# JSP

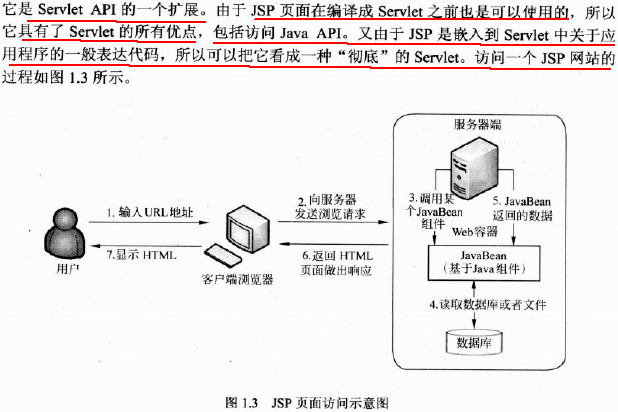
## 1.JSP入门

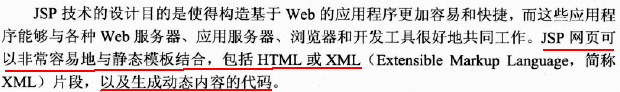
只有Servlet而没有JSP的年代！程序员天天都需要打大量的out.print(“<html>”);

我们希望编程时什么样的呢?前端人员已经搞定html页面，我们希望把html添加到Servlet中的这个动作是自动的，而不是人工的！response.getWriter().print(“<html>”);这就引出了JSP;

### 1.1 JSP概述

Java Server Page(JSP)是一种实现普通静态HTML和动态HTML混合编码的技术,



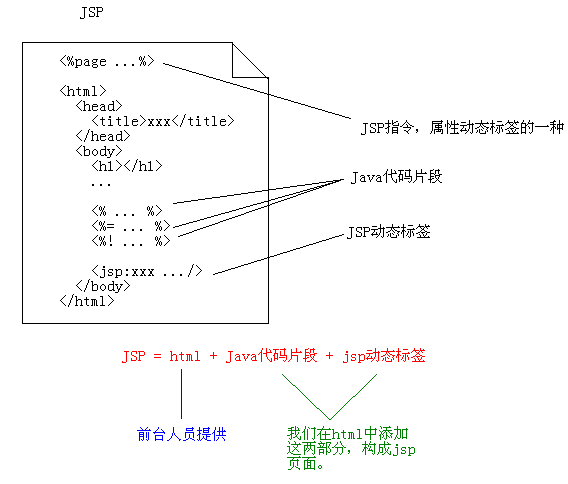


#### 1.1.1　什么是JSP

JSP（Java Server Pages）是JavaWeb服务器端的**动态资源**。它与html页面的作用是相同的，**显示数据和获取数据**。

#### 1.1.2　JSP的组成

JSP = html + Jsp脚本（代码片段） + JSP动态标签



### 1.2　JSP语法

* <%...%>：到真身中，没有改动！
* <html>：到真身中，out.write(“<html>”);
* <%=name%>：到真身中，out.print(name);
* <%! … %> ：到真身中，不在\_jspService()方法之内，它被类体直接包含！

#### 1.2.1　JSP脚本

JSP脚本就是Java代码片段，它分为三种：

<%...%>：Java语句；方法中能放什么，它就能放什么。

<%=…%>：Java表达式；同与resopnse.getWriter().print()。写到这里的东西，都是用来输出的！

* <%!...%>：Java定义类成员；很少有人会用它！有人会考它！定义类的成员！类能包含什么，它就可以包含什么。声明中没有9大内置对象！

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>JSP演示</title>  </head>  <body>  <h1>JSP演示</h1>  <%  // Java语句  String s1 = "hello jsp";  // 不会输出到客户端，而是在服务器端的控制台打印  System.out.println(s1);  %>  <!-- 输出到客户端浏览器上 -->  输出变量：<%=s1 %><br/>  输出int类型常量：<%=100 %><br/>  输出String类型常量：<%="你好" %><br/>  <br/>  使用表达式输出常量是很傻的一件事，因为可以直接使用html即可，下面是输出上面的常量：<br/>  100<br/>  你好  </body>  </html> |

#### 1.2.2　内置对象out

out对象在JSP页面中无需创建就可以使用，它的作用是用来向客户端输出。

|  |
| --- |
| <body>  <h1>out.jsp</h1>  <%  //向客户端输出  out.print("你好！");  %>  </body> |

其中<%=…%>与out.print()功能是相同的！它们都是向客户端输出，例如：

<%=s1%>等同于<% out.print(s1); %>

<%=”hello”%>等同于<% out.print(“hello”); %>，也等同于直接在页面中写hello一样。

#### 1.2.3　多个<%...%>可以通用

在一个JSP中多个<%...%>是相通的。例如：

|  |
| --- |
| <body>  <h1>out.jsp</h1>  <%  String s = "hello";  %>  <%  out.print(s);  %>  </body> |

循环打印表格：

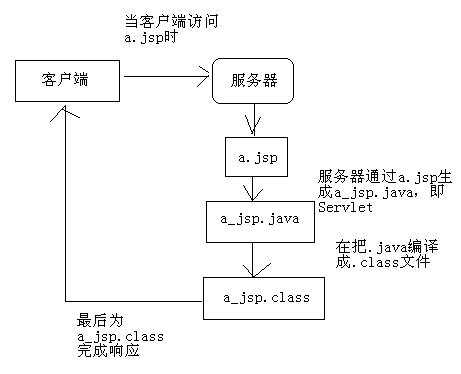
|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>test1.jsp</title>  </head>  <body>  <table border=*"1"* width=*"50%"*>  <tr>  <th>序号</th>  <th>用户名</th>  <th>密码</th>  </tr>  <% String username="username"; %>  <%=username %>  <% out.println(username); %>  <% **for**(**int** i=0;i<10;i++){ %>  <tr>  <td><%=i+1 %></td>  <td>user<%=i %></td>  <td><% out.println(100+1); %></td>  </tr>  <%} %>  </table>  </body>  </html> |



### 1.3　JSP的原理

#### 1.3.1　JSP是特殊的Servlet

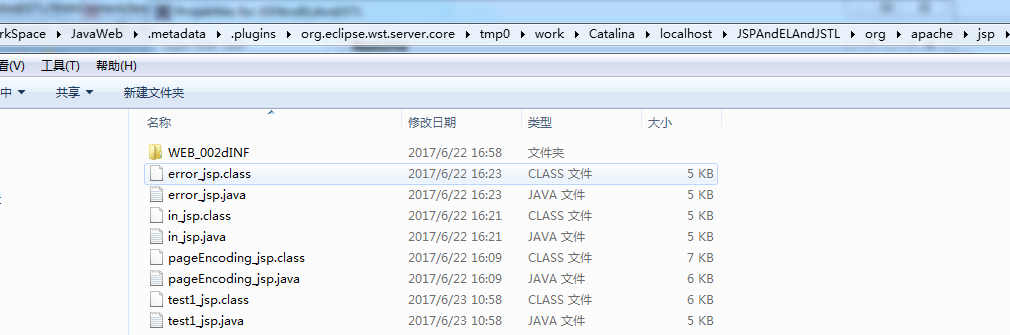
　　JSP是一种特殊的Servlet，当JSP页面首次被访问时，容器（Tomcat）会先把JSP编译成Servlet，然后再去执行Servlet。所以JSP其实就是一个Servlet！



**jsp的原理**

**\* 当第一次访问一个jsp页面时，tomcat会读取这个jsp，然后把它变成一个.java文件**

**\* 这个文件在work目录下**



**\* 然后会把它编码成一个.class**

**\* 创建这个类的对象！**

**\* 调用它的service()方法**

**\* 第二次或第N次访问jsp**

**> 直接调用对象的service()方法！**

#### 1.3.2　JSP真身存放目录

JSP生成的Servlet存放在${CATALANA}/work目录下，我经常开玩笑的说，它是JSP的“真身”。我们打开看看其中的内容，了解一下JSP的“真身”。

你会发现，在JSP中的静态信息（例如<html>等）在“真身”中都是使用out.write()完成打印！这些静态信息都是作为字符串输出给了客户端。

JSP的整篇内容都会放到名为\_jspService的方法中！你可能会说<@page>不在“真身”中，<%@page>我们下面再说。

a\_jsp.java的\_jspService()方法：

|  |
| --- |
| <%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>My JSP 'a.jsp' starting page</title>  <meta http-equiv="pragma" content="no-cache">  <meta http-equiv="cache-control" content="no-cache">  <meta http-equiv="expires" content="0">  <meta http-equiv="keywords" content="keyword1,keyword2,keyword3">  <meta http-equiv="description" content="This is my page">  <!--  <link rel="stylesheet" type="text/css" href="styles.css">  -->  </head>  <body>  <h1>声明</h1>  <%!  int a = 10;  %>  <%a++;%>  <%=a %>  </body>  </html> |

|  |
| --- |
| 在<%=%>和<%%>中不用创建即可使用的对象！  public void \_jspService(final javax.servlet.http.HttpServletRequest request,  在<%=%>和<%%>中不用创建即可使用的对象！  final javax.servlet.http.HttpServletResponse response)  throws java.io.IOException, javax.servlet.ServletException {  <%=%>和<%%>中不用创建即可使用的对象！  final javax.servlet.jsp.PageContext pageContext;  在<%=%>和<%%>中不用创建即可使用的对象！  javax.servlet.http.HttpSession session = null;  在<%=%>和<%%>中不用创建即可使用的对象！  final javax.servlet.ServletContext application;  在<%=%>和<%%>中不用创建即可使用的对象！  final javax.servlet.ServletConfig config;  在<%=%>和<%%>中不用创建即可使用的对象！  javax.servlet.jsp.JspWriter out = null;  在<%=%>和<%%>中不用创建即可使用的对象！  final java.lang.Object page = this;  javax.servlet.jsp.JspWriter \_jspx\_out = null;  javax.servlet.jsp.PageContext \_jspx\_page\_context = null;  try {  response.setContentType("text/html;charset=UTF-8");  pageContext = \_jspxFactory.getPageContext(this, request, response,  null, true, 8192, true);  \_jspx\_page\_context = pageContext;  application = pageContext.getServletContext();  config = pageContext.getServletConfig();  session = pageContext.getSession();  out = pageContext.getOut();  \_jspx\_out = out;  JSP页面中的内容都会在这个位置出现！这时上面所说的对象已经创建完了，所以在JSP页面中是可以使用的  …  } |

### 1.4 再论JSP脚本

JSP脚本一共三种形式：

* <%...%>：内容会直接放到“真身”中；
* <%=…%>：内容会放到out.print()中，作为out.print()的参数；
* <%!…%>：内容会放到\_jspService()方法之外，被类直接包含；

前面已经讲解了<%...%>和<%=…%>，但还没有讲解<%!...%>的作用！

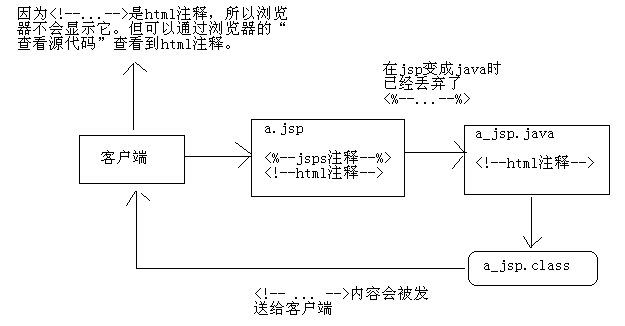
现在我们已经知道了，JSP其实就是一个类，一个Servlet类。<%!...%>的作用是在类中添加方法或成员的，所以<%!...%>中的内容不会出现在\_jspService()中。

|  |
| --- |
| <%!  **private** String name;  **public** String hello() {  **return** "hello JSP!";  }  %> |

