

第十四章 开发 JSF 应用

第十四章 开发JSF应用	1
14.1 前言	1
14.2 介绍	3
14.3 系统需求	3
14.4 快速入门	3
14.4.1 创建HelloJSF项目	4
14.4.2 创建消息包	6
14.4.3 创建受管Bean	7
14.4.4 创建JSP页面	12
14.4.5 添加导航规则	19
14.4.6 运行应用程序	22
14.5 事件监听和导航机制	23
14.5.1 位于Managed Bean中的事件处理方法	23
14.5.2 基于导航规则的命令按钮action	27
14.5.3 加入多个ActionListener类	29
14.6 JSF中的内置依赖注入	31
14.7 JSF整合Spring开发	33
14.7.1 简介	33
14.7.2 创建项目jsfspring并修改Java类	34
14.7.3 修改配置文件并在JSF中注入Bean	36
14.7.4 完全使用Spring配置Bean	40
14.8 JSF + JPA的MyEclipse 官方Blog实例	41
14.7 小结	45
14.8 参考资料	46

14.1 前言

在第十二章的时候，我们讨论了 Struts 2，也稍微提到了 JSF。我们都知道 Java 的 Web 开发日益面临着框架林立，开发工具笨拙，思路独特的限制。然而，Sun 公司的工程师们希望能把基于 Swing 的开发模式应用到 Web 中去，让这些可怜的 Java 程序员不用了解什么是 HTML 和 HTTP 也能工作。这很大程度上是因为微软的 .NET WebForm 给人留下的深刻印象，.NET 的程序员们不用像 Java 的 Web 程序员们那样，要学过一大堆的框架外加 HTML，CSS 和 JavaScript 后才能工作，他们只需要用可视化的 Visual Studio 中的页面设计器，拖放几个控件到页面上，设置下属性，编写以下事件处理代码，就搞定了。因此，Sun 希望在 Java 的世界里也能出现这样的技术和配套的 IDE，因此很早的时候，就开始了 JSF 的研究，而现在更是将它作为 Java EE 5 规范的一部分，所有要开发服务器的厂商都必须支持它。然而，好事多磨，因为 Java 世界的向心力总是不如微软那样强，大家各自打自己的算盘，

所以自从 2004 年 3 月第一个版本发布以来，很长一段时期内无人问津，更有很长时候成为高手们批判的对象。所幸风水轮流转，经过了一阵混乱的开源热风后，程序员已经尝到了非标准化和框架更新换代飞快，API 和开发模式互不兼容的苦头，所以现在 JSF 又重新回到了公众的视野，不过，现实是残酷的，因为不够成熟，它的支持率始终不是那么的高。原因有：配置文件复杂，基于 XML 格式；开发工具不够智能化；组件不够丰富。然而，它毕竟还是让人看到了统一化编程的方向，所以各大公司和开源组织不遗余力的探索和改进，目前新的规范正在制定当中，也就是第三代 Web 框架：基于标注的后台模式和基于组件的前台模式。

首先，JSF 是一组规范和接口，不包含具体的实现类。就其规范内容来说，因为要涵盖方方面面，所以里面的内容还是蛮复杂的。和 Sun 一贯的作风一样，它把 JSF 分成了两部分：面向开发工具制造商的部分（提供 JSF 的底层实现，组件库，以及配套开发工具）和面向开发人员的部分。虽然底层的实现各不相同，但是对于开发人员来说，标准的组件其用法都是一样的，这样就允许厂商在自己的 JSF 实现中在保持 API 不变的方式进行性能等的竞争。另一方面，不像其它的 Web 框架，JSF 首次提出组件库的概念，它可以让厂商和开发人员很容易的对 JSF 进行扩展，而使用者却不需要关心底层的实现，它只需要按照要求包含类库和标签库，就能工作了，这样也利于商业化。

在这里笔者不打算对 JSF 进行细致入微的介绍，我们只需要关注一下对开发人员带来的东西就可以了。首先是概念上的转变，像 Struts 2 一样，JSF 要求开发人员不再考虑 HTTP，Request，Response 这些东西，而把注意力转移到业务类和表示层上去。另外，第二个是服务器端事件驱动的编程模式。例如：<button onclick="alert('hello')">，点击按钮后会弹出一个对话框，然而现在这些功能可以转移到服务器端来进行了，更重要的是页面的每一个组件都可以这样做，这是 Struts（包括 Struts 2）里面所无法很轻松就能实现的，因为首先您要保存页面的当前状态，然后判断点击的是哪个按钮，之后在根据这个按钮再触发对应的处理方法，所幸的是这些功能 JSF 都已经做好了（或者说是要求 JSF 的框架开发人员必须实现这些）。读者不需要再去考虑表单提交之类的内容，它们已经被封装好了。第三个改变就是您可以使用可视化的设计器进行开发，就像.NET 的 Visual Studio 开发工具一样。像读者期待的那样，可以在页面里面拖放一张数据库表格，直接对上面的数据进行编辑等等，更重要的是这些功能都是做好的，无须自己从头开始。而且，JSF 目前也提供了大量支持 AJAX 的组件和标签库。

在厂商的支持上，JSF 目前获得了大多数 Java EE 服务器厂商的支持，例如 Oracle，BEA 等等，还有国产的金蝶软件的 Apusic 服务器；当然还有 Sun 自己的 GlassFish。开源上，则为数众多，例如 ICEfaces，JBoss RichFaces，Apache MyFaces 等等，最后，还有国产的金蝶开源的 Apusic OperaMasks。开发工具上，则有众多的选择（大部分都是免费的，并支持可视化的设计器），包括 Oracle JDeveloper，Bea Workshop，MyEclipse，Apusic Studio 以及目前来说做的开源里面最优秀的 Netbeans IDE 6。图 14.1 展示了 Netbeans 中正在进行可视化 JSF 开发的截屏。相比较而言，MyEclipse 的 JSF 开发功能则相对较弱，但是也足够我们使用的了。

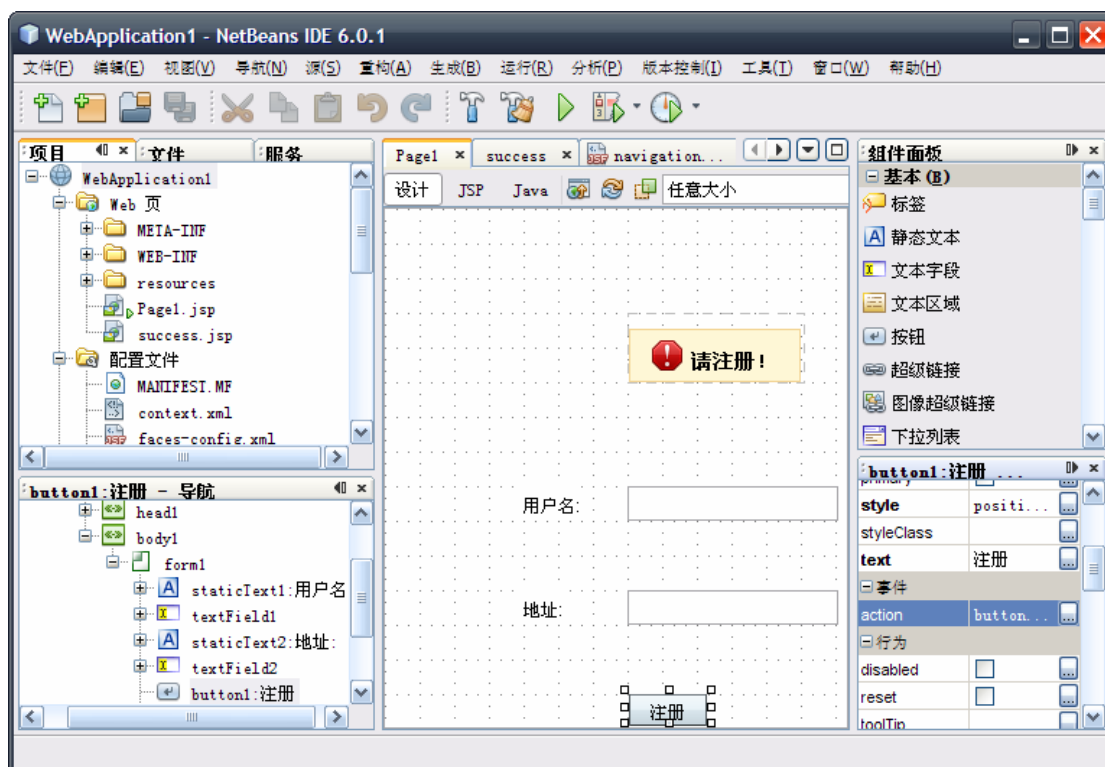


图 14.1 可视化开发 JSF 的 Netbeans 6

面对这么多的 JSF 框架，开发人员又该如何抉择呢？建议读者要学习 JSF 的话，尽量选择 Sun 公司的 JSF 参考实现，这样是比较规范和标准的。然而，这些 JSF 框架各具特色，例如有的尤其擅长 AJAX 功能，读者可以自己根据实际情况进行抉择。

14.2 介绍

由于 JSF 框架的目前市场占有率并不是特别的高，因此在本章呢，也不打算特别深入和全面的对其进行介绍。读者如有兴趣可以查看本章末尾的参考资料一章来获取更多信息。我们主要来做一个非常简单的登录应用，来展示基本的开发过程。读者很快就可以发现后台类的开发和 Struts 2 有很相似的地方。实际上，JSF 和我们以前介绍的 Struts 1 和 Struts 2 都有很多相似的概念。当然，读者可以完全在没有接触过 Struts 的情况下学习 JSF。

14.3 系统需求

我们的第一个练习只需要 MyEclipse 6.0 + Tomcat 就可以了，或者其它的 Web 服务器也可以。

后面的部分需要 MyEclipse Derby 数据库。

14.4 快速入门

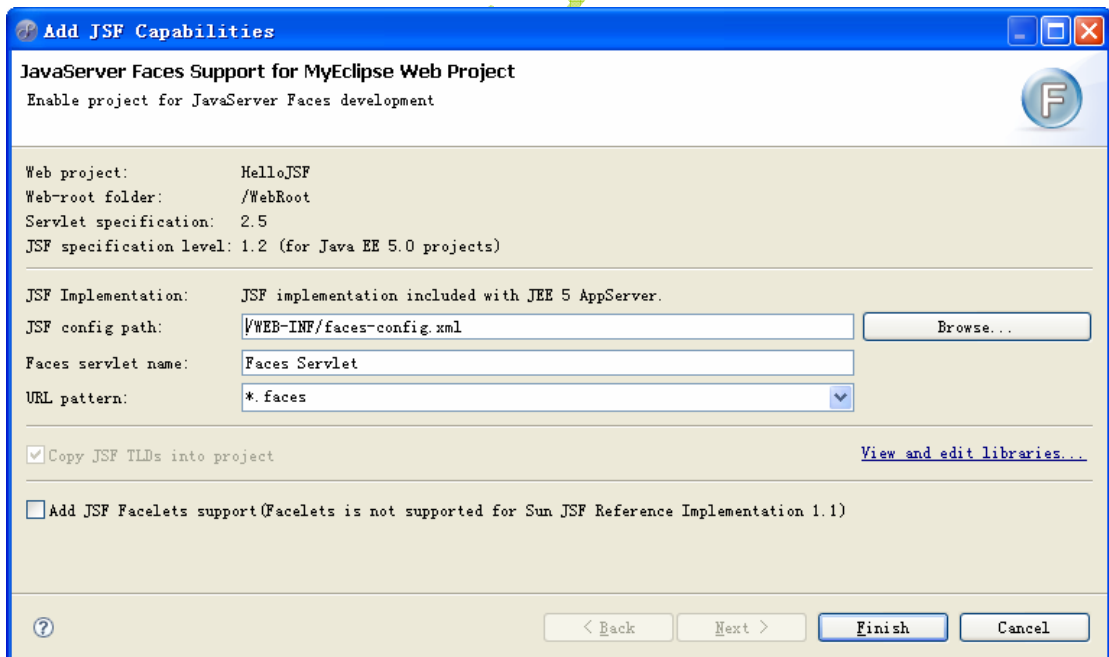
本节我们将会做一个非常简易的登录示例应用。

14.4.1 创建 HelloJSF 项目

我们需要在 MyEclipse 中创建一个新的 Web 项目，并向它添加必要的 Struts 2 类库和配置文件。现在我们来创建一个名为 *HelloJSF* 的 Web 项目。选择菜单 **File > New > Web Project**，可以启动创建 Web 项目的向导，如图 8.3 所示。在弹出的对话框的 **Project Name** 中输入 *HelloJSF*，然后选中 **J2EE Specification Level** 下面的 **Java EE 5.0** 单选钮，最后点击 **Finish** 按钮就可以创建好这个 Web 项目了。

注意：因为开发 JSF 程序的时候流行和 JSTL 类库搭配，所以在项目创建时如果您选中的是 J2EE 1.4，建议您选中复选框 **Add JSTL libraries to WEB-INF/lib folder?**，并选中 **JSTL 1.1** 单选钮。当然，您也可以在以后再来添加 JSTL 类库，通过选择菜单项 **MyEclipse > Project Capabilities > Add JSTL Libraries...**来完成。

Web 项目创建完毕后，我们需要给它添加 JSF 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，选择弹出菜单中的 **MyEclipse > Add JSF Capabilities**；也可以通过选择菜单 **MyEclipse > Project Capabilities > Add JSF Capabilities ...** 来启动 **Add JSF Capabilities** 向导，如图 14.2 所示。JSF 对话框的默认值一般来说不需要修改就可以使用，不过您熟悉的话也可以通过修改对话框里面的默认值来改变配置。图中默认的 JSF 实现是 Java EE 5.0 Sun 公司的版本，不支持 Facelets。对话框中的 **JSF config path** 指定了 JSF 配置文件的位置，而 **Faces servlet name** 则指定了 JSF 的核心 Servlet 的名字，**URL pattern** 制定了 JSF Servlet 的默认监听的 URL 类型的名字为 **.faces*，这些值都可以加以修改。如果选中了复选框 **Add JSF Facelets support**，那么就可以点击 **Next** 按钮进入下一页设置 Facelets。



