算结果推断，转向不同的页面。假如 `action` 等于 `product_input`，控制权就转到 `ProductForm.jsp` 页面。如果 `action` 等于 `product_save`，控制权则转到 `ProductDetails.jsp` 页面。

```
// forward to a view
String dispatchUrl = null;
if (action.equals("Product_input")) {
    dispatchUrl = "/jsp/ProductForm.jsp";
} else if (action.equals("Product_save")) {
    dispatchUrl = "/jsp/ProductDetails.jsp";
}
if (dispatchUrl != null) {
    RequestDispatcher rd =
        request.getRequestDispatcher(dispatchUrl);
    rd.forward(request, response);
}
```

## Action 类

这个应用程序中只有一个 Action 类，它负责将产品保存到某个存储设备中，比如数据库。这个 Action 类命名为 `SaveProductAction`，如代码清单 10-4 所示。

代码清单 10-4 `SaveProductAction` 类

---

```
package app10a.action;

public class SaveProductAction {
    public void save(Product product) {
        // insert Product to the database
    }
}
```

---

在本例中，`SaveProductAction` 类没有其 `save` 方法的实现，我们会在本章的下一个范例中提供一个方法实现。

## View

这个应用程序用了两个 JSP 页面作为 View。第一个页面是 `ProductForm.jsp`，当 `action` 为 `product_input` 时显示。第二个页面是 `ProductDetails.jsp`，当 `action` 为 `product_save` 时显示。`ProductForm.jsp` 如代码清单 10-5 所示，`ProductDetails.jsp` 如代码清单 10-6 所示。

代码清单 10-5 `ProductForm.jsp` 页面

---

```
<!DOCTYPE HTML>
<html>
<head>
<title>Add Product Form</title>
<style type="text/css">@import url(css/main.css);</style>
</head>
<body>
<div id="global">
    <h3>Add a product</h3>
    <form method="post" action="product_save">
        <table>
            <tr>
                <td>Product Name:</td>
                <td><input type="text" name="name"/></td>
            </tr>
            <tr>
                <td>Description:</td>
                <td><input type="text" name="description"/></td>
            </tr>
            <tr>
                <td>Price:</td>
                <td><input type="text" name="price"/></td>
            </tr>
        </table>
    </form>
</div>
</body>
</html>
```

---

```

</tr>
<tr>
    <td><input type="reset"/></td>
    <td><input type="submit" value="Add Product"/></td>
</tr>
</table>
</form>
</div>
</body>
</html>

```

代码清单 10-6 ProductDetails.jsp 页面

```

<!DOCTYPE HTML>
<html>
<head>
<title>Save Product</title>
<style type="text/css">@import url(css/main.css);</style>
</head>
<body>
<div id="global">
    <h4>The product has been saved.</h4>
    <p>
        <h5>Details:</h5>
        Product Name: ${product.name}<br/>
        Description: ${product.description}<br/>
        Price: $$ ${product.price}
    </p>
</div>
</body>
</html>

```

ProductForm.jsp 页面中包含了一个用来输入产品信息的 HTML 表单。ProductDetails.jsp 页面利用 Expression Language (EL) 来访问 HttpServletRequest 中 product 限域对象。

在这个应用程序中，像大多数 Model 2 应用程序一样，必须防止直接从浏览器访问 JSP 页面。它的实现方法有很多种，包括：

- 将 JSP 页面放在 WEB-INF 目录下。在 WEB-INF 下或 WEB-INF 子目录下的所有内容都是受到保护的。如果将 JSP 页面放在 WEB-INF 目录下，那么将无法直接通过浏览器访问它们，但是 Controller 仍然可以将请求分发给那些页面。不过我们不建议这么做，因为并非所有的容器都实现这项特性。
- 使用一个 Servlet 过滤器，过滤 JSP 页面的请求。
- 在部署描述符中使用安全限制。这比使用过滤器要容易许多，因为不需要编写过滤器类。本例正是选择采用这种方法。

## 测试应用程序

假设你是在端口 8080 本机上运行这个应用程序，就可以利用下面的 URL 调用应用程序：

```
http://localhost:8080/app10a/product_input
```

在你的浏览器上就会看到如图 10-2 所示的内容。

在提交表单时，以下 URL 会被发送到服务器：

```
http://localhost:8080/app10a/product_save
```

使用 Servlet Controller 时，允许用 Servlet 作为欢迎页面。这是一项重要的特性，因为可以通过配置应用程序，以便只需要在浏览器的地址栏中输入域名（如 <http://example.com>）就可以调用这个 Servlet Controller。对过滤器则不能这么处理。

## 10.4 基于 Filter Dispatcher 的 Model 2

虽然 Servlet 是 Model 2 应用程序中最常用的 Controller，但也可以用过滤器作为 Controller。不过注意，过滤器无权充当欢迎页面。只输入域名将不会调用过滤器分发器（Filter Dispatcher）。Struts 2 用过滤器作为 Controller，因为过滤器还用于提供静态的内容。

下面的范例（app10b）是一个使用过滤器分发器的 Model 2 应用程序。app10b 的目录结构如图 10-5 所示。

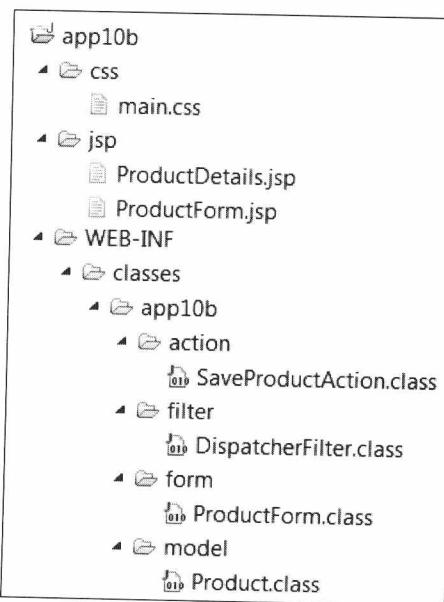


图 10-5 app10b 目录结构

JSP 页面和 Product 类与 app10a 中的一样。但它不是用 Servlet 作为 Controller，而是用了一个名为 FilterDispatcher 的过滤器（如代码清单 10-7 所示）。

代码清单 10-7 DispatcherFilter 类

---

```

package app10b.filter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

import app10b.action.SaveProductAction;
import app10b.form.ProductForm;
import app10b.model.Product;

@WebFilter(filterName = "DispatcherFilter",
           urlPatterns = { "/" })
public class DispatcherFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig)
        throws ServletException {
    }

    @Override
    public void destroy() {
    }

    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response, FilterChain filterChain)
        throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        String uri = req.getRequestURI();
        /*
         * uri is in this form: /contextName/resourceName, for
         * example /app01b/product_input. However, in the
         * case of a default context, the context name is empty,
         * and uri has this form /resourceName, e.g.:
         * /product_input
         */
        // action processing
        int lastIndex = uri.lastIndexOf("/");
        String action = uri.substring(lastIndex + 1);
        if (action.equals("product_input")) {
    }
}

```

```

        // do nothing
    } else if (action.equals("product_save")) {
        // create form
        ProductForm productForm = new ProductForm();
        // populate action properties
        productForm.setName(request.getParameter("name"));
        productForm.setDescription(
            request.getParameter("description"));
        productForm.setPrice(request.getParameter("price"));

        // create model
        Product product = new Product();
        product.setName(productForm.getName());
        product.setDescription(product.getDescription());
        try {
            product.setPrice(Float.parseFloat(
                productForm.getPrice()));
        } catch (NumberFormatException e) {
        }
        // execute action method
        SaveProductAction saveProductAction =
            new SaveProductAction();
        saveProductAction.save(product);

        // store model in a scope variable for the view
        request.setAttribute("product", product);
    }

    // forward to a view
    String dispatchUrl = null;
    if (action.equals("product_input")) {
        dispatchUrl = "/jsp/ProductForm.jsp";
    } else if (action.equals("product_save")) {
        dispatchUrl = "/jsp/ProductDetails.jsp";
    }
    if (dispatchUrl != null) {
        RequestDispatcher rd = request
            .getRequestDispatcher(dispatchUrl);
        rd.forward(request, response);
    } else {
        // let static contents pass
        filterChain.doFilter(request, response);
    }
}
}

```

**doFilter** 方法完成了 app10a 中 **process** 方法所做的工作。

由于过滤器的目标是所有包含静态内容的 URL，因此如果没有调用任何 Action，那么就需要调用 **filterChain.doFilter()** 方法。

```

    } else {
        // let static contents pass
        filterChain.doFilter(request, response);
    }
}

```

测试应用程序时，要在浏览器中打开以下 URL：

[http://localhost:8080/app10b/product\\_input](http://localhost:8080/app10b/product_input)

## 10.5 验证器

在执行 Action 时，输入验证是一个重要的步骤。验证范围从简单的任务到复杂的都有，例如简单的有检验某个输入域中是否有值，复杂的有验证信用卡号码，等等。事实上，由于验证起着如此重要的作用，Java 社区还专门发布了 JSR 303 “Bean Validation”，将 Java 中的输入验证标准化。现代的 MVC 框架经常同时提供编程式和声明式的验证方法。在编程式验证中，是通过编写代码来验证用户的输入。在声明式验证中，则是在 XML 文档或者属性文件中提供验证规则。

以下例子中提供了一个新的应用程序范例（app10c），它继承了 app10a 中基于 Servlet Controller 的 Model 2 应用程序。这个应用程序中整合了一个产品验证器，它的类如代码清单 10-8 所示。

代码清单 10-8 ProductValidator 类

---

```

package app10c.validator;
import java.util.ArrayList;
import java.util.List;
import app10c.form.ProductForm;

public class ProductValidator {

    public List<String> validate(ProductForm productForm) {
        List<String> errors = new ArrayList<String>();
        String name = productForm.getName();
        if (name == null || name.trim().isEmpty()) {
            errors.add("Product must have a name");
        }
        String price = productForm.getPrice();
        if (price == null || price.trim().isEmpty()) {
            errors.add("Product must have a price");
        } else {
            try {
                Float.parseFloat(price);
            } catch (NumberFormatException e) {
                errors.add("Invalid price value");
            }
        }
    }
}

```

```

        }
        return errors;
    }
}

```

---

代码清单 10-8 中的 `ProductValidator` 类提供了一个作用于 `ProductForm` 的 `validate` 方法。验证器可以确保产品的名称不为空，以及其价格是一个有效的数字。`validate` 方法返回一个包含验证错误消息的 `String` 的 `List`。`List` 为空意味着验证成功。

有了验证器之后，就要告诉 Controller 去使用它。代码清单 10-9 中展示了一个修改过的 `ControllerServlet`。请特别注意一下粗体部分的代码行。

代码清单 10-9 app10c 中的 `ControllerServlet` 类

```

package app10c.controller;
import java.io.IOException;
import java.util.List;
import javax.servlet.DispatcherType;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import app10c.action.SaveProductAction;
import app10c.form.ProductForm;
import app10c.model.Product;
import app10c.validator.ProductValidator;

@WebServlet(name = "ControllerServlet", urlPatterns = {
    "/product_input", "/product_save" })
public class ControllerServlet extends HttpServlet {

    private static final long serialVersionUID = 98279L;

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

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

    private void process(HttpServletRequest request,
                         HttpServletResponse response)

```

```

        throws IOException, ServletException {

    String uri = request.getRequestURI();
    /*
     * uri is in this form: /contextName/resourceName,
     * for example: /app10a/product_input.
     * However, in the case of a default context, the
     * context name is empty, and uri has this form
     * /resourceName, e.g.: /product_input
     */
    int lastIndex = uri.lastIndexOf("/");
    String action = uri.substring(lastIndex + 1);
    String dispatchUrl = null;

    if (action.equals("product_input")) {
        // no action class, there is nothing to be done
        dispatchUrl = "/jsp/ProductForm.jsp";
    } else if (action.equals("product_save")) {
        // instantiate action class
        ProductForm productForm = new ProductForm();
        // populate action properties
        productForm.setName(
            request.getParameter("name"));
        productForm.setDescription(
            request.getParameter("description"));
        productForm.setPrice(request.getParameter("price"));

        // validate ProductForm
        ProductValidator productValidator = new
            ProductValidator();
        List<String> errors =
            productValidator.validate(productForm);
        if (errors.isEmpty()) {
            // create Product from ProductForm
            Product product = new Product();
            product.setName(productForm.getName());
            product.setDescription(
                productForm.getDescription());
            product.setPrice(Float.parseFloat(
                productForm.getPrice()));

            // no validation error, execute action method
            SaveProductAction saveProductAction = new
                SaveProductAction();
            saveProductAction.save(product);

            // store action in a scope variable for the view
            request.setAttribute("product", product);
            dispatchUrl = "/jsp/ProductDetails.jsp";
        } else {
            request.setAttribute("errors", errors);
    }
}

```

```

        request.setAttribute("form", productForm);
        dispatchUrl = "/jsp/ProductForm.jsp";
    }
}

// forward to a view
if (dispatchUrl != null) {
    RequestDispatcher rd =
        request.getRequestDispatcher(dispatchUrl);
    rd.forward(request, response);
}
}
}

```

---

代码清单 10-9 中新增的 ControllerServlet 类中插入了代码，用于实例化 ProductValidator 类，并在 product\_save 中调用它的 validate 方法。

```

// validate ProductForm
ProductValidator productValidator = new
    ProductValidator();
List<String> errors =
    productValidator.validate(productForm);

```

validate 方法用一个 ProductForm 封装在 HTML 表单中输入的产品信息。如果没有 ProductForm，就只好将 ServletRequest 传给验证器。

如果验证成功，validate 方法就会返回一个空的 List，此时就会创建一个 Product，并传给 SaveProductAction。验证成功之后，Controller 就会将 Product 保存在 ServletContext 中，并转向 ProductDetails.jsp 页面，显示产品的详细信息。如果验证失败，Controller 会将 errors List 和 ProductForm 保存在 ServletContext 中，并返回到 ProductForm.jsp 页面。

```

if (errors.isEmpty()) {
    // create Product from ProductForm
    Product product = new Product();
    product.setName(productForm.getName());
    product.setDescription(
        productForm.getDescription());
    product.setPrice(Float.parseFloat(
        productForm.getPrice()));

    // no validation error, execute action method
    SaveProductAction saveProductAction = new
        SaveProductAction();
    saveProductAction.save(product);

    // store action in a scope variable for the view
    request.setAttribute("product", product);
    dispatchUrl = "/jsp/ProductDetails.jsp";
} else {
}

```

```

        request.setAttribute("errors", errors);
        request.setAttribute("form", productForm);
        dispatchUrl = "/jsp/ProductForm.jsp";
    }
}

```

app10c 中的 ProductForm.jsp 页面已经被修改为能够显示错误消息，以及重新显示无效的值。代码清单 10-10 中展示了 app10c 中的 ProductForm.jsp。

代码清单 10-10 app10c 中的 ProductForm.jsp

---

```

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE HTML>
<html>
<head>
<title>Add Product Form</title>
<style type="text/css">@import url(css/main.css);</style>
</head>
<body>
<div id="global">
    <h3>Add a product</h3>
    <c:if test="${requestScope.errors != null}">
        <p id="errors">
            Error(s) !
            <ul>
                <c:forEach var="error" items="${requestScope.errors}">
                    <li>${error}</li>
                </c:forEach>
            </ul>
        </p>
    </c:if>
    <form method="post" action="product_save">
        <table>
            <tr>
                <td>Product Name:</td>
                <td><input type="text" name="name"
                           value="${form.name}"/></td>
            </tr>
            <tr>
                <td>Description:</td>
                <td><input type="text" name="description"
                           value="${form.description}"/></td>
            </tr>
            <tr>
                <td>Price:</td>
                <td><input type="text" name="price"
                           value="${form.price}"/></td>
            </tr>
            <tr>
                <td><input type="reset"/></td>
                <td><input type="submit" value="Add Product"/></td>
            </tr>
        </table>
    </form>
</div>

```

```

</table>
</form>
</div>
</body>
</html>

```

通过调用 `product_input` Action 可以对 `app10c` 进行测试：

`http://localhost:8080/app10c/product_input`

与前面的例子不同的是，当你提交 `Product` 表单时，如果它里面包含无效的值，那么显示无效值的同时也会显示一条错误消息。图 10-6 展示了两条验证错误的消息。

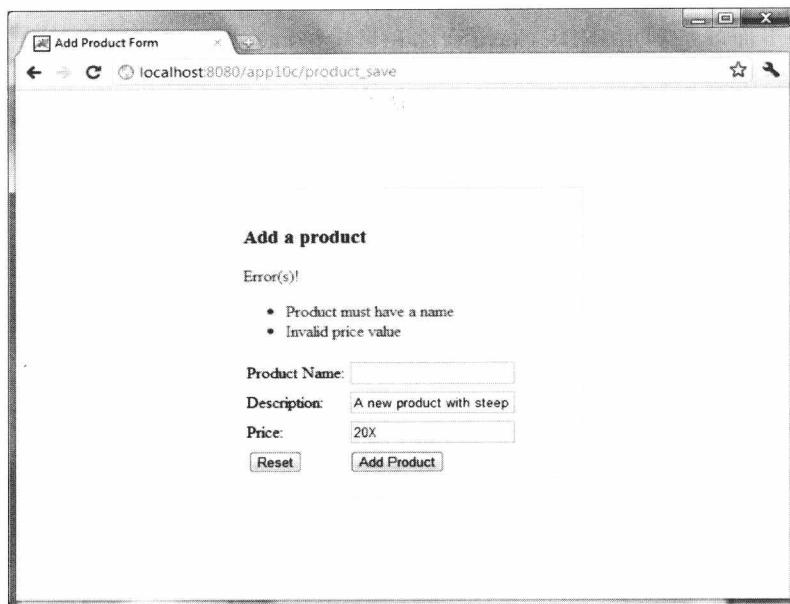


图 10-6 显示错误消息的 ProductForm

## 10.6 数据库访问

大多数的 Web 应用程序都是通过数据库来使用数据的。因此，知道如何利用 JDBC（Java 数据库连接）API 来访问和操作数据，这一点很重要。本节不会介绍 JDBC 的内容，因为这里假定你对这方面已经有了一定的工作经验。本节要介绍的是在使用数据库时会遇到的两大主题：连接池和 DAO（数据访问对象）模式。本节的最后提供了一个范例应用程序（`app10d`）。

### 10.6.1 连接池

在数据库中访问数据时，最重要且最费时的操作经常是建立连接。按规则，设计良好的应用程序数据库连接应该始终是采用连接池的。虽然管理连接可能比较复杂和困难，但值得庆幸的是，有 Java 类库可以很好地来完成这些工作。下面就是 Java 数据库连接类库的例子：

- Apache Commons DBCP (<http://commons.apache.org/dbcp>)
- C3P0 (<http://sourceforge.net/projects/c3p0>)
- Tomcat 7 中的 Tomcat JDBC Connection Pool

早期版本中的 Apache Commons DBCP 因其复杂和笨重而备受批评。因此，C3P0 似乎是更明智的选择。但是，如果你使用的是 Tomcat 7，那么它里面则已经内建了连接池。

从几年前开始，管理连接池的最流行方法已经不是亲自管理了，而是让 Servlet/JSP 容器来替你完成。下面就是利用 JNDI（Java 命名和目录接口）查找，从容器管理的连接池中获取 JDBC 连接的代码范例。

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;
import java.sql.Connection;
import java.sql.SQLException;

...
Connection connection = null;
try {
    Context context = new InitialContext();
    DataSource dataSource = (DataSource)
        context.lookup(jndiName);
    connection = dataSource.getConnection();
} catch (NamingException e) {
    ...
} catch (SQLException e) {
    ...
} catch (Exception e) {
    ...
}
```

调用 `DataSource` 中的 `getConnection` 方法比较快，因为连接永远不会被关闭；关闭连接时，只要将连接返回到池中即可。但是，JNDI 查找比较慢，因此，被返回的 `DataSource` 经常会被缓存起来。

为了让 Servlet 容器来管理连接池，需要对容器进行配置。在 Tomcat 中，是通过在应用程序的 `Context` 元素下声明 `Resource` 元素来实现的。例如，下面的 Tomcat 上下文中就包含了一个带有内部连接池的 `DataSource` 资源。

```
<Context path="/appName" docBase="...">
    <Resource name="jdbc/dataSourceName"
        auth="Container"
        type="javax.sql.DataSource"
        username="..."
        password="..."
        driverClassName="..."
        url="...">
    />
</Context>
```

需要分别在 `username` 和 `password` 属性中输入正确的数据库用户名和密码。还需要分别在 `driverClassName` 和 `url` 属性中提供 JDBC 驱动类名和数据库 URL。此外，在应用程序目录的 `WEB-INF/lib` 目录下还需要包括 JDBC 驱动类库。关于配置资源的更多信息请查看附录 A 的相关内容。

例如，下面的 Tomcat Context 声明中定义了一个应用程序 (`app10d`)，它带有一个 `DataSource` 资源，用来维护到 MySQL 数据库 `test` 的连接池。`DataSource` 的 JNDI 名称为 `jdbc/myDataSource`，并假设 MySQL 数据库就装在运行 Tomcat 的这台计算机上。访问数据库的用户名和密码分别为 `testuser` 和 `secret`。

```
<Context path="/app10d" docBase="/path/to/app" reloadable="true">
    <Resource name="jdbc/myDataSource"
        auth="Container"
        type="javax.sql.DataSource"
        username="testuser"
        password="secret"
        driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/test"/>
</Context>
```

虽然近来的应用程序大多是利用一个依赖注入框架来管理数据库连接，但是目前运行的许多应用程序仍然还是依赖早期的 JNDI 查找方法，这也正是我们之所以要在 `app10d` 应用程序中尝试它的原因。

## 10.6.2 DAO 模式

在数据库中访问数据的一种好办法是单独利用一个模块来管理获得连接和构建 SQL 语句的代码复杂性。DAO（Data Access Object，数据访问对象）设计模式是能够很好地完成这项工作的一种简单模式。这种模式有几种变体，最简单的一种如图 10-7 所示。

使用这个模式，要为需要持久化的每一种类型对象都编写一个类。例如，如果应用程序需要持久化三种类型的对象：`Product`、`Customer` 和 `Order`，那么就需要三个 DAO 类，每一个类负责一种对象类型。因此，你就有了这几个类：`ProductDAO`、`CustomerDAO` 和 `OrderDAO`。类名称中的 DAO 后缀表示这个类是一个 DAO 类。按照规范应该这么做，除非

你有可以不这么做的充分理由。

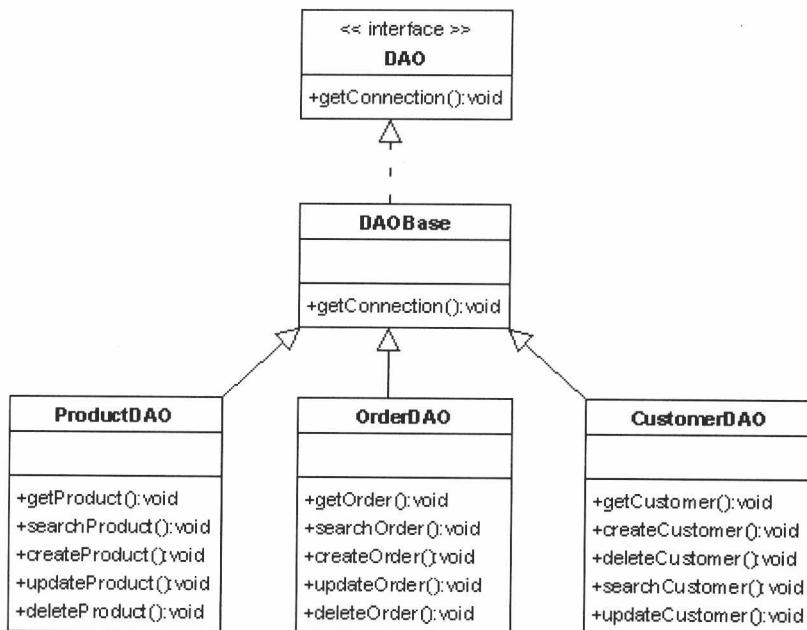


图 10-7 DAO 模式

典型的 DAO 类会负责对象的增加、删除、修改和获取，以及查找那些对象。例如，`ProductDAO` 类能够支持以下方法：

```

void addProduct(Product product)
void updateProduct(Product product)
void deleteProduct(int productId)
Product getProduct(int productId)
List<Product> findProducts(SearchCriteria searchCriteria)
  
```

在 DAO 实现类中，可以手工编写 SQL 语句，也可以利用像 Hibernate 这样的 JPA（Java Persistence API）实现来管理数据库数据。遗憾的是，JPA 不是本书讨论的范畴，但你应该知道它是一种流行的技术，许多人都会选择用 JPA 进行数据访问。

### 10.6.3 带有数据库访问的 Model 2 应用程序

`app10d` 应用程序是带有数据库访问的 Model 2 范例，它以前一个范例（`app10c`）为基础，要访问一个包含一个 `products` 表格的 MySQL 数据库 `test`。经过修改之后的应用程序中也多了一个 Action： `product_list`，它负责显示数据库中的所有数据。此外，这个应用程序中

还有一个 web.xml 文件，它声明了一个欢迎页面。

创建数据库的 SQL 语句和表格如代码清单 10-11 所示。

#### 代码清单 10-11 创建数据库和表格的 SQL 语句

---

```
CREATE DATABASE /*!32312 IF NOT EXISTS*/'test' /*!40100 DEFAULT
    CHARACTER SET latin1 */;
USE 'test';
DROP TABLE IF EXISTS 'products';

CREATE TABLE 'products' (
    'id' int(11) NOT NULL auto_increment,
    'name' varchar(255) NOT NULL,
    'description' varchar(1000) default NULL,
    'price' decimal(10,0) NOT NULL,
    PRIMARY KEY ('id')
) ENGINE=MyISAM AUTO_INCREMENT=6 DEFAULT CHARSET=utf8;
```

---

app10d 中添加了一个 DAO 模块，这个模块中包含以下几个接口和类：

- 代码清单 10-12 中的 DAO 接口，所有 DAO 接口均衍生于此。
- 代码清单 10-13 中的 BaseDAO 类，它为所有的 DAO 类提供基础实现。
- 代码清单 10-14 中的 DataSourceCache 类，这是一个辅助工具类，负责查找容器管理的 DataSource，并进行缓存。
- 代码清单 10-15 中的 DAOFactory 类，这是一个创建各种 DAO 实现的工厂类（Factory Class）。
- 代码清单 10-16 中的 DAOException 类，这是 DAO 方法在运行时抛出的异常事件。
- 代码清单 10-17 中的 ProductDAO 和代码清单 10-18 中的 ProductDAOImpl 类。这两个类都提供了方法来持久化 Product 实例，以及从数据库中获取那些实例。

#### 代码清单 10-12 DAO 接口

---

```
package app10d.dao;
import java.sql.Connection;

public interface DAO {
    Connection getConnection() throws DAOException;
}
```

---

#### 代码清单 10-13 BaseDAO 类

---

```
package app10d.dao;
import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.Statement;
import javax.sql.DataSource;

public class BaseDAO implements DAO {
```

```

public Connection getConnection() throws DAOException {
    DataSource dataSource =
        DataSourceCache.getInstance().getDataSource();
    try {
        return dataSource.getConnection();
    } catch (Exception e) {
        e.printStackTrace();
        throw new DAOException();
    }
}

protected void closeDBObjects(ResultSet resultSet, Statement
    statement,
    Connection connection) {
    if (resultSet != null) {
        try {
            resultSet.close();
        } catch (Exception e) {
        }
    }
    if (statement != null) {
        try {
            statement.close();
        } catch (Exception e) {
        }
    }
    if (connection != null) {
        try {
            connection.close();
        } catch (Exception e) {
        }
    }
}
}
}

```

代码清单 10-14 DataSourceCache 类

---

```

package app10d.dao;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.sql.DataSource;

public class DataSourceCache {
    private static DataSourceCache instance;
    private DataSource dataSource;
    static {
        instance = new DataSourceCache();
    }

    private DataSourceCache() {

```

```

        Context context = null;
        try {
            context = new InitialContext();
            dataSource = (DataSource) context.lookup(
                "java:comp/env/jdbc/myDataSource");
        } catch (NamingException e) {
        }
    }

    public static DataSourceCache getInstance() {
        return instance;
    }

    public DataSource getDataSource() {
        return dataSource;
    }
}

```

代码清单 10-15 DAOFactory 类

```

package app10d.dao;
public class DAOFactory {
    public static ProductDAO getProductDAO() {
        return new ProductDAOImpl();
    }
}

```

代码清单 10-16 DAOException 类

```

package app10d.dao;

public class DAOException extends Exception {
    private static final long serialVersionUID = 19192L;

    public DAOException() {
    }
    public DAOException(String message) {
        this.message = message;
    }
    public String getMessage() {
        return message;
    }
    public void setMessage(String message) {
        this.message = message;
    }
    private String message;

    public String toString() {
        return message;
    }
}

```

代码清单 10-17 ProductDAO 接口

---

```
package app10d.dao;
import java.util.List;
import app10d.model.Product;

public interface ProductDAO extends DAO {
    List<Product> getProducts() throws DAOException;
    void insert(Product product) throws DAOException;
}
```

---

代码清单 10-18 ProductDAOImpl 类

```
package app10d.dao;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
import app10d.model.Product;

public class ProductDAOImpl extends BaseDAO implements ProductDAO {

    private static final String GET_PRODUCTS_SQL =
        "SELECT name, description, price FROM products";
    public List<Product> getProducts() throws DAOException {
        List<Product> products = new ArrayList<Product>();
        Connection connection = null;
        PreparedStatement pStatement = null;
        ResultSet resultSet = null;
        try {
            connection = getConnection();
            pStatement = connection.prepareStatement(
                GET_PRODUCTS_SQL);
            resultSet = pStatement.executeQuery();
            while (resultSet.next()) {
                Product product = new Product();
                product.setName(resultSet.getString("name"));
                product.setDescription(
                    resultSet.getString("description"));
                product.setPrice(resultSet.getFloat("price"));
                products.add(product);
            }
        } catch (SQLException e) {
            throw new DAOException("Error getting products. "
                + e.getMessage());
        } finally {
            closeDBObjects(resultSet, pStatement, connection);
        }
        return products;
    }
}
```

```

private static final String INSERT_PRODUCT_SQL =
    "INSERT INTO products " +
    "(name, description, price) " +
    "VALUES (?, ?, ?);"
public void insert(Product product)
    throws DAOException {
    Connection connection = null;
    PreparedStatement pStatement = null;
    try {
        connection = getConnection();
        pStatement = connection.prepareStatement(
            INSERT_PRODUCT_SQL);
        pStatement.setString(1, product.getName());
        pStatement.setString(2,
            product.getDescription());
        pStatement.setFloat(3, product.getPrice());
        pStatement.execute();
    } catch (SQLException e) {
        throw new DAOException("Error adding product. " +
            + e.getMessage());
    } finally {
        closeDBObjects(null, pStatement, connection);
    }
}

```

---

这些 DAO 接口和类的作用都是不言自明的。我们有了进行数据库访问的 DAO 模块之后，SaveProductAction 类终于可以做它要做的事情了，也就是保存产品。app10d 中的 SaveProductAction 如代码清单 10-19 所示。

**代码清单 10-19 app10d 中的 SaveProductAction 类**

```

package app10d.action;
import app10d.dao.DAOException;
import app10d.dao.DAOFactory;
import app10d.dao.ProductDAO;
import app10d.model.Product;

public class SaveProductAction {
    public void save(Product product) {
        ProductDAO productDAO = DAOFactory.getProductDAO();
        try {
            productDAO.insert(product);
        } catch (DAOException e) {
            e.printStackTrace();
        }
    }
}

```

---

因为有了 DAO 模式，SaveProductAction 中的 save 方法就可以保持条理性。它只要从 DAOFactory 中获得一个 ProductDAO，并在 ProductDAO 中调用 insert 方法即可。

app10d 中新增了一个 Action：product\_list，它基于代码清单 10-20 中的 GetProductsAction 类。它有一个 getProducts 方法，用来返回一个 Product 的 List。

代码清单 10-20 GetProductsAction 类

---

