

第6章

JSP标签技术



本章内容

- 6.1 自定义标签的开发
- 6.2 理解TLD文件
- 6.3 几种类型标签的开发
- 6.4 **JSP**标准标签库

本章内容

- 从JSP1.1版开始就可以在JSP页面中使用标签了，使用标签不但可以实现代码重用，而且可以使JSP代码更简洁。
- 为了进一步简化JSP的开发，在JSP 2.0的标签扩展API中又增加了SimpleTag接口和其实现类SimpleTagSupport，使用它们可以开发简单标签。

-

6.1 自定义标签的开发

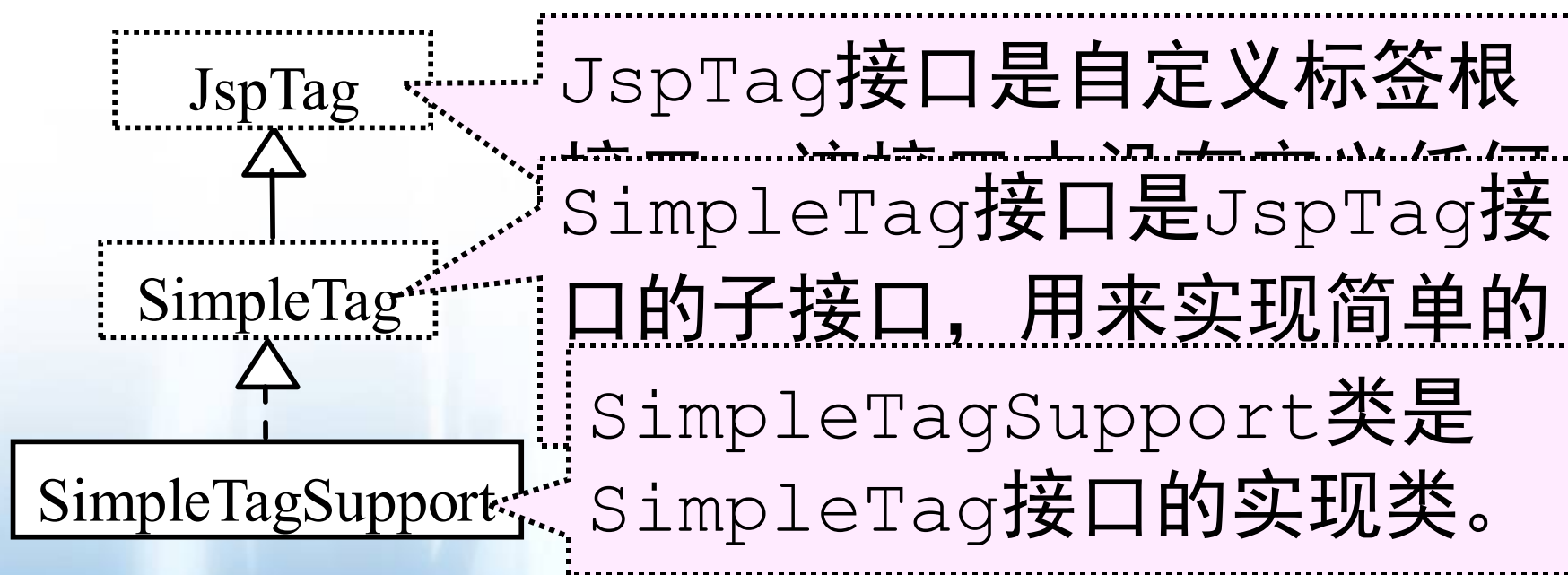
- 6.1.1 标签扩展API
- 6.1.2 自定义标签的开发步骤
- 6.1.3 SimpleTag接口及其生命周期
- 6.1.4 SimpleTagSupport类

6.1 自定义标签的开发

- 所谓自定义的标签是用Java语言开发的程序，当其在JSP页面中使用时将执行某种动作，所以有时自定义标签又叫作自定义动作（custom action）。
- 在JSP页面中可以使用两类自定义标签。一类是简单的（simple）自定义标签，一类是传统的（classic）自定义标签。传统的自定义标签是JSP 1.1中提供的，简单的自定义标签是JSP 2.0增加的。

6.1.1 标签扩展API

- 要开发自定义标签，需要使用 `javax.servlet.jsp.tagext` 包中的接口和类，这些接口和类称为标签扩展API。图6-1给出了简单标签扩展API的层次结构。



6.1.1 标签扩展API

- 除上面的接口和类外，标签处理类还要使用到`javax.servlet.jsp`包中定义的两个异常类：`JspException`和`JspTagException`类。

6.1.2 自定义标签的开发步骤

- 创建自定义标签需要创建下面两步：
 - 创建标签处理类。
 - 创建标签库描述文件TLD。

1. 创建标签处理类

- **标签处理类**（tag handler）是实现某个标签接口或继承某个标签类的实现类，程序6.1给出了一个标签处理类，它实现了SimpleTag接口，该标签的功能是向JSP页面输出一条消息。
- [程序6.1 HelloTag.java](#)

2. 创建标签库描述文件

- 标签库描述文件 (Tag Library Descriptor, TLD) 用来定义使用标签的URI和对标签的描述，它是XML格式的文件，扩展名一般为`.tld`。下面的TLD文件定义了一个名为`hello`的标签。
- [程序6.2 mytaglib.tld](#)
- TLD文件一般存放在Web应用程序的WEB-INF目录或其子目录下。

3. 在JSP页面中使用标签

- 在JSP页面使用自定义标签，需要通过taglib指令声明自定义标签的前缀和标签库的URI，格式如下：

```
<%@ taglib prefix="prefixName" uri="tag library uri" %>
```

- prefix属性值为标签的前缀，uri属性值为标签库的URI。在JSP的taglib指令中，前缀名称不能使用JSP的保留前缀名，它们包括jsp、jspx、java、javax、servlet、sun、sunw。
- [程序6.3 helloTag.jsp](#)

6.1.3 SimpleTag接口及其生命周期

- SimpleTag接口中定义了简单标签的生命周期方法。
- SimpleTag接口中的方法有两个目的。第一，它允许在Java类和JSP之间传输信息。第二，它是由Web容器调用来初始化SimpleTag操作。该接口共定义了**5**个方法：

