第十一章 开发 Spring+Struts+Hibernate 应用

第十一章 开发Spring+Struts+Hibernate应用 1
11.1 创建数据库 3
11.2 快速开发 Struts 应用 3
11.3 添加 Hibernate 功能 10
11.4 添加 Spring 功能 12
11.5 Spring 整合 Hibernate 12
11.6 开发业务层代码 21
11.7 Spring 整合 Struts 23
11.7.1 给Action类加入message属性 23
11.7.2 在 Struts 配置文件中加入Spring配置信息 24
11.7.3 在Spring配置文件中加入Action的bean定义 26
11.7.4 测试，Asm出错和log4j.properties 文件 26
11.7.5 Spring整合Struts的其它方式 29
11.8 完成整合：修改Action代码注入业务层 32
11.9 测试运行 35
11.10 原理探索：模拟 Action 代理类实现 Spring + Struts 35
11.11 开发增删改查的综合用户管理例子 38
11.11.1 创建新项目 38
11.11.2 用Struts设计器制作前台业务流程 38
11.11.3 设计业务层功能 43
11.11.4 开发业务层和DAO层代码 45
11.11.5 开发前台页面流程 53
11.11.6 整合Spring，Struts和Hibernate 68
11.11.7 发布，运行，测试 73
11.11.8 思考与练习 74
11.12 collections.SequencedHashMap 异常的解决方案 74
11.13 小结 75

本章内容将会给大家介绍目前比较流行的开源软件架构：SSH，也就是Spring + Struts 1.x + Hibernate，或者常说的Spring 整合Struts1.x，Hibernate开发。本章将会介绍在上一章的Spring整合Hibernate基础上阐述如何**将Spring和Struts相整合**（这是本章的重点内容），来开发一个简单的用户登录功能（也就是将 第九章 开发Struts 1.x应用所开发的用户登录应用改用Spring整合Hibernate的方式来完成）。为了方便读者，我们先采用最简单的方式进行整合，然后再会讨论不同的Spring + Struts整合策略来作为扩展部分，让读者能有所对比。最后我们会实现一个增删查改的用户管理应用。

我们先来探讨一个问题，为什么要用 Spring 来整合 Struts 呢？难道在 Struts 里面直接新建一个 **ApplicationContext** 然后通过 *getBean()*获取对应的 DAO 层不就也算完成整合了

嘛？没错，这样也算整合，但是显然这样一来，所有的 Struts 类的代码都得改写，而且还有一个很大的缺陷：我们知道 Web 应用的访问是十分频繁的，例如有的站点一天要支持几百万的访问量，每次都来创建一个新的 Spring 容器类是十分消耗资源的，所以这种做法是比较容易想到但是却不实用的。而我们的目的是尽量不改或者少改 Struts 里面的代码来完成整合功能，实际情况是让 Spring 来创建 Action 类，然后注入需要的 DAO 层等等的对象，也就是说要把 Struts 的 Action 类变成 Spring 配置文件中的一个普通 Java 类定义，可以用 property 等标签来设置对应的属性。换句话说，就是要把 Action 代码中的直接调用 Spring 类的代码：

```
// StrutsXXXAction.java
public ActionForward execute(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response) {
  // 手工创建 Spring 容器类
  ...
  ApplicationContext ctx =
                new ClassPathXmlApplicationContext("applicationContext.xml");

  StudentDAO dao = (StudentDAO)ctx.getBean("StudentDAO");

  dao.xxx();// 调用 DAO
  ...
}
```

变成注入方式：

```
private StudentDAO dao;

public StudentDAO getStudentDAO() {
  return dao;
}

public void setStudentDAO (StudentDAO dao) {
  this.dao = dao;
}

public ActionForward execute(ActionMapping mapping, ActionForm form,
                HttpServletRequest request, HttpServletResponse response) {
  //直接使用被 Spring 注入的 DAO 的实例
  dao.xxx();// 调用 DAO，或者用代码 getStudentDAO().xxx();
  ...
}
```

外加一个 Spring 中的 Bean 定义：

```
<bean name="/login" class="com.xxx. StrutsXXXAction">
        <property name="studentDAO">
            <ref bean="dao" />
        </property>
</bean>
```

注意 bean 的 **name** 属性，只有这个属性才能加入带特殊字符的路径名，而对应的属性 **studentDAO** 则通过配置文件来注入上一章已经定义好的 Hibernate DAO 类就可以了，而这个 DAO 因为是 Spring 所定义的，所以可以和上一章的内容一样进行 Spring 整合 Hibernate 的自动事务代理功能开发，而 bean 的 **class** 则指向写好的 Struts Action 类。这样，就可以解决上文所提到的性能问题。

这就是整合的真正目的所在，稍后我们会讨论详细的过程。

11.1 创建数据库

我们的项目所用的表结构和以前的没有任何区别。这里我们使用 MySQL 数据库来完成练习。

建表的 SQL 如下所示：

```
CREATE TABLE Student (
   id int NOT NULL auto_increment,
   username varchar(200) NOT NULL,
   password varchar(20) NOT NULL,
   age int,
   PRIMARY KEY   (id)
) ENGINE=MyISAM DEFAULT CHARSET=GBK
```

。详细内容请参考 5.2 创建数据库表格一节的内容。

11.2 快速开发 Struts 应用

第一阶段我们就是创建一个极其简单的登录功能的程序，前台采用Struts，后台采用Spring＋Hibernate完成。关于本节的详细操作说明可以参考 第九章 开发Struts 1.x应用。这个应用的流程图（用MyEclipse绘制，文件名是*模型.umr*）如下所示：

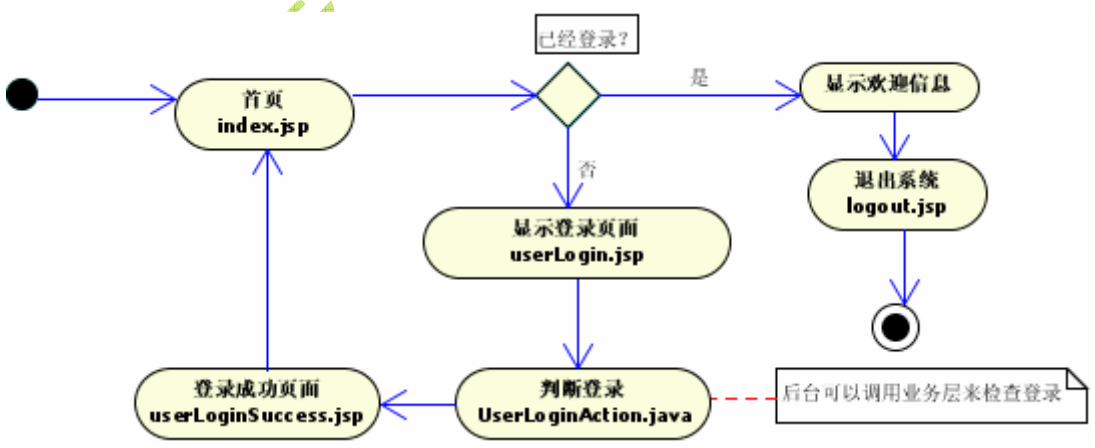

图 11.1 登录应用的页面流程

首先我们新建一个 Web 项目 *ssh1*，也就是 Spring+Struts+Hibernate 第一个项目的意思。选择菜单 **File > New > Web Project**，可以启动创建 Web 项目的向导对话框，在对话框的 **Project Name** 中输入 *ssh1*，然后选中 **J2EE Specification Level** 下面的 **Java EE 5.0** 单选钮，最后点击 **Finish** 按钮就可以建好这个 Web 项目了。

Web 项目创建完毕后，我们需要给它添加 Struts 功能。这个操作可以通过在 **Package Explorer** 视图的项目根节点上右键点击，选择上下文菜单中的 **MyEclipse > Add Struts Capabilities**；或者选择菜单 **MyEclipse > Project Capabilities > Add Struts Capabilities**，接着就启动了添加 Struts 功能的向导。保持所有选项为默认值，选中 **Struts specification** 右侧的 *Struts 1.2* 这个单选钮，然后点击 **Finish** 按钮关闭向导来完成添加 Struts 开发功能的过程。这个过程结束之后，Struts 的类库以及相应的配置文件将会出现在项目的目录结构中。

选择菜单 **File > New > JSP(Advanced Template)**来创建*userLoginSuccess.jsp* 页面，详细操作请参考 9.4.2 创建登录成功页面，最后调整此页码的代码为如下所示：

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean" %>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>
<html:html lang="true">
  <head>
    <html:base />
    <title>登录结果页面</title>
  </head>
  <body>
    你好 <bean:write name="userName" scope="session" />，你已经登录成功!
<br>
    <a href="index.jsp">返回首页</a>
  </body>
</html:html>
```

这个代码和以前的不同点就是加入了一个**返回首页**的链接，这也是一个小小的原则：开发的页码应该有入口，有出口，尽量不要让用户还得点后退才能返回刚开始的页面。

现在双击**WebRoot/WEB-INF/struts-config.xml**就可以打开Struts配置文件编辑器，在设计器网格面板上点击右键，选择菜单**New > Form, Action and JSP**，启动创建ActionForm以及Action和JSP的向导，请参考 9.4.3 使用新建Form，Action和JSP的向导创建关键组件一节的内容完成练习。完成后得到如下的几个文件：登录表单页面*WebRoot/userLogin.jsp*（表示层）：

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>

<html>
    <head>
        <title>JSP for UserLoginForm form</title>
    </head>
   <script type="text/javascript">
// 验证输入不为空的脚本代码
function checkForm(form) {
    if(form.userName.value == "") {
        alert("用户名不能为空!");
        form.userName.focus();
```

```
        return false;
    }

    if(form.password.value == "") {
        alert("密码不能为空!");
        form.password.focus();
        return false;
    }

    return true;
}
</script>
    <body>
        <html:form action="/userLogin" onsubmit="return
checkForm(this);">
            用户名: <html:text property="userName"/><html:errors
property="userName"/><br/>
            密码: <html:password property="password"/><html:errors
property="password"/><br/>

            <html:submit value="登录"/><html:cancel value="取消"/>
        </html:form>
    </body>
</html>
```

大家可以注意到这个文件也加入了 JavaScript 来防止用户输入空的表单值，也就是说 Struts 的 *html:form* 标签定义的表单输入组件也是完全可以来用 JavaScript 来做提交验证的。

Struts 配置文件 *WebRoot/WEB-INF/struts-config.xml*：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 1.2//EN"
"http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
  <data-sources />
  <form-beans >
    <form-bean name="userLoginForm"
type="com.yourcompany.struts.form.UserLoginForm" />

  </form-beans>

  <global-exceptions />
  <global-forwards />
  <action-mappings >
    <action
```

```
      attribute="userLoginForm"
      input="/userLogin.jsp"
      name="userLoginForm"
      path="/userLogin"
      scope="request"
      type="com.yourcompany.struts.action.UserLoginAction">
      <forward name="failed" path="/userLogin.jsp" />
      <forward name="success" path="/userLoginSuccess.jsp" />
    </action>

  </action-mappings>

  <message-resources
parameter="com.yourcompany.struts.ApplicationResources" />
</struts-config>
```

以及加入了硬编码的登录身份验证功能（粗斜体代码，通过简单的字符串比较来判断）的 Action 类 *com.yourcompany.struts.action. UserLoginAction* 的代码（控制器层）：

```
package com.yourcompany.struts.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import com.yourcompany.struts.form.UserLoginForm;

public class UserLoginAction extends Action {
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        UserLoginForm userLoginForm = (UserLoginForm) form;// TODO
Auto-generated method stub
        if(userLoginForm.getUserName().equals("myeclipse") &&
userLoginForm.getPassword().equals("myeclipse"))
        {
          // 用户登录的一般做法是把信息放入session（会话）中
          request.getSession(true).setAttribute("loginedUser", userLoginForm);

          return mapping.findForward("success");
        }

        return mapping.findForward("failed");
    }
}
```

注意代码中的粗斜体部分就是硬编码的登录判断和流程跳转。

对应的 FormBean 类 *com.yourcompany.struts.form.UserLoginForm* 的代码（严格来说算是表示层的部分）：

```
package com.yourcompany.struts.form;

import org.apache.struts.action.ActionForm;

public class UserLoginForm extends ActionForm {

    /** password property */
    private String password;

    /** userName property */
    private String userName;

    /*
     * Generated Methods
     */

    /**
     * Returns the password.
     * @return String
     */
    public String getPassword() {
        return password;
    }

    /**
     * Set the password.
     * @param password The password to set
     */
    public void setPassword(String password) {
        this.password = password;
    }

    /**
     * Returns the userName.
     * @return String
     */
    public String getUserName() {
        return userName;
    }

    /**
```

```
     * Set the userName.
     * @param userName The userName to set
     */
    public void setUserName(String userName) {
        this.userName = userName;
    }
}
```

另外，作为实际项目会遇到的问题，建议按照 9.4.4 调整生成的代码一节加入解决中文字符编码的过滤器。

最后，以前我们所做的应用都是没对首页 *index.jsp* 做任何修改，而是让用户通过地址栏来手工输入登录页面的地址，很显然这样的应用是对用户很不友好的，大家看到一长串的地址要我来记，本能的就会抵触，觉得你的东西做的不好。因此我们将在首页做登录的判断，如果已经登录了，则显示**退出系统**和**重新登录**链接；否则，就显示**登录**链接。*index.jsp* 页面的代码如下所示：

```
<%@ page pageEncoding="GBK"%>
<%
    String path = request.getContextPath();
    String basePath = request.getScheme() + "://"
            + request.getServerName() + ":" + request.getServerPort()
            + path + "/";
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
    <head>
        <base href="<%=basePath%>">
        <title>用户自服务系统</title>
        <meta http-equiv="pragma" content="no-cache">
        <meta http-equiv="cache-control" content="no-cache">
        <meta http-equiv="expires" content="0">
    </head>
    <body>
        <%
            // 检查session判断是否登录
            if (session.getAttribute("loginedUser") != null) {
        %>
        欢迎您，${loginedUser.userName}。您现在可以：<a
href="userLogin.jsp">重新登录</a>
        或者<a href="logout.jsp">退出系统</a>。
        <%
            } else {
        %>
        游客，欢迎您，请<a href="userLogin.jsp">登录</a>。
        <%
            }
```

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

退出系统的页面对应的功能就是使 session 失效，然后重定向到首页，代码清单如下：

logout.jsp

