



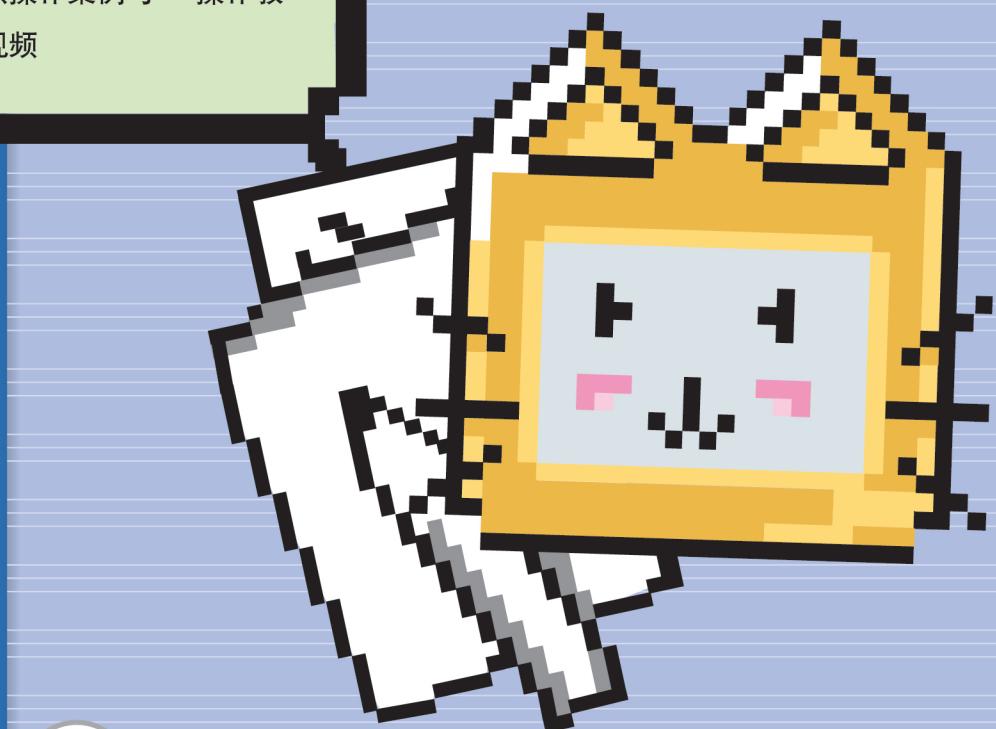
JWorld@TW技术论坛版主
Java权威技术顾问与专业讲师 林信良 全新改版！

JSP&Servlet

学习笔记(第2版)

- 实践教学经验集合
- 涵盖SCWCD考试范围
- Servlet 3.0新功能介绍
- 全新综合练习/微博开发
- 提供操作案例与IDE操作教学视频

林信良 编著



JSP & **Servlet**

学习笔记_(第 2 版)

林信良 编著

清华大学出版社

北京

内 容 简 介

本书针对 Servlet 3.0 的新功能全面改版，无论章节架构还是范例程序代码，都做了全面更新。书中详细介绍了 Servlet/JSP 与 Web 容器之间的关系，必要时从 Tomcat 源代码分析，了解 Servlet/JSP 如何与容器互动。本书还涵盖了文本处理、图片验证、自动登录、验证过滤器、压缩处理、JSTL 应用与操作等各种实用范例。

本书在讲解的过程中，以“微博”项目贯穿全书，随着每一章的讲述都在适当的时候将 JSP & Servlet 技术应用于“微博”程序之中，使读者能够了解完整的应用程序构建方法。

本书是作者多年来教学实践经验的总结，汇集了教学过程中学生在学习 JSP & Servlet 时遇到的概念、操作、应用或认证考试等各种问题及解决方案。本书适合 JSP & Servlet 初学者，以及广大的 JSP & Servlet 技术应用人员。

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 13501256678 13801310933

图书在版编目(CIP)数据

ISBN 978-7-302-28366-9

中国版本图书馆 CIP 数据核字(2012)第 号

责任编辑：王 定

封面设计：久久度文化

版式设计：康 博

责任校对：成凤进

责任印制：

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

印 刷 者：

装 订 者：

经 销：全国新华书店

开 本：185×260 印 张： 字 数：千字

版 次：2012 年 4 月第 1 版 印 次：2012 年 4 月第 1 次印刷

印 数：1~

定 价：58.00 元

产品编号：

导 读

这份导读可以让你了解如何使用本书。

字体

本书内文中与代码相关的文字，都用等宽字体来加以呈现，以与一般名词相区别。例如，JSP 是一般名词，而 `HttpServlet` 类为代码相关文字，使用了等宽字体。

程序范例



本书中许多范例都使用完整的程序实现来展现，如果是用以下方式示范程序代码：

FirstServlet HelloServlet.java

```
package cc.openhome;

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

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

@WebServlet("/hello.view")
public class HelloServlet extends HttpServlet { ← ① 继承 HttpServlet
    @Override
    protected void doGet(HttpServletRequest request, ← ② 重新定义 doGet()
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8"); ← ③ 设定响应内容类型
        PrintWriter out = response.getWriter(); ← ④ 取得回应输出对象
        String name = request.getParameter("name"); ← ⑤ 取得请求参数

        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello Servlet</title>");
        out.println("</head>");
        out.println("<body>");
```

```

        out.println("<h1> Hello! " + name + " !</h1>");
        out.println("</body>");
        out.println("</html>");

        out.close();
    }
}

```

范例开始的左边名称为 FirstServlet，表示可以在本书配套光盘的 samples 文件夹中查找相应章节目录，即可找到对应的 FirstServlet 项目，而右边名称为 HelloServlet.java，表示可以在项目中找到 HelloServlet.java 文件。如果代码中出现标号与提示文字，表示后续的内文中会有对应于标号及提示的更详细说明。

原则上，建议每个项目范例都亲自动手编写，但如果由于教学时间或实现时间上的考量，本书有建议进行的练习，如果在范例开始前有个  图示，表示建议动手实践，而且在本书配套光盘的 labs 文件夹中会有练习项目的基础，可以导入项目后，完成项目中遗漏或必须补齐的代码或设定。

如果文中使用以下的代码形成呈现，则表示它是一个完整的程序内容，不是项目的一部分，主要用来展现一个完整的文件如何编写：

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<html>
<head>
    <title>SimpleJSP</title>
</head>
<body>
    <h1><%= new java.util.Date() %></h1>
</body>
</html>
```

如果文中使用以下的代码形式呈现，则表示它是一个程序代码片段，主要展现程序编写时需要特别注意的片段：

```

// 略 ...
public void _jspService(HttpServletRequest request,
                        HttpServletResponse response)
    throws java.io.IOException, ServletException {
// 略...
try {
    response.setContentType("text/html;charset=UTF-8");
    //略...
    out = pageContext.getOut();
    // 略...
} catch (Throwable t) {
    // 略 ...
} finally {
    // 略 ...
}

```

由于受书籍页面宽度的限制，有些过长的程序代码可能会在一行容纳不下，不得不隔行表示，此时会使用箭头符号表示两行实际上是必须连接在一起的。例如：

JDBC Demo context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/JDBCDemo">
    <Resource name="jdbc/demo"
        auth="Container" type="javax.sql.DataSource"
        maxActive="100" maxIdle="30" maxWait="10000" username="root"
        password="123456" driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/demo?
            useUnicode=true&characterEncoding=UTF8"/>
</Context>
```

在上面的程序代码片段中，在 url 属性的设定中，完整的设置其实是 jdbc:mysql://localhost:3306/demo?useUnicode=true&characterEncoding=UTF8，当中不可以中断。

操作步骤

本书将 IDE 设定的相关操作步骤，也作为练习的一部分，你会看到如下的操作步骤说明：



- (1) 选择 File/New/Dynamic Web Project 命令，在出现的 New Dynamic Web Project 对话框的 Project name 文本框中输入 FirstServlet。
- (2) 确定 Target runtime 为刚才设置的 Apache Tomcat v7.0，单击 Finish 按钮。
- (3) 展开新建项目中的 Java Resources 节点，在 src 上单击鼠标右键，从弹出的快捷菜单中选择 New/Servlet 命令。
- (4) 在弹出的 Create Servlet 对话框的 Java package 文本框中输入 cc.openhome，在 Class name 文本框中输入 HelloServlet，单击 Next 按钮。
- (5) 选择 URL mappings 中的 HelloServlet，单击右边的 Edit 按钮，将 Pattern 改为 /hello.view 后，单击 OK 按钮。
- (6) 单击 Create Servlet 对话框中的 Finish 按钮。

如果操作步骤旁有个 图示，表示光盘的 videos 文件夹中对应的章节文件夹有操作步骤的录像，可打开观看以更了解实际操作过程。

特殊段落

在本书中会出现以下特殊段落：

提示»» 针对课程中提到的概念，提供一些额外的资源或思考方向，暂时忽略这些提示对课程进行并没有影响，但有时间的话，针对这些提示多作阅读、思考或讨论是有帮助的。

注意»» 针对课程中提到的概念，以特殊段落方式特别呈现出必须注意的一些使用方式、陷阱或避开问题的方法，看到这个特殊段落时请集中精神阅读。

综合练习

本书以“微博”项目作为范例贯穿全书，随着每一章的进行，都会在适当的时候将新学习到的技术，应用至“微博”程序之中并作适当的修改，以了解完整的应用程序基本上是如何建构出来的。

附 录

本书配套光盘中包括本书所有的范例，提供 Eclipse 范例项目，附录 A 说明如何使用这些范例项目，本书也说明如何在 Web 应用程序中整合数据库，范例中使用的数据库为 MySQL；附录 B 包括了 MySQL 的入门简介。

关于认证

本书涵盖了 Oracle Certified Professional, Java Platform, Enterprise Edition JavaServer Pages and Servlet Developer 考试范围，也就是原 Sun Certified Web Component Developer (SCWCD)，不过第 9 章整合数据库与第 11 章 JavaMail 入门不在考试范围，只是为了 Web 应用程序相关技术范围完整性而作介绍。

关于 Java 认证介绍，建议直接参考 Oracle University 网站上的认证介绍：

http://education.oracle.com/pls/web_prod-plq-dad/db_pages.getpage?page_id=140

每章最后的“重点复习”是针对该章的重要提示，可作为考前复习时使用。每章都会有“课后练习”，与认证相关的是选择题，分为单选、复选题两种形式，实训题是与每个章节相关的程序练习。

联系作者

若有勘误反馈等相关书籍问题，可通过网站与作者联系：

<http://openhome.cc>

第 1 版序

在完成本书之前，意外翻出了这张车票，94.08.15 从高雄到台北的座位证！一时之间还想不起这张车票是哪里来的，反倒是我老婆提醒了我，不过我却想起了更久之前的事情……

大学时代参加的社团是社会服务团，寒、暑假时会到一些地方举办营队，在学期中，即将参与营队的队友们必须负责各自的课程、准备教材、设计教具、验收教案等，出营队时则上台实行课程。

除了社团之外，自己平常也爱写些东西。大学时代正值 WWW 兴起之时，自己学会如何写 HTML，也常将学计算机时的心得写下来放到 Web 上，像如何安装 Apache、CGI 留言版之类的，说来写作的习惯应该是从那时养成的。

在大学最后一年考完硕士研究生入学考试之后，我在 BBS 的 Job 版上发现了几个短期打工需求，有一天接到一个电话，问我是否想写书。虽然主题只是网页制作，但第一次要写完整的一本书，合同载明页数必须有 400 页以上，着实有很大的压力，甚至还因此失眠了好几次，所幸在当兵前夕完成了这本书，成了我的第一本著作。事后在市面上发现，这本书重印了 4 次，心里还蛮感到安慰的。

当兵期间所属的单位是学校，平时除了连队勤务或卫哨之外，所做的事就是协助教官编写教材、教案、上课担任助教等。退伍后的第一份工作是在高雄，公司的业务之一是出版计算机图书，因为早有写作及出版图书的经验，自然也在公司的名义之下写了几本书。

2003 年 3 月底，开始将一些东西以“良葛格学习笔记”的名称放在网络上，随着时间的累积，伴随着网络传播的力量，越来越多人知道了这个网站的存在，我也在网络上结交了许多朋友，并因此得以在 Javaworld@TW 前站长林上杰 (Browser)先生的介绍下，认识了暮峰编辑江佳慧 (Novia) 小姐，出版了第二本有个人名号的书籍。

想到这里，发现在我过去的经验中，怎么都跟上台、写作、课程有关？还有一点不知道是否也有相关，我岳父岳母也都是老师……

这就是看到 94.08.15 从高雄到台北的座位证时，突然涌出的一连串回忆……

1994 年 8 月 15 日是什么日子？隔两天就是“2005 Java TWO 社区大会”！这张车票是为了参加 Java TWO 大会而买的。这是我第一次参加 Java TWO 大会，目的之一是想看看许多网络上认识的但未曾谋面的朋友，还有一个原因是暮峰也在大

会上设摊，其中有卖我的书，想去看看反应如何……在大会上，碰到了王森(Moli)先生，他跟我说：“想要请你帮忙写个教材……”，不过那时场面很混乱，反正就是一堆人哈啦来哈啦去的，话题很难继续，直到后来出现了 Moli 先生想加我的 MSN，哈啦过后，才确定这件事是真的！

之后又因为一连串的因缘机会，开始了我江湖卖艺……呃……讲课的日子！时光匆匆，岁月如梭……转眼来到了 2009 年 3 月，Novia 小姐问我有没有新的写作计划？我想了一下，这些日子以来，许许多多的授课经验积累了不少的想法，也了解了不少学员在实际学习时所遇到的问题，不如写下来吧！而这些写下来的东西就成了你眼前的这本书(篇幅有限，这本书只针对 Servlet / JSP)！

我不太知道人有没有宿命，但回首时总会发现许多的巧合，过去的种种经验，好像是在为了将来的某个事件而准备似的。当然，你也可以说，这是因为回忆是选择性地挑选拼凑而成的。无论如何，这些事情过去总得发生过，未来的你才有的拼凑。

一张车票引发了一连串回忆，也终于知道要怎么写这本书的序了，这本书就是这么来的……

林信良

2009 年 5 月 26 日

第 2 版序

“序”应该表达些什么？写一本书的动机？写一本书的过程？写完一本书的感想？

在本书第 1 版手稿完成后，思考着如何写序的那几天，在整理旧书时从一本书中掉出了一张车票，于是我写了一张车票引发一连串回忆的故事。在本书第 2 版手稿完成后，思考着如何写序的这几天，我回顾改版的这段漫长日子，想着一脚踏入陌生领域、探索一切未知的过程。

现在的你，也许在某个领域有擅长的事务，有没有想过，或许哪天，你会接触另一个完全未知的世界，到时候，你会怎么办？

我在信息领域的知识，大多都是自学而来，对于信息领域知识的搜寻、过滤、验证与实践，自认为颇有心得，改版过程中，乍然面对一切毫无所知的世界，也曾一度乱了手脚。某个下午带着慌乱的心路过了书局，突然心里有了答案：“我一直认为收集与过滤是我最大的能力，不用在这个时候，那要用在什么时候？”

你有没有听过类似的事呢？某人拥有高学历，却在生完小孩之后，毅然决然在家带小孩，某人在某领域拥有很好的经历，却在大家觉得他即将迈向巅峰时，投入另一个领域重新开始。像这类的情况，旁人通常都会为他们可惜。

我面对着完全未知的世界，开始发挥大量阅读的能力，极尽可能地寻找相关的书籍，在网络上搜寻各种相关资料，逐步勾勒出这个世界应有的方向，就如同当初从电机转换入信息，一切从未知开始累积，一切从头开始建立基础，既然是初学者，那就一切从头开始建构。

高学历带小孩不好吗？也许是自愿或被迫这么做，但如果可以发挥出高学历下该有的学习态度，好好学习如何让小孩子健康、快乐成长，那不也是件好事吗？放弃原有领域的经历不好吗？把建立原有领域经历的方式应用在新领域经历的建立，因此而有所成就的案例也不在少数！

一切都是动心转念之间，无论如何，保有一颗初学者的心，保有一颗赤子之心，放下熟悉领域拥有的一切，重新出发，方向就会逐步建立，所有的基础，后续的成就，就交由时间慢慢验证。

林信良

2011 年 5 月 26 日

目 录

| | |
|--------------------------------------------------------|----|
| 第 1 章 Web 应用程序简介 | 1 |
| 1.1 Web 应用程序基础知识..... | 2 |
| 1.1.1 关于 HTML..... | 2 |
| 1.1.2 URL、URN 与 URI..... | 3 |
| 1.1.3 关于 HTTP..... | 5 |
| 1.1.4 有关 URL 编码..... | 9 |
| 1.1.5 动态网页与静态网页 | 11 |
| 1.2 Servlet/JSP 简介..... | 13 |
| 1.2.1 何谓 Web 容器..... | 13 |
| 1.2.2 Servlet 与 JSP 的关系..... | 15 |
| 1.2.3 关于 MVC/Model 2..... | 18 |
| 1.2.4 Java EE 简介 | 21 |
| 1.3 重点复习 | 22 |
| 1.4 课后练习 | 23 |
| 第 2 章 编写与设置 Servlet | 25 |
| 2.1 第一个 Servlet..... | 26 |
| 2.1.1 准备开发环境 | 26 |
| 2.1.2 第一个 Servlet 程序 | 28 |
| 2.2 在 HelloServlet 之后 | 31 |
| 2.2.1 关于 HttpServlet | 31 |
| 2.2.2 使用 @WebServlet | 33 |
| 2.2.3 使用 web.xml | 34 |
| 2.2.4 文件组织与部署 | 36 |
| 2.3 进阶部署设置 | 37 |
| 2.3.1 URL 模式设置 | 38 |
| 2.3.2 Web 目录结构 | 40 |
| 2.3.3 使用 web-fragment.xml | 42 |
| 2.4 重点复习 | 45 |
| 2.5 课后练习 | 46 |
| 第 3 章 请求与响应 | 48 |
| 3.1 从容器到 HttpServlet | 49 |
| 3.1.1 Web 容器做了什么 | 49 |
| 3.1.2 doXXX()方法 | 51 |
| 3.2 关于 HttpServletRequest | 54 |
| 3.2.1 处理请求参数与标头 | 54 |
| 3.2.2 请求参数编码处理 | 57 |
| 3.2.3 getReader()、getInputStream() 读取 Body 内容 | 60 |
| 3.2.4 getPart()、getParts()取得上传 文件 | 64 |
| 3.2.5 使用 RequestDispatcher 调派 请求 | 69 |
| 3.3 关于 HttpServletResponse | 75 |
| 3.3.1 设置响应标头、缓冲区 | 76 |
| 3.3.2 使用 getWriter()输出字符 | 77 |
| 3.3.3 使用 getOutputStream()输出 二进制字符 | 80 |
| 3.3.4 使用 sendRedirect(), sendError() | 82 |
| 3.4 综合练习 / 微博 | 84 |
| 3.4.1 微博应用程序功能概述 | 84 |
| 3.4.2 实现会员注册功能 | 86 |
| 3.4.3 实现会员登录功能 | 90 |
| 3.5 重点复习 | 92 |
| 3.6 课后练习 | 93 |

| | | | |
|-------------------------------------------------------|------------|--------------------------------------------------------------------------|------------|
| 第 4 章 会话管理..... | 96 | 5.4.1 AsyncContext 简介 | 161 |
| 4.1 会话管理基本原理 | 97 | 5.4.2 模拟服务器推播 | 164 |
| 4.1.1 使用隐藏域 | 97 | 5.4.3 更多 AsyncContext 细节 | 167 |
| 4.1.2 使用 Cookie..... | 100 | 5.5 综合练习 / 微博 | 168 |
| 4.1.3 使用 URL 重写..... | 104 | 5.5.1 创建 UserService..... | 168 |
| 4.2 HttpSession 会话管理..... | 107 | 5.5.2 设置过滤器 | 174 |
| 4.2.1 使用 HttpSession..... | 107 | 5.5.3 重构微博 | 175 |
| 4.2.2 HttpSession 会话管理 原理 | 111 | 5.6 重点复习 | 180 |
| 4.2.3 HttpSession 与 URL 重写.... | 113 | 5.7 课后练习 | 182 |
| 4.3 综合练习 / 微博 | 115 | 第 6 章 使用 JSP | 186 |
| 4.3.1 修改微博应用程序 | 116 | 6.1 从 JSP 到 Servlet | 187 |
| 4.3.2 新增与删除信息 | 117 | 6.1.1 JSP 生命周期 | 187 |
| 4.3.3 会员网页显示信息 | 120 | 6.1.2 Servlet 至 JSP 的简单 转换 | 190 |
| 4.4 重点复习 | 122 | 6.1.3 指示元素 | 194 |
| 4.5 课后练习 | 123 | 6.1.4 声明、Scriptlet 与表达式 元素 | 198 |
| 第 5 章 Servlet 进阶 API、过滤器 与监听器..... | 125 | 6.1.5 注释元素 | 202 |
| 5.1 Servlet 进阶 API..... | 126 | 6.1.6 隐式对象 | 203 |
| 5.1.1 Servlet、ServletConfig 与 GenericServlet | 126 | 6.1.7 错误处理 | 205 |
| 5.1.2 使用 ServletConfig | 128 | 6.2 标准标签 | 210 |
| 5.1.3 使用 ServletContext | 131 | 6.2.1 <jsp:include>、<jsp:forward> 标签 | 210 |
| 5.2 应用程序事件、监听器 | 134 | 6.2.2 <jsp:useBean>、<jsp:setProperty> 与<jsp:getProperty>简介 | 211 |
| 5.2.1 ServletContext 事件、 监听器 | 134 | 6.2.3 深入<jsp:useBean>、 <jsp:setProperty> 与 <jsp:getProperty> | 214 |
| 5.2.2 HttpSession 事件、 监听器 | 137 | 6.2.4 谈谈 Model 1 | 218 |
| 5.2.3 HttpServletRequest 事件、 监听器 | 144 | 6.2.5 XML 格式标签 | 220 |
| 5.3 过滤器 | 145 | 6.3 表达式语言(EL)..... | 221 |
| 5.3.1 过滤器的概念 | 145 | 6.3.1 EL 简介 | 221 |
| 5.3.2 实现与设置过滤器 | 147 | 6.3.2 使用 EL 取得属性 | 223 |
| 5.3.3 请求封装器 | 152 | 6.3.3 EL 隐式对象 | 226 |
| 5.3.4 响应封装器 | 157 | 6.3.4 EL 运算符 | 227 |
| 5.4 异步处理 | 161 | | |

| | | | |
|----------------------------------------------------|------------|--------------------------------------------------------|------------|
| 6.3.5 自定义 EL 函数 | 228 | 第 8 章 自定义标签 | 293 |
| 6.4 综合练习 / 微博 | 230 | 8.1 Tag File 自定义标签 | 294 |
| 6.4.1 改用 JSP 实现视图 | 230 | 8.1.1 Tag File 简介 | 294 |
| 6.4.2 重构 UserService 与 member.jsp | 234 | 8.1.2 处理标签属性与 Body | 297 |
| 6.4.3 创建 register.jsp、index.jsp、 user.jsp | 240 | 8.1.3 TLD 文件 | 299 |
| 6.5 重点复习 | 245 | 8.2 Simple Tag 自定义标签 | 301 |
| 6.6 课后练习 | 247 | 8.2.1 Simple Tag 简介 | 301 |
| 第 7 章 使用 JSTL | 249 | 8.2.2 了解 API 架构与生命 周期 | 304 |
| 7.1 JSTL 简介 | 250 | 8.2.3 处理标签属性与 Body | 306 |
| 7.2 核心标签库 | 252 | 8.2.4 与父标签沟通 | 310 |
| 7.2.1 流程处理标签 | 252 | 8.2.5 TLD 文件 | 314 |
| 7.2.2 错误处理标签 | 255 | 8.3 Tag 自定义标签 | 315 |
| 7.2.3 网页导入、重定向、URL 处理标签 | 257 | 8.3.1 Tag 简介 | 315 |
| 7.2.4 属性处理与输出标签 | 258 | 8.3.2 了解架构与生命周期 | 317 |
| 7.3 I18N 兼容格式标签库 | 261 | 8.3.3 重复执行标签 Body | 319 |
| 7.3.1 I18N 基础 | 261 | 8.3.4 处理 Body 运行结果 | 321 |
| 7.3.2 信息标签 | 264 | 8.3.5 与父标签沟通 | 324 |
| 7.3.3 地区标签 | 267 | 8.4 综合练习 / 微博 | 327 |
| 7.3.4 格式标签 | 272 | 8.4.1 实现首页最新信息 | 327 |
| 7.4 XML 标签库 | 275 | 8.4.2 自定义 Blahs 标签 | 330 |
| 7.4.1 XPath、XSLT 基础 | 276 | 8.5 重点复习 | 332 |
| 7.4.2 解析、设置与输出 标签 | 279 | 8.6 课后练习 | 334 |
| 7.4.3 流程处理标签 | 280 | 第 9 章 整合数据库 | 338 |
| 7.4.4 文件转换标签 | 281 | 9.1 JDBC 入门 | 339 |
| 7.5 函数标签库 | 283 | 9.1.1 JDBC 简介 | 339 |
| 7.6 综合练习 / 微博 | 284 | 9.1.2 连接数据库 | 344 |
| 7.6.1 修改 register.jsp | 285 | 9.1.3 使用 Statement、 ResultSet | 350 |
| 7.6.2 修改 member.jsp | 285 | 9.1.4 使用 PreparedStatement、 CallableStatement | 355 |
| 7.6.3 修改 user.jsp | 287 | 9.2 JDBC 进阶 | 359 |
| 7.7 重点复习 | 288 | 9.2.1 使用 DataSource 取得 连接 | 359 |
| 7.8 课后练习 | 290 | 9.2.2 使用 ResultSet 卷动、更新 数据 | 362 |

| | |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 9.2.3 批次更新 364 9.2.4 Blob 与 Clob 366 9.2.5 事务简介 374 9.2.6 metadata 简介 381 9.2.7 RowSet 简介 384 9.3 使用 SQL 标签库 390 9.3.1 数据源、查询标签 390 9.3.2 更新、参数、事务标签 391 9.4 综合练习 / 微博 392 9.4.1 重构 / 使用 DAO 393 9.4.2 使用 JDBC 实现 DAO 395 9.4.3 设置 JNDI 部署描述 400 9.5 重点复习 401 9.6 课后练习 402 第 10 章 Web 容器安全管理 404 10.1 了解与实现 Web 容器安全 管理 405 10.1.1 Java EE 安全基本概念 405 10.1.2 声明式基本身份验证 408 10.1.3 容器基本身份验证 原理 413 10.1.4 声明式窗体验证 414 10.1.5 容器窗体验证原理 415 10.1.6 使用 HTTPS 保护 数据 416 10.1.7 编程式安全管理 419 10.1.8 标注访问控制 422 10.2 综合练习 / 微博 424 10.2.1 使用容器窗体验证 424 10.2.2 设置 DataSourceRealm 426 10.3 重点复习 429 10.4 课后练习 430 第 11 章 JavaMail 入门 432 11.1 使用 JavaMail 433 11.1.1 传送纯文字邮件 433 | 11.1.2 发送多重内容邮件 436 11.2 综合练习 / 微博 440 11.2.1 实现取回密码功能 440 11.2.2 接收重送密码请求 445 11.3 重点复习 447 11.4 课后练习 447 第 12 章 从模式到框架 449 12.1 认识设计模式 450 12.1.1 Template Method 模式(Gof 设计模式) 450 12.1.2 Intercepting Filter 模式(Java EE 设计模式) 451 12.1.3 Model-View-Controller 模式 (架构模式) 452 12.2 重构、模式与框架 453 12.2.1 Business Delegate 模式 453 12.2.2 Service Locator 模式 454 12.2.3 Transfer Object 模式 455 12.2.4 Front Controller 模式 455 12.2.5 库与框架 456 12.3 重点复习 457 12.5 课后练习 459 附录 A 如何使用本书项目 460 附录 B MySQL 入门 463 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

编写与设置 Servlet

Chapter

2

学习目标：

- 开发环境准备与使用
- 了解 Web 应用程序架构
- Servlet 编写与部署设置
- 了解 URL 模式对应
- 使用 web-fragment.xml

2.1 第一个 Servlet

从本章开始，会正式学习 Servlet/JSP 的编写，如果想要打好坚实基础，就别急着从 JSP 开始学，要先从 Servlet 开始了解。正如第 1 章谈过的，JSP 终究会转译为 Servlet，了解 Servlet，JSP 也就学了一半了，而且不会被看似奇怪的 JSP 错误搞得稀里糊涂。

一开始先准备开发环境，会使用 Apache Tomcat(<http://tomcat.apache.org>)作为容器，而本书除了介绍 Servlet/JSP 之外，也会一并介绍集成开发环境(Integrated Development Environment)的使用，简称 IDE。毕竟在了解 Servlet/JSP 的原理与编程之外，了解如何善用 IDE 这样的开发工具来加快程序开发速度也是必要的，也符合业界需求。

2.1.1 准备开发环境



第 1 章曾经谈过，从抽象层面来说，Web 容器是 Servlet/JSP 唯一认得的 HTTP 服务器，所以开发工具的准备中，自然就要有 Web 容器的存在。这里使用 Apache Tomcat 作为 Web 容器，可以从以下网址下载：

<http://tomcat.apache.org/download-70.cgi>

注意»» 本书要介绍的 Servlet/JSP 版本是 Servlet 3.0/JSP 2.2，支持此版本的 Tomcat 版本是 Tomcat 7.x 以上。也可以使用光盘中提供的 apache-tomcat-7.0.8.zip。

在第 1 章中看过图 2.1。

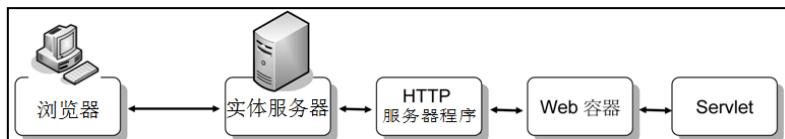


图 2.1 从请求到 Servlet 处理的线性关系

要注意的是，Tomcat 主要提供 Web 容器的功能，而不是 HTTP 服务器的功能，然而为了给开发者便利，下载的 Tomcat 会附带简单的 HTTP 服务器，相较于真正的 HTTP 服务器而言，Tomcat 附带的 HTTP 服务器功能太过简单，仅作开发用途，不建议以后直接上线服务。

接着准备 IDE。本书使用 Eclipse(<http://www.eclipse.org/>)，这是业界普遍采用的 IDE。可以从以下网址下载：

<http://www.eclipse.org/downloads/>



Eclipse 根据开发用途的不同，提供多种功能组合不同的版本。本书使用 Eclipse IDE for Java EE Developers，这个版本足以满足开发 Servlet/JSP 的需求。也可以使用光盘中提供的 eclipse-jee-helios-SR2-win32.zip。

当然，必须有 Java 运行环境，Java EE 6 搭配的版本为 Java SE 6。如果还没安装，可以从以下网址下载：

<http://www.oracle.com/technetwork/java/javase/downloads/index.html>

也可以直接使用光盘中附带的 jdk-6u24-windows-i586.exe。总结目前所需用到的工具有：

- JDK6
- Eclipse(建议 3.6 以上版本)
- Tomcat 7

JDK6 的安装请参考 Java 入门书籍(可参考 <http://caterpillar.onlyfun.net/Gossip/JavaEssence/InstallJDK.html>)。至于 Eclipse 与 Tomcat，如果愿意，可以配合本书的环境配置。本书制作范例时，将 Eclipse 与 Tomcat 都解压缩在 C:\workspace 中，如图 2.2 所示。

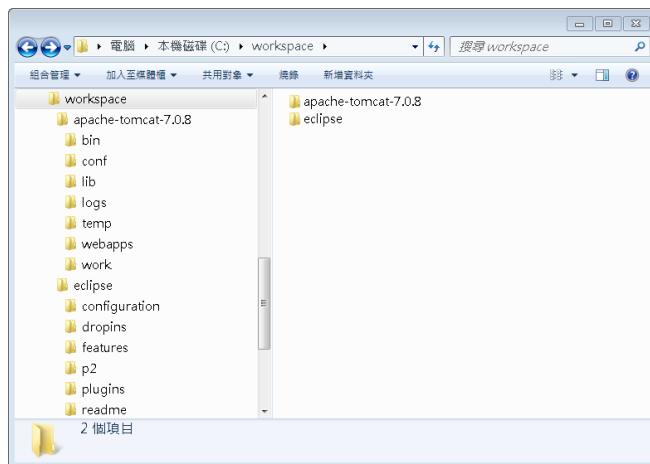


图 2.2 范例基本环境配置

提示»» 如果想放在别的目录中，请不要放在有中文或空格符的目录中，Eclipse 或 Tomcat 对此会有点感冒。

接着要在 Eclipse 中配置 Web 容器为 Tomcat，让以后开发的 Servlet/JSP 运行于 Tomcat 上，请按照以下步骤运行。



- (1) 运行 eclipse 目录中的 eclipse.exe。
- (2) 出现 Workspace Launcher 对话框时，将 Workspace: 设置为 C:\workspace，单击 OK 按钮。

(3) 选择 Window | Preferences 命令，在出现的 Preferences 对话框中，展开左边的 Server 节点，并选择其中的 Runtime Environment 节点。

(4) 单击右边 Server Runtime Environments 中的 Add 按钮，在出现的 New Server Runtime Environment 对话框中选择 Apache Tomcat v7.0，单击 Next 按钮。

(5) 单击 Tomcat installation directory 旁的 Browse 按钮，选取 C:\workspace 中解压缩的 Tomcat 目录，单击“确定”按钮。

(6) 在单击 Finish 按钮后，应该会看到图 2.3 所示的画面，单击 OK 完成配置。

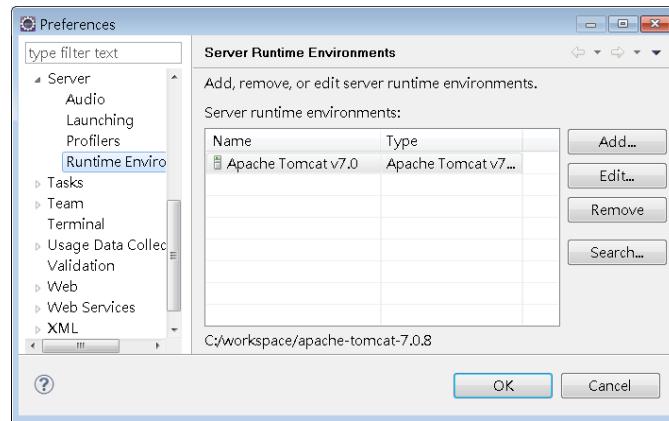


图 2.3 配置 Tomcat

接着要配置工作区(Workspace)预设的文字编码。Eclipse 默认会使用操作系统默认的文字编码，在 Windows 上就是 MS950，在这里建议使用 UTF-8。除此之外，CSS、HTML、JSP 等相关编码设置，也建议都设为 UTF-8，这可以避免日后遇到一些编码处理上的问题。请按照以下步骤进行：



(1) 选择 Window/Preferences 命令，在出现的 Preferences 对话框中，展开左边的 Workspace 节点。

(2) 在右边的 Text file encoding 中选择 Other，在下拉菜单中选择 UTF-8。

(3) 展开左边的 Web 节点，选择 CSS Files，在右边的 Encoding 选择 UTF-8。

(4) 选择 HTML Files，在右边的 Encoding 选择 UTF-8。

(5) 单击 Preferences 对话框中的 OK 按钮完成设置。

2.1.2 第一个 Servlet 程序

接着可以开始编写第一个 Servlet 程序了，目的是用 Servlet 接收用户名并显示招呼语。由于 IDE 是集成开发工具，会使用项目来管理应用程序相关资源，在 Eclipse 中则是要创建 Dynamic Web Project，之后创建第一个 Servlet。请按照以下步骤进行操作：



(1) 选择 File | New | Dynamic Web Project 命令, 出现 New Dynamic Web Project 对话框, 在 Project name 文本框中输入 FirstServlet。

(2) 确定 Target runtime 为刚才设置的 Apache Tomcat v7.0, 单击 Finish 按钮。

(3) 展开新建项目中的 Java Resources 节点, 在 src 上右击, 从弹出的快捷菜单中选择 New | Servlet 命令。

(4) 弹出 Create Servlet 对话框, 在 Java package 文本框中输入 cc.openhome, 在 Class name 文本框中输入 HelloServlet, 单击 Next 按钮。

(5) 选择 URL mappings 中的 HelloServlet, 单击右边的 Edit 按钮, 将 Pattern 改为 /hello.view 后, 单击 OK 按钮。

(6) 单击 Create Servlet 对话框中的 Finish 按钮。

接着就可以编写第一个 Servlet 的内容了。在创建的 HelloServlet.java 中编辑以下内容:

FirstServlet HelloServlet.java

```
package cc.openhome;

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

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

@WebServlet("/hello.view")
public class HelloServlet extends HttpServlet { ← ① 继承 HttpServlet
    @Override
    protected void doGet(HttpServletRequest request, ← ② 重新定义 doGet()
        HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8"); ← ③ 设置响应内容类型器
        PrintWriter out = response.getWriter(); ← ④ 取得响应输出对象
        String name = request.getParameter("name"); ← ⑤ 取得"请求参数"
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Hello Servlet</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1> Hello! " + name + " !</h1>"); ← ⑥ 跟用户说 Hello!
        out.println("</body>");
        out.println("</html>");

        out.close();
    }
}
```

范例中继承了 `HttpServlet`①，并重新定义了 `doGet()`方法②，当浏览器 GET 方法发送请求时，会调用此方法。

在 `doGet()`方法上可以看到 `HttpServletRequest` 与 `HttpServletResponse` 两个参数，容器接收到客户端的 HTTP 请求后，会收集 HTTP 请求中的信息，并分别创建代表请求与响应的 Java 对象，而后在调用 `doGet()` 时将这两个对象当作参数传入。可以从 `HttpServletRequest` 对象中取得有关 HTTP 请求相关信息，在范例中是通过 `HttpServletRequest` 的 `getParameter()` 并指定请求参数名称，来取得用户发送的请求参数值③。

注意»» 范例中的 `@Override` 是 JDK5 之后所提供的标注(Annotation)，作用是协助检查是否正确地重新定义了父类中继承下来的某个方法。就编写 Servlet 而言，没有 `@Override` 并没有影响。

由于 `HttpServletResponse` 对象代表对客户端的响应，因此可以通过其 `setContentType()` 设置正确的内容类型④。范例中是告知浏览器，返回的响应要以 `text/html` 解析，而采用的字符编码是 UTF-8。接着再使用 `getWriter()`方法取得代表响应输出的 `PrintWriter` 对象⑤，通过 `PrintWriter` 的 `println()`方法来对浏览器输出响应的文字信息，在范例中是输出 HTML 以及根据用户名说声 Hello! ⑥。

提示»» 在 Servlet 的 Java 代码中，以字符串输出 HTML，当然是很笨的行为。别担心，在谈到 JSP 时，会有个有趣的练习，让你将 Servlet 转为 JSP，从中明了 Servlet 与 JSP 的对应。

接着要来运行 Servlet，你会对这个 Servlet 作请求，同时附上请求参数。请按照以下步骤进行：



- (1) 在 `HelloServlet.java` 上右击，从弹出的快捷菜单中选择 `Run As | Run on Server` 命令。
- (2) 在弹出的 `Run on Server` 对话框中，确定 `Server runtime environment` 为先前设置的 Apache Tomcat v7.0，单击 `Finish` 按钮。
- (3) 在 Tomcat 启动后，会出现内嵌于 Eclipse 的浏览器，将地址栏设置为：

`http://localhost:8080/FirstServlet/hello.view?name=caterpillar`

按以上步骤操作之后，就会看到图 2.4 所示的画面。



图 2.4 第一个 Servlet 程序



Tomcat 默认会使用 8080 端口，注意到地址栏中，请求的 Web 应用程序路径是 FirstServlet 吗？默认项目名称就是 Web 应用程序路径，那为何请求的 URL 是 /hello.view 呢？记得 HelloServlet.java 中有这么一行吗？

```
@WebServlet("/hello.view")
```

这表示，如果请求的 URL 是 /hello.view，就会由 HelloServlet 来处理请求。关于 Servlet 的设置，还有更多的细节。事实上，由于到目前为止，借助了 IDE 的辅助，有许多细节都被省略了，所以接下来得先讨论这些细节。