1. SimpleTag接口的方法

- **public void setJspContext(JspContext pc):** 该方法由容器调用，用来设置JspContext对象，使其在标签处理类中可用。
- **public void setParent(JspTag parent):** 该方法由容器调用，用来设置父标签对象。
- **public void setJspBody(JspFragment jspBody):** 若标签带标签体，容器调用该方法将标签体内容存放到JspFragment中。
- **public JspTag getParent():** 返回当前标签的父标签。
- **public void doTag() throws JspException, IOException:** 该方法是简单标签的核心方法，由容器调用完成简单标签的操作。

2. 简单标签的生命周期

- 当容器在JSP页面中遇到自定义标签时，它将加载标签处理类并创建一个实例，然后调用标签类的生命周期方法。标签的生命周期有下面几个主要阶段：
 - 1) 调用setJspContext()
- 容器为该方法传递一个JspContext类的实例，该实例称为JSP上下文对象。可将该对象保存到一个实例变量中以备以后使用。
- javax.servlet.jsp.JspContext类定义了允许标签处理类访问JSP页面作用域中属性的方法，如setAttribute()、getAttribute()、removeAttribute()和findAttribute()等。该类还提供了getOut()，它返回JspWriter对象，用来向JSP输出信息。

2. 简单标签的生命周期

2) 调用setParent()

- 标签可以相互嵌套。在相互嵌套的标签中，外层标签称为父标签（parent tag），内层标签称为子标签（child tag）。如果标签是嵌套的，容器调用setParent()设置标签的父标签对象。因为setParent()返回一个JspTag对象，所以返回的父标签可以是实现SimpleTag、Tag、IterationTag或BodyTag等接口的对象。

2. 简单标签的生命周期

3) 调用属性的修改方法

- 如果自定义标签带属性，那么容器在运行时将调用属性修改方法设置属性值。由于方法格式依赖于属性名和类型，这些方法在标签处理类中定义。
- 例如，假设标签处理类提供了下面属性：

```
private boolean condition = false;
```

- 则应该提供下面的属性修改方法：

```
public void setCondition(boolean condition){  
    this.condition = condition;  
}
```


2. 简单标签的生命周期

4) 调用setJspBody()

- 如果标签包含标签体内容，容器将调用 **setJspBody(JspFragment jspBody)** 方法设置标签体。它将标签体中的内容存放到 JspFragment 对象中，以后调用该对象的 **invoke()** 输出标签体。在本章的后面将详细讨论 JspFragment 类。

2. 简单标签的生命周期

5) 调用doTag()

- 该方法是简单标签的核心方法，在doTag()中完成标签的功能。该方法不返回任何值，当它返回时，容器返回到前面的处理任务中。不需要调用特殊的方法，使用常规的Java代码，就可以控制所有迭代和标签体的内容。

6.1.4 SimpleTagSupport类

- SimpleTagSupport类是SimpleTag接口的实现类，它除实现了SimpleTag接口中的方法外，还提供了另外三个方法。
- **protected JspContext getJspContext()**: 返回标签中要处理的JspContext对象。
- **protected JspFragment getJspBody()**: 返回JspFragment对象，它存放了标签体的内容。
- **public static final JspTag findAncestorWithClass(JspTag from, Class klass)**: 根据给定的实例和类型查找最接近的实例。该方法主要用在开发协作标签中。

6.1.4 SimpleTagSupport类

- 编写简单标签处理类通常不必实现SimpleTag接口，而是继承SimpleTagSupport类，并且仅需覆盖该类的doTag()。修改HelloTag.java代码使其继承SimpleTagSupport可实现与程序6.1相同的功能。

```
public class HelloTag extends SimpleTagSupport{  
    public void doTag() throws JspException, IOException{  
        JspWriter out = getJspContext().getOut();  
        out.print("<font color='blue'>Hello, A simple  
tag.</font><br>");  
        out.print("现在时间是: "+ new java.util.Date());  
    }  
}
```

6.2 理解TLD文件

- 6.2.1 <taglib>元素
- 6.2.2 <uri>元素
- 6.2.3 <tag>元素
- 6.2.4 <attribute>元素
- 6.2.5 <body-content>元素

6.2 理解TLD文件

- 自定义标签需要在**TLD**文件中声明。当在**JSP**页面中使用自定义标签时，容器将读取**TLD**文件，从中获取有关自定义标签的信息，如标签名、标签处理类名、是否是空标签以及是否有属性等。
- **TLD**文件的第一行是声明，它的根元素是**<taglib>**，该元素定义了一些子元素。下面详细说明这些元素的使用。

6.2.1 <taglib>元素

- <taglib>元素是TLD文件的根元素，该元素带若干属性，它们指定标签库的命名空间、版本等信息等。下面是<taglib>元素的DTD定义：

```
<!ELEMENT taglib (description*, display-  
name*, icon*, tlib-version,  
short-name*,uri?, validator?, listener*,  
tag*,function*) >
```

- 只有<tlib-version>和<short-name>元素是必须的，其他元素都是可选的。

6.2.2 <uri>元素

- <uri>元素指定在JSP页面中使用taglib指令时uri属性的值。例如，若该元素的定义如下：

```
<uri>http://www.mydomain.com/sample</uri>
```

- 则在JSP页面中taglib指令应该如下所示：

```
<%@ taglib prefix="demo"  
uri="http://www.mydomain.com/sample" %>
```

- 这里的< uri>元素值看上去像一个Web资源的URI，但实际上它仅仅是一个逻辑名称，并不与任何Web资源对应，容器使用它仅完成URI与TLD文件的映射。

6.2.2 <uri>元素

- Web应用中可以使用三种类型的URI:
- 绝对URI。例如，
<http://www.mydomain.com/sample>和
<http://localhost:8080/taglibs>都是绝对URI。
- 根相对URI。以“/”开头且不带协议、主机名或端口号的URI。它被解释为相对于Web应用程序文档根目录。[/mytaglib](#)和[/taglib1/helloLib](#)是根相对URI。
- 非根相对URI。不以“/”开头也不带协议、主机名或端口号的URI。它被解释为相对于当前JSP页面或相对于WEB-INF目录，这要看它在哪使用的。[HelloLib](#) 和[taglib2/helloLib](#)是非根相对URI。