### 1.5　JSP注释

我们现在已经知道JSP是需要先编译成.java，再编译成.class的。其中<%-- ... --%>中的内容在JSP编译成.java时会被忽略的，即JSP注释。

也可以在JSP页面中使用html注释：<!-- … -->，但这个注释在JSP编译成的.java中是存在的，它不会被忽略，而且会被发送到客户端浏览器。但是在浏览器显示服务器发送过来的html时，因为<!-- … -->是html的注释，所以浏览器是不会显示它的。



## 2.JSP指令

### 2.1 JSP指令概述

#### 2.1.1 语法格式

JSP指令的语法格式：<%@指令名 attr1=”” attr2=”” %>，一般都会把JSP指令放到JSP文件的最上方，但这不是必须的。

|  |
| --- |
| <%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>  <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %> |

#### 2.1.2 JSP中的指令

JSP中有三大指令：page、include、taglib，最为常用，也最为复杂的就是page指令了。

### 2.2　page指令

page指令是最为常用的指定，也是属性最多的属性！

page指令没有必须属性，都是可选属性。例如<%@page %>，没有给出任何属性也是可以的！

**在JSP页面中，任何指令都可以重复出现！**

<%@ page language=”java”%>

<%@ page import=”java.util.\*”%>

<%@ page pageEncoding=”utf-8”%>

这也是可以的！

#### 2.2.1　page指令的pageEncoding和contentType（重点）

pageEncoding指定当前JSP页面的编码！这个编码是给服务器看的，服务器需要知道当前JSP使用的编码，不然服务器无法正确把JSP编译成java文件。所以这个编码只需要与真实的页面编码一致即可！在MyEclipse中，在JSP文件上点击右键，选择属性就可以看到当前JSP页面的编码了。

contentType属性与response.setContentType()方法的作用相同！它会完成两项工作，一是设置响应字符流的编码，二是设置content-type响应头。例如：<%@ contentType=”text/html;charset=utf-8”%>，它会使“真身”中出现response.setContentType(“text/html;charset=utf-8”)。

无论是page指令的pageEncoding还是contentType，它们的默认值都是ISO-8859-1，我们知道ISO-8859-1是无法显示中文的，所以JSP页面中存在中文的话，一定要设置这两个属性。

其实pageEncoding和contentType这两个属性的关系很“暧昧”：

* 当设置了pageEncoding，而没设置contentType时： contentType的默认值为pageEncoding；
* 当设置了contentType，而没设置pageEncoding时： pageEncoding的默认值与contentType；

**也就是说，当pageEncoding和contentType只出现一个时，那么另一个的值与出现的值相同。如果两个都不出现，那么两个属性的值都是ISO-8859-1。所以通过我们至少设置它们两个其中一个！**

#### 2.2.2　page指令的import属性

import是page指令中一个很特别的属性！import属性值对应“真身”中的import语句。**import属性值可以使逗号隔开**：<%@page import=”java.net.\*,java.util.\*,java.sql.\*”%>,import属性是唯一可以重复出现的属性：

<%@page import=”java.util.\*” import=”java.net.\*” import=”java.sql.\*”%>

**但是，我们一般会使用多个page指令来导入多个包：**

<%@ page import=”java.util.\*”%>

<%@ page import=”java.net.\*”%>

<%@ page import=”java.text.\*”%>

#### 2.2.3　page指令的errorPage和isErrorPage

我们知道，在一个JSP页面出错后，Tomcat会响应给用户错误信息（500页面）！如果你不希望Tomcat给用户输出错误信息，那么可以使用page指令的errorPage来指定错误页！也就是自定义错误页面，例如：<%@page errorPage=”xxx.jsp”%>。这时，在当前JSP页面出现错误时，会请求转发到xxx.jsp页面。

a.jsp

|  |
| --- |
| <%@ page import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <%@ page errorPage=*"b.jsp"* %>  <%  **if**(**true**)  **throw** **new** Exception("哈哈~");  %> |

b.jsp

|  |
| --- |
| <%@ page pageEncoding=*"UTF-8"*%>  <html>  <body>  <h1>出错啦！</h1>  </body>  </html> |

在上面代码中，a.jsp抛出异常后，会请求转发到b.jsp。在浏览器的地址栏中还是a.jsp，因为是请求转发！

而且客户端浏览器收到的响应码为200，表示请求成功！如果希望客户端得到500，那么需要指定b.jsp为错误页面。

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  设置b.jsp页面为错误页，在错误页中就可以使用exception隐藏对象了。  注意：一旦转发到错误页，那么Tomcat会把状态码设置为500，而不在是200了。  <%@ page isErrorPage=*"true"* %>  <html>  <body>  <h1>出错啦！</h1>  <%=exception.getMessage() %>  </body>  </html> |

注意，当isErrorPage为true时，说明当前JSP为错误页面，即专门处理错误的页面。那么这个页面中就可以使用一个内置对象exception了。其他页面是不能使用这个内置对象的！

温馨提示：*IE会在状态码为500时，并且响应正文的长度小于等于512B时不给予显示！而是显示“网站无法显示该页面”字样。这时你只需要添加一些响应内容即可，例如上例中的b.jsp中我给出一些内容，IE就可以正常显示了！*

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@page errorPage=*"error.jsp"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>in.jsp</title>  </head>  <body>  <%  **int** i = 10/0;  %>  </body>  </html> |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@page isErrorPage=*"true"* %>>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>error.jsp</title>  </head>  <body>  <h>这是错误页面,替代Tomcat为我们生成的错误页面!</h>  <%=exception.getMessage() %>>  </body>  </html> |

2.2.3.1　web.xml中配置错误页面

　　不只可以通过JSP的page指令来配置错误页面，还可以在web.xml文件中指定错误页面。这种方式其实与page指令无关，但想来想去还是在这个位置来讲解比较合适！

web.xml(通常都是在web.xml中进行错误配置，而不会为某一个页面进行错误页面的配置！)

|  |
| --- |
| <error-page>  <error-code>404</error-code>  <location>/error404.jsp</location>  </error-page>  <error-page>  <error-code>500</error-code>  <location>/error500.jsp</location>  </error-page>  如果没有配置这个<error-page>，在抛出RuntimeException时会转发到error500.jsp页面。即匹配响应码500  <error-page>  <exception-type>java.lang.RuntimeException</exception-type>  <location>/error.jsp</location>  </error-page> |

<error-page>有两种使用方式：

* <error-code>和<location>子元素；
* <exception-type>和<location>子元素；

　　其中<error-code>是指定响应码；<location>指定转发的页面；<exception-type>是指定抛出的异常类型。

在上例中：

* 当出现404时，会跳转到error404.jsp页面；
* 当出现RuntimeException异常时，会跳转到error.jsp页面；
* 当出现非RuntimeException的异常时，会跳转到error500.jsp页面。

　　这种方式会在控制台看到异常信息！而使用page指令时不会在控制台打印异常信息。

#### 2.2.4　page指令的autoFlush和buffer

buffer表示当前JSP的输出流（out隐藏对象）的缓冲区大小，**默认为8kb**。

autoFlush表示在out对象的缓冲区满时如果处理！当autoFlush为true时，表示缓冲区满时把缓冲区数据输出到客户端；当authFlush为false时，表示缓冲区满时，抛出异常。**authFlush的默认值为true**。

　　这两个属性一般我们也不会去特意设置，都是保留默认值！

#### 2.2.5　page指令的isELIgnored

　　后面我们会讲解EL表达式语言，page指令的isElIgnored属性表示当前JSP页面是否忽略EL表达式，默认值为false，表示不忽略（即支持）。

#### 2.2.6　page指令的其他属性

* language：**只能是Java**，这个属性可以看出JSP最初设计时的野心！希望JSP可以转换成其他语言！但是，到现在JSP也只能转换成Java代码；
* info：JSP说明性信息；

isThreadSafe：默认为false，为true时，JSP生成的Servlet会去实现一个过时的标记接口SingleThreadModel(已经作废了！)，这时JSP就只能处理单线程的访问；

* session：默认为true，表示当前JSP页面可以使用session对象，如果为false表示当前JSP页面不能使用session对象；
* extends：指定当前JSP页面生成的Servlet的父类；

#### 2.2.7　<jsp-config>（了解）

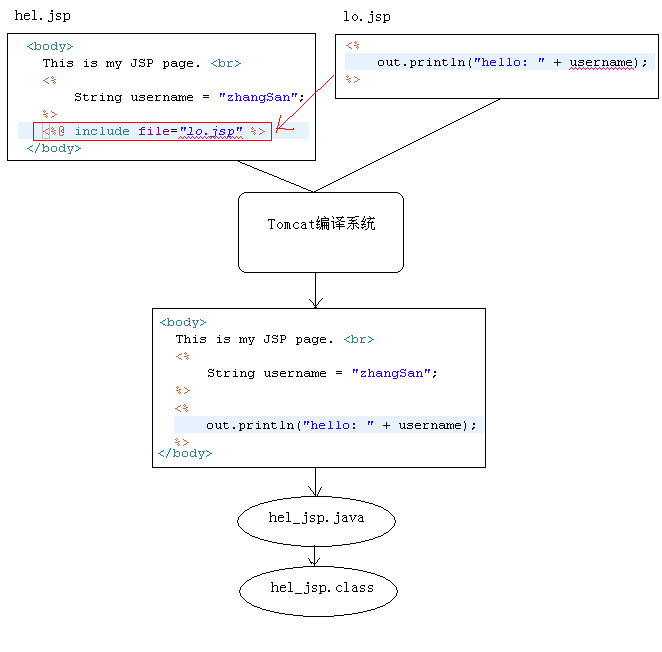
在web.xml页面中配置<jsp-config>也可以完成很多page指定的功能！

|  |
| --- |
| <jsp-config>  <jsp-property-group>  对所有jsp进行配置  <url-pattern>\*.jsp</url-pattern>  忽略EL表达式  <el-ignored>true</el-ignored>  指定页面编码为utf-8  <page-encoding>UTF-8</page-encoding>  禁用Java脚本！如果在JSP页面中使用了Java脚本就会抛出异常。  <scripting-invalid>true</scripting-invalid>  </jsp-property-group>  </jsp-config> |

### 2.3　include指令

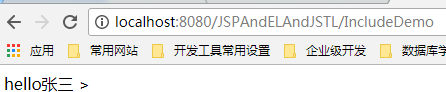
include指令表示静态包含！即目的是把多个JSP合并成一个JSP文件！include指令只有一个属性：file，指定要包含的页面，例如：<%@include file=”b.jsp”%>。

静态包含：当hel.jsp页面包含了lo.jsp页面后，在编译hel.jsp页面时，需要把hel.jsp和lo.jsp页面合并成一个文件，然后再编译成Servlet（Java文件）。



很明显，在ol.jsp中在使用username变量，而这个变量在hel.jsp中定义的，所以只有这两个JSP文件合并后才能使用。通过include指定完成对它们的合并！

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>Insert title here</title>  </head>  <body>  <%String username = "张三"; %>  <%@ include file=*'b.jsp'* %>>  </body>  </html> |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>Insert title here</title>  </head>  <body>  <%out.println("hello"+ username);%>  </body>  </html> |



