# 自定义标签之传统标签

## 标签API简单介绍

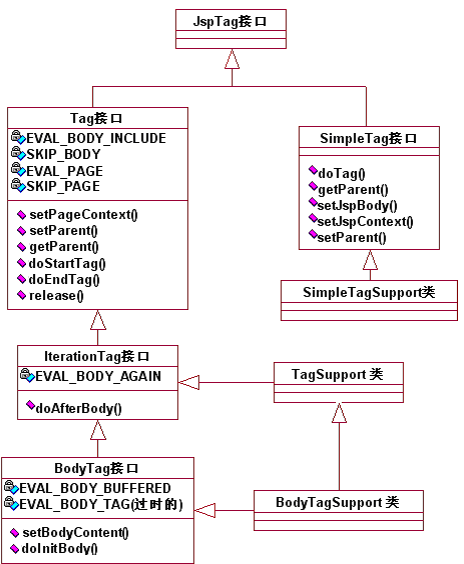
**JSTL标签库**只提供了简单的输出等功能，没有实现任何的HTML代码封装，并且某些复杂类型转换，或者逻辑处理的时候，JSTL标签库完成不了，需要自定义标签!

**编写自定义标签的步骤：**

编写一个**实现Tag接口**的Java类【标签处理器类】

在WEB-INF目录下创建**tld**(Tag Library Descriptor)文件，在tld文件中对标签处理类（实现Tag接口的Java类）进行描述。

**标签技术的API类继承关系**



把**实现Tag接口的自定义标签也叫做传统标签**，把**实现SimpleTag接口的自定义标签叫做简单标签**。

## Tag接口

**public** **interface** **Tag** **extends** JspTag {  
 **int** SKIP\_BODY = 0;  
 **int** EVAL\_BODY\_INCLUDE = 1;  
 **int** SKIP\_PAGE = 5;  
 **int** EVAL\_PAGE = 6;  
 **void** setPageContext(PageContext var1);  
 **void** setParent(Tag var1);  
 Tag getParent();  
 **int** doStartTag() **throws** JspException;  
 **int** doEndTag() **throws** JspException;  
 **void** release(); }

**程序的执行流程：**

* + JSP引擎遇到自定义标签，首先创建标签处理器类的**实例对象**。
  + JSP引擎实例化完标签处理器类后，调用setPageContext()方法，将pageContext对象传递给标签处理器类，使得标签处理器类可**以通过pageContext对象与JSP页面进行通信！**
  + setPageContext()方法执行完后，调用setParent()方法，将当前标签的父标签传递给当前处理器类，如果当前标签没有父标签，则传入null
  + 当WEB容器执行到自定义标签的开始标记时，调用doStartTag()方法。
  + 当WEB容器执行到自定义标签的结束标记时，调用doEndTag()方法。
  + 一般来说，当WEB容器执行完自定义标签后，标签处理器类会驻留在内存中，直至停止WEB应用时，WEB容器才会调用release()方法。

**Tag接口的源码还有4个变量：**

控制JSP页面某一部分（标签体）是否执行

控制整个JSP页面是否执行

控制JSP页面内容重复执行

修改JSP页面内容输出

再看回4个变量的名字，我们可以发现，这4个变量就是用来做逻辑判断的！

* 如果doStartTag方法返回EVAL\_BODY\_INCLUDE，WEB容器就会接着执行自定义标签的标签体；如果doStartTag方法返回SKIP\_BODY，WEB容器就会忽略自定义标签的标签体，直接解释执行自定义标签的结束标记。
* 如果doEndTag方法返回常量EVAL\_PAGE，WEB容器就会接着执行JSP页面中位于结束标记后面的JSP代码；如果doEndTag方法返回SKIP\_PAGE，WEB容器就会忽略JSP页面中位于结束标记后面的所有内容。

### IterationTag接口

在Tag接口的基础上增加了一个**doAfterBody方法和一个EVAL\_BODY\_AGAIN常量**。实现IterationTag接口的标签除了可以完成Tag接口所能完成的功能外，还能够通知WEB容器**是否重复执行标签体内容。**

doAfterBody方法可以向WEB容器**返回常量EVAL\_BODY\_AGAIN或SKIP\_BODY。**如果doAfterBody方法返回EVAL\_BODY\_AGAIN，WEB容器就会把标签体内容再重复执行一次，执行完后再调用doAfterBody方法，如此往复，直到doAfterBody方法返回常量SKIP\_BODY，WEB容器才会开始处理标签的结束标记和调用doEndTag方法。

//定义一个变量，规定标签体循环的次数

int x = 0;

@Override

public int doStartTag() throws JspException {

try {

pageContext.getOut().write("hello");

} catch (IOException e) {

e.printStackTrace();

}

//执行标签体

return EVAL\_BODY\_INCLUDE;

}

@Override

public int doAfterBody() throws JspException {

x++;

if (x >= 10) {

return SKIP\_BODY;

}

//标签体不断循环，直到doAfterBody()返回的是SKIP\_BODY

return EVAL\_BODY\_AGAIN;

}

我们在编写自定义标签的标签处理器类时，可以**继承和扩展TagSupport类**，这相比实现IterationTag接口将简化开发工作。

### TagSupport实现类

TagSupport是Tag的一个模板类，**实现了pageContext，parent的getter、setter方法以及一些其他的功能。**我们要做的**就是**重写doStartTag()和doEndTag()方法。

想要自定义标签带有属性也非常简单，只要在标签处理器类上加一个成员变量和setter、getter()，再在**tld文件中描述**下该属性即可！它的原理是这样的：当标签使用到属性的时候，引擎就会调用它的setter()方法

### BodyTag接口

BodyTag接口继承IterationTag接口，在IterationTag接口的基础上增加了两个方法（setBodyContent、doInitBody）和一个EVAL\_BODY\_BUFFERED常量。实现BodyTag接口的标签除了可以完成IterationTag接口所能完成的功能，还可以**对标签体内容进行修改。**

**编写一个类实现BodyTag接口，控制doStartTag()方法返回EVAL\_BODY\_BUFFERED，则web服务器会创建BodyContent对象捕获标签体，然后在doEndTag()方法体内，得到代表标签体的bodyContent对象，从而就可以对标签体进行修改操作。**

**BodyTag的源码**！

**public** **interface** **BodyTag** **extends** IterationTag {  
 /\*\* @deprecated \*/  
 **int** EVAL\_BODY\_TAG = 2;  
 **int** EVAL\_BODY\_BUFFERED = 2;  
 **void** setBodyContent(BodyContent var1);

**void** doInitBody() **throws** JspException;  
 }

* **其实使用BodyTag十分简单**
  + 如果doStartTag()方法返回的是EVAL\_BODY\_BUFFERED，把标签体的**内容缓存起来**
  + 调用setBodyContent()方法和doInitBody()方法，封装标签体内容到**BodyContent对象**中
  + 接着调用doEndTag()方法（**BodyContent继承着JspWriter，它与JspWriter最大的区别是：BodyContent类的任何写入的内容并不自动地向页面输出！**）
  + 对于标签体的内容，我们可以通过getBodyContenet()来获取！

我们在编写能够修改标签体内容的自定义标签的标签处理器类时，可以**继承和扩展BodyTagSupport类**，这相比实现BodyTag接口将简化开发工作。

### BodyTagSupport实现类

**源代码：**

**public** **class** **BodyTagSupport** **extends** TagSupport **implements** BodyTag {  
 **protected** BodyContent bodyContent;  
 **public** BodyTagSupport() {  
 }  
 **public** **int** doStartTag() **throws** JspException {  
 **return** 2;  
 }  
 **public** **int** doEndTag() **throws** JspException {  
 **return** **super**.doEndTag();  
 }  
 **public** **void** setBodyContent(BodyContent b) {  
 **this**.bodyContent = b;  
 }  
 **public** **void** doInitBody() **throws** JspException {  
 }  
 **public** **int** doAfterBody() **throws** JspException {  
 **return** 0;  
 }  
 **public** **void** release() {  
 **this**.bodyContent = **null**;  
 **super**.release();  
 }  
 **public** BodyContent getBodyContent() {  
 **return** **this**.bodyContent;  
 }  
 **public** JspWriter getPreviousOut() {  
 **return** **this**.bodyContent.getEnclosingWriter();  
 }  
 }

* BodyTagSupport主要扩充了以下的内容:
  + **把BodyContent直接定义为成员变量**，在获取标签体内容的时候就不需要通过getBodyContent()获取了
  + **提供获取JspWriter的方法**，不需要从pageConext中获取了
  + 以上的两个扩充都简化了我们的代码书写！

**BodyTagSupport类实现了BodyTag接口，TagDemo4继承 BodyTagSupport类**

public class TagDemo4 extends BodyTagSupport {

/\* 控制doStartTag()方法返回EVAL\_BODY\_BUFFERED

\* @see javax.servlet.jsp.tagext.BodyTagSupport#doStartTag()

\*/

@Override

public int doStartTag() throws JspException {

return BodyTag.EVAL\_BODY\_BUFFERED;

}

@Override

public int doEndTag() throws JspException {

//this.getBodyContent()得到代表标签体的bodyContent对象

BodyContent bodyContent = this.getBodyContent();

//拿到标签体

String content = bodyContent.getString();

//修改标签体里面的内容，将标签体的内容转换成大写

String result = content.toUpperCase();

try {

//输出修改后的内容

this.pageContext.getOut().write(result);

} catch (IOException e) {

throw new RuntimeException(e);

}

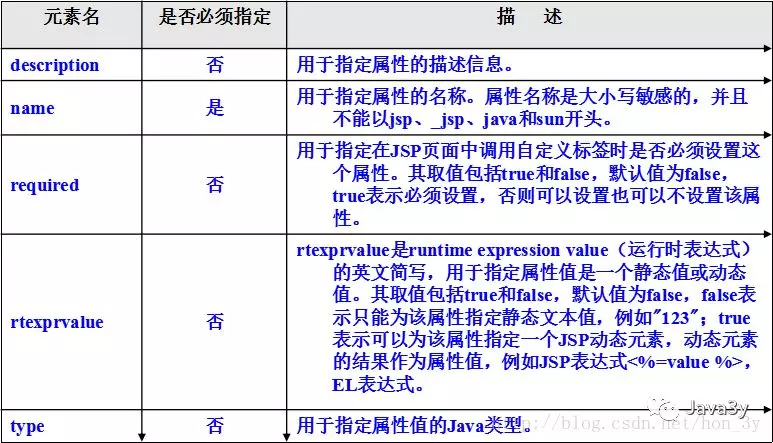
return Tag.EVAL\_PAGE;

}

}

### tld文件详细

**<tlib-version>**1.0**</tlib-version>**  
 **<short-name>**myshortname**</short-name>**  
 **<uri>**/xz **</uri>**  
 **<tag>**  
 **<name></name>**  
 **<tag-class></tag-class>**  
 **<body-content></body-content>**  
 **</tag>**

* shortname推荐使用prefix
* uri就是引入这个标签库使用的uri，<!-- 为自定义标签库**设置一个uri**，**uri以/开头，/后面的内容随便写**
* name为标签名
* tagclass为实现类
* bodycontent为标签体的限制，它有4个值：**EMPTY【不允许有标签体】，JSP【允许有JSP代码】 ，scriptless【不允许有脚本代码（也就是<%%>），允许有EL表达式，文本，JSP行为】 ， tagdepentend【标签体内的JSP代码不会被解析，直接输出文本】**