6.2.2 <uri>元素

- 在TLD文件中也可以不指定<uri>元素，这时容器会尝试将taglib指令中的uri属性看作TLD文件的实际路径（以“/”开头）。例如，对HelloTag标签，如果没有在TLD文件中指定<uri>元素，在JSP页面中可以像下面这样访问标签库：

```
<%@ taglib prefix="demo" uri="/WEB-INF/mytaglib.tld" %>
```

1. 容器如何查找TLD文件

- 容器是如何找到正确的TLD文件呢？实际上，在部署一个Web应用时，容器会自动建立一个URI与TLD之间的映射。
- 只要把TLD文件放在容器会查找的位置上，容器就会找到这个TLD，并为标签库建立一个映射。
- 容器自动查找TLD文件的位置包括：
 - 在/WEB-INF目录或其子目录中查找。
 - 在/WEB-INF/lib目录下的JAR文件中的META-INF目录或其子目录中查找。

2. 在DD文件中定义URI

- 在JSP 2.0之前，开发人员必须在DD文件（web.xml）中为URI指定其TLD文件的具体位置。然后，容器会查找web.xml文件的<taglib>元素，建立URI与TLD之间的映射。
- 例如，对于上述标签库，可以将下面代码加到web.xml文件的<web-app>元素中。

```
<jsp-config>
```

```
  <taglib>
```

```
    <taglib-uri>http://www.mydomain.com/sample
```

```
    </taglib-uri>
```

```
    <taglib-location>/WEB-INF/mytaglib.tld
```

```
    </taglib-location>
```

```
  </taglib>
```

```
</jsp-config>
```

6.2.3 <tag>元素

- <taglib>元素可以包含一个或多个<tag>元素，每个<tag>元素都提供了关于标签的信息，如在JSP页面中使用的标签名、标签处理类及标签的属性等。<tag>元素的DTD定义如下：

```
<!ELEMENT tag (description* , display-name* , icon* , name, tag-class, tei-class?, body-content?, variable*, attribute*, example?) >
```

6.2.3 <tag>元素

- 在一个TLD中不能定义多个同名的标签，因为容器不能解析标签处理类。因此，下面代码是非法的。

```
<tag>
```

```
  <name>hello</name>
```

```
  <tag-class>com.mytag.HelloTag</tag-class >
```

```
</tag>
```

```
<tag>
```

```
  <name>hello</name>
```

```
  <tag-class>com.mytag.WelcomeTag</tag-  
  class>
```

```
</tag>
```

6.2.3 <tag>元素

- 但是，可以使用一个标签处理类定义多个名称不同的标签。例如：

```
<tag>
```

```
  <name>hello</name>
```

```
  <tag-class>com.mytag.HelloTag</tag-class>
```

```
</tag>
```

```
<tag>
```

```
  <name>welcome</name>
```

```
  <tag-class>com.mytag.HelloTag</tag-class>
```

```
</tag>
```

- 在JSP页面中，假设使用demo作为前缀，则
<demo:hello>和**<demo:welcome>**两个标签都将调用com.mytag.HelloTag类。

6.2.4 <attribute>元素

- 如果自定义标签带属性，则每个属性的信息应该在<attribute>元素中指定。下面是<attribute>元素的DTD定义：

<!ELEMENT attribute (description*,name,required?,rtexprvalue?,type?) >

- 在<attribute>元素中，只有<name>元素是必须的且只能出现一次。所有其他元素都是可选的并最多只能出现一次。

6.2.5 <body-content>元素

- <tag>的子元素<body-content>指定标签体的内容类型，在简单标签中它的值是下面三者之一：
 - empty（默认值）
 - scriptless
 - tagdependent。

1. empty

- `<body-content>`元素值指定为empty，表示标签不带标签体。下面的例子声明了`<hello>`标签并指定标签体为空。

`<tag>`

`<name>hello</name>`

`<tag-class>com.mytag.HelloTag</tag-class>`

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

`</tag>`

1. empty

- 对空标签，如果使用时页面作者指定了标签体，容器在转换时产生错误。下面对该标签的使用是不合法的。

<demo:hello>john</demo:hello>

<demo:hello><%= "john" %></demo:hello>

<demo:hello> </demo:hello >

<demo:hello>

</demo:hello>

2. scriptless

- `<body-content>`元素值指定为**scriptless**，表示标签体中不能包含JSP脚本元素（JSP声明`<%! >`、表达式`<%=>`和小脚本`<% >`），但可以包含普通模板文本、HTML、EL表达式、标准动作、甚至在该标签中嵌套其他自定义标签。下面的例子声明了`<if>`标签，并指定标签体中不能使用脚本。

`<tag>`

`<name>if</name>`

`<tag-class>com.mytag.IfTag</tag-class>`

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

`</tag>`

2. scriptless

- 因此，下面对<if>标签的使用是合法的：

```
<demo:if condition="true">
```

```
  <demo:hello user="john" />
```

```
    2+3 = ${2+3}
```

```
</demo:if>
```

3. tagdependent

- `<body-content>` 元素值指定为 `tagdependent`，表示容器不会执行标签体，而是在请求时把它传递给标签处理类，由标签处理类根据需要决定处理标签体。

`<demo:query>`

`SELECT * FROM customers`

`</demo:query>`

3. tagdependent

- 对该标签，<body-content>元素值必须指定为tagdependent。

<tag>

<name>query</name>

<tag-class>com.mytag.QueryTag</tag-class>

<body-content>**tagdependent**<body-content>

</tag>

6.3 几种类型标签的开发

- 6.3.1 空标签的开发
- 6.3.2 带属性标签的开发
- 6.3.3 带标签体的标签
- 6.3.4 迭代标签
- 6.3.5 在标签中使用**EL**
- 6.3.6 使用动态属性
- 6.3.7 编写协作标签

6.3.1 空标签的开发

- 空标签是不含标签体的标签，它主要向JSP发送静态信息。下面是一个标签处理类的实现，它是一个空标签。当它在页面中使用时打印一个红色的星号（*）字符。
- [程序6.4 RedStarTag.java](#)

6.3.1 空标签的开发

- 下面在**TLD**文件中通过**<tag>**元素描述该标签的定义。

<tag>

<name>star</name>

<tag-class>com.mytag.RedStarTag</tag-class>

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

</tag>

6.3.1 空标签的开发

- 在**JSP**页面中访问空标签有两种写法，一种是由一对开始标签和结束标签组成，中间不含任何内容，例如：
`<prefix:tagName></prefix:tagName>`
- 另一种写法是简化的格式，即在开始标签末尾使用一个斜线（/）表示标签结束，例如：
`<prefix:tagName />`
- [程序6.5 register.jsp](#)

