Jstl 开发使用手册

李晓东

2009-9-3

Jstl 介绍

在 JSTL1.1.2 中有以下这些标签库是被支持的：Core 标签库、XML processing 标签库、 I18N formatting 标签库、 Database access 标签库、 Functions 标签库。对应的标识符见表 2 所示：

表 2 标签库的标识符

标签库	URI	前缀
Core	http://java.sun.com/jsp/jstl/core	c
XML processing	http://java.sun.com/jsp/jstl/xml	x
I18N formatting	http://java.sun.com/jsp/jstl/fmt	fmt
Database access	http://java.sun.com/jsp/jstl/sql	sql
Functions	http://java.sun.com/jsp/jstl/functions	fn

下面看例 5 ，简单使用标签库的示例。

例 5 ：简单 JSTL 标签库示例

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
  <html>
    <body>
      <c:forEach var="i" begin="1" end="10" step="1">
        ${i}
        <br />
      </c:forEach>
    </body>
   </html>
```

在该示例的 JSP 页面中声明了将使用 Core 标签库，它的 URI 为“ http://java.sun.com/jsp/jstl/core ”，前缀为“ c ”。之后，

页面中〈c:forEach〉标签就是使用了 JSTL 的标签进行了工作。对于该标签的功能，这里暂时不作具体讲解，只是让读者能够有个简单的概念，了解怎样定义和使用标签库。

Core 标签库

Core 标签库，又被称为核心标签库，该标签库的工作是对于 JSP 页面一般处理的封装。在该标签库中的标签一共有 14 个，被分为了四类，分别是：

运算式标签：〈c:out〉、〈c:set〉、〈c:remove〉、〈c:catch〉。

流程控制标签：〈c:if〉、〈c:choose〉、〈c:when〉、〈c:otherwise〉。

循环控制标签：〈c:forEach〉、〈c:forTokens〉。

URL 相关标签：〈c:import〉、〈c:url〉、〈c:redirect〉、〈c:param〉。

以下是各个标签的用途和属性以及简单示例。

用于显示的 <c:out> 标签

〈c:out〉标签是一个最常用的标签，用于在 JSP 中显示数据，就像是<%= scripting-language %>一样。例如：

```
Hello ! 〈c:out value="${username}" />
```

语法 1：没有本体（body）内容

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

语法 2：有本体内容

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

<c:out> 标签属性和说明

属性	描述
value	需要显示出来的值，可以是 EL 表达式或常量（必须）
default	当 value 为 null 时显示 default 的值（可选）
escapeXml	当设置为 true 时会主动更换特殊字符，比如 “ <,>,& ”（可选，默认为 true ）

Null 和错误处理：

如果 vlaue 为 null 时，会显示 default 的值，假设没有设定 default 值时，则显示一个空字符串。

范例：

```
1.<c:out value="Hello JSP 2.0 !! " />
2.<c:out value="${ 3 + 5 }" />
3.<c:out value="${ param.data }" default="No Data" />
4.<c:out value="<p>有特殊字符</p>" />
5.<c:out value="<p> </p>" escapeXml="false" />
```

1. 在网页上显示 Hello JSP 2.0 !!
2. 在网页上显示 8
3. 在网页上显示有表单传送过来的 data 参数值，假设没有 data 参数活着 data 参数为 null,则在网页上显示 No Data
4. 在网页上显示 有特殊字符
5. 在网页上显示 有特殊字符

用于赋值的 <c:set> 标签

<c:set> 标签主要用来将变量存储到 jsp 范围中或 JavaBean 中的属性中。

语法 1：将 value 值存储到范围为 scoped 的 varName 变量中

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

语法 2：将本体内容存储到范围为 scoped 的 varName 变量中

```
<c:set var="varName"
       scope="{ page|request|session|application }">
...本体内容
</c:set>
```

语法 3：将 value 值存储到 target 物件的属性中

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

语法 4：将本体内容值存储到 target 物件的属性中

```
<c:set target="target" property="propertyName">
... 本体内容
</c:set>
```

<c:set> 标签属性和说明

属性	描述
value	要被存放的值，可以是 EL 表达式或常量
target	被赋值的 JavaBean 实例的名称，若存在该属性则必须存在 property 属性（可选）
property	JavaBean 实例的变量属性名称（可选）
var	被赋值的变量名（可选）
scope	变量的作用范围，若没有指定，默认为 page （可选）

说明：

使用<c:set>时，var 主要用来存放运算式的结果；scope 用来设定存储的范围，例如：假设 scope="session" ，则将会把结果存放到 session 中。如果没有指定 scope，则它会存储到默认的 page 中。

我们考虑下列写法：

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

把 1+1 测结果存放到变量 number 中。如果没有 value 属性时，此时的 value 值为<c:set>和</c:set>本体内容，我们看一下下面范例：

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

```
</c:set>
```

上面的<c:out value="${1+1}" />部分我们可以改写成 2 或是<%=1+1%>结果都会是一样。也就是说<c:set>把本体运算结果当作 value 值。

范例：

```
<c:set var="number" scope="request" value="${1 + 1}" />
<c:set var="number" scope="session" />
${3 + 5}
</c:set>
<c:set var="number" scope="request"
       value="${ param.number }" />
<c:set target="User" property="name"
       value="${ param.Username}" />
```

1. 将 2 存放到 request 范围的变量 number 中
2. 将 8 存放到 request 范围的变量 number 中
3. 假设${param.number}为 null 时，则移除 request 范围的 number 变量；若${param.number}不为 null 时，则将${param.number}的值存放到 request 范围的 number 变量中
4. 假设${param.Username}为 null 时，怎设定 User 的 name 属性为 null, 若${param.Username}不 null 时,则将${param.Username}的值存入 User 的 name 属性中（setter 机制）

用于删除的 <c:remove> 标签

〈c:remove〉标签用于删除存在于 scope 中的变量。

语法：

```
<c:remove var="varName"
          scope="{ page|request|session|application }" />
```

〈c:remove〉标签属性和说明

属性	描述
var	需要被删除的变量名 （必须）
scope	变量的作用范围，若没有指定，默认为全部查找（可选）

范例：

```
<c:remove var="sampleValue" scope="session"/>
   ${sessionScope.sampleValue}<br>
```

该示例将存在于 Session 中名为“ sampleValue ”的变量删除。下一句 EL 表达式显示该变量时，该变量已经不存在了。

用于异常捕获的〈c:catch〉标签

〈c:catch〉标签允许在 JSP 页面中捕捉异常。它包含一个 var 属性，是一个描述异常的变量，改变量可选。若没有 var 属性的定义，那么仅仅捕捉异常而不做任何事情，若定义了 var 属性，则可

以利用 var 所定义的异常变量进行判断转发到其他页面或提示报错信息。

语法：

```
<c:catch var="varName" >
...将要抓取错误的地方
   </c:catch>
```

<c:catch>标签属性和说明

属性	描述
var	存储错误信息的变量（可选）

说明：

<c:catch>主要将可能发生错误的地方放在<c:catch>和</c:catch>之间。如果发生错误将错误信息存放到变量 var 中。另外，当错误发生在<c:catch>和</c:catch>之间时，则只有<c:catch>和</c:catch>之间的代码会被终止忽略，整个网页不会被终止。

范例：

```
<c:catch var="err">
      ${param.sampleSingleValue[9] == 3}
   </c:catch>
${err}
```

当“ ${param.sampleSingleValue[9] == 3} ”表达式有异常时，可以从 var 属性“ err ”得到异常的内容，通常判断“ err ”是否为 null 来决定错误信息的提示。

用于判断的 <c:if> 标签

<c:if> 标签用于简单的条件语句，和我们一般在代码中的 if 一样。

语法 1：没有本体内容

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

语法 2：有本体内容

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

<c:if> 标签属性和说明

属性	描述
test	需要判断的条件 （必须）
var	保存判断结果 true 或 false 的变量名,该变量可供之后的工作使用（可选）
scope	变量的作用范围，若没有指定，默认为保存于 page 范围中的变量（可选）

说明:<c:if>标签必须有 test 属性，当 test 属性中的运算式为 true 时，会执行本体内容，若为 false,则不会执行。例如：${param.username = = 'admin'}，如果 param.username 等于 admin 是结果为 true,否则为 false.

范例：

```
<c:if test="${param.username = = 'admin' }">
 ADMIN !! 你好
</c:if>
```

该示例将判断 如果为true,则打印ADMIN !! 你好。否则不执行体内容。

用于复杂判断的 <c:choose> 、 <c:when> 、 <c:otherwise> 标签

这三个标签用于实现复杂条件判断语句，类似“ if,elseif ”的条件语句。

<c:choose> 标签没有属性，可以被认为是父标签，

<c:when> 、 <c:otherwise> 将作为其子标签来使用。

<c:when> 标签等价于“ if ”语句，它包含一个 test 属性，该属性表示需要判断的条件。

<c:otherwise> 标签没有属性，它等价于“ else ”语句。

下面看一个复杂条件语句的示例。

```
<c:choose>
    <c:when test="${paramValues.sampleValue[2] == 11}">
            not 12 not 13,it is 11
    </c:when>
    <c:when test="${paramValues.sampleValue[2] == 12}">
           not 11 not 13,it is 12
    </c:when>
    <c:when test="${paramValues.sampleValue[2] == 13}">
          not 11 not 12,it is 13
    </c:when>
    <c:otherwise>
         not 11 、 12 、 13
    </c:otherwise>
  </c:choose>
```

该示例将判断 request 请求提交的传入控件数组参数中，下标为“ 2 ”控件内容是否为“ 11 ”或“ 12 ”或“ 13 ”，并根据判断结果显示各自的语句，若都不是则显示“ not 11 、 12 、 13 ”。

用于循环的 <c:forEach> 标签

<c:forEach> 为循环控制标签，可以将集合中的成员循序遍历一遍。运作方式为当条件符合时，就会重复执行<c:forEach>本体内容。

语法 1：迭代集合中说有成员

```
<c:forEach var="varName" items="collection"
           varStatus="varStatusName" begin="begin"
            end="end" step="step">
   本体内容
</c:forEach>
```

语法 2：指定迭代次数

