Java自学宝典

黑马程序员 / 编著



添加QQ或微信号208695827，获取教学视频、源码，抢“助学红包”。

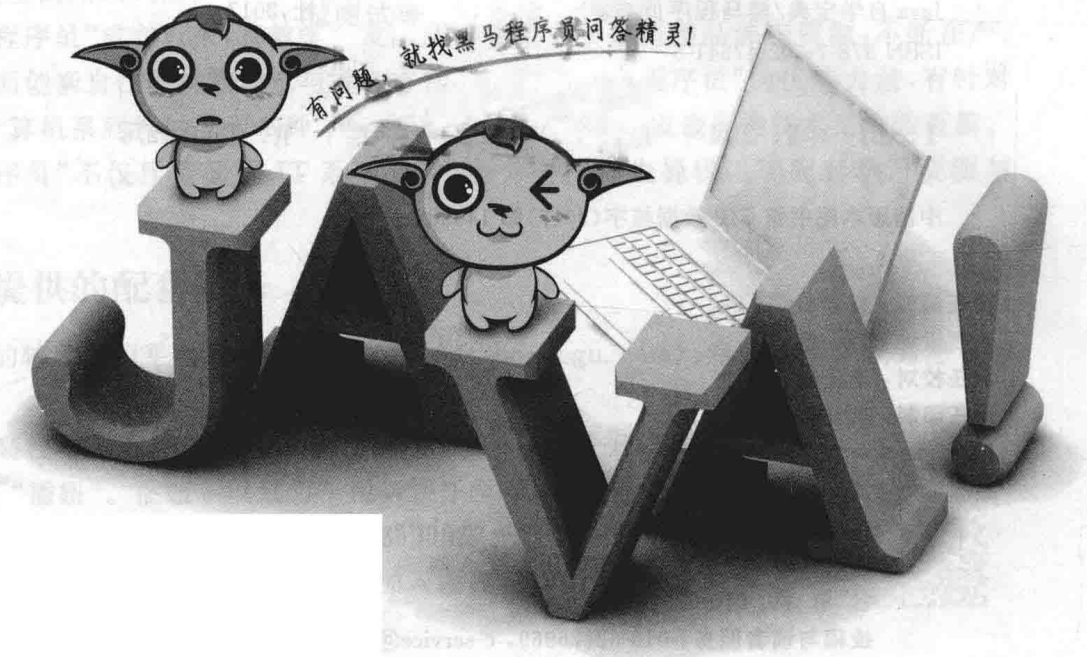
本书几乎涵盖Java学习的所有知识，包括Java基础、Java Web、SSH框架以及项目实战。

提供免费教学资源，包括案例源代码、长达90小时的教学视频等。

清华大学出版社

Java自学宝典

黑马程序员 / 编著



清华大学出版社
北京

内 容 简 介

本书从初学者的角度出发，详细讲解了从 Java 基础到 SSH 框架应用的各种技术知识。全书可分为四个部分，分别为基础篇、Web 篇、框架篇和实战篇，其中前 6 章为基础篇，主要讲解 Java 环境的搭建、语法规则、面向对象、集合和 IO 等内容；第 7～16 章为 Web 篇，主要对 Web 开发中常用的 JSP 技术、Servlet 技术、jQuery 框架、MySQL 数据库和 JDBC 等知识进行详细讲解；第 17～31 章为框架篇，主要对当前主流的 SSH(Struts2、Spring、Hibernate)框架进行详细讲解；第 32～33 章为实战篇，主要讲解 SSH 框架的整合知识以及一个基于 SSH 框架的实战项目——学生信息管理系统。

本书既可作为高等院校本/专科计算机相关专业的程序设计课程教材，也可作为 Java 技术的培训教材，是一本适合广大计算机编程爱好者的学习教材。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13701121933

图书在版编目(CIP)数据

Java 自学宝典/黑马程序员编著. —北京：清华大学出版社，2017
ISBN 978-7-302-47541-5

Ⅰ. ①J…　Ⅱ. ①黑…　Ⅲ. ①JAVA 语言－程序设计　Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 140532 号

责任编辑：袁勤勇
封面设计：韩　冬
责任校对：李建庄
责任印制：李红英

出版发行：清华大学出版社
网　　址：http://www.tup.com.cn，http://www.wqbook.com
地　　址：北京清华大学学研大厦 A 座　　**邮　　编**：100084
社 总 机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
课件下载：http://www.tup.com.cn,010-62795954
印 装 者：北京泽宇印刷有限公司
经　　销：全国新华书店
开　　本：185mm×260mm　　**印　　张**：44　　**字　　数**：1068 千字
版　　次：2017 年 10 月第 1 版　　**印　　次**：2017 年 10 月第1次印刷
印　　数：1～2000
定　　价：128.00 元

产品编号：075645-01



播妞

播妞——IT技术女神，由传智播客旗下高端教育品牌黑马程序员推出，专门服务于计算机相关专业的大学生及IT爱好者；可随时提供教材源代码、免费视频教程等。

播妞倾情寄语：

你加，或者不加我
我就在这里
不离 不弃
来我这里
或者
让我住进你的心里

★ 小心！此处常有“助学金红包”出没！★
快来领取！

播妞QQ：208695827

播妞微信：208695827

教师获取教材配套资源

添加微信/QQ

2011168841



添加播妞微信：208695827
QQ：208695827
提供教材源代码、免费视频教程



序言

传智播客和“黑马程序员”

江苏传智播客教育科技股份有限公司(简称传智播客)是一家专门致力于培养高素质软件开发人才的科技公司,“黑马程序员”是传智播客旗下的高端IT教育品牌。

“黑马程序员”的学员多为大学毕业后,想从事IT行业,但各方面条件还不成熟的年轻人。“黑马程序员”的学员筛选制度非常严格,筛选制度包括了严格的技术测试和自学能力测试,还包括性格测试、压力测试、品德测试等。百里挑一的残酷筛选制度确保了学员质量,并且降低了企业的用人风险。

自“黑马程序员”成立以来,其教学研发团队一直致力于打造精品课程资源,不断在产、学、研三个层面创新自己的执教理念与教学方针,并集中“黑马程序员”的优势力量,有针对性地出版了计算机系列教材五十多种,制作了教学视频数十套,发表各类技术文章数百篇。

“黑马程序员”不仅斥资研发IT系列教材,还为高校师生提供以下配套学习资源与服务。

为大学生提供的配套服务

1. 专业的辅助学习平台“博学谷”(http://yx.boxuegu.com),专业老师在线为您答疑解惑。

2. 针对高校学生在学习过程中存在的压力等问题,我们还面向大学生量身打造了“播妞”。播妞不仅致力于推行快乐学习,还会有定期的助学红包雨,同学们快来添加播妞微信/QQ:208695827。



3. 高校学生也可以扫描右方二维码,加入播妞粉丝团,获取最新学习资源,与播妞一起快乐学习。

为IT教师提供的配套服务

针对高校教学,“黑马程序员”为IT系列教材精心设计了“教案+授课资源+考试系统+题库+教学辅助案例”的系列教学资源,高校老师请关注码大牛老师微信/QQ:2011168841,获取教材配套资源,也可以扫描右方二维码,加入专为IT教师打造的师资服务平台——教学好助手,获取“黑马程序员”最新教师教学辅助资源相关动态。



前言

本书发展及概要

Java 语言自问世以来，已有二十多年的历史，与之相关的 Web 技术和应用也发展得非常快。在当下的网络时代，Java Web 技术更是无处不在，无论是大型互联网的开发，还是小型企业应用，都有 Java Web 技术的身影。通常所说的 Java Web 是指所有用于 Web 开发的 Java 技术总称，这些技术主要包括 Servlet、JSP、JDBC 和一些框架技术等。对于一个想从事 Web 开发的人员来说，学好 Java Web 技术知识就变得尤为重要。

为什么要学习《Java 自学宝典》

作为一本自学教材，最重要也是最难的一件事就是要将一些非常复杂、难以理解的编程思想和问题简单化，让读者能够轻松理解并快速掌握。《Java 自学宝典》采用基础知识与案例相结合的编写方式，通过基础知识的讲解和典型案例的巩固，可以很容易地帮助初学者学以致用。

本书讲解了从 Java 基础到框架应用所涉及的各种技术。读者不但能够通过自学的方式对这些技术的使用有一定的了解，还能应用这些技术开发出实际可用的功能。为了帮助读者掌握企业中应用最多的框架技术，本书在最后还讲解了一个项目实战案例——学生信息管理系统。

如何使用本书

本书共分为 33 个章节，接下来分别对每个章节进行简单介绍，具体如下。

- 第 1 章主要是 Java 语言的入门讲解，主要内容包括 Java 的相关特性、JDK 的安装和使用、环境变量的配置、第一个 Java 程序的编写、Java 的运行机制以及 Eclipse 的安装和使用。学习本章时，读者要熟记 Java 的特性，掌握 Java 开发环境的搭建，能独立编写一个简单的 Java 程序，并了解 Java 的运行机制以及 Eclipse 开发工具的使用。
- 第 2 章主要讲解学习 Java 所需的基础知识，主要内容包括 Java 语言的基本语法、常量和变量的定义、常见运算符的使用、选择结构语句和循环结构语句的概念及其使用、方法的基本概念、数组的相关操作等基础知识点。对于本章中的语法知识，读者要熟练掌握，以便为后续章节的学习打下基础。
- 第 3 章详细讲解 Java 面向对象的知识，内容主要包括面向对象的封装、继承、多态和抽象等内容。学习本章时，读者要着重理解 Java 面向对象的思想、了解类与对象的

关系、掌握构造方法中静态方法及 this 等关键字的使用，读者尽量在理解面向对象三大特性的基础上多动手练习书中的代码编写。

- 第 4～6 章主要讲解 Java 中集合、I/O 和多线程的知识。在学习这几章时，读者要熟练掌握常用集合类和输入/输出流的使用，以及创建多线程的两种方式。
- 第 7 章主要介绍开发 Web 页面时使用的基础技术知识，包括 HTML、CSS 和 JavaScript 的基础知识。学习本章时，要求读者对 HTML＋CSS＋JavaScript 基础知识形成大致的了解，并能够通过这些知识实现页面大多数的功能。
- 第 8 章讲解 Java Web 开发的一些基础技术，包括 XML、HTTP 和 Tomcat 服务器的使用等。学习本章时，读者只需要熟悉 XML 的语法、约束以及 HTTP 的请求消息和响应消息的知识，并能够掌握如何在 Eclipse 中使用 Tomcat。
- 第 9～14 章主要讲解 Java Web 中几个十分重要的核心技术，主要内容包括 Servlet 技术、请求和响应对象的应用技术、会话及会话技术、JSP 技术、EL 表达式和 JSTL 的使用技术等。学习这几章时，读者要着重掌握 Servlet 和 JSP 的使用，熟悉 HttpServletResponse 对象和 HttpServletRequest 对象的使用，掌握使用 Cookie 和 Session 保存信息，能够使用 EL 表达式和 JSTL 获取和输出信息，并能够使用过滤器和监听器实现特定的功能。
- 第 15 章讲解 Web 开发时常用的 JavaScript 库——jQuery。学习本章时，读者要熟练掌握 jQuery 的选择器和 Ajax 的使用。
- 第 16 章主要讲解 Java 数据库应用开发的相关知识，主要包括 MySQL 数据库和 JDBC 的使用。学习本章时，读者需要重点掌握 MySQL 数据库及单表的增、删、改、查操作和多表的查询操作，熟悉 JDBC 的常用 API，掌握 JDBC 操作数据库的步骤，并掌握数据库连接池的使用。
- 第 17～22 章对 Struts2 框架进行详细讲解，主要内容包括 Struts2 框架的入门程序编写、执行流程的分析、核心配置文件的介绍、拦截器和标签库的使用、OGNL 和值栈的使用，以及 Struts2 的文件上传和下载。学习这几章时，读者要重点掌握 Struts2 的核心配置文件编写以及拦截器和标签库的使用，熟悉 OGNL 表达式和值栈的应用，并能使用 Struts2 进行文件上传和下载。
- 第 23～27 章对 Hibernate 框架进行详细讲解，主要内容包括 Hibernate 的入门程序编写、核心配置文件介绍、常用 API 的使用、持久化对象的状态、缓存、关联关系映射、常用的检索方式以及 Hibernate 的事务处理等。学习这几章时，读者要着重掌握如何使用 Hibernate 增、删、改、查操作数据库，熟悉映射文件和配置文件中的配置信息，熟悉持久化类的几种状态及转换方式，掌握常用的两种关联关系及反转和级联的操作，掌握 HQL 和 QBC 检索方式的使用。
- 第 28～31 章对 Spring 框架进行详细讲解，主要内容包括 Spring 的 IoC、Bean 以及 AOP 的相关知识。学习这几章时，读者要重点掌握 Spring 的 IoC 容器配置以及依赖注入的两种方式，要熟悉实例化 Bean 和装配 Bean 的三种方式，掌握 Spring AOP 程序的编写，并熟悉 Spring 的事务处理机制。
- 第 32 章主要讲解 SSH 框架整合。学习本章时，读者要熟练掌握 SSH 框架的整合过程，并学会如何测试整合是否成功。

- 第33章讲解的是一个基于SSH框架的实战项目——学生信息管理系统。学习本章时，读者要熟悉项目各个模块的功能，并能够独立按照教材中的步骤编写功能代码。当读者熟悉教材中讲解的功能后，可自行按照所提供的其他功能页面，完善学生信息管理系统。

上述的这些章节按照不同的学习阶段，可以分为四个部分，即基础篇、Web篇、框架篇和实战篇。其中第1～6章为基础篇，第7～16章为Web篇，第17～31章为框架篇，第32～33章为实战篇。在学习过程中，读者一定要亲自实践书中的案例代码。如果读者不能完全理解书中所讲的某个知识点，建议读者先不要纠结于该知识点，可以先往后学习。通常来讲，对各种知识的学习逐渐深入后，前面看不懂的知识点一般就能理解了。如果读者在动手练习的过程中遇到问题，建议多思考，理清思路，认真分析问题发生的原因，并在问题解决后多总结。

致谢

本教材的编写和整理工作由传智播客教育科技股份有限公司完成。主要参与人员有吕春林、陈欢、韩永蒙、马丹、金鑫、潮康、杜宏、梁桐、王友军、冯佳等，全体人员在这近一年的编写过程中付出了很多辛勤的汗水，在此一并表示衷心的感谢。

意见反馈

尽管我们尽了最大的努力，但教材中难免会有不妥之处，欢迎各界专家和读者朋友们来信来函给予宝贵意见，我们将不胜感激。您在阅读本书时，如发现任何问题或有不认同之处，可以通过电子邮件与我们取得联系。

请发送电子邮件至itcast_book@vip.sina.com。

黑马程序员

2017年7月于北京

目录

第 1 章 Java 开发入门 …… 1

1.1 Java 概述 …… 1
1.2 JDK 的使用 …… 2
1.2.1 JDK 的安装 …… 2
1.2.2 JDK 的目录介绍 …… 2
1.3 环境变量 …… 4
1.3.1 path 环境变量 …… 4
1.3.2 classpath 环境变量 …… 5
1.4 第一个 Java 程序 …… 6
1.4.1 编写第一个 Java 程序 …… 6
1.4.2 Java 程序的运行机制 …… 7
1.5 Eclipse 的安装与使用 …… 7
1.5.1 Eclipse 的安装与启动 …… 8
1.5.2 Eclipse 进行程序开发 …… 9

第 2 章 Java 编程基础 …… 14

2.1 Java 的基本语法 …… 14
2.1.1 Java 代码的基本格式 …… 14
2.1.2 Java 中的注释 …… 14
2.1.3 Java 中的标识符 …… 15
2.1.4 Java 中的关键字 …… 15
2.2 Java 中的变量 …… 16
2.2.1 变量的定义 …… 16
2.2.2 变量的数据类型 …… 16
2.2.3 变量的类型转换 …… 17
2.2.4 变量的作用域 …… 19
2.3 Java 中的常量 …… 19
2.4 Java 中的运算符 …… 20
2.4.1 算术运算符 …… 20
2.4.2 赋值运算符 …… 21

专属于老师及学生的在线教育平台
yx.boxuegu.com

让 IT教学更简单

教师获取教材配套资源

添加微信/QQ
2011168841

让 IT学习更有效

学生获取配套源码及免费视频教程

添加播妞微信/Q Q
208695827

学习问答精灵：ask.boxuegu.com
更多学习视频：dvd.boxuegu.com

专属大学生的圈子

2.4.3 比较运算符 …… 22
2.4.4 逻辑运算符 …… 22
2.4.5 运算符的优先级 …… 24
2.5 选择结构语句 …… 24
2.5.1 if 条件语句 …… 24
2.5.2 switch 条件语句 …… 27
2.6 循环结构语句 …… 29
2.6.1 while 循环语句 …… 29
2.6.2 do...while 循环语句 …… 30
2.6.3 for 循环语句 …… 31
2.6.4 循环嵌套 …… 32
2.6.5 跳转语句 …… 33
2.7 方法 …… 34
2.7.1 方法的概念 …… 34
2.7.2 方法的重载 …… 35
2.8 数组 …… 37
2.8.1 数组的定义 …… 37
2.8.2 数组的常见操作 …… 38
2.8.3 Arrays 工具类 …… 41
2.9 String 类和 StringBuffer 类 …… 42
2.9.1 String 类 …… 42
2.9.2 StringBuffer 类 …… 44
2.10 包装类 …… 46

第 3 章 面向对象 …… 47

3.1 面向对象的概念和特征 …… 47
3.2 类与对象 …… 47
3.2.1 类与对象的关系 …… 47
3.2.2 类的定义 …… 48
3.2.3 对象的创建 …… 48
3.2.4 类的封装 …… 49
3.3 构造方法 …… 51
3.3.1 构造方法的定义 …… 51
3.3.2 构造方法的重载 …… 53
3.4 this 关键字 …… 54
3.5 static 关键字 …… 56
3.5.1 静态变量 …… 56
3.5.2 静态方法 …… 57
3.5.3 静态代码块 …… 58

3.6 类的继承 …… 59
3.6.1 继承的概念 …… 59
3.6.2 重写父类方法 …… 59
3.6.3 super 关键字 …… 61
3.7 抽象类和接口 …… 63
3.7.1 抽象类 …… 63
3.7.2 接口 …… 65
3.8 多态 …… 67
3.8.1 多态概述 …… 67
3.8.2 对象的类型转换 …… 69
3.8.3 匿名内部类 …… 71
3.9 异常 …… 72
3.9.1 异常的概念 …… 72
3.9.2 常见的异常类 …… 73
3.9.3 try...catch 和 finally …… 74
3.9.4 throws 关键字 …… 75
3.10 访问控制 …… 77
第 4 章 集合类 …… 78
4.1 初识集合 …… 78
4.1.1 集合概述 …… 78
4.1.2 Collection 接口简介 …… 79
4.2 List 接口 …… 79
4.2.1 List 接口简介 …… 79
4.2.2 ArrayList 集合 …… 80
4.2.3 Iterator 接口 …… 81
4.2.4 foreach 循环 …… 82
4.2.5 泛型 …… 83
4.3 Set 接口 …… 84
4.3.1 Set 接口简介 …… 84
4.3.2 HashSet 集合 …… 84
4.4 Map 接口 …… 85
4.4.1 Map 接口简介 …… 85
4.4.2 HashMap 集合 …… 86
4.4.3 Properties 集合 …… 87
第 5 章 输入/输出 …… 89
5.1 I/O 流概述 …… 89
5.2 字节流 …… 89

5.2.1 字节流的概念 …… 89
5.2.2 字节流的读/写操作 …… 91
5.2.3 文件的复制 …… 93
5.2.4 字节流的缓冲区 …… 94
5.2.5 字节缓冲流 …… 95
5.3 字符流 …… 96
5.3.1 字符流的概念 …… 96
5.3.2 字符流的读/写操作 …… 97
5.3.3 字符缓冲流 …… 98
5.3.4 转换流 …… 99
5.4 File 类 …… 100
5.4.1 File 类的常用方法 …… 100
5.4.2 遍历目录下的文件 …… 101
5.4.3 删除文件及目录 …… 102
第 6 章 多线程 …… 103
6.1 线程概述 …… 103
6.2 线程的创建 …… 103
6.2.1 继承 Thread 类 …… 103
6.2.2 实现 Runnable 接口 …… 105
6.2.3 Thread 类和 Runnable 接口对比分析 …… 106
6.3 线程的状态和转换 …… 109
6.4 多线程同步 …… 110
6.4.1 线程安全 …… 110
6.4.2 同步代码块 …… 112
6.4.3 同步方法 …… 113
第 7 章 网页开发基础 …… 115
7.1 HTML 技术 …… 115
7.1.1 HTML 简介 …… 115
7.1.2 单标记和双标记 …… 117
7.1.3 文本控制与文本样式标记 …… 117
7.1.4 图像标记 …… 118
7.1.5 表格标记 …… 119
7.1.6 表单标记 …… 121
7.1.7 列表标记和超链接标记 …… 124
7.1.8 标记 …… 126
7.2 CSS 技术 …… 127

7.2.1 CSS 简介 …… 127
7.2.2 CSS 样式的引用方式 …… 127
7.2.3 CSS 选择器和常用属性 …… 130
7.3 JavaScript 基础 …… 133
7.3.1 DOM 相关知识 …… 133
7.3.2 JavaScript 概述 …… 134
7.3.3 JavaScript 的使用 …… 139

第 8 章 Java Web 概述 …… 142

8.1 Web 开发的相关知识 …… 142
8.1.1 C/S 与 B/S 体系结构 …… 142
8.1.2 Web 资源 …… 143
8.2 XML 基础 …… 144
8.2.1 XML 概述 …… 144
8.2.2 XML 语法 …… 145
8.2.3 DTD 约束 …… 146
8.2.4 Schema 约束 …… 149
8.3 HTTP …… 152
8.3.1 HTTP 概述 …… 152
8.3.2 HTTP 请求消息 …… 153
8.3.3 HTTP 响应消息 …… 156
8.4 Tomcat …… 158
8.4.1 Tomcat 简介 …… 158
8.4.2 Tomcat 的安装和启动 …… 158
8.4.3 Tomcat 中的 Web 应用 …… 159
8.4.4 在 Eclipse 中配置 Tomcat …… 161

第 9 章 Servlet 基础 …… 165

9.1 Servlet 概述 …… 165
9.2 Servlet 开发入门 …… 166
9.2.1 Servlet 接口及其实现类 …… 166
9.2.2 实现第一个 Servlet 程序 …… 167
9.2.3 Servlet 的生命周期 …… 172
9.3 Servlet 虚拟路径映射的配置 …… 173
9.3.1 Servlet 的多重映射 …… 173
9.3.2 Servlet 映射路径中使用通配符 …… 174
9.3.3 默认 Servlet …… 175
9.4 ServletConfig 和 ServletContext …… 176

9.4.1 ServletConfig 接口 …… 176
9.4.2 ServletContext 接口 …… 178

第 10 章 请求和响应 …… 182

10.1 请求和响应概述 …… 182
10.2 HttpServletRequest 对象 …… 183
10.2.1 获取请求行信息的相关方法 …… 183
10.2.2 获取请求消息头的相关方法 …… 185
10.3 HttpServletRequest 应用 …… 186
10.3.1 获取请求参数 …… 186
10.3.2 通过 Request 对象传递数据 …… 189
10.3.3 解决请求参数的中文乱码问题 …… 190
10.4 RequestDispatcher 对象的应用 …… 191
10.4.1 RequestDispatcher 接口 …… 191
10.4.2 请求转发 …… 191
10.5 HttpServletResponse 对象 …… 193
10.5.1 发送状态码相关的方法 …… 193
10.5.2 发送响应消息头相关的方法 …… 194
10.5.3 发送响应消息体相关的方法 …… 195
10.6 HttpServletResponse 应用 …… 195
10.6.1 解决中文输出乱码问题 …… 195
10.6.2 实现请求重定向 …… 197

第 11 章 JSP 技术 …… 201

11.1 JSP 概述 …… 201
11.1.1 JSP …… 201
11.1.2 第一个 JSP 文件的创建 …… 201
11.1.3 JSP 运行原理 …… 203
11.2 JSP 的脚本元素和注释 …… 204
11.2.1 JSP 脚本元素 …… 204
11.2.2 JSP 注释 …… 206
11.3 JSP 指令 …… 207
11.3.1 page 指令 …… 207
11.3.2 include 指令 …… 208
11.4 JSP 隐式对象 …… 209
11.4.1 隐式对象的概述 …… 209
11.4.2 out 对象 …… 210
11.4.3 pageContext 对象 …… 211

11.4.4 exception 对象 …… 213
11.5 JSP 动作元素 …… 215
11.5.1 <jsp:include>动作元素 …… 215
11.5.2 <jsp:forward>动作元素 …… 215
11.6 JSP 开发模型 …… 217
11.6.1 JavaBean …… 217
11.6.2 JSP 设计模式 …… 218

第 12 章 Cookie 和 Session …… 221

12.1 Cookie 对象 …… 221
12.1.1 Cookie …… 221
12.1.2 Cookie API …… 222
12.1.3 显示用户上次访问的时间 …… 223
12.2 Session 对象 …… 225
12.2.1 Session …… 225
12.2.2 HttpSession API …… 226
12.2.3 Session 超时管理 …… 227
12.2.4 实现用户登录 …… 228

第 13 章 EL 表达式和 JSTL …… 238

13.1 EL 表达式 …… 238
13.1.1 EL 概述 …… 238
13.1.2 EL 中的标识符 …… 238
13.1.3 EL 中的保留字 …… 239
13.1.4 EL 中的变量 …… 239
13.1.5 EL 中的常量 …… 239
13.1.6 EL 中的运算符 …… 240
13.1.7 EL 隐式对象 …… 243
13.2 JSTL …… 248
13.2.1 JSTL …… 248
13.2.2 JSTL 的配置和测试 …… 249
13.2.3 JSTL 中的 Core 标签库 …… 250

第 14 章 Servlet 高级 …… 257

14.1 Filter …… 257
14.1.1 Filter 概述 …… 257
14.1.2 实现第一个 Filter 程序 …… 258
14.1.3 Filter 映射 …… 261
14.1.4 Filter 链 …… 264

14.1.5 FilterConfig 接口 …… 267
14.1.6 统一全站编码 …… 269
14.2 Listener …… 273
14.2.1 Servlet 事件监听器概述 …… 273
14.2.2 统计当前在线人数 …… 274

第 15 章 jQuery 框架 …… 278

15.1 jQuery 概述 …… 278
15.1.1 jQuery …… 278
15.1.2 jQuery 的下载与使用 …… 278
15.2 jQuery 选择器 …… 280
15.2.1 基本选择器 …… 280
15.2.2 层次选择器 …… 282
15.2.3 过滤选择器 …… 284
15.2.4 表单选择器 …… 293
15.3 jQuery 的事件处理 …… 296
15.3.1 常用事件 …… 296
15.3.2 页面加载事件 …… 296
15.3.3 事件绑定与切换 …… 297
15.4 jQuery 的 Ajax 操作 …… 299
15.4.1 Ajax 概述 …… 299
15.4.2 jQuery 的 Ajax 操作 …… 300

第 16 章 MySQL 和 JDBC …… 306

16.1 数据库基础知识 …… 306
16.1.1 数据库概述 …… 306
16.1.2 MySQL 的安装与配置 …… 306
16.1.3 MySQL 的图形化工具 …… 307
16.2 MySQL 的基本使用 …… 309
16.2.1 数据库的基本操作 …… 309
16.2.2 表的基本操作 …… 310
16.2.3 数据的基本操作 …… 313
16.2.4 单表查询 …… 314
16.3 MySQL 的多表操作 …… 320
16.3.1 外键约束 …… 320
16.3.2 内连接 …… 322
16.3.3 外连接 …… 323
16.3.4 子查询 …… 324
16.4 JDBC 技术简介 …… 326

16.5 JDBC 常用 API …… 327
16.5.1 Driver 接口 …… 327
16.5.2 DriverManager 类 …… 327
16.5.3 Connection 接口 …… 327
16.5.4 Statement 接口 …… 327
16.5.5 PreparedStatement 接口 …… 328
16.5.6 ResultSet 接口 …… 329
16.6 实现第一个 JDBC 程序 …… 329
16.7 数据库连接池 …… 334
16.7.1 数据库连接池 …… 334
16.7.2 DataSource 接口 …… 334
16.7.3 C3P0 数据源 …… 335

第 17 章 Struts2 框架入门 …… 338

17.1 Struts2 简介 …… 338
17.1.1 Struts2 概述 …… 338
17.1.2 Struts2 的下载 …… 339
17.1.3 Struts2 的使用 …… 339
17.2 Struts2 入门 …… 340
17.2.1 第一个 Struts2 程序 …… 340
17.2.2 Struts2 的执行流程分析 …… 345

第 18 章 Struts2 的核心配置 …… 348

18.1 struts.xml 文件的配置 …… 348
18.1.1 struts.xml 文件 …… 348
18.1.2 常量配置 …… 349
18.1.3 包配置 …… 350
18.1.4 包含配置 …… 350
18.2 Action 的配置 …… 351
18.2.1 实现 Action 控制类 …… 351
18.2.2 配置 Action …… 352
18.2.3 使用通配符 …… 353
18.3 Action 访问 Servlet API …… 354
18.3.1 通过 ActionContext 访问 …… 354
18.3.2 通过 ServletActionContext 访问 …… 358
18.4 Action 处理请求参数 …… 360
18.4.1 属性驱动 …… 360
18.4.2 模型驱动 …… 364
18.5 Result 结果类型 …… 365

18.5.1 配置 Result …… 365
18.5.2 预定义的结果类型 …… 366
18.5.3 dispatcher 结果类型 …… 367
18.5.4 redirect 结果类型 …… 368

第 19 章 Struts2 的拦截器 …… 370

19.1 拦截器简介 …… 370
19.1.1 拦截器概述 …… 370
19.1.2 拦截器的配置 …… 371
19.2 Struts2 的内建拦截器 …… 373
19.2.1 内建拦截器的介绍 …… 373
19.2.2 内建拦截器的配置 …… 375
19.3 自定义拦截器 …… 376
19.3.1 实现自定义拦截器 …… 376
19.3.2 使用拦截器实现权限控制 …… 377

第 20 章 Struts2 的标签库 …… 385

20.1 Struts2 标签库概述 …… 385
20.1.1 Struts2 标签库的分类 …… 385
20.1.2 Struts2 标签的使用 …… 386
20.2 Struts2 的控制标签 …… 386
20.2.1 <s:if>、<s:elseif>、<s:else>标签 …… 386
20.2.2 <s:iterator>标签 …… 386
20.3 Struts2 的数据标签 …… 389
20.3.1 <s:property>标签 …… 389
20.3.2 <s:a>标签 …… 390
20.3.3 <s:debug>标签 …… 390
20.3.4 <s:include>标签 …… 390
20.3.5 <s:param> …… 392
20.4 Struts2 的模板和主题 …… 392
20.5 Struts2 的表单标签 …… 393
20.5.1 表单标签的公共属性 …… 393
20.5.2 <s:form>标签 …… 394
20.5.3 <s:textfield>和<s:textarea>标签 …… 394
20.5.4 <s:password>标签 …… 394
20.5.5 <s:radio>标签 …… 395
20.5.6 <s:reset>标签 …… 395
20.5.7 <s:submit>标签 …… 396
20.5.8 <s:checkboxlist>标签 …… 397

20.5.9 <s:select>标签 …… 398
20.5.10 <s:optgroup>标签 …… 399
20.5.11 <s:file>标签 …… 400
20.5.12 <s:hidden>标签 …… 400
20.6 Struts2 的非表单标签 …… 401

第 21 章 Struts2 的 OGNL 和值栈 …… 403

21.1 OGNL …… 403
21.1.1 OGNL 概述 …… 403
21.1.2 OGNL 表达式 …… 404
21.1.3 使用 OGNL 访问对象方法和静态方法 …… 408
21.2 值栈 …… 412
21.2.1 值栈的意义 …… 412
21.2.2 值栈的内部结构 …… 413
21.2.3 值栈在开发中的应用 …… 415
21.2.4 通过 EL 访问值栈的数据 …… 420

第 22 章 Struts2 的文件上传和下载 …… 421

22.1 文件上传 …… 421
22.1.1 文件上传的概述 …… 421
22.1.2 单个文件上传 …… 423
22.1.3 限制文件的大小和类型 …… 426
22.2 文件下载 …… 428
22.2.1 文件下载的概述 …… 428
22.2.2 文件下载 …… 429
22.2.3 中文文件的下载 …… 431

第 23 章 Hibernate 框架入门 …… 435

23.1 Hibernate 概述 …… 435
23.2 Hibernate 下载和目录介绍 …… 435
23.3 第一个 Hibernate 程序 …… 437
23.3.1 编写 Hibernate 程序 …… 437
23.3.2 测试 Hibernate 程序 …… 441
23.4 Hibernate 的运行机制 …… 447
23.5 Hibernate 的核心文件 …… 448
23.5.1 *.hbm.xml 映射文件 …… 448
23.5.2 hibernate.cfg.xml 配置文件 …… 452
23.6 Hibernate 的核心 API …… 454

23.6.1 Configuration …… 454
23.6.2 SessionFactory …… 454
23.6.3 Session …… 455
23.6.4 Transaction …… 456
23.6.5 Query …… 457
23.6.6 Criteria …… 459

第 24 章 Hibernate 的持久化对象状态和一级缓存 …… 462

24.1 Hibernate 持久化对象的状态 …… 462
24.1.1 持久化对象的状态及状态转换 …… 462
24.1.2 演示持久化对象的状态转换 …… 463
24.2 Hibernate 的一级缓存 …… 467
24.2.1 一级缓存 …… 467
24.2.2 Hibernate 的快照技术 …… 469
24.2.3 一级缓存常用操作 …… 470

第 25 章 Hibernate 的关联关系映射 …… 474

25.1 关联关系映射 …… 474
25.1.1 三种关联关系 …… 474
25.1.2 一对多关联关系映射 …… 475
25.1.3 多对多关联关系映射 …… 481
25.2 关联关系中的反转与级联 …… 485
25.2.1 反转操作 …… 485
25.2.2 级联操作 …… 487

第 26 章 Hibernate 的检索方式 …… 493

26.1 Hibernate 检索方式的概述 …… 493
26.1.1 导航对象图检索方式 …… 493
26.1.2 OID 检索方式 …… 493
26.1.3 HQL 检索方式 …… 493
26.1.4 QBC 检索方式 …… 494
26.1.5 本地 SQL 检索方式 …… 495
26.2 HQL 检索 …… 495
26.2.1 指定别名 …… 495
26.2.2 投影查询 …… 496
26.2.3 动态实例查询 …… 497
26.2.4 条件查询 …… 498
26.2.5 分页查询 …… 501

26.3 QBC 检索 …… 502
26.3.1 组合查询 …… 502
26.3.2 分页查询 …… 504

第 27 章 Hibernate 的事务处理和二级缓存 …… 506

27.1 事务 …… 506
27.1.1 事务的特性 …… 506
27.1.2 事务的隔离级别 …… 506
27.2 Hibernate 的事务处理 …… 507
27.2.1 事务的配置 …… 507
27.2.2 悲观锁 …… 508
27.2.3 乐观锁 …… 514
27.3 Hibernate 的二级缓存 …… 516
27.3.1 二级缓存的原理和分类 …… 516
27.3.2 二级缓存的并发访问策略 …… 517
27.3.3 二级缓存的常用插件 …… 518
27.3.4 二级缓存的配置和使用 …… 518

第 28 章 Spring 框架入门 …… 522

28.1 Spring 基本知识 …… 522
28.1.1 Spring 框架概述 …… 522
28.1.2 Spring 的体系结构 …… 523
28.1.3 Spring 的下载及目录结构 …… 524
28.2 Spring 的 IoC 容器 …… 526
28.2.1 BeanFactory …… 526
28.2.2 ApplicationContext …… 526
28.3 编写第一个 Spring 程序 …… 527
28.4 依赖注入 …… 530

第 29 章 Spring 中的 Bean …… 533

29.1 Bean 的配置 …… 533
29.2 Bean 的实例化 …… 534
29.2.1 构造器实例化 …… 534
29.2.2 静态工厂方式实例化 …… 536
29.2.3 实例工厂方式实例化 …… 538
29.3 Bean 的作用域 …… 540
29.3.1 作用域的种类 …… 540
29.3.2 singleton 作用域 …… 540
29.3.3 prototype 作用域 …… 541

29.4 Bean 的生命周期 …… 542
29.5 Bean 的装配方式 …… 543
29.5.1 基于 XML 的装配 …… 543
29.5.2 基于 Annotation 的装配 …… 546
29.5.3 自动装配 …… 550

第 30 章 面向切面编程(Spring AOP) …… 552

30.1 Spring AOP 概述 …… 552
30.2 手动代理 …… 552
30.2.1 JDK 动态代理 …… 553
30.2.2 CGLIB 代理 …… 556
30.3 声明式工厂 Bean …… 559
30.3.1 Spring 通知类型 …… 559
30.3.2 声明式 Spring AOP …… 559
30.4 AspectJ …… 562
30.4.1 基于 XML 的声明式 …… 562
30.4.2 基于 Annotation 的声明式 …… 566

第 31 章 Spring 事务管理 …… 571

31.1 Spring JDBCTemplate …… 571
31.2 Spring 事务管理的三个核心接口 …… 573
31.3 Spring 声明式事务管理 …… 574
31.3.1 基于 XML 方式 …… 574
31.3.2 基于 Annotation 方式 …… 580

第 32 章 SSH 框架整合 …… 584

32.1 准备整合环境 …… 584
32.1.1 数据库环境 …… 584
32.1.2 配置 Struts2 环境 …… 584
32.1.3 配置 Spring 环境 …… 586
32.1.4 配置 Hibernate 环境 …… 588
32.2 Spring 与 Hibernate 整合 …… 590
32.3 Spring 与 Struts2 整合 …… 594
32.4 测试框架分层的整合 …… 597
32.5 验证事务是否生效 …… 600
32.5.1 测试事务只读方法 …… 600
32.5.2 测试事务回滚 …… 602

第 33 章 学生信息管理系统 …… 603

33.1 系统概述 …… 603
33.1.1 系统的功能模块 …… 603
33.1.2 系统架构设计 …… 604
33.2 数据库设计 …… 605
33.3 系统环境搭建 …… 611
33.4 员工登录模块 …… 618
33.4.1 系统登录 …… 618
33.4.2 实现登录拦截 …… 628
33.4.3 注销用户 …… 630
33.5 人力资源部模块 …… 630
33.5.1 部门管理 …… 631
33.5.2 职务管理 …… 639
33.5.3 员工管理 …… 649
33.6 教学部模块 …… 656
33.6.1 课程类别 …… 656
33.6.2 班级管理 …… 664

第1章 Java开发入门

Java是一门面向对象的编程语言，广泛应用于计算机、手机、家用电器等领域。在当今互联网时代，Java更具备了显著的优势和广阔的前景。本章将对Java语言的特点、开发环境、运行机制以及如何编写Java程序等内容进行详细讲解。

1.1 Java概述

Java语言于1995年5月由Sun公司(已被Oracle公司收购)推出，它具有简单易用、安全可靠、面向对象等特点。下面针对Java语言的几个突出特性进行介绍。

1. 简单易用

Java是一种相对简单的编程语言，它通过提供最基本的方法完成指定的任务，只需要掌握一些基础的概念，就可以编写出适用于各种场景的应用程序。

2. 安全可靠

Java通常用于网络环境中，为此，Java提供了安全机制以防恶意代码的攻击。Java程序运行之前会利用字节确认器进行代码的安全检查，确保程序不会存在非法访问本地资源和文件系统的可能，保证了程序在网络间传送的安全性。

3. 跨平台性

Java引进了虚拟机的概念，通过Java虚拟机(Java Virtual Machine，简称JVM)可以在不同的操作系统(如Windows、Linux等)运行Java程序，从而实现跨平台特性。

4. 面向对象

Java将一切事物都看成对象，通过面向对象的方式，将现实世界的事物抽象成对象，将现实世界中的关系(如父子关系)抽象为继承。这种面向对象的方法，更利于人们理解、分析、设计和编写复杂程序。

5. 支持多线程

Java语言内置了多线程控制，可使用户程序并发执行。利用Java的多线程编程接口，开发人员可以方便地写出多线程的应用程序。Java语言提供的同步机制可保证各线程对共享数据的正确操作。在硬件条件允许的情况下，这些线程可以直接分布到各个CPU上，充分发挥硬件性能，提高程序执行效率。

为了使软件开发人员、设备生产商和服务提供商可以针对特定的市场进行开发，Sun公司将Java划分为三种不同的技术平台，分别是Java SE、Java EE和Java ME。

- Java SE(Java Platform Standard Edition)标准版，是为开发普通桌面和商务应用程序提供的解决方案。
- Java EE(Java Platform Enterprise Edition)企业版，是为开发企业级应用程序提供的解决方案。
- Java ME(Java Platform Micro Edition)小型版，是为开发电子消费产品和嵌入式设备提供的解决方案。

1.2 JDK 的使用

Sun 公司提供了一套 Java 开发环境，简称 JDK(Java Development Kit)，它是 Java 的核心，其中包括 Java 编译器、Java 运行环境、Java 打包工具、Java 文档生成工具等。其中 Java 运行环境(Java Runtime Environment，简称 JRE)只能运行事先编写好的程序，不能用于编译代码，因此通常提供给普通用户使用。由于 JDK 中自带了 JRE，因此只需要在计算机中安装 JDK 即可，不需要专门安装 JRE。为了满足用户的需求，JDK 的版本也在不断升级，从 JDK 1.1 开始一直到 JDK 8.0，本书使用的是 JDK 7.0，接下来将针对 JDK 的安装与使用进行详细讲解。

1.2.1 JDK 的安装

Oracle 公司根据不同的操作系统提供了不同的 JDK，在 Oracle 官网 http://www.oracle.com/technetwork/java/javase/downloads/index.html 即可下载对应的 JDK 安装文件。接下来以 64 位的 Windows 7 系统为例，通过图例演示 JDK 7.0 的安装过程，如图 1-1 所示。

在图 1-1 中，单击安装向导界面中的【下一步】按钮进入 JDK 的安装界面，单击该界面右侧的【更改】按钮可以指定 JDK 的安装路径，路径选择完成后单击【确定】按钮，当前界面关闭后单击安装界面中的【下一步】按钮等待安装，安装完成后单击【关闭】按钮，关闭当前窗口即可。

需要注意的是，安装过程中会弹出一个 JRE 的安装窗口，此处不需要单独安装 JRE，因为 JDK 中包含了 JRE。

1.2.2 JDK 的目录介绍

JDK 安装完成后，会在指定的路径下生成 JDK 的安装目录，如图 1-2 所示。

为了让初学者更好地了解 JDK 安装目录下各个子目录的意义和作用，下面分别对 JDK 安装目录下的子目录进行简单介绍。

- bin 目录：用于存放一些可执行程序。
- db 目录：是一个小型的数据库。
- jre 目录：为 Java 程序提供运行环境。
- include 目录：由于 JDK 是通过 C 和 C++ 实现的，因此用于存放 C 语言的头文件。
- lib 目录：lib 是 library 的缩写，是 Java 类库或库文件，是开发工具使用的归档包文件。
- src.zip 文件：src 中存放了 JDK 核心类的源代码。



图 1-1　JDK 7.0 的安装界面



图 1-2　JDK 的目录结构

注意：在 JDK 的 bin 目录下存放着很多可执行的程序，其中最重要的就是 javac.exe 和 java.exe。

- javac.exe 是 Java 编译器工具，可以将 Java 文件编译成 Java 字节码文件(可执行的 Java 程序)，Java 源文件的扩展名为 java。
- java.exe 是 Java 运行工具，它会启动一个 Java 虚拟机(JVM)进程，Java 虚拟机相当于一个虚拟的操作系统，它专门负责运行由 Java 编译器生成的字节码文件(class 文件)。

1.3 环境变量

JDK 安装完成后要进行 JDK 环境变量的配置，分别是 path 环境变量和 classpath 环境变量。其中 path 环境变量用于告知操作系统到指定路径寻找 JDK，classpath 环境变量则用于告知 JDK 到指定路径查找类文件(class 文件)，本节将针对这两个环境变量的配置进行详细讲解。

1.3.1 path 环境变量

path 环境变量用于保存一系列的路径，每个路径之间用英文“;”相隔，当在命令行窗口运行一个可执行文件时，操作系统首先会在当前目录下寻找是否存在该文件，如果不存在则会在 path 环境变量中定义的路径下寻找这个文件，如果未找到，则会报错。配置 path 环境变量的步骤如图 1-3 所示。

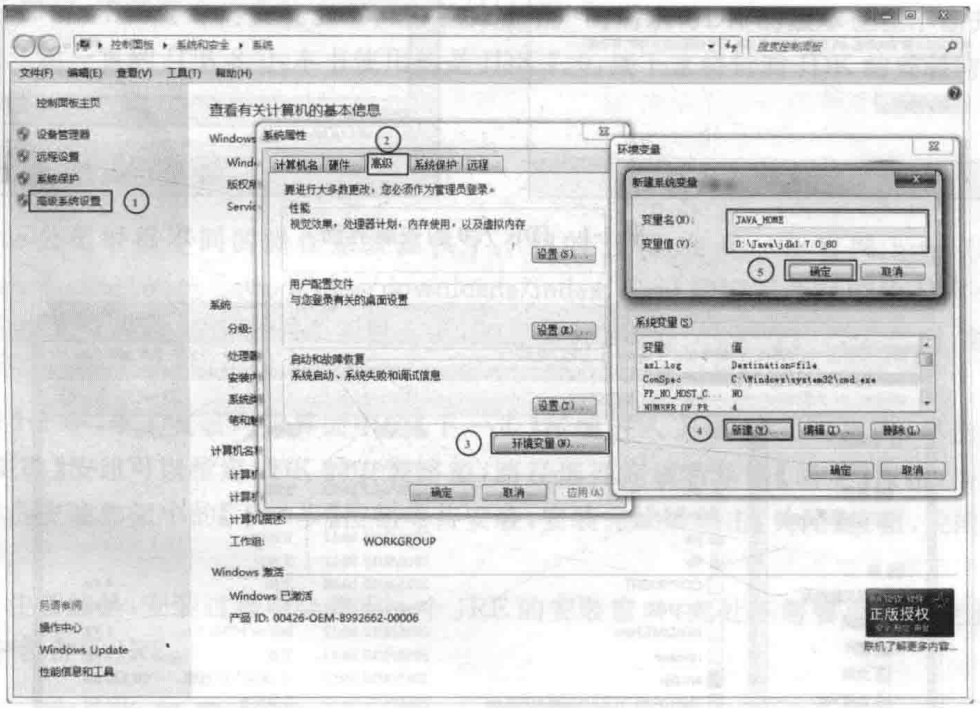


图 1-3 环境变量的窗口

在图 1-3 中，单击“新建系统变量”窗口中的【确定】按钮即可完成 JAVA_HOME 的配置。需要注意的是，要使新建的系统变量生效，还需要将 JAVA_HOME 配置到 path 环境变量中，双击系统变量中变量名称为“path”的环境变量，将“%JAVA_HOME%\bin;”添加到变量值中，其中%JAVA_HOME%代表环境变量 JAVA_HOME 的当前值，用英文分号(;)结束，与其他 path 变量值路径隔开，如图 1-4 所示。

在图 1-4 中，单击【确定】按钮完成配置，为了查看配置是否成功，打开【开始】菜单，单击【所有程序】选项，在【附件】选项中找到【命令提示符】窗口并在该窗口中执行 javac 命令，如图 1-5 所示。



图 1-4　编辑环境变量的窗口

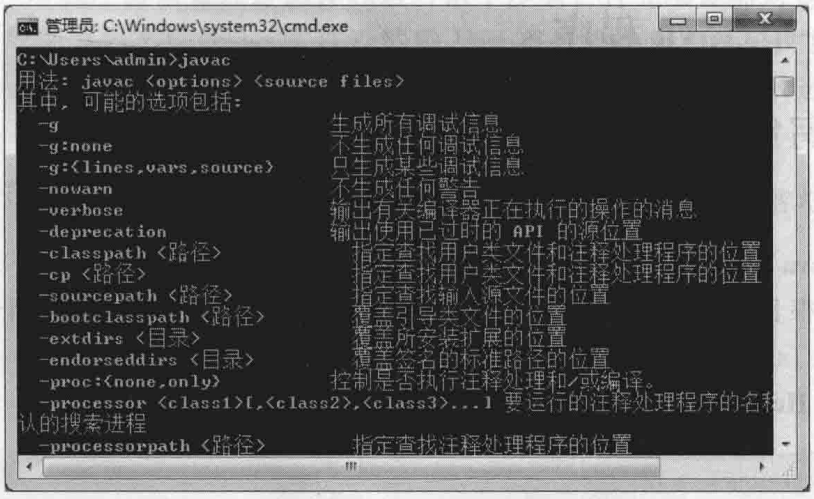


图 1-5　javac 命令帮助信息

在图 1-5 中，执行 javac 命令后，如果能正常显示帮助信息，则说明系统 path 环境变量配置成功。

1.3.2　classpath 环境变量

classpath 环境变量也用于保存一系列路径，它和 path 环境变量的查看与配置的方式完全相同。当 Java 虚拟机需要运行一个类时，会在 classpath 环境变量中所定义的路径下寻找所需的 class 文件和类包。其配置的方式与配置 JAVA_HOME 环境变量类似，只不过新建的变量名为“CLASSPATH”，变量值为“.;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\tools.jar”，如图 1-6 所示。

需要注意的是，在设置 classpath 变量时，必须在配置路径前添加“.;”(当前目录)，用于识别当前目录下的 Java 类。从 JDK 5.0 开始，如果 classpath 环境变量没有进行设置，Java 虚拟机会自动将其设置为“.”，也就是当前目录。



图 1-6 环境变量窗口

1.4 第一个 Java 程序

1.4.1 编写第一个 Java 程序

为了让初学者更好地体验 Java 程序，下面分步骤进行讲解。

1. 编写 Java 源文件

在 D 盘的根目录下新建文本文档，重命名为 HelloWorld.java。然后用记事本打开该文档，编写一段 Java 代码，如文件 1-1 所示。

文件 1-1 HelloWorld.java

```
1  class HelloWorld {
2      public static void main(String[] args) {
3          System.out.println("Java 入门程序");
4      }
5  }
```

2. 编译 Java 源文件

首先进入 Java 文件所在的目录(D 盘根目录)，然后在命令行窗口中输入 javac HelloWorld.java 命令，对源文件进行编译，如图 1-7 所示。



图 1-7 编译 HelloWorld.java 源文件

javac 命令执行完毕后，会在 D 盘根目录下生成一个字节码文件 HelloWorld.class。

3. 运行 Java 程序

在命令行窗口中输入 java HelloWorld 命令，运行 HelloWorld.class，便会输出程序结果，运行结果如图 1-8 所示。

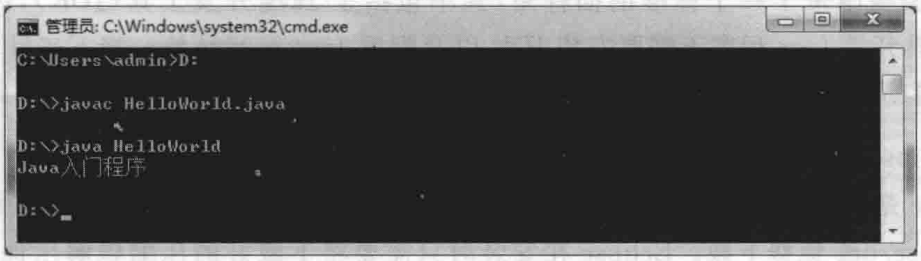


图 1-8　运行 HelloWorld 程序

1.4.2　Java 程序的运行机制

成功运行 Java 程序必须要经过编译和运行两个步骤。首先编译器对源文件(java 文件)进行编译生成字节码文件(class 文件)，然后 Java 虚拟机会对字节码文件进行解释执行，并显示结果。下面以 HelloWorld.java 文件为例，详细分析 Java 程序的运行机制，具体如图 1-9 所示。

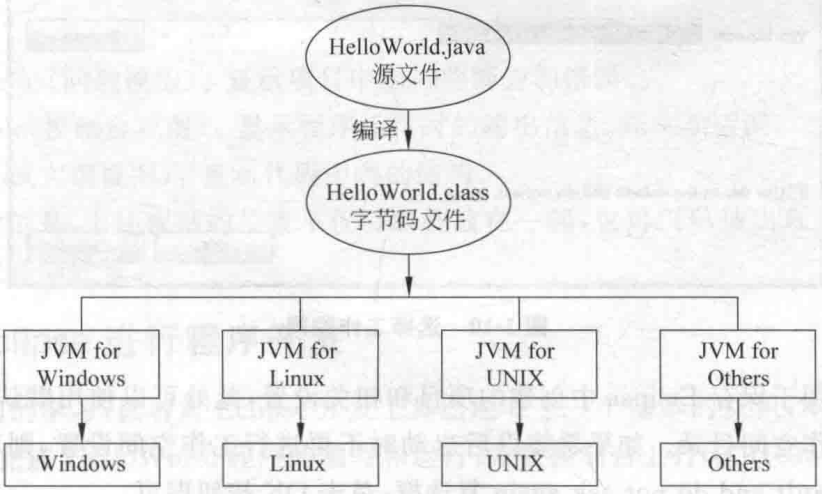


图 1-9　不同版本的虚拟机

从图 1-9 中可以看出，不同的操作系统需要使用不同版本的虚拟机，这使得 Java 语言具有“一次编写，到处运行”的特性，解决了 Java 语言在不同操作系统编译时产生不同机器代码的问题，大幅度降低了程序开发和维护的成本。

1.5　Eclipse 的安装与使用

“工欲善其事，必先利其器”，高效的开发工具，不但能带来高体验的开发环境，还能带来高效的纠错与开发提示等功能，下面介绍一种 Java 常用的开发工具——Eclipse。

1.5.1 Eclipse 的安装与启动

Eclipse 的设计思想是“一切皆插件”。就其本身而言，Eclipse 只是一个框架和一组服务，其所有功能都是将插件组件加入到 Eclipse 框架中实现的。Eclipse 作为一款优秀的开发工具，其自身附带了一个标准的插件集，其中包括了 Java 开发工具(JDK)，因此，使用 Eclipse 工具开发 Java 程序不需要安装 JDK 以及配置 Java 运行环境。接下来针对 Eclipse 的安装与启动进行详细讲解。

1. Eclipse 的安装

Eclipse 是针对 Java 编程的集成开发环境(IDE)，初学者可以登录 Eclipse 官网 http://www.eclipse.org 免费下载。Eclipse 在安装时只需要将下载好的压缩包解压保存到指定目录(如 D:\java)下就可以使用了。本书使用的 Eclipse 版本是 Mars.2 Release(4.5.2)。

2. Eclipse 的启动

Eclipse 解压完成后，在该目录中双击 eclipse.exe 文件即可运行，此时会弹出一个对话框，提示选择工作空间(Workspace)，如图 1-10 所示。

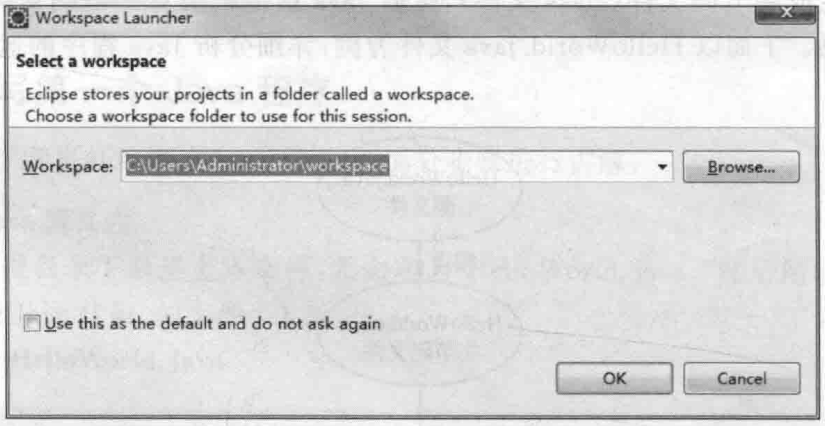


图 1-10 选择工作空间

工作空间用于保存 Eclipse 中创建的项目和相关设置，此处可以使用默认的工作空间，也可以更改工作空间目录。如果希望以后启动时不再进行工作空间设置，则可以选中 Use this as the default and do not ask again 复选框，单击 OK 按钮即可。

3. Eclipse 的工作台

Eclipse 的工作台主要由标题栏、菜单栏、工具栏、透视图四部分组成，如图 1-11 所示。

从图 1-11 中可以看到，工作台界面中有包资源管理视图、文本编辑器视图、大纲视图等多个模块，这些视图大多都是用于显示信息的层次结构和实现代码编辑，下面是 Eclipse 工作台上的几种主要视图的作用。

- Package Explorer(包资源管理器视图)：用于显示项目文件的组成结构。
- Editor(文本编辑器)：用于编写代码的区域，编辑器具有代码提示、自动补全等

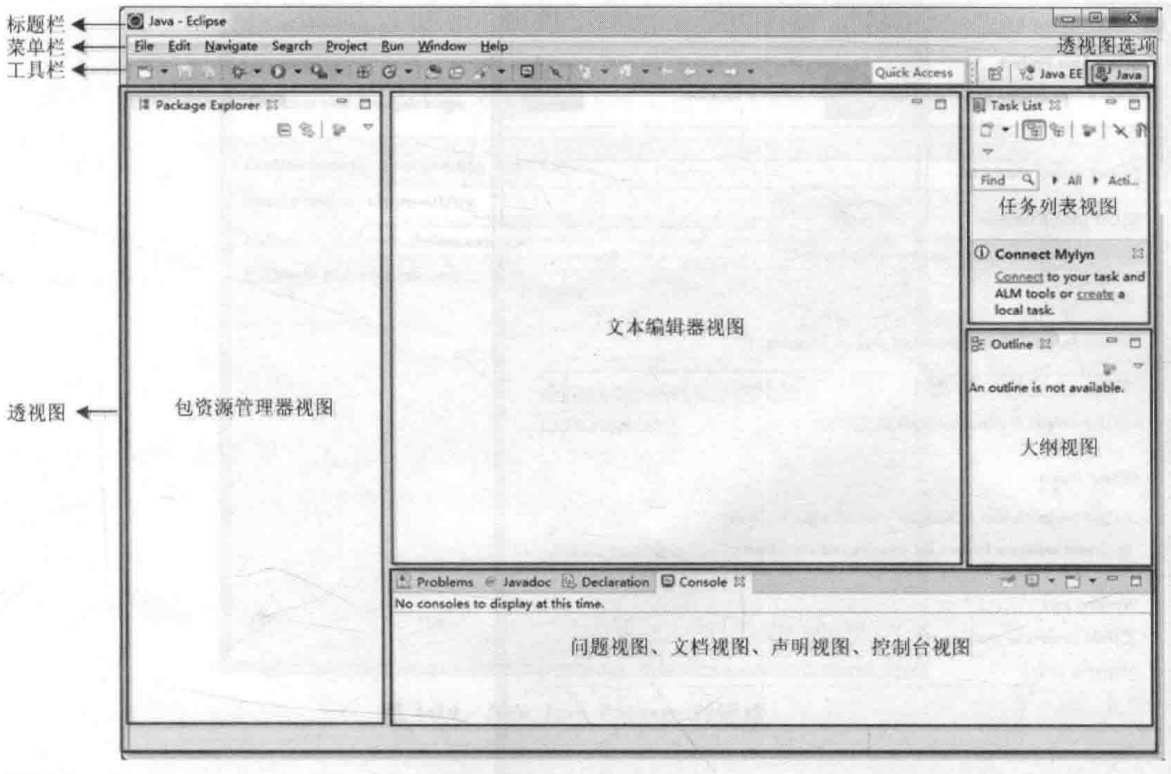


图 1-11　Eclipse 工作台

功能。

- Problems(问题视图)：显示项目中的一些警告和错误。
- Console(控制台视图)：显示程序运行时的输出信息、异常和错误。
- Outline(大纲视图)：显示代码中类的结构。

需要注意的是，上述视图的位置不仅可以叠放在一起，也可以单独出现，并且位置可以随意拖动改变布局效果。

1.5.2　Eclipse 进行程序开发

通过前面的学习，读者对 Eclipse 开发工具已经有了一个基本的认识，本节将介绍如何使用 Eclipse 完成 HelloWorld 程序的编写和运行，并在控制台上打印“Hello World!”，具体步骤如下。

1. 创建 Java 项目

在 Eclipse 窗口中选择 File→New→Java Project，然后选择 New→Java Project，会出现一个 New Java Project 对话框，如图 1-12 所示。

图 1-12 对话框中的 Project name 文本框表示项目的名称，这里将项目命名为 chapter01(本书以章节编号作为项目名称，例如第 2 章的项目名为 chapter02)，选择 JRE 第二项 Use a project specific JRE，然后单击 Finish 按钮完成项目的创建。这时，在 Package Explorer 视图中便会出现一个名称为 chapter01 的 Java 项目，如图 1-13 所示。

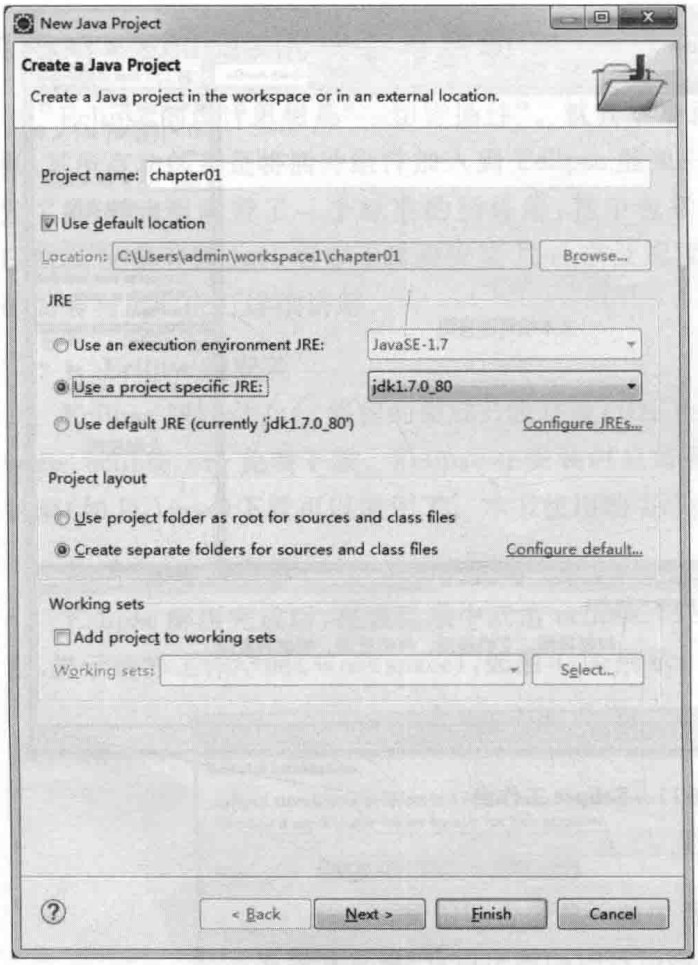


图 1-12 New Java Project 对话框

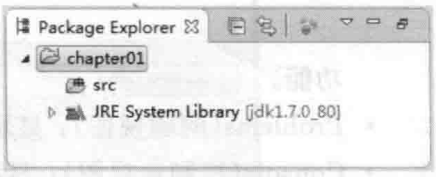


图 1-13 Package Explorer 页面视图

2. 在项目下创建包

在 Package Explorer 视图中，右击 chapter01 项目下的 src 文件夹，选择 New→Package，会出现一个 New Java Package 对话框，其中 Source folder 文本框表示项目所在的目录，Name 文本框表示包的名称，这里将包命名为 com. itheima. example(包名通常以公司域名的反写作为前缀)，如图 1-14 所示。

3. 创建 Java 类

右击包名，选择 New→Class，会出现一个 New Java Class 对话框，如图 1-15 所示。

图 1-15 对话框中的 Name 文本框表示类名，这里创建一个 HelloWorld 类，单击 Finish 按钮，就完成了 HelloWorld 类的创建。此时，在 com. itheima. example 包下出现了一个 HelloWorld. java 文件，如图 1-16 所示。

4. 编写程序代码

创建 HelloWorld 类后，在文本编辑器中只写入 main()方法和一条输出语句“System. out. println("Hello World !");”，如图 1-17 所示。

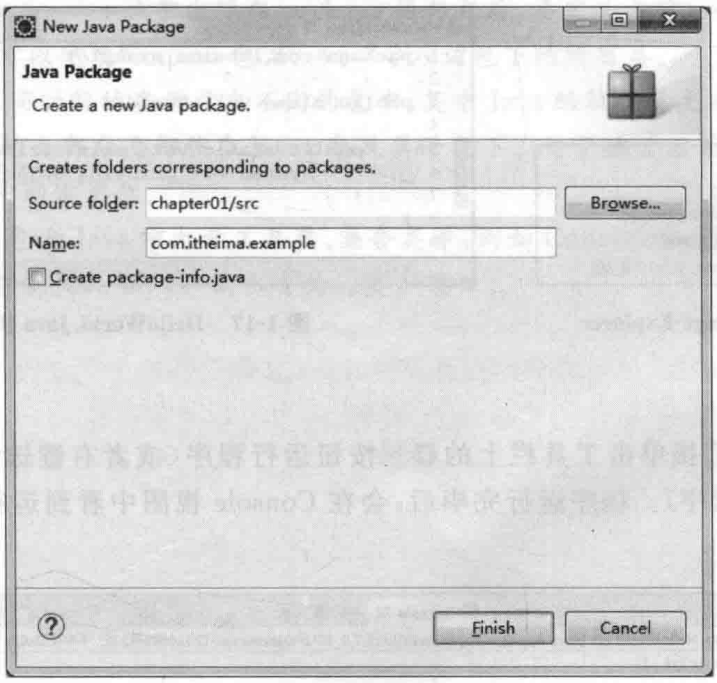


图 1-14　New Java Package 对话框



图 1-15　New Java Class 对话框

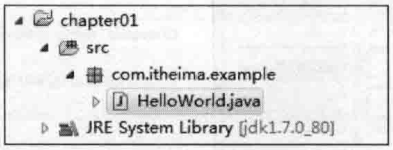


图 1-16 Package Explorer

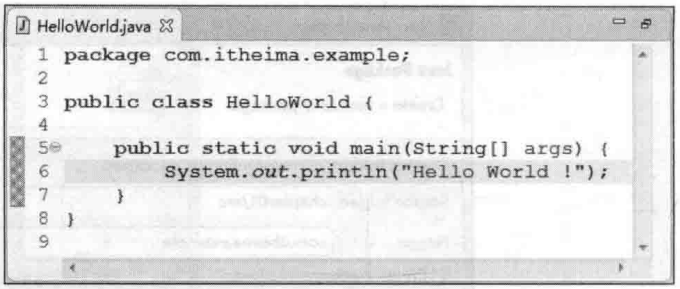


图 1-17 HelloWorld.java 代码

5. 运行程序

选中文件后，直接单击工具栏上的 ▶ 按钮运行程序（或者右键选择 Run As→Java Application 运行程序）。程序运行完毕后，会在 Console 视图中看到运行结果，如图 1-18 所示。

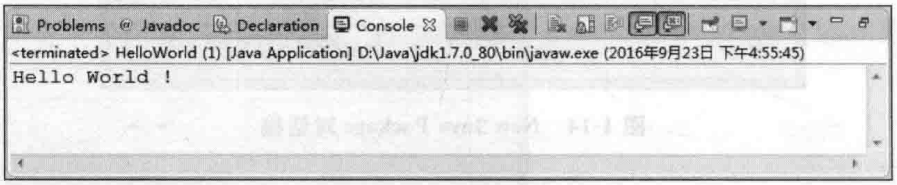


图 1-18 运行结果



多学一招：包的定义与使用

为了便于对硬盘上的文件进行管理，通常都会将文件分目录进行存放。同理，在程序开发中，也需要将编写的类在项目中分目录存放，以便于文件管理。为此，Java 引入了包（package）机制，程序可以通过声明包的方式对 Java 类定义目录。

Java 中的包是专门用于存放类的，通常功能相同的类存放在相同的包中。在声明包时，使用 package 语句，具体示例如下：

```
package com.itheima.example;          //使用 package 关键字声明包
public class Example01{…}
```

在实际程序开发过程中，定义的类都是含有包名的，如果没有特别声明 package 语句，则创建的类会处于默认包下，在实际开发中，这种情况是不应该出现的，本书的示例代码主要展现的是功能部分的代码，所以在大多数示例代码中没有为示例指定包名，但是在提供的源代码中，都已使用包名。

在开发时，一个项目可能会使用很多包，当一个包中的类需要调用另一个包中的类时，就需要使用 import 关键字引入需要的类。使用 import 可以在程序中一次导入某个指定包下的类，这样就不必在每次用到该类时都书写完整类名，从而简化了代码量。使用 import 关键字的具体格式如下所示：

```
import 包名.类名;
```

需要注意的是，import 通常出现在 package 语句之后，类定义之前。如果需要用到一个包中的许多类，则可以使用“import 包名. *；”导入该包下的所有类。

在 JDK 中，不同功能的类都放在不同的包中，其中 Java 的核心类主要放在 java 包及其子包下，Java 扩展的大部分类都放在 javax 包及其子包下。为了便于后面的学习，接下来简单介绍 Java 语言中的常用包。

- java.util：包含 Java 中大量工具类、集合类等，例如 Collections、List、Set 等。
- java.io：包含 Java 输入和输出有关的类和接口。

第 2 章

Java编程基础

学习任何一门语言，其语法结构都是关键，Java 基础语法部分包含变量、常量、运算符、结构语句以及数组等，本章将针对 Java 基础语法知识进行详细讲解。

2.1 Java 的基本语法

2.1.1 Java 代码的基本格式

在 Java 程序中，使用 class 关键字定义类，所有程序代码都放在该类中。class 关键字的前面还可以使用修饰符进行修饰，格式如下：

```
修饰符 class 类名{
    程序代码
}
```

在编写 Java 代码时，需要特别注意几个关键点。

- Java 语言是严格区分大小写的。在定义类时，不能将 class 写成 Class，否则编译会报错。
- 每条功能执行语句的最后都必须用分号结束。例如 System. out. println("hello");。
- Java 程序中一个连续的字符串不能分开在两行中书写。如果字符串太长，可以将长字符串分成两个字符串，然后用加号(+)将两个字符串连起来，再在加号(+)处断行。

为了便于理解，下面列出了格式规范的代码，具体如下：

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("hello"+
        "world");
    }
}
```

2.1.2 Java 中的注释

在编写代码时，为了使代码更容易理解，通常会在实现某种功能时添加一些注释予以说明，注释的内容不会被程序解析执行。Java 中的注释有三种不同的类型。

1. 单行注释

单行注释通常用于对程序中的某一行代码进行解释，用符号//表示，//后面为注释的内

容，具体示例如下：

```
int x= 1;        //定义一个整型变量 x
```

2. 多行注释

多行注释指注释的内容可以为多行，它以符号/ * 开头，以符号 * /结尾，具体示例如下：

```
/* int x=1;
   int y=2; */
```

3. 文档注释

文档注释通常用于对类或方法的说明，它以符号/ * * 开头，以符号 * /结尾。这种方式可以使用 Eclipse 工具将文档注释导出并生成帮助文档，方便他人使用，具体示例如下：

```
/**
 * Person 实体类
 */
```

2.1.3 Java 中的标识符

在实际编写程序的过程中，经常需要定义特殊符号标记一些名称，如包名、类名、变量名、方法名以及参数名等，这些符号被称为标识符。标识符可以由字母、数字、美元符号（$）和下画线（_）组成，但标识符不能以数字开头，也不可以使用 Java 中的关键字。

例如，合法的标识符有：

```
name, name1, _name
```

例如，不合法的标识符有：

```
1name, class, 100
```

在 Java 程序中定义的标识符必须严格遵守以上的规范，否则程序在编译时会报错。除了上面列出的规范以外，为了增强代码的可读性，建议初学者在定义标识符时还应该遵循以下规则。

（1）类名和接口名每个单词的首字母一律大写。例如：Cat、Animal。

（2）包名所有字母一律小写。例如：com. itheima. chapter02。

（3）常量名所有字母一律大写，单词之间用下画线连接。例如：MY_SCORE。

（4）变量名和方法名的第一个单词首字母小写，从第二个单词开始每个单词首字母大写。例如：studentName。

（5）在程序中，应该定义有实际意义的英文单词以定义标识符，使程序代码便于阅读，而且不能使用关键字定义标识符。

2.1.4 Java 中的关键字

关键字是编程语言中已定义好的并具有特殊含义的单词。Java 中预留了很多常用的

关键字，并且这些关键字都是小写的，表 2-1 列举出了 Java 中所有的关键字。

表 2-1 Java 中所有的关键字

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

2.2 Java 中的变量

2.2.1 变量的定义

在程序运行期间，随时可能产生一些临时数据，应用程序会将这些数据保存在一些内存单元中，每个内存单元都用一个标识符标识。这些内存单元被称为变量，定义的标识符就是变量名，内存单元中存储的数据就是变量的值。接下来，通过具体的代码学习变量的定义。

```
int x=3;
int y;
y=x-1;
```

上述代码中，分别定义了两个 int 类型变量 x 和 y，x 的初始值为 3，y 没有初始值，第三行代码中将 $x-1$ 的结果赋值给变量 y，此时 y 的值为 2。

2.2.2 变量的数据类型

在 Java 中变量的数据类型分为两种，即基本数据类型和引用数据类型。Java 中所有的数据类型如图 2-1 所示。

1. 整数类型变量

整数类型变量用于存储整数数值，整数类型分为四种：字节型(byte)、短整型(short)、整型(int)和长整型(long)。值得注意的是，long 类型的变量在赋值时超出了 int 类型的取值范围，需要在后面加上一个字母 L(或小写 l，但一般不建议使用小写 l，因为很难和数字 1 区分)。具体示例如下：

```
int num=3;                  //定义 int 类型的变量并赋值为 3
long num=2300000000L;       //所赋的值超出了 int 类型的取值范围，后面必须加上大写字母 L(或
                            //小写字母 l)
```

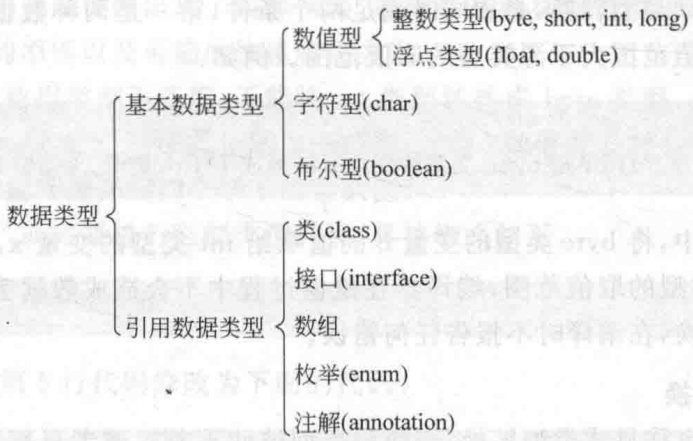


图 2-1　数据类型

2. 浮点数类型变量

浮点数类型变量用于存储小数数值。在 Java 中，浮点数类型分为两种：单精度浮点数(float)和双精度浮点数(double)，double 型所表示的浮点数比 float 型更精确。值得注意的是，float 类型变量赋值的后面需要加上字母 F(或者小写 f)。具体示例如下：

```
float f=111.2f;                //为一个 float 类型的变量赋值,后面必须加上字母 f
double d=111.2;                //为一个 double 类型的变量赋值,后面可以省略字母 d
```

3. 布尔类型变量

布尔类型变量用于存储布尔值，在 Java 中用 boolean 表示，该类型的变量只有两个值，即 true 和 false。具体示例如下：

```
boolean b=false;               //声明一个 boolean 类型的变量,初始值为 false
b=true;                        //改变 b 变量的值为 true
```

4. 字符类型变量

字符类型变量用于存储一个单一字符，在 Java 中用 char 类型表示。Java 中每个 char 类型的字符变量都会占用 2 个字节。在给 char 类型的变量赋值时，需要用一对英文半角格式的单引号(' ')把字符引起来，如'b'。具体示例如下：

```
char c='b';                    //为一个 char 类型的变量赋值字符 b
```

2.2.3　变量的类型转换

在程序中，当把一种数据类型的值赋给另一种数据类型的变量时，需要进行数据类型转换。根据转换方式的不同可分为两种类型：自动类型转换和强制类型转换。

1. 自动类型转换

自动类型转换(隐式类型转换)，指的是两种数据类型在转换的过程中不需要显式地进

行声明。要实现自动类型转换，必须同时满足两个条件，第一是两种数据类型彼此兼容，第二是目标类型的取值范围大于源类型的取值范围。例如：

```
byte b=1;
int i=b;          //程序把 byte 类型的变量 b 转换成了 int 类型，无须特殊声明
```

在上面的语句中，将 byte 类型的变量 b 的值赋给 int 类型的变量 x，由于 int 类型的取值范围大于 byte 类型的取值范围，编译器在赋值过程中不会造成数据丢失，所以编译器能够自动完成这种转换，在编译时不报告任何错误。

2. 强制类型转换

强制类型转换也称显式类型转换，当两种类型彼此不兼容或者目标类型的取值范围小于源类型时，将无法进行自动类型转换，此时就需要强制类型转换。下面通过案例演示如何进行强制类型转换。

首先创建一个 chapter02 项目，在该项目中创建一个 com. itheima. example 包。本章的 Java 类都放在该包下，类中的第一句代码就是包的声明，如 package com. itheima. example; 表示当前类位于 example 包中，读者根据这句代码就知道当前类位于哪个包中，然后在该包下创建一个 Example01 类，具体代码如文件 2-1 所示。

文件 2-1　Example01. java

```
1  package com.itheima.example;
2  public class Example01 {
3      public static void main(String[] args) {
4          int i=5;
5          byte b=i;
6          System.out.println(b);
7      }
8  }
```

编译程序报错，结果如图 2-2 所示。

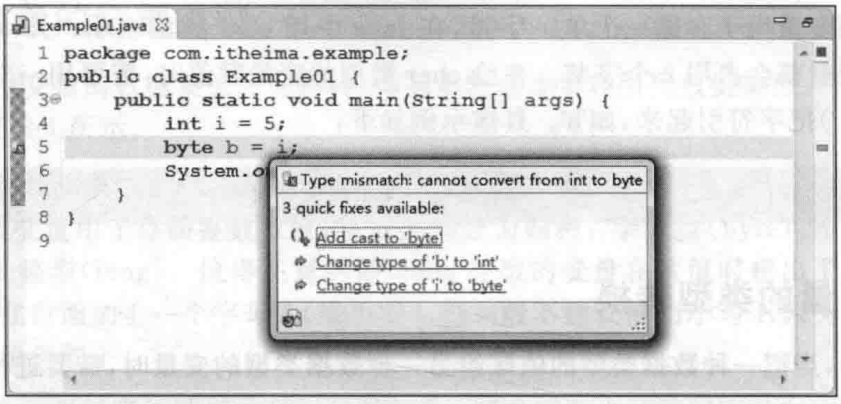


图 2-2　程序编译报错

从图 2-2 可以看出，Eclipse 在保存源文件时会自动对源代码进行编译，如果保存后出

现红色波浪线，说明此处代码出现了编译错误，将鼠标光标移到错误上方，会出现一个悬浮框，提示编译错误的原因以及可能的快速解决方案。

此处错误提示数据类型不匹配，不能将 int 类型转换成 byte 类型。出现该错误的原因是 int 类型的取值范围大于 byte 类型的取值范围，当将 i 赋值给 b 时会导致数值溢出，也就是说一个字节的变量无法存储四个字节的整数值。

在这种情况下，就需要进行强制类型转换，具体格式如下：

```
目标类型 变量=(目标类型)值
```

将文件 2-1 中第 5 行代码修改为下面的代码：

```
byte b=(byte) i;
```

修改后保存源文件，编译不会报错，程序运行结果如图 2-3 所示。

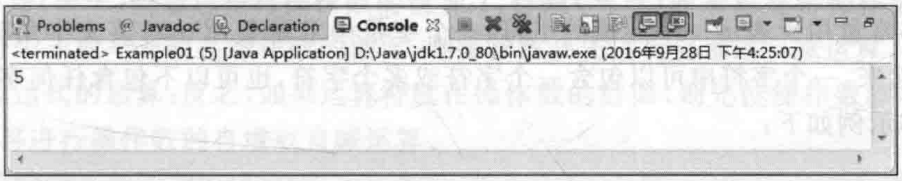


图 2-3　运行结果

2.2.4　变量的作用域

在程序中，变量一定会被定义在某一对大括号中，该大括号所包含的代码区域便是这个变量的作用域，变量只能在这个区域内使用。下面通过一个代码片段分析变量的作用域：

```
class Hello{
    int x=1;
    public static void main(String[] args)  {
        int y=2;
    }
}
```

y 的作用域

x 的作用域

在上述的代码中，有两层大括号。其中，外层大括号所标识的代码区域就是变量 x 的作用域，内层大括号所标识的代码区域就是变量 y 的作用域。

2.3　Java 中的常量

常量就是不能改变的数据，即在程序中固定不变的值，例如数字 1、字符'c'、浮点数 10.1 等。常量包括整型常量、浮点数常量、布尔常量、字符常量等。

1. 整型常量

整型常量是整数类型的数据，有二进制、八进制、十进制和十六进制四种表示形式。

2. 浮点数常量

浮点数常量其实就是数学中的小数，包括单精度浮点数 float 和双精度浮点数 double，其中单精度浮点数以 F 或者 f 结尾，双精度浮点数以 D 或 d 结尾，浮点数也可以不使用后缀，这样虚拟机会将其默认为双精度浮点数。具体示例如下：

```
1.2f  1F  1.23d  1.6
```

3. 字符常量

字符常量用于表示一个字符，一个字符常量要用一对英文半角格式的单引号(' ')引起来，它可以是标点符号、数字、英文字母以及由转义序列表示的特殊字符。具体示例如下：

```
'1'  'a'  '&'  '\n'
```

4. 字符串常量

字符串常量用于表示一串连续的字符，一个字符串常量要用一对英文半角格式的双引号(" ")引起来，一个字符串可以包含一个字符或多个字符，也可以不包含任何字符，即长度为零。具体示例如下：

```
"HelloJava"  "11"  "Hello \n XXX" ""
```

5. 布尔常量

布尔常量有两个值，分别为 true 和 false，该常量用于区分一个事物的真与假。

6. null 常量

null 常量只有一个值 null，表示对象的引用为空。

2.4 Java 中的运算符

运算符用于对数据进行算术运算、赋值和比较等操作。在 Java 中，运算符可分为算术运算符、赋值运算符、比较运算符以及逻辑运算符。

2.4.1 算术运算符

数学运算中的加、减、乘、除被称为四则运算，在 Java 中算术运算符就是用于处理四则运算的符号。表 2-2 给出了 Java 中的算术运算符及其用法。

表 2-2 算术运算符

运 算 符	运 算	范 例	结 果
+	正号	+3	3
−	负号	b=4;−b;	−4
+	加	5+5	10
−	减	6−4	2

续表

运 算 符	运　　算	范　　例	结　　果
*	乘	3 * 4	12
/	除	5/5	1
%	取模(即算术中的求余数)	7%5	2
++	自增(前)	a=2;b=++a;	a=3;b=3;
++	自增(后)	a=2;b=a++;	a=3;b=2;
--	自减(前)	a=2;b=--a	a=1;b=1;
--	自减(后)	a=2;b=a--	a=1;b=2;

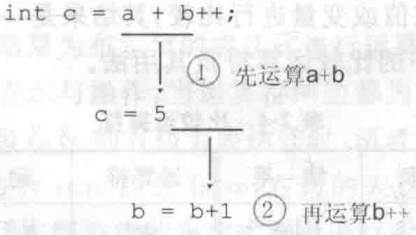
算术运算符看上去都比较简单,也很容易理解,但在实际使用时还需要注意两个常见问题。

(1) 自增(++)的意思是将操作数的值加 1,自减(--)的意思是将操作数的值减 1,如果运算符++或--放在操作数的前面,则操作数会先进行自增或自减运算,再使操作数参与其他表达式的运算;反之,如果运算符放在操作数的后面,则先使操作数参与其他表达式的运算,再进行操作数的自增或自减运算。

请仔细阅读下面的代码块,思考运行的结果。

```
int a=2;
int b=3;
int c=a+b++;
System.out.print("b="+b);
System.out.print("c="+c);
```

上面的代码块运行结果为 b=4,c=5,具体分析如下:



当进行 a+b++运算时,由于运算符++写在了变量 b 的后面,属于先运算再自增,因此变量 b 在参与加法运算时其值仍然为 3,而 c 的值应为 5。变量 b 在参与运算之后会进行自增,因此 b 的最终值为 4。

(2) 在进行除法运算时,当除数和被除数都为整数时,得到的结果也是一个整数。如果除法运算有小数参与,得到的结果会是一个小数。例如,3190/1000 属于整数之间相除,会忽略小数部分,得到的结果是 3,而 31.9/10 的结果为 3.19。

2.4.2　赋值运算符

赋值运算符的作用就是将常量、变量或表达式的值赋给某一个变量。表 2-3 给出了

Java 中的赋值运算符及其用法。

表 2-3 赋值运算符

运 算 符	运 算	范 例	结 果
=	赋值	a=3;b=2;	a=3;b=2;
+=	加等于	a=3;b=2;a+=b;	a=5;b=2;
-=	减等于	a=3;b=2;a-=b;	a=1;b=2;
*=	乘等于	a=3;b=2;a*=b;	a=6;b=2;
/=	除等于	a=3;b=2;a/=b;	a=1;b=2;
%=	模等于	a=3;b=2;a%=b;	a=1;b=2;

在赋值过程中，运算顺序从右至左，将右边表达式的结果赋值给左边的变量。在赋值运算符的使用中，需要注意以下两个问题。

(1) 在 Java 中可以通过一条赋值语句对多个变量进行赋值，具体示例如下：

```
int a, b, c;
a=b=c=3;                    //为三个变量同时赋值
```

在上述代码中，一条赋值语句将变量 a,b,c 的值同时赋值为 3。需要特别注意的是，不能直接写成 int a=b=c=3。

(2) 在表 2-3 中，除了=以外，其他的都是特殊的赋值运算符，以+=为例，x+=2 就相当于 x=x+2，首先会进行加法运算 x+2，再将运算结果赋值给变量 x。-=、*=、/=、%=赋值运算符都可以此类推。

2.4.3 比较运算符

比较运算符用于对两个数值或变量进行比较，其结果是一个布尔值，即 true 或者 false。接下来通过表 2-4 列出 Java 中的比较运算符及其用法。

表 2-4 比较运算符

运算符	运 算	范 例	结 果	运算符	运 算	范 例	结 果
==	相等于	4==3	false	>	大于	4>3	true
!=	不等于	4!=3	true	<=	小于等于	4<=3	false
<	小于	4<3	false	>=	大于等于	4 >= 3	true

需要注意的是，==和=是两个不同的运算符，==是比较运算符，=是赋值运算符。

2.4.4 逻辑运算符

逻辑运算符用于对布尔型的数据进行操作，其结果仍是一个布尔值。表 2-5 给出了 Java 中的逻辑运算符及其用法。

在使用逻辑运算符的过程中，需要注意以下几个细节。

表 2-5　逻辑运算符

运　算　符	运　　算	范　　例	结　　果
&	与	true & true	true
		true & false	false
		false & false	false
		false &true	false
|	或	true | true	true
		true | false	true
		false| false	false
		false| true	true
^	异或	true^ true	false
		true^ false	true
		false^ false	false
		false^ true	true
!	非	!true	false
		!false	true
&&	短路与	true && true	true
		true && false	false
		false && false	false
		false && true	false
||	短路或	true || true	true
		true || false	true
		false|| false	false
		false|| true	true

(1) 逻辑运算符可以针对结果为布尔值的表达式进行运算，如 x>1&&y!=1。

(2) 运算符 & 和 && 都表示与操作，当运算符两边都为 true 时，其结果才为 true，否则结果为 false。当运算符 & 和 && 的右边为表达式时，两者在使用上还有一定的区别，在使用 & 进行运算时，无论左边为 true 还是 false，右边的表达式都会进行运算；如果使用 && 进行运算，当左边为 false 时，则右边的表达式不会进行运算，而是直接返回 false，因此 && 被称为短路与。

(3) 运算符|和||都表示或操作，当运算符两边的操作数任何一边的值为 true 时，其结果为 true，当两边的值都为 false 时，其结果才为 false。两者所不同的是，当运算符||的左边为 true 时，右边的表达式不会进行运算，而是直接返回 true，因此||被称为短路或。下面通过一个示例演示短路或的使用，具体如下：

```
int x=1;
int y=1;
boolean b=x==1||y++>0;
```

上面的代码块执行完毕后，b 的值为 true，y 的值仍为 1。出现该结果的原因是，运算符“||”的左边 x==1 结果为 true，那么右边表达式将不会进行运算，y 的值不发生任何变化。

(4) 运算符^表示异或操作，当运算符两边的布尔值相同时(都为 true 或都为 false)，其结果为 false；当两边布尔值不相同时，其结果为 true。

2.4.5 运算符的优先级

在对一些比较复杂的表达式进行运算时，要明确表达式中所有运算符参与运算的先后顺序，通常把这种顺序称为运算符的优先级。表 2-6 列出了 Java 中运算符的优先级，数字越小优先级越高。

表 2-6 运算符优先级

优 先 级	运 算 符
1	. [] ()
2	++ -- ~ ! (数据类型)
3	* / %
4	+ -
5	<< >> >>>
6	< > <= >=
7	== !=
8	&
9	^
10	|
11	&&
12	||
13	?:
14	= *= /= %= += -= <<= >>= >>>= &= ^= |=

根据表 2-5 所示的运算符优先级，分析下面代码的运行结果。

```
int a=3;
int b=(a+2) * a;
System.out.println(b);
```

运行结果为 15，由于运算符()的优先级最高，因此先运算括号内的 a+2，得到的结果是 5，再将 5 与 a 相乘，得到最后的结果为 15。

2.5 选择结构语句

2.5.1 if 条件语句

if 条件语句是指一种表示假设的主从复合句，由 if 语句引导条件(条件分为成立和不成

立两种情况），通过对条件是否成立的判断，执行不同的代码。if 语句实际上有三种形式，每种格式都有其自身的特点。

1. if 语句

if 语句表示如果条件成立，则执行某个操作。例如，前方如果是红灯，则需要停车等候。if 条件语句的语法格式如下：

```
if (条件语句){
    代码块
}
```

上述格式中，判断条件是一个布尔值，当判断条件为 true 时，{}中的执行语句才会被执行。if 条件语句的执行流程如图 2-4 所示。

2. if...else 语句

if...else 条件语句是指如果满足某种条件，则进行某种处理，否则进行另一种处理。if...else 语句具体语法格式如下：

```
if (判断条件){
    执行语句 1
    ⋮
}else{
    执行语句 2
    ⋮
}
```

上述格式中，判断条件是一个布尔值。当判断条件为 true 时，执行语句 1 会被执行；当判断条件为 false 时，执行语句 2 会被执行。if...else 语句的执行流程如图 2-5 所示。

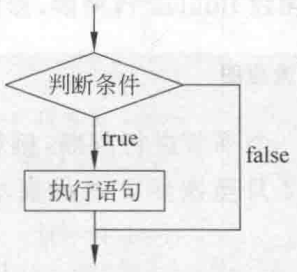


图 2-4　if 语句流程图

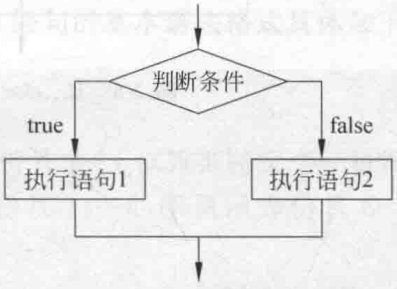


图 2-5　if...else 语句流程图

3. if...else if...else 语句

if...else if...else 语句用于对多个条件进行判断，进行多种不同的处理，具体语法格式如下：

```
if (判断条件 1) {
    执行语句 1
} else if (判断条件 2) {
```

```
    执行语句 2
}
⋮
else if (判断条件 n) {
    执行语句 n
} else {
    执行语句 n+1
}
```

上述格式中，判断条件是一个布尔值。当判断条件 1 为 true 时，if 后面{}中的执行语句 1 会被执行；当判断条件 1 为 false 时，会继续执行判断条件 2，如果为 true 则执行语句 2，以此类推；如果所有的判断条件都为 false，则意味着所有条件均未满足，else 后面{}中的执行语句 n+1 会被执行。if...else if...else 语句的执行流程如图 2-6 所示。

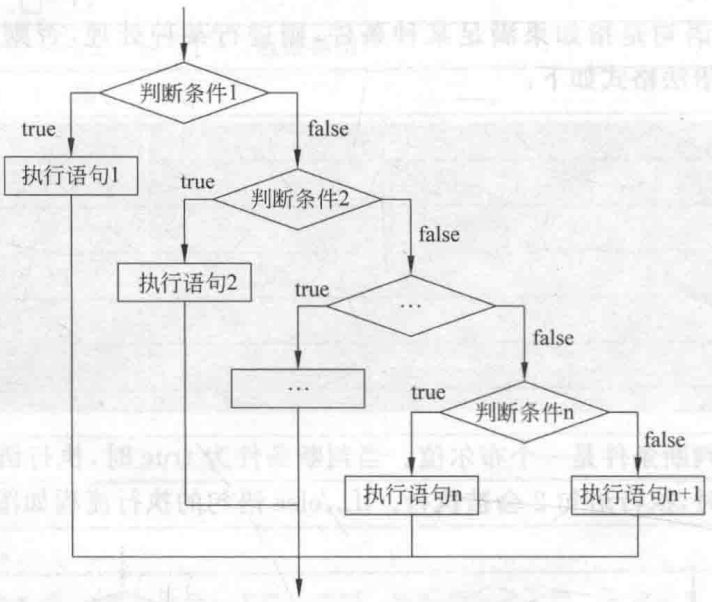


图 2-6 if...else if...else 语句的流程图

接下来通过一个案例实现对 12 个月份分别属于哪一个季节进行判断，例如 3—5 月份表示春季，6—8 月份表示夏季，9—11 月份表示秋季，12 月至次年 2 月份表示冬季，如文件 2-2 所示。

文件 2-2 Example02.java

```
1  package com.itheima.example;
2  public class Example02 {
3      public static void main(String[] args) {
4          int month=1;                          //定义月份
5          if (month >=3 && month <=5) {
6              System.out.println("春季");
7          } else if (month >=6 && month <=8 ) {
8              System.out.println("夏季");
```

```
9           } else if ( month >=9 && month <=11) {
10              System.out.println("秋季");
11          } else if(month==1||month==2||month==12 ){
12              System.out.println("冬季");
13          }else {
14              System.out.println("输入错误");
15          }
16      }
17  }
```

运行结果如图 2-7 所示。

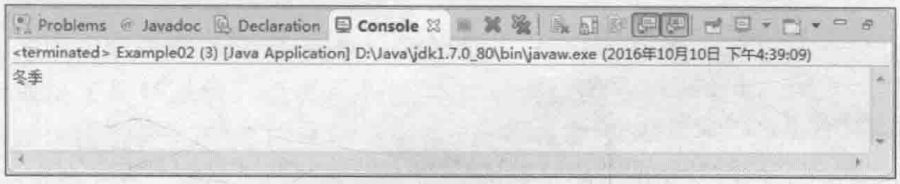


图 2-7　运行结果

文件 2-2 中，定义了月份 month 为 1。满足第三个判断条件 month ==1||month==2||month==12，因此会显示“冬季”。

2.5.2　switch 条件语句

switch 语句与 if 语句类似，同样用于条件判断。不同的是，if 语句通常用于对复合条件进行判断，switch 语句只能针对某个表达式进行判断。

switch 语句可以拥有多条 case 语句，每条 case 语句后面加一个要比较的目标值，当 switch 中表达式的值与某个 case 的目标值匹配时，会执行对应 case 下的语句，如果未找到任何匹配的值，则执行 default 后的语句，switch 语句的基本语法格式具体如下：

```
switch (表达式){
    case 目标值 1:
        执行语句 1
        break;
    case 目标值 2:
        执行语句 2
        break;
        ⋮
    case 目标值 n:
        执行语句 n
        break;
    default:
        执行语句 n+1
        break;
}
```

需要注意的是，在 JDK 5.0 之前，switch 语句中的表达式只能是 byte、short、char、int 类型的值，如果输入其他类型的值，程序会报错。在 JDK 5.0 中，引入的新特性 enum 枚举可以作为 switch 语句表达式的值，在 JDK 7.0 中也引入了新特性，switch 语句可以接收一个 String 类型的值。下面通过案例演示 switch 语句的用法，如文件 2-3 所示。

文件 2-3 Example03.java

```
1  package com.itheima.example;
2  public class Example03 {
3      public static void main(String[] args) {
4          String country="中国";
5          switch (country) {
6          case "中国":
7              System.out.println("你好: "+country);
8              break;
9          case "美国":
10             System.out.println("Hello: "+country);
11             break;
12         case "韩国":
13             System.out.println("안녕하세요: "+country);
14             break;
15         default:
16             System.out.println("输入错误");
17         }
18     }
19 }
```

运行结果如图 2-8 所示。

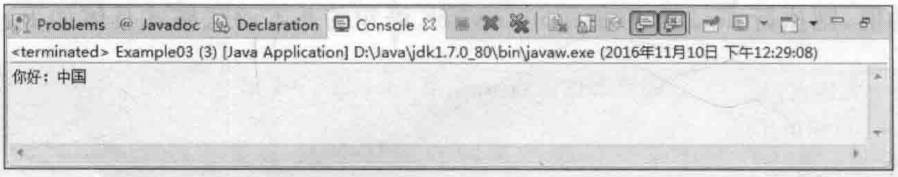


图 2-8 运行结果

在文件 2-3 中，由于变量 country 的值为“中国”，switch 语句判断的结果满足文件 2-3 中第 6 行代码的条件，因此显示“你好：中国”，文件中的 default 语句用于处理和前面的 case 都不匹配的值，将第 4 行代码替换为 String country ="德国"，再次运行程序，输出结果如图 2-9 所示。

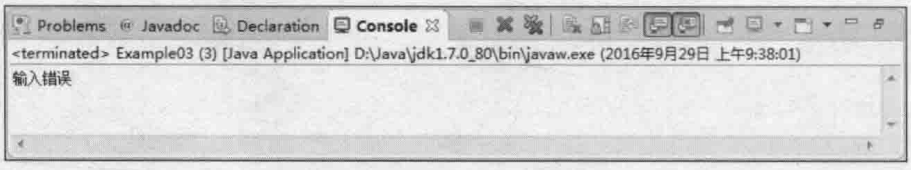


图 2-9 运行结果

2.6　循环结构语句

2.6.1　while 循环语句

while 语句会反复进行条件判断，只要条件成立，就会执行{}内的语句，直到条件不成立，while 循环结束。while 循环语句的语法结构如下：

```
while(循环条件){
    执行语句
}
```

在上面的语法结构中，{}中的执行语句被称为循环体，循环体是否执行取决于循环条件。当循环条件为 true 时，循环体就会执行。循环体执行完毕时循环条件会继续判断，如果条件仍为 true，则会继续执行，直到循环条件为 false 时，整个循环过程才会结束。

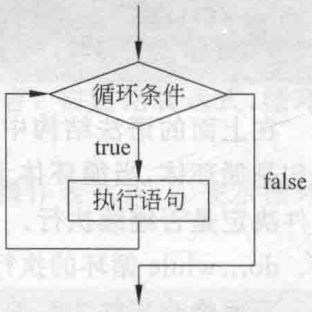


图 2-10　while 循环的流程图

while 循环的执行流程如图 2-10 所示。

下面通过一个案例演示如何使用 while 循环对 1～10 进行求和，如文件 2-4 所示。

文件 2-4　Example04.java

```
1   package com.itheima.example;
2   public class Example04 {
3       public static void main(String[] args) {
4           int i=1;                        //定义循环变量 i
5           int sum=0;                      //定义变量 sum,用于记住累加的和
6           while (i <=10) {
7               sum=sum+i;                  //实现 sum 与 i 的累加
8               i++;
9           }
10          System.out.println("sum="+sum);      //打印累加的和
11      }
12  }
```

运行结果如图 2-11 所示。

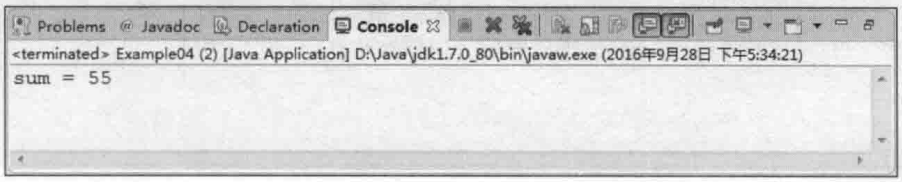


图 2-11　运行结果

在文件 2-4 中，分别初始化变量 i 和变量 sum，接着判断条件 i<=10 是否为 true，如果为 true 则执行循环体 sum+=i 语句，然后执行操作表达式 i++，i 的值变为 2，然后继续进行条件判断，开始下一次循环，直到 i=11 时，条件 i<=10 为 false，跳出循环，执行 while 循环后面的代码，最后输出结果为 sum=55。

2.6.2 do...while 循环语句

do...while 循环语句和 while 循环语句功能类似，不同的是，do…while 循环语句将循环条件放在了循环体的后面，这也就意味着，无论循环条件是否成立，循环体都会无条件执行一次，其语法结构如下：

```
do {
    执行语句
} while(循环条件);
```

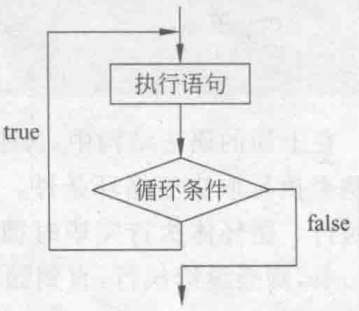


图 2-12 do...while 循环的执行流程

在上面的语法结构中，关键字 do 后面的{}中的执行语句是循环体，当循环体无条件执行一次后，再根据循环条件决定是否继续执行。

do...while 循环的执行流程如图 2-12 所示。

下面修改文件 2-4，使用 do...while 循环对 1～10 进行求和，如文件 2-5 所示。

文件 2-5 Example05.java

```
1  package com.itheima.example;
2  public class Example05 {
3      public static void main(String[] args) {
4          int i=1;                                    //定义循环变量 i
5          int sum=0;                                  //定义变量 sum,用于记住累加的和
6          do {
7              sum=sum+i;                              //实现 sum 与 i 的累加
8              i++;
9          }while(i <=10);
10         System.out.println("sum="+sum);             //打印累加的和
11     }
12 }
```

运行结果如图 2-13 所示。

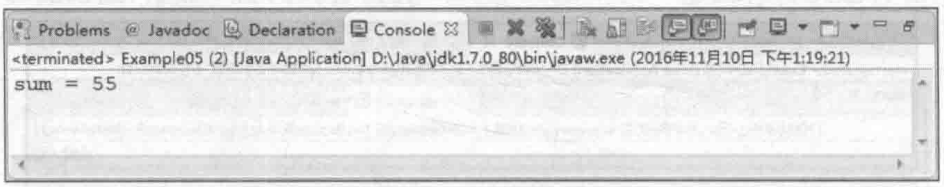


图 2-13 运行结果

图 2-13 与图 2-11 的运行结果一致，说明通过 do...while 循环同样实现了对 1～10 进行求和。不同的是，如果循环条件不成立，那么 while 语句的循环体不会执行，而 do...while 还会执行一次。若将两个案例中的循环条件 i<=10 改为 i<1，那么文件 2-5 会显示 sum=1，而文件 2-4 什么都不会显示。

2.6.3　for 循环语句

for 循环语句是最常用的循环语句，一般用于循环次数已知的情况下。for 循环语句的语法格式如下：

```
for(初始化表达式; 循环条件; 操作表达式){
    执行语句
}
```

在上面的语法结构中，for 关键字后面的()中包括了三部分内容：初始化表达式、循环条件和操作表达式，它们之间用(;)分隔，{}中的执行语句为循环体。

接下来分别用①表示初始化表达式，②表示循环条件，③表示操作表达式，④表示循环体，通过序号具体分析 for 循环的执行流程。具体如下：

```
for(① ; ② ; ③){
    ④
}
第一步，执行①
第二步，执行②，如果判断结果为 true，执行第三步；如果判断结果为 false，执行第五步
第三步，执行④
第四步，执行③，然后重复执行第二步
第五步，退出循环
```

接下来再次修改文件 2-4，使用 for 循环对 1～10 进行求和，如文件 2-6 所示。

文件 2-6　Example06.java

```
1  package com.itheima.example;
2  public class Example06 {
3      public static void main(String[] args) {
4          int sum=0;                          //定义变量 sum,用于记住累加的和
5          for (int i=1; i <=10; i++) {        //i 的值会在 1~10 之间变化
6              sum=sum+i;                      //实现 sum 与 i 的累加
7          }
8          System.out.println("sum="+sum); //打印累加的和
9      }
10 }
```

运行结果如图 2-14 所示。

图 2-14 与图 2-11 的运行结果一致，说明通过 for 循环同样实现了对 1～10 进行求和。

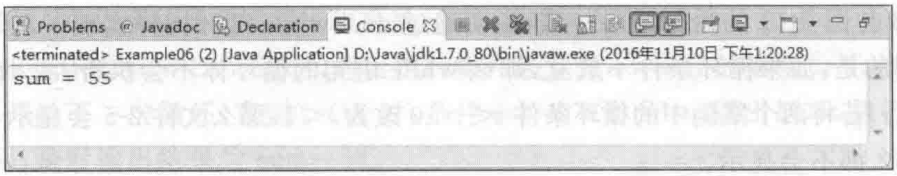


图 2-14 运行结果

2.6.4 循环嵌套

嵌套循环是指在一条循环语句的循环体中再定义一条循环语句的语法结构。while、do...while、for 循环语句都可以进行嵌套，并且它们之间也可以互相嵌套，其中最常见的是在 for 循环中嵌套 for 循环，格式如下：

```
for(初始化表达式; 循环条件; 操作表达式) {
    for(初始化表达式; 循环条件; 操作表达式) {
        执行语句
    }
}
```

正同通过一个案例实现使用 # 输出直角三角形，如文件 2-7 所示。

文件 2-7 Example07.java

```
1  package com.itheima.example;
2  public class Example07 {
3      public static void main(String[] args) {
4          int i, j;                                    //定义两个循环变量
5          for (i=1; i <=8; i++) {                      //外层循环
6              for (j=1; j <=i; j++) {                  //内层循环
7                  System.out.print("#");               //打印#
8              }
9              System.out.print("\n");                  //换行
10         }
11     }
12 }
```

运行结果如图 2-15 所示。

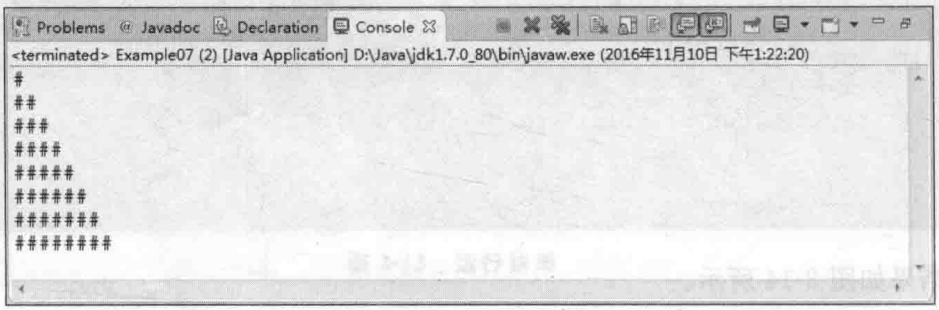


图 2-15 运行结果

文件 2-7 定义了两层 for 循环，分别为外层循环和内层循环，外层循环控制打印的行数，内层循环控制打印“#”，外层循环每执行一次，内层循环相比上一次循环将会多执行一次，所以每一行 # 的个数会逐行增加一个，最后输出一个直角三角形的图形。

2.6.5　跳转语句

跳转语句用于实现循环执行过程中程序流程的跳转，在 Java 中的跳转语句有 break 语句和 continue 语句，下面分别进行详细讲解。

1. break 语句

在 switch 条件语句和循环语句中都可以使用 break 语句。当 break 语句出现在 switch 条件语句中时，其作用是终止某个 case 并向下执行；当 break 语句出现在循环语句中时，其作用是跳出循环语句，执行后面的代码。下面通过案例实现输出 1～5 之间的自然数，当值为 4 时，使用 break 语句跳出循环，如文件 2-8 所示。

文件 2-8　Example08.java

```
1  package com.itheima.example;
2  public class Example08 {
3      public static void main(String[] args) {
4          int x=1;                                  //定义变量 x,初始值为 1
5          while (x <=5) {                           //循环条件
6              System.out.println("x="+x);           //条件成立,打印 x 的值
7              if (x==4) {
8                  break;
9              }
10             x++;                                  //x 进行自增
11         }
12     }
13 }
```

运行结果如图 2-16 所示。

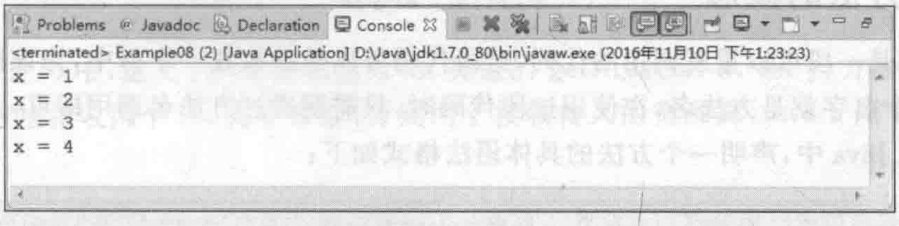


图 2-16　运行结果

在文件 2-8 中，通过 while 循环输出 x 的值，当 x 的值为 4 时，使用 break 语句跳出循环。因此运行结果中并没有出现 x=5。

2. continue 语句

continue 语句用于循环语句中，它的作用是终止本次循环，执行下一次循环。下面通过案例对 1～100 之内的偶数进行求和，如文件 2-9 所示。

文件 2-9 Example09.java

```
1  package com.itheima.example;
2  public class Example09 {
3      public static void main(String[] args) {
4          int sum=0;                          //定义变量 sum,用于记住累加的和
5          for (int i=1; i <=100; i++) {
6              if (i %2 !=0) {                 //i 是一个奇数,不累加
7                  continue;                   //终止本次循环,执行下一次循环
8              }
9              sum+=i;                         //实现 sum 和 i 的累加
10         }
11         System.out.println("sum="+sum);
12     }
13 }
```

运行结果如图 2-17 所示。

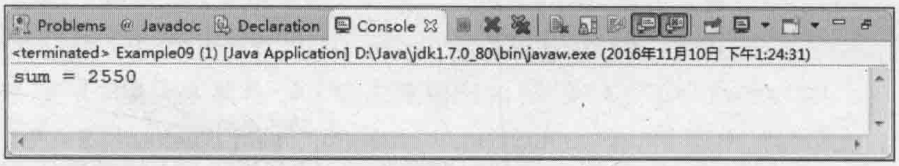


图 2-17 运行结果

在文件 2-9 中，使用 for 循环让变量 i 的值在 1～100 之间循环，在循环过程中，当 i 的值为奇数时，执行 continue 语句终止本次循环，进入下一次循环；当 i 的值为偶数时，sum 和 i 进行累加，最终得到 1～100 之间所有偶数的和，输出 sum=2550。

2.7 方法

2.7.1 方法的概念

方法就是一段可以重复调用的代码，这段代码可以单独放在一个{}中，为该段代码起一个名字，这个名字就是方法名，在使用这段代码时，只需要通过方法名调用即可，无须重复编写代码。在 Java 中，声明一个方法的具体语法格式如下：

```
修饰符 返回值类型 方法名([参数类型 参数名 1,参数类型 参数名 2,...]){
    执行语句
    return 返回值;
}
```

对于上面的语法格式的具体说明如下。

- 修饰符：方法的修饰符比较多，可对访问权限进行限定。
- 参数类型：用于限定调用方法时传入参数的数据类型。

- 参数名：是一个变量，用于接收调用方法时传入的数据。
- 返回值类型：用于限定方法返回值的数据类型。
- return 关键字：用于结束方法以及返回方法指定类型的值。
- 返回值：被 return 语句返回的值，该值会返回给调用者。

需要特别注意的是，方法中的“参数类型 参数名 1，参数类型 参数名 2”被称为参数列表，用于描述方法在被调用时需要接收的参数，如果方法不需要接收任何参数，则参数列表为空，即()内不写任何内容。方法的返回值必须为方法声明的返回值类型，如果方法中没有返回值，则返回值类型要声明为 void，此时，方法中的 return 语句可以省略。

下面通过一个案例演示方法的基本使用，如文件 2-10 所示。

文件 2-10　Example10.java

```
1  package com.itheima.example;
2  public class Example10 {
3      public static void main(String[] args) {
4          int area=getArea(4,5);          //调用 getArea 方法
5          System.out.println("矩形的面积为："+area);
6      }
7      //下面定义了一个求矩形面积的方法，接收两个参数，其中 x 为长，y 为宽
8      public static int getArea(int x, int y) {
9          int temp=x * y;                 //使用变量 temp 记住运算结果
10         return temp;                    //将变量 temp 的值返回
11     }
12 }
```

运行结果如图 2-18 所示。

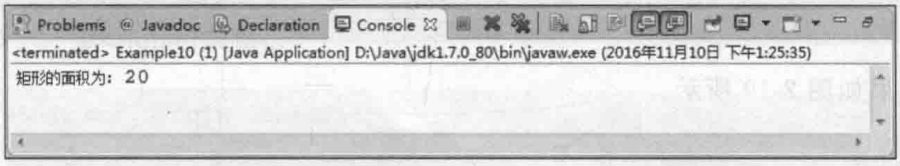


图 2-18　运行结果

在文件 2-10 中，定义了一个返回值为 int 类型的 getArea(int x, int y)方法，用于求矩形面积，该方法接收两个 int 类型参数，分别用于接收传入的宽和高，return 用于将结果返回给调用者。

2.7.2　方法的重载

假设在程序中定义了一个对数字进行求和的方法，但由于参与求和的数字个数和类型都不确定，因此要针对不同的情况设计不同的方法，如果每个方法的名称都不相同，那么在调用时就很难区分应该调用哪个方法。为此，我们可以使用方法重载，所谓的方法重载就是在一个程序中可以定义多个同名方法，但要求每个方法具有不同的参数类型或参数个数，下面通过一个使用方法重载形式的案例实现求和功能，如文件 2-11 所示。

文件 2-11 Example11.java

```
1  package com.itheima.example;
2  public class Example11 {
3      public static void main(String[] args) {
4          //下面是针对求和方法的调用
5          int sum1=add(2, 3);
6          int sum2=add(2, 3, 4);
7          double sum3=add(2.5, 3.5);
8          //下面的代码是打印求和的结果
9          System.out.println("sum1="+sum1);
10         System.out.println("sum2="+sum2);
11         System.out.println("sum3="+sum3);
12     }
13     //下面的方法实现了两个整数相加
14     public static int add(int x, int y) {
15         return x+y;
16     }
17     //下面的方法实现了三个整数相加
18     public static int add(int x, int y, int z) {
19         return x+y+z;
20     }
21     //下面的方法实现了两个小数相加
22     public static double add(double x, double y) {
23         return x+y;
24     }
25 }
```

运行结果如图 2-19 所示。

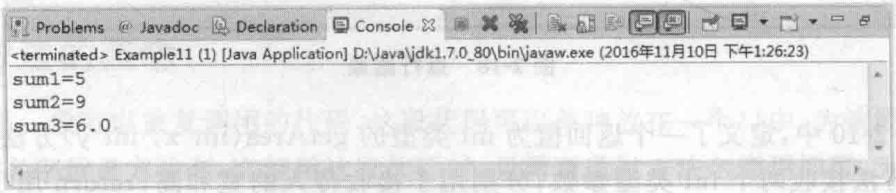


图 2-19 运行结果

在文件 2-11 中，定义了三个同名的 add()方法，但它们的参数个数或类型不同，从而形成了方法的重载。在 main()方法中调用 add()方法时，通过传入不同的参数便可以确定调用哪个重载的方法，如 add(2,3)调用的是两个整数求和的方法。值得注意的是，方法的重载与返回值类型无关，它需要满足两个条件：第一是方法名相同，第二是参数个数或参数类型不相同。

2.8 数组

2.8.1 数组的定义

数组是指一组数据的集合，数组中的每个数据被称为元素。数组可以存放任意类型的元素，但同一个数组中存放的元素类型必须一致，例如使用一个 int 类型数组保存某个学生的各科成绩，这样就避免了定义多个变量保存成绩的麻烦。

在 Java 中，可以使用以下格式定义一个数组：

```
int[] x=new int[10];
```

或

```
int[] x;
x=new int[10];
```

上述语句定义了一个 int 类型数组，[]中的数字 10 表示数组的长度，即可以存放 10 个元素，相当于在内存中定义了 10 个 int 类型的变量，第一个变量的名称为 x[0]，第二个变量的名称为 x[1]，以此类推，第 10 个变量的名称为 x[9]，这些变量的初始值都是 0。

每个数组元素都有一个索引(也称角标)，通过这个索引可以访问数组中的元素，例如要访问第 1 个元素，则可以通过 x[0]的形式。需要注意的是，数组的索引最小为 0，最大为“数组的长度－1”，获取数组长度可以通过 length 属性，在程序中通过“数组名. length”即可获取数组长度。

下面通过案例学习数组的基本使用，如文件 2-12 所示。

文件 2-12　Example12. java

```
1  package com.itheima.example;
2  public class Example12 {
3      public static void main(String[] args) {
4          int[] arr=new int[4];          //定义可以存储 4 个元素的整数类型数组
5          arr[0]=2;                      //为第 1 个元素赋值 2
6          arr[1]=3;                      //为第 2 个元素赋值 3
7          //依次打印数组中每个元素的值
8          System.out.println("arr[0]="+arr[0]);
9          System.out.println("arr[1]="+arr[1]);
10         System.out.println("arr[2]="+arr[2]);
11         System.out.println("arr[3]="+arr[3]);
12         System.out.println("数组的长度是: "+arr.length);
13     }
14 }
```

运行结果如图 2-20 所示。

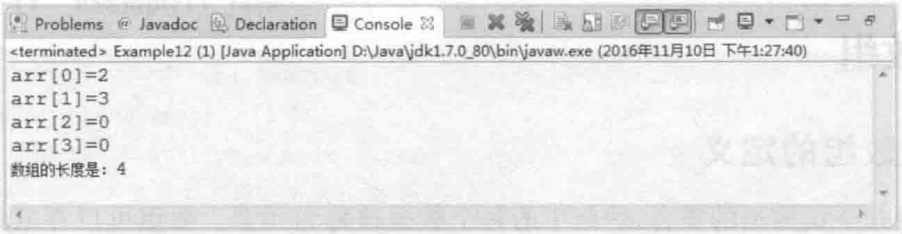


图 2-20 运行结果

在文件 2-12 中，第 4 行代码创建了一个长度为 4 的数组，并将数组在内存中的地址赋值给变量 arr。在第 5～6 行代码中，通过角标访问数组中的元素，第 12 行代码通过 length 属性获取数组的长度。

2.8.2 数组的常见操作

1. 数组遍历

数组的遍历指依次访问数组中的每个元素，下面通过案例学习如何使用 for 循环遍历数组，如文件 2-13 所示。

文件 2-13 Example13.java

```
1  package com.itheima.example;
2  public class Example13 {
3      public static void main(String[] args) {
4          int[] arr={ 3, 5, 1, 2, 6 };                    //定义数组
5          //使用 for 循环遍历数组的元素
6          for (int i=0; i <arr.length; i++) {
7              System.out.println(arr[i]);                 //通过索引获取元素
8          }
9      }
10 }
```

运行结果如图 2-21 所示。

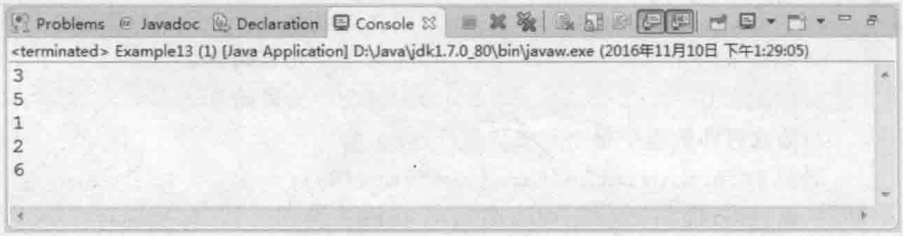


图 2-21 运行结果

在文件 2-13 中，定义了一个长度为 5 的数组 arr，通过 for 循环中定义的变量 i 的值在循环过程中依次访问数组中的每个元素，并将元素的值输出到控制台。

2. 数组最值

在操作数组时，经常需要获取数组中元素的最值。下面通过案例演示如何获取数组中

的最大元素，如文件 2-14 所示。

文件 2-14　Example14.java

```
1  package com.itheima.example;
2  public class Example14 {
3      public static void main(String[] args) {
4          int[] arr={ 4, 8, 3, 9, 7, 5 };          //定义一个数组
5          int max=getMax(arr);                     //调用获取元素最大值的方法
6          System.out.println("max="+max);          //打印最大值
7      }
8      static int getMax(int[] arr) {
9          int max=arr[0]; //定义变量 max 用于记住最大元素，首先假设第一个元素为最大值
10         //下面通过一个 for 循环遍历数组中的元素
11         for (int x=1; x <arr.length; x++) {
12             if (arr[x] >max) {                   //比较 arr[x]的值是否大于 max
13                 max=arr[x];                      //条件成立，将 arr[x]的值赋给 max
14             }
15         }
16         return max;                              //返回最大值 max
17     }
18 }
```

运行结果如图 2-22 所示。

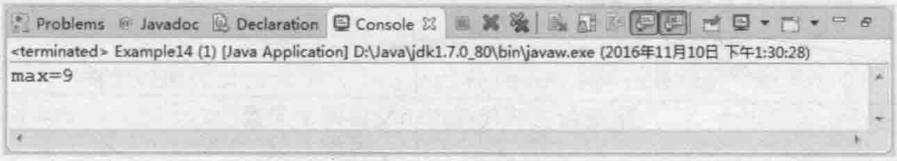


图 2-22　运行结果

在文件 2-14 中，定义了一个临时变量 max，用于记住数组中的最大元素。假设数组中第一个元素 arr[0]为数组的最大元素，然后使用 for 循环对数组进行遍历，在遍历的过程中与 max 进行比较，如果该元素大于 max，则将该元素赋值给 max。因此，变量 max 就能够在循环结束时记住数组中的最大元素。

3. 数组排序

数组排序是一个很常见的操作，通常使用冒泡排序实现这个功能。所谓的冒泡排序就是模仿水中气泡的上升过程，大的气泡向下沉，小的气泡向上浮。在数组中就是相邻的两个元素依次比较，如果前一个元素比后一个元素大，则交换位置，直到最后两个元素比较完成，则最后一个元素就是数组中的最大值，此时也就完成了第一轮比较。以此类推，除了最大的元素以外，将剩余元素继续两两比较完成后，将数组中第二大的数放在倒数第二的位置上。重复上述步骤，直到没有可比较的元素为止，下面通过案例演示冒泡排序的过程，如文件 2-15 所示。

文件 2-15 Example15.java

```
1  package com.itheima.example;
2  public class Example15 {
3      public static void main(String[] args) {
4          int[] arr={ 4, 8, 3, 9, 7, 5 };
5          System.out.print("排序前 : ");
6          printArray(arr);                          //打印数组元素
7          bubbleSort(arr);                          //调用排序方法
8          System.out.print("排序后 : ");
9          printArray(arr);                          //打印数组元素
10     }
11     //定义打印数组元素的方法
12     public static void printArray(int[] arr) {
13         //循环遍历数组的元素
14         for (int i=0; i <arr.length; i++) {
15             System.out.print(arr[i]+" ");         //打印元素和空格
16         }
17         System.out.print("\n");
18     }
19     //定义对数组排序的方法
20     public static void bubbleSort(int[] arr) {
21         //定义外层循环
22         for (int i=0; i <arr.length-1; i++) {
23             //定义内层循环
24             for (int j=0; j <arr.length-i-1; j++) {
25                 if (arr[j] >arr[j+1]) {           //比较相邻元素
26                     //下面的三行代码用于交换两个元素
27                     int temp=arr[j];
28                     arr[j]=arr[j+1];
29                     arr[j+1]=temp;
30                 }
31             }
32             System.out.print("第"+(i+1)+"轮排序后: ");
33             printArray(arr);                      //每轮比较结束打印数组元素
34         }
35     }
36 }
```

运行结果如图 2-23 所示。

在文件 2-15 中，bubbleSort()方法通过一个嵌套 for 循环实现了冒泡排序。其中，外层循环用于控制进行多少轮比较，每一轮比较都可以确定一个元素的位置，由于最后一个元素不需要进行比较，因此外层循环的次数为数组长度－1。内层循环的循环变量用于控制每一轮比较的次数，它被当作角标比较数组的元素，由于变量在循环过程中是自增的，这样就可以实现相邻元素依次进行比较，在每次比较时如果前者小于后者，就交换两个元素的位置。

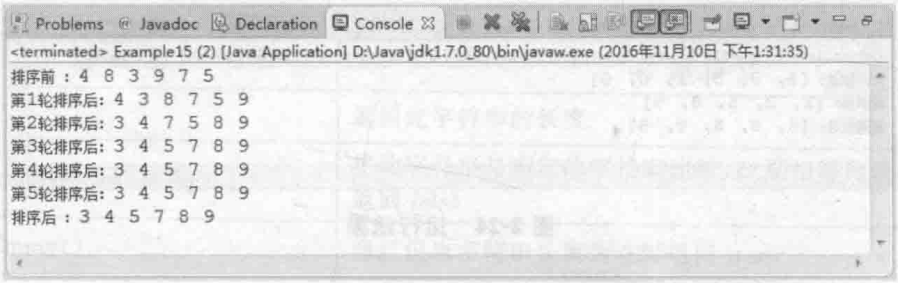


图 2-23　运行结果

直到外层循环结束，数组中的元素也就完成了排序。

2.8.3　Arrays 工具类

Arrays 是一个专门用于操作数组的工具类，该类位于 java.util 包中。Arrays 工具类提供了大量的静态方法，常用的方法如表 2-7 所示。

表 2-7　Arrays 工具类的常用方法

方法声明	功能描述
static void sort(int[] a)	对指定的 int 型数组按数字升序进行排序
static int binarySearch(Object[] a, Object key)	使用二分搜索法搜索指定数组，以获得指定对象
static int[] copyOfRange(int[] original, int from, int to)	将指定数组的指定范围复制到一个新数组
static void fill(Object[] a, Object val)	将指定的 Object 引用分配给指定 Object 数组的每个元素
static String toString(int[] arr)	返回指定数组内容的字符串表示形式

下面通过案例学习 Arrays 工具类的使用，如文件 2-16 所示。

文件 2-16　Example16.java

```
1  package com.itheima.example;
2  import java.util.*;
3  public class Example16 {
4      public static void main(String[] args) {
5          int[] arr={ 9, 8, 3, 5, 2 };
6          int[] copied=Arrays.copyOfRange(arr, 1, 7);          //复制数组
7          System.out.println("复制数组："+Arrays.toString(copied));
8          Arrays.sort(arr);                //排序数组
9          System.out.println("排序后："+Arrays.toString(arr));
10         Arrays.fill(arr, 8);             //填充数组
11         System.out.println("填充数组："+Arrays.toString(arr));
12     }
13 }
```

运行结果如图 2-24 所示。

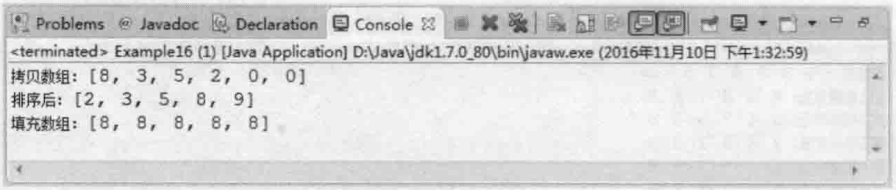


图 2-24 运行结果

在文件 2-16 中，在分别使用 Arrays 的 copyOfRange()、sort()和 fill()方法时，只需要将数组作为参数传递给方法即可，至于内部的实现方式不需要关心。可见，使用这些方法不仅可以大幅度减少代码的书写量，而且能够使操作更加简单。

2.9 String 类和 StringBuffer 类

在应用程序中经常会用到字符串，所谓字符串就是指一连串的字符，它由许多单个字符连接而成，如多个英文字母所组成的一个英文单词。字符串中可以包含任意字符，这些字符必须包含在一对双引号(“”)之内，例如“abc”。在 Java 中定义了 String 和 StringBuffer 两个类封装字符串，并提供了一系列操作字符串的方法，它们都位于 java.lang 包中，因此不需要导包就可以直接使用。下面针对 String 类和 StringBuffer 类进行详细讲解。

2.9.1 String 类

在使用 String 类之前，首先需要对 String 类进行初始化，初始化可以通过两种形式实现。

(1) 使用字符串常量直接初始化一个 String 对象，具体代码如下：

```
String str1="abc";
```

(2) 使用 String 的构造方法初始化 String 对象，具体代码如下：

```
String str1=new String();          //初始化一个空的字符串
String str2=new String("abc");     //根据指定字符串内容初始化
```

String 类在实际开发中的应用非常广泛，因此，灵活地使用 String 类是非常重要的，下面介绍 String 类常用的一些方法，如表 2-8 所示。

表 2-8 String 类的常用方法

方法声明	功能描述
int indexOf(int ch)	返回指定字符在此字符串中第一次出现处的索引
int indexOf(String str)	返回指定子字符串在此字符串中第一次出现处的索引
char charAt(int index)	返回字符串中 index 位置上的字符，其中 index 的取值范围是 0～字符串长度－1
boolean endsWith(String suffix)	判断此字符串是否以指定的字符串结尾

续表

方法声明	功能描述
int length()	返回此字符串的长度
boolean equals(Object anObject)	将此字符串与指定的字符串比较，如果相等则返回 true，否则返回 false
boolean isEmpty()	当且仅当字符串长度为 0 时返回 true
boolean startsWith(String prefix)	判断此字符串是否以指定的字符串开始
boolean contains(CharSequence cs)	判断此字符串中是否包含指定的字符序列
String toLowerCase()	使用默认语言环境的规则将 String 中的所有字符转换为小写
String toUpperCase()	使用默认语言环境的规则将 String 中的所有字符转换为大写
char[] toCharArray()	将此字符串转换为一个字符数组
String replace(CharSequence oldstr, CharSequence newstr)	返回一个新的字符串，它是通过利用 newstr 替换此字符串中出现的所有 oldstr 得到的
String[] split(String regex)	根据参数 regex 将原来的字符串分割为若干个子字符串
String substring(int beginIndex)	返回一个新字符串，它包含从指定的 beginIndex 处开始，直到此字符串末尾的所有字符
String substring(int beginIndex, int endIndex)	返回一个新字符串，它包含从指定的 beginIndex 处开始，直到索引 endIndex－1 处的所有字符
String trim()	返回一个新字符串，它去除了原字符串首尾的空格

下面通过案例介绍 String 类中常用方法的使用，如文件 2-17 所示。

文件 2-17　Example17.java

```
1  package com.itheima.example;
2  public class Example17 {
3      public static void main(String[] args) {
4          String s="ababcabcdedcba";          //声明字符串
5          System.out.println("字符串的长度为："+s.length());
6          System.out.println("字符串中第一个字符："+s.charAt(3));
7          System.out.println("字符 c 第一次出现的位置："+s.indexOf('c'));
8          System.out.println("子字符串第一次出现的位置:"+s.indexOf("ab"));
9          System.out.println("字符 c 最后一次出现的位置："+s.lastIndexOf('d'));
10         System.out.println("子字符串最后一次出现的位置:"+s.lastIndexOf
           ("ab"));
11     }
12 }
```

运行结果如图 2-25 所示。

从图 2-25 中可以看出，String 类提供的方法可以很方便地获取字符串的长度，获取指定位置的字符以及指定字符的位置。

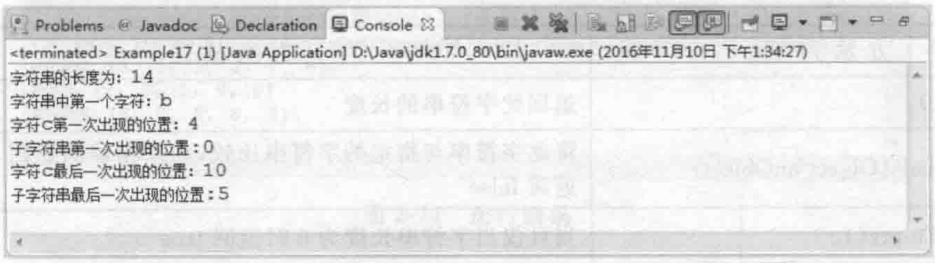


图 2-25 运行结果

2.9.2 StringBuffer 类

由于 String 字符串是常量，一旦创建，其内容和长度不可改变。如果需要对一个字符串进行修改，则只能创建新的字符串。为此，JDK 中提供了一个 StringBuffer 类（也称字符串缓冲区），与 String 类不同的是，它的内容和长度可变。StringBuffer 类如同一个字符容器，当在其中添加或删除字符时，并不会产生新的 StringBuffer 对象。

针对添加和删除字符的操作，StringBuffer 类提供了一系列的方法，具体如表 2-9 所示。

表 2-9 StringBuffer 类的常用方法

方法声明	功能描述
StringBuffer append(char c)	添加参数到 StringBuffer 对象中
StringBuffer insert(int offset,String str)	在字符串中的 offset 位置插入字符串 str
StringBuffer delete(intstart,int end)	删除 StringBuffer 对象中指定范围的字符或字符串序列
StringBuffer deleteCharAt(int index)	移除此序列指定位置的字符
StringBuffer replace(int start,int end,String s)	在 StringBuffer 对象中替换指定的字符或字符串序列
void setCharAt(int index, char ch)	修改指定位置 index 处的字符序列
StringBuffer reverse()	将此字符序列用其反转形式取代
String toString()	返回 StringBuffer 缓冲区中的字符串

表 2-9 中列出了 StringBuffer 的一系列常用方法，对于初学者来说比较难以理解。下面通过案例学习一下表中方法的具体使用，如文件 2-18 所示。

文件 2-18 Example18.java

```
1  package com.itheima.example;
2  public class Example18 {
3      public static void main(String[] args) {
4          System.out.println("**************1、添加**************");
5          add();
6          System.out.println("**************2、删除**************");
7          remove();
8          System.out.println("**************3、修改**************");
9          alter();
```

```
10      }
11      public static void add() {
12          StringBuffer sb=new StringBuffer();              //定义一个字符串缓冲区
13          sb.append("abcdefg");                            //在末尾添加字符串
14          System.out.println("append 添加结果: "+sb);
15          sb.insert(3, "321");                             //在指定位置插入字符串
16          System.out.println("insert 添加结果: "+sb);
17      }
18      public static void remove() {
19          StringBuffer sb=new StringBuffer("abcdefg");
20          sb.delete(2, 4);                                 //指定范围删除
21          System.out.println("删除指定位置结果: "+sb);
22          sb.deleteCharAt(3);                              //指定位置删除
23          System.out.println("删除指定位置结果: "+sb);
24          sb.delete(0, sb.length());                       //清空缓冲区
25          System.out.println("清空缓冲区结果: "+sb);
26      }
27      public static void alter() {
28          StringBuffer sb=new StringBuffer("abcdef");
29          sb.setCharAt(2, 'h');                            //修改指定位置字符
30          System.out.println("修改指定位置字符结果: "+sb);
31          sb.replace(1, 3, "yy");                          //替换指定位置字符串或字符
32          System.out.println("替换指定位置字符(串)结果: "+sb);
33          System.out.println("字符串翻转结果: "+sb.reverse());
34      }
35  }
```

运行结果如图 2-26 所示。

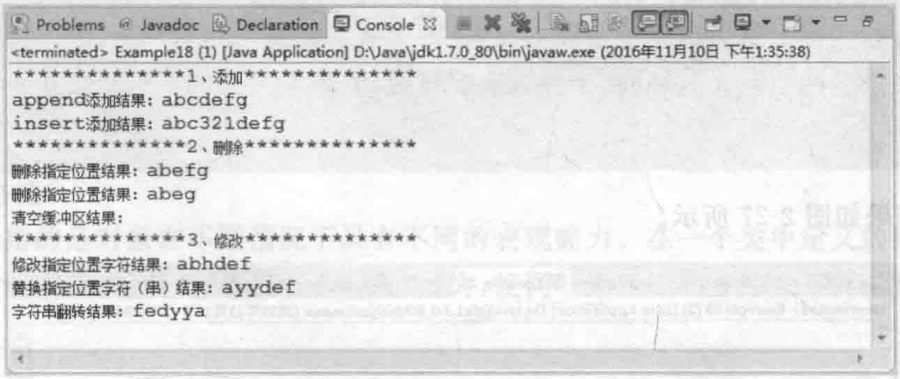


图 2-26　运行结果

在文件 2-18 中，提供了很多 StringBuffer 类的相关方法，其中 append()和 insert()方法是最常用的，并且这两个方法有很多重载形式，它们都用于添加字符。delete()方法用于删除指定位置的字符，setCharAt()以及 replace()方法用于替换指定位置的字符。

2.10 包装类

在 Java 中，很多类的方法都需要接收引用类型的对象，此时就无法将一个基本数据类型的值传入。为了解决这样的问题，JDK 中提供了一系列的包装类，通过这些包装类可以将基本数据类型的值包装为引用数据类型的对象。在 Java 中，每种基本类型都有对应的包装类，具体如表 2-10 所示。

表 2-10 基本类型对应的包装类

基本数据类型	对应的包装类	基本数据类型	对应的包装类
char	Character	long	Long
byte	Byte	float	Float
int	Integer	double	Double
short	Short	boolean	Boolean

表 2-10 列举了 8 种基本数据类型及其对应的包装类。其中，除了 Integer 和 Character 类以外，其他包装类的名称和基本数据类型的名称一致，只是类名的第一个字母需要大写。

包装类和基本数据类型在进行转换时，引入了装箱和拆箱的概念，其中装箱是指将基本数据类型的值转为引用数据类型；反之，拆箱是指将引用数据类型的对象转为基本数据类型。下面介绍拆箱和装箱的过程，如文件 2-19 所示。

文件 2-19 Example19.java