14.2 添加 JSF 功能向导对话框

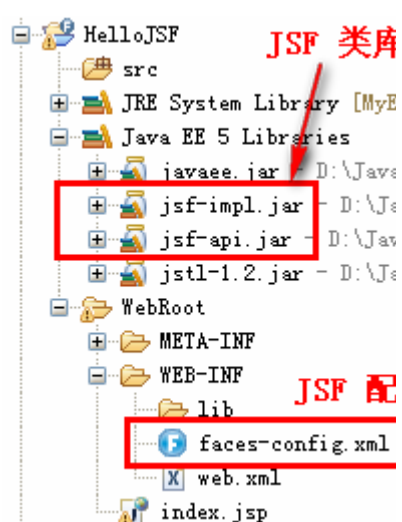


图 14.3 添加完 JSF 功能的项目目录结构

当向导结束后,可以看到项目的目录结构将如图 14.3 所示。其中的两个 jar 是 JSF 的类库,读者完全可以替换成其它类型的 JSF 实现,例如 MyFaces,而不必修改 web.xml 中的 Servlet 定义。faces-config.xml 则是 JSF 的核心配置文件,这和 Struts 等其它框架是相似的,都有核心的配置文件。

在这时候,空白的 faces-config.xml 的内容如下:

```
<?xml version='1.0' encoding='UTF-8'?>
<faces-config
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  </faces-config>
```

。web.xml 的代码清单如下所示:

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="2.5"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml</param-value>
  </context-param>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>
  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>
</web-app>
```

。javax.faces.CONFIG_FILES 这个变量的取值指定了 JSF 的配置文件的位置所在,读者可以根据自己的需要进行修改。

14.4.2 创建消息包

JSF 支持国际化的消息文件，因此在项目里面可以使用消息包来显示多国语言。现在让我们来创建一个 **MessageBundle** 文件，这是一个简单的属性文件，它将保存所有的消息字符串以及相关的主键，然后这个消息包可以用在我们的任何 **JSP** 文件中来让我们的应用很容易的支持国际化语言。就像过去看到的那样，**Struts1** 和 **2** 在这个领域提供了相似的支持，通过使用 **ApplicationResources.properties** 文件或者 **package.properties** 文件，以及不同的 **<bean:message />** 标签，或者 **<s:text>** 标签，就可以在页面中根据消息包来显示国际化的提示信息。在 **JSF** 页面里面，我们可以在 **JSP** 页面里加入一行代码来加载消息包：

```
<f:loadBundle basename="com.jsfdemo.MessageBundle" var="bundle"/>
```

注意：这一行代码创建了一个生命周期为页面的消息包，这个包可以稍后在页面中通过变量名 **'bundle'** 来进行引用，可以用来查找消息主键，然后返回对应的消息值。另外，**JSF** 可以完全不用消息驱动包也能进行开发，所以读者不要误会，这一节的内容和后面的一些内容是可选的。

要创建消息包文件，我们可以使用新建文件向导，在 **Package Explorer** 视图中右键点击项目的 **src** 目录，在弹出的菜单中选择 **New > File**，在文件新建对话框的 **File name** 中输入 **Messages.properties**，然后将文件内容修改为如下所示：

```
login_label=Please Login:
username_label=User name:
```

。同样的我们还需要一份中文的消息文件内容，具体制作方法可以参考 [12.3.3 使用国际化消息](#) 一节的内容。这份文件名为 **Messages_zh_CN.properties**，文件内容如下：

```
login_label=\u8bf7\u767b\u5f55:
username_label=\u7528\u6237\u540d:
```

。这两行转码过的文字内容分别是：**请登录:**和**用户名:**。主键将来会用在 **JSF** 页面中显示消息之用。

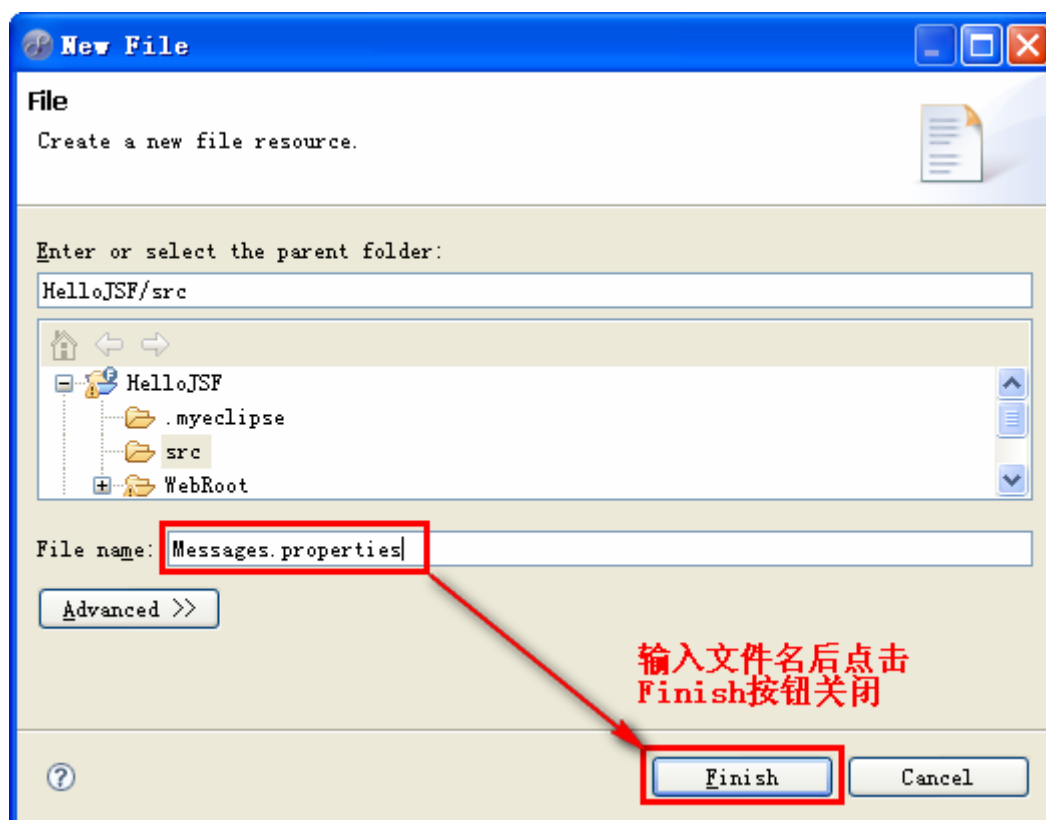


图 14.4 创建国际化消息资源文件

现在我们的 `MessageBundle` 创建完毕了，接下来我们要创建受管 Bean(`ManagedBean`)，它将处理我们的用户登录操作。很快你就会发现 `ManagedBean` 和 Struts 2 中的 `Action` 类很相似。

14.4.3 创建受管 Bean

在这一部分我们将会介绍如何创建 `ManagedBean`，这个 Bean 将会执行登录 JSP 页面所提示的登录操作，以及从页面表单中提取参数，并保存用户输入的用户名和密码到 Bean 的属性中（类似于 Struts 2 的 `Action` 所做的那样）。出于演示的目的，我们的登录操作只是简单的检查用户名和密码是否都为 "myeclipse"，然后将用户重新指引到登录成功的 `userLoginSuccess.jsp` 页面，如果登录失败就返回到输入密码页面。


首先使用 *MyEclipse JSF Cofnig Editor* 打开文件 `faces-config.xml`，用鼠标双击此文件即可打开。在工具箱(Palette)上，我们可以点击工具栏按钮创建各种元素，包括创建和管理应用程序流程（即导航规则 `Navigation Case`）。而在画布上点击右键，则可以看到快捷菜单，包括：

- ◆ Undo Text Change 撤销文本修改
- ◆ Redo Text Change 重做文本修改
- ◆ Add NavigationRule 添加导航规则
- ◆ Export Image 导出为图片

。点击底部的标签 **Design/Source** 则可以在设计视图和源代码视图（源代码是 XML 格式的）之间切换。



图 14.5 JSF 配置文件编辑器及其快捷菜单

除此之外，此时的大纲视图可以显示当前文件的内容结构，还可以创建几乎所有的 JSF 元素，点中大纲上的节点的话也会选中对应设计面板上的元素或者是定位到对应的 XML 源代码。从 **Outline** 视图上，你可以右键点击选择对应的快捷菜单，然后来激活向导创建对应类型的组件，或者来使用向导编辑存在的组件，如图 14.6 左侧图所示；另一种方式就是点击 **Outline** 视图的  按钮则弹出相关的创建 JSF 组件的菜单，点击对应的菜单项就可以启动相应的对话框来创建所需要的元素，如图 14.6 右侧图所示。

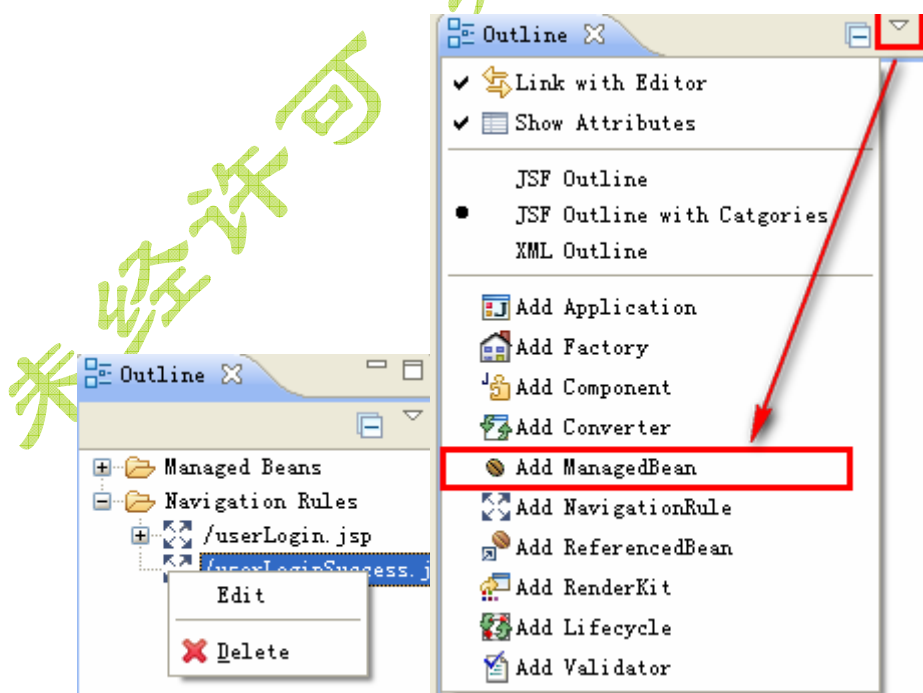


图 14.6 Outline 视图以及启动创建 ManagedBean 向导

当我们按照图 14.6 中右图步骤操作时，即可启动新建 Managed Bean 向导对话框，此

对话框如图 14.7 所示。我们按照图中进行设置即可。**Name** 处输入 *UserBean*。**Class** 输入 *com.jsfdemo.UserBean*。**Scope** 设置这个 Bean 的作用域，取值包括 *request*、*session*、*application* 和 *none*（无），这里选中 *session*。对话框的最下方则需要选中复选框 *Generate Java code*（生成 Java 代码）和 *Generate missing getters/setters for properties*（生成缺少的属性所对应的 *getter* 和 *setter* 方法），这样我们才能在中间的 **Properties** 标签下的属性列表中添加、删除和修改 Bean 的属性，最后还能生成对应的代码。

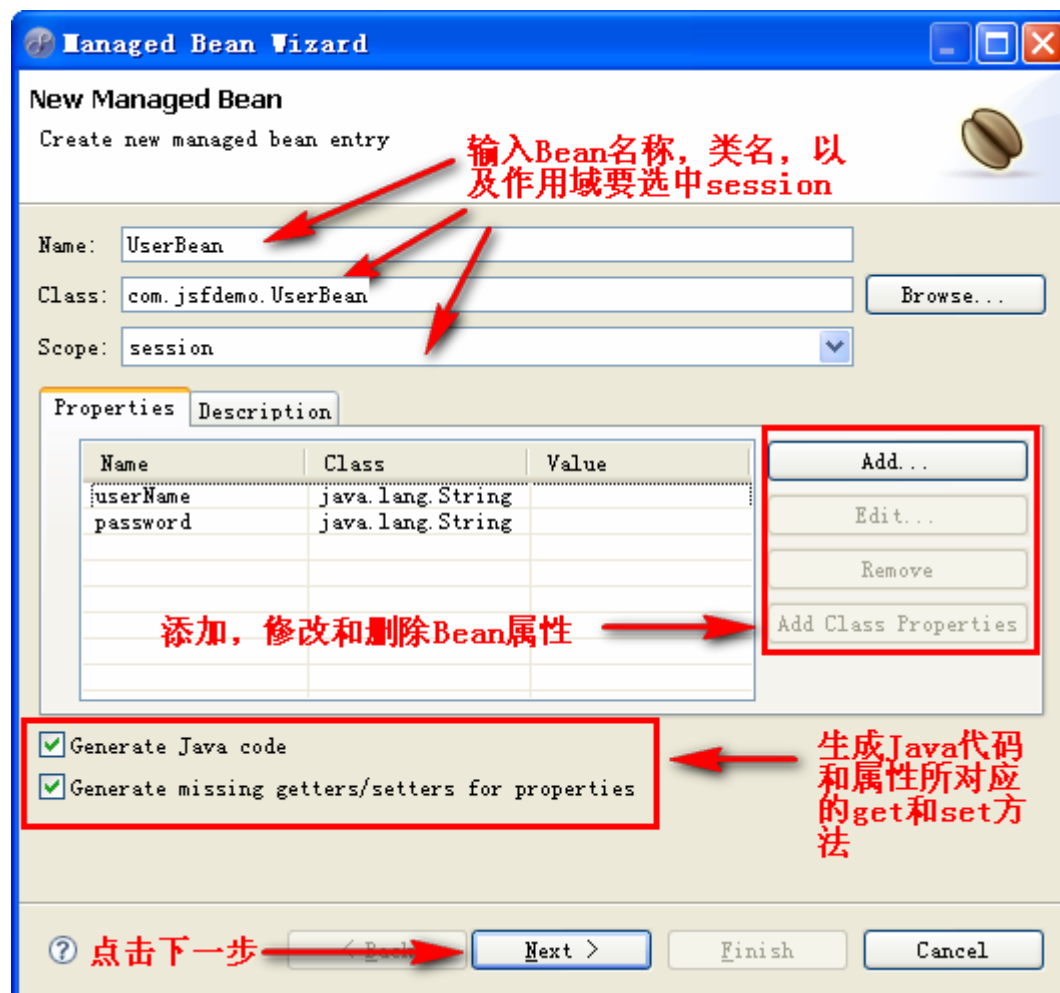


图 14.7 设置新 ManagedBean 的类和属性

点击 **Add...** 按钮后将会弹出添加属性的对话框，一共有两页，如图 14.8 所示。第一页可以设置 **Name**（属性名），**Property Kind**（属性种类，支持简单类型，以及 Map 和列表），以及 **Class**（属性的完整类名，包括 *int*、*java.lang.String* 等等，注意不是任意类型都可以的，一般来说只能用简单的类型，或者有转换器—*Converter* 的类型，限于篇幅就不多做介绍了），**Description** 可以输入类型的描述信息。第二页的 **Property Value** 则说明了 JSF 具有简单的属性值注入能力了，可以设置初始化时候的取值，默认情况下是 *null*。点击 **Finish** 按钮关闭对话框即可完成属性的添加。

在第一页设置完毕后，我们还必须点击图 14.7 中的 **Next** 按钮进行第二页的设置，这一页相对非常简单，就是设置最终代码所生成的位置，默认情况下点击 **Finish** 按钮就可以了，读者也可以根据情况进行一些必要的调整。现在，整个向导结束了，稍等片刻后，即会创建我们需要的 *JavaBean* 以及会自动修改相关的配置文件。

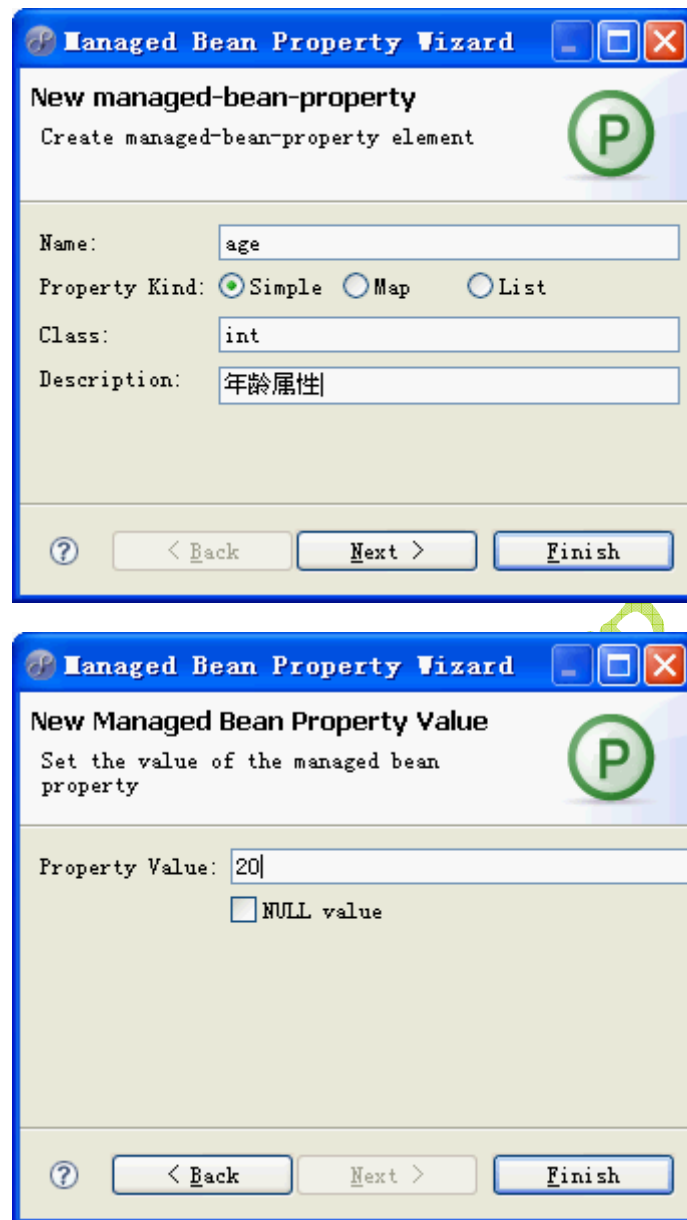


图 14.8 添加属性的对话框

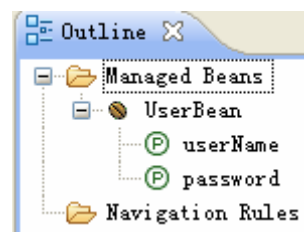


图 14.9 大纲视图中的 UserBean

现在我们可以看到 UserBean 出现在 JSF 配置文件编辑器的大纲视图中，同时呢，新建的 **UserBean.java** 源文件也在 Java 编辑器中打开了，其源代码清单如下所示：

