第 3 章 JavaWeb 服务器端开发基础

3.1 建立开发平台

3.1.1 安装 JDK

在正式开始开发 Java 程序之前，要先安装好 JDK，首先进入官方网站下载最新版本的 JDK（地址为 http://www.oracle.com/technetwork/java/javase/downloads/index.html）。安装完成后需要确认安装是否成功，打开 CMD，输入 java-version，如果可以看到 Java 的版本信息说明安装已经成功了，如图 3.1-1 所示。

图 3.1-1 检验是否安装成功

安装成功后继续配置 Java 的环境变量以便让系统知道 JDK 的具体位置。打开“我的电脑→属性→高级系统设置→高级”，单击环境变量按钮进行环境变量的设置，如图 3.1-2 所示。

在系统变量中添加一个新的变量 JAVA_HOME，并设置其值为 JDK 的安装目录，可以直接用文件浏览器打开相关目录直接复制文件路径粘贴。然后将 JAVA_HOME 添加到系统变量 Path 之中，具体方法是编辑打开系统变量 Path，在原有的变量值前添加双引号内的代码“%JAVA_HOME %/bin;”，其中的分号起分隔作用，不可缺少，如图 3.1-3 所示。

3.1.2 安装 Tomcat

在进行 Java Web 开发之前，还需要安装服务器软件 Tomcat，Tomcat 是一个免费的开放源代码的 Web 应用服务器，属于轻量级应用服务器。首先进入 Tomcat 的官方网站下载其安装包。安装过程中用户可以选择需要安装的组件，一般选择默认即可，进入下一步，设

置 Tomcat 的常用端口，一般采用默认端口。如果你的计算机某些端口已经被其他软件占用，也可以改用其他端口。此外，也可以设置 Tomcat 的用户账号和密码加强其安全性，在测试环境中一般采用默认设置。

图 3.1-2　设置环境变量(1)

图 3.1-3　设置环境变量(2)

进入下一步，选择 Java 虚拟机的安装路径。因为 Tomcat 在执行 Java 应用的时候需要调用 Jre 去执行 Java 字节码，而之前已经安装好了 JDK 和 JRE 并且设置好了环境变量，所以这一步一般会自动填写好路径。如果安装包没有自动寻找到 JRE 的路径，请检查前面的安装和设置工作是否完成或者手动选择路径。

进入下一步，选择 Tomcat 的安装路径，可以自定义一个安装路径来安装 Tomca。最后单击“安装”按钮，等待片刻后显示安装完成。

单击“完成”按钮 Tomcat 便会启动，单击系统右下角的 Tomcat 图标会弹出可视化管理界面，可以停止或启动它的服务并进行一些常用的设置，如图 3.1-4 所示。

最后，打开浏览器，输入地址查看安装是否成功，如果可以显示图 3.1-5 所示的界面，说明 Tomcat 已经可以正常工作了。

3.1.3　安装 PostgreSQL 数据库

数据库的种类很多，比如流行的 MySQL、企业级的 Oracle 等，本书以 PostgreSQL 为例讲解数据库的安装。PostgreSQL 是以加州大学伯克利分校计算机系开发的对象关系型数据库管理系统。它支持大部分 SQL 标准并且拥有许多其他现代特性：复杂查询、外键、触发器、视图、事务完整性、MVCC。同样，PostgreSQL 可以用许多方法扩展，不管是私用、商用、还是学术研究使用，比如通过增加新的数据类型、函数、操作符、聚集函数、索引免费使用、修改和分发 PostgreSQL。

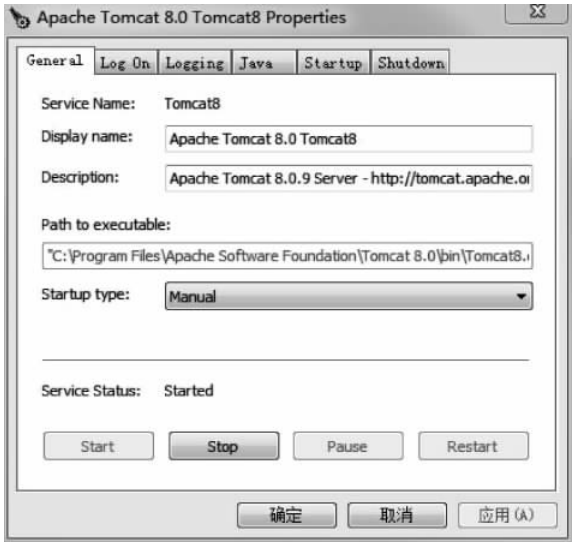

图 3.1-4 Tomcat 启动界面

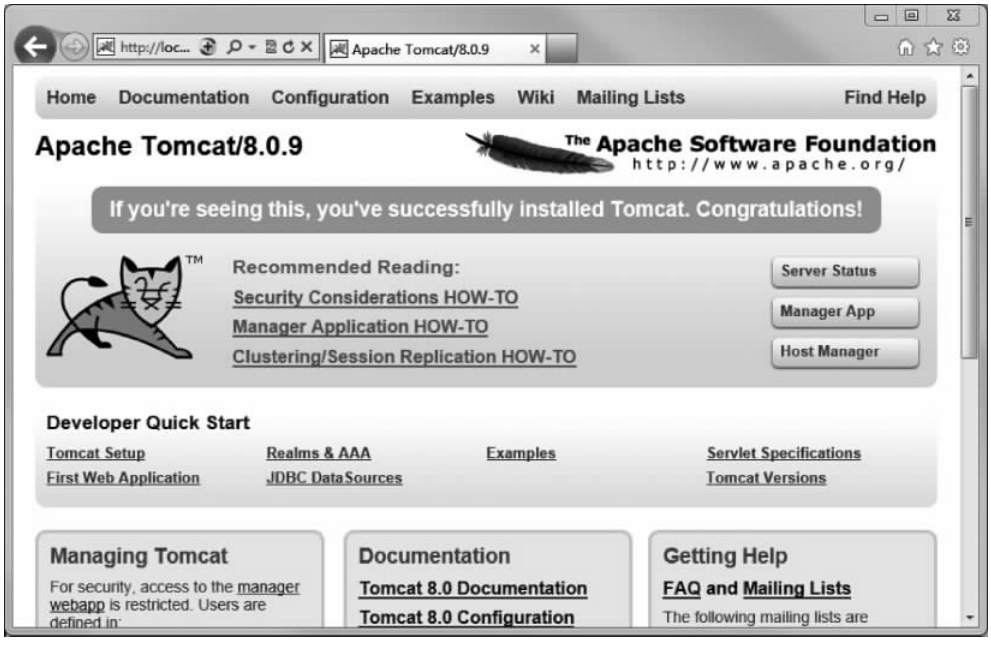

图 3.1-5 Tomcat 成功启动的界面

首先,进入 PostgreSQL 官方网站 http://www.postgresql.org/,下载最新的安装包,下载完成后单击安装包进入安装界面。

进入“下一步”,选择数据库的安装目录。

进入“下一步”,选择数据文件的存放位置,默认为数据库安装目录下,也可以自定义。

进入“下一步”,设置数据库管理员的密码。

设置数据库的访问端口,默认为 5432,也可以根据需要改成其他端口。

进入“下一步”选择所在区域,可以直接选择默认选项,最后单击“安装”完成安装。

3.1.4 安装 Eclipse

在进行开发前,首先要选择一个集成开发环境,这能给开发工作带来很多方便,本书中选择普遍使用的 Eclipse 来进行 Java 的开发,首先进入其官方网站下载 http://www.eclipse.org/home/index.php。

Eclipse 是一个很受欢迎的开发软件,因此其版本也十分多,为了方便进行 Java 开发,选择 JavaEE 的版本,它与其他版本的区别是默认集成了一些开发 Java 程序需要用到的插件,可以省去自行安装的麻烦。

3.2 MVC 模式及对象持久化

3.2.1 开发框架简介

在了解 MVC 之前,首先来了解框架的概念。要理解框架的含义得从开发的实际需求说起。在软件开发过程中总有很多基础的功能是相同或者相近的,所以在实际开发中再花费时间重复基础工作显然是人们不愿意做的事情,所以开发者们将一些可重用的、易扩展的,并且经过良好测试的组件独立出来抽象成一个框架,当开发新的项目时便可以直接基于框架开发,可以有更多的精力放在分析和构建业务逻辑上,从而避免繁琐的底层事务和代码工程,这就是框架的由来。

3.2.2 MVC 的层结构

MVC 是 Model View Controller 的缩写,是一种开发模式,因为其合理的设计,诞生了很多以 MVC 开发模式为主导的框架,称为 MVC 框架,如图 3.2-1 所示。

(1) Model(模型):表示程序的核心,处理一些底层的业务逻辑。

(2) View(视图):显示页面数据。

(3) Controller(控制器):控制输入/输出,协调各个层的关系。

3.2.3 对象关系映射 ORM 技术

虽然关系型数据库已经大大地方便了程序员进行程序开发,但在一些大型的项目中数据关系比较复杂,业务逻辑要求可能更多,此时再从底层进行数据库的操作显得比较费事,于是人们提出了 ORM(Object Relational Mapping)技术,它是一种将关系型数据映射到类的技术。使用 ORM 技术进行开发,程序员不需要去处理繁琐的底层数据,而是通过映射类进行数据操作,更加符合软件工程面向对象的思想。使用 ORM 技术,开发者可以访问到底

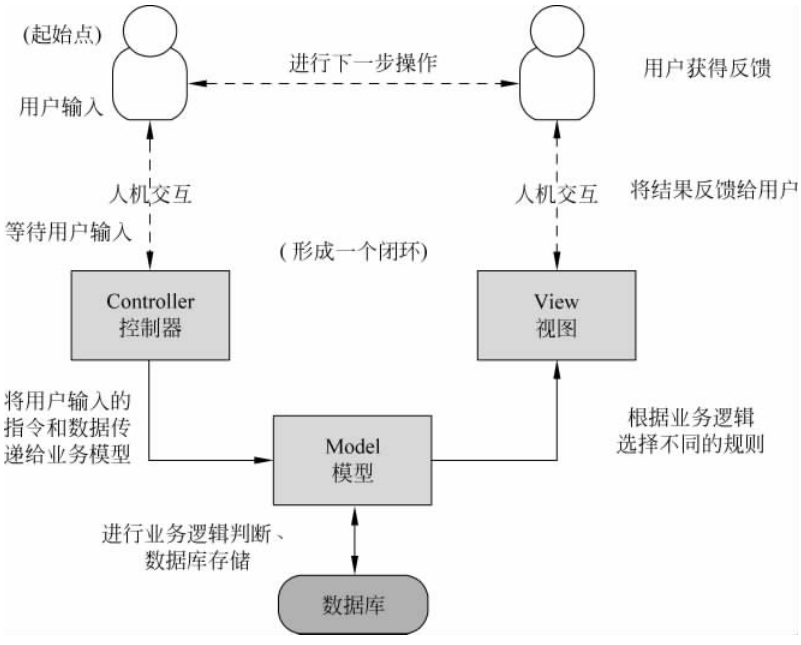

图 3.2-1 MVC 基本结构

层的数据,但完全不用去关心底层数据的结构,这个过程也可以称为持久化的过程。

3.2.4 SSH 集成开发框架

SSH 为 Struts+Spring+Hibernate 的一个集成框架,是目前较流行的一种 Web 应用程序开源框架。Struts2 是一个 Java 的 MVCWeb 开发框架,Struts2 以 WebWork 为核心,采用拦截器的机制来处理用户的请求,这样的设计也使得业务逻辑控制器能够与 ServletAPI 完全脱离开。Spring 是一个开源框架,是为了解决企业应用程序开发复杂性而创建的。Hibernate 是一个开放源代码的对象关系映射框架,它对 JDBC 进行了轻量级的对象封装,使得 Java 程序员可以随心所欲地使用对象编程思维来操作数据库。Hibernate 可以应用在任何使用 JDBC 的场合,既可以在 Java 的客户端使用,也可以在 Servlet/JSP 的 Web 应用中使用。

3.3 Struts2 框架的使用

3.3.1 Struts2 框架的下载及部署

在使用 Struts2 前需要去官网下载最新的框架包,地址为 http://struts.apache.org/release/2.0.x/。可以选择下载完整包,里面除了包含必需的文件外还包含了文档和案例,也可以选择只下载运行包。这里选择下载完整的压缩包,解压后会看到目录结构如图 3.3-1

