

JSF入门

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

繁简转换与修订：jnzgy@163.com

声明：为方便简体中文读者阅读，特将原文转换为简体，对一些习惯用语做了替换，同时对部分内容做了修订。版权归原文作者所有，未经许可，不得用于商业目的。

2006年3月23日

1. 入门.....	2
1.1 简介 JSF.....	2
1.2 第一个 JSF 程序.....	4
1.3 简单的导航 Navigation.....	7
1.4 导航规则设置.....	9
1.5 JSF Expression Language	11
1.6 国际化信息.....	15
2. Managed Beans	17
2.1 Backing Beans.....	18
2.2 Beans 的配置与设定	18
2.3 Beans 上的 List, Map	21
3. 数据转换与验证.....	23
3.1 标准转换器.....	23
3.2 自定义转换器.....	26
3.3 标准验证器.....	31
3.4 自定义验证器.....	32
3.5 错误信息处理.....	35
3.6 自定义转换, 验证标签.....	38
4. 事件处理.....	42
4.1 动作事件.....	42
4.2 即时事件.....	45
4.3 值变事件.....	50
4.4 Phase 事件	53

如果您是从使用的角度来看JSF，则您不用理会HTTP、数据转换等细节，JSF将细节都隐藏起来了，无论您是网页设计人员或是应用程序设计人员，都可以使用自己熟悉的方式来看JSF。

1. 入门

借由以下的几个主题，可以大致了解JSF的轮廓与特性，我们来看看网页设计人员与应用程序设计人员各负责什么。

1.1 简介 JSF

Web应用程序的开发与传统的单机程序开发在本质上存在着太多的差异，Web应用程序开发人员至今不可避免的必须处理HTTP的细节，而HTTP无状态的（stateless）本质，与传统应用程序必须维持程序运行过程中的信息有明显的违背，再则Web应用程序面对网站上不同的使用者同时的存取，其执行线程安全问题以及数据验证、转换处理等问题，又是复杂且难以解决的。

另一方面，本质上是静态的HTML与本质上是动态的应用程序又是一项违背，这造成不可避免的，处理网页设计的美术人员与程序设计人员，必须被彼此加入至视图组件中的逻辑互相干扰，即便一些视图呈现逻辑以标签的方式呈现，试图展现对网页设计美术人员的亲切，但它终究必须牵涉到相关的流程逻辑。

有很多方案试着解决种种的困境，而各自的着眼点各不相同，有的从程序设计人员的角度来解决，有的从网页设计人员的角度来解决，各种的框架被提出，所造成的是各种不统一的标签与框架，为了促进产能的集成开发环境（IDE）难以整合这些标签与框架，另一方面，开发人员的学习负担也不断的加重，他们必须一人了解多个角色的工作。

JavaServer Faces 的提出在试图解决这个问题，它试图在不同的角度上提供网页设计人员、应用程序设计人员、组件开发人员解决方案，让不同技术的人员可以彼此合作又不互相干扰，它综合了各家厂商现有的技术特点，由 Java Community Process（JCP）团队研拟出来的一套标准，并在 2004 年三月发表了 JavaServer Faces 1.0 实现成果。

从网页设计人员的角度来看，JavaServer Faces提供了一套像是新版本的HTML标签，但它不是静态的，而是动态的，可以与后端的动态程序结合，但网页设计人员不需要理会后端的动态部份，网页设计人员甚至不太需要接触JSTL这类的标签，也可以动态的展现数据（像是动态的查询表格内容），JavaServer Faces提供标准的标签，这可以与网页编辑程序结合在一起，另一方面，JavaServer Faces也允许您自定义标签。

从应用程序设计人员的角度来看，JavaServer Faces提供一个与传统应用程序开发相类似的模型（当然因某些本质上的差异，模型还是稍有不同），他们可以基于事件驱动来开发程序，不必关切HTTP的处理细节，如果必须处理一些视觉组件的属性的话，他们也可以直接在整合开发环境上拖拉这些组件，点选设定组件的属性，JavaServer Faces甚至还为应用程序设计人员处理了对象与字符串（HTTP传送本质上就是字符串）间不匹配的转换问题。

从UI组件开发人员的角度来看,他们可以设计通用的UI组件,让应用程序的开发产能提高,就如同在设计Swing组件等,UI开发人员可以独立开发,只要定义好相关的属性选项来调整细节,而不用受到网页设计人员或应用程序设计人员的干扰。

三个角色的知识领域原则上可以互不干扰,根据您的角色,您只要了解其中一个知识领域,就可以运用 JavaServer Faces,其它角色的知识领域您可以不用了解太多细节。

当然,就其中一个角色单独来看,JavaServer Faces隐藏了许多细节,若要全盘了解,其实JavaServer Faces是复杂的,每一个处理的环境都值得深入探讨,所以学习JavaServer Faces时,您要选择的是通盘了解,还是从使用的角度来了解,这就决定了您学习时所要花费的心力。

要使用JSF,首先您要先取得JavaServer Faces参考实现(JavaServer Faces Reference Implementation),在将来,JSF会与Container整合在一起,届时您只要下载支持的Container,就可以使用JSF的功能。

请至 JSF 官方网站的 下载区 下载参考实现,在下载压缩文件并解压缩之后,将其 lib 目录下的 jar 文件复制至您的Web应用程序的/WEB-INF/lib目录下,另外您还需要 jstl.jar 与 standard.jar 文件,这些文件您可以在 sample 目录下,解压缩其中的一个范例,在它的/WEB-INF/lib目录下找到,将之一并复制至您的Web应用程序的/WEB-INF/lib目录下,您总共需要以下的文件:

```
* jsf-impl.jar
* jsf-api.jar
* commons-digester.jar
* commons-collections.jar
* commons-beanutils.jar
* jstl.jar
* standard.jar
```

接下来配置 Web 应用程序的 web.xml,使用 JSF 时,所有的请求都通过 FacesServlet 来处理,您可以如下定义:

- web.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<web-app xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd"
  version="2.4">

  <description>
    JSF Demo
  </description>
  <display-name>JSF Demo</display-name>
  <servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>
```

```

        javax.faces.webapp.FacesServlet
    </servlet-class>
    <load-on-startup>1</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.html</welcome-file>
</welcome-file-list>
</web-app>

```

在上面的定义中，我们将所有.faces 的请求交由 FacesServlet 来处理，FacesServlet 会唤起相对的.jsp 网页，例如请求是/index.faces 的话，则实际上会唤起/index.jsp 网页，完成以上的配置，您就可以开始使用 JSF 了。

1.2 第一个 JSF 程序

现在可以开发一个简单的程序了，我们将设计一个简单的登入程序，使用者提交名称，之后由程序显示使用者名称及欢迎信息。

程序开发人员

先看看应用程序开发人员要作些什么事，我们编写一个简单的JavaBean：

- UserBean.java

```

package onlyfun.caterpillar;

public class UserBean {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}

```

这个Bean将存贮使用者的名称，编译好之后放置在/WEB-INF/classes下。

接下来设计页面流程，我们将先显示一个登入网页/pages/index.jsp，使用者填入名称并提交表单，之后在/pages/welcome.jsp中显示Bean中的使用者名称

与欢迎信息。

为了让JSF知道我们所设计的Bean以及页面流程，我们定义一个 /WEB-INF/faces-config.xml：

- faces-config.xml

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <navigation-rule>
    <from-view-id>/pages/index.jsp</from-view-id>
    <navigation-case>
      <from-outcome>login</from-outcome>
      <to-view-id>/pages/welcome.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <managed-bean>
    <managed-bean-name>user</managed-bean-name>
    <managed-bean-class>
      onlyfun.caterpillar.UserBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>
```

在<navigation-rule>中，我们定义了页面流程，当请求来自<from-view-id>中指定的页面，并且指定了<navigation-case>中的<from-outcome>为login时，则会将请求导向至<to-view-id>所指定的页面。

在<managed-bean>中我们可以统一管理我们的Bean，我们设定Bean对象的存活范围是session，也就是使用者开启浏览器与程序互动过程中都存活。

接下来要告诉网页设计人员的信息是，他们可以使用的Bean名称，即<managed-bean-name>中设定的名称，以及上面所定义的页面流程。

网页设计人员

首先网页设计人员编写index.jsp网页：

- index.jsp

```
<%@taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=GB2312"%>
<html>
```

```

<head>
<title>第一个 JSF 程序</title>
</head>
<body>
  <f:view>
    <h:form>
      <h3>请输入您的名称</h3>
      名称: <h:inputText value="#{user.name}"/><p>
      <h:commandButton value="送出" action="login"/>
    </h:form>
  </f:view>
</body>
</html>

```

我们使用了JSF的core与html标签库，core是有关于UI组件的处理，而html则是有关于HTML的进阶标签。

<f:view>与<html>有类似的作用，当您要开始使用JSF组件时，这些组件一定要在<f: view>与</f:view>之间，就如同使用HTML时，所有的标签一定要在<html>与< /html>之间。

html标签库中几乎都是与HTML标签相关的进阶标签，<h: form>会产生一个表单，我们使用<h: inputText>来显示user这个Bean对象的name属性，而<h:commandButton>会产生一个提交按钮，我们在action属性中指定将根据之前定义的login页面流程中前往welcome.jsp页面。

网页设计人员不必理会表单传送之后要作些什么，他只要设计好欢迎页面就好了：

- welcome.jsp

```

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=GB2312"%>
<html>
<head>
<title>第一个 JSF 程序</title>
</head>
<body>
  <f:view>
    <h:outputText value="#{user.name}"/> 您好！
    <h3>欢迎使用 JavaServer Faces！</h3>
  </f:view>
</body>
</html>

