

# 第4章 JSP技术模型

---

本章主要内容：

4.1 JSP语法概述

4.2 JSP页面生命周期

4.3 理解page指令属性

4.4 JSP脚本元素

4.5 JSP隐含变量

4.6 作用域对象

4.7 JSP组件包含



# JSP页面元素

JSP页面元素		简要说明	标签语法
脚本元素	声明	声明变量与定义方法。	<b>&lt;%!Java 声明 %&gt;</b>
	小脚本	执行业务逻辑的Java代码。	<b>&lt;%Java代码 %&gt;</b>
	表达式	用于在JSP页面输出表达式的值。	<b>&lt;%=表达式 %&gt;</b>
指令		指定页面转换时向容器发出的指令。	<b>&lt;%@指令 %&gt;</b>
动作		指示容器在页面执行期间完成某种任务。	<b>&lt;jsp:动作名 /&gt;</b>
EL表达式		JSP 2.0引进的表达式语言。	<b><b>\${applicationScope.email}</b></b>
注释		用于文档注释。	<b>&lt;%-- 任何文本 -- %&gt;</b>
模板文本		HTML标签和文本	同HTML规则

- 程序4.1 counter.jsp

```
<% @ page contentType='text/html;charset =  
gb2312' %>
```

```
<html><body>
```

```
<%! int count = 0; %>
```

```
<% count++; %>
```

该页面已被访问 <%= count %> 次。

```
</body></html>
```

- 运行: <http://localhost/ch04/counter.jsp>

# 4.1 JSP语法概述

---

**4.1.1 JSP 脚本元素**

**4.1.2 JSP 指令**

**4.1.3 JSP 动作**

**4.1.4 JSP表达式语言**

**4.1.5 JSP注释**



# 4.1.1 JSP脚本元素

---

- JSP声明
- JSP小脚本
- JSP表达式



# JSP声明

- 声明用来在JSP页面中**声明变量和定义方法**。声明是以**<%!**开头，以**%>**结束的标签。
- 其中可以包含任意数量的合法的Java声明语句,所声明的变量在**整个JSP页面中有效**。
- 下面是JSP声明的一个例子：

```
<%! int count = 0; %>
```

- 下面的代码在一个标签中声明了一个变量和一个方法:

**<%!**

```
String color[] = {"red", "green", "blue"};
```

```
String getColor(int i){
```

```
    return color[i];
```

```
}
```

**%>**

- 声明语句也可以写在两个JSP声明标签中

```
<%! String color[] = {"red", "green", "blue"}; %>
```

**<%!**

```
String getColor(int i){
```

```
    return color[i];
```

```
}
```

**%>**

# JSP小脚本

---

- 小脚本（**scriptlets**）是嵌入在JSP页面中的Java代码段。
- 当客户端向服务器端提交了包含JSP脚本的JSP页面请求时，Web服务器将**执行**脚本并将结果**发送**到客户端浏览器中。
- 小脚本是以**<%**开头，以**%>**结束的标签。例如：**<% count++; %>**
- 小脚本在**每次**访问页面时**都被执行**，因此**count**变量在每次请求时都增1。





➤ 使用小脚本打印HTML模板文本。例如：

```
<%@ page contentType="text/html;charset = gb2312"%>
```

```
<%! int count = 0; %>
```

```
<%
```

```
    out.print("<html><body>");
```

```
    count++;
```

```
    out.print("该页面已被访问" + count + "次。");
```

```
    out.print("</body></html>");
```

```
%>
```

注意：小脚本中的代码必须是合法的Java语言代码，例如下面的代码是错误的。原因是？

```
<% out.print(count) %>
```

# JSP表达式

---

- 表达式以`<%=`开头，以`%>`结束的标签，表达式结果会以字符串的形式发送到客户端显示。

`<%= count %>`

- 表达式不能以分号结束，非法的：

`<%= count; %>`

- 在JSP表达式的百分号和等号之间不能有空格。

`<% = count; %>`

- [程序4.2 expression.jsp](#)
- 运行: <http://localhost/ch04/expression.jsp>

## 4.1.2 JSP 指令

---

- **指令**（directive）是向容器提供的关于 JSP 页面的总体信息。
- 指令是以 **<%@** 开头，以 **%>** 结束的标签。
- 指令有三种类型：**page** 指令、**include** 指令和 **taglib** 指令。三种指令的语法格式如下：