```
<c:forEach var="varName" items="collection"
           varStatus="varStatusName" begin="begin"
            end="end" step="step">
```

<c:forEach> 标签属性和说明

属性	描述
items	进行循环的集合（可选）
begin	开始位置（可选）有时必须大于等于 0
end	结束位置（可选）有时必须大于 begin
step	循环的步长（可选）有时必须亚于等于 0
var	做循环的对象变量名，若存在 items 属性，则表示循环集合中对象的变量名（可选）
varStatus	显示循环状态的变量（可选）

集合循环的范例：

```
<%ArrayList arrayList = new ArrayList();
                        arrayList.add("aa");
                        arrayList.add("bb");
                        arrayList.add("cc");
   %>
   <%request.getSession().setAttribute("arrayList",
   arrayList);%>
   <c:forEach items="${sessionScope.arrayList}"
              var="arrayListI"> ${arrayListI}
   </c:forEach>
```

该示例将保存在 Session 中的名为“ arrayList ”的 ArrayList 类型集合参数中的对象依次读取出来， items 属性指向了 ArrayList 类型集合参数， var 属性定义了一个新的变量来接收集合中的对象。最后直接通过 EL 表达式显示在页面上。

范例：

```
<c:forEach var="i" begin="1" end="10" step="1"> ${i}<br />
</c:forEach>
```

该示例从“ 1 ”循环到“ 10 ”，并将循环中变量“ i ”显示在页面上。

用于分隔字符的 <c:forTokens> 标签

〈c:forTokens〉标签可以根据某个分隔符分隔指定字符串，遍历所有成员。相当于 java.util.StringTokenizer 类。

语法：

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

〈c:forTokens〉标签 属性和说明

属性	描述
items	进行分隔的 EL 表达式或常量，被迭代的字串（必须）
delims	定义用来分割字串的分隔符
begin	开始位置（可选） 必须大于等于 0
end	结束位置（可选） 必须大于 begin
step	循环的步长，默认为 1 （可选）必须大于等于 0
var	做循环的对象变量名（可选）
varStatus	显示循环状态的变量（可选）

范例：

```
<c:forTokens items="aa,bb,cc,dd" begin="0" end="2" step="2"
             delims="," var="aValue"> ${aValue}
</c:forTokens>
```

需要分隔的字符串为“ aa,bb,cc,dd ”，分隔符为“ , ”。 begin 属性 指定从第一个“ , ”开始分隔， end 属性指定分隔到第三个“ , ”,并将做循环的变量名指定为“ aValue ”。由于步长为“ 2 ”，使用 EL 表达式 ${aValue} 只能显示“ aa

用于包含页面的 <c:import>

<c:import> 标签可以把其他静态或动态文件包含到本页面来。和<jsp:include>最大的区别是:<jsp:include>只能包含和自己同一个应用下的文件;<c:import>即可包含和自己同一个应用下的文件，也可以包含和自己不同的应用下的文件。

语法 1：

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

语法 2：

```
<c:import url="url" [context="context"]
varReader="varReaderName" [charEncoding="charEncoding"]>
  本体内容
```

```
</c:import>
```

<c:import> 标签属性和说明

属性	描述
url	需要导入页面的 URL
context	Web Context 该属性用于在不同的 Context 下导入页面,当出现 context 属性时,必须以“ / ”开头，此时也需要 url 属性以“ / ”开头（可选）
charEncoding	导入页面的字符集（可选）
var	可以定义导入文本的变量名（可选）
scope	导入文本的变量名作用范围（可选）
varReader	接受文本的 java.io.Reader 类变量名（可选）

范例：

```
<c:import url="/MyHtml.html" var="thisPage" />

<c:import url="/MyHtml.html" context=" /sample2"

          var="thisPage"/>

<c:import url="www.sample.com/MyHtml.html" var="thisPage"/>
```

该示例演示了三种不同的导入方法，第一种是在同一 Context 下的导入，第二种是在不同的 Context 下导入，第三种是导入任意一个 URL 。如果 url 为 null 或空的时 ，会抛出 jspException。

用于得到 URL 地址的 <c:url> 标签

<c:url> 标签用于得到一个 URL 地址。

语法 1：没有本体内容

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

语法 2：本体内容代表参数

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

<c:url> 标签属性和说明

属性	描述
value	页面的 URL 地址
context	Web Context 该属性用于得到不同 Context 下的 URL 地址，当出现 context 属性时，必须以“ / ”开头，此时也需要 url 属性以“ / ”开头（可选）
charEncoding	URL 的字符集（可选）
var	存储 URL 的变量名（可选）
scope	变量名作用范围（可选）

范例：

```
<a href="
<c:url value="http:// www.javaworld.com.tw" >
<c:param name="param" value="value"/>
</c:url>"> Java </a>
```

点击 java 时产生的 url=”http:// www.javaworld.com.tw?&param=value”。

用于页面重定向的 <c:redirect> 标签

〈c:redirect〉可以将客户端请求从一个页面重定向到其他文件。该标签的作用相当于 response.setRedirect 方法的工作。它包含 url 和 context 两个属性，属性含义和〈C:url〉标签相同。

语法 1：没有本体内容

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

语法 2：本体内容代表参数

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

〈c:redirect〉标签属性和说明

属性	描述
url	重定向的目标地址（必须）
context	Web Context 该属性用于得到不同 Context 下的 URL 地址，当出现 context 属性时，必须以“ / ”开头，此时也需要 url 属性以“ / ”开头（可选）

范例：

```
<c:redirect url="/MyHtml.html">
  <c:param name="param" value="value"/>
</c:redirect>
```

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

用于包含传递参数的<c:param>标签

<c:param>用来为包含或重定向的页面传递参数。它的属性和描述如表 11 所示：

表 11　　<c:param>标签属性和说明

属性	描述
name	传递的参数名
value	传递的参数值（可选）

范例：

```
<c:redirect url="/MyHtml.jsp">
   <c:param name="userName" value=" RW" />
</c:redirect>
```

该示例将为重定向的“ MyHtml.jsp ”传递指定参数“ userName=’RW’ ”。

Fmt 标签库

JSTL fmt 库中的定制标记主要分成四组。第一组允许您设置本地化上下文，其它标记将在其中进行操作。换句话说，这组标记允许

页面作者显式地设置其它 fmt 标记在格式化数据时将要使用的语言环境和时区。第二组和第三组标记分别支持对日期和数字进行格式化和解析。最后一组标记侧重于对文本消息进行本地化。既然我们已经有了些基本了解，那就让我们集中精力逐个研究这四组标记，并演示其用法。

用户语言环境

正如我们已经讨论过的那样，JSTL 标记在格式化数据时所使用的语言环境往往是通过查看用户浏览器发送的每个 HTTP 请求所包含的 Accept-Language 头来确定的。如果没有提供这样的头，那么 JSTL 提供一组 JSP 配置变量，您可以设置这些变量以指定缺省的语言环境。如果尚未设置这些配置变量，那么就使用 JVM 的缺省语言环境，该缺省语言环境是从 JSP 容器所运行的操作系统中获取的。fmt 库提供了其自身的定制标记，以覆盖这个确定用户语言环境的过程：<fmt:setLocale> 。正如下面的代码片段所示，<fmt:setLocale> 操作支持三个属性：

```
<fmt:setLocale value="expression" scope="scope"
               variant="expression"/>
```

其中只有一个属性是必需的： value 属性。该属性的值应当是命名该语言环境的一个字符串或者是 java.util.Locale 类的一个实例。语言环境名称是这样组成的：小写的两字母 ISO语言代码，可选地，后面可以跟下划线或连字符以及大写的两字母 ISO 国家或地区代

码。例如， en 是英语的语言代码， US 是美国的国家或地区代码，因此 en_US （或 en-US）将是美式英语的语言环境名称。类似的， fr 是法语的语言代码， CA 是加拿大的国家或地区代码，因此 fr_CA （或 fr-CA ）是加拿大法语的语言环境名称（请参阅参考资料以获取所有有效的 ISO 语言和国家或地区代码的链接）。当然，由于国家或地区代码是可选的，因此 en 和 fr 本身就是有效的语言环境名称，适用于不区别这些相应语言特定方言的应用程序。

<fmt:setLocale> 的可选属性 scope 用来指定语言环境的作用域。 page 作用域指出这项设置只适用于当前页，而 request 作用域将它应用于请求期间访问的所有 JSP 页面。如果将 scope 属性设置成 session ，那么指定的语言环境被用于用户会话期间访问的所有JSP 页面。值 application 指出该语言环境适用于该 Web 应用程序所有 JSP 页面的全部请求和该应用程序所有用户的全部请求。variant 属性（也是可选的）允许您进一步针对特定的 Web 浏览器平台或供应商定制语言环境。例如， MAC 和 WIN 分别是 Apple Macintosh 和 Microsoft Windows 平台的变体名。

下面的代码片段说明了如何使用 <fmt:setLocale> 标记来显式指定用户会话的语言环境设置：

```
<fmt:setLocale value="fr_CA" scope="session"/>
```

JSP 容器处理完该 JSP 代码段之后，将忽略用户浏览器设置中所指定的语言首选项。

用于时区

〈fmt:setTimeZone〉操作像〈fmt:setLocale〉一样，可以用来设置其它 fmt 定制标记所使用的缺省时区值。它的语法如下所示：

```
<fmt:setTimeZone value="expression" var="name"
                 scope="scope"/>
```

和〈fmt:setLocale〉一样，只有 value 属性是必需的，但是在本例中它应当是时区名或 java.util.TimeZone 类的实例。遗憾的是，对于时区命名目前还没有任何被广泛接受的标准。因此您可以用于〈fmt:setTimezone〉标记的 value 属性的时区名是特定于 Java 平台的。您可以通过调用 java.util.TimeZone 类的getAvailableIDs()静态方法来检索有效的时区名列表。示例包括 US/Eastern 、 GMT+8 和 Pacific/Guam 。和〈fmt:setLocale〉的情况一样，您可以使用可选的 scope 属性来指出时区设置的作用域。下面的代码演示了〈fmt:setTimeZone〉的用法，它用来指定适用于单个用户会话的时区：