```
<%@ page pageEncoding="GBK"%>
<%
// 退出登录页面
session.invalidate();
// 重定向到首页
response.sendRedirect("index.jsp");
%>
```

最后，不要忘了参考 **9.4.4 调整生成的代码**一节的内容，在 Tomcat 下开发的时候加入那个解决表单提交中文问题的过滤器。实际开发中千万不要漏了这一步，否则用户输入信息的时候用不了汉字岂不是很光火。

到此为止，这个小项目的第一阶段就算开发完毕了，下图显示了这个小项目第一阶段的文件列表（没有包含过滤器部分的类文件）：

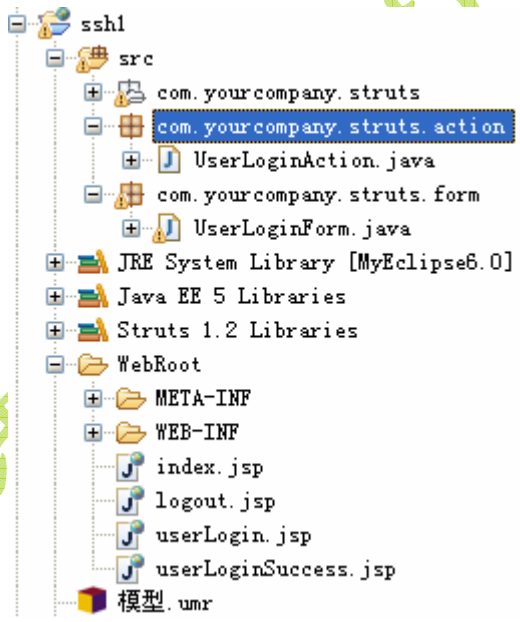

图 11.2 登录项目的目录结构

那么在实际的开发中，到了这一阶段通常会做代码的正确性验证。怎么验证？把项目发布，运行，并在浏览器中测试，就可以了。在**Package Explorer**视图中选中项目节点 **ssh1**，然后选择菜单**Run > Run As > 3 MyEclipse Server Application**，之后MyEclipse可能会显示一个可用的服务器列表，选中其中的服务器之一例如*MyEclipse Tomcat*并点击**OK**按钮后，项目就会自动发布，对应的服务器会启动。好了，现在让我们在浏览器（可以用MyEclipse自带的，或者用IE，Firefox，Opera等都可以）键入地址 http://localhost:8080/ssh1/ ，首页显示会像这样：

游客，欢迎您，请 登录。

接着点击首页的**登录**链接，然后用户名和密码中都输入 *myeclipse*，之后点击**登录**按钮调用

后台的 Struts Action 类进行登录：

用户名：myeclipse

密码：*********

登录 取消

登录成功后显示下面的页面：

你好 myeclipse，你已经登录成功！
返回首页

再点击“**返回首页**”这个链接，可以注意到这时候显示的首页内容有所不同：

欢迎您，myeclipse。您现在可以：重新登录 或者 退出系统。

点击**退出系统**就可以取消登录状态并重返首页。

好了，可以看到很成功，接下来我们会把后台的身份验证改成用 Hibernate 访问数据库来实现。

11.3 添加 Hibernate 功能

现在我们要把Hibernate功能添加到当前项目，但是这一步不会再反向工程生成DAO了，因为我们最终的目的是使用Spring整合的Hibernate DAO 类。现在您可以参考 7.4.3 添加 Hibernate Capabilities 到现有项目一章的内容来进行操作。使用Hibernate可以大大加快

首先在 **Package Explorer** 中选择 *HibernateDemo* 项目，接下来，从 MyEclipse 单栏选择 **MyEclipse > Project Capabilities > Add Hibernate Capabilities ...** 来启动 **Add Hibernate Capabilities** 向导。然后在向导的第一页的 **JAR Library Installation**（JAR 类库安装）处点击选中单选钮 **Copy checked Library Jars to project folder and add to build-path**（选中的类库的 JAR 文件将会被复制到项目目录并添加到构造路径中去）然后再在此单选钮下方的 **Library Folder** 会自动选中*/WebRoot/WEB-INF/lib* 目录，此过程如下图所示。

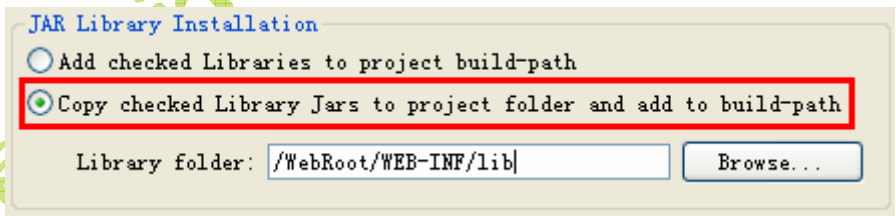

图 11.3 添加 Hibernate 类库时的选项

注意：如果您不选择这个选项，目前来说 MyEclipse 自带的一些类库存在冲突现象，这时候要修正这个问题将会非常困难，因为 MyEclipse 自带的 Library 定义是不能修改的，只能到发布后的 Tomcat 的 webapps 下面对应的应用的下 *WEB-INF/lib* 来修改 jar 文件，而且以后每次重新发布原有文件都被删掉重新复制，都要重新再修改一次才可以，非常不方便；选中这个选择后，因为所有的 jar 文件都会出现在*/WebRoot/WEB-INF/lib*，我们可以非常方便的替换一次里面的 jar 文件就能解决冲突问题。

接着点击 **Next** 按钮，进入第二页，在这一页保持默认的选项来创建 Hibernate 的全局配置文件 *hibernate.cfg.xml*，然后再点击 **Next** 按钮进入第三页，点击 **DB Driver** 右侧的现有数据库连接列表，选择以前创建好的数据库连接例如 *mysql5*，然后再点击 **Next** 按钮进

入最后一页，这一页选择是否生成 Session Factory 类，去掉复选框 **Create SessionFactory class?**的选中状态，最后点击 **Finish** 按钮完成整个添加过程。这最后一页的设置如下所示：

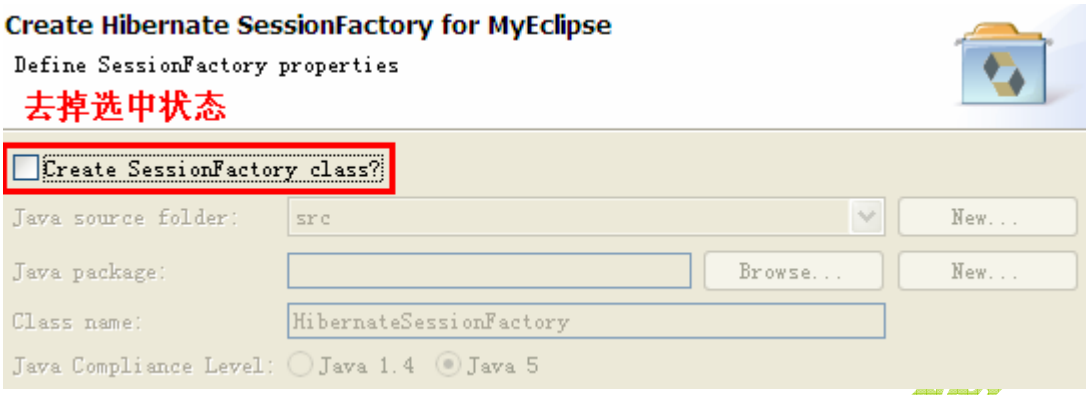

图 11.4 添加 Hibernate 功能向导的最后一页

稍等片刻之后，类库将会添加当前项目，所有的 jar 文件会出现在目录 */WebRoot/WEB-INF/lib* 下，然后修改文件 *src/hibernate.cfg.xml* 的内容如下所示（已改动部分已粗斜体显示）：

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<!-- Generated by MyEclipse Hibernate Tools.                   -->
<hibernate-configuration>

    <session-factory>
        <property name="connection.username">root</property>
        <property
name="connection.url">jdbc:mysql://localhost:3306/test?useUnicode=tru
e&amp;characterEncoding=GBK</property>
        <property
name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="myeclipse.connection.profile">mysql5</property>
        <property
name="connection.driver_class">com.mysql.jdbc.Driver</property>

    </session-factory>

</hibernate-configuration>
```

这么做的原因是 XML 文件中**&**是个特殊字符，必须用转义后的**&**来代替。

11.4 添加 Spring 功能

最后这个项目需要加入的内容就是Spring功能了，详细的操作过程可以参考 10.5.2.1 创建项目，添加必要的开发功能一节内容。

选择菜单 **MyEclipse > Project Capabilities > Add Spring Capabilities ...** 来启动 **Add Spring Capabilities** 向导。在这个向导的第一页有两个地方需要设置：第一个地方是选择类库的时候要选中 *Spring ORM* 和 *Spring Web* 的包，在 **Select the libraries to add to the buildpath**（选择添加到类路径的类库）一栏的下侧，选中 **Spring 2.0 Persistence Core Libraries** 和 **Spring 2.0 Web Libraries**；第二个地方是在 **JAR Library Installation**（JAR 类库安装）处点击选中单选钮 **Copy checked Library contents to project folder (TLDs always copied)**，这个选项将选中的类库的 JAR 文件复制到项目目录，在此单选钮下方的 **Library Folder** 会自动选中*/WebRoot/WEB-INF/lib* 目录，点击 **Next** 按钮进入第二页。那么第二页选择 Spring 配置文件的位置保持原来的不变即可，再点击进入第三页设置 LocalSessionFactory，这一页也保持默认，然后点击 **Finish** 按钮结束整个向导。稍等片刻后，所有的 jar 文件都会复制到*/WebRoot/WEB-INF/lib* 目录，这时候点击并展开 **Package Explorer** 视图中项目的 *Referenced Libraries* 节点就可以看到所有的 Spring 和 Hibernate 相关的 jar 文件都已经添加完毕了。

11.5 Spring 整合 Hibernate

在 **10.5.2 Hibernate 整合 Spring 开发** 一节我们已经详细讨论了 Spring 整合 Hibernate 的步骤和不同的方式。为了简化开发，我们将会使用 **10.5.2.5 用 Spring 2.0 的 @Transactional 标注解决事务提交问题（最佳方案）**一节的标注方式来生成 DAO 层。具体步骤和代码请参考那一节的介绍即可。为了让大家有个对照，我这里把 DAO 类代码，配置文件源码以及测试类的代码贴出来供参考用，当然为了节省版面，我已经去掉了绝大部分无用的注释。

先贴的是两个配置文件的，包括 Spring 的和 Hibernate 的。

Hibernate 配置文件 **hibernate.cfg.xml**：

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd
">

<hibernate-configuration>

    <session-factory>
        <property name="connection.username">root</property>
```

```
        <property name="connection.url">

    jdbc:mysql://localhost:3306/test?useUnicode=true&amp;amp;chara
cterEncoding=GBK
        </property>
        <property name="dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property
name="myeclipse.connection.profile">mysql5</property>
        <property name="connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <mapping resource="dao/Student.hbm.xml" />

    </session-factory>

</hibernate-configuration>
```

Spring 配置文件 **applicationContext.xml**：

```
<?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-2.0.xsd
      http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
      http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <tx:annotation-driven transaction-manager="transactionManager"
proxy-target-class="true"/>
```

```
    <bean id="sessionFactory"

    class="org.springframework.orm.hibernate3.LocalSessionFactoryB
ean">
        <property name="configLocation"
            value="classpath:hibernate.cfg.xml">
        </property>
    </bean>

    <bean id="StudentDAO" class="dao.StudentDAO">
        <property name="sessionFactory">
            <ref bean="sessionFactory" />
        </property>
    </bean>

    <!-- 声明一个 Hibernate 3 的 事务管理器供代理类自动管理事务用 -->
    <bean id="transactionManager"

    class="org.springframework.orm.hibernate3.HibernateTransaction
Manager">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>

</beans>
```

实体类映射配置文件 **dao/Student.hbm.xml**：

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping
DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
```

```
<hibernate-mapping>
    <class name="dao.Student" table="student" catalog="test">
        <id name="id" type="java.lang.Integer">
            <column name="id" />
            <generator class="increment" />
        </id>
        <property name="username" type="java.lang.String">
            <column name="username" length="200" not-null="true" />
        </property>
        <property name="password" type="java.lang.String">
            <column name="password" length="20" not-null="true" />
        </property>
        <property name="age" type="java.lang.Integer">
            <column name="age" />
        </property>
    </class>
</hibernate-mapping>
```

DAO 层实体类 **dao.Student.java**：

```
package dao;

/**
 * Student 实体类.
 */
public class Student implements java.io.Serializable {
    private Integer id;
    private String username;
    private String password;
    private Integer age;

    public Student() {
    }

    public Student(Integer id, String username, String password) {
        this.id = id;
        this.username = username;
        this.password = password;
    }
```

```
    public Student(Integer id, String username, String password, Integer
age) {
        this.id = id;
        this.username = username;
        this.password = password;
        this.age = age;
    }

    public Integer getId() {
        return this.id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getUsername() {
        return this.username;
    }

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

    public String getPassword() {
        return this.password;
    }

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

    public Integer getAge() {
        return this.age;
    }

    public void setAge(Integer age) {
        this.age = age;
    }

}
```

DAO 层核心类 **dao.StudentDAO.java**：

```
package dao;
```

```
import java.util.List;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.LockMode;
import org.springframework.context.ApplicationContext;
import
org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.transaction.annotation.Transactional;

/**
 * Student 的 DAO 层，加入了事务标注。
 */
@Transactional
public class StudentDAO extends HibernateDaoSupport {
    private static final Log log = LogFactory.getLog(StudentDAO.class);
    // property constants
    public static final String USERNAME = "username";
    public static final String PASSWORD = "password";
    public static final String AGE = "age";

    protected void initDao() {}

    public void save(Student transientInstance) {
        log.debug("saving Student instance");
        try {
            getHibernateTemplate().save(transientInstance);
            log.debug("save successful");
        } catch (RuntimeException re) {
            log.error("save failed", re);
            throw re;
        }
    }

    public void delete(Student persistentInstance) {
        log.debug("deleting Student instance");
        try {
            getHibernateTemplate().delete(persistentInstance);
            log.debug("delete successful");
        } catch (RuntimeException re) {
            log.error("delete failed", re);
            throw re;
        }
    }
```

```
    public Student findById(java.lang.Integer id) {
        log.debug("getting Student instance with id: " + id);
        try {
            Student instance = (Student) getHibernateTemplate().get(
                    "dao.Student", id);
            return instance;
        } catch (RuntimeException re) {
            log.error("get failed", re);
            throw re;
        }
    }

    public List findByExample(Student instance) {
        log.debug("finding Student instance by example");
        try {
            List results =
getHibernateTemplate().findByExample(instance);
            log.debug("find by example successful, result size: "
                    + results.size());
            return results;
        } catch (RuntimeException re) {
            log.error("find by example failed", re);
            throw re;
        }
    }

    public List findByProperty(String propertyName, Object value) {
        log.debug("finding Student instance with property: " +
propertyName
                + ", value: " + value);
        try {
            String queryString = "from Student as model where model."
                    + propertyName + "= ?";
            return getHibernateTemplate().find(queryString, value);
        } catch (RuntimeException re) {
            log.error("find by property name failed", re);
            throw re;
        }
    }

    public List findByUsername(Object username) {
        return findByProperty(USERNAME, username);
    }
```

```
    public List findByPassword(Object password) {
        return findByProperty(PASSWORD, password);
    }

    public List findByAge(Object age) {
        return findByProperty(AGE, age);
    }

    public List findAll() {
        log.debug("finding all Student instances");
        try {
            String queryString = "from Student";
            return getHibernateTemplate().find(queryString);
        } catch (RuntimeException re) {
            log.error("find all failed", re);
            throw re;
        }
    }

    public Student merge(Student detachedInstance) {
        log.debug("merging Student instance");
        try {
            Student result = (Student) getHibernateTemplate().merge(
                    detachedInstance);
            log.debug("merge successful");
            return result;
        } catch (RuntimeException re) {
            log.error("merge failed", re);
            throw re;
        }
    }

    public void attachDirty(Student instance) {
        log.debug("attaching dirty Student instance");
        try {
            getHibernateTemplate().saveOrUpdate(instance);
            log.debug("attach successful");
        } catch (RuntimeException re) {
            log.error("attach failed", re);
            throw re;
        }
    }

    public void attachClean(Student instance) {
```

```
        log.debug("attaching clean Student instance");
        try {
            getHibernateTemplate().lock(instance, LockMode.NONE);
            log.debug("attach successful");
        } catch (RuntimeException re) {
            log.error("attach failed", re);
            throw re;
        }
    }

    public static StudentDAO
getFromApplicationContext(ApplicationContext ctx) {
        return (StudentDAO) ctx.getBean("StudentDAO");
    }
}
```

测试类 **test.StudentDAOTest.java**：

```
package test;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import dao.*;

public class StudentDAOTest{

    public static void main(String[] args) {
        ApplicationContext ctx =
            new
ClassPathXmlApplicationContext("applicationContext.xml");

        StudentDAO dao = (StudentDAO)ctx.getBean("StudentDAO");

        Student user  = new Student();
        user.setPassword("密码");
        user.setUsername("spring hibernate 标注事务测试");
        user.setAge(200);

        dao.save(user);
    }

}
```

在本节开发完毕后我们应该进行一些测试，运行最后给出的测试类 *test.StudentDAOTest*，然后到数据库中去检查记录是否已经成功插入数据库。实际开发中我们也应该这样，确保新加入的功能都没有问题然后再继续往下进行进行后续的开发。如果

贪图快速，先把所有模块的代码都写好，然后再运行，一般来说最后都会出现大批的错误，除非你是非常非常熟练的专家。

11.6 开发业务层代码

在实际的应用开发中，我们一般是要把 DAO 层和业务层分开来设计的，这样便于以后进行升级和维护。举个简单的例子，业务层可能需要一些新的判断条件，这些代码不能出现在 DAO 层，因为 DAO 层只相当于实现增删改查这样的功能，尽量不要进行任何的复杂的条件判断，这样 DAO 层可以根据需要选择别的实现，例如用 JPA 和 JDBC 来实现。最主要的是便于团队成员间合作开发，例如将 DAO 层和业务层的开发人员分离开来。再这里我们需要创建一个 *service.StudentManager* 类作为业务层，里面只需要一个委托给 DAO 层的登录验证方法，而且加入必要的参数验证功能。因为要使用 DAO 类的功能，所以需要在这个类中编写一个名称为 *studentDAO* 的属性和一个名为 *checkLogin* 的业务方法。好了，选择菜单 **File > New > Class** 来创建这个类。下面是这个类的清单代码：

```
package service;

import dao.*;

/**
 * 学生（用户）业务层类。
 */
public class StudentManager {
    // 需要注入的 dao 层
    private StudentDAO studentDAO;

    public StudentDAO getStudentDAO() {
        return studentDAO;
    }

    public void setStudentDAO(StudentDAO studentDAO) {
        this.studentDAO = studentDAO;
    }

    /**
     * 根据用户名和密码来检查用户帐号是否有效。
     * @param username 用户名
     * @param password 密码
     * @return boolean - 登录是否成功
     */
    public boolean checkLogin(String username, String password) {
        if(username == null || password == null) {
            return false;
        }
```

```
        return findStudent(username, password) != null ;
    }

    /**
     * 根据用户名和密码来查找用户。
     * @param username 用户名
     * @param password 密码
     * @return 找到的Student对象，如果找不到返回值为null
     */
    public Student findStudent(String username, String password) {
        Student student = new Student();
        student.setUsername(username);
        student.setPassword(password);

        java.util.List results =
getStudentDAO().findByExample(student);

        if(results != null && results.size() > 0) {
            return (Student)results.get(0);
        }

        return null;
    }
}
```

