# day13 jspEL表达式EL函数 java web之设计模式

* JSTL标签库（重点）
* 自定义标签（理解）
* MVC设计模式（重点中的重点）
* Java三层框架（重点中的重点）

## JSTL标签库

###### 1　什么是JSTL

JSTL是apache对EL表达式的扩展（也就是说JSTL依赖EL），JSTL是标签语言！JSTL标签使用以来非常方便，它与JSP动作标签一定，只不过它不是JSP内置的标签，需要我们自己导包，以及指定标签库而已！

如果你使用MyEclipse开发JavaWeb，那么在把项目发布到Tomcat时，你会发现，MyEclipse会在lib目录下存放jstl的Jar包！如果你没有使用MyEclipse开发那么需要自己来导入这个JSTL的Jar包：jstl-1.2.jar。

###### 2　JSTL标签库

JSTL一共包含四大标签库：

* core：核心标签库，我们学习的重点；
* fmt：格式化标签库，只需要学习两个标签即可；
* sql：数据库标签库，不需要学习了，它过时了；
* xml：xml标签库，不需要学习了，它过时了。

###### 3　使用taglib指令导入标签库

除了JSP动作标签外，使用其他第三方的标签库都需要：

* 导包；
* 在使用标签的JSP页面中使用taglib指令导入标签库；

下面是导入JSTL的core标签库：

<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

* prefix="c"：指定标签库的前缀，这个前缀可以随便给值，但大家都会在使用core标签库时指定前缀为c；
* uri="http://java.sun.com/jstl/core"：指定标签库的uri，它不一定是真实存在的网址，但它可以让JSP找到标签库的描述文件；

###### 4　core标签库常用标签

4.1　out和set

　out

|  |  |
| --- | --- |
| <c:out value=”aaa”/> | 输出aaa字符串常量 |
| <c:out value=”${aaa}”/> | 与${aaa}相同 |
| <c:out value=”${aaa}” default=”xxx”/> | 当${aaa}不存在时，输出xxx字符串 |
| <%  request.setAttribute("a","<script>alert('hello');</script>");  %>  <c:out value="${a }" default="xxx" escapeXml="false" /> | 当escapeXml为false，不会转换“<”、“>”。这可能会受到JavaScript攻击。 |

set

|  |  |
| --- | --- |
| <c:set var=”a” value=”hello”/> | 在pageContext中添加name为a，value为hello的数据。 |
| <c:set var=”a” value=”hello” scope=”session”/> | 在session中添加name为a，value为hello的数据。 |

4.2　remove

|  |  |
| --- | --- |
| <%  pageContext.setAttribute("a", "pageContext");  request.setAttribute("a", "session");  session.setAttribute("a", "session");  application.setAttribute("a", "application");  %>  <c:remove var="a"/>  <c:out value="${a }" default="none"/> | 删除所有域中name为a的数据！ |
| <c:remove var="a" scope=”page”/> | 删除pageContext中name为a的数据！ |

4.3　url

　　url标签会在需要URL重写时添加sessionId。

|  |  |
| --- | --- |
| <c:url value="/"/> | 输出上下文路径：/day08\_01/ |
| <c:url value="/" var="a" scope="request"/> | 把本该输出的结果赋给变量a。范围为request |
| <c:url value="/AServlet"/> | 输出：/day08\_01/AServlet |
| <c:url value="/AServlet">  <c:param name="username" value="abc"/>  <c:param name="password" value="123"/>  </c:url> | 输出：/day08\_01/AServlet?username=abc&password=123  如果参数中包含中文，那么会自动使用URL编码！ |

4.4　if

　　if标签的test属性必须是一个boolean类型的值，如果test的值为true，那么执行if标签的内容，否则不执行。

|  |
| --- |
| <c:set var=*"a"* value=*"hello"*/>  <c:if test="${**not empty** a }">  <c:out value="${a }"/>  </c:if> |