**<%@ page attribute-list %>**

**<%@ include attribute-list %>**

**<%@ taglib attribute-list %>**



# 1、page指令

➤ page指令通知容器关于JSP页面的总体特性。例：

```
<%@ page contentType="text/html;charset =  
gb2312" %>
```

# 2、include指令

➤ include指令告诉容器把另一个文件(HTML、JSP等)的内容包含到当前页面中。例：

```
<%@ include file="copyright.html" %>
```

### 3、taglib指令

- taglib指令用来指定在JSP页面中使用标准标签或自定义标签的**前缀**与标签库的**URI**。例：

```
<%@ taglib prefix = "test"
```

```
uri = "/WEB-INF/mytaglib.tld" %>
```

#### ■ 关于指令的使用需注意下面几个问题：

- 标签名、属性名及属性值都是**大小写敏感**的；
- 属性值必须使用一对**单引号或双引号**括起来；
- 在等号（=）与值之间**不能有空格**。

## 4.1.3 JSP动作

---

➤ **动作（actions）** 是页面发给容器的**命令**，它指示容器在**页面执行期间完成某种任务**。

➤ 动作的一般语法为：

**<prefix:actionName attribute-list />**

- 在JSP页面中可以使用3种动作：
- **JSP标准**动作
  - **标准标签库（JSTL）** 中的动作
  - 用户 **自定义**动作。

➤ 例： **<jsp:include page="copyright.jsp" />**



## 4.1.4 表达式语言

---

- **表达式语言**（Expression Language, EL）是JSP 2.0新增加的特性，它的格式为：  
**`${expression}`**。
- 表达式语言是以**\$**开头，后面是一对**大括号**，括号里面是合法的**EL表达式**。
- 该结构可以出现在JSP页面的模板文本中，也可以出现在JSP标签的属性中。
- **例：****`${param.userName}`** 该EL显示请求参数userName的值。