```
package com.jsfdemo;
```

```
// 类似于Action类的用户受管 Bean
public final class UserBean extends Object {

    /**
     * 进行登录操作.
     * @return 导航规则地址
     */
    public String login() {
        if(getUserName().equals("myeclipse") && getPassword().equals("myeclipse"))
        {
            return "success";
        }

        return "failure";
    }

    private String userName;
    private String password;

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

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

当然，和以前的习惯一样，那个粗斜体的内容，是我们经过修改后手工加入的代码，用来处理登录业务逻辑。和 Struts 2 一样，JSF 也支持 POJO 的普通 Java 类模式的开发，所以一个受管 Bean 中除了属性之外，还可以加入任意多个业务功能定义，例如：


```
public String doLogin();
```

只要其返回值为字符串即可，这个字符串对应的是一个浏览规则的别名，类似于原来 Struts 中所说的 ActionForward。UserBean 类并没有继承或者实现绑定到 JSF 的任何类或者接

口，它仅仅是一个简单的包含了额外的逻辑来执行有用的操作的 **JavaBean**，用 **Struts** 的术语来说，它包含了 **Struts Form** 和 **Struts Action** 的所有功能，方便的合并在一个类中；或者说它就是 **Struts 2** 中的 **Action** 类。另一个和 **Struts** 的不同之处是这些方法没有返回任何特殊的类，例如 **ActionForward**，因为导航信息是在 **faces-config.xml** 部署描述符中通过配置完成的。我们将在后面章节展示如何创建并配置导航规则。

14.4.4 创建 JSP 页面

在这一小节里面将集中精力为我们的示例 **JSF** 应用创建 **JSP** 页面，来模拟一个简单的网站常见的登录页面。最后的结果只需要 2 个 **JSP** 页面，一个用来提示用户来输入登录用户和密码，另一个用来告诉用户登录成功。我们把这两个页面分别命名为 **userLogin.jsp** 和 **userLoginSuccess.jsp**。为了简化过程，如果用户尝试登录的时候验证身份失败，就将用户重定向回页面 **userLogin.jsp**。为了避免引起混淆，我们没有在这个例子里使用任何 **JSF** 自带的验证机制，不过你可以很容易的为 **JSF** 的 **inputText/Secret** 组件添加验证器 (**Validator**)，可以使用这些输入框来验证用户输入的值的长度并且可以在登录失败时给用户显示错误信息。

要创建我们的 **userLogin.jsp** 页面，首先需要打开 **faces-config.xml**，在设计器的工具箱中点击一下 **JSP** 按钮 ，然后再点击到画布上，或者拖住按钮然后放到画布上。当新建 **JSP** 页面向导对话框弹出时，根据图 14.10 的提示在 **File Name** (文件名) 输入框中输入值 **userLogin.jsp** 并选择创建 **JSP** 页面的模板为 **Default JSF template**。同样再重复一次这个步骤，创建 **JSP** 文件 **userLoginSuccess.jsp** (注意一定要选中 **JSF** 模板)。随后就可以看到这两个 **JSP** 页面作为图标形式显示在了画布上。

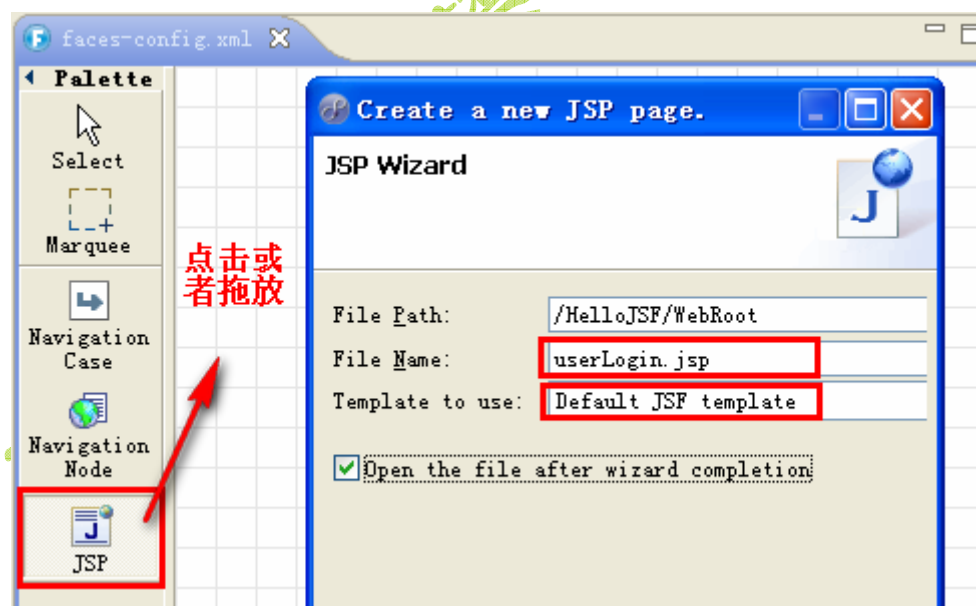


图 14.10 使用 **faces-config.xml** 编辑器创建 **userLogin.jsp**

现在我们可以看看 **faces-config.xml** 的源代码清单，已经包含了上面我们所创建的 **UserBean** 以及 **JSP** 页面的定义：

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

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
```

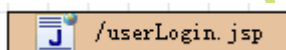
```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
version="1.2">

<managed-bean>
  <managed-bean-name>UserBean</managed-bean-name>
  <managed-bean-class>com.jsfdemo.UserBean</managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
  <managed-property>
    <property-name>userName</property-name>
    <property-class>java.lang.String</property-class>
    <value></value>
  </managed-property>
  <managed-property>
    <property-name>password</property-name>
    <property-class>java.lang.String</property-class>
    <value></value>
  </managed-property>
</managed-bean>
<navigation-rule>
  <from-view-id>/userLogin.jsp</from-view-id>
</navigation-rule>
<navigation-rule>
  <from-view-id>/userLoginSuccess.jsp</from-view-id>
</navigation-rule></faces-config>

```

。对应我们上面所介绍的开发过程，大家可以很容易的辨认出这些标记的含义。
接着我们可以打开页面 `userLogin.jsp` 来给我们的应用加入功能，可以在 JSF 画布上双击



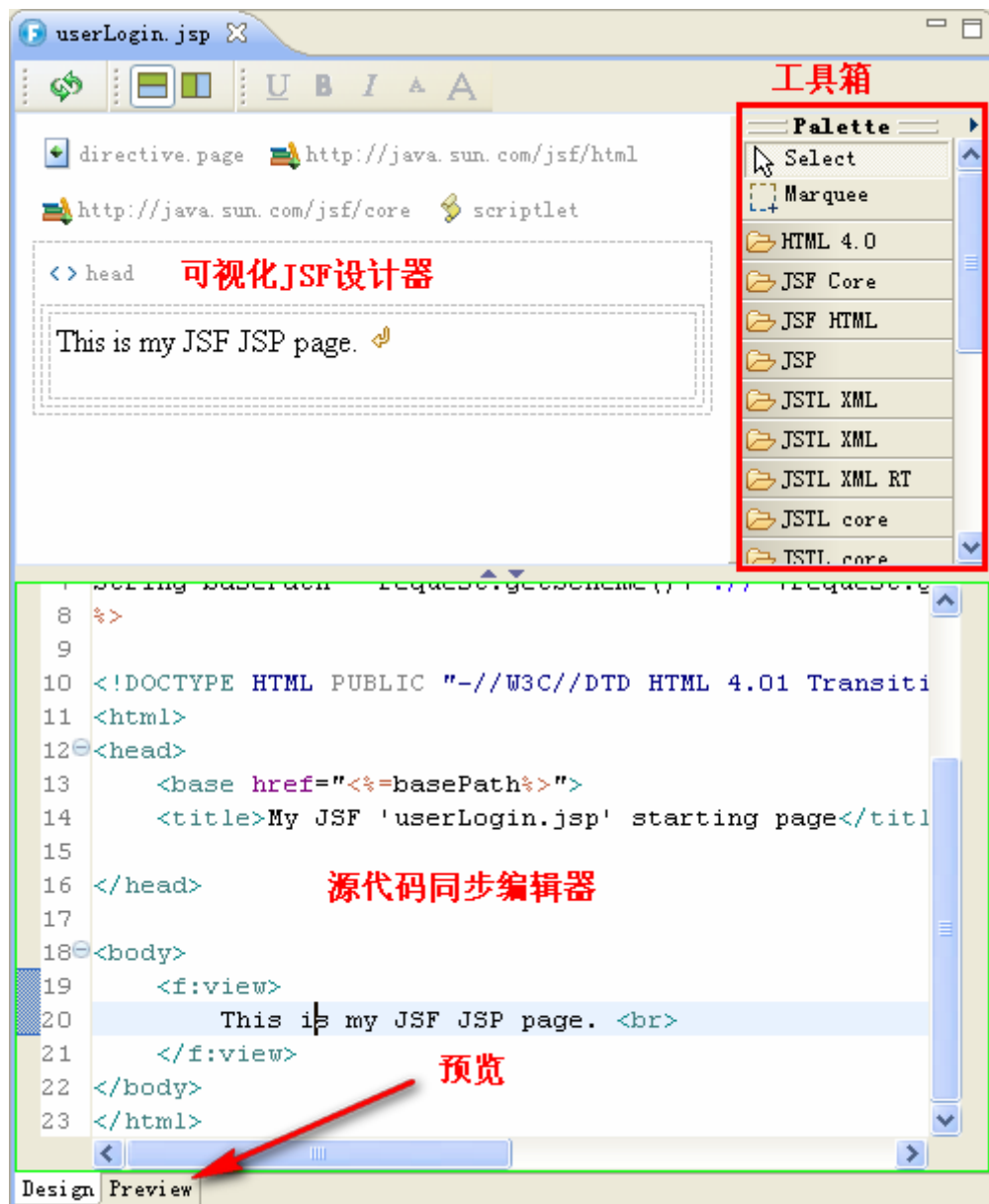
图标来打开并编辑这个页面，也可以在 **Package Explorer** 视图的 **WebRoot** 目录下找到此文件然后双击打开。此时将会用可视化的 JSF 编辑器打开页面，如图 14.11 所示。在 **Palette**（工具箱）上，我们可以展开各个目录，看到对应的组件（或者说控件，页面元素，自定义标签等），点击后即可放到页面，然后即可添加到页面并可以在 **Properties** 视图中设置所需的属性（有的组件添加时会弹出对话框进行必要的设置）。和 JSF 开发有关的两个目录是 **JSF Core** 和 **JSF HTML**。注意刚开始的时候 **Palette** 是折叠的，点击向左的小箭头展开它。

现在，我们需要对这个页面进行如下修改：

- 修改页面的编码为 GBK
- 加入国际化消息包调用 `f:loadBundle`（需要说明的是消息驱动包是可选的）
- 添加一个 `h:outputText` 来输出消息包中的国际化欢迎信息 `login_label`
- 添加一个表单 `h:form`
- 给 `username` 属性添加一个 `h:inputText` 组件（文本框）
- 给 `password` 属性添加一个 `h:inputSecret` 组件（密码框）
- 给 `username` 的 `inputText` 添加一个 `h:outputLabel` 来输出消息包中的国际化欢迎

信息 username_label

- 添加登录的命令按钮 h:commandButton



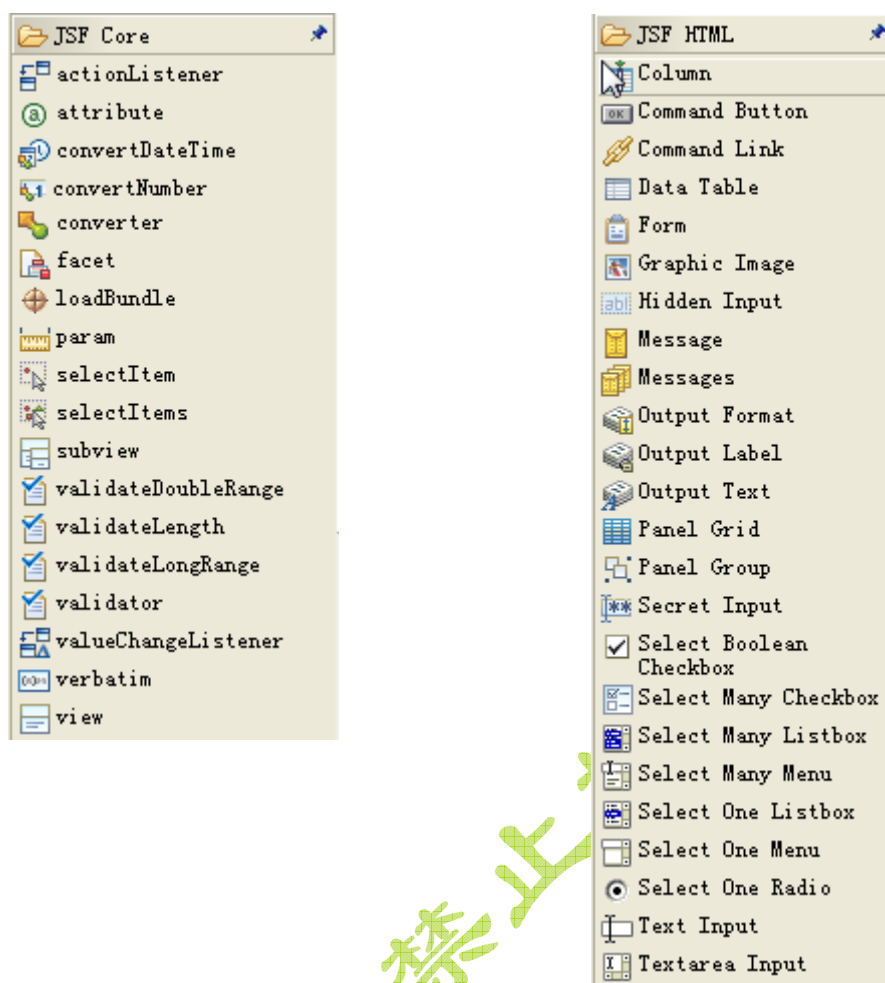


图 14.11 可视化的 JSF 编辑器及工具箱 JSF 组件列表

好了，现在让我们依次来完成，首先把页面第一行的编码修改为中文：
`<%@ page language="java" pageEncoding="GBK"%>`。

接着我们将页面中的 *This is my JSF JSP page* 这句生成的代码删除掉。现在请单击展开 JSF Core，然后选择里面的 **loadBundle** 这个按钮，点击一下然后再点到页面上，或者点中不放拖到页面设计视图中松开鼠标，即可将加载消息资源的 JSF 标签加入到页面中，初始的代码如下所示：