```
package app10d.action;
import java.util.List;
import app10d.dao.DAOException;
import app10d.dao.DAOFactory;
import app10d.dao.ProductDAO;
import app10d.model.Product;

public class GetProductsAction {
    public List<Product> getProducts() {
        ProductDAO productDAO = DAOFactory.getProductDAO();
        List<Product> products = null;
        try {
            products = productDAO.getProducts();
        } catch (DAOException e) {

        }
        return products;
    }
}
```

---

getProducts 方法获得一个 ProductDAO，并调用它的 getProducts 方法。

app10d 中最后那部分代码中值得关注的是代码清单 10-21 所示的部署描述符。

代码清单 10-21 app10d 的部署描述符

---

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
  version="3.0">
  <welcome-file-list>
    <welcome-file>product_list</welcome-file>
  </welcome-file-list>
</web-app>
```

---

还要注意的是，Controller Servlet 中新增了一个 URL 模式（/product\_list），表示这个欢迎文件实际上是引用了 Controller Servlet。

```

@WebServlet(name = "ControllerServlet", urlPatterns = {
    "/product_input", "/product_save", "/product_list" })
public class ControllerServlet extends HttpServlet {

```

测试 app10d 时，使用下面这个 URL。由于欢迎文件是在部署描述符中定义的，因此不需要指定具体的资源。

<http://localhost:8080/app10d>

图 10-8 中展示了数据库中的所有产品。一开始，应该只会看到一个空白列表，因为还需要插入产品。因此，现在先插入产品，之后再重新访问该页面。

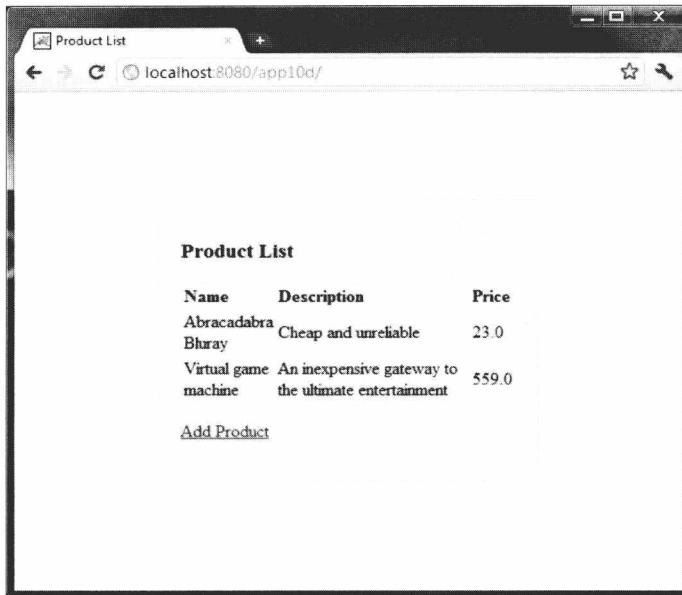


图 10-8 app10d 中的欢迎文件

## 10.7 依赖注入

在过去的几年间，为了能够编写出易于测试的代码，依赖注入（Dependency Injection）也作为其中一种解决方案被广泛使用。事实上，依赖注入的背后有着像 Spring 和 Struts 2 这样的框架支持。那么，到底什么是依赖注入呢？

关于这个问题，Martin Fowler 曾写过一篇很精彩的论文，详细内容请查看以下网址：

<http://martinfowler.com/articles/injection.html>

在 Fowler 将其命名为“依赖注入”之前，人们经常用“控制反转”（Inversion of Control, IoC）来表示它。如 Fowler 在其文中谈到的那样，这两者其实是一回事。

假如有两个组件 A 和 B，A 依赖 B，那么就可以说 A 依赖于 B，或者说 B 是 A 的一个依赖。假设 A 有一个方法 importantMethod，它像下面这样使用 B：

```
public class A {
    public void importantMethod() {
        B b = ... // get an instance of B
        b.usefulMethod();
        ...
    }
    ...
}
```

在 A 使用 B 之前，A 必须获得 B 的一个实例。如果 B 是一个 Java 实体类，那么就会像使用 new 关键字一样简单，但是如果 B 不是 Java 实体类，并且 B 还有许多个实现，那就会比较麻烦了。你必须选择一个 B 实现，并且这么做还会降低 A 的可重用性，因为你还没有选择好 B 的实现，将无法使用 A。

以 app10d 中的 ProductDAO 接口（如代码清单 10-17 所示）为例。它的实现 ProductDAOImpl（如代码清单 10-18 所示）依赖于其父类 BaseDAO 的 getConnection 方法来获得一个连接。问题在于，这个连接来自一个由容器维护的 JNDI 对象。那么在容器之外如何对 ProductDAO 进行测试呢？除非修改 BaseDAO 类。由于 ProductDAO（以及可能添加的所有 DAO 实现）依赖于另一个对象，它的易测试性降低了。要想测试 app10d 中的 DAO 对象，必须先将整个应用程序部署在一个容器中，并用一个浏览器输入 DAO 对象的值。这么做会损失多少生产效率呢？

有了依赖注入，每个组件都将其依赖注入其中，这使得对每个组件的测试变得更加轻松，甚至可以说是易如反掌！对于要在依赖注入环境中使用的类，必须提前将它注入。一种方法是给每个依赖都创建一个 set 方法。例如代码清单 10-22 中的 SaveProductAction 类和代码清单 10-23 中的 getProductsAction 类都是 app10e 应用程序的一部分。使这个类区别于前一个例子中的祖先类的是，它们有一个 setProductDAO 方法，依赖注入框架可以调用它来注入 ProductDAO。注入也可以在构造器或者类域中进行，但在本书的范例中是坚持在 set 方法中进行的。

#### 代码清单 10-22 提前注入的 SaveProductAction 修改版

---

```
package app10e.action;
import app10e.dao.DAOException;
import app10e.dao.ProductDAO;
import app10e.model.Product;

public class SaveProductAction {

    private ProductDAO productDAO;
    public void setProductDAO(ProductDAO productDAO) {
```

```

        this.productDAO = productDAO;
    }
    public void save(Product product) {
        try {
            productDAO.insert(product);
        } catch (DAOException e) {
            e.printStackTrace();
        }
    }
}

```

代码清单 10-23 GetProductsAction 类

```

package app10e.action;
import java.util.List;
import app10e.dao.DAOException;
import app10e.dao.ProductDAO;
import app10e.model.Product;

public class GetProductsAction {
    private ProductDAO productDAO;
    public void setProductDAO(ProductDAO productDAO) {
        this.productDAO = productDAO;
    }

    public List<Product> getProducts() {
        List<Product> products = null;
        try {
            products = productDAO.getProducts();
        } catch (DAOException e) {

        }
        return products;
    }
}

```

一旦预先注入好所有的类，就可以选择一种依赖注入框架，并将它导入到你的工程中。Spring Framework、Google Guice、Weld 和 PicoContainer 都是一些不错的选择。

**提示** Java 中的依赖注入是在 JSR 330 和 JSR 299 中规定的，这两者都不在本书讨论的范畴之内。

目前为止，理论方面讲的够多了，现在我们来看一个依赖注入的范例 app10e。

为了便于理解，app10e 应用程序用代码清单 10-24 中的 **DependencyInjector** 类代替依赖注入框架。（在现实的应用程序中，当然必须使用某种适当的框架。）这个类是专门为 app10e 设计的，有没有容器都很容易被实例化。一旦被实例化，就必须调用它的 **start** 方法，准备一个可供其他组件使用的 C3P0 连接池。用完之后，应该调用它的 **shutdown** 方法

以释放资源。

代码清单 10-24 DependencyInjector 类

```

package app10e.util;
import javax.sql.DataSource;
import app10e.action.GetProductsAction;
import app10e.action.SaveProductAction;
import app10e.dao.ProductDAO;
import app10e.dao.ProductDAOImpl;
import app10e.validator.ProductValidator;
import com.mchange.v2.c3p0.ComboPooledDataSource;
import com.mchange.v2.c3p0.DataSources;

public class DependencyInjector {
    private DataSource dataSource;

    public void start() {
        // create dataSource
        ComboPooledDataSource cpds = new ComboPooledDataSource();
        try {
            cpds.setDriverClass("com.mysql.jdbc.Driver");
        } catch (Exception e) {
            e.printStackTrace();
        }
        cpds.setJdbcUrl("jdbc:mysql://localhost:3306/test");
        cpds.setUser("testuser");
        cpds.setPassword("secret");

        // to override default settings:
        cpds.setMinPoolSize(5);
        cpds.setAcquireIncrement(5);
        cpds.setMaxPoolSize(20);
        dataSource = cpds;
    }

    public void shutDown() {
        // destroy dataSource
        try {
            DataSources.destroy(dataSource);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

/*
 * Returns an instance of type. type is of type Class
 * and not String because it's easy to misspell a class name
 */
public Object getObject(Class type) {
    if (type == ProductValidator.class) {

```

```

        return new ProductValidator();
    } else if (type == ProductDAO.class) {
        return createProductDAO();
    } else if (type == GetProductsAction.class) {
        return createGetProductsAction();
    } else if (type == SaveProductAction.class) {
        return createSaveProductAction();
    }
    return null;
}

private GetProductsAction createGetProductsAction() {
    GetProductsAction getProductsAction = new
    GetProductsAction();
    // inject a ProductDAO to getProductsAction
    getProductsAction.setProductDAO(createProductDAO());
    return getProductsAction;
}

private SaveProductAction createSaveProductAction() {
    SaveProductAction saveProductAction = new
        SaveProductAction();
    // inject a ProductDAO to saveProductAction
    saveProductAction.setProductDAO(createProductDAO());
    return saveProductAction;
}

private ProductDAO createProductDAO() {
    ProductDAO productDAO = new ProductDAOImpl();
    // inject a DataSource to productDAO
    productDAO.setDataSource(dataSource);
    return productDAO;
}
}

```

---

**DependencyInjector** 类硬编码了许多值，包括 JDBC URL 和数据库的用户名及密码。在框架中，属性值一般来自可编辑的配置文件，形式通常为 XML 文档。在本例中我们把它简化了。

为了从 **DependencyInjector** 中获得对象，需调用其 **getObject** 方法，传递目标对象的 Class。**DependencyInjector** 在 app10e 中支持以下类型：**ProductValidator**、**ProductDAO**、**GetProductsAction** 及 **SaveProductAction**。例如，为了获得一个 **ProductDAO** 实例，要通过传递 **ProductDAO.class** 来调用 **getObject**：

```

ProductDAO productDAO = (ProductDAO)
    dependencyInjector.getObject(ProductDAO.class);

```

**DependencyInjector**（及所有依赖注入框架）的魅力在于，它所返回的对象也和依赖一同被注入了。如果某一个依赖中还具有其他的依赖，那么这个依赖的依赖也会随之一起被注入。

app10e 中的 Servlet Controller 如代码清单 10-25 所示。注意，它在其 init 方法中实例化 DependencyInjector，并在它的 destroy 方法中调用 DependencyInjector 的 shutdown 方法。Servlet 不再创建自己的依赖，而是从 DependencyInjector 处获得。

代码清单 10-25 app10e 中的 ControllerServlet 类

```
package app10e.servlet;
import java.io.IOException;
import java.util.List;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import app10e.util.DependencyInjector;
import app10e.action.GetProductsAction;
import app10e.action.SaveProductAction;
import app10e.form.ProductForm;
import app10e.model.Product;
import app10e.validator.ProductValidator;

@WebServlet(name = "ControllerServlet", urlPatterns = {
    "/product_input", "/product_save", "/product_list" })
public class ControllerServlet extends HttpServlet {

    private static final long serialVersionUID = 6679L;
    private DependencyInjector dependencyInjector;

    @Override
    public void init() {
        dependencyInjector = new DependencyInjector();
        dependencyInjector.start();
    }

    @Override
    public void destroy() {
        dependencyInjector.shutDown();
    }

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

    @Override
    public void doPost(HttpServletRequest request,
                      HttpServletResponse response)
```

```

        throws IOException, ServletException {
    process(request, response);
}

private void process(HttpServletRequest request,
                     HttpServletResponse response)
    throws IOException, ServletException {

    String uri = request.getRequestURI();
    /*
     * uri is in this form: /contextName/resourceName,
     * for example: /app10a/product_input.
     * However, in the case of a default context, the
     * context name is empty, and uri has this form
     * /resourceName, e.g.: /product_input
     */
    int lastIndex = uri.lastIndexOf("/");
    String action = uri.substring(lastIndex + 1);
    String dispatchUrl = null;

    if (action.equals("product_input")) {
        // no action class, there is nothing to be done
        dispatchUrl = "/jsp/ProductForm.jsp";
    } else if (action.equals("product_save")) {
        // instantiate action class
        ProductForm productForm = new ProductForm();
        // populate action properties
        productForm.setProductName(
            request.getParameter("productName"));
        productForm.setDescription(
            request.getParameter("description"));
        productForm.setPrice(request.getParameter("price"));

        // validate ProductForm
        ProductValidator productValidator = (ProductValidator)
            dependencyInjector.getObject(
                ProductValidator.class);
        List<String> errors =
            productValidator.validate(productForm);
        if (errors.isEmpty()) {
            // create Product from ProductForm
            Product product = new Product();
            product.setProductName(
                productForm.getProductName());
            product.setDescription(
                productForm.getDescription());
            product.setPrice(Float.parseFloat(
                productForm.getPrice()));

            // no validation error, execute action method
            SaveProductAction saveProductAction =
    
```

```

        (SaveProductAction)
        dependencyInjector.getObject(
            SaveProductAction.class);
        saveProductAction.save(product);

        // store action in a scope variable for the view
        request.setAttribute("product", product);
        dispatchUrl = "/jsp/ProductDetails.jsp";
    } else {
        request.setAttribute("errors", errors);
        dispatchUrl = "/jsp/ProductForm.jsp";
    }
} else if (action.equals("product_list") ||
    action.isEmpty()) {
    GetProductsAction getProductsAction =
        (GetProductsAction)
        dependencyInjector.getObject(
            GetProductsAction.class);
    List<Product> products =
        getProductsAction.getProducts();
    request.setAttribute("products", products);
    dispatchUrl = "/jsp/ProductList.jsp";
}

// forward to a view
if (dispatchUrl != null) {
    RequestDispatcher rd =
        request.getRequestDispatcher(dispatchUrl);
    rd.forward(request, response);
}
}
}

```

---

注意，在 app10e 中，我们编写了自己的依赖注入器，并在 Servlet 初始化时将它实例化。如果你使用的是 Java 6 EE 容器，如 Glassfish，还可以让容器将依赖注入到 Servlet 中。这个 Servlet 应该像这样：

```

public class ControllerServlet extends HttpServlet {
    @Inject ProductValidator productValidator;
    @Inject GetProductsAction getProductsAction;
    @Inject SaveProductAction saveProductAction;
    ...

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

```

```

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

```

运行 app10e 时，Tomcat 的 Context 不需要有 Resource 声明，只要像下面这样：

```
<Context path="/app10e" docBase="/path/to/app">
</Context>
```

调用下面这个 URL 来测试 app10e：

<http://localhost:8080/app10e>

因为有了依赖注入器，app10e 中的每一个组件都可以独立进行测试。例如，运行代码清单 10-26 中的 ProductDAOTest 类可以测试 ProductDAO 的 insert 方法。

代码清单 10-26 ProductDAOTest 类

---

```

package test.app10e.dao;
import util.DependencyInjector;
import app10e.dao.DAOException;
import app10e.dao.ProductDAO;
import app10e.model.Product;

public class ProductDAOTest {

    public static void main(String[] args) {
        DependencyInjector injector = new DependencyInjector();
        try {
            injector.start();
            ProductDAO productDAO = (ProductDAO)
                injector.getObject(ProductDAO.class);
            Product product = new Product();
            product.setName("New Product");
            product.setDescription("Testing");
            product.setPrice(20.20f);
            try {
                productDAO.insert(product);
            } catch (DAOException e) {
                e.printStackTrace();
            }
        } finally {
            injector.shutDown();
        }
    }
}

```

---

为了能够成功地运行 ProductDAOTest 类, classpath 中必须包含 C3P0 类库及 MySQL JDBC 驱动程序。当然, 还要用一个测试框架 (如 JUnit) 进行单元测试。

## 10.8 小结

本章学习了基于 MVC 模式的 Model 2 架构, 并学习了如何利用 Servlet Controller 或者 Filter Dispatcher 编写 Model 2 应用程序。这两种 Model 2 应用程序分别在 app10a 和 app10b 中进行了示范。用 Servlet 代替 Filter 作为 Controller 的一大明显好处在于, 可以将 Servlet 配置成一个欢迎页面。在 Model 2 应用程序中, 经常用 JSP 页面作为 View, 当然也可以使用其他技术, 如 Apache Velocity 和 FreeMarker。如果用 JSP 页面作为 Model 2 架构中的 View, 那么将只用那些页面来显示值, 上面不应该出现任何脚本元素。

在本章中, 我们还构建了一个简单的 MVC 框架, 将这些组件整合在一个验证器和一个依赖注入框架中。虽然自制框架可以作为一个很好的教育工具, 但是实际的 MVC 项目还是应该基于成熟的 MVC 框架, 如 Struts 2 或者 Spring MVC, 不必重复发明轮子。

# 第 11 章 文件上传

在 Servlet 技术出现之后不久的那段时期，文件上传编程仍然是一项比较具有挑战性的任务，包括在服务器端解析原始的 HTTP 响应。为了减轻编程负担，开发者通常会使用商业的文件上传组件，它们当中有些价格不菲。值得庆幸的是，Apache Software Foundation 于 2003 年发布了它的开源 Commons FileUpload 组件，并且很快就受到世界各地 Servlet/JSP 程序员的追捧。

几年之后，Servlet 的设计者才意识到文件上传的重要性，最终，文件上传在 Servlet 3 中成了一项内置的特性。Servlet 3 的开发者不再需要将 Commons FileUpload 组件导入到他们的工程中去。

本章介绍如何使用 Servlet 3 文件上传特性，以及在客户端使用时还需要哪些东西。本章还将示范如何采用 HTML 5 来提升用户体验。

## 11.1 客户端编程

要上传文件，必须利用 `multipart/form-data` 设置 HTML 表单的 `enctype` 属性值，像下面这样：

```
<form action="action" enctype="multipart/form-data" method="post">
    Select a file <input type="file" name="fieldName"/>
    <input type="submit" value="Upload"/>
</form>
```

这个表单中必须包含类型 `file` 的一个输入元素，它会被显示成一个按钮，单击它时，会打开一个对话框，供我们选择文件。表单中还可以包含其他域类型，如文本域或隐藏域。

在 HTML 5 之前，如果想要上传多个文件，则不得不使用多个文件 `input` 元素。但是 HTML 5 在 `input` 元素中引入了 `multiple` 属性，使得上传多个文件变得更加简单。我们可以在 HTML 5 中编写以下任意一行代码，以便生成一个按钮供选择多个文件。

```
<input type="file" name="fieldName" multiple/>
<input type="file" name="fieldName" multiple="multiple"/>
<input type="file" name="fieldName" multiple="" />
```

## 11.2 服务器端编程

Servlet 中的服务器端文件上传编程主要围绕着 `MultipartConfig` 注解类型和 `javax.servlet.http.Part` 接口进行。处理上传文件的 Servlet 必须用 `@MultipartConfig` 进行标注。`MultipartConfig` 可以带有以下属性，这些全部都是可选的：

- `maxFileSize`, 表示最多可上传的文件容量。超过设定值的文件将会遭到拒绝。  
`maxFileSize` 的默认值为 `-1`, 表示不受限制。

- `maxRequestSize`, 表示允许多部分 HTTP 请求的最大容量。它的默认值为 `-1`, 意味着它是不受限制的。

- `location`, 将上传的文件保存到磁盘中的指定位置, 调用 `Part` 中的 `write` 方法将用到它。

- `fileSizeThreshold`, 设定一个溢出尺寸, 超过这个值之后, 上传的文件将被写入磁盘。

在一个由多部分组成的请求中, 每一个表单域, 包括非文件域, 都会被转换成一个 `Part`。`HttpServletRequest` 接口定义了以下方法来处理多部分的请求:

```
Part getPart(java.lang.String name)
```

返回与指定名称相关的 `Part`。

```
java.util.Collection<Part> getParts()
```

返回这个请求中的所有 `Part`。

`Part` 接口中还具有以下方法:

```
java.lang.String getName()
```

获取到这部分的名称, 例如相关表单域的名称。

```
java.lang.String getContentType()
```

如果 `Part` 是一个文件, 那么将返回 `Part` 的内容类型, 否则返回 `null`。

```
java.util.Collection<java.lang.String> getHeaderNames()
```

返回这个 `Part` 中的所有标头名称。

```
java.lang.String getHeader(java.lang.String headerName)
```

返回指定标头名称的值。

```
java.util.Collection<java.lang.String> getHeaders(java.lang.String headerName)
```

返回这个 `Part` 中所有标头的名称。

```
void write(java.lang.String path)
```

将上传的文件写入磁盘中。如果 *path* 是一个绝对路径，那么将写入指定的路径。如果 *path* 是一个相对路径，那么将被写入相对于 `MultipartFileConfig` 注解的 `location` 属性值的指定路径。

```
void delete()
```

删除该文件对应的存储，包括相关的临时文件。

```
java.io.InputStream getInputStream()
```

以 `InputStream` 的形式返回上传文件的内容。

如果相关的 HTML 输入是一个文件 `input` 元素，则 `Part` 将返回这些标头：

```
content-type:contentType  
content-disposition:form-data; name="fieldName"; filename="fileName"
```

例如，上传输入域中一个名为 `document` 的 `note.txt` 文件时，将导致相关的部分也具有这些标头：

```
content-type:text/plain  
content-disposition:form-data; name="document"; filename="note.txt"
```

如果没有选择任何文件，还是会为该文件域创建一个 `Part`，但是相关标头将如下：

```
content-type:application/octet-stream  
content-disposition:form-data; name="document"; filename=""
```

`Part` 接口的 `getName` 返回与这部分有关的域名称，而不是上传文件的名称。要想返回后者，需要解析 `content-disposition` 标头。

对于非文件的域，`Part` 将只有 `content-disposition` 标头，其格式如下：

```
content-disposition:form-data; name="fieldName"
```

在 Servlet 中处理上传文件时，需要：

- 通过查看是否存在 `content-type` 标头，检验一个 `Part` 是属于普通的表单域，还是文件域。你也可以通过在 `Part` 中调用 `getContentType` 方法或者 `getHeader("content-type")` 来完成检查。
- 如果有 `content-type` 标头，那么将查看上传文件名称是否为空。文件名为空，表示有文件类型的域存在，但是没有选择要上传的文件。
- 如果文件存在，就可以调用 `Part` 中的 `write` 方法来写入磁盘，调用时同时传递一个绝对路径，或是相对于 `MultipartConfig` 注解的 `location` 属性的路径。

### 11.3 上传 Servlet 范例

下面的例子中示范了如何编写一个多部分的 Servlet。这个 Servlet 类名为 SingleUploadServlet，如代码清单 11-1 所示，它可以处理单个文件上传，并且用一个 @MultipartConfig 进行了标注。

代码清单 11-1 SingleUploadServlet 类

---

```

package appila.servlet;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.ServletException;
import javax.servlet.annotation.MultipartConfig;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Part;

@WebServlet(urlPatterns = { "/singleUpload" })
@MultipartConfig
public class SingleUploadServlet extends HttpServlet {

    private static final long serialVersionUID = 8593038L;

    private String getFilename(Part part) {
        String contentDispositionHeader =
            part.getHeader("content-disposition");
        String[] elements = contentDispositionHeader.split(";");
        for (String element : elements) {
            if (element.trim().startsWith("filename")) {
                return element.substring(element.indexOf('=') + 1)
                    .trim().replace("\"", "");
            }
        }
        return null;
    }

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

        // save uploaded file to WEB-INF
        Part part = request.getPart("filename");
        String fileName = getFilename(part);
        if (fileName != null && !fileName.isEmpty()) {
            part.write(getServletContext().getRealPath(
                "/WEB-INF") + "/" + fileName);
        }
    }
}

```

```

    // write to browser
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.print("<br/>Uploaded file name: " + fileName);
    writer.print("<br/>Size: " + part.getSize());
    String author = request.getParameter("author");
    writer.print("<br/>Author: " + author);
}
}

```

---

多部分 Servlet 中一个很重要的方法是 `getFilename` 方法，它返回上传的文件名称。注意，`Part` 接口的 `getName` 方法是返回域名称，而不是上传的文件名称。

`getFilename` 方法获取 `content-disposition` 标头，将它分解，然后遍历这些元素，直到它找到 `filename` 元素为止。

```

private String getFilename(Part part) {
    String contentDispositionHeader =
        part.getHeader("content-disposition");
    String[] elements = contentDispositionHeader.split(";");
    for (String element : elements) {
        if (element.trim().startsWith("filename")) {
            return element.substring(element.indexOf('=') + 1)
                .trim().replace("\\\"", "\"");
        }
    }
    return null;
}

```

为了测试多部分的 Servlet，可以打开下面这个 URL 来调用代码清单 11-2 中的 `singleUpload.jsp` 页面。

`http://localhost:8080/app11a/singleUpload.jsp`

代码清单 11-2 singleUpload.jsp 页面

---

```

<!DOCTYPE HTML>
<html>
<body>
<h1>Select a file to upload</h1>
<form action="singleUpload" enctype="multipart/form-data"
      method="post">
    Author: <input type="text" name="author"/><br/>
    Select file to upload <input type="file" name="filename"/><br/>
    <input type="submit" value="Upload"/>
</form>
</body>
</html>

```

---

图 11-1 展示了显示完成的页面。注意，文件 input 元素被显示成一个可单击的按钮，用来打开一个 Open File 对话框。不同浏览器上的按钮标签是不同的。在 Chrome 中，为 Choose File。在 Internet Explorer 和 Firefox 中，为 Browse。而且，Chrome 会在按钮的右侧显示所选择的文件，Firefox 和 IE 则不显示。

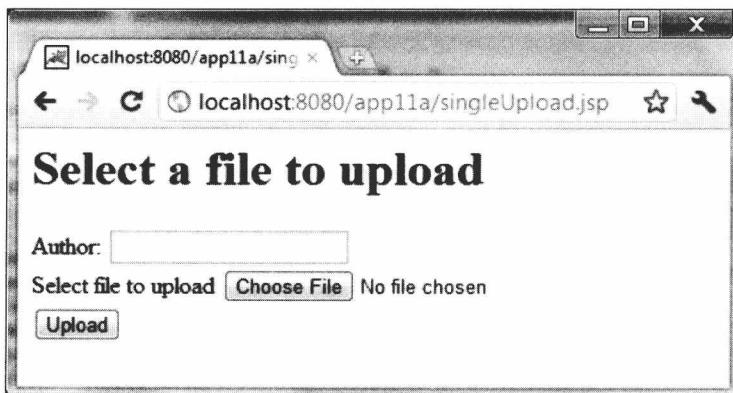


图 11-1 文件上传表单

现在选择一个文件，在 Author 域中输入一个值，并单击 Upload 按钮。你会在浏览器中看到上传的文件名称，以及文件大小，如图 11-2 所示。

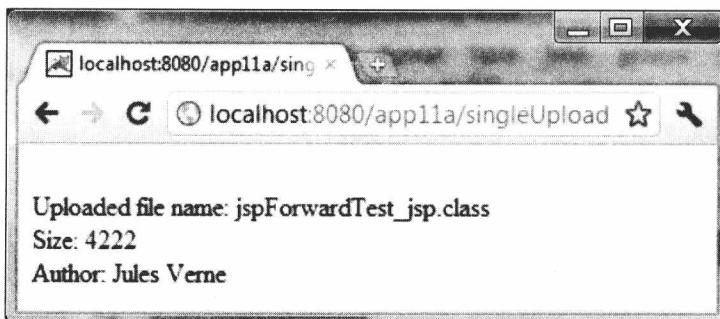


图 11-2 SingleUploadServlet 的输出

## 11.4 多文件上传

再举个例子。代码清单 11-3 中的 `MultipleUploadsServlet` 展示了如何同时上传多个文件。这个 Servlet 与前一个例子中的类似，只不过它是遍历 `Part` 集合，努力核实 `Part` 中是包含了一个文件呢，还是包含了一个域值。

## 代码清单 11-3 MultipleUploadsServlet 类

```

package appl1a.servlet;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Collection;
import javax.servlet.ServletException;
import javax.servlet.annotation.MultipartConfig;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.Part;

@WebServlet(urlPatterns = { "/multipleUploads" })
@MultipartConfig
public class MultipleUploadsServlet extends HttpServlet {

    private static final long serialVersionUID = 9991L;

    private String getFilename(Part part) {
        String contentDispositionHeader =
            part.getHeader("content-disposition");
        String[] elements = contentDispositionHeader.split(";");
        for (String element : elements) {
            if (element.trim().startsWith("filename")) {
                return element.substring(element.indexOf('=') + 1)
                    .trim().replace("\"", "");
            }
        }
        return null;
    }

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response) throws ServletException,
                           IOException {
        response.setContentType("text/html");
        PrintWriter writer = response.getWriter();

        Collection<Part> parts = request.getParts();
        for (Part part : parts) {
            if (part.getContentType() != null) {
                // save file Part to disk
                String fileName = getFilename(part);
                if (fileName != null && !fileName.isEmpty()) {
                    part.write(getServletContext().getRealPath(
                        "/WEB-INF") + "/" + fileName);
                    writer.print("<br/>Uploaded file name: " +
                               fileName);
                    writer.print("<br/>Size: " + part.getSize());
                }
            } else {
        }
    }
}

```

```

        // print field name/value
        String partName = part.getName();
        String fieldValue = request.getParameter(partName);
        writer.print("<br/>" + partName + ":" + 
                    fieldValue);
    }
}
}
}

```

---

首先，MultipleUploadsServlet 类的 `doPost` 方法要获得 `HttpServletRequest` 中的所有 `Part`:

```
Collection<Part> parts = request.getParts();
```

然后，它迭代整个集合和每一个 `Part`，通过调用 `getContentType` 方法，查看是否存在 `content-type` 标头。

```

for (Part part : parts) {
    if (part.getContentType() != null) {
        ...
    }
}

```

内容类型表示在客户端存在文件 `input` 元素，你可以核实是否上传了文件。如果找到了文件，`doPost` 方法就会将文件的内容写入磁盘，并将某些信息发送到浏览器。

```

// save file Part to disk
String fileName = getFilename(part);
if (fileName != null && !fileName.isEmpty()) {
    part.write(getServletContext().getRealPath(
        "/WEB-INF") + "/" + fileName);
    writer.print("<br/>Uploaded file name: " +
                fileName);
    writer.print("<br/>Size: " + part.getSize());
}

```

假如不存在内容类型，则意味着 `Part` 表示的是一个非文件域，可以在 `HttpServletRequest` 中调用 `getParameter` 来获取域值:

```

// print field name/value
String partName = part.getName();
String fieldValue = request.getParameter(partName);
writer.print("<br/>" + partName + ":" + 
            fieldValue);

```

要想测试多个文件上传的 Servlet，可以打开以下 URL 来调用代码清单 11-4 中的 `multipleUploads.jsp` 页面:

<http://localhost:8080/app11a/multipleUploads.jsp>

代码清单 11-4 multipleUploads.jsp 页面

---

```
<!DOCTYPE HTML>
<html>
<body>
<h1>Select a file to upload</h1>
<form action="multipleUploads" enctype="multipart/form-data"
      method="post">
    Author : <input name="author"/><br/>
    First file to upload <input type="file" name="filename"/>
    <br/>
    Second file to upload <input type="file" name="filename"/>
    <br/>
    <input type="submit" value="Upload"/>
</form>
</body>
</html>
```

---

## 11.5 上传客户端

虽然 Servlet 3 中的文件上传特性使得在服务器端进行文件上传编程变得不费吹灰之力，但它并不能提升用户的体验。单独一个 HTML 表单并不能帮你显示进度条，或者显示上传成功的文件数。开发者们使用了不同的技术来改善用户界面，例如单独用一个浏览器线程来向服务器发出请求，以便可以报告上传进度，或者利用像 Java Applets、Adobe Flash 或者 Microsoft Silverlight 这样的第三方技术。

