

# 前言

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。

- 入门
  - 藉由以下的几个主题，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。
    - 简介 JSF
    - 第一个 JSF 程序
    - 简单的导航 Navigation
    - 导航规则设置
    - JSF Expression Language
    - 国际化讯息
- Managed Beans
  - JSF 使用 Bean 来达到逻辑层与表现层分离的目的，Bean 的管理集中在组态档案中，您只要修改组态档案，就可以修改 Bean 之间的相依关系。
  - Backing Beans
  - Beans 的组态与设定
  - Beans 上的 List, Map
- 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）。
  - 标准转换器
  - 自订转换器
  - 标准验证器
  - 自订验证器
  - 错误讯息处理
  - 自订转换，验证标签
- 事件处理
  - JSF 的事件模型提供一个近似的桌面 GUI 事件模式，让熟悉 GUI 设计的人员也能快速上手 Web 程序设计。
  - 动作事件
  - 实时事件
  - 值变事件
  - Phase 事件

JSF 标签

网页设计人员要作的就是了解 JSF 的标签的使用方式，这就像是学习进阶的 HTML 标签，另一件事就是与程序设计人员沟通好各个 Bean 的名称绑定。

- 标签入门
  - 卷标的相关属性查询，您可以参考 [Tag Library Documentation](#)，这边的介绍只是一些简单的入门实例。
  - o 简介 JSF 标准标签
  - o 输出类标签
  - o 输入类标签
  - o 命令类标签
  - o 选择类标签 一
  - o 选择类标签 二
  - o 其它标签
- 表格处理
  - 对于必须使用表格方式呈现的数据，JSF 的 `<h:dataTable>` 卷标协助您进行动态表格数据的输出。
  - o 简单的表格
  - o 表头，表尾
  - o `TableModel` 类别

#### 自订组件

JSF 让您可以自订组件，每个组件都是可替换的，这使得组件在搭配时更有弹性，但相对的却使开发组件的过程复杂的多，这边对自订 JSF 组件只是个入门砖，更多有关自订组件的细节可得要专书来说明。

- JSF 生命周期与组件概述
  - 要开发 JSF 组件，您需要更深入了解 JSF 的一些处理细节，包括了 JSF 生命周期以及 JSF 框架。
  - o JSF 生命周期
  - o 概述自订组件
- 简单实例
  - 在不考虑组件有子组件的情况下，这边以实际的一个例子来说明开发组件的过程，至于考虑子组件的情况请参考专书介绍。
  - o 编码，解码
  - o 组件卷标
  - o 使用自订组件
  - o 自订 `Renderer`

# 入门

## 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，其执行绪安全问题以及资料验证、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卷标的方式呈现，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，为了促进产能的整合开发环境（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](#)<sup>■</sup> 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 实作成果。

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 这类的卷标，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 也允许您自订标签。

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 参考实作 (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](#) 下载参考实作,在下载压缩档并解压缩之后,将其 lib 目录下的 jar 档案复制至您的 Web 应用程序的 /WEB-INF/lib 目录下,另外您还需要 jstl.jar 与 standard.jar 档案,这些档案您可以在 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,您总共需要以下的档案:

```
* 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,所有的请求都透过 FacesServlet 来处理,您可以如下定义:

- web.xml

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。

## 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，使用者送出名称，之后由程序显示使用者名称及欢迎讯息。

## 程序开发人员

先看看应用程序开发人员要作些什么事，我们撰写一个简单的 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 将储存使用者的名称，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，使用者填入名称并送出窗体，之后在 /pages/welcome.jsp 中显示 Bean 中的使用者名称与欢迎讯息。

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

首先网页设计人员撰写 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=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>
      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 卷标库中几乎都是与 HTML 卷标相关的进阶卷标，<h:form>会产生一个窗体，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窗体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=Big5"%>
<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作的。

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，填入名称并送出窗体，您的欢迎页面就会显示了。

## 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，决定要显示欢迎讯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 errorMessage;

    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 setErrMsg(String errMsg) {
        this.errMsg = errMsg;
    }

    public String getErrMsg() {
        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";
        }
    }
}

```

在 `UserBean` 中，我们增加了密码与错误讯息属性，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，并会设定错误讯息，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>
</faces-config>

```

```

</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=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"
        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，这个错误讯息就可以显示在页面上。

## 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）都有一个独特的识别（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 会将扩展名改为 /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> 定义当窗体结果符合的条件时，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>，进一步规范窗体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，预设都是使用 **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块，例如在 `/admin/` 下的页面，您可以在 `<from-view-id>` 中使用 **wildcards**，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 `*` 显式的在定义档中表明，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>
....

```

## 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，将变量或表达式放置在

Unknown macro: { 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，可以透过使用 '.' 运算符来存取它的属性，例如：

```

...

<f:view>

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

...

```

在 JSF 卷标的属性上，" 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宣告的名称，或是 JSF EL 预设的隐含对象，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=Big5"%>

<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 预设的隐含对象变量，它代表 **request** 所有参数的集合，实际是一个 **java.util.Map** 型态对象，JSF 所提供的隐含对象，大致上对应于 [JSP 隐含物件](#)，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** 型态对象，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 型态或数组的话，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 ' 或双自变量 " 括住，如此才不会被认为是变量名称。

在宣告变量名称时，要留意不可与 JSF 的保留字或关键词同名，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。

算术运算子有：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：

表达式	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:

表达式	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:

表达式	结果
<code>#{1 &lt; 2}</code>	true
<code>#{1 lt 2}</code>	true



<code>#{1 &gt; (4/2)}</code>	false
<code>#{1 &gt; (4/2)}</code>	false
<code>#{4.0 &gt;= 3}</code>	true
<code>#{4.0 ge 3}</code>	true
<code>#{4 &lt;= 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:

表达式	结果
<code>#{'a' &lt; 'b'}</code>	true
<code>#{'hip' &gt; 'hit'}</code>	false
<code>#{'4' &gt; 3}</code>	true

EL 运算符的执行优先级与 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。

## 国际化讯息

JSF 的国际化(Internationalization)讯息处理是基于 Java 对国际化的支持,您可以在一个讯息资源文件中统一管理讯息资源,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_TW.properties
```

没有指定语言与地区的 **basename** 是预设的资源档名称，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档名称，例如：

***Accept-Language: zh\_TW, en-US, en***

如果浏览器送来这些 **header**，则预设会使用繁体中文，接着是美式英文，再来是英文语系，如果找不到对应的讯息资源文件，则会使用预设的讯息资源文件。

由于讯息资源文件必须是 **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，例如您可以先在讯息资源文件中写下以下内容：

- **messages\_zh\_TW.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 Big5 messages\_zh\_TW.txt  
messages\_zh\_TW.properties***

转换后的内容会如下：

- **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

```

接下来您可以使用<f:loadBundle>卷标来指定加载讯息资源，一个例子如下：

- index.jsp

#### 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，如果您的浏览器预设接受 zh\_TW 语系的话，则页面上就可以显示中文，否则预设将以英文显示，也就是 messages.properties 的内容，为了能显示多国语系，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_TW">
  <f:loadBundle basename="messages" var="msgs"/>
</f:view>
```

直接指定以上的话，则会使用繁体中文来显示，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讯息资源文件，就上例而言，就是使用 messages\_zh\_TW.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"/>
</f:view>
```

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_TW</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，如果您可以提供一個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"/>
</f:view>
```

在页面中设定一个窗体，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_TW"
    itemLabel="#{msgs.zh_TWText}"/>
  <f:selectItem itemValue="en"
    itemLabel="#{msgs.enText}"/>
</h:selectOneRadio>
```

# Managed Beans

## 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 类别，`<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（例如注销程序），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 的卷标上之属性设定时，只接受 JSF 表示语言。

## Beans 的组态与设定

JSF 预设会读取 faces-config.xml 中关于 Bean 的定义，如果想要自行设置定义档的名称，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>
```

定义档可以有多个，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、类别与存活范围，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>

....
```

如果要在其它类别中取得 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` 等等型态，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：

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

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-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 型态的对象，我们设定为前一个名称为 user 的 UserBean 对象。

## Beans 上的 List, Map

如果您的 Bean 上有接受 List 或 Map 型态的属性，则您也可以在组态档案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 型态的属性, 我们使用 `<list-entries>` 卷标指定将设定一个 List 对象, 其中 `<value-class>` 指定将存入 List 的型态, 而 `<value>` 指定其值, 如果是基本型态, 则会尝试使用指定的 `<value-class>` 来作 Wrapper 类别。

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>
....
```

## 数据转换与验证

### 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 类别，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`

**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 型态的参数, 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

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"%>  
  
<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>可以显示错误讯息，for 属性参考至<h:inputText> 的 id 属性，表示将有关 dateField 的错误讯息显示出来。