名称	修改日期	类型	大小
apps	2014/5/2 18:04	文件夹	
docs	2014/5/2 18:04	文件夹	
lib	2014/5/2 18:04	文件夹	
src	2014/5/2 18:04	文件夹	
ANTLR-LICENSE.txt	2014/5/2 17:19	文本文档	2 KB
CLASSWORLDS-LICENSE.txt	2014/5/2 17:19	文本文档	2 KB
FREEMARKER-LICENSE.txt	2014/5/2 17:19	文本文档	3 KB
LICENSE.txt	2014/5/2 17:19	文本文档	10 KB
NOTICE.txt	2014/5/2 17:19	文本文档	1 KB
OGNL-LICENSE.txt	2014/5/2 17:19	文本文档	3 KB
OVAL-LICENSE.txt	2014/5/2 17:19	文本文档	12 KB
SITEMESH-LICENSE.txt	2014/5/2 17:19	文本文档	3 KB
XPP3-LICENSE.txt	2014/5/2 17:19	文本文档	3 KB
XSTREAM-LICENSE.txt	2014/5/2 17:19	文本文档	2 KB

图 3.3-1　Struts2 的框架包

所示。

需要用到的是 lib 文件夹下的 jar 文件，现在复制这些文件到之前新建的工程中的 WebContent/WEB-INF/lib 文件夹下，以便在工程中使用这些文件。lib 文件夹中通常会放置工程需要用到的 jar 文件，引用相关的 Java 类时会自动加载该文件夹下的资源，可以理解为一个资源库，需要什么取什么。

在 Eclipse 中刷新之前的工程可以看到在 WebAppLibraries 节点下复制过来的 jar 文件，说明已经成功将 Struts2 的文件部署到了工程中，如果不能看到文件请重新检查复制的路径是否正确。

Struts2 框架的部署本质上就是将文件添加到我们的工程中以便调用而已，而在很多书中可能会将该过程描述为安装的过程，将简单的过程抽象了，十分不利于学习，这点读者需要清楚。

3.3.2　Struts2 配置

在部署好了相关的 Struts2 文件后只是做好了预备工作，只有框架是不能够运行出什么效果的，现在需要进一步对框架进行配置才能使其工作。

从前面讲述 JSP 显示简单页面的内容已经知道，我们不需要任何配置文件，只需要在应用的根目录下建立新的 JSP 文件，然后通过网址/文件名就可以访问动态页面了，这是最基本的 Web 开发形式，因为和本地浏览文档的逻辑几乎一样。但复杂的 Web 程序通常不会以文件为地址单位进行开发，这不利于项目的开发和维护，并且会暴露工程的目录结构。通常会统一使用一个路口来管理应用的访问，这好比进入一幢楼的某个房间必须先经过大门一样，而不是直接就能进入某个房间。在 Java Web 开发中，扮演入口角色的通常是 web.xml 文件，在里面可以进行工程的各种配置工作，所以想要让 Struts2 的框架正常工作起来，首先要配置 web.xml 这张地图，只有通过 web.xml 程序才能找到 Struts2 的入口。

在 WEB-INF 文件夹新建一个 web.xml 文件。编辑该文件如下：

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< web - app id = "WebApp_9" version = "2.4" xmlns = "http://java.sun.com/xml/ns/j2ee" xmlns:xsi
 = "http://www.w3.org/2001/XMLSchema - instance" xsi:schemaLocation = " http://java.sun.com/
xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web - app_2_4.xsd">
    < display - name > Struts Blank </display - name >
    < filter >
            < filter - name > struts2 </filter - name >
                < filter - class >
    org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
                </filter - class >
    </filter >
    < filter - mapping >
            < filter - name > struts2 </filter - name >
            < url - pattern >/ * </url - pattern >
    </filter - mapping >
    < welcome - file - list >
            < welcome - file > index.html </welcome - file >
    </welcome - file - list >
</web - app >
```

这里要提醒的是，不同版本的 Struts2 可能文件的配置会略有不同，所以在下载完新版本的 Struts2 后使用本书或者其他资料中的文件未必正确，最好的方法是参考查看下载文件中的 webapp 范例中的配置文件。

可以看到，该文件中添加了一个过滤器，将过滤所有访问请求，＜filter-name＞struts2＜/filter-name＞声明了该过滤器的名字为 struts2，而＜url-pattern＞/ * ＜/url-pattern＞指明了过滤的路径为工程下的所有访问路径，这里用到的正则表达，/表示为根目录，而 * 用于匹配所有字符。

在 filter 标签中声明了该过滤器用到的类为 org. apache. struts2. dispatcher. ng. filter. StrutsPrepareAndExecuteFilter，这个类就在之前添加到 lib 文件夹中的 jar 文件中。现在来总结 web. xml 将如何工作，当用户访问我们的应用时会先经过预定好的 Struts2 过滤器，org. apache. struts2. dispatcher. ng. filter. StrutsPrepareAndExecuteFilter 过滤器类会将所有的访问请求全部转交给 Struts2 处理。

同样，Struts2 本身也有自己的 xml 配置文件，担当着类似 web. xml 的作用，在 src 文件夹下再新建一个名为 struts. xml 的文件。编辑该文件内容如下：

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE struts PUBLIC
    " - //Apache Software Foundation//DTD Struts Configuration 3.3//EN"
    "http://struts.apache.org/dtd s/struts - 3.3.dtd ">
< struts >
    < constant name = "struts.enable.DynamicMeth odInvocation" value = "false" />
    < constant name = "struts.devMode" value = "true" />
    < package name = "default" namespace = "/" extends  = "struts - default">
```

```
    </package>
</struts>
```

在该文件中添加了一个 package，命名为 helloworld，在 struts.xml 文件中，package 起着分类管理 Action 的作用，对于大型的项目，将所有的 action 放置在一起显然会让人眼花缭乱，不利于开发和维护，有了 Action 便可以对它们进行分类管理。

3.3.3　创建第一个 Action 实例

现在开始编写第一个 Action 类，右击 src 节点，单击 New→Class。在弹出的对话框中填写类名为 HelloWorld。然后单击 Finish 按钮完成类的新建，如图 3.3-2 所示。

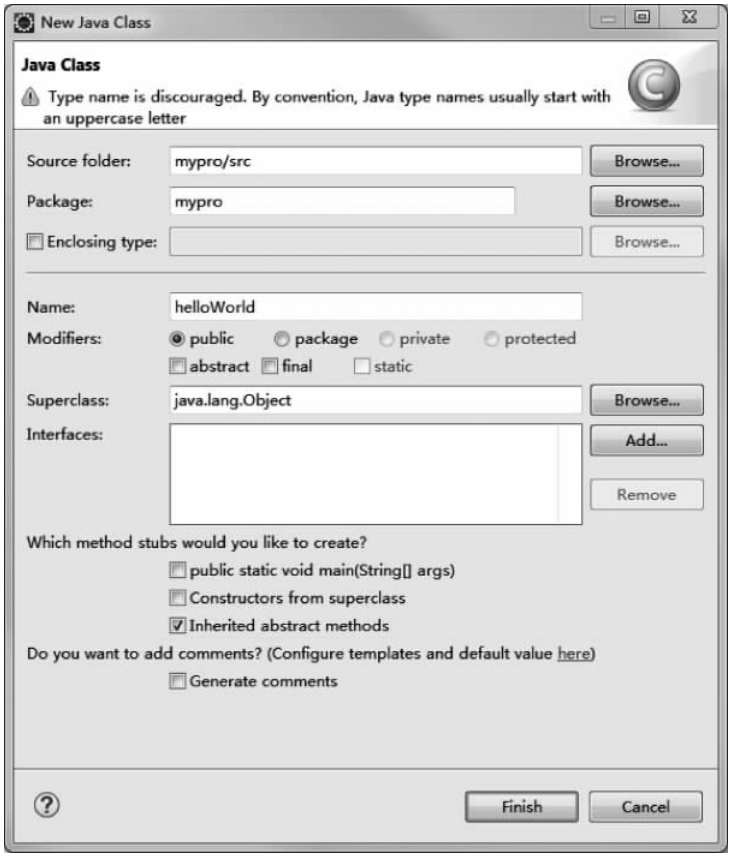

图 3.3-2　设置类的信息

可以看到在 src 节点下的 mypro 包中新建了一个 Java 文件。编辑这个文件如下：

```
package mypro;
import com.opensymphony.xwork2.ActionSupport;
public class helloWorld extends ActionSupport {
```

```
public String message = "Hello, world!";
public String execute() throws Exception {
        return SUCCESS;
    }
}
```

现在已经编写完了一个最基本的 Action 类，在这个类中，添加了一个名为 execute()的方法，在 Action 类被执行的时候它会默认执行这个方法来处理客户端发送的请求。

编写完 Action 类后还需要创建一个视图文件，用于显示返回的内容。在 WebConent 文件夹下新建一个 templets 的文件夹用于放置模板文件，在该文件夹下新建一个 HelloWorld.jsp 的文件。需要提醒的是，文件的设定和命名并不是固定的，实际开发中可以根据项目的规划来自己设定目录的结构。编辑这个文件的内容如下：

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<html>
<head>
<title>Hello World!</title>
</head>
<body>
    <h2>
        <s:property value="message" />
    </h2>
</body>
</html>
```

第一行的作用是使得该页面可以支持 Struts2 特有的标签，通过 Struts2 的 property 标签可以输出相关的信息，这些信息一般为所对应的 Action 类中的成员变量，到这里一个基本的 Action 和它的视图就完成了，接下来将去实现在浏览器中显示出添加的页面。

3.3.4 使用 Struts2 的动作

最后实现 Action 的工作需要在 struts.xml 文件中添加相关配置，前面已经说了在 JavaWeb 工程中 xml 文件相当于一张地图，只有在地图中注册了相关信息，程序才知道该在什么时候以及什么情况执行什么内容。打开该文件添加 Action 配置内容，文件修改如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 3.3//EN"
    "http://struts.apache.org/dtd s/struts-3.3.dtd ">
<struts>
<constant name="struts.enable.DynamicMeth odInvocation" value="false" />
<constant name="struts.devMode" value="true" />
<package name="default" namespace="/" extends ="struts-default">
<default-action-ref name="index" />
```

```
    <global-results>
        <result name="error">/error.jsp</result>
    </global-results>
    <global-exception-mappings>
        <exception-mapping exception="java.lang.Exception" result="error"/>
    </global-exception-mappings>
    <action name="helloWorld" class="mypro.helloWorld">
        <result>/templets/helloWorld.jsp</result>
    </action>
</package>
</struts>
```

该文件中添加了一个 Action 标签，命名为 HelloWorld，并指明了该动作对应的 Action 类为 mypro.HelloWorld。当该 Action 执行完毕后会跳转到/templets/HelloWorld.jsp 这个模板文件返回给前台显示。

启动本地的 server，打开浏览器输入地址 http://localhost:8080/mypro/helloWorld.action，可以看到数据已经正确地被显示出来了，如图 3.3-3 所示。

图 3.3-3　使用 Action 显示 HelloWorld

3.3.5　通过 Action 接收前台数据

在进一步深入学习相关知识前，让我们重新部署之前编写的登录界面，以便更好地进行效果演示。首先在 templets 下新建一个文件夹 login.jsp，并将之前编写的 HTML 登录界面复制进去。

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<head>
```

```
    <meta charset = "utf - 8">
    <link href = "leeui/style/leeui - base.css" type = "text/css" rel = "stylesheet">
    <style type = "text/css">
        body{text - align: center;}
        #login - box{width :300px;margin:0 auto;margin - top: 100px;}
    </style>
</head>
<body>
    <div id = "login - box">
        <h1>基于 struts2 的登录 DEMO</h1>
        <form action = "" meth od = "post">
        <ul class = "lee - form - normal">
            <li>
                <div class = "lee - input">
                    <label>邮箱</label>
                    <input type = "text" name = "email" value = "">
                </div>
            </li>
            <li>
                <div class = "lee - input">
                    <label>密码</label>
                    <input type = "password " name = "password " value = "">
                </div>
            </li>
            <li>
                <input class = "lee - button" style = "width :80px" type = "submit" value = "提交">
            </li>
        </ul>
        </form>
    </div>
</body>
</html>
```

然后将 leeui 的文件夹放置到 WebContent 根目录下，新建一个 login 的 Java 类，编辑内容如下：

```
package mypro;
import com.opensymphony.xwork2.ActionSupport;
public class login extends ActionSupport {
public String execute() throws Exception {
return SUCCESS;
    }
}
```

修改 struts.xml 配置文件，在其中添加一个新的 Action：

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