那么现在我们简要描述一下这个登录的业务类。首先通过用户名和密码来查找用户信息是否存在，具体方法是 ***public Student findStudent(String username, String password)***，它在实现的时候是通过 StudentDAO 的 *findByExample()*方法来查找用户信息。然后在登录的具体实现 ***public boolean checkLogin(String username, String password)*** 中则委托到刚刚介绍的方法，如果找不到对应的用户对象，就认为登录失败了。

注意：必须在 checkLogin()方法中做参数判断，要求两个参数都不能为空，否则你会发现当两个参数都为 null 的时候，仍然可以登录成功。

因为这个类需要访问底层的数据库，在这里可以通过 Spring 来设置对应的 DAO 类的实例，因此我们还要在 Spring 的配置文件 **applicationContext.xml** 中加入下面的 bean 配置：

```
    <!-- 用户业务类 -->
    <bean id="studentManager" class="service.StudentManager">
        <property name="studentDAO">
            <ref local="StudentDAO" />
        </property>
    </bean>
```

。

最后，当然是要对刚刚写的这些内容进行测试了。现在我们新建一个测试类 **test.StudentManagerTest**，其代码如下所示：

```
package test;
import org.springframework.context.ApplicationContext;
```

```
import
org.springframework.context.support.ClassPathXmlApplicationContext;
import service.StudentManager;

public class StudentManagerTest {

    public static void main(String[] args) {
        ApplicationContext ctx =
            new
ClassPathXmlApplicationContext("applicationContext.xml");

        StudentManager studentManager =
(StudentManager)ctx.getBean("studentManager");

        System.out.println("登录检查＝" +
studentManager.checkLogin("spring hibernate 标注事务测试", "密码"));
    }

}
```

。运行这个类，得到如下的输出：

```
log4j:WARN No appenders could be found for logger
(org.springframework.context.support.ClassPathXmlApplicationContext).
log4j:WARN Please initialize the log4j system properly.
登录检查＝true
```

。Well done! 输出结果很正确。现在将用户名或者密码参数稍作修改后，例如改为空或者 null，再运行应该会输出 false。测试嘛，一定要全面才对。

11.7 Spring 整合 Struts

上面我们已经创建了必要的 DAO 和业务层代码，还有 Struts 表示层的内容，现在是时候进行整合了。那么这一节我们主要介绍 Spring 和 Struts 的整合开发。首先我们介绍最简单的整合方案，然后再介绍其它的两种稍微麻烦一点的办法。Spring 整合 Struts 的本质，就是将 Struts 的 Action 类中的属性（通过 JavaBean 的 *getXXX()*和 *setXXX()*来规定）运行时通过 Spring 来注入（也就是赋值的意思），要实现这一过程，本质上是将每个需要 Action 实例的地方都通过 Spring 来产生，也就是：*applicationContext.getBean("/login")*代替了 *new LoginAction()*。

11.7.1 给 Action 类加入 message 属性

本着由浅入深的原则，我们整合的**第一步**不是给 Action 类注入一个复杂的 DAO 类的实例，而是先加入一个名为 message 的属性，并且在登录的时候打印这个属性。修改后的 **LoginAction** 代码如下所示（注释已删除）：

```
package com.yourcompany.struts.action;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import com.yourcompany.struts.form.UserLoginForm;

public class UserLoginAction extends Action {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        System.out.println(getMessage());
        UserLoginForm userLoginForm = (UserLoginForm) form;// TODO
                                                           // Auto-generated
                                                           // method stub
        if (userLoginForm.getUserName().equals("myeclipse")
                && userLoginForm.getPassword().equals("myeclipse")) {
            // 用户登录的一般做法是把信息放入session（会话）中
            request.getSession(true).setAttribute("loginedUser", userLoginForm);
            return mapping.findForward("success");
        }

        return mapping.findForward("failed");
    }
}
```


粗斜体部分就是我们新加入的属性代码，这个属性类型为 String，名称为 *message*，通过 JavaBean 规范的 getter 和 setter 方法来完成这个属性的定义。并且在 execute()方法执行的开始就向控制台输出值，便于我们观察是否整合成功，成功的话应该能打印出注入的属性值来的。

11.7.2 在 Struts 配置文件中加入 Spring 配置信息

要让 Struts 获取 Action 的时候通过 Spring 来完成，不得不在 *struts-config.xml* 中做一

点手脚，这也是整合的**第二步**吧（第一步就是像上一节所述加入属性）。这里我们先采用最简单的办法，不修改原有的配置信息，只加入新的内容。现在看看修改后的 Struts 配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 1.2//EN"
"http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
    <data-sources />
    <form-beans>
        <form-bean name="userLoginForm"
            type="com.yourcompany.struts.form.UserLoginForm" />

    </form-beans>

    <global-exceptions />
    <global-forwards />
    <action-mappings>
        <action attribute="userLoginForm" input="/userLogin.jsp"
            name="userLoginForm" path="/userLogin" scope="request"
            type="com.yourcompany.struts.action.UserLoginAction">
            <forward name="failed" path="/userLogin.jsp" />
            <forward name="success" path="/userLoginSuccess.jsp" />
        </action>

    </action-mappings>

    <controller
    processorClass="org.springframework.web.struts.DelegatingRequestProcessor"
/>

    <message-resources
        parameter="com.yourcompany.struts.ApplicationResources" />

    <plug-in      className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
            value="/WEB-INF/classes/applicationContext.xml" />
    </plug-in>

</struts-config>
```

上面的粗斜体部分就是两处需要修改的内容。一个是加入了 Spring 的代理请求处理器（相当于代售火车票的）；另一个是加载 Spring 的插件，**contextConfigLocation** 的取值指定了哪些配置文件需要加载，这个插件可以让 Struts 启动时创建 Spring 的核心 Bean 容器。

注意：如何加载多个 Spring 配置文件？可以在 **contextConfigLocation** 的 value 中写

入多个以逗号（英文半角字符）分开的配置文件位置即可，如下所示：

```
<plug-in      className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
            value="/WEB-INF/classes/service.xml, /WEB-INF/classes/dao.xml " />
    </plug-in>
```

这些配置文件的路径可以放在 web 应用的任意位置，包括 WEB-INF 目录里面和外面，都可以，有的人喜欢把它直接放在*/WEB-INF/user.xml* 这个地方，和其它的配置文件放在一个目录是为了便于寻找和修改这些配置文件，当然对此没有任何特殊的规定，全凭个人爱好来决定。

11.7.3 在 Spring 配置文件中加入 Action 的 bean 定义

Spring 整合 Struts 的最后一步操作就是需要在 Spring 的配置文件将对应的 Struts Action 类配置成一个 bean，如果不这样做，Spring 从哪里来给这些 Action 类注入值呢？配置的要点，就是一定要将这个 bean 的 **name** 和 Struts 配置文件中对应的 action 的 **path** 相对应，例如上文介绍的 LoginAction，要在 Spring 配置文件 *applicationContext.xml* 中加入如下的 bean 定义：

```
    <!-- Struts Action 类 -->
    <bean name="/userLogin"
class="com.yourcompany.struts.action.UserLoginAction">
        <property name="message">
        <value>Spring Struts</value>
        </property>
    </bean>
```

注意代码中的粗斜体部分，这个 name 的取值一定要和 Struts 配置文件 action 中的 path 的值相对应，否则整合就会失败：

```
<action path="/userLogin" ......
```

。既然现在这个 Action 类处于 Spring 的配置和管理之下，自然就可以注入任意的属性，包括 message，以及以后可以加入的 Service 层对象了，也就是说你现在可以完全把它看作一个普通的 JavaBean 来处理就行了，想加入 AOP 什么的都可以。在这里为了测试方便，我们把 *message* 属性注入一个*"Spring Struts"*的值。

11.7.4 测试，Asm 出错和 log4j.properties 文件

现在必要的代码都已经写好，我们应该来验证一下是不是能够正确的运行这个例子。在 **Package Explorer** 视图中选中项目节点 **ssh1**，然后选择菜单 **Run > Run As > 3 MyEclipse Server Application**，之后 MyEclipse 可能会显示一个可用的服务器列表，选中其中的服务器之一例如 *MyEclipse Tomcat* 并点击 **OK** 按钮后，项目就会自动发布，对应的服务器会启动。然而很不幸的可以看到后台服务器启动的时候有报错信息：

```
严重: action: null
org.springframework.beans.factory.BeanCreationException: Error
creating bean with name 'StudentDAO' defined in ServletContext resource
```

```
[/WEB-INF/classes/applicationContext.xml]: Initialization of bean
failed; nested exception is java.lang.NoClassDefFoundError
Caused by: java.lang.NoClassDefFoundError
    at
org.springframework.aop.framework.Cglib2AopProxy.createEnhancer(Cglib
2AopProxy.java:223)
    at
org.springframework.aop.framework.Cglib2AopProxy.getProxy(Cglib2AopPr
oxy.java:150)
    at
org.springframework.aop.framework.ProxyFactory.getProxy(ProxyFactory.
java:110)
    ....
    at org.apache.catalina.startup.Bootstrap.main(Bootstrap.java:413)
2008-1-23 17:59:03 org.apache.catalina.core.ApplicationContext log
信息: Marking servlet action as unavailable
2008-1-23 17:59:03 org.apache.catalina.core.StandardContext
loadOnStartup
严重: Servlet /ssh1 threw load() exception
javax.servlet.UnavailableException
    at
org.apache.struts.action.ActionServlet.initModulePlugIns(ActionServle
t.java:880)
    ....
```

这时候再去测试后台的 Struts 登录，会出现 404 页面没有找到的错误。

这个错误的确非常难处理，因为真实的出错信息我们看不到！为什么这样？这要怪 Spring 用了 LOG4J 做出错信息的输出，虽然后台有很多的真实出错信息，然而在这里您却是看不到的。所以，请参考上一章开发 *HelloSpring* 这个项目时的内容，在 *src* 目录下创建配置文件，选择菜单 **File > New > File**，文件名输入 *log4j.properties*，文件内容如下所示：

```
log4j.rootLogger=WARN, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d %p [%c] - %m%n
```

，加入了这个文件后，就可以看到比较完整的后台日志了。当然更方便的做法是将那个配置文件从文件浏览器或者从 *HelloSpring* 项目中复制到当前项目的 *src* 目录下就可以了。然后重新发布项目后再次启动 Tomcat 服务器，这次我们就可以看到完整的出错信息了：

```
2008-01-24 16:24:07,140 WARN [net.sf.ehcache.config.Configurator] - No
configuration found. Configuring ehcache from ehcache-failsafe.xml found
in the classpath:
jar:file:/E:/workspace/.metadata/.plugins/com.genuitec.eclipse.easie.
tomcat.myeclipse/tomcat/webapps/ssh1/WEB-INF/lib/ehcache-1.1.jar!/ehc
ache-failsafe.xml
2008-01-24 16:24:07,531 ERROR
[org.hibernate.proxy.BasicLazyInitializer] - CGLIB Enhancement failed:
```

```
dao.Student
java.lang.NoSuchMethodError:
org.objectweb.asm.ClassVisitor.visit(IILjava/lang/String;Ljava/lang/S
tring;[Ljava/lang/String;Ljava/lang/String;)V
    at
net.sf.cglib.core.ClassEmitter.begin_class(ClassEmitter.java:77)
....
Caused by: java.lang.NoSuchMethodError:
org.objectweb.asm.ClassVisitor.visit(IILjava/lang/String;Ljava/lang/String;[Ljava/lang/S
tring;Ljava/lang/String;)V
    at
net.sf.cglib.core.ClassEmitter.begin_class(ClassEmitter.java:77)
```

。完整的出错信息很长很长，然而这里我们用粗斜体显示了造成错误的真正原因：找不到对应方法错误。没错，这就是著名的使用 MyEclipse 开发 SSH 框架的 Asm 出错。造成这个错误的真正原因是什么呢？这是因为 Spring 和 Hibernate 都用到了 Asm 这个框架来进行一些必要的操作，例如字节码增强，然而当发布到 Web 项目后，一些包的加载次序发生了冲突，具体一点就是：

asm-2.2.3.jar 和 **asm.jar** 这两个文件中正确的包是 **asm.jar**，而 **asm-2.2.3.jar** 中则包含了老版本的类库，这些类库在 Spring 中使用时就会出错。怎么解决这个问题呢？删除文件 **WebRoot/WEB-INF/lib/asm-2.2.3.jar**，接着重新发布项目，再启动服务器后就看不到出错信息了。具体的操作过程：在 **Package Explorer** 视图中选中目录*/WebRoot/WEB-INF/lib*，然后点击工具栏上的按钮在 Windows 的文件浏览器中打开这个目录，删除文件 **asm-2.2.3.jar**，然后再回到 MyEclipse 窗口，按下键盘上的快捷键 **F5** 或者在目录上点右键选择菜单 **Refresh**，就完成了这个删除的过程了；也可以在 **Package Explorer** 视图中的 **Referenced Libraries** 中先选中文件 **asm-2.2.3.jar** 并点击右键选择菜单 **Build Path > Remove from Build Path** 将其从构造路径中删除，然后就可以在*/WebRoot/WEB-INF/lib* 目录下看到该文件，单击选中它，然后从右键菜单中选择 **Delete** 即可（相对来说用这种方式要快一些）。

注意：在本章开头，**图 11.3 添加 Hibernate 类库时的选项**中介绍添加 Spring 和 Hibernate 开发功能时务必选择复制 JAR 文件到项目的*/WebRoot/WEB-INF/lib* 目录下，就是为了这里便于删除我们不需要的 jar 文件，如果当时您不幸选择了添加 Library，现在就只能束手无策了。如果读者是自己下载 Spring 和 Hibernate 类库文件进行开发的话，就不用担心会出现这个问题了。这个问题是 MyEclipse 的自带的包不兼容导致的。

之后让我们键入地址 http://localhost:8080/ssh1/ 进行测试。点击首页的**登录**链接，然后用户名和密码中都输入*myeclipse*，之后点击**登录**按钮，不论是否成功登录，我们都可以看到在Console视图的Tomcat服务器输出日志中显示了：

```
信息: Server startup in 17473 ms
Spring Struts
```

只要您看到了粗斜体所示的 Spring Struts 这个字符串，那么恭喜！我们的 Spring 整合 Struts 开发成功了！后面要改的就是将业务层代码 StudentManager 注入到 Action 类中后就可以完成整个整合的步骤了。

11.7.5 Spring 整合 Struts 的其它方式

出于维护项目和传递更多信息的需要，本节再介绍 SS 整合的其它方式，作为参考之用，实际开发中只需要掌握上面的那一种方式就足够了。关于详细的说明文档，可以参考 Spring 开发参考手册的 **15.4. Struts** 一节的内容。

方式二：使用 ContextLoaderPlugIn，替换 Struts Action 的类型并在 Spring 中配置对应的类型。

这种方式下的Action类的源码和上面 11.7.1 给Action类加入message属性一节所介绍的是一样的，Spring的配置文件也和 11.7.3 在Spring配置文件中加入Action的bean定义一节所介绍的内容一样。唯一不同的是Struts的配置文件，注意看下面的配置文件 *struts-config.xml*源码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 1.2//EN"
"http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
    <data-sources />
    <form-beans>
        <form-bean name="userLoginForm"
            type="com.yourcompany.struts.form.UserLoginForm" />

    </form-beans>

    <global-exceptions />
    <global-forwards />
    <action-mappings>
        <action attribute="userLoginForm" input="/userLogin.jsp"
            name="userLoginForm" path="/userLogin" scope="request"
            type="org.springframework.web.struts.DelegatingActionProxy">
            <forward name="failed" path="/userLogin.jsp" />
            <forward name="success" path="/userLoginSuccess.jsp" />
        </action>

    </action-mappings>

    <controller
    processorClass="org.springframework.web.struts.DelegatingRequestProcessor"
/>

    <message-resources
        parameter="com.yourcompany.struts.ApplicationResources" />
```

```
    <plug-in

    className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
            value="/WEB-INF/classes/applicationContext.xml" />
    </plug-in>

</struts-config>
```