6.3.2 带属性标签的开发

- 自定义标签可以具有属性，属性可以是必选的，也可以是可选的。
- 对必选的属性，如果没有指定值，容器在 **JSP** 页面转换时将给出错误。
- 对可选的属性，如果没有指定值，标签处理类将使用默认值。默认值依赖于标签处理类的实现。

6.3.2 带属性标签的开发

- 在JSP页面中使用带属性的自定义标签的格式如下。

```
<prefix:tagName attrib1="fixedValue"  
attrib2="${elVariable}"  
attrib3="<%= someJSPExpression %>"  
>
```

- 属性值可以是常量或EL表达式，也可以是JSP表达式。表达式是在请求时计算的，并传递给相应的标签处理类。

6.3.2 带属性标签的开发

- 当标签接受属性时，对每个属性需要做三件重要的事情。
 - 必须在标签处理类中声明一个实例变量存放属性的值。
 - 如果属性不是必须的，则必须要么提供一个默认值，要么在代码中处理相应的null实例变量。
 - 对每个属性，必须实现适当的修改方法。
- 下面开发一个名为welcome的标签，它接受一个名为user的属性，它在输出中打印欢迎词。
- [程序6.6 WelcomeTag.java](#)

6.3.2 带属性标签的开发

- 下面的<tag>元素是在TLD文件中对该标签的描述。

<tag>

<name>**welcome**</name>

<tag-class>com.mytag.WelcomeTag</tag-class>

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

<attribute>

<name>**user**</name>

<required>**false**</required>

<rtexprvalue>**true**</rtexprvalue>

</attribute>

</tag>

6.3.2 带属性标签的开发

- 对上述定义的<welcome>标签，若使用demo前缀，则下面的使用是合法的。

```
<demo:welcome />
```

```
<demo:welcome></demo:welcome>
```

```
<demo:welcome user="john" />
```

```
<demo:welcome user='<%=  
    request.getParameter("userName") %>' />
```

```
<demo:welcome  
    user="${param.userName}"></demo:hello>
```


6.3.2 带属性标签的开发

- 属性值的指定也可以使用JSP的标准动作 `<jsp:attribute>`，通过该标签的 `name` 属性指定属性名，属性值在标签体中指定。

```
<demo:welcome>
```

```
    <jsp:attribute  
    name="user">${param.userName}</jsp:att  
    ribute>
```

```
</demo:welcome>
```

- [程序6.7 welcome.jsp](#)

6.3.3 带标签体的标签

- 在起始标签和结束标签之间包含的内容称为**标签体（body content）**。
- 对于SimpleTag标签，标签体可以是文本、HTML、EL表达式等，但不能包含JSP脚本（如声明、表达式和小脚本）。
- 如果需要访问标签体，应该调用简单标签类的**getJspBody()**，它返回一个抽象类**JspFragment**对象。

6.3.3 带标签体的标签

- **JspFragment**类只定义了两个方法。
- **public JspContext getJspContext():** 返回与JspFragment有关的JspContext对象。
- **public void invoke(Writer out) :** 执行标签体中的代码并将结果发送到Writer对象。如果将结果输出到JSP页面，参数应该为null。
- [程序6.8 BodyTagDemo.java](#)

6.3.3 带标签体的标签

- 由于简单标签的标签体中不能包含脚本元素，所以在TLD中应将<body-content>的值指定为scriptless或tagdependent，如下所示。

<tag>

<name>dobody</name>

<tag-class>com.mytag.BodyTagDemo</tag-class>

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

</tag>

- [程序6.9 dobody.jsp](#)

6.3.3 带标签体的标签

- 如果希望多次执行标签体，可以在doTag()中使用循环结构，多次调用JspFragment的invoke(null)即可。修改SimpleTagExample类的doTag()中的代码。

```
for(int i = 0 ; i<5 ; i++){  
    getJspBody().invoke(null);  
}
```

6.3.3 带标签体的标签

- 如果需要对标签体进行处理，可以将标签体内容保存到**StringWriter**对象中，然后将修改后的输出流对象发送到**JspWrier**对象。
- 下面的**marker**标签从标签体中查找指定的字符串，然后将其使用蓝色大字输出。
- [程序6.10 MarkerTag.java](#)

6.3.3 带标签体的标签

- 在TLD文件中使用下面代码定义该标签。

`<tag>`

`<name>marker</name>`

`<tag-class>com.mytag.MarkerTag</tag-class>`

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

`<attribute>`

`<name>search</name>`

`<required>true</required>`

`</attribute>`

`</tag>`

- 下面的JSP页面使用了marker标签。
- [程序6.11 marker.jsp](#)

6.3.4 迭代标签

- 所谓**迭代标签**就是能够多次访问标签体的标签，它实现了类似于编程语言的循环的功能。
- 下面的迭代标签通过一个名为**count**的属性指定对标签体的迭代次数。
- [程序6.12 LoopTag.java](#)

6.3.4 迭代标签

- 下面的<tag>元素在TLD文件中描述了该循环标签。

<tag>

<name>loop</name>

<tag-class>com.mytag.LoopTag</tag-class>

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

<attribute>

<name>count</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

6.3.4 迭代标签

- 下面是使用loop标签的JSP页面。
- [程序6.13 loop.jsp](#)



6.3.5 在标签中使用EL

- 在标签体中还可以使用EL表达式，例如：

<demo:dobody>

商品名称为:**`${product}`**。

</demo:dobody>

6.3.5 在标签中使用EL

- 那么在标签处理类中的doTag()应该如下。

```
public void doTag() throws  
    JspException, IOException{  
    getJspContext().setAttribute("product", "  
    苹果iPhone 5手机");  
    getJspBody().invoke(null);  
}
```

6.3.5 在标签中使用EL

- 标签体中的EL表达式可以是一个集合（数组、List或Map）对象，在标签体中可以访问它的每个元素，这只需要在doTag()中使用循环即可，例如：

```
<table border='1'>  
  <demo:dobody>  
    <tr><td>${product}</td></tr>  
  </demo:dobody>  
</table>
```