使用"**<%@taglib uri="标签库的uri"  prefix="标签的使用前缀"%>**"指令引入要使用的标签库。

<!-- 使用taglib指令引用标签库，标签库的**前缀(prefix)可以随便设置，**如这里设置成 prefix="xdp" -->

# JSP简单标签

## 快速入门

为了简化标签开发的复杂度，在JSP 2.0中定义了一个更为简单、便于编写和调用的SimpleTag接口来实现标签的功能。

**源码：**

**public** **interface** **SimpleTag** **extends** JspTag {  
 **void** doTag() **throws** JspException, IOException;  
 **void** setParent(JspTag var1);

JspTag getParent();  
 **void** setJspContext(JspContext var1);  
 **void** setJspBody(JspFragment var1);  
 }

* 明显地：
  + doTag()用于完成所有的标签逻辑，包括输出、迭代、修改标签体内容等。在doTag方法中可以**抛出javax.servlet.jsp.SkipPageException异常**，用于通知WEB容器不再执行JSP页面中位于结束标记后面的内容，这等效于在传统标签的doEndTag方法中返回Tag.SKIP\_PAGE常量的情况。
  + setJspContext(JspContext var1)是将**PageContext对象传递给标签处理器类**(PageContext是JspContext的子类)
  + setJspBody(JspFragment var1)把**代表标签体的JspFragment对象传**递给标签处理器对象

## SimpleTagSupport类的执行顺序

当web容器**开始执行标签时**，会调用如下方法完成标签的初始化：

1. WEB容器调用标签处理器对象的setJspContext方法，将代表JSP页面的pageContext对象传递给标签处理器对象。
2. WEB容器调用标签处理器对象的setParent方法，将父标签处理器对象传递给这个标签处理器对象。注意，只有在标签存在父标签的情况下，WEB容器才会调用这个方法。
3. 如果调用标签时设置了属性，容器将调用每个属性对应的setter方法把属性值传递给标签处理器对象。如果标签的属性值是EL表达式或脚本表达式，则**WEB容器首先计算表达式的值，然后把值传递给标签处理器对象。**
4. 如果简单标签有标签体，WEB容器将调用setJspBody方法把代表标签体的**JspFragment对象**传递进来。
5. 执行标签时WEB容器调用标签处理器的doTag()方法，开发人员在方法体内通过**操作JspFragment**对象，就可以实现是否执行、迭代、修改标签体的目的。

## 开发简单标签实现页面逻辑

### 控制jsp页面某一部分内容是否执行

public void doTag() throws JspException, IOException {

//得到代表jsp标签体的JspFragment

JspFragment jspFragment = this.getJspBody();

//得到jsp页面的的PageContext对象

//PageContext pageContext = (PageContext) jspFragment.getJspContext();

//调用JspWriter将标签体的内容输出到浏览器

//jspFragment.invoke(pageContext.getOut());

//将标签体的内容输出到浏览器

jspFragment.invoke(null);

}

在WEB-INF目录下新建一个simpletag.tld文件，然后在simpletag.tld文件中添加对该标签处理类的描述

<!--

为自定义标签库设置一个uri，uri以/开头，/后面的内容随便写，如这里的/simpletag ，

在Jsp页面中引用标签库时，需要通过uri找到标签库

在Jsp页面中就要这样引入标签库：<%@taglib uri="/simpletag" prefix="gacl"%>

-->

<uri>/simpletag</uri>

<!--一个taglib(标签库)中包含多个自定义标签，每一个自定义标签使用一个tag标记来描述 -->

<!-- 一个tag标记对应一个自定义标签 -->

<tag>

<description>SimpleTag(简单标签)Demo1</description>

<!--

为标签处理器类配一个标签名，在Jsp页面中使用标签时是通过标签名来找到要调用的标签处理器类的

通过demo1就能找到对应的me.gacl.web.simpletag.SimpleTagDemo1类

-->

<name>demo1</name>

<!-- 标签对应的处理器类-->

<tag-class>me.gacl.web.simpletag.SimpleTagDemo1</tag-class>

<!--

tld文件中有四种标签体类型 ：**empty JSP scriptless tagdepentend**

**在简单标签(SampleTag)中标签体body-content的值只允许是empty和scriptless，不允许设置成JSP,如果设置成JSP就会出现异常**

**在传统标签中标签体body-content的值只允许是empty和JSP**

如果标签体body-content的值设置成tagdepentend，那么就表示标签体里面的内容是给标签处理器类使用的，

例如：开发一个查询用户的sql标签，此时标签体重的SQL语句就是给SQL标签的标签处理器来使用的

<gacl:sql>SELECT \* FROM USER</gacl:sql>

在这种情况下，sql标签的<body-content>就要设置成tagdepentend，tagdepentend用得比较少，了解一下即可

-->

<body-content>scriptless</body-content>

</tag>

</taglib>

### 控制jsp页面内容重复执行

public void doTag() throws JspException, IOException {

// 得到代表jsp标签体的JspFragment

JspFragment jspFragment = this.getJspBody();

for (int i = 0; i < 5; i++) {

jspFragment.invoke(null);

}

### 修改jsp页面内容输出

编写一个类继承SimpleTagSupport，然后再重写doTag方法，在doTag方法调用jspFrament.invoke方法时，让执行结果写一个自定义的缓冲中即可，然后开发人员可以取出缓冲的数据修改输出。

public void doTag() throws JspException, IOException {

// 得到代表jsp标签体的JspFragment

JspFragment jspFragment = this.getJspBody();

StringWriter sw = new StringWriter();

//将标签体的内容写入到sw流中

jspFragment.invoke(sw);

//获取sw流缓冲区的内容

String content = sw.getBuffer().toString();

content = content.toUpperCase();

PageContext pageContext = (PageContext) this.getJspContext();

//将修改后的content输出到浏览器中

pageContext.getOut().write(content);

}

### 控制整个jsp页面是否执行

编写一个类继承SimpleTagSupport，然后再重写doTag方法，在doTag方法抛出SkipPageException异常即可，jsp收到这个异常，将忽略标签余下jsp页面的执行。

public void doTag() throws JspException, IOException {

//抛出一个SkipPageException异常就可以控制标签余下的Jsp不执行

throw new SkipPageException();

}

## 简单标签开发的一些注意细节

### 标签类编写细节

应该继承SimpleTagSupport类，SimpleTagSupport类是SimpleTag接口的一个默认实现类，通过继承SimpleTagSupport类，就可以直接使用SimpleTagSupport类已经实现的那些方法，如果SimpleTagSupport类的方法实现不满足业务要求，那么就可以根据具体的业务情况将相应的方法进行重写。

### tld文件中标签体类型设置细节

开发好一个标签后，在tld文件中使用<tag>来描述一个标签，描述的内容包括标签名(name)，标签处理器类(tag-class)，标签体的内容(body-content)。

　　tld文件中有四种标签体(body-content)类型 ：**empty、JSP、scriptless、tagdependent**

**简单标签标签体的细节注意问题：**  
     **在简单标签(SampleTag)中标签体body-content的值只允许是empty、scriptless、tagdependent，不允许设置成JSP,如果设置成JSP就会出现异常**：

**body-content的值**如果设置成**empty**，那么**就表示该标签没有标签体**，如果是设置成**scriptless**，那么**表示该标签是有标签体的，但是标签体中的内容不可以是<%java代码%>**

如果**传统标签**和**简单标签**的标签体body-content的值设置成**tagdependent，**那么就表示标签体里面的内容是给标签处理器类使用的，tagdependent用得比较少，了解一下即可

### tld文件中标签库的uri设置细节

在两个标签库的引用uri一样的情况下，为了能够在jsp中区别到底引用的是哪个标签库，可以换一种引用方式：**<%@taglib uri="要引用的标签库的tld文件目录" prefix="gacl"%>**，使用taglib指令引入标签库时，taglib指令的uri属性指定为标签库的tld文件目录，这样就可以区别开了，例如：

　　引用gacl.tld标签库：<%@taglib uri="**/WEB-INF/tag.tld**" prefix="gacl"%>、

　　引用simpletag.tld标签库：<%@taglib uri="**/WEB-INF/simpletag.tld**" prefix="gacl"%>

　　所以当在项目中引用了多个标签库，如果标签库的uri刚好是一样的，就可以用这种方式解决。

## JspFragment类介绍

WEB容器在处理简单标签的标签体时，会把标签体内容用一个JspFragment对象表示，并调用标签处理器对象的setJspBody方法把JspFragment对象传递给标签处理器对象。

JspFragment类中只定义了**两个方法**，如下所示：

　　getJspContext方法 //用于返回代表调用页面的JspContext对象.  
　　public abstract void invoke(java.io.Writer out)

用于执行JspFragment对象所代表的JSP代码片段，参数out用于指定将JspFragment对象的执行结果**写入到哪个输出流对象**中，如果传递给参数out的值为null，则将执行结果写入到JspContext.getOut()方法返回的输出流对象中。(简而言之，可以理解为写给浏览器)

**invoke方法详解**

在标签处理器中如果没有调用JspFragment.invoke方法，其结果就相当于忽略标签体内容；  
　　在标签处理器中重复调用JspFragment.invoke方法，则标签体内容将会被重复执行；  
　　若想在标签处理器中修改标签体内容，只需在调用invoke方法时指定一个可取出结果数据的输出流对象（例如StringWriter），让标签体的执行结果输出到该输出流对象中，然后从该输出流对象中取出数据进行修改后再输出到目标设备，即可达到修改标签体的目的。

如果标签的属性值是8种基本数据类型，那么在JSP页面在传递字符串时，JSP引擎会自动转换成相应的类型，但如果标签的属性值是复合数据类型，那么JSP引擎是无法自动转换的。

<%--如果一定要给标签的复合属性赋值，那么可以**采用表达式的方式给复合属性赋值**，如下所示： --%>

<%

Date d = new Date();

request.setAttribute("date", d);

%>

<gacl:demo6 date="${date}"/>

<hr/>

<gacl:demo6 date="<%=new Date()%>"/>

## 自定义标签的应用

### 防盗链

模拟下场景：**1.jsp页面是海贼王资源，2.jsp页面提示非法盗链，index1.jsp是我的首页。别人想要看我的海贼王资源，就必须通过我的首页点进去看，否则就是非法盗链！**

* **编写标签处理器类**

public class RefererTag extends SimpleTagSupport {

/\*\*

\* 网站域名

\*/

private String site;

/\*\*

\* 要跳转的页面

\*/

private String page;

@Override

public void doTag() throws JspException, IOException {

//获取jsp页面的PageContext对象

PageContext pageContext = (PageContext) this.getJspContext();

//通过PageContext对象来获取HttpServletRequest对象

HttpServletRequest request = (HttpServletRequest) pageContext.getRequest();

//获取请求的来路(Referer)

String referer = request.getHeader("referer");

//如果来路是null或者来路不是来自我们自己的site，那么就将请求重定向到page页面

if (referer == null || !referer.startsWith(site)) {

//获取HttpServletResponse对象

HttpServletResponse response = (HttpServletResponse)pageContext.getResponse();

String webRoot = request.getContextPath();

if (page.startsWith(webRoot)) {

//重定向到page页面

response.sendRedirect(page);

