Broadview
www.broadview.com.cn

几十所院校、培训机构共同的选择

轻量级Java EE企业应用实战（第3版）

——Struts 2+Spring 3+Hibernate整合开发

李刚 编著

疯狂源自梦想

技术成就辉煌

疯狂源自梦想

技术成就辉煌

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

内 容 简 介

本书是《轻量级 Java EE 企业应用实战》的第 3 版，第 3 版保持了第 2 版内容全面、深入的特点，主要完成全部知识的升级。

本书介绍了 Java EE 领域的三个开源框架：Struts 2、Spring 和 Hibernate。其中 Struts 2 升级到 2.2.1，Spring 升级到 3.0.5，Hibernate 升级到了 3.6.0。本书还全面介绍了 Servlet 3.0 的新特性，以及 Tomcat 7.0 的配置和用法，本书的示例应该在 Tomcat 7.0 上运行。

本书重点介绍如何整合 Struts 2.2+Spring 3.0+Hibernate 3.6 进行 Java EE 开发，主要包括三部分。第一部分介绍 Java EE 开发的基础知识，以及如何搭建开发环境。第二部分详细讲解 Struts 2.2、Spring 3.0 和 Hibernate 3.6 三个框架的用法，介绍三个框架时，从 Eclipse IDE 的使用来上手，一步步带领读者深入三个框架的核心。这部分内容是笔者讲授“疯狂 Java 实训”的培训讲义，因此是本书的重点部分，既包含了笔者多年开发经历的领悟，也融入了丰富的授课经验。第三部分示范开发了一个包含 7 个表、表之间具有复杂的关联映射、继承映射等关系，且业务也相对复杂的工作流案例，希望让读者理论联系实际，将三个框架真正运用到实际开发中去，该案例采用目前最流行、最规范的 Java EE 架构，整个应用分为领域对象层、DAO 层、业务逻辑层、MVC 层和视图层，各层之间分层清晰，层与层之间以松耦合的方法组织在一起。该案例既提供了 IDE 无关的、基于 Ant 管理的项目源码，也提供了基于 Eclipse IDE 的项目源码，最大限度地满足读者的需求。

本书不再介绍 Struts 1.X 相关内容，如果读者希望获取《轻量级 J2EE 企业应用实战》第一版中关于 Struts 1.X 的知识，请登录 http://www.crazyit.org 下载。当读者阅读此书时如果遇到技术难题，也可登录 http://www.crazyit.org 发帖，笔者将会及时予以解答。

阅读本书之前，建议先认真阅读笔者所著的《疯狂 Java 讲义》一书。本书适合于有较好的 Java 编程基础，或有初步 JSP、Servlet 基础的读者。尤其适合于对 Struts 2、Spring、Hibernate 了解不够深入，或对 Struts 2+Spring+Hibernate 整合开发不太熟悉的开发人员阅读。

图书在版编目（CIP）数据

轻量级 Java EE 企业应用实战：Struts 2＋Spring 3＋Hibernate 整合开发 / 李刚编著. —3 版. —北京：电子工业出版社，2012.4
ISBN 978-7-121-16085-1

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

中国版本图书馆 CIP 数据核字（2012）第 028389 号

策划编辑：张月萍
责任编辑：张月萍
印　　刷：北京天宇星印刷厂
装　　订：三河市皇庄路通装订厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：850×1168　1/16　印张：52　字数：1440 千字　彩插：1
印　　次：2012 年 4 月第 1 次印刷
印　　数：5000 册　　定价：99.00 元（含光盘 1 张）

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

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

服务热线：（010）88258888。

前　言

经过多年沉淀，Java EE 平台已经成为电信、金融、电子商务、保险、证券等各行业的大型应用系统的首选开发平台。目前 Java 行业的软件开发已经基本稳定，这两三年内基本没有出现什么具有广泛影响力的新技术。Java EE 开发大致可分为两种方式：以 Spring 为核心轻量级 Java EE 企业开发平台；以 EJB 3+JPA 为核心的经典 Java EE 开发平台。无论使用哪种平台进行开发，应用的性能、稳定性都有很好的保证，开发人群也有很稳定的保证。

本书介绍的开发平台，就是以 Struts 2.2+Spring 3.0+Hibernate 3.6（实际项目中可能以 JPA 来代替 Hibernate ）为核心的轻量级 Java EE，这种组合在保留经典 Java EE 应用架构、高度可扩展性、高度可维护性的基础上，降低了 Java EE 应用的开发、部署成本，对于大部分中小型企业应用是首选。在一些需要具有高度伸缩性、高度稳定性的企业应用（比如银行系统、保险系统）里，以 EJB 3+JPA 为核心的经典 Java EE 应用则具有广泛的占有率。本书的姊妹篇《经典 Java EE 企业应用实战》主要介绍了后一种 Java EE 开发平台。

本书主要升级了《轻量级 Java EE 企业应用实战》的知识，采用最新的 Tomcat 7 作为 Web 服务器，全面而细致地介绍了 Servlet 3.0 的新特性，并将 Struts 升级到 Struts 2.2.1，Spring 升级到 3.0.5，Hibernate 升级到 3.6.0。书中详细介绍了 Spring 和 Hibernate 的“零配置”特性，并充分介绍了 Struts 2 的 Convention（约定）支持。本书不仅介绍了 Spring 2.x 的 AOP 支持，详细介绍了 Spring 2.x 中 Schema 配置所支持的 util、aop、tx 等命名空间，还简要讲解了 AspectJ 的相关内容。本书也重点介绍了 Spring 3.0 的新功能：SpEL，SpEL 不仅可以作为表达式语言单独使用，也可与 Spring 容器结合来扩展 Spring 容器的功能。

本书创作感言

笔者首先要感谢广大读者对本书第 2 版的认同，在将近 2 年的时间内，本书第 2 版的销量高达 178 万码洋，得到无数 Java 学习者的认同，成为 Java EE 开发者首选的经典图书。考虑到目前技术的升级，笔者现将本书的全部技术升级到最新版、最前沿，以飨读者。

还有一个值得介绍的消息：本书姊妹篇《经典 Java EE 企业应用实战》（由电子工业出版社出版，ISBN 978-7-121-11534-9）现已上市。学习本书时可以采用“轻经合参”的方式来学习：“轻”指的是以“SSH”整合的轻量级 Java EE 开发平台，“经”指的是以“EJB 3+JPA”整合的经典 Java EE 开发平台；这两种平台本身具有很大的相似性，将两种 Java EE 开发平台结构放在一起参考、对照着学习，能更好地理解 Spring、Hibernate 框架的设计思想，从而更深入地掌握它们。与此同时，也可以深入理解 EJB 3 与 Spring 容器中的 Bean、EJB 容器与 Spring 容器之间的联系和区别，从而融会贯通地掌握 EJB 3+JPA 整合的开发方式。

经常有读者写邮件来问笔者，为何你能快而且全面地掌握各种 Java 开发技术？笔者以前做过一些零散的回复。这里简单地介绍笔者学习 Java 的一些历史与方法，希望广大读者从中借鉴值得学习的地方，避开一些弯路。

笔者大约是 1999 年开始接触 Java，开始主要做点 Applet 玩（当时笔者对 Applet 做出来的动画十分倾心）。后来开始流行 ASP、JSP，笔者再次喜欢上 ASP、JSP 那种极其简单的语法、短期

内的快速上手，后来断断续续用 ASP、JSP 写了多个小型企业网站、BBS、OA 系统之类——不知道其他人是什么经历，笔者选择编程一方面是因为个人爱好和“自豪感”（觉得能做出各种软件，有点成就感），另一方面是因为编写软件可以轻易地卖点钱（是不是很俗？），但这个目的笔者无法回避——由于出生在湖北一个贫穷的乡下，所以在同济念书时笔者常常为了开饭而写代码，或许有一些程序员和笔者会有相同的感触。

在后来的开发过程中，笔者发现纯粹的 JSP 开发虽然前期很方便，但由于开发时代码重复得厉害，所以后期升级、维护很痛苦，于是开始大规模地修改自己写的一堆“垃圾”代码，不断地思考怎样对 JSP 脚本进行提取、封装到 Java Bean 中，这个过程并不顺利，经常遭遇各种性能问题、并发问题。原本可以运行良好的应用，反而被改得经常出现问题。

大约到了 2000 年，笔者接触到 EJB，对 EJB 许下的“承诺”无比欣羡，于是义无反顾地投入 EJB 的怀抱，不过 EJB 的学习并不顺利，当时用的好像是 WebLogic 5 的服务器，那时候觉得 WebLogic 5 所报的错误晦涩、难以阅读，动辄几屏的错误信息，让人感觉很有压力。

不过笔者是一个顽固的人，遇到错误总是不断地修改、不断地尝试，在这样的尝试中，不知不觉中，天色已经发白。说来惭愧，第一个 Hello World 级的 Entity EJB 居然花了将近一个月的时间才弄完（绝不建议读者从 EJB 1.1 或 EJB 2 开始学习，这只会给学习徒增难度，而且现在 EJB 1.1、EJB 2 都已被淘汰）。在那段时间内，笔者连最心爱的 C 几乎完全没碰过。

在接下来的 2 年多时间内，笔者一直沉浸在 EJB 中，不断地搜寻各种关于 EJB 的资料、不断地深入钻研着关于 EJB 规范、EJB 的运行、EJB 容器的运行机制。随着时间的流逝，EJB、EJB 容器的运行原理逐渐明朗起来。

那是一段让人怀念的、“神话”般的岁月，年轻的人，似乎拥有无穷的精力，那也是笔者 Java 技术增长最迅速的 3 年，笔者的 Java EE 功底也是在那 3 年内打下的，后来接触的各种“新”技术只是在那个基础上“修修补补”，或者“温故而知新”。

2004 年初，笔者开始接触到 Spring 框架，从接触 Spring 的第一天开始，直到今天，笔者一直觉得 Spring 和 EJB 之间有很大的相似性：

- Spring 本身也是一个容器，只是 EJB 容器管理的是 EJB，Spring 容器管理的是普通 Java 对象。
- Spring 对 Bean 类的要求很少，EJB 容器对 EJB 的要求略多一些——所以初学者学习 EJB 上手较难，但学习 Spring 就简单得多。

因为找到这种类比性，笔者学习 Spring 时，总是不断地将 EJB 与 Spring 进行类比，然后再找出它们之间的不同之处。由于采用了这种“温故而知新”的学习方式，所以笔者很容易就理解了 Spring 的设计，而且更加透彻。

很多 Java 学习者在学习过程中往往容易感觉 Java 开发内容纷繁芜杂，造成这种感觉的原因就是因为没有进行很好的归纳、总结、类比。为了避免“知识越学越多”的混乱感，读者应该充分利用已掌握的知识，温故而知新——一方面对已有的知识进行归纳、总结，另一方面将新的内容与已掌握的知识进行类比，这样既能把已有的知识掌握得更有条理、更系统，也能更快、更透彻地掌握新的知识。

出于以上理由，笔者在介绍非常专业的编程知识之时，总会通过一些浅显的类比来帮助读者更好地理解。“简单、易读”成为笔者一贯坚持的创作风格，也是疯狂 Java 体系丛书的特色。另一方面，疯狂 Java 体系图书的知识也很全面、实用。笔者希望读者在看完疯狂 Java 体系的图书之后，可以较为轻松地理解书中所介绍的知识，并切实学会一种实用的开发技术，进而将之应用到实际开发中。如果读者在学习过程中遇到无法理解的问题，可以登录疯狂 Java 联盟（http://www.crazyit.

org）与广大 Java 学习者交流，笔者也会通过该平台与大家一起交流、学习。

本书有什么特点

本书保持了《轻量级 Java EE 企业应用实战》第 2 版简单、实用的优势，同样坚持让案例说话、以案例来介绍知识点的风格，在书的最后同样示范开发了企业工作流案例，希望读者通过该案例真正步入实际企业开发的殿堂。

本书依然保留了《轻量级 J2EE 企业应用实战》第 2 版的三个特色。

1．经验丰富，针对性强

笔者既担任过软件开发的技术经理，也担任过软件公司的培训导师，还从事过职业培训的专职讲师，这些经验影响了笔者写书的目的，不是一本学院派的理论读物，而是一本实际的开发指南。

2．内容实际，实用性强

本书所介绍的 Java EE 应用范例，采用了目前企业流行的开发架构，绝对严格遵守 Java EE 开发规范，而不是将各种技术杂乱地糅合在一起号称 Java EE。读者参考本书的架构，完全可以身临其境地感受企业实际开发。

3．高屋建瓴，启发性强

本书介绍的几种架构模式，几乎是时下最全面的 Java EE 架构模式。这些架构模式可以直接提升读者对系统架构设计的把握。

本书写给谁看

如果你已经掌握了 Java SE 内容，或已经学完了《疯狂 Java 讲义》一书，那么你非常适合阅读此书。除此之外，如果你已有初步的 JSP、Servlet 基础，甚至对 Struts 2、Spring 3.0、Hibernate 3.6 有所了解，但希望掌握它们在实际开发中的应用，本书也将非常适合你。如果你对 Java 的掌握还不熟练，则建议遵从学习规律，循序渐进，暂时不要购买、阅读此书。

光盘说明

一、光盘内容

本光盘是《轻量级 Java EE 企业应用开发实战》一书的配书光盘，书中的代码按章、按节存放，即第二章、第二节所使用的代码放在 codes 文件夹的 02\2.2 文件夹下，依次类推。

另：书中每份源代码也给出与光盘源文件的对应关系，方便读者查找。

本光盘 codes 目录下有 10 个文件夹,其内容和含义说明如下：

（1）01～10 个文件夹名对应于《轻量级 Java EE 企业应用开发实战》中的章名，即第二章所使用的代码放在 codes 文件夹的 02 文件夹下，依次类推。

（2）10 文件夹下有 HRSystem 和 HRSystem_Eclipse 两个文件夹，它们是同一个项目的源文件，其中 HRSystem 是 IDE 平台无关的项目，使用 Ant 来编译即可；而 HRSystem_Eclipse 是该项目在 Eclipse IDE 工具中的项目文件。

（3）codes\03\3.2\Struts2Demo 目录、codes\05\5.2\HibernateDemo 目录、codes\07\7.2\myspring 目录和 codes\10\HRSystem_Eclipse 目录下有.classpath、.project 等文件，它们是 Eclipse 项目文件，请不要删除。

二、运行环境

本书中的程序在以下环境调试通过：

（1）安装 jdk-6u22-windows-i586-p.exe，安装完成后，添加 CLASSPATH 环境变量，该环境变量的值为;%JAVA_HOME%/lib/tools.jar;%JAVA_HOME%/lib/dt.jar。如果为了可以编译和运行 Java 程序，还应该在 PATH 环境变量中增加%JAVA_HOME%/bin。其中 JAVA_HOME 代表 JDK（不是 JRE）的安装路径。

（2）安装 Apache 的 Tomcat7.0.6，不要使用安装文件安装，而是采用解压缩的安装方式。安装 Tomcat 请参看第一章。安装完成后，将 Tomcat 安装路径的 lib 下的 jsp-api.jar 和 servlet-api.jar 两个 JAR 文件添加到 CLASSPATH 环境变量之后。

（3）安装 apache-ant-1.8.1。

将下载的 Ant 压缩文件解压缩到任意路径，然后增加 ANT_HOME 的环境变量，让变量的值为 Ant 的解压缩路径。并在 PATH 环境变量中增加%ANT_HOME%/bin 环境变量。

（4）安装 MySQL5.1 或更高版本，安装 MysSQL 时候选择 GBK 的编码方式。

（5）安装 Eclipse-jee-helios 版（也就是 Eclipse 3.6 for Java EE Developers）。

关于如何安装上面工具，请参考本书的第一章。

三、注意事项

（1）独立应用程序的代码中都包括 build.xml 文件，在 Dos 或 Shell 下进入 build.xml 文件所在路径，执行如下命令：

```
ant compile -- 编译程序
ant run --运行程序
```

（2）对于 Web 应用，将该应用复制到%TOMCAT_HOME%/webapps 路径下，然后进入 build.xml 所在路径，执行如下命令：

```
ant compile -- 编译应用
```

启动 Tomcat 服务器，使用浏览器即可访问该应用。

（3）对于 Eclipse 项目文件，导入 Eclipse 开发工具即可。

（4）第 10 章的案例，请参看项目下的 readme.txt。

（5）代码中有大量代码需要连接数据库，读者应修改数据库 URL 以及用户名、密码让这些代码与读者运行环境一致。如果项目下有 SQL 脚本，导入 SQL 脚本即可，如果没有 SQL 脚本，系统将在运行时自动建表，读者只需创建对应数据库即可。

（6）在使用本光盘的程序时，请将程序拷贝到硬盘上，并去除文件的只读属性。

四、技术支持

如果您使用本光盘中遇到不懂的技术问题，您可以登录如下网站与作者联系：

网站：http://www.crazyit.org

目 录 CONTENTS

第 1 章 Java EE 应用和开发环境……1
1.1 Java EE 应用概述……2
1.1.1 Java EE 应用的分层模型……2
1.1.2 Java EE 应用的组件……3
1.1.3 Java EE 应用结构和优势……4
1.1.4 常用的 Java EE 服务器……4
1.2 轻量级 Java EE 应用相关技术……5
1.2.1 JSP、Servlet 3.0 和 JavaBean 及替代技术……5
1.2.2 Struts 2.2 及替代技术……5
1.2.3 Hibernate 3.6 及替代技术……6
1.2.4 Spring 3.0 及替代技术……6
1.3 Tomcat 的下载和安装……7
1.3.1 安装 Tomcat 服务器……8
1.3.2 配置 Tomcat 的服务端口……9
1.3.3 进入控制台……10
1.3.4 部署 Web 应用……12
1.3.5 配置 Tomcat 的数据源……13
1.4 Eclipse 的安装和使用……14
1.4.1 Eclipse 的下载和安装……15
1.4.2 在线安装 Eclipse 插件……15
1.4.3 从本地压缩包安装插件……17
1.4.4 手动安装 Eclipse 插件……17
1.4.5 使用 Eclipse 开发 Java EE 应用……18
1.4.6 导入 Eclipse 项目……21
1.4.7 导入非 Eclipse 项目……22
1.5 Ant 的安装和使用……23
1.5.1 Ant 的下载和安装……23
1.5.2 使用 Ant 工具……24
1.5.3 定义生成文件……25
1.5.4 Ant 的任务（task）……29
1.6 使用 CVS 进行协作开发……31
1.6.1 安装 CVS 服务器……32
1.6.2 配置 CVS 资源库……34
1.6.3 安装 CVS 客户端……35
1.6.4 发布项目到服务器……35
1.6.5 从服务器下载项目……37
1.6.6 同步（Update）本地文件……38
1.6.7 提交（Commit）修改……39
1.6.8 添加文件和目录……39
1.6.9 删除文件和目录……40
1.6.10 查看文件的版本变革……41
1.6.11 提取文件以前版本的内容……42
1.6.12 从以前版本重新开始……42
1.6.13 创建标签……43
1.6.14 创建分支……43
1.6.15 沿着分支开发……44
1.6.16 使用 Eclipse 作为 CVS 客户端……44
1.7 本章小结……46
第 2 章 JSP/Servlet 及相关技术详解……47
2.1 Web 应用和 web.xml 文件……48
2.1.1 构建 Web 应用……48
2.1.2 配置描述符 web.xml……49
2.2 JSP 的基本原理……50
2.3 JSP 注释……54
2.4 JSP 声明……54
2.5 输出 JSP 表达式……56
2.6 JSP 脚本……56
2.7 JSP 的 3 个编译指令……59
2.7.1 page 指令……59
2.7.2 include 指令……63
2.8 JSP 的 7 个动作指令……63
2.8.1 forward 指令……64
2.8.2 include 指令……66
2.8.3 useBean、setProperty、getProperty 指令……67
2.8.4 plugin 指令……70
2.8.5 param 指令……70
2.9 JSP 脚本中的 9 个内置对象……70
2.9.1 application 对象……72
2.9.2 config 对象……77
2.9.3 exception 对象……79
2.9.4 out 对象……81
2.9.5 pageContext 对象……82
2.9.6 request 对象……84
2.9.7 response 对象……91
2.9.8 session 对象……95
2.10 Servlet 介绍……97
2.10.1 Servlet 的开发……97
2.10.2 Servlet 的配置……99
2.10.3 JSP/Servlet 的生命周期……101
2.10.4 load-on-startup Servlet……101
2.10.5 访问 Servlet 的配置参数……102
2.10.6 使用 Servlet 作为控制器……104

2.11 JSP2 的自定义标签 ····· 108
2.11.1 开发自定义标签类 ····· 109
2.11.2 建立 TLD 文件 ····· 109
2.11.3 使用标签库 ····· 110
2.11.4 带属性的标签 ····· 111
2.11.5 带标签体的标签 ····· 115
2.11.6 以页面片段作为属性的标签 ····· 117
2.11.7 动态属性的标签 ····· 118
2.12 Filter 介绍 ····· 120
2.12.1 创建 Filter 类 ····· 120
2.12.2 配置 Filter ····· 121
2.12.3 使用 UrL Rewrite 实现网站伪静态 ····· 125
2.13 Listener 介绍 ····· 126
2.13.1 实现 Listener 类 ····· 127
2.13.2 配置 Listener ····· 128
2.13.3 使用 ServletContextAttribute-Listener ····· 129
2.13.4 使用 ServletRequestListener 和 ServletRequestAttribute-Listener ····· 130
2.13.5 使用 HttpSessionListener 和 HttpSessionAttributeListener ····· 131
2.14 JSP 2 特性 ····· 136
2.14.1 配置 JSP 属性 ····· 136
2.14.2 表达式语言 ····· 138
2.14.3 Tag File 支持 ····· 146
2.15 Servlet 3.0 新特性 ····· 148
2.15.1 Servlet 3.0 的 Annotation ····· 148
2.15.2 Servlet 3.0 的 Web 模块支持 ····· 149
2.15.3 Servlet 3.0 提供的异步处理 ····· 151
2.15.4 改进的 Servlet API ····· 154
2.16 本章小结 ····· 156
第 3 章 Struts 2 的基本用法 ····· 157
3.1 MVC 思想概述 ····· 158
3.1.1 传统 Model 1 和 Model 2 ····· 158
3.1.2 MVC 思想及其优势 ····· 159
3.2 Struts 2的下载和安装 ····· 160
3.2.1 为 Web 应用增加 Struts 2 支持 ····· 160
3.2.2 在 Eclipse 中使用 Struts 2 ····· 161
3.2.3 增加登录处理 ····· 162
3.3 Struts 2的流程 ····· 165
3.3.1 Struts 2 应用的开发步骤 ····· 165
3.3.2 Struts 2 的流程 ····· 166
3.4 Struts 2的常规配置 ····· 167
3.4.1 常量配置 ····· 167
3.4.2 包含其他配置文件 ····· 173
3.5 实现 Action ····· 174
3.5.1 Action 接口和 ActionSupport 基类 ····· 175
3.5.2 Action 访问 Servlet API ····· 177
3.5.3 Action 直接访问 Servlet API ····· 179
3.5.4 使用 ServletActionContext 访问 Servlet API ····· 181
3.6 配置 Action ····· 182
3.6.1 包和命名空间 ····· 182
3.6.2 Action 的基本配置 ····· 185
3.6.3 使用 Action 的动态方法调用 ····· 186
3.6.4 指定 method 属性及使用通配符 ····· 188
3.6.5 配置默认 Action ····· 194
3.6.6 配置 Action 的默认处理类 ····· 194
3.7 配置处理结果 ····· 195
3.7.1 理解处理结果 ····· 195
3.7.2 配置结果 ····· 195
3.7.3 Struts 2 支持的结果类型 ····· 197
3.7.4 plainText 结果类型 ····· 198
3.7.5 redirect 结果类型 ····· 200
3.7.6 redirectAction 结果类型 ····· 201
3.7.7 动态结果 ····· 202
3.7.8 Action 属性值决定物理视图资源 ····· 202
3.7.9 全局结果 ····· 204
3.7.10 使用 PreResultListener ····· 205
3.8 配置 Struts 2 的异常处理 ····· 206
3.8.1 Struts 2 的异常处理机制 ····· 207
3.8.2 声明式异常捕捉 ····· 208
3.8.3 输出异常信息 ····· 210
3.9 Convention 插件与“约定”支持 ····· 211
3.9.1 Action 的搜索和映射约定 ····· 211
3.9.2 按约定映射 Result ····· 214
3.9.3 Action 链的约定 ····· 216
3.9.4 自动重加载映射 ····· 218
3.9.5 C zonvention 插件的相关常量 ····· 218
3.9.6 Convention 插件相关 Annotation ····· 219
3.10 使用 Struts 2 的国际化 ····· 219
3.10.1 Struts 2 中加载全局资源文件 ····· 219
3.10.2 访问国际化消息 ····· 220
3.10.3 输出带占位符的国际化消息 ····· 222
3.10.4 加载资源文件的方式 ····· 224
3.10.5 加载资源文件的顺序 ····· 228

3.11 使用 Struts 2 的标签库 ····· 228
3.11.1 Struts 2 标签库概述 ····· 228
3.11.2 使用 Struts 2 标签 ····· 229
3.11.3 Struts 2 的 OGNL 表达式语言 ····· 230
3.11.4 OGNL 中的集合操作 ····· 232
3.11.5 访问静态成员 ····· 233
3.11.6 Lambda（λ）表达式 ····· 234
3.11.7 控制标签 ····· 234
3.11.8 数据标签 ····· 244
3.11.9 主题和模板 ····· 254
3.11.10 自定义主题 ····· 256
3.11.11 表单标签 ····· 257
3.11.12 非表单标签 ····· 270
3.12 本章小结 ····· 273
第 4 章 深入使用 Struts 2 ····· 274
4.1 详解 Struts 2 的类型转换 ····· 275
4.1.1 Struts 2 内建的类型转换器 ····· 276
4.1.2 基于 OGNL 的类型转换 ····· 276
4.2.3 指定集合元素的类型 ····· 279
4.1.4 自定义类型转换器 ····· 280
4.1.5 注册类型转换器 ····· 283
4.1.6 基于 Struts 2 的自定义类型转换器 ····· 284
4.1.7 处理 Set 集合 ····· 285
4.1.8 类型转换中的错误处理 ····· 288
4.2 使用 Struts 2 的输入校验 ····· 293
4.2.1 编写校验规则文件 ····· 294
4.2.2 国际化提示信息 ····· 296
4.2.3 使用客户端校验 ····· 298
4.2.4 字段校验器配置风格 ····· 300
4.2.5 非字段校验器配置风格 ····· 301
4.2.6 短路校验器 ····· 302
4.2.7 校验文件的搜索规则 ····· 304
4.2.8 校验顺序和短路 ····· 305
4.2.9 内建校验器 ····· 306
4.2.10 基于 Annotation 的输入校验 ····· 316
4.2.11 手动完成输入校验 ····· 318
4.3 使用 Struts 2 控制文件上传 ····· 322
4.3.1 Struts 2 的文件上传 ····· 322
4.3.2 实现文件上传的 Action ····· 322
4.3.3 配置文件上传的 Action ····· 325
4.3.4 手动实现文件过滤 ····· 326
4.3.6 拦截器实现文件过滤 ····· 328
4.3.7 输出错误提示 ····· 329
4.3.8 文件上传的常量配置 ····· 330
4.4 使用 Struts 2 控制文件下载 ····· 330
4.4.1 实现文件下载的 Action ····· 330
4.4.2 配置 Action ····· 332
4.4.3 下载前的授权控制 ····· 332
4.5 详解 Struts 2 的拦截器机制 ····· 334
4.5.1 拦截器在 Struts 2 中的作用 ····· 334
4.5.2 Struts 2 内建的拦截器 ····· 334
4.5.3 配置拦截器 ····· 336
4.5.4 使用拦截器 ····· 338
4.5.5 配置默认拦截器 ····· 338
4.5.6 实现拦截器类 ····· 340
4.5.7 使用拦截器 ····· 342
4.5.8 拦截方法的拦截器 ····· 343
4.5.9 拦截器的执行顺序 ····· 345
4.5.10 拦截结果的监听器 ····· 347
4.5.11 覆盖拦截器栈里特定拦截器的参数 ····· 348
4.5.12 使用拦截器完成权限控制 ····· 349
4.6 使用 Struts 2 的 Ajax 支持 ····· 351
4.6.1 使用 stream 类型的 Result 实现 Ajax ····· 352
4.6.2 JSON 的基本知识 ····· 354
4.6.3 实现 Action 逻辑 ····· 356
4.6.4 JSON 插件与 json 类型的 Result ····· 357
4.6.5 实现 JSP 页面 ····· 359
4.7 本章小结 ····· 361
第 5 章 Hibernate 的基本用法 ····· 362
5.1 ORM 和 Hibernate ····· 363
5.1.1 对象/关系数据库映射（ORM） ····· 363
5.1.2 基本映射方式 ····· 364
5.1.3 流行的 ORM 框架简介 ····· 365
5.1.4 Hibernate 概述 ····· 366
5.2 Hibernate 入门 ····· 366
5.2.1 Hibernate 下载和安装 ····· 366
5.2.2 Hibernate 的数据库操作 ····· 367
5.2.3 在 Eclipse 中使用 Hibernate ····· 371
5.3 Hibernate 的体系结构 ····· 376
5.4 深入 Hibernate 的配置文件 ····· 377
5.4.1 创建 Configuration 对象 ····· 377
5.4.2 hibernate.properties 文件与 hibernate.cfg.xml 文件 ····· 380
5.4.3 JDBC 连接属性 ····· 380
5.4.4 数据库方言 ····· 381

5.4.5 JNDI 数据源的连接属性……382
5.4.6 Hibernate 事务属性……382
5.4.7 二级缓存相关属性……383
5.4.8 外连接抓取属性……383
5.4.9 其他常用的配置属性……383
5.5 深入理解持久化对象……384
5.5.1 持久化类的要求……384
5.5.2 持久化对象的状态……385
5.5.3 改变持久化对象状态的方法……386
5.6 深入 Hibernate 的映射文件……389
5.6.1 映射文件结构……389
5.6.2 映射主键……392
5.6.3 映射普通属性……393
5.6.4 映射集合属性……398
5.6.5 集合属性的性能分析……407
5.6.6 有序集合映射……409
5.6.7 映射数据库对象……411
5.7 映射组件属性……414
5.7.1 组件属性为集合……416
5.7.2 集合属性的元素为组件……418
5.7.3 组件作为 Map 的索引……420
5.7.4 组件作为复合主键……422
5.7.5 多列作为联合主键……425
5.8 使用 JPA Annotation 标注实体……426
5.8.1 增加 JPA Annotation 支持……426
5.8.2 Annotation？还是 XML 映射文件…429
5.9 本章小结……429
第 6 章 深入使用 Hibernate……430
6.1 Hibernate 的关联映射……431
6.1.1 单向 N－1 关联……431
6.1.2 单向 1－1 关联……436
6.1.3 单向 1－N 关联……439
6.1.4 单向 N－N 关联……443
6.1.5 双向 1－N 关联……443
6.1.6 双向 N－N 关联……448
6.1.7 双向 1－1 关联……450
6.1.8 组件属性包含的关联实体……453
6.1.9 基于复合主键的关联关系……456
6.1.10 复合主键的成员属性为关联实体…458
6.1.11 持久化的传播性……461
6.2 继承映射……462
6.2.1 采用 subclass 元素的继承映射……466
6.2.2 采用 joined-subclass 元素的继承映射……467
6.2.3 采用 union-subclass 元素的继承映射……470
6.3 Hibernate 的批量处理……472
6.3.1 批量插入……473
6.3.2 批量更新……474
6.3.3 DML 风格的批量更新/删除……474
6.4 使用 HQL 查询……476
6.4.1 HQL 查询……476
6.4.2 HQL 查询的 from 子句……478
6.4.3 关联和连接……478
6.4.4 HQL 查询的 select 子句……481
6.4.5 HQL 查询的聚集函数……482
6.4.6 多态查询……483
6.4.7 HQL 查询的 where 子句……483
6.4.8 表达式……484
6.4.9 order by 子句……486
6.4.10 group by 子句……486
6.4.11 子查询……487
6.4.12 命名查询……488
6.5 条件查询……488
6.5.1 关联和动态关联……491
6.5.2 投影、聚合和分组……492
6.5.3 离线查询和子查询……495
6.6 SQL 查询……496
6.6.1 标量查询……496
6.6.2 实体查询……498
6.6.3 处理关联和继承……500
6.6.4 命名 SQL 查询……501
6.6.5 调用存储过程……502
6.6.6 使用定制 SQL……503
6.7 数据过滤……505
6.8 事务控制……508
6.8.1 事务的概念……508
6.8.2 Session 与事务……509
6.8.3 上下文相关的 Session……511
6.9 二级缓存和查询缓存……511
6.9.1 开启二级缓存……512
6.9.2 管理缓存和统计缓存……515
6.9.3 使用查询缓存……516
6.10 事件机制……518
6.10.1 拦截器……519
6.10.2 事件系统……521
6.11 本章小结……525
第 7 章 Spring 的基本用法……526
7.1 Spring 简介和 Spring 3.0 的变化…527

7.1.1 Spring 简介 ····· 527
7.1.2 Spring 3.0 的变化 ····· 528
7.2 Spring 的下载和安装 ····· 528
7.2.1 在 Java SE 应用中使用 Spring ····· 528
7.2.2 在 Web 应用中使用 Spring ····· 529
7.2.3 在 Eclipse 中开发 Spring 应用 ····· 530
7.3 Spring 的核心机制：依赖注入 ····· 533
7.3.1 理解依赖注入 ····· 533
7.3.2 设值注入 ····· 534
7.3.3 构造注入 ····· 538
7.3.4 两种注入方式的对比 ····· 539
7.4 使用 Spring 容器 ····· 539
7.4.1 Spring 容器 ····· 540
7.4.2 使用 ApplicationContext ····· 541
7.4.3 ApplicationContext 的国际化支持 ··· 542
7.4.4 ApplicationContext 的事件机制 ····· 544
7.4.5 让 Bean 获取 Spring 容器 ····· 546
7.5 Spring 容器中的 Bean ····· 548
7.5.1 Bean 的基本定义 ····· 548
7.5.2 容器中 Bean 的作用域 ····· 551
7.5.3 配置依赖 ····· 553
7.5.4 设置普通属性值 ····· 555
7.5.5 配置合作者 Bean ····· 557
7.5.6 使用自动装配注入合作者 Bean ····· 557
7.5.7 注入嵌套 Bean ····· 560
7.5.8 注入集合值 ····· 561
7.5.9 组合属性名称 ····· 565
7.5.10 Spring 的 Bean 和 JavaBean ····· 566
7.6 Spring 30 提供的 Java 配置管理 ··· 567
7.7 Bean 实例的创建方式及依赖配置 ····· 570
7.7.1 使用构造器创建 Bean 实例 ····· 570
7.7.2 使用静态工厂方法创建 Bean ····· 572
7.7.3 调用实例工厂方法创建 Bean ····· 575
7.8 深入理解容器中的 Bean ····· 577
7.8.1 使用抽象 Bean ····· 577
7.8.2 使用子 Bean ····· 578
7.8.3 Bean 继承与 Java 继承的区别 ····· 579
7.8.4 容器中的工厂 Bean ····· 580
7.8.5 获得 Bean 本身的 id ····· 582
7.8.6 强制初始化 Bean ····· 583
7.9 容器中 Bean 的生命周期 ····· 583
7.9.1 依赖关系注入之后的行为 ····· 584
7.9.2 Bean 销毁之前的行为 ····· 585
7.9.3 协调作用域不同步的 Bean ····· 588
7.10 深入理解依赖关系配置 ····· 591
7.10.1 注入其他 Bean 的属性值 ····· 592
7.10.2 注入其他 Bean 的 Field 值 ····· 594
7.10.3 注入其他 Bean 的方法返回值 ····· 595
7.11 基于 XML Schema 的简化配置方式 ····· 598
7.11.1 使用 p 名称空间配置属性 ····· 599
7.11.2 使用 util Schema ····· 600
7.12 Spring 3.0 提供的表达式语言（SpEL） ····· 602
7.12.1 使用 Expression 接口进行表达式求值 ····· 603
7.12.2 Bean 定义中的表达式语言支持 ··· 604
7.12.3 SpEL 语法详述 ····· 606
7.13 本章小结 ····· 611
第 8 章 深入使用 Spring ····· 612
8.1 两种后处理器 ····· 613
8.1.1 Bean 后处理器 ····· 613
8.1.2 Bean 后处理器的用处 ····· 617
8.1.3 容器后处理器 ····· 617
8.1.4 属性占位符配置器 ····· 619
8.1.5 重写占位符配置器 ····· 620
8.2 Spring 的“零配置”支持 ····· 621
8.2.1 搜索 Bean 类 ····· 621
8.2.2 指定 Bean 的作用域 ····· 624
8.2.3 使用@Resource 配置依赖 ····· 625
8.2.4 使用@PostConstruct 和@PreDestroy 定制生命周期行为 ····· 626
8.2.5 Spring 3.0 新增的 Annotation ····· 626
8.2.6 自动装配和精确装配 ····· 627
8.3 资源访问 ····· 629
8.3.1 Resource 实现类 ····· 630
8.3.2 ResourceLoader 接口和 ResourceLoaderAware 接口 ····· 635
8.3.3 使用 Resource 作为属性 ····· 638
8.3.4 在 ApplicationContext 中使用资源 ····· 639
8.4 Spring 的 AOP ····· 643
8.4.1 为什么需要 AOP ····· 643
8.4.2 使用 AspectJ 实现 AOP ····· 644
8.4.3 AOP 的基本概念 ····· 649
8.4.4 Spring 的 AOP 支持 ····· 650
8.4.5 基于 Annotation 的“零配置”方式 ····· 651
8.4.6 基于 XML 配置文件的管理方式 ··· 666