第三方技术也是可行的，但或多或少会有一些局限性。使用这些技术的第一个缺点在于，所有主流浏览器中都没有内建对它们的支持。例如，只有已经在电脑上安装了 Java 的用户才能运行 Java Applets。虽然有些计算机厂商，如 Dell 和 HP，他们的产品中出厂时已经安装了 Java，但是像 Lenovo 这些厂商则没有安装。虽然有办法检测用户的机器上是否安装了 Java，如果没有安装则引导用户去进行安装，但这毕竟是一种打扰，并非所有的用户都能够接受。除此之外，在默认情况下，Java Applets 对本地文件系统的访问也非常受限制，除非它们进行了数字签名。这些都明显地增加了程序的成本和复杂性。

Flash 也具有与 Applets 同样的问题。Flash 程序需要用一个播放器来运行，但并非所有的平台都支持 Flash。用户必须安装一种 Flash 播放器，才能支持浏览器中的 Flash 程序。此外，Apple 公司也不允许在 iPad 和 iPhone 中运行 Flash，Adobe 公司最终中止了 Flash 在移动平台上的运行。

Microsoft Silverlight 也需要用播放器来运行，非 IE 的浏览器都不会自带。因此，Silverlight 程序员基本上或多或少都会有 Applet 和 Flash 开发者们遇到的那些问题。

令我们高兴的是，HTML 5 帮助我们解决了这个难题。

HTML 5 在它的 DOM 中添加了一个 File API，允许进行本地文件访问。与 Applets、Flash 和 Silverlight 相比，针对客户端的文件上传局限性，HTML 5 似乎是完美的解决方案了。令人遗憾的是，在编写本书之时，Internet Explorer 9 还不能完全支持这个 API。但你可以用最新版的 Firefox、Chrome 和 Opera 对这个例子进行测试。

为了示范 HTML 5 的强大功能，app11b 中的 html5.jsp 页面（如代码清单 11-5 所示）利用 JavaScript 和 HTML 5 File API 来提供进度条，以报告上传的进度。app11b 应用程序中也包含了一个 MultipleUploadsServlet 类，用来将上传的文件保存在服务器中。但是，因为 Javascript 不在本书讨论的范畴，因此将只是粗略地一笔带过。

简而言之，我们关注的是 HTML 5 input 元素的 change 事件，它在 input 元素的值发生变化时触发。我们还关注 HTML 5 在 XMLHttpRequest 对象中添加的 progress 事件。XMLHttpRequest 当然是 AJAX 的核心了。当异步使用 XMLHttpRequest 对象来上传文件时，它会不断地触发 progress 事件，直到上传进程完成或取消，或者直到该进程因为错误而中止。通过监听 progress 事件，我们可以很容易地追踪到文件上传操作的进程。

代码清单 11-5 html5.jsp 页面

---

```

<!DOCTYPE HTML>
<html>
<head>
<script>
    var totalFileLength, totalUploaded, fileCount, filesUploaded;

    function debug(s) {
        var debug = document.getElementById('debug');
        if (debug) {
            debug.innerHTML = debug.innerHTML + '<br/>' + s;
        }
    }

    function onUploadComplete(e) {
        totalUploaded += document.getElementById('files').
            files[filesUploaded].size;
        filesUploaded++;
        debug('complete ' + filesUploaded + " of " + fileCount);
        debug('totalUploaded: ' + totalUploaded);
        if (filesUploaded < fileCount) {
            uploadNext();
        } else {
            alert('Finished uploading file(s)');
        }
    }

    function onFileSelect(e) {
        var files = e.target.files; // FileList object
        var output = [];

```

```

fileCount = files.length;
totalFileLength = 0;
for (var i=0; i<fileCount; i++) {
    var file = files[i];
    output.push(file.name, ' (',
                file.size, ' bytes, ',
                file.lastModifiedDate.toLocaleDateString(), ')'
    );
    output.push('<br/>');
    debug('add ' + file.size);
    totalFileLength += file.size;
}
document.getElementById('selectedFiles').innerHTML =
    output.join('');
debug('totalFileLength:' + totalFileLength);
}

function onUploadProgress(e) {
    if (e.lengthComputable) {
        var percentComplete = parseInt(
            (e.loaded + totalUploaded) * 100
            / totalFileLength);
        var bar = document.getElementById('bar');
        bar.style.width = percentComplete + '%';
        bar.innerHTML = percentComplete + ' % complete';
    } else {
        debug('unable to compute');
    }
}

function onUploadFailed(e) {
    alert("Error uploading file");
}

function uploadNext() {
    var xhr = new XMLHttpRequest();
    var fd = new FormData();
    var file = document.getElementById('files').
        files[filesUploaded];
    fd.append("fileToUpload", file);
    xhr.upload.addEventListener(
        "progress", onUploadProgress, false);
    xhr.addEventListener("load", onUploadComplete, false);
    xhr.addEventListener("error", onUploadFailed, false);
    xhr.open("POST", "multipleUploads");
    debug('uploading ' + file.name);
    xhr.send(fd);
}

function startUpload() {
    totalUploaded = filesUploaded = 0;
}

```

```

        uploadNext();
    }
    window.onload = function() {
        document.getElementById('files').addEventListener(
            'change', onFileSelect, false);
        document.getElementById('uploadButton').
            addEventListener('click', startUpload, false);
    }
</script>
</head>
<body>
<h1>Multiple file uploads with progress bar</h1>
<div id='progressBar' style='height:20px;border:2px solid green'>
    <div id='bar' style='height:100%;background:#33dd33;width:0%'>
    </div>
</div>
<form id='form1' action="multipleUploads"
      enctype="multipart/form-data" method="post">
    <input type="file" id="files" multiple/>
    <br/>
    <output id="selectedFiles"></output>
    <input id="uploadButton" type="button" value="Upload"/>
</form>
<div id='debug'
      style='height:100px;border:2px solid green;overflow:auto'>
</div>
</body>
</html>

```

---

html5.jsp 页面中的用户界面主要包含一个名为 **progressBar** 的 div 元素、一个表单、另一个名为 **debug** 的 div 元素。没错，**progressBar** div 是用来显示上传进度的，**debug** 是用来显示调试信息的。表单中有一个类型文件的 input 元素和一个按钮。

在这个表单中要注意两件事情。第一，被标识为 **files** 的 input 元素有一个 **multiple** 属性，可以支持多文件选择；第二，这个按钮不是一个提交按钮。因此，单击它时不会提交表单。事实上，脚本是利用 XMLHttpRequest 对象来完成上传的。

现在，我们来看一下 Javascript 代码。这里假设读者已经具备了一定的脚本语言知识。

执行脚本时，它做的第一件事是为 4 个变量分配空间。

```
var totalFileLength, totalUploaded, fileCount, filesUploaded;
```

**totalFileLength** 变量保存上传文件的总长度。**totalUploaded** 是目前为止已经上传的字节数。**fileCount** 包含了要上传的文件数，**filesUploaded** 表示已经上传的文件数。

当页面加载完成之后，将调用赋给 **window.onload** 的函数。

```
window.onload = function() {
    document.getElementById('files').addEventListener(
```

```

        'change', onFileSelect, false);
document.getElementById('uploadButton').
    addEventListener('click', startUpload, false);
}

```

这段代码用 `onFileSelect` 函数映射 `files input` 元素的 `change` 事件，用 `startUpload` 映射按钮的 `click` 事件。

每当用户在本地目录下修改了文件时，都会触发 `change` 事件。与该事件相连的事件处理器只要在一个输出元素中打印出所选文件的名称和大小即可。下面这段还是事件处理器：

```

function onFileSelect(e) {
    var files = e.target.files; // FileList object
    var output = [];
    fileCount = files.length;
    totalFileLength = 0;
    for (var i=0; i<fileCount; i++) {
        var file = files[i];
        output.push(file.name, ' (',
                    file.size, ' bytes, ',
                    file.lastModifiedDate.toLocaleDateString(), ')'
                );
        output.push('<br/>');
        debug('add ' + file.size);
        totalFileLength += file.size;
    }
    document.getElementById('selectedFiles').innerHTML =
        output.join('');
    debug('totalFileLength:' + totalFileLength);
}

```

当用户单击 `Upload` 按钮时，会调用 `startUpload` 函数，接着调用 `uploadNext` 函数。`uploadNext` 上传所选文件集合中的下一个文件。它先创建一个 `XMLHttpRequest` 对象和一个 `FormData` 对象，下一个要上传的文件就是添加到 `FormData` 对象中的。

```

var xhr = new XMLHttpRequest();
var fd = new FormData();
var file = document.getElementById('files').
    files[filesUploaded];
fd.append("fileToUpload", file);

```

随后，`uploadNext` 函数将 `XMLHttpRequest` 对象的 `progress` 事件添加到 `onUploadProgress` 中，并分别将 `load` 事件和 `error` 事件添加到 `onUploadComplete` 和 `onUploadFailed` 上。

```

xhr.upload.addEventListener(
    "progress", onUploadProgress, false);
xhr.addEventListener("load", onUploadComplete, false);
xhr.addEventListener("error", onUploadFailed, false);

```

接下来，它打开一个服务器连接，并发送 `FormData`:

```
xhr.open("POST", "multipleUploads");
debug('uploading ' + file.name);
xhr.send(fd);
```

在上传进程中，会重复调用 `onUploadProgress` 函数，让它有机会更新进度条。更新包括计算已经上传的总字节比例，所选文件的字节数，以及放大 `progressBar` `div` 元素中的 `div` 元素，以不断地显示上传完成的百分比。

```
function onUploadProgress(e) {
    if (e.lengthComputable) {
        var percentComplete = parseInt(
            (e.loaded + totalUploaded) * 100
            / totalFileLength);
        var bar = document.getElementById('bar');
        bar.style.width = percentComplete + '%';
        bar.innerHTML = percentComplete + ' % complete';
    } else {
        debug('unable to compute');
    }
}
```

上传完成之时，会调用 `onUploadComplete` 函数。这个事件处理器会添加 `totalUploaded`，即已经完成上传的文件大小，并递增 `filesUploaded` 的数量。随后，它会查看是否已经上传了所选择的全部文件。假如是，会显示一条消息，告诉用户已经成功地完成上传。如果不是，则会再次调用 `uploadNext`。为了便于查看，下面重新把 `onUploadComplete` 函数内容列出:

```
function onUploadComplete(e) {
    totalUploaded += document.getElementById('files').
        files[filesUploaded].size;
    filesUploaded++;
    debug('complete ' + filesUploaded + " of " + fileCount);
    debug('totalUploaded: ' + totalUploaded);
    if (filesUploaded < fileCount) {
        uploadNext();
    } else {
        alert('Finished uploading file(s)');
    }
}
```

你可以利用下面这个 URL 测试应用程序:

<http://localhost:8080/app11b/html5.jsp>

选择一组文件，单击 `Upload` 按钮，你将会看到一个进度条，以及关于上传文件的信息，如图 11-3 所示。

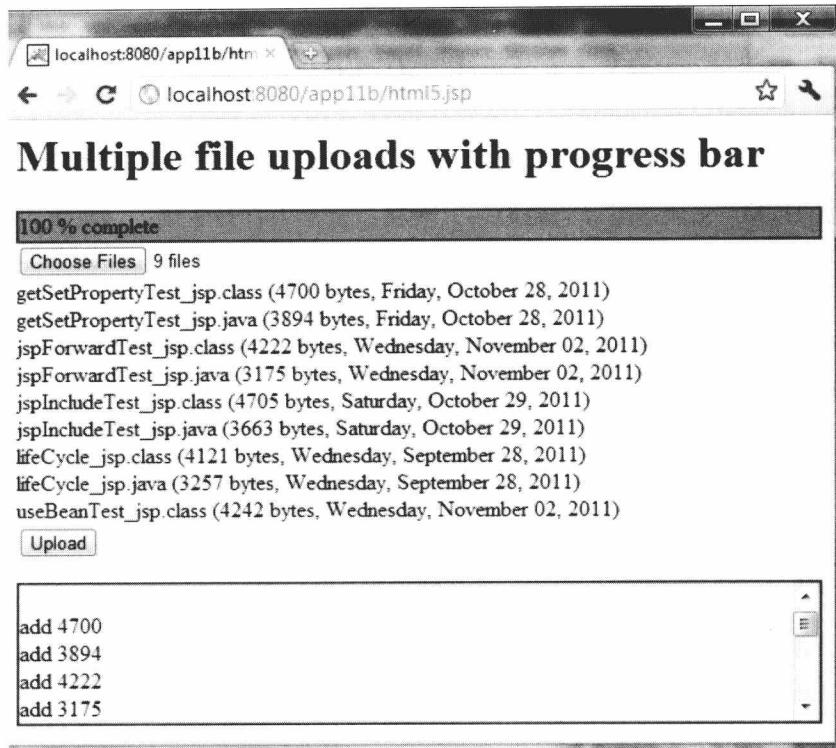


图 11-3 文件上传的进度条

## 11.6 小结

本章学习了 Servlet 3 中文件上传功能，尤其重要的是 `MultipartConfig` 注解类型（所有多部分的 Servlet 都必须用它进行标注）和 `Part` 接口。`Part` 表示一个 HTML 输入域，如果 `Part` 对应于某一个文件 `input` 元素，那么 `Part` 将会暴露被上传文件的各种属性，例如文件名和文件大小。`Part` 也可以帮助你轻松地将文件写入磁盘。

虽然 Servlet 3.0 中新增了一项很棒的特性，但它依然没有解决浏览器中缺乏有效用户界面的问题。为此，我们可以使用 HTML 5。本章介绍了如何利用 HTML 5 中的 File API 来显示和更新进度条。

# 第 12 章 文件下载

像图片或 HTML 文件这类静态资源，只要在浏览器中打开正确的网址就可以下载。只要资源放在应用程序目录或其下的子目录中，但不在 WEB-INF 下，Servlet/JSP 容器就会将资源发送到浏览器。但有的时候，静态资源被保存在应用程序目录之外，或是保存在数据库中，或者有时候你需要控制让某些人能够看到这个资源，同时又要防止其他网站跨站引用它。每当遇到这类情况时，就必须通过编程来发送资源。

简而言之，通过编程实现的文件下载可以让你有选择地将某一个文件发送到浏览器。本章将介绍通过编程发送资源给浏览器时需要做些什么，并提供了两个例子。

## 12.1 文件下载概述

为了将资源比如文件发送到浏览器，需要在 Servlet 中完成以下工作。一般不用 JSP 页面，因为要发送的是二进制代码，浏览器上不会显示内容。

1. 将响应的内容类型设置为文件的内容类型。标头 Content-Type 用来规定实体主体中的数据类型，包含媒体类型和子类型标识符。关于标准内容类型的信息，请查看这个网站：<http://www.iana.org/assignments/media-types>。如果你不了解什么是内容类型，或者希望浏览器总是显示另存为（Save As）对话框时，那么就将它设置为 APPLICATION/OCTET-STREAM。这个值是不区分大小写的。

2. 添加一个名为 Content-Disposition 的 HTTP 响应标头，给它赋值 attachment; filename=*filename*，这里的 *fileName* 是指在文件下载（File Download）对话框中显示出来的默认文件名。它通常与文件名相同，但是也可以不同。

例如，以下就是将一个文件发送到浏览器的代码范例。

```
FileInputStream fis = new FileInputStream(file);
BufferedInputStream bis = new BufferedInputStream(fis);
byte[] bytes = new byte[bis.available()];
response.setContentType(contentType);
OutputStream os = response.getOutputStream();
bis.read(bytes);
os.write(bytes);
```

如果想要把一个文件中的部分内容通过编程的方式发送到浏览器，那么首先要将该文件当成是一个 FileInputStream，并将内容添加到一个字符数组中。然后，获取

HttpServletResponse 的 OutputStream，并在调用它的 write 方法时传递字节数组给它。

**警告** 一定要确保你没有在无意中发送超出实际文件内容以外的任何字符。这有可能在你毫不知情的情况下发生。例如，如果需要在 JSP 页面中使用 page 指令，可以这么写：

```
<%@ page import="java.io.FileInputStream"%>
<jsp:useBean id="DBBeanId" scope="page" class="MyBean" />
```

在你毫不察觉的情况下，page 指令后面的回车换行符就会被发送给浏览器。为了防止发送多余的字符，需要像下面这样编写这个指令：

```
<%@ page import="java.io.FileInputStream"
%><jsp:useBean id="DBBeanId" scope="page" class="MyBean" />
```

这些代码看起来不太常见，但是很有帮助。

## 12.2 范例 1：隐藏资源

app12a 应用程序示范了如何将一个文件发送到浏览器。在这个应用程序中，我们用一个 FileDownloadServlet 将 secret.pdf 文件发送到浏览器。但是，只有授权用户才能浏览。如果用户没有登录，应用程序就会跳转到 Login 页面。在这里，用户可以在表单中输入用户名和密码，这些内容都将被提交给另一个 Servlet：LoginServlet。

app12a 应用程序的目录结构如图 12-1 所示。

secret.pdf 文件放在 WEB-INF/data 目录下，不允许进行直接访问。

代码清单 12-1 中的 FileDownloadServlet 类展示了一个负责发送 secret.pdf 文件的 Servlet。只有当用户的 HttpSession 中包含有 loggedIn 属性时，表示用户已经成功登录，此时才允许他访问。



图 12-1 应用程序目录

### 代码清单 12-1 FileDownloadServlet 类

```
package filedownload;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;
import javax.servlet.RequestDispatcher;
```

```

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

@WebServlet(urlPatterns = { "/download" })
public class FileDownloadServlet extends HttpServlet {

    private static final long serialVersionUID = 7583L;

    public void doGet(HttpServletRequest request,
                       HttpServletResponse response) throws ServletException,
                                              IOException {
        HttpSession session = request.getSession();
        if (session == null ||
            session.getAttribute("loggedIn") == null) {
            RequestDispatcher dispatcher =
                request.getRequestDispatcher("/login.jsp");
            dispatcher.forward(request, response);
            // must return after dispatcher.forward(). Otherwise,
            // the code below will be executed
            return;
        }
        String dataDirectory = request.
            getServletContext().getRealPath("/WEB-INF/data");
        File file = new File(dataDirectory, "secret.pdf");
        if (file.exists()) {
            response.setContentType("application/pdf");
            response.addHeader("Content-Disposition",
                               "attachment; filename=secret.pdf");
            byte[] buffer = new byte[1024];
            FileInputStream fis = null;
            BufferedInputStream bis = null;
            // if you're using Java 7, use try-with-resources
            try {
                fis = new FileInputStream(file);
                bis = new BufferedInputStream(fis);
                OutputStream os = response.getOutputStream();
                int i = bis.read(buffer);
                while (i != -1) {
                    os.write(buffer, 0, i);
                    i = bis.read(buffer);
                }
            } catch (IOException ex) {
                System.out.println(ex.toString());
            } finally {
                if (bis != null) {
                    bis.close();
                }
                if (fis != null) {

```

```
        fis.close();
    }
}
}
}
```

它只实现了 `doGet` 方法，因为不允许使用 HTTP 的 `post` 方法。`doGet` 方法会查看用户 `session` 中是否有 `loggedIn` 属性。如果没有，则会将用户带到登录页面。

```
HttpSession session = request.getSession();
if (session == null ||
    session.getAttribute("loggedIn") == null) {
    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/login.jsp");
    dispatcher.forward(request, response);
    // must return after dispatcher.forward(). Otherwise,
    // the code below will be executed
    return;
}
```

注意，在 `RequestDispatcher` 中调用 `forward` 会将控制权转到不同的资源上。但是，它不会中止当前在调用对象的代码执行。因此，跳转之后必须返回。

如果用户已经成功登录，`doGet` 方法就会打开所要的资源，并将它引到 `ServletResponse` 的 `OutputStream`。

```
response.setContentType("application/pdf");
response.addHeader("Content-Disposition",
    "attachment; filename=secret.pdf");
byte[] buffer = new byte[1024];
FileInputStream fis = null;
BufferedInputStream bis = null;
// if you're using Java 7, use try-with-resources
try {
    fis = new FileInputStream(file);
    bis = new BufferedInputStream(fis);
    OutputStream os = response.getOutputStream();
    int i = bis.read(buffer);
    while (i != -1) {
        os.write(buffer, 0, i);
        i = bis.read(buffer);
    }
} catch (IOException ex) {
    System.out.println (ex.toString());
} finally {
    if (bis != null) {
        bis.close();
    }
    if (fis != null) {
```

```
        fis.close();
    }
}
}
```

如果你使用的是 Java 7，那么它的新特性 `try-with-resources` 将是一种更加安全的资源处理方式。

未授权的用户会被转到如代码清单 12-2 所示的 `login.jsp` 页面。这个页面中包含了一个 HTML 表单，其中有两个输入域：`userName` 和 `password`。

代码清单 12-2 `login.jsp` 页面

---

```
<html>
<head>
<title>Login</title>
</head>
<body>
<form action="login" method="post">
    <table>
        <tr>
            <td>User Name:</td>
            <td><input name="userName"/></td>
        </tr>
        <tr>
            <td>Password:</td>
            <td><input type="password" name="password"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Login"/>
            </td>
        </tr>
    </table>
</form>
</body>
</html>
```

---

提交表单时会调用 `LoginServlet`，它的类如代码清单 12-3 所示。注意，对于这个应用程序而言，用户名 / 密码必须为 `ken/secret`。

代码清单 12-3 `LoginServlet` 类

---

```
package filedownload;
import java.io.IOException;
import javax.servlet.DispatcherType;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```

import javax.servlet.http.HttpSession;

@WebServlet(urlPatterns = { "/login" })
public class LoginServlet extends HttpServlet {

    private static final long serialVersionUID = -920L;

    public void doPost(HttpServletRequest request,
                       HttpServletResponse response) throws ServletException,
                           IOException {
        String userName = request.getParameter("userName");
        String password = request.getParameter("password");
        if (userName != null && userName.equals("ken")
            && password != null && password.equals("secret")) {
            HttpSession session = request.getSession(true);
            session.setAttribute("loggedIn", Boolean.TRUE);
            response.sendRedirect("download");
            // must call return or else the code after this if
            // block, if any, will be executed
            return;
        } else {
            RequestDispatcher dispatcher =
                request.getRequestDispatcher("/login.jsp");
            dispatcher.forward(request, response);
        }
    }
}

```

用户成功登录之后，会设置一个 `loggedIn` 会话属性，并将用户转到 `FileDownload Servlet`。

```

String userName = request.getParameter("userName");
String password = request.getParameter("password");
if (userName != null && userName.equals("ken")
    && password != null && password.equals("secret")) {
    HttpSession session = request.getSession(true);
    session.setAttribute("loggedIn", Boolean.TRUE);
    response.sendRedirect("download");
    // must call return or else the code after this if
    // block, if any, will be executed
    return;
}

```

在 `HttpServletResponse.sendRedirect` 之后，还是必须返回，以防止执行后面的代码行。登录失败时，则会将用户转到 `login.jsp` 页面：

```

} else {
    RequestDispatcher dispatcher =
        request.getRequestDispatcher("/login.jsp");
    dispatcher.forward(request, response);
}

```

利用下面这个 URL 调用 `FileDownloadServlet`, 可以对 `app12a` 应用程序进行测试:

`http://localhost:8080/app12a/download`

### 12.3 范例 2: 防止跨站引用

竞争对手很可能试图通过跨站引用来自“窃取”你的网络资产, 例如将你的贵重物品显示在他们的网站上, 好像那些东西就是他们的一样。如果通过编程的方式, 仅当 `referer` 标头中包含你的域名时才发送资源, 那么就可以防止上述情况的发生。当然, 那种意志坚定的窃贼还是有可能下载到你的资产, 但是那就需要费一番功夫了。

`app12b` 应用程序中使用了一个 Servlet, 当且仅当 `referer` 标头不为空时, 才将图片发送到浏览器。这样可以防止直接在浏览器中输入其网址就能下载到图片。`ImageServlet` Servlet 如代码清单 12-4 所示。

代码清单 12-4 `ImageServlet` 类

---

```
package filedownload;
import java.io.BufferedReader;
import java.io.BufferedInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.OutputStream;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = { "/getImage" })
public class ImageServlet extends HttpServlet {

    private static final long serialVersionUID = -99L;

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response) throws ServletException,
                                              IOException {
        String referrer = request.getHeader("referer");
        if (referrer != null) {
            String imgId = request.getParameter("id");
            String imgDirectory = request.getServletContext().
                getRealPath("/WEB-INF/image");
            File file = new File(imgDirectory,
                                imgId + ".jpg");
            if (file.exists()) {
                response.setContentType("image/jpg");
                byte[] buffer = new byte[1024];
                FileInputStream fis = null;
```

```
    BufferedInputStream bis = null;
    // if you're using Java 7, use try-with-resources
    try {
        fis = new FileInputStream(file);
        bis = new BufferedInputStream(fis);
        OutputStream os = response.getOutputStream();
        int i = bis.read(buffer);
        while (i != -1) {
            os.write(buffer, 0, i);
            i = bis.read(buffer);
        }
    } catch (IOException ex) {
        System.out.println(ex.toString());
    } finally {
        if (bis != null) {
            bis.close();
        }
        if (fis != null) {
            fis.close();
        }
    }
}
}
}
```

**ImageServlet** 类原则上与 **FileDownloadServlet** 一样。但是，**doGet** 方法开头处的 if 语句将确保只在 **referer** 标头不为空的情况下才会发送图片。

我们可以利用代码清单 12-5 中的 **images.html** 文件来测试这个应用程序。

代码清单 12-5 images.html 文件

---

```
<html>
<head>
    <title>Photo Gallery</title>
</head>
<body>










</body>
</html>
```

---

要想看到 **ImageServlet** 的效果, 请在浏览器中打开以下网址:

<http://localhost:8080/app12b/images.html>

图 12-2 中展示了由 **ImageServlet** 发送的图片。

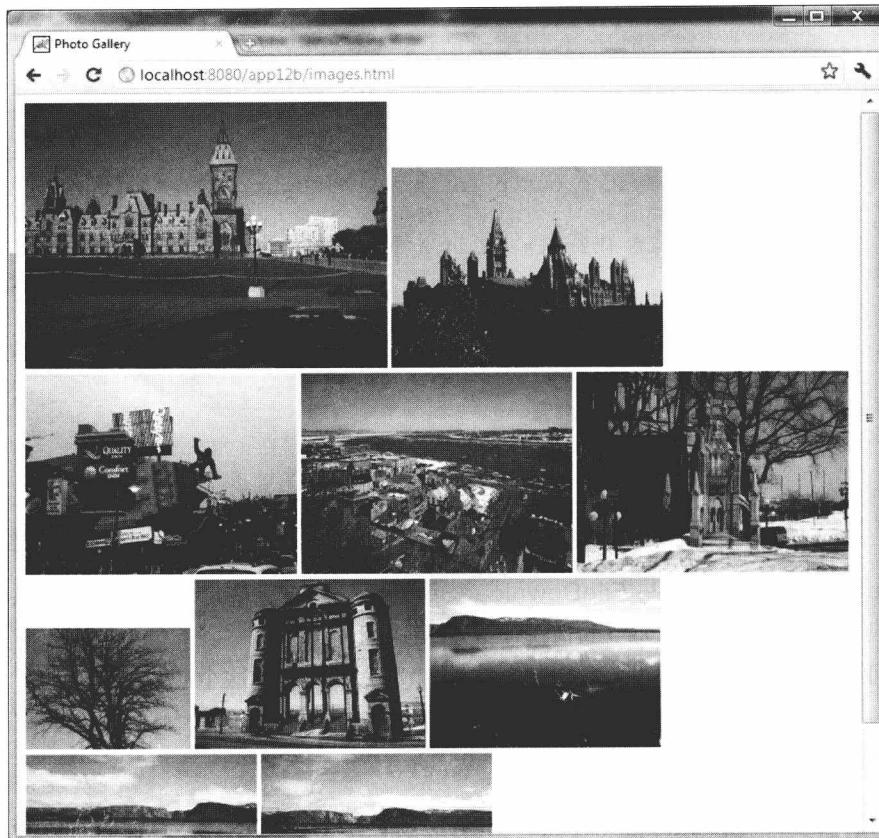


图 12-2 ImageServlet 效果示例

## 12.4 小结

本章学习了如何通过编程控制 Servlet 应用程序中的文件下载。我们还学习了如何选择文件, 并把它发送到浏览器。

# 第 13 章 请求和响应的装饰

Servlet API 中有 4 个包装类，可以用来改变 Servlet 请求和 Servlet 响应的行为。这些包装类允许将任何方法都“包装”在 `ServletRequest` 和 `ServletResponse`，或其 HTTP 对等体中（分别为 `HttpServletRequest` 和 `HttpServletResponse`）。这些包装类遵循 Decorator 或 Wrapper 模式，要使用它们，需要先了解什么是模式。

本章首先将解释什么是 Decorator 模式，并举例说明如何通过包装来改变 `HttpServletRequest` 对象的行为。也可以利用这种方法来包装 `HttpServletResponse` 对象。

## 13.1 Decorator 模式

即使没有某一个对象的类的源代码，甚至即便这个类是声明为 `final` 的，Decorator 模式和 Wrapper 模式都允许装饰或包装（直白地说，就是修改）这个对象（的行为）。

Decorator 模式适用于无法使用继承的情况（比如，所指对象的类为 `final`），或者你不想亲自创建对象，而是想从另一个子系统中获取。例如，Servlet 容器创建了一个 `ServletRequest` 和一个 `ServletResponse`，并将它们传给 Servlet 的 `service` 方法。改变 `ServletRequest` 和 `ServletResponse` 行为的唯一方法是将它们包在其他对象中。唯一必须满足的条件是，被装饰对象的类要实现一个接口，并且要包装的方法必须从这个接口处继承。

请看图 13-1 中的 UML 类图。

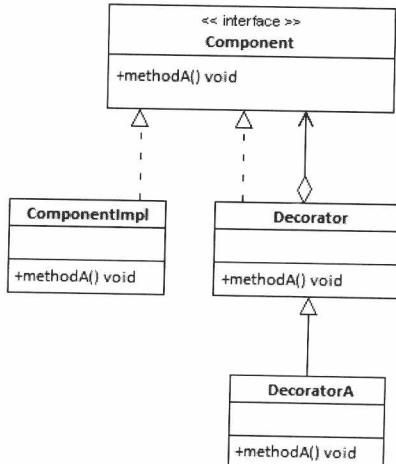


图 13-1 Decorator 模式

图 13-1 中的类图中展示了一个 Component 接口，它带有一个名为 ComponentImpl 的实现类。Component 接口定义了一个 methodA 方法。为了装饰 ComponentImpl 的实例，要创建一个 Decorator 类，它实现 Component 接口，并继承 Decorator，在子类中编写新行为的程序。在图 13.1 中，DecoratorA 就是 Decorator 的一个子类。

每一个 Decorator 实例都需要包含一个 Component 实例。Decorator 类的代码如下所示。(注意，构造器中带有一个 Component 实例，这表示只能通过传递一个 Component 实例来创建 Decorator。)

```
public class Decorator implements Component {
    private Component decorated;

    // constructor takes a Component implementation
    public Decorator(Component component) {
        this.decorated = component;
    }

    // undecorated method
    @Override
    public void methodA(args) {
        decorated.methodA(args);
    }

    // decorated method
    @Override
    public void methodB(args) {
        decorated.methodB(args)
    }
}
```

在 Decorator 类中，被装饰的方法是指其行为要在子类中进行修改的那个方法。未被装饰的方法是指不会在子类中被覆盖的方法。无论是否装饰，所有方法都要在 Component 中调用其对等的方法。Decorator 类不仅仅是一个便利类，它还为其中的每一个方法都提供默认的实现。行为的改变则由一个子类提供。

必须记住的一点是，Decorator 类和被装饰对象的类必须实现同一个接口。这样，就可以将被装饰的对象包装在 decorator 中，并将 decorator 作为 Component 的实例进行传递。我们可以将任何 Component 接口的实现传给 decorator。事实上，还可以将你的 decorator 传递给另一个 decorator，对某个对象进行双重的装饰。

## 13.2 Servlet Wrapper 类

Servlet API 中提供了 4 个类，它们很少用到，但是功能非常强大，分别是：ServletRequestWrapper、ServletResponseWrapper，以及 HttpServletRequestWrapper 和 Http-