```
1  package com.itheima.example;
2  public class Example19 {
3      public static void main(String args[]) {
4          Integer a=3;            //自动装箱
5          int b=a+4;              //自动拆箱
6          System.out.println(b);
7      }
8  }
```

运行结果如图 2-27 所示。

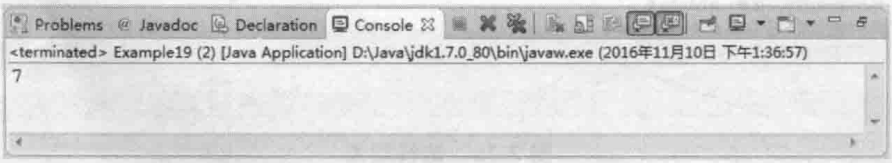


图 2-27 运行结果

文件 2-19 演示了包装类 Integer 的拆箱和装箱过程，首先将 int 类型数字 3 赋值给 Integer 类型，此过程称为自动装箱。Integer 类型的数值 3 与 int 类型的数值 4 相加，得出的结果赋值给了 int 类型，此过程称为自动拆箱。

第3章 面向对象

Java是一种面向对象的编程语言，即“万物皆为对象”。面向对象的思想是最为简单的编程思想，也最接近人类的思维习惯，本章将详细讲解关于面向对象的编程知识。

3.1 面向对象的概念和特征

现实生活中存在各种形态不同的事物，这些事物之间存在着各种各样的联系。在程序中使用对象映射现实中的事物，使用对象的关系描述事物之间的联系，这种思想就是面向对象。面向对象的特征主要可以概括为封装性、继承性和多态性，下面针对这三种特征进行简单介绍。

1. 封装性

封装是面向对象的核心思想，是指将对象的属性和行为封装起来，不需要让外界知道内部是如何实现细节的，这就是封装的思想。例如，使用电视机的用户不需要了解电视机内部复杂工作的具体细节，他们只需要知道开、关、选台、调台等设置与操作就可以了。

2. 继承性

继承性是描述类与类之间的关系，在已有类的基础上扩展出新的类。例如，有一个火车类，该类描述了火车的特性和功能，而高铁类中不仅应该包含火车的特性和功能，还应该增加高铁特有的功能，这时可以让高铁类继承火车类，在高铁类中单独添加高铁特有的方法就可以了。继承不仅增强了代码的复用性，提高了开发效率，同时还为后期的代码维护提供了便利。

3. 多态性

多态性指的是对象在不同情况下具有不同的表现能力。在一个类中定义的属性和方法被其他的类继承后，它们可以表现出不同的行为，使同一个属性和方法在不同的类中具有不同的意义。

3.2 类与对象

3.2.1 类与对象的关系

面向对象的编程思想是让程序代码中对事物的描述和在现实中事物的形态相关联。为了实现这些联系，在面向对象的思想中提出了两个概念，即类和对象。其中，类是一组具有

共同特征和行为的对象的抽象描述，而对象是表示该类事物的具体个体。类与对象的关系如图 3-1 所示。

从图 3-1 中可以看出，汽车图纸就相当于一个类，每个汽车就相当于一个对象。因为汽车本身属于一种广义的概念，并不能代表具体的东西，所以，从汽车类到具体的某辆汽车便可以看出类用于描述多个对象的共同特征，它是对象的模板，而对象是用于描述现实中的个体，它是类的实例。

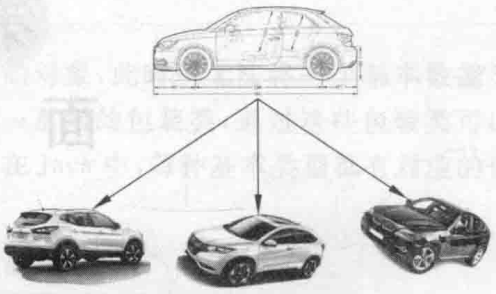


图 3-1 类与对象

3.2.2 类的定义

对象是面向对象思想中的核心，为了在程序中创建对象，首先需要定义一个类。类是通过“class”关键字定义的，类中可以定义成员变量和成员方法，其中成员变量用于描述对象的特征（也称为属性），成员方法用于描述对象的行为（简称为方法）。

假设要在程序中描述汽车的相关信息，可以先设计一个汽车类，在这个类中定义两个属性 color 和 num，分别表示汽车的颜色和轮胎，定义一个方法 run()表示汽车跑的行为。接下来根据这个描述设计一个 Car 类，首先创建一个 chapter03 项目，然后在该项目下创建一个 com.itheima.example01 包，在该包下创建一个 Car 类，如文件 3-1 所示。

文件 3-1 Car.java

```
1  package com.itheima.example01;
2  public class Car {
3      String color;                    //定义一个颜色属性
4      int num;                         //定义一个轮胎属性
5      public void run() {
6          //方法中打印属性 color 和 num 的值
7          System.out.println("这辆车的颜色是"+color+",轮胎数量是"+num);
8      }
9  }
```

在文件 3-1 中，定义了一个 Car 类。其中，Car 是类名，color 和 num 是成员变量，run()是成员方法。在成员方法 run()中可以直接访问成员变量 color 和 num。

3.2.3 对象的创建

3.2.2 节提到了对象是面向对象的核心，因此仅有类是不够的，还需要创建对象，通过对象的引用访问类中的成员。在 Java 中，通过 new 关键字创建对象，具体格式如下：

```
类名 对象名=new 类名();
```

例如，创建 Car 类的实例对象代码如下：

```
Car c=new Car();
```

在上述代码中，通过 new 关键字创建了一个 Car 的实例对象，Car c 则声明了一个 Car 类型的变量 c。中间的等号用于将 Car 对象在内存中的地址赋值给变量 c，这样变量 c 便持有了对象的引用。在创建 Car 对象后，便可以通过“对象的引用.对象成员”的方式访问该对象中所有的成员，下面在当前包中创建一个 Example01 类访问对象的成员，如文件 3-2 所示。

文件 3-2　Example01.java

```
1  package com.itheima.example01;
2  class Example01 {
3      public static void main(String[] args) {
4          Car c1=new Car();          //创建第一个 Car 对象
5          Car c2=new Car();          //创建第二个 Car 对象
6          c1.color="red";            //为 color 属性赋值
7          c1.num=4;
8          c1.run();                  //调用对象的方法
9          c2.run();
10     }
11 }
```

运行结果如图 3-2 所示。

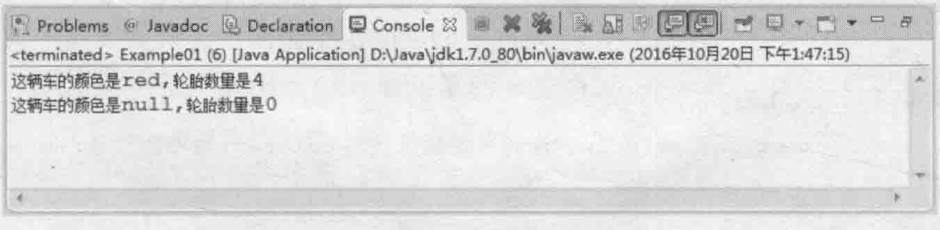


图 3-2　运行结果

从图 3-2 所示的运行结果中可以看出，c2 对象的 color 属性也是有值的，其默认值为 null，这是因为在实例化对象时，JVM 会自动为成员变量进行初始化，针对不同数据类型的成员变量，JVM 会赋予不同的默认值，如 long 类型默认初始值为 0L，boolean 默认初始值为 false 等。

3.2.4　类的封装

类的封装是指，在定义一个类时将类中的属性进行私有化，即使用 private 关键字进行修饰，私有化属性只能在当前所在类中被访问，如果外界想要访问私有属性，需要提供一些 public 修饰的公共方法，其中包括用于获取属性值的 getter 方法和设置属性值的 setter 方法。

接下来在 src 目录下创建一个 com.itheima.example02 包（由于一个包中不能出现同名的类，因此需要再创建一个包），在该包下创建一个 Car 类，实现类的封装，如文件 3-3 所示。

文件 3-3 Car.java

```
1  package com.itheima.example02;
2  public class Car {
3      private String color;              //将 color 属性私有化
4      private int num;                   //将 num 属性私有化
5      //下面是公有的 getter 和 setter 方法
6      public String getColor() {
7          return color;
8      }
9      public void setColor(String color) {
10         this.color=color;          //this.color 访问的是成员变量,后面会详细讲解
11     }
12     public int getNum() {
13         return num;
14     }
15     public void setNum(int carNum) {
16         //对传入的参数进行检查
17         if (carNum !=4) {
18             System.out.println("输入的轮胎数量不正确!");
19         } else {
20             num=carNum;                //给属性赋值
21         }
22     }
23     public void run() {
24         System.out.println("这辆车的颜色是"+color+",轮胎数量是"+num);
25     }
26 }
```

在文件 3-3 中,Car 类定义了两个私有化的成员变量 color 和 num,并为这两个属性提供了公共的 getter 和 setter 方法,其中 getColor()和 getNum()方法用于获取属性的值,而 setColor()和 setNum()方法用于设置属性的值。

接下来创建测试类 Example02,在该类中创建一个 Car 对象,然后调用 setNum()、setColor()以及 run()方法,如文件 3-4 所示。

文件 3-4 Example02.java

```
1  package com.itheima.example02;
2  public class Example02 {
3      public static void main(String[] args) {
4          Car c=new Car();
5          c.setNum(-1);
6          c.setColor("red");
7          c.run();
8      }
9  }
```

运行结果如图 3-3 所示。

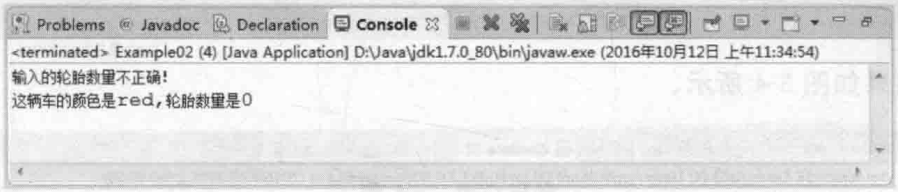


图 3-3　运行结果

从图 3-3 的运行结果中可以看出，在 setNum()方法中对参数 carNum 进行了校验，由于－1 不符合规则，因此在控制台输出为“输入的轮胎数量不正确!”，num 属性没有被赋值，仍为初始值 0。

3.3　构造方法

3.3.1　构造方法的定义

在实例化对象赋值时，不仅可以通过 setter 方法完成，还可以通过构造方法完成。所谓的构造方法是类的一个特殊成员，它会在类实例化对象时被自动调用。在定义构造方法时，必须同时满足以下三个条件。

(1) 方法的名称和类名必须相同。

(2) 在方法名称前没有返回值类型的声明。

(3) 在方法体中不可以使用 return 语句返回一个值，但允许单独写 return 语句作为方法的结束。

下面通过案例演示如何在类中定义构造方法，如文件 3-5 所示。

文件 3-5　Car.java

```
1  package com.itheima.example03;
2  public class Car {
3      //下面是类的构造方法
4      public Car() {
5          System.out.println("无参数的构造方法执行了...");
6      }
7  }
```

在文件 3-5 中，Car 类只定义了一个无参数的构造方法，用于对象的初始化。接下来创建测试类 Example03，调用无参数构造方法，如文件 3-6 所示。

文件 3-6　Example03.java

```
1  package com.itheima.example03;
2  public class Example03 {
3      public static void main(String[] args) {
4          Car c=new Car();          //实例化 Car 对象
```

```
5      }
6  }
```

运行结果如图 3-4 所示。

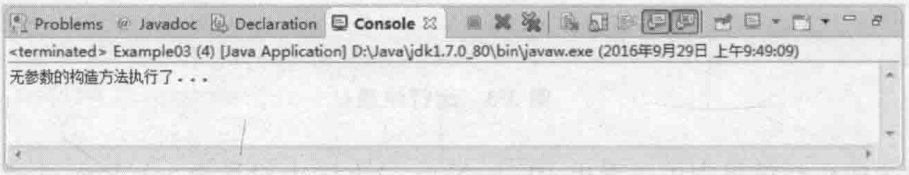


图 3-4　运行结果

从图 3-4 的运行结果中可以看出，Car 类的无参数构造方法被执行，这是因为在实例化对象时会自动调用该类的构造方法。

在类中既然可以定义无参数的构造方法，那么是否可以定义有参数的构造方法并通过参数对属性赋值呢？下面通过案例进行演示，如文件 3-7 所示。

文件 3-7　Car.java

```
1  package com.itheima.example04;
2  public class Car {
3      String color;
4      //定义有参数的构造方法
5      public Car (String c) {
6          color=c;                    //为 color 属性赋值
7      }
8      public void run() {
9          System.out.println("这辆车的颜色是"+color);
10     }
11 }
```

在文件 3-7 中，定义了一个成员变量和一个有参数的构造方法，该方法执行时，会自动为其成员变量进行赋值。接下来创建测试类 Example04，调用有参数构造方法，如文件 3-8 所示。

文件 3-8　Example04.java

```
1  package com.itheima.example04;
2  public class Example04 {
3      public static void main(String[] args) {
4          Car c=new Car("red");          //实例化 Car 对象
5          c.run();
6      }
7  }
```

运行结果如图 3-5 所示。

从图 3-5 的运行结果中可以看出，Car 对象在调用 run()方法时，其 color 属性已经被赋值为 red。

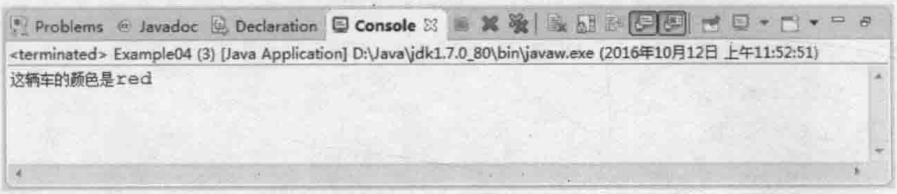


图 3-5　运行结果

3.3.2　构造方法的重载

构造方法与普通方法一样也可以实现重载，由于构造方法的方法名与类名相同，因此只要每个构造方法的参数类型和参数个数不同即可实现构造方法的重载。在创建实例对象时，可以通过调用不同的构造方法为不同的属性进行赋值。下面通过案例学习构造方法的重载，如文件 3-9 所示。

文件 3-9　Car.java

```
1  package com.itheima.example05;
2  public class Car {
3      String color;
4      int num;
5      //定义两个参数的构造方法
6      public Car(String c_color, int c_num) {
7          color=c_color;              //为 color 属性赋值
8          num=c_num;                  //为 num 属性赋值
9      }
10     //定义一个参数的构造方法
11     public Car(String c_color) {
12         color=c_color;              //为 color 属性赋值
13     }
14     public void run() {
15         //打印 color 和 num 的值
16         System.out.println("这辆车的颜色是"+color+",轮胎数量是"+num);
17     }
18 }
```

在文件 3-9 中，定义了两个重载的构造方法，在创建对象时，根据传入参数的不同，分别调用不同的构造方法。接下来创建测试类 Example05，分别调用两个构造方法，如文件 3-10 所示。

文件 3-10　Example05.java

```
1  package com.itheima.example05;
2  public class Example05 {
3      public static void main(String[] args) {
4          //分别创建两个对象 c1 和 c2
5          Car c1=new Car("red");
```

```
        Car c2=new Car("black", 4);
        //通过对象 c1 和 c2 调用 run()方法
        c1.run();
        c2.run();
    }
}
```

运行结果如图 3-6 所示。

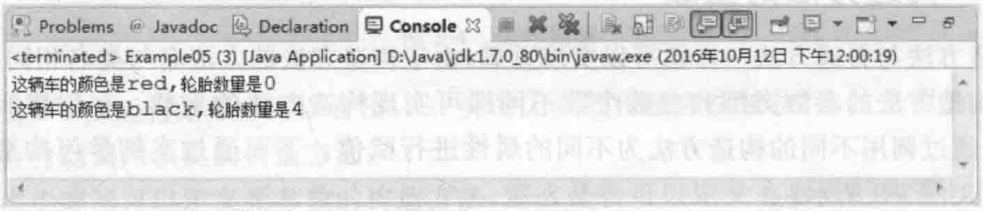


图 3-6 运行结果

从图 3-6 的运行结果中可以看出，两个构造方法对属性赋值的情况是不一样的，其中一个参数的构造方法只针对 color 属性进行赋值，这时 num 属性的值为默认值 0。

3.4 this 关键字

在 3.3.2 节的案例中，成员变量使用的是 num，构造方法使用的是 c_num，这样类似的变量一旦增多，则可读性会变得很差，这时需要在一个类中使用统一的变量名称表示年龄。为了解决这样的问题，Java 中提供了一个 this 关键字表示当前对象，可以在方法中调用其他的成员。

下面详细讲解 this 关键字在程序中的三种常见用法。

(1) 通过 this 关键字可以明确地访问一个类的成员变量，解决与局部变量名称相同的问题。具体示例代码如下：

```
public class Car {
    String color;
    public Car(String color) {
        this.color=color;
    }
    public int getColor() {
        return this.color;
    }
}
```

在上面的代码中，构造方法的参数被定义为 color，它是一个局部变量，在类中还定义了一个成员变量，名称也是 color。在构造方法中如果使用 color，则访问局部变量，但如果使用 this.color，则访问成员变量。

(2) 通过 this 关键字调用成员方法，具体示例代码如下：

```
1  public class Car {
2      public void show() {
3      }
4      public void run() {
5          this.show();
6      }
7  }
```

在上面的 run()方法中，使用 this 关键字调用 show()方法。注意，此处的 this 关键字可以省略不写，也就是说在上面的代码中，写成 this. show()和 show()，效果是完全一样的。

构造方法在实例化对象时被 JVM 自动调用，在程序中不能像调用其他方法一样调用构造方法，但可以在一个构造方法中使用“this([参数 1，参数 2...])”的形式调用其他构造方法。具体代码如文件 3-11 所示。

文件 3-11　Car. java

```
1  package com.itheima.example06;
2  public class Car {
3      public Car() {
4          System.out.println("无参数的构造方法执行了...");
5      }
6      public Car(String color) {
7          this();                          //调用无参数的构造方法
8          System.out.println("有参数的构造方法执行了...");
9      }
10 }
```

在文件 3-11 中，定义了两个重载的构造方法，在上述代码的第 7 行调用了 this()，此方法会调用无参数的构造方法。下面创建测试类 Example06，调用有参数的构造方法，如文件 3-12 所示。

文件 3-12　Example06. java

```
1  package com.itheima.example06;
2  public class Example06 {
3      public static void main(String[] args) {
4          Car c=new Car("red");          //实例化 Car 对象
5      }
6  }
```

运行结果如图 3-7 所示。

从图 3-6 的运行结果中可以看出，先执行了无参数的构造方法，之后再执行有参数的构造方法。需要注意的是，使用 this 关键字调用其他构造方法时只能出现在构造方法中，并且只能位于构造方法的第一行且只能出现一次。另外，不能在两个构造方法中使用 this 相互调用，否则会出现编译错误。

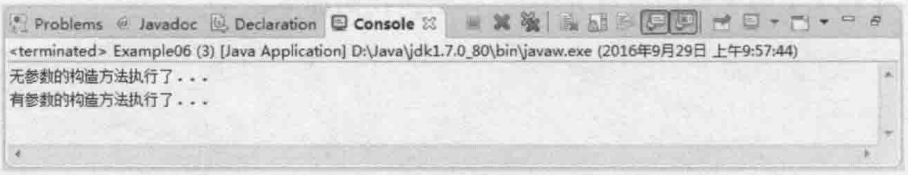


图 3-7 运行结果

3.5 static 关键字

3.5.1 静态变量

通过前面的学习读者已经了解，如果使用一个类，则会在产生实例化对象时分别在堆内存中分配空间，在堆内存中要保存对象中的属性，每个对象都有自己的属性，如果有些属性希望被所有对象共享，就必须使用 static 关键字修饰成员变量，该变量被称为静态变量，可以直接使用“类名.变量名”的形式调用，如文件 3-13 所示。

文件 3-13 Car.java

```
1  package com.itheima.example07;
2  public class Car {
3        static String carName="大众";          //定义静态变量 carName
4  }
```

在文件 3-13 中，定义了一个静态变量 carName，用于表示汽车所在的厂商，它被所有的实例所共享。接下来创建测试类 Example07，在该类中分别通过 Car. carName 和 c2. carName 的方式调用静态变量，如文件 3-14 所示。

文件 3-14 Example07.java

```
1  package com.itheima.example07;
2  public class Example07 {
3      public static void main(String[] args) {
4          Car c1=new Car();
5          Car.carName="大众";          //为静态变量赋值
6          System.out.println("该辆车的厂商是："+Car.carName);
7          System.out.println("该辆车的厂商是："+c1.carName);
8      }
9  }
```

运行结果如图 3-8 所示。

从图 3-8 的运行结果中可以看出，Car. carName 和 c2. carName 是两种不同的访问方式，输出结果均为“大众”。

注意：static 关键字只能用于修饰成员变量，不能用于修饰局部变量，否则编译会报错，如下列代码就是非法的。

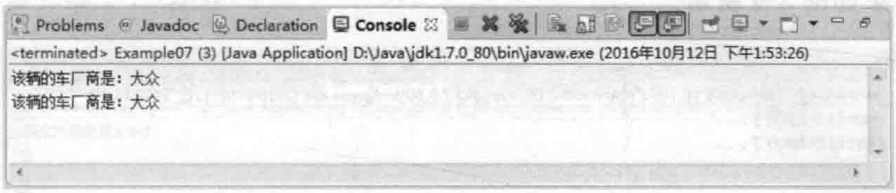


图 3-8　运行结果

```
public class Car {
    public void run() {
        static int number=4;            //这行代码是非法的,编译会报错
    }
}
```

3.5.2　静态方法

通过前面的讲解可知，要想调用某个方法，必须要创建一个对象，那么我们的头脑中一定会有这样一个疑问：有没有一种方法可以不创建对象就能直接调用呢？实际上有一种方法可以这样操作，那就是静态方法，静态方法与普通方法的区别是在方法前面加一个 static 关键字，这种方法称为静态方法。同静态变量一样，静态方法可以使用“类名.方法名”的方式访问，也可以通过类的实例对象访问。下面通过案例学习静态方法的使用，如文件 3-15 所示。

文件 3-15　Car.java

```
1  package com.itheima.example08;
2  public class Car {
3      public static void run() {          //定义静态方法
4          System.out.println("run()方法执行了...");
5      }
6  }
```

在文件 3-15 中可以看出，该类中只定义了一个静态方法 run()。接下来创建测试类 Example08，分别使用两种方式调用静态方法，如文件 3-16 所示。

文件 3-16　Example08.java

```
1  package com.itheima.example08;
2  public class Example08 {
3      public static void main(String[] args) {
4          //1.类名.方法的方式调用静态方法
5          Car.run();
6          //2.实例化对象的方式调用静态方法
7          Car c=new Car();
8          c.run();
9      }
10 }
```

运行结果如图 3-9 所示。

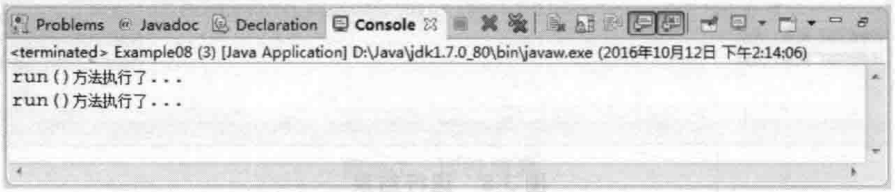


图 3-9 运行结果

从图 3-9 的运行结果中可以看出，两种不同的方式都可以调用静态方法，这说明通过实例化的对象同样可以调用静态方法。

注意：在一个静态方法中只能访问用 static 修饰的成员，原因在于没有被 static 修饰的成员需要先创建对象才能访问，而静态方法在被调用时不需要创建任何对象。

3.5.3 静态代码块

在 Java 类中，使用一对大括号包围起来的若干行代码被称为一个代码块，用 static 关键字修饰的代码块称为静态代码块。当类被加载时，静态代码块会被执行，由于类只加载一次，所以静态代码块只会执行一次。在程序中，通常会使用静态代码块对类的成员变量进行初始化。下面通过案例了解静态代码块的使用，如文件 3-17 所示。

文件 3-17 Car.java

```
1  package com.itheima.example09;
2  public class Car {
3      static String color;
4      //下面是一个静态代码块
5      static {
6          color="red";
7          System.out.println("这辆车的颜色是"+color);
8      }
9  }
```

在文件 3-17 中可以看出，当静态代码块被执行时，会为其静态变量赋值为 red，并输出结果。接下来创建测试类 Example09，在 main()方法中创建两个 Car 的实例对象 c1 和 c2，如文件 3-18 所示。

文件 3-18 Example09.java

```
1  package com.itheima.example09;
2  public class Example09 {
3      public static void main(String[] args) {
4          //下面的代码创建了两个 Car 对象
5          Car c1=new Car();
6          Car c2=new Car();
7      }
8  }
```

运行结果如图 3-10 所示。

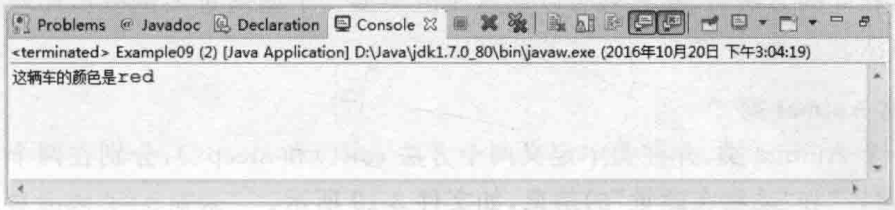


图 3-10　运行结果

从图 3-10 的运行结果中可以看出，在两次创建对象的过程中，静态代码块中的内容只输出了一次，这就说明静态代码块在类中只加载一次。

3.6　类的继承

3.6.1　继承的概念

在 Java 中，类的继承是指在一个现有类的基础上产生一个新的类，产生的新类被称为子类，现有类被称为父类，子类会自动拥有父类的属性和方法。例如定义一个 Animal 类作为父类，该类拥有一个 call()方法，当子类 Cow 和 Sheep 继承自 Animal 类时，就会自动拥有 call()方法。下面通过图例描述类的继承关系，如图 3-11 所示。



图 3-11　Animal 类的继承关系图

在程序代码中，如果想定义一个类继承另一个类，需要使用 extends 关键字。如果一个类没有使用 extends 关键字明确标识继承另一个类，那么这个类就是默认继承 Object 类。Object 类是所有类的父类，该类中的方法适用于其所有子类，其类中常用的方法有 toString()和 hashCode()方法等。

注意：在类的继承中，需要注意以下几个问题。

(1) 在 Java 中，类只支持单继承，不允许多重继承，也就是说一个类只能有一个直接父类。例如 Cow 类继承 Animal 类之后，就不允许继承其他类。

(2) 多个类可以继承一个父类，例如 Cow 和 Sheep 类都可以继承 Animal 类。

(3) 在 Java 中，多层继承是允许的，即一个类的父类可以再继承其他的父类，例如 Zebra 类继承自 Horse 类，而 Horse 类又可以继承 Animal 类，即“子孙三代”。

3.6.2　重写父类方法

在继承关系中，子类会继承父类中定义的方法，但子类也可以在父类的基础上拥有自己

的特征，即对父类的方法进行重写。需要注意的是，在子类中重写的方法必须与父类被重写的方法具有相同的方法名、参数列表以及返回值类型。下面通过案例演示如何实现子类重写父类的方法。

1. 创建 Animal 类

创建一个 Animal 类，并在类中定义两个方法 call()和 sleep()，分别在两个方法中输出“动物发出声音”和“动物在睡觉”的信息，如文件 3-19 所示。

文件 3-19　Animal.java

```
1  package com.itheima.example10;
2  //定义 Animal 类
3  public class Animal {
4      //定义动物发出声音的方法
5      public void call() {
6          System.out.println("动物发出声音...");
7      }
8      //定义动物睡觉的方法
9      public void sleep() {
10         System.out.println("动物在睡觉...");
11     }
12 }
```

2. 创建 Cow 类

创建一个 Cow 类，使其继承 Animal 类，并在 Animal 类中重写 call()方法，如文件 3-20 所示。

文件 3-20　Cow.java

```
1  package com.itheima.example10;
2  //定义 Cow 类继承 Animal 类
3  public class Cow extends Animal {
4      //定义一个打印 name 的方法
5      public void call() {
6          System.out.println("哞...");
7      }
8  }
```

3. 创建测试类

创建一个测试类 Example10，分别调用 call()方法和 sleep()方法，如文件 3-21 所示。

文件 3-21　Example10.java

```
1  package com.itheima.example10;
2  public class Example10 {
3      public static void main(String[] args) {
```

```
4           Cow c=new Cow();                    //创建一个 Cow 类的实例对象
5           c.call();                           //调用 Cow 类重写的 call()方法
6           c.sleep();                          //调用父类 Animal 的 sleep()方法
7       }
8   }
```

运行结果如图 3-12 所示。

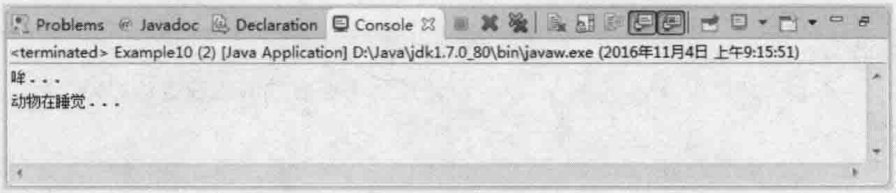


图 3-12　运行结果

从图 3-12 的运行结果中可以看出，当子类继承父类时，会拥有父类中所有的成员，由此可知，在调用 sleep()方法时，会自动调用父类的 sleep()方法。当子类重写了父类方法之后，创建的子类对象会自动调用重写的方法，即调用的是子类的 call()方法。

注意：子类重写父类方法时，不可以使用比父类中被重写的方法更严格的访问权限，如父类中的方法是 public 的，子类的方法就不可以是 private 的。

3.6.3　super 关键字

从图 3-12 的运行结果中可以看出，当子类重写父类的方法后，子类对象将不能访问父类被重写的方法。为了解决这个问题，在 Java 中专门提供了 super 关键字用于访问父类的成员变量、成员方法和构造方法，具体格式如下：

```
super.成员变量                          //访问成员变量
super.成员方法([参数 1,参数 2...])      //访问成员方法
super([参数 1,参数 2...])               //访问构造方法
```

下面通过案例学习 super 的使用，具体步骤如下。

1. 创建 Animal 类

创建一个 Animal 类，在类中定义一个有参数的构造方法，并在该方法中输出一句话，如文件 3-22 所示。

文件 3-22　Animal.java

```
1  package com.itheima.example11;
2  //定义 Animal 类
3  public class Animal {
4      //定义 Animal 类有参数的构造方法
5      public Animal(String name) {
6          System.out.println("我是一头"+name);
7      }
8  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其继承 Animal 类，并定义一个无参数的构造方法，该方法中使用 super 关键字，调用父类有参数的构造方法，如文件 3-23 所示。

文件 3-23 Cow.java

```
1  package com.itheima.example11;
2  //定义 Cow 类继承 Animal 类
3  class Cow extends Animal {
4      public Cow() {
5          super("黄牛");                    //调用父类有参数的构造方法
6      }
7  }
```

3. 创建测试类

创建测试类 Example11，在 main()方法中实例化子类对象，如文件 3-24 所示。

文件 3-24 Example11.java

```
1  package com.itheima.example11;
2  //定义测试类
3  public class Example11 {
4      public static void main(String[] args) {
5          Cow c=new Cow();                  //实例化子类 Cow 对象
6      }
7  }
```

运行结果如图 3-13 所示。

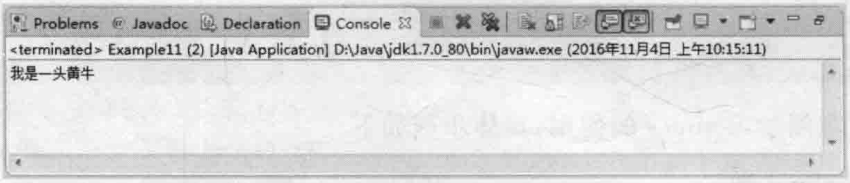


图 3-13 运行结果

从图 3-13 的运行结果中可以看出，Cow 类的构造方法在被调用时父类的构造方法也被调用了。需要注意的是，通过 super 关键字调用父类构造方法的代码必须位于子类构造方法的第一行，并且只能出现一次。

4. final 关键字

在 Java 中，如果父类的某些方法不希望再被子类重写，则必须把它们修饰为最终方法，即使用 final 修饰。final 关键字可用于修饰类、变量和方法，表示不能改变或者最终，因此被 final 修饰的类、变量和方法将具有以下特性。

- final 修饰的类不能被继承。
- final 修饰的方法不能被子类重写。

• final 修饰的变量(成员变量和局部变量)是常量,并且只能赋值一次。

接下来通过具体示例演示 final 关键字的三种特性。

(1) 在 Java 中,使用 final 关键字修饰的类不能有子类,具体示例代码如下所示:

```
final class A {}                    //使用 final 修饰类,不能被继承
class B extends A{}                 //编译失败,不能继承被 final 修饰的类
```

(2) 在 Java 中,使用 final 关键字修饰的方法不能被子类覆盖,具体示例代码如下所示:

```
class A {
    public final void print(){}                    //使用 final 关键字声明的方法不能被重写
}
class B extends A {
    public final void print(){}                    //编译失败,不能重写用 final 修饰的方法
}
```

(3) 在 Java 中,被 final 修饰的变量将变为常量,常量是不能被修改的,具体示例代码如下所示:

```
class A {
    final String NAME="XXX";                //使用 final 声明的变量就是常量
    public void print(){
        NAME="YYY";                         //编译失败,常量不可修改
    }
}
```

3.7　抽象类和接口

3.7.1　抽象类

在 Java 程序中,允许在定义方法时不写方法体,这种方法被称为抽象方法,抽象方法必须使用 abstract 关键字修饰。抽象方法的出现,解决了程序中某些方法的不确定实现,例如前面在定义 Animal 类时,call()方法用于表示动物的叫声,但是针对不同的动物,叫声也是不同的,因此在 call()方法中无法准确描述动物的叫声。抽象方法的语法格式如下:

```
abstract void call();                   //定义抽象方法 call()
```

如果一个类中定义了抽象方法,则该类必须定义为抽象类,抽象类也同样使用 abstract 关键字修饰,具体示例如下:

```
//定义抽象类 Animal
abstract class Animal {
    abstract int call();                //定义抽象方法 call()
}
```

需要注意的是，包含抽象方法的类必须声明为抽象类，但抽象类可以不包含任何抽象方法，只需要使用 abstract 关键字修饰即可。另外，抽象类是不可以被实例化的，因为抽象类中有可能包含抽象方法，抽象方法是没有方法体的，不可以被调用。如果想调用抽象类中定义的方法，则需要创建一个子类，在子类中将抽象类中的抽象方法进行实现。

下面通过案例学习如何实现抽象类中的方法。

1. 创建 Animal 类

创建一个 Animal 抽象类，并在类中定义一个抽象方法 call()，如文件 3-25 所示。

文件 3-25 Animal.java

```
1  package com.itheima.example12;
2  //定义抽象类 Animal
3  public abstract class Animal {
4      abstract void call();              //定义抽象方法 call()
5  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其继承抽象类 Animal，并在 Animal 类中重写抽象方法 call()，如文件 3-26 所示。

文件 3-26 Cow.java

```
1  package com.itheima.example12;
2  //定义 Cow 类继承抽象类 Animal
3  public class Cow extends Animal {
4      //实现抽象方法 call()
5      void call() {
6          System.out.println("哞...");
7      }
8  }
```

3. 创建测试类

创建一个测试类 Example12，在 main()方法中创建一个子类 Cow 对象，并调用该对象的 call()方法，如文件 3-27 所示。

文件 3-27 Example12.java

```
1  package com.itheima.example12;
2  //定义测试类
3  public class Example12 {
4      public static void main(String[] args) {
5          Cow c=new Cow();               //创建 Cow 类的实例对象
6          c.call();                      //调用 cow 对象的 call()方法
7      }
8  }
```

运行结果如图 3-14 所示。

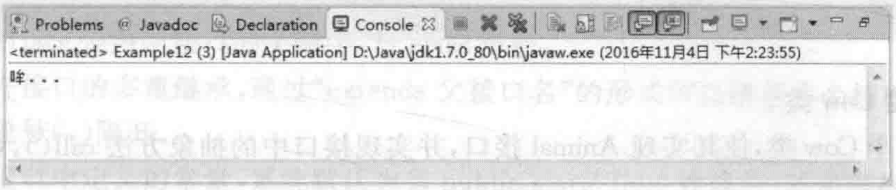


图 3-14　运行结果

从图 3-14 的运行结果中可以看出，子类实现了父类的抽象方法后，可以正常进行实例化，并通过实例化对象调用子类中的方法。

3.7.2　接口

在 Java 中还存在一种特殊的抽象类，该类中只存在常量和抽象方法，而不存在变量的定义和方法实现，这种特殊的类被称为接口。接口是由常量和抽象方法组成的特殊类，是对抽象类的进一步抽象。定义接口时，需要使用 interface 关键字，具体示例代码如下：

```
interface Animal{
    String ANIMAL_ACTION="动物的行为动作";
    void call();
}
```

从上述代码中可以看出，接口中定义了一个全局常量和一个抽象方法，全局常量默认使用了“public static final”修饰，抽象方法默认使用了“public abstract”修饰。需要注意的是，在接口定义方法时，所有的方法必须都是抽象的，所以不能通过实例化对象的方式调用接口中的方法。此时需要定义一个类，并使用 implements 关键字实现接口中所有的方法。接口实现类的示例代码具体如下：

```
class Cow implements Animal{
    public void call(){...}
}
```

下面通过案例学习接口的使用，具体步骤如下。

1. 创建 Animal 接口

创建一个 Animal 接口，并在接口中定义一个全局常量和抽象方法 call()，如文件 3-28 所示。

文件 3-28　Animal.java

```
1  package com.itheima.example13;
2  //定义 Animal 接口
3  interface Animal {
4      //全局常量，默认修饰为 public static final
5      String ANIMAL_ACTION="动物的行为动作";
```

```
    void call();        //抽象方法 call(),默认修饰为 public abstract
}
```

2. 创建 Cow 类

创建一个 Cow 类,使其实现 Animal 接口,并实现接口中的抽象方法 call(),如文件 3-29 所示。

文件 3-29 Cow.java

```
package com.itheima.example13;
//Cow 类实现了 Animal 接口
class Cow implements Animal {
    //实现 call()方法
    public void call() {
        System.out.println(ANIMAL_ACTION+":"+"哞...");
    }
}
```

3. 创建测试类

创建一个测试类 Example13,在 main()方法中创建了一个 Cow 对象,并调用该对象的 call()方法,如文件 3-30 所示。

文件 3-30 Example13.java

```
package com.itheima.example13;
//定义测试类
public class Example13 {
    public static void main(String args[]) {
        Cow c=new Cow();            //创建 Cow 类的实例对象
        c.call();                   //调用 cow 对象的 call()方法
    }
}
```

运行结果如图 3-15 所示。

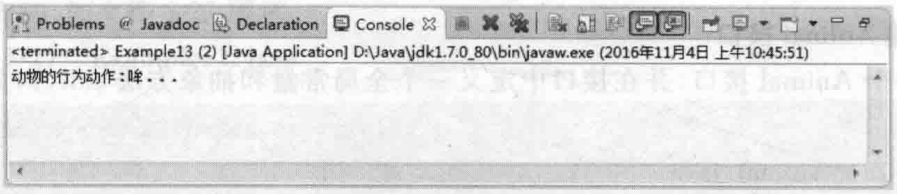


图 3-15 运行结果

从图 3-15 的运行结果中可以看出,Cow 类在实现了 Animal 接口后是可以被实例化的,并且实例化后就可以调用 Cow 类中的方法。需要注意的是,一个类实现一个接口,必须实现接口中所有的方法,如果不能实现某个方法,则必须写出一个空实现的方法。

为了加深读者对接口的了解，接下来对接口的特性进行归纳，具体如下。

- 接口的访问限定只有 public 和默认。
- interface 是声明接口的关键字，与 class 类似。
- 允许接口的多重继承，通过“extends 父接口名”的形式可以继承多个接口，接口之间用逗号(,)隔开。
- 在接口中定义的常量，系统默认为是 public static final 修饰的，不需要指定。
- 在接口中声明的方法，系统默认为是 public abstract 修饰的。
- 接口中的方法都是抽象的，不能实例化对象。

3.8　多态

3.8.1　多态概述

多态就是指在同一个方法中，由于参数类型不同而出现执行效果各异的现象。例如在实现动物叫的方法中，由于每种动物叫声不同，因此可以在方法中接收一个动物类型的参数，当传入具体的某个动物时发出具体的叫声。在 Java 中为了实现多态，允许使用父类类型的变量引用一个子类类型对象，根据子类对象特征的不同，得到不同的运行结果。下面通过案例演示多态的使用，具体步骤如下。

1. 创建 Animal 接口

创建一个 Animal 接口，并在接口中定义一个抽象方法 call()，如文件 3-31 所示。

文件 3-31　Animal.java

```
1  package com.itheima.example14;
2  //定义接口 Animal
3  interface Animal {
4      void call();                    //定义抽象 call()方法
5  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其实现 Animal 接口，并在类中实现抽象方法 call()，如文件 3-32 所示。

文件 3-32　Cow.java

```
1  package com.itheima.example14;
2  //定义 Cow 类实现 Animal 接口
3  class Cow implements Animal {
4      //实现 call()方法
5      public void call() {
6          System.out.println("哞...");
7      }
8  }
```

3. 创建 Sheep 类

创建一个 Sheep 类，与 Cow 类相似，同样实现了 Animal 接口，并实现了抽象方法 call()，如文件 3-33 所示。

文件 3-33 Sheep.java

```
1  package com.itheima.example14;
2  //定义 Sheep 类实现 Animal 接口
3  class Sheep implements Animal {
4      //实现 shout()方法
5      public void call() {
6          System.out.println("咩...");
7      }
8  }
```

4. 创建测试类

创建一个测试类 Example14，在该类中通过父类类型变量引用不同的子类对象，当调用 animalCall()方法时，将父类引用的两个不同子类对象分别传入该方法，如文件 3-34 所示。

文件 3-34 Example14.java

```
1  package com.itheima.example14;
2  public class Example14 {
3      public static void main(String[] args) {
4          Animal a1=new Cow();      //创建 Cow 对象，使用 Animal 类型的变量 a1 引用
5          Animal a2=new Sheep();    //创建 Sheep 对象，使用 Animal 类型的变量 a2 引用
6          animalCall(a1);           //调用 animalCall()方法，将 a1 作为参数传入
7          animalCall(a2);           //调用 animalCall()方法，将 a2 作为参数传入
8      }
9      //定义静态的 animalCall()方法，接收一个 Animal 类型的参数
10     public static void animalCall(Animal a) {
11         a.call();                 //调用实际参数的 call()方法
12     }
13 }
```

运行结果如图 3-16 所示。

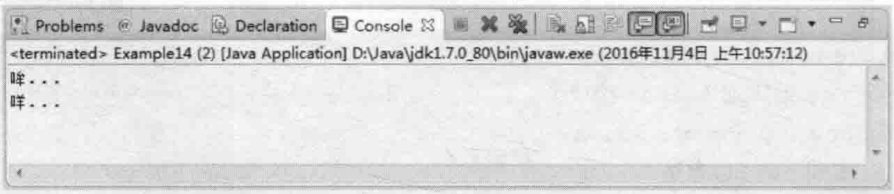


图 3-16 运行结果

从图 3-16 的运行结果中可以看出，两个不同的子类对象调用 call()时，分别输出了“咩...”和“咩...”。

3.8.2　对象的类型转换

在多态的学习中，涉及将子类对象当作父类类型使用的情况，此种情况在 Java 的语言环境中称为“向上转型”，例如下面两行代码：

```
Animal a1=new Cow();          //将 Cow 对象当作 Animal 类型使用
Animal a2=new Sheep();        //将 Sheep 对象当作 Animal 类型使用
```

需要注意的是，将子类对象当作父类对象使用时不需要任何显式转换，但此时不能通过父类变量调用子类中的特有方法。

下面通过案例演示对象的类型转换错误的情况。

1. 创建 Animal 接口

创建一个 Animal 接口，并在接口中定义一个抽象方法 call()，如文件 3-35 所示。

文件 3-35　Animal.java

```
1  package com.itheima.example15;
2  //定义 Animal 类接口
3  interface Animal {
4      void call();          //定义抽象方法 call()
5  }
```

2. 创建 Cow 类

创建一个 Cow 类，使其实现 Animal 接口，并在 Animal 接口重写抽象方法 call()，如文件 3-36 所示。

文件 3-36　Cow.java

```
1  package com.itheima.example15;
2  //定义 Cow 类实现 Animal 接口
3  public class Cow implements Animal {
4      //实现 call()方法
5      public void call() {
6          System.out.println("哞...");
7      }
8  }
```

3. 创建 Sheep 类

创建一个 Sheep 类，与 Cow 类相似，同样实现了 Animal 接口，并重写了抽象方法 call()，如文件 3-37 所示。

文件 3-37　Sheep.java

```
1  package com.itheima.example15;
2  //定义 Sheep 类实现 Animal 接口
```

```
3  public class Sheep implements Animal {
4      //实现 call()方法
5      public void call() {
6          System.out.println("咩...");
7      }
8  }
```

4. 创建测试类

创建测试类 Example15，在该类中创建一个 Sheep 对象，该对象作为参数的形式传入 animalCall()方法中。然后在 animalCall()方法中，将 Sheep 对象强制转换成 Cow 类型，并调用 call()方法，如文件 3-38 所示。

文件 3-38　Example15.java

```
1  package com.itheima.example15;
2  //定义测试类
3  public class Example15 {
4      public static void main(String[] args) {
5          Sheep s=new Sheep();          //创建 Sheep 类型的实例对象
6          animalCall(s);                //调用 animalCall()方法，将 Sheep 作为参数传入
7      }
8      //定义静态方法 animalShout()，接收一个 Animal 类型的参数
9      public static void animalCall(Animal a) {
10         Cow c=(Cow) a;                //将 a 对象强制转换成 Cow 类型
11         c.call();                     //调用 Cow 的 call()方法
12     }
13 }
```

在文件 3-38 中，首先创建了一个 Sheep 对象，该对象作为参数的形式传入 animalCall()方法中。然后在 animalCall()方法中，将 Sheep 对象强制转换成 Cow 类型，并调用 call()方法，运行结果如图 3-17 所示。

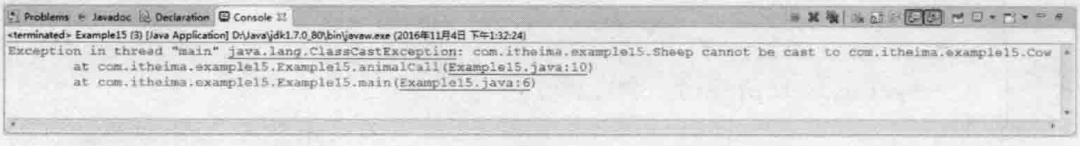


图 3-17　运行结果

从图 3-17 的运行报错中可以看出，提示 Sheep 类型不能转换成 Cow 类型。出错的原因是，在调用 animalCall()方法时传入一个 Sheep 对象，在强制类型转换时，Animal 类型的变量无法强制转换为 Cow 类型。

针对这种情况，Java 提供了一个关键字 instanceof，它可以判断一个对象是否为某个类（或接口）的实例或者子类实例，语法格式如下：

```
对象(或者对象引用变量) instanceof 类(或接口)
```

接下来对文件 3-38 的 animalCall()方法进行修改，具体代码如下：

```
public static void animalCall(Animal a) {
    if (a instanceof Cow) {                    //判断 a 对象是否是 Cow 类的实例对象
        Cow c=(Cow)a;                          //将 a 对象强制转换为 Cow 类型
        c.call();                              //调用 Cow 的 call()方法
    } else {
        System.out.println("这个动物不是牛!");
    }
}
```

运行结果如图 3-18 所示。

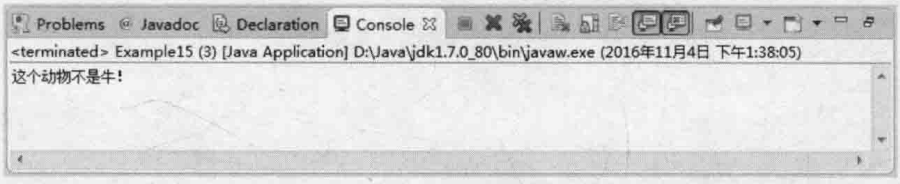


图 3-18　运行结果

从上述代码中可以看出，可以使用 instanceof 关键字判断 animalCall()方法中传入的对象的类型，如果是 Cow 类型则进行强制类型转换，否则就输出“这个动物不是牛!”。在该文件中，由于传入的对象为 Sheep 类型，因此出现图 3-18 的运行结果。

3.8.3　匿名内部类

在前面的多态讲解中，如果方法的参数被定义为一个接口类型，那么就需要定义一个类实现接口，并根据该类进行对象实例化。除此之外，还可以使用匿名内部类实现接口。当程序中使用匿名内部类时，在定义匿名内部类的地方往往直接创建该类的一个对象。

为了便于初学者理解，首先看一下匿名内部类的格式，具体如下：

```
new 父类(参数列表) 或 父接口(){
    //匿名内部类实现部分
}
```

为了让读者能更容易地理解匿名内部类的使用，下面通过案例学习匿名内部类的使用。

1. 创建 Animal 接口

创建一个 Animal 接口，并在接口中定义一个抽象方法 call()，如文件 3-39 所示。

文件 3-39　Animal.java

```
1  package com.itheima.example16;
2  //定义动物类接口
3  interface Animal {                  //定义动物类接口
4      void call();                    //定义方法 call()
5  }
```

2. 创建测试类

创建一个测试类 Example16，在该类中通过匿名内部类的形式实现接口，如文件 3-40 所示。

文件 3-40 Example16.java

```
1  package com.itheima.example16;
2  public class Example16 {
3      public static void main(String[] args) {
4          //定义匿名内部类作为参数传递给 animalCall()方法
5          animalCall(new Animal() {
6              //实现 call()方法
7              public void call() {
8                  System.out.println("哞...");
9              }
10         });
11     }
12     //定义静态方法 animalCall()
13     public static void animalCall(Animal a) {
14         a.call();              //调用传入对象 a 的 call()方法
15     }
16 }
```

在文件 3-40 中，调用 animalCall()方法时，在方法的参数位置写上 new Animal(){}，这相当于创建了一个实例对象，并将对象作为参数传递给 animalCall()方法。在 new Animal()后面有一对大括号，表示创建了 Animal 的子类对象，该子类是匿名的，运行结果如图 3-19 所示。

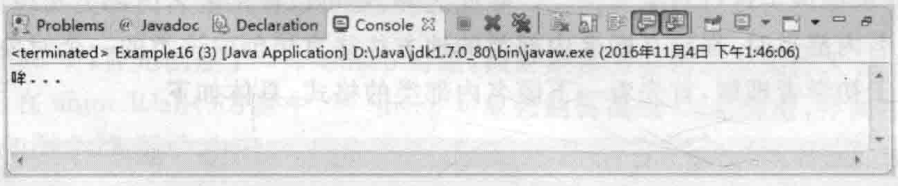


图 3-19 运行结果

从图 3-19 的运行结果中可以看出，使用匿名内部类的方式实现了 Animal 接口，调用 call()方法并输出结果为“哞...”。

3.9 异常

3.9.1 异常的概念

在程序运行的过程中，可能会发生异常(Exception)情况，例如运行时内存溢出、磁盘空间不足、网络中断等。针对此类状况，Java 中提供了异常处理机制，以异常类的形式对这些不正常的情况进行封装，通过异常处理机制对程序代码发生的各种问题进行有针对性的处理。下面通过案例了解一下什么是异常，如文件 3-41 所示。

文件 3-41　Example17.java

```
1  package com.itheima.example17;
2  public class Example17 {
3      public static void main(String[] args) {
4          int res=calculate(5, 0);          //调用 calculate()方法
5          System.out.println(res);
6      }
7      //下面的方法实现了两个整数相除
8      public static int calculate(int a, int b) {
9          int res=a / b;                    //定义一个变量 res 记录两个数相除的结果
10         return res;                       //将结果返回
11     }
12 }
```

运行结果如图 3-20 所示。

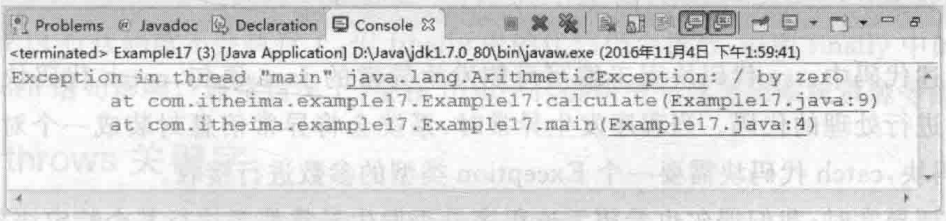


图 3-20　运行结果

从图 3-20 的运行结果中可以看出，程序出现了算术异常(ArithmeticException)，这个异常是由于在第 4 行代码调用 divide()方法时传入了参数 0，而在 calculate()方法中，运算时出现了被 0 除的情况。程序出现异常后导致程序立即结束，无法继续向下执行。

3.9.2　常见的异常类

在第 3.9.1 节中产生的 ArithmeticException 异常只是 Java 异常体系中的一种，Java 还提供了大量的异常类，这些异常类都是 java.lang.Throwable 类的子类。

Throwable 类的继承体系如图 3-21 所示。

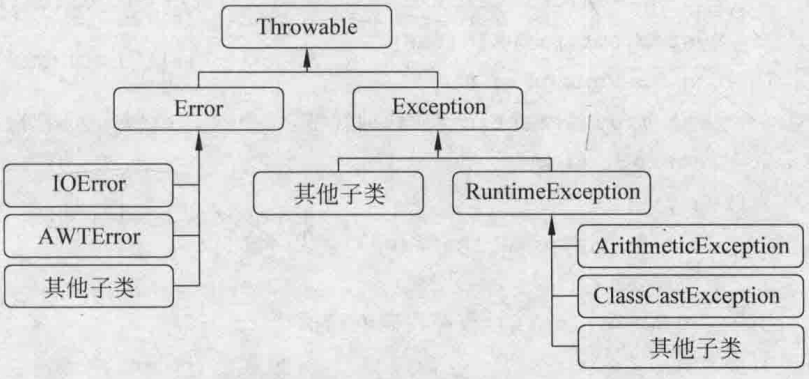


图 3-21　Throwable 体系架构

通过图3-21可以看出，Throwable有两个直接子类Error和Exception，其中Error表示程序代码中出现的错误，Exception表示程序代码中出现的异常，二者的区别在于，错误表示系统内部错误或资源耗尽的错误，仅靠程序本身是不能恢复执行的，而异常表示程序本身可以处理的错误。

3.9.3 try...catch和finally

在文件3-41中，由于出现了异常导致程序代码无法继续向下执行。为了解决这样的问题，Java中提供了一种对异常进行处理的方式，即异常捕获。异常捕获通常使用try…catch语句，具体语法格式如下：

```
try{
    //程序代码块
}catch(异常类型(Exception类或其子类) 变量名){
    //对异常的处理
}
```

在上述代码中，try代码块用于编写可能发生异常的Java语句，catch代码块用于编写针对异常进行处理的代码。当程序发生异常时，系统会将异常信息封装成一个对象传递给catch代码块，catch代码块需要一个Exception类型的参数进行接收。

在处理异常时，我们偶尔也希望无论程序是否发生异常都要执行某个特定语句，此时就需要在try...catch后面加上一个finally语句，无论程序是否发生异常，finally语句中的内容都会执行。接下来使用try...catch和finally语句对文件3-41中出现的异常进行捕获，如文件3-42所示。

文件3-42 Example18.java

```
1  package com.itheima.example18;
2  public class Example18 {
3      public static void main(String[] args) {
4          //下面的代码定义了一个try...catch...finally语句用于捕获异常
5          try {
6              int res=calculate(5, 0);              //调用calculate()方法
7              System.out.println(res);
8          } catch (Exception e) {
9              System.out.println("捕获的异常是："+e.getMessage());
10             return;
11         } finally {
12             System.out.println("finally代码块");
13         }
14         System.out.println("程序代码继续执行...");
15     }
16     //下面的方法实现了两个整数相除
```

```
17      public static int calculate(int a, int b) {
18          int res=a / b;          //定义一个变量 res 记录两个数相除的结果
19          return res;             //将结果返回
20      }
21  }
```

运行结果如图 3-22 所示。

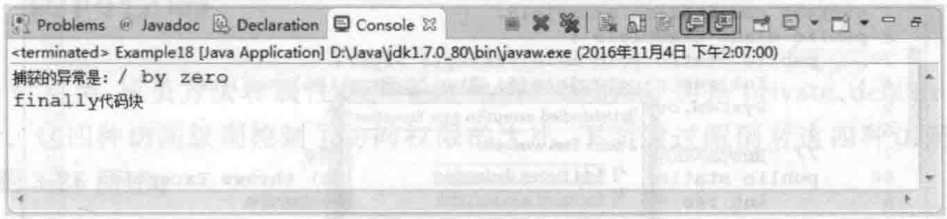


图 3-22　运行结果

在文件 3-42 的 catch 代码块中增加了一个 return 语句，用于终止当前方法的执行，此时程序的第 14 行代码就不会执行了，但 finally 中的代码仍会执行，说明 finally 中的代码并不会被 return 语句影响。通常情况下，会在 finally 代码块中完成关闭系统资源等操作。

3.9.4　throws 关键字

在第 3.9.3 节中已经了解到异常的发生可以使用 try...catch 语句的方式进行处理，除了此方法以外，在 Java 中还提供了另一种对异常的处理方式，即抛出异常。抛出异常使用 throws 关键字对外声明该方法有可能发生的异常，这样在调用方法时，就很清楚该方法是否有异常，并必须对异常进行针对性的处理，否则编译失败。具体语法格式如下：

```
修饰符 返回值类型 方法名([参数 1,参数 2...])throws ExceptionType1[,ExceptionType2...]{
}
```

从上述语法格式中可以看出，throws 关键字需要写在方法声明的后面，并且在 throws 后面需要声明方法中发生异常的类型，通常将这种做法称为方法声明抛出异常。接下来对文件 3-42 进行修改，在 calculate()方法上声明抛出异常，如文件 3-43 所示。

文件 3-43　Example19.java

```
1   package com.itheima.example19;
2   public class Example19 {
3       public static void main(String[] args) {
4           int res=calculate(6, 3);          //调用 calculate()方法
5           System.out.println(res);
6       }
7       //下面的方法实现了两个整数相除，并使用 throws 关键字声明抛出异常
8       public static int calculate(int a, int b) throws Exception {
```

```
9           int res=a / b;              //定义一个变量 res 记录两个数相除的结果
10          return res;                 //将结果返回
11      }
12  }
```

在 Eclipse 中，编译程序报错，结果如图 3-23 所示。

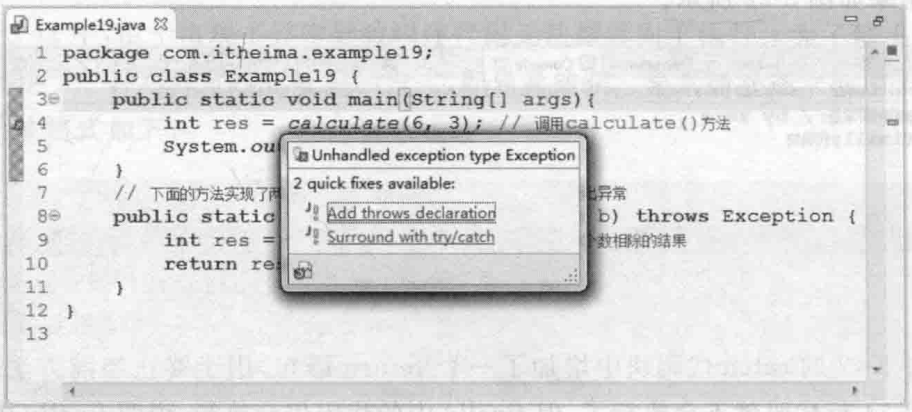


图 3-23 编译结果

在图 3-23 中，第 4 行代码调用 calculate()方法时，虽然程序在运行时不会出现被 0 除的异常，但是由于定义 calculate()方法时声明抛出了异常，调用者在调用 calculate()方法时就必须进行处理，否则就会出现编译报错。下面对文件 3-43 进行修改，如文件 3-44 所示。

文件 3-44 Example20.java

```
1   package com.itheima.example20;
2   public class Example20 {
3       public static void main(String[] args) throws Exception {
4           int res=calculate(6, 3);            //调用 calculate()方法
5           System.out.println(res);
6       }
7       //下面的方法实现了两个整数相除，并使用 throws 关键字声明抛出异常
8       public static int calculate(int x, int y) throws Exception {
9           int res=x / y;                      //定义一个变量 res 记录两个数相除的结果
10          return res;                         //将结果返回
11      }
12  }
```

运行结果如图 3-24 所示。

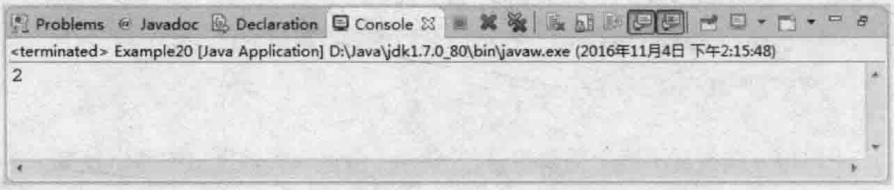


图 3-24 运行结果

在文件 3-44 中，使用 throws 关键字将异常抛出，这样程序就可以通过编译，运行后正确地输出运行结果“2”。

注意：在程序开发中有两种常见异常，一种是编译时的异常，另一种是运行时的异常，区别是编译时产生的异常必须进行处理，否则编译不通过，而运行时的异常是在程序运行中产生的，即使不进行异常处理也能通过编译。

3.10　访问控制

Java 针对类、成员方法和属性提供了四种访问级别，分别是 private、default、protected 和 public。这四种访问级别控制了访问权限的大小，下面通过图例对这四种访问级别进行排序，如图 3-25 所示。

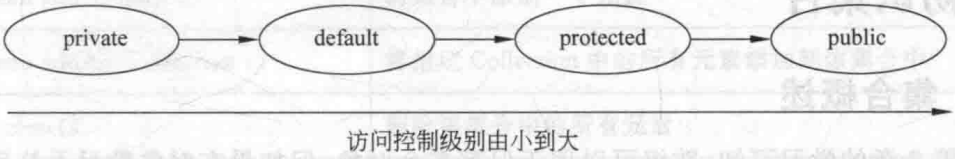


图 3-25　访问级别

- private(类访问级别)：表示私有的，使用 private 修饰的成员只能被该类的其他成员访问，其他类无法直接访问。
- default(包访问级别)：若没有访问修饰符，则系统默认使用 default。该类可以被本类包中的所有类访问。
- protected(子类访问级别)：表示受保护的，该类可以被同一包下的其他类访问，也能被不同包下的子类访问。
- public(公共访问级别)：表示公有的，该类或者类中的成员可以被所有的类访问。

接下来通过一个表将这四种访问级别更加直观地表示出来，如表 3-1 所示。

表 3-1　访问控制级别

访问范围	private	default	protected	public
同一类中	√	√	√	√
同一包中		√	√	√
子类中			√	√
全局范围				√

第4章 集 合 类

面向对象语言都是以对象的形式体现的，所以为了方便对多个对象进行存取操作，Sun公司在Java中提供了集合类。本章将针对Java中的集合类进行详细讲解。

4.1 初识集合

4.1.1 集合概述

通过第2章的学习可知，数组可以用于保存多个对象，但如果在对象数目无法确定的情况下，数组是不适用的，因为数组的长度不可变。为了保存这些数目不确定的对象，JDK中提供了集合类(又称容器类)，这些类可以存储任意类型的对象，并且长度可变。所有的集合类都位于java.util包中，在使用时需要导入该包，否则会出现异常。

集合按照存储结构可以分为两大类，分别是单列集合(Collection)和双列集合(Map)，这两种集合的特点如下所示。

- Collection：单列集合类的根接口，用于存储一系列符合某种规则的元素，它有两个重要的子接口，分别是List和Set。其中，List的特点是元素有序，元素可重复；Set的特点是元素无序，而且不可重复。List接口的主要实现类有ArrayList和LinkedList，Set接口的主要实现类有HashSet和TreeSet。
- Map：双列集合类的根接口，用于存储具有键(Key)、值(Value)映射关系的元素，每个元素都包含一对键值，在使用Map集合时可以通过指定的Key找到对应的Value，例如根据一个员工的工号就可以找到对应的员工。Map接口的主要实现类有HashMap和TreeMap。

为了让读者更清楚地了解集合之间的关系，接下来通过一张图描述整个集合类的继承体系，如图4-1所示。

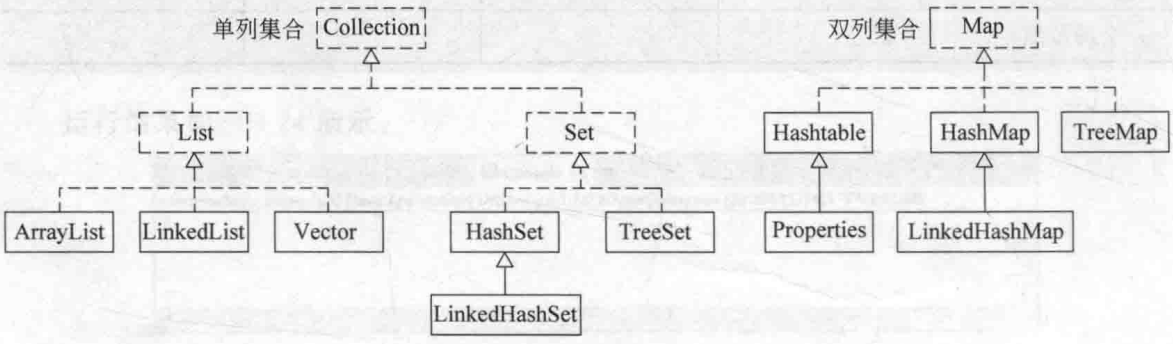


图4-1 集合框架体系

图 4-1 列出了程序中常用的一些集合类，其中，虚线框中填写的都是接口类型，而实线框中填写的都是具体的实现类。本章将针对图中的常用集合进行讲解。

4.1.2　Collection 接口简介

Collection 是所有单列集合的父接口，因此 Collection 接口中定义了很多单列集合(List 和 Set)通用的方法，这些方法分别用于在集合中添加元素、删除元素、获取集合中元素的个数等，具体如表 4-1 所示。

表 4-1　Collection 接口的常用方法

方法声明	功能描述
boolean add(Object o)	向集合中添加一个元素
boolean addAll(Collection c)	将指定 Collection 中的所有元素添加到该集合中
void clear()	删除该集合中的所有元素
boolean remove(Object o)	删除该集合中指定的元素
boolean removeAll(Collection c)	删除指定集合中的所有元素
boolean isEmpty()	判断该集合是否为空
boolean contains(Object o)	判断该集合中是否包含某个元素
boolean containsAll(Collection c)	判断该集合中是否包含指定集合中的所有元素
Iterator iterator()	返回在该集合的元素上进行迭代的迭代器(Iterator)，用于遍历该集合所有元素
int size()	获取该集合元素的个数

4.2　List 接口

4.2.1　List 接口简介

List 接口继承自 Collection 接口，是单列集合的一个重要分支，习惯性地将 List 接口的对象称为 List 集合。在 List 集合中可以出现相同的元素，所有的元素是以一种线性方式进行存储的，因此使用此集合能够精确地控制每个元素插入的位置，用户能够使用索引访问 List 中的元素。另外，List 集合还有一个特点就是元素有序，即元素的存入顺序和取出顺序一致。

List 与 Collection 不同的是，在 List 接口中大量扩充了 Collection 接口，拥有比 Collection 接口中更多的方法定义，表 4-2 列出了 List 接口中常用的方法。

表 4-2 List 集合的常用方法

方法声明	功能描述
void add(int index,Object element)	将元素 element 插入到 List 集合的 index 处
boolean addAll(int index,Collection c)	将集合 c 所包含的所有元素插入到 List 集合的 index 处
Object get(int index)	返回集合索引 index 处的元素
Object remove(int index)	删除 index 索引处的元素
Object set(int index, Object element)	将索引 index 处元素替换成 element 对象,并将替换后的元素返回
int indexOf(Object o)	返回对象 o 在 List 集合中出现的位置索引
int lastIndexOf(Object o)	返回对象 o 在 List 集合中最后一次出现的位置索引
List subList(int fromIndex, int toIndex)	返回从索引 fromIndex(包括)到 toIndex(不包括)处所有元素集合组成的子集合

4.2.2 ArrayList 集合

ArrayList 是 List 接口的一个实现类,在程序开发中经常使用。在 ArrayList 内部封装了一个长度可变的数组,当存入的元素超过数组长度时,ArrayList 会在内存中分配一个更大的数组存储这些元素,因此可以将 ArrayList 集合看作一个长度可变的数组。

ArrayList 集合中大多数的方法都是从父接口 Collection 和 List 继承的,其中 add()方法和 get()方法用于实现元素的存取。下面通过案例学习 ArrayList 集合如何存取元素,如文件 4-1 所示。

文件 4-1 Example01.java

```
1  package com.itheima.example;
2  import java.util.*;
3  public class Example01 {
4      public static void main(String[] args) {
5          ArrayList list=new ArrayList();      //创建 ArrayList 集合
6          list.add("zhangsan");                //向集合中添加元素
7          list.add("lisi");
8          list.add("wangwu");
9          list.add("zhaoliu");
10         System.out.println("集合的长度是: "+list.size());
11         System.out.println("第 2 个元素是: "+list.get(2));
12     }
13 }
```

运行结果如图 4-2 所示。

由于 ArrayList 集合的底层使用一个数组保存元素,在增加或删除指定位置的元素时,会导致创建新的数组,极大地降低了效率,因此该集合不适合进行大量的增删操作。

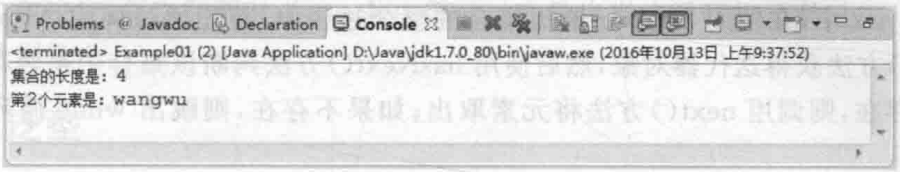


图 4-2　运行结果

但这种数组的结构允许程序通过索引的方式访问元素，因此 ArrayList 集合的优势是查找元素效率高。

4.2.3　Iterator 接口

在使用集合类时，一定会遇到需要遍历所有元素的情况。为此，JDK 中专门提供了一个接口 Iterator。Iterator 接口也是集合框架中的一员，它与 Collection 和 Map 接口不同的是，Collection 接口与 Map 接口主要用于存储元素，而 Iterator 主要用于遍历 Collection 和其子类中的元素，因此 Iterator 对象也被称为迭代器。

下面通过案例学习如何使用 Iterator 迭代器遍历 ArrayList 集合中的元素，如文件 4-2 所示。

文件 4-2　Example02.java

```
1  package com.itheima.example;
2  import java.util.*;
3  public class Example02 {
4      public static void main(String[] args) {
5          ArrayList list=new ArrayList();   //创建 ArrayList 集合
6          list.add("zhangsan");             //向该集合中添加元素
7          list.add("lisi");
8          list.add("wangwu");
9          list.add("zhaoliu");
10         Iterator it=list.iterator();     //得到 Iterator 对象
11         while (it.hasNext()) {           //判断 ArrayList 集合中是否有下一个元素
12             Object obj=it.next();        //取出 ArrayList 集合中的元素
13             System.out.println(obj);
14         }
15     }
16 }
```

运行结果如图 4-3 所示。

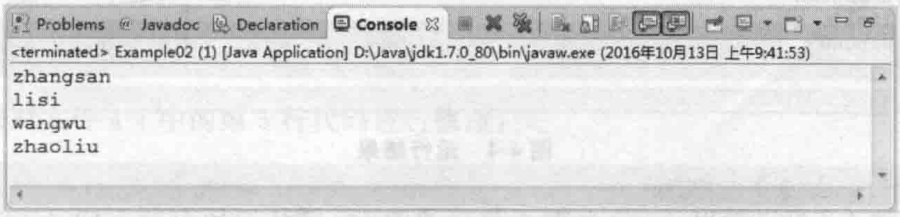


图 4-3　运行结果

在文件 4-2 中，在使用 Iterator 遍历集合的整个过程中，首先通过调用 ArrayList 集合的 iterator()方法获得迭代器对象，然后使用 hasNext()方法判断该集合中是否存在下一个元素，如果存在，则调用 next()方法将元素取出；如果不存在，则跳出 while 循环停止遍历元素。

4.2.4 foreach 循环

虽然 Iterator 可以用于遍历集合中的元素，但在写法上比较麻烦，为了能够简化书写，从 JDK 5.0 开始，Java 提供了 foreach 循环。foreach 循环是一种更加简洁的 for 循环，也称增强 for 循环。foreach 循环用于遍历集合或者数组中的元素，具体语法格式如下：

```
for(集合内储存类型 变量名 : 集合的变量名) {
    执行语句部分
}
```

从上面的格式可以看出，与普通 for 循环相比，foreach 循环在遍历集合时，不需要知道集合的长度，也无须知道索引，foreach 能自动遍历集合中的每个元素。

下面通过案例对 foreach 循环进行详细讲解，如文件 4-3 所示。

文件 4-3 Example03.Java

```
1  package com.itheima.example;
2  import java.util.*;
3  public class Example03 {
4      public static void main(String[] args) {
5          ArrayList list=new ArrayList();  //创建 ArrayList 集合
6          list.add("zhangsan");            //向 ArrayList 集合中添加元素
7          list.add("lisi");
8          list.add("wangwu");
9          for (Object obj : list) {        //使用 foreach 循环遍历 ArrayList 集合
10             System.out.println(obj);     //取出并打印 ArrayList 集合中的元素
11         }
12     }
13 }
```

运行结果如图 4-4 所示。

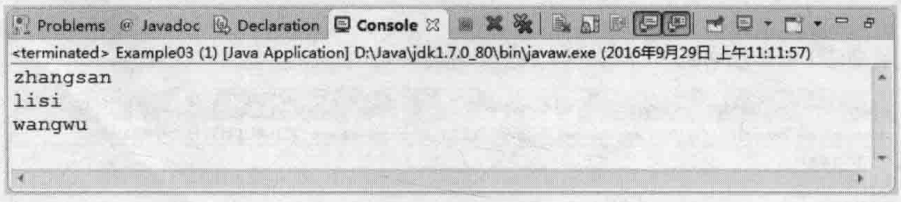


图 4-4 运行结果

通过文件 4-3 可以看出，foreach 循环与普通的 for 循环遍历有所不同，foreach 循环没有循环条件，也没有迭代语句，所有这些工作都交给 JVM 执行。foreach 循环的次数由集合

中元素的个数决定，每次循环时，foreach 中都通过变量将当前循环的元素记住，从而将集合中的元素分别输出。

4.2.5　泛型

通过前面几个小节的学习，读者已经了解到集合可以存储任何类型的对象，但是存储一个对象到集合后，集合会“忘记”这个对象的类型，当该对象从集合中取出时，这个对象的编译类型就变成了 Object 类型。那么在取出元素时，如果进行强制类型转换就很容易抛出异常。下面通过案例演示这种情况，如文件 4-4 所示。

文件 4-4　Example04.java

```
1  package com.itheima.example;
2  import java.util.*;
3  public class Example04 {
4      public static void main(String[] args) {
5          ArrayList list=new ArrayList();          //创建 ArrayList 集合
6          list.add("zhangsan");                    //添加字符串
7          list.add("lisi");
8          list.add(18);                            //添加 Integer 类型的数据
9          for (Object obj : list) {                //遍历集合
10             String str=(String) obj;             //强制转换成 String 类型
11         }
12     }
13 }
```

运行结果如图 4-5 所示。

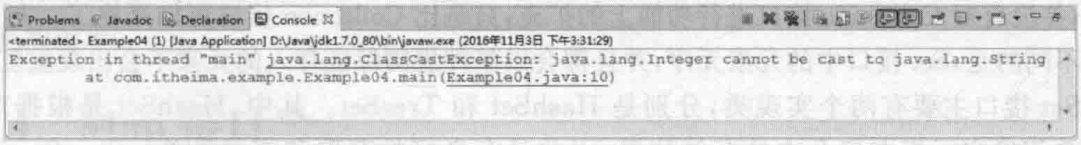


图 4-5　运行结果

在文件 4-4 中，向 List 集合存入了三个元素，分别是两个字符串类型和一个 int 类型。在取出这些元素时，都将它们强制转换为 String 类型，由于 Integer 对象不能转换为 String 类型，因此在程序运行时抛出了类转换异常。为了解决这个问题，Java 引入了“参数化类型(parameterized type)”这个概念，即泛型，它可以限定方法操作的数据类型，在定义集合类时，使用“<参数化类型>”的方式指定该类中方法操作的数据类型，具体格式如下：

```
ArrayList<参数化类型>list=new ArrayList<参数化类型>();
```

接下来对文件 4-4 中的第 5 行代码进行修改：

```
ArrayList<String>list=new ArrayList<String>();          //创建集合对象并指定泛型为 String
```

上面这种写法限定了 ArrayList 集合只能存储 String 类型元素，改写后的程序在

Eclipse 中编译时会出现错误提示，如图 4-6 所示。

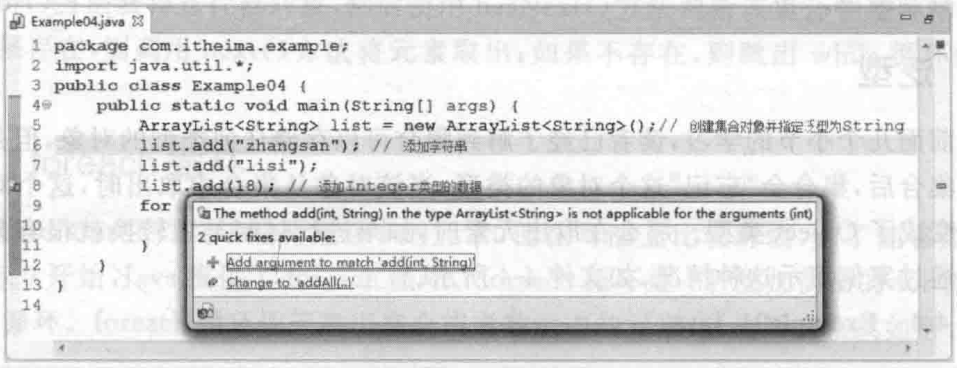


图 4-6 运行结果

在图 4-6 中，程序编译报错的原因是修改后的代码限定了集合元素的数据类型，ArrayList<String>这样的集合只能存储 String 类型的元素，程序在编译时，编译器检查出 Integer 类型的元素与 List 集合的规定类型不匹配，编译不通过。这样就可以在编译时解决错误，避免程序在运行时发生错误。

需要注意的是，在使用泛型后，每次遍历集合元素时，可以指定元素类型为 String，而不是 Object，这样就避免了在程序中进行强制类型转换。

4.3 Set 接口

4.3.1 Set 接口简介

Set 接口和 List 接口一样，同样继承自 Collection 接口，它与 Collection 接口中的方法基本一致，并没有对 Collection 接口进行功能上的扩充，只是比 Collection 接口更加严格了。与 List 接口不同的是，Set 接口中的元素无序，并且都会以某种规则保证存入的元素不会出现重复。

Set 接口主要有两个实现类，分别是 HashSet 和 TreeSet。其中，HashSet 是根据对象的哈希值确定元素在集合中的存储位置，因此具有良好的存取和查找性能。TreeSet 则是以二叉树的方式存储元素，它可以实现对集合中的元素进行排序。下面以 HashSet 为例，对 Set 集合进行详细讲解。

4.3.2 HashSet 集合

HashSet 是 Set 接口的一个实现类，HashSet 按 Hash 算法存储集合中的元素，因此具有很好的存取和查找性能，并且存储的元素是无序的和不可重复的。下面通过案例演示 HashSet 集合的用法，如文件 4-5 所示。

文件 4-5 Example05.java

```
1  package com.itheima.example;
2  import java.util.*;
3  public class Example05 {
4      public static void main(String[] args) {
```

```
5           HashSet hashSet=new HashSet();   //创建 HashSet 集合
6           hashSet.add("zhangsan");         //向该 HashSet 集合中添加元素
7           hashSet.add("lisi");
8           hashSet.add("wangwu");
9           hashSet.add("lisi");             //向该 HashSet 集合中添加重复元素
10          Iterator it=hashSet.iterator(); //得到 Iterator 对象
11          while (it.hasNext()) {           //通过 while 循环,判断集合中是否有元素
12              Object obj=it.next();     //如果有元素,则使用迭代器的 next()方法获取元素
13              System.out.println(obj);
14          }
15      }
16  }
```

运行结果如图 4-7 所示。

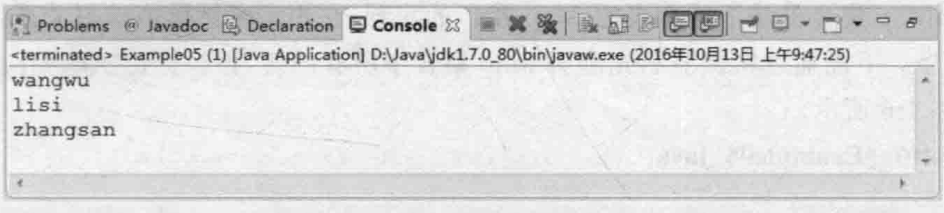


图 4-7　运行结果

HashSet 集合之所以能确保不出现重复的元素，是因为在向 Set 中添加对象时，会先调用此对象所在类的 hashCode()方法，计算此对象的哈希值，此哈希值决定了此对象在 Set 中的存储位置。若此位置之前没有对象存储，则将这个对象直接存储到此位置。若此位置已有对象存储，再通过 equals()比较这两个对象是否相同，如果相同，则后一个对象就不能再添加进来。

4.4　Map 接口

4.4.1　Map 接口简介

Map 集合是一种双列集合，集合中的每个元素都包含一个键对象 Key 和一个值对象 Value，键和值是一一对应的关系，称为映射。也就是说，根据键就能找到对应的值，类似于生活中一张身份证对应一个人一样。为了便于对 Map 集合的操作，Map 集合中提供了很多方法，如表 4-3 所示。

表 4-3　Map 集合中的方法

方法声明	功能描述
void add(int index,Object element)	将元素 element 插入到 List 集合的 index 处
boolean addAll(int index,Collection c)	将集合 c 所包含的所有元素插入到 List 集合的 index 处

续表

方法声明	功能描述
Object get(int index)	返回集合索引 index 处的元素
Object remove(int index)	删除 index 索引处的元素
Object set(int index, Object element)	将索引 index 处元素替换成 element 对象，并将替换后的元素返回
int indexOf(Object o)	返回对象 o 在 List 集合中出现的位置索引
int lastIndexOf(Object o)	返回对象 o 在 List 集合中最后一次出现的位置索引
List subList(int fromIndex, int toIndex)	返回从索引 fromIndex(包括)到 toIndex(不包括)处所有元素集合组成的子集合

4.4.2 HashMap 集合

HashMap 集合是基于哈希表的 Map 接口的实现，它用于存储键值映射关系，但不保证映射的顺序。下面通过案例演示先遍历 Map 集合中所有的键，再根据键获取相应的值的方式，如文件 4-6 所示。

文件 4-6 Example06.java

```
1  package com.itheima.example;
2  import java.util.*;
3  public class Example06 {
4      public static void main(String[] args) {
5          HashMap hashMap=new HashMap();                    //创建 HashMap 集合
6          hashMap.put("001", "zhangsan");                   //存储键和值
7          hashMap.put("002", "lisi");
8          hashMap.put("003", "wangwu");
9          Set keySet=hashMap.keySet();                      //获取键的集合
10         Iterator it=keySet.iterator();                    //迭代键的集合
11         while (it.hasNext()) {
12             Object key=it.next();
13             Object value=hashMap.get(key);                //获取每个键所对应的值
14             System.out.println(key+"="+value);
15         }
16     }
17 }
```

运行结果如图 4-8 所示。

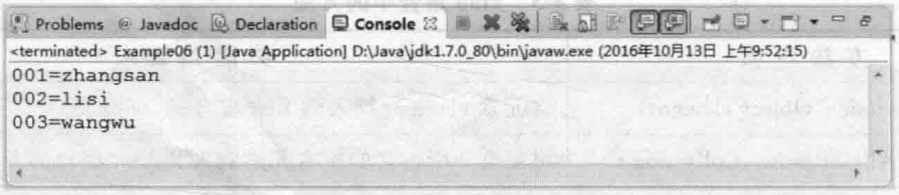


图 4-8 运行结果

在文件 4-6 中，首先调用 HashMap 集合的 KeySet()方法，获得存储 Map 中所有键的 Set 集合，然后通过 Iterator 迭代 Set 集合的每一个元素，即每一个键，最后通过调用 get (String key)方法，根据键获取对应的值。

Map 集合的另外一种遍历方式是先获取集合中的所有映射关系，然后从映射关系中取出键和值。下面通过案例演示这种遍历方式，如文件 4-7 所示。

文件 4-7　Example07.java

```
1  package com.itheima.example;
2  import java.util.*;
3  public class Example07 {
4      public static void main(String[] args) {
5          HashMap hashMap=new HashMap();                    //创建 HashMap 集合
6          hashMap.put("001", "zhangsan");                   //存储键和值
7          hashMap.put("002", "lisi");
8          hashMap.put("003", "wangwu");
9          Set entrySet=hashMap.entrySet();
10         Iterator it=entrySet.iterator();                  //获取 Iterator 对象
11         while (it.hasNext()) {
12             Map.Entry entry= (Map.Entry)it.next();        //获取集合中键值对映射关系
13             Object key=entry.getKey();                    //获取 Entry 中的键
14             Object value=entry.getValue();                //获取 Entry 中的值
15             System.out.println(key+"="+value);
16         }
17     }
18 }
```

运行结果如图 4-9 所示。

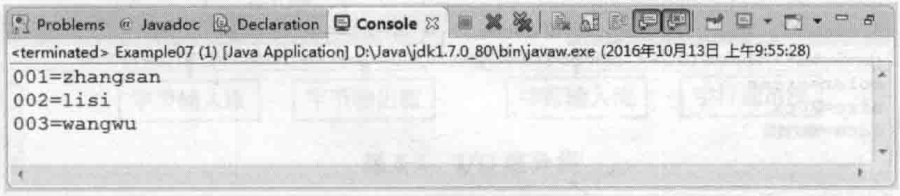


图 4-9　运行结果

在文件 4-7 中，首先调用 Map 对象的 entrySet()方法获得存储在 Map 中所有映射的 Set 集合，这个集合中存放了 Map.Entry 类型的元素(Entry 是 Map 内部接口)，每个 Map.Entry 对象代表 Map 中的一个键值对，然后迭代 Set 集合，获得每一个映射对象，并分别调用映射对象的 getKey()和 getValue()方法获取键和值。

4.4.3　Properties 集合

Properties 主要用于存储字符串类型的键和值，由于 Properties 类实现了 Map 接口，因此，Properties 类本质上是一种简单的 Map 集合。在实际开发中，经常使用 Properties 集合存取应用的配置项。假设有一个文本编辑工具，要求文本的字体为微软雅黑，字号为 20px，

字体颜色为 green，其配置项应该如下所示：

```
face=微软雅黑
size=20px
color=green
```

在程序中可以使用 Properties 集合对这些配置项进行存取，下面通过案例学习 Properties 集合的使用，如文件 4-8 所示。

文件 4-8　Example08.java

```
1  package com.itheima.example;
2  import java.util.Properties;
3  import java.util.Set;
4  public class Example08 {
5      public static void main(String[] args) {
6          Properties p=new Properties();          //创建 Properties 集合
7          p.setProperty("face", "微软雅黑");
8          p.setProperty("size", "20px");
9          p.setProperty("color", "green");
10         Set<String>names=p.stringPropertyNames();     //获取 Set 集合对象
11         for(String key : names) {
12             String value=p.getProperty(key);
13             System.out.println(key+"="+value);
14         }
15     }
16 }
```

运行结果如图 4-10 所示。

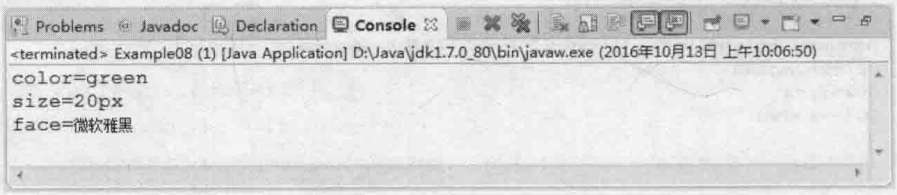


图 4-10　运行结果

在文件 4-8 的 Properties 类中，针对字符串的存取提供了两个专用的方法：setProperty() 和 getProperty()。其中，setProperty()方法用于将配置项的键和值添加到 Properties 集合中，getProperty()方法用于根据键获取对应的值。在第 10 行代码中通过调用 Properties 的 stringPropertyNames()方法得到一个所有键的 Set＜String＞集合，然后在遍历所有的键时，通过调用 getProperty()方法获得键所对应的值。

第5章 输入/输出

输入/输出(I/O)是指程序与外部设备或者计算机之间的交互操作,例如从键盘输入数据,在显示器显示数据等,通过输入/输出操作可以从外界接收信息,或者是把信息传递给外界。在Java中,将这种不同输入/输出设备(键盘、内存、网络、显示器)之间的数据传输抽象为“流”,通过“流”的方式在输入/输出设备之间进行数据传递。由于Java中的流都位于java.io包中,因此称为I/O流。本章将针对I/O流进行详细讲解

5.1 I/O流概述

I/O流分为很多种,按照操作数据的不同,可以分为字节流和字符流,按照数据传输方向的不同又可以分为输入流和输出流,程序从输入流中读取数据,向输出流中写入数据。在I/O包中,字节流的输入/输出流分别用java.io.InputStream和java.io.OutputStream表示,字符流的输入/输出流分别用java.io.Reader和java.io.Writer表示,具体分类如图5-1所示。

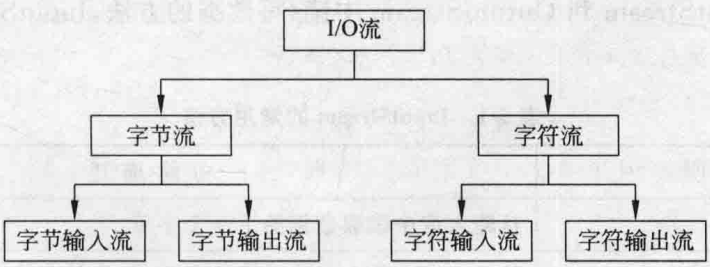


图5-1 I/O流分类

5.2 字节流

5.2.1 字节流的概念

字节流是指针对字节输入/输出提供的一系列流。在计算机中,无论是文本、图片还是音频和视频,都是以字节的形式存在的,因此要对这些内容进行传输就需要使用字节流。字节流是程序中最常用的流,根据数据传输方向的不同可将其分为字节输入流和字节输出流。在JDK中,提供了两个抽象类InputStream和OutputStream表示字节输入流和字节输出流,它们是字节流的两个顶级父类,字节输入流都是InputStream的子类,字节输出流都是OutputStream的子类。字节流的继承体系如图5-2和图5-3所示。

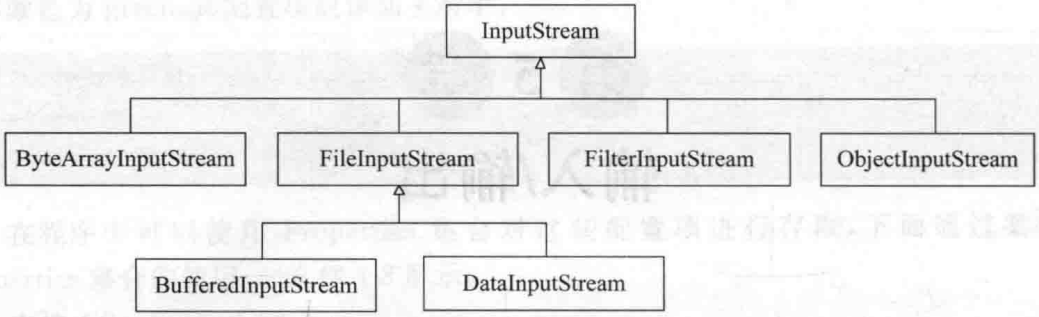


图 5-2 InputStream 的子类

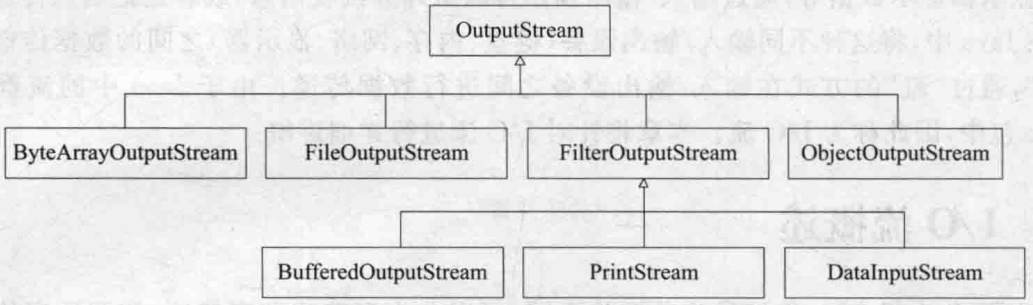


图 5-3 OutputStream 的子类

图 5-2 和图 5-3 所列出的 I/O 流都是程序中很常见的，并且可以看出 InputStream 和 OutputStream 的子类有很多是大致对应的，例如，ByteArrayInputStream 和 ByteArrayOutputStream、FileInputStream 和 FileOutputStream 等。

下面介绍 InputStream 和 OutputStream 中读/写数据的方法，InputStream 中的常用方法如表 5-1 所示。

表 5-1 InputStream 的常用方法

方法声明	功能描述
int read()	从输入流中读取数据的下一个字节
int read(byte[] b)	从输入流中读取一定数量的字节，并将其存储在缓冲区数组 b 中
int read(byte[] b,int off,int len)	从输入流中读取若干字节，把它们保存到参数 b 指定的字节数组中，off 指定字节数组开始保存数据的起始下标，len 表示读取的字节数目
void close()	关闭此输入流并释放与该流关联的所有系统资源

与 InputStream 对应的是 OutputStream。OutputStream 用于写数据，因此 OutputStream 提供了一些与写数据有关的方法，如表 5-2 所示。

表 5-2 OutputStream 的常用方法

方法名称	方法描述
void write(int b)	将指定的字节写入此输出流
void write(byte[] b)	将 b.length 个字节从指定的 byte 数组写入此输出流

续表

方法名称	方法描述
void write(byte[] b,int off,int len)	将指定 byte 数组中从偏移量 off 开始的 len 个字节写入此输出流
void flush()	刷新此输出流并强制写出所有缓冲的输出字节
void close()	关闭此输出流并释放与此流有关的所有系统资源

5.2.2　字节流的读/写操作

由于计算机中的数据基本都保存在硬盘的文件中，因此操作文件中的数据是一种很常见的操作。在操作文件时，最常见的就是从文件中读取数据并将数据写入文件，即文件的读/写。针对文件的读/写，JDK 专门提供了两个类，分别是 FileInputStream 和 FileOutputStream，它们分别包含了一套所有输入与输出需要使用的方法，可以完成最基本的读取与写出功能。

下面通过案例实现字节流对文件数据的读取。首先在 Eclipse 项目的 src 目录下创建一个文本文件 itheima.txt，在文件中输入内容“Hello”并保存；然后创建一个读取文本文件的类，具体代码如文件 5-1 所示。

文件 5-1　Example01.java

```
1  package com.itheima.example;
2  import java.io.*;
3  public class Example01 {
4      public static void main(String[] args) throws Exception {
5          //创建一个文件字节输入流
6          FileInputStream input=new FileInputStream("itheima.txt");
7          int i=0;                    //定义一个 int 类型的变量 i,记住每次读取的一个字节
8          while (true) {
9              i=input.read();         //变量 i 记住读取的一个字节
10             if (i==-1) {            //如果读取的字节为-1,跳出 while 循环
11                 break;
12             }
13             System.out.println(i);          //否则将 i 写出
14         }
15         input.close();                      //关闭资源
16     }
17 }
```

运行结果如图 5-4 所示。

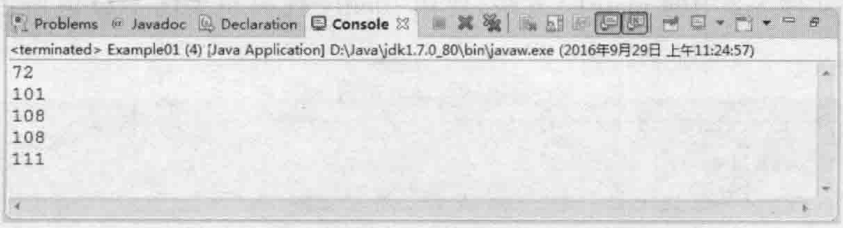


图 5-4　运行结果

在文件 5-1 中，创建的字节流 FileInputStream 通过 read()方法读取 itheima. txt 文件中的数据并输出。从图 5-4 的运行结果中可以看出，结果分别为 72、101、108、108 和 111。通常情况下读取文件输出结果为字符，之所以输出数字是因为在硬盘上存储的文件都是以字节的形式存在的。在 itheima. txt 文件中，字符'H''e''l''l''o'各占一个字节，因此，最终在控制台显示的结果就是文件"itheima. txt"中的五个字节所对应的十进制数。

与 FileInputStream 对应的是 FileOutputStream，专门用于把数据写入文件中。下面通过案例演示如何将数据写入文件，如文件 5-2 所示。

文件 5-2 Example02. java

```
1  package com.itheima.example;
2  import java.io.*;
3  public class Example02 {
4      public static void main(String[] args) throws Exception {
5          //创建一个文件字节输出流
6          FileOutputStream output=new FileOutputStream("test.txt");
7          String s="www.itheima.com";          //定义一个 String 类型字符串
8          byte[] arr=s.getBytes();             //将字符串变成字节数组
9          for (int i=0; i <arr.length; i++) {
10             output.write(arr[i]);            //将数组中的数据写入到目标文件中
11         }
12         output.close();                      //关闭资源
13     }
14 }
```

程序运行结束后，对项目 chapter05 进行刷新(按 F5 键)后，会在当前目录下生成一个新的文本文件 test. txt，打开此文件，会看到如图 5-5 所示的内容。

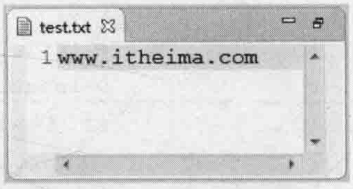


图 5-5 文件内容

从运行结果中可以看出，通过 FileOutputStream 写数据时，自动创建了文件 test. txt，并将数据写入文件。需要注意的是，如果是通过 FileOutputStream 向一个已经存在的文件中写入数据，那么该文件中的数据首先会被清空，再写入新的数据。若希望在已存在的文件内容之后增加新内容，只需要在字节输出流的末尾追加一个参数值 true 即可。

由于 I/O 流在进行数据读/写操作时会出现异常，为了代码的简洁，在上面的程序中使用了 throws 关键字将异常抛出。然而一旦遇到 I/O 异常，I/O 流的 close()方法将无法得到执行，流对象所占用的系统资源将无法释放，因此，为了保证 I/O 流的 close()方法必须执行，通常将关闭流的操作写在 finally 代码块中。

下面改写文件 5-2 中的 main()方法，使用 finally 代码块关闭 I/O 流所占用的系统资源，具体如下：

```
1  package com.itheima.example;
2  import java.io.*;
3  public static void main(String[] args) {
4      FileOutputStream output=null;            //创建一个文件字节输出流
```

```
    try {
        output=new FileOutputStream("test.txt");
        String s="www.itheima.com";          //定义一个 String 类型字符串
        byte[] arr=s.getBytes();             //将字符串变成字节数组
        for (int i=0; i <arr.length; i++) {
            output.write(arr[i]);            //将数组中的数据写入到目标文件中
        }
    } catch (IOException e) {
        e.printStackTrace();
    } finally {
        if (output !=null) {
            output.close();                  //关闭资源
        }
    }
}
```

5.2.3　文件的复制

在应用程序中，通常情况下输入流和输出流都是同时使用的。例如，文件的复制就需要通过输入流读取文件中的数据，通过输出流将数据写入文件。下面通过案例演示如何进行文件的复制操作。

首先在 chapter05 项目下创建文件夹 source 和 target（右击项目名称后选择 New→Folder），然后在 source 文件夹中存放“黑马程序员.jpg”，复制文件的代码如文件 5-3 所示。

文件 5-3　Example03.java

```
package com.itheima.example;
import java.io.*;
public class Example03 {
    public static void main(String[] args) throws Exception {
        //创建一个字节输入流，用于读取当前目录下 source 文件夹中的图片
        FileInputStream input=new FileInputStream("source\黑马程序员.jpg");
        //创建一个文件字节输出流，用于将读取的数据写入 target 目录下的图片
        FileOutputStream output=new FileOutputStream("target\黑马程序员.jpg");
        //定义一个 int 类型的变量 length，记住每次读取的一个字节
        int length;
        long startTime=System.currentTimeMillis();      //获取复制图片前的系统时间
        while((length=input.read())!=-1) {      //读取一个字节并判断是否读到文件末尾
            output.write(length);                       //将读到的字节写入文件
        }
        long endTime=System.currentTimeMillis(); //获取图片复制结束时的系统时间
        System.out.println("复制图片所消耗的时间是："+(endTime-startTime)+"毫秒");
        input.close();
        output.close();
    }
}
```

程序运行结束后，刷新并打开 target 文件夹，发现 source 文件夹中的“黑马程序员.jpg”文件被成功复制到了 target 文件夹，如图 5-6 所示。

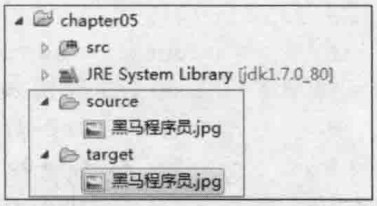


图 5-6 复制前后的文件夹

运行结果如图 5-7 所示。

需要注意的是，文件 5-3 在定义文件路径时使用了\\，这是因为在 Windows 中的目录符号为反斜线\，但反斜线\在 Java 中是特殊字符，表示转义符，所以在使用反斜线\时，前面应该再添加一个反斜线，即\\。除此之外，目录符号也可以用正斜线/表示，如“source/黑马程序员.jpg”。

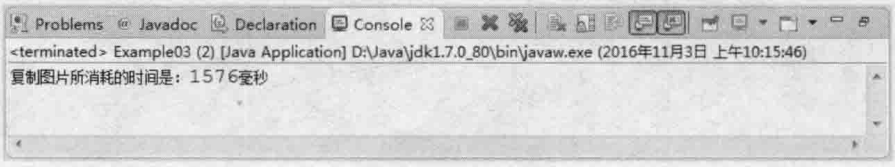


图 5-7 运行结果

5.2.4 字节流的缓冲区

5.2.3 节中实现了对图片的复制，但是单个字节的读/写效率非常低。为了提高效率，可以定义一个字节数组作为缓冲区。在复制文件时，可以将读取的单个字节保存到字节数组中，然后将字节数组中的数据一次性写入文件。下面通过修改文件 5-3 学习如何使用字节数组复制文件，如文件 5-4 所示。

文件 5-4 Example04.java

```
1  package com.itheima.example;
2  import java.io.*;
3  public class Example04 {
4      public static void main(String[] args) throws Exception {
5          FileInputStream input=new FileInputStream("source\黑马程序员.jpg");
6          FileOutputStream output=new FileOutputStream("target\黑马程序员.jpg");
7          byte[] buffer=new byte[1024];      //定义一个字节数组作为缓冲区
8          int length;                        //定义变量 length 记住读取缓冲区的字节数
9          long startTime=System.currentTimeMillis();
10         while ((length=input.read(buffer)) !=-1) {      //判断是否读到文件末尾
11             //从第一个字节开始，向文件写入 length 个字节
12             output.write(buffer, 0, length);
13         }
14         long endTime=System.currentTimeMillis();
15         System.out.println("复制图片所消耗的时间是："+(endTime-startTime)+"毫秒");
16         input.close();
17         output.close();
18     }
19 }
```

运行结果如图 5-8 所示。

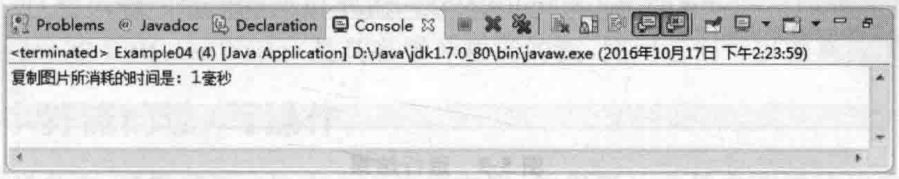


图 5-8　运行结果

通过比较图 5-7 和图 5-8 可以看出，复制图片所消耗的时间减少了，这说明使用字节数组作为缓冲区读/写文件可以有效提高程序的执行效率。

5.2.5　字节缓冲流

在进行文件复制时，使用字节流缓冲区可以提高程序的效率，与此同时，还可以使用 java.io 包中自带缓冲功能的字节缓冲流，它们分别是 BufferedInputStream 和 BufferedOutputStream，这两个流在实例化时需要接收 InputStream 和 OutputStream 类型的对象作为参数。接下来使用字节缓冲流 BufferedInputStream 和 BufferedOutputStream 实现文件的复制，如文件 5-5 所示。

文件 5-5　Example05.java

```
1  package com.itheima.example;
2  import java.io.*;
3  public class Example05 {
4      public static void main(String[] args) throws Exception {
5          //创建一个带缓冲区的输入流
6          BufferedInputStream bufferInput=new BufferedInputStream(
7                  new FileInputStream("source\黑马程序员.jpg"));
8          //创建一个带缓冲区的输出流
9          BufferedOutputStream bufferOutput=new BufferedOutputStream(
10                 new FileOutputStream("target\黑马程序员.jpg"));
11         long startTime=System.currentTimeMillis();
12         int length;
13         while ((length=bufferInput.read()) !=-1) {
14             bufferOutput.write(length);
15         }
16         long endTime=System.currentTimeMillis();
17         System.out.println("复制图片所消耗的时间是："+(endTime-startTime)+"毫秒");
18         bufferInput.close();
19         bufferOutput.close();
20     }
21 }
```

运行结果如图 5-9 所示。

通过比较图 5-8 和图 5-9 可以看出，复制图片所消耗的时间相同，这说明除了使用字节

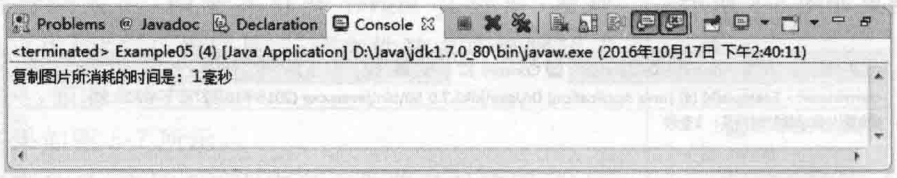


图 5-9 运行结果

数组作为缓冲区能提高程序执行效率以外，使用 I/O 流自带的字节缓冲流同样能提高程序的执行效率。

5.3 字符流

字符流的目标通常是文本文件。Reader 和 Writer 是 java.io 包中所有字符流的抽象父类，定义了在 I/O 流中读/写字符数据的通用 API。在 Java 中，字符采用的是 Unicode 字符编码，常见的字符输入/输出流是由 Reader 和 Writer 抽象类派生出来的，处理数据时以字符为基本单位。

5.3.1 字符流的概念

前面已经讲解过 InputStream 类和 OutputStream 类在读/写文件时操作的都是字节，如果希望在程序中操作字符，使用这两个类就不太方便，为此，JDK 提供了字符流。同字节流一样，字符流也有两个抽象的顶级父类，分别是 Reader 和 Writer。其中 Reader 是字符输入流，用于从某个源设备读取字符；Writer 是字符输出流，用于向某个目标设备写入字符。

Reader 和 Writer 作为字符流的顶级父类，也有许多子类，其继承体系如图 5-10 和图 5-11 所示。

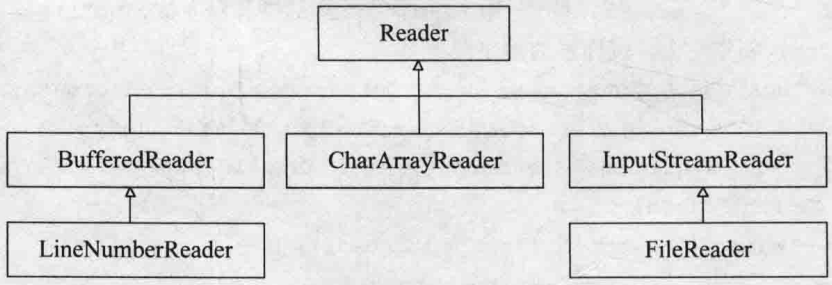


图 5-10 Reader 的子类

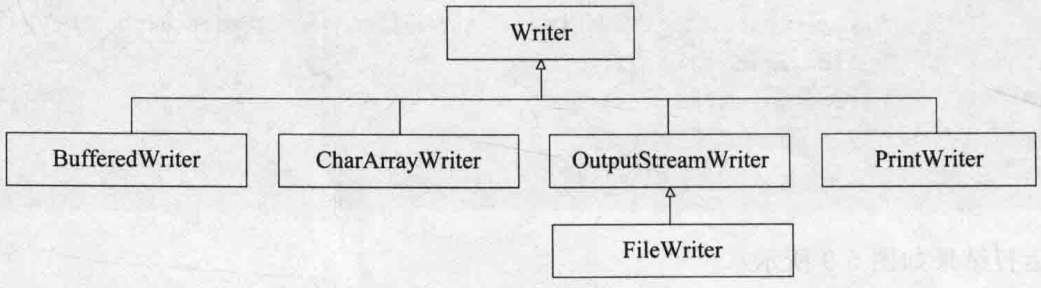


图 5-11 Writer 的子类

从图 5-10 和图 5-11 可以看到，字符流的继承关系与字节流的继承关系有些类似，很多子类都是成对（输入流和输出流）出现的，其中 FileReader 和 FileWriter 用于读/写文件，BufferedReader 和 BufferedWriter 是具有缓冲功能的流，使用它们可以提高读/写效率。

5.3.2　字符流的读/写操作

在程序开发中，经常需要读取文本文件的内容，如果要从文件中直接读取字符，便可以使用字符输入流 FileReader，通过此流可以从关联的文件中读取一个或一组字符。接下来在 chapter05 项目下创建文本文件“read. txt”，并在其中输入字符“www. itheima. com”，然后创建一个字符输入流 FileReader 读取文本文件中的内容，如文件 5-6 所示。

文件 5-6　Example06. java

```
1  package com.itheima.example;
2  import java.io.*;
3  public class Example06 {
4      public static void main(String[] args) throws Exception {
5          //创建一个 FileReader 对象用于读取文件中的字符
6          FileReader fileReader=new FileReader("read.txt");
7          int i;                                    //定义一个变量用于记录读取的字符
8          while ((i=fileReader.read()) !=-1) {      //循环判断是否读取到文件的末尾
9              System.out.print((char) i);           //不是字符流末尾就转为字符打印
10         }
11         fileReader.close();                       //关闭资源
12     }
13 }
```

运行结果如图 5-12 所示。

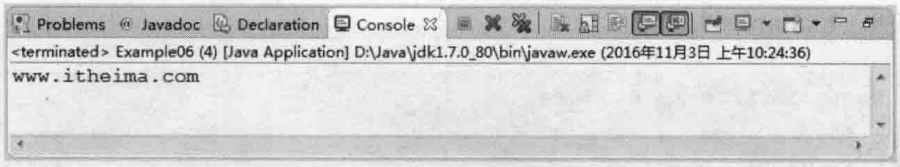


图 5-12　运行结果

在文件 5-6 中，创建一个 FileReader 对象并指定“read. txt”文件，然后通过 while 循环每次从文件中读取单个字符并输出到控制台上，这样便实现了 FileReader 读取文本文件字符的操作。

在文件 5-6 中，讲解了如何使用 FileReader 读取文本文件中的字符，如果要向文件中写入字符就需要使用 FileWriter 类。FileWriter 是 Writer 的一个子类，下面通过案例学习如何使用 FileWriter 将字符写入文件，如文件 5-7 所示。

文件 5-7　Example07. java

```
1  package com.itheima.example;
2  import java.io.*;
```

```
3  public class Example07 {
4      public static void main(String[] args) throws Exception {
5          //创建一个 FileWriter 对象用于向文件中写入数据
6          FileWriter fileWriter=new FileWriter("write.txt");
7          String s="你好,www.itheima.com";
8          fileWriter.write(s);          //将字符数据写入到文本文件中
9          fileWriter.close();           //关闭写入流,释放资源
10     }
11 }
```

程序运行结束后,会在当前目录下生成 write.txt 文件,打开此文件会看到如图 5-13 所示的内容。

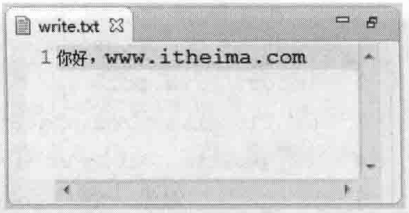


图 5-13 writer.txt

FileWriter 同 FileOutputStream 一样,如果指定的文件不存在,则会先创建文件,再写入数据,如果文件存在,则会先清空文件中的内容,再进行写入。若希望在已存在的文件内容之后增加新内容,只需要在字符输出流的末尾追加一个参数值 true 即可。

5.3.3 字符缓冲流

从 5.2.5 节中可以看出,字节缓冲流可以有效提高读/写数据的效率,而字符流同样提供了带缓冲区的输入流与输出流,分别是 BufferedReader 和 BufferedWriter,其中 BufferedReader 用于对字符输入流提供缓冲区,BufferedWriter 用于对字符输出流提供缓冲区,需要注意的是,在 BufferedReader 中有一个重要的方法 readLine(),该方法用于一次读取一行文本。下面通过案例学习如何使用字符缓冲流实现文本文件的复制,如文件 5-8 所示。

文件 5-8 Example08.java

```
1  package com.itheima.example;
2  import java.io.*;
3  public class Example08 {
4      public static void main(String[] args) throws Exception {
5          //创建一个 BufferedReader 缓冲对象
6          BufferedReader bufferReader=new BufferedReader(
7                  new FileReader("read.txt"));
8          //创建一个 BufferdWriter 缓冲区对象
9          BufferedWriter bufferWriter=new BufferedWriter(
10                 new FileWriter("dest.txt"));
11         String s=null;
12         //每次读取一行文本,判断是否到文件末尾
13         while ((s=bufferReader.readLine()) !=null) {
14             bufferWriter.write(s);
15             //写入一个换行符,该方法会根据不同的操作系统生成相应的换行符
```

```
16              bufferWriter.newLine();
17          }
18          bufferReader.close();
19          bufferWriter.close();
20      }
21  }
```

程序运行结束后，按 F5 键刷新 chapter05 项目后打开 dest.txt 文件，执行结果如图 5-14 所示。

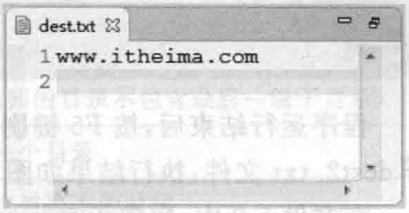


图 5-14　dest.txt

需要注意的是，由于字符缓冲流内部使用了缓冲区，在循环中调用 BufferedWriter 的 write()方法写入字符时，这些字符会写入缓冲区中，当缓冲区填充不下或者调用 close()方法时，缓冲区中的字符才会一次性写入目标文件中。因此在循环结束时一定要调用 close()方法，否则很容易会导致缓冲区的部分数据无法写入目标文件中。

5.3.4　转换流

在程序开发中，有时会遇到字节流和字符流之间需要进行转换的可能。在 java.io 包中提供了两个类可以将字节流转换为字符流，它们分别是 InputStreamReader 和 OutputStreamWriter。

OutputStreamWriter 是 Writer 的子类，能够将一个字节输出流转换成字符输出流，方便直接写入字符，而 InputStreamReader 是 Reader 的子类，能够将一个字节输入流转换成字符输入流，方便直接读取字符。下面通过案例学习如何将字节流转换为字符流，为了提高读/写效率，通过字符缓冲流实现转换的操作，如文件 5-9 所示。

文件 5-9　Example09.java

```
1  package com.itheima.example;
2  import java.io.*;
3  public class Example09 {
4      public static void main(String[] args) throws Exception {
5          //创建字节输入流
6          FileInputStream input=new FileInputStream("read.txt");
7          //将字节输入流转换成字符输入流
8          InputStreamReader streamReader=new InputStreamReader(input);
9          //赋予字符输入流对象缓冲区
10         BufferedReader bufferReader=new BufferedReader(streamReader);
11         //创建字节输出流
12         FileOutputStream output=new FileOutputStream("dest2.txt");
13         //将字节输出流转换成字符输出流
14         OutputStreamWriter streamWriter=new OutputStreamWriter(output);
15         //赋予字符输出流对象缓冲区
16         BufferedWriter bufferWriter=new BufferedWriter(streamWriter);
```

```
17          String line=null;
18          //判断是否读到文件末尾
19          while ((line=bufferReader.readLine()) !=null) {
20              bufferWriter.write(line);          //输出读取到的文件
21          }
22          bufferReader.close();
23          bufferWriter.close();
24      }
25  }
```

程序运行结束后，按 F5 键刷新 chapter05 项目后打开 dest2. txt 文件，执行结果如图 5-15 所示。

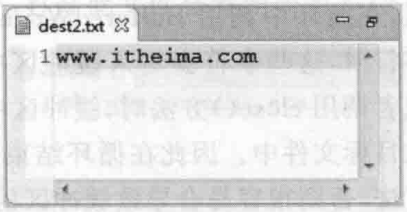


图 5-15 dest2. txt

在文件 5-9 中，实现了字节流和字符流之间的转换，将字节流转换为字符流，从而实现直接对字符的读/写。需要注意的是，在使用转换流时，只能针对操作文本文件的字节流进行转换，如果字节流操作的是图片或者音频，此时转换为字符流就会造成数据丢失。

5.4 File 类

File 类提供了目录和文件管理的功能，主要用于文件命名、文件属性查看和文件目录管理、文件夹创建等操作。File 类同样位于 java. io 包中，但是 File 类不能对文件内容进行读/写操作。接下来围绕 File 类展开详细讲解。

5.4.1 File 类的常用方法

File 类用于封装一个路径，这个路径可以指向一个文件，也可以指向一个目录，File 类提供了针对这些文件或目录的一些常规操作。File 类常用的构造方法如表 5-3 所示。

表 5-3 File 类常用的构造方法

方法声明	功能描述
File(String pathname)	通过将给定路径名字符串转换为抽象路径名创建一个新 File 实例
File(String parent,String child)	根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例
File(File parent,String child)	根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例

在表 5-3 中，如果程序只处理一个目录或文件，并且知道该目录或文件的路径，则使用第一个构造方法更方便；如果程序处理的是一个公共目录中的若干子目录或者文件，那么使用第二或第三个构造方法更方便。

File 类中提供了一系列方法，用于操作其内部封装的路径指向的文件或者目录，如创建目录、删除目录、返回目录名称等，File 类中的常用方法如表 5-4 所示。

表 5-4　File 类的常用方法

方法声明	功能描述
boolean exists()	判断 File 对象对应的文件或目录是否存在
boolean delete()	删除 File 对象对应的文件或目录
boolean createNewFile()	当 File 对象对应的文件不存在时，将创建一个此 File 对象指定的新文件
String getName()	返回由此抽象路径名表示的文件或目录的名称
String getPath()	将此抽象路径名转换为一个路径名字符串
String getParent()	返回 File 对象对应目录的父目录(即返回的目录不包含最后一级子目录)
boolean isDirectory()	测试此抽象路径名表示的文件是否是一个目录
long lastModified()	返回此抽象路径名表示的文件最后一次被修改的时间
long length()	返回由此抽象路径名表示的文件的长度
String[] list()	列出指定目录的全部内容，仅列出名称
File[] listFiles()	返回一个包含了 File 对象所有子文件和子目录的 File 数组

5.4.2　遍历目录下的文件

在程序开发中，经常需要遍历某个指定目录下的所有文件的名称，下面通过案例演示如何利用 File 类进行遍历，如文件 5-10 所示。

文件 5-10　Example10.java

```
1  package com.itheima.example;
2  import java.io.File;
3  public class Example10 {
4      public static void main(String[] args) throws Exception {
5          //创建 File 对象
6          File f=new File("C:/Users/admin/workspace/chapter05");
7          if (f.isDirectory()) {                    //判断 File 对象对应的目录是否存在
8              String[] ss=f.list();                 //获得目录下的所有文件的文件名
9              for (String s : ss) {
10                 System.out.println(s);            //循环遍历依次输出文件名
11             }
12         }
13     }
14 }
```

运行结果如图 5-16 所示。

在文件 5-10 中，创建了一个 File 对象，并在对象中封装了一个路径，通过调用 File 对象的 isDirectory()方法判断该路径是否存在，如果存在则调用 list()方法并返回 String 类型的数组 ss，数组中包含该路径下所有文件的文件名。然后循环遍历数组 ss，依次输出每个文件的文件名到控制台上。

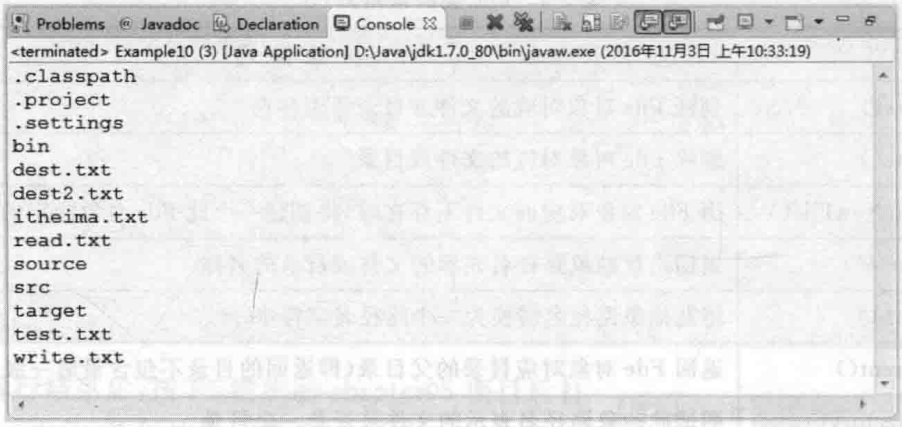


图 5-16 运行结果

5.4.3 删除文件及目录

在程序开发中，可能会遇到需要删除指定目录下的某个文件或者删除整个目录的情况，此时可以通过 File 对象的删除方法实现。首先在 C 盘的根目录下创建一个名称为 hello 文件夹，然后在该文件夹中创建任意数量的 Java 文件。下面通过案例演示如何删除指定目录下的所有 Java 文件，如文件 5-11 所示。

文件 5-11 Example11.java

```
1  package com.itheima.example;
2  import java.io.*;
3  public class Example11 {
4      public static void main(String[] args) {
5          File folder=new File("c:/hello");          //创建一个代表目录的 File 对象
6          File[] files=folder.listFiles();           //得到 File 数组
7          for(File file:files){                      //遍历所有的子目录和文件
8              if(file.getName().endsWith(".java")){  //筛选该目录下的所有 java 文件
9                  file.delete();                     //如果是以 java 为后缀名的文件，则直接删除
10             }
11         }
12     }
13 }
```

运行成功后，观察 hello 目录下的所有文件，可以发现该目录下的所有 Java 文件都已经被成功删除。需要注意的是，在使用 File 对象删除目录时，是从 JVM 直接删除而不经过回收站，因此文件一旦删除则无法恢复。

第6章 多线程

多线程是指在软件或者硬件上实现多个线程并发执行的技术，具有多线程能力的计算机因为有硬件支持而使其能够在同一时间执行多个线程，进而提升整体的处理性能。本章将针对多线程进行详细讲解。

6.1 线程概述

在学习线程之前，需要先了解一下什么是进程。在一个操作系统中，每个独立执行的程序都可称为一个进程(正在运行的程序)，目前几乎所有的操作系统都支持多任务，即能够同时执行多个程序，例如 Windows、Linux、UNIX 等。在多任务操作系统中，表面上支持进程并发执行，例如可以一边听音乐，一边看小说，但实际上这些程序并不是同时执行的。因为 CPU 具备分时机制，每个时间点只能执行一个程序，只不过由于 CPU 运行速度非常快，能够在极短的时间内在不同的进程之间进行切换，所以给人以同时执行多个进程的感觉。

每个运行中的程序都是一个进程，在这个进程的内部包含了多个执行单元，而每个执行单元就是一个线程，而且每个进程中至少包含一个线程，当一个 Java 程序启动时，就会产生一个进程，该进程会默认创建一个线程，称为主线程，在主线程上会运行 main()方法中的代码。而在前面学习的程序中，代码都是按照顺序依次调用的，这种程序称为单线程程序。如果希望程序中实现多段程序代码交替执行的效果，则需要创建多线程程序。多线程程序在运行时，每个线程之间都是独立的，它们可以并发执行，这里的并发是相对的，也就是说看起来是同时执行的，而实际上和进程一样，也是由 CPU 轮流调度的。

6.2 线程的创建

在 Java 中要想实现多线程操作有两种方式，一种是继承 Thread 类，另一种是实现 Runnable 接口。接下来针对这两种创建多线程的方式分别进行讲解，并比较它们的优缺点。

6.2.1 继承 Thread 类

Thread 类是在 java.lang 包中定义的，一个类只要继承了 Thread 类，此类就是多线程的子类。在 Thread 子类中，必须重写该类的 run()方法，此方法为线程的主体。接下来通过案例演示通过继承 Thread 类的方式实现多线程。

1. 创建 ThreadDemo 类

创建的 ThreadDemo 需要继承 Thread 类，并重写 Thread 类中的 run()方法，在 run()方法中通过 for 循环输出五次“ThreadDemo 类的 run()方法执行了”，如文件 6-1 所示。

文件 6-1 ThreadDemo.java

```
1  package com.itheima.example01;
2  public class ThreadDemo extends Thread {
3      public void run() {
4          for (int i=0; i <5; i++) {
5              System.out.println("ThreadDemo 类的 run()方法执行了");
6          }
7      }
8  }
```

2. 创建测试类

创建测试类 Example01，在该类中创建一个线程对象 threadDemo，并调用 start()方法启动子线程，然后通过 for 循环在 main()方法（主线程）中输出五次“主方法 main()执行了”，如文件 6-2 所示。

文件 6-2 Example01.java

```
1  package com.itheima.example01;
2  public class Example01 {
3      public static void main(String[] args) {
4          ThreadDemo threadDemo=new ThreadDemo(); //创建线程 ThreadDemo 的线程对象
5          threadDemo.start();                     //开启线程
6          for (int i=0; i <5; i++) {
7              System.out.println("主方法 main()执行了");
8          }
9      }
10 }
```

运行结果如图 6-1 所示。

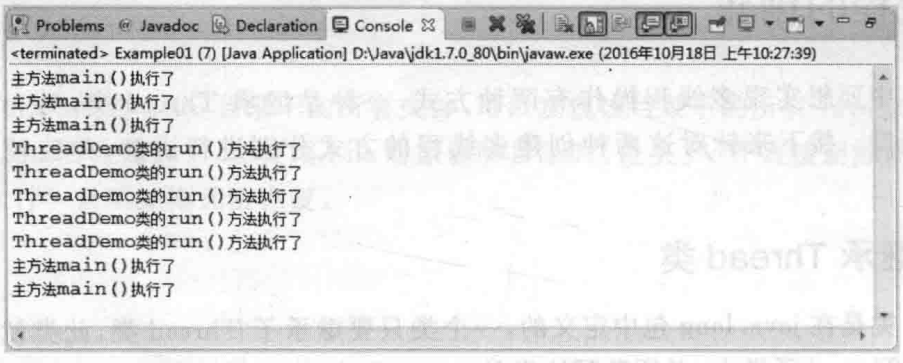


图 6-1 运行结果

从图 6-1 中可以看出，两个 for 循环中的输出语句交错执行了，说明该程序实现了多线程。为了让读者更好地了解多线程程序，下面通过一个图例分析多线程程序的执行流程，如图 6-2 所示。

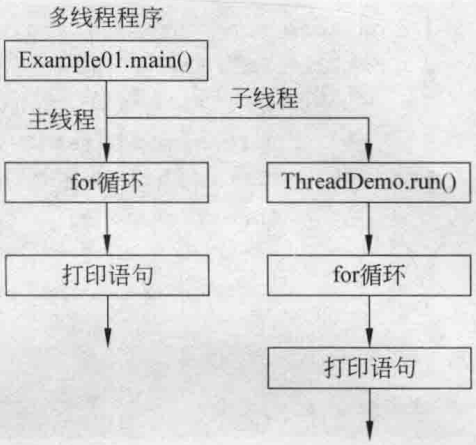


图 6-2　多线程执行流程

从图 6-2 中可以看出，Example01 程序是一个多线程程序，在 main()方法执行主线程的 for 循环语句前，单独开启了一个子线程执行 ThreadDemo 中的 for 循环语句，两个线程相互独立，互不影响。

6.2.2　实现 Runnable 接口

文件 6-1 通过继承 Thread 类实现了多线程，但是这种方式有一定的局限性。因为 Java 中只支持单继承，一个类一旦继承了某个父类就无法再继承 Thread 类，例如猫类 Cat 继承了动物类 Animal，就无法通过继承 Thread 类实现多线程。

为了克服这种弊端，在 Thread 类中提供了 public Thread(Runnable target)和 public Thread(Runnable target, String name)两个构造方法，这两个构造方法都可以接收 Runnable 的子类实例对象，这样创建的线程将调用实现了 Runnable 接口类中的 run()方法作为运行代码，而不需要调用 Thread 类的 run()方法，所以就可以依靠 Runnable 接口的实现类启动多线程。下面通过案例演示如何通过实现 Runnable 接口的方式创建多线程。

1. 创建 ThreadDemo 类

创建 ThreadDemo 类实现 Runnable 接口，并重写 Runnable 接口中的 run()方法，如文件 6-3 所示。

文件 6-3　ThreadDemo.java

```
1  package com.itheima.example02;
2  public class ThreadDemo implements Runnable {
3      public void run() {   //线程的代码段，当调用 start()方法时，线程从此处开始执行
4          for (int i=0; i <5; i++) {
5              System.out.println("ThreadDemo 类的 run()方法执行了");
6          }
7      }
8  }
```

2. 创建测试类

创建测试类 Example02，在该类中创建一个线程对象 threadDemo，并通过 Thread 类的构造方法将 threadDemo 对象作为参数传入，然后创建 Thread 对象，通过 start()方法启动线程，如文件 6-4 所示。

文件 6-4 Example02.java

```
1  package com.itheima.example02;
2  public class Example02 {
3      public static void main(String[] args) {
4          ThreadDemo threadDemo=new ThreadDemo();  //创建 ThreadDemo 的实例对象
5          Thread thread=new Thread(threadDemo);    //创建线程对象
6          thread.start();                          //开启线程,执行线程中的 run()方法
7          for (int i=0; i <5; i++) {
8              System.out.println("主方法 main()执行了");
9          }
10     }
11 }
```

运行结果如图 6-3 所示。

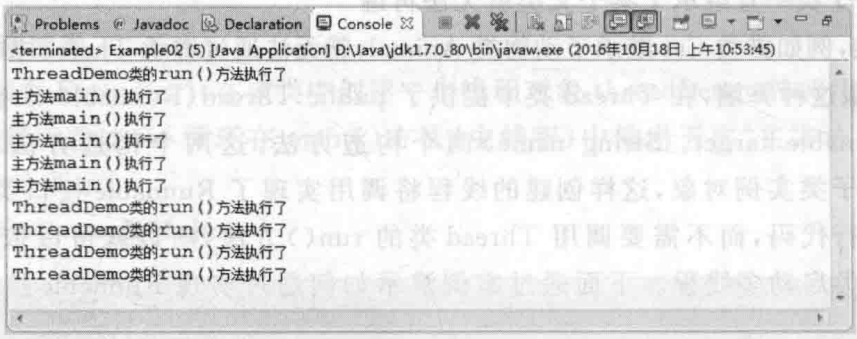


图 6-3 运行结果

从图 6-3 中可以看出,main()方法和 run()方法中的输出语句交错执行了,说明实现 Runnable 接口同样也实现了多线程。

6.2.3 Thread 类和 Runnable 接口对比分析

既然直接继承 Thread 类和实现 Runnable 接口都能实现多线程,那么这两种实现多线程的方式在实际应用中又有什么区别呢?接下来通过一种应用场景进行分析。

假设售票厅有四个窗口可发售某日某次列车的 100 张车票,这时,100 张车票可以看作共享资源,四个售票窗口需要创建四个线程。下面分别通过继承 Thread 类的方式和实现 Runnable 接口的方式实现多线程的创建。

1. 创建 Ticket 类

创建 Ticket 类继承自 Thread 类,并重写 Thread 类中的 run()方法,在 run()方法中通过使用 while 循环依次对 tickets 属性进行自减操作以模拟售票的过程,直到 tickets 属性值为 0 跳出循环,如文件 6-5 所示。

文件 6-5 Ticket.java

```
1  package com.itheima.example03;
2  public class Ticket extends Thread {
```

```
3       private int tickets=100;
4       private String name;
5       public Ticket(String name) {
6           this.name=name;
7       }
8       public void run() {
9           while (true) {          //通过死循环语句打印语句
10              if (tickets >0) {
11                  System.out.println(name+" 正在发售第 "+tickets--+" 张票 ");
12              } else {
13                  break;
14              }
15          }
16      }
17  }
```

2. 创建测试类

创建测试类 Example03，在该类中创建四个线程对象，代表售票厅的四个窗口，并调用 start()方法启动四个线程，如文件 6-6 所示。

文件 6-6　Example03.java

```
1   package com.itheima.example03;
2   public class Example03 {
3       public static void main(String[] args) {
4           //定义四个线程对象,代表四个售票窗口
5           Ticket t1=new Ticket("窗口 1");
6           Ticket t2=new Ticket("窗口 2");
7           Ticket t3=new Ticket("窗口 3");
8           Ticket t4=new Ticket("窗口 4");
9           //启动四个线程
10          t1.start();
11          t2.start();
12          t3.start();
13          t4.start();
14      }
15  }
```

运行结果如图 6-4 所示。

从图 6-4 中可以看出，100 张票在每个窗口都输出了一次。出现这种情况的原因是四个线程没有共享 100 张票，而是各自出售了 100 张票。在程序中创建了四个 Ticket 对象，就相当于创建了四个售票程序，每个程序中都有 100 张票，每个线程在独立处理各自的资源。

由于现实中铁路系统的火车票资源是共享的，因此上面的运行结果显然不合理。为了

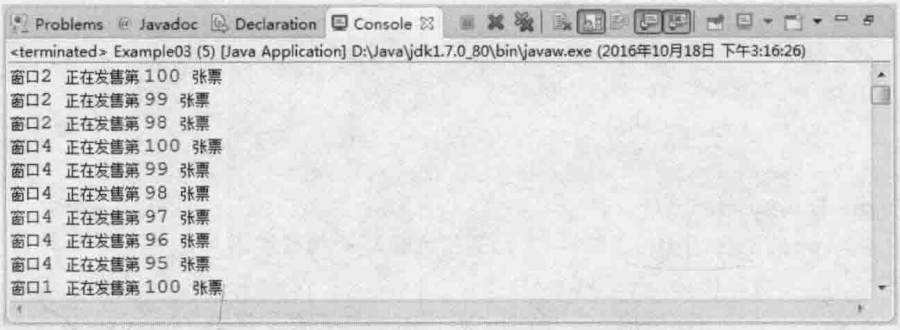


图 6-4 运行结果

保证资源共享，在程序中只能创建一个售票对象，然后开启多个线程运行同一个售票对象的售票方法。下面通过实现 Runnable 接口的方式实现多线程的创建。将文件 6-5 和文件 6-6 进行修改，具体步骤如下。

1. 创建 Ticket 类

创建 Ticket 类，使其实现 Runnable 接口，并重写 Runnable 接口中的 run()方法模拟售票过程，如文件 6-7 所示。

文件 6-7 Ticket.java

```
1  package com.itheima.example04;
2  public class Ticket implements Runnable {
3      private int tickets=100;
4      public void run() {
5          while (true) {              //通过死循环语句打印语句
6              if (tickets >0) {
7                  //获得当前线程的名称
8                  String name=Thread.currentThread().getName();
9                  System.out.println(name+" 正在发售第 "+tickets--+" 张票 ");
10             } else {
11                 break;
12             }
13         }
14     }
15 }
```

2. 创建测试类

创建测试类 Example04，在该类中只需要创建一个 Ticket 对象，然后开启四个线程即可，如文件 6-8 所示。

文件 6-8 Example04.java

```
1  package com.itheima.example04;
2  public class Example04 {
3      public static void main(String[] args) {
```

```
4           //创建任务对象
5           Ticket ticket=new Ticket();
6           //创建四个线程对象
7           Thread t1=new Thread(ticket, "窗口 1");
8           Thread t2=new Thread(ticket, "窗口 2");
9           Thread t3=new Thread(ticket, "窗口 3");
10          Thread t4=new Thread(ticket, "窗口 4");
11          //开启四个线程
12          t1.start();
13          t2.start();
14          t3.start();
15          t4.start();
16      }
17  }
```

运行结果如图 6-5 所示。

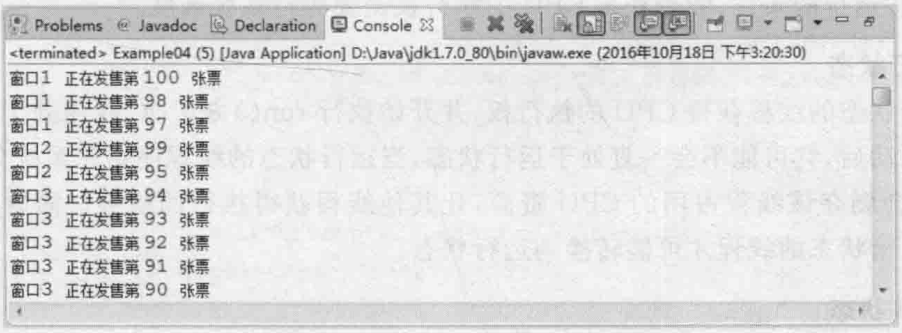


图 6-5　运行结果

从图 6-5 中可以看出，四个线程访问的是同一个 tickets 变量，共享了 100 张车票。由此可知，实现 Runnable 接口相对于继承 Thread 类来说，有如下显著的优势。

- 适合多个程序代码相同的线程处理同一资源的情况。
- 可以避免由于 Java 的单继承特性带来的局限。

所以在实际开发中建议使用第二种方式，即实现 Runnable 接口方式创建多线程。

6.3　线程的状态和转换

在 Java 中，要想实现多线程，就必须在主线程中创建新的线程对象。当线程对象创建完成时，线程的生命周期也就开始了，当 run()方法正常执行完毕，或者出现未捕获的异常或者错误时，线程的生命周期便结束了。线程的生命周期中包含五种状态，分别是新建状态、就绪状态、运行状态、阻塞状态和死亡状态，线程的不同状态表明了线程当前正在进行的活动。在程序中，通过一些操作可以使线程在不同状态之间进行转换，如图 6-6 所示。

图 6-6 展示了线程各种状态的转换关系，箭头表示可转换的方向，其中，单箭头表示状态只能单向转换，双箭头表示两种状态可以互相转换，下面针对线程的五种状态分别进行详

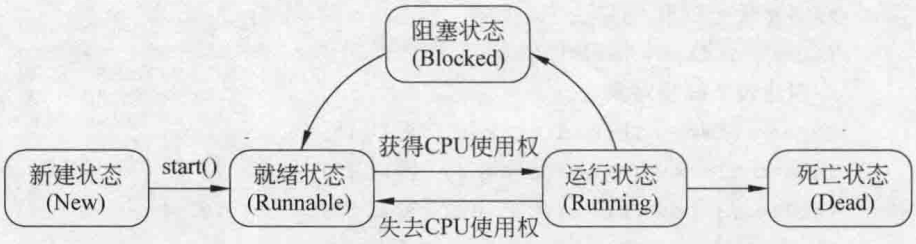


图 6-6 线程状态转换图

细讲解。

1. 新建状态

当线程对象创建成功后，线程就处于新建状态，处于新建状态的线程仅仅是在 Java 虚拟机中分配了内容空间，此时还不能运行。

2. 就绪状态

当线程对象调用了 start()方法后，就进入了就绪状态，处于就绪状态的线程位于可运行池中，具备运行的条件，能否获得 CPU 的执行权需要等待系统调度。

3. 运行状态

当就绪状态的线程获得 CPU 的执行权，并开始执行 run()方法时，线程处于运行状态。一个线程启动后，它可能不会一直处于运行状态，当运行状态的线程使用完系统分配的时间后，系统就会剥夺该线程占用的 CPU 资源，让其他线程获得执行的机会。需要注意的是，只有处于就绪状态的线程才可能转换到运行状态。

4. 阻塞状态

一个正在执行的线程在某些特殊情况下，如果被人为挂起或需要执行耗时的输入/输出操作时，会让出 CPU 的执行权进入阻塞状态。进入阻塞状态的线程不能进入排队队列，只有当引起阻塞的原因被消除后，线程才可以转入就绪状态。