粗斜体部分显示了代码的不同之处。第一个不同点，就是 Action 类的 **type** 从原来的 *com.yourcompany.struts.action.UserLoginAction* 变成了现在的由 Spring 所提供的代理类 *org.springframework.web.struts.DelegatingActionProxy*，这个类也是 Struts Action 类的一个子类，只不过它的幕后工作就是从 Spring 配置文件中查找定义的 **name** 为*/userLogin* 的 bean 定义来作为真正的 Action 对象，然后委托给它来处理剩下的业务逻辑。第二个不同点，就是删除了下面这句配置语句：*<controller processorClass="org.springframework.web.struts.DelegatingRequestProcessor" />*。那么这种方式总体上来说代码改动量也不太大，实际开发中可以采用。

方式三：使用 Spring 提供的 ActionSupport 类。这种方式，因为 Action 类的代码要做大量修改，已经违背了 Spring 提倡的无侵入这个理念，而且的确出现代码改动量太多的这个问题，因此在实际开发中不提倡使用这种方式。

首先我们先来看采用这种方式的时候相关配置文件的写法。对于 Struts 的配置文件，strus-config.xml 的内容如下所示：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 1.2//EN"
"http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
    <data-sources />
    <form-beans>
        <form-bean name="userLoginForm"
            type="com.yourcompany.struts.form.UserLoginForm" />

    </form-beans>

    <global-exceptions />
    <global-forwards />
    <action-mappings>
        <action attribute="userLoginForm" input="/userLogin.jsp"
            name="userLoginForm" path="/userLogin" scope="request"
            type="com.yourcompany.struts.action.UserLoginActionSupport
">
            <forward name="failed" path="/userLogin.jsp" />
            <forward name="success" path="/userLoginSuccess.jsp" />
```

```
        </action>

    </action-mappings>

    <message-resources
        parameter="com.yourcompany.struts.ApplicationResources" />

    <plug-in       className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
            value="/WEB-INF/classes/applicationContext.xml" />
    </plug-in>

</struts-config>
```

相比较起没加入 Spring 之前的 Struts 的配置文件，其不同之处就如代码中粗斜体部分所示，加入了一个插件配置。那么在这种方式下，不需要在 Spring 的配置文件中加入 Action 类的 bean 定义，相对比的，您需要首先继承自 ActionSupport 类，然后通过在 Action 类中写代码来自己查找 Bean 类：

```
package com.yourcompany.struts.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.springframework.context.ApplicationContext;
import org.springframework.web.struts.ActionSupport;
import service.StudentManager;

/** 基于 Spring ActionSupport 的登录类。 */
public class UserLoginActionSupport extends ActionSupport {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        ApplicationContext ctx = this.getWebApplicationContext();

        StudentManager studentManager = (StudentManager) ctx
                .getBean("studentManager");

        System.out.println("登录检查="
                + studentManager.checkLogin("spring hibernate 标注事务测试", "密码
"));
        // 更多业务代码
        return mapping.findForward("success");
    }
```

```
}
```

代码中的粗斜体部分就是新增的内容，主要就是获得 Spring 的核心容器类：ApplicationContext（可以认为和 BeanFactory 这个类工厂是一回事），然后从里面获取定义好的 bean 类，之后调用上面的业务逻辑。其实这种方式和自己亲手写代码创建 BeanFactoy 然后加载对象是差不多的，对现有代码的改动也比较大，既有继承，还有额外代码，还得导入 Spring 的包，Spring 文档上说推荐这种做法，我怎么看也看不出这样到底有多少方便的地方。

提示：如果您需要其它类型的 Action，例如 *DispatchAction* 等等，那么需要继承自对应的 Spring 版本的类－－ **ActionSupport**，**DispatchActionSupport**，**LookupDispatchActionSupport**，**MappingDispatchActionSupport**，只不过是类名后加了一个 *Support*，这些类位于包 **org.springframework.web.struts** 下面。

最后，我们再介绍 Struts1 加载 Spring 的另一种方式，修改 **web.xml** 配置加载 spring 上下文环境，其配置方式如下：

```
    <context-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/applicationContext.xml</param-value>
    </context-param>
```

通过 listener 加载：

```
   <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener
-class>
    </listener>
```

或者利用 severlet 类在启动时加载：

```
<servlet>
        <servlet-name>context</servlet-name>
        <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-c
lass>
        <load-on-startup>1</load-on-startup>
      </servlet>
```

注意：使用这种方式的时候，就不要在 Strurts 的配置文件中写那个 plugin 配置信息了。

11.8 完成整合：修改 Action 代码注入业务层

有了上一节 Struts Action 中通过 Spring 注入 String 类型的 message 属性的成功经验之后，现在我们就可以着手进行最后一步的整合工作：将业务层代码注入进来。简单说，分两步：第一步是在 Action 中编写一个名为 *studentManager* 的类型为 StudentManager 的属性；第二步就是将 Action 类中的硬编码的登录判断修改成使用注入的 *studentManager* 对象来完成；第三步是将 Spring 配置文件中对应的 Action bean 注入 *studentManager*。下面是 **UserLoginAction** 类完整的代码：

```
package com.yourcompany.struts.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

import service.StudentManager;

import com.yourcompany.struts.form.UserLoginForm;

import dao.Student;

/**
 * 登录验证功能。
 * 作者：BeanSoft
 */
public class UserLoginAction extends Action {
    private String message;

    private StudentManager studentManager;

    public StudentManager getStudentManager() {
        return studentManager;
    }

    public void setStudentManager(StudentManager studentManager) {
        this.studentManager = studentManager;
    }

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        System.out.println(getMessage());
        UserLoginForm userLoginForm = (UserLoginForm) form;
```

```
        // 使用业务层代码检查登录并保存找到的用户实体信息到session中
        if (getStudentManager().checkLogin(userLoginForm.getUserName()
                , userLoginForm.getPassword()) == true) {
            Student student =
getStudentManager().findStudent(userLoginForm.getUserName()
                , userLoginForm.getPassword());
            // 用户登录的一般做法是把信息放入session（会话）中
            request.getSession(true).setAttribute("loginedUser", student);
            return mapping.findForward("success");
        }

        return mapping.findForward("failed");
    }
}
```

修改部分已粗斜体来显示。相对于以前来说发生了两个变化，除了上面提到的加入 *studentManager* 属性和用它来做登录验证外，最主要的是放入 session 中的信息发生了变化，以前我们放入 session 的是 *userLoginForm*，这样来判断用户是否登录，然而，一般来说我们会把用户自己的信息实体放入 session 中去，或者为了节约内存，仅仅放入用户的实体的 ID，这样做是因为一般来说，用户登录之后，都会提供用户维护和修改自己信息的功能，例如修改个人资料，修改密码等等，这样获取登录用户自身的信息就会非常的方便，而 UserLoginForm 则主要是为了给 Struts 的类提供表单提交信息用的，放入 session 显然不是那么合适，甚至还会出现某些属性为 null 的情况。这样一来，表示层的代码也必须要有所改动，我们在两个 JSP 文件 **index.jsp** 和 **userLoginSuccess.jsp** 中所编写的输出登录后用户信息的 EL 表达式 *${loginedUser.userName}*修改为 *${loginedUser.username}*。

接下来我们需要将上一节所介绍的 Spring 配置文件 **applicationContext.xml** 中已经加入的 Action 类定义加入对 studentManager 属性的设置信息，代码判断为：

```
    <!-- Struts Action 类 -->
    <bean name="/userLogin"
class="com.yourcompany.struts.action.UserLoginAction">
        <property name="message">
        <value>Spring Struts</value>
        </property>

        <!-- 注入业务层的bean定义 -->
        <property name="studentManager">
            <ref local="studentManager" />
        </property>
    </bean>
```

粗斜体的部分就是加入的内容。

至此，所有的整合代码都已经完成了。

11.9 测试运行

所有的项目开发结束之后，都要进行正确性验证，首先在自己的电脑上进行运行和测试。在**Package Explorer**视图中选中项目节点 **ssh1**，然后选择菜单**Run > Run As > 3 MyEclipse Server Application**，之后MyEclipse可能会显示一个可用的服务器列表，选中其中的服务器之一例如*MyEclipse Tomcat*并点击**OK**按钮后，项目就会自动发布，对应的服务器会启动。好了，现在让我们在浏览器（可以用MyEclipse自带的，或者用IE，Firefox，Opera等都可以）键入地址 http://localhost:8080/ssh1/ ，首页显示会像这样：

游客，欢迎您，请 登录。

接着点击首页的**登录**链接，然后用户名和密码中输入数据库中 Student 表所实际存在的记录例如我这里输入的是**张三**和 **1234**，之后点击**登录**按钮调用后台的 Struts Action 类进行登录，Struts Action 类再通过 Spring 去调用后台的业务类，业务类最终调用 Hibernate 访问数据库。当登录成功后显示下面的页面：

你好 张三，你已经登录成功！
返回首页

再点击“**返回首页**”这个链接，可以注意到这时候显示的首页内容有所不同：

欢迎您，张三。您现在可以：重新登录 或者 退出系统。

点击**退出系统**就可以取消登录状态并重返首页。

这样，经过测试，所有功能都很完善了。那么最后我们要做的是什么呢？就是做一个增删改查的综合例子。

11.10 原理探索：模拟 Action 代理类实现 Spring + Struts

至此，可能有部分读者对 Spring 究竟是如何取代 Struts 中的 Action 类的生成机制心存疑问。您可能问，到底 Spring 是如何做到了运行的时候根据请求的路径自动请求到对应的 Action 类的 bean 定义的呢？在这里呢我们就来通过实地模拟一下基于代理模式的使用 Spring 的 *DelegatingActionProxy* 所实现的 Spring + Struts 整合策略，来帮助读者理解这个过程。当然读者也可以跳过本节内容，这一节内容仅仅作为研究和提高，对开发没有直接的影响。

我们已经知道 Spring 整合 Struts 的关键点在于三个，第一个是加载 Spring 的 BeanFactory，第二个是设法代替 Struts 配置文件中的 Action 类的类型（或者类似的策略例如替换其核心的 Processor），第三个是 Spring 的 Bean 配置文件中必须配置出来 name 和 Struts 的 Action 路径（path）相同的 Action bean 的声明。那好，我们来写一个 SpringProxyAction 来帮助您理解为什么让所有的 Struts 中的 Action 的 type（类型）都替换成同一个类但是对应的功能依然能够正确工作而且可以被注入对象。下面就是我们自己所写的 Spring 代理 Action 类：

```
package com.yourcompany.struts.action;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.apache.struts.action.Action;
```

```
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * 模拟的Spring代理Action。
 */
public class SpringProxyAction extends Action {
    // 相当于 Struts 配置文件中启动 Spring 的plug-in代码，一般是单例
    ApplicationContext ctx = new
    ClassPathXmlApplicationContext(
            "applicationContext.xml");

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response)
throws Exception {
        // 首先获取请求的路径
        String path = request.getRequestURI();// /ssh1/userLogin.do
        String context = request.getContextPath();// /ssh1
        // 分析出对应的 bean 的 name
        String beanName = path.substring(context.length(),
path.lastIndexOf(".do"));// /userLogin
        System.out.println("SpringProxyAction:beanName=" + beanName);

        // 从 Spring 中获取对应的 bean
        Action action = (Action)ctx.getBean(beanName);

        // 将业务逻辑委托给所获取的 Spring bean 实例
        if(action != null) {
            try {
                return action.execute(mapping, form,
                        request, response);
            } catch (Exception e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }

        throw new Exception("No Action bean found for path:" + beanName);
    }
}
```

至于这个类的执行流程，也不复杂，我已经在主要的点作出了注释了，其实 Spring 底层的实现也是和这个类似的。现在我们要做的就是将 Struts 配置文件中原来的 Action 的 Type 替换掉，这样也能实现 Spring 整合 Struts 了。下面是完整的配置文件 **struts-config.xml**：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 1.2//EN"
"http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
    <data-sources />
    <form-beans>
        <form-bean name="userLoginForm"
            type="com.yourcompany.struts.form.UserLoginForm" />

    </form-beans>

    <global-exceptions />
    <global-forwards />
    <action-mappings>
        <action attribute="userLoginForm" input="/userLogin.jsp"
            name="userLoginForm" path="/userLogin" scope="request"
            type="com.yourcompany.struts.action.SpringProxyAction">
            <forward name="failed" path="/userLogin.jsp" />
            <forward name="success" path="/userLoginSuccess.jsp" />
        </action>

    </action-mappings>

    <message-resources
        parameter="com.yourcompany.struts.ApplicationResources" />

</struts-config>
```

代码中粗斜体的部分就是新的代理 Action，Spring 的配置文件不需要做额外的修改。如果有其它的普通 Action，我们只需要将其类型也替换为 *com.yourcompany.struts.action.SpringProxyAction* 即可，不管有多少个都是这样做，当然前提是他们必须是普通的 Action 类，如果是 Struts 的 DispatchAction，我们这个简单的类这样替换恐怕是不行的。然后我们现在可以重新发布一下项目，运行，并在浏览器中输入登录的用户名和密码进行测试，这时候可以在控制台产生如下输出：

```
SpringProxyAction:beanName=/userLogin
Spring Struts
```

非常好，这就说明我们的代理 Action 类工作了，而且很正确的注入了 message 和业务层的属性对应的 bean。

所以，Spring 整合 Struts 的核心，就在于接替 Struts 创建 Action 类的工作，而是让每个使用 Action 的地方都跑去 Spring 那里去取一个 Action 的 Bean 定义出来。这就像售票处

一样，现在您只要告诉售票处想买什么票就可以了，售票处自己知道怎么去取票出来，而不用麻烦您再亲自去车站的票务系统印好的票中查找一张出来了，这就是代理的作用，也或者说是外包服务吧，更贴切一些。Spring 整合大多数 Web 层框架的核心，就是用代理来代替原来的产生类的方式，替之以 Spring 的 Bean 工厂，也只有这样，Spring 才能随心所欲的注入或者 AOP 一下。

11.11 开发增删改查的综合用户管理例子

在**9.5 编写 Struts 整合 Hibernate 的分页应用**一节我们已经开发了一个分页显示用户信息的实例应用 *StrutsPageDemo*，那么在这里我们要把这里例子改用 Spring 来开发，并完成以前尚未加入的增加，删除，修改和查询功能。当然，分页也是必不可少的。

11.11.1 创建新项目

首先我们要基于 *StrutsPageDemo* 项目来开发，先复制一份新的。先打开那个项目，然后在 **Package Explorer** 视图中选中项目根结点，之后按下快捷键 **Ctrl + C** 复制一次，再按下 **Ctrl + V** 粘贴一次，再提示的复制项目对话框的 **Project name** 处输入 *ssh2*，然后点击 **OK** 按钮，这样一个新的基于原项目代码的新项目就建好了，如下图所示。

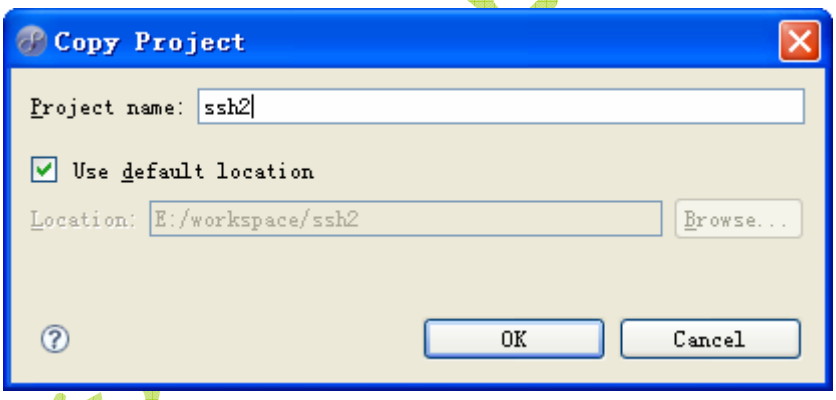

图 11.5 复制项目对话框

接下来就是对此项目做修改了，对于前台来说，需要加入增加，修改删除和查询用户所对应的页面，后台功能通过调用 Struts Action 来完成，Struts 的 Action 再委托给业务层其实就是 StudentManager 类来完成。根据个人爱好的不同，你既可以先完成前台的页面，也可以先把后台的业务类写完整，或者是两个人分工的话一人负责一部分。

11.11.2 用 Struts 设计器制作前台业务流程

在实际项目开发中，第一步要做的可能是设计应用的整个流程，而不是立即着手制作 JSP 或者编写 Java 代码。这就像我们做任何事，除了本能的那些不需要经过思考的反应（例如眼前出现闪光，会本能的闭眼）之外，大概都要先做个计划，然后再付之于实施，虽然并非所有的计划都要写成一份文档，但是大部分情况下，我们也会在脑子里有一个大致的想法吧。那么做实际的项目开发，如果是大中型项目，思考项目到底该使用哪些技术和架构，用

