**JSP标签编程**

使用标签库编程的主要目的就是为了减少页面中的Scriptlet代码，使程序更容易理解和修改。

要想实现一个标签，可以直接继承TagSupport类，若要定义的标签内没有标签体，则直接覆写TagSupport类中的doStartTag()方法即可。

范例：定义一个简单的标签

package org.wp.tagdemo;

import javax.servlet.jsp.JspException;

import javax.servlet.jsp.JspWriter;

import javax.servlet.jsp.tagext.TagSupport;

public class HelloTag extends TagSupport{ //定义标签的操作类

@Override

public int doStartTag()throws JspException{

JspWriter out = super.pageContext.getOut() ; //取得页面输出流对象

try{

out.println("<h1>Hello World!!!!</h1>") ; //进行页面输出

}catch(Exception e){

e.printStackTrace() ;

}

return TagSupport.SKIP\_BODY ; //没有标签体

}

}

<?xml version=*"1.0"* encoding=*"GBK"* ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns=*"http://www.w3.org/1999/xhtml"*>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

**<%@ taglib prefix=*"mytag"* uri = *"wp\_hello"* %>**

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"* />

<title>在jsp中使用标签</title>

</head>

<body>

<h1><mytag:hello /></h1> <!-- 访问标签 -->

</body>

</html>

**标签描述文件：**

<?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\_1.xsd"*

version=*"2.1"*>

<tlib-version>1.0</tlib-version>

<short-name>firsttag</short-name> <!-- 表示标签库在TLD中的描述名称 -->

<tag>

<name>hello</name> <!-- 表示标签在jsp中的使用名称 -->

<tag-class>org.wp.tagdemo.HelloTag</tag-class> <!-- 表示这个标签指向的class文件 -->

<body-content>empty</body-content> <!-- 表示标签体内容为空 -->

</tag>

</taglib>

在以上程序中，在hellotag页面中，首先通过**<%@taglib%>**定义了一个标签的前缀名称“mytag”，之后通过此前缀名称调用了hellotag.tld文件中定义的标签的名称“hello”.

也可以在web.xml文件中，设置\*.tld文件的映射路径，设置映射路径以后，<%@taglib%>中的uri = ”映射路径名”。

**定义有属性的标签：**

例如：<jsp:forword page = “跳转路径”>标签，语句中jsp为标签前缀，forWord为标签名称，page即为标签属性。可以通过设置跳转路径值来设置一个跳转路径。实际上在使用自定义标签时用户也可以定义属性，例如现在要开发一个格式化日期显示标签，则用户就需要通过属性指定格式化日期的格式。

范例：格式化日期标签类---DateTag.java

package org.wp.tagdemo;

import java.text.SimpleDateFormat;

import java.util.Date;

import javax.servlet.jsp.JspException;

import javax.servlet.jsp.tagext.TagSupport;

/\*\*

\* 定义有属性的标签

\* \*/

public class DateTag extends TagSupport{ //格式化日期标签类

private String format ;

@Override

public int doStartTag() throws JspException {

SimpleDateFormat sdf = new SimpleDateFormat(this.format) ;

try{ //输出格式化后的日期

super.pageContext.getOut().write(sdf.format(new Date()));

}catch(Exception e){

e.printStackTrace() ;

}

return TagSupport.SKIP\_BODY ;

}

public void setFormat(String format){

this.format = format ;

}

public String getFormat(){

return this.format ;

}

}

在本类中定义了一个format属性，并且编写了getter和setter方法，当用户通过标签设置属性时，就会调用其中的setter方法自动完成属性的赋值。

<?xml version=*"1.0"* encoding=*"GBK"* ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns=*"http://www.w3.org/1999/xhtml"*>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ taglib prefix = *"mytag"* uri = *"wp\_date"* %>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"* />