## 4.1.5 JSP注释

---

**<html><body>**

**Welcome!**

**<% - - JSP 注释 - - %>**

**<% //Java 注释 %>**

**<!-- - HTML 注释 - ->**

**</body></html>**





## 4.2 JSP页面生命周期

---

**4.2.1 JSP页面也是Servlet**

**4.2.2 JSP生命周期阶段**

**4.2.3 JSP生命周期方法示例**

**4.2.4 理解页面转换过程**

**4.2.5 理解转换单元**



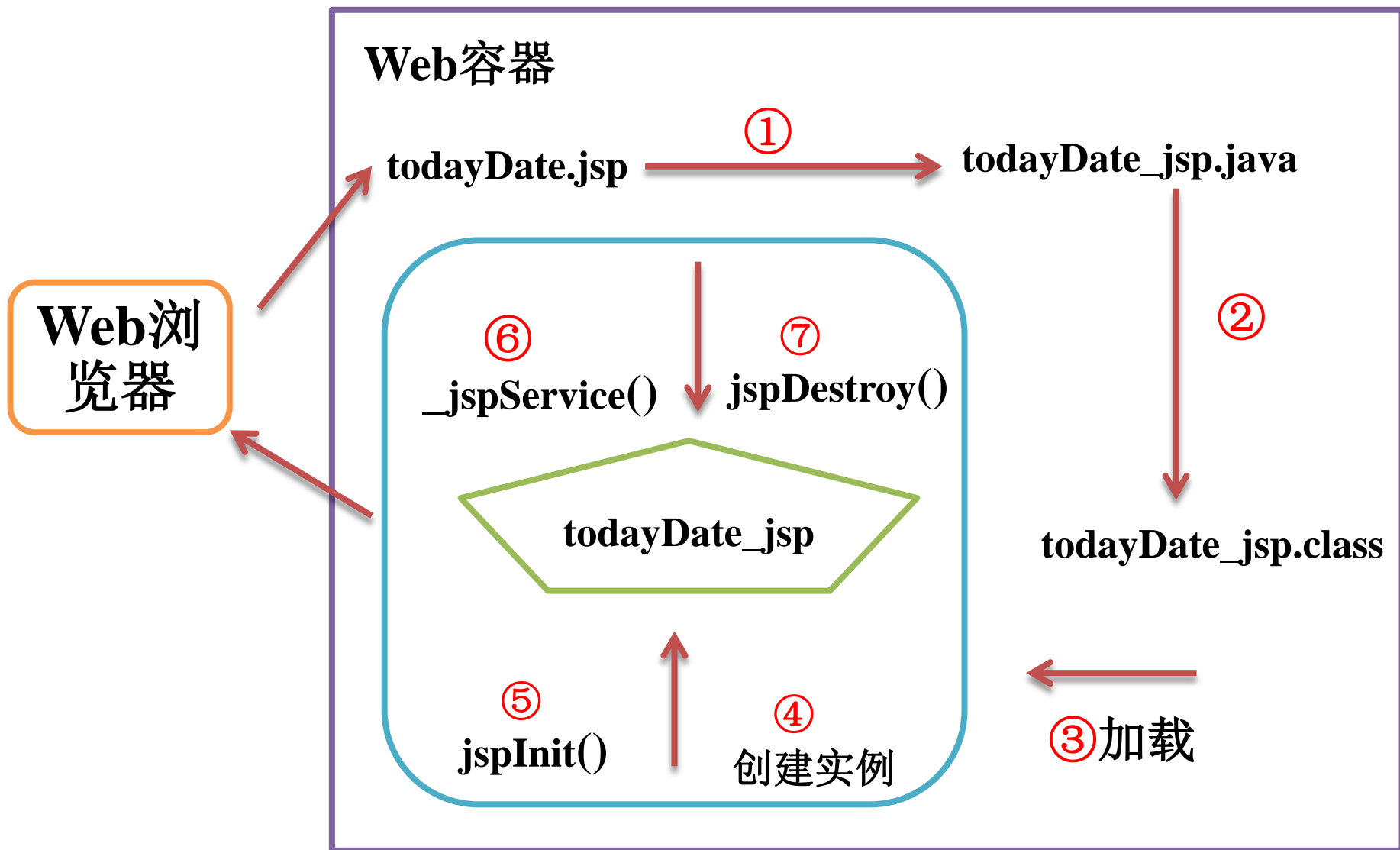
## 4.2.1 JSP页面也是Servlet

---

- JSP页面尽管从结构上看与HTML页面类似，但它实际上是作为**Servlet**运行的。
- 当JSP页面第一次被访问时，Web容器**解析**JSP文件并将其**转换**成相应的**Java**文件，该文件声明了一个**Servlet**类，我们将该类称为**页面实现类**。
- 接下来，Web容器编译该类并将其**装入**内存，然后与其他Servlet一样**执行**并将其**输出**结果**发送**到客户端。



## 4.2.2 JSP生命周期阶段



阶段名称	说明
<u>① 页面转换</u>	对页面解析并创建一个包含对应Servlet的Java源文件
<u>② 页面编译</u>	对Java源文件编译
<u>③ 加载类</u>	将编译后的类加载到容器中
<u>④ 创建实例</u>	创建一个Servlet实例
<u>⑤ 调用jspInit()</u>	调用其他方法之前调用该方法初始化
<u>⑥ 调用_jspService()</u>	对每个请求调用一次该方法
<u>⑦ 调用jspDestroy()</u>	当Servlet容器决定停止Servlet服务时调用该方法

### 程序4.3 todayDate.jsp


运行: <http://localhost/ch04/todayDate.jsp>

# 1.转换阶段

---

- 在这个阶段，**容器需要检验JSP页面所使用的标签的语法**，如果发现错误将不能转换。例如，下面的指令就是非法的。

```
<%@ Page import="java.util.*" %>
```