2.2 在 HelloServlet 之后

现在在 IDE 中编写了 HelloServlet，并成功运行出应有的结果，那这一切是如何串起来的，IDE 又代劳了哪些事情？你在 IDE 的项目管理中看到的文件组织结构真的是应用程序上传之后的结构吗？

记住：Web 容器是 Servlet/JSP 唯一认得 HTTP 服务器，你要了解 Web 容器会读取哪些设置？又要求什么样的文件组织结构？Web 容器对于请求到来，又会如何调用 Servlet？IDE 很方便，但不要过分依赖 IDE。

2.2.1 关于 HttpServlet

注意到 HelloServlet.java 中 import 的语句区段：

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

如果要编译 HelloServlet.java，则类路径(Classpath)中必须包括 Servlet API 的相关类，如果使用的是 Tomcat，则这些类通常是封装在 Tomcat 目录的 lib 子目录中的 servlet-api.jar。假设 HelloServlet.java 位于 src 目录下，并放置于对应包的目录中，则可以像以下这样进行编译：

```
% cd YourWorkspace/FirstServlet
% javac -classpath Yourlibrary/YourTomcat/lib/servlet-api.jar -d ./classes
src/cc/openhome/HelloServlet.java
```

注意下划线部分必须修改为实际的目录位置，编译出的.class 文件会出现在 classes 目录中，并有对应的包层级(因为使用 javac 时加了-d 自变量)。事实上，如果遵照 2.1 节的操作，Eclipse 就会自动完成类路径设置，并完成编译等事宜。展开 Project Explorer 中的 Libraries/Apache Tomcat v7.0 节点，就会看到相关 JAR(Java ARchive)文件的类路径设置，如图 2.5 所示。

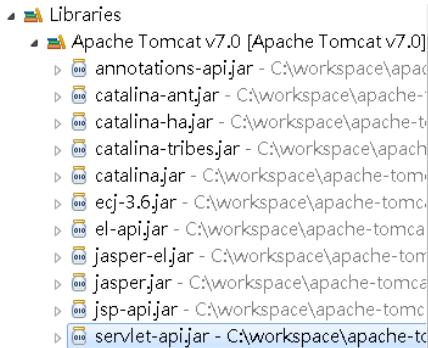


图 2.5 IDE 会自动设置项目的类路径

再进一步思考一个问题，为什么要在继承 `HttpServlet` 之后重新定义 `doGet()`？又为什么 HTTP 请求为 GET 时会自动调用 `doGet()`？首先来讨论范例中看到的相关 API 架构图，如图 2.6 所示。

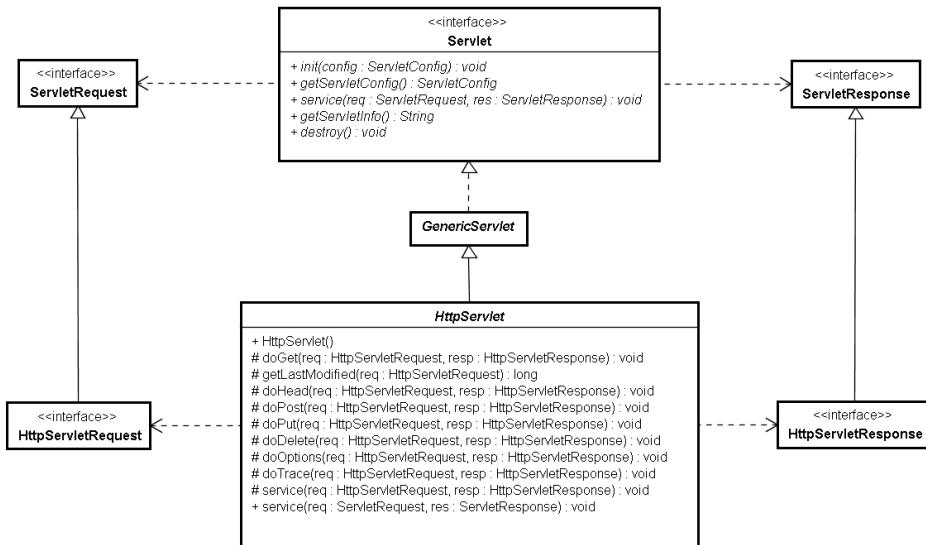


图 2.6 `HttpServlet` 相关 API 类图

首先看到 `Servlet` 接口，它定义了 `Servlet` 应当有的基本行为。例如，与 `Servlet` 生命周期相关的 `init()`、`destroy()` 方法，提供服务时所要调用的 `service()` 方法等。

实现 `Servlet` 接口的类是 `GenericServlet` 类，它还实现了 `ServletConfig` 接口，将容器调用 `init()` 方法时所传入的 `ServletConfig` 实例封装起来，而 `service()` 方法直接标示为 `abstract` 而没有任何的实现。在本章中将暂且忽略对 `GenericServlet` 的讨论，只需先知道有它的存在(第 5 章会加以讨论)。

在这里只要先注意到一件事，`GenericServlet` 并没有规范任何有关 HTTP 的相关方法，而是由继承它的 `HttpServlet` 来定义。在最初定义 `Servlet` 时，并不限定它只能用于 HTTP，所以并没有将 HTTP 相关服务流程定义在 `GenericServlet` 之中，而是定义在 `HttpServlet` 的 `service()` 方法中。



提示»» 可以注意到包(package)的设计，与 Servlet 定义相关的类或接口都位于 javax.servlet 包中，如 Servlet、GenericServlet、ServletRequest、ServletResponse 等。而与 HTTP 定义相关的类或接口都位于 javax.servlet.http 包中，如 HttpServlet、HttpServletRequest、HttpServletResponse 等。

HttpServlet 的 service() 方法中的流程大致如下：

```
protected void service(HttpServletRequest req,
                      HttpServletResponse resp)
    throws ServletException, IOException {
    String method = req.getMethod(); // 取得请求的方法
    if (method.equals(METHOD_GET)) { // HTTP GET
        // 略...
        doGet(req, resp);
        // 略 ...
    } else if (method.equals(METHOD_HEAD)) { // HTTP HEAD
        // 略 ...
        doHead(req, resp);
    } else if (method.equals(METHOD_POST)) { // HTTP POST
        // 略 ...
        doPost(req, resp);
    } else if (method.equals(METHOD_PUT)) { // HTTP PUT
        // 略 ...
    }
}
```

当请求来到时，容器会调用 Servlet 的 service() 方法。可以看到，HttpServlet 的 service() 中定义的，基本上就是判断 HTTP 请求的方式，再分别调用 doGet()、doPost() 等方法，所以若想针对 GET、POST 等方法进行处理，才会只需要在继承 HttpServlet 之后，重新定义相对应的 doGet()、doPost() 方法。

注意»» 这其实是使用了设计模式(Design Pattern)中的 Template Method 模式。所以不建议也不应该在继承了 HttpServlet 之后，重新定义 service() 方法，这会覆盖掉 HttpServlet 中定义的 HTTP 预设处理流程。

2.2.2 使用@WebServlet

编写好 Servlet 之后，接下来要告诉 Web 容器有关于这个 Servlet 的一些信息。在 Servlet 3.0 中，可以使用标注(Annotation)来告知容器哪些 Servlet 会提供服务以及额外信息。例如在 HelloServlet.java 中：

```
@WebServlet("/hello.view")
public class HelloServlet extends HttpServlet {
```

只要在 Servlet 上设置 @WebServlet 标注，容器就会自动读取当中的信息。上面的 @WebServlet 告诉容器，如果请求的 URL 是 “/hello.view”，则由 HelloServlet 的实例提供服务。可以使用 @WebServlet 提供更多信息。

```
@WebServlet(
    name="Hello",
```

```

    urlPatterns={"/hello.view"},
    loadOnStartup=1
)
public class HelloServlet extends HttpServlet {

```

上面的 @WebServlet 告知容器，HelloServlet 这个 Servlet 的名称是 Hello，这是由 name 属性指定的，而如果客户端请求的 URL 是 /hello.view，则由具 Hello 名称的 Servlet 来处理，这是由 urlPatterns 属性来指定的。在 Java EE 相关应用程序中使用标注时，可以记得的是，没有设置的属性通常会有默认值。例如，若没有设置 @WebServlet 的 name 属性，默认值会是 Servlet 的类完整名称。

当应用程序启动后，事实上并没有创建所有的 Servlet 实例。容器会在首次请求需要某个 Servlet 服务时，才将对应的 Servlet 类实例化、进行初始化操作，然后再处理请求。这意味着第一次请求该 Servlet 的客户端，必须等待 Servlet 类实例化、进行初始动作所必须花费的时间，才真正得到请求的处理。

如果希望应用程序启动时，就先将 Servlet 类载入、实例化并做好初始化动作，则可以使用 loadOnStartup 设置。设置大于 0 的值（默认值为 -1），表示启动应用程序后就要初始化 Servlet（而不是实例化几个 Servlet）。数字代表了 Servlet 的初始顺序，容器必须保证有较小数字的 Servlet 先初始化，在使用标注的情况下，如果有多个 Servlet 在设置 loadOnStartup 时使用了相同的数字，则容器实现厂商可以自行决定要如何载入哪个 Servlet。

2.2.3 使用 web.xml

使用标注来定义 Servlet 是 Java EE 6 中 Servlet 3.0 之后才有的功能，在先前的版本中，必须在 Web 应用程序的 WEB-INF 目录中，建立一个 web.xml 文件来定义 Servlet 相关信息。在 Servlet 3.0 中，也可以使用 web.xml 文件来定义 Servlet。

例如，可以在先前的 FirstServlet 项目的 Project Explorer 中：



- (1) 展开 WebContent/WEB-INF 节点，在 WEB-INF 节点上右击，从弹出的快捷菜单中选择 New | File 命令。
- (2) 在 File name 文本框中输入 web.xml 后单击 Finish 按钮。
- (3) 在打开的 web.xml 下面单击 Source 标签，并输入以下内容：

FirstServlet web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.0" xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd">
  <servlet>
    <servlet-name>HelloServlet</servlet-name>
    <servlet-class>cc.openhome.HelloServlet</servlet-class>

```

```

<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>HelloServlet</servlet-name>
    <url-pattern>/helloUser.view</url-pattern>
</servlet-mapping>
</web-app>

```

如这样的文件称为部署描述文件(Deployment Descriptor, 简称 DD 文件)。使用 web.xml 定义是比较麻烦一些，不过 web.xml 中的设置会覆盖 Servlet 中的标注设置，可以使用标注来作默认值，而 web.xml 来作日后更改设置值之用。在上例中，若有客户端请求/helloUser.view，则由 HelloServlet 这个 Servlet 来处理，这分别是由< servlet-mapping >中的< url-pattern >与< servlet-name >来定义，而 HelloServlet 名称的 Servlet，实际上是 cc.openhome.HelloServlet 类的实例，这分别是由< servlet >中的< servlet-name >与< servlet-class >来定义。如果有多个 Servlet 在设置< load-on-startup >时使用了相同的数字，则依其在 web.xml 中设置的顺序来初始 Servlet，如图 2.7 所示。

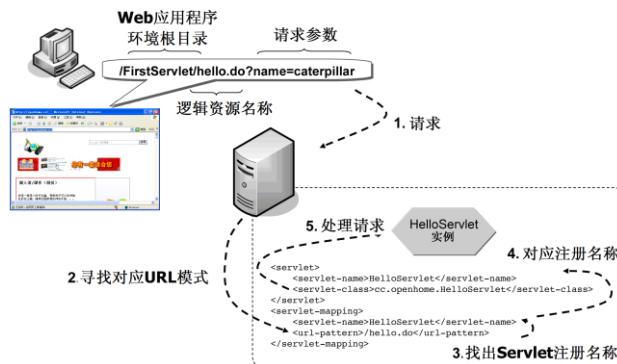


图 2.7 Servlet 的请求对应

图 2.7 中，Web 应用程序环境根目录(Context Root)是可以自行设置的，不过设置方式会因使用的 Web 应用程序服务器而有所不同。例如，Tomcat 默认会使用应用程序目录作为环境根目录，在 Eclipse 中，可以在项目上右击，从弹出的快捷菜单中选择 Properties 命令，在 Web Project Settings 中进行设置，如图 2.8 所示。

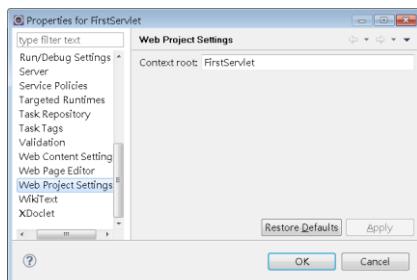


图 2.8 在 Eclipse 中设置 Context root

无论使用 @WebServlet 标注，还是使用 web.xml 设置，应该已经知道请求时的 URL 是个逻辑名称(Logical Name)，请求/hello.view 并不是指服务器上真的有个实体文件叫 hello.view，而会再由 Web 容器对应至实际处理请求的文件或程序实体名称(Physical Name)。如果愿意，也可以再用个像 hello.jsp 之类的名称来伪装资源。

到目前为止，你可以知道，一个 Servlet 在 web.xml 中会有三个名称设置：`<url-pattern>`设置的逻辑名称，`<servlet-name>`注册的 Servlet 名称，以及`<servlet-class>`设置的实体类名称。

注意»» 除了可将@WebServlet 的设置当作默认值，web.xml 用来覆盖默认值的好处外，想一下，在 Servlet 3.0 之前，只能使用 web.xml 设置时的问题。写好了一个 Servlet 并编译完成，现在要寄给同事或客户，你还得跟他说如何在 web.xml 中设置。在 Servlet 3.0 之后，只要在 Servlet 中使用@WebServlet 设置好标注信息，寄给同事或客户后，他只要将编译好的 Servlet 放到 WEB-INF/classes 目录中就可以了(稍后就会谈到这个目录)，部署上简化了许多。

2.2.4 文件组织与部署

IDE 为了管理项目资源，会有其项目专属的文件组织，那并不是真正上传至 Web 容器之后该有的架构。Web 容器要求应用程序部署时，必须遵照图 2.9 所示结构。

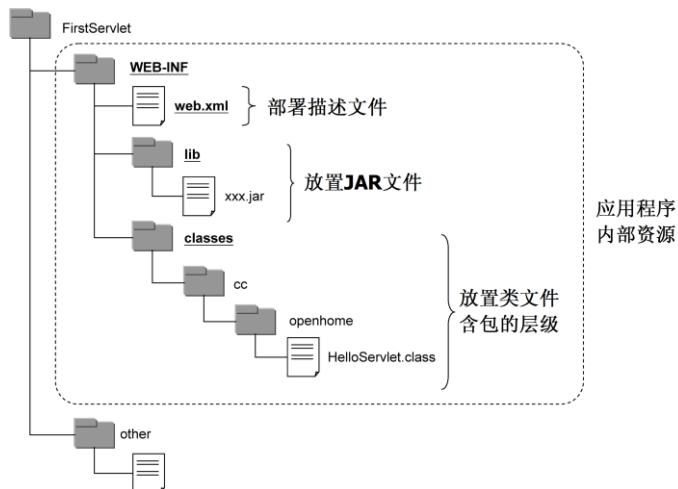


图 2.9 Web 应用程序文件组织

图 2.9 中有几个重要的目录与文件位置说明如下。

- **WEB-INF：**这个目录名称是固定的，而且一定是位于应用程序根目录下。放置在 WEB-INF 中的文件或目录，对外界来说是封闭的，也就是客户端无法使用 HTTP 的任何方式直接访问到 WEB-INF 中的文件或目录。若有这类需要，则必须通过 Servlet/JSP 的请求转发(Forward)。不想让外界存取的资源，可以放置在这个目录下。



- **web.xml:** 这是 Web 应用程序部署描述文件，一定是放在 WEB-INF 根目录下，名称一定是 web.xml。
- **lib:** 放置 JAR 文件的目录，一定是放在 WEB-INF 根目录下，名称一定是 lib。
- **classes:** 放置编译过后.class 文件的目录，一定是放在 WEB-INF 目录下，名称一定是 classes。编译过后的类文件，必须有与包名称相符的目录结构。

如果使用 Tomcat 作为 Web 容器，则可以将符合图 2.9 的 FirstServlet 整个目录复制至 Tomcat 目录下 webapps 于目录，然后至 Tomcat 的 bin 目录下，运行 startup 命令来启动 Tomcat。接着使用以下的 URL 来请求应用程序(假设 URL 模式为 /helloUser.view)：

```
http://localhost:8080/FirstServlet/helloUser.view?name=caterpillar
```

实际上在部署 Web 应用程序时，会将 Web 应用程序封装为一个 WAR 文件，也是一个后缀为*.war 的文件。WAR 文件可使用 JDK 所附的 jar 工具程序来建立。例如，当按图 2.9 所示的方式组织好 Web 应用程序文件之后，可进入 FirstServlet 目录，然后运行以下命令：

```
jar cvf ../FirstServlet.war *
```

这会在 FirstServlet 目录外建立一个 FirstServlet.war 文件，在 Eclipse 中，则可以直接在项目中右击，从弹出的快捷菜单中选择 Export/WAR file 命令导出 WAR 文件。

WAR 文件是使用 zip 压缩格式封装的，可以使用解压缩软件来查看其中的内容。如果使用 Tomcat，则可以将所建立的 WAR 文件复制至 webapps 目录下，重新启动 Tomcat，容器若发现 webapps 目录中有 WAR 文件，会将其解压缩，并载入 Web 应用程序。

提示»» 不同的应用服务器，会提供不同的命令或接口让你部署 WAR 文件。有关 Tomcat 7 更多的部署方式，可以查看以下网址：

```
http://tomcat.apache.org/tomcat-7.0-doc/deployer-howto.html
```

2.3 进阶部署设置

初学 Servlet/JSP，了解本章之前所说明的目录结构与部署设置已经足够，然而在 Servlet 3.0 中，确实增加了一些新的部署设置方式，可以让 Servlet 的部署更方便、更模块化、更具弹性。

由于接下来的内容是比较进阶或 Servlet 3.0 新增的功能，如果是第一次接触 Servlet，急着想要了解如何使用 Servlet 相关 API 开发 Web 应用程序，则可以先跳过这一节的内容，日后想要了解更多部署设置时再回来查看。

2.3.1 URL 模式设置

一个请求 URI 实际上是由三个部分组成的：

```
requestURI = contextPath + servletPath + pathInfo
```

1. 环境路径

可以使用 `HttpServletRequest` 的 `getRequestURI()` 来取得这项信息，其中 `contextPath` 是环境路径(Context path)，是容器用来决定该挑选哪个 Web 应用程序的依据(一个容器上可能部署多个 Web 应用程序)，环境路径的设置方式标准中并没有规范，如上一节谈过的，这依使用的应用程序服务器而有所不同。

可使用 `HttpServletRequest` 的 `getContextPath()` 来取得环境路径。如果应用程序环境路径与 Web 服务器环境根路径相同，则应用程序环境路径为空字符串，如果不是，则应用程序环境路径以“/”开头，不包括“/”结尾。

提示»» 下一章就会细谈 `HttpServletRequest`，目前你大概也可以察觉，有关请求的相关信息，都可以使用这个对象来取得。

一旦决定是哪个 Web 应用程序来处理请求，接下来就进行 Servlet 的挑选，Servlet 必须设置 URL 模式(URL pattern)。可以设置的格式分别说明如下。

- 路径映射(Path mapping): 以“/”开头但以“/*”结尾的 URL 模式。例如，若设置 URL 模式为“/guest/*”，则请求 URI 扣去环境路径的部分若为 /guest/test.view、/guest/home.view 等以/guest/作为开头的，都会交由该 Servlet 处理。
- 扩展映射(extension mapping): 以“*.”开头的 URL 模式。例如，若 URL 模式设置为*.view，则所有以.view 结尾的请求，都会交由该 Servlet 处理。
- 环境根目录(Context root)映射: 空字符串""是个特殊的 URL 模式，对应至环境根目录，也就是“/”的请求，但不用于设置`<url-pattern>`或 `urlPattern` 属性。例如，若环境根目录为 App，则 `http://host:port/App/` 的请求，路径信息是“/”，而 Servlet 路径与环境路径都是空字符串。
- 预设 Servlet: 仅包括“/”的 URL 模式，当找不到适合的 URL 模式对应时，就会使用预设 Servlet。
- 完全匹配(Exact match): 不符合以上设置的其他字符串，都要作路径的严格对应。例如，若设置/guest/test.view，则请求不包括请求参数部分，必须是/guest/test.view。

如果 URL 模式在设置比对的规则在某些 URL 请求时有所重叠，例如若有 `/admin/login.do`、`/admin/*` 与 `*.do` 三个 URL 模式设置，则请求时比对的原则是从最



严格的 URL 模式开始符合。如果请求/admin/login.do，则一定是由 URL 模式设置为/admin/login.do 的 Servlet 来处理，而不会是/admin/* 或 *.do。如果请求/admin/setup.do，则是由/admin/*的 Servlet 来处理，而不会是*.do。

2. Servlet 路径

在最上面的 requestURI 中，servletPath 的部分是指 Servlet 路径(Servlet path)，不包括路径信息(Path info)与请求参数(Request parameter)。Servlet 路径直接对应至 URL 模式信息，可使用 HttpServletRequest 的 `getServletPath()` 来取得，Servlet 路径基本上是以“/”开头，但“/*”与“”的 URL 模式比对而来的请求除外，在“/*”与“”的情况下，`getServletPath()`取得的 Servlet 路径是空字符串。

例如，若某个请求是根据/hello.do 对应至某个 Servlet，则 `getServletPath()` 取得的 Servlet 路径就是/hello.do 如果是通过/servlet/*对应至 Servlet，则 `getServletPath()` 取得的 Servlet 路径就是/servlet，但如果是通过“/*”或“”对应至 Servlet，则 `getServletPath()` 取得的 Servlet 路径就是空字符串。

3. 路径信息

在最上面的 requestURI 中，pathInfo 的部分是指路径信息(Path info)，路径信息不包括请求参数，指的是不包括环境路径与 Servlet 路径部分的额外路径信息。可使用 HttpServletRequest 的 `getPathInfo()` 来取得。如果没有额外路径信息，则为 null(扩展映射、预设 Servlet、完全匹配的情况下，`getPathInfo()` 就会取得 null)，如果有额外路径信息，则是一个以“/”开头的字符串。

如果编写以下 Servlet：

FirstServlet PathServlet.java

```
package cc.openhome;

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

@WebServlet("/servlet/*")
public class PathServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req,
                         HttpServletResponse resp)
        throws ServletException, IOException {
        PrintWriter out = resp.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet Pattern</title>");
        out.println("</head>");
        out.println("<body>");
        out.println(req.getRequestURI() + "<br>");
    }
}
```

```
    out.println(req.getContextPath() + "<br>");
    out.println(req.getServletPath() + "<br>");
    out.println(req.getPathInfo());
    out.println("</body>");
    out.println("</html>");
    out.close();
}
}
```

如果在浏览器中输入的 URL 为：

http://localhost:8080/FirstServlet/servlet/path.view

那么看到的结果就如图 2.10 所示。

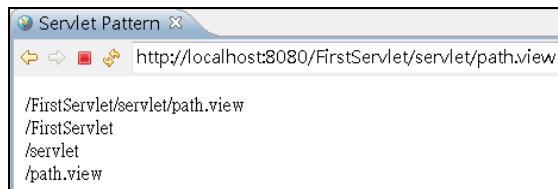


图 2.10 请求的路径信息

提示»» 这一节刚才与接下来要介绍的设置相关细节相当琐碎，实际操作中并不需要记忆，知道哪边或如何找到文件可以查询就可以了。当然，如果要应付考试，那就另当别论了。

2.3.2 Web 目录结构

在第一个 Servlet 中简要介绍过 Web 应用程序目录架构，这里再做个详细的说明。一个 Web 应用程序基本上会由以下项目组成：

- 静态资源(HTML、图片、声音等)
- Servlet
- JSP
- 自定义类
- 工具类
- 部署描述文件(web.xml 等)、设置信息(Annotation 等)

Web 应用程序目录结构必须符合规范。举例来说，如果一个应用程序的环境路径(Context path)是/openhome，则所有的资源项目必须以/openhome 为根目录依规定结构摆放。基本上根目录中的资源可以直接下载，例如若 index.html 位于/openhome 下，则可以直接以/openhome/index.html 来取得。

Web 应用程序存在一个特殊的/WEB-INF 目录，此目录中存在的资源项目不会被列入应用程序根目录中可直接访问的项。也就是说，客户端(例如浏览器)不可以



直接请求/WEB-INF 中的资源(直接在网址上指明访问 /WEB-INF)，否则就是 404 Not Found 的错误结果。/WEB-INF 中的资源项目有着一定的名称与结构。例如：

- /WEB-INF/web.xml 是部署描述文件。
- /WEB-INF/classes 用来放置应用程序用到的自定义类(.class)，必须包括包(Package)结构。
- /WEB-INF/lib 用来放置应用程序用到的 JAR 文件。

Web 应用程序用到的 JAR 文件，其中可以放置 Servlet、JSP、自定义类、工具类、部署描述文件等，应用程序的类载入器可以从 JAR 中载入对应的资源。

可以在 JAR 文件的/META-INF/resources 目录中放置静态资源或 JSP 等，例如若在 /META-INF/resources 中放个 index.html，若请求的 URL 中包括 /openhome/index.html，但实际上/openhome 根目录下不存在 index.html，则会使用 JAR 中的/META-INF/resources/index.html。

如果要用到某个类，则 Web 应用程序会到/WEB-INF/classes 中试着载入类，若无，再试着从/WEB-INF/lib 的 JAR 文件中寻找类文件(若还没有找到，则会到容器实现本身存放类或 JAR 的目录中寻找，但位置视实现厂商而有所不同，以 Tomcat 而言，搜寻的路径是 Tomcat 安装目录下的 lib 目录)。

客户端不可以直接请求/WEB-INF 中的资源，但可以通过程序的控制，让程序来取得/WEB-INF 中的资源，如使用 `ServletContext` 的 `getResource()` 与 `getResourceAsStream()`，或是通过 `RequestDispatcher` 请求调派。这在之前的章节会看到实际范例。

如果 Web 应用程序的 URL 最后是以“/”结尾，而且确实存在该目录，则 Web 容器必须传回该目录下的欢迎页面，可以在部署描述文件 `web.xml` 中包括以下的定义，指出可用的欢迎页面名称为何。Web 容器会依序看看是否有对应的文件存在，如果有，则返回给客户端：

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

如果找不到以上的文件，则会尝试至 JAR 的/META-INF/resources 中寻找已放置的资源页面。如果 URL 最后是以“/”结尾，但不存在该目录，则会使用预设 Servlet(如果有定义的话，参考 2.3.1 节的说明)。

整个 Web 应用程序可以被封装为一个 WAR 文件，如 `openhome.war`，以便部署至 Web 容器。

2.3.3 使用 web-fragment.xml

在 Servlet 3.0 中，可以使用标注来设置 Servlet 的相关信息。实际上，Web 容器并不仅读取/WEB-INF/classes 中的 Servlet 标注信息，如果一个 JAR 文件中有使用标注的 Servlet，Web 容器也可以读取标注信息、载入类并注册为 Servlet 进行服务。

在 Servlet 3.0 中，JAR 文件可用来作为 Web 应用程序的部分模块。事实上，不仅是 Servlet，监听器、过滤器等也可以在编写、定义标注完毕后，封装在 JAR 文件中，视需要放置至 Web 应用程序的/WEB-INF/lib 中，弹性抽换 Web 应用程序的功能性。

1. web-fragment.xml

一个 JAR 文件中，除了可使用标注定义的 Servlet、监听器、过滤器外，也可以拥有自己的部署描述文件，这个文件的名称是 web-fragment.xml，必须放置在 JAR 文件的 META-INF 目录中。基本上，web.xml 中可定义的元素，在 web-fragment.xml 中也可以定义。举个例子来说，可以在 web-fragment.xml 中定义如下内容：

```
<?xml version="1.0" encoding="UTF-8"?>
<web-fragment xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-fragment_3_0.xsd"
    version="3.0">
    <name>WebFragment1</name>
    <servlet>
        <servlet-name>hi</servlet-name>
        <servlet-class>cc.openhome.HiServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>hi</servlet-name>
        <url-pattern>/hi.view</url-pattern>
    </servlet-mapping>
</web-fragment>
```

注意»» web-fragment.xml 的根标签是 `<web-fragment>` 而不是 `<web-app>`。实际上，web-fragment.xml 中所指定的类，不一定要在 JAR 文件中，也可以是在 web 应用程序的/WEB-INF/classes 中。

在 Eclipse 中内置 Web Fragment Project，如果想要尝试使用 JAR 文件部署 Servlet，或者使用 web-fragment.xml 部署的功能，可以按照以下步骤练习：



- (1) 选择 File | New | Other 命令，在出现的对话框中选择 Web 节点中的 Web Fragment Project 节点，单击 Next 按钮。



(2) 在 New Web Project Fragment Project 对话框中，注意可以设置 Dynamic Web Project membership。这里可以选择 Web Fragment Project 产生的 JAR 文件，将会部署于哪一个项目中，这样就不用手动产生 JAR 文件，并将之复制至另一应用程序的 WEB-INF/lib 目录中。

(3) 在 Project name 文本框中输入 FirstWebFrag，单击 Finish 按钮。

(4) 展开新建立的 FirstWebFrag 项目中 src/META-INF 节点，可以看到预先建立的 web-fragment.xml。可以在该项目中建立 Servlet 等资源，并设置 web-fragment.xml 的内容。

(5) 在 FirstServlet 项目上右击(刚才 Dynamic Web Project membership 设置的对象)，从弹出的快捷菜单中选择 Properties 命令，展开 Deployment Assembly 节点，可以看到 FirstWebFrag 项目建构而成的 FirstWebFrag.jar，将会自动部署至 FirstServlet 项目 WEB-INF/lib 中。

接着可以在 FirstWebFrag 中新增 Servlet 并设置标注，看看运行结果是什么，再在 web-fragment.xml 中设置相关信息，并再次实验运行结果是什么。

2. web.xml 与 web-fragment.xml

Servlet 3.0 对 web.xml 与标注的配置顺序并没有定义，对 web-fragment.xml 及标注的配置顺序也没有定义，然而可以决定 web.xml 与 web-fragment.xml 的配置顺序，其中一个设置方式是在 web.xml 中使用 `<absolute-ordering>` 定义绝对顺序。例如，在 web.xml 中定义：

```
<web-app ...>
  <absolute-ordering>
    <name>WebFragment1</name>
    <name>WebFragment2</name>
  </absolute-ordering>
  ...
</web-app>
```

各个 JAR 文件中 web-fragment.xml 定义的名称不得重复，若有重复，则会忽略掉重复的名称。另一个定义顺序的方式，是直接在每个 JAR 文件的 web-fragment.xml 中使用 `<ordering>`，在其中使用 `<before>` 或 `<after>` 来定义顺序。以下是一个例子，假设有三个 web-fragment.xml 分别存在于三个 JAR 文件中：

```
<web-fragment ...>
  <name>WebFragment1</name>
  <ordering>
    <after><name>MyFragment2</name>
  </after></ordering>
  ...
</web-fragment>

<web-fragment ...>
  <name>WebFragment2</name>
  ...

```

```

</web-fragment>

<web-fragment ...>
    <name>WebFragment3</name>
    <ordering>
        <before><others/></before>
    </ordering>
    ...
</web-fragment>

```

而 web.xml 没有额外定义顺序信息：

```

<web-app ...>
    ...
</web-app>

```

则载入定义的顺序是 web.xml，`<name>`名称为 WebFragment3、WebFragment2、WebFragment1 的 web-fragment.xml 中的定义。

3. metadata-complete 属性

如果将 web.xml 中 `<web-app>` 的 `metadata-complete` 属性设置为 `true`(默认是 `false`)，则表示 web.xml 中已完成 Web 应用程序的相关定义，部署时将不会扫描标注与 web-fragment.xml 中的定义，如果有 `<absolute-ordering>` 与 `<ordering>` 也会被忽略。例如：

```

<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
        http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0"
    metadata-complete="true">
    ...
</web-app>

```

如果 web-fragment.xml 中指定的类可以在 web 应用程序的 /WEB-INF/classes 中找到，就会使用该类。要注意的是，如果该类本身有标注，而 web-fragment.xml 又定义该类为 Servlet，则此时会有两个 Servlet 实例。如果将 `<web-fragment>` 的 `metadata-complete` 属性设置为 `true`(默认是 `false`)，就只会处理自己 JAR 文件中的标注信息。

可以参考 Servlet 3.0 说明书(JSR 315)中第 8 章内容，其中有更多的 web.xml、web-fragment.xml 的定义范例。

2.4 重点复习

Tomcat 提供的主要是 Web 容器的功能，而不是 HTTP 服务器的功能。然而为了给开发者便利，下载的 Tomcat 会附带一个简单的 HTTP 服务器，相较于真正的 HTTP 服务器而言，Tomcat 附带的 HTTP 服务器功能太过简单，仅作开发用途，不建议今后直接上线服务。



要编译 HelloServlet.java，类路径(Classpath)中必须包括 Servlet API 的相关类，如果使用的是 Tomcat，则这些类通常是封装在 Tomcat 目录的 lib 子目录的 servlet-api.jar 中。

要编写 Servlet 类，必须继承 HttpServlet 类，并重新定义 doGet()、doPost() 等对应 HTTP 请求的方法。容器会分别建立代表请求、响应的 HttpServletRequest 与 HttpServletResponse，可以从前者取得所有关于该次请求的相关信息，从后者对客户端进行各种响应。

在 Servlet 的 API 定义中，Servlet 是个接口，其中定义了与 Servlet 生命周期相关的 init()、destroy() 方法，以及提供服务的 service() 方法等。GenericServlet 实现了 Servlet 接口，不过它直接将 service() 标示为 abstract，GenericServlet 还实现了 ServletConfig 接口，将容器初始化 Servlet 调用 init() 时传入的 ServletConfig 封装起来。

真正在 service() 方法中定义了 HTTP 请求基本处理流程是 HttpServlet，而在 doGet()、doPost() 中传入的参数是 HttpServletRequest、HttpServletResponse，而不是通用的 ServletRequest、ServletResponse。

在 Servlet 3.0 中，可以使用 @WebServlet 标注(Annotation)来告知容器哪些 Servlet 会提供服务以及额外信息，也可以定义在部署描述文件 web.xml 中。一个 Servlet 至少会有三个名称，即类名称、注册的 Servlet 名称与 URL 模式(Pattern)名称。

Web 应用程序有几个要注意的目录与结构，WEB-INF 中的数据客户端无法直接请求取得，而必须通过请求的转发才有可能访问。web.xml 必须位于 WEB-INF 中。lib 目录用来放置 Web 应用程序会使用到的 JAR 文件。classes 目录用来放置编译好的.class 文件。可以将整个 Web 应用程序使用到的所有文件与目录封装为 WAR(Web Archive)文件，即后缀为.war 的文件，再利用 Web 应用程序服务器提供的工具来进行应用程序的部署。

一个请求 URI 实际上是由三个部分组成的：

```
requestURI = contextPath + servletPath + pathInfo
```

一个 JAR 文件中，除了可使用标注定义的 Servlet、监听器、过滤器外，也可以拥有自己的部署描述文件，这个文件的名称是 web-fragment.xml，必须放置在 JAR 文件的 META-INF 目录中。基本上，web.xml 中可定义的元素，在 web-fragment.xml 中也可以定义。

Servlet 3.0 对 web.xml 与标注的配置顺序并没有定义，对 web-fragment.xml 及标注的配置顺序也没有定义，然而可以决定 web.xml 与 web-fragment.xml 的配置顺序。

如果将 web.xml 中 <web-app> 的 metadata-complete 属性设置为 true(默认是 false)，则表示 web.xml 中已完成 Web 应用程序的相关定义，部署时将不会扫描标注与 web-fragment.xml 中的定义。

2.5 课后练习

2.5.1 选择题

1. 若要针对 HTTP 请求编写 Servlet 类, 以下()是正确的做法。
A. 实现 `Servlet` 接口 B. 继承 `GenericServlet`
C. 继承 `HttpServlet` D. 直接定义一个结尾名称为 `Servlet` 的类
2. 续上题, ()可以针对 HTTP 的 GET 请求进行处理与响应。
A. 重新定义 `service()` 方法 B. 重新定义 `doGet()` 方法
C. 定义一个方法名称为 `doService()` D. 定义一个方法名称为 `get()`
3. `HttpServlet` 是定义在()包中。
A. `javax.servlet` B. `javax.servlet.http`
C. `java.http` D. `javax.http`
4. 在 `web.xml` 中定义了以下内容:

```
<servlet>
    <servlet-name>Goodbye</servlet-name>
    <servlet-class>cc.openhome.LogoutServlet</servlet-class>
</servlet>
<servlet-mapping>
    <servlet-name>Goodbye</servlet-name>
    <url-pattern>/goodbye</url-pattern>
</servlet-mapping>
```

()URL 可以正确地要求 Servlet 进行请求处理。
A. /GoodBye B. /goodbye.do
C. /LogouotServlet D. /goodbye
5. 在 Web 容器中, 以下()两个接口的实例分别代表 HTTP 请求与响应回对象。
A. `HttpRequest` B. `HttpServletRequest`
C. `HttpServletResponse` D. `HttpPrintWriter`
6. 在 Web 应用程序中, ()负责将 HTTP 请求转换为 `HttpServletRequest` 对象。
A. Servlet 对象 B. HTTP 服务器
C. Web 容器 D. JSP 网页
7. 在 Web 应用程序的文件与目录结构中, `web.xml` 是放置在()中。
A. WEB-INF 目录 B. conf 目录
C. lib 目录 D. classes 目录



2.5.2 实训练习题

1. 编写一个 Servlet，当用户请求该 Servlet 时，显示用户于几点几分从哪个 IP(Internet Protocol)地址连线至服务器，以及发出的查询字符串(Query String)。

提示» 查询一下 `ServletRequest` 或 `HttpServletRequest` 的 API 帮助文档，了解有哪些方法可以使用。

2. 编写一个应用程序，可以让用户在窗体网页上输入名称、密码，若名称为 `caterpillar` 且密码为 `123456`，则显示一个 HTML 页面响应并有“登录成功”字样，否则显示“登录失败”字样，并由一个超链接连回窗体网页。注意：不可在地址栏上出现用户输入的名称、密码。

Servlet 进阶 API、

过滤器与监听器

学习目标：

- 了解 Servlet 生命周期
- 使用 ServletConfig 与 ServletContext
- 各种监听器的使用
- 实现 Filter 接口来开发过滤器

5.1 Servlet 进阶 API

每个 Servlet 都必须由 web 容器读取 Servlet 设置信息(无论使用标注还是 web.xml)、初始化等，才可以真正成为一个 Servlet。对于每个 Servlet 的设置信息，web 容器会为其生成一个 `ServletConfig` 作为代表对象，你可以从该对象取得 Servlet 初始参数，以及代表整个 web 应用程序的 `ServletContext` 对象。

本节将以讨论 Servlet 的生命周期作为开始，知道 `ServletConfig` 如何设置给 Servlet，如何设置为取得 Servlet 初始参数，以及如何使用 `ServletContext`。

5.1.1 Servlet、ServletConfig 与 GenericServlet

在 `Servlet` 接口上，定义了与 `Servlet` 生命周期及请求服务相关的 `init()`、`service()` 与 `destroy()` 三个方法。3.1.1 节曾经介绍，每一次请求来到容器时，会产生 `HttpServletRequest` 与 `HttpServletResponse` 对象，并在调用 `service()` 方法时当作参数传入(参考图 3.4)。

在 Web 容器启动后，会读取 Servlet 设置信息，将 Servlet 类加载并实例化，并为每个 Servlet 设置信息产生一个 `ServletConfig` 对象，而后调用 `Servlet` 接口的 `init()` 方法，并将产生的 `ServletConfig` 对象当作参数传入。如图 5.1 所示。

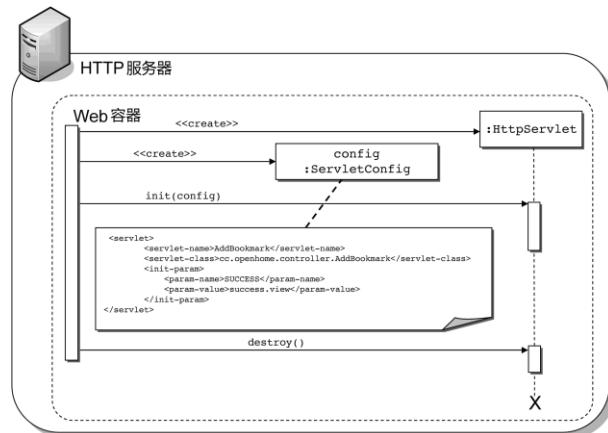


图 5.1 容器根据设置信息创建 Servlet 与 ServletConfig 实例

这个过程只会在创建 Servlet 实例后发生一次，之后每次请求到来，就如第 3 章所介绍的，调用 Servlet 实例的 `service()` 方法进行服务。

`ServletConfig` 即每个 `Servlet` 设置的代表对象，容器会为每个 `Servlet` 设置信息产生一个 `Servlet` 及 `ServletConfig` 实例。`GenericServlet` 同时实现了 `Servlet` 及 `ServletConfig`。如图 5.2 所示。

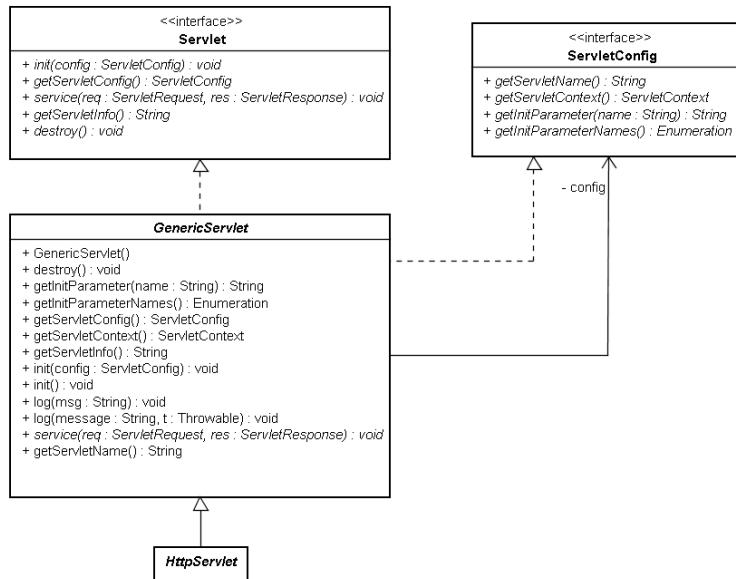


图 5.2 Servlet 类架构图

GenericServlet 主要的目的，就是将初始 Servlet 调用 `init()` 方法传入的 `ServletConfig` 封装起来：