```
<f:loadBundle />
```

。不过这样是不完整的，此时可以在 **Properties** 视图中快速编辑其属性，如图 14.12 所示。



图 14.12 编辑加载消息包的标记

Var 定义了这个消息包的别名，可以在页面的后面进行引用，**Basename** 则是在第 14.4.2

节中创建的消息包的名字（不包含扩展名.properties），所以取值为 *Messages*（如果放在某个包下面，对应的则应该是 *com.xxxx.Messages*），此时完整的代码如下所示：

```
<f:loadBundle basename="Messages" var="bundle"/>
```


当然，我们也可以直接将这段代码敲入页面源码中，效果是一样的。在代码编辑区键入的时候，先敲入小于号 <，稍等片刻就能够看到 MyEclipse 自动弹出的标记的完成提示，选择一个后双击提示或者按下回车键，就能输入完整的标记，这样便于我们快速编写，如图 14.13 所示。




图 14.13 JSF 编辑器的代码自动完成提示

接下来要做的，是从刚才加载的国际化资源文件中，输出登录提示的信息，加入下列代码即可：


```
<h:outputText value="#{bundle.login_label}"/>
```

在输入这段代码时，当我们键入 *bundle.*后，稍等片刻，它就能列出对应的属性文件中的键列表，进行自动完成的提示，总之 JSF 编辑器在多方面都提供了完成提示，读者可以慢慢体会。这段代码将会输出一段国际化的文字，从 *bundle* 中读取属性为 *login_label* 的主键所对应的值，最终它会读取 *Messages.properties* 文件中（如果是中文环境则对读取中文的 *Messages_zh_CN.properties* 中的内容），并在运行时显示出来（最后的值是 *Please Login*：或者中文的 *请登录*：）。*value* 以 *#{}* 括起来表示这是一段动态的需要执行的属性代码，相当于 EL 表达式，类似于调用 Java 代码：*bundle.get("login_Label")*；如果是 *JavaBean*，那么这样的值会对应一个方法 *bundle.getLogin_label()*。在后面我们会看到表单输入框的取值将会采用这样的方式和后台的 *Managed Bean* 中的属性值相对应，有个属于叫做 *bind*（绑定）。如果写作 *value="请登录"*，那么这样是可以直接输出值的。如果不想记忆这段代码的话，可以在 *Patette* 的 *JSF HTML* 一栏下可以找到对应的工具栏按钮： *Output Text*，点击它可以获得同样的一段 JSP 标签。

随后需要加入的是表单，点击在 *Patette* 的 *JSF HTML* 一栏下的  即可加入表单标记：*<h:form></h:form>*。

接着我们将光标放在表单中，然后给用户名添加一个文本输入框，点击点击在 *Patette* 的 *JSF HTML* 一栏下的  *Text Input* 即可加入此标签，可以通过在 *Properties* 视图的 *Quick Edit* 标签中编辑其属性。在 *ID*: 右侧的输入框中输入 *userName*，在 *Value*: 右侧输入框中输入值 *#{UserBean.userName}*，如前所述，这将把输入框的值和后台的受管 *Bean*: *UserBean* 的 *userName* 属性关联起来。当然，我们也可以直接键入标签，最后得到的代码如下所示：

```
<h:inputText id="userName" value="#{UserBean.userName}"/></h:inputText>
```

接下来要加入的是给 *password* 属性添加一个密码输入框，点击点击在 *Patette* 的 *JSF HTML* 一栏下的  *Secret Input* 即可加入此标签，可以通过在 *Properties* 视图的 *Quick Edit* 标签中编辑其属性。在 *ID*: 右侧的输入框中输入 *password*，在 *Value*: 右侧输入框中输入值 *#{UserBean.password}*，如前所述，这将把输入框的值和后台的受管 *Bean*: *UserBean* 的 *password* 属性关联起来。当然，我们也可以直接键入标签，最后得到的代码如下所示：

```
<h:inputSecret id="password" value="#{UserBean.password}"/>
```

接着我们在登录输入框前加入一点提示标签，同样的点击在 **Patette** 的 **JSF HTML** 一栏下的 **Output Label**，具体的设置就是每个 **Label** 要对应一个输入框，如下所示：

```
<h:outputLabel for="userName" value="#{bundle.username_label}"></h:outputLabel>
```

。标签的取值是从资源文件里读取的国际化信息，当然也可以设置静态值，**for** 指定了这个标签所对应的控件的 **id**。其实这是和 **HTML** 里面的 **label** 标签的用法是很相似的。

最后一步，就是给我们的登录表单加入一个命令按钮了。**JSF** 中的命令按钮功能很强大，可以执行受管 **Bean** 中的任意的不带参数的返回值为字符串的方法，还可以加入事件监听器（一个组件可能产生多种事件，例如树），把页面中的组件作为一个 **Java** 对象来使用。我们先来看最简单的设置。将光标放在 **</h:form>** 的前面，点击在 **Patette** 的 **JSF HTML** 一栏下的 **Command Button**，即可在页面中加入一个命令按钮。此时在属性编辑器中将会显示可以设置的属性，我们按照图 14.14 来进行设置即可。

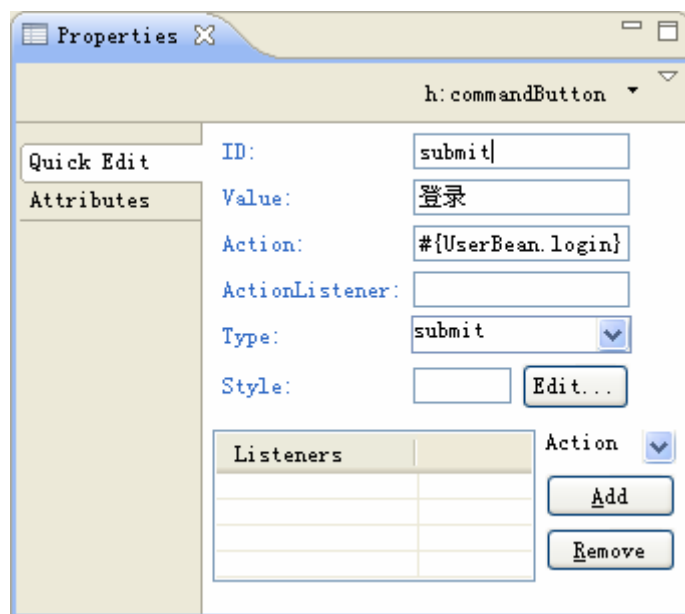


图 14.14 JSF 的命令按钮属性

关于这些设置的相关说明，我们简单加以介绍，如下表所示：

名称	取值	说明
ID	submit	按钮的标识（名字）
Value	登录	显示的文字（可以动态和静态）
Action	#{UserBean.login}	点击按钮后执行的动作，对应着一个返回值为字符串的方法:public String login(); 也可以是一个导航规则的名字
ActionListener	示例:#{ubean.doAct}	动作监听器，需要编写一个事件监听方法
Type	submit	按钮的类型，包括 submit 和 reset
Style	示例:color:red;left:10px	组件的 CSS 风格
Listeners	一个或者多个类	当前按钮的多个单独的事件处理监听器

。最重要的属性当属 **Action** 了，它有两种取值，一种是对应 **Action** 类中的某个处理方法，而另一种则是直接对应某个导航规则，这时候它的取值是纯字符串。在后面我们将会开发一个新的例子，给大家展示事件监听和另一种直接导航的处理方法。现在我们可以看到对应的代码如下：

```
<h:commandButton id="submit" type="submit" action="#{UserBean.login}" value="登录"
```

"></h:commandButton>

。实际上，在一个页面中可以加入多个 `commandButton`，然后在 `UserBean` 中写入多个业务处理方法，例如再加入注册（`public String register()`），那么让新加入的按钮的 **Action** 取值为 `#{UserBean.register}` 就可以点击时调用此方法了。

现在看看页面，大部分的功能已经完工，不过界面布局却是很乱，让人不敢恭维……我们可以添加一些空格和换行（`
`），也可以用快捷键 **Shift+Enter** 直接输入 `
`，或者按下回车键输入 `<p>`，或者实用表格或 `DIV` 来进行布局和页面美化。最后经过调整后的 `userLogin.jsp` 页面代码如下所示：

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<%
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>请登录</title>
</head>

<body>
    <f:view>
        <f:loadBundle basename="Messages" var="bundle"/>
        <h:outputText value="#{bundle.login_label}"/>
        <h:form id="loginForm">
            <h:outputLabel for="userName"
value="#{bundle.username_label}"></h:outputLabel>
            <h:inputText id="userName"
value="#{UserBean.userName}"></h:inputText>
            <br>密码: <h:inputSecret id="password"
value="#{UserBean.password}"></h:inputSecret>
            <br><h:commandButton id="submit" type="submit"
action="#{UserBean.login}" value="登录"></h:commandButton>
        </h:form>

    </f:view>
</body>
```

```
</html>
```

。接下来，打开 **userLoginSuccess.jsp**，修改内容，让它从 **session** 中获取用户登录的信息，显示登录用户所输入的名字，代码清单如下所示：

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<%
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>登录成功</title>