- 除了检查语法外，容器还将执行其他有效性检查，其中一些涉及到验证：
    - 指令中属性/值对与标准动作的合法性；
    - 同一个JavaBean名称在一个转换单元中没有被多次使用；
    - 如果使用了自定义标签库，标签库是否合法、标签的用法是合法；
- 

- 一旦验证完成，容器将创建一个**public**的包含**Servlet类的Java源文件**。该文件位于  
`<CATALINA_HOME>\work\Catalina\localhost\ch04\org\apache\jsp\`目录中，它包含对应于JSP文件的Servlet。
- <http://tomcat.apache.org/tomcat-8.0-doc/index.html>

- `org.apache.jasper.runtime.HttpJspBase`类，该类提供了Servlet接口的所有方法的默认实现和JspPage接口的两个方法`jspInit()`和`jspDestroy()`的默认实现。在转换阶段，容器把`_jspService()`添加到JSP页面的实现类中，这样使该类成为三个接口的一个具体子类。

- JspPage 接口只声明了两个方法：**jspInit()** 和 **jspDestroy()**。
- HttpJspPage 接口中声明了一个方法：**\_jspService()**。
- JSP 页面产生的 Servlet 类实现了这三个接口中所有的方法，故也被称为页面实现类。

```
public void jspInit();  
public void _jspService(HttpServletRequest  
    request, HttpServletResponse response)  
    throws ServletException, IOException;  
public void jspDestroy();
```



## 2.编译阶段

---

- 容器将调用**Java编译器javac**编译该文件。
- 在编译阶段，编译器将**检验在声明中、小脚本中以及表达式中所写的全部Java代码**。
- 例如，下面的声明标签尽管能够通过转换阶段，但由于声明语句没以分号结束，所以不是合法的Java声明语句，因此在编译阶段会被查出。

**<%! int count = 0 %>**





- 当JSP页面被首次访问时，服务器响应要比以后的访问慢一些。
- 这是因为在JSP页面向客户提供服务之前必须要转换成Servlet类的实例。
- 对每个请求，容器要检查JSP页面源文件的时间戳以及相应的Servlet类文件以确定页面是否是新的或是否已经转换成类文件。
- 因此，如果修改了JSP页面，将JSP页面转换成Servlet的整个过程要重新执行一遍。

- 为了加快JSP页面的执行，可以使用**预编译**请求参数**jsp\_precompile**强行编译JSP页面而不执行它。
- 例如，如果要在不执行的情况下编译todayDate.jsp页面，可以通过下面方式访问页面：  
[http://localhost/ch04/todayDate.jsp?jsp\\_precompile=true](http://localhost/ch04/todayDate.jsp?jsp_precompile=true)
- 这里值得指出的是所有以jsp为前缀的请求参数名都是保留的。



## 3. 加载类



- Web容器将页面实现类编译成类文件，然后**加载**到内存中。

## 4. 实例化

- Web容器调用页面实现类的构造方法创建一个Servlet类的**实例**。

## 5. 调用jspInit()



- Web容器调用**jspInit()**初始化Servlet实例。该方法是在任何其他方法调用之前调用的，并在页面生命期内只调用一次。
- 通常在该方法中完成初始化或只需一次的设置工作，如获得资源及初始化JSP页面中使用**<%! ... %>**声明的实例变量。



## 6.调用\_jspService()

---



- 对该页面的每次请求容器都调用一次 **\_jspService()**，并给它传递请求和响应对象。
- JSP页面中所有的HTML元素，JSP小脚本以及JSP表达式在转换阶段都成为该方法的一部分。



## 7.调用jspDestroy()



- 当容器决定停止该实例提供服务时，它将调用**jspDestroy()**，这是在Servlet实例上调用的最后一个方法，它主要用来清理**jspInit()**获得的资源。
- 一般不需要实现**jspInit()**和**jspDestroy()**，因为它们已经由基类实现了。但可以根据需要使用JSP的声明标签**<%! ... %>**覆盖这两个方法。然而，**不能覆盖\_jspService()**，因为该方法由Web容器自动产生。