```

private transient ServletConfig config;
public void init(ServletConfig config) throws ServletException {
    this.config = config;
    this.init();
}
public void init() throws ServletException {
}

```

GenericServlet 在实现 `Servlet` 的 `init()` 方法时，也调用了另一个无参数的 `init()` 方法，在编写 `Servlet` 时，如果有一些初始时所要运行的动作，可以重新定义这个无参数的 `init()` 方法，而不是直接重新定义有 `ServletConfig` 参数的 `init()` 方法。

注意» 当有一些对象实例化后所要运行的操作，必须定义构造器。在编写 `Servlet` 时，若想要运行与 `web` 应用程序资源相关的初始化动作，则要重新定义 `init()` 方法。举例来说，若想要使用 `ServletConfig` 来作一些事情，则不能在构造器中定义，因为实例化 `Servlet` 时，容器还没有调用 `init()` 方法传入 `ServletConfig`，所以不会有 `ServletConfig` 实例。

GenericServlet 也包括了 `Servlet` 与 `ServletConfig` 所定义方法的简单实现，实现内容主要是通过 `ServletConfig` 来取得一些相关信息。例如：

```

public ServletConfig getServletConfig() {
    return config;
}
public String getInitParameter(String name) {
    return getServletConfig().getInitParameter(name);
}

```

```

public Enumeration getInitParameterNames() {
    return getServletConfig().getInitParameterNames();
}
public ServletContext getServletContext() {
    return getServletConfig().getServletContext();
}

```

所以在继承 `HttpServlet` 实现 `Servlet` 时，就可以通过这些方法来取得所要的相关信息，而不用直接意识到 `ServletConfig` 的存在。

提示»» `GenericServlet` 还定义了 `log()` 方法。例如：

```

public void log(String msg) {
    getServletContext().log(getServletName() + ":" + msg);
}

```

这个方法主要是通过 `ServletContext` 的 `log()` 方法来运行日志功能。不过因为这个日志功能简单，实际上很少使用这个 `log()` 方法，而会使用功能更强大的日志 API，如 JDK 本身附带的日志包或 Log4j 等。

如果是使用 Tomcat，`ServletContext` 的 `log()` 方法所保存的日志文件会存放在 Tomcat 目录的 logs 目录下。

5.1.2 使用 `ServletConfig`

`ServletConfig` 相当于个别 `Servlet` 的设置信息代表对象，这意味着可以从 `ServletConfig` 中取得 `Servlet` 设置信息。`ServletConfig` 定义了 `getInitParameter()`、`getInitParameterNames()` 方法，可以取得设置 `Servlet` 时的初始参数。

若要使用标注设置个别 `Servlet` 的初始参数，可以在 `@WebServlet` 中使用 `@WebInitParam` 设置 `initParams` 属性。例如：

```

...
@WebServlet(name="ServletConfigDemo", urlPatterns={"/conf"},
    initParams={
        @WebInitParam(name = "PARAM1", value = "VALUE1"),
        @WebInitParam(name = "PARAM2", value = "VALUE2")
    }
)
public class ServletConfigDemo extends HttpServlet {
    private String PARAM1;
    private String PARAM2;
    public void init() throws ServletException {
        PARAM1 = getServletConfig().getInitParameter("PARAM1");
        PARAM2 = getServletConfig().getInitParameter("PARAM2");
    }
    ...
}

```



若要在 web.xml 中设置个别 Servlet 的初始参数，可以在 <servlet> 标签中使用 <init-param> 等标签进行设置，web.xml 中的设置会覆盖标注的设置。例如：

```
...
<servlet>
    <servlet-name>ServletConfigDemo</servlet-name>
    <servlet-class>cc.openhome.ServletConfigDemo</servlet-class>
    <init-param>
        <param-name>PARAM1</param-name>
        <param-value>VALUE1</param-value>
    </init-param>
    <init-param>
        <param-name>PARAM2</param-name>
        <param-value>VALUE2</param-value>
    </init-param>
</servlet>
...
```

注意»» 若要用 web.xml 覆盖标注设置，web.xml 的 <servlet-name> 设置必须与 @WebServlet 的 name 属性相同。

由于 ServletConfig 必须在 Web 容器将 Servlet 实例化后，调用有参数的 init() 方法再将之传入，是与 Web 应用程序资源相关的对象，所以在继承 HttpServlet 后，通常会重新定义无参数的 init() 方法以进行 Servlet 初始参数的取得。GenericServlet 定义了一些方法，将 servletConfig 封装起来，便于取得设置信息，所以取得 Servlet 初始参数的代码也可以改写为：

```
...
@WebServlet(name="ServletConfigDemo", urlPatterns={"/conf"},
    initParams={
        @WebInitParam(name = "PARAM1", value = "VALUE1"),
        @WebInitParam(name = "PARAM2", value = "VALUE2")
    }
)
public class AddMessage extends HttpServlet {
    private String PARAM1;
    private String PARAM2;
    public void init() throws ServletException {
        PARAM1 = getInitParameter("PARAM1");
        PARAM2 = getInitParameter("PARAM2");
    }
    ...
}
```

提示»» Servlet 初始参数通常作为常数设置，可以将一些 Servlet 程序默认值使用标注设为初始参数，之后若想变更那些信息，可以创建 web.xml 进行设置，以覆盖标注设置，而不用进行修改源代码、重新编译、部署的操作。

下面这个范例简单地示范如何设置、使用 Servlet 初始参数，其中登录成功与失败的网页，可以由初始参数设置来决定：

ConfigDemo Login.java

```
package cc.openhome;

import java.io.*;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.annotation.WebInitParam;

@WebServlet(
    name="Login", ←① 设置 Servlet 名称
    urlPatterns = {" /login.do" },
    initParams = {
        @WebInitParam(name = "SUCCESS", value = "success.view"),
        @WebInitParam(name = "ERROR", value = "error.view")
    }
)
public class Login extends HttpServlet {
    private String SUCCESS_VIEW;
    private String ERROR_VIEW;

    @Override
    public void init() throws ServletException {
        SUCCESS_VIEW = getInitParameter("SUCCESS"); ←③ 取得初
        ERROR_VIEW = getInitParameter("ERROR");始参数
    }

    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=UTF-8");
        String name = request.getParameter("name");
        String passwd = request.getParameter("passwd");
        if ("caterpillar".equals(name) && "123456".equals(passwd)) {
            request.getRequestDispatcher(SUCCESS_VIEW) ←④ 登录成功
                .forward(request, response);
        } else {
            request.getRequestDispatcher(ERROR_VIEW) ←⑤ 登录失败
                .forward(request, response);
        }
    }
}
```

注意 @WebServlet 的 name 属性设置①，如果 web.xml 中的设置要覆盖标注设置，<servlet-name> 的设置必须与 @WebServlet 的 name 属性相同，如果不设置 name 属性，将使用类名作为 servlet 名称。



性， 默认是类完整名称。程序中使用标注设置默认初始参数❸，并在 `init()` 中读取❹，成功❺或失败❻时所发送的网页 URL 是由初始参数来决定的。如果想使用 `web.xml` 来覆盖这些初始参数设置，则可以如下：

ConfigDemo web.xml

```
...
<servlet>
    <servlet-name>Login</servlet-name>      ← 注意 Servlet 名称
    <servlet-class>cc.openhome.Login</servlet-class>
    <init-param>
        <param-name>SUCCESS</param-name>
        <param-value>success.html</param-value>
    </init-param>
    <init-param>
        <param-name>ERROR</param-name>
        <param-value>error.html</param-value>
    </init-param>
</servlet>
<servlet-mapping>
    <servlet-name>Login</servlet-name>
    <url-pattern>/login.do</url-pattern>
</servlet-mapping>
...

```

以上设置 `web.xml` 你的成功与失败网页就分别设置为 `success.html` 及 `error.html` 了。

5.1.3 使用 `ServletContext`

`ServletContext` 接口定义了运行 `Servlet` 的应用程序环境的一些行为与观点，可以使用 `ServletContext` 实现对象来取得所请求资源的 URL、设置与储存属性、应用程序初始参数，甚至动态设置 `Servlet` 实例。

`ServletContext` 本身的名称令人困惑，因为它以 `Servlet` 名称作为开头，容易被误认为仅是单一 `Servlet` 的代表对象。事实上，当整个 Web 应用程序加载 Web 容器之后，容器会生成一个 `ServletContext` 对象作为整个应用程序的代表，并设置给 `ServletConfig`，只要通过 `ServletConfig` 的 `getServletContext()` 方法就可以取得 `ServletContext` 对象。以下则先简介几个需要注意的方法：

1. `getRequestDispatcher()`

用来取得 `RequestDispatcher` 实例，使用时路径的指定必须以“/”作为开头，这个斜杠代表应用程序环境根目录(Context Root)。正如 3.2.5 节中的说明，取得 `RequestDispatcher` 实例之后，就可以进行请求的转发(Forward)或包含(Include)。

```
context.getRequestDispatcher("/pages/some.jsp")
    .forward(request, response);
```

提示» 以“/”作为开头有时称为环境相对(Context-relative)路径，没有以“/”作为开头则称为请求相对(Request-relative)路径。实际上 `HttpServletRequest` 的 `getRequestDispatcher()` 方法在实现时，若是环境相对路径，则直接委托给 `ServletContext` 的 `getRequestDispatcher()`；若是请求相对路径，则转换为环境相对路径，再委托给 `ServletContext` 的 `getRequestDispatcher()` 来取得 `RequestDispatcher`。

2. `getResourcePaths()`

如果想要知道 Web 应用程序的某个目录中有哪些文件，则可以使用 `getResourcePaths()` 方法。例如：

```
for(String avatar : getServletContext().getResourcePaths("/")) {  
    // 显示 avatar 文字...  
}
```

使用时指定路径必须以“/”作为开头，表示相对于应用程序环境根目录，返回的路径会如以下所示：

```
/welcome.html  
/catalog/  
/catalog/index.html  
/catalog/products.html  
/customer/  
/customer/login.jsp  
/WEB-INF/  
/WEB-INF/web.xml  
/WEB-INF/classes/com.acme.OrderServlet.class
```

可以看到，这个方法会连同 WEB-INF 的信息都列出来。如果是个目录信息，则会以“/”作结尾。以下这个范例利用了 `getResourcePaths()` 方法，自动取得 avatars 目录下的图片路径，并通过``标签来显示图片：

ContextDemo Avatar.java

```
package cc.openhome;  
  
import java.io.*;  
  
import javax.servlet.ServletException;  
import javax.servlet.annotation.WebServlet;  
import javax.servlet.annotation.WebInitParam;  
import javax.servlet.http.HttpServlet;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpServletResponse;  
  
@WebServlet(  
    urlPatterns = {"/avatar.view"},  
    initParams = {  
        @WebInitParam(name = "AVATAR_DIR", value = "/avatars")  
    }  
)  
public class Avatar extends HttpServlet {  
    private String AVATAR_DIR;
```

```

@Override
public void init() throws ServletException {
    AVATAR_DIR = getInitParameter("AVATAR_DIR");
}

protected void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
    response.setContentType("text/html;charset=UTF-8");
    PrintWriter out = response.getWriter();
    out.println("<html>");
    out.println("<head>");
    out.println("<title>头像显示</title>");
    out.println("</head>");
    out.println("<body>");
    for (String avatar : getServletContext()
            .getResourcePaths(AVATAR_DIR)) { ← 取得头像路径
        avatar = avatar.replaceFirst("/", "");
        out.println("<img src='" + avatar + "'>"); ← 设置<img>的 src 属性
    }
    out.println("</body>");
    out.println("</html>");
    out.close();
}
}

```

3. getResourceAsStream()

如果想在 Web 应用程序中读取某个文件的内容，则可以使用 `getResourceAsStream()` 方法，使用时指定路径必须以“/”作为开头，表示相对于应用程序环境根目录，或者相对是/WEB-INF/lib 中 JAR 文件里 META-INF/resources 的路径，运行结果会返回 `InputStream` 实例，接着就可以运用它来读取文件内容。

在 3.3.3 节中有个读取 PDF 的范例，其中示范过 `getResourceAsStream()` 方法的使用，可以直接参考该范例，这里不再重复示范。

注意»» 你也许会想到使用 `java.io` 下的 `File`、`FileReader`、`FileInputStream` 等与文件读取相关的类。使用这些类时，可以指定绝对路径或相对路径。绝对路径自然是指文件在服务器上的真实路径。必须注意的是，用相对路径指定时，此时路径不是相对于 Web 应用程序根目录，而是相对于启动 Web 容器时的命令执行目录，这是许多初学者都会有的误解。以 Tomcat 来说，若在 Servlet 中执行以下语句：

```
out.println(new File("filename").getAbsolutePath());
```

则会显示 `filename` 是位于 Tomcat 目录下的 bin 目录中，例如：

C:\Program Files\Apache Software Foundation\Apache Tomcat 7.0.8\bin\filename
这样的路径。

每个 Web 应用程序都会有一个相对应的 `ServletContext`，针对“应用程序”初始化时需用到的一些参数数据，可以在 `web.xml` 中设置应用程序初始参数，通常这会结合 `ServletContextListener` 来做。关于监听器(Listener)的使用，在下一节进行说明。

5.2 应用程序事件、监听器

Web 容器管理 Servlet/JSP 相关的对象生命周期，若对 `HttpServletRequest` 对象、`HttpSession` 对象、`ServletContext` 对象在生成、销毁或相关属性设置发生的时机点有兴趣，则可以实现对应的监听器(Listener)，做好相关的设置，这样在对应的时机点发生时，Web 容器就会调用监听器上相对应的方法，让你在对应的时机点做些处理。

5.2.1 ServletContext 事件、监听器

与 `ServletContext` 相关的监听器有 `ServletContextListener` 与 `ServletContextAttributeListener`。

1. ServletContextListener

`ServletContextListener` 是“生命周期监听器”，如果想要知道何时 Web 应用程序已经初始化或即将结束销毁，可以实现 `ServletContextListener`:

```
package javax.servlet;
import java.util.EventListener;
public interface ServletContextListener extends EventListener {
    public void contextInitialized(ServletContextEvent sce);
    public void contextDestroyed(ServletContextEvent sce);
}
```

在 Web 应用程序初始化后或即将结束销毁前，会调用 `ServletContextListener` 实现类相对应的 `contextInitialized()` 或 `contextDestroyed()`。可以在 `contextInitialized()` 中实现应用程序资源的准备动作，在 `contextDestroyed()` 实现释放应用程序资源的动作。



例如，可以实现 `ServletContextListener`，在应用程序初始过程中，准备好数据库连线对象、读取应用程序设置等动作，如放置使用头像的目录信息，就不宜将目录名称写死，以免日后目录变动名称或位置时，所有相关的 Servlet 都需要进行源代码的修改。这时可以这么做：

ContextDemo2 ContextParameterReader.java

```
package cc.openhome;
import javax.servlet.ServletContext;
```

```

import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;
@WebListener ← ① 使用 @WebListener 标注
public class ContextParameterReader implements ServletContextListener {
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext context = sce.getServletContext(); ← ③ 取得 ServletContext
        String avatars = context.getInitParameter("AVATAR"); ← ④ 取得初始参数
        context.setAttribute("avatars", avatars); ← ⑤ 设置 ServletContext 属性
    }
    public void contextDestroyed(ServletContextEvent sce) {}
}

```

② 实现 ServletContextListener

ServletContextListener 可以直接使用 `@WebListener` 标注 ①，而且必须实现 `ServletContextListener` 接口 ②，这样容器就会在启动时加载并运行对应的方法。当 Web 容器调用 `contextInitialized()` 或 `contextDestroyed()` 时，会传入 `ServletContextEvent`，其封装了 `ServletContext`，可以通过 `ServletContextEvent` 的 `getServletContext()` 方法取得 `ServletContext` ③，通过 `ServletContext` 的 `getInitParameter()` 方法来读取初始参数 ④，因此 Web 应用程序初始参数常被称为 `ServletContext` 初始参数。

在整个 Web 应用程序生命周期，Servlet 需共享的资料可以设置为 `ServletContext` 属性。由于 `ServletContext` 在 Web 应用程序存活期间都会一直存在，所以设置为 `ServletContext` 属性的数据，除非主动移除，否则也是一直存活于 Web 应用程序中。

可以通过 `ServletContext` 的 `setAttribute()` 方法设置对象为 `ServletContext` 属性 ⑤，之后可通过 `ServletContext` 的 `getAttribute()` 方法取出该属性。若要移除属性，则通过 `ServletContext` 的 `removeAttribute()` 方法。



因为 `@WebListener` 没有设置初始参数的属性，所以仅适用于无须设置初始参数的情况。如果需要设置初始参数，可以在 `web.xml` 中设置：

ContextDemo2 web.xml

```

...
<context-param>
    <param-name>AVATAR</param-name>
    <param-value>/avatars</param-value>
</context-param>
...

```



在 `web.xml` 中，使用 `<context-param>` 标签来定义初始参数。由于先前的 `ContextParameterReader` 读取的初始参数已设置为 `ServletContext` 属性，因此先前的头像范例，必须做点修改：

ContextDemo2 Avatar.java

```
import java.io.*;
```

```

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

@WebServlet("/avatar.view")  
① 仅设置 URL 模式
public class Avatar extends HttpServlet {
    private String AVATAR_DIR;

    @Override
    public void init() throws ServletException {
        AVATAR_DIR =
            (String) getServletContext().getAttribute("avatars");  
② 取得 ServletContext
    }
    ...
}

```

程序中仅列出了改写后需要注意的部分。主要就是不再需要设置 `ServletConfig` 初始参数①，以及从 `ServletContext` 中取出先前所设置的属性②。

在 Servlet 3.0 之前，`ServletContextListener` 实现类必须在 `web.xml` 中设置。例如：

```

...
<listener>
    <listener-class>cc.openhome.ContextParameterReader</listener-class>
</listener>
...

```

在 `web.xml` 中，也使用了 `<listener>` 与 `<listener-class>` 标签来定义实现了 `ServletContextListener` 接口的类名称。

有些应用程序的设置，必须在 Web 应用程序初始时进行，例如 4.2.2 节中谈过，若要改变 `HttpSession` 的一些 Cookie 设置，可以在 `web.xml` 中定义。另一个方式，则是取得 `ServletContext` 后，使用 `getSessionCookieConfig()` 取得 `SessionCookieConfig` 进行设置，不过这个动作必须在应用程序初始时进行。例如：

```

...
@WebListener()
public class SomeContextListener implements ServletContextListener {
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext context = sce.getServletContext();
        context.getSessionCookieConfig()
            .setName("caterpillar-sessionId");
    }
    @Override
    public void contextDestroyed(ServletContextEvent sce) {}
}

```



2. ServletContextAttributeListener

`ServletContextAttributeListener` 是“监听属性改变的监听器”，如果想要对象被设置、移除或替换 `ServletContext` 属性，可以收到通知以进行一些操作，则可以实现 `ServletContextAttributeListener`。

```
package javax.servlet;
import java.util.EventListener;
public interface ServletContextAttributeListener extends EventListener{
    public void attributeAdded(ServletContextAttributeEvent scab);
    public void attributeRemoved(ServletContextAttributeEvent scab);
    public void attributeReplaced(ServletContextAttributeEvent scab);
}
```

当在 `ServletContext` 中添加属性、移除属性或替换属性时，相对应的 `attributeAdded()`、`attributeRemoved()` 与 `attributeReplaced()` 方法就会被调用。

如果希望容器在部署应用程序时，实例化实现 `ServletContextAttributeListener` 的类并注册给应用程序，同样也是在实现类上标注 `@WebListener`，并实现 `ServletContextAttributeListener` 接口：

```
...
@WebListener()
public class SomeContextAttrListener
    implements ServletContextAttributeListener {
...
}
```

另一个方式是在 `web.xml` 中设置：

```
...
<listener>
    <listener-class>cc.openhome.SomeContextAttrListener</listener-class>
</listener>
...
```

5.2.2 HttpSession 事件、监听器

与 `HttpSession` 相关的监听器有四个：`HttpSessionListener`、`HttpSessionAttributeListener`、`HttpSessionBindingListener` 与 `HttpSessionActivationListener`。

1. HttpSessionListener

`HttpSessionListener` 是“生命周期监听器”，如果想要在 `HttpSession` 对象创建或结束时，做些相对应动作，则可以实现 `HttpSessionListener`。

```
package javax.servlet.http;
import java.util.EventListener;
public interface HttpSessionListener extends EventListener {
    public void sessionCreated(HttpSessionEvent se);
    public void sessionDestroyed(HttpSessionEvent se);
}
```

在 `HttpSession` 对象初始化或结束前，会分别调用 `sessionCreated()` 与 `sessionDestroyed()` 方法，可以通过传入的 `HttpSessionEvent`，使用 `getSession()` 取得 `HttpSession`，以针对会话对象作出相对应的创建或结束处理操作。

举个例子，有些网站为了防止用户重复登录，会在数据库中以某个字段代表用户是否登录，用户登录后，在数据库中设置该字段信息，代表用户已登录，而用户注销后，再重置该字段。如果用户已登录，在注销前尝试再用另一个浏览器进行登录，应用程序会检查数据库中代表登录与否的字段，如果发现已被设置为登录，则拒绝用户重复登录。

现在的问题在于，如果用户在注销前不小心关闭浏览器，没有确实运行注销操作，那么数据库中代表登录与否的字段就不会被重置。为此，可以实现 `HttpSessionListener`，由于 `HttpSession` 有其存活期限，当容器销毁某个 `HttpSession` 时，就会调用 `sessionDestroyed()`，就可以在当中判断要重置哪个用户数据库中代表登录与否的字段。例如：

```
...
@WebListener()
public class ResetLoginHelper implements HttpSessionListener {
    @Override
    public void sessionCreated(HttpSessionEvent se) {}
    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        HttpSession session = se.getSession();
        String user = session.getAttribute("login");
        // 修改数据库字段为注销状态
    }
}
```

如果在实现 `HttpSessionListener` 的类上标注 `@WebListener`，则容器在部署应用程序时，会实例化并注册给应用程序。另一个方式是在 `web.xml` 中设置：

```
...
<listener>
    <listener-class>cc.openhome.ResetLoginHelper</listener-class>
</listener>
...
```

下面来看另一个 `HttpSessionListener` 的应用实例。假设有个应用程序在用户登录后会使用 `HttpSession` 对象来进行会话管理。例如：

SessionListenerDemo Login.java

```
package cc.openhome;

import java.util.*;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```

```

@WebServlet("/login.do")
public class Login extends HttpServlet {
    private Map<String, String> users;

    public Login() {
        users = new HashMap<String, String>();
        users.put("caterpillar", "123456");
        users.put("momor", "98765");
        users.put("hamimi", "13579");
    }

    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {
        String name = request.getParameter("name");
        String passwd = request.getParameter("passwd");

        String page = "form.html";
        if(users.containsKey(name) && users.get(name).equals(passwd)) {
            request.getSession().setAttribute("user", name);
            page = "welcome.view";
        }
        response.sendRedirect(page);
    }
}

```

这个 **Servlet** 在用户验证通过后，会取得 `HttpSession` 实例并设置属性。如果想要在应用程序中加上显示目前已登录在线人数的功能，则可以实现 `HttpSessionListener` 接口。例如：

SessionListenerDemo OnlineUserCounter.java

```

package cc.openhome;

import javax.servlet.annotation.WebListener;
import javax.servlet.http.HttpSessionEvent;
import javax.servlet.http.HttpSessionListener;

@WebListener
public class OnlineUserCounter implements HttpSessionListener {
    private static int counter;

    public static int getCounter() {
        return counter;
    }

    @Override
    public void sessionCreated(HttpSessionEvent se) {
        OnlineUserCounter.counter++;
    }

    @Override
    public void sessionDestroyed(HttpSessionEvent se) {
        OnlineUserCounter.counter--;
    }
}

```

`OnlineUserCounter` 中有个静态(`static`)变量，在每一次 `HttpSession` 创建时会递增，而销毁 `HttpSession` 时会递减，也就是通过统计 `HttpSession` 的实例，来作登录用户的计数功能。

只要在想要显示在线人数的页面，使用 `OnlineUserCounter.getCounter()`，就可以取得目前的在线人数并显示，如图 5.3 所示。例如，在登录成功的欢迎页面上，一并显示在线人数：

SessionListenerDemo Welcome.java

```
package cc.openhome;

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

@WebServlet("/welcome.view")
public class Welcome extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        HttpSession session = request.getSession(false);
        out.println("<html>");
        out.println("<head>");
        out.println("<title>欢迎</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>目前在线人数 " +
                   OnlineUserCounter.getCounter() + " 人</h1>");
        if(session != null) {
            String user = (String) session.getAttribute("user");
            out.println("<h1>欢迎: " + user + "</h1>");
            out.println("<a href='logout.do'>注销</a>");
        }
        out.println("</body>");
        out.println("</html>");
        out.close();
    }
}
```



图 5.3 在线人数统计

提示» 可以把这个例子进一步扩充，不只统计在线人数，还可以实现一个查看在线用户信息的列表。本章课后练习中，有个实训题要求实现这个功能。

2. HttpSessionAttributeListener

`HttpSessionAttributeListener` 是“属性改变监听器”，当在会话对象中加入属性、移除属性或替换属性时，相对应的 `attributeAdded()`、`attributeRemoved()` 与 `attributeReplaced()` 方法就会被调用，并分别传入 `HttpSessionBindingEvent`。

```
package javax.servlet.http;
import java.util.EventListener;
public interface HttpSessionAttributeListener extends EventListener {
    public void attributeAdded(HttpSessionBindingEvent se);
    public void attributeRemoved(HttpSessionBindingEvent se);
    public void attributeReplaced(HttpSessionBindingEvent se);
}
```

`HttpSessionBindingEvent` 有个 `getName()` 方法，可以取得属性设置或移除时指定的名称，而 `getValue()` 可以取得属性设置或移除时的对象。

如果希望容器在部署应用程序时，实例化实现 `HttpSessionAttributeListener` 的类并注册给应用程序，则同样也是在实现类上标注 `@WebListener`:

```
...
@WebListener()
public class HttpSessionAttrListener
    implements HttpSessionAttributeListener {
...
}
```

另一个方式是在 `web.xml` 下进行设置:

```
...
<listener>
    <listener-class>cc.openhome.HttpSessionAttrListener</listener-class>
</listener>
...
```

3. HttpSessionBindingListener

`HttpSessionBindingListener` 是“对象绑定监听器”，如果有对象即将加入 `HttpSession` 的属性对象，希望在设置给 `HttpSession` 成为属性或从 `HttpSession` 中移

除时，可以收到 `HttpSession` 的通知，则可以让该对象实现 `HttpSessionBindingListener` 接口。

```
package javax.servlet.http;
import java.util.EventListener;
public interface HttpSessionBindingListener extends EventListener {
    public void valueBound(HttpSessionBindingEvent event);
    public void valueUnbound(HttpSessionBindingEvent event);
}
```

这个接口即是实现加入 `HttpSession` 的属性对象，不需注释或在 `web.xml` 中设置。当实现此接口的属性对象被加入 `HttpSession` 或从中移除时，就会调用对应的 `valueBound()` 与 `valueUnbound()` 方法，并传入 `HttpSessionBindingEvent` 对象，可以通过该对象的 `getSession()` 取得 `HttpSession` 对象。

下面介绍这个接口使用的一个范例。假设修改前一个范例程序的登录 `Servlet` 如下：

SessionListenerDemo2 Login.java

```
package cc.openhome;

import java.util.*;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet("/login.do")
public class Login extends HttpServlet {
    ...
    @Override
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        ...
        String page = "form.html";
        if(users.containsKey(name) && users.get(name).equals(passwd)) {
            User user = new User(name);
            request.getSession().setAttribute("user", user);
            page = "welcome.view";
        }
        response.sendRedirect(page);
    }
}
```

当用户输入正确的名称与密码时，首先会以用户名来创建 `User` 实例，而后加入 `HttpSession` 中作为属性。希望 `User` 实例被加入成为 `HttpSession` 属性时，可以从数据库中加载用户的其他数据，如地址、照片等，或是在日志中记录用户登录的信息，可以让 `User` 类实现 `HttpSessionBindingListener` 接口。例如：



SessionListenerDemo2 User.java

```

package cc.openhome;

import javax.servlet.http.HttpSessionBindingEvent;
import javax.servlet.http.HttpSessionBindingListener;

public class User implements HttpSessionBindingListener {
    private String name;
    private String data;
    public User(String name) {
        this.name = name;
    }

    public void valueBound(HttpSessionBindingEvent event) {
        this.data = name + " 来自数据库的数据...";
    }

    public void valueUnbound(HttpSessionBindingEvent event) {}

    public String getData() {
        return data;
    }

    public String getName() {
        return name;
    }
}

```

在 `valueBound()` 中，可以实现查询数据库的功能(也许是委托给一个负责查询数据库的服务对象)，并补齐 `User` 对象中的相关数据。当 `HttpSession` 失效前会先移除属性，或者主动移除属性时，则 `valueUnbound()` 方法会被调用。

4. HttpSessionActivationListener

`HttpSessionActivationListener` 是“对象迁移监听器”，其定义了两个方法 `sessionWillPassivate()` 与 `sessionDidActivate()`。很多情况下，几乎不会使用到 `HttpSessionActivationListener`。在使用到分布式环境时，应用程序的对象可能分散在多个 JVM 中。当 `HttpSession` 要从一个 JVM 迁移至另一个 JVM 时，必须先在原本的 JVM 上序列化(Serialize)所有的属性对象，在这之前若属性对象有实现 `HttpSessionActivationListener`，就会调用 `sessionWillPassivate()` 方法，而 `HttpSession` 迁移至另一个 JVM 后，就会对所有属性对象作反序列化，此时会调用 `sessionDidActivate()` 方法。

提示» 要可以序列化的对象必须实现 `Serializable` 接口。如果 `HttpSession` 属性对象中有些类成员无法作序列化，则可以在 `sessionWillPassivate()` 方法中做些替代处理来保存该成员状态，而在 `sessionDidActivate()` 方法中做些恢复该成员状态的动作。

5.2.3 HttpServletRequest 事件、监听器

与请求相关的监听器有三个：`ServletRequestListener`、`ServletRequestAttributeListener`与`AsyncListener`。第三个是在Servlet 3.0中新增的监听器，这在之后谈到异步处理时还会说明。以下先说明前两个监听器。

1. ServletRequestListener

`ServletRequestListener`是“生命周期监听器”，如果想要在`HttpServletRequest`对象生成或结束时做些相对应的操作，则可以实现`ServletRequestListener`。

```
package javax.servlet;
import java.util.EventListener;
public interface ServletRequestListener extends EventListener {
    public void requestDestroyed(ServletRequestEvent sre);
    public void requestInitialized(ServletRequestEvent sre);
}
```

在`ServletRequest`对象初始化或结束前，会调用`requestInitialized()`与`requestDestroyed()`方法，可以通过传入的`ServletRequestEvent`来取得`ServletRequest`，以针对请求对象做出相对应的初始化或结束处理动作。例如：

```
...
@WebListener()
public class SomeRequestListener implements ServletRequestListener {
    ...
}
```

如果在实现`ServletRequestListener`的类上标注`@WebListener`，则容器在部署应用程序时，会实例化类并注册给应用程序。另一个方式是在`web.xml`中进行设置：

```
...
<listener>
    <listener-class>cc.openhome.SomeRequestListener</listener-class>
</listener>
...
```

2. ServletRequestAttributeListener

`ServletRequestAttributeListener`是“属性改变监听器”，在请求对象中加入属性、移除属性或替换属性时，相对应的`attributeAdded()`、`attributeRemoved()`与`attributeReplaced()`方法就会被调用，并分别传入`ServletRequestAttributeEvent`。

`ServletRequestAttributeEvent`有个`getName()`方法，可以取得属性设置或移除时指定的名称，而`getValue()`则可以取得属性设置或移除时的对象。

如果希望容器在部署应用程序时，实例化实现`ServletRequestAttributeListener`的类并注册给应用程序，同样也是在实现类上标注`@WebListener`：

```
...
@WebListener()
```



```
public class SomeRequestAttrListener
    implements ServletRequestAttributeListener {
    ...
}
```

另一个方式是在 web.xml 中进行设置：

```
...
<listener>
    <listener-class>cc.openhome.SomeRequestListener</listener-class>
</listener>
...
```

提示» 生命周期监听器与属性改变监听器都必须使用 @WebListener 或在 web.xml 中设置，容器才会知道要加载、读取监听器相关设置。

5.3 过滤器

在容器调用 Servlet 的 service() 方法前，Servlet 并不会知道有请求的到来，而在 Servlet 的 service() 方法运行后，容器真正对浏览器进行 HTTP 响应之前，浏览器也不会知道 Servlet 真正的响应是什么。过滤器(Filter)正如其名称所示，是介于 Servlet 之前，可拦截过滤浏览器对 Servlet 的请求，也可以改变 Servlet 对浏览器的响应。

本节将介绍过滤器的运用概念，了解如何实现 Filter 接口来编写过滤器，如何在 web.xml 中设置过滤器、改变过滤器的顺序等，以及如何使用请求封装器(Wrapper)及响应封装器，将容器产生的请求与响应对象加以包装，针对某些请求信息或响应进行加工处理。

5.3.1 过滤器的概念

想象已经开发好应用程序的主要商务功能了，但现在有几个需求出现：

- (1) 针对所有的 Servlet，产品经理想要了解从请求到响应之间的时间差。
- (2) 针对某些特定的页面，客户希望只有特定几个用户才可以浏览。
- (3) 基于安全方面的考量，用户输入的特定字符必须过滤并替换为无害的字符。
- (4) 请求与响应的编码从 Big5 改用 UTF-8。

以第一个需求而言，也许你的直觉就是，打开每个 Servlet，在 doXXX() 开头与结尾取得系统时间，计算时间差，但如果页面有上百个或上千个，怎么完成这些需求？如果产品经理在你完成需求后，又要求拿掉计算时间差的功能，你怎么办？

收到这些需求的你，在急忙打开相关源代码文档进行修改之前，请先分析一下这些需求：

- (1) 运行 Servlet 的 `service()` 方法“前”，记录起始时间，Servlet 的 `service()` 方法运行“后”，记录结束时间并计算时间差。
- (2) 运行 Servlet 的 `service()` 方法“前”，验证是否为允许的用户。
- (3) 运行 Servlet 的 `service()` 方法“前”，对请求参数进行字符过滤与替换。
- (4) 运行 Servlet 的 `service()` 方法“前”，对请求与响应对象设置编码。

经过以上分析，可以发现这些需求，可以在真正运行 Servlet 的 `service()` 方法“前”与 Servlet 的 `service()` 方法运行“后”中间进行实现，如图 5.4 所示。

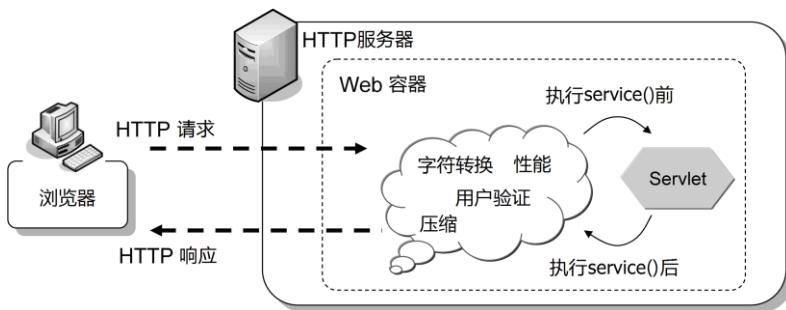


图 5.4 介于 `service()` 方法运行前、后的请求

性能评测、用户验证、字符替换、编码设置等需求，基本上与应用程序的业务需求没有直接的关系，只是应用程序额外的元件服务之一。你可能只是短暂需要它，或者需要整个系统应用相同设置，不应该为了一时的需要而修改代码强加入原有业务流程中。例如，性能的评测也许只是开发阶段才需要的，上线之后就要拿掉性能评测的功能，如果直接将性能评测的代码编写在业务流程中，那么要拿掉这个功能，就又得再修改一次源代码。

因此，如性能评测、用户验证、字符替换、编码设置这类的需求，应该设计为独立的元件，随时可以加入应用程序中，也随时可以移除，或随时可以修改设置而不用修改原有的程序。这类元件就像是一个过滤器，安插在浏览器与 Servlet 中间，可以过滤请求与响应而作进一步的处理，如图 5.5 所示。

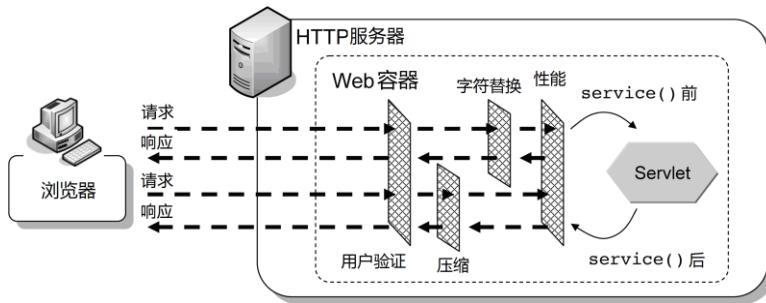


图 5.5 将服务需求设计为可抽换的元件

Servlet/JSP 提供了过滤器机制让你实现这些元件服务，就如图 5.5 所示，可以视需求抽换过滤器或调整过滤器的顺序，也可以针对不同的 URL 应用不同的过滤器。甚至在不同的 Servlet 间请求转发或包含时应用过滤器，如图 5.6 所示。

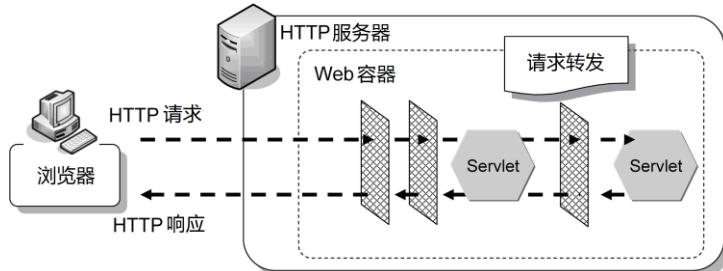


图 5.6 在请求转发时应用过滤器

5.3.2 实现与设置过滤器

在 Servlet/JSP 中要实现过滤器，必须实现 `Filter` 接口，并使用 `@WebFilter` 标注或在 `web.xml` 中定义过滤器，让容器知道该加载哪些过滤器类。`Filter` 接口有三个要实现的方法：`init()`、`doFilter()` 与 `destroy()`。

```
package javax.servlet;
import java.io.IOException;
public interface Filter {
    public void init(FilterConfig filterConfig) throws ServletException;
    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain) throws IOException, ServletException;
    public void destroy();
}
```

`FilterConfig` 类似于 `Servlet` 接口 `init()` 方法参数上的 `ServletConfig`，`FilterConfig` 是实现 `Filter` 接口的类上使用标注或 `web.xml` 中过滤器设置信息的代表对象。如果在定义过滤器时设置了初始参数，则可以通过 `FilterConfig` 的 `getInitParameter()` 方法来取得初始参数。

`Filter` 接口的 `doFilter()` 方法则类似于 `Servlet` 接口的 `service()` 方法。当请求来到容器，而容器发现调用 `Servlet` 的 `service()` 方法前，可以应用某过滤器时，就会调用该过滤器的 `doFilter()` 方法。可以在 `doFilter()` 方法中进行 `service()` 方法的前置处理，而后决定是否调用 `FilterChain` 的 `doFilter()` 方法。

如果调用了 `FilterChain` 的 `doFilter()` 方法，就会运行下一个过滤器，如果没有下一个过滤器了，就调用请求目标 `Servlet` 的 `service()` 方法。如果因为某个情况(如用户没有通过验证)而没有调用 `FilterChain` 的 `doFilter()`，则请求就不会继续交给接下来的过滤器或目标 `Servlet`。这时就是所谓的拦截请求(从 `Servlet` 的观点来看，根本不知道浏览器有发出请求)。`FilterChain` 的 `doFilter()` 实现，概念上类似以下：

```
Filter filter = filterIterator.next();
if(filter != null) {
    filter.doFilter(request, response, this);
}
else {
    targetServlet.service(request, response);
}
```

在陆续调用完 `Filter` 实例的 `doFilter()` 仍至 `Servlet` 的 `service()` 之后，流程会以堆栈顺序返回，所以在 `FilterChain` 的 `doFilter()` 运行完毕后，就可以针对 `service()` 方法做后续处理。

```
// service()前置处理
chain.doFilter(request, response);
// service()后置处理
```

只需要知道 `FilterChain` 运行后会以堆栈顺序返回即可。在实现 `Filter` 接口时，不用理会这个 `Filter` 前后是否有其他 `Filter`，应该将之作为一个独立的元件设计。

如果在调用 `Filter` 的 `doFilter()` 期间，因故抛出 `UnavailableException`，此时不会继续下一个 `Filter`，容器可以检验异常的 `isPermanent()`，如果不是 `true`，则可以在稍后重试 `Filter`。

提示» `Servlet/JSP` 提供的过滤器机制，其实是 Java EE 设计模式中 `Interceptor Filter` 模式的实现。如果希望可以弹性地抽换某功能的前置与后置处理元件(例如 `Servlet/JSP` 中 `Servlet` 的 `service()` 方法的前置与后置处理)，就可以应用 `Interceptor Filter` 模式。

以下实现一个简单的性能评测过滤器，可用来记录请求与响应间的时间差，了解 `Servlet` 处理请求到响应所需花费的时间。

FilterDemo PerformanceFilter.java

```
package cc.openhome;
import java.io.IOException;
```



```

import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;

@WebFilter(filterName="performance", urlPatterns={"/"})  
① 使用 @WebFilter 标注
public class PerformanceFilter implements Filter {  
② 实现 Filter 接口
    private FilterConfig config;

    @Override
    public void init(FilterConfig config) throws ServletException {
        this.config = config;
    }

    @Override
    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
        throws IOException, ServletException {
        long begin = System.currentTimeMillis();
        chain.doFilter(request, response);
        config.getServletContext().log("Request process in " +
            (System.currentTimeMillis() - begin) + " milliseconds");
    }

    @Override
    public void destroy() {}
}

```

当过滤器类被载入容器时并实例化后，容器会运行其 `init()` 方法并传入 `FilterConfig` 对象作为参数。在 `doFilter()` 的实现中，先记录目前的系统时间，接着调用 `FilterChain` 的 `doFilter()` 继续接下来的过滤器或 `Servlet`，当 `FilterChain` 的 `doFilter()` 返回时，取得系统时间并减去先前记录的时间，就是请求与响应间的时间差。

过滤器的设置与 `Servlet` 的设置很类似。`@WebFilter` 中的 `filterName` 设置过滤器名称，`urlPatterns` 设置哪些 URL 请求必须应用哪个过滤器①，可应用的 URL 模式与 `Servlet` 基本上相同，而 “`/*`” 表示应用在所有的 URL 请求，过滤器还必须实现 `Filter` 接口②。

如果要在 `web.xml` 中设置，则可以如下所示，标注的设置会被 `web.xml` 中的设置覆盖：

```

...
<filter>
    <filter-name>performance</filter-name>
    <filter-class>cc.openhome.PerformanceFilter</filter-class>
</filter>
<filter-mapping>
    <filter-name>performance</filter-name>

```

```

<url-pattern>/*</url-pattern>
</filter-mapping>
...