```
<fmt:setTimeZone value="Australia/Brisbane"
                 scope="session"/>
```

您还可以使用〈fmt:setTimeZone〉操作将 TimeZone 实例的值存储在限定了作用域的变量中。在本例中，您可以使用 var 属性来命名限定了作用域的变量，用 scope 属性来指定该变量的作用域（例如，就象这两个属性用在〈c:set〉和〈c:if〉操作中）。请注意，当您以这种方式使用〈fmt:setTimeZone〉操作时，它唯一的副作用就

是设置指定的变量。当指定 var 属性时，对于任何其它 JSTL 标记使用什么时区，不会对 JSP 环境作任何更改。这组中的最后一个标记是 <fmt:timeZone> 操作：

```
<fmt:timeZone value="expression">
        body content
</fmt:timeZone>
```

和 <fmt:setTimeZone> 一样，您可以使用该标记来指定将由其它 JSTL 标记使用的时区。但是， <fmt:timeZone> 操作的作用域仅限于其标记体内容。在 <fmt:timeZone> 标记体中，由标记的 value 属性指定的时区覆盖了 JSP 环境中现有的任何其它时区设置。和 <fmt:setTimeZone> 的情况一样， <fmt:timeZone> 标记的 value 属性应当是时区名或者是 java.util.TimeZone 实例。后面的清单 1 中提供了一个如何使用 <fmt:timeZone>的示例。

用于日期标记

fmt 库包含了用来与日期和时间进行交互的两个标记：<fmt:formatDate> 和 <fmt:parseDate> 。顾名思义，<fmt:formatDate> 用来格式化和显示日期和时间（数据输出），而 <fmt:parseDate> 用来解析日期和时间值（数据输入）。

<fmt:formatDate> 的语法如下所示：

```
<fmt:formatDate value="expression" timeZone="expression"
                type="field" dateStyle="style"
```

```
                timeStyle="style" var="name"
                pattern=" expression" scope="scope"/>
```

只有 value 属性才是必需的。其值应当是 java.util.Date 类的实例，指定要进行格式化和显示的日期和 / 或时间数据。可选的 timeZone 属性指出将要显示哪个时区的日期和 / 或时间。如果没有显式地指定 timeZone 属性，那么就使用周围任何 <fmt:timeZone> 标记所指定的时区。如果 <fmt:timeZone> 标记的主体部分没有包含 <fmt:formatDate> 操作，那么就使用任何适用的<fmt:setTimeZone> 操作所设置的时区。如果没有相关的<fmt:setTimeZone> 操作，那么就使用 JVM 的缺省时区(也就是,专为本地操作系统而设置的时区)。type 属性指出要显示指定的 Date 实例的哪些字段，应当是 time 、date 或 both 。该属性的缺省值是 date ，因此如果没有给出 type 属性,那么 <fmt:formatDate> 标记(名符其实)将只显示与 Date 实例相关的日期信息，这个信息用该标记的 value 属性指定dateStyle 和 timeStyle 属性分别指出应当如何格式化日期和时间信息。有效的样式有 default 、 short 、 medium 、 long 和 full 。缺省值自然是 default ，指出应当使用特定于语言环境的样式。其它四个样式值的语义与 java.text.DateFormat 类定义的一样。可以使用 pattern 属性来指定定制样式，而不必依赖于内置样式。给出定制样式后，该模式属性的值应当是符合 java.text.SimpleDateFormat 类约定的模式字符串。这些模式基于用对应的日期和时间字段代替模式内指定的字符。例如，模式 MM/dd/yyyy 表明应当显示用正斜杠分隔

的两位数的月份和日期值以及四位数的年份值。如果指定了 var 属性，那就把包含格式化日期的 String 值指派给指定的变量。否则，<fmt:formatDate> 标记将写出格式化结果。当指定了 var 属性后，scope 属性指定所生成变量的作用域。清单 1（它是本系列第 2 部分清单 8 的扩展）包含了 <fmt:formatDate> 标记的两种用法。在第一种用法中，<fmt:formatDate> 只用来显示第一个 weblog 项的创建时间戳记的日期部分。此外，为 dateStyle 属性指定了一个 full 值，这样一来所有的日期字段就将用一种特定于语言环境的格式进行显示。

清单 1. 使用 <fmt:formatDate> 标记来显示日期和时间值

```
<table>
 <fmt:timeZone value="US/Eastern">
  <c:forEach items="${entryList}" var="blogEntry"
             varStatus="status">
   <c:if test="${status.first}">
   <tr><td align="left" class="blogDate">
    <fmt:formatDate value="${blogEntry.created}"
                    dateStyle="full"/>
   </td></tr>
   </c:if>
   <tr><td align="left" class="blogTitle">
   <c:out value="${blogEntry.title}" escapeXml="false"/>
```

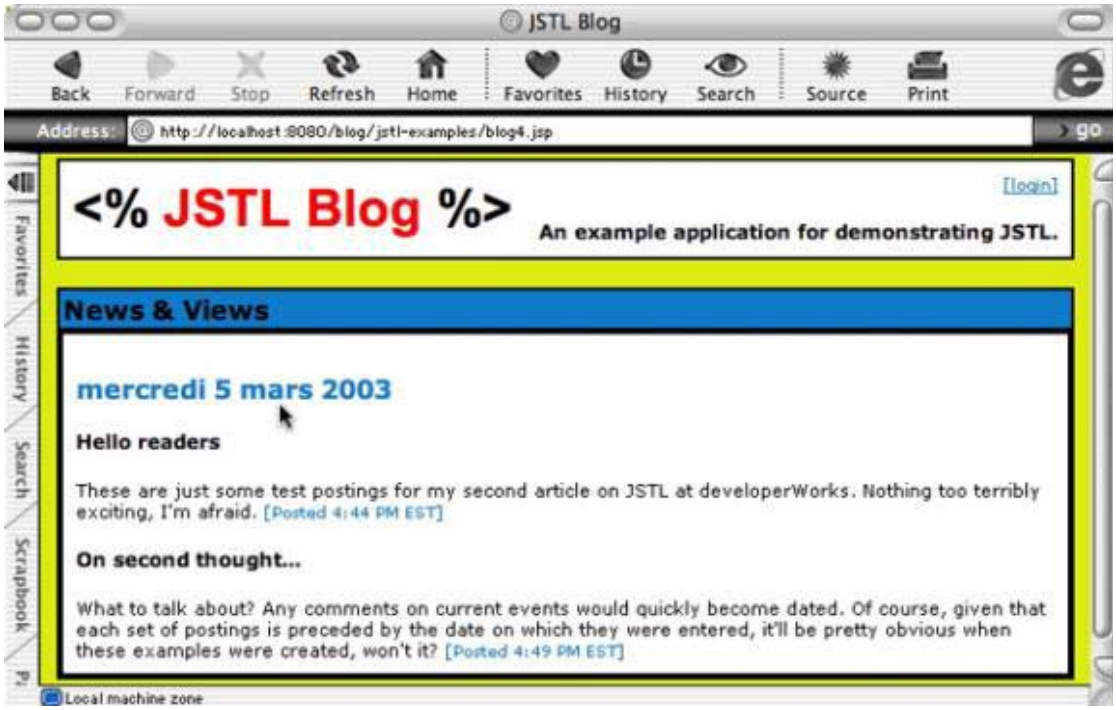
```
    </td></tr>
    <tr><td align="left" class="blogText">
    <c:out value="${blogEntry.text}" escapeXml="false"/>
    <font class="blogPosted">
          [Posted <fmt:formatDate
                    value="${blogEntry.created}"
                    pattern="h:mm a zz"/>]
    </font>
    </td></tr>
  </c:forEach>
 </fmt:timeZone>
</table>
```

在 <c:forEach> 循环体中，第二个 <fmt:formatDate> 操作只用来显示每个项的创建日期的时间部分。在本例中， pattern 属性用来控制时间值的格式化、并控制指定一位数的小时显示（如果可能的话）、12 小时的时钟和缩写时区的输出。输出如图 2 所示：

图 2. 清单 1 中 en_US 语言环境的输出

更准确地说，用户浏览器设置指定首选项是英语时，就会产生图 2 中所示的输出。但是由于〈fmt:formatDate〉对用户语言环境敏感，所以浏览器首选项的改变将导致生成不同的内容。例如，当给定的首选项是法语语言环境时，则结果会如图 3 所示：

图 3. 清单 1 中 fr_CA 语言环境的输出



<fmt:formatDate> 生成了 java.util.Date 实例的本地化字符串表示，而<fmt:parseDate>操作执行相反的操作：给定一个表示日期和 / 或时间的字符串，它将生成相应的 Date 对象

<fmt:parseDate> 操作有两种格式，如下所示：

```
<fmt:parseDate value="expression" type="field"
               dateStyle="style" timeStyle="style"
               pattern="expression" var="name"
               timeZone=" expression"
               parseLocale="expression" scope="scope"/>
<fmt:parseDate type="field" dateStyle="style"
               timeStyle="style"
               pattern="expression" timeZone="expression"
               var="name"  parseLocale="expression"
               scope="scope">
     body content
</fmt:parseDate>
```

对于第一种格式，只有 value 属性才是必需的，它的值应当是指定日期、时间或这两者组合的字符串。对于第二种格式，没有必需的属性，表示要解析的值的字符串被指定为 <fmt:parseDate> 标记必需的标记体内容。

type 、 dateStyle 、 timeStyle 、 pattern 和 timeZone 属性对 <fmt:parseDate> 和对<fmt:formatDate> 起一样的作用，不同

之处仅在于对于前者，它们控制日期值的解析而非显示。parseLocale 属性用来指定一种语言环境，将根据这种语言环境来解析该标记的值，它应当是语言环境的名称或 Locale 类的实例。var 和 scope 属性用来指定限定了作用域的变量（作为 <fmt:parseDate> 的结果），将把Date 对象赋给该变量。如果没有给出 var 属性，则使用 Date 类的 toString() 方法将结果写到 JSP 页面中。清单 2 显示了 <fmt:parseDate> 操作的一个示例：

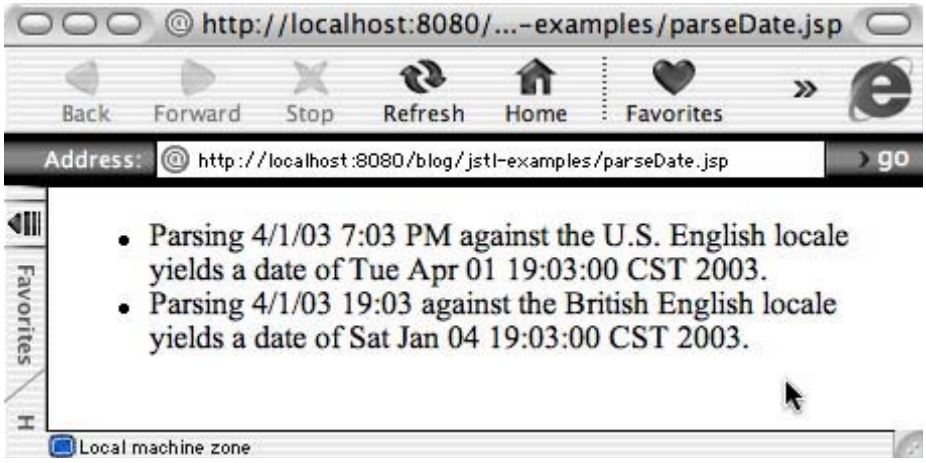
清单 2. 使用 <fmt:parseDate> 标记来解析日期和时间