什么数据库和操作系统，最后应该实现成何种样子，有哪些模块，系统的工作流程如何等等，这些内容，相当于设计阶段。对于我们要做的这个项目来说，具体的技术和系统结构已经选好，就是 SSH 架构。那么现在，在加入 Spring 整合功能之前，我们有必要把系统的表示层的工作流程搞清楚。做这个流程设计有很多办法，可以用画图工具例如 Microsoft Visio，或者 UML 建模工具开发一个活动图等等都可以。因为我们使用的是 MyEclipse，所以我们这里可以用它自带的 Struts 设计器快速的画出所有的应用流程来。

现在双击 **WebRoot/WEB-INF/struts-config.xml** 就可以打开 Struts 配置文件编辑器，在设计器网格面板上点击右键，选择菜单来创建 Action，Forward，JSP 等等。为了更好的反映系统的工作流程，我们还可以使用设计器工具栏上的  按钮来给一些页面加入连接关系，但是如果弹出对话框询问是否设置为 Action 的 input 元素时，我们选择 **No** 按钮。大致来说，我们的应用需要增删改查四个模块。下面是最后完成的反映在 struts-config.xml 中的系统业务流程图：

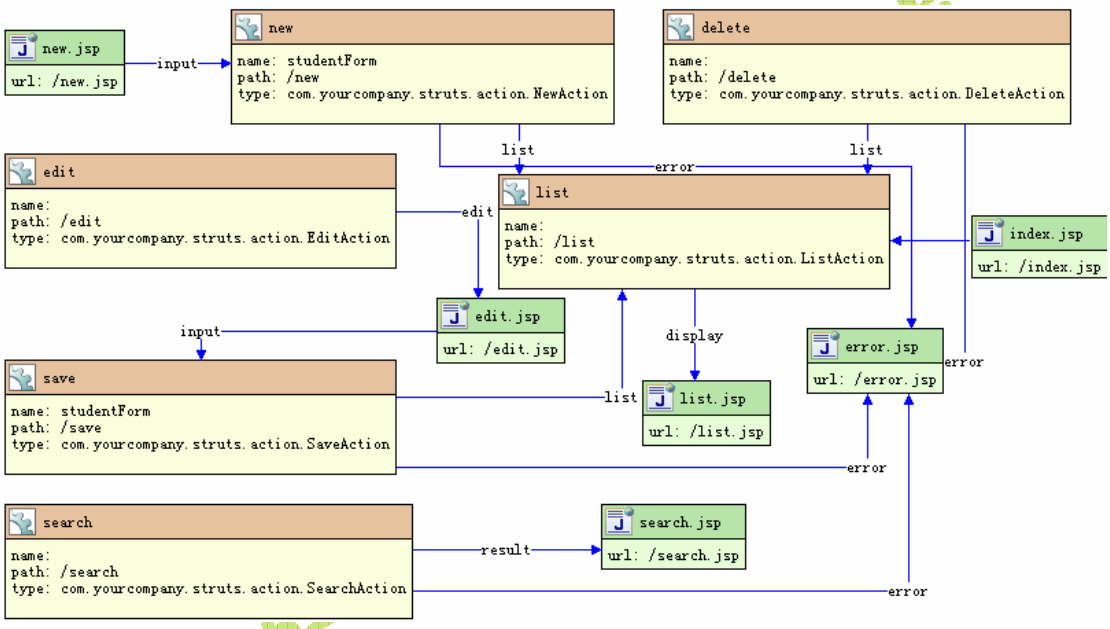

图 11.6 用户管理应用业务流程图

为了方便读者进行核对，我把这幅图所对应的配置文件源码列出来，大家可以通过在 MyEclipse 中双击打开此配置文件后，接着在主菜单栏选择 **View > Auto Layout** 来对流程图进行自动布局。然而我更希望的是大家能够对这图自己使用设计器把它画出来。下面就是配置文件 **struts-config.xml** 的源代码：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 1.2//EN"
"http://struts.apache.org/dtds/struts-config_1_2.dtd">

<struts-config>
    <data-sources />
    <form-beans>
        <form-bean name="studentForm"
            type="com.yourcompany.struts.form.StudentForm" />
```

```
</form-beans>

<global-exceptions />

<global-forwards>
</global-forwards>

<action-mappings>
    <action path="/list"
        type="com.yourcompany.struts.action.ListAction">
        <forward name="display" path="/list.jsp" />
    </action>
    <action path="/edit"
        type="com.yourcompany.struts.action.EditAction">
        <forward name="edit" path="/edit.jsp" />
    </action>
    <action attribute="studentForm" input="/edit.jsp"
        name="studentForm" path="/save" scope="request"
        type="com.yourcompany.struts.action.SaveAction">
        <forward name="list" path="/list.do" />
        <forward name="error" path="/error.jsp" />

    </action>
    <action attribute="studentForm" input="/new.jsp"
        name="studentForm" path="/new" scope="request"
        type="com.yourcompany.struts.action.NewAction">
        <forward name="list" path="/list.do" />
        <forward name="error" path="/error.jsp" />

    </action>
    <action path="/delete"
        type="com.yourcompany.struts.action.DeleteAction">
        <forward name="list" path="/list.do" />
        <forward name="error" path="/error.jsp" />
    </action>
    <action path="/search"
        type="com.yourcompany.struts.action.SearchAction">
        <forward name="result" path="/search.jsp" />
        <forward name="error" path="/error.jsp" />
    </action>

</action-mappings>

<message-resources
```

```
        parameter="com.yourcompany.struts.ApplicationResources" />
</struts-config>
```

需要说明的是为了便于编写一个默认的入口欢迎页面，方便用户的使用，我们已经把原来显示用户列表的 *index.jsp* 重命名成了 *list.jsp*。另外，到底这些功能该如何进行分工和组织，也没有绝对的标准，读者可以根据自己的需要或者习惯来进行分工和命名。例如，新建用户这个功能的命名可以是 new，也可以起名为 add 或者 create。另外通用或者共用的 Forward 路径可以设置成 **Global Forward**，即全局转向，例如上图中所显示的所有到 */error.jsp* 的转向，可以统一定义为一个全局的名为 *error* 的 Forward。

这个系统我们简要列出了几个功能的流程，如下表所示：

功能模块	流程
分页列出用户	index.jsp→list.do
创建新用户（new）	new.jsp→new.do→list.do
删除用户(delete)	Delete.do→list.do
修改用户(edit)	edit.do→edit.jsp→save.do→list.do
查找用户(search)	Search.do→search.jsp

表 11.1 业务流程说明

相比较于 Java 源代码来说，给 JSP 页面加入一些文档比较困难些，因为 Java 文件可以用 Javadoc 格式的注释来加入并生成说明，但是 JSP 没有这方面的规定和好用的文档工具。如果你想让自己的项目做的相对来说比较正规，必须对绝大多数的页面及其功能作出说明，如果能有流程说明，则更佳。这样做有两个考虑：1）便于自己核对是否全部页面都按照预期实现；2）方便自己或者别人日后对项目方便进行修改，升级和维护。软件开发本质上属于集体劳动，所以作为其中的一员，尽量不要只把系统的设计蓝图或者注意事项放在自己的脑子里，一定要想办法把它表示成可见的文档来便于大家的交流和日后维护。现在我们把 JSP 页面的功能用表格的方式列出来，如下表所示：

页面名	功能描述
index.jsp	入口页面
list.jsp	用户信息列表显示页面（核心页面）
edit.jsp	用户信息修改表单输入页面
error.jsp	出错信息显示页面
new.jsp	添加新用户表单输入页面
search.jsp	显示查找用户结果的页面

表 11.2 页面功能描述

此外还有个问题需要大家解决，就是进入修改和删除功能的 Action 时，需要在 URL 里以地址带参数的方式来传递到底是修改或者删除哪个用户。举个例子，修改用户我们用 *edit.do?id=1* 这种形式。另外用户的 ID 是不可修改的，因此如果表单里要传递这个参数，一般就以表单的 **hidden**（隐藏）类型的输入域来表示，具体代码好似这样：*<input type="hidden" name="id" value="<%=id%>">*。另外如果前台需要一些数据，后台可以通过 request.setAttribute()方法来设置，前台则需要用 request.getAttribute()方法或者 EL 表达式来读取这些值。那么最好呢，还是给我们设计的每个 Action 进行必要的描述：

Action 路径	类名	功能描述

/list	ListAction	分页显示用户信息
/edit	EditAction	根据 ID 获取用户信息并传递给 *edit.jsp*
/save	SaveAction	保存对用户信息的修改
/new	NewAction	创建新用户
/delete	DeleteAction	根据 ID 删除用户
/search	SearchAction	根据用户名查找用户，并将结果返回给 *search.jsp*

表 11.3 Struts Action 类描述

注意：上表所列的类均位于包 **com.yourcompany.struts.action** 下，然而这个包如果是实际的开发，可以在包名加入能够分开各个功能模块的英文单词在前，例如：com.yourcompany.***user***.struts.action，然后把 DAO 层的代码放入包 com...*user*.***dao*** 中。

到现在位置，应用的整个前台大框架已经确定，接下来就是考虑后台的业务层的功能了。毫无疑问，在本章我们已经确定了要使用 Spring + Hibernate 来完成后台功能。简言之，Struts 的 Action 将会调用业务层，业务层再会去调用数据访问层和其它的一些辅助功能层（工具类或者第三方类库）。那么整个系统从横向分层就分成了下面的部分：

表示层 控制层 业务层 数据访问层 辅助功能层

。当然并没有人规定到底怎样分层才是完全正确的，一般来说大家可以按照软件工程书籍上的办法进行分层，然而更现实的却是貌似大家都在按照自己公司的方式进行分层。这时对于初学者来说必须避免的一个误区就是喋喋不休的对采用何种标准或者为什么教科书上所讲的东西和现实不一样或者说对于某个东西的原理进行一次次的争论，或者用大家熟悉的话来概括：钻牛角尖。钻牛角尖表面看是勤奋好学，对某种内容具有韧性或者探索精神，然而我们却不得不面对这样一个场面：如果我们企图完全弄懂一个问题，就不得不把和它关联的问题也搞懂。但是尴尬的是，首先我们并不能无限制的得到任何自己需要的资料（大部分技术都是黑盒子，外人看不到内部实现的），另一方面，我们不得不承认人脑的记忆和理解是有限的这个现实。当你高高兴兴的以为自己弄懂了一个又一个问题时，最后却尴尬的发现似乎自己高兴过后只能记起其中的一两个（当然有个好办法就是把这些问题都记下来便于以后查找）。我想做程序员首先要有钻研精神，其次，也得有实用主义的精神。学技术，就是为了解决实际的问题的，如果你想为技术而技术，请先填饱肚子。遇到自己不会的东西怎么办？首先是能用就行，我们把它看成黑盒子就行了，就像用遥控器的人永远不需要还得懂红外线，集成电路，振荡器，发射器到底是怎么回事一样。说到这里似乎已经扯远了话题，然而看到过很多学生天天追求后台原理或者软件工程，然而却连简单的应用也作不出或者设计图都画不好。不是他们不努力，而是他们努力走向了另一个方向：钻研理论。对于初学者，第一步要的就是：能用。其余的，在这基础之上才能去谈进一步提高。而且，在国内的现状就是，绝大多数人是开发应用（简单的说就是编写固定流程的东西），而不是深入软件的底层来做开发工具或者自己动手写或者改进虚拟机。

OK，让我们回到正题。那么对于这个项目来说，当前台的结构设计的差不多的时候，我们也可以转而先去做后台的业务层，将前台的功能进行提取，看看后台能不能先把大部分都实现了。这也是实际开发两人或者多人分工协作可能出现的情况，例如：一个写前台，一个写后台。当然我们并不是要让大家非得按照软件工程或者项目管理规范，把所有的文档都写好了才能真正开始编码，我们只需要让想法表示出来，就可以了，用代码来表示自己的想法，其实也是一种很不错的办法（好像一直备受赞扬的敏捷开发——Agile，就是这样）。

11.11.3 设计业务层功能

对于业务层的设计，我们可以先写接口，然后再具体实现。当然直接写成类的方式也是没人禁止的，尤其是自己学习的时候也是很合适的做法。不过因为写接口就意味着我们并不需要写好具体的实现代码，所以设计阶段一般来说还是鼓励使用接口来做，这样大家只需要讨论好哪些功能需要，哪些不需要就成了。当然，最重要的是给这个接口加上足够多的注释（或者文档），否则这些当时讨论热烈所定下的接口中一些内容的具体意义很可能在几天之后拌饭吃了（就是忘了）。

现在让我们来具体进行业务层的设计，首先是定义一个接口 *IStudentManager*（这个以 I 开头的名称来自于 Eclipse 的源码中的定义接口的习惯，I 表示 Interface，便于开发人员一眼看出这是个接口，当然请不要误会所有的接口必须以 I 开头才能定义），把它放在包 *service* 中，用来定义用户管理模块应有的功能。放在包 *service* 中是为了和我们上一节所介绍的项目保持一致，当然用 *manager* 作为业务层的包还是别的名称都是无所谓的。先前的 *StrutsPageDemo* 项目中的类 *manager.StudentManager* 已经实现了分页部分的功能，所以分页部分的方法定义可以参考那个类中的现有代码，其余的部分需要我们分开一步步的进行考虑。所以分页部分的接口设计如下所示：

```
public int getTotalCount(); //  得到用户总数
public int getTotalPage(int pageSize) ;//  获取总页面数
public List findPagedAll(int currentPage, int pageSize);//  分页显示数据
```

。

本项目的主要功能是增删改查，再结合我们上一节所讨论的业务流程，并考虑到方法的参数和返回值等情况，我们得以一个个的定义这些功能所对应的方法声明。

对于新建用户，我们定义一个 *public boolean save(Student student)* 方法即可，返回值告诉调用者是否保存成功。

对于保存用户，我们上一节的流程是首先通过 ID 找到 Student 对象，然后再保存更新，所以一共需要两个方法的定义：

```
public Student findById(int id);//  根据 ID 查找
public boolean update(Student student);//  更新用户对象
```

对于删除用户，非常简单，根据 ID 进行删除就可以了，所以定义一个方法：

*public boolean delete(Student student)*即可，返回值同样是成功时返回 **true**，其它情况下返回 **false**。

对于查找用户，我们则只需要一个方法就可以了，在这里我们只提供根据用户名进行查找的功能，所以这个方法定义为：

```
public List<Student> findStudentByUsername(String username);
```

方法的参数是用户名，返回值则是包含找到的用户对象的列表，如果找不到的话就返回 **null**。

这样我们的初期设计就算完成了，不要忘了加上文档，那么最后的业务层类图如下所示：

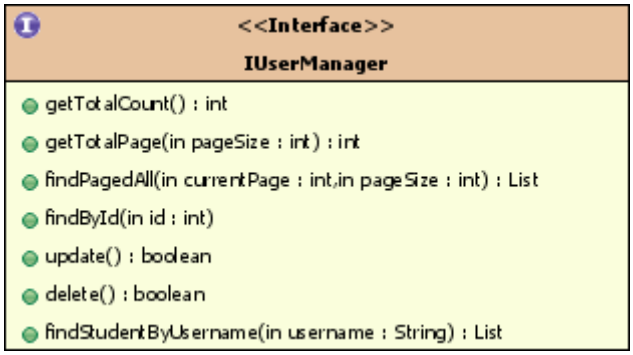

图 11.7 业务层 UML 类图

注意：这个图我们放在项目 *ssh2* 的 UML 模型仓库文件 **model.umr** 中，您可以在 MyEclipse 中双击看到它。

对应的类 **service.IStudentManager** 的完整源码清单如下所示：

```
package service;

import java.util.List;
import dao.Student;

/**
 * 用户管理模块功能抽象定义，包括了增删改查和分页。
 * @author BeanSoft
 */
public interface IStudentManager {

    /**
     * 得到用户总数
     * @return 用户记录总数
     */
    public int getTotalCount();

    /**
     * 获取总页面数.
     *
     * @param pageSize
     *            一页显示数据量
     * @return 页面总数
     */
    public int getTotalPage(int pageSize);

    /**
     * 分页显示数据.
     * @param currentPage 当前页码, 从 1 开始
     * @param pageSize 每页显示数据量
```

```
     * @return 分页后的数据列表 - List<Student>
     */
    public List<Student> findPagedAll(int currentPage, int pageSize);

    /**
     * 根据ID查找用户信息。
     * @param id 用户编号
     * @return 找到的用户对象，找不到时返回null
     */
    public Student findById(int id);

    /**
     * 更新用户对象。
     * @param student 被更新的用户
     * @return 更新成功与否
     */
    public boolean update(Student student);

    /**
     * 删除用户对象。
     * @param student 被删除的用户
     * @return 删除成功与否
     */
    public boolean delete(Student student);