假设 faces-config.xml 是这样定义的：

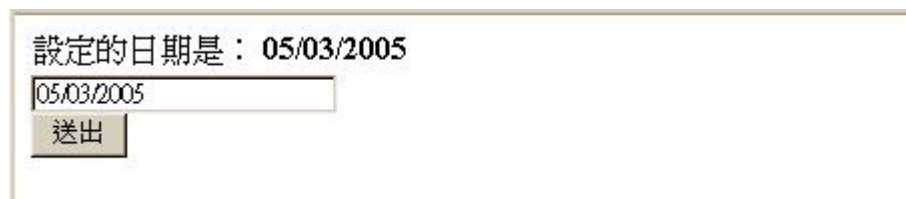
#### faces-config.xml

```
<?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：



設定的日期是： 05/03/2005

05/03/2005

送出

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送出后，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，则会出现以下的讯息：

<f:convertDateTime>卷标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。

## 自订转换器

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，您可以实作 `javax.faces.convert.Converter` 接口，这个接口有两个要实作的方法：

```
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订对象，这个自订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** 类别：

- User.java

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 类别是我们转换器的目标对象，而您有一个 GuestBean 类别：

- GuestBean.java

#### 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 型态的参数，我们来实作一个简单的转换器，为 HTTP 字符串与 User 对象进行转换：

- UserConverter.java

#### 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;
}
}

```

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

- faces-config.xml

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，需提供转换器识别（Converter ID）与转换器类别，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识别，例如：

- index.jsp

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"%>

<f:view>

<html>
<head>
<title>自订转换器</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: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>卷标并使用 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

#### 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}"/>

```

## 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字段值,或是字段值的长度、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 [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

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>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，这表示这个字段一定要输入值，我们也在</h:inputSecret>设定了<f:validateLength>，并设定其 minimum 属性为 6，这表示这个字段最少需要 6 个字符。

这一次在错误訊息的显示上，我们使用<h:messages>标签，当有验证错误发生时，相关的错误訊息会收集起来，使用<h:messages>卷标可以一次将所有的错误訊息显示出来。

下面是一个验证错误的讯息显示：

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。

## 自订验证器

您可以自订自己的验证器，所需要的是实作 javax.faces.validator.Validator 接口，例如我们实作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

```

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);
        }
    }
}

```

您要实作 `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, 分别表示讯息的严重程度 (`INFO`、`WARN`、`ERROR`、`FATAL`)、讯息概述与详细讯息内容, 这些讯息将可以使用 `<h:messages>` 或 `<h: message>` 标签显示在页面上。

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

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

```

要使用自订的验证器，我们可以使用 `<f:validator>` 卷标并设定 `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

#### 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}"/>

.....

```

## 错误讯息处理

在使用标准转换器或验证器时，当发生错误时，会有一些预设的错误讯息显示，这些讯息可以使用<h:messages>或<h:message>标签来显示出来，而这些预设的错误讯息也是可以修改的，您所要作的是提供一个讯息资源文件，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讯息，而 javax.faces.component.UIInput.REQUIRED 是在标签设定了 required 为 true，而使用者没有在字段输入时显示的错误讯息。



您要在 faces-config.xml 中告诉 JSF 您使用的讯息文件名称，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>
```

在这边我们设定了讯息档案的名称为 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讯息如果有设定的话，就使用设定值，如果没有设定的话，就使用默认值。

验证器错误讯息，除了上面的 javax.faces.component.UIInput.REQUIRED 之外，还有以下几个：

讯息识别	预设讯息	用于
javax.faces.validator.NOT_IN_RANGE	Validation Error: Specified attribute is not between the expected values of {0} and {1}.	DoubleRangeValidator 与 LongRangeValidator, {0}与{1}分别代表 minimum 与 maximum 所设定的属性
javax.faces.validator.DoubleRangeValidator.MAXIMUM、 javax.faces.validator.LongRangeValidator.MAXIMUM	Validation Error: Value is greater than allowable maximum of '{0}'.	DoubleRangeValidator 或 LongRangeValidator, {0}表示 maximum 属性
javax.faces.validator.DoubleRangeValidator.MINIMUM、	Validation Error: Value is less than allowable	DoubleRangeValidator 或

javax.faces.validator.LongRangeValidator.MINIMUM	minimum of '{0}'.	LongRangeValidator, {0}代表 minimum 属性
javax.faces.validator.DoubleRangeValidator.TYPE、 javax.faces.validator.LongRangeValidator.TYPE	Validation Error: Value is not of the correct type.	DoubleRangeValidator 或 LongRangeValidator
javax.faces.validator.LengthValidator.MAXIMUM	Validation Error: Value is greater than allowable maximum of "{0}".	LengthValidator, {0}代表 maximum
javax.faces.validator.LengthValidator.MINIMUM	Validation Error: Value is less than allowable minimum of "{0}".	LengthValidator, {0}代表 minimum 属性

在您提供自订讯息的时候, 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讯息。

讯息的显示有概述讯息与详述讯息, 如果是详述讯息, 则在识别上加上 "\_detail", 例如:

```

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

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

....

```

除了在讯息资源文件中提供讯息, 您也可以在程序中使用 FacesMessage 来提供讯息, 例如在 自订验证器 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);
}

....

```

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

```
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讯息资源文件的内容，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();

    // 取得讯息档案主名称

    String messageFileName =
        application.getMessageBundle();

    // 取得当前 Locale 对象

    Locale locale = context.getViewRoot().getLocale();

    // 取得讯息绑定 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、

概述讯息与详述讯息，严重程度有 SEVERITY\_FATAL、SEVERITY\_ERROR、SEVERITY\_WARN 与 SEVERITY\_INFO 四种。

如果需要在讯息资源文件中设定{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，在显示讯息时，onlyfun.caterpillar.message2 的{0}与{1}的位置就会被"param1"与"param2"所取代。

## 自订转换，验证标签

在 [自订验证器](#) 中，我们的验证器只能验证一种 pattern (.+[0-9]+)，我们希望在 JSF 页面上自订匹配的 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>卷标来设定属性，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卷标，并提供相关属性设定，这需要了解 JSP Tag Library 的撰写，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`，这个类别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 [自订验证器](#) 中的 `PasswordValidator` 为例：

- `PasswordValidator.java`

#### `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` 来撰写自己的验证标签：

#### 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` 中注册验证器的识别（Validator ID）：

- `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>
....
    <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 描述档了，我们简单的定义一下：

- taglib.tld

#### 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>
    </tag>

```

```

    <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

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=Big5"%>
<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]+"/>
```



如果要自订转换器标签，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。

## 事件处理

### 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 [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，一个按钮系结至一个方法，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` 系结至一个动作方法（action method），实际上 JSF 会为其自动产生一个「预设的 `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讯息，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`

```
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 setErrorMessage(String errorMessage) {
        this.errorMessage = errorMessage;
    }

    public String getErrorMessage() {
        return errorMessage;
    }

    public void verify(ActionEvent e) {
        if(!name.equals("justin") ||
            !password.equals("123456")) {
            errorMessage = "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，我们故意在错误讯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讯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`

**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设的 `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讯息, 您可以实作 `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>卷标向组件注册事件，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。

## 实时事件

所谓的实时事件（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 的事件模型中会有所谓实时事件，导因于 Web 应用程序的先天特性不同于 GUI 程序，所以 JSF 的事件模式与 GUI 程序的事件模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伺服端的特定对象。

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

- 回复画面（Restore View）

依客户端传来的 session 数据或伺服端上的 session 数据，回复 JSF 画面组件。

- 套用请求值（Apply Request Values）

JSF 画面组件各自获得请求中的值属于自己的值，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。

- 绘制回应画面（Render Response）

对先前的请求处理完之后，产生画面以响应客户端执行结果。

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、更新模式值阶段，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，例如只影响画面的事件。

举个例子来说，在窗体中可能有使用者名称、密码等字段，并提供有一个地区选项按钮，使用者可以在不填下按钮的情况下，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，这会引来不必要的错误。

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，直接进行画面绘制以响应请求，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立即事件。

一个例子如下：

- index.jsp

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，您在输入字段与密码字段中填入的值，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`

#### `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

**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>

```

讯息资源文件的内容则是如下：

- messages\_en.properties

#### 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，另一个讯息资源文件的内容则是英文讯息的翻译而已，其转换为 Java Unicode Escape 格式结果如下：

- messages\_zh\_TW.properties

#### 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：

## 值变事件

如果使用者改变了 JSF 输入组件的值后送出窗体，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_TW" itemLabel="Chinese"/>
    <f:selectItem itemValue="en" itemLabel="English"/>
</h:selectOneMenu>

```

为了仿真 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，立即送出窗体，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_TW";
        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 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";
    }
}
}

```

另一个方法是实作 `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>` 卷标，并设定其 `type` 属性，例如：

```

{code:borderStyle=solid}
<h:selectOneMenu value="#{user.locale}"
    onchange="this.form.submit();">
    <f:valueChangeListener

```

```

        type="onlyfun.caterpillar.SomeListener"/>
        <f:selectItem itemValue="zh_TW" itemLabel="Chinese"/>
        <f:selectItem itemValue="en" itemLabel="English"/>
    </h:selectOneMenu>

```

下面这个页面是对 立即事件 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` 类别来使用:

**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_TW"
                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>
```

## Phase 事件

在 [实时事件](#) 中我们提到，JSF的请求执行到响应，完整的过程会经过六个阶段：

- 回复画面（Restore View）

依客户端传来的 session 数据或伺服器上的 session 数据，回复 JSF 画面组件。

- 套用请求值（Apply Request Values）

JSF 画面组件各自获得请求中的值属于自己的值，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。

- 绘制回应画面（Render Response）

对先前的请求处理完之后，产生画面以响应客户端执行结果。

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，则可以实作 `javax.faces.event.PhaseListener`，并向 `javax.faces.lifecycle.Lifecycle` 登记这个 Listener，以有适当的时候通知事件的发生。

`PhaseListener` 有三个必须实作的方法 `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() 中撰写阶段前后撰写分别想要处理的动作，例如下面这个简单的类别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;
    }
}
```

撰写好 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>
```

```
</lifecycle>
.....
</faces-config>
```

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

## JSF 标签

### 简介 JSF 标准标签

JSF 提供了标准的 HTML Renderer Kit，可以让您搭配 JSF 组件输出 HTML 文件，标准的 HTML Renderer Kit 主要包括了几个类别：

- 输出（Outputs）

其名称以 **output** 作为开头，作用为输出指定的讯息或绑定值。

- 输入（Inputs）

其名称以 **input** 作为开头，其作用为提供使用者输入字段。

- 命令（Commands）

其名称以 **command** 作为开头，其作用为提供命令或连结按钮。

- 选择（Selections）

其名称以 **select** 作为开头，其作用为提供使用者选项的选取。

- 其它

包括了 **form**、**message**、**messages**、**graphicImage** 等等未分类的标签。

JSF 标准 HTML 标签包括了几个共通的属性，整理如下：

属性名称	适用	说明
id	所有组件	可指定 id 名称, 以让其它卷标或组件参考
binding	所有组件	绑定至 UIComponent
rendered	所有组件	是否显示组件

styleClass	所有组件	设定 Cascading stylesheet (CSS)
value	输入、输出、命令组件	设定值或绑定至指定的值
valueChangeListener	输入组件	设定值变事件处理器
converter	输入、输出组件	设定转换器
validator	输入组件	设定验证器
required	输入组件	是否验证必填字段
immediate	输入、命令组件	是否为立即事件

除了共通的属性之外,您还可以在某些组件上设定卷标 HTML 4.01 的属性,像是 `size`、`alt`、`width` 等属性,或者是设定 DHTML 事件属性,例如 `onchange`、`onclick` 等等。

除了 JSF 的标准 HTML 标签之外,您还需要一些标准核心卷标,这些卷标是独立于 `Renderer Kit` 的,JSF 并不限制在 HTML 输出表示层,核心标签可以搭配其它的 `Renderer Kit` 来使用。

详细的 HTML 卷标或核心卷标的使用与属性说明可以查询 [Tag Library Documentation](#) 文件

## 输出类标签

输出类的标签包括了 `outputLabel`、`outputLink`、`outputFormat` 与 `outputText`,分别举例说明如下:

### outputLabel

产生 `<label>` HTML 卷标,使用 `for` 属性指定组件的 client ID,例如:

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

这会产生像是以下的标签:

```
<input id="user" type="text" name="user" value="guest" />
<label for="user">
```

### outputLink

产生 `<a>` HTML 标签,例如:



```
<h:outputLink value=" ../index.jsp"/>
```

value 所指定的内容也可以是 JSF EL 绑定。

## outputFormat

产生指定的文字讯息，可以搭配<f:param>来设定讯息的参数以格式化文字讯息，例如：

```
<f:loadBundle basename="messages" var="msgs"/>
<h:outputFormat value="#{msgs.welcomeText}">
  <f:param value="Hello"/>
  <f:param value="Guest"/>
</h:outputFormat>
```

如果您的 messages.properties 包括以下的内容：

```
welcomeText={0}, Your name is {1}.
```

则{0}与{1}会被取代为<f:param>设定的文字，最后显示的文字会是：

```
Hello, Your name is Guest.
```

另一个使用的方法则是：

```
<h:outputFormat value="{0}, Your name is {1}.">
  <f:param value="Hello"/>
  <f:param value="Guest"/>
</h:outputFormat>
```

## outputText

简单的显示指定的值或绑定的讯息，例如：

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

## 输入类标签

输入类标签包括了 inputText、inputTextarea、inputSecret、inputHidden，分别举例说明如下：

## inputText

显示单行输入字段，即输出<input> HTML 卷标，其 type 属性设定为 text，例如：

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

## inputTextarea

显示多行输入文字区域，即输出<textarea> HTML 卷标，例如：

```
<h:inputTextarea value="#{user.command}"/>
```

## inputSecret

显示密码输入字段，即输出<input> HTML 卷标，其 type 属性设定为 password，例如：

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

您可以设定 redisplay 属性以决定是否要显示密码字段的值，预设是 false。

## inputHidden

隐藏字段，即输出<input> HTML 卷标，其 type 属性设定为 hidden，隐藏字段的值用于保留一些讯息于客户端，以在下一次发送窗体时一并送出，例如：

```
<h:inputHidden value="#{user.hiddenInfo}"/>
```

## 命令类标签

命令类标签包括 commandButton 与 commandLink，其主要作用在于提供一个命令按钮或连结，以下举例说明：

### commandButton

显示一个命令按钮，即输出<input> HTML 卷标，其 type 属性可以设定为 button、submit 或 reset，预设是 submit，按下按钮会触发 javax.faces.event.ActionEvent，使用例子如下：

```
<h:commandButton value="送出" action="#{user.verify}"/>
```

您可以设定 `image` 属性，指定图片的 URL，设定了 `image` 属性的话，`<input>` 卷标的 `type` 属性会被设定为 `image`，例如：

```
<h:commandButton value="#{msgs.commandText}"
    image="images/logo/wiki.jpg"
    action="#{user.verify}"/>
```

## commandLink

产生超级链接，会输出 `<a>` HTML 卷标，而 `href` 属性会有 `'#'`，而 `onclick` 属性会含有一段 JavaScript 程序，这个 JavaScript 的目的是按下连结后自动提交窗体，具体来说其作用就像按钮，但外观却是超级链接，包括在本体部份的内容都会成为超级链接的一部份，一个使用的例子如下：

```
<h:commandLink value="#{msgs.commandText}"
    action="#{user.verify}"/>
```

产生的 HTML 输出范例如下：

```
<a href="#" onclick="document.forms['_id3']['_id3:_idcl'].value='_id3:_id13';
document.forms['_id3'].submit(); return false;">Submit</a>
```

如果搭配 `<f:param>` 来使用，则所设定的参数会被当作请求参数一并送出，例如：

```
<h:commandLink>
    <h:outputText value="welcome"/>
    <f:param name="locale" value="zh_TW"/>
</h:commandLink>
```

## 选择类标签 一

选择类的标签可略分为单选卷标与多选卷标，依外型的不同可以分为单选钮（Radio）、复选框（CheckBox）、列示方块（ListBox）与选单（Menu），以下分别先作简单的说明。

### <h:selectBooleanCheckbox>

在视图上呈现一个复选框，例如：

```
我同意 <h:selectBooleanCheckbox value="#{user.agree}"/>
```

value 所绑定的属性必须接受与传回 boolean 型态。这个组件在网页上呈现的外观如下：

我同意 ☐

<h:selectOneRadio>、<h:selectOneListbox>、<h:selectOneMenu>

这三个标签的作用，是让使用者从其所提供的选项中选择个项目，所不同的就是其外观上的差别，例如：

```
<h:selectOneRadio value="#{user.education}">
  <f:selectItem itemLabel="高中" itemValue="高中"/>
  <f:selectItem itemLabel="大学" itemValue="大学"/>
  <f:selectItem itemLabel="研究所以上" itemValue="研究所以上"/>
</h:selectOneRadio><p>
```

value 所绑定的属性可以接受字符串以外的型态或是自订型态，但记得如果是必须转换的型态或自订型态，必须搭配 标准转换器 或 自订转换器 来转换为对象，<h:selectOneRadio>的外观如下：

☐ 高中 ☐ 大學 ☐ 研究所以上

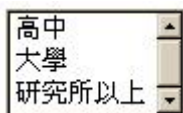
您也可以设定 layout 属性，可设定的属性是 lineDirection、pageDirection，预设是 lineDirection，也就是由左到右来排列选项，如果设定为 pageDirection，则是由上至下排列选项，例如设定为：

```
<h:selectOneRadio layout="pageDirection"
  value="#{user.education}">
  <f:selectItem itemLabel="高中" itemValue="高中"/>
  <f:selectItem itemLabel="大学" itemValue="大学"/>
  <f:selectItem itemLabel="研究所以上" itemValue="研究所以上"/>
</h:selectOneRadio><p>
```

则外观如下：

☐ 高中  
☐ 大學  
☐ 研究所以上

<h:selectOneListbox>、<h:selectOneMenu>的设定方法类似于<h:selectOneRadio>，以下分别列出<h:selectOneListbox>、<h:selectOneMenu>的外观：



### <h:selectManyCheckbox>、<h:selectManyListbox>、<h:selectManyMenu>

这三个卷标提供使用者复选项目的功能，一个<h:selectManyCheckbox>例子如下：