ServletResponseWrapper。

ServletRequestWrapper（以及另外3个Wrapper类）使用起来非常方便，由于它为调用被包装ServletRequest中的对等方法的每一个方法都提供了默认实现。通过继承ServletRequestWrapper，只需要覆盖想要修改的方法即可。如果没有ServletRequestWrapper，则只好直接实现ServletRequest，并为接口中的每一个方法都提供实现。

图13-2展示了Decorator模式中的ServletRequestWrapper类。每当调用Servlet的服务方法时，Servlet容器都会创建一个ServletRequest实例：ContainerImpl。你可以继承ServletRequestWrapper来装饰ServletRequest。

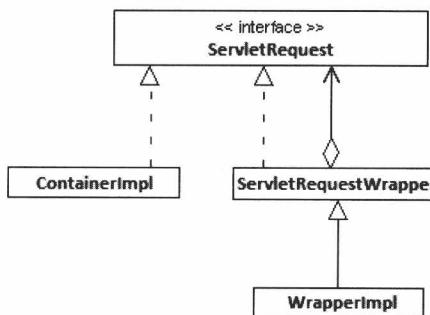


图13-2 装饰ServletRequest

### 13.3 范例：AutoCorrect过滤器

在Web应用程序中，用户经常会在输入值时，在其前面或者后面添加一些空格，甚至在词与词之间也会有多余的空格。你又不想到应用程序的逐个Servlet中进行检查并删除多余的空格。那么本节介绍的AutoCorrect过滤器的特性就可以帮你完成这些工作。这个过滤器中包含一个HttpServletRequestWrapper的子类，命名为AutoCorrectHttpServletRequestWrapper，并覆盖返回一个或多个参数值的下列方法：getParameter、getParameterValues和getParameterMap。过滤器类如代码清单13-1所示。

代码清单13-1 AutoCorrectFilter

---

```

package filter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.Collection;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import javax.servlet.Filter;
import javax.servlet.FilterChain;
  
```

```
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletRequestWrapper;

@WebFilter(filterName = "AutoCorrectFilter",
           urlPatterns = { "/" })
public class AutoCorrectFilter implements Filter {

    @Override
    public void init(FilterConfig filterConfig)
        throws ServletException {
    }

    @Override
    public void destroy() {
    }

    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response, FilterChain filterChain)
        throws IOException, ServletException {
        HttpServletRequest httpServletRequest =
            (HttpServletRequest) request;
        AutoCorrectHttpServletRequestWrapper wrapper = new
            AutoCorrectHttpServletRequestWrapper(
                httpServletRequest);
        filterChain.doFilter(wrapper, response);
    }

    class AutoCorrectHttpServletRequestWrapper extends
        HttpServletRequestWrapper {
        private HttpServletRequest httpServletRequest;

        public AutoCorrectHttpServletRequestWrapper(
            HttpServletRequest httpServletRequest) {
            super(httpServletRequest);
            this.httpServletRequest = httpServletRequest;
        }

        @Override
        public String getParameter(String name) {
            return autoCorrect(
                httpServletRequest.getParameter(name));
        }

        @Override
        public String[] getParameterValues(String name) {
```

```
    return autoCorrect(httpServletRequest
                      .getParameterValues(name));
}

@Override
public Map<String, String[]> getParameterMap() {
    final Map<String, String[]> parameterMap =
        httpServletRequest.getParameterMap();

    Map<String, String[]> newMap = new Map<String,
        String[]>() {

        @Override
        public int size() {
            return parameterMap.size();
        }

        @Override
        public boolean isEmpty() {
            return parameterMap.isEmpty();
        }

        @Override
        public boolean containsKey(Object key) {
            return parameterMap.containsKey(key);
        }

        @Override
        public boolean containsValue(Object value) {
            return parameterMap.containsValue(value);
        }

        @Override
        public String[] get(Object key) {
            return autoCorrect(parameterMap.get(key));
        }

        @Override
        public void clear() {
            // this will throw an IllegalStateException,
            // but let the user get the original
            // exception
            parameterMap.clear();
        }

        @Override
        public Set<String> keySet() {
            return parameterMap.keySet();
        }

        @Override
```

```

        public Collection<String[]> values() {
            return autoCorrect(parameterMap.values());
        }

        @Override
        public Set<Map.Entry<String,
                String[]>> entrySet() {
            return autoCorrect(parameterMap.entrySet());
        }

        @Override
        public String[] put(String key, String[] value) {
            // this will throw an IllegalStateException,
            // but let the user get the original
            // exception
            return parameterMap.put(key, value);
        }

        @Override
        public void putAll(
            Map<?, ? extends String, ? extends
                String[]> map) {
            // this will throw an IllegalStateException,
            // but let
            // the user get the original exception
            parameterMap.putAll(map);
        }

        @Override
        public String[] remove(Object key) {
            // this will throw an IllegalStateException,
            // but let
            // the user get the original exception
            return parameterMap.remove(key);
        }
    };
    return newMap;
}

private String autoCorrect(String value) {
    if (value == null) {
        return null;
    }
    value = value.trim();
    int length = value.length();
    StringBuilder temp = new StringBuilder();
    boolean lastCharWasSpace = false;
    for (int i = 0; i < length; i++) {
        char c = value.charAt(i);
        if (c == ' ') {
            if (!lastCharWasSpace) {

```

```

        temp.append(c);
    }
    lastCharWasSpace = true;
} else {
    temp.append(c);
    lastCharWasSpace = false;
}
}
return temp.toString();
}

private String[] autoCorrect(String[] values) {
    if (values != null) {
        int length = values.length;
        for (int i = 0; i < length; i++) {
            values[i] = autoCorrect(values[i]);
        }
        return values;
    }
    return null;
}

private Collection<String[]> autoCorrect(
    Collection<String[]> valueCollection) {
    Collection<String[]> newCollection =
        new ArrayList<String[]>();
    for (String[] values : valueCollection) {
        newCollection.add(autoCorrect(values));
    }
    return newCollection;
}

private Set<Map.Entry<String, String[]>> autoCorrect(
    Set<Map.Entry<String, String[]>> entrySet) {
    Set<Map.Entry<String, String[]>> newSet = new
        HashSet<Map.Entry<String, String[]>>();
    for (final Map.Entry<String, String[]> entry
        : entrySet) {
        Map.Entry<String, String[]> newEntry = new
            Map.Entry<String, String[]>() {
            @Override
            public String getKey() {
                return entry.getKey();
            }

            @Override
            public String[] getValue() {
                return autoCorrect(entry.getValue());
            }

            @Override
            public String[] setValue(String[] value) {

```

```

        return entry.setValue(value);
    }
}
newSet.add(newEntry);
}
return newSet;
}
}

```

---

过滤器的 `doFilter` 方法非常简单。它为 `ServletRequest` 创建了一个 `decorator`, 并将 `decorator` 传给 `doFilter` 方法:

```

HttpServletRequest httpServletRequest =
    (HttpServletRequest) request;
AutoCorrectHttpServletRequestWrapper wrapper = new
    AutoCorrectHttpServletRequestWrapper(
        httpServletRequest);
filterChain.doFilter(wrapper, response);

```

在过滤器后面调用的每一个 `Servlet` 都会获得一个包装在 `AutoCorrectHttpServletRequestWrapper` 中的 `HttpServletRequest`。包装类很长, 但是很容易理解。一般来说, 它会传递所有的调用给这个 `autoCorrect` 方法的, 以获取去掉空格的参数值。

```

private String autoCorrect(String value) {
    if (value == null) {
        return null;
    }
    value = value.trim();
    int length = value.length();
    StringBuilder temp = new StringBuilder();
    boolean lastCharWasSpace = false;
    for (int i = 0; i < length; i++) {
        char c = value.charAt(i);
        if (c == ' ') {
            if (!lastCharWasSpace) {
                temp.append(c);
            }
            lastCharWasSpace = true;
        } else {
            temp.append(c);
            lastCharWasSpace = false;
        }
    }
    return temp.toString();
}

```

我们可以分别利用代码清单 13-2 和代码清单 13-3 中所示的 `test1.jsp` 和 `test2.jsp` 页面来测试过滤器。

## 代码清单 13-2 test1.jsp 页面

---

```
<!DOCTYPE HTML>
<html>
<head>
<title>User Form</title>
</head>
<body>
<form action="test2.jsp" method="post">
    <table>
        <tr>
            <td>Name:</td>
            <td><input name="name"/></td>
        </tr>
        <tr>
            <td>Address:</td>
            <td><input name="address"/></td>
        </tr>
        <tr>
            <td colspan="2">
                <input type="submit" value="Login"/>
            </td>
        </tr>
    </table>
</form>
</body>
</html>
```

---

## 代码清单 13-3 test2.jsp 页面

---

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions"
    prefix="fn"%>
<!DOCTYPE HTML>
<html>
<head>
<title>Form Values</title>
</head>
<body>
<table>
    <tr>
        <td>Name:</td>
        <td>
            ${param.name}
            (length:${fn:length(param.name) })
        </td>
    </tr>
    <tr>
        <td>Address:</td>
        <td>
            ${param.address}
            (length:${fn:length(param.address) })
        </td>
    </tr>
</table>
</body>
</html>
```

---

```
</td>
</tr>
</table>
</body>
</html>
```

---

利用下面这个 URL 调用 test1.jsp 页面：

`http://localhost:8080/app13a/test1.jsp`

输入参数值时，故意在其前面或者后面添加一些空格，或者在词与词之间增加多余的空间，然后单击 Submit（提交）按钮。你在屏幕中将会看到，输入的值已经被自动纠正了。

## 13.4 小结

Servlet API 中提供了 4 个包装类 (`ServletRequestWrapper`、`ServletResponseWrapper`、`HttpServletRequestWrapper` 以及 `HttpServletResponseWrapper`)，可以继承它们来装饰 Servlet 请求和 Servlet 响应。之后，可以利用过滤器或者监听器创建一个包装类，并将它传给 Servlet 的 `service` 方法，如本章 `AutoCorrectFilter` 范例中所示。

# 第 14 章 异步处理

Servlet 3 引入了一项新的特性，它可以让 Servlet 异步处理请求。本章就来介绍这项新特性，并举例说明它的用法。

## 14.1 概述

计算机的内存是有限的。Servlet/JSP 容器的设计者很清楚这一点，因此他们提供了一些可以进行配置的设置，以确保容器能够在宿主机器中正常运行。例如，在 Tomcat 7 中，处理进来请求的最多线程数量为 200。如果是多处理器的服务器，则可以放心地增加线程数量，不过建议你还是尽量使用这个默认值。

Servlet 或 Filter 一直占用着请求处理线程，直到它完成任务。如果完成任务花费了很长时间，并发用户的数量就会超过线程数量，容器将会遇到超出线程的风险。如果发生这种情况，Tomcat 就会将超出的请求堆放在一个内部的服务器 Socket 中（其他容器的处理方式可能会有所不同）。如果继续进来更多的请求，它们将会遭到拒绝，直到有资源可以处理请求为止。

异步处理特性可以帮助你节省容器线程。这项特性适用于长时间运行的操作。它的工作是等待任务完成，并释放请求处理线程，以便另一个请求能够使用该线程。注意，异步支持只适用于长时间运行的任务，并且你想让用户知道任务的执行结果。如果只有长时间运行的任务，但用户不需要知道处理的结果，那么则只要提供一个 `Runnable` 给 `Executor`，并立即返回。例如，如果需要产生一份报表（需要花点时间），并在报表准备就绪之后通过电子邮件将报表发送出去，那么就不适合使用异步处理特性了。相反，如果需要产生一份报表，并且报表完成之后要展示给用户看，那么就可以使用异步处理。

## 14.2 编写异步的 Servlet 和 Filter

`WebServlet` 和 `WebFilter` 注解类型可以包含新的 `asyncSupport` 属性。为了编写能够支持异步处理的 Servlet 和 Filter，`asyncSupported` 属性必须设为 `true`：

```
@WebServlet(asyncSupported=true ...)  
@WebFilter(asyncSupported=true ...)
```

另一种方法是，利用 `servlet` 或 `filter` 元素中的 `async-supported` 元素在部署描述符中进行设定。例如，下面的 Servlet 就是配置成可以支持异步处理的：

```
<servlet>
    <servlet-name>AsyncServlet</servlet-name>
    <servlet-class>servlet.MyAsyncServlet</servlet-class>
    <async-supported>true</async-supported>
</servlet>
```

支持异步处理的 Servlet 或者 Filter 可以通过在 `ServletRequest` 中调用 `startAsync` 方法来启动新的线程。`startAsync` 有两个重载方法：

```
AsyncContext startAsync() throws java.lang.IllegalStateException
AsyncContext startAsync(ServletRequest servletRequest,
    ServletResponse servletResponse) throws
    java.lang.IllegalStateException
```

这两个重载方法都返回一个 `AsyncContext` 实例，该实例中提供了各种方法，还包含一个 `ServletRequest` 和一个 `ServletResponse`。第一个重载方法很简单，也很容易使用。生成的 `AsyncContext` 中也将包含原始的 `ServletRequest` 和 `ServletResponse`。第二个重载方法允许将原始的 `ServletRequest` 和 `ServletResponse` 进行包装，并将它们传给 `AsyncContext`。注意，只能将原始的 `ServletRequest` 和 `ServletResponse` 或其包装器（Wrapper）传给第二个 `startAsync` 重载方法。第 13 章已经讨论过 `ServletRequest` 和 `ServletResponse` 包装。

注意，重复调用 `startAsync` 方法将会返回相同的 `AsyncContext`。如果在不支持异步处理的 Servlet 或 Filter 中调用 `startAsync` 方法，将会抛出一个 `java.lang.IllegalStateException` 异常。还要注意，`AsyncContext` 的 `start` 方法不会造成阻塞，因此，即使它派发的线程还没有启动，也会继续执行下一行代码。

### 14.3 编写异步的 Servlet

编写异步的 Servlet 或 Filter 相对比较简单。如果你有一个任务需要相对比较长的时间才能完成，最好创建一个异步的 Servlet 或者 Filter。在异步的 Servlet 或者 Filter 类中需要完成以下工作：

1. 在 `ServletRequest` 中调用 `startAsync` 方法。`startAsync` 会返回一个 `AsyncContext`。
2. 在 `AsyncContext` 中调用 `setTimeout()` 方法，设置一个容器必须等待指定任务完成的毫秒数。这个步骤是可选的，但是如果设置这个时限，将会采用容器的默认时间。如果任务没能在规定时限内完成，将会抛出异常。
3. 调用 `asyncContext.start` 方法，传递一个执行长时间任务的 `Runnable`。

4. 任务完成时，通过 Runnable 调用 `asyncContext.complete` 方法或者 `asyncContext.dispatch` 方法。

下面是异步 Servlet 的 `doGet` 或者 `doPost` 方法的主要内容：

```
final AsyncContext asyncContext = servletRequest.startAsync();
asyncContext.setTimeout( ... );
asyncContext.start(new Runnable() {
    @Override
    public void run() {

        // long running task
        asyncContext.complete() or asyncContext.dispatch()
    }
})
```

举个例子，代码清单 14-1 中展示了一个支持异步处理的 Servlet。

代码清单 14-1 带有 dispatch 的简单 servlet

---

```
package servlet;
import java.io.IOException;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(name = "AsyncDispatchServlet",
           urlPatterns = { "/asyncDispatch" },
           asyncSupported = true)
public class AsyncDispatchServlet extends HttpServlet {
    private static final long serialVersionUID = 222L;

    @Override
    public void doGet(final HttpServletRequest request,
                     HttpServletResponse response)
        throws ServletException, IOException {
        final AsyncContext asyncContext = request.startAsync();
        request.setAttribute("mainThread",
                             Thread.currentThread().getName());
        asyncContext.setTimeout(5000);
        asyncContext.start(new Runnable() {
            @Override
            public void run() {
                // long-running task
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                }
            }
        });
    }
}
```

```
        request.setAttribute("workerThread",
                Thread.currentThread().getName());
        asyncContext.dispatch("/threadNames.jsp");
    }
})
}
}
```

代码清单 14-1 中的 Servlet 支持异步处理，它的长时间运行任务简化为休眠 3 秒钟。为了证明长时间运行的任务是在主线程（执行 Servlet 的 `doGet` 方法）以外的其他线程中执行的，它在 `ServletRequest` 中添加了主线程的名称，以及工作线程的名称，并将它们发到一个 `test.jsp` 页面。`test.jsp` 页面（如代码清单 14-2 所示）中显示了 `mainThread` 和 `workerThread` 变量。它们应该会显示出不同的线程名称。

代码清单 14-2 `threadNames.jsp` 页面

```
<!DOCTYPE HTML>
<html>
<head>
<title>Asynchronous servlet</title>
</head>
<body>
Main thread: ${mainThread}
<br/>
Worker thread: ${workerThread}
</body>
</html>
```

注意，任务结束之后，必须在 `AsyncContext` 中调用 `dispatch` 或 `complete` 方法，以便它不用一直等达到规定的时限。

在浏览器中打开下面这个 URL，可以对这个 Servlet 做个测试：

```
http://localhost:8080/app14a/asyncDispatch
```

图 14-1 展示了主线程和工作线程的名称。你在浏览器中看到的可能会有所不同，但名称是不同的，这就证明工作线程与主线程是不同的。

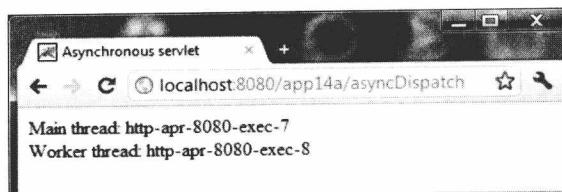


图 14-1 `AsyncDispatchServlet`

如果不想在任务完成之时分配给另一个资源，也可以在 `AsyncContext` 中调用 `complete` 方法。这个方法向 Servlet 容器表明，任务已经完成。

再举个例子，请看代码清单 14-3 中的 Servlet。这个 Servlet 每秒钟发送一次进程更新，以便用户能够追踪进程。它发送 HTML 响应和一个简单的 JavaScript 代码，用来更新一个 HTML `div` 元素。

代码清单 14-3 发送进程更新的异步 Servlet

---

```

package servlet;
import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class AsyncCompleteServlet extends HttpServlet {
    private static final long serialVersionUID = 78234L;

    @Override
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        final PrintWriter writer = response.getWriter();
        writer.println("<html><head><title>" +
                      "Async Servlet</title></head>");
        writer.println("<body><div id='progress'></div>");
        final AsyncContext asyncContext = request.startAsync();
        asyncContext.setTimeout(60000);
        asyncContext.start(new Runnable() {
            @Override
            public void run() {
                System.out.println("new thread:" +
                                   Thread.currentThread());
                for (int i = 0; i < 10; i++) {
                    writer.println("<script>");
                    writer.println("document.getElementById('" +
                                  "'progress').innerHTML = '" +
                                  (i * 10) + "% complete'");
                    writer.println("</script>");
                    writer.flush();
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                    }
                }
            }
        });
    }
}

```

```
        writer.println("<script>");
        writer.println("document.getElementById(" +
                      "'progress').innerHTML = 'DONE!'");
        writer.println("</script>");
        writer.println("</body></html>");
        asyncContext.complete();
    }
}
}
}
```

下面这个代码片段负责发送进程更新：

```
writer.println("<script>");
writer.println("document.getElementById(" +
              "'progress').innerHTML = '" +
              (i * 10) + "% complete'");
writer.println("</script>");
```

浏览器会收到这个字符串，这里的 x 是一个 10 至 100 之间的数字。

```
<script>
document.getElementById('progress').innerHTML = 'x% complete'
</script>
```

为了说明如何通过在部署描述符中声明来编写异步的 Servlet，代码清单 14-3 中的 **AsyncCompleteServlet** 类并没有用 **@WebServlet** 进行标注。部署描述符（web.xml 文件）如代码清单 14-4 所示。

代码清单 14-4 部署描述符

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
          version="3.0">
  <servlet>
    <servlet-name>AsyncComplete</servlet-name>
    <servlet-class>servlet.AsyncCompleteServlet</servlet-class>
    <async-supported>true</async-supported>
  </servlet>

  <servlet-mapping>
    <servlet-name>AsyncComplete</servlet-name>
    <url-pattern>/asyncComplete</url-pattern>
  </servlet-mapping>
</web-app>
```

在浏览器中打开以下网址，对 `AsyncCompleteServlet` 范例做个测试：

`http://localhost:8080/app14a/asyncComplete`

图 14-2 展示了最终的结果。



图 14-2 接收进程更新的 HTML 页面

## 14.4 异步监听器

除了支持 `Servlet` 和 `Filter` 执行异步操作之外，`Servlet 3.0` 还新增了一个 `AsyncListener` 接口，以便通知用户在异步处理期间发生的情况。`AsyncListener` 接口定义了以下方法，当某个事件发生时，其中某一个方法就会被调用。

`void onStartAsync(AsyncEvent event)`

在刚启动一个异步操作时调用这个方法。

`void onComplete(AsyncEvent event)`

当一个异步操作已经完成时调用这个方法。

`void onError(AsyncEvent event)`

当一个异步操作失败时调用这个方法。

`void onTimeout(AsyncEvent event)`

当一个异步操作已经超时的时候调用这个方法，即当它没能在规定时限内完成的时候。

这 4 个方法都会收到一个 `AsyncEvent` 事件，你可以分别通过调用它的 `getAsyncContext`、`getSuppliedRequest` 和 `getSuppliedResponse` 方法从中获得相关的 `AsyncContext`、`ServletRequest` 和 `ServletResponse` 实例。

举个例子。代码清单 14-5 中的 `MyAsyncListener` 类实现了 `AsyncListener` 接口，以便当异步操作中有事件发生时能够收到通知。与其他 Web 监听器不同的是，它没有用 `@WebListener` 标注 `AsyncListener` 的实现。

## 代码清单 14-5 异步监听器

---

```

package listener;
import java.io.IOException;
import javax.servlet.AsyncEvent;
import javax.servlet.AsyncListener;

// do not annotate with @WebListener
public class MyAsyncListener implements AsyncListener {

    @Override
    public void onComplete(AsyncEvent asyncEvent)
        throws IOException {
        System.out.println("onComplete");
    }

    @Override
    public void onError(AsyncEvent asyncEvent)
        throws IOException {
        System.out.println("onError");
    }

    @Override
    public void onStartAsync(AsyncEvent asyncEvent)
        throws IOException {
        System.out.println("onStartAsync");
    }

    @Override
    public void onTimeout(AsyncEvent asyncEvent)
        throws IOException {
        System.out.println("onTimeout");
    }
}

```

---

由于 `AsyncListener` 类没有用 `@WebListener` 进行标注，因此对于你有兴趣收到事件通知的每一个 `AsyncContext` 都需要手动注册一个 `AsyncListener`，做法是通过在 `AsyncContext` 中调用 `addListener` 方法来注册一个 `AsyncListener`：

```
void addListener(AsyncListener listener)
```

代码清单 14-6 中的 `AsyncListenerServlet` 类是一个 `async` Servlet，它利用代码清单 14-5 中的监听器来获取事件通知。

代码清单 14-6 使用 `AsyncListener`


---

```

package servlet;
import java.io.IOException;
import javax.servlet.AsyncContext;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;

```

```

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

import listener.MyAsyncListener;

@WebServlet(name = "AsyncListenerServlet",
            urlPatterns = { "/asyncListener" },
            asyncSupported = true)
public class AsyncListenerServlet extends HttpServlet {
    private static final long serialVersionUID = 62738L;

    @Override
    public void doGet(final HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        final AsyncContext asyncContext = request.startAsync();
        asyncContext.setTimeout(5000);

        asyncContext.addListener(new MyAsyncListener());
        asyncContext.start(new Runnable() {
            @Override
            public void run() {
                try {
                    Thread.sleep(3000);
                } catch (InterruptedException e) {
                }
                String greeting = "hi from listener";
                System.out.println("wait....");
                request.setAttribute("greeting", greeting);
                asyncContext.dispatch("/test.jsp");
            }
        });
    }
}

```

---

在浏览器中打开下面这个网址，对这个 Servlet 做下测试：

<http://localhost:8080/app14a/asyncListener>

## 14.5 小结

Servlet 3.0 中新增了一项特性，用来处理异步的操作。当 Servlet/JSP 应用程序中有一个或者多个长时间运行的操作时，这项特性就特别好用。它会将那些操作分配给一个新的线程，从而将这个请求处理线程返回到池中，准备服务下一个请求。本章学习了如何编写支持异步处理的 Servlet，以及如何编写在处理期间发生某些事件时能够收到通知的监听器。

# 第 15 章 安 全 性

在 Web 应用程序的开发和部署中，安全性是一个非常重要的方面。由于任何人都可以通过浏览器访问 Web 应用程序，以及访问万维网，因此安全性显得尤为重要。保护应用程序的安全可以通过声明或者编程两种方式来实现。以下 4 个方面是 Web 安全性的基础：验证、授权、保密性以及数据完整性。

验证（Authentication）是指证明一个 Web 实体的身份，特别是访问应用程序的用户的身份。我们通常通过要求用户输入用户名和密码来对用户进行验证。

授权（Authorization）一般是在验证成功之后进行，主要关注被验证用户的访问级别。它要努力回答这样一个问题：是否允许某个在验证的用户进入这个应用程序的某一个区域？

保密性（Confidentiality）是一个重要的主题，因为敏感数据，例如信用卡资料或者社会安全号码（后者是指美国公民的社会安全卡号，俗称工号——译者注），这些都是应该受到保护的。你也知道，在互联网上，数据在到达目的地之前，它是从一台计算机分程传递到另一台计算机的。从技术角度来看，拦截数据并不困难。因此敏感数据在互联网上传输时应该进行加密。

由于数据包很容易被拦截，因此对于具备这方面知识和工具的人来说，要想篡改数据也易如反掌。值得庆幸的是，我们还可以通过确保敏感数据经由某一个安全通道来传输，来保持数据的完整性（Data Integrity）。

本章会详细介绍安全性的这些方面，并且还会详细介绍 SSL，这是在互联网上创建安全通道时采用的协议。

## 15.1 验证和授权

验证是检验某个人是否真是他 / 她所声称的那个人的过程。在 Servlet/JSP 应用程序中，验证一般是通过要求用户输入用户名和密码来完成的。

授权主要是确定一个用户具有什么样的访问级别。它适用于包含多个访问区域的应用程序，使用户能够访问应用程序的某一个部分，但是不能访问其他部分。例如，网上商店可以分成公用区（供一般的公共浏览和查找商品），买家区（供已注册用户下单用），以及需要最高访问级别的管理区。不仅管理员用户自身也需要进行验证，并且还必须是已经被允许访问管理区的用户才行。

访问级别经常被称作角色。在部署时，Servlet/JSP 应用程序可以很方便地进行分区和配

置，使每个区都可以被特定角色的用户访问到。具体做法是在部署描述符中声明安全约束，这也称作声明式安全。另一方面，内容方面的限制一般也是通过编程的方式来实现的，具体做法是尽量将用户名和密码与数据库中保存的值进行匹配。

在大多数 Servlet/JSP 应用程序中，验证和授权都是通过编程方式来完成的，具体做法是先将用户名和密码与数据库表进行验证。一旦验证成功，就会检查另外的表格，或者检查保存用户名和密码的表格中的某一个字段。采用声明式安全可以使你避免部分编程，因为 Servlet/JSP 容器会负责验证和授权的过程。另外，可以配置 Servlet/JSP 容器，由它负责对应应用程序所用的数据库进行验证。除此之外，有了声明式验证，浏览器就可以在将用户名和密码发送给服务器之前对其进行加密。声明式安全的不足之处在于，支持数据加密的验证方法只能在默认的登录对话框中使用，它的外观无法定制。单就这一个原因就足以让人对声明式安全敬而远之了。令人遗憾的是，声明式安全中唯一允许使用定制 HTML 表单的方法，却不能对所传输的数据进行加密。

Web 应用程序的某些部分，比如管理模块，客户是看不到的，因此它的登录表单外观无关紧要。这时候仍然可以使用声明式安全。

声明式安全中值得关注的部分自然是安全性约束不用写到 Servlet 中去，只要在部署应用程序时，在部署描述符中对它们进行声明即可。因此，在确定访问应用程序或其中各个部分的用户和角色时，具有相当大的灵活性。

使用声明式安全时，首先要定义用户和角色。根据所使用的容器，可以将用户和角色信息保存在文件中，也可以保存在数据库表中。然后，针对某一个资源，或者某一个集合，或者应用程序中的某几个资源强加约束。

那么，如果不通过编程，又该如何验证用户呢？你稍后会发现，答案就在 HTTP 规范中，而不是在 Servlet 规范中。

### 15.1.1 定义用户和角色

每一种兼容的 Servlet/JSP 容器都必须提供一种定义用户和角色的方法。如果你使用的是 Tomcat，就可以通过编辑 conf 目录下的 tomcat-users.xml 文件来创建用户和角色。tomcat-users.xml 文件范例如代码清单 15-1 所示。

代码清单 15-1 tomcat-users.xml 文件

---

```
<?xml version='1.0' encoding='utf-8'?>
<tomcat-users>
    <role rolename="manager"/>
    <role rolename="member"/>
    <user username="tom" password="secret" roles="manager,member"/>
    <user username="jerry" password="secret" roles="member"/>
</tomcat-users>
```

---

`tomcat-users.xml` 文件是一个带有 `tomcat-users` 根元素的 XML 文档。其中有 `role` 和 `user` 元素。`role` 元素定义角色，`user` 元素定义用户。`role` 元素有一个 `rolename` 属性，用来指定角色的名称。`user` 元素有 `username`、`password` 和 `roles` 属性。`username` 属性指定用户名，`password` 属性指定密码，`roles` 属性则指定该用户所属的一个或多个角色。

代码清单 15-1 中的 `tomcat-users.xml` 文件声明了两个角色（`manager` 和 `member`）和两个用户（`tom` 和 `jerry`）。用户 `tom` 的角色既是 `member` 又是 `manager`，而 `jerry` 则只属于 `member` 角色。显然，`tom` 比 `jerry` 拥有访问应用程序的更多权限。

Tomcat 也支持将角色和用户与数据库表进行比对。我们可以把 Tomcat 配置为利用 JDBC 来验证用户。

### 15.1.2 强加安全性约束

前面讲过，将静态资源和 JSP 页面保存在 `WEB-INF` 或其下的某一个目录下，可以将它们隐藏起来。放在这里的资源无法直接通过输入网址而访问到，但是仍然可以通过一个 Servlet 或者 JSP 页面跳转到那里。虽然这种方法简单直接，但缺点是藏在这里的资源就永远被藏起来了。它们永远无法被直接访问到。如果只是想避免未授权的用户访问这些资源，那么可以将它们放在应用程序目录下的某一个目录中，并在部署描述符中声明一个安全性约束。

`security-constraint` 元素用于指定一个资源集合，以及可以访问这些资源的一个或多个角色。这个元素可以有两个子元素：`web-resource-collection` 和 `auth-constraint`。

`web-resource-collection` 元素用于指定一个资源集合，并且可以带有以下子元素：`web-resource-name`、`description`、`url-pattern`、`http-method` 和 `http-method-omission`。

`web-resource-collection` 元素可以带有多个 `url-pattern` 子元素，每一个子元素都表示所包含安全性约束适用的一种 URL 模式。我们可以在 `url-pattern` 元素中用一个星号 (\*) 表示一种特定的资源类型（如 `*.jsp`），或者表示某个目录下的所有资源（如 `/*` 或 `/jsp/*`）。但是这两种不能合二为一，比如，无法表示某一个特定目录下的某一种特定的类型。因此，如果用 `/jsp/*.jsp` 表示 `jsp` 目录下所有 JSP 页面，那么这个 URL 模式将是无效的，而是要用 `/jsp/*` 来表示，但是这样又限制了 `jsp` 目录下那些非 JSP 的页面。

`http-method` 元素中定义了一个 HTTP 方法，包含的安全性约束即应用到该方法中。例如，`web-resource-collection` 元素带有一个 GET `http-method` 元素，表示 `web-resource-collection` 元素只应用于 HTTP 的 Get 方法。包含资源集合的安全性约束并不能保护其他的 HTTP 方法，例如 Post 方法和 Put 方法。如果没有 `http-method` 元素，表示这个安全性约束将会限制对所有 HTTP 方法的访问。同一个 `web-resource-collection` 元素中可以带有多个 `http-method` 元素。