```

<filter>标签中使用<filter-name>与<filter-class>设置过滤器名称与类名称。而在<filter-mapping>中，则用<filter-name>与<url-pattern>来设置哪些 URL 请求必须应用哪个过滤器。

在过滤器的请求应用上，除了指定 URL 模式之外，也可以指定 Servlet 名称，这可以通过@WebServlet 的servletNames来设置：

```
@WebFilter(filterName="performance", servletNames={"SomeServlet"})
```

或在 web.xml 的<filter-mapping>中使用<servlet-name>来设置：

```

...
<filter-mapping>
    <filter-name>performance</filter-name>
    <servlet-name>SomeServlet</servlet-name>
</filter-mapping>
...

```

如果想一次符合所有的 Servlet 名称，则可以使用星号(*)。如果在过滤器初始化时，想要读取一些参数，可以在@WebFilter 中使用@WebInitParam 设置initParams。例如：

```

...
@WebFilter(
    filterName="performance",
    urlPatterns={"//*"}, servletNames={""},
    initParams={
        @WebInitParam(name = "PARAM1", value = "VALUE1"),
        @WebInitParam(name = "PARAM2", value = "VALUE2")
    }
)
public class PerformanceFilter implements Filter {
    private String PARAM1;
    private String PARAM2;

    @Override
    public void init(FilterConfig config) throws ServletException {
        PARAM1 = config.getInitParameter("PARAM1");
        PARAM2 = config.getInitParameter("PARAM2");
    }
    ...
}

```

若要在 web.xml 中设置过滤器的初始参数，可以在<filter>标签中使用<init-param>进行设置，web.xml 中的设置会覆盖标注的设置。例如：

```

...
<filter>
    <filter-name>PerformanceFilter</filter-name>
    <filter-class>cc.openhome.PerformanceFilter</filter-class>
    <init-param>

```



```

<param-name>PARAM1</param-name>
<param-value>VALUE1</param-value>
</init-param>
<init-param>
    <param-name>PARAM2</param-name>
    <param-value>VALUE2</param-value>
</init-param>
</filter>
...

```

触发过滤器的时机，默认是浏览器直接发出请求。如果是那些通过 RequestDispatcher 的 forward() 或 include() 的请求，设置 @WebFilter 的 dispatcherTypes。例如：

```

@WebFilter(
    filterName="some",
    urlPatterns={"/some"},
    dispatcherTypes={
        DispatcherType.FORWARD,
        DispatcherType.INCLUDE,
        DispatcherType.REQUEST,
        DispatcherType.ERROR, DispatcherType.ASYNC
    }
)

```

如果不设置任何 dispatcherTypes，则默认为 REQUEST。 FORWARD 就是指通过 RequestDispatcher 的 forward() 而来的请求可以套用过滤器。 INCLUDE 就是指通过 RequestDispatcher 的 include() 而来的请求可以套用过滤器。 ERROR 是指由容器处理例外而转发过来的请求可以触发过滤器。 ASYNC 是指异步处理的请求可以触发过滤器(之后还会说明异步处理)。

若要在 web.xml 中设置，则可以使用 <dispatcher> 标签。例如：

```

...
<filter-mapping>
    <filter-name>SomeFilter</filter-name>
    <servlet-name>*.do</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
    <dispatcher>ERROR</dispatcher>
    <dispatcher>ASYNC</dispatcher>
</filter-mapping>
...

```

可以通过 <url-pattern> 或 <servlet-name> 来指定，哪些 URL 请求或哪些 Servlet 可应用过滤器。如果同时具备 <url-pattern> 与 <servlet-name>，则先比对 <url-pattern>，再比对 <servlet-name>。如果有某个 URL 或 Servlet 会应用多个过滤器，则根据 <filter-mapping> 在 web.xml 中出现的先后顺序，来决定过滤器的运行顺序。

5.3.3 请求封装器

以下举两个实际的例子，来说明请求封装器的实现与应用，分别是字符替换过滤器与编码设置过滤器。

1. 实现字符替换过滤器

假设有个留言版程序已经上线并正常运作中，但是现在发现，有些用户会在留言中输入一些 HTML 标签。基于安全性的考量，不希望用户输入的 HTML 标签直接出现在留言中而被浏览器当作 HTML 的一部分。例如，并不希望用户在留言中输入OpenHome.cc这样的信息，你不想信息在留言显示中直接变成超链接，让用户有机会在留言版中打广告，如图 5.7 所示。

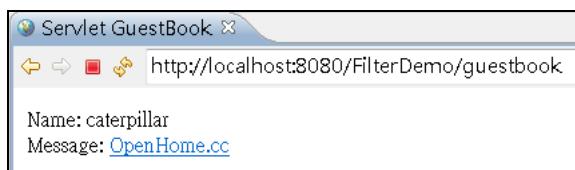


图 5.7 留言版被拿来打广告了

希望将一些 HTML 过滤掉，如将<、>角括号置换成 HTML 实体字符 < 与 >。如果不想直接修改留言版程序，则可以使用过滤器的方式，将用户请求参数中的角括号字符进行替换。但问题在于，虽然可以使用 `HttpServletRequest` 的 `getParameter()` 取得请求参数值，但就是没有一个像 `setParameter()` 的方法，可以将处理过后的请求参数重新设置给 `HttpServletRequest`。

对于容器产生的 `HttpServletRequest` 对象，无法直接修改某些信息，如请求参数值就是一个例子。你也许会想要亲自实现 `HttpServletRequest` 接口，让 `getParameter()` 返回过滤后的请求参数值，但这么做的话，`HttpServletRequest` 接口定义的方法都要实现，实现所有方法非常麻烦。

所幸，有个 `HttpServletRequestWrapper` 帮你实现了 `HttpServletRequest` 接口，只要继承 `HttpServletRequestWrapper` 类，并编写想要重新定义的方法即可。相对应于 `ServletRequest` 接口，也有个 `ServletRequestWrapper` 类可以使用。如图 5.8 所示。

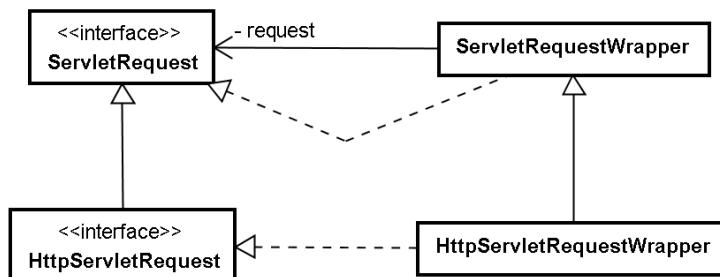


图 5.8 ServletRequestWrapper 与 HttpServletRequestWrapper



以下的范例通过继承 HttpServletRequestWrapper 实现了一个请求封装器，可以将请求参数中的 HTML 符号替换为 HTML 实体字符。

FilterDemo EscapeWrapper.java

```

package cc.openhome;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletRequestWrapper;
import org.apache.commons.lang.StringEscapeUtils;

public class EscapeWrapper extends HttpServletRequestWrapper {
    public EscapeWrapper(HttpServletRequest request) {
        super(request); ← ② 必须调用父类构造器，传入 HttpServletRequest 实例
    }

    @Override
    public String getParameter(String name) ← ③ 重新定义 getParameter() 方法
        String value = getRequest().getParameter(name);
        return StringEscapeUtils.escapeHtml(value); ← ④ 将取得的请求参数值进行字符替换
    }
}
  
```

EscapeWrapper 类继承了 HttpServletRequestWrapper①，并定义了一个接受 HttpServletRequest 的构造器，真正的 HttpServletRequest 将通过此构造器传入，必须使用 super() 调用 HttpServletRequestWrapper 接受 HttpServletRequest 的构造器②，之后如果要取得被封装的 HttpServletRequest，则可以调用 getRequest() 方法。

之后若有 Servlet 要取得请求参数值，都会调用 getParameter()，所以这里重新定义了 getParameter() 方法③，在此方法中，将真正从封装的 HttpServletRequest 对象上取得的请求参数值进行字符替换的动作④。

提示» 在这里直接使用了 Apache Commons Lang 程序库中，StringEscapeUtils 类提供的 escapeHtml() 方法来进行字符替换。可以在这里下载：

<http://commons.apache.org/lang/>

将下载的文件解开，将其中的 JAR 文件放至 Web 应用程序的 WEB-INF/lib 文件夹中即可。



可以使用这个请求封装器类搭配过滤器，以进行字符过滤的服务。例如：

FilterDemo EscapeFilter.java

```
package cc.openhome;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.http.HttpServletRequest;

@WebFilter("/")
public class EscapeFilter implements Filter {
    public void init(FilterConfig fConfig) throws ServletException {}

    public void doFilter(ServletRequest request,
                         ServletResponse response,
                         FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest requestWrapper =           ① 将原请求对象包裹至
            new EscapeWrapper((HttpServletRequest) request); EscapeWrapper 中
        chain.doFilter(requestWrapper, response); ② 将 EscapeWrapper 对
    }                                              象当作请求对象传入
                                                    doFilter()
    public void destroy() {}
}
```

在 `Filter` 的 `doFilter()` 中，创建 `EscapeWrapper` 实例，并将原请求对象传入构造器进行封装①。然后将 `EscapeWrapper` 实例传入 `FilterChain` 的 `doFilter()` 中作为请求对象②。之后的 `Filter` 或 `Servlet` 实例不需要也不会知道请求对象已经被封装，在必须取得请求参数时，同样调用 `getParameter()` 即可。

当将这个过滤器挂上去之后，如果有用户试图输入 HTML 标签，由于角括号都被替换为实体字符，所以出现的留言将会变成图 5.9 所示的画面。



图 5.9 挂上过滤器并输入 HTML 标签后的留言信息

实际上输入的 `OpenHome.cc` 会被替换为 “`OpenHome.cc`”，浏览器会在视觉上呈现 `OpenHome.cc`，但不会被当作 HTML 标签语法来解释。



2. 实现编码设置过滤器

在先前的范例中，如果要设置请求字符编码，都是在个别的 Servlet 中处理。可以在过滤器中进行字符编码设置，如果日后要改变编码，就不用每个 Servlet 逐一修改设置。

在 3.2.2 节中介绍字符编码设置时谈过，`HttpServletRequest` 的 `setCharacterEncoding()` 方法是针对请求 Body 内容，对于 GET 请求，必须取得请求参数的字节阵列后，重新指定编码建构字符串。这个需求与上一个范例类似，可搭配请求封装器来实现。

FilterDemo EncodingWrapper.java

```
package cc.openhome;

import java.io.UnsupportedEncodingException;
import javax.servlet.http.HttpServletRequest;           ① 继承
import javax.servlet.http.HttpServletRequestWrapper;    HttpServletRequestWrapper
public class EncodingWrapper extends HttpServletRequestWrapper { ←
    private String ENCODING;
    public EncodingWrapper(HttpServletRequest request, String ENCODING) {
        super(request); ← ② 必须调用父类构造器，传入 HttpServletRequest 实例
        this.ENCODING = ENCODING;
    }
    @Override
    public String getParameter(String name) { ← ③ 重新定义 getParameter() 方法
        String value = getRequest().getParameter(name);
        if(value != null) {
            try {
                byte[] b = value.getBytes("ISO-8859-1");
                value = new String(b, ENCODING); ← ④ 将取得的请求参数
            } catch (UnsupportedEncodingException e) {           值进行编码转换
                throw new RuntimeException(e);
            }
        }
        return value;
    }
}
```

`EncodingWrapper` 类的实现与上一个范例类似，其继承了 `HttpServletRequestWrapper` ①，并定义了一个接受 `HttpServletRequest` 的构造器，真正的 `HttpServletRequest` 将通过此构造器传入，必须使用 `super()` 调用 `HttpServletRequestWrapper` 接受 `HttpServletRequest` 的构造器 ②，之后如果要取得被封装的 `HttpServletRequest`，则可以调用 `getRequest()` 方法。

之后若有 Servlet 要取得请求参数值，都会调用 `getParameter()`，所以这里重新定义了 `getParameter()` 方法 ③。在此方法中，将真正从封装的 `HttpServletRequest` 对象上取得的请求参数值，进行编码替换的动作 ④。

至于编码过滤器的实现，如下所示：

FilterDemo EncodingFilter.java

```

package cc.openhome;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.http.HttpServletRequest;

@WebFilter(
    urlPatterns = { "/" },
    initParams = {
        @WebInitParam(name = "ENCODING", value = "UTF-8")
    }
)
public class EncodingFilter implements Filter {
    private String ENCODING;

    public void init(FilterConfig config) throws ServletException {
        ENCODING = config.getInitParameter("ENCODING"); ← ② 读取初始参数
    }

    public void doFilter(ServletRequest request,
                        ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        if("GET".equals(req.getMethod())) {
            req = new EncodingWrapper(req, ENCODING); ← ③ GET 请求时创建封装器
        }
        else {
            req.setCharacterEncoding(ENCODING);
        }
        chain.doFilter(req, response); ← ④ 调用 FilterChain 的 doFilter()
    }

    public void destroy() {}
}

```

① 设置过滤器初始参数



② 读取初始参数



③ GET 请求时创建封装器



④ 调用 FilterChain 的 doFilter()

请求参数的编码设置是通过过滤器初始参数来设置的①，并在过滤器初始化方法 `init()` 中读取②，过滤器仅在 GET 请求时创建 `EncodingWrapper` 实例③，其他方法则通过 `HttpServletRequest` 的 `setCharacterEncoding()` 设置编码，最后都调用 `FilterChain` 的 `doFilter()` 方法传入 `EncodingWrapper` 实例或原请求对象。



5.3.4 响应封装器

在 Servlet 中，是通过 `HttpServletResponse` 对象来对浏览器进行响应的。如果想要对响应的内容进行压缩处理，就要想办法让 `HttpServletResponse` 对象具有压缩处理的功能。先前介绍过请求封装器的实现，而在响应封装器的部分，可以继承 `HttpServletResponseWrapper` 类(父类 `ServletResponseWrapper`)来对 `HttpServletResponse` 对象进行封装，如图 5.10 所示。

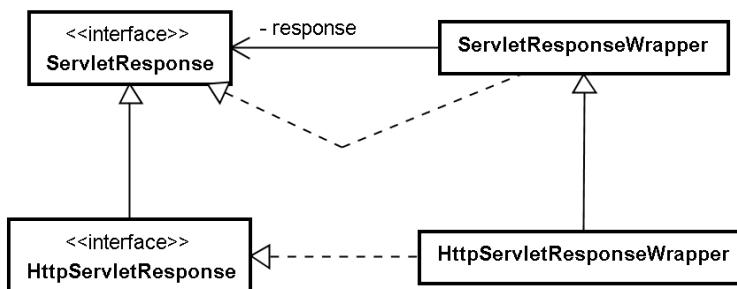


图 5.10 `ServletResponseWrapper` 与 `HttpServletResponseWrapper`

若要对浏览器进行输出响应，必须通过 `getWriter()` 取得 `PrintWriter`，或是通过 `getOutputStream()` 取得 `ServletOutputStream`。所以针对压缩输出的需求，主要就是继承 `HttpServletResponseWrapper` 之后，通过重新定义这两个方法来达成。

在这里压缩的功能将采 GZIP 格式，这是浏览器可以接受的压缩格式，可以使用 `GZIPOutputStream` 类来实现。由于 `getWriter()` 的 `PrintWriter` 在创建时，也是必须使用到 `ServletOutputStream`，所以在这里先扩展 `ServletOutputStream` 类，让它具有压缩的功能。

FilterDemo GZipServletOutputStream.java

```

package cc.openhome;

import java.io.IOException;
import java.util.zip.GZIPOutputStream;
import javax.servlet.ServletOutputStream;
public class GZipServletOutputStream extends ServletOutputStream {
    private GZIPOutputStream gzipOutputStream;
    public GZipServletOutputStream(
        ServletOutputStream servletOutputStream) throws IOException {
        this.gzipOutputStream =
            new GZIPOutputStream(servletOutputStream); ← ❷ 使用 GZIPOutputStream 来
    }                                                 增加压缩功能
    public void write(int b) throws IOException {
        gzipOutputStream.write(b); ← ❸ 输出时通过 GZIPOutputStream 的
    }                                                 write() 来压缩输出
  
```

❶ 继承 `ServletOutputStream`
来进行扩展

```

public GZIPOutputStream getGzipOutputStream() {
    return gzipOutputStream;
}
}

```

GzipServletOutputStream 继承 ServletOutputStream 类①，使用时必须传入 ServletOutputStream 类，由 GZIPOutputStream 来增加压缩输出串流的功能②。范例中重新定义 write() 方法，并通过 GZIPOutputStream 的 write() 方法来作串流输出③，GZIPOutputStream 的 write() 方法实现了压缩的功能。

在 HttpServletResponse 对象传入 Servlet 的 service() 方法前，必须封装它，使得调用 getOutputStream() 时，可以取得这里所实现的 GzipServletOutputStream 对象，而调用 getWriter() 时，也可以利用 GzipServletOutputStream 对象来构造 PrintWriter 对象。

FilterDemo CompressionWrapper.java

```

package cc.openhome;

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

public class CompressionWrapper extends HttpServletResponseWrapper {
    private GZipServletOutputStream gzServletOutputStream;
    private PrintWriter printWriter;

    public CompressionWrapper(HttpServletRequest resp) {
        super(resp);
    }

    @Override
    public ServletOutputStream getOutputStream() throws IOException {
        if(printWriter != null) { ←① 已调用过 getWriter(), 再调用
            throw new IllegalStateException(); ←① 已调用过
            getOutputStream()就抛出异常
        }
        if (gzServletOutputStream == null) {
            gzServletOutputStream = new GZipServletOutputStream(←② 创建有压缩功能的
               getResponse().getOutputStream()); ←② 创建有压缩功能的
                GZipServletOutputStream 对象
        }
        return gzServletOutputStream;
    }

    @Override
    public PrintWriter getWriter() throws IOException {
        if(gzServletOutputStream != null) { ←③ 已调用过
            throw new IllegalStateException(); ←③ 已调用过
            getOutputStream(), 再调用
            getWriter()就抛出异常
        }
        if (printWriter == null) {
            gzServletOutputStream = new GZipServletOutputStream(←④ 已调用过
               getResponse().getOutputStream());
            OutputStreamWriter osw = new OutputStreamWriter(

```

```

        gzServletOutputStream,
        getResponse().getCharacterEncoding());
printWriter = new PrintWriter(osw);
}
return printWriter;
}

@Override
public void setContentLength(int len) {}  
⑤不实现此方法内容，因为
public GZIPOutputStream getGZIPOutputStream() {  
真正的输出会被压缩
    if (this.gzServletOutputStream == null) {
        return null;
    }
    return this.gzServletOutputStream.getGzipOutputStream();
}
}

```

在上例中要注意，由于 Servlet 规格书中规定，在同一个请求期间，`getWriter()` 与 `getOutputStream()` 只能择一调用，否则必须抛出 `IllegalStateException`，因此建议在实现响应封装器时，也遵循这个规范。因此在重新定义 `getOutputStream()` 与 `getWriter()` 方法时，分别要检查是否已存在 `PrintWriter`**①** 与 `ServletOutputStream` 实例**②**。

在 `getOutputStream()` 中，会创建 `GzipServletOutputStream` 实例并返回。在 `getWriter()` 中调用 `getOutputStream()` 取得 `GzipServletOutputStream` 对象，作为构造 `PrintWriter` 实例时使用**④**，这样创建的 `PrintWriter` 对象也就具有压缩功能。由于真正的输出会被压缩，忽略原来的内容长度设置**⑤**。

接下来可以实现一个压缩过滤器，使用上面开发的 `CompressionWrapper` 来封装原 `HttpServletResponse`。

FilterDemo CompressionFilter.java

```

package cc.openhome;

import java.io.*;
import java.util.zip.GZIPOutputStream;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.WebFilter;

@WebFilter(filterName="CompressionFilter", urlPatterns={"/"})
public class CompressionFilter implements Filter {
    public void init(FilterConfig filterConfig) {}

    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;

```

```

String encodings = req.getHeader("accept-encoding");
if ((encodings != null) && (encodings.indexOf("gzip") > -1)) {  

    CompressionWrapper responseWrapper =
        new CompressionWrapper(res); ← ❷ 创建响应封装器
    responseWrapper.setHeader("content-encoding", "gzip"); ←  

    chain.doFilter(request, responseWrapper); ← ❸ 设置响应内容编
    GZIPOutputStream gzipOutputStream = ❹ 下一个过滤器
        responseWrapper.getGZIPOutputStream();
    if (gzipOutputStream != null) {
        gzipOutputStream.finish(); ← ❺ 调用 GZIPOutputStream 的
    }                                         finish()方法完成压缩输出
}
else {
    chain.doFilter(request, response); ← ❻ 不接受压缩, 直接进行
}                                         下一个过滤器
}  

public void destroy() {}  

}

```

浏览器是否接受 GZIP 压缩格式，可以通过检查 accept-encoding 请求标头中是否包括 gzip 字符串来判断①。如果可以接受 GZIP 压缩，创建 CompressionWrapper 封装原响应对象②，并设置 content-encoding 响应标头为 gzip，这样浏览器就会知道响应内容是 GZIP 压缩格式③。接着调用 FilterChain 的 doFilter() 时，传入的响应对象为 CompressionWrapper 对象④。当 FilterChain 的 doFilter() 结束时，必须调用 GZIPOutputStream 的 finish() 方法，这才会将 GZIP 后的资料从缓冲区中全部移出并进行响应⑤。

如果客户端不接受 GZIP 压缩格式，则直接调用 FilterChain 的 doFilter() ⑥，这样就可以让不接受 GZIP 压缩格式的客户端也可以收到原有的响应内容。

注意» 在这个过滤器真正完成压缩处理之前，Servlet/JSP 必须全部清除缓冲的响应资料，否则压缩的数据将是不完整的。

5.4 异步处理

Web 容器会为每个请求分配一个线程，默认情况下，响应完成前，该线程占用的资源都不会被释放。若有些请求需要长时间处理(例如长时间运算、等待某个资源)，就会长时间占用线程所需资源，若这类请求很多，许多线程资源都被长时间占用，会对系统的性能造成负担。

Servlet 3.0 新增了异步处理，可以先释放容器分配给请求的线程与相关资源，减轻系统负担，原先释放了容器所分配线程的请求，其响应将被延后，可以在处理完成(例如长时间运算完成、所需资源已获得)时再对客户端进行响应。

提示» 异步请求本身就是个进阶话题，常需搭配其他技术来完成，如 JavaScript，初学者可先略过此节内容。

5.4.1 AsyncContext 简介

为了支持异步处理，在 Servlet 3.0 中，在 `ServletRequest` 上提供了 `startAsync()` 方法：

```
AsyncContext startAsync() throws java.lang.IllegalStateException;
AsyncContext startAsync(ServletRequest servletRequest,
                      ServletResponse servletResponse)
throws java.lang.IllegalStateException
```

这两个方法都会返回 `AsyncContext` 接口的实现对象，前者会直接利用原有的请求与响应对象来创建 `AsyncContext`，后者可以传入自行创建的请求、响应封装对象。在调用了 `startAsync()` 方法取得 `AsyncContext` 对象之后，此次请求的响应会被延后，并释放容器分配的线程。

可以通过 `AsyncContext` 的 `getRequest()`、`getResponse()` 方法取得请求、响应对象，此次对客户端的响应将暂缓至调用 `AsyncContext` 的 `complete()` 或 `dispatch()` 方法为止，前者表示响应完成，后者表示将调派指定的 URL 进行响应。

若要能调用 `ServletRequest` 的 `startAsync()` 以取得 `AsyncContext`，必须告知容器此 Servlet 支持异步处理，如果使用 `@WebServlet` 来标注，则可以设置其 `asyncSupported` 为 `true`。例如：

```
@WebServlet(urlPatterns = "/some.do", asyncSupported = true)
public class AsyncServlet extends HttpServlet {
    ...
}
```

如果使用 `web.xml` 设置 Servlet，则可以在 `<servlet>` 中设置 `<async-supported>` 标签为 `true`：

```
...
<servlet>
    <servlet-name>AsyncServlet</servlet-name>
    <servlet-class>cc.openhome.AsyncServlet</servlet-class>
    <async-supported>true</async-supported>
</servlet>
...
```

如果 Servlet 将会进行异步处理，若其前端有过滤器，则过滤器亦需标示其支持异步处理，如果使用 `@WebFilter`，同样可以设置其 `asyncSupported` 为 `true`。例如：

```
@WebFilter(urlPatterns = "/some.do", asyncSupported = true)
public class AsyncFilter implements Filter{
    ...
}
```

如果使用 `web.xml` 设置过滤器，则可以设置 `<async-supported>` 标签为 `true`：

```
...
<filter>
```

```
<filter-name>AsyncFilter</filter-name>
<filter-class>cc.openhome.AsyncFilter</filter-class>
<async-supported>true</async-supported>
</filter>
...

```

下面示范一个异步处理的简单例子：

AsyncContextDemo AsyncServlet.java

```
package cc.openhome;

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

@WebServlet(name="AsyncServlet", urlPatterns={"/async.do"},
    asyncSupported = true) ← ① 告诉容器此 Servlet 支持异步处理
public class AsyncServlet extends HttpServlet {
    private ExecutorService executorService =
        Executors.newFixedThreadPool(10);
    @Override
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html; charset=UTF8");
        AsyncContext ctx = request.startAsync(); ← ② 开始异步处理，释放请求线程
        executorService.submit(new AsyncRequest(ctx)); ← ③ 创建 AsyncRequest，调度线程
    }
    @Override
    public void destroy() {
        executorService.shutdown(); ← ④ 关闭线程池
    }
}
```

首先告诉容器，这个 Servlet 支持异步处理①，对于每个请求，Servlet 会取得其 AsyncContext②，并释放容器所分配的线程，响应被延后。对于这些被延后响应的请求，创建一个实现 Runnable 接口的 AsyncRequest 对象，并将其调度一个线程池(Thread pool)③，线程池的线程数量是固定的，让这些必须长时间处理的请求，在这些有限数量的线程中完成，而不用每次请求都占用容器分配的线程。

AsyncRequest 是个实现 Runnable 的类，其模拟了长时间处理：

AsyncContextDemo AsyncRequest.java

```
package cc.openhome;

import java.io.PrintWriter;
import javax.servlet.AsyncContext;
public class AsyncRequest implements Runnable {
```



```

private AsyncContext ctx;
public AsyncRequest(AsyncContext ctx) { ←① 建构时接受 AsyncContext
    this.ctx = ctx;
}
@Override
public void run() {
    try {
        Thread.sleep(10000); ←② 模拟冗长请求
        PrintWriter out = ctx.getResponse().getWriter();
        out.println("久等了...XD"); ←③ 输出结果
        ctx.complete(); ←④ 对客户端完成响应
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
}

```

请求与响应对象都封装在 `AsyncContext` 中，所以 `AsyncRequest` 建构时必须接受 `AsyncContext` 实例。范例中以暂停线程的方式来模拟长时间处理②，并输出简单的字符串作为响应文字③，最后调用 `AsyncContext` 的 `complete()` 对客户端完成响应④。

5.4.2 模拟服务器推播

HTTP 是基于请求、响应模型，HTTP 服务器无法直接对客户端(浏览器)传送信息，因为没有请求就不会有响应。在这种请求、响应模型下，如果客户端想要获得服务器端应用程序的最新状态，必须以定期(或不定期)方式发送请求，查询服务器端的最新状态。

持续发送请求以查询服务器端最新状态，这种方式的问题在于耗用网络流量，如果多次请求过程后，服务器端应用程序状态并没有变化，那这多次的请求耗用的流量就是浪费的。

一个解决的方式是，服务器端将每次请求的响应延后，直到服务器端应用程序状态有变化时再进行响应。当然这样的话，客户端将会处于等待响应状态，如果是浏览器，可以搭配 Ajax 异步请求技术，而用户将不会因此而被迫停止网页的操作。然而服务器端延后请求的话，若是 Servlet/JSP 技术，等于该请求占用一个线程，若客户端很多，每个请求都占用线程，将会使得服务器端的性能负担很重。

Servlet 3.0 中提供的异步处理技术，可以解决每个请求占用线程的问题，若搭配浏览器端 Ajax 异步请求技术，就可达到类似服务器端主动通知浏览器的行为，也就是所谓的服务器端推播(Server push)。

提示» 后面这个范例中会用到 Ajax，但本书不讨论 Ajax。若对 JavaScript 与 Ajax 有兴趣研究，可以参考以下网址：

<http://caterpillar.onlyfun.net/Gossip/JavaScript>

以下是实际的例子，模拟应用程序不定期产生最新数据。这个部分由实现 `ServletContextListener` 的类负责，会在应用程序启动时进行：

ServerPushDemo WebInitListener.java

```
package cc.openhome;
import java.util.*;
import javax.servlet.*;
import javax.servlet.annotation.WebListener;
@WebListener()
public class WebInitListener implements ServletContextListener {
    private List<AsyncContext> asyncs = new ArrayList<AsyncContext>();
    @Override
    public void contextInitialized(ServletContextEvent sce) {
        sce.getServletContext().setAttribute("asyncs", asyncs);
        new Thread(new Runnable() {
            @Override
            public void run() {
                while (true) {
                    try {
                        Thread.sleep((int) (Math.random() * 10000));
                        double num = Math.random() * 10;
                        synchronized (asyncs) {
                            for(AsyncContext ctx : asyncs) {
                                ctx.getResponse().getWriter().println(num);
                                ctx.complete();
                            }
                            asyncs.clear();
                        }
                    } catch (Exception e) {
                        throw new RuntimeException(e);
                    }
                }
            }
        }).start();
    }
    @Override
    public void contextDestroyed(ServletContextEvent sce) {}
}
```

在这个 `ServletContextListener` 中，有个 `List` 会储存所有异步请求的 `AsyncContext` ①，并在不定时产生数字后 ②，逐一对客户端响应，并调用 `AsyncContext` 的 `complete()` 来完成请求 ③。

负责接受请求的 `Servlet`，一收到请求，就将之加入 `List` 中：



ServerPushDemo AsyncNumServlet.java

```

package cc.openhome;

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

@WebServlet(name="AsyncNumServlet", urlPatterns={"/asyncNum.do"},  

           asyncSupported = true)
public class AsyncNumServlet extends HttpServlet {  

    private List<AsyncContext> asyncs;  

    @Override  

    public void init() throws ServletException {  

        asyncs = (List<AsyncContext>) getServletContext()  

            .getAttribute("asyncs"); ←①取得储存 AsyncContext  

            的 List  

    }  

    @Override  

    protected void doGet(HttpServletRequest request,  

                         HttpServletResponse response)  

        throws ServletException, IOException {  

        AsyncContext ctx = request.startAsync(); ←②开始异步处理  

        synchronized(asyncs) {  

            asyncs.add(ctx); ←③加入维护 AsyncContext 的 List 中  

        }
    }
}

```

由于维护 `AsyncContext` 的 `List` 是储存为 `ServletContext` 属性，所以在这个 `Servlet` 中，必须从 `ServletContext` 中取出①，在每次请求来到时，调用 `HttpServletRequest` 的 `startAsync()` 进行异步处理②，并将取得 `AsyncContext` 加入至维护 `AsyncContext` 的 `List` 中③。

可以使用一个简单的 HTML，其中使用 Ajax 技术，发送异步请求至服务器端，这个请求会被延迟，直到服务器端完成响应后，更新网页上对应的资料，并再度发送异步请求：

ServerPushDemo async.html

```

<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"  

"http://www.w3.org/TR/html4/loose.dtd">  

<html>  

<head>  

<title>实时资料</title>  

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">  

<script type="text/javascript">  

    function asyncUpdate() {  

        var xhr;  

        if(window.XMLHttpRequest) {  

            xhr = new XMLHttpRequest();

```

```

        }
        else if(window.ActiveXObject) {
            xhr = new ActiveXObject('Microsoft.XMLHTTP');
        }

        xhr.onreadystatechange = function() {
            if(xhr.readyState === 4) {
                if(xhr.status === 200) {
                    document.getElementById('data')
                        .innerHTML = xhr.responseText;
                    asyncUpdate();
                }
            }
        };
        xhr.open('GET', 'asyncNum.do?timestamp='
            + new Date().getTime());
        xhr.send(null);
    }

    window.onload = asyncUpdate;
</script>
</head>
<body>
    实时资料: <span id="data">0</span>
</body>
</html>

```

可以试着使用多个浏览器视窗来请求这个页面，你会看到每个浏览器视窗的资料都是同步的。

5.4.3 更多 `AsyncContext` 细节

如果 `Servlet` 或过滤器的 `asyncSupported` 被标示为 `true`，则它们支持异步请求处理，在不支持异步处理的 `Servlet` 或过滤器中调用 `startAsync()`，会抛出 `IllegalStateException`。

当在支持异步处理的 `Servlet` 或过滤器中调用请求对象的 `startAsync()` 方法时，该次请求会离开容器所分配的线程，这意味着必须响应处理流程会返回，也就是若有过滤器，也会依序返回(也就是各自完成 `FilterChain` 的 `doFilter()` 方法)，但最终的响应被延迟。

可以调用 `AsyncContext` 的 `complete()` 方法完成响应，而后调用 `forward()` 方法，将响应转发给别的 `Servlet/JSP` 处理，`AsyncContext` 的 `forward()` 就如同 3.2.5 节中介绍的功能，将请求的响应权派送给别的页面来处理，给定的路径是相对于 `ServletContext` 的路径。不可以自行在同一个 `AsyncContext` 上同时调用 `complete()` 与 `forward()`，否则会抛出 `IllegalStateException`。



不可以在两个异步处理的 Servlet 间派送前，连续调用两次 `startAsync()`，否则会抛出 `IllegalStateException`。

将请求从支持异步处理的 Servlet(`asyncSupported`被标示为 `true`)派送至一个同步处理的 Servlet 是可行的(`asyncSupported`被标示为 `false`)，此时，容器会负责调用 `AsyncContext` 的 `complete()`。

如果从一个同步处理的 Servlet 派送至一个支持异步处理的 Servlet，在异步处理的 Servlet 中调用 `AsyncContext` 的 `startAsync()`，将会抛出 `IllegalStateException`。

如果对 `AsyncContext` 的起始、完成、超时或错误发生等事件有兴趣，可以实现 `AsyncListener`。其定义如下：

```
package javax.servlet;
import java.io.IOException;
import java.util.EventListener;
public interface AsyncListener extends EventListener {
    void onComplete(AsyncEvent event) throws IOException;
    void onTimeout(AsyncEvent event) throws IOException;
    void onError(AsyncEvent event) throws IOException;
    void onStartAsync(AsyncEvent event) throws IOException;
}
```

`AsyncContext` 有个 `addListener()` 方法，可以加入 `AsyncListener` 的实现对象，在对应事件发生时会调用 `AsyncListener` 实现对象的对应方法。

如果调用 `AsyncContext` 的 `dispatch()`，将请求调派给别的 Servlet，则可以通过请求对象的 `getAttribute()` 取得以下属性：

- `javax.servlet.async.request_uri`
- `javax.servlet.async.context_path`
- `javax.servlet.async.servlet_path`
- `javax.servlet.async.path_info`
- `javax.servlet.async.query_string`

这几个属性的值分别等于调用 `HttpServletRequest` 的 `getRequestURI()`、`getServletContext()`、`getServletPath()`、`getPathInfo()` 与 `getQueryString()` 所取得的结果。

5.5 综合练习

接下来要再进行综合练习，不过这次，不会在当前的微博应用程序中新增任何的功能，而是先停下来检查当前的应用程序，有哪些维护上的问题，在不改变当前应用程序的功能下，代码必须做出哪些调整，让每个代码职责上变得更为清晰，对于将来的维护更有帮助。

另外，本章谈到了一些 `Servlet`、`ServletContext` 初始参数设置，可用来设置一些共享的常数，也谈到了过滤器，介绍到如何过滤特殊字符以提升应用程序安全性，还谈过如何设置过滤器来改变请求参数的字符编码，这些都可以应用在当前的微博应用程序中。

5.5.1 创建 UserService

以微博应用程序作为综合练习，第 3 章先实现了基本的会员注册与登录功能，其中会员注册时，会通过检查用户目录是否存在，确定新注册的用户名是否使用，若可以使用，则会创建用户目录与相关文件。这些代码位于 `cc.openhome.controller.Register` 这个 `Servlet` 中：

```
...
    private boolean isInvalidUsername(String username) {
        for (String file : new File(USER).list()) {
            if (file.equals(username)) {
                return true;
            }
        }
        return false;
    }
    private void createUserData(String email, String username,
                               String password) throws IOException {
        File userhome = new File(USER + "/" + username);
        userhome.mkdir();
        BufferedWriter writer = new BufferedWriter(
            new FileWriter(userhome + "/profile"));
        writer.write(email + "\t" + password);
        writer.close();
    }
...
}
```

第 4 章使用 `HttpSession` 来进行用户登录会话管理，其中在登录检查时，通过检查用户目录是否存在，并且以文件 I/O 读取用户密码确认登录正确，来判断用户登录是否成功。这是实现在 `cc.openhome.controller.Login` 这个 `Servlet` 中：

```
...
    private boolean checkLogin(String username, String password)
        throws IOException {
        if(username != null && password != null) {
            for (String file : new File(USER).list()) {
                if (file.equals(username)) {
                    BufferedReader reader = new BufferedReader(
                        new FileReader(USER + "/" + file + "/profile"));
                    String passwd = reader.readLine().split("\t")[1];
                    if(passwd.equals(password)) {
                        return true;
                    }
                }
            }
        }
        return false;
    }
}
```

信息的新增，则是以文件 I/O 在用户目录中创建文件以储存信息。这是实现在 `cc.openhome.controller.Message` 这个 Servlet 中：

```
...
private void addMessage(String username, String blabla)
    throws IOException {
    String file = USERS + "/" + username + "/"
        + new Date().getTime() + ".txt";
    BufferedWriter writer = new BufferedWriter(
        new OutputStreamWriter(
            new FileOutputStream(file), "UTF-8"));
    writer.write(blabla);
    writer.close();
}
```

信息的删除，则是以文件 I/O 在用户目录中创建文件以储存信息。这是实现在 `cc.openhome.controller.Delete` 这个 Servlet 中：

```
...
File file = new File(USERS + "/" + username + "/" + message + ".txt");
if(file.exists()) {
    file.delete();
}
...
```

信息的显示，则是以文件 I/O 读取用户目录中的信息文件。这是实现在 `cc.openhome.view.Member` 这个 Servlet 中：

```
...
private class TxtFilenameFilter implements FilenameFilter {
    @Override
    public boolean accept(File dir, String name) {
        return name.endsWith(".txt");
    }
}

private TxtFilenameFilter filenameFilter = new TxtFilenameFilter();

private class DateComparator implements Comparator<Date> {
    @Override
    public int compare(Date d1, Date d2) {
        return -d1.compareTo(d2);
    }
}

private DateComparator comparator = new DateComparator();

private Map<Date, String> readMessage(String username)
    throws IOException {
    File border = new File(USERS + "/" + username);
    String[] txts = border.list( filenameFilter );
    Map<Date, String> messages =
        new TreeMap<Date, String>(comparator);
```

```

        for(String txt : txts) {
            BufferedReader reader = new BufferedReader(
                new InputStreamReader(
                    new FileInputStream(
                        USERS + "/" + username + "/" + txt), "UTF-8"));
            String text = null;
            StringBuilder builder = new StringBuilder();
            while((text = reader.readLine()) != null) {
                builder.append(text);
            }
            Date date = new Date(
                Long.parseLong(txt.substring(0, txt.indexOf(".txt"))));
            messages.put(date, builder.toString());
            reader.close();
        }
        return messages;
    }
    ...