```
<h:selectManyCheckbox layout="pageDirection"
    value="#{user.preferColors}">
    <f:selectItem itemLabel="红" itemValue="false"/>
    <f:selectItem itemLabel="黄" itemValue="false"/>
    <f:selectItem itemLabel="蓝" itemValue="false"/>
</h:selectManyCheckbox><p>
```

value 所绑定的属性必须是数组或集合（Collection）对象，在这个例子中所使用的是 boolean 数组，例如：

#### UserBean.java

```
package onlyfun.caterpillar;

public class UserBean {
    private boolean[] preferColors;

    public boolean[] getPreferColors() {
        return preferColors;
    }

    public void setPreferColors(boolean[] preferColors) {
        this.preferColors = preferColors;
    }

    .....
}
```

如果是其它型态的对象，必要时必须搭配转换器（Converter）进行字符串与对象之间的转换。

下图是<h:selectManyCheckbox>的外观，这是将 layout 设定为 pageDirection 的外观：



<h:selectManyListbox>的设定方法类似，其外观如下：



<h:selectManyMenu>在不同的浏览器中会有不同的外观，在 Mozilla Firefox 中是这样的：



在 Internet Explorer 则是这样的：



## 选择类标签 二

选择类标签可以搭配<f:selectItem>或<f:selectItems>卷标来设定选项，例如：

```
<f:selectItem itemLabel="高中"
               itemValue="高中"
               itemDescription="学历"
               itemDisabled="true"/>
```

itemLabel 属性设定显示在网页上的文字，itemValue 设定发送至伺服器时的值，itemDescription 设定文字描述，它只作用于一些工具程序，对 HTML 没有什么影响，itemDisabled 设定是否选项是否作用，这些属性也都可以使用 JSF Expression Language 来绑定至一个值。

<f:selectItem>也可以使用 value 来绑定一个传回 javax.faces.model.SelectItem 的方法，例如：

```
<f:selectItem value="#{user.sex}"/>
```

则绑定的 Bean 上必须提供下面这个方法：

```
....

public SelectItem getSex() {
```

```

        return new SelectItem("男");
    }
    ....

```

如果要一次提供多个选项，则可以使用<f:selectItems>，它的 value 绑定至一个提供传回 SelectItem?的数组、集合，或者是 Map 对象的方法，例如：

```

<h:selectOneRadio value="#{user.education}">
    <f:selectItems value="#{user.educationItems}"/>
</h:selectOneRadio><p>

```

这个例子中<f:selectItems>的 value 绑定至 user.educationItems，其内容如下：

```

....

private SelectItem[] educationItems;

public SelectItem[] getEducationItems() {
    if(educationItems == null) {
        educationItems = new SelectItem[3];
        educationItems[0] =
            new SelectItem("高中", "高中");
        educationItems[1] =
            new SelectItem("大学", "大学");
        educationItems[2] =
            new SelectItem("研究所以上", "研究所以上");
    }

    return educationItems;
}
....

```

在这个例子中，SelectItem 的第一个建构参数用以设定 value，而第二个参数用以设定 label，SelectItem 还提供有数个建构函式，记得可以参考一下线上 API 文件。

您也可以提供一个传回 Map 对象的方法，Map 的 key-value 会分别作为选项的 label-value，例如：

```

<h:selectManyCheckbox layout="pageDirection"
    value="#{user.preferColors}">
    <f:selectItems value="#{user.preferColorItems}"/>
</h:selectManyCheckbox><p>

```

您要提供下面的程序来搭配上这个例子：

```

....

```

```

private Map preferColorItems;

public Map getPreferColorItems() {
    if(preferColorItems == null) {
        preferColorItems = new HashMap();
        preferColorItems.put("红", "Red");
        preferColorItems.put("黄", "Yellow");
        preferColorItems.put("蓝", "Blue");
    }

    return preferColorItems;
}
....

```

## 其它标签

<h:messages>或<h:message>标签的介绍, 在 [错误讯息处理](#) 中已经有介绍了。

## <h:graphicImage>

这个卷标会绘制一个 HTML <img>卷标, value 可以指定路径或图片 URL, 路径可以指定相对路径或绝对路径, 例如:

```
<h:graphicImage value="/images/logowiki.jpg"/>
```

## <h:panelGrid>

这个卷标可以用来作简单的组件排版, 它会使用 HTML 表格卷标来绘制表格, 并将组件置于其中, 主要指定 columns 属性, 例如设定为 2:

```

<h:panelGrid columns="2">
    <h:outputText value="Username"/>
    <h:inputText id="name" value="#{userBean.name}"/>
    <h:outputText value="Password"/>
    <h:inputText id="password" value="#{userBean.password}"/>
    <h:commandButton value="submit" action="login"/>
    <h:commandButton value="reset" type="reset"/>
</h:panelGrid>

```

则自动将组件分作 2 个 column 来排列, 排列出来的样子如下:



Username	<input type="text"/>
Password	<input type="password"/>
<input type="button" value="submit"/>	<input type="button" value="reset"/>

<h:panelGrid>的本体间只能包括 JSF 组件，如果想要放入非 JSF 组件，例如简单的样版（template）文字，则要使用 <f:verbatim>包括住，例如：

```
<h:panelGrid columns="2">
  <f:verbatim>Username</f:verbatim>
  <h:inputText id="name" value="#{userBean.name}"/>
  <f:verbatim>Password</f:verbatim>
  <h:inputText id="password" value="#{userBean.password}"/>
  <h:commandButton value="submit" action="login"/>
  <h:commandButton value="reset" type="reset"/>
</h:panelGrid>
```

## <h:panelGroup>

这个组件用来将数个 JSF 组件包装起来，使其看来像是一个组件，例如：

```
<h:panelGrid columns="2">
  <h:outputText value="Username"/>
  <h:inputText id="name" value="#{userBean.name}"/>
  <h:outputText value="Password"/>
  <h:inputText id="password" value="#{userBean.password}"/>
  <h:panelGroup>
    <h:commandButton value="submit" action="login"/>
    <h:commandButton value="reset" type="reset"/>
  </h:panelGroup>
</h:panelGrid>
```

在<h:panelGroup>中包括了两个<h:commandButton>，这使得<h:panelGrid>在处理时，将那两个<h:commandButton>看作是一个组件来看待，其完成的版面配置如下所示：

Username	<input type="text"/>
Password	<input type="password"/>
<input type="button" value="submit"/>	<input type="button" value="reset"/>

## 表格处理

对于必须使用表格方式呈现的数据，JSF 的 <h:dataTable> 卷标协助您进行动态表

格数据的输出。

## 简单的表格

很多数据经常使用表格来表现，JSF 提供<h:dataTable>卷标让您得以列举数据并使用表格方式来呈现，举个实际的例子来看，假设您撰写了以下的两个类别：

### UserBean.java

```
package onlyfun.caterpillar;

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

    public UserBean() {
    }

    public UserBean(String name, String password) {
        this.name = name;
        this.password = password;
    }

    public String getName() {
        return name;
    }

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

    public String getPassword() {
        return password;
    }

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

### TableBean.java

```
package onlyfun.caterpillar;

import java.util.*;

public class TableBean {
```

```

private List userList;

public List getUserList() {
    if(userList == null) {
        userList = new ArrayList();
        userList.add(
            new UserBean("caterpillar", "123456"));
        userList.add(
            new UserBean("momor", "654321"));
        userList.add(
            new UserBean("becky", "7890"));
    }

    return userList;
}
}

```

在 TableBean 中，我们假设 `getUserList()` 方法实际上是从数据库中查询出 UserBean 的内容，之后传回 List 对象，若我们的 `faces-config.xml` 如下：

#### faces-config.xml

```

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

<!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>
    <managed-bean>
        <managed-bean-name>tableBean</managed-bean-name>
        <managed-bean-class>
            onlyfun.caterpillar.TableBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
    <managed-bean>
        <managed-bean-name>userBean</managed-bean-name>
        <managed-bean-class>
            onlyfun.caterpillar.UserBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
    </managed-bean>
</faces-config>

```

我们可以如下使用<h:dataTable>来产生表格数据:

index.jsp

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

<html>
<body>
<f:view>
    <h:dataTable value="#{tableBean.userList}" var="user">
        <h:column>
            <h:outputText value="#{user.name}"/>
        </h:column>
        <h:column>
            <h:outputText value="#{user.password}"/>
        </h:column>
    </h:dataTable>
</f:view>
</body>
</html>
```

<h:dataTable>的 value 值绑定 tableBean 的 userList 方法, 它会一个一个取出 List 中的数据并设定给 var 设定的 user, 之后在每一个 column 中我们可以显示所列举出的 user.name 与 user.password, 程序的结果会如下所示:

所产生的 HTML 表格卷标如下:

```
<table>

  <tbody>

    <tr>

      <td>caterpillar</td>

      <td>123456</td>

    </tr>

    <tr>

      <td>momor</td>

      <td>654321</td>

    </tr>

    <tr>

      <td>becky</td>

      <td>7890</td>

    </tr>
```

```

</tbody>
</table>

```

<h:dataTable>的 value 值绑定的对象可以是以下的型态:

- 数组
- java.util.List 的实例
- java.sql.ResultSet 的实例
- javax.servlet.jsp.jstl.sql.Result 的实例
- javax.faces.model.DataModel 的实例

## 表头, 表尾

<h:dataTable>配合<h:column>来以表格的方式显示数据, <h:column>中只能包括 JSF 组件或者是<f:facet>, JSF 支援两种 facet: header 与 footer。分别用以设定表格的表头与表尾文字, 一个设定的例子如下:

```

<h:dataTable value="#{tableBean.userList}" var="user">
  <h:column>
    <f:facet name="header">
      <h:outputText value="Name"/>
    </f:facet>
    <h:outputText value="#{user.name}"/>
    <f:facet name="footer">
      <h:outputText value="****"/>
    </f:facet>
  </h:column>
  <h:column>
    <f:facet name="header">
      <h:outputText value="Password"/>
    </f:facet>
    <h:outputText value="#{user.password}"/>
    <f:facet name="footer">
      <h:outputText value="****"/>
    </f:facet>
  </h:column>
</h:dataTable>

```

所产生的表格如下所示:

Name	Password
carollee	123456
roselee	654321
lucylee	7890
xxxx	xxxx

另外，对于表头、表尾仍至于每一行列，都可以分别设定 CSS 风格，例如下面这个 styles.css 摘录自 Core JSF 一书：

**styles.css**

```
.orders {  
    border: thin solid black;  
}  
  
.ordersHeader {  
    text-align: center;  
    font-style: italic;  
    color: Snow;  
    background: Teal;  
}  
  
.evenColumn {  
    height: 25px;  
    text-align: center;  
    background: MediumTurquoise;  
}  
  
.oddColumn {  
    text-align: center;  
    background: PowderBlue;  
}
```

可以在我们的页面中如下加入：

```
....  
<link href="styles.css" rel="stylesheet" type="text/css"/>  
....  
<h:dataTable value="#{tableBean.userList}" var="user"  
               styleClass="orders"  
               headerClass="ordersHeader"  
               rowClasses="evenColumn,oddColumn">
```

```

<h:column>
    <f:facet name="header">
        <h:outputText value="Name"/>
    </f:facet>
    <h:outputText value="#{user.name}"/>
    <f:facet name="footer">
        <h:outputText value="****"/>
    </f:facet>
</h:column>
<h:column>
    <f:facet name="header">
        <h:outputText value="Password"/>
    </f:facet>
    <h:outputText value="#{user.password}"/>
    <f:facet name="footer">
        <h:outputText value="****"/>
    </f:facet>
</h:column>
</h:dataTable>

```

则显示的表格结果如下：

<i>Name</i>	<i>Password</i>
caterpillar	123456
momor	654321
becky	7890
****	****

## TableModel 类别

在 [简单的表格](#) 中曾经提过，<h:dataTable>可以列举以下几种型态的数据：

- 数组
- java.util.List 的实例
- java.sql.ResultSet 的实例
- javax.servlet.jsp.jstl.sql.Result 的实例
- javax.faces.model.DataModel 的实例

对于前四种型态，JSF 实际上是以 javax.faces.model.DataModel 加以包装，DataModel 是个抽象类别，其子类别都是位于 javax.faces.model 这个 package 下：

- ArrayDataModel
- ListDataModel
- ResultDataModel
- ResultSetDataModel
- ScalarDataModel

如果您想要对表格数据有更多的控制，您可以直接使用 **DataModel** 来设定表格数据，呼叫 **DataModel** 的 **setWrappedObject()** 方法可以让您设定对应型态的数据，呼叫 **getWrappedObject()** 则可以取回数据，例如：

**TableBean.java**

```
package onlyfun.caterpillar;

import java.util.*;
import javax.faces.model.DataModel;
import javax.faces.model.ListDataModel;

public class TableBean {
    private DataModel model;
    private int rowIndex = -1;

    public DataModel getUsers() {
        if(model == null) {
            model = new ListDataModel();
            model.setWrappedData(getUserList());
        }

        return model;
    }

    private List getUserList() {
        List userList = new ArrayList();
        userList.add(new UserBean("caterpillar", "123456"));
        userList.add(new UserBean("momor", "654321"));
        userList.add(new UserBean("becky", "7890"));

        return userList;
    }

    public int getSelectedRowIndex() {
        return rowIndex;
    }

    public String select() {
        rowIndex = model.getRowIndex();
    }
}
```



```

        return "success";
    }
}

```

在这个 Bean 中，我们直接设定 `DataModel?`，将 `userList` 设定给它，如您所看到的，我们还可以取得 `DataModel?` 的各个变项，在这个例子中，`select()` 将作为点选表格之后的事件处理方法，我们可以藉由 `DataModel?` 的 `getRowIndex ()` 来取得所点选的是哪一 row 的资料，例如：

**index.jsp**

```

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

<html>
<link href="styles.css" rel="stylesheet" type="text/css"/>
<body>
<f:view>
    <h:form>
        <h:dataTable value="#{tableBean.users}" var="user"
            styleClass="orders"
            headerClass="ordersHeader"
            rowClasses="evenColumn,oddColumn">
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Name"/>
                </f:facet>
                <h:commandLink action="#{tableBean.select}">
                    <h:outputText value="#{user.name}"/>
                </h:commandLink>

                <f:facet name="footer">
                    <h:outputText value="****"/>
                </f:facet>
            </h:column>
            <h:column>
                <f:facet name="header">
                    <h:outputText value="Password"/>
                </f:facet>
                <h:outputText value="#{user.password}"/>
                <f:facet name="footer">
                    <h:outputText value="****"/>
                </f:facet>
            </h:column>
        </h:dataTable>
    </h:form>
    Selected Row: <h:outputText

```