    /**
     * 根据用户名查找用户。
     * @param username 用户名
     * @return 包含此用户名的用户列表
     */
    public List<Student> findStudentByUsername(String username);
}
```


至此，我们所有的初步设计工作都已完成了，下一步就可以着手进行详细设计了。但是因为这个项目相对比较规模小，所以我们接下来就进入具体的开发过程中。如果您想进行详细设计，那么您必须要把所有的 JSP 页面的样子画出来，每个类的 UML 图也要画出来，以及每个页面和类的具体实现思路也得进行说明，例如：页面中加入的表单验证脚本代码，Action 类的工作流程，DAO 层采用何种技术实现等等（当然这里可以使用 Spring 中的 HibernateDaoSupport 或者 HibernateTemplate）。不过实际开发中你会发现基本上所有的详细设计都会有纰漏或者错误之处，需要在开发过程中逐步进行必要的修改。

11.11.4 开发业务层和 DAO 层代码

到底是先做前台页面还是先实现后台的功能类呢？这个问题其实并无标准答案，全依赖

于开发人员的习惯而决定。由于在上一节我们已经将业务层接口确定，所以如果假设这个项目有两个人进行开发的话，是可以分头进行的，只需要最后在 Spring 配置文件中将接口的具体实现类指定就可以了。相对来说，这个项目的业务层代码比较简单，所以我打算先实现后台功能。前台的 JSP，Struts 的部分则放在后面进行开发。

由于后台改用Spring来实现，所以现在先参考 11.4 添加 Spring 功能一节的内容给项目加入Spring开发功能，包括创建一个bean配置文件并加入依赖的Spring类库，还不要忘了创建用于Spring中的Hibernate sessionFactory的bean定义。

因为现在的项目是复制过来的，在原来已经加入了**Hibernate 3.1 Core Libraries**，但是在 11.4 节和 11.7.4 一节我们已经讨论了所有的类库必须以复制的方式加入到项目的目录*/WebRoot/WEB-INF/lib*下，这样做的原因也已经解释过主要是为了解决Spring整合Struts过程中因为类库冲突导致的ASM报错。所以我们这个新项目在加入Spring功能之前必须先把这个引用方式的Hibernate类库删除掉。在**Package Explorer**视图中单击选中节点**Hibernate 3.1 Core Libraries**，接着选择上下文菜单中的**Build Path > Remove from Build Path**。接着大家在添加Spring类库的对话框中除了按照 11.4 节的选项进行外，类库也要多选择一个**Hibernate 3.2Core Libraries**的，如下图所示：

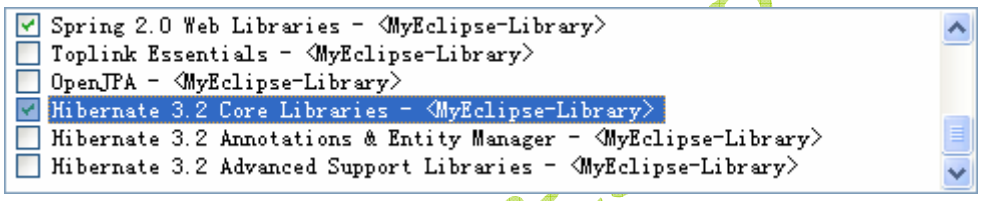

图 11.8 添加 Spring 类库时选中并复制 Hibernate 类库

为了便于以后开发过程中及时看到并纠正可能出现的错误，大家可能还需要按照 11.7.4 测试，Asm出错和log4j.properties 文件一节的内容来删除文件**asm-2.2.3.jar**并加入LOG4J的配置文件。

接下来先实现DAO层的代码。分页部分的功能可以参考原来的*StrutsPageDemo*项目的DAO的代码。可以按照 11.5 Spring 整合 Hibernate一节的内容自动生成DAO层类代码。当然最简单的办法是将项目*ssh1* 中的已经写好的DAO层类的代码复制到这里修改即可。对于这个DAO，我们只需要加入分页和更新功能即可。然后要给类加入Spring事务标注。下面列出了这个类**dao.StudentDAO**的完整代码：

```
package dao;

import java.util.List;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.hibernate.LockMode;
import org.hibernate.Query;
import org.springframework.context.ApplicationContext;
import
org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.transaction.annotation.Transactional;

/**
 * Student 的 DAO 层，加入了事务标注，分页和更新功能。
```

```
 */
@Transactional
public class StudentDAO extends HibernateDaoSupport {
    private static final Log log = LogFactory.getLog(StudentDAO.class);
    // property constants
    public static final String USERNAME = "username";
    public static final String PASSWORD = "password";
    public static final String AGE = "age";

    protected void initDao() {}

    /**
     * 得到记录总数
     *
     * @return int - 记录总数
     */
    public int getTotalCount() {
        Query q = getSession().createQuery("select count(*) from Student");

        List cc = q.list();

        Integer a = (Integer) cc.get(0);
        return a.intValue();
    }

    /**
     * 分页显示数据.
     *
     * @param currentPage
     *            当前页码, 从 1 开始
     * @param pageSize
     *            每页显示数据量
     * @return 分页后的数据列表 - List<Student>
     */
    public List findPagedAll(int currentPage, int pageSize) {
        log.debug("分页查找");
        try {

            if (currentPage == 0) {
                currentPage = 1;
            }
            String queryString = "from Student";
            Query queryObject = getSession().createQuery(queryString);
            queryObject.setFirstResult((currentPage - 1) * pageSize);
```

```
            queryObject.setMaxResults(pageSize);
            return queryObject.list();
        } catch (RuntimeException re) {
            log.error("find all failed", re);
            throw re;
        }
    }

    /**
     * 更新用户对象。
     * @param transientInstance 被更新的对象
     */
    public void update(Student transientInstance) {
        log.debug("update Student instance");
        try {
            getHibernateTemplate().update(transientInstance);
            log.debug("update successful");
        } catch (RuntimeException re) {
            log.error("update failed", re);
            throw re;
        }
    }

    /**
     * 根据用户名进行模糊查找。
     * @param username 用户名
     * @return 找到的用户列表
     */
    public List<Student> findByUsername(Object username) {
        String param = "%" + username + "%";//加入模糊匹配
        try {
            String queryString = "from Student s where s.username like ?";
            return getHibernateTemplate().find(queryString, param);
        } catch (RuntimeException re) {
            log.error("find by username failed", re);
            throw re;
        }
    }

    public void save(Student transientInstance) {
        log.debug("saving Student instance");
        try {
            getHibernateTemplate().save(transientInstance);
            log.debug("save successful");
```

```
        } catch (RuntimeException re) {
            log.error("save failed", re);
            throw re;
        }
    }

    public void delete(Student persistentInstance) {
        log.debug("deleting Student instance");
        try {
            getHibernateTemplate().delete(persistentInstance);
            log.debug("delete successful");
        } catch (RuntimeException re) {
            log.error("delete failed", re);
            throw re;
        }
    }

    public Student findById(java.lang.Integer id) {
        log.debug("getting Student instance with id: " + id);
        try {
            Student instance = (Student) getHibernateTemplate().get(
                    "dao.Student", id);
            return instance;
        } catch (RuntimeException re) {
            log.error("get failed", re);
            throw re;
        }
    }

    public List findByExample(Student instance) {
        log.debug("finding Student instance by example");
        try {
            List results =
getHibernateTemplate().findByExample(instance);
            log.debug("find by example successful, result size: "
                    + results.size());
            return results;
        } catch (RuntimeException re) {
            log.error("find by example failed", re);
            throw re;
        }
    }

    public List findByProperty(String propertyName, Object value) {
```

```
        log.debug("finding Student instance with property: " +
propertyName
                + ", value: " + value);
        try {
            String queryString = "from Student as model where model."
                    + propertyName + "= ?";
            return getHibernateTemplate().find(queryString, value);
        } catch (RuntimeException re) {
            log.error("find by property name failed", re);
            throw re;
        }
    }

    public List findByPassword(Object password) {
        return findByProperty(PASSWORD, password);
    }

    public List findByAge(Object age) {
        return findByProperty(AGE, age);
    }

    public List findAll() {
        log.debug("finding all Student instances");
        try {
            String queryString = "from Student";
            return getHibernateTemplate().find(queryString);
        } catch (RuntimeException re) {
            log.error("find all failed", re);
            throw re;
        }
    }

    public Student merge(Student detachedInstance) {
        log.debug("merging Student instance");
        try {
            Student result = (Student) getHibernateTemplate().merge(
                    detachedInstance);
            log.debug("merge successful");
            return result;
        } catch (RuntimeException re) {
            log.error("merge failed", re);
            throw re;
        }
    }
```

```
    public void attachDirty(Student instance) {
        log.debug("attaching dirty Student instance");
        try {
            getHibernateTemplate().saveOrUpdate(instance);
            log.debug("attach successful");
        } catch (RuntimeException re) {
            log.error("attach failed", re);
            throw re;
        }
    }

    public void attachClean(Student instance) {
        log.debug("attaching clean Student instance");
        try {
            getHibernateTemplate().lock(instance, LockMode.NONE);
            log.debug("attach successful");
        } catch (RuntimeException re) {
            log.error("attach failed", re);
            throw re;
        }
    }

    public static StudentDAO
getFromApplicationContext(ApplicationContext ctx) {
        return (StudentDAO) ctx.getBean("StudentDAO");
    }
}
```

需要加入的分页和更新的具体实现代码已经以粗斜体格式显示出来了。注意查找用户的方法 findByUsername()已经做了修改，加入了模糊匹配功能。至此 DAO 层的功能代码已经编写完毕了。

接下来我们来做业务层的代码，修改或者新建类 *service.StudentManager*，来实现先前定义的 *IStudentManager* 接口中的功能。为了便于使用 Spring 整合，我们将其中的 DAO 层设置成一个属性 *dao*，通过方法 *getDao()*和 *setDao()*定义，类型为 *dao.StudentDAO*。业务层的绝大部分功能都是委托给 dao 来实现的，而对于那些需要返回布尔值来确定是否操作成功的方法，我们用 try-catch 语句来实现这样的需求，即：没有异常时返回 *true*，否则返回 *false*。下面就是我们的 **service.StudentManager** 类的完整代码清单：

```
package service;

import java.util.List;
import dao.Student;

/**
 * 用户管理模块实现类
```

```
 * @author BeanSoft
 */
public class StudentManager implements IStudentManager {
    /** 用户管理 DAO */
    private dao.StudentDAO dao;

    public int getTotalCount(){
        return dao.getTotalCount();
    }

    public int getTotalPage(int pageSize) {
        int totalCount = getTotalCount();

        // 得到页面总数
        int totalPageCount = ((totalCount + pageSize) - 1) / pageSize;

        return totalPageCount;
    }

    public List<Student> findPagedAll(int currentPage, int pageSize) {
        return dao.findPagedAll(currentPage, pageSize);
    }

    public boolean delete(Student student) {
        try {
            dao.delete(student);
            return true;
        } catch (Exception e) { }
        return false;
    }

    public Student findById(int id) {
        return dao.findById(id);
    }

    public List<Student> findStudentByUsername(String username) {
        return dao.findByUsername(username);
    }

    public boolean update(Student student) {
        try {
            dao.update(student);
            return true;
        } catch (Exception e) { }
```

```
        return false;
    }

    public dao.StudentDAO getDao() {
        return dao;
    }

    public void setDao(dao.StudentDAO dao) {
        this.dao = dao;
    }
}
```

。这个业务层只要通过调用 *setDao()*方法设置了具体的 DAO 对象的时候，就可以正常工作了。

接下来，就是使用Spring的配置文件将这些东东组装起来，具体的配置文件源码其实和11.5 Spring 整合 Hibernate一节的Spring 配置文件 **applicationContext.xml**的代码是一样的，外加一个StudentManager的bean定义，参考 11.6 开发业务层代码一节的内容。同样的最后我们需要对业务层进行测试，具体的代码仍然参考 11.6 一节的内容。

至此为止，我们的业务层就算开发完毕了，接下来要做的就是开发前台的 JSP 和 Struts Action 类。

11.11.5 开发前台页面流程

前台怎么开发呢？先前的时候我们已经在 11.11.2 用Struts设计器制作前台业务流程一节中完成了必要的功能设计。对于我们来说，按照**表 11.1 业务流程说明**中的流程进行开发，遇到JSP，就创建或者修改JSP页面；遇到.do，就根据**图 11.6** 的流程开发或者修改Action类。这样一个流程一个流程的开发，最后再进行测试，就可以完成了。

具体的编码过程不再赘述，按照流程创建一个个JSP页面，最后再修改后台的Action类代码。需要提示大家的是，像EditAction这样的功能的写法，需要根据ID查找一个用户然后传递给前台进行修改，虽然没有明确的在设计图中反映出来，如果不幸找不到用户信息的话，务必需要告诉使用人员要修改的用户找不到，可以转向到*/error.jsp*进行显示。接下来要做的就是先将传递过来的字符串类型的ID参数通过**Integer.parseInt(String)**方法转换成**int**，然后再将通过后台业务层（也即*IStudentManager*）找到的用户信息通过request对象设置属性传递给前台的JSP页面，前台页面则使用EL表达式加以显示。实现过程中一定要全面考虑到客户端验证（JavaScript表单验证）和服务器端验证（在Action类中进行必要的数据验证），不要轻易的让系统出现莫明其妙的 500 或者 404 错误页面。还有就是中文问题，如果在Tomcaat下开发，则建议按照 9.4.4 调整生成的代码一节加入解决中文字符编码的过滤器。

现在我们按照**表 11.1 业务流程说明**来列出 JSP 和 Action 类的代码清单。为了读者进行对照，我们先在这里列出 src 目录和 WebRoot 目录下的文件列表截图。

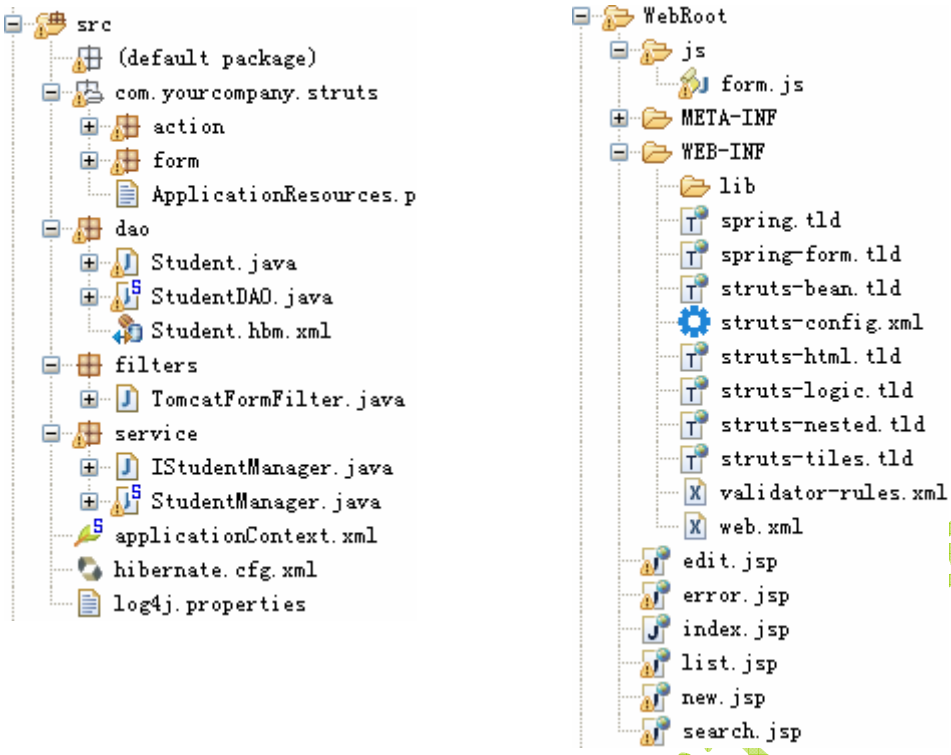

图 11.9 项目 ssh2 源代码目录结构

文件 **js/form.js** 是抽取出来公用的表单验证功能的脚本代码，其代码清单如下所示：

```
// 验证表单输入不为空的脚本代码
function checkForm(form) {
    if(form.username.value == "") {
        alert("用户名不能为空!");
        form.username.focus();
        return false;
    }

    if(form.password.value == "") {
        alert("密码不能为空!");
        form.password.focus();
        return false;
    }

    if(form.age.value == "") {
        alert("年龄不能为空!");
        form.age.focus();
        return false;
    }

    // 确保年龄有效并在 0 ~ 120之间
    if(isNaN(parseInt(form.age.value)) || form.age.value <=0 ||
form.age.value > 120) {
```

```
        alert("请输入有效的年龄!");
        form.age.focus();
        return false;
    }

    return true;
}
```

接下来是一个显示出错信息的 **error.jsp**：

```
<%@ page language="java" pageEncoding="GBK"%>
<%
String path = request.getContextPath();
String basePath =
request.getScheme()+"://"+request.getServerName()+":"+request.getServ
erPort()+path+"/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">