```
<c:set var="usDateString">4/1/03 7:03 PM</c:set>
<fmt:parseDate value="${usDateString}" parseLocale="en_US"
               type="both" dateStyle="short"
               timeStyle="short"var="usDate"/>
<c:set var="gbDateString">4/1/03 19:03</c:set>
<fmt:parseDate value="${gbDateString}" parseLocale="en_GB"
               type="both" dateStyle="short"
               timeStyle="short" var="gbDate"/>
<ul>
<li> Parsing <c:out value="${usDateString}"/> against
     theU.S. English locale yields a date of
     <c:out value="${usDate}"/>.
</li>
<li> Parsing <c:out value="${gbDateString}"/> against the
```

```
    British English locale yields a date of
    <c:out value="${gbDate}"/>.
</li>
</ul>
```

清单 2 的输出如图 4 所示。

图 4. 清单 2 的输出



由 <fmt:parseDate> 所执行的解析非常严格，注意这一点很重要。正如清单 2 所暗示的那样，要解析的值必须严格符合特定（特定于语言环境）的样式或模式。这当然更加受限制。另一方面，数据的解析并不是一个非常适合于表示层的任务。对于生产代码，文本输入的验证和转换最好由后端代码(比如 servlet)来处理，而不是通过 JSP 定制标记来处理。

用于数字标记

就象 <fmt:formatDate> 和 <fmt:parseDate> 标记用于格式化和解析日期一样， <fmt:formatNumber> 和 <fmt:parseNumber> 标记对数字数据执行类似的功能。

〈fmt:formatNumber〉标记用来以特定于语言环境的方式显示数字数据，包括货币和百分数。〈fmt:formatNumber〉操作由语言环境确定，例如，使用句点还是使用逗号来定界数字的整数和小数部分。下面是它的语法：

```
<fmt:formatNumber value="expression" type="type"
                  pattern="expression"
                  currencyCode="expression"
                 currencySymbol="expression"
                  maxIntegerDigits="expression"
                  minIntegerDigits="expression"
                  maxFractionDigits="expression"
                  minFractionDigits="expression"
                  groupingUsed="expression"
                  var="name" scope="scope"/>
```

如〈fmt:formatDate〉的情况一样，只有 value 属性才是必需的。它用来指定将被格式化的数值。 var 和 scope 属性对〈fmt:formatNumber〉操作所起的作用，如它们在〈fmt:formatDate〉中所起的作用一样。type 属性的值应当是 number 、 currency 或 percentage ，并指明要对哪种类型的数值进行格式化。该属性的缺省值是 number 。 pattern 属性优先于 type 属性，允许对遵循 java.text.DecimalFormat 类模式约定的数值进行更精确的格式化。当 type 属性的值为 currency 时， currencyCode 属性可以用来显

式地指定所显示的数值的货币单位。与语言和国家或地区代码一样，货币代码也是由 ISO 标准管理的（请参阅参考资料以获取所有有效的 ISO 货币符号代码的链接）。该代码用来确定作为已格式化值的一部分显示的货币符号。另外，您可以使用 currencySymbol 属性来显式地指定货币符号。请注意，由于 JDK 1.4和相关的 java.util.Currency 类的引入，〈fmt:formatNumber〉操作的 currencyCode 属性优先权超过 currencySymbol 属性。但是对于较老版本的 JDK 而言， currencySymbol属性具有优先权。

maxIntegerDigits 、minIntegerDigits 、maxFractionDigits 和 minFractionDigits 属性用来控制小数点前后所显示的有效数字的个数。这些属性要求是整数值。groupingUsed 属性带有布尔值并控制是否要对小数点前面的数字分组。例如，在英语语言环境中，将较大数的每三个数字分为一组，每组用逗号定界。其它语言环境用句点或空格来定界这样的分组。该属性的缺省值为 true 。

清单 3 显示了一个简单的货币示例，它本身是清单 1 的扩展。在本例中，不指定 currencyCode 或 currencySymbol 属性。而货币是由语言环境设置确定的。

清单 3. 使用 <fmt:formatNumber> 标记显示货币值

```
<table>
   <fmt:timeZone value="US/Eastern">
    <c:forEach items="${entryList}" var="blogEntry"
                varStatus="status">
```

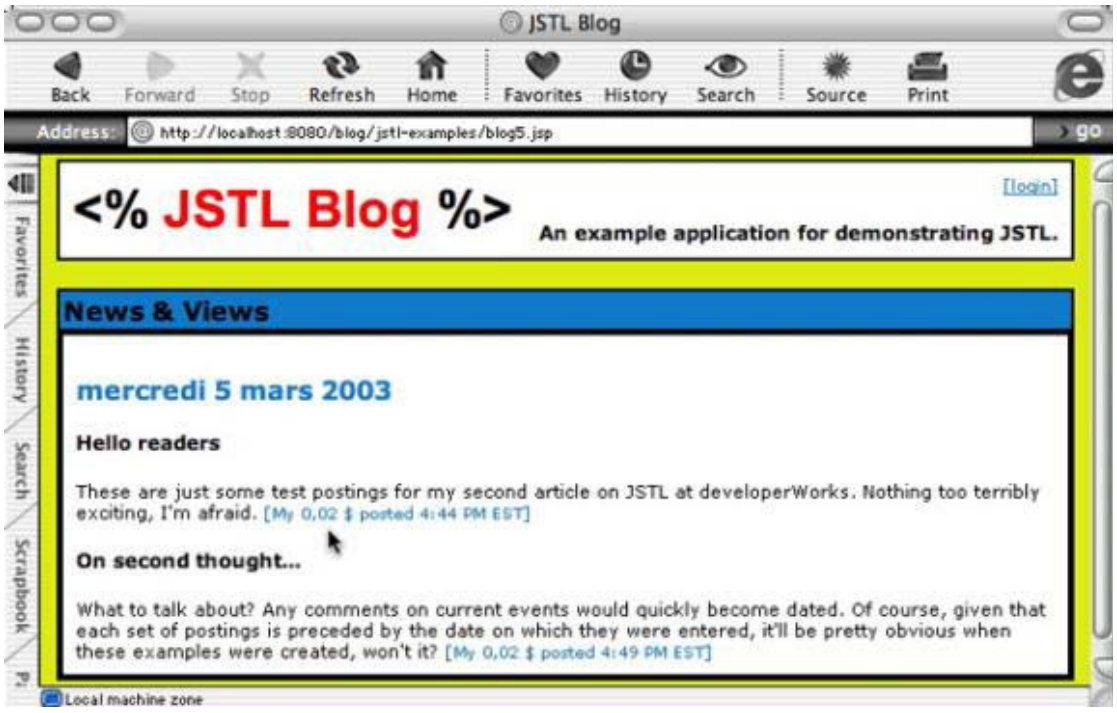
```
        <c:if test="${status.first}">
          <tr><td align="left" class="blogDate">
         <fmt:formatDate value="${blogEntry.created}"
                         dateStyle="full"/>
          </td></tr>
        </c:if>
        <tr><td align="left" class="blogTitle">
        <c:out value="${blogEntry.title}"
               escapeXml="false"/>
        </td></tr>
        <tr><td align="left" class="blogText">
        <c:out value="${blogEntry.text}" escapeXml="false"/>
        <font class="blogPosted">
         [My <fmt:formatNumber value="0.02"
                type="currency"/>posted at
                <fmt:formatDatevalue="${blogEntry.created}"
                pattern="h:mm a zz"/>]
       </font>
       </td></tr>
     </c:forEach>
   </fmt:timeZone>
</table>
```

en_US 语言环境的输出如图 5 所示：

图 5. 清单 3 的 en_US 语言环境的输出

fr_CA 语言环境的输出如图 6 所示：

图 6. 清单 3 的 fr_CA 语言环境的输出



〈fmt:parseNumber〉操作解析了一个数值，该数值是通过 value 属性或该操作的标记体内容以特定于语言环境的方式提供的，将结果

作为 java.lang.Number 类的实例返回。 type 和 pattern 属性对<fmt:parseNumber> 和对 <fmt:formatNumber> 起一样的作用。同样， parseLocale 、 var 和 scope 属性对 <fmt:parseNumber> 起与 <fmt:parseDate> 一样的作用。