```

        value="#{tableBean.selectedRowIndex}"/>
    </f:view>
</body>
</html>

```

DataModel 的 rowIndex 是从 0 开始计算，当处理 ActionEvent 时，JSF 会逐次递增 rowIndex 的值，这让您得知目前正在处理的是哪一个 row 的数据，一个执行的图示如下：

Name	Password
caterpillar	123456
momor	654321
becky	7890
*****	*****

Selected Row: 2

## 自订组件

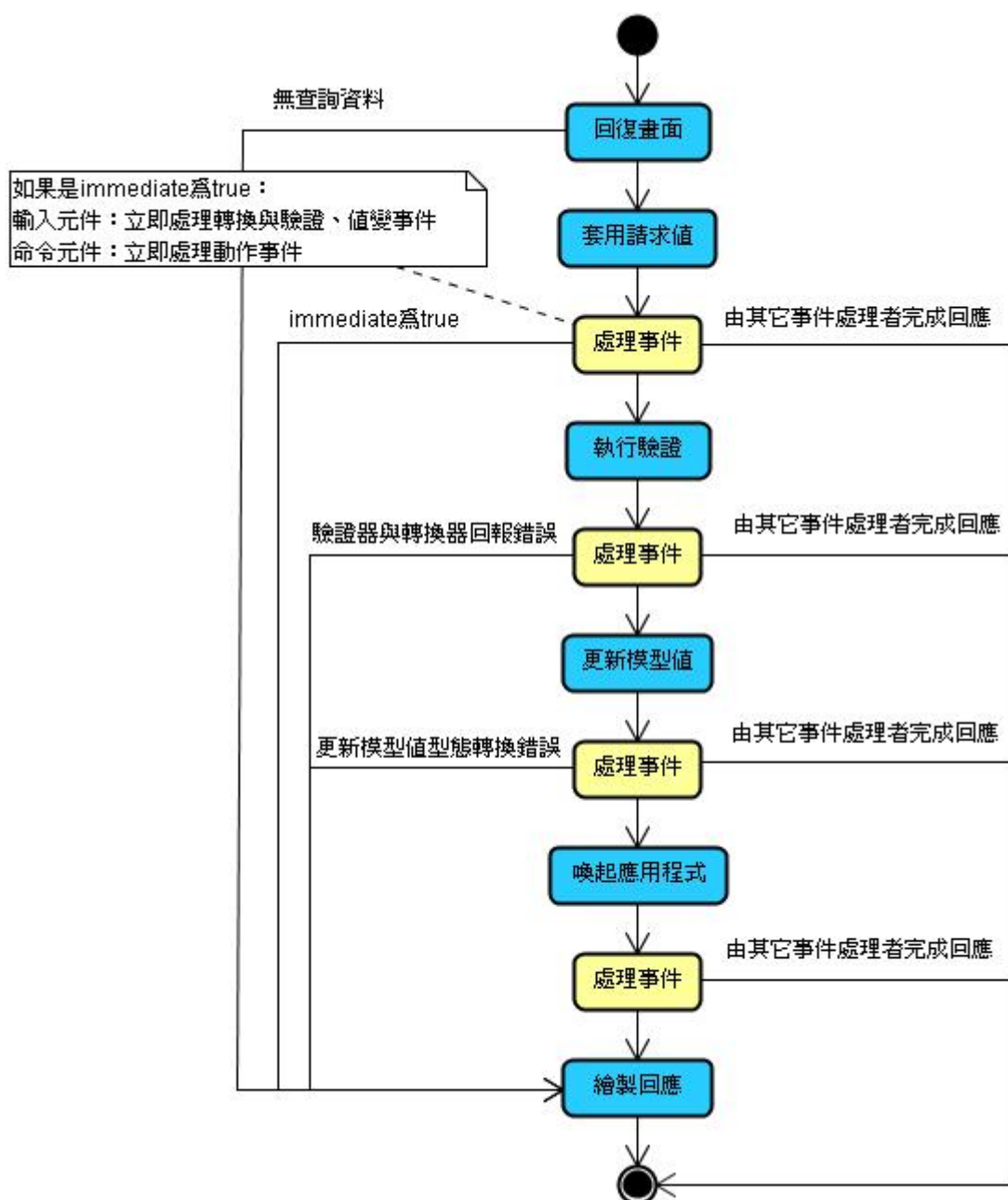
### JSF 生命周期与组件概述

要开发 JSF 组件，您需要更深入了解 JSF 的一些处理细节，包括了 JSF 生命周期以及 JSF 框架。

### JSF 生命周期

JSF 的每个组件基本上都是可替换的，像是转换器 (Converter)、验证器 (Validator)、组件 (Component)、绘制器 (Renderer) 等等，每个组件都可以替换让 JSF 在使用时更有弹性，但相对的所付出的就是组件组合时的复杂性，为此，最基本的，如果您打算自订一些 JSF 组件，那么您对于 JSF 处理请求的每个阶段必须要有所了解。

下图是 JSF 处理请求时的每个阶段与简单说明，起始状态即使用者端发出请求时，终止状态则相当于绘制器发出响应时：



扣除事件处理，JSF 总共必须经过六个阶段：

- 回复画面（Restore View）

对于选择的页面如果是初次浏览则建立新的组件树。如果是会话阶段，会从使用者端或服务端的数据找寻数据以回复每个组件的状态并重建组件树，如果不包括请求参数，则直接跳过接下来的阶段直接绘制响应。

- 套用申请值（Apply Request Values）

每个组件尝试从到来的请求中找寻自己的参数并更新组件值，在这边会触发 `ActionEvent`，这个事件会被排入队列中，然后在唤起应用程序阶段之后才会真正由事件处

理者进行处理。

然而对于设定 `immediate` 为 `true` 的命令（`Command`）组件来说，会立即处理事件并跳过之后的阶段直接绘制响应，而对于设定 `immediate` 为 `true` 的输入（`Input`）组件，会马上进行转换验证并处理值变事件，之后跳过接下来的阶段，直接绘制响应。

- 执行验证（`Process Validations`）

进行转换与验证处理，如果验证错误，则会跳过之后的阶段，直接绘制响应，结果是重新呼叫同一页绘制结果。

- 更新模型值（`Update Model Values`）

更新每一个与组件绑定的 `backing bean` 或模型对象。

- 唤起应用程序（`Invoke Application`）

处理动作事件，并进行后端应用程序逻辑。

- 绘制回应（`Render Response`）

使用绘制器绘制页面。

如果您只是要「使用」JSF，则您最基本的只需要知道「执行验证」、「更新模型值」、与「唤起应用程序」这三个阶段及中间的事件触发，JSF 参考实作将这三个阶段之外的其它阶段之复杂性隐藏起来了，您不需要知道这几个阶段的处理细节。

然而如果您要自订组件，则您还必须知道「回复画面」、「套用请求值」与「绘制响应」这些阶段是如何处理的，这几个阶段相当复杂，所幸的是您可以使用 JSF 所提供的框架来进行组件自订，JSF 提供的框架已经很大程度上降低了组件制作的复杂性。

当然，即使JSF框架降低了复杂性，但实际上要处理JSF自订组件还是很复杂的一件事，在尝试开发自订组件之前，您可以先搜寻一些网站，像是 [Apache MyFaces](http://myfaces.apache.org/) <http://myfaces.apache.org/>，看看是不是已经有相关类似的组件已经开发完成，省去您重新自订组件的气力。

## 概述自订组件

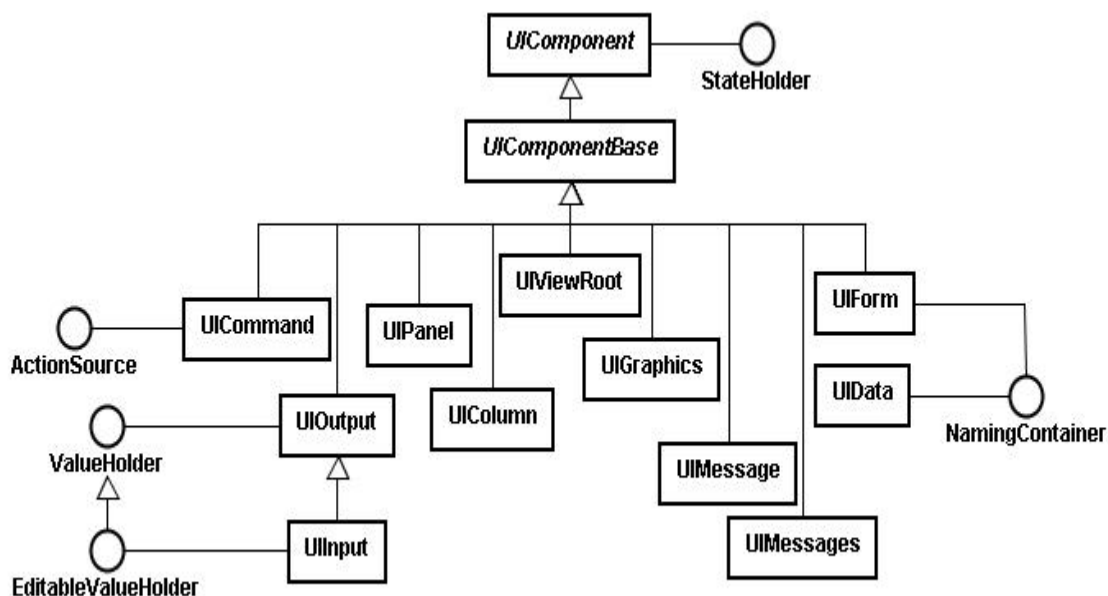
所谓的「自订 JSF 组件」是一个概略的称呼，事实上，一个 JSF 组件包括了三个部份：`Tag`、`Component` 与 `Renderer`。

`Tag` 即之前一直在使用的 JSF 卷标，类似于 HTML 卷标，JSF 卷标主要是方便网页设计人员进行版面配置与数据呈现的一种方式，实际的处理中，JSF 标签的目的在于设定 `Component` 属性、设定验证器、设定数据绑定、设定方法绑定等等。

Component 的目的在于处理请求，当请求来到服务端应用程序时，每一个 Component 都有机会根据自己的 client id，从请求中取得属于自己的值，接着 Component 可以将这个值作处理，然后设定给绑定的 bean。

当请求来到 Web 应用程序时，HTTP 中的字符串内容可以转换为 JSF 组件所需的值，这个动作称之为译码（decode），相对的，将 JSF 组件的值转换为 HTTP 字符串数据并送至客户端，这个动作称之为编码（encode），Component 可自己处理编码、译码的任务，也可以将之委托给 Renderer 来处理。

当您要自订 Component 时，您可以继承 UIComponent 或其相关的子类别，这要根据您实际要自订的组件而定，如果您要自订一个输出组件，可以继承 UIOutput，如果要自订一个输入组件，则可以继承 UIInput，每一个标准的 JSF 组件实际上都对应了一个 UIComponent 的子类别，下图为一个大致类别继承架构图：



实际上要自订一个组件是复杂的一件工作，您首先要学会的是一个完整的自订组件流程，实际上要自订一个组件时，您可以参考一下网络上的一些成品，例如 Apache MyFaces <http://myfaces.apache.org/>，接下来后面的几个主题所要介绍的，将只是一个自订组件的简单流程。

Renderer 是一个可替换的组件，您的 Component 可以搭配不同的 Renderer，而不用自行担任绘制响应或译码的动作，这会让您的 Component 可以重用，当您需要将响应从 HTML 转换为其它的媒介时（例如行动电话网络），则只要替换 Renderer 就可以了，这是一个好处，或者您可以简单的替换掉一个 Renderer，就可以将原先简单的 HTML 响应，替换为有 JavaScript 功能的 Renderer。

当您开始接触自订组件时，您会开始接触到 JSF 的框架（Framework），也许有几个类别会是您经常接触的：

- javax.faces.component.UIComponent

自订 Component 所要继承的父类别，但通常，您是继承其子类别，例如 `UIInput`、`UIOutput` 等等。

- `javax.faces.webapp.UIComponentTag`

自订 JSF 标签所要继承的父类别，继承它可以帮您省去许多 JSF 标签处理的细节。

- `javax.faces.context.FacesContext`

包括了 JSF 相关的请求信息，您可以透过它取得请求对象或请求参数，或者是 `javax.faces.application.Application` 物件。

- `javax.faces.application.Application`

包括了一个应用程序所共享的信息，像是 `locale`、验证器、转换器等等，您可以透过一些 [工厂方法](#) 取得相关的信息。

## 简单实例

- 在不考虑组件有子组件的情况下，这边以实际的一个例子来说明开发组件的过程，至于考虑子组件的情况请参考专书介绍。

## 编码, 解码

Component 可以自己负责将对象数据编码为 HTML 文件或其它的输出文件，也可以将这个任务委托给 `Renderer`，这边先介绍的是让 Component 自己负责编码的动作。

这边着重的是介绍完成自订组件所必须的流程，所以我们不设计太复杂的组件，这边将完成以下的组件，这个组件会有一个输入文字字段以及一个送出按钮：



您要继承 `UIComponent` 或其子类别来自订 Component，由于文字字段是一个输入字段，为了方便，您可以继承 `UIInput` 类别，这可以让您省去一些处理细节的功夫，在继承 `UIComponent` 或其子类别后，与编码相关的主要有三个方法：

- `encodeBegin()`
- `encodeChildren()`
- `encodeEnd()`

其中 `encodeChildren()` 是在包括子组件时必须定义，Component 如果它的 `getRendersChildren()` 方法传回 `true` 时会呼叫 `encodeChildren()` 方法，预设上，`getRendersChildren()` 方法传回 `false`。

由于我们的自订组件相当简单，所以将编码的动作写在 `encodeBegin()` 或是 `encodeEnd()` 都可以，我们这边是定义 `encodeBegin ()` 方法：

**UITextWithCmd.java**