5. 死亡状态

线程调用 stop()方法时或 run()方法执行结束后，即处于死亡状态。处于死亡状态的线程不具有继续运行的能力，也不能转换到其他状态。

6.4 多线程同步

一个多线程的程序如果是通过 Runnable 接口实现的，则意味着类中的属性将被多个线程共享，这样一来也会引发一些安全问题，例如统计车间工人数量时，经常有人来回出入，很难统计正确。为了解决这样的问题，就需要实现多线程同步，限制某个资源在某一时刻只能被一个线程访问，接下来针对多线程同步进行详细讲解。

6.4.1 线程安全

针对文件 6-7 和文件 6-8 中的售票案例很可能出现一张票被打印多次或者出现负数的情况，这些情况都是由多线程操作共享资源 tickets 所导致的线程安全问题，为了能清楚地

看出问题所在，接下来对文件 6-7 和文件 6-8 进行修改，模拟四个窗口出售 100 张票，在售票的代码中使用 sleep()方法，使每次售票时线程休眠 10ms。

1. 创建 Ticket 类

创建 Ticket 类，使其实现 Runnable 接口，并重写 Runnable 接口中的 run()方法，在模拟售票的过程中加入线程睡眠操作，如文件 6-9 所示。

文件 6-9　Ticket.java

```
1  package com.itheima.example05;
2  public class Ticket implements Runnable {
3      private int tickets=100;
4      public void run() {
5          while (true) {
6              try {
7                  if (tickets >0) {
8                      Thread.sleep(10);          //使线程休眠 10ms
9                      //获得当前线程的名称
10                     String name=Thread.currentThread().getName();
11                     System.out.println(name+" 正在发售第 "+tickets--+" 张票 ");
12                 } else {
13                     break;
14                 }
15             } catch (InterruptedException e) {
16                 e.printStackTrace();
17             }
18         }
19     }
20 }
```

2. 创建测试类

创建测试类 Example05，该类的代码与文件 6-8 一致，直接复制即可，如文件 6-10 所示。

文件 6-10　Example05.java

```
1  package com.itheima.example05;
2  public class Example05 {
3      public static void main(String[] args) {
4          //创建任务对象
5          Ticket ticket=new Ticket();
6          //创建四个线程对象
7          Thread t1=new Thread(ticket, "窗口 1");
8          Thread t2=new Thread(ticket, "窗口 2");
9          Thread t3=new Thread(ticket, "窗口 3");
10         Thread t4=new Thread(ticket, "窗口 4");
11         //开启四个线程
```

```
        t1.start();
        t2.start();
        t3.start();
        t4.start();
    }
}
```

运行结果如图 6-7 所示。

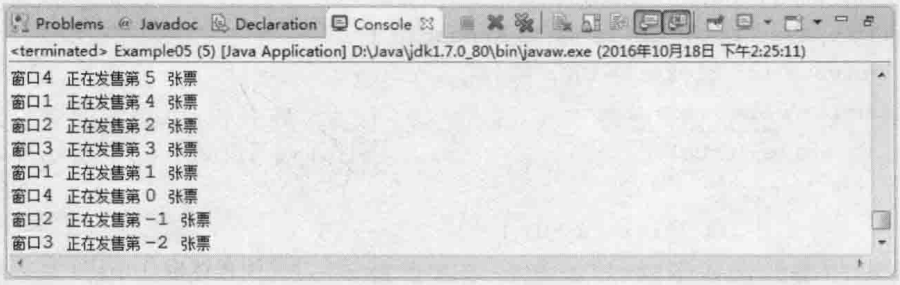


图 6-7 运行结果

从图 6-7 中可以看出，在一个线程还没有对票数进行减操作之前，其他线程就已经将票数减少了，这样一来就会出现票数为负的情况。

6.4.2 同步代码块

要解决文件 6-9 中的线程安全问题，就必须使用同步。同步是指多个操作在同一个时间段内只能有一个线程进行，其他线程要等待此线程完成之后才可以继续执行。Java 为线程的同步操作提供了 synchronized 关键字，使用该关键字修饰的代码块被称为同步代码块，其语法格式如下：

```
synchronized(同步对象){
    需要同步的代码;
}
```

从上面的代码中可以看出，在使用同步代码块时必须指定一个需要同步的对象，也称为锁对象。一般情况下都将当前对象 this 设置为同步对象。

下面对文件 6-9 进行修改，将售票的代码放到 synchronized 区域中，如文件 6-11 所示。

文件 6-11 Ticket.java

```
1  package com.itheima.example05;
2  public class Ticket implements Runnable {
3    private int tickets=100;
4    public void run() {
5        while (true) {
6            synchronized (this) {          //将 this 设置为锁对象
7                try {
```

```
                if (tickets >0) {
                    Thread.sleep(10);     //使线程休眠 10ms
                    String name=Thread.currentThread().getName();
                    System.out.println(name+" 正在发售第 "+tickets--+" 张票 ");
                } else {
                    break;
                }
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

接下来重新运行测试类 Example05，运行结果如图 6-8 所示。

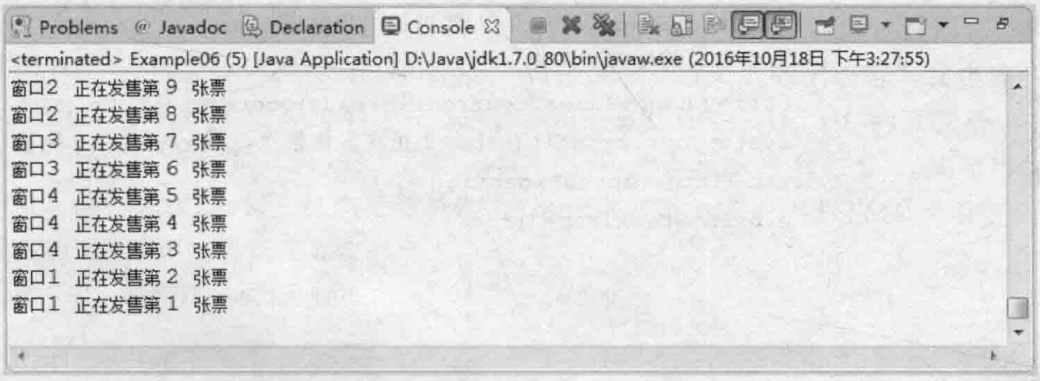


图 6-8　运行结果

从图 6-8 中可以看出，售出的票不再出现 0 和负数的情况，这是因为售票的代码实现了同步，之前出现的线程安全问题得到解决。

6.4.3　同步方法

除了可以将需要的代码设置成同步代码块以外，也可以使用 synchronized 关键字将一个方法修饰成同步方法，它能实现和同步代码块同样的功能，具体语法格式如下：

```
权限修饰符 synchronized 返回值类型 方法名([参数 1,...]){
    需要同步的代码;
}
```

被 synchronized 修饰的方法在某一时刻只允许一个线程访问，访问该方法的其他线程都会发生阻塞，直到当前线程访问完毕后，其他线程才有机会执行该方法。

接下来再对文件 6-9 进行修改，使用 synchronized 关键字把 saleTicket()修饰为同步方法，使用同步方法的方式解决线程安全问题，如文件 6-12 所示。

文件 6-12 Ticket.java

```
1  package com.itheima.example05;
2  class Ticket implements Runnable {
3      private int tickets=100;
4      public void run() {
5          while (true) {
6              saleTicket();                    //调用售票方法
7              if (tickets <=0) {
8                  break;
9              }
10         }
11     }
12     //定义一个同步方法 saleTicket()
13     private synchronized void saleTicket() {
14         if (tickets >0) {
15             try {
16                 Thread.sleep(10);     //经过的线程休眠 10ms
17                 //获得当前线程的名称
18                 String name=Thread.currentThread().getName();
19                 System.out.println(name+" 正在发售第 "+tickets--+" 张票 ");
20             } catch (InterruptedException e) {
21                 e.printStackTrace();
22             }
23         }
24     }
25 }
```

再运行测试类 Example05，运行结果如图 6-9 所示。

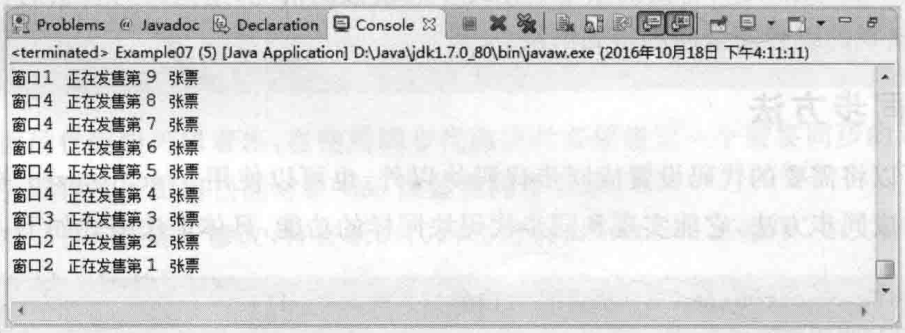


图 6-9 运行结果

从图 6-9 中可以看出，此代码完成了与之前同步代码块同样的功能。需要注意的是，同步方法的锁是当前调用该方法的对象，也就是 this 指向的对象。

第7章 网页开发基础

在学习 Java Web 开发之前，读者有必要先了解一些网页开发的基础知识，以便能够在项目的开发过程中看懂页面信息。本章将对 HTML 技术、CSS 技术以及 JavaScript 技术的基础知识进行详细讲解。

7.1 HTML 技术

7.1.1 HTML 简介

HTML(Hyper Text Markup Language)，中文译为“超文本标记语言”，其主要作用是通过 HTML 标记对网页中的文本、图片、声音等内容进行描述。HTML 网页就是一个文件名后缀为.html 或.htm 的文件，它可以用记事本打开，所以简单的 HTML 代码可以在记事本中编写。编写完成后，将文件名后缀修改为“html”，即可生成一个 HTML 网页。

基本的 HTML 文档结构如文件 7-1 所示。

文件 7-1　htmlDemo01.html

```
1  <! DOCTYPE HTML>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>Insert title here</title>
6  </head>
7  <body>
8  </body>
9  </html>
```

在文件 7-1 中，带有＜ ＞符号的元素被称为 HTML 标记，标记是放在标记符(＜ ＞)中的用于表示某个功能的编码命令，也称为 HTML 标签或 HTML 元素。下面介绍文件 7-1 中的各个标记。

1. ＜!DOCTYPE＞标记

＜!DOCTYPE＞标记位于文档的最前面，用于向浏览器说明当前文档使用的是哪种 HTML 标准规范，如文件 7-1 中使用的是 HTML5。通常情况下，网页必须在开头处使用＜!DOCTYPE＞标记为所有的 HTML 文档指定 HTML 版本和类型，只有这样浏览器才能将该网页作为有效的 HTML 文档，并按指定的文档类型进行解析。＜!DOCTYPE＞标

记和浏览器的兼容性相关，删除<!DOCTYPE>后，会把如何展示 HTML 页面的权力交给浏览器，这时，有多少种浏览器，页面就有可能有多少种显示效果，在实际开发中，这是不被允许的。

2. <html>标记

<html>标记位于<!DOCTYPE> 标记之后，也称为根标记，用于告知浏览器其自身是一个 HTML 文档。该标记以<html>开始，以</html>结束，在<html>和</html>之间的内容是文档的头部和主体。

3. <head>标记

<head>标记用于定义 HTML 文档的头部信息，也称为头部标记，紧跟在<html>标记之后，主要用于封装其他位于文档头部的标记，如<title>、<meta>、<link>及<style>等。其中<title>标记用于描述文档的标题，<meta>标记可提供有关页面的元信息，也可指定字符编码。<link>标记用于定义文档与外部资源的关系，其最常见的用途是链接样式表，<style>标记用于为 HTML 文档定义样式信息。

需要注意的是，一个 HTML 文档只能含有一对<head>标记，并且绝大多数文档头部包含的数据都不会真正作为内容显示在页面中。

4. <body>标记

<body>标记用于定义 HTML 文档所要显示的内容，也称为主体标记。浏览器中显示的所有文本、图像、音频和视频等信息都必须位于<body>标记内，也就是说，<body>标记中的信息才是最终展示给用户的信息。一个 HTML 文档只能含有一对<body>标记，<body>标记必须在<html>标记内，且位于<head>头部标记之后，与<head>标记是并列关系。

了解了文件 7-1 中的 HTML 标记后，下面学习 HTML 文件的创建以及运行。以 htmlDemo01.html 文件为例，首先，创建一个名称为 chapter07 的文件夹，然后在文件夹中新建一个 txt 结尾的文本文件，将文件 7-1 中的内容编写到文件中，保存文件后，将文件的名称更改为 htmlDemo01，后缀名改为 html，此时 htmlDemo01.html 文件就是一个 HTML 页面，如图 7-1 所示。

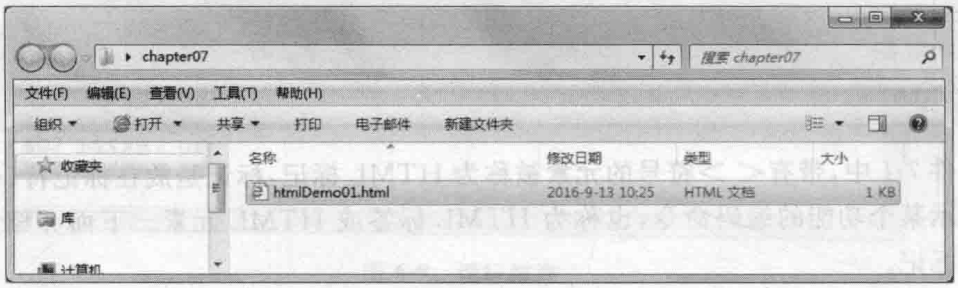


图 7-1 HTML 页面

双击图 7-1 中的 htmlDemo01.html 文件，即可通过系统默认的浏览器打开页面。由于此时的 HTML 页面中没有添加任何内容，所以打开后是一个空白页面。为了查看页面的显示效果，可以使用记事本打开文件，并在两个<body>标记之间添加一句话，如下所示：

```
<body>
    这是一个 HTML!
</body>
```

保存后，再次使用浏览器打开该页面，显示结果如图 7-2 所示。

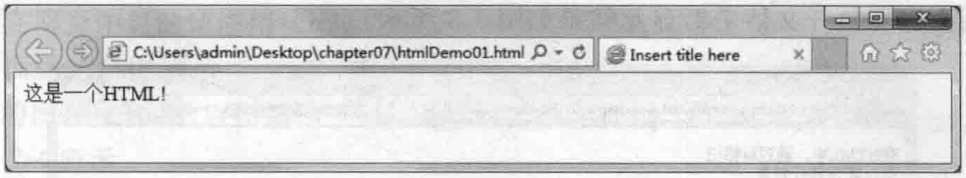


图 7-2　第一个 HTML 页面

从图 7-2 中可以看出，<body></body>标记中所添加的内容已在浏览器中显示出来。

7.1.2　单标记和双标记

HTML 中的标记分为单标记和双标记。其中单标记也称空标记，是指用一个标记符号即可完整描述某个功能的标记。其基本语法格式如下：

```
<标记名/>
```

例如标记<hr/>就是单标记，该标记用于定义一条水平线。需要注意的是，在标记名与/之间有一个空格，虽然在显示效果上有没有空格都一样，但是按照规范的要求，建议加上空格。

双标记也称体标记，是指由开始和结束两个标记符组成的标记。其基本语法格式如下：

```
<标记名>内容</标记名>
```

在上述的语法中，<标记名>表示该标记的作用开始，一般称为开始标记(start tag)，</标记名>表示该标记的作用结束，一般称为结束标记(end tag)，和开始标记相比，结束标记只是在前面增加了一个关闭符/，如文件 7-1 中的<html></html>和<head></head>等都是双标记。

7.1.3　文本控制与文本样式标记

**1. 段落标记 和换行标记
**

为了使网页中的文字有条理地显示出来，HTML 中提供了段落标记 ，如果希望某段文本强制换行显示，则可以使用换行标记
。下面通过一个案例演示这两种标记的使用。

在 chapter07 文件夹中新建一个名称为 htmlDemo02 的 HTML 文件，编辑后的主要代码如文件 7-2 所示。

文件 7-2 htmlDemo02.html

```
<body>
    <p>在 HTML 中,通过 br 标记<br />可以实现换行效果</p>
</body>
```

使用浏览器打开文件 7-2,显示结果如图 7-3 所示。

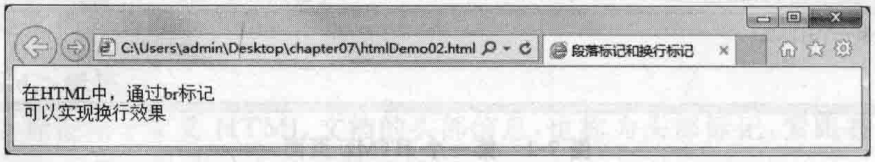


图 7-3 换行标记的使用

从图 7-3 中可以看出,使用换行标记
的文本实现了强制换行的效果。

2. 文本样式标记

在 HTML 中,<font>标记用于控制网页中文本的样式,如字体、字号和字体颜色等。其基本语法格式如下:

```
<font 属性="属性值">文本内容</font>
```

下面通过案例演示<font>标记的使用。在 chapter07 文件夹中新建一个名称为 htmlDemo03 的 HTML 文件,编辑后的主要代码如文件 7-3 所示。

文件 7-3 htmlDemo03.html

```
<body>
    这是默认样式的文本。<br/>
    <font face="微软雅黑" size="5" color="blue"><br/>
    这是 5 号蓝色文本,字体是微软雅黑! </font>
</body>
```

在文件 7-3 中,第 2 行的文本为 HTML 默认文本样式,第 3 行代码使用<font>标记的 face、size 以及 color 属性分别设置了文本的字体、字号以及字体颜色。使用浏览器打开文件 7-3,显示结果如图 7-4 所示。

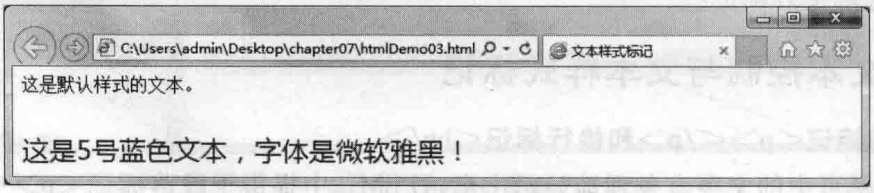


图 7-4 使用<font>标记设置文本样式

7.1.4 图像标记

<img/>标记也称为图像标记,主要用于显示 HTML 页面中的图像。其基本的语法

格式如下：

```
<img src="图像 URL" />
```

在上述的语法中，src 属性用于指定图像文件的路径，该属性是＜img/＞标记必需的属性。

要想在网页中灵活地应用＜img/＞标记，只使用 src 属性是不够的，还需要其他属性的配合。下面通过案例演示＜img/＞标记的用法。在 chapter07 文件夹中添加一个名称为 logo.jpg 的图片文件，然后创建一个名称为 htmlDemo04 的 HTML 文件，编辑后的主要代码如文件 7-4 所示。

文件 7-4　htmlDemo04.html

```
1  <body>
2      显示图片:<img src="logo.jpg" width="271" height="94" border="0" />
3  </body>
```

在文件 7-4 中，src 属性的值可以是绝对路径，也可以是相对路径。width 和 height 属性分别用于设置图像的宽度和高度，单位为像素，border 属性用于设置图像的边框，border＝"0"表示无边框。使用浏览器打开文件 7-4，显示结果如图 7-5 所示。

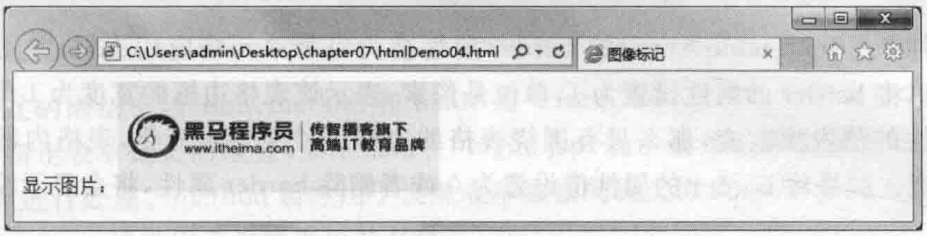


图 7-5　使用＜img/＞标记显示图片

从图 7-5 中可以看出，浏览器中已经显示出了相应的图片。需要注意的是，路径中的图片文件必须存在，如果不存在，则浏览器将无法显示。

7.1.5　表格标记

在 HTML 文件中，表格标记用于在页面中创建表格，其基本语法格式如下所示：

```
<table>
    <tr>
        <td>单元格内的文字</td>
    </tr>
</table>
```

在上述的语法中，包含三对 HTML 标记，分别为＜table＞＜/table＞、＜tr＞＜/tr＞和＜td＞＜/td＞，它们是创建表格的基本标记，缺一不可。＜table＞＜/table＞用于定义一个表格，＜tr＞＜/tr＞用于定义表格中的行，必须嵌套在＜table＞＜/table＞标记中，＜td＞＜/td＞用于定义表格中的单元格，也可称为表格中的列，必须嵌套在＜tr＞＜/tr＞标记中。

下面通过案例演示<table>标记的使用。在 chapter07 文件夹中创建一个名称为 htmlDemo05 的 HTML 文件，编辑后的关键代码如文件 7-5 所示。

文件 7-5 htmlDemo05.html

```
<body>
<table border="1">
    <tr>
        <td>姓名</td>
        <td>性别</td>
        <td>年龄</td>
        <td>职业</td>
    </tr>
    <tr>
        <td>韩强</td>
        <td>男</td>
        <td>30</td>
        <td>课程设计师</td>
    </tr>
</table>
</body>
```

在文件 7-5 中，<table>标记的 border 属性会为每个单元格应用边框，并用边框围绕表格。这里将 border 的属性设置为 1，单位是像素，表示该表格边框的宽度为 1 像素。如果 border 属性的值发生改变，那么只有围绕表格的边框尺寸会发生变化，表格内部的边框还是 1 像素宽。如果将 border 的属性值设置为 0 或者删除 border 属性，将会显示没有边框的表格。

使用浏览器打开文件 7-5，其显示结果如图 7-6 所示。

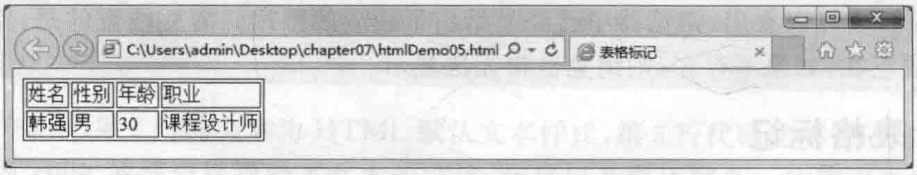


图 7-6 有边框表格的显示效果

将 border 属性值设置为 0 后，保存文件并刷新浏览器页面，其显示结果如图 7-7 所示。

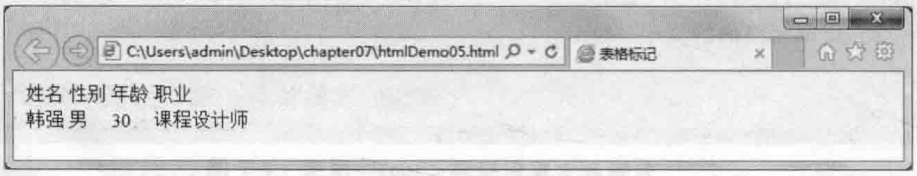


图 7-7 无边框表格的显示效果

从图 7-7 的显示结果中可以看出，表格中的内容依然整齐有序地排列着，但已看不到边框。需要注意的是，在实际开发中，最好不要规定边框，而是使用 CSS 添加边框样式和

颜色。

7.1.6　表单标记

简单地说，表单就是在网页上用于输入信息的区域，其主要功能是收集用户信息，并将这些信息传递给后台服务器。表单在互联网上随处可见，例如，网站注册页面中的用户名和密码输入框、性别选择按钮、提交按钮等都是用表单中的相关标记定义的。

表单主要由表单域、表单控件和提示信息三部分构成，其详细介绍如下所示。

- 表单域：相当于一个容器，用于容纳所有的表单控件和提示信息。
- 表单控件：包含了具体的表单功能项，如单行文本输入框、密码输入框、复选框提交按钮等。
- 提示信息：一个表单中通常还需要包含一些说明性的文字，即表单控件前的文字说明，用于提示用户进行填写和操作。

1. 表单域

在 HTML 中，<form>标记用于定义一个表单域，即创建一个表单，其基本语法如下所示：

```
<form action="url 地址" method="提交方式" name="表单名称">
    各种表单控件
</form>
```

在上述的语法中，action、method 和 name 是<form>标记中的常用属性，其中，action 属性用于指定表单提交的地址，如 action="login. jsp"表示表单数据会提交到名为 login. jsp 的页面进行处理。method 属性用于设置表单数据的提交方式，其取值为 GET(默认值)或 POST。name 属性用于设置表单的名称。

需要注意的是，GET 方式提交的数据会显示在浏览器的地址栏中，保密性差且有数据量限制(2Kb 左右)，而使用 POST 提交方式不但保密性好，还可以提交大量的数据，所以开发中通常使用 POST 方式提交表单。

2. 表单控件<input/>

在一个表单中，大部分的表单控件(如单行文本输入框、单选按钮、复选框等)都是由<input/>元素定义的，该元素定义表单控件的基本语法格式如下：

```
<input type="控件类型" />
```

在上述语法中，type 属性是其基本的属性，它有很多属性值，通过不同的属性值可以指定不同的控件类型，例如，当其值为 text 时，表示文本输入框控件；当其值为 button 时，表示按钮控件等。

除了 type 属性以外，<input/>控件还可以定义很多其他属性，如 id、name、value、size 等，它们分别用于指定<input/>控件的 ID 值、名称、控件中的默认值和控件在页面中的显示宽度。

下面通过案例演示<input/>控件的使用。在 chapter07 文件夹下创建一个名称为

htmlDemo06 的 HTML 文件，在该文件中使用表单控件实现一个注册页面，其关键代码如文件 7-6 所示。

文件 7-6 htmlDemo06.html

```
<body>
<fieldset>
<legend>注册新用户</legend>
    <! --表单数据的提交方式为 POST -->
    <form action="#" method="post">
        <table cellpadding="2" align="center">
            <tr>
                <td align="right">用户名:</td>
                <td>
                    <!--1.文本输入框控件 -->
                    <input type="text" name="username" />
                </td>
            </tr>
            <tr>
                <td align="right">密码:</td>
                <!--2.密码输入框控件 -->
                <td><input type="password" name="password" /></td>
            </tr>
            <tr>
                <td align="right">性别:</td>
                <td>
                    <!--3.单选输入框控件,由于无法输入 value, 所以预先定义好 -->
                    <input type="radio" name="gender" value="male" />男
                    <input type="radio" name="gender" value="female" />女
                </td>
            </tr>
            <tr>
                <td align="right">兴趣:</td>
                <td>
                    <!--4.复选框控件 -->
                    <input type="checkbox" name="interest" value="film" />看电影
                    <input type="checkbox" name="interest" value="code" />敲代码
                    <input type="checkbox" name="interest" value="game" />玩游戏
             </td>
            </tr>
              <tr>
                  <td align="right">头像:</td>
                  <td>
                      <!--5.文件上传控件 -->
                      <input type="file" name="photo" />
```

```
41                  </td>
42              </tr>
43              <tr>
44                  <td colspan="2" align="center">
45                      <!--6.提交按钮控件 -->
46                      <input type="submit" value="注册" />
47                      <!--7.重置按钮控件,单击后会清空当前 form -->
48                      <input type="reset" value="重填" />
49                  </td>
50              </tr>
51          </table>
52      </form>
53    </fieldset>
54 </body>
```

在文件 7-6 中，分别使用<input />标记定义了文本输入框控件、密码输入框控件、单选按钮和复选框控件、文件上传控件以及提交和重置按钮控件。在上述控件中，name 属性代表控件名称，value 属性表示该控件的值。上述代码中，还使用了<fieldset>和<legend>标记，<fieldset>标记的作用是将表单内的元素分组，而<legend>标记则为<fieldset>标记定义标题。

需要注意的是，单选按钮控件和复选框控件必须指定相同的 name 值，这是为了方便在处理页面数据时获取表单传递的值(表单所传递的就是该控件的 value 值)。

与其他语言类似，HTML 中也提供了代码注释，它以<!--开头，以-->结束，其基本语法格式如下所示：

```
<! --注释内容-->
```

文件 7-6 大量使用了 HTML 注释，被注释的内容不会显示到浏览器页面上，但是通过查看浏览器中的源代码可以看到 HTML 中的注释信息。

使用浏览器打开文件 7-6，显示结果如图 7-8 所示。

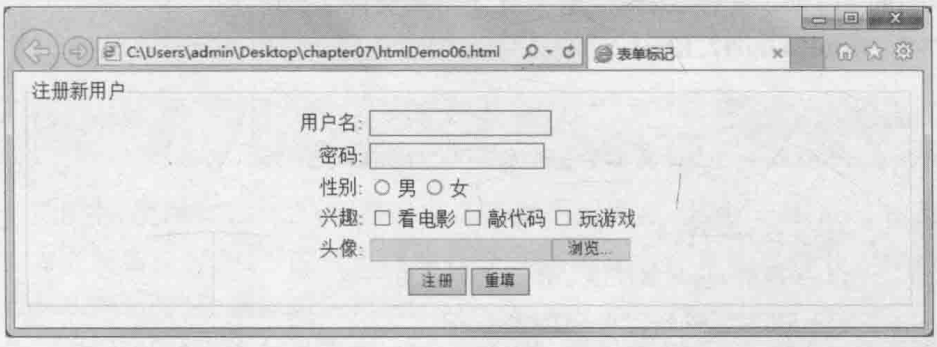


图 7-8　表单控件的使用

填写完图 7-8 中的表单数据后，页面如图 7-9 所示。

从图 7-9 中可以看出，密码输入框中的内容为不可见状态，单选按钮只能选择一个值，



图 7-9 表单控件的使用

而复选框可以选择多个值。

7.1.7 列表标记和超链接标记

1. 无序列表<ul>

在 HTML 中，专门用于展示列表信息的标记被称为列表标记。列表标记分为有序列表和无序列表，由于在 HTML 中无序列表较为常用，所以下面只针对无序列表进行讲解。

无序列表是指列表的各个列表项之间没有顺序级别之分，通常是并列的列表。定义无序列表的基本语法格式如下所示：

```
<ul>
    <li>列表项 1</li>
    <li>列表项 2</li>
    <li>列表项 3</li>
    ⋮
</ul>
```

在上述语法中，
标记用于定义无序列表，
标记嵌套在
标记中，用于描述具体的列表项，每对
中至少应包含一对
。

下面通过案例演示
标记的使用。在 chapter07 文件夹中创建一个名称为 htmlDemo07 的 HTML 文件，编辑后的关键代码如文件 7-7 所示。

文件 7-7 htmlDemo07.html

```
1  <body>
2      <font size="5">计算机专业相关学科</font><br />
3      <ul>
4          <li>JAVA</li>
5          <!--指定 type 属性值 ,disc 为默认值-->
6          <li type="disc">Android</li>
7          <li type="square">PHP</li>
8          <li type="circle">UI 设计 </li>
9      </ul>
10 </body>
```

在文件 7-7 中，
标记的 type 属性用于指定列表项目符号，type 常用的属性值有三种：disc、square 和 circle，它们的显示效果分别是●、■和○，type 属性的默认值是 disc。使用浏览器打开文件 7-7，显示结果如图 7-10 所示。

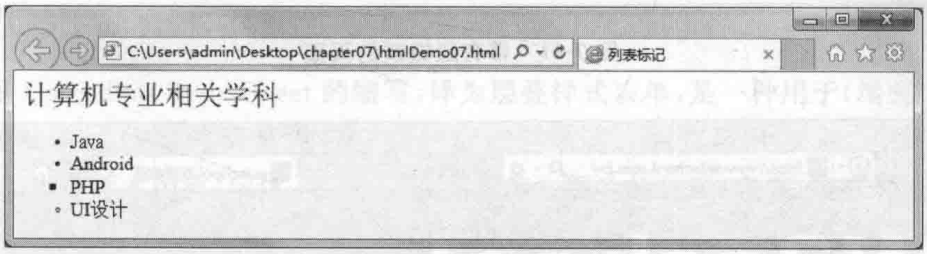


图 7-10　无序列表的显示效果

2. 超链接标记

一个网站通常由多个页面构成，进入网站时首先看到的是首页面，如果要从首页面跳转到子页面，就需要在首页面的相应位置添加超链接。在 HTML 中，创建超链接非常简单，只需要用<a>标记环绕需要被链接的对象即可。其基本语法格式如下：

```
<a href="跳转目标" target="目标窗口的弹出方式">文本或图像</a>
```

在上述语法中，<a>标记用于定义超链接，href 属性用于指定链接指向的页面的 URL，当在<a>标记中使用 href 属性时，该标记就具有了超链接的功能；target 属性用于指定页面的打开方式，其取值为_self 和_blank，其中_self 为默认值，意为在原窗口打开，_blank 为在新窗口打开。

下面通过案例演示<a>标记的使用。在 chapter07 文件夹中创建一个 HTML 文件 htmlDemo08，其关键代码如文件 7-8 所示。

文件 7-8　htmlDemo08.html

```
<body>
    在新窗口打开：
    <a href="http://www.w3school.com.cn/" target="_blank">w3school 在线教程</a>
    <br />
    在原窗口打开：
    <a href="https://www.baidu.com/" target="_self">百度</a>
</body>
```

在文件 7-8 中，使用<a>标记定义了两个超链接，其中 w3school 首页链接的打开方式设置为在新窗口打开，百度首页面的打开方式设置为在原窗口打开。使用浏览器打开 htmlDemo08.html 的效果如图 7-11 所示。

从图 7-11 中可以看到两个超链接，单击“w3school 在线教程”链接后，页面跳转效果如图 7-12 所示。

从图 7-12 中可以看出，当单击“w3school 在线教程”链接时，w3school 首页面在新窗口被打开。关闭新窗口，然后回到原窗口单击“百度”链接时，页面跳转效果如图 7-13 所示。

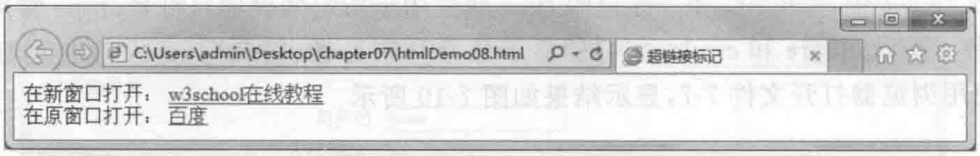


图 7-11 带有超链接的页面



图 7-12 链接页面在新窗口打开



图 7-13 链接页面在原窗口打开

从图 7-13 中可以看出，当单击“百度”链接后，“百度”首页面已在原窗口被打开。

7.1.8 标记

div 是 DIVision 的缩写，意为“分割、区域”。 标记简单而言就是一个区块容器标记，可以将网页分割为独立且不同的部分，以实现网页的规划和布局。在 HTML 页面中，它以 开头，并以 结尾，在 与 之间可以容纳段落、标题、图像等各种网页元素，也就是说，大多数 HTML 标记都可以嵌套在 标记中，并且 还可以嵌套多层 ，其使用方式如下：

```
<div>
    <h3>这是一个标题</h3>
    <p>这是一个段落</p>
</div>
```

在实际开发中， 标记常常与 CSS 技术搭配使用，在第 7.2 节讲解 CSS 技术的过程中会对 标记的具体使用进行举例，这里不再赘述。

7.2 CSS 技术

7.2.1 CSS 简介

CSS 是 Cascading Style Sheet 的缩写，译为层叠样式表单，是一种用于(增强)控制网页样式并允许将样式信息与网页内容分离的标记性语言。在实际开发中，主要用于设置 HTML 页面中的文本内容(字体、字号、对齐方式等)、图片的外形(宽高、边框样式、边距等)以及版面的布局等外观显示样式。

CSS 定义的规则具体如下：

```
选择器{属性 1:属性值 1; 属性 2:属性值 2; 属性 3:属性值 3;}
```

在上述样式规则中，选择器用于指定 CSS 样式作用的 HTML 对象，大括号内的属性是对该对象设置的具体样式。其中，属性和属性值以“键值对”的形式出现，它们之间用“:”(英文冒号)连接，多个“键值对”之间用“;”(英文分号)进行分隔。

接下来通过 CSS 样式对＜div＞标记进行设置，具体示例如下：

```
div{ border:2px solid blue; width: 300px;  height: 200px;}
```

上面的代码就是一个完整的 CSS 样式。其中，div 为选择器，表示 CSS 样式作用的 HTML 对象；border、width 和 height 为 CSS 属性，分别表示边框、宽度和高度，其中 border 属性的三个值“2px solid blue;”分别表示该边框为 2 像素、实心边框线、蓝色。

在 CSS 中，通常使用像素单位 px 作为计量文本和边框等元素的标准量，px 是相对于显示器屏幕分辨率而言的，而百分比(%)是相对于父对象而言的，例如一个元素呈现的宽度是 400px，子元素设置为 50%，那么子元素所呈现的宽度为 200px。

在 CSS 中，颜色的取值方式主要有三种。

(1) 预定义的颜色值：如 red、green、blue 等。

(2) RGB 代码：如红色可以用 rgb(255,0,0)或 rgb(100%,0%,0%)表示。如果使用 RGB 代码百分比方式取颜色值，即使值为 0，也不能省略百分号，必须写为 0%。

(3) 十六进制：如＃FF0000、＃FF6600、＃29D794 等。在实际工作中，十六进制是最常用的定义颜色的方式。

7.2.2 CSS 样式的引用方式

要使用 CSS 修饰网页，就需要在 HTML 文档中引入 CSS。通常在实际项目开发中，引入 CSS 会采用两种方式：内嵌式和链入式。下面将对这两种引入方式进行详细讲解。

1. 内嵌式

内嵌式是将 CSS 代码集中写在 HTML 文档的＜head＞头部标记中，并用＜style＞标记定义，其基本语法格式如下：

```
<head>
    <style type="text/css">
        选择器 {属性 1:属性值 1; 属性 2:属性值 2; 属性 3:属性值 3;}
    </style>
</head>
```

在上述语法中，<style>标记一般位于<head>标记中的<title>标记之后，因为浏览器是从上向下解析代码的，把 CSS 代码放在头部便于提前被加载和解析，以避免网页内容加载后由于没有样式修饰而带来的问题。同时，必须设置 type 的属性值为 text/css，这样浏览器才知道<style>标记包含的是 CSS 代码。

下面通过案例学习如何在 HTML 文件中使用内嵌式引入 CSS。在 chapter07 文件夹中创建一个名称为 cssDemo01 的 HTML 文件，编辑后的关键代码如文件 7-9 所示。

文件 7-9　cssDemo01.html

```
1  <head>
2  <meta charset="UTF-8">
3  <title>使用 CSS 内嵌式</title>
4  <style type="text/css">
5      /* 定义标题标记居中对齐 */
6      h2{ text-align:center;}
7      /* 定义 div 标记样式 */
8      div{ border: 1px solid blue; width: 500px; height: 50px; color:red;}
9  </style>
10 </head>
11 <body>
12     <h2>CSS 样式</h2>
13     <div>
14         style 标记一般位于 head 标记中，使用它可定义内嵌式 CSS 样式表。
15     </div>
16 </body>
```

在文件 7-9 中，HTML 文档的头部使用 style 标记定义内嵌式 CSS 样式，第 6 行使用了标题标记<h2>设置标题，第 8 行定义了 标记的样式。使用浏览器打开文件 7-9，显示效果如图 7-14 所示。

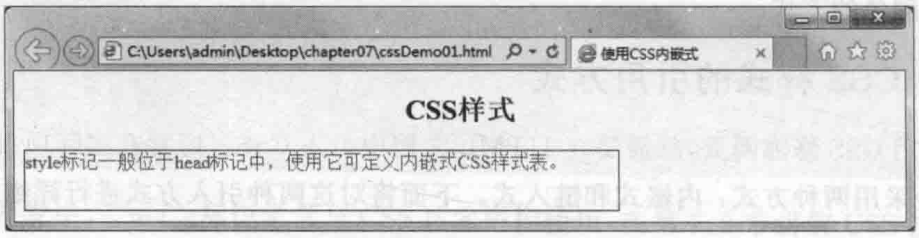


图 7-14　内嵌式效果展示

内嵌式引入 CSS 只对其所在的 HTML 页面有效，因此，仅设计一个页面时，使用内嵌

式是个很好的选择。但如果是一个网站，则不建议使用这种方式，因为它不能充分发挥 CSS 代码的重用优势。

2. 链入式

链入式是将所有的样式放在一个或多个以 css 为扩展名的外部样式表文件中，通过＜link/＞标记将外部样式表文件链接到 HTML 文件中，其基本语法格式如下：

```
<head>
    <link href="CSS 文件的路径" type="text/css" rel="stylesheet" />
</head>
```

在上述语法中，＜link/＞标记需要放在＜head＞头部标记中，并且必须指定＜link/＞标记的三个属性。

(1) href：定义所链接外部样式表文件的地址，可以是相对路径，也可以是绝对路径。

(2) type：定义所链接文档的类型，这里需要指定为 text/css，表示链接的外部文件为 CSS。

(3) rel：定义当前文档与被链接文档之间的关系，这里需要指定为 stylesheet，表示被链接的文档是一个样式表文件。

接下来通过修改文件 7-9 演示链入式 CSS 的引用方式。具体步骤如下。

(1) 创建 CSS 文件。在 chapter07 文件夹中创建一个名为 style. css 的文件，使用记事本打开后，将文件 7-9 中＜style＞＜/style＞标记之间的内容复制到 style. css 中并保存。

(2) 创建 HTML 文件，引入 CSS。在 chapter07 文件夹中创建一个名为 cssDemo02 的 HTML 文件，将文件 7-9 中的代码复制到 cssDemo02 中，并将＜style＞标记更改为链入式的形式，更改后的代码如下所示：

```
<head>
    <meta charset="UTF-8">
    <title>使用链入式 CSS 样式表</title>
    <link href="style.css" type="text/css" rel="stylesheet" />
</head>
```

在上述代码中，使用＜link/＞标记链入了 style. css 文件，代替了内嵌式的＜style＞标记。使用浏览器打开文件 cssDemo02. html，显示效果如图 7-15 所示。

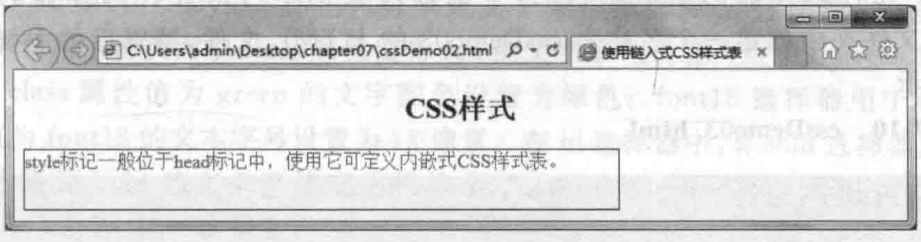


图 7-15　链入式效果展示

从图 7-15 中可以看到，使用链入式与内嵌式引入 css 文件的显示效果是一样的。在实

际开发中，链入式是使用频率最高且最实用的引入方式，它将 HTML 代码与 CSS 代码分离为两个或多个文件，实现了结构和表现的完全分离，同一个 CSS 文件可以被不同的 HTML 页面链接使用，同时，一个 HTML 页面也可以通过多个<link/>标记链接多个 CSS 文件，大大提高了网页开发的工作效率。

7.2.3 CSS 选择器和常用属性

要想将 CSS 样式应用于特定的 HTML 元素，首先需要找到该目标元素。在 CSS 中，执行这一任务的部分被称为选择器，本节将对 CSS 的基础选择器进行讲解。

1. 标记选择器

标记选择器是指用 HTML 标记名称作为选择器，按标记名称分类，为页面中的某一类标记指定统一的样式。其基本语法格式如下：

```
标记名{属性 1:属性值 1; 属性 2:属性值 2; 属性 3:属性值 3; }
```

上述语法中，所有的 HTML 标记都可以作为标记选择器的标记名，例如<body>标记、<h1>标记、<p>标记等。通过标记选择器定义的样式对页面中该类型的所有标记都有效，这是它的优点，但同时这也是其缺点，因为这样就不能设计差异化样式。

2. 类选择器

类选择器使用“.”(英文点号)进行标识，后面紧跟类名，其基本语法格式如下：

```
.类名{属性 1:属性值 1; 属性 2:属性值 2; 属性 3:属性值 3; }
```

上述语法中，类名即为 HTML 页面中元素的 class 属性值，大多数 HTML 元素都可以定义 class 属性。类选择器最大的优势是可以为元素对象定义单独或相同的样式。

3. id 选择器

id 选择器使用“#”进行标识，后面紧跟 id 名，其基本语法格式如下：

```
#id 名{属性 1:属性值 1; 属性 2:属性值 2; 属性 3:属性值 3; }
```

在上述语法中，id 名即为 HTML 页面中元素的 id 属性值，大多数 HTML 元素都可以定义 id 属性，元素的 id 值是唯一的，只能对应于文档中某一个具体的元素。

了解了几种选择器的语法结构后，下面通过案例学习这几种选择器的使用。在 chapter07 文件夹中创建一个名称为 cssDemo03 的 HTML 文件，编辑后的主要代码如文件 7-10 所示。

文件 7-10 cssDemo03.html

```
1  <html>
2  <head>
3  <title>选择器</title>
4  <style type="text/css">
5  /* 1.类选择器的定义 */
```

```
6   .red {
7       color: red;
8   }
9   .green {
10      color: green;
11  }
12  .font18 {
13      font-size: 18px;
14  }
15  /* 2.id选择器的定义 */
16  #bold {
17      font-weight: bold;
18  }
19  #font24 {
20      font-size: 24px;
21  }
22  </style>
23  </head>
24  <body>
25      <!--类选择器的使用-->
26      <h1 class="red">标题一:class="red",设置文字为红色。</h1>
27      <p class="green font18">
28          段落一: class="green font18",设置文字为绿色,字号为 18px。
29      </p>
30      <p class="red font18">
31          段落二: class="red font18",设置文字为红色,字号为 18px。
32       </p>
33      <!--id选择器的使用-->
34      <p id="bold">段落 1:id="bold",设置粗体文字。</p>
35      <p id="font24">段落 2:id="font24",设置字号为 24px。</p>
36      <p id="font24">段落 3:id="font24",设置字号为 24px。</p>
37      <p id="bold font24">段落 4:id="bold font24",同时设置粗体和字号 24px。</p>
38  </body>
39  </html>
```

在文件 7-10 中,在＜style＞标记内分别定义了类选择器和 id 选择器。在类选择器中,.red 选择器用于将页面中 class 属性值为 red 的文字颜色设置为红色;.green 选择器用于将页面中 class 属性值为 green 的文字颜色设置为绿色;.font18 选择器用于将页面中 class 属性值为 font18 的文本字号设置为 18 像素。在 id 选择器中,#bold 选择器用于将页面中 id 属性值为 bold 的文本字体变为粗体文字;#font24 用于将页面中 id 属性值为 font24 的文本字号设置为 24 像素。

使用浏览器打开文件 cssDemo03.html,显示结果如图 7-16 所示。

在图 7-16 中,标题一和段落二的文本内容均显示为红色,这说明同一个类选择器可以被多个标记引用。段落 2 和段落 3 由于引用了相同的 id 选择器,所以它们的字号均为 24

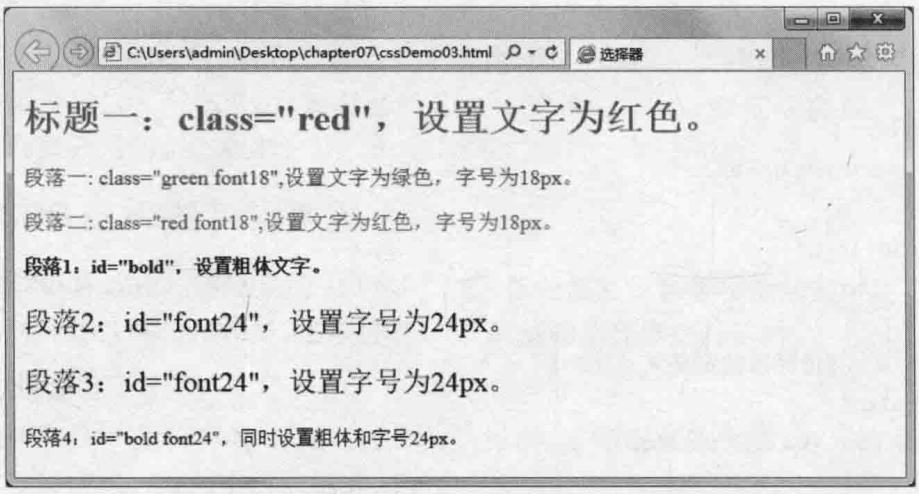


图 7-16　选择器的使用

像素，虽然浏览器并没有报错，但是这种做法是不被允许的，因为在 JavaScript 等脚本语言中 id 值是唯一的。段落 4 没有显示任何 CSS 样式，这说明同一个标记对象不能同时引用多个 id 选择器，如 id="bold font24"的引用方式是完全错误的，如果一个标记想要使用多个样式，可以使用类选择器。

一些常用的 CSS 属性如表 7-1 所示。

表 7-1　CSS 常用属性

属性名称	功能描述
margin	用于指定对象的外边距，也就是对象与对象之间的距离。该属性可指定 1～4 个属性值，各属性值之间以空格分隔
padding	用于指定对象的内边距，即对象的内容与对象边框之间的距离。该属性可指定 1～4 个属性值，各属性值之间以空格分隔
background	用于设置背景颜色、背景图片、背景图片的排列方式、是否固定背景图片及其位置。该属性可指定多个属性值，各属性值之间以空格分隔，没有先后顺序
border	用于设置边框的宽度、样式和颜色。该属性可以指定多个属性值，各属性值之间以空格分隔，没有先后顺序
font	用于设置字体样式、小型的大写字体、字体粗细、文字的大小、行高和文字的字体
height	用于指定对象的高度
width	用于指定对象的宽度
color	用于指定文本的颜色
text-align	用于指定文本的对齐方式
text-decoration	用于指定文本的显示样式，其属性值包括 line-through(删除线)、overline(上画线)、underline(下画线)、blink(闪烁效果，Firefox 和 Opera 中可以看到效果)和 none(无效果)等
display	用于指定对象的显示形式

7.3　JavaScript 基础

7.3.1　DOM 相关知识

DOM 是 Document Object Model(文档对象模型)的简称，是 W3C 组织推荐的处理可扩展标志语言的标准编程接口，它可以以一种独立于平台和语言的方式访问和修改一个文档的内容和结构。

W3C 中将 DOM 标准分为三个不同的部分：核心 DOM、XML DOM 和 HTML DOM，其中核心 DOM 是针对任何结构化文档的标准模型，XML DOM 是针对 XML 文档的标准模型，而 HTML DOM 是针对 HTML 文档的标准模型。由于本章中主要讲解的是网页开发的基础知识，主要涉及 HTML DOM，所以本节将主要介绍 HTML DOM 的相关知识。

HTML DOM 模型被构造为对象的树，该树的根节点是文档(Document)对象，该对象有一个 documentElement 的属性引用，表示文档根元素的 Element 对象。HTML 文档中表示文档根元素的 Element 对象是＜html＞元素，＜head＞和＜body＞元素可以视为树的枝干。

HTML DOM 树的结构如图 7-17 所示。

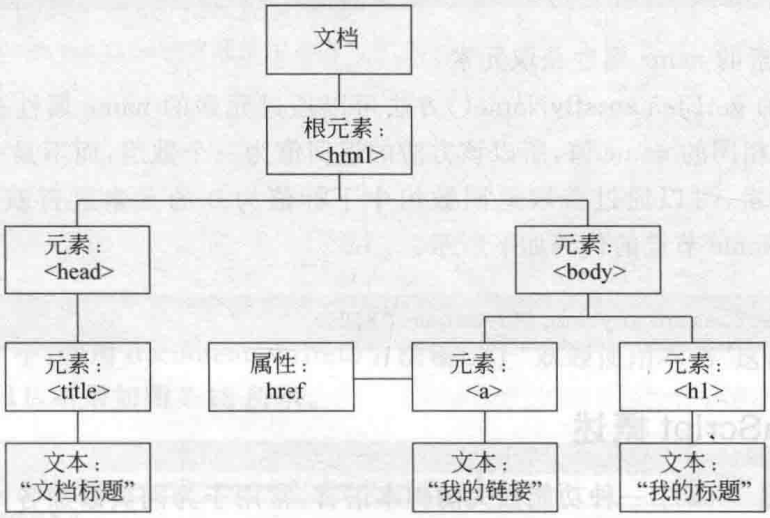


图 7-17　DOM 对象的文档层次结构

在图 7-17 中，每个元素被称为一个节点，直接位于一个节点之下的节点被称为该节点的子节点(childNode)，直接位于一个节点之上的节点被称为该节点的父节点(parentNode)，具有相同父节点的两个节点称为兄弟节点(siblingNode)。

1. 节点的访问

在 DOM 中，HTML 文档的各个节点被视为各种类型的 Node 对象。如果想要通过某个节点的子节点找到该元素，则其语法如下所示：

```
父节点对象=子节点对象.parentNode;
```

Node 对象的常用属性如表 7-2 所示。

表 7-2 Node 对象的常用属性

属　性	类　型	描　述
parentNode	Node	返回节点的父节点，没有父节点时为 null
childNodes	NodeList	返回节点到子节点的节点列表
firstChild	Node	返回节点的首个子节点，没有时为 null
lastChild	Node	返回节点的最后一个子节点，没有时为 null

2. 获取文档中的指定元素

通过遍历节点的访问可以找到文档中指定的元素，但是这种方法有些麻烦，document 对象中提供了直接搜索文档中指定元素的方法，具体如下。

(1) 通过元素的 id 属性获取元素

Document 的 getElementById()方法可以通过元素的 id 属性获取元素。例如，获取 id 属性值为 userId 节点的代码如下所示：

```
document.getElementById("userId");
```

(2) 通过元素的 name 属性获取元素

Document 的 getElementsByName()方法可以通过元素的 name 属性获取元素。由于多个元素可能有相同的 name 值，所以该方法的返回值为一个数组，而不是一个元素。如果想获得唯一的元素，可以通过获取返回数组中下标值为 0 的元素进行获取。例如，获取 name 值为 userName 节点的代码如下所示：

```
document.getElementsByName("userName")[0];
```

7.3.2 JavaScript 概述

JavaScript 是 Web 中一种功能强大的脚本语言，常用于为网页添加各式各样的动态功能，它不需要进行编译，直接嵌入在 HTML 页面中，就可以把静态的页面转变成支持用户交互并响应事件的动态页面。

1. JavaScript 的引入

在 HTML 文档中，引入 JavaScript 的方式主要有两种：一种是在 HTML 文档中直接嵌入 JavaScript 脚本，称为内嵌式；另一种是链接外部 JavaScript 脚本文件，称为外链式。

(1) 内嵌式

在 HTML 文档中，通过<script></script>标签及其相关属性可以引入 JavaScript 代码。当浏览器读取到<script>标签时，就会解释执行其中的脚本。JavaScript 内嵌式的使用方式如下：

```
<script type="text/javascript">
    //此处为 JavaScript 代码
</script>
```

在上述代码中，type 属性用于指定 HTML 文档引用的脚本语言类型，当 type 属性的值为 text/javascript 时，表示＜script＞＜/script＞元素中包含的是 JavaScript 脚本。

需要注意的是，在低版本的浏览器中，引入 JavaScript 脚本时，type＝"text/javascript" 是必须编写的，但现在已经可以不使用了，因为 JavaScript 是所有现代浏览器以及 HTML5 中的默认脚本语言。

了解了一些简单的 JavaScript 知识后，下面通过具体的案例演示如何在 HTML 文档中使用内嵌式 JavaScript。在 chapter07 文件夹中创建一个名为 jsDemo01 的 HTML 文件，其代码如文件 7-11 所示。

文件 7-11　jsDemo01.html

```
1  <!DOCTYPE HTML>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>内嵌式</title>
6  <script type="text/javascript">
7      document.write("欢迎使用本书");
8  </script>
9  </head>
10 <body>
11     <br/>学习 Java
12 </body>
13 </html>
```

在文件 7-11 中，使用 document.write()方法输出了“欢迎使用本书”这段文本。使用浏览器打开文件 7-11，结果如图 7-18 所示。

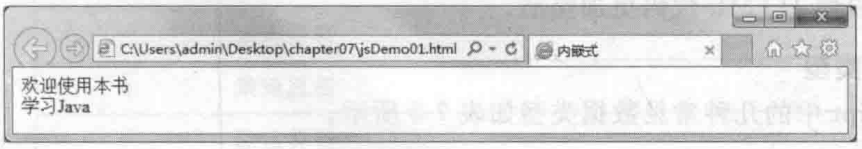


图 7-18　内嵌式

在图 7-18 中显示了两段文本，其中第一段文本“欢迎使用本书”就是由 JavaScript 代码定义的。

(2) 外链式

当脚本代码比较复杂或者同一段代码需要被多个网页文件使用时，可以将这些脚本代码放置在一个扩展名为 js 的文件中，然后通过外链式引入该 js 文件。在 Web 页面中使用外链式引入 JavaScript 文件的方式如下：

```
<script type="text/javascript" src="JS 文件的路径"></script>
```

下面通过修改文件 7-11 演示如何在 HTML 文件中通过外链式引入 JavaScript。

首先在 chapter07 文件夹中创建一个名为 demo01.js 的文件，使用记事本将该文件打开，并将文件 7-11 中的第 7 行代码“document.write("欢迎使用本书");”复制到该文件中并保存。

然后创建一个名称为 jsDemo02 的 HTML 文件，将 jsDemo01 中的代码复制到 jsDemo02 中，并更改为外链式，如文件 7-12 所示。

文件 7-12　jsDemo02.html

```
1  <!DOCTYPE HTML>
2  <html>
3  <head>
4  <meta charset="UTF-8">
5  <title>外链式</title>
6      <script type="text/javascript" src="demo01.js"></script>
7  </head>
8  <body>
9      <br/>学习 Java
10 </body>
11 </html>
```

在上述代码中，使用外链式引入了 demo01.js 文件，运行文件 7-12，结果如图 7-19 所示。

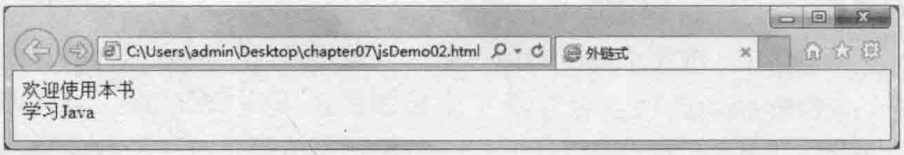


图 7-19　外链式

从图 7-19 中可以看出，外链式与内嵌式的显示结果相同。在实际开发中，如果页面中需要编写的 js 代码很少，则可以使用内嵌式，但是如果 js 代码很多，则通常使用外链式，使用外链式可以使 HTML 代码更加整洁。

2. 数据类型

JavaScript 中的几种常见数据类型如表 7-3 所示。

表 7-3　JavaScript 数据类型

类　型	含　义	说　明
Number	数值型	数值型数据不区分整型和浮点型，数值型数据不要用引号括起来
String	字符串类型	字符串是用单引号或双引号括起来的一个或多个字符
Boolean	布尔类型	只有 true 或 false 两个值
Object	对象类型	一组数据和功能的键值对集合
Null	空类型	没有任何值
Undefined	未定义类型	指变量被创建，但未赋值时所具有的值

3. 变量

在 JavaScript 中，使用 var 命令声明变量，由于 JavaScript 是一种弱类型语言，所以在声明变量时，不需要指定变量的类型，变量的类型将根据变量的赋值确定，其语法格式如下：

```
var  number=27;
var  str="Java";
```

变量的命名必须遵循命名规则，变量名可以由字母、下画线(_)、美元符号($)甚至中文组成，但不建议使用中文命名的方式，中间可以是数字、字母或下画线，但是不能有空格、加号、减号等符号，不能使用 JavaScript 的关键字。JavaScript 中的关键字如下所示：

```
abstract    continue    finally     instanceof    private       this
boolean     default     float       int           public        throw
break       do          for         interface     return        typeof
byte        double      function    long          short         true
case        else        goto        native        static        var
catch       extends     implements  new           super         void
char        false       import      null          switch        while
class       final       in          package       synchronized  with
```

需要注意的是，以上关键字同样不可以用作函数名、对象名及自定义的方法名等。

4. 运算符

运算符是程序执行特定算术或操作的符号，用于执行程序代码运算。JavaScript 中的运算符主要包括算术运算符、比较运算符、赋值运算符、逻辑运算符和条件运算符五种，常用的运算符如表 7-4 所示。

表 7-4　JavaScript 中的运算符

类　型	算术运算符	描　述
算术运算符	+	加运算符
	−	减运算符
	*	乘运算符
	/	除运算符
	++	自增运算符。该运算符有 i++(在使用 i 之后，使 i 的值加 1)和++i(在使用 i 之前，先使 i 的值加 1)两种
	−−	自减运算符。该运算符有 i−−(在使用 i 之后，使 i 的值减 1)和−−i(在使用 i 之前，先使 i 的值减 1)两种
比较运算符	<	小于
	>	大于
	<=	小于等于

续表

类　型	算术运算符	描　述
比较运算符	>=	大于等于
	==	等于。只根据表面值进行判断，不涉及数据类型。例如，"27"==27 的值为 true
	!=	不等于。只根据表面值进行判断，不涉及数据类型。例如，"27"!=27 的值为 false
逻辑运算符	&&	逻辑与，只有当两个操作数 a 和 b 的值都为 true 时，a&&b 的值才为 true；否则为 false
	||	逻辑或，只有当两个操作数 a 和 b 的值都为 false 时，a||b 的值才为 false；否则为 true
	!	逻辑非，!true 的值为 false，而!false 的值为 true
赋值运算符	=	例如，username="name"
	+=	例如，a+=b，相当于 a=a+b
	-=	例如，a-=b，相当于 a=a-b
	*=	例如，a*=b，相当于 a=a*b
	/=	例如，a/=b，相当于 a=a/b
	%=	例如，a%=b，相当于 a=a%b

除了表 7-4 中所列出的四种运算符以外，JavaScript 中还有一种条件运算符，它与 Java 中的三目运算符的使用类似，其语法格式如下：

```
操作数? 结果 1: 结果 2
```

如果操作数的值为 true，则整个表达式的结果为“结果 1”，否则为“结果 2”。其示例代码如下所示：

```
<script type="text/javascript">
    var a=5;
    var b=5;
    alert((a==b)? true:false);
</script>
```

在上述 js 代码中，由于声明的变量 a 和 b 的值相同，所以通过使用比较运算符==的比较结果为 true，此时整个 alert()语句的结果就为 true；如果变量 a 与 b 的值不相等时，则整个语句的执行结果为 false。

5. 条件语句 if

JavaScript 中最常用条件语句的是 if 判断语句，它的使用与 Java 语言中的 if 判断语句相似，通过判断条件表达式的值为 true 或者 false 从而确定是否执行某一条语句。其基本语法格式如下：

```
if(执行条件){
    语句 1;
}else{
    语句 2;
}
```

7.3.3　JavaScript 的使用

1. 函数的定义及调用

在 JavaScript 中，函数的定义是通过 function 语句实现的，其语法格式如下：

```
function functionName([parameter1,parameter2,...]){
    statements;
    [return expression;]
}
```

在上述语法中，functionName 是必选项，用于指定函数名，在同一个页面中，函数名必须是唯一的，并且区分大小写；parameter1，parameter2，... 是可选项，代表参数列表，当使用多个参数时，参数间使用逗号进行分隔，一个函数最多可以有 255 个参数；statements 是必选项，代表用于实现函数功能的语句；return expression 是可选项，用于返回函数值，expression 为任意表达式、变量或常量。

在 JavaScript 中，由于函数区分大小写，所以在调用函数时需要注意函数名称的大小写。不带参数的函数使用函数名加上括号即可调用，带参数的函数需要根据参数的个数和类型在括号中传递相应的参数进行调用。如果函数有返回值，则可以使用赋值语句将函数值赋给一个变量进行返回。

2. 事件处理

事件驱动是 JavaScript 语言的一个最为基本的特征，事件是指用户在访问页面时执行的操作。当浏览器探测到一个事件时，例如，单击鼠标或按键，它可以触发与这个事件相关联的事件处理程序。事件处理的过程通常分为三步：发生事件、启动事件处理程序和事件处理程序做出反应。

值得一提的是，在上述的事件处理过程中，要使事件处理程序启动，就需要调用事件处理程序，事件处理程序可以是任意的 JavaScript 语句，但通常使用特定的自定义函数(Function)对事件进行处理。下面通过案例演示如何在 HTML 中调用事件处理程序。

在 chapter07 文件夹中创建一个名称 jsDemo03 的 HTML 文件，编辑后的关键代码如文件 7-13 所示。

文件 7-13　jsDemo03.html

```
1  <body>
2      <input type="button" name="btn" value="点这儿" onclick="alert('Hello');"/>
3  </body>
```

在文件 7-13 中，onclick 代表当前事件类型为鼠标单击事件；alert()函数主要用于弹出

警示对话框，起到对用户进行提示的作用。

使用浏览器打开文件 7-13，显示结果如图 7-20 所示。

单击图中的按钮后，将弹出如图 7-21 所示的警示框。

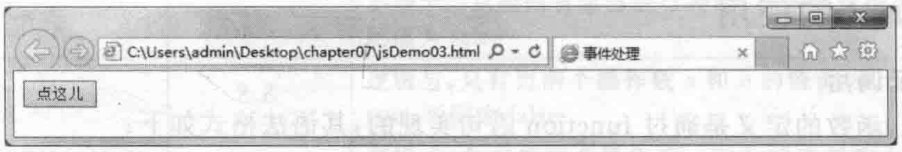


图 7-20 调用事件处理程序



图 7-21 弹出警示框

除了 onclick 事件以外，在 JavaScript 中还有很多常用的事件类型，如表 7-5 所示。

表 7-5 JavaScript 的常用事件

类 别	事 件	事件说明
表单事件	onblur	当前元素失去焦点时触发此事件
	onchange	当前元素失去焦点并且元素内容发生改变时触发此事件
	onfocus	当某个元素获得焦点时触发此事件
	onreset	当表单被重置时触发此事件
	onsubmit	当表单被提交时触发此事件
页面事件	onload	当页面加载完成时触发此事件

3. 常用对象

(1) Window 对象

Window 对象表示整个浏览器窗口，它处于对象层次的顶端，可用于获取浏览器窗口的大小和位置，或设置定时器等。在使用时，JavaScript 允许省略 Window 对象的名称。其常用的属性和方法如表 7-6 所示。

表 7-6 Window 对象的常用属性和方法

属性/方法	说 明
document、history、location、navigator、screen	返回相应对象的引用。例如，document 属性返回 document 对象的引用
parent、self、top	分别返回父窗口、当前窗口和最顶层窗口的对象引用
innerWidth、innerHeight	分别返回窗口文档显示区域的宽度和高度
outerWidth、outerHeight	分别返回窗口的外部宽度和高度
open()、close()	打开或关闭浏览器窗口
alert()、confirm()、prompt()	分别表示弹出警告框、确认框、用户输入框
setTimeout()、clearTimeout()	设置或清除普通定时器

(2) Date 对象

Date 对象是一个有关日期和时间的对象，它具有动态性，必须使用 new 关键字创建一个实例，语法如下所示。

```
var Mydate=new Date();
```

Date 对象没有提供直接访问的属性，只有获取和设置日期的方法，如表 7-7 所示。

表 7-7　Date 对象的常用方法

获取方法	说　明	设置方法	说　明
getFullYear()	返回 4 位数年份	setFullYear()	设置年用 4 位数的年份
getMonth()	返回月份值(0～11)	setMonth()	设置月份值(0～11)
getDate()	返回日期值(1～31)	setDate()	设置日期值(1～31)
getDay()	返回星期值(0～6)	setDay()	设置星期值(0～6)
getHours()	返回小时值(0～23)	setHours()	设置小时值(0～23)
getMinutes()	返回分钟值(0～59)	setMinutes()	设置分钟值(0～59)
getSeconds()	返回秒数值(0～59)	setSeconds()	设置秒数值(0～59)
getTime()	返回从 1970 年 1 月 1 日至今的毫秒数	setTime()	使用毫秒形式设置 Date 对象

(3) String 对象

String 对象是 JavaScript 提供的字符串处理对象，创建对象实例后才能引用，它提供了对字符串进行处理的属性和方法，具体如表 7-8 所示。

表 7-8　String 对象的常用属性和方法

类型	名　称	说　明
属性	length	返回字符串中字符的个数。注：一个汉字也是一个字符
方法	indexOf(str[,startIndex])	从前向后检索字符串
	lastIndexOf(search[,startIndex])	从后向前搜索字符串
	substr(startIndex[, length])	返回从起始索引号提取字符串中指定数目的字符
	substring(startIndex [,endIndex])	返回字符串中两个指定的索引号之间的字符
	split(separator [,limitInteger])	把字符串分割为字符串数组
	search(substr)	检索字符串中指定的子字符串或与正则表达式相匹配的值
	replace(substr,replacement)	替换与正则表达式匹配的子串
	toLowerCase()	把字符串转换为小写
	toUpperCase()	把字符串转换为大写
	localeCompare()	用本地特定的顺序比较两个字符串

第 8 章

Java Web概述

通过前面章节的学习，读者已经掌握了 Java 语言的基础知识，然而要想完成实际项目的开发，这些基础知识是明显不够用的，还需要掌握大量的 Web 以及框架方面的知识。从本章开始，将针对 Java 中的 Web 相关知识进行详细讲解。

8.1 Web 开发的相关知识

8.1.1 C/S 与 B/S 体系结构

随着网络技术的不断发展，单机软件已经难以满足网络计算的需要。因此，各种网络程序开发体系结构应运而生。其中，应用最多的网络应用程序开发体系结构可以分为两种：一种是基于客户端/服务器的 C/S(Client/Server 的简写)结构，另一种是基于浏览器/服务器的 B/S(Browser/Server 的简写)结构。

1. C/S 结构

C/S 结构是早期出现的一种分布式架构。在这种结构中，客户端需要安装专门的客户端软件，而服务器通常会采用高性能的 PC 或工作站，并采用大型数据库系统(如 Oracle、SQL Server 等)，如图 8-1 所示。

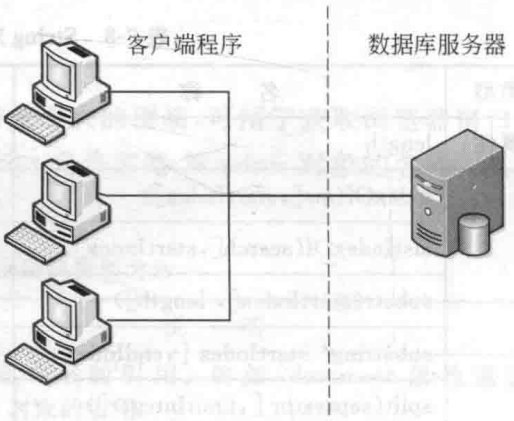


图 8-1 C/S 体系结构

在 C/S 结构中，客户端程序与数据库会直接建立连接，并且客户端程序需要利用客户机的数据处理能力，完成应用程序中绝大多数的业务逻辑和界面展示，这种方式充分利用了两端硬件的环境优势，将任务合理地分配到客户端和服务器，从而降低系统的通信开销。

2. B/S 结构

B/S 结构是对 C/S 结构的一种改进，是 Web 兴起后的一种网络结构模式。在这种结构中，客户端不需要开发任何软件，而是统一使用浏览器向 Web 服务器发送请求，由服务器进行处理并返回给客户端，如图 8-2 所示。

在图 8-2 所示的 B/S 体系结构中，浏览器提交的请求会传送给 Web 服务器，服务器在接收到这个请求后会进行相应的处理(如访问数据库服务器)，然后将处理后的结果返回给浏览器并显示。

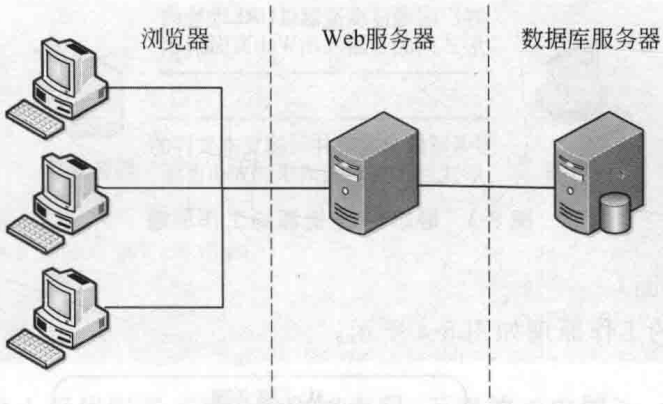


图 8-2　B/S 体系结构

3. C/S 结构与 B/S 结构的区别

C/S 结构与 B/S 结构的主要区别如下。

- 开发成本方面：C/S 结构对于不同客户端需要开发不同的程序；B/S 结构中的客户端不需要开发任何软件，只需要通过浏览器联网就可以使用。
- 系统维护方面：C/S 结构的软件安装、调试以及升级都需要在不同的客户机上进行，必须整体考虑，有时候为了升级可能需要重新实现一个新系统；在 B/S 结构中，由于程序的业务逻辑处理都集中在 Web 服务器，系统的维护与升级都可以在服务器端进行，所以大大节省了系统维护成本。
- 用户接口方面：C/S 结构由于客户端软件的原因，界面表现方法有限；B/S 结构使用浏览器作为展示界面，有更加丰富的表现方式。
- 硬件环境方面：C/S 结构大多建立在局域网的基础上；B/S 结构建立在广域网的基础上。
- 系统跨平台性：C/S 结构的软件与操作系统相关，兼容性差；B/S 结构的系统借助于 Java 这样的跨平台语言，可以兼容于所有主流数据库和操作系统。

由于 B/S 结构相比于 C/S 结构有诸多优点，因此，B/S 结构是目前各类网站和系统的首选体系结构，Web 服务器采用的就是 B/S 结构。

8.1.2　Web 资源

Internet 上供外界访问的文件或程序被称为 Web 资源。根据呈现效果的不同，Web 资源可分为动态 Web 资源和静态 Web 资源。在互联网发展的初期，网络上的页面都是由一些 HTML 语言编写的，当浏览器在不同时刻或者不同条件下访问时，所获得的页面内容不会发生变化，因此将这些页面称为静态 Web 资源(或页面)。

静态 Web 资源的工作原理如图 8-3 所示。

从图 8-3 中可以看出，当用户通过浏览器以 URL 地址的形式向服务器发出访问 Web 页面的请求后，Web 服务器会处理接收到的用户请求，然后将处理结果回传给客户端浏览器并显示。

随着网络的发展，静态 Web 技术已经不能满足用户的需求。用户希望根据自己的请求，让服务器返回不同的内容，例如在 12306 网站查看某次列车的剩余车票时，浏览器在不同时刻所访问的页面内容会随着车票的剩余情况而变化。这种由程序动态生成的页面称为

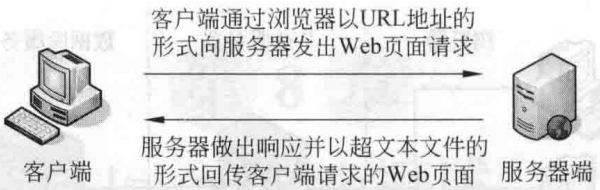


图 8-3 静态 Web 资源的工作原理

动态 Web 资源(或页面)。

动态 Web 资源的工作原理如图 8-4 所示。

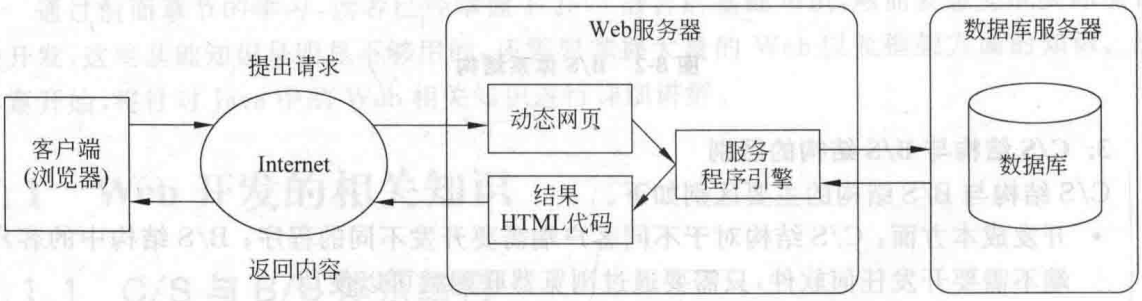


图 8-4 动态 Web 资源的工作原理

从图 8-4 中可以看出,当客户端通过浏览器向 Web 服务器发出访问动态网页的请求时,Web 服务器不会将程序的源代码直接返回客户端,而是根据用户的请求将页面交给了服务程序引擎,服务程序引擎会扫描整个页面,并根据页面中的脚本代码实现不同的功能,如访问数据库、执行运算等,之后会将执行结果交给 Web 服务器,由 Web 服务器将结果和页面上的 HTML 内容一同传送给客户端并显示。

在基于动态 Web 资源的网站上,客户端浏览器看到的内容与传统的静态资源页面没有任何形式上的区别,但是动态的资源页面中所生成的内容是动态变化的,每次返回的结果都可能不同。

8.2 XML 基础

为了简化 Internet 的文档信息传输,W3C 组织推出了一种新的数据交换标准——XML,它是一种通用的数据交换格式,可以使数据在各种应用程序之间轻松实现数据的交换。本节将对 XML 进行详细讲解。

8.2.1 XML 概述

XML 是 Extensible Markup Language 的缩写,它是一种类似于 HTML 的标记语言,称为可扩展标记语言,可扩展指的是用户可以按照 XML 的规则自定义标记。

下面通过一个 XML 文档讲解 XML 的结构,如文件 8-1 所示。

文件 8-1 city. xml

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

```
3       <内蒙古>
4           <城市>呼和浩特</城市>
5           <城市>阿尔山</城市>
6       </内蒙古>
7       <黑龙江>
8           <城市>哈尔滨</城市>
9           <城市>齐齐哈尔</城市>
10      </黑龙江>
11  </中国>
```

在文件 8-1 中，第 1 行代码是 XML 的文档声明，下面的＜中国＞、＜河北＞、＜城市＞都是用户自己创建的标记，它们都可以称为元素，这些元素必须成对出现，即包括开始标记和结束标记。例如，在＜中国＞元素中的开始标记为＜中国＞，结束标记为＜/中国＞。＜中国＞被视为整个 XML 文档的根元素，其下面有两个子元素，分别是＜河北＞和＜山西＞，而在这两个子元素中又分别包含另外两个子元素(＜城市＞)。

在 XML 文档中，通过元素的嵌套关系可以很准确地描述具有树状层次结构的复杂信息，因此，越来越多的应用程序都采用 XML 格式存放相关的配置信息，以便于读取和修改。

XML 和 HTML 都是标记文本，虽然它们在结构上大致相同，都是以标记的形式描述信息，但它们有着本质的区别，具体如下。

(1) XML 用于传输和存储数据，而 HTML 用于显示数据。

(2) XML 严格区分大小写，而 HTML 不区分大小写。

(3) 格式良好的 XML 有且只有一个根元素，而 HTML 可以有多个根元素。

(4) XML 中的空格不会自动删除，HTML 中的空格是自动过滤的。

(5) XML 中的标记可以根据需要自行定义，并且可扩展，而 HTML 中的标记是预定义好的。

需要注意的是，XML 既不是 HTML 的升级，也不是 HTML 的替代产品，虽然两者有些相似，但它们的应用领域完全不同。

8.2.2　XML 语法

1. 文档声明

在一个完整的 XML 文档中，必须包含一个 XML 文档的声明，并且该声明必须位于文档的第一行。该声明表示该文档是一个 XML 文档，以及该文档遵循哪个 XML 版本的规范。XML 文档声明的语法格式如下：

```
<? xml 版本信息[编码信息][文档独立性信息]? >
```

从上面的语法格式中可以看出，文档声明以符号“＜?”开头，以符号“?＞”结束，中间可以声明版本信息、编码信息以及文档独立性信息。

需要注意的是，在＜和?、? 和＞以及第一个? 和 xml 之间不能有空格；另外，中括号([])括起来的部分表示其是可选的。

接下来，针对语法格式中的版本信息、编码信息以及文档独立性信息编写一段示例代

码，具体如下：

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

在上述示例代码中，version 属性表示 XML 的版本。目前，最常用的 XML 版本是 1.0。encoding 属性用于说明 XML 文档所使用的编码方式，在默认情况下，XML 文档使用 UTF-8 编码方式。standalone 属性用于声明该文档是否是独立的文档，在默认情况下，standalone 属性的值为 no，表示该文档依赖于外部文档。

2. 元素定义

在 XML 文档中，主体内容都是由元素(Element)组成的。元素一般由开始标记、属性、元素内容和结束标记构成，具体示例如下：

```
<城市>北京</城市>
```

在上面的示例中，＜城市＞和＜/城市＞就是 XML 文档中的标记，标记的名称也就是元素的名称。在一个元素中可以嵌套若干子元素。如果一个元素没有嵌套在其他元素内，则这个元素称为根元素。根元素是 XML 文档定义的第一个元素。如果一个元素中既没有嵌套子元素，也没有包含文本内容，则这样的元素称为空元素，空元素可以不使用结束标记，但必须在起始标记的＞前增加一个正斜杠/以说明该元素是个空元素，例如，＜img＞＜/img＞可以简写成＜img/＞。

3. 属性定义

在 XML 文档中，可以为元素定义属性，属性是对元素的进一步描述和说明。在一个元素中可以有多个属性，并且每个属性都有自己的名称和取值，具体示例如下：

```
<售价 单位="元">68</售价>
```

在上面的示例中，＜售价＞元素中定义了一个属性“单位”。需要注意的是，在 XML 文档中属性的命名规范和元素相同，属性值必须要用双引号("")或者单引号(')引起来，否则会被视为错误。

4. 注释

如果要在 XML 文档中插入一些附加信息，如作者姓名、地址或电话等，或者需要暂时屏蔽某些 XML 语句，这时可以通过注释的方式实现，被注释的内容会被程序忽略而不被解析和处理。XML 注释和 HTML 注释的写法基本一致，具体语法格式如下：

```
<!--注释信息-->
```

8.2.3 DTD 约束

在 XML 文档中，定义了一套规则对文档中的内容进行约束，这套约束被称为 XML 约束。XML 约束根据语法规则的不同，形成了不同的 XML 约束语言。目前，最常用的两种 XML 约束语言是 DTD 和 Schema。

1. DTD 约束

DTD 约束是早期出现的一种 XML 约束模式语言，根据其语法创建的文件称为 DTD 文件。在一个 DTD 文件中，可以包含元素的定义、元素之间关系的定义、元素属性的定义以及实体和符号的定义。

下面通过简单的案例认识一下 DTD 约束，如文件 8-2 和文件 8-3 所示。

文件 8-2　book. xml

```
1  <? xml version="1.0" encoding="UTF-8"? >
2  <书架>
3      <书>
4          <书名>Java 基础案例教程</书名>
5          <作者>黑马程序员</作者>
6          <书号>978-7-115-43937-6</书号>
7      </书>
8      <书>
9          <书名>Java Web 程序设计任务教程</书名>
10         <作者>黑马程序员</作者>
11         <书号>978-7-115-43936-9</书号>
12     </书>
13 </书架>
```

文件 8-3　book. dtd

```
1  <? xml version="1.0" encoding="UTF-8"? >
2  <!ELEMENT 书架 (书+)>
3  <!ELEMENT 书 (书名,作者,书号)>
4  <!ELEMENT 书名 (#PCDATA)>
5  <!ELEMENT 作者 (#PCDATA)>
6  <!ELEMENT 书号 (#PCDATA)>
```

文件 8-3 所示的 book. dtd 是一个简单的 DTD 约束文档。在文件 8-2 中，book. xml 中定义的每个元素都是按照 book. dtd 文档所规定的约束进行编写的。下面对文件 8-3 所示的约束文档进行详细讲解，具体如下。

- 在第 2 行中，使用<! ELEMENT...>语句定义了一个元素，其中“书架”是元素的名称，“(书+)”表示书架元素中有一个或者多个书元素，字符+用于表示它所修饰的成分必须出现一次或者多次。
- 在第 3 行中，“书”是元素名称，“(书名，作者，书号)”表示元素书包含书名、作者、书号这三个子元素，并且这些子元素要按照顺序依次出现。
- 在第 4～6 行中，“书名”“作者”和“书号”都是元素名称，“(#PCDATA)”表示元素中嵌套的内容是普通的文本字符串。

2. DTD 的引入

对 DTD 文件有了大致了解后，如果想使用 DTD 文件约束 XML 文档，必须在 XML 文

档中引入 DTD 文件。在 XML 文档中引入 DTD 文件通常采用两种方式，具体如下：

```
<!DOCTYPE 根元素名称 SYSTEM "外部 DTD 文件的 URI">
<!DOCTYPE 根元素名称 PUBLIC "DTD 名称" "外部 DTD 文件的 URI">
```

在上述两种引入 DTD 文件的方式中，第一种方式用于引用本地的 DTD 文件，第二种方式用于引用公共的 DTD 文件，其中“外部 DTD 文件的 URI(统一资源标识符)”指的是 DTD 文件的存放位置，对于第一种方式，它既可以是相对于 XML 文档的相对路径，也可以是一个绝对路径；而对于第二种方式，它是 Internet 上的一个绝对 URL 地址。

接下来对文件 8-2 进行修改，在 XML 文档中引入本地的 DTD 文件 book. dtd，如文件 8-4 所示。

文件 8-4　book. xml

```
<? xml version="1.0" encoding="UTF-8"? >
<!DOCTYPE 书架 SYSTEM "book.dtd">
<书架>
    <书>
        <书名>Java 基础案例教程</书名>
        <作者>黑马程序员</作者>
        <书号>978-7-115-43937-6</书号>
    </书>
    <书>
        <书名>Java Web 程序设计任务教程</书名>
        <作者>黑马程序员</作者>
        <书号>978-7-115-43936-9</书号>
    </书>
</书架>
```

在文件 8-4 中，由于引入的是本地的 DTD 文件，因此，使用的是 SYSTEM 属性的 DOCTYPE 声明语句。另外，在 XML 文档的声明语句中，standalone 属性不能设置为 yes。

如果希望引入一个公共的 DTD 文件，则需要在 DOCTYPE 声明语句中使用 PUBLIC 属性，具体示例如下：

```
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
```

其中，"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"是 DTD 名称，它用于说明 DTD 符合的标准、所有者的名称以及对 DTD 描述的文件进行说明，虽然 DTD 名称看上去比较复杂，但这完全是 DTD 文件发布者需要考虑的事情，XML 文件的编写者只要将 DTD 文件发布者事先定义好的 DTD 标识名称进行复制并使用即可。

8.2.4　Schema 约束

1. Schema 约束

XML Schema 也是一种用于定义和描述 XML 文档结构与内容的模式语言，它的出现克服了 DTD 的局限性。下面通过对 XML Schema 与 DTD 进行比较，给出 XML Schema 所具有的一些显著优点。

(1) DTD 采用非 XML 语法格式，缺乏对文档结构、元素、数据类型等的全面描述。而 XML Schema 采用 XML 语法格式，而且它本身也是一种 XML 文档，因此，XML Schema 的语法格式比 DTD 更容易理解。

(2) XML 具有非常严格的合法性要求，虽然 DTD 和 XML Schema 都用于对 XML 文档进行描述，且都被作为验证 XML 合法性的基础，但是 DTD 自身合法性的验证必须采用另外一套机制，而 XML Schema 则采用与 XML 文档相同的合法性验证机制。

(3) XML Schema 对名称空间的支持度非常高，而 DTD 几乎不支持名称空间。

(4) DTD 支持的数据类型非常有限。例如，DTD 可以指定元素中必须包含字符文本(PCDATA)，但无法指定元素中必须包含非负整数(nonNegativeInteger)，而 XML Schema 比 XML DTD 支持更多的数据类型，包括用户自定义的数据类型。

(5) DTD 定义约束的能力非常有限，无法对 XML 实例文档做出更细致的语义限制，例如，其无法很好地指定一个元素中的某个子元素必须出现 7～12 次；而 XML Schema 定义约束的能力非常强大，可以对 XML 实例文档做出细致的语义限制。

通过上面的比较可以发现，XML Schema 的功能比 DTD 强大很多，但相应的语法也比 DTD 复杂很多，一个简单的 Schema 文档如文件 8-5 所示。

文件 8-5　Simple. xsd

```
1  <? xml version="1.0"? >
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3      <xs:element name="root" type="xs:string"/>
4  </xs:schema>
```

在文件 8-5 中，第 1 行是文档声明，第 2 行中以 xs:schema 作为根元素，表示模式定义的开始。由于根元素 xs:schema 的属性都在 http://www. w3. org/2001/XMLSchema 名称空间中，因此，在根元素上必须声明该名称空间。

2. 名称空间

一个 XML 文档可以引入多个约束文档，但是，由于约束文档中的元素或属性都是自定义的，极有可能出现代表不同含义的同名元素或属性，从而导致名称发生冲突。因此，XML 文档提供了名称空间，使用名称空间唯一标识一个元素或者属性。

要使用名称空间，就必须先声明名称空间。名称空间的声明就是在 XML 文档中为某个模式文档的名称空间指定一个临时的简写名称，它通过一系列的保留属性进行声明，这种属性的名称必须以“xmlns”或者“xmlns:”作为前缀。其声明的语法格式如下：

```
<元素名 xmlns:prefixname="URI">
```

在上述语法格式中，元素名指在哪一个元素上声明名称空间，在这个元素上声明的名称空间适用于声明它的元素和属性，以及该元素中嵌套的所有元素及其属性。xmlns:prefixname 指的是该元素的属性名，它所对应的值是一个 URI 引用，用于标识该名称空间的名称。

需要注意的是，如果有两个 URI 并且其组成的字符完全相同，就可以认为它们标识的是同一个名称空间。

下面通过案例学习如何声明名称空间，如文件 8-6 所示。

文件 8-6 book.xml

```
1  <? xml version="1.0" encoding="UTF-8"? >
2  <itheima:书架 xmlns:itheima="http://www.itheima.com/xmlbook/schema">
3      <itheima:书>
4          <itheima:书名>Java 邮件开发详解</itheima:书名>
5          <itheima:作者>张孝祥</itheima:作者>
6          <itheima:售价>39.00 元</itheima:售价>
7      </itheima:书>
8  </itheima:书架>
```

在文件 8-6 中，itheima 作为多个元素名称的前缀部分，必须通过名称空间声明将它关联到唯一标识某个名称空间的 URI 上，xmlns:itheima="http://www.itheima.com/xmlbook/schema"语句就是将前缀名 itheima 关联到名称空间"http://www.itheima.com/xmlbook/schema"上。由此可见，名称空间的应用就是将一个前缀(如 itheima)绑定到代表某个名称空间的 URI(如 http://www.itheima.com/xmlbook/schema)上，然后将前缀添加到元素名称的前面(例如，itheima:书)从而说明该元素属于哪个模式文档。

需要注意的是，在声明名称空间时，有两个前缀是不允许使用的，分别是 xml 和 xmlns。xml 前缀被定义为与名称空间名字 http://www.w3.org/XML/1998/namespace 绑定，只能用于 XML 1.0 规范中定义的 xml:space 和 xml:lang 属性。前缀 xmlns 仅用于声明名称空间的绑定，它被定义为与名称空间名字 http://www.w3.org/2000/xmlns 绑定。

3. 引入 Schema 文档

如果想通过 XML Schema 文件对某个 XML 文档进行约束，必须将 XML 文档与 Schema 文件进行关联。在 XML 文档中引入 Schema 文件有两种方式。

(1) 使用名称空间引入 XML Schema 文档

在使用名称空间引入 XML Schema 文档时，需要通过属性 xsi:schemaLocation 声明名称空间的文档，xsi:schemaLocation 属性是在标准名称空间“http://www.w3.org/2001/XMLSchema-instance”中定义的，该属性中包含两个 URI，这两个 URI 之间用空白符分隔。其中，第一个 URI 是名称空间的名称，第二个 URI 是文档的位置。下面通过案例演示如何使用名称空间引入 XML Schema 文档，如文件 8-7 所示。

文件 8-7　namespacebook. xml

```
<? xml version="1.0" encoding="UTF-8"? >
<书架 xmlns="http://www.itheima.com/xmlbook/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="http://www.itheima.com/xmlbook/schema
                         http://www.itheima.com/xmlbook.xsd">
    <书>
        <书名>Java 基础案例教程</书名>
        <作者>黑马程序员</作者>
        <书号>978-7-115-43937-6</书号>
    </书>
</书架>
```

在文件 8-7 中，schemaLocation 属性用于指定名称空间所对应的 XML Schema 文档的位置，由于 schemaLocation 属性是在另外一个公认的标准名称空间中定义的，因此，在使用 schemaLocation 属性时，必须要声明该属性所属的命名空间。

需要注意的是，一个 XML 实例文档可能引用多个名称空间，这时，可以在 schemaLocation 属性值中包含多对名称空间与它们所对应的 XML Schema 文档的存储位置，每一对名称空间的设置信息之间采用空格分隔，具体示例如下：

```
<? xml version="1.0" encoding="UTF-8"? >
<书架 xmlns="http://www.itheima.com/xmlbook/schema"
    xmlns:demo="http://www.itheima.com/demo/schema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.itheima.com/xmlbook/schema
                        http://www.itheima.com/xmlbook.xsd
                        http://www.itheima.com/demo/schema
                        http://www.itheima.com/demo.xsd">
    ...
</书架>
```

(2) 不使用名称空间引入 XML Schema 文档

在 XML 文档中不仅可以通过 xsi:schemaLocation 属性引入名称空间的文档，还可以通过 xsi:noNamespaceSchemaLocation 属性直接指定 XMLSchema 文档。noNamespaceSchemaLocation 属性也是在标准名称空间 http://www.w3.org/2001/XMLSchema-instance 中定义的，它用于定义指定文档的位置。其使用方式如文件 8-8 所示。

文件 8-8　nonamespacebook. xml

```
<? xml version="1.0" encoding="UTF-8"? >
<书架 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="xmlbook.xsd">
    <书>
```

```
        <书名>Java 基础案例教程</书名>
        <作者>黑马程序员</作者>
        <书号>978-7-115-43937-6</书号>
    </书>
</书架>
```

在文件 8-8 中，文档 xmlbook.xsd 与引用它的实例文档位于同一目录中。

8.3 HTTP

8.3.1 HTTP 概述

1. HTTP

HTTP 是 Hyper Text Transfer Protocol 的缩写，即超文本传输协议，它是一种请求/响应式的协议，客户端在与服务器端建立连接后，就可以向服务器端发送 URL 请求，这种请求被称为 HTTP 请求，服务器端在接收到请求后会做出响应，称为 HTTP 响应。

客户端与服务器端在 HTTP 下的交互过程如图 8-5 所示。

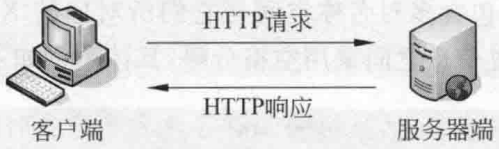


图 8-5 客户端与服务器端的交互过程

HTTP 协议具有如下特点。

- 支持客户端(浏览器就是一种 Web 客户端)/服务器端模式。
- 简单快速：客户端向服务器端请求服务时，只需要传送请求方式和路径。常用的请求方式有 GET 和 POST 等，每种方式规定了客户端与服务器端联系的类型不同。由于 HTTP 比较简单，使得 HTTP 服务器的程序规模较小，因而通信速度很快。
- 灵活：HTTP 允许传输任意类型的数据，正在传输的数据类型由 Content-Type 加以标记。
- 无状态：HTTP 是无状态协议。无状态是指协议对于事务处理没有记忆能力，如果后续处理需要前面的信息，则 HTTP 必须重传，这样可能导致每次连接传送的数据量增大。

2. URL 地址

在 Internet 上的 Web 服务器中，每一个网页文件都会有一个用于唯一标识其位置的访问标记符，这个访问标记符被称为 URL(Uniform Resource Locator，统一资源定位符)。

在一个 URL 中，通常会包含 Web 服务器的主机名、端口号、资源名以及所使用的网络协议，其示例如下：

```
http://www.itheima.com:80/index.html
```

在上面的 URL 中，http 表示传输数据所使用的协议，www. itheima. com 表示要请求的服务器主机名，80 表示要请求的端口号，index. html 表示要请求的资源名称。

3. HTTP 1.1

HTTP 自诞生以来，先后经历了很多版本，目前网络上所使用的主要版本是 HTTP 1.1。HTTP 1.1 版本支持持久连接，也就是说，在一个 TCP 连接上可以传送多个 HTTP 请求和响应，从而减少了建立和关闭连接的消耗和延时。

基于 HTTP 1.1 的客户端和服务器端的交互过程如图 8-6 所示。

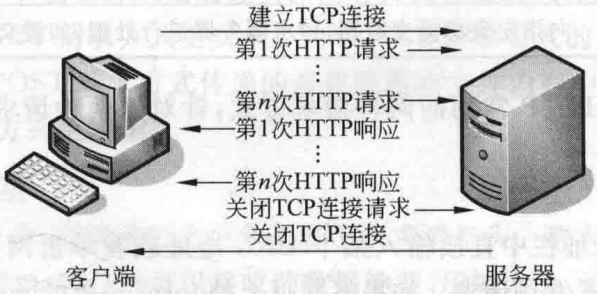


图 8-6　HTTP 1.1 协议通信原理

在图 8-6 中，当客户端与服务器端建立连接后，客户端可以向服务器端发送多个请求，并且在发送下一个请求时，无须等待上次请求的返回结果。服务器端必须按照接收客户端请求的先后顺序依次返回响应结果，以保证客户端能够区分每次请求的响应内容。

4. HTTP 消息

当用户在浏览器中访问某个 URL 地址、单击网页的某个超链接或者提交网页上的 form 表单时，浏览器都会向服务器发送请求数据，即 HTTP 请求消息。服务器接收到请求数据后，会将处理后的数据回送给客户端，即 HTTP 响应消息。HTTP 请求消息和 HTTP 响应消息统称为 HTTP 消息。

8.3.2　HTTP 请求消息

在 HTTP 中，一个完整的请求消息由请求行、请求头和实体内容三部分组成，其中，每部分都有各自不同的作用。下面主要围绕开发时需要熟悉的请求行和请求头进行详细讲解。

1. HTTP 请求行

HTTP 请求行位于请求消息的第一行，它包括三个部分，分别是请求方式、资源路径以及所使用的 HTTP 版本，具体示例如下：

```
GET /index.html HTTP/1.1
```

上面的示例就是一个 HTTP 请求行，其中，GET 是请求方式，index. html 是请求资源路径，HTTP/1.1 是通信使用的协议版本。需要注意的是，请求行中的每个部分需要用空格分隔，最后要以回车换行结束。

关于请求资源和协议版本，读者比较容易理解，而 HTTP 请求方式对读者来说会比较

陌生，下面对 HTTP 的请求方式进行简单介绍。

在 HTTP 的请求消息中，请求方式有 GET、POST、HEAD、OPTIONS、DELETE、TRACE、PUT 和 CONNECT 共八种，每种方式都指明了操作服务器中指定 URI 资源的方式。然而在实际开发中，使用最多的请求方式是 GET 和 POST，其说明如表 8-1 所示。

表 8-1 HTTP 常用的两种请求方式

请求方式	含 义
GET	请求获取请求行的 URI 所标识的资源
POST	向指定资源提交数据，请求服务器进行处理(如提交表单或者上传文件)

表 8-1 中列举了 HTTP 常用的两种请求方式，针对这两种请求方式的详细讲解如下所示。

(1) GET 方式

当用户在浏览器地址栏中直接输入某个 URL 地址或者单击网页上的某个超链接时，浏览器将使用 GET 方式发送请求。如果将网页上的 form 表单的 method 属性设置为 GET 或者不设置 method 属性(默认值是 GET)，那么当用户提交表单时，浏览器也将使用 GET 方式发送请求。

如果浏览器请求的 URL 中有参数部分，那么在浏览器生成的请求消息中，参数部分将附加在请求行中的资源路径后面。先来看一个 URL 地址，具体如下：

```
http://www.itheima.com/javaForum? name=zhangsan&age=23
```

在上述 URL 中，? 后面的内容为参数信息。参数是由参数名和参数值组成的，并且在中间使用等号(=)进行连接。需要注意的是，如果 URL 地址中有多个参数，参数之间需要用 & 分隔。

当浏览器向服务器发送请求消息时，上述 URL 中的参数部分会附加在需要访问的 URI 资源后面，如下所示：

```
GET /javaForum? name=itcast&age=23 HTTP/1.1
```

需要注意的是，使用 GET 方式传送的数据量有限，最多不能超过 1KB。

(2) POST 方式

如果将网页上的 form 表单的 method 属性设置为 POST，那么当用户提交表单时，浏览器将使用 POST 方式提交表单内容，并把各个表单元素及数据作为 HTTP 消息的实体内容发送给服务器，而不是作为 URI 地址的参数传递。另外，在使用 POST 方式向服务器传递数据时，Content-Type 消息头会自动设置为 application/x-www-form-urlencoded，Content-Length 消息头会自动设置为实体内容的长度，如下所示：

```
POST /javaForum HTTP/1.1
Host: www.itheima.com
Content-Type: application/x-www-form-urlencoded
```

```
Content-Length: 17
name=lee&psd=hnxy
```

对于使用 POST 方式传递的请求信息，服务器端程序会采用与获取 URI 后面参数相同的方式获取表单各个字段的数据。

在实际开发中，通常都会使用 POST 方式发送请求，其原因主要有两个。第一，POST 传输的数据大小无限制。由于 GET 请求方式是通过请求参数传递数据的，因此最多可传递 1KB 的数据。而 POST 请求方式是通过实体内容传递数据的，因此传递数据的大小没有限制。第二，POST 比 GET 请求方式更安全。由于 GET 请求方式的参数信息都会在 URL 地址栏中明文显示，而 POST 请求方式传递的参数隐藏在实体内容中，用户是看不到的，因此，POST 比 GET 请求方式更安全。

2. HTTP 请求消息头

在 HTTP 请求消息中，在请求行之后便是若干请求消息头。请求消息头主要用于向服务器端传递附加消息，例如，客户端可以接收的数据类型、压缩方法、语言以及发送请求的超链接所属页面的 URL 地址等信息，具体示例如下：

```
Host: localhost:8080
Accept: image/gif, image/x-xbitmap, *
Referer: http://localhost:8080/itcast/
Accept-Language: zh-cn,zh;q=0.8,en-us;q=0.5,en;q=0.3
Accept-Encoding: gzip, deflate
Content-Type: application/x-www-form-urlencoded
User-Agent: Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1; GTB6.5; CIBA)
Connection: Keep-Alive
Cache-Control: no-cache
```

从上面的请求消息头中可以看出，每个请求消息头都是由一个头字段名称和一个值构成的，头字段名称和值之间用冒号(:)和空格分隔，在每个请求消息头之后使用一个回车换行符标志结束。

需要注意的是，头字段名称不区分大小写，但习惯上将单词的第一个字母大写。

当浏览器发送请求给服务器时，根据功能需求的不同，发送的请求消息头也不相同，下面通过一张表列举常用的请求头字段，如表 8-2 所示。

表 8-2　常用请求头字段

头 字 段	说　明
Accept	Accept 头字段用于指出客户端程序(通常是浏览器)能够处理的 MIME (Multipurpose Internet Mail Extension，多用途 Internet 邮件扩展)类型
Accept-Charset	Accept-Charset 头字段用于告知服务器端客户端所使用的字符集
Accept-Encoding	Accept-Encoding 头字段用于指定客户端能够进行解码的数据编码方式，这里的编码方式通常指某种压缩方式
Accept-Language	Accept-Language 头字段用于指定客户端期望服务器返回哪个国家语言的文档

续表

头 字 段	说 明
Authorization	当客户端访问受口令保护的网页时，Web 服务器会发送 401 响应状态码和 WWW-Authenticate 响应头，要求客户端使用 Authorization 请求头应答
Proxy-Authorization	Proxy-Authorization 头字段的作用及用法与 Authorization 头字段基本相同，只不过 Proxy-Authorization 请求头是服务器端向代理服务器发送的验证信息
Host	Host 头字段用于指定资源所在的主机名和端口号
If-Match	当客户机再次向服务器请求这个网页文件时，可以使用 If-Match 头字段附带以前缓存的实体标签内容，这个请求被视为一个条件请求
If-Modified-Since	If-Modified-Since 请求头的作用和 If-Mach 类似，只不过它的值为 GMT 格式的时间
Range	用于指定服务器只需返回文档中的部分内容及内容范围，这对较大文档的断点续传非常有用
If-Range	If-Range 头字段只能伴随 Range 头字段一起使用，其设置值可以是实体标签或 GMT 格式的时间
Max-Forward	指定当前请求可以途经的代理服务器数量，每经过一台代理服务器，此数值就减 1
Referer	Referer 头字段非常有用，常被网站管理人员用于追踪网站的访问者是如何导航进入网站的。同时，Referer 头字段还可以用于网站的防盗链
User-Agent	User-Agent 中文名为用户代理，简称 UA，它用于指定浏览器或者其他客户端程序使用的操作系统及版本、浏览器及版本、浏览器渲染引擎、浏览器语言等，以便服务器针对不同类型的浏览器而返回不同的内容

8.3.3 HTTP 响应消息

当服务器收到浏览器的请求后，会回送响应消息给客户端。一个完整的响应消息主要包括响应状态行、响应消息头和实体内容，其中，每个组成部分都代表了不同的含义。下面主要围绕开发时需要熟悉的响应状态行和响应消息头进行详细讲解。

1. HTTP 响应状态行

HTTP 响应状态行位于响应消息的第一行，它包括三个部分，分别是 HTTP 版本、一个表示成功或错误的整数代码(状态码)和对状态码进行描述的文本信息，具体示例如下：

```
HTTP/1.1 200 OK
```

上面的示例就是一个 HTTP 响应消息的状态行，其中 HTTP 1.1 是通信使用的协议版本(200 是状态码)，OK 是状态描述，说明客户端请求成功。需要注意的是，请求行中的每个部分都需要用空格分隔，最后需要以回车换行结束。

关于协议版本和文本信息，读者比较容易理解，而 HTTP 的状态码对读者来说则比较陌生。状态码由三位数字组成，表示请求是否被处理。HTTP 响应状态码的第一个数字定义了响应的类别，后面两位没有具体的分类。第一个数字有以下五种可能的取值。

- 1xx：表示请求已接收，需要继续处理。
- 2xx：表示请求已成功被服务器接收、理解并接受。
- 3xx：为完成请求，客户端需要进一步细化请求。
- 4xx：客户端的请求有错误。

• 5xx：服务器端出现错误。

HTTP 的状态码数量众多，其中大部分无须记忆，但在 Web 开发中，比较常见的几个状态码还是需要读者了解的，如表 8-3 所示。

表 8-3　常见状态码

状态码	说　明
200	表示服务器成功处理了客户端的请求。客户端的请求成功，响应消息返回正常的请求结果
302	表示请求的资源临时从不同的 URI 响应请求，但请求者应继续使用原有位置进行以后的请求。例如，在请求重定向中，临时 URI 应该是响应的 Location 头字段所指向的资源
304	如果客户端有缓存的文档，那么它会在发送的请求消息中附加一个 If-Modified-Since 请求头，表示只有请求的文档在 If-Modified-Since 指定的时间之后发生过更改，服务器才需要返回新文档。状态码 304 表示客户端缓存的版本是最新的，客户端应该继续使用它；否则，服务器将使用状态码 200 返回所请求的文档
404	表示服务器找不到请求的资源。例如，访问服务器不存在的网页经常会返回此状态码
500	表示服务器发生错误，无法处理客户端的请求。大部分情况下，是由于服务器端的 CGI、ASP、JSP 等程序发生了错误，一般服务器会在相应消息中提供具体的错误信息

2. HTTP 响应消息头

在 HTTP 响应消息中，第一行为响应状态行，随后就是若干响应消息头。服务器端通过响应消息头向客户端传递附加信息，包括服务程序名、被请求资源需要的认证方式、客户端请求资源的最后修改时间、重定向地址等信息。

HTTP 响应消息头的具体示例如下：

```
Server: Apache-Coyote/1.1
Content-Encoding: gzip
Content-Length: 80
Content-Language: zh-cn
Content-Type: text/html; charset=GB2312
Last-Modified: Mon,18 Nov 2013 18:23:51 GMT
Expires: -1
Cache-Control: no-cache
Pragma: no-cache
```

从上面的响应消息头中可以看出，它们的格式和 HTTP 请求消息头的格式相同。当服务器向客户端回送响应消息时，根据情况的不同，发送的响应消息头也不相同。常用的响应消息头字段如表 8-4 所示。

表 8-4　常用的响应消息头字段

头 字 段	说　明
Accept-Range	用于说明服务器是否接受客户端使用 Range 请求头字段请求资源
Age	用于指出当前网页文档可以在客户端或代理服务器中缓存的有效时间，设置值为一个以秒为单位的时间数

续表

头 字 段	说 明
Etag	用于向客户端传送代表实体内容特征的标记信息，这些标记信息称为实体标签，每个版本的资源的实体标签是不同的，通过实体标签可以判断在不同时间获得的同一资源路径下的实体内容是否相同
Location	用于通知客户端获取请求文档的新地址，其值为一个使用绝对路径的 URL 地址
Retry-After	可以与 503 状态码配合使用，告诉客户端在什么时间可以重新发送请求，也可以与任何一个 3xx 状态码配合使用，告诉客户端处理重定向的最小延时时间，Retry-After 头字段的值可以是 GMT 格式的时间，也可是一个以秒为单位的时间数
Server	用于指定服务器软件产品的名称
Vary	用于指定影响服务器所生成的响应内容的请求头字段名
WWW-Authenticate	当客户端访问受口令保护的网页文件时，服务器会在响应消息中回送 401 (Unauthrized)响应状态码和 WWW-Authenticate 响应头，指示客户端应该在 Authorization 请求头中使用 WWW-Authenticate 响应头指定的认证方式提供用户名和密码信息
Proxy-Authenticate	Proxy-Authenticate 头字段是针对代理服务器的用户信息验证，其他的作用与用法与 WWW-Authenticate 头字段类似
Refresh	用于告诉浏览器自动刷新页面的时间，它的值是一个以秒为单位的时间数
Content-Disposition	如果服务器希望浏览器不直接处理响应的实体内容，而是让用户选择将响应的实体内容保存到一个文件中，此时需要使用 Content-Disposition 头字段

8.4 Tomcat

一个动态 Web 资源(项目)开发完毕后，需要发布到 Web 服务器中才能被外界访问。因此，在学习 Web 开发之前，需要学会安装和使用一台 Web 服务器。下面将针对开发中常用的一种 Web 服务器 Tomcat 进行详细讲解。

8.4.1 Tomcat 简介

Tomcat 是 Apache 组织的 Jakarta 项目中的一个重要子项目，它是 Sun 公司(已被 Oracle 收购)推荐的运行 Servlet 和 JSP 的容器(引擎)，其源代码是完全公开的。Tomcat 不仅具有 Web 服务器的基本功能，还提供了数据库连接池等许多通用组件功能。

Tomcat 运行稳定、可靠、效率高，不仅可以和目前大部分主流的 Web 服务器(如 Apache 和 IIS 服务器)一起工作，还可以作为独立的 Web 服务器软件。因此，越来越多的软件公司和开发人员都使用 Tomcat 作为运行 Servlet 和 JSP 的平台。

Tomcat 的版本在不断升级，其功能也在不断完善与增强，目前其官方网站上的最新版本为 Tomcat 9.0，本书使用的 Tomcat 版本是当前企业使用较多的 7.0 版本，建议初学者下载与本书相对应的版本进行学习。

8.4.2 Tomcat 的安装和启动

本书使用的 Tomcat 的版本是 Tomcat 7.0.70，读者可以在 Tomcat 的官方网站

http://tomcat.apache.org/进行下载(建议初学者下载压缩包,通过解压的方式安装 Tomcat)。需要注意的是,在安装 Tomcat 之前需要先安装 JDK,建议使用 JDK 7.0 版本。由于篇幅所限,关于 Tomcat 7.0 的下载过程此处不再详细讲解,请读者自行查阅相关详细资料进行下载。

将已下载的压缩包解压到指定的目录即可完成 Tomcat 的安装。打开解压后的文件夹,可以看到 Tomcat 安装目录中包含了一系列的子目录,如 bin、conf、lib、webapps 等,在 bin 子目录下存放了许多脚本文件,其中,startup.bat 就是启动 Tomcat 的脚本文件。双击 startup.bat 文件便会启动 Tomcat 服务器,此时,可以在启动窗口的命令行中看到一些启动信息,如图 8-7 所示。

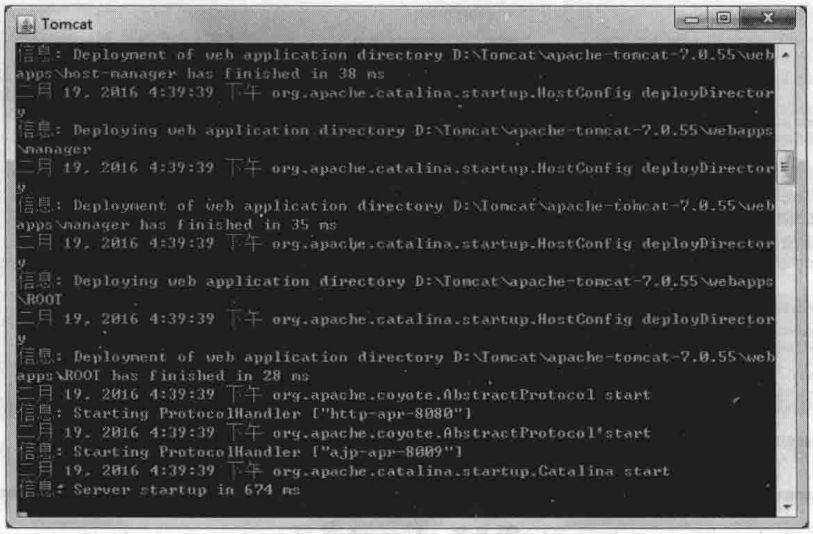


图 8-7　Tomcat 启动信息

Tomcat 服务器启动后,在浏览器的地址栏中输入 http://localhost:8080 或者 http://127.0.0.1:8080 即可访问 Tomcat 服务器。如果浏览器中的显示界面如图 8-8 所示,则说明 Tomcat 服务器已经安装成功。

8.4.3　Tomcat 中的 Web 应用

在 Tomcat 中运行的 Web 资源都是以 Web 应用形式呈现的。Web 应用(也称为 Web 应用程序或 Web 工程)指多个 Web 资源按照一定的结构组成的集合。一个 Web 应用由多个 Web 资源或其他文件组成,其中包括 html 文件、css 文件、js 文件、动态 Web 页面、Java 程序、支持 jar 包、配置文件等。开发人员在开发 Web 应用时,应按照一定的目录结构存放这些文件,否则 Web 应用在 Tomcat 服务器中可能会出现启动报错或启动后无法访问等问题。

Tomcat 中的 Web 应用的目录结构如图 8-9 所示。

从图 8-9 中可以看出,一个 Web 应用需要包含多个目录,这些目录用于存储不同类型的文件。其中,所有的 Web 资源都可以直接存放在 Web 应用的根目录(图中的 Web 应用目录)下,在 Web 应用的根目录中还有一个特殊的子目录 WEB-INF,所有的配置文件都存放在这个目录中。此外,WEB-INF 还有 classes 和 lib 两个子目录,其中 classes 目录用于存

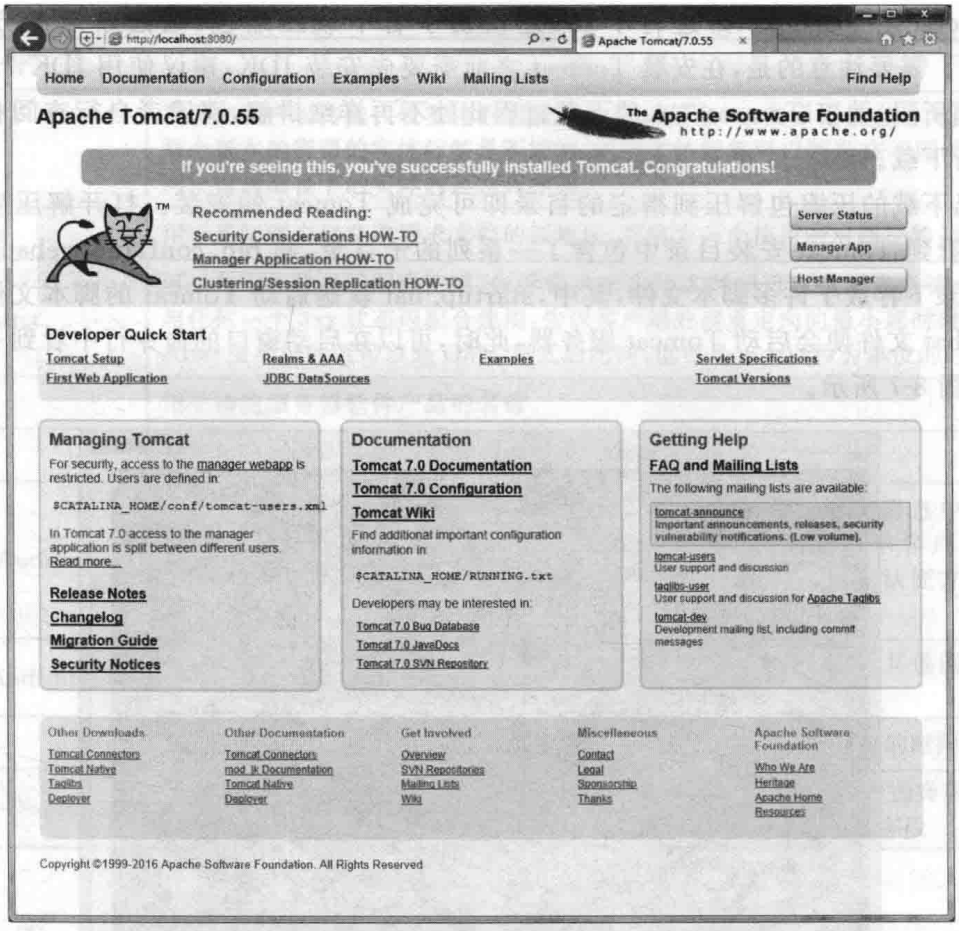


图 8-8　Tomcat 首页

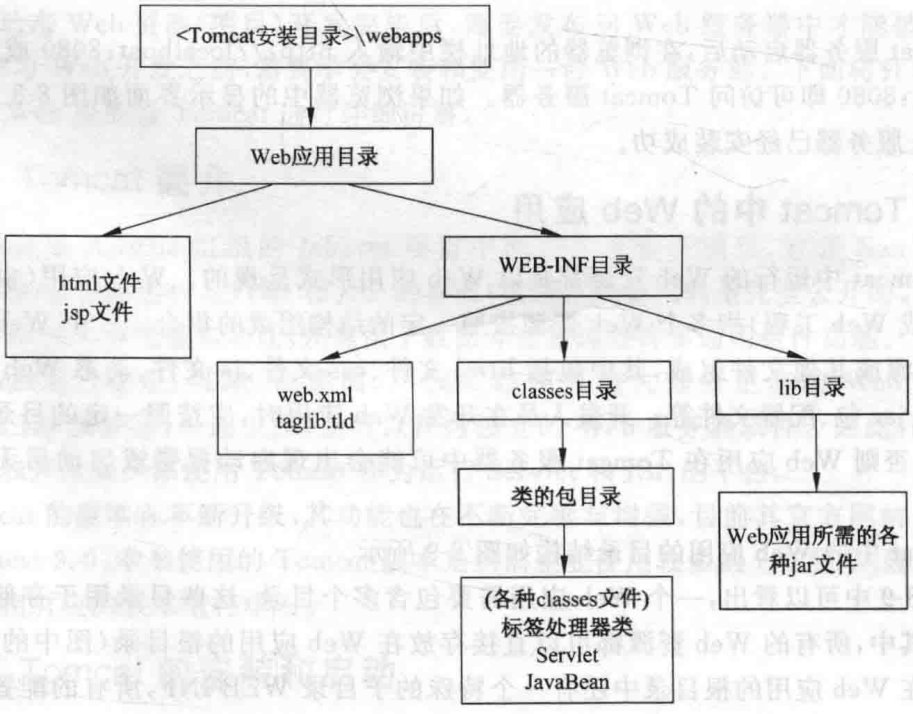


图 8-9　Web 应用目录

放各种 class 文件，lib 目录用于存放 Web 应用所需要的各种 jar 文件。

当访问一个 Web 应用程序时，通常需要指定访问的资源名称，如果没有指定资源名称，则会访问默认的页面。例如在访问 w3school 网站首页面时需要输入 http://www.w3school.com.cn/index.html，有时用户也希望只输入 http://www.w3school.com.cn 就能访问 w3school 的首页面。要想实现这样的需求，只需要修改 WEB-INF 目录下的 web.xml 文件即可。

为了帮助初学者更好地理解默认页面的配置方式，先来查看 Tomcat 服务器安装目录下的 web.xml 文件是如何配置的。打开 Tomcat 安装目录中 conf 文件夹下的 web.xml 文件，可以看到如下所示的一段代码：

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

在上述代码中，＜welcome-file-list＞元素用于配置默认页面列表，它包含多个＜welcome-file＞子元素，每个＜welcome-file＞子元素都可以指定一个页面文件。当用户访问 Web 应用时，如果没有指定具体要访问的页面资源，Tomcat 则会按照＜welcome-file-list＞元素指定默认页面的顺序，依次查找这些默认页面，如果找到，则将其返回给用户，并停止查找后面的默认页面，如果没有找到，则返回访问资源不存在的错误提示页面。

在实际项目应用中，如果要配置默认页面，则只需在 Web 应用的 web.xml 文件内按照上面的配置方式添加默认的页面信息。例如，将应用中的 welcome.html 页面配置为默认页面，只需在 web.xml 中进行如下配置：

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

配置完成后，项目的默认访问页面就是 welcome.html。

8.4.4　在 Eclipse 中配置 Tomcat

Eclipse 作为一款强大的软件集成开发工具，对 Web 服务器也提供了非常好的支持，它可以集成各种 Web 服务器，以方便程序员进行 Web 开发。下面将按步骤讲解如何在 Eclipse 工具中配置 Tomcat 服务器。

(1) 在 Eclipse 的工具栏中，选择 Window→Preferences 选项，在弹出的 Preferences 窗口中选择左侧菜单中的 Server→Runtime Environments 选项，打开 Server Runtime Environments 选项卡，在当前选项卡中单击右侧 Add 按钮，在弹出的 New Server Runtime Environment 窗口中选择 Apache Tomcat v7.0 选项，然后单击 Next 按钮，在弹出的新窗口中，单击 Browse 按钮配置之前安装的 Tomcat 服务器(假设之前 Tomcat 服务器的下载安装目录为 D:\Tomcat\apache-tomcat-7.0.70)，最后单击 Finish 按钮即可，如图 8-10 所示。

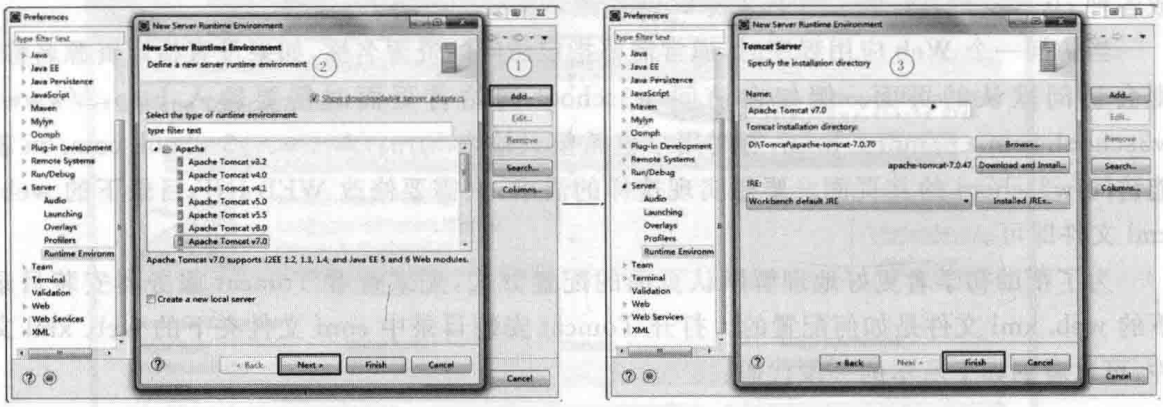


图 8-10 选择 Tomcat 服务器的安装目录

(2) 单击 Eclipse 下侧窗口的 Servers 选项卡标签(如果没有这个选项卡,则可以通过 Windows→Show View 找到并打开),在该选项卡中可以看到一个“No servers available. Define a new server from the new server wizard...”的链接,单击这个链接,会弹出一个 New Server 窗口,如图 8-11 所示。

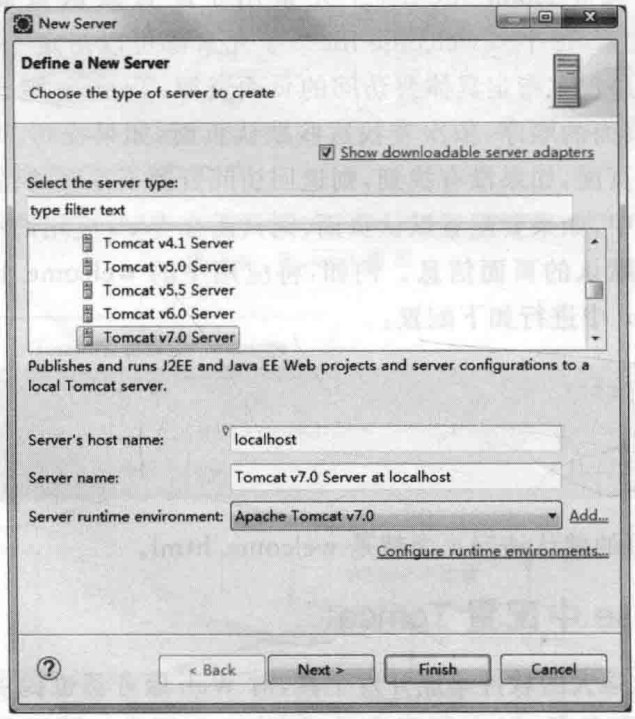


图 8-11 New Server 窗口

(3) 选中图 8-11 所示的 Tomcat v7.0 Server 选项,单击 Finish 按钮,即可完成 Tomcat 服务器的创建。这时,在 Servers 选项卡中会出现一个 Tomcat v7.0 Server at localhost 选项,如图 8-12 所示。

此时如果创建项目并使用 Eclipse 发布后,项目则会发布到 Eclipse 的 metadata 文件夹中。为了方便查找发布后的项目目录,读者也可以将项目直接发布到 Tomcat 中,这时就需要对 Server 进行配置。具体配置方法如下。

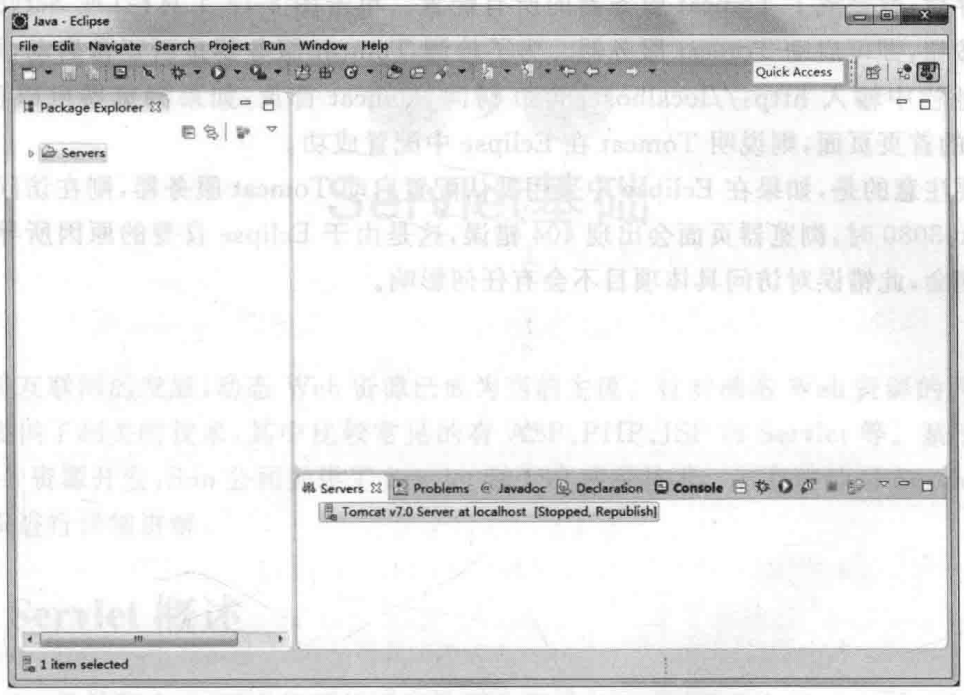


图 8-12　Eclipse 中创建 Tomcat 服务器

双击图 8-12 中 Servers 窗口内创建好的 Tomcat 服务器，在打开的 Overview 页面中，选择 Server Locations 选项中的 Use Tomcat installation，并将 Deploy path 文本框的内容修改为“webapps”，如图 8-13 所示。

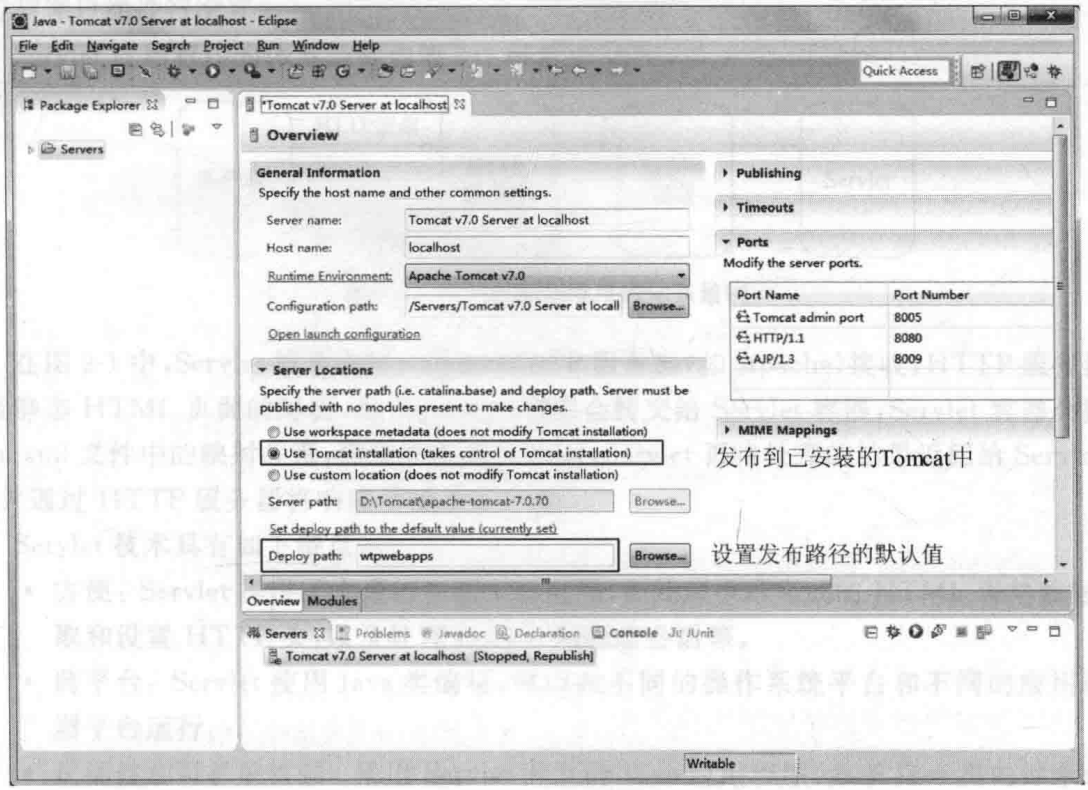


图 8-13　Overview 页面

保存后，就完成了 Tomcat 服务器的所有配置。单击图 8-13 工具栏(或 Servers 窗口)中的按钮，即可启动 Tomcat 服务器。为了检测 Tomcat 服务器是否正常启动，可以在浏览器地址栏中输入 http://localhost:8080 访问 Tomcat 首页，如果浏览器可以正常显示 Tomcat 的首页页面，则说明 Tomcat 在 Eclipse 中配置成功。

需要注意的是，如果在 Eclipse 中采用默认配置启动 Tomcat 服务器，则在访问 http://localhost:8080 时，浏览器页面会出现 404 错误，这是由于 Eclipse 自身的原因所导致的，读者不必理会，此错误对访问具体项目不会有任何影响。

第9章 Servlet基础

随着互联网的发展，动态 Web 资源已成为当前主流。针对动态 Web 资源的开发，很多公司都提供了相关的技术，其中比较常见的有 ASP、PHP、JSP 和 Servlet 等。基于 Java 的动态 Web 资源开发，Sun 公司提供了 Servlet 和 JSP 两种技术。本章将针对 Servlet 技术的相关知识进行详细讲解。

9.1 Servlet 概述

Servlet 是使用 Java 语言编写的运行在服务器端的程序，通常来说，Servlet 是指所有实现了 Servlet 接口的类。Servlet 主要用于处理客户端传来的 HTTP 请求，并返回一个响应，它能够处理的请求有 doGet()和 doPost()等。

Servlet 由 Servlet 容器提供，Servlet 容器是指提供了 Servlet 功能的服务器（如 Tomcat）。Servlet 容器会将 Servlet 动态加载到服务器上，然后通过 HTTP 请求和 HTTP 响应与客户端进行交互。

Servlet 应用程序的体系结构如图 9-1 所示。

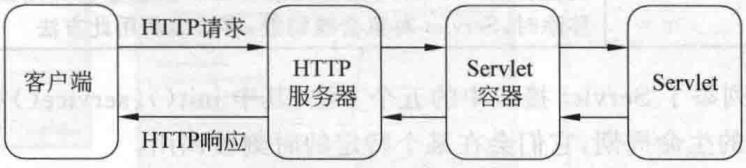


图 9-1 Servlet 应用程序的体系结构

在图 9-1 中，Servlet 的请求首先会被 HTTP 服务器（如 Apache）接收，HTTP 服务器只负责静态 HTML 页面的解析，而 Servlet 的请求会转交给 Servlet 容器，Servlet 容器会根据 web.xml 文件中的映射关系，调用相应的 Servlet，Servlet 再将处理的结果返回给 Servlet 容器，并通过 HTTP 服务器将响应传输给客户端。

Servlet 技术具有如下特点。

- 方便：Servlet 提供了大量的实用工具例程，如处理很难完成的 HTML 表单数据、读取和设置 HTTP 头，以及处理 Cookie 和跟踪会话等。
- 跨平台：Servlet 使用 Java 类编写，可以在不同的操作系统平台和不同的应用服务器平台运行。
- 灵活性和可扩展性强：采用 Servlet 开发的 Web 应用程序，由于 Java 类的继承性及构造函数等特点，使得应用灵活，可随意扩展。

除了上述几点以外，Servlet 还具有功能强大、能够在各个程序之间共享数据、安全性强等特点，此处不再详细说明，读者简单了解即可。

9.2 Servlet 开发入门

9.2.1 Servlet 接口及其实现类

Sun 公司提供了一系列的接口和类用于 Servlet 技术的开发，其中最重要的接口是 javax. servlet. Servlet。在 Servlet 接口中定义了五个抽象方法，如表 9-1 所示。

表 9-1 Servlet 接口的抽象方法

方法声明	功能描述
void init(ServletConfig config)	容器在创建好 Servlet 对象后，就会调用此方法。该方法接收一个 ServletConfig 类型的参数，Servlet 容器通过该参数向 Servlet 传递初始化配置信息
ServletConfig getServletConfig()	用于获取 Servlet 对象的配置信息，返回 Servlet 的 ServletConfig 对象
String getServletInfo()	返回一个字符串，其中包含关于 Servlet 的信息，如作者、版本和版权等信息
voidservice(ServletRequest request,ServletResponse response)	负责响应用户的请求，当容器接收到客户端访问 Servlet 对象的请求时，就会调用此方法。容器会构造一个表示客户端请求信息的 ServletRequest 对象和一个用于响应客户端的 ServletResponse 对象作为参数传递给 service() 方法。在 service() 方法中，可以通过 ServletRequest 对象得到客户端的相关信息和请求信息，在对请求进行处理后，调用 ServletResponse 对象的方法设置响应信息
void destroy()	负责释放 Servlet 对象占用的资源。当服务器关闭或者 Servlet 对象被移除时，Servlet 对象会被销毁，容器会调用此方法

在表 9-1 中，列举了 Servlet 接口中的五个方法，其中 init()、service()和 destroy()方法可以表现 Servlet 的生命周期，它们会在某个特定的时刻被调用。

针对 Servlet 的接口，Sun 公司提供了两个默认的接口实现类：GenericServlet 和 HttpServlet。其中，GenericServlet 是一个抽象类，该类为 Servlet 接口提供了部分实现，它并没有实现 HTTP 请求处理；HttpServlet 是 GenericServlet 的子类，它继承了 GenericServlet 的所有方法，并且为 HTTP 请求中的 GET 和 POST 等类型提供了具体的操作方法。通常情况下，编写的 Servlet 类都继承自 HttpServlet，在开发中使用的也是 HttpServlet 对象。

HttpServlet 类中包含两个常用方法，这两个方法的说明如表 9-2 所示。

表 9-2 HttpServlet 类的常用方法

方法声明	功能描述
protected void doGet(HttpServletRequest req, HttpServletResponse resp)	用于处理 GET 类型的 HTTP 请求的方法
protected void doPost(HttpServletRequest req, HttpServletResponse resp)	用于处理 POST 类型的 HTTP 请求的方法

HttpServlet 主要有两大功能，具体如下。

(1) 根据用户请求方式的不同，定义相应的 doXxx()方法处理用户请求。例如，与 GET 请求方式对应的 doGet()方法，与 POST 方式对应的 doPost()方法。

(2) 通过 service()方法将 HTTP 请求和响应分别强转为 HttpServletRequest 和 HttpServletResponse 类型的对象。

需要注意的是，由于 HttpServlet 类在重写的 service()方法中，为每一种 HTTP 请求方式都定义了对应的 doXxx()方法，因此，当定义的类继承 HttpServlet 后，只需要根据请求方式重写对应的 doXxx()方法即可，而不需要重写 service()方法。

9.2.2　实现第一个 Servlet 程序

为了帮助读者快速学习 Servlet 开发，下面分步骤实现一个 Servlet 程序。

1. 创建 Web 项目

选择 Eclipse 上方工具栏中的 File→New→Other 选项，在 Select a wizard 窗口中选择 Dynamic Web Project 选项，单击 Next 按钮，进入填写项目信息的界面，选择的运行环境是 Tomcat 7.0，动态 Web 模块版本为 2.5，持续单击 Next 按钮，直到进入 Configure web module settings 配置界面，如图 9-2 所示。

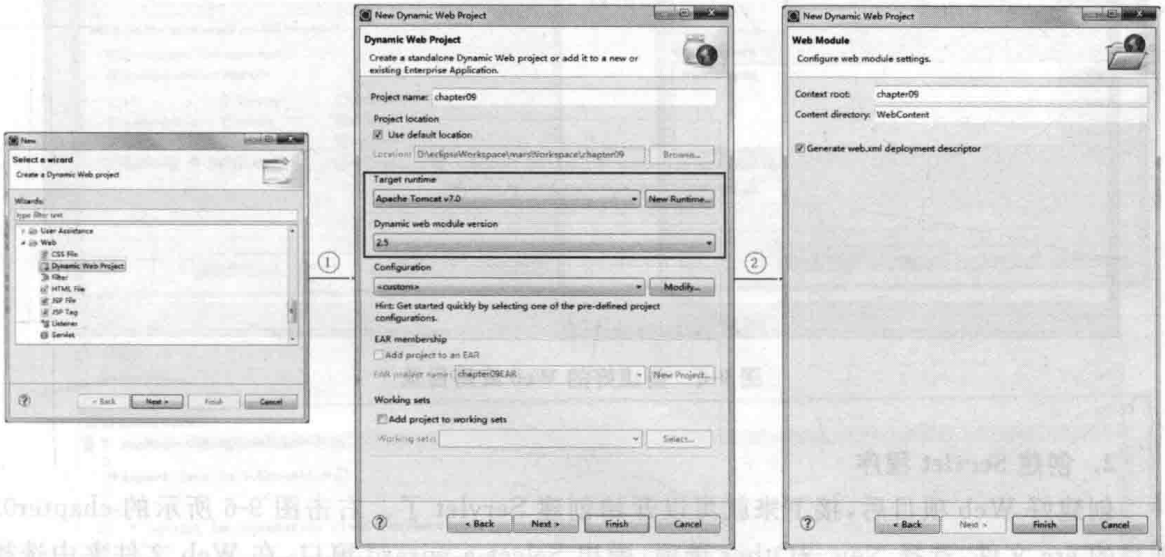


图 9-2　新建项目

在图 9-2 中，Context root 选项用于指定 Web 项目的根目录，Content directory 选项用于指定存放 Web 资源的目录。这里采用默认设置的目录，将 chapter09 作为 Web 资源的根目录，将 WebContent 作为存放 Web 资源的目录。单击 Finish 按钮，完成 Web 项目的配置。

需要注意的是，如果 Eclipse 中使用的是 Java 视图，则单击 Finish 后，会弹出 Open Associated Perspective 确认提示框(如图 9-3 所示)，出现此窗口的原因是所创建的 Web 项目关联了 Java EE 视图，由于两种视图在开发使用时区别不大，而且在 Eclipse 中可做相应设置，所以此处依照个人习惯选择相应视图即可。

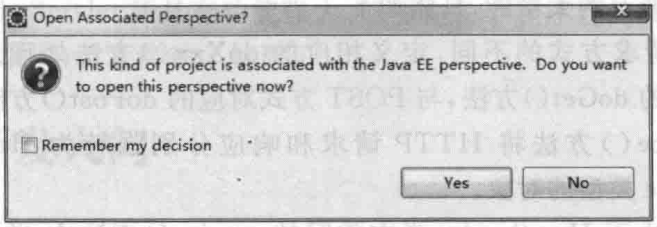


图 9-3 开启关联视图确认框

本书为了能够更清晰地展示项目结构，所以均使用 Java 视图，单击 No 按钮后，Web 应用目录如图 9-4 所示。

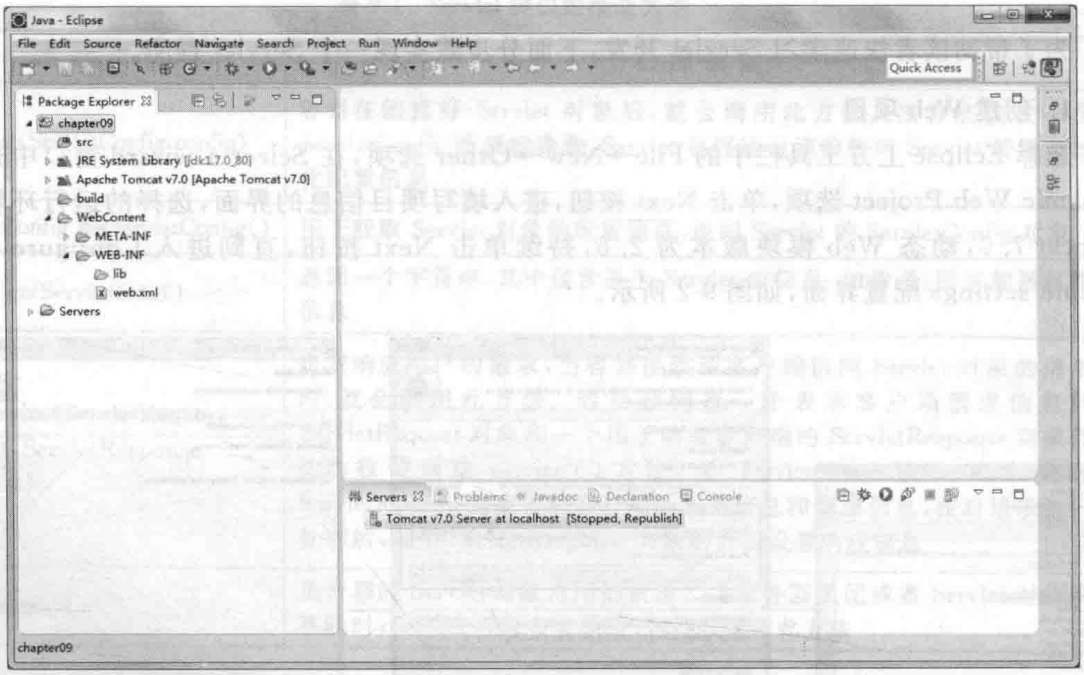


图 9-4 创建好的 Web 应用目录

2. 创建 Servlet 程序

创建好 Web 项目后，接下来就可以开始创建 Servlet 了。右击图 9-6 所示的 chapter09 项目的 src 文件，选择 New→Other 选项，弹出 Select a wizard 窗口，在 Web 文件夹中选择 Servlet，单击 Next 按钮，进入填写 Servlet 信息的界面。其中，Java package 用于指定 Servlet 所在包的名称，Class name 用于指定 Servlet 的名称，填写完成后单击 Next 按钮，进入配置 Servlet 的界面，如图 9-5 所示。

在图 9-5 中，Servlet 的配置界面的 Name 选项用于指定 web. xml 文件中<servlet-name>元素的内容，URL mapping 文本框用于指定 web. xml 文件中<url-pattern>元素的内容，这两个选项的内容都是可以修改的，此处不做任何修改，采用默认设置的内容，单击 Next 按钮，进入下一个配置界面，在该界面中可以选择需要的创建方法，这里只选择 Inherited abstract methods、doGet 和 doPost 方法，单击 Finish 按钮，即可完成 Servlet 的创建。

TestServlet01 创建后的内容如图 9-6 所示。

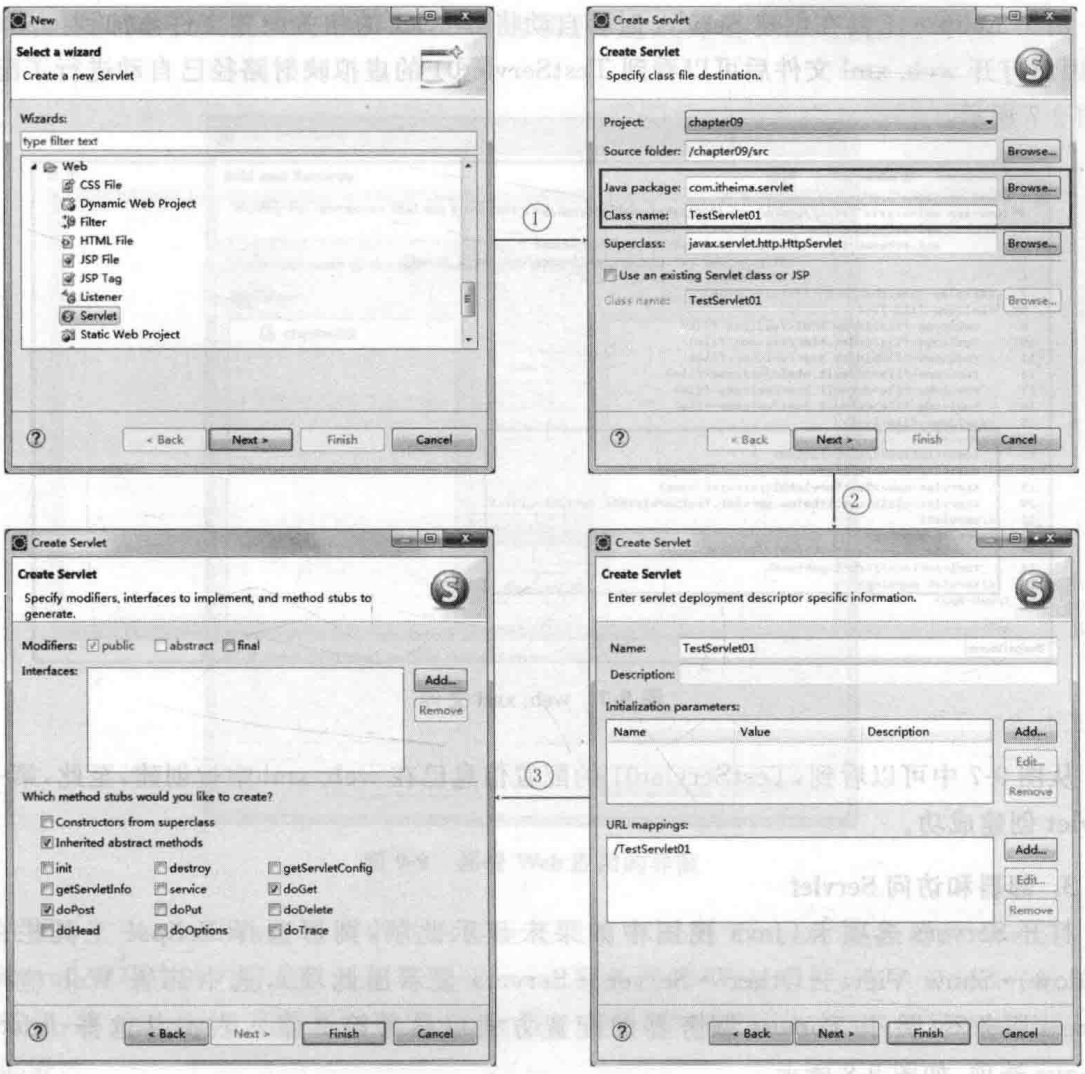


图 9-5　填写 Servlet 的界面

TestServlet01.java