<title>格式化日期标签</title>

</head>

<body>

<h1><mytag:date format=*"yyyy-MM-dd HH:mm:ss.SSS"* /></h1>

</body>

</html>

**标签描述文件：**

<?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\_1.xsd"*

version=*"2.1"*>

<tlib-version>1.0</tlib-version>

<short-name>firsttag</short-name> <!-- 表示标签库在TLD中的描述名称 -->

<tag>

<name>date</name> <!-- 表示标签在jsp中的使用名称 -->

<tag-class>org.wp.tagdemo.DateTag</tag-class> <!-- 表示这个标签指向的class文件 -->

<body-content>empty</body-content> <!-- 表示标签体内容为空 -->

<attribute>

<name>format</name> <!-- format为属性名 -->

<required>true</required> <!-- 表示此值必须设置 -->

<rtexpvalue>true</rtexpvalue> <!-- 支持表达式输出-->

</attribute>

</tag>

</taglib>

在datetag.jsp中需要通过<%@ prefix = “标签前缀” format = “映射名称/\*.tld文件路径”%>来定义引入标签的前缀和标签描述文件的映射路径和format定义的时间输出格式。

TagSupport类：

基本的标签掌握后可以发现标签的实现都需要继承TagSupport这个类，此类定义如下：

Public class TagSupport extends Object implements IterationTag,Serializable

IterationTag接口定义如下：

Public interface IterationTag extends Tag{

Public static final int EVAL\_BODY\_AGAIN ;

Public int doAfterBody() throws JspException ;

}

IterationTag接口本身就是Tag接口的子接口.

在整个TagSupport类中，最重要的四个方法如下：doStartTag(),doEndTag();doAfterTag();release();

doStartTag():此方法在标签开始时执行，有如下两种返回值：

SKIP\_BODY:表示忽略标签体的内容，而将执行权转交给doEndTag()方法;

EVAL\_BODY\_INCLUDE:表示执行标签体的内容。

doAfterTag():此方法是IterationTag和Tag接口的区别所在，用来重复执行标签体的内容，有如下两种返回值：

SKIP\_BODY:表示标签体内容会被忽略，并将执行权叫给doEndTag()方法执行。

EVAL\_BODY\_AGAIN:表示重复执行标签体的内容，会重复调用doAfterTag()方法，一直循环下去，知道doAfterTag()方法返回SKIP\_BODY为止。