} else {

//重定向到page页面

response.sendRedirect(webRoot+page);

}

//重定向后，控制保护的页面不要执行

throw new SkipPageException();

}

}

public void setSite(String site) {

this.site = site;

}

public void setPage(String page) {

this.page = page;

}

}

* **在WEB-INF目录下tld文件中添加对该标签的描述**

<tag>

<!-- 标签名 -->

<name>referer</name>

<!-- 标签处理器类 -->

<tag-class>me.gacl.web.simpletag.RefererTag</tag-class>

<!-- 标签体允许的内容 -->

<body-content>empty</body-content>

<!-- 标签的属性描述 -->

<attribute>

<description>描述标签的site属性</description>

<!-- 标签的site属性 -->

<name>site</name>

<required>true</required>

<!-- rtexprvalue用来指示标签的属性值是否可以是一个表达式， 一般设置为true,true就表示允许标签的属性值可以是一个表达式 -->

<rtexprvalue>true</rtexprvalue>

</attribute>

<attribute>

<description>描述标签的page属性</description>

<!-- 标签的page属性 -->

<name>page</name>

<required>true</required>

<!-- rtexprvalue用来指示标签的属性值是否可以是一个表达式， 一般设置为true,true就表示允许标签的属性值可以是一个表达式 -->

<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

</taglib>

* **测试**

<%@ page language="java" pageEncoding="UTF-8"%>

<%--在jsp页面中导入自定义标签库 --%>

<%@taglib uri="/gaclTagLib" prefix="gacl" %>

<%--在jsp页面中也可以使用这种方式导入标签库，直接把uri设置成标签库的tld文件所在目录

<%@taglib uri="/WEB-INF/TagLib.tld" prefix="gacl"%>

--%>

<%--在Jsp页面中使用防盗链标签

当用户尝试直接通过URL地址(http://localhost:8080/JavaWeb\_JspTag\_study\_20140816/simpletag/refererTagTest.jsp)访问这个页面时，

防盗链标签的标签处理器内部就会进行处理，将请求重定向到/index.jsp

--%>

<gacl:referer site="http://localhost:8080" page="/index.jsp"/>

<!DOCTYPE HTML>

<html>

<head>

<title>防盗链标签测试</title>

</head>

<body>

网站内部资料

</body>

</html>

### forEach标签

**public** **class** **Demo1** **extends** SimpleTagSupport {  
 //遍历的是List集合，定义List集合成员变量  
 **private** List items;  
 //以var为关键字存储到PageContext  
 **private** String var;  
 @Override  
 **public** **void** doTag() **throws** JspException, IOException {  
 //获取到集合的迭代器  
 Iterator iterator = items.iterator();  
 //获取到代表标签体内容的对象  
 JspFragment jspFragment = **this**.getJspBody();  
 //遍历集合  
 **while** (iterator.hasNext()) {  
 Object o = iterator.next();  
 //把遍历出来的对象存储到page范围中，关键字为标签的属性var（在标签体中使用EL表达式${var}，就能够获取到集合的对象了！）  
 **this**.getJspContext().setAttribute(var, o);  
 //每设置了一个属性，我就执行标签体  
 jspFragment.invoke(**null**);  
 }  
 }  
 **public** **void** setItems(List items) {  
 **this**.items = items;  
 }  
 **public** **void** setVar(String var) {  
 **this**.var = var;  
 }  
}

**测试的jsp代码**

**<%**  
 List list = **new** ArrayList();  
 list.add("zhongfucneng");  
 list.add("1");  
 list.add("2");  
 list.add("3");  
 request.setAttribute("list",list);  
 **%>**  
 **<zhongfucheng:forEach** items="${list}" var="str"**>**  
 ${str}  
 **</zhongfucheng:forEach>**

上面写的仅仅能够遍历List集合，做一个通用的forEach标签麻烦的是在：**不知道传进来的是什么类型的数组、什么类型集合！，需要逐一去判断**

* **我们的实现思路就是将所有的集合或数组都转成是Collection类型的！**

还有int[],byte[],char[]等八大基本数据类型，这八大基本数据类型就**不能用Arrays.asList()**把引用传进去了。因为JDK5以后会把引用自动装箱成Interger[]、Byte[]等等，变成对象了。而不是获取到数组的元素数据。

**由于JDK5的新特性，我们又有另外的解决方案，Class对象能够判断是否为数组类，reflect反射包下Array类**：

* 其实，**无论Map集合、还是任何类型的数组、都可以使用Colletion进行遍历！。**
* **如果是八大基本数据类型的数组，我们就使用反射来进行构建出Collection集合。**

**public** **class** **Demo1** **extends** SimpleTagSupport {  
 //遍历的是未知的集合或数组，定义成Object  
 **private** Object items;  
 //每次被遍历的对象存储关键字  
 **private** String var;  
 //Colletion  
 **private** Collection collection;  
 //在WEB容器设置标签的属性的时候，判断是什么类型的数组和集合  
 **public** **void** setItems(Object items) {  
 **this**.items = items;  
 //如果items是Collection类型的，就强转为Colletion  
 **if** (items **instanceof** Collection) {  
 collection = (Collection) items;  
 }  
 //如果itmes是Map类型的，那么就强转为Map，再获取到<Map.Entry<K, V>，这个是Set集合的！  
 **if** (items **instanceof** Map) {  
 Map map = (Map) items;  
 collection = (Collection) map.entrySet();  
 }  
 //可以这样解决，Class对象判断是否是一个数组类  
 **if** (items.getClass().isArray()) {  
 //创建Collection集合添加数组的元素！  
 collection = **new** ArrayList();  
 //再利用reflect包下的Array类获取到该数组类的长度  
 **int** len = Array.getLength(items);  
 //遍历并添加到集合中  
 **for** (**int** i = 0; i < len; i++) {  
 collection.add(Array.get(items, i));  
 }  
 }  
 }  
 **public** **void** setVar(String var) {  
 **this**.var = var;  
 }  
 @Override  
 **public** **void** doTag() **throws** JspException, IOException {  
 //获取到代表标签体内容的对象  
 JspFragment jspFragment = **this**.getJspBody();  
 Iterator iterator = collection.iterator();  
 //遍历集合  
 **while** (iterator.hasNext()) {  
 Object o = iterator.next();  
 //把遍历出来的对象存储到page范围中（在标签体中使用EL表达式${var}，就能够获取到集合的对象了！）  
 **this**.getJspContext().setAttribute(var, o);  
 jspFragment.invoke(**null**);  
 }  
 }  
}

**测试代码**：

<%

/\*list集合\*/

List list = new ArrayList();

list.add("zhongfucneng");

list.add("1");

list.add("2");

list.add("3");

request.setAttribute("list",list);

/\*基本数据类型数组\*/

int[] ints = new int[]{1, 2, 3, 4, 5};

request.setAttribute("ints", ints);

/\*对象数组\*/

Object[] objects = new Object[]{2, 3, 4, 5, 6};

request.setAttribute("objects", objects);

/\*map集合\*/

Map map = new HashMap();

map.put("aa", "aa");

map.put("bb", "bb");

map.put("cc", "cc");

request.setAttribute("map",map);

%>

List集合：

<zhongfucheng:forEach items="${list}" var="str">

${str}

</zhongfucheng:forEach>

<hr>

<hr>

基本数据类型数组:

<zhongfucheng:forEach items="${ints}" var="i">

${i}

</zhongfucheng:forEach>

<hr>

<hr>

对象数组：

<zhongfucheng:forEach items="${objects}" var="o">

${o}

</zhongfucheng:forEach>

<hr>

<hr>

map集合：

<zhongfucheng:forEach items="${map}" var="me">

${me.key} = ${me.value}

</zhongfucheng:forEach>

### HTML转义标签

要开发这个标签就很简单了，只要获取到标签体的内容，再通过经过方法转义下标签体内容，输出给浏览器即可！

标签处理器代码：

public class Demo1 extends SimpleTagSupport {

@Override

public void doTag() throws JspException, IOException {

//获取到标签体的内容再修改

StringWriter stringWriter = new StringWriter();

JspFragment jspFragment = this.getJspBody();

jspFragment.invoke(stringWriter);

String content = stringWriter.toString();

//经过filter()转义，该方法在Tomcat可以找到

content = filter(content);

//再把转义后的内容输出给浏览器

this.getJspContext().getOut().write(content);

}

private String filter(String message) {

if (message == null)

return (null);

char content[] = new char[message.length()];

message.getChars(0, message.length(), content, 0);

StringBuffer result = new StringBuffer(content.length + 50);

for (int i = 0; i < content.length; i++) {

switch (content[i]) {

case '<':

result.append("&lt;");

break;

case '>':

result.append("&gt;");

break;

case '&':

result.append("&amp;");

break;

case '"':

result.append("&quot;");

break;

default:

result.append(content[i]);

}

}

return (result.toString());

}

}

测试代码：

<zhongfucheng:filter><a href="2.jsp">你好啊</a> </zhongfucheng:filter>

<br>

<a href="2.jsp">你好啊

### if else标签

在JSTL中并没有if else的标签，JSTL给予我们的是choose，when，otherwise标签，现在我们模仿choose，when，otherwise开发标签

思路：when标签有个test属性，但otherwise怎么判断标签体是执行还是不执行呢？这时就需要choose标签的支持了！choose标签默认定义一个Boolean值为false，。当when标签体被执行了，就把Boolean值变成true，只要Boolean值为false就执行otherwise标签体的内容。

看程序就容易理解上面那句话了：！

choose标签处理器

public class Choose extends SimpleTagSupport {

private boolean flag;

@Override

public void doTag() throws JspException, IOException {

this.getJspBody().invoke(null);

}

public boolean isFlag() {

return flag;

}

public void setFlag(boolean flag) {

this.flag = flag;

}

}

When标签处理器

public class When extends SimpleTagSupport {

private boolean test ;

@Override

public void doTag() throws JspException, IOException {

Choose choose = (Choose) this.getParent();

//如果test为true和flag为false，那么执行该标签体

if (test == true && choose.isFlag() == false) {

this.getJspBody().invoke(null);

//修改父标签的flag

choose.setFlag(true);

}

}

public void setTest(boolean test) {

this.test = test;

}

}

OtherWise标签处理器

public class OtherWise extends SimpleTagSupport {

@Override

public void doTag() throws JspException, IOException {

Choose choose = (Choose) this.getParent();

//如果父标签的flag为false，就执行标签体（如果when标签没执行，flag值就不会被修改！when标签没执行，就应该执行otherwise标签！）

if (choose.isFlag() == false) {

getJspBody().invoke(null);

//改父标签的flag为false

choose.setFlag(true);

}

}

}

测试代码：

<zhongfucheng:choose>

<zhongfucheng:when test="${user!=null}">

user为空

</zhongfucheng:when>

<zhongfucheng:otherwise>

user不为空

</zhongfucheng:otherwise>

</zhongfucheng:choose>

### 将taglib项目打包成jar包

# [JSTL标签库之核心标签](https://www.cnblogs.com/xdp-gacl/p/3943390.html)

**JSTL全称为 JSP Standard Tag Library 即JSP标准标签库**。

JSTL作为最基本的标签库，**提供了一系列的JSP标签，实现了基本的功能：集合的遍历、数据的输出、字符串的处理、数据的格式化等等！**

* **Scriptlet的可读性，维护性，重用性都十分差！**JSTL与HTML代码十分类似，遵循着XML标签语法，**使用JSTL让JSP页面显得整洁，可读性非常好，重用性非常高，可以完成复杂的功能！**
* 在JSP中不推荐使用scriptlet输出，推荐使用JSP标签。