8.5 Spring 的事务……672
8.5.1 Spring 支持的事务策略……673
8.5.2 使用 TransactionProxyFactoryBean 创建事务代理……678
8.5.3 Spring 2.X 的事务配置策略……681
8.5.4 使用@Transactional……685
8.6 Spring 整合 Struts 2……686
8.6.1 启动 Spring 容器……686
8.6.2 MVC 框架与 Spring 整合的思考……688
8.6.3 让 Spring 管理控制器……689
8.6.4 使用自动装配……692
8.7 Spring 整合 Hibernate……695
8.7.1 Spring 提供的 DAO 支持……695
8.7.2 管理 Hibernate 的 SessionFactory……696
8.7.3 使用 HibernateTemplate……697
8.7.4 使用 HibernateCallback……701
8.7.5 实现 DAO 组件……703
8.7.6 使用 IoC 容器组装各种组件……705
8.7.7 使用声明式事务……707
8.8 Spring 整合 JPA……708
8.8.1 管理 EntityManager……709
8.8.2 使用 JpaTemplate……711
8.8.4 使用 JpaCallback……713
8.8.5 借助 JpaDaoSupport 实现 DAO 组件……714
8.8.6 使用声明式事务……714
8.9 本章小结……715
第 9 章 企业应用开发的思考和策略……716
9.1 企业应用开发面临的挑战……717
9.1.1 可扩展性、可伸缩性……717
9.1.2 快捷、可控的开发……718
9.1.3 稳定性、高效性……719
9.1.4 花费最小化，利益最大化……719
9.2 如何面对挑战……719
9.2.1 使用建模工具……719
9.2.2 利用优秀的框架……720
9.2.3 选择性地扩展……722
9.2.4 使用代码生成器……722
9.3 常见设计模式精讲……722
9.3.1 单例模式……723
9.3.2 简单工厂……724
9.3.3 工厂方法和抽象工厂……730
9.3.4 代理模式……733
9.3.5 命令模式……739
9.3.6 策略模式……741
9.3.7 门面模式……743
9.3.8 桥接模式……746
9.3.9 观察者模式……750
9.4 常见的架构设计策略……753
9.4.1 贫血模型……753
9.4.2 领域对象模型……756
9.4.3 合并业务逻辑对象与 DAO 对象……758
9.4.4 合并业务逻辑对象和 Domain Object……759
9.4.5 抛弃业务逻辑层……761
9.5 本章小结……762
第 10 章 简单工作流系统……763
10.1 项目背景及系统结构……764
10.1.1 应用背景……764
10.1.2 系统功能介绍……764
10.1.3 相关技术介绍……765
10.1.4 系统结构……766
10.1.5 系统的功能模块……766
10.2 Hibernate 持久层……767
10.2.1 设计持久化实体……767
10.2.2 创建持久化实体类……768
10.2.3 映射持久化实体……772
10.3 实现 DAO 层……777
10.3.1 DAO 组件的定义……778
10.3.2 实现 DAO 组件……783
10.3.3 部署 DAO 层……787
10.4 实现 Service 层……789
10.4.1 业务逻辑组件的设计……789
10.4.2 实现业务逻辑组件……789
10.4.3 事务管理……795
10.4.4 部署业务逻辑组件……795
10.5 实现任务的自动调度……797
10.5.1 使用 Quartz……797
10.5.2 在 Spring 中使用 Quartz……802
10.6 实现系统 Web 层……804
10.6.1 Struts 2 和 Spring 的整合……804
10.6.2 控制器的处理顺序……805
10.6.3 员工登录……806
10.6.4 进入打卡……808
10.6.5 处理打卡……810
10.6.6 进入申请……811
10.6.7 提交申请……812
10.6.8 使用拦截器完成权限管理……814
10.7 本章小结……816

JSP（Java Server Page）和 Servlet 是 Java EE 规范的两个基本成员，它们是 Java Web 开发的重点知识，也是 Java EE 开发的基础知识。JSP 和 Servlet 的本质是一样的，因此 JSP 最终必须编译成 Servlet 才能运行，或者说 JSP 只是生成 Servlet 的“草稿”文件。JSP 比较简单，它的特点是在 HTML 页面中嵌入 Java 代码片段，或使用各种 JSP 标签，包括使用用户自定义标签，从而可以动态地提供页面内容。

早期使用 JSP 页面的用户非常广泛，一个 Web 应用可以全部由 JSP 页面组成，只辅以少量的 JavaBean 即可。自 Java EE 标准出现以后，人们逐渐认识到使用 JSP 充当过多的角色是不合适的。因此，JSP 慢慢发展成单一的表现层技术，不再承担业务逻辑组件及持久层组件的责任。

随着 Java EE 技术的发展，又出现了 FreeMarker、Velocity、Tapestry 等表现层技术，虽然这些技术基本可以取代 JSP 技术，但实际上 JSP 依然是应用最广泛的表现层技术。本书介绍的 JSP 技术是基于 JSP 2.2、Servlet 3.0 规范的，因此请使用支持 Java EE 6 规范的应用服务器或支持 Servlet 3.0 的 Web 服务器（比如 Tomcat 7.0.X）。

除了介绍 JSP 技术之外，本章也会讲解 JSP 的各种相关技术：Servlet、Listener、Filter 以及自定义标签库等技术。

2.1 Web 应用和 web.xml 文件

JSP、Servlet、Listener 和 Filter 等都必须运行在 Web 应用中，所以我们先来学习如何构建一个 Web 应用。

➤➤2.1.1 构建 Web 应用

在 1.5.5 节中已经介绍了如何通过 Eclipse 来构建一个 Web 应用，但笔者坚持认为：如果你仅学会在 Eclipse 等 IDE 工具中单击“下一步”、“确定”等按钮，那你将很难成为一个真正的程序员。

笔者一直相信：要想成为一个优秀的程序员，应该从基本功练起，所有的代码都应该用简单的文本编辑器（包括 EditPlus、UltraEdit 等工具）完成。

坚持使用最原始的工具来学习技术，会让你对整个技术的每个细节有更准确的把握。比如说你掌握了 1.4.5 节的内容，但你是否知道 Eclipse 创建 Web 应用时为你做了些什么？如果你还不清楚 Eclipse 所干的每件事情，那你还不能使用它。

实际上，真正优秀的程序员当然应该使用 IDE 工具，但即使使用 vi（UNIX 下无格式编辑器）、记事本也一样可以完成非常优秀的项目。笔者对于 IDE 工具的态度是：可以使用 IDE 工具，但绝不可依赖于 IDE 工具。学习阶段，千万不可使用 IDE 工具；开发阶段，使用 IDE 工具。

提示：对于 IDE 工具，业内有一个说法：IDE 工具会加快高手的开发效率，但会使初学者更白痴。

下面我们将“徒手”建立一个 Web 应用，请按如下步骤进行：

① 在任意目录下新建一个文件夹，笔者将以 webDemo 文件夹建立一个 Web 应用。

② 在第 1 步所建的文件夹内建一个 WEB-INF 文件夹（注意大小写，这里区分大小写）。

③ 进入 Tomcat 或任何其他 Web 容器内，找到任何一个 Web 应用，将 Web 应用的 WEB-INF 下的 web.xml 文件复制到第 2 步所建的 WEB-INF 文件夹下。

对于 Tomcat 而言，位于它的 webapps 路径下有大量的示例 Web 应用；对于 Jetty 而言，它的 webapps 路径下也有多个 Web 应用。

④ 修改复制后的 web.xml 文件，将该文件修改成只有一个根元素的 XML 文件。修改后的 web.xml 文件代码如下。

程序清单：codes\02\2.1\webDemo\WEB-INF\web.xml

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

```
<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">
</web-app>
```

在第 2 步所建的 WEB-INF 路径下，新建两个文件夹：classes 和 lib，这两个文件夹的作用完全相同：都是用于保存 Web 应用所需要的 Java 类文件，区别是 classes 保存单个*.class 文件；而 lib 保存打包后的 JAR 文件。

经过以上步骤，已经建立了一个空 Web 应用。将该 Web 应用复制到 Tomcat 的 webapps 路径下，该 Web 应用将可以自动部署在 Tomcat 中。

通常我们只需将 JSP 放在 Web 应用的根路径下（对本例而言，就是放在 webDemo 目录下），然后就可以通过浏览器来访问这些页面了。

根据上面介绍，不难发现 Web 应用应该有如下文件结构：

<webDemo>——这是 Web 应用的名称，可以改变

|－WEB-INF

|　　|－classes

|　　|－lib

|　　|－web.xml

|－<a.jsp>——这里存放任意多个 JSP 页面

上面的 webDemo 是 Web 应用所对应文件夹的名字，可以更改；a.jsp 是该 Web 应用下 JSP 页面的名字，也可以修改（还可以增加更多的 JSP 页面）。其他文件夹、配置文件都不可以修改。

a.jsp 页面的内容如下。

程序清单：codes\02\2.1\webDemo\a.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<html>
<head>
     <title>欢迎</title>
</head>
<body>
     欢迎学习 Java Web 知识
</body>
</html>
```

上面的页面实际上是一个静态 HTML 页面，在浏览器中浏览该页面将看到如图 2.1 所示的界面。

看到如图 2.1 所示的页面即表示 Web 应用构建成功，并已经将其成功地部署到 Tomcat 中了。

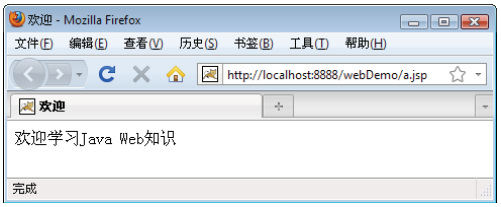

图 2.1　构建 Web 应用

➤➤2.1.2　配置描述符 web.xml

上一节介绍的、位于每个 Web 应用的 WEB-INF 路径下的 web.xml 文件被称为配置描述符，这个 web.xml 文件对于 Java Web 应用十分重要，在 Servlet 2.5 规范之前，每个 Java Web 应用都必须包含一个 web.xml 文件，且必须放在 WEB-INF 路径下。

提示：对于 Servlet 3.0 规范而言，WEB-INF 路径下的 web.xml 文件不再是必需的，但通常还是建议保留该配置文件。

对于 Java Web 应用而言，WEB-INF 是一个特殊的文件夹，Web 容器会包含该文件夹下的内容，

客户端浏览器无法访问 WEB-INF 路径下的任何内容。

在 Servlet 2.5 规范之前，Java Web 应用的绝大部分组件都通过 web.xml 文件来配置管理，Servlet 3.0 规范可通过 Annotation 来配置管理 Web 组件，因此 web.xml 文件可以变得更加简洁，这也是 Servlet 3.0 的重要简化。接下来我们介绍的如下内容会同时介绍两种配置管理方式：

- 配置 JSP。
- 配置和管理 Servlet。
- 配置和管理 Listener。
- 配置和管理 Filter。
- 配置标签库。
- 配置 JSP 属性。

除此之外，web.xml 还负责配置、管理如下常用内容：

- 配置和管理 JAAS 授权认证。
- 配置和管理资源引用。
- Web 应用首页。

web.xml 文件的根元素是<web-app.../>元素，在 Servlet 3.0 规范中，该元素新增了如下属性。

- metadata-complete：该属性接受 true 或 false 两个属性值。当该属性值为 true 时，该 Web 应用将不会加载 Annotation 配置的 Web 组件（如 Servlet、Filter、Listener 等）。

在 web.xml 文件中配置首页使用 welcome-file-list 元素，该元素能包含多个 welcome-file 子元素，其中每个 welcome-file 子元素配置一个首页。例如如下配置片段：

```
<!-- 配置 Web 应用的首页列表 -->
<welcome-file-list>
    <welcome-file>index.html</welcome-file>
    <welcome-file>index.htm</welcome-file>
    <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

上面的配置信息指定该 Web 应用的首页依次是 index.html、index.htm 和 index.jsp，意思是说：当 Web 应用中包含 index.html 页面时，如果浏览者直接访问该 Web 应用，系统将会把该页面呈现给浏览者；当 index.html 页面不存在时，则由 index.htm 页面充当首页，依此类推。

每个 Web 容器都会提供一个系统的 web.xml 文件，用于描述所有 Web 应用共同的配置属性。例如，Tomcat 的系统 web.xml 放在 Tomcat 的 conf 路径下，而 Jetty 的系统 web.xml 文件放在 Jetty 的 etc 路径下，文件名为 webdefault.xml。

2.2 JSP 的基本原理

JSP 的本质是 Servlet，当用户向指定 Servlet 发送请求时，Servlet 利用输出流动态生成 HTML 页面，包括每一个静态的 HTML 标签和所有在 HTML 页面中出现的内容。

由于包括大量的 HTML 标签、大量的静态文本及格式等，导致 Servlet 的开发效率极为低下。所有的表现逻辑，包括布局、色彩及图像等，都必须耦合在 Java 代码中，这的确让人不胜其烦。JSP 的出现弥补了这种不足，JSP 通过在标准的 HTML 页面中嵌入 Java 代码，其静态的部分无须 Java 程序控制，只有那些需要从数据库读取或需要动态生成的页面内容，才使用 Java 脚本控制。

从上面的介绍可以看出，JSP 页面的内容由如下两部分组成。

- 静态部分：标准的 HTML 标签、静态的页面内容，这些内容与静态 HTML 页面相同。
- 动态部分：受 Java 程序控制的内容，这些内容由 Java 程序来动态生成。

下面是一个最简单的 JSP 页面代码。

程序清单：codes\02\2.2\jspPrinciple\first.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<html>
<head>
<title>欢迎</title>
</head>
<body>
欢迎学习 Java Web 知识，现在时间是：
<%out.println(new java.util.Date());%>
</body>
</html>
```

上面的页面中粗体字代码放在<%和%>之间，表明这些是 Java 脚本，而不是静态内容，通过这种方式就可以把 Java 代码嵌入 HTML 页面中，这就变成了动态的 JSP 页面。在浏览器中浏览该页面，将看到如图 2.2 所示的页面。

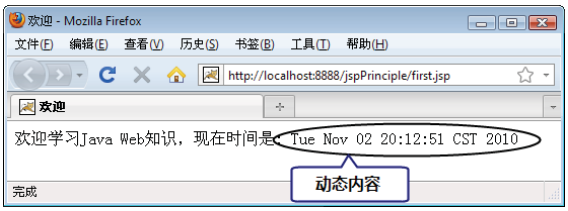

图 2.2　JSP 页面的静态部分和动态部分

上面 JSP 页面必须放在 Web 应用中才有效，所以编写该 JSP 页面之前应该先构建一个 Web 应用。本章后面介绍的内容都必须运行在 Web 应用中，所以也必须先构建 Web 应用。

从表面上看，JSP 页面已经不再需要 Java 类，似乎完全脱离了 Java 面向对象的特征。事实上，JSP 的本质依然是 Servlet（一个特殊的 Java 类），每个 JSP 页面就是一个 Servlet 实例——JSP 页面由系统编译成 Servlet，Servlet 再负责响应用户请求。也就是说，JSP 其实也是 Servlet 的一种简化，使用 JSP 时，其实还是使用 Servlet，因为 Web 应用中的每个 JSP 页面都会由 Servlet 容器生成对应的 Servlet。对于 Tomcat 而言，JSP 页面生成的 Servlet 放在 work 路径对应的 Web 应用下。

再看如下一个简单的 JSP 页面。

程序清单：codes\02\2.2\jspPrinciple\test.jsp

```
<!-- 表明这是一个 JSP 页面 -->
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title> 第二个 JSP 页面 </title>
</head>
<body>
<!-- 下面是 Java 脚本 -->
<%for(int i = 0 ; i < 7; i++)
{
out.println("<font size='" + i + "'>");
%>
疯狂 Java 训练营(Wild Java Camp)</font>
<br/>
<%}%>
</body>
</html>
```

当启动 Tomcat 之后，可以在 Tomcat 的 work\Catalina\localhost\jspPrinciple\org\apache\jsp 目录下找到如下文件（本 Web 应用名为 jspPrinciple，上面 JSP 页的名为 test.jsp）：test_jsp.java 和 test_jsp.class。这两个文件都是由 Tomcat 生成的，Tomcat 根据 JSP 页面生成对应 Servlet 的 Java 文件和 class 文件。

下面是 test1_jsp.java 文件的源代码，这是一个特殊的 Java 类，是一个 Servlet 类。

程序清单：codes\02\2.2\test.java

```
//JSP 页面经过 Tomcat 编译后默认的包
package org.apache.jsp;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.jsp.*;
//继承 HttpJspBase 类，该类其实是 Servlet 的子类
public final class test_jsp extends org.apache.jasper.runtime.HttpJspBase
      implements org.apache.jasper.runtime.JspSourceDependent {
      private static final JspFactory _jspxFactory = JspFactory.getDefaultFactory();
      private static java.util.List<String> _jspx_dependants;
      private javax.el.ExpressionFactory _el_expressionfactory;
      private org.apache.tomcat.InstanceManager _jsp_instancemanager;
      public java.util.List<String> 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() {
      }
      //用于响应用户请求的方法
      public void _jspService(HttpServletRequest request, HttpServletResponse response)
            throws java.io.IOException, ServletException {
            PageContext pageContext = null;
            HttpSession session = null;
            ServletContext application = null;
            ServletConfig config = null;
            JspWriter out = null;
            Object page = this;
            JspWriter _jspx_out = null;
            PageContext _jspx_page_context = null;

            try {
                  response.setContentType("text/html; charset=GBK");
                  pageContext = _jspxFactory.getPageContext(this, request, response,
                  "", true, 8192, true);
                  _jspx_page_context = pageContext;
                  application = pageContext.getServletContext();
                  config = pageContext.getServletConfig();
                  session = pageContext.getSession();
                  out = pageContext.getOut();
                  _jspx_out = out;
                  out.write("\r\n");
                  out.write("\r\n");
                  out.write("\r\n");
                  out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional
                  //EN\"\r\n");
                  out.write("\t\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional. dtd\">
                  \r\n");
                  out.write("<html xmlns=\"http://www.w3.org/1999/xhtml\">\r\n");
                  out.write("<head>\r\n");
                  out.write("\t<title> 第二个 JSP 页面 </title>\r\n");
                  out.write("\t<meta name=\"website\" content=\http://www.crazyit.org\ />\
                  r\n");
                  out.write("</head>\r\n");
                  out.write("<body>\r\n");
                  out.write("<!-- 下面是 Java 脚本 -->\r\n");
                  for(int i = 0 ; i < 7; i++)
                  {
                  out.println("<font size='" + i + "'>");
                  out.write("\r\n");
                  out.write("疯狂 Java 训练营(Wild Java Camp)</font>\r\n");
                  out.write("<br/>\r\n");
                  }
```

```
                out.write("\r\n");
                out.write("</body>\r\n");
                out.write("</html>");
            } catch (Throwable t) {
                if (!(t instanceof 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.handlePage
                    Exception(t);
                }
            } finally {
                _jspxFactory.releasePageContext(_jspx_page_context);
            }
        }
    }
```

初学者看到上面的 Java 类可能有点难以阅读，其实这就是一个 Servlet 类的源代码，该 Java 类主要包含如下三个方法（去除方法名中的_jsp 前缀，再将首字母小写）。

- init()：初始化 JSP/Servlet 的方法。
- destroy()：销毁 JSP/Servlet 之前的方法。
- service()：对用户请求生成响应的方法。

即使读者暂时不了解上面提供的 Java 代码，也依然不会影响 JSP 页面的编写，因为这都是由 Web 容器负责生成的，后面介绍了编写 Servlet 的知识之后再来看这个 Java 类将十分清晰。浏览该页面可看到如图 2.3 所示的页面。

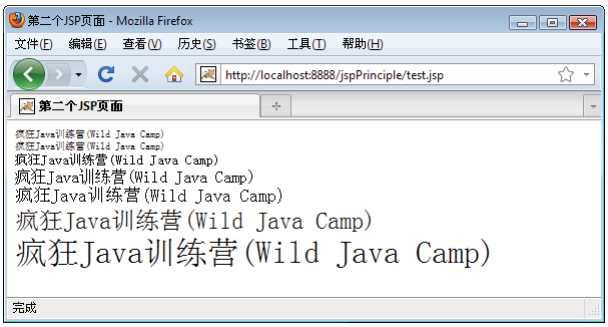

图 2.3　使用 Java 代码控制静态内容

从图 2.3 中可以看出，JSP 页面里的 Java 代码不仅仅可以输出动态内容，还可以动态控制页面里的静态内容，例如，从图 2.3 中看到将“疯狂 Java 训练营(Wild Java Camp)”重复输出了 7 次。

根据图 2.3 所示的执行效果，再次对比 test1.jsp 和 test1_jsp.java 文件，可得到一个结论：JSP 页面中的所有内容都由 test1_jsp.java 文件的页面输出流来生成。图 2.4 显示了 JSP 页面的工作原理。

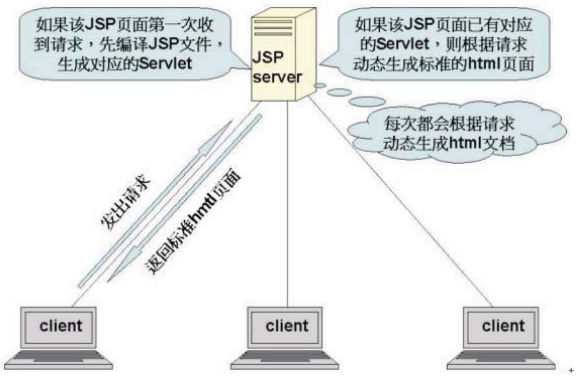

图 2.4　JSP 页面的工作原理

根据上面的 JSP 页面工作原理图，可以得到如下 4 个结论：

- JSP 文件必须在 JSP 服务器内运行。
- JSP 文件必须生成 Servlet 才能执行。
- 每个 JSP 页面的第一个访问者速度很慢，因为必须等待 JSP 编译成 Servlet。
- JSP 页面的访问者无须安装任何客户端，甚至不需要可以运行 Java 的运行环境，因为 JSP 页面输送到客户端的是标准 HTML 页面。

JSP 技术的出现，大大提高了 Java 动态网站的开发效率，所以得到了 Java 动态网站开发者的广泛支持。

2.3 JSP 注释

JSP 注释用于标注在程序开发过程中的开发提示，它不会输出到客户端。

JSP 注释的格式如下：

```
<%-- 注释内容 --%>
```

与 JSP 注释形成对比的是 HTML 注释，HTML 注释的格式是：

```
<!-- 注释内容 -->
```

看下面的 JSP 页面。

程序清单：codes\02\2.3\basicSyntax\comment.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title> 注释示例 </title>
</head>
<body>
注释示例
<!-- 增加 JSP 注释 -->
<%-- JSP 注释部分 --%>
<!-- 增加 HTML 注释 -->
<!-- HTML 注释部分 -->
</body>
</html>
```

上面的页面中粗体字代码是 JSP 注释，其他注释都是 HTML 注释。在浏览器中浏览该页面，并查看页面源代码，页面的源代码如下：

```
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title> 注释示例 </title>
</head>
<body>
注释示例
<!-- 增加 JSP 注释 -->

<!-- 增加 HTML 注释 -->
<!-- HTML 注释部分 -->
</body>
</html>
```

在上面的源代码中可看到，HTML 的注释可以通过源代码查看到，但 JSP 的注释是无法通过源代码查看到的。这表明 JSP 注释不会被发送到客户端。

2.4 JSP 声明

JSP 声明用于声明变量和方法。在 JSP 声明中声明方法看起来很特别，似乎不需要定义类就可直接定义方法，方法似乎可以脱离类独立存在。实际上，JSP 声明将会转换成对应 Servlet 的成员变量或成员方法，因此 JSP 声明依然符合 Java 语法。

JSP 声明的语法格式如下：

```
<%! 声明部分 %>
```

看下面使用 JSP 声明的示例页面。

程序清单：codes\02\2.3\basicSyntax\declare.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 声明示例 </title>
</head>
<!-- 下面是 JSP 声明部分 -->
<%!
//声明一个整型变量
public int count;
//声明一个方法
public String info()
{
      return "hello";
}
%>
<body>
<%
//将 count 的值输出后再加 1
out.println(count++);
%>
<br/>
<%
//输出 info()方法的返回值
out.println(info());
%>
</body>
</html>
```

在浏览器中测试该页面时，可以看到正常输出了 count 值，每刷新一次，count 值将加 1，同时也可以看到正常输出了 info 方法的返回值。

上面的粗体字代码部分声明了一个整型变量和一个普通方法，表面上看起来这个变量和方法不属于任何类，似乎可以独立存在，但这只是一个假象。打开 Tomcat 的 work\Catalina\localhost\basicSyntax\org\apache\jsp 目录下 declare_jsp.java 文件，看到如下代码片段：

```
public final class declare_jsp extends org.apache.jasper.runtime.HttpJspBase
      implements org.apache.jasper.runtime.JspSourceDependent {
      //声明一个整型变量
      public int count;
      //声明一个方法
      public String info()
      {
            return "hello";
      }
      ...
}
```

上面的粗体字代码与 JSP 页面的声明部分完全对应，这表明 JSP 页面的声明部分将转换成对应 Servlet 的成员变量或成员方法。

提示：

由于 JSP 声明语法定义的变量和方法对应于 Servlet 类的成员变量和方法，所以 JSP 声明部分定义的变量和方法可以使用 private、public 等访问控制符修饰，也可使用 static 修饰，

将其变成类属性和类方法。但不能使用 abstract 修饰声明部分的方法，因为抽象方法将导致 JSP 对应 Servlet 变成抽象类，从而导致无法实例化。

打开多个浏览器，甚至可以在不同的机器上打开浏览器来刷新该页面，将发现所有客户端访问的 count 值是连续的，即所有客户端共享了同一个 count 变量。这是因为：JSP 页面会编译成一个 Servlet 类，每个 Servlet 在容器中只有一个实例；在 JSP 中声明的变量是成员变量，成员变量只在创建实例时初始化，该变量的值将一直保存，直到实例销毁。

值得注意的是，info()的值也可正常输出。因为 JSP 声明的方法其实是在 JSP 编译中生成的 Servlet 的实例方法——Java 里的方法是不能独立存在的，即使在 JSP 页面中也不行。

JSP 声明中独立存在的方法，只是一种假象。

2.5 输出 JSP 表达式

JSP 提供了一种输出表达式值的简单方法，输出表达式值的语法格式如下：

```
<%=表达式%>
```

看下面的 JSP 页面，该页面使用输出表达式的方式输出变量和方法返回值。

程序清单：codes\02\2.3\basicSyntax\outputEx.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 输出表达式值 </title>
</head>
<%!
public int count;

public String info()
{
      return "hello";
}
%>
<body>
<!-- 使用表达式输出变量值 -->
<%=count++%>
<br/>
<!--  使用表达式输出方法返回值 -->
<%=info()%>
</body>
</html>
```

上面的页面中粗体字代码使用输出表达式的语法代替了原来的 out.println 输出语句，该页面的执行效果与前一个页面的执行效果没有区别。由此可见，输出表达式将转换成 Servlet 里的输出语句。

输出表达式语法后不能有分号。

2.6 JSP 脚本

以前 JSP 脚本的应用非常广泛，因此 JSP 脚本里可以包含任何可执行的的 Java 代码。通常来说，

所有可执行性 Java 代码都可通过 JSP 脚本嵌入 HTML 页面。看下面使用 JSP 脚本的示例程序。

程序清单：codes\02\2.3\basicSyntax\scriptlet.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 小脚本测试 </title>
</head>
<body>
<table bgcolor="#9999dd" border="1" width="300px">
<!-- Java 脚本，这些脚本会对 HTML 的标签产生作用 -->
<%
for(int i = 0; i < 10; i++)
{
%>
      <!-- 上面的循环将控制<tr>标签循环 -->
      <tr>
            <td>循环值:</td>
            <td><%=i%></td>
      </tr>
<%
}
%>
<table>
</body>
</html>
```

上面的页面中粗体字代码就是使用 JSP 脚本的代码，这些代码可以控制页面中静态内容。上面例子程序将<tr.../>标签循环 10 次，即生成一个 10 行的表格，并在表格中输出表达式值。

在浏览器中浏览该页面，将看到如图 2.5 所示的效果。

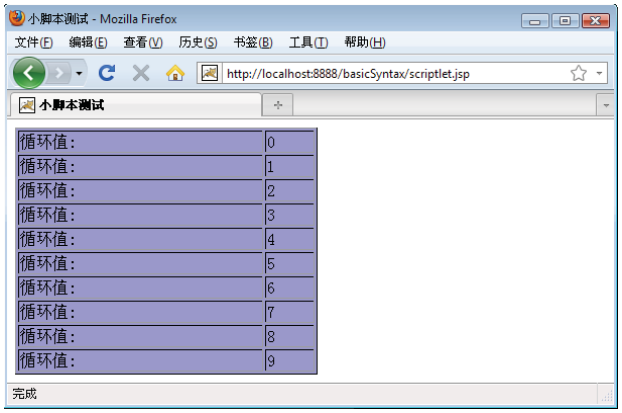

图 2.5　使用脚本动态生成 10 行

接下来我们打开 Tomcat 的 work\Catalina\localhost\basicSyntax\org\apache\jsp 路径下的 scriptlet_jsp.java 文件，将看到如下代码片段：

```
public final class scriptlet_jsp extends org.apache.jasper.runtime.HttpJspBase
      implements org.apache.jasper.runtime.JspSourceDependent {
      ...
      public void _jspService(HttpServletRequest request, HttpServletResponse response)
      throws java.io.IOException, ServletException {
            ...
            out.write("\r\n");
            out.write("\r\n");
            out.write("\r\n");
            out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Transitional//EN
            \"\r\n");
            out.write("\t\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd\">\r\n");
```

```
            out.write("<html xmlns=\"http://www.w3.org/1999/xhtml\">\r\n");
            out.write("<head>\r\n");
            out.write("\t<title> 小脚本测试 </title>\r\n");
            out.write("\t<meta name=\"website\" content=\"http://www.crazyit.org\" >\r\n");
            out.write("</head>\r\n");
            out.write("<body>\r\n");
            out.write("<table bgcolor=\"#9999dd\" border=\"1\" width=\"300px\">\r\n");
            out.write("<!-- Java 脚本，这些脚本会对 HTML 的标签产生作用 -->\r\n");
            for(int i = 0 ; i < 10 ; i++)
            {
                out.write("\r\n");
                out.write("\t<!-- 上面的循环将控制<tr>标签循环 -->\r\n");
                out.write("\t<tr>\r\n");
                out.write("\t\t<td>循环值:</td>\r\n");
                out.write("\t\t<td>");
                out.print(i);
                out.write("</td>\r\n");
                out.write("\t</tr>\r\n");
            }
            out.write("\r\n");
            out.write("<table>\r\n");
            out.write("</body>\r\n");
            out.write("</html>");
            ...
        }
    }
```

上面的代码片段中粗体字代码完全对应于 scriptlet.jsp 页面中的小脚本部分。由上面代码片段可以看出，JSP 脚本将转换成 Servlet 里_ jspService 方法的可执行性代码。这意味着在 JSP 小脚本部分也可以声明变量，但在 JSP 脚本部分声明的变量是局部变量，但不能使用 private、public 等访问控制符修饰，也不可使用 static 修饰。

提示：

实际上不仅 JSP 小脚本部分会转换成_ jspService 方法里的可执行性代码，JSP 页面里的所有静态内容都将由_ jspService 方法里输出语句来输出，这就是 JSP 脚本可以控制 JSP 页面中静态内容的原因。由于 JSP 脚本将转换成_ jspService 方法里的可执行性代码，而 Java 语法不允许在方法里定义方法，所以 JSP 脚本里不能定义方法。

因为 JSP 脚本中可以放置任何可执行性语句，所以可以充分利用 Java 语言的功能，例如连接数据库和执行数据库操作。看下面的 JSP 页面执行数据库查询。

程序清单：codes\02\2.3\basicSyntax\connDb.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.sql.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 小脚本测试 </title>
</head>
<body>
<%
//注册数据库驱动
Class.forName("com.mysql.jdbc.Driver");
//获取数据库连接
Connection conn = DriverManager.getConnection(
    "jdbc:mysql://localhost:3306/javaee","root","32147");
//创建 Statement
Statement stmt = conn.createStatement();
//执行查询
ResultSet rs = stmt.executeQuery("select * from news_inf");
%>
```

```
<table bgcolor="#9999dd" border="1" width="300">
<%
//遍历结果集
while(rs.next())
{%>
      <tr>
            <!-- 输出结果集 -->
            <td><%=rs.getString(1)%></td>
            <td><%=rs.getString(2)%></td>
      </tr>
<%}%>
<table>
</body>
</html>
```

上面程序中的粗体字脚本执行了连接数据库，执行 SQL 查询，并使用输出表达式语法来输出查询结果。在浏览器中浏览该页面，将看到如图 2.6 所示的效果。

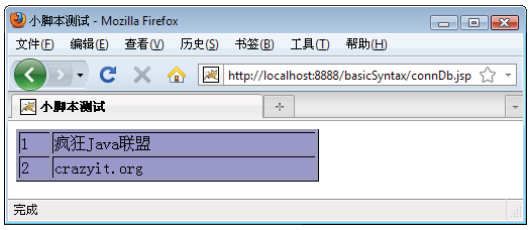

图 2.6　JSP 脚本查询数据库

上面的页面执行 SQL 查询需要使用 MySQL 驱动程序，所以读者应该将 MySQL 驱动的 JAR 文件放在 Tomcat 的 lib 路径下（所有 Web 应用都可使用 MySQL 驱动），或者将 MySQL 驱动复制到该 Web 应用的 WEB-INF/lib 路径下（只有该 Web 应用可使用 MySQL 驱动）。除此之外，由于本 JSP 需要查询 javaee 数据库下的 newsinf 数据表，所以不要忘记了将 codes\01 路径下的 test.sql 导入数据库。

2.7　JSP 的 3 个编译指令

JSP 的编译指令是通知 JSP 引擎的消息，它不直接生成输出。编译指令都有默认值，因此开发人员无须为每个指令设置值。

常见的编译指令有如下三个。

- page：该指令是针对当前页面的指令。
- include：用于指定包含另一个页面。
- taglib：用于定义和访问自定义标签。

使用编译指令的语法格式如下：

```
<%@ 编译指令名 属性名="属性值"…%>
```

下面主要介绍 page 和 include 指令，关于 taglib 指令，将在自定义标签库处详细讲解。

➤➤2.7.1　page 指令

page 指令通常位于 JSP 页面的顶端，一个 JSP 页面可以使用多条 page 指令。page 指令的语法格式如下：

```
<%@page
[language="Java"]
[extends="package.class"]
[import="package.class|package.*,…"]
[session="true|false"]
```

```
[buffer="none | 8KB | size Kb"]
[autoFlush="true | false"]
[isThreadSafe="true | false"]
[info="text"]
[errorPage="relativeURL"]
[contentType="mimeType[;charset=characterSet]" | "text/html;charSet=ISO-8859-1"]
[pageEncoding="ISO-8859-1"]
[isErrorPage="true | false"]
%>
```

下面依次介绍 page 指令各属性的意义。

- language：声明当前 JSP 页面使用的脚本语言的种类，因为页面是 JSP 页面，该属性的值通常都是 java，该属性的默认值也是 java，所以通常无须设置。
- extends：指定 JSP 页面编译所产生的 Java 类所继承的父类，或所实现的接口。
- import：用来导入包。下面几个包是默认自动导入的，不需要显式导入。默认导入的包有：java.lang.*、javax.servlet.*、javax.servlet.jsp.*、javax.servlet.http.*。
- session：设定这个 JSP 页面是否需要 HTTP Session。
- buffer：指定输出缓冲区的大小。输出缓冲区的 JSP 内部对象：out 用于缓存 JSP 页面对客户浏览器的输出，默认值为 8KB，可以设置为 none，也可以设置为其他的值，单位为 Kb。
- autoFlush：当输出缓冲区即将溢出时，是否需要强制输出缓冲区的内容。设置为 true 时为正常输出；如果设置为 false，则会在 buffer 溢出时产生一个异常。
- info：设置该 JSP 程序的信息，也可以看做其说明，可以通过 Servlet.getServletInfo()方法获取该值。如果在 JSP 页面中，可直接调用 getServletInfo()方法获取该值，因为 JSP 页面的实质就是 Servlet。
- errorPage：指定错误处理页面。如果本页面产生了异常或者错误，而该 JSP 页面没有对应的处理代码，则会自动调用该属性所指定的 JSP 页面。

因为 JSP 内建了异常机制支持，所以 JSP 可以不处理异常，即使是 checked 异常。

- isErrorPage：设置本 JSP 页面是否为错误处理程序。如果该页面本身已是错误处理页面，则通常无须指定 errorPage 属性。
- contentType：用于设定生成网页的文件格式和编码字符集，即 MIME 类型和页面字符集类型，默认的 MIME 类型是 text/html；默认的字符集类型为 ISO-8859-1。
- pageEncoding：指定生成网页的编码字符集。

从 2.6 节中执行数据库操作的 JSP 页面中可以看出，在 codes\02\2.3\jspPrinciple\connDb.jsp 页面的头部，使用了两条 page 指令：

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.sql.*" %>
```

其中第二条指令用于导入本页面中使用的类，如果没有通过 page 指令的 import 属性导入这些类，则需在脚本中使用全限定类名——即必须带包名。可见，此处的 import 属性类似于 Java 程序中的 import 关键字的作用。

如果删除第二条 page 指令，则执行效果如图 2.7 所示。

看下面的 JSP 页面，该页面使用 page 指令的 info 属性指定了 JSP 页面的描述信息，又使用 getServletInfo()方法输出该描述信息。

程序清单：codes\02\2.7\directive\jspInfo.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!-- 指定 info 信息 -->
<%@ page info="this is a jsp"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 测试 page 指令的 info 属性 </title>
</head>
<body>
<!--  输出 info 信息 -->
<%=getServletInfo()%>
</body>
</html>
```

以上页面的第一段粗体字代码设置了 info 属性，用于指定该 JSP 页面的描述信息；第二段粗体字代码使用了 getServletInfo()方法来访问该描述信息。

在浏览器中执行该页面，将看到如图 2.8 所示的效果。

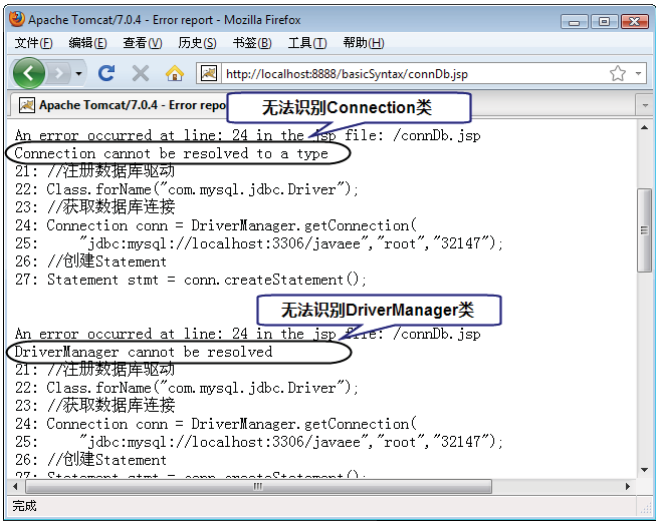

图 2.7　不使用 import 属性导包的出错效果

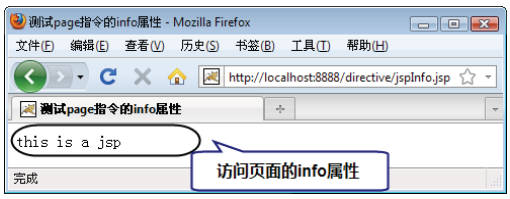

图 2.8　测试 page 指令的 info 属性

errorPage 属性的实质是 JSP 的异常处理机制，JSP 脚本不要求强制处理异常，即使该异常是 checked 异常。如果 JSP 页面在运行中抛出未处理的异常，系统将自动跳转到 errorPage 属性指定的页面；如果 errorPage 没有指定错误页面，系统则直接把异常信息呈现给客户端浏览器——这是所有的开发者都不愿意见到的场景。

看下面的 JSP 页面，该页面设置了 page 指令的 errorPage 属性，该属性指定了当本页面发生异常时的异常处理页面。

程序清单：codes\02\2.7\directive\errorTest.jsp

```
<%@ page contentType="text/html; charset=GBK"
      language="java" errorPage="error.jsp" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> new document </title>
</head>
<body>
<%
//下面代码将出现运行时异常
int a = 6;
int b = 0;
int c = a / b;
%>
</body>
</html>
```