### 2.4　taglib指令

这个指令需要在学习了自定义标签后才会使用，现在只能做了了解而已！在JSP页面中使用第三方的标签库时，需要使用taglib指令来“导包”。例如：

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

其中prefix表示标签的前缀，这个名称可以随便起。uri是由第三方标签库定义的，所以你需要知道第三方定义的uri。

## 3.JSP九大内置对象

### 3.1　什么是JSP九大内置对象

在JSP中无需创建就可以使用的9个对象，它们是：

* out（JspWriter）：等同与response.getWriter()，用来向客户端发送文本数据；

config（ServletConfig）：(基本没用)对应“真身”中的ServletConfig；

* page（当前JSP的真身类型）(Object page = this;，真的没用)：当前JSP页面的“this”，即当前对象；
* pageContext（PageContext）：页面上下文对象，它是最后一个没讲的域对象；
* exception（Throwable）：(isErrorPage=”true”时，这个页面才有exceiption对象)只有在错误页面中可以使用这个对象；
* request（HttpServletRequest）：即HttpServletRequest类的对象；
* response（HttpServletResponse）：即HttpServletResponse类的对象；
* application（ServletContext）：即ServletContext类的对象；
* session（HttpSession）：即HttpSession类的对象，不是每个JSP页面中都可以使用，如果在某个JSP页面中设置<%@page session=”false”%>，说明这个页面不能使用session。

**在这9个对象中有很多是极少会被使用的，例如：config、page、exception基本不会使用。**

**在这9个对象中有两个对象不是每个JSP页面都可以使用的：exception、session。**

**在这9个对象中有很多前面已经学过的对象：out、request、response、application、session、config。**

### 3.2　通过“真身”来对照JSP

　　我们知道JSP页面的内容出现在“真身”的\_jspService()方法中，而在\_jspService()方法开头部分已经创建了9大内置对象。

|  |
| --- |
| //request和response两个对象是方法参数。  public void \_jspService(HttpServletRequest request, HttpServletResponse response)  throws java.io.IOException, ServletException {  //pageContext对象  PageContext pageContext = null;  //session对象  HttpSession session = null;  //application对象  ServletContext application = null;  //config对象  ServletConfig config = null;  //out对象  JspWriter out = null;  //page对象  Object page = this;  JspWriter \_jspx\_out = null;  PageContext \_jspx\_page\_context = null;  try {  response.setContentType("text/html;charset=UTF-8");  pageContext = \_jspxFactory.getPageContext(this, request, response,  null, true, 8192, true);  \_jspx\_page\_context = pageContext;  application = pageContext.getServletContext();  config = pageContext.getServletConfig();  session = pageContext.getSession();  out = pageContext.getOut();  \_jspx\_out = out;  JSP页面的内容从这里才开始，所以可以使用上面定义的变量！这就是9大内置对象为什么可以在JSP页面中无需创建就可以使用的奥秘了！  **从这里开始，才是JSP页面的内容**  }… |

### 3.3　pageContext对象

　　在JavaWeb中一共四个域对象，其中Servlet中可以使用的是request、session、application三个对象，而在JSP中可以使用pageContext、request、session、application四个域对象。

pageContext 对象是PageContext类型，它的主要功能有：

* 域对象功能；
* 代理其它域对象功能；
* 获取其他内置对象；

#### 3.3.1　域对象功能

pageContext也是域对象，它的范围是当前页面。它的范围也是四个域对象中最小的！

* void setAttribute(String name, Object value)；
* Object getAttrbiute(String name, Object value)；
* void removeAttribute(String name, Object value)；

#### 3.3.2　代理其它域对象功能

还可以使用pageContext来代理其它3个域对象的功能，也就是说可以使用pageContext向request、session、application对象中存取数据，例如：

|  |
| --- |
| //向pageContext中存储数据  pageContext.setAttribute("x", "X");  //向request中存储数据  pageContext.setAttribute("x", "XX", PageContext.REQUEST\_SCOPE);  //向session中存储数据  pageContext.setAttribute("x", "XXX", PageContext.SESSION\_SCOPE);  //向application中存储数据  pageContext.setAttribute("x","XXXX",PageContext.APPLICATION\_SCOPE); |

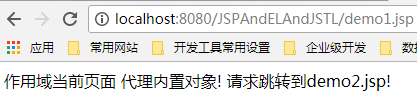
* void setAttribute(String name, Object value, int scope)：在指定范围中添加数据；
* Object getAttribute(String name, int scope)：获取指定范围的数据；
* void removeAttribute(String name, int scope)：移除指定范围的数据；
* Object findAttribute(String name)：依次在page、request、session、application范围查找名称为name的数据，如果找到就停止查找。这说明在这个范围内有相同名称的数据，那么page范围的优先级最高！

#### 3.3.3　获取其他内置对象

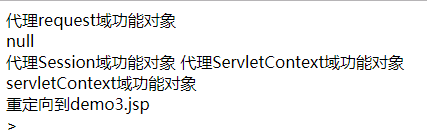
一个pageContext对象等于所有内置对象，即1个当9个。这是因为可以使用pageContext对象获取其它8个内置对象：

* JspWriter getOut()：获取out内置对象；
* ServletConfig getServletConfig()：获取config内置对象；
* Object getPage()：获取page内置对象；
* ServletRequest getRequest()：获取request内置对象；
* ServletResponse getResponse()：获取response内置对象；
* HttpSession getSession()：获取session内置对象；
* ServletContext getServletContext()：获取application内置对象；
* Exception getException()：获取exception内置对象；

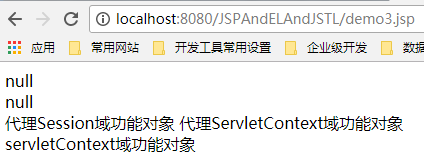
|  |
| --- |
| <%@page import=*"org.omg.PortableServer.REQUEST\_PROCESSING\_POLICY\_ID"*%>  <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@ page errorPage=*"error.jsp"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>demo1.jsp</title>  </head>  <body>  <% pageContext.setAttribute("pageContext", "作用域当前页面"); %>  <% pageContext.setAttribute("request", "代理request域功能对象",pageContext.REQUEST\_SCOPE); %>  <% pageContext.setAttribute("session", "代理Session域功能对象",pageContext.SESSION\_SCOPE); %>  <% pageContext.setAttribute("application","代理ServletContext域功能对象",pageContext.APPLICATION\_SCOPE); %>  <% application.setAttribute("ServletContext", "servletContext域功能对象"); %>  <% pageContext.getOut().println(pageContext.getAttribute("pageContext")+" 代理内置对象!"); %>  <%="请求跳转到demo2.jsp!" %>  </body>  </html> |



|  |
| --- |
| <%@page import=*"org.omg.PortableServer.REQUEST\_PROCESSING\_POLICY\_ID"*%>  <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@ page errorPage=*"error.jsp"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>demo1.jsp</title>  </head>  <body>  <% pageContext.setAttribute("pageContext", "作用域当前页面"); %>  <% pageContext.setAttribute("request", "代理request域功能对象",pageContext.REQUEST\_SCOPE); %>  <% pageContext.setAttribute("session", "代理Session域功能对象",pageContext.SESSION\_SCOPE); %>  <% pageContext.setAttribute("application","代理ServletContext域功能对象",pageContext.APPLICATION\_SCOPE); %>  <% application.setAttribute("ServletContext", "servletContext域功能对象"); %>  <% pageContext.getOut().println(pageContext.getAttribute("pageContext")+" 代理内置对象!"); %>  <%="请求跳转到demo2.jsp!" %>    <% request.getRequestDispatcher("/demo2.jsp").forward(request, response); %>  </body>  </html> |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@ page errorPage=*"error.jsp"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>demo2.jsp</title>  </head>  <body>  <% out.println(request.getAttribute("request")+"</br>") ;%>  <%= pageContext.getAttribute("pageContext")+"</br> "%>  <%= session.getAttribute("session")+" " %>  <%= application.getAttribute("application")+" </br> " %>  <%= application.getAttribute("ServletContext")+"</br>" %>  <%="重定向到demo3.jsp"+"</br>" %>>  </body>  </html> |



|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@ page errorPage=*"error.jsp"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>demo2.jsp</title>  </head>  <body>  <% out.println(request.getAttribute("request")+"</br>") ;%>  <%= pageContext.getAttribute("pageContext")+"</br> "%>  <%= session.getAttribute("session")+" " %>  <%= application.getAttribute("application")+" </br> " %>  <%= application.getAttribute("ServletContext")+"</br>" %>  <%="重定向到demo3.jsp"+"</br>" %>>  <% response.sendRedirect(("/JSPAndELAndJSTL/demo3.jsp")); %>  </body>  </html> |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@ page errorPage=*"error.jsp"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>demo3.jsp</title>  </head>  <body>  <% out.println(request.getAttribute("request")+"</br>") ;%>  <%= pageContext.getAttribute("pageContext")+"</br> "%>  <%= session.getAttribute("session")+" " %>  <%= application.getAttribute("application")+" </br> " %>  <%= application.getAttribute("ServletContext")+"</br>" %>  </body>  </html> |



## 4.JSP动作标签

### 4.1　JSP动作标签概述

动作标签的作用是用来简化Java脚本的！

JSP动作标签是JavaWeb内置的动作标签，它们是已经定义好的动作标签，我们可以拿来直接使用。

如果JSP动作标签不够用时，还可以使用自定义标签（后面单独介绍）。JavaWeb一共提供了20个JSP动作标签，但有很多基本没有用，这里只介绍一些有坐标的动作标签。

JSP动作标签的格式：<jsp:标签名 …>

### 4.2　<jsp:include>

<jsp:include>标签的作用是用来包含其它JSP页面的！你可能会说，前面已经学习了include指令了，它们是否相同呢？虽然它们都是用来包含其它JSP页面的，但它们的实现的级别是不同的！

include指令是在编译级别完成的包含，即把当前JSP和被包含的JSP合并成一个JSP，然后再编译成一个Servlet。

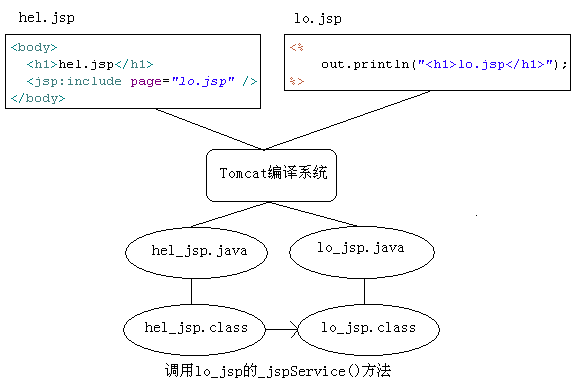
include动作标签是在运行级别完成的包含，即当前JSP和被包含的JSP都会各自生成Servlet，然后在执行当前JSP的Servlet时完成包含另一个JSP的Servlet。它与RequestDispatcher的include()方法是相同的！

hel.jsp

|  |
| --- |
| <body>  <h1>hel.jsp</h1>  <jsp:include page=*"lo.jsp"* />  </body> |

lo.jsp

|  |
| --- |
| <%  out.println("<h1>lo.jsp</h1>");  %> |



　　其实<jsp:include>在“真身”中不过是一句方法调用，即调用另一个Servlet而已。

### 4.3　<jsp:forward>

forward标签的作用是请求转发！forward标签的作用与RequestDispatcher#forward()方法相同。

hel.jsp

|  |
| --- |
|  |

lo.jsp

|  |
| --- |
| <%  out.println("<h1>lo.jsp</h1>");  %> |

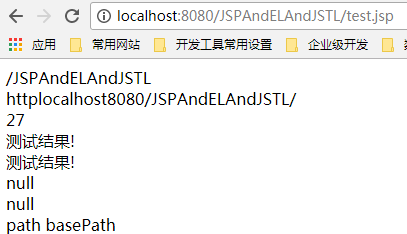
注意，最后客户端只能看到lo.jsp的输出，而看不到hel.jsp的内容。也就是说在hel.jsp中的<h1>hel.jsp</h1>是不会发送到客户端的。<jsp:forward>的作用是“别在显示我，去显示它吧！”。