**使用JSTL标签库步骤：**

1. **导入jstl.jar和standard.jar开发包**
2. **在JSP页面中用tablib指令引入需要用到的JSTL标签**

**JSTL标签库的分类**

* **核心标签(用得最多)**
* 国际化标签(I18N格式化标签)
* 数据库标签(SQL标签，很少使用)
* XML标签(几乎不用)
* JSTL函数(EL函数)

**JSTL的核心标签库标签共13个**，使用这些标签能够完成JSP页面的基本功能，减少编码工作。

　　从功能上可以分为4类：表达式控制标签、流程控制标签、循环标签、URL操作标签。  
　　　　（1）**表达式控制标签**：**out标签、set标签、remove标签、catch标签。**  
　　　　（2）**流程控制标签**：**if标签、choose标签、when标签、otherwise标签**。  
　　　　（3）**循环标签**：**forEach标签、forTokens标签**。  
　　　　（4）**URL操作标签**：**import标签、url标签、redirect标签、param标签**。

**core标签库**

* **core标签库是JSTL的核心标签库，实现了最基本的功能：流程控制、迭代输出等操作**！
* **core标签库的前缀一般是c**

## c:out标签

<c:out>标签主要是**用来输出数据对象（字符串、表达式）的内容或结果**。  
　　在使用Java脚本输出时常使用的方式为： <% out.println(“字符串”)%> 或者 <%=表达式%> ，在web开发中，为了避免暴露逻辑代码会尽量减少页面中的Java脚本，使用<c:out>标签就可以实现以上功能。

**1 <c:out value=”字符串”>**

**2 <c:out value=”EL表达式”>**

　　JSTL的使用是和EL表达式分不开的，EL表达式虽然可以直接将结果返回给页面，但有时得到的结果为空，<c:out>有特定的结果处理功能，EL的单独使用会降低程序的易读性**，建议把EL的结果输入放入<c:out>标签中。**

<body>

<h3><c:out value="下面的代码演示了c:out的使用，以及在不同属性值状态下的结果。"/></h3>

<hr/>

<ul>

<%--（1）直接输出了一个字符串。 --%>

<li>（1）<c:out value="JSTL的out标签的使用" /></li>

**<li>（2）<c:out value="<a href='http://www.cnblogs.com/'>点击链接到博客园</a>" /></li>**

**<%--escapeXml="false"表示value值中的html标签不进行转义，而是直接输出 --%>**

**<li>（3）<c:out value="<a href='http://www.cnblogs.com/'>点击链接到博客园</a>" escapeXml="false"/></li>**

<%--(4）字符串中有转义字符，但在默认情况下没有转换。 --%>

<li>（4）<c:out value="&lt未使用字符转义&gt" /></li>

<%--（5）使用了转义字符&lt和&gt分别转换成<和>符号。 --%>

<li>（5）<c:out value="&lt使用字符转义&gt" **escapeXml="false"**></c:out></li>

<%--（6）设定了默认值，从EL表达式${null}得到空值，所以直接输出设定的默认值。 --%>

<li>（6）<c:out value="${null}">使用了默认值</c:out></li>

<%--（7）未设定默认值，输出结果为空。 --%>

<li>（7）<c:out value="${null}"></c:out></li>

<%--（8）设定了默认值，从EL表达式${null}得到空值，所以直接输出设定的默认值。 --%>

<li>（8）<c:out value="${null}" default="默认值"/></li>

<%--（9）未设定默认值，输出结果为空。 --%>

<li>（9）<c:out value="${null}"/></li>

我们发现上面的代码实现的效果和EL表达式是一样的，它出色的地方就多了两个属性,**default和escapeXml属性**。如果我们用到这两个属性，我们就使用该标签，如果没有用到这两个属性就用EL表达式就可以了。

## set标签

该标签有5个属性，用起来有稍微有些复杂了！现在要记住的就是：**var属性操作的是Integer、Double、Float、String等类型的数据，target属性操作的是JavaBean或Map对象的数据，scope代表的是Web域，value是值，property是对象的属性**！

既然var属性只能操作Integer、Double、String等类型**，那么存在var属性就一定没有property属性**（property代表的是对象的成员属性，Integer、String这些类型哪来的成员变量呀）

<c:set>标签用于把某一个对象存在指定的域范围内，或者将某一个对象存储到Map或者JavaBean对象中。

### <c:set>标签的语法

<c:set>标签的编写共有4种语法格式。

　　语法1：存值，把一个值放在指定的域范围内。

　　　　<c:set value=”值1” var=”name1” [scope=”page|request|session|application”]/>

　　　　含义：把一个变量名为name1值为“值1”的变量存储在指定的scope范围内。

　　语法2：

　　　　<c:set var=”name2” [scope=”page|request|session|application”]>

　　　　　　值2

　　　　</c:set>

　　　　含义：把一个变量名为name2，值为值2的变量存储在指定的scope范围内。

　　语法3：

　　　　<c:set value=”值3” target=”JavaBean对象” property=”属性名”/>

　　　　含义：把一个值为“值3”赋值给指定的JavaBean的属性名。相当与setter()方法。

　　语法4：

　　　　<c:set target=”JavaBean对象” property=”属性名”>

　　　　　　值4

　　　　</c:set>

　　　　含义：把一个值4赋值给指定的JavaBean的属性名。

从功能上分语法1和语法2、语法3和语法4的效果是一样的，只是把value值放置的位置不同，至于使用那个根据个人的喜爱，语法1和语法2是向scope范围内存储一个值，语法3和语法4是给指定的JavaBean赋值。

### <c:set>标签的使用范例

**使用var属性和scope属性实现计数器**

<%--由于下面变量需要做加法运算，所以要定义出来，不然服务器是不知道我的变量是Integer类型的--%>

<%

Integer sessionCount = 0;

Integer applicationCount = 0;

%>

<c:set var="sessionCount" value="${sessionCount+1}" scope="session"/>

<c:set var="applicationCount" value="${applicationCount+1}" scope="application"/>

* **使用target属性与之配对的是property属性，target属性只能操作JavaBean或Map对象，property就是对应的成员变量或key了。**。
* **既然target属性操作的是JavaBean或Map对象，那么一定是通过EL表达式来获取到对象了。taget属性如果获取不到数据会抛出异常！使用target属性就一定没有scope属性**

<%--创建出JavaBean对象，设置为session范围的属性--%>

<jsp:useBean id="person" class="domain.Person" scope="session"/>

<%--获取到person对象，设置age属性的值为32--%>

<c:set target="${person}" property="age" value="32"/>

${person.age}

<%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>

<%--引入JSTL核心标签库 --%>

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

<%--使用JSP的指令元素指定要使用的JavaBean --%>

<jsp:useBean id="person" class="javabean.Person"/>

<%--**负责实例化Bean**，id指定实例化后的对象名，可以通过${person}得到person在内存中的值

（或者使用person.toString()方法）--%>

<!DOCTYPE HTML>

<html>

<head>

<title>JSTL: --表达式控制标签“set”标签的使用</title>

</head>

<body>

<h3>代码给出了给指定scope范围赋值的示例。</h3>

<ul>

<%--通过<c:set>标签将data1的值放入page范围中。--%>

<li>把一个值放入page域中:<c:set var="data1" value="xdp" scope="page"/></li>

<%--使用EL表达式从pageScope得到data1的值。--%>

<li>从page域中得到值：${pageScope.data1}</li>

<%--通过<c:set>标签将data2的值放入request范围中。--%>

<li>把一个值放入request域中:<c:set var="data2" value="gacl" scope="request"/></li>

<%--使用EL表达式从requestScope得到data2的值。--%>

<li>从request域中得到值：${requestScope.data2}</li>

<%--通过<c:set>标签将值name1的值放入session范围中。--%>

<li>把一个值放入session域中。<c:set value="孤傲苍狼" var="name1" scope="session"></c:set></li>

<%--使用EL表达式从sessionScope得到name1的值。--%>

<li>从session域中得到值:${sessionScope.name1} </li>

<%--把name2放入application范围中。 --%>

<li>把一个值放入application域中。<c:set var="name2" scope="application">白虎神皇</c:set></li>

<%--使用EL表达式从application范围中取值，用<c:out>标签输出使得页面规范化。 --%>

<li>使用out标签和EL表达式嵌套从application域中得到值：

<c:out value="${applicationScope.name2}">未得到name的值</c:out>

</li>

<%--不指定范围使用EL自动查找得到值 --%>

<li>未指定scope的范围，会从不同的范围内查找得到相应的值：${data1}、${data2}、${name1}、${name2}</li>

</ul>

<hr/>

<h3>使用Java脚本实现以上功能</h3>

<ul>

<li>把一个值放入page域中。<%pageContext.setAttribute("data1","xdp");%></li>

<li>从page域中得到值:<%out.println(pageContext.getAttribute("data1"));%></li>

<li>把一个值放入request域中。<%request.setAttribute("data2","gacl");%></li>

<li>从request域中得到值:<%out.println(request.getAttribute("data2"));%></li>

<li>把一个值放入session域中。<%session.setAttribute("name1","孤傲苍狼");%></li>

<li>从session中域得到值:<%out.println(session.getAttribute("name1"));%></li>

<%--out.println()方法与<%=%>表达式输出功能一样

但使用表达式输出（<%=%>）明显要比使用out.println()输出更好。

--%>

<li><%=session.getAttribute("name1") %></li>

<li>把另一个值放入application域中。<%application.setAttribute("name2","白虎神皇");%></li>

<li> 从application域中得到值：<%out.println(application.getAttribute("name2"));%></li>

<li><%=application.getAttribute("name2")%></li>

<li>未指定scope的范围，会从不同的范围内查找得到相应的值：

<%=pageContext.findAttribute("data1")%>、

<%=pageContext.findAttribute("data2")%>、

<%=pageContext.findAttribute("name1")%>、

<%=pageContext.findAttribute("name2")%>

</li>

</ul>

<hr/>

<h3>操作JavaBean，设置JavaBean的属性值</h3>

<%--设置JavaBean的属性值，等同与setter方法，**Target指向实例化后的对象**，property指向要插入值的参数名。

**注意：使用target时一定要指向实例化后的JavaBean对象，也就是要跟<jsp:useBean>配套使用,也可以java脚本实例化，但这就失去了是用标签的本质意义。**

使用Java脚本实例化：

<%@page import="javabean.Person"%>

<% Person person=new Person(); %>

<c:set target="${person}" property="name">孤傲苍狼</c:set>

<c:set target="${person}" property="age">25</c:set>

<c:set target="${person}" property="sex">男</c:set>

<c:set target="${person}" property="home">中国</c:set>

<ul>

<li>使用的目标对象为：${person}</li>

<li>从Bean中获得的name值为：<c:out value="${person.name}"></c:out></li>

<li>从Bean中获得的age值为：<c:out value="${person.age}"></c:out></li>

<li>从Bean中获得的sex值为：<c:out value="${person.sex}"></c:out></li>

<li>从Bean中获得的home值为：<c:out value="${person.home}"></c:out></li>

</ul>

<hr/>

<h3>操作Map</h3>

<%

Map map = new HashMap();

request.setAttribute("map",map);

