第 10 章 自定义标签的开发与应用

作者：李宁
网名：银河使者
Blog：http://nokiaguy.blogjava.net/
Email：techcast@126.com

在 JSTL 提供了四个标签库（核心标签库、国际化标签库、数据库标签库和 XML 标签库），涉及到了几十个标签。虽然这些标签可以完成比较复杂的工作，但它们仍然无法满足程序中的特殊需求。因此，就需要用户根据自己的需要来定制 JSP 标签，这种由用户自己实现的 JSP 标签被称为自定义标签。

10.1 自定义标签基础

自定义标签和 JSTL 中的标签从技术上看没有任何区别，可以将这些标签统称为 JSP 标签。JSP 标签在 JSP 页面中通过 XML 语法格式被调用，当 JSP 引擎将 JSP 页面翻译成 Servlet 时，就将这些调用转换成执行相应的 Java 代码。也就是说，JSP 标签实际上就是调用了某些 Java 代码，只是在 JSP 页面中以另外一种形式（XML 语法格式）表现出来。

10.1.1 编写输出随机数的标签

开始自定义标签的学习之前，在这一节先来实现一个简单的自定义标签，以使读者做一下热身，同时读者通过本节的例子可以对自定义标签的实现方法和过程有一个感性的认识。

【实例 10-1】 实现输出随机数的标签

1. 实例说明

在本例实现的自定义标签（random 标签）的功能是输出一个指定范围的随机整数。random 标签有如下几个特征：

没有标签体。

有两个属性：min 和 max。其中 min 属性表示生成随机数的最小值，max 属性表示生成随机数的最大值。min 属性的默认值是 0，max 属性的默认值是 Integer.MAX_VALUE。

生成随机数的范围是 min <= random < max。

random 标签的标准调用形式如下：

```
<ct:random min="1" max="1000" />
```

其中“ct”是调用标签时的前缀，通过 taglib 指定的 prefix 属性指定。上面的代码的功能是输出一个在

1（包括）和 1000（不包括）之间的随机数。

2. 编写标签类

标签类是自定义标签的核心部分。实现标签类的方法有很多，但最简单的方法是编写一个从 javax.servlet.jsp.tagext.TagSupport 类继承的 Java 类，并在该类中覆盖 TagSupport 类的 doStartTag 方法。为了读取标签中的属性值，还需要在标签类中为每一个标签属性提供一个相应数据类型的标签类属性以及该属性的 setter 方法（不需要 getter 方法）。生成随机数的代码需要放在标签类的 doStartTag 方法中。该标签类的实现代码如下：

```
package chapter10;
import java.io.IOException;
import java.util.Random;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
public class RandomTag extends TagSupport
{
    //  封装 random 标签的两个属性的 JavaBean 属性
    private int min = 0;
    private int max = Integer.MAX_VALUE;
    //  min 属性的 setter 方法
    public void setMin(int min)
    {
        this.min = min;
    }
    //  max 属性的 setter 方法
    public void setMax(int max)
    {
        this.max = max;
    }
    //  覆盖 TagSupport 类的 doStartTag 方法
    //  当遇到标签（也就是<ct:random>）的开始标记时调用该方法
    @Override
    public int doStartTag() throws JspException
    {
        try
        {
            Random random = new Random();
            //  生成一个在 min 和 max 之间的随机数
            int result = min + random.nextInt(max - min);
            //  将生成的随机数输出到客户端
            pageContext.getOut().write(String.valueOf(result));
        }
        catch (IOException e)
        {
        }
        //  TagSupport 类的 doStartTag 方法默认返回 SKIP_BODY，表示忽略自定义标签体
        return super.doStartTag();
    }
}
```

在 RandomTag 类的 doStartTag 方法中使用了一个 pageContext 变量来获得 JspWriter 对象（JSP 的 out 内置对象）。pageContext 变量是在 TagSupport 类中定义的一个类变量，该变量通过 TagSupport 类中的 setPageContext 方法进行赋值。实际上，setPageContext 方法是在 Tag 接口中定义的，而 TagSupport 实现了 Tag 接口的 setPageContext 方法。Servlet 容器在调用 doStartTag 方法之前，会先调用 Tag 接口的 setPageContext

方法来初始化 pageContext 变量。Tag 接口将在 10.1.3 节介绍，在这里只要知道所有的标签类都必须实现 Tag 接口。为了简化标签类的实现，JSP API 提供了一个 TagSupport 类，有了 TagSupport 类，用户在编写标签类时就不需要实现 Tag 接口的所有方法了。

3. 编写标签库描述符文件（TLD 文件）

在 JSTL 中有若干 TLD 文件（详见图 9.1），其中 tld 是 Tag Library Descriptor（标签库描述符）的缩写。要想使一个自定义标签正常工作，必须在 TLD 文件中对该标签进行配置。在 TLD 文件中主要配置如下两部分内容：

标签库的配置信息

标签库中具体标签的配置信息

标签库的配置信息主要包括如下的内容：

标签库的版本：使用<tlib-version>标签设置。

正常使用标签库中的标签所需要的最低 JSP 版本：使用<jsp-version>标签设置。

标签库的默认前缀：使用<short-name>标签设置。

标签库的 URI：使用<uri>标签设置。

标签库的描述信息：使用<description>标签设置。

标签的配置信息主要包括如下的内容：

标签名：使用<name>标签设置。

标签对应的 Java 类：使用<tag-class>标签设置。

标签体的类型：使用<body-content>标签设置。

标签的描述信息：使用<description>标签设置。

标签属性的信息：每一个标签属性对应一个<attribute>标签。在<attribute>标签中可以指定标签名（使用<name>标签设置）、该属性是否必须指定（使用<required>标签设置）、该属性是否支持动态属性值（使用<rtexprvalue>标签指定）等配置信息。

在 WEB-INF 目录中建立一个 jsp-taglib.tld 文件，并在该文件中输入如下的内容：

```
<!--  标签库描述符文件头  -->
<?xml version="1.0" encoding="UTF-8" ?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd" version="2.0">
   <!--  标签库的配置信息  -->
   <description>自定义标签库</description>
   <tlib-version>1.1</tlib-version>
   <short-name>ct</short-name>
   <uri>http://nokiaguy.blogjava.net</uri><taglib>
   <!--  random 标签的配置信息  -->
   <tag>
      <description>产生一个指定范围的随机数</description>
      <name>random</name>
      <tag-class>chapter10.RandomTag</tag-class>
      <body-content>empty</body-content>
      <attribute>
         <name>min</name>
         <required>false</required>
         <rtexprvalue>false</rtexprvalue>
      </attribute>
      <attribute>
         <name>max</name>
```

```
            <required>false</required>
            <rtexprvalue>false</rtexprvalue>
        </attribute>
    </tag>
</taglib>
```

上面的配置代码分为三部分：标签库描述符文件头、标签库配置信息和 random 标签的配置信息。其中描述库文件头的内容并不需要读者去记忆，读者只需要将 JSTL 中的任何一个 TLD 文件打开，将其中的标签库描述文件头部分复制过来即可。如果在标签库描述符文件中包含中文，需要将 encoding 属性改成“UTF-8”或“GBK”。

标签库的 URI 被<uri>元素指定为“http://nokiaguy.blogjava.net”，该标签值就是 taglib 指令的 uri 属性值。<short-name>元素指定了标签库的默认前缀。要注意的是，该默认前缀并不等于 taglib 指令的 prefix 属性值，也就是说，taglib 指令的 prefix 属性值和<short-name>元素的值毫无关系。<short-name>元素值实际上只是个推荐的标签库前缀，如国际化标签库的描述符文件（fmt.tld）中的<short-name>元素值是“fmt”。在使用 taglib 指令引用某个标签库时，应尽量使用<short-name>元素推荐的标签库前缀，当然，也可以设置其他的前缀名。

random 标签的标签体类型为“empty”（<body-content>元素的值），表示该标签不支持标签体。关于<body-content>元素支持的其他值，将在后面的部分详细介绍。random 标签的 min 属性和 max 属性都被设置可选的属性，并且都不支持动态属性值。因此，在 JSP 页面中设置 random 标签的这两个属性时只能直接为它们赋值。

4. 测试 random 标签

在<Web 根目录>\chapter10 目录建立一个 random.jsp 文件（在本章的所有 JSP 文件都放在该目录下），并输入如下的内容：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<!--  使用 forEach 标签产生 10 个 100 至 200 之间的随机数  -->
<c:forEach begin="1" end="10">
    <!--  调用 random 标签  -->
    <ct:random min="100" max="200" />
</c:forEach>
```

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/random.jsp
```

浏览器显示的输出结果如图 10.1 所示。

图 10.1　使用 random 标签产生 10 个 100 至 200 之间的随机数

5. 程序总结

在部署和安装自定义标签时，TLD 文件应放在 WEB-INF 目录或其子目录中（包括 classes 和 lib 目录）。

根据本例实现的 random 标签，可以将开发自定义标签的基本步骤总结如下：

实现一个标签类。该标签类可以实现 Tag 接口、继承 TagSupport 类或实现其他的接口（这些接口将在后面的部分详细介绍）。

在标签库描述符文件（TLD 文件）中配置自定义标签。

部署和安装自定义标签。主要是将.class 文件放在 WEB-INF\classes 目录中，并且将 TLD 文件放在 WEB-INF 目录或其子目录中。

10.1.2 自定义标签能做什么

自定义标签除了可以读取标签的属性值外，还可以完成如下的工作：

单次执行标签体中的内容。

重复执行标签体中的内容。

修改标签体中的内容。

忽略 JSP 页面中位于自定义标签后面的内容。

在上一节介绍了 Tag 接口中的 doStartTag 方法，该方法在 Web 容器执行到自定义标签的开始标记时被调用。除了这个方法，在 Tag 接口中还有 doEndTag 方法，该方法在 Web 容器中执行到自定义标签的结束标记时被调用。

doStartTag 方法可以通过返回如下两个值来控制 Web 容器是否执行自定义标签的标签体：

EVAL_BODY_INCLUDE：执行自定义标签的标签体。

SKIP_BODY：忽略（不执行）自定义标签的标签体。

doEndTag 方法可以通过返回如下两个值来控制 Web 容器是否忽略 JSP 页面中位于自定义标签后面的内容：

EVAL_PAGE：继续执行自定义标签后面的内容。

SKIP_PAGE：忽略自定义标签后面的内容。

其中 EVAL_BODY_INCLUDE、SKIP_BODY、EVAL_PAGE 和 SKIP_PAGE 是在 Tag 接口中定义的整型常量，所有实现 Tag 接口的类都可以直接使用这些常量。

除此之外，实现 IterationTag 接口的标签类还可以重复执行标签体。IterationTag 是 Tag 接口的子接口。在 IterationTag 接口中有一个 doAfterBody 方法，该方法可以通过返回如下两个值来决定是否重复执行自定义标签体的内容：

EVAL_BODY_AGAIN：重复执行标签体的内容。

SKIP_BODY：不再执行标签体的内容。

其中 EVAL_BODY_AGAIN 是在 IterationTag 接口中定义的整型常量。如果 doAfterBody 方法返回 SKIP_BODY，Web 容器会继续执行自定义标签的结束标记，同时会调用 doEndTag 方法。

从上面的描述可以将 Web 容器执行自定义标签的过程总结如下：

1. Web 容器首先会执行自定义标签的开始标记，同时会调用标签类的 doStartTag 方法。

2. 如果 doStartTag 方法返回 EVAL_BODY_INCLUDE，Web 容器在执行完标签体的内容后，会调用标签类的 doAfterBody 方法；如果 doStartTag 方法返回 SKIP_BODY，doAfterBody 方法不会被调用，Web 容器会直接调用标签类的 doEndTag 方法。

3. 如果 doAfterBody 方法被调用，并且该方法返回 EVAL_BODY_AGAIN，Web 容器会再次执行标签体的内容；如果 doAfterBody 方法返回 SKIP_BODY，Web 容器会调用标签类的 doEndTag 方法。

4. 如果 doEndTag 方法返回 EVAL_PAGE，Web 容器会执行自定义标签后面的内容；如果 doEndTag 方法返回 SKIP_PAGE，Web 容器会忽略自定义标签后面的内容。

10.1.3　自定义标签 API

自定义标签 API 中除了前面介绍的 Tag 接口和 IterationTag 接口外，还有另外三个核心接口：JspTag、BodyTag 和 SimpleTag。为了简化自定义标签的编程工作，在自定义标签 API 中提供了 TagSupport 类、SimpleTagSupport 类和 BodyTagSupport 类。其中 TagSupport 类实现了 IterationTag 接口，SimpleTagSupport 类实现了 SimpleTag 接口，BodyTagSupport 类是 TagSupport 的子类，并实现了 BodyTag 接口。上述五个接口和三个类的关系如图 10.2 所示。

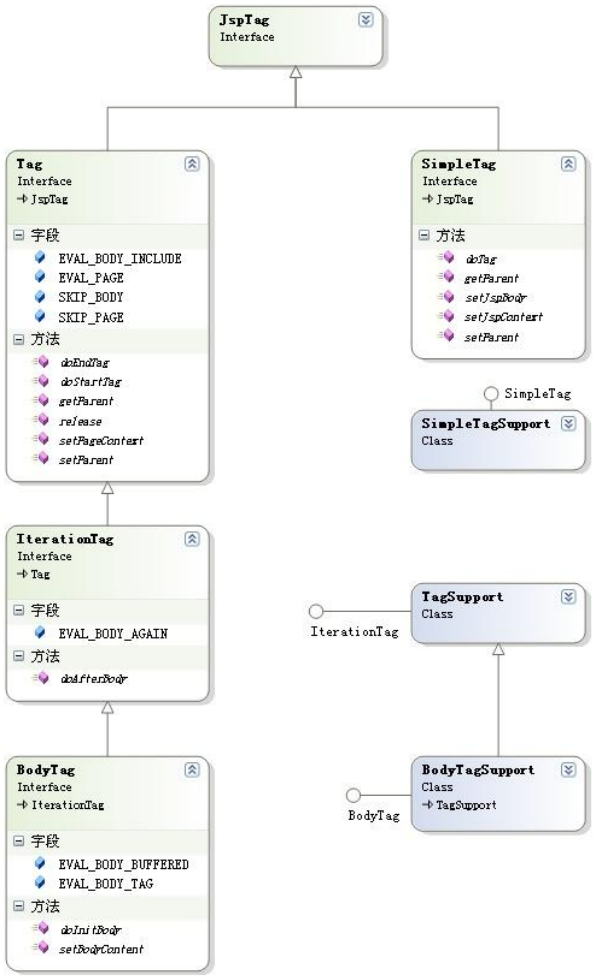

图 10.2　自定义标签 API 的核心接口和类的关系图

上述的接口和类都在 javax.servlet.jsp.tagext 包中，读者可以从如下的网址来查询这些接口和类的详细介绍：

```
    http://java.sun.com/javaee/5/docs/api/javax/servlet/jsp/tagext/package-summary.html
```