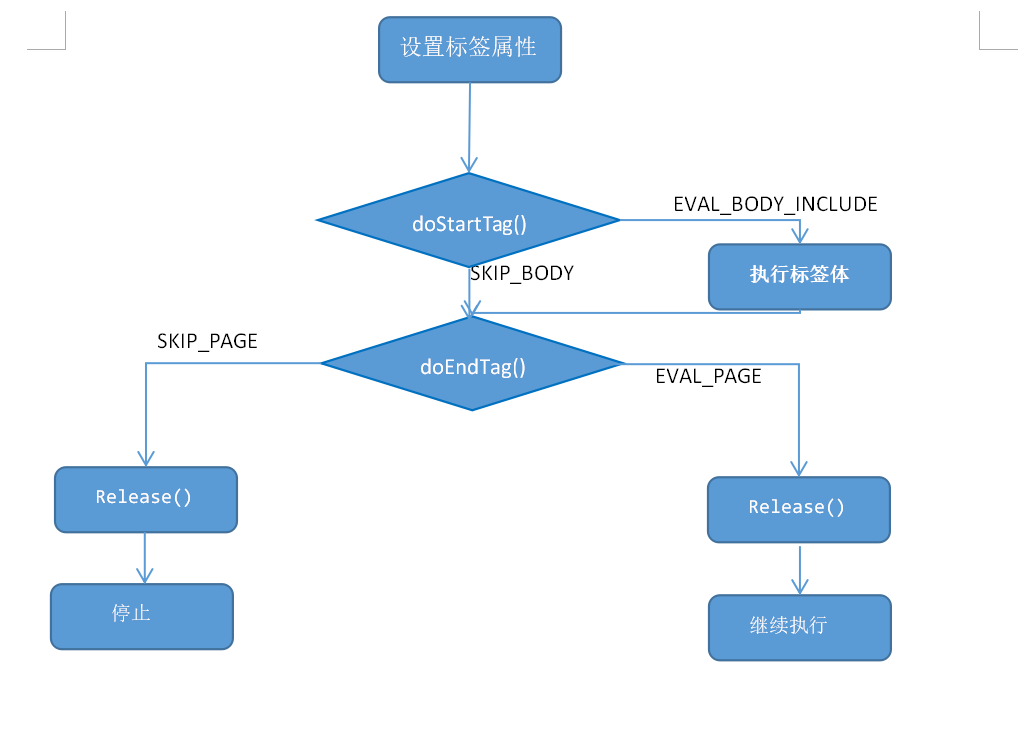
doEndTag():此方法在标签结束时执行，有如下两种返回值：

SKIP\_PAGE:表示JSP页面应该立即停止执行，并将所有的输出回传到浏览器上。

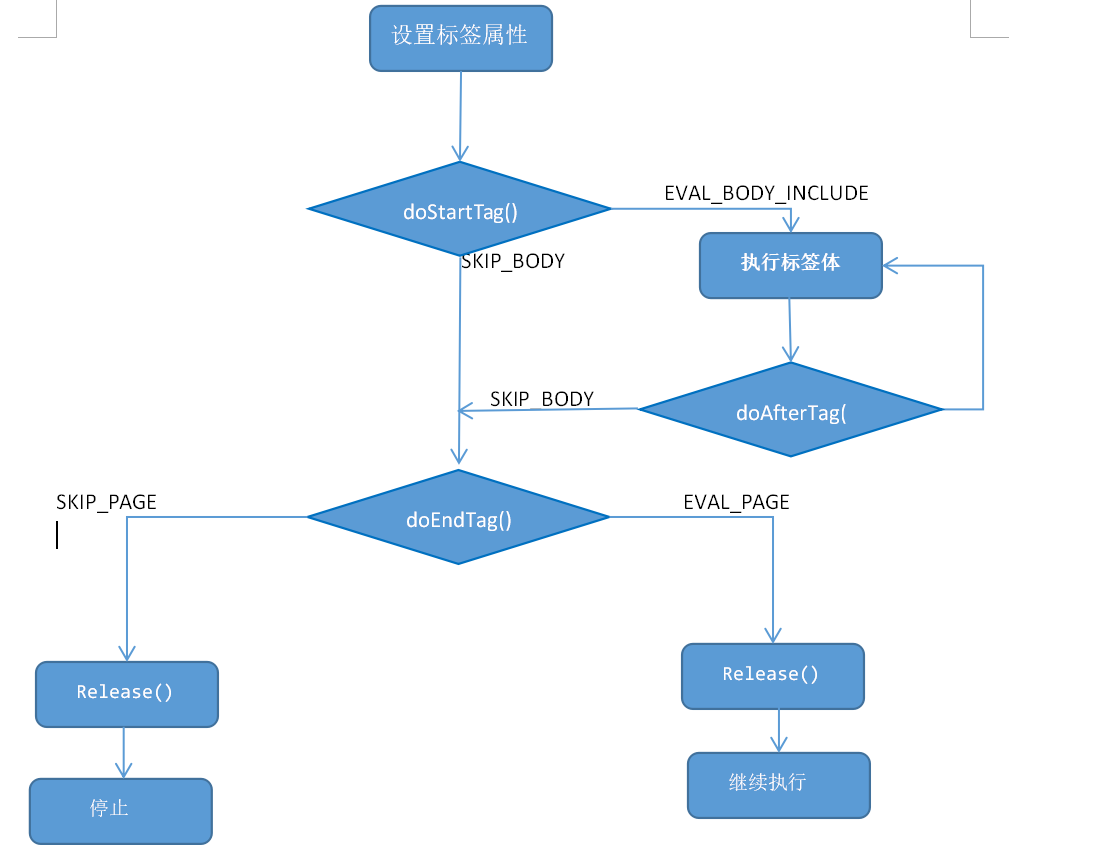
EVAL\_PAGE:表示jsp可以正常的运行完毕。

release():将标签处理类所产生或是获得的资源全部释放，并等待用户下次使用。

Tag和IterationTag接口的主要区别在doAfterTag()方法上，下面用流程图来表示：



**Tag接口执行流程**



**InterationTag接口执行流程**

在上述流程图中可以发现，当执行doStartTag()方法时，如果返回的是SKIP\_BODY,则会执行doEndTag()方法，如果返回的是EVAL\_BODY\_INCLUDE，则执行标签体内的内容。