### 4.4　<jsp:param>

还可以在<jsp:include>和<jsp:forward>标签中使用<jsp:param>子标签，它是用来传递参数的。下面用<jsp:include>来举例说明<jsp:param>的使用。

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>a.jsp</title>  </head>    <body>  <h1>a.jsp</h1>  <hr/>  <jsp:include page=*"/b.jsp"*>  //给被包含的页面b.jsp传递参数。  <jsp:param value=*"zhangSan"* name=*"username"*/>  </jsp:include>  </body>  </html> |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>b.jsp</title>  </head>    <body>  <h1>b.jsp</h1>  <hr/>  <%获取参数  String username = request.getParameter("username");  out.print("你好：" + username);  %>  </body>  </html> |

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>test.jsp</title>  </head>  <body>  <% String path = request.getContextPath(); %>  <% String basePath = request.getScheme()+request.getServerName()+request.getServerPort()+path+"/" ;%>  <% request.setAttribute("age", "27") ;%>  <% out.write("测试动作标签forward是否会携带request和response!"); %>  <%=path+"</br>" %>  <%=basePath+"</br>" %>  <jsp:include page=*"test1.jsp"* />  <jsp:forward page=*"/WEB-INF/test2.jsp"*>  <jsp:param value=*"path"* name=*"path"*/>  <jsp:param value=*"basePath"* name=*"basePath"*/>  </jsp:forward>  </body>  </html> |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>Insert title here</title>  </head>  <body>  <% String path = request.getContextPath(); %>  <% String basePath = request.getScheme()+request.getServerName()+request.getServerPort()+path+"/" ;%>  <%=path+"</br>" %>  <%=basePath+"</br>" %>  <%=request.getAttribute("age")+"</br>" %>  <%=" 测试结果!"+"</br>" %>  <%out.write(" 测试结果!"+"</br>"); %>  <%=request.getAttribute("path")+"</br>" %>  <%=request.getAttribute("basePath")+"</br>" %>  <%=request.getParameter("path") %>  <%=request.getParameter("basePath") %>    </body>  </html> |



## 5.JavaBean

### 5.1　JavaBean概述

#### 5.1.1　什么是JavaBean

JavaBean是一种规范，也就是对类的要求。它要求Java类的成员变量提供getter/setter方法，这样的成员变量被称之为JavaBean属性。

JavaBean还要求类必须提供仅有的无参构造器，例如：public User() {…}

User.java

|  |
| --- |
| **package** cn.itcast.domain;  **public** **class** User {  **private** String username;  **private** String password;  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  } |

#### 5.1.2　JavaBean属性

JavaBean属性是具有getter/setter方法的成员变量。

* 也可以只提供getter方法，这样的属性叫只读属性；
* 也可以只提供setter方法，这样的属性叫只写属性；
* 如果属性类型为boolean类型，那么读方法的格式可以是get或is。例如名为abc的boolean类型的属性，它的读方法可以是getAbc()，也可以是isAbc()；

JavaBean属性名要求：前两个字母要么都大写，要么都小写：

|  |
| --- |
| **public** **class** User {  错误的JavaBean属性  **private** String iD;  正确  **private** String ID;  错误的JavaBean属性  **private** String qQ;  正确  **private** String QQ;  …  } |

JavaBean可能存在属性，但不存在这个成员变量，例如：

|  |
| --- |
| **public** **class** User {  **public** String getUsername() {  **return** "zhangSan";  }  } |

上例中User类有一个名为username的只读属性！但User类并没有username这个成员变量！

还可以并变态一点：

|  |
| --- |
| **public** **class** User {  **private** String hello;    **public** String getUsername() {  **return** hello;  }    **public** **void** setUsername(String username) {  **this**.hello = username;  }  } |

上例中User类中有一个名为username的属性，它是可读可写的属性！而Use类的成员变量名为hello！也就是说**JavaBean的属性名取决与方法名称，而不是成员变量的名称。但通常没有人做这么变态的事情。**

#### 5.1.3 JavaBean的作用

可以通过JavaBean和数据库中的表形成映射,便于开发;

### 5.2　内省（了解）

**内省的目标是得到JavaBean属性的读、写方法的反射对象，通过反射对JavaBean属性进行操作的一组API。例如User类有名为username的JavaBean属性，通过两个Method对象（一个是getUsenrmae()，一个是setUsername()）来操作User对象。**

如果你还不能理解内省是什么，那么我们通过一个问题来了解内省的作用。现在我们有一个Map，内容如下：

|  |
| --- |
| Map<String,String> map = **new** HashMap<String,String>();  map.put("username", "admin");  map.put("password", "admin123"); |
| **public** **class** User {  **private** String username;  **private** String password;  **public** User(String username, String password) {  **this**.username = username;  **this**.password = password;  }  **public** User() {  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** String toString() {  **return** "User [username=" + username + ", password=" + password + "]";  }  } |

现在需要把map的数据封装到一个User对象中！User类有两个JavaBean属性，一个叫username，另一个叫password。

你可能想到的是反射，通过map的key来查找User类的Field！这么做是没有问题的，但我们要知道类的成员变量是私有的，虽然也可以通过反射去访问类的私有的成员变量，但我们也要清楚反射访问私有的东西是有“危险”的，所以还是建议通过getUsername和setUsername来访问JavaBean属性。

#### 5.2.1　内省之获取BeanInfo

我们这里不想去对JavaBean规范做过多的介绍，所以也就不在多介绍BeanInfo的“出身”了。你只需要知道如何得到它，以及BeanInfo有什么。

通过java.beans.Introspector的getBeanInfo()方法来获取java.beans.BeanInfo实例。

|  |
| --- |
| BeanInfo beanInfo = Introspector.*getBeanInfo*(User.**class**); |

#### 5.2.2　得到所有属性描述符（PropertyDescriptor）

通过BeanInfo可以得到这个类的所有JavaBean属性的PropertyDescriptor对象。然后就可以通过PropertyDescriptor对象得到这个属性的getter/setter方法的Method对象了。

|  |
| --- |
| PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors(); |

每个PropertyDescriptor对象对应一个JavaBean属性：

* String getName()：获取JavaBean属性名称；
* Method getReadMethod：获取属性的读方法；
* Method getWriteMethod：获取属性的写方法。

#### 5.2.3　完成Map数据封装到User对象中

|  |
| --- |
| **public** **void** fun1() **throws** Exception {  Map<String,String> map = **new** HashMap<String,String>();  map.put("username", "admin");  map.put("password", "admin123");  //获取User类型的BeanInfo实例  BeanInfo beanInfo = Introspector.*getBeanInfo*(User.**class**);  //获取User类型的所有JavaBean属性的属性描述符对象  PropertyDescriptor[] pds = beanInfo.getPropertyDescriptors();  //创建User实例  User user = **new** User();  //循环遍历每个属性描述符对象  **for**(PropertyDescriptor pd : pds) {  //获取JavaBean属性名  String name = pd.getName();  //获取Map中对应属性的值  String value = map.get(name);  //如果在Map中存在这个属性的值  **if**(value != **null**) {  //获取该属性的写方法  Method writeMethod = pd.getWriteMethod();  //把Map中的值写入到user对象的对应属性中  writeMethod.invoke(user, value);  }  }  System.*out*.println(user);  } |

### 5.3　commons-beanutils

提到内省，不能不提commons-beanutils这个工具。它底层使用了内省，对内省进行了大量的简化！

使用beanutils需要的jar包：

* commons-beanutils.jar；
* commons-logging.jar；

#### 5.3.1　设置JavaBean属性

|  |
| --- |
| User user = **new** User();  //设置user对象的username属性为admin  BeanUtils.*setProperty*(user, "username", "admin");  //设置user对象的password属性为admin123  BeanUtils.*setProperty*(user, "password", "admin123");  System.*out*.println(user); |

#### 5.3.2　获取JavaBean属性

|  |
| --- |
| User user = **new** User("admin", "admin123");  //获取user对象的username属性值  String username = BeanUtils.*getProperty*(user, "username");  //获取user对象的password属性值  String password = BeanUtils.*getProperty*(user, "password");  System.*out*.println("username=" + username + ", password=" + password); |

#### 5.3.3　封装Map数据到JavaBean对象中

|  |
| --- |
| Map<String,String> map = **new** HashMap<String,String>();  map.put("username", "admin");  map.put("password", "admin123");  User user = **new** User();  //把map中的数据封装到user对象中，要求user的属性名要与map的key一致。  BeanUtils.*populate*(user, map);  System.*out*.println(user); |

|  |
| --- |
| **package** practice.javaBean;  **import** java.beans.BeanInfo;  **import** java.beans.IntrospectionException;  **import** java.beans.Introspector;  **import** java.beans.PropertyDescriptor;  **import** java.lang.reflect.InvocationTargetException;  **import** java.lang.reflect.Method;  **import** java.util.HashMap;  **import** java.util.Map;  **import** org.apache.commons.beanutils.BeanUtils;  **import** org.junit.Test;  **public** **class** Demo1 {  @Test  **public** **void** mapTest(){  Map<String, String> map = **new** HashMap<String,String>();  map.put("username", "27");  map.put("sex", "man");  //获取Person实例  Person person = **new** Person();  //获取Person类型的BeanInfo实例  **try** {  BeanInfo beanInfo = Introspector.*getBeanInfo*(Person.**class**);  //获取Person类型的所有JavaBean属性的属性描述符对象  PropertyDescriptor[] propertyDescriptors = beanInfo.getPropertyDescriptors();  //遍历JavaBean的属性描述符对象  **for** (PropertyDescriptor propertyDescriptor : propertyDescriptors) {  //获取JavaBean的属性名  String name = propertyDescriptor.getName();  //获取map的值  String value = map.get(name);  //判断value不为null,  **if**(value != **null**){  //获取写方法  Method writeMethod = propertyDescriptor.getWriteMethod();  //调用方法,写入参数  writeMethod.invoke(person, value);  }  }  } **catch** (IntrospectionException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (IllegalAccessException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (IllegalArgumentException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  } **catch** (InvocationTargetException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.*out*.println(person);  }    @Test  **public** **void** test(){  Person person = **new** Person();  //设置JavaBean属性  **try** {  BeanUtils.*setProperty*(person, "username", "27");  BeanUtils.*setProperty*(person, "sex", "man");  } **catch** (IllegalAccessException | InvocationTargetException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.*out*.println(Person.**class**);  System.*out*.println(Person.**class**.equals(**new** Person()));  System.*out*.println(person);  }  @Test  **public** **void** test1(){  Map<String, String> map = **new** HashMap<String,String>();  map.put("username", "27");  map.put("sex", "man");  //把map属性,放入到JavaBean中  Person person = **new** Person();  **try** {  BeanUtils.*populate*(person, map);  } **catch** (IllegalAccessException | InvocationTargetException e) {  // **TODO** Auto-generated catch block  e.printStackTrace();  }  System.*out*.println(person);  }  } |







### 5.4　JSP与JavaBean相关的动作标签

在JSP中与JavaBean相关的标签有：

* <jsp:useBean>：创建JavaBean对象；
* <jsp:setProperty>：设置JavaBean属性；
* <jsp:getProperty>：获取JavaBean属性；