访问上面的网址将得到如图 10.3 所示的页面。

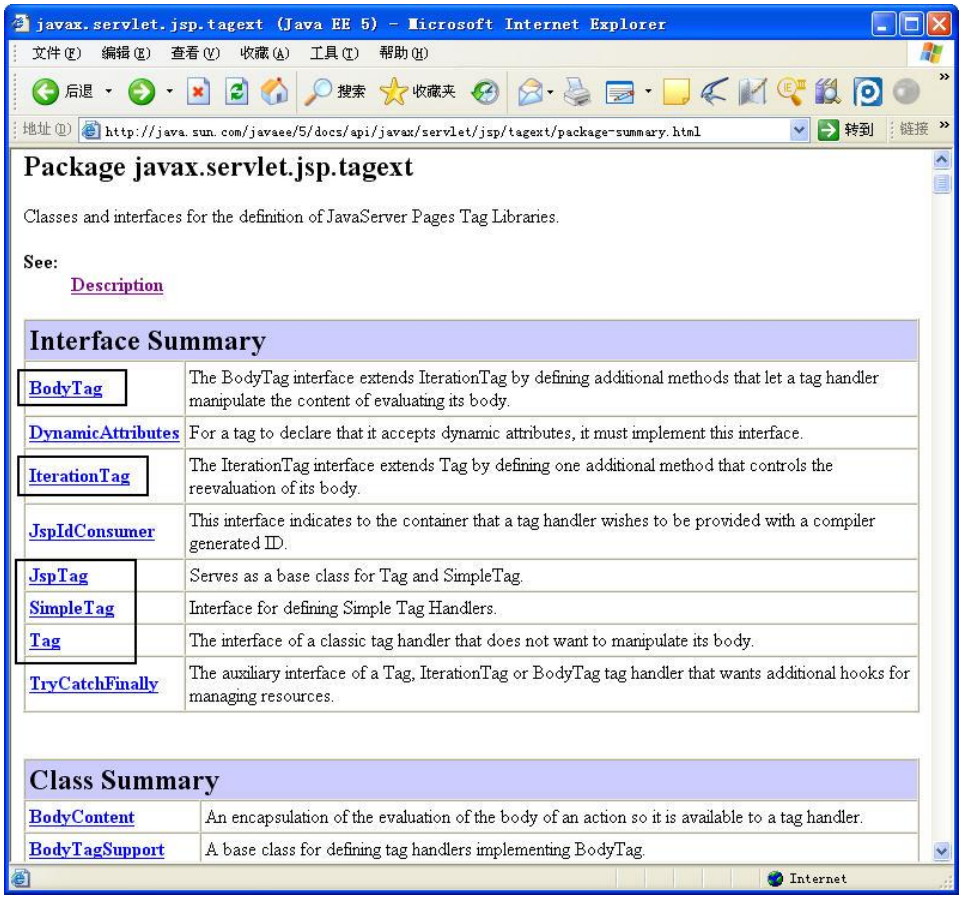

图 10.3　自定义标签 API 的官方文档

上述五个接口的功能和作用如下：

1. JspTag 接口

JspTag 接口是所有自定义标签的父接口。它没有任何属性和方法。Tag 接口和 SimpleTag 接口是 JspTag 的两个直接子接口。其中 JspTag 和 SimpleTag 是 JSP2.0 新增的接口。在 JSP2.0 之前的版本的所有自定义标签的父接口是 Tag。因此，可以将所有实现 Tag 接口的自定义标签称为传统标签，把所有实现 SimpleTag 接口的标签称为简单标签。在本章主要介绍传统标签，简单标签将在下一章详细介绍。

2. Tag 接口

Tag 接口是所有传统标签的父接口。该接口有两个核心方法（doStartTag 和 doEndTag）以及四个常量（EVAL_BODY_INCLUDE、SKIP_BODY、EVAL_PAGE 和 SKIP_PAGE）。其中 doStartTag 方法可以返回 EVAL_BODY_INCLUDE 和 SKIP_BODY，用于控制 Web 容器是否执行标签体的内容；doEndTag 方法可以返回 EVAL_PAGE 和 SKIP_PAGE，用于控制 Web 容器是否执行自定义标签后面的内容。

3. IterationTag 接口

IterationTag 接口继承了 Tag 接口。IterationTag 接口可用于实现需要循环执行标签体内容的自定义标签。在 IterationTag 接口中只有一个 doAfterBody 方法和一个 EVAL_BODY_AGAIN 常量。doAfterBody 方法通

过返回 EVAL_BODY_AGAIN 常量或 Tag 接口中的 SKIP_BODY 常量来控制 Web 容器是否重复执行标签体中的内容。doStartTag 方法、doAfterBody 方法和 doEndTag 方法的调用关系详见 10.1.2 节中的介绍。

并不是每一个自定义标签都需要循环执行标签体的内容，或是控制自定义标签后面的内容是否被执行。因此，JSP API 中提供了一个实现 IterationTag 接口的 TagSupport 类。在 TagSupport 类中对 Tag 接口和 IterationTag 接口中定义的方法都提供了默认的实现。如 doStartTag 方法、doEndTag 方法和 doAfterBody 方法都提供了默认的返回值，代码如下：

```
public class TagSupport implements IterationTag, Serializable
{
   public TagSupport(){ }
   public int doStartTag() throws JspException
   {
      return SKIP_BODY;
   }
   public int doEndTag() throws JspException
   {
      return EVAL_PAGE;
   }
   public int doAfterBody() throws JspException
   {
      return SKIP_BODY;
   }
   //  此处省略了 TagSupport 类中的其他方法和属性
   ... ...
}
```

标签类通过继承 TagSupport 类，就不需要实现 Tag 接口和 Iteration 接口中的每一个方法了，这样将大大简化自定义标签的开发工作。

4. BodyTag 接口

BodyTag 接口继承了 IterationTag 接口。BodyTag 接口不仅拥有 IterationTag 接口的所有功能，而且还可以初始化和修改标签体的内容。在 BodyTag 接口中定义了两个方法（doInitBody 和 setBodyContent）和两个常量（EVAL_BODY_BUFFERED 和 EVAL_BODY_TAG），这两个常量的含义相同。其中 EVAL_BODY_TAG 常量是在 JSP1.2 中的遗留产物，在 JSP 的后续版本中可能不支持该常量，因此，建议使用 EVAL_BODY_BUFFERE 常量。如果 doStartTag 方法返回 EVAL_BODY_BUFFERE，Web 容器就会将标签体的执行结果保存在 BodyContent 对象中，然后 Web 容器在处理标签时会调用标签类的 setBodyContent 方法将 BodyContent 对象传入标签类的对象实例，接下来就可以在标签类的对象实例中处理标签体的执行结果了。

由于 BodyTag 接口及其父接口中定义了很多方法，为了在实现 BodyTag 接口的类中不用再实现所有的方法，JSP API 提供了一个 BodyTagSupport 类，该类是 TagSupport 类的子类，并且实现了 BodyTag 接口。在 BodyTagSupport 类中改变了 doStartTag 方法的默认返回值，并且覆盖了其他的核心方法，代码如下：

```
public class BodyTagSupport extends TagSupport implements BodyTag
{
   protected BodyContent  bodyContent;
   public BodyTagSupport()
   {
      super();
   }
   public void setBodyContent(BodyContent b)
   {
      this.bodyContent = b;
```

```
    }
    public BodyContent getBodyContent()
    {
        return bodyContent;
    }
    //  改变了 doStartTag 方法的默认返回值
    public int doStartTag() throws JspException
    {
        return EVAL_BODY_BUFFERED;
    }
    public int doEndTag() throws JspException
    {
        return super.doEndTag();
    }
    public void doInitBody() throws JspException
    {
    }
    public int doAfterBody() throws JspException
    {
       return SKIP_BODY;
    }
    //  此处省略了 BodyTagSupport 类的其他方法和属性
    ... ...
}
```

从前面的内容可知，自定义标签 API 涉及到了三个方法（doStartTag、doAfterBody 和 doEndTag）以及这三个方法可能返回的六个常量（EVAL_BODY_INCLUDE、EVAL_BODY_BUFFERE、SKIP_BODY、EVAL_BODY_AGAIN、EVAL_PAGE 和 SKIP_PAGE）。表 10.1 给出了这三个方法的返回值和这六个常量的关系。

表 10.1　doStartTag、doAfterBody 和 doEndTag 方法的返回值及其作用

方法名 / 返回值	doStartTag	doAfterBody	doEndTag
EVAL_BODY_INCLUDE	执行标签体中的内容(直接将标签体的执行结果输出到 out 对象的缓冲区中)	*	*
EVAL_BODY_BUFFERE	执行标签体中的内容，并将标签体的执行结果保存在 BodyContent 对象中，以备后续处理和加工	*	*
SKIP_BODY	忽略标签体的内容	不再重复执行标签体的内容	*
EVAL_BODY_AGAIN	*	重复执行标签体的内容	*
EVAL_PAGE	*	*	继续执行自定义标签后面的内容
SKIP_PAGE	*	*	忽略自定义标签后面的内容

5. SimpleTag 接口

SimpleTag 是 JSP2.0 新增的一个接口。该接口只有一个 doTag 方法，这个方法只在 Web 容器执行自定义标签时调用一次。所有的处理逻辑（包括是否执行标签体、重复执行标签体等）都要写在 doTag 方法中，因此，SimpleTag 接口的功能相当于 BodyTag 接口，只是 SimpleTag 接口要比 BodyTag 接口更容易使用。为了更进一步简化自定义标签的开发工作，JSP API 提供了一个实现 SimpleTag 接口的 SimpleTagSupport

类，建议读者在编写简单标签时，标签类从 SimpleTagSupport 类继承。

10.2　实现基本的自定义标签

一个自定义标签最基本的功能就是控制是否执行标签体的内容，以及控制是否执行自定义标签后面的内容。通过实现 Tag 接口的 doStartTag 方法和 doEndTag 方法可以很容易地完成这两个功能。在 Tag 接口中还有定义了一些其他的方法（setPageContext、setParent 等），通过实现这些方法，可以编写更高级的自定义标签。

10.2.1　Tag 接口

javax.servlet.jsp.tagext.Tag 接口是所有传统标签的父接口，该接口定义了 Web 容器处理自定义标签的基本方法。这些方法主要包括 Web 容器执行到标签的开始标记时发生的标签开始事件（调用 Tag 接口的 doStartTag 方法）和执行到标签的结束标记时发生的标签结束事件（调用 Tag 接口的 doEndTag 方法）。在 Tag 接口中定义了四个常量，并由 doStartTag 方法和 doEndTag 方法返回相应的常量以通知 Web 容器如何执行自定义标签。Tag 接口中定义的常量和方法如下所示：

1. Tag 接口中定义的常量

在 Tag 接口中定义了如下四个整型常量：
EVAL_BODY_INCLUDE
SKIP_BODY
EVAL_PAGE
SKIP_PAGE

其中 EVAL_BODY_INCLUDE 和 SKIP_BODY 将作为 doStartTag 方法的返回值返回给 Web 容器，Web 容器根据 doStartTag 方法的返回值来决定是否执行自定义标签的标签体。EVAL_PAGE 和 SKIP_PAGE 将作为 doEndTag 方法的返回值返回给 Web 容器， Web 容器根据 doEndTag 方法的返回值决定是否执行的自定义标签后面的内容。

2. setPageContext 方法

在 Web 容器创建标签类的对象实例后，会首先调用标签类的 setPageContext 方法将 PageContext 对象实例传入标签类的对象实例，然后会调用标签类中的其他方法，这样在标签类中的其他方法就可以使用 PageContext 对象了。setPageContext 方法的定义如下：

```
public void setPageContext(PageContext pageContext)
```

3. setParent 方法和 getParent 方法

Web 容器在调用标签类的 setPageContext 方法后，会调用标签类的 setParent 方法，该方法用来设置当前标签的父标签的对象实例。通过 getParent 标签可以获得当前标签的父标签的对象实例。如果当前标签没有父标签，setParent 方法的参数值为 null。setParent 方法和 getParent 方法的定义如下：

```
public void setParent(Tag t)
public Tag getParent()
```

4. doStartTag 方法

Web 容器在依次调用 setPageContext 方法、setParent 方法以及设置当前标签属性的 setter 方法后，就会调用标签类的 doStartTag 方法。当 Web 容器调用 doStartTag 方法时，就意味着 Web 容器已经开始处理当前标签的开始标记。如果 doStartTag 方法返回 EVAL_BODY_INCLUDE，Web 容器会在执行完标签体后调用标签类的 doEndTag 方法；如果 doEndTag 方法返回 SKIP_BODY，Web 容器并不会执行标签体，而是直接

调用标签类的 doEndTag 方法。doStartTag 方法的定义如下：

```
int doStartTag() throws JspException
```

5. doEndTag 方法

Web 容器在遇到标签的结束标记时，会调用标签类的 doEndTag 方法。如果 doEndTag 方法返回 EVAL_PAGE，Web 容器会继续执行结束标记后面的内容；如果 doEndTag 方法返回 SKIP_PAGE，结束标签后面的内容都不会被执行，也就是说，Web 容器在执行当前 JSP 页面时，执行到标签的结束标记处就会终止执行当前的 JSP 页面。doEndTag 方法的定义如下：

```
int doEndTag() throws JspException
```

6. release 方法

JSP 规范要求 Web 容器必须在垃圾回收器回收标签类的对象实例时调用 release 方法，以便自定义标签可以利用该方法释放所占用的相关资源。但 JSP 规范并没有规定 Web 容器调用 release 方法的具体时间，因此，release 方法的调用时间由具体的 Web 容器厂商决定。release 方法的定义如下：

```
public void release()
```

10.2.2 标签类中方法的调用顺序

在本节给出一个例子来测试标签类中方法的调用顺序。本示例测试了 Web 容器在调用自定义标签时调用标签类中的 setPageContext 方法、setParent 方法、设置标签属性的 setter 方法、doStartTag 方法、doEndTag 方法和 release 方法的顺序。

【实例 10-2】 测试标签类中方法的调用顺序

1. 编写 InvokeOrderTag 类

InvokeOrderTag 是一个标签类，在该类中覆盖了 TagSupport 类中的相关方法，并在这些方法的调用轨迹输出到控制台。InvokeOrderTag 类的实现代码如下：

```
package chapter10;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;
public class InvokeOrderTag extends TagSupport
{
    //  定义了两个属性，用于设置和读取标签的相应属性值
    private String attr1;
    private String attr2;
    //  设置 attr1 属性的值
    public void setAttr1(String attr1)
    {
        System.out.println("setAttr1");
        this.attr1 = attr1;
    }
    //  设置 attr2 属性的值
    public void setAttr2(String attr2)
    {
        System.out.println("setAttr2");
        this.attr2 = attr2;
    }
    @Override
    public void setPageContext(PageContext pageContext)
```

```
    {
        System.out.println("pageContext");
        super.setPageContext(pageContext);
    }
    @Override
    public void setParent(Tag t)
    {
        System.out.println("setParent");
        System.out.print("父标签: ");
        System.out.println(t);
        super.setParent(t);
    }
    public int doStartTag() throws JspException
    {
        System.out.println("doStartTag");
        return super.doStartTag();
    }
    public int doEndTag() throws JspException
    {
        System.out.println("doEndTag");
        return super.doEndTag();
    }
    @Override
    public void release()
    {
        System.out.println("release");
        super.release();
    }
}
```