```
1  package com.itheima.servlet;
2
3* import java.io.IOException;
8
9  /**
10  * Servlet implementation class TestServlet01
11  */
12 public class TestServlet01 extends HttpServlet {
13     private static final long serialVersionUID = 1L;
14
15     /**
16      * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse response)
17      */
18     protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
19         // TODO Auto-generated method stub
20         response.getWriter().append("Served at: ").append(request.getContextPath());
21     }
22
23     /**
24      * @see HttpServlet#doPost(HttpServletRequest request, HttpServletResponse response)
25      */
26     protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
27         // TODO Auto-generated method stub
28         doGet(request, response);
29     }
30
31 }
```

图 9-6　创建后的 TestServlet01 类

由于 Eclipse 工具在创建 Servlet 时会自动将 Servlet 的相关配置文件添加到 web. xml 中，因此打开 web. xml 文件后可以看到 TestServlet01 的虚拟映射路径已自动进行了配置，如图 9-7 所示。

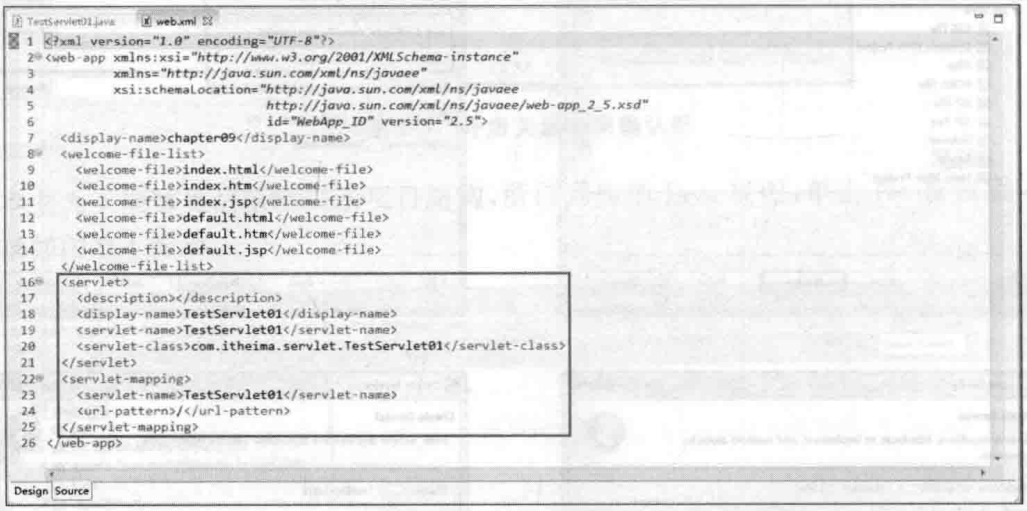


图 9-7　web. xml 文件

从图 9-7 中可以看到，TestServlet01 的配置信息已在 web. xml 中被创建，至此，第一个 Servlet 创建成功。

3. 部署和访问 Servlet

打开 Servers 选项卡(Java 视图中如果未显示此项，则可选择 Eclipse 工具栏中的 Window→Show View→Other→Server→Servers 显示出此项)，选中部署 Web 应用的 Tomcat 服务器(关于 Tomcat 服务器的配置方式可参考第 2 章)，右击并选择 Add and Remove 选项，如图 9-8 所示。

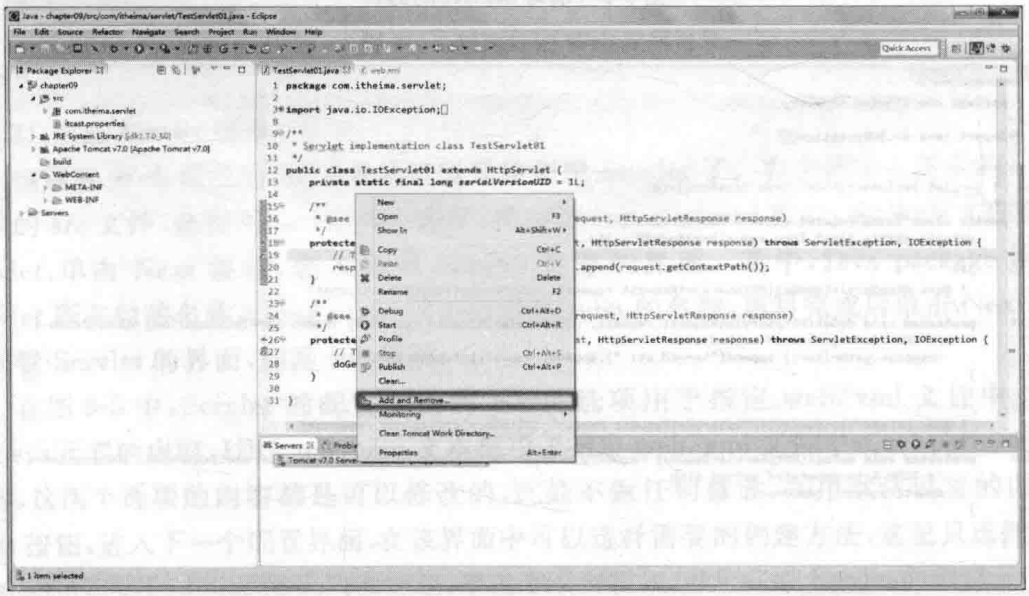


图 9-8　Add and Remove 选项

单击图 9-8 所示的 Add and Remove 选项后，进入部署 Web 应用的界面，如图 9-9 所示。

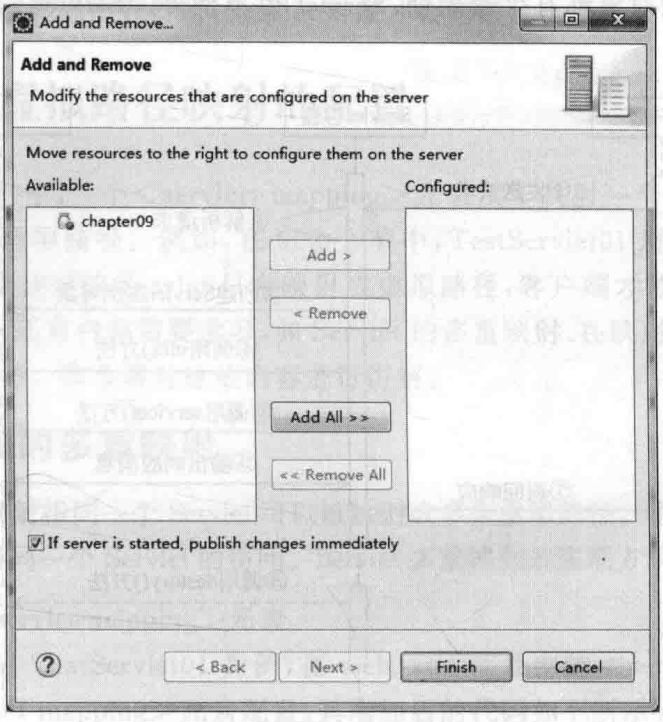


图 9-9　部署 Web 应用的界面

在图 9-9 中，Available 选项中的内容是还没有部署到 Tomcat 服务器的 Web 项目，Configured 选项中的内容是已经部署到 Tomcat 服务器的 Web 项目，选中 chapter09，单击 Add 按钮，将 chapter09 项目添加到 Tomcat 服务器后，单击 Finish 按钮，即可完成 Web 应用的部署。

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter09/TestServlet01 访问 TestServlet01，浏览器的显示结果如图 9-10 所示。

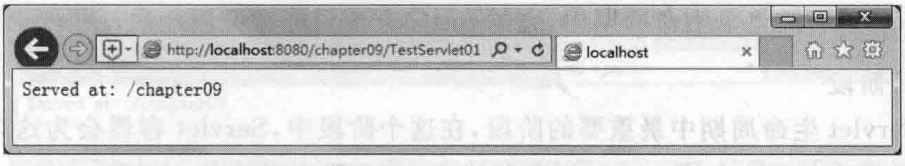


图 9-10　访问 EclipseServlet

从图 9-10 中可以看出，浏览器中已经显示出了 TestServlet01 中 doGet()方法需要输出的内容，至此，已经完成了使用 Eclipse 工具开发的第一个 Servlet 程序。

需要注意的是，不同版本的 Eclipse 创建 Servlet 后的 doGet()和 doPost()方法中的内容可能有所不同，如果这两个方法中没有编写输出代码，那么使用浏览器访问时不会显示任何内容，此时读者可仿照图 9-10 中的内容自行编写输出语句。

9.2.3 Servlet 的生命周期

在 Java 中，任何对象都有生命周期，Servlet 也不例外。Servlet 的生命周期如图 9-11 所示。

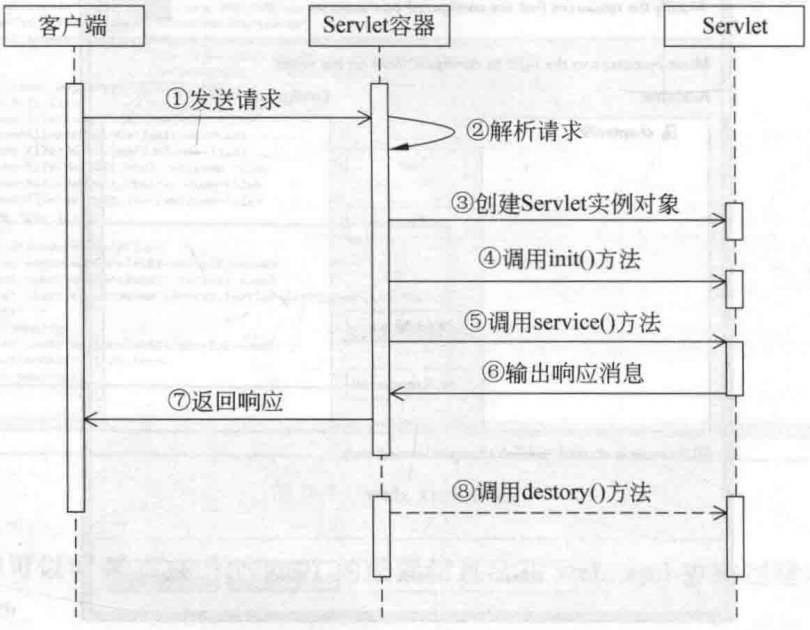


图 9-11 Servlet 的生命周期

图 9-11 描述了 Servlet 的生命周期。按照功能的不同，大致可以将 Servlet 的生命周期分为三个阶段，分别是初始化阶段、运行阶段和销毁阶段。

1. 初始化阶段

当客户端向 Servlet 容器发出 HTTP 请求要求访问 Servlet 时，Servlet 容器首先会解析请求，检查内存中是否已经有了该 Servlet 对象，如果有，则直接使用该 Servlet 对象，如果没有，则创建 Servlet 实例对象，然后通过调用 init()方法实现 Servlet 的初始化工作。需要注意的是，在 Servlet 的整个生命周期内，它的 init()方法只能被调用一次。

2. 运行阶段

这是 Servlet 生命周期中最重要的阶段，在这个阶段中，Servlet 容器会为这个请求创建代表 HTTP 请求的 ServletRequest 对象和代表 HTTP 响应的 ServletResponse 对象，然后将它们作为参数传递给 Servlet 的 service()方法。service()方法从 ServletRequest 对象中获得客户请求信息并处理该请求，通过 ServletResponse 对象生成响应结果。在 Servlet 的整个生命周期内，对于 Servlet 的每一次访问请求，Servlet 容器都会调用一次 Servlet 的 service()方法，并且创建新的 ServletRequest 和 ServletResponse 对象，也就是说，service()方法在 Servlet 的整个生命周期中会被调用多次。

3. 销毁阶段

当服务器关闭或 Web 应用被移除出容器时，Servlet 随着 Web 应用的关闭而销毁。在销毁 Servlet 之前，Servlet 容器会调用 Servlet 的 destroy()方法，以便让 Servlet 对象释放它

所占用的资源。在 Servlet 的整个生命周期中，destroy()方法也只能被调用一次。

需要注意的是，Servlet 对象一旦创建就会驻留在内存中等待客户端的访问，直到服务器关闭或 Web 应用被移除出容器时，Servlet 对象才会销毁。

9.3　Servlet 虚拟路径映射的配置

在 web.xml 文件中，一个<servlert-mapping>元素用于映射一个 Servlet 的对外访问路径，该路径也称为虚拟路径。例如，在 9.2.2 节中，TestServlet01 所映射的虚拟路径为“/TestServlet01”。创建好的 Servlet 只有映射成虚拟路径，客户端才能对其进行访问。但是在映射 Servlet 时，还有内容需要学习，如 Servlet 的多重映射、在映射路径中使用通配符、配置默认的 Servlet 等。本节将对这些内容进行讲解。

9.3.1　Servlet 的多重映射

Servlet 的多重映射指同一个 Servlet 可以被映射成多条虚拟路径。也就是说，客户端可以通过多条路径实现对同一个 Servlet 的访问。Servlet 多重映射的实现方式有以下两种。

1. 配置多个<servlet-mapping>元素

以 9.2.2 节中的 TestServlet01 为例，在 web.xml 文件中的<servlet-mapping>元素下，增加一个<servlet-mapping>元素配置，其增加后的代码如下所示：

```
<servlet-mapping>
    <!--映射为 Test01-->
    <servlet-name>TestServlet01</servlet-name>
    <url-pattern>/Test01</url-pattern>
</servlet-mapping>
```

重启 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter09/TestServlet01 访问 TestServlet01，浏览器的显示结果如图 9-12 所示。

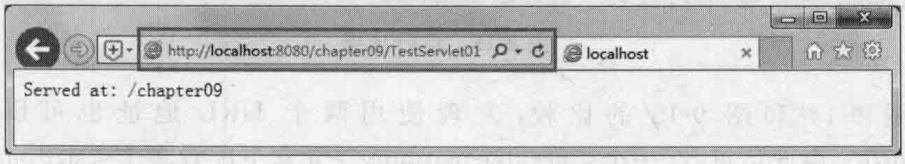


图 9-12　运行结果

在浏览器的地址栏中输入地址 http://localhost:8080/chapter09/Test01 访问 TestServlet01，浏览器显示的结果如图 9-13 所示。

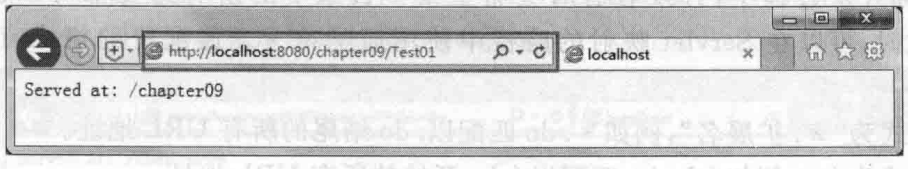


图 9-13　运行结果

通过图 9-12 和图 9-13 的比较，发现使用两个 URL 地址都可以正常访问 TestServlet01。由此可见，通过配置多个<servlet-mapping>元素可以实现 Servlet 的多重映射。

2. 配置多个<url-pattern>子元素

同样以 TestServlet01 为例，在 web.xml 文件中对 TestServlet01 的<servlet-mapping>元素进行修改，在该元素中配置多个<url-pattern>，修改后的代码如下所示：

```
<servlet-mapping>
    <!--映射为 TestServlet01 和 Test02-->
    <servlet-name>TestServlet01</servlet-name>
    <url-pattern>/TestServlet01</url-pattern>
    <url-pattern>/Test02</url-pattern>
</servlet-mapping>
```

重启 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter09/TestServlet01 访问 TestServlet01，浏览器的显示结果如图 9-14 所示。

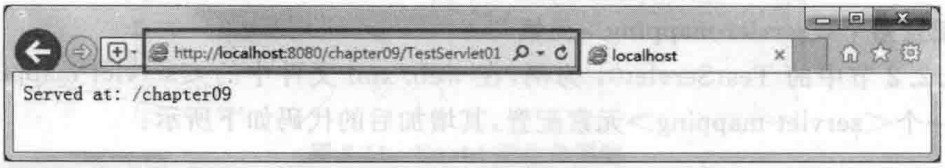


图 9-14 运行结果

在浏览器的地址栏中输入地址 http://localhost:8080/chapter09/Test02 访问 TestServlet01，浏览器的显示结果如图 9-15 所示。

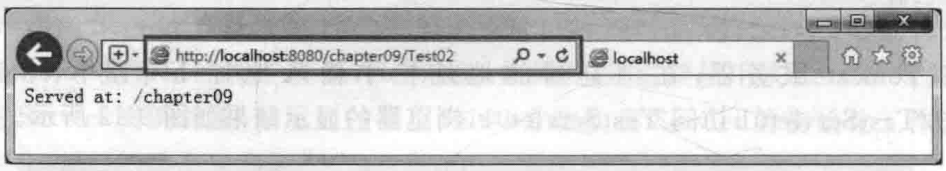


图 9-15 运行结果

通过图 9-14 和图 9-15 的比较，发现使用两个 URL 地址也可以正常访问 TestServlet01。由此可见，在一个<servlet-mapping>元素下配置多个<url-pattern>子元素同样可以实现 Servlet 的多重映射。

9.3.2 Servlet 映射路径中使用通配符

在实际开发过程中，开发者有时会希望某个目录下的所有路径都可以访问同一个 Servlet，这时，可以在 Servlet 映射的路径中使用通配符 *。通配符的格式有两种，具体如下。

- 格式为“*.扩展名”，例如 *.do 匹配以.do 结尾的所有 URL 地址。
- 格式为/*，例如/abc/* 匹配以/abc 开始的所有 URL 地址。

需要注意的是，这两种通配符的格式不能混合使用，例如，/abc/ * . do 是不合法的映射路径。另外，当客户端访问一个 Servlet 时，如果请求的 URL 地址能够匹配多条虚拟路径，那么 Tomcat 将采取最具体匹配原则查找与请求 URL 最接近的虚拟映射路径。例如，对于如下所示的一些映射关系：

```
/abc/*        映射到 Servlet1
/*            映射到 Servlet2
/abc          映射到 Servlet3
*.do          映射到 Servlet4
```

- 当请求 URL 为/abc/a. html，/abc/ * 和/ * 都可以匹配这个 URL，Tomcat 会调用 Servlet1。
- 当请求 URL 为/abc、/ * 、/abc/ * 和/abc 都可以匹配这个 URL，Tomcat 会调用 Servlet3。
- 当请求 URL 为/abc/a. do、/ * 、* . do 和/abc/ * 都可以匹配这个 URL，Tomcat 会调用 Servlet1。
- 当请求 URL 为/a. do，/ * 和 * . do 都可以匹配这个 URL，Tomcat 会调用 Servlet2。
- 当请求 URL 为/xxx/yyy/a. do，* . do 和/ * 都可以匹配这个 URL，Tomcat 会调用 Servlet2。

9.3.3　默认 Servlet

如果某个 Servlet 的映射路径仅仅是一个正斜线(/)，那么这个 Servlet 就是当前 Web 应用的默认 Servlet。Servlet 服务器在接收到访问请求时，如果在 web. xml 文件中找不到匹配的<servlet-mapping>元素的 URL，则会将访问请求交给默认 Servlet 处理，也就是说，默认 Servlet 用于处理其他 Servlet 都不处理的访问请求。

下面对 9. 2. 2 节中的 web. xml 文件进行修改，将其设置为默认的 Servlet，具体如下：

```
<servlet>
    <servlet-name>TestServlet01</servlet-name>
    <servlet-class>com.itheima.servlet.TestServlet01</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>TestServlet01</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入任意地址，如 http://localhost:8080/chapter09/abcde，浏览器的显示结果如图 9-16 所示。

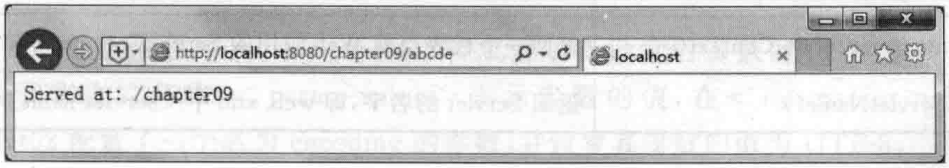


图 9-16　运行结果

从图 9-16 中可以看出，当 URL 地址和 TestServlet01 的虚拟路径不匹配时，浏览器仍然可以正常访问 TestServlet01。

需要注意的是，在 Tomcat 安装目录下的 web.xml 文件中也配置了一个默认的 Servlet，配置信息如下所示：

```
<servlet>
    <servlet-name>default</servlet-name>
    <servlet-class>org.apache.catalina.servlets.DefaultServlet</servlet-
class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>default</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

在上面的配置信息中，org.apache.catalina.servlets.DefaultServlet 被设置为默认的 Servlet，它对 Tomcat 服务器上所有的 Web 应用都起作用。当 Tomcat 服务器中的某个 Web 应用没有默认 Servlet 时，都会将 DefaultServlet 作为默认的 Servlet。当客户端访问 Tomcat 服务器中的某个静态 HTML 文件时，DefaultServlet 会判断 HTML 是否存在，如果存在，则会将数据以流的形式回送给客户端，否则会报告 404 错误。

9.4 ServletConfig 和 ServletContext

9.4.1 ServletConfig 接口

在运行 Servlet 程序时，可能需要一些辅助信息，例如，文件使用的编码、使用 Servlet 程序的共享信息等，这些信息可以在 web.xml 文件中使用一个或多个<init-param>元素进行配置。当 Tomcat 初始化一个 Servlet 时，会将该 Servlet 的配置信息封装到 ServletConfig 对象中，此时可以通过调用 init(ServletConfig config)方法将 ServletConfig 对象传递给 Servlet。

ServletConfig 接口中定义了一系列获取配置信息的方法，如表 9-3 所示。

表 9-3 ServletConfig 接口的常用方法

方法说明	功能描述
String getInitParameter(String name)	根据初始化参数名返回对应的初始化参数值
Enumeration getInitParameterNames()	返回一个 Enumeration 对象，其中包含了所有的初始化参数名
ServletContext getServletContext()	返回一个代表当前 Web 应用的 ServletContext 对象
String getServletName()	返回 Servlet 的名字，即 web.xml 中<servlet-name>元素的值

下面以 getInitParameter()方法为例，分步骤讲解该方法的使用。

1. 创建 Servlet

在 com.itheima.servlet 包中创建一个名称为 TestServlet02 的 Servlet 类，并在类中编写用于读取 web.xml 文件中参数信息的代码，如文件 9-1 所示。

文件 9-1　TestServlet02.java

```
1  package com.itheima.servlet;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class TestServlet02 extends HttpServlet {
6  protected void doGet(HttpServletRequest request,
7      HttpServletResponse response) throws ServletException, IOException {
8          PrintWriter out=response.getWriter();
9          ServletConfig config=this.getServletConfig();
10         String param=config.getInitParameter("encoding");
11          out.println("encoding="+param);
12     }
13  protected void doPost(HttpServletRequest request,
14     HttpServletResponse response) throws ServletException, IOException {
15         this.doGet(request, response);
16     }
17 }
```

2. 配置参数信息

在 web.xml 文件中为 Servlet 配置一些参数信息，具体的配置代码如下所示：

```
1  <servlet>
2      <servlet-name>TestServlet02</servlet-name>
3      <servlet-class>com.itheima.servlet.TestServlet02</servlet-class>
4      <init-param>
5          <param-name>encoding</param-name>
6          <param-value>UTF-8</param-value>
7      </init-param>
8  </servlet>
9  <servlet-mapping>
10      <servlet-name>TestServlet02</servlet-name>
11      <url-pattern>/TestServlet02</url-pattern>
12 </servlet-mapping>
```

在上面的参数信息中，<init-param>节点表示要设置的参数，该节点中的<param-name>表示参数的名称，<param-value>表示参数的值，在<init-param>节点中为 TestServlet02 配置了一个名为 encoding 的参数，并设置其参数的值为 UTF-8。

3. 运行项目并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter09/TestServlet02 访问 TestServlet02，结果如图 9-17 所示。

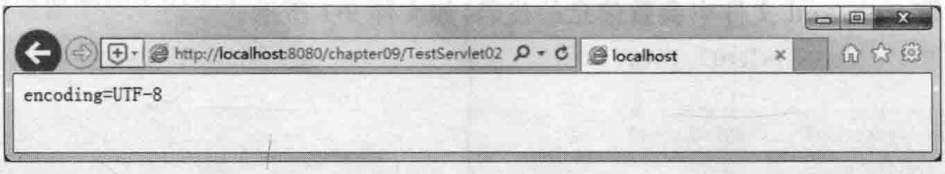


图 9-17 运行结果

从图 9-17 中可以看出，web.xml 文件中为 TestServlet02 配置的编码信息被读取了出来。由此可见，通过 ServletConfig 对象可以获得 web.xml 文件中的参数信息。

9.4.2 ServletContext 接口

当 Tomcat 启动时，Tomcat 会为每个 Web 应用创建一个唯一的 ServletContext 对象代表当前的 Web 应用，该对象封装了当前 Web 应用的所有信息。可以利用该对象获取 Web 应用程序的初始化信息、读取资源文件等。下面对 ServletContext 接口的不同作用分别进行讲解。

1. 获取 Web 应用程序的初始化参数

在 web.xml 文件中，不仅可以配置 Servlet 的映射信息，还可以配置整个 Web 应用的初始化信息。Web 应用初始化参数的配置方式具体如下所示：

```
<context-param>
    <param-name>XXX</param-name>
    <param-value>xxx</param-value>
</context-param>
<context-param>
    <param-name>AAA</param-name>
    <param-value>aaa</param-value>
</context-param>
```

在上面的示例中，<context-param>元素位于根元素<web-app>中，它的子元素<param-name>和<param-value>分别用于指定参数的名字和参数值。要想获取这些参数名和参数值的信息，可以使用 ServletContext 接口中定义的 getInitParameterNames()和 getInitParameter(String name)方法分别获取。

下面通过案例演示如何使用 ServletContext 接口获取 Web 应用程序的初始化参数。

(1) 在 chapter09 项目的 web.xml 文件中配置初始化参数信息和 Servlet 信息，其代码如下所示：

```
<context-param>
    <param-name>username</param-name>
    <param-value>admin</param-value>
```

```
</context-param>
<context-param>
    <param-name>password</param-name>
    <param-value>1234</param-value>
</context-param>
<servlet>
    <servlet-name>TestServlet03</servlet-name>
    <servlet-class>com.itheima.servlet.TestServlet03</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>TestServlet03</servlet-name>
    <url-pattern>/TestServlet03</url-pattern>
</servlet-mapping>
```

(2) 在项目的 com. itheima. servlet 包中创建一个名称为 TestServlet03 的类，该类中使用 ServletContext 接口获取 web. xml 中的配置信息，如文件 9-2 所示。

文件 9-2　TestServlet03. java

```
1  package com.itheima.servlet;
2  import java.io.*;
3  import java.util.*;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  public class TestServlet03 extends HttpServlet {
7      public void doGet(HttpServletRequest request,
8      HttpServletResponse response)throws ServletException, IOException {
9          response.setContentType("text/html;charset=utf-8");
10         PrintWriter out=response.getWriter();
11         //得到 ServletContext 对象
12         ServletContext context=this.getServletContext();
13         //得到包含所有初始化参数名的 Enumeration 对象
14         Enumeration<String>paramNames=context.getInitParameterNames();
15         //遍历所有的初始化参数名,得到相应的参数值并打印
16         while (paramNames.hasMoreElements()) {
17             String name=paramNames.nextElement();
18             String value=context.getInitParameter(name);
19             out.println(name+": "+value);
20             out.println("<br/>");
21         }
22     }
23     public void doPost(HttpServletRequest request,
24       HttpServletResponse response)throws ServletException, IOException {
25         this.doGet(request, response);
26     }
27 }
```

在文件 9-2 中，当通过 this. getServletContext()方法获取到 ServletContext 对象后，首先调用 getInitParameterNames()方法，获取到包含所有初始化参数名的 Enumeration 对象，然后遍历 Enumeration 对象，根据获取到的参数名，通过 getInitParamter(String name)方法得到对应的参数值。

(3) 启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter09/TestServlet03 访问 TestServlet03，浏览器的显示结果如图 9-18 所示。

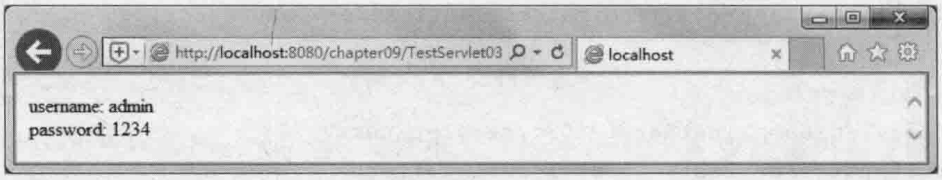


图 9-18 运行结果

从图 9-18 中可以看出，web. xml 文件中配置的信息被读取了出来。由此可见，通过 ServletContext 对象可以获取 Web 应用的初始化参数。

2. 读取 Web 应用下的资源文件

在实际开发中，有时会需要读取 Web 应用中的一些资源文件，如配置文件和日志文件等。为此，在 ServletContext 接口中定义了一些读取 Web 资源的方法，这些方法是依靠 Servlet 容器实现的。Servlet 容器根据资源文件相对于 Web 应用的路径，返回关联资源文件的 I/O 流或资源文件在系统的绝对路径等。

表 9-4 中列举了 ServletContext 接口中用于获取资源路径的相关方法。

表 9-4 ServletContext 接口的常用方法

方法说明	功能描述
Set getResourcePaths(String path)	返回一个 Set 集合，集合中包含资源目录中子目录和文件的路径名称。参数 path 必须以正斜线(/)开始，指定匹配资源的部分路径
String getRealPath(String path)	返回资源文件在服务器文件系统上的真实路径(文件的绝对路径)。参数 path 代表资源文件的虚拟路径，它应该以正斜线(/)开始，/表示当前 Web 应用的根目录，如果 Servlet 容器不能将虚拟路径转换为文件系统的真实路径，则返回 null
URL getResource(String path)	返回映射到某个资源文件的 URL 对象。参数 path 必须以正斜线(/)开始，/表示当前 Web 应用的根目录
InputStream getResourceAsStream (String path)	返回映射到某个资源文件的 InputStream 输入流对象。参数 path 的传递规则和 getResource()方法完全一致

了解了 ServletContext 接口中用于获得 Web 资源路径的方法后，下面通过案例分步骤演示如何使用 ServletContext 对象读取资源文件。

(1) 在 chapter09 项目的 src 目录中创建一个名称为 itcast. properties 的文件，在创建好的文件中输入如下所示的配置信息：

```
username=admin
password=1234
```

(2) 在 com.itheima.servlet 包中创建一个名称为 TestServlet04 的 Servlet 类，使用该类读取 itcast.properties 资源文件的内容，其实现代码如文件 9-3 所示。

文件 9-3　TestServlet04.java

```
1  package com.itheima.servlet;
2  import java.io.*;
3  import java.util.Properties;
4  import javax.servlet.*;
5  import javax.servlet.http.*;
6  public class TestServlet04 extends HttpServlet {
7      public void doGet(HttpServletRequest request,
8        HttpServletResponse response)throws ServletException, IOException {
9          response.setContentType("text/html;charset=utf-8");
10         ServletContext context=this.getServletContext();
11         PrintWriter out=response.getWriter();
12         //获取相对路径中的输入流对象
13         InputStream in=context
14             .getResourceAsStream("/WEB-INF/classes/itcast.properties");
15         Properties pros=new Properties();
16         pros.load(in);
17         out.println("username="+pros.getProperty("username")+"<br/>");
18         out.println("password="+pros.getProperty("password ")+"<br/>");
19     }
20     public void doPost(HttpServletRequest request,
21       HttpServletResponse response)throws ServletException, IOException {
22         this.doGet(request, response);
23     }
24 }
```

在文件 9-3 中，使用 ServletContext 的 getResourceAsStream(String path)方法获得了关联 itcast.properties 资源文件的输入流对象，其中的 path 参数必须以正斜线(/)开始，表示 itcast.properties 文件相对于 Web 应用的相对路径。

(3) 启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter09/TestServlet04 访问 TestServlet04，浏览器的显示结果如图 9-19 所示。

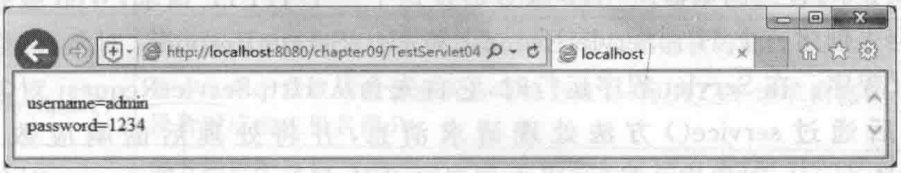


图 9-19　运行结果

从图 9-19 中可以看出，itcast.properties 资源文件中的内容已经被读取了出来。由此可见，使用 ServletContext 可以读取 Web 应用中的资源文件。

第 10 章 请求和响应

通过第 9 章的学习，读者对 Servlet 的基础知识已经有了一个初步的掌握。Servlet 最主要的作用就是处理客户端请求，并向客户端做出响应。本章将围绕 Servlet 的请求和响应进行详细讲解。

10.1 请求和响应概述

针对 Servlet 的每次请求，Web 服务器在调用 service() 方法之前，都会创建 HttpServletRequest 和 HttpServletResponse 对象。其中，HttpServletRequest 对象用于封装 HTTP 请求消息，简称 request 对象。HttpServletResponse 对象用于封装 HTTP 响应消息，简称 response 对象。浏览器访问 Servlet 的交互过程如图 10-1 所示。

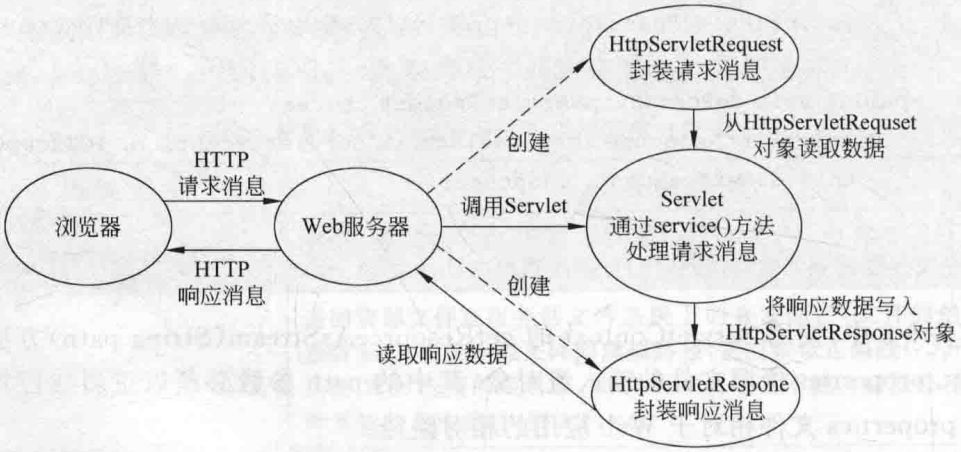


图 10-1 浏览器访问 Servlet 过程

在图 10-1 中，首先浏览器向 Web 服务器发送了一个 HTTP 请求，Web 服务器根据收到的请求，会先创建一个 HttpServletRequest 和 HttpServletResponse 对象，然后再调用相应的 Servlet 程序。在 Servlet 程序运行时，它首先会从 HttpServletRequest 对象中读取数据信息，然后通过 service() 方法处理请求消息，并将处理后的响应数据写入到 HttpServletResponse 对象中。最后，Web 服务器会从 HttpServletResponse 对象中读取到响应数据，并发送给浏览器。

需要注意的是，在 Web 服务器运行阶段，每个 Servlet 都只会创建一个实例对象，针对每次 HTTP 请求，Web 服务器都会调用所请求 Servlet 实例的 service(HttpServletRequest request, HttpServletResponse response) 方法，并重新创建一个 request 对象和一个 response

对象。

10.2　HttpServletRequest 对象

HttpServletRequest 接口继承自 ServletRequest 接口，其主要作用是封装 HTTP 请求消息。由于 HTTP 请求消息分为请求行、请求消息头和请求消息体三部分，因此，在 HttpServletRequest 接口中定义了获取请求行、请求头和请求消息体的相关方法。本节将针对常用的获取请求行和请求消息头的相关方法进行讲解。

10.2.1　获取请求行信息的相关方法

当访问 Servlet 时，所有请求消息将被封装到 HttpServletRequest 对象中，请求消息的请求行中包含请求方法、请求资源名、请求路径等信息，为了获取这些信息，HttpServletRequest 接口定义了一系列方法，如表 10-1 所示。

表 10-1　获取请求行信息的常用方法

方法声明	功能描述
String getMethod()	该方法用于获取 HTTP 请求消息中的请求方式(如 GET、POST 等)
String getRequestURI()	该方法用于获取请求行中的资源名称部分，即位于 URL 的主机和端口之后、参数部分之前的部分
String getQueryString()	该方法用于获取请求行中的参数部分，也就是资源路径后问号(?)以后的所有内容
String getContextPath()	该方法用于获取请求 URL 中属于 Web 应用程序的路径，这个路径以/开头，表示相对于整个 Web 站点的根目录，路径结尾不含/。如果请求 URL 属于 Web 站点的根目录，那么返回结果为空字符串("")
String getServletPath()	该方法用于获取 Servlet 的名称或 Servlet 所映射的路径
String getRemoteAddr()	该方法用于获取请求客户端的 IP 地址，其格式类似于 192.168.0.3
String getRemoteHost()	该方法用于获取请求客户端的完整主机名，其格式类似于 pc1.itheima.com。需要注意的是，如果无法解析出客户机的完整主机名，那么该方法将会返回客户端的 IP 地址
int getRemotePort()	该方法用于获取请求客户端网络连接的端口号
String getLocalAddr()	该方法用于获取 Web 服务器上接收当前请求网络连接的 IP 地址
String getLocalName()	该方法用于获取 Web 服务器上接收当前网络连接 IP 所对应的主机名
int getLocalPort()	该方法用于获取 Web 服务器上接收当前网络连接的端口号
String getServerName()	该方法用于获取当前请求所指向的主机名，即 HTTP 请求消息中 Host 头字段所对应的主机名部分
int getServerPort()	该方法用于获取当前请求所连接的服务器端口号，即 HTTP 请求消息中 Host 头字段所对应的端口号部分
StringBuffer getRequestURL()	该方法用于获取客户端发出请求时的完整 URL，包括协议、服务器名、端口号、资源路径等信息，但不包括后面的查询参数部分。注意，getRequestURL()方法返回的结果是 StringBuffer 类型，而不是 String 类型，这样更便于对结果进行修改

表 10-1 列出了一系列用于获取请求消息行信息的方法，为了使读者更好地理解这些方法，下面通过案例演示这些方法的使用。

在 Eclipse 中创建一个名称为 chapter10 的项目，在 src 下新建一个名称为 com.itheima.request 的包，在包中创建一个名为 RequestLineServlet 的 Servlet 类，在该类中编写用于获取请求行中相关信息的方法，如文件 10-1 所示。

文件 10-1 RequestLineServlet.java

```
1  package com.itheima.request;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class RequestLineServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7          HttpServletResponse response)throws ServletException, IOException {
8          PrintWriter out=response.getWriter();
9          //获取请求行的相关信息
10         out.println("getMethod : "+request.getMethod()+"<br />");
11         out.println("getRequestURI : "+request.getRequestURI()+"<br />");
12         out.println("getQueryString:"+request.getQueryString()+"<br />");
13         out.println("getContextPath:"+request.getContextPath()+"<br />");
14         out.println("getServletPath:"+request.getServletPath()+"<br />");
15         out.println("getRemoteAddr : "+request.getRemoteAddr()+"<br />");
16         out.println("getRemoteHost : "+request.getRemoteHost()+"<br />");
17         out.println("getRemotePort : "+request.getRemotePort()+"<br />");
18         out.println("getLocalAddr : "+request.getLocalAddr()+"<br />");
19         out.println("getLocalName : "+request.getLocalName()+"<br />");
20         out.println("getLocalPort : "+request.getLocalPort()+"<br />");
21         out.println("getServerName : "+request.getServerName()+"<br />");
22         out.println("getServerPort : "+request.getServerPort()+"<br />");
23         out.println("getRequestURL : "+request.getRequestURL()+"<br />");
24     }
25     public void doPost(HttpServletRequest request,
26         HttpServletResponse response)throws ServletException, IOException {
27           doGet(request, response);
28     }
29 }
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter10/RequestLineServlet 访问 RequestLineServlet，浏览器的显示结果如图 10-2 所示。

从图 10-2 中可以看出，浏览器显示出了请求 RequestLineServlet 时发送的请求行信息。由此可见，通过 HttpServletRequest 对象可以很方便地获取请求行的相关信息。

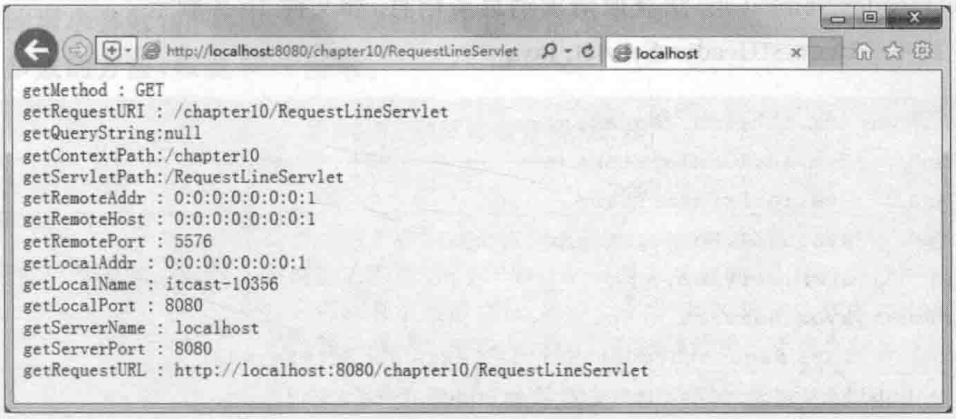


图 10-2　运行结果

10.2.2　获取请求消息头的相关方法

当浏览器发送 Servlet 请求时，需要通过请求消息头向服务器传递附加信息，例如，客户端可以接收的数据类型、压缩方式、语言等。为此，在 HttpServletRequest 接口中定义了一系列用于获取 HTTP 请求头字段的方法，如表 10-2 所示。

表 10-2　获取请求消息头的方法

方法声明	功能描述
String getHeader(String name)	该方法用于获取一个指定头字段的值，如果请求消息中没有包含指定的头字段，则 getHeader()方法返回 null；如果请求消息中包含多个指定名称的头字段，则 getHeader()方法返回其中第一个头字段的值
Enumeration getHeaders(String name)	该方法返回一个 Enumeration 集合对象，该集合对象由请求消息中出现的某个指定名称的所有头字段值组成。在多数情况下，一个头字段名在请求消息中只出现一次，但有时可能会出现多次
Enumeration getHeaderNames()	该方法用于获取一个包含所有请求头字段的 Enumeration 对象
int getIntHeader(String name)	该方法用于获取指定名称的头字段，并且将其值转为 int 类型。需要注意的是，如果指定名称的头字段不存在，则返回值为−1；如果获取到的头字段的值不能转为 int 类型，则将发生 NumberFormatException 异常
long getDateHeader(String name)	该方法用于获取指定头字段的值，并将其按 GMT 时间格式转换为一个代表日期/时间的长整数，该长整数是自 1970 年 1 月 1 日 0 时 0 分 0 秒算起的以毫秒为单位的时间值
String getContentType()	该方法用于获取 Content-Type 头字段的值，结果为 String 类型
int getContentLength()	该方法用于获取 Content-Length 头字段的值，结果为 int 类型
String getCharacterEncoding()	该方法用于返回请求消息的实体部分的字符集编码，通常是从 Content-Type 头字段中进行提取，结果为 String 类型

表 10-2 列出了一系列用于读取 HTTP 请求消息头字段的方法，为了更好地掌握这些方法，下面通过案例学习这些方法的使用。

在 com.itheima.request 包中创建一个名为 RequestHeadersServlet 的 Servlet 类，该类

中使用 getHeaderNames()方法获取请求消息头信息，如文件 10-2 所示。

文件 10-2 RequestHeadersServlet.java

```
1  package com.itheima.request;
2  import java.io.IOException;
3  import java.io.PrintWriter;
4  import java.util.Enumeration;
5  import javax.servlet.*;
6  import javax.servlet.http.*;
7  public class RequestHeadersServlet extends HttpServlet {
8      public void doGet(HttpServletRequest request,
9          HttpServletResponse response)throws ServletException, IOException {
10           PrintWriter out=response.getWriter();
11           //获取请求消息中的所有头字段
12           Enumeration headerNames=request.getHeaderNames();
13           //使用循环遍历所有请求头,并通过 getHeader()方法获取一个指定名称的头字段
14           while(headerNames.hasMoreElements()) {
15               String headerName=(String) headerNames.nextElement();
16               out.print(headerName+" : "
17                   +request.getHeader(headerName)+"<br />");
18         }
19     }
20     public void doPost(HttpServletRequest request,
21         HttpServletResponse response)throws ServletException, IOException {
22             doGet(request, response);
23     }
24 }
```

启动 Tomcat 服务器，在浏览器的地址栏中输入 http://localhost:8080/chapter10/RequestHeadersServlet 访问 RequestHeadersServlet，浏览器的显示结果如图 10-3 所示。

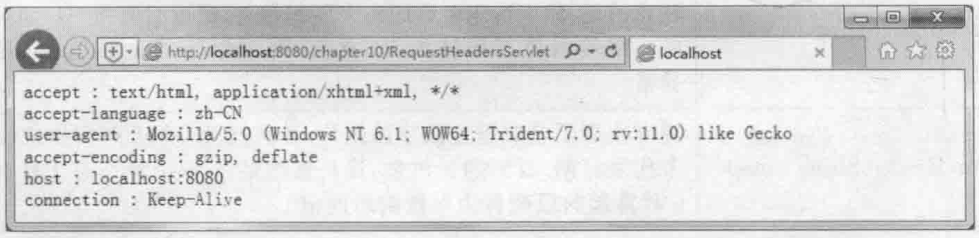


图 10-3 运行结果

10.3 HttpServletRequest 应用

10.3.1 获取请求参数

在实际开发中，经常需要获取用户提交的表单数据，例如用户名和密码等，为了方便获

取表单中的请求参数，在 HttpServletRequest 接口的父类 ServletRequest 中定义了一系列获取请求参数的方法，如表 10-3 所示。

表 10-3　获取请求参数的方法

方法声明	功能描述
String getParameter(String name)	该方法用于获取某个指定名称的参数值，如果请求消息中没有包含指定名称的参数，则 getParameter()方法返回 null；如果指定名称的参数存在但没有设置值，则返回一个空串；如果请求消息中包含多个该指定名称的参数，则 getParameter()方法返回第一个出现的参数值
String[] getParameterValues(String name)	HTTP 请求消息中可以有多个相同名称的参数(通常由一个包含多个同名的字段元素的 form 表单生成)，如果要获得 HTTP 请求消息中的同一个参数名所对应的所有参数值，那么就应该使用 getParameterValues()方法，该方法用于返回一个 String 类型的数组
Enumeration getParameterNames()	getParameterNames()方法用于返回一个包含请求消息中所有参数名的 Enumeration 对象，在此基础上，可以对请求消息中的所有参数进行遍历处理
Map getParameterMap()	getParameterMap()方法用于将请求消息中的所有参数名和值装入一个 Map 对象中返回

在表 10-3 中，getParameter()方法用于获取某个指定的参数，而 getParameterValues()方法用于获取多个同名的参数。下面通过具体的案例讲解这两个方法的使用：

在 chapter10 项目的 WebContent 根目录下创建一个表单文件 form.html，编辑后如文件 10-3 所示。

文件 10-3　form.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset=" UTF-8">
<title>Insert title here</title>
</head>
<body>
    <form action="/chapter10/RequestParamsServlet" method="POST">
        用户名：<input type="text" name="username"><br />
        密     码：<input type="password" name="password">
        <br />
        爱好：
        <input type="checkbox" name="hobby" value="sing">唱歌
        <input type="checkbox" name="hobby" value="dance">跳舞
        <input type="checkbox" name="hobby" value="game">玩游戏<br />
        <input type="submit" value="提交">
    </form>
</body>
</html>
```

在 com.itheima.request 包中编写一个名称为 RequestParamsServlet 的 Servlet 类，使用该 Servlet 获取请求参数，如文件 10-4 所示。

文件 10-4 RequestParamsServlet.java

```
1  package com.itheima.request;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class RequestParamsServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7          HttpServletResponse response)throws ServletException, IOException {
8          String name=request.getParameter("username");
9          String password=request.getParameter("password");
10         System.out.println("用户名:"+name);
11         System.out.println("密 码:"+password);
12         //获取参数名为"hobby"的值
13         String[] hobbys=request.getParameterValues("hobby");
14         System.out.print("爱好:");
15         for (int i=0; i <hobbys.length; i++) {
16             System.out.print(hobbys[i]+", ");
17         }
18     }
19     public void doPost(HttpServletRequest request,
20         HttpServletResponse response)throws ServletException, IOException {
21           doGet(request, response);
22     }
23 }
```

在文件 10-4 中，由于参数名为 hobby 的值可能有多个，因此，需要使用 getParameterValues()方法获取多个同名参数的值，返回一个 String 类型的数组，通过遍历数组，输出每个 hobby 参数对应的值。

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter10/form.html 访问 form.html 页面，并填写表单的相关信息，填写后的页面如图 10-4 所示。

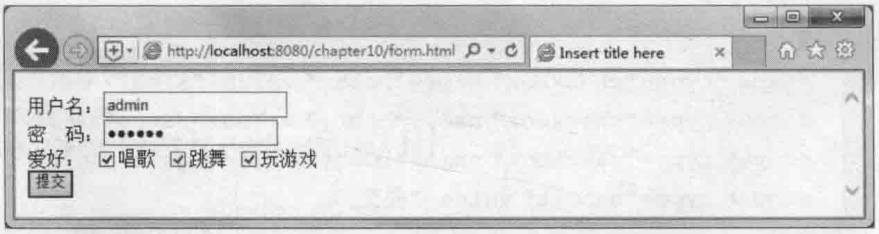


图 10-4 运行结果

单击图 10-4 中的提交按钮，在 Eclipse 的控制台输出了每个参数的信息，如图 10-5 所示。

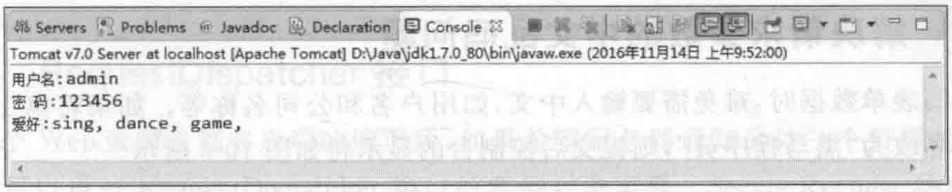


图 10-5　运行结果

10.3.2　通过 Request 对象传递数据

Request 对象不仅可以获取一系列数据，还可以通过属性传递数据。ServletRequest 接口中定义了一系列操作属性的方法。

1. setAttribute()方法

该方法用于将一个对象与一个名称关联后存储到 ServletRequest 对象中，其完整语法定义如下：

```
public void setAttribute(java.lang.String name,java.lang.Object o);
```

需要注意的是，如果 ServletRequest 对象中已经存在指定名称的属性，则 setAttribute()方法将会先删除原来的属性，然后再添加新的属性。如果传递给 setAttribute()方法的属性值对象为 null，则删除指定名称的属性，这时的效果等同于 removeAttribute()方法。

2. getAttribute()方法

该方法用于从 ServletRequest 对象中返回指定名称的属性对象，其完整的语法定义如下：

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

3. removeAttribute()方法

该方法用于从 ServletRequest 对象中删除指定名称的属性，其完整的语法定义如下：

```
public void removeAttribute(java.lang.String name);
```

4. getAttributeNames()方法

该方法用于返回一个包含 ServletRequest 对象中的所有属性名的 Enumeration 对象，在此基础上，可以对 ServletRequest 对象中的所有属性进行遍历处理。getAttributeNames()方法的完整语法定义如下：

```
public java.util.Enumeration getAttributeNames();
```

需要注意的是，只有属于同一个请求中的数据才可以通过 ServletRequest 对象传递数

据。关于 ServletRequest 对象操作属性的具体用法，本书将在后续章节进行详细讲解，在此读者只需了解即可。

10.3.3 解决请求参数的中文乱码问题

在填写表单数据时，难免需要输入中文，如用户名和公司名称等。如果将第 10.3.1 节中的用户名改为“黑马程序员”，则提交后控制台的显示将如图 10-6 所示。

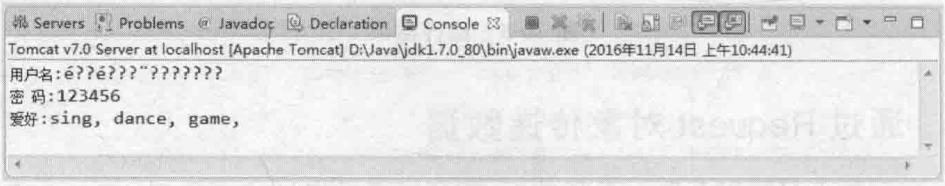


图 10-6 运行结果

从图 10-6 中可以看出，当输入的用户名为中文时，出现了“é??é??...”的乱码。本节将介绍如何处理请求参数中的中文乱码。

在 HttpServletRequest 接口中提供了一个 setCharacterEncoding()方法，该方法用于设置 request 对象的解码方式。接下来对文件 10-4 进行修改，在第 7～8 行代码之间添加如下代码：

```
request.setCharacterEncoding("utf-8");        //设置 request 对象的解码方式
```

重启 Tomcat 服务器，再次输入中文“黑马程序员”并提交表单信息后，控制台显示的信息如图 10-7 所示。

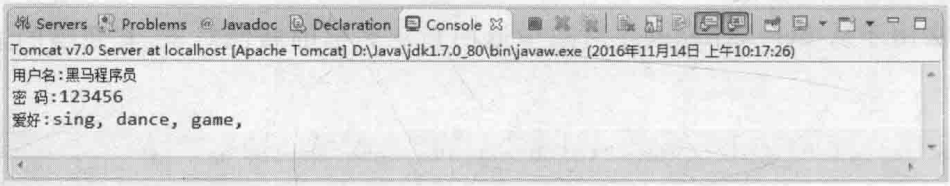


图 10-7 运行结果

需要注意的是，这种解决乱码的方式只对 POST 方式有效，而对 GET 方式无效。如果将 form.html 文件中 method 属性的值改为 GET，重新访问 form.html 页面并填写中文信息，则控制台依然会出现如图 10-6 所示的乱码问题。

为了解决 GET 方式提交表单时出现的中文乱码问题，可以先使用错误码表 ISO-8859-1 将用户名重新编码，然后使用码表 UTF-8 进行解码。再次对文件 10-4 进行修改，在第 9～10 行代码之间增加一行代码，如下所示：

```
name=new String(name.getBytes("iso8859-1"),"utf-8");
```

重启 Tomcat 服务器，再次访问 form.html 网页，输入中文用户名“黑马程序员”，这时，控制台显示的信息将不会出现乱码。

10.4　RequestDispatcher 对象的应用

10.4.1　RequestDispatcher 接口

当一个 Web 资源收到客户端的请求后，如果希望服务器通知另外一个资源处理请求，那么这时可以通过 RequestDispatcher 接口的实例对象实现。ServletRequest 接口中定义了一个获取 RequestDispatcher 对象的方法，如表 10-4 所示。

表 10-4　获取 RequestDispatcher 对象的方法

方法声明	功能描述
RequestDispatcher getRequestDispatcher (String path)	返回封装了某条路径所指定资源的 RequestDispatcher 对象。其中，参数 path 必须以/开头，用于表示当前 Web 应用的根目录。需要注意的是，WEB-INF 目录中的内容对 RequestDispatcher 对象也是可见的，因此，传递给 getRequestDispatcher(String path) 方法的资源可以是 WEB-INF 目录中的文件

获取到 RequestDispatcher 对象后，最重要的工作就是通知其他 Web 资源处理当前的 Servlet 请求，为此，RequestDispatcher 接口定义了两个相关方法，如表 10-5 所示。

表 10-5　RequestDispatcher 接口的方法

方法声明	功能描述
forward(ServletRequest request, ServletResponse response)	该方法用于将请求从一个 Servlet 传递给另一个 Web 资源。在 Servlet 中，可以对请求做一个初步处理，然后通过调用这个方法，将请求传递给其他资源进行响应。需要注意的是，该方法必须在响应提交给客户端之前被调用，否则将抛出 IllegalStateException 异常
include(ServletRequest request, ServletResponse response)	该方法用于将其他的资源作为当前响应内容包含进来

10.4.2　请求转发

在 RequestDispatcher 接口中，forward()方法可以实现请求转发，include()方法可以实现请求包含，本节将以请求转发为例，讲解 forward()方法的使用。

在 Servlet 中，如果当前 Web 资源不想处理请求，则可以通过 forward()方法将当前请求传递给其他的 Web 资源进行处理，这种方式称为请求转发。请求转发的工作原理如图 10-8 所示。

从图 10-8 中可以看出，当客户端访问 Servlet1 时，可以通过 forward()方法将请求转发给其他 Web 资源，其他 Web 资源处理完请求后，直接将响应结果返回到客户端。

了解了 forward()方法的工作原理后，下面通过案例演示 forward()方法的使用。在 chapter10 项目的 com. itheima. request 包中创建一个名为 RequestForwardServlet 的 Servlet 类，该类使用 forword()方法将请求转发到一个新的 Servlet 页面，如文件 10-5 所示。

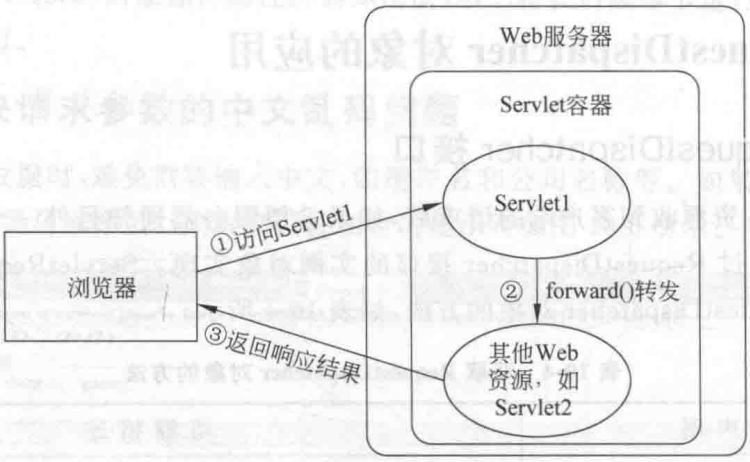


图 10-8 forward()方法的工作原理

文件 10-5 RequestForwardServlet.java

```
1  package com.itheima.request;
2  import java.io.IOException;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class RequestForwardServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7        HttpServletResponse response)throws ServletException, IOException {
8          response.setContentType("text/html;charset=utf-8");
9          request.setAttribute("username", "张三");      //将数据存储到 request 对象中
10         RequestDispatcher dispatcher=request
11                 .getRequestDispatcher("/ResultServlet");
12         dispatcher.forward(request, response);
13     }
14     public void doPost(HttpServletRequest request,
15       HttpServletResponse response)throws ServletException, IOException {
16         doGet(request, response);
17     }
18 }
```

在文件 10-5 中，通过使用 forward()方法，将当前 Servlet 的请求转发到 ResultServlet 页面，在 com.itheima.request 包中编写一个名为 ResultServlet 的 Servlet 类，该类用于获取 RequestForwardServlet 类中存储在 request 对象中的数据并输出，ResultServlet 类的代码实现如文件 10-6 所示。

文件 10-6 ResultServlet.java

```
1  package com.itheima.request;
2  import java.io.*;
3  import javax.servlet.*;
```

```
4  import javax.servlet.http.*;
5  public class ResultServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7        HttpServletResponse response)throws ServletException, IOException {
8          response.setContentType("text/html;charset=utf-8");
9          PrintWriter out=response.getWriter();   //获取 PrintWriter 对象用于输出信息
10         String username=(String) request.getAttribute("username");
11         if (username!=null) {
12             out.println("用户名: "+username+"<br />");
13         }
14     }
15     public void doPost(HttpServletRequest request,
16       HttpServletResponse response)throws ServletException, IOException {
17         doGet(request, response);
18     }
19 }
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter10/RequestForwardServlet 访问 RequestForwardServlet，浏览器的显示结果如图 10-9 所示。

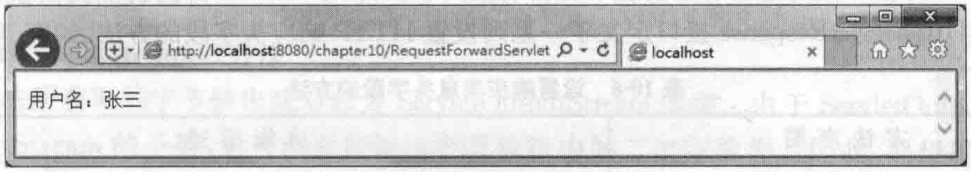


图 10-9　运行结果

从图 10-9 中可以看出，地址栏中显示的仍然是 RequestForwardServlet 的请求路径，但是浏览器却显示出了 ResultServlet 中要输出的内容。这是因为请求转发是发生在服务器内部的行为，从 RequestForwardServlet 到 ResultServlet 属于一次请求，在一次请求中是可以使用 request 属性进行数据共享的。

10.5　HttpServletResponse 对象

HttpServletResponse 接口继承自 ServletResponse 接口，主要用于封装 HTTP 响应消息。由于 HTTP 响应消息分为状态行、响应消息头、消息体三部分，因此，在 HttpServletResponse 接口中定义了向客户端发送响应状态码、响应消息头、响应消息体的方法，本节将针对这些方法进行详细讲解。

10.5.1　发送状态码相关的方法

当 Servlet 向客户端回送响应消息时，需要在响应消息中设置状态码。因此，HttpServletResponse 接口定义了两个发送状态码的方法。

1. setStatus(int status)方法

该方法用于设置 HTTP 响应消息的状态码，并生成响应状态行。由于响应状态行中的状态描述信息直接与状态码相关，而 HTTP 版本由服务器确定，因此，只要通过 setStatus(int status)方法设置了状态码，即可实现状态行的发送。需要注意的是，在正常情况下，Web 服务器会默认产生一个状态码为 200 的状态行。

2. sendError(int sc)方法

该方法用于发送表示错误信息的状态码，例如，404 状态码表示找不到客户端请求的资源。response 对象提供了两个重载的 sendError(int sc)方法，具体如下：

```
public void sendError(int code) throws java.io.IOException
public void sendError(int code, String message) throws java.io.IOException
```

在上面重载的两个方法中，第一个方法只发送错误信息的状态码，而第二个方法除了发送状态码以外，还可以增加一条用于提示说明的文本信息，该文本信息将出现在发送给客户端的正文内容中。

10.5.2 发送响应消息头相关的方法

Servlet 向客户端发送的响应消息中包含响应头字段，由于 HTTP 协议的响应头字段有很多种，因此，HttpServletResponse 接口定义了一系列设置 HTTP 响应头字段的方法，如表 10-6 所示。

表 10-6 设置响应消息头字段的方法

方法声明	功能描述
void addHeader(String name, String value)	这两个方法都是用于设置 HTTP 协议的响应头字段，其中，参数 name 用于指定响应头字段的名称，参数 value 用于指定响应头字段的值。不同的是，addHeader()方法可以增加同名的响应头字段，而 setHeader()方法则会覆盖同名的头字段
void setHeader(String name, String value)	
void addIntHeader(String name,int value)	这两个方法专门用于设置包含整数值的响应头，避免了使用 addHeader()与 setHeader()方法时需要将 int 类型的设置值转换为 String 类型的麻烦
void setIntHeader(String name,int value)	
void setContentType(String type)	该方法用于设置 Servlet 输出内容的 MIME 类型，对于 HTTP 协议来说，就是设置 Content-Type 响应头字段的值。例如，如果发送到客户端的内容是 jpeg 格式的图像数据，就需要将响应头字段的类型设置为 image/jpeg。需要注意的是，如果响应的内容为文本，setContentType()方法还可以设置字符编码，如 text/html;charset=UTF-8
void setLocale(Locale loc)	该方法用于设置响应消息的本地化信息。对 HTTP 来说，就是设置 Content-Language 响应头字段和 Content-Type 头字段中的字符集编码部分。需要注意的是，如果 HTTP 消息没有设置 Content-Type 头字段，则 setLocale()方法设置的字符集编码不会出现在 HTTP 消息的响应头中，如果调用 setCharacterEncoding()或 setContentType()方法指定了响应内容的字符集编码，则 setLocale()方法将不再具有指定字符集编码的功能

续表

方法声明	功能描述
void setCharacterEncoding(String charset)	该方法用于设置输出内容使用的字符编码，对 HTTP 协议来说，就是设置 Content-Type 头字段中的字符集编码部分。如果没有设置 Content-Type 头字段，则 setCharacterEncoding 方法设置的字符集编码不会出现在 HTTP 消息的响应头中。setCharacterEncoding()方法比 setContentType()和 setLocale()方法的优先权高，它的设置结果将覆盖 setContentType()和 setLocale()方法所设置的字符码表

需要注意的是，在表 10-6 列举的一系列方法中，addHeader()、setHeader()、addIntHeader()、setIntHeader()方法都用于设置各种头字段，而 setContetType()、setLoacale()和 setCharacterEncoding()方法用于设置字符编码，这些设置字符编码的方法可以有效解决乱码问题。

10.5.3　发送响应消息体相关的方法

由于在 HTTP 响应消息中，大量的数据都是通过响应消息体传递的，因此，ServletResponse 遵循以 I/O 流传递大量数据的设计理念。在发送响应消息体时，定义了两个与输出流相关的方法。

1. getOutputStream()方法

该方法所获取的字节输出流对象为 ServletOutputStream 类型。由于 ServletOutputStream 是 OutputStream 的子类，它可以直接输出字节数组中的二进制数据。因此，要想输出二进制格式的响应正文，就需要使用 getOutputStream()方法。

2. getWriter()方法

该方法所获取的字符输出流对象为 PrintWriter 类型。由于 PrintWriter 类型的对象可以直接输出字符文本内容，因此，要想输出内容全部为字符文本的网页文档，则需要使用 getWriter()方法。

注意：虽然 response 对象的 getOutputStream()和 getWriter()方法都可以发送响应消息体，但是，它们之间互相排斥，不可同时使用，否则会发生 IllegalStateException 异常。

10.6　HttpServletResponse 应用

10.6.1　解决中文输出乱码问题

由于计算机中的数据都是以二进制形式存储的，因此，当传输文本数据时，会发生字符和字节之间的转换。字符与字节之间的转换是通过查码表完成的，将字符转换成字节的过程称为编码，将字节转换成字符的过程称为解码，如果编码和解码使用的码表不一致，则会导致乱码问题。下面通过案例演示乱码问题的产生原因以及解决方式。

在 chapter10 项目中新建一个名称为 com.itheima.response 的包，在该包中新建一个名为 ChineseServlet 的类，在类中定义一个中文字符串，然后使用字符输出流输出，如文

件 10-7 所示。

文件 10-7 ChineseServlet.java

```
1  package com.itheima.response;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class ChineseServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7        HttpServletResponse response)throws ServletException, IOException {
8          String data="中国";
9          PrintWriter out=response.getWriter();
10         out.println(data);
11     }
12     public void doPost(HttpServletRequest request,
13       HttpServletResponse response)throws ServletException, IOException {
14         doGet(request, response);
15     }
16 }
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter10/ChineseServlet 访问 ChineseServlet，浏览器的显示结果如图 10-10 所示。



图 10-10 运行结果

从图 10-10 中可以看出，浏览器显示的内容是??，说明发生了乱码。实际上此处产生乱码的原因是 response 对象的字符输出流在编码时采用的字符码表是 ISO-8859-1，该码表不兼容中文，会将“中国”编码为 63 63（在 ISO-8859-1 的码表中查不到的字符就会显示 63）。当浏览器对接收到的数据进行解码时，会采用默认的码表 GB2312，将 63 解码为?，因此，浏览器将“中国”两个字符显示为??，具体分析如图 10-11 所示。



图 10-11 编码错误分析

为了解决上述编码错误，HttpServletResponse 对象提供了两种解决乱码的方式，具体如下。

第一种方式：

```
response.setCharacterEncoding("utf-8");        //设置 HttpServletResponse 使用 utf-8 编码
response.setHeader("Content-Type","text/html;charset=utf-8");
                                                //通知浏览器使用 utf-8 解码
```

第二种方式：

```
response.setContentType("text/html;charset=utf-8");  //包含第一种方式的两个功能
```

在通常情况下，为了使代码更加简洁，一般会采用第二种方式。接下来对文件 10-7 进行修改，在第 7～8 行代码之间加入第二种方式的代码，重新启动 Tomcat 服务器并使用浏览器访问 ChineseServlet，浏览器显示出了正确的中文字符，如图 10-12 所示。



图 10-12　运行结果

10.6.2　实现请求重定向

在某些情况下，针对客户端的请求，一个 Servlet 类可能无法完成全部工作。这时，可以使用请求重定向完成这一工作。请求重定向指 Web 服务器接收到客户端的请求后，可能由于某些条件的限制，不能访问当前请求 URL 所指向的 Web 资源，而是指定了一个新的资源路径，让客户端重新发送请求。

为了实现请求重定向，HttpServletResponse 接口定义了一个 sendRedirect()方法，该方法用于生成 302 响应码和 Location 响应头，从而通知客户端重新访问 Location 响应头中指定的 URL，sendRedirect()方法的完整语法如下所示：

```
public void sendRedirect(java.lang.String location) throws java.io.IOException
```

在上述方法代码中，参数 location 可以使用相对 URL，Web 服务器会自动将相对 URL 翻译成绝对 URL，再生成 Location 头字段。

sendRedirect()方法的工作原理如图 10-13 所示。

在图 10-13 中，当客户端访问 Servlet1 时，由于在 Servlet1 中调用了 sendRedirect()方法将请求重定向到 Servlet2，因此，浏览器收到 Servlet1 的响应消息后，立刻向 Servlet2 发送请求，Servlet2 对请求处理完毕后，再将响应消息回送给客户端浏览器并显示。

下面通过一个用户登录的案例分步骤讲解 sendRedirect()方法的使用。

1. 创建页面文件

在 chapter10 项目的 WebContent 目录下创建一个用户登录的页面 login.html 和登录成功的页面 welcome.html，编辑后如文件 10-8 和文件 10-9 所示。

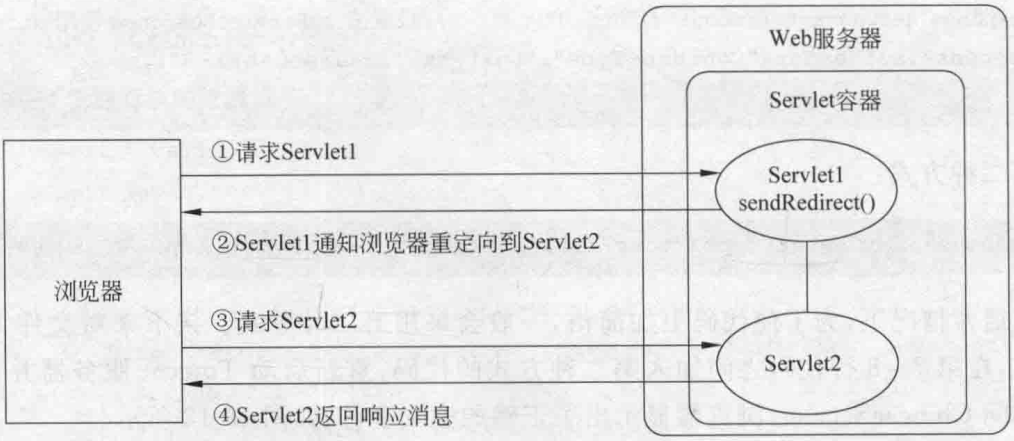


图 10-13 sendRedirect()方法的工作原理

文件 10-8 login.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>用户登录</title>
</head>
<body>
    <!--把表单内容提交到 chapter10 工程下的 LoginServlet-->
    <form action="/chapter10/LoginServlet" method="post">
    用户名：<input type="text" name="username" /><br />
    密    码：<input type="password" name="password"/><br />
        <input type="submit" value="登录" />
    </form>
</body>
</html>
```

文件 10-9 welcome.html

```
<!DOCTYPE HTML>
<html>
<head>
<meta charset="UTF-8">
<title>欢迎页面</title>
</head>
<body>
    欢迎你,登录成功!
</body>
</html>
```

2. 创建 Servlet

在 chapter10 项目的 com.itheima.response 包中创建一个名为 LoginServlet 的 Servlet 类，用于处理用户登录请求，如文件 10-10 所示。

文件 10-10　LoginServlet.java

```
1  package com.itheima.response;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class LoginServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7        HttpServletResponse response)throws ServletException, IOException {
8          response.setContentType("text/html;charset=utf-8");
9          //用 HttpServletRequest 对象的 getParameter()方法获取用户名和密码
10         String username=request.getParameter("username");
11         String password=request.getParameter("password");
12         //假设用户名和密码分别为 admin 和 123456
13         if (("admin").equals(username) &&("123456").equals(password)) {
14             //如果用户名和密码正确，重定向到 welcome.html
15              response.sendRedirect("/chapter10/welcome.html");
16         } else {
17             //如果用户名和密码错误，重定向到 login.html
18             response.sendRedirect("/chapter10/login.html");
19         }
20     }
21     public void doPost(HttpServletRequest request,
22       HttpServletResponse response)throws ServletException, IOException {
23         doGet(request, response);
24     }
25 }
```

在文件 10-10 中，首先通过 getParameter()方法分别获取用户名和密码，然后判断表单中输入的用户名和密码是否为指定的“admin”和“123456”，如果是，则将请求重定向到 welcome.html 页面，否则重定向到 login.html 页面。

3. 运行项目并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter10/login.html 访问 login.html，浏览器的显示结果如图 10-14 所示。

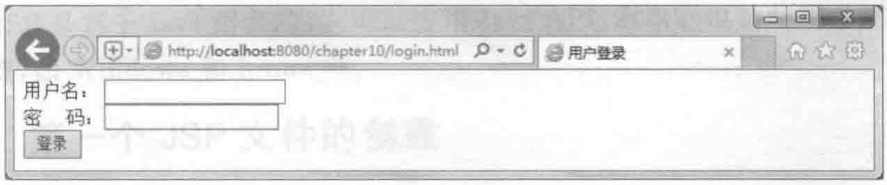


图 10-14　运行结果

在图 10-14 所示的界面中填写用户名“admin”和密码“123456”，单击【登录】按钮，浏览器的显示结果如图 10-15 所示。

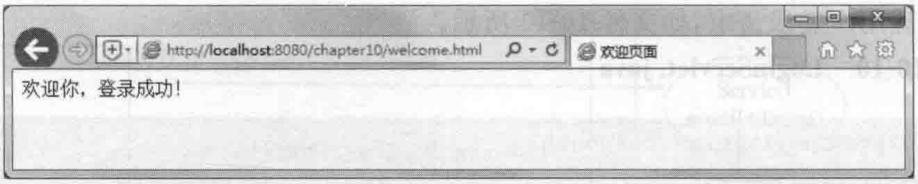


图 10-15　运行结果

从图 10-15 中可以看出，当用户名和密码输入正确后，浏览器跳转到了 welcome. html 页面。但是，如果用户名或者密码输入错误，则会跳转到图 10-14 所示的登录页面。

第 11 章 JSP 技 术

在动态 Web 项目的开发中，经常需要动态生成 HTML 内容(如系统中的当前在线人数需要动态生成)。如果使用 Servlet 实现 HTML 页面数据的统计，则需要使用大量的输出语句。同时，如果静态内容和动态内容混合在一起，那么也将导致程序非常臃肿。为了克服 Servlet 的这些缺点，Oracle(Sun)公司推出了 JSP 技术。本章将围绕 JSP 技术进行详细讲解。

11.1 JSP 概述

11.1.1 JSP

JSP(Java Server Pages)是建立在 Servlet 规范之上的动态网页开发技术，其实质是一个简化的 Servlet。在 JSP 文件中，HTML 代码与 Java 代码共同存在，其中，HTML 代码用于实现网页中静态内容的显示，Java 代码用于实现网页中动态内容的显示。为了与传统 HTML 有所区别，JSP 文件的扩展名为 jsp。

JSP 技术所开发的 Web 应用程序是基于 Java 的，其使用具有如下几点特征。

1. 预编译

预编译指在用户第一次通过浏览器访问 JSP 页面时，服务器将对 JSP 页面代码进行编译，并且仅执行一次编译。编译好的代码将被保存，在用户下一次访问时会直接执行编译好的代码。这样不仅节约了服务器的 CPU 资源，还大幅度提升了客户端的访问速度。

2. 业务代码相分离

在使用 JSP 技术开发 Web 应用时，可以将界面的开发与应用程序的开发分离。

3. 组件重用

JSP 可以使用 JavaBean 编写业务组件，也就是使用一个 JavaBean 类封装业务处理代码或者将其作为一个数据存储模型，在 JSP 页面甚至整个项目中，都可以重复使用这个 JavaBean，同时，JavaBean 也可以应用到其他 Java 应用程序中。

4. 跨平台

由于 JSP 是基于 Java 语言的，它可以使用 Java API，所以它也是跨平台的，可以应用于不同的系统，如 Windows 和 Linux 等。

11.1.2 第一个 JSP 文件的创建

在 Eclipse 中创建一个名称为 chapter11 的 Web 项目，然后右击 WebContent 目录→

New→Other，在弹出的窗口中找到JSP文件，选择JSP File后，单击Next按钮，在新窗口的File name文本框中填写JSP文件名称hello.jsp，单击Next按钮，进入选择模板窗口，如图11-1所示。

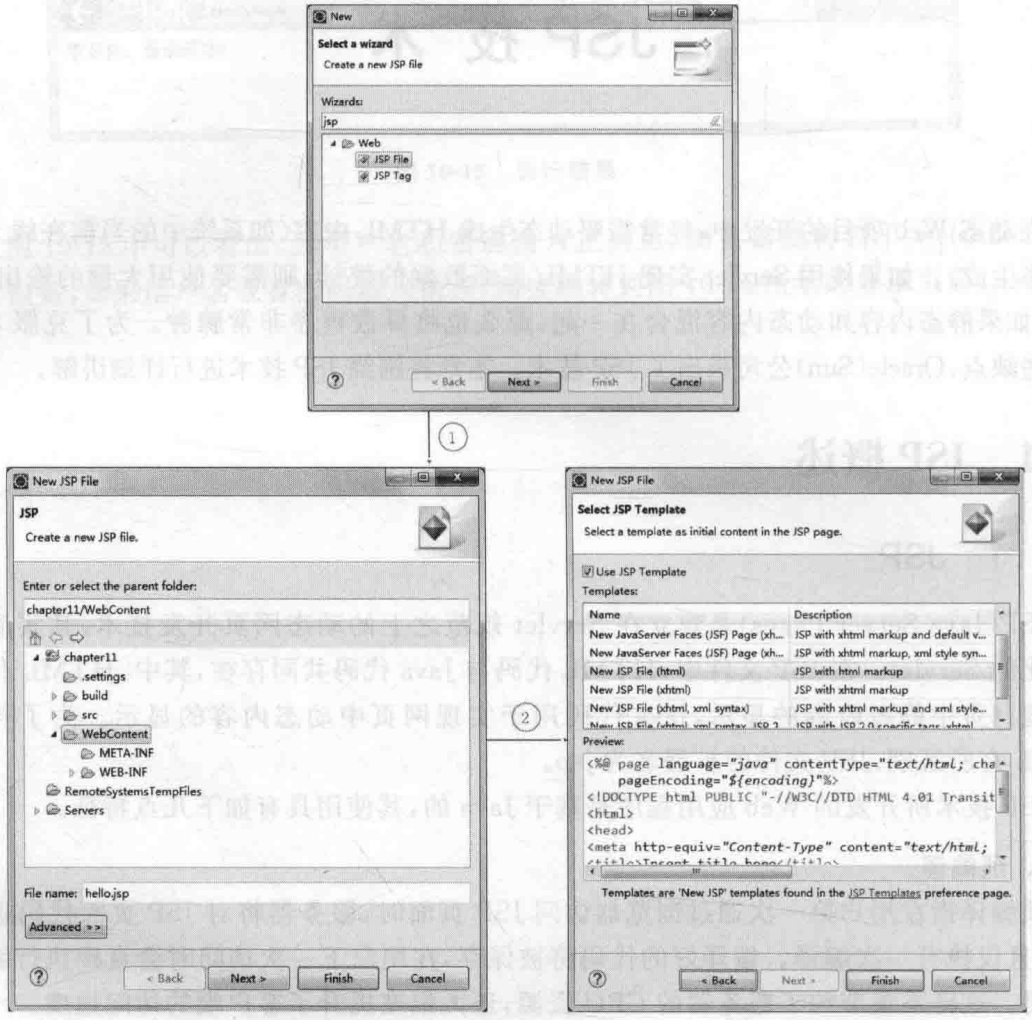


图11-1 创建JSP文件

此处选择默认的模板，单击图11-1中的Finish按钮后，第一个JSP文件就创建成功了。创建后的JSP文件如图11-2所示。

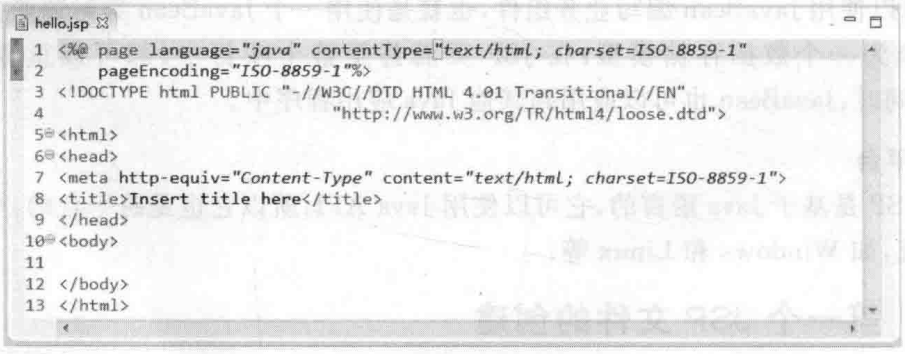


图11-2 hello.jsp文件

从图 11-2 中可以看出，新创建的 JSP 文件与传统的 HTML 文件类似，主要的区别在于页面代码最上方增加了一条 page 指令，并且该文件的扩展名是 jsp，而不是 html。关于 page 指令的相关内容会在第 11.3.1 节中详细讲解，此处了解即可。

JSP 文件必须发布到 Web 容器的某个 Web 应用中才能查看出效果。在 hello.jsp 的＜body＞元素内添加内容“This is My First JSP!”并保存后，将 chapter11 项目发布到 Tomcat 中并启动。在浏览器的地址栏中输入地址 http://localhost:8080/chapter11/hello.jsp，此时浏览器的显示效果如图 11-3 所示。

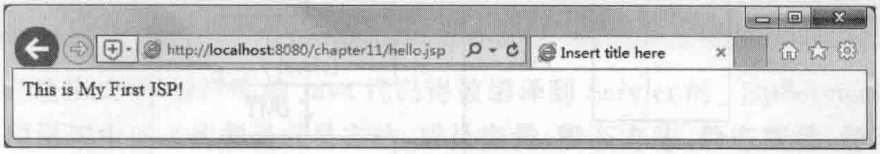


图 11-3　hello.jsp 文件显示效果

从图 11-3 中可以看出，hello.jsp 中添加的内容已被显示出来，这说明了 HTML 中的元素可以被 JSP 容器解析。实际上，JSP 只是在原有的 HTML 文件中加入了一些具有 Java 特点的代码，这些代码具有其独有的特点，称为 JSP 的语法元素。

需要注意的是，在 hello.jsp 中，＜body＞元素内所添加的内容必须是在英文状态下输入的，如果是在中文状态下输入的文字或符号，在保存时会出现不能保存的问题。出现此问题的原因是 JSP 页面中的编码方式是 ISO-8859-1，而此种编码不支持中文。为了解决此问题，可以将 JSP 页面中的编码统一修改为 UTF-8，修改后即可在页面中保存中文，并可以在浏览器中正常显示。本书后续所有的 JSP 页面编码都将统一为 UTF-8。

11.1.3　JSP 运行原理

JSP 的工作模式是请求/响应模式，客户端首先发出 HTTP 请求，JSP 程序收到请求后将进行处理并返回处理结果。在一个 JSP 文件第一次被请求时，JSP 引擎（容器）把该 JSP 文件转换成一个 Servlet，而这个引擎本身也是一个 Servlet。JSP 的运行原理如图 11-4 所示。

JSP 的运行过程具体如下。

（1）客户端发出请求，请求访问 JSP 文件。

（2）JSP 容器先将 JSP 文件转换成一个 Java 源文件（Java Servlet 源程序），在转换过程中，如果发现 JSP 文件中存在任何语法错误，则中断转换过程，并向服务端和客户端返回出错信息。

（3）如果转换成功，则 JSP 容器会将生成的 Java 源文件编译成相应的字节码文件 *.class。该 class 文件就是一个 Servlet，Servlet 容器会像处理其他 Servlet 一样处理它。

（4）由 Servlet 容器加载转换后的 Servlet 类（class 文件）创建一个该 Servlet（JSP 页面的转换结果）的实例，并执行 Servlet 的 jspInit()方法。jspInit()方法在 Servlet 的整个生命周期中只会执行一次。

（5）执行 jspService()方法处理客户端的请求。对于每一个请求，JSP 容器都会创建一个新的线程处理它。如果多个客户端同时请求该 JSP 文件，则 JSP 容器会创建多个线程，

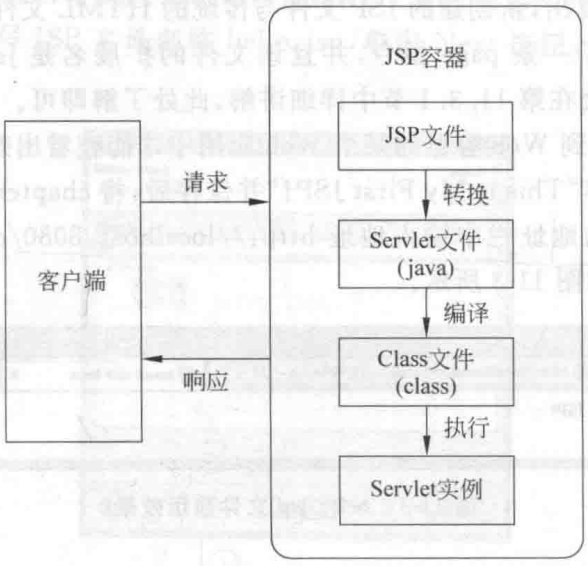


图 11-4 JSP 的运行原理

使每一个客户端请求都对应一个线程。

(6) 如果 JSP 文件被修改了，则服务器将根据设置决定是否对该文件重新进行编译，如果需要重新编译，则使用重新编译后的结果取代内存中的 Servlet，并继续上述处理过程。需要注意的是，虽然 JSP 效率很高，但在第一次调用时往往由于需要转换和编译，所以会产生一些轻微的延迟。

(7) 如果系统出现资源不足等问题，JSP 容器可能会以某种不确定的方式将 Servlet 从内存中移除，发生这种情况时，首先会调用 jspDestroy()方法，然后 Servlet 实例会被作为“垃圾”进行处理。

(8) 当请求处理完成后，响应对象由 JSP 容器接收，并将 HTML 格式的响应信息发送回客户端。

11.2 JSP 的脚本元素和注释

11.2.1 JSP 脚本元素

JSP 脚本元素是指嵌套在<%和%>中的一条或多条 Java 程序代码。通过 JSP 脚本元素可以将 Java 代码嵌入 HTML 页面中，所有可执行的 Java 代码都可以通过 JSP 脚本执行。

JSP 脚本元素主要包含如下三种类型：JSP Scriptlets 和 JSP 声明语句和 JSP 表达式，具体说明如下。

1. JSP Scriptlets

JSP Scriptlets 是一段代码段。当需要使用 Java 实现一些复杂操作或控制时就可以使用它。JSP Scriptlets 的语法格式如下所示：

```
<%java 代码(变量、方法、表达式等)%>
```

在 JSP Scriptlets 中声明的变量是 JSP 页面的局部变量，调用 JSP Scriptlets 时，会为局部变量分配内存空间，调用结束后，释放局部变量占用的内存空间。

2. JSP 声明语句

JSP 的声明语句用于声明变量和方法，它以＜%!开始，以%＞结束，其语法格式如下所示：

```
<%!
    定义的变量或方法等
%>
```

在上述语法格式中，被声明的 Java 代码将被编译到 Servlet 的_ jspService()方法之外，即在 JSP 声明语句中定义的都是成员方法、成员变量、静态方法、静态变量、静态代码块等。在 JSP 声明语句中声明的方法在整个 JSP 页面内有效，但是在方法内定义的变量只在该方法内有效。当声明的方法被调用时，会为方法内定的变量分配内存，而调用结束后会立刻释放所占的内存。

在一个 JSP 页面中可以有多条 JSP 声明语句，单个声明中的 Java 语句可以是不完整的，但是多个声明组合后的结果必须是完整的 Java 语句。

3. JSP 表达式

JSP 表达式(expression)用于将程序数据输出到客户端，它将要输出的变量或者表达式直接封装在以＜%＝开头和以%＞结尾的标记中，其基本的语法格式如下所示：

```
<%=expression %>
```

在上述语法格式中，JSP 表达式中的变量或表达式的计算结果将被转换成一个字符串，然后插入到 JSP 页面输出结果的相应位置。

需要注意的是，＜%＝和%＞标记之间插入的是表达式，不能插入语句。＜%＝是一个完整的符号，＜%和＝之间不能有空格，且 JSP 表达式中的变量和表达式后面不能有分号(;)。

下面通过案例演示 JSP 脚本元素的使用。在 chapter11 项目的 WebContent 目录下创建一个名称为 script.jsp 的文件，编辑后如文件 11-1 所示。

文件 11-1　script.jsp

```
1  <%@page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4                        "http://www.w3.org/TR/html4/loose.dtd">
5  <html>
6  <head>
7  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8  <title>JSP 声明语句</title>
9  </head>
10 <%!
```

```
11      int a=2, b=3;                          //定义两个变量 a,b
12  %>
13  <%!
14      public String print() {                //定义 print 方法
15          String str="www.baidu.com";        //方法内定义的变量 str
16          return str;
17      }
18  %>
19  <body>
20      <%
21          out.println(a+b);                  //输出两个变量的和
22      %>
23      <br />
24       <%=print() %>
25  </body>
26  </html>
```

在文件 11-1 中，首先使用<%!和%>定义了两个变量 a、b 以及 print()方法，然后分别使用了<%和%>输出了两个常量的和，并使用表达式输出了 print()方法中的返回信息。

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter11/script.jsp 访问 script.jsp 文件，显示效果如图 11-5 所示。

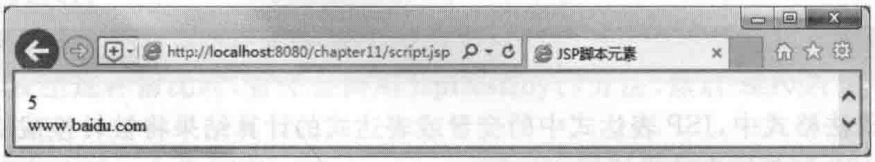


图 11-5 script.jsp 的执行结果

从图 11-5 中可以看到，浏览器中已经显示出了需要输出的结果。需要注意的是，<%!和%>里面定义的属性是成员属性，相当于类的属性，方法相当于全局的方法，也相当于类里面的方法，但是它不可以进行输出，因为它只进行方法的定义和属性的定义。

<%和%>可以进行属性的定义，也可以输出内容，但是它不可以进行方法的定义。因为这对标签里面的内容是在 JSP 被编译为 Servlet 时，放在_ jspService()方法里面的。这个方法就是服务器向客户端输出内容的地方，它本身就是一个方法，所以如果在它里面定义方法的话，那么就相当于在类的方法里面嵌套定义了方法，这在 Java 中是不被允许的。但是可以在里面定义自己的私有变量，因为方法里面也可以定义变量，也可以调用方法，唯独不可以再定义方法。

总之，<%!和%>用于定义成员变量属性和方法，<%和%>主要用于输出内容，因此如果涉及成员变量的操作，那么就应该使用<%!和%>，而如果涉及输出内容，那么就应该使用<%和%>。

11.2.2 JSP 注释

同其他编程语言一样，JSP 也有自己的注释方式，其基本语法格式如下：

```
<%--注释信息--%>
```

Tomcat 在将 JSP 页面编译成 Servlet 程序时，会忽略 JSP 页面中被注释的内容，因此不会将注释信息发送到客户端。

11.3　JSP 指令

为了设置 JSP 页面中的一些信息，Sun 公司提供了不同的 JSP 指令，其中较为常用的两个指令是 page 和 include。本节将针对 page 和 include 指令进行详细讲解。

11.3.1　page 指令

在 JSP 页面中，经常需要对页面的某些特性进行描述，例如页面的编码方式和 JSP 页面采用的语言等，这时可以通过 page 指令实现。

page 指令的语法格式如下：

```
<%@page 属性名 1="属性值 1" 属性名 2="属性值 2" ...%>
```

在上面的语法格式中，page 用于声明指令名称，属性名用于指定 JSP 页面的某些特性。page 指令的常用属性如表 11-1 所示。

表 11-1　page 指令的常用属性

属性名称	取值范围	描　述
language	java	指明解释该 JSP 文件时采用的语言，默认为 Java
import	任何包名和类名	指定在 JSP 页面翻译成的 Servlet 源文件中导入的包或类。import 是唯一可以声明多次的 page 指令属性。一个 import 属性可以引用多个类，中间用英文逗号隔开
session	true 和 false	指明该 JSP 内是否内置 Session 对象，如果为 true，则说明内置了 Session 对象，可以直接使用，否则没有内置 Session 对象。在默认情况下，session 属性的值为 true
isErrorPage	true 和 false	指定该页面是否为错误处理页面，如果为 true，则该 JSP 内置有一个 Exception 对象的 exception，可直接使用。在默认情况下，isErrorPage 的值为 false
errorPage	某个 JSP 页面的相对路径	指定一个错误页面，如果该 JSP 程序抛出一个未捕捉的异常，则转到 errorPage 指定的页面。errorPage 指定页面的 isErrorPage 属性为 true，且内置的 exception 对象为未捕捉的异常
contentType	有效的文档类型	客户端浏览器根据该属性判断文档类型，例如，HTML 格式为 text/html、纯文本格式为 text/plain、JPG 图像为 image/jpeg、GIF 图像为 image/gif、Word 文档为 application/msword
pageEnCoding	当前页面	指定页面编码格式

在表 11-1 中，除了 import 属性以外，其他的属性都只能出现一次，否则会编译失败。需要注意的是，page 指令的属性名称都是区分大小写的。

page 指令的使用示例如下：

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

在上述示例代码中，使用了 page 指令的 language、contentType、pageEncoding 和 import 属性。page 指令对整个页面都有效，与其书写的位置无关，但是习惯上把 page 指令写在 JSP 页面的最前面。

11.3.2 include 指令

在实际开发时，有时需要在 JSP 页面静态包含一个文件（如 HTML 文件），这时可以通过 include 指令实现。

include 指令的语法格式如下：

```
<%@include file="被包含的文件地址"%>
```

include 指令只有一个 file 属性，该属性用于指定插入到 JSP 页面目标位置的文件资源。插入文件资源的路径一般不以/开头，而是使用相对路径。

下面通过案例讲解 include 指令的具体用法。在 chapter11 项目的 WebContent 目录下创建两个 JSP 页面文件 date.jsp 和 include.jsp，在 include.jsp 文件中使用 include 指令将 date.jsp 文件包含其中，其代码如文件 11-2 和文件 11-3 所示。

文件 11-2 date.jsp

```
1  <%@page language="java" contentType="text/html; charset=UTF-8"%>
2  <html>
3  <head>
4      <title>Insert title here</title>
5  </head>
6  <body>
7      <!--输出当前时间-->
8      <%out.println(new java.util.Date().toLocaleString());%>
9  </body>
10 </html>
```

文件 11-3 include.jsp

```
1  <%@page language="java" contentType="text/html; charset=UTF-8"%>
2  <html>
3  <head>
4      <title>欢迎页面</title>
5  </head>
6  <body>
7      您访问页面的时间是：<%@include file="date.jsp"%>
```

```
8   </body>
9   </html>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter11/include.jsp，其显示结果如图 11-6 所示。

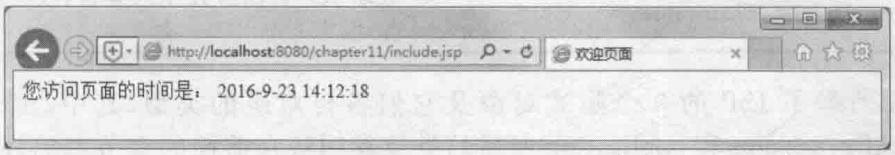


图 11-6　运行结果

从图 11-6 中可以看出，date.jsp 文件中用于输出当前日期的语句已被显示出来，这说明 include 指令成功地将 date.jsp 文件中的代码合并到了 include.jsp 文件中。

关于 include 指令的具体应用，有很多问题需要注意，具体如下。

(1) 被引入的文件必须遵循 JSP 语法，其中的内容可以包含静态 HTML、JSP 脚本元素和 JSP 指令等普通 JSP 页面所具有的一切内容。

(2) 除了指令元素以外，被引入的文件中的其他元素都被转换成相应的 Java 源代码，然后插入到当前 JSP 页面所翻译成的 Servlet 源文件中，插入位置与 include 指令在当前 JSP 页面中的位置保持一致。

(3) file 属性的设置值必须使用相对路径，如果以/开头，表示相对于当前 Web 应用程序的根目录(注意不是站点根目录)，否则表示相对于当前文件。需要注意的是，这里的 file 属性指定的相对路径是相对于文件(file)，而不是相对于页面(page)。

11.4　JSP 隐式对象

11.4.1　隐式对象的概述

在 JSP 页面中，有一些对象需要频繁使用，如果每次都重新创建这些对象则会非常麻烦。为了简化 Web 应用程序的开发，JSP 2.0 规范中提供了 9 个隐式(内置)对象，它们是 JSP 默认创建的，可以直接在 JSP 页面中使用。隐式对象的名称、类型和描述如表 11-2 所示。

表 11-2　JSP 隐式对象

名　称	类　型	描　述
out	javax.servlet.jsp.JspWriter	用于页面输出
request	javax.servlet.http.HttpServletRequest	得到用户请求信息
response	javax.servlet.http.HttpServletResponse	服务器向客户端的回应信息
config	javax.servlet.ServletConfig	服务器配置，可以取得初始化参数
session	javax.servlet.http.HttpSession	用于保存用户的信息
application	javax.servlet.ServletContext	所有用户的共享信息

续表

名　称	类　型	描　述
page	java. lang. Object	指当前页面转换后的 Servlet 类的实例
pageContext	javax. servlet. jsp. PageContext	JSP 的页面容器
exception	java. lang. Throwable	表示 JSP 页面所发生的异常，仅在错误页中起作用

表 11-2 列举了 JSP 的 9 个隐式对象及它们各自对应的类型，其中，由于 request、response、config、session 和 application 所属的类及其用法在前面的章节都已经讲解过，而 page 对象在 JSP 页面中很少被使用。因此，在后续几节中将针对 out、pageContext 和 exception 对象进行详细讲解。

11.4.2 out 对象

out 对象是 javax. servlet. jsp. JspWriter 类的实例对象，它的作用与 ServletResponse. getWriter()方法返回的 PrintWriter 对象非常相似，都是用于向客户端发送文本形式的实体内容。不同的是，out 对象的类型为 JspWriter，它相当于一种带缓存功能的 PrintWriter。

JSP 页面的 out 对象与 Servlet 引擎提供的缓冲区之间的工作关系如图 11-7 所示。

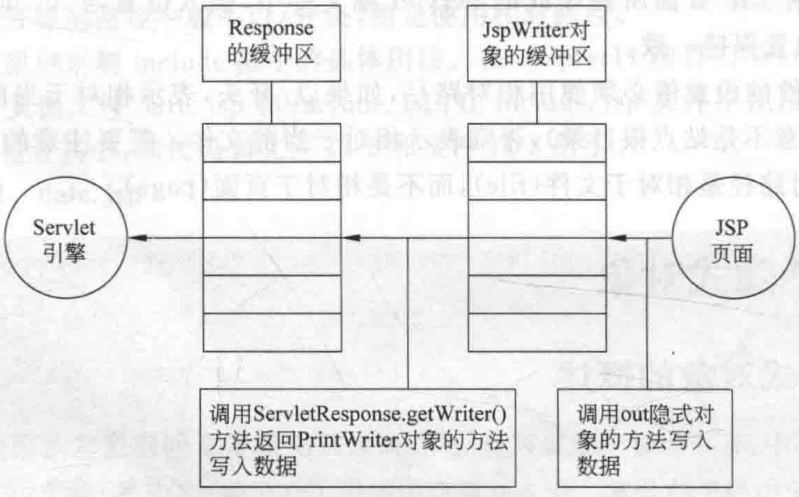


图 11-7　out 对象与 Servlet 引擎的关系

从图 11-7 中可以看出，在 JSP 页面中，通过 out 隐式对象写入数据相当于将数据插入到 JspWriter 对象的缓冲区中，只有调用了 ServletResponse. getWriter()方法，缓冲区中的数据才能真正写入到 Servlet 引擎所提供的缓冲区中。为了验证上述说法，下面通过具体的案例演示 out 对象的使用。

在 chapter11 项目的 WebContent 目录下创建一个名称为 out 的 JSP 页面，如文件 11-4 所示。

文件 11-4　out. jsp

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

```
<head>
    <title>out 对象</title>
</head>
<body>
<%
out.println("First<br />");
response.getWriter().println("Second<br />");
%>
</body>
</html>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter11/out.jsp 后，其显示结果如图 11-8 所示。

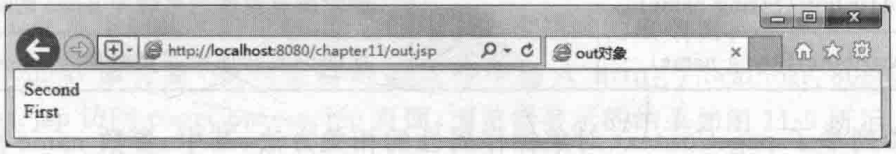


图 11-8　运行结果

从图 11-8 中可以看出，尽管 out.println();语句位于 response.getWriter().println();语句之前，但它的输出内容却在后面。由此可以说明，out 对象通过 print 语句写入数据后，直到整个 JSP 页面结束，out 对象中输入缓冲区的数据（即 First）才真正写入 Serlvet 引擎提供的缓冲区中，而 response.getWriter().println();语句则是直接把内容（即 Second）写入 Servlet 引擎提供的缓冲区中，Servlet 引擎按照缓冲区中的数据存放顺序输出内容。

11.4.3　pageContext 对象

pageContext 对象是 javax.servlet.jsp.PageContext 类的实例对象，它代表当前 JSP 页面的运行环境。在 JSP 页面中，使用 pageContext 对象可以获取 JSP 的其他八个隐式对象，它提供了一系列用于获取其他隐式对象的方法，如表 11-3 所示。

表 11-3　pageContext 获取隐式对象的方法

方 法 名	功 能 描 述
JspWriter getOut()	用于获取 out 隐式对象
Object getPage()	用于获取 page 隐式对象
ServletRequest getRequest()	用于获取 request 隐式对象
ServletResponse getResponse()	用于获取 response 隐式对象
HttpSession getSession()	用于获取 session 隐式对象
Exception getException()	用于获取 exception 隐式对象
ServletConfig getServletConfig()	用于获取 config 隐式对象
ServletContext getServletContext()	用于获取 application 隐式对象

表 11-3 中列举了 pageContext 获取其他隐式对象的方法，这样，当传递一个 pageContext 对象后，就可以通过这些方法轻松获取到其他八个隐式对象。

pageContext 对象不仅提供了获取隐式对象的方法，还提供了存储数据的功能。pageContext 对象存储数据是通过操作属性实现的，表 11-4 列举了 pageContext 操作属性的一系列方法。

表 11-4 pageContext 操作属性的相关方法

方法名称	功能描述
void setAttribute(String name,Object value,int scope)	用于设置 pageContext 对象的属性
Object getAttribute(String name,int scope)	用于获取 pageContext 对象的属性
void removeAttribute(String name,int scope)	删除指定范围内名称为 name 的属性
void removeAttribute(String name)	删除所有范围内名称为 name 的属性
Object findAttribute(String name)	从四个域对象中查找名称为 name 的属性

表 11-4 列举了 pageContext 对象操作属性的相关方法，其中，参数 name 指定属性名称，参数 scope 指定属性的作用范围。pageContext 对象的作用范围有四个值，具体如下。

(1) pageContext. PAGE_SCOPE：表示页面范围。

(2) pageContext. REQUEST_SCOPE：表示请求范围。

(3) pageContext. SESSION_SCOPE：表示会话范围。

(4) pageContext. APPLICATION_SCOPE：表示 Web 应用程序范围。

需要注意的是，当使用 findAttribute()方法查找名称为 name 的属性时，会按照 page、request、session 和 application 的顺序依次进行查找，如果找到，则返回属性的名称，否则返回 null。下面通过案例演示 pageContext 对象的使用。

在 chapter11 项目的 WebContent 目录下创建一个名称为 pageContext. jsp 的页面，编辑后如文件 11-5 所示。

文件 11-5 pageContext. jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"%>
<html>
<head>
    <title>pageContext 对象</title>
</head>
<body>
    <%
      //获取 request 对象
      HttpServletRequest req=(HttpServletRequest) pageContext.getRequest();
      //设置 page 范围内属性
      pageContext.setAttribute("str", "Java 基础案例教程",pageContext.PAGE_SCOPE);
      //设置 request 范围内属性
      req.setAttribute("str", "Java Web 程序设计任务教程");
```

```
        //获得的 page 范围属性
        String str1=(String)pageContext.getAttribute("str",pageContext.PAGE_SCOPE);
        //获得的 request 范围属性
        String str2=(String)pageContext.getAttribute("str",pageContext.
        REQUEST_SCOPE);
    %>
    <%="page 范围: "+str1 %><br />
    <%="request 范围: "+str2 %><br />
</body>
</html>
```

在上述代码中，首先使用 pageContext 获取了 request 对象，并设置 page 范围内属性；然后使用获取的 request 对象设置 request 范围内属性，接下来使用 pageContext 对象获得 page 和 request 范围内的相应属性，最后使用 JSP 表达式输出数据。

启动 Tomcat 服务器，在浏览器的地址栏中输入 http://localhost:8080/chapter11/pageContext.jsp 访问 pageContext.jsp 页面，浏览器显示的结果如图 11-9 所示。

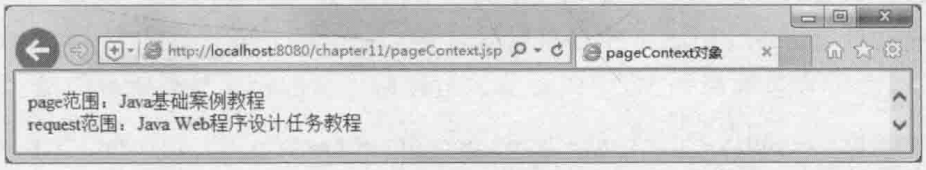


图 11-9　运行结果

从图 11-9 的显示结果中可以看出，通过 pageContext 对象可以获取到 request 对象，并且还可以获取不同范围内的属性。

11.4.4　exception 对象

exception 对象是 java.lang.Exception 类的实例对象，它用于封装 JSP 中抛出的异常信息。exception 对象只有在错误处理页面才可以使用，即 page 指令中指定了属性<%@ page isErrorPage="true"%>的页面。下面通过案例学习 exception 对象的使用。

在 chapter11 项目的 WebContent 目录下创建一个名称为 exception.jsp 的页面，在其中编写发生异常的代码，如文件 11-6 所示。

文件 11-6　exception.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" errorPage="error.jsp"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>exception object test</title>
</head>
```

```
<body>
    <%
        int a=8;
        int b=0;
    %>
    输出结果为<%=(a/b)%><!--此处会产生异常 -->
</body>
</html>
```

在 WebContent 目录下创建一个名称为 error.jsp 的页面，在其中获取从 exception.jsp 页面中传递过来的 exception 对象。需要注意的是，在其 page 指令中要将 isErrorPage 属性设置为 true，如文件 11-7 所示。

文件 11-7　error.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" isErrorPage="true"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>error page</title>
</head>
<body>
    <!--显示异常信息-->
    <%=exception.getMessage() %><br />
</body>
</html>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter11/exception.jsp，浏览器显示的界面如图 11-10 所示。

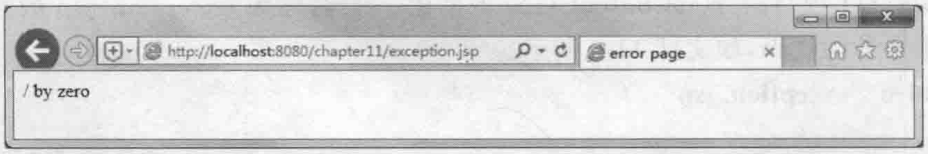


图 11-10　运行结果

从图 11-10 中可以看出，浏览器将错误的信息显示了出来，说明当 exception.jsp 页面发生错误时，会自动调用 error.jsp 页面进行错误处理。

需要注意的是，当 IE 浏览器不能够显示出结果并且报出 500 错误时，可以单击浏览器菜单栏中的【工具】(IE 11 可以单击右上角的齿轮图标)→Internet 选项→高级，将显示友好错误 HTTP 错误信息复选框中的√去掉，然后单击窗口的【确认】按钮并刷新页面即可。

11.5　JSP 动作元素

JSP 动作元素用于控制 JSP 的行为，执行一些常用的 JSP 页面动作。通过动作元素可以实现使用多行 Java 代码才能够实现的效果，如包含页面文件和实现请求转发等。

11.5.1　<jsp:include>动作元素

在 JSP 页面中，为了把其他资源的输出内容插入到当前 JSP 页面的输出内容中，JSP 技术提供了<jsp:include>动作元素。

<jsp:include>动作元素的使用语法格式如下：

```
<jsp:include page="relativeURL" flush="true|false" />
```

在上述语法格式中，page 属性用于指定被引入资源的相对路径，flush 属性用于指定是否将当前页面的输出内容刷新到客户端，在默认情况下，flush 属性的值为 false。

<jsp:include>包含的原理是将被包含的页面编译处理后将结果包含在页面中。当浏览器第一次请求一个使用<jsp:include>包含其他页面的页面时，Web 容器首先会编译被包含的页面，然后将编译处理后的返回结果包含在页面中，之后编译包含页面，最后将两个页面组合的结果回应给浏览器。

虽然 include 指令和<jsp:include>标签都能够包含一个文件，但它们之间有很大的区别，具体如下。

- <jsp:include>标签中要引入的资源和当前 JSP 页面是两个彼此独立的执行实体，即被动态引入的资源必须能够被 Web 容器独立执行。而 include 指令只能引入遵循 JSP 格式的文件，被引入文件与当前 JSP 文件需要共同合并才能翻译成一个 Servlet 源文件。
- <jsp:include>标签中引入的资源是在运行时才包含的，而且只包含运行结果。而 include 指令引入的资源是在编译时包含的，包含的是源代码。
- <jsp:include>标签运行原理与 RequestDispatcher. include()方法类似，即被包含的页面不能改变响应状态码或者设置响应头，而 include 指令没有这方面的限制。

11.5.2　<jsp:forward>动作元素

<jsp:forward>动作元素将当前请求转发到其他 Web 资源(HTML 页面、JSP 页面和 Servlet 等)，在执行请求转发后的当前页面将不再执行，而是执行该元素指定的目标页面。其使用语法格式如下：

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

在上述语法格式中，page 属性用于指定请求转发到的资源的相对路径，该路径是相对于当前 JSP 页面的 URL。为了使读者更好地理解<jsp:forward>动作元素，下面通过案例学习<jsp:forward>动作元素的使用。

首先编写一个用于实现转发功能的 forward.jsp 和一个用于显示当前时间的 welcome.jsp，如文件 11-8 和文件 11-9 所示。

文件 11-8 forward.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>forword page</title>
</head>
<body>
    <jsp:forward page="welcome.jsp" />
</body>
</html>
```

文件 11-9 welcome.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>welcome page</title>
</head>
<body>
    你好,欢迎进入首页,您本次访问的时间是:
    <%
        out.print(new java.util.Date());
    %>
</body>
</html>
```

启动 Tomcat 服务器，通过浏览器访问地址 http://localhost:8080/chapter11/forward.jsp，浏览器的显示结果如图 11-11 所示。

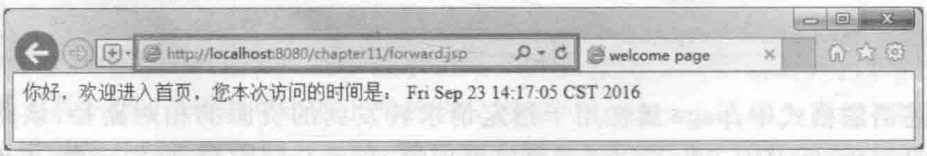


图 11-11 运行结果

从图 11-11 中可以看出，虽然浏览器访问的是 forward.jsp，但浏览器显示出了 welcome.jsp 页面的输出内容。出现这种情况的原因是，请求转发是服务器端的操作，浏览器并不知道请求的页面，所以浏览器的地址栏不会发生变化。

11.6　JSP 开发模型

11.6.1　JavaBean

在 JSP 网页开发的初期阶段，并没有框架与逻辑分层的概念，需要将 Java 代码嵌入到网页中处理 JSP 页面中的一些业务逻辑，如字符串处理和数据库操作等。这种开发方式虽然简单，但是如果将大量的 Java 代码嵌入到 JSP 页面，必定会为后期的修改和维护带来一定的困难。同时，将 Java 代码写入 JSP 页面中起不到代码重用的作用，也不能体现出面向对象的思想。为了弥补这种开发方式的缺点，JavaBean 应运而生。

JavaBean 是 Java 开发语言中一个可以重复使用的组件，它本质上就是一个 Java 类。我们可以引用 JavaBean 组件封装数据或者完成某些业务逻辑，如封装表单数据、字符串处理和数据库操作等。使用 JavaBean 后，人们通常会将 HTML 与 Java 代码相分离，将 Java 代码单独封装成一个处理某种业务逻辑的类中，然后在 JSP 页面中调用此类，这样就降低了 HTML 与 Java 代码之间的耦合度，简化了 JSP 页面，提高了 Java 代码的重用性和灵活性。这种与 HTML 代码相分离后，使用 Java 代码封装的类就是 JavaBean 组件。

为了规范 JavaBean 的开发，Sun 公司发布了 JavaBean 的编写规范，具体如下。

- JavaBean 必须具有一个公共的、无参的构造方法，这个方法可以是编译器自动产生的默认构造方法。
- JavaBean 提供公共的 getter 和 setter 方法，以方便对属性进行读/写操作。

为了让读者对 JavaBean 有一个直观的认识，下面讲解一个简单的封装数据的 JavaBean，如文件 11-10 所示。

文件 11-10　Book.java

```
1  package com.itheima.javabean;
2  public class Book {
3      private double price;
4      public double getPrice() {
5          return price;
6      }
7      public void setPrice(double price) {
8          this.price=price;
9      }
10 }
```

在文件 11-10 中，定义了一个 Book 类，该类就是一个 JavaBean，它没有定义构造方法，Java 编译器在编译时，会自动为这个类提供一个默认的构造方法。Book 类中定义了一个 price 属性，并提供了公共的 getPrice()和 setPrice()方法供外界访问属性。

在讲解面向对象时，经常会使用类的属性，类的属性指类的成员变量。在 JavaBean 中同

样也有属性，但是它和成员变量不是一个概念，它是以方法定义的形式出现的，这些方法必须遵循一定的命名规范，例如，如果在 JavaBean 中包含一个 String 类型的属性 name，那么在 JavaBean 中必须至少包含 getName()和 setName()方法中的一个，这两个方法的声明如下所示：

```
public String getName(){...}
public void setName(String name){...}
```

在上述两个方法声明中，方法体内的内容用“...”表示省略。关于这两个方法的相关讲解具体如下。

- getName()方法：称为 getter 方法或者属性访问器，该方法必须以小写的 get 前缀开始，后跟属性名，属性名的第一个字母必须大写，例如，nickName 属性的 getter 方法为 getNickName()。
- setName()方法：称为 setter 方法或者属性修改器，该方法必须以小写的 set 前缀开始，后跟属性名，属性名的第一个字母必须大写，例如，nickName 属性的 setter 方法为 setNickName()。

如果一个属性只有 getter 方法，则该属性为只读属性，如果一个属性只有 setter 方法，则该属性为只写属性，如果一个属性既有 getter 方法，又有 setter 方法，则该属性为读/写属性。通常来说，在开发 JavaBean 时，其属性都定义为读/写属性。

需要注意的是，对于 JavaBean 属性的命名方式有一个特殊情况，如果属性的类型为 boolean，它的命名方式应该使用 is/set 而不是 get/set。例如，有一个 boolean 类型的属性 married，则该属性所对应的方法声明如下所示：

```
public boolean isMarried(){...}
public void setMarried(boolean married){...}
```

从上面的代码中可以看出，married 属性的 setter 方法命名方式没有变化，而 getter 方法变成了 isMarried()方法，当然，写成 getMarried()也是可以的，只不过 isMarried()更符合命名规范。

11.6.2 JSP 设计模式

在 Web 开发中，为了更方便地使用 JSP 技术，Sun 公司为 JSP 技术提供了两种开发模型：JSP Model1 和 JSP Model2。JSP Model1 简单轻便，适合小型 Web 项目的快速开发；JSP Model2 模型是在 JSP Model1 的基础上提出的，它提供了更清晰的代码分层，更适用于多人合作开发的大型 Web 项目。在实际开发过程中，可以根据项目需求选择合适的模型。本节将详细介绍这两种开发模型。

1. JSP Model1

在讲解 JSP Model1 之前，先来了解一下 JSP 开发的早期模型。在早期使用 JSP 开发的 Java Web 应用中，JSP 文件是一个独立的、能自主完成所有任务的模块，它负责处理业务逻辑、控制网页流程和向用户展示页面等。JSP 早期模型的工作原理如图 11-12 所示。

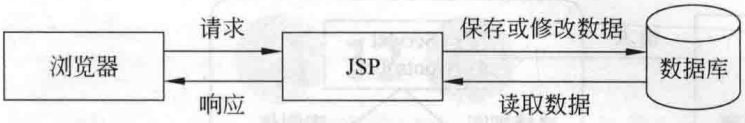


图 11-12　早期模型的工作原理

在图 11-12 中，首先浏览器会发送请求给 JSP，然后 JSP 会直接对数据库进行读取、保存或修改等操作，最后 JSP 会将操作结果响应给浏览器显示。在此程序中，由于 JSP 页面内编写了大量的功能代码，这将导致代码的可读性差、不利于修改和维护等问题。为了解决上述问题，Sun 公司提供了一种 JSP 开发的架构模型——JSP Model1。

JSP Model1 采用 JSP+JavaBean 的技术，将页面显示和业务逻辑分开。其中，JSP 实现流程控制和页面显示，JavaBean 对象实现封装数据和业务逻辑处理。

JSP Model1 的工作原理如图 11-13 所示。

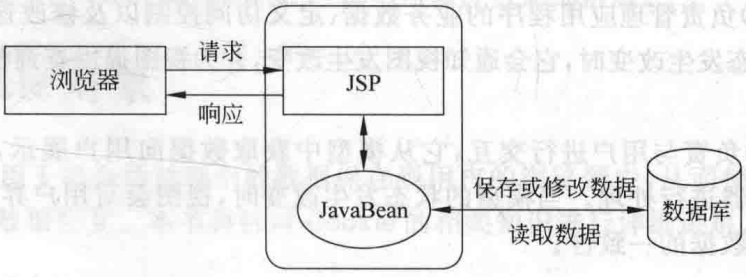


图 11-13　JSP Model1 模型的工作原理

在图 11-13 中，JSP Model1 模型将封装数据和处理数据的业务逻辑交给了 JavaBean 组件，JSP 只负责接收用户请求和调用 JavaBean 组件响应用户的请求。这种设计实现了数据、业务逻辑和页面显示的分离，在一定程度上实现了程序开发的模块化，降低了程序修改和维护的难度。

2. JSP Model2

JSP Model1 虽然将数据和部分的业务逻辑从 JSP 页面中分离了出去，但是 JSP 页面仍然需要负责流程控制和产生用户界面，对于一个业务流程复杂的大型应用程序来说，在 JSP 页面中依旧会嵌入大量的 Java 代码，这样会给项目管理带来很大的麻烦。为了解决这样的问题，Sun 公司在 Model1 的基础上又提出了 JSP Model2 架构模型。

JSP Model2 架构模型采用 JSP+Servlet+JavaBean 的技术，此技术将原本 JSP 页面中的流程控制代码提取出来，封装到 Servlet 中，从而实现了整个程序页面显示、流程控制和业务逻辑的分离。实际上 JSP Model2 模型就是 MVC(模型 Model-视图 View-控制器 Controller)设计模式，其中控制器的角色由 Servlet 实现，视图的角色由 JSP 页面实现，模型的角色由 JavaBean 实现。JSP Model2 的工作原理如图 11-14 所示。

从图 11-14 中可以看出，Servlet 充当了控制器的角色，它首先接收浏览器发送的请求，然后根据请求信息实例化 JavaBean 对象用于封装操作数据库后返回的数据，最后选择相应的 JSP 页面将响应结果显示在浏览器中。

3. MVC 设计模式

MVC 设计模式是施乐帕克研究中心在 20 世纪 80 年代为编程语言 Smalltalk-80 发明

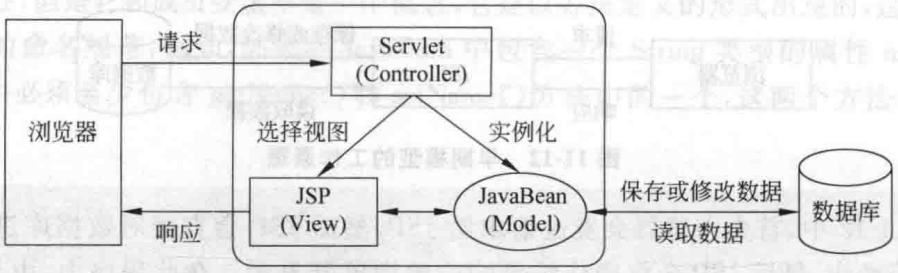


图 11-14 JSP Model2 模型的工作原理

的一种软件设计模式，它提供了一种按功能对软件进行模块划分的方法。MVC 设计模式将软件程序分为三个核心模块：模型(Model)、视图(View)和控制器(Controller)，这三个模块的作用如下所示。

(1) 模型

模型(Model)负责管理应用程序的业务数据、定义访问控制以及修改这些数据的业务规则。当模型的状态发生改变时，它会通知视图发生改变，并为视图提供查询模型状态的方法。

(2) 视图

视图(View)负责与用户进行交互，它从模型中获取数据向用户展示，同时也能将用户请求传递给控制器进行处理。当模型的状态发生改变时，视图会对用户界面进行同步更新，从而保持与模型数据的一致性。

(3) 控制器

控制器(Controller)是负责应用程序中处理用户交互的部分，它负责从视图中读取数据，控制用户输入，并向模型发送数据。

为了帮助读者更加清晰直观地看到这三个模块之间的关系，下面通过一张图描述它们之间的关系和功能，如图 11-15 所示。

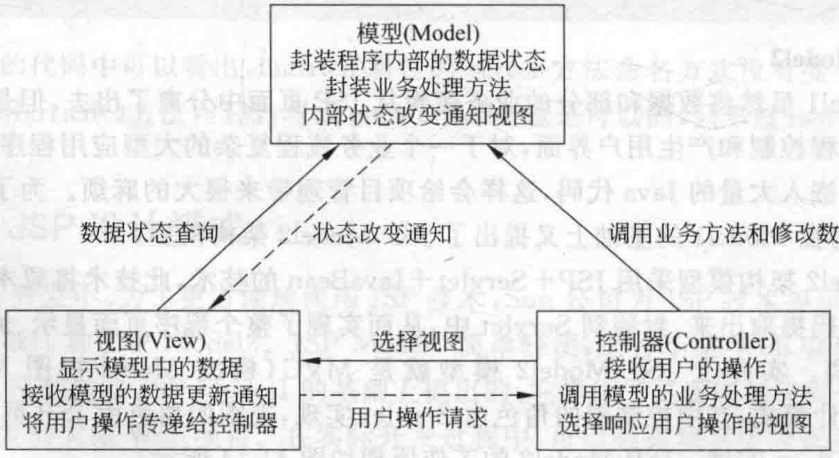


图 11-15 MVC 三个模型的关系和功能

从图 11-15 中可以看出这三个模块之间的关系：当控制器接收到用户的请求后，它根据请求信息调用模型组件的业务方法，业务方法处理完毕后，根据模型的返回结果选择相应的视图组件显示处理结果和模型中的数据。

第 12 章

Cookie和Session

在 Web 应用中，一个客户端(浏览器)与 Web 服务器端之间连续发生的一系列请求和响应的过程称为一个会话(如一个用户在某购物网站上的整个购物过程就是一个会话)。在整个会话过程中，客户端与服务器端的交互会产生一些数据，而 Web 服务器需要对这些数据进行保存。在 Servlet 技术中，提供了两个用于保存会话数据的对象，分别是 Cookie 和 Session。本章将针对 Cookie 和 Session 的相关知识进行详细讲解。

12.1 Cookie 对象

Cookie 主要用于将会话过程中的数据保存到用户的浏览器中，从而使浏览器和服务器可以更好地进行数据交互。本节将针对 Cookie 的相关知识进行详细讲解。

12.1.1 Cookie

Cookie 是浏览器端使用的一种记录用户状态的机制，它实际上是一小段保存在浏览器中的文本信息。当客户端通过浏览器请求服务器时，如果服务器需要记录该用户的状态，就会使用 response 对象向客户端浏览器颁发一个 Cookie，客户端浏览器会把 Cookie 保存起来。当通过浏览器再次请求该网站时，浏览器会把请求的网址连同该 Cookie 一同提交给服务器，服务器会检查该 Cookie，以此辨认用户状态。

服务器向客户端发送 Cookie 时，会在 HTTP 响应头字段中增加 Set-Cookie 响应头字段。Set-Cookie 头字段中设置的 Cookie 遵循一定的语法格式，如下所示：

```
Set-Cookie: user=admin; Path=/;
```

在上述示例中，user 表示 Cookie 的名称，admin 表示 Cookie 的值，Path 表示 Cookie 的属性。需要注意的是，Cookie 必须以键值对的形式存在，其属性可以有多个，但这些属性之间必须用英文分号(;)和空格分隔。

Cookie 在浏览器和服务器之间的传输过程如图 12-1 所示。

图 12-1 描述了 Cookie 在浏览器和服务器之间的传输过程。当用户第一次访问服务器时，服务器会在响应消息中增加 Set-Cookie 头字段，将用户信息以 Cookie 的形式发送给浏览器。一旦用户浏览器接收了服务器发送的 Cookie 信息，就会将它保存在浏览器的缓冲区中，这样，当浏览器后续访问该服务器时，会在请求消息中将用户信息以 Cookie 的形式发送给 Web 服务器，从而使服务器端分辨出当前请求是由哪个用户发出的，并可以针对这个用户送出一些特定的内容信息。

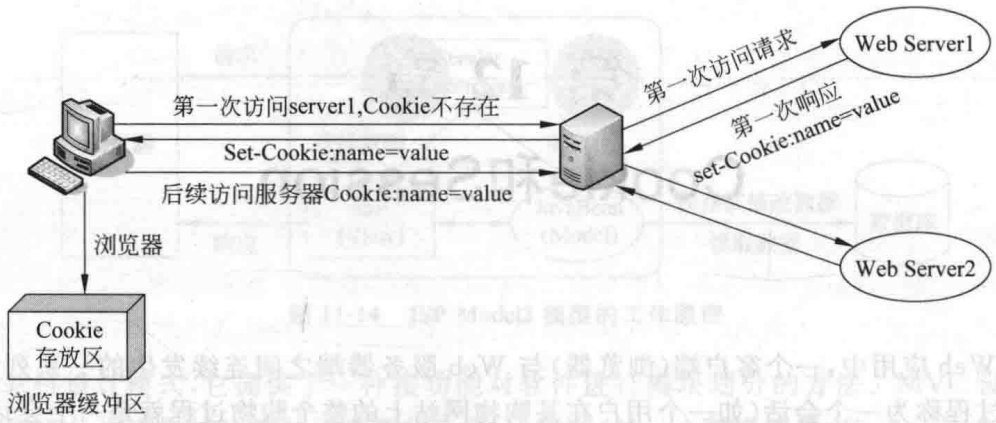


图 12-1 Cookie 在浏览器和服务器之间传输的过程

12.1.2 Cookie API

为了封装 Cookie 信息，在 Servlet API 中提供了一个 javax. servlet. http. Cookie 类，该类包含了生成 Cookie 信息和提取 Cookie 信息各个属性的方法。Cookie 的构造方法和常用方法具体如下。

1. 构造方法

Cookie 类有且仅有一个构造方法，具体语法格式如下：

```
public Cookie(java.lang.String name,java.lang.String value)
```

在 Cookie 的构造方法中，参数 name 用于指定 Cookie 的名称，value 用于指定 Cookie 的值。需要注意的是，Cookie 一旦创建，它的名称就不能更改，Cookie 的值可以为任意值，创建后允许被修改。

2. Cookie 类的常用方法

通过 Cookie 的构造方法创建 Cookie 对象后，便可调用该类的所有方法，表 12-1 列举了 Cookie 的常用方法。

表 12-1 Cookie 类的常用方法

方法声明	功能描述
String getName()	用于返回 Cookie 的名称
void setValue(String newValue)	用于为 Cookie 设置一个新的值
String getValue()	用于返回 Cookie 的值
void setMaxAge(int expiry)	用于设置 Cookie 在浏览器客户机上保持有效的秒数
int getMaxAge()	用于返回 Cookie 在浏览器客户机上保持有效的秒数
void setPath(String uri)	用于设置该 Cookie 项的有效目录路径
String getPath()	用于返回该 Cookie 项的有效目录路径
void setDomain(String pattern)	用于设置该 Cookie 项的有效域
String getDomain()	用于返回该 Cookie 项的有效域

表 12-1 列举了 Cookie 类的常用方法，下面针对表中比较难以理解的方法进行讲解。

(1) setMaxAge(int expiry)和 getMaxAge()方法

上面的这两个方法用于设置和返回 Cookie 在浏览器上保持有效的秒数。如果设置的值为一个正整数，浏览器会将 Cookie 信息保存在本地硬盘中。从当前时间开始，在没有超过指定的秒数之前，这个 Cookie 都保持有效，并且同一台计算机上运行的浏览器都可以使用这个 Cookie 信息；如果设置值为负整数，浏览器会将 Cookie 信息保存在缓存中，当浏览器关闭时，Cookie 信息会被删除；如果设置值为 0，则表示通知浏览器立即删除这个 Cookie 信息。在默认情况下，Max-Age 属性的值是－1。