以上页面的粗体字代码指定 errorTest.jsp 页面的错误处理页面是 error.jsp。下面是 error.jsp 页面，

该页面本身是错误处理页面，因此将 isErrorPage 设置成 true。

程序清单：codes\02\2.7\directive\error.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java"
    isErrorPage="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> 错误提示页面 </title>
</head>
<body>
<!-- 提醒客户端系统出现异常 -->
系统出现异常<br/>
</body>
</html>
```

上面页面的粗体字代码指定 error.jsp 页面是一个错误处理页面。在浏览器中浏览 errorTest.jsp 页面的效果如图 2.9 所示。

提示：

有些读者使用 Internet Explorer 浏览器时可能无法看到如图 2.9 所示的效果，而是看到代号为 500 的错误页面，这是 Internet Explorer 浏览器“自作聪明”的结果，读者可以选择更换 FireFox 浏览器。如果坚持使用 Internet Explorer 浏览器，则请单击 Internet Explorer 浏览器的“工具”主菜单的“选项”菜单项，并打开“Internet 选项”对话框的“高级”标签页，然后取消选择“显示友好 HTTP 错误信息”复选框即可；如图 2.10 所示。

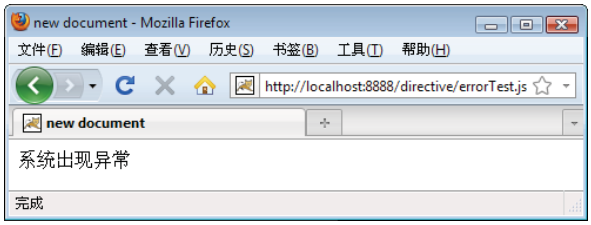

图 2.9 设置 errorPage 属性的效果

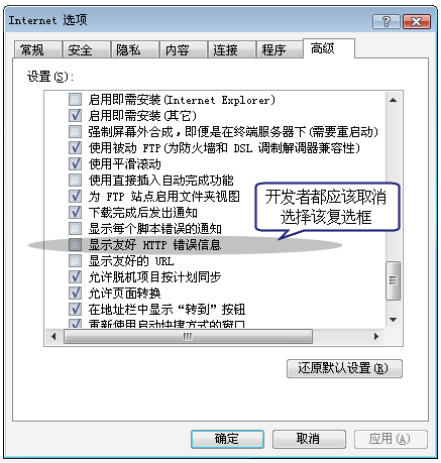

图 2.10 取消 IE 的“显示友好 HTTP 错误信息”复选框

如果将前一个页面中 page 指令的 errorPage 属性删除，再次通过浏览器浏览该页面，执行效果如图 2.11 所示。

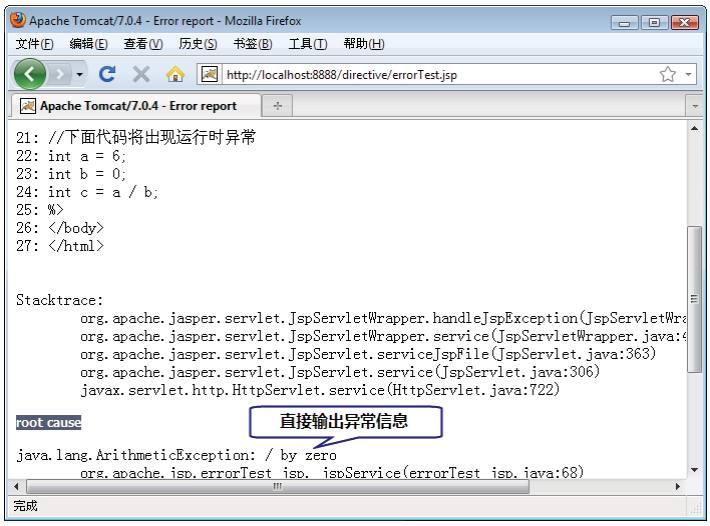

图 2.11　没有设置 errorPage 属性的效果

可见，使用 errorPage 属性控制异常处理的效果在表现形式上要好得多。

关于 JSP 异常，本章在介绍 exception 内置对象时还会有更进一步的解释。

2.7.2　include 指令

使用 include 指令，可以将一个外部文件嵌入到当前 JSP 文件中，同时解析这个页面中的 JSP 语句（如果有的话）。这是个静态的 include 语句，它会把目标页面的其他编译指令也包含进来，但动态 include 则不会。

include 既可以包含静态的文本，也可以包含动态的 JSP 页面。静态的 include 编译指令会将被包含的页面加入本页面，融合成一个页面，因此被包含页面甚至不需要是一个完整的页面。

include 编译指令的语法如下：

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

如果被嵌入的文件经常需要改变，建议使用<jsp:include>操作指令，因为它是动态的 include 语句。

下面的页面是使用静态导入的示例代码。

程序清单：codes\02\2.7\directive\staticInclude.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title> 静态 include 测试 </title>
</head>
<body>
<!-- 使用 include 编译指定导入页面 -->
<%@include file="errorTest.jsp"%>
</body>
</html>
```

以上页面中粗体字代码使用静态导入的语法将 scriptlet.jsp 页面导入本页，该页面的执行效果与 scriptlet.jsp 的执行效果相同。

查看 Tomcat 的 work\Catalina\localhost\directive\org\apache\jsp 路径下的 staticInclude_jsp.java 文件，从 staticInclude.jsp 编译后的源代码可看到，staticInclude.jsp 页面已经完全将 errorTest.jsp 的代码融入到本页面中。下面是 staticInclude_jsp.java 文件的片段：

```
out.write("<table bgcolor=\"#9999dd\" border=\"1\" width=\"300px\">\r\n");
```

```
out.write("<!-- Java 脚本，这些脚本会对 HTML 的标签产生作用 -->\r\n");
for(int i = 0 ; i < 10 ; i++)
{
      out.write("\r\n");
      out.write("\t<!-- 上面的循环将控制<tr>标签循环 -->\r\n");
      out.write("\t<tr>\r\n");
      out.write("\t\t<td>循环值:</td>\r\n");
      out.write("\t\t<td>");
      out.print(i);
      out.write("</td>\r\n");
      out.write("\t</tr>\r\n");
}
```

上面这些页面代码并不是由 staticInclude.jsp 页面所生成的，而是由 scriptlet.jsp 页面生成的。也就是说，scriptlet.jsp 页面的内容被完全融入 staticImport.jsp 页面所生成的 Servlet 中，这就是静态包含意义：包含页面在编译时将完全包含了被包含页面的代码。

需要指出的是，静态包含还会将被包含页面的编译指令也包含进来，如果两个页面的编译指令冲突，那么页面就会出错。

2.8 JSP 的 7 个动作指令

动作指令与编译指令不同，编译指令是通知 Servlet 引擎的处理消息，而动作指令只是运行时的动作。编译指令在将 JSP 编译成 Servlet 时起作用；而处理指令通常可替换成 JSP 脚本，它只是 JSP 脚本的标准化写法。

JSP 动作指令主要有如下 7 个。

- jsp:forward：执行页面转向，将请求的处理转发到下一个页面。
- jsp:param：用于传递参数，必须与其他支持参数的标签一起使用。
- jsp:include：用于动态引入一个 JSP 页面。
- jsp:plugin：用于下载 JavaBean 或 Applet 到客户端执行。
- jsp:useBean：创建一个 JavaBean 的实例。
- jsp:setProperty：设置 JavaBean 实例的属性值。
- jsp:getProperty：输出 JavaBean 实例的属性值。

下面依次讲解这些动作指令。

2.8.1 forward 指令

forward 指令用于将页面响应转发到另外的页面。既可以转发到静态的 HTML 页面，也可以转发到动态的 JSP 页面，或者转发到容器中的 Servlet。

JSP 的 forward 指令的格式如下。

对于 JSP 1.0，使用如下语法：

```
<jsp:forward page="{relativeURL|<%=expression%>}"/>
```

对于 JSP 1.1 以上规范，可使用如下语法：

```
<jsp:forward page="{relativeURL|<%=expression%>}">
     {<jsp:param.../>}
</jsp:forward>
```

第二种语法用于在转发时增加额外的请求参数。增加的请求参数的值可以通过 HttpServletRequest 类的 getParameter()方法获取。

下面示例页面使用了 forward 动作指令来转发用户请求。

程序清单：codes\02\2.7\directive\jsp-forward.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> forward 的原始页 </title>
</head>
<body>
<h3>forward 的原始页</h3>
<jsp:forward page="forward-result.jsp">
      <jsp:param name="age" value="29"/>
</jsp:forward>
</body>
</html>
```

这个 JSP 页面非常简单，它包含了简单的 title 信息，页面中也包含了简单的文本内容，页面的粗体字代码则将客户端请求转发到 forward-result.jsp 页面，转发请求时增加了一个请求参数：参数名为 age，参数值为 29。

在 forward-result.jsp 页面中，使用 request 内置对象（request 内置对象是 HttpServletRequest 的实例，关于 request 的详细信息参看下一节）来获取增加的请求参数值。

程序清单：codes\02\2.7\directive\forward-result.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title>forward 结果页</title>
</head>
<body>
<!-- 使用 request 内置对象获取 age 参数的值 -->
<%=request.getParameter("age")%>
</body>
</html>
```

forward-result.jsp 页面中的粗体字代码设置了 title 信息，并输出了 age 请求参数的值，在浏览器中访问 jsp-forward.jsp 页面的执行效果如图 2.12 所示。

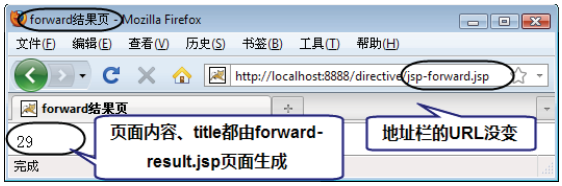

图 2.12　forward 动作指令的效果

从图 2.12 中可以看出，执行 forward 指令时，用户请求的地址依然没有发生改变，但页面内容却完全变为被 forward 目标页的内容。

执行 forward 指令转发请求时，客户端的请求参数不会丢失。看下面表单提交页面的例子，该页面没有任何动态的内容，只是一个静态的表单页，作用是将请求参数提交到 jsp-forward.jsp 页。

程序清单：codes\02\2.7\directive\form.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 提交 </title>
</head>
<body>
```

```
<!-- 表单提交页面 -->
<form id="login" method="post" action="jsp-forward.jsp">
<input type="text" name="username">
<input type="submit" value="login">
</form>
</body>
</html>
```

修改 forward-result.jsp 页，增加输出表单参数的代码，也就是在 forward-result.jsp 页面上增加如下代码：

```
<!-- 输出 username 请求参数的值 -->
<%=request.getParameter("username")%>
```

在表单提交页面中的文本框中输入任意字符串后提交该表单，即可看到如图 2.13 所示的执行效果。

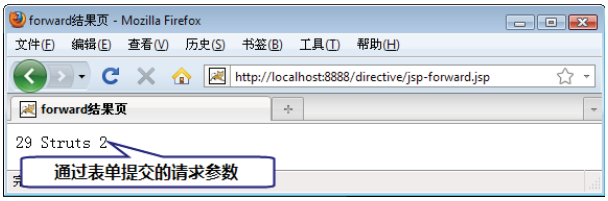

图 2.13　执行 forward 时不会丢失请求参数

从图 2.13 中可看到，forward-result.jsp 页面中不仅可以输出 forward 指令增加的请求参数，还可以看到表单里 username 表单域对应的请求参数，这表明执行 forward 时不会丢失请求参数。

提示：

从表面上看，<jsp:forward.../>指令给人一种感觉：它是将用户请求“转发”到了另一个新页面，但实际上，<jsp:forward.../>并没有重新向新页面发送请求，它只是完全采用了新页面来对用户生成响应——请求依然是一次请求，所以请求参数、请求属性都不会丢失。

➤➤2.8.2 include 指令

include 指令是一个动态 include 指令，也用于包含某个页面，它不会导入被 include 页面的编译指令，仅仅将被导入页面的 body 内容插入本页面。

下面是 include 动作指令的语法格式：

```
<jsp:include page="{relativeURL | <%=expression%>}" flush="true"/>
```

或者

```
<jsp:include page="{relativeURL | <%=expression%>}" flush="true">
    <jsp:param name="parameterName" value="patameterValue"/>
</jsp:include>
```

flush 属性用于指定输出缓存是否转移到被导入文件中。如果指定为 true，则包含在被导入文件中；如果指定为 false，则包含在原文件中。对于 JSP 1.1 旧版本，只能设置为 false。

对于第二种语法格式，则可在被导入页面中加入额外的请求参数。

下面的页面使用了动态导入语法来导入指定 JSP 页面。

程序清单：codes\02\2.7\directive\jsp-include.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> jsp-include 测试 </title>
</head>
<body>
<!-- 使用动态 include 指令导入页面 -->
<jsp:include page="scriptlet.jsp" />
</body>
</html>
```

以上页面中粗体字代码使用了动态导入语法来导入了 scriptlet.jsp。表面上看，该页面的执行效果与使用静态 include 导入的页面并没有什么不同。但查看 jsp-include.jsp 页面生成 Servlet 的源代码，可以看到如下片段：

```
//使用页面输出流，生成 HTML 标签内容
out.write("</head>\r\n");
out.write("<body>\r\n");
out.write("<!-- 使用动态 include 指令导入页面 -->\r\n");
org.apache.jasper.runtime.JspRuntimeLibrary.include(request
    , response, "scriptlet.jsp", out, false);
out.write("\r\n");
out.write("</body>\r\n");
```

以上代码片段中粗体字代码显示了动态导入的关键：动态导入只是使用一个 include 方法来插入目标页面的内容，而不是将目标页面完全融入本页面中。

归纳起来，静态导入和动态导入有如下三点区别：

- 静态导入是将被导入页面的代码完全融入，两个页面融合成一个整体 Servlet；而动态导入则在 Servlet 中使用 include 方法来引入被导入页面的内容。

- 静态导入时被导入页面的编译指令会起作用；而动态导入时被导入页面的编译指令则失去作用，只是插入被导入页面的 body 内容。
- 动态包含还可以增加额外的参数。

除此之外，执行 include 动态指令时，还可增加额外的请求参数，如下面 JSP 页面所示。

程序清单：codes\02\2.7\directive\jsp-include2.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> jsp-include 测试 </title>
</head>
<body>
<jsp:include page="forward-result.jsp" >
      <jsp:param name="age" value="32"/>
</jsp:include>
</body>
</html>
```

在上面的 JSP 页面中的粗体字代码同样使用<jsp:include.../>指令包含页面，而且在 jsp:include 指令中还使用 param 指令传入参数，该参数可以在 forward-result.jsp 页面中使用 request 对象获取。

forward-result.jsp 前面已经给出，此处不再赘述。页面执行的效果如图 2.14 所示。

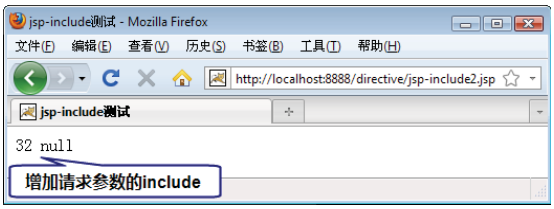

图 2.14　增加请求参数的 include

提示：实际上，forward 动作指令和 include 动作指令十分相似（它们的语法就很相似），它们都采用方法来引入目标页面，通过查看 JSP 页面所生成 Servlet 代码可以得出：forward 指令使用_jspx_page_context 的 forward()方法来引入目标页面，而 include 指令则使用通过 JspRuntimeLibrary 的 include()方法来引入目标页面。区别在于，执行 forward 时，被 forward 的页面将完全代替原有页面；而执行 include 时，被 include 的页面只是插入原有页面。简而言之：forward 拿目标页面代替原有页面，而 include 则拿目标页面插入原有页面。

2.8.3　useBean、setProperty、getProperty 指令

这三个指令都是与 JavaBean 相关的指令，其中 useBean 指令用于在 JSP 页面中初始化一个 Java 实例；setProperty 指令用于为 JavaBean 实例的属性设置值；getProperty 指令用于输出 JavaBean 实例的属性。

如果多个 JSP 页面中需要重复使用某段代码，我们可以把这段代码定义成 Java 类的方法，然后让多个 JSP 页面调用该方法即可，这样可以达到较好的代码复用。

useBean 的语法格式如下：

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

其中，id 属性是 JavaBean 的实例名，class 属性确定 JavaBean 的实现类。scope 属性用于指定 JavaBean

实例的作用范围，该范围有以下 4 个值。

- page：该 JavaBean 实例仅在该页面有效。
- request：该 JavaBean 实例在本次请求有效。
- session：该 JavaBean 实例在本次 session 内有效。
- application：该 JavaBean 实例在本应用内一直有效。

提示：
本章后面有关于这 4 个作用范围的详细介绍。

setProperty 指令的语法格式如下：

```
<jsp:setProperty name="BeanName" proterty="propertyName" value="value"/>
```

其中，name 属性确定需要设定 JavaBean 的实例名；property 属性确定需要设置的属性名；value 属性则确定需要设置的属性值。

getProperty 的语法格式如下：

```
<jsp:getProperty name="BeanName" proterty="propertyName" />
```

其中，name 属性确定需要输出的 JavaBean 的实例名；property 属性确定需要输出的属性名。

下面的 JSP 页面示范了如何使用这 3 个动作指令来操作 JavaBean。

程序清单：codes\02\2.7\directive\beanTest.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> Java Bean 测试 </title>
</head>
<body>
<!-- 创建 lee.Person 的实例，该实例的实例名为 p1 -->
<jsp:useBean id="p1" class="lee.Person" scope="page"/>
<!-- 设置 p1 的 name 属性值 -->
<jsp:setProperty name="p1" property="name" value="wawa"/>
<!-- 设置 p1 的 age 属性值 -->
<jsp:setProperty name="p1" property="age" value="23"/>
<!-- 输出 p1 的 name 属性值 -->
<jsp:getProperty name="p1" property="name"/><br/>
<!-- 输出 p1 的 age 属性值 -->
<jsp:getProperty name="p1" property="age"/>
</body>
</html>
```

以上页面中粗体字代码示范了使用 useBean、setProperty 和 getProperty 来操作 JavaBean 的方法。

对于上面的 JSP 页面中的 setProperty 和 getProperty 标签而言，它们都要求根据属性名来操作 JavaBean 的属性。实际上 setProperty 和 getProperty 要求的属性名，与 Java 类中定义的属性有一定的差别，例如 setProperty 和 getProperty 需要使用 name 属性，但 JavaBean 中是否真正定义了 name 属性并不重要，重要的是在 JavaBean 中提供了 setName()和 getName()方法即可。事实上，当页面使用 setProperty 和 getProperty 标签时，系统底层就是调用 setName()和 getName()方法来操作 Person 实例的属性的。

下面是 Person 类的源代码。

程序清单：codes\02\2.7\directive\WEB-INF\src\lee\Person.java

```
public class Person
{
      private String name;
      private int age;
      //无参数的构造器
```

```
        public Person()
        {
        }
        //初始化全部属性的构造器
        public Person(String name , int age)
        {
              this.name = name;
              this.age = age;
        }
        //name 属性的 setter 和 getter 方法
        public void setName(String name)
        {
              this.name = name;
        }
        public String getName()
        {
              return this.name;
        }
        //age 属性的 setter 和 getter 方法
        public void setAge(int age)
        {
              this.age = age;
        }
        public int getAge()
        {
              return this.age;
        }
}
```

上面的 Person.java 只是源文件，我们将该文件放在 Web 应用的 WEB-INF/src 路径下，实际上 Java 源文件对 Web 应用不起作用，所以我们会使用 Ant 来编译它，并将编译得到的二进制文件放入 WEB-INF/classes 路径下。而且，当我们为 Web 应用提供了新的 class 文件后，必须重启该 Web 应用，让它可以重新加载这些新的 class 文件。

该页面的执行效果如图 2.15 所示。

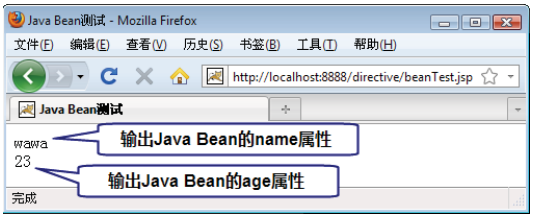

图 2.15　操作 JavaBean

对于上面三个标签完全可以不使用，将 beanTest.jsp 修改成如下代码，其内部的执行是完全一样的。

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> Java Bean 测试 </title>
</head>
<body>
<%
//实例化 JavaBean 实例，实现类为 lee.Person，该实例的实例名为 p1
Person p1 = new Person();
//将 p1 放置到 page 范围中
pageContext.setAttribute("p1" , p1);
//设置 p1 的 name 属性值
p1.setName("wawa");
//设置 p1 的 age 属性值
p1.setAge(23);
%>
```

```
<!-- 输出 p1 的 name 属性值 -->
<%=p1.getName()%><br/>
<!-- 输出 p1 的 age 属性值 -->
<%=p1.getAge()%>
</body>
</html>
```

使用 useBean 标签时，除在页面脚本中创建了 JavaBean 实例之外，该标签还会将该 JavaBean 实例放入指定 scope 中，所以我们通常还需要在脚本中将该 JavaBean 放入指定 scope 中，如下面的代码片段所示：

```
//将 p1 放入 page 的生存范围中
pageContext.setAttribute("p1" , p1);
//将 p1 放入 request 的生存范围中
request.setAttribute("p1",p1);
//将 p1 放入 session 的生存范围中
session.setAttribute("p1",p1);
//将 p1 放入 application 的生存范围中
application.setAttribute("p1",p1);
```

提示：关于 page、request、session 和 application 4 个生存范围请参看下一节介绍。

2.8.4 plugin 指令

plugin 指令主要用于下载服务器端的 JavaBean 或 Applet 到客户端执行。由于程序在客户端执行，因此客户端必须安装虚拟机。

提示：实际由于现在很少使用 Applet ,而且就算要使用 Applet ,我们完全可以使用支持 Applet 的 HTML 标签，所以 jsp:plugin 标签的使用场景并不多。因此为了节省篇幅起见，本书不再详细介绍 plugin 指令的用法。

2.8.5 param 指令

param 指令用于设置参数值，这个指令本身不能单独使用，因为单独的 param 指令没有实际意义。param 指令可以与以下三个指令结合使用。

- jsp:include
- jsp:forward
- jsp:plugin

当与 include 指令结合使用时，param 指令用于将参数值传入被导入的页面；当与 forward 指令结合使用时，param 指令用于将参数值传入被转向的页面；当与 plugin 指令结合使用时，则用于将参数传入页面中的 JavaBean 实例或 Applet 实例。

param 指令的语法格式如下：

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

关于 param 的具体使用，请参考前面的示例。

2.9 JSP 脚本中的 9 个内置对象

JSP 脚本中包含 9 个内置对象，这 9 个内置对象都是 Servlet API 接口的实例，只是 JSP 规范对它们进行了默认初始化（由 JSP 页面对应 Servlet 的_jspService()方法来创建这些实例）。也就是说，它们已经是对象，可以直接使用。9 个内置对象依次如下。

- application：javax.servlet.ServletContext 的实例，该实例代表 JSP 所属的 Web 应用本身，可用于 JSP 页面，或者在 Servlet 之间交换信息。常用的方法有 getAttribute(String attName)、setAttribute(String attName , String attValue)和 getInitParameter(String paramName)等。
- config：javax.servlet.ServletConfig 的实例，该实例代表该 JSP 的配置信息。常用的方法有 getInitParameter(String paramName)和 getInitParameternames()等方法。事实上，JSP 页面通常无须配置，也就不存在配置信息。因此，该对象更多地在 Servlet 中有效。
- exception：java.lang.Throwable 的实例，该实例代表其他页面中的异常和错误。只有当页面是错误处理页面，即编译指令 page 的 isErrorPage 属性为 true 时，该对象才可以使用。常用的方法有 getMessage()和 printStackTrace()等。
- out：javax.servlet.jsp.JspWriter 的实例，该实例代表 JSP 页面的输出流，用于输出内容，形成 HTML 页面。
- page：代表该页面本身，通常没有太大用处。也就是 Servlet 中的 this，其类型就是生成的 Servlet 类，能用 page 的地方就可用 this。
- pageContext：javax.servlet.jsp.PageContext 的实例，该对象代表该 JSP 页面上下文，使用该对象可以访问页面中的共享数据。常用的方法有 getServletContext()和 getServletConfig()等。
- request：javax.servlet.http.HttpServletRequest 的实例，该对象封装了一次请求，客户端的请求参数都被封装在该对象里。这是一个常用的对象，获取客户端请求参数必须使用该对象。常用的方法有 getParameter(String paramName)、getParameterValues(String paramName)、setAttribute(String attrName,Object attrValue) 、 getAttribute(String attrName) 和 setCharacterEncoding(String env)等。
- response：javax.servlet.http.HttpServletResponse 的实例，代表服务器对客户端的响应。通常很少使用该对象直接响应，而是使用 out 对象，除非需要生成非字符响应。而 response 对象常用于重定向，常用的方法有 getOutputStream()、sendRedirect(java.lang.String location)等。
- session：javax.servlet.http.HttpSession 的实例，该对象代表一次会话。当客户端浏览器与站点建立连接时，会话开始；当客户端关闭浏览器时，会话结束。常用的方法有：getAttribute(String attrName)、setAttribute(String attrName, Object attrValue)等。

进入 Tomcat 的 work\Catalina\localhost\jspPrinciple\org\apache\jsp 路径下，打开任意一个 JSP 页面对应生成的 Servlet 类文件，看到如下代码片段：

```
public final class test_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
    ...
    //用于响应用户请求的方法
    public void _jspService(HttpServletRequest request, HttpServletResponse response)
        throws java.io.IOException, ServletException {
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
        JspWriter _jspx_out = null;
        PageContext _jspx_page_context = null;
```

```
            try {
                response.setContentType("text/html; charset=gb2312");
                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();
                ...
            }
        }
    }
```

几乎所有的 JSP 页面编译后 Servlet 类都有如上所示的结构，上面 Servlet 类的粗体字代码表明：request、response 两个对象是_jspService()方法的形参，当 Tomcat 调用该方法时会初始化这两个对象。而 page、pageContext、application、config、session、out 都是_jspService()方法的局部变量，由该方法完成初始化。

通过上面的代码不难发现 JSP 内置对象的实质：它们要么是_jspService()方法的形参，要么是_jspService()方法的局部变量，所以我们直接在 JSP 脚本（脚本将对应于 Servlet 的_jspService()方法部分）中调用这些对象，无须创建它们。

提示：

由于 JSP 内置对象都是在_jspService()方法中完成初始化的，因此只能在 JSP 脚本、JSP 输出表达式中使用这些内置对象。千万不要在 JSP 声明中使用它们！否则，系统将提示找不到这些变量。

当我们编写 JSP 页面时，一定不要仅停留在 JSP 页面本身来看问题，这样可能导致许多误解，导致我们无法理解 JSP 的运行方式。很多书籍上随意介绍这些对象，也是形成误解的原因之一。

细心的读者可能已经发现了：上面的代码中并没有 exception 内置对象，这与前面介绍的正好相符：只有当页面的 page 指令的 isErrorPage 属性为 true 时，才可使用 exception 对象。也就是说：只有异常处理页面对应 Servlet 时才会初始化 exception 对象。

➤➤2.9.1 application 对象

在介绍 application 对象之前，先简单介绍一些 Web 服务器的实现原理。虽然绝大部分读者都不需要、甚至不曾想过自己开发 Web 服务器，但了解一些 Web 服务器的运行原理，对于更好地掌握 JSP 知识将有很大的帮助。

虽然常把基于 Web 应用称为 B/S（Browser/Server）架构的应用，但其实 Web 应用一样是 C/S（Client/Server）结构的应用，只是这种应用的服务器是 Web 服务器，而客户端是浏览器。

现在我们抛开 Web 应用直接看 Web 服务器和浏览器，对于大部分浏览器而言，它通常负责完成三件事情：

（1）向远程服务器发送请求。

（2）读取远程服务器返回的字符串数据。

（3）负责根据字符串数据渲染出一个丰富多彩的页面。

提示：

实际上，浏览器是一个非常复杂的网络通信程序，它除了可以向服务器发送请求、读

取网络数据之外，最大的技术难点在于将 HTML 文本渲染成页面，建立 HTML 页面的 DOM 模型，支持 JavaScript 脚本程序等。通常浏览器有 Internet Explorer、FireFox、Opera 等，至于其他如 MyIE、傲游等浏览器可能只是对它们进行了简单的包装。

Web 服务器则负责接收客户端请求，每当接收到客户端连接请求之后，Web 服务器应该使用单独的线程为该客户端提供服务：接收请求数据、送回响应数据。图 2.16 显示了 Web 服务器的运行机制。

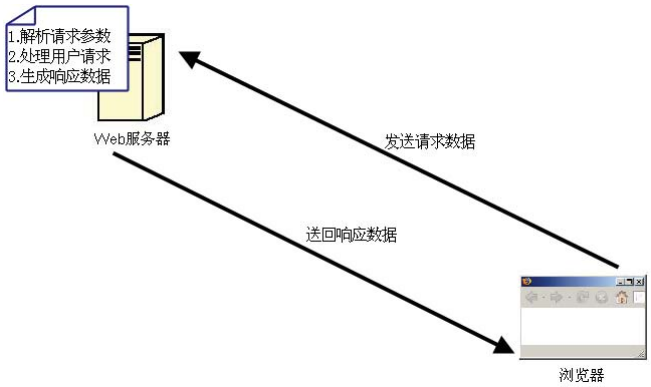

图 2.16　Web 服务器运行机制

如图 2.16 所示的应用架构总是先由客户端发送请求，服务器接收到请求后送回响应的数据，所以也将这种架构称做“请求/响应”架构。根据如图 2.16 所示的机制进行归纳，对于每次客户端请求而言，Web 服务器大致需要完成如下几个步骤：

① 启动单独的线程。
② 使用 I/O 流读取用户的请求数据。
③ 从请求数据中解析参数。
④ 处理用户请求。
⑤ 生成响应数据。
⑥ 使用 IO 流向客户端发送请求数据。

在上面 6 个步骤中，第 1、2 和 6 步是通用的，可以由 Web 服务器来完成，但第 3、4 和 5 步则存在差异：因为不同请求里包含的请求参数不同，处理用户请求的方式也不同，所生成的响应自然也不同。那么 Web 服务器到底如何执行第 3、4 和 5 步呢？

实际上，Web 服务器会调用 Servlet 的_jspService()方法来完成第 3、4 和 5 步，当我们编写 JSP 页面时，页面里的静态内容、JSP 脚本都会转换成_jspService()方法的执行代码，这些执行代码负责完成解析参数、处理请求、生成响应等业务功能，而 Web 服务器则负责完成多线程、网络通信等底层功能。

Web 服务器在执行了第 3 步解析到用户的请求参数之后，将需要通过这些请求参数来创建 HttpServletRequest、HttpServletResponse 等对象，作为调用_jspService()方法的参数，实际上一个 Web 服务器必须为 Servlet API 中绝大部分接口提供实现类。

从上面介绍可以看出，Web 应用里的 JSP 页面、Servlet 等程序都将由 Web 服务器来调用，JSP、Servlet 之间通常不会相互调用，这就产生了一个问题：JSP、Servlet 之间如何交换数据？

为了解决这个问题，几乎所有 Web 服务器（包括 Java、ASP、PHP、Ruby 等）都会提供 4 个类似 Map 的结构，分别是 application、session、request、page，并允许 JSP、Servlet 将数据放入这 4 个类似 Map 的结构中，并允许从这 4 个 Map 结构中取出数据。这 4 个 Map 结构的区别是范围不同。

- application：对于整个 Web 应用有效，一旦 JSP、Servlet 将数据放入 application 中，该数

据将可以被该应用下其他所有的 JSP、Servlet 访问。

- session：仅对一次会话有效，一旦 JSP、Servlet 将数据放入 session 中，该数据将可以被本次会话的其他所有的 JSP、Servlet 访问。
- request：仅对本次请求有效，一旦 JSP、Servlet 将数据放入 request 中，该数据将可以被该次请求的其他 JSP、Servlet 访问。
- page：仅对当前页面有效，一旦 JSP、Servlet 将数据放入 page 中，该数据只可以被当前页面的 JSP 脚本、声明部分访问。

就像现实生活中有两个人，他们的钱需要相互交换，但他们两个人又不能相互接触，那么只能让 A 把钱存入银行，而 B 从银行去取钱。因此，我们可以把 application、session、request 和 page 理解为类似银行的角色。

把数据放入 application、session、request 或 page 之后，就相当于扩大了该数据的作用范围，所以我们也认为 application、session、request 和 page 中的数据分别处于 application、session、request 和 page 范围之内。

JSP 中的 application、session、request 和 pageContext 4 个内置对象分别用于操作 application、session、request 和 page 范围中的数据。

application 对象代表 Web 应用本身，因此使用 application 来操作 Web 应用相关数据。application 对象通常有如下两个作用：

- 在整个 Web 应用的多个 JSP、Servlet 之间共享数据。
- 访问 Web 应用的配置参数。

1．让多个 JSP、Servlet 共享数据

application 通过 setAttribute(String attrName,Object value)方法将一个值设置成 application 的 attrName 属性，该属性的值对整个 Web 应用有效，因此该 Web 应用的每个 JSP 页面或 Servlet 都可以访问该属性，访问属性的方法为 getAttribute(String attrName)。

看下面的页面，该页面仅仅声明了一个整型变量，每次刷新该页面时，该变量值加 1，然后将该变量的值放入 application 内。下面是页面的代码。

程序清单：codes\02\2.9\jspObject\put-application.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title>application 测试</title>
</head>
<body>
<!-- JSP 声明 -->
<%!
int i;
%>
<!-- 将 i 值自加后放入 application 的变量内 -->
<%
application.setAttribute("counter",String.valueOf(++i));
%>
<!-- 输出 i 值 -->
<%=i%>
</body>
</html>
```

以上页面的粗体字代码实现了每次刷新该页面时，变量 i 都先自加，并被设置为 application 的 counter 属性的值，即每次 application 中的 counter 属性值都会加 1。

再看下面的 JSP 页面，该页面可以直接访问到 application 的 counter 属性值。

程序清单：codes\02\2.9\jspObject\get-application.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title>application 测试</title>
</head>
<body>
<!-- 直接输出 application 变量值 -->
<%=application.getAttribute("counter")%>
</body>
</html>
```

以上页面中粗体字代码直接输出 application 的 counter 属性值，虽然这个页面和 put-application.jsp 没有任何关系，但它一样可以访问到 application 的属性，因为 application 的属性对于整个 Web 应用的 JSP、Servlet 都是共享的。

在浏览器的地址栏中访问第一个 put-application.jsp 页面，经多次刷新后，看到如图 2.17 所示的页面。

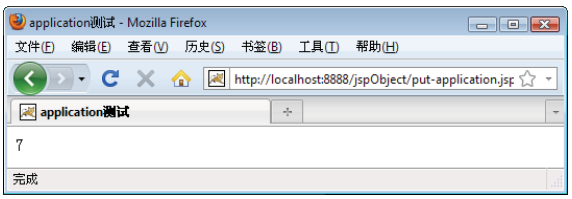

图 2.17　将变量值放入 application 中

访问 get-application.jsp 页面，也可看到类似于图 2.17 所示的效果，因为 get-application.jsp 页面可以访问 application 的 counter 属性值。

application 不仅可以用于两个 JSP 页面之间共享数据，还可以用于 Servlet 和 JSP 之间共享数据。我们可以把 application 理解成一个 Map 对象，任何 JSP、Servlet 都可以把某个变量放入 application 中保存，并为之指定一个属性名，而该应用里的其他 JSP、Servlet 就可以根据该属性名来得到这个变量。

下面的 Servlet 代码示范了如何在 Servlet 中访问 application 里的变量。

程序清单：codes\02\2.9\jspObject\WEB-INF\src\lee\GetApplication.java

```
@WebServlet(name="get-application",
      urlPatterns={"/get-application"})
public class GetApplication extends HttpServlet
{
      public void service(HttpServletRequest request,
            HttpServletResponse response)throws IOException
      {
            response.setContentType("text/html;charset=gb2312");
            PrintWriter out = response.getWriter();
            out.println("<html><head><title>");
            out.println("测试 application");
            out.println("</title></head><body>");
            ServletContext sc = getServletConfig().getServletContext();
            out.print("application 中当前的 counter 值为:");
            out.println(sc.getAttribute("counter"));
            out.println("</body></html>");
      }
```

```
}
```

由于在 Servlet 中并没有 application 内置对象，所以上面程序第一行粗体字代码显式获取了该 Web 应用的 ServletContext 实例，每个 Web 应用只有一个 ServletContext 实例，在 JSP 页面中可通过 application 内置对象访问该实例，而 Servlet 中则必须通过代码获取。程序第二行粗体字代码访问、输出了 application 中的 counter 变量。

该 Servlet 类同样需要编译成 class 文件才可使用，实际上该 Servlet 还使用了@WebServlet Annotation 进行部署，关于 Servlet 的用法请参看 2.10 节。编译 Servlet 时可能由于没有添加环境出现异常，如果安装了 Java EE 6 SDK，只需将 Java EE 6 SDK 路径的 javaee.jar 文件添加到 CLASSPATH 环境变量中；如果没有安装 Java EE SDK，可以将 Tomcat 7 的 lib 路径下的 jsp-api.jar、servlet-api.jar 两个文件添加到 CLASSPATH 环境变量中。

将 Servlet 部署在 Web 应用中，在浏览器中访问 Servlet，出现如图 2.18 所示的页面。

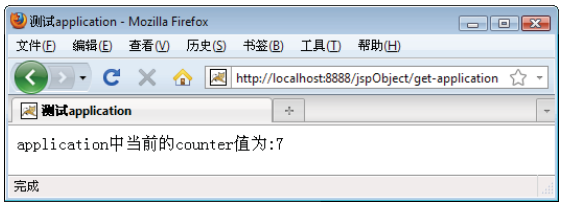

图 2.18　Servlet 访问 application 变量

最后要指出的是：虽然使用 application（即 ServletContext 实例）可以方便多个 JSP、Servlet 共享数据，但不要仅为了 JSP、Servlet 共享数据就将数据放入 application 中！由于 application 代表整个 Web 应用，所以通常只应该把 Web 应用的状态数据放入 application 里。

2．获得 Web 应用配置参数

application 还有一个重要用处：可用于获得 Web 应用的配置参数。看如下 JSP 页面，该页面访问数据库，但访问数据库所使用的驱动、URL、用户名及密码都在 web.xml 中给出。

程序清单：codes\02\2.9\jspObject\getWebParam.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.sql.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title>application 测试</title>
</head>
<body>
<%
//从配置参数中获取驱动
String driver = application.getInitParameter("driver");
//从配置参数中获取数据库 url
String url = application.getInitParameter("url");
//从配置参数中获取用户名
String user = application.getInitParameter("user");
//从配置参数中获取密码
String pass = application.getInitParameter("pass");
//注册驱动
Class.forName(driver);
//获取数据库连接
Connection conn = DriverManager.getConnection(url,user,pass);
//创建 Statement 对象
```

```
Statement stmt = conn.createStatement();
//执行查询
ResultSet rs = stmt.executeQuery("select * from news_inf");
%>
<table bgcolor="#9999dd" border="1" width="480">
<%
//遍历结果集
while(rs.next())
{
%>
      <tr>
            <td><%=rs.getString(1)%></td>
            <td><%=rs.getString(2)%></td>
      </tr>
<%
}
%>
<table>
</body>
</html>
```

上面的程序中粗体字代码使用 application 的 getInitParameter(String paramName)来获取 Web 应用的配置参数，这些配置参数应该在 web.xml 文件中使用 context-param 元素配置，每个<context-param…/>元素配置一个参数，该元素下有如下两个子元素。

- param-name：配置 Web 参数名。
- param-value：配置 Web 参数值。

web.xml 文件中使用<context-param…/>元素配置的参数对整个 Web 应用有效，所以也被称为 Web 应用的配置参数。与整个 Web 应用有关的数据，应该通过 application 对象来操作。