```

这个页面没什么需要解释的了，如您所看到的，在网页上没有程序逻辑，网页设计人员所作的就是遵照页面流程，使用相关名称取出数据，而不用担心实际上程序是如何运行的。

接下来启动 Container，连接上您的应用程序网址，例如：
`http://localhost:8080/jsfDemo/pages/index.faces`，填入名称并提交表单，
您的欢迎页面就会显示了。

1.3 简单的导航 Navigation

在第一个JSF程序中，我们简单的定义了页面的流程由 `index.jsp` 到 `welcome.jsp`，接下来我们扩充程序，让它可以根据使用者输入的名称与密码是否正确，决定要显示欢迎信息或是将使用者送回原页面进行重新登入。

首先我们修改一下UserBean：

- UserBean.java

```
package onlyfun.caterpillar;

public class UserBean {
    private String name;
    private String password;
    private String errMessage;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

    public String getPassword() {
        return password;
    }

    public void setErrMessage(String errMessage) {
        this.errMessage = errMessage;
    }

    public String getErrMessage() {
        return errMessage;
    }
}
```

```
public String verify() {  
    if(!name.equals("justin") ||  
        !password.equals("123456")) {  
        errMsgage = "名称或密码错误";  
        return "failure";  
    }  
    else {  
        return "success";  
    }  
}
```

在UserBean中，我们增加了密码与错误信息属性，在verify()方法中，我们检查使用者名称与密码，它传回一个字符串，"failure"表示登入错误，并会设定错误信息，而"success"表示登入正确，这个传回的字符串将决定页面的流程。

接下来我们修改一下 faces-config.xml 中的页面流程定义：

- faces-config.xml

```
<?xml version="1.0"?>  
<!DOCTYPE faces-config PUBLIC  
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"  
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">  
  
<faces-config>  
    <navigation-rule>  
        <from-view-id>/pages/index.jsp</from-view-id>  
        <navigation-case>  
            <from-outcome>success</from-outcome>  
            <to-view-id>/pages/welcome.jsp</to-view-id>  
        </navigation-case>  
        <navigation-case>  
            <from-outcome>failure</from-outcome>  
            <to-view-id>/pages/index.jsp</to-view-id>  
        </navigation-case>  
    </navigation-rule>  
  
    <managed-bean>  
        <managed-bean-name>user</managed-bean-name>  
        <managed-bean-class>  
            onlyfun.caterpillar.UserBean  
        </managed-bean-class>  
        <managed-bean-scope>session</managed-bean-scope>  
    </managed-bean>  
</faces-config>
```


根据上面的定义，当传回的字符串是"success"时，将前往 welcome.jsp，如果是"failure"的话，将送回 index.jsp。

接下来告诉网页设计人员Bean名称与相关属性，以及决定页面流程的verify名称，我们修改 index.jsp 如下：

- index.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=GB2312"%>
<html>
<head>
<title>第一个 JSF 程序</title>
</head>
<body>
  <f:view>
    <h:form>
      <h3>请输入您的名称</h3>
      <h:outputText value="#{user.errorMessage}" /><p>
      名称: <h:inputText value="#{user.name}" /><p>
      密码: <h:inputSecret value="#{user.password}" /><p>
      <h:commandButton value="送出"
        action="#{user.verify}" />
    </h:form>
  </f:view>
</body>
</html>
```

当要根据verify运行结果来决定页面流程时，action属性中使用 JSF Expression Language "#{user.verify}"，如此JSF就知道必须根据verify传回的结果来导航页面。

<h:outputText>可以取出指定的Bean之属性值，当使用者因验证错误而被送回原页面时，这个错误信息就可以显示在页面上。

1.4 导航规则设置

在JSF中是根据faces-config.xml中<navigation-rule>设定，以决定在符合的条件成立时，该连结至哪一个页面，一个基本的设定如下：

```
....
<navigation-rule>
  <from-view-id>/pages/index.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/pages/welcome.jsp</to-view-id>
  </navigation-case>
```

```

    <navigation-case>
      <from-outcome>failure</from-outcome>
      <to-view-id>/pages/index.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>
  ....

```

对于JSF，每一个视图（View）都有一个独特的标识（identifier），称之为View ID，在JSF中的View ID是从Web应用程序的环境相对路径开始计算，设定时都是以/作为开头，如果您请求时的路径是/pages/index.faces，则JSF会将副文件名改为/pages/index.jsp，以此作为view-id。

在<navigation-rule>中的<from-view-id>是个选择性的定义，它规定了来源页面的条件，<navigation-case>中定义各种导览条件，<from-outcome>定义当表单结果符合的条件时，各自改导向哪一个目的页面，目的页面是在<to-view-id>中定义。

您还可以在<navigation-case>中加入<from-action>，进一步规范表单结果必须根据哪一个动作方法（action method），当中是使用 JSF Expression Language 来设定，例如：

```

  ....
  <navigation-rule>
    <from-view-id>/pages/index.jsp</from-view-id>
    <navigation-case>
      <from-action>#{user.verify}</from-action>
      <from-outcome>success</from-outcome>
      <to-view-id>/pages/welcome.jsp</to-view-id>
    </navigation-case>
    ....
  </navigation-rule>
  ....

```

在导航时，预定义都是使用forward的方式，您可以在<navigation-case>中加入一个<redirect/>，让JSF发出让浏览器重新导向（redirect）的header，让浏览器主动要求新网页，例如：

```

  ....
  <navigation-rule>
    <from-view-id>/pages/index.jsp</from-view-id>
    <navigation-case>
      <from-outcome>success</from-outcome>
      <to-view-id>/pages/welcome.jsp</to-view-id>
      <redirect/>
    </navigation-case>
    ....
  </navigation-rule>
  ....

```

您的来源网页可能是某个特定模组，例如在/admin/下的页面，您可以在<from-view-id>中使用wildcards（通配符），也就是使用 * 字符，例如：

```

....
<navigation-rule>
  <from-view-id>/admin/*</from-view-id>
  <navigation-case>
    <from-action>#{user.verify}</from-action>
    <from-outcome>success</from-outcome>
    <to-view-id>/pages/welcome.jsp</to-view-id>
  </navigation-case>
  ....
</navigation-rule>
....

```

在上面的设定中,只要来源网页是从/admin来的,都可以开始测试接下来的<navigation-case>。

<from-view-id>如果没有设定,表示来源网页不作限制,您也可以使用 * 显式的在定义文件中表明,例如:

```

....
<navigation-rule>
  <from-view-id>/*</from-view-id>
  <navigation-case>
    ....
  </navigation-rule>
....

```

或者是这样:

```

....
<navigation-rule>
  <from-view-id>*</from-view-id>
  <navigation-case>
    ....
  </navigation-rule>
....

```

1.5 JSF Expression Language

JSF Expression Language 搭配 JSF 标签来使用,是用来存取数据对象的一个简易语言。

JSF EL是以#开始,将变量或运算式放置在 { 与 } 之间,例如:

```
#{someBeanName}
```

变量名称可以是faces-config.xml中定义的名称,如果是Bean的话,可以通过使用 '.' 运算符来存取它的属性,例如:

```

...
<f:view>
  <h:outputText value="#{userBean.name}"/>
</f:view>
...

```

在JSF标签的属性上，" 与 "（或'与'）之间如果含有EL，则会加以运算，您也可以这么使用它：

```
...
<f:view>
    名称, 年龄 : <h:outputText
        value="#{userBean.name}, #{userBean.age}"/>
</f:view>
...
```

一个执行的结果可能是这样显示的：

```
名称, 年龄 : Justin, 29
```

EL的变量名也可以程序执行过程中所声明的名称，或是JSF EL预定义的隐含对象，例如下面的程序使用param隐含对象来取得使用者输入的参数：

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=GB2312"%>

<html>
<head>
<title></title>
</head>
<body>
<f:view>
    <b> 您好, <h:outputText value="#{param.name}"/> </b>
</f:view>

</body>
</html>
```

param是JSF EL预定义的隐含对象变量，它代表request所有参数的集合，实际是一个java.util.Map类型对象，JSF所提供的隐含对象，大致上对应于JSP隐含对象，不过JSF隐含对象移除了pageScope与pageContext，而增加了facesContext与view，它们分别对应于 javax.faces.context.FacesContext与 javax.faces.component.UIViewRoot。

对于Map类型对象，我们可以使用 '.' 运算符指定key值来取出对应的value，也可以使用 [与] 来指定，例如：

```
...
<f:view>
    <b> 您好, <h:outputText value="#{param['name']}"/> </b>
</f:view>
...
```

在 [与] 之间，也可以放置其它的变量值，例如：

```
...
<f:view>
    <h:outputText value="#{someBean.someMap[user.name]}"/>
</f:view>
```

...