(2) setPath(String uri)和 getPath()方法

上述两个方法是针对 Cookie 的 path 属性的。如果创建的某个 Cookie 对象没有设置 path 属性，那么该 Cookie 只对当前访问路径所属的目录及其子目录有效；如果想让某个 Cookie 项对站点所有目录下的访问路径都有效，则应调用 Cookie 对象的 setPath()方法将其 path 属性设置为/。

(3) setDomain(String pattern)和 getDomain()方法

上述两个方法是针对 Cookie 的 domain 属性的。domain 属性用于指定浏览器访问的域。例如，百度的域为 baidu. com。那么，当设置 domain 属性时，其值必须以. 开头，如 domain=. baidu. com。在默认情况下，domain 属性的值为当前主机名，浏览器在访问当前主机下的资源时，都会将 Cookie 信息回送给服务器。需要注意的是，domain 属性的值是不区分大小写的。

12.1.3　显示用户上次访问的时间

当用户访问某些 Web 应用时，应用经常会以提示的形式显示该用户上一次访问的时间。例如，QQ 登录成功后会显示用户上次登录的时间。本节将通过案例讲解如何使用 Cookie 技术实现显示用户上次访问的时间。

1. 创建项目

在 Eclipse 中新建一个名称为 chapter12 的 Web 项目，在 src 下新建一个名称为 com. itheima. cookie 的包，并在包中创建一个名称为 LastAccessServlet 的 Servlet 类，该类主要用于实现获取 Cookie 信息并将当前时间作为 Cookie 值发送给客户端，其实现代码如文件 12-1 所示。

文件 12-1　LastAccessServlet. java

```
1  package com.itheima.cookie;
2  import java.io.IOException;
3  import java.text.SimpleDateFormat;
4  import java.util.Date;
5  import javax.servlet.ServletException;
6  import javax.servlet.http.*;
7  public class LastAccessServlet extends HttpServlet {
8      private static final long serialVersionUID=1L;
9      public void doGet(HttpServletRequest request,
```

```
10                          HttpServletResponse response)
11              throws ServletException, IOException {
12          //指定服务器输出内容的编码方式为 UTF-8,防止发生乱码
13          response.setContentType("text/html;charset=utf-8");
14          String lastAccessTime=null;
15           //获取所有的 cookie,并将这些 cookie 存放在数组中
16          Cookie[] cookies=request.getCookies();
17          //遍历 cookies 数组
18          for (int i=0; cookies !=null && i <cookies.length; i++) {
19              if ("lastAccess".equals(cookies[i].getName())) {
20                  //如果 cookie 的名称为 lastAccess,则获取该 cookie 的值
21                  lastAccessTime=cookies[i].getValue();
22                  break;
23              }
24          }
25          //判断是否存在名称为 lastAccess 的 cookie
26          if (lastAccessTime==null) {
27              response.getWriter().print("您是首次访问本站!!!");
28          } else {
29              response.getWriter().print("您上次的访问时间是: "
30                          +lastAccessTime);
31          }
32          //创建 cookie,将当前时间作为 cookie 的值发送给客户端
33          String currentTime=new SimpleDateFormat("yyyy-MM-dd hh:mm:ss")
34                  .format(new Date());
35          Cookie cookie=new Cookie("lastAccess",currentTime);
36          //发送 cookie
37          response.addCookie(cookie);
38      }
39      public void doPost(HttpServletRequest req, HttpServletResponse resp)
40              throws ServletException, IOException {
41          this.doPost(req, resp);
42      }
43  }
```

在文件 12-1 中，第 14 行代码定义了一个 lastAccessTime 变量，用于保存用户上一次的访问时间；第 16 行代码用于获取所有的 Cookie；第 18～24 行代码用于遍历所有的 Cookie，并获取名为 lastAccess 的 Cookie，然后将获取的 Cookie 值保存在 lastAccessTime 变量中；第 26～31 行代码判断是否存在名称为 lastAccess 的 Cookie，不存在则 lastAccessTime 的值为 null，说明用户是首次访问本站，否则不是第一次访问，并输出上一次访问的时间；第 33～35 行代码用于记录当前访问的时间，保存到名为 lastAccess 的 Cookie 中；第 37 行代码用于向客户端发送 Cookie。

2. 运行项目并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter12/LastAccessServlet 访问 LastAccessServlet，由于是第一次访问 LastAccessServlet，因此会在浏览器中看到“您是首次访问本站!!!”的信息，如图 12-2 所示。

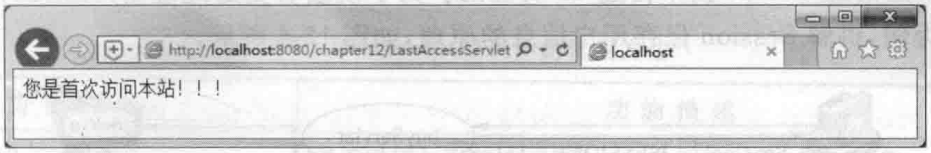


图 12-2　运行结果

刷新浏览器后，浏览器的显示结果如图 12-3 所示。

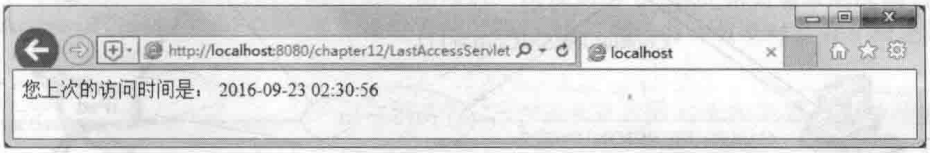


图 12-3　运行结果

图 12-3 显示了用户上次的访问时间，这是因为用户第一次访问 LastAccessServlet 时，LastAccessServlet 向浏览器发送了保存用户访问时间的 Cookie 信息。但是，当我们将图 12-3 所示的浏览器关闭后，再次打开浏览器访问地址 http://localhost:8080/chapter12/LastAccessServlet 时，浏览器中将再次显示图 12-2 中“您是首次访问本站!!!”的信息。之所以出现这样的情况，是因为在默认情况下，Cookie 对象的 Max-Age 属性的值是－1，即当浏览器关闭时会删除这个 Cookie 对象。因此，为了让 Cookie 对象在客户端有较长的存活时间，可以通过 setMaxAge()方法进行设置。例如，在文件 12-1 的第 35～36 行代码之间增加一行代码，将 Cookie 的有效时间设置为 1h，增加的代码如下所示：

```
cookie.setMaxAge(60 * 60);
```

这时，通过浏览器访问 LastAccessServlet 时，只要 Cookie 设置的有效时间没有结束，用户就会一直可以看到上次的访问时间。

12.2　Session 对象

Cookie 可以将用户的信息保存在用户的浏览器中，并且可以在多次请求下实现数据的共享。但是，如果传递的信息比较多时，使用 Cookie 技术则会增大服务器端程序处理的难度。这时，可以使用 Session 实现信息的存储。本节将针对 Session 进行详细讲解。

12.2.1　Session

Session 是服务器端使用的一种记录客户端状态的机制。当浏览器访问 Web 服务器

时，Servlet 容器就会创建一个 Session 对象和 ID 属性，当客户端后续访问服务器时，只要将标识(ID)号传递给服务器，服务器就能判断出该请求是哪个客户端发送的，从而选择与之对应的 Session 对象并为其服务。

需要注意的是，由于客户端需要接收、记录和回送 Session 对象的 ID，因此，在通常情况下，Session 是借助 Cookie 技术传递 ID 属性的。为了使读者更好地理解 Session，接下来以网站购物为例，讲解 Session 保存用户信息的原理，如图 12-4 所示。

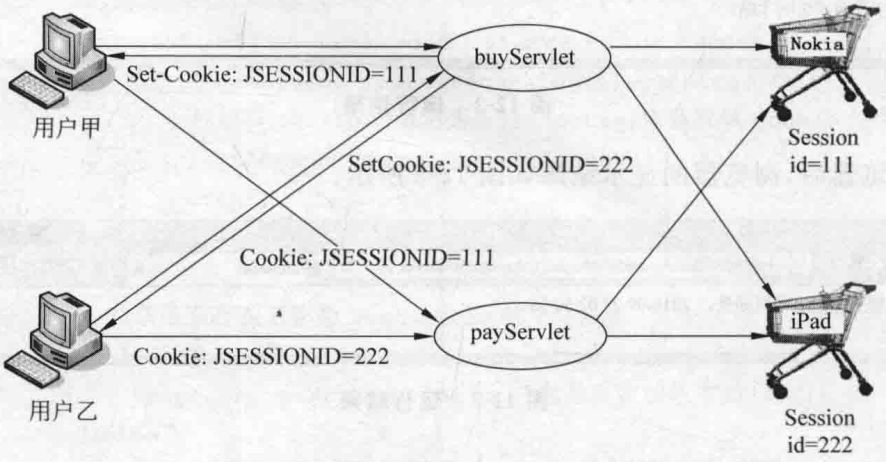


图 12-4 Session 保存用户信息的过程

在图 12-4 中，用户甲和用户乙都调用 buyServlet 将商品添加到购物车，然后调用 payServlet 进行商品结算。由于甲和乙购买商品的过程类似，在此，以用户甲为例进行详细说明。当用户甲访问购物网站时，服务器为用户甲创建了一个 Session 对象(相当于购物车)。当用户甲将 Nokia 手机添加到购物车时，Nokia 手机的信息便存放到了 Session 对象中。同时，服务器将 Session 对象的 ID 属性以 Cookie(Set-Cookie：JSESSIONID＝111)的形式返回给用户甲所使用的浏览器。当用户甲完成购物并进行结账时，需要向服务器发送结账请求，这时，浏览器将自动在请求消息头中将 Cookie(Cookie：JSESSIONID＝111)信息回送给服务器，服务器会根据 ID 属性找到为用户甲所创建的 Session 对象，并将 Session 对象中所存放的 Nokia 手机信息取出并进行结算。

12.2.2 HttpSession API

由于 Session 是与每个请求消息紧密相关的，因此，在 HttpServletRequest 中定义了用于获取 Session 对象的 getSession()方法，该方法有两种重载形式，具体如下：

```
public HttpSession getSession(boolean create);
public HttpSession getSession();
```

上述重载的两个方法都用于返回与当前请求相关的 HttpSession 对象，不同的是，第一个 getSession()方法根据传递的参数判断是否创建新的 HttpSession 对象，如果参数为 true，则在相关的 HttpSession 对象不存在时创建并返回新的 HttpSession 对象，否则不创建新的 HttpSession 对象，而是返回 null。第二个 getSession()方法则相当于第一个方法参数为 true 时的情况，在相关的 HttpSession 对象不存在时总是创建新的 HttpSession 对象。

需要注意的是，由于 getSession()方法可能会产生发送会话标识号的 Cookie 头字段，因此必须在发送任何响应内容之前调用 getSession()方法。

要想使用 HttpSession 对象管理会话数据，不仅需要获取 HttpSession 对象，还需要了解 HttpSession 对象的相关方法。HttpSession 接口中定义的操作会话数据的常用方法如表 12-2 所示。

表 12-2　HttpSession 接口中的常用方法

方法声明	功能描述
String getId()	用于返回与当前 HttpSession 对象关联的会话标识号
long getCreationTime()	返回 Session 创建的时间，这个时间是创建 Session 的时间与 1970 年 1 月 1 日 0 时 0 分 0 秒之间时间差的毫秒表示形式
long getLastAccessedTime()	返回客户端最后一次发送与 Session 相关请求的时间，这个时间是发送请求的时间与 1970 年 1 月 1 日 0 时 0 分 0 秒之间时间差的毫秒表示形式
void setMaxInactiveInterval(int interval)	用于设置当前 HttpSession 对象可空闲的以秒为单位的最长时间，也就是修改当前会话的默认超时间隔
boolean isNew()	判断当前 HttpSession 对象是否是新创建的
void invalidate()	用于强制使 Session 对象无效
ServletContext getServletContext()	用于返回当前 HttpSession 对象所属于的 Web 应用程序对象，即代表当前 Web 应用程序的 ServletContext 对象
void setAttribite(String name, Object value)	用于将一个对象与一个名称关联后存储到当前的 HttpSession 对象中
String getAttribute()	用于从当前 HttpSession 对象中返回指定名称的属性对象
void removeAttribute(String name)	用于从当前 HttpSession 对象中删除指定名称的属性

表 12-2 列举了 HttpSession 接口中的常用方法，这些方法都是用于操作 HttpSession 对象的。

12.2.3　Session 超时管理

当客户端第一次访问某个能开启会话功能的资源时，Web 服务器就会创建一个与该客户端对应的 HttpSession 对象。在 HTTP 协议中，Web 服务器无法判断当前的客户端浏览器是否还会继续访问，也无法检测客户端浏览器是否关闭，所以，即使客户端已经离开或关闭了浏览器，Web 服务器还是会保留与之对应的 HttpSession 对象。随着时间的推移，这些不再使用的 HttpSession 对象会在 Web 服务器中积累得越来越多，从而使 Web 服务器的内存耗尽。

为了解决上述问题，Web 服务器采用了超时限制的办法判断客户端是否还在继续访问。在一定时间内，如果某个客户端一直没有请求访问，那么 Web 服务器就会认为该客户端已经结束请求，并且将与该客户端会话所对应的 HttpSession 对象变成垃圾对象，等待垃圾收集器将其从内存中彻底清除。如果浏览器在超时后再次向服务器发出请求访问，那么 Web 服务器则会创建一个新的 HttpSession 对象，并为其分配一个新的 ID 属性。

在会话过程中，会话的有效时间可以在 web.xml 文件中设置，其默认值由 Servlet 容器定义。在 Tomcat 安装目录的 conf 文件夹下的 web.xml 文件中，可以找到如下一段配置信息：

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

在上面的配置信息中，设置的时间值以分钟为单位，即 Tomcat 服务器的默认会话超时间隔为 30 分钟。如果将<session-timeout>元素中的时间值设置成 0 或一个负数，则表示会话永不超时。由于<Tomcat 安装目录>\conf\web.xml 文件对站点内的所有 Web 应用程序都起作用，因此，如果想单独设置某个 Web 应用程序的会话超时间隔，则需要在相应项目的 web.xml 文件中进行设置。

12.2.4 实现用户登录

在 Web 应用中，用户登录功能是必不可少的。当某个用户进入应用系统的首页后，想要进一步操作时，如果该用户还未登录，则需要进入登录页面填写用户名和密码进行登录操作。在登录时，如果用户名和密码都正确，则登录成功；否则登录失败，并提示用户名或密码错误信息。在登录成功后，用户还可以单击【退出】按钮退出登录状态，回到系统首页。

为了实现上述需求，整个程序将定义 1 个 JSP 文件、1 个 JavaBean 和 3 个 Servlet，具体如下。

(1) login.jsp：用于显示用户登录的界面。在该文件的 form 表单中，有两个文本输入框，分别用于填写用户名和密码，还有一个提交按钮。

(2) User.java：该类用于封装登录页面中的用户名和密码信息。

(3) IndexServlet.java：该文件用于显示网站的首界面。如果用户没有登录，那么首界面需要提示用户登录，否则显示用户已经登录的信息。为了判断用户是否登录，该类在实现时，需要获取保存用户信息的 Session 对象。

(4) LoginServlet.java：该文件用于显示用户登录成功后的界面。如果用户登录成功，则跳转到网站首界面，否则，在页面进行友好提示用户名或密码错误，登录失败。

(5) LogoutServlet.java：用于完成用户注销功能。当用户单击【退出】按钮时，该类需要将 Session 对象中的用户信息移除，并跳转到网站的首界面。

用户登录的流程如图 12-5 所示。

图 12-5 描述了用户登录的整个流程，当用户访问某个应用的首界面时，首先会判断用户是否登录，如果已经登录，则在首界面中显示用户登录信息，否则进入登录页面，完成用户登录，然后显示用户登录信息。在用户登录的情况下，如果单击用户登录界面中的【退出】按钮，则会注销当前用户的信息，返回首界面。

1. 创建登录页面

在 chapter12 项目的 WebContent 目录下创建一个名称为 login.jsp 的文件，该页面中包含用户登录表单信息，如文件 12-2 所示。

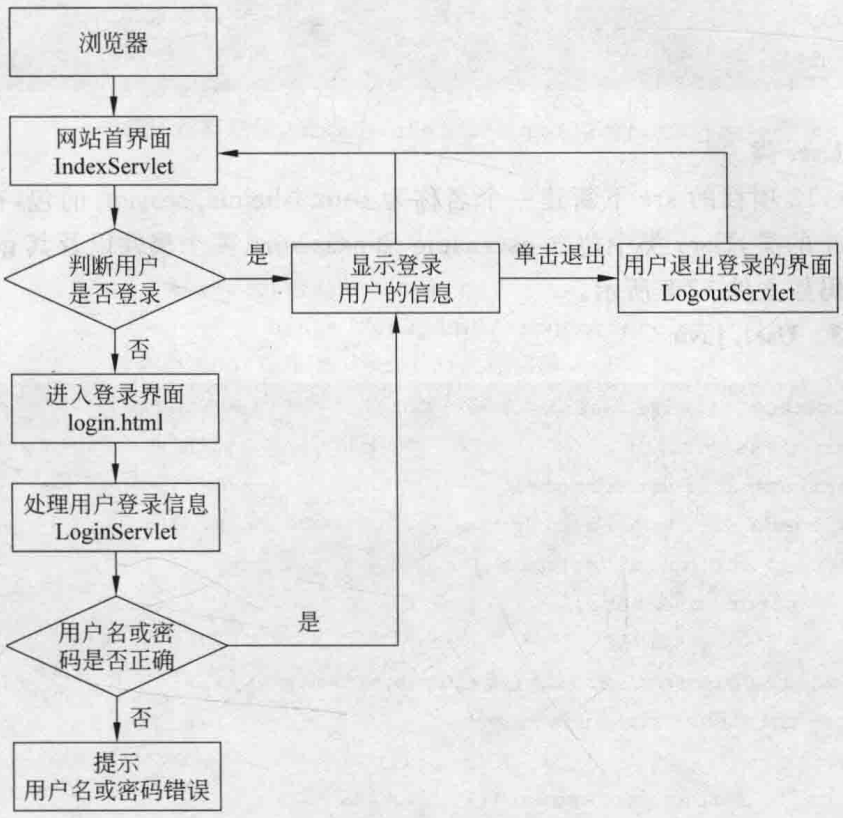


图 12-5　用户登录的流程

文件 12-2　login. jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>用户登录</title>
<style type="text/css">
    input[type=text], input[type=password] {
        width: 150px;
    }
</style>
</head>
<body>
    <form name="reg" action="/chapter12/LoginServlet" method="post">
        用户名：<input name="username" type="text" /><br />
        密    码：<input name="password" type="password" /><br />
        <input type="submit"value="提交" id="bt" />
    </form>
```

```
21  </body>
22  </html>
```

2. 创建 User 类

在 chapter12 项目的 src 下新建一个名称为 com. itheima. session 的包，在包中编写一个名称为 User 的类，User 类中包含 username 和 password 两个属性以及其 getter 和 setter 方法，实现代码如文件 12-3 所示。

文件 12-3　User. java

```
1  package com.itheima.session;
2  public class User {
3      private String username;
4      private String password;
5      public String getUsername() {
6          return username;
7      }
8      public void setUsername(String username) {
9          this.username=username;
10     }
11     public String getPassword() {
12         return password;
13     }
14     public void setPassword(String password) {
15         this.password=password;
16     }
17 }
```

3. 创建 Servlet

(1) 在 com. itheima. session 包中创建 IndexServlet 类，该类用于显示网站的首界面，编辑后的代码如文件 12-4 所示。

文件 12-4　IndexServlet. java

```
1  package com.itheima.session;
2  import java.io.IOException;
3  import javax.servlet.ServletException;
4  import javax.servlet.http.*;
5  public class IndexServlet extends HttpServlet {
6  public void doGet(HttpServletRequest request,HttpServletResponse response)
7              throws ServletException, IOException {
8          //解决乱码问题
9          response.setContentType("text/html;charset=utf-8");
10         //创建或者获取保存用户信息的 Session 对象
11         HttpSession session=request.getSession();
```

```
        User user=(User) session.getAttribute("user");
        if (user==null) {
            response.getWriter().print(
            "您还没有登录,请<a href='/chapter12/login.jsp'>登录</a>");
        } else {
            response.getWriter().print(
            "您已登录,欢迎你,"+user.getUsername()+"!");
            response.getWriter().print(
                    "<a href='/chapter12/LogoutServlet'>退出</a>");
            //创建 Cookie 存放 Session 的标识号
            Cookie cookie=new Cookie("JSESSIONID", session.getId());
            cookie.setMaxAge(60 * 30);
            cookie.setPath("/chapter12");
            response.addCookie(cookie);
        }
    }
    public void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
        doGet(request, response);
    }
}
```

在文件 12-4 中,如果用户没有登录,那么首界面会提示用户登录,否则显示用户已经登录的信息。为了判断用户是否登录,该类在实现时,获取了保存用户信息的 Session 对象。

(2) 在 com.itheima.session 包中编写 LoginServlet,该 Servlet 用于显示用户登录成功后的界面,其实现代码如文件 12-5 所示。

文件 12-5　LoginServlet.java

```
package com.itheima.session;
import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.http.*;
public class LoginServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
        response.setContentType("text/html;charset=utf-8");
        String username=request.getParameter("username");
        String password=request.getParameter("password");
        PrintWriter pw=response.getWriter();
        //假设正确的用户名是 admin,密码是 123
        if (("admin").equals(username) && ("123").equals(password)) {
```

```
            User user=new User();
            user.setUsername(username);
            user.setPassword(password);
            request.getSession().setAttribute("user", user);
            response.sendRedirect("/chapter12/IndexServlet");
        } else {
            pw.write("用户名或密码错误,登录失败");
        }
    }
    public void doPost(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
        doGet(request, response);
    }
}
```

在文件 12-5 中,如果用户登录成功,则跳转到网站首界面,否则在页面进行友好提示“用户名或密码错误,登录失败”。

(3) 在 com.itheima.session 包中编写 LogoutServlet,该 Servlet 用于完成用户注销功能,其实现代码如文件 12-6 所示。

文件 12-6 LogoutServlet.java

```
package com.itheima.session;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.http.*;
public class LogoutServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
            HttpServletResponse response) throws ServletException,
            IOException {
        //将 Session 对象中的 User 对象移除
        request.getSession().removeAttribute("user");
        response.sendRedirect("/chapter12/IndexServlet");
    }
    public void doPost(HttpServletRequest request,
     HttpServletResponse response)throws ServletException, IOException {
        doGet(request, response);
    }
}
```

在文件 12-6 中,当用户单击【退出】按钮时,该类会将 Session 对象中的用户信息移除,并跳转到网站的首界面。

4. 运行程序并查看结果

启动 Tomcat 服务器,在浏览器的地址栏中输入地址 http://localhost:8080/chapter12/

login.jsp 访问 login.jsp，浏览器显示的结果如图 12-6 所示。

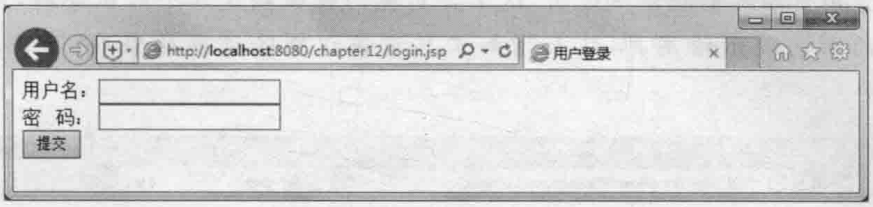


图 12-6　运行结果

在图 12-6 中的用户名和密码输入框中输入用户名“admin”和密码“123”后，单击【提交】按钮，其页面显示效果如图 12-7 所示。

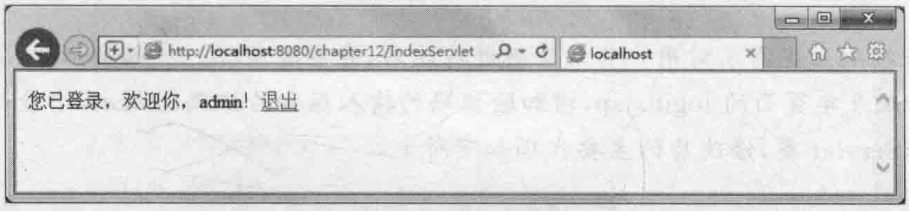


图 12-7　登录后的页面显示

从图 12-7 中可以看出，用户登录成功，提示信息为“您已登录，欢迎你，admin!”。如果用户想退出登录，则可以单击【退出】按钮，此时，浏览器显示的结果如图 12-8 所示。

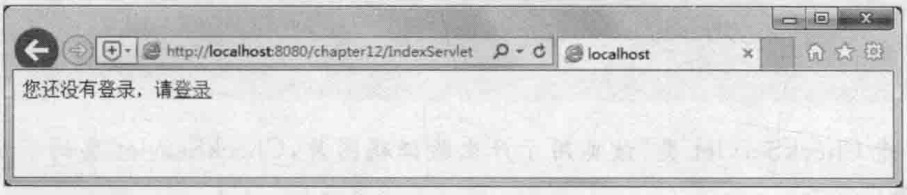


图 12-8　退出后的页面

如果用户输入的用户名或者密码错误，那么，当单击【提交】按钮时登录会失败，浏览器显示的效果如图 12-9 所示。

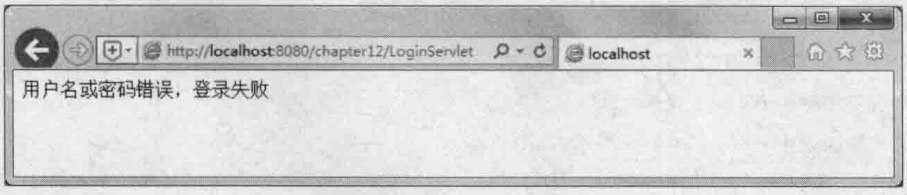


图 12-9　登录失败的页面

至此，用户登录功能已经实现成功。

多学一招：利用 Session 实现一次性验证码

在 12.2.4 节中，使用 Session 实现了用户登录的功能，但在实际开发中，为了保证用户信息的安全，都会在网站登录的界面中添加一次性验证码限制人们使用软件暴力猜测密码。一次性验证码的功能同样可以使用 Session 实现。下面将对用户登录案例进行修改，在用

户登录时，填写一次性验证码。

为了避免用户输入的验证码过长，这里要实现的验证码是 4 个随机字符，同时，验证码还会以图片的形式展示给用户，从而增加工具程序识别验证码的难度，验证码的效果如图 12-10 所示。

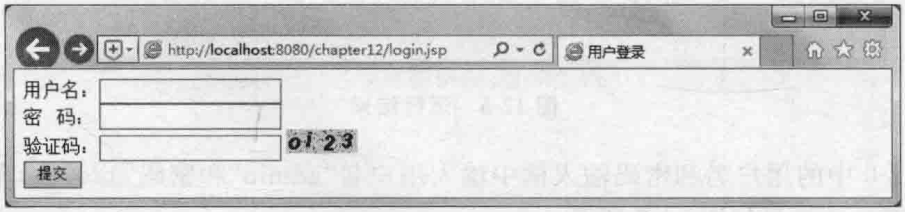


图 12-10 带有验证码的登录页面

接下来按照上述需求对用户登录案例进行改写，分步骤实现一次性验证码，具体如下。

(1) 修改表单页面的 login.jsp，增加验证码的输入框和验证码图片，其中验证码的图片来自 CheckServlet 类，修改后的主要代码如下所示：

```
<form name="reg" action="/chapter12/LoginServlet" method="post">
    用户名：<input name="username" type="text" /><br />
    密   码：<input name="password" type="password" /><br />
    验证码：<input name="check_code" type="text" />
          <img src="/chapter12/CheckServlet"><br />
          <input type="submit" value="提交" id="bt" />
</form>
```

(2) 创建 CheckServlet 类，该类用于产生验证码图片，CheckServlet 类的实现代码如文件 12-7 所示。

文件 12-7 CheckServlet.java

```
1  package com.itheima.session;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  import java.awt.*;
6  import java.awt.image.*;
7  import javax.imageio.ImageIO;
8  public class CheckServlet extends HttpServlet
9  {
10     private static int WIDTH=60;        //验证码图片宽度
11     private static int HEIGHT=20;       //验证码图片高度
12     public void doGet(HttpServletRequest request,HttpServletResponse response)
13           throws ServletException,IOException{
14        HttpSession session=request.getSession();
15        response.setContentType("image/jpeg");
```

```
16          ServletOutputStream sos=response.getOutputStream();
17          //设置浏览器不要缓存此图片
18          response.setHeader("Pragma","No-cache");
19          response.setHeader("Cache-Control","no-cache");
20          response.setDateHeader("Expires", 0);
21          //创建内存图像并获得其图形上下文
22          BufferedImage image=
23              new BufferedImage(WIDTH, HEIGHT, BufferedImage.TYPE_INT_RGB);
24          Graphics g=image.getGraphics();
25          //产生随机的验证码
26          char [] rands=generateCheckCode();
27          //产生图像
28          drawBackground(g);
29          drawRands(g,rands);
30          //结束图像的绘制过程,完成图像
31          g.dispose();
32          //将图像输出到客户端
33          ByteArrayOutputStream bos=new ByteArrayOutputStream();
34          ImageIO.write(image, "JPEG", bos);
35          byte [] buf=bos.toByteArray();
36          response.setContentLength(buf.length);
37          //下面的语句也可写成: bos.writeTo(sos);
38          sos.write(buf);
39          bos.close();
40          sos.close();
41          //将当前验证码存入到 Session 中
42          session.setAttribute("check_code",new String(rands));
43      }
44      //生成一个 4 个字符的验证码
45      private char [] generateCheckCode()
46      {
47          //定义验证码的字符表
48          String chars="0123456789abcdefghijklmnopqrstuvwxyz";
49          char [] rands=new char[4];
50          for(int i=0; i<4; i++)
51          {
52              int rand= (int)(Math.random() * 36);
53              rands[i]=chars.charAt(rand);
54          }
55          return rands;
56      }
57      private void drawRands(Graphics g , char [] rands)
58      {
```

```
        g.setColor(Color.BLACK);
        g.setFont(new Font(null,Font.ITALIC|Font.BOLD,18));
        //在不同的高度上输出验证码的每个字符
        g.drawString(""+rands[0],1,17);
        g.drawString(""+rands[1],16,15);
        g.drawString(""+rands[2],31,18);
        g.drawString(""+rands[3],46,16);
    }
    private void drawBackground(Graphics g)
    {
        //画背景
        g.setColor(new Color(0xDCDCDC));
        g.fillRect(0, 0, WIDTH, HEIGHT);
        //随机产生 120 个干扰点
        for(int i=0; i<120; i++)
        {
            int x=(int)(Math.random() * WIDTH);
            int y=(int)(Math.random() * HEIGHT);
            int red=(int)(Math.random() * 255);
            int green=(int)(Math.random() * 255);
            int blue=(int)(Math.random() * 255);
            g.setColor(new Color(red,green,blue));
            g.drawOval(x,y,1,0);
        }
    }
}
```

(3) 对 LoginServlet 类进行修改，增加对验证码的判断，修改后的代码如文件 12-8 所示。

文件 12-8 LoginServlet.java

```
package com.itheima.session;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
public class LoginServlet extends HttpServlet {
public void doGet(HttpServletRequest request,HttpServletResponse response)
                                        throws ServletException, IOException {
        response.setContentType("text/html;charset=utf-8");
        String username=request.getParameter("username");
        String password=request.getParameter("password");
        String checkCode=request.getParameter("check_code");
        String savedCode=(String) request.getSession().getAttribute("check_code");
```

```
        PrintWriter pw=response.getWriter();
        if (("admin").equals(username) && ("123").equals(password)
                && checkCode.equals(savedCode)) {
            User user=new User();
            user.setUsername(username);
            user.setPassword(password);
            request.getSession().setAttribute("user", user);
            response.sendRedirect("/chapter12/IndexServlet");
        } else if (checkCode.equals(savedCode)) {
            pw.write("用户名或密码错误,登录失败");
        } else {
            pw.write("验证码错误");
        }
    }
    public void doPost(HttpServletRequest request,
        HttpServletResponse response)throws ServletException, IOException {
        doGet(request, response);
    }
}
```

(4) 启动 Tomcat 服务器，在浏览器中通过地址 http://localhost:8080/chapter12/login.jsp 访问 login.jsp，浏览器将显示出如图 12-10 所示的界面。此时如果在页面中输入用户名 admin 和密码 123，但是不填写验证码，单击【提交】按钮后，页面会提示验证码错误，如图 12-11 所示。



图 12-11　显示结果

如果填写了正确的用户名、密码以及验证码，即可登录成功。

第 13 章

EL表达式和JSTL

为了获取 Servlet 域对象中存储的数据，JSP 2.0 规范提供了 EL 表达式语言，同时为了降低 JSP 页面的复杂度，增强代码的重用性，Sun 公司还制定了一套标准标签库 JSTL。本章将针对 EL 表达式以及 JSTL 标签库进行详细讲解。

13.1 EL 表达式

13.1.1 EL 概述

在 JSP 开发中，为了获取 Servlet 域对象中存储的数据，经常需要书写很多 Java 代码，这些代码会使 JSP 页面的可读性变差，不利于页面的维护，为此，JSP 2.0 规范提供了 EL 表达式。EL 是 Expression Language 的缩写，它是一种简单的数据访问语言。

EL 表达式的使用十分简单，它以“${”符号开始，以“}”符号结束，具体格式如下：

```
${表达式}
```

需要注意的是，${表达式}中的表达式必须符合 EL 语法要求。关于 EL 语法的相关知识将在后续小节进行详细讲解。

13.1.2 EL 中的标识符

在 EL 表达式中，经常需要使用一些符号标记一些名称，如变量名和自定义函数名等，这些符号被称为标识符。EL 表达式中的标识符可以由任意的大小写字母、数字和下画线组成，为了避免出现非法的标识符，在定义标识符时还需要遵循以下规范：

- 不能以数字开头。
- 不能是 EL 中的保留字，如 and、or、gt。
- 不能是 EL 隐式对象，如 pageContext。
- 不能包含单引号(')、双引号(")、减号(-)和正斜线等特殊字符。

下面的标识符都是合法的。

username　　username123　　user_name　　_userName

注意：下面的标识符都是不合法的。

123username　　or　　user"name　　pageContext

13.1.3　EL 中的保留字

保留字就是编程语言中事先定义好并赋予了特殊含义的单词，和其他语言一样，EL表达式中也定义了许多保留字，具体如下：

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

需要注意的是，EL 表达式中的这些保留字不能作为标识符，以免在程序编译时发生错误。

13.1.4　EL 中的变量

EL 表达式中的变量就是一个基本的存储单元，EL 表达式可以将变量映射到一个对象上，具体示例如下所示：

```
${product}
```

在上述示例中，product 就是一个变量。EL 表达式中的变量不用事先定义就可以直接使用，例如，表达式 ${product}就可以访问变量 product 的值。

13.1.5　EL 中的常量

EL 表达式中的常量又称字面量，它是不能改变的数据。在 EL 表达式中包含多种常量，下面分别对这些常量进行介绍。

1. 布尔常量

布尔常量用于区分一个事物的正反两面，它的值只有两个，分别是 true 和 false。

2. 整型常量

整型常量与 Java 中的十进制整型常量相同，它的取值范围在 Java 语言中定义的常量 Long. MIN_VALUE 到 Long. MAX_VALUE 之间，即$(-2)^{63}$～$2^{63}-1$ 之间的整数。

3. 浮点数常量

浮点数常量使用整数部分加小数部分表示，也可以使用指数形式表示，例如，1. 2E4 和 1. 2 都是合法的浮点数常量，它的取值范围在 Java 语言中定义的常量 Double. MIN_VALUE 到 Double. MAX_VALUE 之间，即 4. 9E－324～1. 8E308 之间的浮点数。

4. 字符串常量

字符串常量是使用单引号或双引号引起来的一连串字符。由于字符串常量需要使用单引号或双引号引起来，所以字符串本身包含的单引号或双引号需要用反斜杠(\)进行转义，即用\'表示字面意义上的单引号，用\"表示字面意义上的双引号。如果字符串本身包含反斜杠(\)，那么也要进行转义，即用\\表示字面意义上的一个反斜杠。

需要注意的是，只有字符串常量用单引号引起来时，字符串本身包含的单引号才需要进行转义，而双引号不必进行转义；只有字符串常量用双引号引起来时，字符串本身

包含的双引号才需要进行转义，而单引号不必转义，例如"ab'4c\"d5\\e"表示的字符串是ab'4c"d5\e。

5. Null 常量

Null 常量用于表示变量引用的对象为空，它只有一个值，用 null 表示。

13.1.6 EL 中的运算符

EL 表达式支持简单的运算，例如，加(+)、减(-)、乘(*)、除(/)等。为此，EL 提供了多种运算符，根据运算方式的不同，EL 中的运算符包括以下几种。

1. 点运算符(.)

EL 表达式中的点运算符用于访问 JSP 页面中某些对象的属性，如 JavaBean 对象、List 集合、Array 数组等，其语法格式如下：

```
${customer.name}
```

在上述语法格式中，表达式 ${customer. name} 中的点运算符的作用就是访问 customer 对象中的 name 属性。

2. 方括号运算符([])

EL 表达式中的方括号运算符与点运算符的功能相同，都用于访问 JSP 页面中某些对象的属性，当获取的属性名中包含一些特殊符号，如-或? 等并非字母或数字的符号，就只能使用方括号运算符访问该属性，其语法格式如下：

```
${user["My-Name"]}
```

需要注意的是，在访问对象的属性时，在通常情况下都会使用点运算符作为简单的写法，但在实际上，方括号运算符比点运算符的应用更加广泛。下面对比一下这两种运算符在实际开发中的应用，具体如下。

- 点运算符和方括号运算符在某种情况下可以互换，如 ${student. name} 等价于 ${student["name"]}。
- 方括号运算符还可以访问 List 集合或数组中指定索引的某个元素，如表达式 ${users[0]}用于访问集合或数组中的第一个元素。在这种情况下，只能使用方括号运算符，而不能使用点运算符。
- 方括号运算符和点运算符可以相互结合使用，例如，表达式 ${users[0]. userName} 可以访问集合或数组中的第一个元素的 userName 属性。

3. 算术运算符

EL 表达式中的算术运算符用于对整数和浮点数的值进行算术运算。使用这些算术运算符可以非常方便地在 JSP 页面进行算术运算，并且可以简化页面的代码量。EL 表达式中所有的算术运算符如表 13-1 所示。

表 13-1 列举了 EL 表达式中所有的算术运算符，在使用这些运算符时需要注意两个问题，一运算符既可以作为减号，也可以作为负号，/或 div 运算符在进行除法运算时，商为小数。

表 13-1　算术运算符

算术运算符	说　明	算术表达式	结　果
+	加	${10+2}	12
-	减	${10-2}	8
*	乘	${10 * 2}	20
/(或 div)	除	${10/4}或${10 div 4}	2.5
%(或 mod)	取模(取余)	${10%4}或${10 mod 4}	2

4. 比较运算符

EL 表达式中的比较运算符用于比较两个操作数的大小，操作数可以是各种常量、EL 变量或 EL 表达式，所有的运算符执行的结果都是布尔类型。表 13-2 列举了 EL 表达式中所有的比较运算符。

表 13-2　比较运算符

比较运算符	说　明	算术表达式	结　果
==(或 eq)	等于	${10==2}或${10 eq 2}	false
!=(或 ne)	不等于	${10!=2}或${10 ne 2}	true
<(或 lt)	小于	${10<2}或${10 lt 2}	false
>(或 gt)	大于	${10>2}或${10 gt 2}	true
<=(或 le)	小于等于	${10<=2}或${10 le 2}	false
>=(或 ge)	大于等于	${10>=2}或${10 ge 2}	true

表 13-2 列举了 EL 表达式中所有的比较运算符，在使用这些运算符时需要注意两个问题，具体如下。

- 比较运算符中的==是两个等号，千万不能只写一个等号。
- 为了避免与 JSP 页面的标签产生冲突，对于后四种比较运算符，EL 表达式中通常使用括号内的表示方式，例如，使用 lt 代替<运算符，如果运算符后面是数字，在运算符和数字之间至少要有一个空格，例如 ${1lt 2}，但后面如果有其他符号，则可以不加空格，例如 ${1lt(1+1)}。

5. 逻辑运算符

EL 表达式中的逻辑运算符用于对结果为布尔类型的表达式进行运算，运算的结果仍为布尔类型。表 13-3 列举了 EL 表达式中所有的逻辑运算符。

表 13-3　逻辑运算符

逻辑运算符	说　明	算术表达式	结　果
&&(and)	逻辑与	${true&&false}或${true and false}	false
||(or)	逻辑或	${false\|\|true}或${false or true}	true
!(not)	逻辑非	${!true}或${not true}	false

表13-3列出了EL表达式中的三种逻辑运算符，需要注意的是，在使用&&运算符时，如果有一个表达式的结果为false，则结果必为false，在使用||运算符时，如果有一个表达式的结果为true，则结果必为true。

6. empty运算符

EL表达式中的empty运算符用于判断某个对象是否为null或""，结果为布尔类型，其基本的语法格式如下所示：

```
${empty var}
```

7. 条件运算符

EL表达式中的条件运算符用于执行某种条件判断，它类似于Java语言中的if-else语句，其语法格式如下：

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

在上述语法格式中，表达式A的计算结果为布尔类型，如果表达式A的计算结果为true，则执行表达式B并返回B的值；如果表达式A的计算结果为false，则执行表达式C并返回C的值，例如表达式${(1==2)? 3:4}的结果就为4。

8. ()运算符

EL表达式中的圆括号用于改变其他运算符的优先级，例如表达式${a * b+c}，在正常情况下会先计算a * b的积，然后再将计算的结果与c相加，如果在这个表达式中加一个圆括号运算符，将表达式修改为${a * (b+c)}，则先计算b与c的和，再将计算的结果与a相乘。

需要注意的是，EL表达式中的运算符都有不同的运算优先级，当EL表达式中包含多种运算符时，它们必须按照各自优先级的大小进行运算。表13-4描述了这些运算符的优先级。

表13-4 运算符的优先级

优先级	运算符	优先级	运算符
1	[] .	6	< > <= >= lt gt le ge
2	()	7	== != eq ne
3	-(负号) not ! empty	8	&& and
4	* / div % mod	9	|| or
5	+ -(减号)	10	?:

表13-4列举了不同运算符各自的优先级大小。对于初学者来说，这些运算符的优先级不必刻意记忆。为了防止产生歧义，建议读者尽量使用()运算符实现想要的运算顺序。

注意：在应用EL表达式取值时，没有数组的下标越界，没有空指针异常，没有字符串的拼接。

13.1.7　EL 隐式对象

EL 中的隐式对象共有 11 个，具体如表 13-5 所示。

表 13-5　EL 中的隐式对象

隐式对象名称	描　　述
pageContext	对应于 JSP 页面中的 pageContext 对象
pageScope	代表 page 域中用于保存属性的 Map 对象
requestScope	代表 request 域中用于保存属性的 Map 对象
sessionScope	代表 session 域中用于保存属性的 Map 对象
applicationScope	代表 application 域中用于保存属性的 Map 对象
param	表示一个保存了所有请求参数的 Map 对象
paramValues	表示一个保存了所有请求参数的 Map 对象，它对于某个请求参数，返回的是一个 String 类型数组
header	表示一个保存了所有 http 请求头字段的 Map 对象
headerValues	表示一个保存了所有 http 请求头字段的 Map 对象，返回 String 类型数组
cookie	用于取得使用者的 cookie 值，cookie 的类型是 Map
initParam	表示一个保存了所有 Web 应用初始化参数的 Map 对象

在表 13-5 列举的隐式对象中，pageContext 可以获取其他 10 个隐式对象，pageScope、requestScope、sessionScope、applicationScope 是用于获取指定域的隐式对象，param 和 paramValues 是用于获取请求参数的隐式对象，header 和 headerValues 是用于获取 HTTP 请求消息头的隐式对象，cookie 是用于获取 Cookie 信息的隐式对象，initParam 是用于获取 Web 应用初始化信息的隐式对象，本节将针对常用的几个隐式对象进行详细讲解。

1. pageContext 对象

JSP 页面的隐式对象可以使用 EL 表达式中的 pageContext 隐式对象获取。pageContext 隐式对象的使用示例如下：

```
${pageContext.response.characterEncoding}
```

在上述示例中，pageContext 对象用于获取 response 对象中的 characterEncoding 属性。下面通过案例演示 pageContext 隐式对象的具体用法。

在 Eclipse 中创建一个名称为 chapter13 的 Web 项目，在项目的 WebContent 目录下创建一个名为 pageContext.jsp 的文件，编辑后如文件 13-1 所示。

文件 13-1　pageContext.jsp

```
1  <%@page language="java" contentType="text/html; charset=utf-8"
2      pageEncoding="utf-8"%>
3  <html>
```

```
<head>
<title>pageContext</title>
</head>
<body>
    请求 URI 为${pageContext.request.requestURI} <br />
    Content-Type 响应头：${pageContext.response.contentType} <br />
    服务器信息为${pageContext.servletContext.serverInfo} <br />
    Servlet 注册名为${pageContext.servletConfig.servletName} <br />
</body>
</html>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter13/pageContext.jsp 访问 pageContext.jsp 页面，浏览器窗口中显示的结果如图 13-1 所示。

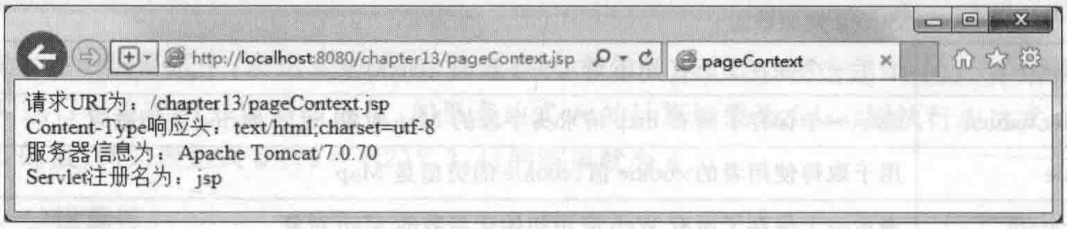


图 13-1 pageContext.jsp 页面

从图 13-1 中可以看出，使用 EL 表达式中的 pageContext 对象成功获取到了 request、response、servletContext 和 servletConfig 对象中的属性。

需要注意的是，不要将 EL 表达式中的隐式对象与 JSP 中的隐式对象混淆，只有 pageContext 对象是它们所共有的，其他隐式对象则毫不相关。

2. Web 域相关对象

在 Web 开发中，PageContext、HttpServletRequest、HttpSession 和 ServletContext 这四个对象之所以可以存储数据，是因为它们内部都定义了一个 Map 集合，但是这些 Map 集合也是有一定作用范围的，例如，HttpServletRequest 对象存储的数据只在当前请求中可以获取。习惯性上，人们把这些 Map 集合称为域，把这些 Map 集合内的对象称为域对象。在 EL 表达式中，为了获取指定域中的数据，提供了 pageScope、requestScope、sessionScope 和 applicationScope 这四个隐式对象，其示例代码如下：

```
${pageScope.userName}
${requestScope.userName}
${sessionScope.userName}
${applicationScope.userName}
```

需要注意的是，EL 表达式只能在这四个作用域中获取数据。下面通过案例演示这四个隐式对象如何访问 JSP 域对象中的属性。在 chapter13 项目的 WebContent 目录下，新建

一个名为 scopes.jsp 的文件，编辑后如文件 13-2 所示。

文件 13-2　scopes.jsp

```
<%@page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<html>
<head>
<title>域对象</title>
</head>
<body>
    <%pageContext.setAttribute("userName", "admin"); %>
    <%request.setAttribute("bookName", "《Java Web 程序设计任务教程》"); %>
    <%session.setAttribute("userName", "itheima"); %>
    <%application.setAttribute("bookName", "《Java 基础案例教程》"); %>
    表达式\${pageScope.userName}的值为：${pageScope.userName} <br />
    表达式\${requestScope.bookName}的值为：${requestScope.bookName} <br />
    表达式\${sessionScope.userName}的值为：${sessionScope.userName} <br />
    表达式\${applicationScope.bookName}的值为：${applicationScope.bookName}
     <br />
    表达式\${userName}的值为：${userName}
</body>
</html>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter13/scopes.jsp 访问 scopes.jsp 页面，浏览器窗口中显示的结果如图 13-2 所示。

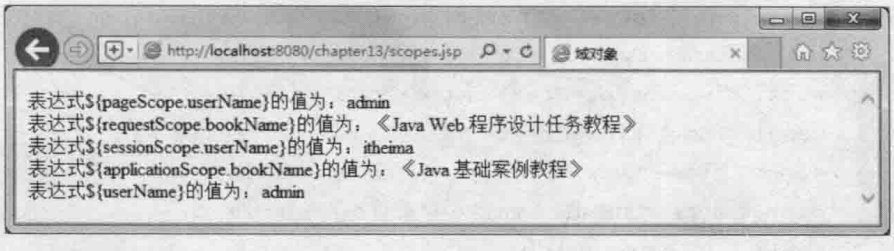


图 13-2　scopes.jsp 页面

从图 13-2 中可以看出，使用 pageScope、requestScope、sessionScope 和 applicationScope 这四个隐式对象成功获取到了相应 JSP 域对象中的属性值。

需要注意的是，使用 EL 表达式获取某个域对象中的属性时，也可以不使用这些隐式对象指定查找域，而是直接引用域中的属性名称，例如表达式 ${userName} 就是在 page、request、session、application 这四个作用域内按顺序依次查找 userName 属性的。

3. param 和 paramValues 对象

在 JSP 页面中，经常需要获取客户端传递的请求参数，为此，EL 表达式提供了 param 和 paramValues 两个隐式对象，这两个隐式对象专门用于获取客户端访问 JSP 页面时传递的请求参数。下面针对这两个对象进行讲解。

param 对象用于获取请求参数的某个值，它是 Map 类型，与 request.getParameter()方法相同，在使用 EL 获取参数时，如果参数不存在，则返回的是空字符串，而不是 null。param 对象的语法格式比较简单，具体示例如下：

```
${param.num}
```

如果一个请求参数有多个值，则可以使用 paramValues 对象获取请求参数的所有值，该对象用于返回请求参数所有值组成的数组，如果要获取某个请求参数的第一个值，则可以使用如下代码：

```
${paramValues.nums[0]}
```

为了让读者更好地学习这两个隐式对象，下面通过案例演示 param 和 paramValues 隐式对象如何获取请求参数的值。

在项目的 WebContent 目录下，新建一个名为 param.jsp 的文件，代码如文件 13-3 所示。

文件 13-3　param.jsp

```
<%@page language="java" contentType="text/html; charset=utf-8"
    pageEncoding="utf-8"%>
<html>
<head>
<title>param 和 paramValues</title>
</head>
<body style="text-align: center;">
    <form action="${pageContext.request.contextPath}/param.jsp">
        num1: <input type="text" name="num1"><br />
        num2: <input type="text" name="num"><br />
        num3: <input type="text" name="num"><br /><br />
        <input type="submit" value="提交" />  
        <input type="submit" value="重置" /><hr />
        num1: ${param.num1}<br />
        num2: ${paramValues.num[0]}<br />
        num3: ${paramValues.num[1]}<br />
    </form>
</body>
</html>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter13/param.jsp 访问 param.jsp 页面，浏览器窗口中会显示一个表单，在这个表单中输入三个数字，分别为 1、2、3，然后单击【提交】按钮，浏览器窗口中显示的结果如图 13-3 所示。

从图 13-3 中可以看出，输入的三个数字全部都在浏览器中显示出来了，这是因为在文件 13-3 中使用 param 对象获取的请求参数 num1 的值为 1，使用 paramValues 对象获取了同一个请求参数 num 的两个值，分别为 2 和 3。

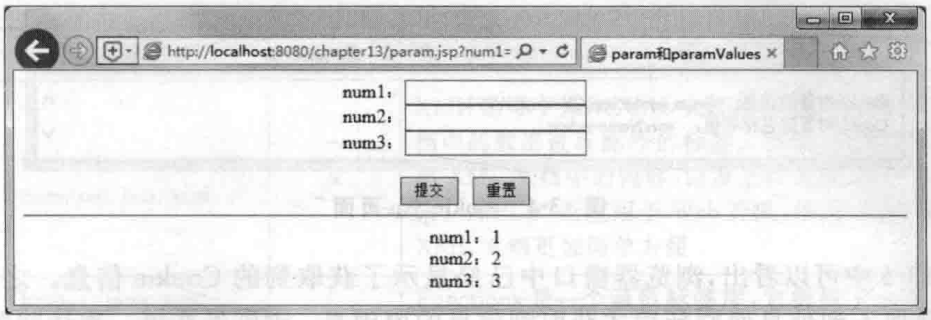


图 13-3　param.jsp 页面

需要注意的是，如果一个请求参数有多个值，那么在使用 param 获取请求参数时，会返回请求参数的第一个值。

4. cookie 对象

在 JSP 开发中，经常需要获取客户端的 Cookie 信息，为此，EL 表达式提供了 Cookie 隐式对象，该对象是一个代表所有 Cookie 信息的 Map 集合，Map 集合中元素的键为各个 Cookie 的名称，值则为对应的 Cookie 对象，具体示例如下：

```
获取 cookie 对象的信息：${cookie.userName}
获取 cookie 对象的名称：${cookie.userName.name}
获取 cookie 对象的值：${cookie.userName.value}
```

为了让读者更好地学习 Cookie 隐式对象，下面通过案例演示如何获取 Cookie 对象中的信息。在 chapter13 项目的 WebContent 目录下，新建一个名为 cookie.jsp 的文件，编辑代码如文件 13-4 所示。

文件 13-4　cookie.jsp

```
1  <%@page language="java" contentType="text/html; charset=utf-8"
2      pageEncoding="utf-8"%>
3  <html>
4  <head>
5  <title>cookie 对象</title>
6  </head>
7  <body>
8      <%response.addCookie(new Cookie("userName", "admin")); %>
9      Cookie 对象的信息：${cookie.userName } <br />
10     Cookie 对象的名称和值：${cookie.userName.name }=${cookie.userName.value }
11 </body>
12 </html>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter13/cookie.jsp 访问 cookie.jsp 页面，由于浏览器是第一次访问 cookie.jsp 页面，此时，服务器还没有接收到名为 userName 的 Cookie 信息，因此，在浏览器窗口中不会显示。接下来刷新浏览器，第二次访问 cookie.jsp 页面，此时浏览器窗口中显示的结果如图 13-4 所示。

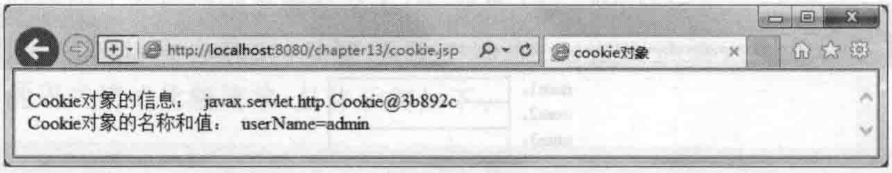


图 13-4 cookie.jsp 页面

从图 13-4 中可以看出，浏览器窗口中已经显示了获取到的 Cookie 信息。之所以出现首次访问获取不到信息而刷新后才获取到信息的原因是：浏览器在第一次访问服务器时，服务器会向浏览器回写一个 Cookie，此时的 Cookie 信息是存储在浏览器中的，当刷新浏览器，第二次访问 cookie.jsp 页面时，由于浏览器中已经存储了名为 userName 的 Cookie 信息，所以当再次访问相同资源时，浏览器会将此 Cookie 信息一同发送给服务器，这时使用表达式 ${cookie.userName.name} 和 ${cookie.userName.value} 便可以获取 Cookie 的名称和值。

13.2 JSTL

13.2.1 JSTL

JSTL 是 JSP Standard Tag Library 的简称，它是 Sun 公司制定的一套 JSP 标准标签库，这个标签库由五个不同功能的标签库共同组成。

JSTL 1.1 规范为这五个标签库分别指定了不同的 URI 以及建议使用的前缀，如表 13-6 所示。

表 13-6 JSTL 包含的标签库

标签库	标签库的 URI	前缀	描　述
Core	http://java.sun.com/jsp/jstl/core	c	Core 是一个核心标签库，它包含了实现 Web 应用中通用操作的标签。例如，用于输出文本内容的<c:out>标签、用于条件判断的<c:if>标签、用于迭代循环的<c:forEach>标签
I18N	http://java.sun.com/jsp/jstl/fmt	fmt	I18N 是一个国际化/格式化标签库，它包含实现 Web 应用程序的国际化标签和格式化标签。例如，设置 JSP 页面的本地信息、设置 JSP 页面的时区、使日期按照本地格式显示等
SQL	http://java.sun.com/jsp/jstl/sql	sql	SQL 是一个数据库标签库，它包含了用于访问数据库和对数据库中的数据进行操作的标签。例如，从数据库中获得数据库连接、从数据库表中检索数据等。在软件分层开发模型中，由于 JSP 页面仅作为表示层，一般不会在 JSP 页面中直接操作数据库，因此，JSTL 中提供的这套标签库不经常使用

续表

标签库	标签库的 URI	前缀	描　述
XML	http://java.sun.com/jsp/jstl/xml	x	XML 是一个操作 XML 文档的标签库，它包含对 XML 文档中的数据进行操作的标签。例如，解析 XML 文件、输出 XML 文档中的内容，以及迭代处理 XML 文档中的元素。XML 广泛应用于 Web 开发，使用 XML 标签库处理 XML 文档更加简单方便
Functions	http://java.sun.com/jsp/jstl/functions	fn	Functions 是一个函数标签库，它提供了一套自定义 EL 函数，包含了 JSP 网页制作者经常用到的字符串操作。例如，提取字符串中的子字符串、获取字符串的长度等

13.2.2　JSTL 的配置和测试

通过第 13.2.1 节的学习，读者已经了解了什么是 JSTL 以及 JSTL 中的五个标签库，然而要想在 JSP 页面中使用 JSTL，还需要配置 JSTL。下面分步骤讲解 JSTL 的配置和测试过程。

1. 下载 JSTL 包

读者可以从网址 http://archive.apache.org/dist/jakarta/taglibs/standard/binaries/ 中下载 JSTL 的安装包 jakarta-taglibs-standard-1.1.2.zip。将下载好的 JSTL 安装包进行解压后，可以在解压目录的 lib 文件夹中找到 jstl.jar 和 standard.jar，其中 jstl.jar 文件包含 JSTL 规范中定义的接口和相关类，standard.jar 文件包含用于实现 JSTL 的 class 文件以及 JSTL 中的五个标签库描述符文件(TLD)。

2. 引入 JAR 包

将 jstl.jar 和 standard.jar 复制到 chapter13 项目的 lib 目录下，如图 13-5 所示。

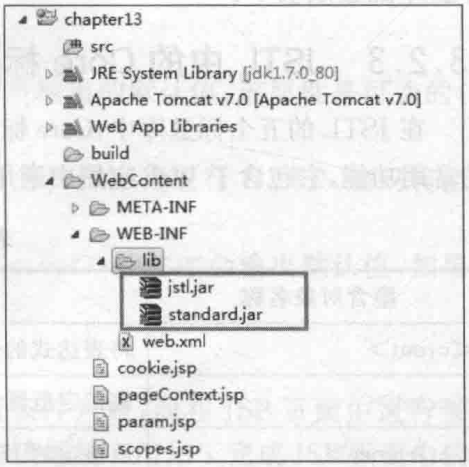


图 13-5　导入 jstl.jar 和 standard.jar 文件

3. 测试 JSTL

导入 JAR 包后，就可以在 JSP 文件中使用 JSTL 的标签库。下面以＜c:out＞标签为例演示 JSTL 标签的使用。

在 chapter13 项目的 WebContent 目录下创建一个 JSP 文件 test.jsp，在该文件中使用＜c:out＞标签输出“Hello World !”信息，编辑代码后如文件 13-5 所示。

文件 13-5　test.jsp

```
1  <%@page language="java" contentType="text/html; charset=utf-8"
2      pageEncoding="utf-8"%>
3  <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
4  <html>
5  <head>
```

```
6   <title>JSTL</title>
7   </head>
8   <body>
9       <c:out value="Hello World!"></c:out>
10  </body>
11  </html>
```

在上述代码中，第 3 行代码使用了 taglib 指令导入 Core 标签库，其中 taglib 指令的 uri 属性用于指定引入标签库描述符文件的 URI，prefix 属性用于指定引入标签库描述符文件的前缀，在 JSP 文件中使用这个标签库中的某个标签时，都需要使用这个前缀。

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter13/test.jsp 访问 test.jsp 页面，此时，浏览器窗口中显示的结果如图 13-6 所示。

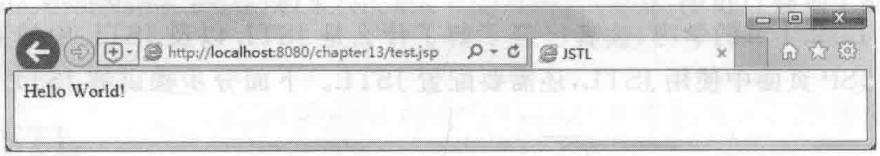


图 13-6　test.jsp 页面

从图 13-6 中可以看出，浏览器已经成功显示了“Hello World!”信息，由此可见，JSTL 标签库配置成功了。

13.2.3　JSTL 中的 Core 标签库

在 JSTL 的五个标签库中，Core 标签库是 JSTL 中的核心标签库，主要用于完成 JSP 页面的常用功能，它包含了 Web 应用中通用操作的标签，这些常用的标签及其作用如表 13-7 所示。

表 13-7　Core 标签库

隐含对象名称	描　　述
<c:out>	将表达式的值输出到 JSP 页面，相当于 JSP 表达式<%=表达式%>
<c:set>	在指定范围中定义变量，或者为指定的对象设置属性值
<c:remove>	从指定的 JSP 范围中移除指定的变量
<c:catch>	捕获程序中出现的异常，相当于 Java 中的 try...catch 语句
<c:import>	导入站内或其他网站的静态和动态文件到 Web 页面中
<c:redirect>	将客户端发出的 Request 请求重定向到其他资源
<c:url>	使用正确的 URL 重写规则构造一个 URL
<c:param>	为其他标签提供参数信息，通常与其他标签结合使用
<c:if>	根据不同的条件处理不同的业务，与 Java 语言的 if 语句类似
<c:choose>、<c:when>、<c:otherwise>	根据不同的条件完成指定的业务逻辑，如果没有符合的条件，则执行默认条件的业务逻辑，相当于 Java 中的 switch 语句
<c:forEach>	根据循环条件，遍历数组和集合类中的所有或者部分数据

表 13-7 列举了 Core 标签库中的主要标签，为了使读者能够更快地掌握这些标签的使用，下面以表中几个常用标签的使用为例进行详细讲解。

1. <c:out>标签

<c:out>标签可以将一段文本内容或表达式的结果输出到客户端，如果<c:out>标签输出的文本内容中包含需要进行转义的特殊字符(如>、<、&、'、" 等)，则<c:out>标签会默认将它们进行 HTML 编码转换后再输出。

<c:out>标签的使用分为有标签体和没有标签体两种情况，具体如下。

语法 1：有标签体的情况。

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

语法 2：没有标签体的情况。

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

在有标签体的情况下，在标签体中输出指定的默认值，在没有标签体的情况下，需要使用 default 属性指定默认值。从上述两个语法中可以看到<c:out>标签有多个属性，下面针对这些属性进行讲解。

- value 属性用于指定输出的文本内容。
- default 属性用于指定当 value 属性为 null 时所输出的默认值，该属性是可选的(方括号中的属性都是可选的)。
- escapeXml 属性用于指定是否将>、<、&、'、" 等特殊字符进行 HTML 编码转换后再进行输出，默认值为 true。

需要注意的是，只有当 value 属性值为 null 时，<c:out>标签才会输出默认值，如果没有指定默认值，则默认输出空字符串。

2. <c:if>标签

在程序开发过程中，我们经常会使用 if 语句进行条件判断，而在 JSP 页面中进行条件判断时，则可以使用 Core 标签库提供的<c:if>标签，该标签专门用于完成 JSP 页面中的条件判断，它有两种语法格式，具体如下：

语法 1：在有标签体的情况下，在标签体中指定要输出的内容。

```
<c:if test="testCondition" var="result"
[scope="{page|request|session|application}"]>
    body content
</c:if>
```

语法 2：在没有标签体的情况下。

```
<c:if test="testCondition" var="result"
[scope="{page|request|session|application}"]/>
```

在上述语法格式中，可以看到<c:if>标签有三个属性，下面将针对这三个属性进行讲解。

- test 属性用于设置逻辑表达式。
- var 属性用于指定逻辑表达式中变量的名称。
- scope 属性用于指定 var 变量的作用范围，默认值为 page。如果属性 test 的计算结果为 true，那么标签体将被执行，否则标签体不会被执行。

3. <c:choose>标签

在程序开发中，除了经常使用 if 条件语句以外，还经常会使用 if-else 语句。为了在 JSP 页面中也可以完成同样的功能，Core 标签库提供了<c:choose>标签，该标签用于指定多个条件选择的组合边界，它必须与<c:when>和<c:otherwise>标签一起使用。

<c:choose>标签是没有属性的，在使用时标签体中只能嵌套一个或多个<c:when>标签和零个或一个<c:otherwise>标签，并且同一个<c:choose>标签中所有的<c:when>子标签必须出现在<c:otherwise>子标签之前，其语法格式如下：

```
<c:choose>
    Body content(<when>and <otherwise>subtags)
</c:choose>
```

<c:when>标签只有一个 test 属性，该属性的值为布尔类型。test 属性支持动态值，其值可以是一个条件表达式，如果条件表达式的值为 true，则执行这个<c:when>标签体的内容，其语法格式如下：

```
<c:when test="testCondition">
    Body content
</c:when>
```

<c:otherwise>标签也没有属性，使用时必须作为<c:choose>标签的最后分支出现，当所有的<c:when>标签的 test 条件都不成立时，才执行和输出<c:otherwise>标签体的内容，其语法格式如下：

```
<c:otherwise>
    conditional block
</c:otherwise>
```

4. <c:forEach>标签

<c:forEach>标签专门用于迭代集合对象中的元素，如 Set、List、Map、数组等，它有两种语法格式，具体如下。

语法 1：迭代包含多个对象的集合。

```
<c:forEach [var="varName"] items="collection" [varStatus="varStatusName"]
[begin="begin"] [end="end"] [step="step"]>
    body content
</c:forEach>
```

语法 2：迭代指定范围内的集合。

```
<c:forEach [var="varName"] [varStatus="varStatusName"] begin="begin"
end="end" [step="step"]>
    body content
</c:forEach>
```

在上述语法格式中，可以看到<c:forEach>标签有多个属性，下面针对这些属性进行讲解。

- var 属性用于指定将当前迭代到的元素保存到 page 域中的名称。
- items 属性用于指定将要迭代的集合对象。
- varStatus 用于指定当前迭代状态信息的对象保存到 page 域中的名称。
- begin 属性用于指定从集合中第几个元素开始进行迭代，begin 的索引值从 0 开始，如果没有指定 items 属性，则从 begin 指定的值开始迭代，直到迭代结束为止。
- step 属性用于指定迭代的步长，即迭代因子的增量。

下面通过具体的案例学习<c:forEach>标签的使用。在 chapter13 项目的 WebContent 目录下创建一个名称为 foreach.jsp 的文件，在文件中分别使用<c:foreach>标签迭代数组和 Map 集合，如文件 13-6 所示。

文件 13-6　foreach.jsp

```
1  <%@page language="java" contentType="text/html; charset=utf-8"
2      pageEncoding="utf-8" import="java.util.*"%>
3  <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
4  <html>
5  <head>
6  <title>forEach 标签</title>
7  </head>
8  <body>
9      <%
10         String[] fruits={ "lemon", "orange", "apple", "pear" };
11     %>
12     String 数组中的元素：<br />
13     <c:forEach var="name" items="<%=fruits%>">
14         ${name}<br />
15     </c:forEach>
16     <%
17         Map userMap=new HashMap();
18         userMap.put("Jack", "123456");
19         userMap.put("Jim", "123456");
20         userMap.put("Lucy", "123456");
21     %>
22     <hr />
23     HashMap 集合中的元素：<br />
```

```
24      <c:forEach var="entry" items="<%=userMap%>">
25          ${entry.key}   ${entry.value}<br />
26      </c:forEach>
27  </body>
28  </html>
```

在文件 13-6 中，首先分别定义了一个包含元素的数组和集合，并分别使用<c:forEach>标签将数组和集合中的元素迭代出来。在迭代时，数组中的元素赋值给了<c:forEach>标签的 items 属性，而 Map 集合中的对象同样赋值给了 items 属性，之后使用 EL 表达式就可以获取迭代出来的数组和集合中的数据。

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter13/foreach.jsp 访问 foreach.jsp 页面，此时，浏览器窗口中的显示结果如图 13-17 所示。

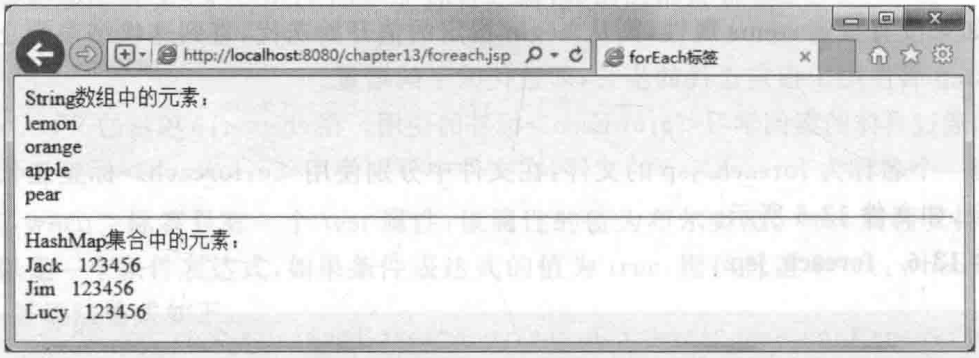


图 13-7 foreach.jsp 页面

从图 13-7 中可以看出，数组中存入的元素以及 Map 集合中存入的用户名和密码全部被显示出来了。在使用<c:forEach>标签时，只需将 userMap 集合对象赋值给 items 属性，之后通过 entry 变量就可以获取集合中的键和值。

5. <c:param>标签和<c:url>标签

<c:param>标签用于在 URL 地址中附加参数，它通常嵌套在<c:url>标签内使用。<c:param>标签有两种语法格式，具体如下。

语法 1：使用 value 属性指定参数的值。

```
<c:param name="name" value="value">
```

语法 2：在标签体中指定参数的值。

```
<c:param name="name">
    parameter value
</c:param>
```

从上述语法格式中可以看出，<c:param>标签有两个属性。下面针对这两个属性进行讲解。

- name 属性用于指定参数的名称。

- value 属性用于指定参数的值，当使用<c:param>标签为一个 URL 地址附加参数时，它会自动对参数值进行 URL 编码，例如，如果传递的参数值为“北京”，则将其转换为“%e5%8c%97%e4%ba%ac”后再附加到 URL 地址后面。

<c:url>标签可以在 JSP 页面中构造一个新的地址，实现 URL 的重写。<c:url>标签有两种语法格式，具体如下。

语法 1：在有标签实体的情况下，在标签体中指定构造 URL 参数。

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

语法 2：在没有标签实体的情况下。

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

在上述语法格式中，可以看到<c:url>标签中有多个属性，这些属性的含义具体如下。

- value 属性用于指定构造的 URL。
- context 属性用于指定导入同一个服务器下其他 Web 应用的名称。
- var 属性用于指定将构造的 URL 地址保存到域对象的属性名称。
- scope 属性用于指定将构造好的 URL 保存到域对象中。

为了使读者更好地学习<c:url>标签，下面通过具体的案例演示该标签的使用。在 chapter13 项目的 WebContent 目录下新建一个名为 c_url.jsp 的文件，编辑代码后如文件 13-7 所示。

文件 13-7　c_url.jsp

```
1  <%@page language="java" contentType="text/html; charset=utf-8"
2  pageEncoding="utf-8" import="java.util.*"%>
3  <%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
4  <html>
5  <head>
6  <title>param 和 url 标签</title>
7  </head>
8  <body>
9      使用绝对路径构造 URL:<br />
10     <c:url var="myURL" value="http://localhost:8080/chapter13/info.jsp">
11         <c:param name="user" value="张三" />
12         <c:param name="addr" value="北京" />
13     </c:url>
14     <a href="${myURL}">info.jsp</a><br />
15     使用相对路径构造 URL:<br />
16     <c:url var="myURL" value="login.jsp? username=admin&pwd=123456" />
```

```
17      <a href="${ myURL}">login.jsp</a>
18  </body>
19  </html>
```

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter13/c_url.jsp 访问 c_url.jsp 页面，此时，浏览器窗口中显示的结果如图 13-8 所示。

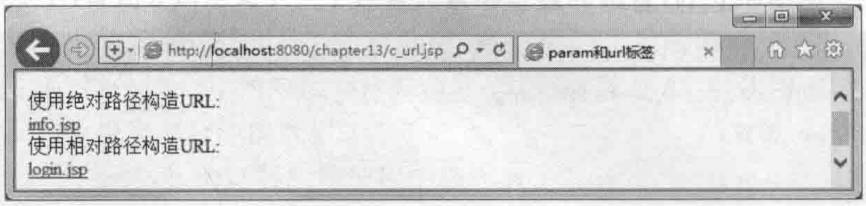


图 13-8　c_url.jsp 页面

从图 13-8 中可以看出，在浏览器窗口中已经显示了 c_url.jsp 页面的内容，此时查看该页面的源文件，如图 13-9 所示。

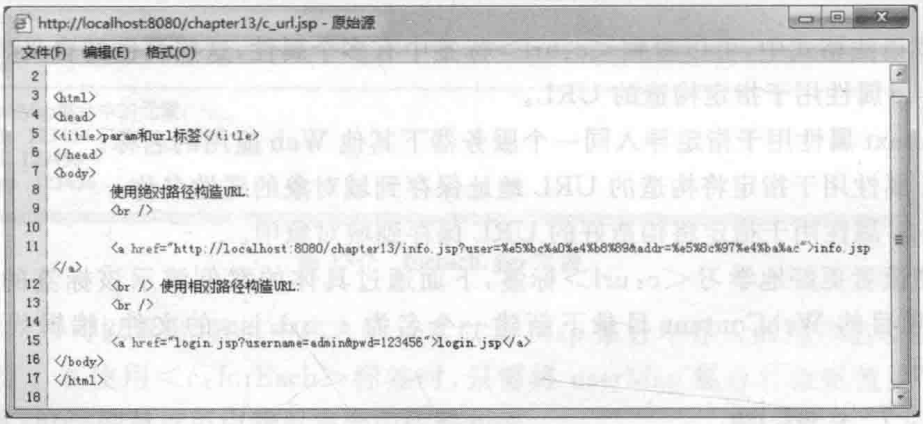


图 13-9　源文件信息

从图 13-9 中可以看出，在 c_url.jsp 中的 URL 地址会变成一个超链接，并且使用＜param＞标签指定的中文参数进行了 URL 编码，将参数“张三”转换为“%e5%bc%a0%e4%b8%89”，将参数“北京”转换为“%e5%8c%97%e4%ba%ac”，这样就构造了一个新的 URL 地址，从而实现了 URL 的重写功能。

第 14 章
Servlet高级

Filter 是 Servlet 的过滤器，是 Servlet 2.3 规范中新增加的一个功能，主要用于完成一些通用的操作，如编码的过滤、判断用户的登录状态等。Listener 是 Servlet 的监听器，它可以监听客户端的请求、服务器端的操作等，通过监听器，可以自动激发一些操作。如监听域对象的生命周期、统计当前在线人数等。本章将针对 Filter 和 Listener 进行详细讲解。

14.1 Filter

14.1.1 Filter 概述

在现实生活中，人们可以使用污水净化设备对水源进行过滤净化。同样，在程序中人们也可以使用 Filter 对请求和响应信息进行过滤。Filter 被称为过滤器，其主要作用是对 Servlet 容器调用 Servlet 的过程进行拦截，从而在 Servlet 进行响应处理的前后实现一些特殊功能。Filter 在 Web 应用中的拦截过程如图 14-1 所示。

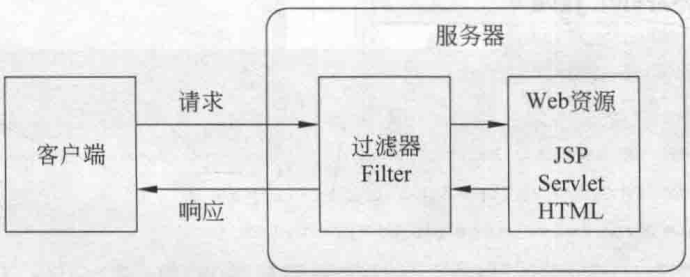


图 14-1 Filter 拦截过程

在图 14-1 中，当用户通过浏览器访问服务器中的目标资源时，首先会被 Filter 拦截，在 Filter 中进行预处理操作，然后再将请求转发给目标资源。当服务器接收到这个请求后会对其进行响应，在服务器处理响应的过程中，也需要将响应结果经过滤器处理后，才发送给客户端。

本质上，Filter 过滤器就是一个实现了 javax.servlet.Filter 接口的类，在 javax.servlet.Filter 接口中定义了三个方法，如表 14-1 所示。

表 14-1 中的三个方法都是可以表现 Filter 生命周期的方法，其中 init()方法在 Web 应用程序加载时会被调用，destroy()方法在 Web 应用程序卸载(或关闭)时被调用，这两个方法都只会被调用一次，而 doFilter()方法会被调用多次(只要客户端有请求时就会被调用)，Filter 所有的工作集中在 doFilter()方法中。

表 14-1 Filter 接口中的方法

方法声明	功能描述
init(FilterConfig filterConfig)	init()方法用于初始化过滤器，开发人员可以在 init()方法中完成与构造方法类似的初始化功能，如果初始化代码中要使用到 FilterConfig 对象，那么，这些初始化代码就只能在 Filter 的 init()方法中编写，而不能在构造方法中编写
doFilter (ServletRequest request, ServletResponse response, FilterChain chain)	doFilter()方法有多个参数，其中，参数 request 和 response 为 Web 服务器或 Filter 链中的上一个 Filter 传递过来的请求和响应对象；参数 chain 代表当前 Filter 链的对象，只有在当前 Filter 对象中的 doFilter()方法内部需要调用 FilterChain 对象的 doFilter()方法，才能把请求交付给 Filter 链中的下一个 Filter 或者目标程序处理
destroy()	destroy()方法在 Web 服务器卸载 Filter 对象之前被调用，该方法用于释放被 Filter 对象打开的资源，例如关闭数据库和 I/O 流

14.1.2 实现第一个 Filter 程序

为了帮助读者快速了解 Filter 的开发过程，下面通过案例演示 Filter 程序如何对 Servlet 程序的调用过程进行拦截。

1. 创建 Servlet

在 Eclipse 中创建一个名为 chapter14 的 Web 项目，并在项目的 src 下创建一个名为 com.itheima.filter 的包，在该包中创建一个名为 MyServlet 的 Servlet 类，该类用于访问时在浏览器中输出“Hello MyServlet”。MyServlet 类的实现代码如文件 14-1 所示。

文件 14-1 MyServlet.java

```
1  package com.itheima.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class MyServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7      HttpServletResponse response)
8              throws ServletException, IOException {
9        response.getWriter().write("Hello MyServlet ");
10     }
11     public void doPost(HttpServletRequest request,
12                        HttpServletResponse response)
13         throws ServletException, IOException {
14     doGet(request, response);
15   }
16 }
```

2. 创建 Filter 类

右击 com.itheima.filter 包，然后选择 New→Other...，在弹出窗口中的 Web 文件夹下

找到 Filter，单击 Next 按钮，在新窗口的 Class name 文本框中填写所创建的 Filter，单击 Next 按钮进入 Filter 映射信息的配置窗口，如图 14-2 所示。

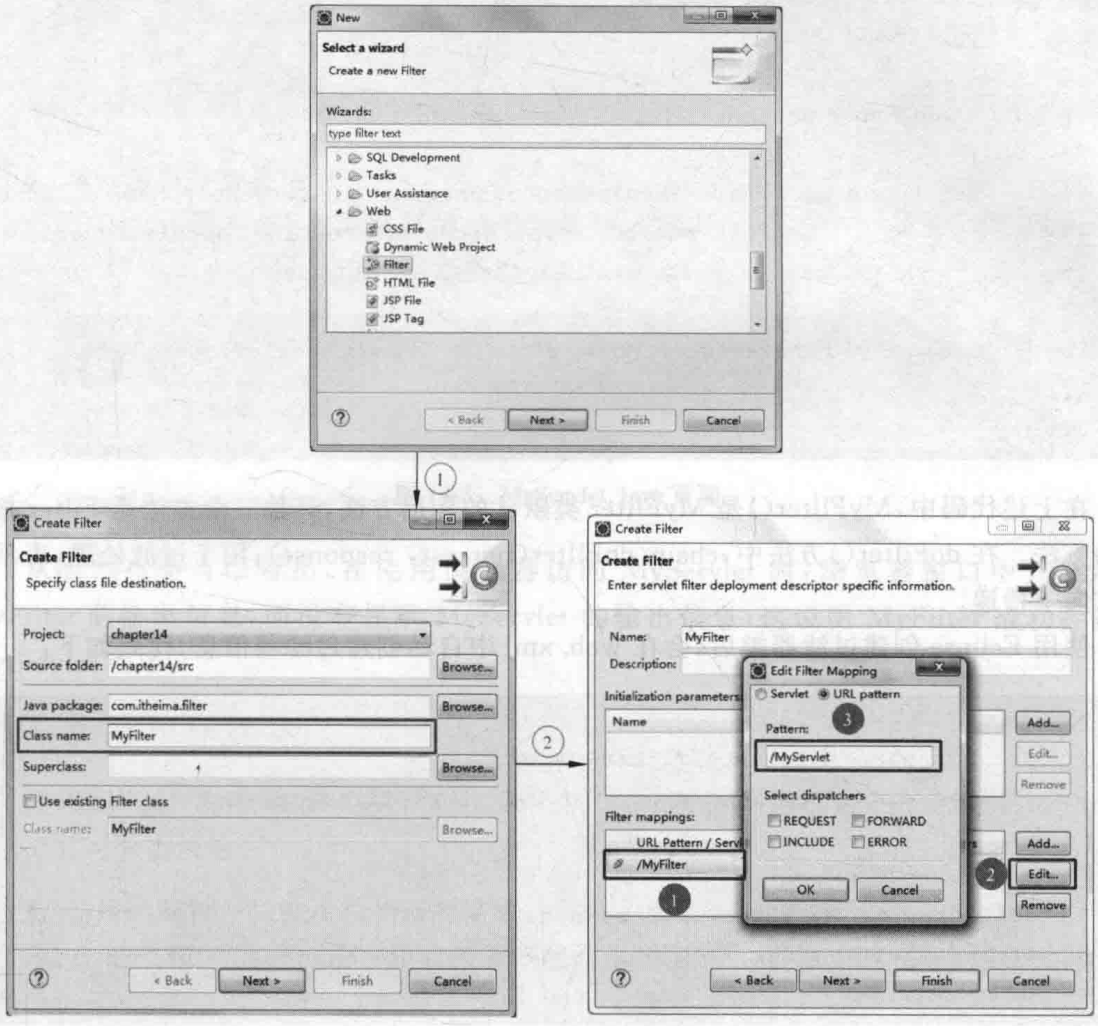


图 14-2　创建一个 Filter

在图 14-2 中，单击 Filter Mappings 窗口中的/MyFilter，然后单击右侧的 Edit 按钮，在弹出窗口中的 Pattern 文本框中将/MyFilter 更改为/MyServlet(/MyFilter 是默认拦截的 URL，/MyServlet 是需要拦截的 URL，这里指 MyServlet 类)。

依次单击 OK 和 Finish 按钮即可完成 Filter 类的创建。创建后的 MyFilter 类的主要代码如文件 14-2 所示。

文件 14-2　MyFilter

```
1  package com.itheima.filter;
2  import java.io.IOException;
3  import javax.servlet.Filter;
4  import javax.servlet.FilterChain;
5  import javax.servlet.FilterConfig;
6  import javax.servlet.ServletException;
```

```
7  import javax.servlet.ServletRequest;
8  import javax.servlet.ServletResponse;
9  public class MyFilter implements Filter {
10     public MyFilter() {
11     }
12     public void destroy() {
13     }
14     public void doFilter(ServletRequest request, ServletResponse response,
15                     FilterChain chain)throws IOException, ServletException {
16         chain.doFilter(request, response);
17     }
18     public void init(FilterConfig fConfig) throws ServletException {
19     }
20 }
```

在上述代码中，MyFilter()是 MyFilter 类默认的构造方法，其他三个方法是 Filter 接口中的方法。在 doFilter()方法中，chain. doFilter(request, response);用于过滤处理，表示将请求向下传递。

使用 Eclipse 创建过滤器类后，会在 web. xml 中自动创建过滤器信息，代码如下：

```
<filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>com.itheima.filter.MyFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/MyServlet</url-pattern>
</filter-mapping>
```

在上述代码中，设置了过滤器对/MyServlet 请求资源进行拦截，将在请求到达 MyServlet 程序前执行 MyFilter 程序。过滤器的配置信息中包含多个元素，这些元素分别具有不同的作用。

(1) <filter>根元素用于注册一个 Filter。

(2) <filter-name>子元素用于设置 Filter 名称。

(3) <filter-class>子元素用于设置 Filter 类的完整名称。

(4) <filter-mapping>根元素用于设置一个过滤器所拦截的资源。

(5) <filter-name>子元素必须与<filter>中的<filter-name>子元素相同。

(6) <url-pattern>子元素用于匹配用户请求的 URL，例如/MyServlet，这个 URL 还可以使用通配符 * 表示，例如 * . do 适用于所有以. do 结尾的 Servlet 路径。

3. 修改 Filter

为了演示 Filter 的拦截效果，对文件 14-2 中的 doFilter()方法进行修改，修改后的代码如下：

```
public void doFilter(ServletRequest request,ServletResponse response,
    FilterChain chain) throws IOException, ServletException {
        PrintWriter out=response.getWriter();
        out.write("Hello MyFilter");
}
```

4. 运行项目并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter14/MyServlet 访问 MyServlet，此时，浏览器窗口显示的结果如图 14-3 所示。

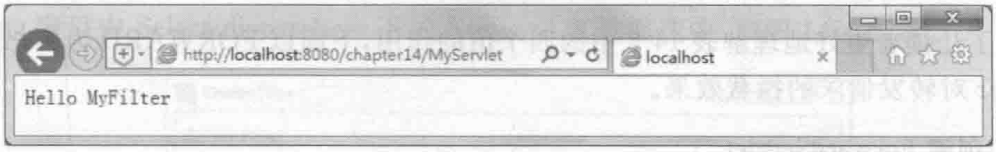


图 14-3　MyServlet.java 页面

从图 14-3 中可以看出，在使用浏览器访问 MyServlet 时，浏览器窗口中只显示了 MyFilter 的输出信息，而没有显示 MyServlet 的输出信息，这说明 MyFilter 成功拦截了 MyServlet 程序。

14.1.3　Filter 映射

使用 Eclipse 创建 Filter 后，在 web.xml 中的对应配置信息就是 Filter 映射。Filter 的映射方式可分为两种。

1. 使用通配符 * 拦截用户的所有请求

Filter 的＜filter-mapping＞元素用于配置过滤器拦截的资源信息，如果想让过滤器拦截所有的请求，那么可以使用通配符 * 实现，具体实现方式如下：

```
<filter>
    <filter-name>MyFilter</filter-name>
    <filter-class>com.itheima.filter.MyFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>MyFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

2. 拦截不同方式的访问请求

在 web.xml 文件中，每一个＜filter-mapping＞元素都可以配置一个 Filter 所负责拦截的资源。在＜filter-mapping＞元素中有一个特殊的子元素＜dispatcher＞，该元素用于指定过滤器所拦截的资源被 Servlet 容器调用的方式。＜dispatcher＞元素的值共有四个，如表 14-2 所示。

表 14-2 <dispatcher>元素的值

名 称	功能描述
REQUEST	当用户直接访问页面时，Web 容器将会调用过滤器。如果目标资源通过 RequestDispatcher 的 include()或 forward()方法访问，那么该过滤器将不会被调用
INCLUDE	如果目标资源通过 RequestDispatcher 的 include()方法访问，那么该过滤器将会被调用。除此之外，该过滤器不会被调用
FORWARD	如果目标资源通过 RequestDispatcher 的 forward()方法访问，那么该过滤器将会被调用。除此之外，该过滤器不会被调用
ERROR	如果目标资源通过声明式异常处理机制调用，那么该过滤器将会被调用。除此之外，该过滤器不会被调用

为了让读者更好地理解表 14-2 中的四个值的作用，下面以 FORWARD 为例，分步骤演示 Filter 对转发请求的拦截效果。

1. 创建 ForwardServlet

在 chapter14 项目的 com.itheima.filter 包中创建一个名为 ForwardServlet 的 Servlet 类，该类用于将请求转发给 first.jsp 页面，如文件 14-3 所示。

文件 14-3 ForwardServlet.java

```
1  package com.itheima.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class ForwardServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7          HttpServletResponse response)throws ServletException,
8              IOException {
9          request.getRequestDispatcher("/first.jsp").forward(request,response);
10     }
11     public void doPost(HttpServletRequest request,
12         HttpServletResponse response)throws ServletException, IOException {
13         doGet(request, response);
14     }
15 }
```

2. 创建 first.jsp

在项目的 WebContent 目录中创建一个名称为 first.jsp 的页面，该页面用于输出内容，编辑代码如文件 14-4 所示。

文件 14-4 first.jsp

```
1  <%@page language="java" contentType="text/html; charset=utf-8"
2      pageEncoding="utf-8"%>
3  <html>
4  <head>
```

```
<title>first</title>
</head>
<body>
    first.jsp
</body>
</html>
```

3. 创建并编辑过滤器 ForwardFilter

在 com.itheima.filter 包中创建一个名称为 ForwardFilter 的过滤器，该过滤器专门用于对 first.jsp 页面进行拦截。需要注意的是，在填写配置信息时，需要将 Edit Filter Mapping 窗口中 Select dispatchers 下的 Forward 复选框选中，如图 14-4 所示。

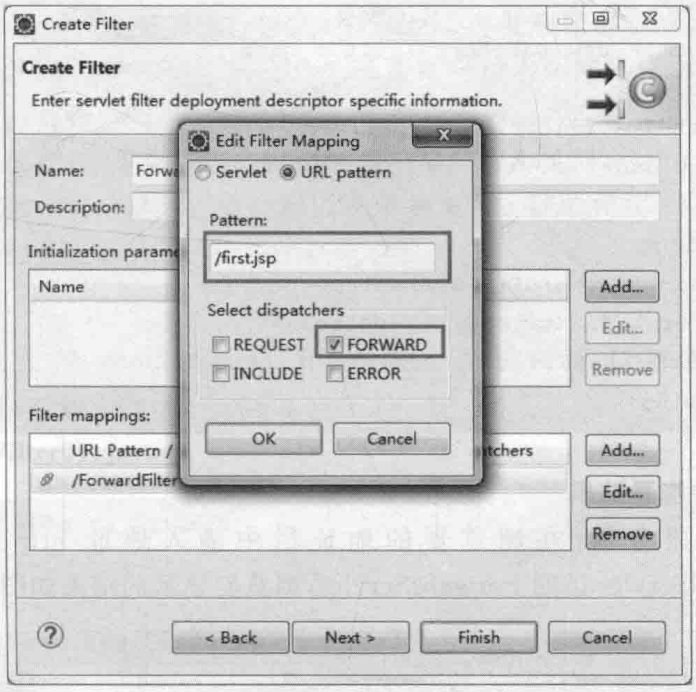


图 14-4　配置拦截信息

创建完成后，编辑 ForwardFilter 类。编辑后的实现代码如文件 14-5 所示。

文件 14-5　ForwardFilter.java

```
package com.itheima.filter;
import java.io.*;
import javax.servlet.*;
public class ForwardFilter implements Filter {
    public void init(FilterConfig fConfig) throws ServletException {
        //过滤器对象在初始化时调用,可以配置一些初始化参数
    }
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
```

```
        //用于拦截用户的请求,如果和当前过滤器的拦截路径匹配,该方法会被调用
        PrintWriter out=response.getWriter();
        out.write("Hello FilterTest");
    }
    public void destroy() {
        //过滤器对象在销毁时自动调用,释放资源
    }
}
```

此时如果查看 web.xml 文件,会发现在 ForwardFilter 映射信息中增加了一个<dispatcher>子元素,并且该元素的值为 FORWARD,如下所示:

```
<filter>
    <filter-name>ForwardFilter</filter-name>
    <filter-class>
        com.itheima.filter.ForwardFilter
    </filter-class>
</filter>
<filter-mapping>
    <filter-name>ForwardFilter</filter-name>
    <url-pattern>/first.jsp</url-pattern>
    <dispatcher>FORWARD</dispatcher>
</filter-mapping>
```

4. 运行项目并查看结果

启动 Tomcat 服务器,在浏览器的地址栏中输入地址 http://localhost:8080/chapter14/ForwardServlet 访问 ForwardServlet,浏览器显示的结果如图 14-5 所示。

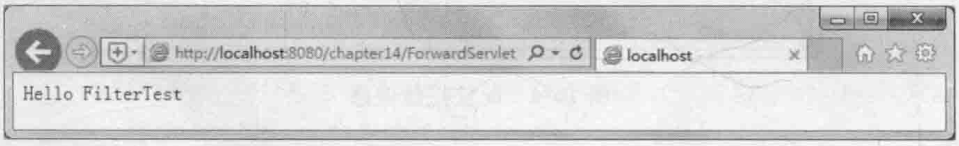


图 14-5 运行结果

从图 14-5 中可以看出,浏览器窗口显示的是 ForwardFilter 类中的内容,而 first.jsp 页面的输出内容没有显示。由此可见,ForwardServlet 中通过 forward()方法转发的 first.jsp 页面被成功拦截。

14.1.4 Filter 链

在一个 Web 应用程序中可以注册多个 Filter 程序,每个 Filter 程序都可以针对某一个 URL 进行拦截。如果多个 Filter 程序都对同一个 URL 进行拦截,那么这些 Filter 就会组成一个 Filter 链(也称过滤器链)。Filter 链用 FilterChain 对象表示,FilterChain 对象中有一个 doFilter()方法,该方法的作用是让 Filter 链上的当前过滤器放行,使请求进入下一个

Filter。

Filter 链的拦截过程如图 14-6 所示。

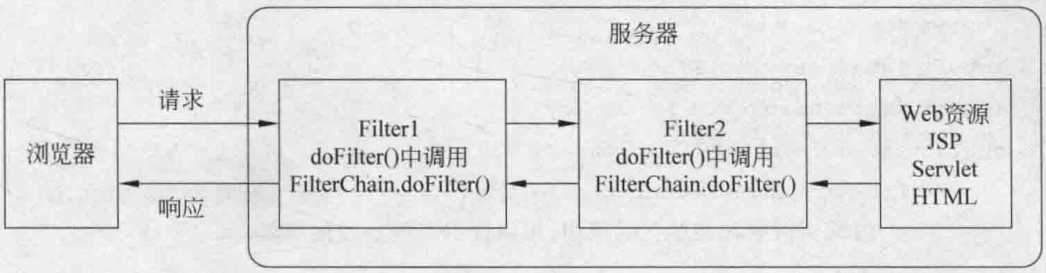


图 14-6　Filter 链

在图 14-6 中，当浏览器访问 Web 服务器中的资源时，需要经过两个过滤器 Filter1 和 Filter2。首先 Filter1 会对这个请求进行拦截，在 Filter1 中处理完请求后，通过调用 Filter1 的 doFilter()方法将请求传递给 Filter2，Filter2 处理用户请求后同样调用 doFilter()方法，最终将请求发送给目标资源。当 Web 服务器对这个请求做出响应时，也会被过滤器拦截，但这个拦截顺序与之前相反，最终将响应结果发送给客户端浏览器。

为了便于读者理解 Filter 链的拦截过程以及掌握 Filter 链的使用，下面通过案例演示如何使用 Filter 链拦截 MyServlet 的同一个请求。

1. 创建过滤器

在 chapter14 项目的 com.itheima.filter 包中新建两个过滤器 MyFilter01 和 MyFilter02，如文件 14-6 和文件 14-7 所示。

文件 14-6　MyFilter01.java

```
1  package com.itheima.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  public class MyFilter01 implements Filter {
5      public void init(FilterConfig fConfig) throws ServletException {
6          //过滤器对象在初始化时调用,可以配置一些初始化参数
7      }
8      public void doFilter(ServletRequest request, ServletResponse response,
9              FilterChain chain) throws IOException, ServletException {
10         //用于拦截用户的请求,如果和当前过滤器的拦截路径匹配,则该方法会被调用
11         PrintWriter out=response.getWriter();
12         out.write("MyFilter01<br />");
13         chain.doFilter(request, response);
14     }
15     public void destroy() {
16         //过滤器对象在销毁时自动调用,释放资源
17     }
18 }
```

文件 14-7 MyFilter02.java

```
package com.itheima.filter;
import java.io.*;
import javax.servlet.Filter;
import javax.servlet.*;
public class MyFilter02 implements Filter {
    public void init(FilterConfig fConfig) throws ServletException {
        //过滤器对象在初始化时调用,可以配置一些初始化参数
    }
    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        //用于拦截用户的请求,如果和当前过滤器的拦截路径匹配,则该方法会被调用
        PrintWriter out=response.getWriter();
        out.write("MyFilter02 Before<br />");
        chain.doFilter(request, response);
        out.write("<br />MyFilter02 After<br />");
    }
    public void destroy() {
        //过滤器对象在销毁时自动调用,释放资源
    }
}
```

2. 修改 web.xml

为了防止其他过滤器影响此次 Filter 链的演示效果,需要先将 web.xml 文件中的其他过滤器的配置信息注释掉,然后将 MyFilter01 和 MyFilter02 过滤器的映射信息配置在 MyServlet 配置信息前面,具体如下所示。

```
<filter>
    <filter-name>MyFilter01</filter-name>
    <filter-class>com.itheima.filter.MyFilter01</filter-class>
</filter>
<filter-mapping>
    <filter-name>MyFilter01</filter-name>
    <url-pattern>/MyServlet</url-pattern>
</filter-mapping>
<filter>
    <filter-name>MyFilter02</filter-name>
    <filter-class>com.itheima.filter.MyFilter02</filter-class>
</filter>
<filter-mapping>
    <filter-name>MyFilter02</filter-name>
```

```
    <url-pattern>/MyServlet</url-pattern>
</filter-mapping>
<servlet>
    <servlet-name>MyServlet</servlet-name>
    <servlet-class>com.itheima.filter.MyServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>MyServlet</servlet-name>
    <url-pattern>/MyServlet</url-pattern>
</servlet-mapping>
```

3. 运行项目并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入 http://localhost:8080/chapter14/MyServlet，此时，浏览器窗口中的显示结果如图 14-7 所示。

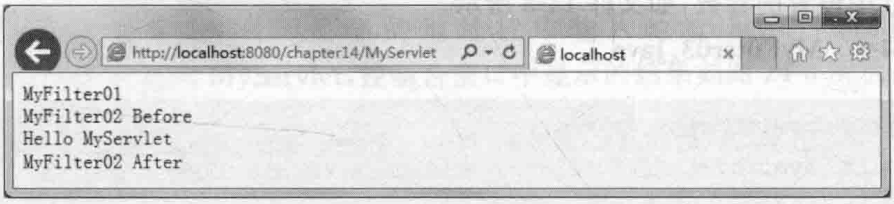


图 14-7 运行结果

从图 14-7 中可以看出，MyServlet 首先被 MyFilter01 拦截了，显示出 MyFilter01 中的内容，然后被 MyFilter02 拦截，直到 MyServlet 被 MyFilter02 放行后，浏览器才显示出 MyServlet 中的输出内容。

需要注意的是，Filter 链中各个 Filter 的拦截顺序与它们在 web.xml 文件中<filter-mapping>元素的映射顺序一致，由于 MyFilter01 的<filter-mapping>元素位于 MyFilter02 的<filter-mapping>元素前面，因此，用户的访问请求首先会被 MyFilter01 拦截，然后再被 MyFilter02 拦截。

14.1.5 FilterConfig 接口

FilterConfig 是 Servlet API 提供的一个用于获取 Filter 程序在 web.xml 文件中的配置信息的接口，该接口封装了 Filter 程序在 web.xml 中的所有注册信息，并且提供了一系列获取这些配置信息的方法，具体如表 14-3 所示。

表 14-3 FilterConfig 接口中的方法

方法声明	功能描述
String getFilterName()	getFilterName()方法用于返回在 web.xml 文件中为 Filter 所设置的名称，也就是返回<filter-name>元素的设置值
String getInitParameter(String name)	getInitParameter(String name)方法用于返回在 web.xml 文件中为 Filter 所设置的某个名称的初始化参数值，如果指定名称的初始化参数不存在，则返回 null

续表

方法声明	功能描述
Enumeration getInitParameterNames()	getInitParameterNames()方法用于返回一个 Enumeration 集合对象，该集合对象包含在 web.xml 文件中为当前 Filter 设置的所有初始化参数的名称
ServletContext getServletContext()	getServletContext()方法用于返回 FilterConfig 对象中所包装的 ServletContext 对象的引用

表 14-3 列举了 FilterConfig 接口中的一系列方法，为了让读者更好地掌握这些方法，下面以 getInitParameter(String name)方法的使用为例演示 FilterConfig 接口的使用。

1. 创建过滤器

在 chapter14 项目的 com.itheima.filter 包中创建过滤器 MyFilter03，使用该过滤器获取 web.xml 中设置的参数，如文件 14-8 所示。

文件 14-8 MyFilter03.java

```
1  package com.itheima.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  public class MyFilter03 implements Filter {
5      private String characterEncoding;
6      FilterConfig fc;
7      public void init(FilterConfig fConfig) throws ServletException {
8          //获取 FilterConfig 对象
9          this.fc=fConfig;
10     }
11     public void doFilter(ServletRequest request, ServletResponse response,
12             FilterChain chain) throws IOException, ServletException {
13         //输出参数信息
14         characterEncoding=fc.getInitParameter("encoding");
15         System.out.println("encoding 初始化参数的值为："+characterEncoding);
16         chain.doFilter(request, response);
17     }
18     public void destroy() {
19     }
20 }
```

2. 配置过滤器

在 web.xml 文件中配置过滤器信息。为了防止其他过滤器影响 MyFilter03 的拦截效果，需要先将 web.xml 文件中其他过滤器的配置信息注释掉，然后将 MyFilter03 过滤器的映射信息配置在 MyServlet 配置信息的前面，具体代码如下所示：

```
<filter>
    <filter-name>MyFilter03</filter-name>
    <filter-class>com.itheima.filter.MyFilter03</filter-class>
    <init-param>
        <param-name>encoding</param-name>
        <param-value>UTF-8</param-value>
    </init-param>
</filter>
<filter-mapping>
    <filter-name>MyFilter03</filter-name>
    <url-pattern>/MyServlet</url-pattern>
</filter-mapping>
```

3. 运行项目并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter14/MyServlet 访问 MyServlet，控制台窗口中显示的结果如图 14-8 所示。

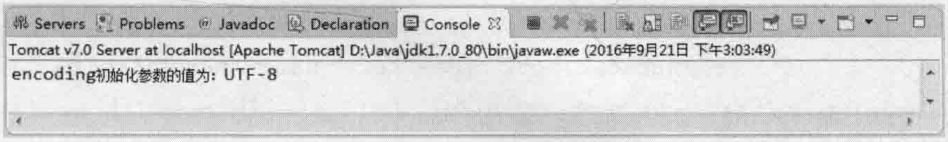


图 14-8　控制台窗口

从图 14-8 中可以看出，使用 Filter 成功获取到配置文件中的初始化参数。当 Tomcat 服务器启动时，会加载所有的 Web 应用，在加载 chapter14 时，MyFilter03 会被初始化调用 init()方法，从而可以得到 FilterConfig 对象，然后在doFilter()方法中通过调用 FilterConfig 对象的 getInitParameter()方法便可以获取在 web.xml 文件中配置的某个参数信息。

14.1.6　统一全站编码

在开发 Web 项目时，解决中文乱码问题是不可避免的。在前面所学的知识中，解决乱码的通常做法是在 Servlet 程序中设置编码方式，但是，当多个 Servlet 程序都需要设置编码方式时，就会书写大量重复的代码。为了解决这一问题，我们可以在 Filter 中对获取到的请求和响应消息进行编码处理，这样就可以实现全站编码方式的统一。本节将分步骤演示如何使用 Filter 实现全站编码的统一。

1. 编写 form.jsp 页面

在 chapter14 项目的 WebContent 目录中编写一个 form.jsp 页面，该页面用于提交用户登录的表单信息，如文件 14-9 所示。

文件 14-9　form.jsp

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

```
<html>
<head>
<title>用户登录</title>
</head>
<center>
    <h3>用户登录</h3>
</center>
<body style="text-align: center;">
<a href="<%=request.getContextPath()%>/CharacterServlet? name=张三 &password=
1234">
    单击超链接登录
</a>
<form action="<%=request.getContextPath()%>/CharacterServlet"
    method="post">
    <table border="1" width="600px" cellpadding="0" cellspacing="0"
        align="center">
        <tr>
            <td height="30" align="center">用户名：</td>
            <td> <input type="text" name="name" /></td>
        </tr>
        <tr>
            <td height="30" align="center">密   码：</td>
            <td> <input type="password" name="password" /></td>
        </tr>
        <tr>
            <td height="30" colspan="2" align="center">
                <input type="submit" value="登录" />
                        
                <input type="reset" value="重置" />
            </td>
        </tr>
    </table>
</form>
</body>
<html>
```

在文件 14-9 中，第 11～14 行代码是一个请求方式为 GET 的超链接；第 15～35 行代码是一个 POST 提交方式 form 表单。

2. 创建 Servlet

在 chapter14 项目的 com. itheima. filter 包中创建一个名称为 CharacterServlet 的 Servlet 类，该类用于获取用户输入的请求参数，并将参数输出到控制台，其代码如文件 14-10 所示。

文件 14-10　CharacterServlet.java

```
1  package com.itheima.filter;
2  import java.io.IOException;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class CharacterServlet extends HttpServlet {
6      public void doGet(HttpServletRequest request,
7              HttpServletResponse response)throws ServletException, IOException {
8          System.out.println(request.getParameter("name"));
9          System.out.println(request.getParameter("password"));
10     }
11     public void doPost(HttpServletRequest request,
12         HttpServletResponse response)throws ServletException, IOException {
13         doGet(request, response);
14     }
15 }
```

3. 创建过滤器

在 chapter14 项目的 com.itheima.filter 包中创建一个名称为 CharacterFilter 的 Filter 类，该类用于拦截用户的请求访问，并实现全站编码的统一，其具体实现代码如文件 14-11 所示。

文件 14-11　CharacterFilter.java

```
1  package com.itheima.filter;
2  import java.io.*;
3  import javax.servlet.*;
4  import javax.servlet.http.*;
5  public class CharacterFilter implements Filter {
6      public void init(FilterConfig filterConfig) throws ServletException {
7      }
8      public void doFilter(ServletRequest req, ServletResponse resp,
9              FilterChain chain) throws IOException, ServletException {
10         HttpServletRequest request=(HttpServletRequest) req;
11         HttpServletResponse response=(HttpServletResponse) resp;
12         //拦截所有的请求,解决全站中文乱码,指定 request 和 response 的编码
13         request.setCharacterEncoding("utf-8");      //只对消息体有效
14         response.setContentType("text/html;charset=utf-8");
15         //对 request 进行包装
16         CharacterRequest characterRequest=new CharacterRequest(request);
17         chain.doFilter(characterRequest, response);
18     }
19     public void destroy() {
```

```
    }
}
//继承默认包装类 HttpServletRequestWrapper
class CharacterRequest extends HttpServletRequestWrapper {
    private HttpServletRequest request;
    public CharacterRequest(HttpServletRequest request) {
        super(request);
        this.request=request;
    }
    //子类继承父类一定会覆写一些方法,此处用于重写 getParamter()方法
    public String getParameter(String name) {
        //调用被包装对象的 getParameter()方法,获得请求参数
        String value=super.getParameter(name);
        if (value==null){
            return null;
        }
        String method=super.getMethod();      //判断请求方式
        if ("get".equalsIgnoreCase(method)) {
            try {
                value=new String(value.getBytes("iso-8859-1"), "utf-8");
            } catch (UnsupportedEncodingException e) {
                throw new RuntimeException(e);
            }
        }
        return value;              //解决乱码后返回结果
    }
}
```

在文件 14-11 中,针对请求的方式不同,采用了不同的乱码解决方式。其中,由于 POST 方式的请求参数存放在消息体中,所以通过 setCharacterEncoding()方法进行设置,而 GET 方式的请求参数存放在消息头中,通过 HttpServletRequestWrapper 类对 HttpServletRequest 类进行包装,并通过重写 getParameter()的方式设置 GET 方式提交参数的编码。

需要注意的是,由于要拦截用户访问资源的所有请求,因此需要将 CharacterFilter 映射信息中<filter-mapping>元素拦截的路径设置为"/*",如下所示:

```
<filter>
    <filter-name>CharacterFilter</filter-name>
     <filter-class>com.itheima.filter.CharacterFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>CharacterFilter</filter-name>
    <url-pattern>/*</url-pattern>
</filter-mapping>
```

4. 运行项目并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter14/form.jsp，此时，浏览器窗口中会显示一个用户登录的表单，在这个表单中输入用户名“张三”和密码“1234”，如图 14-9 所示。

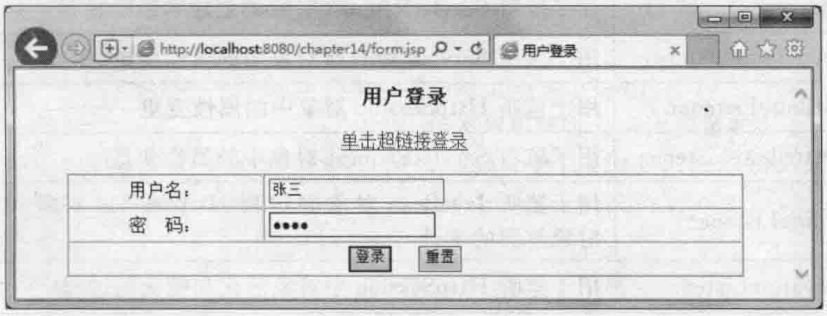


图 14-9　运行结果

单击图 14-9 中的【登录】按钮提交表单，此时，Eclipse 控制台显示的结果如图 14-10 所示。

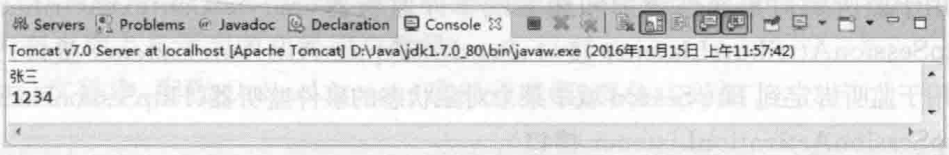


图 14-10　控制台窗口

从图 14-10 中可以看出，form.jsp 表单中输入的信息已经显示在了控制台窗口中，而且中文的用户名也没有出现乱码。需要注意的是，由于 form.jsp 表单的提交方式是 POST，因此可以说明使用 CharacterFilter 类可以解决 POST 提交方式的中文乱码问题。

接下来验证 GET 方式提交表单的乱码问题是否可以解决。单击图 14-9 中的“单击超链接登录”链接后（这种提交方式相当于 GET 方式提交信息），Eclipse 控制台窗口显示的结果同样如图 14-10 所示。因此可以说明，使用 CharacterFilter 类可以解决 GET 提交方式的中文乱码问题。

14.2　Listener

14.2.1　Servlet 事件监听器概述

Servlet 事件监听器是一个实现了特定接口的 Java 程序，这个程序专门用于监听 Web 应用中 ServletContext、HttpSession 和 ServletRequest 等域对象的创建和销毁过程、监听这些域对象属性的修改以及感知绑定到 HttpSession 域中的某个对象的状态。

Servlet 规范中定义了八种监听器，这八种监听器的类型及作用如表 14-4 所示。

在表 14-4 中，HttpSessionActivationListener 的描述中涉及活化和钝化的概念，HttpSession 对象从内存中转移至硬盘的过程称为钝化，HttpSession 对象从持久化状态变为运行状态的过程称为活化。

表 14-4 Servlet 事件监听器

类 型	描 述
ServletContextListener	用于监听 ServletContext 对象的创建与销毁过程
HttpSessionListener	用于监听 HttpSession 对象的创建和销毁过程
ServletRequestListener	用于监听 ServletRequest 对象的创建和销毁过程
ServletContextAttributeListener	用于监听 ServletContext 对象中的属性变更
HttpSessionAttributeListener	用于监听 HttpSession 对象中的属性变更
ServletRequestAttributeListener	用于监听 ServletRequest 对象中的属性变更
HttpSessionBindingListener	用于监听 JavaBean 对象绑定到 HttpSession 对象和从 HttpSession 对象解绑的事件
HttpSessionActivationListener	用于监听 HttpSession 中对象活化和钝化的过程

根据监听事件的不同，可以将表中的监听器分为如下三类。

(1) 用于监听域对象创建和销毁的事件监听器(ServletContextListener 接口、HttpSessionListener 接口、ServletRequestListener 接口)。

(2) 用于监听域对象属性增加和删除的事件监听器(ServletContextAttributeListener 接口、HttpSessionAttributeListener 接口、ServletRequestAttributeListener 接口)。

(3) 用于监听绑定到 HttpSession 域中某个对象状态的事件监听器(HttpSessionBindingListener 接口、HttpSessionActivationListener 接口)。

在 Servlet 规范中，这三类事件监听器都定义了相应的接口，在编写事件监听器程序时只需实现对应的接口即可。在使用监听程序时，Web 服务器会根据监听器所实现的接口，把它注册到被监听的对象上，当触发了某个对象的监听事件时，Web 容器将会调用 Servlet 监听器与之相关的方法对事件进行处理。

14.2.2 统计当前在线人数

在许多网站或者软件应用中都有统计当前在线人数这一功能。在通常情况下，当一个用户进入网站时就会创建一个 HttpSession 对象，而当用户离开网站时，HttpSession 对象就会被销毁。在 Servlet 的八种监听器中，HttpSessionListener 监听器就可以监听 Web 应用中 HttpSession 对象的创建和销毁过程。因此，可以使用实现了 HttpSessionListener 接口的监听器统计网站的当前在线人数。下面将分步骤讲解如何使用监听器统计网站当前在线人数。

1. 创建监听器

在 chapter14 项目的 com.itheima.listener 包中，创建一个实现了 HttpSessionListener 接口的监听器类 OnlineListener，在该类中编写用于实现统计网站在线人数的代码，如文件 14-12 所示。

文件 14-12 OnlineListener.java

```
1  package com.itheima.listener;
2  import javax.servlet.*;
3  import javax.servlet.http.*;
```

```
4  public class OnlineListener implements HttpSessionListener {
5      private int count=0;          //用于统计在线人数
6      public void sessionCreated(HttpSessionEvent hse) {
7          count++;                  //session 对象创建时 count 变量加 1
8          ServletContext context=hse.getSession().getServletContext();
9          context.setAttribute("count", new Integer(count));
10     }
11     public void sessionDestroyed(HttpSessionEvent hse) {
12         count--;                  //session 对象销毁时 count 变量减 1
13         ServletContext context=hse.getSession().getServletContext();
14         context.setAttribute("count", new Integer(count));
15     }
16 }
```

在文件 14-12 中，首先定义了一个用于统计在线人数的成员变量 count。当每次监听到 HttpSession 对象创建时，count 成员变量会加 1；当每次监听到 HttpSession 对象销毁时，count 成员变量会减 1。在每次修改 count 成员变量后，还应该将 count 成员变量的值保存到 ServletContext 对象中，以便其他 JSP 页面程序可以从 ServletContext 对象中取出 count 成员变量，从而在页面上显示当前在线用户数量。

在 web.xml 文件中，配置 OnlineListener 事件监听器，具体代码如下：

```
<listener>
    <listener-class>
        com.itheima.listener.OnlineListener
    </listener-class>
</listener>
```

需要注意的是，这个 HttpSession 对象在不活动时的最长存活时间仍然是 2min。

2. 创建首页和注销页面

(1) 实现首页。在项目的 WebContext 目录中，创建一个名称为 index.jsp 的页面，该页面用于保存在 ServletContext 对象中的 count 变量值，即相当于显示当前在线人数，如文件 14-13 所示。

文件 14-13 index.jsp

```
1  <%@page language="java" contentType="text/html; charset=utf-8"
2  pageEncoding="utf-8"%>
3  <html>
4  <head>
5  <title>首页</title>
6  </head>
7  <body>
8      <h3>
9          当前在线人数为：<%=application.getAttribute("count")%>
```

```
    </h3>
    <a href="<%=response.encodeUrl("logout.jsp")%>">退出登录</a>
</body>
</html>
```

(4) 实现注销页面。在项目的 WebContext 目录中，编写注销页面 logout.jsp，logout.jsp 页面用于使当前 Session 对象失效，即完成用户注销功能，如文件 14-14 所示。

文件 14-14　logout.jsp

```
<%@page language="java" contentType="text/html; charset=utf-8"
pageEncoding="utf-8"%>
<html>
<head>
<title>注销页面</title>
</head>
<body>
    <%session.invalidate();%>
    <h3>您已退出本系统</h3>
</body>
</html>
```

3. 运行项目并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter14/index.jsp 访问 index.jsp 页面，然后再分别开启两个不同的浏览器窗口(如火狐和谷歌浏览器)同时访问 index.jsp 页面(目的是防止同一个浏览器共享同一个 Session 会话)，此时，各个浏览器窗口中显示的结果如图 14-11 所示。

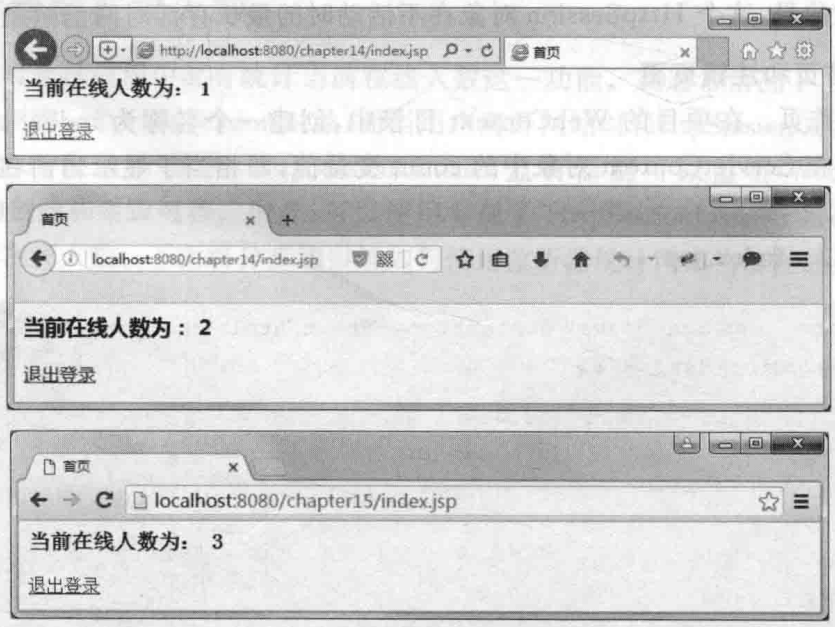


图 14-11　login.jsp 页面

单击图 14-11 中任意一个浏览器窗口中的“退出登录”超链接，都会访问 logout.jsp 页面，而 logout.jsp 页面中的 session.invalidate()语句会使当前 Session 失效，从而触发 HttpSession 对象的销毁事件并调用监听器 OnlineListener 中的 sessionDestroyed()方法，使统计用户数量的 count 变量减 1。然后，刷新其余的浏览器窗口，此时，各个浏览器窗口中显示的结果如图 14-12 所示。

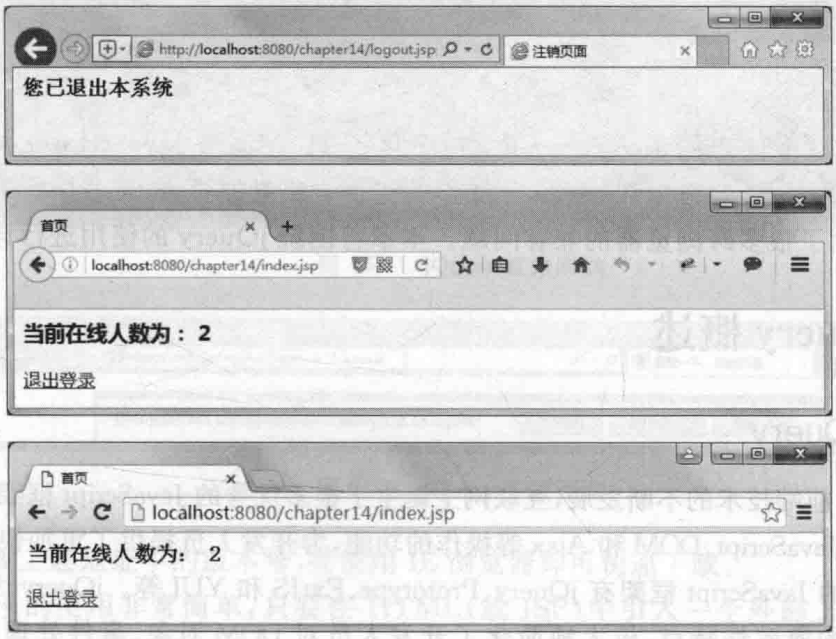


图 14-12　logout.jsp 页面

从图 14-12 中可以看出，IE 浏览器中的用户已被注销，而其余两个浏览器窗口中的在线人数显示为 2。

需要注意的是，如果直接关闭图 14-12 所示的某个浏览器窗口或保持浏览器窗口不刷新，与之对应的 HttpSession 对象会在 2 分钟之后才被销毁，即统计的在线人数的 count 变量会在 2 分钟后自动减 1。因此，可以说明网站统计的在线人数并不是真正意义上的并发访问人数，而是指某一个时间段内的访问人数。

第 15 章 jQuery框架

jQuery 是一个优秀的 JavaScript 库，它简化了 HTML 与 JavaScript 之间的操作，使 DOM 对象、事件处理、动画效果、Ajax 等操作的实现语法更加简洁，显著提高了程序的开发效率，同时消除了很多跨浏览器的兼容问题。本章将围绕 jQuery 的使用进行详细讲解。

15.1 jQuery 概述

15.1.1 jQuery

随着 Web 前端技术的不断发展，互联网上诞生了很多优秀的 JavaScript 框架，这些框架基本上都封装了 JavaScript、DOM 和 Ajax 等操作的功能，为开发人员提供了更加快捷、强大的开发方式。常见的 JavaScript 框架有 jQuery、Prototype、ExtJS 和 YUI 等。jQuery 因其简洁的语法和跨浏览器的兼容性特点，极大地简化了开发人员对 DOM 对象、事件处理、动画效果和 Ajax 的操作，目前已经从其他框架中脱颖而出，成为 Web 开发人员的最佳选择。

jQuery 框架的特点可以归纳为以下几点。

- jQuery 是一个轻量级的脚本，其代码非常小巧。
- 语法简洁易懂，学习速度快，文档丰富。
- 支持 CSS1～CSS3 定义的属性和选择器。
- 跨浏览器，支持的浏览器包括 IE、FireFox、Chrome 等。
- 实现了 JavaScript 脚本和 HTML 代码的分离，便于后期编辑和维护。
- 插件丰富，可以通过插件扩展更多功能。

15.1.2 jQuery 的下载与使用

jQuery 的官方网站（http://jquery.com/）提供了 jQuery 框架最新版本的下载，如图 15-1 所示。

从图 15-1 中可以看出，目前的 jQuery 最新版本是 v3.1.1。由于 jQuery 在 2.x 版本以后就不再支持 IE 6/7/8 浏览器，所以本书中讲解的是 jQuery 1.x 系列中的 1.11.3 版本，此版本还保持着对早期浏览器的支持。读者可以使用 IE 浏览器通过访问地址 https://code.jquery.com/jquery-1.11.3.min.js 下载此版本。在 IE 浏览器中输入网址并按 Enter 键后，浏览器的显示结果如图 15-2 所示。

从图 15-2 中可以看出，浏览器已经弹出了 jquery-1.11.3.min.js 的下载框，此时通过单击保存按钮就可以将文件保存到计算机的相应位置。如果需要下载其他版本的 jQuery，



图 15-1　jQuery 官方网站

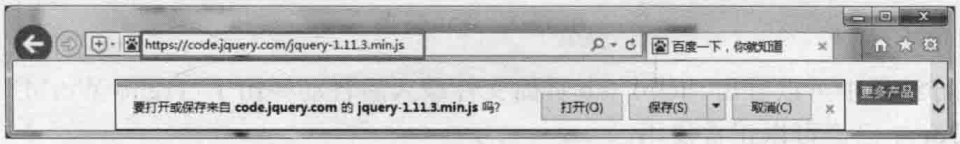


图 15-2　jQuery 下载页面

只需要更改上述地址中的版本号，并使用 IE 浏览器即可快速下载。

jQuery 的使用非常简单，只要在 HTML(或 JSP)中引入一个外部 JavaScript 文件即可，其引入代码如下所示：

```
<script src="jquery-1.11.3.min.js"></script>
```

上述代码表示引入当前目录下的“jquery-3. 1. 1. min. js”文件。将 jQuery 文件引入之后，就可以使用 jQuery 的各种功能了，下面通过简单的案例演示 jQuery 的使用。

1. 创建项目

在 Eclipse 中创建一个名称为 chapter15 的 Web 项目，在项目的 WebContent 目录下创建一个名称为 js 的文件夹，并将下载的 jquery-1. 11. 3. min. js 文件放置在文件夹中。

2. 创建 hello. jsp 文件

在 WebContent 目录下创建一个名称为 hello. jsp 的文件，在该文件的<head>标记中引入 jQuery 类库文件，并使用 jQuery 编写一个弹出对话框，其代码如下所示：

```
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        alert("Hello World !");
    })
</script>
```

在上述代码中，$(document). ready(function(){ … })表示页面加载完成后执行匿名

函数 function(),相当于 JavaScript 中的 windows. onload 函数。这里还可以简写成 $ (). ready(function(){...})或 $ (function(){...})这样的形式。

3. 运行程序并查看结果

发布项目并启动服务器,在浏览器的地址栏中输入 http://localhost:8080/chapter15/hello. jsp 访问 hello. jsp,浏览器的显示结果如图 15-3 所示。



图 15-3 显示结果

从运行结果中可以看出,hello. jsp 页面文件载入后自动弹出了“Hello World!”对话框,这说明 jQuery 已经可以正常使用。

在 jQuery 中,符号 $ 的使用最为频繁。$ 本质上是一个函数,该函数根据其参数的不同可以实现不同的功能,如作为选择器使用、作为功能函数的前缀、创建页面的 DOM 节点等。此外,还可以使用 jQuery 代替 $,例如 $ (document)可以写成 jQuery(document)。$ 实际上是 jQuery 的简写形式。

15.2 jQuery 选择器

在页面中为某个元素添加属性或事件时,必须先准确地找到该元素。jQuery 的选择器就是用于完成这一工作的。在 jQuery 中,事件的处理、遍历 DOM 和 Ajax 操作都依赖于选择器。本节将针对 jQuery 的选择器进行详细讲解。

15.2.1 基本选择器

基本选择器是 jQuery 中最常用的选择器,它的语法与 CSS 选择器非常相似,都是通过 id、类名和标签名的方式查找网页中的元素。通过基本选择器可以实现大多数页面元素的查找。

使用 jQuery 选择器的基本语法为 $ (选择器),其详细说明如表 15-1 所示。

表 15-1 基本选择器

选 择 器	功 能 描 述	示 例
*	匹配所有元素	$ ("*")选取所有的元素
#id	根据指定 id 匹配一个元素	$ ("#test") 选取 id 为 test 的元素
.class	根据指定类名匹配所有元素	$ (".test") 选取所有 class 为 test 的元素
element	根据指定元素名匹配所有元素	$ ("div") 选取所有的 元素
selector1,...,selectorN	将每个选择器匹配到的元素合并后一起返回	$ ("div,span,p.test")选取所有标签为 、和<pclass="test">的一组元素

表 15-1 列举了基本选择器的功能描述和示例。为了使读者更好地学习基本选择器的使用，下面通过案例演示 #id 和. class 选择器的使用。

在项目的 WebContent 目录下创建一个名称为 base. jsp 的文件，在文件中编写代码，通过 jQuery 实现单击按钮改变相应元素的颜色功能。其具体代码如文件 15-1 所示。

文件 15-1　base. jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>基本选择器</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function(){
            //获取 class 为 box 的元素，并将该元素的背景颜色设置为蓝色
            $(".box").css("background-color", "blue");
        });
    })
</script>
<style type="text/css">
    .box {width:100px;height:50px;background-color:#ccc;}
</style>
</head>
<body>
    <div class="box" />
    <input id="btn" type="button" value="点我" />
</body>
</html>
```

启动项目，在浏览器的地址栏中输入地址 http://localhost:8080/chapter15/base. jsp 访问 base. jsp，浏览器的显示结果如图 15-4 所示。

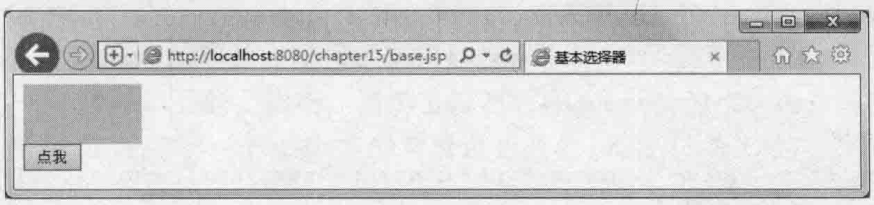


图 15-4　显示结果

在文件 15-1 中，CSS 代码将 box 的背景色设置为灰色，当单击按钮后，将通过 jQuery 把背景色改为蓝色，如图 15-5 所示。

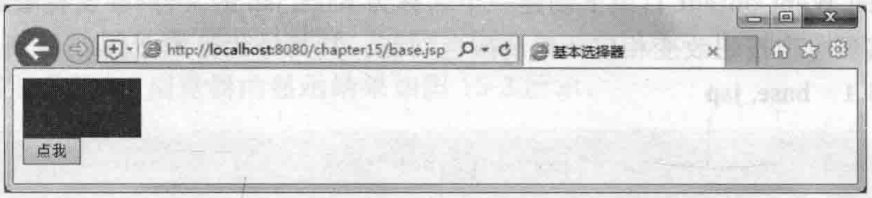


图 15-5 单击按钮后的显示结果

从上述案例中可以看出，当通过 class 获取指定元素时，CSS 和 jQuery 的语法同样都是.box。

15.2.2 层次选择器

层次选择器是通过 DOM 元素间的层次关系获得元素的。例如后代元素、子元素、相邻元素和兄弟元素等，其详细说明如表 15-2 所示。

表 15-2 层次选择器

选 择 器	功 能 描 述	示 例
ancestor descendant	选取祖先(ancestor)元素匹配所有后代(descendant)元素	$("div span")选取 中的所有后代元素
parent>child	根据父元素匹配所有的子元素	$("div > span")选取 中的所有子元素
prev+next	匹配所有紧跟在 prev 元素后的相邻元素	$(".test+div")选取 class 为 test 的下一个 元素
prev~siblings	匹配 prev 元素之后的所有兄弟元素	$("#test~div")选取 id 为 test 的元素后面的所有 兄弟元素

表 15-2 列举了层次选择器的功能描述和示例。为了使读者更好地学习层次选择器，下面通过案例演示层次选择器的使用。

在项目的 WebContent 目录下创建一个名称为 hierarchy.jsp 的文件，编辑代码后如文件 15-2 所示。

文件 15-2 hierarchy.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>层次选择器</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
```

```
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function() {
            //为<body>内所有<div>增加边框
            $("body div").css("border", "1px solid #000");
            //改变 id 为 A 的下一个<div>的背景颜色为蓝色
            $("#A +div").css("background-color", "blue");
            //改变 id 为 C 的子<div>的背景颜色为灰色
            $("#C >div").css("background-color", "grey");
        });
    })
</script>
<style type="text/css">
    div {margin: 2px auto;}
    div>div {font-weight: bold;}
</style>
</head>
<body>
    <div id="A">A</div>
    <div id="B">B</div>
    <div id="C">C
        <div id="D">D</div>
    </div>
    <input id="btn" type="button" value="点我" />
</body>
</html>
```

启动服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter15/hierarchy.jsp 访问 hierarchy.jsp，其显示结果如图 15-6 所示。

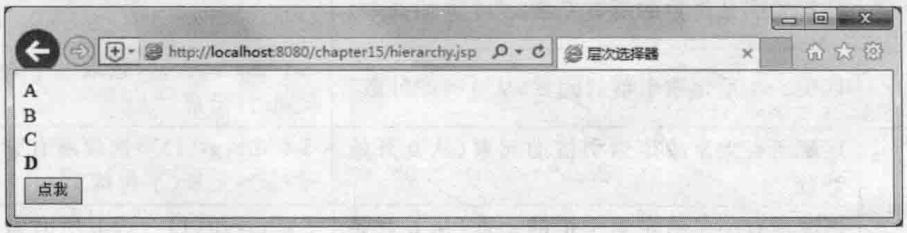


图 15-6　运行结果

在文件 15-2 中，CSS 代码通过子元素选择器将＜div＞的子元素＜div＞设置了文本加粗，所以图 15-6 中只有 D 受到了影响。当单击按钮后，jQuery 会为＜body＞内所有＜div＞增加边框，改变 id 为 A 的下一个＜div＞的背景色为蓝色，改变 id 为 C 的子＜div＞的背景色为灰色，如图 15-7 所示。

从上述案例中可以看出，jQuery 选择器和 CSS 选择器的用法是相同的。需要注意的是，IE 6 浏览器不支持 CSS 的子元素选择器，但是 jQuery 的选择器支持所有的浏览器。

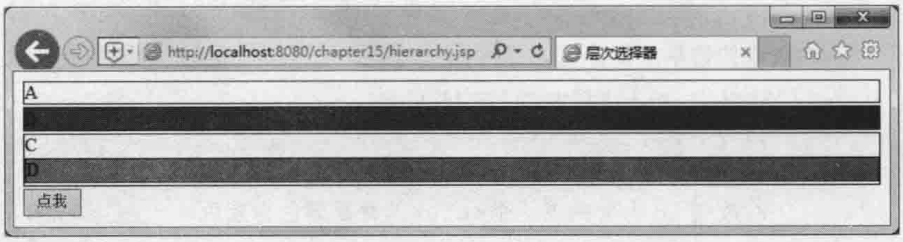


图 15-7 运行结果

15.2.3 过滤选择器

过滤选择器通过特定的过滤规则筛选出所需的 DOM 元素。过滤规则的语法与 CSS 伪类选择器的语法类似，都以英文冒号(:)开头。按照不同的过滤规则，过滤选择器可以分为基本过滤、内容过滤、可见性过滤、属性过滤、子元素过滤和表单对象属性过滤。下面针对这些过滤选择器进行详细讲解。

1. 基本过滤选择器

基本过滤选择器是过滤选择器中使用最为广泛的一种，其详细说明如表 15-3 所示。

表 15-3 基本过滤选择器

选 择 器	功 能 描 述	示 例
:first	匹配找到的第一个元素	$("div:first")选取所有 元素中第一个 元素
:last	匹配找到的最后一个元素	$("div:last")选取所有 元素中最后一个 元素
:not(selector)	匹配所有不匹配给定选择器的元素	$("div:not(.test)")选取所有 class 不是 test 的 元素
:even	匹配索引是偶数的所有元素(从 0 开始计数)	$("div:even")选取所有索引是偶数的 元素
:odd	匹配索引是奇数的所有元素(从 0 开始计数)	$("div:odd")选取所有索引是奇数的 元素
:eq(index)	匹配一个给定索引值的元素(从 0 开始计数)	$("div:eq(1)")选取一个索引等于 1 的 元素
:gt(index)	匹配所有大于给定索引值的元素(从 0 开始计数)	$("div:gt(1)")选取所有索引大于 1 的 元素(不包括 1)
:lt(index)	匹配所有小于给定索引值的元素(从 0 开始计数)	$("div:lt(1)")选取所有索引小于 1 的 元素(不包括 1)
:header	匹配所有的标题元素(如 h1,h2,h3 等)	$(":header")选取所有的标题元素<h1>、<h2>、<h3>等
:animated	匹配所有当前正在执行动画的元素	$("div:animated")选取所有正在执行动画的 元素

表 15-3 列举了基本过滤选择器的功能描述和示例。为了使读者更好地学习基本过滤选择器，下面通过案例演示基本过滤选择器:odd 的使用。

在项目的 WebContent 目录下创建一个名称为 basic_filter.jsp 的文件，编辑代码后如

文件 15-3 所示。

文件 15-3　basic_filter.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>基本过滤选择器</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function() {
            //改变<table>内所有<tr>索引为奇数的背景色为灰色
            $("table tr:odd").css("background-color","grey");
        });
    })
</script>
<style type="text/css">
    table {border-collapse:collapse;width:80%;}
</style>
</head>
<body>
    <table border="1">
        <tr><th>A</th><th>B</th><th>C</th></tr>
        <tr><td>1</td><td>1</td><td>1</td></tr>
        <tr><td>2</td><td>2</td><td>2</td></tr>
        <tr><td>3</td><td>3</td><td>3</td></tr>
        <tr><td>4</td><td>4</td><td>4</td></tr>
        <tr><td>5</td><td>5</td><td>5</td></tr>
     </table>
    <input id="btn" type="button" value="点我" />
</body>
</html>
```

启动服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter15/basic_filter.jsp 访问 basic_filter.jsp，其显示结果如图 15-8 所示。

当单击按钮后，表格中的奇数行改变了背景色，如图 15-9 所示。

需要注意的是，过滤选择器从 0 开始计数，所以表格的标题行没有被选取，如果选取的是偶数行，则标题行会被选取。

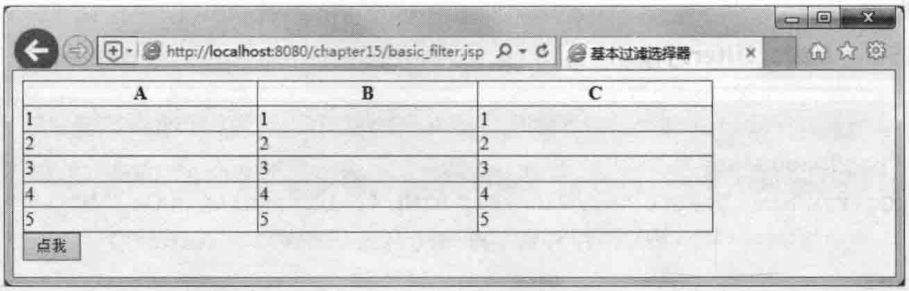


图 15-8 运行结果

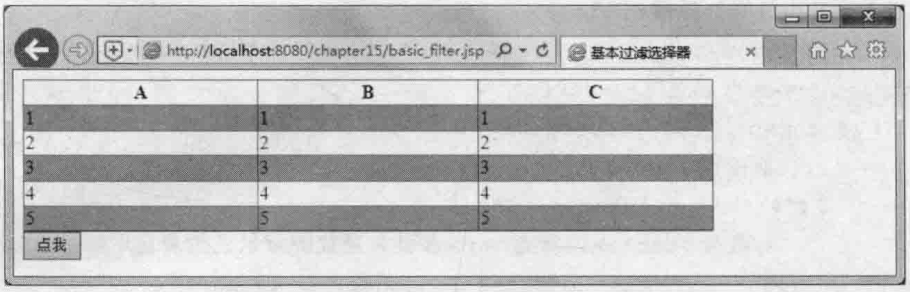


图 15-9 单击按钮后的运行结果

2. 内容过滤选择器

内容过滤选择器的过滤规则主要体现在它所包含的子元素或文本内容上，其详细说明如表 15-4 所示。

表 15-4 内容过滤选择器

选 择 器	功 能 描 述	示 例
:contains(text)	匹配包含给定文本的元素	$("div:contains(it)") 选取含有文本 it 的 元素
:empty	匹配所有不包含子元素或者文本的空元素	$("div:empty")选取不包含子元素或者文本元素的 元素
:has(selector)	匹配含有选择器所匹配的元素的元素	$("div:has(p)")选取含有<p>元素的 元素
:parent	匹配含有子元素或者文本的元素	$("div:parent")选取拥有子元素(包括文本元素)的 元素

表 15-4 列举了内容过滤选择器的功能描述和示例。为了使读者更好地学习内容过滤选择器，下面通过案例演示内容过滤选择器的使用。

在项目的 WebContent 目录下创建一个名称为 content_filter.jsp 的文件，编辑代码后如文件 15-4 所示。

文件 15-4 content_filter.jsp