4.5　choose

choose标签对应Java中的if/else if/else结构。when标签的test为true时，会执行这个when的内容。当所有when标签的test都为false时，才会执行otherwise标签的内容。

|  |
| --- |
| <c:set var=*"score"* value="${param.score }"/>  <c:choose>  <c:when test="${score > 100 || score < 0}">错误的分数：${score }</c:when>  <c:when test="${score >= 90 }">A级</c:when>  <c:when test="${score >= 80 }">B级</c:when>  <c:when test="${score >= 70 }">C级</c:when>  <c:when test="${score >= 60 }">D级</c:when>  <c:otherwise>E级</c:otherwise>  </c:choose> |

4.6　forEach

forEach当前就是循环标签了，forEach标签有多种两种使用方式：

* 使用循环变量，指定开始和结束值，类似for(int i = 1; i <= 10; i++) {}；
* 循环遍历集合，类似for(Object o : 集合)；

循环变量方式：

|  |
| --- |
| <c:set var=*"sum"* value=*"0"* />  <c:forEach var=*"i"* begin=*"1"* end=*"10"*>  <c:set var=*"sum"* value="${sum + i}" />  </c:forEach>  <c:out value="*sum =* ${sum }"/> |
| <c:set var=*"sum"* value=*"0"* />  <c:forEach var=*"i"* begin=*"1"* end=*"10"* step=*"2"*>  <c:set var=*"sum"* value="${sum + i}" />  </c:forEach>  <c:out value="*sum =* ${sum }"/> |

遍历集合或数组方式：

|  |
| --- |
| <%  String[] names = {"zhangSan", "liSi", "wangWu", "zhaoLiu"};  pageContext.setAttribute("ns", names);  %>  <c:forEach var=*"item"* items="${ns }">  <c:out value="*name:* ${item }"/><br/>  </c:forEach> |

遍历List

|  |
| --- |
| <%  List<String> names = **new** ArrayList<String>();  names.add("zhangSan");  names.add("liSi");  names.add("wangWu");  names.add("zhaoLiu");  pageContext.setAttribute("ns", names);  %>  <c:forEach var=*"item"* items="${ns }">  <c:out value="*name:* ${item }"/><br/>  </c:forEach> |

遍历Map

|  |
| --- |
| <%  Map<String,String> stu = **new** LinkedHashMap<String,String>();  stu.put("number", "N\_1001");  stu.put("name", "zhangSan");  stu.put("age", "23");  stu.put("sex", "male");  pageContext.setAttribute("stu", stu);  %>  <c:forEach var=*"item"* items="${stu }">  <c:out value="${item.key }*:* ${item.value }"/><br/>  </c:forEach> |

forEach标签还有一个属性：varStatus，这个属性用来指定接收“循环状态”的变量名，例如：<forEach varStatus=”vs” …/>，这时就可以使用vs这个变量来获取循环的状态了。

* count：int类型，当前以遍历元素的个数；
* index：int类型，当前元素的下标；
* first：boolean类型，是否为第一个元素；
* last：boolean类型，是否为最后一个元素；
* current：Object类型，表示当前项目。

|  |
| --- |
| <c:forEach var=*"item"* items="${ns }" varStatus=*"vs"*>  <c:if test="${vs.first }">第一行：</c:if>  <c:if test="${vs.last }">最后一行：</c:if>  <c:out value="*第*${vs.count }*行:* "/>  <c:out value="*[*${vs.index }*]:* "/>  <c:out value="*name:* ${vs.current }"/><br/>  </c:forEach> |

###### 5　fmt标签库常用标签

fmt标签库是用来格式化输出的，通常需要格式化的有时间和数字。

格式化时间：

|  |
| --- |
| <%@ taglib prefix=*"fmt"* uri=*"http://java.sun.com/jsp/jstl/fmt"* %>  ......  <%  Date date = **new** Date();  pageContext.setAttribute("d", date);  %>  <fmt:formatDate value="${d }" pattern=*"yyyy-MM-dd HH:mm:ss"*/> |

格式化数字：

|  |
| --- |
| <%  **double** d1 = 3.5;  **double** d2 = 4.4;  pageContext.setAttribute("d1", d1);  pageContext.setAttribute("d2", d2);  %>  <fmt:formatNumber value="${d1 }" pattern=*"0.00"*/><br/>  <fmt:formatNumber value="${d2 }" pattern=*"#.##"*/> |