%>

<%--将data对象的值存储到map集合中 --%>

<c:set property="data" value="gacl" target="${map}"/>

${map.data}

</body>

</html>

**运行结果:**

## remove标签

**<c:remove>标签主要用来从指定的JSP范围内移除指定的变量。**

<c:remove var=”变量名” [scope=”page|request|session|application”]/>,其中var属性是必须的，scope可以以省略。

<%--创建出JavaBean对象，设置为session范围的属性--%>

<jsp:useBean id="person" class="domain.Person" scope="session"/>

<%--获取到person对象，设置age属性的值为32--%>

<c:set target="${person}" property="age" value="32"/>

${person.age}

<br>

<%--删除session属性--%>

<c:remove var="person" scope="session"></c:remove>

${person.age==null?"存在session的person对象被删除了！":"我还在呢！"。

## c:catch标签

<c:catch>标签用于捕获嵌套在标签体中的内容抛出的异常。

其语法格式如下：<c:catch [var="varName"]>容易产生异常的代码</c:catch>  
　　var属性用于标识<c:catch>标签捕获的异常对象，它将保存在page这个Web域中。

<%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>

<%--引入JSTL核心标签库 --%>

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

<!DOCTYPE HTML>

<html>

<head>

<title>JSTL: --表达式控制标签“catch”标签实例</title>

</head>

<body>

<h4>catch标签实例</h4>

<hr>

<%--把容易产生异常的代码放在<c:catch></c:catch>中，

自定义一个变量errorInfo用于存储异常信息 --%>

<%--创建出JavaBean对象，设置为session范围的属性--%>

<jsp:useBean id="person" class="domain.Person" scope="session"/>

<c:catch var="errorInfo">

<%--实现了一段异常代码，向一个不存在的JavaBean中插入一个值--%>

<c:set target="person" property="hao"></c:set>

</c:catch>

<%--用EL表达式得到errorInfo的值，并使用<c:out>标签输出 --%>

异常：<c:out value="${errorInfo}" /><br />

异常 errorInfo.getMessage：<c:out value="${errorInfo.message}" /><br />

异常 errorInfo.getCause：<c:out value="${errorInfo.cause}" /><br />

异常 errorInfo.getStackTrace：<c:out value="${errorInfo.stackTrace}" />

</body>

</html>

## C:\Users\Administrator\AppData\Local\Microsoft\Windows\Temporary Internet Files\Content.MSO\365490E8.tmpc:if标签

【语法1】：没有标签体内容(body)

　　　　<c:if test="testCondition" var="varName" [scope="{page|request|session|application}"]/>

【语法2】：有标签体内容

　　　　<c:if test="testCondition" [var="varName"] [scope="{page|request|session|application}"]>

　　　　　　标签体内容

　　　　</c:if>

【参数说明】：  
　　　　（1）test属性用于**存放判断的条件，一般使用EL表达式来编写。**  
　　　　（2）**var属性用来存放判断的结果，**类型为true或false。  
　　　　（3）scopes属性用来指定var属性存放的范围。

**根据传递过来的参数的不同显示不同的页面！**

<%--如果带过来的名字是zhongfucheng，那么可以登陆--%>

<c:if test="${param.name=='zhongfucheng'}">

用户名：<input type="text" name="username"><br>

密码：<input type="password" name="password"><br>

<input type="submit" value="登陆">

</c:if>

<%--如果带过来的名字是ouzicheng，那么就是注册--%>

<c:if test="${param.name=='ouzicheng'}">

用户名：<input type="text" name="username"><br>

密码：<input type="password" name="password"><br>

<input type="submit" value="注册">

</c:if>

## c:choose标签

**<c:choose>、<c:when>和<c:otherwise>这3个标签通常情况下是一起使用的，<c:choose>标签作为<c:when>和<c:otherwise>标签的父标签来使用。**

<c:choose>

<c:when test="条件1">

　　　　　　//业务逻辑1

<c:when>

　　　<c:when test="条件2">

　　　　　　//业务逻辑2

<c:when>

　　　<c:when test="条件n">

　　　　　　//业务逻辑n

<c:when>

<c:otherwise>

　　　　　　//业务逻辑

　　　</c:otherwise>

　</c:choose>

<body>

<h4>choose及其嵌套标签示例</h4>

<hr/>

<%--通过set标签设定score的值为85 --%>

<c:set var="score" value="85"/>

<c:choose>

<%--使用<c:when>进行条件判断。

如果大于等于90，输出“您的成绩为优秀”；

如果大于等于70小于90，输出“您的成绩为良好”；

大于等于60小于70，输出“您的成绩为及格”；

其他（otherwise）输出“对不起，您没能通过考试”。

--%>

<c:when test="${score>=90}">

你的成绩为优秀！

</c:when>

<c:when test="${score>70 && score<90}">

您的成绩为良好!

</c:when>

<c:when test="${score>60 && score<70}">

您的成绩为及格

</c:when>

<c:otherwise>

对不起，您没有通过考试！

</c:otherwise>

</c:choose>

</body>

</html>

## c:forEach标签

该标签根据循环条件遍历集合（Collection）中的元素。

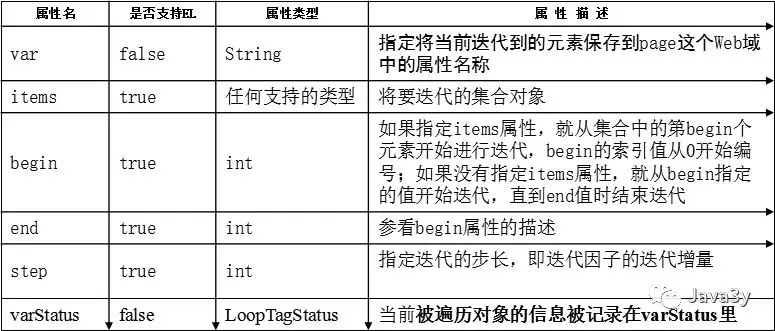
　<c:forEach   
    　　var=”name”   
    　　items=”Collection”   
    　　varStatus=”StatusName”   
    　　begin=”begin”   
    　　end=”end”   
    　　step=”step”>  
    本体内容  
 </c:forEach>

【参数解析】：  
　　（1）var设定变量名用于存储从集合中取出元素。  
　　（2）items指定要遍历的集合。  
　　（3）varStatus设定变量名，该变量用于存放集合中元素的信息。      
　　（4）begin、end用于指定遍历的起始位置和终止位置（可选）。  
　　（5）step指定循环的步长。

* **varStatus代表着当前对象被迭代的信息，它有以下的属性**。
  + **index【返回当前是第几个对象，从0开始计数】**
  + **count【已经遍历多少个对象了，从1开始计数】**
  + first【是否是第一个】
  + last【是否是最后一个】
  + **current【当前被迭代的对象】**
  + begin【开始的位置】
  + end【最后的位置】
  + step【步长】

<c:forEach var="list" items="${list}" varStatus="varStatus" >

${list}您的下标是:${varStatus.index}<br>

</c:forEach>

<%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>

<%--引入JSTL核心标签库 --%>

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

<%@page import="java.util.ArrayList"%>

<!DOCTYPE HTML>

<html>

<head>

<title>JSTL: -- forEach标签实例</title>

</head>

<body>

<h4><c:out value="forEach实例"/></h4>

<%

List<String>list = new ArrayList<String>();

list.add(0, "贝贝");

list.add(1, "晶晶");

list.add(2, "欢欢");

list.add(3, "莹莹");

list.add(4, "妮妮");

request.setAttribute("list", list);

%>

<B><c:out value="不指定begin和end的迭代：" /></B><br>

<%--不使用begin和end的迭代，从集合的第一个元素开始，遍历到最后一个元素。 --%>

<c:forEach var="fuwa" items="${list}">

&nbsp;<c:out value="${fuwa}"/><br/>

</c:forEach>

<B><c:out value="指定begin和end的迭代：" /></B><br>

<%--指定begin的值为1、end的值为3、step的值为2，

从第二个开始首先得到晶晶，每两个遍历一次，

则下一个显示的结果为莹莹，end为3则遍历结束。 --%>

<c:forEach var="fuwa" items="${list}" begin="1" end="3" step="2">

&nbsp;<c:out value="${fuwa}"/><br/>

</c:forEach>

<B><c:out value="输出整个迭代的信息：" /></B><br>

<%--指定varStatus的属性名为s，并取出存储的状态信息 --%>

<c:forEach var="fuwa"

items="${list}"

begin="3"

end="4"

varStatus="s"

step="1">

&nbsp;<c:out value="${fuwa}" />的四种属性：<br>

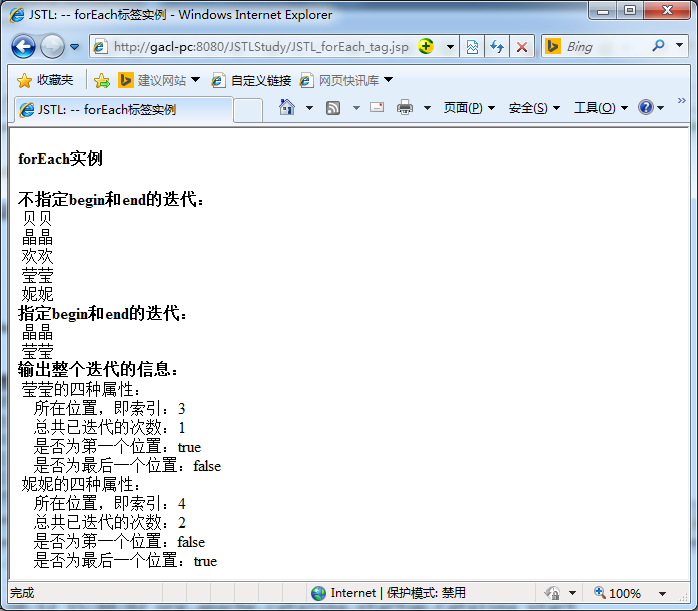
&nbsp;&nbsp;&nbsp;&nbsp;所在位置，即索引：<c:out value="${s.index}" /><br>

&nbsp;&nbsp;&nbsp;&nbsp;总共已迭代的次数：<c:out value="${s.count}" /><br>

&nbsp;&nbsp;&nbsp;&nbsp;是否为第一个位置：<c:out value="${s.first}" /><br>

&nbsp;&nbsp;&nbsp;&nbsp;是否为最后一个位置：<c:out value="${s.last}" /><br>

</c:forEach>

 </body>

## c:forTokens标签

该标签用于浏览字符串，并根据指定的字符将字符串截取。

语法：  
<c:forTokens items=”strigOfTokens”   
            delims=”delimiters”  
            [var=”name”   
            begin=”begin”   
            end=”end”   
            step=”len”   
            varStatus=”statusName”] >  
    本体内容  
</c:forTokens>  
【参数说明】  
　　（1）items指定被迭代的字符串。  
　　（2）delims指定使用的分隔符。  
　　（3）var指定用来存放遍历到的成员。  
　　（4）begin指定遍历的开始位置（int型从取值0开始）。  
　　（5）end指定遍历结束的位置（int型，默认集合中最后一个元素）。  
　　（6）step遍历的步长（大于0的整型）。  
　　（7）varStatus存放遍历到的成员的状态信息。、