```
1  <%@page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4                        "http://www.w3.org/TR/html4/loose.dtd">
```

```
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>内容过滤选择器</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function() {
            //加粗<div>元素中包含 title 的文本的内容
            $("div:contains('title')").css("font-weight", "bold");
            //将<div>元素中不包含子元素或者文本为空的元素的 html 内容设置为 is empty
            $("div:empty").html('is empty');
            //为包含 span 元素的 div 添加边框
            $('div:has(span)').css('border', '1px solid #000');
            //将拥有子元素的<ul>元素的背景色设置为红色
            $("ul:parent").css("background-color", "red");
        });
    })
</script>
</head>
<body>
    <div>title</div>
    <div />
    <div>
        <span>span</span>
    </div>
    <div>
        <ul>
            <li>列表</li>
        </ul>
    </div>
    <input id="btn" type="button" value="点我" />
</body>
</html>
```

启动服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter15/content_filter.jsp 访问 content_filter.jsp，其显示结果如图 15-10 所示。

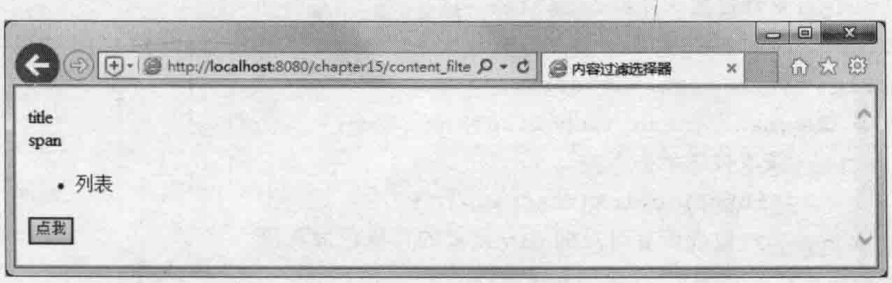


图 15-10　显示结果

单击按钮后的显示结果如图 15-11 所示。

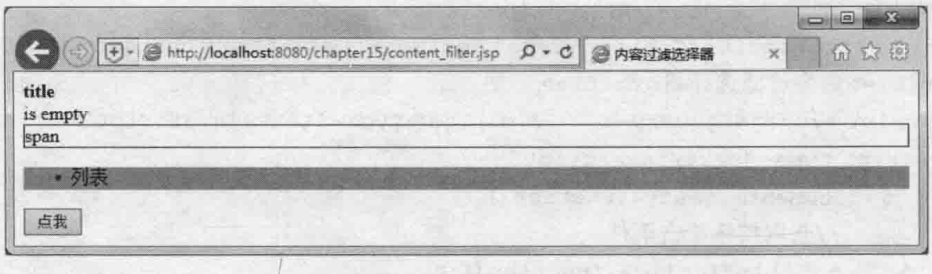


图 15-11　单击按钮后的显示结果

从图 15-11 中可以看出，当单击按钮后，页面中的元素已根据 jQuery 代码的需求改变了样式。

3. 可见性过滤选择器

可见性过滤选择器根据元素的可见与不可见状态选取元素，其详细说明如表 15-5 所示。

表 15-5　可见性过滤选择器

选　择　器	功 能 描 述	示　　例
:hidden	匹配所有不可见的元素	$(":hidden")选取所有不可见的元素 $("input:hidden")选取所有不可见的 input 元素
:visiable	匹配所有可见的元素	$("div:visiable")选取所有可见的 元素

表 15-5 列举了可见性过滤选择器的功能描述和示例。为了使读者更好地学习可见性过滤选择器，下面通过案例演示可见性过滤选择器的使用。

在项目的 WebContent 目录下创建一个名称为 visibility_filter.jsp 的文件，编辑代码后如文件 15-5 所示。

文件 15-5　visibility_filter.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>可见性过滤选择器</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function() {
            //改变所有可见的 div 元素的背景色为灰色
            $("div:visible").css("background-color","grey");
```

```
            //选取所有不可见的 div 元素,利用 jQuery 中的 show()方法将它们显示出
            //来,然后为其添加边框
            $("div:hidden").show().css('border', '1px solid #000');
        });
    })
</script>
</head>
<body>
    <div>可见的 div</div><br />
    <div style="display: none">隐藏的 div</div><br />
    <input id="btn" type="button" value="点我" />
</body>
</html>
```

启动服务器,在浏览器的地址栏中输入地址 http://localhost:8080/chapter15/visibility_filter.jsp 访问 visibility_filter.jsp,其显示结果如图 15-12 所示。

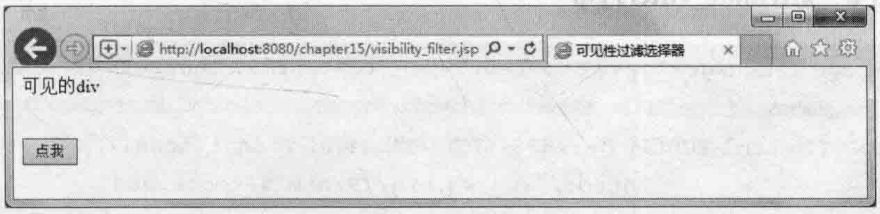


图 15-12　显示结果

单击按钮后的显示结果如图 15-13 所示。

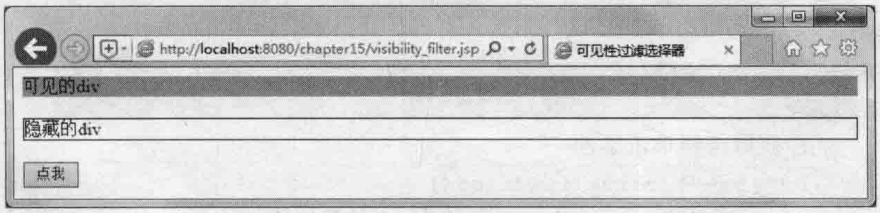


图 15-13　单击按钮后的显示结果

4. 属性过滤选择器

属性过滤选择器根据元素的属性选取元素,语法为[过滤规则]。其详细说明如表 15-6 所示。

表 15-6　属性过滤选择器

选　择　器	功 能 描 述	示　　例
[attribute]	匹配拥有此属性的元素	$("div[id]")选取拥有属性为 id 的元素
[attribute=value]	匹配属性的值为 value 的元素	$("div[it=cast]")选取属性为 it,属性值为 cast 的 元素
[attribute!=value]	匹配属性值不是 value 的元素	$("div[it! =cast]")选取属性为 it,且值不为 cast 的 元素(没有 it 属性也会被选中)

续表

选 择 器	功 能 描 述	示 例
[attribute^=value]	匹配属性值以 value 开始的元素	$("div[it^=cast]")选取属性为 it,属性值以 cast 开始的 元素
[attribute$=value]	匹配属性值以 value 结束的元素	$("div[it$=cast]")选取属性为 it,属性值以 cast 结束的 元素
[attribute*=value]	匹配属性值含有 value 的元素	$("div[it*=cast]")选取属性为 it,属性值含有 cast 的 元素
[selector1][selector2]…[selectorN]	复合属性选择器,需要在同时满足多个条件时使用	$("div[id][it^=cast]")选取拥有 id 和 it 属性,并且 it 属性值为 cast 的 元素

表 15-6 列举了属性过滤选择器的功能描述和示例。为了使读者更好地学习属性过滤选择器,下面通过案例演示属性过滤选择器的使用。

在项目的 WebContent 目录下创建一个名称为 attribute_filter.jsp 的文件,编辑代码后如文件 15-6 所示。

文件 15-6 attribute_filter.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>属性过滤选择器</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function() {
            //为所有<div>中元素属性为 id 的元素添加边框
            $("div[id]").css('border', '1px solid #000');
            //将<div>中属性 id 值等于 num1 的元素的背景色设置为灰色
            $("div[id=num1]").css("background-color","grey");
            //将<div>中属性 id 值以 img 开始的元素的背景色设置为蓝色
            $("div[id^=img]").css("background-color","blue");
            //将<div>中属性 id 值以 le 结束的元素字体加粗
            $("div[id$=le]").css("font-weight", "bold");
            //将<div>中属性 id 值含有 it 的元素字体大小设置为 200%
            $("div[id*=it]").css("font-size", "200%");
        });
    })
</script>
</head>
```

```
<body>
    <div id="num1">编号</div>
    <div id="title">标题 1</div>
    <div id="img1">图片 1</div>
    <div id="img2">图片 2</div>
    <div id="item">项目</div>
    <input id="btn" type="button" value="点我" />
</body>
</html>
```

启动服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter15/attribute_filter.jsp 访问 attribute _filter.jsp，其显示结果如图 15-14 所示。

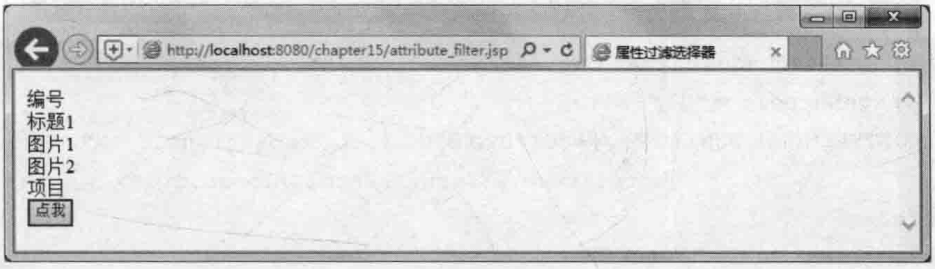


图 15-14　显示结果

单击按钮后的显示结果如图 15-15 所示。

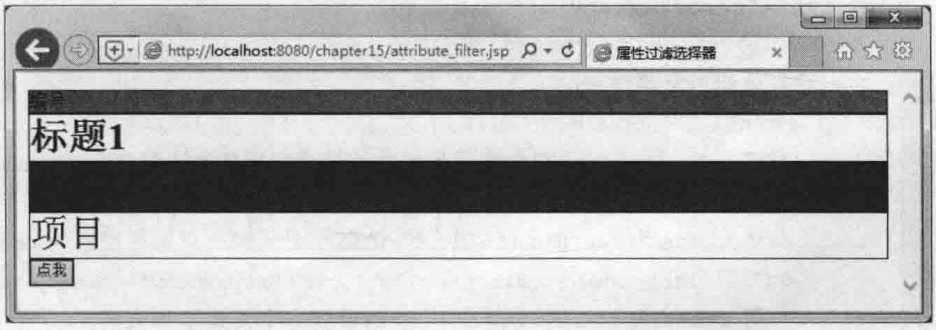


图 15-15　单击按钮后的显示结果

从图 15-15 中可以看出，当单击按钮后，页面中的元素已经改变了相应的样式。

5. 子元素过滤选择器

通过子元素过滤选择器可以获取父元素中指定的元素，其详细说明如表 15-7 所示。

表 15-7　子元素过滤选择器

选　择　器	功 能 描 述	示　　例
:nth-child(index|odd|even)	匹配父元素下特定位置的元素（从 1 开始计数）	$("ul li:nth-child(2)")选取 中的第二个 元素
:first-child	匹配父元素下的第一个子元素	$("ul li:first-child")选取 中的第一个 元素

续表

选 择 器	功 能 描 述	示 例
:last-child	匹配父元素下的最后一个子元素	$("ul li:last-child")选取 中的最后一个 元素
:only-child	如果某个元素是父元素中唯一的子元素，则会被匹配	$("ul li:only-child")选取 中是唯一子元素的 元素

表 15-7 列举了子元素过滤选择器的功能描述和示例。为了使读者更好地学习子元素过滤选择器，下面通过案例演示字元素过滤选择器的使用。

在项目的 WebContent 目录下创建一个名称为 child_filter.jsp 的文件，编辑代码后如文件 15-7 所示。

文件 15-7 child_filter.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>子元素过滤选择器</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function() {
            //将<ul>中<li>元素的索引为偶数的元素的字体大小设置为 200%
            $("ul li:nth-child(even)").css("font-size", "200%");
            //将 class 为 one 的<ul>中<li>的第一个元素的背景色设置为#ddd
            $("ul[class=one] li:first-child").css("background-color", "#ddd");
            //为 class 为 one 的<ul>中<li>的最后一个元素添加边框
            $("ul[class=one] li:last-child").css("border", "2px solid #000");
            //将<ul>中唯一子元素为<li>的元素字体加粗
            $("ul li:only-child").css("font-weight", "bold");
        });
    })
</script>
</head>
<body>
    <ul class="one">
        <li>Java 基础入门</li>
        <li>Java Web 程序开发入门</li>
        <li>Java Web 程序开发进阶</li>
        <li>SSH 框架整合实战教程</li>
```

```
    </ul>
    <ul>
        <li>MySQL 数据库入门</li>
    </ul>
    <input id="btn" type="button" value="点我" />
</body>
</html>
```

启动服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter15/child_filter.jsp 访问 child _filter.jsp，其显示结果如图 15-16 所示。

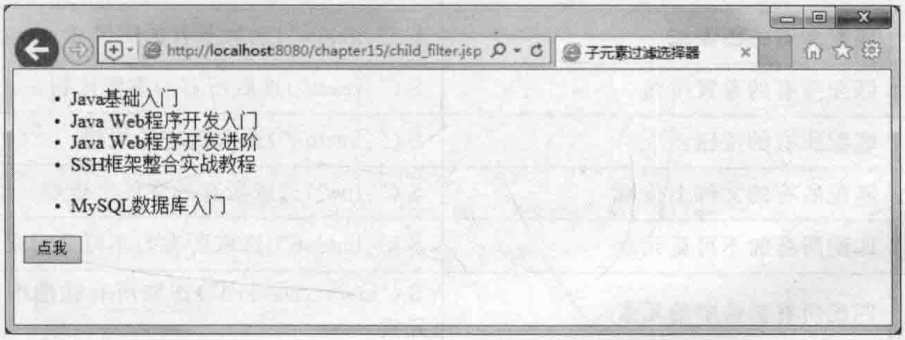


图 15-16　显示结果

单击按钮后的显示结果如图 15-17 所示。

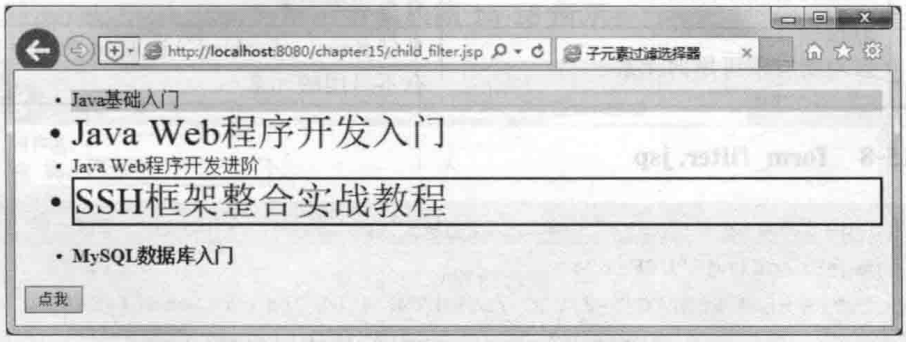


图 15-17　单击按钮后的显示结果

从图 15-17 中可以看出，当单击按钮后，页面中的元素已经按要求改变了样式。

15.2.4　表单选择器

表单的作用是提交数据。在前端程序开发中，对表单的处理占据了重要的地位。为了灵活地操作表单，jQuery 专门加入了表单选择器，方便用户获取表单元素。关于表单选择器的详细说明如表 15-8 所示。

表 15-8 列举了表单选择器的功能描述和示例。为了使读者更好地学习表单选择器，下面通过案例演示表单选择器的使用。

在项目的 WebContent 目录下创建一个名称为 form_filter.jsp 的文件，编辑代码后如文件 15-8 所示。

表 15-8 表单选择器

选 择 器	功 能 描 述	示 例
:input	匹配所有的<input>、<textarea>、<select>和<button>元素	$(":input")选取所有<input>、<textarea>、<select>、<button>元素
:text	匹配所有的单行文本框	$(":text")选取所有的单行文本框
:password	匹配所有的密码框	$(":password")选取所有的密码框
:radio	匹配所有的单选按钮	$(":radio")选取所有的单选按钮
:checkbox	匹配所有的多选框	$(":checkbox")选取所有的多选框
:submit	匹配所有的提交按钮	$(":submit")选取所有的提交按钮
:image	匹配所有的图像域	$(":image")选取所有的图像域<img>
:reset	匹配所有的重置按钮	$(":reset")选取所有的重置按钮
:button	匹配所有的按钮	$(":button")选取所有的按钮
:file	匹配所有的文件上传域	$(":file")选取所有的文件上传域
:hidden	匹配所有的不可见元素	$(":hidden")选取所有的不可见元素
:checked	匹配所有被选中的元素	$("input :checked")选取所有被选中的<input>元素
:selected	匹配所有被选中的选项元素	$("select :selected")选取所有被选中的<option>元素
:enabled	匹配所有可用的元素	$("#test :enabled")选取 id 为 test 的表单内所有可用的元素
:disabled	匹配所有不可用的元素	$("#test :disabled")选取 id 为 test 的表单内所有不可用的元素

文件 15-8 form_filter.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>表单选择器</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function() {
            //获取单行文本框的值,并赋值给变量 user
            var user=$(":text").val();
            //获取密码输入框的值,并赋值给变量 pwd
```

```
            var pwd=$(":password").val();
            //通过弹出框显示输入的值
            alert("您输入的用户名是："+user+"密码是："+pwd);
        });
    })
</script>
<style type="text/css">
    input[type=text], input[type=password] {
        width: 150px;
    }
</style>
</head>
<body>
    <form>
        用户名：<input id="username" name="username" type="text" /><br />
        密   码：<input id="pwd" name="pwd" type="password" /><br />
        <input id="btn" type="button" value="点我" />
    </form>
</body>
</html>
```

启动服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter15/form_filter.jsp 访问 form_filter.jsp，其显示结果如图 15-18 所示。

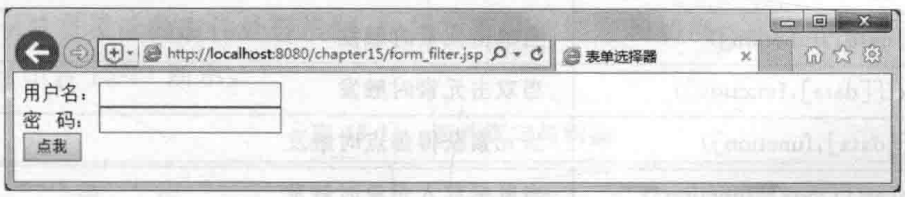


图 15-18　显示结果

在图 15-18 所示的用户名输入框中输入“admin”，在密码输入框中输入“1234”，单击按钮后的显示结果如图 15-19 所示。

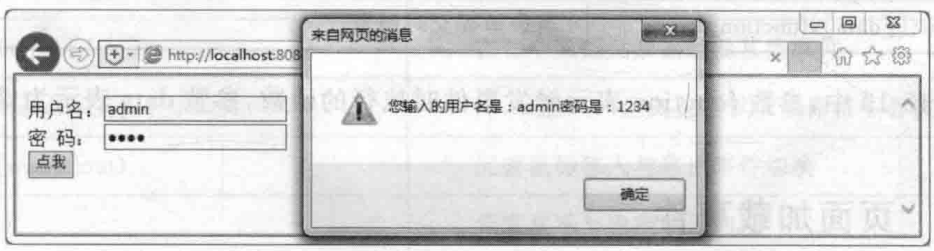


图 15-19　单击按钮后的显示结果

从图 15-19 中可以看出，当单击按钮后，两个输入框中的值已经显示在弹出窗口中。

15.3 jQuery 的事件处理

15.3.1 常用事件

在 JavaScript 中，当为一个元素添加单击事件时，需要在元素的 HTML 标记中添加 onclick 事件属性，然后调用 JavaScript 相应的函数进行处理，而 jQuery 实现了 JavaScript 与 HTML 代码的分离，在添加事件时不需要修改 HTML 标记，并且它还解决了不同浏览器的兼容问题。

在 jQuery 中，为元素添加事件的语法非常简单，直接调用 jQuery 对象中的事件方法即可。例如为按钮元素添加单击事件，具体代码如下：

```
$("button").click(function(){
    //处理语句
});
```

在上述代码中，click()是单击的事件方法，function()是该事件的处理函数。

在 jQuery 中，类似 click()的事件方法还有很多，常用的事件方法如表 15-9 所示。

表 15-9 jQuery 常用事件

方 法	说 明
blur([[data],function])	当元素失去焦点时触发
change([[data],function])	当元素的值发生改变时触发
click([[data],function])	当单击元素时触发
dblclick([[data],function])	当双击元素时触发
focus([[data],function])	当元素获得焦点时触发
mouseover([[data],function])	当鼠标移入对象时触发
mouseout([[data],function])	在鼠标从元素上离开时触发
scroll([[data],function])	当滚动条发生变化时触发
select([[data],function])	当文本框(包括 input 和 textarea)中的文本被选中时触发
submit([[data],function])	当表单提交时触发

在表 15-19 中，参数 function 表示触发事件时执行的函数，参数 data 表示为函数传入的参数。

15.3.2 页面加载事件

页面加载事件方法在前面的学习中已经使用过，它有以下三种语法形式：

```
//第一种形式
$(document).ready(function(){ })
```

```
//第二种形式
$().ready(function(){ })
//第三种形式
$(function(){ })
```

在上述语法中，第一种是完整写法，即调用 document 元素的 ready()事件方法。第二种语法省略了 document，第三种语法省略了 ready()。这三种语法的功能完全相同，读者可自行选择使用哪种形式。

在传统的 JavaScript 中，若要实现页面加载事件，需要在＜body＞中添加 onload 事件属性，或用 window. onload 方式注册事件，但是都必须等待网页中所有内容加载完成后才能执行。与之相比，jQuery 提供的 ready()方法更加完善。通过 ready()方法可以在页面加载后立即执行任务，并允许注册多个事件处理程序。

ready()方法与 window. onload 方法的对比如表 15-10 所示。

表 15-10　页面加载事件

方　法	window. onload	$(document). ready()
执行时机	必须等待网页中的所有内容加载完成后(包括图片)才能执行	网页中的所有 DOM 结构绘制完成后就执行(可能关联内容并未加载完成)
编写个数	不能同时编写多个	能够同时编写多个
简化写法	无	$()

15.3.3　事件绑定与切换

jQuery 具有灵活的事件处理机制，包括对事件的绑定和切换。关于事件绑定与切换的方法和说明如表 15-11 所示。

表 15-11　事件绑定与切换

语　法	说　明
on(events,[selector],[data],function)	在匹配元素上绑定一个或多个事件处理函数
off(events,[selector],[function])	在匹配元素上移除一个或多个事件处理函数
one(type,[data],function)	为每个匹配元素的事件绑定一次性的处理函数
trigger(type,[data])	在每个匹配元素上触发某类事件
triggerHandler(type,[data])	同 trigger()，但浏览器默认动作将不会被触发
hover([over,]out)	元素鼠标移入与移出事件切换
toggle()	元素显示与隐藏事件切换

表 15-11 列举了事件绑定与切换的方法，在这些方法的参数中，events 表示事件名(多个用空格分隔)，data 表示将要传递给事件处理函数的数据，selector 表示选择器，function 表示事件处理函数，type 表示添加到元素的事件(多个用空格分隔)，over 和 out 分别表示

鼠标移入和移出时的事件处理函数。

下面分别演示事件绑定与切换方法的使用案例。

(1) 事件的绑定与取消绑定

```
//on()方法绑定事件
$("div").on("click",function(){
    alert("绑定事件");
});
//off()方法取消绑定
$("div").off("click");
```

(2) 绑定单次事件

```
$("div").one("click",function(){
    alert("绑定单次事件");
});
```

(3) 多个事件绑定同一个函数

```
$("div").on("mouseover mouseout",function(){
    alert("鼠标移入或移出");
});
```

(4) 多个事件绑定不同的函数

```
$("div").on({
    mouseover:function(){
        alert("鼠标移入");
    },
    mouseout:function(){
        alert("鼠标移出");
    }
});
```

(5) 绑定自定义事件

```
//绑定自定义事件
$("div").on("CustomEvent",function(){
    alert("已触发自定义事件");
});
//触发自定义事件
$("div").click(function(){
    $("div").trigger("CustomEvent");
});
```

(6) 传递数据到事件处理函数

```
function myFunc(event){
    alert("收到消息: "+event.data.msg);
}
$("div").on("click",{msg:"测试数据"},myFunc)
```

(7) 为以后创建的元素委派事件

```
//为<body>的子元素<div>委派事件
$("body").on("click","div",function(){
    alert("收到");
});
//创建<div>元素
$("body").append("<div>测试</div>");
```

(8) 鼠标移入和移出事件切换

```
$("div").hover(function(){
    alert("切换-鼠标移入")
},function(){
    alert("切换-鼠标移出");
});
```

(9) 隐藏与显示事件切换

```
//第一次调用时隐藏
$("div").toggle();
//第二次调用时显示
$("div").toggle();
```

从以上事件绑定与切换的例子中可以看出,jQuery 事件处理方法的功能非常丰富,通过灵活地运用,可以实现很多复杂的页面交互效果。

注意: on()方法与 off()方法是 jQuery 从 1.7 版本开始新增的方法。jQuery 官方推荐使用 on()方法进行事件绑定,在新版本中已经取代了 bind()、delegate()和 live()方法。

15.4　jQuery 的 Ajax 操作

15.4.1　Ajax 概述

Ajax 是 Asynchronous Javascript And XML(异步 JavaScript 和 XML)的缩写,是指一种创建交互式网页应用的网页开发技术,这种技术并不是一门新的技术,而是 javaScript、XML、CSS、DOM 等多种已有技术的组合,它可以实现客户端的异步请求操作,从而使页面可以在不刷新的情况下与服务器进行通信。

在传统的 Web 应用模式中，页面中用户的每一次操作都将触发一次返回 Web 服务器的 HTTP 请求，然后在服务器对请求进行相应的处理后，再返回一个 HTML 页面给客户端。而在 Ajax 应用中，页面中的用户操作将通过 Ajax 引擎与服务器端进行通信，服务器会将返回结果提交给客户端页面的 Ajax 引擎，再由 Ajax 引擎将这些数据插入到页面的指定位置。

传统 Web 应用模式与使用 Ajax 的 Web 应用模式的工作方式如图 15-20 所示。

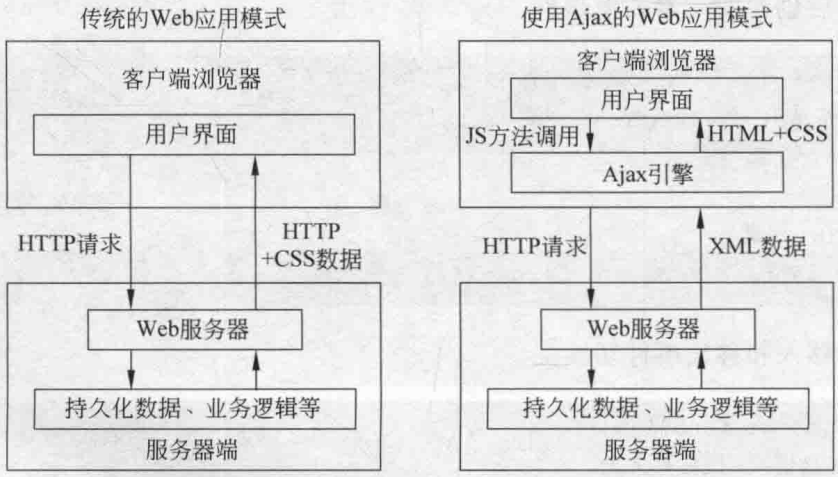


图 15-20　传统应用与 Ajax 应用的工作方式

从图 15-20 中可以看出，在传统的 Web 应用模式中，对于用户的每一个行为，都将生成一次 HTTP 请求，而在使用 Ajax 的 Web 应用中，用户的行为将变成对 Ajax 引擎的一次 JavaScript 调用，通过 JavaScript 实现在不刷新整个页面的情况下，对部分数据进行更新，从而给用户带来更好的体验。

与传统的 Web 应用技术相比，Ajax 技术具有如下优点。

- 减轻了服务器的负担。由于页面中是按需求获取数据的，这可以最大程度地减少冗余请求和响应对服务器造成的负担。
- 无刷新更新页面。Ajax 使用 XMLHttpRequest 对象发送请求并得到服务器响应，在不需要重新载入整个页面的情况下，就可以通过 DOM 及时将更新内容显示在页面中。
- 可以调用 XML 等外部数据，促进了页面显示和数据的分离。
- 使用简单方便。不需要下载插件或者小程序，即可在 Web 应用中直接使用。

15.4.2　jQuery 的 Ajax 操作

对于开发者而言，编写常规的 Ajax 代码并不容易，因为不同的浏览器对 Ajax 的实现并不相同，这也就意味着开发者需要编写额外的代码对浏览器进行测试。jQuery 框架很好地解决了这个难题，使用 jQuery 只需要一些简单的代码就可以实现 Ajax 功能。下面将针对 jQuery 中的 Ajax 操作进行详细讲解。

jQuery 提供了多个与 Ajax 有关的方法。通过 jQuery Ajax 方法，可以使用 HTTP 的 Get 和 Post 方法从服务器上获取请求的 HTML、XML 或 JSON 数据，并将这些数据直接载

入网页的被选元素中显示出来。在 jQuery 中，常用的 Ajax 操作方法如表 15-12 所示。

表 15-12　jQuery 常用的 Ajax 方法

方　法	说　明
load(url,[data],[function])	载入远程 HTML 文件代码并插入到 DOM 元素中
$.ajax(url,[settings])	通用的 Ajax 方法可发送请求并载入数据
$.get(url,[data],[function],[type])	通过 GET 方式发送请求并载入数据
$.post(url,[data],[function],[type])	通过 POST 方式发送请求并载入数据
$.getJSON(url,[data],[function])	通过 GET 方式发送请求并载入 JSON 数据
$.getScript(url,[function])	通过 GET 方式发送请求并载入 JavaScript 数据

在表 15-12 中，参数 url 表示请求的 URL 地址；参数 data 表示请求时发送的数据；参数 function 表示载入成功时执行的函数；参数 type 表示返回数据的格式，例如 xml、html、script、json、text 等；参数 settings 是对 Ajax 的请求设置，其常用的设置参数如表 15-13 所示。

表 15-13　settings 参数的常用设置参数

设置参数	说　明
type	用于指定请求方式，可以设置为 GET 或 POST，默认值为 GET
data	用于指定发送到服务器的数据。如果数据不是字符串，则将自动转换为请求字符串格式
dataType	用于指定服务器返回数据的类型，其常见返回类型有 xml，html 和 json 等
async	设置发送请求的方式，默认为 true，是异步请求方式，当设置为 false 时，表示同步请求方式
error(jqXHR,textStatus,errorThrown)	用于设置请求失败时调用的函数。这里的 jqXHR 是一个 jqXHR 对象，该对象不仅包括 XMLHttpRequest 对象，还包含其他更多的详细属性和信息；textStatus 表示返回的状态；errorThrown 表示服务器抛出返回的错误信息
success(data,textStatus,jqXHR)	用于设置请求成功时调用的函数。其中参数 data 表示处理后的数据，textStatus 表示描述状态的字符串，jqXHR 表示 jqXHR 对象

在表 15-12 的 Ajax 操作方法中，$.ajax()是通用方法，通过该方法的 setting 参数，可以实现与 $.get()、$.post()、$.getJSON()和 $.getScript()方法同样的功能。在编写 $.ajax()方法时，url 是 $.ajax()方法的必须参数，它既可以在 settings 参数之前编写，也可以在 settings 参数中编写。

以 $.ajax()方法为例，其基本的语法格式如下：

```
$.ajax({
    Type:POST",          //请求方式(GET 或 POST),默认为 GET
    url:rl,              //请求地址
    dataType:json',      //返回格式为 json
```

```
    data:ata,                    //参数值
    success:allback              //请求成功时的处理
    error:callback               //请求错误时的处理
});
```

为了使读者更好地学习 jQuery 的 Ajax 操作，下面通过检测用户名是否已存在的案例演示 Ajax 的使用。

1. 创建 register.jsp 文件

在项目的 WebContent 目录下创建一个名称为 register.jsp 的文件，编辑代码后如文件 15-9 所示。

文件 15-9 register.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>检查用户名是否存在</title>
<script src="js/jquery-1.11.3.min.js" type="text/javascript"></script>
<script type="text/javascript">
    $(document).ready(function() {
        //获取按钮单击事件
        $("#btn").click(function() {
            //将 id 为 userMsg 元素的内容设置为红色
            $("#userMsg").css("color","red");
            //获取 id 为 username 元素的值，并赋值给变量 username
            var username=$("#username").val();
            //判断 username 的值是否为空，如果不为空，则使用 jQuery 的 Ajax 方法
            //检查用户名是否存在
            if(username !=""){
            $.ajax({
                Type:"POST", //请求方式(GET 或 POST)，默认为 GET
                //请求地址
                url:"${pageContext.request.contextPath}/UserServlet",
                data:{
                    username:username          //传入的数据
                },
                success:function(msg) {
                    //返回的消息在 id 为 userMsg 的元素中显示
                    $("#userMsg").html(msg);
                }
            });
```

```
            }else{              //如果 username 的值为空,提示用户填写用户名
                $("#userMsg").html("请填写用户名");
            }
        });
    })
</script>
</head>
<body>
    <form method="POST">
        用户名: <input id="username" name="username" type="text"/>
        <input id="btn" type="button" value="检查用户名是否存在"/><br/>
        <span id="userMsg"/><br/>
        密    码: <input id="pwd" name="pwd" type="password"/>
        <br/>
        <input id="submit" type="button" value="注册"/><br />
    </form>
</body>
</html>
```

从文件 15-9 中可以看出,当单击【检测用户名是否存在】按钮后,将会通过 jQuery 代码中的 Ajax 检测所输入的用户名。

2. 创建 UserServlet 类

在项目的 src 目录下创建一个名称为 com. itheima. servlet 的包,并在该包中创建 UserServlet,编辑后的代码如文件 15-10 所示。

文件 15-10　UserServlet. java

```
package com.itheima.servlet;
import java.io.IOException;
import java.io.PrintWriter;
import java.util.Arrays;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class UserServlet extends HttpServlet {
    private static final long serialVersionUID=1L;
    public void doGet(HttpServletRequest request, HttpServletResponse
    response) throws IOException, ServletException {
        response.setContentType("text/html;charset=utf-8");
        //定义一个用于存储用户的数组
        String[] userList={"Mr","zhangsan","admin","lilei"};
        //对数组进行排序
```

```
        Arrays.sort(userList);
        //获取 username
        String username=request.getParameter("username");
        //在数组中查找 username
        int result=Arrays.binarySearch(userList, username);
        //获取 out 对象
        PrintWriter out=response.getWriter();
        String msg="";
        if (result >-1) {
            msg="用户名已存在,请重新填写用户名";
            out.print(msg);
        } else {
            msg="恭喜您,该用户没有被注册";
            out.print(msg);
        }
    }
    protected void doPost(HttpServletRequest request,
    HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

3. 运行程序并查看结果

启动服务器,在浏览器的地址栏输入地址 http://localhost:8080/chapter15/register.jsp 访问 register.jsp,其显示结果如图 15-21 所示。



图 15-21 运行结果

在用户名输入框中输入 admin,当单击【检测用户名是否存在】按钮后,页面的显示效果如图 15-22 所示。

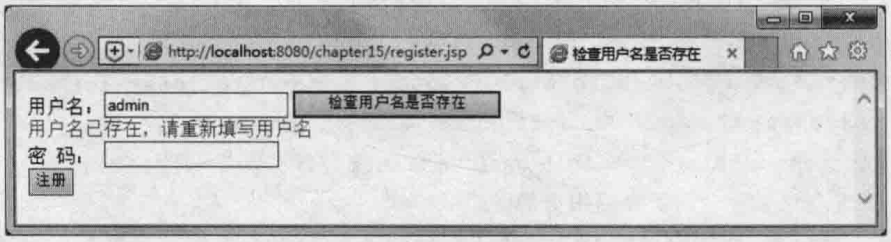


图 15-22 单击按钮后的运行结果

如果输入一个不存在的用户名(如 lisi),则检测后页面的显示效果如图 15-23 所示。

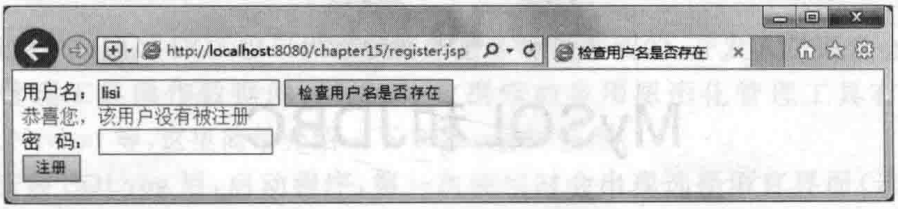


图 15-23　页面显示结果

第16章 MySQL和JDBC

在实际项目中，通常所有的数据信息都是存储在数据库中的，而针对项目中的某项操作，也会通过代码和SQL语句最终转换为对数据库的操作，因此作为一名开发人员，学会数据库的使用就变得十分重要。Java作为当前主流的编程语言，对数据库的访问也提供了支持，它拥有一套访问数据库的标准Java类库——JDBC。本章将围绕数据库和JDBC的相关知识进行详细讲解。

16.1 数据库基础知识

16.1.1 数据库概述

数据库(Database，DB)是按照数据结构组织、存储和管理数据的仓库，其本身可看作电子化的文件柜，用户可以对文件柜中的数据进行增加、删除、修改及查找等操作。需要注意的是，这里所说的数据(Data)不仅包括普通意义上的数字，还包括文字、图像及声音等。也就是说，凡是在计算机中用于描述事物的记录都可称为数据。数据库具有数据结构化、可实现数据共享、数据独立性高等特点。

在操作数据库时，使用的都是SQL(Structured Query Language，即结构化查询语言)。SQL是一种数据库查询语言和程序设计语言，主要用于管理数据库中的数据，如存取数据、查询数据和更新数据等。数据库中的操作都是通过SQL语句完成的，在Java应用程序中，可以通过调用SQL语句完成对数据的插入、修改、删除和查询等操作。

随着数据库技术的发展，数据库产品越来越多，当前主流的数据库有Oracle、DB2、MongoDB、SQL Server、MySQL等。本书将以MySQL为例，讲解一些数据库的基础知识。

16.1.2 MySQL的安装与配置

MySQL数据库支持多个平台，不同平台下的安装和配置过程也不相同。Windows平台的MySQL安装文件有两个版本，一种是以msi作后缀的二进制分发版，一种是以zip作为后缀的压缩文件。其中msi的安装文件提供了图形化的安装向导，按照向导提示进行操作即可完成安装，zip的压缩文件直接进行解压就可以完成MySQL的安装。本书使用的是二进制分发版MySQL 5.5，关于MySQL的安装和配置这里不做详细讲解，读者可以自行查找相关资料进行安装和使用。

16.1.3　MySQL 的图形化工具

在开发过程中，通过 DOS 窗口操作数据库是非常不方便的，开发人员通常会使用数据库的图形化管理工具操作数据库，MySQL 数据库的常用图形化管理工具有 SQLyog、Workbench、Navicat 等，这里简单介绍一下 SQLyog 的使用。

下载并安装 SQLyog 后，启动程序，第一次使用时会出现选择语言界面（选择简体中文），确认后会显示连接到我的 SQL 主机对话框，单击【新建】按钮，设置一个连接名称，如 chapter16，如图 16-1 所示。



图 16-1　连接到 MySQL 主机对话框

单击【确定】按钮，在界面密码处输入连接 MySQL 数据库的密码（这里为 root），如果之前已经创建好了数据库，可以在【数据/库】输入框中填入需要连接的数据库名（这里也可以不填写，不填写时会将 MySQL 中当前用户下的所有数据库显示出来），如图 16-2 所示。



图 16-2　连接 MySQL 配置信息对话框

完成后可以先单击【测试连接】按钮测试连接，成功后单击【连接】按钮会弹出是否保存更改的提示框，选择【是】即可。接下来会进入 SQLyog 的主窗口，如图 16-3 所示。

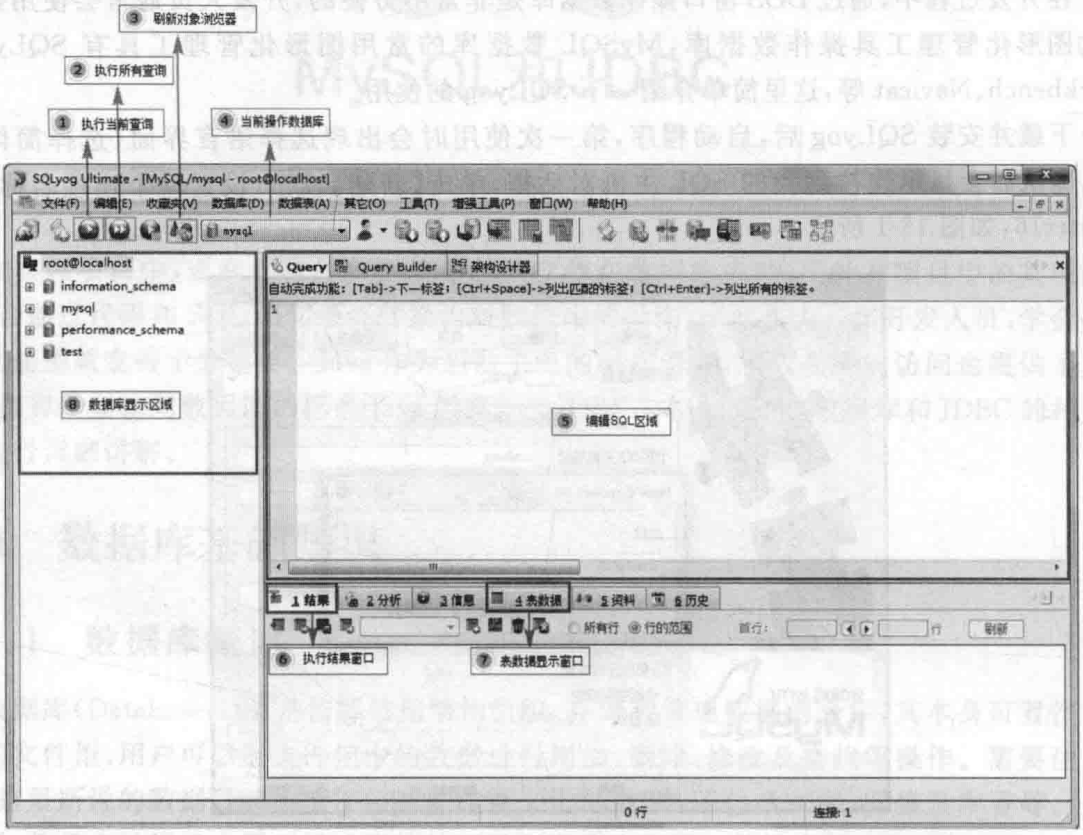


图 16-3 SQLyog 图像界面

图 16-3 展示了 SQLyog 工具的主界面，图中的主要功能介绍如下。

① 执行当前查询：在编辑 SQL 区域中编写了多条 SQL 语句时，如果只想执行其中的一条 SQL 语句，可以选中要执行的 SQL 语句，然后单击该图标按钮执行查询。

② 执行所有查询：在编辑 SQL 区域中编写了多条 SQL 语句时，单击该图标按钮，会执行所有的 SQL 语句。

③ 刷新对象浏览器：执行完 SQL 语句后（如创建数据库或表），数据库显示区域中可能不会立即显示创建结果，单击此图标按钮对数据库显示区域执行刷新操作后，即可显示。

④ 当前操作数据库：此处显示的是当前正在操作的数据库名称。

⑤ 编辑 SQL 区域：在此区域中，可以编写需要执行的 SQL 语句。

⑥ 执行结果窗口：执行完 SQL 语句后，会在此窗口中显示执行结果。

⑦ 表数据显示窗口：通过此窗口可以查看数据表中的数据。

⑧ 数据库显示区域：此区域中显示的是所有数据库信息，图 16-3 中显示的是 MySQL 中默认的四个数据库，新创建的数据库也会在此区域中显示。

16.2　MySQL 的基本使用

16.2.1　数据库的基本操作

1. 创建和查看数据库

创建数据库就是在数据库系统中划分一块存储数据的空间。在 MySQL 中，创建数据库的基本语法格式如下所示：

```
CREATE DATABASE 数据库名称;
```

在上述语法格式中，“CREATE DATABASE”是固定的 SQL 语句，专门用于创建数据库。“数据库名称”需要自定义，在定义时要保证其唯一性。

例如，要创建一个名称为 chapter16 的数据库，其 SQL 语句如下所示：

```
CREATE DATABASE chapter16;
```

将上述 SQL 语句写入编辑 SQL 区域，执行成功后，单击【刷新】按钮，此时数据库显示区域会新增一个名为 chapter16 的数据库。除了此种方法以外，还可以通过 SQL 语句查看所有数据库，查看数据库的 SQL 语句如下所示：

```
SHOW DATABASES;
```

执行查看 SQL 语句后，主窗口中的显示结果如图 16-4 所示。

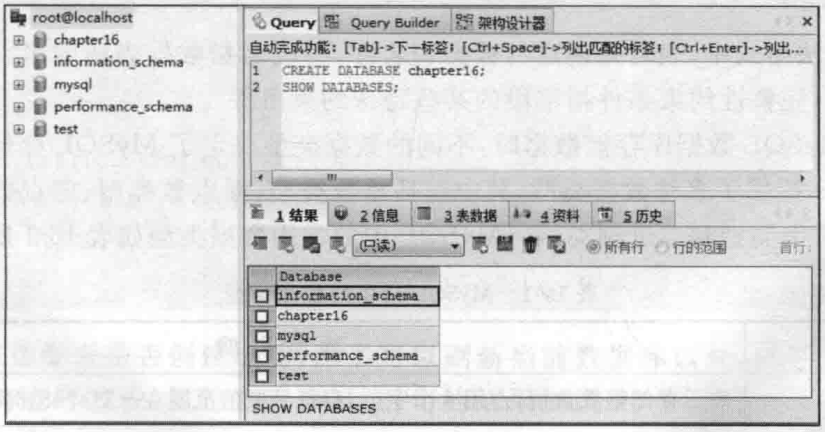


图 16-4　执行结果

从上述执行结果中可以看出，通过 SQL 语句已经将数据库系统中的 5 个数据库全部查出，并显示在了执行结果窗口中。

2. 删除数据库

删除数据库是将数据库系统中已经存在的数据库删除。成功删除数据库后，数据库中的所有数据都将被清除，原来分配的空间也将被回收。在 MySQL 中，删除数据库的基本语法格式如下所示：

```
DROP DATABASE 数据库名称;
```

在上述语法格式中，DROP DATABASE 是删除数据库的 SQL 语句，数据库名称是要删除的数据库名称。需要注意的是，如果要删除的数据库不存在，则删除会失败。

例如，要删除名称为 chapter16 的数据库，其 SQL 语句如下：

```
DROP DATABASE chapter16;
```

执行上述 SQL 语句后，刷新左侧数据库区域，会发现 chapter16 数据库已经成功删除。

16.2.2 表的基本操作

1. 创建数据表

数据库创建成功后，就需要创建数据表。创建数据表指在已经存在的数据库中建立新表。需要注意的是，在操作数据表之前，应该使用 USE 数据库名语句指定操作的是哪个数据库，否则会抛出 No database selected 的错误。

创建数据表的基本语法格式如下：

```
CREATE TABLE 表名
(
    字段名 1,数据类型[完整性约束条件],
    字段名 2,数据类型[完整性约束条件],
    ⋮
    字段名 n,数据类型[完整性约束条件]
)
```

在上述语法格式中，表名指创建的数据表名称，字段名指数据表的列名，数据类型指表中字段的类型，完整性约束条件指字段的某些特殊约束条件。

在使用 MySQL 数据库存储数据时，不同的数据类型决定了 MySQL 存储数据的方式。MySQL 数据库提供了多种数据类型，其中包括整数类型、浮点数类型、定点数类型、日期和时间类型、字符串类型和二进制类型。MySQL 中的常用数据类型如表 16-1 所示。

表 16-1 MySQL 常用的数据类型

数据类型	类型说明
INT	表示存储整数数值，占用 4 个字节，有符号取值范围在－2147483648～2147483648
FLOAT	表示存储小数，占用 4 个字节，有符号取值范围在－3.402823466E＋38～－1.175494351E－38
DATE	表示存储日期，日期格式为 YYYY-MM-DD
TIMESTAMP	表示存储日期和时间，日期格式为 YYYY-MM-DD HH:MM:SS
CHAR	表示固定长度的字符串
VARCHAR	表示可变长度的字符串

在上述创建表的语法中，设置表的完整性约束条件是为了防止数据表中插入错误的数

据。在 MySQL 中，定义了一些维护数据库完整性的规则，如表 16-2 所示。

表 16-2　表的约束

约束条件	说　明
PRIMARY KEY	主键约束，用于唯一标识对应的记录
FOREIGN KEY	外键约束
NOT NULL	非空约束
UNIQUE	唯一性约束
DEFAULT	默认值约束，用于设置字段的默认值
AUTO_INCREMENT	自动生成唯一的 ID，默认情况下，从 1 开始自增

在表 16-2 中，列举出的约束条件都是针对表的字段进行的限制，从而保证数据表中数据的正确性和唯一性。下面以在 chapter16 数据库中创建一个用于存储用户信息的 user 表为例，演示建表 SQL 语句的使用。

所需创建的 user 表结构如表 16-3 所示。

表 16-3　user 表

字段名称	数据类型	备注说明
id	INT(10)	用户编号
username	VARCHAR(20)	用户姓名
age	INT(10)	用户年龄

创建数据表的 SQL 语句如下所示：

```
CREATE DATABASE chapter16;
USE chapter16;
CREATE TABLE user
(
    id    INT(10)  PRIMARY KEY,
    username  VARCHAR(20),
    age  INT(10)
);
```

为了验证数据表是否创建成功，除了可以刷新当前数据库以外，还可以使用 SHOW TABLES 语句进行查看，执行查看 SQL 语句的结果如图 16-5 所示。

	Tables_in_chapter16
☐	user

图 16-5　执行结果

从上述执行结果中可以看出，chapter16 数据库中已经存在了数据表 user，说明数据表创建成功了。

2. 查看数据表

在 MySQL 中，使用 DESCRIBE 语句可以查看表的字段信息，其中包括字段名、字段类型等信息。DESCRIBE 语句的基本语法格式如下所示：

```
DESCRIBE 表名;
```

或

```
DESC 表名;
```

使用 DESCRIBE 语句查看 user 表，其 SQL 语句如下所示：

```
DESCRIBE user;
```

执行结果如图 16-6 所示。

	Field	Type		Null	Key	Default		Extra
□	id	int(10)	7B	NO	PRI	(NULL)	0K	
□	username	varchar(20)	11B	YES		(NULL)	0K	
□	age	int(10)	7B	YES		(NULL)	0K	

图 16-6 执行结果

上述执行结果显示出了 user 表的字段信息。下面针对执行结果中的不同字段进行详细讲解。

- NULL：表示该列是否可以存储 NULL 值。
- Key：表示该列是否已经编制索引。
- Default：表示该列是否有默认值。
- Extra：表示获取到与给定列相关的附加信息。

3. 修改数据表

在 MySQL 中，修改表名的基本语法格式如下所示：

```
ALTER TABLE 旧表名 RENAME [TO] 新表名;
```

在上述格式中，旧表名指修改前的表名，新表名指修改后的表名，关键字 TO 是可选的，其在 SQL 语句中是否出现不会影响语句的执行。下面将数据库 chapter16 中的 user 表名改为 t_user 表，其 SQL 语句如下所示：

```
ALTER TABLE user RENAME TO t_user;
```

为了检测表名是否修改正确，再次使用 SHOW TABLES 语句查看数据库中的所有表，执行结果如图 16-7 所示。

	Tables_in_chapter16
□	t_user

图 16-7 执行结果

从上述执行结果中可以看出，数据库中的 user 表已经成功修改为 t_user。

4. 删除数据表

在 MySQL 中，使用 DROP TABLE 语句可以删除没有被其他表关联的数据表，其基本的语法格式如下所示：

```
DROP TABLE 表名;
```

在上述格式中，表名指要删除的数据表。例如，如果要删除数据表 t_user，则其 SQL 语

句如下所示：

```
DROP TABLE t_user;
```

16.2.3　数据的基本操作

1. 添加数据

在 MySQL 中，使用 INSERT 语句向表中添加数据。向表中添加新记录时，可以在 INSERT 语句中列出表的所有字段名，其语法格式如下所示：

```
INSERT INTO 表名(字段名 1,字段名 2,...) VALUES(值 1,值 2,...);
```

在上述语法格式中，“字段名 1,字段名 2,...”表示数据表中的字段名称，此处必须列出表所有字段的名称；“值 1,值 2,...”表示每个字段的值，每个值的顺序和类型必须与对应的字段名相匹配。

例如，要向 goods 表中添加一条新记录，就需要使用上述 insert 语句。在添加记录之前，需要在 chapter16 数据库中创建一个 goods 表存储商品信息。创建 goods 表的 SQL 语句如下：

```
CREATE TABLE goods(
    id   INT(4),
    name    VARCHAR(20) NOT NULL,
    price    FLOAT
);
```

使用 INSERT 语句向 goods 表中插入一条数据，其 SQL 语句如下所示：

```
INSERT INTO goods(id,name,price) VALUES(1,'铅笔',0.5);
```

当上述 SQL 语句执行成功后，会在 goods 表中添加一条数据。为了验证数据是否添加成功，可以单击新建的 goods 表，并通过表数据显示窗口查看，goods 表中的数据如图 16-8 所示。

	id	name	price
□	1	铅笔	0.5

图 16-8　执行结果

需要注意的是，使用 INSERT 语句添加记录时，表名后的字段顺序可以与其在表中定义的顺序不一致，它们只需要与 VALUES 中值的顺序一致即可。

2. 更新数据

更新数据是指对表中存在的记录进行修改，这是数据库常见的操作，如某个商品改动了价格，就需要对其记录信息中的 price 字段值进行修改。MySQL 中使用 UPDATE 语句更新表中的记录，其基本的语法格式如下所示：

```
UPDATE 表名
SET 字段名 1=值 1[,字段名 2=值 2,...]
[WHERE 条件表达式]
```

在上述语法格式中，字段名 1 和字段名 2 用于指定要更新的字段名称，值 1 和值 2 用于表示字段更新的新数据。WHERE 条件表达式是可选的，用于指定更新数据需要满足的条件。

下面以更新 goods 表中 id 字段值为 1 的记录为例，演示 UPDATE 语句的使用。将 id 为 1 的记录中的 price 字段的值更改为 0.8，其 SQL 语句如下所示：

```
UPDATE goods set price=0.8 WHERE id=1;
```

上述 SQL 语句执行成功后，会将 id 为 1 的数据进行更新。此时 goods 表中 id 为 1 的记录信息如图 16-9 所示。

	id	name	price
□	1	铅笔	0.8

图 16-9 执行结果

从执行结果中可以看到，id 字段值为 1 的记录发生了更新，记录中 price 字段的值变为 0.8。

3. 删除数据

删除数据是指对表中存在的记录进行删除，这是数据库的常见操作，如某个商品卖完了，就需要在 goods 表中将其信息记录删除。MySQL 中使用 DELETE 语句删除表中的记录，其语法格式如下所示：

```
DELETE FROM 表名 [WHERE 条件表达式]
```

在上面的语法格式中，表名用于指定要执行删除操作的表，WHERE 子句为可选参数，用于指定删除的条件，满足条件的记录会被删除。例如，要删除 goods 表中 id 字段值为 1 的记录的 SQL 语句如下所示：

```
DELETE FROM goods WHERE id=1;
```

执行上述 SQL 语句后，会发现 goods 表中 id 为 1 的记录已经被成功删除。

16.2.4 单表查询

MySQL 从数据表中查询数据的基本语句是 SELECT 语句。在 SELECT 语句中，可以根据自己对数据的需求，使用不同的查询条件，SELECT 语句的基本语法格式如下：

```
SELECT [DISTINCT] * |{字段名 1, 字段名 2, 字段名 3,...}
        FROM 表名
        [WHERE 条件表达式 1]
        [GROUP BY 字段名 [HAVING 条件表达式 2]]
        [ORDER BY 字段名 [ASC|DESC]]
        [LIMIT [OFFSET] 记录数]
```

从上述语法格式中可以看出，一个 SELECT 语句由多个子句组成，其各子句的含义如下所示。

- SELECT[DISTINCT] * |{字段名 1, 字段名 2,...}：“字段 1,字段 2,...”表示从表中查询的指定字段，星号(*)通配符表示表中所有字段，两者为互斥关系，任选其

一。“DISTINCT”是可选参数，用于剔除查询结果中重复的数据。

- FROM 表名：表示从指定的表中查询数据。
- WHERE 条件表达式 1：WHERE 是可选参数，用于指定查询条件。
- GROUP BY 字段名 [HAVING 条件表达式 2]：GROUP BY 是可选参数，用于将查询结果按照指定字段进行分组，HAVING 也是可选参数，用于对分组后的结果进行过滤。
- ORDER BY 字段名 [ASC|DESC]：ORDER BY 是可选参数，用于将查询结果按照指定字段进行排序。排序方式由参数 ASC 或 DESC 控制，其中 ASC 表示按升序进行排列，DESC 表示按降序进行排列。如果不指定参数，则默认为升序排列。
- LIMIT [OFFSET]记录数：LIMIT 是可选参数，用于限制查询结果的数量。LIMIT 后面可以跟两个参数，第一个参数 OFFSET 表示偏移量，如果偏移量为 0，则从查询结果的第一条记录开始，如果偏移量为 1，则从查询结果的第二条记录开始，以此类推。OFFSET 为可选值，如果不指定其默认值为 0。第二个参数记录数表示返回查询记录的条数。

SELECT 语句相对来说比较复杂，对于初学者来说目前可能无法完全理解。下面将通过具体的案讲解 SELECT 语句的使用。

1. 查询所有字段

在 SELECT 语句中，可以通过列出所有字段名查询表中的所有数据，其语法格式如下所示：

```
SELECT  字段名 1,字段名 2,... FROM  表名
```

在上述语法格式中，字段名 1，字段名 2 表示查询的字段名，这里需要列出表中所有的字段名。

除了使用列出所有字段名的方式以外，MySQL 中还可以使用星号(*)通配符代替所有的字段名，其语法格式如下所示：

```
SELECT * FROM 表名;
```

下面通过简单的案例演示如何使用 SELECT 语句查询所有字段。在查询之前，先向 goods 表中插入一些数据，其 SQL 语句如下所示：

```
INSERT INTO goods(id,name,price)
    VALUES (1,'铅笔',0.5),
           (2,'钢笔',5.0),
           (3,'圆珠笔',1.5),
           (4,'水彩笔',7.5),
           (5,'尺子',2.0),
           (6,'橡皮',1.5),
           (7,'书包',35.0),
           (8,'计算器',9.5);
```

查询 goods 表中所有字段的 SQL 语句如下所示：

```
SELECT id,name,price FROM goods;
```

或

```
SELECT * FROM goods;
```

执行上述 SQL 语句后，其查询结果如图 16-10 所示。

需要注意的是，虽然使用星号(*)通配符的方式比较简单，并且同样可以查出表中的所有字段数据，但其执行结果只能按照字段在表中定义的顺序显示。

2. 带关系运算符的查询

在查询操作中，最常见的操作就是使用 WHERE 子句的指定查询条件对数据进行过滤，其语法格式如下所示：

```
SELECT 字段名 1,字段名 2,...
FROM 表名
WHERE 条件表达式
```

在上述语法格式中，条件表达式是 SELECT 语句的查询条件。在 MySQL 中，可以使用关系运算符连接操作数作为条件表达式的查询条件，其常用的关系运算符有大于(>)、小于(<)、等于(=)、不等于(!=或<>)、小于等于(<=)和大于等于(>=)。

以查询 goods 表中 id 为 3 的商品信息为例，其 SQL 语句如下所示：

```
SELECT * FROM goods WHERE id=3;
```

执行结果如图 16-11 所示。

	id	name	price
□	1	铅笔	0.5
□	2	钢笔	5
□	3	圆珠笔	1.5
□	4	水彩笔	7.5
□	5	尺子	2
□	6	橡皮	1.5
□	7	书包	35
□	8	计算器	9.5

图 16-10 查询结果

	id	name	price
□	3	圆珠笔	1.5

图 16-11 执行结果

从执行结果中可以看到，id 为 3 的商品信息只有一条，其他均不满足查询条件。

3. 带 BETWEEN AND 关键字的查询

BETWEEN AND 用于判断某个字段的值是否在指定的范围之内，如果字段的值在指定范围内，则满足条件，该字段所在的记录将被查询出来，反之则不会被查询出来。其语法格式如下所示：

```
SELECT *|{字段名 1,字段名 2,...}
FROM 表名
WHERE 字段名 [NOT] BETWEEN 值 1 AND 值 2
```

在上述语法格式中，值 1 表示范围条件的起始值，值 2 表示范围条件的结束值。NOT 是可选参数，使用 NOT 表示查询指定范围之外的记录，在通常情况下值 1 小于值 2，否则查询不到任何结果。

以查询 goods 表中 id 值为 3～6 的商品信息为例，其查询 SQL 语句如下所示：

```
SELECT * FROM goods WHERE id BETWEEN 3 AND 6;
```

执行结果如图 16-12 所示。

从执行结果中可以看到，查询出了 id 字段值为 3～6 的所有记录，并且起始值 3 和结束值 6 也包括在内。

	id	name	price
□	3	圆珠笔	1.5
□	4	水彩笔	7.5
□	5	尺子	2
□	6	橡皮	1.5

图 16-12　执行结果

4. 带 LIKE 关键字的查询

虽然使用关系运算符＝可以查询出相应条件的数据，但这要求所查询的数据信息必须准确。那么，当我们不能够准确记住所要查询字段的信息，只是记住了几个字时，该怎么办呢？这时，可以采用 MySQL 中提供的一种带 LIKE 关键字的查询，也就是俗称的模糊查询。

使用 LIKE 关键字的 SELECT 语句的语法格式如下所示：

```
SELECT *|{字段名 1,字段名 2,...}
FROM 表名
WHERE 字段名 [NOT] LIKE '匹配字符串';
```

在上述语法格式中，NOT 是可选参数，使用 NOT 表示查询与指定字符串不匹配的记录。匹配字符串指定用于匹配的字符串，其值可以是一个普通字符串，也可以是包含百分号(%)的通配字符串。百分号称为通配符，在通配字符串中有特殊含义，它会匹配任意长度的字符串，包括空字符串。例如，字符串 c%会匹配以字符 c 开始，任意长度的字符串，如 ct、cut、current 等。

以查找 goods 表中 name 字段值中包含“笔”的商品信息为例，其查询 SQL 语句如下所示：

```
SELECT * FROM goods WHERE NAME LIKE "%笔%";
```

执行上述 SQL 语句后，其结果如图 16-13 所示。

从执行结果中可以看到，已经查询出了 name 字段值中所有包含笔的数据。需要注意的是，百分号通配符可以出现在通配符字符串的任意位置。

5. 带 AND 关键字的多条件查询

在使用 SELECT 语句查询数据时，有时为了使执行结果更加精确，可以使用多个查询条件。在 MySQL 中，提供了一个 AND 关键字，使用 AND 关键字可以连接两个或者多个查询条件，只有满足所有条件的记录才会被返回。其语法格式如下所示：

```
SELECT *|{字段名 1,字段名 2,...}
FROM 表名
WHERE 条件表达式 1 AND 条件表达式 2 [... AND 条件表达式 n];
```

从上述语法格式中可以看到，在 WHERE 关键字的后面有多个条件表达式，每两个条件表达式之间用 AND 关键字分隔。

以查询 goods 表中 id 字段值小于 5，并且 price 字段值为 1.5 的商品信息为例，其 SQL 语句如下所示：

```
SELECT * FROM goods WHERE id<5 AND price=1.5;
```

执行结果如图 16-14 所示。

	id	name	price
□	1	铅笔	0.5
□	2	钢笔	5
□	3	圆珠笔	1.5
□	4	水彩笔	7.5

图 16-13 执行结果

	id	name	price
□	3	圆珠笔	1.5

图 16-14 执行结果

从执行结果中可以看到，已经查询出了一条满足条件的数据信息。

6. 带 OR 关键字的多条件查询

在使用 SELECT 语句查询数据时，也可以使用 OR 关键字连接多个查询条件。与 AND 关键字不同，在使用 OR 关键字时，只要记录满足任意一个条件就会被查询出来。其语法格式如下所示：

```
SELECT * |{字段名 1,字段名 2,...}
FROM 表名
WHERE 条件表达式 1 OR 条件表达式 2 [... OR 条件表达式 n];
```

从上述语法格式中可以看到，在 WHERE 关键字的后面有多个条件表达式，每两个条件表达式之间用 OR 关键字分隔。

以查询 goods 表中 id 字段值小于 5 或者 price 字段值为 1.5 的商品信息为例，其 SQL 语句如下所示：

```
SELECT * FROM goods WHERE id<5 OR price=1.5;
```

执行上述 SQL 语句后，其查询结果如图 16-15 所示。

从执行结果中可以看到，只要记录满足 OR 关键字连接的任意一个条件就会被查询出来，不需要同时满足两个条件表达式。

7. 聚合函数

在实际开发中，经常需要对某些数据进行统计，例如统计某个字段的最大值、最小值、平均值等，为此，MySQL 中提供了聚合函数实现这些功能，常用的聚合函数如表 16-4 所示。

表 16-4 中的聚合函数用于对一组值进行统计，并返回唯一值。下面就以 count()和 sum()为例对聚合函数的用法进行讲解。

例如，查询 goods 表中一共有多少条记录，可以使用如下 SQL 语句：

id	name	price
1	铅笔	0.5
2	钢笔	5
3	圆珠笔	1.5
4	水彩笔	7.5
6	橡皮	1.5

图 16-15　执行结果

表 16-4　聚合函数

函数名称	作　用
COUNT()	返回某列的行数
SUM()	返回某列值的和
AVG()	返回某列的平均值
MAX()	返回某列的最大值
MIN()	返回某列的最小值

```
SELECT COUNT(*) FROM goods;
```

上述 SQL 语句的执行结果如图 16-16 所示。

从执行结果中可以看出，goods 表中一共有 8 条记录。

如果要求出 goods 表中 price 字段的总和，可以使用如下 SQL 语句：

```
SELECT SUM(price) FROM goods;
```

执行结果如图 16-17 所示。

COUNT(*)
8

图 16-16　执行结果

SUM(price)
62.5

图 16-17　执行结果

从执行结果中可以看到，所有商品 price 字段的总和为 62.5。

8. 对查询结果进行排序

从表中查询出来的数据可能是无序的，或者其排列顺序不是用户期望的。为了使执行结果满足用户的要求，可以使用 ORDER BY 对执行结果进行排序，其语法格式如下所示：

```
SELECT 字段名 1,字段名 2,...
FROM 表名
ORDER BY 字段名 1 [ASC | DESC],字段名 2 [ASC | DESC]...
```

在上述语法格式中，指定的字段名 1 和字段名 2 等是对执行结果排序的依据。参数 ASC 表示按照升序进行排序，DESC 表示按照降序进行排序。在默认情况下，按照 ASC 方式进行排序。

使用参数 DESC 按照 id 字段降序方式排列 goods 表中数据的 SQL 语句如下所示：

```
SELECT * FROM goods ORDER BY id DESC;
```

执行结果如图 16-18 所示。

id	name	price
8	计算器	9.5
7	书包	35
6	橡皮	1.5
5	尺子	2
4	水彩笔	7.5
3	圆珠笔	1.5
2	钢笔	5
1	铅笔	0.5

图 16-18　执行结果

9. LIMIT 分页查询

在查询数据时，可能会一次性返回很多条记录，如果这些记录全部在一个页面中显示，那么用户将很难快速找到需要的数据。为此，MySQL 提供了一个关键字 LIMIT，使用它可以指定执行结果从哪一条记录开始以及一共查询多少条信息，其使用语法格式如下所示：

```
SELECT 字段名 1,字段名 2,...
FROM 表名
LIMIT [OFFSET,] 记录数
```

在上述语法格式中，LIMIT 后面可以跟两个参数，第一个参数 OFFSET 表示偏移量，如果偏移量为 0，则从查询结果的第一条记录开始，如果偏移量为 1，则从查询结果的第二条记录开始，以此类推。OFFSET 为可选值，如果不指定，则其默认值为 0。第二个参数记录数表示返回查询记录的条数。

以查询 goods 表中的前 4 条记录为例，其 SQL 语句如下所示：

```
SELECT * FROM goods LIMIT 4;
```

执行结果如图 16-19 所示。

从执行结果中可以看到，执行语句中没有指定返回记录的偏移量，仅指定了查询记录的条数为 4，因此返回结果从第一条记录开始，一共返回 4 条记录。

如果要查询 goods 表中 id 字段值从第 5 位到第 8 位的商品信息(从高到低)，可以通过如下 SQL 语句查询：

```
SELECT * FROM goods ORDER BY id LIMIT 4,4;
```

在上述执行语句中，LIMIT 后面跟了两个参数，第一个参数表示偏移量为 4，即从第 5 条记录开始查询，第二个参数表示一共返回 4 条记录，即从第 5 条到第 8 条的商品信息。

执行上述 SQL 语句后，其查询结果如图 16-20 所示。

	id	name	price
☐	1	铅笔	0.5
☐	2	钢笔	5
☐	3	圆珠笔	1.5
☐	4	水彩笔	7.5

图 16-19 执行结果

	id	name	price
☐	5	尺子	2
☐	6	橡皮	1.5
☐	7	书包	35
☐	8	计算器	9.5

图 16-20 执行结果

16.3 MySQL 的多表操作

第 16.2 节所讲解的内容都是针对一张表的操作，即单表操作。然而在实际的开发中，很多功能都会涉及两张以上的表的操作，即多表操作。本节将针对多表操作的相关知识进行讲解。

16.3.1 外键约束

外键是指在一张表中定义的、能确定另一张表记录的字段。该字段中引用了另一张表

中的一列或多列，被引用的列应该具有主键约束或唯一性约束。外键用于建立和加强两个表数据之间的链接。

为了使初学者更好地理解外键的定义，接下来通过两张表讲解什么是外键。在数据库 chapter16 中创建一个班级表(t_grade)和一个学生表(t_student)，具体语句如下所示：

```
USE chapter16;
CREATE TABLE t_grade(
    id int(5) NOT NULL PRIMARY KEY,
    name varchar(40)
);
CREATE TABLE t_student(
    sid int(5) NOT NULL PRIMARY KEY,
    sname varchar(40),
    gid int(5) NOT NULL
);
```

在上述两个 SQL 建表语句中，如果学生表(t_student)中的 gid 字段的值是学生所在的班级 id，并且它引入了班级表(t_grade)中的主键 id，那么 gid 就可以作为表 t_student 的外键。在这两个表中，被引用的 t_grade 表是主表；引用外键的 t_student 表是从表，它们是主从关系。

要想真正连接两张表的数据，就需要为表添加外键约束。为表添加外键约束的语法格式如下所示：

```
alter table 表名 add constraint FK_ID foreign key(外键字段名) REFERENCES 外表表名
(主键字段名);
```

为 t_student 表添加外键约束的语句如下所示：

```
alter tablet_student add constraint FK_ID foreign key(gid) REFERENCES t_grade
(id);
```

上述 SQL 语句执行成功后，两张表就已经通过外键关联了起来。通过 SQLyog 的资料窗口，可以查看所执行操作的详细信息，如图 16-21 所示。

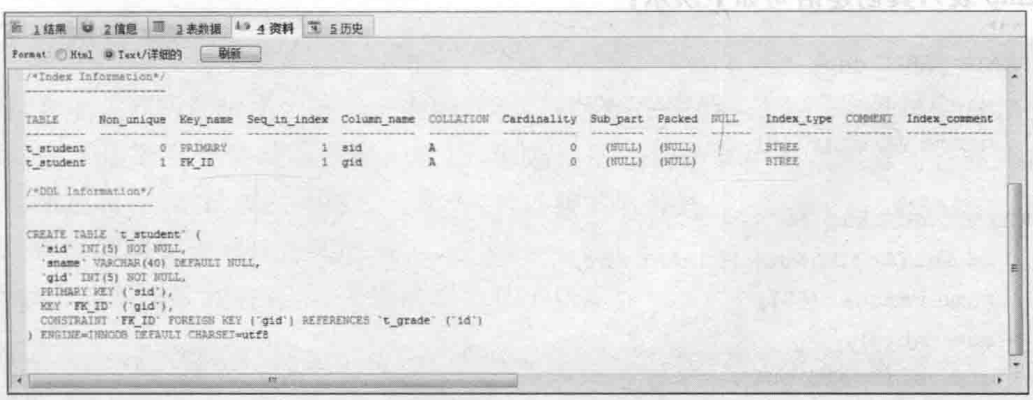


图 16-21　执行结果

从图 16-21 中可以明确地看出，gid 为 t_student 表的外键，并且 gid 外键依赖于 t_grade 表中的 id 主键。

成功添加外键后，t_student 表和 t_grade 表之间就已经形成多对一的关系。因为外键列只能插入参照列存在的值，所以如果要为两个表添加数据，就需要先为主表 t_grade 添加数据，再为从表 t_student 添加数据，具体如下所示：

```
INSERT INTO t_grade(id,name)VALUES(1,'一班');
INSERT INTO t_grade(id,name)VALUES(2,'二班');
INSERT INTO t_student(sid,sname,gid)VALUES(1,'张三',1);
INSERT INTO t_student(sid,sname,gid)VALUES(2,'李四',1);
INSERT INTO t_student(sid,sname,gid)VALUES(3,'王五',2);
INSERT INTO t_student(sid,sname,gid)VALUES(4,'赵六',2);
```

添加成功后，t_grade 表和 t_student 表中的数据如图 16-22 和图 16-23 所示。

id	name
1	一班
2	二班

图 16-22 t_grade 表中的数据

sid	sname	gid
1	张三	1
2	李四	1
3	王五	2
4	赵六	2

图 16-23 t_student 表中的数据

16.3.2 内连接

内连接(INNER JOIN)又称简单连接或自然连接，是一种常见的连接查询。内连接使用比较运算符对两个表中的数据进行比较，并列出与连接条件匹配的数据行，组合成新的记录，也就是说在内连接查询中，只有满足条件的记录才能出现在查询结果中。

内连接查询的语法格式如下所示：

```
SELECT 查询字段 FROM 表 1 [INNER] JOIN 表 2 ON 表 1.关系字段=表 2.关系字段
```

在上述语法格式中，INNER JOIN 用于连接两个表，ON 用于指定连接条件，其中 INNER 可以省略。

下面通过具体的案例演示如何实现内连接。在 chapter16 数据库中创建两个表(dept 表和 emp 表)，其创建语句如下所示：

```
CREATE TABLE dept (
    did int(5) NOT NULL PRIMARY KEY,
    dname varchar(40)
);
CREATE TABLE emp (
    id int(5) NOT NULL PRIMARY KEY,
    name varchar(40),
    age int(3),
    did int(5) NOT NULL
);
```

上述语句执行成功后，在两个表中分别插入相关数据，具体语句如下所示：

```
INSERT INTO dept(did,dname)VALUES(1,'运维部');
INSERT INTO dept(did,dname)VALUES(2,'测试部');
INSERT INTO dept(did,dname)VALUES(3,'研发部');
INSERT INTO dept(did,dname)VALUES(5,'人事部');
INSERT INTO emp(id,name,age,did)VALUES(1,'张三',21,1);
INSERT INTO emp(id,name,age,did)VALUES(2,'李四',25,1);
INSERT INTO emp(id,name,age,did)VALUES(3,'王五',21,2);
INSERT INTO emp(id,name,age,did)VALUES(4,'赵六',23,4);
```

添加成功后，使用内连接查询 dept 表和 emp 表数据的 SQL 语句如下所示：

```
SELECT emp.name, dept.dname FROM dept JOIN emp ON dept.did=emp.did;
```

上述语句执行成功后，结果如图 16-24 所示。

从上述结果中可以看出，只有 dept. did 与 emp. did 相等的员工才会被显示。

需要注意的是，在 MySQL 中，还可以使用 WHERE 条件语句实现同样的功能，具体如下所示：

```
SELECT emp.name, dept.dname FROM dept, emp WHERE dept.did=emp.did;
```

上述语句执行成功后，结果如图 16-25 所示。

	name	dname
□	张三	运维部
□	李四	运维部
□	王五	测试部

图 16-24　执行结果

	name	dname
□	张三	运维部
□	李四	运维部
□	王五	测试部

图 16-25　执行结果

从执行结果中可以看出，使用 WHERE 子句的查询结果与使用 INNER JOIN 的查询结果是一致的。

需要注意的是，这两个语句的查询结果虽然相同，但是 INNER JOIN 是内连接语句，WHERE 是条件判断语句，在 WHERE 语句后可以直接添加其他条件，而 INNER JOIN 语句则不可以。

16.3.3　外连接

在第 16.3.2 节所讲解的内连接查询中，返回的结果只包含符合查询条件和连接条件的数据，然而有时还需要包含没有关联的数据，即返回的查询结果中不仅包含符合条件的数据，还要包括左表(左连接或左外连接)、右表(右连接或右外连接)或两个表(全外连接)中的所有数据，此时就需要使用外连接查询。

外连接分为左连接和右连接，其语法格式如下所示：

```
SELECT 所查字段 FROM 表 1 LEFT|RIGHT [OUTER] JOIN 表 2
ON 表 1.关系字段=表 2.关系字段 WHERE 条件
```

外连接的语法格式和内连接类似，只不过使用的是 LEFT JOIN 和 RIGHT JOIN 关键字，其中关键字左边的表被称为左表，关键字右边的表被称为右表。

在使用左连接和右连接查询时，查询结果是不一致的，具体如下。

- LEFT JOIN(左连接)：返回包括左表中的所有记录和右表中符合连接条件的记录。
- RIGHT JOIN(右连接)：返回包括右表中的所有记录和左表中符合连接条件的记录。

1. LEFT JOIN(左连接)

左连接的结果包括 LEFT JOIN 子句中指定的左表的所有记录和所有满足连接条件的记录。如果左表中的某条记录在右表中不存在，则在右表中显示为空。

在 dept 表和 emp 表之间使用左连接查询，SQL 语句如下所示：

```
SELECT dept.did, dept.dname, emp.name FROM dept LEFT JOIN emp on dept.did=
emp.did;
```

上述语句执行成功后的返回结果如图 16-26 所示。

从上述结果中可以看出，显示了 5 条记录，并且人事部没有 did 等于 5 的员工。

2. RIGHT JOIN(右连接)

右连接与左连接正好相反，它会返回右表中的所有记录和所有满足连接条件的左表记录。如果右表中的某条记录在左表中没有匹配，则左表将返回空值。

在 dept 表和 emp 表之间使用右连接查询的 SQL 语句如下所示：

```
SELECT dept.did, dept.dname, emp.name FROM dept RIGHT JOIN emp ON dept.did=
emp.did;
```

上述语句执行成功后，结果如图 16-27 所示。

did	dname	name
1	运维部	张三
1	运维部	李四
2	测试部	王五
3	研发部	(NULL)
5	人事部	(NULL)

图 16-26 执行结果

did	dname	name
1	运维部	张三
1	运维部	李四
2	测试部	王五
(NULL)	(NULL)	赵六

图 16-27 执行结果

从上述结果中可以看出，显示了 4 条记录，并且 name 值为赵六的员工并没有被分配部门。

16.3.4 子查询

子查询是指一个查询语句嵌套在另一个查询语句内部的查询，它可以嵌套在一个 SELECT、SELECT...INTO 语句或 INSERT...INTO 等语句中。在执行查询语句时，首先会执行子查询中的语句，然后将返回的结果作为外层查询的过滤条件进行查询。在子查询中，通常可以使用 IN、EXISTS、ANY、ALL 操作符。下面将针对子查询进行详细讲解。

1. 带 IN 关键字的子查询

使用 IN 关键字进行子查询时，内层查询语句仅返回一个数据列，这个数据列中的值将供外层查询语句进行比较操作。

例如，要查询存在年龄为 21 岁的员工的部门，其 SQL 语句如下所示：

```
SELECT * FROM dept WHERE did IN(SELECT did FROM emp WHERE age=21);
```

上述语句执行成功后的结果如图 16-28 所示。

从上述结果中可以看出，只有运维部和测试部有年龄为 21 岁的员工。在查询的过程中，首先会执行内层子查询，得到年龄为 21 岁的员工的部门 did，然后根据部门 did 与外层查询的比较条件进行查询，从而最终得到符合条件的数据。

需要注意的是，SELECT 语句中还可以使用 NOT IN 关键字进行查询，其作用正好与 IN 相反。

2. 带 ANY 关键字的子查询

ANY 关键字表示满足其中任意一个条件，它允许创建一个表达式对子查询的返回值列表进行比较，只要满足内层子查询中的任意一个比较条件，就返回一个结果作为外层查询条件。

使用带 ANY 关键字查询满足条件部门的 SQL 语句如下所示：

```
SELECT * FROM dept WHERE did>any(select did from emp);
```

上述语句执行成功后的结果如图 16-29 所示。

	did	dname
□	1	运维部
□	2	测试部

图 16-28　执行结果

	did	dname
□	2	测试部
□	3	研发部
□	5	人事部

图 16-29　执行结果

上述语句在执行的过程中，首先会将子句的 emp 表中的所有 did 查询出来，然后将 dept 表中 did 的值与之进行比较，只要大于 emp. did 中的任意一个值，就是符合条件的查询结果。由于 dept 表中的测试部、研发部、人事部的 did 都大于 emp 表中的 did(did=1)，因此输出结果为测试部、研发部和人事部。

3. 带 ALL 关键字的子查询

ALL 关键字与 ANY 关键字的作用有点类似，只不过带 ALL 关键字的子查询返回的结果需要同时满足所有内层查询条件。

使用带 ALL 关键字的子查询，查询满足条件部门的 SQL 语句如下所示：

```
SELECT * FROM dept WHERE did>all(select did from emp);
```

上述语句执行成功后的结果如图 16-30 所示。

上述语句在执行的过程中，首先子查询会将 emp 表中的所有 did 查询出来，然后将 dept 表中 did 的值与之进行比较，只有大于 emp. did 的所有值，才是符合条件的执行结果。由于只有人事部的 did 值大于 emp 表中的所有 did 值，因此最终得到的查询结果为人事部。

4. 带比较运算符的子查询

在前面讲解的 ANY 关键字和 ALL 关键字的子查询中都使用了＞比较运算符，子查询中还可以使用其他的比较运算符，如＜、＞＝、＝、!＝等。

使用带比较运算符的子查询，查询李四是哪个部门员工的 SQL 语句如下所示：

```
SELECT * FROM dept WHERE did=(select did from emp where name='李四');
```

上述语句执行成功后的结果如图 16-31 所示。

	did	dname
□	5	人事部

图 16-30 执行结果

	did	dname
□	1	运维部

图 16-31 执行结果

从图 16-31 所示的执行结果中可以看出，李四是运维部的员工。

16.4 JDBC 技术简介

在实际项目中，数据库的操作都是通过程序完成的，不同的开发语言也都对不同的数据库提供了支持，Java 也不例外，它为了能够操作数据库，提供了一套访问数据库的标准 Java 类库，即 JDBC。

JDBC(Java Database Connectivity)是一套用于执行 SQL 语句的 Java API(主要位于 java. sql 包中)，应用程序通过这套 API 可以连接到关系型数据库，并使用 SQL 语句对数据库中的数据进行添、删、改、查等操作。

不同类型的数据库(如 Oracle、MySQL)其内部处理数据的方式是不同的，如果直接使用数据库厂商提供的访问接口操作数据库，会导致应用程序的可移植性变差。例如，当前用户使用的是 Oracle 提供的接口操作数据库，如果需要更换数据库，则需要重新编写操作数据库的部分代码，这样导致了程序的可移植性非常差。如果使用 JDBC 的 API 操作数据库就不会出现这种情况。JDBC 要求各个数据库厂商按照统一标准提供数据库驱动，应用程序直接通过 JDBC 和数据库驱动连接数据库，开发人员就不需要了解底层数据库的交互，大大提高程序的可移植性。

应用程序通过 JDBC 访问数据库的过程如图 16-32 所示。

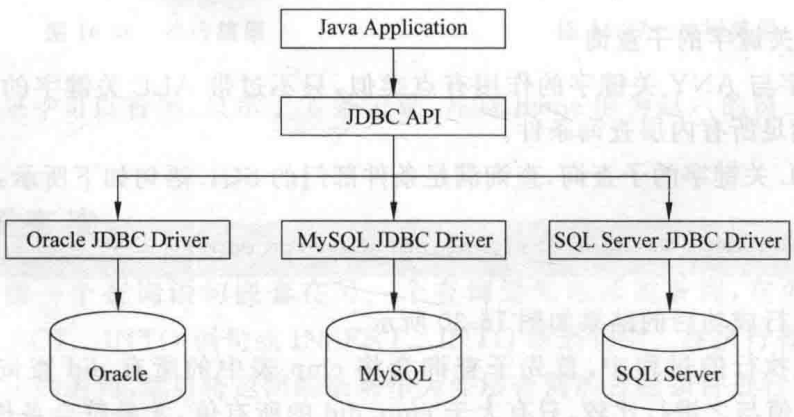


图 16-32 使用 JDBC 访问数据库

从图 16-32 中可以看出，JDBC 是连接 Java 程序与数据库驱动之间的桥梁，当应用程序使用 JDBC 访问某个数据时，只需要通过数据库驱动连接指定的数据库即可，无须直接的操作数据库。

16.5　JDBC 常用 API

16.5.1　Driver 接口

Driver 是每个驱动程序类必须实现的接口，专门提供给数据厂商使用。需要注意的是，在编写 JDBC 程序时，必须要把所使用的数据库驱动程序和类库（指 MySQL 的驱动 JAR 包）加载到项目的 classpath 中。

16.5.2　DriverManager 类

DriverManager 类负责管理 JDBC 驱动程序，其作用于用户和驱动程序之间，负责跟踪可用的驱动程序，并在数据库和相应的驱动程序之间建立连接。在 DriverManager 类中，定义了两个静态方法，如表 16-5 所示。

表 16-5　DriverManager 类的方法

方法名称	功能描述
static Connection getConnection(String url,String user,String pwd)	建立到给定数据库 URL 的连接，并返回表示连接的 Connection 对象
static void registerDriver(Driver driver)	向 DriverManager 注册给定驱动程序

16.5.3　Connection 接口

Connection 接口用于创建数据库的连接，只有获得该连接对象后才可以访问数据库，并操作数据表。在 Connection 接口中，定义了一系列方法，其常用方法如表 16-6 所示。

表 16-6　Connection 接口中的常用方法

方法名称	功能描述
Statement createStatement()	创建一个 Statement 对象从而将 SQL 语句发送到数据库
PreparedStatement prepareStatement (String sql)	创建一个 PreparedStatement 对象从而将参数化的 SQL 语句发送到数据库

16.5.4　Statement 接口

Statement 接口用于将普通的 SQL 语句发送到数据库中。该接口的对象通过 Connection. createStatement()方法获得。利用该对象把普通的 SQL 语句发送到数据库进行编译，然后返回数据库的处理结果。在 Statement 接口中，提供了三个常用的执行 SQL 语句的方法，如表 16-7 所示。

表 16-7 Statement 接口中的方法

方法名称	功能描述
boolean execute(String sql)	用于执行给定的 SQL 语句，该语句可能返回多个结果
int executeUpdate(String sql)	用于执行给定的 SQL 语句，该语句可能为 INSERT、UPDATE 或 DELETE 语句，或者是不返回任何内容的 SQL 语句
ResultSet executeQuery(String sql)	用于执行给定的 SQL 语句，该语句返回单个 ResultSet 对象

16.5.5 PreparedStatement 接口

Statement 接口封装了 JDBC 执行 SQL 语句的方法，Statement 对象每次执行 SQL 语句时，都会进行一次编译，这样大大降低了程序的执行效率。为此，JDBC API 提供了一个 PreparedStatement 接口，PreparedStatement 是 Statement 的子接口，拥有 Statement 接口的所有方法，并且可以对 SQL 语句进行预编译，预编译后的数据会存储在 PreparedStatement 对象中。当执行相同的 SQL 语句时，程序会使用 PreparedStatement 对象中的数据，而不需要对 SQL 语句再次进行编译，这样就极大地提高了对数据库的访问效率。

PreparedStatement 接口还扩展了带有参数 SQL 语句的执行操作，应用于该接口中的 SQL 语句可以使用占位符“?”代替其参数，然后通过 setXxx()方法为 SQL 语句的参数赋值。在 PreparedStatement 接口中，提供了一些常用方法，如表 16-8 所示。

表 16-8 PreparedStatement 接口中的方法

方法名称	功能描述
int executeUpdate()	在此 PreparedStatement 对象中执行 SQL 语句，该语句必须是一个 SQL 数据操作语言(Data Manipulation Language，DML)语句，如 INSERT、UPDATE 或 DELETE 语句；或者是无返回内容的 SQL 语句，如 DDL 语句
ResultSet executeQuery()	在此 PreparedStatement 对象中执行 SQL 查询，并返回该查询生成的 ResultSet 对象
void setInt(int parameterIndex, int x)	将指定参数设置为给定的 int 值
void setFloat(int parameterIndex, float x)	将指定参数设置为给定的 float 值
void setString(int parameterIndex, String x)	将指定参数设置为给定的 String 值
void setDate(int parameterIndex, Date x)	将指定参数设置为给定的 Date 值

需要注意的是，表 16-8 中的 setDate()方法可以设置日期内容，但参数 Date 的类型是 java.sql.Date，而不是 java.util.Date。

在通过 setXxx()方法为 SQL 语句中的参数赋值时，可以通过参数与 SQL 类型相匹配的方法(例如，如果参数具有的 SQL 类型为 String，那么应该使用 setString 方法)，也可以通过 setObject()方法设置多种类型的输入参数。具体如下所示：

```
String sql=" SELECT * FROM goods WHERE id<? AND name=? AND price=?";
PreparedStatement ps=conn.prepareStatement(sql);
ps.setInt(1, 1);                    //使用参数的已定义 SQL 类型
```

```
ps.setString(2, "水彩笔");          //使用参数的已定义 SQL 类型
ps.setObject(3, 7.5);               //使用 setObject()方法设置参数
ps.executeUpdate();
```

16.5.6　ResultSet 接口

ResultSet 接口用于保存 JDBC 执行查询时返回的结果集，该结果集与数据库表字段相对应。在 ResultSet 接口内部有一个指向表格数据行的游标（或指针），ResultSet 接口对象的游标最初位于结果集的第一行的前面，当执行一次 next()方法之后才会将指针指向第一个记录。每调用一次 next()方法，数据库游标向下移动一行，直到记录集最后一行。在应用程序中经常使用 next()方法作为 while 循环的条件以迭代 ResultSet 结果集。ResultSet 接口中的常用方法如表 16-9 所示。

表 16-9　ResultSet 接口中的常用方法

方 法 名 称	功 能 描 述
String getString(int columnIndex)	以 Java 编程语言中 String 的形式获取此 ResultSet 对象的当前行中指定列的值
String getString(String columnName)	以 Java 编程语言中 String 的形式获取此 ResultSet 对象的当前行中指定列的值
int getInt(int columnIndex)	以 Java 编程语言中 int 的形式获取此 ResultSet 对象的当前行中指定列的值
int getInt(String columnName)	以 Java 编程语言中 int 的形式获取此 ResultSet 对象的当前行中指定列的值
Date getDate(int columnIndex)	以 Java 编程语言中 java. sql. Date 对象的形式获取此 ResultSet 对象的当前行中指定列的值
Date getDate(String columnName)	以 Java 编程语言中的 java. sql. Date 对象的形式获取此 ResultSet 对象的当前行中指定列的值
boolean next()	将光标从当前位置向前移一行

16.6　实现第一个 JDBC 程序

通过前两节的学习，读者对 JDBC 及其常用 API 已经有了大致的了解，本节将讲解如何使用 JDBC 的常用 API 实现一个 JDBC 程序。

通常，JDBC 的使用可以按照以下几个步骤进行。

1. 加载并注册数据库驱动

注册数据库驱动的具体方式如下所示：

```
DriverManager.registerDriver(Driver driver);
```

或

```
Class.forName("DriverName");
```

2. 通过 DriverManager 获取数据库连接

获取数据库连接的具体方式如下所示：

```
Connection conn = DriverManager.getConnection(String url, String user, String pwd);
```

从上述代码中可以看出，getConnection()方法中有三个参数，它们分别表示连接数据库的 URL 地址、登录数据库的用户名和密码。以 MySQL 数据库为例，其 URL 地址的书写格式如下所示：

```
jdbc:mysql://hostname:port/databasename
```

在上述代码中，jdbc:mysql:是固定的写法，mysql 指 MySQL 数据库。hostname 指主机的名称(如果数据库在本机中，则 hostname 可以为 localhost 或 127.0.0.1；如果要连接的数据库在其他计算机上，则 hostname 为所要连接计算机的 IP)，port 指连接数据库的端口号(MySQL 端口号默认为 3306)，而 databasename 指 MySQL 中相应数据库的名称。

3. 通过 Connection 对象获取 Statement 对象

Connection 创建 Statement 的方式有以下三种。

(1) createStatement()：创建基本的 Statement 对象。

(2) prepareStatement()：创建 PreparedStatement 对象。

(3) prepareCall()：创建 CallableStatement 对象。

以创建基本的 Statement 对象为例，其创建方式如下所示：

```
Statement stmt=conn.createStatement();
```

4. 使用 Statement 执行 SQL 语句

所有的 Statement 都有以下三种执行 SQL 语句的方法。

(1) execute()：可以执行任何 SQL 语句。

(2) executeUpdate()：主要用于执行 DML 和 DDL 语句。执行 DML 语句，如 INSERT、UPDATE 或 DELETE 时，返回受 SQL 语句影响的行数，执行 DDL 语句返回 0。

(3) executeQuery()：通常执行查询语句，执行后返回代表结果集的 ResultSet 对象。

以 executeQuery()方法为例，其使用方式如下所示：

```
//执行 SQL 语句，获取结果集 ResultSet
ResultSet rs=stmt.executeQuery(sql);
```

5. 操作 ResultSet 结果集

如果执行的 SQL 语句是查询语句，则执行结果将返回一个 ResultSet 对象，该对象中保存了 SQL 语句查询的结果。程序可以通过操作该 ResultSet 对象取出执行结果。

6. 关闭连接并释放资源

每次操作数据库结束后都要关闭数据库连接，释放资源，包括关闭 ResultSet、

Statement 和 Connection 等资源。

下面依照上面所讲解的步骤，通过案例演示如何通过 JDBC 实现对数据库的操作。

1. 创建数据库环境

本案例以数据库 chapter16 中 goods 表为例演示操作，此时 goods 表中的数据如图 16-33 所示。

2. 搭建项目环境和导入数据库驱动

在 Eclipse 中新建一个名称为 chapter16 的 Web 项目，将下载好的 MySQL 的数据库驱动文件 mysql-connector-java-5.0.8-bin.jar 复制到项目的 lib 目录中，并发布到类路径下（MySQL 驱动文件可以在其官网地址 http://dev.mysql.com/downloads/connector/j/ 页面中下载，在浏览器中输入该地址后，即可进入下载页面，单击页面 Generally Available (GA) Releases 窗口中的 Looking for previous GA versions 超链接后，在显示的下拉框中，下载所需的驱动版本即可）。加入驱动后的项目结构如图 16-34 所示。

	id	name	price
□	1	铅笔	0.5
□	2	钢笔	5
□	3	圆珠笔	1.5
□	4	水彩笔	7.5
□	5	尺子	2
□	6	橡皮	1.5
□	7	书包	35
□	8	计算器	9.5

图 16-33　goods 表中的数据

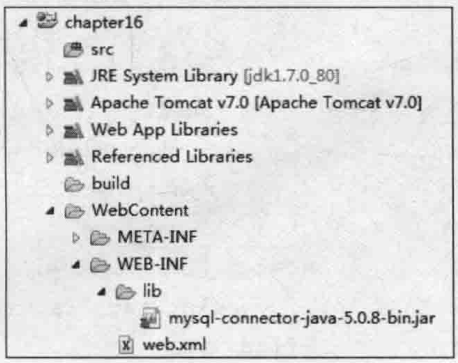


图 16-34　项目结构

3. 编写 JDBC 程序

在项目 chapter16 的 src 目录下，新建一个名称为 com.itheima.jdbc 的包，在该包中创建类 GoodsDao，该类用于读取数据库中 goods 表的数据信息，并将表中所有数据输出到控制台，其代码如文件 16-1 所示。

文件 16-1　GoodsDao.java

```
1  package com.itheima.jdbc;
2  import java.sql.Connection;
3  import java.sql.DriverManager;
4  import java.sql.ResultSet;
5  import java.sql.SQLException;
6  import java.sql.Statement;
7  public class GoodsDao {
8      public void findAllGoods() {
9          Statement stmt=null;
10         ResultSet rs=null;
11         Connection conn=null;
12         try {
```

```
13            //1. 通过反射加载数据库的驱动
14            Class.forName("com.mysql.jdbc.Driver");
15            //2.通过 DriverManager 获取 Connection 连接对象
16            String url="jdbc:mysql://localhost:3306/chapter16";
17            String username="root";
18            String password="root";
19            conn=DriverManager.getConnection(url, username, password);
20            //3.通过 Connection 对象获取 Statement 对象
21            stmt=conn.createStatement();
22            //4.使用 Statement 执行 SQL 语句
23            String sql="select * from goods";
24            rs=stmt.executeQuery(sql);
25            //5. 操作 ResultSet 结果集
26            System.out.println("id\t| name\t| price\t ");
27            while (rs.next()) {
28                int id=rs.getInt("id");          //通过列名获取指定字段的值
29                String name=rs.getString("name");
30                String price=rs.getString("price");
31                String number=rs.getString("number");
32                System.out.println(id+"\t|"+name+"\t|"+price+"\t");
33            }
34        } catch (Exception e) {
35            e.printStackTrace();
36        } finally {
37            if (rs !=null) {
38                try {
39                    rs.close();
40                } catch (SQLException e) {
41                    e.printStackTrace();
42                }
43            }
44            rs=null;
45            try {
46                stmt.close();
47            } catch (SQLException e) {
48                e.printStackTrace();
49            }
50            stmt=null;
51            try {
52                conn.close();
53            } catch (SQLException e) {
54                e.printStackTrace();
55            }
56            conn=null;
```

```
57            }
58        }
59    }
```

在文件 16-1 中，首先注册了 MySQL 数据库驱动，通过 DriverManager 获取一个 Connection 对象，然后使用 Connection 对象创建一个 Statement 对象，Statement 对象通过 executeQuery()方法执行 SQL 语句，并返回结果集 ResultSet，下面通过遍历 ResultSet 得到最终的执行结果，最后关闭连接，回收了数据库资源。

4. 编写测试类

在 com. itheima. jdbc 包下创建一个名为 FirstDemoTest 的类，编辑后如文件 16-2 所示。

文件 16-2　FirstDemoTest. java

```
1   package com.itheima.jdbc;
2   public class FirstDemoTest {
3       public static void main(String[] args) {
4           //调用 GoodsDao 类的 findAllGoods()方法
5           new GoodsDao().findAllGoods();
6       }
7   }
```

测试类程序执行成功后，控制台的显示结果如图 16-35 所示。

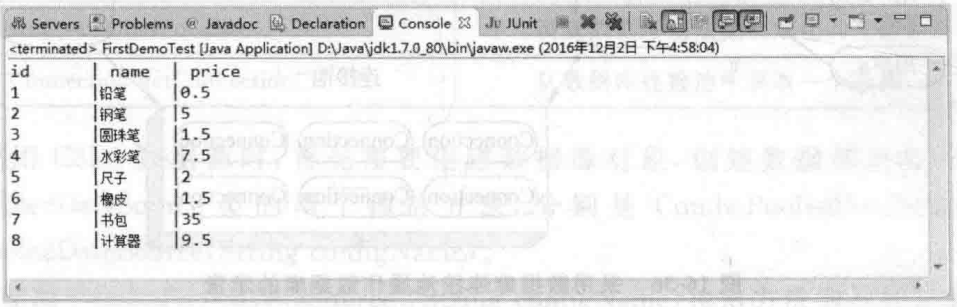


图 16-35　运行结果

从图 16-35 中可以看到，goods 表中的数据已经被显示在了控制台。至此第一个 JDBC 程序实现成功。

脚下留心：

在实现第一个 JDBC 程序时，还有两个地方需要注意，具体如下：

1. 注册驱动

虽然使用 DriverManager. registerDriver(new com. mysql. jdbc. Driver())方法也可以完成注册，但此方法会使数据库驱动被注册两次。这是因为 Driver 类的源码中，已经在静态代码块中完成了数据库驱动的注册。所以，为了避免数据库驱动被重复注册，只需要在程序中使用 Class. forName()方法加载驱动类即可。

2. 释放资源

每次操作数据库结束后都要及时关闭数据库资源，释放连接占用的数据库和 JDBC 资源，以免影响系统的运行速度。为了保证资源的释放，在 Java 程序中，应该将最终必须要执行的操作放在 finally 代码块中。

16.7 数据库连接池

16.7.1 数据库连接池

在 JDBC 编程中，每次创建和断开 Connection 对象都会消耗一定的时间和 I/O 资源。这是因为每一次的数据访问请求都必须经历建立数据库连接（创建 Connection）、打开数据库、存取数据和关闭数据库连接等步骤，而数据库处理通常是整个业务处理中最频繁的步骤，并且在各种数据库操作步骤中，数据库的连接和释放通常又特别耗时，这必然导致系统性能急剧下降甚至崩溃。

为了避免频繁地创建和释放数据库连接，数据库连接池技术应运而生。数据库连接池负责分配、管理和释放数据库连接，它允许应用程序重复使用现有的数据库连接，而不是重新建立。应用程序如何通过连接池连接数据库如图 16-36 所示。

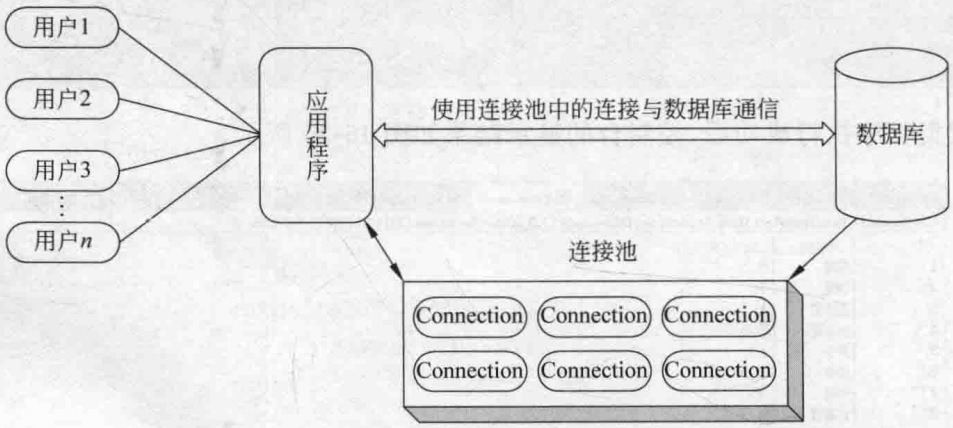


图 16-36 采用数据库连接池操作数据库的示意

从图 16-36 中可以看出，数据库连接池在初始化时将创建一定数量的数据库连接放到连接池中，当应用程序访问数据库时并不是直接创建 Connection，而是向连接池“申请”一个 Connection。如果连接池中有空闲的 Connection，则将其返回，否则创建新的 Connection。使用完毕后，连接池会将该 Connection 回收，并交付给其他的线程使用，以减少创建和断开数据库连接的次数，提高数据库的访问效率。

16.7.2 DataSource 接口

DataSource 接口用于表示数据源，顾名思义，数据源即数据的来源。在数据源中存储了所有建立数据库连接的信息。就像通过指定文件名称可以在文件系统中找到文件一样，通过提供正确的数据源名称，也可以找到相应的数据库连接。在 DataSource 接口中，定义了两个返回值为 Connection 对象的方法，具体如下：

- Connection getConnection();
- Connection getConnection(String username, String password)。

上述两个重载的方法，都能用于获取 Connection 对象。不同的是，第一个方法是通过无参的方式建立与数据库的连接；第二个方法是通过传入登录信息的方式建立与数据库的连接。

16.7.3　C3P0 数据源

C3P0 是一个开放源代码的 JDBC 连接池，它包括了实现 jdbc3 和 jdbc2 扩展规范说明的 Connection 和 Statement 的 DataSources 对象。在使用 C3P0 数据源开发时，需要了解 C3P0 中 DataSource 接口的实现类 ComboPooledDataSource，它是 C3P0 的核心类，提供了数据源对象的相关方法，该类的常用方法介绍如表 16-10 所示。

表 16-10　ComboPooledDataSource 类的常用方法

方法名称	功能描述
void setDriverClass()	设置连接数据库的驱动名称
void setJdbcUrl()	设置连接数据库的路径
void setUser()	设置数据库的登录账号
void setPassword()	设置数据库的登录密码
void setMaxPoolSize()	设置数据库连接池最大的连接数目
void setMinPoolSize()	设置数据库连接池最小的连接数目
void setInitialPoolSize()	设置数据库连接池初始化的连接数目
Connection getConnection()	从数据库连接池中获取一个连接

当使用 C3P0 数据源时，首先需要创建数据源对象，创建数据源对象可以使用 ComboPooledDataSource 类的两个构造方法，分别是 ComboPooledDataSource() 和 ComboPooledDataSource(String configName)。

下面使用 ComboPooledDataSource (String configName)构造方法演示如何通过读取配置文件创建数据源对象，具体步骤如下。

1. 导入 JAR 包并创建配置文件

在 chapter16 中导入 C3P0 程序所需的 JAR 包 c3p0-0.9.1.2.jar，然后在 src 根目录下创建一个 c3p0-config.xml 文件，该文件用于设置数据库的连接信息和数据源的初始化信息，如文件 16-3 所示。

文件 16-3　c3p0-config.xml

```
1  <? xml version="1.0" encoding="UTF-8"? >
2  <c3p0-config>
3      <default-config>
4          <property name="driverClass">com.mysql.jdbc.Driver</property>
```

```
        <property name="jdbcUrl">jdbc:mysql://localhost:3306/chapter16
        </property>
        <property name="user">root</property>
        <property name="password">root</property>
        <property name="checkoutTimeout">30000</property>
        <property name="initialPoolSize">10</property>
        <property name="maxIdleTime">30</property>
        <property name="maxPoolSize">100</property>
        <property name="minPoolSize">10</property>
        <property name="maxStatements">200</property>
    </default-config>
    <named-config name="itheima">
        <property name="driverClass">com.mysql.jdbc.Driver</property>
        <property name="jdbcUrl">jdbc:mysql://localhost:3306/chapter16
        </property>
        <property name="user">root</property>
        <property name="password">root</property>
        <property name="initialPoolSize">6</property>
        <property name="maxPoolSize">20</property>
    </named-config>
</c3p0-config>
```

在文件 16-3 中，c3p0-config. xml 文件提供了两种配置方式，其中＜default-config＞...＜/default-config ＞是默认配置，在没有指定配置时默认使用该配置创建 C3P0 数据源对象；＜named-config＞...＜/ named-config ＞是自定义配置，一个配置文件中可以有零个或多个自定义配置，当用户需要使用自定义配置时，调用 ComboPooledDataSource(String configName)方法，传入＜named-config＞节点中 name 属性的值即可创建 C3P0 数据源对象。这种设置的好处是，当程序在后期更换数据源配置时，只需要修改构造方法中对应的 name 值即可。

2. 创建测试类

在 com. itheima. jdbc 包下创建一个名称为 C3P0Demo 的类，该类中使用 C3P0 数据源从配置文件中获取 Connection 对象，如文件 16-4 所示。

文件 16-4　C3P0Demo. java

```
package com.itheima.jdbc;
import java.sql.SQLException;
import javax.sql.DataSource;
import com.mchange.v2.c3p0.ComboPooledDataSource;
public class C3P0Demo {
    public static DataSource ds=null;
    //初始化 C3P0 数据源
    static {
        //使用 c3p0-config.xml 配置文件中的 named-config 节点中 name 属性的值
```

```
10          ComboPooledDataSource cpds=new ComboPooledDataSource("itheima");
11          ds=cpds;
12      }
13      public static void main(String[] args) throws SQLException {
14          System.out.println(ds.getConnection());
15      }
16  }
```

运行结果如图 16-37 所示。

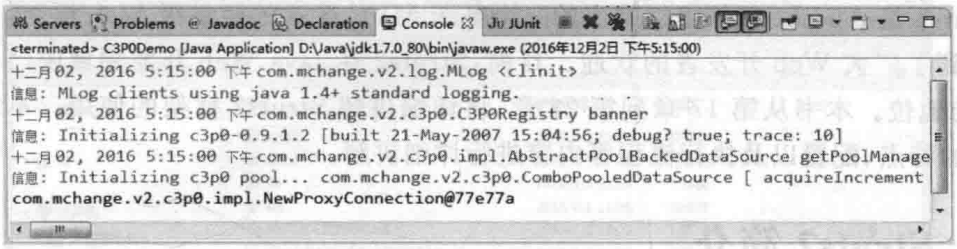


图 16-37　运行结果

从图 16-37 中可以看出，C3P0 数据源对象成功获取到了数据库连接对象。

需要注意的是，在使用 ComboPooledDataSource(String configName)方法创建对象时必须遵循以下两点。

- 配置文件名称必须为 c3p0-config. xml 或者 c3p0. properties，并且位于该项目的 src 根目录下。
- 当传入的 configName 值为空或者不存在时，则使用默认的配置方式创建数据源。

第 17 章 Struts2框架入门

Struts2 是 Apache 软件组织推出的一个基于 MVC 模式的轻量级 Web 框架，自问世以来，就受到了广大 Web 开发者的欢迎。目前，Struts2 在 Java Web 开发领域中已占据了十分重要的地位。本书从第 17 章到第 22 章，将详细讲解 Struts2 框架的使用。本章将针对 Struts2 的特点、配置以及执行流程等内容进行详细讲解。

17.1 Struts2 简介

17.1.1 Struts2 概述

目前为止，Struts 框架拥有两个主要版本：Struts1. x 和 Struts2. x。Struts1 是最早的基于 MVC 模式的轻量级 Web 框架，它能够合理划分代码结构，并包含验证框架、国际化框架等多种实用工具框架。随着技术的不断进步，Struts1 的局限性也越来越多地暴露出来。为了符合更加灵活、高效的开发需求，Struts2 框架应运而生，并在逐渐取代 Struts1 框架。

Struts2 是在 Struts1 和 WebWork 技术（WebWork 是由 OpenSymphony 组织开发的、致力于组件化和代码重用的 J2EE Web 框架，它也是一个 MVC 框架）的基础上进行合并后的全新框架。虽然 Struts2 的名字与 Struts1 相似，但其设计思想有很大不同，因为 Struts2 是以 WebWork 为核心的，它是 WebWork 技术与 Struts1 技术的结合，所以 Struts2 可以理解为 WebWork 的更新产品。

Struts2 拥有优良的设计和功能，其优势具体如下。

- 项目开源，使用及拓展方便。
- 通过简单、集中的配置调度业务类，使配置和修改都非常容易。
- 提供简单、统一的表达式语言访问所有可供访问的数据。
- 提供标准、强大的验证框架和国际化框架。
- 提供强大、可以有效减少页面代码的标签。
- 提供 Exception 处理机制，并且具有良好的 Ajax 支持。
- Result 方式的页面导航，通过 Result 标签很方便地实现重定向和页面跳转。
- 拥有智能的默认设置，不需要另外进行繁琐的设置。使用默认设置就可以完成大多数项目程序开发所需要的功能。

上面列举的就是 Struts2 框架的一系列技术优势，读者只需了解即可，在后面的学习中，读者会慢慢对这些技术优势有更深刻的理解和体会。

17.1.2　Struts2 的下载

Struts2 的官方下载网址为 http://struts.apache.org，在该网址中可以下载最新版本的 Struts2。由于本书使用的 Struts2 版本为 2.3.24，因此建议读者也下载该版本（可以通过地址 archive.apache.org/dist/struts/2.3.24/struts-2.3.24-all.zip 在 Struts2 的版本库中直接进行下载）。

下载成功后，将 struts-2.3.24-all.zip 压缩包进行解压，解压后的目录如图 17-1 所示。



图 17-1　Struts2 的文件夹结构

下面针对图 17-1 中所示的目录进行简单介绍，具体如下。

- apps：用于存放官方提供的 Struts2 示例程序，这些程序可以作为学习者的参考资料。各示例均为 war 文件，可以通过 zip 方式进行解压。
- docs：用于存放官方提供的 Struts2 文档，包括 Struts2 的快速入门、Struts2 的文档，以及 API 文档等内容。
- lib：用于存放 Struts2 的核心类库，以及 Struts2 的第三方插件类库。
- src：用于存放该版本 Struts2 框架对应的源代码。

17.1.3　Struts2 的使用

了解了 Struts2 的文件夹结构后，下面学习一下如何在项目中使用 Struts2 框架。将图 17-1 中的 lib 文件夹打开，可以看到 Struts2 开发中可能用到的所有 JAR 包（此版本有 107 个 JAR 包）。在一般的 Web 开发中，只需要将 lib 文件夹下所依赖的几个基础 JAR 包复制到 Web 项目的 WEB-INF/lib 路径下即可使用 Struts2 框架。

下面针对 Struts2 项目所依赖的基础 JAR 包进行介绍，如表 17-1 所示。

从表 17-1 中可以看出，此版本的 Struts2 项目所依赖的基础 JAR 包共 13 个。Struts2 根据版本的不同所依赖的基础 JAR 包可能不完全相同，不过基本变化不大，读者可以视情况而定。

表 17-1 Struts2 项目依赖的基础 JAR 包

文 件 名	说 明
asm-3.3.jar	操作 Java 字节码的类库
asm-commons-3.3.jar	提供了基于事件的表现形式
asm-tree-3.3.jar	提供了基于对象的表现形式
struts2-core-2.3.24.jar	Struts2 框架的核心类库
xwork-core-2.3.24.jar	WebWork 核心库，Struts2 的构建基础
ognl-3.0.6.jar	对象图导航语言(Object Graph Navigation Language)，Struts2 框架通过其读/写对象的属性
freemarker-2.3.22.jar	Struts2 标签模板使用的类库
javassist-3.11.0.GA.jar	JavaScript 字节码解释器
commons-fileupload-1.3.1.jar	Struts2 文件上传组件依赖包
commons-io-2.2.jar	Struts2 的输入/输出，传文件依赖的 JAR
commons-lang-2.4.jar	包含一些数据类型工具，是对 java.lang 包的增强
log4j-api-2.2.jar	Struts2 的日志管理组件依赖包的 API
log4j-core-2.2.jar	Struts2 的日志管理组件依赖包



多学一招：快速添加 Struts2 所需的 JAR 包

在图 17-1 中的 apps 目录下，找到 struts2-blank.war 文件，此文件是 Struts2 官方提供的一个空示例项目，里面包含了一些基础 JAR 包和一些 xml 配置文件。开发中为了方便快速地使用 Struts2 进行开发，读者可将此文件的扩展名改为 zip(即 struts2-blank.zip)，然后以压缩包方式解压打开。将 WEB-INF/lib 文件夹下所有的 JAR 文件复制添加到自己所建立的 Web 应用程序的 WEB-INF/lib 中，即完成了 Struts2 的添加工作，这些 JAR 文件刚好就是 Struts2 所依赖的基础包。

17.2 Struts2 入门

17.2.1 第一个 Struts2 程序

通过第 17.1 节的学习，读者已经了解了什么是 Struts2 框架，并掌握了在项目中添加 Struts2 框架的方法。为了帮助读者更快地掌握 Struts2 框架的使用，本节将使用 Struts2 框架实现一个 HelloWorld 程序。

1. 创建项目

在 Eclipse 中新建一个名为 chapter17 的 Web 项目，将 Struts2 框架依赖的基础 JAR 包复制到此项目的 WEB-INF/lib 路径中，然后选中所有 JAR 并右击，进行 Build Path→Add to Build Path 操作(此时所选 JAR 包会被添加到项目的 classpath 中)，添加后的项目结构如图 17-2 所示。

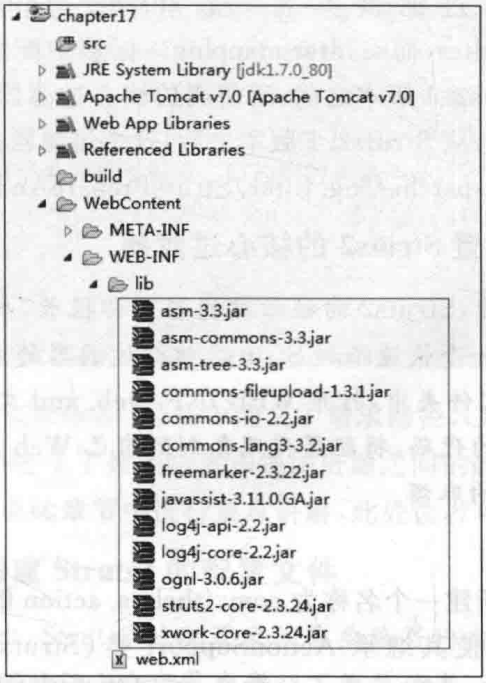


图 17-2　Struts2 的 JAR 文件

2. 配置 Struts2 的核心过滤器

打开项目的 web. xml 文件，在文件中添加 Struts2 的核心过滤器 org. apache. struts2. dispatcher. ng. filter. StrutsPrepareAndExecuteFilter，具体如文件 17-1 所示。

文件 17-1　web. xml

```
<? xml version="1.0" encoding="UTF-8"? >
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns="http://java.sun.com/xml/ns/javaee"
        xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                            http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    id="WebApp_ID" version="2.5">
    <!--配置 Struts2 核心过滤器-->
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
            org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

在 web. xml 文件中，<filter>标签中配置的信息就是 Struts2 的核心过滤器，该过

滤器的名称为 struts2，过滤器类为 org. apache. struts2. dispatcher. ng. filter. StrutsPrepareAndExecuteFilter，而＜filter-mapping＞标签中配置的是该过滤器的映射。

需要注意的是，在 Struts2. 1 版本之前，所使用的核心过滤器类是 org. apache. struts2. dispatcher. FilterDispatcher，从 Struts2. 1 版本之后，这个过滤器类已经不推荐使用了，而是使用 org. apache. struts2. dispatcher. ng. filter. StrutsPrepareAndExecuteFilter 类。



多学一招：快捷配置 Struts2 的核心过滤器

从文件 17-1 中可以看出，Struts2 的核心过滤器名称很长，如果手写的话，则很容易出现错误。为此，这里教读者一个快速添加 Struts2 核心过滤器的方法。在第 17.1 节多学一招中解压的 struts2-blank 文件夹中，打开 WEB-INF/web. xml 文件，即可找到文件 17-1 中 Struts2 的核心过滤器配置的代码，将配置代码复制到自己 Web 应用程序的 web. xml 文件中，就省去了自己手动编写的麻烦。

3. 创建 Action 类

在项目的 src 目录下，新建一个名称为 com. itheima. action 的包，在包中创建一个名为 HelloWorldAction 的类，并使其继承 ActionSupport 类（Struts2 框架所提供的类）。在 HelloWorldAction 类中定义一个 execute()方法，返回值为 SUCCESS，编辑后的代码如文件 17-2 所示。

文件 17-2　HelloWorldAction. java

```
1  package com.itheima.action;
2  import com.opensymphony.xwork2.ActionSupport;
3  public class HelloWorldAction extends ActionSupport {
4      public String execute() throws Exception{
5          return SUCCESS;
6      }
7  }
```

在文件 17-2 中，execute()方法的返回值 SUCCESS 是从父类中继承的常量字段，用于表示执行成功，并返回结果页面。execute()方法的返回值对应 struts. xml 文件中＜result＞标签中 name 属性的值，Struts2 框架会根据 name 属性值执行对应处理结果下的视图资源。

4. 编写 Struts2 的配置文件

在 src 目录下新建一个名称为 struts. xml 的文件，编辑代码后如文件 17-3 所示。

文件 17-3　struts. xml

```
1  <? xml version="1.0" encoding="UTF-8"? >
2  <!--指定 Struts2 配置文件的 DTD 信息-->
3  <!DOCTYPE struts PUBLIC
4      "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
5      "http://struts.apache.org/dtds/struts-2.3.dtd">
6  <!--Struts2 配置文件的根元素-->
7  <struts>
```

```
<!--Struts2 的 Action 必须放在指定的包空间下定义-->
<package name="hello" namespace="/" extends="struts-default">
  <!--定义 action,该 action 对应的类为 com.itheima.action.HelloWorldAction 类-->
  <action name="helloWorld" class="com.itheima.action.HelloWorldAction">
    <!--定义处理结果和视图资源之间的映射关系-->
    <result name="success">/success.jsp</result>
  </action>
 </package>
</struts>
```

在 struts.xml 文件中，<action>标签中定义了请求路径，以及与其对应的 Action 类的映射关系，子标签<result>定义了处理结果和视图资源之间的映射关系。关于 struts.xml 文件的详细配置信息，将在后续章节中进行重点讲解，此处读者可不必关心其细节问题。



多学一招：快速创建 Struts2 的配置文件

从文件 17-3 中可以看出，Struts2 的配置文件需要编写的内容有很多。尤其是文件上面的 dtd 信息，如果手动编写的话十分容易出错。下面教读者一个快速创建 struts.xml 的方法：在第 17.1 节多学一招中解压的 struts2-blank 文件夹中，将 WEB-INF/classes 目录下的 struts.xml 文件复制到自己所建立的 Web 项目的 src 目录下，然后在此基础上进行修改和配置即可。

5. 创建视图首页文件

在 WebContent 目录下创建一个名称为 index.jsp 的页面文件，在其中编写一个超链接，用于访问 Action 对象，此链接指向的地址为 helloWorld.action，如文件 17-4 所示。

文件 17-4　index.jsp

```
<%@page language="java" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>首页</title>
</head>
<body>
    <a href="${pageContext.request.contextPath }/helloWorld.action">
        第一个 Struts2 程序!
    </a>
</body>
</html>
```

6. 创建返回页面

在 WebContent 下再创建一个名称为 success.jsp 的文件，作为 Action 对象处理成功后

的返回页面，如文件 17-5 所示。

文件 17-5 success.jsp

```
<%@page language="java" import="java.util.*" pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>成功页面</title>
</head>
<body>
    您的第一个小程序执行成功，欢迎来到 Struts2 的世界！
</body>
</html>
```

7. 启动项目并查看结果

将项目 chapter17 发布到 Tomcat 服务器后启动服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter17/index.jsp 后，浏览器的显示结果如图 17-3 所示。

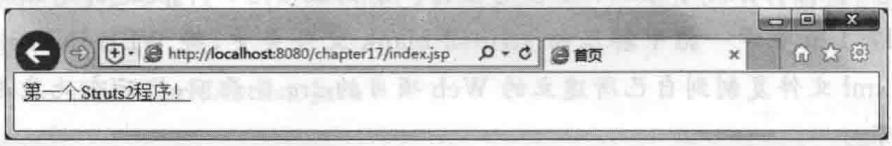


图 17-3 index.jsp

单击图 17-3 中的超链接后，浏览器会发送一个以 helloWorld.action 为结尾的请求，此时浏览器的显示结果如图 17-4 所示。



图 17-4 success.jsp

从图 17-4 的显示结果中可以看出，程序顺利返回到了 success.jsp 页面，这说明 Struts2 入门案例执行成功。

上面所讲解的 Struts2 入门程序，实际上就是一个请求发出到响应结束的过程，为了让读者更直观地了解该案例的执行全过程，下面通过简单的执行流程图进行说明，如图 17-5 所示。

从图 17-5 中可以看出，在客户端浏览器上单击 index.jsp 中的超链接时，会发送一个 helloWorld.action 请求，该请求被核心过滤器拦截后，会通过 struts.xml 文件中的配置找到请求对应的 HelloWorldAction，并默认调用 HelloWorldAction 中的 execute()方法返回逻辑视图名，然后再通过配置文件找到并转发给对应的视图页面 success.jsp 中，最后生成响应内容并输出响应的返回结果。

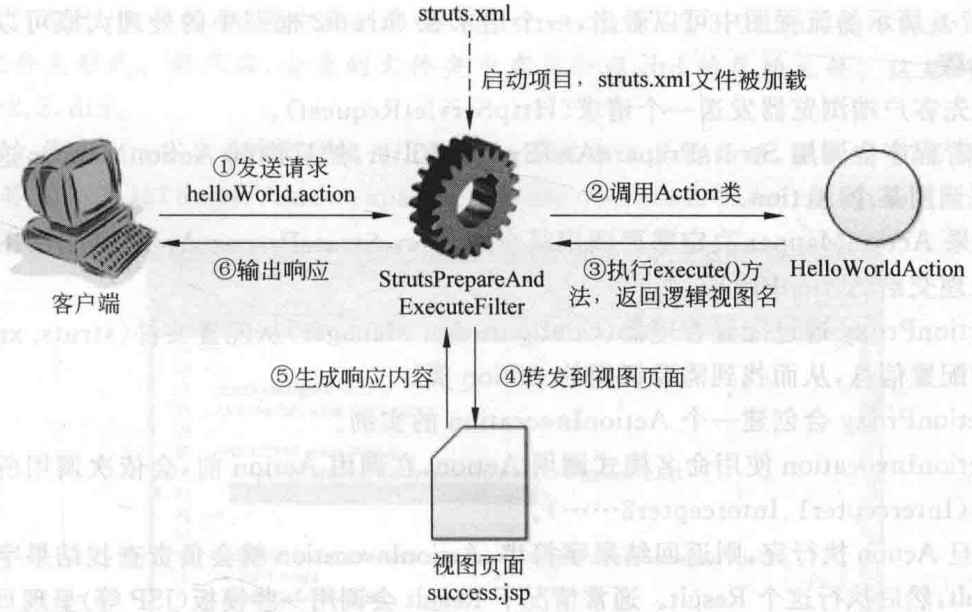


图 17-5　Struts2 入门案例执行流程

17.2.2　Struts2 的执行流程分析

通过第 17.2.1 节所讲解的 Struts2 的入门案例，相信读者对 Struts2 框架的使用已经有了一个初步的了解。为了帮助读者能够更好地学习 Struts2 框架，下面通过图例深入分析一下 Struts2 的基本执行流程，如图 17-6 所示。

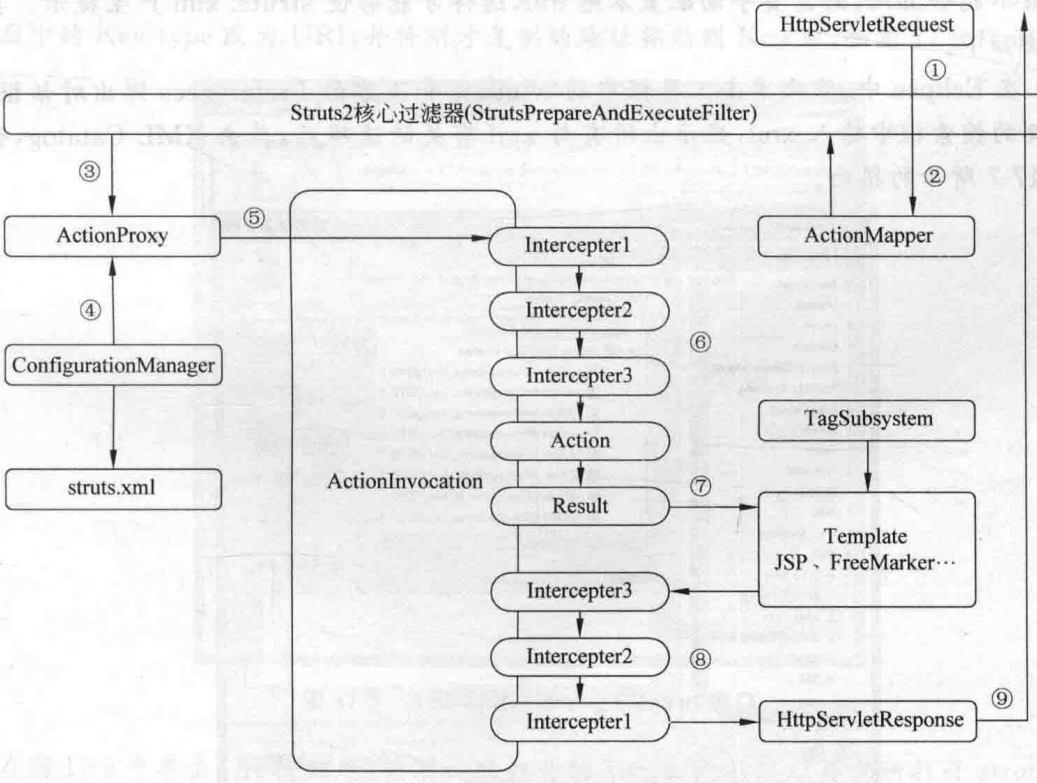


图 17-6　Struts2 的执行流程

从图 17-6 所示的流程图中可以看出，一个请求在 Struts2 框架中的处理大概可以分为以下几个步骤。

(1) 首先客户端浏览器发送一个请求(HttpServletRequest)。

(2) 接着程序会调用 StrutsPrepareAndExecuteFilter，然后询问 ActionMapper 这个请求是否需要调用某个 Action。

(3) 如果 ActionMapper 决定需要调用某个 Action，StrutsPrepareAndExecuteFilter 会把请求的处理交给 ActionProxy。

(4) ActionProxy 通过配置管理器(Configuration Manager)从配置文件(struts. xml)中读取框架的配置信息，从而找到需要调用的 Action 类。

(5) ActionProxy 会创建一个 ActionInvocation 的实例。

(6) ActionInvocation 使用命名模式调用 Action，在调用 Action 前，会依次调用所有配置的拦截器(Intercepter1、Intercepter2……)。

(7) 一旦 Action 执行完，则返回结果字符串，ActionInvocation 就会负责查找结果字符串对应的 Result，然后执行这个 Result。通常情况下 Result 会调用一些模板(JSP 等)呈现页面。

(8) 产生的 Result 信息返回给 ActionInvocation，在此过程中拦截器会被再次执行(顺序与 Action 执行之前相反)。

(9) 最后产生一个 HttpServletResponse 的响应行为，通过 StrutsPrepareAndExecuteFilter 反馈给客户端。

多学一招：使用 DTD 获得本地 struts. xml 提示

在开发过程中如果可以上网，struts. xml 会自动缓存 dtd，提供提示功能，方便代码编写。如果不能够上网，则需要手动配置本地 dtd，这样才能够使 struts. xml 产生提示。具体配置方法如下。

(1) 在 Eclipse 中，依次单击工具栏中的 window 和下方的 Preferences 弹出对话框，然后在左侧的搜索框中输入 xml，显示出所有与 xml 有关的选项后，单击 XML Catalog，会出现如图 17-7 所示的界面。

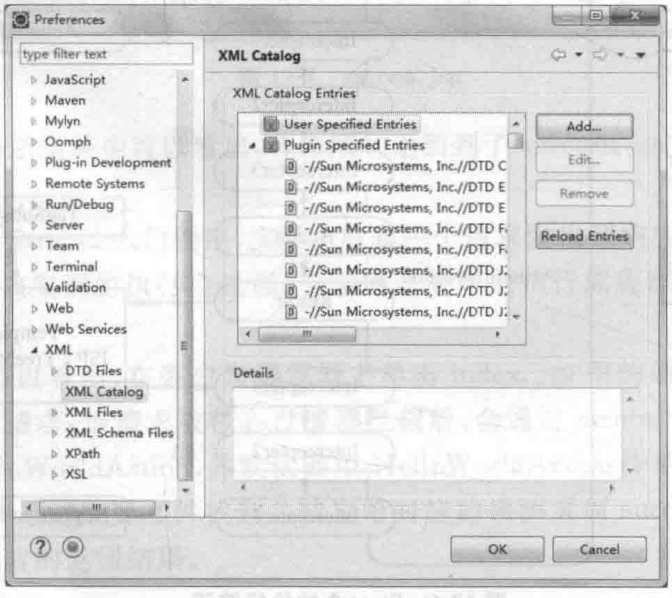


图 17-7 XML Catalog 窗口

(2) 在 Struts2 的 lib 包中找到其核心包 struts2-core-2.3.24.jar，使用解压工具将其解压成文件夹形式。解压后，会看到文件夹中有几个以 dtd 结尾的文件。这里所使用的是 struts-2.3.dtd。

(3) 将 struts-2.3.dtd 文件使用 EditPlus 等文本工具打开后，找到图 17-8 中选中的内容，将其 http 地址(http://struts.apache.org/dtds/struts-2.3.dtd)复制，如图 17-8 所示。

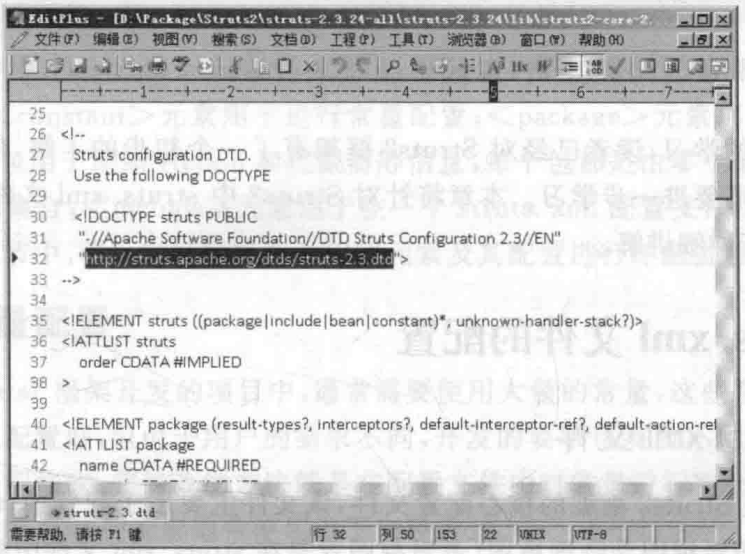


图 17-8　struts-2.3.dtd 文件

(4) 单击 Eclipse 中弹出对话框中右侧的 Add...按钮，此时会弹出 Add XML Catalog Element 界面。单击 File System 按钮，找到本地刚才解压文件夹中的 struts-2.3.dtd，然后将界面中的 Key type 改为 URI，并将刚才复制的地址粘贴到 Key 中，如图 17-9 所示。

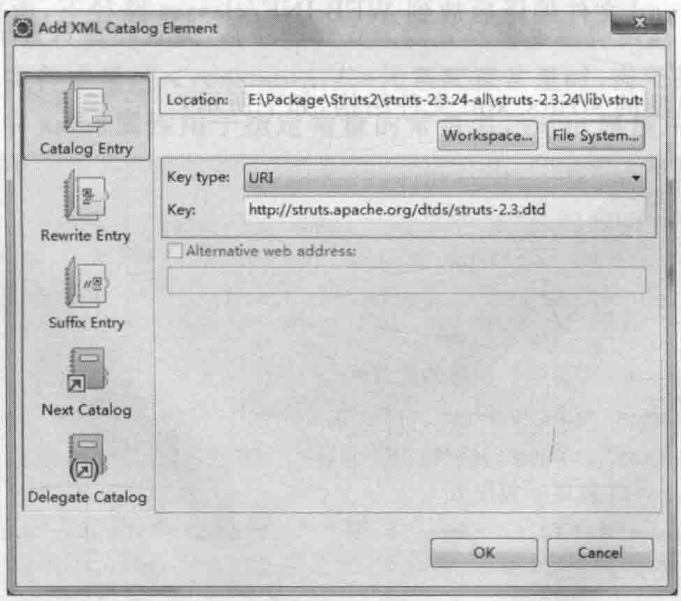


图 17-9　Add XML Catalog Element 窗口

在图 17-9 中单击 OK 按钮后，关闭已经打开的 struts.xml，然后再重新打开 struts.xml，此时再编写 struts.xml 内容时，就会有提示了。

第 18 章 Struts2的核心配置

通过第 17 章的学习，读者已经对 Struts2 框架有了一个初步的了解，但是对于各知识点的实现细节，还需要进一步学习。本章将针对 Struts2 中 struts. xml 文件的配置、Action 的配置等内容进行详细讲解。

18.1 struts. xml 文件的配置

18.1.1 struts. xml 文件

struts. xml 是 Struts2 框架的核心配置文件，该文件主要用于配置 Action 和请求的对应关系，以及配置逻辑视图和物理视图(逻辑视图就是在 struts. xml 文件中配置的＜result＞元素，它的 name 属性值就是逻辑视图名；物理视图是指＜result＞元素中配置的结果页面，如 JSP 文件)资源的对应关系。

struts. xml 文件通常放在 Web 应用的 WEB-INF/classes 目录下，在该目录下的 struts. xml 文件可以被 Struts2 框架自动加载。由于 Eclipse 开发工具在编译时会自动将 src 路径下的 struts. xml 文件编译后放到 WEB-INF/classes 路径下，所以直接将其放到项目的 src 路径下即可。

下面来看一个典型的 struts. xml 文件结构，如下所示：

```
<? xml version="1.0" encoding="UTF-8" ? >
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
"http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
    <!--<constant>元素用于常量的配置-->
    <constant name="struts.enable.DynamicMethodInvocation" value="false" />
    <constant name="struts.devMode" value="true" />
    <!--<package>元素用于包配置-->
    <package name="default" namespace="/" extends="struts-default">
        <!--配置 Action-->
        <action name="index" class="Xxx">
            <!--配置 Result-->
            <result type="dispatcher">
                <param name="location">/index.jsp</param>
```

```
            </result>
        </action>
    </package>
    <!--<include>元素用于包含配置-->
    <include file="example.xml"/>
</struts>
```

在上述文件中，＜struts＞元素是文件的根元素，所有其他元素都放在＜struts＞＜/struts＞中；＜constant＞元素用于进行常量配置；＜package＞元素用于进行包配置，在 Struts2 框架中，包用于组织 Action 和拦截器等信息，每个包都是由零个或多个拦截器以及 Action 所组成的集合；＜include＞元素用于在一个 struts. xml 配置文件中包含其他的配置文件。在下面几节中，将分别针对这些常用的元素及其配置进行详细讲解。

18.1.2　常量配置

在使用 Struts2 框架开发的项目中，通常需要使用大量的常量，这些常量大多在默认的配置文件中已经配置好，但由于用户的需求不同，开发的要求也有所不同，这时可能就需要对这些常量值进行修改，其修改的方法就是在配置文件中对常量进行重新配置。

在 Struts2 中，常量的配置通常使用以下三种方式。

(1) 在 struts. xml 文件中使用＜constant＞元素配置常量(常用方式)。

(2) 在 struts. properties 文件中配置常量。

(3) 在 web. xml 文件中通过＜init-param＞元素配置常量。

上述三种配置方式中，在 struts. xml 文件中通过＜constant＞元素配置常量，是项目开发时最常用的方式，本书中的常量配置也是通过此种方式实现的，所以这里只介绍通过＜constant＞元素配置常量的方式，其他两种方式读者只需了解即可。

在 struts. xml 文件中通过＜constant.../＞元素配置常量时，需要指定两个必填的属性 name 和 value。其中 name 属性用于指定常量的常量名，value 属性用于指定常量的常量值。其配置的示例代码如下所示：

```
<struts>
    <!--设置默认编码集为 UTF-8-->
    <constant name="struts.i18n.encoding" value="UTF-8" />
    <!--设置使用开发模式-->
    <constant name="struts.devMode" value="true" />
</struts>
```