`http-method-omission` 元素指定的是不在所包含安全性约束中的 HTTP 方法。因此，

指定 <http-method-omission>GET</http-method-omission> 限制了对除 GET 之外的所有 HTTP 方法的访问。

http-method 元素和 http-method-omission 元素不能同时出现在同一个 web-resource-collection 元素中。

部署描述符中可以有多个 security-constraint 元素。如果安全性约束元素中没有 auth-constraint 元素，那么这个资源集合将不能受到保护。此外，如果你指定了一个没有在容器中定义的角色，那么将没有人能够直接访问这个资源集合。但是，你还是可以通过一个 Servlet 或者 JSP 页面跳转到集合中的某个资源。

举个例子。请看代码清单 15-2，其 web.xml 文件中的 security-constraint 元素限制了对所有 JSP 页面的访问。由于 auth-constraint 元素中没有包含 role-name 元素，因此不能通过 URL 来访问这些资源。

**代码清单 15-2 防止访问某些目录下的资源**

---

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
    version="3.0"
    >
    <!-- restricts access to JSP pages -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>JSP pages</web-resource-name>
            <url-pattern>*.jsp</url-pattern>
        </web-resource-collection>
        <!-- must have auth-constraint, otherwise the
            specified web resources will not be restricted -->
        <auth-constraint/>
    </security-constraint>
</web-app>

```

---

现在，可以在浏览器中打开下面这个网址，对它做一个测试：

<http://localhost:8080/app15a/jsp/1.jsp>

Servlet 容器会发送一条 HTTP 403 错误，温柔地告诉你：Access to the requested resource has been denied（禁止访问）。

接下来讲解如何对用户进行验证和授权。

## 15.2 验证方法

学会如何对一个资源集合强加安全性约束之后，还应该知道如何对访问这些资源的用户进行验证。对于声明式保护的资源，可以在部署描述符中使用 `security-constraint` 元素，通过使用 HTTP 1.1 提供的解决方案来完成验证：基本访问验证（Basic Access Authentication）和摘要访问验证（Digest Access Authentication）。此外，也可以使用基于表单的访问验证。

HTTP 验证是在 RFC 2617 中定义的，在下面这个网站可以下载到规范：

<http://www.ietf.org/rfc/rfc2617.txt>

### 15.2.1 基本访问验证

基本访问验证，简称基本验证，是一种接受用户名和密码的 HTTP 验证。在基本访问验证中，如果用户访问受保护的资源，将会遭到服务器的拒绝，并返回一个 401（未授权）响应。该响应中包含一个 `WWW-Authenticate` 标头，其中至少包含一个适用于被请求资源的角色，例如：

```
HTTP/1.1 401 Authorization Required
Server: Apache-Coyote/1.1
Date: Wed, 21 Dec 2011 11:32:09 GMT
WWW-Authenticate: Basic realm="Members Only"
```

随后，浏览器屏幕上会显示一个登录对话框，供用户输入用户名和密码。当用户单击 `Login`（登录）按钮时，用户名后面会被添上一个冒号，并和密码结合在一起。这个字符串被送到服务器之前，将先用 Base64 算法进行编码。登录成功之后，服务器就会发出被请求的资源。

Base64 是一种很弱的算法，因此 Base64 消息很容易被破解。因此，要考虑使用摘要访问验证来代替。

`app15b` 应用程序范例中展示了如何使用基本访问验证。代码清单 15-3 展示了这个应用程序的部署描述符。第一个 `security-constraint` 元素是避免 JSP 页面被直接访问。第二个是限制只有 `manager` 和 `member` 角色的用户才能访问 `Servlet1` Servlet。`Servlet1` 类是一个跳转到 `1.jsp` 页面的简单 Servlet，如代码清单 15-4 所示。

代码清单 15-3 `app15b` 的部署描述符

---

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
  version="3.0">
```

```

>
<!-- restricts access to JSP pages -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>JSP pages</web-resource-name>
        <url-pattern>*.jsp</url-pattern>
    </web-resource-collection>
    <!-- must have auth-constraint, otherwise the
        specified web resources will not be restricted -->
    <auth-constraint/>
</security-constraint>

<security-constraint>
    <web-resource-collection>
        <web-resource-name>Servlet1</web-resource-name>
        <url-pattern>/servlet1</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>member</role-name>
        <role-name>manager</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Members Only</realm-name>
</login-config>
</web-app>

```

---

代码清单 15-3 的部署描述符中最重要的元素是 `login-config` 元素。它有两个子元素：`auth-method` 和 `realm-name`。使用基本访问验证时，它的值必须为 **BASIC**（全部大写）。`realm-name` 元素要有一个名称，用来显示在浏览器的登录对话框中。

#### 代码清单 15-4 Servlet1 类

```

package servlet;
import java.io.IOException;
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns = { "/servlet1" })
public class Servlet1 extends HttpServlet {

    private static final long serialVersionUID = -15560L;

    public void doGet(HttpServletRequest request,

```

```
HttpServletResponse response) throws ServletException,  
IOException {  
    RequestDispatcher dispatcher =  
        request.getRequestDispatcher("/jsp/1.jsp");  
    dispatcher.forward(request, response);  
}  
}
```

想要测试 app15b 中的基本访问验证，可试着利用下面的网址来访问它里面的受限资源：

<http://localhost:8080/app15b/servlet1>

你在屏幕上不是看到 Servlet1 的输出结果，而是将被提示输入用户名和密码，如图 15-1 所示。

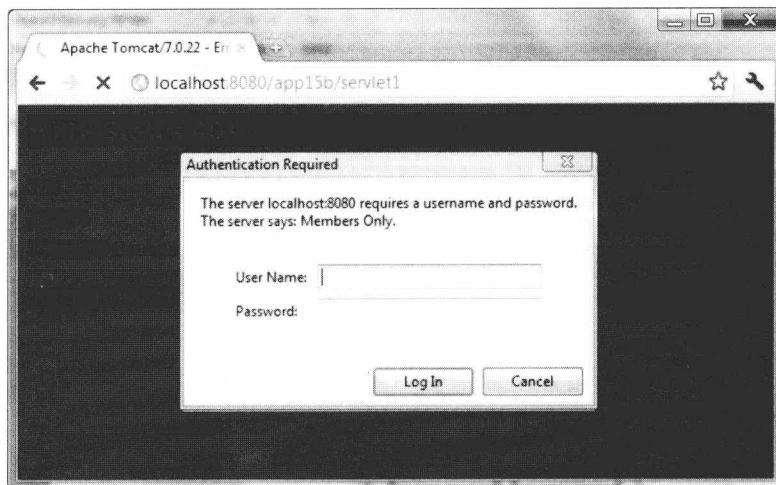


图 15-1 Basic 验证

由于映射到 Servlet1 的 auth-constraint 元素指定了 manager 和 member 两种角色，因此通过 tom 或者 jerry 都可以登录。

### 15.2.2 摘要访问验证

摘要访问验证 (Digest Access Authentication)，或简称摘要验证，它也是一种 HTTP 验证，与基本访问验证相似。摘要访问验证不是使用软弱的 Base64 加密算法，而是用 MD5 算法创建一个散列 (hash)，其中结合了用户名、realm 名称以及密码，并将这个散列 (hash) 发送到服务器。摘要访问验证旨在取代基本访问验证，因为它提供了更加安全的环境。

Servlet/JSP 容器没有强制要求支持摘要访问验证，但是大多数容器都支持。

将应用程序配置成使用摘要访问验证，做法与使用基本访问验证时相似。事实上，它们

之间唯一的区别在于 `login-config` 元素中的 `auth-method` 元素的值。在摘要访问验证中，这个元素的值必须为 **DIGEST**（全部大写）。

举个例子。`app15c` 应用程序示范了摘要访问验证的使用方法。这个应用程序的部署描述符如代码清单 15-5 所示。

代码清单 15-5 Digest 验证的部署描述符

---

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
    version="3.0"
>
    <!-- restricts access to JSP pages -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>JSP pages</web-resource-name>
            <url-pattern>*.jsp</url-pattern>
        </web-resource-collection>
        <!-- must have auth-constraint, otherwise the
            specified web resources will not be restricted -->
        <auth-constraint/>
    </security-constraint>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Servlet1</web-resource-name>
            <url-pattern>/servlet1</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>member</role-name>
            <role-name>manager</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>DIGEST</auth-method>
        <realm-name>Digest authentication</realm-name>
    </login-config>
</web-app>

```

---

在浏览器中打开下面的网址，对这个应用程序做一个测试：

<http://localhost:8080/app15c/servlet1>

图 15-2 中展示了摘要访问验证的登录对话框。

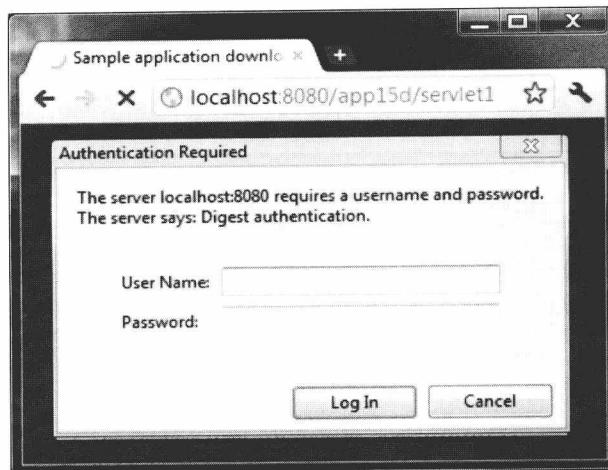


图 15-2 摘要验证

### 15.2.3 基于表单的验证

基本和摘要访问验证都不允许使用定制的登录表单。如果你必须使用定制的表单，那么可以使用基于表单的验证方法。由于它所传输的值没有进行加密，因此这个应该与 SSL 结合起来使用。

使用基于表单的验证时，需要创建一个 Login 页面和一个 Error 页面，它们可以是 HTML 页面，也可以是 JSP 页面。当第一次请求某一个受保护的资源时，Servlet/JSP 容器将会发出 Login 页面。登录成功之后，再发出被请求的资源。但是如果登录失败，用户将会看到 Error 页面。

使用基于表单的验证时，部署描述符中的 auth-method 元素值必须设为 FORM（全部大写）。此外，login-config 元素必须有一个 form-login-config 元素，它带有两个子元素：form-login-page 和 form-error-page。下面就是在基于表单的验证中，login-config 元素的范例。

```
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/login.html</form-login-page>
        <form-error-page>/error.html</form-error-page>
    </form-login-config>
</login-config>
```

代码清单 15-6 中展示了 app15d 的部署描述符，这是一个使用基于表单的验证的范例。

**代码清单 15-6 基于表单验证的部署描述符**

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
    version="3.0"
>
    <!-- restricts access to JSP pages -->
    <security-constraint>
        <web-resource-collection>
            <web-resource-name>JSP pages</web-resource-name>
            <url-pattern>*.jsp</url-pattern>
        </web-resource-collection>
        <!-- must have auth-constraint, otherwise the
            specified web resources will not be restricted -->
        <auth-constraint/>
    </security-constraint>

    <security-constraint>
        <web-resource-collection>
            <web-resource-name>Servlet1</web-resource-name>
            <url-pattern>/servlet1</url-pattern>
        </web-resource-collection>
        <auth-constraint>
            <role-name>member</role-name>
            <role-name>manager</role-name>
        </auth-constraint>
    </security-constraint>

    <login-config>
        <auth-method>FORM</auth-method>
        <form-login-config>
            <form-login-page>/login.html</form-login-page>
            <form-error-page>/error.html</form-error-page>
        </form-login-config>
    </login-config>
</web-app>

```

**form-login-page** 元素引用代码清单 15-7 中的 login.html 页面, **form-error-page** 元素引用代码清单 15-8 中的 error.html 页面。

**代码清单 15-7 login.html 页面**

```

<!DOCTYPE HTML>
<html>
<head>
    <title>Login</title>
</head>
<body>

```

```

<h1>Login Form</h1>
<form action='j_security_check' method='post'>
<div>
    User Name: <input name='j_username' />
</div>
<div>
    Password: <input type='password' name='j_password' />
</div>
<div>
    <input type='submit' value='Login' />
</div>
</form>
</body>
</html>

```

代码清单 15-8 error.html 页面

```

<!DOCTYPE HTML>
<html>
<head>
<title>Login error</title>
</head>
<body>
Login failed.
</body>
</html>

```

在浏览器中打开以下网址，对 app15d 中基于表单的验证做一个测试：

<http://localhost:8080/app15d/servlet1>

图 15-3 展示了供用户登录的 login.html 页面。

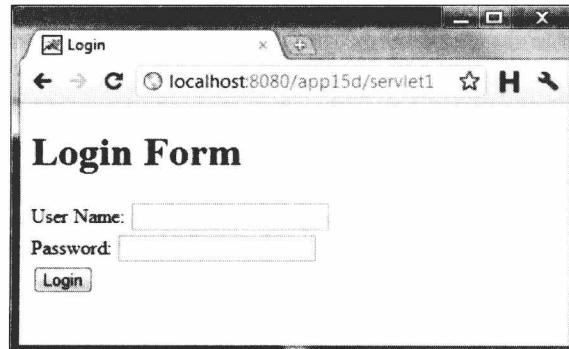


图 15-3 基于表单的验证

## 客户端证书验证

客户端证书验证是在 HTTPS（用 SSL 加密了的 HTTP）上进行的，它要求每一个客户端都要有一个客户端证书。这是一种非常健壮的验证机制，但是不适用于在互联网上部署的应用程序，因为不可能要求每一个互联网用户都拥有一个数字证书。然而，这种验证方法可以用来访问公司内部网的应用程序。

## 15.3 SSL

SSL（Secure Socket Layer，加密套接字层）协议最早是 Netscape 公司开发出来的，它可以针对互联网上的通信进行加密，同时确保数据的保密性和完整性。

要想透彻地了解 SSL 的工作原理，需要学习很多技术，从密码学，到公私钥对，再到证书，等等。本节将详细介绍 SSL 及其组成内容。

### 15.3.1 密码学

有时候，总会需要加密的通信渠道，也就是说，在这样的渠道里，信息是安全的，其他人根本无法理解信息的内容，即使他们能够访问到，也无法擅自进行篡改。

从历史上看，密码学只是关注加密和解密，在这里交换信息的双方都可以放心，因为可以确保只有他们才能够解读这些信息。一开始，人们是利用对称加密进行加密和解密的。在对称加密（Symmetric Cryptography）中，用同一个密钥对信息进行加密和解密。这是一种非常简单的加密 / 解密技术。

假设，加密方法是用一个加密数字将字母表中的每一个字母都往前移。因此，如果加密数字为 2，“ThisFriday”经过加密之后就变成了“VjkuHtkfca”。当你读完字母表时，又从头开始，因此 y 变成了 a。接收方知道密钥为 2，可以很容易地破解信息。

但是，对称加密要求双方都要提前知道加密 / 解密的密钥。基于以下原因，对称加密不适用于互联网：

- 交换信息的双方经常是互相不认识的。例如，在 Amazon 网站上买书时，要将你的个人资料和信用卡资料发过去。如果使用对称加密术，你将不得不在进行交易之前给 Amazon 网站打电话，双方先确定一下密钥。
- 大家都希望能够与许多其他方进行通信。如果使用对称加密术，大家只好保留许多不同的唯一密钥，每个不同的通信方一个密钥。
- 由于你不认识即将进行通信的实体，因此必须确认对方是否真的是他们所声称的那个人。
- 在互联网上传递的信息要经过许多不同的计算机，因此要窃取别人的信息是很容易的事情。对称加密不能确保第三方不会篡改数据。

因此，当今互联网上的加密通信是采用非对称加密（Asymmetric Cryptography），它具有以下三个特征：

- 加密 / 解密。加密过的信息对第三方隐藏了信息。只有既定的接收方才能对信息进行解密。
- 验证。验证是证明某个实体是否为它所声明的那一个。
- 数据完整性。在互联网上发送的信息要传过许多计算机，因此必须确保所发送的数据没有被修改过，并且是完整的。

在非对称加密中，使用的是公钥加密法。在这种加密方法中，数据的加密和解密是通过一对非对称的密钥来实现的，即一个公钥和一个私钥。私钥是私人的，主人必须把它保存在一个安全的地方，绝不能被任何其他方占有。公钥则是分发给公众的，通常想要与这些公钥的主人进行通信时，就可以下载到。配对的公钥和私钥可以利用工具来生成。本章稍后会讲到这些工具。

公钥加密的魅力在于，用公钥加密的数据只能利用对应的私钥才能进行解密；由此类推，用私钥加密的数据也只能用对应的公钥才能进行解密。这种完美的算法是基于非常巨大的质数，由麻省理工学院（MIT）的 Ron Rivest、Adi Shamir 和 Len Adleman 发明于 1977 年。他们将这种算法简单地称作 RSA 算法，即以他们三个人姓的首字母组合而成。

事实证明，RSA 算法非常适用于互联网，尤其适用于电子商务，因为卖家只需要有一个密码对，就可以与他的所有买家进行加密通信。

在举例说明公钥加密方法时，通常使用两个实体，称作 Bob 和 Alice，在这里我们也照此沿用。

### 15.3.2 加密 / 解密

想要交换信息的双方当中，必须要有一方拥有一对密钥。假设 Alice 想要与 Bob 进行通信，Bob 有一个公钥和一个私钥。Bob 要将他的公钥发给 Alice，Alice 就可以用它对要发给 Bob 的信息进行加密。只有 Bob 能够对这些信息进行解密，因为他拥有对应的私钥。为了将信息发送给 Alice，Bob 要利用他的私钥进行加密，Alice 就可以用 Bob 的公钥进行解密。

但是，除非 Bob 能够亲自见到 Alice 并把他的公钥交给她，否则这种方法就很不完善了。但凡拥有密钥对的人都可能声明自己是 Bob，Alice 根本无法查证。在互联网上，交换信息的双方经常住得相隔遥远，见面通常是不太可能的。

### 15.3.3 验证

在 SSL 中，验证是通过引入证书来解决的。证书包含以下内容：

- 一个公钥
- 相关主题的信息，如公钥的所有者

□ 证书颁发商的名称

□ 某种时间戳，使证书过了一定的时期之后会过期

证书最为关键的是，必须通过可信任的证书颁发商进行数字签名，例如 VeriSign 或者 Thawte。对一个电子文件（文档、压缩文件等）进行数字签名，就是将你的签名添加到文档 / 文件中。原始文件没有进行加密，签名的真正目的是要确保文档 / 文件没有被篡改。对一个文档进行签名，要包括创建文档的摘要信息，并利用颁发商的私钥对摘要进行加密。为了查看这个文档是否仍然保持原封不动，要执行以下两个步骤：

1. 利用签署者的公钥对文档的摘要信息进行解密。你很快就会学到，可信任证书颁发商的公钥是很容易获取到的。
2. 给文档创建摘要信息。
3. 将上述第 1 步和第 2 步的结果进行比较。如果这两个结果匹配，则表示文件没有被篡改。

这种验证方法之所以可行，是因为只有私钥的所有者能够对文档摘要进行解密，并且这个摘要信息也只能通过对应的公钥进行解密。假如你能确定自己拥有的是原版的公钥，那么就会知道这个文件有没有被修改过。

**注意** 由于证书可以通过可信任的证书颁发商进行数字签名，因此人们可以将他们的证书供大家公开下载，而不是采用他们的公钥。

证书颁发商有很多，包括 VeriSign 和 Thawte。证书颁发商会有一对公钥和私钥。申请证书时，Bob 必须生成一对密钥，并将其公钥发给证书颁发商，随后证书颁发商就会请 Bob 发一份护照或者其他身份证明给他，由此证明 Bob 的身份。对 Bob 完成验证之后，证书颁发商就会用它的私钥对证书进行签名。“签名”的意思就是加密。因此，证书只能利用证书颁发商的公钥才能解读。证书颁发商的公钥一般很容易下载到。例如，Internet Explorer、Netscape、FireFox 和其他浏览器都默认包含几家证书颁发商的公钥。

例如，在 IE 中，选择 Tools → Internet Options → Content → Certificates → Trusted Root Certification Authorities 键，就可以看到它的证书列表（如图 15-4 所示）。

现在，有了数字证书之后，Bob 在与别人交换信息之前，可以先发布他的数字证书，而不是发布他的公钥了。

其工作流程如下：

A->B Bob，你好！我要跟你通话，但我需要先确认一下你是否真是 Bob。

B->A 当然可以，这是我的证书。

A->B 这些还不够，我还需要你其他的身份证明资料。

B->A Alice，真的是我+[用 Bob 的密钥加密过的信息摘要]。

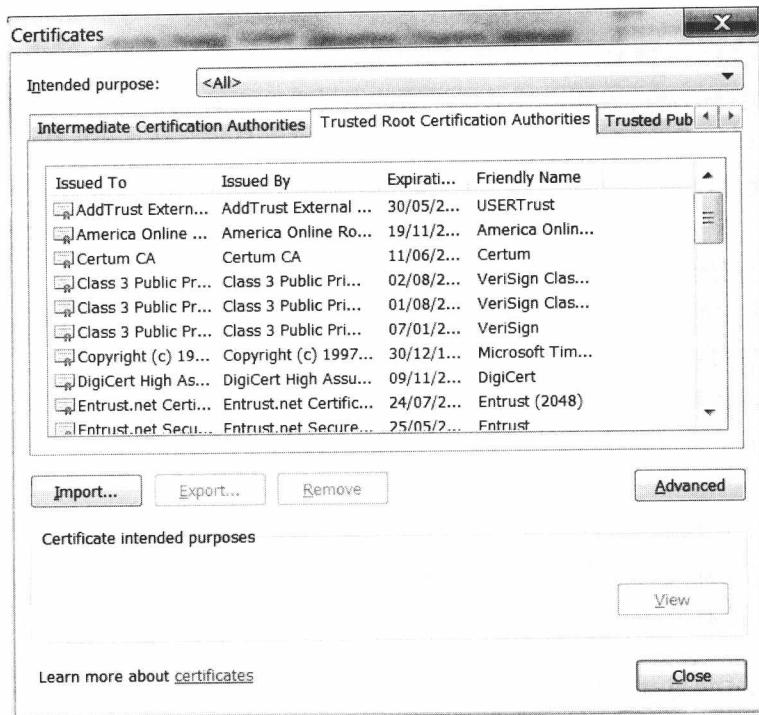


图 15-4 将公钥放在 Internet Explorer 中的证书颁发商

在 Bob 发给 Alice 的最后一条消息中，也就是用 Bob 的私钥进行签名的那条消息，可以向 Alice 证明这条信息是可信的。这就是验证的过程。Alice 联系 Bob，Bob 将其证书发过去。但是，光有证书还不够，因为任何人都可能得到 Bob 的证书。别忘了，Bob 可是将他的证书发给所有想要与他通信的人啊！因此，Bob 还要给她发送一条消息（“Alice，真的是我”），并附上曾用其私钥对这条信息进行过加密的摘要。

Alice 从证书中获得 Bob 的公钥。这个她可以做到，因为证书是利用证书颁发商的私钥签名的，Alice 可以访问证书颁发商的公钥（她的浏览器上就有一份）。现在，她又收到一条消息，以及用 Bob 私钥加密过的摘要。Alice 只要给这条消息创建一个摘要，并将它与 Bob 发给她的加密摘要进行对比即可。Alice 之所以能够解密，是因为它是用 Bob 的私钥加密的，而 Alice 又有一份 Bob 的公钥。如果两者匹配，Alice 就可以确定对方就是真正的 Bob。

Alice 对 Bob 完成验证之后，第一件事就是发送一个加密密钥，这个将在接下来的信息交流中使用。是的，一旦建立了加密渠道，SSL 就会采用对称加密法，因为它的速度要比不对称加密法快得多。

这里还漏掉了一件事情没有交代。互联网上的信息是要在许多计算机上传递的，那么你如何确保那些信息的完整性呢？因为在传输途中，任何人都可能截获那些信息啊。

### 15.3.4 数据完整性

假设 Mallet 是一个怀有恶意的人，他很可能就坐在 Alice 和 Bob 之间，试图破解正在传递的信息。令 Mallet 遗憾的是，虽然他能够复制到信息，但是因为进行过加密，他并不知道密钥。但 Mallet 仍然可能破坏信息，或者不转播其中的某一个部分。为了解决这个问题，SSL 引用了一个消息验证码（Message Authentication Code, MAC）。MAC 是通过一个密钥和一些传输数据计算出来的一组数据。由于 Mallet 不知道密钥，因此他无法算出摘要的正确值。信息的接收者则可以，并且因此可以发现是否有人试图破坏数据，或者发现数据是否不完整。如果答案是肯定的，那么双方可以停止通信。

这类信息摘要算法之一为 MD5，是由 RSA 发明的，非常安全。例如，如果使用 128 位的 MAC 值，那么恶意破坏者猜中正确值的几率大约为 18 446 744 073 709 551 616 分之一，也就是说，几乎永远不可能猜中。

### 15.3.5 SSL 工作原理

知道了 SSL 如何解决加密 / 解密、验证及数据完整性的问题，现在来看一下 SSL 的工作原理。这次我们以 Amazon（代替 Bob）和一个买家（代替 Alice）为例。就像其他任何合法的电子商务卖家一样，Amazon 也向一个可信任的证书颁发商申请了一份证书。买家使用的是 Internet Explorer 浏览器，其中嵌有可信任证书颁发商的公钥。买家并不需要真正了解 SSL 的工作原理，也不需要拥有公钥或者私钥。他只需要确保在输入重要的资料时，例如信用卡号码时，用 HTTPS 代替 HTTP 协议即可。这个必须显示在地址栏中的。因此，不要用 <http://www.amazon.com>，而是必须用 https 开头，例如：<https://secure.amazon.com>。有些浏览器还会在地址栏中显示一个加密图标。图 15-5 展示了 IE 中的加密图标。

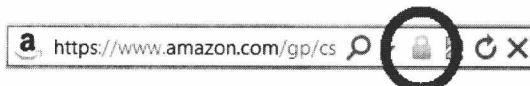


图 15-5 IE 中的加密图标

当买家输入一个加密页面（当他购物完成时），在后台中，这个浏览器和 Amazon 服务器之间会发生以下一系列事件。

浏览器：你真的是 Amazon.com 吗？

服务器：是的，这是我的证书。

然后，浏览器会利用证书颁发商的公钥进行解密，来验证证书的有效性。如果有哪里不对，例如证书过期，浏览器就会向用户发出警告。如果用户不顾证书过期，同意继续，浏览器就会继续。

浏览器：光有证书还不够，请再发点别的身份证明给我。

服务器：我真的是 Amazon.com+[ 这条信息用 Amazon.com 的私钥加密过的摘要 ]。

浏览器利用 Amazon 的公钥对消息摘要进行解密，并为“我真的是 Amazon.com”创建一条消息摘要。如果两者匹配，表示验证成功。随后，浏览器就会产生一个随机密钥，利用 Amazon 的公钥对它进行加密。这个随机密钥将用于对后续的消息进行加密和解密。换句话说，一旦完成对 Amazon 的验证，就会采用对称加密，因为它的速度比非对称加密要快得多。除了消息之外，双方还要发送消息摘要，以确保这些消息是完整的，没有被修改过。

在附录 C 中，将解释如何创建你自己的数字证书，并将逐步说明如何生成公 / 私密钥对，以及如何请一个可信任的证书颁发商为公钥进行数字签名，使之成为一份数字证书。

## 15.4 通过编程确保安全性

尽管声明式安全简单易用，但偶尔还是需要通过编写代码来确保应用程序的安全性。为此，可以在 `HttpServletRequest` 接口中使用安全注解类型和方法。这两者都将在下面进行讨论。

### 15.4.1 安全注解类型

在前一节中已经学过如何在部署描述符中利用 `security-constraint` 元素限制对某些资源的访问。这个元素的其中一个特征是，使用一种与要限制的资源 URL 匹配的 URL 模式。Servlet 3 中提供了可以在 Servlet 级别中完成同样工作的注解类型。使用这些注解类型时，不需要在部署描述符中添加 `security-constraint` 元素就可以限制对某一个 Servlet 的访问。但是，在部署描述符中仍然需要用一个 `login-config` 元素来选择一种验证方法。

`javax.servlet.annotation` 包中与安全相关的 3 个注解类型是：`ServletSecurity`、`HttpConstraint` 和 `HttpMethodConstraint`。

Servlet 类中是利用 `ServletSecurity` 注解类型在 Servlet 中强加安全约束的。`ServletSecurity` 注解可以带有 `value` 和 `httpMethodConstraints` 属性。

`HttpConstraint` 注解类型用来定义一个安全约束，并且只能赋给 `ServletSecurity` 注解的 `value` 属性。如果包含的 `ServletSecurity` 注解中没有出现 `httpMethodConstraints` 属性，那么 `HttpConstraint` 注解所强加的安全约束将应用于所有的 HTTP 方法。否则，安全约束将只应用于 `httpMethodConstraints` 属性中定义的 HTTP 方法。例如，以下 `HttpConstraint` 注解表示所注解的 Servlet 只能由 `manager` 角色中的访客进行访问。

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "manager"))
```

当然，上述注解也可以改写成：

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "manager"))
```

但仍然需要在部署描述符中声明一个 `login-config` 元素，以便容器能够对用户进行验证。将一个 `HttpConstraint` 注解的 `transportGuarantee` 属性设置为 `TransportGuarantee.CONFIDENTIAL`，将使这个 `Servlet` 只能通过某个秘密渠道进行访问，例如 SSL。

```
@ServletSecurity(@HttpConstraint(transportGuarantee =
    TransportGuarantee.CONFIDENTIAL))
```

如果 `Servlet/JSP` 容器通过 HTTP 接收到针对这种 `Servlet` 的请求，它会将浏览器重定向到该 URL 的 HTTPS 版本。

`HttpMethodConstraint` 注解类型用于定义某个安全约束所要应用到的 HTTP 方法。它只能放在 `ServletSecurity` 注解的 `httpMethodConstraints` 属性值数组中。例如，下面的 `HttpMethodConstraint` 注解限制了只有 `manager` 角色的用户可以通过 HTTP Get 访问被注解的 `Servlet`。对于其他的 HTTP 方法，则不存在任何限制。

```
@ServletSecurity(httpMethodConstraints = {
    @HttpMethodConstraint(value = "GET", rolesAllowed = "manager")
})
```

注意，如果 `HttpMethodConstraint` 注解中没有出现 `rolesAllowed` 属性，那么所定义的 HTTP 方法将不受任何限制。例如，以下 `ServletSecurity` 注解中使用了 `value` 和 `httpMethodConstraints` 这两个属性。`httpConstraint` 注解定义了可以访问被注解 `servlet` 的角色，没有 `rolesAllowed` 属性的 `HttpMethodConstraint` 注解，则覆盖了对 Get 方法的约束。因此，任何用户都可以通过 Get 方法访问这个 `Servlet`。另一方面，则只有 `manager` 角色中的用户可以通过其他所有的 HTTP 方法进行访问。

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "manager"),
    httpMethodConstraints = {@HttpMethodConstraint("GET")})
```

但是，如果 `HttpMethodConstraint` 注解类型的 `emptyRoleSemantic` 属性值为 `EmptyRoleSemantic.DENY`，那么该方法将限制所有用户访问。例如，用下列 `ServletSecurity` 进行注解的 `Servlet`，将禁止通过 Get 方法进行访问，但是允许 `member` 角色中的所有用户通过其他的 HTTP 方法进行访问。

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "member"),
    httpMethodConstraints = {@HttpMethodConstraint(value = "GET",
        emptyRoleSemantic = EmptyRoleSemantic.DENY)})
```

## 15.4.2 Servlet 安全 API

除了上一节中讲到的注解类型，也可以在 `HttpServletRequest` 接口中利用以下方法来

实现编程式的安全性。

`java.lang.String getAuthType()`

返回用来保护 Servlet 的验证方案，如果该 Servlet 中没有应用任何安全约束，则返回 null。

`java.lang.String getRemoteUser()`

返回发出该请求的用户的登录名，如果该用户未经验证，则返回 null。

`boolean isUserInRole(java.lang.String role)`

返回一个布尔值，表明该用户是否属于规定的角色。

`java.lang.Principal getUserPrincipal()`

返回一个包含当前被验证用户详细信息的 `java.security.Principal`，如果该用户未经验证，则返回 null。

`boolean authenticate(HttpServletRequest response) throws  
java.io.IOException`

命令浏览器显示一个登录窗口，以便对用户进行验证。

`void login(java.lang.String userName, java.lang.String password)  
throws javax.servlet.ServletException`