**定义有标签体的标签库：**

**范例：判断在某一属性范围内是否存在某一属性，如果存在就进行输出。**

package org.wp.tagdemo;

import javax.servlet.jsp.JspException;

import javax.servlet.jsp.PageContext;

import javax.servlet.jsp.tagext.TagSupport;

public class AttributeTag extends TagSupport{ //标签处理类

private String name ; //接收属性名称

private String scope ; //接收属性范围

@Override

public int doStartTag() throws JspException {

Object value = null ;

if("page".equals(this.scope)){ //是否是page范围

value = super.pageContext.getAttribute(name, PageContext.PAGE\_SCOPE) ;

}else if("request".equals(this.scope)){

value = super.pageContext.getAttribute(name, PageContext.REQUEST\_SCOPE) ;

}else if("session".equals(this.scope)){ //取得属性名称

value = super.pageContext.getAttribute(name, PageContext.SESSION\_SCOPE) ;

}else{

value = super.pageContext.getAttribute(name, PageContext.APPLICATION\_SCOPE) ;

}

if(value == null){ //没有查找到属性

return TagSupport.SKIP\_BODY ; //不执行标签体内容

}else{

return TagSupport.EVAL\_BODY\_AGAIN ; //执行标签体内容

}

}

public String getName(){

return this.name ;

}

public void setName(String name){

this.name = name ;

}

public String getScope(){

return this.scope ;

}

public void setScope(String scope){

this.scope = scope ;

}

}

定义标签描述文件：

<tag>

<name>present</name> <!-- 表示标签在jsp中的使用名称 -->

<tag-class>org.wp.tagdemo.AttributeTag</tag-class> <!-- 表示这个标签指向的class文件 -->

<body-content>JSP</body-content> <!-- 表示标签体的内容 -->

<attribute>

<name>name</name> <!-- name为属性名 -->

<required>true</required> <!-- 表示此值必须设置 -->

<rtexpvalue>true</rtexpvalue> <!-- 表示属性值是请求时表达式的结果 -->

</attribute>

<attribute>

<name>scope</name>

<required>true</required>

<rtexpvalue>true</rtexpvalue>

</attribute>

</tag>

<?xml version=*"1.0"* encoding=*"GBK"* ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns=*"http://www.w3.org/1999/xhtml"*>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ taglib prefix = *"mytag"* uri = *"wp\_tag\_1"* %>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"* />

<title></title>

</head>

<body>

<%

String scope = "session" ; //定义一个变量表示查找范围

session.setAttribute("username","文攀") ; //设置一个session属性

%>

<mytag:present name=*"username"* scope=*"session"*>

<h2><%=scope %>范围存在属性，内容是："${sessionScope.username }"</h2> <!-- 标签体 -->

</mytag:present>

<mytag:present name=*"allusers"* scope=*"request"*>

<h2>request范围存在属性，内容是："${requestScope.allusers}"</h2>

</mytag:present>

</body>

</html>

在presenttag.jsp文件中首先设置了一个session属性，之后利用<mytag:present>判断属性是否存在，如果存在则执行标签体内容。

范例：开发迭代标签：

package org.wp.tagdemo;

import java.util.Iterator;

import java.util.List;

import javax.servlet.jsp.JspException;

import javax.servlet.jsp.PageContext;

import javax.servlet.jsp.tagext.TagSupport;

public class IterateTag extends TagSupport{ //迭代标签处理类

private String name ; //属性名称

private String scope ; //属性保存范围

private String id ; //每次迭代的对象

private Iterator<?> iter ; //所有接收到的数据

@Override

public int doStartTag() throws JspException{

Object value = null ;

if("page".equals(this.scope)){

value = super.pageContext.getAttribute(name, PageContext.PAGE\_SCOPE) ;

}else if("request".equals(this.scope)){

value = super.pageContext.getAttribute(name, PageContext.REQUEST\_SCOPE) ;

}else if("session".equals(this.scope)){

value = super.pageContext.getAttribute(name, PageContext.SESSION\_SCOPE) ;

}else{

value = super.pageContext.getAttribute(name, PageContext.APPLICATION\_SCOPE) ;