```
<fmt:parseNumber value="expression" type="type"
                 pattern="expression"
parseLocale="expression"
                 integerOnly="expression"
                 var="name" scope="scope"/>
<fmt:parseNumber type="type" var="name" scope="scope"
                 pattern="expression"
parseLocale="expression"
                 integerOnly="expression"/>
     body content
</fmt:parseNumber>
```

先前有关 <fmt:parseDate> 的说明同样适用于<fmt:parseNumber>。

解析数据并不是一项非常适合于表示层的任务。如果解析和验证数据作为应用程序业务逻辑的一部分实现，那么软件维护将会得到简化。由于这个原因，通常建议大家在产品 JSP 页面中避免同时使用 <fmt:parseDate> 和 <fmt:parseNumber> 。只有 integerOnly 属性才<fmt:parseNumber> 所独有的。该属性获取一个布尔值，指出是否应当只解析所给值的整数部分。如果该属性的值为 true ，那么就忽

略要被解析的字符串中跟在小数点后面的任何数字。该属性的缺省值为 false 。

用于消息标记

在 JSTL 中用 <fmt:message> 标记实现文本的本地化。该标记允许您从特定于语言环境的资源束中检索文本消息并显示在 JSP 页面上。而且，由于该操作利用 java.text.MessageFormat 类所提供的功能，所以可以将参数化的值替换进这样的文本消息，以便动态地定制本地化内容。用于存储特定于语言环境消息的资源束采用类或特性文件的形式，这些类或特性文件符合标准命名约定，在这种命名约定中基名和语言环境名组合在一起。例如，请研究名为 Greeting.properties 的特性文件，它驻留在我们的 weblog 应用程序的类路径中，该类路径位于与 com.taglib.weblog 包相对应的子目录中。您可以通过在同一目录下指定两个新的特性文件，从而将该特性文件所描述的资源束本地化为英语和法语，通过追加相应的语言代码来命名。具体而言，这两个文件应当分别命名为 Greeting_en.properties 和 Greeting_fr.properties 。如果希望另一个本地化为加拿大法语，您可以引入第三个特性文件，在其名称中包含了相应的国家或地区代码（比如Greeting_fr_CA.properties）。这些文件都可以定义相同的特性，但是应当将这些特性的值定制成对应的语言或方言。这种方法如清单 4 和清单 5 所示，它们给出了 Greeting_en.properties 和 Greeting_fr.properties 文件的样本

内容。在这些示例中，定义了两个已本地化的消息。它们可以通过 com.taglib.weblog.Greeting.greeting 和 com.taglib.weblog.Greeting.return 键识别。但是已经将与这些键相关联的值本地化为文件名中所确定的语言。请注意，出现在 com.taglib.weblog.Greeting.greeting 消息的两个值中的 {0} 模式使已参数化的值能够在内容生成期间动态地插入到消息中。

清单 4. Greeting_en.properties 本地化资源束的内容

```
com.taglib.weblog.Greeting.greeting=Hello {0}, and welcome
to the JST L Blog.
com.taglib.weblog.Greeting.return=Return
```

清单 5. Greeting_fr.properties 本地化资源束的内容

```
com.taglib.weblog.Greeting.greeting=Bonjour {0},
et bienvenue au JSTLBlog.
com.taglib.weblog.Greeting.return=Retournez
```

用 JSTL 显示这样的本地化内容，第一步就是指定资源束。 fmt 库为完成这一任务提供了两个定制标记： <fmt:setBundle> 和 <fmt:bundle> ，它们的行为和前面介绍的 <fmt:setTimeZone> 和 <fmt:timeZone> 标记相似。<fmt:setBundle> 操作设置了一个缺省资源束，供 <fmt:message> 标记在特定作用域内使用，而 <fmt:bundle> 指定了为嵌套在其标记体内容中的全部和任意 <fmt:message> 操作所用的资源束。下面的代码片段显示了 <fmt:setBundle> 标记的语法。 basename 属性是必需的，它标识了设为缺省值的资源束。请注意， basename 属性的值不应当包含任何

本地化后缀或文件扩展名。清单 4 和清单 5 中给出的示例资源束的基名为 com.taglib.weblog.Greeting 。

```
<fmt:setBundle basename="expression" var="name"
                scope="scope"/>
```

可选的 scope 属性指明缺省资源束设置所应用的 JSP 作用域。如果没有显式地指定该属性，就假定为 page 作用域。如果指定了可选的 var 属性，那么将把由 basename 属性所标识的资源束赋给该属性值所命名的变量。在这种情况下， scope 属性指定变量的作用域；没有将缺省资源束赋给相应的 JSP 作用域。您使用 <fmt:bundle> 标记（其语法如下所示）在其标记体内容的作用域内设置缺省资源束。和 <fmt:setBundle> 一样，只有basename 属性才是必需的。您可以使用可选的 prefix属性来为任何嵌套的 <fmt:message> 操作的 key 值指定缺省前缀。

```
<fmt:setBundle basename="expression" var="name"
               scope="scope"/>
```

一旦设置了资源束，真正起到显示本地化消息作用的是 <fmt:message> 标记。该操作支持两种不同的语法，这取决于是否需要任何嵌套的 <fmt:param> 标记：

```
<fmt:message key="expression"  bundle="expression"
             var="name" scope="scope"/>
<fmt:message key="expression" bundle="expression"
             var="name" scope="scope">
```

```
<fmt:param value="expression"/>
...
</fmt:message>
```

对于 <fmt:message> ，只有 key 属性才是必需的。 key 属性的值用来确定要显示在资源束中定义的哪些消息。您可以使用 bundle 属性来指定一个显式的资源束，用来查找由 key 属性标识的消息。请注意，该属性的值必须是实际的资源束，比如当指定<fmt:setBundle>操作的 var 属性时由该操作所赋予的资源束。 <fmt:message> 的 bundle 属性不支持字符串值（比如 <fmt:bundle><fmt:setBundle> 的 basename 属性）。如果指定了 <fmt:message> 的 var 属性，那么将由该标记所生成的文本消息赋给指定的变量，而不是写到 JSP 页面。通常，可选的 scope 属性用来指定由 var 属性指定的变量的作用域。需要的时候您可以通过使用 <fmt:param> 标记的 value 属性来提供文本消息的参数化值。或者，可以将该值指定为 <fmt:param> 标记体内容，在这种情况下省略该属性。无论参数化值模式出现在消息文本中的什么地方，由 <fmt:param> 标记指定的值都将合并到从资源束检索的消息，这java.text.MessageFormat 类的行为一致。因为参数化值可以通过其下标进行标识，因此嵌套的 <fmt:param> 标记的顺序很重要。<fmt:bundle> 、 <fmt:message> 和 <fmt:param> 标记的交互作用如清单 6 所示。此处<fmt:bundle> 标记通过两个嵌套的 <fmt:message> 标记指定了要在其中检索本地化消息的资源束。这两个 <fmt:message> 标记的第一个对应于带有

一个参数化值的消息，还出现了对应的用于该值的〈fmt:param〉标记。

清单 6. 使用 <fmt:message> 标记显示本地化消息

```
<fmt:bundle basename="com.taglib.weblog.Greeting">
<fmt:message key="com.taglib.weblog.Greeting.greeting">
<fmt:param value="${user.fullName}"/>
</fmt:message>
<br>
<center>
 <a href="<c:url value='/index.jsp'/>">
   <fmt:messagekey="com.taglib.weblog.Greeting.return"/>
 </a>
</center>
</fmt:bundle>
```

清单 7 演示了〈fmt:bundle〉的 prefix 属性的用法；为 prefix 属性提供的值在嵌套的〈fmt:message〉操作中自动地预先添加到所有 key 值上。因此清单 7 相当于清单 6，只是清单 7 利用了这一便利的特性，使得能够在两个〈fmt:message〉标记中使用缩略的key值。

清单 7. <fmt:bundle> 的 prefix 属性对 <fmt:message> 标记的影响

```
<fmt:bundle basename="com.taglib.weblog.Greeting"
            prefix="com.taglib.weblog.Greeting.">
<fmt:message key="greeting">
```

```
<fmt:param value="${user.fullName}"/>
</fmt:message>
<br>
 <center>
   <a href="<c:url value='/index.jsp'/>">
      <fmt:message key="return"/>
   </a>
 </center>
</fmt:bundle>
```

图 7 和图 8 演示了正在工作的 fmt 库与消息相关的标记，显示了由清单 7 中代码所产生的输出，以及清单 4 和清单 5 中的本地化资源束。图 7 显示了当浏览器首选项为英语语言环境时的结果。

图 7. 清单 7 中 en_US 语言环境的输出

图 8 显示了指定法语的语言环境的输出。

图 8. 清单 7 的 fr_CA 语言环境的输出

Xml 标签

根据设计，XML 提供灵活的方式来表示结构化数据，这些数据同时准备进行验证，因此它

尤其适应于在松散联合的系统之间交换数据。这反过来使其成为Web 应用程序极具吸引力

的集成技术。

与使用 XML 表示的数据进行交互的第一步是把数据作为一个XML 文件，对其进行检索并进行分解，以创建数据结构来接入该文件中的内容。在分解文件后，您可以有选择的对其进行转换以创建新的XML 文件，您可以对新的XML 文件进行相同的操作。最终，文件中的

数据可以被提取，然后显示或使用作为输入数据来运行其它操作。

这些步骤都在用于控制 XML 的JSTL 标记中反映出。根据我们在第2部分探讨核心中所讨论的，我们使用core 库中的<c:import>标记来检索XML 文件。然后使用<x:parse>标记来分解该文件，支持标准的XML 分解技术，如文件对象模式（Document Object Model，D

OM)和简单XML API（Simple API for XML，SAX)。<x:transform>标记可用于转换XML文件并依赖标准技术来转换XML 数据：扩展样式表语言（ Extensible Stylesheet Language，XSL)。最后，我们提供多个标记来接入和控制分解后的XML 数据，但是所有这一切都依赖于另一种标准- XML 路径语言（XML Path Language，XPath)，以引用分解后的XML 文件中的内容。

分解 XML

<x:parse>标记有多种格式，取决于用户希望的分解类型。这一项操作最基本的格式使用以下语法：

```
<x:parse xml="expression" var="name" scope="scope"
         filter="expression" systemId="expression"/>
```

在这五种属性中，只有xml 属性是需要的，其值应该是包含要分解的XML 文件的字符串，或者是java.io.Reader 实例，通过它可以读取要被分解的文件。此外，您可以使用以下语法，根据<x:parse>标记的主体内容来规定要被分解的文件：

```
<x:parse var="name" scope="scope"filter="expression"
         systemId="expression">
        body content
</x:parse>
```

var 和scope 属性规定存储分解后的文件的scoped 变量。然后xml 库中的其它标记可以使用这一变量来运行其它操作。注意，当var 和

scope 属性存在时，JSTL 用于表示分解后的文件的数据结构类型以实施为导向，从而厂商可以对其进行优化。如果应用程序需要对 JSTL 提供的分解后的文件进行处理，它可以使用另一种格式的<x:parse>，它要求分解后的文件坚持使用一个标准接口。在这种情况下，该标记的语法如下：