为了给 Web 应用配置参数，应在 web.xml 文件中增加如下片段。

程序清单：codes\02\2.9\jspObject\WEB-INF\web.xml

```
<!-- 配置第一个参数：driver -->
<context-param>
      <param-name>driver</param-name>
      <param-value>com.mysql.jdbc.Driver</param-value>
</context-param>
<!-- 配置第二个参数：url -->
<context-param>
      <param-name>url</param-name>
      <param-value>jdbc:mysql://localhost:3306/javaee</param-value>
</context-param>
<!-- 配置第三个参数：user -->
<context-param>
      <param-name>user</param-name>
      <param-value>root</param-value>
</context-param>
<!-- 配置第四个参数：pass -->
<context-param>
      <param-name>pass</param-name>
      <param-value>32147</param-value>
</context-param>
```

在浏览器中浏览 getWebParam.jsp 页面时，可看到数据库连接、数据查询完全成功。可见，使用 application 可以访问 Web 应用的配置参数。

通过这种方式，可以将一些配置信息放在 web.xml 文件中配置，避免使用硬编码方式写在代码中，从而更好地提高程序的移植性。

➤➤2.9.2 config 对象

config 对象代表当前 JSP 配置信息，但 JSP 页面通常无须配置，因此也就不存在配置信息。该对象在 JSP 页面中比较少用，但在 Servlet 中则用处相对较大，因为 Servlet 需要在 web.xml 文件中进行配置，可以指定配置参数。关于 Servlet 的使用将在 2.10 节介绍。

看如下 JSP 页面代码，该 JSP 代码使用了 config 的一个方法 getServletName()。

程序清单：codes\02\2.9\jspObject\configTest.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title>测试 config 内置对象</title>
</head>
<body>
<!-- 直接输出 config 的 getServletName 的值 -->
<%=config.getServletName()%>
</body>
</html>
```

上面的代码中粗体字代码输出了 config 的 getServletName()方法的返回值，所有的 JSP 页面都有相同的名字：jsp，所以粗体字代码输出为 jsp。

实际上，我们也可以在 web.xml 文件中配置 JSP（只是比较少用），这样就可以为 JSP 页面指定配置信息，并可为 JSP 页面另外设置一个 URL。

config 对象是 ServletConfig 的实例，该接口用于获取配置参数的方法是 getInitParameter(String paramName)。下面的代码示范了如何在页面中使用 config 获取 JSP 配置参数。

程序清单：codes\02\2.9\jspObject\configTest2.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title>测试 config 内置对象</title>
</head>
<body>
<!-- 输出该 JSP 名为 name 的配置参数 -->
name 配置参数的值:<%=config.getInitParameter("name")%><br/>
<!-- 输出该 JSP 名为 age 的配置参数 -->
age 配置参数的值:<%=config.getInitParameter("age")%>
</body>
</html>
```

上面的代码中两行粗体字代码输出了 config 的 getInitParameter()方法返回值，它们分别获取 name、age 两个配置参数的值。

配置 JSP 也是在 web.xml 文件中进行的，JSP 被当成 Servlet 配置，为 Servlet 配置参数使用 init-param 元素，该元素可以接受 param-name 和 param-value 两个子元素，分别指定参数名和参数值。

在 web.xml 文件中增加如下配置片段，即可将 JSP 页面配置在 Web 应用中。

```
<servlet>
      <!-- 指定 Servlet 名字 -->
      <servlet-name>config</servlet-name>
      <!-- 指定将哪个 JSP 页面配置成 Servlet -->
      <jsp-file>/configTest2.jsp</jsp-file>
      <!-- 配置名为 name 的参数，值为 yeeku -->
      <init-param>
            <param-name>name</param-name>
            <param-value>yeeku</param-value>
```

```
        </init-param>
        <!-- 配置名为 age 的参数，值为 30 -->
        <init-param>
            <param-name>age</param-name>
            <param-value>30</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <!-- 指定将 config Servlet 配置到/config 路径 -->
        <servlet-name>config</servlet-name>
        <url-pattern>/config</url-pattern>
    </servlet-mapping>
```

上面的配置文件片段中粗体字代码为该 Servlet（其实是 JSP）配置了 2 个参数：name 和 age。上面的配置片段把 configTest2.jsp 页面配置成名为 config 的 Servlet，并将该 Servlet 映射到/config 处，这就允许我们通过/config 来访问该页面。在浏览器中访问/config 看到如图 2.19 所示的界面。

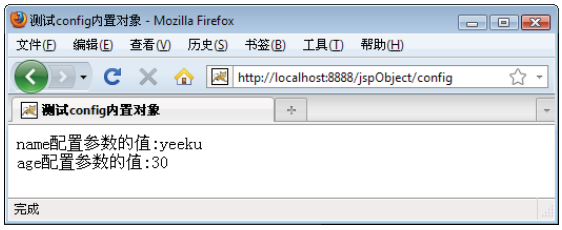

图 2.19　输出 JSP 配置参数值

从图 2.19 中可以看出，通过 config 可以访问到 web.xml 文件中的配置参数。实际上，我们也可以直接访问 configTest2.jsp 页面，在浏览器中访问该页面将看到如图 2.20 所示的界面。

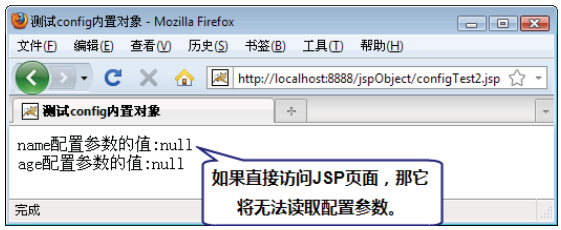

图 2.20　直接访问 JSP 页面将不能访问配置参数

对比图 2.19 和 2.20 不难看出，如果希望 JSP 页面可以获取 web.xml 配置文件中的配置信息，则必须通过为该 JSP 配置的路径来访问该页面，因为只有这样访问 JSP 页面才会让配置参数起作用。

2.9.3　exception 对象

exception 对象是 Throwable 的实例，代表 JSP 脚本中产生的错误和异常，是 JSP 页面异常机制的一部分。

在 JSP 脚本中无须处理异常，即使该异常是 checked 异常。事实上，JSP 脚本包含的所有可能出现的异常都可交给错误处理页面处理。

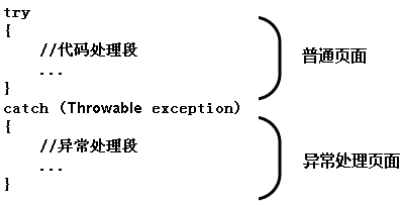

图 2.21　异常处理结构

看如图 2.21 所示的异常处理结构，这是典型的异常捕捉处理块。在 JSP 页面中，普通的 JSP 脚本只执行第一个部分——代码处理段，而异常处理页面负责第二个部分——异常处理段。在异常处理段中，可以看到有个异常对象，该对象就是内置对象 exception。

exception 对象仅在异常处理页面中才有效，通过前面的异常处理结构，读者可以非常清晰地看出这点。

打开普通 JSP 页面所生成的 Servlet 类，将可以发现如下代码片段：

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
      throws java.io.IOException, ServletException {
      ...
      try {
            //所有 JSP 脚本、静态 HTML 部分都会转换成此部分代码
            response.setContentType("text/html; charset=gb2312");
            ...
            out.write("</body>\r\n");
            out.write("</html>\r\n");
      } catch (Throwable t) {
            ...
            //处理该异常
            if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
      } finally {
            //释放资源
            _jspxFactory.releasePageContext(_jspx_page_context);
      }
}
```

从上面代码的粗体字代码中可以看出，JSP 脚本和静态 HTML 部分都将转换成_jspService()方法里的执行性代码——这就是 JSP 脚本无须处理异常的原因：因为这些脚本已经处于 try 块中。一旦 try 块捕捉到 JSP 脚本的异常，并且_jspx_page_context 不为 null，就会由该对象来处理该异常，如上面粗体字代码所示。

_jspx_page_context 对异常的处理也非常简单：如果该页面的 page 指令指定了 errorPage 属性，则将请求 forward 到 errorPage 属性指定的页面，否则使用系统页面来输出异常信息。

由于只有 JSP 脚本、输出表达式才会对应于_jspService()方法里的代码，所以这两个部分的代码无须处理 checked 异常。但 JSP 的声明部分依然需要处理 checked 异常，JSP 的异常处理机制对 JSP 声明不起作用。

在 JSP 的异常处理机制中，一个异常处理页面可以处理多个 JSP 页面脚本部分的异常。异常处理页面通过 page 指令的 errorPage 属性确定。

下面的页面再次测试了 JSP 脚本的异常机制。

程序清单：codes\02\2.9\jspObject\throwEx.jsp

```
<!-- 通过 errorPage 属性指定异常处理页面 -->
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="error.jsp" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> JSP 脚本的异常机制 </title>
</head>
<body>
<%
int a = 6;
int c = a / 0;
%>
</body>
</html>
```

以上页面的粗体字代码将抛出一个 ArithmeticEception，则 JSP 异常机制将会转发到 error.jsp 页面，

error.jsp 页面代码如下。

程序清单：codes\02\2.9\jspObject\error.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" isErrorPage="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title> 异常处理页面 </title>
</head>
<body>
异常类型是:<%=exception.getClass()%><br/>
异常信息是:<%=exception.getMessage()%><br/>
</body>
</html>
```

以上页面 page 指令的 isErrorPage 属性被设为 true，则可以通过 exception 对象来访问上一个页面所出现的异常。在浏览器中请求 throwEx.jsp 页面，将看到如图 2.22 所示的界面。

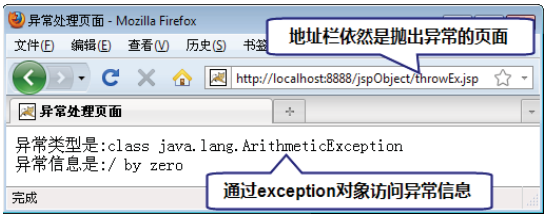

图 2.22　使用 exception 对象

打开 error.jsp 页面生成的 Servlet 类，在_jspService()方法中发现如下代码片段：

```
public void _jspService(HttpServletRequest request, HttpServletResponse response)
     throws java.io.IOException, ServletException {
     PageContext pageContext = null;
     HttpSession session = null;
     //初始化 exception 对象
     Throwable exception = org.apache.jasper.runtime.
          JspRuntimeLibrary.getThrowable(request);
     if (exception != null) {
          response.setStatus(HttpServletResponse.SC_INTERNAL_SERVER_ERROR);
     }
     ...
}
```

从以上代码片段的粗体字代码中可以看出，当 JSP 页面 page 指令的 isErrorPage 为 true 时，该页面就会提供 exception 内置对象。

[redacted]

应将异常处理页面中 page 指令的 isErrorPage 属性设置为 true。只有当 isErrorPage 属性设置为 true 时才可访问 exception 内置对象。

2.9.4　out 对象

out 对象代表一个页面输出流，通常用于在页面上输出变量值及常量。一般在使用输出表达式的地方，都可以使用 out 对象来达到同样效果。

看下面的 JSP 页面使用 out 来执行输出。

程序清单：codes\02\2.9\jspObject\outTest.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.sql.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
```

```
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
        <title> out 测试 </title>
</head>
<body>
<%
//注册数据库驱动
Class.forName("com.mysql.jdbc.Driver");
//获取数据库连接
Connection conn = DriverManager.getConnection(
        "jdbc:mysql://localhost:3306/javaee","root","32147");
//创建 Statement 对象
Statement stmt = conn.createStatement();
//执行查询，获取 ResultSet 对象
ResultSet rs = stmt.executeQuery("select * from news_inf");
%>
<table bgcolor="#9999dd" border="1" width="400">
<%
//遍历结果集
while(rs.next())
{
        //输出表格行
        out.println("<tr>");
        //输出表格列
        out.println("<td>");
        //输出结果集的第二列的值
        out.println(rs.getString(1));
        //关闭表格列
        out.println("</td>");
        //开始表格列
        out.println("<td>");
        //输出结果集的第三列的值
        out.println(rs.getString(2));
        //关闭表格列
        out.println("</td>");
        //关闭表格行
        out.println("</tr>");
}
%>
<table>
</body>
</html>
```

从 Java 的语法上看，上面的程序更容易理解，out 是个页面输出流，负责输出页面表格及所有内容，但使用 out 则需要编写更多代码。

所有使用 out 的地方，都可使用输出表达式来代替，而且使用输出表达式更加简洁。<%= ...%>表达式的本质就是 out.write(...);。通过 out 对象的介绍，读者可以更好地理解输出表达式的原理。

➤➤2.9.5 pageContext 对象

这个对象代表页面上下文，该对象主要用于访问 JSP 之间的共享数据。使用 pageContext 可以访问 page、request、session、application 范围的变量。

pageContext 是 PageContext 类的实例，它提供了如下两个方法来访问 page、request、session、application 范围的变量。

- getAttribute(String name)：取得 page 范围内的 name 属性。

- getAttribute(String name,int scope)：取得指定范围内的 name 属性，其中 scope 可以是如下 4 个值。
- PageContext.PAGE_SCOPE：对应于 page 范围。
- PageContext.REQUEST_SCOPE：对应于 request 范围。
- PageContext.SESSION_SCOPE：对应于 session 范围。
- PageContext.APPLICATION_SCOPE：对应于 application 范围。

与 getAttribute()方法相对应，PageContext 也提供了 2 个对应的 setAttribute()方法，用于将指定变量放入 page、request、session、application 范围内。

下面的 JSP 页面示范了使用 pageContext 来操作 page、request、session、application 范围内的变量。

程序清单：codes\02\2.9\jspObject\pageContextTest.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> pageContext 测试 </title>
</head>
<body>
<%
//使用 pageContext 设置属性，该属性默认在 page 范围内
pageContext.setAttribute("page","hello");
//使用 request 设置属性，该属性默认在 request 范围内
request.setAttribute("request","hello");
//使用 pageContext 将属性设置在 request 范围中
pageContext.setAttribute("request2","hello"
      , pageContext.REQUEST_SCOPE);
//使用 session 将属性设置在 session 范围中
session.setAttribute("session","hello");
//使用 pageContext 将属性设置在 session 范围中
pageContext.setAttribute("session2","hello"
      , pageContext.SESSION_SCOPE);
//使用 application 将属性设置在 application 范围中
application.setAttribute("app","hello");
//使用 pageContext 将属性设置在 application 范围中
pageContext.setAttribute("app2","hello"
      , pageContext.APPLICATION_SCOPE);
//下面获取各属性所在的范围：
out.println("page 变量所在范围：" +
      pageContext.getAttributesScope("page") + "<br/>");
out.println("request 变量所在范围：" +
      pageContext.getAttributesScope("request") + "<br/>");
out.println("request2 变量所在范围："+
      pageContext.getAttributesScope("request2") + "<br/>");
out.println("session 变量所在范围：" +
      pageContext.getAttributesScope("session") + "<br/>");
out.println("session2 变量所在范围：" +
      pageContext.getAttributesScope("session2") + "<br/>");
out.println("app 变量所在范围：" +
      pageContext.getAttributesScope("app") + "<br/>");
out.println("app2 变量所在范围：" +
      pageContext.getAttributesScope("app2") + "<br/>");
%>
</body>
</html>
```

以上页面的粗体字代码使用 pageContext 将各变量分别放入 page、request、session、application 范围内，程序的斜体字代码还使用 pageContext 获取各变量所在的范围。

浏览以上页面，可以看到如图 2.23 所示的效果。

图 2.23　使用 pageContext 操作各范围属性的效果

图 2.23 中显示了使用 pageContext 获取各属性所在的范围，其中这些范围获取的都是整型变量，这些整型变量分别对应如下 4 个生存范围。

1：对应 page 生存范围。

2：对应 request 生存范围。

3：对应 session 生存范围。

4：对应 application 生存范围。

不仅如此，pageContext 还可用于获取其他内置对象，pageContext 对象包含如下方法。

- ServletRequest getRequest()：获取 request 对象。
- ServletResponse getResponse()：获取 response 对象。
- ServletConfig getServletConfig()：获取 config 对象。
- ServletContext getServletContext()：获取 application 对象。
- HttpSession getSession()：获取 session 对象。

因此一旦在 JSP、Servlet 编程中获取了 pageContext 对象，就可以通过它提供的上面方法来获取其他内置对象。

2.9.6　request 对象

request 对象是 JSP 中重要的对象，每个 request 对象封装着一次用户请求，并且所有的请求参数都被封装在 request 对象中，因此 request 对象是获取请求参数的重要途径。

除此之外，request 可代表本次请求范围，所以还可用于操作 request 范围的属性。

1．获取请求头/请求参数

Web 应用是请求/响应架构的应用，浏览器发送请求时通常总会附带一些请求头，还可能包含一些请求参数发送给服务器，服务器端负责解析请求头/请求参数的就是 JSP 或 Servlet，而 JSP 和 Servlet 取得请求参数的途径就是 request。request 是 HttpServletRequest 接口的实例，它提供了如下几个方法来获取请求参数。

- String getParameter(String paramName)：获取 paramName 请求参数的值。
- Map getParameterMap()：获取所有请求参数名和参数值所组成的 Map 对象。
- Enumeration getParameterNames()：获取所有请求参数名所组成的 Enumeration 对象。
- String[] getParameterValues(String name)：paramName 请求参数的值，当该请求参数有多个值时，该方法将返回多个值所组成的数组。

HttpServletRequest 提供了如下方法来访问请求头。

- String getHeader(String name)：根据指定请求头的值。
- java.util.Enumeration<String> getHeaderNames()：获取所有请求头的名称。
- java.util.Enumeration<String> getHeaders(String name)：获取指定请求头的多个值。

- int getIntHeader(String name)：获取指定请求头的值，并将该值转为整数值。

对于开发人员来说，请求头和请求参数都是由用户发送到服务器的数据，区别在于请求头通常由浏览器自动添加，因此一次请求总是包含若干请求头；而请求参数则通常需要开发人员控制添加，让客户端发送请求参数通常分两种情况。

- GET 方式的请求：直接在浏览器地址栏输入访问地址所发送的请求或提交表单发送请求时，该表单对应的 form 元素没有设置 method 属性，或设置 method 属性为 get，这几种请求都是 GET 方式的请求。GET 方式的请求会将请求参数的名和值转换成字符串，并附加在原 URL 之后，因此可以在地址栏中看到请求参数名和值。且 GET 请求传送的数据量较小，一般不能大于 2KB。
- POST 方式的请求：这种方式通常使用提交表单（由 form HTML 元素表示）的方式来发送，且需要设置 form 元素的 method 属性为 post。POST 方式传送的数据量较大，通常认为 POST 请求参数的大小不受限制，但往往取决于服务器的限制，POST 请求传输的数据量总比 GET 传输的数据量大。而且 POST 方式发送的请求参数以及对应的值放在 HTML HEADER 中传输，用户不能在地址栏里看到请求参数值，安全性相对较高。

对比上面两种请求方式，由此可见我们通常应该采用 POST 方式发送请求。

几乎每个网站都会大量使用表单，表单用于收集用户信息，一旦用户提交请求，表单的信息将会提交给对应的处理程序，如果为 form 元素设置 method 属性为 post，则表示发送 POST 请求。

下面是表单页面的代码。

程序清单：codes\02\2.9\jspObject\form.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCtype html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 收集参数的表单页 </title>
</head>
<body>
<form id="form1" method="post" action="request1.jsp">
用户名：<br/>
<input type="text" name="name"><hr/>
性别：<br/>
男：<input type="radio" name="gender" value="男">
女：<input type="radio" name="gender" value="女"><hr/>
喜欢的颜色：<br/>
红：<input type="checkbox" name="color" value="红">
绿：<input type="checkbox" name="color" value="绿">
蓝：<input type="checkbox" name="color" value="蓝"><hr/>
来自的国家：<br/>
<select name="country">
      <option value="中国">中国</option>
      <option value="美国">美国</option>
      <option value="俄罗斯">俄罗斯</option>
</select><hr/>
<input type="submit" value="提交">
<input type="reset" value="重置">
</form>
</body>
</html>
```

这个页面没有动态的 JSP 部分，它只是包含一个收集请求参数的表单，且粗体字部分设置了该表单的 action 为 request1.jsp，这表明提交该表单时，请求将发送到 request1.jsp 页面；粗体字代码还设置了 method 为 post，这表明提交表单将发送 POST 请求。

除此之外，表单里还包含 1 个文本框、2 个单选框、3 个复选框及 1 个下拉列表框，另外包括“提交”和“重置”2 个按钮。页面的执行效果如图 2.24 所示。

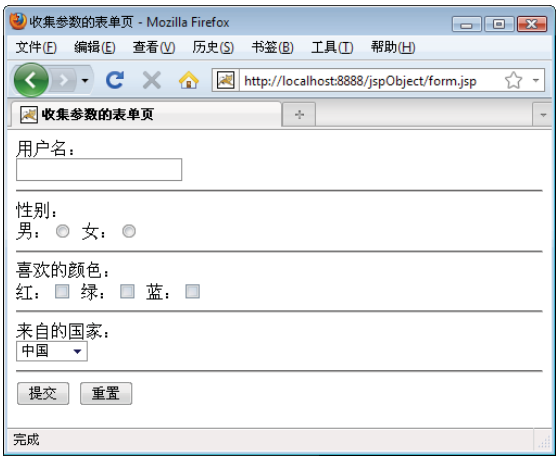

图 2.24　表单页

在该页面中输入相应信息后，单击“提交”按钮，表单域所代表的请求参数将通过 request 对象的 getParameter()方法来取得。

提示：

并不是每个表单域都会生成请求参数的，而是有 name 属性的表单域才生成请求参数。

关于表单域和请求参数的关系遵循如下 4 点：

- **每个有 name 属性的表单域对应一个请求参数。**
- **如果有多个表单域有相同的 name 属性，则多个表单域只生成一个请求参数，只是该参数有多个值。**
- **表单域的 name 属性指定请求参数名，value 指定请求参数值。**
- **如果某个表单域设置了 disabled="disabled"属性，则该表单域不再生成请求参数。**

上面的表单页向 request1.jsp 页面发送请求，request1.jsp 页面的代码如下。

程序清单：codes\02\2.9\jspObject\request1.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.util.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 获取请求头/请求参数 </title>
</head>
<body>
<%
//获取所有请求头的名称
Enumeration<String> headerNames = request.getHeaderNames();
while(headerNames.hasMoreElements())
{
      String headerName = headerNames.nextElement();
      //获取每个请求、及其对应的值
      out.println(headerName + "-->" + request.getHeader(headerName) + "<br/>");
}
out.println("<hr/>");
//设置解码方式，对于简体中文，使用 gb2312 解码
request.setCharacterEncoding("gb2312");
//下面依次获取表单域的值
String name = request.getParameter("name");
String gender = request.getParameter("gender");
//如果某个请求参数有多个值，将使用该方法获取多个值
String[] color = request.getParameterValues("color");
```

```
String national = request.getParameter("country");
%>
<!-- 下面依次输出表单域的值 -->
您的名字：<%=name%><hr/>
您的性别：<%=gender%><hr/>
<!-- 输出复选框获取的数组值 -->
您喜欢的颜色：<%for(String c : color)
{out.println(c + " ");}%><hr/>
您来自的国家：<%=national%><hr/>
</body>
</html>
```

上述页面代码中粗体字代码示范了如何获取请求头、请求参数，在获取表单域对应的请求参数值之前，先设置 request 编码的字符集（如粗斜体代码所示）——如果 POST 请求的请求参数里包含非西欧字符，则必须在获取请求参数之前先调用 setCharacterEncoding()方法设置编码的字符集。

如果发送请求的表单页采用 gb2312 字符集，该表单页发送的请求也将采用 gb2312 字符集，所以本页面需要先执行如下方法。

setCharacterEncoding("gb2312")：设置 request 编码所用的字符集。

在表单提交页的各个输入域内输入对应的值，然后单击“提交”按钮，request1.jsp 就会出现如图 2.25 所示的效果。

图 2.25　获取 POST 方式的请求参数

如果需要传递的参数是普通字符串，而且仅需传递少量参数，可以选择使用 GET 方式发送请求参数，GET 方式发送的请求参数被附加到地址栏的 URL 之后，地址栏的 URL 将变成如下形式：

url?param1=value1&param2=value2&…paramN=valueN：URL 和参数之间以“?”分隔，而多个参数之间以“&”分隔。

下面的 JSP 页面示范了如何通过 request 来获取 GET 请求参数值。

程序清单：codes\02\2.9\jspObject\request2.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title> 获取 GET 请求参数 </title>
</head>
<body>
<%
```

```
//获取 name 请求参数的值
String name = request.getParameter("name");
//获取 gender 请求参数的值
String gender = request.getParameter("gender");
%>
<!-- 输出 name 变量值 -->
您的名字：<%=name%><hr/>
<!-- 输出 gender 变量值 -->
您的性别：<%=gender%><hr/>
</body>
</html>
```

上面的页面中粗体字代码用于获取 GET 方式的请求参数，从这些代码不难看出：request 获取 POST 请求参数的代码和获取 GET 请求参数代码完全一样。向该页面发送请求时直接在地址栏里增加一些 GET 方式的请求参数，执行效果如图 2.26 所示。

细心的读者可能发现上面两个请求参数值都由英文字符组成，如果请求参数值里包含非西欧字符，那么是不是应该先调用 setCharacterEncoding()来设置 request 编码的字符集呢？读者可以试一下。答案是不行，如果 GET 方式的请求值里包含了非西欧字符，则获取这些参数比较复杂。

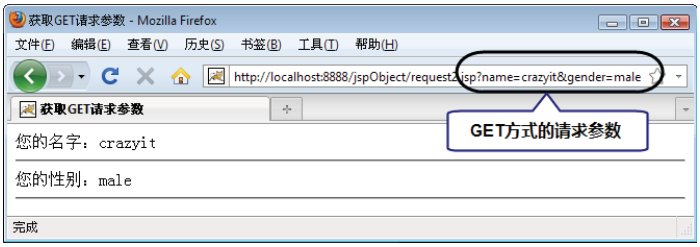

图 2.26　获取 GET 方式的请求参数

下面的页面示范了如何获取 GET 请求里的中文字符。

程序清单：codes\02\2.9\jspObject\request3.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 获取包含非西欧字符的 GET 请求参数 </title>
</head>
<body>
<%
//获取请求里包含的查询字符串
String rawQueryStr = request.getQueryString();
out.println("原始查询字符串为：" + rawQueryStr + "<hr/>");
//使用 URLDecoder 解码字符串
String queryStr = java.net.URLDecoder.decode(
      rawQueryStr , "gbk");
out.println("解码后的查询字符串为：" + queryStr + "<hr/>");
//以&符号分解查询字符串
String[] paramPairs = queryStr.split("&");
for(String paramPair : paramPairs)
{
      out.println("每个请求参数名、值对为：" + paramPair + "<br/>");
      //以=来分解请求参数名和值
      String[] nameValue = paramPair.split("=");
      out.println(nameValue[0] + "参数的值是：" +
            nameValue[1]+ "<hr/>");
}
%>
</body>
</html>
```

上面的程序中粗体字代码就是获取 GET 请求里中文参数值的关键代码，为了获取 GET 请求里的

中文参数值，必须借助于 java.net.URLDecoder 类。关于 URLDecoder 和 URLEncoder 两个类的用法请参考疯狂 Java 体系的《疯狂 Java 讲义》一书的 17.2 节。

读者可以编写一个表单，并让表单以 GET 方式提交请求到 request3.jsp 页面，将可看到如图 2.27 所示的效果。

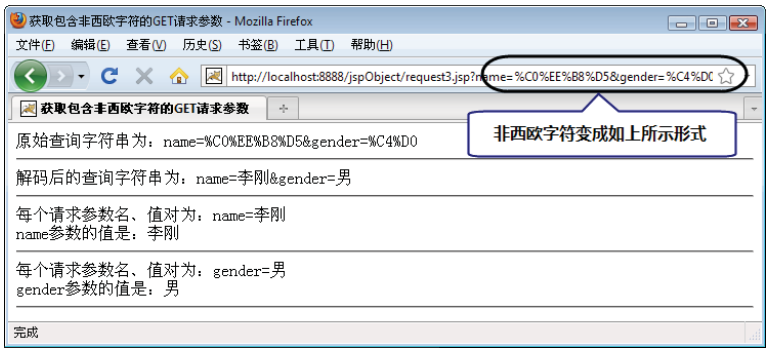

图 2.27　获取 GET 请求的中文请求参数

如果读者不想这样做，还可以在获取请求参数值之后对请求参数值重新编码。也就是先将其转换成字节数组，再将字节数组重新解码成字符串。例如，可通过如下代码来取得 name 请求参数的参数值。

```
//获取原始的请求参数值
String rawName = request.getParameter("name");
//将请求参数值使用 ISO-8859-1 字符串分解成字节数组
byte[] rawBytes = rawName.getBytes("ISO-8859-1");
//将字节数组重新解码成字符串
String name = new String(rawBytes , "gb2312");
通过上面代码片段也可处理 GET 请求里的中文请求参数值。
```

2．操作 request 范围的属性

HttpServletRequest 还包含如下两个方法，用于设置和获取 request 范围的属性。

- setAttribute(String attName , Object attValue)：将 attValue 设置成 request 范围的属性。
- Object getAttribute(String attName)：获取 request 范围的属性。

当 forward 用户请求时，请求的参数和请求属性都不会丢失。看下一个 JSP 页面，这个 JSP 页面是个简单的表单页，用于提交用户请求。

程序清单：codes\02\2.9\jspObject\draw.jsp

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 取钱的表单页 </title>
</head>
<body>
<!-- 取钱的表单 -->
<form method="post" action="first.jsp">
      取钱：<input type="text" name="balance">
      <input type="submit" value="提交">
</form>
</body>
</html>
```

该页面向 first.jsp 页面请求后，balance 参数将被提交到 first.jsp 页面，下面是 first.jsp 页面的实现代码。

程序清单：codes\02\2.9\jspObject\first.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.util.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> request 处理 </title>
</head>
<body>
<%
//获取请求的钱数
String bal = request.getParameter("balance");
//将钱数的字符串转换成双精度浮点数
double qian = Double.parseDouble(bal);
//对取出的钱进行判断
if (qian < 500)
{
      out.println("给你" + qian + "块");
      out.println("账户减少" + qian);
}
else
{
      //创建了一个 List 对象
      List<String> info = new ArrayList<String>();
      info.add("1111111");
      info.add("2222222");
      info.add("3333333");
      //将 info 对象放入 request 范围内
      request.setAttribute("info" , info);
%>
<!-- 实现转发 -->
<jsp:forward page="second.jsp"/>
<%}%>
</body>
</html>
```

first.jsp 页面首先获取请求的取钱数，然后对请求的钱数进行判断。如果请求的钱数小于 500，则允许直接取钱；否则将请求转发到 second.jsp。转发之前，创建了一个 List 对象，并将该对象设置成 request 范围的 info 属性。

接下来在 second.jsp 页面中，不仅获取了请求的 balance 参数，而且还会获取 request 范围的 info 属性。second.jsp 页面的代码如下。

程序清单：codes\02\2.9\jspObject\second.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.util.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> request 处理 </title>
</head>
<body>
<%
//取出请求参数
String bal = request.getParameter("balance");
double qian = Double.parseDouble(bal);
//取出 request 范围内的 info 属性
List<String> info  = (List<String>)request.getAttribute("info");
for (String tmp : info)
{
      out.println(tmp + "<br/>");
}
out.println("取钱" + qian + "块");
out.println("账户减少" + qian);
%>
```

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

如果页面请求的钱数大于 500，请求将被转发到 second.jsp 页面处理，而且在 second.jsp 页面中可以获取到 balance 请求参数值，也可获取到 request 范围的 info 属性，这表明：forward 用户请求时，请求参数和 request 范围的属性都不会丢失，即 forward 动作还是原来的请求，并未再次向服务器发送请求。

如果请求取钱的钱数为 654，则页面的执行效果如图 2.28 所示。

图 2.28　操作 request 范围的属性

3．执行 forward 或 include

request 还有一个功能就是执行 forward 和 include，也就是代替 JSP 所提供的 forward 和 include 动作指令。前面我们需要 forward 时都是通过 JSP 提供的动作指令进行的，实际上 request 对象也可以执行 forward。

HttpServletRequest 类提供了一个 getRequestDispatcher (String path)方法，其中 path 就是希望 forward 或者 include 的目标路径，该方法返回 RequestDispatcher，该对象提供了如下两个方法。

- forward(ServletRequest request, ServletResponse response)：执行 forward。
- include(ServletRequest request, ServletResponse response)：执行 include。

如下代码行可以将 a.jsp 页面 include 到本页面中：

```
getRequestDispatcher("/a.jsp").include(request , response);
```

如下代码行则可以将请求 forward 到 a.jsp 页面：

```
getRequestDispatcher("/a.jsp").forward(request , response);
```

[redacted]

使用 request 的 getRequestDispatcher(String path)方法时，该 path 字符串必须以斜线开头。

➤➤2.9.7　response 对象

response 代表服务器对客户端的响应。大部分时候，程序无须使用 response 来响应客户端请求，因为有个更简单的响应对象——out，它代表页面输出流，直接使用 out 生成响应更简单。

但 out 是 JspWriter 的实例，JspWriter 是 Writer 的子类，Writer 是字符流，无法输出非字符内容。假如需要在 JSP 页面中动态生成一幅位图、或者输出一个 PDF 文档，使用 out 作为响应对象将无法完成，此时必须使用 response 作为响应输出。

除此之外，还可以使用 response 来重定向请求，以及用于向客户端增加 Cookie。

1．response 响应生成非字符响应

对于需要生成非字符响应的情况，就应该使用 response 来响应客户端请求。下面的 JSP 页面将在客户端生成一张图片。response 是 HttpServletResponse 接口的实例，该接口提供了一个 getOutputStream()

方法，该方法返回响应输出字节流。

程序清单：codes\02\2.9\jspObject\img.jsp

```
<%-- 通过 contentType 属性指定响应数据是图片 --%>
<%@ page contentType="image/jpeg" language="java"%>
<%@ page import="java.awt.image.*,javax.imageio.*,java.io.*,java.awt.*"%>
<%
//创建 BufferedImage 对象
BufferedImage image = new BufferedImage(340 ,
     160, BufferedImage.TYPE_INT_RGB);
//以 Image 对象获取 Graphics 对象
Graphics g = image.getGraphics();
//使用 Graphics 画图，所画的图像将会出现在 image 对象中
g.fillRect(0,0,400,400);
//设置颜色：红
g.setColor(new Color(255,0,0));
//画出一段弧
g.fillArc(20, 20, 100,100, 30, 120);
//设置颜色：绿
g.setColor(new Color(0 , 255, 0));
//画出一段弧
g.fillArc(20, 20, 100,100, 150, 120);
//设置颜色：蓝
g.setColor(new Color(0 , 0, 255));
//画出一段弧
g.fillArc(20, 20, 100,100, 270, 120);
//设置颜色：黑
g.setColor(new Color(0,0,0));
g.setFont(new Font("Arial Black", Font.PLAIN, 16));
//画出三个字符串
g.drawString("red:climb" , 200 , 60);
g.drawString("green:swim" , 200 , 100);
g.drawString("blue:jump" , 200 , 140);
g.dispose();
//将图像输出到页面的响应
ImageIO.write(image , "jpg" , response.getOutputStream());
%>
```

以上页面的粗体字代码先设置了服务器响应数据是 image/jpeg，这表明服务器响应是一张 JPG 图片。接着创建了一个 BufferedImage 对象（代表图像），并获取该 BufferedImage 的 Graphics 对象（代表画笔），然后通过 Graphics 向 BufferedImage 中绘制图形，最后一行代码将直接将 BufferedImage 作为响应发送给客户端。

请直接在浏览器中请求该页面，将看到浏览器显示一张图片，效果如图 2.29 所示。

图 2.29　使用 response 生成非字符响应

也可以在其他页面中使用 img 标签来显示这个图片页面，代码如下：

```
<img src="img.jsp">
```

使用这种临时生成图片的方式就可以非常容易地实现网页上的图形验证码功能。不仅如此，使用

response 生成非字符响应还可以直接生成 PDF 文件、Excel 文件，这些文件可直接作为报表使用。

2．重定向

重定向是 response 的另外一个用处，与 forward 不同的是，重定向会丢失所有的请求参数和 request 范围的属性，因为重定向将生成第二次请求，与前一次请求不在同一个 request 范围内，所以发送一次请求的请求参数和 request 范围的属性全部丢失。

HttpServletResponse 提供了一个 sendRedirect(String path)方法，该方法用于重定向到 path 资源，即重新向 path 资源发送请求。

下面的 JSP 页面将使用 response 执行重定向。

程序清单：codes\02\2.9\jspObject\doRedirect.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%
//生成页面响应
out.println("====");
//重定向到 forward-result.jsp 页面
response.sendRedirect("redirect-result.jsp");
%>
```

以上页面的粗体字代码用于执行重定向，向该页面发送请求时，请求会被重定向到 redirect-result.jsp 页面。例如，在地址栏中输入 http://localhost:8888/jspObject/ doRedirect.jsp?name=yeeku，然后按回车键，将看到如图 2.30 所示的效果。

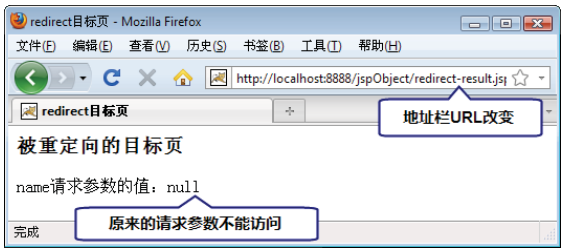

图 2.30　redirect 效果

注意地址栏的改变，执行重定向动作时，地址栏的 URL 会变成重定向的目标 URL。

重定向会丢失所有的请求参数，使用重定向的效果，与在地址栏里重新输入新地址再按回车键的效果完全一样，即发送了第二次请求。

从表面上来看，forward 动作和 redirect 动作有些相似：它们都可将请求传递到另一个页面。但实际上 forward 和 redirect 之间存在较大的差异，forward 和 redirect 的差异如表 2.1 所示。

表 2.1　forward 和 redirect 对比

转发（forward）	重定向（redirect）
执行 forward 后依然是上一次请求	执行 redirect 后生成第二次请求
forward 的目标页面可以访问原请求的请求参数，因为依然是同一次请求，所有原请求的请求参数、request 范围的属性全部存在	redirect 的目标页面不能访问原请求的请求参数，因为是第二次请求了，所有原请求的请求参数、request 范围的属性全部丢失
地址栏里请求的 URL 不会改变	地址栏改为重定向的目标 URL。相当于在浏览器地址栏里输入新的 URL 后按回车键

3．增加 Cookie

Cookie 通常用于网站记录客户的某些信息，比如客户的用户名及客户的喜好等。一旦用户下次登录，网站可以获取到客户的相关信息，根据这些客户信息，网站可以对客户提供更友好的服务。Cookie

与 session 的不同之处在于：session 会随浏览器的关闭而失效，但 Cookie 会一直存放在客户端机器上，除非超出 Cookie 的生命期限。

由于安全性的原因，使用 Cookie 客户端浏览器必须支持 Cookie 才行。客户端浏览器完全可以设置禁用 Cookie。

增加 Cookie 也是使用 response 内置对象完成的，response 对象提供了如下方法。

- void addCookie(Cookie cookie)：增加 Cookie。

正如在上面的方法中见到的，在增加 Cookie 之前，必须先创建 Cookie 对象。增加 Cookie 请按如下步骤进行。

1. 创建 Cookie 实例，Cookie 的构造器为 Cookie(String name, String value)。
2. 设置 Cookie 的生命期限，即该 Cookie 在多长时间内有效。
3. 向客户端写 Cookie。