## 4.2.3 JSP生命周期方法示例

---


- 下面的lifeCycle.jsp页面覆盖了jspInit()和jspDestroy(), 当该页面第一次被访问时将在控制台中看到“jspInit...”, 当应用程序关闭时, 将会看到“jspDestroy...”。
- [程序4.4 lifeCycle.jsp](#)  
运行: <http://localhost/ch04/lifeCycle.jsp>



## 4.2.4 理解页面转换过程

---

❑ 容器根据下面规则将JSP页面中的元素**转换**成Servlet代码：

- 所有JSP**声明**都转换成页面实现类的**成员**，它们被原样拷贝。例如，声明的变量转换成实例变量，声明的方法转换成实例方法。
  - 所有JSP**小脚本**都转换成页面实现类的 **\_jspService()**的一部分，它们也被原样拷贝。小脚本中声明的**变量**转换成\_jspService()的**局部变量**，小脚本中的**语句**转换成\_jspService()中的**语句**。
- 



- 所有的JSP表达式都转换成为\_jspService()的一部分，表达式的值使用out.print()语句输出。
- 有些指令在转换阶段产生Java代码，例如，page指令的import属性转换成页面实现类的import语句。
- 所有的JSP动作都通过调用针对厂商的类来替换。
- 所有表达式语言EL通过计算后使用out.write()语句输出。
- 所有模板文本都成为\_jspService()的一部分，模板内容使用out.write()语句输出。
- 所有的JSP注释都被忽略。

## 4.2.5 理解转换单元

---

- 被转换成单个Servlet类的**页面集合**称为**转换单元**。
- 有些JSP标签**影响整个转换单元**而不只是它们所声明的页面
  - **page指令**影响整个转换单元；
  - 在一个转换单元中一个**变量**不能多次声明。
  - 在一个转换单元中不能使用`<jsp:useBean>`动作对一个**bean**声明两次。



## 4.3 理解page指令属性

- page指令用于告诉容器关于**JSP页面的总体属性**，该指令适用于**整个转换单元**而不仅仅是它所声明的页面。
- page指令的13种可能的属性

属性名	说明	默认值
<u>import</u>	导入在JSP页面中使用的Java类和接口，其间用逗号分隔	java.lang.*; javax.servlet.*; javax.servlet.jsp.*; javax.servlet.http.*;
<u>contentType</u>	指定输出的内容类型和字符集	text/html; charset=ISO-8859-1

属性名	说明	默认值
<u>pageEncoding</u>	指定JSP文件的字符编码	ISO-8859-1
<u>session</u>	用布尔值指定JSP页面是否参加HTTP会话	true
errorPage	用相对URL指定另一个JSP页面用来处理当前页面的错误	null
isErrorPage	用一个布尔值指定当前JSP页面是否用来处理错误	false
language	指定容器支持的脚本语言	java
extends	任何合法的实现了 javax.servlet.jsp.jspPage 接口的java类	与实现有关

属性名	说明	默认值
<b>buffer</b>	指定输出缓冲区的大小	与实现有关
<b>autoFlush</b>	指定是否当缓冲区满时自动刷新	<b>true</b>
<b>info</b>	关于JSP页面的任何文本信息	与实现有关
<b>isThreadSafe</b>	指定页面是否同时为多个请求服务	<b>true</b>
<b>isELIgnored</b>	指定是否在此转换单元中对EL表达式求值	若web.xml采用Servlet 2.4格式, 默认值为true

## 4.3.1 import属性

---

- 在转换阶段，容器对使用**import属性**声明的每个包都转换成生成的Servlet类的一个**import语句**。
- 可以在一个import属性中**导入多个包**，包名用**逗号**分开即可，例如：

```
<%@ page import="java.util.*, java.io.*,  
    java.text.*, com.myserver.*,  
    com.myserver.util.MyClass " %>
```