```
<x:parse xml="expression" varDom="name" scopeDom="scope"
         filter="expression" systemId="expression"/>
```

当您使用<x:parse>的这一版本时，表示分解后的XML 文件的对象必须使用org.w3c.dom.Document 接口。当根据<x:parse>中的主体内容来规定XML 文件时，您还可以使用varDom 和scopeDom 属性来代替 var 和 scope 属性，语法如下：

```
<x:parse varDom="name" scopeDom="scope" filter="expression"
         systemId="expression">
    body content
</x:parse>
```

其它两个属性 filter 和 systemId 可以实现对分解流程的精确控制。filter 属性规定org.xml.sax.XMLFilter 类的一个实例，以在分解之前对文件进行过滤。如果要被分解的文件非常大，但目前的工作只需要处理一小部分内容时这一属性尤其有用。systemId 属性表示要被分解的文件的URI 并解析文件中出现的任何相关的路径。当被分解的XML 文件使用相关的URL 来引用分解流程中需要接入的其它文件或资源时需要这种属性清单 1 展示了<x:parse> 标记的使用，

包括与 <c:import>的交互。此处<c:import> 标记用于检索众所周知的Slashdot Web 网站的RDF Site Summary (RSS)反馈，然后使用<x:parse>分解表示RSS 反馈的XML 文件，表示分解后的文件的以实施为导向的数据结构被保存到名为rss 的变量（带有page 范围）中。

清单 1：<x:parse>与<c:import>的交互

```
<c:import var="rssFeed"
          url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="${rssFeed}"/>
```

转换 XML

XML 通过XSL 样式表来转换。JSTL 使用<x:transform>标记来支持这一操作。与<x:parse>的情况一样，<x:transform> 标记支持多种不同的格式。<x:transform> 最基本的格式的语法是：

```
<x:transform xml="expression" xslt="expression"var="name"
             scope="scope" xmlSystemId="expression"
             xsltSystemId="expression">
<x:param name="expression" value="expression"/>
...
</x:transform>
```

此处，xml 属性规定要被转换的文件，xslt 属性规定定义这次转换的样式表。这两种属性是必要的，其它属性为可选。与<x:parse>的xml 属性一样，<x:transform>的xml 属性值可以是包含XML文件的字

符串，或者是接入这类文件的Reader。此外，它还可以采org.w3c.dom.Document 类或javax.xml.transform.Source 类的实例格式。最后，它还可以是使用<x:parse> 操作的var 或varDom 属性分配的变量值。而且，您可以根据<x:transform> 操作的主体内容来包含要被转换的XML 文件。在这种情况下，<x:transform> 的语法是：

```
<x:transform xslt="expression" var="name" scope="scope"
             xmlSystemId="expression"
             xsltSystemId="expression">
body content
<x:param name="expression" value="expression"/>
...
</x:transform>
```

在这两种情况下，规定XSL 样式表的xslt 属性应是字符串、Reader 或javax.xml.transform.Source 实例。如果 var 属性存在，转换后的XML 文件将分配给相应的scoped 变量，作为org.w3c.dom.Document 类的一个实例。通常,scope 属性规定这类变量分配的范围。<x:transform> 标记还支持将转换结果存储javax.xml.transform.Result 类的一个实例中，而不是作为org.w3c.dom.Document 的一个实例。如果var 和 scope 属性被省略，result对象规定作为result属性的值，<x:transform>标记将使用该对象来保存应用该样式表的结果。清单2 中介绍了使用<x:transform> 的result 属性的这两种语法的变化：

清单 2：使用result 属性来提供javax.xml.transform.Result 实例时，<x:transform>操作的语法变化

```
<x:transform xml="expression" xslt="expression"
             result="expression" xmlSystemId="expression"
             xsltSystemId="expression">
<x:param name="expression" value="expression"/>
...
</x:transform>
<x:transform  xslt="expression" result="expression"
              xmlSystemId="expression"
              xsltSystemId="expression">
   body content
<x:param name="expression" value="expression"/>
...
</x:transform>
```

无论您采用这两种<x:transform>格式中的那一种，您都必须从定制标记单独创建javax.xml.transform.Result 对象。该对象自身作为result 属性的值提供。如果既不存在 var 属性，也不存在result 属性，转换的结果将简单地插入到JSP 页面，作为处理<x:transform>操作的结果。当样式表用于将数据从XML转换成HTML时尤其有用，如清单3 所示：

```
<c:import var="rssFeed"
          url="http://slashdot.org/slashdot.rdf"/>
```

```
<c:import var="rssToHtml"
          url="/WEB-INF/xslt/rss2html.xsl"/>
<x:transform xml="${rssFeed}" xslt="${rssToHtml}"/>
```

在本例中，使用 <c:import> 标记来读取RSS 反馈和适当的样式表。样式表的输出结果是HTML，通过忽略<x:transform>的var 和result 属性来直接显示。图1 显示了实例结果：

图 1：清单3 的输出结果

与<x:parse>的systemId 属性一样，<x:transform>的xmlSystemId 和 xsltSystemId 属性用于解析XML 文件中相关的路径。在这种情况下，xmlSystemId 属性应用于根据标记的 xml 属性值提供的文件，而xsltSystemId 属性用于解析根据标记的xslt 属性规定的样式表中的相关路径。如果正在推动文件转换的样式表使用了参数，我们使用<x:param> 标记来规定这些参数。如果参数存在，那么这些标记必须在<x:transform> 标记主体内显示。如果根据主体内容规定了要被转换的XML 文件，那么它必须先于任何 <x:param> 标记。<x:param> 标记有两种必要的属性 -- name 和 value -- 就象本系列第 2 部分

和 第 3部分中讨论的<c:param> 和 <fmt:param> 标记一样。

处理 XML 内容

XML 文件的分解和转换操作都是基于整个文件来进行。但是，在您将文件转换成一种可用的格式之后，一项应用程序通常只对文件中包含的一部分数据感兴趣。鉴于这一原因，xml库包括多个标记来接入和控制XML 文件内容的各个部分。如果您已经阅读了本系列第 2 部分 (探讨核心)，您将对这些xml 标记的名字非常熟悉。它们都基于JSTL core 库相应的标记。但是，这些core 库标记使用EL 表达式，通过它们的value 属性来接入JSP 容器中的数据，而它们在xml 库中的副本使用XPath 表达式，通过select 属性接入XML 文件中的数据。XPath 是引用XML 文件中元素及它们的属性值和主体内容的标准化符号。正如其名字代表的一样，这种符号与文件系统路径的表示方法类似，使用短斜线来分开XPath 语句的组分。这些组分对映于XML 文件的节点，连续的组分匹配嵌套的Element。此外，星号可以用于作为通配符来匹配多个节点，括号内的表达式可以用于匹配属性值和规定索引。有多种在线参考资料介绍XPath 和它的使用(见参考资料)。要显示 XML 文件的数据的一个Element，使用<x:out> 操作，它与core 库的<c:out> 标记类似。但是，<c:out> 使用名为value 和 escapeXml 的属性，<x:out> 的属性为select和escapeXml：

```
<x:out select="XPathExpression" escapeXml="boolean"/>
```

当然，两者的区别在于<x:out> 的select 属性值必须是XPath 表达

式，而<c:out> 的value 属性必须是EL 表达式。两种标记escapeXml 属性的意义是相同的。清单 4 显示了<x:out> 操作的使用。注意，规定用于select 属性的XPath 表达式由一个EL 表达式规定scoped 变量，尤其是$rss。这一EL 表达式根据将被求值的XPath 语句来识别分解后的XML 文件。该语句在此处查找名为title 且父节点名为channel 的Element，从而选择它找到的第一个Element(根据表达式尾部[1]索引规定)。这一<x:out> 操作的结果是显示这一Element 的主体内容，关闭正在转义（Escaping）的XML 字符。

清单 4：使用<x:out>操作来显示XML Element 的主体内容

```
<c:import var="rssFeed"
          url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="${rssFeed}"/>
<x:out
   select="$rss//*[name()='channel']/*[name()='title'][1]"
   escapeXml="false"/>
```

除了<x:out>之外，JSTL xml 库包括以下控制XML 数据的标记：

- <x:set> ，向JSTL scoped 变量分配XPath 表达式的值
- <x:if> ，根据XPath 表达式的布尔值来条件化内容
- <x:choose>、<x:when>和<x:otherwise>，根据XPath 表达式来实施互斥的条件化
- <x:forEach> ，迭代根据XPath 表达式匹配的多个Elements

每个这些标记的操作与 core 库中相应的标记类似。例如，清单5 中显示的<x:forEach>的使

用， <x:forEach> 操作用于迭代XML 文件中表示RSS 反馈的所有名为item 的Element。

注意，<x:forEach>主体内容中嵌套的两个<x:out> 操作中的XPath 表达式与<x:forEach>标记正在迭代的节点相关。它们用于检索每个item element 的子节点link 和 title。

清单 5：使用<x:out> 和<x:forEach>操作来选择和显示XML 数据

```
<c:import var="rssFeed"
          url="http://slashdot.org/slashdot.rdf"/>
<x:parse var="rss" xml="${rssFeed}"/>
<a href="<x:out
    select="$rss//*[name()='channel']/*[name()='link'][1]
"/>"><x:out
select="$rss//*[name()='channel']/*[name()='title'][1]"
escapeXml="false"/></a>
<x:forEach select="$rss//*[name()='item']">
<li> <a href="<x:out select="./*[name()='link']"/>">
<x:out select="./*[name()='title']" escapeXml="false"/>
</a>
</x:forEach>
```

清单 5 中JSP 程序代码的输出结果与清单 3 类似，它在图 1 中显示。xml 库以XPath 为导向的标记提供备选的样式表来转换XML 内容，尤其是当最后的输出结果是HTML 的情况。

sql 标签

建立数据源

正如其名字代表的一样，该库提供与关系数据库交互的标记。尤其是sql 库定义规定数据源、发布查询和更新以及将查询和更新编组到事务处理中的标记。Datasource 是获得数据库连接的工厂。它们经常实施某些格式的连接库来最大限度地降低与创建和初始化连接相关的开销。Java 2 Enterprise Edition (J2EE)应用程序服务器通常内置了Datasource 支持，通过 Java 命名和目录接口（ Java Naming and Directory Interface，JNDI)它可用于J2EE 应用程序。JSTL 的sql 标记依赖于Datasource 来获得连接。实际上包括可选的dataSource 属性以明确规定它们的连接工厂作为javax.sql.DataSource 接口实例，或作为JNDI 名。您可以使用<sql:setDataSource>标记来获得javax.sql.DataSource 实例，它采用以下两种格式：