    <title>出错了！</title>
  </head>

  <body>
    出错了！<br/>
    详细信息是：<br/>
    ${message}<br/><br/>
    <a href="javascript:history.back();">返回</a>
  </body>
</html>
```

出错的信息可以放在 **request** 的 **attribute** 里面，例如调用 *request.setAttribute("message","出错了！")*;之后相关的出错信息就可以显示在这个页面中。另外还提供了一个**返回**链接让用户能够回去继续操作。

先看第一个流程 **index.jsp→list.do**。用户进来之后默认所看到的首页面的 **index.jsp** 的源代码清单：

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html" %>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html:html lang="true">
  <head>
    <html:base />
```

```
    <title>欢迎使用用户管理系统</title>

  </head>

  <body>
    <a href="list.do">查看用户列表</a> <br>
  </body>
</html:html>
```

在这个页面中有个标签*<html:base />*，它的显示作用是什么呢？大家通过查看运行时的HTML源代码就可以看到其输出是：*<base href="http://localhost:8080/ssh2/">*，这句话可以让我们的页面总能正确找到引用的图片，CSS等外部资源。假设页面有个图片写着：*<img src="gif/a.gif">*,那么即使这个页面的内容通过forward变成了*/ssh2/user/welcome.do*，而图片依然会从地址 *http://localhost:8080/ssh2/gif/a.gif* 来读取，反过来就不行了。

这个页面只有一个链接，顺着**查看用户列表**超链接浏览过去的是显示分页的 Struts 的控制器 **ListAction.java**：

```
package com.yourcompany.struts.action;

import java.util.List;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import dao.Student;
import service.IStudentManager;

/** 分页显示用户信息 */
public class ListAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        // 分析当前页码
        String pageString=request.getParameter("page");
        if(pageString == null || pageString.length() == 0) {
            pageString = "1";
        }
        int currentPage= 0 ;
        try {
            currentPage = Integer.parseInt(pageString);// 当前页码
        } catch(Exception e) {}

        if(currentPage == 0) {
            currentPage = 1;
        }

        int pageSize = 3;//每页显示的数据数
        // 读取数据
```

```
        List<Student> users = manager.findPagedAll(currentPage,
pageSize);

        request.setAttribute("users",users);// 保存用户列表

        request.setAttribute("totalPage",
manager.getTotalPage(pageSize));// 保存总页数
        request.setAttribute("totalCount", manager.getTotalCount());//
保存记录总数
        request.setAttribute("currentPage", currentPage);// 保存当前页码
        return mapping.findForward("display");
    }

    private IStudentManager manager;

    public IStudentManager getManager() {
        return manager;
    }

    public void setManager(IStudentManager manager) {
        this.manager = manager;
    }
}
```

这个页面的功能代码细节可以参考我们以前章节对于分页的讨论。当取到了数据和分页相关的信息后，我们将其存放在 request 的属性里面，然后转到前台的显示层用户列表页面 **list.jsp** 来显示用户和功能链接:

```
<%@ page contentType="text/html;charset=GBK"%>
<%-- 我们使用 JSTL 来访问数据 --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%
String path = request.getContextPath();
String basePath =
request.getScheme()+"://"+request.getServerName()+":"+request.getServ
erPort()+path+"/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">

    <title>用户列表页面</title>
    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="cache-control" content="no-cache">
```

```
    <meta http-equiv="expires" content="0">
<style>
/* 给链接加入鼠标移过变色和去除下划线功能 */
a:hover {color:red;text-decoration:none}
</style>
  </head>

  <body><b>用户列表页面</b><br>
  <%-- 输出用户列表 --%><br>
  <table width="80%" border="1"  cellpadding="0"
style="border-collapse: collapse; " bordercolor="#000000">
  <tr>
  <td><b>用户ID</b></td>
  <td><b>用户名</b></td>
  <td><b>操作</b></td>
  </tr>
  <c:forEach items="${users}" var="user" >
  <tr>
  <td>${user.id}</td>
  <td>${user.username}</td>
  <td><a href="edit.do?id=${user.id}">修改</a> <a
href="delete.do?id=${user.id}">删除</a></td>
  </tr>
  </c:forEach>
  </table> 共${totalCount}个用户

  第${currentPage}页/共${totalPage}页
    <%-- 输出页面跳转代码, 分链接和静态文字两种 --%>
    <c:if test="${currentPage > 1}">
      [ <a
href="${pageContext.request.contextPath}/list.do?page=${currentPage-1
}">上一页</a> ]
    </c:if>
    <c:if test="${currentPage <= 1}">
      [ 上一页 ]
    </c:if>
    <c:if test="${currentPage < totalPage}">
      [ <a
href="${pageContext.request.contextPath}/list.do?page=${currentPage+1
}">下一页</a> ]
     </c:if>
    <c:if test="${currentPage >= totalPage}">
      [ 下一页 ]
    </c:if>
```

```
    <%-- 输出 JavaScript 跳转代码 --%>
    <script>
    // 页面跳转函数
    // 参数: 包含页码的表单元素，例如输入框，下拉框等
    function jumpPage(input) {
    // 页码相同就不做跳转
    if(input.value == ${currentPage}) {
        return;
    }
    var newUrl = "${pageContext.request.contextPath}/list.do?page=" +
input.value;
    document.location = newUrl;
    }
    </script>
    转到
      <!-- 输出 HTML SELECT 元素，并选中当前页面编码 -->
      <select onchange='jumpPage(this);'>

      <c:forEach var="i" begin="1" end="${totalPage}">
        <option value="${i}"

        <c:if test="${currentPage == i}">
        selected
        </c:if>

    >第${i}页</option>
      </c:forEach>

      </select>
      输入页码: <input type="text" value="${currentPage}" id="jumpPageBox"
size="3">
      <input type="button" value="跳转"
onclick="jumpPage(document.getElementById('jumpPageBox'))"><br>
    <a href="new.jsp">添加用户</a>

    <form action="search.do">
    <fieldset><legend>查找用户</legend>
    用户名:<input name="username"> <input type="submit" value="查找">
    </fieldset>
    </form>
  </body>
</html>
```

这个页面的代码看起来稍微有些复杂，它是我们这个用户管理应用的中心页面，所有的功能都从这里出发。为了方便读者比较，我们将它运行时的截图放在下面：

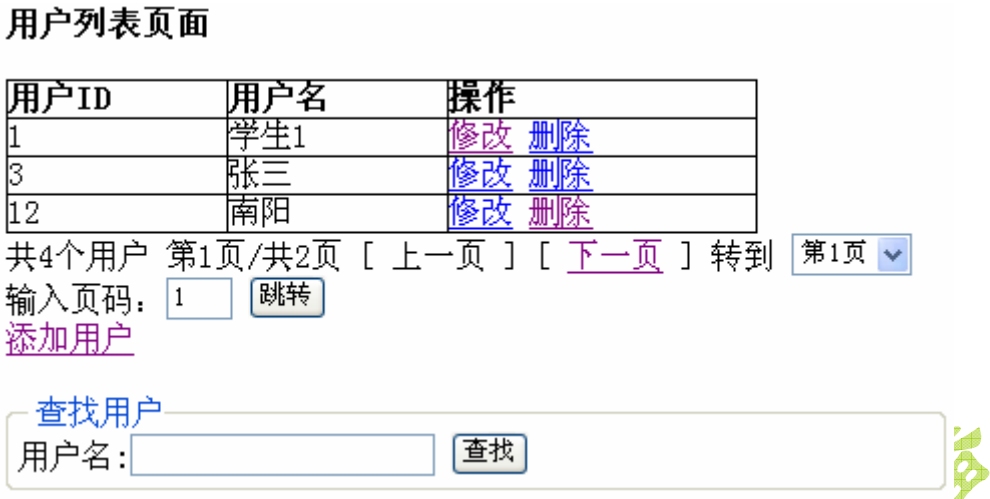

图 11.10 用户列表页面外观

当点击**添加用户**链接后，我们就进入另一个模块：添加用户，流程是 **new.jsp→new.do→list.do**。首先要显示一个添加新用户表单输入页面 **edit.jsp**，其源代码清单如下所示：

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
<%@ taglib uri="http://struts.apache.org/tags-html" prefix="html"%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <html:base />

    <title>添加用户</title>

    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="cache-control" content="no-cache">
    <meta http-equiv="expires" content="0">
    <script type="text/javascript" src="js/form.js">
    </script>

  </head>

  <body>
  <h3>添加用户</h3>
    <html:form action="/new" onsubmit="return checkForm(this);">

    <table width="100%" border="0">
        <tbody><tr>
        <td> 用户名：</td>
        <td> <html:text property="username"/> <br></td></tr>
        <tr>
```

```
        <td> 密码:</td>
        <td> <html:text property="password"/></td></tr>

        <tr>
        <td> 年龄：</td>
        <td> <html:text property="age"/></td></tr>
        <tr>
        <td> <input type="submit" value="添加" name="button1"></td>
        <td> <input type="Reset" value="重填"
name="button2"></td></tr>
        </tbody>
    </table>
    </html:form>

    <input type="button" onclick="document.location='list.do';" value="
返回列表">
  </body>
</html>
```

在这个页面中，表单是用 Struts 的 HTML 标签库编写的，而表单提交时仍然加有 JavaScript 编写的表单验证代码，因此，虽然这里用 Struts 标签库编写，也还是可以加入对应的客户端验证代码的。另外切莫忘记让使用者有来有去，因此页面底部还提供了一个返回按钮供使用者返回列表页面。此页面运行时截屏如下：

图 11.11 添加用户页面外观

当用户提交按钮后，就进入了保存用户的 Action 类 **NewAction.java**：

```
package com.yourcompany.struts.action;

import javax.servlet.http.*;
import org.apache.struts.action.*;
import dao.Student;
import service.IStudentManager;
import com.yourcompany.struts.form.StudentForm;
/** 创建新用户 */
public class NewAction extends Action {
```

```
    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        StudentForm studentForm = (StudentForm) form;
        try {
            Student student = new Student();

            student.setAge(studentForm.getAge());
            student.setUsername(studentForm.getUsername());
            student.setPassword(studentForm.getPassword());

            if(manager.save(student)) {
                return mapping.findForward("list");
            } else {
                request.setAttribute("message", "用户信息保存失败。");
            }

        } catch (Exception e) {
            request.setAttribute("message", "用户信息保存失败:" + e);
        }


        return mapping.findForward("error");
    }

    private IStudentManager manager;

    public IStudentManager getManager() {
        return manager;
    }

    public void setManager(IStudentManager manager) {
        this.manager = manager;
    }
}
```

这个 Struts 控制器的功能是相当的简单，首先创建一个 *Student* 实体类，然后从 *StudentForm* 中将相关的信息复制进去，最后委托业务层 *IStudentManager* 来保存，保存成功后重新回到 *list* 这个转向（列表页面*/list.do*），如果保存失败则跳转到 *error* 这个转向（出错信息*/error.jsp*）。

接下来我们要介绍的这个模块也更为简单，即：删除用户，流程是 **Delete.do→list.do**。我们在超链接的地址中加入 *id* 这个参数来指定要删除哪个用户，例如：*delete.do?id=3*。之后在 Action 中通过读取表单参数 *id*，就可以获取到底是哪个用户要被删除。之后委托给业务层来完成，成功后回到 *list*，失败则转到 *error*。此功能的实现代码 **DeleteAction.java** 清单如下所示：

```
package com.yourcompany.struts.action;
```

```
import javax.servlet.http.*;
import org.apache.struts.action.*;
import dao.Student;
import service.IStudentManager;

/**根据ID删除用户，例如delete.do?id=3。*/
public class DeleteAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        try {
            int id = Integer.parseInt(request.getParameter("id"));
            Student student = manager.findById(id);

            if(manager.delete(student)) {
                return mapping.findForward("list");
            }
        } catch (Exception e) {

        }

        request.setAttribute("message", "无法删除用户。");
        return mapping.findForward("error");
    }

    private IStudentManager manager;

    public IStudentManager getManager() {
        return manager;
    }

    public void setManager(IStudentManager manager) {
        this.manager = manager;
    }
}
```

接下来我们要看的是用户修改功能，它的流程是：**edit.do→edit.jsp→save.do→list.do**。一共分两步走：第一步，根据 ID 获取到用户的信息并显示在修改信息的 JSP 页面中；第二步，提交并保存被修改过的用户信息。先来看看根据 ID 获取用户信息的 **EditAction.java**：

```
package com.yourcompany.struts.action;

import javax.servlet.http.*;
import org.apache.struts.action.*;
```

```
import dao.Student;
import service.IStudentManager;

/** 根据ID获取用户信息并传递给edit.jsp */
public class EditAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        try {
            int id = Integer.parseInt(request.getParameter("id"));
            Student student = manager.findById(id);

            if(student != null) {
                request.setAttribute("student", student);
                return mapping.findForward("edit");
            }
        } catch (Exception e) {}

        request.setAttribute("message", "请选择有效的用户进行修改。");
        return mapping.findForward("error");
    }

    private IStudentManager manager;

    public IStudentManager getManager() {
        return manager;
    }

    public void setManager(IStudentManager manager) {
        this.manager = manager;
    }
}
```

这个 Action 先根据地址栏里带过来的参数（例如 *edit.do?id=1*）来从业务层获得要修改的用户对象，之后成功就转向到前台的用户修改信息输入页面 **edit.jsp** 来进行修改：

```
<%@ page language="java" pageEncoding="GBK"%>
<%
String path = request.getContextPath();
String basePath =
request.getScheme()+"://"+request.getServerName()+":"+request.getServ
erPort()+path+"/";
%>

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

```
  <head>
    <base href="<%=basePath%>">

    <title>修改用户信息</title>

    <script type="text/javascript" src="js/form.js">
    </script>

  </head>

  <body><h3>
    修改用户信息</h3>

    <form action="save.do" method="post" onsubmit="return
checkForm(this);">
    <input type="hidden" value="${student.id}" name="id">
    <table width="100%" border="0">
<tbody><tr>
<td> 用户名：</td>
<td> <input type="text" value="${student.username}"
name="username"> <br></td></tr>
<tr>
<td> 密码:</td>
<td> <input type="text" value="${student.password}"
name="password"></td></tr>

<tr>
<td> 年龄：</td>
<td> <input type="text" value="${student.age}"
name="age"></td></tr>
<tr>
<td> <input type="submit" value="保存" name="button1"></td>
<td> <input type="Reset" value="重填" name="button2"></td></tr>
</tbody></table>
    </form>
    <input type="button" onclick="document.location='list.do';" value="
返回列表">
  </body>
</html>
```


如前文所述，在修改用户的时候有个问题必须解决，那就是被修改用户的 ID 该如何传递。一般来说我们是通过表单的隐藏字段解决的（*<input type="hidden" value="${student.id}" name="id">*），也可是显示在一个只读的输入框中（*<input type="text" value="${student.id}" name="id" readonly="readonly">*）。同样的在本页面中也加入了必要的 JavaScript 编写的表单验证功能代码。另外为了方便用户，还加入了一个**返回列表**的按钮供其随时返回用户列

表页面，这些 JavaScript 进行跳转的功能代码我们加在了按钮的 **onclick** 事件中。此页面运行时的截屏如下所示：

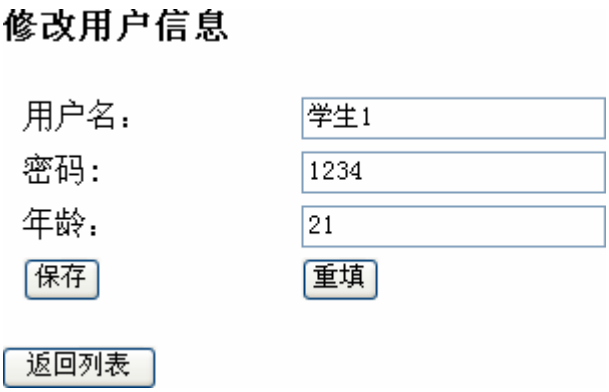

图 11.12 修改用户信息页面外观

当页面提交后，就转而执行*/save.do* 所对应的 **SaveAction.java** 代码中的更新用户的功能。注意看我们的修改功能的实现代码，首先我们是从业务层（委托给 DAO 层）根据所传递过来的 id 来查找对应的实体类对象，然后再把修改过的值复制到此实体类上，之后再进行保存。为什么要这样做呢？这时因为随着项目的开发，很可能会出现以后给 Student 加入了其它的信息，例如住址（Address），此时如果你直接新建一个 Student 对象，然后把此处修改过的值赋给它然后进行 update 的话，就会漏掉新加入的属性，从而导致只有修改过的属性才保存了下来，而其它的属性全部被置空的错误出现。所以常说的写代码要注意可扩展性，就体现在这里。我们的代码不能因为有一个类新加入了一些新的属性，就有故障发生，当然这大多时候都是逻辑错误，不是编译错误，直接看代码不思考一下是解决不了的。OK，回到正题。更新成功，就跳转到用户列表页面，否则就转向出错页面显示出错信息。