}

if(value != null && value instanceof List<?>){ //如果是list接口实例

this.iter = ((List<?>)value).iterator() ;

if(iter.hasNext()){

super.pageContext.setAttribute(id, iter.next()) ;

return TagSupport.EVAL\_BODY\_AGAIN ; //执行标签体的内容

}else{

return TagSupport.SKIP\_BODY ; //退出标签执行

}

}else{ //不是List接口的实例，不处理

return TagSupport.SKIP\_BODY ; //退出标签执行

}

}

@Override

public int doAfterBody() throws JspException{

if(iter.hasNext()){ //判断是否还有内容

super.pageContext.setAttribute(id, iter.next()) ;

return TagSupport.EVAL\_BODY\_AGAIN ; //重复执行标签体

}else{

return TagSupport.SKIP\_BODY ; //退出标签执行

}

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getScope() {

return scope;

}

public void setScope(String scope) {

this.scope = scope;

}

public String getId() {

return id;

}

public void setId(String id) {

this.id = id;

}

}

<?xml version=*"1.0"* encoding=*"GBK"* ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns=*"http://www.w3.org/1999/xhtml"*>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ page import = *"java.util.\*"* %>

<%@ taglib prefix = *"mytag"* uri = *"wp\_tag"* %>

<%@ taglib prefix = *"mytag1"* uri = *"/WEB-INF/wptag.tld"* %>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"* />

<title></title>

</head>

<body>

<%

List<String> all = **new** ArrayList<String>() ;

all.add("www.baidu.com") ;

all.add("www.wenpan.cn") ;

all.add("www.xihuaedu.com") ;

request.setAttribute("all", all) ; //设置在request属性范围

%>

<mytag:present name=*"all"* scope=*"request"*> <!-- 在request属性范围是否存在名为all的属性 -->

<mytag1:iterate name=*"all"* scope=*"request"* id=*"url"*> <!-- 迭代输出 -->

<h3>网站：${url}</h3>

</mytag1:iterate>

</mytag:present>

</body>

</html>

<tag>

<name>iterate</name>

<tag-class>

org.wp.tagdemo.IterateTag

</tag-class>

<body-content>JSP</body-content>

<attribute>

<name>id</name> <!-- ID为属性名 -->

<required>true</required>

<rtexpvalue>true</rtexpvalue>

</attribute>

<attribute>

<name>name</name>

<required>true</required>

<rtexpvalue>true</rtexpvalue>

</attribute>

<attribute>

<name>scope</name>

<required>true</required>

<rtexpvalue>true</rtexpvalue>

</attribute>

</tag>

在本程序中首先在request属性范围保存了一个List集合,之后使用<mytag:present>标签判断在request属性范围内是否存在此属性，如果存在，则使用迭代标签输出全部内容。在迭代标签中，会将集合中的每一个对象都保存在page属性范围中，并且属性名称就是ID所指定的，所以在输出的时候直接使用表达式输出即可。

BodyTagSupport类和TagExtraInfo类和VariableInfo类的主要作用就是，将标签中的属性变为一个变量，使该属性在Scriptlet表达式中也可以使用

Eg:<jsp:useBean id = “simple” scope = “page” class = “org.wp.demo.SimpleBean”/>

<%

Simple.setName(“文攀”) ;

Simple.setAge(18) ;

%>

在此段代码中，useBean标签中，simple是属性值，也是一个对象。但是也可以在下面的scriptlet语句中使用。

**简单标签：**

为了简化标签开发的复杂度，专门增加了一个制作简单标签库的SimpleTagSuppot类，直接覆写其中的doTag()方法即可完成。

下面使用简单标签实现日期格式化操作显示

范例：

package org.wp.tagdemo;

import java.io.IOException;

import java.text.SimpleDateFormat;

import java.util.Date;

import javax.servlet.jsp.JspException;

import javax.servlet.jsp.tagext.SimpleTagSupport;

public class SimpleDateTag extends SimpleTagSupport{ //定义简单标签处理类

private String format ; //接收格式化模板

@Override

public void doTag()throws JspException ,IOException{ //省去了控制返回值

SimpleDateFormat sdf = new SimpleDateFormat(this.format) ;

try{ //输出格式化后的日期

super.getJspContext().getOut().write(sdf.format(new Date()));

}catch(Exception e){

e.printStackTrace() ;