看如下 JSP 页面，该页面可以用于向客户端写一个 username 的 Cookie。

程序清单：codes\02\2.9\jspObject\addCookie.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 增加 Cookie </title>
</head>
<body>
<%
//获取请求参数
String name = request.getParameter("name");
//以获取到的请求参数为值，创建一个 Cookie 对象
Cookie c = new Cookie("username" , name);
//设置 Cookie 对象的生存期限
c.setMaxAge(24 * 3600);
//向客户端增加 Cookie 对象
response.addCookie(c);
%>
</body>
</html>
```

如果浏览器没有阻止 Cookie，在地址栏输入 http://localhost:8888/jspObject/addCookie.jsp?name=crazyit，执行该页面后，网站就会向客户端机器写入一个名为 username 的 Cookie，该 Cookie 将在客户端硬盘上一直存在，直到超出该 Cookie 的生存期限（本 Cookie 设置为 24 小时 。

访问客户端 Cookie 使用 request 对象，request 对象提供了 getCookies()方法，该方法将返回客户端机器上所有 Cookie 组成的数组，遍历该数组的每个元素，找到希望访问的 Cookie 即可。

下面是访问 Cookie 的 JSP 页面的代码。

程序清单：codes\02\2.9\jspObject\readCookie.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 读取 Cookie </title>
</head>
<body>
<%
//获取本站在客户端上保留的所有 Cookie
Cookie[] cookies = request.getCookies();
//遍历客户端上的每个 Cookie
for (Cookie c : cookies)
{
```

```
        //如果 Cookie 的名为 username，表明该 Cookie 是我们需要访问的 Cookie
        if(c.getName().equals("username"))
        {
              out.println(c.getValue());
        }
  }
  %>
  </body>
  </html>
```

上面的粗体字代码就是通过 request 读取 Cookie 数组，并搜寻指定 Cookie 的关键代码，访问该页面即可读出刚才写在客户端的 Cookie。

使用 Cookie 对象必须设置其生存期限，否则 Cookie 将会随浏览器的关闭而自动消失。

默认情况下，Cookie 值不允许出现中文字符，如果我们需要值为中文内容的 Cookie 怎么办呢？同样可以借助于 java.net.URLEncoder 先对中文字符串进行编码，将编码后的结果设为 Cookie 值。当程序要读取 Cookie 时，则应该先读取，然后使用 java.net.URLDecoder 对其进行解码。

如下代码片段示范了如何存入值为中文的 Cookie。

程序清单：codes\02\2.9\jspObject\cnCookie.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
...
<%
//以编码后的字符串为值，创建一个 Cookie 对象
Cookie c = new Cookie("cnName"
      , java.net.URLEncoder.encode("孙悟空" , "gbk"));
//设置 Cookie 对象的生存期限
c.setMaxAge(24 * 3600);
//向客户端增加 Cookie 对象
response.addCookie(c);

//获取本站在客户端上保留的所有 Cookie
Cookie[] cookies = request.getCookies();
//遍历客户端上的每个 Cookie
for (Cookie cookie : cookies)
{
      //如果 Cookie 的名为 username，表明该 Cookie 是我们需要访问的 Cookie
      if(cookie.getName().equals("cnName"))
      {
            //使用 java.util.URLDecoder 对 Cookie 值进行解码
            out.println(java.net.URLDecoder
                  .decode(cookie.getValue()));
      }
}
%>
```

上面的程序中两行粗体字代码是存入值为中文的 Cookie 的关键：存入之前先用 java.net.URLEncoder 进行编码；读取时需要对读取的 Cookie 值用 java.net.URLDecoder 进行解码。

➤➤2.9.8 session 对象

session 对象也是一个非常常用的对象，这个对象代表一次用户会话。一次用户会话的含义是：从客户端浏览器连接服务器开始，到客户端浏览器与服务器断开为止，这个过程就是一次会话。

session 通常用于跟踪用户的会话信息，如判断用户是否登录系统，或者在购物车应用中，用于跟踪用户购买的商品等。

session 范围内的属性可以在多个页面的跳转之间共享。一旦关闭浏览器，即 session 结束，session

范围内的属性将全部丢失。

session 对象是 HttpSession 的实例，HttpSession 有如下两个常用的方法。

- setAttribute(String attName，Object attValue)：设置 session 范围内 attName 属性的值为 attValue。
- getAttribute(String attName)：返回 session 范围内 attName 属性的值。

下面的示例演示了一个购物车应用，以下是陈列商品的 JSP 页面代码。

程序清单：codes\02\2.9\jspObject\shop.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCtype html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 选择物品购买 </title>
</head>
<body>
<form method="post" action="processBuy.jsp">
      书籍：<input type="checkbox" name="item" value="book"/><br/>
      电脑：<input type="checkbox" name="item" value="computer"/><br/>
      汽车：<input type="checkbox" name="item" value="car"/><br/>
      <input type="submit" value="购买"/>
</form>
</body>
</html>
```

这个页面几乎没有动态的 JSP 部分，全部是静态的 HTML 内容。该页面包含一个表单，表单里包含三个复选按钮，用于选择想购买的物品，表单由 processBuy.jsp 页面处理，其页面的代码如下：

程序清单：codes\02\2.9\jspObject\processBuy.jsp

```
<%@ page contentType="text/html; charset=gb2312" language="java" %>
<%@ page import="java.util.*"%>
<%
//取出 session 范围的 itemMap 属性
Map<String,Integer> itemMap = (Map<String,Integer>)session
      .getAttribute("itemMap");
//如果 Map 对象为空，则初始化 Map 对象
if (itemMap == null)
{
      itemMap = new HashMap<String,Integer>();
      itemMap.put("书籍" , 0);
      itemMap.put("电脑" , 0);
      itemMap.put("汽车" , 0);
}
//获取上一个页面的请求参数
String[] buys = request.getParameterValues("item");
//遍历数组的各元素
for (String item : buys)
{
      //如果 item 为 book，表示选择购买书籍
      if(item.equals("book"))
      {
            int num1 = itemMap.get("书籍").intValue();
            //将书籍 key 对应的数量加 1
            itemMap.put("书籍" , num1 + 1);
      }
      //如果 item 为 computer，表示选择购买电脑
      else if (item.equals("computer"))
      {
            int num2 = itemMap.get("电脑").intValue();
            //将电脑 key 对应的数量加 1
            itemMap.put("电脑" , num2 + 1);
      }
      //如果 item 为 car，表示选择购买汽车
```

```
        else if (item.equals("car"))
        {
              int num3 = itemMap.get("汽车").intValue();
              //将汽车 key 对应的数量加 1
              itemMap.put("汽车" , num3 + 1);
        }
  }
  //将 itemMap 对象放到设置成 session 范围的 itemMap 属性
  session.setAttribute("itemMap" , itemMap);
  %>
  <!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
  <html xmlns="http://www.w3.org/1999/xhtml">
  <head>
        <title> new document </title>
  </head>
  <body>
  您所购买的物品：<br/>
  书籍：<%=itemMap.get("书籍")%>本<br/>
  电脑：<%=itemMap.get("电脑")%>台<br/>
  汽车：<%=itemMap.get("汽车")%>辆
  <p><a href="shop.jsp">再次购买</a></p>
  </body>
  </html>
```

以上页面中粗体字代码使用 session 来保证 itemMap 对象在一次会话中有效，这使得该购物车系统可以反复购买，只要浏览器不关闭，购买的物品信息就不会丢失，图 2.31 显示的是多次购买后的效果。

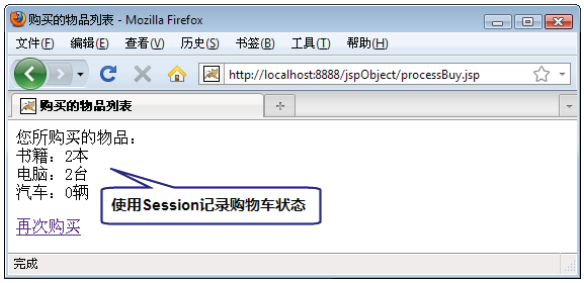

图 2.31　利用 session 记录购物车信息

考虑 session 本身的目的，通常只应该把与用户会话状态相关的信息放入 session 范围内。不要仅仅为了两个页面之间交换信息，就将该信息放入 session 范围内。如果仅仅为了两个页面交换信息，可以将该信息放入 request 范围内，然后 forward 请求即可。

关于 session 还有一点需要指出，session 机制通常用于保存客户端的状态信息，这些状态信息需要保存到 Web 服务器的硬盘上，所以要求 session 里的属性值必须是可序列化的，否则将会引发不可序列化的异常。

session 的属性值可以是任何可序列化的 Java 对象。

2.10　Servlet 介绍

前面已经介绍过，JSP 的本质就是 Servlet，开发者把编写好的 JSP 页面部署在 Web 容器中之后，Web 容器会将 JSP 编译成对应的 Servlet。但直接使用 Servlet 的坏处是：Servlet 的开发效率非常低，

特别是当使用 Servlet 生成表现层页面时，页面中所有的 HTML 标签，都需采用 Servlet 的输出流来输出，因此极其烦琐。而且 Servlet 标准的 Java 类，必须由程序员开发、修改，美工人员难以参与 Servlet 页面的开发。这一系列的问题，都阻碍了 Servlet 作为表现层的使用。

自 MVC 规范出现后，Servlet 的责任开始明确下来，仅仅作为控制器使用，不再需要生成页面标签，也不再作为视图层角色使用。

2.10.1 Servlet 的开发

前面介绍的 JSP 的本质就是 Servlet，Servlet 通常被称为服务器端小程序，是运行在服务器端的程序，用于处理及响应客户端的请求。

Servlet 是个特殊的 Java 类，这个 Java 类必须继承 HttpServlet。每个 Servlet 可以响应客户端的请求。Servlet 提供不同的方法用于响应客户端请求。

- doGet：用于响应客户端的 GET 请求。
- doPost：用于响应客户端的 POST 请求。
- doPut：用于响应客户端的 PUT 请求。
- doDelete：用于响应客户端的 DELETE 请求。

事实上，客户端的请求通常只有 GET 和 POST 两种，Servlet 为了响应这两种请求，必须重写 doGet()和 doPost()两个方法。如果 Servlet 为了响应 4 个方式的请求，则需要同时重写上面的 4 个方法。

大部分时候，Servlet 对于所有请求的响应都是完全一样的。此时，可以采用重写一个方法来代替上面的几个方法：只需重写 service()方法即可响应客户端的所有请求。

另外，HttpServlet 还包含两个方法。

- init(ServletConfig config)：创建 Servlet 实例时，调用该方法的初始化 Servlet 资源。
- destroy()：销毁 Servlet 实例时，自动调用该方法的回收资源。

通常无须重写 init()和 destroy()两个方法，除非需要在初始化 Servlet 时，完成某些资源初始化的方法，才考虑重写 init 方法。如果需要在销毁 Servlet 之前，先完成某些资源的回收，比如关闭数据库连接等，才需要重写 destroy 方法。

不用为 Servlet 类编写构造器，如果需要对 Servlet 执行初始化操作，应将初始化操作放在 Servlet 的 init()方法中定义。如果重写了 init(ServletConfig config)方法，则应在重写该方法的第一行调用 super.init(config)。该方法将调用 HttpServlet 的 init 方法。

下面提供一个 Servlet 的示例，该 Servlet 将获取表单请求参数，并将请求参数显示给客户端。

程序清单：codes\02\2.10\servletDemo\WEB-INF\src\lee\FirstServlet.java

```
//Servlet 必须继承 HttpServlet 类
@WebServlet(name="firstServlet"
      , urlPatterns={"/firstServlet"})
public class FirstServlet extends HttpServlet
{
      //客户端的响应方法，使用该方法可以响应客户端所有类型的请求
      public void service(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException,java.io.IOException
      {
            //设置解码方式
            request.setCharacterEncoding("GBK");
            response.setContentType("text/html;charSet=GBK");
            //获取 name 的请求参数值
```

```
            String name = request.getParameter("name");
            //获取 gender 的请求参数值
            String gender = request.getParameter("gender");
            //获取 color 的请求参数值
            String[] color = request.getParameterValues("color");
            //获取 country 的请求参数值
            String national = request.getParameter("country");
            //获取页面输出流
            PrintStream out = new PrintStream(response.getOutputStream());
            //输出 HTML 页面标签
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Servlet 测试</title>");
            out.println("</head>");
            out.println("<body>");
            //输出请求参数的值：name
            out.println("您的名字：" + name + "<hr/>");
            //输出请求参数的值：gender
            out.println("您的性别：" + gender + "<hr/>");
            //输出请求参数的值：color
            out.println("您喜欢的颜色：");
            for(String c : color)
            {
                 out.println(c + " ");
            }
            out.println("<hr/>");
            out.println("您喜欢的颜色：");
            //输出请求参数的值：national
            out.println("您来自的国家：" + national + "<hr/>");
            out.println("</body>");
            out.println("</html>");
        }
    }
```

上面的 Servlet 类继承了 HttpServlet 类，表明它可作为一个 Servlet 使用。程序的粗体字代码定义了 service 方法来响应用户请求。对比该 Servlet 和 2.9.6 节中的 request1.jsp 页面，该 Servlet 和 request1.jsp 页面的效果完全相同，都通过 HttpServletRequest 获取客户端的 form 请求参数，并显示请求参数的值。

Servlet 和 JSP 的区别在于：

- Servlet 中没有内置对象，原来 JSP 中的内置对象都必须由程序显式创建。
- 对于静态的 HTML 标签，Servlet 都必须使用页面输出流逐行输出。

这也正是笔者在前面介绍的：JSP 是 Servlet 的一种简化，使用 JSP 只需要完成程序员需要输出到客户端的内容，至于 JSP 脚本如何嵌入一个类中，由 JSP 容器完成。而 Servlet 则是个完整的 Java 类，这个类的 service()方法用于生成对客户端的响应。

普通 Servlet 类里的 service()方法的作用，完全等同于 JSP 生成 Servlet 类的_jspService()方法。因此原 JSP 页面的 JSP 脚本、静态 HTML 内容，在普通 Servlet 里都应该转换成 service()方法的代码或输出语句；原 JSP 声明中的内容，对应为在 Servlet 中定义的成员变量或成员方法。

提示：

上面 Servlet 类中粗体字代码所定义的@WebServlet 属于 Servlet 3.0 的 Annotation，下面会详细介绍。

2.10.2 Servlet 的配置

编辑好的 Servlet 源文件并不能响应用户请求，还必须将其编译成 class 文件。将编译后的 FirstServlet.class 文件放在 WEB-INF/classes 路径下，如果 Servlet 有包，则还应该将 class 文件放在对应的包路径下（例如，本例的 FirstServlet.class 就放在 WEB-INF/classes/lee 路径下）。

如果需要直接采用 javac 命令来编译 Servlet 类，则必须将 Servlet API 接口和类添加到系统的 CLASSPATH 环境变量里。也就是将 Tomcat 7 安装目录下 lib 目录中 servlet-api.jar 和 jsp-api.jar 添加到 CLASSPATH 环境变量中。

为了让 Servlet 能响应用户请求，还必须将 Servlet 配置在 Web 应用中。配置 Servlet 时，需要修改 web.xml 文件。

从 Servlet 3.0 开始，配置 Servlet 有两种方式：

- 在 Servlet 类中使用@WebServlet Annotation 进行配置。
- 通过在 web.xml 文件中进行配置。

上面开发 Servlet 类时使用了@WebServlet Annotation 修饰该 Servlet 类，使用@WebServlet 时可指定如表 2.2 所示的常用属性。

表 2.2　@WebServlet 支持的常用属性

属　性	是否必需	说　明
asyncSupported	否	指定该 Servlet 是否支持异步操作模式。关于 Servlet 的异步调用请参考 2.15 节
displayName	否	指定该 Servlet 的显示名
initParams	否	用于为该 Servlet 配置参数
loadOnStartup	否	用于将该 Servlet 配置成 load-on-startup 的 Servlet
name	否	指定该 Servlet 的名称
urlPatterns/value	否	这两个属性的作用完全相同。都指定该 Servlet 处理的 URL

如果打算使用 Annotation 来配置 Servlet，有两点需要指出：

- 不要在 web.xml 文件的根元素（<web-app.../>）中指定 metadata-complete="true"。
- 不要在 web.xml 文件中配置该 Servlet。

如果打算使用 web.xml 文件来配置该 Servlet，则需要配置如下两个部分。

- 配置 Servlet 的名字：对应 web.xml 文件中的<servlet/>元素。
- 配置 Servlet 的 URL：对应 web.xml 文件中的<servlet-mapping/>元素。这一步是可选的。但如果没有为 Servlet 配置 URL，则该 Servlet 不能响应用户请求。

接下来的 Servlet、Filter、Listener 等相关配置，笔者都会同时介绍使用 web.xml 配置、使用 Annotaion 配置的两种方式。但实际项目中只要采用任意一种配置方式即可，不需要同时使用两种配置方式。

因此，配置一个能响应客户请求的 Servlet，至少需要配置两个元素。关于上面的 FirstServlet 的配置如下。

程序清单：codes\02\2.10\servletDemo\WEB-INF\web.xml

```
<!-- 配置 Servlet 的名字 -->
<servlet>
      <!-- 指定 Servlet 的名字，
            相当于指定@WebServlet 的 name 属性 -->
      <servlet-name>firstServlet</servlet-name>
      <!-- 指定 Servlet 的实现类 -->
      <servlet-class>lee.FirstServlet</servlet-class>
</servlet>
<!-- 配置 Servlet 的 URL -->
```

```
<servlet-mapping>
      <!-- 指定 Servlet 的名字 -->
      <servlet-name>firstServlet</servlet-name>
      <!-- 指定 Servlet 映射的 URL 地址，
            相当于指定@WebServlet 的 urlPatterns 属性-->
      <url-pattern>/aa</url-pattern>
</servlet-mapping>
```

如果在 web.xml 文件中增加了如上所示的粗体字配置片段，则该 Servlet 的 URL 为/aa。如果没有在 web.xml 文件中增加上面的粗体字配置片段，那么该 Servlet 类上的@WebServlet Annotation 就会起作用，该 Servlet 的 URL 为/firstServlet。

将 2.9.6 节中的 form.jsp 复制到本应用中，并对其进行简单修改，将 form 表单元素的 action 修改成 aa，在表单域中输入相应的数据，然后单击“提交”按钮，效果如图 2.32 所示。

图 2.32　Servlet 处理用户请求

在这种情况下，Servlet 与 JSP 的作用效果完全相同。

➤➤2.10.3　JSP/Servlet 的生命周期

JSP 的本质就是 Servlet，开发者编写的 JSP 页面将由 Web 容器编译成对应的 Servlet，当 Servlet 在容器中运行时，其实例的创建及销毁等都不是由程序员决定的，而是由 Web 容器进行控制的。

创建 Servlet 实例有两个时机。

- 客户端第一次请求某个 Servlet 时，系统创建该 Servlet 的实例：大部分的 Servlet 都是这种 Servlet。
- Web 应用启动时立即创建 Servlet 实例，即 load-on-startup Servlet。

每个 Servlet 的运行都遵循如下生命周期。

（1）创建 Servlet 实例。

（2）Web 容器调用 Servlet 的 init 方法，对 Servlet 进行初始化。

（3）Servlet 初始化后，将一直存在于容器中，用于响应客户端请求。如果客户端发送 GET 请求，容器调用 Servlet 的 doGet 方法处理并响应请求；如果客户端发送 POST 请求，容器调用 Servlet 的 doPost 方法处理并响应请求。或者统一使用 service()方法处理来响应用户请求。

（4）Web 容器决定销毁 Servlet 时，先调用 Servlet 的 destroy 方法，通常在关闭 Web 应用之时销毁 Servlet。

Servlet 的生命周期如图 2.33 所示。

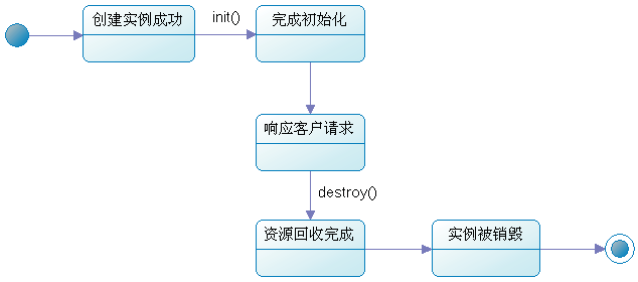

图 2.33　Servlet 的生命周期

2.10.4　load-on-startup Servlet

上一节中已经介绍过，创建 Servlet 实例有两个时机：用户请求之时或应用启动之时。应用启动时就创建 Servlet，通常是用于某些后台服务的 Servlet，或者需要拦截很多请求的 Servlet；这种 Servlet 通常作为应用的基础 Servlet 使用，提供重要的后台服务。

配置 load-on-startup 的 Servlet 有两种方式：

- 在 web.xml 文件中通过<servlet.../>元素的<load-on-startup.../>子元素进行配置。
- 通过@WebServlet Annotation 的 loadOnStartup 属性指定。

<load-on-startup.../>元素或 loadOnStartup 属性都只接收一个整型值，这个整型值越小，Servlet 就越优先实例化。

下面是一个简单的 Servlet，该 Servlet 不响应用户请求，它仅仅执行计时器功能，每隔一段时间会在控制台打印出当前时间。

程序清单：codes\02\2.10\servletDemo\WEB-INF\src\lee\TimerServlet.java

```
@WebServlet(loadOnStartup=1)
public class TimerServlet extends HttpServlet
{
    public void init(ServletConfig config)throws ServletException
    {
        super.init(config);
        Timer t = new Timer(1000,new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                System.out.println(new Date());
            }
        });
        t.start();
    }
}
```

这个 Servlet 没有提供 service()方法，这表明它不能响应用户请求，所以无须为它配置 URL 映射。由于它不能接收用户请求，所以只能在应用启动时实例化。

以上程序中粗体字代码 Annotation 即可将该 Servlet 配置了 load-on-startup Servlet。除此之外，还可以在 web.xml 文件中增加如下配置片段。

程序清单：codes\02\2.10\servletDemo\WEB-INF\web.xml

```
<servlet>
    <!-- Servlet 名 -->
    <servlet-name>timerServlet</servlet-name>
    <!-- Servlet 的实现类 -->
    <servlet-class>lee.TimerServlet</servlet-class>
    <!-- 配置应用启动时，创建 Servlet 实例
        ，相当于指定@WebServlet 的 loadOnStartup 属性-->
    <load-on-startup>1</load-on-startup>
</servlet>
```

以上配置片段中粗体字代码指定 Web 应用启动时，Web 容器将会实例化该 Servlet，且该 Servlet 不能响应用户请求，将一直作为后台服务执行：每隔 1 分钟输出一次系统时间。

2.10.5 访问 Servlet 的配置参数

配置 Servlet 时，还可以增加额外的配置参数。通过使用配置参数，可以实现提供更好的可移植性，避免将参数以硬编码方式写在程序代码中。

为 Servlet 配置参数有两种方式：

- 通过@WebServlet 的 initParams 属性来指定。
- 通过在 web.xml 文件的<servlet.../>元素中添加<init-param.../>子元素来指定。

第二种方式与为 JSP 配置初始化参数极其相似，因为 JSP 的实质就是 Servlet，而且配置 JSP 的实质就是把 JSP 当 Servlet 使用。

访问 Servlet 配置参数通过 ServletConfig 对象完成，ServletConfig 提供如下方法。

- java.lang.String getInitParameter(java.lang.String name)：用于获取初始化参数。

[redacted]

JSP 的内置对象 config 就是此处的 ServletConfig。

下面的 Servlet 将会连接数据库，并执行 SQL 查询，但程序并未直接给出数据库连接信息，而是将数据库连接信息放在 web.xml 文件中进行管理。

程序清单：codes\02\2.10\servletDemo\WEB-INF\src\lee\TestServlet.java

```
@WebServlet(name="testServlet"
      , urlPatterns={"/testServlet"}
      , initParams={
            @WebInitParam(name="driver", value="com.mysql.jdbc.Driver"),
            @WebInitParam(name="url", value="jdbc:mysql://localhost:3306/javaee"),
            @WebInitParam(name="user", value="root"),
            @WebInitParam(name="pass", value="32147")})
public class TestServlet extends HttpServlet
{
      //重写 init 方法，
      public void init(ServletConfig config)
            throws ServletException
      {
            //重写该方法，应该首先调用父类的 init 方法
            super.init(config);
      }
      //响应客户端请求的方法
      public void service(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException,java.io.IOException
      {
            try
            {
                  //获取 ServletConfig 对象
                  ServletConfig config = getServletConfig();
                  //通过 ServletConfig 对象获取配置参数：dirver
                  String driver = config.getInitParameter("driver");
                  //通过 ServletConfig 对象获取配置参数：url
                  String url = config.getInitParameter("url");
                  //通过 ServletConfig 对象获取配置参数：user
                  String user = config.getInitParameter("user");
                  //通过 ServletConfig 对象获取配置参数：pass
                  String pass = config.getInitParameter("pass");
                  //注册驱动
```

```
                Class.forName(driver);
                //获取数据库驱动
                Connection conn = DriverManager.getConnection(url,user,pass);
                //创建 Statement 对象
                Statement stmt = conn.createStatement();
                //执行查询，获取 ResuletSet 对象
                ResultSet rs = stmt.executeQuery("select * from news_inf");
                response.setContentType("text/html;charSet=gbk");
                //获取页面输出流
                PrintStream out = new PrintStream(response.getOutputStream());
                //输出 HTML 标签
                out.println("<html>");
                out.println("<head>");
                out.println("<title>访问 Servlet 初始化参数测试</title>");
                out.println("</head>");
                out.println("<body>");
                out.println("<table bgcolor=\"#9999dd\" border=\"1\"" +
                     "width=\"480\">");
                //遍历结果集
                while(rs.next())
                {
                     //输出结果集内容
                     out.println("<tr>");
                     out.println("<td>" + rs.getString(1) + "</td>");
                     out.println("<td>" + rs.getString(2) + "</td>");
                     out.println("</tr>");
                }
                out.println("</table>");
                out.println("</body>");
                out.println("</html>");
           }
           catch (Exception e)
           {
                e.printStackTrace();
           }
      }
}
```

ServletConfig 获取配置参数的方法和 ServletContext 获取配置参数的方法完全一样，只是 ServletConfig 是取得当前 Servlet 的配置参数，而 ServletContext 是获取整个 Web 应用的配置参数。

以上程序中粗体字@WebServlet 中的 initParams 属性用于为该 Servlet 配置参数，initParams 属性值的每个@WebInitParam 配置一个初始化参数，每个@WebInitParam 可指定如下两个属性。

- name：指定参数名。
- value：指定参数值。

类似地，在 web.xml 文件中为 Servlet 配置参数使用<init-param.../>元素，该元素可以接受如下两个子元素。

- param-name：指定配置参数名。
- param-value：指定配置参数值。

下面是该 Servlet 在 web.xml 文件中的配置片段。

程序清单：codes\02\2.10\servletDemo\WEB-INF\web.xml

```
<servlet>
     <!-- 配置 Servlet 名 -->
     <servlet-name>testServlet</servlet-name>
     <!-- 指定 Servlet 的实现类 -->
     <servlet-class>lee.TestServlet</servlet-class>
     <!-- 配置 Servlet 的初始化参数：driver -->
     <init-param>
          <param-name>driver</param-name>
          <param-value>com.mysql.jdbc.Driver</param-value>
     </init-param>
```

```
        <!-- 配置 Servlet 的初始化参数：url -->
        <init-param>
            <param-name>url</param-name>
            <param-value>jdbc:mysql://localhost:3306/javaee</param-value>
        </init-param>
        <!-- 配置 Servlet 的初始化参数：user -->
        <init-param>
            <param-name>user</param-name>
            <param-value>root</param-value>
        </init-param>
        <!-- 配置 Servlet 的初始化参数：pass -->
        <init-param>
            <param-name>pass</param-name>
            <param-value>32147</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <!-- 确定 Servlet 名 -->
        <servlet-name>testServlet</servlet-name>
        <!-- 配置 Servlet 映射的 URL -->
        <url-pattern>/testServlet</url-pattern>
    </servlet-mapping>
```

以上配置片段的粗体字代码配置了 4 个配置参数，Servlet 通过这 4 个配置参数就可连接数据库。在浏览器中浏览该 Servlet，可看到数据库查询成功（如果数据库的配置正确）。

➤➤2.10.6 使用 Servlet 作为控制器

正如前面见到的，使用 Servlet 作为表现层的工作量太大，所有的 HTML 标签都需要使用页面输出流生成。因此，使用 Servlet 作为表现层有如下三个劣势。

- 开发效率低，所有的 HTML 标签都需使用页面输出流完成。
- 不利于团队协作开发，美工人员无法参与 Servlet 界面的开发。
- 程序可维护性差，即使修改一个按钮的标题，都必须重新编辑 Java 代码，并重新编译。

在标准的 MVC 模式中，Servlet 仅作为控制器使用。Java EE 应用架构正是遵循 MVC 模式的，对于遵循 MVC 模式的 Java EE 应用而言，JSP 仅作为表现层（View）技术，其作用有两点：

- 负责收集用户请求参数。
- 将应用的处理结果、状态数据呈现给用户。

Servlet 则仅充当控制器（Controller）角色，它的作用类似于调度员：所有用户请求都发送给 Servlet，Servlet 调用 Model 来处理用户请求，并调用 JSP 来呈现处理结果；或者 Servlet 直接调用 JSP 将应用的状态数据呈现给用户。

Model 通常由 JavaBean 来充当，所有业务逻辑、数据访问逻辑都在 Model 中实现。实际上隐藏在 Model 下的可能还有很多丰富的组件，例如 DAO 组件、领域对象等。

下面介绍一个使用 Servlet 作为控制器的 MVC 应用，该应用演示了一个简单的登录验证。

下面是本应用的登录页面。

程序清单：codes\02\2.10\servletDemo\login.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> new document </title>
</head>
<body>
<!-- 输出出错提示 -->
<span style="color:red;font-weight:bold">
```

```
<%if (request.getAttribute("err") != null)
{
      out.println(request.getAttribute("err") + "<br/>");
}%>
</span>
请输入用户名和密码:
<!-- 登录表单，该表单提交到一个 Servlet -->
<form id="login" method="post" action="login">
用户名：<input type="text" name="username"/><br/>
密   码：<input type="password" name="pass"/><br/>
<input type="submit" value="登录"/><br/>
</form>
</body>
</html>
```

以上页面除了粗体字代码使用 JSP 脚本输出错误提示之外，该页面其实是一个简单的表单页面，用于收集用户名及密码，并将请求提交到指定 Servlet，该 Servlet 充当控制器角色。

根据严格的 MVC 规范，上面的 login.jsp 页面也不应该被客户端直接访问，客户的请求应该先发送到指定 Servlet，然后由 Servlet 将请求 forward 到该 JSP 页面。

控制器 Servlet 的代码如下。

程序清单：codes\02\2.10\servletDemo\WEB-INF\src\lee\LoginServlet.java

```
@WebServlet(name="login"
      , urlPatterns={"/login"})
public class LoginServlet extends HttpServlet
{
      //响应客户端请求的方法
      public void service(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException,java.io.IOException
      {
            String errMsg = "";
            //Servlet 本身并不输出响应到客户端，因此必须将请求转发
            RequestDispatcher rd;
            //获取请求参数
            String username = request.getParameter("username");
            String pass = request.getParameter("pass");
            try
            {
                  //Servlet 本身，并不执行任何的业务逻辑处理，它调用 JavaBean 处理用户请求
                  DbDao dd = new DbDao("com.mysql.jdbc.Driver",
                        "jdbc:mysql://localhost:3306/liuyan","root","32147");
                  //查询结果集
                  ResultSet rs = dd.query("select pass from user_table "
                        + "where name = ?", username);
                  if (rs.next())
                  {
                        //用户名和密码匹配
                        if (rs.getString("pass").equals(pass))
                        {
                              //获取 session 对象
                              HttpSession session = request.getSession(true);
                              //设置 session 属性，跟踪用户会话状态
                              session.setAttribute("name" , username);
                              //获取转发对象
                              rd = request.getRequestDispatcher("/welcome.jsp");
                              //转发请求
                              rd.forward(request,response);
                        }
                        else
```

```
                        {
                              //用户名和密码不匹配时
                              errMsg += "您的用户名密码不符合,请重新输入";
                        }
                  }
                  else
                  {
                        //用户名不存在时
                        errMsg += "您的用户名不存在,请先注册";
                  }
            }
            catch (Exception e)
            {
                  e.printStackTrace();
            }
            //如果出错，转发到重新登录
            if (errMsg != null && !errMsg.equals(""))
            {
                  rd = request.getRequestDispatcher("/login.jsp");
                  request.setAttribute("err" , errMsg);
                  rd.forward(request,response);
            }
      }
}
```

控制器负责接收客户端的请求，它既不直接对客户端输出响应，也不处理用户请求，只调用 JavaBean 来处理用户请求，如程序中粗体字代码所示；JavaBean 处理结束后，Servlet 根据处理结果，调用不同的 JSP 页面向浏览器呈现处理结果。

上面 Servlet 使用@WebServlet Annotation 为该 Servlet 配置了 URL 为/login，因此向/login 发送的请求将会交给该 Servlet 处理。

下面是本应用中 DbDao 的源代码。

程序清单：codes\02\2.10\servletDemo\WEB-INF\src\lee\DbDao.java

```
public class DbDao
{
      private Connection conn;
      private String driver;
      private String url;
      private String username;
      private String pass;
      public DbDao()
      {
      }
      public DbDao(String driver, String url
            , String username, String pass)
      {
            this.driver = driver;
            this.url = url;
            this.username = username;
            this.pass = pass;
      }
      //下面是各个成员属性的 setter 和 getter 方法
      public void setDriver(String driver) {
            this.driver = driver;
      }
      public void setUrl(String url) {
            this.url = url;
      }
      public void setUsername(String username) {
            this.username = username;
      }
      public void setPass(String pass) {
            this.pass = pass;
      }
```

```
public String getDriver() {
      return (this.driver);
}
public String getUrl() {
      return (this.url);
}
public String getUsername() {
      return (this.username);
}
public String getPass() {
      return (this.pass);
}
//获取数据库连接
public Connection getConnection() throws Exception
{
      if (conn == null)
      {
            Class.forName(this.driver);
            conn = DriverManager.getConnection(url,username,
                  this. pass);
      }
      return conn;
}
//插入记录
public boolean insert(String sql , Object... args)
      throws Exception
{
      PreparedStatement pstmt = getConnection().prepareStatement(sql);
      for (int i = 0; i < args.length; i++ )
      {
            pstmt.setObject( i + 1 , args[i]);
      }
      if (pstmt.executeUpdate() != 1)
      {
            return false;
      }
      return true;
}
//执行查询
public ResultSet query(String sql, Object... args)
      throws Exception
{
      PreparedStatement pstmt = getConnection().prepareStatement(sql);
      for (int i = 0; i < args.length; i++ )
      {
            pstmt.setObject( i + 1, args[i]);
      }
      return pstmt.executeQuery();
}
//执行修改
public void modify(String sql, Object... args)
      throws Exception
{
      PreparedStatement pstmt = getConnection().prepareStatement(sql);
      for (int i = 0; i < args.length ; i++ )
      {
            pstmt.setObject( i + 1 , args[i]);
      }
      pstmt.executeUpdate();
      pstmt.close();
}
//关闭数据库连接的方法
public void closeConn()
      throws Exception
{
      if (conn != null && !conn.isClosed())
      {
```

```
                conn.close();
            }
        }
    }
```

上面 DbDao 负责完成查询、插入、修改等操作。从上面这个应用的结构来看，整个应用的流程非常清晰，下面是 MVC 中各个角色的对应组件。

- M：Model，即模型，对应 JavaBean。
- V：View，即视图，对应 JSP 页面。
- C：Controller，即控制器，对应 Servlet。

[redacted]

本应用需要底层数据库的支持，读者可以向 MySQL 数据库中导入 codes\02\2.10\db.sql 脚本，这些脚本提供了本应用所需的数据库支持。

2.11 JSP 2 的自定义标签

在 JSP 规范的 1.1 版中增加了自定义标签库规范，自定义标签库是一种非常优秀的表现层组件技术。通过使用自定义标签库，可以在简单的标签中封装复杂的功能。

为什么要使用自定义标签呢？主要是为了取代丑陋的 JSP 脚本。在 HTML 页面中插入 JSP 脚本有如下几个坏处：

- JSP 脚本非常丑陋，难以阅读。
- JSP 脚本和 HTML 代码混杂，维护成本高。
- HTML 页面中嵌入 JSP 脚本，导致美工人员难以参与开发。

出于以上三点的考虑，我们需要一种可在页面中使用的标签，这种标签具有和 HTML 标签类似的语法，但有可以完成 JSP 脚本的功能——这种标签就是 JSP 自定义标签。

在 JSP 1.1 规范中开发自定义标签库比较复杂，JSP 2 规范简化了标签库的开发，在 JSP 2 中开发标签库只需如下几个步骤。

1. 开发自定义标签处理类；
2. 建立一个*.tld 文件，每个*.tld 文件对应一个标签库，每个标签库可包含多个标签；
3. 在 JSP 文件中使用自定义标签。

[redacted]

标签库是非常重要的技术，通常来说，初学者、普通开发人员自己开发标签库的机会很少，但如果希望成为高级程序员，或者希望开发通用框架，就需要大量开发自定义标签了。所有的 MVC 框架，如 Struts 2、SpringMVC、JSF 等都提供了丰富的自定义标签。

2.11.1 开发自定义标签类

在 JSP 页面使用一个简单的标签时，底层实际上由标签处理类提供支持，从而可以通过简单的标签来封装复杂的功能，从而使团队更好地协作开发（能让美工人员更好地参与 JSP 页面的开发）。

自定义标签类应该继承一个父类：javax.servlet.jsp.tagext.SimpleTagSupport，除此之外，JSP 自定义标签类还有如下要求：

- 如果标签类包含属性，每个属性都有对应的 getter 和 setter 方法。
- 重写 doTag()方法，这个方法负责生成页面内容。

下面开发一个最简单的自定义标签，该标签负责在页面上输出 HelloWorld。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\lee\HelloWorldTag.java

```
public class HelloWorldTag extends SimpleTagSupport
{
    //重写 doTag 方法，该方法在标签结束生成页面内容
    public void doTag()throws JspException,
        IOException
    {
        //获取页面输出流，并输出字符串
        getJspContext().getOut().write("Hello World "
            + new java.util.Date());
    }
}
```

上面这个标签处理类非常简单，它继承了 SimpleTagSupport 父类，并重写 doTag()方法，而 doTag()方法则负责输出页面内容。该标签没有属性，因此无须提供 setter 和 getter 方法。

➤➤2.11.2 建立 TLD 文件

TLD 是 Tag Library Definition 的缩写，即标签库定义，文件的后缀是 tld，每个 TLD 文件对应一个标签库，一个标签库中可包含多个标签。TLD 文件也称为标签库定义文件。

标签库定义文件的根元素是 taglib，它可以包含多个 tag 子元素，每个 tag 子元素都定义一个标签。通常我们可以到 Web 容器下复制一个标签库定义文件，并在此基础上进行修改即可。例如 Tomcat 7.0，在 webapps\examples\WEB-INF\jsp2 路径下包含了一个 jsp2-example-taglib.tld 文件，这就是一个 TLD 文件的范例。

将该文件复制到 Web 应用的 WEB-INF/路径，或 WEB-INF 的任意子路径下，并对该文件进行简单修改，修改后的 mytaglib.tld 文件代码如下。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\mytaglib.tld