6.3.5 在标签中使用EL

- 在标签处理类的doTag()中的代码如下。

```
public void doTag() throws  
    JspException, IOException{  
    String products[]={  
        "苹果iPhone 5手机","OLYMPUS数码相机",  
        "文曲星电子词典"};  
    for( int i = 0; i<products.length; i++){  
        getJspContext().setAttribute("product",  
            products[i]);  
        getJspBody().invoke(null);  
    }  
}
```

6.3.5 在标签中使用EL

- 在自定义标签的属性值中还可以使用**EL**表达式。
- 下面示例首先在**ProductServlet**中连接数据库查询**products**表中的指定商品，创建一个**ArrayList<Product>**对象并存储在会话作用域中，最后将控制重定向到**showProduct.jsp**页面，在**JSP**页面中使用**<showProduct>**标签显示商品信息，并为其传递**productList**属性。
- [程序6.14 ProductServlet.java](#)
- [程序6.15 ProductTag.java](#)

6.3.5 在标签中使用EL

- 在TLD文件中使用下面代码定义showProduct标签。

<tag>

<name>showProduct</name>

<tag-class>com.mytag.ProductTag</tag-class>

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

<attribute>

<name>productList</name>

<required>true</required>


<rtexprvalue>true</rtexprvalue>

</attribute>

</tag>

6.3.5 在标签中使用EL

- 下面的JSP页面使用showProduct标签显示商品信息。
- [程序6.16 showProduct.jsp](#)



The screenshot shows a web browser window with the title '商品信息' (Product Information). The address bar displays the URL 'http://localhost:8080/helloweb/showProduct.jsp'. The main content area contains a table with four columns: '商品号' (Product ID), '商品名' (Product Name), '价格' (Price), and '库存量' (Inventory). The table lists three products: P3 (笔记本电脑, 4900.0, 8), P1 (数码相机, 1330.0, 3), and P2 (MP4播放器, 1990.0, 5).

商品号	商品名	价格	库存量
P3	笔记本电脑	4900.0	8
P1	数码相机	1330.0	3
P2	MP4播放器	1990.0	5

6.3.6 使用动态属性

- 在简单标签中还可以处理动态属性。所谓**动态属性（dynamic attribute）**，就是不需要在TLD文件中指定的属性。
- 要在简单标签中使用动态属性，标签处理类应该实现**DynamicAttributes**接口，该接口中只定义了一个名为**setDynamicAttribute()**的方法，它用来处理动态属性，格式为：

6.3.6 使用动态属性

```
public void setDynamicAttribute(  
    String uri, String localName,  
    Object value) throws JspException
```

- 参数uri表示属性的命名空间，如果属于默认命名空间，其值为null；参数localName表示要设置的动态属性名；value表示属性值。当标签声明允许接受动态属性，而传递的属性又没有在TLD中声明时将调用该方法。

6.3.6 使用动态属性

- 下面程序定义了一个带动态属性的标签处理类。在该类中创建了一个**String**对象 **output**，对每个动态属性它将被 **setDynamicAttribute()**更新。一旦结束读取属性，它将调用**doTag()**，把该**String**对象发送给**JSP**显示。
- [程序6.17 MathTag.java](#)