```
<!DOCTYPE struts PUBLIC
    " - //Apache Software Foundation//DTD Struts Configuration 3.3//EN"
    "http://struts.apache.org/dtd s/struts - 3.3.dtd ">
< struts >
< constant name = "struts.enable.DynamicMeth odInvocation" value = "false" />
< constant name = "struts.devMode" value = "true" />
< package name = "default" namespace = "/" extends  = "struts - default">
< default - action - ref name = "index" />
< global - results >
    < result name = "error">/error.jsp </result >
</global - results >
< global - exception - mappings >
    < exception - mapping exception = "java.lang.Exception" result = "error"/>
</global - exception - mappings >
< action name = "helloWorld" class = "mypro.helloWorld">
    < result >/templets/helloWorld.jsp </result >
</action >
< action name = "login" class = "mypro.login">
    < result >/templets/login.jsp </result >
</action >
</package >
</struts >
```

这样就完成了一个基本登录 Action 的编写了，重启服务器在浏览器输入地址 http://localhost:8080/mypro/login，如果能显示图 3.3-4 所示页面就说明新添加的 Action 已经成功工作了。

图 3.3-4　通过 Action 显示登录界面

接下来继续实现下一项功能：通过这个登录页面将用户输入的邮箱和密码信息发送到后台并且显示出来。创建一个新的 Action 用来接收和处理前台传送过来的参数，新建一个 Java 类名为 handleLogin，并编辑内容如下：

```
package mypro;
import com.opensymphony.xwork2.ActionSupport;
public class handleLogin extends ActionSupport {
    private String email;
    private String password ;
public String execute() throws Exception {
return SUCCESS;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword () {
        return password ;
    }
    public void setPassword (String password ) {
        this.password = password ;
    }
}
```

Java 几乎是完全面向对象的语言，所以在类的外部一般不会直接去调用类的内部变量，因为这不符合面向对象的思想，通常会对每个成员变量定义相应的操作函数，就是 get 和 set 方法，通过 get 方法来获取类的内部成员变量，通过 set 方法来设定类的内部成员变量，这种通过类方法操作变量的方法更符合面向对象的思想，利于程序的编写。

这里介绍一个小技巧，在编写复杂的类时可能要创建较多的 get 和 set 方法，这些繁琐重复的工作可以直接让程序完成，具体方法是定义好成员变量后右击空白处，单击 Source→GenerrateGettersandSetters。

在弹出的设置界面中勾选要定义 get 和 set 方法的成员变量，然后单击“完成”按钮，系统会自动完成相关成员方法的编写，不仅十分方便，还能避免不必要的拼写错误，如图 3.3-5 所示。

在 Struts 的 Action 中，会通过 get 和 set 自动完成一些参数的接收和传递的作用。当用户发送请求到 Action 时，Struts2 会自动调用同名的 set 方法将传递的参数赋值给类成员变量，在对应 Action 的 jsp 模板页面中，只有定义了成员变量相应的 get 函数，才可以输出相应的变量值。

在创建完 Action 后，还需要创建一个 jsp 模板文件用来显示接收到的数据。在 templets 文件夹下新建一个文件名为 loginResult.jsp，使用 Struts2 的 property 标签输出接

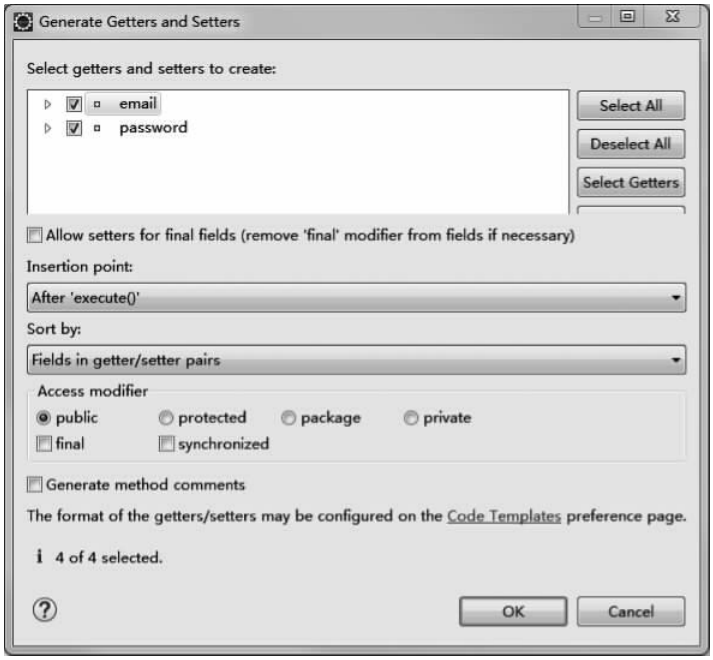

图 3.3-5　选择要创建 get 和 set 方法的成员变量

受到的内容，编辑内容如下：

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>登录参数显示</title>
<link href="leeui/style/leeui-base.css" type="text/css" rel="stylesheet">
<style type="text/css">
        body{text-align: center;}
        #login-box{width :300px;margin:0 auto;margin-top: 100px;}
    </style>
</head>
<body>
    <div id="login-box">
        <h1>用户输入的参数为</h1>
        <ul class="lee-form-normal">
            <li>
                <div class="lee-input">
```

```
                    <label>邮箱</label>
                    <input type="text" name="email" value="<s:property value="email" />">
                </div>
            </li>
            <li>
                <div class="lee-input">
                    <label>密码</label>
                    <input type="text" name="password" value="<s:property value=
                "password" />">
                </div>
            </li>
        </ul>
    </div>
</body>
</html>
```

在 struts.xml 中添加一个新的 Action：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 3.3//EN"
    "http://struts.apache.org/dtd s/struts-3.3.dtd ">
<struts>
    <constant name="struts.enable.DynamicMeth odInvocation" value="false" />
    <constant name="struts.devMode" value="true" />
    <package name="default" namespace="/" extends ="struts-default">
    <default-action-ref name="index" />
    <global-results>
            <result name="error">/error.jsp</result>
    </global-results>
    <global-exception-mappings>
            <exception-mapping exception="java.lang.Exception" result="error"/>
    </global-exception-mappings>
    <action name="helloWorld" class="mypro.helloWorld">
            <result>/templets/helloWorld.jsp</result>
    </action>
    <action name="login" class="mypro.login">
            <result>/templets/login.jsp</result>
    </action>
    <action name="handleLogin" class="mypro.handleLogin">
        <result>/templets/loginResult.jsp</result>
    </action>
</package>
</struts>
```

现在登录和显示登录信息的界面已经都做好了，还需要修改登录界面的目标地址。否则，在用户提交登录表单的时候，程序无法知道向什么地址提交数据，编辑 Login.jsp 页面

中的 form 标签的 Action 属性，该属性的作用是当用户单击“提交”按钮时前台会将数据提交到这个地址。编辑后的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<head>
    <meta charset="utf-8">
    <link href="leeui/style/leeui-base.css" type="text/css" rel="stylesheet">
    <style type="text/css">
        body{text-align: center;}
        #login-box{width:300px;margin:0 auto;margin-top: 100px;}
    </style>
</head>
<body>
    <div id="login-box">
        <h1>基于 struts2 的登录 DEMO</h1>
        <form action="handleLogin.action" method="post">
        <ul class="lee-form-normal">
            <li>
                <div class="lee-input">
                    <label>邮箱</label>
                    <input type="text" name="email" value="">
                </div>
            </li>
            <li>
                <div class="lee-input">
                    <label>密码</label>
                    <input type="password" name="password" value="">
                </div>
            </li>
            <li>
                <input class="lee-button" style="width:80px" type="submit" value="提交">
            </li>
        </ul>
        </form>
    </div>
</body>
</html>
```

现在，所有的工作都已经完成了，打开浏览器输入地址 http://localhost:8080/mypro/login，登录到之前编写的登录界面，输入邮箱和密码，然后提交表单，如图 3.3-6 所示。

在提交表单之后跳转到登录参数显示的页面，并且显示除了之前输入的数据，说明系统已经正常工作了，如图 3.3-7 所示。

上面的案例中使用的是 post 方法提交数据，在 Web 应用中提交数据的方法主要有两

图 3.3-6　提交一个登录表单

图 3.3-7　后台接收表单参数并显示出来

种，分别是 get 方法和 post 方法，两者的区别在于：get 方法的参数是跟在提交的 URL 地址的后面的，它的基本格式为：根地址？参数名 1＝值 1&参数名 2＝值 2，而 post 方法则是把提交的数据放在 HTTP 包体中。

在浏览器输入地址 http://localhost:8080/mypro/handleLogin? email＝test@test.

com&password =123456,可以看到出现了同样的显示结果,说明 email 和 password 参数也被正确接受了。

post 方法和 get 方法对比如表 3.3-1 所示。

表 3.3-1 post 方法和 get 方法对比

方法	大小限制	安全性	适用范围
post	理论上无大小限制,但实际上取决于服务器配置	高	提交数据对服务器内容进行修改
get	最大为 1024 字节	低	通常用于获取数据,也可以用于请求数据修改

本质上来讲,get 是向服务器发送一个获取数据的请求,而 post 是向服务器提交数据(意味着可能会改变服务器上的数据),在实际应用中可以灵活运用这两种方法。

3.3.6 通过 Session 记录登录状态

在使用一个具有会员系统的 Web 应用时会遇到登录状态保持的问题,在 Web 应用中,每个页面都是独立访问的,不可能让用户访问每个页面前都输入一次密码。在 GIS 项目的开发中同样会涉及用户权限管理的问题,所以如何实现用户登录状态的保持也十分重要。

实现用户登录状态保持的方法有很多,但本质上都是一样的,即用户登录时在服务器保存一份凭证,在用户登录后访问其他页面时通过检验是否存在合法凭证来判断用户是否有访问该页面的权限。在用户注销登录后系统再将该凭证作废或者删除。

在计算机系统中既然要保存凭证,必然会涉及数据的存储,所以理论上只要能将用户凭证的数据保存下来,就可以实现用户登录的保持,比如保存登录信息到文件系统、数据库甚至是内存等。

下面讲解如何通过 Session 实现用户登录的保持,Session 是一种用来解决客户端和服务器会话保持的方法,它是一个通用的概念,而非 Java 特有的,但用于 Web 开发的语言或者框架都会提供相应实现 Session 的工具方便开发者来实现对会话的控制。现在来尝试使用 Session 进行登录状态的保持,过程其实非常简单,只需要在之前编写处理登录的 handleLogin 类里添加记录 Session 的程序就可以了。打开 handleLogin 这个类,进一步编辑。在这一个类中实现登录和注销两种操作,这样可以省去再次创建一个新的类的麻烦。每个 Action 里面都默认调用 execute()这一方法,所以要做到实现多功能,通过客户端发送不同的参数就可以实现了。现在为客户端增加一个名为 type 参数,用来表示用户的操作是登录还是注销登录,当 type 的值为 login 时执行登录的操作,为 unlogin 时执行注销登录的操作。

```
package mypro;
import java.util.Map;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class handleLogin extends ActionSupport {
```

```
    private String email;
    private String password ;
    private String type;
    public String execute() throws Exception {
        if(type.equals("login")){
            Map session = ActionContext.getContext().getSession();
            session.put("email", email);
            session.put("password ", password );
            return "login";
        }
        if(type.equals("unlogin")){
            Map session = ActionContext.getContext().getSession();
            session.remove("email");
            session.remove("password ");
            return "unlogin";
        }
        return "wrong";
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword () {
        return password ;
    }
    public void setPassword (String password ) {
        this.password = password ;
    }
}
```

该文件中通过Map session = ActionContext.getContext().getSession();获取 session 的 map 对象，然后对其进行 Session 的修改操作，操作方法和操作普通的 map 对象没有什么区别。通过 put 方法写入了两个属性，分别为 email 和 password，它们的值为前台传送过来的参数。通过 remove 方法移除这两个值。如果前台没有传送 type 这个参数，则视为非法的方法。

这里的 return 的字符串为自定义字符串，你可以根据需要取相应的名字，它们在写 strtus.xml 配置文件的时候会用到，现在开始编辑该文件。修改原来添加的 handleLogin

的 Action。

```
<action name = "handleLogin" class = "mypro.handleLogin">
<result name = "login">/templets/loginResult.jsp</result>
<result name = "unlogin">/templets/unlogin.jsp</result>
<result name = "wrong">/templets/wrong.jsp</result>
</action>
```