2. 安装 invokeOrder 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 invokeOrder 标签：

```
<tag>
    <description>测试标签类中方法的调用顺序</description>
    <name>invokeOrder</name>
    <tag-class>chapter10.InvokeOrderTag</tag-class>
    <body-content>empty</body-content>
    <!--  设置 invokeOrder 标签的两个属性  -->
    <attribute>
        <name>attr1</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>attr2</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

3. 编写 invokeorder.jsp 页面

invokeorder.jsp 页面用来调用 invokeOrder 标签。当访问 invokeorder.jsp 页面后，就会在 Tomcat 控制台输出相应的方法调用轨迹。invokeorder.jsp 页面的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
```

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<c:if test="true">
    <!-  调用 invokeOrder 标签  -->
    <ct:invokeOrder attr2="value2" attr1="value1"  />
</c:if>
```

4. 测试 invokeOrder 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/invokeorder.jsp
```

Tomcat 的控制台将输出如下的信息：

```
pageContext
setParent
父标签：org.apache.taglibs.standard.tag.rt.core.IfTag@8c436b
setAttr2
setAttr1
doStartTag
doEndTag
```

从上面的输出信息可知，Web 容器会根据使用标签时属性的位置来调用设置属性值的 setter 方法（在 invokeorder.jsp 页面中调用 invokeOrder 标签时，attr2 属性在 attr1 属性前面）。

10.2.3 控制是否执行标签体

在本节的例子通过 doStartTag 方法的返回值来控制 Web 容器是否执行标签体的内容。

【实例 10-3】 控制是否执行标签体

1. 实例说明

本示例根据是否存在 body 请求参数来决定 doStartTag 方法的返回值。如果存在 body 请求参数，doStartTag 方法返回 EVAL_BODY_INCLUDE，否则返回 SKIP_BODY。

2. 编写 DisplayBodyTag 类

DisplayBodyTag 是一个标签类。该类负责判断是否存在 body 请求参数，并根据判断结果决定 doStartTag 方法的返回值。DisplayBodyTag 类的实现代码如下：

```
package chapter10;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;
public class DisplayBodyTag extends TagSupport
{
    public int doStartTag() throws JspException
    {
        //  获得 body 请求参数的值
        String body = pageContext.getRequest().getParameter("body");
        //  存在 body 请求参数，返回 EVAL_BODY_INCLUDE
        if(body != null)
        {
            return this.EVAL_BODY_INCLUDE;
        }
        //  不存在 body 请求参数，返回 SKIP_BODY
        else
```

```
            {
                return this.SKIP_BODY;
            }
        }
    }
```

3. 安装 displayBody 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 displayBody 标签：

```
<tag>
    <description>显示标签体的内容</description>
    <name>displayBody</name>
    <tag-class>chapter10.DisplayBodyTag</tag-class>
    <body-content>JSP</body-content>
</tag>
```

在上面的配置代码中的<body-content>元素的值是 JSP，该值表示标签体的内容可以是任意 JSP 页面元素，JSP 也是<body-content>元素的默认值。

4. 编写 displaybody.jsp 页面

displaybody.jsp 页面用来调用 displayBody 标签，在 displayBody 的标签体中有一行字符串，如果访问 displaybody.jsp 页面时包含 body 请求参数，则 displayBody 标签会显示这行字符串。displaybody.jsp 页面的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
displayBody 标签体的内容：
<ct:displayBody>
    如果存在请求参数 body，则输出标签体的内容
</ct:displayBody>
```

5. 测试 displayBody 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/displaybody.jsp?body
```

浏览器显示的信息如图 10.4 所示。

图 10.4　访问 displaybody.jsp 页面时带 body 请求参数

如果在访问 displaybody.jsp 页面时不带 body 请求参数，则在浏览器中输出的信息如图 10.5 所示。

图 10.5　访问 displaybody.jsp 页面时不带 body 请求参数

从图 10.5 所示的输出结果可以看出，当访问 displaybody.jsp 页面的 URL 不包含 body 请求时，<ct:displayBody>标签体中的内容并没有输出。

10.2.4　控制是否执行标签后面的内容

在本节的例子通过 doEndTag 方法的返回值来控制 Web 容器是否执行标签后面的内容。

【实例 10-4】　控制是否执行标签后面的内容

1. 实例说明

本示例根据访问 JSP 页面的客户端 IP 地址来决定是否执行 localPage 标签后面的内容。如果 IP 地址为 127.0.0.1，则允许执行 localPage 标签后面的内容，否则，Web 容器只执行到 localPage 标签的结束标记为止。

2. 编写 LocalPageTag 标签

LocalPageTag 是一个标签类，负责验证客户端的 IP 地址，并根据验证结果决定是否执行 localPage 标签后面的内容。LocalPageTag 类的代码如下：

```
package chapter10;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.tagext.TagSupport;
public class LocalPageTag extends TagSupport
{
   public int doEndTag() throws JspException
   {
       //  获得客户端的 IP 地址
       String remoteAddr = pageContext.getRequest().getRemoteAddr();
       //  如果客户端的 IP 地址是 127.0.0.1，则返回 EVAL_PAGE
       if(remoteAddr.equals("127.0.0.1"))
       {
          return this.EVAL_PAGE;
       }
      //  如果是其他的客户端 IP 地址，则返回 SKIP_PAGE
       else
       {
          return this.SKIP_PAGE;
       }
```

```
        }
    }
```

3. 安装 localPage 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 localPage 标签：

```
    <tag>
        <description>
            只有使用本地的 IP（127.0.0.1）或主机名（localhost)才能执行标签后面的内容
        </description>
        <name>localPage</name>
        <tag-class>chapter10.LocalPageTag</tag-class>
        <body-content>empty</body-content>
    </tag>
```

4. 编写 localpage.jsp 页面

localpage.jsp 页面负责调用 localPage 标签。如果访问 localpage.jsp 页面的客户端 IP 地址是 127.0.0.1，则 Web 容器会继续执行 localPage 标签后面的内容，否则，Web 容器将不会执行 localPage 标签后面的内容。localpage.jsp 页面的代码如下：

```
    <%@ page language="java" contentType="text/html; charset=UTF-8"%>
    <%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
    localPage 标签前的内容<br>
    <ct:localPage/>
    localPage 标签后的内容
```

5. 使用本地主机名访问 localpage.jsp 页面

在浏览器地址栏中输入如下的 URL：

```
    http://localhost:8080/demo/chapter10/localpage.jsp
```

浏览器显示的输出结果如图 10.6 所示。

图 10.6　执行 localPage 标签后面的内容

6. 使用远程 IP 访问 localpage.jsp 页面

假设本地的 IP 地址为 192.168.17.127。在浏览器地址栏中输入如下的 URL：

```
    http://192.168.17.127:8080/demo/chapter10/localpage.jsp
```

浏览器显示的输出结果如图 10.7 所示。

图 10.7　未执行 localPage 标签后面的内容

10.2.5　限定自定义标签的父标签

Tag 接口的 getParent 方法可以获得当前标签的父标签的对象实例。利用这个功能可以限定自定义标签的父标签，如限定某一个标签只能在<c:forEach>标签的标签体中使用。

【实例 10-5】　可以生成表格的自定义标签

1. 实例说明

在这个示例中将编写一个可以生成 HTML 表格的标签，该标签只能在<c:forEach>标签的标签体中使用。用于根据<c:forEach>标签的 items 属性指定的集合生成表格的行（<tr>...</tr>）。items 属性指定的集合中的元素必须是 java.util.List 类型。否则该标签将抛出异常。

2. 编写 TRTag 类

TRTag 是一个标签类，负责根据<c:forEach>标签的 items 属性指定的集合元素生成相应的<tr>...</tr>。TRTag 类的代码如下：

```
package chapter10;
import java.io.IOException;
import org.apache.taglibs.standard.tag.rt.core.ForEachTag;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
public class TRTag extends TagSupport
{
    @Override
    public int doStartTag() throws JspException
    {
        //  返回 tr 标签的父标签的对象实例
        Tag tag = getParent();
        //  如果 tr 标签的父标签的对象实例是 ForEachTag，则继续执行下面的操作
        if (tag instanceof ForEachTag)
        {
            ForEachTag forEachTag = (ForEachTag) tag;
            //  获得 forEach 标签当前迭代的元素
            Object current = forEachTag.getLoopStatus().getCurrent();
            //  如果当前迭代的元素为 null，不处理该元素
            if(current == null) return this.SKIP_BODY;
            //  如果当前迭代的元素类型是 java.util.List，继续处理当前迭代的元素
```

```
            if (current instanceof java.util.List)
            {
                java.util.List row = (java.util.List) current;
                //  开始生成<tr>...</tr>
                String tr = "<tr>";
                for (Object cell : row)
                {
                    tr += "<td>" + cell + "</td>";
                }
                tr += "</tr>";
                try
                {
                    //  将生成的<tr>...</tr>输出的客户端
                    pageContext.getOut().write(tr);
                }
                catch (IOException e)
                {
                }
            }
            //  如果当前迭代的元素类型不是 java.util.List，则抛出 JspException 异常
            else
            {
                throw new JspException("当前元素必须是 java.util.List 类型！");
            }
        }
        //  如果 tr 标签的父标签的对象实例不是 ForEachTag，则抛出 JspException 异常
        else
        {
            throw new JspException("continue 标签必须在 forEach 标签体中使用！");
        }
        return this.SKIP_BODY;
    }
}
```

3. 安装 tr 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 tr 标签：

```
<tag>
    <description>生成表格中的 tr 标签</description>
    <name>tr</name>
    <tag-class>chapter10.TRTag</tag-class>
    <body-content>empty</body-content>
</tag>
```

4. 编写 tr.jsp 页面

tr.jsp 页面使用<c:forEach>标签和<ct:tr>标签生成一个表格。其中<c:forEach>标签对一个 java.util.List 对象进行迭代，该对象中的元素也是 java.util.List 对象，每当<c:forEach>标签迭代到一个元素时，就会调用<ct:tr>标签来生成表格行（<tr>...</tr>）。tr.jsp 页面的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%
    java.util.List<java.util.List> rows = new java.util.ArrayList<java.util.List>();
    java.util.List row1 = new java.util.ArrayList();
    java.util.List row2 = new java.util.ArrayList();
```

```
    java.util.List row3 = new java.util.ArrayList();
    row1.add("足球");
    row1.add(30);
    row1.add(12);
    row2.add("跑步机");
    row2.add(3210);
    row2.add(25);
    row3.add("自行车");
    row3.add(221);
    row3.add(6);
    rows.add(row1);
    rows.add(row2);
    rows.add(row3);
    request.setAttribute("rows", rows);
%>
<table border="1">
    <tr>
        <th>商品名称</th>
        <th>单价</th>
        <th>数量</th>
    </tr>
    <c:forEach items="${rows}">
        <!--  生成表格的行代码  -->
        <ct:tr/>
    </c:forEach>
</table>
```

5. 测试 tr 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/tr.jsp
```

浏览器的输出信息如图 10.8 所示。

图 10.8　使用<ct:tr>标签生成表格

如果在<ct:tr>标签的父标签不是<c:forEach>，在调用<ct:tr>标签时就会抛出如图 10.9 所示的异常信息。

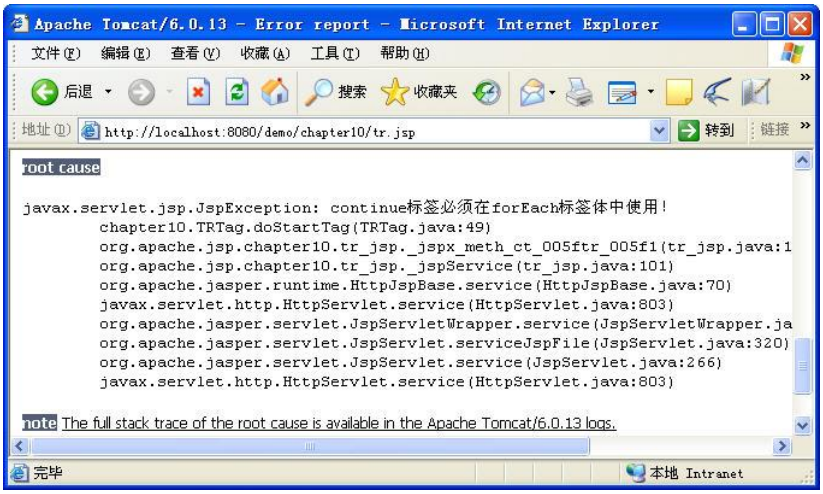

图 10.9　<ct:tr>标签的父标签不是<c:forEach>时抛出的异常信息

10.3　自定义标签的属性

自定义标签可以定义一个或多个属性。在前面的例子已经多次使用到自定义标签的属性。在自定义标签中使用属性的一般形式如下：

```
<prefix:tag attr1 = "value1" attr2 = "value2" ... attrn = "valuen"/>
```

10.3.1　定义标签属性

可按如下两步向自定义标签添加属性：

1. 在标签类中编写和标签属性对应的 setter 方法。
2. 在 TLD 文件中使用<attribute>元素配置每一个标签属性。

假设标签 mytag 有两个属性：uri 和 name。首先需要在 mytag 标签所对应的标签类（MyTag 类）中编写两个 setter 方法（setUri 和 setName），代码如下：

```
public class MyTag extends TagSupport
{
    private String uri;
    private String name;
    public void setUri(String uri)
    {
        this.uri = uri;
    }
    public void setName(String name)
    {
        this.name = name;
    }
    //  此处省略了 MyTag 类的其他属性和方法
    ... ...
}
```

Web 容器在调用 mytag 标签时，如果发现 MyTag 类中有 setUri 方法和 setName 方法，会自动将 mytag 标签的 uri 属性和 name 属性的值分别通过 setUri 方法和 setName 方法传入 MyTag 类的对象实例，这些在 MyTag 类中就可以直接使用 uri 属性和 name 属性的值了。如下面的调用 mytag 标签的代码将“/uri-tag”和

“test”分别作为 MyTag 对象的 uri 属性和 name 属性值传入 MyTag 对象。

```
<prefix:mytag uri="/uri-tag" name="test"/>
```

如果和某个标签属性对应的 setter 方法需要传递非字符串类型的参数值，如 int、double、boolean 等，Web 容器会调用相应类的 valueOf 方法进行类型转换。假设 doubleTag 标签有一个 value 属性（该属性的值是“12.34”），与 value 属性对应的 setter 方法的参数类型是 double。Web 容器在调用 setter 方法时，会使用 Double.valueOf("12.34")方法将字符串“12.34”转换成双精度浮点数 12.34。Java 中的所有基本类型（byte、short、int、long、float、double、boolean）都支持这种转换机制。