在上述示例代码中，配置了常量 struts. i18n. encoding 和 struts. devMode。其中 struts. i18n. encoding 用于指定 Struts2 应用程序的默认编码集为 UTF-8，struts. devMode 用于指定项目使用的是开发模式。

Struts2 所支持的常量数量很多，在 struts2-core-2. 3. 24. jar 压缩文件的 org/apache/struts2 路径下有一个 default. properties 文件，该文件里为 Struts2 的所有常量都指定了默认值，读者可以通过查看该文件了解 Struts2 所支持的常量。

除了可以通过上述的三种方式配置常量以外，在 Struts2 内置的一些配置文件中也有对常量的配置。因此，如果在多个文件中配置了同一个 Struts2 常量，则 Struts2 框架加载常量是有一定顺序的，通常的搜索顺序如下。

(1) default. properties：该文件保存在 struts2-core-2. 3. 24. jar 中的 org. apache. struts2 包中。

(2) struts-default. xml：该文件保存在 struts2-core-2. 3. 24. jar 文件中。

(3) struts-plugin. xml：该文件保存在 struts-Xxx-2. 3. 24. jar 等 Struts2 插件 JAR 包中。

(4) struts. xml：该文件是 Web 应用自身的 Struts2 配置文件。

(5) struts. properties：该文件是 Web 应用默认的 Struts2 配置文件。

(6) web. xml：该文件是 Web 应用的配置文件。

上面指定了 Struts2 框架搜索 Struts2 常量的顺序，需要注意的是，如果在多个文件中配置了同一个 Struts2 常量，则后一个文件中配置的常量值会覆盖前面文件中配置的常量值。

18.1.3 包配置

在 Struts2 框架中，包用于管理 Action 和拦截器，每个包就是多个 Action、多个拦截器、多个拦截器引用的集合。在 struts. xml 文件中，使用＜package＞元素定义包配置，每个＜package＞元素都定义了一个包配置，其示例代码如下所示：

```
<package name="default" namespace="/" extends="struts-default">
    ...
</package>
```

在上述代码中，＜package＞元素用到了三个关键属性 name、namespace、extends，具体说明如下。

- name：必填属性，用于指定该包的名称(该名称在配置文件中必须是唯一的)，此名称是该包被其他包引用的 Key。
- namespace：可选属性，用于定义该包的命名空间。
- extends：可选属性，用于指定该包继承自其他包。其属性值必须是另一个包的 name 属性值，但该属性值通常都设置为 struts-default，这样该包中的 Action 就具有了 Struts2 框架默认的拦截器等功能。

18.1.4 包含配置

Struts2 允许将一个配置文件分解成多个配置文件，从而提高配置文件的可读性，＜include＞元素用于在一个 struts. xml 配置文件中包含其他的配置文件。Struts2 默认只加载 WEB-INF/classes 下的 struts. xml 文件，一旦通过多个 XML 文件配置 Action，就必须通过＜include＞元素包含其他配置文件。

在 struts. xml 文件中包含配置的示例代码如下所示：

```
<struts>
    <!--不指定路径,默认在 src 下时的方式-->
    <include file="struts-post.xml"/>
    <include file="struts-user.xml"/>
    <include file="struts-dept.xml"/>
    <!--配置文件在具体包中时的方式-->
    <include file="com/itheima/action/struts-product.xml">
</struts>
```

在上述代码中,struts. xml 文件通过<include>元素包含了四个配置文件,其中 file 属性用于指定被包含配置文件的名称。如果被包含的配置文件在 src 路径下,则直接指定文件名即可,如果被包含的配置文件在具体的包中,则需要引入被包含文件的包路径。

需要注意的是,每一个被包含的配置文件都是标准的 Struts2 配置文件,同样包含 DTD 信息、Struts2 配置文件的根元素等信息。由于 struts. xml 文件包含了其他的配置文件,所以在 Struts2 框架自动加载 struts. xml 文件时,就会完成所有配置信息的加载。

18.2　Action 的配置

18.2.1　实现 Action 控制类

在 Struts2 框架中,Action 是框架的核心类,被称为业务逻辑控制器,主要用于实现对用户请求的处理。一个 Action 类代表一次请求或调用,每个请求的动作都对应一个相应的 Action 类。也就是说,用户的每次请求,都会转到一个相应的 Action 类中,由这个 Action 类进行处理。简而言之,Action 就是用于处理一次用户请求的对象。

实现 Action 控制类通常采用两种方式,分别是实现 Action 接口和继承 ActionSupport 类。接下来分别对它们进行讲解,具体如下。

1. 实现 Action 接口

当 Action 类处理用户请求成功后,有人习惯返回 index 字符串,有人习惯返回 success 字符串,这会导致在一个 Action 中可能会返回各种不同的值,十分不利于项目的统一管理。为了让用户更规范地创建 Action 类,Struts2 提供了一个 Action 接口,该接口定义了 Action 类应该实现的规范,用户在创建 Action 时,可以实现这个接口。Action 接口中的具体代码如下所示:

```
package com.opensymphony.xwork2;
public interface Action {
    //定义 Action 接口中包含的一些结果字符串
    public static final String SUCCESS="success";
    public static final String NONE="none";
    public static final String ERROR="error";
    public static final String INPUT="input";
    public static final String LOGIN="login";
```

```
    //定义处理用户请求的 execute()方法
    public String execute() throws Exception;
}
```

从上述代码中可以看出，Action 接口位于 com. opensymphony. xwork2 包中，并且接口中只定义了五个字符串常量和一个 execute()方法。其中，execute()方法是 Action 类的默认请求处理方法，该方法返回一个字符串，而上面五个字符串常量的作用是统一 execute()方法的返回值。

2. 继承 ActionSupport 类

由于 Xwork 的 Action 接口十分简单，为开发者提供的帮助较小，所以在实际开发过程中，通常都是采用继承 ActionSupport 类的方式创建 Action。其示例代码如下所示：

```
import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport{
private static final long serialVersionUID=1L;
    @Override
    public String execute() throws Exception {
        return super.execute();
    }
}
```

ActionSupport 是 Action 接口的默认实现类，所以继承 ActionSupport 就相当于实现了 Action 接口。除 Action 接口以外，ActionSupport 类还实现了 Validateable、ValidationAware、TextProvider、LocaleProvider 和 Serializable 等接口，这为用户提供了更多的功能。

ActionSupport 类中提供了许多默认方法，这些默认方法包括数据校验的方法、默认的处理用户请求的方法等。如果开发者的 Action 类继承 ActionSupport 类，会大大简化 Action 的开发。

需要注意的是，由于自定义的 Action 类继承了 ActionSupport 类，因此必须定义一个变量 serialVersionUID。这是因为 ActionSupport 类实现了 Serializable 接口，任何实现了 Serializable 接口的类都必须声明变量 serialVersionUID，如下所示：

```
private static final long serialVersionUID=1L;
```

在学习过程中，细心的读者可能会发现，即使不加上述代码，程序也可以正常执行。但是在实际项目开发中，必须加上上述代码。

18.2.2 配置 Action

配置 Action 主要就是配置 struts. xml 文件中 Action 的映射信息。Action 映射是指将一个请求的 URL 映射到一个 Action 类，当一个请求匹配某个 Action 名称时，Struts2 框架就使用这个 Action 确定如何处理请求。

struts. xml 文件是通过<action>元素对请求的 Action 和 Action 类进行配置的，其示例代码如下所示：

```
<action name="userAction" class="com.itheima.action.UserAction" method="add">
    ...
</action>
```

在上述代码中，包含了<action>元素的三个常用属性 name、class 和 method，这三个属性的具体说明如下。

- name：必填属性，表示 Action 的名称(该名称必须唯一)，它指定了 Action 所处理请求的 URL。该属性将在其他地方被引用，如作为 JSP 页面 form 表单的 action 属性值。
- class：可选属性，用于指定 Action 的实现类，如果没有指定 class 属性值，则其默认值为 com. opensymphony. xwork2. ActionSupport 类。
- method：可选属性，指定请求 Action 时调用的方法。如果指定了 method 属性，则该 Action 会调用 method 属性中指定的方法，如果不指定 method 属性，则 Action 会调用 execute()方法。

18.2.3　使用通配符

由于在一个 Action 类中可能有多个业务逻辑处理方法，在配置 Action 时，就需要使用多个<action>元素。在实现同样功能的情况下，为了减少 struts. xml 配置文件的代码量，可以借助于通配符映射信息。

下面以一段 Action 的配置代码为例，说明如何使用通配符进行配置，如下所示：

```
<package name="user" namespace="/user" extends="struts-default">
    <action name="userAction_*" class="com.itheima.action.UserAction" method="{1}">
        <result>/index.jsp</result>
    </action>
</package>
```

在上述代码中，method 属性值中的数字 1 表示匹配第 1 个 * 。当客户端发送/user/userAction_login. action 这样的请求时，<action>元素的 name 属性值就被设置成 userAction_login，method 属性值就被设置成 login。当客户端发送/user/userAction_register. action 这样的请求时，<action>元素的 name 属性值就被设置为 userAction_register，method 属性值也被设置成 register。

另外，对<result>元素也可以采用通配符配置，代码如下所示：

```
<result>/{1}.jsp</result>
```

当客户端发送 userAction_login 这样的请求时，<result>元素被设置成跳转到 login. jsp 页面。当客户端发送 userAction_ register 这样的请求时，<result>元素被设置成跳转到

register.jsp 页面。

18.3 Action 访问 Servlet API

在 Struts2 中，虽然 Action 已经与 Servlet API 完全分离，但在实现业务逻辑时，还是经常要访问 Servlet API 中的对象。通常开发时需要访问 Servlet API 中的 HttpServletRequest、HttpSession 和 ServletContext 三个接口，它们分别对应 JSP 内置对象 request、session 和 application。在 Struts2 中，访问 Servlet API 通常采用两种方式，分别是通过 ActionContext 访问和通过 ServletActionContext 访问，本节将针对这两种访问方式进行讲解。

18.3.1 通过 ActionContext 访问

ActionContext 是 Action 执行的上下文对象，在 ActionContext 中保存了 Action 执行所需要的所有对象，包括 request、session 和 application 等。ActionContext 类访问 Servlet API 的几个常用方法如表 18-1 所示。

表 18-1 ActionContext 类访问 Servlet API 的常用方法

方法声明	功能描述
void put(String key, Object value)	将 key-value 键值对放入 ActionContext 中，模拟 Servlet API 中的 HttpServletRequest 的 setAttribute()方法
Object get(String key)	通过参数 key 查找当前 ActionContext 中的值
Map<String,Object> getApplication()	返回一个 Application 级的 Map 对象
static ActionContext getContext()	获取当前线程的 ActionContext 对象
Map<String,Object> getParameters()	返回一个包含所有 HttpServletRequest 参数信息的 Map 对象
Map<String,Object> getSession()	返回一个 Map 类型的 HttpSession 对象

要访问 Servlet API，可以通过如下示例代码方式进行：

```
ActionContext context=ActionContext.getContext();
context.put("name", "itheima");
context.getApplication().put("name", "itheima");
context.getSession().put("name", "itheima");
```

在上述示例代码中，通过 ActionContext 类中的方法调用，分别在 request、application 和 session 中放入了("name", "itheima")键值对。通过代码可以看到，ActionContext 类可以非常简单地访问 JSP 内置对象的属性。

为了让读者更好地掌握如何通过 ActionContext 类访问 Servlet API，下面通过具体的案例演示 ActionContext 的使用，具体步骤如下。

1. 创建项目

在 Eclipse 中创建一个名称为 chapter18 的 Web 项目，将 Struts2 所需的 JAR 包复制到项目的 lib 目录中，并发布到类路径下。

2. 创建页面文件

在 WebContent 目录下分别创建登录页面 login.jsp、登录成功页面 success.jsp 和错误显示页面 error.jsp，如文件 18-1、文件 18-2 和文件 18-3 所示。

文件 18-1　login.jsp

```
<%@page language="java" import="java.util.*" pageEncoding="utf-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
<title>登录页面</title>
<style type="text/css">
    input[type=text], input[type=password] {width: 150px;}
</style>
</head>
<body>
    <div align="center">
        <form action="login" method="post">
            用户名:<input type="text" name="username" /><br />
            密     码:<input type="password" name="password"/>
            <br/>
            <input type="reset" value="重置" />
            <input type="submit" value="登录" />
        </form>
    </div>
</body>
</html>
```

文件 18-2　success.jsp

```
<%@page language="java" import="java.util.*" pageEncoding="utf-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>登录成功页面</title>
</head>
<body>
    <p>${success }<br /></p>
    <h2>用户登录信息</h2>
    用户名:${username }<br />
    密 码:${password }<br />
</body>
</html>
```

文件 18-3 error.jsp

```
<%@page language="java" import="java.util.*" pageEncoding="utf-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
<head>
    <title>登录失败页面</title>
</head>
<body>
    <p>${error }<br /></p>
</body>
</html>
```

3. 添加过滤器

在 web.xml 文件中配置 Struts2 的核心过滤器，配置后的代码如文件 18-4 所示。

文件 18-4 web.xml

```
<? xml version="1.0" encoding="UTF-8"? >
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
                       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                       xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
                       http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
    <filter>
        <filter-name>struts2</filter-name>
        <filter-class>
        org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>struts2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

4. 创建 Action

在 src 目录下创建一个名称为 com.itheima.action 的包，在该包中创建一个名称为 LoginAction 的类，该类主要用于业务逻辑处理，其代码如文件 18-5 所示。

文件 18-5 LoginAction.java

```
package com.itheima.action;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class LoginAction extends ActionSupport {
    private String username;          //用户名
```

```
    private String password;          //密码
    //username 的 getter 和 setter 方法
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username=username;
    }
    //password 的 getter 和 setter 方法
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password=password;
    }
    @Override
    public String execute() throws Exception {
        //获取 ActionContext 对象
        ActionContext context=ActionContext.getContext();
        if ("admin".equals(username) && "123456".equals(password)) {
            //将用户名和密码信息放入 context 对象中
            context.put("username", username);
            context.put("password", password);
            context.put("success", "用户登录成功!");
            return SUCCESS;
        } else {
            //定义登录失败的错误信息
            context.put("error", "用户名或密码错误,请重新登录!");
            return ERROR;
        }
    }
}
```

5. 创建配置文件

在 src 目录下创建 struts. xml 文件，编辑后如文件 18-6 所示。

文件 18-6　struts. xml

```
<? xml version="1.0" encoding="UTF-8" ? >
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
    <package name="default" extends="struts-default">
        <action name="login" class="com.itheima.action.LoginAction">
```

```
8            <result name="success">/success.jsp</result>
9            <result name="error">/error.jsp</result>
10         </action>
11      </package>
12  </struts>
```

6. 运行程序并查看结果

启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter18/login.jsp 访 login.jsp 页面问，浏览器的显示结果如图 18-1 所示。

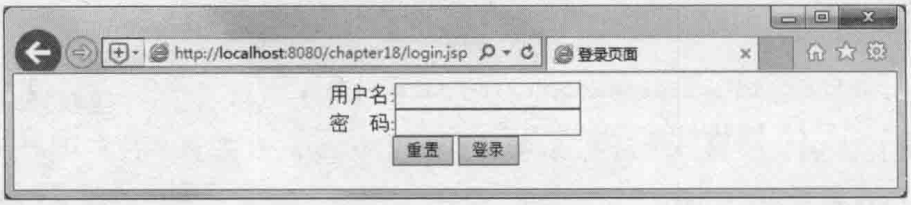


图 18-1 登录页面

在图 18-1 中输入用户名“admin”和错误的密码“123”后，单击【登录】按钮，结果页面如图 18-2 所示。

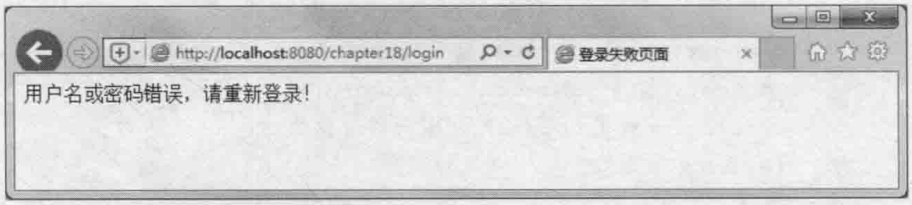


图 18-2 登录失败页面

在图 18-1 中输入用户名“admin”和正确的密码“123456”后，单击【登录】按钮，结果页面如图 18-3 所示。

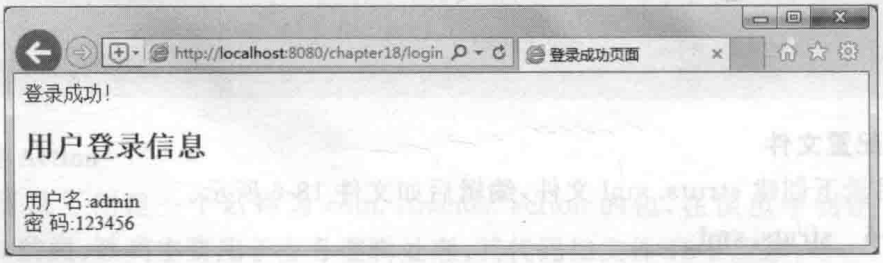


图 18-3 登录成功页面

从图 18-3 中可以看出，在 Action 中放入 ActionContext 的 key-value 键值对都被取出来了，说明通过 ActionContext 类能够访问 Servlet API。

18.3.2 通过 ServletActionContext 访问

除了通过 ActionContext 类访问 Servlet API 以外，Struts2 框架还提供了 ServletActionContext

类访问 Servlet API，该类中的方法都是静态方法，其常见方法如表 18-2 所示。

表 18-2　ServletActionContext 类访问 Servlet API 的常用方法

方法声明	功能描述
static PageContext getPageContext()	获取 Web 应用的 PageContext 对象
static HttpServletRequest getRequest()	获取 Web 应用的 HttpServletRequest 对象
static HttpServletResponse getResponse()	获取 Web 应用的 HttpServletResponse 对象
static ServletContext getServletContext()	获取 Web 应用的 ServletContext 对象

下面通过案例讲解如何通过 ServletActionContext 访问 Servlet API。

1. 创建 MessageAction 类

在 chapter18 项目中的 com.itheima.action 包下创建 MessageAction 类，编写后的代码如文件 18-7 所示。

文件 18-7　MessageAction.java

```
1  package com.itheima.action;
2  import org.apache.struts2.ServletActionContext;
3  import com.opensymphony.xwork2.ActionSupport;
4  public class MessageAction extends ActionSupport {
5      public String execute() throws Exception {
6          ServletActionContext.getRequest().setAttribute("message",
7                  "通过 ServletActionContext 类访问 Servlet API");
8          return SUCCESS;
9      }
10 }
```

2. 配置 struts.xml

在 package 元素中添加一个名称为 message 的 action 配置信息，添加代码如下所示：

```
<action name="message" class="com.itheima.action.MessageAction">
    <result name="success">/message.jsp</result>
</action>
```

3. 创建 message.jsp 页面

在 WebContent 目录下创建一个名称为 message.jsp 的页面，通过 EL 表达式访问存放在 Request 对象中键值为 message 的值，其页面主体部分代码如下所示：

```
<div align="center">${requestScope.message}</div>
```

4. 运行程序并查看结果

启动 chapter18 项目，在浏览器的地址栏中输入地址 http://localhost:8080/chapter18/message 后，浏览器的显示结果如图 18-4 所示。

从图 18-4 中可以看出，使用 ServletActionContext 类也可以在 Action 中访问 Servlet

API。虽然如此，该 Action 依然与 Servlet API 直接耦合，这不利于程序的解耦。因此，建议在开发中优先使用 ActionContext，以避免和 Servlet API 耦合。

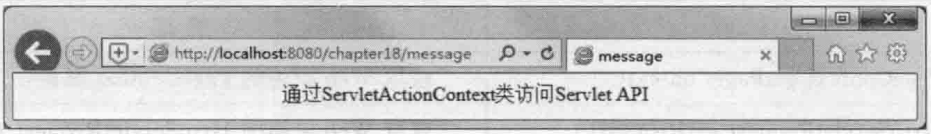


图 18-4 登录页面

18.4 Action 处理请求参数

在 Struts2 框架中，页面的请求数据和 Action 有两种基本的对应方式，分别是字段驱动(FieldDriven，也称为属性驱动)方式和模型驱动(ModelDriver)方式。本节将针对这两种 Action 处理请求参数的方式进行详细讲解。

18.4.1 属性驱动

属性驱动是指在 Action 中通过字段属性进行与页面之间的数据传递，通常使用时会包括两种情况：一种是与基本数据类型的属性对应，另一种是直接使用域对象。

1. 基本数据类型字段驱动方式的数据传递

在 Struts2 中，可以直接在 Action 中定义各种 Java 基本数据类型的字段，使这些字段与表单数据相对应，并利用这些字段进行数据传递，如下面的代码所示：

```
public class UserAction extends ActionSupport {
    private String username;                    //用户名
    private String password;                    //密码
    //此处省略两个属性的 getter 和 setter 方法
    private String execute() throws Exception{
        return SUCCESS;
    }
}
```

在上述 Action 类的代码中，定义了两个字符串字段 username 和 password，这两个字段分别用于对应页面上的用户名和密码这两个表单域。

2. 直接使用域对象字段驱动方式的数据传递

在基本数据类型字段驱动方式中，如果传入的数据很多，那么 Action 的属性也会变得很多，再加上属性对应的 getter/setter 方法，势必会导致 Action 非常臃肿。为了解决这一问题，我们可以把属性的 getter/setter 方法从 Action 中提取出来单独作为一个域对象，并在相应的 Action 中直接使用这个域对象。此种方式中的域对象一般以 JavaBean 的形式实现，JavaBean 中所封装的属性要和页面中表单的属性一一对应。此时 JavaBean 将成为数据传递的载体，并可以在其他 Action 中使用。下面通过具体的案例演示域对象字段的驱动方式的使用。

(1) 在 chapter18 项目的 src 目录下创建一个名称为 com. itheima. domain 的包，在包中创建一个 User 类，编写代码后如文件 18-8 所示。

文件 18-8　User. java

```
1  package com.itheima.domain;
2  public class User {
3      private String username;
4      private String password;
5      //username 的 getter 和 setter 方法
6      public String getUsername() {
7          return username;
8      }
9      public void setUsername(String username) {
10         this.username=username;
11     }
12     //password 的 getter 和 setter 方法
13     public String getPassword() {
14         return password;
15     }
16     public void setPassword(String password) {
17         this.password=password;
18     }
19 }
```

(2) 在 com. itheima. action 包中创建一个名称为 UserLoginAction 的类，在类中定义一个 User 类型的域模型以及其 getter 和 setter 方法，如文件 18-9 所示。

文件 18-9　UserLoginAction. java

```
1  package com.itheima.action;
2  import com.itheima.domain.User;
3  import com.opensymphony.xwork2.ActionContext;
4  import com.opensymphony.xwork2.ActionSupport;
5  public class UserLoginAction extends ActionSupport {
6      private User user;              //定义 User 属性
7      //user 属性的 getter 和 setter 方法
8      public User getUser() {
9          return user;
10     }
11     public void setUser(User user) {
12         this.user=user;
13     }
14     public String execute() throws Exception {
15         //获取 Context 对象
16         ActionContext context=ActionContext.getContext();
17         if ("admin".equals(user.getUsername())
```

```
                && "123456".equals(user.getPassword())) {
            //将用户名和密码放入 session 中
        context.getSession().put("username", user.getUsername());
        context.getSession().put("password", user.getPassword());
            return SUCCESS;
        } else {
            context.getSession().put("error", "用户名或密码错误!");
            return ERROR;
        }
    }
}
```

在文件 18-9 中，首先声明了一个 User 类型的 user 对象，以及其 getter 和 setter 方法，然后在 execute()方法中，获取了 ActionContext 对象以访问 Servlet API，下面通过对比从 User 对象获取的用户名和密码来判断是否登录成功。

需要注意的是，在使用域对象的属性驱动方式传值时，如果 JSP 页面是负责取值的，那么取值的格式必须为“对象名.属性名”；如果 JSP 页面是负责传值的，那么传值的格式可以为“模型对象名.属性名”，也可以直接是“属性名”。

(3) 分别创建登录页面 userLogin.jsp、登录成功页面 loginSuccess.jsp 和登录失败页面 loginError.jsp，如文件 18-10、文件 18-11 和文件 18-12 所示。

文件 18-10　userLogin.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<title>用户登录页面</title>
<style type="text/css">
    input[type=text], input[type=password] {width: 150px;}
</style>
</head>
<body>
    <div align="center">
        <form name="form1" action="userlogin.action" method="post">
            用户名: <input type="text" name="user.username"><br />
密     码: <input type="password" name="user.password"><br />
            <input type="reset" value="重置"/>
            <input type="submit" value="登录">
        </form>
    </div>
</body>
</html>
```

文件 18-11　loginSuccess.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<title>用户登录成功页面</title>
</head>
<body>
    <p align="center">
        您的用户名是<%=request.getAttribute("user.username")%>
        <br />
        密码是<%=request.getAttribute("user.password")%>
    </p>
</body>
</html>
```

文件 18-12　loginError.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<html>
<head>
<title>错误页面</title>
</head>
<body>
    <p align="center">
        ${error}<br />
        <a href="userLogin.jsp">请单击此链接返回登录页面重新登录</a>
    </p>
</body>
</html>
```

(4) 在配置文件 struts.xml 中的<package>元素中添加 UserLoginAction 的信息，添加内容如下所示：

```
<action name="userlogin" class="com.itheima.action.UserLoginAction">
    <result name="success">/loginSuccess.jsp</result>
    <result name="error">/loginError.jsp</result>
</action>
```

(5) 部署 chapter18 项目到 Tomcat 服务器并启动服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter18/userLogin.jsp 访问 userLogin.jsp，浏览器的显示结果如图 18-5 所示。

在图 18-5 中输入正确的用户名“admin”和密码“123456”，单击【登录】按钮后，浏览器的

显示结果如图 18-6 所示；如果用户名或密码错误，则进入错误页面，如图 18-7 所示。

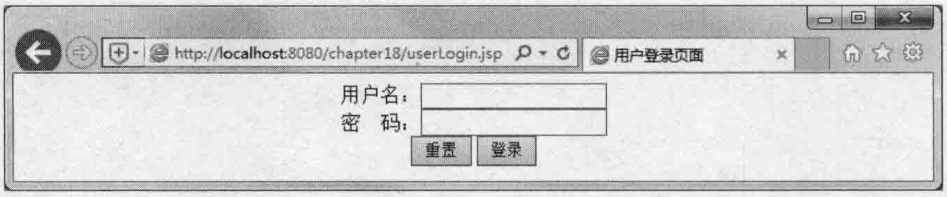


图 18-5 登录页面

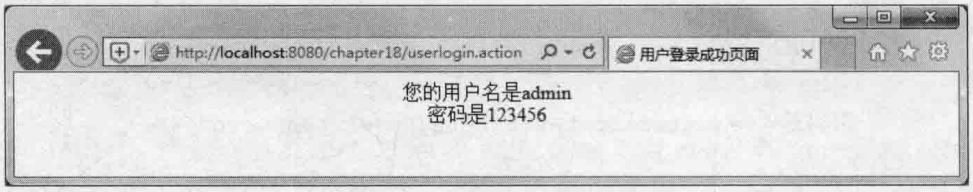


图 18-6 登录成功页面

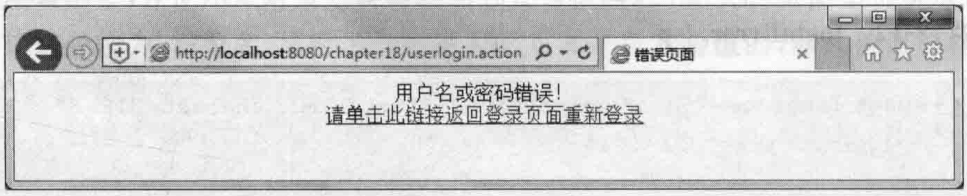


图 18-7 错误页面

18.4.2 模型驱动

在 Struts2 中，Action 还有另外一种方式处理请求参数，称为模型驱动(ModelDriven)。模型驱动方式要求 Action 需要通过实现 ModelDriven 接口接收请求参数，并且要重写 getModel()方法。getModel()方法返回的就是 Action 所使用的数据模型对象。

与属性驱动中直接使用域对象字段驱动方式的数据传递类似，模型驱动方式也是通过 JavaBean 模型进行数据传递的。只要是普通的 JavaBean，就可以充当模型部分，并且 JavaBean 中所封装的属性要与表单的属性一一对应，JavaBean 就是数据传递的载体。使用模型驱动方式时，Action 类中通过 getModel()方法获取模型，其示例代码如下所示：

```
public class LoginAction extends ActionSupport implements ModelDriven<User>{
    private User user=new User();
    public User getModel() {
        return user;
    }
    public String execute() throws Exception {
        return "success";
    }
}
```

使用模型驱动时，其对应的表单页面也要做相应调整，调整后的代码片段如下所示：

```
<form name="form1" action="loginAction" method="post">
    用户名：<input type="text" name="username"><br />
    密码：<input type="password" name="password"><br />
    <input type="submit" value="登录">
</form>
```

从上述代码中可以看出，使用 ModelDriver 的方式后，表单中的文本域名称已经不需要添加 user 前缀，页面上的 username 会自动对应到这个 Model 的 username 属性。

与属性驱动相比，模型驱动不需要在 Action 类中定义与表单元素一一对应的所有属性及其各属性的 getter 和 setter 方法，这减少了 Action 类中的代码量。在项目应用中具体使用哪种驱动方法，现给出以下几点建议。

(1) 要统一整个系统中 Action 的驱动方法，即要么都使用属性驱动，要么都使用模型驱动。

(2) 如果持久层对象与表单中的属性是一一对应的关系，那么建议使用模型驱动，因为模型驱动方法使 Action 类中的代码更加整洁。

(3) 如果持久层对象与表单中的属性不是一一对应的关系，那么建议使用属性驱动，因为不是一一对应的关系时，系统中需要提供两个 JavaBean(一个对应表单提交的数据，一个用于持久层对象)。

总之，属性驱动的方法和模型驱动的方法各有优缺点，在实际开发中，需要根据项目实际情况选择使用哪种驱动方式。

18.5　Result 结果类型

18.5.1　配置 Result

在 struts.xml 文件中，<result>元素用于配置 Result 逻辑视图与物理视图之间的映射关系，它有两个可选属性 name 和 type。其中，name 属性用于指定逻辑视图的名称，默认值为 success；type 属性用于指定返回的视图资源的类型，不同的类型代表不同的结果输出，它的默认值是 dispatcher。

struts.xml 文件中的<result>元素配置示例代码如下所示：

```
<action name="loginAction" class="com.itheima.action.LoginAction">
    <result name="success" type="dispatcher">
        <param name="location">/success.jsp</param>
    </result>
</action>
```

上述代码为 Action 配置一个了 name 为 success 的 Result 映射，该映射的值可以是 JSP 页面，也可以是一个 Action 的 name 值；这里使用 param 子元素为其指定了 Result 映射对应的物理视图资源为 success.jsp。

<param>子元素的 name 属性有两个值，分别如下。

- location：指定该逻辑视图所对应的实际视图资源。
- parse：指定在逻辑视图资源名称中是否可以使用 OGNL(对象图导航语言)表达式。默认值为 true，表示可以使用，如果设为 false，则表示不支持。

其实，上述配置代码还可以简化为如下形式：

```
<action name="loginAction" class="com.itheima.action.LoginAction">
    <result>/success.jsp</result>
</action>
```

需要注意的是，在 Result 配置中指定实际资源位置时，可以使用绝对路径，也可以使用相对路径。绝对路径以斜杠(/)开头，例如＜result＞/success. jsp＜/result＞，相当于当前 Web 应用程序的上下文路径；相对路径不以斜杠(/)开头，例如＜result＞success. jsp＜/result＞，相当于当前执行的 Action 路径。

18.5.2 预定义的结果类型

在使用 Struts2 框架编写项目时，当框架调用 Action 对请求进行处理后，就要向用户呈现一个结果视图。在 Struts2 中，预定义了多种 ResultType(结果类型)展示结果视图。

一个结果类型就是实现了 com. opensymphony. xwork2. Result 接口的类，Struts2 把内置的＜result-type＞都放在 struts-default 包中，struts-default 包就是配置包的父包，这个包定义在 struts2-core-2. 3. 24. jar 包的根目录下的 struts-default. xml 文件中，在该文件中可以找到相关的＜result-type＞的定义，其代码如下所示：

```
<result-types>
<result-type name="chain"
    class="com.opensymphony.xwork2.ActionChainResult"/>
<result-type name="dispatcher"
    class="org.apache.struts2.dispatcher.ServletDispatcherResult"
    default="true"/>
<result-type name="freemarker"
    class="org.apache.struts2.views.freemarker.FreemarkerResult"/>
<result-type name="httpheader"
    class="org.apache.struts2.dispatcher.HttpHeaderResult"/>
<result-type name="redirect"
    class="org.apache.struts2.dispatcher.ServletRedirectResult"/>
<result-type name="redirectAction"
    class="org.apache.struts2.dispatcher.ServletActionRedirectResult"/>
<result-type name="stream"
    class="org.apache.struts2.dispatcher.StreamResult"/>
<result-type name="velocity"
    class="org.apache.struts2.dispatcher.VelocityResult"/>
<result-type name="xslt"
    class="org.apache.struts2.views.xslt.XSLTResult"/>
<result-type name="plainText"
    class="org.apache.struts2.dispatcher.PlainTextResult" />
```

```
<result-type name="postback"
    class="org.apache.struts2.dispatcher.PostbackResult" />
</result-types>
```

在上述代码中，每个<result-type>元素都是一种视图技术或者跳转方式的封装，其中 name 属性指出在<result>元素中如何引用这种视图技术或者跳转方式，它对应着<result>元素的 type 属性。class 属性表示这种结果类型的对应类。

Struts2 中预定义的 ResultType 说明如表 18-3 所示。

表 18-3　Struts2 中预定义的 ResultType

属　性	说　明
chain	用于处理 Action 链，被跳转的 Action 中仍能获取上个页面的值，如 request 信息
dispatcher	用于转向页面，通常处理 JSP，是默认的结果类型
freemarker	用于整合 FreeMarker 模板结果类型
httpheader	用于处理特殊的 HTTP 行为结果类型
redirect	重定向到一个 URL，被跳转的页面中丢失传递的信息
redirectAction	重定向到一个 Action，跳转的页面中丢失传递的信息
stream	向浏览器发送 InputStream 对象，通常用于处理文件下载，还可用于 Ajax 数据
velocity	用于整合 Velocity 模板结果类型
xslt	用于整合 XML/XSLT 结果类型
plainText	显示原始文件内容，如文件源代码
postback	使当前请求参数以表单形式提交

表 18-3 列举了 Struts2 中预定义的全部 11 种结果类型，其中 dispatcher 是默认的结果类型，主要用于与 JSP 整合。在这全部 11 种结果类型中，dispatcher 和 redirect 是比较常用的结果类型。

需要注意的是，redirect 与 dispatcher 结果类型非常相似，所不同的是 dispatcher 结果类型是将请求转发到 JSP 视图资源，而 redirect 结果类型是将请求重定向到 JSP 视图资源。如果重定向了请求，那么将丢失所有参数，包括 Action 的处理结果。

18.5.3　dispatcher 结果类型

dispatcher 是 Struts2 的默认结果类型，它用于表示转发到指定结果资源。由于 Struts2 在后台使用 RequestDispatcher 的 forward()方法转发请求，所以在用户的整个请求/响应过程中，保持的是同一个请求对象，即目标 JSP/Servlet 接收到的请求/响应对象与最初的 JSP/Servlet 的请求/响应对象相同。

dispatcher 结果类型的对应类是 org. apache. struts2. dispatcher. ServletDispatcherResult，该类有 location 和 parse 两个属性，可以通过 struts. xml 配置文件中的<result>元素的<param>子元素设置，代码如下所示：

```
<result name="success" type="dispatcher">
    <param name="location">/success.jsp</param>
    <param name="parse">true</param>
</result>
```

在上述代码中，location 参数用于指定 Action 执行完毕后要转向的目标资源；parse 参数是一个布尔类型的值，默认是 true，表示解析 location 参数中的 OGNL 表达式，如果为 false，则不解析。

18.5.4 redirect 结果类型

redirect 结果类型用于重定向到指定的结果资源，该资源可以是 JSP 文件，也可以是 Action 类。使用 redirect 结果类型时，系统将调用 HttpServletResponse 的 sendRedirect() 方法将请求重定向到指定的 URL。

redirect 结果类型的对应类是 org. apache. struts2. dispatcher. ServletRedirectResult。在使用 redirect 时，用户要完成一次和服务器之间的交互，浏览器需要发送两次请求，请求过程如图 18-8 所示。

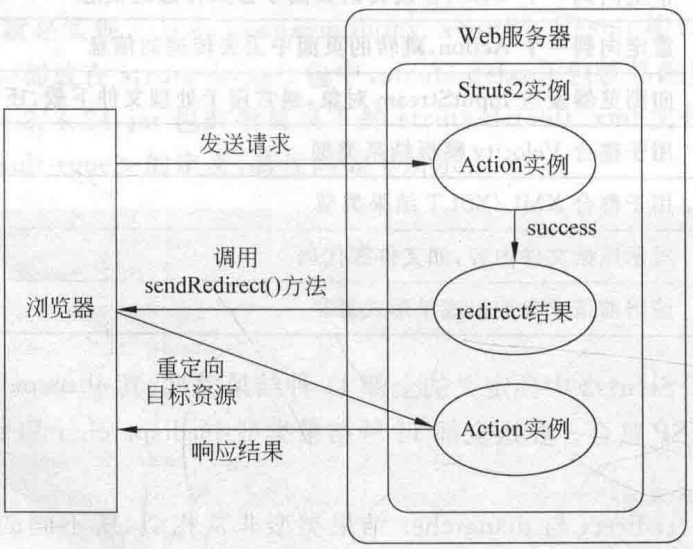


图 18-8 Redirect 结果类型的工作原理

使用 redirect 结果类型的工作过程如下。

(1) 浏览器发出一个请求，Struts2 框架调用对应的 Action 实例对请求进行处理。

(2) Action 返回 success 结果字符串，Struts2 框架根据这个结果选择对应的结果类型，这里使用的是 redirect 结果类型。

(3) ServletRedirectResult 在内部使用 HttpServletResponse 的 sendRedirect()方法将请求重新定向到目标资源。

(4) 浏览器重新发起一个针对目标资源的新请求。

(5) 目标资源作为响应呈现给用户。

下面通过修改第 18. 3. 1 节的登录案例，演示如何使用 redirect 结果类型。修改配置文

件 struts. xml 中 LoginAction 的配置信息，具体如下所示：

```
<action name="login" class="com.itheima.action.LoginAction">
    <result name="success" type="redirect">/success.jsp</result>
    <result name="error" type="dispatcher">/error.jsp</result>
</action>
```

在上述配置代码中，将成功登录的结果类型设置为 redirect，它表示当 Action 处理请求后会重新生成一个请求。将错误的结果类型设置为 dispatcher，这也是结果类型的默认值。

启动项目，在浏览器的地址栏中输入地址 http://localhost:8080/chapter18/login. jsp，成功访问页面后，分别输入用户名“admin”和密码“123456”，单击【登录】按钮后，如果用户名和密码正确，将重定向到 success. jsp 页面，请求地址栏中显示的是 success. jsp，而不是 login. action，如图 18-9 所示。

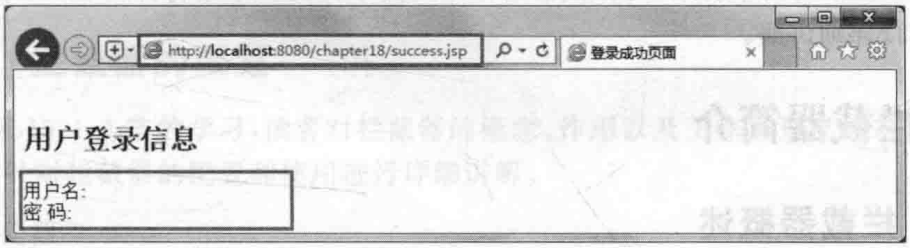


图 18-9　重定向页面的效果

从图 18-9 中可以看出，地址栏中显示的是 success. jsp。由于使用 redirect 重定向到其他资源时，将重新产生一个请求，并且原来的请求内容和参数将全部丢失，所以页面中的用户名和密码没有显示。

当用户名和密码错误时，使用的是 dispatcher 结果类型，此时页面将跳转到 error. jsp，由于是请求转发行为，所以地址栏中显示的还是 login 的请求信息，但页面中显示的是 error. jsp 中的内容，如图 18-10 所示。

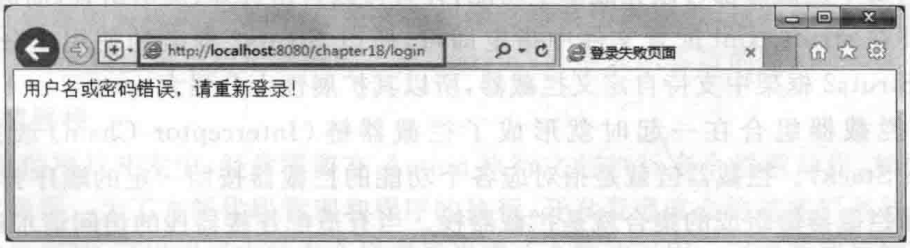


图 18-10　转发页面的效果

第 19 章 Struts2的拦截器

在 Struts2 框架中，拦截器是其重要的组成部分，Struts2 的很多功能（数据校验、对象类型转换、文件上传等）都是构建在拦截器之上的。通常情况下，开发者通过 Struts2 内建的拦截器可以完成大部分的操作，只有在内建拦截器不能满足需求时，才会自己扩展。可以这么说，Struts2 框架的简单易用，与拦截器的作用是分不开的。本章将针对 Struts2 拦截器的相关知识进行详细讲解。

19.1 拦截器简介

19.1.1 拦截器概述

拦截器（Interceptor）是 Struts2 框架的核心组成部分，它类似于 Servlet 中的过滤器，是一种可以在请求之前或之后执行的 Struts2 的组件，也可以将其理解为动态拦截 Action 调用的对象。

在早期的 MVC 框架中，通常会将一些通用的操作（如类型转换、数据校验、解析上传的文件等）强制写在控制器中，而这些常用操作又不是所有的请求都需要实现的，这就导致了框架的灵活性不足、可扩展性低等问题。在 Struts2 框架中，这些通用的核心功能都放到了拦截器中实现，而不是集中放在核心控制器中实现。由于框架中各个功能对应的拦截器是分开定义的，每个拦截器都可以完成单个功能，并且可以自由选择、灵活组合，而需要哪些拦截器时，只要在 struts. xml 配置文件中指定即可，所以 Struts2 框架的使用十分灵活。同时，由于在 Struts2 框架中支持自定义拦截器，所以其扩展性十分强大。

当多个拦截器组合在一起时就形成了拦截器链（Interceptor Chain）或拦截器栈（Interceptor Stack）。拦截器链就是指对应各个功能的拦截器按照一定的顺序排列在一起形成的链，而拦截器链组成的集合就是拦截器栈。当有适配连接器栈的访问请求进来时，这些拦截器就会按照之前定义的顺序被调用。

在通常情况下，拦截器都是以代理方式调用的，它在一个 Action 执行前后进行拦截，围绕着 Action 和 Result 的执行而执行，其工作方式如图 19-1 所示。

从图 19-1 中可以看出，Struts2 拦截器的实现原理与 Servlet 过滤器的实现原理类似，它以链式执行，对真正要执行的方法（execute()）进行拦截。在执行 Action 的 execute()方法之前会执行一次拦截，在 Action 和 Result 执行之后，拦截器会再次执行（与先前的调用顺序相反）。在此链式执行的过程中，每一个拦截器都可以直接返回，从而终止余下的拦截器、Action 及 Result 的执行。

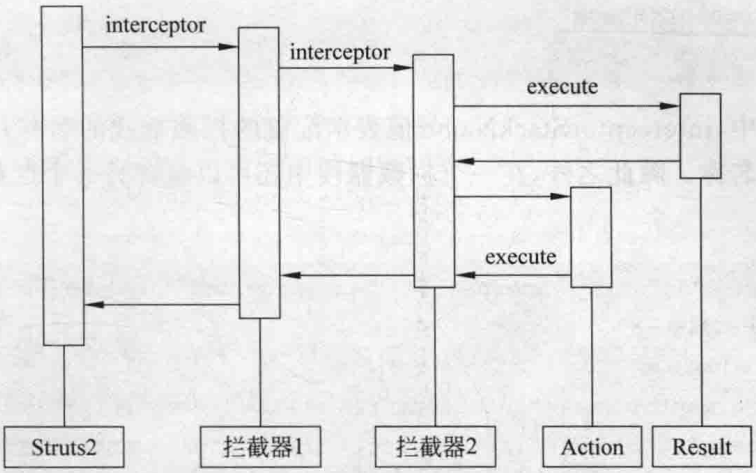


图 19-1　Struts2 拦截器的工作方式

19.1.2　拦截器的配置

通过第 19.1.1 节的学习，读者对拦截器的概念、作用以及工作原理已经有了一定的了解，本节将针对拦截器的配置和使用进行详细讲解。

1. 拦截器

要使用拦截器，首先要对它进行配置。拦截器的配置是在 struts.xml 文件中完成的，它通常以＜interceptor＞标签开头，以＜/interceptor＞标签结束。定义拦截器的语法格式如下所示：

```
<interceptor name="interceptorName" class="interceptorClass">
    <param name="paramName">paramValue</param>
</interceptor>
```

上述语法格式中，＜interceptor＞元素的 name 属性用于指定拦截器的名称，class 属性用于指定拦截器的实现类。有时，在定义拦截器时需要传入参数，这时需要使用＜param＞标签，其中 name 属性用于指定参数的名称，paramValue 表示参数的值。

2. 拦截器栈

在实际的项目开发中，经常需要在 Action 执行之前执行多个拦截动作，如登录日志记录、权限管理等。为了方便代码管理和程序的执行，开发者通常会将这些拦截器组成一个拦截器栈，在使用时，可以将栈内的多个拦截器当成一个整体引用。当拦截器栈被附加到一个 Action 上时，在执行 Action 之前必须先执行拦截器栈中的每一个拦截器。

定义拦截器栈使用＜interceptors＞元素和＜interceptor-stack＞子元素，当配置多个拦截器时，需要使用＜interceptor-ref＞元素指定多个拦截器，配置语法如下所示：

```
<interceptors>
    <interceptor-stack name="interceptorStackName">
        <interceptor-ref name="interceptorName" />
        ...
```

```
    </interceptor-stack>
</interceptors>
```

在上述语法中，interceptorStackName 值表示配置的拦截器栈的名称；interceptorName 值表示拦截器的名称。除此之外，在一个拦截器栈中还可以包含另一个拦截器栈，示例代码如下所示：

```
<package name="default" namespace="/" extends="struts-default">
    <!--声明拦截器-->
    <interceptors>
        <interceptor name="interceptor1" class="interceptorClass"/>
        <interceptor name="interceptor2" class="interceptorClass"/>
        <!--定义一个拦截器栈 myStack,该拦截器栈中包含两个拦截器和一个拦截器栈-->
        <interceptor-stack name="myStack">
            <interceptor-ref name="defaultStack" />
            <interceptor-ref name="interceptor1" />
            <interceptor-ref name="interceptor2" />
        </interceptor-stack>
    </interceptors>
</package>
```

在上述代码中，定义的拦截器栈的名称是 myStack，在 myStack 栈中，除了引用了两个自定义的拦截器 interceptor1 和 interceptor2 以外，还引用了一个内置拦截器栈 defaultStack，这个拦截器是必须要引入的。

3. 默认拦截器

如果想对一个包下的 Action 使用相同的拦截器，则需要为该包中的每个 Action 都重复指定同一个拦截器，这样写显然过于繁琐。为了解决此问题，Struts2 中支持使用默认拦截器，它可以对其指定的包中的所有 Action 都起到拦截作用。一旦为某一个包指定了默认拦截器，并且该包中的 Action 未显示指定拦截器，则会使用默认拦截器。反之，若此包中的 Action 显示的指定了某个拦截器，则该默认拦截器将会被屏蔽。此时，如果还想使用默认拦截器，则需要用户手动配置该默认拦截器的引用。

配置默认拦截器需要使用＜default-interceptor-ref＞元素，此元素为＜package＞元素的子元素。其语法格式如下所示：

```
<default-interceptor-ref name="拦截器(栈)的名称"/>
```

在上述语法格式中，name 属性的值必须是已经存在的拦截器或拦截器栈的名称。下面用该语法格式配置一个默认拦截器，示例代码如下所示：

```
<package name="default" namespace="/" extends="struts-default">
    <!--声明拦截器-->
    <interceptors>
        <interceptor name="interceptor1" class="interceptorClass"/>
```

```
        <interceptor name="interceptor2" class="interceptorClass"/>
    <!--定义一个拦截器栈 myStack,该拦截器栈包含两个拦截器和一个拦截器栈-->
        <interceptor-stack name="myStack">
            <interceptor-ref name="interceptor1" />
            <interceptor-ref name="interceptor2" />
            <interceptor-ref name="defaultStack" />
        </interceptor-stack>
    </interceptors>
    <!--配置包下的默认拦截器,既可以是拦截器,也可以是拦截器栈-->
    <default-interceptor-ref name="myStack"/>
    <action name="login" class="com.itheima.action.LoginAction">
        <result name="input">/login.jsp</result>
    </action>
</package>
```

在上述代码中,指定了包下面的默认拦截器为一个拦截器栈,该拦截器栈将会作用于包下所有的 Action。

注意:每一个包下只能定义一个默认拦截器,如果需要多个拦截器作为默认拦截器,则可以将这些拦截器定义为一个拦截器栈,再将这个拦截器栈作为默认拦截器即可。

19.2　Struts2 的内建拦截器

Struts2 框架中内置了许多拦截器,这些拦截器以 name-class 对的形式配置在 struts-default.xml 文件中,其中,name 是拦截器的名称,也就是引用的名字;class 指定了该拦截器所对应的实现。只要自定义的包继承了 Struts2 的 struts-default 包,就可以使用包中定义的内建拦截器,否则需要自行定义拦截器。

19.2.1　内建拦截器的介绍

在 struts-default.xml 中,每一个拦截器都具有不同的意义,如表 19-1 所示。

表 19-1　内建拦截器的名称及说明

名　称	说　明
alias	在不同请求之间将请求参数在不同名称间转换,请求内容不变
autowiring	用于实现 Action 的自动装配
chain	让前一个 Action 的属性可以被后一个 Action 访问,现在和 chain 类型的 result() 结合使用
conversionError	将错误从 ActionContext 中添加到 Action 的属性字段中
cookies	使用配置的 Name 和 Value 指定 Cookies
cookieProvider	该类是一个 Cookie 工具,方便开发者向客户端写 Cookie
clearSession	用于清除一个 HttpSession 实例

续表

名　称	说　明
createSession	自动创建 HttpSession，用于为需要使用 HttpSession 的拦截器服务
debugging	提供不同的调试用的页面展现内部的数据状况
execAndWait	在后台执行 Action，同时将用户带到一个中间的等待页面
exception	将异常定位到一个画面
fileUpload	提供文件上传功能
i18n	记录用户选择的 locale
logger	输出 Action 的名称
model-driven	如果一个类实现了 ModelDriven，将 getModel 得到的结果放在 Value Stack 中
scoped-model-driven	如果一个 Action 实现了 ScopedModelDriven，则这个拦截器会从相应的 Scope 中取出 model 调用 Action 的 setModel 方法，将其放入 Action 内部
params	将请求中的参数设置到 Action 中
actionMappingParams	用于负责在 Action 配置中传递参数
prepare	如果 Acton 实现了 Preparable，则该拦截器调用 Action 类的 prepare 方法
staticParams	将 struts. xml 文件中＜action＞标签的参数内容设置到对应的 Action 中
scope	将 Action 状态存入 session 和 application 范围
servletConfig	提供访问 HttpServletRequest 和 HttpServletResponse 方法，以 Map 方式访问
timer	输出 Action 执行的时间
token	通过 Token 避免双击
tokenSession	和 Token Interceptor 一样，不过双击时把请求的数据存储在 Session 中
validation	使用 action-validation. xml 文件中定义的内容校验提交的数据
workflow	调用 Action 的 validate 方法，一旦有错误返回，则重新定位到 INPUT 画面
store	存储或者访问实现 ValidationAware 接口的 Action 类出现的消息、错误和字段错误等
checkbox	添加了 checkbox 自动处理代码，将没有选中的 checkbox 的内容设定为 false，而 html 在默认情况下不提交没有选中的 checkbox
datetime	日期拦截器
profiling	通过参数激活 profile
roles	确定用户是否具有 JAAS 指定的 Role，否则不予执行
annotationWorkflow	利用注解代替 XML 配置，使用 annotationWorkflow 拦截器可以使用注解，执行流程为 before-execute-feforeResult-after
multiselect	检测是否有像＜select＞标签一样被选中的多个值，然后添加一个空参数
deprecation	当日志级别设置为调试模式(debug)并且没有特殊参数时，在 devMode 模式中，会检查应用程序使用过时或未知的常量，并且显示警告

Struts2 框架除了提供这些有用的拦截器以外，还定义了一些拦截器栈，在开发 Web 应

用时，可以直接引用这些拦截器栈，而无须自定义拦截器。

注意：随着 Struts2 版本的发展，内建拦截器的数量也在相应地增多，不同版本的 Struts2 拦截器的数量有一些差异，此版本的 Struts2 内建拦截器共有 35 个。这些内建拦截器读者不需要记忆，只需要了解即可。

19.2.2　内建拦截器的配置

通过第 19.2.1 节的学习，读者对 Struts2 的内建拦截器以及它们的意义已经有了一定的了解。为了使读者更好地掌握 Struts2 拦截器的知识，下面讲解在 struts-default.xml 中定义的拦截器的部分配置。

在 struts-core-2.3.24.jar 包中的根目录下找到 struts-default.xml 文件，打开后找到<interceptors>元素下的内建拦截器和拦截器栈，其部分代码如下所示：

```
<package name="struts-default" abstract="true">
…
<interceptors>
<!--系统内建拦截器部分,上小节介绍的内容-->
    <interceptor name="alias"
    class="com.opensymphony.xwork2.interceptor.AliasInterceptor"/>
    <interceptor name="chain"
    class="com.opensymphony.xwork2.interceptor.ChainingInterceptor"/>
    …
    <!--定义 Basic stack 拦截器栈-->
    <interceptor-stack name="basicStack">
        <!--引用系统定义的 exception 拦截器-->
        <interceptor-ref name="exception"/>
        …
    </interceptor-stack>
    …
    <!-定义 Sample model-driven stack -->
    <interceptor-stack name="modelDrivenStack">
        <!--引用系统定义的 modelDriven 拦截器-->
        <interceptor-ref name="modelDriven"/>
        <!--引用系统定义的 basicStack 拦截器栈-->
        <interceptor-ref name="basicStack"/>
    </interceptor-stack>
    …
    <!--定义 defaultStack 拦截器栈-->
    <interceptor-stack name="defaultStack">
        <interceptor-ref name="exception"/>
        <interceptor-ref name="alias"/>
        <interceptor-ref name="i18n"/>
        …
        <interceptor-ref name="validation">
```

```
            <param name="excludeMethods">input,back,cancel,browse</param>
        </interceptor-ref>
        ...
    </interceptor-stack>
</interceptors>
<!--将 defaultStack 拦截器栈配置为系统默认拦截器栈-->
<default-interceptor-ref name="defaultStack"/>
<!-默认 action 类是 ActionSupport-->
<default-class-ref class="com.opensymphony.xwork2.ActionSupport" />
</package>
```

在上述内建拦截器的配置代码中，defaultStack 拦截器组合了多个拦截器，这些拦截器可以满足大部分 Web 应用程序的需求。使用时，只要在 struts. xml 定义包的过程中继承 struts-default 包，那么 defaultStack 拦截器栈就是默认拦截器的引用。由于本书篇幅有限，这里没有列出所有的内建拦截器和拦截器栈，读者需要时可以自行查阅 struts-default. xml 文件。

19.3 自定义拦截器

在实际的项目开发中，虽然 Struts2 的内建拦截器可以完成大部分的拦截任务，但是，一些与系统逻辑相关的通用功能(如权限的控制和用户登录控制等)，则需要通过自定义拦截器实现。本节将详细讲解如何自定义拦截器。

19.3.1 实现自定义拦截器

在 Struts2 框架中，通常开发人员所编写的自定义拦截器类都会直接或间接地实现 com. opensymphony. xwork2. interceptor. Interceptor 接口。Interceptor 接口中的主要代码如下所示：

```
public interface Interceptor extends Serializable {
    void init();
    void destroy();
    String intercept(ActionInvocation invocation) throws Exception;
}
```

从上述代码中可以看出，该接口共提供了以下三个方法。

- void init()：该方法在拦截器被创建后会立即被调用，它在拦截器的生命周期内只被调用一次。可以在该方法中对相关资源进行必要的初始化。
- void destroy()：该方法与 init()方法相对应，在拦截器实例被销毁之前，将调用该方法释放和拦截器相关的资源，它在拦截器的生命周期内，也只被调用一次。
- String intercept(ActionInvocation invocation) throws Exception：该方法是拦截器的核心方法，用于添加真正执行拦截工作的代码，实现具体的拦截操作，它返回一个字符串作为逻辑视图，系统根据返回的字符串跳转到对应的视图资源。每拦截一个

动作请求，该方法就会被调用一次。该方法的 ActionInvocation 参数包含了被拦截的 Action 的引用，可以通过该参数的 invoke()方法，将控制权转给下一个拦截器或者转给 Action 的 execute()方法。

除了实现 Interceptor 接口可以自定义拦截器以外，在实际开发过程中，更常用的一种方式是继承抽象拦截器类 AbstractIntercepter。该类实现了 Interceptor 接口，并且提供了 init()方法和 destroy()方法的空实现。使用时，可以直接继承该抽象类，而不用实现那些不必要的方法。AbstractInterceptor 类中定义的方法如下所示：

```
public abstract class AbstractInterceptor implements Interceptor {
    public void init() {}
    public void destroy() {}
    public abstract String intercept(ActionInvocation invocation) throws Exception;
}
```

从上述代码中可以看出，AbstractInterceptor 类已经实现了 Interceptor 接口的所有方法，一般情况下，只需继承 AbstractInterceptor 类，实现 interceptor()方法就可以创建自定义拦截器。

需要注意的是，只有当自定义的拦截器需要打开系统资源时，才需要覆盖 AbstractInterceptor 类的 init()方法和 destroy()方法。与实现 Interceptor 接口相比，继承 AbstractInterceptor 类的方法更为简单。

19.3.2　使用拦截器实现权限控制

网上购物对于人们并不陌生，当人们浏览购物网站，看中某个商品并选择购买后，如果之前用户没有登录该网站，则页面会跳转到登录页面并提示用户尚未登录；如果用户已登录并单击购买商品时，则会跳转到支付（或结算）页面中进行后续操作。在上述需求中，对未登录用户的权限控制就可以通过拦截器实现。下面编写案例模拟购物网站流程，要求使用 Struts2 的拦截器实现对用户的权限控制。

1. 创建项目

在 Eclipse 中创建一个 Web 项目 chapter19，将 Struts2 框架所需的 JAR 包添加到项目 WEB-INF 目录下的 lib 文件夹中，并发布到类路径下。

2. 配置 web.xml 文件

在 web.xml 文件中配置过滤器和首页信息，其配置代码如文件 19-1 所示。

文件 19-1　web.xml

```
1  <? xml version="1.0" encoding="UTF-8"? >
2  <web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
3          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4          xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
5                          http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
6       <!--定义 filter-->
7      <filter>
```

```
        <!--filter 名字-->
        <filter-name>struts2</filter-name>
        <!--fiter 的实现类,此处是 Struts2 的核心过滤器-->
        <filter-class>
            org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
        </filter-class>
    </filter>
    <filter-mapping>
        <!--filter 的名字,必须是 filter 元素中已经声明过的过滤器的名字-->
        <filter-name>struts2</filter-name>
        <!--定义 filter 负责拦截的 URL 地址-->
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!--首页-->
    <welcome-file-list>
        <welcome-file>main.jsp</welcome-file>
    </welcome-file-list>
</web-app>
```

3. 创建用户实体类

在 src 目录下创建一个名为 com. itheima. domain 的包,在该包下创建 User 类,如文件 19-2 所示。

文件 19-2 User. java

```
package com.itheima.domain;
public class User {
    private String username; //用户名
    private String password; //密码
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username=username;
    }
    public String getPassword() {
        return password;
    }
    public void setPassword(String password) {
        this.password=password;
    }
}
```

在文件 19-2 中,定义了 username 和 password 两个属性,以及其 getter 和 setter 方法。

4. 创建 Action

(1) 创建登录 Action。在 src 目录下创建包 com. itheima. action，在该包下创建 LoginAction 类，编辑后如文件 19-3 所示。

文件 19-3　LoginAction. java

```
1  package com.itheima.action;
2  import com.itheima.domain.User;
3  import com.opensymphony.xwork2.ActionContext;
4  import com.opensymphony.xwork2.ActionSupport;
5  import com.opensymphony.xwork2.ModelDriven;
6  public class LoginAction extends ActionSupport implements ModelDriven<User>{
7      private static final long serialVersionUID=1L;
8      private User user=new User();
9      public User getModel() {
10         return user;
11     }
12     public String execute() throws Exception {
13         //获取 ActionContext
14         ActionContext actionContext=ActionContext.getContext();
15         if ("admin".equals(user.getUsername())&&"123456".
16         equals(user.getPassword()))
17             //将用户存储在 session 中
18             actionContext.getSession().put("user", user);
19             return SUCCESS;
20         } else {
21             actionContext.put("msg", "用户名或密码错误,请重新登录!");
22             return INPUT;
23         }
24     }
25 }
```

(2) 创建商品 Action。在 com. itheima. action 包下创建 BookAction 类，编辑后如文件 19-4 所示。

文件 19-4　BookAction. java

```
1  package com.itheima.action;
2  import com.opensymphony.xwork2.ActionSupport;
3  public class BookAction extends ActionSupport {
4      private static final long serialVersionUID=1L;
5      //购买图书
6      public String buy() {
7          return SUCCESS;
8      }
9  }
```

5. 创建拦截器

通过之前对拦截器的学习，可以将自定义拦截器的使用过程分为以下三步。

(1) 用户自定义的拦截器类，必须实现 Interceptor 接口或继承 AbstractInterceptor 类。

(2) 需要在 Struts. xml 中定义自定义的拦截器。

(3) 在 Struts. xml 中的 Action 中使用拦截器。

下面就按照上述步骤创建拦截器。

(1) 创建拦截器类。在 src 目录下创建一个名称为 com. itheima. interceptor 的包，在该包下创建一个名称为 PrivilegeInterceptor. java 的文件，如文件 19-5 所示。

文件 19-5 PrivilegeInterceptor. java

```
1  package com.itheima.interceptor;
2  import com.opensymphony.xwork2.Action;
3  import com.opensymphony.xwork2.ActionContext;
4  import com.opensymphony.xwork2.ActionInvocation;
5  import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
6  public class PrivilegeInterceptor extends AbstractInterceptor {
7      private static final long serialVersionUID=1L;
8      public String intercept(ActionInvocation invocation) throws Exception {
9          //得到 ActionContext
10         ActionContext actionContext=invocation.getInvocationContext();
11         //获取 user 对象
12         Object user=actionContext.getSession().get("user");
13         if (user !=null) {
14             return invocation.invoke();      //继续向下执行
15         } else {
16             actionContext.put("msg", "您还未登录,请先登录!");
17             return Action.LOGIN;             //如果用户不存在,则返回 login 值
18         }
19     }
20 }
```

(2) 配置 struts. xml 文件。在 src 目录下创建 struts. xml 文件，此文件用于声明自定义拦截器、拦截器栈以及对 book 操作的 Action，其代码如文件 19-6 所示。

文件 19-6 struts. xml

```
1  <? xml version="1.0" encoding="UTF-8" ? >
2  <!DOCTYPE struts PUBLIC
3      "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
4                      "http://struts.apache.org/dtds/struts-2.3.dtd">
5  <struts>
6      <package name="struts2" namespace="/" extends="struts-default">
7          <!--声明拦截器-->
```

```
        <interceptors>
            <interceptor name="privilege"
                class="com.itheima.interceptor.PrivilegeInterceptor"/>
            <interceptor-stack name="myStack">
                <interceptor-ref name="defaultStack" />
                <interceptor-ref name="privilege">
            </interceptor-stack>
        </interceptors>
        <!--用户登录操作-->
        <action name="login" class="com.itheima.action.LoginAction">
            <result>/main.jsp</result>
            <result name="input">/login.jsp</result>
        </action>
        <!--关于 book 操作-->
        <action name="book_*" class="com.itheima.action.BookAction" method="{1}">
            <result>/success.jsp</result>
            <result name="login">/login.jsp</result>
            <!--在 action 中使用自定义拦截器-->
            <interceptor-ref name="myStack" />
        </action>
    </package>
</struts>
```

在文件 19-6 中，首先声明了一个名称为 privilege 的拦截器，并将该拦截器放入自定义的拦截器栈 myStack 中；然后配置了用户登录的 action 信息；最后定义了关于对页面 book 操作的 action 信息，并在该 action 中引用自定义的拦截器栈 myStack。

6. 创建显示页面

在 WebContent 目录下创建三个视图页面，分别为主页 main.jsp、登录页面 login.jsp 和操作成功页面 success.jsp。其中主页 main.jsp 的代码如文件 19-7 所示。

文件 19-7　main.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>主页</title>
</head>
<body>
<body>
```

```
    <table border="0">
        <tr>
            <td>《SSH 框架整合实战教程》</td>
            <td><a href="/chapter19/book_buy">购买</a></td>
        </tr>
    </table>
</body>
</html>
```

登录页面 login.jsp 的代码如文件 19-8 所示。

文件 19-8　login.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>登录</title>
</head>
<style type="text/css">
    input[type=text], input[type=password] {width: 150px;}
</style>
<body>
<body>
<center>
    <form action="/chapter19/login.action" method="post">
     <table>
        <tr>
            <td><label style="text-align: right;">用户名:</label></td>
            <td><input type="text" name="username"></td>
            <td><span style="color: #F00">${requestScope.msg}</span></td>
        </tr>
        <tr>
            <td>
<label style="text-align: right;">密     码:</label>
            </td>
            <td><input type="password" name="password"></td>
        </tr>
        <tr>
            <td align="right" colspan="2"><input type="submit" value="登录">
```

```
            </td>
        </tr>
    </table>
    </form>
</center>
</body>
</html>
```

操作成功页面 success.jsp 的主要代码如下所示。

```
<body>
    尊敬的会员${user.username},您已成功购买商品,祝您购物愉快!
</body>
```

7. 运行项目并查看结果

启动 chapter19 项目,在浏览器的地址栏中输入地址 http://localhost:8080/chapter19/main.jsp 访问其主页,浏览器的显示结果如图 19-2 所示。

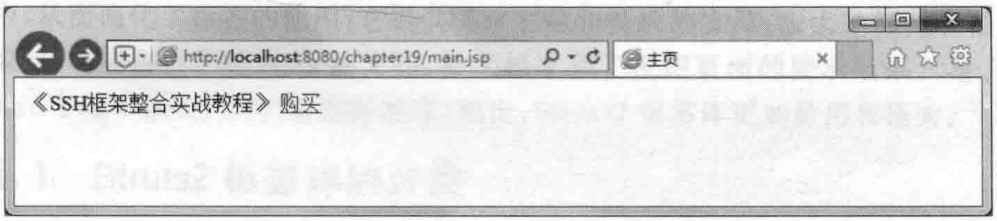


图 19-2　主页

当单击购买链接后,页面将跳转到登录页面中,并提示用户登录,如图 19-3 所示。

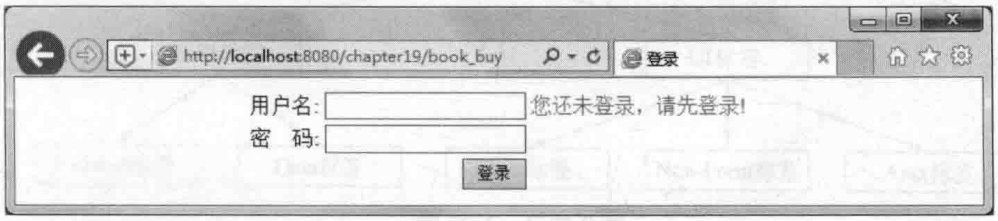


图 19-3　登录页面

如果输入错误的用户名和密码,将提示用户名或密码错误信息,如图 19-4 所示。

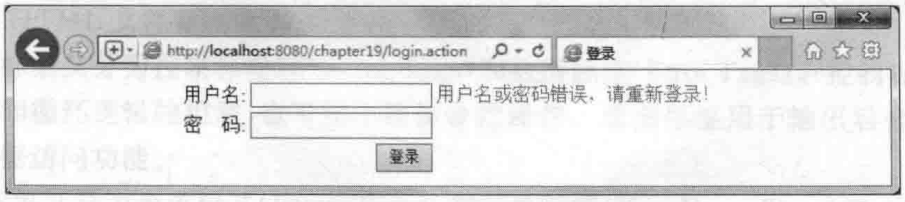


图 19-4　登录页面

当输入正确的用户名和密码并登录成功后,会再次跳转到主页,此时如果单击购买链

接，将提示用户已经成功购买商品，如图 19-5 所示。

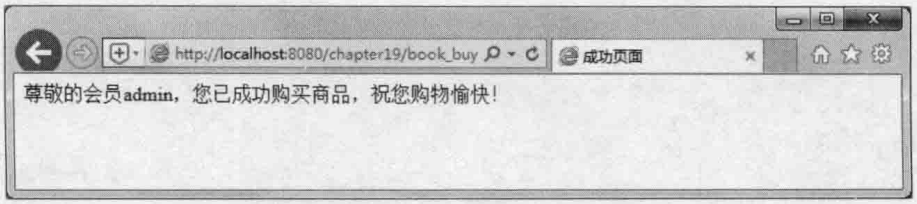


图 19-5　成功页面

第 20 章

Struts2的标签库

Struts2 是一个优秀的 MVC 框架，其实现重点主要放在了业务逻辑控制器部分和视图页面部分。控制器部分主要由 Action 提供支持，而视图页面部分则由大量的标签提供支持。本章将针对 Struts2 的标签库进行详细讲解。

20.1 Struts2 标签库概述

Struts2 的标签库是一个比较完善且功能强大的标签库，它将所有标签都统一到一个标签库中，从而简化了标签的使用；它提供了对主题和模板的支持，极大地简化了视图页面代码的编写；它还提供了对 Ajax 的支持，极大地丰富了视图页面的展示效果。与 JSTL(JSP Standard Tag Library，JSP 标准标签库)相比，Struts2 标签库更加易用和强大。

20.1.1 Struts2 标签库的分类

Struts2 框架对整个标签库进行了分类，按其功能大致可分为两类，如图 20-1 所示。

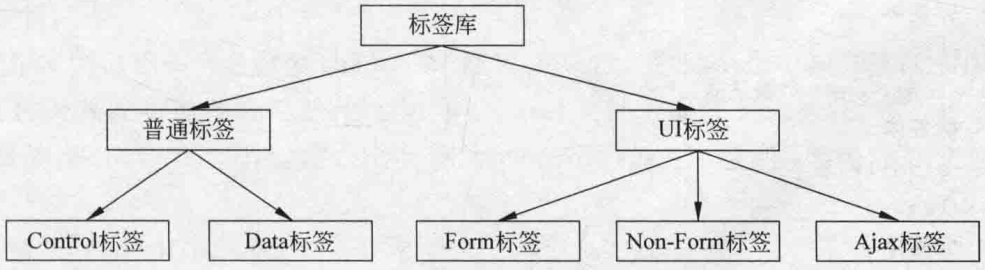


图 20-1 标签分类

从图 20-1 中可以看出，Struts2 的标签库主要分为两大类：普通标签和 UI 标签。普通标签的主要功能是在页面生成时控制页面代码的执行流程；UI 标签的主要功能是以丰富且可复用的 HTML 文件显示数据。

普通标签又分为控制标签(Control Tags)和数据标签(Data Tags)。控制标签用于完成条件逻辑和循环逻辑的控制，也可用于做集合的操作。数据标签用于输出后台的数据和完成其他数据访问功能。

UI 标签又分为表单标签(Form Tags)、非表单标签(Non-Form Tags)和 Ajax 标签。表单标签主要用于生成 HTML 页面中的表单元素，非表单标签主要用于生成非表单的可视化元素，如输出 Action 中封装的信息等。Ajax 标签主要用于提供对 Ajax 技术的支持。

20.1.2 Struts2 标签的使用

使用 Struts2 的标签库非常简单，一般只需在 JSP 文件内使用 taglib 指令导入 Struts2 标签库即可，其导入代码如下所示：

```
<%@taglib prefix="s" uri="/struts-tags" %>
```

在上述代码中，taglib 指令的 uri 属性用于指定引入标签库描述符文件的位置，prefix 属性用于指定引入标签库描述符文件的前缀。在 JSP 文件中，所有的 Struts2 标签都建议使用 s 作为前缀。

20.2 Struts2 的控制标签

针对页面中经常需要实现的流程控制和循环等操作，Struts2 标签库中提供了控制标签。常用的逻辑控制标签主要包括：<s:if>、<s:elseif>、<s:else>和<s:iterator>等。本节将针对这四种常用标签进行详细讲解。

20.2.1 <s:if>、<s:elseif>、<s:else>标签

Struts2 标签库中的<s:if>、<s:elseif>、<s:else>标签与 Java 中的 if、elseif 和 else 语句功能类似，主要用于程序的分支逻辑控制。其中，只有<s:if>标签可以单独使用，而<s:elseif>、<s:else>都必须与<s:if>标签结合才能使用。其使用语法格式如下所示：

```
<s:if test="表达式 1">
    标签体
</s:if>
<s:elseif test="表达式 2">
    标签体
</s:elseif>
<s:else>
    标签体
</s:else>
```

在上述语法格式中，<s:if>和<s:elseif>标签必须指定 test 属性，该属性用于设置标签的判断条件，其值是一个 boolean 类型的条件表达式。

20.2.2 <s:iterator>标签

<s:iterator>标签主要用于对集合中的数据进行迭代，它可以根据条件遍历集合中的数据。<s:iterator>标签的属性及相关说明如表 20-1 所示。

在表 20-1 中，如果在<s:iterator>标签中指定 status 属性，就可以通过该属性获取迭代过程中的状态信息，如元素数、当前索引值等。通过 status 属性获取信息的方法如表 20-2 所示(假设其属性值为 st)。

表 20-1　Iterator 标签的属性

属 性	是否必须	默 认 值	类　型	描　　述
begin	否	0	Integer	迭代数组或集合的起始位置
end	否	数组或集合的长度大小减 1，若 step 为负，则为 0	Integer	迭代数组或集合的结束位置
status	否	false	Boolean	迭代过程中的状态
step	否	1	Integer	指定每一次迭代后索引增加的值
value	否	无	String	迭代的数组或集合对象
var	否	无	String	将生成的 Iterator 设置为 page 范围的属性
id	否	无	String	指定了集合元素的 id，现已用 var 代替

表 20-2　status 获取状态信息

方　　法	说　　明
st. count	返回当前已经遍历的集合元素的个数
st. first	返回当前遍历元素是否为集合的第一个元素
st. last	返回当前遍历元素是否为集合的最后一个元素
st. index	返回遍历元素的当前索引值
st. even	返回当前遍历的元素的索引是否为偶数
st. odd	返回当前遍历的元素的索引是否为奇数

为了让读者更好地掌握<s:if>、<s:else>和<s:iterator>标签的使用，接下来通过一个案例进行演示。

在 Eclipse 中创建一个名称为 chapter20 的 Web 项目，将 Struts2 框架所需的 JAR 包添加到 lib 文件夹并发布到类路径下，然后在 web. xml 文件中配置 Struts2 的核心过滤器信息。在项目的 WebContent 下创建一个名称为 controlTags. jsp 文件，编辑后如文件 20-1 所示。

文件 20-1　controlTags. jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>控制标签</title>
</head>
<body>
```

```
<center>
    <table border="1px" cellpadding="1">
        <s:iterator var="name" value="{'apple','orange','pear','banana'}"
                    status="st">
            <s:if test="#st.odd">
                <tr style="background-color: white;">
                    <td><s:property value="name"/></td>
                </tr>
            </s:if>
            <s:else>
                <tr style="background-color: grey;">
                    <td><s:property value="name"/></td>
                </tr>
            </s:else>
        </s:iterator>
    </table>
</center>
</body>
</html>
```

在文件 20-1 中，首先引入了 Struts2 的标签库，然后在＜table＞标签内，使用＜s:iterator＞标签循环输出集合中的，并将通过该标签 status 属性的 odd 方法获取的值作为＜s:if＞和＜s:else＞标签的判断条件，对表格进行控制显示。

需要注意的是，文件中使用＜s:property＞标签输出值，关于该标签的具体功能会在第 20.3.1 节中讲解，这里读者只需知道使用该标签可以输出值即可。

启动项目后，在浏览器的地址栏中输入地址 http://localhost:8080/chapter20/controlTags.jsp 访问 controlTags.jsp，浏览器的显示效果如图 20-2 所示。

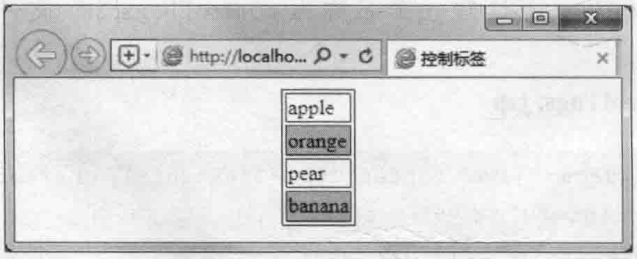


图 20-2 ＜s:if＞、＜s:else＞和＜s:iterator＞标签的使用效果

从图 20-2 的运行结果中可以看出，表格的奇数行变为白色，偶数行变为灰色。这是因为在文件 20-1 中使用＜s:iterator＞遍历集合时，通过＜if＞标签判断其所在索引的奇偶从而改变表格的颜色。

20.3　Struts2 的数据标签

数据标签主要用于提供各种和数据访问相关的功能，如输出信息和显示调试信息等。常用的数据标签有<s:property>、<s:a>、< s:debug>、<s:include>和<s:param>等。

20.3.1　<s:property>标签

<s:property>标签的作用是输出指定的值，通常输出的是 value 属性指定的值，<s:property>标签的属性及属性说明如下。

- value：可选属性，指定需要输出的属性值，如果没有指定该属性，则默认输出 ValueStack 栈顶的值(关于值栈内容会在下一章节进行讲解)。
- id：可选属性，指定该元素的标识。
- default：可选属性，如果要输出的属性值为 null，则显示 default 属性的指定值。
- escape：可选属性，指定是否忽略 HTML 代码。默认值是 true，即忽略输出值中的 HTML 代码。

为了使读者更好地掌握<s:property>标签的使用，下面编写案例进行演示。在项目的 WebContent 目录下创建一个名称为 propertyTags.jsp 的文件，编辑后如文件 20-2 所示。

文件 20-2　propertyTags.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<html>
<head>
<title>property 标签</title>
</head>
<body>
    输出字符串：
    <s:property value="'this is a string'" /><br />
    输出默认值：
    <s:property value="" default="default_value" /><br />
    忽略 HTML 代码：
    <s:property value="'<h2>www.w3school.com.cn</h2>'" escape="true" /><br />
    不忽略 HTML 代码：
    <s:property value="'<h2>www.w3school.com.cn</h2>'" escape="false" /><br />
</body>
</html>
```

在文件 20-2 中，分别对<s:property>标签的 value、default 和 escape 属性的使用进行了演示。启动项目后，在浏览器的地址栏中输入地址 http://localhost:8080/chapter20/

propertyTags.jsp 访问 propertyTags.jsp，浏览器的显示结果如图 20-3 所示。

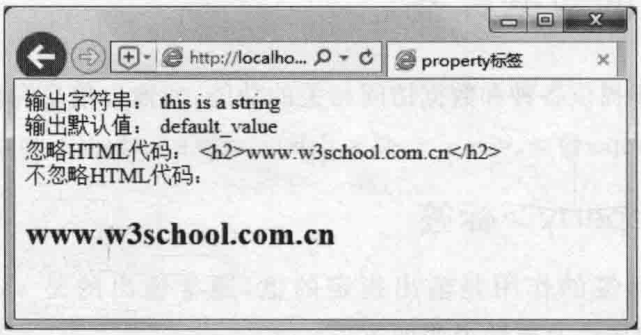


图 20-3 property 标签的输出

20.3.2 <s:a>标签

<s:a>标签与 HTML 中的<a>标签类似，主要用于构造 HTML 页面中的超链接。<s:a>标签的属性及相关说明如表 20-3 所示。

表 20-3 <s:a>标签的常用属性及描述

属 性	是否必须	类 型	描 述
action	否	String	指定超链接 Action 地址
href	否	String	超链接地址
namespace	否	String	指定 Action 地址
id	否	String	指定其 id
method	否	String	指定 Action 调用方法

<s:a>标签的使用格式如下所示：

```
<s:a href="链接地址"></s:a>
<s:a namespace="" action="">www.baidu.com</s:a>
```

20.3.3 <s:debug>标签

<s:debug>标签用于输出服务端对象（如 request、application、ValueStack 等）中的信息，该标签可用于辅助调试 Java Web 程序。

<s:debug>标签只有一个 id 属性，表示<s:debug>标签的一个引用，通常不使用该属性。在使用<s:debug>标签后，网页中会生成一个 Debug 的链接，单击该链接，网页中将输出各种服务器对象的信息，如图 20-4 所示。

20.3.4 <s:include>标签

<s:include>标签用于在当前页面中包含另一个 Web 资源（如 HTML、JSP、Servlet 等）。该标签有两个属性 id 和 value。其中 id 是可选属性，表示该标签的引用；value 是必填

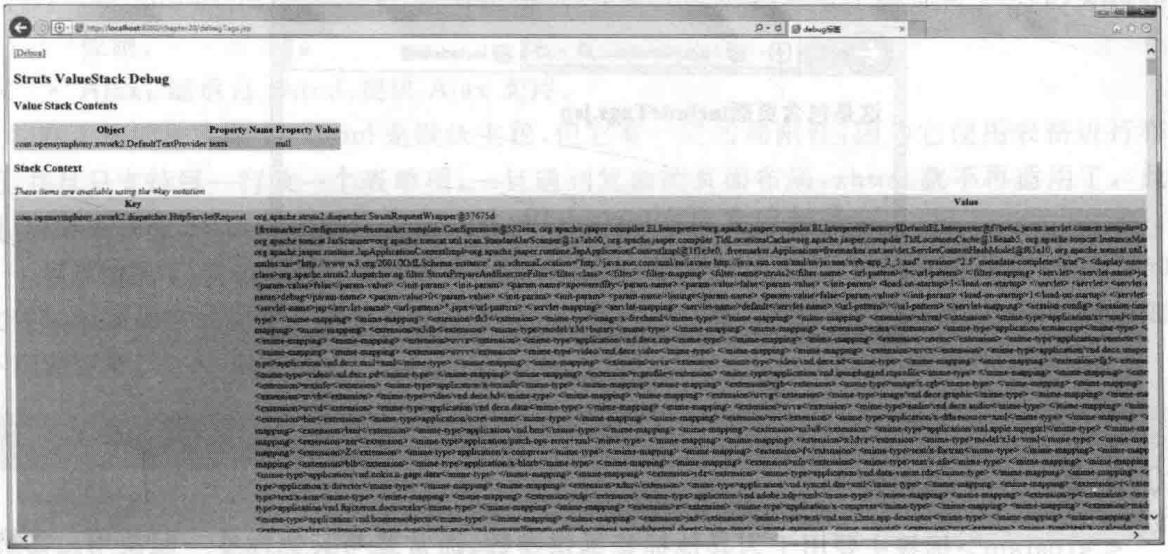


图 20-4　Debug 标签

属性，用于指定被包含的 Web 资源文件。

在<s:include>标签中还可以指定多个<s:param/>子标签给被包含的 Web 资源传递请求参数。下面通过案例演示<s:include>标签的使用。

在 chapter20 项目的 WebContent 目录下创建一个名称为 includeTags.jsp 的页面和一个名称为 file.jsp 的页面，在 includeTags.jsp 页面中使用<s:include>标签包含 file.jsp 页面。其中被包含页面 file.jsp 的主要代码如文件 20-3 所示。

文件 20-3　file.jsp

```
<body>
    <h2>这是被包含页面 includefile.jsp</h2><br />
    传递的参数为：<%out.print(request.getParameter("username"));%>
</body>
```

包含页面 includeTags.jsp 的主要代码如文件 20-4 所示。

文件 20-4　includeTags.jsp

```
<body>
    <h3>这是包含页面 includeTags.jsp</h3><br />
    <s:include value="file.jsp">
        <s:param name="username" value="'小韩'"/>
    </s:include>
</body>
```

启动项目后，在浏览器的地址栏中输入 http://localhost:8080/chapter20/includeTags.jsp 访问 includeTags.jsp，浏览器的显示结果如图 20-5 所示。

从图 20-5 中可以看出，includeTags.jsp 页面使用<s:include>标签已经成功将 file.jsp 页面包含在其中，并且通过<s:param>标签已将参数和参数值传递给了被包含页面 file.jsp。



图 20-5 <s:include>标签的使用

20.3.5 <s:param>

<s:param>标签主要用于为其他标签提供参数，通常要与其他标签一起使用。在第 20.3.4 节中，使用<s:include>标签时，就使用了<s:param>标签给被包含的页面传递参数。<s:param>标签有两种用法，具体如下。

一种用法是通过标签体指定参数值，用法如下所示：

```
<s:param name="color">red</s:param>
```

另一种用法是使用 value 属性指定参数值，用法如下所示：

```
<s:param name="color" value="'red'" />
```

上述两种用法的功能一样，不同点在于使用 value 属性设置参数值时，需要添加单引号，而使用标签体设置参数值时，不需要添加单引号。

注意：在使用 value 属性指定参数时，如果不添加单引号，则表示该值为一个引用对象，如果该对象不存在，则为其属性赋值为 null。

20.4 Struts2 的模板和主题

Struts2 的 UI 标签都是基于模板和主题的。模板就是一些代码，Struts2 标签使用这些代码渲染生成相应的 HTML 代码。模板是一个 UI 标签的外在表现形式，并且每个标签都会有自己对应的模板。如果为所有的 UI 标签提供样式和视觉效果相似的模板，那么这一系列的模板就形成了一个主题。

Struts2 默认提供了四种主题，分别为 simple、xhtml、css_xhtml 和 Ajax，这四种主题的作用如下。

- simple 主题：这是最简单的主题，使用该主题时，每个 UI 标签只生成最基本的 HTML 元素，没有任何附加功能。
- xhtml 主题：这是 Struts2 的默认主题，它对 simple 主题进行了扩展，提供了布局功能、Label 显示名称以及与验证框架和国际化框架的集成。

- css_xhtml：该主题是对 xhtml 的扩展，在 xhtml 的基础之上添加对 CSS 的支持和控制。
- Ajax：继承自 xhtml，提供 Ajax 支持。

在上述四种主题中，xhtml 是默认主题，但它有一定的局限性，因为它使用表格进行布局，并且只支持每一行放一个表单项，一旦遇到复杂的页面布局，xhtml 就不再适用了。此时，就需要改变 Struts2 的默认主题。

通常情况下，可以通过设置常量 struts. ui. theme 改变默认主题，具体方法是在 struts. xml 文件中增加相应的配置。例如想要使用 simple 的主题，那么只需要在 struts. xml 中增加如下配置即可。

```
<constant name="struts.ui.theme" value="simple"/>
```

20.5　Struts2 的表单标签

Struts2 的表单标签用于向服务器提交用户输入的信息，绝大多数的表单标签都有其对应的 HTML 标签。读者可以结合 HTML 的标签对比学习 Struts2 的表单标签。

20.5.1　表单标签的公共属性

每一个表单标签在 Struts2 核心 JAR 文件的 org. apache. struts2. components 包中都有一个对应的类，这些对应的类都继承自 UIBean 类。UIBean 类提供了一组公共属性，这些属性是所有表单标签的通用属性，具体如表 20-4 所示。

表 20-4　表单标签的通用属性

属 性 名	主　题	数据类型	说　明
title	simple	String	设置表单元素的 title 属性
disabled	simple	String	设置表单元素是否可用
label	xhtml	String	设置表单元素的 label 属性
labelPosition	xhtml	String	设置 label 元素的显示位置，可选值为 top 和 left(默认)
name	simple	String	设置表单元素的 name 属性，与 Action 中的属性名对应
value	simple	String	设置表单元素的值
cssClass	simple	String	设置表单元素的 class
cssStyle	simple	String	设置表单元素的 style 属性
required	xhtml	Boolean	设置表单元素为必填项
requiredpositon	xhtml	String	设置必填标记(默认为 *)相对于 label 元素的位置，可选值为 left 和 right(默认)
tabindex	simple	String	设置表单元素的 tabindex 属性

除了这些常用的通用属性以外，表单标签还有很多其他属性。由于本书篇幅有限，这里就不再一一列举。需要注意的是，表单标签的 name 和 value 属性基本等同于 HTML 组件的 name 和 value，但也有所不同：表单标签在生成 HTML 时，如果标签没有设置 value 属性，则会从值栈中按照 name 获取相应的值，并把这个值设置成 HTML 组件的 value。简单而言，就是表单标签的 value 在生成 HTML 时会自动设置值，而这个值是从值栈中获取的。关于值栈问题将在下一章进行讲解。

20.5.2 <s:form>标签

<s:form>标签用于呈现 HTML 语言中的表单元素，其常用属性如表 20-5 所示。

表 20-5 <form>标签的常用属性及描述

属 性 名	是否必填	类 型	说 明
action	否	String	指定提交时对应的 action，不需要 action 后缀
enctype	否	String	HTML 表单 enctype 属性
method	否	String	HTML 表单 method 属性
namespace	否	String	所提交 action 的命名空间

在使用<s:form>标签时，一般会包含其他的表单元素，如<s:textfield>和<s:radio>等标签。在提交表单时，这些表单元素对应的 name 属性，将其作为参数传入 Struts2 框架进行处理。

20.5.3 <s:textfield>和<s:textarea>标签

<s:textfield>和<s:textarea>标签的作用比较相似，都用于创建文本框，其主要区别在于<s:textfield>创建的是单行文本框，而<s:textarea>创建的是多行文本框。二者的使用也比较简单，一般只需要指定其 label 和 name 属性即可。两个标签的用法如下所示。

<s:textfield>标签的用法：

```
<s:textfield label="用户名" name="username"/>
```

<s:textarea>标签的用法：

```
<s:textarea label="描述" name="description"/>
```

在上述两个用法中，name 属性用于指定单行/多行文本框的名称，在 Action 中，通过 name 属性即可获取单行/多行文本框的值。

此外，<textarea>标签还可以通过 value 属性指定单行/多行文本框的当前值，通过 cols 和 rows 属性分别指定多行文本框的列数和行数。

20.5.4 <s:password>标签

<s:password>标签用于创建一个密码输入框，它可以生成 HTML 中的<input type="password" />标签。<s:password>标签的常用属性说明如表 20-6 所示。

表 20-6　<s:password>标签的常用属性说明

属 性 名	说　明
name	用于指定密码输入框的名称
size	用于指定密码输入框的显示宽度，以字符数为单位
maxlength	用于限定密码输入框的最大输入字符串个数
showPassword	是否显示初始值，即使显示也仍为密文显示，用掩码代替

<s:password>标签的使用方法如下所示：

```
<s:password label="password" name="password" maxlength="20"/>
```

需要注意的是，Struts2 的<s:password>标签与 HTML 的<input type="password"/>标签也略有不同：<input type="password" />标签只要设置 value 属性就可以将 value 的属性值作为默认显示值；而 Struts2 的<s:password>标签除了要设置 value 属性值以外，还要设置 showPassword 属性值为 true。

20.5.5　<s:radio>标签

<s:radio>标签用于创建单选按钮，生成 HTML 中的<input type="radio" />标签。<s:radio>标签的常用属性说明如表 20-7 所示。

表 20-7　<s:radio>标签的属性及说明

属性名	是否必填	类　型	说　明
list	是	Collection, Map Enmumeration, Iterator, Array	用于生成单选框中的集合
listKey	否	String	指定集合对象中的哪个属性作为选项的 value
listValue	否	String	指定集合对象中的哪个属性作为选项的内容

表 20-7 中的三个属性必须要配合使用，由 list 属性指定从集合中获得元素，由 listKey 属性指定获得元素之后使用元素的哪个属性作为生成<input type="radio" />的 value 属性，由 listValue 属性指定生成的<input type="radio" />后展示给用户的内容。

20.5.6　<s:reset>标签

<s:reset>标签用于创建一个重置按钮，会生成 HTML 中的<input type="reset"/>标签，该标签的使用比较简单，其常用属性为 name 和 value。其中，name 属性用于指定重置按钮的名称，在 Action 中，可以通过 name 属性获取重置按钮的值，value 属性用于显示按钮的值。该标签的用法如下所示：

```
<s:reset value="reset" />
<s:reset name="reset" value="重置" />
```

20.5.7 <s:submit>标签

<s:submit>标签主要用于产生 HTML 中的提交按钮，该表单元素中，可以指定提交时的 Action 对应的方法。该标签通常与<s:form>标签一起使用，该标签的常用属性如表 20-8 所示。

表 20-8 <s:submit>标签的常用属性

属 性 名	是否必填	类 型	说 明
action	否	String	指定提交时对应的 Action
method	否	String	指定 Action 中调用的方法

下面通过简单的用户注册案例，演示前几个小节中所讲解的标签的使用。在 chapter20 项目的 WebContent 目录下创建一个名称为 register.jsp 的文件，编辑后如文件 20-5 所示。

文件 20-5 register.jsp

```
<%@page language="java" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>用户注册</title>
</head>
<body>
    <s:form action="login">
        <s:textfield label="昵称" name="nickname" />
        <s:textfield label="用户名" name="username" />
        <s:password label="密码" name="password1" />
        <s:password label="确认密码" name="password2" />
        <s:radio name="sex" label="性别" list="#{'0':'男','1':'女'}" value="0" />
        <s:textarea label="个性签名" name="description" rows="3" cols="17" />
        <s:reset value="重置" />
        <s:submit value="注册" />
    </s:form>
</body>
</html>
```

在文件 20-5 中，分别使用了<s:form>、<s:textfield>、<s:textarea>、<s:password>、<s:radio>、<s:reset>和<s:submit>共 7 种标签编写了一个表单页面。在<s:radio>标签中，使用 list 属性定义了一个 Map 集合，并使用 value 属性指定其默认显示值为男(value=0 表示集合中 key 为 0 的元素)。在<s:textarea>标签中，使用 rows 属性和 cols 属性定义了多行文本框的列数为 3，行数为 17。

启动项目后，在浏览器的地址栏中输入地址 http://localhost:8080/chapter20/register.jsp 访问 register.jsp，浏览器的显示结果如图 20-6 所示。



图 20-6　注册页面

20.5.8　<s:checkboxlist>标签

<s:checkboxlist>标签用于一次性创建多个复选框，用户可以选择创建零到多个复选框，它可以产生一组<input type="checkbox" />标签。<s:checkboxlist>标签的常用属性及其说明如表 20-9 所示。

表 20-9　<s:checkboxlist>标签的常用属性及说明

属 性 名	是否必填	类　　型	说　　明
name	否	String	指定该元素的 name
lable	否	String	指定复选框前显示的文本
labelposition	否	String	指定 label 属性的文字内容显示在标签位置(top/left)
list	是	Collection，Map Enmumeration，Iterator，Array	用于生成多选框的集合
listKey	否	String	生成 checkbox 的 value 属性
listValue	否	String	生成 checkbox 后面显示的文字

在表 20-9 的属性中，listKey 和 listValue 属性主要用于集合中，通常存放的是 JavaBean，可以使用这两个属性从 JavaBean 的众多属性中筛选需要的值。为了让读者更好地掌握<s:checkboxlist>标签的使用，下面通过案例演示如何创建多个复选框。

在 WebContent 目录下创建一个名称为 checkboxlistTags.jsp 的页面，编辑后如文件 20-6 所示。

文件 20-6　checkboxlistTags.jsp

```
1  <%@page language="java" contentType="text/html;
2  charset=UTF-8" pageEncoding="UTF-8"%>
3  <%@taglib prefix="s" uri="/struts-tags"%>
```

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>checkboxlist 标签</title>
</head>
<body>
    <s:form>
        <s:checkboxlist name="fruit" label="喜欢吃的水果" labelposition="top"
            list="{'苹果','橘子','香蕉'}" />
        <s:checkboxlist name="sport" label="喜欢的运动" labelposition="top"
            list="#{'a':'羽毛球','b':'乒乓球','c':'网球'}"
            listKey="key" listValue="value" />
    </s:form>
</body>
</html>
```

启动项目后，在浏览器的地址栏中输入地址 http://localhost:8080/chapter20/checkboxlistTags.jsp 访问 checkboxlistTags.jsp，浏览器的显示结果如图 20-7 所示。

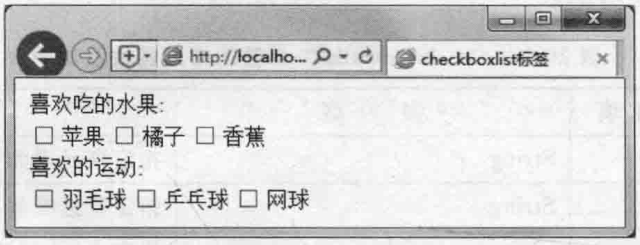


图 20-7 <s:checkboxlist>标签的使用

20.5.9 <s:select>标签

<s:select>标签用于创建一个下拉列表框，它会生成 HTML 中的<select>标签。<s:select>标签的常用属性及其说明如表 20-10 所示。

表 20-10 select 标签的常用属性及说明

属 性 名	是否必填	类 型	说 明
list	是	Collection, Map Enmumeration, Iterator, Array	用于生成下拉框的集合
listKey	否	String	生成选项的 value 属性
listValue	否	String	生成选项的显示文字
headerKey	否	String	在所有的选项前再添加一个额外的选项作为其标题的 value 值
headerValue	否	String	显示在页面中 header 选项的内容

续表

属　性　名	是否必填	类　　型	说　　明
multiple	否	Boolean	指定是否多选，默认为 false
emptyOption	否	Boolean	是否在标题和真实的选项之间加一个空选项
size	否	Integer	下拉框的高度，即最多可以同时显示多少个选项

在表 20-10 的属性中，headerKey 和 headerValue 属性需要同时使用，使用时会在所有的真实选项之前加添一项作为标题项。如选择城市时，可以在所有的具体城市之前添加一项“请选择”，这一项不会作为备选的值。size 属性可以让下拉框同时显示多个值，multiple 属性让用户同时选择多个值，只是在后台的 Action 接收下拉框值时，不能使用 String 类型，而是使用 String[]或者 List<String>。

20.5.10　<s:optgroup>标签

<s:optgroup>标签用于生成选项组，通常作为<s:select>标签的子标签使用。由于在一个下拉列表框中可以包含多个选项组，所以<s:select>标签也能够包含多个<s:optgroup>标签。

使用<s:optgroup>标签时，也需要指定该标签的 list、listKey、listValue 等属性，这些属性的含义与<s:select>标签的相应属性含义相同。此外，<s:optgroup>标签的 label 属性表示选项组的组名，选项组的组名是不能被选中的。为了让读者更好地掌握<s:optgroup>标签，下面通过案例演示<s:optgroup>标签的使用。

在项目 chapter20 中创建一个名称为 optgroupTags.jsp 的文件，编辑后如文件 20-7 所示。

文件 20-7　OptgroupTags.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>optgroup 标签</title>
</head>
<body>
    <s:form>
        <s:select label="请选择所在城市" name="city"
            list="#{'BJ':'北京','SH':'上海'}"
            listKey="key" listValue="value">
        <s:optgroup label="河北"
```

```
            list="#{'BD':'保定','HS':'衡水','SJZ':'石家庄'}"
            listKey="key" listValue="value"/>
        <s:optgroup label="河南"
            list="#{'LY':'洛阳','KF':'开封','SQ':'商丘'}"
            listKey="key" listValue="value"/>
        </s:select>
    </s:form>
</body>
</html>
```

在文件 20-7 中，首先使用了<s:select>标签创建下拉列表框，然后在该标签中，使用了两种<s:optgroup>标签，得到两个城市选项组。

启动项目后，在浏览器的地址栏中输入地址 http://localhost:8080/chapter20/optgroupTags.jsp 访问 optgroupTags.jsp，浏览器的显示结果如图 20-8 所示。

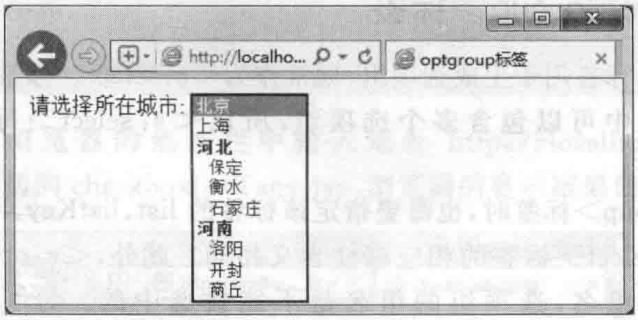


图 20-8 <s:optgroup>标签的使用

20.5.11 <s:file>标签

<s:file>标签用于创建一个文件选择框，它会生成 HTML 中的<input type="file" />标签。该标签常用的属性及其说明如表 20-11 所示。

表 20-11 file 标签常用的属性及说明

属性名	是否必填	类 型	说 明
name	否	String	指定表单元素名
accept	否	String	指定可接收的文件 MIME 类型，默认为 input

<s:file>标签的一般用法如下所示：

```
<s:file name="upLoadFile" accept="text/*"/>
<s:file name="otherUploadFile" accept="text/html,text/plain"/>
```

20.5.12 <s:hidden>标签

<s:hidden>标签用于创建隐藏表单元素，它会生成 HTML 中的隐藏域标签<input type="hidden" />。该标签在页面上没有任何显示，可用于保存或交换数据，如需要提交

表单时，要传递一个值到请求参数中，就可以使用该标签。

该标签的使用非常简单，通常只需要设置 name 和 value 属性即可，其使用语法如下所示：

```
<s:hidden name="id" value="%{id}"/>
```

20.6　Struts2 的非表单标签

Struts2 的非表单标签主要用于在页面中生成非表单的可视化元素，以及输出在 Action 中封装的信息，如输出错误提示信息等。常用的非表单标签有<s:actionmessage>、<s:actionerror>和<s:fielderror>标签，它们分别用于显示动作信息、动作错误信息和字段错误信息。如果信息为空，则不显示。各标签的具体功能介绍如下。

- <s:actionmessage>标签：如果 Action 实例的 getActionMessage()方法返回不为 null，则该标签负责输出该方法返回的系列消息。
- <s:actionerror>标签：如果 Action 实例的 getActionError()方法返回不为 null，则该标签负责输出该方法返回的系列错误。
- <s:fielderror>标签：如果 Action 实例存在表单域的类型转换错误和校验错误，则该标签负责输出这些错误提示。

为了让读者更好地了解这三个非表单标签的作用，下面通过具体的案例演示这三个非表单标签的使用。

1. 创建 Action

在 chapter20 项目的 src 目录下新建一个名称为 com.itheima.action 的包，并在包中新建一个名称为 MsgAction 的类，编辑后如文件 20-8 所示。

文件 20-8　MsgAction.java

```
1  package com.itheima.action;
2  import com.opensymphony.xwork2.ActionSupport;
3  public class MsgAction extends ActionSupport{
4      public String execute() throws Exception {
5          this.addActionMessage("<s:actionmessage>标签输出的内容");
6          this.addActionError("<s:actionerror>标签输出的内容");
7          this.addFieldError("msg", "<s:fielderror>标签输出的内容");
8          return SUCCESS;
9      }
10 }
```

在文件 20-8 中，分别使用了 addActionMessage、addActionError 和 addFieldError 三个方法输出错误信息。

2. 创建配置文件

在 src 目录下创建 Struts2 的配置文件 struts.xml，并添加 MsgAction 类的配置信息，其配置代码如下所示：

```
<action name="msg" class="com.itheima.action.MsgAction">
        <result>/non_formTags.jsp</result>
</action>
```

3. 创建显示页面

在项目的 WebContent 目录下创建一个名称为 non_formTags.jsp 的页面，编辑后如文件 20-9 所示。

文件 20-9　non_formTags.jsp

```
1  <%@page language="java" contentType="text/html;
2  charset=UTF-8" pageEncoding="UTF-8"%>
3  <%@taglib prefix="s" uri="/struts-tags"%>
4  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
5                        "http://www.w3.org/TR/html4/loose.dtd">
6  <html>
7  <head>
8  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
9  <title>errorTags</title>
10 </head>
11 <body>
12     <s:actionmessage />
13     <s:actionerror />
14      <s:fielderror />
15     <s:fielderror value="msg" /><!--有无 value 效果一样-->
16 </body>
17 </html>
```

4. 运行项目并查看结果

在浏览器的地址栏中输入地址 http://localhost:8080/chapter20/msg，浏览器的显示结果如图 20-9 所示。

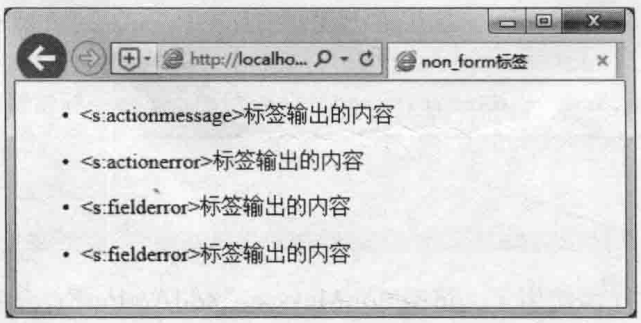


图 20-9　错误信息提示标签

从图 20-9 中可以看到，浏览器页面使用三个标签已经成功输出了 Action 中的信息，其中在<s:fielderror>标签中，有无属性 value 值的显示效果一样。

第 21 章

Struts2的OGNL和值栈

在 Struts2 框架中，OGNL 是其默认的表达式语言，它主要用于从框架的不同区域以一种一致的方式引入数据。同时，为了便于数据的统一管理，Struts2 框架还引入了值栈的概念。本章将针对 Struts2 框架中的 OGNL 和值栈进行详细讲解。

21.1 OGNL

21.1.1 OGNL 概述

OGNL 的全称是 Object-Graph Navigation Language，即对象图导航语言，它是一种功能强大的开源表达式语言。使用这种表达式语言可以通过某种表达式语法存取 Java 对象的任意属性，调用 Java 对象的方法，以及实现类型转换等。

OGNL 具有以下特点。

- 支持对象方法调用。如 objName. methodName()。
- 支持类静态方法调用和值访问，表达式的格式为@[类全名(包括包路径)]@[方法名|值名]。如@java. lang. String@format('fruit %s', 'frt')。
- 支持赋值操作和表达式串联。如 price=100，discount=0. 8，在方法 calculatePrice()中进行乘法计算会返回 80。
- 访问 OGNL 上下文(OGNL context)和 ActionContext。
- 操作集合对象。

了解 OGNL 的概念和特点后，下面讲解一下 OGNL 的组成。OGNL 的操作实际上是围绕 OGNL 结构的三个要素进行的，分别是表达式(expression)、上下文对象(context)和根对象(root)。这三个要素的具体说明如下。

1. 表达式

表达式是整个 OGNL 的核心，OGNL 会根据表达式到对象中取值。所有 OGNL 操作都是针对表达式解析后进行的，它表明了此次 OGNL 操作要“做什么”。实际上，表达式就是一个带有语法含义的字符串，这个字符串规定了操作的类型和操作的内容。

2. 上下文对象

上下文对象规定了 OGNL 操作“在哪里进行”。context 对象是一个 Map 类型的对象，在表达式中访问 context 中的对象，需要使用＃号加对象名称，即“＃对象名称”的形式。例如要获取 context 对象中 user 对象的 username 值，可以如下书写：

```
#user.username
```

3. 根对象

根对象可以理解为 OGNL 的操作对象，OGNL 可以对根对象进行取值或写值等操作，表达式规定了“做什么”，而根对象则规定了“对谁操作”。实际上根对象所在的环境就是 OGNL 的上下文对象环境。

21.1.2 OGNL 表达式

在实际开发过程中，OGNL 表达式要结合 struts2 的标签使用，主要就是%、#和$ 三个符号的使用。由于 $ 被广泛应用于 EL 中，所以本小节将重点介绍%和#的用法。

1. #符号

在 Struts2 框架中，#符号有三种用途，分别如下：

(1) 访问非根对象的属性，如访问 OGNL 上下文和 Action 上下文。由于 Struts2 中值栈被视为根对象，所以访问其他非根对象时，需要加 # 前缀。# 相当于 ActionContext. getContext()，例如 # session. user 表达式相当于 ActionContext. getContext(). getSession(). getAttribute("user")，# request. userName 表达式相当于 request. getAttribute("userName")。

(2) 用于过滤和投影集合，如 books. {? #this. price>25}。

(3) 构造 Map。如 #{key1:value1,key2:value2}，这种方式常用于给 radio 或 select、checkbox 等标签赋值。如果要在页面中取一个 Map 的值可以如下书写：<s:property value="#myMap['key']"/>。

2. %符号

%是在标签的属性值被理解为字符串类型时，告诉执行环境%{}中的是 OGNL 表达式，并计算 OGNL 表达式的值。

3. $ 符号

$ 符号主要用于在 Struts2 配置文件中引入 OGNL 表达式。例如下面的配置所示：

```
<action name="userAction_*" class="userAction" method="{1}">
    <result name="add" type="redirect">
        userAction_findById? userId=${User.userId}
    </result>
</action>
```

为了让读者更好地理解和掌握 OGNL，下面通过一个案例演示 OGNL 的使用。

1. 创建项目

在 Eclipse 中创建一个名称为 chapter21 的 Web 项目，并引入 Struts2 所需 JAR 包，然后添加上过滤器信息。

2. 创建实体类

在项目的 src 目录下创建一个名称为 com. itheima. ognl 的包，在包中新建一个名称为

Person 的类，该类中包含 name、age 和 birthday 三个属性以及其 getter 和 setter 方法，如文件 21-1 所示。

文件 21-1　Person.java

```
1  package com.itheima.ognl;
2  import java.util.Date;
3  public class Person {
4      private String name;
5      private int age;
6      private Date birthday;
7      public String getName() {
8          return name;
9      }
10     public void setName(String name) {
11         this.name=name;
12     }
13     public int getAge() {
14         return age;
15     }
16     public void setAge(int age) {
17         this.age=age;
18     }
19     public Date getBirthday() {
20         return birthday;
21     }
22     public void setBirthday(Date birthday) {
23         this.birthday=birthday;
24     }
25 }
```

3. 创建 Action

在 com.itheima.ognl 包中创建一个名称为 OgnlAction 的 Action 类，编辑后如文件 21-2 所示。

文件 21-2　OgnlAction.java

```
1  package com.itheima.ognl;
2  import java.util.Date;
3  import java.util.LinkedList;
4  import java.util.List;
5  import javax.servlet.http.HttpServletRequest;
6  import org.apache.struts2.ServletActionContext;
7  import com.opensymphony.xwork2.ActionContext;
8  import com.opensymphony.xwork2.ActionSupport;
9  public class OgnlAction extends ActionSupport{
```

```
10      private List<Person>persons;                    //声明 Person 对象
11      public List<Person>getPersons() {
12          return persons;
13      }
14      public void setPersons(List<Person>persons) {
15          this.persons=persons;
16      }
17      //测试方法
18      public String ognlTest() throws Exception {
19          //获得 ActionContext 实例
20          ActionContext ctx=ActionContext.getContext();
21          ctx.getApplication().put("msg", "application 信息"); //存入 application
22          ctx.getSession().put("msg", "seesion 信息");        //保存 session
23          HttpServletRequest request=ServletActionContext.getRequest();
24          //保存到 request
25          request.setAttribute("msg", "request 信息");
26          persons=new LinkedList<Person>();                   //为 persons 赋值
27          Person person1=new Person();                        //person1
28          person1.setName("AAA");
29          person1.setAge(26);
30          person1.setBirthday(new Date());
31          persons.add(person1);
32          Person person2=new Person();                        //person2
33          person2.setName("BBB");
34          person2.setAge(36);
35          person2.setBirthday(new Date());
36          persons.add(person2);
37          Person person3=new Person();                        //person3
38          person3.setName("CCC");
39          person3.setAge(16);
40          person3.setBirthday(new Date());
41          persons.add(person3);
42          return SUCCESS;
43      }
44  }
```

在文件 21-2 中，首先声明了 Person 对象，并实现了它的 getter 和 setter 方法。然后在测试方法 ognlTest()中，首先获得了 ActionContext 实例，然后通过 ActionContext 的实例分别向 Application、Session 和 Request 中存入相应信息，最后创建了 Person 对象，并向 Person 对象中添加了三个人员信息。

4. 创建配置文件

在项目的 src 目录下创建配置文件 struts. xml，并配置 Action 的信息。编辑后如文

件 21-3 所示。

文件 21-3　struts. xml

```
<? xml version="1.0" encoding="UTF-8" ? >
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
        "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
    <package name="struts2" namespace="/" extends="struts-default">
        <action name="ognl" class="com.itheima.ognl.OgnlAction"
        method="ognlTest">
            <result>/ognl.jsp</result>
        </action>
    </package>
</struts>
```

5. 创建显示页面文件

在项目的 WebContent 目录下创建一个名称为 ognl. jsp 的文件，编辑代码后如文件 21-4 所示。

文件 21-4　ognl. jsp

```
<%@page language="java" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>OGNL 的使用</title>
 </head>
 <body>
    <h3>访问 OGNL 上下文和 Action 上下文</h3>
    <!--使用 OGNL 访问属性值-->
    <p>request.msg: <s:property value="#request.msg" /></p>
    <p>session.msg: <s:property value="#session.msg" /></p>
    <p>application.msg: <s:property value="#application.msg" /></p>
    <p>attr.msg: <s:property value="#attr.msg" /></p>
    <hr />
    <h3>用于过滤和投影集合</h3>
    <p>年龄大于 20</p>
    <ul>
        <!--判断年龄-->
        <s:iterator value="persons.{? #this.age>20}">
          <li><s:property value="name" />-年龄: <s:property value="age" /></li>
```

```
        </s:iterator>
    </ul>
    <p>姓名为 AAA 的年龄:
    <s:property value="persons.{? #this.name=='AAA'}.{age}[0]"/></p>
    <hr />
    <h3>构造 Map</h3>
    <s:set name="foobar" value="#{'foo1':'bar1', 'foo2':'bar2'}" />
    <p>The value of key "foo1" is <s:property value="#foobar['foo1']" /></p>
    <hr />
    <h3>%符号的用法</h3>
    <s:set name="key" value="#{'key1':'value1', 'key2':'value2'}" />
    <p>key1 的值是 <s:property value="#key['key1']" /></p>
    <p>不使用%: <s:url value="#key['key1']" /></p>
    <p>使用%: <s:url value="%{#key['key1']}" /></p>
    <hr />
    <%
      request.setAttribute("req", "request scope");
      request.getSession().setAttribute("sess", "session scope");
      request.getSession().getServletContext().setAttribute
      ("app","aplication scope");
    %>
    <h3>通过 OGNL 表达式获取属性范围中的值</h3><br />
    <s:property value="#request.req" /><br />
    <s:property value="#session.sess" /><br />
    <s:property value="#application.app" /><br />
</body>
</html>
```

6. 运行项目并查看结果

启动服务器后,在浏览器的地址栏中输入地址 http://localhost:8080/chapter21/ognl,浏览器的显示结果如图 21-1 所示。

从图 21-1 中可以看出,使用 OGNL 表达式已经成功获取了所需要的信息。OGNL 表达式简单、易懂、功能强大,读者可以访问其官方网站 www.ognl.org 获取更多相关资料。

21.1.3 使用 OGNL 访问对象方法和静态方法

除了在 JSP 页面中可以使用 OGNL 表达式以外,OGNL 还支持在 Java 代码中访问对象方法和静态方法,接下来本节中将讲解 OGNL 是如何调用方法的。

1. OGNL 访问对象方法

OGNL 表达式通过“对象名.方法名()”的形式调用对象方法表达式,如果是调用根对象的方法,则可以直接使用方法的名称调用,其调用语法如下所示:

```
Ognl.getValue("方法名", 对象名);
```

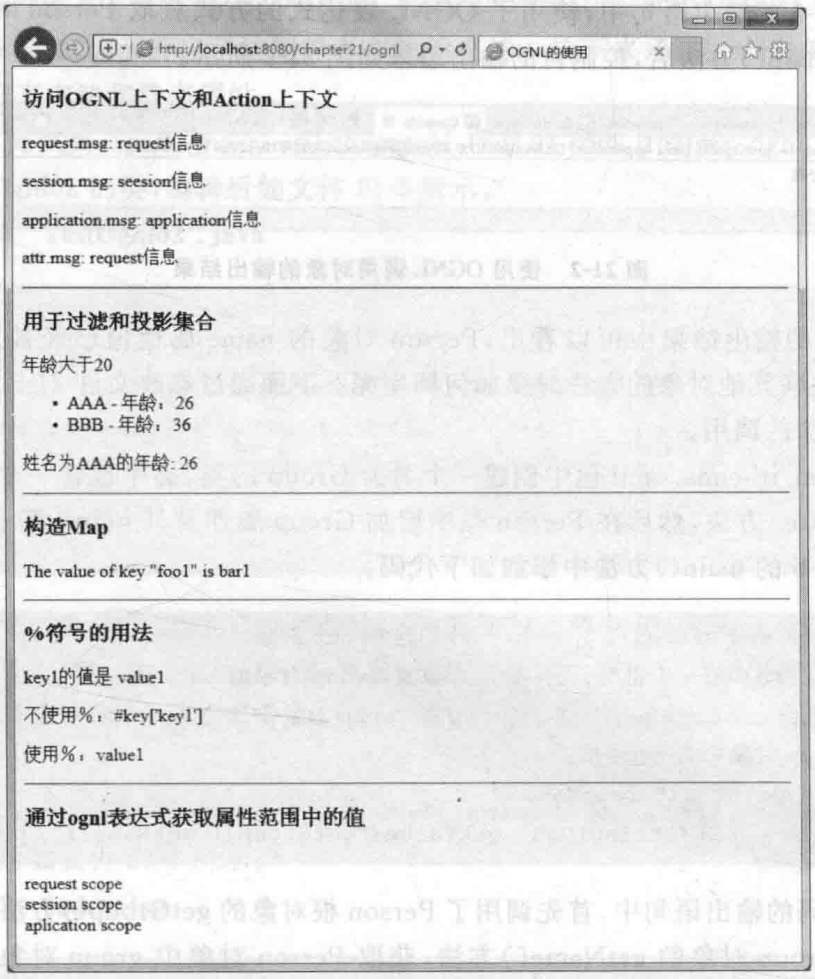


图 21-1　OGNL 的使用

其实无论 OGNL 表达式以哪种方式取值或者调用方法，最终都是调用其本身的 getValue()方法。接下来通过一个案例演示 OGNL 如何在 Java 代码中调用对象方法。

在 chapter21 项目的 com.itheima.ognl 包中创建一个名为 TestOgnl01 的类，编辑后如文件 21-5 所示。

文件 21-5　TestOgnl01.java

```
1  package com.itheima.ognl;
2  import ognl.Ognl;
3  import ognl.OgnlException;
4  public class TestOgnl01 {
5      public static void main(String[] args) throws OgnlException {
6          Person person=new Person();
7          person.setName("小韩");
8          //使用 OGNL 表达式获取 Person 对象的 name 值
9          System.out.println("name="+(String) Ognl.getValue("getName()", person));
10     }
11 }
```

在文件 21-5 的输出语句中，使用了 OGNL 表达式的方式获取 Person 对象中的 name 属性值，运行 main()方法后，控制台的输出结果如图 21-2 所示。

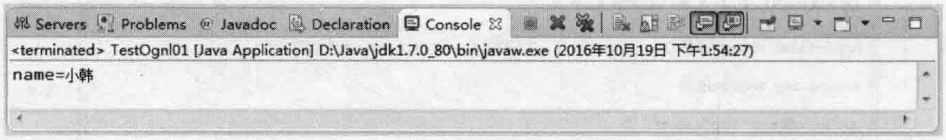


图 21-2 使用 OGNL 调用对象的输出结果

从图 21-2 的输出结果中可以看出，Person 对象的 name 属性值已经被成功取出。那么，根对象在关联其他对象的方法时要如何调用呢？下面通过修改文件 21-5 演示根对象关联其他对象方法的调用。

首先在 com.itheima.ognl 包中创建一个名为 Group 的类，类中包含一个 name 属性及其 getter 和 setter 方法，然后在 Person 类中增加 Group 属性及其 getter 和 setter 方法，接下来在文件 21-5 的 main()方法中添加如下代码：

```
Group group=new Group();                    //创建 Group 对象
group.setName("第一小组");                   //设置 Group 对象的 name 值
person.setGroup(group);                     //将 Group 对象添加到 Pserson 对象中
//输出 Group 对象中的 name 值
System.out.println("groupName="  .
                +(String)Ognl.getValue("getGroup().getName()", person));
```

在上述代码的输出语句中，首先调用了 Person 根对象的 getGroup()方法得到 group 对象，然后调用 group 对象的 getName()方法，获取 Person 对象中 group 对象的 name 属性值。运行 TestOgnl01 的 main()方法后，控制台的输出结果如图 21-3 所示。

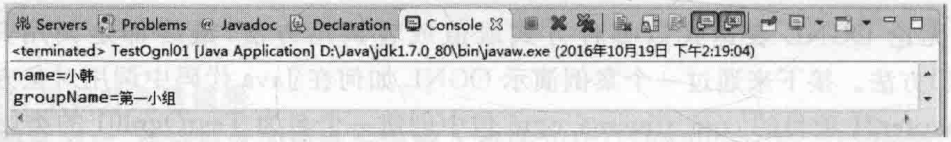


图 21-3 根对象关联其他对象方法时的输出结果

2. OGNL 访问静态方法和静态属性

OGNL 同时支持静态方法和静态属性的调用，其语法格式如下所示：

```
@类的全路径名@属性名称
@类的全路径名@方法名称(参数列表)
```

需要注意的是，在低版本的 Struts2 中，已经默认开启了对访问类静态方法的支持，但是高版本的 Struts2 默认是关闭这项支持的，也就是说，要想访问类的静态方法，需要手动开启设置。开启设置的方法十分简单，只需要在 struts.xml 中进行如下配置即可。

```
<constant name="struts.ognl.allowStaticMethodAccess" value="true" />
```

在上述代码配置了一个名称为 struts. ognl. allowStaticMethodAccess 的常量，并将其值设置为 true，这就表示开启了访问静态方法的支持。接下来通过一个案例演示如何使用 OGNL 访问静态方法和静态属性。

(1) 创建测试类 TestOgnl02。在 chapter21 项目的 com. itheima. ognl 包中，创建一个名称为 TestOgnl02 的类，编辑后如文件 21-6 所示。

文件 21-6　TestOgnl02 . java

```
1  package com.itheima.ognl;
2  public class TestOgnl02 {
3      public static String staticValue="这是静态属性值";
4      public static void testMethod() {
5          System.out.println("这是静态方法 ");
6      }
7  }
```

在文件 21-6 中，首先创建了一个 String 类型的静态属性 staticValue，并将其属性值设为“这是静态属性值”，然后创建了一个静态方法，并在方法中使用输出语句打印信息。

(2) 修改配置文件。在配置文件 struts. xml 中，添加用于开启对静态方法访问的支持代码。

(3) 创建 index. jsp 页面。在项目的 WebContent 目录下，创建一个名称为 index. jsp 的页面，编辑后如文件 21-7 所示。

文件 21-7　index. jsp

```
1  <%@page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3  <%@taglib prefix="s" uri="/struts-tags"%>
4  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
5                        "http://www.w3.org/TR/html4/loose.dtd">
6  <html>
7  <head>
8  <title>OGNL 访问静态属性和静态方法</title>
9  </head>
10 <body>
11     获取的静态属性值为:
12     <s:property value="@com.itheima.ognl.TestOgnl02@staticValue" /><br />
13     <hr>
14     调用静态方法的结果请查看控制台
15     <s:property value="@com.itheima.ognl.TestOgnl02@testMethod()" />
16 </body>
17 </html>
```

在文件 21-7 中，首先引入了 Struts2 的标签库，然后在其＜body＞元素内，分别使用 OGNL 获取了 TestOgnl02 中定义的静态属性值和静态方法。

(4) 启动项目并查看结果。启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter21/index.jsp 访问 index.jsp，浏览器的输出结果如图 21-4 所示。

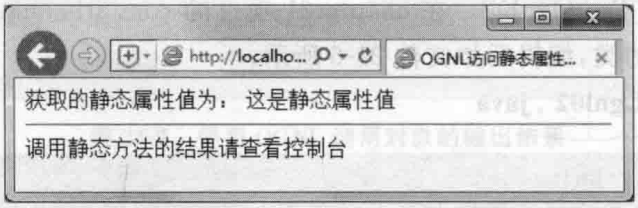


图 21-4 使用 OGNL 访问静态属性和静态方法的浏览器输出

Eclipse 控制台的输出结果如图 21-5 所示。

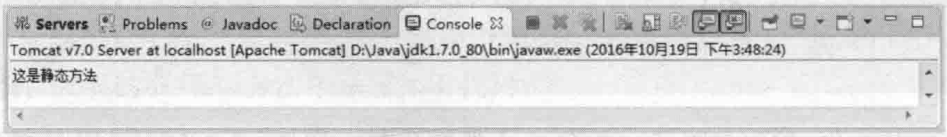


图 21-5 使用 OGNL 访问静态属性和静态方法的控制台输出

从图 21-4 和图 21-5 的显示结果中可以看出，已经成功在页面中使用 OGNL 访问 Java 类中的静态属性和静态方法。

21.2 值栈

21.2.1 值栈的定义

值栈(ValueStack)就是 OGNL 表达式存取数据的地方。在一个值栈中，封装了一次请求所需要的所有数据。在使用 Struts2 的项目中，Struts2 会为每个请求创建一个新的值栈，也就是说，值栈和请求是一一对应的关系，这种一一对应的关系使值栈能够线程安全地为每个请求提供公共的数据存取服务。

1. 值栈的作用

值栈可以作为一个数据中转站在前台与后台之间传递数据，最常见的就是将 Struts2 的标签与 OGNL 表达式结合使用。值栈实际上是一个接口，在 Struts2 中利用 OGNL 时，实际上使用的就是实现了该接口的 OgnlValueStack 类，这个类是 OGNL 的基础。

2. 值栈的生命周期

值栈贯穿整个 Action 的生命周期，每个 Action 类的对象实例都拥有一个 ValueStack 对象，在 ValueStack 对象中保存了当前 Action 对象和其他相关对象。Struts2 框架把 ValueStack 对象保存在一个名为 struts.valueStack 的 request 属性中，也就是说，值栈与 Action 的生命周期一致。值栈的生命周期随着 request 的创建而创建，随着 request 的销毁而销毁。

3. 值栈的获取方式

要获取值栈中存储的数据，首先应该获取值栈。值栈的获取有两种方式，具体如下。

(1) 在 request 中获取值栈

ValueStack 对象在 request 范围内的存储方式为 request. setAttribute("struts. valueStack",valuestack),可以通过如下方式从 request 中取出值栈的信息。

```
//获取 ValueStack 对象,通过 request 对象获取
ValueStack valueStack=(ValueStack)ServletActionContext.getRequest()
            .getAttribute(ServletActionContext.STRUTS_VALUESTACK_KEY);
```

在上述示例代码中,ServletActionContext. STRUTS _ VALUESTACK _ KEY 是 ServletActionContext 类中的常量,它的值为 struts. valueStack。

(2) 在 ActionContext 中获取值栈

在使用 Struts2 框架时,可以使用 OGNL 操作 Context 对象从 ValueStack 中存取数据,也就是说,可以从 Context 对象中获取 ValueStack 对象。实际上,Struts2 框架中的 Context 对象就是 ActionContext。ActionContext 获取 ValueStack 对象的方式如下所示:

```
//通过 ActionContext 获取 valueStack 对象
ValueStack valueStack=ActionContext.getContext().getValueStack();
```

ActionContext 对象是在 StrutsPrepareAndExcuteFilter 的 doFilter()方法中被创建的,在源码中用于创建 ActionContext 对象的 createActionContext()方法内可以找到获取的 ValueStack 对象的信息,方法中还有这样一段代码:ctx = new ActionContext(stack. getContext()),从代码中可以看出,ValueStack 对象中的 Context 对象被作为参数传递给了 ActionContext 对象,这也就说明 ActionContext 对象中持有了 ValueStack 对象的引用,因此可以通过 ActionContext 对象获取 ValueStack 对象。

21.2.2　值栈的内部结构

通过第 21.2.1 节内容的学习,读者对值栈的知识已经有了一定的了解。为了让初学者更好地理解值栈,本节将讲解值栈的内部结构。

ValueStack 对象的内部有两个逻辑部分。

- ObjectStack(对象栈):是 CompoundRoot 类型,用 ArrayList 定义,Struts2 把动作和相关对象压入 ObjectStack 中。
- ContextMap(Map 栈):是 OgnlContext 类型,是一个 Map 集合,Struts2 把各种各样的映射关系(一些 Map 类型的对象)压入 ContextMap 中。

了解了 ValueStack 对象的内部组成后,下面通过运行项目查看 ValueStack 中的信息。

首先,在 chapter21 项目的 com. itheima. ognl 包中新创建一个名称为 ValueStackAction 的类,并在类中编写一个获取 ValueStack 对象的方法,如文件 21-8 所示。

文件 21-8　ValueStackAction. java

```
1  package com.itheima.ognl;
2  import com.opensymphony.xwork2.ActionContext;
```

```
3  import com.opensymphony.xwork2.ActionSupport;
4  import com.opensymphony.xwork2.util.ValueStack;
5  public class ValueStackAction extends ActionSupport {
6      public String execute() throws Exception {
7      //通过 ActionContext 获取 valueStack 对象
8      ValueStack valueStack=ActionContext.getContext().getValueStack();
9          System.out.println(valueStack);
10         return SUCCESS;
11     }
12 }
```

然后，将新建的 Action 信息添加到 struts.xml 中，所添加的代码如下所示：

```
<action name="valueStack" class="com.itheima.ognl.ValueStackAction">
    <result name="success">index.jsp</result>
</action>
```

最后，在文件 21-8 中的第 9 行处设置断点，以 Debug 模式启动项目。在浏览器的地址栏中输入地址 http://localhost:8080/chapter21/valueStack.action 后，Eclipse 会进入 Debug 模式。从 Variables 窗口中可以看到 valueStack 的结构信息，如图 21-6 所示。

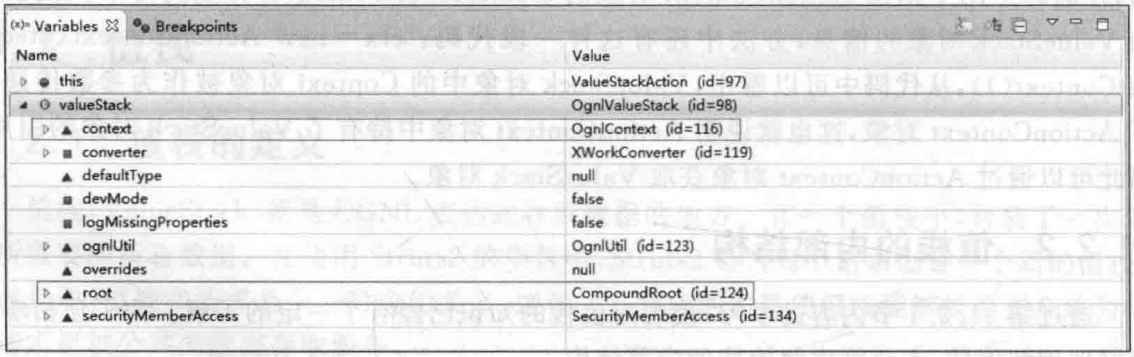


图 21-6 valueStack 结构

在图 21-6 所示的 ValueStack 结构中，只需关注 context 对象和 root 对象。从图 21-6 中可以看到，context 对象的类型为 OgnlContext，root 对象的类型为 CompoundRoot。如果要查看这两个类的源码，则可以看到如下语句：

```
OgnlContext extends Object implements Map
CompoundRoot extends ArrayList
```

从上述两个类的源码中可以看出，context 对象实际上就是一个 Map，root 对象实际上就是一个 ArrayList。也就说明了 ValueStack 的两个逻辑部分 ObjectStack 对应 ArrayList(root)，ContextMap 对应 Map(context)。

一般情况下，root 对象在存储 Action 的相关信息时会把相关的映射压入 ContextMap 中，这些相关的映射具体如下。

- parameters：该 Map 中包含当前请求的请求参数。
- request：该 Map 中包含当前 request 对象中的所有属性。
- session：该 Map 中包含当前 session 对象中的所有属性。
- application：该 Map 中包含当前 application 对象中的所有属性。
- attr：该 Map 按如下顺序检索某个属性：request,session,application。

需要注意的是，在 context 对象中包含了 root 对象，从 context 对象的结构中就可以查看到相关信息，具体如图 21-7 所示。

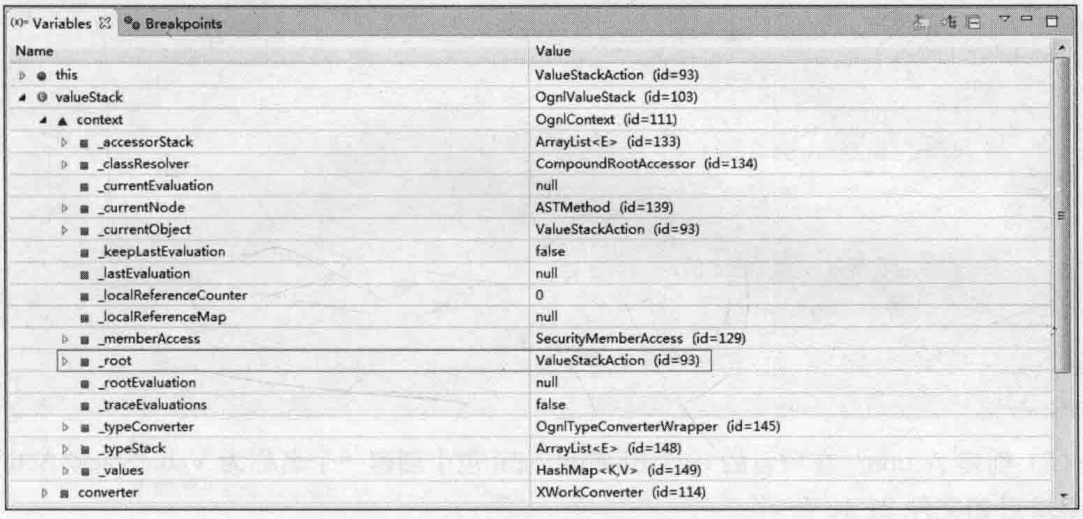


图 21-7　context 对象结构

从图 21-7 中可以看到 context 对象中的 root 对象，这也就证明了 context 对象中包含了 root 对象，并且该对象的类型为 CompoundRoot。

21.2.3　值栈在开发中的应用

在 Struts2 中，值栈的主要作用就是解决从 Action 到页面的数据交换问题。在采用属性驱动和模型驱动交换数据时，Struts2 会将对象自动存储到 ValueStack 中，其存储说明如下。

- 属性驱动：每次请求访问 Action 的对象时，Action 中的属性对象会被自动压入 ValueStack 中。
- 模型驱动：Action 如果实现了 ModelDriven 接口，那么 ModelDrivenInterceptor 拦截器会生效，会将 model 对象压入到 ValueStack 中。

属性对象或 model 对象存储到 ValueStack 中后，就可以直接从 ValueStack 中获取页面所需的数据。下面通过具体案例演示以上两种从 ValueStack 中获取数据的方式，具体如下：

1. 属性驱动

(1) 创建实体类 User。在 chapter21 项目的 src 下创建包 com.itheima.domain，在该包下创建一个名称为 User 的类，该类中包含 name 和 age 两个属性，以及其 getter 和 setter 方法，如文件 21-9 所示。

文件 21-9 Product.java

```
1  package com.itheima.domain;
2  public class User {
3      private String name;
4      private int age;
5      public String getName() {
6          return name;
7      }
8      public void setName(String name) {
9          this.name=name;
10     }
11     public int getAge() {
12         return age;
13     }
14     public void setAge(int age) {
15         this.age=age;
16     }
17 }
```

(2) 创建 Action。在项目的 com.itheima.ognl 包中创建一个名称为 ValueStack2Action 的类，编辑后如文件 21-10 所示。

文件 21-10 ValueStack2Action.java

```
1  package com.itheima.ognl;
2  import com.itheima.domain.User;
3  import com.opensymphony.xwork2.ActionSupport;
4  public class ValueStack2Action extends ActionSupport {
5      private static final long serialVersionUID=1L;
6      private User user;
7      public User getUser() {
8          return user;
9      }
10     public void setUser(User user) {
11         this.user=user;
12     }
13     public String execute() throws Exception {
14         return SUCCESS;
15     }
16 }
```

在文件 21-10 中，声明了一个 User 类型的属性 user 及其 getter 和 setter 方法，由于只进行存取值的操作，所以直接使用 Action 的 execute()方法返回 SUCCESS 即可。

(3) 修改配置文件。在配置文件 struts.xml 的<package>元素中，增加 action 的配置信息，具体如下所示：

```
<action name="valueStack2" class="com.itheima.ognl.ValueStack2Action">
    <result>/field.jsp</result>
</action>
```

(4) 创建页面文件。在项目的 WebContent 目录下创建一个名称为 field.jsp 的文件，编辑后如文件 21-11 所示。

文件 21-11　field.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>属性驱动方式获取值栈中的数据</title>
</head>
<body>
    <s:debug></s:debug>
    姓名：<s:property value="user.name" /><br />
    年龄：<s:property value="user.age" />
</body>
</html>
```

在文件 21-11 中，首先编写了一个<s:debug>标签，用于查看值栈的信息，然后分别使用<s:property>标签获取用户的姓名和年龄。

(5) 启动项目并查看结果。启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter21/valueStack2?user.name=Tom&user.age=15 后，浏览器的显示结果如图 21-8 所示。

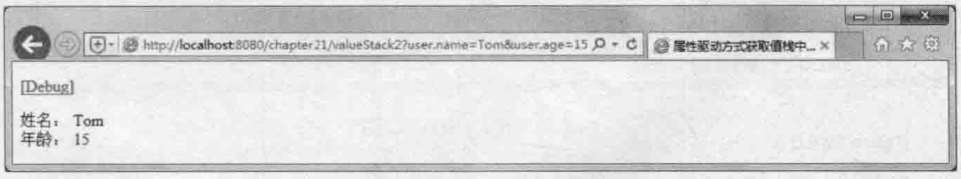


图 21-8　访问结果

从图 21-8 中可以看出，从请求地址中传入的参数值已经成功地在页面中显示出来，这就是使用属性驱动获取 ValueStack 中对象的方式。User 对象传入 Action 对象时，会被自动压入 ValueStack 中，所以在 Action 的 execute()方法中不存储值也能在页面取出。此时单击 Debug 标签链接便可以查看到值栈中的参数传递情况，如图 21-9 所示。

从图 21-9 的参数列表中可以到 Property Name 列表中的参数 user 就是页面中获取的对象，在 Property Value 中包含了 user 的属性值。

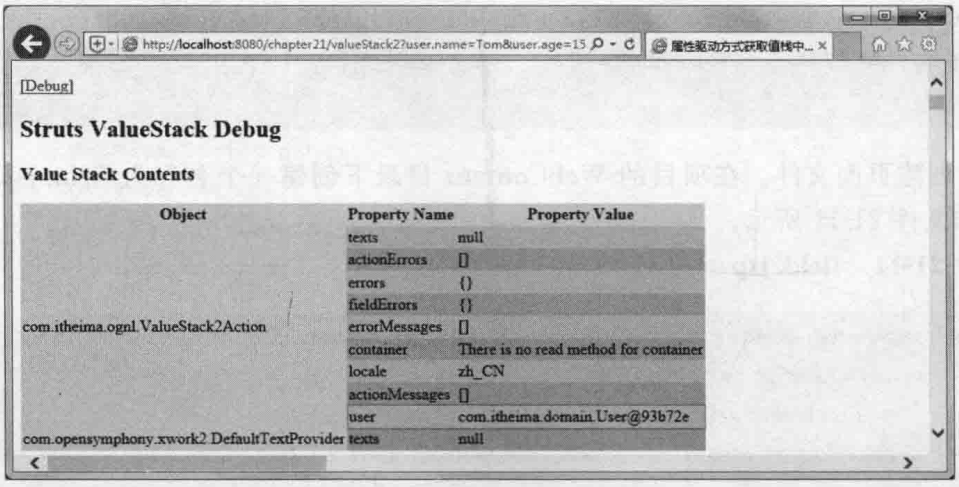


图 21-9 参数列表

2. 模型驱动

采用模型驱动的方式时，Struts2 会将 Action 中的 model 信息存入到值栈中。下面通过具体案例演示使用模型驱动时获取 ValueStack 中的值的方式。

(1) 创建 Action。在 chapter21 项目的 com.itheima.ognl 包中创建一个名称为 ValueStack3Action 的类，编辑后如文件 21-12 所示。

文件 21-12 ValueStack3Action.java