Result 标签中可以通过指定不同的 name 值实现在 Action 执行完成后跳转到不同的模板页面。继续创建相应的模板页面，在 templets 文件下添加一个新的文件 unlogin.jsp 作为注销成功后的页面，编辑内容如下：

```
<%@ page language = "java" contentType = "text/html; charset = UTF - 8"
    pageEncoding = "UTF - 8" %>
<!DOCTYPE html PUBLIC " - //W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd ">
<html>
<head>
    <meta charset = "utf - 8">
    <link href = "leeui/style/leeui - base.css" type = "text/css" rel = "stylesheet">
    <style type = "text/css">
        body{text - align: center;}
        #login - box{width :300px;margin:0 auto;margin - top: 100px;}
    </style>
</head>
<body>
    <div id = "login - box">
        <h1>注销成功!</h1>
    </div>
</body>
</html>
```

然后再新建一个文件名为 wrong.jsp，作为缺少参数非法访问后的提示页面。

```
<%@ page language = "java" contentType = "text/html; charset = UTF - 8"
    pageEncoding = "UTF - 8" %>
<!DOCTYPE html PUBLIC " - //W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd ">
<head>
    <meta charset = "utf - 8">
    <link href = "leeui/style/leeui - base.css" type = "text/css" rel = "stylesheet">
    <style type = "text/css">
        body{text - align: center;}
        #login - box{width :300px;margin:0 auto;margin - top: 100px;}
    </style>
</head>
<body>
```

```
    <div id = "login - box">
        <h1 >缺少参数!</h1 >
    </div >
</body >
</html >
```

最后编辑之前编写的登录界面的模板,并在里面添加一个 type 参数。

```
<% @ page language = "java" contentType = "text/html; charset = UTF - 8"
    pageEncoding = "UTF - 8" %>
<! DOCTYPE html PUBLIC " - //W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd ">
<html >
<head >
    <meta charset = "utf - 8">
    <link href = "leeui/style/leeui - base.css" type = "text/css" rel = "stylesheet">
    <style type = "text/css">
        body{text - align: center;}
        # login - box{width :300px;margin:0 auto;margin - top: 100px;}
    </style >
</head >
<body >
    <div id = "login - box">
        <h1 >基于 struts2 的登录 DEMO</h1 >
        <form action = "handleLogin.action" meth od = "post">
        <input type = "hidden" name = "type" value = "login">
        <ul class = "lee - form - normal">
            <li >
                <div class = "lee - input">
                    <label >邮箱</label >
                    <input type = "text" name = "email" value = "">
                </div >
            </li >
            <li >
                <div class = "lee - input">
                    <label >密码</label >
                    <input type = "password " name = "password " value = "">
                </div >
            </li >
            <li >
                <input class = "lee - button" style = "width :80px" type = "submit" value = "提交">
            </li >
        </ul >
        </form >
    </div >
</body >
</html >
```

其中添加了一个类型为 hidden 的 input 标签，并将 value 赋值为 login，很多时候程序需要传送数据参数，但又不希望用户看到这个参数的时候就使用该标签，当用户提交表单的时候系统会将该参数作为参数之一传送到后台，而用户在编辑信息的时候并不会看到这个参数。

以上步骤全部完成后重启服务器，输入地址 http://localhost:8080/mypro/login.action 访问，看能否正常工作。可能读者会感到奇怪，执行效果和之前并没有什么区别，到目前为止执行效果确实不会有什么区别，因为程序只做了登录信息的录入工作，并没有进行权限控制的操作，接下来继续了解如何通过已经录入的 Session 信息来实现页面的权限访问控制。

3.3.7 使用拦截器阻止非法访问

前面已经将用户的访问凭证进行了记录，那么实现权限的控制就不是什么难事了，最直接的办法是在每个 Action 中添加 Session 信息的查询验证，并根据验证结果跳转到不同的页面，但在实际应用中这么做可能会造成大量的重复工作，并会给以后的维护带来不必要的麻烦，在 Struts2 框架中提供了一套拦截器机制，通过拦截器可以很方便地对各种页面的访问进行权限的控制，其工作原理如图 3.7-8 所示。

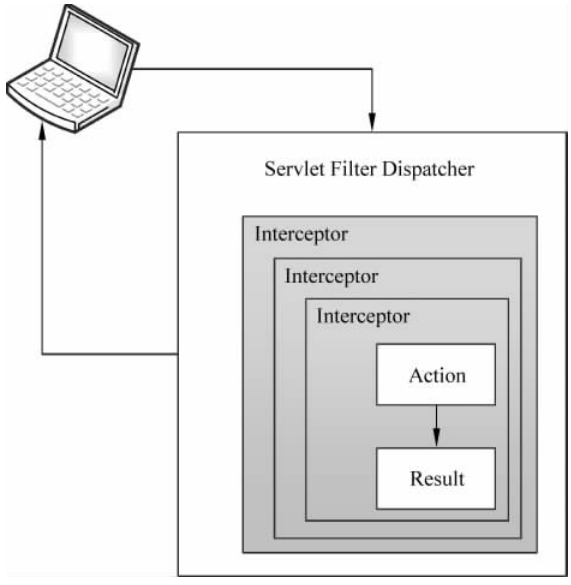

图 3.3-8 拦截器工作原理图

在 Struts2 介绍中已经知道了在访问 Action 之前首先会经过拦截器。形象地说，拦截器相当于一个引路人，可以控制访问究竟去执行哪一个 Action；同样，Action 本身也可以执行相同的作用，通过返回不同的值跳转到不同的 JSP 模板页面。

创建一个拦截器有如下步骤：

(1) 自定义一个实现 Interceptor 的接口(或者继承自 AbstractInterceptor)的类。

(2) 在 struts. xml 中注册上一步中定义的拦截器。

(3) 在需要使用的 Action 中引用上述定义的拦截器，为了方便也可将拦截器定义为默认的拦截器，这样在不加特殊声明的情况下所有的 Action 都被这个拦截器拦截。

```
public interfaceInterceptor extends Serializable {
    void destroy();
    void init();
    String intercept(ActionInvocationinvocation) throwsException;
}
```

Interceptor 接口声明了三个方法：

(1) Init 方法：Init 方法在拦截器类创建之后，在对 Action 镜像拦截之前调用，相当于一个 post-constructor 方法，使用这个方法可以给拦截器类做必要的初始化操作。

(2) Destory 方法：Destroy 方法在拦截器被垃圾回收之前调用，用来回收 init 方法初始化的资源。

(3) Intercept 方法：Intercept 方法是拦截器的主要拦截方法，如果需要调用后续的 Action 或者拦截器，只需要在该方法中调用 invocation. invoke()方法即可，在该方法调用的前后可以插入 Action 调用前后拦截器需要做的方法。如果不需要调用后续的方法，则返回一个 String 类型的对象即可，例如 Action. SUCCESS。

在 login 类的同级目录，即 mypro 包下新建一个拦截器类，名为 loginInterceptor，要注意的是拦截器的命名必须为自定义名＋Interceptor，并且必须和所作用的 Action 在同一级目录，编辑这个类的内容如下：

```
package mypro;
import java.util.Map;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionInvocation;
import com.opensymphony.xwork2.interceptor.AbstractInterceptor;
public class loginInterceptor extends AbstractInterceptor{
    public String intercept(ActionInvocation actionInvocation) throws Exception
    {
            Map session = actionInvocation.getInvocationContext().getSession();
            if(null == session.get("email")){
    return "unlogin";
            }
return actionInvocation.invoke();
    }
}
```

在这个拦截器类中，实现了查询 Session 中是否存在 email 变量，如果不存在则返回字符串 unlogin，否则返回 actionInvocation. invoke()继续执行相应的 Action。继续编辑

struts.xml 配置拦截器。

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE struts PUBLIC
    " - //Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtd s/struts - 2.3.dtd ">
< struts >
< constant name = "struts.enable.DynamicMeth odInvocation" value = "false" />
< constant name = "struts.devMode" value = "true" />
< package name = "default" namespace = "/" extends  = "struts - default">
    < interceptors >
        < interceptor name = "loginCheck" class = "myPro.loginInterceptor">
        </interceptor >
        < interceptor - stack name = "myStack">
            < interceptor - ref name = "loginCheck"/>
            < interceptor - ref name = "defaultStack"/>
        </interceptor - stack >
    </interceptors >
    < default - interceptor - ref name = "myStack"></default - interceptor - ref >
    < global - results >
        < result name = "unlogin">/templets/login.jsp </result >
        < result name = "error">/error.jsp </result >
    </global - results >
    < global - exception - mappings >
    < exception - mapping exception = "java.lang.Exception" result = "error"/>
    </global - exception - mappings >
    < action name = "helloWorld" class = "mypro.helloWorld">
        < result >/templets/helloWorld.jsp </result >
    </action >
    < action name = "login" class = "mypro.login">
        < result >/templets/login.jsp </result >
    </action >
    < action name = "handleLogin" class = "mypro.handleLogin">
        < result name = "login">/templets/loginResult.jsp </result >
        < result name = "unlogin">/templets/unlogin.jsp </result >
        < result name = "wrong">/templets/wrong.jsp </result >
    </action >
</package >
</struts >
```

这里通过<interceptors>声明一个拦截器，通过 global-results 定义了当身份验证失败时返回的 JSP 页面，当身份验证失败时将重新跳转到登录界面要求重新登录。为了更好地演示权限控制，再新建一个页面作为用户的主页，在 templets 下新建一个名为 home.jsp 的页面，编辑内容如下：

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

```
    pageEncoding = "UTF - 8" %>
<! DOCTYPE html PUBLIC " - //W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd ">
< html >
< head >
    < meta charset = "utf - 8">
    < link href = "leeui/style/leeui - base.css" type = "text/css" rel = "stylesheet">
    < style type = "text/css">
        body{text - align: center;}
        # login - box{width :300px;margin:0 auto;margin - top: 100px;}
    </style >
</head >
< body >
    < div class = "lee - container">
        < h1 >用户管理页面< a href = "mypro/handleLogin.action?type = unlogin">【注销登录】
</a ></h1 >
        < table class = "lee - table lee - table - strip lee - table - th color">
                < tr >
                    < th > ID </th >
                    < th >用户邮箱</th >
                    < th >密码</th >
                </tr >
                < tr >
                    < td > 1 </td >
                    < td >张三@test.com </td >
                    < td > 123456 </td >

                </tr >
                < tr >
                    < td > 2 </td >
                    < td >李四@test.com </td >
                    < td > 123456 </td >
                </tr >
                < tr >
                    < td > 3 </td >
                    < td >小王@test.com </td >
                    < td > 123456 </td >
                </tr >
            </table >
    </div >
</body >
</html >
```

在这个文件中创建了一张表格，然后使用这个表格显示用户的数据，现在仅放入一些测试数据进行演示。需要注意，该文件中通过一个 a 标签增加了注销登录的功能，本质上则是通过 GET 请求实现的。现在继续为这个模板页面创建一个名为 home 的 action 类，编辑内

容如下：

```
package mypro;
import com.opensymphony.xwork2.ActionSupport;
public class home extends ActionSupport {
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

继续配置 struts.xml 文件使得这个 action 可以运行，在其中添加一个新的 action。

```
<action name = "home" class = "mypro.home">
    <result>/templets/home.jsp</result>
    <interceptor-ref name = "defaultStack" />
    <interceptor-ref name = "loginCheck" />
</action>
```

现在在浏览器中输入地址 http://localhost:8080/mypro/home.action，这个页面本该显示的是新建的用户管理表单页面，但是现在却跳转到了登录页面，说明拦截器已经开始工作了。

输入任意的用户和密码登录，如图 3.3-9 所示。

图 3.3-9　输入登录信息

显示登录信息说明登录已经成功，程序也已经将用户的输入信息写入 Session，如图 3.3-10 所示。

图 3.3-10　登录信息注册成功

现在再次输入 home 的地址 http://localhost:8080/mypro/home.action 来看看能否访问，现在访问 home.action，已经可以访问用户管理界面了，说明配置的拦截器已经可以进行权限的管理了。现在单击注销登录，如图 3.3-11 所示。

图 3.3-11　注销登录信息

已经注销成功，说明程序已经将之前输入的用户信息从 Session 中删除，如果用户再次访问 home 页面，又会再次跳转到登录页面提示用户重新登录。到此为止已经基本实现了

用户的登录状态保持和基本的页面权限控制问题，当然这只是一个十分基本的实现，实际工程中的管理系统还有很多工作要做，比如用户信息的校验，针对不同的用户进行不同级别的权限管理，这些功能都需要用到数据库的知识，在学习完数据库的基本使用后会继续完善这个系统。Struts2 提供的不同拦截器说明如表 3.3-2 所示。