除了在标签类中编写相应的 setter 方法外，还需要在 TLD 文件中使用<attribute>元素来配置这些属性，配置代码如下：

```
<tag>
    <description>带属性的标签</description>
    <name>mytag</name>
    <tag-class>MyTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>uri</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>name</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

上面代码中的黑体字部分配置了 uri 属性和 name 属性。<attribute>元素的所有子元素的含义如表 10.2 所示。

表 10.2　<attribute>元素的所有子元素的含义

元素名	是否必须指定	含义
name	是	用于设置标签属性的名称。属性名称是大小写敏感的。
required	否	用于指定当前标签属性是否必须指定。默认值是 false。如果该元素的值为 true，表示在使用自定义标签时必须指定当前属性；如果该元素的值为 false，表示在使用自定义标签时可以不指定当前属性。
rtexprvalue	否	rtexprvalue 是 runtime expression value（运行时表达式值）的英文缩写，用于指定当前属性是否支持动态属性值。默认值是 false。如果该元素值设为 true，表示属性值可以是 JSP 表达式或 EL，否则，属性值只能是字符串。
type	否	用于指定标签属性值对应的 Java 数据类型。
description	否	用于指定属性的描述信息。

10.3.2　编写数学运算标签

在本节给出一个数学运算标签(math 标签）的例子。math 标签可以调用 java.lang.Math 类中所有只含有一个 double 类型参数的静态方法（如 sin、cos、tan、sqrt 等）。math 标签有如下四个属性：

method（必选）：指定 Math 类中的静态方法名。

value（必选）：指定要调用的静态方法的参数值。该属性值必须可转换成 double 数据类型。

angle（可选）：如果调用的是三角函数，该属性指定 value 属性的值是否表示角度。默认值是 false。如果该属性值为 true，表示 value 属性的值是弧度。

pattern（可选）：指定格式化静态方法返回值的模式字符串。默认值是“0.00”。模式字符串的格式和 DecimalFormat 类所使用的模式字符串格式相同，详见 8.2.12 节的内容。

【实例 10-6】 数学运算标签

1. 编写 MathTag 类

MathTag 是一个标签类，负责读取 math 标签的属性值，并调用相应的静态方法进行数学运算，MathTag 类的代码如下：

```
package chapter10;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
public class MathTag extends TagSupport
{
    //  下面四个属性和 math 标签的四个属性相对应
    private String method;
    private double value;
    private String pattern = "0.00";
    private boolean angle = false;
    public void setMethod(String method)
    {
        this.method = method;
    }
    public void setValue(double value)
    {
        this.value = value;
    }
    public void setangle(boolean angle)
    {
        this.angle= angle;
    }
    public void setPattern(String pattern)
    {
        this.pattern = pattern;
    }
    public int doStartTag() throws JspException
    {
        try
        {
            //  获得指定静态方法的 Method 对象
            java.lang.reflect.Method m = Math.class.getMethod(method,
                   new Class[]{ double.class });
            java.text.NumberFormat nf =java.text.NumberFormat.getNumberInstance();
            java.text.DecimalFormat df = (java.text.DecimalFormat )nf;
            //  应用模式字符串
            df.applyPattern(pattern);
            //  如果 angle 属性值为 true（value 属性值表示角度），将 value 属性值转换成弧度
            if(angle)
                value = Math.toRadians(value);
            //  动态调用 Math 类指定的静态方法，并格式化返回值
            String result =  df.format(m.invoke(null, value));
            //  将格式化后的结果输出的客户端
            pageContext.getOut().println(result);
```

```
            }
            catch (Exception e)
            {
                throw  new JspException(e.getMessage());
            }
            return super.doStartTag();
        }
    }
```

2. 安装 math 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 math 标签：

```
<tag>
    <description>
        调用 Math 类中的静态数学方法（只调用有一个 double 类型参数的静态方法）
    </description>
    <name>math</name>
    <tag-class>chapter10.MathTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>method</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>value</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>pattern</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>angle</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

3. 编写 math.jsp 页面

math.jsp 页面通过 math 标签调用了 Math 类中的 sin、cos、tan、sqrt 和 log 方法进行相应的数学运算。math.jsp 页面的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
sin(π) = <ct:math value="3.14" method="sin" />    
cos(π) = <ct:math value="3.14" method="cos" /><br>
tan(1.45) = <ct:math value="1.45" method="tan" /><br>
sin(90°) = <ct:math value="90" method="sin" angle="true" /><br>
sin(50°) = <ct:math value="50" method="sin" angle="true" pattern="0.0000" /><br>
tan(39°) = <ct:math value="39" method="tan" angle="true" pattern="0.00000" /><br>
sqrt(25) =  <ct:math value="25" method="sqrt" pattern="0.0" /><br>
log(221) =  <ct:math value="221" method="log" pattern="0.0000" />（以 e 为底的对数）
```

4. 测试 math 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/math.jsp
```

浏览器的输出结果如图 10.10 所示。

图 10.10　使用 math 标签进行数学运算

10.3.3　动态属性值

如果在配置标签属性时将 rtexprvalue 元素值设为 true，则当前的标签属性值可以是 JSP 表达式或 EL。例如，将 10.3.2 节中的数学运算标签的四个属性的 rtexprvalue 元素值都设为 true，配置代码如下：

```
<tag>
    <description>支持动态属性值的数学运算标签</description>
    <name>dynMath</name>
    <tag-class>chapter10.MathTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>method</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>value</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>pattern</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>angle</name>
        <required>false</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
```

```
    </tag>
```

在上面的配置代码中将<trexprvalue>元素的值设为 true，并且为数学运算标签重新设置了一个标签名：dynMath。下面的代码通过 EL 来设置 dynMath 标签的属性值：

```
    <ct:dynMath method="${param.method}"  value="${param.value}" />
```

如果使用 JSP 表达式或 EL 作为标签属性的值，Web 容器不会对 JSP 表达式或 EL 返回的值进行类型转换，因此，JSP 表达式或 EL 的返回值类型必须与属性值的类型相同，否则 Java 编译器会报告编译错误。

dynmath.jsp 页面通过四个表单来提交和 dynMath 标签的属性相对应的请求参数，同时，dynMath 标签从请求参数的获得相应的属性值，并输出计算结果。dynmath.jsp 页面的代码如下：

```
    <%@ page language="java" contentType="text/html; charset=UTF-8"%>
    <%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
    <%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
    <!--  如果 method 和 value 请求参数不为 null，调用 dynMath 标签进行数学运算  -->
    <c:if test="${param.method != null && param.value != null}">
       ${param.method}(${param.value}${param.angle=='true'?'°':''})=
       <ct:dynMath value="${param.value}"
                   method="${param.method}"
                   <!--  如果 angle 请求参数为空字符串，使用默认值 false  -->
                   angle="${(param.angle == '')?'false':param.angle}"
                   <!--  如果 pattern 请求参数为空字符串，使用默认值 0.00  -->
                   pattern="${param.pattern == ''?'0.00':param.pattern}" />
    </c:if>
    <form method="post">
       <table>
          <tr>
             <td>方法名：</td>
             <td><input type="text" name="method" value="${param.method}" /></td>
          </tr>
          <tr>
             <td>参数值：</td>
             <td><input type="text" name="value" value="${param.value}" /></td>
          </tr>
          <tr>
             <td>是否为角度：</td>
             <td><input type="checkbox" name="angle" value="true"
                ${param.angle == 'true'?"checked='checked'":''} />
             </td>
          </tr>
          <tr>
              <td>模式字符串：</td>
              <td><input type="text" name="pattern"
                         value="${param.pattern}" />
              </td>
          </tr>
       </table>
       <p/><input type="submit" value="计算" />
    </form>
```

在浏览器地址栏中输入如下的 URL：

```
    http://localhost:8080/demo/chapter10/dynmath.jsp
```

在页面中的【方法名】文本框中输入“sin”，在【参数值】文本框中输入“59”，选中【是否为角度】复选框，在【模式字符串】文本框中输入“0.000”，单击【计算】按钮，页面显示的效果如图 10.11 所示。

图 10.11　使用支持动态属性值的 dynMath 标签进行数学运算

10.3.4　标签的动态属性

JSP 规范允许标签中的属性不需要在 TLD 文件中定义就可以使用，这种属性称为动态属性。使用动态属性的好处是可以根据需要为自定义标签添加相应的属性，而无需修改 TLD 文件，这无疑会大大增强自定义标签的灵活性。为自定义标签增加动态属性功能需要如下两步：

1. 标签类需要实现 javax.servlet.jsp.tagext.DynamicAttributes 接口。
2. 在 TLD 文件中定义标签时需要使用<dynamic-attributes>元素打开标签的动态属性功能。

在 DynamicAttributes 接口只定义了一个 setDynamicAttribute 方法，该方法的定义如下：

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

其中 uri 参数表示属性的命名空间，如果属性在默认的命名空间中，该参数值为 null，localName 参数表示动态属性的名称，value 参数表示动态属性的值。

在使用标签的动态属性功能时应注意如下几点：

标签中未使用<attribute>元素定义的属性都是动态属性，Web 容器每遇到一个动态属性，就会调用一次 setDynamicAttribute 方法。如果想在标签类中使用这些动态属性，通常将这些动态属性添加到 Map 对象中，再在其他的方法（如 doStartTag、doEndTag 等）中处理这些动态属性。

如果在标签中使用动态属性，必须将<dynamic-attributes>元素的值设为 true。<dynamic-attributes>元素的默认值是 false。

如果<dynamic-attributes>元素的值为 true，标签类必须实现 DynamicAttributes 接口，否则在调用标签时会抛出异常。

10.3.5　使用动态属性生成不同风格的列表框

使用标签的动态属性可以实现很多非常灵活的功能。可以将动态属性的名称和属性值看作是 key-value 对（属性名称相当于 key，属性值相当于 value）。根据动态属性的这个特点可以生成由 key-value 对组成的 HTML 代码。如 HTML 列表框，代码如下：

```
<select>
```

```
    <option value="key">value</option>
    ...
</select>
```

【实例 10-7】 使用动态属性生成不同风格的列表框

1. 实例说明

本示例编写的标签（options 标签）可以使用动态属性生成下拉列表框和可以显示多个选项的列表框。options 标签除了动态属性外，还有如下几个在 TLD 文件中定义的属性：

name：指定生成的列表框的 name 属性值。

isMultiple：指定生成的列表框是下拉列表框，还是可以显示多个选项的列表框。默认值为 false，表示生成的是下拉列表框。

style：指定生成的列表框的 CSS 样式属性值，也就是<select>元素的 style 属性值。

2. 编写 OptionsTag 类

OptionsTag 是一个标签类，负责根据在 TLD 文件中定义的属性以及动态属性生成不同风格的列表框。OptionsTag 类的代码如下：

```
package chapter10;

import java.io.IOException;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
import javax.servlet.jsp.tagext.DynamicAttributes;
import java.util.Map;
import java.util.Set;
import java.util.HashMap;

public class OptionsTag extends TagSupport implements DynamicAttributes
{
   private String name;
   private boolean isMultiple = false;
   private String style;
   private Map<String, String> dynAttributes = new HashMap<String, String>();
   public void setName(String name)
   {
      this.name = name;
   }
   public void setIsMultiple(boolean isMultiple)
   {
      this.isMultiple = isMultiple;
   }
   public void setStyle(String style)
   {
      this.style = style;
   }
   //  实现处理动态属性的 setDynamicAttribute 方法
   public void setDynamicAttribute(String uri, String localName, Object value)
          throws JspException
   {
      //  将所有的动态属性名称和属性值保存在 Map 对象中
      dynAttributes.put(localName, value.toString());
   }
   @Override
   public int doStartTag() throws JspException
```

```
    {
        try
        {
            //  获得动态属性名称集合
            Set<String> keys = dynAttributes.keySet();
            //  生成<select>元素的开始标记
            String html = "<select name = '" + name + "' style='" + style + "'" +
            ((isMultiple == true) ? "multiple='multiple'" : "") + ">";
            //  根据动态属性名称查找动态属性值，并根据每一个动态属性生成<option>元素
            for (String key : keys)
            {
                String value = dynAttributes.get(key);
                html += "<option value='" + key + "'>" + value + "</option>";
            }
            html += "</select>";
            //  将生成的 select 元素的完整 HTML 代码发送到客户端
            pageContext.getOut().println(html);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        return this.SKIP_BODY;
    }

}
```

3. 安装 options 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 options 标签：

```
<tag>
    <description>使用动态属性值和动态属性生成不同类型的选项列表</description>
    <name>options</name>
    <tag-class>chapter10.OptionsTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
        <name>name</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>isMultiple</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>style</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <!--  打开 options 标签的动态属性功能  -->
    <dynamic-attributes>true</dynamic-attributes>
</tag>
```

4. 编写 options.jsp 页面

options.jsp 页面使用 options 标签生成了两种列表框，代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
<center>
    选择你的爱好
    <br>
    <ct:options name="hobby" tennis="网球" internet="上网" tour="旅游"
                reading="阅读" style="width:100px" />
    <hr>
    选择你最喜欢的科幻电影：<br>
    <ct:options name="movie" movie1="独立日"
                             movie2="变形金刚（真人版）"
                             movie3="黑超特警组"
                             movie4="回到未来三步曲"
                             movie5="火星任务"
                             movie6="人工智能"
                             movie7="其他"
                             isMultiple="true"
                             style="width:150px;height:200px" />
</center>
```

5. 测试 options 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/options.jsp
```

浏览器显示的效果如图 10.12 所示。

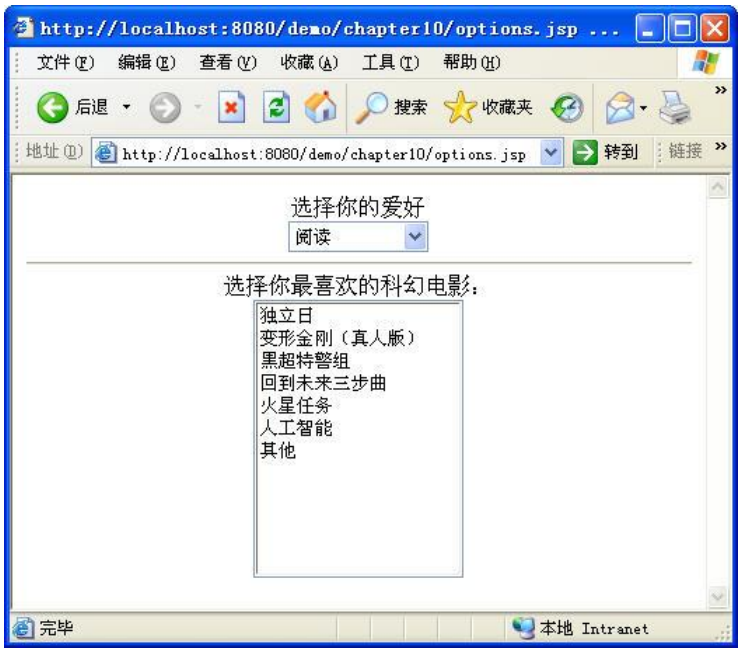

图 10.12　使用 options 标签生成列表框

如果读者查询 options.jsp 页面生成的源代码，可以看到如下的内容：

```
<center>
    选择你的爱好
```

```
        <br>
        <select name = 'hobby' style='width:100px'>
            <option value='reading'>阅读</option>
            <option value='tour'>旅游</option>
            <option value='tennis'>网球</option>
            <option value='internet'>上网</option>
        </select>
        <hr>
        选择你最喜欢的科幻电影：<br>
        <select name = 'movie' style='width:150px;height:200px'multiple='multiple'>
            <option value='movie1'>独立日</option>
            <option value='movie2'>变形金刚（真人版）</option>
            <option value='movie3'>黑超特警组</option>
            <option value='movie4'>回到未来三步曲</option>
            <option value='movie5'>火星任务</option>
            <option value='movie6'>人工智能</option>
            <option value='movie7'>其他</option>
        </select>
    </center>
```