</head>

<body>
    <f:view>
        你好 <h:outputText
value="#{UserBean.userName}"></h:outputText>, 您已经成功登录!
    </f:view>
</body>
</html>
```

。标签 **h:outputText** 用来输出文字，取值可以是动态的（以 **#{}** 包围），也可以是静态的字符串。

到现在为止，所有的 JSP 页面都已经开发完毕了。

14.4.5 添加导航规则

类似于 Struts 中的 **ActionForward**，以及 Struts 2 中的 **Result**，我们需要给一些页面的跳转流程以别名的方式定义，这就是 **Navigation Case**（导航规则），例如我们可以给跳转到 **userLoginSuccess.jsp** 这个页面的过程给一个别名为 **success**，以后在应用中引用它就可以了。现在页面已经创建完毕，我们要做的就是把他们用正确的导航规则(Navigation Case)连接到一块，这个可以通过可视化的修改 **faces-config.xml** 文件来完成，所以先要打开这个文件。文件打开后，通过下列步骤来创建导航规则：

1. 点击 **Navigation Case** 工具按钮；

2. 点击 `userLogin.jsp` 文件；
3. 然后点击到 `userLoginSuccess.jsp` 文件；
4. 这时将会出现一个向导来提示你创建导航规则。

操作步骤如图 14.15 所示。

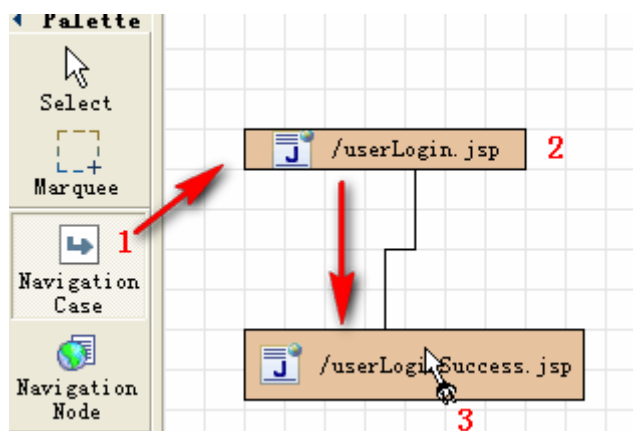


图 14.15 创建导航规则的鼠标操作

接着会弹出创建导航规则的对话框，如图 14.16 所示。

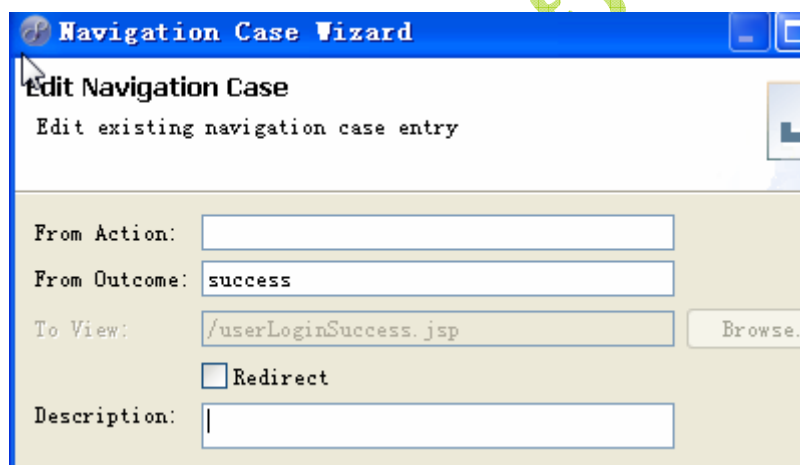


图 14.16 创建导航规则的对话框

现在，在 **From Outcome** 中输入 `success`，点击 **Finish** 按钮即可完成创建成功(success) 浏览的导航规则，在 **Description** 中可以输入描述信息：登录成功。如果希望这个导航规则进行重定向的操作，可以选中对话框中的 **Redirect** 复选框。

接下来用类似的过程创建失败(failure) 导航规则，我们简单的重复如上的步骤，只不过是点击两次文件 `userLogin.jsp`，来创建一个循环的导航规则(自己导航到自己)，然后在 **From Outcome** 中输入 `failure`，描述信息我们输入登录失败。当这两个导航规则创建完毕后，可以看到画布显示如下图所示的流程：

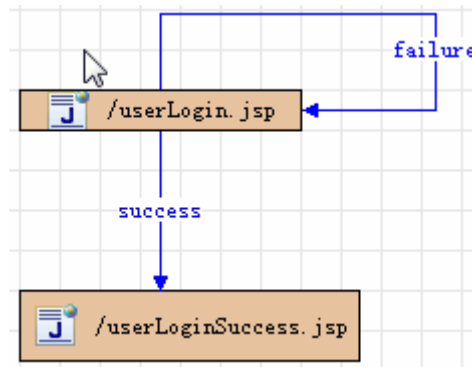


图 14.17 导航规则流程图

。此时对应的配置文件 *faces-config.xml* 源代码清单如下所示：

```

<?xml version='1.0' encoding='UTF-8'?>

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">

  <managed-bean>
    <managed-bean-name>UserBean</managed-bean-name>
    <managed-bean-class>com.jsfdemo.UserBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>userName</property-name>
      <property-class>java.lang.String</property-class>
      <value></value>
    </managed-property>
    <managed-property>
      <property-name>password</property-name>
      <property-class>java.lang.String</property-class>
      <value></value>
    </managed-property>
  </managed-bean>
  <navigation-rule>
    <from-view-id>/userLogin.jsp</from-view-id>
    <navigation-case>
      <description>登录成功</description>
      <from-outcome>success</from-outcome>
      <to-view-id>/userLoginSuccess.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <description>登录失败</description>

```

```

        <from-outcome>failure</from-outcome>
        <to-view-id>/userLogin.jsp</to-view-id>
    </navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>/userLoginSuccess.jsp</from-view-id>
</navigation-rule>
</faces-config>

```

。到现在为止，应用的全部内容都已经开发完毕了。

14.4.6 运行应用程序

现在，我们可以将这个应用发布到服务器上，或者在 **Package Explorer** 视图中选中项目节点 **HelloJSF**，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/HelloJSF/userLogin.faces>，来打开登录页面，如图 14.18 所示。

注意：URL 以 **.faces** 结尾而不是 **.jsp** 的原因是因为在上面，我们将 **FacesServlet** 映射到了 ***.faces** 扩展名，这意味着为了能使我们的 **JSF** 获得机会来处理请求并且构造组件树，我们必须使用 **.faces** 扩展名来访问真正的页面。如果你不这样做，你将获得一个异常信息，包含下列信息：**"FacesContext cannot be found"**。

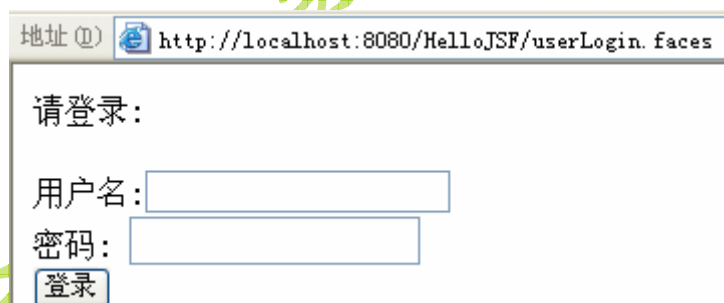


图 14.18 登录应用首页

现在可以尝试输入错误的用户名和密码进行测试，只有当两个值都输入了 **myeclipse** 时，才会显示登录成功。读者可以尝试把用户名的验证改成中文，可以发现依然是可以运行成功的。还有读者可能希望直到如果业务逻辑中返回值为 **null** 会出现什么情况，答案是仍然会返回先前的表单提交页面。读者可以把 **UserBean** 中的登录方法修改为如下内容然后进行测试：

```

/**
 * 进行登录操作.
 * @return 导航规则地址
 */
public String login() {
    if(getUserName().equals("张三") &&

```

```

getPassword().equals("myeclipse")) {
    return "success";
}

return null;
}

```

。那么我们一定要了解执行流程：JSF 容器启动并监听路径*.faces → 创建 UserBean 的示例并存入 session → 从 userLogin.jsp 生成响应 userLogin.faces → 提交 → 将表单参数封装到 UserBean 的属性中 → 执行 login() 方法 → 获取 login() 的方法返回值：导航路径 → 根据导航路径跳转到结果页面 userLoginSuccess.faces → 根据 userLoginSuccess.jsp 生成响应。这个过程和 Struts 2 的流程是差不多的。

14.5 事件监听和导航机制

14.5.1 位于 Managed Bean 中的事件处理方法

JSF 的一大特色就是将 JSP 页面中的组件和服务器对应起来，这样每个组件都可以加入自己的服务器端的事件监听机制。整个页面中的 JSF 组件是一个有机的整体，可以通过方法在服务器端来设置其属性。关于此，大家可以查看图 14.19 的 JSF 生命周期进行了解。

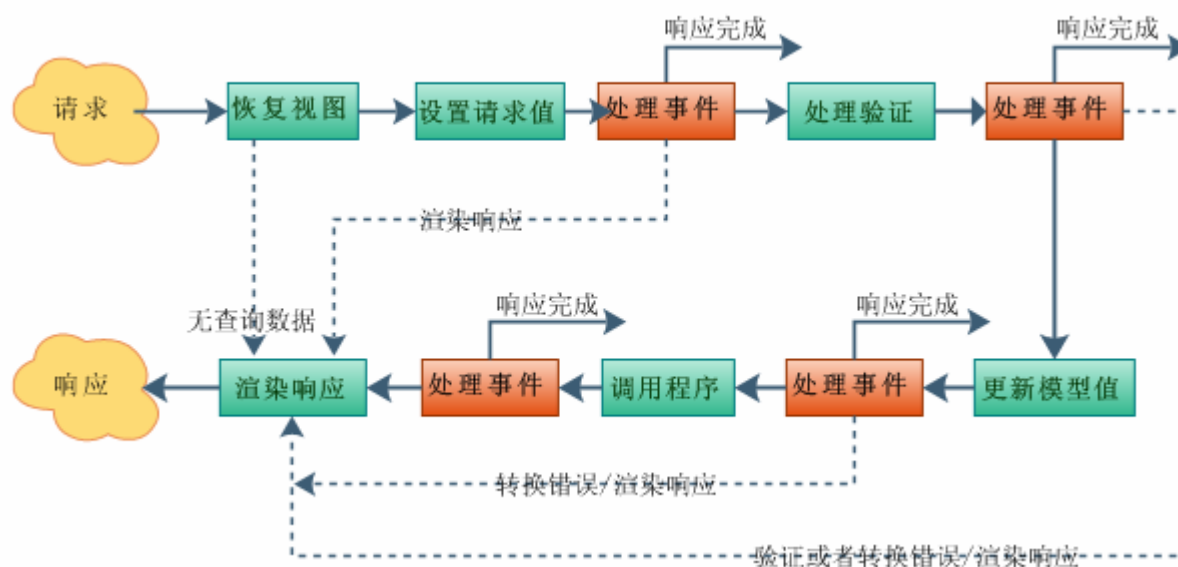


图 14.19 JSF 生命周期图

JSF 支持事件处理模型，虽然由于 HTTP 本身无状态（stateless）的特性，使得这个模型多少有些地方和图形界面编程的仍不太相同，但 JSF 所提供的事件处理模型已足以让一些传统 GUI 程序的设计人员，可以用类似的模型来开发程序。

在简单的 JSF 应用中，我们根据动作方法（action method）的结果来决定要导向的网页，一个命令按钮连接到一个方法，这样的做法实际上就是 JSF 所提供的简化的事件处理程序，在按钮上使用 action 连接到一个动作方法，实际上 JSF 会为其自动产生一个预先设

置好的 `ActionListener` 来处理事件，并根据其传回值来决定导向的页面。根据生命周期图中的顺序，添加的时间处理将会先于调用程序（其实就是 `Action` 方法）。

在接下来的例子中，我们将会对上面的简单登录应用略作修改，来演示如何开发事件处理和最简单的使用导航规则的命令按钮。

首先我们要基于 `HelloJSF` 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 `JSFEvent`，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。不过这个复制后的项目默认发布后的应用名称仍然是 `HelloJSF`，所以我们可以选择菜单 **Project > Properties**，然后在项目属性对话框里面进行修改，如图 14.20 红色部分所示。

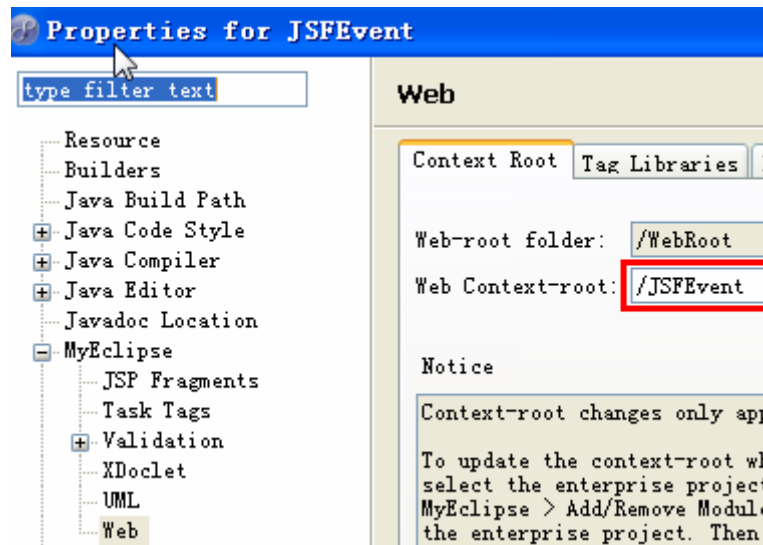


图 14.20 修改项目的发布目录

如果您需要使用同一个方法来应付多种事件来源，并想要取得事件来源的相关信息，您可以让处理事件的方法接收一个 `javax.faces.event.ActionEvent` 事件参数。现在我们在 `UserBean` 中加以修改，给它添加一个事件处理方法 `public void checkInput(ActionEvent evt)`，这个方法将对用户输入的用户名的取值进行验证，如果发现为空，就设置 `UserBean` 的 `message` 属性值为出错提示信息，并设置状态变量 `hasError` 为 `true`。然后先前的进行登录操作的方法 `login()` 的判断流程也略加修改，使之在发现 `hasError` 为 `true` 的情况下，停止向下执行，返回到原始页面（返回值为 `null`）。下面是 `UserBean.java` 的完整代码，修改过的内容以粗斜体的形式显示：

```
package com.jsfdemo;

import javax.faces.event.ActionEvent;

// 用户 Bean
public final class UserBean extends Object {
    boolean hasError;//是否有错误发生

    /**
     * 进行登录操作.
     * @return 导航规则地址
    */
}
```

```

    */
    public String login() {
        System.out.println("login()");
        if(hasError) {
            return null;// 回到输入页面
        }
        if(getUserName().equals("myeclipse") &&
getPassword().equals("myeclipse")) {
            return "success";
        }

        return "failure";
    }

    /**
     * 事件监听器，用来检查用户的输入是否有效，这里只检查用户名，出错就设置状态变量
     和提示信息。
     * 事件的调用发生在Action之前。
     * @param evt
     */
    public void checkInput(ActionEvent evt) {
        System.out.println(evt);
        if(getUserName() == null || getUserName().equals("")) {
            setMessage("用户名不能为空");
            hasError = true;
            return;
        }

        hasError = false;
    }

    private String userName;
    private String password;
    private String message;

    public String getMessage() {
        return message;
    }

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

```

```

public String getUserName() {
    return userName;
}

public void setUserName(String userName) {
    this.userName = userName;
}

public String getPassword() {
    return password;
}

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

```

。在 `login()` 和 `checkInput(ActionEvent evt)` 这两个方法中，我用 `System.out.println` 向控制台输出了一些信息，这样在最后执行的时候，加以比较，大家就可以看出来事件代码和动作代码的执行顺序。

定义完了这些事件后，我们就要使用了。还记得我们在 [14.4.4](#) 一节简单介绍过编辑 `CommandButton` 时候的属性吧，现在就可以通过 `actionListener` 属性给它加入事件监听器，修改过的代码如下：