表 3.3-2 Struts2 提供的不同拦截器功能说明表

拦 截 器	名 字	说 明
AliasInterceptor	alias	在不同请求之间，将请求参数在不同名字间转换，请求内容不变
ChainingInterceptor	chain	让前一个 Action 的属性可以被后一个 Action 访问，现在和 chain 类型的 result(＜resulttype＝“chain”＞)结合使用
CheckboxInterceptor	checkbox	添加了 checkbox 自动处理代码，将没有选中的 checkbox 的内容设定为 false，而 html 默认情况下不提交没有选中的 checkbox
CookiesInterceptor	cookies	使用配置的 name，value 是指 cookies
ConversionErrorInterceptor	conversionError	将错误从 ActionContext 中添加到 Action 的属性字段中
CreateSessionInterceptor	createSession	自动创建 HttpSession，用来为需要使用到 HttpSession 的拦截器服务
DebuggingInterceptor	debugging	提供不同调试用的页面来展现内部数据状况
ExecuteandWaitInterceptor	execAndWait	在后台执行 Action，同时将用户带到一个中间的等待页面
ExceptionInterceptor	exception	将异常定位到一个画面
FileUploadInterceptor	fileUpload	提供文件上传功能
I18nInterceptor	i18n	记录用户选择的 locale
LoggerInterceptor	logger	输出 Action 的名字
MessageStoreInterceptor	store	存储或者访问实现 ValidationAware 接口的 Action 类出现的消息错误、字段错误等
ModelDrivenInterceptor	model-driven	如果一个类实现了 ModelDriven，将 getModel 得到的结果放在 ValueStack 中
ScopedModelDriven	scoped-model-driven	如果一个 Action 实现了 ScopedModelDriven，则这个拦截器会从相应的 Scope 中取出 model 调用 Action 的 setModel 方法将其放入 Action 内部
ParametersInterceptor	params	将请求中的参数设置到 Action 中去
PrepareInterceptor	prepare	如果 Acton 实现了 Preparable，则该拦截器调用 Action 类的 prepare 方法
ScopeInterceptor	scope	将 Action 状态存入 session 和 application 的简单方法
ServletConfigInterceptor	servletConfig	提供访问 HttpServletRequest 和 HttpServletResponse 的方法，以 Map 的方式访问
Static ParametersInterceptor	Static Params	从 struts.xml 文件中将＜action＞中的＜param＞中的内容设置到对应的 Action 中

续表

拦 截 器	名 字	说 明
RolesInterceptor	roles	确定用户是否具有 JAAS 指定的 Role，否则不予执行
TimerInterceptor	timer	输出 Action 执行的时间
TokenInterceptor	token	通过 Token 来避免双击
TokenSessionInterceptor	tokenSession	和 TokenInterceptor 一样，不过双击的时候把请求的数据存储在 Session 中
ValidationInterceptor	validation	使用 action-validation. xml 文件中定义的内容校验提交的数据
WorkflowInterceptor	workflow	调用 Action 的 validate 方法，一旦有错误就返回，重新定位到 INPUT 画面
ParameterFilterInterceptor	N/A	从参数列表中删除不必要的参数
ProfilingInterceptor	profiling	通过参数激活 profile

3.3.8 文件的上传

在任何一个基本的系统中都不免会遇到文件上传的问题，比如在 GIS 项目中大量的数据不可能通过手工输入完成，而是通过数据文件上传，通过程序导入到数据库。本节将继续讲解如何使用 Struts2 来实现图片文件的上传，在这个例子中将实现上传一张图片到后台并且显示出来。

我们先来分析一下该如何实现这个功能，上传部分自然是需要一个上传的动作类和一个视图页面，上传成功后还需要一个用于显示图片的视图页面。实际开发中通常会将业务的处理工作从 action 动作类中分离出来，但为了方便，在该例子中将在 action 中实现上传的功能。

上传文件的原理是使用 POST 数据提交，将文件数据流发送到服务器端，服务器将该文件保存到指定的路径下，同时将路径地址返回给预览页面。

现在新建一个类，命名为 HandleUploadFile，这个类用来接收上传的文件并保存到指定文件夹，编辑内容如下：

```
package mypro;
import java.io.File;
import org.apache.commons.io.FileUtils;
import org.apache.struts2.ServletActionContext;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;
public class HandleUploadFile extends ActionSupport{
private File image;                          //上传的文件
private String imageFileName;                //文件名称
private String imageContentType;             //文件类型
private String filePath ;                    //文件保存的路径
public String execute() throws Exception {
```

```
        String realpath = ServletActionContext.getServletContext().getRealPath ("/uploads");
        System.out.println("realpath : " + realpath );
            if (image != null) {
              File savefile = new File(new File(realpath ), imageFileName);
              if (!savefile.getParentFile().exists())
                savefile.getParentFile().mkdirs();
              FileUtils.copyFile(image, savefile);
              ActionContext.getContext().put("message", "文件上传成功");
              filePath = "uploads/" + imageFileName;
            }
            return "success";
        }
        public String getFilePath () {
            return filePath ;
        }
        public void setFilePath (String filePath ) {
            this.filePath = filePath ;
        }
        public File getImage() {
            return image;
        }
        public void setImage(File image) {
            this.image = image;
        }
        public String getImageFileName() {
            return imageFileName;
        }
        public void setImageFileName(String imageFileName) {
            this.imageFileName = imageFileName;
        }
        public String getImageContentType() {
            return imageContentType;
        }
        public void setImageContentType(String imageContentType) {
            this.imageContentType = imageContentType;
        }
    }
```

在上传的 action 类中，一般至少要定义三个变量用于接收前台的参数：首先是 File 类型的成员变量，它用于接收前台传输的文件，可以任意命名；其次分别是两个 String 类型的变量，用于接收文件的名称和类型，它们的命名必须和接收文件的参数名保持统一，这个文件中将 File 命名为 image，所以其他两个参数分别命名为 imageFileName 和 imageContentType。

通过 ServletActionContext. getServletContext(). getRealPath ("/uploads")可以获取网站应用运行的实际路径，uploads 是定义的放置上传文件的目录，需要在 WebContent 下

新建这个文件夹。

通过 FileUtils.copyFile(image, savefile)方法可以很方便地保存文件。再新建一个action类来显示上传页面,命名为 uploadFile,编辑内容如下:

```
package mypro;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;public class UploadFile extends ActionSupport{
    public String execute() throws Exception {
        return "success";
    }
}
```

同时再创建一个上传的模板页面,命名为 uploadFile.jsp,并编辑内容如下:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<title>文件上传</title>
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta charset="utf-8">
        <link href="leeui/style/leeui-base.css" type="text/css" rel="stylesheet">
        <style type="text/css">
            body{text-align: center;}
            #upload{width :300px;margin:0 auto;margin-top: 100px;}
        </style>
</head>
<body>
    <div id="upload">
<form action="uploadFile.action" enctype="multipart/form-data" meth od="post">
    <label>选择文件: </label><input type="file" name="image">
<input type="submit" value="上传" />
</form>
<br/>
<s:fielderror />
</div>
</body>
```

除了上传的模板页面,还需要新建一个页面用于上传后文件的显示,新建文件displayUpload.jsp,需要注意,在这个文件中 form 的 action 属性要填写定义的处理图片上传的 action 类地址 handleUploadFile.action,同时选择图片的 input 标签的 name 属性必须和后台接收的参数名一致,这是因为在 handleUploadFile 中接收图片的成员变量名为

image,所以这里的 name 属性必须设置为 image 才能保证数据接收,全部文件内容编辑内容如下:

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<title>文件上传</title>
<meta http-equiv="pragma" content="no-cache">
<meta http-equiv="cache-control" content="no-cache">
<meta http-equiv="expires" content="0">
<meta charset="utf-8">
        <link href="leeui/style/leeui-base.css" type="text/css" rel="stylesheet">
        <style type="text/css">
            body{text-align: center;}
            #upload{width :300px;margin:0 auto;margin-top: 100px;}
        </style>
</head>
<body>
    <div id="upload">
    <h1>上传文件 DEMO</h1>
<form action="handleUploadFile.action" enctype="multipart/form-data" method="post">
<ul class="lee-form-normal">
    <li>
                    <input class="" type="file" name="image" value="选择文件">
                </li>
                <li>
                    <input class="lee-button" type="submit" value="确定">
                </li>
            </ul>
</form>
<br/>
<s:fielderror />
</div>
</body>
</html>
```

再创建一个 displayFile.jsp 文件用于上传图片的显示,在该文件中通过 struts2 提供的 property 标签输出 handleUploadFile 类中的 filePath 变量来显示图片,文件完整内容编辑如下:

```
<%@ taglib prefix="s" uri="/struts-tags"%>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
```

```
<!DOCTYPE html PUBLIC " - //W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd ">
<html>
<head>
<title>文件上传</title>
<meta http - equiv = "pragma" content = "no - cache">
<meta http - equiv = "cache - control" content = "no - cache">
<meta http - equiv = "expires" content = "0">
<meta charset = "utf - 8">
        <link href = "leeui/style/leeui - base.css" type = "text/css" rel = "stylesheet">
        <style type = "text/css">
            body{text - align: center;}
            #upload{width :500px;margin:0 auto;margin - top: 100px;}
        </style>
</head>
<body>
    <div id = "upload">
        <h1>文件的相对路径为: <s:property value = "filePath " /></h1>
    <img src = "<s:property value = "filePath " />" width = "300px;" border = "0"/>
</div>
</body>
</html>
```

继续修改 struts.xml 配置文件，添加两个新的 action，分别是 handleUploadFile 和 uploadFile，需要注意，这里没有对图片上传失败作返回处理，实际应用中还应该对图片上传的异常做相应的返回处理，当上传处理完成时会将图片的相对路径传送到 displayUpload.jsp 中用于图片显示。

```
<action name = "uploadFile" class = "mypro.UploadFile">
    <result name = "success">/templets/uploadFile.jsp</result>
</action>
<action name = "handleUploadFile" class = "mypro.HandleUploadFile">
    <result name = "success">/templets/displayUpload.jsp</result>
</action>
```

现在打开浏览器，输入地址 http://localhost:8080/mypr/uploadFile.action，可以看到以下界面，选择一张图片上传，如图 3.3-12 所示。

如果可以看到上传的图片能够正常显示出来，说明上传功能已经可以正常工作了，如图 3.3-13 所示。

在实际项目，比如在 GIS 的项目中，不可避免会遇到上传文件的需求，比如某个地图要素的图片上传，或者一些附件文件的上传等，以上案例仅仅实现了图片的上传和保存，还不能满足实际项目的需求，在实际应用中还需要将文件保存的路径放置到关系型数据库中以便今后查找，上面的例子中虽然可以在上传后将文件正确显示出来，但是今后查找却面临困难，因为靠人记忆路径去查看文件是不切实际的。

图 3.3-12　上传文件界面

图 3.3-13　成功上传后显示所上传的图片

到这里，关于 struts2 的基本使用就基本介绍完了，后面的章节会学习数据库开发的基础知识。

3.4 Hibernate 框架的使用

3.4.1 配置数据库连接

在使用 Hibernate 之前，必须知道 Hibernate 是一个持久化框架，它用于承担一些数据库的通用操作，可以将开发者从数据库的底层操作解放出来，但归根结底数据库的连接方式是一样的，通常都会通过 JDBC 来连接数据库。什么是 JDBC 呢？JDBC 的全称为 Java Data Base Connectivity，它是一种执行 SQL 语句的 Java API，所以不论是直接对数据库进行底层操作，还是通过数据库框架来实现数据库的编程，其实都是通过 JDBC 接口来实现的。

既然如此，在使用 Hibernate 之前就要下载 JDBC，它是进行数据库编程的基石。进入 postgresql 的官方网站下载 JDBC。地址是 http://jdbc.postgresql.org/download.html。

下载到的文件中可能包含很多文件，我们只需要将其中的 jar 文件复制到工程 WebContent/WEB-INFO/lib 下就可以使用了。

在继续编写程序之前，先在数据库中建立一张记录用户数据的表，结构如表 3.3-3 所示。

表 3.3-3 表的结构

字 段 名	数 据 类 型	作 用
Id	整形	主键
Email	字符串	保存邮箱
Password	字符串	保存密码

这里只记录了之前案例中用户登录输入的信息，没有增加其他信息，而在实际工程中数据量肯定会多很多，但操作的原理基本是一样的。这里的数据类型采用中文标示，原因是不同的数据库中相同的数据类型命名会有所不同，在 GIS 开发项目中可以选择很多数据库，比如 Oracle、MySQL 等。GIS 开发项目较其他项目不同的一点是数据库会涉及空间数据的保存，这点需要注意。