6.3.6 使用动态属性

- 在TLD件的<tag>标签中，动态属性需要使用<dynamic-attributes>元素定义并将其值指定为true，如下所示。

<tag>

<name>mathtag</name>

<tag-class>com.mytag.MathTag</tag-class>

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

<attribute>

<name>num</name>

<required>true</required>

<rtexprvalue>true</rtexprvalue>

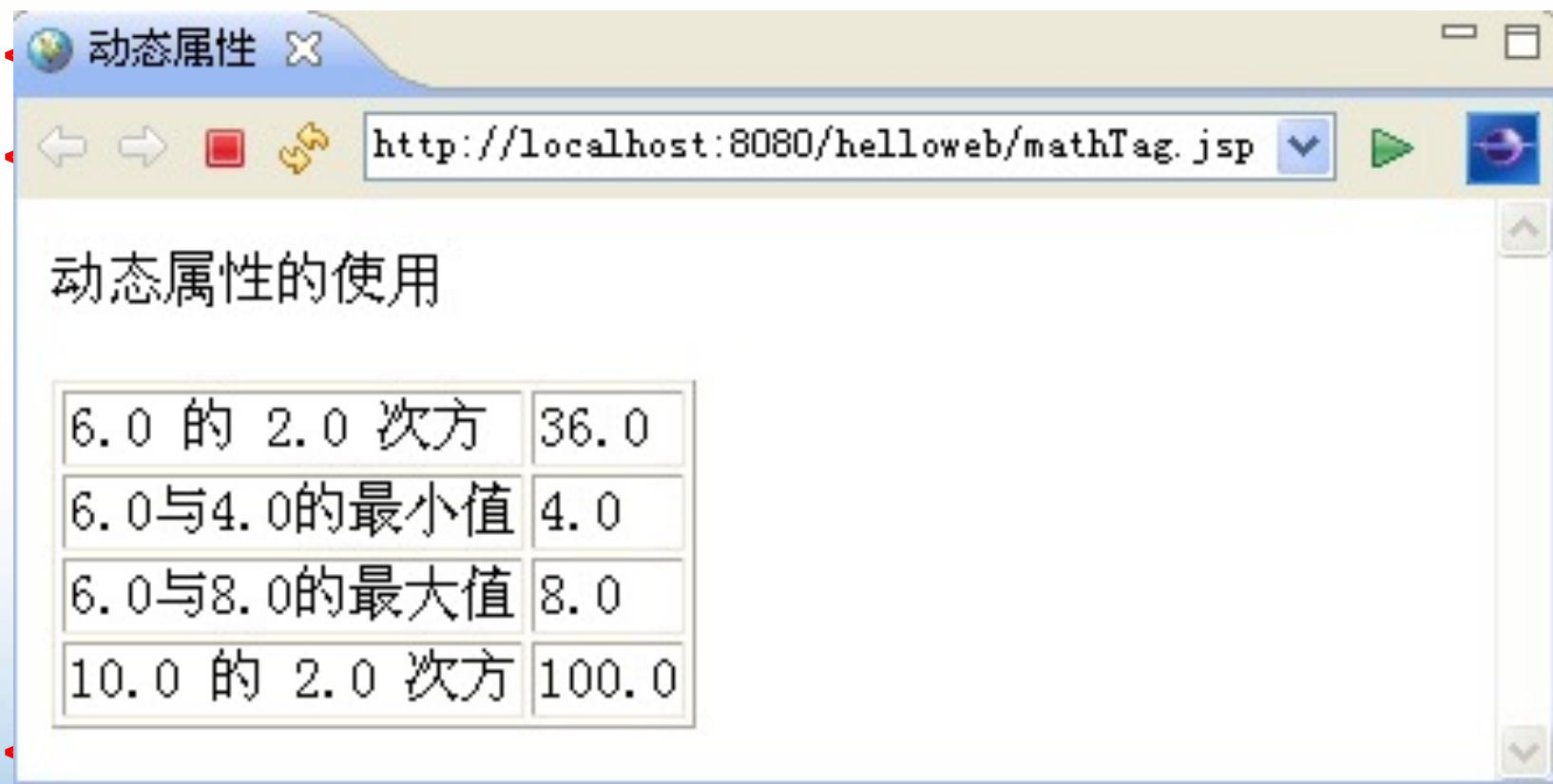
</attribute>

<dynamic-attributes>true</dynamic-attributes>

</tag>

6.3.6 使用动态属性

- 下面代码给出了如何在**JSP**中使用该标签。
- [程序6.18 mathTag.jsp](#)



6.3.7 编写协作标签

- 在标签的设计和开发中，通常一组标签协同工作，这些标签称为协作标签（**cooperative tags**）。
- 协作标签的一个最简单的例子是实现类似于**Java**编程语言提供的**switch-case**功能。来看下面三个标签：**<switch>**、**<case>**和**<default>**，它们可以用在**JSP**页面中，如下所示。
- [程序6.19 switchTag.jsp](#)

6.3.7 编写协作标签

- 程序6.20 SwitchTag.java
- 程序6.21 CaseTag.java
- 程序6.22 DefaultTag.java

6.4 JSP标准标签库

- 由于使用自定义标签可能造成程序员对标签的重复定义，因此从JSP2.0开始，JSP规范将标准标签库（JSP Standard Tag Library, JSTL）作为标准支持，它可以简化JSP页面和Web应用程序的开发。

6.4 JSP标准标签库

- 6.4.1 JSTL概述
- 6.4.2 JSTL核心标签库
- 6.4.3 通用目的标签
- 6.4.4 条件控制标签
- 6.4.5 循环控制标签
- 6.4.6 URL相关的标签

6.4.1 JSTL概述

- 在使用JSTL前，首先应该获得JSTL包，并安装到Tomcat服务器中。
- 可以到 Jakarta 网站 下载 JSTL 包，地址为 <http://jakarta.apache.org>。JSTL目前的最新版本是1.1.2。下载的是一个ZIP文件，文件名为jakarta-taglibs-standard-1.1.2.zip。将该文件解压到一个目录中，然后将lib目录中的两个文件jstl.jar和standard.jar复制到应用程序的WEB-INF/lib目录中，将tld目录中的文件复制到WEB-INF/tld目录中，这样就完成了JSTL的安装。

6.4.1 JSTL概述

- 实际上在Tomcat 服务器安装的examples示例应用程序的WEB-INF\lib目录中就包含jstl.jar和standard.jar两个文件，将它们复制到你的Web应用的WEB-INF/lib目录中即可。

6.4.1 JSTL概述

- JSTL共提供了5个库，每个子库提供了一组实现特定功能的标签，具体来说，这些子库包括：
 - 核心标签库，包括通用处理的标签。
 - XML标签库，包括解析、查询和转换XML数据的标签。
 - 国际化和格式化库，包括国际化和格式化的标签。
 - SQL标签库，包括访问关系数据库的标签。
 - 函数库，包括管理String和集合的函数。

6.4.2 JSTL核心标签库

- 本节主要介绍核心（**core**）标签库，该库的标签可以分成4类，如表6-5所示。
- 通用目的
 - `<c:out>`
 - `<c:set>`
 - `<c:remove>`
 - `<c:catch>`

6.4.2 JSTL核心标签库

- 条件控制
 - `<c:if>`
 - `<c:choose>`
 - `<c:when>`
 - `<c:otherwise>`

6.4.2 JSTL核心标签库

- 循环控制
 - `<c:forEach>`
 - `<c:forEachTokens>`
- URL处理
 - `<c:url>`
 - `<c:import>`
 - `<c:redirect>`
 - `<c:param>`

6.4.2 JSTL核心标签库

- 在JSP页面中使用JSTL，必须使用taglib指令来引用标签库，例如，要使用核心标签库，必须在JSP页面中使用下面的taglib指令。

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

6.4.3 通用目的标签

- 通用目的的标签包括
 - `<c:out>`
 - `<c:set>`
 - `<c:remove>`
 - `<c:catch>`

1. <c:out>标签

- <c:out>标签使用很简单，它有两种语法格式。

【格式1】不带标签体的情况

```
<c:out value = "value" [escapeXml="{true|false}"]  
    default = "defaultValue" />
```

- 如果escapeXml的值为true（默认值），表示将value属性值中包含的<、>、'、"或&等特殊字符转换为相应的实体引用（或字符编码），如小于号（<）将转换为<，大于号（>）将转换为>。如果escapeXml的值为false将不转换。

1. <c:out>标签

【格式2】带标签体的情况

```
<c:out value = "value"  
  [escapeXml="{true|false}"]>  
  default value
```

```
</c:out>
```

在【格式2】中默认值是在标签体中给出的。

1. <c:out>标签

- 在value属性的值中可以使用EL表达式，例如：

```
<c:out  
  value="${pageContext.request.remoteAdd  
r}" />
```

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

- 上述代码分别输出客户地址和number变量的值。
- 从<c:out>标签的功能可以看到，它可以替换JSP的脚本表达式。

2. <c:set>标签

- <c:set>标签设置作用域变量以及对象（如JavaBeans与Map）的属性值。该标签有下面4种语法格式。

【格式1】不带标签体的情况

```
<c:set var = "varName" value= "value"  
[scope = "{page| request| session|  
application}"] />
```

2. <c:set>标签

【格式2】带标签体的情况

```
c:set var = "varName" [scope =  
    "{page|request|session|application}"]>  
    body content  
</c:set>
```