<body>

<h4><c:out value="forToken实例"/></h4>

<hr/>

<%--提示：分隔符的作用是根据标识，截取字符串。

如果未设定分隔符或在字符串中没有找到分隔付，将把整个元素作为一个元素截取。

在实际应用中用于在除去某些符号在页面中显示。 --%>

<c:forTokens var="str" items="北、京、欢、迎、您" delims="、">

<c:out value="${str}"></c:out><br/>

</c:forTokens>

<br/>

<c:forTokens items="123-4567-8854" delims="-" var="t">

<c:out value="${t}"></c:out><br/>

</c:forTokens>

<br/>

<c:forTokens items="1\*2\*3\*4\*5\*6\*7" delims="\*" begin="1" end="3" var="n"

varStatus="s">

&nbsp;<c:out value="${n}" />的四种属性：<br>

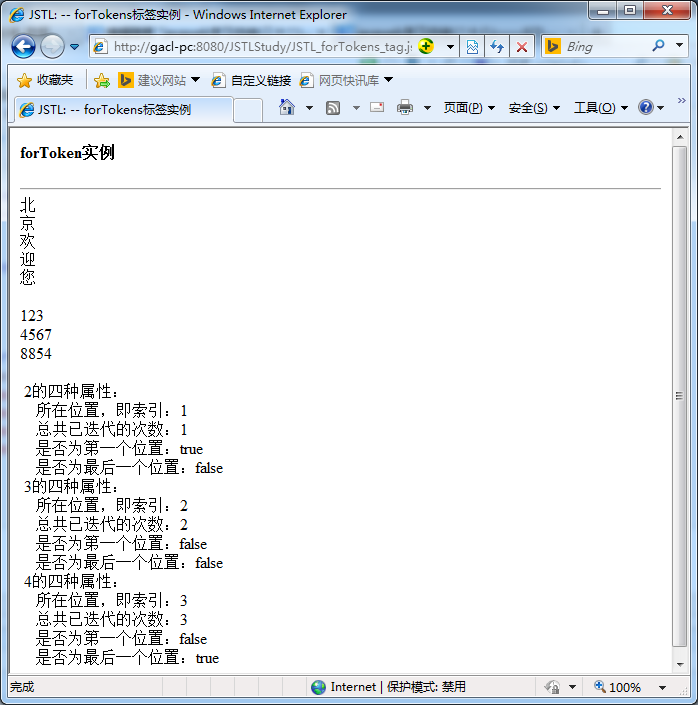
&nbsp;&nbsp;&nbsp;&nbsp;所在位置，即索引：<c:out value="${s.index}" /><br>

&nbsp;&nbsp;&nbsp;&nbsp;总共已迭代的次数：<c:out value="${s.count}" /><br>

&nbsp;&nbsp;&nbsp;&nbsp;是否为第一个位置：<c:out value="${s.first}" /><br>

&nbsp;&nbsp;&nbsp;&nbsp;是否为最后一个位置：<c:out value="${s.last}" /><br>

</c:forTokens>

 </body>

## import标签

该标签可以把其他静态或动态文件包含到本JSP页面，与<jsp:include>的区别为：<jsp:include>只能包含同一个web应用中的文件。而<c:import>可以包含其他web应用中的文件，甚至是网络上的资源。

【参数说明】：  
（1）URL为资源的路径，当引用的资源不存在时系统会抛出异常，因此该语句应该放在<c:catch></c:catch>语句块中捕获。  
（2）引用资源有两种方式：绝对路径和相对路径。  
    使用绝对路径的示例如下：<c:import url=”http://www.baidu.com”>  
    使用相对路径的示例如下：<c:import url=”aa.txt”>，aa.txt放在同一文件目录。  
（3）**如果以“/”开头表示应用的根目录下。**例如：tomcat应用程序的根目录文件夹为webapps。导入webapps下的文件bb.txt的编写方式为：<c:import url=”/bb.txt”>  
    如果访问webapps管理文件夹中其他web应用就要用context属性。  
（4）context属性用于在访问其他web应用的文件时，指定根目录。例如，访问root下的index.jsp的实现代码为：<c:import url=”/index.jsp” context=”/root”>  
    等同于webapps/root/index.jsp  
（5）var、scope、charEncoding、varReader是可选属性。

import标签的属性：

1. **url【指定要包含的路径，Internet所有的url都可以】**
2. **context【访问同一个web容器的其他资源，以"/"开头】**
3. **var【保存导入的文件的内容，以String类型存储】**
4. **socpe【保存的范围，默认是page】**
5. **charEncoding【字符编码】**
6. **varReader【保存导入文件的内容，以Reader类型存储】**

### <c:import>标签的使用范例

<%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>

<%--引入JSTL核心标签库 --%>

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

<!DOCTYPE HTML>

<html>

<head>

<title>JSTL: -- import标签实例</title>

</head>

<body>

<h4><c:out value="import实例"/></h4>

<hr/>

<h4><c:out value="绝对路径引用的实例" /></h4>

<%--使用绝对路径导入百度首页，

导入时使用<c:catch></c:catch>捕获异常。

--%>

<c:catch var="error1">

<c:import url="http://wwww.baidu.com" charEncoding="utf-8"/>

</c:catch>

${error1}

<hr/>

<h4>

<c:out value="相对路径引用本应用中的文件" />

</h4>

<%--使用相对路径导入同一文件夹下的“JSTL的import标签使用说明”文件，

接收的字符编码格式使用charEncoding设置为utf-8。 --%>

<c:catch var="error2">

<c:import url="JSTL的import标签使用说明" charEncoding="utf-8"/>

</c:catch>

${error2}

<hr/>

<h4><c:out value="使用字符串输出相对路径引用的实例，并保存在session范围内" /></h4>

<%--导入“JSTL的import标签使用说明.txt”，

使用var定义的变量接收要导入的文件，并存储在session中，

如果在其他页面同样也要导入该文件，只须使用<c:out>输出“JSTL的import标签使用说明.txt”的值即可。

--%>

<c:catch var="error3">

<c:import

var="myurl"

url="JSTL的import标签使用说明"

scope="session"

charEncoding="utf-8"/>

<c:out value="${myurl}"></c:out>

<hr/>

<c:out value="${myurl}" />

</c:catch>

${error3}

</body>

</html>

## c:param标签

在JSP页面进行URL的相关操作时，经常要在URL地址后面附加一些参数。<c:param>标签可以嵌套在<c:import>、<c:url>或<c:redirect>标签内，为这些标签所使用的URL地址附加参数。  
　　<c:param>标签在为一个URL地址附加参数时，将自动对参数值进行URL编码，例如，如果传递的参数值为“中国”， 则将其转换为“%d6%d0%b9%fa”后再附加到URL地址后面，这也就是使用<c:param>标签的最大好处。

　　示例1：与<c:url>标签嵌套使用

<c:url value="http://www.baidu.com" var="paramUrl">

<c:param name="userName" value="孤傲苍狼"/>

<c:param name="pwd">123456</c:param>

</c:url>

<a href="${paramUrl}">百度首页(带参数)</a>

　　示例2：与<c:redirect>标签嵌套使用

<c:redirect url="http://www.baidu.com">

<%--在重定向时使用<c:param>标签为URL添加了两个参数：uname=GACL和password=123 --%>

<c:param name="uname">GACL</c:param>

<c:param name="password">123</c:param>

</c:redirect>

 　　关于JSTL核心标签库中的标签掌握以上的那些标签基本上就可以应付开发了。

## url标签

<c:url>标签用于在JSP页面中构造一个URL地址，其主要目的是实现URL重写。

url标签十分实用！**在浏览器禁用Cookie的时候**，我们之前学Servlet时解决办法是：response.encodeURL()。**url标签也可以实现这样的功能，再配合param标签使用，就十分实用了！**

### <c:url>标签使用范例

<body>

<c:out value="url标签使用"></c:out>

<h4>使用url标签生成一个动态的url，并把值存入session中.</h4>

<hr/>

<c:url value="http://www.baidu.com" var="url" scope="session">

</c:url>

<a href="${url}">百度首页(不带参数)</a>

<hr/>

<h4>

配合 &lt;c:param&gt;标签给url加上指定参数及参数值，生成一个动态的url然后存储到paramUrl变量中

</h4>

<c:url value="http://www.baidu.com" var="paramUrl">

<c:param name="userName" value="孤傲苍狼"/>

<c:param name="pwd">123456</c:param>

</c:url>

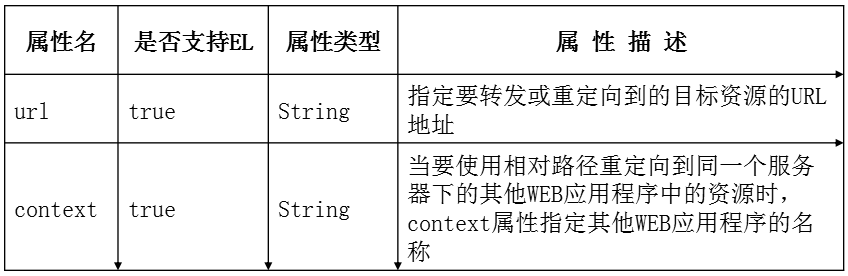
<a href="${paramUrl}">百度首页(带参数)</a>

</body>

</html>

## redirect标签

该标签用来实现请求的重定向。同时可以配合使用<c:param>标签在url中加入指定的参数。

【参数说明】：  
（1）url指定重定向页面的地址，可以是一个string类型的绝对地址或相对地址。  
（2）context用于导入其他web应用中的页面。

<body>

<c:redirect url="http://www.baidu.com">

<%--在重定向时使用<c:param>标签为URL添加了两个参数：uname=GACL和password=123 --%>

<c:param name="uname">GACL</c:param>

<c:param name="password">123</c:param>

</c:redirect>

</body>

</html>

# EL表达式

## EL表达式

**EL表达式是用"${}"括起来的脚本，用来更方便的读取对象！**

**EL的隐式对象包括：pageContext、initParam（访问上下文参数）、param（访问请求参数）、paramValues、header（访问请求头）、headerValues、cookie（访问cookie）、四大域**

**initParam用来读取设置在web.xml中的参数值。例如${initParam.repeat}，等价于：(String)application.getInitParameter(“repeat”);** **headerValues则用来取得所有的头信息，等价于调用request.getHeaders()方法。**

**EL主要作用：**  
**1、获取数据**  
　　　　EL表达式主要用于替换JSP页面中的脚本表达式，以从各种类型的web域 中检索java对象、获取数据。(某个web域 中的对象，访问javabean的属性、访问list集合、访问map集合、访问数组)  
**2、执行运算**  
　　　　利用EL表达式可以在JSP页面中执行一些基本的关系运算、逻辑运算和算术运算，以在JSP页面中完成一些简单的逻辑运算。${user==null}  
**3、获取web开发常用对象**  
　　　　EL 表达式定义了一些隐式对象，利用这些隐式对象，web开发人员可以很轻松获得对web常用对象的引用，从而获得这些对象中的数据。  
**4、调用Java方法**  
　　　　EL表达式允许用户开发自定义EL函数，以在JSP页面中通过EL表达式调用Java类的方法。