```
1  package com.itheima.ognl;
2  import com.itheima.domain.User;
3  import com.opensymphony.xwork2.ActionSupport;
4  import com.opensymphony.xwork2.ModelDriven;
5  public class ValueStack3Action extends ActionSupport
6                                  implements ModelDriven<User>{
7      private static final long serialVersionUID=1L;
8      private User user=new User();
9      public User getModel() {
10         return user;
11     }
12     @Override
13     public String execute() throws Exception {
14         return super.execute();
15     }
16 }
```

在文件 21-12 中，创建了一个 User 类型的对象 user，实现了 ModelDriven 接口的 getModel()方法，并在 getModel()方法中返回 user 对象，这样就可以接收到和 user 对象中属性名称相同的参数的值。

(2) 修改配置文件。在配置文件 struts.xml 的<package>元素中，增加 action 的配置信息，具体如下所示：

```
<action name="valueStack3" class="com.itheima.ognl.ValueStack3Action">
    <result>/model.jsp</result>
</action>
```

(3) 创建页面文件。在项目的 WebContent 目录下创建一个名称为 model.jsp 的页面，编辑后如文件 21-13 所示。

文件 21-13　model.jsp

```
<%@page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                      "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>模型驱动方式获取值栈中的数据</title>
</head>
<body>
    <s:debug></s:debug>
    姓名：<s:property value="model.name" /><br />
    年龄：<s:property value="model.age" />
</body>
</html>
```

(4) 启动项目并查看结果。启动 Tomcat 服务器，在浏览器的地址栏中输入地址 http://localhost:8080/chapter21/valueStack3? name＝Tom&age＝15 后，浏览器的显示结果如图 21-10 所示。

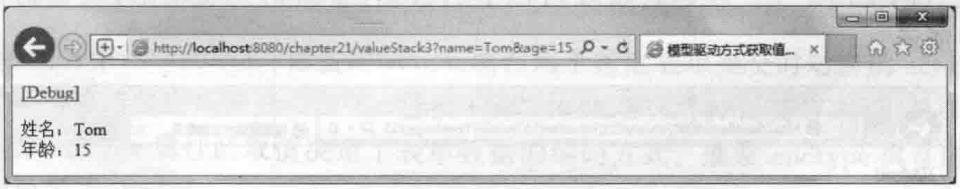


图 21-10　访问结果

从图 21-10 中可以看到，从请求中传入的参数 name 和 age 已经成功地在页面中显示出来。由于是模型驱动，所以不需要用 user.name 的形式传递参数。接下来单击 Debug 链接查看 ValueStack 中存储的参数值，如图 21-11 所示。

从图 21-11 中可以看出，采用模型驱动的方式时，User 对象的信息被存储在值栈的栈顶。由于 Action 中提供了一个 getModel()方法，所以使用模型驱动时可以从 Action 中获取 model 对象，而在传递参数时，直接用 name 和 age 即可，这就是采用模型驱动获取值栈中数据的方式。

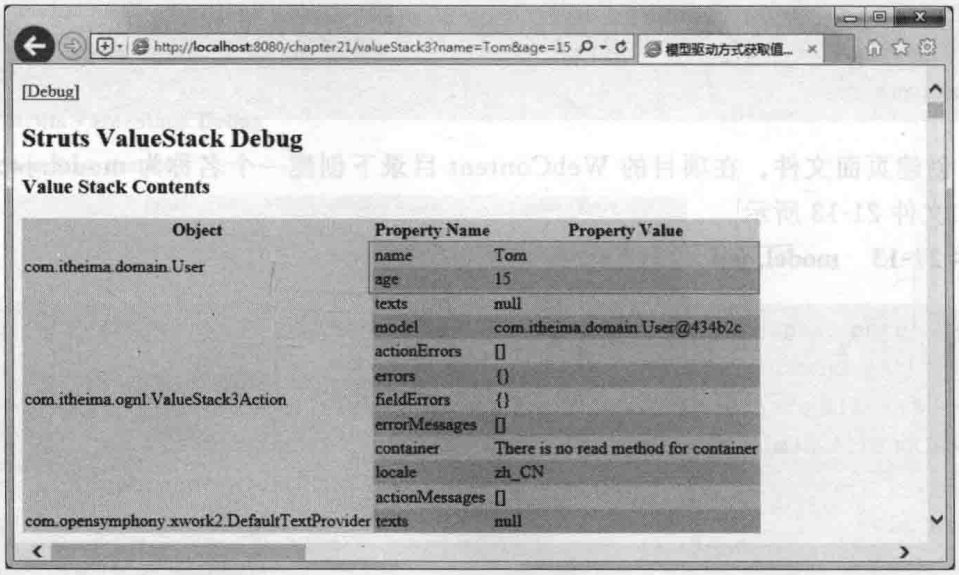


图 21-11 参数列表

21.2.4 通过 EL 访问值栈的数据

在 Struts2 框架中，使用 EL 表达式也可以直接访问值栈中的数据。在文件 21-13 中，添加如下代码：

```
<hr>
使用 EL 表达式访问值栈中的数据：<br />
姓名：${model.name}<br />
年龄：${model.age}
```

上述代码使用了 EL 表达式获取值栈中的数据，重新访问页面后，浏览器的显示如图 21-12 所示。

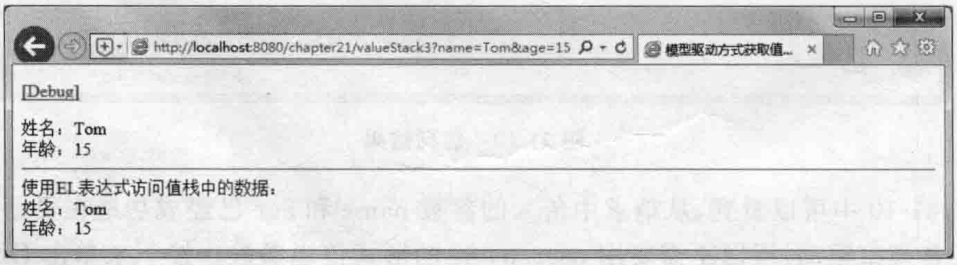


图 21-12 访问结果

从图 21-12 中可以看出，直接用 EL 表达式的方式同样取出了值栈中的数据。EL 表达式能够访问值栈中数据的原因，是因为 Struts2 框架对请求进行了增强，如果在 request 作用域中没有查找到对应的值，就会到值栈中查找。查找时，它会先被当作属性在 root 中查找，查找不到再被当作 key 在 contextMap 中查找对应的 value，查找到后会将数据取出。

第 22 章

Struts2的文件上传和下载

在互联网应用中，文件的上传和下载是常用的功能，最常用的就是将照片上传到空间相册或者在音乐网站中下载音乐等。在 Struts2 框架中，同样提供了对文件上传和下载功能的支持。本章将对 Struts2 框架中的文件上传和下载功能进行详细讲解。

22.1 文件上传

22.1.1 文件上传的概述

文件上传是指将本地文件上传到服务器的指定目录下。使用 Struts2 框架进行文件上传时，首先需要对 form 表单进行设置，将其 enctype 属性设置为 multipart/form-data，method 属性设置为 post，并在<s:form>标签中使用<s:file>标签作为文件选择框上传文件。文件上传页面的示例代码如下所示：

```
<s:form action="xxx" method="post" enctype="multipart/form-data">
    <s:file name="xxx" label="xxx" />
    <s:submit value="上传" />
</s:form>
```

在上述代码中，<s:form>标签的 action 属性用于指定表单提交时对应的 action 方法，method 属性用于指定表单的提交方式(包括 GET 和 POST 两种提交方式，默认提交方式是 GET)，而 enctype 属性的取值决定了表单数据的编码方式。通常 enctype 属性有三个属性值，具体如下。

- application/x-www-form-urlencoded：该值为默认的编码方式，它只处理表单域中的 value 属性，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。这种编码方式按 ISO-8859-1 的编码方式将表单上传到服务器，只是此种编码方式上传不了文件。
- multipart/form-data：这种编码方式会以二进制流的方式处理表单数据，并且它会把文件域指定文件的内容也封装到请求参数中。在进行文件上传时，采用的就是此种方式。
- text/plain：主要作用是直接通过表单发送邮件，目前已不再采用。

设置完 form 表单后，还需要在 Action 类中编写相应的代码。文件上传 Action 类的示例代码如下所示：

```
public class FileUploadAction extends ActionSupport{
    private File xxx;                          //用户上传的文件
    private String xxxFileName;                //上传文件的文件名
    private String xxxContentType;             //上传文件的类型
    //此处省略三个属性的 getter 和 setter 方法
    public String execute() throws Exception
    {
        //此处省略具体执行的方法
        return SUCCESS;
    }
}
```

在上述代码中，包括三个类型的属性，具体如下。

- File 类型的属性 xxx：该属性用于指定上传文件的内容。这里将该属性假设指定为 xxx。
- String 类型的属性 xxxFileName：该属性名称必须为 xxxFileName，其中 xxx 表示 File 类型的属性名称，该属性用于指定上传文件名。
- String 类型的属性 xxxContentType：该属性名称必须为 xxxContentType，其中 xxx 表示 File 类型的属性名称，该属性用于指定上传文件的文件类型。

需要注意的是，上述代码中的 xxx 指 JSP 页面中 file 标签的名字。

最后，在 struts.xml 中需要对 Action 进行配置，其配置方式与其他 Action 的配置相似，只是一般要在 Action 的配置中加入文件过滤拦截器 fileUpload。fileUpload 拦截器在 struts-default 中已经配置，使用时需要重新配置。struts.xml 配置信息的示例代码如下所示：

```
<package name="struts2" namespace="/" extends="struts-default">
<action name="Xxx" class="com.itheima.Xxx">
    <result name="success">/result.jsp</result>     <!--返回结果-->
    <interceptor-ref name="defaultStack">           <!--引用拦截器-->
        <!--限制上传文件最大值为 4M-->
        <param name="fileUpload.maximumSize">4194304</param>
        <!--限制上传文件的扩展名-->
        <param name="fileUpload.allowedExtensions">.txt,.doc,.jpg</param>
        <!--限制上传文件的类型-->
        <param name="fileUpload.allowedTypes">
            text/plain,application/msword,image/jpeg
        </param>
    </interceptor-ref>
</action>
</package>
```

在上述配置代码中，使用了 fileUpload 的 maximumSize、allowedExtension 和 allowedTypes 三个参数，分别对上传文件进行了限制。其中 maximumSize 用于指定允许上

传文件的文件大小，单位是字节；allowedExtension 用于指定上传文件的扩展名；allowedTypes 用于指定允许上传文件的类型，当有多个文件类型时，用逗号隔开即可。

22.1.2　单个文件上传

通过第 22.1.1 节的学习，读者对如何使用 Struts2 框架进行文件上传已经有了一定的了解。本节将通过具体案例演示如何在 Struts2 项目中实现单个文件上传。此案例要求在文件上传成功后，跳转到上传结果页面，页面中要显示上传文件的信息。

1. 创建项目

在 Eclipse 中创建一个名称为 chapter22 的 Web 项目，添加 Struts2 所需的 JAR 包，并在 web.xml 中配置 Struts2 的核心过滤器。

2. 创建上传页面

在项目的 WebContent 目录下创建一个名称为 fileUpload.jsp 的文件，使用该文件作为文件上传页面。编辑后如文件 22-1 所示。

文件 22-1　fileUpload.jsp 页面

```
<%@page language="java" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                        "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<title>上传页面</title>
</head>
<body>
    <center>
        <s:form action="fileUpload" method="post" enctype="multipart/form-data">
            <s:file name="file" label="上传文件" />
            <s:submit value="上传" />
        </s:form>
    </center>
</body>
</html>
```

在上述页面代码中，创建了一个 form 表单，并修改 form 表单的 enctype 属性值为 multipart/form-data，以及修改 method 属性值为 post，然后使用<s:file>标签创建一个上传文本框，并使用<s:submit>标签创建了上传按钮。

3. 创建 Action

在 src 下新建一个名称为 com.itheima.action 的包，在包中新建一个名称为 FileUploadAction 的类，该类用于接收上传的文件，并执行上传方法，编辑后如文件 22-2 所示。

文件 22-2 FileUploadAction.java

```
1  package com.itheima.action;
2  import java.io.File;
3  import java.io.FileInputStream;
4  import java.io.FileOutputStream;
5  import java.io.InputStream;
6  import java.io.OutputStream;
7  import org.apache.struts2.ServletActionContext;
8  import com.opensymphony.xwork2.ActionSupport;
9  public class FileUploadAction extends ActionSupport{
10     private static final long serialVersionUID=1L;
11     private File file;                    //提交的文件
12     private String fileFileName;          //提交的 file 的名字
13     private String fileContentType;       //提交的 file 的类型
14     //此处省略三个属性的 getter 和 setter 方法
15     public String execute() throws Exception
16     {
17         //文件输入流
18         InputStream is=new FileInputStream(file);
19         //设置文件保存的目录
20         String uploadPath=ServletActionContext.getServletContext()
21                                   .getRealPath("/upload");
22         //设置目标文件
23         File toFile=new File(uploadPath, this.getFileFileName());
24         //文件输出流
25         OutputStream os=new FileOutputStream(toFile);
26         byte[] buffer=new byte[1024];
27         int length=0;
28         //读取 file 文件输出到 toFile 文件中
29         while(-1 != (length=is.read(buffer, 0, buffer.length)))
30         {
31             os.write(buffer);
32         }
33         //关闭输入流和输出流
34         is.close();
35         os.close();
36         return SUCCESS;
37     }
38 }
```

在文件 22-2 中，声明了 file（上传的文件）、fileFileName（上传的文件名称）和 fileContentType（上传的文件类型）三个属性，其中 file 属性的名称与表单中文本域的参数必须相同，并且要为 File 类型，它用于保存封装的上传文件。一般来说，Action 中的属性与表单中提交的参数是相对应的，但在此表单中并没有提交 fileFileName 和 fileContentType

这两个参数，因为这两个参数是由 FileUploadIntercepter(文件上传拦截器)负责填充的，FileUploadIntercepter 会对其属性值进行设置。fileFileName 属性用于保存上传文件名称，fileContentType 属性用于保存上传文件的类型。在 execute()方法中通过输入流读取 File 文件中的数据，通过取得上传文件名和上传目录设置一个目标文件，并通过循环输出流将数据写入到目标文件中，最后关闭输入/输出流。

4. 创建配置文件

在 src 下创建 Struts2 的配置文件 struts. xml，编辑后如文件 22-3 所示。

文件 22-3　struts. xml

```
<? xml version="1.0" encoding="UTF-8" ? >
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
    <package name="struts2" namespace="/" extends="struts-default">
        <action name="fileUpload" class="com.itheima.action.FileUploadAction">
            <result name="success">/result.jsp</result>
        </action>
    </package>
</struts>
```

5. 创建结果页面

在项目的 WebContent 目录下新建一个名称为 result 的页面文件，此文件用于显示上传的结果，包括上传文件的名称及类型，编辑后如文件 22-4 所示。

文件 22-4　result. jsp

```
<%@page language="java" contentType="text/html;
charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib prefix="s" uri="/struts-tags"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
                    "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>结果页面</title>
</head>
<body>
    <h2>文件上传成功!</h2><br />
    上传文件名: ${fileFileName}<br />
    上传文件类型: ${fileContentType}
</body>
</html>
```

部署项目，在浏览器的地址栏中输入地址 http://localhost:8080/chapter22/fileUpload.jsp 后，即可显示如图 22-1 所示的上传页面。

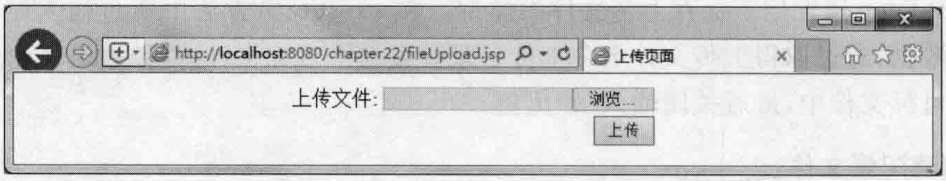


图 22-1 上传页面

单击图 22-1 中的【浏览】按钮，打开文件选择框，选择需要上传的文件后，单击【上传】按钮进行文件上传操作，上传成功后跳转到上传结果页面，如图 22-2 所示。

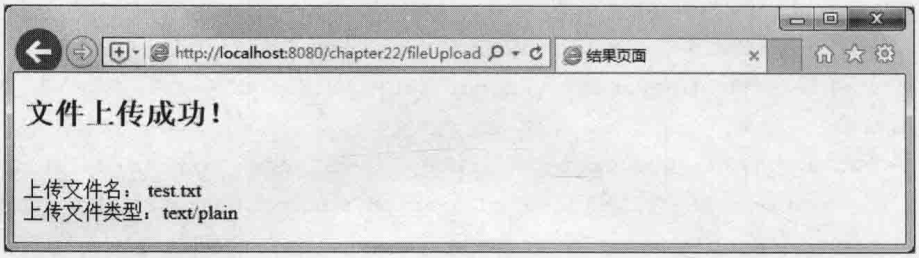


图 22-2 上传结果页面

打开上传目录，可以看到新上传的文件，如图 22-3 所示。

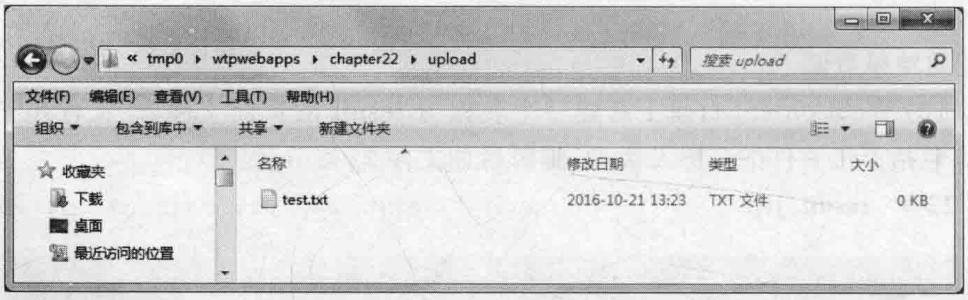


图 22-3 上传目录 upload

需要注意的是，在上传文件前，需要在项目的 WebContent 路径下新建一个名称为 upload 的文件夹作为上传目录，如果没有此目录，则会出现系统找不到文件路径的异常。

22.1.3 限制文件的大小和类型

在网站中上传图片或者其他资源时，通常都会有一些特殊的要求，如文件大小、上传格式等。使用 Struts2 框架进行文件上传时，通过设置上传拦截器 FileUpload，同样可以实现这些功能。

在 FileUpload 中，有三个属性可以设置，具体如下。

- maximumSize：上传文件的最大长度(以字节为单位)，默认值为 2MB。
- allowedTypes：允许上传文件的类型，各类型之间以逗号分隔。

• allowedExtensions：允许上传文件的扩展名，各扩展名之间以逗号分隔。

在项目的 struts.xml 中覆盖这三个属性即可对上传的文件大小和类型进行限制。修改第 22.1.2 节中的 struts.xml 文件，在 FileUploadAction 中添加拦截器，如文件 22-5 所示。

文件 22-5　struts.xml

```
<? xml version="1.0" encoding="UTF-8" ? >
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
   "http://struts.apache.org/dtds/struts-2.3.dtd">
<struts>
<!--设置文件上传允许最大值为 10MB-->
<constant name="struts.multipart.maxSize" value="10485760"/>
<package name="struts2" namespace="/" extends="struts-default">
    <action name="fileUpload" class="com.itheima.action.FileUploadAction">
     <result name="success">/result.jsp</result>
     <!--定义上传出错要跳转的页面-->
     <result name="input">/fileUpload.jsp</result>
     <interceptor-ref name="defaultStack">
         <!--限制上传文件最大值为 4MB-->
         <param name="fileUpload.maximumSize">4194304</param>
         <!--当下面两个配置同时存在时，要同时满足下面两个条件，才能上传-->
         <!--限制上传文件的扩展名-->
         <param name="fileUpload.allowedExtensions">.txt,.doc,.jpg</param>
         <!--限制上传文件的类型-->
         <param name="fileUpload.allowedTypes">
            text/plain,application/msword,image/jpeg
         </param>
     </interceptor-ref>
 </action>
</package>
</struts>
```

在文件 22-5 中，首先通过设置常量 struts.multipart.maxSize 将允许上传文件的最大值设置为 10MB（Struts2 默认大小为 2MB），然后增加了上传文件出错时的跳转页面 fileUpload.jsp。最后通过设置 FileUpload 拦截器的三个属性，分别将上传文件的最大值设置为 4MB，上传文件的扩展名设置为 txt、doc 和 jpg，上传文件的扩展名类型设置为 text/plain、application/msword 和 image/jpeg。

重新发布项目并访问 fileUpload.jsp 页面，在所允许的类型中选择一个文件大小超过 4MB 的文件上传时，浏览器页面的显示结果如图 22-4 所示。

从图 22-4 中可以看出，页面中提示了文件过大的英文错误信息，由于设定的允许最大值是 4MB，当上传的文件大小超过此数值时，就会报出此错误。

当上传一个不符合所设置的文件类型的文件（如 PDF 文件）时，浏览器页面的显示结果如图 22-5 所示。

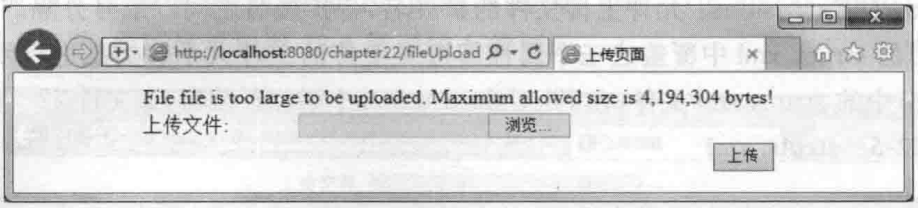


图 22-4 上传文件超过设定值

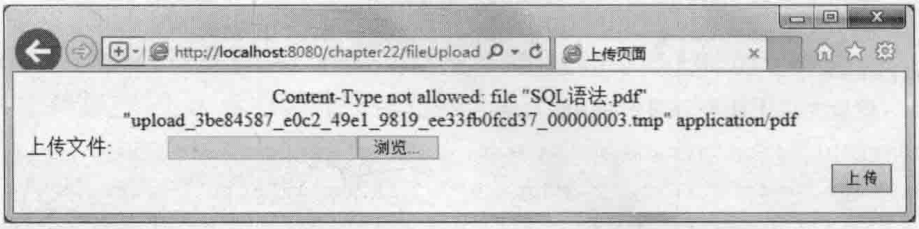


图 22-5 上传文件类型不符合设定值

从图 22-5 中可以看出，当上传的文件类型与设定值不匹配时，将提示上传文件类型不被允许的信息。

注意：allowedExtensions 的配置和 allowedTypes 的配置既可以单独配置也可以同时配置，当两个配置同时存在时，要同时满足这两个条件才能上传文件。对于文件的扩展名所对应的文件类型，读者可以打开 Tomcat 目录下的 conf 文件夹，找到其中的 web.xml 文件，在该文件中列出了几乎所有的文件类型。读者可以通过文件扩展名找到相对应的文件类型。

22.2 文件下载

22.2.1 文件下载的概述

文件上传是将本地文件上传到服务器中，而文件下载正好相反，是指将文件从服务器下载到本地机器。Struts2 框架不但可以实现文件上传，同样也提供了对文件下载功能的支持。要使用 Struts2 框架的文件下载，首先要在 Action 类中添加一个新方法，该方法返回值为一个 InputStream，InputStream 代表了被下载文件的入口，其示例代码如下所示：

```
public class DownLoadAction extends ActionSupport{
    //定义了返回 InputStream 的方法，该方法作为被下载文件的入口
    public InputStream getDownloadFile(){
        return ServletActionContext.getServletContext()
            .getResourceAsStream("/upload/Struts.txt");
    }
    @Override
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

在上述示例代码中，定义了 getDownloadFile() 方法，该方法的返回值是一个 InputStream，在 InputStream 中指定了下载文件的路径以及下载文件的名称。

实现了 Action 中的方法后，还需要在配置文件 struts.xml 中设置 Action 的返回值为 stream 类型，并在 stream 结果类型中指定一个 inputName 参数，该参数用于指定被下载文件的入口。

配置 stream 结果类型时，可以指定以下几个属性。

- contentType：指定下载文件的文件类型，该文件类型要与互联网 MIME 标准中规定的类型相一致，如 text/plain 代表纯文本。
- inputName：指定下载文件的输入流入口，在 Action 中需要指定该输入流入口。
- contentDisposition：指定文件下载的处理方式，有内联(Inline，直接显示文件)和附件(Attachment，弹出文件保存对话框)两种方式，默认为内联。
- bufferSize：用于设置下载文件时的缓存大小，默认为 1024。

Action 配置信息的示例代码如下所示：

```
<!--文件下载 action-->
<action name="XXX" class="XXX">
    <result type="stream">
        <!--文件类型-->
        <param name="contentType">text/plain</param>
        <!--指定文件名-->
        <param name="contentDisposition">attachment;filename="XXX.txt"</param>
        <!--输入流入口-->
        <param name="inputName">downloadFile</param>
        <!--缓存大小-->
        <param name="bufferSize">1024</param>
    </result>
</action>
```

22.2.2　文件下载

通过第 22.2.1 节文件下载知识的学习，读者已经了解了使用 Struts2 框架实现在指定的目录中下载指定文件的功能。下面通过案例演示文件下载功能。

1. 创建下载页面

在 chapter22 项目的 WebContent 目录下创建一个名称为 download.jsp 的页面文件，在文件中添加一个用于文件下载的链接，其主要代码如下所示：

```
<s:a href="simpledownload? filename=test.txt" name="test">test.txt</s:a>
```

2. 创建 Action

在 com.itheima.action 包中新建一个名称为 SimpleDownLoadAction 的类，该类主要用于处理文件下载的核心操作，其代码如文件 22-6 所示。

文件 22-6 SimpleDownLoadAction.java

```
package com.itheima.action;
import java.io.InputStream;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class SimpleDownLoadAction extends ActionSupport {
    private String filename;
    public String getFilename() {
        return filename;
    }
    public void setFilename(String filename) {
        this.filename=filename;
    }
    //定义了返回 InputStream 的方法,该方法作为被下载文件的入口
    public InputStream getDownloadFile() {
        //获取下载文件名称
        String filename=ServletActionContext.getRequest().
        getParameter("filename");
        //定义下载文件路径
        String filePath="/upload/"+filename;
        //返回一个流对象
        return ServletActionContext.getServletContext().
        getResourceAsStream(filePath);
    }
}
```

3. 编写配置文件信息

在 struts.xml 中增加 action 的配置,其代码如下所示:

```
<action name="simpledownload" class="com.itheima.action.SimpleDownLoadAction">
    <result type="stream">
        <!--文件类型-->
        <param name="contentType">text/plain</param>
        <!--指定文件名-->
        <param name="contentDisposition">attachment;filename=${filename}</param>
        <!--输入流-->
        <param name="inputName">downloadFile</param>
    </result>
</action>
```

在上述配置代码中,分别通过 stream 结果类型的属性设置了所要下载文件的类型、名称和输入流。其中 ${filename} 表示在项目运行时,将 action 中的 filename 属性动态地填充在 ${} 中间部分,这样就可以动态地获取所要下载的文件名称。

4. 运行项目并查看结果

启动项目后，在浏览器的地址栏中输入地址 http://localhost:8080/chapter22/download.jsp，即可显示如图 22-6 所示的页面。

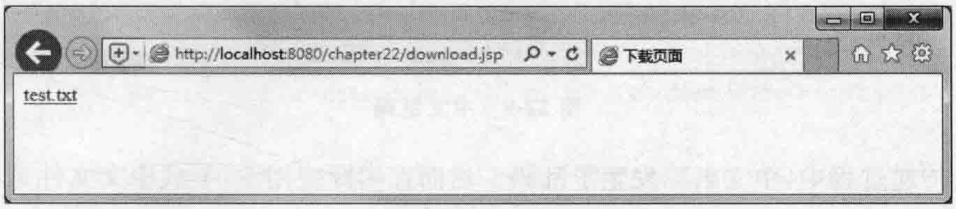


图 22-6　下载页面

单击下载页面的超链接后，浏览器的显示结果如图 22-7 所示。



图 22-7　上传页面

22.2.3　中文文件的下载

通过第 22.2.2 节的学习，读者已经学会了如何使用 Struts2 框架实现文件下载，但是细心的读者会发现，如果上传一个中文名称的文件(如文本文件.txt)，再次下载此文件时，页面会报出 500 错误，如图 22-8 所示。

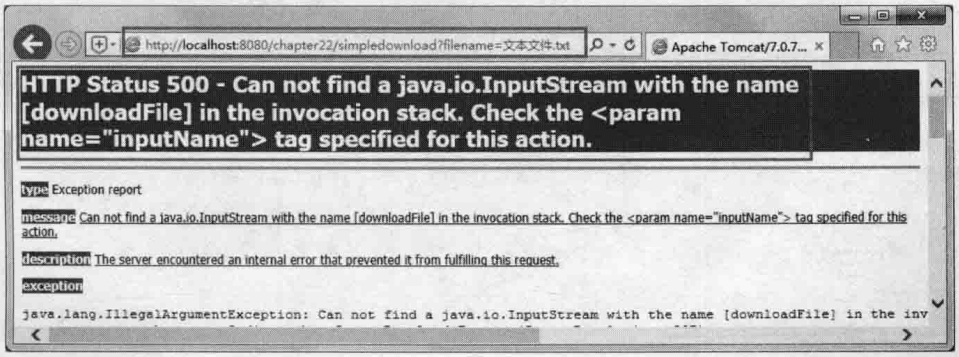


图 22-8　中文文件下载报错

之所以出现图 22-8 中的错误，是因为所要下载的文件不存在，导致 inputStream 为 null，所以报出了 500 错误。这里读者一定会很费解，明明上传了一个新文件，并可以确认文件在文件夹中是存在的，那么为什么还会报出此错误呢？

如果在文件 22-6 的第 16 行代码中设置断点，并采用 Debug 模式调试执行程序，会发现传递到 Action 中的中文名称变成了乱码，如图 22-9 所示。

从图 22-9 中可以看到，页面中传递的“文本文件.txt”已经变为“æ□ □ æ□ ¬æ□ □ ä»¶.txt”，

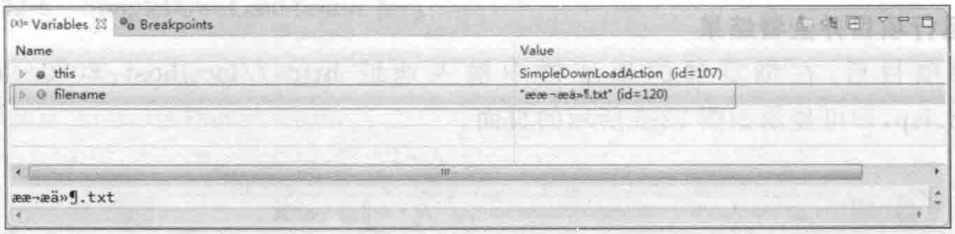


图 22-9 中文乱码

这说明在传递过程中，中文名称发生了乱码。然而在实际应用中，下载中文文件是不可避免的。下面本节将通过一个具体的案例讲解如何使用 Struts2 框架下载中文文件。

1. 创建文件下载页面

在项目的 WebContent 目录中创建一个名称为 filedownload.jsp 的文件，在文件中添加一个中文链接，其代码如下所示：

```
<s:a href="filedownload? filename=文本文件.txt">文本文件.txt</s:a>
```

2. 创建 Action 类

在 com.itheima.action 包中新建一个名称为 FileDownLoadAction 的类，该类是用于下载文件的 Action，编辑后如文件 22-7 所示。

文件 22-7 FileDownLoadAction.java

```
1  package com.itheima.action;
2  import java.io.IOException;
3  import java.io.InputStream;
4  import java.io.UnsupportedEncodingException;
5  import java.net.URLEncoder;
6  import org.apache.struts2.ServletActionContext;
7  import sun.misc.BASE64Encoder;
8  import com.opensymphony.xwork2.ActionSupport;
9  public class FileDownLoadAction extends ActionSupport {
10     private String filename;              //代表下载文件的名称
11     private String contentType;           //下载文件的 MimeType
12     /**
13      * 获取文件的名称
14      */
15     public String getFilename() throws IOException {
16         //对不同浏览器传过来的文件名进行编码
17         return encodeDownloadFilename(filename,
18                 ServletActionContext.getRequest().getHeader("User-Agent"));
19     }
20     public void setFilename(String filename) throws UnsupportedEncodingException {
21         //对文件名称编码
22         filename=new String(filename.getBytes("iso8859-1"), "utf-8");
```

```
23          this.filename=filename;
24      }
25      /**
26       * 获取文件的类型
27       */
28      public String getContentType() {
29          return ServletActionContext.getServletContext().getMimeType(filename);
30      }
31      public void setContentType(String contentType) {
32          this.contentType=contentType;
33      }
34      //定义了返回 InputStream 的方法,该方法作为被下载文件的入口
35      public InputStream getFileDownload(){
36          //要下载的文件的路径
37          String filepath="/upload/"+filename;
38          return ServletActionContext.getServletContext().
39          getResourceAsStream(filepath);
40      }
41      /**
42       * 对不同浏览器传过来的文件名称进行转码
43       * @param name 文件名称
44       * @param agent 浏览器
45       * @return 转码后的名称
46       */
47      public String encodeDownloadFilename(String name, String agent) throws
48      IOException {
49          if (agent.contains("Firefox")) {          //火狐浏览器
50              name="=? UTF-8? B?"+new BASE64Encoder().encode(name.getBytes("utf-8"))
51                      +"? =";
52          } else {                                  //IE 及其他浏览器
53              name=URLEncoder.encode(name, "utf-8");
54          }
55          return name;
56      }
57  }
```

在文件 22-7 中,声明了 filename 和 contentType 两个属性,分别代表所要下载文件的名称和类型。在 filename 属性的 getter()方法中,通过 encodeDownloadFilename(String name, String agent)方法对不同浏览器传过来的名称进行编码,在其 setter()方法中,对其文件名称又进行了一次编码,使文件名称统一为 UTF-8 编码格式。通过 contentType 属性的 getContentType()方法获取了文件的类型,并在 getFileDownload()方法中定义了要下载的文件路径,并返回一个输入流。

3. 修改配置文件

在 struts.xml 文件中，增加 FileDownLoadAction 的配置信息，其代码如下所示：

```
<action name="filedownload" class="com.itheima.action.FileDownLoadAction">
    <result type="stream">
        <!--文件类型-->
        <param name="contentType">${contentType}</param>
        <!--指定文件名-->
        <param name="contentDisposition">attachment;filename=${filename}</param>
        <!--输入流入口-->
        <param name="inputName">fileDownload</param>
    </result>
</action>
```

在上述配置代码中，${contentType}和${filename}会在项目运行时，将 action 中的对象属性动态地填充在${}中间部分，相当于 action.getContentType() 和 action.getFilename()，通过此种方式动态地获取文件的类型和名称。

4. 运行程序并查看结果

在浏览器的地址栏中输入地址 http://localhost:8080/chapter22/filedownload.jsp，成功访问后，单击页面中的“文本文件.txt”链接，其效果如图 22-10 所示。



图 22-10 中文文件下载

第 23 章 Hibernate框架入门

Hibernate 是一款免费开源的持久层框架，它对 JDBC 进行了轻量级的对象封装，将对象与数据库表建立了映射关系，使 Java 编程人员可以随心所欲地使用面向对象的编程思想操作数据库。Hibernate 具有简单易用、灵活性强、扩展性强等特点，大大提高了程序的开发效率，本章将针对 Hibernate 框架的基础知识进行详细讲解。

23.1 Hibernate 概述

使用传统的 JDBC 开发小型应用系统，并不会有什么麻烦，但是对于大型应用系统的开发，使用 JDBC 就会显得力不从心。例如对几十、几百张包含几十个字段的表进行插入操作时，编写的 SQL 语句不但很长，而且繁琐，容易出错；在读取数据时，需要编写多条语句从结果集中取出各个字段的信息，工作量非常大。为了提高数据访问层的编程效率，Gavin King 开发了一个当今最流行的 ORM（即 Object-Relational Mapping 对象关系映射）框架——Hibernate 框架。

ORM 就是利用描述对象和关系型数据库之间的映射信息，自动将 Java 应用程序中的对象持久化到关系型数据库的表中。通过操作 Java 对象，就可以完成对数据库表的操作。可以把 ORM 理解为关系型数据和对象之间的一个纽带，开发人员只需要关注纽带一端映射的对象即可。

与其他操作数据库的技术相比，Hibernate 具有以下几点优势。

- Hibernate 对 JDBC 访问数据库的代码进行了轻量级封装，大大简化了数据访问层繁琐的重复性代码，并且减少了内存消耗，加快了运行效率。
- Hibernate 使用 Java 的反射机制，而不是使用字节码增强程序类，并实现了透明性。
- Hibernate 的功能强大，映射的灵活性很出色。它支持很多关系型数据库，从一对一到多对多的各种复杂关系。
- 可扩展性强，由于源代码的开源以及 API 的开放，当本身功能不够用时，可以自行编码进行扩展。

23.2 Hibernate 下载和目录介绍

目前企业主流使用的 Hibernate 版本以 Hibernate 3. x 版本为主，这里以 Hibernate 3. 6. 10 版本为例。初学者可以从官方网址 http://sourceforge. net/projects/hibernate/files/hibernate3/中下载对应的版本。Hibernate 3. 6. 10 版本的下载过程，如图 23-1 所示。

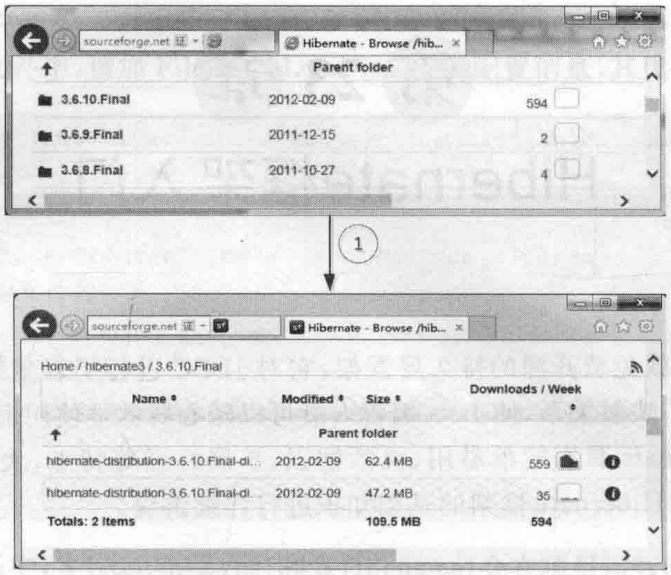


图 23-1 Hibernate 的下载过程

在图 23-1 中，单击 3.6.10.Final 链接，进入下载页面，在该页面中有两个压缩包链接，其中 hibernate-distribution-3.6.10.Final-dist.zip 是 Windows 下的版本，Final 表示版本号为正式版；hibernate-distribution-3.6.10.Final-dist.tar.gz 是 Linux 下的版本，读者可以根据自己的环境需求下载指定版本的 Hibernate，本书下载的是 Windows 版本的 ZIP 压缩包。

下载完成后，对 Hibernate 3.6.10 进行解压，解压后的目录结构，如图 23-2 所示。



图 23-2 Hibernate 压缩包的文件结构

从图 23-2 中可以看出，Hibernate 3.6.10 安装目录中包含一系列的子目录，这些子目录分别用于存放不同功能的文件，下面针对这些子目录进行简单介绍。

- documentation 文件夹：该路径下存放了 Hibernate 的相关文档，包括 Hibernate 的参考文档和 API 文档等。
- lib 文件夹：该路径下存放了 Hibernate 3 的核心类库，以及编译和运行所依赖的第三方类库。其中 lib 路径下的 required 子目录中包含了运行 Hibernate 3 所必须的 JAR 包。

- project 文件夹：该路径下存放了 Hibernate 各种相关项目的源代码。
- hibernate3.jar 文件：此 JAR 包是 Hibernate 3 的核心 JAR。

在实际开发中，需要将 Hibernate 3 的核心 JAR 以及 lib/required 子目录中的 JAR 包添加到项目中，除此之外，通常还需要添加 Hibernate 3 对 JPA 支持的 JAR，以及实现日志功能的 JAR。使用 Hibernate 3 开发项目时，所需的 JAR 包及其说明如表 23-1 所示。

表 23-1　Hibernate 3 所需的 JAR 包的说明

JAR 包名称	描　述
hibernate3.jar	Hibernate 的核心包
antlr-2.7.6.jar	语言转换工具，Hibernate 利用它实现 HQL 到 SQL 的转换
commons-collections-3.1.jar	collections Apache 的工具集，用于增强 Java 对集合的处理能力
dom4j-1.6.1.jar	dom4j XML 解析器
javassist-3.12.0.GA.jar	代码生成工具，Hibernate 利用它在运行时扩展 Java 类
jta-1.1.jar	标准的 JAVA 事务(跨数据库)处理接口
slf4j-api-1.6.1.jar	只是一个接口，用于整合 log4j
hibernate-jpa-2.0-api-1.0.1.Final.jar	JPA 接口开发包
log4j-1.2.17.jar	Log4j 日志文件核心 JAR
slf4j-log4j12-1.7.12.jar	Hibernate 使用的一个日志系统

从表 23-1 中可以看出，Hibernate 3 所依赖的 JAR 包共有 10 个。其中 hibernate-jpa-2.0-api-1.0.1.Final.jar 是 JPA 接口的开发包，它位于 Hibernate 的 lib 子目录的 jpa 文件夹中。由于 Hibernate 并没有提供对日志的实现，所以需要 slf4j 和 log4j 开发包整合 Hibernate 的日志系统到 log4j。

Hibernate 的环境搭建非常简单，将表 23-1 中的所有 JAR 包复制到项目的 WEB-INF/lib 目录下即可。需要注意的是，Hibernate 框架要想与数据库建立连接，使用时还需要导入相关数据库的驱动 JAR 包。

23.3　第一个 Hibernate 程序

23.3.1　编写 Hibernate 程序

通过前面的学习，读者对 Hibernate 已经有了一个初步的了解。下面通过一个完整的增、删、改、查案例讲解 Hibernate 框架的使用。

1. 创建项目并导入 JAR 包

在 Eclipse 中创建一个名称为 chapter23 的 Web 项目，将 Hibernate 所需 JAR 包和 MySQL 的驱动包(mysql-connector-java-5.0.8-bin.jar)复制到项目的 WEB-INF/lib 目录中，并将所有 JAR 添加到类路径下。添加 JAR 包后的项目结构如图 23-3 所示。

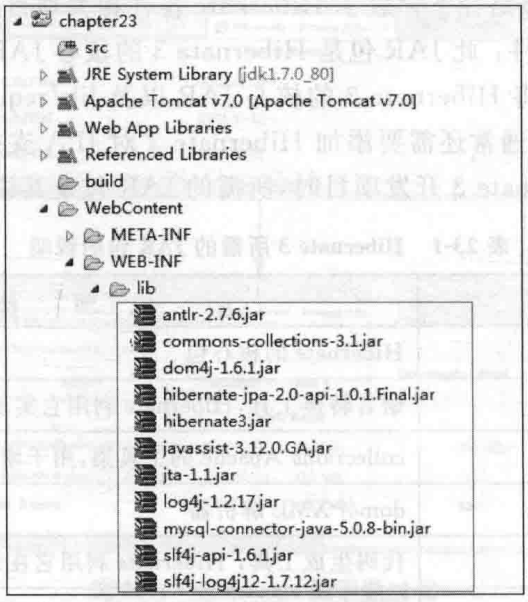


图 23-3 Hibernate 所需的 JAR 包

2. 创建数据库

在 MySQL 中创建一个名称为 hibernate 的数据库，然后在该数据库中创建一个 user 表，创建数据库和表的 SQL 语句如下所示：

```
CREATE DATABASE hibernate;
USE hibernate;
CREATE TABLE user(
    id INT(32) PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(20),
    age INT(4),
    gender VARCHAR(4)
);
```

在 SQLyog 工具中，使用 desc user 语句查询 user 表结构，其执行结果如图 23-4 所示。

	Field	Type		Null	Key	Default		Extra
□	id	int(32)	7B	NO	PRI	(NULL)	OK	auto_increment
□	name	varchar(20)	11B	YES		(NULL)	OK	
□	age	int(4)	6B	YES		(NULL)	OK	
□	gender	varchar(4)	10B	YES		(NULL)	OK	

图 23-4 user 表结构

3. 编写实体类(持久化类)

持久化类是应用程序中的业务实体类，符合基本的 JavaBean 编码规范。Hibernate 操作的持久化类基本上都是普通的 Java 对象(Plain Ordinary Java Object，POJO)，这些普通 Java 对象中包含的是与数据库表相对应的各个属性，并且这些属性可以通过 getter 和 setter 方法访问。

下面编写一个持久化类 User。在项目的 src 目录下创建一个名称为 com. itheima. domain 的包，在包中创建一个实体类 User(对应数据库表 user)，并定义其属性(包括 id、name、age 和 gender)以及相应的 getter 和 setter 方法。如文件 23-1 所示。

文件 23-1　User. java

```
1  package com.itheima.domain;
2  public class User {
3      private Integer id;                    //唯一标识 id
4      private String name;                   //姓名
5      private Integer age;                   //年龄
6      private String gender;                 //性别
7      public Integer getId() {
8          return id;
9      }
10     public void setId(Integer id) {
11         this.id=id;
12     }
13     public String getName() {
14         return name;
15     }
16     public void setName(String name) {
17         this.name=name;
18     }
19     public Integer getAge() {
20         return age;
21     }
22     public void setAge(Integer age) {
23         this.age=age;
24     }
25     public String getGender() {
26         return gender;
27     }
28     public void setGender(String gender) {
29         this.gender=gender;
30     }
31     //重写 toString()方法
32     @Override
33     public String toString() {
34         return "User [id="+id+", name="+name
35                 +", age="+age+", gender="+gender+"]";
36     }
37 }
```

文件 23-1 中定义的四个属性分别与 user 表中的字段相对应。由于 Hibernate 默认访问的是各个属性的 getter 和 setter 方法，所以为了实现类的封装性，建议为持久化类的各个属性添加 getter 和 setter 方法。

通常持久化类的编写应该遵循一些规则，具体如下。

- 提供一个无参数的 public 访问控制符的构造器。
- 持久化类中所有属性使用 private 修饰。
- 所有属性提供 public 修饰的 setter 和 getter 方法。
- 提供一个标识属性 OID，映射数据表主键字段，例如 User 表的 id 属性。
- 标识属性应尽量使用基本数据类型的包装类型，目的是为了与数据库表的字段默认值 null 一致。
- 不要用 final 修饰持久化类，否则无法生成代理对象。

4. 编写映射文件

实体类 User 目前还不具备持久化操作的能力，为了使该类具备这种能力，需要通知 Hibernate 框架将 User 实体类映射到数据库的某一张表中，以及类中的哪个属性对应数据表的哪个字段，这些都需要在映射文件中配置。

在 com.itheima.domain 包中，创建一个名称为 User.hbm.xml 的映射文件，编辑后如文件 23-2 所示。

文件 23-2　User.hbm.xml

```
<? xml version="1.0" encoding="UTF-8"? >
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <!--name 代表的是类名，table 代表的是表名-->
    <class name="com.itheima.domain.User" table="user">
        <!--name 代表的是 User 类中的 id 属性，column 代表的是 user 表中的主键 id-->
        <id name="id" column="id">
            <!--主键生成策略-->
            <generator class="native" />
        </id>
        <!--其他属性使用 property 标签映射-->
        <property name="name" column="name" type="string" />
        <property name="age" column="age" type="integer" />
        <property name="gender" column="gender" type="string" />
    </class>
</hibernate-mapping>
```

文件 23-2 展示了实体类 User 与数据表 user 的映射关系。在上述映射文件中，<class>节点用于配置实体类的映射信息，其中 name 属性表示实体类的完整类名，table 属性表示数据表的名称。<id>节点用于定义实体类的唯一标识（对应数据表的主键），是<class>节点下的必须节点，其 name 属性对应实体类的属性，<column>用于对应数据库表中的列，这里即为表的主键，<generator>子节点用于指定主键的生成策略。<property>节点用于映射普通属性，其 name 属性对应实体中的属性，column 属性对应数据库表中的字段，type 属性表示其属性的类型。

关于映射文件中的节点元素，在第 23.5.1 节中会进行更为详细的讲解，此处读者了解

即可。

5. 编写核心配置文件 hibernate.cfg.xml

Hibernate 映射文件反映了持久化类和数据表的映射信息，而 Hibernate 核心配置文件则用于配置 Hibernate 连接数据库的相关信息，如数据库的驱动、URL、用户名、密码等。

在 src 目录下创建一个名称为 hibernate.cfg.xml 的文件，如文件 23-3 所示。

文件 23-3　hibernate.cfg.xml

```
<? xml version="1.0" encoding="UTF-8"? >
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!--指定方言-->
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <!--数据库驱动-->
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <!--连接数据库的 url-->
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/hibernate
        </property>
        <!--数据库的用户名-->
        <property name="hibernate.connection.username">root </property>
        <!--数据库的密码-->
        <property name="hibernate.connection.password">root </property>
        <!--显示 sql 语句-->
        <property name="hibernate.show_sql">true</property>
        <!--格式化 sql 语句-->
        <property name="format_sql">true</property>
        <!--用于关联 hbm 配置文件-->
        <mapping resource="com/itheima/domain/User.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

在文件 23-3 中，设置了数据库连接的相关配置和其他的一些常用属性，其中<mapping>元素用于将对象的映射信息加入到 Hibernate 的核心配置文件中。

23.3.2　测试 Hibernate 程序

在项目的 src 目录下创建一个名称为 com.itheima.test 的包，在该包下创建一个名称为 UserTest 的测试类。

1. 添加数据

在 UserTest 类中创建一个名称为 testInsert 的方法，该方法用于向数据库中添加数据，其代码如文件 23-4 所示。

文件 23-4 UserTest.java

```
1  package com.itheima.test;
2  import org.hibernate.Session;
3  import org.hibernate.SessionFactory;
4  import org.hibernate.Transaction;
5  import org.hibernate.cfg.Configuration;
6  import org.junit.Test;
7  import com.itheima.domain.User;
8  public class UserTest {
9      //添加操作
10     @Test
11     public void testInsert() {
12         //1.创建 Configuraiont 对象并加载 hibernate.cfg.xml 配置文件
13         Configuration config=new Configuration().configure();
14         //2.获取 SessionFactory
15         SessionFactory sessionFactory=config.buildSessionFactory();
16         //3.得到一个 Session
17         Session session=sessionFactory.openSession();
18         //4.开启事务
19         Transaction transaction=session.beginTransaction();
20         //5.执行持久化操作
21         User user=new User();
22         user.setName("zhangsan");
23         user.setAge(21);
24         user.setGender("男");
25         //将对象保存到到表中
26         session.save(user);
27         //6.提交事务
28         transaction.commit();
29         //7.关闭资源
30         session.close();
31         sessionFactory.close();
32     }
33 }
```

在文件 23-4 中，首先创建 Configuration 类的实例，并加载解析 hibernate.cfg.xml 核心配置文件。然后获取 SessionFactory 读取解析映射文件信息，并将 Configuration 对象中的所有配置信息加载到 SessionFactory 内存中。接着得到一个 Session，开启一个事务，之后创建 User 对象，向对象中设置添加数据，通过 session.save()方法完成向数据库中保存数据的操作。最后提交事务，并关闭资源。

多学一招：添加 JUnit 测试

在软件开发过程中，需要有相应的测试工作。依据测试目的的不同，可以将软件测试分为单元测试、集成测试、系统测试等。其中单元测试在软件开发阶段是最底层的测试，它易于及时发现并解决问题。JUnit 就是一个进行单元测试的开源框架。

本程序使用的就是单元测试框架 JUnit4，使用此框架测试时，不需要 main 方法，就可以直接对类中的方法进行运行测试。其中@Test 用于测试的注解，要测试哪个方法，只需要在相应测试的方法上添加此注解即可。当在需要测试方法上写入@Test 后，Eclipse 会在@Test 处报出 Test cannot be resolved to a type 的错误，将鼠标移到@Test 处，在显示的弹出窗口中，单击 Add JUnit4 library to the build path，程序会自动将 JUnit4 的支持包加入到项目中，如图 23-5 所示。

添加后在项目中会多出两个关于 JUnit4 的 JAR 包，在 Eclipse 中的显示如图 23-6 所示。

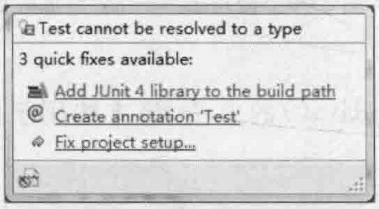


图 23-5　添加 JUnit4

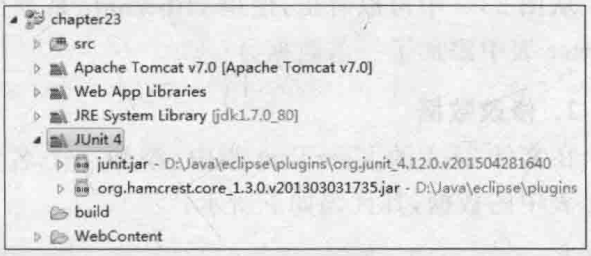


图 23-6　JUnit 框架支持包

在 UserTest 类中，右击 testInsert()方法，在弹出的快捷菜单中选择 Run As→JUnit Test 选项运行测试，如图 23-7 所示。

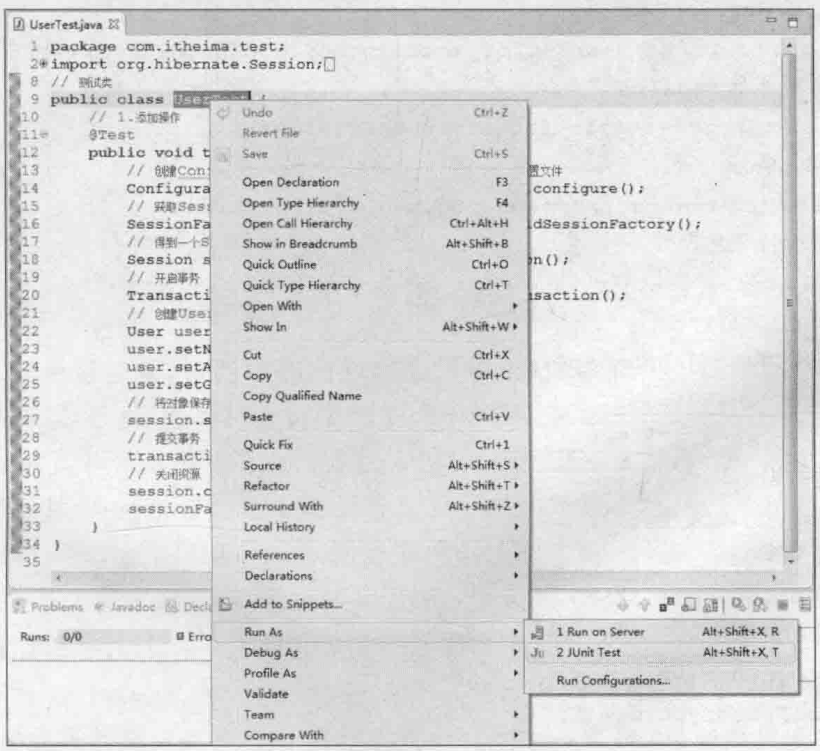


图 23-7　运行 JUnit

单击 JUnit Test 选项后，运行的结果如图 23-8 所示。

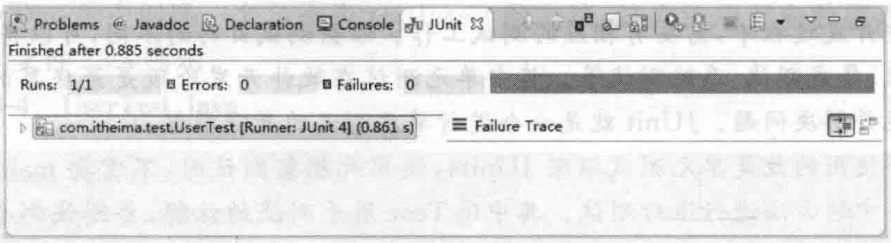


图 23-8 运行结果

在图 23-8 中，当 JUnit 选项卡的进度条为绿色时，表明运行结果正确，如果进度条为红色，则表示有错误，并且会在窗口中显示所报的错误信息。运行成功后，查询数据库 user 表中的数据，其结果如图 23-9 所示。

	id	name	age	gender
□	1	zhangsan	21	男

图 23-9 使用 Hibernate 添加数据

从图 23-9 中可以看出，使用 Hibernate 框架成功地向 user 表中添加了一条数据。

2. 修改数据

在文件 23-4 的 UserTest 类中，添加一个名为 testUpdate()的方法，该方法用于修改 user 表中的数据，其代码如下所示：

```
@Test
public void testUpdate() {
    //1.创建 Configuraiont 对象并加载 hibernate.cfg.xml 配置文件
    Configuration config=new Configuration().configure();
    //2.获取 SessionFactory
    SessionFactory sessionFactory=config.buildSessionFactory();
    //3.得到一个 Session
    Session session=sessionFactory.openSession();
    //4.开启事务
    Transaction transaction=session.beginTransaction();
    //5.执行持久化操作
    User user=new User();
    user.setId(1);
    user.setName("zhangsan");
    user.setAge(19);
    user.setGender("男");
    //更新数据
    session.update(user);
    //6.提交事务
    transaction.commit();
    //7.关闭资源
    session.close();
    sessionFactory.close();
}
```

从上述代码中可以看出，Hibernate 修改数据的方法与其保存数据的方法类似，只是在修改数据后通过 session. update( ) 方法执行的更新操作。

	id	name	age	gender
□	1	zhangsan	19	男

图 23-10　更新数据后的数据表

使用 JUnit 运行此方法，执行成功后，查询 user 表中的数据，其结果如图 23-10 所示。

从图 23-10 中可以看出，表中 age 字段已经被修改为 19，说明使用 Hibernate 成功更新了 user 表中的数据。

3. 查询数据

在 UserTest 类中，添加一个名称为 testFindById( )的方法，查询 id 为 1 的表数据，其代码如下所示：

```
@Test
public void findByIdTest() {
    //1.创建 Configuraiont 对象并加载 hibernate.cfg.xml 配置文件
    Configuration config=new Configuration().configure();
    //2.获取 SessionFactory
    SessionFactory sessionFactory=config.buildSessionFactory();
    //3.得到一个 Session
    Session session=sessionFactory.openSession();
    //4.开启事务
    Transaction transaction=session.beginTransaction();
    //5.执行查询操作
    User user=(User) session.get(User.class, 1);
    System.out.println(user.getId()+" "+user.getName()+" "
                    +user.getAge()+" "+user.getGender());
    //6.提交事务
    transaction.commit();
    //7.关闭资源
    session.close();
    sessionFactory.close();
}
```

从上述代码中可以看出，与增加和修改数据相比，查询操作中通过使用 session. get( ) 方法获取 User 对象数据，然后将查询出的 User 对象数据输出到控制台。

使用 JUnit4 运行 findByIdTest( )方法后，控制台的显示如图 23-11 所示。

在图 23-11 中，控制台开始处出现的警告部分是由于没有配置 log4j 才显示的，此处可先不必理会。中间部分是 Hibernate 自动生成的 SQL 语句信息，这是由于在 hibernate. cfg. xml 中增加了显示 SQL 语句和格式化 SQL 的配置信息才显示出来的。最后就是所查询出的数据输出信息。

注意：在 Hibernate 中，除了使用 get( )方法加载数据以外，还可以使用 load( )方法加载数据，它们都能将数据从数据库中取出。两者的区别是使用 get( )方法加载数据时，如果指定的记录不存在，则返回 null；而使用 load( )方法加载数据时，如果指定的记录不存在，则

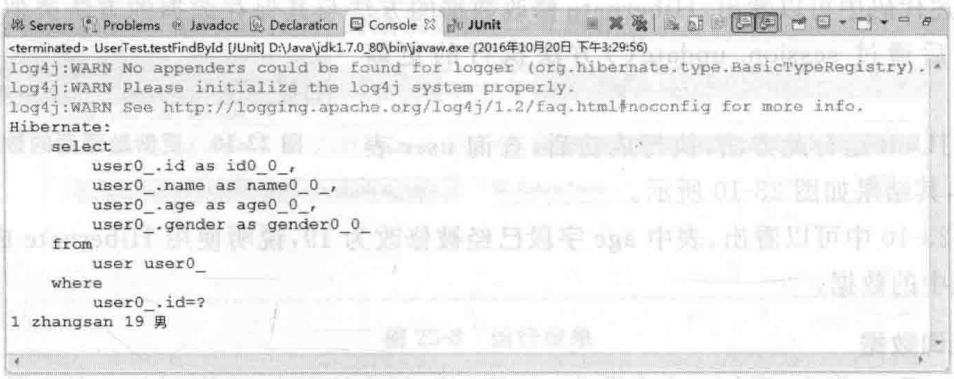


图 23-11 查询结果

会报出 ObjectNotfountException 异常，这一异常说明使用load()方法加载数据时，要求记录必须存在。

4. 删除数据

在 UserTest 类中，添加一个名称为 deleteByIdTest() 的方法，在该方法中使用 Hibernate 将 id 为 1 的这条记录从数据表中删除，其方法代码如下所示：

```
@Test
public void testDeleteById() {
    //1.创建 Configuraiont 对象并加载 hibernate.cfg.xml 配置文件
    Configuration config=new Configuration().configure();
    //2.获取 SessionFactory
    SessionFactory sessionFactory=config.buildSessionFactory();
    //3.得到一个 Session
    Session session=sessionFactory.openSession();
    //4.开启事务
    Transaction transaction=session.beginTransaction();
    //5.执行持久化操作,需要先查询,再删除
    User user=(User) session.get(User.class, 1);
    session.delete(user);
    //6.提交事务
    transaction.commit();
    //7.关闭资源
    session.close();
    sessionFactory.close();
}
```

从上述代码中可以看出，在执行删除操作时，先通过 get()方法获取了 id 为 1 的 User 对象，然后通过 delete()方法删除了该对象数据。

使用 JUnit 运行此方法，执行成功后，再次查询数据库 user 表，会发现 user 表中的数据已经为空，这表明使用 Hibernate 框架删除数据的操作执行成功。

23.4　Hibernate 的运行机制

通过第 23.3 节的学习，读者对 Hibernate 的使用已经有了一个初步的了解。在第 23.3 节的 Hibernate 程序中，主要涉及四个接口的使用，分别为 Configuration 接口、SessionFactory 接口、Session 接口和 Transaction 接口，除了这四个接口以外，其常用接口还有 Query 和 Criteria 等。关于这几个接口的详细内容将在第 23.6 节进行讲解。本节将讲解 Hibernate 的运行机制。

Hibernate 在运行时的执行流程如图 23-12 所示。

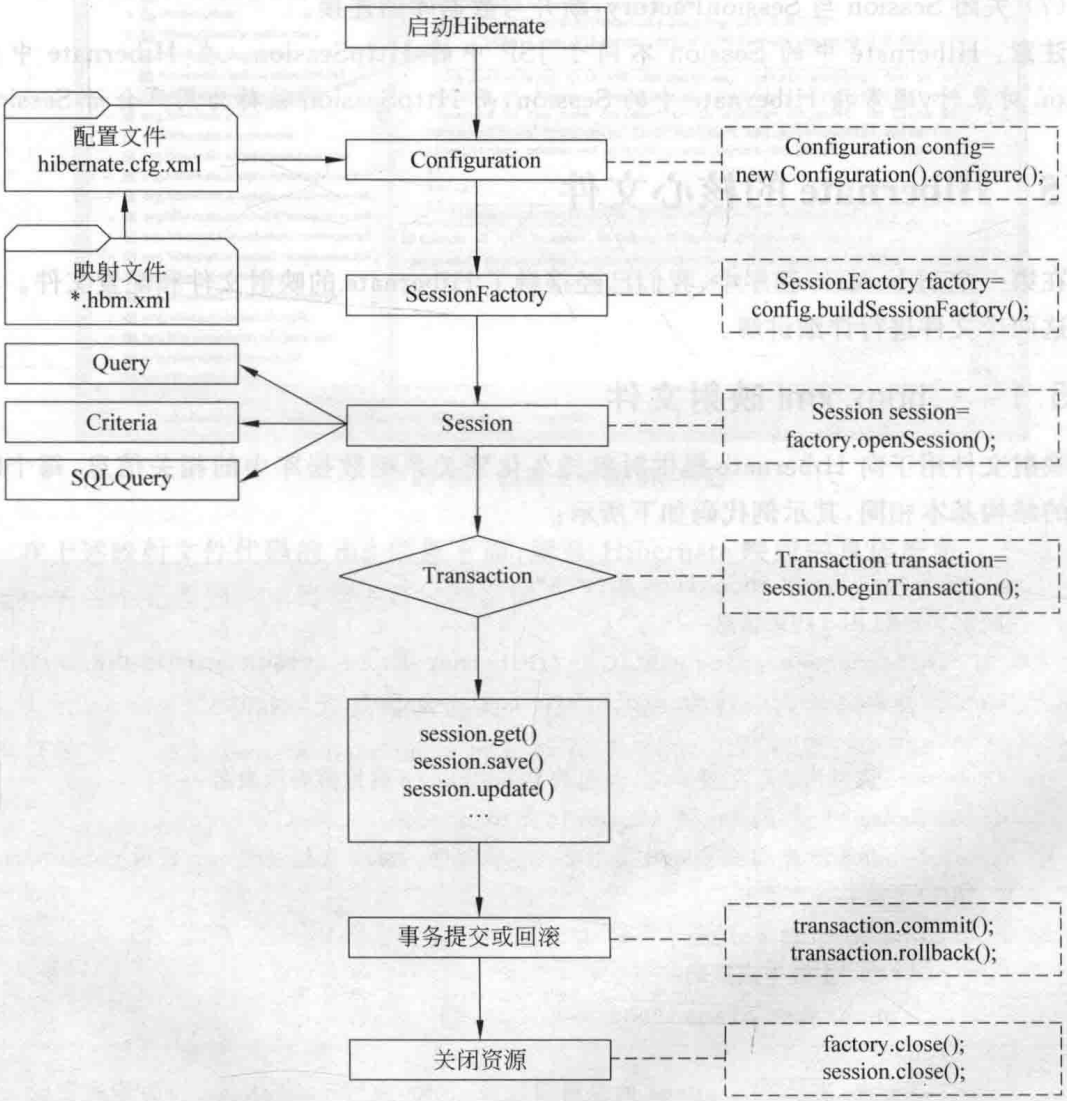


图 23-12　Hibernate 的执行流程

图 23-12 所示的 Hibernate 的执行流程过程具体如下。

(1) 创建 Configuration 实例，加载 Hibernate 核心配置文件和映射文件信息到 Configuration 对象中。

(2) 创建 SessionFactory 实例。通过 Configuration 对象读取到的配置文件信息创建

SessionFactory 对象，该对象中保存了当前数据库的配置信息和所有映射关系等信息。

(3) 创建 Session 实例，建立数据库连接。Session 主要负责执行持久化对象的增、删、改、查操作，创建一个 Session 就相当于创建一个新的数据库连接。

(4) 创建 Transaction 实例，开启一个事务。Transaction 用于事务管理，一个 Transaction 对象对应的事务可以包含多个操作。在使用 Hibernate 进行增、删、改操作时，必须先创建 Transaction 对象。需要注意的是，Hibernate 的事务默认是关闭的，需要手动开启事务和关闭事务。

(5) 利用 Session 接口通过的各种方法进行持久化操作。

(6) 提交事务，对实体对象持久化操作后，必须提交事务。

(7) 关闭 Session 与 SessionFactory，断开与数据库的连接。

注意：Hibernate 中的 Session 不同于 JSP 中的 HttpSession。在 Hibernate 中使用 Session 对象时，通常指 Hibernate 中的 Session，而 HttpSession 被称为用户会话 Session。

23.5 Hibernate 的核心文件

在第一个 Hibernate 程序中，我们已经接触了 Hibernate 的映射文件和配置文件。本节将对这两个文件进行详细讲解。

23.5.1 *.hbm.xml 映射文件

映射文件用于向 Hibernate 提供对象持久化到关系型数据库中的相关信息，每个映射文件的结构基本相同，其示例代码如下所示：

```
<? xml version="1.0" encoding="UTF-8"? >
<!--映射文件的 dtd 约束信息-->
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <!--name 属性值表示完整类名(含包的类名) table 属性值表示表名-->
    <class name="className" table="tableName">
        <!--name 表示 className 类中唯一标识属性，如 id 属性 column 表示 tableName 表
        中的主键-->
        <id name="id" column="id">
            <!--主键生成策略-->
            <generator class="native"/>
        </id>
        <!--name 表示 className 的普通属性 column 表示 tableName 表的普通字段 type
        表示字段类型-->
        <property name="attrName" column="fieldName" type="string" />
    </class>
</hibernate-mapping>
```

在上述代码中，首先进行了 xml 声明，然后定义了映射文件的 dtd 信息，此 dtd 信息读

者不需要去手写，可以在项目的 Web App Libraries 目录(或 Referenced Libraries 目录)中，找到 hibernate 的核心 JAR 包 hibernate3.jar，打开此 JAR 包后，在 org.hibernate 包中即可找到 hibernate-mapping-3.0.dtd 文件。打开此文件后，在文件的最上方即有此 dtd 信息，只需要将此 dtd 信息复制到映射文件中使用即可。hibernate-mapping-3.0.dtd 文件中的 dtd 信息如图 23-13 所示。

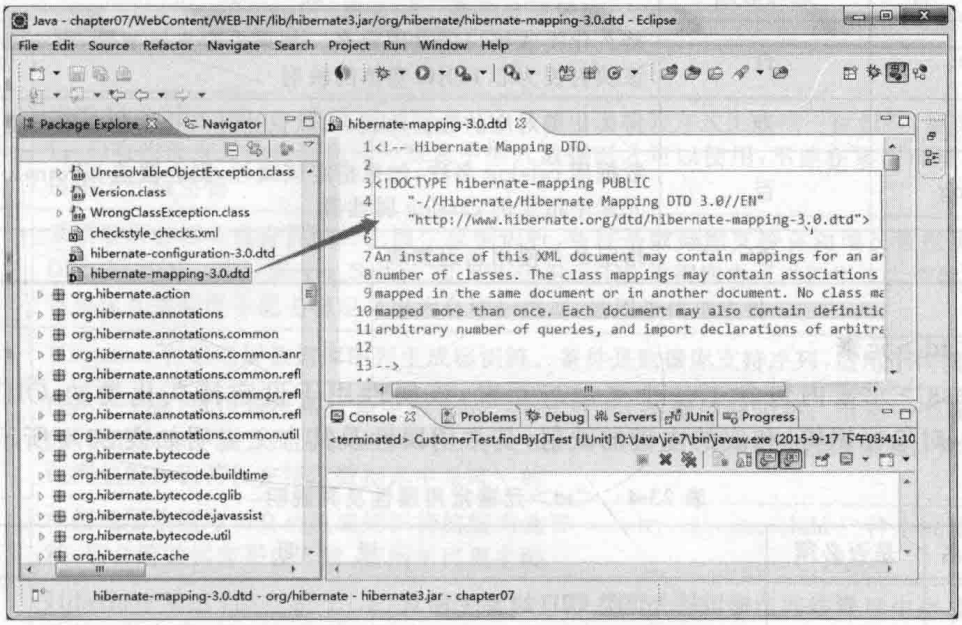


图 23-13　映射文件的 dtd 信息

在上述映射文件代码的 dtd 信息下面，就是 Hibernate 映射的具体配置。下面将对其配置中的每个元素的常用属性进行详细讲解。

1. <hibernate-mapping>元素

<hibernate-mapping>元素是映射文件的根元素，它所定义的属性在映射文件的所有节点都有效。<hibernate-mapping>元素所包含的常用属性及其含义说明如表 23-2 所示。

表 23-2　<hibernate-mapping>元素的常用属性及其说明

属 性 名	是否必须	说　明
package	否	为映射文件中的类指定一个包前缀，用于非全限定类名
schema	否	指定数据库 schema 名
catalog	否	指定数据库 catalog 名
default-access	否	指定 Hibernate 用于访问属性时所使用的策略，默认为 property。当 default- access="property"时，使用 getter 和 setter 方法访问成员变量；当 default- access="field"时，使用反射访问成员变量
default-cascade	否	指定默认的级联样式，默认为空
default-lazy	否	指定 Hibernate 默认所采用的延迟加载策略，默认为 true

2. <class>元素

<class>元素主要用于指定持久化类和数据表的映射关系，它是 XML 配置文件中的主要配置内容。其常用属性及其含义说明如表 23-3 所示。

表 23-3 <class>元素的常用属性及其说明

属 性 名	是否必须	说 明
name	否	持久化类或接口的全限定名。如果未定义该属性，则 Hibernate 将该映射视为非 POJO 实体的映射
table	否	持久化类对应的数据库表名，默认为持久化类的非限定类名
catalog	否	数据库 catalog 名称，如果指定该属性，则会覆盖 hibernate-mapping 元素中指定的 catalog 属性值
lazy	否	指定是否使用延迟加载

3. <id>元素

<class>元素内包含了一个<id>元素，该属性用于设定持久化类的 OID(Object identifier，对象标识符)和表的主键的映射，其常用属性及其含义说明如表 23-4 所示。

表 23-4 <id>元素常用属性及其说明

属 性 名	是否必须	说 明
name	否	标识持久化类 OID 的属性名
type	否	持久化类中标识属性的数据类型。如果没有为某个属性显式设定映射类型，Hibernate 会运用反射机制先识别出持久化类的特定属性的 Java 类型，然后自动使用与之对应的默认 Hibernate 映射类型。Java 的基本数据类型和包装类型对应相同的 Hibernate 映射类型，基本数据类型无法表达 null，所以对于持久化类的 OID，推荐使用包装类型(integer，long，string 等)
column	否	设置标识属性所映射的数据列的列名(主键字段的名字)
access	否	指定 Hibernate 对标识属性的访问策略，默认为 property。若此处指定了该属性，则会覆盖<hibernate-mapping>元素中指定的 default-access 属性

除了上面五个元素以外，<id>元素还可以包含一个可选的子元素<generator>。<generator>元素指定了主键的生成方式。对于不同的关系型数据库和业务应用来说，其主键的生成方式往往也不同，有的是依赖数据库自增字段生成主键，有的是按照具体的应用逻辑决定，通过<generator>元素就可以指定这些不同的实现方式。这些实现方式在 Hibernate 中，又称为主键生成策略。

在讲解 Hibernate 的主键生成策略之前，需要先了解两个概念：自然主键和代理主键。关于这两个概念的描述如下。

- 自然主键：把具有业务含义的字段作为主键，称为自然主键。例如在 user 表中，如果把 name 字段作为主键，其前提条件必须是：每一个用户的姓名不允许为 null，不允许用户重名，并且不允许修改用户姓名。尽管这些是可行的，但是不能满足不断变化的业务需求，一旦出现了允许用户重名的业务需求，就必须修改数据模型，重新定义表的主键，这给数据库的维护增加了难度。

- 代理主键：把不具备业务含义的字段作为主键，称为代理主键。该字段一般取名为 ID，通常为整数类型，因为整数类型比字符串类型可以节省更多的数据库空间。在上述例子中，显然更合理的方式是使用代理主键。

在 Hibernate 中，提供了几个内置的主键生成策略，其常用主键生成策略的名称和描述如表 23-5 所示。

表 23-5　主键生成策略

名　称	描　述
increment	用于 long、short 或 int 类型，由 Hibernate 自动以递增的方式生成唯一标识符，每次增量为 1。只有当没有其他进程向同一张表中插入数据时才可以使用，不能在集群环境下使用，适用于代理主键
identity	采用底层数据库自身提供的主键生成标识符，条件是数据库支持自动增长数据类型。在 DB2、MySQL、SQL Server、Sybase 和 HypersonicSQL 数据库中可以使用该生成器，该生成器要求在数据库中把主键定义成为自增长类型，适用于代理主键
sequence	Hibernate 根据底层数据库序列生成标识符。条件是数据库支持序列，适用于代理主键
hilo	使用一个高/低位算法高效地生成 long、short 或 int 类型的标识符。给定一个表和字段（默认的表和字段分别为 hibernate_unique_key 和 next_hi）作为高位值的来源。高/低位算法产生的标识符仅在特定数据库中是唯一的
native	根据底层数据库对自动生成标识符的能力选择 identity、sequence、hilo 三种生成器中的一种，适合跨数据库平台开发，适用于代理主键
uuid	Hibernate 采用 128 位的 UUID 算法生成标识符。该算法能够在网络环境中生成唯一的字符串标识符，其 UUID 被编码为一个长度为 32 位的十六进制字符串。这种策略并不流行，因为字符串类型的主键比整数类型的主键占用更多的数据库空间，适用于代理主键
assigned	由 java 程序负责生成标识符，如果不指定 id 元素的 generator 属性，则默认使用该主键生成策略，适用于自然主键

4. <property>元素

<class>元素内可以包含多个<property>子元素，它用于表示持久化类的其他属性和数据表中非主键字段的映射关系。<property>元素所包含的常用属性及其含义说明如表 23-6 所示。

表 23-6　<property>元素常用属性

属 性 名	说　明
name	持久化类属性的名称，以小写字母开头
column	数据表字段名
type	数据表的字段类型
length	数据表字段定义的长度
lazy	指定当持久化类的实例首次被访问时，是否对该属性使用延迟加载，其默认值是 false
unique	是否对映射列产生一个唯一性约束。常在产生 DDL 语句或创建数据库对象时使用
not-null	是否允许映射列为空

23.5.2 hibernate.cfg.xml 配置文件

Hibernate 的配置文件包含了数据库连接的相关信息，以及映射文件的基本信息。通常情况下，配置文件默认放在 src 目录下，发布后，该文件会在项目的 WEB-INF/classes 路径下。配置文件的常用配置信息如下所示：

```
<? xml version="1.0" encoding="UTF-8"? >
<!--配置文件的 dtd 信息-->
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!--指定方言-->
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <!--数据库驱动-->
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <!--连接数据库的 url-->
        <property name="hibernate.connection.url">
            jdbc:mysql:///hibernate
        </property>
        <!--数据库的用户名-->
        <property name="hibernate.connection.username">
            root
        </property>
        <!--数据库的密码-->
        <property name="hibernate.connection.password">
            root
        </property>
        <!--其他配置-->
        <!--显示 SQL 语句-->
        <property name="hibernate.show_sql">true</property>
        <!--格式化 SQL 语句-->
        <property name="format_sql">true</property>
        <!--用于关联 hbm 配置文件-->
        <mapping resource="com/itheima/domain/User.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

在上述代码中，首先是 xml 声明，然后是配置文件的 dtd 信息，该信息同样可以在核心包 hibernate3.jar 下的 org.hibernate 包中的 hibernate-configuration-3.0.dtd 文件中找到，

初学者只需要复制并使用即可。

Hibernate 配置文件的根元素是＜hibernate-configuration＞，该元素中包含一个＜session-factory＞子元素，在＜session-factory＞元素中又包含多个＜property＞元素，这些 property 元素用于对 Hibernate 连接数据库的一些信息进行配置。例如，数据库的方言、驱动、URL、用户名、密码等。＜mapping＞元素用于加载映射文件的信息。

Hibernate 配置文件的一些常用属性名称及用途如表 23-7 所示。

表 23-7　Hibernate 常用配置属性

名　称	描　述
hibernate. dialect	操作数据库方言
hibernate. connection. driver_class	连接数据库驱动程序
hibernate. connection. url	连接数据库 URL
hibernate. connection. username	数据库用户名
hibernate. connection. password	数据库密码
hibernate. show_sql	在控制台输出 SQL 语句
hibernate. format_sql	格式化控制台输出的 SQL 语句
hibernate. hbm2ddl. auto	当 SessionFactory 创建时是否根据映射文件自动验证表结构或自动创建、自动更新数据库表结构。该参数的取值为 validate、update、create 和 create-drop
hibernate. connection. autocommit	事务是否自动提交

Hibernate 实现了一种插件结构，它可以集成任何连接池软件。Hibernate 对 C3P0 连接池提供了内嵌支持，所以可以在 Hibernate 中直接配置和使用 C3P0。下面就介绍在 hibernate. cfg. xml 中如何配置和使用 C3P0。

首先，导入 C3P0 的 JAR 包(c3p0-0. 9. 1. jar)。此 JAR 包可以在已经下载的 hibernate 包的 lib 文件夹下的子目录 optional 中找到。

然后，在 hibernate. cfg. xml 中添加 C3P0 的配置信息，其添加代码如下所示：

```
<!--C3P0 连接池设定-->
<!--使用 C3P0 连接池配置连接池提供的供应商-->
<property name="connection.provider_class">
    org.hibernate.connection.C3P0ConnectionProvider
</property>
<!--在连接池中可用的数据库连接的最少数目-->
<property name="c3p0.min_size">5</property>
<!--在连接池中所有数据库连接的最大数目-->
<property name="c3p0.max_size">20</property>
<!--设定数据库连接的过期时间,以 ms 为单位, 如果连接池中的某个数据库连接空闲状态的时间
超过 timeout 时间,则会从连接池中清除-->
<property name="c3p0.timeout">120</property>
<!--每 3000s 检查所有连接池中的空闲连接 以 s 为单位-->
<property name="c3p0.idle_test_period">3000</property>
```

在上述配置代码中，首先声明了 C3P0 的供应商信息，然后配置了连接池中的连接最小数目和最大数目等。配置完这些信息后，C3P0 连接池就可以使用了。

最后，通过 JUnit 运行 UserTest 中的 testInsert()方法，如果执行成功，则表明 C3P0 连接池已经正确配置。

23.6 Hibernate 的核心 API

在 Hibernate 中，有六个常用的核心接口，它们分别是 Configuration、SessionFactory、Session、Transaction、Query 和 Criteria。在接下来的几个小节中，将对这六个核心接口进行详细讲解。

23.6.1 Configuration

Configuration 主要用于启动、加载和管理 Hibernate 的配置文件信息，在启动 Hibernate 的过程中，Configuration 实例首先确定 Hibernate 文件的位置，然后读取相关配置，最后创建一个唯一的 SessionFactory 实例。

Hibernate 通常使用 Configuration config＝new Configuration(). configure();的方式创建实例，此种方式默认会在 src 下读取 hibernate. cfg. xml 配置文件。如果不希望配置文件放置在 src 目录下，则可以在 configure()方法中传入一个参数指定文件位置，其代码如下所示：

```
Configuration config=new Configuration().configure("文件的位置");
```

此种写法中 Hibernate 会到指定的位置查询配置文件。例如，读取 src 下 config 包中的 hibernate. cfg. xml 文件，可以将代码写成如下形式：

```
Configuration config=new Configuration().configure("/config/hibernate.cfg.
xml");
```

需要注意的是，Configuration 对象只存在于系统的初始化阶段，它将 SessionFactory 创建完成后，就完成了自己的使命。

23.6.2 SessionFactory

SessionFactory 接口负责读取并解析映射文件，以及建立 Session 对象，它在 Hibernate 中起到一个缓冲区的作用，会将 Configuration 对象中的所有配置信息、Hibernate 自动生成的 SQL 语句以及某些可重复利用的数据加载到缓冲区中。同时，它还维护了 Hibernate 的二级缓存。

通常所使用的 SessionFactory 实例是通过 Configuration 对象获取的，其获取方法如下所示：

```
SessionFactory sessionFactory=config.buildSessionFactory();
```

SessionFactory 具有以下特点。

- 它是线程安全的，它的同一个实例能够供多个线程共享。
- 它是重量级的，不能随意创建和销毁它的实例。

由于 SessionFactory 是一个重量级的对象，占用的内存空间较大，所以通常情况下，一个应用程序只需要一个 SessionFactory 实例，只有应用中存在多个数据源时，才为每个数据源建立一个 SessionFactory 实例。为此，在实际开发时，通常会抽取出一个工具类提供 Session 对象。下面就介绍一个简单的抽取方式，如文件 23-5 所示。

文件 23-5　HibernateUtils.java

```
1  public class HibernateUtils {
2      //声明一个私有的静态 final 类型的 Configuration 对象
3      private static final Configuration config;
4      //声明一个私有的静态 final 类型的 SessionFactory 对象
5      private static final SessionFactory factory;
6      //通过静态代码块构建 SessionFactory
7      static {
8          config=new Configuration().configure();
9          factory=config.buildSessionFactory();
10     }
11     //提供一个公有的静态方法供外部获取，并返回一个 sesison 对象
12     public static Session getSession() {
13         return factory.openSession();
14     }
15 }
```

在文件 23-5 中，首先声明了一个私有的静态 final 类型的 Configuration 对象和 SessionFactory 对象，供类中的其他成员使用，下面通过静态方法构建了 SessionFactory 实例，最后提供了一个公有的静态方法供外部获取 session 对象。使用此工具类，就可以直接通过 HibernateUtils.getSession()的方式获取 session 对象。

23.6.3　Session

Session 是 Java 应用程序和 Hibernate 进行交互时所使用的主要接口，是持久化操作的核心 API。它主要用于读取、创建和删除映射对象的实例，这一系列的操作将被转换为数据表中的增加、修改、查询和删除操作。Session 是轻量级的，实例的创建和销毁不需要消耗太多的资源，同时它还是 Hibernate 的一级缓存，这个缓存主要用于存放当前工作单元加载的对象。

获取 Session 实例有两种方式，一种是通过 openSession()方法，另一种是通过 getCurrentSession()方法。两种方法获取 Session 的代码如下所示：

```
//采用 openSession 方法创建 session
Session session=sessionFactory.openSession();
//采用 getCurrentSession()方法创建 session
Session session=sessionFactory.getCurrentSession();
```

以上两种获取 Session 实例的主要区别是：采用 openSession()方法获取 Session 实例时，SessionFactory 直接创建一个新的 Session 实例，并且在使用完成后需要调用 close()方法进行手动关闭；而 getCurrentSession()方法创建的 Session 实例会被绑定到当前线程中，它在提交或回滚操作时会自动关闭。

在 Session 中，提供了多个持久化的操作方法，其常用方法如表 23-8 所示。

表 23-8 Session 中的常用方法

名　称	描　述	名　称	描　述
save()	用于执行添加对象操作	load()	根据主键查询数据
update()	用于执行修改对象操作	createQuery()	用于数据库操作对象
saveOrUpdate()	用于执行添加或修改对象操作	createSQLQuery()	用于数据库操作对象
delete()	用于执行删除对象操作	createCriteria()	面向对象的条件查询
get()	根据主键查询数据		

需要注意的是，Session 是线程不安全的，当多个并发线程同时操作一个 Session 实例时，就可能导致 Session 数据存取的混乱(当方法内部定义和使用 Session 时，不会出现线程问题)。因此设计软件架构时，应避免多个线程共享一个 Session 实例。

23.6.4 Transaction

Transaction 接口主要是用于管理事务，它是 Hibernate 的数据库事务接口，且对底层的事务接口进行了封装。Transaction 接口的实例对象是通过 Session 对象开启的，其开启方式如下所示：

```
tTransaction transaction=session.beginTransaction();
```

在 Transaction 接口中，提供了事务管理的常用方法，具体如下。

- commit()方法：提交相关联的 session 实例。
- rollback()方法：撤销事务操作。
- wasCommitted()方法：检查事务是否提交。

当应用程序执行完持久化操作后，需要调用 commit()方法提交事务，或者调用 rollback()方法回滚事务(取消事务)，可以通过以下代码加以理解：

```
try {
    transaction=session.beginTrsaction();          //开启事务
    session.save(user);                            //执行操作
    transaction.commit();                          //提交事务
}catch(Exception e) {
    transaction.rollback();                        //回滚事务
}finally {
    session.close();                               //关闭资源
}
```

从上述代码中可以看出，Session 执行完保存操作后，使用了 Transaction 接口的 commit()方法进行事务提交。只有事务执行提交后，才能真正地将数据操作同步到数据库中。在发生异常时，需要使用 rollback()方法进行事务回滚，以避免数据发生错误。

23.6.5　Query

Query 接口是 Hibernate 的查询接口，主要用于执行 Hibernate 的查询操作。Query 中包装了一个 HQL(Hibernate Query Language)查询语句，该语句采用了面向对象的查询方式，具有丰富灵活的查询特征。因此，Hibernate 官方推荐使用 HQL 语言进行查询。

在 Hibernate 中，使用 Query 对象的步骤如下。

(1) 获得 Hibernate Session 对象。

(2) 编写 HQL 语句。

(3) 调用 session. createQuery 创建查询对象。

(4) 如果 HQL 语句包含参数，则调用 Query 的 setXxx 设置参数。

(5) 调用 Query 对象的 list()或 uniqueResult()方法执行查询。

了解了使用 Query 对象的步骤后，接下来通过具体示例演示 Query 对象的查询操作。

(1) 向 user 表中插入 4 条数据，插入数据的 SQL 语句如下所示：

```
INSERT INTO user(id,NAME,age,gender) VALUES(1,'zhangsan',21,'男');
INSERT INTO user(id,NAME,age,gender) VALUES(2,'xiaozhang',21,'男');
INSERT INTO user(id,NAME,age,gender) VALUES(3,'lisisi',22,'女');
INSERT INTO user(id,NAME,age,gender) VALUES(4,'wangwu',19,'男');
```

插入后，user 表中的数据如图 23-14 所示。

	id	name	age	gender
□	1	zhangsan	21	男
□	2	xiaozhang	21	男
□	3	lisisi	22	女
□	4	wangwu	19	男
*	(NULL)	(NULL)	(NULL)	(NULL)

图 23-14　user 表中的数据

(2) 在 com. itheima. test 包中创建一个名称为 QueryTest 的类，在类中添加一个名称为 testFindAll()的方法，编辑后如文件 23-6 所示。

文件 23-6　QueryTest. java

```
1  package com.itheima.test;
2  import java.util.List;
3  import org.hibernate.Query;
4  import org.hibernate.Session;
5  import org.hibernate.SessionFactory;
6  import org.hibernate.Transaction;
```

```
7  import org.hibernate.cfg.Configuration;
8  import org.junit.Test;
9  import com.itheima.domain.User;
10 public class QueryTest {
11     @Test
12     public void testFindAll() {
13         Configuration config=new Configuration().configure();
14         SessionFactory sessionFactory=config.buildSessionFactory();
15         //1.得到一个 Session
16         Session session=sessionFactory.openSession();
17         Transaction transaction=session.beginTransaction();
18         //2.编写 HQL,其中的 User 代表的是类
19         String hql="from User";
20         //3.创建 Query 查询对象
21         Query query=session.createQuery(hql);
22         //4.使用 query.list()方法查询数据,并放入 list 集合
23         List<User>list=query.list();
24         for (User u : list) {
25             System.out.println(u);
26         }
27         transaction.commit();
28         session.close();
29         sessionFactory.close();
30     }
31 }
```

在文件 23-6 中,按照 Query 对象的使用步骤,对 user 表的数据进行了查询。使用 Junit 运行测试方法成功后,执行结果如图 23-15 所示。

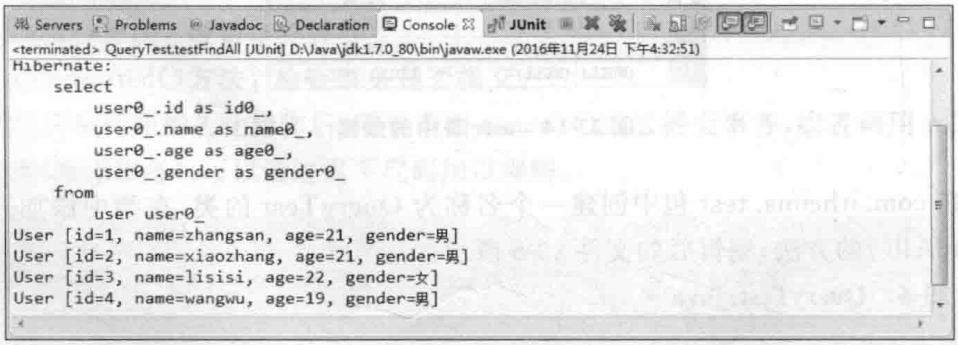


图 23-15 Query 查询输出结果

从图 23-15 中可以看出,user 表中的数据已经全部被查询并输出。

在 Query 接口中,除了上面代码中使用的 list()方法以外,还包含其他一些常用方法,如表 23-9 所示。

表 23-9　Query 的常用方法

名　称	描　述
setter	Query 接口中提供了一系列的 setter 方法用于设置查询语句中的参数，针对不同的数据类型，需要用到不同的 setter 方法
Iterator iterator()	该方法用于查询语句，返回的结果是一个 Iterator 对象，在读取时只能按照顺序方式读取，它仅把使用到的数据转换成 Java 实体对象
Object uniqueResult()	该方法用于返回唯一的结果，在确保只有一条记录的查询时可以使用该方法
int executeUpdate()	该方法是 Hibernate 3 的新特性，它支持 HQL 语句的更新和删除操作
Query setFirstResult(int firstResult)	该方法可以设置获取第一个记录的位置，也就是它表示从第几条记录开始查询，默认从 0 开始计算
Query setMaxResult(int maxResults)	该方法用于设置结果集的最大记录数，通常与 setFirstResult()方法结合使用，用于限制结果集的范围，以实现分页功能

23.6.6　Criteria

Criteria 接口是 Hibernate 提供的一个面向对象的查询条件接口，通过它完全不需要考虑数据库底层如何实现，以及 SQL 语句如何编写。Criteria 查询又称为 QBC 查询(Query By Criteria)，是 Hibernate 的另一种对象检索方式。

一个单独的查询就是 Criterion 接口的一个实例，用于限制 Criteria 对象的查询，在 Hibernate 中 Criterion 对象的创建通常是通过 Restrictions 工厂类完成的，它提供了一系列的条件查询方法，如表 23-10 所示。

表 23-10　Criteria 常用查询方法

名　称	描　述
Criteria add(Criterion criterion)	用于设置查询的条件，这个方法的参数为 Criterion 对象的实例
Criteria addOrder(Order order)	用于设置结果集的排序规则，其参数为一个 Order 对象实例
Criteria createCriteria(String associationPath)	用于创建一个新的 Criteria，这个方法用于执行符合查询时设置查询条件
List list()	用于执行数据库查询，返回查询的结果
Criteria setFirstResult(int firstResult)	设置获取第一个记录的位置，这个位置从 0 开始算
Object uniqueResult()	这个方法用于得到唯一的结果对象实例。在确保最多只有一个满足条件的查询结果时，可以选择适用这个方法

通常情况下，使用 Criteria 对象查询数据的主要步骤如下。

(1) 获得 Hibernate 的 Session 对象。

(2) 通过 Session 获得 Criteria 对象。

(3) 使用 Restrictions 的静态方法创建 Criterion 条件对象。Restrictions 类中提供了一系列用于设定查询条件的静态方法，这些静态方法都返回 Criterion 实例，每个 Criterion 实例代表一个查询条件。

(4) 向 Criteria 对象中添加 Criterion 查询条件。Criteria 的 add()方法用于加入查询条件。

(5) 执行 Criteria 的 list() 或 uniqueResult() 获得结果。

了解了 Criteria 对象的使用步骤后，下面通过案例演示如何使用 Criteria 对象实现查询操作。在 chapter23 项目的 com.itheima.test 包中新建一个名称为 CriteriaTest 的类，在类中添加一个名称为 testQBC()的方法，如文件 23-7 所示。

文件 23-7 CriteriaTest.java

```
1  package com.itheima.test;
2  import java.util.List;
3  import org.hibernate.Criteria;
4  import org.hibernate.Session;
5  import org.hibernate.SessionFactory;
6  import org.hibernate.Transaction;
7  import org.hibernate.cfg.Configuration;
8  import org.hibernate.criterion.Restrictions;
9  import org.junit.Test;
10 import com.itheima.domain.User;
11 public class CriteriaTest {
12     @Test
13     public void testQBC() {
14         Configuration config=new Configuration().configure();
15         SessionFactory sessionFactory=config.buildSessionFactory();
16         //1.得到一个 Session
17         Session session=sessionFactory.openSession();
18         Transaction transaction=session.beginTransaction();
19         //2.通过 session 获得 Criteria 对象
20         Criteria criteria=session.createCriteria(User.class);
21         //3.使用 Restrictions 的 eq 方法设定查询条件为 name="zhangsan",
22         //4.向 Criteria 对象中添加查询条件
23         criteria.add(Restrictions.eq("name","zhangsan"));
24         //5.执行 Criterita 的 list()方法获得结果
25         List<User>list=criteria.list();
26         for (User u : list) {
27             System.out.println(u);
28         }
29         transaction.commit();
30         session.close();
31         sessionFactory.close();
32     }
33 }
```

在文件 23-7 中，按照 Criteria 对象查询数据的步骤查询了 user 表中 name 等于 zhangsan 的数据。使用 JUnit 测试运行 testQBC()方法后，控制台的显示结果如图 23-16

所示。

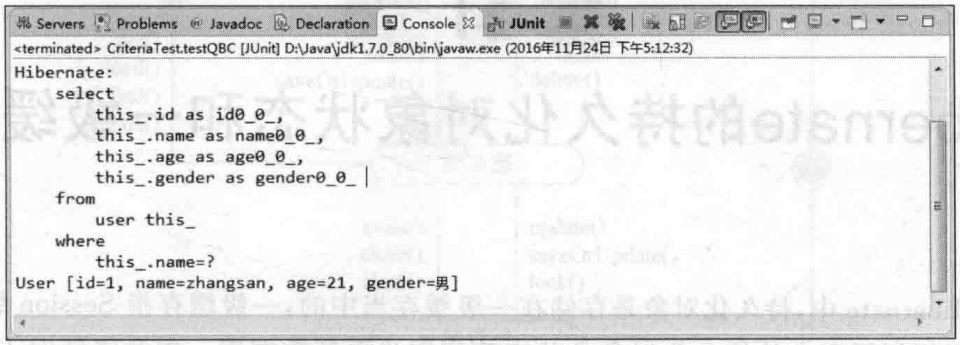


图 23-16　使用 Criteria 条件查询

从图 23-16 中可以看到，使用 Criteria 对象的查询方法，已将数据表中“name＝"zhangsan"”的数据查询输出。

第 24 章

Hibernate的持久化对象状态和一级缓存

在 Hibernate 中，持久化对象是存储在一级缓存当中的，一级缓存指 Session 级别的缓存，它可以根据缓存中的持久化对象的状态改变同步更新数据库。本章将针对 Hibernate 的持久化对象状态和一级缓存进行详细讲解。

24.1 Hibernate 持久化对象的状态

24.1.1 持久化对象的状态及状态转换

Hibernate 是持久层的 ORM 框架，专注于数据的持久化工作。在进行数据持久化操作时，持久化对象可能处于不同的状态当中。这些状态可分为三种，分别为瞬时态、持久态和脱管态。下面分别针对这三种状态进行简单介绍。

1. 瞬时态(transient)

瞬时态也称为临时态或者自由态，瞬时态的对象是由 new 关键字开辟内存空间的对象，不存在持久化标识 OID(相当于主键值)，且未与任何的 Session 实例相关联，在数据库中也没有记录，失去引用后将被 JVM 回收。瞬时对象在内存孤立存在，它是携带信息的载体，不和数据库的数据有任何关联关系。

2. 持久态(persistent)

持久态的对象存在一个持久化标识 OID，当对象加入到 Session 缓存中时，就与 Session 实例相关联。它在数据库中存在与之对应的记录，每条记录只对应唯一的持久化对象。需要注意的是，持久态对象是在事务还未提交前变成持久态的。

3. 脱管态(detached)

脱管态也称离线态或者游离态，当持久化对象与 Session 断开时就变成了脱管态，但是脱管态依然存在持久化标识 OID，只是失去了与当前 Session 的关联。需要注意的是，脱管态对象发生改变时 Hibernate 是不能检测到的。

在 Hibernate 运行时，持久化对象的三种状态是可以通过一系列的方法进行转换的。这三种状态之间的转换关系具体如图 24-1 所示。

从图 24-1 中可以看出，当一个对象通过 new 关键字创建后，该对象处于瞬时态；当对瞬时态对象执行 Session 的 save()或 saveOrUpdate()方法后，该对象将被放入 Session 的一级缓存中，此时该对象处于持久态；当对持久态对象执行 evict()、close()或 clear()操作后，对象会进入脱管态；当直接执行 Session 的 get()、load()、find()或 iterate()等方法从数据库中

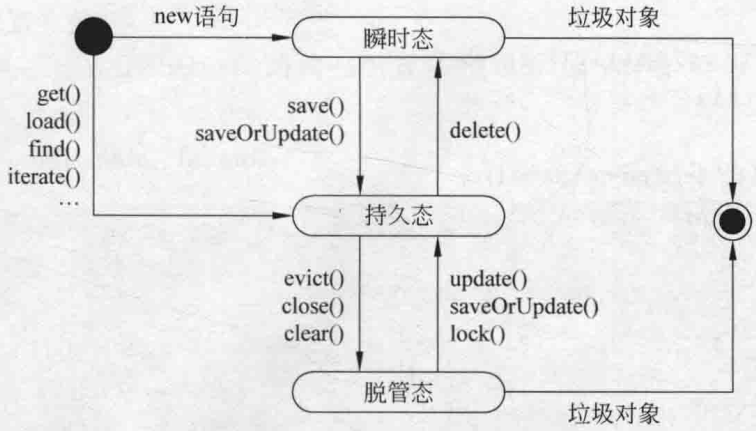


图 24-1　持久化对象的状态转换

查询出对象时，查询到的对象也会处于持久态；当对数据库中的纪录进行 update()、saveOrUpdate()以及 lock()等操作后，此时脱管态的对象就过渡到持久态；由于瞬时态和脱管态的对象不在 session 的管理范围内，所以会在一段时间后被 JVM 回收。

24.1.2　演示持久化对象的状态转换

通过第 24.1.1 节的学习，读者已经了解了持久化对象的三种状态，以及三种状态之间的转换方式。本节将通过案例演示这三种状态的转换。

1. 创建项目并导入 JAR 包

在 Eclipse 中创建一个名为 chapter24 的 Web 项目，将 Hibernate 所必需的 JAR 包添加到 WEB-INF/lib 目录中，并发布到类路径下，如图 24-2 所示。

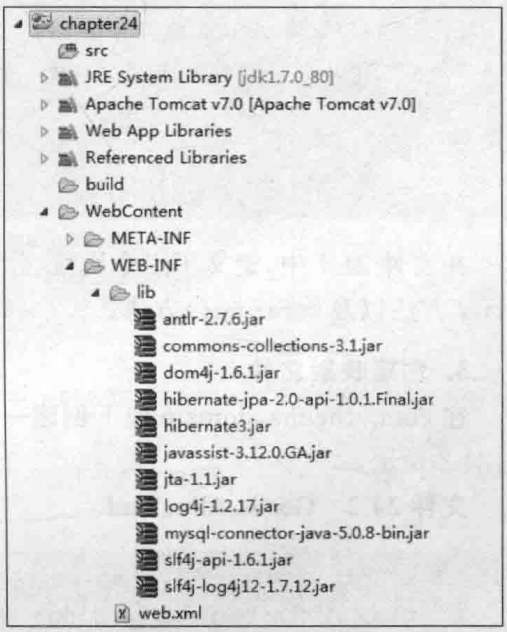


图 24-2　导入 Hibernate 框架所需的 JAR 文件

2. 创建实体类

在 src 目录下创建一个名为 com. itheima. domain 的包，并在该包下创建一个名为 Goods 的实体类，编辑后如文件 24-1 所示。

文件 24-1　Goods. java

```
1  package com.itheima.domain;
2  public class Goods {
3      private Integer id;                    //标识 id
4      private String name;                   //商品名称
5      private Double price;                  //商品价格
6      public Integer getId() {
7          return id;
```

```
    }
    public void setId(Integer id) {
        this.id=id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name=name;
    }
    public Double getPrice() {
        return price;
    }
    public void setPrice(Double price) {
        this.price=price;
    }
    //重写 toString()方法
    public String toString() {
        return "Goods [id="+id+", name="+name+", price="+price+"]";
    }
}
```

在文件 24-1 中，定义了三个属性，分别为 id、name 和 price，并提供了各属性的 getter 和 setter 方法以及 toString()方法。

3. 创建映射文件

在 com.itheima.domain 包下创建一个名为 Goods.hbm.xml 的映射文件，编辑后如文件 24-2 所示。

文件 24-2 Goods.hbm.xml

```
<? xml version="1.0" encoding="UTF-8"? >
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
<class name="com.itheima.domain.Goods" table="goods">
        <id name="id" column="id" type="integer" >
            <generator class="native"/>
        </id>
        <property name="name" column="name"/>
         <property name="price" column="price"/>
    </class>
</hibernate-mapping>
```

在文件 24-2 中可以看出，实体类 Goods 与所对应的是数据库中的 goods 表，实体类的属性与数据表的字段名相对应。

4. 创建配置文件

在 src 目录下创建 Hibernate 的核心配置文件 hibernate.cfg.xml，编辑后如文件 24-3 所示。

文件 24-3　hibernate.cfg.xml

```
<? xml version="1.0" encoding="UTF-8"? >
<!--配置文件的 dtd 信息-->
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <!--指定方言-->
        <property name="hibernate.dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <!--数据库驱动-->
        <property name="hibernate.connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <!--连接数据库的 url-->
        <property name="hibernate.connection.url">
            jdbc:mysql://localhost:3306/hibernate
        </property>
        <!--数据库的用户名-->
        <property name="hibernate.connection.username">root</property>
        <!--数据库的密码-->
        <property name="hibernate.connection.password">root</property>
        <!--其他配置-->
        <!--显示 SQL 语句-->
        <property name="hibernate.show_sql">true</property>
        <!--格式化 SQL 语句-->
        <property name="hibernate.format_sql">true</property>
        <!--自动建表-->
        <property name="hibernate.hbm2ddl.auto">update</property>
        <!--用于关联 hbm 配置文件-->
        <mapping resource="com/itheima/domain/Goods.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

在文件 24-3 中，除了配置数据库的基本信息、显示 SQL 语句、格式化 SQL 语句以及导入关联映射文件信息以外，还增加了一个自动建表的配置信息，该配置会自动检测数据库中是否存在对应的表，如果不存在，则会自动在数据库中创建数据表，反之则不创建。

5. 创建工具类

在 src 目录下创建一个名为 com. itheima. utils 的包，在该包下创建一个名为 HibernateUtils 的类，该类用于获取 Session 实例对象，其实现代码请参见本书第 23 章中的文件 23-5。

6. 创建测试类

在 src 目录下创建一个名为 com. itheima. test 的包，在该包下创建一个名为 GoodsTest 的类，编辑后如文件 24-4 所示。

文件 24-4 GoodsTest. java

```
1  package com.itheima.test;
2  import org.hibernate.Session;
3  import org.junit.Test;
4  import com.itheima.domain.Goods;
5  import com.itheima.utils.HibernateUtils;
6  public class GoodsTest {
7      //演示持久化对象的三种状态
8      @Test
9      public void test1() {
10         Session session=HibernateUtils.getSession();   //得到 session 对象
11         session.beginTransaction();                    //开启事务
12         Goods goods=new Goods();                       //瞬时态
13         goods.setName("铅笔");
14         goods.setPrice(0.5);
15         session.save(goods);                           //持久态
16         session.getTransaction().commit();             //提交事务
17         session.close();                               //关闭资源
18         System.out.println(goods);                     //脱管态
19     }
20 }
```

在文件 24-5 中，使用 new 关键字创建 Goods 对象时，该对象是没有标识(OID)的，也没有与 Session 进行关联，它处于瞬时状态。执行 save()方法后，Goods 对象已经处于 Session 的管理范围了，并且有了自己的 OID，此时的 Goods 对象已转换为持久态。当执行 commit()方法并关闭 Session 后，Goods 对象就已不在 Session 的管理范围，此时 Goods 对象从持久态转换为托管态。

7. 运行程序并查看结果

为了验证上面所讲解的内容，接下来在代码的第 15 行设置断点，用 Debug 方式执行 test1()方法，程序运行至断点位置，观察 Variables 窗口的变量值，如图 24-3 所示。

从图 24-3 中可以看出，此时 Goods 对象的 id 值为 null，按 F6 键进行单步跳过操作，save()方法执行后，继续观察变量的变化情况，如图 24-4 所示。

从图 24-4 中可以看出，执行完 save()方法后，Good 对象的 id 属性已被赋值，该值就是

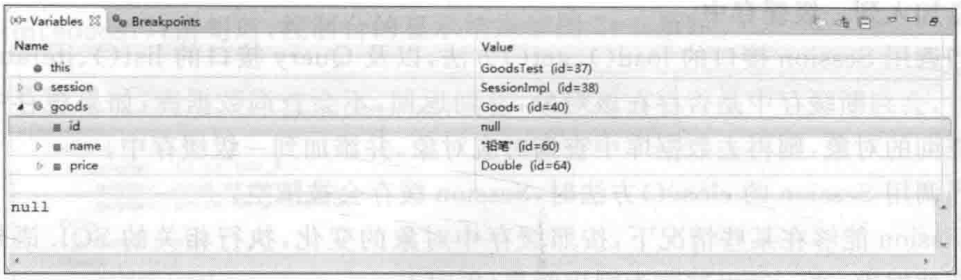


图 24-3　Variables 窗口

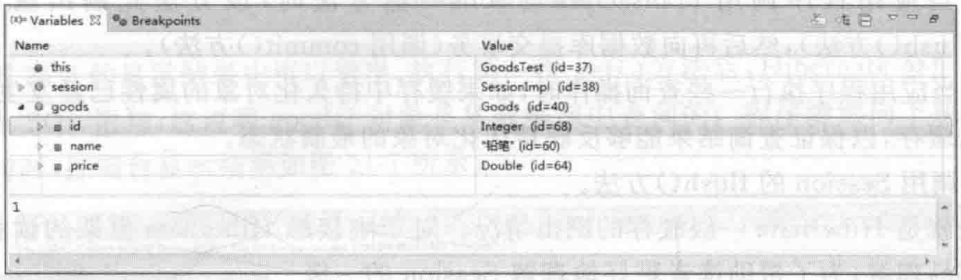


图 24-4　Variables 窗口

Goods 对象的持久化标识 OID，这说明持久化对象在事务提交前就已经变成了持久态，也说明了瞬时态对象和持久态对象的区别就是，持久态对象与 Session 进行了关联并且 OID 有值。当执行完 close()方法后，Goods 对象与 Session 不再存在关联关系，此时的 Goods 对象会由持久态对象转换为脱管态，但通过控制台的输出内容可以发现 Goods 对象的 OID 属性依然是存在的，这说明脱管态对象与持久态对象的区别就是脱管状态对象没有了 Session 关联。

24.2　Hibernate 的一级缓存

Hibernate 中的缓存分为一级缓存和二级缓存，这两个级别的缓存都位于持久化层，并且存储的都是数据库数据的备份。其中一级缓存是 Hibernate 的内置缓存，在前面的学习中已经使用过。在接下来的几节中将针对 Hibernate 的一级缓存进行详细讲解。

24.2.1　一级缓存

一级缓存其实就是 Session 缓存。Session 缓存是一块内存空间，用于存储与管理 Java 对象。在使用 Hibernate 查询对象时，首先会使用对象的 OID 值在 Hibernate 的一级缓存中查找，如果找到匹配的对象，则直接将该对象从一级缓存中取出使用；如果没有找到匹配的对象，则会去数据库中查询对应的数据。当从数据库中查询到所需数据时，该数据信息会存储到一级缓存中。由此可知，Hibernate 一级缓存的作用就是减少对数据库的访问次数。

Hibernate 的一级缓存具有如下特点。

当应用程序调用 Session 接口的 save()、update()、saveOrUpdate 时，如果 Session 缓存中没有相应的对象，则 Hibernate 就会自动把从数据库中查询到的相应对象信

息加入到一级缓存中。

- 当调用 Session 接口的 load()、get()方法，以及 Query 接口的 list()、iterator()方法时，会判断缓存中是否存在该对象，有则返回，不会查询数据库，如果缓存中没有要查询的对象，则再去数据库中查询对应对象，并添加到一级缓存中。
- 当调用 Session 的 close()方法时，Session 缓存会被清空。
- Session 能够在某些情况下，按照缓存中对象的变化，执行相关的 SQL 语句同步更新数据库，这一过程被称为刷出缓存(flush)。

在默认情况下，Session 在如下几种情况中会刷出缓存。

(1) 当应用程序调用 Transaction 的 commit()方法时，该方法先刷出缓存(调用 session.flush()方法)，然后再向数据库提交事务(调用 commit()方法)。

(2) 当应用程序执行一些查询操作时，如果缓存中持久化对象的属性已经发生了变化，会先刷出缓存，以保证查询结果能够反映持久化对象的最新状态。

(3) 调用 Session 的 flush()方法。

以上就是 Hibernate 一级缓存的刷出情况。对于刚接触 Hibernate 框架的读者来说并不是很容易理解，为了帮助读者更好地理解 Session 的一级缓存，下面通过具体案例演示一级缓存的使用。

以 goods 表为例，此时 goods 表中的数据如图 24-5 所示。

	id	name	price
□	1	铅笔	0.5

图 24-5 goods 表查询结果

在 GoodsTest 类中添加一个名为 test2()的方法，使用该方法证明一级缓存的存在，其代码如下所示：

```
//证明一级缓存的存在
@Test
public void test2() {
    Session session=HibernateUtils.getSession();
    session.beginTransaction();
    //获取 goods1 对象时，由于一级缓存中没有数据，所以会发送 SQL 语句，查询数据库中的数据
    Goods goods1= (Goods) session.get(Goods.class, 1);
    System.out.println(goods1);
    //获取 goods2 对象时，不会发出 SQL 语句，会从 Session 缓存中获取数据
    Goods goods2= (Goods) session.get(Goods.class, 1);
    System.out.println(goods2);
    session.getTransaction().commit();
    session.close();
}
```

在上述代码中，第一次执行 get()方法时，由于一级缓存中不存在 id 为 1 的 Goods 对象，所以 Hibernate 会发出 SQL 语句查询数据库中 id 为 1 的数据。查到后，会将该数据信息保存到一级缓存中。当再次调用 get()方法获取该对象时，此时将不会发出 SQL 语句，这是因为该对象是在一级缓存中获取的。

为了验证上面的描述，接下来在 Goods goods1=(Goods) session.get (Goods.class, 1);这一行设置断点，用 Debug 方式执行 test2()方法。程序进入断点后，按 F6 键执行 System.

out.println(goods1);语句后，控制台的显示结果如图 24-6 所示。

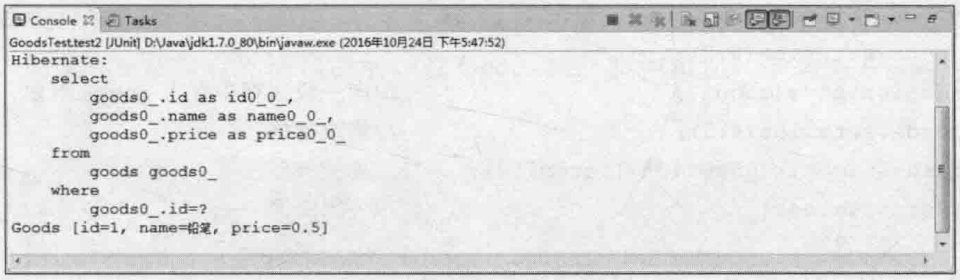


图 24-6　输出结果

从图 24-6 的显示结果中可以看出，执行第一次 get()方法后，Hibernate 会向数据库中发出一条 SQL 语句，这说明 goods1 对象是从数据库中查询的。程序继续向下执行第二个输出语句时，控制台显示结果如图 24-7 所示。

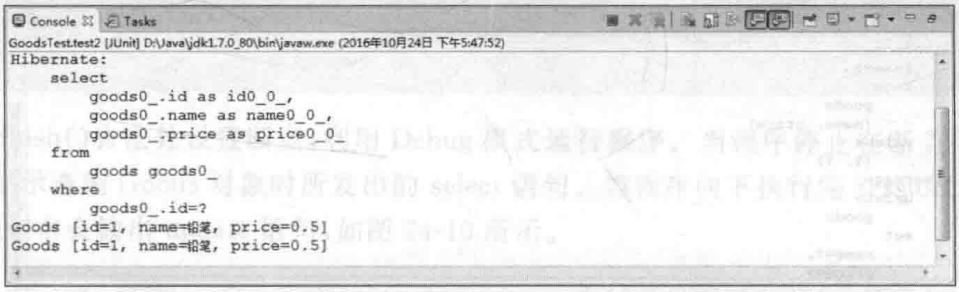


图 24-7　输出结果

从图 24-7 的显示结果中可以看出，goods2 对象的查询结果被直接输出了，这说明第二次调用 get()方法获取 goods2 对象时没有查询数据库，而是直接从一级缓存中获取了 goods2 对象。

24.2.2　Hibernate 的快照技术

通过第 24.2.1 节的学习，读者对 Hibernate 的一级缓存已经有了一定的了解。为了确保一级缓存中的数据和数据库中的数据保持一致，在 Hibernate 框架中提供了快照技术。

Hibernate 向一级缓存中存入数据的同时，还会复制一份数据存入 Hibernate 快照中。当调用 commit()方法时，会清理一级缓存中的数据(flush 操作)，同时会检测一级缓存中的数据和快照区的数据是否相同。如果不同，则会执行 update()方法，将一级缓存的数据同步到数据库中，并更新快照区；反之，则不会执行 update()方法。

下面通过具体示例演示 Hibernate 快照技术的应用。在文件 24-4 中添加一个名为 test3()的方法，其代码如下所示：

```
//hibernate 快照
@Test
public void test3() {
    Session session=HibernateUtils.getSession();
    session.beginTransaction();
```

```
        Goods goods=new Goods();
        goods.setName("钢笔");
        goods.setPrice(5.0);
        session.save(goods);                    //向一级缓存中存入 goods 对象
        goods.setPrice(4.5);                    //修改价格
        session.getTransaction().commit();      //提交事务
        session.close();                        //关闭资源
    }
```

在上述代码中，创建了一个 Goods 对象，并为该对象的属性进行赋值。在执行 save() 方法时，会将数据保存到一级缓存中。接着修改对象的 price 属性值，提交事务并关闭 session 对象。

使用 JUnit 测试运行 test3()方法后，控制台的显示结果如图 24-8 所示。

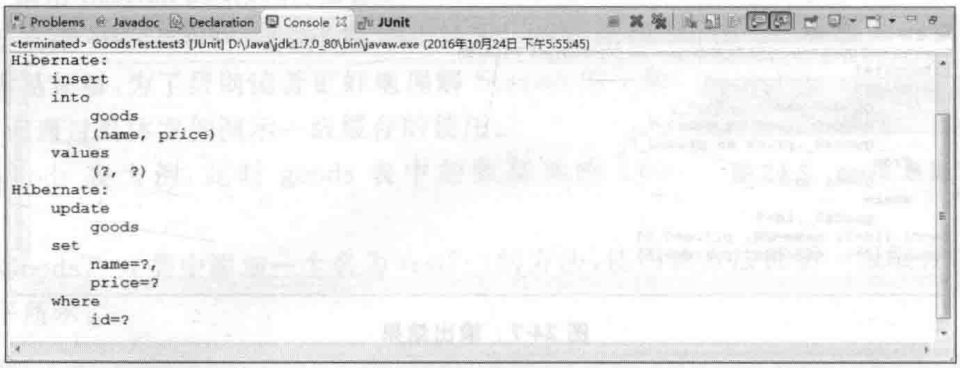


图 24-8 输出结果

从图 24-8 的显示结果中可以看出，在执行上述 test3()方法时共发出了两条 SQL 语句。如果使用 Debug 模式调试执行程序，会发现在执行 save()方法后发出了第一条 insert 语句，直到提交事务后才会发出第二条 update 语句。

此时读者一定会费解，为什么会发出 update 语句呢？这是因为在执行 session 的 save()方法时，Hibernate 一级缓存中保存了 Goods 对象的数据，并且会复制一份数据放入到 Hibernate 快照中，在执行 commit()方法时，Hibernate 会检测快照中的数据与一级缓存中的数据是否一致，由于在此之前 Goods 对象的 price 属性值发生了变化，导致快照中的数据与一级缓存中的数据已经不一致，因此发送了 update 语句，更新了 Hibernate 一级缓存中的数据。

此时，goods 表中数据如图 24-9 所示。

	id	name	price
☐	1	铅笔	0.5
☐	2	钢笔	4.5

图 24-9 数据查询结果

从图 24-9 的查询结果中可以看出，goods 表中插入的数据是一级缓存中更新后的数据。

24.2.3 一级缓存常用操作

在 Hibernate 中，一级缓存有三个常见的操作，分别为刷出、清除和刷新操作，下面结合本章前面的案例演示这三种常见的操作。

1. 刷出(flush)

一级缓存刷出功能是指调用 Session 的 flush()方法时会执行刷出缓存的操作。下面通过具体示例演示一级缓存的刷出功能。在 GoodsTest 类中添加一个名为 test4()的方法，具体代码如下所示：

```
//刷出
@Test
public void test4() {
    Session session=HibernateUtils.getSession();
    session.beginTransaction();
    Goods goods= (Goods) session.get(Goods.class, 2);
    goods.setPrice(5.5);
    session.flush();                    //执行刷出操作,此时会发送 update 语句
    session.getTransaction().commit();
    session.close();
}
```

在 flush()方法处设置断点，利用 Debug 模式运行程序。当程序停止在断点处时，控制台中会显示查询 Goods 对象时所发出的 select 语句。当程序向下执行完 session. flush()语句时，控制台会输出 update 语句，如图 24-10 所示。

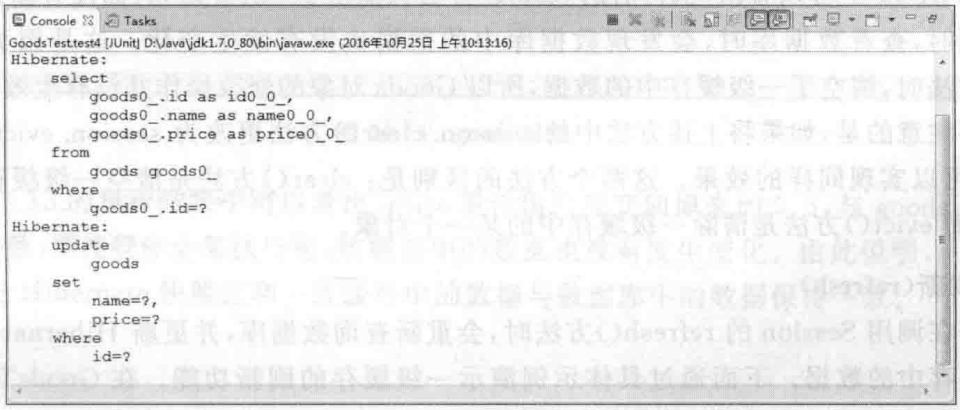


图 24-10　断点执行和输出结果

在第 24.2.2 节执行 test3()方法中的 commit()方法时，也同样发出了 update 语句，这说明在提交事务前，Hibernate 程序会默认先执行 flush()方法。

2. 清除(clear)

程序在调用 Session 的 clear()方法时，可以执行清除缓存数据的操作。下面通过具体示例演示一级缓存中的清除功能，在 GoodsTest 类中添加一个名为 test5()的方法，其代码如下所示：

```
//清除
@Test
public void test5() {
```

```
    Session session=HibernateUtils.getSession();
    session.beginTransaction();
    Goods goods=(Goods) session.get(Goods.class, 2);
    System.out.println(goods);
    goods.setPrice(6.5);
    session.clear();                    //清空一级缓存
    session.getTransaction().commit();
    session.close();
}
```

使用 JUnit 测试运行 test5()方法后，控制台的显示结果如图 24-11 所示。

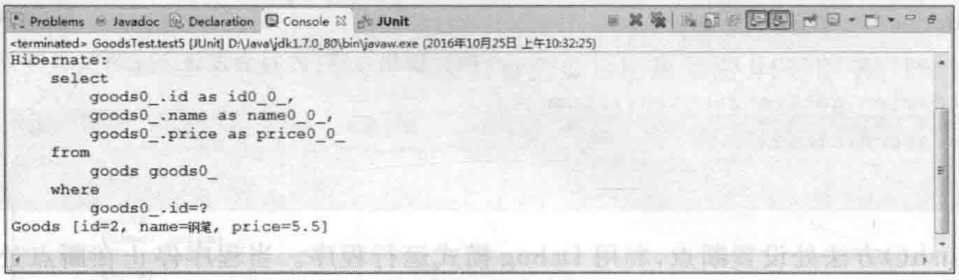


图 24-11 输出结果

从图 24-11 中可以看出，执行后的 test5()方法只输出了 select 语句，而没有输出 update 语句。同时，查看数据库时，会发现数据库中的数据也没有发生变化。这是因为在执行 clear()方法时，清空了一级缓存中的数据，所以 Goods 对象的修改操作并没有生效。

需要注意的是，如果将上述方法中的 session. clear()方法更改为 session. evict(goods)方法，也可以实现同样的效果。这两个方法的区别是：clear()方法是清空一级缓存中所有的数据，而 evict()方法是清除一级缓存中的某一个对象。

3. 刷新(refresh)

程序在调用 Session 的 refresh()方法时，会重新查询数据库，并更新 Hibernate 快照区和一级缓存中的数据。下面通过具体示例演示一级缓存的刷新功能。在 GoodsTest 类中添加一个名为 test6()的方法，其代码如下所示：

```
//刷新
@Test
public void test6() {
    Session session=HibernateUtils.getSession();
    session.beginTransaction();
    Goods goods=(Goods)session.get(Goods.class, 2);
    goods.setPrice(7.5);
    session.refresh(goods);              //查询数据库，恢复快照和一级缓存中的数据
    session.getTransaction().commit();
    session.close();
}
```

在 refresh()方法处设置断点，利用 Debug 模式调试程序，观察 Variables 窗口中 goods 对象的 price 属性值，如图 24-12 所示。

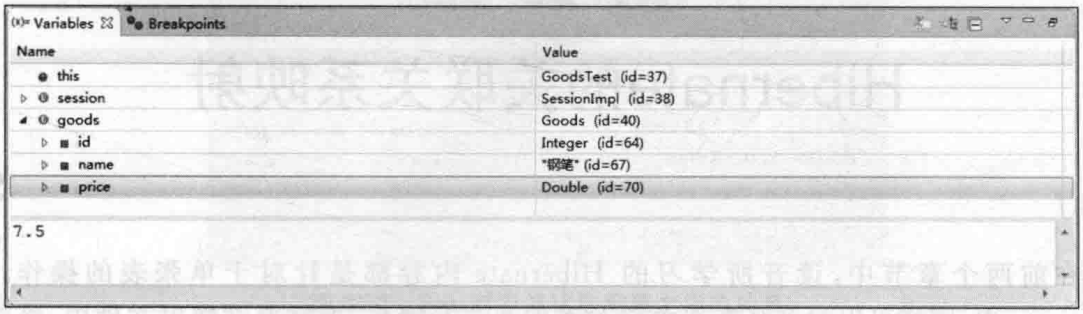


图 24-12　goods 对象

从图 24-12 的显示结果中可以看出，此时的 price 属性值为 7.5，按 F6 键继续向下执行，当执行完 refresh()方法后，Variables 窗口中 goods 对象的 price 属性值如图 24-13 所示。

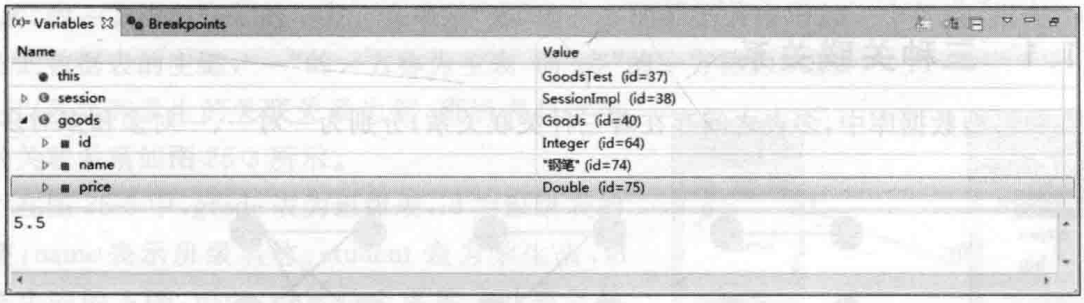


图 24-13　goods 对象

从图 24-13 的显示结果中可以看出，price 属性值已经变回原来的 5.5，与 goods 表中的数据保持一致，即使程序全部执行完，数据表中的数据也没有发生变化。由此说明，refresh()方法可以使 Hibernate 快照区和一级缓存中的数据与数据库中的数据保持一致。

第 25 章

Hibernate的关联关系映射

在前两个章节中，读音所学习的 Hibernate 内容都是针对于单张表的操作。在 Hibernate 中，同样可以对多张存在关联关系的表进行操作，但需要在映射文件中，将对象的关联关系与数据库表的外键关联进行映射。本章将对 Hibernate 的关联关系映射进行详细讲解。

25.1 关联关系映射

25.1.1 三种关联关系

在关系型数据库中，多表之间存在着三种关联关系，分别为一对一、一对多和多对多，如图 25-1 所示。

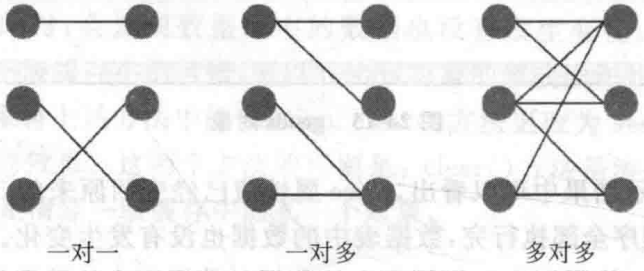


图 25-1 关系型数据库中多表之间的三种关联关系

这三种关联关系的具体说明如下。

- 一对一：在任意一方引入对方主键作为外键。
- 一对多：在“多”的一方，添加“一”的一方的主键作为外键。
- 多对多：产生中间关系表，引入两张表的主键作为外键，两个主键成为联合主键。

通过数据库表可以描述实体数据之间的关系，同样，通过对象也可以进行描述。在 Hibernate 中，使用 Java 实体对象之间的关系描述数据表之间的关系，如图 25-2 所示。

从图 25-2 中可以看出，一对一的关系就是在本类中定义对方类型的对象，如 A 类中定义 B 类类型的属性 b，B 类中定义 A 类类型的属性 a；一对多的关系，在一个 A 类类型对应多个 B 类类型的情况下，需要在 A 类以 Set 集合的方式引入 B 类类型的对象，在 B 类中定义 A 类类型的属性 a；多对多的关系，在 A 类中定义 B 类类型的 Set 集合，在 B 类中定义 A 类类型的 Set 集合，这里用 Set 集合的目的是避免数据的重复。

以上就是 Java 对象中三种实体类之间的关联关系，由于一对一的关联关系在开发中不

```
一对一                    一对多                    多对多

class A {                 class A {                 class A {
  B b;                      Set<B> bs ; // B的集合      Set<B> bs ; // B的集合
}                         }                         }

class B {                 class B {                 class B {
  A a;                      A a;                      Set<A> as ; // A的集合
}                         }                         }
```

图 25-2　Java 对象描述数据表之间的关系

常使用，所以本书中未单独讲解，读者作为了解即可。

25.1.2　一对多关联关系映射

在三种关联关系中，一对多（或多对一）是最为常见的一种关联关系，一对多映射关系是由“多”的一方指向“一”的一方。在表示“多”的一方的数据表中增加一个外键，指向“一”的一方的数据表的主键，“一”的一方称为主表，而“多”的一方称为从表。

以班级和学生的关联关系为例，班级表与学生表的关联关系如图 25-3 所示。

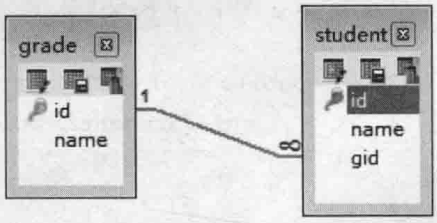


图 25-3　班级表和学生表的关联关系

在图 25-3 中，grade 表为班级表，id 为班级表的主键，name 表示班级名称；student 表为学生表，id 为学生表的主键，name 表示学生名称，gid 表示外键，它对应了班级表中的主键 id。由于一个班级可以拥有多个学生，那么可以说班级和学生是一对多的关系，在班级类中以集合 Set 的方式引入学生，在映射文件中通过＜set＞标签进行映射。

为了让读者快速掌握一对多关联关系映射的使用，下面通过具体的案例演示如何实现一对多关联关系映射。

1. 创建项目导入 JAR 包

在 Eclipse 中创建一个名为 chapter25 的 Web 项目，并将 Hibernate 所必需的 JAR 包添加到 WEB-INF/lib 目录中，并发布到类路径下。

2. 创建实体类

(1) 创建班级实体类。在项目的 src 目录下创建一个名为 com. itheima. onetomany 的包，在该包下创建一个 Grade 类，编辑后如文件 25-1 所示。

文件 25-1　Grade. java

```
1  package com.itheima.onetomany;
2  import java.util.HashSet;
```

```
3  import java.util.Set;
4  public class Grade {
5      private Integer id;
6      private String name;                    //班级名称
7      private Set<Student>students=new HashSet<Student>();
8      public Integer getId() {
9          return id;
10     }
11     public void setId(Integer id) {
12         this.id=id;
13     }
14     public String getName() {
15         return name;
16     }
17     public void setName(String name) {
18         this.name=name;
19     }
20     public Set<Student>getStudents() {
21         return students;
22     }
23     public void setStudents(Set<Student>students) {
24         this.students=students;
25     }
26 }
```

在文件 25-1 中，定义了三个属性，分别是 id、name 和 students，并提供了各属性的 getter 和 setter 方法。其中是 students 是一个集合类型对象，用于存储一个班级的学生。

(2) 创建学生实体类。在 com.itheima.onetomany 包下创建一个名为 Student 的类，编辑后如文件 25-2 所示。

文件 25-2 Student.java

```
1  package com.itheima.onetomany;
2  public class Student {
3      private Integer id;
4      private String name;              //学生名称
5      private Grade grade;              //学生从属于某个班级
6      public Integer getId() {
7          return id;
8      }
9      public void setId(Integer id) {
10         this.id=id;
11     }
```

```
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name=name;
    }
    public Grade getGrade() {
        return grade;
    }
    public void setGrade(Grade grade) {
        this.grade=grade;
    }
}
```

在文件 25-2 中，同样也定义了三个属性，分别是 id、name 和 grade，并提供了各属性的 getter 和 setter 方法。其中 grade 是一个 Grade 类型的对象，用于表示该学生从属于某一个班级。

3. 创建映射文件

(1) 创建班级类的映射文件。在 com. itheima. onetomany 包下创建一个名为 Grade. hbm. xml 的映射文件，如文件 25-3 所示。

文件 25-3　Grade. hbm. xml

```
<? xml version="1.0" encoding="UTF-8"? >
<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
        "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.itheima.onetomany.Grade" table="grade">
        <id name="id" column="id">
            <generator class="native"></generator>
        </id>
        <property name="name" column="name" length="40"/>
        <!--一对多的关系使用 set 集合映射-->
        <set name="students">
            <!--确定关联的外键列-->
            <key column="gid" />
            <!--映射到关联类属性-->
            <one-to-many class="com.itheima.onetomany.Student" />
        </set>
    </class>
</hibernate-mapping>
```

在文件 25-3 中，使用＜set＞标签描述被映射类中的 Set 集合，其中＜key＞标签的 column 属性用于确定关联的外键列，＜one-to-many＞标签用于描述持久化类的一对多关

联，其中 class 属性表示映射的关联类。

(2) 创建学生类的映射文件。在 com. itheima. onetomany 包下创建一个名为 Student. hbm. xml 的映射文件，如文件 25-4 所示。

文件 25-4 Student. hbm. xml

```
<? xml version="1.0" encoding="UTF-8"? >
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://www.hibernate.org/dtd/hibernate-mapping-3.0.dtd">
<hibernate-mapping>
    <class name="com.itheima.onetomany.Student" table="student">
        <id name="id" column="id">
            <generator class="native"/>
        </id>
        <property name="name" column="name" length="40"/>
        <!--多对一关系映射-->
        <many-to-one name="grade" class="com.itheima.onetomany.Grade"
        column="gid"/>
    </class>
</hibernate-mapping>
```

在文件 25-4 中，<many-to-one>标签定义了三个属性，分别是 name、class 和 column 属性。其中，name 属性表示 Student 类中的 grade 属性名称，class 属性表示指定映射的类，column 属性表示表中的外键类名。需要注意的是，该 column 属性与 Grade. hbm. xml 映射文件的<key>标签的 column 属性要保持一致。

4. 创建配置文件和工具类

在 src 目录下创建一个名为 hibernate. cfg. xml 的配置文件，在该文件行中配置数据库基本信息和其他配置信息后，将 Grade. hbm. xml 和 Student. hbm. xml 映射文件添加到配置文件中，具体如下所示：

```
<mapping resource="com/itheima/onetomany/Grade.hbm.xml" />
<mapping resource="com/itheima/onetomany/Student.hbm.xml" />
```

在 src 目录下创建一个名为 com. itheima. utils 的包，在包中创建工具类文件 HibernateUtils. java(可参见本书第 23 章中的文件 23-5)。

5. 创建测试类

在 com. itheima. onetomany 包下创建一个名为 OneToManyTest 的类，编辑后如文件 25-5 所示。

文件 25-5 OneToManyTest. java

```
package com.itheima.onetomany;
import org.hibernate.Session;
import org.junit.Test;
```

```
4  import com.itheima.utils.HibernateUtils;
5  public class OneToManyTest {
6      //添加数据
7      @Test
8      public void test1() {
9          Session session=HibernateUtils.getSession();
10         session.beginTransaction();
11         //创建一个班级
12         Grade g=new Grade();
13         //创建两个学生对象
14         g.setName("一班");
15         Student s1=new Student();
16         s1.setName("张三");
17         Student s2=new Student();
18         s2.setName("李四");
19         //描述关系---学生属于某个班级
20         s1.setGrade(g);
21         s2.setGrade(g);
22         //描述关系---班级里有多个学生
23         g.getStudents().add(s1);
24         g.getStudents().add(s2);
25         //先存班级再存学生
26         session.save(g);
27         session.save(s1);
28         session.save(s2);
29         session.getTransaction().commit();
30         session.close();
31     }
32 }
```

在文件 25-5 中，首先创建了一个班级对象和两个学生对象，然后分别描述了两个学生与班级之间的关系，又描述了一个班级与两个学生之间的关系，这就是一对多的双向关联关系。

6. 运行项目并查看结果

在 Hibernate 配置文件中添加自动建表信息后，在运行程序时，程序会自动创建两张表，并且插入数据。使用 JUnit 测试运行 test1()方法后，控制台显示结果如图 25-4 所示。

从图 25-4 的显示结果中可以看出，控制台总共输出了三条 insert 语句和两条 update 语句，此时查询两张表中的数据，如图 25-5 所示。

从图 25-5 查询结果中可以看出，grade 表和 student 表都已经成功创建，并插入了相应的数据。

在 Hibernate 中，像班级和学生这种关联关系，使用双向关联会执行两次 SQL 语句，这样会导致程序执行效率降低。如果将文件 25-5 中的第 23 行和第 24 行代码删除，并将