```

到这里为止，你发现了什么？从会员注册开始、会员登录、信息新增、读取、显示等，相关的代码都与文件 I/O 读取有关，而这些代码散落在各个 Servlet 中，这造成了维护上的麻烦。何谓维护上的麻烦？想象一下，如果将来你的会员相关信息不再以文件存储，而要改为数据库存储，那要修改几个 Servlet 中的代码？如果你的会员信息处理相关代码继续散落在各个对象中，就造成了所谓职责分散的问题，任何将来要维护会员信息处理的相关代码就会越来越难以维护。

提示»» 接下来的练习重点在重构(Refactor)，主要是在不改变应用程序现有功能的情况下，调整应用程序架构与对象职责，练习过程可用复制现有代码、粘贴到新类的方式来完成，因此请直接使用上一章的综合练习成果来作为以下练习的开始。



为了解决以上所谈到的问题，这里将以上提到的相关代码集中在一个 cc.openhome.model.UserService 类中，若有需要会员注册开始、会员登录、信息新增、读取、显示等需求，都由 UserService 类提供。UserService 类如下所示：

Gossip UserService.java

```

package cc.openhome.model;

import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.FilenameFilter;
import java.io.IOException;
import java.io.InputStreamReader;

```

```

import java.io.OutputStreamWriter;
import java.util.Comparator;
import java.util.Date;
import java.util.Map;
import java.util.TreeMap;

public class UserService {
    private String USERS;

    public UserService(String USERS) {
        this.USERS = USERS; ← ① 设置用户目录
    }

    public boolean isInvalidUsername(String username) { ← ② 是否为不合
        for (String file : new File(USERS).list()) {             法用户名
            if (file.equals(username)) {
                return true;
            }
        }
        return false;
    }

    public void createUserData(String email, String username, ← ③ 创建用户目录与
                               String password) throws IOException {           基本资料
        File userhome = new File(USERS + "/" + username);
        userhome.mkdir();
        BufferedWriter writer = new BufferedWriter(
            new FileWriter(userhome + "/profile"));
        writer.write(email + "\t" + password);
        writer.close();
    }

    public boolean checkLogin(String username, String password) ← ④ 检查登录用户
                           throws IOException {                         名称与密码
        if (username != null && password != null) {
            for (String file : new File(USERS).list()) {
                if (file.equals(username)) {
                    BufferedReader reader = new BufferedReader(
                        new FileReader(USERS + "/" + file + "/profile"));
                    String passwd = reader.readLine().split("\t") [1];
                    if (passwd.equals(password)) {
                        return true;
                    }
                }
            }
        }
        return false;
    }

    private class TxtFilenameFilter implements FilenameFilter {
        @Override
        public boolean accept(File dir, String name) {
            return name.endsWith(".txt");
        }
    }
}

```

```

private TxtFilenameFilter filenameFilter = new TxtFilenameFilter();

private class DateComparator implements Comparator<Date> {
    @Override
    public int compare(Date d1, Date d2) {
        return -d1.compareTo(d2);
    }
}

private DateComparator comparator = new DateComparator();

public Map<Date, String> readMessage(String username) ←⑤ 读取用户的信息
throws IOException {
    File border = new File(USER + "/" + username);
    String[] txts = border.list(filenameFilter);

    Map<Date, String> messages =
        new TreeMap<Date, String>(comparator);
    for(String txt : txts) {
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(
                new FileInputStream(
                    USER + "/" + username + "/" + txt), "UTF-8"));
        String text = null;
        StringBuilder builder = new StringBuilder();
        while((text = reader.readLine()) != null) {
            builder.append(text);
        }
        Date date = new Date(
            Long.parseLong(txt.substring(0, txt.indexOf(".txt"))));
        messages.put(date, builder.toString());
        reader.close();
    }

    return messages;
}

public void addMessage(String username, String blabla) ←⑥ 新增信息
throws IOException {
    String file = USER + "/" + username + "/"
        + new Date().getTime() + ".txt";
    BufferedWriter writer = new BufferedWriter(
        new OutputStreamWriter(new FileOutputStream(file), "UTF-8"));
    writer.write(blabla);
    writer.close();
}

public void deleteMessage(String username, String message) { ←⑦ 删除信息
    File file = new File(USER + "/" + username + "/" + message + ".txt");
    if(file.exists()) {
        file.delete();
    }
}
}

```



由于用户的相关数据都是存储在与用户名称相同的目录中，所有用户目录位于指定的文件夹，这个文件夹可以在构造 `UserService` 时指定①。检查是否为合法用户名称②、创建用户目录与基本资料③、检查登录用户名与密码④、读取用户的信息⑤、新增信息⑥、删除信息⑦等功能，都改以 `UserService` 的公开方法来提供，将来若要改变这几个功能的文件存储来源，则只需要修改 `UserService` 的源代码，这就是集中相关职责于同一对象的好处。

提示»» 将分散各处的职责集中于单一或某几个对象，是改善可维护性的一种设计方式，但并不是集中职责就一定具有可维护性，有时在对象本身所负担的职责过于庞大时，也有可能将某些职责分割，再分散于不同的专职对象。最主要的是记得，设计是一个不断检查改进的过程。

稍后会利用这个 `UserService` 来修改当前的微博应用程序，先来看看过滤器要如何应用在这个应用程序中。

5.5.2 设置过滤器

在 5.3 节中谈到了过滤器，并实现了 `EscapeFilter`、`EncodingFilter` 两个过滤器，分别用来过滤特殊字符以及改变请求参数字符编码，这可以直接应用到微博应用程序中。

除此之外，在当前的微博应用程序中，有些功能必须在用户登录之后才可使用。为了确认用户是否登录，经常会在 `Servlet` 中看到类似以下的代码：

```
if(request.getSession().getAttribute("login") != null) {
    // 做一些登录用户可以做的事
}
```

这样的代码在数个 `Servlet` 中重复出现，重复出现的代码在设计上不是好事。这个检查用户是否登录的动作，其实可以在过滤器中进行。为此，可以设计以下的过滤器：

Gossip MemberFilter.java

```
package cc.openhome.web;

import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.annotation.WebFilter;
import javax.servlet.annotation.WebInitParam;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
```



```

@WebFilter(
    urlPatterns = { "/delete.do", "/logout.do",
                    "/message.do", "/member.view" },
    initParams = {
        @WebInitParam(name = "LOGIN_VIEW", value = "index.html")
    }
)
public class MemberFilter implements Filter {
    private String LOGIN_VIEW;
    public void init(FilterConfig config) throws ServletException {
        this.LOGIN_VIEW = config.getInitParameter("LOGIN_VIEW");  
① 设置登录页面
    }

    public void doFilter(ServletRequest request,
                         ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        if(req.getSession().getAttribute("login") != null) {
            chain.doFilter(request, response);  
② 只有在具备 login 属性时,
        }  
才调用 doFilter()
        else {
            HttpServletResponse resp = (HttpServletResponse) response;
            resp.sendRedirect(LOGIN_VIEW);  
③ 否则重新定向至登录页面
        }
    }

    public void destroy() {}
}

```

如果用户未登录，必须重定向到登录页面，登录页面可通过初始参数来设置①。由于登录成功的用户，`HttpSession`中会有`login`属性，所以只有在具备`login`属性时，才调用`doFilter()`，让请求可以往后由`Servlet`处理②，否则重新定向至登录页面，让用户可以进行窗体登录③。

5.5.3 重构微博

由于先前将一些用户信息 I/O 的职责集中在`UserService`对象，所以原先几个自行负责用户信息 I/O 的`Servlet`，将改使用`UserService`对象提供的公开方法服务，但在这之前必须先想想，各个`Servlet`如何取得`UserService`对象？何时产生`UserService`？



由于`UserService`是数个`Servlet`都会使用到的对象，也由于它本身不具备状态，可考虑将`UserService`作为整个应用程序都会使用到的一个服务对象。因此可将`UserService`对象存放在`ServletContext`属性中，而你可以在应用程序初始时，创建`UserService`对象，并存放在`ServletContext`中作为属性，这个需求可通过实现`ServletContextListener`来实现：

**Gossip GossipListener.java**

```
package cc.openhome.web;

import javax.servlet.ServletContext;
import javax.servlet.ServletContextEvent;
import javax.servlet.ServletContextListener;
import javax.servlet.annotation.WebListener;
import cc.openhome.model.UserService;

@WebListener
public class GossipListener implements ServletContextListener {
    public void contextInitialized(ServletContextEvent sce) {
        ServletContext context = sce.getServletContext();
        String USERS =
            sce.getServletContext().getInitParameter("USERS");
        context.setAttribute("userService", new UserService(USERS));
    }

    public void contextDestroyed(ServletContextEvent sce) {}
}
```



用户根目录可通过 `ServletContext` 初始参数设置，因此创建 `web.xml` 设置如下：

Gossip GossipListener.java

```
<?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"
    xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
<context-param>
    <param-name>USERS</param-name>
    <param-value>c:/workspace/Gossip/users</param-value>
</context-param>
</web-app>
```

接下来就是调整各 `Servlet` 的源代码，最主要的修改，就是删除原本在各 `Servlet` 中负责用户信息处理的 I/O 代码，改为采用从 `ServletContext` 取得 `UserService`，并调用所需的公开方法，并删除检查用户是否登录的代码，因为这个部分已经由先前设计的 `MemberFilter` 负责。另外，一些页面信息改为从 `ServletConfig` 取得。



为了节省篇幅，以下仅列出一些修改后有差异的部分代码，详细代码请参考本书配套光盘中的范例文件。首先是注册时的 `Servlet`：

Gossip Register.java

```
package cc.openhome.controller;
...
@WebServlet(
    urlPatterns={"/register.do"},
    initParams={
```

```
    @WebInitParam(name = "SUCCESS_VIEW", value = "success.view"),
    @WebInitParam(name = "ERROR_VIEW", value = "error.view")
}
}

public class Register extends HttpServlet {
    private String SUCCESS_VIEW;
    private String ERROR_VIEW;

    @Override
    public void init() throws ServletException {
        SUCCESS_VIEW = getServletConfig().getInitParameter("SUCCESS_VIEW");
        ERROR_VIEW = getServletConfig().getInitParameter("ERROR_VIEW");
    }

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws ServletException, IOException {
        ...

        UserService userService =
            (UserService) getServletContext().getAttribute("userService");
        ...

        if (userService.isInvalidUsername(username)) {
            errors.add("用户名为空或已存在");
        }
        ...

        String resultPage = ERROR_VIEW;
        if (!errors.isEmpty()) {
            request.setAttribute("errors", errors);
        } else {
            resultPage = SUCCESS_VIEW;
            userService.createUserData(email, username, password);
        }
        ...
    }
}
```

以下是登录用的 Servlet:

Gossip Login.java

```
package cc.openhome.controller;
...

@WebServlet(
    urlPatterns={"/login.do"},
    initParams={
        @WebInitParam(name = "SUCCESS_VIEW", value = "member.view"),
        @WebInitParam(name = "ERROR_VIEW", value = "index.html")
    }
)
public class Login extends HttpServlet {
    private String SUCCESS_VIEW;
    private String ERROR_VIEW;
    @Override
    public void init() throws ServletException {
        SUCCESS_VIEW = getServletConfig().getInitParameter("SUCCESS_VIEW");
        ERROR_VIEW = getServletConfig().getInitParameter("ERROR_VIEW");
    }
}
```

```

    ERROR_VIEW = getServletConfig().getInitParameter("ERROR_VIEW");
}

protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
    ...
    UserService userService =
        (UserService) getServletContext().getAttribute("userService");
    if(userService.checkLogin(username, password)) {
        request.getSession().setAttribute("login", username);
        page = SUCCESS_VIEW;
    }
    ...
}
}

```

进行注销的 Servlet 主要是改用 Servlet 初始参数设置登录窗体的 URL，去掉检查 HttpSession 中是否有 login 属性的代码。

Gossip Logout.java

```

package cc.openhome.controller;
...
@WebServlet(
    urlPatterns={"/logout.do"},
    initParams={
        @WebInitParam(name = "LOGIN_VIEW", value = "index.html")
    }
)
public class Logout extends HttpServlet {
    private String LOGIN_VIEW;

    @Override
    public void init() throws ServletException {
        LOGIN_VIEW = getServletConfig().getInitParameter("LOGIN_VIEW");
    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
                         throws ServletException, IOException {
        request.getSession().invalidate();
        response.sendRedirect(LOGIN_VIEW);
    }
}

```

新增信息的 Servlet 不需要调用请求对象的 `setCharacterEncoding()` 来设置字符编码，因为这改由过滤器负责了。修改后的重点部分如下：

Gossip Message.java

```

package cc.openhome.controller;
...
@WebServlet(

```

```
urlPatterns={"/message.do"},  
initParams={  
    @WebInitParam(name = "SUCCESS_VIEW", value = "member.view"),  
    @WebInitParam(name = "ERROR_VIEW", value = "member.view")  
}  
}  
  
public class Message extends HttpServlet {  
    private String SUCCESS_VIEW;  
    private String ERROR_VIEW;  
  
    @Override  
    public void init() throws ServletException {  
        SUCCESS_VIEW = getServletConfig().getInitParameter("SUCCESS_VIEW");  
        ERROR_VIEW = getServletConfig().getInitParameter("ERROR_VIEW");  
    }  
  
    protected void doPost(HttpServletRequest request,  
                         HttpServletResponse response)  
        throws ServletException, IOException {  
        ...  
        if(blabla != null && blabla.length() != 0) {  
            if(blabla.length() < 140) {  
                UserService userService = (UserService)  
                    getServletContext().getAttribute("userService");  
                userService.addMessage(username, blabla);  
                response.sendRedirect(SUCCESS_VIEW);  
            }  
        }  
    }  
}
```

删除信息的 Servlet 如下所示：

Gossip Delete.java

```
package cc.openhome.controller;  
...  
@WebServlet(  
    urlPatterns={"/delete.do"},  
    initParams={  
        @WebInitParam(name = "SUCCESS_VIEW", value = "member.view")  
    }  
)  
public class Delete extends HttpServlet {  
    private String SUCCESS_VIEW;  
    @Override  
    public void init() throws ServletException {  
        SUCCESS_VIEW = getServletConfig().getInitParameter("SUCCESS_VIEW");  
    }  
  
    protected void doGet(HttpServletRequest request,  
                         HttpServletResponse response)  
        throws ServletException, IOException {  
        ...  
        UserService userService =  
            (UserService) getServletContext().getAttribute("userService");  
    }  
}
```

```

        userService.deleteMessage(username, message);
        response.sendRedirect(SUCCESS_VIEW);
    }
}

```

会员网页的 Servlet 如下所示：

Gossip Member.java

```

package cc.openhome.view;
...
@WebServlet("/member.view")
public class Member extends HttpServlet {
    protected void processRequest(HttpServletRequest request,
                                  HttpServletResponse response)
        throws ServletException, IOException {
    ...
    UserService userService =
        (UserService) getServletContext().getAttribute("userService");
    Map<Date, String> messages = userService.readMessage(username);
    ...
}

```

原本未修改前，只有控制器与视图，也因此一些非控制器负责的代码散落在各控制器中，在相关职责集中至 `UserService` 后，`UserService` 就担任模型的角色，而各 `Servlet` 专心负责取得请求参数、验证请求参数、转发请求等职责，担任视图的 `Member`，也从 `UserService` 中取得信息资料并加以显示。

在经过这些修改后，其实已经可以略为看出 `MVC/Model 2` 的雏形与流程。由于目前视图的部分，依旧由 `Servlet` 来负责，所以还无法完全看出 `MVC/Model 2` 的好处，在之后学到 `JSP`、`JSTL` 之后，会用 `JSP` 与 `JSTL` 等来改写目前负责画面显示的部分，那时就可以更加深刻地看到 `MVC/Model 2` 的样子与好处。

5.6 重点复习

`Servlet` 接口上，与生命周期及请求服务相关的三个方法是 `init()`、`service()` 与 `destroy()` 方法。当 `Web` 容器加载 `Servlet` 类并实例化之后，会生成 `ServletConfig` 对象并调用 `init()` 方法，将 `ServletConfig` 对象当作参数传入。`ServletConfig` 相当于 `Servlet` 在 `web.xml` 中的设置代表对象，可以利用它来取得 `Servlet` 初始参数。

`GenericServlet` 同时实现了 `Servlet` 及 `ServletConfig`。主要的目的，就是将初始 `Servlet` 调用 `init()` 方法所传入的 `ServletConfig` 封装起来。

当希望编写代码在 `Servlet` 初始化时运行，要重新定义无参数的 `init()` 方法，而不是有 `ServletConfig` 参数的 `init()` 方法或构造器。

`ServletConfig` 上还定义了 `getServletContext()` 方法，这可以取得 `ServletContext` 实例，这个对象代表了整个 Web 应用程序，可以从这个对象取得 `ServletContext` 初始参数，或是设置、取得、移除 `ServletContext` 属性。

每个 Web 应用程序都会有一个相对应的 `ServletContext`，针对应用程序初始化时所需用到的一些参数资料，可以在 `web.xml` 中设置应用程序初始参数，设置时使用 `<context-param>` 标签来定义。每一对初始参数要使用一个 `<context-param>` 来定义。

在整个 Web 应用程序生命周期，`Servlet` 所需共享的资料可以设置为 `ServletContext` 属性。由于 `ServletContext` 在 Web 应用程序存活期间都会一直存在，所以设置为 `ServletContext` 属性的资料，除非主动移除，否则也是一直存活于 Web 应用程序中。

监听器顾名思义，就是可监听某些事件的发生，然后进行一些想做的事情。在 `Servlet/JSP` 中，如果想要在 `ServletRequest`、`HttpSession` 与 `ServletContext` 对象创建、销毁时收到通知，则可以实现以下相对应的监听器：

- `ServletRequestListener`
- `HttpSessionListener`
- `ServletContextListener`

`Servlet/JSP` 中可以设置属性的对象有 `ServletRequest`、`HttpSession` 与 `ServletContext`。如果想在这些对象被设置、移除、替换属性时收到通知，则可以实现以下相对应的监听器：

- `ServletRequestAttributeListener`
- `HttpSessionAttributeListener`
- `ServletContextAttributeListener`

`Servlet/JSP` 中如果某个对象即将加入 `HttpSession` 中成为属性，而你想要该对象在加入 `HttpSession`、从 `HttpSession` 移除、`HttpSession` 对象在 JVM 间迁移时收到通知，则可以在将成为属性的对象上，实现以下相对应的监听器：

- `HttpSessionBindingListener`
- `HttpSessionActivationListener`

在 `Servlet/JSP` 中要实现过滤器，必须实现 `Filter` 接口，并在 `web.xml` 中定义过滤器，让容器知道加载哪个过滤器类。`Filter` 接口有三个要实现的方法，`init()`、`doFilter()` 与 `destroy()`，三个方法的作用与 `Servlet` 接口的 `init()`、`service()` 与 `destroy()` 类似。

`Filter` 接口的 `init()` 方法的参数是 `FilterConfig`，`FilterConfig` 为过滤器定义的代表对象，可以通过 `FilterConfig` 的 `getInitParameter()` 方法来取得初始参数。



当请求来到过滤器时，会调用 `Filter` 接口的 `doFilter()` 方法，`doFilter()` 上除了 `ServletRequest` 与 `ServletResponse` 之外，还有一个 `FilterChain` 参数。如果调用了 `FilterChain` 的 `doFilter()` 方法，就会运行下一个过滤器，如果没有下一个过滤器了，就调用请求目标 `Servlet` 的 `service()` 方法。如果因为某个条件(例如用户没有通过验证)而不调用 `FilterChain` 的 `doFilter()`，则请求就不会继续至目标 `Servlet`，这时就是所谓的拦截请求。

在实现 `Filter` 接口时，不用理会这个 `Filter` 前后是否有其他 `Filter`，完全作为一个独立的元件进行设计。

对于容器产生的 `HttpServletRequest` 对象，无法直接修改某些信息，如请求参数值。可以继承 `HttpServletRequestWrapper` 类(父类 `ServletRequestWrapper`)，并编写想要重新定义的方法。对于 `HttpServletResponse` 对象，则可以继承 `HttpServletResponseWrapper` 类(父类 `ServletResponseWrapper`)来对 `HttpServletResponse` 对象进行封装。

5.7 课后练习

5.7.1 选择题

1. 如果是整个应用程序会共享的数据，则适合存放在()对象中成为属性。

- A. `ServletConfig`
- B. `ServletContext`
- C. `ServletRequest`
- D. `Session`

2. 如果要取得 `ServletContext` 初始参数，则可以执行()方法。

- A. `getContextParameter()`
- B. `getParameter()`
- C. `getInitParameter()`
- D. `getAttribute()`

3. 假设有段程序代码如下，其中 `PARAM` 为设定于 `web.xml` 中的初始参数：

```
public class SomeServlet extends HttpServlet {
    private String param;
    public SomeServlet() {
        param = getInitParameter("PARAM");
    }
    ...
}
```

以下正确的是()。

- A. `param` 被设定为 `web.xml` 中的初始参数值
- B. 无法通过编译

- C. 应该改用 `getServletParameter()` 方法
D. 发生 `NullPointerException`
4. 继承 `HttpServlet` 之后，若要进行 `Servlet` 初始化，重新定义()方法才是正确的做法。
- A. `public void init(ServletConfig config) throws ServletException`
B. `public void init() throws ServletException`
C. `public String getInitParameter(String name)`
D. `public Enumeration getInitParameterNames()`
5. 提供 `getAttribute()` 方法的对象有()。
- A. `ServletRequest` B. `HttpSession`
C. `ServletConfig` D. `ServletContext`
6. 关于过滤器的描述，正确的是()。
- A. `Filter` 接口定义了 `init()`、`service()` 与 `destroy()` 方法
B. 会传入 `ServletRequest` 与 `ServletResponse` 至 `Filter`
C. 要执行下一个过滤器，必须执行 `FilterChain` 的 `next()` 方法
D. 如果要取得初始参数，要使用 `FilterConfig` 对象
7. 关于 `FilterChain` 的描述，正确的是()。
- A. 如果不调用 `FilterChain` 的 `doFilter()` 方法，则请求略过接下来的过滤器而直接交给 `Servlet`
B. 如果有下一个过滤器，调用 `FilterChain` 的 `doFilter()` 方法，会将请求交给下一个过滤器
C. 如果没有下一个过滤器，调用 `FilterChain` 的 `doFilter()` 方法，会将请求交给 `Servlet`
D. 如果没有下一个过滤器，调用 `FilterChain` 的 `doFilter()` 方法没有作用
8. 关于 `FilterConfig` 的描述，错误的是()。
- A. 会在 `Filter` 接口的 `init()` 方法调用时传入
B. 为 `@WebServlet`、`web.xml` 中 `<filter>` 设定的代表对象
C. 可读取 `<servlet>` 标签中 `<init-param>` 设定的初始参数
D. 可使用 `getInitParameter()` 方法读取初始参数
9. 以下的程序代码将实现请求封装器：

```
public class MyRequestWrapper _____ {  
    public MyRequestWrapper(HttpServletRequest request) {  
        super(request);  
    }  
}
```

...
}

请问空白处应该填上代码段()。

- A. implements ServletRequest
- B. extends ServletRequestWrapper
- C. implements HttpServletRequest
- D. extends HttpServletRequestWrapper

10. 关于请求封装器, 以下描述正确的是()。

- A. 可以实现 ServletRequest 接口
- B. 可以继承 ServletRequestWrapper 类
- C. 一定要继承 ServletRequestWrapper 类
- D. HttpServletRequestWrapper 是 ServletRequestWrapper 的子类

11. 关于 HttpServletRequestWrapper 与 HttpServletResponseWrapper 的描述, 有误的是()。

- A. 分别实现了 HttpServletRequest 接口与 HttpServletResponse 接口
- B. 分别继承了 ServletRequestWrapper 与 ServletResponseWrapper 类
- C. 实现时, 至少要重新定义一个父类中的方法
- D. 实现时必须在构造器中调用父类构造器

12. 在开发过滤器时, 以下说法正确的是()。

- A. 必须考虑前后过滤器之间的关系
- B. 挂上过滤器后不改变应用程序原有的功能
- C. 设计 Servlet 时必须考虑到未来加装过滤器的需求
- D. 每个过滤器要设计为独立互不影响的组件

13. 关于 Filter 接口上的 doFilter() 方法的说明, 有误的是()。

- A. 会传入两个参数 ServletRequest、ServletResponse
- B. 会传入三个参数 ServletRequest、ServletResponse、FilterChain
- C. 前一个过滤器调用 FilterChain 的 doFilter() 后, 会执行目前过滤器的 doFilter() 方法
- D. 前一个过滤器的 doFilter() 执行过后, 会执行目前过滤器的 doFilter() 方法

14. 你有一段代码:

```
HttpSession session = request.getSession();
User user = new User();
session.setAttribute("user", user);
```

以下的做法()，可以在不修改代码的情况下，实现统计在线人数。

- A. 实现 HttpSessionBindingListener
- B. 实现 HttpSessionListener
- C. 实现 HttpSessionActivationListener
- D. 以上皆非

15. 以下监听器中，不需要使用 @WebListener 或在 web.xml 中设定的是()。

- A. HttpSessionListener
- B. HttpSessionBindingListener
- C. ServletContextListener
- D. ServletAttributeListener

实训题

1. 扩充 5.2.2 节中的范例，不仅统计在线人数，还可以在页面上显示目前登录用户的名称、浏览器信息、最后活动时间，如图 5.11 所示。

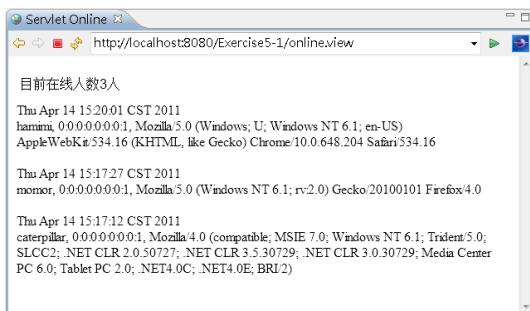


图 5.11 在线用户信息

2. 在 5.2.2 节中，使用 HttpSessionBindingListener 在用户登录后进行数据库查询功能，请改用 HttpSessionAttributeListener 来实现这个功能。

3. 你的应用程序不允许用户输入 HTML 标签，但可以允许用户输入一些代码做些简单的样式。例如：

- [b]粗体[/b]
- [i]斜体[/i]
- [big]放大字体[/big]
- [small]缩小字体[/small]

HTML 的过滤功能，可以直接使用本章所开发的字符过滤器，并基于该字符过滤器进行扩充。

4. 在 5.3.3 节开发的字符替换过滤器与编码设置过滤器，继承 HttpServletRequestWrapper 后都仅重新定义了 getParameter() 方法。事实上，为了完整性，getParameterValues()、getParameterMap() 等方法也要重新定义，请加强 5.3.3 节中的字符替换过滤器与替换设置过滤器，针对 getParameterValues()、getParameterMap() 重新定义。

整合数据库

学习目标：

- 了解 JDBC 架构
- 使用基本的 JDBC
- 通过 JNDI 取得 DataSource
- 在 Web 应用程序中整合数据库

9.1 JDBC 入门

JDBC 是用于执行 SQL 的解决方案，开发人员使用 JDBC 的标准接口，数据库厂商则对接口进行实现，开发人员无须了解底层数据库驱动程序的差异性，直接使用即可。

在这个章节中，会说明一些 JDBC 基本 API 的使用与概念，以便对 Java 如何访问数据库有所认识，并了解如何在 Servlet/JSP 中使用整合 JDBC。

9.1.1 JDBC 简介

在正式介绍 JDBC 之前，要先来认识应用程序如何与数据库进行沟通。数据库本身是个独立运行的应用程序，编写的应用程序是利用网络通信协议与数据库进行命令交换，以进行数据的增删查找。如图 9.1 所示。

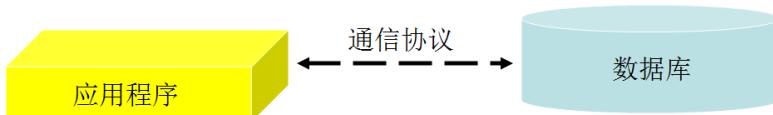


图 9.1 应用程序与数据库利用通信协议沟通

通常应用程序会利用一组专门与数据库进行通信协议的程序库，以简化与数据库沟通时的程序编写，如图 9.2 所示。

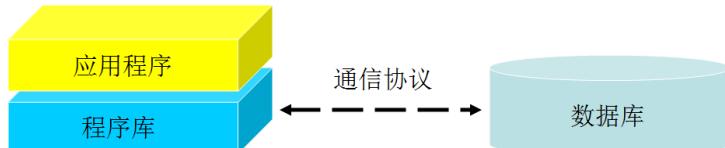


图 9.2 应用程序调用程序库以简化程序编写

问题的重点在于，你的应用程序如何调用这组程序库？不同的数据库通常会有不同的通信协议，用以连接不同数据库的程序库在 API 上也会有所不同，如果应用程序直接使用这些程序库，例如：

```
XySqlConnection conn = new XySqlConnection("localhost", "root", "1234");
conn.selectDB("gossip");
XySqlQuery query = conn.query("SELECT * FROM T_USER");
```

假设这段代码中的 API 是某 Xy 数据库厂商程序库所提供，应用程序中要使用到数据库连接时，都会直接调用这些 API，若哪天应用程序打算改用 Ab 厂商数据库及其提供的数据库连接 API，那就得修改相关的代码。

另一个考量是，若 Xy 数据库厂商的程序库底层实际上使用了与操作系统相关的功能，若只打算换个操作系统，则就还得先权衡一下，是否有提供该平台的数据库的程序库。

更换数据库的需求并不是没有，应用程序跨平台也是经常的需求。JDBC 基本上就是用来解决这些问题，JDBC 全名 Java DataBase Connectivity，是 Java 数据库连接的标准规范。具体而言，它定义一组标准类与接口，应用程序需要连接数据库时就调用这组标准 API，而标准 API 中的接口会由数据库厂商实现，通常称为 JDBC 驱动程序(Driver)，如图 9.3 所示。

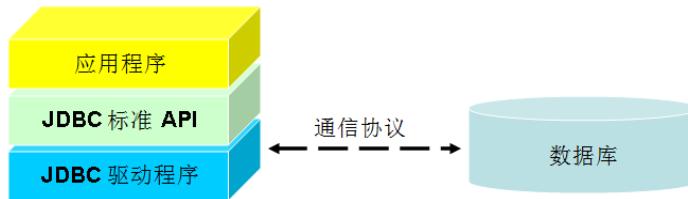


图 9.3 应用程序调用 JDBC 标准 API

JDBC 标准主要分为两个部分：JDBC 应用程序开发者接口(Application Developer Interface)以及 JDBC 驱动程序开发者接口(Driver Developer Interface)。如果应用程序需要连接数据库，就是调用 JDBC 应用程序开发者接口(见图 9.4)，相关 API 主要在 `java.sql` 与 `javax.sql` 两个包中，也是本章节说明的重点。JDBC 驱动程序开发者接口则是数据库厂商要实现驱动程序时的规范，一般开发者并不用了解，本章节不予说明。

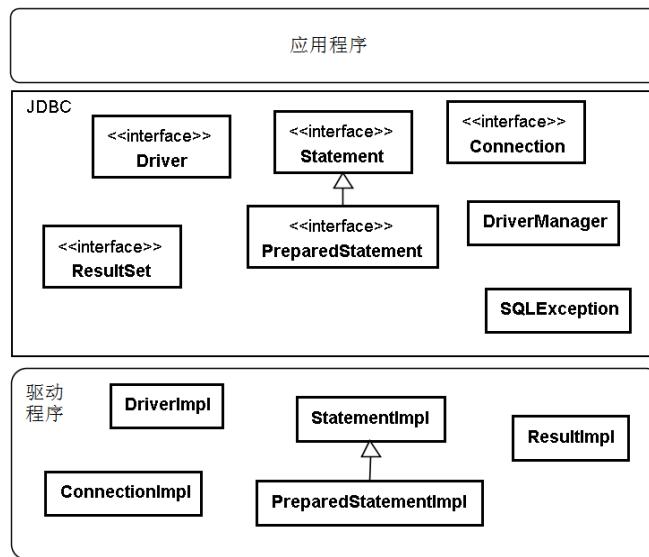


图 9.4 JDBC 应用程序开发者接口

举个例子来说，你的应用程序会使用 JDBC 连接数据库：

```
Connection conn = DriverManager.getConnection(...);
Statement st = conn.createStatement();
ResultSet rs = st.executeQuery("SELECT * FROM T_USER");
```

其中粗体字的部分就是标准类(如 `DriverManager`)与接口(如 `Connection`、`Statement`、`ResultSet`)等标准 API，假设这段代码是连接 MySQL 数据库，你会需要在 Classpath 中设置 JDBC 驱动程序。具体来说，就是在 Classpath 中设置一个 JAR 文件，此时应用程序、JDBC 与数据库的关系如图 9.5 所示。



图 9.5 应用程序、JDBC 与数据库的关系

如果将来要换为 Oracle 数据库，那么只要置换 Oracle 驱动程序。具体来说，就是在 Classpath 改设为 Oracle 驱动程序的 JAR 文件，而应用程序本身不用修改，如图 9.6 所示。

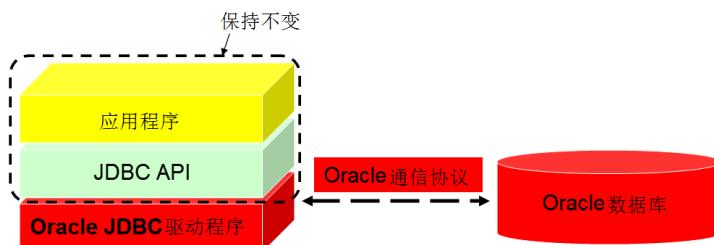


图 9.6 置换驱动程序不用修改应用程序

如果开发应用程序需要操作数据库，是通过 JDBC 所提供的接口来设计程序，则理论上在必须更换数据库时，应用程序无须进行修改，只需要更换数据库驱动程序实现，即可对另一个数据库进行操作。

JDBC 希望达到的目的，是希望让 Java 程序员在编写数据库操作程序的时候，可以有个统一的接口，无须依赖于特定的数据库 API，希望达到“写一个 Java 程序，操作所有的数据库”的目的。

提示»» 实际上在编写 Java 程序时，会因为使用了数据库或驱动程序特定的功能，而在转移数据库时仍得对程序进行修改。例如，使用了特定于某数据库的 SQL 语法、数据类型或内建函数调用等。

厂商在实现 JDBC 驱动程序时，按方式可将驱动程序分为四种类型：

- Type 1: JDBC-ODBC Bridge Driver

ODBC(Open DataBase Connectivity)是由 Microsoft 主导的数据库连接标准(基本上 JDBC 是参考 ODBC 所制订出来), 所以 ODBC 在 Microsoft 的系统上也最为成熟。例如, Microsoft Access 数据库访问就是使用 ODBC。Type 1 驱动程序会将 JDBC 的调用转换为对 ODBC 驱动程序的调用, 由 ODBC 驱动程序来操作数据库, 如图 9.7 所示。

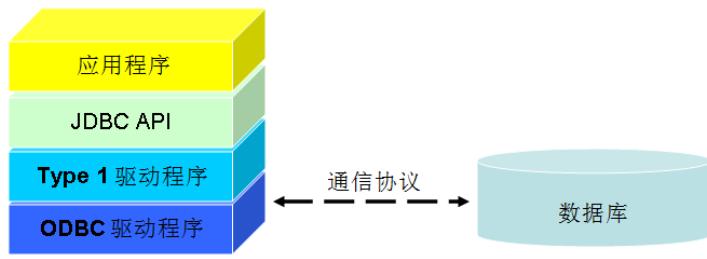


图 9.7 JDBC-ODBC Bridge Driver

由于利用现成的 ODBC 架构, 只需要将 JDBC 调用转换为 ODBC 调用, 所以要实现这种驱动程序非常简单。在 Oracle/Sun JDK 中就附带有驱动程序, 包名称以 `sun.jdbc.odbc` 开头。

不过由于 JDBC 与 ODBC 并非一对一的对应, 所以部分调用无法直接转换。因此有些功能是受限的, 而多层次调用转换的结果, 访问速度也会受到限制, ODBC 本身需在平台上先设置好, 弹性不足, ODBC 驱动程序本身也有跨平台的限制。

■ Type 2: Native API Driver

这个类型的驱动程序会以原生(Native)方式, 调用数据库提供的原生程序库(通常由 C/C++ 实现), JDBC 的方法调用都会转换为原生程序库中的相关 API 调用, 如图 9.8 所示。由于使用了原生程序库, 所以驱动程序本身与平台相依, 没有达到 JDBC 驱动程序的目标之一: 跨平台。不过由于是直接调用数据库原生 API, 因此在速度上, 有机会成为四种类型中最快的驱动程序。