如果变量是List类型或数组的话，则可以在 [] 中指定索引，例如：

```
....
<f:view>
  <h:outputText value="#{someBean.someList[0]}" />
  <h:outputText value="#{someBean.someArray[1]}" />
  <h:outputText
    value="#{someBean.someListOrArray[user.age]}" />
</f:view>
....
```

您也可以指定字面常数，对于true、false、字符串、数字，JSF EL会尝试进行转换，例如：

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

<h:outputText value="#{'This is a test'}" />
....
```

如果要输出字符串，必须以单引号 ' 或双引号 " 括住，如此才不会被认为是变量名称。

在声明变量名称时，要留意不可与JSF的保留字或关键字同名，例如不可取以下这些名称：

true false null div mod and or not eq ne lt gt le ge instanceof empty

使用EL，您可以直接实行一些算术运算、逻辑运算与关系运算，其使用就如同在一般常见的程序语言中之运算。

算术运算符有：加法 (+)，减法 (-)，乘法 (*)，除法 (/ or div) 与余除 (% or mod)。下面是算术运算的一些例子：

运算式	结果
<code>#{1}</code>	1
<code>#{1 + 2}</code>	3
<code>#{1.2 + 2.3}</code>	3.5
<code>#{1.2E4 + 1.4}</code>	12001.4
<code>#{ -4 - 2}</code>	-6
<code>#{21 * 2}</code>	42
<code>#{3/4}</code>	0.75
<code>#{3 div 4}</code>	0.75，除法

<code>#{3/0}</code>	Infinity
<code>#{10%4}</code>	2
<code>#{10 mod 4}</code>	2, 也是余除
<code>#{(1==2) ? 3 : 4}</code>	4

如同在Java语法一样 (`expression ? result1 : result2`) 是个三元运算, `expression`为true显示`result1`, `false`显示`result2`。

逻辑运算有: `and`(或`&&`)、`or`(或`!!`)、`not`(或`!`)。一些例子为:

运算式	结果
<code>#{true and false}</code>	false
<code>#{true or false}</code>	true
<code>#{not true}</code>	false

关系运算有: 小于Less-than (`<` or `lt`)、大于Greater-than (`>` or `gt`)、小于或等于Less-than-or-equal (`<=` or `le`)、大于或等于Greater-than-or-equal (`>=` or `ge`)、等于Equal (`==` or `eq`)、不等于Not Equal (`!=` or `ne`) , 由英文名称可以得到`lt`、`gt`等运算符之缩写词, 以下是Tomcat的一些例子:

运算式	结果
<code>#{1 < 2}</code>	true
<code>#{1 lt 2}</code>	true
<code>#{1 > (4/2)}</code>	false
<code>#{1 > (4/2)}</code>	false
<code>#{4.0 >= 3}</code>	true
<code>#{4.0 ge 3}</code>	true
<code>#{4 <= 3}</code>	false
<code>#{4 le 3}</code>	false
<code>#{100.0 == 100}</code>	true
<code>#{100.0 eq 100}</code>	true
<code>#{(10*10) != 100}</code>	false
<code>#{(10*10) ne 100}</code>	false

左边是运算符的使用方式，右边的是运算结果，关系运算也可以用来比较字符或字符串，按字典顺序来决定比较结果，例如：

运算式	结果
<code>#{'a' < 'b'}</code>	true
<code>#{'hip' > 'hit'}</code>	false
<code>#{'4' > 3}</code>	true

EL运算符的执行优先顺序与Java运算符对应，如果有疑虑的话，也可以使用括号()来自行决定先后顺序。

1.6 国际化信息

(注：本节原文是针对繁体中文进行说明的，在进行繁简转换的过程中，将原来针对繁体中文的描述全部改为简体中文了。)

JSF的国际化(Internationalization)信息处理是基于Java对国际化的支持，您可以在一个信息资源文件中统一管理信息资源，资源文件的名称是.properties，而内容是名称与值的配对，例如：

- messages.properties

```
titleText=JSF Demo
hintText=Please input your name and password
nameText=name
passText=password
commandText=Submit
```

资源文件名称由basename加上语言与地区来组成，例如：

```
* basename.properties
* basename_en.properties
* basename_zh_CN.properties
```

没有指定语言与地区的basename是预定义的资源文件名称，JSF会根据浏览器送来的Accept-Language header中的内容来决定该使用哪一个资源文件名称，例如：

```
Accept-Language: zh_CN, en-US, en
```

如果浏览器送来这些header，则预定义会使用简体中文，接着是美式英文，再来是英文语系，如果找不到对应的信息资源文件，则会使用预定义的信息资源文件。

由于信息资源文件必须是ISO-8859-1编码，所以对于非西方语系的处理，必须先将之转换为Java Unicode Escape格式，例如您可以先在信息资源文件中写下以下内容：

- messages_zh_CN.txt

```

titleText=JSF 示范
hintText=请输入名称与密码
nameText=名称
passText=密码
commandText=送出

```

然后使用JDK的工具程序native2ascii来转换，例如：

```
native2ascii -encoding GB2312 messages_zh_CN.txt messages_zh_CN.properties
```

转换后的内容会如下：

```

titleText=JSF\u793a\u8303
hintText=\u8bf7\u8f93\u5165\u540d\u79f0\u4e0e\u5bc6\u7801
nameText=\u540d\u79f0
passText=\u5bc6\u7801
commandText=\u9001\u51fa

```

接下来您可以使用<f:loadBundle>标签来指定载入信息资源，一个例子如下：

- index.jsp

```

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=UTF8"%>

<f:view>
<f:loadBundle basename="messages" var="msgs"/>

<html>
<head>
<title><h:outputText value="#{msgs.titleText}"/></title>
</head>
<body>

    <h:form>
        <h3><h:outputText value="#{msgs.hintText}"/></h3>
        <h:outputText value="#{msgs.nameText}"/>:
            <h:inputText value="#{user.name}"/><p>
        <h:outputText value="#{msgs.passText}"/>:
            <h:inputSecret value="#{user.password}"/><p>
        <h:commandButton value="#{msgs.commandText}"
            actionListener="#{user.verify}"
            action="#{user.outcome}"/>
    </h:form>

</body>
</html>

```



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

如此一来，如果您的浏览器预定义接受zh_CN语系的话，则页面上就可以显示中文，否则预定义将以英文显示，也就是messages.properties的内容，为了能显示多普通话系，我们设定网页编码为UTF8。

<f:view>可以设定locale属性，直接指定所要使用的语系，例如：

```
<f:view locale="zh_CN">
  <f:loadBundle basename="messages" var="msgs"/>
```

直接指定以上的话，则会使用简体中文来显示，JSF会根据<f:loadBundle>的basename属性加上<f:view>的locale属性来决定要使用哪一个信息资源文件，就上例而言，就是使用 messages_zh_CN.properties，如果设定为以下的话，就会使用messages_en.properties：

```
<f:view locale="en">
  <f:loadBundle basename="messages" var="msgs"/>
```

您也可以在faces-config.xml中设定语系，例如：

```
<faces-config>
  <application>
    <local-config>
      <default-locale>en</default-locale>
      <supported-locale>zh_CN</supported-locale>
    </local-config>
  </application>
  .....
</faces-config>
```

在<local-config>一定有一个<default-locale>，而<supported-locale>可以有好几个，这告诉JSF您的应用程序支持哪些语系。

当然，如果您可以提供一项选项让使用者选择自己的语系会是更好的方式，例如根据user这个Bean的locale属性来决定页面语系：

```
<f:view locale="#{user.locale}">
  <f:loadBundle basename="messages" var="msgs"/>
```

在页面中设定一个表单，可以让使用者选择语系，例如设定单选钮：

```
<h:selectOneRadio value="#{user.locale}">
  <f:selectItem itemValue="zh_CN"
    itemLabel="#{msgs.zh_CNText}"/>
  <f:selectItem itemValue="en"
    itemLabel="#{msgs.enText}"/>
</h:selectOneRadio>
```

2. Managed Beans

JSF 使用 Bean 来达到逻辑层与表现层分离的目的，Bean 的管理集中在配置文件中，您只要修改配置文件，就可以修改 Bean 之间的相依关系。

2.1 Backing Beans

JSF使用 JavaBean 来达到程序逻辑与视图分离的目的,在JSF中的Bean其角色是属于Backing Bean,又称之为Glue Bean,其作用是在真正的业务逻辑Bean及UI组件之间搭起桥梁,在Backing Bean中会呼叫业务逻辑Bean处理使用者的请求,或者是将业务处理结果放置其中,等待UI组件取出当中的值并显示结果给使用者。

JSF将Bean的管理集中在faces-config.xml中,一个例子如下:

```
....
<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>
    onlyfun.caterpillar.UserBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
....
```

这个例子我们在 第一个JSF程序 看过,<managed-bean-class>设定所要使用的Bean类,<managed-bean-name>设定之名称,可供我们在JSF页面上使用 Expression Language来取得或设定Bean的属性,例如:

```
<h:inputText value="#{user.name}"/>
```

<managed-bean-scope>设定Bean的存活范围,您可以设定为request、session 与 application,设定为request时,Bean的存活时间为请求阶段,设定为session则在使用者应用程序交互开始,直到关闭浏览器或显式的结束会话为止(例如登出程序),设定为application的话,则Bean会一直存活,直到应用程序关闭为止。

您还可以将存活范围设定为none,当设定为none时会在需要的时候生成一个新的Bean,例如您在一个method中想要生成一个临时的Bean,就可以将之设定为none。

在JSF页面上要取得Bean的属性,是使用 JSF表示语言(Expression Language),要注意的是,JSF表示语言是写成 #{expression},而 JSP表示语言是写成 \${expression},因为表示层可能是使用JSP,所以必须特别区分,另外要注意的是,JSF的标签上的属性设定时,只接受JSF表示语言。

2.2 Beans 的配置与设定

JSF预定义会读取faces-config.xml中关于Bean的定义,如果想要自行设置定义文件的名称,我们是在web.xml中提供javax.faces.CONFIG_FILES参数,例如:

```
<web-app>
  <context-param>
    <param-name>javax.faces.CONFIG_FILES</param-name>
    <param-value>/WEB-INF/beans.xml</param-value>
```

```
</context-param>
```

```
...
```

```
</web-app>
```

定义文件可以有多个，中间以 "," 区隔，例如：

```
/WEB-INF/navigation.xml,/WEB-INF/beans.xml
```

一个Bean最基本要定义Bean的名称、类与存活范围，例如：