}

}

public String getFormat(){

return this.format ;

}

public void setFormat(String format){

this.format = format ;

}

}

<?xml version=*"1.0"* encoding=*"GBK"* ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns=*"http://www.w3.org/1999/xhtml"*>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ taglib prefix = *"mytag"* uri = *"/WEB-INF/wptag.tld"* %>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"* />

<title>简单标签实例</title>

</head>

<body>

<h1>

<mytag:simpledate format=*"yyyy-MM-dd HH:mm:ss.SSS"* />

</h1>

</body>

</html>

简单标签与前面的标签开发相比，简单了许多，也不用去控制返回值。

**DynamicAttributes接口**

此接口的主要功能是用于完成动态属性设置，即用户在使用一个标签时可以根据自己的需要任意设置多个属性。该接口定义如下：

Public interface SynamicAttributes{

Public void setDanamicAttribute(String uri,String localName,Object value) throws JspException;

}

在该接口中只定义了一个setDynamicAttribute()方法，在此方法中只需要关心localName（表示动态设置的属性名称）和value（表示动态设置的属性内容）这两个参数即可。

范例：完成一个动态的加法操作

**package** org.wp.tagdemo;

**import** java.io.IOException;

**import** java.util.HashMap;

**import** java.util.Iterator;

**import** java.util.Map;

**import** javax.servlet.jsp.JspException;

**import** javax.servlet.jsp.tagext.DynamicAttributes;

**import** javax.servlet.jsp.tagext.SimpleTagSupport;

**public** **class** DynamicAddTag **extends** SimpleTagSupport **implements** DynamicAttributes{

**private** Map<String, Float> num = **new** HashMap<String,Float>() ;

@Override

**public** **void** doTag()**throws** JspException,IOException{

Float sum = 0.0f ;

Iterator<Map.Entry<String, Float>> iter = **this**.num.entrySet().iterator() ;//将map集合变为set集合便于用迭代器输出

**while**(iter.hasNext()){

Map.Entry<String, Float> value = iter.next() ; //取出每一个Map.entry

sum = sum + value.getValue() ;

}

**super**.getJspContext().getOut().write(sum + ""); //输出计算结果

}

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

JspException{//取出设置的动态属性内容

num.put(localName, Float.*parseFloat*(value.toString())) ;

}

}

<?xml version=*"1.0"* encoding=*"GBK"* ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns=*"http://www.w3.org/1999/xhtml"*>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ taglib prefix=*"mytag\_11"* uri=*"wp\_tag"*%>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"* />

<title></title>

</head>

<body>

<h2>计算结果：

<mytag\_11:add num1 = *"11.2"* num2 = *"12.3"* num3 = *15.9* num4 = *"50.2"*/>

</h2>

</body>

</html>

<tag>

<name>add</name>

<tag-class>

org.wp.tagdemo.DynamicAddTag

</tag-class>

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

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

</tag>

在配置文件中最重要的就是增加了一个<dynamic-attributes>元素，其中内容设置为true，表示可以设置动态属性。

在本标签处理中实现了DnamicAttributes接口，所以可以动态的设置标签中的属性，每一个设置到标签中的动态属性都通过循环调用setDynamicAttribute()方法保存在Map集合中，最后在doTag()方法中完成累加操作，并直接输出。

**要想实现一个标签就要编写标签处理类，标签描述文件，还要在web.xml文件中进行配置。**

**标签处理类可以通过继承TagSupport和BodyTagSupport类实现。**

**标签编程可以让一个jsp页面完全摆脱掉Scriptlet的困扰，使开发的jsp页面更加的美观。**

**在简单标签中提供了SimpleTagSupport类用户开发时可以避免处理各个方法的返回值问题，而直接在doTag()方法中编写标签处理即可。**

**Tomcat数据源**

动态web开发的最大特点就是可以进行数据库操作。传统的JDBC操作由于步骤的重复性会造成程序的性能降低，此时就可以通过数据源提升与数据库操作性能。

传统的JDBC操作原理及步骤：

1. 加载数据库驱动程序，数据库驱动程序通过classpath配置。
2. 通过DriverManager类取得数据区连接对象。
3. 通过Connection实例化PreparedStatement对象，编写SQL命令操作数据库。
4. 关闭数据库。