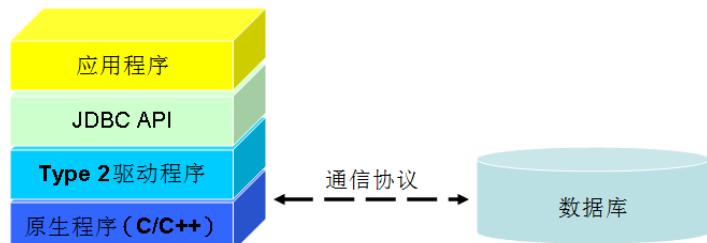


图 9.8 Native API Driver

Type 2 驱动程序有机会成为速度最快的驱动程序，速度的优势是在于获得数据库响应数据后，构造相关 JDBC API 实现对象，然而驱动程序本身无法跨平台，使用前必须先在各平台进行驱动程序的安装设置(如安装数据库专属的原生程序库)。

■ Type 3: JDBC-Net Driver

这个类型的 JDBC 驱动程序会将 JDBC 的方法调用，转换为特定的网络协议(Protocol)调用，目的是与远程与数据库特定的中介服务器或组件进行协议操作，而中介服务器或组件再真正与数据库进行操作。如图 9.9 所示。

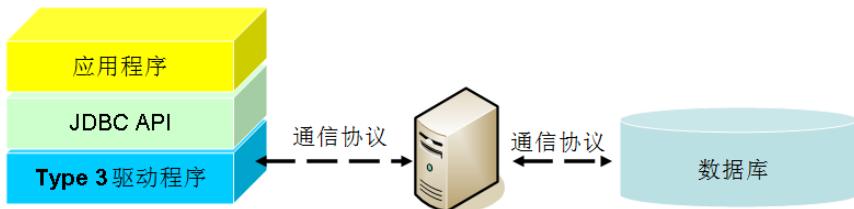


图 9.9 JDBC-Net Driver

■ 由于实际与中介服务器或组件进行沟通时，是利用网络协议的方式，所以客户端这里安装的驱动程序可以使用纯粹的 Java 技术来实现(基本上就是将 JDBC 调用对应至网络协议而已)，因此这个类型的驱动程序可以跨平台。使用这个类型驱动程序的弹性高，例如可以设计一个中介组件，JDBC 驱动程序与中介组件间的协议是固定的，如果需要更换数据库系统，则只需要更换中介组件，而客户端不受影响，驱动程序也无须更换。但由于通过中介服务器转换，速度较慢。获得架构上的弹性是使用这个类型驱动程序的目的。

■ Type 4: Native Protocol Driver

这个类型的驱动程序实现通常由数据库厂商直接提供，驱动程序实现会将 JDBC 的调用转换为与数据库特定的网络协议，以与数据库进行沟通操作。如图 9.10 所示。

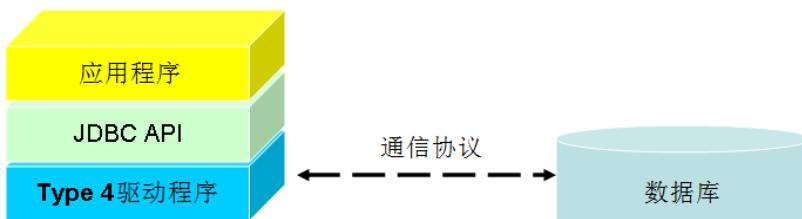


图 9.10 Native Protocol Driver

由于这个类型驱动程序主要的作用，是将 JDBC 的调用转换为特定的网络协议，所以驱动程序可以使用纯粹 Java 技术来实现，因此这个类型的驱动程序可以跨平台，在性能上也能有不错的表现。在不需要如 Type 3 获得架构上的弹性时，通常会使用该类型驱动程序，它算是最常见的驱动程序类型。

在接下来的内容中，将使用 MySQL 数据库系统进行操作，并使用 Type 4 驱动程序。可以在以下网址取得 MySQL 的 JDBC 驱动程序：

<http://www.mysql.com/products/connector/j/index.html>

提示»» 数据库系统的使用与操作是个很大的话题，本书中并不针对这方面详加探讨，请寻找相关的数据库系统相关书籍自行学习。为了能顺利练习这个章节的范例，附录中包括了一个 MySQL 数据库系统的简介，足够让你了解这一个章节所用到的一些数据库操作命令。

9.1.2 连接数据库

为了要连接数据库系统，必须要有厂商实现的 JDBC 驱动程序，可以将驱动程序 JAR 文件放在 Web 应用程序的/WEB-INF/lib 文件夹中。基本数据库操作相关的 JDBC 接口或类位于 `java.sql` 包中。要取得数据库连接，必须有几个操作：

- 注册 `Driver` 实现对象
- 取得 `Connection` 实现对象
- 关闭 `Connection` 实现对象

注册 `Driver` 实现对象

实现 `Driver` 接口的对象是 JDBC 进行数据库访问的起点，以 MySQL 实现的驱动程序为例，`com.mysql.jdbc.Driver` 类实现了 `java.sql.Driver` 接口，管理 `Driver` 实现对象的类是 `java.sql.DriverManager`，基本上，必须调用其静态方法 `registerDriver()` 进行注册：

```
DriverManager.registerDriver(new com.mysql.jdbc.Driver());
```

不过实际上很少自行编写代码进行这个操作，只要想办法加载 `Driver` 接口的实现类.class 文件，就会完成注册。例如，可以通过 `java.lang.Class` 类的 `forName()`，动态加载驱动程序类：

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
}  
catch(ClassNotFoundException e) {  
    throw new RuntimeException("找不到指定的类");  
}
```

}

如果查看 MySQL 的 `Driver` 类实现源代码：

```
package com.mysql.jdbc;
import java.sql.SQLException;
public class Driver extends NonRegisteringDriver
    implements java.sql.Driver {
    static {
        try {
            java.sql.DriverManager.registerDriver(new Driver());
        } catch (SQLException E) {
            throw new RuntimeException("Can't register driver!");
        }
    }

    public Driver() throws SQLException {}
}
```

可以发现，在 `static` 区块中进行了注册 `Driver` 实例的操作，而 `static` 区块会在加载.class 文件时执行。使用 JDBC 时，要求加载.class 文件的方式有四种：

- (1) 使用 `Class.forName()`。
- (2) 自行创建 `Driver` 接口实现类的实例。
- (3) 启动 JVM 时指定 `jdbc.drivers` 属性。
- (4) 设置 JAR 中 `/services/java.sql.Driver` 文件。

第一种方式刚才已经说明。第二种方式就是直接编写代码：

```
java.sql.Driver driver = new com.mysql.jdbc.Driver();
```

由于要创建对象，基本上就要加载.class 文件，自然也就会运行类的静态区块完成驱动程序注册。第三种方式就是运行 `java` 命令时如下：

```
> java -Djdbc.drivers=com.mysql.jdbc.Driver;ooo.XXXDriver YourProgram
```

你的应用程序可能同时连接多个厂商的数据库，所以 `DriverManager` 也可以注册多个驱动程序实例。以上方式如果需要指定多个驱动程序类时，就是用分号隔开。第四种方式则是 Java SE 6 之后 JDBC 4.0 的新特性，只要在驱动程序实现的 JAR 文件/services 文件夹中，放置一个 `java.sql.Driver` 文件，当中编写 `Driver` 接口的实现类名称全名，`DriverManager` 会自动读取这个文件并找到指定类进行注册。

➊ 取得 `Connection` 实现对象

`Connection` 接口的实现对象，是数据库连接代表对象。要取得 `Connection` 实现对象，可以通过 `DriverManager` 的 `getConnection()`：

```
Connection conn = DriverManager.getConnection(
    jdbcUrl, username, password);
```

除了基本的用户名、密码之外，还必须提供 JDBC URL，其定义了连接数据库时的协议、子协议、数据源标识：

协议:子协议:数据源标识

实际上除了“协议”在 JDBC URL 中总是 jdbc 开始之外, JDBC URL 格式各家数据库都不相同, 必须查询数据库产品使用手册。以 MySQL 为例, “子协议”是桥接的驱动程序、数据库产品名称或连接机制。例如, 若使用 MySQL, 子协议名称是 mysql。“数据源标识”标出数据库的地址、端口、名称、用户、密码等信息。举个例子来说, MySQL 的 JDBC URL 编写方式如下:

```
jdbc:mysql://主机名称:连接端口/数据库名称?参数=值&参数=值
```

主机名称可以是本机(localhost)或其他连接主机名称、地址, MySQL 连接端口默认为 3306。例如, 要连接 demo 数据库, 并指明用户名与密码, 可以这样指定:

```
jdbc:mysql://localhost:3306/demo?user=root&password=123456
```

如果要使用中文访问, 还必须给定参数 useUnicode 及 characterEncoding, 表明是否使用 Unicode 并指定字符编码方式。例如(假设数据库表格编码使用 UTF8):

```
jdbc:mysql://localhost:3306/demo?user=root&password=123&useUnicode=true&characterEncoding=UTF8
```

有时会将 JDBC URL 编写在 XML 配置文档中, 此时不能直接在 XML 中写 & 符号, 而必须改写为 & 替代字符。例如:

```
jdbc:mysql://localhost:3306/demo?user=root&password=123&useUnicode=true&characterEncoding=UTF8
```

如果要直接通过 `DriverManager` 的 `getConnection()` 连接数据库, 一个比较完整的代码段如下:

```
Connection conn = null;
SQLException ex = null;
try {
    String url = "jdbc:mysql://localhost:3306/demo";
    String user = "root";
    String password = "123456";
    conn = DriverManager.getConnection(url, user, password);
    ...
}
catch(SQLException e) {
    ex = e;
}
finally {
    if(conn != null) {
        try {
            conn.close();
        }
        catch(SQLException e) {
            if(ex == null) {
                ex = e;
            }
        }
    }
}
if(ex != null) {
```

```

        throw new RuntimeException(ex);
    }
}

```

`SQLException` 是在处理 JDBC 时经常遇到的一个异常对象，为数据库操作过程发生错误时的代表对象。`SQLException` 是受检异常(Checked Exception)，必须使用 `try...catch` 明确处理，在异常发生时尝试关闭相关资源。

提示»» `SQLException` 有个子类 `SQLWarning`，如果数据库执行过程中发生了一些警示信息，会创建 `SQLWarning` 但不会抛出(throw)，而是以链接方式收集起来，可以使用 `Connection`、`Statement`、`ResultSet` 的 `getWarnings()` 来取得第一个 `SQLWarning`，使用这个对象的 `getNextWarning()` 可以取得下一个 `SQLWarning`。由于它是 `SQLException` 的子类，所以必要时也可当作异常抛出。

► 关闭 Connection 实现对象

取得 `Connection` 对象之后，可以使用 `isClosed()` 方法测试与数据库的连接是否关闭。在操作完数据库之后，若确定不再需要连接，则必须使用 `close()` 来关闭与数据库的连接，以释放连接时相关的必要资源，如连接相关对象、授权资源等。

以上是编写程序上的一些简介，然而在底层，`DriverManager` 如何进行连接呢？`DriverManager` 会在循环中逐一取出注册的每个 `Driver` 实例，使用指定的 JDBC URL 来调用 `Driver` 的 `connect()` 方法，尝试取得 `Connection` 实例。以下是 `DriverManager` 中相关源代码的重点节录：

```

SQLException reason = null;
for (int i = 0; i < drivers.size(); i++) { // 逐一取得 Driver 实例
    ...
    DriverInfo di = (DriverInfo) drivers.elementAt(i);
    ...
    try {
        Connection result = di.driver.connect(url, info); // 尝试连接
        if (result != null) {
            return (result); // 取得 Connection 就返回
        }
    } catch (SQLException ex) {
        if (reason == null) { // 记录第一个发生的异常
            reason = ex;
        }
    }
}
if (reason != null) {
    println("getConnection failed: " + reason);
    throw reason; // 如果有异常对象就抛出
}
throw new SQLException( // 没有适用的 Driver 实例，抛出异常
    "No suitable driver found for " + url, "08001");

```

`Driver` 的 `connect()` 方法在无法取得 `Connection` 时会返回 `null`，所以简单来说，`DriverManager` 就是逐一使用 `Driver` 实例尝试连接。如果连接成功就返回 `Connection` 对象，如果有异常发生，`DriverManager` 会记录第一个异常，并继续尝试其他的 `Driver`。在所有 `Driver` 都试过了也无法取得连接，若原先尝试过程中有记录异常就抛出，没有的话，也是抛出异常告知没有适合的驱动程序。

提示»» 偶而为了调试或其他目的，也可自行创建 `Driver` 实例并调用其 `connect()` 方法以取得 `Connection` 对象。例如：

```
Properties props = new Properties();
props.put("user", "root");
props.put("password", "123456");
Driver driver = new com.mysql.jdbc.Driver();
conn = driver.connect(url, props);
```

以下先来示范连接数据库的完整范例，假设使用了以下命令在 MySQL 后创建了 demo 数据库：

```
CREATE schema demo;
```

下面编写一个简单的 JavaBean 来测试一下可否连接数据库并取得 `Connection` 实例

JDBC Demo DbBean.java

```
package cc.openhome;

import java.sql.*;
import java.io.*;

public class DbBean implements Serializable {
    private String jdbcUrl;
    private String username;
    private String password;

    public DbBean() {
        try {
            Class.forName("com.mysql.jdbc.Driver"); ← 加载驱动程序
        } catch (ClassNotFoundException ex) {
            throw new RuntimeException(ex);
        }
    }

    public boolean isConnectedOK() {
        boolean ok = false;
        Connection conn = null;
        SQLException ex = null;
        try {
            conn = DriverManager.getConnection( ← 取得 Connection 对象
                jdbcUrl, username, password);
            if (!conn.isClosed()) {
                ok = true;
            }
        }
```

```

        } catch (SQLException e) {
            ex = e;
        } finally {
            if (conn != null) {
                try {
                    conn.close(); ← 关闭连线
                } catch (SQLException e) {
                    if(ex == null) {
                        ex = e;
                    }
                }
            }
            if(ex != null) {
                throw new RuntimeException(ex);
            }
        }
        return ok;
    }

    public void setPassword(String password) {
        this.password = password;
    }

    public void setJdbcUrl(String jdbcUrl) {
        this.jdbcUrl = jdbcUrl;
    }

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

```

可以通过调用 `isConnectedOK()` 方法来看看是否可以连接成功。例如，可以写个简单的 JSP 网页如下：

JDBC Demo conn.jsp

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<jsp:useBean id="db" class="cc.openhome.DbBean"/>
<c:set target="${db}" property="jdbcUrl"
       value="jdbc:mysql://localhost:3306/demo"/>
<c:set target="${db}" property="username" value="root"/>
<c:set target="${db}" property="password" value="123456"/>
<!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>测试数据库连接</title>
    </head>
    <body>

```

```

<c:choose>
    <c:when test="\${db.connectedOK}">连接成功! </c:when>
    <c:otherwise>连接失败! </c:otherwise>
</c:choose>
</body>
</html>

```

在这个 JSP 页面中，通过`<jsp:useBean>`来创建 JavaBean 实例，并通过 JSTL 的`<c:set>`标签来设置 JavaBean 的每个属性，而后通过`<c:when>`与 EL 来测试一下`isConnectedOK()`的返回值。若为`true`则显示“连接成功！”，否则会显示`<c:otherwise>`中的“连接失败！”。

提示»» 实际上 Web 应用程序很少直接从`DriverManager`中取得`Connection`，而是会通过 JNDI 从服务器上取得设置好的`DataSource`，再从`DataSource`取得`Connection`，这稍后就会介绍。

9.1.3 使用 Statement、ResultSet

`Connection`是数据库连接的代表对象，接下来若要执行 SQL，必须取得`java.sql.Statement`对象，它是 SQL 语句的代表对象，可以使用`Connection`的`createStatement()`来创建`Statement`对象：

```
Statement stmt = conn.createStatement();
```

取得`Statement`对象之后，可以使用`executeUpdate()`、`executeQuery()`等方法来执行 SQL。`executeUpdate()`主要是用来执行`CREATE TABLE`、`INSERT`、`DROP TABLE`、`ALTER TABLE`等会改变数据库内容的 SQL。例如，可以在`demo`数据库中创建一个`t_message`表格：

```

Use demo;
CREATE TABLE t_message (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name CHAR(20) NOT NULL,
    email CHAR(40),
    msg TEXT NOT NULL
) CHARSET=UTF8;

```

如果要在这个表格中插入一笔数据，可以如下使用`Statement`的`executeUpdate()`方法：

```
stmt.executeUpdate("INSERT INTO t_message VALUES(1, 'justin', " +
    "'justin@mail.com', 'mesage...')");
```

`Statement`的`executeQuery()`方法则是用于`SELECT`等查询数据库的 SQL，`executeUpdate()`会返回`int`结果，表示数据变动的笔数，`executeQuery()`会返回`java.sql.ResultSet`对象，代表查询的结果，查询的结果会是一笔一笔的数据。可以使用`ResultSet`的`next()`来移动至下一笔数据，它会返回`true`或`false`表示是否有下一笔数据，接着可以使用`getXXX()`来取得数据，如`getString()`、`getInt()`、`getFloat()`、

`getDouble()` 等方法，分别取得相对应的字段类型数据。`getXXX()` 方法都提供有依据字段名称取得数据，或是依据字段顺序取得数据的方法。一个例子如下，指定字段名称来取得数据：

```
ResultSet result = stmt.executeQuery("SELECT * FROM t_message");
while(result.next()) {
    int id = result.getInt("id");
    String name = result.getString("name");
    String email = result.getString("email");
    String msg = result.getString("msg");
    // ...
}
```

使用查询结果的字段顺序来显示结果的方式如下(注意索引是从 1 开始)：

```
ResultSet result = stmt.executeQuery("SELECT * FROM t_message");
while(result.next()) {
    int id = result.getInt(1);
    String name = result.getString(2);
    String email = result.getString(3);
    String msg = result.getString(4);
    // ...
}
```

`Statement` 的 `execute()` 可以用来执行 SQL，并可以测试所执行的 SQL 是执行查询或更新，返回 `true` 的话表示 SQL 执行将返回 `ResultSet` 表示查询结果，此时可以使用 `getResultSet()` 取得 `ResultSet` 对象。如果 `execute()` 返回 `false`，表示 SQL 执行会返回更新笔数或没有结果，此时可以使用 `getUpdateCount()` 取得更新笔数。如果事先无法得知是进行查询或更新，就可以使用 `execute()`。例如：

```
if(stmt.execute(sql)) {
    ResultSet rs = stmt.getResultSet(); // 取得查询结果 ResultSet
    ...
}
else { // 这是个更新操作
    int updated = stmt.getUpdateCount(); // 取得更新笔数
    ...
}
```

视需求而定，`Statement` 或 `ResultSet` 在不使用时，可以使用 `close()` 将之关闭，以释放相关资源，`Statement` 关闭时，所关联的 `ResultSet` 也会自动关闭。



接下来实现一个简单的留言板作为示范，这个简单的留言板采用 Model 1 架构，使用 JSP 结合 JavaBean 来完成。首先是 JavaBean 的实现：

JDBC Demo GuestBookBean.java

```
package cc.openhome;

import java.sql.*;
import java.util.*;
import java.io.*;
```

```

public class GuestBookBean implements Serializable {
    private String jdbcUrl = "jdbc:mysql://localhost:3306/demo";
    private String username = "root";
    private String password = "123456";
    public GuestBookBean() {
        try {
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException ex) {
            throw new RuntimeException(ex);
        }
    }

    public void setMessage(Message message) { ← ① 这个方法会在数据库中新增留言
        Connection conn = null;
        Statement statement = null;
        SQLException ex = null;
        try {
            conn = DriverManager.getConnection( ← ② 取得 Connection 对象
                jdbcUrl, username, password);
            statement = conn.createStatement(); ← ③ 创建 Statement 对象
            statement.executeUpdate( ← ④ 执行 SQL 陈述句
                "INSERT INTO t_message(name, email, msg) VALUES ('"
                + message.getName() + "', '"
                + message.getEmail() + "', '"
                + message.getMsg() + "')");
        } catch (SQLException e) {
            ex = e;
        } finally { ← ⑤ 在 finally 中关闭 Statement 与 Connection
            if (statement != null) {
                try {
                    statement.close();
                }
                catch(SQLException e) {
                    if(ex == null) {
                        ex = e;
                    }
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                }
                catch(SQLException e) {
                    if(ex == null) {
                        ex = e;
                    }
                }
            }
            if(ex != null) {
                throw new RuntimeException(ex);
            }
        }
    }
}

```

```
}

public List<Message> getMessages() {❶这个方法会从数据库中查询所有留言
    Connection conn = null;
    Statement statement = null;
    ResultSet result = null;
    SQLException ex = null;
    List<Message> messages = null;
    try {
        conn = DriverManager.getConnection(
            jdbcUrl, username, password);
        statement = conn.createStatement();
        result = statement.executeQuery("SELECT * FROM t_message");
        messages = new ArrayList<Message>();
        while (result.next()) {
            Message message = new Message();
            message.setId(result.getLong(1));
            message.setName(result.getString(2));
            message.setEmail(result.getString(3));
            message.setMsg(result.getString(4));
            messages.add(message);
        }
    } catch (SQLException e) {
        ex = e;
    } finally {
        if (statement != null) {
            try {
                statement.close();
            }
            catch(SQLException e) {
                if(ex == null) {
                    ex = e;
                }
            }
        }
        if (conn != null) {
            try {
                conn.close();
            }
            catch(SQLException e) {
                if(ex == null) {
                    ex = e;
                }
            }
        }
        if(ex != null) {
            throw new RuntimeException(ex);
        }
    }
    return messages;
}
```

```
}
```

这个对象会从 `DriverManager` 取得 `Connection` 对象②。`setMessage()` 会接受一个 `Message` 对象①，实现中会在数据库中利用 `Statement` 对象③执行 SQL 语句来添加一个留言④。`getMessages()` 会从数据库中将所有留言取回，并放在一个 `List` 对象中返回⑤。最重要的是注意到，在不使用 `Connection`、`Statement` 或 `ResultSet` 时，要将之关闭以释放相关资源⑥。

提示»» JDBC 规范提到关闭 `Connection` 时，会关闭相关资源，但没有明确说明是哪些相关资源。通常驱动程序实现时，会在关闭 `Connection` 时，一并关闭关联的 `Statement`，但最好留意是否真的关闭了资源，自行关闭 `Statement` 是比较保险的做法。在处理 `SQLException` 的 `try...catch` 中释放相关资源很重要，但结果就是冗长琐碎的代码，有兴趣的话，可以了解 Spring 的 `JdbcTemplate`，了解如何实现与使用，这有助于避免 `SQLException` 处理时琐碎的代码。

可以编写一个简单的 JSP 页面来使用这个 JavaBean。例如：

```
JDBC Demo guestbook.jsp
```

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<c:set target="\${pageContext.request}"
       property="characterEncoding" value="UTF-8"/>———— 设置请求编码处
<jsp:useBean id="guestbook"———— 使用 GuestBookBean
              class="cc.openhome.GuestBookBean" scope="application"/>
<c:if test="\${param.msg != null}"———— 如果是要新增留言的话
      <jsp:useBean id="newMessage" class="cc.openhome.Message"/>
      <jsp:setProperty name="newMessage" property="*"/>
      <c:set target="\${guestbook}"———— 调用 setMessage()方法新增留言
             property="message" value="\${newMessage}"/>
</c:if>
<!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>访客留言板</title>
  </head>
  <body>
    <table style="text-align: left; width: 100%;" border="0"
           cellpadding="2" cellspacing="2">
      <tbody>
        <c:forEach var="message" items="\${guestbook.messages}">
          <tr>
            <td>\${message.name}</td>
            <td>\${message.email}</td>
            <td>\${message.msg}</td>
          </tr>
        </c:forEach>
      </tbody>
    </table>
  </body>
</html>
```

↑ 调用 getMessages()
方法取得留言

```

</c:forEach>
</tbody>
</table>
</body>
</html>

```

这个 JSP 页面基本上就是利用 GuestBookBean 来新增留言或取得留言并显示它。由于加载驱动程序的操作只需要一次，而且这个 JavaBean 没有状态，所以将 GuestBookBean 设置为 application 范围。这样只有在第一次请求时会创建 GuestBook Bean，之后 GuestBookBean 实例就存在应用程序范围中。图 9.11 所示为执行时的一个参考画面。

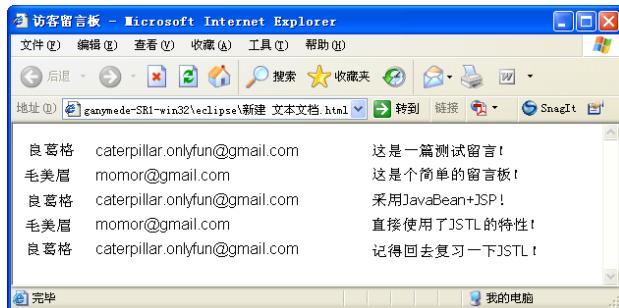


图 9.11 结合数据库访问的简单留言板

提示»» 第 7 章已经介绍过 JSTL 了。之后的范例若需要使用到 JSP，都会充分利用 JSTL 的特性来呈现页面逻辑。如果对有些 JSTL 不熟，记得复习一下第 7 章。

9.1.4 使用 PreparedStatement、CallableStatement

Statement 在执行 executeQuery()、executeUpdate() 等方法时，如果有些部分是动态的数据，必须使用 + 运算子串接字符串以组成完整的 SQL 语句，十分不方便。例如，先前范例中在新增留言时，必须如下串接 SQL 语句：

```

statement.executeUpdate(
    "INSERT INTO t_message(name, email, msg) VALUES ('"
    + message.getName() + "', '"
    + message.getEmail() + "', '"
    + message.getMsg() + "')");

```

如果有些操作只是 SQL 语句中某些参数会有所不同，其余的 SQL 子句皆相同，则可以使用 `java.sql.PreparedStatement`。可以使用 Connection 的 `prepareStatement()` 方法创建好一个预编译(precompile)的 SQL 语句，当中参数会变动的部分，先指定 “?” 这个占位字符。例如：

```

PreparedStatement stmt = conn.prepareStatement(
    "INSERT INTO t_message VALUES(?, ?, ?, ?)");

```

等到需要真正指定参数执行时，再使用相对应的 `setInt()`、`setString()` 等方法，指定“?”处真正应该有的参数。例如：

```
stmt.setInt(1, 2);
stmt.setString(2, "momor");
stmt.setString(3, "momor@mail.com");
stmt.setString(4, "message2...");
stmt.executeUpdate();
stmt.clearParameters();
```

要让 SQL 执行生效，需执行 `executeUpdate()` 或 `executeQuery()` 方法（如果是查询的话）。在这次的 SQL 执行完毕后，可以调用 `clearParameters()` 清除所设置的参数，之后就可以再使用这个 `PreparedStatement` 实例，所以使用 `PreparedStatement`，可以让你先准备好一段 SQL，并重复使用这段 SQL 语句。

可以使用 `PreparedStatement` 改写先前 `GuestBookBean` 中 `setMessage()` 执行 SQL 语句的部分。例如：

```
public void setMessage(Message message) {
    Connection conn = null;
    PreparedStatement statement = null;
    SQLException ex = null;
    try {
        conn = DriverManager.getConnection(
            jdbcUrl, username, password);
        statement = conn.prepareStatement(
            "INSERT INTO t_message(name, email, msg) VALUES (?, ?, ?)");
        statement.setString(1, message.getName());
        statement.setString(2, message.getEmail());
        statement.setString(3, message.getMsg());
        statement.executeUpdate();
    } catch (SQLException e) {
        // 略...
    }
    // 略...
}
```

这样的写法显然比串接 SQL 的方式好的多。不过，使用 `PreparedStatement` 的好处不仅如此，之前提过，在这次的 SQL 执行完毕后，可以调用 `clearParameters()` 清除设置的参数，之后就可以再使用这个 `PreparedStatement` 实例。也就是说，必要的话，可以考虑制作语句池(Statement Pool)将一些频繁使用的 `PreparedStatement` 重复使用，减少生成对象的负担。

在驱动程序支持的情况下，使用 `PreparedStatement`，可以将 SQL 语句预编译为数据库的运行命令。由于已经是数据库的可执行命令，运行速度可以快许多[例如若使用 Java DB，其驱动程序可以将 SQL 预编译为比特码(byte code)格式，在 JVM 中运行就快许多了]，而不像 `Statement` 对象，是在执行时将 SQL 直接送到数据库，由数据库做解析、直译再执行。

使用 `PreparedStatement` 在安全上也可以有点贡献。举个例子来说，如果原先使用串接字符串的方式来执行 SQL：

```
Statement statement = connection.createStatement();
String queryString = "SELECT * FROM user_table WHERE username='"
+ username + "' AND password='" + password + "'";
ResultSet resultSet = statement.executeQuery(queryString);
```

其中 `username` 与 `password` 若是来自用户的请求参数，原本是希望用户正确地输入名称和密码，组合之后的 SQL 应该这样：

```
SELECT * FROM user_table
WHERE username='caterpillar' AND password='123456'
```

但如果用户在密码的部分，输入了 “' OR '1'='1” 这样的字符串，而程序又没有针对请求参数的部分进行字符检查过滤动作，这个奇怪的字符串最后组合出来的 SQL 会是以下：

```
SELECT * FROM user_table
WHERE username='caterpillar' AND password=''' OR '1'='1''
```

方框是密码请求参数的部分，将方框拿掉会更清楚地看出这个 SQL 有什么问题！

```
SELECT * FROM user_table
WHERE username='caterpillar' AND password=''' OR '1'='1''
```

AND 子句之后的判断式永远成立，也就是说，用户不用输入正确的密码，也可以查询出所有的数据，这就是 SQL Injection 的简单例子。

以串接的方式组合 SQL 语句基本上就会有 SQL Injection 的隐患，如果这样改用 `PreparedStatement` 的话：

```
PreparedStatement stmt = conn.prepareStatement(
    "SELECT * FROM user_table WHERE username=? AND password=?");
stmt.setString(1, username);
stmt.setString(2, password);
```

在这里 `username` 与 `password` 将被视作是 SQL 中纯粹的字符串，而不会被当作 SQL 语法来解释，所以就可避免这个例子的 SQL Injection 问题。

提示»» 先前介绍过滤器时，也曾提过用户在字段中直接输入 HTML 字符的问题。这类安全问题的防治基本在于，不允许用户输入的特殊字符，一开始就应该适当地过滤或取代掉。

其实问题不仅是在串接字符串本身麻烦，以及 SQL Injection 发生的可能性。由于+串接字符串会产生新的 `String` 对象，如果串接字符串动作经常进行(例如在循环中进行 SQL 串接的动作)，那会是性能负担上的隐忧(如果真的非得串接 SQL，至少要考虑使用 `StringBuffer` 或 JDK 5.0 之后的 `StringBuilder`)。

如果编写数据库的存储过程(Stored Procedure)，并想使用 JDBC 来调用，则可使用 `java.sql.CallableStatement`。调用的基本语法如下：

```
{?= call <程序名称>[<自变量 1>,<自变量 2>, ...]}
{call <程序名称>[<自变量 1>,<自变量 2>, ...]}
```

`CallableStatement` 的 API 使用，基本上与 `PreparedStatement` 差别不大，除了必须调用 `prepareCall()` 创建 `CallableStatement` 时异常，一样是使用 `setXXX()` 设置参数，如果是查询操作，使用 `executeQuery()`；如果是更新操作，使用 `executeUpdate()`。另外，可以使用 `registerOutParameter()` 注册输出参数等。

提示» 使用 JDBC 的 `CallableStatement` 调用存储过程，重点是在于了解各个数据库的存储过程如何编写及相关事宜，用 JDBC 调用存储过程，也表示你的应用程序将与数据库产生直接的相关性。

在使用 `PreparedStatement` 或 `CallableStatement` 时，必须注意 SQL 类型与 Java 数据类型的对应，因为两者本身并不是一对一对应，`java.sql.Types` 定义了一些常数代表 SQL 类型。表 9.1 所示为 JDBC 规范建议的 SQL 类型与 Java 类型的对应。

表 9.1 Java 类型与 SQL 类型对应

| Java 类型 | SQL 类型 |
|-----------------------------------|------------------------------------------------------------------------|
| <code>boolean</code> | <code>BIT</code> |
| <code>byte</code> | <code>TINYINT</code> |
| <code>short</code> | <code>SMALLINT</code> |
| <code>int</code> | <code>INTEGER</code> |
| <code>long</code> | <code>BIGINT</code> |
| <code>float</code> | <code>FLOAT</code> |
| <code>double</code> | <code>DOUBLE</code> |
| <code>byte[]</code> | <code>BINARY</code> 、 <code>VARBINARY</code> 、 <code>LONGBINARY</code> |
| <code>java.lang.String</code> | <code>CHAR</code> 、 <code>VARCHAR</code> 、 <code>LONGVARCHAR</code> |
| <code>java.math.BigDecimal</code> | <code>NUMERIC</code> 、 <code>DECIMAL</code> |
| <code>java.sql.Date</code> | <code>DATE</code> |
| <code>java.sql.Time</code> | <code>TIME</code> |
| <code>java.sql.Timestamp</code> | <code>TIMESTAMP</code> |

其中要注意的是，日期时间在 JDBC 中，并不是使用 `java.util.Date`，这个对象可代表的日期时间格式是“年、月、日、时、分、秒、毫秒”。在 JDBC 中要表示日期，是使用 `java.sql.Date`，其日期格式是“年、月、日”；要表示时间的话则是使用 `java.sql.Time`，其时间格式为“时、分、秒”；如果要表示“时、分、秒、微秒”的格式，则是使用 `java.sql.Timestamp`。

9.2 JDBC 进阶

上一节介绍了 JDBC 入门观念与相关 API，在这一节，将说明更多进阶 API 的使用，如使用 `DataSource` 取得 `Connection`、使用 `PreparedStatement` 和 `ResultSet` 进行更新操作等。

9.2.1 使用 `DataSource` 取得连接

先前的 `DbBean`、`GuestBookBean` 范例自行负责了加载 JDBC 驱动程序、告知 `DriverManager` 有关 JDBC URL、用户名、密码等信息，以取得 `connection` 对象。假设日后需要更换驱动程序、修改数据库服务器主机位置，或者是为了打算重复利用 `Connection` 对象而想要加入连接池(Connection Pool)机制等情况，就要针对相对应的代码进行修改。

提示» 要取得数据库连接，必须打开网络连接(中间经过实体网络)，连接至数据库服务器后，进行协议交换(当然也就是数次的网络数据往来)以进行验证名称、密码等确认动作。也就是取得数据库连接是件耗时间及资源的动作。尽量利用已打开的连接，也就是重复利用取得的 `Connection` 实例，是改善数据库连接性能的一个方式，采用连接池是基本做法。

由于取得 `Connection` 的方式，依使用的环境及程序需求而有所不同，直接在代码中写死取得 `Connection` 的方式并不是明智之举。在 Java EE 的环境中，将取得连接等与数据库来源相关的行为规范在 `javax.sql.DataSource` 接口，实际如何取得 `Connection` 则由实现接口的对象来负责。

所以问题简化到如何取得 `DataSource` 实例。为了让应用程序在需要取得某些与系统相关的资源对象时，能与实际的系统资源配置、实体机器位置、环境架构等无关，在 Java 应用程序中可以通过 JNDI(Java Naming Directory Interface)来取得所需的资源对象。举例来说，如果是在 Web 应用程序中想要获得 `DataSource` 实例，可以这样进行：

```
try {
    Context initContext = new InitialContext();
    Context envContext = (Context) initContext.lookup("java:/comp/env");
    dataSource = (DataSource) envContext.lookup("jdbc/demo");
} catch (NamingException ex) {
    ...
}
```

在创建 `Context` 对象的过程中会收集环境相关数据，之后根据 JNDI 名称 `jdbc/demo` 向 JNDI 服务器查找 `DataSource` 实例并返回。在这个代码段中，不会知道实际的资源配置、实体机器位置、环境架构等信息，应用程序不会与这些信息发生相关。

提示»» 如果只是利用 JNDI 来查找某些资源对象，上面这个代码段就是你对 JNDI 所需要知道的东西了，其他的细节就交给服务器管理员做好相关设置，让 jdbc/demo 对应取得 DataSource 实例即可(如果你的职责不在于管理机器的话)。



举个实际的例子来说，如果你只负责编写 Web 应用程序，或更具体一点，如果只是要编写如先前范例的 DbBean 类，且已经有服务器管理员设置好 jdbc/demo 这个 JNDI 名称的对应资源了，那么可以这么编写程序：

JDBC Demo DatabaseBean.java

```
package cc.openhome;

import java.io.Serializable;
import java.sql.*;
import javax.naming.*;
import javax.sql.DataSource;

public class DatabaseBean implements Serializable {
    private DataSource dataSource;

    public DatabaseBean() {
        try {
            Context initContext = new InitialContext();
            Context envContext = (Context)
                initContext.lookup("java:/comp/env");
            dataSource = (DataSource) envContext.lookup("jdbc/demo");
        } catch (NamingException ex) {
            throw new RuntimeException(ex);
        }
    }

    public boolean isConnectedOK() {
        boolean ok = false;
        Connection conn = null;
        SQLException ex = null;
        try {
            conn = dataSource.getConnection(); ← 通过 DataSource 对象取得连线
            if (!conn.isClosed()) {
                ok = true;
            }
        } catch (SQLException e) {
            ex = e;
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    if (ex == null) {
                        ex = e;
                    }
                }
            }
        }
    }
}
```

↑ 查找 jdbc/demo 对应的
DataSource 对象

```

        if(ex != null) {
            throw new RuntimeException(ex);
        }
    }
    return ok;
}
}

```



只看这里的代码的话，不会知道实际上使用哪个驱动程序、数据库用户名、密码是什么(或许数据库管理员本来就不想让你知道)、数据库实体地址、连接端口、名称、是否有使用连接池等。这些都该由数据库管理员或服务器管理员负责设置，你唯一要知道的就是 `jdbc/demo` 这个 JNDI 名称，并且要告诉 Web 容器，也就是要在 `web.xml` 中设置：

JDBC Demo web.xml

```

</web-app ...>
    // 略...
<resource-ref>
    <res-ref-name>jdbc/demo</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
</web-app>

```

在 `web.xml` 中设置的目的，是要让 Web 容器提供 JNDI 查找时所需的相关环境信息，这样创建 `Context` 对象时就不用设置一大堆参数。接着可以编写一个简单的 JSP 来使用 `DatabaseBean`：

JDBC Demo conn2.jsp

```

<%@page contentType="text/html" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<jsp:useBean id="db" class="cc.openhome.DatabaseBean"/>
<!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>测试数据库连接</title>
    </head>
    <body>
        <c:choose>
            <c:when test="${db.connectedOK}">连接成功! </c:when>
            <c:otherwise>连接失败! </c:otherwise>
        </c:choose>
    </body>
</html>

```

就一个 Java 开发人员来说，工作已经完成了。现在假设你是服务器管理员，职责就是设置 JNDI 相关资源，但设置的方式并非标准的一部分，而是依应用程序服务器而有所不同。假设应用程序将部署在 Tomcat 7 上，则可以要求 Web 应用程序在封装为 WAR 文件时，必须在 META-INF 文件夹中包括一个 context.xml：

JDBC Demo context.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/JDBC Demo">
    <Resource name="jdbc/demo"
        auth="Container" type="javax.sql.DataSource"
        maxActive="100" maxIdle="30" maxWait="10000" username="root"
        password="123456" driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/demo?
            useUnicode=true&characterEncoding=UTF8"/>
</Context>
```

最主要的可以看到 name 属性是设置 JNDI 名称为 jdbc/demo，username 与 password 是数据库用户名与密码，driverClassName 为驱动程序类名称，url 为 JDBC URL，因为是编写在 XML 中，所以必须使用 & 取代。至于其他的属性设置，则是与 DBCP(Database Connection Pool)有关，这是内置在 Tomcat 中的连接池机制。有兴趣的话，可以访问 <http://commons.apache.org/dbcp/> 了解它提供的连接池功能。

当应用程序部署之后，Tomcat 会根据 META-INF 中 context.xml 的设置，寻找指定的驱动程序，所以必须将驱动程序的 JAR 文件放置在 Tomcat 的 lib 目录中，接着 Tomcat 就会为 JNDI 名称 jdbc/demo 设置相关的资源。