```
....
```

```
<managed-bean>
```

```
  <managed-bean-name>user</managed-bean-name>
```

```
    <managed-bean-class>
```

```
      onlyfun.caterpillar.UserBean
```

```
    </managed-bean-class>
```

```
  <managed-bean-scope>session</managed-bean-scope>
```

```
</managed-bean>
```

```
....
```

如果要在其它类中取得Bean对象，则可以先取得 `javax.faces.context.FacesContext`，它代表了JSF目前的执行环境对象，接着尝试取得 `javax.faces.el.ValueBinding` 对象，从中取得指定的Bean对象，例如：

```
FacesContext context = FacesContext.getCurrentInstance();
```

```
ValueBinding binding =
```

```
  context.getApplication().createValueBinding("#{user}");
```

```
UserBean user = (UserBean) binding.getValue(context);
```

如果只是要尝试取得Bean的某个属性，则可以如下：

```
FacesContext context = FacesContext.getCurrentInstance();
```

```
ValueBinding binding =
```

```
  context.getApplication().createValueBinding(
      "#{user.name}");
```

```
String name = (String) binding.getValue(context);
```

如果有必要在启始Bean时，自动设置属性的初始值，则可以如下设定：

```
....
```

```
<managed-bean>
```

```
  <managed-bean-name>user</managed-bean-name>
```

```
    <managed-bean-class>
```

```
      onlyfun.caterpillar.UserBean
```

```
    </managed-bean-class>
```

```
  <managed-bean-scope>session</managed-bean-scope>
```

```
  <managed-property>
```

```
    <property-name>name</property-name>
```

```
    <value>caterpillar</value>
```

```
  </managed-property>
```

```
  <managed-property>
```

```
    <property-name>password</property-name>
```

```
    <value>123456</value>
```

```
    </managed-property>
</managed-bean>
....
```

如果要设定属性为 null 值，则可以使用<null-value/>标签，例如：

```
....
<managed-property>
  <property-name>name</property-name>
  <null-value/>
</managed-property>
<managed-property>
  <property-name>password</property-name>
  <null-value/>
</managed-property>
....
```

当然，您的属性不一定是字符串值，也许会是int、float、boolean等等类型，您可以设定<value> 值时指定这些值的字符串名称，JSF会尝试进行转换，例如设定为true时，会尝试使用Boolean.valueOf()方法转换为boolean的 true，以下是一些可能进行的转换：

类型	转换
short、int、long、float、double、byte，或相应的 Wrapper 类	尝试使用 Wrapper 的 valueOf() 进行转换，如果没有设置，则设为 0
boolean 或 Boolean	尝试使用 Boolean.valueOf() 进行转换，如果没有设置，则设为 false
char 或 Character	取设置的第一个字符，如果没有设置，则设为 0
String 或 Object	即设定的字符串值，如果没有设定，则为空字符串 new String("")

您也可以将其它产生的Bean设定给另一个Bean的属性，例如：

```
....
<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>
    onlyfun.caterpillar.UserBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>

<managed-bean>
  <managed-bean-name>other</managed-bean-name>
  <managed-bean-class>
    onlyfun.caterpillar.OtherBean
  </managed-bean-class>
</managed-bean>
```

```

        </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
    <managed-property>
        <property-name>user</property-name>
        <value>#{user}</value>
    </managed-property>
</managed-bean>
....

```

在上面的设定中，在OtherBean中的user属性，接受一个UserBean类型的对象，我们设定为前一个名称为user的UserBean对象。

2.3 Beans 上的 List, Map

如果您的Bean上有接受List或Map类型的属性，则您也可以在配置文件中直接设定这些属性的值，一个例子如下：

```

....
<managed-bean>
    <managed-bean-name>someBean</managed-bean-name>
    <managed-bean-class>
        onlyfun.caterpillar.SomeBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>

    <managed-property>
        <property-name>someProperty</property-name>
        <list-entries>
            <value-class>java.lang.Integer</value-class>
            <value>1</value>
            <value>2</value>
            <value>3</value>
        </list-entries>
    </managed-property>
</managed-bean>
....

```

这是一个设定接受List类型的属性，我们使用<list-entries>标签指定将设定一个List对象，其中<value-class>指定将存入List的类型，而<value>指定其值，如果是基本类型，则会尝试使用指定的 <value-class>来作Wrapper类。

设定Map的话，则是使用<map-entries>标签，例如：

```

....
<managed-bean>
    <managed-bean-name>someBean</managed-bean-name>
    <managed-bean-class>
        onlyfun.caterpillar.SomeBean
    </managed-bean-class>

```

```

<managed-bean-scope>session</managed-bean-scope>

<managed-property>
  <property-name>someProperty</property-name>
  <map-entries>
    <value-class>java.lang.Integer</value-class>
    <map-entry>
      <key>someKey1</key>
      <value>100</value>
    </map-entry>
    <map-entry>
      <key>someKey2</key>
      <value>200</value>
    </map-entry>
  </map-entries>
</managed-property>
</managed-bean>
....

```

由于Map对象是以key-value对的方式来存入，所以我们在每一个<map-entry>中使用<key>与<value>标签来分别指定。

您也可以直接像设定Bean一样，设定一个List或Map对象，例如在JSF附的范例如中，有这样的设定：

```

....
<managed-bean>
  <description>
    Special expense item types
  </description>
  <managed-bean-name>specialTypes</managed-bean-name>
  <managed-bean-class>
    java.util.TreeMap
  </managed-bean-class>
  <managed-bean-scope>application</managed-bean-scope>
  <map-entries>
    <value-class>java.lang.Integer</value-class>
    <map-entry>
      <key>Presentation Material</key>
      <value>100</value>
    </map-entry>
    <map-entry>
      <key>Software</key>
      <value>101</value>
    </map-entry>
    <map-entry>
      <key>Balloons</key>

```

```

        <value>102</value>
    </map-entry>
</map-entries>
</managed-bean>
....

```

而范例中另一个设定List的例子如下：

```

....
<managed-bean>
    <managed-bean-name>statusStrings</managed-bean-name>
    <managed-bean-class>
        java.util.ArrayList
    </managed-bean-class>
    <managed-bean-scope>request</managed-bean-scope>
    <list-entries>
        <null-value/>
        <value>Open</value>
        <value>Submitted</value>
        <value>Accepted</value>
        <value>Rejected</value>
    </list-entries>
</managed-bean>
....

```

3. 数据转换与验证

转换器 (Converter) 协助模型与视图之间的数据转换，验证器 (Validator) 协助进行语意检验 (Semantic Validation)。

3.1 标准转换器

Web应用程序与浏览器之间是使用HTTP进行沟通，所有传送的数据基本上都是字符串文字，而Java应用程序本身基本上则是对象，所以对象数据必须经由转换传送给浏览器，而浏览器送来的数据也必须转换为对象才能使用。

JSF定义了一系列标准的转换器 (Converter)，对于基本数据类型 (primitive type) 或是其Wrapper类，JSF会使用 javax.faces.Boolean、 javax.faces.Byte、 javax.faces.Character、 javax.faces.Double、 javax.faces.Float、 javax.faces.Integer、 javax.faces.Long、 javax.faces.Short 等自动进行转换，对于 BigDecimal、 BigInteger，则会使用 javax.faces.BigDecimal、 javax.faces.BigInteger 自动进行转换。

至于 DateTime、 Number，我们可以使用 <f:convertDateTime>、 <f:convertNumber> 标签进行转换，它们各自提供有一些简单的属性，可以让我们在转换时指定一些转换的格式细节。

来看个简单的例子，首先我们定义一个简单的Bean：

- UserBean.java

```
package onlyfun.caterpillar;

import java.util.Date;

public class UserBean {
    private Date date = new Date();

    public Date getDate() {
        return date;
    }

    public void setDate(Date date) {
        this.date = date;
    }
}
```

这个Bean的属性接受Date类型的参数，按理来说，接收到HTTP传来的数据中若有相关的日期信息，我们必须剖析这个信息，再转换为Date对象，然而我们可以使用JSF的标准转换器来协助这项工作，例如：

- index.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=GB2312"%>

<f:view>

<html>
<head>
<title>转换器示范</title>
</head>
<body>
    设定的日期是：
        <b>
            <h:outputText value="#{user.date}">
                <f:convertDateTime pattern="dd/MM/yyyy" />
            </h:outputText>
        </b>

    <h:form>
        <h:inputText id="dateField" value="#{user.date}">
```



```

        <f:convertDateTime pattern="dd/MM/yyyy"/>
    </h:inputText>
    <h:message for="dateField" style="color:red"/>
    <br>
    <h:commandButton value="送出" action="show"/>
</h:form>
</body>
</html>

</f:view>

```

在<f:convertDateTime>中，我们使用pattern指定日期的样式为dd/MM/yyyy，即「日/月/公元」格式，如果转换错误，则<h:message>可以显示错误信息，for属性参考至<h:inputText>的id属性，表示将有关dateField的错误信息显示出来。

假设faces-config.xml是这样定义的：

- faces-config.xml

```

<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <navigation-rule>
    <from-view-id>*/</from-view-id>
    <navigation-case>
      <from-outcome>show</from-outcome>
      <to-view-id>/pages/index.jsp</to-view-id>
    </navigation-case>
  </navigation-rule>

  <managed-bean>
    <managed-bean-name>user</managed-bean-name>
    <managed-bean-class>
      onlyfun.caterpillar.UserBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
  </managed-bean>
</faces-config>

```

首次连上页面时显示的画面如下：



如您所看到的，转换器自动依pattern设定的样式将Date对象格式化了，当您依格式输入数据并提交后，转换器也会自动将您输入的数据转换为Date对象，如果转换时发生错误，则会出现以下的信息：