```
package onlyfun.caterpillar;

import java.io.IOException;
import java.util.Map;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

public class UITextWithCmd extends UIInput {
    private static final String TEXT = ".text";
    private static final String CMD = ".cmd";

    public UITextWithCmd() {
        setRendererType(null);
    }

    public void encodeBegin(FacesContext context)
        throws IOException {
        ResponseWriter writer = context.getResponseWriter();
        String clientId = getClientId(context);

        encodeTextField(writer, clientId);
        encodeCommand(writer, clientId);
    }

    public void decode(FacesContext context) {

        // .....

    }

    private void encodeTextField(ResponseWriter writer,
        String clientId) throws IOException {
        writer.startElement("input", this);
        writer.writeAttribute("name", clientId + TEXT, null);

        Object value = getValue();
        if(value != null) {
            writer.writeAttribute("value",
                value.toString(), null);
        }
    }
}
```

```

        String size = (String) getAttributes().get("size");
        if(size != null) {
            writer.writeAttribute("size", size, null);
        }

        writer.endElement("input");
    }

    private void encodeCommand(ResponseWriter writer,
                               String clientId) throws IOException {
        writer.startElement("input", this);
        writer.writeAttribute("type", "submit", null);
        writer.writeAttribute("name", clientId + CMD, null);
        writer.writeAttribute("value", "submit", null);
        writer.endElement("input");
    }
}

```

在 `encodeBegin()` 方法中, 我们取得 `ResponseWriter` 对象, 这个对象可以协助您输出 HTML 卷标、属性等, 我们使用 `getClientId()` 取得组件的 id, 这个 id 是每个组件的唯一识别, 预设上如果您没有指定, 则 JSF 会自动为您产生 id 值。

接着我们分别对输入文字字段及送出钮作 HTML 标签输出, 在输出时, 我们将 `name` 属性设成 `clientId` 与一个字符串值的结合 (即 TEXT 或 CMD), 这是为了方便在译码时, 取得对应 `name` 属性的请求值。

在 `encodeTextField` 中我们有呼叫 `getValue()` 方法, 这个方法是从 `UIOutput` 继承下来的, `getValue()` 方法可以取得 `Component` 的设定值, 这个值可能是静态的属性设定值, 也可能是 JSF Expression 的绑定值, 预设会先从组件的属性设定值开始找寻, 如果找不到, 再从绑定值 (`ValueBinding` 对象) 中找寻, 组件的属性值或绑定值的设定, 是在定义 Tag 时要作的事。

编编的部份总结来说, 是取得 `Component` 的值并作适当的 HTML 标签输出, 再来我们看看译码的部份, 这是定义在 `decode()` 方法中, 将下面的内容加入至上面的类别定义中:

```

.....

public void decode(FacesContext context) {
    Map reqParaMap = context.getExternalContext().
        getRequestParameterMap();
    String clientId = getClientId(context);

    String submittedValue =
        (String) reqParaMap.get(clientId + TEXT);
    setSubmittedValue(submittedValue);
}

```



```

        setValid(true);
    }
    ....

```

我们必须先取得 `RequestParamerMap`，这个 `Map` 对象中填入了所有客户端传来的请求参数，`Component` 在这个方法中有机会查询这些请求参数中，是否有自己所想要取得的数据，记得我们之前译码时，是将输入字段的 `name` 属性译码为 `client id` 加上一个字符串值（即 `TEXT` 设定的值），所以这时，我们尝试从 `RequestParamerMap` 中取得这个请求值。

取得请求值之后，您可以将数据藉由 `setSumittedValue()` 设定给绑定的 `bean`，最后呼叫 `setValid()` 方法，这个方法设定为 `true` 时，表示组件正确的获得自己的值，没有任何的错误发生。

由于我们先不使用 `Renderer`，所以在建构函式中，我们设定 `RendererType` 为 `null`，表示我们不使用 `Renderer` 进行译码输出：

```

public UITextWithCmd() {
    setRendererType(null);
}

```

在我们的例子中，我们都是处理字符串对象，所以这边不需要转换器，如果您需要使用转换器，可以呼叫 `setConverter()` 方法加以设定，在不使用 `Renderer` 的时候，`Component` 要设定转换器来自行进行字符串与对象的转换。

## 组件卷标

完成 `Component` 的自订，接下来要设定一个自订 `Tag` 与之对应，自订 `Tag` 的目的，在于设定 `Component` 属性，取得 `Componenty` 型态，取得 `Renderer` 型态值等；属性的设定包括了设定静态值、设定绑定值、设定验证器等等。

要自订与 `Component` 对应的 `Tag`，您可以继承 `UIComponentTag`，例如：

**TextWithCmdTag.java**

```

package onlyfun.caterpillar;

import javax.faces.application.Application;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import javax.faces.webapp.UIComponentTag;

public class TextWithCmdTag extends UIComponentTag {
    private String size;
    private String value;
}

```

```
public String getComponentType() {
    return "onlyfun.caterpillar.TextWithCmd";
}

public String getRendererType() {
    return null;
}

public void setProperties(UIComponent component) {
    super.setProperties(component);

    setStringProperty(component, "size", size);
    setStringProperty(component, "value", value);
}

private void setStringProperty(UIComponent component,
                               String attrName, String attrValue) {
    if(attrValue == null)
        return;

    if(isValueReference(attrValue)) {
        FacesContext context =
            FacesContext.getCurrentInstance();
        Application application =
            context.getApplication();
        ValueBinding binding =
            application.createValueBinding(attrValue);
        component.setValueBinding(attrName, binding);
    }
    else {
        component.getAttributes().
            put(attrName, attrValue);
    }
}

public void release() {
    super.release();
    size = null;
    value = null;
}

public String getSize() {
    return size;
}
```

```
}

public void setSize(String size) {
    this.size = size;
}

public String getValue() {
    return value;
}

public void setValue(String value) {
    this.value = value;
}
}
```

首先看到这两个方法:

```
public String getComponentType() {
    return "onlyfun.caterpillar.TextWithCmd";
}

public String getRendererType() {
    return null;
}
```

由于我们的 Component 目前不使用 Renderer, 所以 `getRendererType()` 传回 null 值, 而 `getComponentType()` 在于让 JSF 取得这个 Tag 所对应的 Component, 所传回的值在 `faces-config.xml` 中要有定义, 例如:

```
....
<component>
    <component-type>
        onlyfun.caterpillar.TextWithCmd
    </component-type>
    <component-class>
        onlyfun.caterpillar.UITextWithCmd
    </component-class>
</component>
....
```

藉由 `faces-config.xml` 中的定义，JSF 可以得知 `onlyfun.caterpillar.TextWithCmd` 的真正类别，而这样的定义方式很显然的，您可以随时换掉 `<component- class>` 所对应的类别，也就是说，Tag 所对应的 Component 是可以随时替换的。

在设定 Component 属性值时，可以由 `component.getAttributes()` 取得 Map 对象，并将卷标属性值存入 Map 中，这个 Map 对象可以在对应的 Component 中使用 `getAttributes()` 取得，例如在上一个主题中的 `UITextWithCmd` 中可以如下取得存入 Map 的 size 属性：

**UITextWithCmd.java**

```
package onlyfun.caterpillar;

import java.io.IOException;
import java.util.Map;

import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;

public class UITextWithCmd extends UIInput {
    ....
    private void encodeTextField(ResponseWriter writer,
                                String clientId) throws IOException {
        ....