最后一个功能模块，乃是根据用户名模糊查找，其流程为：**Search.do→search.jsp**。这个功能相对比较简单，首先其实在 *list.jsp* 中已经设置了一个表单来进行搜索，位于页面的最下方。我们先来看看 **SearchAction.java** 的源代码清单：

```
package com.yourcompany.struts.action;

import java.util.List;
import javax.servlet.http.*;
import org.apache.struts.action.*;
import dao.Student;
import service.IStudentManager;

/** 根据用户名查找用户，并将结果返回给search.jsp */
public class SearchAction extends Action {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
            HttpServletRequest request, HttpServletResponse response) {
        List<Student> users =
manager.findStudentByUsername(request.getParameter("username"));

        if(users == null) {
```

```
            request.setAttribute("message", "找不到用户信息");
            return mapping.findForward("error");
        }
        request.setAttribute("users",users);// 保存用户列表
        return mapping.findForward("result");
    }

    private IStudentManager manager;

    public IStudentManager getManager() {
        return manager;
    }

    public void setManager(IStudentManager manager) {
        this.manager = manager;
    }
}
```

功能非常直接，查找后将列表存入 request 的属性中，带到前台；如果找不到则定向到出错页面告诉用户找不到。前台显示用户列表的页面 **search.jsp** 其实大部分代码都是和 *list.jsp* 是一样的，其源代码清单如下所示：

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ page contentType="text/html;charset=GBK"%>
<%-- 我们使用 JSTL 来访问数据 --%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%
String path = request.getContextPath();
String basePath =
request.getScheme()+"://"+request.getServerName()+":"+request.getServ
erPort()+path+"/";
%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<html>
  <head>
    <base href="<%=basePath%>">

    <title>用户查找结果</title>
    <meta http-equiv="pragma" content="no-cache">
    <meta http-equiv="cache-control" content="no-cache">
    <meta http-equiv="expires" content="0">
<style>
/* 给链接加入鼠标移过变色和去除下划线功能 */
a:hover {color:red;text-decoration:none}
</style>
```

```
  </head>

  <body><b>用户查找结果</b><br>
  <%-- 输出用户列表 --%><br>
  <table width="80%" border="1"  cellpadding="0"
style="border-collapse: collapse; " bordercolor="#000000">
  <tr>
  <td><b>用户ID</b></td>
  <td><b>用户名</b></td>
  <td><b>密码</b></td>
  <td><b>年龄</b></td>
  <td><b>操作</b></td>
  </tr>
  <c:forEach items="${users}" var="user" >
  <tr>
  <td>${user.id}</td>
  <td>${user.username}</td>
  <td>${user.password}</td>
  <td>${user.age}</td>
  <td><a href="edit.do?id=${user.id}">修改</a> <a
href="delete.do?id=${user.id}">删除</a></td>
  </tr>
  </c:forEach>
  </table>

    <form action="search.do">
    <fieldset><legend>查找用户</legend>
    用户名:<input name="username"> <input type="submit" value="查找">
    </fieldset>
    </form>

    <input type="button" onclick="document.location='list.do';" value="
返回列表">
  </body>
</html>
```

俗话说做程序员，是越做越轻松，道理就在这里。第一遍开发总是很困难的，以后你就可以躺在自己的成果上，复制粘贴＋修改，很快就完工了。当然，好的人是会精益求精，直到最后将某一固定模块做到自己很满意为止。在这个页面，首先显示的是查找到的用户信息，然后也提供了一个**查找用户**的输入框和**返回列表**的按钮。

至此为止，我们的前台功能都已经搞定了。

11.11.6 整合 Spring，Struts 和 Hibernate

项目开发的最后，就是修改 Struts 的配置文件和 Spring 的配置文件，将 Action 和业务

层以及 Hibernate 的 DAO 整合在一起，关于详细的内容，在前几节以及第十章都有过讨论了。现在为了便于读者对照，我把这三个配置文件的代码清单给大家列出来。

Spring 的 **applicationContext.xml**：

```
<?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-2.0.xsd
      http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
      http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <tx:annotation-driven transaction-manager="transactionManager"
        proxy-target-class="true" />

    <bean id="sessionFactory"

    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean
">
        <property name="configLocation"
            value="classpath:hibernate.cfg.xml">
        </property>
    </bean>

    <bean id="StudentDAO" class="dao.StudentDAO">
        <property name="sessionFactory">
            <ref bean="sessionFactory" />
        </property>
    </bean>

    <!-- 声明一个 Hibernate 3 的 事务管理器供代理类自动管理事务用 -->
    <bean id="transactionManager"

    class="org.springframework.orm.hibernate3.HibernateTransactionMan
ager">
        <property name="sessionFactory">
            <ref local="sessionFactory" />
        </property>
    </bean>
```

```
<!-- 用户业务类 -->
<bean id="studentManager" class="service.StudentManager">
    <property name="dao">
        <ref local="StudentDAO" />
    </property>
</bean>

<!-- Struts Action 类 -->

<bean name="/list"
    class="com.yourcompany.struts.action.ListAction">

    <!-- 注入业务层的bean定义 -->
    <property name="manager">
        <ref local="studentManager" />
    </property>
</bean>

<bean name="/edit"
    class="com.yourcompany.struts.action.EditAction">
    <property name="manager">
        <ref local="studentManager" />
    </property>
</bean>

<bean name="/save"
    class="com.yourcompany.struts.action.SaveAction">
    <property name="manager">
        <ref local="studentManager" />
    </property>
</bean>

<bean name="/delete"
    class="com.yourcompany.struts.action.DeleteAction">
    <property name="manager">
        <ref local="studentManager" />
    </property>
</bean>

<bean name="/new" class="com.yourcompany.struts.action.NewAction">
    <property name="manager">
        <ref local="studentManager" />
    </property>
</bean>
```

```
    <bean name="/search"
        class="com.yourcompany.struts.action.SearchAction">
        <property name="manager">
            <ref local="studentManager" />
        </property>
    </bean>
</beans>
```

Hibernate 的 **hibernate.cfg.xml**：

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"

"http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>
        <!-- 显示后台的 SQL, 便于调试 -->
        <property name="show_sql">true</property>
        <property name="connection.username">root</property>
        <property name="connection.url">

    jdbc:mysql://localhost:3306/test?useUnicode=true&amp;amp;characte
rEncoding=GBK
        </property>
        <property name="dialect">
            org.hibernate.dialect.MySQLDialect
        </property>
        <property
name="myeclipse.connection.profile">mysql5</property>
        <property name="connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <mapping resource="dao/Student.hbm.xml" />

    </session-factory>

</hibernate-configuration>
```

Struts 的 **struts-config.xml**：

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD
Struts Configuration 1.2//EN"
"http://struts.apache.org/dtds/struts-config_1_2.dtd">
```

```
<struts-config>
    <data-sources />
    <form-beans>
        <form-bean name="studentForm"
            type="com.yourcompany.struts.form.StudentForm" />
    </form-beans>

    <global-exceptions />

    <global-forwards>
    <forward name="error" path="/error.jsp" />
    </global-forwards>

    <action-mappings>
        <action path="/list"
            type="com.yourcompany.struts.action.ListAction">
            <forward name="display" path="/list.jsp" />
        </action>
        <action path="/edit"
            type="com.yourcompany.struts.action.EditAction">
            <forward name="edit" path="/edit.jsp" />
        </action>
        <action attribute="studentForm" input="/edit.jsp"
            name="studentForm" path="/save" scope="request"
            type="com.yourcompany.struts.action.SaveAction">
            <forward name="list" path="/list.do" />
            <forward name="error" path="/error.jsp" />

        </action>
        <action attribute="studentForm" input="/new.jsp"
            name="studentForm" path="/new" scope="request"
            type="com.yourcompany.struts.action.NewAction">
            <forward name="list" path="/list.do" />
            <forward name="error" path="/error.jsp" />

        </action>
        <action path="/delete"
            type="com.yourcompany.struts.action.DeleteAction">
            <forward name="list" path="/list.do" />
            <forward name="error" path="/error.jsp" />
        </action>
        <action path="/search"
            type="com.yourcompany.struts.action.SearchAction">
```

```
            <forward name="result" path="/search.jsp" />
            <forward name="error" path="/error.jsp" />
        </action>

    </action-mappings>

    <controller

    processorClass="org.springframework.web.struts.DelegatingRequestP
rocessor" />

    <message-resources
        parameter="com.yourcompany.struts.ApplicationResources" />

    <plug-in

    className="org.springframework.web.struts.ContextLoaderPlugIn">
        <set-property property="contextConfigLocation"
            value="/WEB-INF/classes/applicationContext.xml" />
    </plug-in>
</struts-config>
```

11.11.7 发布，运行，测试

开发完了，我们就要在自己的电脑上发布，运行，在浏览器里测试一下。不过这时候如果您参考 11.9 测试运行一节的内容来发布的话，会发现发布后的项目放在了Tomcat的*webapps/StrutsPageDemo*下面，要访问的地址也是以此开头的。怎么办？这是因为我们的项目是复制过来的，所以一些配置信息依然是旧的。在**Package Explorer**视图中选中项目*ssh2*，点击右键选择菜单**Properties**，在弹出的项目属性对话框中选中左侧的选项树上的节点**MyEclipse > Web**，然后在右侧的**Context Root**标签页下的**Web Context-root**右侧输入框中将原来的值*StrutsPageDemo*修改为*ssh2*，然后再重新发布项目即可发布到正确的目录下，此过程如图 11.13 所示。然后即可按照正确过程发布，运行，测试即可。

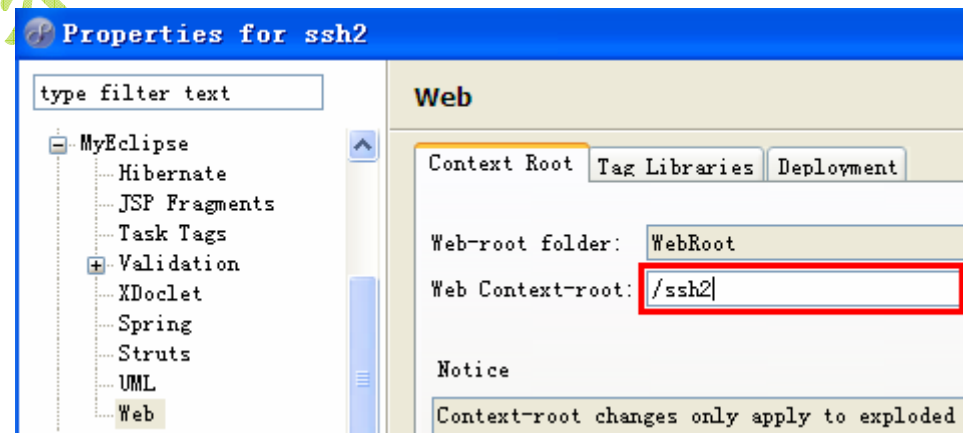

图 11.13 修改发布的目录

发布完毕并运行后，就可以启动浏览器，键入 http://localhost:8080/ssh2/ 进行测试了。

11.11.8 思考与练习

假设用户已经浏览到了第 20 页，然后新建或者修改信息之后，用户发现自己又回到了第一页，不得不翻半天再来到了第 20 页，这也许会对用户的心理留下很沉重的阴影。你想想看有没有什么办法能让用户操作完成后总是回到正确的页面呢？

答案提示：可以参考用户修改这一模块的做法（用隐藏字段），或者通过在 cookie 或者 session 中放置当前页码数来解决此问题。

请读者自己练习一下，将某个模块加入此返回后保存页码的功能。

另外，如果处于贴心的考虑，搜索用户模块的也要加入分页的功能。

还有个很严重的问题，以前一直没给大家提起，现在是时候了，那就是假设我们要用这个例子里面的代码来做用户注册的话，要严防用户输入 HTML 代码或者 JavaScript 代码。举个例子，假设有人输入了自己的名字是 *BeanSoft*，您可以自己在浏览器里试试是什么效果，不光是显示了个大字，鼠标移过去的时候还会弹出 JavaScript 警告对话框。其实要解决这个问题也很简单，在所有字符串类型的输入表单域获取后如果要避免 HTML 代码，就用字符串类的 *public String replaceAll(String regex, String replacement)*方法将左尖括号**<**替换为转义字符**<**，右尖括号**>**替换为**>**就可以了，下面是一段示意代码：

```
StudentForm studentForm = (StudentForm) form;
String username = studentForm.getUsername();
if(username != null) {
   username = username.replaceAll(“<”, “&lt;”). replaceAll(“>”, “&gt;”);
   //... 更多处理代码
}
```

.

总之，如果想改进，总是有改进的地方的。

最后，一般这样的用户管理界面是要加密码保护的，大家可以结合过滤器和以前介绍的用户登录，来给所有的页面都加入未登录不能操作的保护功能。

11.12 collections.SequencedHashMap 异常的解决方案

在实际的开发中，不少读者都遇到了这个使用 MyEclipse 6 开发 SSH 时的出错的问题。具体情况如下：用 MyEclipse 6.0.0 开发 Struts + Hibernate 应用的时候，单独测试 *Hibernate 的类没有问题，但是当 Web 层和 Struts 整合后，就抛出如下异常：*

%%%% Error Creating SessionFactory %%%% java.lang.SecurityException: class "org.apache.commons.collections.SequencedHashMap"'s signer information does not match signer information of other classes in the same package

因为我用的 6.0.1，所以怎么也没出现这个异常。后来Google搜到很多人遇到了这个故障，猜测是*commons-collections.jar*的问题，可能和MyEclipse 6.0.0 自带的包有问题导致。例如

下面的是一个解决方案：**http://ttitfly.javaeye.com/blog/131955**

解决方案：

1）下载 MyEclipse 6.0.1 来开发；

2）或者去 **http://commons.apache.org/collections/** 下载一个新的包，把原来的commons-collections-xxx.jar给删了，然后用新的包替换。注意：一定要到应用的发布目录去做这个工作，而且替换后不能重新发布应用，然后立即重启Tomcat再测试。

如果再报下面的错误：

ERROR [org.hibernate.proxy.BasicLazyInitializer] - CGLIB Enhancement failed: dao.User
java.lang.NoSuchMethodError:
org.objectweb.asm.ClassVisitor.visit(IILjava/lang/String;Ljava/lang/String;[Ljava/lang/String;Ljava/lang/String;)V
at net.sf.cglib.core.ClassEmitter.begin_class(ClassEmitter.java:77)

则是因为 Spring 和 Hibernate 共用的一些 jar 文件发生了版本冲突，删除 WEB-INF/lib/asm-2.2.3.jar 即可。关于这个 ASM 出错的问题我们在本章前面部分已经提及。

为了方便读者，我特地在第 11 章的代码中（文件名是 *11 章代码.rar*）加入了一个 *commons-collections-2.1.1.jar* 来期望能对大家有所帮助。如果读者自己有条件，尽可以自己去 Hibernate 和 Spring 的官方站点来下载官方发布的类库，一般来说用它们提供的是不会出问题的。而 MyEclipse 6 自带的类库，早期版本总是有一些问题。

11.13 小结

在本章中我们详细讨论了 Spring 整合 Struts 的方式和原理，并通过实际开发一个简单登录的例子和一个稍微复杂些的增删改插例子，展示了 SSH（Struts 1、Spring 2 和 Hibernate3.2）开发的完整过程，使读者能够真实的感受到在 MyEclipse 6 下进行 SSH 应用开发的全过程。现在您可以自己动手做一些简单的练习了！读者还可以结合以前所介绍的 Struts 文件上传等内容来完成类似于简易 BBS，带后台管理的留言板之类的练习。

结合第十章和第十一章，读者能够掌握常见的 SSH 整合开发的概念和方法，大家可以注意到从 Spring＋Hibernate 的 3 种方式，再配上 Spring＋Struts 的 3 种方式，一共大约有不下 9 种 SSH 开发的组合。总而言之，由于 SSH 是三家组织开发的框架，要组合在一起，显然也不是那么容易。然而，对于读者来说，只要了解并掌握其中的一两种方式，再结合自己公司的实际项目，本着实用为主的目的，相信很快就能够胜任一些常见的开发任务，千万不要为了 Spring 而 Spring，为了 Hibernate 而 Hibernate，陷入所谓的高级特性中不能自拔。对于初学者来说，学会了一些框架的用法之后，急需提升的是内力，也就是核心 JDK 类的掌握。只需要一个原则：用框架如果不能减轻负担，加快开发进度，就不要用它！

当然作为一名实事求是的作者，我想我的话仅供大家参考，真正的决定权还是要取决于公司的领导和自己的实际情况了，学习过程最忌讳把某些人或者技术当成了神（作为曾经的初学者，我也走过这样的弯路），发言之前，还是自己先亲自试试才好。本书后续章节的内容，也会尽量保持相互之间的独立性，读者可以根据自己需要进行阅读，不需要您把每一章的内容都掌握了。