ServletContext对象的时候讲过一个方法**findAttribute(String name),EL表达式语句在执行的时候会调用该方法，用标识符作为关键字分别从page、request、session、application四个域中查找相应的对象。**这也解释了为什么EL表达式可以仅仅通过标识符就能够获取到存进域对象的数据！

EL表达式如果找不到相应的对象属性，返回的的空白字符串“”，而不是null，这是EL表达式最大的特点！

### 获取各类数据

<%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>

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

<%@page import="me.gacl.domain.Person"%>

<%@page import="me.gacl.domain.Address"%>

<!DOCTYPE HTML>

<html>

<head>

<title>el表达式获取数据</title>

</head>

<body>

<%

request.setAttribute("name","孤傲苍狼");

%>

<%--${name}等同于pageContext.findAttribute("name") --%>

使用EL表达式获取数据：${name}

<hr>

<!-- 在jsp页面中，使用el表达式可以获取bean的属性 -->

<%

Person p = new Person();

p.setAge(12);

request.setAttribute("person",p);

%>

使用el表达式可以获取bean的属性：${person.age}

<hr>

<!-- 在jsp页面中，使用el表达式可以获取bean中的。。。。。。。。。的属性 -->

<%

Person person = new Person();

Address address = new Address();

person.setAddress(address);

request.setAttribute("person",person);

%>

${person.address.name}

<hr>

<!-- 在jsp页面中，使用el表达式获取list集合中指定位置的数据 -->

<%

Person p1 = new Person();

p1.setName("孤傲苍狼");

Person p2 = new Person();

p2.setName("白虎神皇");

List<Person> list = new ArrayList<Person>();

list.add(p1);

list.add(p2);

request.setAttribute("list",list);

%>

<!-- 取list指定位置的数据 -->

${list[1].name}

<!-- 迭代List集合 -->

<c:forEach var="person" items="${list}">

${person.name}

</c:forEach>

<hr>

<!-- 在jsp页面中，使用el表达式获取map集合的数据 -->

<%

Map<String,String> map = new LinkedHashMap<String,String>();

map.put("a","aaaaxxx");

map.put("b","bbbb");

map.put("c","cccc");

map.put("1","aaaa1111");

request.setAttribute("map",map);

%>

<!-- 根据关键字取map集合的数据 -->

${map.c}

${map["1"]}//如果**Map集合存储的关键字是一个数字，就不能使用"."号运算符了**

<hr>

<!-- 迭代Map集合 -->

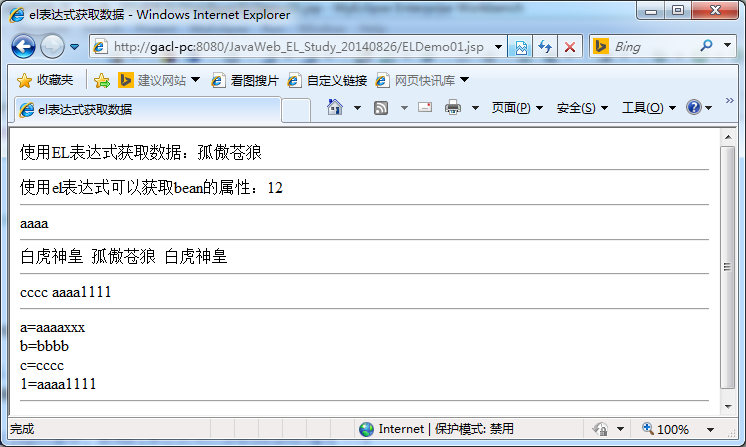
<c:forEach var="me" items="${map}">

${me.key}=${me.value}<br/>

</c:forEach>

<hr>

</body>

</html>

### EL运算符

<body>

<h3>el表达式进行四则运算：</h3>

加法运算：${365+24}<br/>

减法运算：${365-24}<br/>

乘法运算：${365\*24}<br/>

除法运算：${365/24}<br/>

<h3>el表达式进行关系运算：</h3>

<%--${user == null}和 ${user eq null}两种写法等价--%>

${user == null}<br/>

${user eq null}<br/>

<h3>el表达式使用empty运算符检查对象是否为null(空)</h3>

<%

List<String> list = new ArrayList<String>();

list.add("gacl");

list.add("xdp");

request.setAttribute("list",list);

%>

<%--使用empty运算符检查对象是否为null(空) --%>

<c:if test="${!empty(list)}">

<c:forEach var="str" items="${list}">

${str}<br/>

</c:forEach>

</c:if>

<br/>

<%

List<String> emptyList = null;

%>

<%--使用empty运算符检查对象是否为null(空) --%>

<c:if test="${empty(emptyList)}">

对不起，没有您想看的数据

</c:if>

<br/>

<h3>EL表达式中使用二元表达式</h3>

<%

session.setAttribute("user",new User("孤傲苍狼"));

%>

${user==null? "对不起，您没有登陆 " : user.username}

<br/>

<h3>EL表达式数据回显</h3>

<%

User user = new User();

user.setGender("male");

//数据回显

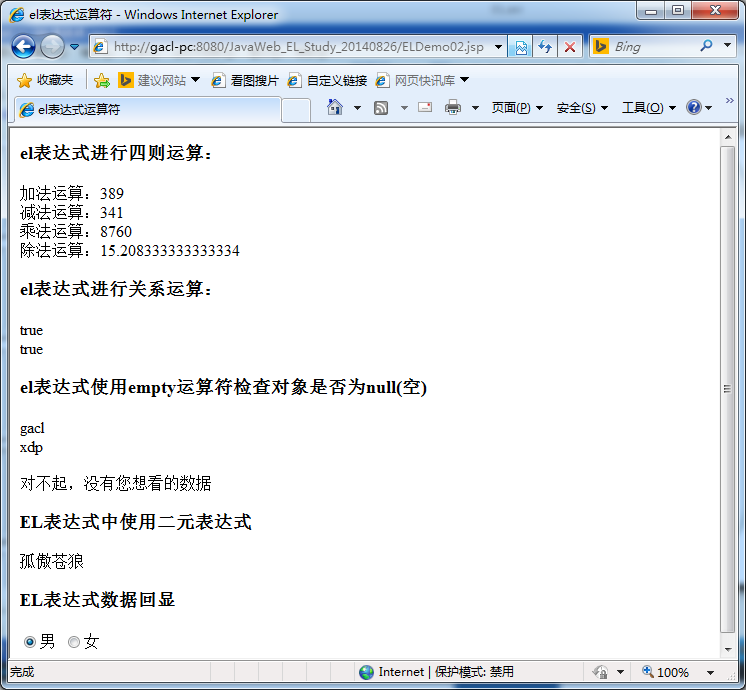
request.setAttribute("user",user);

%>

<input type="radio" name="gender" value="male" ${user.gender=='male'?'checked':''}>男

<input type="radio" name="gender" value="female" ${user.gender=='female'?'checked':''}>女

<br/>65 </body>

</html>

### 获得web开发常用对象

EL表达式主要是来对内容的显示，为了显示的方便，EL表达式提供了11个内置对象。

1. pageContext 对应于JSP页面中的pageContext对象（注意：取的是pageContext对象）
2. pageScope 代表page域中用于保存属性的Map对象
3. requestScope 代表request域中用于保存属性的Map对象
4. sessionScope 代表session域中用于保存属性的Map对象
5. applicationScope 代表application域中用于保存属性的Map对象
6. param 表示一个保存了所有请求参数的Map对象
7. paramValues表示一个保存了所有请求参数的Map对象，它对于某个请求参数，返回的是一个string[]
8. header 表示一个保存了所有http请求头字段的Map对象
9. headerValues同上，返回string[]数组。

10.cookie 表示一个保存了所有cookie的Map对象

11.initParam 表示一个保存了所有web应用初始化参数的map对象

 测试EL表达式中的11个隐式对象：

<body>

<br/>---------------1、pageContext对象：获取JSP页面中的pageContext对象-------------------<br/>

${pageContext}

<br/>---------------2、pageScope对象：从page域(pageScope)中查找数据------------------------<br/>

<%

pageContext.setAttribute("name","孤傲苍狼"); //map

%>

${pageScope.name}

<br/>---------------3、requestScope对象：从request域(requestScope)中获取数据-----------------<br/>

<%

request.setAttribute("name","白虎神皇"); //map

%>

${requestScope.name}

<br/>--------------4、sessionScope对象：从session域(sessionScope)中获取数据------------------<br/>

<%

session.setAttribute("user","xdp"); //map

%>

${sessionScope.user}

<br/>------------5、applicationScope对象：从application域(applicationScope)中获取数据-------<br/>

<%

application.setAttribute("user","gacl"); //map

%>

${applicationScope.user}

<br/>--------------6、param对象：获得用于保存请求参数map，并从map中获取数据----------<br/>

<!-- http://localhost:8080/JavaWeb\_EL\_Study\_20140826/ELDemo03.jsp?name=aaa -->

${param.name}

<!-- 此表达式会经常用在数据回显上 -->

<form action="${pageContext.request.contextPath}/servlet/RegisterServlet" method="post">

<input type="text" name="username" value="${param.username}">

<input type="submit" value="注册">

</form>

<br/>--------------7、paramValues对象：paramValues获得请求参数 //map{"",String[]}--------<br/>

<!-- http://localhost:8080/JavaWeb\_EL\_Study\_20140826/ELDemo03.jsp?like=aaa&like=bbb -->

${paramValues.like[0]}

${paramValues.like[1]}

<br/>--------------8、header对象：header获得请求头------------------------<br/>

${header.Accept}<br/>

<%--${header.Accept-Encoding} 这样写会报错

测试headerValues时，如果头里面有“-” ，例Accept-Encoding，则要headerValues[“Accept-Encoding”]

--%>

${header["Accept-Encoding"]}

<br/>--------------9、headerValues对象：headerValues获得请求头的值------------------------<br/>

<%--headerValues表示一个保存了所有http请求头字段的Map对象，它对于某个请求参数，返回的是一个string[]数组

例如：headerValues.Accept返回的是一个string[]数组 ，headerValues.Accept[0]取出数组中的第一个值

--%>

${headerValues.Accept[0]}<br/>

<%--${headerValues.Accept-Encoding} 这样写会报错

测试headerValues时，如果头里面有“-” ，例Accept-Encoding，则要headerValues[“Accept-Encoding”]

headerValues["Accept-Encoding"]返回的是一个string[]数组，headerValues["Accept-Encoding"][0]取出数组中的第一个值

--%>

${headerValues["Accept-Encoding"][0]}

<br/>--------------10、cookie对象：cookie对象获取客户机提交的cookie------------------------<br/>

<!-- 从cookie隐式对象中根据名称获取到的是cookie对象,要想获取值，还需要.value -->

${cookie.JSESSIONID.value} //保存所有cookie的map

<br/>-----11、initParam对象：initParam对象获取在web.xml文件中配置的初始化参数-------------<br/>

<%--

<!-- web.xml文件中配置初始化参数 -->

<context-param>

<param-name>xxx</param-name>

<param-value>yyyy</param-value>

</context-param>

<context-param>

<param-name>root</param-name>

<param-value>/JavaWeb\_EL\_Study\_20140826</param-value>

</context-param>

--%>

<%--获取servletContext中用于保存初始化参数的map --%>

${initParam.xxx}<br/>

${initParam.root}

</body>

</html>

注意:  
　　**测试header和headerValues时，如果头里面有“-” ，例Accept-Encoding，则要header["Accept-Encoding"]、headerValues["Accept-Encoding"]**  
**测试cookie时，例${cookie.key}取的是cookie对象，如访问cookie的名称和值，须${cookie.key.name}或${cookie.key.value}**