## 自定义标签

###### 1　自定义标签概述

1.1　自定义标签的步骤

其实我们在JSP页面中使用标签就等于调用某个对象的某个方法一样，例如：<c:if test=””>，这就是在调用对象的方法一样。自定义标签其实就是自定义类一样！

* 定义标签处理类：必须是Tag或SimpleTag的实现类；
* 编写标签库描述符文件（TLD）；

SimpleTag接口是JSP2.0中新给出的接口，用来简化自定义标签，所以现在我们基本上都是使用SimpleTag。

Tag是老的，传统的自定义标签时使用的接口，现在不建议使用它了。

1.2　SimpleTag接口介绍

SimpleTag接口内容如下：

* void doTag()：标签执行方法；
* JspTag getParent()：获取父标签；
* void setParent(JspTag parent)：设置父标签
* void setJspContext(JspContext context)：设置PageContext
* void setJspBody(JspFragment jspBody)：设置标签体对象；

请记住，万物皆对象！在JSP页面中的标签也是对象！你可以通过查看JSP的“真身”清楚的知道，所有标签都会变成对象的方法调用。标签对应的类我们称之为“标签处理类”！

标签的生命周期：

1. 当容器（Tomcat）第一次执行到某个标签时，会创建标签处理类的实例；
2. 然后调用setJspContext(JspContext)方法，把当前JSP页面的pageContext对象传递给这个方法；
3. 如果当前标签有父标签，那么使用父标签的标签处理类对象调用setParent(JspTag)方法；
4. 如果标签有标签体，那么把标签体转换成JspFragment对象，然后调用setJspBody()方法；
5. 每次执行标签时，都调用doTag()方法，它是标签处理方法。

HelloTag.java

|  |
| --- |
| **public** **class** HelloTag **implements** SimpleTag {  **private** JspTag parent;  **private** PageContext pageContext;  **private** JspFragment jspBody;    **public** **void** doTag() **throws** JspException, IOException {  pageContext.getOut().print("Hello Tag!!!");  }  **public** **void** setParent(JspTag parent) {  **this**.parent = parent;  }  **public** JspTag getParent() {  **return** **this**.parent;  }  **public** **void** setJspContext(JspContext pc) {  **this**.pageContext = (PageContext) pc;  }  **public** **void** setJspBody(JspFragment jspBody) {  **this**.jspBody = jspBody;  }  } |

1.3　标签库描述文件（TLD）

标签库描述文件是用来描述当前标签库中的标签的！标签库描述文件的扩展名为tld，你可以把它放到WEB-INF下，这样就不会被客户端直接访问到了。

hello.tld

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <taglib version=*"2.0"* xmlns=*"http://java.sun.com/xml/ns/j2ee"*  xmlns:xml=*"http://www.w3.org/XML/1998/namespace"*  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-jsptaglibrary\_2\_0.xsd "*>  <tlib-version>1.0</tlib-version>  <short-name>itcast</short-name>  <uri>http://www.itcast.cn/tags</uri>  <tag>  <name>hello</name>  <tag-class>cn.itcast.tag.HelloTag</tag-class>  <body-content>empty</body-content>  </tag>  </taglib> |

1.4　使用标签

在页面中使用标签分为两步：

* 使用taglib导入标签库；
* 使用标签；

|  |
| --- |
| <%@ taglib prefix=*"it"* uri=*"/WEB-INF/hello.tld"* %>  ......  <it:hello/> |

###### 2　自定义标签进阶

2.1　继承SimpleTagSupport

　　继承SimpleTagSuppport要比实现SimpleTag接口方便太多了，现在你只需要重写doTag()方法，其他方法都已经被SimpleTagSuppport完成了。

|  |
| --- |
| **public** **class** HelloTag **extends** SimpleTagSupport {  **public** **void** doTag() **throws** JspException, IOException {  **this**.getJspContext().getOut().write("<p>Hello SimpleTag!</p>");  }  } |