我们需要先创建一个JavaBean类：

User.java

|  |
| --- |
| **package** cn.itcast.domain;  **public** **class** User {  **private** String username;  **private** String password;  **public** User(String username, String password) {  **this**.username = username;  **this**.password = password;  }  **public** User() {  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getPassword() {  **return** password;  }  **public** **void** setPassword(String password) {  **this**.password = password;  }  **public** String toString() {  **return** "User [username=" + username + ", password=" + password + "]";  }  } |

#### 5.4.1　<jsp:useBean>

<jsp:useBean>标签的作用是创建JavaBean对象：

* 在当前JSP页面创建JavaBean对象；
* 把创建的JavaBean对象保存到域对象中；

|  |
| --- |
| <jsp:useBean id=*"user1"* class=*"cn.itcast.domain.User"* /> |

上面代码表示在当前JSP页面中创建User类型的对象，并且把它保存到page域中了。下面我们把<jsp:useBean>标签翻译成Java代码：

|  |
| --- |
| <%  cn.itcast.domain.User user1 = **new** cn.itcast.domain.User();  pageContext.setAttribute("user1", user1);  %> |

这说明我们可以在JSP页面中完成下面的操作：

|  |
| --- |
| <jsp:useBean id=*"user1"* class=*"cn.itcast.domain.User"* />  <%=user1 %>  <%  out.println(pageContext.getAttribute("user1"));  %> |

<jsp:useBean>标签默认是把JavaBean对象保存到page域，还可以通过scope标签属性来指定保存的范围：

|  |
| --- |
| <jsp:useBean id=*"user1"* class=*"cn.itcast.domain.User"* scope=*"page"*/>  <jsp:useBean id=*"user2"* class=*"cn.itcast.domain.User"* scope=*"request"*/>  <jsp:useBean id=*"user3"* class=*"cn.itcast.domain.User"* scope=*"session"*/>  <jsp:useBean id=*"user4"* class=*"cn.itcast.domain.User"* scope=*"applicatioin"*/> |

<jsp:useBean>标签其实不一定会创建对象！！！其实它会先在指定范围中查找这个对象，如果对象不存在才会创建，我们需要重新对它进行翻译：

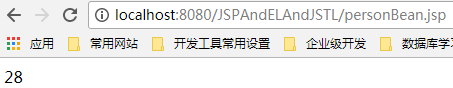
|  |
| --- |
| <jsp:useBean id=*"user4"* class=*"cn.itcast.domain.User"* scope=*"applicatioin"*/> |
| <%  cn.itcast.domain.User user4 = (cn.itcast.domain.User)application.getAttribute("user4");  **if**(user4 == **null**) {  user4 = **new** cn.itcast.domain.User();  application.setAttribute("user4", user4);  }  %> |

#### 5.4.2　<jsp:setProperty>和<jsp:getProperty>

<jsp:setProperty>标签的作用是给JavaBean设置属性值，而<jsp:getProperty>是用来获取属性值。在使用它们之前需要先创建JavaBean：

|  |
| --- |
| <jsp:useBean id=*"user1"* class=*"cn.itcast.domain.User"* />  <jsp:setProperty property=*"username"* name=*"user1"* value=*"admin"*/>  <jsp:setProperty property=*"password"* name=*"user1"* value=*"admin123"*/>  用户名：<jsp:getProperty property=*"username"* name=*"user1"*/><br/>  密　码：<jsp:getProperty property=*"password"* name=*"user1"*/><br/> |

|  |
| --- |
| <%@page import=*"practice.javaBean.Person"*%>  <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>Insert title here</title>  </head>  <body>  <!-- 演示usebean标签 -->  <jsp:useBean id=*"person1"* class=*"practice.javaBean.Person"*></jsp:useBean>  <jsp:setProperty property=*"username"* name=*"person1"* value=*"28"*/>  <jsp:useBean id=*"person2"* class=*"practice.javaBean.Person"* scope=*"application"*></jsp:useBean>  <jsp:useBean id=*"person3"* class=*"practice.javaBean.Person"* scope=*"page"*></jsp:useBean>  <% practice.javaBean.Person person = (practice.javaBean.Person)application.getAttribute("person2");  **if**(person ==**null**){  person = **new** practice.javaBean.Person();  application.setAttribute("person", person);  }  %>  <jsp:getProperty property=*"username"* name=*"person1"*/>  </body>  </html> |



|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <head>  <title>My JSP 'regist.jsp' starting page</title>  <meta http-equiv=*"pragma"* content=*"no-cache"*>  <meta http-equiv=*"cache-control"* content=*"no-cache"*>  <meta http-equiv=*"expires"* content=*"0"*>  <meta http-equiv=*"keywords"* content=*"keyword1,keyword2,keyword3"*>  <meta http-equiv=*"description"* content=*"This is my page"*>  <!--  <link rel="stylesheet" type="text/css" href="styles.css">  -->  </head>  <body>  <h1>注册</h1>  <form action=*"/JSPAndELAndJSTL/RegistServlet"* method=*"post"*>  姓名：<input type=*"text"* name=*"username"*/><br/>  年龄：<input type=*"text"* name=*"age"*/><br/>  性别：<input type=*"text"* name=*"sex"*/><br/>  <input type=*"submit"* value=*"注册"*/>  </form>  </body>  </html> |
| package practice.javaBean;  import java.io.IOException;  import java.lang.reflect.InvocationTargetException;  import java.util.Map;  import javax.servlet.ServletException;  import javax.servlet.annotation.WebServlet;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  import org.apache.commons.beanutils.BeanUtils;  import cn.itcast.commons.CommonUtils;  /\*\*获取表单参数,封装到JavaBean中  \* @author Yorick  \*/  @WebServlet("/RegistServlet")  public class RegistServlet extends HttpServlet {  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  request.setCharacterEncoding("utf-8");  response.setContentType("text/html;charset=utf-8");  // //获取所有参数封装到map中  // Map<String, String[]> map = request.getParameterMap();  // Person person = new Person();  // //把map传给bean  // try {  // BeanUtils.populate(person, map);  // } catch (IllegalAccessException e) {  // // TODO Auto-generated catch block  // e.printStackTrace();  // } catch (InvocationTargetException e) {  // // TODO Auto-generated catch block  // e.printStackTrace();  // }  // System.out.println(person);  //简单操作  Person person = CommonUtils.toBean(request.getParameterMap(), Person.class);  System.out.println(person);  }  } |



# EL

## 1. EL（表达式语言）

### 1.1　EL概述

#### 1.1.1　EL的作用

JSP2.0要把html和css分离、要把html和javascript分离、要把Java脚本替换成标签。标签的好处是非Java人员都可以使用。

JSP2.0 – 纯标签页面，即：不包含<% … %>、<%! … %>，以及<%= … %>

EL（Expression Language）是一门表达式语言，它对应<%=…%>。我们知道在JSP中，表达式会被输出，所以EL表达式也会被输出。

#### 1.1.2　EL的格式

格式：${…}

例如：${1 + 2}

#### 1.1.3　关闭EL

如果希望整个JSP忽略EL表达式，需要在page指令中指定isELIgnored=”true”。

如果希望忽略某个EL表达式，可以在EL表达式之前添加“\”，例如：\${1 + 2}。

#### 1.1.4　EL运算符

|  |  |  |  |
| --- | --- | --- | --- |
| 运算符 | 说明 | 范例 | 结果 |
| + | 加 | ${17+5} | 22 |
| - | 减 | ${17-5} | 12 |
| \* | 乘 | ${17\*5} | 85 |
| /或div | 除 | ${17/5}或${17 div 5} | 3 |
| %或mod | 取余 | ${17%5}或${17 mod 5} | 2 |
| ==或eq | 等于 | ${5==5}或${5 eq 5} | true |
| !=或ne | 不等于 | ${5!=5}或${5 ne 5} | false |
| <或lt | 小于 | ${3<5}或${3 lt 5} | true |
| >或gt | 大于 | ${3>5}或${3 gt 5} | false |
| <=或le | 小于等于 | ${3<=5}或${3 le 5} | true |
| >=或ge | 大于等于 | ${3>=5}或${3 ge 5} | false |
| &&或and | 并且 | ${true&&false}或${true and false} | false |
| !或not | 非 | ${!true}或${not true} | false |
| ||或or | 或者 | ${true||false}或${true or false} | true |
| empty | 是否为空 | ${empty “”}，可以判断字符串、数据、集合的长度是否为0，为0返回true。empty还可以与not或!一起使用。${not empty “”} | true |

#### 1.1.5　EL不显示null

　　当EL表达式的值为null时，会在页面上显示空白，即什么都不显示。

### 1.2　EL表达式格式

先来了解一下EL表达式的格式！现在还不能演示它，因为需要学习了EL11个内置对象后才方便显示它。

* 操作List和数组：${list[0]}、${arr[0]}；
* 操作bean的属性：${person.name}、${person[‘name’]}，对应person.getName()方法；
* 操作Map的值：${map.key}、${map[‘key’]}，对应map.get(key)。

### 1.3　EL内置对象

EL一共11个内置对象，无需创建即可以使用。这11个内置对象中有**10个是Map类型**的，最后一个是pageContext对象。

* pageScope
* requestScope
* sessionScope
* applicationScope

以上四个内置对象，对应四大域！它们都是Map<String,Object>类型！

使用EL操作Map：${map.key}以及${map[‘key’]}

* param：Map<String,String>，它对应参数！！！；
* paramValues：Map<String,String[]>；
* header：Map<String,String>，对应请求头；
* headerValues：Map<String,String[]>；
* initParam：Map<String,Object>，对应的是<context-param>中的初始化参数！；
* cookie：Map<String,Cookie>，如果要获取一个cookie值：${cookie.xxx.value}；
* pageContext：PageContext类型的！；

#### 1.3.1　域相关内置对象（重点）

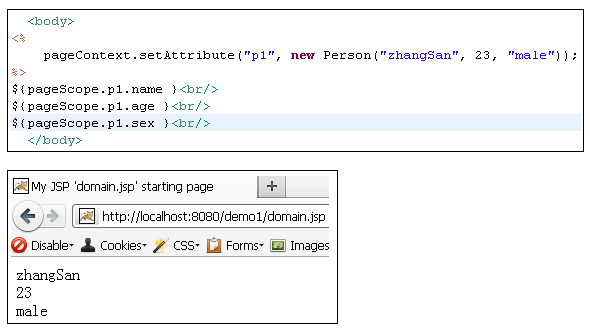
域内置对象一共有四个：

* pageScope：${pageScope.name}等同与pageContext.getAttribute(“name”)；
* requestScope：${requestScope.name}等同与request.getAttribute(“name”)；
* sessionScoep： ${sessionScope.name}等同与session.getAttribute(“name”)；
* applicationScope：${applicationScope.name}等同与application.getAttribute(“name”)；

如果在域中保存的是JavaBean对象，那么可以使用EL来访问JavaBean属性。因为**EL只做读取操作，所以JavaBean一定要提供get方法，而set方法没有要求。**

Person.java

|  |
| --- |
| **package** practice.javaBean;  **public** **class** Person {  //成员变量  **private** String username;  **private** String sex;  **private** **int** age;    **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getSex() {  **return** sex;  }  //JavaBean属性  **public** **void** setSex(String sex) {  **this**.sex = sex;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** "Person [username=" + username + ", age=" + age + ", sex=" + sex + "]";  }  } |