从上面的 HTML 代码可以看出，动态属性名称就是<option>元素的 value 属性值，动态属性值就是<option>元素本身的值。

10.4. 迭代标签

在实际应用中经常要使用自定义标签来迭代输出数组或集合对象中的元素。为此，JSP 规范专门提供了一个 IterationTag 接口来解决这个问题。IterationTag 是 Tag 接口的子接口。所有实现 IterationTag 接口的标签类都可以循环执行标签体中的内容。

10.4.1 IterationTag 接口

IterationTag 接口在 Tag 接口的基础上增加了一个 doAfterBody 方法。如果该方法返回 EVAL_BODY_AGAIN，Web 容器会再次执行标签体的内容；如果 doAfterBody 方法返回 SKIP_BODY，Web 容器将不再执行标签体的内容。Web 容器每执行完一次标签体，就会调用一次 doAfterBody 方法，而 doStartTag 方法只在 Web 容器遇到标签的开始标记时执行一次。因此，可以利用 doStartTag 方法和 doAfterBody 方法的调用方式来完成迭代的初始化和移动到下一个迭代元素的工作。

如果想让标签体中的代码可以访问当前迭代的元素，需要将当前迭代的元素保存在 Web 域中（通过为 page 域），然后使用 EL 或 JSP 表达式从 Web 域中取得当前迭代的元素。当 Web 容器调用 doStartTag 方法时，需要将迭代集合的第一个元素保存在 page 域中，并且 doStartTag 方法要返回 EVAL_BODY_INCLUDE，使得 Web 容器执行标签体中的内容，并且调用 doAfterBody 方法。在 doAfterBody 方法中需要对迭代集合的当前索引增加指定的步长，如果当前索引超过迭代集合的最后一个元素的索引或指定的最大索引，doAfterBody 方法返回 SKIP_BODY（不再执行标签体的内容），否则返回 EVAL_BODY_AGAIN（再次执行标签体的内容）。

10.4.2 编写迭代集合的标签

在本节将实现一个标签（iterator 标签）来模拟 JSTL 中的<c:forEach>标签的功能。iterator 标签有如下

几个属性：

items：指定要迭代的集合对象。
var：指定保存在 page 域中的当前迭代元素的属性名。
begin：指定迭代的开始索引。
end：指定迭代的最大索引。
step：指定迭代的步长。
index：指定保存在 page 域中的迭代索引的属性名。
iterator 标签在如下的几种情况不对集合进行迭代：
集合对象为 null。
集合对象中没有元素（size 为 0）。
end 属性值小于 begin 属性值。
begin 属性值大于或等于集合中的元素个数。

【实例 10-8】 编写迭代集合的标签

1. 编写 IteratorTag 类

IteratorTag 是一个标签类，负责对集合对象进行迭代。该类的代码如下：

```
package chapter10;
import java.io.IOException;
import java.util.List;
import javax.servlet.jsp.tagext.Tag;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.TagSupport;
public class IteratorTag extends TagSupport
{
   private List items = null;
   private String var;
   private int begin = 0;
   private int end = Integer.MAX_VALUE;
   private int step = 1;
   private String count = null;
   //  i 变量表示当前已经迭代的集合元数个数
   private int i = 1;
   public void setItems(List items)
   {
      this.items = items;
   }
   public void setVar(String var)
   {
      this.var = var;
   }
   public void setBegin(int begin)
   {
      this.begin = begin;
   }
   public void setEnd(int end)
   {
      this.end = end;
   }
   public void setStep(int step)
   {
      this.step = step;
   }
```

```
    public void setCount(String count)
    {
        this.count = count;
    }
    @Override
    public int doStartTag() throws JspException
    {

        if (items != null && items.size() > 0 && begin < items.size()
                && begin <= end)
        {
            //  将集合的第一个元素保存在 page 域中
            pageContext.setAttribute(var, items.get(begin));
            //  如果 count 属性不为 null，将当前已经迭代的集合元素个数保存在 page 域中
            if(count != null)
                pageContext.setAttribute(count, i);
            begin += step;
            i++;
            return this.EVAL_BODY_INCLUDE;
        }
        else
            return this.SKIP_BODY;
    }

    @Override
    public int doAfterBody() throws JspException
    {
        if (begin < items.size() && begin < end)
        {
            //  将下一个集合元素保存在 page 域中
            pageContext.setAttribute(var, items.get(begin));
            //  将当前已经迭代的集合元素个数保存在 page 域中
            pageContext.setAttribute(count, i);
            i++;
            begin += step;
            return this.EVAL_BODY_AGAIN;
        }
        else
        {
            //  重新初始化 begin 和 i 变量
            begin = 0;
            i = 0;
            return this.SKIP_BODY;
        }
    }
}
```

由于标签缓存的原因，需要对 begin 和 i 变量进行初始化。这是因为 begin 和 i 变量在 IteratorTag 类中被改变了，如果不进行初始化，下次再调用 iterator 标签时，Web 容器会从标签缓存中获得 IteratorTag 对象实例，如果在 iterator 标签中不指定 begin 属性，begin 和 i 变量变会保持上一次调用 iterator 标签的结果，在这种情况也，iterator 标签就不会按着预定的方式进行迭代了。读者可以将重新初始化 begin 和 i 变量的代码注释掉，看看会发生什么情况。

2. 安装 iterator 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 iterator 标签：

```
<tag>
    <description>迭代集合的标签</description>
    <name>iterator</name>
    <tag-class>chapter10.IteratorTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>items</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    <attribute>
        <name>var</name>
        <required>true</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>begin</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>end</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>step</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>count</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

3. 编写 iterator.jsp 页面

iterator.jsp 页面使用 iterator 标签生成了两个列表框。iterator.jsp 页面的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
<%
    java.util.List<String> teleplays = new java.util.ArrayList<String>();
    teleplays.add("李小龙传奇");
    teleplays.add("绝密押运");
    teleplays.add("神探狄仁杰 3");
    teleplays.add("新大秦帝国");
    teleplays.add("士兵突击");
    request.setAttribute("teleplays", teleplays);
%>
<center>
    国产电视剧<p/>
    <select name="teleplays1" multiple="multiple"
        style="width: 120px; height: 150px">
        <ct:iterator items="${teleplays}" var="teleplay" count="count">
            <option value="${count}">${teleplay}</option>
```

```
        </ct:iterator>
    </select>
                
    <select name="teleplays2" multiple="multiple"
            style="width: 120px; height: 150px">
        <ct:iterator items="${teleplays}" var="teleplay" count="count" begin="1"
                     end="4" step="2">
            <option value="${count}">${teleplay}</option>
        </ct:iterator>
    </select>
</center>
```

4. 测试 iterator 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/iterator.jsp
```

浏览器输出的效果如图 10.13 所示。

图 10.13　使用 iterator 标签生成列表框

使用 iterator 标签的列表框的 HTML 代码如下：

```
<center>
    国产电视剧<p/>
    <select name="teleplays1" multiple="multiple"
            style="width: 120px; height: 150px">
        <option value="1">李小龙传奇</option>
        <option value="2">绝密押运</option>
        <option value="3">神探狄仁杰 3</option>
        <option value="4">新大秦帝国</option>
        <option value="5">士兵突击</option>
    </select>
                
    <select name="teleplays2" multiple="multiple"
            style="width: 120px; height: 150px">
        <option value="1">绝密押运</option>
```

```
            <option value="2">新大秦帝国</option>
        </select>
    </center>
```

10.5 处理标签体内容

在实际应用中，有时也需要修改标签体的执行结果。JSP API 定义了一个 BodyTag 接口，所有实现 BodyTag 接口的标签类都可以修改标签体的执行结果。与 Tag、IterationTag 接口一样，BodyTag 接口也提供了一个默认的实现类 BodyTagSupport，该类实现了 BodyTag 接口中的所有方法，并为有返回值的方法提供了默认返回值。

10.5.1 BodyTag 接口

BodyTag 接口的父接口是 IterationTag。BodyTag 接口在 IterationTag 接口的基础上增加了两个方法和一个常量。这两个方法是 setBodyContent 和 doInitBody，常量是 EVAL_BODY_BUFFERED。如果标签类实现 BodyTag 接口，doStartTag 方法除了可以返回 EVAL_BODY_INCLUDE 和 SKIP_BODY 外，还可以返回 EVAL_BODY_BUFFERED。如果 doStartTag 方法返回 EVAL_BODY_BUFFERED,Web 容器会调用标签类的 setBodyContent 方法将标签体的执行结果作为 javax.servlet.jsp.BodyContent 类的对象实例传入标签类的对象实例，在调用完 setBodyContent 方法后，就会调用 doInitBody 方法。当 doInitBody 方法执行完后，Web 容器将继续执行标签体的内容，并将标签体的执行结果保存在 BodyContent 对象中，然后调用 doAfterBody 方法。如果 doAfterBody 方法返回 EVAL_BODY_AGAIN，Web 容器会再次执行标签体，并将执行结果保存在 BodyContent 对象中，直到 doAfterBody 方法返回 SKIP_BODY，Web 容器才会调用 doEndTag 方法。

BodyTag 接口中定义的两个方法的详细描述如下：

setBodyContent

setBodyContent 方法用于将 Web 容器创建的 BodyContent 对象传入标签类的对象实例，该方法的定义如下：

```
public void setBodyContent(BodyContent b)
```

doInitBody

doInitBody 方法在 setBodyContent 方法之后调用。通常在该方法中初始化 BodyContent 对象的内容。如果在 doInitBody 方法中向 BodyContent 对象中写入一些内容，这些内容将出现在标签体执行结果的前面。doInitBody 方法的定义如下：

```
public void doInitBody() throws JspException
```

Web 容器调用自定义标签的流程如图 10.14 所示。

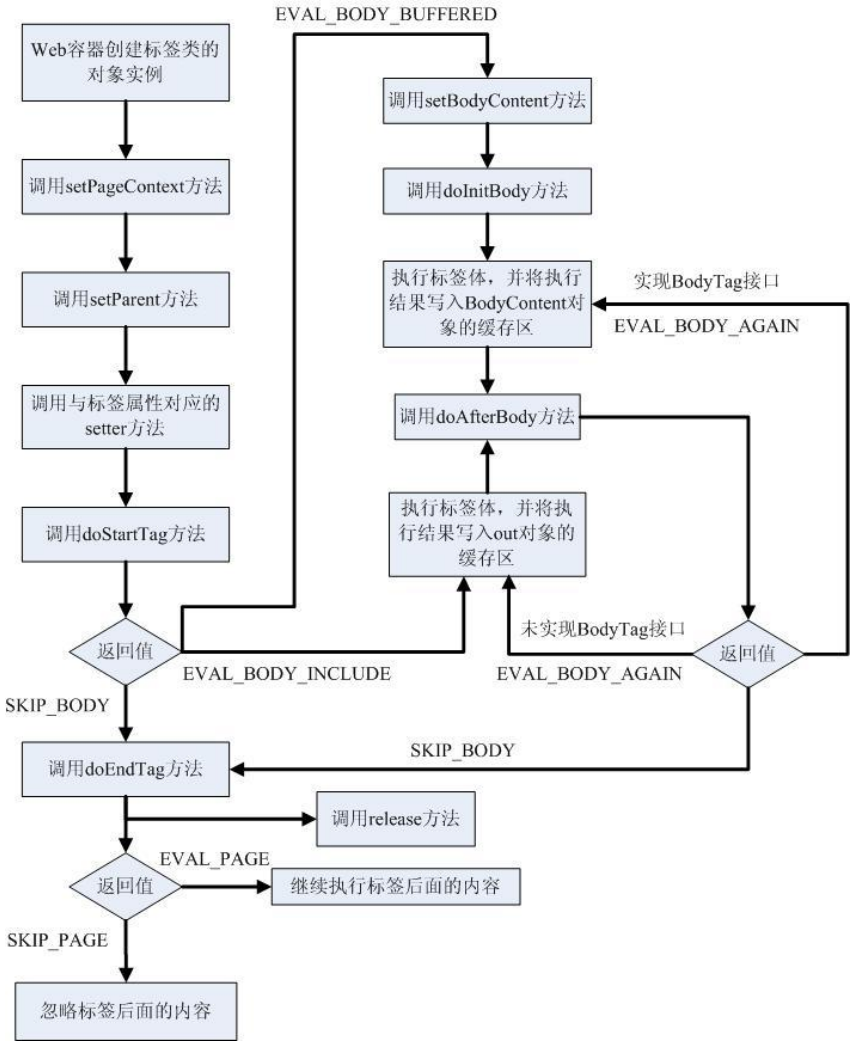

图 10.14　Web 容器调用自定义标签的流程图

10.5.2　BodyContent 类

BodyContent 是 JspWriter 类的子类。BodyContent 类在 JspWriter 类的基础上增加了保存标签体执行结果的缓冲区。在 BodyContent 类中还定义了一些用于操作缓冲区的方法，这些方法可以将缓冲区中的内容转换成字符串或返回封装缓冲区内容的 java.io.Reader 对象，也可以将缓冲区中的内容写入其他的数据流。

PageContext 类中有一个用于表示 JspWriter 对象的 out 变量，通过 PageContext 类的 getOut 方法可以返回 out 变量的值。如果 doStartTag 方法返回 EVAL_BODY_BUFFERED，Web 容器会调用 PageContext 类的 pushBody 方法产生一个 BodyContent 对象，在产生 BodyContent 对象后，会将 out 变量原来指向的 JspWriter 对象压栈，并使 out 变量重新指向新产生的 BodyContent 对象。当 Web 容器遇到标签结束标记后，就会调用 doEndTag 方法，在调用完 doEndTag 方法后，Web 容器会调用 PageContext 类的 popBody 方法将 out 变量原来的 JspWriter 对象从堆栈中弹出，并使 out 变量重新指向这个 JspWriter 对象。因此，在标签后面使用 PageContext.getOut 方法返回的是 JspWriter 对象。

BodyContent 类常用方法的定义如下：

1. getString 方法

getString 方法用于以字符串形式返回 BodyContent 对象缓冲区中保存的内容，该方法的定义如下：

```
public abstract String getString()
```

2. getReader 方法

getReader 方法用于返回一个可以读取 BodyContent 对象缓冲区内容的 java.io.Reader 对象。该方法的定义如下：

```
public abstract Reader getReader()
```

3. getEnclosingWriter 方法

getEnclosingWriter 方法用于返回一个 JspWriter 对象。通过该对象可以向 BodyContent 对象缓冲区写入内容，这个 JspWriter 对象是在 Web 容器创建 BodyContent 对象时通过 BodyContent 类的构造方法传入 BodyContent 对象的。getEnclosingWriter 方法的定义如下：