通过提供的用户名和密码进行登录。如果登录成功，该方法将不返回任何内容。否则，将抛出一个 `ServletException`。

`void logout() throws javax.servlet.ServletException`

注销用户。

举个例子。代码清单 15-9 中的 `ProgrammaticServlet` 类属于 `app15e` 范例应用程序中的一个部分，它展示了如何通过编程来完成对用户的验证。它还配有代码清单 15-10 中的部署描述符，用来声明一个采用摘要访问验证的 `login-config` 元素。

#### 代码清单 15-9 ProgrammaticServlet 类

---

```
package servlet;
import java.io.IOException;
import java.io.PrintWriter;

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

@WebServlet(urlPatterns = { "/prog" })
```

```

public class ProgrammaticServlet extends HttpServlet {
    private static final long serialVersionUID = 87620L;
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
        if (request.authenticate(response)) {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("Welcome");
        } else {
            // user not authenticated
            // do something
            System.out.println("User not authenticated");
        }
    }
}

```

---

代码清单 15-10 app15e 中的部署描述符

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
          version="3.0"
>
    <login-config>
        <auth-method>DIGEST</auth-method>
        <realm-name>Digest authentication</realm-name>
    </login-config>
</web-app>

```

---

当用户第一次请求该 Servlet 时，用户还没有进行验证，验证方法将返回 false。因此，Servlet/JSP 容器会发出一个 **WWW-Authenticate** 标头，使浏览器显示一个登录对话框，进行摘要访问验证。当用户提交填有正确用户名和密码的表单时，**authenticate** 方法返回 true，并显示欢迎消息。

利用以下 URL 可以对应用程序进行测试：

<http://localhost:8080/app15e/prog>

## 15.5 小结

本章学习了如何实现 Web 安全的 4 大支柱：验证、授权、保密性及数据的完整性。Servlet 技术可以帮助你通过声明方式或者编程方式来保护应用程序的安全。

# 第 16 章 部署

Servlet 3 应用程序的部署几乎不费吹灰之力。你只要借助于 Servlet 注解类型，并且根据应用程序的复杂程度，即使不用部署描述符也可以完成对 Servlet/JSP 应用程序的部署。但是，在许多需要更具体配置的情况下，则仍然需要使用部署描述符。每当有部署描述符时，它必须命名为 `web.xml`，并且必须放在 `WEB-INF` 目录下。Java 类必须放在 `WEB-INF/classes` 目录中，Java 类库必须放在 `WEB-INF/lib` 目录中。所有的应用程序资源则必须打包成一个以 `.war` 为扩展名的文件。`.war` 文件相当于 `.jar` 文件。

本章将讨论部署和部署描述符，这是应用程序的一个重要组成部分。

## 16.1 部署描述符概述

在 Servlet 3 之前的版本中，部署时总是需要用一个 `web.xml` 文件，即部署描述符，你可以在其中配置应用程序的各个方面。但在 Servlet 3 中，部署描述符变成是可选的，因为你可以利用注解将一个资源映射为一个 URL 模式。但是，如果属于以下情况之一，则必须使用部署描述符：

- 需要给 `ServletContext` 传递初始参数。
- 有多个过滤器，并且想要指定过滤器的调用顺序时。
- 需要修改会话的超时或过期值时。
- 想要限制对某个资源集合的访问，并且提供一种方法供用户自行完成验证时。

代码清单 16-1 中展示了部署描述符的大体框架。它必须命名为 `web.xml`，并且必须放在应用程序目录的 `WEB-INF` 目录下。

代码清单 16-1 部署描述符的大体框架

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
  version="3.0"
  [metadata-complete="true|false"]
>
...
</web-app>
```

`xsi:schemaLocation` 属性指定了放置该 Schema 的位置，并据此对部署描述符进行验证。`version` 属性则是指定 Servlet 规范的版本。

可选的 `metadata-complete` 属性用于规定部署描述符是否完整。如果它的值为 `true`，Servlet/JSP 容器就必须忽略针对 Servlet 的注解。如果该元素设为 `false`，或者不存在，那么容器就必须在应用程序部署的类文件中查找针对 Servlet 的注解，并对 web fragment 进行扫描。

`web-app` 元素为根元素，并且可以利用子元素来指定以下内容：

- servlet 声明
- servlet 映射
- `ServletContext` 初始参数
- 会话配置
- 监听器类
- 过滤器定义和映射
- MIME 类型映射
- welcome 文件列表
- 错误页面
- JSP 相关的特定设置
- JNDI 设置

关于部署描述符中可能出现的每一种元素的规则，请查阅可从以下网站下载到的 `web-app_3_0.xsd` Schema：

[http://java.sun.com/xml/ns/javaee/web-app\\_3\\_0.xsd](http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd)

`web-app_3.0.xsd` Schema 中还包含了另一个 Schema (`web-common_3_0.xsd`)，其中包含了大部分信息，这个可以在以下网站中找到。

[http://java.sun.com/xml/ns/javaee/web-common\\_3\\_0.xsd](http://java.sun.com/xml/ns/javaee/web-common_3_0.xsd)

`web-common_3.0.xsd` Schema 中又包含了另外两个 Schema：

- `javaee_6.xsd`，定义了其他 Java EE 6 部署类型 (EAR、JAR 和 RAR) 共用的通用元素。
- `jsp_2_2.xsd`，定义了根据 JSP 2.2 规范配置应用程序中 JSP 部分的元素。

下面将列举出部署描述符中可能出现的 Servlet 和 JSP 元素。这里面不包括 Servlet 或 JSP 规范中所没有的 Java EE 元素。

### 16.1.1 核心元素

本节将更详细地讨论一些比较重要的元素。`<web-app>` 的子元素可以任意顺序显示。某些元素，如 `session-config`、`jsp-config` 和 `login-config`，则只能出现一次。其他的元素，如

**servlet**、**filter** 和 **welcome-file-list**，则可以出现多次。

可以直接在 **<web-app>** 中显示的较重要元素将单独放在一个分节中讨论。为了查看某个非直接放在 **<web-app>** 下的元素描述，可到其父元素中查找。例如，关于 **taglib** 元素的内容请查阅“**jsp-config**”小节，关于 **load-on-startup** 元素的内容请查阅“**Servlet**”小节。下面将按照字母顺序进行讨论。

#### context-param

**context-param** 元素用于为 **ServletContext** 赋值。这些值可以通过任何 Servlet/JSP 页面读取。该元素中包含了一个名称 / 值对，它可以通过在 **ServletContext** 中调用 **getInitParameter** 方法获取到。只要其参数名称在整个应用程序中是唯一的，则可以有多个 **context-param** 元素。**ServletContext.getInitParameterNames()** 将返回所有的 **ServletContext** 参数名称。

**context-param** 元素中必须包含一个 **param-name** 元素和一个 **param-value** 元素。**param-name** 元素中包含参数名称，**param-value** 元素中则包含参数值。**description** 元素是可选的，它也可以用来描述参数。

下面举两个 **context-param** 元素的例子。

```
<context-param>
    <param-name>location</param-name>
    <param-value>localhost</param-value>
</context-param>
<context-param>
    <param-name>port</param-name>
    <param-value>8080</param-value>
    <description>The port number used</description>
</context-param>
```

#### distributable

**distributable** 元素用来表明这个应用程序应该部署到一个分布式的 Servlet/JSP 容器中。**distributable** 元素必须为空。例如，下面就是一个 **distributable** 元素。

```
<distributable/>
```

#### error-page

**error-page** 元素中包含了一个 HTTP 错误码与一个资源路径之间的映射，或者是一种 Java 异常类型与一个资源路径之间的映射。**error-page** 元素命令容器，在发生 HTTP 错误事件，或者抛出指定的异常时，应该返回指定的资源。

该元素中必须包含以下子元素：

- **error-code**，指定一个 HTTP 错误码。

□ **exception-type**, 指定要捕捉的 Java 异常类型的全类名。

□ **location**, 指定要在错误事件或者异常中显示的资源的位置。**location** 元素必须以 / 开头。

例如, 下面就是一个 **error-page** 元素, 它告诉 Servlet/JSP 容器, 每当出现 HTTP 404 错误码时, 必须显示放在应用程序目录下的 error.html 页面:

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

下面是一个 **error-page** 元素, 它用 **exceptions.html** 页面映射所有的 Servlet 异常。

```
<error-page>
    <exception-type>javax.servlet.ServletException</exception-type>
    <location>/exception.html</location>
</error-page>
```

#### filter

该元素用来指定一种 Servlet 过滤器, 它至少包含一个 **filter-name** 元素和一个 **filter-class** 元素。另外, 它还可以可选地包含以下元素: **icon**、**display-name**、**description**、**init-param** 和 **async-supported**。

**filter-name** 元素定义过滤器的名称。过滤器的名称在整个应用程序中必须是唯一的。**filter-class** 元素指定过滤器类的全类名。**init-param** 元素用来指定过滤器的初始参数, 其元素描述与 **<context-param>** 的相同。filter 元素中可以有多个 **init-param** 元素。

下面是两个 filter 元素, 其名称分别为 Upper Case Filter 和 Image Filter。

```
<filter>
    <filter-name>Upper Case Filter</filter-name>
    <filter-class>com.example.UpperCaseFilter</filter-class>
</filter>
<filter>
    <filter-name>Image Filter</filter-name>
    <filter-class>com.example.ImageFilter</filter-class>
    <init-param>
        <param-name>frequency</param-name>
        <param-value>1909</param-value>
    </init-param>
    <init-param>
        <param-name>resolution</param-name>
        <param-value>1024</param-value>
    </init-param>
</filter>
```

### filter-mapping

**filter-mapping** 元素定义过滤器要应用到的一个或多个资源。过滤器也可以应用于 Servlet 或者 URL 模式。将过滤器映射到 Servlet，将致使过滤器作用于 Servlet。将过滤器映射到 URL 模式，则将使过滤器作用于所有其 URL 与 URL 模式匹配的任意资源。它们的过滤顺序与 **filter-mapping** 元素在部署描述符中出现的顺序一致。

**filter-mapping** 元素中包含了一个 **filter-name** 元素和一个 **url-pattern** 元素，或者 **servlet-name** 元素。

**filter-name** 值必须与利用 **filter** 元素声明的某一个过滤器名称相匹配。

下面是两个 **filter** 元素和两个 **filter-mapping** 元素：

```
<filter>
    <filter-name>Logging Filter</filter-name>
    <filter-class>com.example.LoggingFilter</filter-class>
</filter>
<filter>
    <filter-name>Security Filter</filter-name>
    <filter-class>com.example.SecurityFilter</filter-class>
</filter>

<filter-mapping>
    <filter-name>Logging Filter</filter-name>
    <servlet-name>FirstServlet</servlet-name>
</filter-mapping>
<filter-mapping>
    <filter-name>Security Filter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

### listener

**listener** 元素用来注册一个监听器，它包含一个 **listener-class** 元素，用于定义监听器类的全类名，例如：

```
<listener>
    <listener-class>com.example.AppListener</listener-class>
</listener>
```

### locale-encoding-mapping-list 和 locale-encoding-mapping

**locale-encoding-mapping-list** 元素中包含一个或多个 **locale-encoding-mapping** 元素。**locale-encoding-mapping** 元素将一个语言环境的名称映射成一个编码，并且包含一个 **locale** 元素和一个 **encoding** 元素。**<locale>** 的值必须是 ISO 639 中定义的某一个 **language-code**，例如 **en**，或者是一个 **language-code\_country-code**，例如 **en\_US**。如果使用的是

`language-code_country-code`, 那么 `country-code` 部分必须为 ISO 3166 中定义的某一个国家代码。

例如, 下面的 `locale-encoding-mapping-list` 中包含了一个 `locale-encoding-mapping` 元素, 它将 Japanese (日文) 映射成 Shift\_JIS 编码。

```
<locale-encoding-mapping-list>
    <locale-encoding-mapping>
        <locale>ja</locale>
        <encoding>Shift_JIS</encoding>
    </locale-encoding-mapping>
</locale-encoding-mapping-list>
```

### login-config

`login-config` 元素指定用来验证用户的验证方法, 如果采用基于表单的验证, 那么则还可以定义表单登录机制所需的 `realm` 名称和属性。`login-config` 元素中有一个可选的 `auth-method` 元素, 一个可选的 `realm-name` 元素, 和一个可选的 `form-login-config` 元素。

`auth-method` 元素指定访问验证方法, 它的可能值为: `BASIC`、`DIGEST`、`FORM` 或者 `CLIENT-CERT`。

`realm-name` 元素指定要在 Basic 访问验证和 Digest 访问验证中使用的 `realm` 名称。

`form-login-config` 元素指定在基于表单的验证中要用到的登录页面和错误页面。如果没有使用基于表单的验证, 则可以忽略这些元素。

`form-login-config` 元素中有一个 `form-login-page` 元素和一个 `form-error-page` 元素。`form-login-page` 元素指定显示 Login 页面的资源路径。这个路径必须以 / 开头, 并且是相对于应用程序目录的。

`form-error-page` 元素用于指定在登录失败时显示错误页面的资源路径。这个路径必须以 / 开头, 并且是相对于应用程序的目录的。

举个例子, 请看下面的 `login-config` 元素:

```
<login-config>
    <auth-method>DIGEST</auth-method>
    <realm-name>Members Only</realm-name>
</login-config>
```

再举个例子:

```
<login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
        <form-login-page>/loginForm.jsp</form-login-page>
```

```

<form-error-page>/errorPage.jsp</form-error-page>
</form-login-config>
</login-config>

```

#### mime-mapping

**mime-mapping** 元素是将一个 MIME 类型映射成一个扩展，它包含一个 **extension** 元素和一个 **mime-type** 元素。**extension** 元素描述扩展，**mime-type** 元素指定 MIME 类型。例如，下面就是一个 **mime-mapping** 元素的例子。

```

<mime-mapping>
  <extension>txt</extension>
  <mime-type>text/plain</mime-type>
</mime-mapping>

```

#### security-constraint

**security-constraint** 元素可以通过声明的方式来限制对某个资源集合的访问。

**security-constraint** 元素中包含一个可选的 **display-name** 元素，一个或多个 **web-resource-collection** 元素，一个可选的 **auth-constraint** 元素，以及一个可选的 **user-data-constraint** 元素。

**web-resource-collection** 元素定义需要限制访问的资源集合。在这里，可以定义一个或者多个 URL 模式，以及一个或者多个需要受到限制的 HTTP 方法。如果没有指定 HTTP 方法，那么这个安全约束将应用于所有的 HTTP 方法。

**auth-constraint** 元素定义可以访问该资源集合的用户角色。如果没有设置 **auth-constraint** 元素，那么该安全约束将应用于所有角色。

**user-data-constraint** 元素用来指明在客户与 Servlet/JSP 容器之间传输的数据应该如何进行保护。

**web-resource-collection** 元素中包含一个 **web-resource-name** 元素，一个可选的 **description** 元素，有零个或者多个 **url-pattern** 元素，以及有零个或者多个 **http-method** 元素。

**web-resource-name** 元素中包含了一个与被保护资源相关的名称。

**http-method** 元素中可以设置一个 HTTP 方法，如 GET、POST 或者 TRACE。

**auth-constraint** 元素中包含一个可选的 **description** 元素，有零个或者多个 **role-name** 元素。**role-name** 元素中包含某个安全角色的名称。

**user-data-constraint** 元素中包含一个可选的 **description** 元素和一个 **transport-guarantee** 元素。**transport-guarantee** 元素必须具有以下其中一个值：NONE、INTEGRAL 或者 CONFIDENTIAL。NONE 表示该应用程序不需要传输保证。INTEGRAL 表示服务器与客户端之间的数据应该以一种在传输过程中无法更改的方式进行发送。CONFIDENTIAL 表示所传输的数据必须进行加密。在大多数情况下，SSL (Secure Sockets Layer，安全套接字层) 是用于 INTEGRAL 或者 CONFIDENTIAL 的。

以下范例是用一个 **security-constraint** 元素来限制对任何 URL 中带有 /members/\* 的资源的访问权限，它只允许 **playingMember** 角色中的用户访问。**login-config** 元素则要求用户登录，并且使用 Digest 访问验证方法。

```
<security-constraint>
    <web-resource-collection>
        <web-resource-name>Members Only</web-resource-name>
        <url-pattern>/members/*</url-pattern>
    </web-resource-collection>
    <auth-constraint>
        <role-name>payingMember</role-name>
    </auth-constraint>
</security-constraint>

<login-config>
    <auth-method>Digest</auth-method>
    <realm-name>Digest Access Authentication</realm-name>
</login-config>
```

#### **security-role**

**security-role** 元素用来指定安全约束中所用的安全角色声明。这个元素有一个可选的 **description** 元素和一个 **role-name** 元素。下面是一个 **security-role** 元素的示例。

```
<security-role>
    <role-name>payingMember</role-name>
</security-role>
```

#### **servlet**

**servlet** 元素用来声明一个 Servlet，它可以包含以下元素：

- 一个可选的 **icon** 元素
- 一个可选的 **description** 元素
- 一个可选的 **display-name** 元素
- 一个 **servlet-name** 元素
- 一个 **servlet-class** 元素，或者一个 **jsp-file** 元素
- 有零个或者多个 **init-param** 元素
- 一个可选的 **load-on-startup** 元素
- 一个可选的 **run-as** 元素
- 一个可选的 **enabled** 元素
- 一个可选的 **async-supported** 元素
- 一个可选的 **multipart-config** 元素
- 有零个或者多个 **security-role-ref** 元素

一个  **servlet**  元素中至少必须包含一个  **servlet-name**  元素和一个  **servlet-class**  元素，或者一个  **servlet-name**  元素和一个  **jsp-file**  元素。 **servlet-name**  元素定义这个 Servlet 的名称，并且它在整个应用程序中必须是唯一的。

**servlet-class**  元素用于指定 Servlet 的全类名。

**jsp-file**  元素用于指定一个 JSP 页面在应用程序中的完整路径。这个完整路径必须以 / 开头。

**init-param**  子元素可以用于给 Servlet 设置初始参数名称和值。 **init-param**  的元素描述与  **context-param**  的一样。

使用  **load-on-startup**  元素时，可以在 Servlet/JSP 容器启动时自动将 Servlet 加载到内存中。加载 Servlet 意味着将 Servlet 实例化，并调用其  **init**  方法。使用该元素可以避免对 Servlet 初次请求的延迟响应，这是因为将 Servlet 加载到内存中造成的。如果有这个元素，并且指定了一个  **jsp-file**  元素，那么这个 JSP 文件就会预先被编译成一个 Servlet，并加载生成这个 Servlet。

**load-on-startup**  必须为空，或者它的值必须为整数值。这个值表示当同一个应用程序中有多个 Servlet 时，其中这个 Servlet 的加载顺序。例如，如果有两个  **servlet**  元素，它们都包含一个  **load-on-startup**  元素，那么其  **load-on-startup**  值较小的 Servlet 将先被加载。如果  **load-on-startup**  的值为空，或者是一个负数值，那么将由 Web 容器决定什么时候加载这个 Servlet。如果两个 Servlet 的  **load-on-startup**  值相同，那么将无法确定这两个 Servlet 的加载顺序。

定义  **run-as**  将覆盖这个应用程序中调用 Enterprise JavaBean 的 Servlet 安全标识，并以为当前 Web 应用程序定义的其中一个安全角色作为角色名称。

**security-role-ref**  元素把利用  **isUserInRole(name)**  从 Servlet 调用的角色名称，映射成为应用程序定义的一个安全角色名称。 **security-role-ref**  元素中包含一个可选的  **description**  元素，一个  **role-name**  元素，以及一个  **role-link**  元素。

**role-link**  元素用于将一个安全角色引用链接到一个已定义的安全角色。 **role-link**  元素必须包含在  **security-role**  元素中定义的某个安全角色名称。

**async-supported**  元素是一个可选的元素，它的值为  **true**  或  **false** ，表示这个 Servlet 是否支持异步处理。

**enabled**  元素也是一个可选的元素，它的值可能为  **true**  或  **false** 。将这个元素设置为  **false** ，将关闭这个 Servlet。

例如，将安全角色引用 PM 映射到名为  **payingMember**  的安全角色，其语法如下：

```
<security-role-ref>
  <role-name>PM</role-name>
```

```
<role-link>payingMember</role-link>
</security-role-ref>
```

在这个例子中，如果是由属于 payingMember 安全角色的用户调用 Servlet，那么调用 isUserInRole("payingMember") 时，其结果将为 true。

下面举两个  **servlet**  元素的例子：

```
<servlet>
    <servlet-name>UploadServlet</servlet-name>
    <servlet-class>com.brainysoftware.UploadServlet</servlet-class>
    <load-on-startup>10</load-on-startup>
</servlet>
<servlet>
    <servlet-name>SecureServlet</servlet-name>
    <servlet-class>com.brainysoftware.SecureServlet</servlet-class>
    <load-on-startup>20</load-on-startup>
</servlet>

servlet-mapping
```

**servlet-mapping**  元素将一个 Servlet 映射到一个 URL 模式。 **servlet-mapping**  元素必须有一个  **servlet-name**  元素和一个  **url-pattern**  元素。

下面的  **servlet-mapping**  元素将一个 Servlet 映射到 URL 模式 /first 。

```
<servlet>
    <servlet-name>FirstServlet</servlet-name>
    <servlet-class>com.brainysoftware.FirstServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>FirstServlet</servlet-name>
    <url-pattern>/first</url-pattern>
</servlet-mapping>
```

**session-config**

**session-config**  元素为 `javax.servlet.http.HttpSession` 实例定义参数。这个元素可以包含一个或者多个以下元素： **session-timeout** 、 **cookie-config**  或  **tracking-mode**  。

**session-timeout**  元素定义默认会话超时的时间间隔，其单位为分钟。这个值必须为整数值。如果  **session-timeout**  元素的值为 0，或者为负数，那么这个会话将永远不会超时。

**cookie-config**  元素定义会话的配置，用来追踪这个 Servlet/JSP 应用程序创建的 cookie。

**tracking-mode**  元素为该 Web 应用程序创建的会话定义追踪模式，其有效值为 COOKIE、URL 或者 SSL 。

下面的  **session-config**  元素将使当前应用程序中的  `HttpSession`  对象在 12 分钟没有活动之后失效。

```
<session-config>
    <session-timeout>12</session-timeout>
</session-config>
```

#### welcome-file-list

welcome-file-list 元素定义了当用户在浏览器中输入的 URL 不包含 Servlet 名称或 JSP 页面或静态资源时，要显示的文件或者 Servlet。

welcome-file-list 元素中包含一个或者多个 welcome-file 元素。welcome-file 元素中包含默认的文件名称。如果找不到第一个 welcome-file 元素中指定的文件，Web 容器将尝试进行显示第二个，依此类推。

下面列举一个 welcome-file-list 元素：

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

以下范例使用了一个包含两个 welcome-file 元素的 welcome-file-list 元素。第一个 welcome-file 元素定义了应用程序目录下一个名为 index.html 的文件；第二个定义了 servlet 目录下的 welcome Servlet，它放在应用程序目录下：

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

### 16.1.2 特定于 JSP 的元素

<web-app> 下的 jsp-config 元素中包含了特定于 JSP 的元素。它可以有零个或者多个 taglib 元素，以及有零个或者多个 jsp-property-group 元素。taglib 元素和 jsp-property-group 元素将在下面讨论。

#### taglib

taglib 元素描述了一个 JSP 定制标签类库。taglib 元素中包含了一个 taglib-uri 元素和一个 taglib-location 元素。

taglib-uri 元素定义了 Servlet/JSP 应用程序中所用的标签类库 URI。<taglib-uri> 的值是相对于部署描述符的位置。

taglib-location 元素为标签类库定义了 TLD 文件的位置。

下面举一个 taglib 元素的例子。

```
<jsp-config>
  <taglib>
    <taglib-uri>
      http://brainysoftware.com/taglib/complex
    </taglib-uri>
    <taglib-location>/WEB-INF/jsp/complex.tld
  </taglib-location>
</taglib>
</jsp-config>
```

### jsp-property-group

**jsp-property-group** 元素中集合了许多 JSP 文件，以便可以为它们提供通用的属性信息。利用 **<jsp-property-group>** 下的子元素可以完成以下任务：

- 是否忽略 EL
- 是否允许有脚本元素
- 页面编码信息
- 某个资源是一个 JSP 文档（用 XML 编写）
- 前言及代码自动生成的内容

**jsp-property-group** 元素具有以下子元素：

- 一个可选的 **description** 元素
- 一个可选的 **display-name** 元素
- 一个可选的 **icon** 元素
- 一个或多个 **url-pattern** 元素
- 一个可选的 **el-ignored** 元素
- 一个可选的 **page-encoding** 元素
- 一个可选的 **scripting-invalid** 元素
- 一个可选的 **is-xml** 元素
- 零个或者多个 **include-prelude** 元素
- 零个或者多个 **include-code** 元素

**url-pattern** 元素用于指定一个将受属性设置影响的 URL 模式。

**el-ignored** 元素的值可以为 true 或 false。true 表示在 URL 与指定 URL 模式匹配的 JSP 页面中将不运算这些 EL 表达式。这个元素的默认值为 false。

**page-encoding** 元素为 URL 与指定 URL 模式匹配的 JSP 页面指定编码。**page-encoding** 的有效值与匹配 JSP 页面中所用的 **page** 指令的 **pageEncoding** 属性值相同。如果 JSP 页面中 **page** 指令的 **pageEncoding** 属性中的编码，与匹配页面的 JSP 配置元素中的编码不同，那么将会有个翻译时的错误。如果在 XML 语法中文档的前言或者文本声明的编码，与匹配文档的 JSP 配置元素中的编码不同，也会出现翻译时的错误。在多种机制中可以使用相同

的编码。

**scripting-invalid** 元素值为布尔值。true 表示 URL 与指定模式匹配的 JSP 页面中不允许有 scripting。**scripting-invalid** 元素的默认值为 false。

**is-xml** 元素的值也为布尔值。true 表示 URL 与指定模式匹配的 JSP 页面属于 JSP 文档。

**include-prelude** 元素是一个相对于 context 的路径，它必须与 Servlet/JSP 应用程序中的某个元素相对应。当有这个元素时，那么在 URL 与指定模式匹配的每个 JSP 页面开头处都会自动包含指定的路径（如 **include** 指令所示）。

**include-coda** 元素是一个相对于 context 的路径，它必须与应用程序中的某个元素相对应。当有这个元素时，那么这个 **jsp-property-group** 元素中的每个 JSP 页面结尾处都会自动包含指定的路径（如 **include** 指令所示）。

例如，下面就是一个使所有 JSP 页面中的 EL 运算都会被忽略的 **jsp-property-group** 元素。

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <el-ignored>true</el-ignored>
  </jsp-property-group>
</jsp-config>
```

下面是一个 **jsp-property-group** 元素，用于强制整个应用程序中的所有 JSP 页面中都没有脚本。

```
<jsp-config>
  <jsp-property-group>
    <url-pattern>*.jsp</url-pattern>
    <scripting-invalid>true</scripting-invalid>
  </jsp-property-group>
</jsp-config>
```

## 16.2 部署

自从有了 Servlet 开始，部署 Servlet/JSP 应用程序一直都是一件很容易的事情。它只需要将其原始目录结构下的所有应用程序资源压缩成一个 war 文件即可。你也可以使用 JDK 中的 jar 工具，或者像 WinZip 这样的流行工具。总之，你只需要确保压缩好的文件扩展名为 war 即可。如果使用的是 WinZip，压缩好之后重新命名一下即可。

我们必须将所有的类库和类文件以及 HTML 文件、JSP 页面、图片、版权信息（如果有）等，全部放在 war 文件中。但是不要包含 Java 源文件。需要用到该应用程序的任何人只需要复制一份 war 文件，并将它部署到一个 Servlet/JSP 容器中即可。

### 16.3 Web Fragment

Servlet 3 中新增了 web fragments，这是在现成 Web 应用程序中部署插件或者框架 (framework) 的一项新特性。web fragment 旨在对部署描述符进行补充，无须编辑 web.xml 文件。web fragment 基本上就是一个压缩包 (jar 文件)，其中包含了常用的 Web 对象，如 Servlet、过滤器、监听器，以及其他资源，如 JSP 页面和静态图片。web fragment 也可以带有描述符，这是一个与部署描述符类似的 XML 文档。web fragment 描述符必须命名为 web-fragment.xml，并且必须放在压缩包的 META-INF 目录下。web fragment 描述符中可以包含部署描述符中 web-app 元素下的所有元素，以及一些特定于 web fragment 的元素。一个应用程序中可以有多个 web fragment。

代码清单 16-2 中展示了 web fragment 描述符的大体框架，其中粗体字部分的代码强调了它与部署描述符之间的区别。毫无疑问，web fragment 中的根元素就是 **web-fragment**。web-fragment 元素中甚至可以带有 **metadata-complete** 属性。如果 **metadata-complete** 属性的值为 true，那么 web fragment 中包含的类中的注解将会全部被忽略。

代码清单 16-2 web-fragment.xml 文件的大体框架

---

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-fragment 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-fragment_3_0.xsd
  version="3.0"
  [metadata-complete="true|false"]
>

...
</web-fragment>
```

---

为了更好地示范，app16a 应用程序在 fragment.jar 文件中包含了一个 web fragment。这个 jar 文件已经被导入到 app16a 的 WEB-INF/lib 目录下。这个例子的重点不在 app16a，而在 webfragment 工程，它里面包含了一个 Servlet (`fragment.servlet.FragmentServlet`，如代码清单 16-3 所示) 和一个 `web-fragment.xml` 文件 (如代码清单 16-4 所示)。

代码清单 16-3 FragmentServlet 类

---

```
package fragment.servlet;
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 FragmentServlet extends HttpServlet {

    private static final long serialVersionUID = 940L;

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

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("A plug-in");
    }
}

```

代码清单 16-4 工程 webfragment 中的 web fragment 描述符

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-fragment 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-fragment_3_0.xsd"
               version="3.0"
>
    <servlet>
        <servlet-name>FragmentServlet</servlet-name>
        <servlet-class>fragment.servlet.FragmentServlet</servlet-
        class>
    </servlet>
    <servlet-mapping>
        <servlet-name>FragmentServlet</servlet-name>
        <url-pattern>/fragment</url-pattern>
    </servlet-mapping>
</web-fragment>

```

FragmentServlet 是一个简单的 Servlet，它的作用是将一个字符串发送到浏览器。web-fragment.xml 文件负责注册并映射 Servlet。fragment.jar 文件的结构如图 16-1 所示。

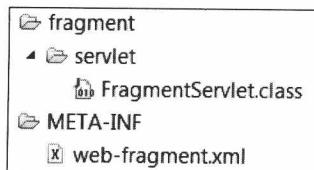


图 16-1 fragment.jar 文件的结构

调用下面这个 URL 可以对 Fragment Servlet 进行测试：

<http://localhost:8080/app16a/fragment>

你将会看到 Fragment Servlet 的输出。

## 16.4 小结

本章介绍了如何配置和部署 Servlet/JSP 应用程序。首先介绍了一个典型的应用程序的目录结构，然后接着介绍部署描述符。

当应用程序准备好要进行部署时，可以使用应用程序的文件和目录结构进行部署，也可以将应用程序压缩成一个 WAR 文件，然后通过单个文件来部署整个应用程序。

# 第 17 章 动态注册和 Servlet 容器初始化

动态注册是 Servlet 3 的一项新特性，它不需要重新加载应用程序便可安装新的 Web 对象（如 Servlet、过滤器和监听器等）。Servlet 容器初始化也是 Servlet 3 中新增的一项特性，它特别适用于框架开发者。本章将介绍其特性，并举例说明。