```
<sql:setDataSource dataSource="expression" var="name"
                   scope="scope"/>
<sql:setDataSource url="expression" driver="expression"
                   user="expression" password="expression"
                   var="name" scope="scope"/>
```

第一种格式只需要 dataSource 属性，而第二种格式只需要url 属性。通过提供 JNDI 名作为dataSource 属性值，您可以使用第一种格式来接入与JNDI 名相关的datasource。第二种格式将创建新的

datasource，使用作为url 属性值提供的JDBC URL。可选的driver 属性规定实施数据库driver 的类的名称，同时需要时user 和password 属性提供接入数据库的登录证书。对<sql:setDataSource>的任何一种格式而言，可选的var 和 scope 属性向scoped 变量分配特定的datasource。如果var 属性不存在，那么 <sql:setDataSource> 操作设置供sql 标记使用的缺省 datasource，它们没有规定明确的datasource。您还可以使用javax.servlet.jsp.jstl.sql.dataSource 参数来配置sql 库的缺省datasource。在实践中，在应用程序的Web.xml 文件中添加与清单6 中显示的类似的程序代码是规定缺省datasource 最方便的方式。使用<sql:setDataSource> 来完成这一操作要求使用JSP 页面来初始化该应用程序，因此可以以某种方式自动运行这一页面。

清单 6：使用JNDI 名来设置JSTL 在web.xml 部署描述符中的缺省datasource

```
<context-param>
<param-name>
        javax.servlet.jsp.jstl.sql.dataSource
</param-name>
<param-value>jdbc/blog</param-value>
</context-param>
```

提交查询和更新

在建立了 datasource 接入之后，您可以使用<sql:query> 操作来执行查询，同时使用<sql:update> 操作来执行数据库更新。查询

和更新使用SQL 语句来规定，它可以使用基于JDBC 的java.sql.PreparedStatement 接口的方法来实现参数化。参数值使用嵌套的<sql:param> 和 <sql:dateParam> 标记来规定。支持以下三种<sql:query> 操作：

```
<sql:query sql="expression" dataSource="expression"
           var="name" scope="scope" maxRows="expression"
           startRow="expression"/>
<sql:query sql="expression" dataSource="expression"
           var="name" scope="scope" maxRows="expression"
           startRow="expression">
<sql:param value="expression"/>
...
</sql:query>
<sql:query dataSource="expression" var="name" scope="scope"
           maxRows="expression" startRow="expression">
   SQL statement
<sql:param value="expression"/>
...
</sql:query>
```

前两种格式只要求 sql 和 var 属性，第三种格式只要求var 属性。var 和 scope 属性规定存储查询结果的scoped 变量。maxRows 属性可以用于限制查询返回的行数，startRow 属性允许忽略一些最开始

的行数(如当结果集（Result set）由数据库来构建时忽略)。在执行了查询之后，结果集被分配给scoped 变量，作为javax.servlet.jsp.jstl.sql.Result 接口的一个实例。这一对象提供接入行、列名称和查询的结果集大小的属性，如表1 所示：

表 1：javax.servlet.jsp.jstl.sql.Result 接口定义的属性

属性	描述
rows	一排 SortedMap 对象，每个对象对映列名和结果集中的单行
rowsByIndex	一排数组，每个对应于结果集中的单行
columnNames	一排对结果集中的列命名的字符串，采用与 rowsByIndex 属性相同的顺序
rowCount	查询结果中总行数
limitedByMaxRows	如果查询受限于maxRows 属性值为真

在这些属性中，rows 尤其方便，因为您可以使用它来在整个结果集中进行迭代和根据名字访问列数据。我们在清单 7 中阐述了这一操作，查询的结果被分配到名为queryResults 的scoped 变量中，然后使用core 库中的<c:forEach>标记来迭代那些行。嵌套的<c:out> 标记利用EL 内置的Map 收集支持来查找与列名称相对应的行数据。(记得在第 1 部分，${row.title}和${row["title"]} 是相等的表达式。)清单 7 还展示了使用<sql:setDataSource> 来关联datasource 和scoped 变量，它由<sql:query> 操作通过其dataSource 属性随后接入。

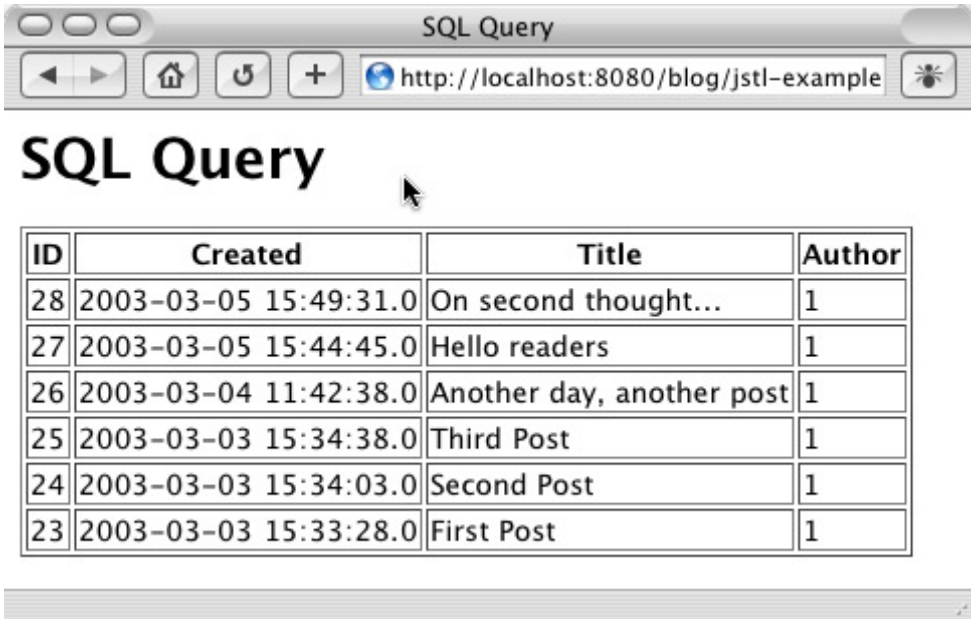
清单 7：使用<sql:query>来查询数据库，使用<c:forEach>来迭代整个结果集

```
<sql:setDataSource var="dataSrc" url="jdbc:mysql:///taglib"
                   driver="org.gjt.mm.mysql.Driver"
                   user="admin" password="secret"/>
<sql:query var="queryResults" dataSource="${dataSrc}">
```

```
          select * from blog group by created desc limit ?
   <sql:param value="${6}"/>
</sql:query>
<table border="1">
<tr>
  <th>ID</th>
  <th>Created</th>
  <th>Title</th>
  <th>Author</th>
</tr>
<c:forEach var="row" items="${queryResults.rows}">
<tr>
<td><c:out value="${row.id}"/></td>
<td><c:out value="${row.created}"/></td>
<td><c:out value="${row.title}"/></td>
<td><c:out value="${row.author}"/></td>
</tr>
</c:forEach>
</table>
```

图 2 显示了清单7 中JSTL 程序代码的实例页面输出结果。注意：清单7 中<sql:query>操作主体中出现的SQL 语句为参数化语句。

图 2：清单7 的输出



在<sql:query> 操作中，SQL 语句根据主体内容来规定，或者使用？字符，通过sql 属性实现参数化。对于SQL 语句中每个这样的参数来说，应有相应的<sql:param> 或 <sql:dateParam> 操作嵌套到 <sql:query> 标记的主体中。<sql:param> 标记只采用一种属性 --value --来规定参数值。此外，当参数值为字符串时，您可以忽略 value 属性并根据<sql:param> 标记的主体内容来提供参数值。表示日期、时间或时间戳的参数值使用<sql:dateParam> 标记来规定，使用以下语法：

```
<sql:dateParam value="expression" type="type"/>
```

对于<sql:dateParam>来说，value 属性的表达式必须求 java.util.Date 类实例的值，同时type 属性值必须是date、time 或timestamp，由SQL 语句需要那类与时间相关的值来决定。与<sql:query> 一样，<sql:update> 操作支持三种格式：

```
<sql:update sql="expression" dataSource="expression"
```

```
            var="name" scope="scope"/>
<sql:update sql="expression" dataSource="expression"
            var="name" scope="scope">
<sql:param value="expression"/>
...
</sql:update>
<sql:update dataSource="expression"var="name"
            scope="scope">
  SQL statement
<sql:param value="expression"/>
...
</sql:update>
```

sql 和dataSource 属性有与<sql:query>相同的<sql:update> 语义。同样，var 和 scope属性可以用于规定scoped 变量，但在这种情况下，分配给scoped 变量的值将是java.lang.Integer 的一个实例，显示作为数据库更新结果而更改的行数。

事务处理

事务处理用于保护作为一个组必须成功或失败的一系列数据库操作。事务处理支持已经嵌入到JSTL 的sql 库中，通过将相应的<sql:query>和<sql:update>操作嵌套到<sql:transaction>标记的主体内容中，从而将一系列查询和更新操作打包到一个事务处理中也

就显得微不足道了。<sql:transaction> 语法如下：

```
<sql:transaction dataSource="
expression" isolation="
isolationLevel">
<sql:query .../>
or <sql:update .../>
...
```

<sql:transaction> 操作没有必需的属性。如果您忽略了dataSource属性，那么使用JSTL的缺省datasource。isolation 属性用于规定事务处理的隔离级别，它可以是read_committed、read_uncommitted、repeatable_read 或serializable。如果您未规定这一属性，事务处理将使用datasource 的缺省隔离级别。您可能希望所有嵌套的查询和更新必须使用与事务处理相同的 datasource。实际上，嵌套到<sql:transaction>操作中的<sql:query> 或 <sql:update> 不能用于规定dataSource 属性。它将自动使用与周围<sql:transaction>标记相关的datasource（显性或隐性）。清单 8 显示了如何使用<sql:transaction> 的一个实例：

清单：使用<sql:transaction>来将数据库更新联合到事务处理中