<f:convertDateTime>标签还有几个可用的属性，您可以参考 Tag Library Documentation 的说明，而依照类似的方式，您也可以使用<f:convertNumber>来转换数值。

您还可以参考 Using the Standard Converters 这篇文章中有关于标准转换器的说明。

3.2 自定义转换器

除了使用标准的转换器之外，您还可以自行定制您的转换器，您可以实现 javax.faces.convert.Converter 接口，这个接口有两个要实现的方法：

```
public Object getAsObject(FacesContext context,
                          UIComponent component,
                          String str);
public String getAsString(FacesContext context,
                          UIComponent component,
                          Object obj);
```

简单的说，第一个方法会接收从客户端经由HTTP传来的字符串数据，您在第一个方法中将之转换为您的自定义对象，这个自定义对象将会自动设定给您指定的Bean对象；第二个方法就是将从您的Bean对象得到的对象转换为字符串，如此才能藉由HTTP传回给客户端。

直接以一个简单的例子来作说明，假设您有一个User类：

- User.java

```
package onlyfun.caterpillar;

public class User {
    private String firstName;
    private String lastName;

    public String getFirstName() {
```

```
        return firstName;
    }

    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
}
```

这个User类是我们转换器的目标对象，而您有一个GuestBean类：

- GuestBean.java

```
package onlyfun.caterpillar;

public class GuestBean {
    private User user;

    public void setUser(User user) {
        this.user = user;
    }

    public User getUser() {
        return user;
    }
}
```

这个Bean上的属性直接传回或接受User类型的参数，我们来实现一个简单的转换器，为HTTP字符串与User对象进行转换：

- UserConverter.java

```
package onlyfun.caterpillar;

import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.convert.Converter;
import javax.faces.convert.ConverterException;

public class UserConverter implements Converter {
```

```

public Object getAsObject(FacesContext context,
                          UIComponent component,
                          String str)
    throws ConverterException {
    String[] strs = str.split(",");

    User user = new User();

    try {
        user.setFirstName(strs[0]);
        user.setLastName(strs[1]);
    }
    catch(Exception e) {
        // 转换错误，简单的丢出例外
        throw new ConverterException();
    }

    return user;
}

public String getAsString(FacesContext context,
                          UIComponent component,
                          Object obj)
    throws ConverterException {
    String firstName = ((User) obj).getFirstName();
    String lastName = ((User) obj).getLastName();

    return firstName + "," + lastName;
}
}

```

实现完成这个转换器，我们要告诉JSF这件事，这是在faces-config.xml中完成注册：

- faces-config.xml

```

<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <navigation-rule>
    <from-view-id>*/</from-view-id>
    <navigation-case>
      <from-outcome>show</from-outcome>

```

```

        <to-view-id>/pages/index.jsp</to-view-id>
    </navigation-case>
</navigation-rule>

<converter>
    <converter-id>onlyfun.caterpillar.User</converter-id>
    <converter-class>
        onlyfun.caterpillar.UserConverter
    </converter-class>
</converter>

<managed-bean>
    <managed-bean-name>guest</managed-bean-name>
    <managed-bean-class>
        onlyfun.caterpillar.GuestBean
    </managed-bean-class>
    <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>

```

注册转换器时，需提供转换器标识（Converter ID）与转换器类，接下来要在JSF页面中使用转换器的话，就是指定所要使用的转换器标识，例如：

- index.jsp

```

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=GB2312"%>

<f:view>

<html>
<head>
<title>自定义转换器</title>
</head>
<body>

Guest 名称是 : <b>
    <h:outputText value="#{guest.user}"
        converter="onlyfun.caterpillar.User"/>
    </b>

    <h:form>
        <h:inputText id="userField"
            value="#{guest.user}"
            converter="onlyfun.caterpillar.User"/>
    </h:form>

```

```

        <h:message for="userField" style="color:red"/>
        <br>
        <h:commandButton value="送出" action="show"/>
    </h:form>
</body>
</html>

</f:view>

```

您也可以<f:converter>标签并使用converterId属性来指定转换器，例如：

```

<h:inputText id="userField" value="#{guest.user}">
    <f:converter converterId="onlyfun.caterpillar.User"/>
</h:inputText>

```

除了向JSF注册转换器之外，还有一个方式可以不用注册，就是直接在Bean上提供一个取得转换器的方法，例如：

- GuestBean.java

```

package onlyfun.caterpillar;

import javax.faces.convert.Converter;

public class GuestBean {
    private User user;
    private Converter converter = new UserConverter();

    public void setUser(User user) {
        this.user = user;
    }

    public User getUser() {
        return user;
    }

    public Converter getConverter() {
        return converter;
    }
}

```

之后可以直接结合 JSF Expression Language 来指定转换器：

```

<h:inputText id="userField"
    value="#{guest.user}"
    converter="#{guest.converter}"/>

```

3.3 标准验证器

当应用程序要求使用者输入数据时，必然考虑到使用者输入数据之正确性，对于使用者的输入必须进行检验，检验必要的两种验证是语法检验（Synatic Validation）与语意检验（Semantic Validation）。

语法检验是要检查使用者输入的数据是否合乎我们所要求的格式，最基本的就是检查使用者是否填入了栏目值，或是栏目值的长度、大小值等等是否符合要求。语意检验是在语法检验之后，在格式符合需求之后，我们进一步验证使用者输入的数据语意上是否正确，例如检查使用者的名称与密码是否匹配。

在 1.3 简单的导航（Navigation）中，我们对使用者名称与密码检查是否匹配，这是语意检验，我们可以使用JSF所提供的标准验证器，为其加入语法检验，例如：

- index.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=GB2312"%>
<html>
<head>
<title>验证器示范</title>
</head>
<body>
  <f:view>
    <h:messages layout="table" style="color:red"/>
    <h:form>
      <h3>请输入您的名称</h3>
      <h:outputText value="#{user.errMessage}"/><p>
      名称: <h:inputText value="#{user.name}"
                      required="true"/><p>
      密码: <h:inputSecret value="#{user.password}"
                      required="true">
        <f:validateLength minimum="6"/>
      </h:inputSecret><p>
      <h:commandButton value="送出"
                      action="#{user.verify}"/>
    </h:form>
  </f:view>
</body>
</html>
```

在<h:inputText>、</h:inputSecret>中，我们设定了required属性为true，这表示这个栏目一定要输入值，我们也在</h:inputSecret>设定了<f:validateLength>，并设定其minimum属性为6，这表示这个栏目最少需要6个字符。

这一次在错误信息的显示上，我们使用<h:messages>标签，当有验证错误发

生时，相关的错误信息会收集起来，使用<h:messages>标签可以一次将所有的错误信息显示出来。

下面是一个验证错误的信息显示：



Validation Error: Value is required.
Validation Error: Value is less than allowable minimum of '6'

請輸入您的名稱

名稱:

密碼:

JSF提供了三种标准验证器：`<f:validateDoubleRange>`、`<f:validateLongRange>`、`<f:validateLength>`，您可以分别查询它们的 Tag Library Documentation，了解他们有哪些属性可以使用，或者是参考 Using the Standard Validators 这篇文章中有关于标准验证器的说明。

3.4 自定义验证器

您可以自定义自己的验证器，所需要的是实现 `javax.faces.validator.Validator` 接口，例如我们实现一个简单的密码验证器，检查字符长度，以及密码中是否包括字符与数字：

- PasswordValidator.java

```
package onlyfun.caterpillar;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class PasswordValidator implements Validator {
    public void validate(FacesContext context,
                        UIComponent component,
                        Object obj)
        throws ValidatorException {
        String password = (String) obj;

        if(password.length() < 6) {
            FacesMessage message = new FacesMessage(
```



```

        FacesMessage.SEVERITY_ERROR,
        "字符长度小于 6",
        "字符长度不得小于 6");
    throw new ValidatorException(message);
}

if(!password.matches("[0-9]+")) {
    FacesMessage message = new FacesMessage(
        FacesMessage.SEVERITY_ERROR,
        "密码必须包括字符与数字",
        "密码必须是字符加数字所组成");
    throw new ValidatorException(message);
}
}
}

```

您要实现 `javax.faces.validator.Validator` 接口中的 `validate()` 方法，如果验证错误，则丢出一个 `ValidatorException`，它接受一个 `FacesMessage` 对象，这个对象接受三个参数，分别表示信息的严重程度（INFO、WARN、ERROR、FATAL）、信息概述与详细信息内容，这些信息将可以使用 `<h:messages>` 或 `<h:message>` 标签显示在页面上。

接下来要在 `faces-config.xml` 中注册验证器的标识（Validator ID），要加入以下内容：

- `faces-config.xml`

```

<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
....
    <validator>
        <validator-id>
            onlyfun.caterpillar.Password
        </validator-id>
        <validator-class>
            onlyfun.caterpillar.PasswordValidator
        </validator-class>
    </validator>
....
</faces-config>

```

要使用自定义的验证器，我们可以使用 `<f:validator>` 标签并设定 `validatorId` 属性，例如：

```
....
```

```
<h:inputSecret value="#{user.password}" required="true">
    <f:validator validatorId="onlyfun.caterpillar.Password"/>
</h:inputSecret><p>
....
```

您也可以让Bean自行负责验证的工作，可以在Bean上提供一个验证方法，这个方法没有传回值，并可以接收FacesContext、UIComponent、Object三个参数，例如：

- UserBean.java

```
package onlyfun.caterpillar;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.ValidatorException;

public class UserBean {
    ....