2.2　有标签体的标签

我们先来看看标签体内容的可选值：

<body-content>元素的可选值有：

* empty：无标签体。
* JSP：传统标签支持它，**SimpleTag已经不再支持使用<body-content>JSP</body-content>。**标签体内容可以是任何东西：EL、JSTL、<%=%>、<%%>，以及html；
* scriptless：标签体内容不能是Java脚本，但可以是EL、JSTL等。在SimpleTag中，**如果需要有标签体，那么就使用该选项**；
* tagdependent：标签体内容不做运算，由标签处理类自行处理，无论标签体内容是EL、JSP、JSTL，都不会做运算。**这个选项几乎没有人会使用！**

自定义有标签体的标签需要：

* 获取标签体对象：JspFragment jspBody = getJspBody();；
* 把标签体内容输出到页面：jspBody.invoke(null)；
* tld中指定标签内容类型：scriptless。

|  |
| --- |
| **public** **class** HelloTag **extends** SimpleTagSupport {  **public** **void** doTag() **throws** JspException, IOException {  PageContext pc = (PageContext) **this**.getJspContext();  HttpServletRequest req = (HttpServletRequest) pc.getRequest();  String s = req.getParameter("exec");  **if**(s != **null** && s.endsWith("true")) {  JspFragment body = **this**.getJspBody();  body.invoke(null);  }  }  } |
| <tag>  <name>hello</name>  <tag-class>cn.itcast.tags.HelloTag</tag-class>  <body-content>scriptless</body-content>  </tag> |
| <itcast:hello>  <h1>哈哈哈~</h1>  </itcast:hello> |

2.3　不执行标签下面的页面内容

　　如果希望在执行了自定义标签后，不再执行JSP页面下面的东西，那么就需要在doTag()方法中使用SkipPageException。

|  |
| --- |
| **public** **class** SkipTag **extends** SimpleTagSupport {  **public** **void** doTag() **throws** JspException, IOException {  **this**.getJspContext().getOut().print("<h1>只能看到我！</h1>");  **throw** **new** SkipPageException();  }  } |
| <tag>  <name>skip</name>  <tag-class>cn.itcast.tags.SkipTag</tag-class>  <body-content>empty</body-content>  </tag> |
| <itcast:skip/>  <h1>看不见我！</h1> |

2.4　带有属性的标签

　　一般标签都会带有属性，例如<c:if test=””>，其中test就是一个boolean类型的属性。完成带有属性的标签需要：

* 在处理类中给出JavaBean属性（提供get/set方法）；
* 在TLD中部属相关属性。

|  |
| --- |
| **public** **class** IfTag **extends** SimpleTagSupport {  **private** **boolean** test;  **public** **boolean** isTest() {  **return** test;  }  **public** **void** setTest(**boolean** test) {  **this**.test = test;  }  @Override  **public** **void** doTag() **throws** JspException, IOException {  **if**(test) {  **this**.getJspBody().invoke(**null**);  }  }  } |
| <tag>  <name>if</name>  <tag-class>cn.itcast.tag.IfTag</tag-class>  <body-content>scriptless</body-content>  <attribute>  <name>test</name>  <required>true</required>  <rtexprvalue>true</rtexprvalue>  </attribute>  </tag> |
| <%  pageContext.setAttribute("one", **true**);  pageContext.setAttribute("two", **false**);  %>  <it:if test="${one }">xixi</it:if>  <it:if test="${two }">haha</it:if>  <it:if test=*"true"*>hehe</it:if> |

## MVC

###### 1　MVC设计模式



MVC设计模式

MVC模式（Model-View-Controller）是[软件工程](http://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E5%B7%A5%E7%A8%8B)中的一种[软件架构](http://zh.wikipedia.org/wiki/%E8%BD%AF%E4%BB%B6%E6%9E%B6%E6%9E%84)模式，把软件系统分为三个基本部分：模型（Model）、视图（View）和控制器（Controller）。

MVC模式最早为[Trygve Reenskaug](http://zh.wikipedia.org/w/index.php?title=Trygve_Reenskaug&action=edit&redlink=1)提出，为[施乐帕罗奥多研究中心](http://zh.wikipedia.org/wiki/%E5%B8%95%E7%BE%85%E5%A5%A7%E5%A4%9A%E7%A0%94%E7%A9%B6%E4%B8%AD%E5%BF%83)（Xerox PARC）的[Smalltalk](http://zh.wikipedia.org/wiki/Smalltalk)语言发明的一种软件设计模式。