```
public JspWriter getEnclosingWriter()
```

4. clearBody 方法

clearBody 方法用于清空 BodyContent 对象缓冲区中的内容。该方法的定义如下：

```
public void clearBody()
```

5. writeOut 方法

writeOut 方法用于将 BodyContent 对象缓冲区中的内容写入指定的 java.io.Writer 对象中。writeOut 方法的定义如下：

```
public abstract void writeOut(Writer out) throws IOException
```

10.5.3 编写将 URL 转换成 a 元素的标签

本节要实现的自定义标签可以将标签体的内容（一个 URL）转换成如下成 a 元素，如下面的代码所示：

```
<a href="http://nokiaguy.blogjava.net">http://nokiaguy.blogjava.net</a>
```

其中“http://nokiaguy.blogjava.net”就是标签体的内容。

【实例 10-9】 实现将 URL 转换成 a 元素的自定义标签

1. 程序说明

本示例的标签（url 标签）类需要继承 BodyTagSupport 类。在 doEndTag 方法中以字符串形式获得 BodyContent 对象缓冲区中的内容（一个 URL），并将该 URL 转换成 a 元素，再使用 JspWriter 类的 write 方法将转换后的内容输出到客户端。url 标签有一个 style 属性，用于设置 a 元素的 style 属性。

2. 编写 UrlTag 类

UrlTag 是一个标签类，负责将标签体中的 URL（标签体中只有一个 URL）转换成 a 元素。UrlTag 类的代码如下：

```
package chapter10;
import java.io.IOException;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyTagSupport;
public class UrlTag extends BodyTagSupport
{
   private String style = null;
```

```
    public void setStyle(String style)
    {
        this.style = style;
    }
    @Override
    public int doAfterBody() throws JspException
    {

        //  以字符串形式获得 BodyContent 对象缓冲区中的内容
        String url = bodyContent.getString();
        //  将从 BodyContent 对象缓冲区中获得的 URL 转换成 a 标签，
        //  并根据 style 属性的值为 a 元素添加 style 属性
        String html = "<a href='" + url + "' "
               + ((style == null) ? "" : "style='" + style + "'") + ">" + url
               + "</a>";
        JspWriter out = bodyContent.getEnclosingWriter();
        //  也可使用如下的代码来获得 JspWriter 对象，并将 a 元素输出的客户端
        //  JspWriter out = pageContext.getOut();
        try
        {
            out.write(html);          //  将生成的 a 元素输出的客户端
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        return this.SKIP_BODY;
    }
}
```

在编写上面代码时应注意如下几点：

由于 setPageContext 方法、setParent 方法、属性的 setter 方法和 doStartTag 方法在 setBodyContent 方法之前被调用，因此，不能在这些方法中使用 BodyContent 对象。也就是说，只能在 setBodyContent 方法后面被调用的方法中才能使用 BodyContent 对象，如 doInitBody、doAfterBody 和 doEndTag 方法。

如果要修改标签体的内容，只能在标签体执行完后再处理，也就是说，只能在 doAfterBody 或 doEndTag 方法中编写处理代码。在本例中修改标签体内容的代码被写在了 doAfterBody 方法中。

在 doAfterBody 方法中使用了 bodyContent 变量。该变量是在 BodyTagSupport 类中定义的，并在 BodyTagSupport 类的 setBodyContent 方法中为 bodyContent 赋值，因此，可以在 doInitbody、doAfterBody 和 doEndTag 方法中使用 bodyContent 变量。

3. 安装 url 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 url 标签：

```
<tag>
    <description>将 URL 转换成 a 元素</description>
    <name>url</name>
    <tag-class>chapter10.UrlTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>style</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

4. 编写 url.jsp 页面

url.jsp 页面使用 url 标签将标签体中的 URL 转换成 a 元素，并设置了 a 元素的 style 属性值。url.jsp 页面的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
<ct:url>
   http://nokiaguy.blogjava.net
</ct:url>
<hr>
<!--  设置 a 元素的 style 属性值  -->
<ct:url style="font-size:25px">
   http://nokiaguy.cnblogs.com
</ct:url>
```

5. 测试 url 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/url.jsp
```

浏览器显示的效果如图 10.15 所示。

图 10.15　使用 url 标签将 URL 转换成 a 元素

10.5.4　编写将文本内容转换成表格的标签

在本节给出一个更复杂的自定义标签（text2Table 标签）的例子。text2Table 标签可以将文本的内容转换成表格，列使用空格、tab 等字符分隔，而行使用“\r\n”分隔。text2Table 标签还可以将文本内容中的特殊的字符（如<、>、空格等）转换成字符实体编码（详见表 9.3 的内容）。text2Table 标签有如下几个属性：

isFile：该属性指定标签体的内容是否为一个指定文本文件的相对路径。默认值是 false。如果该属性为 true，text2Table 标签会将该相对路径指定的文本文件的内容转换成表格。

haveHeader：该属性指定是否将表格的第一行设置为表头（使用 th 元素设置）。默认值是 false。

escapeXML：该属性指定是否将文本内容中的特殊字符进行 HTML 编码转换。默认值是 true。

attributes：该属性指定了 table 元素中的所有属性和属性值。也就是说，text2Table 标签会将 attributes 属性值作为<table ... >中间的部分。

1. 编写 Text2TableTag 类

Text2TableTag 是一个标签类，负责将标签体的内容或相对路径指定的文本文件中的内容转换成表格。

Text2TableTag 类的代码如下：

```
package chapter10;
import java.io.IOException;
import java.io.BufferedReader;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.tagext.BodyTagSupport;
public class Text2TableTag extends BodyTagSupport
{
   private boolean isFile = false;
   private boolean haveHeader = false;
   private boolean escapeXML = true;
   private String attributes = null;

   public void setAttributes(String attributes)
   {
      this.attributes = attributes;
   }

   public void setIsFile(boolean isFile)
   {
      this.isFile = isFile;
   }

   public void setEscapeXML(boolean escapeXML)
   {
      this.escapeXML = escapeXML;
   }

   public void setHaveHeader(boolean haveHeader)
   {
      this.haveHeader = haveHeader;
   }
   //  对 message 参数值进行 HTML 编码转换，并返回转换蛸羔
   private String filter(String message)
   {
      if (message == null)
         return (null);
      char content[] = new char[message.length()];
      message.getChars(0, message.length(), content, 0);
      //  将转换结果添加到 StringBuffer 对象中
      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;
            case ' ':
                result.append(" ");
                break;
            default:
                result.append(content[i]);
            }
        }
        return (result.toString());
    }
    @Override
    public int doEndTag() throws JspException
    {
        BufferedReader br = null;
        try
        {
            //  标签体的内容是指向文本文件的相对路径
            if (isFile)
            {
                String url = bodyContent.getString().trim();
                //  将相对路径转换成本地路径
                String path = pageContext.getServletContext().getRealPath(url);
                //  打开这个文件，并返回 BufferedReader 对象
                br = new BufferedReader(new java.io.FileReader(path));
            }
            //  标签体的内容是普通的文本
            else
            {
                //  使用 BodyContent 类的 getReader 方法返回
                //  可以读取 BodyContent 对象内容的 java.io.Reader 对象，
                //  并根据该 Reader 对象创建 BufferedReader 对象
                br = new BufferedReader(bodyContent.getReader());
            }
            //  开始生成表格（table 元素）
            String html = "<table " + ((attributes == null) ? "" : attributes)
                    + ">";
            String line = null;
            int lineCount = 1;
            //  读取标签体的内容或文本文件中的每一行，并将这些行转换成表格行
            while ((line = br.readLine()) != null)
            {
                //  如果当前行为空字符串，忽略该行
                if (line.trim().equals(""))
                    continue;
                if (escapeXML)
                {
                    //  对当前行进行 HTML 编码转换
                    line = filter(line);
                }
                String cellHtml = "td";
                if (lineCount == 1 && haveHeader)
                {
                    //  设置第一行为表头
                    cellHtml = "th";
                }
                lineCount++;
```

```
                html += "<tr>";
                String[] cells = line.split("\\s+");
                //  生成当前行的每一个单元格
                for (String cell : cells)
                {
                    html += "<" + cellHtml + ">" + cell + "</" + cellHtml + ">";
                }
                html += "</tr>";
            }
            html += "</table>";
            JspWriter out = bodyContent.getEnclosingWriter();
            //  将生成的表格代码输出的客户端
            out.write(html);
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        return this.EVAL_PAGE;
    }
}
```

2. 安装 text2Table 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 text2Table 标签：

```
<tag>
    <description>将文本转换成表格，每一列使用空格、tab 等字符分割</description>
    <name>text2Table</name>
    <tag-class>chapter10.Text2TableTag</tag-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>isFile</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>escapeXML</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>attributes</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>haveHeader</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

3. 编写 text2table.jsp 页面

text2table.jsp 页面使用 text2Table 标签将标签体的内容和指定的文本文件中的内容转换成表格。在编写 text2table.jsp 页面的代码之前，先在<Web 根目录>\chapter10 目录下建立一个 table.txt 文件，内容如下：

```
产品 ID  产品名称        单价        产地
```

```
0001A   复印机       3200    上海
0002C   打印机       1200    北京
0002D   电脑桌       210     北京
0003B   摄像头       67      广州
```

要注意上面内容中的字符串之间使用 tab 字符分隔。

text2table.jsp 页面的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
<ct:text2Table attributes="border='1'"  escapeXML="true">
aaa    bbb    ccc    dd   d
xxx    yyy    z  zz
u   uu    v   vv
ppp
</ct:text2Table>
<hr>
<ct:text2Table attributes="border='1'" haveHeader="true">
产品 ID  产品名称      单价     产地
0001A   复印机       3200    上海
0002C   打印机       1200    北京
0002D   电脑桌       210     北京
0003B   摄像头       67      广州
</ct:text2Table>
<hr>
<!--  将 table.txt 文件中的内容转换成表格  -->
<ct:text2Table attributes="border='1'" isFile="true">
chapter10\table.txt
</ct:text2Table>
```

4. 测试 text2Table 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/text2table.jsp
```

浏览器显示的效果如图 10.16 所示。

图 10.16　使用 text2Table 标签将标签体或文本文件的内容转换成表格

图 10.16 所示的显示效果对应的 HTML 代码如下：

```
<table border='1'>
    <tr>
        <td>aaa</td><td>bbb</td><td>ccc</td><td>dd   d</td>
    </tr>
    <tr>
        <td>xxx</td><td>yyy</td><td>z  zz</td>
    </tr>
    <tr>
        <td>u   uu</td><td>v   vv</td>
    </tr>
    <tr>
        <td>ppp </td>
    </tr>
</table>
<hr>
<table border='1'>
    <tr>
        <th>产品 ID</th><th>产品名称</th><th>单价</th><th>产地</th>
    </tr>
    <tr>
        <td>0001A</td><td>复印机</td><td>3200</td><td>上海</td>
    </tr>
    <tr>
        <td>0002C</td><td>打印机</td><td>1200</td><td>北京</td>
```

```
        </tr>
        <tr>
            <td>0002D</td><td>电脑桌</td><td>210</td><td>北京</td>
        </tr>
        <tr>
            <td>0003B</td><td>摄像头</td><td>67</td><td>广州</td>
        </tr>
    </table>
    <hr>
    <table border='1'>
        <tr>
            <td>产品 ID</td><td>产品名称</td><td>单价</td><td>产地</td>
        </tr>
        <tr>
            <td>0001A</td><td>复印机</td><td>3200</td><td>上海</td>
        </tr>
        <tr>
            <td>0002C</td><td>打印机</td><td>1200</td><td>北京</td>
        </tr>
        <tr>
            <td>0002D</td><td>电脑桌</td><td>210</td><td>北京</td>
        </tr>
        <tr>
            <td>0003B</td><td>摄像头</td><td>67</td><td>广州</td>
        </tr>
    </table>
```

5. text2Table 标签的使用说明

在使用 text2Table 标签时要注意，如果要转换成表格的文本内容中包含空格，需要将 escapeXML 属性值设为 true，将这些空格转换成“ ”，否则 text2Table 标签会将空格作为分隔符来分隔表格中的列。

10.6 标签体的类型

在安装自定义标签时需要使用<body-content>元素指定标签体的类型，在前面已经使用到了两种标签体类型：empty 和 JSP。<body-content>元素除了设置这两个标签体类型外，还可以设置另外两种标签体类型：scriptless 和 tagdependent。这四种标签体类型的含义如下：

empty

表示自定义标签不能有标签体，否则 Web 容器在调用自定义标签时会抛出异常。

JSP

表示自定义标签的标签体可以是任意 JSP 页面的元素，如静态内容、JSP 表达式、EL 等。

scriptless

表示自定义标签的标签体可以包含除了 Java 代码（<%=...%>或<% ... %>）外的任何 JSP 页面元素。如果标签体中包含了 Java 代码，Web 容器在调用自定义标签时将抛出异常。

tagdependent

表示自定义标签的标签体不能包含任何服务端代码（需要服务端的 JSP 引擎解析的代码），如<%...%>、<%=...%>、EL 和自定义标签。如果标签体中包含服务端代码，自定义标签并不会抛出异常，而是按着原样输出这些服务端代码。

读者可以将 10.4.2 节中实现的 iterator 标签的<body-content>元素值改为 tagdependent，并访问 iterator.jsp 页面，将会得到如图 10.17 所示的输出效果。

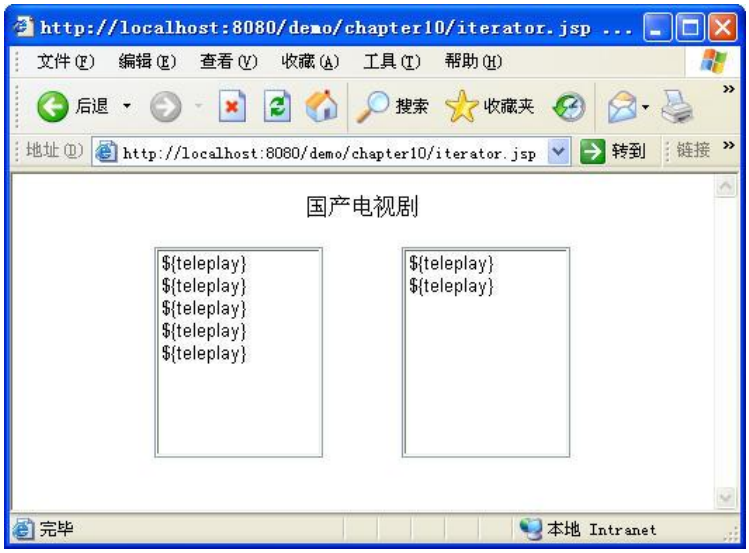

图 10.17　将<body-content>元素值设为 tagdependent

从图 10.17 所示的输出效果可以看出，iterator 标签的标签体中的 EL 并没有被 JSP 引擎翻译执行，而是直接输出到了客户端。

10.7　在自定义标签中使用 Java 变量

如果想在 JSP 页面中使用 Java 变量，必须定义这个变量。如下面的代码使用了 Java 变量 url：