```
<?xml version="1.0" encoding="GBK"?>
<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_0.xsd"
    version="2.0">
    <tlib-version>1.0</tlib-version>
    <short-name>mytaglib</short-name>
    <!-- 定义该标签库的 URI -->
    <uri>http://www.crazyit.org/mytaglib</uri>
    <!-- 定义第一个标签 -->
    <tag>
        <!-- 定义标签名 -->
        <name>helloWorld</name>
        <!-- 定义标签处理类 -->
        <tag-class>lee.HelloWorldTag</tag-class>
        <!-- 定义标签体为空 -->
        <body-content>empty</body-content>
    </tag>
</taglib>
```

上面的标签库定义文件也是一个标准的 XML 文件，该 XML 文件的根元素是 taglib 元素，因此我们每次编写标签库定义文件时都直接添加该元素即可。

taglib 下有如下三个子元素。

- tlib-version：指定该标签库实现的版本，这是一个作为标识的内部版本号，对程序没有太大的作用。
- short-name：该标签库的默认短名，该名称通常也没有太大的用处。
- uri：这个属性非常重要，它指定该标签库的 URI，相当于指定该标签库的唯一标识。如上面

斜体字代码所示，JSP 页面中使用标签库时就是根据该 URI 属性来定位标签库的。

除此之外，taglib 元素下可以包含多个 tag 元素，每个 tag 元素定义一个标签，tag 元素下允许出现如下常用子元素。

- name：该标签库的名称，这个子元素很重要，JSP 页面中就是根据该名称来使用此标签的。
- tag-class：指定标签的处理类，毋庸置疑，这个子元素非常重要，它指定了标签由哪个标签处理类来处理。
- body-content：这个子元素也很重要，它指定标签体内容。该子元素的值可以是如下几个。
- tagdependent：指定标签处理类自己负责处理标签体。
- empty：指定该标签只能作为空标签使用。
- scriptless：指定该标签的标签体可以是静态 HTML 元素、表达式语言，但不允许出现 JSP 脚本。
- JSP：指定该标签的标签体可以使用 JSP 脚本。
- dynamic-attributes：指定该标签是否支持动态属性。只有当定义动态属性标签时才需要该子元素。

因为 JSP 2 规范不再推荐使用 JSP 脚本 ,所以 JSP 2 自定义标签的标签体中不能包含 JSP 脚本。所以，实际上 body-content 元素的值不可以是 JSP。

定义了上面的标签库定义文件后，将标签库文件放在 Web 应用的 WEB-INF 路径或任意子路径下，Java Web 规范会自动加载该文件，则该文件定义的标签库也将生效。

2.11.3 使用标签库

在 JSP 页面中确定指定的标签需要两点。

- 标签库 URI：确定使用哪个标签库。
- 标签名：确定使用哪个标签。

使用标签库分成以下两个步骤。

- 导入标签库：使用 taglib 编译指令导入标签库，就是将标签库和指定前缀关联起来。
- 使用标签：在 JSP 页面中使用自定义标签。

taglib 的语法格式如下：

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

其中 uri 属性确定标签库的 URI，这个 URI 可以确定一个标签库。而 prefix 属性指定标签库前缀，即所有使用该前缀的标签将由此标签库处理。

使用标签的语法格式如下：

```
<tagPrefix:tagName tagAttribute=”tagValue” …>
<tagBody/>
</tagPrefix:tagName>
```

如果该标签没有标签体，则可以使用如下语法格式：

```
<tagPrefix:tagName tagAttribute=”tagValue” …/>
```

上面使用标签的语法里都包含了设置属性值，前面我们介绍的 HelloWorldTag 标签没有任何属性，所以使用该标签只需用<mytag:helloWorld/>即可。其中 mytag 是 taglib 指令为标签库指定的前缀，而 helloWorld 是标签名。

下面是使用 helloWorld 标签的 JSP 页面代码。

程序清单：codes\02\2.11\tagDemo\helloWorldTag.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!-- 导入标签库，指定 mytag 前缀的标签，
     由 http://www.crazyit.org/mytaglib 的标签库处理 -->
<%@ taglib uri="http://www.crazyit.org/mytaglib" prefix="mytag"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title>自定义标签示范</title>
</head>
<body bgcolor="#ffffc0">
<h2>下面显示的是自定义标签中的内容</h2>
<!-- 使用标签 ，其中 mytag 是标签前缀，根据 taglib 的编译指令，
     mytag 前缀将由 http://www.crazyit.org/mytaglib 的标签库处理 -->
<mytag:helloWorld/><br/>
</body>
</html>
```

以上页面中第一行粗体字代码指定了 http://www.crazyit.org/mytaglib 标签库的前缀为 mytag，第二行粗体字代码表明使用 mytag 前缀对应标签库里的 helloWorld 标签。浏览该页面将看到如图 2.34 所示的效果。

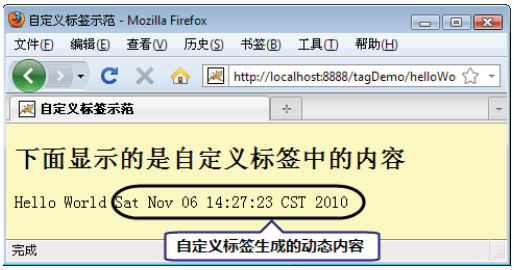

图 2.34　简单标签

2.11.4　带属性的标签

前面的简单标签既没有属性，也没有标签体，用法、功能都比较简单。实际上还有如下两种常用的标签：

- 带属性的标签。
- 带标签体的标签。

正如前面介绍的，带属性标签必须为每个属性提供对应的 setter 和 getter 方法。带属性标签的配置方法与简单标签也略有差别，下面介绍一个带属性标签的示例。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\lee\QueryTag.java

```
public class QueryTag extends SimpleTagSupport
{
     //标签的属性
     private String driver;
     private String url;
     private String user;
     private String pass;
     private String sql;
     //执行数据库访问的对象
     private Connection conn = null;
     private Statement stmt = null;
     private ResultSet rs = null;
     private ResultSetMetaData rsmd = null;
     //省略 driver 属性的 setter 和 getter 方法
     ...
```

```
    //省略 url 属性的 setter 和 getter 方法
    ...
    //省略 user 属性的 setter 和 getter 方法
    ...
    //省略 pass 属性的 setter 和 getter 方法
    ...
    //省略 sql 属性的 setter 和 getter 方法
    ...
    public void doTag()throws JspException,
          IOException
    {
          try
          {
                //注册驱动
                Class.forName(driver);
                //获取数据库连接
                conn = DriverManager.getConnection(url,user,pass);
                //创建 Statement 对象
                stmt = conn.createStatement();
                //执行查询
                rs = stmt.executeQuery(sql);
                rsmd = rs.getMetaData();
                //获取列数目
                int columnCount = rsmd.getColumnCount();
                //获取页面输出流
                Writer out = getJspContext().getOut();
                //在页面输出表格
                out.write("<table border='1' bgColor='#9999cc' width='400'>");
                //遍历结果集
                while (rs.next())
                {
                      out.write("<tr>");
                      //逐列输出查询到的数据
                      for (int i = 1 ; i <= columnCount ; i++ )
                      {
                            out.write("<td>");
                            out.write(rs.getString(i));
                            out.write("</td>");
                      }
                      out.write("</tr>");
                }
          }
          catch(ClassNotFoundException cnfe)
          {
                cnfe.printStackTrace();
                throw new JspException("自定义标签错误" + cnfe.getMessage());
          }
          catch (SQLException ex)
          {
                ex.printStackTrace();
                throw new JspException("自定义标签错误" + ex.getMessage());
          }
          finally
          {
                //关闭结果集
                try
                {
                      if (rs != null)
                            rs.close();
                      if (stmt != null)
                            stmt.close();
                      if (conn != null)
                            conn.close();
                }
                catch (SQLException sqle)
                {
                      sqle.printStackTrace();
```

```
                }
            }
        }
    }
```

上面这个标签稍微复杂一点，它包含了 5 个属性，如程序中粗体字代码所示，程序需要为这 5 个属性提供 setter 和 getter 方法。

该标签输出的内容依然由 doTag()方法决定，该方法会根据 SQL 语句查询数据库，并将查询结果显示在当前页面中。

对于有属性的标签，需要为<tag.../>元素增加<attribute.../>子元素，每个 attribute 子元素定义一个标签属性。<attribute.../>子元素通常还需要指定如下几个子元素。

- name：设置属性名，子元素的值是字符串内容。
- required：设置该属性是否为必需属性，该子元素的值是 true 或 false。
- fragment：设置该属性是否支持 JSP 脚本、表达式等动态内容，子元素的值是 true 或 false。

为了配置上面的 QueryTag 标签，我们需要在 mytaglib.tld 文件中增加如下配置片段。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\mytaglib.tld

```
<!-- 定义第二个标签 -->
<tag>
    <!-- 定义标签名 -->
    <name>query</name>
    <!-- 定义标签处理类 -->
    <tag-class>lee.QueryTag</tag-class>
    <!-- 定义标签体为空 -->
    <body-content>empty</body-content>
    <!-- 配置标签属性:driver -->
    <attribute>
        <name>driver</name>
        <required>true</required>
        <fragment>true</fragment>
    </attribute>
    <!-- 配置标签属性:url -->
    <attribute>
        <name>url</name>
        <required>true</required>
        <fragment>true</fragment>
    </attribute>
    <!-- 配置标签属性:user -->
    <attribute>
        <name>user</name>
        <required>true</required>
        <fragment>true</fragment>
    </attribute>
    <!-- 配置标签属性:pass -->
    <attribute>
        <name>pass</name>
        <required>true</required>
        <fragment>true</fragment>
    </attribute>
    <!-- 配置标签属性:sql -->
    <attribute>
        <name>sql</name>
        <required>true</required>
        <fragment>true</fragment>
    </attribute>
</tag>
```

上面 5 行粗体字代码分别为该标签配置了 driver、url、user、pass 和 sql 5 个属性，并指定这 5 个属性都是必需属性，而且属性值支持动态内容。

配置完毕后，就可在页面中使用标签了，先导入标签库，然后使用标签。使用标签的 JSP 页面片

段如下。

程序清单：codes\02\2.11\tagDemo\queryTag.jsp

```
<!-- 导入标签库，指定 mytag 前缀的标签，
由 http://www.crazyit.org/mytaglib 的标签库处理 -->
<%@ taglib uri="http://www.crazyit.org/mytaglib" prefix="mytag"%>
...
<!-- 其他 HTML 内容 -->
<!-- 使用标签，其中 mytag 是标签前缀，根据 taglib 的编译指令，
mytag 前缀将由 http://www.crazyit.org/mytaglib 的标签库处理 -->
<mytag:query
      driver="com.mysql.jdbc.Driver"
      url="jdbc:mysql://localhost:3306/javaee"
      user="root"
      pass="32147"
      sql="select * from news_inf"/><br/>
```

在浏览器中浏览该页面，效果如图 2.35 所示。

图 2.35　带属性的标签

在 JSP 页面中只需要使用简单的标签，即可完成“复杂”的功能：执行数据库查询，并将查询结果在页面上以表格形式显示。这也正是自定义标签库的目的——以简单的标签，隐藏复杂的逻辑。

当然，并不推荐在标签处理类中访问数据库，因为标签库是表现层组件，它不应该包含任何业务逻辑实现代码，更不应该执行数据库访问，它只应该负责显示逻辑。

提示：

JSTL 是 Sun 提供的一套标签库，这套标签库的功能非常强大。另外，DisplayTag 是 Apache 组织下的一套开源标签库，主要用于生成页面并显示效果。

➤➤2.11.5　带标签体的标签

带标签体的标签，可以在标签内嵌入其他内容（包括静态的 HTML 内容和动态的 JSP 内容），通常用于完成一些逻辑运算，例如判断和循环等。下面以一个迭代器标签为示例，介绍带标签体标签的开发过程。

一样先定义一个标签处理类，该标签处理类的代码如下。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\lee\IteratorTag.java

```
public class IteratorTag extends SimpleTagSupport
{
      //标签属性，用于指定需要被迭代的集合
      private String collection;
      //标签属性，指定迭代集合元素，为集合元素指定的名称
      private String item;
      //省略 collection 属性的 setter 和 getter 方法
```

```
        ...
        //省略 item 属性的 setter 和 getter 方法
        ...
        //标签的处理方法，简单标签处理类只需要重写 doTag 方法
        public void doTag() throws JspException, IOException
        {
              //从 page scope 中获取属性名为 collection 的集合
              Collection itemList = (Collection)getJspContext().
                    getAttribute(collection);
              //遍历集合
              for (Object s : itemList)
              {
                    //将集合的元素设置到 page 范围
                    getJspContext().setAttribute(item, s );
                    //输出标签体
                    getJspBody().invoke(null);
              }
        }
  }
```

上面的标签处理类与前面的处理类并没有太大的不同，该处理类包含两个属性，并为这两个属性提供了 setter 和 getter 方法。标签处理类的 doTag 方法首先从 page 范围内获取了指定名称的 Collection 对象，然后遍历 Collection 对象的元素，每次遍历都调用了 getJspBody()方法，如程序中粗体字代码所示，该方法返回该标签所包含的标签体：JspFragment 对象，执行该对象的 invoke()方法，即可输出标签体内容。该标签的作用是：遍历指定集合，每遍历一个集合元素，即输出标签体一次。

因为该标签的标签体不为空，配置该标签时指定 body-content 为 scriptless，该标签的配置代码片段如下所示。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\mytaglib.tld

```
  <!-- 定义第三个标签 -->
  <tag>
        <!-- 定义标签名 -->
        <name>iterator</name>
        <!-- 定义标签处理类 -->
        <tag-class>lee.IteratorTag</tag-class>
        <!-- 定义标签体不允许出现 JSP 脚本 -->
        <body-content>scriptless</body-content>
        <!-- 配置标签属性:collection -->
        <attribute>
              <name>collection</name>
              <required>true</required>
              <fragment>true</fragment>
        </attribute>
        <!-- 配置标签属性:item -->
        <attribute>
              <name>item</name>
              <required>true</required>
              <fragment>true</fragment>
        </attribute>
  </tag>
```

上面的配置片段中粗体字代码指定该标签的标签体可以是静态 HTML 内容，也可以是表达式语言，但不允许出现 JSP 脚本。

为了测试在 JSP 页面中使用该标签的效果，我们首先把一个 List 对象设置成 page 范围的属性，然后使用该标签来迭代输出 List 集合的全部元素。

JSP 页面中使用该标签的代码片段如下。

程序清单：codes\02\2.11\tagDemo\iteratorTag.jsp

```
  <%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
  <%@ page import="java.util.*"%>
  <!-- 导入标签库，指定 mytag 前缀的标签，
```

```
由 http://www.crazyit.org/mytaglib 的标签库处理 -->
<%@ taglib uri="http://www.crazyit.org/mytaglib" prefix="mytag"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
...
<body>
      <h2>带标签体的标签-迭代器标签</h2><hr/>
      <%
      //创建一个 List 对象
      List<String> a = new ArrayList<String>();
      a.add("疯狂 Java");
      a.add("www.crazyit.org");
      a.add("java");
      //将 List 对象放入 page 范围内
      pageContext.setAttribute("a" , a);
      %>
      <table border="1" bgcolor="#aaaadd" width="300">
            <!-- 使用迭代器标签，对 a 集合进行迭代 -->
            <mytag:iterator collection="a" item="item">
            <tr>
                  <td>${pageScope.item}</td>
            <tr>
            </mytag:iterator>
      </table>
</body>
...
```

上面的页面代码中粗体字代码即可实现通过 iterator 标签来遍历指定集合，浏览该页面即可看到如图 2.36 所示的界面。

图 2.36 显示了使用 iterator 标签遍历集合元素的效果，从 iteratorTag.jsp 页面的代码来看，使用 iterator 标签遍历集合元素比使用 JSP 脚本遍历集合元素要优雅得多，这就是自定义标签的魅力。

图 2.36　带标签体的标签

实际上 JSTL 标签库提供了一套功能非常强大的标签，例如普通的输出标签，像我们刚刚介绍的迭代器标签，还有用于分支判断的标签等，JSTL 都有非常完善的实现。

提示：可能有读者感到疑惑：这个 JSP 页面自己先把多个字符串添加到 ArrayList，然后再使用这个 iterator 标签进行迭代输出，好像意义不是很大啊。实际上这个标签的用处非常大，在严格的 MVC 规范下，JSP 页面只负责显示数据——而数据通常由控制器（Servlet）放入 request 范围内，而 JSP 页面就通过 iterator 标签迭代输出 request 范围内的数据。

2.11.6　以页面片段作为属性的标签

JSP 2 规范的自定义标签还允许直接将一段“页面片段”作为属性，这种方式给自定义标签提供了

更大的灵活性。

以“页面片段”为属性的标签与普通标签区别并不大，只有两个简单的改变：

- 标签处理类中定义类型为 JspFragment 的属性，该属性代表了“页面片段”。
- 使用标签库时，通过<jsp:attribute.../>动作指令为标签库属性指定值。

下面的程序定义了一个标签处理类，该标签处理类中定义了一个 JspFragment 类型的属性，即表明该标签允许使用“页面片段”类型的属性。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\lee\FragmentTag.java

```
public class FragmentTag extends SimpleTagSupport
{
      private JspFragment fragment;
      //fragment 属性的 setter 和 getter 方法
      public void setFragment(JspFragment fragment)
      {
            this.fragment = fragment;
      }
      public JspFragment getFragment()
      {
            return this.fragment;
      }
      @Override
      public void doTag() throws JspException, IOException
      {
            JspWriter out = getJspContext().getOut();
            out.println("<div style='padding:10px;border:1px solid black'>");
            out.println("<h3>下面是动态传入的 JSP 片段</h3>");
            //调用、输出“页面片段”
            fragment.invoke( null );
            out.println("</div");
      }
}
```

上面的程序中定义了 fragment 属性，该属性代表了使用该标签时的“页面片段”，配置该标签与配置普通标签并无任何区别，增加如下配置片段即可。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\mytaglib.tld

```
<tag>
      <!-- 定义标签名 -->
      <name>fragment</name>
      <!-- 定义标签处理类 -->
      <tag-class>lee.FragmentTag</tag-class>
      <!-- 指定该标签不支持标签体 -->
      <body-content>empty</body-content>
      <!-- 定义标签属性：fragment -->
      <attribute>
            <name>fragment</name>
            <required>true</required>
            <fragment>true</fragment>
      </attribute>
</tag>
```

从上面标签库的配置片段来看，这个自定义标签并没有任何特别之处，就是一个普通的带属性标签，该标签的标签题为空。

由于该标签需要一个 fragment 属性，该属性的类型为 JspFragment，因此使用该标签时需要使用<jsp:attribute.../>动作指令来设置属性值，如以下代码片段所示。

程序清单：codes\02\2.11\tagDemo\fragmentTag.jsp

```
<h2>下面显示的是自定义标签中的内容</h2>
<mytag:fragment>
      <!-- 使用 jsp:attribute 标签传入 fragment 参数 -->
      <jsp:attribute name="fragment">
```

```
            <!-- 下面是动态的 JSP 页面片段 -->
            <mytag:helloWorld/>
        </jsp:attribute>
    </mytag:fragment>
    <br/>
    <mytag:fragment>
        <jsp:attribute name="fragment">
            <!-- 下面是动态的 JSP 页面片段 -->
            ${pageContext.request.remoteAddr}
        </jsp:attribute>
    </mytag:fragment>
```

上面的代码片段中粗体字代码用于为标签的 fragment 属性赋值，第一个例子使用了另一个简单标签来生成页面片段；第二个例子使用了 JSP 2 的 EL 来生成页面片段；在浏览器中浏览该页面，将看到如图 2.37 所示效果。

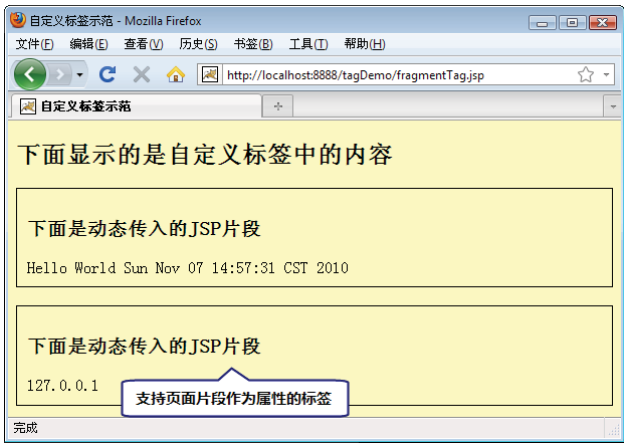

图 2.37　页面片段为属性的标签

➤➤2.11.7　动态属性的标签

前面介绍带属性标签时，那些标签的属性个数是确定的，属性名也是确定的，绝大部分情况下这种带属性的标签能处理得很好，但在某些特殊情况下，我们需要传入自定义标签的属性个数是不确定的，属性名也不确定，这就需要借助于动态属性的标签了。

动态属性标签比普通标签多了如下两个额外要求：

- 标签处理类还需要实现 DynamicAttributes 接口。
- 配置标签时通过<dynamic-attributes.../>子元素指定该标签支持动态属性。

下面是一个动态属性标签的处理类。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\lee\DynaAttributesTag.java

```
public class DynaAttributesTag
    extends SimpleTagSupport implements DynamicAttributes
{
    //保存每个属性名的集合
    private ArrayList<String> keys = new ArrayList<String>();
    //保存每个属性值的集合
    private ArrayList<Object> values = new ArrayList<Object>();
    @Override
    public void doTag() throws JspException, IOException
    {
        JspWriter out = getJspContext().getOut();
        //此处只是简单地输出每个属性
        out.println("<ol>");
        for( int i = 0; i < keys.size(); i++ )
        {
```

```
                String key = keys.get( i );
                Object value = values.get( i );
                out.println( "<li>" + key + " = " + value + "</li>" );
            }
            out.println("</ol>");
        }
        @Override
        public void setDynamicAttribute( String uri, String localName,
            Object value )
            throws JspException
        {
            //添加属性名
            keys.add( localName );
            //添加属性值
            values.add( value );
        }
    }
```

上面的标签处理类实现了 DynaAttributesTag 接口，就是动态属性标签处理类必须实现的接口，实现该接口必须实现 setDynaAttributes 方法，该方法用于为该标签处理类动态地添加属性名和属性值。标签处理类使用 ArrayList<String>类型的 keys 属性来保存标签的所有属性名，使用 ArrayList<Object>类型的 values 属性来保存标签的所有属性值。

配置该标签时需要额外地指定<dynamic-attributes.../>子元素，表明该标签是带动态属性的标签，下面是该标签的配置片段。

程序清单：codes\02\2.11\tagDemo\WEB-INF\src\mytaglib.tld

```
<!-- 定义接受动态属性的标签 -->
<tag>
    <name>dynaAttr</name>
    <tag-class>lee.DynaAttributesTag</tag-class>
    <body-content>empty</body-content>
    <!-- 指定支持动态属性 -->
    <dynamic-attributes>true</dynamic-attributes>
</tag>
```

上面的配置片段指定该标签支持动态属性。

一旦定义了动态属性的标签，接下来在页面中使用该标签时将十分灵活，我们可以为该标签设置任意的属性，如以下页面片段所示。

程序清单：codes\02\2.11\tagDemo\dynaAttrTag.jsp

```
<!-- 导入标签库，指定 mytag 前缀的标签，
    由 http://www.crazyit.org/mytaglib 的标签库处理 -->
<%@ taglib uri="http://www.crazyit.org/mytaglib" prefix="mytag"%>
...
<h2>下面显示的是自定义标签中的内容</h2>
<h4>指定两个属性</h4>
<mytag:dynaAttr name="crazyit" url="crazyit.org"/><br/>
<h4>指定四个属性</h4>
<mytag:dynaAttr 书名="疯狂 Java 讲义" 价格="99.0"
    出版时间="2008 年" 描述="Java 图书"/><br/>
```

上面的页面片段中使用<mytag:dynaAttr.../>时十分灵活：可以根据需要动态地传入任意多个属性，如以上粗体字代码所示。不管传入多少个属性，这个标签都可以处理得很好，使用浏览器访问该页面将看到如图 2.38 所示效果。

图 2.38　动态属性的标签

2.12　Filter 介绍

Filter 可认为是 Servlet 的一种“加强版”，它主要用于对用户请求进行预处理，也可以对 HttpServletResponse 进行后处理，是个典型的处理链。Filter 也可对用户请求生成响应，这一点与 Servlet 相同，但实际上很少会使用 Filter 向用户请求生成响应。使用 Filter 完整的流程是：Filter 对用户请求进行预处理，接着将请求交给 Servlet 进行处理并生成响应，最后 Filter 再对服务器响应进行后处理。

Filter 有如下几个用处。

- 在 HttpServletRequest 到达 Servlet 之前，拦截客户的 HttpServletRequest。
- 根据需要检查 HttpServletRequest，也可以修改 HttpServletRequest 头和数据。
- 在 HttpServletResponse 到达客户端之前，拦截 HttpServletResponse。
- 根据需要检查 HttpServletResponse，也可以修改 HttpServletResponse 头和数据。

Filter 有如下几个种类。

- 用户授权的 Filter：Filter 负责检查用户请求，根据请求过滤用户非法请求。
- 日志 Filter：详细记录某些特殊的用户请求。
- 负责解码的 Filter：包括对非标准编码的请求解码。
- 能改变 XML 内容的 XSLT Filter 等。
- Filter 可负责拦截多个请求或响应；一个请求或响应也可被多个 Filter 拦截。

创建一个 Filter 只需两个步骤：

① 创建 Filter 处理类。
② web.xml 文件中配置 Filter。

➤➤2.12.1　创建 Filter 类

创建 Filter 必须实现 javax.servlet.Filter 接口，在该接口中定义了如下三个方法。

- void init（FilterConfig config）：用于完成 Filter 的初始化。
- void destroy()：用于 Filter 销毁前，完成某些资源的回收。
- void doFilter（ServletRequest request，ServletResponse response,FilterChain chain）：实现过滤功能，该方法就是对每个请求及响应增加的额外处理。

下面介绍一个日志 Filter，这个 Filter 负责拦截所有的用户请求，并将请求的信息记录在日志中。

程序清单：codes\02\2.12\filterTest\WEB-INF\src\lee\LogFilter.java

```
@WebFilter(filterName="log"
      ,urlPatterns={"/*"})
public class LogFilter implements Filter
{
      //FilterConfig 可用于访问 Filter 的配置信息
      private FilterConfig config;
      //实现初始化方法
      public void init(FilterConfig config)
      {
            this.config = config;
      }
      //实现销毁方法
      public void destroy()
      {
            this.config = null;
      }
      //执行过滤的核心方法
      public void doFilter(ServletRequest request,
            ServletResponse response, FilterChain chain)
            throws IOException,ServletException
      {
            //---------下面代码用于对用户请求执行预处理---------
            //获取 ServletContext 对象，用于记录日志
            ServletContext context = this.config.getServletContext();
            long before = System.currentTimeMillis();
            System.out.println("开始过滤...");
            //将请求转换成 HttpServletRequest 请求
            HttpServletRequest hrequest = (HttpServletRequest)request;
            //输出提示信息
            System.out.println("Filter 已经截获到用户的请求的地址： " +
                  hrequest.getServletPath());
            //Filter 只是链式处理，请求依然放行到目的地址
            chain.doFilter(request, response);
            //---------下面代码用于对服务器响应执行后处理---------
            long after = System.currentTimeMillis();
            //输出提示信息
            System.out.println("过滤结束");
            //输出提示信息
            System.out.println("请求被定位到" + hrequest.getRequestURI() +
                  "   所花的时间为: " + (after - before));
      }
}
```

上面的程序中粗体字代码实现了 doFilter()方法，实现该方法就可实现对用户请求进行预处理，也可实现对服务器响应进行后处理——它们的分界线为是否调用了 chain.doFilter()，执行该方法之前，即对用户请求进行预处理；执行该方法之后，即对服务器响应进行后处理。

在上面的请求 Filter 中，仅在日志中记录请求的 URL，对所有的请求都执行 chain.doFilter (request,reponse)方法，当 Filter 对请求过滤后，依然将请求发送到目的地址。如果需要检查权限，可以在 Filter 中根据用户请求的 HttpSession，判断用户权限是否足够。如果权限不够，直接调用重定向即可，无须调用 chain.doFilter(request,reponse)方法。

➤➤2.12.2 配置 Filter

前面已经提到，Filter 可以认为是 Servlet 的“增强版”，因此配置 Filter 与配置 Servlet 非常相似，都需要配置如下两个部分：

- 配置 Filter 名。
- 配置 Filter 拦截 URL 模式。

区别在于，Servlet 通常只配置一个 URL，而 Filter 可以同时拦截多个请求的 URL。因此，在配置 Filter 的 URL 模式时通常会使用模式字符串，使得 Filter 可以拦截多个请求。与配置 Servlet 相似的是，配置 Filter 同样有两种方式：

- 在 Filter 类中通过 Annotation 进行配置。
- 在 web.xml 文件中通过配置文件进行配置。

上面 Filter 类的粗体字代码使用@WebFilter 配置该 Filter 的名字为 log，它会拦截向/*发送的所有的请求。

@WebFilter 修饰一个 Filter 类，用于对 Filter 进行配置，它支持如表 2.3 所示的常用属性。

表 2.3　@WebFilter 支持的常用属性

属　性	是否必需	说　明
asyncSupported	否	指定该 Filter 是否支持异步操作模式。关于 Filter 的异步调用请参考 2.15 节
dispatcherTypes	否	指定该 Filter 仅对那种 dispatcher 模式的请求进行过滤。该属性支持 ASYNC、ERROR、FORWARD、INCLUDE、REQUEST 这 5 个值的任意组合。默认值为同时过滤 5 种模式的请求
displayName	否	指定该 Filter 的显示名
filterName		指定该 Filter 的名称
initParams	否	用于为该 Filter 配置参数
servletNames	否	该属性值可指定多个 Servlet 的名称，用于指定该 Filter 仅对这几个 Servlet 执行过滤
urlPatterns/value	否	这两个属性的作用完全相同。都指定该 Filter 所拦截的 URL

在 web.xml 文件中配置 Filter 与配置 Servlet 非常相似，需要为 Filter 指定它所过滤的 URL，并且也可以为 Filter 配置参数。

在 web.xml 文件中为该 Filter 增加如下配置片段：

```
<!-- 定义 Filter -->
<filter>
      <!-- Filter 的名字，相当于指定@WebFilter
            的 filterName 属性 -->
      <filter-name>log</filter-name>
      <!-- Filter 的实现类 -->
      <filter-class>lee.LogFilter</filter-class>
</filter>
<!-- 定义 Filter 拦截的 URL 地址 -->
<filter-mapping>
      <!-- Filter 的名字 -->
      <filter-name>log</filter-name>
      <!-- Filter 负责拦截的 URL，相当于指定@WebFilter
            的 urlPatterns 属性 -->
      <url-pattern>/*</url-pattern>
</filter-mapping>
```

上面的粗体字代码用于配置该 Filter，从这些代码中可以看出配置 Filter 与配置 Servlet 非常相似，只是配置 Filter 时指定 url-pattern 为/*，即表示该 Filter 会拦截所有用户请求。该 Filter 并未对客户端请求进行额外的处理，仅仅在日志中简要记录请求的信息。

为该 Web 应用提供任意一个 JSP 页面，并通过浏览器来访问该 JSP 页面，即可在 Tomcat 的控制台看到如图 2.39 所示的信息。

图 2.39　Filter 过滤客户端请求

实际上 Filter 和 Servlet 极其相似，区别只是 Filter 的 doFilter()方法里多了一个 FilterChain 的参数，

通过该参数可以控制是否放行用户请求。在实际项目中，Filter 里 doFilter()方法里的代码就是从多个 Servlet 的 service()方法里抽取的通用代码，通过使用 Filter 可以实现更好的代码复用。

假设系统有包含多个 Servlet，这些 Servlet 都需要进行一些的通用处理：比如权限控制、记录日志等，这将导致在这些 Servlet 的 service 方法中有部分代码是相同的——为了解决这种代码重复的问题，我们可以考虑把这些通用处理提取到 Filter 中完成，这样各 Servlet 中剩下的只是特定请求相关的处理代码，而通用处理则交给 Filter 完成。图 2.40 显示了 Filter 的用途。

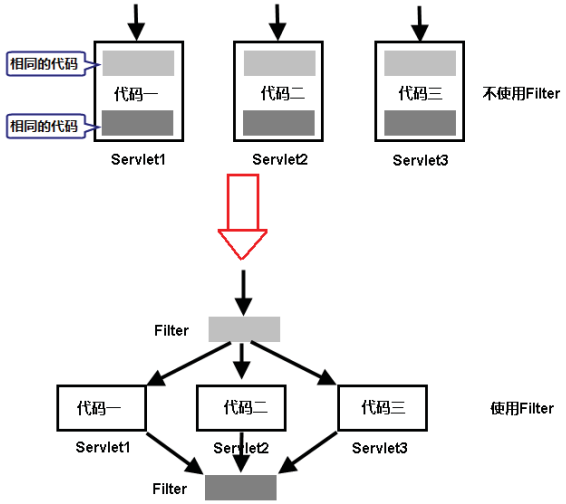

图 2.40　Filter 的作用

由于 Filter 和 Servlet 如此相似，所以 Filter 和 Servlet 具有完全相同的生命周期行为，且 Filter 也可以通过<init-param.../>元素或@WebFilter 的 initParams 属性来配置初始化参数，获取 Filter 的初始化参数则使用 FilterConfig 的 getInitParameter()方法。

下面将定义一个较为实用的 Filter，该 Filter 对用户请求进行过滤，Filter 将通过 doFilter 方法来设置 request 编码的字符集，从而避免每个 JSP、Servlet 都需要设置；而且还会验证用户是否登录，如果用户没有登录，系统直接跳转到登录页面。

下面是该 Filter 的源代码。

程序清单：codes\02\2.12\ filterTest\WEB-INF\src\lee\AuthorityFilter.java

```
@WebFilter(filterName="authority"
      , urlPatterns={"/*"}
      , initParams={
            @WebInitParam(name="encoding", value="GBK"),
            @WebInitParam(name="loginPage", value="/login.jsp"),
            @WebInitParam(name="proLogin", value="/proLogin.jsp")})
public class AuthorityFilter implements Filter
{
      //FilterConfig 可用于访问 Filter 的配置信息
      private FilterConfig config;
      //实现初始化方法
      public void init(FilterConfig config)
      {
            this.config = config;
      }
      //实现销毁方法
      public void destroy()
      {
            this.config = null;
      }
      //执行过滤的核心方法
      public void doFilter(ServletRequest request,
            ServletResponse response, FilterChain chain)
```

```
            throws IOException,ServletException
        {
            //获取该 Filter 的配置参数
            String encoding = config.getInitParameter("encoding");
            String loginPage = config.getInitParameter("loginPage");
            String proLogin = config.getInitParameter("proLogin");
            //设置 request 编码用的字符集
            request.setCharacterEncoding(encoding);        //①
            HttpServletRequest requ = (HttpServletRequest)request;
            HttpSession session = requ.getSession(true);
            //获取客户请求的页面
            String requestPath = requ.getServletPath();
            //如果 session 范围的 user 为 null，即表明没有登录
            //且用户请求的既不是登录页面，也不是处理登录的页面
            if( session.getAttribute("user") == null
                && !requestPath.endsWith(loginPage)
                && !requestPath.endsWith(proLogin))
            {
                //forward 到登录页面
                request.setAttribute("tip" , "您还没有登录");
                request.getRequestDispatcher(loginPage)
                    .forward(request, response);
            }
            //"放行"请求
            else
            {
                chain.doFilter(request, response);
            }
        }
    }
```

上面 Filter 的 doFilter 方法里开始三行粗体字代码用于获取 Filter 的配置参数，而程序中的粗体字代码则是此 Filter 的核心，①号代码按配置参数设置了 request 编码所用的字符集，接下来的粗体字代码判断 session 范围内是否有 user 属性——没有该属性即认为没有登录，如果既没有登录，而且请求地址也不是登录页和处理登录页，系统直接跳转到登录页面。

通过@WebFilter 的 initParams 属性可以为该 Filter 配置初始化参数，它可以接受多个@WebInitParam，每个@WebInitParam 指定一个初始化参数。

在 web.xml 文件中也使用<init-param.../>元素为该 Filter 配置参数，与配置 Servlet 初始化参数完全相同。

如果需要在 web.xml 文件中配置该 Filter，该 Filter 的配置片段如下：

```
<!-- 定义 Filter -->
<filter>
    <!-- Filter 的名字 -->
    <filter-name>authority</filter-name>
    <!-- Filter 的实现类 -->
    <filter-class>lee.AuthorityFilter</filter-class>
    <!-- 下面三个 init-param 元素配置了三个参数 -->
    <init-param>
        <param-name>encoding</param-name>
        <param-value>GBK</param-value>
    </init-param>
    <init-param>
        <param-name>loginPage</param-name>
        <param-value>/login.jsp</param-value>
    </init-param>
    <init-param>
        <param-name>proLogin</param-name>
        <param-value>/proLogin.jsp</param-value>
    </init-param>
</filter>
<!-- 定义 Filter 拦截的 URL 地址 -->
<filter-mapping>
```

```
        <!-- Filter 的名字 -->
        <filter-name>authority</filter-name>
        <!-- Filter 负责拦截的 URL -->
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

上面的配置片段中粗体字代码为该 Filter 指定了三个配置参数，指定 loginPage 为/login.jsp，proLogin 为/proLogin.jsp，这表明，如果没有登录该应用，普通用户只能访问/login.jsp 和/proLogin.jsp 页面。只有当用户登录该应用后才可自由访问其他页面。

➤➤2.12.3 使用 URL Rewrite 实现网站伪静态

对于以 JSP 为表现层开发的动态网站来说，用户访问的 URL 通常有如下形式：

```
xxx.jsp?param=value...
```

大部分搜索引擎都会优先考虑收录静态的 HTML 页面，而不是这种动态的*.jsp、*.php 页面。但实际上绝大部分网站都是动态的，不可能全部是静态的 HTML 页面，因此互联网上的大部分网站都会考虑使用伪静态——就是将*.jsp、*.php 这种动态 URL 伪装成静态的 HTML 页面。

对于 Java Web 应用来说，要实现伪静态非常简单：可以通过 Filter 拦截所有发向*.html 请求，然后按某种规则将请求 forward 到实际的*.jsp 页面即可。现有的 URL Rewrite 开源项目为这种思路提供了实现，使用 URL Rewrite 实现网站伪静态也很简单。

下面详细介绍如何利用 URL Rewrite 实现网站伪静态：

① 登录 http://www.tuckey.org/urlrewrite/站点下载 Url Rewrite 的最新版本，笔者成书时该项目的最新版本是 3.2，建议读者也下载该版本的 Url Rewrite。

提示：笔者成书时，不知为何该站点又被电信网络“封”了，笔者通过代理服务器才可访问该站点，如果读者的网络无法登录该站点，请使用代理服务器登录。

② 下载 URL Rewrite 应下载其 src 项（urlrewritefilter-3.2.0-src.zip），下载完成后得到一个 urlrewritefilter-3.2.0-src.zip 文件，将该压缩文件解压缩，得到如下文件结构。

- api：该路径下存放了 URL Rewrite 项目的 API 文档。
- lib：该路径下存放了 URL Rewrite 项目的编译和运行所需的第三方类库。
- manual：该路径下存放了 URL Rewrite 项目使用手册。
- src：该路径下存放了 URL Rewrite 项目的源代码。
- webapp：该路径是一个 URL Rewrite 的示例应用。
- LICENSE.txt 等杂项文档。

③ 在 web.xml 文件中配置启用 URL Rewrite Filter，在 web.xml 文件中增加如下配置片段。

程序清单：codes\02\2.12\urlrewrite\WEB-INF\web.xml