    public void validate(FacesContext context,
                        UIComponent component,
                        Object obj)
        throws ValidatorException {
        String password = (String) obj;

        if(password.length() < 6) {
            FacesMessage message = new FacesMessage(
                FacesMessage.SEVERITY_ERROR,
                "字符长度小于 6",
                "字符长度不得小于 6");
            throw new ValidatorException(message);
        }

        if(!password.matches("^[0-9]+")) {
            FacesMessage message = new FacesMessage(
                FacesMessage.SEVERITY_ERROR,
                "密码必须包括字符与数字",
                "密码必须是字符加数字所组成");
            throw new ValidatorException(message);
        }
    }
}
```

接着可以在页面下如下使用验证器：

```
.....
```

```
<h:inputSecret value="#{user.password}"
               required="true"
               validator="#{user.validate}"/>
....
```

3.5 错误信息处理

在使用标准转换器或验证器时，当发生错误时，会有一些预定义的错误信息显示，这些信息可以使用<h:messages>或<h:message>标签来显示出来，而这些预定义的错误信息也是可以修改的，您所要作的是提供一个信息资源文件，例如：

- messages.properties

```
javax.faces.component.UIInput.CONVERSION=Format Error.
javax.faces.component.UIInput.REQUIRED=Please input your data.
....
```

javax.faces.component.UIInput.CONVERSION是用来设定当转换器发现错误时显示的信息，而javax.faces.component.UIInput.REQUIRED是在标签设定了required为true，而使用者没有在栏目输入时显示的错误信息。

您要在faces-config.xml中告诉JSF您使用的信息文件名称，例如：

- faces-config.xml

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
  <application>
    <local-config>
      <default-locale>en</default-locale>
      <supported-locale>zh_TW</supported-locale>
    </local-config>
    <message-bundle>messages</message-bundle>
  </application>
  ....
</faces-config>
```

在这边我们设定了信息文件的名称为messages_xx_YY.properties，其中xx_YY是根据您的Locale来决定，转换器或验证器的错误信息如果有设定的话，就使用设定值，如果没有设定的话，就使用预定义值。

验证器错误信息，除了上面的javax.faces.component.UIInput.REQUIRED

之外，还有以下几个：

信息标识	预定义信息	用于
<code>javax.faces.validator.NOT_IN_RANGE</code>	Validation Error: Specified attribute is not between the expected values of {0} and {1}.	<code>DoubleRangeValidator</code> 与 <code>LongRangeValidator</code> , {0} 与 {1} 分别代表 minimum 与 maximum 所设定的属性
<code>javax.faces.validator.DoubleRangeValidator.MAXIMUM</code> 、 <code>javax.faces.validator.LongRangeValidator.MAXIMUM</code>	Validation Error: Value is greater than allowable maximum of '{0}'.	<code>DoubleRangeValidator</code> 或 <code>LongRangeValidator</code> , {0} 表示 maximum 属性
<code>javax.faces.validator.DoubleRangeValidator.MINIMUM</code> 、 <code>javax.faces.validator.LongRangeValidator.MINIMUM</code>	Validation Error: Value is less than allowable minimum of '{0}'.	<code>DoubleRangeValidator</code> 或 <code>LongRangeValidator</code> , {0} 代表 minimum 属性
<code>javax.faces.validator.DoubleRangeValidator.TYPE</code> 、 <code>javax.faces.validator.LongRangeValidator.TYPE</code>	Validation Error: Value is not of the correct type.	<code>DoubleRangeValidator</code> 或 <code>LongRangeValidator</code>
<code>javax.faces.validator.LengthValidator.MAXIMUM</code>	Validation Error: Value is greater than allowable maximum of "{0}".	<code>LengthValidator</code> , {0}代表 maximum
<code>javax.faces.validator.LengthValidator.MINIMUM</code>	Validation Error: Value is less than allowable minimum of "{0}".	<code>LengthValidator</code> , {0}代表 minimum 属性

在您提供自定义信息的时候，也可以提供{0}或{1}来设定显示相对的属性值，以提供详细正确的错误提示信息。

信息的显示有概述信息与详述信息，如果是详述信息，则在标识上加上 "_detail"，例如：

```

javax.faces.component.UIInput.CONVERSION=Error.
javax.faces.component.UIInput.CONVERSION_detail= Detail Error.
....

```

除了在信息资源文件中提供信息，您也可以在程序中使用FacesMessage来提供信息，例如在 3.4自定义验证器 中我们就这么用过：

```

....
if(password.length() < 6) {
    FacesMessage message = new FacesMessage(
        FacesMessage.SEVERITY_ERROR,
        "字符长度小于 6",
        "字符长度不得小于 6");
    throw new ValidatorException(message);
}

```

```

    }
    ....

```

最好的方法是在信息资源文件中提供信息，这么一来如果我们要修改信息，就只要修改信息资源文件的内容，而不用修改程序，来看一个简单的例子，假设我们的信息资源文件中有以下的内容：

```

onlyfun.caterpillar.message1=This is message1.
onlyfun.caterpillar.message2=This is message2 with \{0} and \{1}.

```

则我们可以在程序中取得信息资源文件的内容，例如：

```

package onlyfun.caterpillar;

import java.util.Locale;
import java.util.ResourceBundle;
import javax.faces.context.FacesContext;
import javax.faces.component.UIComponent;
import javax.faces.application.Application;
import javax.faces.application.FacesMessage;

....

public void xxxMethod(FacesContext context,
                      UIComponent component,
                      Object obj) {
    // 取得应用程序代表对象
    Application application = context.getApplication();
    // 取得信息文件主名称
    String messageFileName =
        application.getMessageBundle();
    // 取得当前 Locale 对象
    Locale locale = context.getViewRoot().getLocale();
    // 取得信息绑定 ResourceBundle 对象
    ResourceBundle rsBundle =
        ResourceBundle.getBundle(messageFileName, locale);

    String message = rsBundle.getString(
        "onlyfun.caterpillar.message1");
    FacesMessage facesMessage = new FacesMessage(
        FacesMessage.SEVERITY_FATAL, message, message);
    ....
}
....
....

```

接下来您可以将FacesMessage对象填入ValidatorException或ConverterException后再丢出，FacesMessage建构时所使用的三个参数是严重程度、概述信息与详述信息，严重程度有SEVERITY_FATAL、SEVERITY_ERROR、SEVERITY_WARN与SEVERITY_INFO四种。

如果需要在信息资源文件中设定{0}、{1}等参数，则可以如下：

```
....
String message = rsBundle.getString(
    "onlyfun.caterpillar.message2");
Object[] params = {"param1", "param2"};
message = java.text.MessageFormat.format(message, params);

FacesMessage facesMessage = new FacesMessage(
    FacesMessage.SEVERITY_FATAL, message, message);
....
```

如此一来，在显示信息时，onlyfun.caterpillar.message2的{0}与{1}的位置就会被"param1"与"param2"所取代。

3.6 自定义转换，验证标签

在自定义验证器中，我们的验证器只能验证一种pattern（.+[0-9]+），我们希望可以在JSF页面上自定义匹配的pattern，然而由于我们使用<f:validator>这个通用的验证器标签，为了要能提供pattern属性，我们可以使用<f:attribute>标签来设置，例如：

```
....
<h:inputSecret value="#{user.password}" required="true">
    <f:validator validatorId="onlyfun.caterpillar.Password"/>
    <f:attribute name="pattern" value=".+[0-9]+" />
</h:inputSecret><p>
....
```

使用<f:attribute>标签来设定属性，接着我们可以如下取得所设定的属性：

```
....
public void validate(FacesContext context,
    UIComponent component,
    Object obj)
    throws ValidatorException {
    ....
    String pattern = (String)
        component.getAttributes().get("pattern");
    ....
}
....
```

您也可以开发自己的一组验证标签，并提供相关属性设定，这需要了解JSP Tag Library的编写，所以请您先参考 JSP/Servlet 中有关于JSP Tag Library的介绍。

要开发验证器转用标签，您可以直接继承 javax.faces.webapp.ValidatorTag，这个类可以帮您处理大部份的细节，您所需要的，就是重新定义它的createValidator()方法，我们以改写 自定义验证器中的PasswordValidator为例：

- PasswordValidator.java

```
package onlyfun.caterpillar;

import javax.faces.application.FacesMessage;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.validator.ValidatorException;

public class PasswordValidator implements Validator {
    private String pattern;

    public void setPattern(String pattern) {
        this.pattern = pattern;
    }

    public void validate(FacesContext context,
                        UIComponent component,
                        Object obj)
        throws ValidatorException {
        String password = (String) obj;

        if(password.length() < 6) {
            FacesMessage message = new FacesMessage(
                FacesMessage.SEVERITY_ERROR,
                "字符长度小于 6", "字符长度不得小于 6");
            throw new ValidatorException(message);
        }

        if(pattern != null && !password.matches(pattern)) {
            FacesMessage message = new FacesMessage(
                FacesMessage.SEVERITY_ERROR,
                "密码必须包括字符与数字",
                "密码必须是字符加数字所组成");
            throw new ValidatorException(message);
        }
    }
}
```

主要的差别是我们提供了pattern属性，在validate()方法中进行验证时，是根据我们所设定的pattern属性，接着我们继承 javax.faces.webapp.ValidatorTag来编写自己的验证标签：

- PasswordValidatorTag.java

```

package onlyfun.caterpillar;

import javax.faces.application.Application;
import javax.faces.context.FacesContext;
import javax.faces.validator.Validator;
import javax.faces.webapp.ValidatorTag;

public class PasswordValidatorTag extends ValidatorTag {
    private String pattern;

    public void setPattern(String pattern) {
        this.pattern = pattern;
    }

    protected Validator createValidator() {
        Application application =
            FacesContext.getCurrentInstance().
                getApplication();
        PasswordValidator validator =
            (PasswordValidator) application.createValidator(
                "onlyfun.caterpillar.Password");
        validator.setPattern(pattern);
        return validator;
    }
}