```

<h:commandButton id="submit" type="submit" action="#{UserBean.login}"
value="登录" actionListener="#{UserBean.checkInput}" ></h:commandButton>

```

。粗斜体部分即为加入的事件处理代码。另外，为了显示出错信息，还要加入一个 `h:outputText` 标记来显示 `UserBean` 的 `message` 属性，并设置字体颜色为红色。此时表单部分的代码如下所示：

```

<h:form id="loginForm">
    <font color="red"><h:outputText value="#{UserBean.message}"/></font>
    <br><h:outputLabel for="userName"
value="#{bundle.username_label}"></h:outputLabel>
        <h:inputText id="userName"
value="#{UserBean.userName}"></h:inputText>
        <br>密码: <h:inputSecret id="password"
value="#{UserBean.password}"></h:inputSecret>
        <br><h:commandButton id="submit" type="submit"
action="#{UserBean.login}" value="登录"
actionListener="#{UserBean.checkInput}" ></h:commandButton>
</h:form>

```

。

现在，让我们发布项目并进行测试，运行后键入地址：
<http://localhost:8080/JSFEvent/userLogin.faces>，在页面中保持内容为空，然后点击登录

按钮, 可以看到浏览器显示的内容如图 14.21 所示。可以看到会显示红色的出错提示, 不过, 其实JSF内置有自己的表单验证机制, 我们就不多做介绍了, 大家可以照着 14.8 节参考资料里面的内容做练习。此时看看Console视图中服务器的输出为:

```
javax.faces.event.ActionEvent[source=javax.faces.component.html.HtmlCommandButton@191f801]
```

```
login()
```

, 这说明事件处理方法执行在线, 之后才是登录处理的方法。

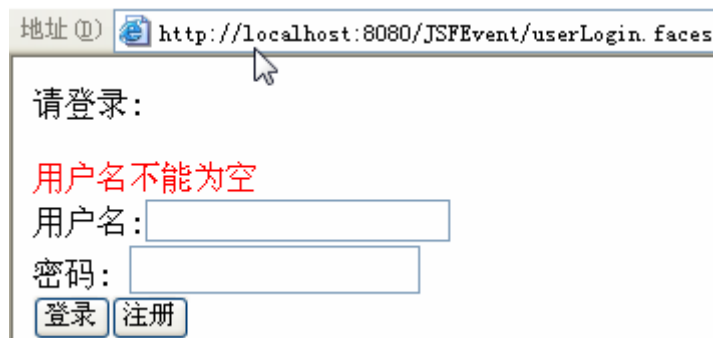


图 14.21 加入事件验证的登录页面执行效果

14.5.2 基于导航规则的命令按钮 action

我们要做的第二个测试, 是让命令按钮的 action 直接为导航规则, 来演示页面跳转的一种方式。首先创建 JSP 页面 **userRegSuccess.jsp**, 内容如下:

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

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

    <title>注册成功</title>

</head>

<body>
    <f:view>
        你好 <h:outputText
value="#{UserBean.userName}"></h:outputText>, 您已经成功注册!
    </f:view>
</body>
</html>
```

, 这个页面的内容无非就是显示一句提示告诉用户成功注册, 并把用户输入的用户名提取出

来。

随后，在 **faces-config.xml** 中新加入一个名为 **regok** 的导航规则，让它直接指向这个 JSP 页面，如图 14.2 所示。对应的配置文件源代码如下所示：

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

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">

  <managed-bean>
    <managed-bean-name>UserBean</managed-bean-name>
    <managed-bean-class>com.jsfdemo.UserBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>userName</property-name>
      <property-class>java.lang.String</property-class>
      <value></value>
    </managed-property>
    <managed-property>
      <property-name>password</property-name>
      <property-class>java.lang.String</property-class>
      <value></value>
    </managed-property>
  </managed-bean>
  <navigation-rule>
    <from-view-id>/userLogin.jsp</from-view-id>
    <navigation-case>
      <description>登录成功</description>
      <from-outcome>success</from-outcome>
      <to-view-id>/userLoginSuccess.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <description>登录失败</description>
      <from-outcome>failure</from-outcome>
      <to-view-id>/userLogin.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <description>注册成功</description>
      <from-outcome>regok</from-outcome>
      <to-view-id>/userRegSuccess.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
```

```

<navigation-rule>
    <from-view-id>/userLoginSuccess.jsp</from-view-id>
</navigation-rule>
<navigation-rule>
    <from-view-id>/userRegSuccess.jsp</from-view-id>
</navigation-rule>
</faces-config>

```

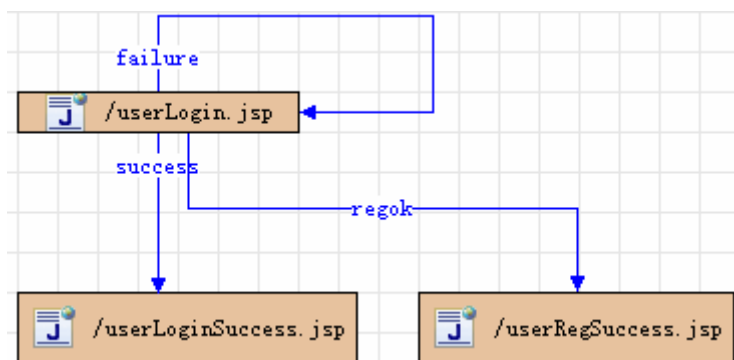


图 14.22 新加入的 regok 导航规则

接着，在 userLogin.jsp 中新加入一个命令按钮，代码片段如下：

```

<h:commandButton id="reg" type="submit" action="regok" value="注册">
</h:commandButton>

```

。代码中的 `action="regok"` 直接指向了一个导航规则，之后再次发布项目，打开页面 <http://localhost:8080/JSFEvent/userLogin.faces>，点击页面中的注册按钮，那么会直接跳转到注册成功的信息提示页面。

14.5.3 加入多个 ActionListener 类

如果您要注册多个 `ActionListener`，例如当用户按下按钮时，顺便做一下日志记录，您可以实现接口 `javax.faces.event.ActionListener`，就像做桌面应用开发时候所做的那样，可以给多个按钮加入一个动作监听器。那么我们要开发的这个事件处理类名为 **LogHandler**，其类的源代码如下：

```

package com.jsfdemo;

import javax.faces.event.*;
// 示例的事件监听器
public class LogHandler implements ActionListener {
    public void processAction(ActionEvent e) {
        // 处理Log

        System.out.print(java.text.SimpleDateFormat.getInstance().format(
            new java.util.Date()));
        System.out.println(" 事件源: " + e.getSource());
    }
}

```

```
}

```

。现在我们修改原来的 `CommandButton` 定义, 可以使用 `<f:actionListener>` 卷标向组件注册事件处理器, 例如:

```
<h:commandButton value="注册" action="regok">
    <!-- 单独编写的事件处理类 -->
    <f:actionListener type="com.jsfdemo.LogHandler"/>
    <!-- 可以加入多个 -->
    <f:actionListener type="xxx.xxx.AnotherHandler"/>
</h:commandButton>
```

。 `<f:actionListener>` 会自动产生 `type` 所指定的对象示例, 并调用组件的 `addActionListener()` 方法注册 `Listener`, 然后操作组件的时候, 事件就可以发生, 然后就调用事件处理器中的方法 `processAction(ActionEvent e)`。

那么最后经过修改后的注册按钮代码片段如下:

```
<h:commandButton id="reg" type="submit" action="regok"
value="注册">
    <!-- 单独编写的事件处理类 -->
    <f:actionListener type="com.jsfdemo.LogHandler"/>
</h:commandButton>
```

。此时重新发布项目, 然后在浏览器中进行测试, 点击注册按钮后, 可以看到 `Console` 中服务器的输出信息如下:

08-3-27 下午 11:04 事件源:

```
javax.faces.component.html.HtmlCommandButton@df303
```

, 这说明虽然是个简单的重定向的命令按钮, 也能触发事件的处理。

为了便于大家对比, 在这里列出完整的 `userLogin.jsp` 的页面代码:

```
<%@ page language="java" pageEncoding="GBK"%>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>

<%
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>请登录</title>
</head>
```

```

<body>
  <f:view>
    <f:loadBundle basename="Messages" var="bundle"/>
    <h:outputText value="#{bundle.login_label}"/>
    <h:form id="loginForm">
      <font color="red"><h:outputText
value="#{UserBean.message}"/></font>
      <br><h:outputLabel for="userName"
value="#{bundle.username_label}"/></h:outputLabel>
      <h:inputText id="userName"
value="#{UserBean.userName}"/></h:inputText>
      <br>密码: <h:inputSecret id="password"
value="#{UserBean.password}"/></h:inputSecret>
      <br><h:commandButton id="submit" type="submit"
action="#{UserBean.login}" value="登录"
actionListener="#{UserBean.checkInput}" ></h:commandButton>
      <h:commandButton id="reg" type="submit" action="regok"
value="注册">
        <!-- 单独编写的事件处理类 -->
        <f:actionListener type="com.jsfdemo.LogHandler"/>
      </h:commandButton>
    </h:form>
  </f:view>
</body>
</html>

```

14.6 JSF 中的内置依赖注入

在学习 Spring 的时候, 我们知道 Spring 的一大功能就是通过配置文件的方式实现依赖注入, 实际上 JSF 也有这样的功能, 可以向 ManagedBean 注入另一个 ManagedBean 对象, 或者注入普通的对象。例如 Spring 中常有这样的配置片段:

```
<bean id="UserManager" class="com.jsfdemo.biz.UserManager" />
```

```

<bean id="UserBean" class="com.jsfdemo.UserBean">
  <property name="userManager">
    <ref bean="UserManager" />
  </property>
  <property name="username">
    <value type="java.lang.String">张三</value>
  </property>
</bean>

```

。各位已经很清楚这段代码的意义, 就是先定义一个名为 *UserManager* 的 bean, 然后将它

作为属性 `userManager` 注入 `UserBean` 的值（用 `ref` 的方式注入），同时还可以注入类型为字符串，值为“张三”的属性 `username`。而在我们前面开发的 `HelloJSF` 应用中的 `faces-config.xml` 中，有这样的代码片段：

```
<managed-property>
    <property-name>userName</property-name>
    <property-class>java.lang.String</property-class>
    <value></value>
</managed-property>
```

，其中 `value` 部分可以修改为：`<value>张三</value>`，这样就实现了类似于 Spring 的 bean 简单属性注入。那么能不能实现类似于 Spring 的 `<ref bean="userManager" />` 这样的注入方式呢？答案是肯定的。JSF 配置文件 `faces-config.xml` 中的对应代码片段是：

```
<managed-bean>
    <managed-bean-name>userManager</managed-bean-name>
    <managed-bean-class>com.jsfdemo.biz.UserManager</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
    <managed-bean-name>UserBean</managed-bean-name>
    <managed-bean-class>com.jsfdemo.UserBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>userName</property-name>
        <property-class>java.lang.String</property-class>
        <value>张三</value>
    </managed-property>
    <managed-property>
        <property-name>userManager</property-name>
        <value>#{userManager}</value>
    </managed-property>
</managed-bean>
```

。 `#{userManager}` 这个取值，就是一个动态的 bean 属性应用定义。

当然，前提是 `UserBean` 要加入一段属性定义的代码：

```
private UserManager userManager;

public UserManager getUserManager() {
    return userManager;
}

public void setUserManager(UserManager userManager) {
    this.userManager = userManager;
}
```

。读者可以在下一节看到完整的两个类（`UserManager` 和 `UserBean`）的代码清单，记住，

这个配置是和 Spring 没有任何关系的，只要是 JSF 就支持。随后用户就可以在 UserBean 中使用注入的 userManager 属性了。读者可以在 *HelloJSF* 项目中加以修改，然后运行进行测试。

14.7 JSF 整合 Spring 开发

14.7.1 简介

随着 Spring 的流行，越来越多的项目开始使用它来整合各种各样的其它框架，在此我们也简单的加以讨论如何使用 Spring 整合 JSF。如我们以前所讨论的，只要 JSF + Spring 成功，那么 Spring + Hibernate 照搬以前的开发过程即可实现完整的 JSF+Spring+Hibernate。整合的文档位于官方文档《Spring Framework 开发参考手册》的 **15.3. JavaServer Faces** 一节，内容并不多，摘录来供大家参考：

JavaServer Faces (JSF) 是一个基于组件的，事件驱动的 Web 框架。这个框架很受欢迎。Spring 与 JSF 集成的关键类是 DelegatingVariableResolver。

15.3.1. DelegatingVariableResolver

将 Spring 中间层与 JSF Web 层整合的最简单办法就是使用 [DelegatingVariableResolver](#) 类。要在应用中配置变量解析器 (Variable Resolver)，你需要编辑 *faces-context.xml* 文件。在 `<faces-config/>` 元素里面增加一个 `<application/>` 元素和一个 `<variable-resolver/>` 元素。变量解析器的值将引用 Spring 的 DelegatingVariableResolver。例如：

```
<faces-config>
  <application>
    <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>
    <locale-config>
      <default-locale>en</default-locale>
      <supported-locale>en</supported-locale>
      <supported-locale>es</supported-locale>
    </locale-config>
    <message-bundle>messages</message-bundle>
  </application>
</faces-config>
```

DelegatingVariableResolver 首先会将查询请求委派到 JSF 实现的默认的解析器中，然后才是 Spring 的“business context” WebApplicationContext。这使得在 JSF 所管理的 bean 中使用依赖注入非常容易。

JSF 所管理的 bean 都定义在 *faces-config.xml* 文件中。下面例子中的 #{userManager} 是一个取自 Spring 的“business context”的 bean。

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
  <managed-bean-class>com.whatever.jsf.UserList</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
```

```

<managed-property>
  <property-name>userManager</property-name>
  <value>#{userManager}</value>
</managed-property>
</managed-bean>