```
<sql:transaction>
<sql:update sql="update blog set title = ? where id = ?">
<sql:param value="New Title"/>
<sql:param value="${23}"/>
```

```
</sql:update>
<sql:update
  sql="update blog set last_modified = now() where id = ?">
<sql:param value="${23}"/>
</sql:update>
</sql:transaction>
```

Functions 标签

称呼 Functions 标签库为标签库，倒不如称呼其为函数库来得更容易理解些。因为 Functions 标签库并没有提供传统的标签来为 JSP 页面的工作服务，而是被用于 EL 表达式语句中。在 JSP2.0 规范下出现的 Functions 标签库为 EL 表达式语句提供了许多更为有用的功能。 Functions 标签库分为两大类，共 16 个函数。长度函数：fn:length ；字符串处理函数：fn:contains 、 fn:containsIgnoreCase 、 fn:endsWith 、 fn:escapeXml 、 fn:indexOf 、 fn:join 、 fn:replace 、 fn:split 、 fn:startsWith 、 fn:substring 、 fn:substringAfter 、 fn:substringBefore 、 fn:toLowerCase 、 fn:toUpperCase 、 fn:trim 以下是各个函数的用途和属性以及简单示例。

长度函数 fn:length 函数

长度函数 fn:length 的出现有重要的意义。在 JSTL1.0 中，有一个功能被忽略了，那就是对集合的长度取值。虽然 java.util.Collection 接口定义了 size 方法，但是该方法不是一个标准的 JavaBean 属性方法（没有 get,set 方法），因此，无法通过 EL 表达式“ ${collection.size}”来轻松取得。

fn:length 函数正是为了解决这个问题而被设计出来的。它的参数为 input，将计算通过该属性传入的对象长度。该对象应该为集合类型或 String 类型。其返回结果是一个 int 类型的值。下面看一个示例。

```
<%ArrayList arrayList1 = new ArrayList();
                              arrayList1.add("aa");
                              arrayList1.add("bb");
                              arrayList1.add("cc");
request.getSession().setAttribute("arrayList1",arrayList1)
%>
${fn:length(sessionScope.arrayList1)}
```

假设一个 ArrayList 类型的实例“arrayList1”，并为其添加三个字符串对象，使用 fn:length 函数后就可以取得返回结果为“3”。

判断函数 fn:contains 函数

fn:contains 函数用来判断源字符串是否包含子字符串。它包括 string 和 substring 两个参数，它们都是 String 类型，分布表示源字符串和子字符串。其返回结果为一个 boolean 类型的值。下面看一个示例。

```
${fn:contains("ABC", "a")}<br>
${fn:contains("ABC", "A")}<br>
```

前者返回“ false ”，后者返回“ true ”。

判断函数 fn:containsIgnoreCase

fn:containsIgnoreCase 函数与 fn:contains 函数的功能差不多，唯一的区别是 fn:containsIgnoreCase 函数对于子字符串的包含比较将忽略大小写。它与 fn:contains 函数相同，包括 string 和 substring 两个参数，并返回一个 boolean 类型的值。下面看一个示例。

```
${fn:containsIgnoreCase("ABC", "a")}<br>
${fn:containsIgnoreCase("ABC", "A")}<br>
```

前者和后者都会返回“ true ”。

词头判断函数 fn:startsWith 函数

fn:startsWith 函数用来判断源字符串是否符合一连串的特定词头。它除了包含一个 string 参数外，还包含一个 subffx 参数，表示词头字符串，同样是 String 类型。该函数返回一个 boolean 类型的值。下面看一个示例。

```
${fn:startsWith ("ABC", "ab")}<br>
${fn:startsWith ("ABC", "AB")}<br>
```

前者返回“ false ”，后者返回“ true ”。

词尾判断函数 fn:endsWith 函数

fn:endsWith 函数用来判断源字符串是否符合一连串的特定词尾。它与 fn:startsWith 函数相同，包括 string 和 subffx 两个参数，并返回一个 boolean 类型的值。下面看一个示例。

```
${fn:endsWith("ABC", "bc")}<br>
${fn:endsWith("ABC", "BC")}<br>
```

前者返回“ false ”，后者返回“ true ”。

字符实体转换函数 fn:escapeXml 函数

fn:escapeXml 函数用于将所有特殊字符转化为字符实体码。它只包含一个 string 参数，返回一个 String 类型的值。

字符匹配函数 fn:indexOf 函数

fn:indexOf 函数用于取得子字符串与源字符串匹配的开始位置，若子字符串与源字符串中的内容没有匹配成功将返回“ -1”它包括 string 和 substring 两个参数，返回结果为 int 类型。下面看一个示例。

```
${fn:indexOf("ABCD","aBC")}<br>
${fn:indexOf("ABCD","BC")}<br>
```

前者由于没有匹配成功，所以返回 -1 ，后者匹配成功将返回位置的下标，为 1 。

分隔符函数 fn:join 函数

fn:join 函数允许为一个字符串数组中的每一个字符串加上分隔符，并连接起来。它的参数、返回结果和描述如表 9.25 所示：

表 9.25 fn:join 函数

参数	描述
array	字符串数组。其类型必须为 String[] 类型
separator	分隔符。其类型必须为 String 类型
返回结果	返回一个 String 类型的值

下面看一个示例。

```
<% String[] stringArray = {"a","b","c"}; %>
<%request.getSession().setAttribute("stringArray",stringArray);%>
${fn:join(sessionScope.stringArray,";")}<br>
```

定义数组并放置到 Session 中，然后通过 Session 得到该字符串数组，使用 fn:join 函数并传入分隔符“ ; ”，得到的结果为“ a;b;c ”。

替换函数 fn:replace 函数

fn:replace 函数允许为源字符串做替换的工作。它的参数、返回结果和描述如表 9.26 所示：

表 9.26 fn:replace 函数

参数	描述
inputString	源字符串。其类型必须为 String 类型
beforeSubstring	指定被替换字符串。其类型必须为 String 类型
afterSubstring	指定替换字符串。其类型必须为 String 类型
返回结果	返回一个 String 类型的值

下面看一个示例。

```
${fn:replace("ABC","A","B")}<br>
```

将“ ABC ”字符串替换为“ BBC ”，在“ ABC ”字符串中用“ B ”替换了“ A ”。

分隔符转换数组函数 fn:split 函数

fn:split 函数用于将一组由分隔符分隔的字符串转换成字符串数组。它的参数、返回结果和描述如表 9.27 所示：

表 9.27 fn:split 函数

参数	描述
string	源字符串。其类型必须为 String 类型
delimiters	指定分隔符。其类型必须为 String 类型
返回结果	返回一个 String[] 类型的值

下面看一个示例。

```
${fn:split("A,B,C",",")}<br>
```

将“ A,B,C ”字符串转换为数组 {A,B,C} 。

字符串截取函数 fn:substring 函数

fn:substring 函数用于截取字符串。它的参数、返回结果和描述如表 9.28 所示：

表 9.28 fn:substring 函数

参数	描述
string	源字符串。其类型必须为 String 类型
beginIndex	指定起始下标（值从 0 开始）。其类型必须为 int 类型
endIndex	指定结束下标（值从 0 开始）。其类型必须为 int 类型
返回结果	返回一个 String 类型的值

下面看一个示例。

```
${fn:substring("ABC","1","2")}<br>
```

截取结果为“ B ”。

起始到定位截取字符串函数 fn:substringBefore 函数

fn:substringBefore 函数允许截取源字符从开始到某个字符串它的参数和 fn:substringAfter 函数相同，不同的是 substring 表示的是结束字符串。下面看一个示例。

```
${fn:substringBefore("ABCD","BC")}<br>
```

截取的结果为“ A ”。

小写转换函数 fn:toLowerCase 函数

fn:toLowerCase 函数允许将源字符串中的字符全部转换成小写字符。它只有一个表示源字符串的参数 string ，函数返回一个 String 类型的值。下面看一个示例。

```
${fn:toLowerCase("ABCD")}<br>
```

转换的结果为“ abcd ”。

大写转换函数 fn:toUpperCase 函数

fn:toUpperCase 函数允许将源字符串中的字符全部转换成大写字符。它与 fn:toLowerCase 函数相同，也只有个 参数，并返回一个 String 类型的值。下面看一个示例。

```
${fn:toUpperCase("abcd")}<br>
```

转换的结果为“ ABCD ”。

空格删除函数 fn:trim 函数

fn:trim 函数将删除源字符串中结尾部分的“空格”以产生一个新的字符串。它与 fn:toLowerCase 函数相同，只有个 String 参数，并返回一个 String 类型的值。下面看一个示例。

```
${fn:trim("AB C ")}D<br>
```

转换的结果为“AB CD”，注意，它将只删除词尾的空格而不是全部。

注意事项

JSTL 的xml 和 sql 库使用定制标记，从而能够在JSP 页面上实施复杂的功能，但是在您的表示层实施这类功能可能不是最好的方法。

对于多位开发人员长期编写的大型应用程序来说，实践证明，用户接口、基本的业务逻辑和数据仓库之间的严格分离能够长期简化软件维护。广泛流行的模式/视图/控制器（ Model-View-Controller，MVC）设计模板是这一“最佳实践”的公式化。在J2EE Web 应用程序领域中，模式是应用程序的业务逻辑，视图是包含表示层的JSP 页面。(控制器是form handlers 和其它服务器端机制，使浏览器操作能够开始更改模式并随后更新视图。) MVC 规定应用程序的三个主要elements--模式、视图和控制器 --相互之间有最小的相关性，从而限制相互之间的交互到统一、精心定义的接口。

应用程序依赖 XML 文件来进行数据交换以及关系数据库来提供数据永久性都是应用程序业务逻辑（也就是其模式）的特征。因此，使用MVC 设计模板建议无需在应用程序的表示层（也就是其视图）反映这些实施细节。如果JSP 用于实施表示层，那么使用 xml 和 sql 库将违反MVC，因为它们的使用将意味着在表示层内暴露基本业务逻辑的elements。鉴于这一原因，xml 和 sql 库最适用于小型项目和原型工作。应用程序服务器对JSP 页面的动态编译也使得这些库中的定制标记可以用于作为调试工具。