```

`application.createValidator()` 方法建立验证器对象时，是根据在 `faces-config.xml` 中注册验证器的标识（`Validator ID`）：

- `faces-config.xml`

```

<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
....
    <validator>
        <validator-id>
            onlyfun.caterpillar.Password
        </validator-id>
        <validator-class>
            onlyfun.caterpillar.PasswordValidator
        </validator-class>
    </validator>

```



```
....
```

```
</faces-config>
```

剩下的工作，就是布署tld描述文件了，我们简单的定义一下：

- taglib.tld

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

```
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    web-jsptaglibrary_2_0.xsd"
  version="2.0">
```

```
<description>PasswordValidator Tag</description>
<tlib-version>1.0</tlib-version>
<jsp-version>2.0</jsp-version>
<short-name>co</short-name>
<uri>http://caterpillar.onlyfun.net</uri>
```

```
<tag>
  <description>PasswordValidator</description>
  <name>passwordValidator</name>
  <tag-class>
    onlyfun.caterpillar.PasswordValidatorTag
  </tag-class>
  <body-content>empty</body-content>
  <attribute>
    <name>pattern</name>
    <required>true</required>
    <rtexprvalue>false</rtexprvalue>
  </attribute>
</tag>
```

```
</taglib>
```

而我们的index.jsp改写如下：

- index.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@ taglib uri="/WEB-INF/taglib.tld" prefix="co" %>
<%@page contentType="text/html; charset=GB2312"%>
<html>
<head>
```

```

<title>验证器示范</title>
</head>
<body>
  <f:view>
    <h:messages layout="table" style="color:red"/>
    <h:form>
      <h3>请输入您的名称</h3>
      <h:outputText value="#{user.errMessage}"/><p>
      名称: <h:inputText value="#{user.name}"
        required="true"/><p>
      密码: <h:inputSecret value="#{user.password}"
        required="true">
        <co:passwordValidator pattern=".[0-9]+"/>
      </h:inputSecret> <p>
      <h:commandButton value="送出"
        action="#{user.verify}"/>
    </h:form>
  </f:view>
</body>
</html>

```

主要的差别是，我们使用了自己的验证器标签：

```
<co:passwordValidator pattern=".[0-9]+"/>
```

如果要自定义转换器标签，方法也是类似，您要作的是继承 `javax.faces.webapp.ConverterTag`，并重新定义其 `createConverter()` 方法。

4. 事件处理

JSF的事件模型提供一个近似的桌面GUI事件方式，让熟悉GUI设计的人员也能快速上手Web程序设计。

4.1 动作事件

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 1.2简单的导航 中，我们根据动作方法 (action method) 的结果来决定要导向的网页，一个按钮绑定一个方法，这样的作法实际上即使JSF所提供的简化的事件处理程序，在按钮上使用action绑定一个动作方法 (action method)，实际上JSF会为其自动产生一个「预定义的ActionListener」来处理事件，并根据其传回值来决定导向的页面。

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，例如：

- UserBean.java

```
package onlyfun.caterpillar;

import javax.faces.event.ActionEvent;

public class UserBean {
    private String name;
    private String password;
    private String errMessage;
    private String outcome;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

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

    public String getPassword() {
        return password;
    }

    public void setErrMessage(String errMessage) {
        this.errMessage = errMessage;
    }

    public String getErrMessage() {
        return errMessage;
    }

    public void verify(ActionEvent e) {
        if(!name.equals("justin") ||
            !password.equals("123456")) {
            errMessage = "名称或密码错误" + e.getSource();
            outcome = "failure";
        }
        else {
```

```

        outcome = "success";
    }
}

public String outcome() {
    return outcome;
}
}

```

在上例中，我们让verify方法接收一个ActionEvent对象，当使用者按下按钮，会自动产生ActionEvent对象代表事件来源，我们故意在错误信息之后如上事件来源的字符串描述，这样就可以在显示错误信息时一并显示事件来源描述。

为了提供ActionEvent的存取能力，您的index.jsp可以改写如下：

- index.jsp

```

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=Big5"%>
<html>
<head>
<title>第一个 JSF 程序</title>
</head>
<body>
    <f:view>
        <h:form>
            <h3>请输入您的名称</h3>
            <h:outputText value="#{user.errMessage}"/><p>
            名称: <h:inputText value="#{user.name}"/><p>
            密码: <h:inputSecret value="#{user.password}"/><p>
            <h:commandButton value="送出"
                            actionListener="#{user.verify}"
                            action="#{user.outcome}"/>
        </h:form>
    </f:view>
</body>
</html>

```

主要改变的是按钮上使用了actionListener属性，这种方法可以使用一个ActionListener，JSF会先检查是否有指定的actionListener，然后再检查是否指定了动作方法并产生预定义的ActionListener，并根据其传回值导航页面。

如果您要注册多个ActionListener，例如当使用者按下按钮时，顺便在记录文件中增加一些记录信息，您可以实现javax.faces.event.ActionListener，例如：

- LogHandler.java

```
package onlyfun.caterpillar;

import javax.faces.event.ActionListener;
....

public class LogHandler implements ActionListener {
    public void processAction(ActionEvent e) {
        // 处理 Log
    }
}
```

- VerifyHandler.java

```
package onlyfun.caterpillar;

import javax.faces.event.ActionListener;
....

public class VerifyHandler implements ActionListener {
    public void processAction(ActionEvent e) {
        // 处理验证
    }
}
```

这么一来，您就可以使用<f:actionListener>标签向组件注册事件，例如：

```
<h:commandButton value="送出" action="#{user.outcome}">
    <f:actionListener type="onlyfun.caterpillar.LogHandler"/>
    <f:actionListener type="onlyfun.caterpillar.VerifyHandler"/>
</h:commandButton>
```

<f:actionListener>会自动产生type所指定的对象，并呼叫组件的addActionListener()方法注册Listener。

4.2 即时事件

所谓的即时事件（Immediate Events），是指JSF视图组件在取得请求中该取得的值之后，即立即处理指定的事件，而不再进行后续的转换器处理、验证器处理、更新模型值等流程。

在JSF的事件模型中之所以会有所谓即时事件，是因为Web应用程序的先天特性不同于GUI程序，所以JSF的事件方式与GUI程序的事件方式仍有相当程度的不同，一个最基本的问题正因为HTTP无状态的特性，使得Web应用程序天生就无法直接唤起服务器端的特定对象。

所有的对象唤起都是在服务器端执行的，至于该唤起什么对象，则是依一个基本的流程：

➤ 重建视图（Restore View）

依客户端传来的session数据或服务器端上的session数据，重建JSF视图组

件。

- **套用请求值 (Apply Request Values)**
JSF视图组件各自获得请求中的属于自己的值，包括旧的值与新的值。
- **执行验证 (Process Validations)**
转换为对象并进行验证。
- **更新模型值 (Update Model Values)**
更新Bean或相关的模型值。
- **唤起应用程序 (Invoke Application)**
执行应用程序相关逻辑。
- **绘制响应页面 (Render Response)**
对先前的请求处理完之后，产生页面以反应客户端执行结果。

对于动作事件 (Action Event) 来说，组件的动作事件是在套用请求值阶段就生成ActionEvent对象了，但相关的事件处理并不是马上进行，ActionEvent会先被排入队列，然后必须再通过验证、更新方式值阶段，之后才处理队列中的事件。

这样的流程对于按下按钮然后执行后端的应用程序来说不成问题，但有些事件并不需要这样的流程，例如只影响页面的事件。

举个例子来说，在表单中可能有使用者名称、密码等栏目，并提供有一个地区选项按钮，使用者可以在不填写名称、密码的情况下，就按下地区选项按钮，如果依照正常的流程，则会进行验证、更新模型值、唤起应用程序等流程，但显然的，使用者名称与密码是空白的，这会引起不必要的错误。

您可以设定组件的事件在套用请求值之后立即被处理，并跳过后续的阶段，直接进行页面绘制以响应请求，对于JSF的input与command组件，都有一个immediate属性可以设定，只要将其设定为true，则指定的事件就成为即时事件。

一个例子如下：

- index.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=UTF8"%>

<f:view locale="#{user.locale}">
<f:loadBundle basename="messages" var="msgs"/>

<html>
<head>
<title><h:outputText value="#{msgs.titleText}"/></title>
</head>
<body>

    <h:form>
        <h3><h:outputText value="#{msgs.hintText}"/></h3>
        <h:outputText value="#{msgs.nameText}"/>:
            <h:inputText value="#{user.name}"/><p>
```

```

        <h:outputText value="#{msgs.passText}"/>:
            <h:inputSecret value="#{user.password}"/><p>
        <h:commandButton value="#{msgs.commandText}"
            action="#{user.verify}"/>
        <h:commandButton value="#{msgs.Text}"
            immediate="true"
            actionListener="#{user.changeLocale}"/>

    </h:form>

</body>
</html>

</f:view>

```

这是一个可以让使用者决定使用语系的示范，最后一个commandButton组件被设定了immediate属性，当按下这个按钮后，JSF套用请求值之后会立即处理指定的actionListener，而不再进行验证、更新模型值，简单的说，就这个程序来说，您在输入栏目与密码栏目中填入的值，不会影响您的user.name与user.password。

基于范例的完整起见，我们列出这个程序Bean对象及faces-config.xml：

- UserBean.java

```

package onlyfun.caterpillar;

import javax.faces.event.ActionEvent;

public class UserBean {
    private String locale = "en";
    private String name;
    private String password;
    private String errMessage;

    public void changeLocale(ActionEvent e) {
        if(locale.equals("en"))
            locale = "zh_TW";
        else
            locale = "en";
    }

    public String getLocale() {
        if (locale == null) {
            locale = "en";
        }
        return locale;
    }
}