```

15.3.2. FacesContextUtils

如果所有属性已经映射到 `faces-config.xml` 文件中相关的bean，一个自定义的 `VariableResolver` 也可以工作的很好。但是有些情况下你需要显式获取一个bean。这时，[FacesContextUtils](#) 可以使这个任务变得很容易。它类似于 `WebApplicationContextUtils`，不过它接受 `FacesContext` 而不是 `ServletContext` 作为参数。

```

ApplicationContext ctx =
FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());

```

我们推荐使用 `DelegatingVariableResolver` 实现 JSF 和 Spring 的集成。如果你想要更全面的集成，可以看看 [JSF-Spring](#) 这个项目。

引文的内容到此结束，正如大家所看到的那样，官方文档对如何整合的描述就这么多，对于老手来说，也许已经很清楚，不过对于初次接触的人来说，包括我也是初次，看了之后似有所得，然而，这段文档却是要结合前文才能读懂的，所以 Spring 的官方文档总是让人读了之后还很迷惑。下面我们就来做整合的例子，最终目的是通过 Spring 来创建受管 Bean，或者向已存在的受管 Bean 中注入其它的 Bean 定义。这个例子基于我们的 HelloJSF 项目修改而来。

14.7.2 创建项目 jsfspring 并修改 Java 类

读者参考 [14.5.1 位于 Managed Bean 中的事件处理方法](#) 一节的内容，用复制粘贴的方式，创建一个新的 Web 项目，名为 `jsfspring`，并记得修改项目的发布目标目录。接着我们要给项目加入 Spring 开发功能，最少的依赖包是 Spring Core Libraries 和 Spring Web Libraries。可以使用 MyEclipse 的添加 Spring 开发功能向导来完成，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 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 Core Libraries** 和 **Spring 2.0 Web Libraries**；如果您要日后整合其它框架例如 Hibernate，除了在选择类库时还要额外选中 **Spring 2.0 Persistence Core Libraries**，还需要在 **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（这样做主要是为了修正 MyEclipse 自带包的一个名为 ASM 出错的 Bug，如果不加入 Hibernate 等包的话，不需要选中此单选钮）。那么第二页选择 Spring 配置文件的位置保持原来的不变即可，然后点击 **Finish** 按钮结束整个向导。稍等片刻后，所需的 Spring 的 jar 文件都会添加到当前项目，这时候点击并展开 **Package Explorer** 视图中项目的 *Referenced Libraries* 节点就可以看到相关的 jar 文件都已经添加完毕了。

接下来我们要修改一些后台的类和配置文件，而前台的页面不需要做任何变动。首先在

src 目录下新建一个模拟用户管理器的业务类: `com.jsfdemo.biz.UserManager`, 其代码清单如下:

```
package com.jsfdemo.biz;

/**
 * 模拟的用户管理业务层类。
 * @author BeanSoft
 */
public class UserManager {
    /**
     * 登录判断。
     * @param username 用户名
     * @param password 密码
     * @return 成功时返回 true
     */
    public boolean checkLogin(String username, String password) {
        System.out.println(this);

        if ("myeclipse".equals(username) &&
            "myeclipse".equals(password)) {
            return true;
        }

        return false;
    }
}
```

。这个类的功能很简单, 就是加入了一个登录判断功能, 读者可以把它改写成调用 JPA 或者 Hibernate 开发的 DAO 类。最后这个类要作为一个属性被注入到我们的受管 Bean 中去。方法中加入了一句打印, 向控制台输出点信息, 便于确认是否真的是执行了这个类中的方法。

随后当然需要让 `UserBean` 做一些改动, 加入 `userManager` 属性定义, 并在登录之时, 使用被注入的 `UserManager` 对象来完成判断功能, `UserBean.java` 完整代码清单如下:

```
package com.jsfdemo;

import com.jsfdemo.biz.UserManager;

// 类似于Action类的用户受管 Bean
public final class UserBean extends Object {
    private UserManager userManager; // 用户业务对象

    public UserManager getUserManager() {
        return userManager;
    }

    public void setUserManager(UserManager userManager) {
```

```

        this.userName = userManager;
    }

    /**
     * 进行登录操作.
     * @return 导航规则地址
     */
    public String login() {
        if(getUserManager().checkLogin(getUserName(), getPassword())) {
            return "success";
        }

        return "failure";
    }

    private String userName;
    private String password;

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getPassword() {
        return password;
    }

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

```

14.7.3 修改配置文件并在 JSF 中注入 Bean

配置文件中第一个需要修改的地方就是 `WebRoot/WEB-INF/web.xml` 文件的内容，除了需要加入 JSF 的启动 Servlet 之外，我们还要配置并启动 Spring 的 Web 容器，并指定哪些配置文件需要被加载。以下是 `web.xml` 的代码清单：

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

```

```

version="2.5"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
  <!-- 用来定位Spring XML文件的上下文配置 -->
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/applicationContext*.xml,classpath*:applicationContext*.xml
    </param-value>
  </context-param>

  <context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/faces-config.xml</param-value>
  </context-param>

  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.faces</url-pattern>
  </servlet-mapping>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <!-- 启动 Spring Bean 工厂的监听器 -->
  <listener>
    <listener-class>
      org.springframework.web.context.ContextLoaderListener
    </listener-class>
  </listener>

</web-app>

```

。粗体部分已经显示了要做的事情，就是指定 Spring 配置文件的位置，以及加载 Spring 的 Web 版本的监听器，这个监听器将会启动 Spring 的 Bean 工厂。在名为 *contextConfigLocation* 的上下文参数中，我们指定了两类能够自动被 Spring 所识别的配置文件位置，当然也可以加入更多的路径，以逗号隔开即可。这些位置分别是 *WEB-INF/*目录下名为 *applicationContext.xml* 的配置文件，或者是名字以 *applicationContext* 开头的单个或者多个 XML 配置文件；以及类路径，确切的说一般是放在 *WEB-INF/classes* 目录下的同样

名称的配置文件。由于 MyEclipse 自动生成的配置文件的名字是 `applicationContext.xml`，所以我们用这样的配置可以很顺利地加载到这个配置文件。配置文件的最后加入的监听器，则会启动 Spring 的 Bean 工厂，并从 `contextConfigLocation` 这个参数所指定的 XML 配置文件中创建 Bean 类。

第二个要改的配置文件，是 Spring 的 Bean 定义文件 `applicationContext.xml`，当然，我们先按照传统的方式来，在这个文件里加入一个名为 `UserManager` 的 bean 定义，而且如果是正式情况下，还会给这个 Bean 注入一些诸如 DAO 之类的其它属性。

`applicationContext.xml` 的代码清单如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">
    <bean id="userManager" class="com.jsfdemo.biz.UserManager" />
</beans>
```

这第三个要改的，当然是我们的 JSF 配置文件了。这一步没什么好说的，参照 Spring 的文档，第一步是加入一个自定义的变量解析器，就是那段 `<variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>`。它有什么作用呢？简而言之就是代替 JSF 的默认变量解析器，这是什么意思呢？在前面的例子中，我们提到 value 的值可以是 `#{bean 名字.属性或者方法名}`，这就是一段变量。那么要执行它，很明显 JSF 必须从 bean 名字找到 bean 的对象，这一步就是变量解析器所做的工作。那好了，现在问题就好办了，Spring 基于 JSF 的变量解析器做了一点改造，每当看到 `#{bean 名字.}` 时，先让 JSF 去 faces 配置文件中找，找到了就返回，找不到了就从 Spring 的 Bean 工厂里面找。图 14.23 展示了这一过程，注意优先级一定是 JSF 的定义比 Spring 定义的高。因此，现在配置一个 Bean 有两种可选方案，JSF 或者 Spring，分别对应于 `applicationContext.xml` 和 `faces-config.xml`，我们先演示基于 JSF 的配置方式，稍后再演示完全用 Spring 的配置方式。而且在 JSP 页面中的命令按钮的 action 标签的取值，也是变量，所以这个变量解析器在 JSP 页面中也会产生作用。好了，说到这里，也许有人已经和 14.6 节的内容联系起来了，不过，稍有不同的是，JSF 配置文件中不需要再额外定义那个 `UserManager` 的受管 Bean 了，此时的 `faces-config.xml` 完整代码清单如下：

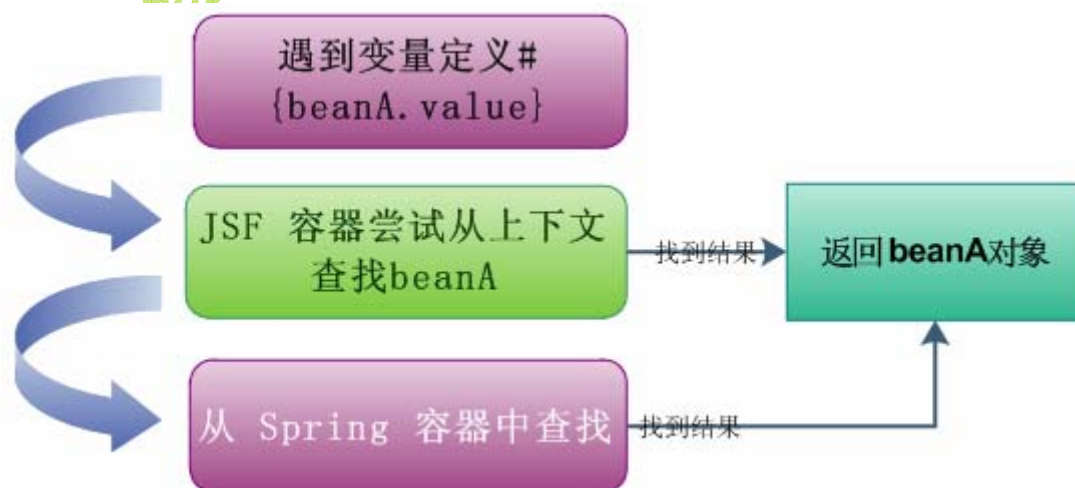


图 14.23 Spring 变量解析器的工作流程

```

<?xml version='1.0' encoding='UTF-8'?>

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>

  <managed-bean>
    <managed-bean-name>UserBean</managed-bean-name>
    <managed-bean-class>com.jsfdemo.UserBean</managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
      <property-name>userName</property-name>
      <property-class>java.lang.String</property-class>
      <value></value>
    </managed-property>
    <managed-property>
      <property-name>password</property-name>
      <property-class>java.lang.String</property-class>
      <value></value>
    </managed-property>
    <managed-property>
      <property-name>userManager</property-name>
      <value>#{userManager}</value>
    </managed-property>
  </managed-bean>

  <navigation-rule>
    <from-view-id>/userLogin.jsp</from-view-id>
    <navigation-case>
      <description>登录成功</description>
      <from-outcome>success</from-outcome>
      <to-view-id>/userLoginSuccess.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
</faces-config>

```

```

</navigation-case>
<navigation-case>
    <description>登录失败</description>
    <from-outcome>failure</from-outcome>
    <to-view-id>/userLogin.jsp</to-view-id>
</navigation-case>
</navigation-rule>
<navigation-rule>
    <from-view-id>/userLoginSuccess.jsp</from-view-id>
</navigation-rule>

</faces-config>

```

。和 14.7 节的代码做比较的话，会发现这个文件中已经不需要名为 `UserManager` 的受管 Bean 定义，那么它在哪里呢？它现在位于 Spring 的配置文件中，和那个 id 为 `UserManager` 的 Bean 定义相匹配。

OK了，别的地方就不需要做改动了，发布项目并运行后，键入地址进行测试：<http://localhost:8080/jsfspring/userLogin.faces>，当然功能还是不变，输入用户名和密码后，可以看到服务器控制台输出如下的信息：

```
com.jsfdemo.biz.UserManager@1e2c841
```

。很好，这说明成功调用了 `UserManger` 类中的登录验证方法了，而这个对象的实例是从 Spring 中获得的。我们推荐这种方式来进行配置，因为可以在 JSF 可视化流程设计器中看到页面流程，还可以进行一些 JSF 的特定设置，例如：作用域设置为 `session`。

14.7.4 完全使用 Spring 配置 Bean

另一种方式，就是完全在 Spring 配置文件中声明所有的受管 Bean。在这种方式下，Spring 的配置文件定义了所有的 Bean 及其依赖关系，不过缺点就是无法定义一些比较细致的内容，例如 JSF 特定的参数，作用域等等。此时的 Spring 配置文件内容 `applicationContext.xml` 代码清单如下：

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="userManager" class="com.jsfdemo.biz.UserManager" />

    <bean id="userBean" class="com.jsfdemo.UserBean">
        <property name="userManager">
            <ref bean="userManager" />
        </property>
    </bean>

</beans>

```

。这页面定义了两个 **Bean**，并配置了依赖关系，之后就可以在 **JSF** 的页面里面访问并使用 **UserBean** 了。**JSF** 的配置文件 **faces-config.xml** 则不需要设置任何 **ManagedBean** 了：

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

<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
  version="1.2">
  <application>
    <variable-resolver>
      org.springframework.web.jsf.DelegatingVariableResolver
    </variable-resolver>
  </application>

  <navigation-rule>
    <from-view-id>/userLogin.jsp</from-view-id>
    <navigation-case>
      <description>登录成功</description>
      <from-outcome>success</from-outcome>
      <to-view-id>/userLoginSuccess.jsp</to-view-id>
    </navigation-case>
    <navigation-case>
      <description>登录失败</description>
      <from-outcome>failure</from-outcome>
      <to-view-id>/userLogin.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  <navigation-rule>
    <from-view-id>/userLoginSuccess.jsp</from-view-id>
  </navigation-rule>
</faces-config>
```

。此时重新发布，并运行项目，会得到同样的结果，唯一的区别就是 **UserBean** 的作用域不是 **session**。

14.8 JSF+JPA 的 MyEclipse 官方 Blog 实例

MyEclipse 6 提供了一套名为 **Example On-Demand** 的官方实例代码库，这些例子都是无须额外设置和修改就可以运行的，数据库用的是 **MyEclipse Derby**。有很多例子，其中有一个是 **JSF+JPA** 开发的简易 **Blog**。我们先来看看这个代码库怎么用，首先选择菜单 **MyEclipse > Example On-Demand...**，将会打开一个名为 **Example On-Demand Browser**

的编辑器窗口，下面列出了可以下载安装的例子列表，如图 14.24 所示。要安装例子，点击页面中的按钮 **Install Project** 就可以了，此时 MyEclipse 6 会下载并安装项目，过程如图 14.25 所示，其实它是从 CVS 中向外导出项目的（check out，也有人叫检出项目）。之后，项目就会出现在工作区并自动打开，也许这时会出现一些错误提示，不用管它。一般项目打开后就会自动启动 Derby 数据库。