9.2.2 使用 ResultSet 卷动、更新数据

在 ResultSet 时，默认可以使用 next() 移动数据光标至下一个数据，而后使用 getXXX() 方法来取得数据。实际上，从 JDBC 2.0 开始，ResultSet 并不仅 can 使用 previous()、first()、last() 等方法前后移动数据光标，还可以调用 updateXXX()、updateRow() 等方法进行数据修改。

在使用 Connection 的 createStatement() 或 prepareStatement() 方法创建 Statement 或 PreparedStatement 实例时，可以指定结果集类型与并行方式：

```
createStatement(int resultSetType, int resultSetConcurrency)
prepareStatement(String sql,
    int resultSetType, int resultSetConcurrency)
```

结果集类型可以指定三种设置：

- ResultSet.TYPE_FORWARD_ONLY(默认)
- ResultSet.TYPE_SCROLL_INSENSITIVE
- ResultSet.TYPE_SCROLL_SENSITIVE

指定为 `TYPE_FORWARD_ONLY`, `ResultSet` 就只能前进数据光标, 指定为 `TYPE_SCROLL_INSENSITIVE` 或 `TYPE_SCROLL_SENSITIVE`, 则 `ResultSet` 可以前后移动数据光标。两者差别在于 `TYPE_SCROLL_INSENSITIVE` 设置下, 取得的 `ResultSet` 不会反应数据库中的数据修改, 而 `TYPE_SCROLL_SENSITIVE` 会反应数据库中的数据修改。

更新设置可以有两种指定:

- `ResultSet.CONCUR_READ_ONLY`(默认)
- `ResultSet.CONCUR_UPDATABLE`

指定为 `CONCUR_READ_ONLY`, 则只能用 `ResultSet` 进行数据读取, 无法进行更新。指定为 `CONCUR_UPDATABLE`, 就可以使用 `ResultSet` 进行数据更新。

在使用 `Connection` 的 `createStatement()` 或 `prepareStatement()` 方法创建 `Statement` 或 `PreparedStatement` 实例时, 若没有指定结果集类型与并行方式, 默认就是 `TYPE_FORWARD_ONLY` 与 `CONCUR_READ_ONLY`。如果想前后移动数据光标并想使用 `ResultSet` 进行更新, 则以下是个 `Statement` 指定的例子:

```
Statement stmt = conn.createStatement(
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATEABLE);
```

以下是个 `PreparedStatement` 指定的例子:

```
PreparedStatement stmt = conn.prepareStatement(
    "SELECT * FROM t_message",
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_UPDATEABLE);
```

在数据光标移动的 API 上, 可以使用 `absolute()`、`afterLast()`、`beforeFirst()`、`first()`、`last()` 进行绝对位置移动, 使用 `relative()`、`previous()`、`next()` 进行相对位置移动, 这些方法如果成功移动就会返回 `true`。也可以使用 `isAfterLast()`、`isBeforeFirst()`、`isFirst()`、`isLast()` 判断当前位置。以下是个简单的程序范例片段:

```
Statement stmt = conn.createStatement("SELECT * FROM t_message",
    ResultSet.TYPE_SCROLL_INSENSITIVE,
    ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery();
rs.absolute(2);           // 移至第 2 行
rs.next();                // 移至第 3 行
rs.first();               // 移至第 1 行
boolean b1 = rs.isFirst(); // b1 是 true
```

如果要使用 `ResultSet` 进行数据修改, 则有些条件限制:

- 必须选择单一表格
- 必须选择主键
- 必须选择所有 `NOT NULL` 的值

在取得 `ResultSet` 之后要进行数据更新, 必须移动至要更新的行(`Row`), 调用 `updateXxx()` 方法(`Xxx` 是类型), 而后调用 `updateRow()` 方法完成更新。如果调用

`cancelRowUpdates()` 可取消更新，但必须在调用 `updateRow()` 前进行更新的取消。一个使用 `ResultSet` 更新数据的例子如下：

```
Statement stmt = conn.prepareStatement("SELECT * FROM t_message",
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery();
rs.next();
rs.updateString(3, "caterpillar@openhome.cc");
rs.updateRow();
```

如果取得 `ResultSet` 后想直接进行数据的新增，则要先调用 `moveToInsertRow()`，之后调用 `updateXxx()` 设置要新增的数据各个字段，然后调用 `insertRow()` 新增数据。一个使用 `ResultSet` 新增数据的例子如下：

```
Statement stmt = conn.prepareStatement("SELECT * FROM t_message",
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery();
rs.moveToInsertRow();
rs.updateString(2, "momor");
rs.updateString(3, "momor@openhome.cc");
rs.updateString(4, "blah..blah");
rs.insertRow();
rs.moveToCurrentRow();
```

如果取得 `ResultSet` 后想直接进行数据的删除，则要移动数据光标至想删除的列，调用 `deleteRow()` 删除数据列。一个使用 `ResultSet` 删除数据的例子如下：

```
Statement stmt = conn.prepareStatement("SELECT * FROM t_message",
        ResultSet.TYPE_SCROLL_INSENSITIVE,
        ResultSet.CONCUR_READ_ONLY);
ResultSet rs = stmt.executeQuery();
rs.absolute(3);
rs.deleteRow();
```

9.2.3 批次更新

如果必须对数据库进行大量数据更新，单纯使用类似以下的代码段并不合适：

```
Statement stmt = conn.createStatement();
while(someCondition) {
    stmt.executeUpdate(
        "INSERT INTO t_message(name,email,msg) VALUES ('...', '...', '...')");
}
```

每一次执行 `executeUpdate()`，其实都会向数据库发送一次 SQL。如果大量更新的 SQL 有一万次，就等于通过网络进行了一万次的信息传送。网络传递信息实际上必须启动 I/O、进行路由等动作，这样进行大量更新，性能上其实不好。

可以使用 `addBatch()` 方法来收集 SQL，并使用 `executeBatch()` 方法将所收集的 SQL 传出去。例如：

```

Statement stmt = conn.createStatement();
while(someCondition) {
    stmt.addBatch(
        "INSERT INTO t_message(name,email,msg) VALUES ('...', '...', '...')");
}
stmt.executeBatch();

```

以 MySQL 驱动程序的 Statement 实现为例，其 `addBatch()` 使用了 `ArrayList` 来收集 SQL。其源代码如下所示：

```

public synchronized void addBatch(String sql) throws SQLException {
    if (this.batchedArgs == null) {
        this.batchedArgs = new ArrayList();
    }
    if (sql != null) {
        this.batchedArgs.add(sql);
    }
}

```

所有收集的 SQL，最后会串为一句 SQL，然后传送给数据库。也就是说，假设大量更新的 SQL 有一万笔，这一万笔 SQL 会连接为一句 SQL，再通过一次网络传送给数据库，节省了 I/O、网络路由等操作所耗费的时间。

既然是使用批次更新，顾名思义，就是仅用在更新操作。所以批次更新的限制是，SQL 不能是 SELECT，否则会抛出异常。

使用 `executeBatch()` 时，SQL 的执行顺序就是 `addBatch()` 时的顺序，`executeBatch()` 会返回 `int[]`，代表每笔 SQL 造成的数据异动列数。执行 `executeBatch()` 时，先前已打开的 `ResultSet` 会被关闭，执行过后收集 SQL 用的 `List` 会被清空，任何的 SQL 错误会抛出 `BatchUpdateException`，可以使用这个对象的 `getUpdateCounts()` 取得 `int[]`，代表先前执行成功的 SQL 所造成的异动笔数。

先前举的例子是 Statement 的例子，如果是 PreparedStatement 要使用批次更新，以下是个范例：

```

PreparedStatement stmt = conn.prepareStatement(
    "INSERT INTO t_message(name,email,msg) VALUES (?, ?, ?)");
while(someCondition) {
    stmt.setString(1, "..");
    stmt.setString(2, "..");
    stmt.setString(3, "..");
    stmt.addBatch(); // 收集参数
}
stmt.executeBatch(); // 送出所有参数

```

PreparedStatement 的 `addBatch()` 会收集占位字符真正的数值。以 MySQL 的 PreparedStatement 实现类为例，其 `addBatch()` 源代码如下：

```

public void addBatch() throws SQLException {
    if (this.batchedArgs == null) {
        this.batchedArgs = new ArrayList();
    }
    this.batchedArgs.add(new BatchParams(this.parameterValues,

```

```

        this.parameterStreams, this.isStream, this.streamLengths,
        this.isNull));
}

```

可以看到，内部是使用 `ArrayList` 来收集占位字符实际的数值。

提示»» 除了在 API 上使用 `addBatch()`、`executeBatch()` 等方法以进行批次更新之外，通常也会搭配关闭自动提交(`auto commit`)，在性能上也会有所影响，这在稍后说明事务时就会提到。驱动程序本身是否支持批次更新也要注意一下。以 MySQL 为例，要支持批次更新，必须在 JDBC URL 上附加 `rewriteBatchedStatements =true` 参数才有实际的作用。

9.2.4 Blob 与 Clob

如果要将文件写入数据库，可以在数据库表格字段上使用 BLOB 或 CLOB 数据类型。BLOB 全名 Binary Large Object，用于储存大量的二进制数据，如图片、影音文件等。CLOB 全名 Character Large Object，用于储存大量的文字数据。

在 JDBC 中提供了 `java.sql.Blob` 与 `java.sql.Clob` 两个类分别代表 BLOB 与 CLOB 数据。以 `Blob` 为例，写入数据时，可以通过 `PreparedStatement` 的 `setBlob()` 来设置 `Blob` 对象，读取数据时，可以通过 `ResultSet` 的 `getBlob()` 取得 `Blob` 对象。

`Blob` 拥有 `getBinaryStream()`、`getBytes()` 等方法，可以取得代表字段来源的 `InputStream` 或字段的 `byte[]` 数据。`Clob` 拥有 `getCharacterStream()`、`getAsciiStream()` 等方法，可以取得 `Reader` 或 `InputStream` 等数据，可以查看 API 文件来获得更详细的信息。

实际也可以把 BLOG 字段对应 `byte[]` 或输入/输出串流。在写入数据时，可以使用 `PreparedStatement` 的 `setBytes()` 来设置要存入的 `byte[]` 数据，使用 `setBinaryStream()` 来设置代表输入来源的 `InputStream`。在读取数据时，可以使用 `ResultSet` 的 `getBytes()` 以 `byte[]` 取得字段中储存的数据，或以 `getBinaryStream()` 取得代表字段来源的 `InputStream`。

以下是取得代表文件来源的 `InputStream` 后，进行数据库储存的片段：

```

InputStream in = readFileAsInputStream("...");
PreparedStatement stmt = conn.prepareStatement(
    "INSERT INTO IMAGES(src, img) VALUE(?, ?)");
stmt.setString(1, "...");
stmt.setBinaryStream(2, in);
stmt.executeUpdate();

```

以下是取得代表字段数据源的 `InputStream` 的片段：

```

PreparedStatement stmt = conn.prepareStatement(
    "SELECT img FROM IMAGES");
ResultSet rs = stmt.executeQuery();
while(rs.next()) {
    InputStream in = rs.getBinaryStream(1);
    //...使用 InputStream 作数据读取
}

```

下面举个实际例子，制作一个简单的 Web 应用程序，可以让用户上传文件储存到数据库、下载或删除数据库中的文件。首先要创建数据库表：

```
CREATE TABLE t_files (
    id INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    filename VARCHAR(255) NOT NULL,
    savedTime TIMESTAMP NOT NULL,
    bytes LONGBLOB NOT NULL
) CHARSET=UTF8;
```

接着编写一个 `FileService` 类，使用 JDBC 负责数据库操作相关细节：

JDBC Demo FileService.java

```
package cc.openhome;

import java.sql.*;
import java.util.*;
import javax.sql.DataSource;
import javax.naming.*;

public class FileService {
    private DataSource dataSource;

    public FileService() {
        try {
            Context initContext = new InitialContext();
            Context envContext = (Context)
                initContext.lookup("java:/comp/env");
            dataSource = (DataSource) envContext.lookup("jdbc/demo");
        } catch (NamingException ex) {
            throw new RuntimeException(ex);
        }
    }

    public File getFile(File file) {
        Connection conn = null;
        PreparedStatement statement = null;
        ResultSet result = null;
        SQLException ex = null;
        try {
            conn = dataSource.getConnection();
            statement = conn.prepareStatement(
                "SELECT filename, bytes FROM t_files WHERE id=?");
            statement.setLong(1, file.getId());
            result = statement.executeQuery();
            while (result.next()) {
                file = new File();
                file.setFilename(result.getString(1));
                file.setBytes(result.getBytes(2)); ← ③ 取得字节数据
            }
        } catch (SQLException e) {
            ex = e;
        }
    }
}
```

```

} finally {
    if (statement != null) {
        try {
            statement.close();
        }
        catch(SQLException e) {
            if(ex == null) {
                ex = e;
            }
        }
    }

    if (conn != null) {
        try {
            conn.close();
        }
        catch(SQLException e) {
            if(ex == null) {
                ex = e;
            }
        }
    }
}

if(ex != null) {
    throw new RuntimeException(ex);
}
}

return file;
}

public List<File> getFileList() {
    Connection conn = null;
    PreparedStatement statement = null;
    ResultSet result = null;
    SQLException ex = null;
    List<File> fileList = null;
    try {
        conn = dataSource.getConnection();
        statement = conn.prepareStatement(
                    "SELECT id, filename, savedTime FROM t_files");
        result = statement.executeQuery();
        fileList = new ArrayList<File>();
        while (result.next()) {
            File file = new File();
            file.setId(result.getLong(1));
            file.setFilename(result.getString(2));
            file.setSavedTime(result.getTimestamp(3));
            fileList.add(file);
        }
    } catch (SQLException e) {
        ex = e;
    } finally {
        略...
    }
}

④取得文件列表，包括
id、文件名与储存时间

```

```

    }

    return fileList;
}

public void save(File file) {
    Connection conn = null;
    PreparedStatement statement = null;
    SQLException ex = null;
    try {
        conn = dataSource.getConnection();
        statement = conn.prepareStatement(
            "INSERT INTO t_files(filename, savedTime, bytes) VALUES(?, ?, ?)");
        statement.setString(1, file.getFilename());
        statement.setTimestamp(2, new Timestamp(file.getSavedTime().getTime()));
        statement.setBytes(3, file.getBytes()); ← ❸ 设置储存的字节数据
        statement.executeUpdate();
    } catch (SQLException e) {
        ex = e;
    } finally {
        ...略
    }
}

public void delete(File file) {
    Connection conn = null;
    PreparedStatement statement = null;
    SQLException ex = null;
    try {
        conn = dataSource.getConnection();
        statement = conn.prepareStatement(
            "DELETE FROM t_files WHERE id=?");
        statement.setLong(1, file.getId());
        statement.executeUpdate();
    } catch (SQLException e) {
        ex = e;
    } finally {
        ...略
    }
}
}

```

⑤ 新增文件至数据库
⑥ 设置储存的字节数据
⑦ 根据 id 删除文件

FileService 在构造时，会通过 JNDI 查找 DataSource ❶，之后通过 DataSource 来取得 Connection，在 getFile() 方法中，主要是通过 id 在数据库中查找对应的文件名与字节数据 ❷，在取得字节数据时，是通过 ResultSet 的 getBytes() 来取得 ❸。如果要取得所有文件列表，可以通过 FileService 的 getFileList() 方法取得 ❹。在 save() 方法中，则是使用 INSERT 将数据新增至数据库中 ❺，其中字节的部分，是通过 PreparedStatement 的 setBytes() 来新增 ❻。如果要删除文件，则是根据 id 来删除 ❼。

文件的上传、下载与删除，都是在 JSP 页面中进行操作：

JDBC Demo file.jsp

```

<%@page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<jsp:useBean id="fileService"
    class="cc.openhome.FileService" %> ① 创建 JavaBean
    scope="application" />
<!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>文件管理</title>
    </head>
    <body>
        <form method="post" enctype="multipart/form-data" action="upload.do"><br>
            选择文件: <input type="file" name="file"><br><br>
            <input type="submit" value="上传">
        </form>
        <hr>
        <table style="text-align: left;" border="1"
            cellpadding="2" cellspacing="2">
            <tbody>
                <tr>
                    <td>文件名称</td>
                    <td>上传日期</td>
                    <td>操作</td>
                </tr>
                <c:forEach var="file" items="${fileService.fileList}">
                    <tr>
                        <td>${file.filename}</td>
                        <td>${file.savedTime}</td>
                        <td><a href="download.do?id=${file.id}">下载</a> /
                            <a href="delete.do?id=${file.id}">删除</a>
                        </td>
                    </tr>
                </c:forEach>
            </tbody>
        </table>
    </body>
</html>

```

① 创建 JavaBean: An arrow points from this annotation to the line `class="cc.openhome.FileService"`. A bracket groups this line and the line `scope="application" />`.

② 上传窗体: An arrow points from this annotation to the line `action="upload.do">`.

③ 显示文件列表: An arrow points from this annotation to the opening tag of the inner table, `<table style="text-align: left;" border="1"`. Another arrow points from the inner table's closing tag, `</table>`, back to this annotation.

④ 根据 id 下载文件: An arrow points from this annotation to the line `下载`.

⑤ 根据 id 删除文件: An arrow points from this annotation to the line `删除`.

为了简化范例，这里利用 JavaBean 的方式创建 `FileService` 实例，并设置为 `application` 范围属性①。实际上，可以利用 `ServletContextListener`，在应用程序初始时创建 `FileService` 实例，并设置为 `ServletContext` 范围属性，在上传窗体的部分，`action` 是设置为 `upload.do`，以 `POST` 的方式发送②，显示文件列表时，使用 JSTL 的 `<c:forEach>`③。调用 `FileService` 的 `getFileList()` 取得列表后，逐一显示文件名称与上传时间。如果要下载文件，则使用 URL 重写的方式，根据 `id` 向 `download.do` 发送

GET 请求④。如果要删除文件，也是使用 URL 重写的方式，根据 `id` 向 `delete.do` 发送 GET 请求⑤。

处理文件上传的 Servlet 如下：

JDBC Demo Upload.java

```
package cc.openhome;

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

@MultipartConfig
@WebServlet("/upload.do")
public class Upload extends HttpServlet {
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        request.setCharacterEncoding("UTF-8");
        Part part = request.getPart("file");
        String filename = getFilename(part);      ┌─ ① 利用 Part 取得上传文件名、字节
        byte[] bytes = getBytes(part);

        File file = new File();
        file.setFilename(filename);
        file.setBytes(bytes);
        file.setSavedTime(new Date()); ← ② 取得系统时间

        FileService service = (FileService)
            getServletContext().getAttribute("fileService");
        service.save(file); ← ③ 使用 FileService 的 save() 储存

        response.sendRedirect("file.jsp");
    }

    private String getFilename(Part part) {
        String header = part.getHeader("Content-Disposition");
        String filename =
            header.substring(header.indexOf("filename=\"") + 10,
                            header.lastIndexOf("\""));
        return filename;
    }

    private byte[] getBytes(Part part) throws IOException {
        InputStream in = part.getInputStream();
        ByteArrayOutputStream out = new ByteArrayOutputStream();
        byte[] buffer = new byte[1024];
        int length = -1;
        while ((length = in.read(buffer)) != -1) {
            out.write(buffer, 0, length);
        }
    }
}
```

```

        }
        in.close();
        out.close();
        return out.toByteArray();
    }
}

```

在这里利用了 3.2.4 节介绍过的 `Part` 对象来取得上传的文件名与字节❶，上传的时间则是直接创建 `Date` 来取得❷。在创建 `File` 对象封装上传文件的文件名、字节与时间相关信息后，利用 `FileService` 的 `save()` 方法来储存文件❸。

处理文件下载的 Servlet 如下：

JDBC Demo Download.java

```

package cc.openhome;

import java.net.URLEncoder;
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/download.do")
public class Download extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws ServletException, IOException {
        String id = request.getParameter("id");
        File file = new File();
        file.setId(Long.parseLong(id));

        FileService fileService = (FileService)
            getServletContext().getAttribute("fileService");
        file = fileService.getFile(file); ← ❶ 根据 id 取得文件
        String filename = null;
        if(request.getHeader("User-Agent").contains("MSIE")) {
            filename = URLEncoder.encode(file.getFilename(), "UTF-8");
        }
        else {
            filename = new String(
                file.getFilename().getBytes("UTF-8"), "ISO-8859-1");
        }

        response.setContentType("application/octet-stream"); ← ❷ 告知浏览器响应类型
        response.setHeader("Content-disposition",
                           "attachment; filename=\"" + filename + "\""); ← ❸ 这个标头会告知浏览器另存为新文件的文件名
        OutputStream out = response.getOutputStream();
        out.write(file.getBytes());
        out.close();
    }
}

```

浏览器会告知想要下载的文件 `id` 是什么，所以 Servlet 中取得 `id` 请求参数，封装为 `File` 对象，调用 `FileService` 的 `getFile()` 取得 `File` 对象①，从中取得文件名与字节。为了让浏览器出现另存为的对话框，必须告知浏览器响应类型为 "application/octet-stream"④，也就是十六进制串流数据，并使用 "Content-disposition" 告知另存新文件时默认的文件名⑤。不过这个文件名的编码会因 Internet Explorer 或其他浏览器在处理上有所不同。Internet Explorer 必须作 URL 编码②，而其他浏览器必须以 ISO-8859-1 编码③。另存新文件时，才可以正确显示中文文件名。

提示» "Content-disposition" 在文件名编码上，这边的范例测试过 Firefox 4、Google Chrome 与 Internet Explorer 9，在下载时可以正确地出现默认的中文文件名。

处理文件删除的 Servlet 如下：

JDBCDelete.java

```
package cc.openhome;

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

@WebServlet("/delete.do")
public class Delete extends HttpServlet {
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws ServletException, IOException {
        String id = request.getParameter("id");
        File file = new File();
        file.setId(Long.parseLong(id));
        FileService fileService = (FileService)
            getServletContext().getAttribute("fileService");
        fileService.delete(file);
        response.sendRedirect("file.jsp");
    }
}
```

这个 Servlet 很简单，删除文件时也是根据 `id`，在封装为 `File` 对象之后，调用 `FileService` 的 `delete()` 即可删除文件。一个执行时的参考画面如图 9.12 所示。

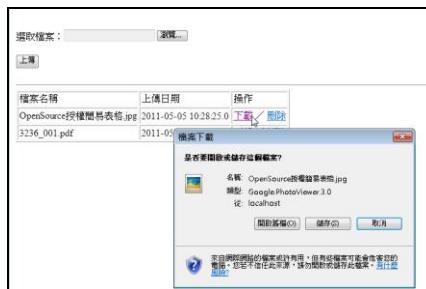


图 9.12 文件上传、下载、删除的简易管理页面

9.2.5 事务简介

事务的四个基本要求是原子性(Atomicity)、一致性(Consistency)、隔离行为(Isolation behavior)与持续性(Durability)，依英文字母首字简称为 ACID。

■ 原子性

一个事务是一个单元工作(Unit of work)，当中可能包括数个步骤，这些步骤必须全部执行成功，若有一个失败，则整个事务声明失败，事务中其他步骤必须撤销曾经执行过的动作，回到事务前的状态。

在数据库上执行单元工作为数据库事务(Database transaction)，单元中每个步骤就是每一句 SQL 的执行。要开始一个事务边界(通常是以一个 BEGIN 的命令开始)，所有 SQL 语句下达之后，COMMIT 确认所有操作变更，此时事务成功，或者因为某个 SQL 错误，ROLLBACK 进行撤销动作，此时事务失败。

■ 一致性

事务作用的数据集合在事务前后必须一致，若事务成功，整个数据集合都必须是事务操作后的状态；若事务失败，整个数据集合必须与开始事务前一样没有变更，不能发生整个数据集合部分有变更，部分没变更的状态。

例如转账行为，数据集合涉及 A、B 两个账户，A 原有 20 000 元，B 原有 10 000 元，A 转 10 000 元给 B，事务成功的话，最后 A 必须变成 10 000 元，B 变成 20 000 元，事务失败的话，A 必须为 20 000 元，B 为 10 000 元，而不能发生 A 为 20 000 元(未扣款)，B 也为 20 000 元(已入款)的情况。

■ 隔离行为

在多人使用的环境下，每个用户可能进行自己的事务，事务与事务之间，必须互不干扰，用户不会意识到别的用户正在进行事务，就好像只有自己在进行操作一样。

■ 持续性

事务一旦成功，所有变更必须保存下来，即使系统故障，事务的结果也不能遗失。这通常需要系统软、硬件架构的支持。

在原子性的要求上，在 JDBC 可以操作 connection 的 `setAutoCommit()` 方法，给它 `false` 自变量，提示数据库启始事务，在下达一连串的 SQL 命令后，自行调用 Connection 的 `commit()`，提示数据库确认(COMMIT)操作。如果中间发生错误，则调用 `rollback()`，提示数据库撤销(ROLLBACK)所有的执行。一个示范的流程如下所示：

```
Connection conn = null;
```

```
try {
    conn = dataSource.getConnection();
    conn.setAutoCommit(false); // 取消自动提交
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO ...");
    stmt.executeUpdate("INSERT INTO ...");
    conn.commit(); // 提交
}
catch(SQLException e) {
    e.printStackTrace();
    if(conn != null) {
        try {
            conn.rollback(); // 回滚
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
finally {
    ...
    if(conn != null) {
        try {
            conn.setAutoCommit(true); // 回复自动提交
            conn.close();
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
```

如果在事务管理时，仅想要撤回某个 SQL 执行点，则可以设置储存点(Save point)。例如：

```
Savepoint point = null;
try {
    conn.setAutoCommit(false);
    Statement stmt = conn.createStatement();
    stmt.executeUpdate("INSERT INTO ...");
    ...
    point = conn.setSavepoint(); // 设置储存点
    stmt.executeUpdate("INSERT INTO ...");
    ...
    conn.commit();
}
catch(SQLException e) {
    e.printStackTrace();
    if(conn != null) {
        try {
            if(point == null) {
                conn.rollback();
            }
        }
```

```

        else {
            conn.rollback(point);           // 撤回储存点
            conn.releaseSavepoint(point); // 释放储存点
        }
    }
    catch(SQLException ex) {
        ex.printStackTrace();
    }
}
finally {
    ...
    if(conn != null) {
        try {
            conn.setAutoCommit(true);
            conn.close();
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
}

```

在批次更新时，不用每一笔都确认的话，也可以搭配事务管理。例如：

```

try {
    conn.setAutoCommit(false);
    stmt = conn.createStatement();
    while(someCondition) {
        stmt.addBatch("INSERT INTO ...");
    }
    stmt.executeBatch();
    conn.commit();
} catch(SQLException ex) {
    ex.printStackTrace();
    if(conn != null) {
        try {
            conn.rollback();
        } catch(SQLException e) {
            e.printStackTrace();
        }
    }
}
finally {
    ...
    if(conn != null) {
        try {
            conn.setAutoCommit(true);
            conn.close();
        }
        catch(SQLException ex) {
            ex.printStackTrace();
        }
    }
}
}

```

提示» 数据表格必须支持事务，才可以执行以上所提到的功能。例如，在 MySQL 中可以创建 InnoDB 类型的表格：

```
CREATE TABLE t_xxx (
    ...
) Type = InnoDB;
```

至于在隔离行为的支持上，JDBC 可以通过 Connection 的 `getTransactionIsolation()` 取得数据库目前的隔离行为设置，通过 `setTransactionIsolation()` 可提示数据库设置指定的隔离行为。可设置常数是定义在 Connection 上的，如下所示：

- TRANSACTION_NONE
- TRANSACTION_UNCOMMITTED
- TRANSACTION_COMMITTED
- TRANSACTION_REPEATABLE_READ
- TRANSACTION_SERIALIZABLE

其中 `TRANSACTION_NONE` 表示对事务不设置隔离行为，仅适用于没有事务功能、以只读功能为主、不会发生同时修改字段的数据库。有事务功能的数据库，可能不理睬 `TRANSACTION_NONE` 的设置提示。

要了解其他隔离行为设置的影响，首先要了解多个事务并行时，可能引发的数据不一致问题有哪些。以下逐一举例说明。

⌚ 更新遗失(Lost update)

基本上就是指某个事务对字段进行更新的信息，因另一个事务的介入而遗失更新效力。举例来说，若某个字段数据原为 ZZZ，用户 A、B 分别在不同的时间点对同一字段进行更新事务，如图 9.13 所示。

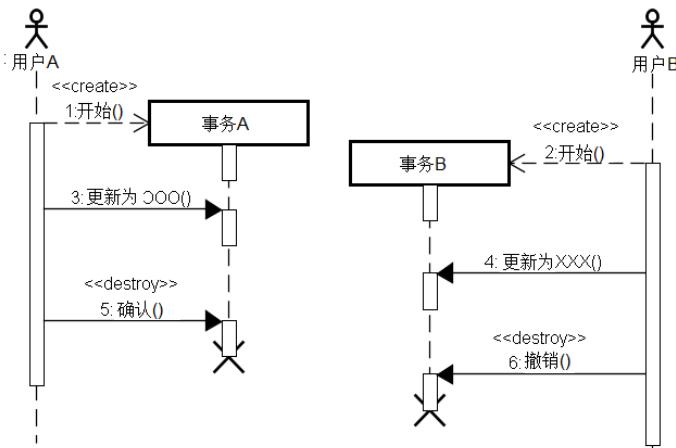


图 9.13 更新遗失

单就用户 A 的事务而言，最后字段应该是 OOO，单就用户 B 的事务而言，最后字段应该是 ZZZ。在完全没有隔离两者事务的情况下，由于用户 B 撤销操作时间在用户 A 确认之后，因此最后字段结果会是 ZZZ，用户 A 看不到他更新确认的 OOO 结果，用户 A 发生更新遗失问题。

提示»» 可想象有两个用户，若 A 用户打开文件之后，后续又允许 B 用户打开文件，一开始 A、B 用户看到的文件都有 ZZZ 文字，A 修改 ZZZ 为 OOO 后储存，B 修改 ZZZ 为 XXX 后又还原为 ZZZ 并储存，最后文件就为 ZZZ，A 用户的更新遗失。

如果要避免更新遗失问题，可以设置隔离层级为“可读取未确认”(Read uncommitted)，也就是 A 事务已更新但未确认的数据，B 事务仅可作读取动作，但不可作更新的动作。JDBC 可通过 Connection 的 setTransactionIsolation() 设置为 TRANSACTION_UNCOMMITTED 来提示数据库指定此隔离行为。

数据库对此隔离行为的基本做法是，A 事务在更新但未确认，延后 B 事务的更新需求至 A 事务确认之后。以上例而言，事务顺序结果会变成图 9.14 所示。

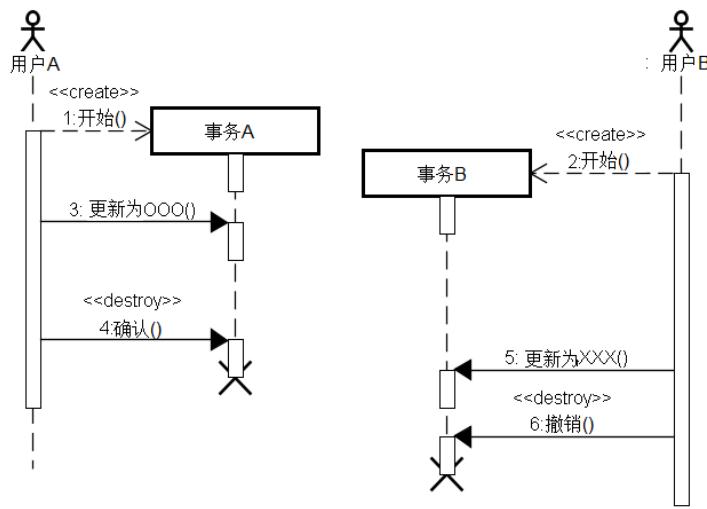


图 9.14 “可读取未确认”避免更新遗失

提示»» 可想象有两个用户，若 A 用户打开文件之后，后续只允许 B 用户以只读方式打开文件，B 用户若要能够写入，至少得等 A 用户修改完成关闭文件后。

提示数据库“可读取未确认”的隔离层次之后，数据库至少得保证事务能避免更新遗失问题，通常这也是具备事务功能的数据库引擎会采取的最低隔离层级。不过这个隔离层级读取错误数据的机率太高，一般默认不会采用这种隔离层级。

▶ 脏读(Dirty read)

两个事务同时进行，其中一个事务更新数据但未确认，另一个事务就读取数据，就有可能发生脏读问题，也就是读到所谓脏数据、不干净、不正确的数据，如图 9.15 所示。

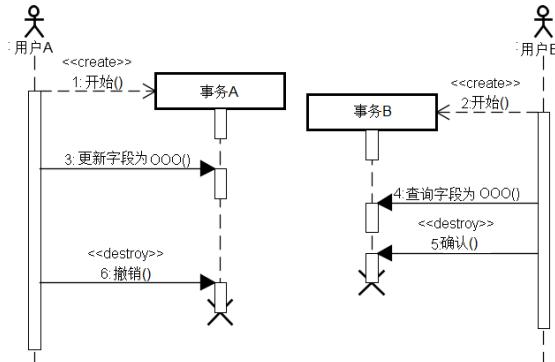


图 9.15 脏读

用户 B 在 A 事务撤销前读取了字段数据为 OOO，如果 A 事务撤销了事务，那么用户 B 读取的数据就是不正确的。

提示» 可想象有两个用户，若 A 用户打开文件并仍在修改期间，B 用户打开文件所读到的数据，就有可能是不正确的。

如果要避免脏读问题，可以设置隔离层级为“可读取确认”(Read committed)，也就是事务读取的数据必须是其他事务已确认的数据。JDBC 可通过 `Connection` 的 `setTransactionIsolation()` 设置为 `TRANSACTION_COMMITTED` 来提示数据库指定此隔离行为。

数据库对此隔离行为的基本做法之一是，读取的事务不会阻止其他事务，未确认的更新事务会阻止其他事务。若是这个他法，事务顺序结果会变成图 9.16 所示(若原字段为 ZZZ)。

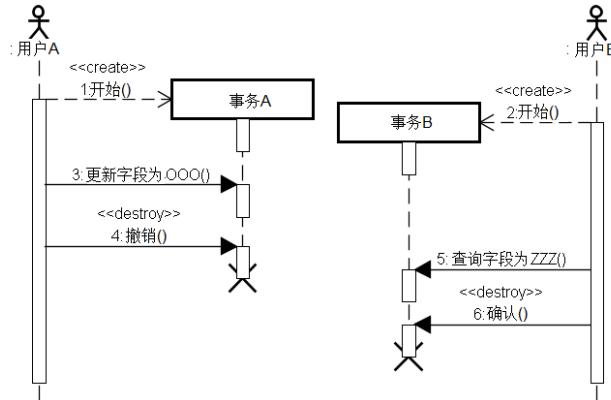


图 9.16 “可读取确认”避免脏读

提示» 可想象有两个用户，若 A 用户打开文件并仍在修改期间，B 用户就不能打开文件。但在数据库上这个做法影响性能较大。另一个基本做法是事务正在更新但尚未确定前先操作暂存表格，其他事务就不至于读取到不正确的数据。JDBC 隔离层级的设置提示，实际在数据库上如何实现，主要得根据各家数据库在性能上的考量而定。

提示数据库“可读取确认”的隔离层次之后，数据库至少得保证事务能避免脏读与更新遗失问题。

无法重复的读取(Unrepeatable read)

某个事务两次读取同一字段的数据并不一致。例如，事务 A 在事务 B 更新前后进行数据的读取，则 A 事务会得到不同的结果，如图 9.17 所示(若字段原为 ZZZ)。

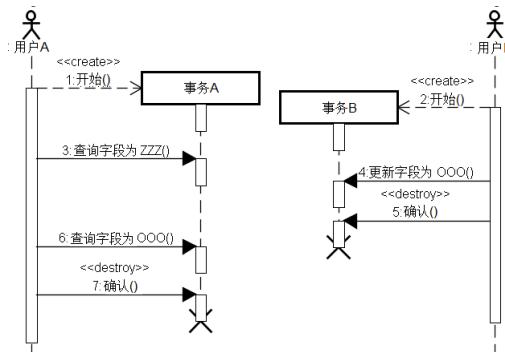


图 9.17 无法重复的读取

如果要避免无法重复的读取问题，可以设置隔离层级为“可重复读取”(Repeatable read)，也就是同一事务内两次读取的数据必须相同。JDBC 可通过 Connection 的 `setTransactionIsolation()` 设置为 `TRANSACTION_REPEATABLE_READ` 来提示数据库指定此隔离行为。

数据库对此隔离行为的基本做法之一是，读取事务在确认前不阻止其他读取事务，但会阻止其他更新事务。若是这个做法，事务顺序结果会变成图 9.18 所示(若原字段为 ZZZ)。

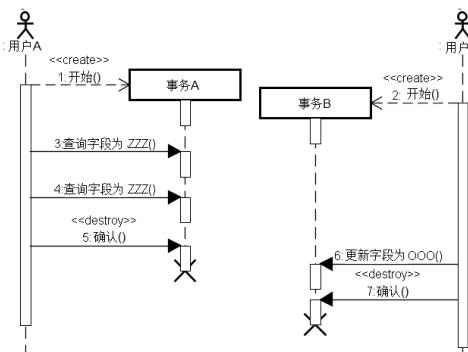


图 9.18 可重复读取

提示» 在数据库上这个做法影响性能较大,另一个基本做法是事务正在读取但尚未确认前,另一事务会在暂存表格上更新。

提示数据库“可重复读取”的隔离层次之后,数据库至少得保证事务能避免无法重复读取、脏读与更新遗失问题。

⌚ 幻读(Phantom read)

同一事务期间,读取到的数据笔数不一致。例如,事务A第一次读取得到五笔数据,此时事务B新增了一笔数据,导致事务B再次读取得到六笔数据。

如果隔离行为设置为可重复读取,但发生幻读现象,可以设置隔离层级为“可循序”(Serializable),也就是在有事务时若有数据不一致的疑虑,事务必须可以按照顺序逐一进行。JDBC可通过Connection的setTransactionIsolation()设置为TRANSACTION_SERIALIZABLE来提示数据库指定此隔离行为。

提示» 事务若真的一个一个循序进行,对数据库的影响性能过于巨大,实际也许未必直接阻止其他事务或真的循序进行,例如采用暂存表格方式。事实上,只要能符合四个事务隔离要求,各家数据库会寻求最有性能的解决方式。

表 9.2 整理了各个隔离行为可预防的问题。

表 9.2 隔离行为与可预防的问题

| 隔 离 行 为 | 更 新 遗 失 | 脏 读 | 无 法 重 复 的 读 取 | 幻 读 |
|---------|---------|-----|---------------|-----|
| 可读取未确认 | 预防 | | | |
| 可读取确认 | 预防 | 预防 | | |
| 可重复读取 | 预防 | 预防 | 预防 | |
| 可循序 | 预防 | 预防 | 预防 | 预防 |

如果想通过 JDBC 得知数据库是否支持某个隔离行为设置,可以通过 Connection 的 getMetaData() 取得 DatabaseMetadata 对象,通过 DatabaseMetadata 的 supportsTransactionIsolationLevel() 得知是否支持某个隔离行为。例如:

```
DatabaseMetadata meta = conn.getMetaData();
boolean isSupported = meta.supportsTransactionIsolationLevel(
    Connection.TRANSACTION_READ_COMMITTED);
```

9.2.6 metadata 简介

Metadata 即“诠读数据的数据”(Data about data),如这个数据库是用来保存数据的地方,然而数据库本身产品名称是什么?数据库中有几个数据表格?表格名称是什么?表格中有几个字段等?这些信息就是所谓 metadata。

在 JDBC 中，可以通过 `Connection` 的 `getMetaData()` 方法取得 `DatabaseMetaData` 对象，通过这个对象提供的种种方法，可以取得数据库整体信息，而 `ResultSet` 表示查询到的数据，而数据本身的字段、类型等信息，则可以通过 `ResultSet` 的 `getMetaData()` 方法，取得 `ResultSetMetaData` 对象，通过这个对象提供的相关方法，就可以取得字段名称、字段类型等信息。

提示» `DatabaseMetaData` 或 `ResultSetMetaData` 本身 API 使用上不难，问题点在于各家数据库对某些名词的定义不同，必须查阅数据库厂商手册搭配对应的 API，才可以取得想要的信息。