```

```
public void setName(String name) {
    this.name = name;
}

public String getName() {
    return name;
}

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

public String getPassword() {
    return password;
}

public void setErrorMessage(String errorMessage) {
    this.errorMessage = errorMessage;
}

public String getErrorMessage() {
    return errorMessage;
}

public String verify() {
    if(!name.equals("justin") ||
        !password.equals("123456")) {
        errorMessage = "名称或密码错误";
        return "failure";
    }
    else {
        return "success";
    }
}
}
```

- faces-config.xml

```
<?xml version="1.0"?>
<!DOCTYPE faces-config PUBLIC
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">

<faces-config>
```



```

<navigation-rule>
  <from-view-id>/pages/index.jsp</from-view-id>
  <navigation-case>
    <from-outcome>success</from-outcome>
    <to-view-id>/pages/welcome.jsp</to-view-id>
  </navigation-case>
  <navigation-case>
    <from-outcome>failure</from-outcome>
    <to-view-id>/pages/index.jsp</to-view-id>
  </navigation-case>
</navigation-rule>

<managed-bean>
  <managed-bean-name>user</managed-bean-name>
  <managed-bean-class>
    onlyfun.caterpillar.UserBean
  </managed-bean-class>
  <managed-bean-scope>session</managed-bean-scope>
</managed-bean>
</faces-config>

```

信息资源文件的内容则是如下：

- messages_en.properties

```

titleText=JSF Demo
hintText=Please input your name and password
nameText=name
passText=password
commandText=Submit
Text=\u4e2d\u6587

```

Text中设定的是「中文」转换为Java Unicode Escape格式的结果，另一个信息资源文件的内容则是英文信息的翻译而已，其转换为Java Unicode Escape格式结果如下：

- messages_zh_TW.properties

```

titleText=JSF\u793a\u7bc4
hintText=\u8acb\u8f38\u5165\u540d\u7a31\u8207\u5bc6\u78bc
nameText=\u540d\u7a31
passText=\u5bc6\u78bc
commandText=\u9001\u51fa
Text=English

```

welcome.jsp就请自行设计了，程序的画面如下：

4.3 值变事件

如果使用者改变了JSF输入组件的值后提交表单，就会发生值变事件（Value Change Event），这会丢出一个`javax.faces.event.ValueChangeEvent`对象，如果您想要处理这个事件，有两种方式，一是直接设定JSF输入组件的`valueChangeListener`属性，例如：

```
<h:selectOneMenu value="#{user.locale}"
    onchange="this.form.submit();"
    valueChangeListener="#{user.changeLocale}">

    <f:selectItem itemValue="zh_CN" itemLabel="Chinese"/>
    <f:selectItem itemValue="en" itemLabel="English"/>
</h:selectOneMenu>
```

为了模拟GUI中选择了选单项目之后就立即发生反应，我们在`onchange`属性中使用了JavaScript，其作用是在选项项目发生改变之后，立即提交表单，而不用按下提交按钮；而`valueChangeListener`属性所绑定的`user.changeLocale`方法必须接受`ValueChangeEvent`对象，例如：

- UserBean.java

```
package onlyfun.caterpillar;

import javax.faces.event.ValueChangeEvent;

public class UserBean {
```

```
private String locale = "en";
private String name;
private String password;
private String errMessage;

public void changeLocale(ValueChangeEvent event) {
    if(locale.equals("en"))
        locale = "zh_CN";
    else
        locale = "en";
}

public void setLocale(String locale) {
    this.locale = locale;
}

public String getLocale() {
    if (locale == null) {
        locale = "en";
    }
    return locale;
}

public void setName(String name) {
    this.name = name;
}

public String getName() {
    return name;
}

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

public String getPassword() {
    return password;
}

public void setErrorMessage(String errMessage) {
    this.errMessage = errMessage;
}

public String getErrorMessage() {
```

```

        return errMsg;
    }

    public String verify() {
        if(!name.equals("justin") ||
            !password.equals("123456")) {
            errMsg = "名称或密码错误";
            return "failure";
        }
        else {
            return "success";
        }
    }
}

```

另一个方法是实现 `javax.faces.event.ValueChangeListener` 接口，并定义其 `processValueChange()` 方法，例如：

- `SomeListener.java`

```

package onlyfun.caterpillar;

....
public class SomeListener implements ValueChangeListener {
    public void processValueChange(ValueChangeEvent event) {
        ....
    }
    ....
}

```

然后在JSF页面上使用 `<f:valueChangeListener>` 标签，并设定其 `type` 属性，例如：

```

<h:selectOneMenu value="#{user.locale}"
    onchange="this.form.submit();">
    <f:valueChangeListener
        type="onlyfun.caterpillar.SomeListener"/>
    <f:selectItem itemValue="zh_CN" itemLabel="Chinese"/>
    <f:selectItem itemValue="en" itemLabel="English"/>
</h:selectOneMenu>

```

下面这个页面是对 4.2 即时事件 中的范例程序作一个修改，将语言选项改以下拉式选单的选择方式呈现，这必须配合上面提供的 `UserBean` 类来使用：

- `index.jsp`

```

<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>
<%@page contentType="text/html; charset=UTF8"%>

```

```

<f:view locale="#{user.locale}">
<f:loadBundle basename="messages" var="msgs"/>

<html>
<head>
<title><h:outputText value="#{msgs.titleText}"/></title>
</head>
<body>

    <h:form>
        <h:selectOneMenu value="#{user.locale}"
            immediate="true"
            onchange="this.form.submit();"
            valueChangeListener="#{user.changeLocale}">

            <f:selectItem itemValue="zh_CN"
                itemLabel="Chinese"/>
            <f:selectItem itemValue="en"
                itemLabel="English"/>
        </h:selectOneMenu>

        <h3><h:outputText value="#{msgs.hintText}"/></h3>
        <h:outputText value="#{msgs.nameText}"/>:
            <h:inputText value="#{user.name}"/><p>
        <h:outputText value="#{msgs.passText}"/>:
            <h:inputSecret value="#{user.password}"/><p>
        <h:commandButton value="#{msgs.commandText}"
            action="#{user.verify}"/>
    </h:form>

</body>
</html>

</f:view>

```

4.4 Phase 事件

在 4.2 即时事件 中我们提到，JSF 的请求执行到反应，完整的过程会经过六个阶段：

- **重建视图 (Restore View)**
依客户端传来的 session 数据或服务器端上的 session 数据，重建 JSF 视图组件。
- **套用请求值 (Apply Request Values)**
JSF 视图组件各自获得请求中的属于自己的值，包括旧的值与新的值。

- **执行验证 (Process Validations)**
转换为对象并进行验证。
- **更新模型值 (Update Model Values)**
更新Bean或相关的模型值。
- **唤起应用程序 (Invoke Application)**
执行应用程序相关逻辑。
- **绘制响应页面 (Render Response)**
对先前的请求处理完之后，产生页面以反应客户端执行结果。
在每个阶段的前后会引发 `javax.faces.event.PhaseEvent`，如果您想尝试在每个阶段的前后捕捉这个事件，以进行一些处理，则可以实现 `javax.faces.event.PhaseListener`，并向 `javax.faces.lifecycle.Lifecycle` 登记这个 `Listener`，以在适当的时候通知事件的发生。

`PhaseListener` 有三个必须实现的方法 `getPhaseId()`、`beforePhase()` 与 `afterPhase()`，其中 `getPhaseId()` 传回一个 `PhaseId` 对象，代表 `Listener` 想要被通知的时机，可以设定的时机有：

```
PhaseId.RESTORE_VIEW
PhaseId.APPLY_REQUEST_VALUES
PhaseId.PROCESS_VALIDATIONS
PhaseId.UPDATE_MODEL_VALUES
PhaseId.INVOKE_APPLICATION
PhaseId.RENDER_RESPONSE
PhaseId.ANY_PHASE
```

其中 `PhaseId.ANY_PHASE` 指的是任何的阶段转换时，就进行通知；您可以在 `beforePhase()` 与 `afterPhase()` 中编写阶段前后编写分别想要处理的动作，例如下面这个简单的类会列出每个阶段的名称：

- `ShowPhaseListener.java`

```
package onlyfun.caterpillar;

import javax.faces.event.PhaseEvent;
import javax.faces.event.PhaseId;
import javax.faces.event.PhaseListener;

public class ShowPhaseListener implements PhaseListener {

    public void beforePhase(PhaseEvent event) {
        String phaseName = event.getPhaseId().toString();
        System.out.println("Before " + phaseName);
    }

    public void afterPhase(PhaseEvent event) {
        String phaseName = event.getPhaseId().toString();
        System.out.println("After " + phaseName);
    }
}
```

```
public PhaseId getPhaseId() {  
    return PhaseId.ANY_PHASE;  
}  
}
```

编写好PhaseListener后,我们可以在faces-config.xml中向Lifecycle进行注册:

- faces-config.xml

```
<?xml version="1.0"?>  
<!DOCTYPE faces-config PUBLIC  
"-//Sun Microsystems, Inc.//DTD JavaServer Faces Config 1.0//EN"  
"http://java.sun.com/dtd/web-facesconfig_1_0.dtd">  
  
<faces-config>  
  <lifecycle>  
    <phase-listener>  
      onlyfun.caterpillar.ShowPhaseListener  
    </phase-listener>  
  </lifecycle>  
  .....  
</faces-config>
```

您可以使用这个简单的类,看看在请求任一个JSF画面时所显示的内容,借此了解JSF每个阶段的流程变化。