```
<%
   out.println(url);
%>
```

要想使上面的代码编译通过，必须在这段代码之前使用如下的代码定义 url 变量：

```
<%
   String url = "http://nokiaguy.blogjava.net";
%>
```

然而到目前为止，自定义标签要想返回处理结果或状态信息，除了将这些信息输出到 JspWriter 对象的缓冲区外，就只有将它们保存在 Web 域中，然后使用 EL、Java 代码等技术从 Web 域中取得这些内容。由于这个过程并未定义保存相应内容的 Java 变量，因此，无法直接使用 Java 变量的方式获得自定义标签返回的内容。

为了使自定义标签更加灵活，JSP 规范允许 JSP 引擎为自定义标签创建 Java 变量，并在该变量中保存相应的内容。这样就可以直接使用 Java 变量来获得自定义标签返回的内容了。

10.6.1　在 TLD 文件中定义 Java 变量

在 10.4.2 节实现的 iterator 标签在标签体内只能通过 EL 来获得当前的迭代元素，代码如下：

```
<select name="teleplays1" multiple="multiple"
   style="width: 120px; height: 150px">
   <ct:iterator items="${teleplays}" var="teleplay" count="count">
      <option value="${count}">${teleplay}</option>
   </ct:iterator>
```

```
    </select>
```

如果将上面代码中的 EL 改成 Java 代码，例如，将${teleplay}改成<%= teleplay %>，在调用 iterator 时将会抛出异常。抛出异常的原因是未定义 teleplay 变量。因此，就需要 JSP 引擎为 iterator 标签产生这个 teleplay 变量。

要想使 JSP 引擎为自定义标签产生 Java 变量，需要在安装自定义标签时使用<variable>元素来配置 Java 变量，代码如下：

```
<tag>
    <description>迭代集合的标签</description>
    <name>iterator</name>
    <tag-class>chapter10.IteratorTag</tag-class>
    <body-content>JSP</body-content>
    <variable>
        <name-given>teleplay</name-given>
    </variable>
    <attribute>
        <name>items</name>
        <required>true</required>
        <rtexprvalue>true</rtexprvalue>
    </attribute>
    ... ...
</tag>
```

上面代码中的黑体字部分在<variable>元素中使用了<name-given>子元素设置了 Java 变量名（teleplay）。

在 javavar.jsp 页面中编写如下的代码：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
<%
    java.util.List<String> teleplays = new java.util.ArrayList<String>();
    teleplays.add("李小龙传奇");
    teleplays.add("绝密押运");
    teleplays.add("神探狄仁杰 3");
    teleplays.add("新大秦帝国");
    teleplays.add("士兵突击");
    request.setAttribute("teleplays", teleplays);
%>
<center>
国产电视剧
    <p/>
    <select name="teleplays1" multiple="multiple"
            style="width: 120px; height: 150px">
        <ct:iterator items="${teleplays}" var="teleplay" count="count">
            <!--  直接使用 teleplay 变量获得当前迭代的元素  -->
            <option value="${count}"><%=teleplay %></option>
        </ct:iterator>
    </select>
</center>
```

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/javavar.jsp
```

浏览器显示的输出效果如图 10.18 所示。

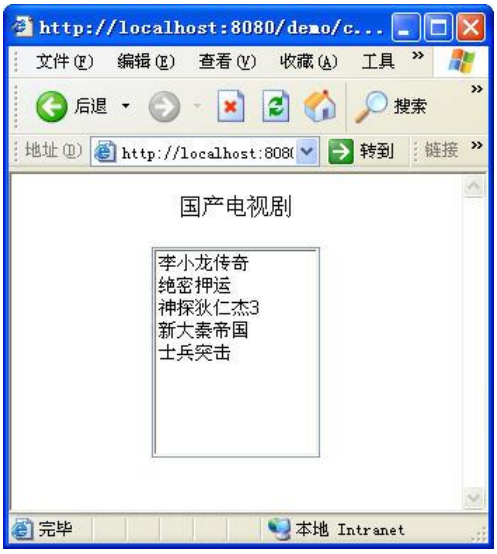

图 10.18　使用 Java 变量获得当前的迭代元素

从图 10.18 所示的输出结果可以看出，<%= teleplay %>正确输出了当前的迭代元素。

10.6.2　TLD 文件中的 Java 变量详解

虽然使用<name-given>元素可以指定 Java 变量名，但使用<name_given>元素指定的 Java 变量名是自定义标签中某个属性的值，也就是说，自定义标签的这个属性值必须和<name_given>元素值相同。如 10.6.1 节中使用<name_given>元素指定了 teleplay 变量，在 iterator 标签中的 var 属性值也必须是 teleplay，否则在调用 iterator 标签时仍然会由于未找到 Java 变量而抛出异常。

JSP 规范允许使用更灵活的方式在 TLD 文件中定义 Java 变量。如果想让自定义标签的某个指定 Java 变量的属性的值可任意指定，最直接的方法就是指定自定义标签的属性名。这个属性名可以使用<name-from-attribute>元素指定，代码如下：

```
<tag>
   <description>迭代集合的标签</description>
   <name>iterator</name>
   <tag-class>chapter10.IteratorTag</tag-class>
   <body-content>JSP</body-content>
   <variable>
      <!--  指定了 iterator 标签的 var 属性名  -->
      <name-from-attribute>var</name-from-attribute>
   </variable>
   ... ...
</tag>
```

如果使用上面的代码来定义 Java 变量，iterator 标签的 var 属性的值可以任意指定，而且无论 var 属性指定什么值，都可以将该值作为 Java 变量使用，如下面的代码所示：

```
<ct:iterator items="${teleplays}" var="abcd" count="count">
   <!--  将 abcd 作为 Java 变量使用  -->
   <option value="${count}"><%=abcd %></option>
</ct:iterator>
```

<variable>元素还有一些其他的子元素用于对 Java 变量更进一步地定义，如<variable-class>、scope 等。

<variable>元素的所有子元素的含义如表 10.3 所示。

表 10.3 <variable>元素的所有子元素的含义

元素名	是否为必选属性	描 述
name-given	是	指定一个固定名称的 Java 变量。JSP 引擎在声明 Java 变量时，会将 name-given 元素的值作为 Java 变量名。
name-from-attribute	是	指定标签的一个属性。JSP 引擎在声明 Java 变量时，会将 name-from-attribute 元素指定的标签属性的值作为 Java 变量名。
variable-class	否	指定 Java 变量的类型，默认值是 java.lang.String。
declare	否	指定 JSP 引擎在将 JSP 页面翻译成 Servlet 源代码时是否创建声明 Java 变量的语句。如果 declare 元素的值为 true，创建声明 Java 变量的语句；如果为 false，则不创建声明 Java 变量的语句。declare 元素的默认值是 true。如果在 JSP 页面中已经定义了同名的 Java 变量，应将 declare 元素的值设为 false。
scope	否	指定 Java 变量的作用范围，该元素有三个可选值：NESTED、AT_BEGIN 和 AT_END，默认值是 NESTED。其中 NESTED 表示定义的 Java 变量只能在标签体中使用，AT_BEGIN 表示定义的 Java 变量可以既可以在标签体内使用，也可以在标签后面的任何位置使用。AT_END 表示定义的 Java 变量不能在标签体内容使用，但可以在标签后面的任何位置使用。
description	否	指定描述变量的信息。

在使用<variable>元素的子元素来定义 Java 变量时应注意如下几点：

<variable>元素可以出现任意次，但每一个<variable>元素只能定义一个 Java 变量或指定一个标签属性。

<name-given>和<name-from-attribute>元素只能在一个<variable>元素中同时出现一个，也就是说这两个元素在同一个<variable>元素中不能共存。

<description>元素可以出现任意次，但所有的<description>元素必须放到<variable>元素的中其他子元素的前面。

10.6.3 JSP 引擎如何创建声明和使用 Java 变量的语句

在本节拿 iterator 标签为例介绍一下 JSP 引擎如何根据<variable>元素中定义 Java 变量的信息来创建声明和使用 Java 变量的语句。假设在安装 iterator 标签时<variable>元素只有一个子元素：<name-from-attribute>，该元素的值为 var，而且 var 属性值为 item。

由于 JSP 引擎翻译 iterator 标签时产生的代码比较复杂，而且不容易理解，因此，在本节使用了示意代码来解释这一过程。在一般情况下，读者并不需要了解 JSP 引擎翻译自定义标签所产生的 Java 代码的实际情况，但理解在安装自定义标签时<variable>元素的各个子元素的值对 JSP 引擎翻译自定义标签所产生的 Java 代码的影响，对深入学习自定义标签将有很大的帮助。

由于 iterator 标签是通过 doStartTag 方法返回 EVAL_BODY_INCLUDE 或 EVAL_BODY_BUFFERED，以及 doAfterBody 方法返回 EVAL_BODY_AGAIN 来循环执行标签体的。因此，JSP 引擎翻译 iterator 标签生成的代码会是如下的形式：

```
//  调用标签类的 doAfterBody 方法， evalDoStartTag 表示 doStartTag 方法的返回值，
//  iterator 表示标签类(IteratorTag 类）的对象实例，
//  在 doStartTag 方法中会将每一个迭代元素保存在 Web 域中
int evalDoStartTag = iterator.doAfterBody();
if (evalDoStartTag != SKIP_BODY)
```

```
{
   java.lang.String item = null;
  //  从 Web 域中获得当前迭代元素的值，并初始化 item 变量
  item = (java.lang.String)pageContext.findAttribute("item");
  do
  {
     //  执行 iterator 标签体中的代码
      ... ...
     //  调用标签类的 doAfterBody 方法，并将 doAfterBody 方法的返回值保存在
     //  evalDoAfterBody 变量中。在 doAfterBody 方法中将迭代集合的相应元素保存在 Web 域中
     int evalDoAfterBody = iterator.doAfterBody();
     //  从 Web 域中获得当前迭代元素的值，并为 item 变量赋值
     item = (java.lang.String)pageContext.findAttribute("item");
     //  如果 doAfterBody 方法的返回值不是 EVAL_BODY_AGAIN，则退出循环，不再执行标签体的内容
     if (evalDoAfterBody != EVAL_BODY_AGAIN)
        break;
  } while (true);
}
//  调用标签类的 doEndTag 方法，如果该方法的返回值是 SKIP_PAGE，
//  则不再执行 iterator 标签后面的内容
if(iterator.doEndTag() == SKIP_PAGE)
{
   return;
}
//  iterator 标签后面的内容生成的 Java 代码
... ...
```

从上面的代码可以看出，如果标签类实现了 IteratorTag 接口，JSP 引擎在翻译该标签时会将其翻译成一个 do...while 循环，并将标签体的内容都放在 do...while 循环体中。JSP 引擎会按着如下的规则来创建声明和使用 Java 变量的语句：

1. 根据<scope>元素的值确定声明 Java 变量的位置

如果<scope>元素的值是 NESTED，则 Java 变量会在 if 语句体中声明（如上面的代码所示），由于标签后面的内容在翻译成 Java 代码时都放在了 if 语句体的外面，所以标签后面的内容是无法使用在 if 语句体中声明的变量的。

如果<scope>元素的值是 AT_BEGIN，则 Java 变量会在 if 语句的前面声明，如下面的代码所示：

```
java.lang.String item = null;
if (evalDoStartTag != SKIP_BODY)
{
   ... ...
}
... ...
```

当 Java 变量在 if 语句前面声明时，无论 if 语句内部（包括 do...while 循环体，也就是标签体），还是 if 语句后面（也就是标签后面的 JSP 代码），都可以使用这个变量了。

如果<scope>元素的值是 AT_END，JSP 引擎会将声明 Java 变量的语句放在调用 doEndTag 方法的后面，代码如下：

```
if (evalDoStartTag != SKIP_BODY)
{
   ... ...
}
if(iterator.doEndTag() == SKIP_PAGE)
{
```

```
    return;
}
java.lang.String item = null;
//  iterator 标签后面的内容生成的 Java 代码
... ...
```

当 Java 变量在 if 语句后面声明时，在 if 语句体中的内容就无法使用这个变量了。

在设置 Java 变量的应用范围时要注意，如果产生 Java 变量的标签在另一个标签体内，例如，在<c:if>标签内，如果将<scope>元素的值设为 AT_BEGIN 或 AT_END，也只能在产生 Java 变量的标签体内、标签后，但在<c:if>标签的结束标记之前使用，如下面的代码所示：

```
<!--  javavar 标签的 xyz 变量的<scope>元素值为 AT_BEGIN  -->
<c:if test="${param.abc!=null}">
    <!--  产生 Java 变量的标签  -->
    <ct:javavar var="xyz">
        <!--  可以访问 xyz 变量  -->
    </ct:javavar>
    <!--  可以访问 xyz 变量  -->
</c:if>
<!--  不能访问 xyz 变量  -->
```

2. 根据<variable-class>元素确定 Java 变量的类型

如果未指定<variable-class>元素，JSP 引擎会使用该元素的默认值（java.lang.String）作为 Java 变量的类型，如果指定了<variable-class>元素，JSP 引擎在翻译标签时并不会检查<variable-class>元素指定的类型是否正确，而是直接将<variable-class>元素的值作为 Java 变量的类型放到由 JSP 页面生成的 Servlet 源代码中，但如果指定的类型不正确，Java 编译器将无法成功编译 Servlet 源代码，因此，一定要使用<variable-class>标签指定正确的类型，否则 Web 容器在调用标签时会抛出异常。

3. 从 Web 域中获得和 Java 变量同名的属性值，并为 Java 变量赋值

JSP 引擎在创建使用 Java 变量的语句时，第一件事就是为 Java 变量赋值。从上面的代码可以看出，为 Java 变量赋值的语句首先从 Web 域中查找和 Java 变量同名的域属性，并将该域属性的值赋给 Java 变量，也就是说，Java 变量的值和同名的域属性的值是相同的。因此，在定义 Java 变量时要指定在 Web 域中存在同名域属性的变量名。如在标签类中将名为“abcd”的属性保存在 Web 域中，就可以使用<name-given>元素指定名为“abcd”的 Java 变量。

4. 根据<declare>元素的值决定是否创建声明 Java 变量的语句

如果在 JSP 页面中已经声明了同名的 Java 变量，应将<declare>元素的值设为 false，这样 JSP 引擎就不会创建声明 Java 变量的语句了，但使用 Java 变量的语句（如为 Java 变量赋值的语句）仍然会被创建。

10.6.4 使用 TagExtraInfo 类定义 Java 变量

除了在 TLD 文件中定义 Java 变量外，还可以使用 TagExtraInfo 类来定义 Java 变量。TagExtraInfo 类有一个 getVariableInfo 方法，该方法返回一个 VariableInfo 对象数组，表示为当前标签创建的所有 Java 变量的信息。每一个 VariableInfo 对象表示一个变量。getVariableInfo 方法有一个 TagData 类型的参数，可以通过该参数获得当前标签的指定属性的值。getVariableInfo 方法的定义如下：