从以上的步骤来看每一个用户进行数据库操作时候都要经过1.2.4三个相同的操作步骤，数据源的核心就是去除1.2.4三个重复步骤。

数据源的操作核心原理就是：在一个对象池中保存多个数据库连接，这样以后再进行数据库操作时，直接从数据库连接池中取出一个数据库连接，当数据库操作完成之后，再将此数据库连接池中，等待其他用户继续使用。

在数据库连接池中都是通过数据源(javax.sql.DataSource)访问的，即可以通过javax.sql.DataSource类取得一个Connection对象，但是要想得到一个DataSource对象需要使用JNDI进行查找。

JNDI:命名及目录接口。其主要功能就是通过一个名称key查找到一个对象“value”.

数据源连接池是在TomCat上进行配置的，所以一定要将数据库的驱动程序复制到%TOMCAT\_HOME%/common/lib文件夹中。

在server.xml文件中配置数据库连接池：

<Context path = "/mldn" docBase = "D:\mldnwebdemo" reloadable "true">

<Resource name = "jdbc/mldn"

auth = "Container"

type = "javax.sql.DataSource"

maxActive = "100"

maxldle = "30"

maxWait = "10000"

username = "scott"

password = "tiger"

driverClassName = "oracle.jdbc.driver.OracleDriver"

url = "jdbc:oracle:thin:@localhost:1521:mldn"

/>

</Context>

配置web.xml

<resource-ref>

<res-ref-name>jdbc/mldn</res-ref-name>

<res-type>javax.sql.DataSource</res-type>

<res-auth>Container</res-auth>

</resource-ref>

**如果使用数据源取得数据库连接操作必须按照以下步骤进行：**

1. **初始化名称查找上下文：Context ctx = new InitialContext() ;**
2. **通过名称查找DataSource对象：DataSource ds = (DataSource)ctx.lookup(JNDI名称) ；**
3. **通过DataSource取得一个数据库连接： Connetion conn = ds.getConnection() ;**

范例：通过数据源取得数据库连接：

<?xml version=*"1.0"* encoding=*"GBK"* ?>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns=*"http://www.w3.org/1999/xhtml"*>

<head>

<%@ page language=*"java"* contentType=*"text/html; charset=GBK"*

pageEncoding=*"GBK"*%>

<%@ page import = *"javax.naming.\*"* %>

<%@ page import = *"javax.sql.\*"* %>

<%@ page import = *"java.sql.\*"* %>

<meta http-equiv=*"Content-Type"* content=*"text/html; charset=GBK"* />

<title></title>

</head>

<body>

<%

String DSNAME = "**java:comp/env**/jdbc/mldn" ; //JNDI名称

Context ctx = **new** InitialContext() ; //初始化名称查找上下文

DataSource ds = (DataSource)ctx.lookup(DSNAME) ;//取得DataSource实例

Connection conn = ds.getConnection() ; //取得数据库连接

%>

<%= conn %>

<%

conn.close() ;

%>

</body>

</html>

范例：修改DAO中的DataBaseConnection

**package** org.wp.tagdemo;

**import** java.sql.Connection;

**import** javax.naming.Context;

**import** javax.naming.InitialContext;

**import** javax.sql.DataSource;

**public** **class** DatabaseConnection {

**private** **static** **final** String ***DSNAME*** = "java:comp/env/jdbc/mldn" ;

**private** Connection conn = **null** ;

**public** DatabaseConnection() **throws** Exception{

**try**{

Context ctx = **new** InitialContext() ; //初始化名称查找上下文

DataSource ds = (DataSource)ctx.lookup(***DSNAME***) ; //取得DataSource的实例

**this**.conn = ds.getConnection() ; //取得数据库连接

}**catch**(Exception e){

e.printStackTrace() ;

}

}

**public** Connection getConnection(){

**return** **this**.conn ;

}

**public** **void** closse()**throws** Exception{

**if**(**this**.conn != **null**){

**try**{

**this**.conn.close() ;

}**catch**(Exception e){

**throw** e ;

}

}

}

}

在以上程序中，使用了数据源连接数据库。省去了对URL,PASSWOERD,UERNAME的编写。