图 14.24 Example On-Demand Browser

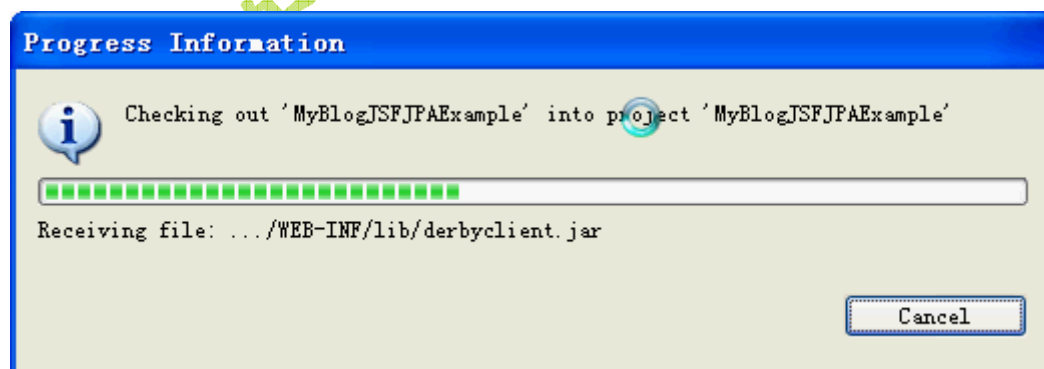


图 14.25 项目下载过程

这些项目的许可协议都是 Apache 2，因此可以用于我们自己的项目中。目前有下面一些项目列表：

项目名称	简介
MyBlogJSFJPAExample	简易 JSF+JPA 的项目
GoogleMapsExample	简易的基于 Google Map 的例子

MERSimpleReportExample	简易报表示例
DWRSpringJPAExample	基于 DWR 和 Spring, JPA 的 AJAX 例子
MyBlogStrutsHibernateExample	基于 SSH 的 Blog 实例
JavaServletExamples	简单 Servlet 例子
Struts13CookbookExample	Struts Cookbook 中的 Struts 1.3 例子
M4MContactManagerExample	简易的管理个人地址簿的 Swing 应用
SimpleJSPEXample	简单的 JSP 例子
SimpleFaceletsExample	简单的 JSF 的 Facelets 例子
TerraServerWSCClientExample	Web Service 客户端例子
JSP20Examples	JSP 2.0 的例子
JSFLoginExample	JSF 的简易登录例子
StrutsLoginExample	Struts 的简易登录例子
ICEfacesTutorialExample	支持 AJAX 的 ICEFaces JSF 例子

。读者可以挑选自己感兴趣的项目下载，运行。不过我们这里讨论的项目是第一个：*MyBlogJSFJPAExample*。像所有的这种例子一样，它有一份 README.txt 的说明文件，内容我们来翻译一下：

MyBlog JSF/JPA CRUD Example Project

MyBlog JSF/JPA 增删改读 示例项目

* 许可

项目以 Apache 2 协议提供，请阅读文件 LICENSE.txt 获取更多信息。

* 说明

MyBlogJSFJPA 是一个简单的使用 JSF 作为界面和 JPA 作为持久层的简单的网络博客应用。

这个项目展示了对单个博客数据库表进行增删改查的功能。同时这个项目已经事先在 MyEclipse 自带的服务器环境中配置好了，例如内置的 Tomcat 6 和 Derby 数据库，数据库中已经包含了一个 MyEclipse 6.0 出厂时就带的示例数据库。

项目的结构由如下几个部分组成：

1. 一个 JSF 页面链接到 4 个对提交的数据库表进行增删改查的功能模块
2. 一个受管 Bean 作为维护 UI 状态的控制器并处理持久化操作
3. 一个发帖的 JPA 实体以及一个处理持久化的 PostDAO
4. 在 MyEclipse 中已经自带了这个例子所需要的 MYBLOG 数据库

为了帮助您理解 MYBLOG 数据库的结构，在项目的 resources 目录下已经包含了它的实体关系图。双击文件 MYBLOG.mer 即可查看数据库的实体关系图。

要生成我们自己的实体关系图，可以通过打开 MyEclipse Database Explorer 透视图，然后右键点击希望生成 ER 图的数据库，然后选择菜单 "New ER Diagram"，之后选中 ER 图需要存放的位置即可。

* 系统需要

- * MyEclipse 6.0 或者更高版本
- * Java SE 5 或者更高版本

* 如何运行

可以右键点击项目，然后选中菜单 **Debug As** 或者 **Run As**，接着选择 "MyEclipse Server Application"。MyEclipse 6.0 或者更高版本将会自动发布项目到 MyEclipse Tomcat 服务器，然后自动启动，并在启动结束后，打开浏览器并显示应用的首页 `index.jsp`，接着就可以进行测试了。

注意：如果发现启动时发生了一个 HTTP 500 错误或者异常，像下面这样的信息：

```
SEVERE: Servlet.service() for servlet jsp threw exception
java.net.ConnectException: Connection refused: connect
```

这是因为有时 MyEclipse Derby 服务器没有自动启动。要修正这个问题，请在 MyEclipse 透视图切换到 **Servers** 视图，手工启动 MyEclipse Derby 服务器，之后在重新启动应用。

* 相关链接

- * MyEclipse JSF 教程 - <http://www.myeclipseide.com/documentation/quickstarts/jsf/>
- * MyEclipse JPA 教程 - <http://www.myeclipseide.com/documentation/quickstarts/jpa/>
- * MyEclipse JSF 设计器教程 - <http://www.myeclipseide.com/documentation/quickstarts/jsfdesigner/>

* 反馈

我们希望这个示例项目能对您有所帮助。如果您在使用示例项目时遇到了任何问题，欢迎发帖到我们的 **Example Project Forum** (<http://www.myeclipseide.com/PNphpBB2-viewforum-f-54.html>) 并通知我们。同时如果您有任何改进建议，或者明显的错误，或者只是问一些问题，我们鼓励您发帖并通知我们！

当然了，这份说明大家可以不看，直接看如何运行。这个项目的代码，我们已经包含在光盘或者相关的邮件里了，首先启动 MyEclipse Derby 服务器，如图 14.26 所示。

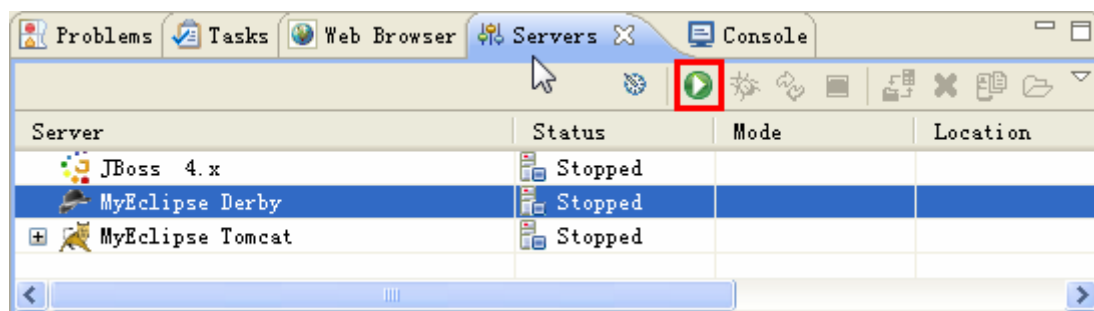



图 14.26 启动 Derby 服务器

点击图中的红色部分的运行按钮，即可启动数据库服务器。启动后可以在 Console 视图看到类似于下面的输出：

Apache Derby Network Server — 10.2.2.0 - (485682) 已启动并且已准备好 2008-03-30 10:18:51.281 GMT 时在端口 1527 上接受连接

。这就是服务器启动成功了，接着要做的就是 在 **Package Explorer** 视图中选中项目节点 **MyBlogJSFJPAExample**，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。服务器启动完毕之后，可以看到在 MyEclipse 6 窗口内出现一个浏览器并显示应用的入口页面，如图 14.27 中上方屏幕所示，之后点击页面中的 **Click here** 链接开始测试，接着就可以显示到相关的链接了，包括列表，新建，修改和删除。限于篇幅的关系，我们就不再这里列出详细的代码了，读者可以自行打开项目，阅读代码进行学习。需要提示的是 JPA 的 DAO 层代码可以用第十三章中的反向工程向导来生成。

也许有读者注意到项目的名字是：**MyBlogJSFJPAExample** [examples.myeclipseide.com]，文件图标下也多了个小小的圆柱体，例如，这是因为这个项目和 CVS 版本控制服务器挂钩了，读者可以阅读相关的 Eclipse 团队工作教程来进行学习，参考最后一节的链接：使用 Eclipse 平台共享代码。

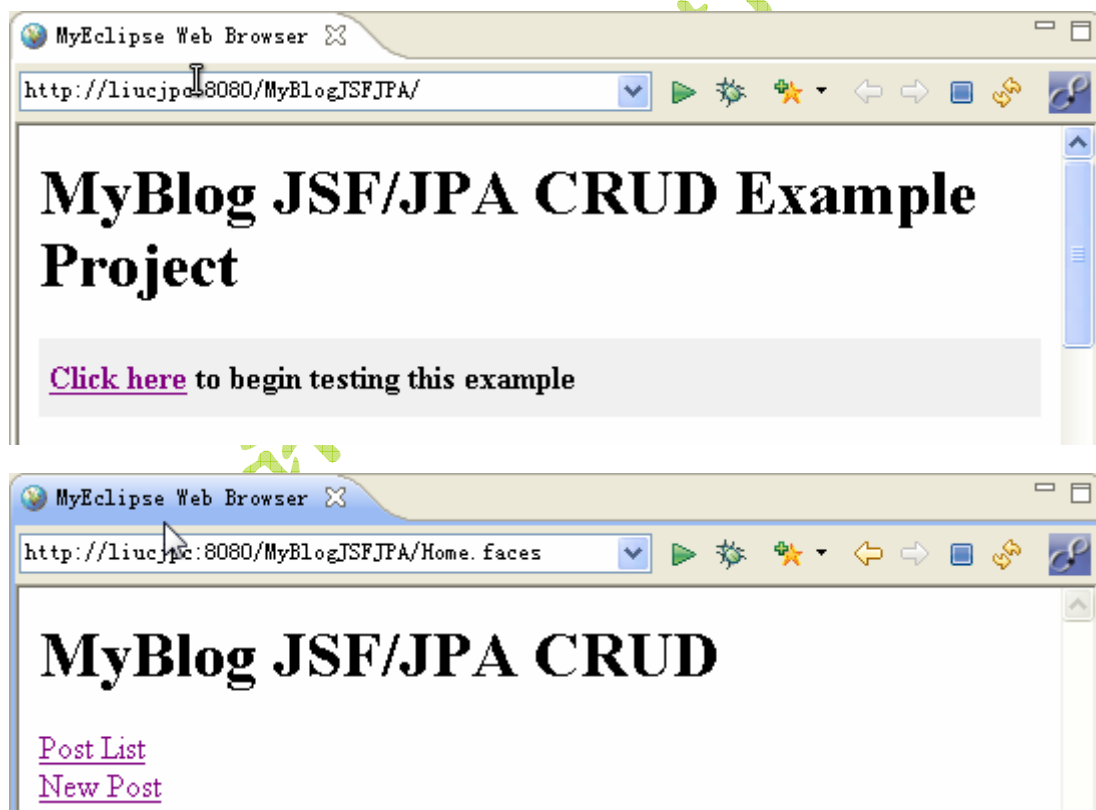


图 14.27 测试示例应用

14.7 小结

在本章，我们就如何在 JSF 中进行可视化开发进行了简要介绍，并介绍了 JSF+Spring

的开发过程。我们介绍的内容并不全面，首先是因为现在国内 JSF 并不如 Struts 系列的框架流行，另外因为 JSF 的各个实现并不兼容，只要大家入门之后，再根据参考资料一节列出的文档进行详细学习，按需学习，即可逐渐掌握。基本上，在项目中只会用到一种 JSF 的实现，例如 AJAX4JSF, ICEFaces 等等，您还需要阅读对应的文档。推荐文档为：《jsf 入门》简体中文版.rar 电子书。

14.8 参考资料

JSF 生命周期详细

[JSF-Spring](#) JSF整合Spring的开源项目

<http://caterpillar.onlyfun.net/Gossip/JSF/JSFLifeCycle.htm> JSF 生命週期

<http://www.hexiao.cn/ajax4jsf/a4jug.html> Ajax4jsf用户指南中文版

<http://www.idon.com/idea/jsf-struts.htm> JSF与Struts的异同

<http://www.icefaces.org/> 开源JSF框架ICEfaces是一个基于Ajax的JSF开发框架。ICEfaces 原本是一个商业产品，现已开源基于Mozilla Public License发布。它提供一整套完整的Java EE应用程序开发组件，能够帮助开发人员用纯Java(not JavaScript)快速开发瘦客户端胖互联网应用程序(Rich Internet Applications:RIA)。

<http://www.myeclipseide.com/documentation/quickstarts/icefaces/> MyEclipse ICEfaces 教程（英文）

<http://labs.jboss.com/jbossrichfaces/> JBoss RichFaces

<http://www.operamasks.org/> 金蝶 Apusic OperaMasks

<http://myfaces.apache.org/> Apache 的MyFaces项目

<http://www.blogjava.net/beansoft/archive/2007/05/10/116486.html> 《jsf入门》简体中文版.rar 电子书下载

<http://www.blogjava.net/beansoft/archive/2007/11/30/164166.html> MyEclipse 6 实战开发讲解视频入门 9 JSF 1.2 入门开发

<http://www.javaworld.com.tw/confluence/pages/viewpage.action?pageId=2630> JSF入门

http://www-128.ibm.com/developerworks/cn/views/java/tutorials.jsp?cv_doc_id=85059

<http://www.blogjava.net/Files/beansoft/j-jsf.zip> IBM 的 JSF 中文入门教程 76KB

<http://www.ibm.com/developerworks/cn/java/j-jsf2/> 怀疑论者的 JSF: JSF 应用程序的生命周期JSF 请求处理生命周期的 5 个阶段之旅

<http://www.ibm.com/developerworks/cn/java/j-jsf3/> 怀疑论者的 JSF: JSF 转换与验证利用 JSF 的转换和验证框架来确保数据模型的完整性

<http://www.ibm.com/developerworks/cn/java/j-jsf4/> 怀疑论者的 JSF: JSF 组件开发省时运动使得构建 JSF 组件轻而易举

<http://www.ibm.com/developerworks/cn/linux/opensource/os-ecshare/index.html> 使用 Eclipse 平台共享代码