```
public VariableInfo[] getVariableInfo(TagData data)
```

VariableInfo 类只有一个构造方法，该构造方法有四个参数，分别可以指定<variable>元素的<name-given>、<variable-class>、<declare>和<scope>子元素的值。VariableInfo 类的构造方法的定义如下：

```
public VariableInfo(String varName, String className, boolean declare, int scope)
```

VariableInfo 类有如下几个方法可以分别返回通过 VariableInfo 类的构造方法设置的四个值：
public String getVarName()
public String getClassName()
public boolean getDeclare()
public int getScope()

JSP 引擎在为标签生成 Java 变量时，将通过调用这四个方法来返回相应的 Java 变量信息。

VariableInfo 类的构造方法的第一个参数（varName）只能设置<name-give>元素的值，而不能设置<name-from-attribute>元素的值，也就是说，varName 参数只表示 Java 变量名，并不代表标签的属性。为了使用 VariableInfo 对象实现<name-from-attribute>元素的功能，需要使用 TagData 类的 getAttributeString 方法返回指定标签属性的值。getAttributeString 方法的定义如下：

```
public String getAttributeString(String attName)
```

如果标签属性 var 的值将产生一个 Java 变量，可以使用如下的代码来创建 VariableInfo 对象：

```
VariableInfo variableInfo = new VariableInfo(tagData.getAttributeString("var"),
                                             "java.lang.String", true,
                                             VariableInfo.AT_BEGIN);
```

上面的代码相当于 JSP 引擎每次翻译指定 var 属性的标签时都会动态设置<name-given>元素的值，这样也就达到<name-from-attribute>元素的效果了。

在编写完 TagExtraInfo 类后，需要在 TLD 文件中使用如下的代码来安装 TagExtraInfo 类：

```
<tag>
   <description>使用 Base64 格式进行编码和解码</description>
   <name>base64</name>
   <tag-class>package.MyTag</tag-class>
   <!--  tei-class 元素必须位于 tag-class 和 body-content 元素之间  -->
   <tei-class>package.MyTagExtraInfo</tei-class>
   <body-content>JSP</body-content>
</tag>
```

由于 TagExtraInfo 类是一个抽象类，不能直接创建 TagExtraInfo 类的对象实例，因此，需要使用 TagExtraInfo 类的子类来创建 TagExtraInfo 对象实例，而 MyTagExtraInfo 就是 TagExtraInfo 类的子类。在安装 TagExtraInfo 类时要注意，不能同时使用<tei-class>和<attribute>元素来指定 Java 变量，否则 JSP 引擎在翻译自定义标签时会抛出异常。

10.6.5　编写使用 Base64 格式编码和解码的标签

在本节给出一个使用 Base64 格式进行编码和解码的自定义标签（base64 标签）。Base64 是一种编码格式，通常用于 E-mail 中传送文本和二进制文件。

【实例 10-10】　编写使用 Base64 格式编码和解码的标签

1. 程序说明

base64 标签可以将标签体的内容使用 Base64 格式进行编码，也可以将标签体的内容作为 Base64 编码进行解码。base64 标签有如下几个属性：

processor：该属性指定 base64 标签是对标签体的内容进行编码还是解码。processor 属性有两个可选值：encoder 和 decoder，分别表示对标签体的内容进行编码和解码。默认值是 encoder。

var：该属性指定保存在 Web 域中的编码或解码结果的域属性名，该属性值也是 JSP 引擎创建的 Java 变量名。如果不指定该属性，base64 标签会直接将编码或解码的结果输出的客户端，JSP 引擎也不会创建 Java 变量。

scope：该属性指定编码或解码结果保存的 Web 域，该属性有四个可选值：page、request、session 和 application。默认值是 page。

2. 编写 Base64Tag 类

Base64Tag 是一个标签类，该类负责根据相应的属性值对 base64 标签体的内容进行 Base64 编码或解码，如果指定 var 属性，base64 标签会将编码或解码的结果保存在 scope 属性所指的 Web 域中，属性名为 var 属性的值。Base64Tag 类的代码如下：

```
package chapter10;
import java.io.IOException;
import sun.misc.BASE64Encoder;
import sun.misc.BASE64Decoder;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.JspException;
import javax.servlet.jsp.PageContext;
import javax.servlet.jsp.tagext.BodyTagSupport;
public class Base64Tag extends BodyTagSupport
{
   private String processor = "encoder";
   private String var = null;
   private String scope = "page";
   public void setProcessor(String processor)
   {
      this.processor = processor;
   }
   public void setVar(String var)
   {
      this.var = var;
   }
   public void setScope(String scope)
   {
      this.scope = scope;
   }
   @Override
   public int doEndTag() throws JspException
   {
      try
      {
         //  以字符串形式获得标签体的内容
         String content = bodyContent.getString();
         JspWriter out = bodyContent.getEnclosingWriter();
         String result = "";
         if (content != null)
         {
            //  去掉标签体内容的前后空格
            content = content.trim();
            //  如果 processor 属性值是 encoder，对标签体的内容进行编码
            if ("encoder".equals(processor))
            {
               //  创建用于编码的 BASE64Encoder 对象
               BASE64Encoder encoder = new BASE64Encoder();
               //  使用 encode 方法进行 Base64 编码
               result = encoder.encode(content.getBytes("UTF-8"));
            }
            //  如果 processor 属性值是 decoder，对标签体的内容进行解码
            else if ("decoder".equals(processor))
```

```
                {
                    //  创建用于解码的 BASE64Decoder 对象
                    BASE64Decoder decoder = new BASE64Decoder();
                    //  使用 decodeBuffer 方法进行 Base64 解码，该方法返回一个字节数组
                    result = new String(decoder.decodeBuffer(content.trim()),"UTF-8");
                }
                int scopeID = PageContext.PAGE_SCOPE;
                //  根据 scope 属性的值设置 Web 域的 ID
                if ("page".equals(scope))
                {
                    scopeID = PageContext.PAGE_SCOPE;
                }
                else if ("request".equals("request"))
                {
                    scopeID = PageContext.REQUEST_SCOPE;
                }
                else if ("session".equals("session"))
                {
                    scopeID = PageContext.SESSION_SCOPE;
                }
                else if ("application".equals("application"))
                {
                    scopeID = PageContext.APPLICATION_SCOPE;
                }
                //  如果未指定 var 属性，将编码或解码结果直接输出的客户端
                if (var == null)
                {
                    out.write(result);
                }
                //  如果指定 var 属性，将编码或解码结果保存在 Web 域中
                else
                {
                    pageContext.setAttribute(var, result, scopeID);
                }
            }
        }
        catch (IOException e)
        {
            e.printStackTrace();
        }
        return this.EVAL_PAGE;
    }
}
```

由于使用 BASE64Encoder 类的 encode 方法进行 Base64 编码时需要操作字节数组，而在上面的代码中将标签体的内容按着 UTF-8 编码格式转换成了字节数组，并使用 UTF-8 格式的字节数组进行编码。因此，在使用 BASE64Decoder 类的 decodeBuffer 方法进行解码后返回的字节数组仍然是 UTF-8 格式的，因此，需要将 String 类的第二个参数值指定为 UTF-8。

要注意的是在对标签体内容进行解码时（标签体的内容是 Base64 编码）需要将标签体内容前后的内容去掉，否则，decodeBuffer 方法会将空格也当成 Base64 编码来处理。

3. 编写 Base64TagExtraInfo 类

Base64TagExtraInfo 是 TagExtraInfo 类的子类，负责设置 Java 变量的信息。如果未指定 var 属性，或 var 属性值为 null，则不设置任何 Java 变量的信息。Base64TagExtraInfo 类的代码如下：

```
package chapter10;
```

```
import javax.servlet.jsp.tagext.TagData;
import javax.servlet.jsp.tagext.TagExtraInfo;
import javax.servlet.jsp.tagext.VariableInfo;
public class Base64TagExtraInfo extends TagExtraInfo
{
    //  返回所有 Java 变量的信息
    @Override
    public VariableInfo[] getVariableInfo(TagData data)
    {
        //  获得 var 属性的值
        String value = data.getAttributeString("var");
        //  如果 var 属性值不为 null，将该属性值设置成 Java 变量名，并设置该 Java 变量的其他信息
        if(value != null)
        {
            VariableInfo variableInfo = new VariableInfo(data
                    .getAttributeString("var"), "java.lang.String", true,
                    VariableInfo.AT_BEGIN);
            return new VariableInfo[]{ variableInfo };
        }
        //  如果未指定 var 属性，或 var 属性值为 null，则调用父类的 getVariableInfo 方法
        else
        {
            return super.getVariableInfo(data);
        }
    }
}
```

4. 安装 base64 标签

在 jsp-taglib.tld 文件中添加如下的内容来安装 base64 标签：

```
<tag>
    <description>使用 Base64 格式进行编码和解码</description>
    <name>base64</name>
    <tag-class>chapter10.Base64Tag</tag-class>
    <!--  安装 Base64TagExtraInfo 类  -->
    <tei-class>chapter10.Base64TagExtraInfo</tei-class>
    <body-content>JSP</body-content>
    <attribute>
        <name>processor</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>scope</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
    <attribute>
        <name>var</name>
        <required>false</required>
        <rtexprvalue>false</rtexprvalue>
    </attribute>
</tag>
```

5. 编写 base64.jsp 页面

base64.jsp 页面使用 base64 标签对相应的信息进行 Base64 编码和解码。base64.jsp 页面的代码如下：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"%>
<%@ taglib uri="http://nokiaguy.blogjava.net" prefix="ct"%>
“今天是星期几”的 Base64 编码：
<!--  直接输出编码结果  -->
<ct:base64>今天是星期几</ct:base64>
<hr>
<!--  对 sourceString 请求参数值进行编码  -->
<ct:base64 var="base64Result" scope="request" processor="encoder">
   ${param.sourceString}
</ct:base64>
<!--  对 base64String 请求参数值进行解码  -->
<ct:base64 var="sourceResult" scope="request" processor="decoder">
   ${param.base64String}
</ct:base64>
<form >
   原字符串：<input type="text" name = "sourceString" value="${param.sourceString}"/>
<input type="submit" value="进行 Base64 编码"/>
</form>
编码后的字符串：<%=base64Result %>
<hr>
<form >
   Base64 字符串：
   <input type="text" name = "base64String" value="${param.base64String}"/>
   <input type="submit" value="进行 Base64 解码"/>
</form>
解码后的字符串：<%=sourceResult %>
```

6. 测试 base64 标签

在浏览器地址栏中输入如下的 URL：

```
http://localhost:8080/demo/chapter10/base64.jsp
```

在【原字符串】文本框中输入“Base64 编码转换”，单击【进行 Base64 编码】按钮，将会显示如图 10.19 所示的效果。

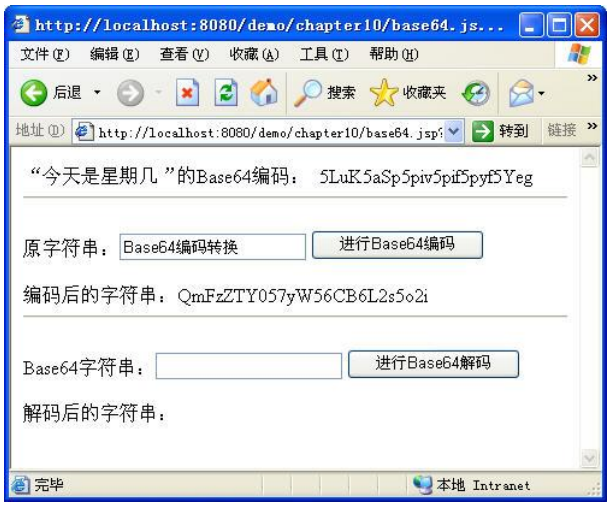

图 10.19　使用 base64 标签对标签体的内容进行编码

如果【原字符串】文本框中出现乱码，需要将<Tomcat 安装目录>\conf\sever.xml 文件中 8080 端口添加一个 URIEncoding 属性，并将该属性值设为“UTF-8”，代码如下：

```
<Connector connectionTimeout="20000" port="8080"
   protocol="HTTP/1.1" redirectPort="8443" URIEncoding="UTF-8"/>
```

将图 10.19 所示的编码结果复制到【Base64 字符串】文本框中，单击【进行 Base64 解码】按钮，将会显示如图 10.20 所示的效果。

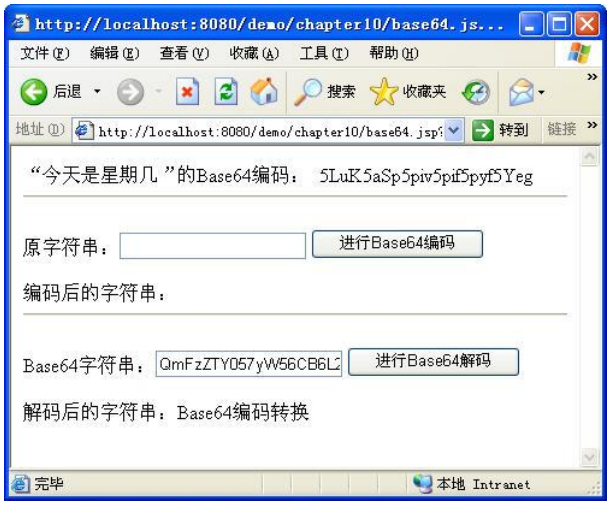

图 10.20　使用 base64 标签对标签体的内容进行解码

7. 程序总结

虽然使用<variable>元素和 TagExtraInfo 类都可以设置 Java 变量的信息，但使用 TagExtraInfo 类设置 Java 变量的信息会更灵活。如 JSTL 中有很多标签都有 var 属性，该属性表示将执行标签体内容的结果保存在 Web 域中的域属性名。如果要使用<variable>元素设置 var 变量的信息，需要在每一个标签的<tag>元素中都包含<variable>子元素，而使用 TagExtraInfo 类来设置 Java 变量，只需要在每一个标签的<tag>元素中包含一个<tei-class>子元素即可。

10.7　小结

本章主要介绍了如何编写传统的自定义标签。传统的自定义标签必须实现 Tag 接口，为了给自定义标签增加更多的功能，JSP API 又提供了两个接口：IterationTag 和 BodyTag。这两个接口分别用来实现自定义标签的迭代功能和截获、修改标签体内容的功能。这两个接口都是 Tag 的子接口，BodyTag 接口是 IterationTag 的子接口。

由于并不是每一个标签类都需要 Tag、IterationTag 或 BodyTag 接口的所有功能，因此，就没有必要每一个自定义标签的标签类都要实现这些接口。为了解决这个问题，JSP API 为 Tag 和 BodyTag 接口分别提供了两个默认实现类，这两个默认实现类的类名就是相应接口名加“Support”后缀。也就是说，Tag 接口的默认实现类是 TagSupport、BodyTag 接口的默认实现类是 BodyTagSupport。

自定义标签还可以使用<variable>元素或 TagExtraInfo 类定义 Java 变量。当 JSP 引擎在翻译自定义标

签时会自动创建声明和使用这些 Java 变量的 Java 语句。通过这种方式，可以在 JSP 页面中直接使用 Java 代码来访问这些 Java 变量。