【格式2】是在标签体中指定变量值。

2. <c:set>标签

- 例如，下面两个标签：

```
<c:set var="number" value="${4*4}"  
scope="session" />
```

与

```
<c:set var="number" scope="session">  
    ${4*4}  
</c:set>
```

都将变量number的值设置为16，且其作用域为会话作用域。

2. <c:set>标签

- 使用<c:set>标签还可以设置指定对象的属性值，对象可以是JavaBeans或Map对象。这可以使用下面两种格式实现。

【格式3】不带标签体的情况

```
<c:set target = "target"  
        property = "propertyName"  
        value = "value" />
```

2. <c:set>标签

- 【格式4】带标签体的情况

```
<c:set target = "target"  
        property = "propertyName" >  
    body content  
</c:set>
```

- **target**属性指定对象名，**property**属性指定对象的属性名（**JavaBeans**的属性或**Map**的键）。与设置变量值一样，属性值可以通过**value**属性或标签体内容指定。

2. <c:set>标签

- 下面程序为一个名为product的JavaBeans对象设置pname属性值。
- [程序6.23 setDemo.jsp](#)

3. <c:remove>标签

- <c:remove>标签用来从作用域中删除变量，它的语法格式为：

<c:remove var="*varName*"

[scope = "{page| request| session|
application}"] />

- **var**属性指定要删除的变量名，可选的**scope**属性指定作用域。如果没有指定**scope**属性，容器将先在**page**作用域查找变量，然后是**request**，接下来是**session**，最后是**application**作用域，找到后将变量清除。

4. <c:catch>标签

- <c:catch>标签的功能是捕获标签体中出现的异常，语法格式为：

```
<c:catch [var = "varName"]>  
    body content
```

```
</c:catch>
```

- 这里，**var**是为捕获到的异常定义的变量名，当标签体中代码发生异常时，将由该变量引用异常对象，变量具有**page**作用域。

4. <c:catch>标签

- 例如:

```
<c:catch var="myexception">
```

```
<%
```



http://localhost:8080/chap06/test.jsp

```
java.lang.ArithmeticException: / by zero  
/ by zero
```

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

```
<c:out value="${myexception.message}" />
```

6.4.4 条件控制标签

- 条件控制标签有4个：
 - `<c:if>`
 - `<c:choose>`
 - `<c:when>`
 - `<c:otherwise>`
- `<c:if>`和`<c:choose>`标签的功能类似于Java语言的if语句和switch-case语句。

1. <c:if>标签

- <c:if>标签用来进行条件判断，它有以下两种语法格式。

【格式1】不带标签体的情况

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

【格式2】带标签体的情况

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


1. <c:if>标签

- 每个<c:if>标签必须有一个名为test的属性，它是一个boolean表达式。对于【格式1】，只将test的结果存于变量varName中。对于【格式2】，若test的结果为true，则执行标签体。
- 例如，在下面代码中如果number的值等于16，则会显示其值。

```
<c:set var="number" value="${4*4}"  
  scope="session" />  
<c:if test="${number == 16}" var="result"  
  scope="session">  
  ${number}<br>  
</c:if> <br>  
<c:out value="${result}" />
```

2. <c:choose>标签

- <c:choose>标签类似于Java语言的switch-case语句，它本身不带任何属性，但包含多个<c:when>标签和一个<c:otherwise>标签，这些标签能够完成多分支结构。例如，下面代码根据color变量的值显示不同的文本。

2. <c:choose>标签

```
<c:set var="color" value="white" scope="session" />
```

```
<c:choose>
```

```
  <c:when test="${color == 'white'}">
```

白色!

```
  </c:when>
```

```
  <c:when test="${color == 'black'}">
```

黑色!

```
  </c:when>
```

```
  <c:otherwise>
```

其他颜色!

```
  </c:otherwise>
```

```
</c:choose>
```

6.4.5 循环控制标签

- 核心标签库的< c:forEach>和< c:forEachTokens>标签允许重复处理标签体内容。使用这些标签，能以三种方式控制循环的次数。
 - 对数的范围使用< c:forEach>以及它的begin、end和step属性。
 - 对Java集合中元素使用< c:forEach>以及它的var和items属性。
 - 对String对象中的令牌（token）使用< c:forEachTokens>以及它的items属性。

1. <c:forEach>标签

- <c:forEach>标签主要实现迭代，它可以对标签体迭代固定的次数，也可以在集合对象上迭代，该标签有两种格式。

【格式1】 迭代固定的次数

```
<c:forEach [var="varName"] [begin="begin"  
end="end" step="step"]  
[varStatus="varStatusName"]>  
body content  
</c:forEach>
```

1. <c:forEach>标签

- <c:forEach>标签还可以嵌套，如下的table99.jsp页面使用了嵌套的<c:forEach>标签实现输出九九表。
- [程序6.24 table99.jsp](#)

1. <c:forEach>标签

- 在<c:forEach>标签中还可以指定varStatus属性值来保存迭代的状态，例如，如果指定：

varStatus="status"

- 则可以通过**status**访问迭代的状态。其中包括：本次迭代的索引、已经迭代的次数、是否是第一个迭代、是否是最后一个迭代等。它们分别用**status.index**、**status.count**、**status.first**、**status.last**访问。

1. <c:forEach>标签

- 下面代码从0计数到10，每3个输出一个数。
- [程序6.25 foreach_1.jsp](#)

1. <c:forEach>标签

【格式2】在集合对象上迭代

```
<c:forEach var="varName" items="collection"  
[varStatus="statusName"] [begin="begin"  
end="end" step="step"]>  
    body content  
</c:forEach>
```

- 这种迭代主要用于对Java集合对象的元素迭代，集合对象如List、Set或Map等。标签对每个元素处理一次标签体内容。这里，items属性值指定要迭代的集合对象，var用来指定一个作用域变量名，该变量只在<c:forEach>标签内部有效。

1. <c:forEach>标签

- 下面例子使用<c:forEach>标签显示List对象的元素。假设有一个Book类定义如下。

```
package com.model;
```

```
public class Book{
```

```
    private String isbn;
```

```
    private String title;
```

```
    private double price;
```

```
// 这里省略了属性的setter方法和getter方法  
}
```

1. <c:forEach>标签

- 下面的BooksServlet创建一个List<Book>对象，然后将控制转发到books.jsp页面，在该页面中使用<c:forEach>标签访问每本书的信息。
- [程序6.26 BooksServlet.java](#)
- 在books.jsp页面中使用<c:forEach>标签访问列表中的元素，代码如下。
- [程序6.27 books.jsp](#)