MVC可对程序的后期维护和扩展提供了方便，并且使程序某些部分的重用提供了方便。而且MVC也使程序简化，更加直观。

* 控制器Controller：对请求进行处理，负责请求转发；
* 视图View：界面设计人员进行图形界面设计；
* 模型Model：程序编写程序应用的功能（实现算法等等）、数据库管理；

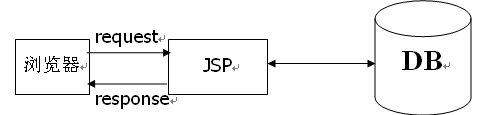
注意，MVC不是Java的东西，几乎现在所有B/S结构的软件都采用了MVC设计模式。但是要注意，MVC在B/S结构软件并没有完全实现，例如在我们今后的B/S软件中并不会有事件驱动！

###### 2　JavaWeb与MVC

　　JavaWeb的经历了JSP Model1、JSP Model1二代、JSP Model2三个时期。

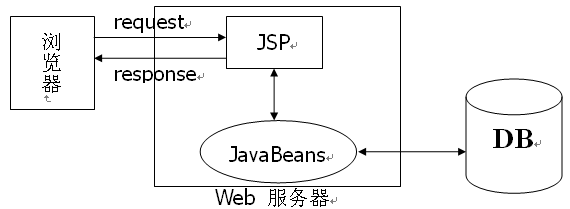
2.1　JSP Model1第一代

JSP Model1是JavaWeb早期的模型，它适合小型Web项目，开发成本低！Model1第一代时期，服务器端只有JSP页面，所有的操作都在JSP页面中，连访问数据库的API也在JSP页面中完成。也就是说，所有的东西都耦合在一起，对后期的维护和扩展极为不利。



2.2　JSP Model1第二代

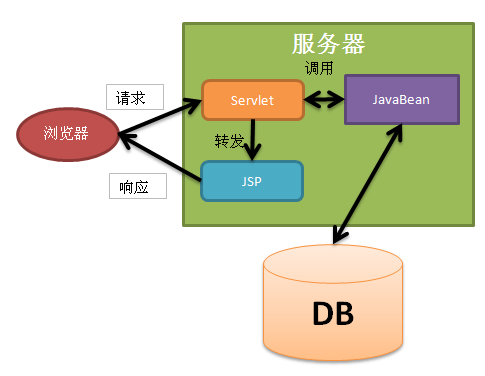
　　JSP Model1第二代有所改进，把业务逻辑的内容放到了JavaBean中，而JSP页面负责显示以及请求调度的工作。虽然第二代比第一代好了些，但还让JSP做了过多的工作，JSP中把视图工作和请求调度（控制器）的工作耦合在一起了。



2.3　JSP Model2

JSP Model2模式已经可以清晰的看到MVC完整的结构了。

* JSP：视图层，用来与用户打交道。负责接收用来的数据，以及显示数据给用户；
* Servlet：控制层，负责找到合适的模型对象来处理业务逻辑，转发到合适的视图；
* JavaBean：模型层，完成具体的业务工作，例如：开启、转账等。



JSP Model2适合多人合作开发大型的Web项目，各司其职，互不干涉，有利于开发中的分工，有利于组件的重用。但是，Web项目的开发难度加大，同时对开发人员的技术要求也提高了。

## JavaWeb经典三层框架

我们常说的三层框架是由JavaWeb提出的，也就是说这是JavaWeb独有的！

所谓三层是表述层（WEB层）、业务逻辑层（Business Logic），以及数据访问层（Data Access）。

* WEB层：包含JSP和Servlet等与WEB相关的内容；
* 业务层：业务层中不包含JavaWeb API，它只关心业务逻辑；
* 数据层：封装了对数据库的访问细节；

　　注意，在业务层中不能出现JavaWeb API，例如request、response等。也就是说，业务层代码是可重用的，甚至可以应用到非Web环境中。业务层的每个方法可以理解成一个万能，例如转账业务方法。业务层依赖数据层，而Web层依赖业务层！