### 使用EL调用Java方法

　　EL表达式语法允许开发人员开发自定义函数，以调用Java类的方法。语法：${prefix：method(params)}  
　　在EL表达式中调用的只能是Java类的静态方法，这个Java类的静态方法需要在TLD文件中描述，才可以被EL表达式调用。  
　　EL自定义函数用于扩展EL表达式的功能，可以让EL表达式完成普通Java程序代码所能完成的功能。

## EL Function开发步骤

　　一般来说， EL自定义函数开发与应用包括以下三个步骤：  
　　1、编写一个Java类的静态方法  
　　2、编写标签库描述符（tld）文件，在tld文件中描述自定义函数。  
　　3、在JSP页面中导入和使用自定义函数

示例：开发对html标签进行转义的el function

编写完标签库描述文件后，需要将它放置到<web应用>\WEB-INF目录中或WEB-INF目录下的除了classes和lib目录之外的任意子目录中。   
　　TLD文件中的<uri> 元素用指定该TLD文件的URI，在JSP文件中需要通过这个URI来引入该标签库描述文件。  
　　<function>元素用于描述一个EL自定义函数，其中：  
　　<name>子元素用于指定EL自定义函数的名称。  
　　<function-class>子元素用于指定完整的Java类名，  
　　<function-signature>子元素用于指定Java类中的静态方法的签名，方法签名必须指明方法的返回值类型及各个参数的类型，各个参数之间用逗号分隔。

注意：有些Tomcat服务器如不能使用EL表达式  
    （1）升级成tomcat6  
    （2）在JSP中加入<%@ page isELIgnored="false" %>

## EL函数库

这些EL函数在JSTL开发包中进行描述，因此在JSP页面中使用SUN公司的EL函数库，需要导入JSTL开发包，并在页面中导入EL函数库，

在页面中使用JSTL定义的EL函数：[%@taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn"%](mailto:%25@taglib%20uri=%22http://java.sun.com/jsp/jstl/functions%22%20prefix=%22fn%22%25)

要对fn方法库做测试，首先**导入开发包（jstl.jar、standard.jar)**

fn方法库全都是跟字符串有关的（可以把它想成是String的方法）

* fn:toLowerCase
* fn:toUpperCase
* fn:trim
* fn:length
* fn:split
* fn:join 【接收字符数组，拼接字符串】
* fn:indexOf
* fn:contains
* fn:startsWith
* fn:replace
* fn:substring
* fn:substringAfter
* fn:endsWith
* fn:escapeXml【忽略XML标记字符】
* fn:substringBefore

### EL函数使用范例

<body>

<h3>fn:toLowerCase函数使用范例：</h3>

<%--fn:toLowerCase函数将一个字符串中包含的所有字符转换为小写形式，并返回转换后的字符串，

它接收一个字符串类型的参数。fn:toLowerCase("")的返回值为空字符串--%>

<%--fn:toLowerCase("Www.CNBLOGS.COM") 的返回值为字符串“www.cnblogs.com” --%>

fn:toLowerCase("Www.CNBLOGS.COM")的结果是：${fn:toLowerCase("Www.CNBLOGS.COM")}

<hr/>

<h3>fn:toUpperCase函数使用范例：</h3>

<%--fn:toUpperCase函数将一个字符串中包含的所有字符转换为大写形式，并返回转换后的字符串，

它接收一个字符串类型的参数。fn:toUpperCase("")的返回值为空字符串--%>

fn:toUpperCase("cnblogs.com")的结果是：${fn:toUpperCase("cnblogs.com")}

<hr/>

<h3>fn:trim函数使用范例：</h3>

<%--fn:trim函数删除一个字符串的首尾的空格，并返回删除空格后的结果字符串，

它接收一个字符串类型的参数。需要注意的是，fn:trim函数不能删除字符串中间位置的空格。--%>

fn:trim(" cnblogs.com ")的结果是：${fn:trim(" cnblogs.com ")}

<hr/>

<h3>fn:length函数使用范例：</h3>

<%--fn:length函数返回一个集合或数组大小，或返回一个字符串中包含的字符的个数，返回值为int类型。

fn:length函数接收一个参数，这个参数可以是<c:forEach>标签的items属性支持的任何类型，

包括任意类型的数组、java.util.Collection、java.util.Iterator、java.util.Enumeration、

java.util.Map等类的实例对象和字符串。

如果fn:length函数的参数为null或者是元素个数为0的集合或数组对象，则函数返回0；如果参数是空字符串，则函数返回0

--%>

<%

List<String> list = Arrays.asList("1","2","3");

request.setAttribute("list",list);

%>

fn:length(list)计算集合list的size的值是：${fn:length(list)}

<br/>

fn:length("cnblogs.com")计算字符串的长度是：${fn:length("cnblogs.com")}

<hr/>

<h3>fn:split函数使用范例：</h3>

<%--

fn:split函数以指定字符串作为分隔符，将一个字符串分割成字符串数组并返回这个字符串数组。

fn:split函数接收两个字符串类型的参数，第一个参数表示要分割的字符串，第二个参数表示作为分隔符的字符串

--%>

fn:split("cnblogs.com",".")[0]的结果是：${fn:split("cnblogs.com",".")[0]}

<hr/>

<h3>fn:join函数使用范例：</h3>

<%--

fn:join函数以一个字符串作为分隔符，将一个字符串数组中的所有元素合并为一个字符串并返回合并后的结果字符串。

fn:join函数接收两个参数，第一个参数是要操作的字符串数组，第二个参数是作为分隔符的字符串。

如果fn:join函数的第二个参数是空字符串，则fn:join函数的返回值直接将元素连接起来。

--%>

<%

String[] StringArray = {"www","iteye","com"};

pageContext.setAttribute("StringArray", StringArray);

%>

<%--fn:join(StringArray,".")返回字符串“www.iteye.com”--%>

fn:join(StringArray,".")的结果是：${fn:join(StringArray,".")}

<br/>

<%--fn:join(fn:split("www,iteye,com",","),".")的返回值为字符串“www.iteye.com”--%>

fn:join(fn:split("www,iteye,com",","),".")的结果是：${fn:join(fn:split("www,iteye,com",","),".")}

<hr/>

<h3>fn:indexOf函数使用范例：</h3>

<%--

fn:indexOf函数返回指定字符串在一个字符串中第一次出现的索引值，返回值为int类型。

fn:indexOf函数接收两个字符串类型的参数，如果第一个参数字符串中包含第二个参数字符串，

那么，不管第二个参数字符串在第一个参数字符串中出现几次，fn:indexOf函数总是返回第一次出现的索引值；

如果第一个参数中不包含第二个参数，则fn:indexOf函数返回-1。如果第二个参数为空字符串，则fn:indexOf函数总是返回0。

--%>

fn:indexOf("www.iteye.com","eye")的返回值为：${fn:indexOf("www.iteye.com","eye")}

<hr/>

<h3>fn:contains函数使用范例：</h3>

<%--

fn:contains函数检测一个字符串中是否包含指定的字符串，返回值为布尔类型。

fn:contains函数在比较两个字符串是否相等时是大小写敏感的。

fn:contains函数接收两个字符串类型的参数，如果第一个参数字符串中包含第二个参数字符串，则fn:contains函数返回true，否则返回false。

如果第二个参数的值为空字符串，则fn:contains函数总是返回true。

实际上，fn:contains(string, substring)等价于fn:indexOf(string, substring) != -1

忽略大小的EL函数：fn:containsIgnoreCase

--%>

<%

User user = new User();

String likes[] = {"sing","dance"};

user.setLikes(likes);

//数据回显

request.setAttribute("user",user);

%>

<%--使用el函数回显数据 --%>

<input type="checkbox" name="like"

vlaue="sing" ${fn:contains(fn:join(user.likes,","),"sing")?'checked':''}/>唱歌

<input type="checkbox" name="like"

value="dance" ${fn:contains(fn:join(user.likes,","),"dance")?'checked':''}/>跳舞

<input type="checkbox" name="like"

value="basketball" ${fn:contains(fn:join(user.likes,","),"basketball")?'checked':''}/>蓝球

<input type="checkbox" name="like"

value="football" ${fn:contains(fn:join(user.likes,","),"football")?'checked':''}/>足球

<hr/>

<h3>fn:startsWith函数和fn:endsWith函数使用范例：</h3>

<%--

fn:startsWith函数用于检测一个字符串是否是以指定字符串开始的，返回值为布尔类型。

fn:startsWith函数接收两个字符串类型的参数，如果第一个参数字符串以第二个参数字符串开始，则函数返回true，否则函数返回false。

如果第二个参数为空字符串，则fn:startsWith函数总是返回true。

与fn:startsWith函数对应的另一个EL函数为：fn:endsWith，用于检测一个字符串是否是以指定字符串结束的，返回值为布尔类型。

--%>

fn:startsWith("www.iteye.com","iteye")的返回值为：${fn:startsWith("www.iteye.com","iteye")}

<br/>

fn:endsWith("www.iteye.com","com")的返回值为：${fn:endsWith("www.iteye.com","com")}

<hr/>

<h3>fn:replace使用范例：</h3>

<%--

fn:replace函数将一个字符串中包含的指定子字符串替换为其它的指定字符串，并返回替换后的结果字符串。

fn:replace方法接收三个字符串类型的参数，第一个参数表示要操作的源字符串，第二个参数表示源字符串中要被替换的子字符串，

第三个参数表示要被替换成的字符串。

--%>

fn:replace("www iteye com ", " ", ".")的返回值为字符串：${fn:replace("www iteye com", " ", ".")}

<hr/>

<h3>fn:substring使用范例：</h3>

<%--

fn:substring函数用于截取一个字符串的子字符串并返回截取到的子字符串。

fn:substring函数接收三个参数，第一个参数是用于指定要操作的源字符串，第二个参数是用于指定截取子字符串开始的索引值，

第三个参数是用于指定截取子字符串结束的索引值，第二个参数和第三个参数都是int类型，其值都从0开始。

--%>

fn:substring("www.it315.org", 4, 9) 的返回值为字符串：${fn:substring("www.it315.org", 4, 9)}

<h3>fn:substringAfter函数和fn:substringBefore函数使用范例：</h3>

<%--

fn:substringAfter函数用于截取并返回一个字符串中的指定子字符串第一次出现之后的子字符串。

fn:substringAfter函数接收两个字符串类型的参数，第一个参数表示要操作的源字符串，第二个参数表示指定的子字符串

与之对应的EL函数为：fn:substringBefore

--%>

fn:substringAfter("www.it315.org",".")的返回值为字符串：${fn:substringAfter("www.it315.org",".")}

<br/>

fn:substringBefore("www.it315.org",".")的返回值为字符串：${fn:substringBefore("www.it315.org",".")}

<hr/>

</body>

</html>

## [国际化(i18n)](https://www.cnblogs.com/xdp-gacl/p/3945800.html)

https://www.cnblogs.com/xdp-gacl/p/3945800.html