## 17.1 动态注册

为了使动态注册成为可能，`ServletContext` 接口中还添加了以下方法，用来动态地创建 Web 对象：

```
<T extends Filter> createFilter(java.lang.Class<T> clazz)
<T extends java.util.EventListener> createListener(
    java.lang.Class<T> clazz)
<T extends Servlet> createServlet(java.lang.Class<T> clazz)
```

例如，假设 `MyServlet` 是一个可以直接或间接实现 `javax.servlet.Servlet` 的类，那么可以通过调用 `createServlet` 方法将 `MyServlet` 实例化：

```
Servlet myServlet = createServlet(MyServlet.class);
```

创建好一个 Web 对象之后，可以利用以下任意一个方法将它添加到 `ServletContext` 中，这也是 Servlet 3 的一项新特性。

```
FilterRegistration.Dynamic addFilter(java.lang.String filterName,
    Filter filter)
<T extends java.util.EventListener> void addListener(T t)
ServletRegistration.Dynamic addServlet(java.lang.String
    servletName, Servlet servlet)
```

同样地，也可以通过在 `ServletContext` 中调用以下任意一个方法，在创建 Web 对象的同时，将它添加到 `ServletContext` 中：

```
FilterRegistration.Dynamic addFilter(java.lang.String filterName,
    java.lang.Class<? extends Filter> filterClass)
FilterRegistration.Dynamic addFilter(java.lang.String filterName,
    java.lang.String className)
void addListener(java.lang.Class<? extends java.util.EventListener>
```

```

    listenerClass)
void addListener(java.lang.String className)
ServletRegistration.Dynamic addServlet(java.lang.String
    servletName, java.lang.String className)
ServletRegistration.Dynamic addServlet(java.lang.String
    servletName, java.lang.String className)

```

在创建或添加监听器时，传给第一个 `addListener` 覆盖方法的类必须实现以下其中一个或多个接口：

- `ServletContextAttributeListener`
- `ServletRequestListener`
- `ServletRequestAttributeListener`
- `HttpSessionListener`
- `HttpSessionAttributeListener`

如果将 `ServletContext` 传给 `ServletContextInitializer` 的 `onStartup` 方法，那么监听器类也可以实现 `ServletContextListener` 接口。关于 `startUp` 方法和 `ServletContextInitializer` 接口的更多详情，请查看接下来的内容。

`addFilter` 或 `addServlet` 方法的返回值是一个 `FilterRegistration.Dynamic` 或者 `ServletRegistration.Dynamic`。

`FilterRegistration.Dynamic` 或者 `ServletRegistration.Dynamic` 都是 `Registration.Dynamic` 的子接口。`FilterRegistration.Dynamic` 可以配置一个过滤器，`ServletRegistration.Dynamic` 则可以配置一个 Servlet。

举个例子，请看 `app17a` 应用程序，其中包含了一个 Servlet： `FirstServlet`，和一个监听器：`DynRegListener`。这个 Servlet 没有用 `@.WebServlet` 进行标注，也没有在部署描述符中进行声明。监听器动态地注册 Servlet，并将它投入使用。

`FirstServlet` 类如代码清单 17-1 所示，`DynRegListener` 类如代码清单 17-2 所示。

代码清单 17-1 `FirstServlet` 类

---

```

package servlet;
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 FirstServlet extends HttpServlet {
    private static final long serialVersionUID = -6045338L;

    private String name;

```

```

@Override
public void doGet(HttpServletRequest request,
    HttpServletResponse response)
    throws ServletException, IOException {
    response.setContentType("text/html");
    PrintWriter writer = response.getWriter();
    writer.println("<html><head><title>First servlet" +
        "</title></head><body>" + name);
    writer.println("</body></head>");
}
}

public void setName(String name) {
    this.name = name;
}
}

```

代码清单 17-2 DynRegListener 类

```

package listener;
import javax.servlet.Servlet;
import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.ServletRegistration;
import javax.servlet.annotation.WebListener;
import servlet.FirstServlet;

@WebListener
public class DynRegListener implements ServletContextListener {

    @Override
    public void contextDestroyed(ServletContextEvent sce) {
    }

    // use createServlet to obtain a Servlet instance that can be
    // configured prior to being added to ServletContext
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext servletContext = sce.getServletContext();

        Servlet firstServlet = null;
        try {
            firstServlet =
                servletContext.createServlet(FirstServlet.class);
        } catch (Exception e) {
            e.printStackTrace();
        }

        if (firstServlet != null && firstServlet instanceof
            FirstServlet) {
            ((FirstServlet) firstServlet).setName(
                "Dynamically registered servlet");
        }
    }
}

```

```

    }

    // the servlet may not be annotated with @WebServlet
    ServletRegistration.Dynamic dynamic = servletContext.
        addServlet("firstServlet", firstServlet);
    dynamic.addMapping("/dynamic");
}

}

```

---

当应用程序启动时，容器会调用监听器的 `contextInitialized` 方法，结果是创建了一个 `FirstServlet` 实例，并注册和映射到 `/dynamic`。如果一切顺利，应该可以利用下面这个 URL 来调用 `FirstServlet`。

`Http://localhost:8080/app17a/dynamic`

## 17.2 Servlet 容器初始化

如果你使用过像 Struts 或 Struts 2 这类 Java Web 框架，就应该知道，在使用框架之前必须先配置应用程序。一般来说，需要通过修改部署描述符，告诉 Servlet 容器你正在使用一个框架。例如，要想在应用程序中使用 Struts 2，可以将以下标签添加到部署描述符中：

```

<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>

```

但在 Servlet 3 中，就不再需要这些了。框架可以进行打包，自动完成 Web 对象的首次注册。

Servlet 容器初始化的核心是 `javax.servlet.ServletContainerInitializer` 接口。这是一个简单的接口，它只有一个方法：`onStartup`。在执行任何 `ServletContext` 监听器之前，由 Servlet 容器调用这个方法。

`onStartup` 的方法签名如下：

```
void onStartup(java.util.Set<java.lang.Class<?>> clazz,
               ServletContext servletContext)
```

实现 `ServletContainerInitializer` 的类必须用 `@HandlesTypes` 注解进行标注，以便声明初始化程序可以处理这些类型的类。

举个例子，本书配套的 `initializer.jar` 类库中包含了一个 Servlet 容器初始化程序，它注册了一个名为 `UsefulServlet` 的 Servlet。图 17-1 展示了 `initializer.jar` 的目录结构。

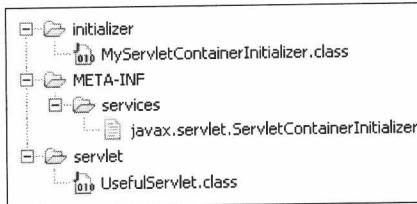


图 17-1 `initializer.jar` 的目录结构

这个类库是一个可插拔的框架，这里有两个重要的资源：初始化类（`initializer.MyServletContainerInitializer`，如代码清单 17-3 所示），和一个元数据文本文件（`javax.servlet.ServletContainerInitializer`）。这个文本文件必须放在 jar 文件的 `META-INF/services` 下，文件中只有一行内容，即实现了 `ServletContainerInitializer` 接口的类的名称，如代码清单 17-4 所示。

代码清单 17-3 `ServletContainerInitializer`

---

```

package initializer;
import java.util.Set;
import javax.servlet.ServletContainerInitializer;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.ServletRegistration;
import javax.annotation.HandlesTypes;
import servlet.UsefulServlet;

@HandlesTypes({UsefulServlet.class})
public class MyServletContainerInitializer implements
    ServletContainerInitializer {

    @Override
    public void onStartup(Set<Class<?>> classes, ServletContext
        servletContext) throws ServletException {

        System.out.println("onStartup");
        ServletRegistration registration =
            servletContext.addServlet("usefulServlet",
                "servlet.UsefulServlet");
        registration.addMapping("/useful");
        System.out.println("leaving onStartup");
    }
}
  
```

---

---

代码清单 17-4 javax.servlet.ServletContainerInitializer 文件

---

```
initializer.MyServletContainerInitializer
```

---

**MyServletContainerInitializer** 中 `onStartup` 方法的主要任务是注册 Web 对象。在本例中，这种对象只有一个，即 `UsefulServlet`，它被映射成 `/useful` 模式。在大型的框架中，注册指令可以来自 XML 文档，如在 Struts 和 Struts 2 中那样。

本章配套的 `app17b` 应用程序中已经在 `WEB-INF/lib` 目录下包含了一个 `initializer.jar` 文件。为了确认应用程序启动时已经成功地注册了 `UsefulServlet`，可以在浏览器中打开下面这个 URL，看看是否能够看到 Servlet 的输出：

```
http://localhost:8080/app17b/useful
```

不难想象，将来总有一天，所有的框架将都可以部署成一个插件。

### 17.3 小结

本章介绍了动态部署应用程序和插件的两项最新特性。第一项是动态注册，它可以帮助你动态地添加 Servlet、过滤器和监听器，不需要重新启动应用程序。第二项是 Servlet 容器初始化，它可以帮助你部署插件，不需要修改用户应用程序的部署描述符。Servlet 容器初始化程序特别适用于框架的开发者。

# 第 18 章 Struts 2 简介

在第 10 章中我们学习了 Model 2 架构的优势，以及如何基于该模型构建应用程序。本章介绍的 Struts 2 是一种框架，它可以提升 Model 2 应用程序的开发速度。我们首先讨论一下 Struts 2 的优势，以及它是如何提升 Model 2 应用程序开发速度的。本章还将介绍 Struts 2 的基本组件：filter dispatcher、action、result 及 interceptor。

本章的另一个目标是介绍 Struts 2 的配置。大多数 Struts 应用程序都会有一个 `struts.xml` 和一个 `struts.properties` 文件。前者比较重要，因为是在这里配置 action。后者则是可选的，它有一个 `default.properties` 文件，其中包含适用于大多数应用程序的标准设置。本章的最后，将对 `app10b` 应用程序进行改写，以便示范 Struts 2 是如何锐减所需编写的代码量的。

学完本章之后，如果你还有兴趣学习更多关于 Struts 2 的内容，则可以参阅笔者的另一本书：《Struts 2 Design and Programming: A Tutorial》(ISBN 978-0980331608)。

## 18.1 Struts 2 的优势

Struts 2 是一个 MVC 框架，它用一个 filter dispatcher 作为 controller。在编写没有框架的 Model 2 应用程序时，你必须提供一个 controller，并且还要编写 action 类。这个 controller 必须能够完成以下工作：

1. 决定从 URI 处调用哪个 action。
2. 将 action 类实例化。
3. 如果有 action 对象，就要给 action 的属性填入请求参数。
4. 如果有 action 对象，就要调用 action 方法。
5. 将请求跳转到视图中（JSP 页面）。

使用 Struts 的第一个优势在于不需要编写 controller，因此可以集中精力编写 action 类中的业务逻辑。下面列出了 Struts 的一些特性，它们使开发的过程变得更加迅速，更富有乐趣。

- ❑ Struts 提供了一个 filter dispatcher，不需要我们编写了。
- ❑ Struts 用一个基于 XML 的配置文件，将 URI 与 action 进行匹配。由于 XML 文档属于文本文档，因此不需要重新编译就可以对应用程序进行许多变更。
- ❑ Struts 将 action 类实例化，并将用户输入的值填入 action 属性中。如果你没有指定 action 类，将会实例化默认的 action 类。
- ❑ Struts 验证用户输入，并在验证失败时，将用户重定向到输入表单。输入验证是可选

的，可以通过编程或声明的方式来实现。此外，Struts 还提供了内置验证器，其适用于在构建 Web 应用程序时可能遇到的大多数任务。

- Struts 调用 action 方法，并且可以通过配置文件来修改 action 的方法。
- Struts 查看 action 结果，并执行结果。最常见的结果类型：Dispatcher，会将控制权传给一个 JSP 页面。但是，Struts 2 中提供了多种结果类型，允许你以不同的方式来完成各种工作，例如产生一个 PDF，重定向到一个外部资源，发送错误消息，等等。

以上展示了 Struts 2 如何协助你完成在开发第 10 章中的 Model 2 应用程序时遇到的各项任务。它的功能还有很多，例如定制显示数据的标签，数据转换，支持 AJAX，支持国际化和本地化，以及通过插件进行扩展，等等。

## 18.2 Struts 2 工作原理

Struts 2 有一个与 app10b 中类似的 filter dispatcher，它在 Struts 2.3 中的全类名为：org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter。（Struts 2 的早期版本使用的是不同的类）。使用 filter 之前，要先用这个 filter 和 filter-mapping 元素在部署描述符（web.xml 文件）中为它注册一下。

```
<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>
```

在 Model 2 应用程序中，filter dispatcher 必须完成大量的工作，Struts 2 的 filter dispatcher 无疑是个例外。因为 Struts 2 支持更多的特性，它的 filter dispatcher 复杂度可以无限地扩展。但 Struts 2 是将 filter dispatcher 中的任务处理分成许多子组件，称作拦截器。你会发现，第一个拦截器是为 action 对象填入请求参数的。在本章稍后将会介绍更多关于拦截器的内容。

在 Struts 2 应用程序中，是在填充完 action 的属性之后执行 action 方法的。action 方法的名称可以任意取，只要它是有效的 Java 方法名称即可。

action 方法会返回一个 String 值。这个值表明 Struts 2 将把控制权传到哪里。成功的 action 方法执行与失败的方法执行相比，它会将控制权传到一个不同的视图。例如，String “success” 表示一个成功的 action 方法执行，“error” 表示在处理期间有一个错误，并且应该

显示错误信息。大多数时候，会用一个 RequestDispatcher 传给 JSP 页面，但是 JSP 页面并非唯一有效的目的地。如果结果中只是返回一个可供下载的文件，那么它就不需要 JSP 页面。只是发出一条重定向命令，或者发出一个要渲染的图表的结果，也都不需要 JSP 页面。即使 action 需要跳转到某一个视图中，这个视图也不一定非得是 JSP 页面，它也可以使用 Velocity 模板，或者 FreeMarker 模板。

了解了 Struts 2 的所有基础组件之后，接下来要继续讲解 Struts 2 的工作原理。由于 Struts 2 用一个 filter dispatcher 作为 controller，因此所有的活动都将从这个对象开始。

## Velocity 和 FreeMarker

JSP 程序员可能会有这样的疑问：为什么要引入新的视图技术，而不继续使用 JSP 呢？问得好！答案是：虽然你可以用 JSP 解决问题，但是学习 Velocity 和 FreeMarker 还是大势所趋。因为 Velocity 和 FreeMarker 模板可以打包成一个 JAR，这正是发布 Struts 2 插件的方式。但你却不能用 JAR 发布 JSP 页面。当然，如果你很坚持的话，当然也能找到解决的办法，但至少完成得相当不容易。此外，Velocity 的速度还比 JSP 要快得多！

filter dispatcher 的第一个任务是验证请求 URI，并确定要调用哪个 action，以及要实例化哪个 Java action 类。`app10b` 中的 filter dispatcher 是利用一个字符串操作方法来完成这些工作的。但是这个做法很不切实际，因为在开发期间，URI 可能会多次变更，而每当 URI 或其他东西发生变更时，你都不得不重新编译 filter。

为了使 action 类与 URI 匹配，Struts 2 使用了一个名为 `struts.xml` 的配置文件。一般来说，需要创建一个 `struts.xml` 文件，并将它放在 `WEB-INF/classes` 下面。应用程序的所有 action 都要在这个文件中定义。每个 action 都有一个名称，与用来调用 action 的 URI 直接对应。每个 action 声明都可以定义一个 action 类的全类名。你必须指定 action 方法的名称，除非它的名称为 `execute`，这是在没有显式指定名称时，Struts 2 假定的默认方法名称。

一个 action 类至少必须有一个结果告诉 Struts 2 在执行完 action 方法之后它要做些什么。如果 action 方法可以根据情况（如用户输入）返回不同的结果，则可能会有多个结果。

Struts 2 启动时会读取 `struts.xml` 文件。在开发模式下，每当 Struts 2 处理一个请求时，都会查看这个文件的时间戳，如果自最后一次加载以来已经发生了变更，那么将重新加载它。因此，如果你处于开发模式，并且告诉 Struts 2，每当应用程序发生变更时，不需要重启 Web 容器，那么将可以节省很多时间。

如果你没有遵循管理 `struts.xml` 文件的规则，配置文件的加载将会失败。如果发生这种情况，Struts 2 将不会启动，你就必须重启容器。有时候，因为没有明确的错误信息，很难搞清楚是哪里搞错了。如果发生这种情况，要尽量对疑似有问题的 action 进行备注，直到你隔离并修正了导致开发暂停的问题为止。

**提示** 在本章稍后将会在讨论 Struts 配置文件时讲解 Struts 2 开发模式。

图 18-1 展示了 Struts 如何处理 action 调用。不包括读取配置文件，那只在应用程序启动时读取一次罢了。

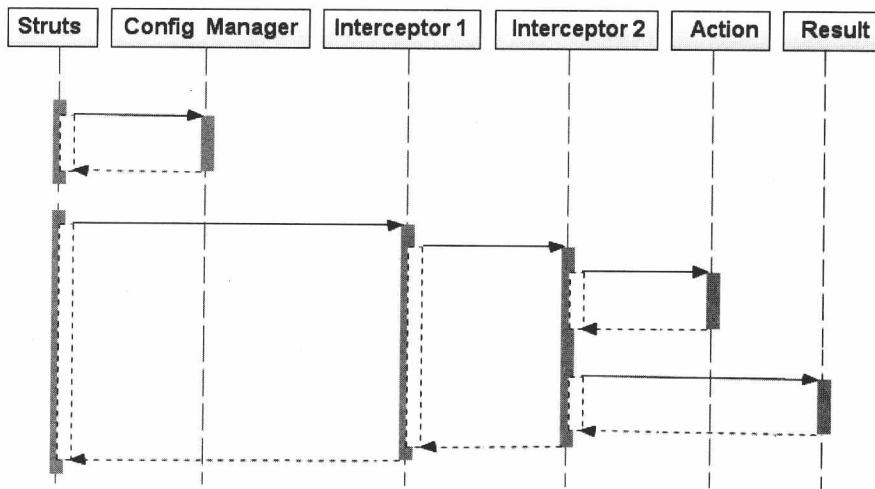


图 18-1 Struts 2 工作原理

对于每一次的 action 调用，filter dispatcher 都要完成以下工作：

1. 询问 Configuration Manager，根据请求 URI 决定要调用哪个 action。
2. 运行为这个 action 注册的每一个拦截器。其中一个拦截器将填入 action 的属性值。
3. 执行 action 方法。
4. 执行结果。

注意，有些拦截器在执行完 action 方法之后，并在执行结果之前，会再次运行。

### 18.3 拦截器

如前所述，filter dispatcher 必须完成很多工作。否则，就必须将 filter dispatcher 类中的代码模块化成拦截器。拦截器的魅力在于，可以通过编辑 Struts 2 配置文件，将它们插入或拔出。事实上，Struts 2 利用这种方法实现了高度的模块化。action 处理的新代码，无须重新编译主框架即可添加。

表 18-1 中列出了 Struts 2 的默认拦截器。“拦截器”一栏中用括号括起来的部分是指在配置文件中用来注册拦截器的名称。没错，你很快就会看到，在使用拦截器之前，必须先在配置文件中注册拦截器。例如，Alias 拦截器的注册名称是 alias。

拦截器有很多，这会给初学者造成一些困扰。关键在于编写 Struts 2 应用程序之前，你

不需要深刻了解拦截器，只需要知道拦截器在 Struts 中起着非常重要的作用即可。

大多数时候，使用默认的拦截器便已足够。但是，如果需要非标准的 action 处理，则可以编写自己的拦截器。编写定制拦截器的内容在《Struts 2 Design and Programming: A Tutorial》一书的第 18 章中有详细的讲解。

表 18-1 Struts 2 的默认拦截器

拦截器	描述
Alias (alias)	转换在各请求之间使用不同名称的类似参数
Chaining (chain)	与 Chain 结果类型一起使用时，这个拦截器会使当前的 action 可以使用前一个 action 的属性
Checkbox (checkbox)	在一个表单中处理复选框，以便发现未经检查的复选框
Cookie (cookie)	在当前 action 中添加一个 cookie
Conversion Error (conversionError)	在 action 的域错误中添加转换错误
Create Session (createSession)	如果当前用户还没有 HttpSession 对象，为它创建一个
Debugging (debugging)	支持调试
Execute and Wait (execAndWait)	在后台执行一个长时间处理的 action，并将用户发送到一个中间等待页面
Exception (exception)	将异常映射成一个结果
File Upload (fileUpload)	支持文件上传
I18n (i18n)	支持国际化和本地化
Logger (logger)	输出 action 名称
Message Store (store)	为其类实现 ValidationAware 的 action 对象保存和获取 action 信息，或 action 错误，或域错误
Model Driven (modelDriven)	为实现 ModelDriven 的 action 类支持模型驱动模式
Scoped Model Driven (scopedModelDriven)	与 Model Driven 拦截器类似，但适用于实现 ScopedModelDriven 的类
Parameters (params)	给 action 的属性填入请求参数
Prepare (prepare)	支持实现 Preparable 接口的 action 类
Scope (scope)	为在 session 或 application 范围中保存 action 状态提供一种机制
Servlet Config (servletConfig)	提供对表示 HttpServletRequest 和 HttpServletResponse 的 Maps 的访问权限
Static Parameters (staticParams)	将静态的属性映射到 action 属性
Roles (roles)	支持基于角色的 action
Timer (timer)	输出执行一个 action 所需的时间
Token (token)	验证是否存在有效 token
Token Session (tokenSession)	验证是否存在有效 token
Validation (validation)	支持输入验证
Workflow (workflow)	在 action 类中调用 validate 方法
Parameter Filter (n/a)	从 action 可用的参数列表中删除参数
Profiling (profiling)	支持 action 剖析

## 18.4 Struts 2 的配置文件

Struts 2 应用程序使用大量的配置文件，最重要的两个是 `struts.xml` 和 `struts.properties`，但是也可以有其他的配置文件。例如，Struts 插件会有一个 `struts-plugin.xml` 配置文件。如果是用 Velocity 作为视图技术，那么还会有 `velocity.properties` 文件。本章将简单介绍 `struts.xml` 和 `struts.properties` 文件。

在 `struts.xml` 中，几乎定义了应用程序的所有方面，包括 `action`、每个 `action` 需要调用的拦截器，以及每个 `action` 的可能结果。

`action` 中使用的拦截器及结果类型在使用之前都必须进行注册。令人高兴的是，Struts 2 配置文件支持继承，并且 `struts2-core-VERSION.jar` 文件中包含了默认的配置文件。`struts-default.xml` 文件是其中一个默认配置文件，它注册默认的结果类型和拦截器。因此，不需要在 `struts.xml` 文件中注册，就可以使用这些默认的结果类型和拦截器，这样该文件更整洁，更简短。

在同一个 JAR 中的 `default.properties` 文件中，包含了适用于所有 Struts 应用程序的设置。因此，除非需要覆盖默认值，否则不需要有 `struts.properties` 文件。

现在来深入探讨 `struts.xml` 和 `struts.properties` 文件。

### 18.4.1 struts.xml 文件

`struts.xml` 文件是一个以 `struts` 为根元素的 XML 文件。我们将在这个文件中定义 Struts 2 应用程序的所有 `action`。下面就是一个 `struts.xml` 文件的主要框架。

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">
<struts>

...
</struts>
```

接下来讨论可能在 `<struts>` 和 `</struts>` 之间出现的比较重要的元素。

#### package 元素

由于 Struts 的设计富有模块性，因此 `action` 都被分组压缩。我们可以将包看成模块。一个典型的 `struts.xml` 文件可以有一个或者多个包：

```
<struts>
<package name="package-1" namespace="namespace-1"
```

```

        extends="struts-default">
<action name="..."/>
<action name="..."/>
    ...
</package>
<package name="package-2" namespace="namespace-2">
    extends="struts-default">
<action name="..."/>
<action name="..."/>
    ...
</package>
    ...
<package name="package-n" namespace="namespace-n">
    extends="struts-default">
<action name="..."/>
<action name="..."/>
    ...
</package>
</struts>
```

**package** 元素必须有 **name** 属性。**namespace** 属性是可选的，如果它不存在，则将使用默认值 “/”。如果 **namespace** 属性有一个非默认的值，那么包中调用 **action** 的 URI 中必须加入命名空间。例如，包中调用 **action** 的 URI 带有默认命名空间时是这样的：

*/context/actionName.action*

为了调用带有非默认命名空间的压缩包中的 **action**，要使用下面这个 URI：

*/context/namespace/actionName.action*

**package** 元素几乎总是继承 **struts-default.xml** 中定义的 **struts-default** 包。这样，包中的所有 **action** 将都可以使用在 **struts-default.xml** 中注册的结果类型和拦截器。下面是 **struts-default** 包的大体框架。为了节省空间，这里面已经删除了拦截器的内容。

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <package name="struts-default">
        <result-types>
            <result-type name="chain" class="com.opensymphony.
                ↲xwork2.ActionChainResult"/>
        <result-type name="dispatcher" class="org.apache.
            ↲struts2.dispatcher.ServletDispatcherResult"
```

```

        default="true"/>
<result-type name="freemarker" class="org.apache.
    ↪struts2.views.freemarker.FreemarkerResult"/>
<result-type name="httpheader" class="org.apache.
    ↪struts2.dispatcher.HttpHeaderResult"/>
<result-type name="redirect" class="org.apache.struts2.
    ↪dispatcher.ServletRedirectResult"/>
<result-type name="redirect-action" class="org.apache.
    ↪struts2.dispatcher.ServletActionRedirectResult"/>
<result-type name="stream" class="org.apache.struts2.
    ↪dispatcher.StreamResult"/>
<result-type name="velocity" class="org.apache.struts2.
    ↪dispatcher.VelocityResult"/>
<result-type name="xslt" class="org.apache.struts2.
    ↪views.xslt.XSLTResult"/>
<result-type name="plaintext" class="org.apache.struts2.
    ↪dispatcher.PlainTextResult"/>
</result-types>

<interceptors>
    [all interceptors]

</interceptors>
</package>
</struts>
```

### include 元素

大型的应用程序中可能会有许多压缩包。为了更易于管理大型应用程序的 struts.xml 文件，最好将它分解成几个小文件，并用 **include** 元素引用这些文件。每个文件最好包含一个包或者几个相关的包。

带有多个 **include** 元素的 struts.xml 文件如下所示：

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
<include file="module-1.xml" />
<include file="module-2.xml" />
...
<include file="module-n.xml" />

</struts>
```

每个 module.xml 文件都有相同的 DOCTYPE 元素和一个 struts 根元素。下面举个例子：

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">

<!-- file module-n.xml -->
<struts>
    <package name="test" extends="struts-default">
        <action name="Test1" class="test.Test1Action">
            <result>/jsp/Result1.jsp</result>
        </action>
        <action name="Test2" class="test.Test2Action">
            <result>/ajax/Result2.jsp</result>
        </action>
    </package>
</struts>

```

### action 元素

action 元素嵌套在 package 元素中，它表示一个 action。action 必须有名称，并且可以给它选择任意名称。一个好的名称可以反映出这个 action 的功能。例如，负责显示输入产品信息表单的 action 可以命名为 displayAddProductForm。按照规范，建议最好使用名称和动词的组合。例如，不要将一个 action 命名为 displayAddProductForm，而是最好命名为 Product\_input。但是总的来说，最终怎么命名还是由你来决定。

action 可以指定或不指定 action 类。因此，action 元素也可以像下面这么简单：

```
<action name="MyAction">
```

没有指定 action 类的 action 将会被提供一个默认 action 类的实例。ActionSupport 类就是默认的 action 类。

但是，如果一个 action 具有非默认的 action 类，则必须利用 class 属性指定全类名。此外，还必须指定 action 方法的名称，这是在调用 action 时，action 类中要被执行的方法。下面举个例子：

```
<action name="Address_save" class="app.Address" method="save">
```

如果有 class 属性，但是没有 method 属性，那么将会假定以 execute 作为方法名称。换句话说，下面这两个 action 元素其实是一回事：

```
<action name="Employee_save" class="app.Employee" method="execute">
<action name="Employee_save" class="app.Employee">
```

### result 元素

<result> 是 <action> 的子元素，它告诉 Struts 2 要将这个 action 转发到哪里去。result 元素对应 action 方法的返回值。由于一个 action 方法可以根据不同的情况返回不同的值，因此一个 action 元素可能会有多个 result 元素，每个元素对应 action 方法的一个可能返回值。也就是说，如果一个方法可能返回 success 和 input，就必须有两个 result 元素。result 元素的 name 属性用一个方法返回值映射一个结果。

---

**提示** 如果一个方法返回一个没有匹配 result 元素的值，Struts 将会尽量在 global-results 元素（关于这个元素的讨论请查看接下来的内容）下查找一个匹配的结果。如果在 global-results 下没有找到对应的 result 元素，将会抛出异常。

---

例如，下面的 action 元素中包含了两个 result 元素：

```
<action name="Product_save" class="app.Product" method="save">
    <result name="success" type="dispatcher">
        /jsp/Confirm.jsp
    </result>
    <result name="input" type="dispatcher">
        /jsp/Product.jsp
    </result>
</action>
```

如果 action 方法 save 返回 success，将执行第一个结果，这时候会显示 Confirm.jsp 页面。如果该方法返回 input，这时候浏览器中将会显示 Product.jsp 页面。

顺便说一下，result 元素的 type 属性是用来指定结果类型的。type 属性值必须是在外围包或外围包继承的父包中注册过的一个结果类型。假如 action Product\_save 是在继承 struts-default 的包中，那么就可以放心地给这个 action 使用一个 Dispatcher 结果，因为在 struts-default 中定义了 Dispatcher 结果类型。

如果 result 元素中省略了 name 属性，将默认采用 success。此外，如果没有 type 属性，将采用默认的结果类型 Dispatcher。因此，下面这两个 result 元素其实是一样的：

```
<result name="success" type="dispatcher">/jsp/Confirm.jsp</result>
<result>/jsp/Confirm.jsp</result>
```

Dispatcher result 元素还有另外一种使用 param 元素的语法，此时 param 元素的参数名称要用 location。换句话说，这个 result 元素：

```
<result>/test.jsp</result>
```

与下面这个是一样的：

```
<result>
    <param name="location">/test.jsp</param>
</result>
```

在本节稍后还会学到更多关于 `param` 元素的内容。

### global-results 元素

`package` 元素中可以包含一个 `global-results` 元素，后者中包含了作为通用结果的结果。假如一个 `action` 无法在其 `action` 声明下方找到一个匹配的结果，如有可能，那么它将会查找 `global-results` 元素。

下面就是一个 `global-results` 元素的例子：

```
<global-results>
    <result name="error">/jsp/GenericErrorPage.jsp</result>
    <result name="login" type="redirect-action">Login</result>
</global-results>
```

### 与 interceptor 相关的元素

`struts.xml` 文件中可能出现 5 个与拦截器相关的元素：`interceptors`、`interceptor`、`interceptor-ref`、`interceptor-stack` 及 `default-interceptor-ref`。这些都将在本节中进行讨论。

`action` 元素中必须包含处理 `action` 对象的一系列拦截器。但在使用拦截器之前，必须利用 `<interceptors>` 下的 `interceptor` 元素为它注册一下。在一个包中定义的拦截器可以被这个包中的所有 `action` 使用。

例如，下面的 `package` 元素中注册了两个拦截器：`validation` 和 `logger`：

```
<package name="main" extends="struts-default">
    <interceptors>
        <interceptor name="validation" class="..."/>
        <interceptor name="logger" class="..."/>
    </interceptors>
</package>
```

为了对一个 `action` 使用拦截器，需使用该 `action` 的 `action` 元素下的 `interceptor-ref` 元素。例如，以下配置中注册了 4 个拦截器，并将它们应用到了 `Product_delete` 和 `Product_save` `action`：

```
<package name="main" extends="struts-default">
    <interceptors>
        <interceptor name="alias" class="..."/>
        <interceptor name="i18n" class="..."/>
        <interceptor name="validation" class="..."/>
        <interceptor name="logger" class="..."/>
    </interceptors>
```

```

<action name="Product_delete" class="...">
    <interceptor-ref name="alias"/>
    <interceptor-ref name="i18n"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="logger"/>
    <result>/jsp/main.jsp</result>
</action>

<action name="Product_save" class="...">
    <interceptor-ref name="alias"/>
    <interceptor-ref name="i18n"/>
    <interceptor-ref name="validation"/>
    <interceptor-ref name="logger"/>
    <result name="input">/jsp/Product.jsp</result>
    <result>/jsp/ProductDetails.jsp</result>
</action>
</package>