        String size = (String) getAttributes().get("size");
        if(size != null) {
            writer.writeAttribute("size", size, null);
        }
        .....
    }
    ....
}
```

可以使用 `isValueReference()` 来测试是否为 JSF Expression Language 的绑定语法，如果是的话，则我们必须建立 `ValueBinding` 对象，并设定值绑定：

```
....

private void setStringProperty(UIComponent component,
                               String attrName, String attrValue) {
    if(attrValue == null)
        return;

    if(isValueReference(attrValue)) {
```

```

FacesContext context =
    FacesContext.getCurrentInstance();
Application application =
    context.getApplication();
ValueBinding binding =
    application.createValueBinding(attrValue);
component.setValueBinding(attrName, binding);
}
else {
    component.getAttributes().
        put(attrName, attrValue);
}
}
....

```

如果是 value 属性，记得在上一个主题中我们提过，从 UIOutput 继承下来的 `getValue()` 方法可以取得 Component 的 value 设定值，这个值可能是静态的属性设定值，也可能是 JSF Expression 的绑定值，预设会先从组件的属性设定值开始找寻，如果找不到，再从绑定值（ValueBinding 对象）中找寻。

最后，我们必须提供自订 Tag 的 tld 档：

#### textcmd.tld

```

<?xml version="1.0" encoding="UTF-8"?>
<taglib version="2.0"
    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">
<tlib-version>1.0</tlib-version>
<jsp-version>2.0</jsp-version>
<short-name>textcmd</short-name>

<uri>http://caterpillar.onlyfun.net/textcmd</uri>

<tag>
    <name>textcmd</name>
    <tag-class>onlyfun.caterpillar.TextWithCmdTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>size</name>
    </attribute>
    <attribute>
        <name>value</name>
    </attribute>

```

```

        <required>true</required>
    </attribute>
</tag>

</taglib>

```

## 使用自订组件

在 Component 与 Tag 自订完成后，这边来看看如何使用它们，首先定义 faces-config.xml:

### faces-config.xml

```

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

<!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>
    <component>
        <component-type>
            onlyfun.caterpillar.TextWithCmd
        </component-type>
        <component-class>
            onlyfun.caterpillar.UITextWithCmd
        </component-class>
    </component>
    <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-bean>
</faces-config>

```

<component>中定义 Component 的型态与实际的类别对应，在您于自订 Tag 中呼叫 `getComponentType()` 方法所返回的值，就是寻找 <component-type> 设定的值对应，并由此得知真正对应的 Component 类别。

我们所撰写的 SomeBean 测试类别如下：

### SomeBean

```
package onlyfun.caterpillar;
```

```
public class SomeBean {  
    private String data;  
  
    public String getData() {  
        return data;  
    }  
  
    public void setData(String data) {  
        this.data = data;  
    }  
}
```

这边写一个简单的网页来测试一下我们撰写的自订组件：

index.jsp

```
<%@ taglib uri="http://java.sun.com/jsf/html" prefix="h" %>  
<%@ taglib uri="http://java.sun.com/jsf/core" prefix="f" %>  
  
<%@ taglib uri="/WEB-INF/textcmd.tld" prefix="oc" %>  
<html>  
<link href="styles.css" rel="stylesheet" type="text/css"/>  
<head>  
<title></title>  
</head>  
<body>  
<f:view>  
    <h:form>  
        Input data: <oc:textcmd size="10"  
                        value="#{someBean.data}"/>  
    </h:form>  
    <h:outputText value="#{someBean.data}"/>  
</f:view>  
</body>  
</html>
```

## 自订 Renderer

Component 可以将译码、编码的动作交给 Renderer，这让您的表现层技术可以轻易的抽换，我们可以将之前的自订组件的译码、编码动作移出至 Renderer，不过由于我们之前设计的 Component 是个很简单的组件，事实上，如果只是想新增一个 Command 在输入字段旁边，我们并不需要大费周章的自订一个新的组件，我们可以直接为输入字段更换一个自订的 Renderer。

要自订一个 `Renderer`, 您要继承 `javax.faces.render.Renderer`, 我们的自订 `Renderer` 如下:

**TextCmdRenderer.java**

```
package onlyfun.caterpillar;

import java.io.IOException;
import java.util.Map;
import javax.faces.component.EditableValueHolder;
import javax.faces.component.UIComponent;
import javax.faces.component.UIInput;
import javax.faces.context.FacesContext;
import javax.faces.context.ResponseWriter;
import javax.faces.render.Renderer;

public class TextCmdRenderer extends Renderer {
    private static final String TEXT = ".text";
    private static final String CMD = ".cmd";

    public void encodeBegin(FacesContext context,
        UIComponent component) throws IOException {
        ResponseWriter writer = context.getResponseWriter();
        String clientId = component.getClientId(context);

        encodeTextField(component, writer, clientId);
        encodeCommand(component, writer, clientId);
    }

    public void decode(FacesContext context,
        UIComponent component) {
        Map reqParaMap = context.getExternalContext().
            getRequestParameterMap();
        String clientId = component.getClientId(context);

        String submittedValue =
            (String) reqParaMap.get(clientId + TEXT);
        ((EditableValueHolder) component).setSubmittedValue(
            submittedValue);
        ((EditableValueHolder) component).setValid(true);
    }

    private void encodeTextField(UIComponent component,
        ResponseWriter writer, String clientId)
        throws IOException {
        writer.startElement("input", component);
    }
}
```



```

        writer.writeAttribute("name", clientId + TEXT, null);

        Object value = ((UIInput) component).getValue();
        if(value != null) {
            writer.writeAttribute("value",
                value.toString(), null);
        }

        String size =
            (String) component.getAttributes().get("size");
        if(size != null) {
            writer.writeAttribute("size", size, null);
        }

        writer.endElement("input");
    }

    private void encodeCommand(UIComponent component,
        ResponseWriter writer,
        String clientId) throws IOException {
        writer.startElement("input", component);
        writer.writeAttribute("type", "submit", null);
        writer.writeAttribute("name", clientId + CMD, null);
        writer.writeAttribute("value", "submit", null);
        writer.endElement("input");
    }
}

```

这个自订的 `Renderer` 其译码、编码过程，与之前直接在 `Component` 中进行译码或编码过程是类似的，所不同的是在译码与编码的方法上，多了 `UIComponent` 参数，代表所代理绘制的 `Component`。

接下来在自订 `Tag` 上，我们的 `TextWithCmdTag` 与之前主题所介绍的没什么差别，只不过在 `getComponentType()` 与 `getRendererType()` 方法上要修改一下：

#### TextWithCmdTag.java

```

package onlyfun.caterpillar;

import javax.faces.application.Application;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import javax.faces.webapp.UIComponentTag;

public class TextWithCmdTag extends UIComponentTag {

```

```
private String size;
private String value;

public String getComponentType() {
    return "javax.faces.Input";
}

public String getRendererType() {
    return "onlyfun.caterpillar.TextCmd";
}
.....
}
```

getComponentType()取得的是"javax.faces.Input"，它实际上对应至 UIInput 类别，而 getRendererType()取回的是"onlyfun.caterpillar.TextCmd"，这会在 faces-config.xml 中定义，以对应至实际的 Renderer 类别：

**faces-config.xml**

```
....
<faces-config>
  <render-kit>
    <renderer>
      <component-family>
        javax.faces.Input
      </component-family>
      <renderer-type>
        onlyfun.caterpillar.TextCmd
      </renderer-type>
      <renderer-class>
        onlyfun.caterpillar.TextCmdRenderer
      </renderer-class>
    </renderer>
  </render-kit>
  ....
</faces-config>
```

为 Component 定义一个 Renderer, 必须由 component family 与 renderer type 共同定义, 这并不难理解, 因为一个 Component 可以搭配不同的 Renderer, 但它是属于同一个 component family, 例如 UIInput 就是属于 javax.faces.Input 这个组件家族, 而我们为它定义一个新的 Renderer。

接下未完成的范例可以取之前主题介绍过的, 我们虽然没有自订组件, 但我们为 UIInput 置换了一个新的 Renderer, 这个 Renderer 会在输入字段上加入一个按钮。

如果您坚持使用之前自订的 UITextWithCmd, 则可以如下修改:

#### UITextWithCmd.java

```
package onlyfun.caterpillar;

import javax.faces.component.UIInput;

public class UITextWithCmd extends UIInput {
    public UITextWithCmd() {
        setRendererType("onlyfun.caterpillar.TextCmd");
    }
}
```

我们只是单纯的继承 UIInput, 然后使用 setRendererType() 设定 "onlyfun.caterpillar.TextCmd", 但并没有为组件加入什么行为, 看来什么事都没有作, 但事实上这是因为继承了 UIInput, 它为我们处理了大多数的细节。

接下来同样的, 设定自订 Tag:

#### TextWithCmdTag.java

```
package onlyfun.caterpillar;

import javax.faces.application.Application;
import javax.faces.component.UIComponent;
import javax.faces.context.FacesContext;
import javax.faces.el.ValueBinding;
import javax.faces.webapp.UIComponentTag;

public class TextWithCmdTag extends UIComponentTag {
    private String size;
    private String value;

    public String getComponentType() {
        return "onlyfun.caterpillar.TextWithCmd";
    }

    public String getRendererType() {
        return "onlyfun.caterpillar.TextCmd";
    }
}
```

```
}  
.....  
}
```

要使用自订的 Component，记得要在 faces-config.xml 中再加入：

```
.....  
  
    <component>  
  
        <component-type>  
  
            onlyfun.caterpillar.TextWithCmd  
  
        </component-type>  
  
        <component-class>  
  
            onlyfun.caterpillar.UITextWithCmd  
  
        </component-class>  
  
    </component>  
  
    ...
```

本书属于个人整理,书中难免小疵,请大家指正,本人不胜感激!!!