在下载完 JDBC 后，还需要下载 Hibernate，进入官方网站 http://hibernate.org/，下载 HibernateORM。解压下载的压缩包后会看到以下的目录结构：Lib 文件夹里包含了开发程序需要用的 Java 包，打开 Lib 文件夹，里面会有很多子文件夹，将 required 文件夹中的文件复制到工程中的 WEB-INF/lib 文件夹下。最后，在 Eclipse 中刷新工程以便让工程识别到新添加的包。现在，最基本的工作已经完成了。

3.4.2 建立持久化类

前面已经介绍了持久化的概念，持久化的作用是将对数据库的操作抽象为对类的操作，这种对数据的操作方式更加符合编程思路和习惯，所以创建相关的类必不可少。接下来创

建一个持久化类，可以这么理解：一个持久化类对应着数据库中的一个表，程序对这个类实例化就相当于对应数据表中的一条记录，对该实例操作的同时 Hibernate 就会相应地对数据库进行操作。

现在新建一个 user 类，编辑内容如下：

```
package mypro;
public class user {
    private int id;
    private String email;
    private String password ;
    public int getId() {
        return id;
    }
    public void setId(int id) {
        this.id = id;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword () {
        return password ;
    }
    public void setPassword (String password ) {
        this.password = password ;
    }
}
```

可以看到，建立持久化其实非常简单，只要定义类的成员变量一一对应数据库中的字段，并设置好对应的 get 和 set 方法即可。需要注意，类中的数据类型必须和数据库中的数据类型保持一致，否则会出现错误。

在这个案例中使用的是手工方式建立实体化类，但在实际工程中会建立很多包含大量字段的数据表，可以通过相应的工具来完成这个工作。

3.4.3 配置映射文件

在建立完实体类后，还需要配置映射文件，映射文件的作用是告诉程序实体类和数据库表的映射关系，好比一张地图，失去了这张地图程序便无法得知实体类的操作该对应执行到哪张表中。

映射文件一般都命名为类名.hbm.xml，放置在 src 目录下，现在在 src 文件夹下新建一个文件命名为 user.cfg.xml，并编辑内容如下：

```
<?xml version = "1.0"?>
<! DOCTYPE hibernate - mapping PUBLIC " - //Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://www.hibernate.org/dtd /hibernate - mapping - 3.0.dtd ">
< hibernate - mapping >
< class name = "mypro.user" table  = "users">
    < id name = "id" type = "int">
        < column name = "id" />
        < generator class = "identity" />
    </id >
    < property name = "email" type = "String ">
        < column name = "email" />
    </property >
    < property name = "password " type = "String ">
        < column name = "password " />
    </property >
</class >
</hibernate - mapping >
```

Class 标签的 name 属性对应着建立的类名，table 属性对应着数据中的表名，property 标签则用来配置类成员属性和数据表字段的对应关系。

在配置好映射文件后，还需要建立一个 Hibernate 的配置文件，在这个文件中将进行 Hibernate 的基本数据库连接配置。通常将该配置文件命名为 Hibernate. cfg. xml，并放置在 src 文件夹下，Hibernate 通过该文件的配置数据连接到数据库，并读取相关的映射文件。新建该文件，编辑内容如下：

```
<?xml version = "1.0" encoding = "utf - 8"?>
<! DOCTYPE hibernate - configuration PUBLIC
" - //Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://www.hibernate.org/dtd /hibernate - configuration - 3.0.dtd ">
< hibernate - configuration >
< session - factory >
< property name = "hibernate.bytecode.use_reflection_optimizer"> false </property >
< property name = "hibernate.connection.driver_class"> org.postgresql.Driver </property >
< property name = "hibernate.connection.url"> jdbc:postgresql://localhost:5432/mydb </property >
< property name = "hibernate.connection.username">这里填写数据库的用户名</property >
< property name = "hibernate.connection.password ">这里填写数据的密码</property >
< property name = "hibernate.dialect"> org.hibernate.spatial.dialect.postgis.PostgisDialect
</property >
< property name = "hibernate.format_sql"> true </property >
< property name = "hibernate.search.autoregister_listeners"> false </property >
< property name = "hibernate.show_sql"> true </property >
< property name = "hibernate.connection.pool_size"> 20 </property >
< property name = "hibernate.proxool.pool_alias"> pool1 </property >
< property name = "hibernate.max_fetch_depth "> 1 </property >
< property name = "hibernate.jdbc.batch_versioned_data"> true </property >
< property name = "hibernate.jdbc.use_streams_for_binary"> true </property >
```

```
< property name = "hibernate. cache. region_prefix"> hibernate. test </property >
< property name = " hibernate. cache. provider_class"> org. hibernate. cache. Hashtable CacheProvider
</property >
< mapping resource = "user. hbm. xml"/>
</session - factory >
</hibernate - configuration >
```

这里配置了 Hibernate 的一些基本参数，现在不需要去深究里面所有标签的意思，下面简单介绍几个比较重要的标签。

```
< property name = "hibernate. connection. driver_class"> org. postgresql. Driver </property >
```

配置正确的 JDBC。

```
< property name = "hibernate. connection. url"> jdbc:postgresql://localhost:5432/mydb </property >
```

这里设置正确的数据库地址，端口和数据库名。

```
< property name = "hibernate. connection. username">这里填写数据库的用户名</property >
< property name = "hibernate. connection. password ">这里填写数据的密码</property >
```

设置连接数据库时的用户和密码，这里的信息和安装数据库时候输入的信息保持一致。到目前为止，基本的底层工作已经完成了，现在可以开始对业务逻辑编写程序了。

3.4.4 写入数据库实例

本节在之前案例的基础上，以添加一个新用户为例，来实现通过 Hibernate 进行数据库的数据写入。

先创建一个类用于数据库的基本操作。新建一个类，名为 userDao，这个类主要用于实现一些数据库的基本操作，现在实现一个 insert 操作的方法用于进行用户数据的插入工作。

```
package mypro;import java. util. List; import org. hibernate. Query;
import org. hibernate. Session;
import org. hibernate. SessionFactory;
import org. hibernate. Transaction;
import org. hibernate. cfg. Configuration; import mypro. user;public class usersDao {
    Static SessionFactory sessionFactory;
    Static Session session ;
    Static Transaction tx ;                    //插入
    public void insert(user user)
    {
        init();
        session. save(user);
        close();
    }
    @SuppressWarnings("deprecation")
    private void init()
```

```
    {
        sessionFactory = new Configuration().configure().buildSessionFactory();
        session = sessionFactory.openSession();
        tx = session.beginTransaction();
    }
    private void close()
    {
        tx.commit();
        session.close();
        sessionFactory.close();
    }
}
```

再新建一个名为 userService 的类，这个类主要用于业务逻辑的处理。单从编程的角度讲，在 userDao 中就可以实现全部的功能，但将永久层和业务层分离利于之后工作的展开。编辑这个类的内容如下：

```
package mypro;import java.util.List;import mypro.usersDao;
import mypro.user;
public class userService {
    private usersDao dao;
    public userService(){
        dao = new usersDao();
    }
    /*
     * 添加一个新的用户
     */
    public boolean addUser(user user){
        dao.insert(user);
        return true;
    }
}
```

这个类仅仅是调用了 userDao 中的方法，可能 userService 这个类会显得多余，但实际工作中业务逻辑会比这个更加复杂，比如在添加新用户时候要检测用户是否重复，数据校验，异常的捕获和处理等通常都会在业务层的类中实现。

接下来创建一个 action 类用于注册请求的接受。创建新类，命名为 handleRegister，编辑内容如下：

```
package mypro;import java.util.Map;
import com.opensymphony.xwork2.ActionContext;
import com.opensymphony.xwork2.ActionSupport;
import mypro.userService;
import mypro.user;public class handleRegister extends ActionSupport {
    private String email;
    private String password ;
```

```
    public String execute() throws Exception {
        userService us = new userService();
        user user = new user();
        user.setEmail(email);
        .setPassword (password );
        us.addUser(user);
        return SUCCESS;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public String getPassword () {
        return password ;
    }
    public void setPassword (String password ) {
        this.password = password ;
    }
}
```

这个类中分别定义 email 和 password 的成员变量，用于接收前台传送来用户输入的参数。同时实例化一个 userService 方法，通过其 addUser()方法插入一个新的用户。

现在后台处理插入的工作已经完成了，接下来处理视图部分，建立一个注册的页面和对应的动作类，新建一个名为 register 的 action 类，编辑内容如下：

```
package mypro;import com.opensymphony.xwork2.ActionSupport;import mypro.userService;
import mypro.user;public class register extends ActionSupport {
    private String type;
    public String execute() throws Exception {
        return type;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
}
```

这个类中没有进行多余的逻辑操作，这个类的主要作用是用于显示注册页面。现在新建一个名为 register 的模板页面，编辑内容如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
```

```
loose.dtd ">
<html>
<head>
    <meta charset = "utf-8">
    <link href = "leeui/style/leeui-base.css" type = "text/css" rel = "stylesheet">
    <style type = "text/css">
        body{text-align: center;}
        #login-box{width :300px;margin:0 auto;margin-top: 100px;}
    </style>
</head>
<body>
    <div id = "login-box">
        <h1>注册新的用户</h1>
        <form action = "handleRegister.action" meth od = "post">
        <input type = "hidden" name = "type" value = "register">
        <ul class = "lee-form-normal">
            <li>
                <div class = "lee-input">
                    <label>邮箱</label>
                    <input type = "text" name = "email" value = "">
                </div>
            </li>
            <li>
                <div class = "lee-input">
                    <label>密码</label>
                    <input type = "password " name = "password " value = "">
                </div>
            </li>
            <li>
                <input class = "lee-button" style = "width :80px" type = "submit" value = "提交">
            </li>
        </ul>
        </form>
    </div>
</body>
</html>
```

最后在 struts.xml 中配置之前编写的 action 类，在文件中添加以下代码：

```
<action name = "register" class = "mypro.register">
<result name = "register">/templets/register.jsp</result>
</action>
<action name = "handleRegister" class = "mypro.handleRegister">
    <result name = "success" type = "redirectAction">
    <param name = "namespace">/</param>
    <param name = "action name">home</param>
</result>
```

```
<result name = "update" type = "redirectAction">
    <param name = "namespace">/</param>
    <param name = "action name"> home </param>
</result>
</action>
```

这里用到了 type 为 redirectAction 的 result，这个类型的 result 的作用是跳转到另外一个 action，当用户添加完成后页面会直接跳转到 home. action 的用户列表页面。但目前还没有做数据库读取的部分，所以新添加的数据还不能显示出来，但可以通过直接查看数据库中的内容查看数据是否添加成功。

现在打开浏览器输入地址 http://localhost:8080/mypro/register. action? type = register，可以看到显示除了添加新用户的注册界面，现在随意输入一个测试用的账户信息，并添加，如图 3.4-1 所示。

图 3.4-1 注册一个新的用户

3.4.5 读取数据库实例

现在已经可以向数据库中添加数据了，本节将讲解如何读取数据库中的数据并通过一个表格显示出来。现在进一步修改之前编写的 userDao 和 userService 类，在其中增加读取数据库的方法。首先编辑 userDao 如下：

```
package mypro;import java.util.List; import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
```

```
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration; import mypro.user;public class usersDao {
    Static SessionFactory sessionFactory;
    Static Session session ;
    Static Transaction tx ;
    //查询所有
    public List<user> loadAll(){
        init();
        Query query = session.createQuery("from user");
        <user> list = query.list();
        close();
        return list;
    }    //插入
    public void insert(user user)
    {
        init();
        session.save(user);
        close();
    }
    @SuppressWarnings("deprecation")
    private void init()
    {
        sessionFactory = new Configuration().configure().buildSessionFactory();
        session = sessionFactory.openSession();
        tx = session.beginTransaction();
    }
    private void close()
    {
        tx.commit();
        session.close();
        sessionFactory.close();
    }
}
```

其中添加了一个查询所有记录的方法，查询并返回类型为 list 的数据，这里还用到了 hibernatesession 的 createQuery 方法，通过这个方法可以执行特定的 SQL 语句。

现在继续编辑 userService 类，在其中增加一个查询的方法调用 userDao 中的方法，编辑内容如下：

```
package mypro;import java.util.List;import mypro.usersDao;
import mypro.user;
public class userService {
    private usersDao dao;
    public userService(){
        dao = new usersDao();
    }
```