```

有了这些设置之后，每当调用 `Product_delete` 或 `Product_save` action 时，这 4 个拦截器就有机会大展身手了。注意，`interceptor-ref` 元素的出现顺序很重要，因为它确定了为这个 action 调用已注册拦截器的顺序。在本例中，将先调用 `alias` 拦截器，接着是 `i18n` 拦截器、`validation` 拦截器，最后是 `logger` 拦截器。

大多数 Struts 2 应用程序都有多个 `action` 元素，每个 `action` 都要重复编写一个拦截器列表，这真是一项很无趣的工作。为了解决这个问题，Struts 允许我们创建拦截器堆栈，将必要的拦截器进行分组。为了从每个 `action` 元素中引用拦截器，引用拦截器堆栈即可。

例如，通常是在下列顺序使用这 6 个拦截器：`exception`、`servletConfig`、`prepare`、`checkbox`、`params` 及 `conversionError`。为了避免在 `action` 声明中一遍又一遍地引用这些拦截器，可以像下面这样创建一个拦截器堆栈：

```

<interceptor-stack name="basicStack">
    <interceptor-ref name="exception"/>
    <interceptor-ref name="servlet-config"/>
    <interceptor-ref name="prepare"/>
    <interceptor-ref name="checkbox"/>
    <interceptor-ref name="params"/>
    <interceptor-ref name="conversionError"/>
</interceptor-stack>

```

要使用这些拦截器，只需要引用这个堆栈即可：

```

<action name="..." class="...">
    <interceptor-ref name="basicStack"/>
    <result name="input">/jsp/Product.jsp</result>
    <result>/jsp/ProductDetails.jsp</result>
</action>

```

**struts-default** 包中定义了几个堆栈。此外，它还定义了一个 **default-interceptor-ref** 元素，用它指定默认的拦截器或拦截器堆栈，以便当 action 中没有定义拦截器时可以使用：

```
<default-interceptor-ref name="defaultStack"/>
```

如果一个 action 需要结合使用其他拦截器和默认堆栈，则必须重新定义默认堆栈，因为如果在一个 action 元素中能够找到 interceptor 元素，那么 **default-interceptor-ref** 元素将会被忽略。

### param 元素

**param** 元素可以嵌套在另一个元素中，例如 **action**、**result-type** 或 **interceptor**，以便将一个值传给包装对象。

**param** 元素用一个 **name** 属性定义参数的名称，其格式如下：

```
<param name="property">value</param>
```

在 **action** 元素内部使用时，**param** 可以用来设置 **action** 属性。例如，下面的 **param** 元素设置了 **action** 的 **sitId** 属性：

```
<action name="customer" class="...">
    <param name="siteId">california01</param>
</action>
```

下面的 **param** 元素则设置了验证 **interceptor-ref** 的 **excludeMethod**：

```
<interceptor-ref name="validation">
    <param name="excludeMethods">input,back,cancel</param>
</interceptor-ref>
```

利用 **excludeMethods** 参数可以在调用包装拦截器时排除某些特定的方法。

### constant 元素

除了 **struts.xml** 文件之外，还可能有一个 **struts.properties** 文件。如果需要覆盖 **default.properties** 文件（在 **struts2-core-VERSION.jar** 文件中）中定义的一个或多个键 / 值对，则需要创建后者。在大多数时候，我们都不要 **struts.properties** 文件，因为 **default.properties** 文件已经够好了。此外，还可以利用 **struts.xml** 文件中的 **constant** 元素覆盖 **default.properties** 文件中的设置。

**constant** 元素有一个 **name** 属性和一个 **value** 属性。例如，**struts.devMode** 设置定义了 Struts 应用程序是否处于开发模式。在默认情况下，这个值为 **false**，表示应用程序不是处于开发模式。

下面的 constant 元素将 struts.devMode 值设为 true:

```
<struts>
    <constant name="struts.devMode" value="true"/>

    ...
</struts>
```

## 18.4.2 struts.properties 文件

如果需要覆盖 default.properties 文件中的设置，需创建一个 struts.properties 文件。例如，下列 struts.properties 文件覆盖了 default.properties 中的 struts.devMode 值：

```
struts.devMode = true
```

struts.properties 文件必须放在 classpath 下，或者在 WEB-INF/classes 中。

为了避免创建新的文件，可以在 struts.xml 文件中使用 constant 元素。另一种方法是在 Struts filter dispatcher 的 filter 声明中使用 init-param 元素：

```
<filter>
    <filter-name>struts</filter-name>
    <filter-class>
        org.apache.struts2.dispatcher.FilterDispatcher
    </filter-class>
    <init-param>
        <param-name>struts.devMode</param-name>
        <param-value>true</param-value>
    </init-param>
</filter>
```

## 18.5 简单的 Struts 应用程序

现在我们要用 Struts 2 改写 app10b，并将这个新的范例应用程序命名为 app18a。我们将使用类似的 JSP 页面，以及一个名为 Product 的 action 类。

app18a 的目录结构如图 18-2 所示。应用程序中的每一个组件都将在接下来的小节中进行讨论。

### 18.5.1 开发描述符和 Struts 配置文件

开发描述符如代码清单 18-1 所示，Struts 2 配置文件如代码清单 18-2 所示。

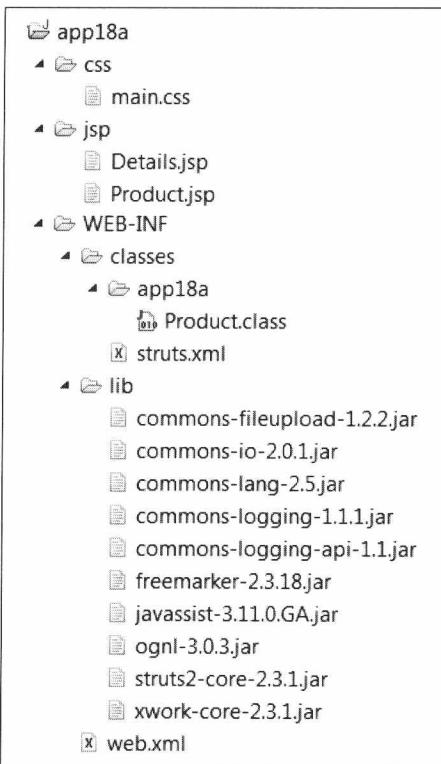


图 18-2 app18a 的目录结构

## 代码清单 18-1 开发描述符 (web.xml 文件)

---

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app 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_3_0.xsd"
  version="3.0"
>
  <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>

```

```

<!-- Restrict direct access to JSPs.
     For the security constraint to work, the auth-constraint
     and login-config elements must be present -->
<security-constraint>
    <web-resource-collection>
        <web-resource-name>JSPs</web-resource-name>
        <url-pattern>/jsp/*</url-pattern>
    </web-resource-collection>
    <auth-constraint/>
</security-constraint>

<login-config>
    <auth-method>BASIC</auth-method>
</login-config>
</web-app>

```

代码清单 18-2 struts.xml

```

<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
    <package name="app18a" namespace="/" extends="struts-default">
        <action name="Product_input">
            <result>/jsp/ProductForm.jsp</result>
        </action>

        <action name="Product_save" class="app18a.Product">
            <result>/jsp/ProductDetails.jsp</result>
        </action>
    </package>
</struts>

```

struts.xml 文件定义了一个带有两个 action 的包 (app18a)：Product\_input 和 Product\_save。Product\_input action 没有 action 类。调用 Product\_input 只是将控制权传给 ProductForm.jsp 页面。这个页面中包含了一个用来输入产品信息的入口。

Product\_save action 中有一个非默认的 action 类 (app18.Product)。由于 action 声明中没有 method 属性，因此将会调用 Product 类中的 execute 方法。

**提示** 在开发期间，可以在包的前面添加这个 constant 元素，将 Struts 2 切换到开发模式：

```
<constant name="struts.devMode" value="true" />
```

### 18.5.2 Action 类

代码清单 18-3 中的 Product 类是 Product\_save action 的 action 类。这个类有 3 个属性 (productName、description 和 price) 和一个 action 方法： execute。

代码清单 18-3 action 类 Product

---

```
package app18a;
import java.io.Serializable;

public class Product implements Serializable {
    private static final long serialVersionUID = 1000L;

    private String name;
    private String description;
    private String price;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    public String getPrice() {
        return price;
    }

    public void setPrice(String price) {
        this.price = price;
    }

    public String execute() {
        return "success";
    }
}
```

---

### 18.5.3 运行应用程序

app18a 应用程序是将第 10 章中的 app10b 应用程序用 Struts 2 改写之后的版本。为了调用第一个 action，可以使用下面的 URL：

[http://localhost:8080/app18a/Product\\_input.action](http://localhost:8080/app18a/Product_input.action)

你将会在浏览器中看到如图 10-2 所示的内容。在域中输入值，并提交表单，浏览器上就会显示出一条如图 10-3 所示的确认信息。

Congratulations. You've just seen Struts 2 in action!

(恭喜！你已经亲身体验到了 Struts 2 的实战效果！)

## 18.6 小结

本章学习了 Struts 2 是通过哪些特性来提升 Model 2 应用程序的开发速度的。还学习了如何配置 Struts 应用程序，并且成功地编写了自己的第一个 Struts 应用程序。

# 附录 A Tomcat

Tomcat 是当今最流行的 Servlet/JSP 容器。它免费、成熟，并且是开源的。为了运行本书配套的范例应用程序，需要 Tomcat 7 或者其他兼容的 Servlet/JSP 容器。本附录介绍了一个快速的安装和配置向导，但不是一个全面的综合教程。

## A.1 下载和配置 Tomcat

首先，要从 <http://tomcat.apache.org> 下载最新版本的 Tomcat，并且应该是 zip 或 gz 格式的最新版二进制发行包。Tomcat 7 要用 Java 6 才能运行。

下载好 zip 或 gz 文件之后，要解压文件，然后就可以在安装目录下看到几个目录。

在 bin 目录下，可以看到几个用来启动和停止 Tomcat 的程序。webapps 目录很重要，因为可以在那里部署应用程序。此外，conf 目录下还包含了配置文件，包括 server.xml 和 tomcat-users.xml 文件。lib 目录也很重要，因为它包含了编译 Servlet 和定制标签所需的 Servlet 和 JSP 的 API。

解压好 zip 或 gz 文件之后，将 JAVA\_HOME 环境变量设置成 JDK 安装目录。

对于 Windows 用户，为了更便于安装，最好下载 Windows 安装程序。

## A.2 启动和停止 Tomcat

一旦下载并解压好 Tomcat 二进制发行包，就可以通过运行 startup.bat (Windows) 或 startup.sh 文件 (Unix/Linux/Mac OS) 来启动 Tomcat 了。这两个文件都放在 Tomcat 安装目录的 bin 目录下。在默认情况下，Tomcat 是在端口 8080 上运行的，因此可以在浏览器中打开下面这个地址来测试一下 Tomcat：

<http://localhost:8080>

要停止 Tomcat 时，运行 bin 目录下的 shutdown.bat (Windows) 或 shutdown.sh 文件 (Unix/Linux/Mac OS)。

## A.3 定义 Context

为了将一个 Servlet/JSP 应用程序部署到 Tomcat 中，需要显式或隐式地定义一个 Tomcat context。每个 Tomcat context 都表示 Tomcat 中的一个 Web 应用程序。

显式定义 Tomcat context 有几种方式：

- 在 Tomcat 的 conf/Catalina/localhost 目录下创建一个 XML 文件。
- 在 Tomcat 的 conf/server.xml 文件中添加一个 Context 元素。

如果决定为每个 context 都创建一个 XML 文件，那么文件名就很重要，因为是从文件名衍生出 context 路径的。例如，如果将一个 commerce.xml 文件放在 conf/Catalina/localhost 目录下，那么应用程序的 context 路径就是 commerce，利用下面这个 URL 就可以调用一个资源：

```
http://localhost:8080/commerce/resourceName
```

context 文件中必须包含一个 Context 元素作为其根元素。这个元素大多数时候都没有子元素，并且它经常是文件中唯一的元素。例如，下面就是一个范例 context 文件，里面只有一行代码：

```
<Context docBase="C:/apps/commerce" reloadable="true"/>
```

它唯一必要的属性是 docBase，用来指定应用程序的位置。reloadable 属性是可选的，但是如果存在，并且它的值设为 true，Tomcat 就会对应用程序进行侦测，包括 Java 类文件和其他资源的添加、删除或者更新，等等。每当发现这种变更时，Tomcat 就会重新载入应用程序。在开发期间，建议将 reloadable 设为 true，但在正式生产环境中则不建议这么做。

在将一个 context 文件添加到指定目录下时，Tomcat 会自动加载应用程序。当你删除 context 文件时，Tomcat 会卸载应用程序。

定义 context 的另一种方式是在 conf/server.xml 文件中添加一个 Context 元素。具体做法是，打开文件，并在 Host 元素下创建一个 Context 元素。与前一个方法不同的是，在这里定义 context 需要为 context 路径指定 path 属性。下面举个例子：

```
<Host name="localhost" appBase="webapps" unpackWARs="true"
      autoDeploy="true">

    <Context path="/commerce"
              docBase="C:/apps/commerce"
              reloadable="true"
    />
</Host>
```

一般来说，不建议通过 server.xml 来管理 context，因为只有在重启 Tomcat 之后，所做的更新才会生效。但是，如果有大量应用程序需要进行快速测试，比如正在练习编写 Servlet

和 JSP 页面的时候，可能就会觉得使用 `server.xml` 会比较理想，因为可以在一个文件中管理所有的应用程序。

最后，还可以通过将一个 war 文件或者整个应用程序复制到 Tomcat 的 `webapps` 目录下，来隐式地部署应用程序。

关于 Tomcat context 的更多信息，可以在下面的网站中找到：

<http://tomcat.apache.org/tomcat-7.0-doc/config/context.html>

## A.4 定义资源

定义一个 JNDI 资源，供应用程序在 Tomcat context 定义中使用。资源用 `Context` 元素下的 `Resource` 元素表示。

例如，为了添加一个打开 MySQL 数据库连接的 `DataSource` 资源，需添加这个 `Resource` 元素：

```
<Context [path="/appName"] docBase="...">
  <Resource name="jdbc/dataSourceName"
    auth="Container"
    type="javax.sql.DataSource"
    username="..."
    password="..."
    driverClassName="com.mysql.jdbc.Driver"
    url="...">
  />
</Context>
```

关于 `Resource` 元素的更多信息，可以在下列网站中找到：

<http://tomcat.apache.org/tomcat-7.0-doc/jndi-resources-howto.html>

## A.5 安装 SSL 证书

Tomcat 支持 SSL，应该利用它来保护机密数据的传输，例如社会安全号码和信用卡资料等。你可以利用 KeyTool 程序生成一个公 / 私密钥对，并花钱请一家可信任的机构为你创建并签发一份数字证书。生成密钥对以及签发的过程将在附录 C 中进行讨论。

一旦你收到证书（Certificate），并且将它导入密钥存储库（keystore）之后，下一步就是将它安装到服务器上。如果使用的是 Tomcat，则只需将密钥存储库复制到服务器中的某个位置，并配置 Tomcat 即可。然后打开 `conf/server.xml` 文件，并将以下 `Connector` 元素添加到 `<service>` 下方：

```
<Connector port="443"
  minSpareThreads="5"
```

```
maxSpareThreads="75"
enableLookups="true"
disableUploadTimeout="true"
acceptCount="100"
maxThreads="200"

scheme="https"
secure="true"
SSLEnabled="true"
keystoreFile="/path/to/keystore"
keyAlias="example.com"
keystorePass="01secret02%%%"
clientAuth="false"
sslProtocol="TLS"
/>>
```

粗体部分就是与 SSL 有关的内容。

## 附录 B Web 注解

Servlet 3 在 `javax.servlet.annotation` 包中提供了一组注解类型，用来标注像 Servlet、Filter 和 Listener 这类 Web 对象。本附录将详细讲解这些注解类型。

### B.1 HandlesTypes

这个注解类型用来声明 `ServletContainerInitializer` 可以处理哪些类型的类。它有一个属性、一个值，用来声明类的类型。例如，下面的 `ServletContainerInitializer` 用 `@HandlesTypes` 进行标注，声明初始化程序可以处理 `UsefulServlet`。

```
@HandlesTypes({UsefulServlet.class})
public class MyInitializer implements ServletContainerInitializer {
    ...
}
```

### B.2 HttpConstraint

`HttpConstraint` 注解类型表示适用于没有对应 `HttpMethodConstraint` 元素的所有 HTTP 协议方法的安全约束。这个注解类型必须放在 `ServletSecurity` 注解中。

`HttpConstraint` 的属性如表 B-1 所示。

表 B-1 `HttpConstraint` 属性

属性	描述
<code>rolesAllowed</code>	表示授权角色的一个字符串数组
<code>transportGuarantee</code>	表示是否有必须满足的数据保护要求，有效值为 <code>ServletSecurity.TransportGuarantee</code> 枚举的成员之一（ <code>CONFIDENTIAL</code> 或者 <code>NONE</code> ）
<code>value</code>	默认的授权语义

例如，下列 `HttpConstraint` 标注声明被标注的 Servlet 只能由属于 `manager` 角色的用户进行访问。由于没有 `HttpMethodConstraint` 标注，这条约束将应用于所有的 HTTP 方法。

```
@ServletSecurity(@HttpConstraint(rolesAllowed = "manager"))
```

### B.3 HttpMethodConstraint

这个注解类型表示特定 HTTP 方法中的一个安全约束。`HttpMethodConstraint` 标注只能

出现在 `ServletSecurity` 注解中。

`HttpMethodConstraint` 的属性如表 B-2 所示。

表 B-2 `HttpMethodConstraint` 属性

属性	描述
<code>emptyRoleSemantic</code>	这是默认的授权语义，这个值必须是 <code>ServletSecurity.EmptyRoleSemantic</code> 枚举的成员之一（ <code>DENY</code> 或 <code>PERMIT</code> ）
<code>rolesAllowed</code>	表示授权角色的一个字符串数组
<code>transportGuarantee</code>	表示是否有必须满足的数据保护要求，它的有效值为 <code>ServletSecurity.TransportGuarantee</code> 枚举的成员之一（ <code>CONFIDENTIAL</code> 或者 <code>NONE</code> ）
<code>value</code>	受影响的 HTTP 方法

例如，下列 `ServletSecurity` 注解使用了 `value` 和 `httpMethodConstraints` 这两个属性。`HttpConstraint` 注解定义了可以访问被标注 Servlet 的角色，`HttpMethodConstraint` 注解（没有 `rolesAllowed` 属性）覆盖了 Get 方法的约束。因此，任何用户都可以通过 Get 访问这个 Servlet。另一方面，通过其他所有 HTTP 方法进行访问的权限则仅限于 `manager` 角色的用户。

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "manager"),
    httpMethodConstraints = {@HttpMethodConstraint("GET")})
```

但是，如果 `HttpMethodConstraint` 标注类型的 `emptyRoleSemantic` 属性值为 `EmptyRoleSemantic.DENY`，那么该方法将限制所有用户的访问。例如，用以下 `ServletSecurity` 注解进行标注的 Servlet 将阻止通过 Get 方法进行访问，但是允许 `member` 角色中的所有用户通过其他 HTTP 方法进行访问。

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "member"),
    httpMethodConstraints = {@HttpMethodConstraint(value = "GET",
        emptyRoleSemantic = EmptyRoleSemantic.DENY)})
```

## B.4 MultipartConfig

`MultipartConfig` 注解类型用来标注一个 Servlet，表示这个 Servlet 的实例是否能够处理 multipart/form-data MIME 类型，一般在上传文件时会用到。

表 B-3 列出了 `MultipartConfig` 的属性。

例如，下列 `MultipartConfig` 标注规定，可以上传的最大文件容量是一百万个字节。

```
@MultipartConfig(maxFileSize = 1000000)
```

表 B-3 MultipartConfig 属性

属性	描述
fileSizeThreshold	超过这个容量界限之后，所上传的文件将被写入磁盘
location	上传的文件被存入磁盘时的保存位置
maxFileSize	上传文件的最大容量。大于指定值的文件会被拒绝。 <code>maxFileSize</code> 的默认值为 -1，表示不受限制
maxRequestSize	表示允许 multipart HTTP 请求的最大容量，默认值为 -1，表示不受限制

## B.5 ServletSecurity

`ServletSecurity` 注解类型用来标注 Servlet 类，并对这个 Servlet 上应用安全约束。在 `ServletSecurity` 标注中可能出现的属性如表 B-4 所示。

表 B-4 ServletSecurity 属性

属性	描述
httpMethodConstrains	指定 HTTP 方法特定约束的一组 <code>HttpMethodConstraint</code>
value	<code>HttpConstraint</code> 注解定义了应用于没有相应 <code>HttpMethodConstraint</code> 的所有 HTTP 方法的安全约束

例如，以下 `ServletSecurity` 注解中包含了一个 `HttpConstraint` 标注，表示被标注的 Servlet 只能由 manager 角色中的用户访问。

```
@ServletSecurity(value = @HttpConstraint(rolesAllowed = "manager"))
```

## B.6 WebFilter

`WebFilter` 注解类型用来标注一个过滤器。表 B-5 展示了可以在 `WebFilter` 注解中出现的属性。这些属性都是可选的。

表 B-5 WebFilter 属性

属性	描述
asyncSupported	表明该过滤器是否支持异步处理
description	该过滤器的描述
dispatcherTypes	该过滤器应用到的一组 <code>DispatcherTypes</code>
displayName	该过滤器的显示名称
filterName	该过滤器的名称
initParams	该过滤器的初始参数
largeIcon	该过滤器的大图标
servletNames	该过滤器应用到的几个 Servlet 的名称
smallIcon	该过滤器的小图标

(续)

属性	描述
urlPatterns	该过滤器应用到的 URL 模式
value	该过滤器应用到的 URL 模式

## B.7 WebInitParam

这个注解类型用于给 Servlet 或 Filter 传递初始化参数。WebInitParam 注解中可能出现的属性如表 B-6 所示。属性名称右侧有星号的表示是必要的属性。

表 B-6 WebInitParam 属性

属性	描述
description	初始化参数的描述
name*	初始化参数的名称
value*	初始化参数的值

## B.8 WebListener

这个注解类型用来给监听器进行标注。它的唯一属性 value 是可选的，其中包含这个监听器的描述。

## B.9 WebServlet

这个注解类型用来标注 Servlet，它的属性如表 B-7 所示。这些属性全部都是可选的。

表 B-7 WebServlet 属性

属性	描述
asyncSupported	表明该 Servlet 是否支持异步处理
description	该 Servlet 的描述
displayName	该 Servlet 的显示名称
initParams	该 Servlet 的初始参数
largeIcon	该 Servlet 的大图标
loadOnStartup	在一个包含多个 Servlet 的应用程序中，各 Servlet 的加载顺序
name	该 Servlet 的名称
smallIcon	该 Servlet 的小图标
urlPatterns	调用该 Servlet 的 URL 模式
value	调用该 Servlet 的 URL 模式

# 附录 C SSL 证书

SSL 证书是保护互联网上的通信安全以及保持数据安全的一种工具。有一种普遍的误解认为，只有电子商务网站和网上银行必须使用 SSL 证书。事实上，大多数使用某些登录页面的网站也都应该使用 SSL 证书，以便用户输入的密码不会被当成普通文本进行传输。

在本附录中，将学习如何利用 KeyTool 程序生成公 / 私钥对，以及如何请可信任的授权机构将公钥签署成一份数字证书。关于在 Tomcat 中安装 SSL 证书的相关信息请参阅附录 A。

## C.1 证书概述

SSL 基于对称和不对称这两种加密方法。后者涉及了一对密钥，即一个公钥，一个私钥，这个内容在第 15 章中已经学过。

公钥通常包在证书中，因为证书是发布公钥的一种更可信任的方式。证书是利用与证书中包含的公钥相对应的私钥来签署的，这叫自签证书。换句话说，自签证书的签署者与颁发者是相同的。

如果人们已经认识发送者，那么用自签证书来验证签署文档的发送者是可以的。但是为了得到更广泛的接受，则需要由数字证书认证中心（Certificate Authority, CA）签署，如 VeriSign 和 Thawte。你需要将自签证书发给他们。

CA 对你完成验证之后，就会给你颁发一份证书，它代替了自签证书。这份新证书也可以是一个证书链。在这个证书链的最上方是“根”，就是指自签证书。接下来是 CA 对你进行验证的证书。如果这家 CA 名气不够大，他们会将它发给更大的 CA，更大的 CA 会验证第一家 CA 的公钥。最后一家 CA 也会发出证书，因此形成了一个证书链。这个更大的 CA 通常会广泛发布他们的公钥，因此人们可以很容易验证他们签署的证书。

Java 提供了一组可以用于处理前面讲过的不对称加密法的工具和 API，用它们可以完成以下工作：

- 生成公钥和私钥对，然后你就可以将生成的公钥发送给证书颁发者，以便获得你自己的证书，当然，这是要收费的。
- 将私钥和公钥保存在一个称作密钥存储库（keystore）的数据库中。密钥存储库有名称，并且有密码保护。
- 将其他人的证书保存在同一个密钥存储库中。
- 通过用自己的私钥进行签署来创建自己的证书。但是，这种证书的用途有限。如果用

作测试，那么使用自签证书就可以了。

- 对一个文件进行数字签名。这个特别重要，因为如果 Applets 是保存在一个签过名的 jar 文件中，那么浏览器将只允许 Applets 访问资源。签过名的 Java 代码可以向用户保证，你真的是这个类的开发者。如果他们信任你，运行这个 Java 类时就不会担心了。现在我们来看一下这个工具。

## C.2 KeyTool 程序

KeyTool 程序是创建和维护公钥和私钥及证书的工具。它在 JDK 中，位于 JDK 的 bin 目录下。Keytool 是一个命令行程序。要想查看它的正确语法，只要在命令窗口中输入 keytool 即可。接下来针对一些重要的功能举几个例子。

### C.2.1 生成密钥对

首先，要了解在 Java 中生成密钥的几个关键点：

1. Keytool 生成一个公钥 / 私钥对，并利用这个私钥创建一份证书（自签证书）。其中，证书包含了公钥及实体的身份信息，证明这是属于谁的密钥。因此，你需要提供名称和其他信息。这个名称称作识别名称（Distinguished Name, DN），包含以下信息：

```
CN= common name, e.g. Joe Sample
OU=organizational unit, e.g. Information Technology
O=organization name, e.g. Brainy Software Corp
L=locality name, e.g. Vancouver
S=state name, e.g. BC
C=country, (two letter country code) e.g. CA
```

2. 密钥将保存在一个称作密钥存储库（keystore）的数据库中。密钥存储库是基于文件并有密码保护的，因此任何未被授权的用户都无法访问保存在其中的私钥。

3. 如果在生成密钥或执行其他功能时没有指定密钥存储库，将使用默认的密钥存储库。在用户的 home 目录下（即 user.home 系统属性定义的目录），默认密钥存储库命名为 .keystore。例如，在 Windows 平台上，Windows XP 的默认密钥存储库是放在 C:\Documents and Settings\userName 目录下。

4. 密钥存储库中有两种条目：

- a. 密钥条目，每个条目都是一个私钥，并配有对应公钥的证书链。
- b. 可信任的证书条目，每个条目中都包含你所信任的实体的公钥。

每个条目也都是有密码保护的，因此有两种密码，一个保护密钥存储库，一个保护条目。

5. 密钥存储库中的每个条目都用一个唯一的名称或者别名表示。你在用 keytool 生成密钥对或做其他事情的时候必须指定一个别名。

6. 如果在生成密钥对时没有指定别名，将默认使用 mykey 作为别名。

生成密钥对的最简短命令是：

```
keytool -genkeypair
```

使用这个命令时，将使用默认的密钥存储库；如果用户的 home 目录下没有密钥存储库，则会创建一个。生成的密钥以 mykey 作为别名。然后你会被要求输入密钥存储库的密码，并提供识别名称的信息。最后，会有弹出窗口要求你输入条目的密码。

再次调用 keytool -genkeypair 会产生一个错误，因为它会试图创建一对密钥，并且再次使用 mykey 作为别名。

指定别名时，可以使用 -alias 参数。例如，以下命令创建了以关键字 email 标识的一个密钥对：

```
keytool -genkeypair -alias email
```

还是使用默认的密钥存储库。

指定密钥存储库时，要使用 -keystore 参数。例如，下列命令生成了一个密钥对，并将它保存在 C:\javakeys 目录下名为 myKeystore 的密钥存储库文件中：

```
keytool -genkeypair -keystore C:\javakeys\myKeyStore
```

调用完程序之后，会要求你输入任务信息。

生成密钥对的完整命令中要使用 genkeypair、alias、keypass、storepass 以及 dname 参数。

例如：

```
keytool -genkeypair -alias email4 -keypass myPassword -dname
"CN=JoeSample, OU=IT, O=Brain Software Corp, L=Surrey, S=BC, C=CA"
-storepass myPassword
```

## C.2.2 认证

当你可以利用 Keytool 生成公钥 / 私钥对和自签证书时，这个证书将只被已经认识你的人所信任。为了得到更广泛的认可，要由数字证书认证中心（CA）为你的证书进行签署，例如 VeriSign、Entrust 或者 Thawte。

如果想要这么做，需要利用 Keytool 的 -certreq 参数生成一个 CSR (Certificate Signing Request，证书签署请求)，其语法如下：

```
keytool -certreq -alias alias -file certregFile
```

这个命令的输入值就是 alias 所引用的证书，其输出是一个 CSR，这是一个由 certregFile

指定路径的文件。将 CSR 发送给 CA，他们会离线对你进行验证，一般会请你提供有效的身份证明资料，例如护照或者驾照复印件。

如果 CA 认可你的身份资料，就会给你发一份新的证书，或者是一个包含你公钥的证书链。这个新证书代替了目前的证书链（自签证书）。一旦你收到回复，就可以利用 Keytool 的 importcert 参数将新证书导入到一个密钥存储库中。

### C.2.3 将证书导入密钥存储库

如果你从第三方收到一个签过名的文档，或者收到 CA 的回复，就可以将它保存在密钥存储库中，并且需要给它起一个容易记住这份证书的别名。

要将证书存入或导入密钥存储库中，需使用 importcert 参数，其语法如下：

```
keytool -importcert -alias anAlias -file filename
```

例如，要将 joeCertificate.cer 中的证书导入密钥存储库，并取别名为 brotherJoe 时，要使用下面这个命令：

```
keytool -importcert -alias brotherJoe -file joeCertificate.cer
```

将证书保存在密钥存储库中有两个好处。第一，有一个受密码保护的集中保存的地方。第二，如果你已经将第三方的证书保存在密钥存储库中，就可以轻松地验证来自他们的签名文档。

### C.2.4 从密钥存储库中导出证书

利用私钥可以给文档进行签名。在给文档签名时，要给文档建立一份摘要信息，然后用私钥对摘要信息进行加密。然后发布文档以及加过密的摘要。

其他人要验证文档，他们必须有你的公钥。为了安全起见，公钥也需要进行签名。你可以自签，也可以让可信任的证书颁发商为它签名。

首先，是从密钥存储库中导出证书，并将它保存成一个文件。然后轻松地发布这个文件。为了从密钥存储库中提取证书，需要使用 -exportcert 参数，并传入包含证书的文件名和别名，其语法如下：

```
keytool -exportcert -alias anAlias -file filename
```

包含证书的文件一般以 .cer 为扩展名。例如，为了提取别名为 Meredith 的证书，并将它存入 meredithcertificate.cer 文件，可以使用以下命令：

```
keytool -exportcert -alias Meredith -file meredithcertificate.cer
```

### C.2.5 列出密钥存储库条目

有了可以保存专用密钥的密钥存储库，以及可信任的证书颁发商，就可以利用 keytool 程序将它列出，来查询它的内容，其做法是使用 list 参数：

```
keytool -list -keystore myKeyStore -storepass myPassword
```

如果缺失 keystore 参数，将再次使用默认的密钥存储库。