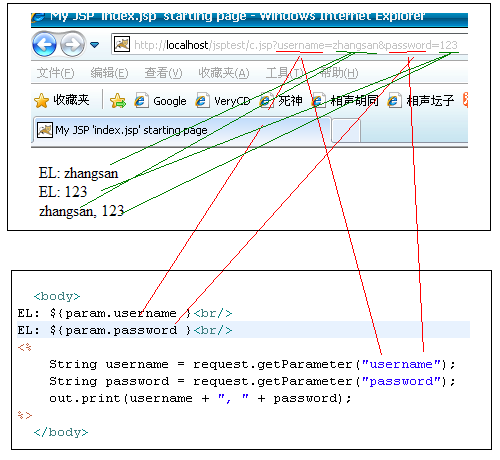


　　全域查找：${person}表示依次在pageScope、requesScopet、sessionScope、appliationScope四个域中查找名字为person的属性。

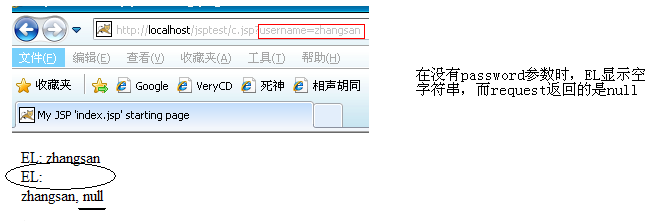
#### 1.3.2　请求参数相关内置对象

　　param和paramValues这两个内置对象是用来获取请求参数的。

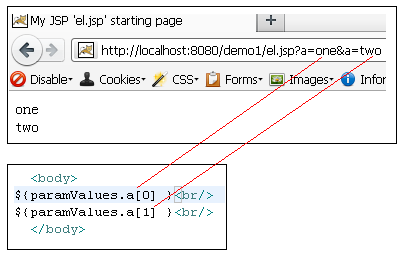
* param：Map<String,String>类型，param对象可以用来获取参数，与request.getParameter()方法相同。



注意，在使用EL获取参数时，如果参数不存在，返回的是空字符串，而不是null。这一点与使用request.getParameter()方法是不同的。



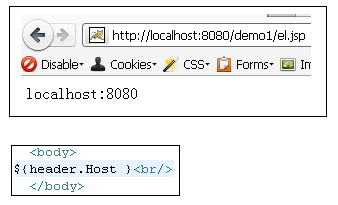
* paramValues：paramValues是Map<String, String[]>类型，当一个参数名，对应多个参数值时可以使用它。



#### 1.3.3　请求头相关内置对象

header和headerValues是与请求头相关的内置对象：

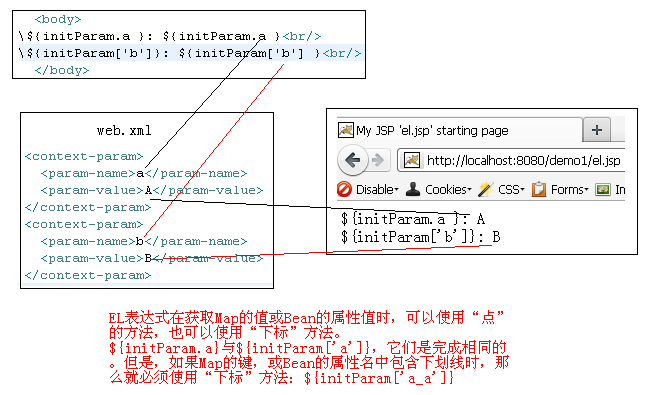
* header： Map<String,String>类型，用来获取请求头。



* headerValues：headerValues是Map<String,String[]>类型。当一个请求头名称，对应多个值时，使用该对象，这里就不在赘述。

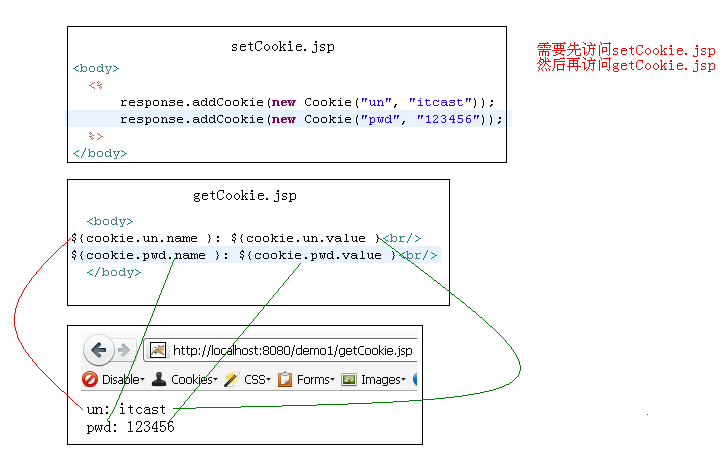
#### 1.3.4　应用初始化参数相关内置对象

* initParam：initParam是Map<String,String>类型。它对应web.xml文件中的<context-param>参数。



#### 1.3.5　Cookie相关内置对象

* cookie：cookie是Map<String,Cookie>类型，其中key是Cookie的名字，而值是Cookie对象本身。



#### 1.3.6　pageContext对象

pageContext：pageContext是PageContext类型！可以使用pageContext对象调用getXXX()方法，例如pageContext.getRequest()，可以${pageContext.request}。也就是读取JavaBean属性！！！

|  |  |
| --- | --- |
| **EL表达式** | **说明** |
| ${pageContext.request.queryString} | pageContext.getRequest().getQueryString(); |
| ${pageContext.request.requestURL} | pageContext.getRequest().getRequestURL(); |
| ${pageContext.request.contextPath} | pageContext.getRequest().getContextPath(); |
| ${pageContext.request.method} | pageContext.getRequest().getMethod(); |
| ${pageContext.request.protocol} | pageContext.getRequest().getProtocol(); |
| ${pageContext.request.remoteUser} | pageContext.getRequest().getRemoteUser(); |
| ${pageContext.request.remoteAddr} | pageContext.getRequest().getRemoteAddr(); |
| ${pageContext.session.new} | pageContext.getSession().isNew(); |
| ${pageContext.session.id} | pageContext.getSession().getId(); |
| ${pageContext.servletContext.serverInfo} | pageContext.getServletContext().getServerInfo(); |

|  |
| --- |
| <%@page import=*"java.util.HashMap"*%>  <%@page import=*"java.util.Map"*%>  <%@page import=*"java.util.Arrays"*%>  <%@page import=*"java.util.ArrayList"*%>  <%@page import=*"java.util.List"*%>  <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>Insert title here</title>  </head>  <body>  El:${"EL的书写格式,用于替代JSP脚本<%= %>"}</br>  EL:${"EL表达式里边为null时,不显示null,显示一个空的字符串!" }El:${null }</br>  <% List<String> list = **new** ArrayList<String>();  list.add("list1");  list.add("list2");  String[] arr = {"arr1","arr2","arr3"};  Map<String,String> map = **new** HashMap<String,String>();  map.put("name", "肖涵");  map.put("sex", "男");  %>  <jsp:useBean id=*"person1"* class=*"practice.javaBean.Person"* scope=*"application"*></jsp:useBean>  <jsp:setProperty property=*"username"* name=*"person1"* value=*"刘磊"*/>  <jsp:setProperty property=*"sex"* name=*"person1"* value=*"男"*/>  <jsp:setProperty property=*"age"* name=*"person1"* value=*"36"*/>  EL:${"使用EL操作list集合:"}${list[0]}</br>  EL:${"使用EL操作arr数组:"}${arr[1] }</br>  El:${"使用EL操作Map集合:"}${map.name}${" "}${map["sex"]}</br>  EL:${"使用EL操作bean属性:"}${ person1.username}${" "}${person1["sex"]}</br>  EL:${"域相关的内置对象 :"}${applicationScope.person1.username }</br>  <jsp:useBean id=*"person2"* class=*"practice.javaBean.Person"* scope=*"session"*></jsp:useBean>  <jsp:setProperty property=*"username"* name=*"person2"* value=*"肖涵"*/>  <jsp:setProperty property=*"sex"* name=*"person2"* value=*"男"*/>  <jsp:setProperty property=*"age"* name=*"person2"* value=*"35"*/>  EL:${"域相关的内置对象 :" }${sessionScope.person2.username }</br>  EL:${"域相关的内置对象 ,不存在返回空字符串,不会报错 :" }${sessionScope.person.username }</br>  EL:${"域相关的内置对象 ,不存在返回空字符串,不会报错 :" }${sessionScope.username }</br>  <h3>请求参数相关内置对象</h3>  <p>param和paramValues这两个内置对象是用来获取请求参数的。</p></br>  EL:${param.username }</br>  EL:${param["username"]}</br>  El:${paramValues.password }</br>  El:${paramValues["password"][0]} }</br>  <h3>header和headerValues</h3>  <p>header和headerValues是与请求头相关的内置对象：</p></br>  EL:${header['User-Agent'] }</br>  <h3>演示initParam</h3>  ${"initParam是Map<String,String>类型。它对应web.xml文件中的<context-param>参数。"}</br>  ${initParam.cxtParamName1 }<br/>  ${initParam.cxtParamName2 }<br/>  <h3>演示cookie</h3>  <p>cookie是Map String,Cookie 类型，其中key是Cookie的名字，而值是Cookie对象本身。</p></br>  ${cookie.JSESSIONID.value }</br>  <h3>演示pageContext</h3>  queryString:${pageContext.request.queryString }</br>  requestURL:${pageContext.request.requestURI }</br>  contextPath:${pageContext.request.contextPath }</br>  method:${pageContext.request.method }</br>  protocol:${pageContext.request.protocol }</br>  remoteUser:${pageContext.request.remoteUser }</br>  RemoteAddr():${pageContext.request.remoteAddr }</br>  getId():${pageContext.session.id }</br>  </body>  </html> |



El:EL的书写格式,用于替代JSP脚本<%= %>  
EL:EL表达式里边为null时,不显示null,显示一个空的字符串!El:  
EL:使用EL操作list集合:  
EL:使用EL操作arr数组:  
El:使用EL操作Map集合:   
EL:使用EL操作bean属性:刘磊 男  
EL:域相关的内置对象 :刘磊  
EL:域相关的内置对象 :肖涵  
EL:域相关的内置对象 ,不存在返回空字符串,不会报错 :  
EL:域相关的内置对象 ,不存在返回空字符串,不会报错 :

请求参数相关内置对象

param和paramValues这两个内置对象是用来获取请求参数的。

EL:username  
EL:username  
El:[Ljava.lang.String;@1717a05  
El:pass1 }

header和headerValues

header和headerValues是与请求头相关的内置对象：

EL:Mozilla/5.0 (Windows NT 6.1; WOW64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/61.0.3128.0 Safari/537.36

演示initParam

initParam是Map<string,string style="color: rgb(0, 0, 0); font-family: "Microsoft YaHei"; font-size: medium; font-style: normal; font-variant-ligatures: normal; font-variant-caps: normal; font-weight: normal; letter-spacing: normal; orphans: 2; text-align: start; text-indent: 0px; text-transform: none; white-space: normal; widows: 2; word-spacing: 0px; -webkit-text-stroke-width: 0px; text-decoration-style: initial; text-decoration-color: initial;">类型。它对应web.xml文件中的参数。  
cxtParamName1  
cxtParamName2

演示cookie

cookie是Map<string,cookie>类型，其中key是Cookie的名字，而值是Cookie对象本身。

2D11439499ADD4394E320C58322CA241

演示pageContext

queryString:username=username&password=pass1&password=pass2  
requestURL:/JSPAndELAndJSTL/ELPractice.jsp  
contextPath:/JSPAndELAndJSTL  
method:GET  
protocol:HTTP/1.1  
remoteUser:  
RemoteAddr():127.0.0.1  
getId():2D11439499ADD4394E320C58322CA241