```
    /*
     * 添加一个新的用户
     */
    public boolean addUser(user user){
        dao.insert(user);
        return true;
    }
    /*
     * 查询所有用户
     */
    public List<user> getAll(){
        return dao.loadAll();
    }
}
```

到目前为止，已经实现了从数据库中查询数据的部分了，接下来要做的是将查询出来的数据显示到指定的页面中。因为之前已经创建了一个静态的 home.jsp 页面用于用户列表的显示，现在需在此基础上修改增加数据的动态显示。打开 home 的 action 类，编辑内容如下：

```
package mypro;
import java.util.List;import com.opensymphony.xwork2.ActionSupport;public class home extends
ActionSupport {
    private List<user> users;
    private userService us;
    public String execute() throws Exception {
        us = new userService();
        users = us.getAll();
        return SUCCESS;
    }
    public List<user> getUsers() {
        return users;
    }
    public void setUsers(List<user> users) {
        this.users = users;
    }
}
```

在这个类中实例化了一个 userService 对象，并调用了查询方法将结果赋值给了该类的成员变量 users，该变量是一个 list 类型，通过其 get 方法可以在 jsp 页面中对数据进行调用。继续编辑 home.jsp 页面，编辑内容如下：

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
```

```
loose.dtd ">
<html>
<head>
    <meta charset = "utf - 8">
    <link href = "leeui/style/leeui - base.css" type = "text/css" rel = "stylesheet">
    <style type = "text/css">
        body{text - align: center;}
        # login - box{width :300px;margin:0 auto;margin - top: 100px;}
    </style>
</head>
<body>
    <div class = "lee - container">
        <h1>用户管理页面<a href = "handleLogin.action?type = unlogin">【注销登录】</a><a
href = "register.action?type = register">【添加用户】</a></h1>
        <table class = "lee - table lee - table - strip lee - table - th color">
            <tr>
                <th>ID</th>
                <th>用户邮箱</th>
                <th>密码</th>
            </tr>
            <s:iterator value = "users" var = "user">
            <tr>
                <td><s:property value = "%{id}"/></td>
                <td><s:property value = "%{email}"/></td>
                <td><s:property value = "%{password }"/></td>
            </tr>
            </s:iterator>
        </table>
    </div>
</body>
</html>
```

这里通过 struts2 的 iterator 标签实现了对 users 的遍历显示，在实际工程中经常要查询批量的数据，因此该标签在实际应用中非常有用。通过该标签可以实现对数组 Map、List 等数据的遍历，这里简单介绍一下遍历 List 的方法，遍历其他类型数据基本大同小异。

```
<s:iterator value = "遍历对象名">
    <tr>
        <td><s:property value = "%{遍历对象的成员变量}"/></td>
    </tr>
</s:iterator>
```

现在打开浏览器，输入地址 http://localhost:8080/mypro/home.action，可以看到之前添加的数据已经被显示出来了，如图 3.4-2 所示。

图 3.4-2 遍历显示数据库的数据

3.4.6 数据库删除实例

现在已经完成了数据库的数据添加和显示，接下来实现数据的删除。新建 action 类，命名为 userManager，编辑内容如下：

```
package mypro;import com.opensymphony.xwork2.ActionSupport;
import mypro.userService;
import mypro.user;public class userManager extends ActionSupport {
    private String type;
    private int userId;
    private userService us;
    public String execute() throws Exception {
        us = new userService();
        if(type.equals("delete")){
            us.delete(userId);
        }
        return type;
    }
    public int getUserId() {
        return userId;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
```

```
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
}
```

当删除某个用户的时候会将该用户的主键 ID 传送到这个 action 中，通过主键进行查找并删除相关的数据。相应地，继续编辑 userService 和 userDao 这两个类，在其中分别添加相应的方法，编辑 userDao 内容如下：

```
package mypro;import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import mypro.user;public class usersDao {
Static SessionFactory sessionFactory;
Static Session session ;
Static Transaction tx ;                    //查询所有
public List<user> loadAll(){
    init();
    Query query = session.createQuery("from user");
    List<user> list = query.list();
    close();
    return list;
}
//插入
public void insert(user user)
{
    init();
    session.save(user);
    close();
}
    //删除
public boolean delete(int id){
    init();
    session.delete((user) session.get(user.class, id));
    close();
    return true;
}
@SuppressWarnings("deprecation")
private void init() {
    sessionFactory = new Configuration().configure().buildSessionFactory();
    session = sessionFactory.openSession();
```

```
        tx = session.beginTransaction();
    }
    private void close() {
        tx.commit();
        session.close();
        sessionFactory.close();
        }
    }
```

这里通过 hibernate session 的 delete 方法可以删除一条数据。继续编辑 userService。

```
package mypro;import java.util.List;
import mypro.usersDao;
import mypro.user;
public class userService {
    private usersDao dao;
    public userService(){
    dao = new usersDao();
    }
    /*
     * 添加一个新的用户
     */
    public boolean addUser(user user){
        dao.insert(user);
        return true;
    }
    /*
     * 查询所有用户
     */
    public List<user> getAll(){
        return dao.loadAll();
    }
    /*
     * 删除一个用户
     */
    public boolean delete(int id){
        return dao.delete(id);
    }
}
```

在 struts.xml 文件中添加对应的 action 配置，在用户被成功删除后页面将直接跳转回用户列表页面。

```
<action name = "userManager" class = "mypro.userManager">
<result name = "delete" type = "redirectAction">
    <param name = "namespace">/</param>
    <param name = "action name"> home </param>
```

```
</result>
</action>
```

在用户列表的模板页面中添加相应的删除按钮，修改后的 home.jsp 页面如下：

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
    <meta charset="utf-8">
    <link href="leeui/style/leeui-base.css" type="text/css" rel="stylesheet">
    <style type="text/css">
        body{text-align: center;}
        #login-box{width :300px;margin:0 auto;margin-top: 100px;}
    </style>
</head>
<body>
    <div class="lee-container">
        <h1>用户管理页面<a href="handleLogin.action?type=unlogin">【注销登录】</a><a
href="register.action?type=register">【添加用户】</a></h1>
        <table class="lee-table lee-table -strip lee-table -th color">
            <tr>
                <th>ID</th>
                <th>用户邮箱</th>
                <th>密码</th>
                <th>操作</th>
            </tr>
            <s:iterator value="users" var ="user">
            <tr>
                <td><s:property value="%{id}"/></td>
                <td><s:property value="%{email}"/></td>
                <td><s:property value="%{password}"/></td>
                  <td><a href="/mypro/userManager.action?type=delete&userId=<s:
property value="%{id}"/>">删除信息</a>|

                </td>
            </tr>
            </s:iterator>
        </table>
    </div>
</body>
</html>
```

继续在列表页面中增加编辑功能，通过增加一个 a 标签来实现对用户的删除，前面已经

讲过前台发送请求到后台主要通过 get 或者 post 两种方法，这里通过 a 标签可以更加方便地实现用户删除接口的调用，在遍历显示用户数据的时候程序动态地将用户的主键 ID 作为参数连接在链接地址的后面。当用户单击该链接时会直接跳转到相应的地址进行删除功能的操作。

现在重新发布程序，在浏览器中测试一下删除功能可否正常工作，如图 3.4-3 所示。

图 3.4-3　删除一条记录

单击删除信息可以看到用户信息已经被删除了，如图 3.4-4 所示。

图 3.4-4　成功删除一条记录

3.5 Spring 框架的使用

3.5.1 Spring 简介

Spring 是一个开源框架，它由 RodJohnson 创建。它是为了解决企业应用开发的复杂性而创建的。Spring 使用基本的 JavaBean 来完成以前只能由 EJB 完成的事情。然而，Spring 的用途不仅限于服务器端的开发。从简单性、可测试性和松耦合的角度而言，任何 Java 应用都可以从 Spring 中受益。Spring 是一个轻量级的控制反转(IoC)和面向切面(AOP)的容器框架。

1. 轻量

就大小与开销两方面而言，Spring 都是轻量的。完整的 Spring 框架可以在一个大小只有 1MB 多的 JAR 文件里发布。并且 Spring 所需的处理开销也是微不足道的。此外，Spring 是非侵入式的：Spring 应用中的对象不依赖于 Spring 的特定类。

Spring 通过一种称作控制反转(IoC)的技术促进了松耦合。当应用了 IoC，一个对象依赖的其他对象会通过被动的方式传递进来，而不是这个对象自己创建或者查找依赖对象。你可以认为 IoC 与 JNDI 相反——不是对象从容器中查找依赖，而是容器在对象初始化时不等对象请求就主动将依赖传递给它。

2. 面向切面

Spring 提供了面向切面编程的丰富支持，允许通过分离应用的业务逻辑与系统级服务(例如审计(auditing)和事务(transaction)管理)进行内聚性的开发。应用对象只实现完成业务逻辑而已。它们并不负责(甚至是意识)其他的系统级关注点，例如日志或事务支持。

3. 容器

Spring 包含并管理应用对象的配置和生命周期，在这个意义上它是一种容器，用户可以配置每个 bean，基于一个可配置原型(prototype)bean 可以创建一个单独的实例或者每次需要时都生成一个新的实例。然而，Spring 不应该被混同于传统的重量级的 EJB 容器，它们经常是庞大与笨重的。

Spring 可以将简单的组件配置，组合成为复杂的应用。在 Spring 中，应用对象被声明式地组合，典型的是在一个 XML 文件里。Spring 也提供了很多基础功能(事务管理、持久化框架集成等)，将应用逻辑的开发留给了用户。

所有 Spring 的这些特征使用户能够编写更干净、更易管理并且更便于测试的代码。它们也为 Spring 中的各种模块提供了基础支持。

3.5.2 Spring 的配置

本章主要讲解 Struts2、Hibernate、Spring 的结合使用，对版本协调有一定要求，如果版本匹配不恰当也可能会造成错误。为了避免不必要的麻烦，可以直接使用下载 struts2 包中 Lib 文件夹下的 Spring 包。

首先将 lib 下 Spring 开头的 Jar 文件复制到项目工程的 WEB-INF/lib 文件夹下，如图 3.5-1 所示。

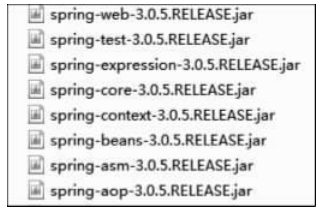

图 3.5-1　需要用到的 Spring 包

同时还需要将 Struts2 的 Spring 插件包 struts2-spring-plugin-3.3.16.jar 放置到工程中，否则程序将无法正常工作。

为了让 Spring 开始工作，在 web.xml 中增加 Spring 的监听，新的 web.xml 编辑如下：

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< web - app id = "WebApp_9" version = "2.4" xmlns = "http://java.sun.com/xml/ns/j2ee" xmlns:xsi
 = "http://www.w3.org/2001/XMLSchema - instance" xsi:schemaLocation = "http://java.sun.com/
xml/ns/j2ee http://java.sun.com/xml/ns/j2ee/web - app_2_4.xsd"><display - name > Struts Blank
</display - name ><filter >
< filter - name > struts2 </filter - name >
< filter - class > org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
</filter - class >
</filter ><filter - mapping >
< filter - name > struts2 </filter - name >
< url - pattern >/ * </url - pattern >
</filter - mapping >
    <! -- spring 监听 -- >
    < context - param >
        < param - name > contextConfigLocation </param - name >
        < param - value >/WEB - INF/applicationContext.xml </param - value >
    </context - param >
    < listener >
        < listener - class >
            org.springframework.web.context.ContextLoaderListener
        </listener - class >
    </listener >
</web - app >
```

其中，＜param-value＞/WEB-INF/applicationContext.xml＜/param-value＞是 Spring 配置文件的路径，org.springframework.web.context.ContextLoaderListener 是 Spring 的监听类，它包含在之前引入的 spring 的包中。配置完成后重启项目，如果没有错误，就说明 Spring 已经正确地配置完成了。

3.5.3　Spring 和 Struts2、Hibernate 的整合

本节以实现用户信息更新为例讲解 Spring 的基本使用。首先要做的工作是创建相应的模板文件、Action 类等。编辑用户信息的第一步是完成用户的编辑界面，当用户单击编辑某一位用户的链接时需要从数据库中查询该用户的信息填充到用户的编辑表中，也就是数据查询和显示的实现。