```
<!-- 配置 Url Rewrite 的 Filter -->
<filter>
    <filter-name>UrlRewriteFilter</filter-name>
    <filter-class>org.tuckey.web.filters.urlrewrite.UrlRewriteFilter</filter-class>
</filter>
<!-- 配置 Url Rewrite 的 Filter 拦截所有请求 -->
<filter-mapping>
    <filter-name>UrlRewriteFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

上面的配置片段指定使用 URL Rewrite Filter 拦截所有的用户请求。

④ 在应用的 WEB-INF 路径下增加 urlrewrite.xml 文件，该文件定义了伪静态映射规则，这份伪

静态规则是基于正则表达式的。

下面是本应用所使用的 urlrewrite.xml 伪静态规则文件。

程序清单：codes\02\2.12\urlrewrite\WEB-INF\urlrewrite.xml

```
<?xml version="1.0" encoding="GBK"?>
<!DOCTYPE urlrewrite PUBLIC "-//tuckey.org//DTD UrlRewrite 3.2//EN"
      "http://tuckey.org/res/dtds/urlrewrite3.2.dtd">
<urlrewrite>
    <rule>
            <!-- 所有配置如下正则表达式的请求 -->
        <from>/userinf-(\w*).html</from>
            <!-- 将被 forward 到如下 JSP 页面，其中$1 代表
                  上面第一个正则表达式所匹配的字符串 -->
        <to type="forward">/userinf.jsp?username=$1</to>
    </rule>
</urlrewrite>
```

上面的规则文件中只定义了一个简单的规则：所有发向/userinf-(\w*).html 的请求都将被 forward 到 user.jsp 页面，并将(\w*)正则表达式所匹配的内容作为 username 参数值。根据这个伪静态规则，我们应该为该应用提供一个 userinf.jsp 页面，该页面只是一个模拟了一个显示用户信息的页面，该页面代码如下。

程序清单：codes\02\2.12\urlrewrite\userinf.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%
//获取请求参数
String user = request.getParameter("username");
%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> <%=user%>的个人信息 </title>
</head>
<body>
<%
//此处应该通过数据库读取该用户对应的信息
//此处只是模拟，因此简单输出：
out.println("现在时间是：" + new java.util.Date() + "<br/>");
out.println("用户名：" + user);
%>
</body>
</html>
```

上面的页面中粗体字代码 username 请求参数来输出用户信息，但因为系统使用了 URL Rewrite，因此用户可以请求类似于 userinf-xxx.html 页面，图 2.41 显示了“伪静态”示意。

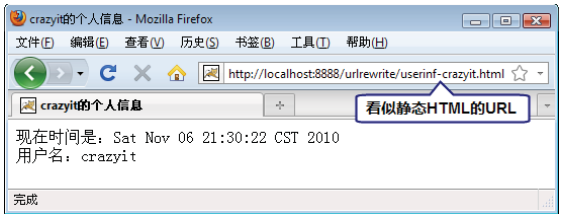

图 2.41　伪静态示意

2.13　Listener 介绍

当 Web 应用在 Web 容器中运行时，Web 应用内部会不断地发生各种事件：如 Web 应用被启动、Web 应用被停止，用户 session 开始、用户 session 结束、用户请求到达等，通常来说，这些 Web 事件对开发者是透明的。

实际上，Servlet API 提供了大量监听器来监听 Web 应用的内部事件，从而允许当 Web 内部事件发生时回调事件监听器内的方法。

使用 Listener 只需要两个步骤：

1. 定义 Listener 实现类。
2. 通过 Annotation 或在 web.xml 文件中配置 Listerner。

2.13.1 实现 Listener 类

与 AWT 事件编程完全相似，监听不同 Web 事件的监听器也不相同。常用的 Web 事件监听器接口有如下几个。

- ServletContextListener：用于监听 Web 应用的启动和关闭。
- ServletContextAttributeListener：用于监听 ServletContext 范围（application）内属性的改变。
- ServletRequestListener：用于监听用户请求。
- ServletRequestAttributeListener：用于监听 ServletRequest 范围（request）内属性的改变。
- HttpSessionListener：用于监听用户 session 的开始和结束。
- HttpSessionAttributeListener：用于监听 HttpSession 范围（session）内属性的改变。

下面先以 ServletContextListener 为例来介绍 Listener 的开发和使用，ServletContextListener 用于监听 Web 应用的启动和关闭。该 Listener 类必须实现 ServletContextListener 接口，该接口包含如下两个方法。

- contextInitialized(ServletContextEvent sce)：启动 Web 应用时，系统调用 Listener 的该方法。
- contextDestroyed(ServletContextEvent sce)：关闭 Web 应用时，系统调用 Listener 的该方法。

通过上面的介绍不难看出，ServletContextListener 的作用有点类似于 load-on-startup Servlet，都可用于在 Web 应用启动时，回调方法来启动某些后台程序，这些后台程序负责为系统运行提供支持。

下面将创建一个获取数据库连接的 Listener，该 Listener 会在应用启动时获取数据库连接，并将获取到的连接设置成 application 范围内的属性。下面是该 Listener 的代码。

程序清单：codes\02\2.13\listenerTest\WEB-INF\src\lee\GetConnListener.java

```
@WebListener
public class GetConnListener implements ServletContextListener
{
      //应用启动时，该方法被调用
      public void contextInitialized(ServletContextEvent sce)
      {
            try
            {
                  //取得该应用的 ServletContext 实例
                  ServletContext application = sce.getServletContext();
                  //从配置参数中获取驱动
                  String driver = application.getInitParameter("driver");
                  //从配置参数中获取数据库 url
                  String url = application.getInitParameter("url");
                  //从配置参数中获取用户名
                  String user = application.getInitParameter("user");
                  //从配置参数中获取密码
                  String pass = application.getInitParameter("pass");
                  //注册驱动
                  Class.forName(driver);
                  //获取数据库连接
```

```
                Connection conn = DriverManager.getConnection(url
                    , user , pass);
                //将数据库连接设置成 application 范围内的属性
                application.setAttribute("conn" , conn);
            }
            catch (Exception ex)
            {
                System.out.println("Listener 中获取数据库连接出现异常"
                    + ex.getMessage());
            }
        }
        //应用关闭时，该方法被调用
        public void contextDestroyed(ServletContextEvent sce)
        {
            //取得该应用的 ServletContext 实例
            ServletContext application = sce.getServletContext();
            Connection conn = (Connection)application.getAttribute("conn");
            //关闭数据库连接
            if (conn != null)
            {
                try
                {
                    conn.close();
                }
                catch (SQLException ex)
                {
                    ex.printStackTrace();
                }
            }
        }
    }
```

上面的程序中粗体字代码重写了 ServletContextListener 的 contextInitialized()、contextDestroyed()方法，这两个方法分别在应用启动、关闭时被触发。上面 ServletContextListener 的两个方法分别实现获取数据库连接、关闭数据库连接的功能，这些功能都是为整个 Web 应用提供服务的。

程序中斜体字代码用于获取配置参数，细心的读者可能已经发现 ServletContextListener 获取的是 Web 应用的配置参数，而不是像 Servlet 和 Filter 获取本身的配置参数。这是因为配置 Listener 时十分简单，只要简单地指定 Listener 实现类即可，不能配置初始化参数。

➤➤2.13.2 配置 Listener

配置 Listener 只要向 Web 应用注册 Listener 实现类即可，无须配置参数之类的东西，因此十分简单。为 Web 应用配置 Listener 也有两种方式：

- 使用@WebListener 修饰 Listener 实现类即可。
- 在 web.xml 文档中使用<listener.../>元素进行配置。

使用@WebListener 时通常无须指定任何属性，只要使用该 Annotation 修饰 Listener 实现类即可向 Web 应用注册该监听器。

在 web.xml 中使用<listener.../>元素进行配置时只要配置如下子元素即可。

- listener-class：指定 Listener 实现类。

若将 ServletContextListener 配置在 Web 容器中，且 Web 容器（支持 Servlet 2.3 以上规范）支持 Listener，则该 ServletContextListener 将可以监听 Web 应用的启动、关闭。

如果选择 web.xml 文件来配置 Listener，则应在 web.xml 文档中增加如下配置片段：

```
<listener>
    <!-- 指定 Listener 的实现类 -->
    <listener-class>lee.GetConnListener</listener-class>
</listener>
```

上面的配置片段向 Web 应用注册了一个 Listener，其实现类为 lee.GetConnListener。当 Web 应用被启动时，该 Listener 的 contextInitialized 方法被触发，该方法会获取一个 JDBC Connection，并放入 application 范围内，这样我们的所有 JSP 页面都可通过 application 获取数据库连接，从而可以非常方便地进行数据库访问。

本例中的 ServletContextListener 把一个数据库连接（Connection 实例）设置成 application 属性，这样将导致所有页面都使用相同的 Connection 实例，实际上这种做法的性能非常差。较为实用的做法是：应用启动时将一个数据源（javax.sql.DataSource 实例）设置成 application 属性，而所有 JSP 页面都通过 DataSource 实例来取得数据库连接，再进行数据库访问，这样就会好得多。关于数据库连接池的介绍请参看疯狂 Java 体系的《疯狂 Java 讲义》的 13.8 节。

2.13.3 使用 ServletContextAttributeListener

ServletContextAttributeListener 用于监听 ServletContext（application）范围内属性的变化，实现该接口的监听器需要实现如下三个方法。

- attributeAdded(ServletContextAttributeEvent event)：当程序把一个属性存入 application 范围时触发该方法。
- attributeRemoved(ServletContextAttributeEvent event)：当程序把一个属性从 application 范围删除时触发该方法。
- attributeReplaced(ServletContextAttributeEvent event)：当程序替换 application 范围内的属性时将触发该方法。

下面是一个监听 ServletContext 范围内属性改变的 Listener。

程序清单：codes\02\2.13\listenerTest\WEB-INF\src\lee\MyServletContextAttributeListener.java

```
@WebListener
public class MyServletContextAttributeListener
      implements ServletContextAttributeListener
{
      //当程序向 application 范围添加属性时触发该方法
      public void attributeAdded(ServletContextAttributeEvent event)
      {
            ServletContext application = event.getServletContext();
            //获取添加的属性名和属性值
            String name = event.getName();
            Object value = event.getValue();
            System.out.println(application + "范围内添加了名为"
                  + name + ", 值为" + value + "的属性!");
      }
      //当程序从 application 范围删除属性时触发该方法
      public void attributeRemoved(ServletContextAttributeEvent event)
      {
            ServletContext application = event.getServletContext();
            //获取被删除的属性名和属性值
            String name = event.getName();
            Object value = event.getValue();
            System.out.println(application + "范围内名为"
                  + name + ", 值为" + value + "的属性被删除了!");
      }
      //当 application 范围的属性被替换时触发该方法
      public void attributeReplaced(ServletContextAttributeEvent event)
      {
```

```
            ServletContext application = event.getServletContext();
            //获取被替换的属性名和属性值
            String name = event.getName();
            Object value = event.getValue();
            System.out.println(application + "范围内名为"
                  + name + ", 值为" + value + "的属性被替换了!");
      }
}
```

上面的 ServletContextAttributeListener 使用了@WebListener Annotation 修饰，这就是向 Web 应用中注册了该 Listener，该 Listener 实现了 attributeAdded、attributeRemoved、attributeReplaced 方法，因此当 application 范围内的属性被添加、删除、替换时，这些对应的监听器方法将会被触发。

2.13.4 使用 ServletRequestListener 和 ServletRequestAttributeListener

ServletRequestListener 用于监听用户请求的到达，实现该接口的监听器需要实现如下两个方法。

- requestInitialized(ServletRequestEvent sre)：用户请求到底、被初始化时触发该方法。
- requestDestroyed(ServletRequestEvent sre)：用户请求结束、被销毁时触发该方法。

ServletRequestAttributeListener 则用于监听 ServletRequest（request）范围内属性的变化，实现该接口的监听器需要实现 attributeAdded、attributeRemoved、attributeReplaced 三个方法。由此可见，ServletRequestAttributeListener 与 ServletContextAttributeListener 的作用相似，都用于监听属性的改变，只是 ServletRequestAttributeListener 监听 request 范围内属性的改变，而 ServletContextAttributeListener 监听的是 application 范围内属性的改变。

需要指出的是，应用程序完全可以采用一个监听器类来监听多种事件，只要让该监听器实现类同时实现多个监听器接口即可，如以下代码所示。

程序清单：codes\02\2.13\listenerTest\WEB-INF\src\lee\RequestListener.java

```
@WebListener
public class RequestListener
      implements ServletRequestListener , ServletRequestAttributeListener
{
      //当用户请求到达、被初始化时触发该方法
      public void requestInitialized(ServletRequestEvent sre)
      {
            HttpServletRequest request = (HttpServletRequest)sre.getServletRequest();
            System.out.println("----发向" + request.getRequestURI()
                  + "请求被初始化----");
      }
      //当用户请求结束、被销毁时触发该方法
      public void requestDestroyed(ServletRequestEvent sre)
      {
            HttpServletRequest request = (HttpServletRequest)sre.getServletRequest();
            System.out.println("----发向" + request.getRequestURI()
                  + "请求被销毁----");
      }
      //当程序向 request 范围添加属性时触发该方法
      public void attributeAdded(ServletRequestAttributeEvent event)
      {
            ServletRequest request = event.getServletRequest();
            //获取添加的属性名和属性值
            String name = event.getName();
            Object value = event.getValue();
            System.out.println(request + "范围内添加了名为"
                  + name + ", 值为" + value + "的属性!");
      }
      //当程序从 request 范围删除属性时触发该方法
      public void attributeRemoved(ServletRequestAttributeEvent event)
      {
            ServletRequest request = event.getServletRequest();
```

```
            //获取被删除的属性名和属性值
            String name = event.getName();
            Object value = event.getValue();
            System.out.println(request + "范围内名为"
                  + name + ", 值为" + value + "的属性被删除了!");
      }
      //当 request 范围的属性被替换时触发该方法
      public void attributeReplaced(ServletRequestAttributeEvent event)
      {
            ServletRequest request = event.getServletRequest();
            //获取被替换的属性名和属性值
            String name = event.getName();
            Object value = event.getValue();
            System.out.println(request + "范围内名为"
                  + name + ", 值为" + value + "的属性被替换了!");
      }
}
```

上面的监听器实现类同时实现了 ServletRequestListener 接口和 ServletRequestAttributerListener 接口，因此它既可以监听用户请求的初始化和销毁，也可监听 request 范围内属性的变化。

由于实现了 ServletRequestListener 接口的监听器可以非常方便地监听到每次请求的创建、销毁，因此 Web 应用可通过实现该接口的监听器来监听访问该应用的每个请求，从而实现系统日志。

➤➤2.13.5 使用 HttpSessionListener 和 HttpSessionAttributeListener

HttpSessionListener 用于监听用户 session 的创建和销毁，实现该接口的监听器需要实现如下两个方法。

- sessionCreated(HttpSessionEvent se)：用户与服务器的会话开始、创建时时触发该方法。
- sessionDestroyed(HttpSessionEvent se)：用户与服务器的会话断开、销毁时触发该方法。

HttpSessionAttributeListener 则用于监听 HttpSession（session）范围内属性的变化，实现该接口的监听器需要实现 attributeAdded、attributeRemoved、attributeReplaced 三个方法。由此可见，HttpSessionAttributeListener 与 ServletContextAttributeListener 的作用相似，都用于监听属性的改变，只是 HttpSessionAttributeListener 监听 session 范围内属性的改变，而 ServletContextAttributeListener 监听的是 application 范围内属性的改变。

实现 HttpSessionListener 接口的监听器可以监听每个用户会话的开始和断开，因此应用可以通过该监听器监听系统的在线用户。

下面是该监听器的实现类。

程序清单：codes\02\2.13\listenerTest\WEB-INF\src\lee\OnlineListener.java

```
@WebListener
public class OnlineListener
      implements HttpSessionListener
{
      //当用户与服务器之间开始 session 时触发该方法
      public void sessionCreated(HttpSessionEvent se)
      {
            HttpSession session = se.getSession();
            ServletContext application = session.getServletContext();
            //获取 session ID
            String sessionId = session.getId();
            //如果是一次新的会话
            if (session.isNew())
            {
                  String user = (String)session.getAttribute("user");
                  //未登录用户当游客处理
                  user = (user == null) ? "游客" : user;
```

```
                Map<String , String> online = (Map<String , String>)
                     application.getAttribute("online");
                if (online == null)
                {
                     online = new Hashtable<String , String>();
                }
                //将用户在线信息放入 Map 中
                online.put(sessionId , user);
                application.setAttribute("online" , online);
           }
      }
      //当用户与服务器之间 session 断开时触发该方法
      public void sessionDestroyed(HttpSessionEvent se)
      {
           HttpSession session = se.getSession();
           ServletContext application = session.getServletContext();
           String sessionId = session.getId();
           Map<String , String> online = (Map<String , String>)
                application.getAttribute("online");
           if (online != null)
           {
                //删除该用户的在线信息
                online.remove(sessionId);
           }
           application.setAttribute("online" , online);
      }
}
```

上面的监听器实现类实现了 HttpSessionListener 接口，该监听器可用于监听用户与服务器之间 session 的开始、关闭，当用户与服务器之间的 session 开始时，如果该 session 是一次新的 session，程序就将当前用户的 session ID、用户名存入 application 范围的 Map 中；当用户与服务器之间的 session 关闭时，程序从 application 范围的 Map 中删除该用户的信息。通过上面的方式，application 范围内的 Map 就记录了当前应用的所有在线用户。

显示在线用户的页面代码很简单，只要迭代输出 application 范围的 Map 即可，如以下代码所示。

程序清单：codes\02\2.13\listenerTest\online.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.util.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title> 用户在线信息 </title>
</head>
<body>
在线用户:
<table width="400" border="1">
<%
Map<String , String> online = (Map<String , String>)application
     .getAttribute("online");
for (String sessionId : online.keySet())
{%>
<tr>
     <td><%=sessionId%>
     <td><%=online.get(sessionId)%>
</tr>
<%}%>
</body>
</html>
```

笔者在本机启动三个不同的浏览器来模拟三个用户访问该应用，访问 online.jsp 页面将可看到如图

2.42 所示页面。

需要指出的是：采用 HttpSessionListener 监听用户在线信息比较“粗糙”，只能监听到有多少人在线，每个用户的 session ID 等基本信息。如果应用需要监听到每个用户停留在哪个页面、本次在线的停留时间、用户的访问 IP 等信息，则应该考虑定时检查 HttpServletRequest 来实现。

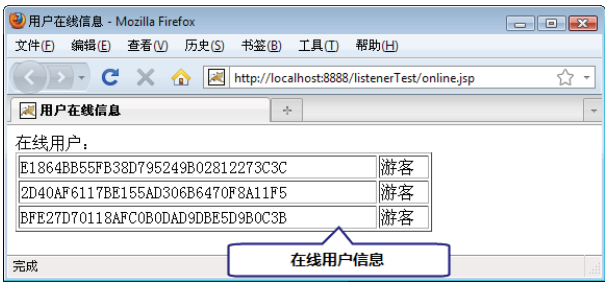

图 2.42　使用 HttpSessionListener 监听在线信息

通过检查 HttpServletRequest 的做法可以更精确地监控在线用户的状态，这种做法的思路是：

- 定义一个 ServletRequestListener，这个监听器负责监听每个用户请求，当用户请求到达时，系统将用户请求的 session ID、用户名、用户 IP、正在访问的资源、访问时间记录下来。
- 启动一条后台线程，这条后台线程每隔一段时间检查上面的每条在线记录，如果某条在线记录的访问时间与当前时间相差超过了指定值，将这条在线记录删除即可。这条后台线程应随着 Web 应用的启动而启动，可考虑使用 ServletContextListener 来完成。

下面先定义一个 ServletRequestListener，它负责监听每次用户请求：每次用户请求到达时，如果是新的用户会话，将相关信息插入数据表；如果是老的用户会话，则更新数据表中已有的在线记录。

程序清单：codes\02\2.13\ online\WEB-INF\src\lee\RequestListener.java

```
@WebListener
public class RequestListener
      implements ServletRequestListener
{
      //当用户请求到达、被初始化时触发该方法
      public void requestInitialized(ServletRequestEvent sre)
      {
            HttpServletRequest request = (HttpServletRequest)sre.getServletRequest();
            HttpSession session = request.getSession();
            //获取 session ID
            String sessionId = session.getId();
            //获取访问的 IP 和正在访问的页面
            String ip = request.getRemoteAddr();
            String page = request.getRequestURI();
            String user = (String)session.getAttribute("user");
            //未登录用户当游客处理
            user = (user == null) ? "游客" : user;
            try
            {
                  DbDao dd = new DbDao("com.mysql.jdbc.Driver"
                        , "jdbc:mysql://localhost:3306/online_inf"
                        , "root"
                        , "32147");
                  ResultSet rs = dd.query("select * from online_inf where session_id=?"
                        , true , sessionId);
                  //如果该用户对应的 session ID 存在，表明是旧的会话
                  if (rs.next())
                  {
                        //更新记录
                        rs.updateString(4, page);
                        rs.updateLong(5, System.currentTimeMillis());
                        rs.updateRow();
                        rs.close();
```

```
                }
                else
                {
                    //插入该用户的在线信息
                    dd.insert("insert into online_inf values(? , ? , ? , ? , ?)",
                        sessionId , user , ip , page , System.currentTimeMillis());
                }
            }
            catch (Exception ex)
            {
                ex.printStackTrace();
            }
        }
        //当用户请求结束、被销毁时触发该方法
        public void requestDestroyed(ServletRequestEvent sre)
        {
        }
}
```

上面的程序中粗体字代码控制用户会话是新的 session，还是已有的 session，新的 session 将插入数据表；旧的 session 将更新数据表中对应的记录。

接下来定义一个 ServletContextListener，它负责启动一条后台线程，这条后台线程将会定期检查在线记录，并删除那些长时间没有重新请求过的记录。该 Listerner 代码如下。

程序清单：codes\02\2.13\ online\WEB-INF\src\lee\OnlineListener.java

```
@WebListener
public class OnlineListener
    implements ServletContextListener
{
    //超过该时间（10 分钟）没有访问本站即认为用户已经离线
    public final int MAX_MILLIS = 10 * 60 * 1000;
    //应用启动时触发该方法
    public void contextInitialized(ServletContextEvent sce)
    {
        //每 5 秒检查一次
        new javax.swing.Timer(1000 * 5 , new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                try
                {
                    DbDao dd = new DbDao("com.mysql.jdbc.Driver"
                        , "jdbc:mysql://localhost:3306/online_inf"
                        , "root"
                        , "32147");
                    ResultSet rs = dd.query("select * from online_inf" , false);
                    StringBuffer beRemove = new StringBuffer("(");
                    while(rs.next())
                    {
                        //如果距离上次访问时间超过了指定时间
                        if ((System.currentTimeMillis() - rs.getLong(5))
                            > MAX_MILLIS)
                        {
                            //将需要被删除的 session ID 添加进来
                            beRemove.append("'");
                            beRemove.append(rs.getString(1));
                            beRemove.append("' , ");
                        }
                    }
                    //有需要删除的记录
                    if (beRemove.length() > 3)
                    {
                        beRemove.setLength(beRemove.length() - 3);
                        beRemove.append(")");
                        //删除所有“超过指定时间未重新请求的记录”
```

```
                                    dd.modify("delete from online_inf where session_id in "
                                          + beRemove.toString());
                              }
                              dd.closeConn();
                        }
                        catch (Exception ex)
                        {
                              ex.printStackTrace();
                        }
                  }
            }).start();
      }
      public void contextDestroyed(ServletContextEvent sce)
      {
      }
}
```

上面的程序中粗体字代码负责收集系统中“超过指定时间未访问”的在线记录，然后程序通过一条 SQL 语句删除这些在线记录。

需要指出的是：上面程序启动的后台线程定期检查的时间间隔为 5 秒，实际项目中这个时间应该适当加大，尤其是在线用户较多时，否则应用将会频繁地检查 online_inf 数据表中的全部记录，这将导致系统开销过大。

显示在线用户的页面十分简单，只要查询 online_inf 表中全部记录，并将这些记录显示出来即可。以下是该页面代码。

程序清单：codes\02\2.13\ online\online.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.sql.*,lee.*" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 用户在线信息 </title>
</head>
<body>
在线用户:
<table width="640" border="1">
<%
DbDao dd = new DbDao("com.mysql.jdbc.Driver"
      , "jdbc:mysql://localhost:3306/online_inf"
      , "root"
      , "32147");
//查询 online_inf 表（在线用户表）的全部记录
ResultSet rs = dd.query("select * from online_inf" , false);
while (rs.next())
{%>
<tr>
      <td><%=rs.getString(1)%>
      <td><%=rs.getString(2)%>
      <td><%=rs.getString(3)%>
      <td><%=rs.getString(4)%>
</tr>
<%}%>
</body>
</html>
```

启动不同浏览器访问该应用的不同页面，然后访问 online.jsp 页面将可看到如图 2.43 所示页面。

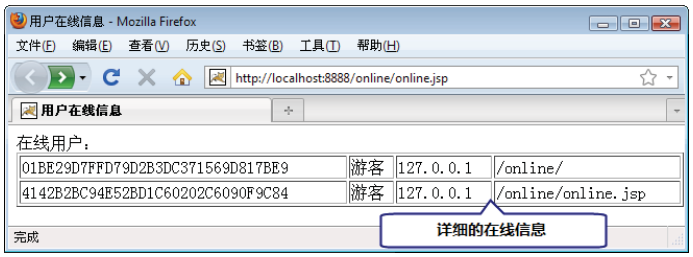

图 2.43　详细的在线信息

对于应用中所有需要统计在线用户页面，只要将上面的 online.jsp 页面包含到页面中即可。

本应用需要使用数据表来保存在线用户信息，因此读者应该先将 codes\02\2.13\online\WEB-INF 目录下的数据表脚本导入数据库。

2.14　JSP 2 特性

2003 年发布的 JSP 2.0 升级了 JSP 1.2 规范，新增了一些额外的特性。JSP 2.0 使得动态网页的设计更加容易，甚至可以无须学习 Java，也可做出 JSP 页面，从而可以更好地支持团队开发。目前 Servlet 3.0 对应于 JSP 2.2 规范，不过 JSP 2.2 与 JSP 2.0 相差并不大，我们将其统称为 JSP 2。

相比 JSP 1.2，JSP 2 主要增加了如下新特性。

- 直接配置 JSP 属性。
- 表达式语言。
- 简化的自定义标签 API。
- Tag 文件语法。

如果需要使用 JSP 2 语法，其 web.xml 文件必须使用 Servlet 2.4 以上版本的配置文件。Servlet 2.4 以上版本的配置文件的根元素写法如下：

```
<?xml version="1.0" encoding="GBK"?>
<!--  不再使用 DTD，而是使用 Schema 描述，版本也升级为 2.4-->
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee http://java.sun.
com/xml/ns/j2ee/web-app_2_4.xsd" version="2.4">
<!-- 此处是 Web 应用的其他配置 -->
...
</web-app>
```

提示：本书所给出的 Web 应用都是使用 Servlet 3.0 规范，也就是对应于 JSP 2.2 规范，因此完全支持 JSP 2 的特性。

2.14.1　配置 JSP 属性

JSP 属性定义使用<jsp-property-group/>元素配置，主要包括如下 4 个方面。

- 是否允许使用表达式语言：使用<el-ignored/>元素确定，默认值为 false，即允许使用表达式语言。
- 是否允许使用 JSP 脚本：使用<scripting-invalid/>元素确定，默认值为 false，即允许使用 JSP

脚本。

- 声明 JSP 页面的编码：使用<page-encoding/>元素确定，配置该元素后，可以代替每个页面里 page 指令 contentType 属性的 charset 部分。
- 使用隐式包含：使用<include-prelude/>和<include-coda/>元素确定，可以代替在每个页面里使用 include 编译指令来包含其他页面。

提示：
此处隐式包含的作用与 JSP 提供的静态包含的作用相似。

下面的 web.xml 文件配置了该应用下的系列属性。

程序清单：codes\02\2.14\jsp2\WEB-INF\web.xml

```
<?xml version="1.0" encoding="GBK"?>
<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">
      <!-- 关于 JSP 的配置信息 -->
      <jsp-config>
            <jsp-property-group>
                  <!-- 对哪些文件应用配置 -->
                  <url-pattern>/noscript/*</url-pattern>
                  <!-- 忽略表达式语言 -->
                  <el-ignored>true</el-ignored>
                  <!-- 页面编码的字符集 -->
                  <page-encoding>GBK</page-encoding>
                  <!-- 不允许使用 Java 脚本 -->
                  <scripting-invalid>true</scripting-invalid>
                  <!-- 隐式导入页面头  -->
                  <include-prelude>/inc/top.jspf</include-prelude>
                  <!-- 隐式导入页面尾 -->
                  <include-coda>/inc/bottom.jspf</include-coda>
            </jsp-property-group>
            <jsp-property-group>
                  <!-- 对哪些文件应用配置 -->
                  <url-pattern>*.jsp</url-pattern>
                  <el-ignored>false</el-ignored>
                  <!-- 页面编码字符集 -->
                  <page-encoding>GBK</page-encoding>
                  <!-- 允许使用 Java 脚本 -->
                  <scripting-invalid>false</scripting-invalid>
            </jsp-property-group>
            <jsp-property-group>
                  <!-- 对哪些文件应用配置 -->
                  <url-pattern>/inc/*</url-pattern>
                  <el-ignored>false</el-ignored>
                  <!-- 页面编码字符集 -->
                  <page-encoding>GBK</page-encoding>
                  <!-- 不允许使用 Java 脚本 -->
                  <scripting-invalid>true</scripting-invalid>
            </jsp-property-group>
      </jsp-config>
</web-app>
```

上面的配置文件中配置了三个 jsp-property-group 元素，每个元素配置一组 JSP 属性，用于指定哪些 JSP 页面应该满足怎样的规则。例如，第一个 jsp-property-group 元素指定：/noscript/下的所有页面应该使用 GBK 字符集进行编码，且不允许使用 JSP 脚本，忽略表达式语言，并隐式包含页面头、页面尾。

如果在不允许使用 JSP 脚本的页面中使用 JSP 脚本，则该页面将出现错误。即/noscript/下的页面中使用 JSP 脚本将引起错误。

看下面的 JSP 页面代码，为 test1.jsp 页面代码。

程序清单：codes\02\2.14\jsp2\noscript\test1.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 页面配置 1 </title>
</head>
<body>
<h2>页面配置 1</h2>
下面是表达式语言输出：<br/>
${1 + 2}
</body>
</html>
```

上面的页面中粗体字代码就是表达式语言，关于表达式语言请看下一节介绍。但由于我们在 web.xml 文件中配置了表达式语言无效，所以浏览该页面将看到系统直接输出表达式语言。在浏览器中浏览该页面的效果如图 2.44 所示。

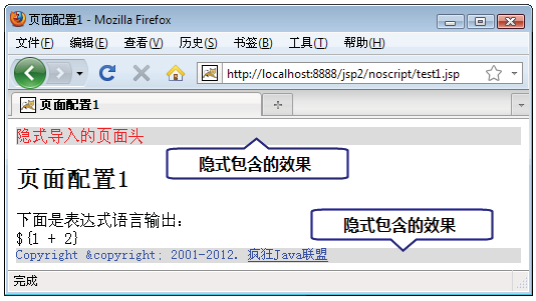

图 2.44　页面配置的运行效果

从图 2.44 中可以看出，test1.jsp 的表达式语言不能正常输出，这是因为我们配置了忽略表达式语言。上面页面中看到隐式 include 的页面头分别是 top.jspf 和 bottom.jspf，这两个文件依然是 JSP 页面，只是将文件名后缀改为了 jspf 而已。

而位于应用根路径下的 JSP 页面则支持表达式语言和 JSP 脚本，但没有使用隐式 include 包含页面头和页面尾。应用根路径下的 test2.jsp 页面代码如下。

程序清单：codes\02\2.14\jsp2\test2.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 页面配置 2 </title>
</head>
<body>
      <h2>页面配置 2</h2>
      下面是表达式语言输出：<br/>
      ${1 + 2}<br/>
      下面是小脚本输出：<br/>
      <%out.println("hello Java");%>
</body>
</html>
```

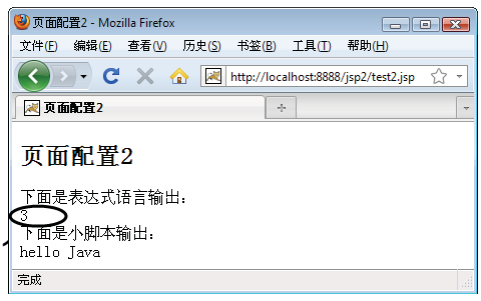

图 2.45　使用表达式语言和 JSP 脚本

上面的页面中两行粗体字代码正是嵌套在 JSP 页面中的

JSP 脚本和表达式语言，浏览该页面将看到如图 2.45 所示的效果。

图 2.45 中椭圆形圈出的 3 就是${1+2}的结果——这就是表达式语言的计算结果。

➤➤2.14.2 表达式语言

表达式语言（Expression Language）是一种简化的数据访问方式。使用表达式语言可以方便地访问 JSP 的隐含对象和 JavaBeans 组件，在 JSP 2 规范中，建议尽量使用表达式语言使 JSP 文件的格式一致，避免使用 Java 脚本。

表达式语言可用于简化 JSP 页面的开发，允许美工设计人员使用表达式语言的语法获取业务逻辑组件传过来的变量值。

提示：

表达式语言是 JSP 2 的一个重要特性，它并不是一种通用的程序语言，而仅仅是一种数据访问语言，可以方便地访问应用程序数据，避免使用 JSP 脚本。

表达式语言的语法格式是：

```
${expression}
```

1．表达式语言支持的算术运算符和逻辑运算符

表达式语言支持的算术运算符和逻辑运算符非常多，所有在 Java 语言里支持的算术运算符，表达式语言都可以使用；甚至 Java 语言不支持的一些算术运算符和逻辑运算符，表达式语言也支持。

下面的 JSP 页面示范了在表达式语言中使用算术运算符。

程序清单：codes\02\2.14\jsp2\arithmeticOperator.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 表达式语言 - 算术运算符 </title>
</head>
<body>
      <h2>表达式语言 - 算术运算符</h2><hr/>
      <table border="1" bgcolor="#aaaadd">
            <tr>
                  <td><b>表达式语言</b></td>
                  <td><b>计算结果</b></td>
            </tr>
            <!-- 直接输出常量 -->
            <tr>
                  <td>\${1}</td>
                  <td>${1}</td>
            </tr>
            <!-- 计算加法 -->
            <tr>
                  <td>\${1.2 + 2.3}</td>
                  <td>${1.2 + 2.3}</td>
            </tr>
            <!-- 计算加法 -->
            <tr>
                  <td>\${1.2E4 + 1.4}</td>
                  <td>${1.2E4 + 1.4}</td>
            </tr>
            <!-- 计算减法 -->
            <tr>
                  <td>\${-4 - 2}</td>
                  <td>${-4 - 2}</td>
```

```
            </tr>
            <!-- 计算乘法 -->
            <tr>
                <td>\${21 * 2}</td>
                <td>${21 * 2}</td>
            </tr>
            <!-- 计算除法 -->
            <tr>
                <td>\${3/4}</td>
                <td>${3/4}</td>
            </tr>
            <!-- 计算除法 -->
            <tr>
                <td>\${3 div 4}</td>
                <td>${3 div 4}</td>
            </tr>
            <!-- 计算除法 -->
            <tr>
                <td>\${3/0}</td>
                <td>${3/0}</td>
            </tr>
            <!-- 计算求余 -->
            <tr>
                <td>\${10%4}</td>
                <td>${10%4}</td>
            </tr>
            <!-- 计算求余 -->
            <tr>
                <td>\${10 mod 4}</td>
                <td>${10 mod 4}</td>
            </tr>
            <!-- 计算三目运算符 -->
            <tr>
                <td>\${(1==2) ? 3 : 4}</td>
                <td>${(1==2) ? 3 : 4}</td>
            </tr>
        </table>
    </body>
    </html>
```

上面的页面中示范了表达式语言所支持的加、减、乘、除、求余等算术运算符的功能，读者可能也发现了表达式语言还支持 div、mod 等运算符。而且表达式语言把所有数值都当成浮点数处理，所以 3/0 的实质是 3.0/0.0，得到结果应该是 Infinity。

浏览 arithmeticOperator.jsp 页面，将看到如图 2.46 所示的效果。

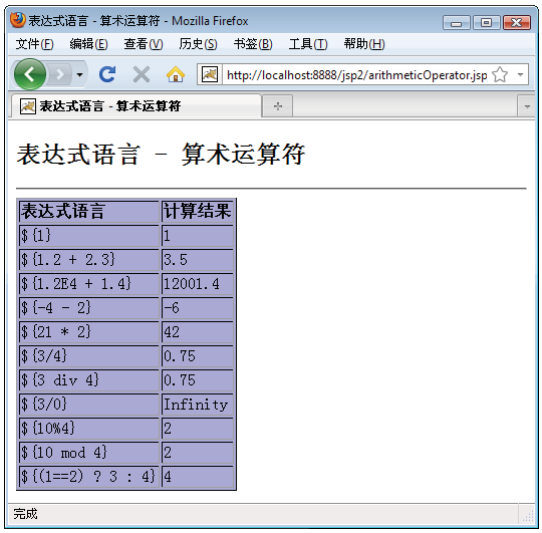

图 2.46　支持算术运算符的表达式语言

如果需要在支持表达式语言的页面中正常输出“$”符号，则在“$”符号前加转义字符“\”，否则系统以为“$”是表达式语言的特殊标记。

也可以在表达式语言中使用逻辑运算符，如下面的 JSP 页面所示。

程序清单：codes\02\2.14\jsp2\logicOperator.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 表达式语言 - 逻辑运算符 </title>
</head>
<body>
      <h2>表达式语言 - 逻辑运算符</h2><hr/>
      数字之间的比较：
      <table border="1" bgcolor="#aaaadd">
            <tr>
                  <td><b>表达式语言</b></td>
                  <td><b>计算结果</b></td>
            </tr>
            <!-- 直接比较两个数字 -->
            <tr>
                  <td>\${1 &lt; 2}</td>
                  <td>${1 < 2}</td>
            </tr>
            <!-- 使用 lt 比较运算符 -->
            <tr>
                  <td>\${1 lt 2}</td>
                  <td>${1 lt 2}</td>
            </tr>
            <!-- 使用>比较运算符 -->
            <tr>
                  <td>\${1 &gt; (4/2)}</td>
                  <td>${1 > (4/2)}</td>
            </tr>
            <!-- 使用 gt 比较运算符 -->
            <tr>
                  <td>\${1 gt (4/2)}</td>
                  <td>${1 gt (4/2)}</td>
            </tr>
            <!-- 使用>=比较运算符 -->
            <tr>
                  <td>\${4.0 &gt= 3}</td>
```