## 2. EL函数库

### 2.1　什么EL函数库

　　EL函数库是由第三方对EL的扩展，我们现在学习的EL函数库是由JSTL添加的。JSTL后面再学！

**EL函数库就是定义一些有返回值的静态方法。然后通过EL语言来调用它们！当然，不只是JSTL可以定义EL函数库，我们也可以自定义EL函数库。**

　　EL函数库中包含了很多对字符串的操作方法，以及对集合对象的操作。例如：${fn:length(“abc”)}会输出3，即字符串的长度。

### 2.2　导入函数库

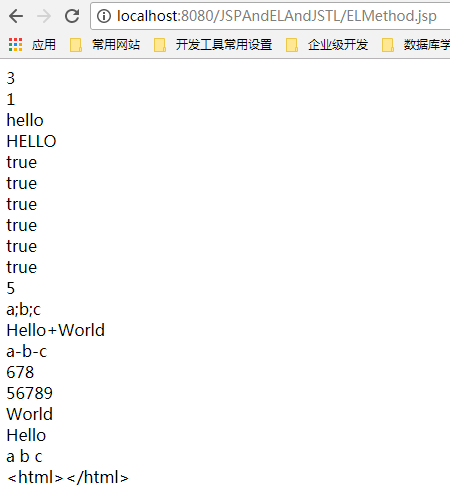
　　因为是第三方的东西，所以需要导入。导入需要使用taglib指令！

<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>

### 2.3　EL函数库介绍

* String toUpperCase(String input)：
* String toLowerCase(String input)：
* int indexOf(String input, String substring)：
* boolean contains(String input, String substring)：
* boolean containsIgnoreCase(String input, String substring)：
* boolean startsWith(String input, String substring)：
* boolean endsWith(String input, String substring)：
* String substring(String input, int beginIndex, int endIndex)：
* String substringAfter(String input, String substring)：hello-world, “-“
* String substringBefore(String input, String substring)：hello-world, “-“
* String escapeXml(String input)：把字符串的“>”、“<”。。。转义了！
* String trim(String input)：
* String replace(String input, String substringBefore, String substringAfter)：
* String[] split(String input, String delimiters)：
* int length(Object obj)：可以获取字符串、数组、各种集合的长度！
* String join(String array[], String separator)：

|  |
| --- |
| <%@taglib prefix=*"fn"* uri=*"http://java.sun.com/jsp/jstl/functions"* %>  …  String[] strs = {"a", "b","c"};  List list = **new** ArrayList();  list.add("a");  pageContext.setAttribute("arr", strs);  pageContext.setAttribute("list", list);  %>  ${fn:length(arr) }<br/><!--3-->  ${fn:length(list) }<br/><!--1-->  ${fn:toLowerCase("Hello") }<br/> <!-- hello -->  ${fn:toUpperCase("Hello") }<br/> <!-- HELLO -->  ${fn:contains("abc", "a")}<br/><!-- true -->  ${fn:containsIgnoreCase("abc", "Ab")}<br/><!-- true -->  ${fn:contains(arr, "a")}<br/><!-- true -->  ${fn:containsIgnoreCase(list, "A")}<br/><!-- true -->  ${fn:endsWith("Hello.java", ".java")}<br/><!-- true -->  ${fn:startsWith("Hello.java", "Hell")}<br/><!-- true -->  ${fn:indexOf("Hello-World", "-")}<br/><!-- 5 -->  ${fn:join(arr, ";")}<br/><!-- a;b;c -->  ${fn:replace("Hello-World", "-", "+")}<br/><!-- Hello+World -->  ${fn:join(fn:split("a;b;c;", ";"), "-")}<br/><!-- a-b-c -->  ${fn:substring("0123456789", 6, 9)}<br/><!-- 678 -->  ${fn:substring("0123456789", 5, -1)}<br/><!-- 56789 -->  ${fn:substringAfter("Hello-World", "-")}<br/><!-- World -->  ${fn:substringBefore("Hello-World", "-")}<br/><!-- Hello -->  ${fn:trim(" a b c ")}<br/><!-- a b c -->  ${fn:escapeXml("<html></html>")}<br/> <!-- <html></html> --> |



### 2.4　自定义EL函数库

* 写一个类，写一个有返回值的静态方法；
* 编写itcast.tld文件，可以参数fn.tld文件来写，把itcast.tld文件放到/WEB-INF目录下；
* 在页面中添加taglib指令，导入自定义标签库。

ItcastFuncations.java

|  |
| --- |
| **package** cn.itcast.el.funcations;  **public** **class** ItcastFuncations {  **public** **static** String test() {  **return** "传智播客自定义EL函数库测试";  }  } |

itcast.tld（放到classes下）

|  |
| --- |
| <?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 http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary\_2\_0.xsd"*  version=*"2.0"*>    <tlib-version>1.0</tlib-version>  <short-name>itcast</short-name>  <uri>http://www.itcast.cn/jsp/functions</uri>  <!-- 一个function元素对应一个函数的部署！ -->  <function>  <!-- 指定函数名称 -->  <name>test</name>  <!-- 指定函数类--><function-class>cn.itcast.el.funcations.ItcastFuncations</function-class>  <!-- 指定函数的签名 -->  <function-signature>String test()</function-signature>  </function>  </taglib> |

index.jsp

|  |
| --- |
| <%@ page language=*"java"* import=*"java.util.\*"* pageEncoding=*"UTF-8"*%>  <%@ taglib prefix=*"itcast"* uri=*"/WEB-INF/itcast.tld"* %>  <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">  <html>  <body>  <h1>${itcast:test() }</h1>  </body>  </html> |

# 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{其实EL原来是JSTL的一部分，现在是JSP的一部分了。}），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。

它们过时了，没人再用了(最后两个过时了,没人再用了!)

## 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"*/><!--在page域中创建名为a的变量-->  <c:if test="${**not empty** a }"><!--判断a变量不为null，没有指定域，表示全域  -->  <c:out value="${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="${param.score }"/>  <c:choose>  <!-if-->  <c:when test="${score > 100 || score < 0}">错误的分数：${score }</c:when>  <!--else if-->  <c:when test="${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>  <!--else-->  <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：

|  |
| --- |
| //创建page域属性sum为0  <c:set var=*"sum"* value=*"0"* />  //设置循环变量i，初始值为1，遍历到10。  <c:forEach var=*"i"* begin=*"1"* end=*"10"*>  //设置sum的值为sum + i  <c:set var=*"sum"* value="${sum + i}" />  </c:forEach>  <c:out value="*sum =* ${sum }"/> |
| <c:set var=*"sum"* value=*"0"* />  //步长为2，默认步长为1。相当于i+=2的意思！而默认为i++。  <c:forEach var=*"i"* begin=*"1"* end=*"10"* step=*"2"*>  <c:set var=*"sum"* value="${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);  %>  //把ns是中每一项赋给item。 ns是被遍历的数组。  <c:forEach var=*"item"* items="${ns }">  //打印每一项  <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);  %>  //与遍历数组没有区别！  for(String s : ns) {  …  }  <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);  %>  //因为遍历的是Map，所以每一项是Entry类型  <c:forEach var=*"item"* items="${stu }">  //获取Entry的key和value  <c:out value="${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。

|  |
| --- |
| //把循环状态赋给vs  <c:forEach var=*"item"* items="${ns }" varStatus=*"vs"*>  //如果是第一行，那么${vs.first}为true  <c:if test="${vs.first }">第一行：</c:if>  //如果是最后一行，那么${vs.last}为true  <c:if test="${vs.last }">最后一行：</c:if>  //返回行数，从1开始  <c:out value="*第*${vs.count }*行:* "/>  //返回行数，从0开始  <c:out value="*[*${vs.index }*]:* "/>  //当前项，与${item}相同  <c:out value="*name:* ${vs.current }"/><br/>  </c:forEach> |

|  |
| --- |
| **package** practice.javaBean;  **public** **class** Person {  //成员变量  **private** String username;  **private** String sex;  **private** **int** age;    **public** Person(String username, String sex, **int** age) {  **super**();  **this**.username = username;  **this**.sex = sex;  **this**.age = age;  }  **public** String getUsername() {  **return** username;  }  **public** **void** setUsername(String username) {  **this**.username = username;  }  **public** String getSex() {  **return** sex;  }  //JavaBean属性  **public** **void** setSex(String sex) {  **this**.sex = sex;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }    @Override  **public** String toString() {  // **TODO** Auto-generated method stub  **return** "Person [username=" + username + ", age=" + age + ", sex=" + sex + "]";  }  } |
| <%@page import=*"practice.javaBean.Person"*%>  <%@page import=*"java.util.LinkedHashMap"*%>  <%@page import=*"java.util.Map"*%>  <%@page import=*"java.util.List"*%>  <%@page import=*"java.util.ArrayList"*%>  <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>for each</title>  </head>  <body>  <h4>普通for循环</h4>  <c:set var=*"sum"* value=*"0"* scope=*"request"*></c:set>  <% **for**(**int** i = 1; i <= 10; i+=2) {  out.print(i);  } %>  <!-- step 代表步数,默认1 -->  <c:forEach var=*"i"* begin=*"1"* end=*"10"* step=*"2"*>  <c:set var=*"sum"* value=*"*${sum +i}*"*></c:set>  </c:forEach>  <c:out value=*"*${sum }*"*></c:out>  <h4>遍历集合和数组</h4>  <% String[] arr = **new** String[]{"张三","赵四","王二麻子"};  pageContext.setAttribute("arr", arr);  List<String> list = **new** ArrayList<String>();  list.add("张三");  list.add("赵四");  list.add("王二麻子");  application.setAttribute("list", list);  %>  <%  **for**(String username : arr){  out.write(username);  }  %>  遍历数组:<br/>  <c:forEach var=*"arr"* items=*"*${arr }*"*>  <c:out value=*"arr:*${arr }*"*></c:out>  </c:forEach>  遍历List集合:<br/>  <c:forEach var=*"list"* items=*"*${list }*"*>  <c:out value=*"list:*${list }*"*></c:out>  </c:forEach>  <%Map<String,String> map = **new** LinkedHashMap<String,String>();  map.put("str1", "str1");  map.put("str2", "str2");  map.put("str3", "str3");  request.setAttribute("map", map);  %>  </br>  遍历List集合:</br>  <% List<Person> personList = **new** ArrayList<Person>();  personList.add(**new** Person("张三","男",25));  personList.add(**new** Person("李四","女",88));  personList.add(**new** Person("王五","男",23));  application.setAttribute("list", personList);  %>  <table width=*"50%"* border=*"1"* align=*"center"*>  <tr>  <th>姓名</th>  <th>年龄</th>  <th>性别</th>  </tr>  <c:forEach var=*"person"* items=*"*${list }*"* varStatus=*"per"*>  <c:choose >  <c:when test=*"*${per.first }*"*>  <c:set var=*"color"* value=*"red"*></c:set>  </c:when>  <c:when test=*"*${per.last }*"*>  <c:set var=*"color"* value=*"blue"*></c:set>  </c:when>  <c:otherwise>  <c:set var=*"color"* value=*"write"*></c:set>  </c:otherwise>  </c:choose>  <tr>  <td>${person.username }</td>  <td>${person.sex}</td>  <td>${person.age }</td>  </tr>  <tr>  <td><c:out value=*"*${per.count }*"*></c:out></td>  <td><c:out value=*"*${per.index }*"*></c:out></td>  <td><c:out value=*"*${per.current }*"*></c:out></td>  </tr>  </c:forEach>  </table>  </br>  遍历map集合:</br>  <c:forEach var=*"map"* items=*"*${requestScope.map }*"*>  <c:out value=*"*${map.key }*:*${map.value }*"*></c:out>  </c:forEach>  </body>  </html> |
| <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@ taglib prefix=*"c"* uri=*"http://java.sun.com/jsp/jstl/core"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>jstl标签练习</title>  </head>  <body>  <pre>JSTL是apache对EL表达式的扩展（也就是说JSTL依赖EL{其实EL原来是JSTL的一部分，  现在是JSP的一部分了。}），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！</pre>  <h3>core标签库--核心标签库</h3>  <h4>out&&set</h4>  <% request.setAttribute("aaa", "域对象aaa");  pageContext.setAttribute("js", "<script>while(true) alert('你死定了！');</script>");  %>  输出字符串aaa:<c:out value=*"aaa"*></c:out></br>  输出域对象aaa,从page域开始查找,从小域到大域:<c:out value=*"*${aaa}*"* /></br>  输出测试,没有此域对象,也没有设置default:<c:out value=*"*${aaaa}*"*></c:out></br>  输出域对象aaa,从page域开始查找,从小域到大域,查找到输出,查找不到输出default:<c:out value=*"*${aaaa}*"* default=*"默认"*></c:out></br>  <c:out value=*"*${js }*"* escapeXml=*"true"*/></br>  <c:out value=*"*${js }*"* /></br>  <c:set var=*"a"* value=*"域对象a"* scope=*"session"*></c:set>  <c:set var=*"a"* value=*"域对象a,添加到pageContext域中;"*></c:set>  <c:set var=*"aaa"* value=*"域对象a,添加到pageContext域中;"*></c:set>  <c:out value=*"*${a }*"*></c:out>  <h4>remove</h4>  删除pageContext域中名称为a的对象:<c:remove var=*"a"* scope=*"page"*/></br>  <c:out value=*"*${a }*"*></c:out></br>  删除所有域中名称为aaa的对象:<c:remove var=*"aaa"*/></br>  <c:out value=*"*${aaa }*"* default=*"没有此对象"*></c:out></br>  <h4>url</h4>  <!-- 输出 -->  <c:url value=*"/"*></c:url>  <!-- 不输出 -->  <c:url value=*"/"* var=*"contextPath"* scope=*"request"*></c:url></br>  <c:url value=*"/RegistServlet"* var=*"path"* scope=*"session"*></c:url>  <c:url value=*"/RegistServlet"* var=*"path1"* scope=*"application"*>  <c:param name=*"username"* value=*"username"*></c:param>  <c:param name=*"password"* value=*"password"*></c:param>  </c:url>  <c:out value=*"*${contextPath }*"* default=*"不能正常显示"*></c:out></br>  <c:out value=*"*${path }*"* default=*"不能正常显示"*></c:out></br>  ${applicationScope.path1 }</br>  <!-- 通常和链接及表单放在一起使用 -->  <a href=*"*${path1 }*"*>点击链接RegistServlet</a>  <h4>if标签</h4>  <c:if test=*"*${5>4 }*"*>  <c:out value=*"对了!"*></c:out>  </c:if>  <h4>choose</h4>  <c:set var=*"scope"* value=*"*${param.scope }*"*></c:set>  <c:out value=*""*></c:out>  <c:choose>  <c:when test=*"*${scope > 100 || scope<0} *"*>成绩输入错误</c:when>  <c:when test=*"*${scope >=90 }*"*>A</c:when>  <c:when test=*"*${scope >=80 }*"*>B</c:when>  <c:when test=*"*${scope >=70 }*"*>C</c:when>  <c:when test=*"*${scope>=60 }*"*>D</c:when>  <c:otherwise>E</c:otherwise>  </c:choose>    </body>  </html> |