2. <c:forToken>标签

- 该标签用来在字符串中的令牌（token）上迭代，它的语法格式为：

```
<c:forTokens items="stringOfTokens"  
  delims="delimiters"  
  [var="varName"]  
  [varStatus="varStatusName"]  
  [begin="begin"] [end="end"] [step="step"]>  
body content  
</c:forTokens>
```

2. <c:forToken>标签

- 下面的JSP页面tokens.jsp使用<forTokens>标签输出一个字符串中各令牌的内容。



6.4.6 URL相关的标签

- 与URL相关的标签有4个：
 - `<c:import>`
 - `<c:url>`
 - `<c:redirect>`
 - `<c:param>`。

1. <c:param>标签

- <c:param>标签主要用于在<c:import>、<c:url>和<c:redirect>标签中指定请求参数，它的格式有以下两种。

【格式1】 参数值使用value属性指定

```
<c:param name="name" value="value" />
```

【格式2】 参数值在标签体中指定

```
<c:param name="name" >
```

```
    param value
```

```
</c:param>
```

2. <c:import>标签

- <c:import>标签的功能与<jsp:include>标准动作的功能类似，可以将一个静态或动态资源包含到当前页面中。<c:import>标签有以下两种语法格式。

【格式1】资源内容作为字符串对象包含

```
<c:import url = "url" [context = "context"] [var =  
    "varName"]  
[scope = "{page| request| session| application}"]  
    [charEncoding = "charEncoding"]>  
    body content  
</c:import>
```


2. <c:import>标签

【格式2】资源内容作为Reader对象包含

```
<c:import url = "url" [context = "context"]  
    [varReader = "varreaderName"]  
    [charEncoding = "charEncoding"]  
    body content  
</c:import>
```

- 这里，varReader用于表示读取的文件的内容。其他属性与上面格式中含义相同。

2. <c:import>标签

- 下面代码使用<c:import>标签包含了 footer.jsp 页面，并向其传递了一个名为 email 的请求参数。
- [程序6.29 importDemo.jsp](#)
- 被包含的页面代码如下。
- [程序6.30 footer.jsp](#)

3. <c:redirect>标签

- <c:redirect>标签的功能是将用户的请求重定向到另一个资源，它有两种语法格式。

【格式1】不带标签体的情况

```
<c:redirect url = "url" [context = "context"] />
```

【格式2】在标签体中指定查询参数

```
<c:redirect url = "url" [context = "context"] >
```

```
    <c:param> subtags
```

```
</c:redirect>
```

3. <c:redirect>标签

- 该标签的功能与 `HttpServletResponse` 的 `sendRedirect()` 的功能相同。它向客户发送一个重定向响应并告诉客户访问由 `url` 属性指定的 URL。
- 与 `<c:import>` 标签一样，可以使用 `context` 属性指定 URL 的上下文，也可以使用 `<c:param>` 标签添加请求参数。

3. <c:redirect>标签

- 下面的代码片段给出了一个<c:redirect>标签如何转向到一个新的URL的例子。

```
<c:redirect url="/content.jsp">
```

```
    <c:param name="par1" value="val1"/>
```

```
    <c:param name="par2" value="val2"/>
```

```
</c:redirect>
```

4. <c:url>标签

- 如果用户浏览器不接受Cookie，那么就需要重写URL来维护会话状态。为此核心库提供了<c:url>标签。通过value属性来指定一个基URL，而转换的URL由JspWriter显示出来或者保存到由可选的var属性命名的变量中。

4. <c:url>标签

- <c:url>标签有如下两种格式。

【格式1】 不带标签体的情况

```
<c:url value="value" [context = "context"]  
      [var="varName"]  
      [scope="{page|request|session|application}"]  
/>
```

- **value**属性指定需要重写的URL，**var**指定的变量存放URL值，**scope**属性来指定**var**的作用域。

4. <c:url>标签

【格式2】带标签体的情况

```
<c:url value="value" [context = "context"]  
    [var="varName"]  
    [scope="{page|request|session|application}"]  
    >  
    <c:param name="name" value="value">  
</c:url>
```


4. <c:url>标签

- 下面是一个简单的例子。

<c:url value="/page.jsp" var="pagename"/>

- 由于**value**参数以斜杠开头，容器将把上下文名（假设为/helloweb）插入到该URL前面。

4. <c:url>标签

- 例如，如果浏览器接受Cookie，前面一行代码中var的值为：

/helloweb/page.jsp

- 如果浏览器不接受Cookie，容器将用其会话ID号重写该URL。在这种情况下，结果可能类似下面形式。

/helloweb/page.jsp;jsessionid=307FC94E10B7B2AEE74C3743964AA6FC

4. <c:url>标签

- 可以在<c:url>的标签中使用<c:param>标签向URL传递请求参数。下面代码给出了实现方法。

```
<c:url value="/page.jsp" var="pagename">  
  <c:param name="param1" value="${2*2}"/>  
  <c:param name="param2" value="${3*3}"/>  
</c:url>
```

- 在<c:param>标签中的参数通过name和value属性指定。如果浏览器接受Cookie，var属性的值将为：

```
/helloweb/page.jsp?param1=4&param2=9
```

6.5 小 结

- 自定义标签可以改善业务逻辑和表示逻辑的分离。自定义标签包括传统的自定义标签和简单的自定义标签。
- 自定义标签的开发首先需要创建标签处理类，该类封装了标签要实现的业务逻辑。自定义标签的类型包括：带或不带主体内容的标签、带属性的标签、迭代标签和嵌套标签等。

6.5 小 结

- JSP 2.0的一个主要目标是简化JSP的开发。为了减少标签处理类中的代码，开发了SimpleTag接口。
- 对于SimpleTag来说，在Web容器执行初始化后，它只需调用一个doTag()，它执行简单标签的所有处理功能。
SimpleTagSupport类是SimpleTag接口的一个实现类。

6.5 小 结

- JSTL是为实现Web应用程序常用功能而开发的标签库，它是由一些专家和用户开发的。使用JSTL可以提高JSP页面的开发效率，也可以避免重复开发标签库。
- JSTL由许多子库组成，每个子库提供了一组实现特定功能的标签，具体来说，这些子库包括core库、xml库、fmt库、sql库、functions库。