首先编辑 userDao 和 userService，添加相应的查询和修改方法，编辑 usrDao 如下：

```
package mypro;import java.util.List;
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;
import mypro.user;public class usersDao {
    Static SessionFactory sessionFactory;
    Static Session session ;
    Static Transaction tx ;
    //读取
    public user get(int id)
    {
        init();
        user obj = (user) session.get(user.class, id);
        close();
        return obj;
    }
    //查询所有
    public List<user> loadAll(){
        init();
        Query query = session.createQuery("from user");
        List<user> list = query.list();
        close();
        return list;
    }
    //更新
    public void update(user user)
    {
        init();
    session.update(user);
    close();
    }
    //插入
    public void insert(user user)
    {
        init();
        session.save(user);
    close();
    }
    //删除
    public boolean delete(int id){
        init();
        session.delete((user) session.get(user.class, id));
        close();
```

```
        return true;
    }
    @SuppressWarnings("deprecation")
    private void init()
    {
        sessionFactory = new Configuration().configure().buildSessionFactory();
        session = sessionFactory.openSession();
        tx = session.beginTransaction();
    }
    private void close()
    {
        tx.commit();
        session.close();
        sessionFactory.close();
    }
}
```

这里通过调用 Hibernate 的 get 方法，并通过用户的主键 Id 查询其数据，通过 update 方法更新数据。在 userService 中增加相应的方法。

```
package mypro;import java.util.List;
import mypro.usersDao;
import mypro.user;
public class userService {
    private usersDao dao;
    public userService(){
        dao = new usersDao();
    }
    /*
     * 添加一个新的用户
     */
    public boolean addUser(user user){
        dao.insert(user);
        return true;
    }
    /*
     * 更新一个用户
     */
    public boolean updateUser(user user){
        dao.update(user);
        return true;
    }
    /*
     * 查询一个用户
     */
    public user getUser(int id){
        return dao.get(id);
```

```
    }
    /*
     * 查询所有用户
     */
    public List<user> getAll(){
        return dao.loadAll();
    }
    /*
     * 删除一个用户
     */
    public boolean delete(int id){
        return dao.delete(id);
    }
}
```

创建一个 Action 类，命名为 editUser，在这个类中通过 type 参数来区分是显示编辑界面还是提交修改数据请求。

```
package mypro;import com.opensymphony.xwork2.ActionSupport;
import mypro.userService;
import mypro.user;
public class editUser extends ActionSupport {
    private user user;
    private userService us;
    private int userId;
    private String type;
    public String execute() throws Exception {
        if(type.equals("update")){
            us.updateUser(user);
        }
        if(type.equals("edit")){
            user = us.getUser(userId);
            return "edit";
        }
        return type;
    }
    public int getUserId() {
        return userId;
    }
    public void setUserId(int userId) {
        this.userId = userId;
    }
    public user getUser() {
        return user;
    }
    public void setUser(user user) {
        this.user = user;
```

```
    }
    public userService getUs() {
        return us;
    }
    public void setUs(userService us) {
        this.us = us;
    }
    public String getType() {
        return type;
    }
    public void setType(String type) {
        this.type = type;
    }
}
```

仔细观察这个类和之前建立的类的区别。在这个类中用到了 userService 这个类，但是并没有通过 new 实例化一个对象，因为程序将通过 Spring 注入这个对象，这样做的好处是可以更清楚地管理各个类之间的关系，降低模块和模块之间的耦合。

在 WEB-INF 文件夹下新建一个文件命名为 applicationContext.xml，并编辑如下：

```
<?xml version = "1.0" encoding = "UTF - 8"?>
< beans xmlns = "http://www.springframework.org/schema/beans"
        xmlns:xsi = "http://www.w3.org/2001/XMLSchema - instance"
        xmlns:aop = "http://www.springframework.org/schema/aop"
        xmlns:tx = "http://www.springframework.org/schema/tx"
        xsi:schemaLocation = "
            http://www.springframework.org/schema/beans http://www.springframework.org/
schema/beans/spring - beans - 2.5.xsd
            http://www.springframework.org/schema/aop http://www.springframework.org/
schema/aop/spring - aop - 2.5.xsd
            http://www.springframework.org/schema/tx http://www.springframework.org/
schema/tx/spring - tx - 2.5.xsd">
        < bean name = "editUser" class = "mypro.editUser">
            < property name = "us">
                < ref bean = "userService"/>
            </property >
        </bean >
            < bean name = "userService" class = "mypro.userService">
        </bean >
</beans >
```

在这个配置文件中通过 bean 标签映射了相应的类，并通过 property 注入了该类中需要实例化的 userService 对象，在程序工作时无需手工 new 实例化对象，这些工作将全权交给 Spring 来完成。

最后，在 home.jsp 中添加编辑用户信息的链接，通过 get 方法跳转到对应的信息编辑

界面。编辑后的文件如下：

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
    <meta charset="utf-8">
    <link href="leeui/style/leeui-base.css" type="text/css" rel="stylesheet">
    <style type="text/css">
        body{text-align: center;}
        #login-box{width :300px;margin:0 auto;margin-top: 100px;}
    </style>
</head>
<body>
    <div class="lee-container">
        <h1>用户管理页面<a href="handleLogin.action?type=unlogin">【注销登录】</a><a
href="register.action?type=register">【添加用户】</a></h1>
        <table class="lee-table lee-table -strip lee-table -th color">
            <tr>
                <th>ID</th>
                <th>用户邮箱</th>
                <th>密码</th>
                <th>操作</th>
            </tr>
            <s:iterator value="users" var ="user">
            <tr>
                <td><s:property value="%{id}"/></td>
                <td><s:property value="%{email}"/></td>
                <td><s:property value="%{password}"/></td>
                <td><a href="/mypro/userManager.action?type=delete&userId=<s:
property value="%{id}"/>">删除信息</a>|
                <a href="/mypro/editUser.action?type=edit&userId=<s:property value
="%{id}"/>">编辑信息</a>
                </td>
            </tr>
            </s:iterator>
        </table>
    </div>
</body>
</html>
```

创建一个新的用户编辑模板页面，命名为 edit.jsp，编辑内容如下：

```
<%@ taglib prefix="s" uri="/struts-tags" %>
```

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
    pageEncoding="UTF-8"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
    <meta charset="utf-8">
    <link href="leeui/style/leeui-base.css" type="text/css" rel="stylesheet">
    <style type="text/css">
        body{text-align: center;}
        #login-box{width:300px;margin:0 auto;margin-top: 100px;}
    </style>
</head>
<body>
    <div id="login-box">
        <h1>编辑用户信息</h1>
        <form action="editUser.action" meth od="post">
        <input type="hidden" name="type" value="update">
        <input type="hidden" name="user.id" value="<s:property value="user.id" />">
        <ul class="lee-form-normal">
            <li>
                <div class="lee-input">
                    <label>邮箱</label>
                    <input type="text" name="user.email" value="<s:property value="
user.email" />">
                </div>
            </li>
            <li>
                <div class="lee-input">
                    <label>密码</label>
                     <input type="password " name="user.password " value="<s:property
value="user.password " />">
                </div>
            </li>
            <li>
                <input class="lee-button" style="width:80px" type="submit" value="提交">
            </li>
        </ul>
        </form>
    </div>
</body>
</html>
```

现在回头查看之前编写的 editUser，在其中并没有定义相关的 email 和 password 变量，而是直接定义了一个 user 对象，在 struts2 中允许以对象的方式传送数据，这个特性在数据繁多的项目中十分有用。

在编辑的模板文件中，我们定义相应的属性为 user 属性名，便可以直接接收整个 user，十分方便。最后配置 struts.xml 文件使得之前创建的 Action 可以正确运行，编辑后的文件如下：

```
<?xml version = "1.0" encoding = "UTF - 8" ?>
<! DOCTYPE struts PUBLIC
    " - //Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtd s/struts - 2.3.dtd ">
< struts >
< constant name = "struts.enable.DynamicMeth odInvocation" value = "false" />
< constant name = "struts.devMode" value = "true" />
< package name = "default" namespace = "/" extends  = "struts - default">
< interceptors >
    < interceptor name = "loginCheck" class = "mypro.loginInterceptor">
    </interceptor >
    < interceptor - stack name = "myStack">
        < interceptor - ref name = "loginCheck"/>
        < interceptor - ref name = "defaultStack"/>
    </interceptor - stack >
</interceptors >
< default - interceptor - ref name = "myStack"></default - interceptor - ref >
< global - results >
    < result name = "unlogin">/templets/login.jsp </result >
    < result name = "error">/error.jsp </result >
</global - results >
< action name = "helloWorld" class = "mypro.helloWorld">
    < result >/templets/helloWorld.jsp </result >
</action >
< action name = "home" class = "mypro.home">
    < result name = "success">/templets/home.jsp </result >
</action >
< action name = "uploadFile" class = "mypro.UploadFile">
    < result name = "success">/templets/uploadFile.jsp </result >
</action >
< action name = "handleUploadFile" class = "mypro.HandleUploadFile">
    < result name = "success">/templets/displayUpload.jsp </result >
</action >
< action name = "register" class = "mypro.register">
    < result name = "register">/templets/register.jsp </result >
</action >
< action name = "handleRegister" class = "mypro.handleRegister">
    < result name = "success" type = "redirectAction">
        param name = "namespace">/</param >
        < param name = "action name"> home </param >
    </result >
    < result name = "update" type = "redirectAction">
        < param name = "namespace">/</param >
```

```
            <param name = "action name"> home </param >
        </result >
</action >
< action name = "userManager" class = "mypro.userManager">
        < result name = "delete" type = "redirectAction">
            < param name = "namespace">/</param >
            < param name = "action name"> home </param >
        </result >
</action >
< action name = "editUser" class = "editUser">
        < result name = "edit">/templets/edit.jsp </result >
        < result name = "update" type = "redirectAction">
            < param name = "namespace">/</param >
            < param name = "action name"> home </param >
        </result >
</action >
</package >
< package name = "login" namespace = "/" extends  = "struts - default">
< action name = "login" class = "mypro.login">
        < result >/templets/login.jsp </result >
</action >
< action name = "handleLogin" class = "mypro.handleLogin">
        < result name = "login">/templets/loginResult.jsp </result >
        < result name = "unlogin">/templets/unlogin.jsp </result >
        < result name = "wrong">/templets/wrong.jsp </result >
</action >
</package >
</struts >
```

打开浏览器测试，首先添加一个新的测试用户，如图 3.5-2 所示。

图 3.5-2　使用 Spring 实现用户信息的编辑

单击编辑信息后可以看到正确的显示出用户的基本信息，如图 3.5-3 所示。

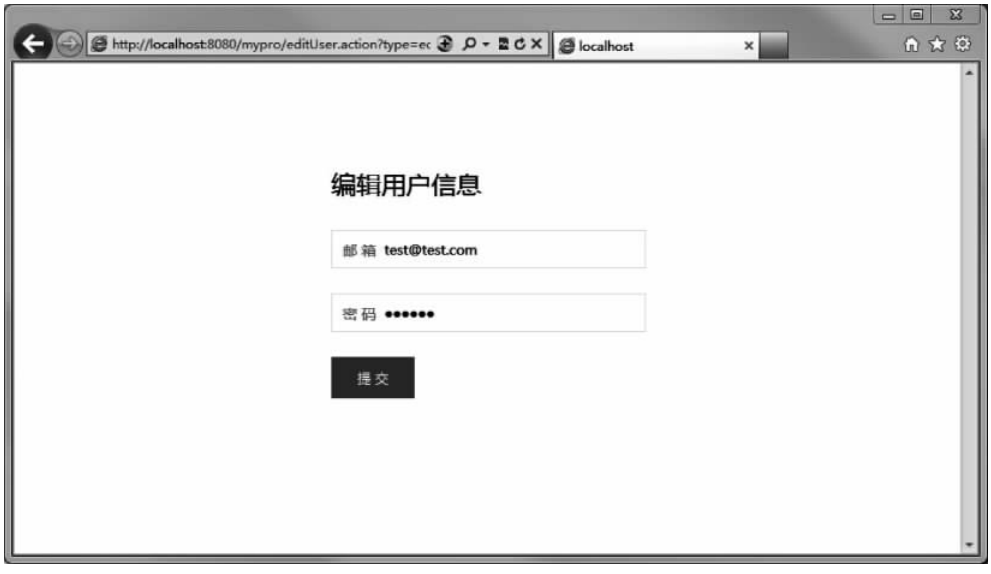

图 3.5-3　使用 Spring 实现用户信息的编辑

修改 test@test.com 为 admin@test.com，并提交修改，可以看到用户类表中的用户信息已经正确地修改了，如图 3.5-4 所示。

图 3.5-4　用户管理页面