JSTL是apache对EL表达式的扩展（也就是说JSTL依赖EL{其实EL原来是JSTL的一部分，

现在是JSP的一部分了。}），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！

core标签库--核心标签库

out&&set

输出字符串aaa:aaa  
输出域对象aaa,从page域开始查找,从小域到大域:域对象aaa  
输出测试,没有此域对象,也没有设置default:  
输出域对象aaa,从page域开始查找,从小域到大域,查找到输出,查找不到输出default:默认  
<script>while(true) alert('你死定了！');</script>  
<script>while(true) alert('你死定了！');</script>  
域对象a,添加到pageContext域中;

remove

删除pageContext域中名称为a的对象:  
域对象a  
删除所有域中名称为aaa的对象:  
没有此对象

url

/JSPAndELAndJSTL/   
/JSPAndELAndJSTL/  
/JSPAndELAndJSTL/RegistServlet  
/JSPAndELAndJSTL/RegistServlet?username=username&password=password  
[点击链接RegistServlet](http://localhost:8080/JSPAndELAndJSTL/RegistServlet?username=username&password=password)

if标签

对了!

choose

E

## 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。

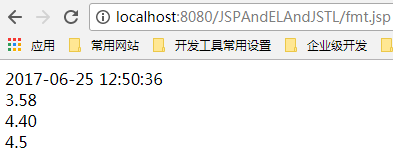
格式化时间：

|  |
| --- |
| <%@ taglib prefix=*"fmt"* uri=*"http://java.sun.com/jsp/jstl/fmt"* %>  ......  <%  Date date = **new** Date();  pageContext.setAttribute("d", date);  %>  //按指定格式输出日期和时间。  <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);  %>  //必须且仅能保留两位小数，如果大于两位，那么只保留两位，并四舍五入，如果小于两位，那么使用0补足两位  <fmt:formatNumber value="${d1 }" pattern=*"0.00"*/><br/>  //最多显示两位，如果小于两位，那么有几位保留几位，不会用0补足。大于两位，只保留两位，并四舍五入！  <fmt:formatNumber value="${d2 }" pattern=*"#.##"*/> |

|  |
| --- |
| <%@page import=*"java.util.Date"*%>  <%@ page language=*"java"* contentType=*"text/html; charset=utf-8"*  pageEncoding=*"utf-8"*%>  <%@ taglib prefix=*"fmt"* uri=*"http://java.sun.com/jsp/jstl/fmt"* %>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <meta http-equiv=*"Content-Type"* content=*"text/html; charset=ISO-8859-1"*>  <title>Insert title here</title>  </head>  <body>  <%--  其中pattern="#.##"：  > 表示保留两位小数，这会自动四舍五入  > 如果要格式化的小数不足两位，那么正常显示，不会补位！  其中pattern="0.00"：  > 表示保留两位小数，这会自动四舍五入  > 如果要格式化的小数不足两位，那么补足到两位  --%>  <% Date date = **new** Date();  session.setAttribute("date", date);  **double** d1 = 3.576;  **double** d2 = 4.4;  **double** d3 = 4.5;  pageContext.setAttribute("d1", d1);  pageContext.setAttribute("d2", d2);  pageContext.setAttribute("d3", d3);  %>  <fmt:formatDate value=*"*${sessionScope.date }*"* pattern=*"yyyy-MM-dd HH:mm:ss"*/></br>  <fmt:formatNumber value=*"*${d1 }*"* pattern=*"0.00"*></fmt:formatNumber></br>  <fmt:formatNumber value=*"*${d2 }*"* pattern=*"0.00"*></fmt:formatNumber></br>  <fmt:formatNumber value=*"*${d3 }*"* pattern=*"#.##"*/>    </body>  </html> |



# 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 {  //获取out对象，输出Hello Tag!!!  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 "*>  //指定当前标签库的版本  <tlib-version>1.0</tlib-version>  //定当前标签库的简称，这个名称无所谓了，随便起。  <short-name>itcast</short-name>  //指定标签库的uri  <uri>http://www.itcast.cn/tags</uri>  //部署一个标签！一个<tag>表示一个标签。  <tag>  //指定标签的名称  <name>hello</name>  //指定标签处理类  <tag-class>cn.itcast.tag.HelloTag</tag-class>  //指定标签体内容类型为空类型，即没有标签体。  <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；

|  |
| --- |
| //指定标签库前缀 指定标签库的tld文件位置  <%@ 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 {  //向页面输出！注意，不能向页面输出<%%>东西！  **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")) {  //获取当前标签的标签体对象  JspFragment body = **this**.getJspBody();  //向页面输出标签体内容。  body.invoke(null);  }  }  } |
| <tag>  <name>hello</name>  <tag-class>cn.itcast.tags.HelloTag</tag-class>  //指定标签体内容为scriptless，即标签体内容可以是正常的html，也可以是EL或JSTL  <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;  }  //该方法会在doTag()之前被调用，传入属性值。  **public** **void** setTest(**boolean** test) {  **this**.test = test;  }  @Override  **public** **void** doTag() **throws** JspException, IOException {  //如果test为true，执行标签体内容，否则什么都不做。  **if**(test) {  **this**.getJspBody().invoke(**null**);  }  }  } |
| <tag>  <name>if</name>  <tag-class>cn.itcast.tag.IfTag</tag-class>  <body-content>scriptless</body-content>  //声明属性  <attribute>  //属性名称为test  <name>test</name>  //为true表示属性是必须的，为false表示属性为可选的。  <required>true</required>  //为true表示属性值可以为EL或JSTL，否则表示只能是常量。  <rtexprvalue>true</rtexprvalue>  </attribute>  </tag> |
| <%  pageContext.setAttribute("one", **true**);  pageContext.setAttribute("two", **false**);  %>  //可以看到xixi  <it:if test="${one }">xixi</it:if>  //不会执行haha  <it:if test="${two }">haha</it:if>  //可以看到hehe  <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" \o "软件工程)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" \o "软件架构)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" \o "Trygve Reenskaug（页面不存在）)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" \o "帕罗奥多研究中心)（Xerox PARC）的[Smalltalk](http://zh.wikipedia.org/wiki/Smalltalk" \o "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；

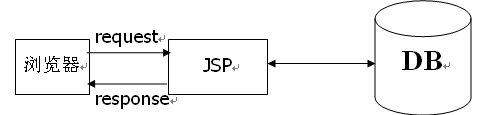
注意，MVC不是Java的东西，几乎现在所有B/S结构的软件都采用了MVC设计模式。但是要注意，MVC在B/S结构软件并没有完全实现，例如在我们今后的B/S软件中并不会有事件驱动！

## 2　JavaWeb与MVC

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

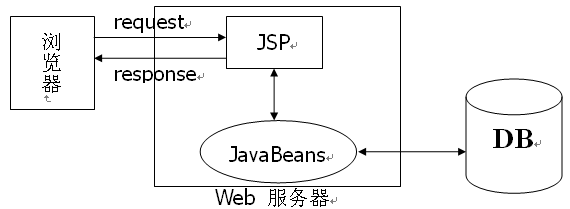
### 2.1　JSP Model1第一代

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

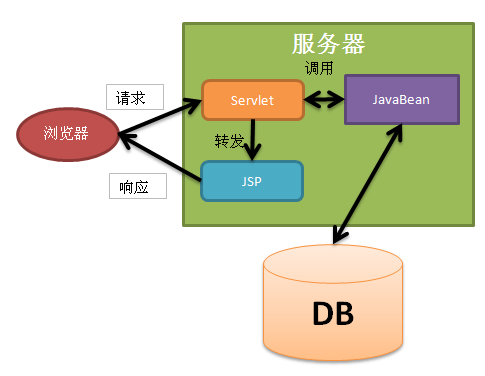
　　JSP Model1第二代有所改进，把业务逻辑的内容放到了JavaBean中，而JSP页面负责显示以及请求调度的工作。虽然第二代比第一代好了些，但还让JSP做了过多的工作，JSP中把视图工作和请求调度（控制器）的工作耦合在一起了。



### 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；

//不要在业务层看到Web相关类！！！

* 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！