```
                <td>${4.0 >= 3}</td>
            </tr>
            <!-- 使用 ge 比较运算符 -->
            <tr>
                <td>\${4.0 ge 3}</td>
                <td>${4.0 ge 3}</td>
            </tr>
            <!-- 使用<=比较运算符 -->
            <tr>
                <td>\${4 &lt;= 3}</td>
                <td>${4 <= 3}</td>
            </tr>
            <!-- 使用 le 比较运算符 -->
            <tr>
                <td>\${4 le 3}</td>
                <td>${4 le 3}</td>
            </tr>
            <!-- 使用==比较运算符 -->
            <tr>
                <td>\${100.0 == 100}</td>
                <td>${100.0 == 100}</td>
            </tr>
            <!-- 使用 eq 比较运算符 -->
            <tr>
                <td>\${100.0 eq 100}</td>
                <td>${100.0 eq 100}</td>
            </tr>
            <!-- 使用!=比较运算符 -->
            <tr>
                <td>\${(10*10) != 100}</td>
                <td>${(10*10) != 100}</td>
            </tr>
            <!--  先执行运算，再进行比较运算，使用 ne 比较运算符-->
            <tr>
                <td>\${(10*10) ne 100}</td>
                <td>${(10*10) ne 100}</td>
            </tr>
        </table>
        字符之间的比较：
        <table border="1" bgcolor="#aaaadd">
            <tr>
                <td><b>表达式语言</b></td>
                <td><b>计算结果</b></td>
            </tr>
            <tr>
                <td>\${'a' &lt; 'b'}</td>
                <td>${'a' < 'b'}</td>
            </tr>
            <tr>
                <td>\${'hip' &gt; 'hit'}</td>
                <td>${'hip' > 'hit'}</td>
            </tr>
            <tr>
                <td>\${'4' &gt; 3}</td>
                <td>${'4' > 3}</td>
            </tr>
        </table>
    </body>
    </html>
```

从上面程序的粗体字代码中可以看出：表达式语言不仅可在数字与数字之间比较，还可在字符与字符之间比较，字符串的比较是根据其对应 UNICODE 值来比较大小的。

2．表达式语言的内置对象

使用表达式语言可以直接获取请求参数值，可以获取页面中 JavaBean 的指定属性值，获取请求头

及获取 page、request、session 和 application 范围的属性值等，这些都得益于表达式语言的内置对象。

表达式语言包含如下 11 个内置对象。

- pageContext：代表该页面的 pageContext 对象，与 JSP 的 pageContext 内置对象相同。
- pageScope：用于获取 page 范围的属性值。
- requestScope：用于获取 request 范围的属性值。
- sessionScope：用于获取 session 范围的属性值。
- applicationScope：用于获取 application 范围的属性值。
- param：用于获取请求的参数值。
- paramValues：用于获取请求的参数值，与 param 的区别在于，该对象用于获取属性值为数组的属性值。
- header：用于获取请求头的属性值。
- headerValues：用于获取请求头的属性值，与 header 的区别在于，该对象用于获取属性值为数组的属性值。
- initParam：用于获取请求 Web 应用的初始化参数。
- cookie：用于获取指定的 Cookie 值。

下面的 JSP 页面示范了如何使用表达式语言的内置对象的方法。

程序清单：codes\02\2.14\jsp2\implicit-objects.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 表达式语言 - 内置对象 </title>
</head>
<body>
      <h2>表达式语言 - 内置对象</h2>
      请输入你的名字：
      <!-- 通过表单提交请求参数 -->
      <form action="implicit-objects.jsp" method="post">
            <!-- 通过${param['name']} 获取请求参数 -->
            你的名字 = <input type="text" name="name" value="${param['name']}"/>
            <input type="submit" value='提交'/>
      </form><br/>
      <% session.setAttribute("user" , "abc");
      //下面三行代码添加 Cookie
      Cookie c = new Cookie("name" , "yeeku");
      c.setMaxAge(24 * 3600);
      response.addCookie(c);
      %>
      <table border="1" width="660" bgcolor="#aaaadd">
            <tr>
                  <td width="170"><b>功能</b></td>
                  <td width="200"><b>表达式语言</b></td>
                  <td width="300"><b>计算结果</b></td>
            <tr>
                  <!-- 使用两种方式获取请求参数值 -->
                  <td>取得请求参数值</td>
                  <td>\${param.name}</td>
                  <td>${param.name} </td>
            </tr>
            <tr>
                  <td>取得请求参数值</td>
                  <td>\${param["name"]}</td>
                  <td>${param["name"]} </td>
            </tr>
            <tr>
                  <!-- 使用两种方式获取指定请求头信息 -->
```

```
                <td>取得请求头的值</td>
                <td>\${header.host}</td>
                <td>${header.host}</td>
            </tr>
            <tr>
                <td>取得请求头的值</td>
                <td>\${header["accept"]}</td>
                <td>${header["accept"]}</td>
            </tr>
            <!-- 获取 Web 应用的初始化参数值 -->
            <tr>
                <td>取得初始化参数值</td>
                <td>\${initParam["author"]}</td>
                <td>${initParam["author"]}</td>
            </tr>
            <!-- 获取 session 返回的属性值 -->
            <tr>
                <td>取得 session 的属性值</td>
                <td>\${sessionScope["user"]}</td>
                <td>${sessionScope["user"]}</td>
            </tr>
            <!-- 获取指定 Cookie 的值 -->
            <tr>
                <td>取得指定 Cookie 的值</td>
                <td>\${cookie["name"].value}</td>
                <td>${cookie["name"].value}</td>
            </tr>
        </table>
    </body>
    </html>
```

上面的页面中粗体字代码就是使用表达式语言内置对象的关键代码。浏览上面页面，并通过页面中表单来提交请求，将看到如图 2.47 所示的效果。

3．表达式语言的自定义函数

表达式语言除了可以使用基本的运算符外，还可以使用自定义函数。通过自定义函数，能够大大加强表达式语言的功能。自定义函数的开发步骤非常类似于标签的开发步骤，定义方式也几乎一样。区别在于自定义标签直接在页面上生成输出，而自定义函数则需要在表达式语言中使用。

图 2.47　表达式语言中的内置对象

提示：

函数功能大大扩充了 EL 的功能，EL 本身只是一种数据访问语言，因此它不支持调用方法。如果需要在 EL 中进行更复杂的处理，就可以通过函数来完成。函数的本质是：提

供一种语法允许在 EL 中调用某个类的静态方法。

下面介绍表达式语言中自定义函数的开发步骤。

① 开发函数处理类：函数处理类就是普通类，这个普通类中包含若干个静态方法，每个静态方法都可定义成一个函数。实际上这个步骤也是可省略的——完全可以直接使用 JDK 或其他项目提供的类，只要这个类包含静态方法即可。

程序清单：codes\02\2.14\jsp2\WEB-INF\src\lee\Functions.java

```
public class Functions
{
      //对字符串进行反转
      public static String reverse( String text )
      {
            return new StringBuffer( text ).reverse().toString();
      }
      //统计字符串的个数
      public static int countChar( String text )
      {
            return text.length();
      }
}
```

完全可以直接使用 JDK 或其他项目提供的类作为函数处理类 ,只要这个类包含静态方法即可。

② 使用标签库定义函数：定义函数的方法与定义标签的方法大致相似。在<taglib.../>元素下增加<tag.../>元素用于定义自定义标签；增加<function.../>元素则用于定义自定义函数。每个<function.../>元素只要三个子元素即可。

- name：指定自定义函数的函数名。
- function-class：指定自定义函数的处理类。
- function-signature：指定自定义函数对应的方法。

下面的标签库定义（TLD）文件将上面的 Functions.java 类中所包含的两个方法定义成两个函数。

程序清单：codes\02\2.14\jsp2\WEB-INF\src\mytaglib.tld

```
<?xml version="1.0" encoding="GBK"?>
<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_0.xsd" version="2.0">
      <tlib-version>1.0</tlib-version>
      <short-name>crazyit</short-name>
      <!-- 定义该标签库的 URI -->
      <uri>http://www.crazyit.org/tags</uri>
      <!-- 定义第一个函数 -->
      <function>
            <!-- 定义函数名:reverse -->
            <name>reverse</name>
            <!-- 定义函数的处理类 -->
            <function-class>lee.Functions</function-class>
            <!-- 定义函数的实现方法-->
            <function-signature>
                  java.lang.String reverse(java.lang.String)</function-signature>
      </function>
      <!-- 定义第二个函数: countChar -->
      <function>
            <!-- 定义函数名:countChar -->
```

```
            <name>countChar</name>
            <!-- 定义函数的处理类 -->
            <function-class>lee.Functions</function-class>
            <!-- 定义函数的实现方法-->
            <function-signature>int countChar(java.lang.String)
                </function-signature>
        </function>
    </taglib>
```

上面的粗体字代码定义了两个函数，不难发现其实定义函数比定义自定义标签更简单，因为自定义函数只需配置三个子元素即可，变化更少。

③ 在 JSP 页面的 EL 中使用函数：一样需要先导入标签库，然后再使用函数。下面是使用函数的 JSP 页面代码。

程序清单：codes\02\2.14\jsp2\useFunctions.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ taglib prefix="crazyit" uri="http://www.crazyit.org/tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title> new document </title>
</head>
<body>
    <h2>表达式语言 - 自定义函数</h2><hr/>
    请输入一个字符串：
    <form action="useFunctions.jsp" method="post">
        字符串 = <input type="text" name="name" value="${param['name']}">
        <input type="submit"  value="提交">
    </form>
    <table border="1" bgcolor="aaaadd">
        <tr>
        <td><b>表达式语言</b></td>
        <td><b>计算结果</b></td>
        <tr>
        <tr>
            <td>\${param["name"]}</td>
            <td>${param["name"]} </td>
        </tr>
        <!--  使用 reverse 函数-->
        <tr>
            <td>\${crazyit:reverse(param["name"])}</td>
            <td>${crazyit:reverse(param["name"])} </td>
        </tr>
        <tr>
            <td>\${crazyit:reverse(crazyit:reverse(param["name"]))}</td>
            <td>${crazyit:reverse(crazyit:reverse(param["name"]))} </td>
        </tr>
        <!-- 使用 countChar 函数 -->
        <tr>
            <td>\${crazyit:countChar(param["name"])}</td>
            <td>${crazyit:countChar(param["name"])} </td>
        </tr>
    </table>
</body>
</html>
```

如上面程序中粗体字代码所示，导入标签库定义文件后（实质上也是函数库定义文件），就可以在表达式语言中使用函数定义库文件里定义的各函数了。

通过上面的介绍不难发现自定义函数的实质：就是将指定 Java 类的静态方法暴露成可以在 EL 中使用的函数，所以可以定义成函数的方法必须用 public static 修饰。

2.14.3 Tag File 支持

Tag File 是自定义标签的简化用法，使用 Tag File 可以无须定义标签处理类和标签库文件，但仍然可以在 JSP 页面中使用自定义标签。

下面以 Tag File 建立一个迭代器标签，其步骤如下。

① 建立 Tag 文件，在 JSP 所支持 Tag File 规范下，Tag File 代理了标签处理类，它的格式类似于 JSP 文件。可以这样理解：如同 JSP 可以代替 Servlet 作为表现层一样，Tag File 则可以代替标签处理类。

Tag File 具有以下 5 个编译指令。

- taglib：作用与 JSP 文件中的 taglib 指令效果相同，用于导入其他标签库。
- include：作用与 JSP 文件中的 include 指令效果相同，用于导入其他 JSP 或静态页面。
- tag：作用类似于 JSP 文件中的 page 指令，有 pageEncoding、body-content 等属性，用于设置页面编码等属性。
- attribute：用于设置自定义标签的属性，类似于自定义标签处理类中的标签属性。
- variable：用于设置自定义标签的变量，这些变量将传给 JSP 页面使用。

下面是迭代器标签的 Tag File，这个 Tag File 的语法与 JSP 语法非常相似。

程序清单：codes\02\2.14\jsp2\WEB-INF\tags\iterator.tag

```
<%@ tag pageEncoding="GBK" import="java.util.List"%>
<!-- 定义了 4 个标签属性 -->
<%@ attribute name="bgColor" %>
<%@ attribute name="cellColor" %>
<%@ attribute name="title" %>
<%@ attribute name="bean" %>
<table border="1" bgcolor="${bgColor}">
<tr>
<td><b>${title}</b></td>
</tr>
<!-- 取出 request 范围的 a 集合 -->
<%List<String> list = (List<String>)
request.getAttribute("a");
//遍历输出 list 集合的元素
for (Object ele : list){%>
<tr>
<td bgcolor="${cellColor}">
<%=ele%>
</td>
</tr>
<%}%>
</table>
```

上面的页面代码中的粗体字代码就是 Tag File 的核心代码，可能细心的读者会发现上面的 Tag File 并不会输出完整的 HTML 页面，它只包含一个 table 元素，即只有一个表格，这是正确的。回忆自定义标签的作用：通过简单的标签在页面上生成一个内容片段。同理，这个 Tag File 也只负责生成一个页面片段，所以它并不需要输出完整的 HTML 页面。

Tag File 的命名必须遵守如下规则：tagName.tag。即 Tag File 的主文件名就是标签名，文件名后缀必须是 tag。将该文件存在 Web 应用的某个路径下，这个路径相当于标签库的 URI 名。笔者将其放在/WEB-INF/tags 下，即笔者的标签库路径为/WEB-INF/tags。

② 在页面中使用自定义标签时，需要先导入标签库，再使用标签。使用 Tag File 标签与普通自定义标签的用法完全相同，只是在导入标签库时存在一些差异。由于此时的标签库没有 URI，只有标签库路径。因此导入标签时，使用如下语法格式：

```
<%@ taglib prefix="tagPrefix" tagdir="path" %>
```

其中，prefix 与之前的 taglib 指令的 prefix 属性完全相同，用于确定标签前缀；而 tagdir 标签库路

径下存放很多 Tag File，每个 Tag File 对应一个标签。

下面是使用 Tag File 标签的 JSP 页面代码。

程序清单：codes\02\2.14\jsp2\useTagFile.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<%@ page import="java.util.*" %>
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
     <title>迭代器 tag file</title>
</head>
<body>
     <h2>迭代器 tag file</h2>
     <%
     //创建集合对象，用于测试 Tag File 所定义的标签
     List<String> a = new ArrayList<String>();
     a.add("hello");
     a.add("world");
     a.add("java");
     //将集合对象放入页面范围
     request.setAttribute("a" , a);%>
     //使用自定义标签
     <tags:iterator bgColor="#99dd99" cellColor="#9999cc"
          title="迭代器标签" bean="a" />
</body>
</html>
```

从上面的粗体字代码可以看出，在 JSP 页面中使用 Tag File 标签也很简单。在该 JSP 页面中，使用了如下代码导入标签：

```
<%@ taglib prefix="tags" tagdir="/WEB-INF/tags" %>
```

即以 tags 开头的标签，使用/WEB-INF/tags 路径下的标签文件处理。在 JSP 页面中则使用如下代码来使用标签：

```
<tags:iterator bgColor="#99dd99" cellColor="#9999cc"
     title="迭代器标签" bean="a" />
```

tags 表明该标签使用/WEB-INF/tags 路径下的 Tag File 来处理标签；而 iterator 是标签名，即使用 WEB-INF/tags 路径下的 iterator.tag 文件负责处理该标签。useTagFile 页面最终的执行效果如图 2.48 所示。

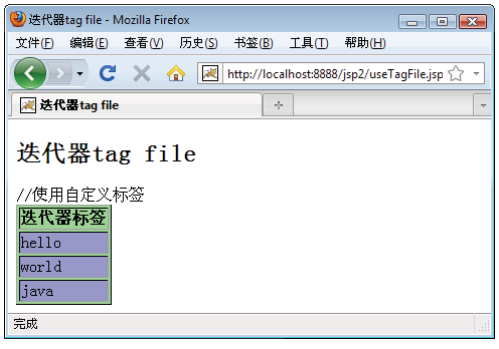

图 2.48　使用 Tag File 的迭代器标签

Tag File 是自定义标签的简化。事实上，就如同 JSP 文件会编译成 Servlet 一样，Tag File 也会编译成标签处理类，自定义标签的后台依然由标签处理类完成，而这个过程由容器完成。打开 Tomcat 的 work\Catalina\localhost\jsp2\org\apache\jsp\tag\web 路径，即可看到 iterator_tag.java、iterator_tag.class 两个文件，这两个文件就是 Tag File 所对应的标签处理类。

通过查看 iterator_tag.java 文件的内容不难发现，Tag File 中只有如下几个内置对象。

- request：与 JSP 脚本中的 request 对象对应。
- response：与 JSP 脚本中的 response 对象对应。
- session：与 JSP 脚本中的 session 对象对应。
- application：与 JSP 脚本中的 application 对象对应。
- config：与 JSP 脚本中的 config 对象对应。
- out：与 JSP 脚本中的 out 对象对应。

2.15 Servlet 3.0 新特性

伴随 Java EE6 一起发布的 Servlet 3.0 规范是 Servlet 规范历史上最重要的变革之一，它的许多特性都极大地简化了 Java Web 应用的开发，例如前面介绍开发 Servlet、Listener、Filter 时所使用的 Annotation。这些变革必将带给广大 Java 开发人员巨大的便利，大大加快 Java web 应用的开发效率。

➤➤2.15.1 Servlet 3.0 的 Annotation

Servlet 3.0 的一个显著改变是“顺应”了潮流，抛弃了采用 web.xml 配置 Servlet、Filter、Listener 的烦琐步骤，允许开发人员使用 Annotation 修饰它们，从而进行部署。

Servlet 3.0 规范在 javax.servlet.annotation 包下提供了如下 Annotation。

- @WebServlet：用于修饰一个 Servlet 类，用于部署 Servlet 类。
- @WebInitParam：用于与@WebServlet 或@WebFilter 一起使用，为 Servlet、Filter 配置参数。
- @WebListener：用于修饰 Listener 类，用于部署 Listener 类。
- @WebFilter：用于修饰 Filter 类，用于部署 Filter 类。
- @MultipartConfig：用于修饰 Servlet，指定该 Servlet 将会负责处理 multipart/form-data 类型的请求（主要用于文件上传）。
- @ServletSecurity：这是一个与 JAAS 有关的 Annotation，修饰 Servlet 指定该 Servlet 的安全与授权控制。
- @HttpConstraint：用于与@ServletSecurity 一起使用，用于指定该 Servlet 的安全与授权控制。
- @HttpMethodConstraint：用于与@ServletSecurity 一起使用，用于指定该 Servlet 的安全与授权控制。

上面这些 Annotation 有一些已经在前面有了详细的介绍，此处不再赘述。@MultipartConfig 的用法将会在 2.15.4 节有更详细的说明。至于上面三个与 JAAS 相关的 Annotation，由于本书并没有涉及 JAAS 方面的内容，因此请参考本书姊妹篇《经典 Java EE 企业应用实战》的相关章节。

➤➤2.15.2 Servlet 3.0 的 Web 模块支持

Servlet 3.0 为模块化开发提供了良好的支持，Servlet 3.0 规范不再要求所有 Web 组件（如 Servlet、Listener、Filter 等）都部署在 web.xml 文件中，而是允许采用“Web 模块”来部署、管理它们。

一个 Web 模块通常对应于一个 JAR 包，这个 JAR 包有如下文件结构：

<webModule>.jar——这是 Web 模块的 JAR 包，可以改变
|－META-INF
|　　|－web-fragment.xml
|－Web 模块所用的类文件、资源文件等。

从上面的文件结构可以看出，Web 模块与普通 JAR 的最大区别在于需要在 META-INF 目录下添加一个 web-fragment.xml 文件，这个文件也被称为 Web 模块部署描述符。

web-fragment.xml 文件与 web.xml 文件的作用、文档结构都基本相似，因为它们都用于部署、管理各种 Web 组件。只是 web-fragment.xml 用于部署、管理 Web 模块而已，但 web-fragment.xml 文件可以多指定如下两个元素。

- <name.../>：用于指定该 Web 模块的名称。
- <ordering.../>：用于指定加载该 Web 模块的相对顺序。

上面<ordering.../>元素用于指定加载当前 Web 模块的相对顺序，该元素的内部结构如图 2.49 所示。

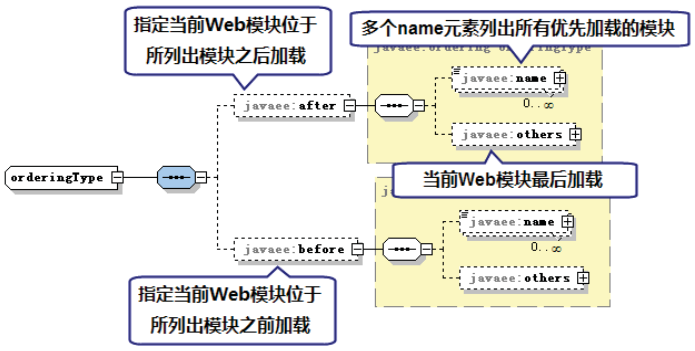

图 2.49　ordering 元素的内部结构

下面我们开发第一个 Web 模块，该 Web 模块内只定义了一个简单的 ServletContextListner，该 Web 模块对应的 web-fragment.xml 文件如下。

程序清单：codes\02\2.15\crazyit\src\META-INF\web-fragment.xml

```
<?xml version="1.0" encoding="GBK"?>
<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">
      <!-- 指定该 Web 模块的唯一标识 -->
      <name>crazyit</name>
      <listener>
            <listener-class>lee.CrazyitListener</listener-class>
      </listener>
      <ordering>
            <!-- 用于配置该 Web 模块必须位于哪些模块之前加载 -->
            <before>
                  <!-- 用于指定位于其他所有模块之前加载 -->
                  <others/>
            </before>
      </ordering>
</web-fragment>
```

上面的 Web 模块部署描述文件的根元素是 web-fragment，粗体字代码指定该 Web 模块的名称是 crazyiit，接下来的粗体字代码指定该 Web 模块将在其他所有 Web 模块之前加载。

接下来再开发一个 Web 模块，接下来的 Web 模块同样只定义了一个 ServletContextListener，该 Web 模块对应的 web-fragment.xml 文件如下。

程序清单：codes\02\2.15\leegang\src\META-INF\web-fragment.xml

```
<?xml version="1.0" encoding="GBK"?>
<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">
      <!-- 指定该 Web 模块的唯一标识 -->
      <name>leegang</name>
```

```
        <!-- 配置 Listener -->
        <listener>
            <listener-class>lee.LeegangListener</listener-class>
        </listener>
        <ordering>
            <!-- 用于配置该 Web 模块必须位于哪些模块之后加载 -->
            <after>
                <!-- 此处可用多个 name 元素列出
                     该模块必须位于这些模块之后加载 -->
                <name>crazyit</name>
            </after>
        </ordering>
    </web-fragment>
```

将这两个 Web 模块打包成 JAR 包，Web 模块 JAR 包的内部结构如图 2.50 所示。

将这两个 Web 模块对应的 JAR 包复制到任意 Web 应用的 WEB-INF/lib 目录下，启动 Web 应用，将可以看到两个 Web 模块被加载：先加载 crazyit 模块，再加载 leegang 模块。

Web 应用除了可按 web-fragment.xml 文件中指定的加载顺序来加载 Web 模块之外，还可以通过 web.xml 文件指定各 Web 模块加载的绝对顺序。在 web.xml 文件中指定的加载顺序将会覆盖 Web 模块中 web-fragment.xml 文件所指定的加载顺序。

图 2.50　Web 模块的内部结构

例如我们在 Web 应用的 web.xml 文件中增加如下配置片段：

```
    <absolute-ordering>
        <!-- 指定 Web 模块按如下顺序加载  -->
        <name>leegang</name>
        <name>crazyit</name>
    </absolute-ordering>
```

上面的配置片段指定了先加载 leegang 模块，后加载 crazyit 模块，如果重新启动该 Web 应用，将可看到 leegang 模块被优先加载。

Servlet 3.0 的 Web 模块支持为模块化开发、框架使用提供了巨大的方便，例如需要在 Web 应用中使用 Web 框架，这就只要将该框架的 JAR 包复制到 Web 应用中即可。因为这个 JAR 包的 META-INF 目录下可以通过 web-fragment.xml 文件来配置该框架所需的 Servlet、Listener、Filter 等，从而避免修改 Web 应用的 web.xml 文件。Web 模块支持对于模块化开发也有很大的帮助，开发者可以将不同模块的 Web 组件部署在不同的 web-fragment.xml 文件中，从而避免所有模块的配置、部署信息都写在 web.xml 文件中，这对以后的升级、维护将更加方便。

➤➤2.15.3　Servlet 3.0 提供的异步处理

在以前的 Servlet 规范中，如果 Servlet 作为控制器调用了一个耗时的业务方法，那么 Servlet 必须等到业务方法完全返回之后才会生成响应，这将使得 Servlet 对业务方法的调用变成一种阻塞式的调用，因此效率比较低。

Servlet 3.0 规范引入了异步处理来解决这个问题，异步处理允许 Servlet 重新发起一条新线程去调

用耗时的业务方法，这样就可避免等待。

Servlet 3.0 的异步处理是通过 AsyncContext 类来处理的，Servlet 可通过 ServletRequest 的如下两个方法开启异步调用、创建 AsyncContext 对象：

- AsyncContext startAsync()
- AsyncContext startAsync(ServletRequest, ServletResponse)

重复调用上面的方法将得到同一个 AsyncContext 对象。AsyncContext 对象代表异步处理的上下文，它提供了一些工具方法，可完成设置异步调用的超时时长，dispatch 用于请求、启动后台线程、获取 request、response 对象等功能。

下面是一个进行异步处理的 Servlet 类。

程序清单：codes\02\2.15\servlet3\WEB-INF\src\lee\AsyncServlet.java

```
@WebServlet(urlPatterns="/async",asyncSupported=true)
public class AsyncServlet extends HttpServlet
{
    @Override
    public void doGet(HttpServletRequest request
        , HttpServletResponse response)throws IOException,ServletException
    {
        response.setContentType("text/html;charset=GBK");
        PrintWriter out = response.getWriter();
        out.println("<title>异步调用示例</title>");
        out.println("进入 Servlet 的时间："
            + new java.util.Date() + ".<br/>");
        out.flush();
        //创建 AsyncContext，开始异步调用
        AsyncContext actx = request.startAsync();
        //设置异步调用的超时时长
        actx.setTimeout(30*1000);
        //启动异步调用的线程
        actx.start(new Executor(actx));
        out.println("结束 Servlet 的时间："
            + new java.util.Date() + ".<br/>");
        out.flush();
    }
}
```

上面的 Servlet 类中粗体字代码创建了 AsyncContext 对象，并通过该对象以异步方式启动了一条后台线程。该线程执行体模拟调用耗时的业务方法，下面是线程执行体的代码。

程序清单：codes\02\2.15\servlet3\WEB-INF\src\lee\Executor.java

```
public class Executor implements Runnable
{
    private AsyncContext actx = null;
    public Executor(AsyncContext actx)
    {
        this.actx = actx;
    }
    public void run()
    {
        try
        {
            //等待 5 秒钟，以模拟业务方法的执行
            Thread.sleep(5 * 1000);
            ServletRequest request = actx.getRequest();
            List<String> books = new ArrayList<String>();
            books.add("疯狂 Java 讲义");
            books.add("经典 Java EE 企业应用实战");
            books.add("疯狂 XML 讲义");
            request.setAttribute("books" , books);
            actx.dispatch("/async.jsp");
        }
```

```
            catch(Exception e)
            {
                e.printStackTrace();
            }
        }
    }
```

该线程执行体内让线程暂停 5 秒来模拟调用耗时的业务方法，最后调用 AsyncContext 的 dispatch 方法把请求 dispatch 到指定 JSP 页面。

被异步请求 dispatch 的目标页面需要指定 session="false"，表明该页面不会重新创建 session。下面是 async.jsp 页面的代码。

程序清单：codes\02\2.15\servlet3\async.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java"
    session="false"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<ul>
<c:forEach items="${books}" var="book">
    <li>${book}</li>
</c:forEach>
</ul>
<%out.println("业务调用结束的时间：" + new java.util.Date());
//完成异步调用
request.getAsyncContext().complete();%>
```

上面的页面只是一个普通 JSP 页面，只是使用了 JSTL 标签库来迭代输出 books 集合，因此读者需要将 JSTL 的两个 JAR 包复制到 Web 应用的 WEB-INF\lib 路径下。

对于希望启用异步调用的 Servlet 而言，开发者必须显式指定开启异步调用，为 Servlet 开启异步调用有两种方式：

- 为@WebServlet 指定 asyncSupported=true。
- 在 web.xml 文件的<servlet.../>元素中增加<async-supported.../>子元素。

例如希望开启上面 Servlet 的异步调用可通过如下配置片段：

```
<servlet>
    <servlet-name>async</servlet-name>
    <servlet-class>lee.AsyncServlet</servlet-class>
    <!-- 开启异步调用支持 -->
    <async-supported>true</async-supported>
</servlet>
<servlet-mapping>
    <servlet-name>async</servlet-name>
    <url-pattern>/async</url-pattern>
</servlet-mapping>
```

对于支持异步调用的 Servlet 来说，当 Servlet 以异步方式启用新线程之后，该 Servlet 的执行不会被阻塞，该 Servlet 将可以向客户端浏览器生成响应——当新线程执行完成后，新线程生成的响应再次被送往客户端浏览器。

通过浏览器访问上面的 Servlet 将看到如图 2.51 所示的页面。

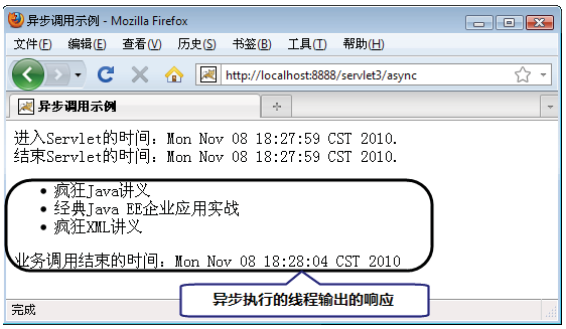

图 2.51　启用异步调用的 Servlet

当 Servlet 启用异步调用的线程之后，该线程的执行过程对开发者是透明的。但在有些情况下，开发者需要了解该异步线程的执行细节，并针对特定的执行结果进行针对性处理，这可借助于 Servlet 3.0 提供的异步监听器来实现。

异步监听器需要实现 AsyncListener 接口，实现该接口的监听器类需要实现如下 4 个方法。

- onStartAsync(AsyncEvent event)：当异步调用开始时触发该方法。
- onComplete(AsyncEvent event)：当异步调用完成时触发该方法。
- onError(AsyncEvent event)：当异步调用出错时触发该方法。
- onTimeout(AsyncEvent event)：当异步调用超时时触发该方法。

接下来为上面的异步调用定义如下监听器类。

程序清单：codes\02\2.15\servlet3\WEB-INF\src\lee\MyAsyncListener.java

```
public class MyAsyncListener
      implements AsyncListener
{
      public void onComplete(AsyncEvent event)
            throws IOException
      {
            System.out.println("------异步调用完成------" + new Date());
      }
      public void onError(AsyncEvent event)
            throws IOException
      {}
      public void onStartAsync(AsyncEvent event)
            throws IOException
      {
            System.out.println("------异步调用开始------" + new Date());
      }
      public void onTimeout(AsyncEvent event)
            throws IOException
      {}
}
```

上面实现的异步监听器类只实现了 onStartAsync、onComplete 两个方法，表明该监听器只能监听异步调用开始、异步调用完成两个事件。提供了异步监听器之后，还需要通过 AsyncContext 来注册监听器，调用该对象的 addListener()方法即可注册监听器。例如在上面的 Servlet 中增加如下代码即可注册监听器：

```
AsyncContext actx = request.startAsync();
//为该异步调用注册监听器
actx.addListener(new MyAsyncListener());
...
```

一旦通过上面的粗体字代码为异步调用注册了监听器之后，接下来的异步调用过程将会不断地触发该监听器的不同方法。

提示：虽然上面的 MyAsyncListener 监听器类可以监听异步调用开始、异步调用完成两个事件，但从实际运行的结果来看，它并不能监听到异步调用开始事件，这可能是因为注册该监听器时异步调用已经开始了的缘故。

需要指出的是，虽然上面介绍的例子都是基于 Servlet 的，但由于 Filter 与 Servlet 具有很大的相似性，因此 Servlet 3.0 规范完全支持在 Filter 中使用异步调用。在 Filter 中进行异步调用与在 Servlet 中进行异步调用的效果完全相似，故此处不再赘述。

2.15.4 改进的 Servlet API

Servlet 3.0 还有一个改变是改进了部分 API，这种改进很好地简化了 Java Web 开发。其中两个较大的改进是：

- HttpServletRequest 增加了对文件上传的支持。
- ServletContext 允许通过编程的方式动态注册 Servlet、Filter。

HttpServletRequest 提供了如下两个方法来处理文件上传。

- Part getPart(String name)：根据名称来获取文件上传域。
- Collection<Part> getParts()：获取所有的文件上传域。

上面两个方法的返回值都涉及一个 API：Part，每个 Part 对象对应于一个文件上传域，该对象提供了大量方法来访问上传文件的文件类型、大小、输入流等，并提供了一个 write(String file)方法将上传文件写入服务器磁盘。

为了向服务器上传文件，需要在表单里使用<input type="file" .../>文件域，这个文件域会在 HTML 页面上产生一个单行文本框和一个“浏览”按钮，浏览者可通过该按钮选择需要上传的文件。除此之外，上传文件一定要为表单域设置 enctype 属性。

表单的 enctype 属性指定的是表单数据的编码方式，该属性有如下三个值。

- application/x-www-form-urlencoded：这是默认的编码方式，它只处理表单域里的 value 属性值，采用这种编码方式的表单会将表单域的值处理成 URL 编码方式。
- multipart/form-data：这种编码方式会以二进制流的方式来处理表单数据，这种编码方式会把文件域指定文件的内容也封装到请求参数里。
- text/plain：这种编码方式当表单的 action 属性为 mailto:URL 的形式时比较方便，这种方式主要适用于直接通过表单发送邮件的方式。

如果将 enctype 设置为 application/x-www-form-urlencoded，或不设置 enctype 属性，提交表单时只会发送文件域的文本框里的字符串，也就是浏览者所选择文件的绝对路径，对服务器获取该文件在客户端上的绝对路径没有任何作用，因为服务器不可能访问客户机的文件系统。

下面定义了一个文件上传的页面。

程序清单：codes\02\2.15\servlet3\upload.jsp

```
<%@ page contentType="text/html; charset=GBK" language="java" errorPage="" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
      "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
      <title> 文件上传 </title>
</head>
<body>
<form method="post" action="upload" enctype="multipart/form-data">
      文件名：<input type="text" id="name" name="name" /><br/>
      选择文件：<input type="file" id="file" name="file" /><br/>
      <input type="submit" value="上传" /><br/>
```

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

上面的页面中的表单需要设置 enctype="multipart/form-data"，这表明该表单可用于上传文件。上面的表单中定义了两个表单域：一个普通的文本框，它将生成普通请求参数；一个文件上传域，它用于上传文件。

对于传统的文件上传需要借助于 common-fileupload 等工具，处理起来极为复杂，借助于 Servlet 3.0 的 API，处理文件上传将变得十分简单。看下面的 Servlet 类代码。

程序清单：codes\02\2.15\servlet3\WEB-INF\src\lee\UploadServlet.java

```
@WebServlet(name="upload" , urlPatterns={"/upload"})
@MultipartConfig
public class UploadServlet extends HttpServlet
{
   public void service(HttpServletRequest request ,
          HttpServletResponse response)
          throws IOException , ServletException
     {
          response.setContentType("text/html;charset=GBK");
          PrintWriter out = response.getWriter();
          //获取普通请求参数
          String fileName = request.getParameter("name");
          //获取文件上传域
          Part part = request.getPart("file");
          //获取上传文件的类型
          out.println("上传文件的类型为："
               + part.getContentType() + "<br/>");
          //获取上传文件的大小
          out.println("上传文件的的大小为："
               + part.getSize()  + "<br/>");
          //获取该文件上传域的 Header Name
          Collection<String> headerNames = part.getHeaderNames();
          //遍历文件上传域的 Header Name、Value
          for (String headerName : headerNames)
          {
               out.println(headerName + "--->"
                    + part.getHeader(headerName) + "<br/>");
          }
          //将上传的文件写入服务器
          part.write(getServletContext().getRealPath("/uploadFiles")
               + "/" + fileName );                    //①
     }
}
```

上面 Servlet 使用了@MultipartConfig 修饰，处理文件上传的 Servlet 应该使用该 Annotation 修饰。接下来该 Servlet 中 HttpServletRequest 就可通过 getPart(String name)方法来获取文件上传域——就像获取普通请求参数一样。

提示：

与 Servlet 3.0 所有 Annotation 相似的是，Servlet 3.0 为@提供了相似的配置元素，我们同样可以通过在<servlet.../>元素中添加<multipart-config.../>子元素来达到相同的效果。

获取了上传文件对应的 Part 之后，可以非常简单地将文件写入服务器磁盘，如上面的①号代码所示。当然也可以通过 Part 获取所上传文件的文件类型、文件大小等各种详细信息。

例如我们选择一个*.png 图片，然后单击上传将可看到如图 2.52 所示页面。

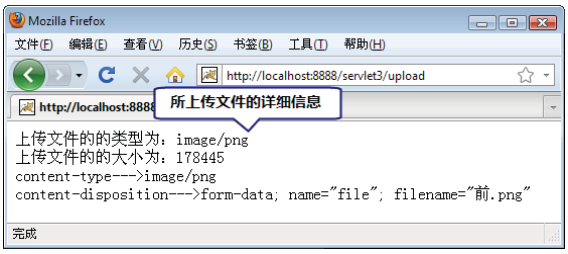

图 2.52　使用 Servlet 3.0 API 上传文件

上面的 Servlet 中将会把上传的文件保存到 Web 应用的根路径下的 uploadFiles 目录下，因此读者还应该在 Web 应用的根路径下创建 uploadFiles 目录。

[redacted]

上面 Servlet 上传时保存的文件名直接使用了 name 请求参数，实际项目中一般不会这么做，因为可能多个用户会填写相同的 name 参数，这样将导致后面用户上传的文件覆盖前面用户上传的图片。实际项目中可借助于 java.util.UUID 工具类生成文件名。

ServletContext 则提供了如下方法来动态地注册 Servet、Filter，并允许动态设置 Web 应用的初始化参数。

- 多个重载的 addServlet：动态地注册 Servlet。
- 多个重载的 addFilter：动态地注册 Filter。
- 多个重载的 addListener：动态地注册 Listener。
- setInitParameter(String name, String value)：为 Web 应用设置初始化参数。

2.16　本章小结

本章系统介绍了 Java Web 编程的相关知识：JSP、Servlet、Listener、Filter 等。本章覆盖了 JSP 所有知识点，包括 JSP 的 3 个编译指令、7 个动作指令、9 个内置对象，详细介绍了 Java Web 编程所涉及的 Servlet、Listener 和 Filter 的使用，还详细介绍了 JSP 2 自定义标签库开发步骤及标签库的用法，包括简单标签、带属性标签和迭代器标签等。本章也全面讲解了 JSP 2 所支持的配置 JSP 属性、表达式语言和 Tag File 标签支持等内容。除此之外，还重点介绍了 Servlet 3.0 新规范带来的巨大改变：Servlet、Listener、Filter 不需要通过 web.xml 进行配置，只需通过 Annotation 修饰即可。Servlet 3.0 带来的 Web 模块支持、改进的 Servlet API 都给 Web 开发带来很大方便，值得掌握。

本章内容是轻量级 Java EE 和经典 Java EE 都需要的表现层技术，因此非常重要。