下面举个例子，利用 JDBC 的 metadata 相关 API 取得先前文件管理范例 `t_files` 表格相关信息。首先定义一个 JavaBean：

JDBC Demo TFileInfo.java

```
package cc.openhome;

import java.io.Serializable;
import java.sql.*;
import java.util.*;
import javax.naming.*;
import javax.sql.DataSource;

public class TFileInfo implements Serializable {
    private DataSource dataSource;

    public TFileInfo() {
        try {
            Context initContext = new InitialContext();
            Context envContext = (Context)
                initContext.lookup("java:/comp/env");
            dataSource = (DataSource) envContext.lookup("jdbc/demo");
        } catch (NamingException ex) {
            throw new RuntimeException(ex);
        }
    }

    public List<ColumnInfo> getAllColumnInfo() {
        Connection conn = null;
        ResultSet crs = null;
        SQLException ex = null;
        List<ColumnInfo> infos = null;
        try {
            conn = dataSource.getConnection();
            DatabaseMetaData meta = conn.getMetaData();
            crs = meta.getColumns(
                "demo", null, "t_files", null); ① 查询 t_files 表格所有字段
            infos = new ArrayList<ColumnInfo>(); ② 用来收集字段信息
            while(crs.next()) {
                ColumnInfo info = new ColumnInfo();
                info.setName(crs.getString("COLUMN_NAME"));
            }
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            if (crs != null) {
                try {
                    crs.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

```

        info.setType(crs.getString("TYPE_NAME"));
        info.setSize(crs.getInt("COLUMN_SIZE"));
        info.setNullable(crs.getBoolean("IS_NULLABLE"));
        info.setDef(crs.getString("COLUMN_DEF"));
        infos.add(info);
    }
} catch (SQLException e) {
    ex = e;
}
finally {
    if(conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {
            if(ex == null) {
                ex = e;
            }
        }
    }
}
if(ex != null) {
    throw new RuntimeException(ex);
}

return infos;
}
}

```

在调用 `getColumnInfo()` 时，会先从 `Connection` 上取得 `DatabaseMetaData`，以查询数据库中指定表格的字段①，这会取得一个 `ResultSet`。接着从 `ResultSet` 上逐一取得各个想要的信息，封装为 `ColumnInfo` 对象③，并收集在 `List` 中返回②。

接着编写一个 JSP 页面来使用 `TFileInfo` 类：

JDBC Demo metadata.jsp

```

<%@page contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<jsp:useBean id="tFileInfo" class="cc.openhome.TFilesInfo" /> ① 以 JavaBean
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"      方式使用
    "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type"
            content="text/html; charset=UTF-8">
        <title>Metadata</title>
    </head>
    <body>
        <table style="text-align: left;" border="1"
            cellpadding="2" cellspacing="2">
            <tbody>
                <tr>

```

```

<td>字段名称</td>
<td>字段类型</td>
<td>可否为空</td>
<td>默认数值</td>
</tr>
<c:forEach var="columnInfo"
           items="${tFileInfo.allColumnInfo}"
           >
<tr>
    <td>${columnInfo.name}</td>
    <td>${columnInfo.type}</td>
    <td>${columnInfo.nullable}</td>
    <td>${columnInfo.def} </td>
</tr>
</c:forEach>
</tbody>
</table>
</body>
</html>

```

② 取得所有字段
信息并显示

为了简化范例，在这里将 `tFileInfo` 当作 JavaBean 来使用，并利用 JSTL 的 `<c:forEach>` 逐一取得 `ColumnInfo` 对象，以表格方式显示字段信息。一个参考画面如图 9.19 所示。



The screenshot shows a table with four columns: 字段名称 (Field Name), 字段类型 (Field Type), 可否为空 (Nullability), and 默认数值 (Default Value). The table contains five rows of data:

| 字段名称 | 字段类型 | 可否为空 | 默认数值 |
|------------------------|-----------|-------|-------------------|
| <code>id</code> | INT | false | |
| <code>filename</code> | VARCHAR | false | |
| <code>savedTime</code> | TIMESTAMP | false | CURRENT_TIMESTAMP |
| <code>bytes</code> | LONGBLOB | false | |

图 9.19 取得字段基本信息

9.2.7 RowSet 简介

JDBC 定义了 `javax.sql.RowSet` 接口，用以代表数据的列集合。这里的数据并不一定是数据库中的数据，可以是试算表数据、XML 数据或任何具有行集合概念的数据源。

`RowSet` 是 `ResultSet` 的子接口，所以具有 `ResultSet` 的行为，可以使用 `RowSet` 对行集合进行增删查改，`RowSet` 也新增了一些行为，如通过 `setCommand()` 设置查询命令、通过 `execute()` 执行查询命令以填充数据等。

提示»» 在 Sun 的 JDK 中附有 `RowSet` 的非标准实现，其包名称是 `com.sun.rowset`。

`RowSet` 定义了行集合基本行为，其下有 `JdbcRowSet`、`CachedRowSet`、`FilteredRowSet`、`JoinRowSet` 与 `WebRowSet` 五个标准行集合子接口，定义在 `javax.sql.rowset` 包中。其继承关系如图 9.20 所示。

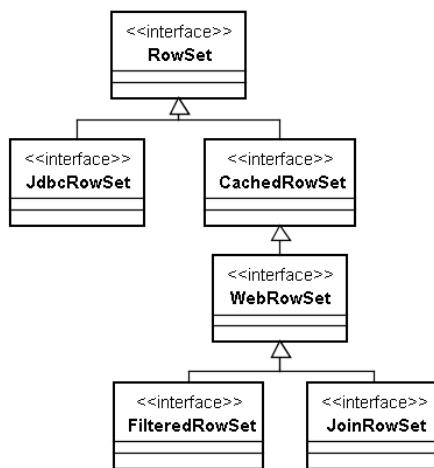


图 9.20 RowSet 接口继承架构

JdbcRowSet 是连接式(Connected)的 RowSet，也就是操作 JdbcRowSet 期间，会保持与数据库的连接，可视为取得、操作 ResultSet 的行为封装，可简化 JDBC 程序的编写，或作为 JavaBean 使用。

CachedRowSet 则为离线式(Disconnected)的 RowSet(其子接口当然也是)，在查询并填充完数据后，就会断开与数据源的连接，而不用占据相关连接资源，必要时也可以再与数据源连接进行数据同步。

以下先以 JdbcRowSet 为例，介绍 RowSet 的基本操作。在这里使用的实现是 Sun JDK 附带的 JdbcRowSetImpl。要使用 RowSet 查询数据，基本上可以如下：

```

JdbcRowSet rowset = new JdbcRowSetImpl();
rowset.setUrl("jdbc:mysql://localhost:3306/demo");
rowset.setUsername("root");
rowset.setPassword("123456");
rowset.setCommand("SELECT * FROM t_messages WHERE id = ?");
rowset.setInt(1, 1);
rowset.execute();
  
```

可以使用 `setUrl()` 设置 JDBC URL，使用 `setUsername()` 设置用户名，使用 `setPassword()` 设置密码，使用 `setCommand()` 设置查询 SQL。

由于 RowSet 是 ResultSet 的子接口，接下来要取得各字段数据，只要如 ResultSet 操作即可。若要使用 RowsSet 进行增删改的动作，也是与 ResultSet 相同。例如，下例使用 JdbcRowSet 改写 9.1.3 节的访客留言板，可以比较使用 JdbcRowSet 之后的差别：

JDBC Demo GuestBookBean2.java

```

package cc.openhome;

import java.io.Serializable;
import java.sql.*;
import java.util.*;
import javax.sql.rowset.JdbcRowSet;
  
```

```

import com.sun.rowset.JdbcRowSetImpl;

public class GuestBookBean2 implements Serializable {
    private JdbcRowSet rowset;
    public GuestBookBean2() throws SQLException {
        rowset = new JdbcRowSetImpl();
        rowset.setDataSourceName("java:/comp/env/jdbc/demo");
        rowset.setCommand("SELECT * FROM t_message");
        rowset.execute();
    }

    public void setMessage(Message message) throws SQLException {
        rowset.moveToInsertRow();
        rowset.updateString(2, message.getName());
        rowset.updateString(3, message.getEmail());
        rowset.updateString(4, message.getMsg());
        rowset.insertRow();
    }

    public List<Message> getMessages() throws SQLException {
        List<Message> messages = new ArrayList<Message>();
        rowset.beforeFirst();
        while (rowset.next()) {
            Message message = new Message();
            message.setId(rowset.getLong(1));
            message.setName(rowset.getString(2));
            message.setEmail(rowset.getString(3));
            message.setMsg(rowset.getString(4));
            messages.add(message);
        }
        return messages;
    }

    @Override
    protected void finalize() throws Throwable {
        if (rowset != null) {
            rowset.close();
        }
    }
}

```

在这个例子中，使用 `setDataSourceName()` 来取得 `DataSource` 并直接利用 `JdbcRowSet` 进行查询与新增留言的操作。`JdbcRowSet` 也有 `setAutocommit()` 与 `commit()` 方法，可以进行事务控制。

如果在查询之后，想要离线进行操作，则可以使用 `CachedRowSet` 或其子接口实现对象。视需求而定，可以直接使用 `close()` 关闭 `CachedRowSet`，若在相关更新操作之后，想再与数据源进行同步，则可以调用 `acceptChanges()` 方法。例如：

```

conn.setAutoCommit(false); // conn 是 Connection
rowSet.acceptChanges(conn); // rowSet 是 CachedRowSet

```

```
conn.setAutoCommit(true);
```

WebRowSet 是 **CachedRowSet** 的子接口，其不仅具备离线操作，还能进行 XML 读写。例如以下的 Servlet，可以读取数据库的表格数据，然后对客户端写出 XML：

JDBC Demo XMLMessage.java

```
package cc.openhome;

import java.io.IOException;
import java.sql.SQLException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import javax.sql.rowset.WebRowSet;
import com.sun.rowset.WebRowSetImpl;

@WebServlet("/xmlMessage")
public class XMLMessage extends HttpServlet {
    private WebRowSet rowset = null;

    @Override
    public void init() throws ServletException {
        try {
            rowset = new WebRowSetImpl();
            rowset.setDataSourceName("java:/comp/env/jdbc/demo");
            rowset.setCommand("SELECT * FROM t_message");
            rowset.execute();
        } catch (SQLException e) {
            throw new ServletException(e);
        }
    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/xml;charset=UTF-8");
        try {
            rowset.writeXml(response.getOutputStream());
        } catch (SQLException e) {
            throw new ServletException(e);
        }
    }
}
```

使用 **WebRowSet** 的 **writeXML()**，可以将 **WebRowSet** 的 **Metadata**、属性与数据以 XML 格式写出。一个执行结果如图 9.21 所示。

```

<?xml version="1.0" encoding="UTF-8"?>
<data>
    - <currentRow>
        <columnValue>1</columnValue>
        <columnValue>良葛格</columnValue>
        <columnValue>caterpillar@openhome.cc</columnValue>
        <columnValue>这是一篇测试留言板！</columnValue>
    </currentRow>
    - <currentRow>
        <columnValue>2</columnValue>
        <columnValue>毛美眉</columnValue>
        <columnValue>momo@openhome.cc</columnValue>
        <columnValue>这是简单的测试留言板！</columnValue>
    </currentRow>
    - <currentRow>
        <columnValue>3</columnValue>
        <columnValue>哈小米</columnValue>
        <columnValue>hamimi@openhome.cc</columnValue>
        <columnValue>采用JavaBean+JSP+JSTL！</columnValue>
    </currentRow>
</data>

```

图 9.21 WebRowSet 写出的 XML 文件数据区段

这让你不用进行烦琐的 XML 操作，就可以将查询的数据以 XML 的方式写出。例如，其他网站取得 XML 之后，可以使用 JSTL 的 XML 格式标签库组织画面：

```

JDCCDemo xmlMessage.jsp
<%@ page contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions"%>
<%@taglib prefix="x" uri="http://java.sun.com/jsp/jstl/xml"%>
<!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>友站的留言</title>
    </head>
    <body>
        <c:import var="xml" url="xmlMessage" charEncoding="UTF-8" />
        <c:set var='xmlns'>
            xmlns="http://java.sun.com/xml/ns/jdbc"
        </c:set>

        <!-- JSTL 1.1 不支持 XML Namespace，所以用空字符串取代掉 -->
        <x:parse var="webRowSet" doc="${fn:replace(xml, xmlns, '')}"/>
        <h2>友站的留言</h2>
        <table border="1">
            <tr bgcolor="#00ff00">
                <th align="left">名称</th>
                <th align="left">邮件</th>
                <th align="left">留言</th>
            </tr>
            <x:forEach var="row" select="$webRowSet//currentRow">
                <tr>
                    <td><x:out select="$row/columnValue[2]" /></td>

```

```

<td><x:out select="$row/columnValue[3]"/></td>
<td><x:out select="$row/columnValue[4]"/></td>
</tr>
</x:forEach>
</table>
</body>
</html>

```

在这里要注意的是，由于 JSTL 1.1 不支持 XML 名称空间，所以使用 EL 函数库中的 \${fn:replace()} 函数，将名称空间部分的字符串取代为空字符串。一个范例执行的参考画面如图 9.22 所示。

| 友站的留言 | | |
|-------|-------------------------|------------------------|
| 名称 | 邮件 | 留言 |
| 良葛格 | caterpillar@openhome.cc | 这是一篇测试留言！ |
| 毛美眉 | momor@openhome.cc | 这是简单的测试留言板！ |
| 哈小米 | hamimi@openhome.cc | 采用 JavaBean+JSP+JSTL ! |

图 9.22 从另一站读入 XML 并显示

FilteredRowSet 可以对行集合进行过滤，实现类似 SQL 中 WHERE 等条件式的功能。可以通过 **setFilter()** 方法，指定实现 **javax.sql.rowset.Predicate** 的对象。其定义如下：

```

boolean evaluate(Object value, int column)
boolean evaluate(Object value, String columnName)
boolean evaluate(RowSet rs)

```

Predicate 的 **evaluate()** 方法返回 **true**，表示该行要包括在过滤后的行集合中。

JoinRowSet 则可以让你结合两个 RowSet 对象，实现类似 SQL 中 JOIN 的功能。可以通过 **setMatchColumn()** 指定要结合的列，然后使用 **addRowSet()** 来加入 RowSet 进行结合。例如：

```

rs1.setMatchColumn(1);
rs2.setMatchColumn(2);
JoinRowSet jrs = JoinRowSet jrs = new JoinRowSetImpl();
jrs.addRowSet(rs1);
jrs.addRowSet(rs2);

```

在这个范例片段执行过后，JoinRowSet 中就会是原本两个 RowSet 结合的结果。也可以通过 **setJoinType()** 指定结合的方式，可指定的常数定义在 JoinRowSet 中，包括 **CROSS_JOIN**、**FULL_JOIN**、**INNER_JOIN**、**LEFT_OUTER_JOIN** 与 **RIGHT_OUTER_JOIN**。

提示»» API 文件对 RowSet 的文件说明是很清楚的，更多有关 RowSet 的说明，也可以参考：

<http://java.sun.com/developer/Books/JDBCTutorial/chapter5.html>

9.3 使用 SQL 标签库

JSTL 提供了 SQL 标签库，可以在 JSP 页面上直接进行数据库增删查找，但无须编写任何 JDBC 代码。对于不复杂的数据库操作，使用 SQL 标签库对于应用程序可以有一定程度的简化。

9.3.1 数据源、查询标签

若要使用 JSTL 的 XML 标签库，必须使用 `taglib` 指示元素进行定义：

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

在进行任何数据库来源之前，得先设置数据源(Data source)。对 JDBC 而言就是设置连接来源，这可以使用 `<sql:setDataSource>` 标签来设置。例如：

```
<sql:setDataSource dataSource="java:/comp/env/jdbc/demo"/>
```

`dataSource` 属性可以是 JNDI 字符串名称或 `DataSource` 实例，或者是直接设置驱动程序类、用户名、密码与 JDBC URL：

```
<sql:setDataSource driver="com.mysql.jdbc.Driver"
                  user="root" password="123456"
                  url="jdbc:mysql://localhost:3306/demo"/>
```

如果要进行数据库查询，可以使用 `<sql:query>` 标签。如果已经使用 `<sql:setDataSource>` 设置数据源，则可以直接进行 SQL 查询：

```
<sql:query sql="SELECT * FROM t_message" var="messages"/>
```

如果属性范围内已经存在 `DataSource`，也可以直接使用 `<sql:query>` 的 `dataSource` 属性来指定。如果 SQL 语句比较复杂，也可以直接编写在标签 Body 中。例如：

```
<sql:query dataSource="${dataSource}" var="messages">
    SELECT * FROM t_message
</sql:query>
```

`<sql:query>` 还有 `startRow` 属性可以指定查询结果的第几笔取得查询结果，`maxRows` 属性可以指定取得几笔结果。`<sql:query>` 的查询结果是 `javax.servlet.jsp.jstl.sql.Result` 类型，具有 `getColumnName()`、`getRowCount()`、`getRows()` 等方法，可配合 JSTL 的 `<c:forEach>` 来取出每一笔数据。例如：

```
<sql:query sql="SELECT * FROM t_message" var="messages"/>
<c:forEach var="message" items="${messages.rows}">
    ${message.name}<br>
    ${message.email}<br>
    ${message.msg}
</c:forEach>
```

`javax.servlet.jsp.jstl.sql.Result` 也有 `getRowsByIndex()` 方法，可以 `Object[][]` 返回查询数据，所以也可根据索引取得字段数据：

```
<sql:query sql="SELECT * FROM t_message" var="messages"/>
```

```
<c:forEach var="message" items="${messages.rowsByIndex}">
    ${message[0]}<br>
    ${message[1]}<br>
    ${message[2]}
</c:forEach>
```

提示» 由于 `getRowsByIndex()` 返回的是 `Object[][]`，所以索引要从 0 开始。

9.3.2 更新、参数、事务标签

如果想通过 SQL 标签库对数据库进行更新动作，则可以使用 `<sql:update>` 标签。例如，要在数据库中新增一笔数据：

```
<sql:update>
    INSERT INTO t_message(name, email, msg)
        VALUES('Justin', 'caterpillar@openhome.cc', 'This is a test!')
</sql:update>
```

如果 SQL 中有部分数据是未定的，例如，可能来自请求参数数据，则以下写法虽可以但不建议：

```
<sql:update>
    INSERT INTO t_message(name, email, msg)
        VALUES(${param.user}, ${param.email}, ${param.msg})
</sql:update>
```

正如 9.1.4 节提过的，直接将请求参数的值未经过滤就安插在 SQL 中，可能会隐含 SQL Injection 的安全问题。可以在 SQL 中使用占位字符，并搭配 `<sql:param>` 标签来设置占位字符的值。例如：

```
<sql:update>
    INSERT INTO t_message(name, email, msg) VALUES(?, ?, ?)
    <sql:param value ="${param.name}"/>
    <sql:param value ="${param.email}"/>
    <sql:param value ="${param.msg}"/>
</sql:update>
```

如果字段是日期时间格式，则可以使用 `<sql:paramDate>` 标签，可以通过 `type` 属性设置，指定使用 "time"、"date" 或 "timestamp" 的值。`<sql:param>`、`<sql:paramDate>` 也可以搭配 `<sql:query>` 使用。

如果有必要指定事务隔离行为，则可以通过 `<sql:transaction>` 标签指定，设置 `isolation` 属性为 "read_uncommitted"、"read_committed"、"repeatable" 或 "serializable" 来指定不同的事务隔离行为。



下面这个程序改写 7.2.1 节的留言板范例，使用纯 JSP 与 SQL 标签库来完成相同的功能：

JDBC Demo guestbook3.jsp

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
```

```

<%@taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@taglib prefix="sql" uri="http://java.sun.com/jsp/jstl/sql"%>
<sql:setDataSource dataSource="jdbc/demo"/>
<c:set target="${pageContext.request}"%
       property="characterEncoding" value="UTF-8"/>
<c:if test="${param.msg != null}">
    <sql:update>
        INSERT INTO t_message(name, email, msg) VALUES (?, ?, ?)
        <sql:param value="${param.name}"/>
        <sql:param value="${param.email}"/>
        <sql:param value="${param.msg}"/>
    </sql:update>
</c:if>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
        <meta http-equiv="Content-Type"
              content="text/html; charset=UTF-8">
        <title>访客留言板</title>
    </head>
    <body>
        <table style="text-align: left; width: 100%;" border="0"
               cellpadding="2" cellspacing="2">
            <tbody>
                <sql:query sql="SELECT name, email, msg FROM t_message"
                           var="messages"/>
                <c:forEach var="message" items="${messages.rows}">
                    <tr>
                        <td>${message.name}</td>
                        <td>${message.email}</td>
                        <td>${message.msg}</td>
                    </tr>
                </c:forEach>
            </tbody>
        </table>
    </body>
</html>

```

9.4 综合练习 / 微博

先前的微博综合练习，都是直接使用文件来储存相关信息，在这一节中，将改用数据库搭配 JDBC 存取数据。不过将文件储存改为数据库储存，就目前应用程序来说，是个不小的变动。因此在这里将导入 DAO(Data Access Object)设计模式，以隔离储存逻辑与业务逻辑。

9.4.1 重构 / 使用 DAO

如果观察一下目前微博应用程序的 `UserService` 类，会发现其中充斥着大量的文件输入输出代码，`UserService` 中检查用户是否存在、确认用户可否登录、用户信息的排序等业务逻辑，混杂在文件输入输出代码中而不易维护。

文件输入输出是一种储存逻辑，将储存逻辑与业务逻辑混杂在一起，缺点就是不易维护。对本书造成最直接的冲击就是，若尝试在现有架构下，将文件输入输出转换为 JDBC，改写上极为麻烦且容易出错。

在正式 `UserService` 中的储存逻辑改写为 JDBC 前，要对应用程序进行重构。先别考虑增加新的功能，而是在隔离储存逻辑与业务逻辑之后，让应用程序仍可以利用文件输入输出来存取数据。

在这里先定义出用户账户的储存行为，包括确认用户是否存在、新增用户与取得用户数据：

Gossip AccountDAO.java

```
package cc.openhome.model;
public interface AccountDAO {
    boolean isUserExisted(Account account);
    void addAccount(Account account);
    Account getAccount(Account account);
}
```

而信息的列表取得、新增信息与删除信息，则定义在另一个接口中：

Gossip BlahDAO.java

```
package cc.openhome.model;
import java.util.List;
public interface BlahDAO {
    List<Blah> getBlahs(Blah blah);
    void addBlah(Blah blah);
    void deleteBlah(Blah blah);
}
```

`UserService` 在进行账户或信息的相关存取时，必须委托给 `AccountDAO` 或 `BlahDAO` 对象，`UserService` 必须根据 `AccountDAO` 与 `BlahDAO` 定义的行为，而不考虑其实现对象如何运作。这么做的目的是清楚地理清储存逻辑与业务逻辑。在经过重构之后，`UserService` 的内容如下：

Gossip UserService.java

```
package cc.openhome.model;

import java.util.*;
import java.io.*;
```

JSP & Servlet

学习笔记(第2版)

```
public class UserService {
    private LinkedList<Blah> newest = new LinkedList<Blah>();
    private AccountDAO accountDAO;
    private BlahDAO blahDAO;

    public UserService(String USERS,
        AccountDAO userDAO, BlahDAO blahDAO) {
        this(userDAO, blahDAO);
    }

    public UserService(AccountDAO userDAO, BlahDAO blahDAO) {
        this.accountDAO = userDAO;
        this.blahDAO = blahDAO;
    }

    public boolean isUserExisted(Account account) {
        return accountDAO.isUserExisted(account);
    }

    public void add(Account account) {
        accountDAO.addAccount(account);
    }

    public boolean checkLogin(Account account) {
        if (account.getName() != null &&
            account.getPassword() != null) {
            Account storedAcct = accountDAO.getAccount(account);
            return storedAcct != null &&
                storedAcct.getPassword().equals(account.getPassword());
        }
        return false;
    }

    private class DateComparator implements Comparator<Blah> {
        @Override
        public int compare(Blah b1, Blah b2) {
            return -b1.getDate().compareTo(b2.getDate());
        }
    }
}

private DateComparator comparator = new DateComparator();

public List<Blah> getBlahs(Blah blah) {
    List<Blah> blahs = blahDAO.getBlahs(blah);
    Collections.sort(blahs, comparator);
    return blahs;
}

public void addBlah(Blah blah) {
    blahDAO.addBlah(blah);
    newest.addFirst(blah);
    if(newest.size() > 20) {
        newest.removeLast();
    }
}
```

```
    }

    public void deleteBlah(Blah blah) {
        blahDAO.deleteBlah(blah);
        newest.remove(blah);
    }

    public List<Blah> getNewest() {
        return newest;
    }
}
```

在代码中，粗体字的部分是依赖在 `AccountDAO` 或 `BlahDAO` 接口定义上，`UserService` 中看不到储存逻辑的实现。需要储存相关信息时，都是委托给 `AccountDAO` 或 `BlahDAO` 对象，`UserService` 中留下的就是商务相关逻辑，如检查用户是否存在、确认用户是否登录、用户信息排序、最新信息储存移除等。

提示»» 实际上，在定义出 `AccountDAO` 与 `BlahDAO` 接口后，是将 `UserService` 中的储存逻辑相关代码分别搬移至 `AccountDAO` 与 `BlahDAO` 的实现类后做适当修改(篇幅限制，这些实现类的源代码就不列出了，可以直接查看本书附带光盘中的范例查看修改成果)，只是碍于平面书籍无法展现修改过程，你看到的 `UserService` 是最后完成的成果。要记得，重构是调整现有代码架构，而非砍掉重练。

这是 DAO 设计模式的实现，在 DAO 设计模式中，会定义出储存逻辑的行为，在 Java 中通常是定义为接口，接口中定义的是与实际储存方案无关的行为。具体来说，就是不会出现任何储存方案 API 的抽象方法。例如，你看不到接口定义时会出现 `SQLException` 等 API 名称，真正采用哪种储存方案，则是由实现接口的类决定。

使用 DAO 隔离储存逻辑与业务逻辑的好处是，只要修改过后应用程序可以运作，将来若储存逻辑要改写为 JDBC，甚至更久之后的某个需求是要通过网络储存至远端服务器，都不用修改原有程序，而 `UserService` 业务逻辑清楚易于调整，修改业务逻辑时也不用担心误改了储存逻辑而发生错误。

提示»» 其实 `UserService` 的公开方法协议，我也做了一些调整，为的是让这些公开方法协议更清楚且易于使用。由于先前的练习，让相关 Servlet 依赖在 `UserService` 上。针对 `UserService` 的公开协议变化，相关 Servlet 也要做些小修改。由于 `UserService` 在先前练习中已封装了大部分代码，所以这些小修改不会太难，可以直接通过本书配套光盘中的范例查看修改成果。

9.4.2 使用 JDBC 实现 DAO

在重构 UserService 之后，接下来要分别实现 AccountDAO 与 BlahDAO。首先要创建数据库与表格，所使用的 SQL 如下：

```
CREATE SCHEMA gossip;
USE gossip;
CREATE TABLE t_account (
    name VARCHAR(15) NOT NULL,
    password VARCHAR(32) NOT NULL,
    email VARCHAR(255) NOT NULL,
    PRIMARY KEY (name)
) CHARSET=UTF8;
CREATE TABLE t_blah (
    name VARCHAR(15) NOT NULL,
    date TIMESTAMP NOT NULL,
    txt TEXT NOT NULL,
    FOREIGN KEY (name) REFERENCES t_account(name)
) CHARSET=UTF8;
```

接着使用 JDBC 实现 AccountDAO：

Gossip2 AccountDAOJdbcImpl.java

```
package cc.openhome.model;

import java.sql.*;
import javax.sql.DataSource;

public class AccountDAOJdbcImpl implements AccountDAO {
    private DataSource dataSource;

    public AccountDAOJdbcImpl(DataSource dataSource) {  
        ← ❶ 依赖在 DataSource  
        this.dataSource = dataSource;  
    }

    @Override
    public boolean isUserExisted(Account account) {
        Connection conn = null;
        PreparedStatement stmt = null;
        SQLException ex = null;
        boolean existed = false;
        try {
            conn = dataSource.getConnection(); ← ❷ 通过 DataSource 取得 Connection
            stmt = conn.prepareStatement(
                "SELECT COUNT(1) FROM t_account WHERE name = ?");
            stmt.setString(1, account.getName());
            ResultSet rs = stmt.executeQuery();
            if(rs.next()) {
                existed = (rs.getInt(1) == 1); ← ❸ 确认有查询结果
            }
        } catch (SQLException e) {
```

```

        ex = e;
    }
    finally {
        ...略
    }
    ...略
    return existed;
}

@Override
public void addAccount(Account account) {
    Connection conn = null;
    PreparedStatement stmt = null;
    SQLException ex = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(
            "INSERT INTO t_account(name, password, email) VALUES(?, ?, ?)");
        stmt.setString(1, account.getName());
        stmt.setString(3, account.getPassword()); ④ 取得 Account 中封装
        stmt.setString(3, account.getEmail()); 的信息更新表格字段
        stmt.executeUpdate();
    } catch (SQLException e) {
        ex = e;
    }
    finally {
        略...
    }
    略...
}

@Override
public Account getAccount(Account account) {
    Connection conn = null;
    PreparedStatement stmt = null;
    SQLException ex = null;
    Account acct = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(
            "SELECT password, email FROM t_account WHERE name = ?");
        stmt.setString(1, account.getName());
        ResultSet rs = stmt.executeQuery();
        if(rs.next()) {
            acct = new Account( ⑤ 查询到的账户数据封装为 Account 对象
                account.getName(), rs.getString(1), rs.getString(2));
        }
    } catch (SQLException e) {
        ex = e;
    }
    finally {
        ...略
    }
}

```

```
    ...略  
    return acct;  
}  
}
```

在实现 `AccountDAOJdbcImpl` 时，采用 JDBC 作为储存方案。`AccountDAOJdbcImpl` 对象创建时，必须传入 `DataSource` 实例①，之后要取得 `Connection` 对象时，就是从 `DataSource` 实例取得②。在查询账户是否存在时使用了 COUNT 语句，由于用户的名称是主键，所以只要确认查询到的笔数是否为 1，就可以知道用户是否存在③。在新增账户数据时，会从 `Account` 对象逐一取得数据，并设置为 `PreparedStatement` 的各字段值④。在取得账户数据时，会将查询到的表格字段逐个取出，并创建 `Account` 实例进行封装⑤。

接着使用 JDBC 实现 `BlahDAO` 接口。同样地，建构实例时，必须传入 `DataSource` 对象：

Gossip2 BlahDAOJdbcImpl.java

```
package cc.openhome.model;  
  
import java.sql.*;  
import java.util.*;  
import javax.sql.DataSource;  
  
public class BlahDAOJdbcImpl implements BlahDAO {  
    private DataSource dataSource;  
  
    public BlahDAOJdbcImpl(DataSource dataSource) {  
        this.dataSource = dataSource;  
    }  
  
    @Override  
    public List<Blah> getBlahs(Blah blah) {  
        Connection conn = null;  
        PreparedStatement stmt = null;  
        SQLException ex = null;  
        List<Blah> blahs = null;  
        try {  
            conn = dataSource.getConnection();  
            stmt = conn.prepareStatement(  
                "SELECT date, txt FROM t_blah WHERE name = ?");  
            stmt.setString(1, blah.getUsername());  
            ResultSet rs = stmt.executeQuery();  
            blahs = new ArrayList<Blah>();  
            while(rs.next()) {  
                blahs.add(new Blah(  
                    blah.getUsername(),  
                    rs.getTimestamp(1),  
                    rs.getString(2)));  
            }  
        } catch (SQLException e) {  
        }  
    }  
}
```

```
        ex = e;
    }
    finally {
        ...略
    }
    ...略
    return blahs;
}

@Override
public void addBlah(Blah blah) {
    Connection conn = null;
    PreparedStatement stmt = null;
    SQLException ex = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(
            "INSERT INTO t_blah(name, date, txt) VALUES(?, ?, ?)");
        stmt.setString(1, blah.getUsername());
        stmt.setTimestamp(2, new Timestamp(
            blah.getDate().getTime()));
        stmt.setString(3, blah.getTxt());
        stmt.executeUpdate();
    } catch (SQLException e) {
        ex = e;
    }
    finally {
        ...略
    }
    ...略
}

@Override
public void deleteBlah(Blah blah) {
    Connection conn = null;
    PreparedStatement stmt = null;
    SQLException ex = null;
    try {
        conn = dataSource.getConnection();
        stmt = conn.prepareStatement(
            "DELETE FROM t_blah WHERE date = ?");
        stmt.setTimestamp(1, new Timestamp(
            blah.getDate().getTime()));
        stmt.executeUpdate();
    } catch (SQLException e) {
        ex = e;
    }
    finally {
        ...略
    }
    ...略
}
```

}

9.4.3 设置 JNDI 部署描述

AccountDAO 与 BlahDAO 的实现都依赖于 DataSource, UserService 则依赖于 AccountDAO 与 BlahDAO, 必须有个地方完成这些对象之间彼此依赖关系的创建。这里将在 GossipListener 中完成。

Gossip2 GossipListener.java

```
package cc.openhome.web;

import javax.naming.*;
import javax.servlet.*;
import javax.servlet.annotation.WebListener;
import javax.sql.DataSource;
import cc.openhome.model.*;

@WebListener
public class GossipListener implements ServletContextListener {
    public void contextInitialized(ServletContextEvent sce) {
        try {
            Context initContext = new InitialContext();
            Context envContext = (Context)
                initContext.lookup("java:/comp/env");
            DataSource dataSource =
                (DataSource) envContext.lookup("jdbc/gossip");
            ServletContext context = sce.getServletContext();
            context.setAttribute("userService", new UserService(
                new AccountDAOJdbcImpl(dataSource),
                new BlahDAOJdbcImpl(dataSource)));
        } catch (NamingException ex) {
            throw new RuntimeException(ex);
        }
    }

    public void contextDestroyed(ServletContextEvent sce) {}
}
```

① 通过 JNDI 取得
DataSource

② 设置
UserService、
AccountDAO、
BlahDAO 与
DataSource
间的依赖关系

在 GossipListener 中通过 JNDI 取得了 DataSource 实例①，并完成了 AccountDAO、BlahDAO 对 DataSource 的依赖，以及 UserService 对 AccountDAO、BlahDAO 的依赖关系②。

由于应用程序中通过 JNDI 取得 DataSource，必须在部署描述文件中加以声明：

Gossip2 web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app ...>
    // 略...
    <resource-ref>
```

```

<res-ref-name>jdbc/BookmarkOnline</res-ref-name>
<res-type>javax.sql.DataSource</res-type>
<res-auth>Container</res-auth>
<res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
</web-app>

```

先前 `GossipListener` 需要从初始参数中取得储存数据文件的文件夹名称，现在已不需要，而可以将对应的初始参数设置从 `web.xml` 中移除。

实际上 JNDI 是服务器上的资源，`web.xml` 中的设置，只是请容器代为向服务器进行沟通，服务器上必须设置好 JNDI，这里将采用 Tomcat 7，所以可在 `META-INF` 文件夹中新增一个 `context.xml`。内容编写如下：

Gossip2 context.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<Context antiJARLocking="true" path="/Gossip2">
    <Resource name="jdbc/gossip"
        auth="Container" type="javax.sql.DataSource"
        maxActive="100" maxIdle="30" maxWait="10000" username="root"
        password="123456" driverClassName="com.mysql.jdbc.Driver"
        url="jdbc:mysql://localhost:3306/gossip?
            useUnicode=true&characterEncoding=UTF8"/>
</Context>

```

这样在应用程序部署之后，Tomcat 7 就会载入 JDBC 驱动程序、创建 DBCP 连接池、创建 JNDI 相关资源。由于 Tomcat 7 必须载入 JDBC 驱动程序，因此要将驱动程序的 JAR 文件放在 Tomcat 7 的 `lib` 文件夹中。

9.5 重点复习

JDBC(Java DataBase Connectivity)是用于执行 SQL 的解决方案，开发人员使用 JDBC 的标准接口，数据库厂商则对接口进行实现，开发人员无须接触底层数据库驱动程序的差异性。

厂商在实现 JDBC 驱动程序时，依方式可将驱动程序分为四种类型：

- Type 1: JDBC-ODBC Bridge Driver
- Type 2: Native API Dirver
- Type 3: JDBC-Net Driver
- Type 4: Native Protocol Driver

数据库操作相关的 JDBC 接口或类都位于 `java.sql` 包中。要连接数据库，可以向 `DriverManager` 取得 `Connection` 对象。`Connection` 是数据库连接的代表对象，一个 `Connection` 对象就代表一个数据库连接。`SQLException` 是在处理 JDBC 时经常遇到的一个异常对象，为数据库操作过程发生错误时的代表对象。

在 Java EE 的环境中，将取得连接等与数据库源相关的行为规范在 `javax.sql.DataSource` 接口中，实际如何取得 `Connection` 则由实现接口的对象来负责。

`Connection` 是数据库连接的代表对象，接下来要执行 SQL 的话，必须取得 `java.sql.Statement` 对象，它是 SQL 语句的代表对象，可以使用 `Connection` 的 `createStatement()` 来创建 `Statement` 对象。

Statement 的 `executeQuery()` 方法则是用于 SELECT 等查询数据库的 SQL, `executeUpdate()` 会返回 `int` 结果, 表示数据变动的笔数, `executeQuery()` 会返回 `java.sql.ResultSet` 对象, 代表查询的结果, 查询的结果会是一笔一笔的数据。可以使用 `ResultSet` 的 `next()` 来移动至下一笔数据, 它会返回 `true` 或 `false` 表示是否有下一笔数据, 接着可以使用 `getXXX()` 来取得数据。

在使用 Connection、Statement 或 ResultSet 时，要将之关闭以释放相关资源。

如果有些操作只是 SQL 语句中某些参数会有所不同，其余的 SQL 子句皆相同，则可以使用 `java.sql.PreparedStatement`。可以使用 `Connection` 的 `prepareStatement()` 方法创建好一个预编译(precompile)的 SQL 命令，其中参数会变动的部分，先指定“?”这个占位字符。等到需要真正指定参数执行时，再使用相对应的 `setInt()`、`setString()` 等方法，指定“?”处真正应该有的参数。

9.6 课后练习

9.6.1 选择题

1. () JDBC 驱动程序可以有跨平台的特性。
(A) TYPE 1 (B) TYPE 2
(C) TYPE 3 (D) TYPE 4
 2. () JDBC 驱动程序是基于数据库所提供的 API 来进行实现的。
(A) TYPE 1 (B) TYPE 2
(C) TYPE 3 (D) TYPE 4
 3. JDBC 相关接口或类，是放在()包之下加以管理。
(A) java.lang (B) javax.sql
(C) java.sql (D) java.util
 4. 使用 JDBC 时，通常会需要处理的受检异常(Checked Exception)是()。
(A) RuntimeException (B) SQLException
(C) DBException (D) DataException

5. 关于 Connection 的描述，以下正确的是()。

 - (A) 可以从 DriverManager 上取得 Connection
 - (B) 可以从 DataSource 上取得 Connection
 - (C) 在方法结束之后 Connection 会自动关闭
 - (D) Connection 是线程安全(Thread-safe)

6. 使用 Statement 来执行 SELECT 等查询用的 SQL 指令时，应使用()方法。

 - (A) executeSQL()
 - (B) executeQuery()
 - (C) executeUpdate()
 - (D) executeFind()

7. 以下()对象在正确使用的情况下，可以适当地避免 SQL Injection 的问题。

 - (A) Statement
 - (B) ResultSet
 - (C) PreparedStatement
 - (D) Command

8. 取得 Connection 之后，可以使用()方法取得 Statement 对象。

 - (A) conn.createStatement()
 - (B) conn.buildStatement()
 - (C) conn.getStatement()
 - (D) conn.createSQLStatement()

9. 以下描述有误的是()。

 - (A) 使用 Statement 一定会发生 SQL Injection
 - (B) 使用 PreparedStatement 就不会发生 SQL Injection
 - (C) 不使用 Connection 时必须加以关闭
 - (D) ResultSet 代表查询的结果集合

10. 使用 Statement 的 executeQuery() 方法，会返回()类型。

 - (A) int
 - (B) boolean
 - (C) ResultSet
 - (D) Table

9.6.2 实训题

在微博应用程序的 `AccountDAOJdbcImpl` 与 `BlahDAOJdbcImpl` 中，为了处理 `SQLException` 与正确地关闭 `Statement`、`Connection`，充斥着大量重复的 `try...catch` 代码，请尝试通过设计的方式，让 `try...catch` 代码可以重复使用，以简化 `AccountDAOJdbcImpl` 与 `BlahDAOJdbcImpl` 的源代码内容。

提示»» 搜寻关键字 JdbcTemplate 了解相关设计方式。