➤ 为了可读性也可以使用多个标签，例如：

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

```
<%@ page import="java.io.*" %>
```

```
<%@ page import="java.text.*" %>
```

```
<%@ page import="com.myserver.*,  
com.myserver.util.MyClass" %>
```

➤ 另外，容器默认导入

java.lang.\*

javax.servlet.\*

javax.servlet.http.\*

javax.servlet.jsp.\*

## 4.3.2 contentType和pageEncoding属性

---

➤ `<%@ page contentType = 'text/html; charset =iso-8859-1' %>`

➤ 与Servlet中的下面一行等价：

`response.setContentType('text/html;charset = iso-8859-1');`

➤ 如果页面需要显示中文，字符集应该指定为 **UTF-8**，如下所示。

`<%@ page contentType='text/html;charset =UTF-8' %>`

`<%@ page pageEncoding="UTF-8" %>`



## 4.3.3 session属性

---

- session属性指示JSP页面是否参加HTTP会话，
- 其默认值为true，在这种情况下容器将声明一个隐含变量session。
- 如果不希望页面参加会话，可以明确地加入下面一行：

```
<% @ page session = "false" %>
```



## 4.3.4 `errorPage`与`isErrorPage`属性

---

□ JSP规范定义了一种更好的方法，它可以使**错误处理代码**与**主页面代码****分离**，从而提高异常处理机制的可重用性。

□ 例如：[程序4.5 helloUser.jsp](#)

运行：<http://localhost/ch04/helloUser.jsp>

**errorHandler.jsp**被指定为错误处理页面

□ [程序4.6 errorHandler.jsp](#)



## 4.3.5 在DD中配置错误页面

➤在DD中配置错误页面需要使用**`<error-page>`**标签，它的子元素有**`<exception-type>`**、**`<error-code>`**和**`<location>`**

```
<error-page>
```

```
    <exception-type>java.lang.ArithmeticException
```

```
    </exception-type>
```

```
    <location>/error/arithmeticError.jsp
```

```
    </location>
```

```
</error-page>
```

➤ 以下代码声明一个更通用的错误处理页面：

```
<error-page>
```

```
  <exception-type>java.lang.Throwable</exception-type>
```

```
  <location>/error/errorPage.jsp</location>
```

```
</error-page>
```

➤ 以下代码为HTTP的状态码404配置了一个错误处理页面，如下所示：

```
<error-page>
```

```
  <error-code>404</error-code>
```

```
  <location>/error/notFoundError.jsp</location>
```

```
</error-page>
```

## 4.3.6 language与extends属性

---

- **language**属性指定在页面的声明、小脚本及表达式中使用的语言，默认值是java。

```
<%@ page language="java" %>
```

- **extends**属性指定页面产生的Servlet的基类，默认的基类是厂商提供的。

```
<%@ page extends =  
    "mypackage.MySpecialBaseServlet" %>
```



## 4.3.7 buffer与autoFlush属性

---

- **buffer属性**指定输出缓冲区的大小。

`<% @ page buffer="32kb" %>`

缓冲区的值是以K字节为单位且kb是必须的

- **autoFlush属性**指定是否在缓冲区满时自动将缓冲区中的数据发送给客户，该属性的默认值为true。

`<% @ page autoFlush = "false" %>`



## 4.3.8 info属性

---

- **info** 属性指定一个字符串值，它可由Servlet调用`getServletInfo()`返回。

`<%@ page info="This is a sample Page. " %>`

- 在页面中使用`<%=getServletInfo()%>`脚本检索该值。
- 该属性的默认值依赖于实现。



## 4.4 JSP脚本元素

---

**4.4.1 变量的声明及顺序**

**4.4.2 使用条件和循环语句**

**4.4.3 请求时属性表达式的使用**





## 4.4.1 变量的声明及顺序

---

### 1. 声明的顺序:

- 在JSP页面的声明中定义的变量和方法都变成产生的Servlet类的成员，因此它们在页面中出现的**顺序无关紧要**。

程序4.7 area.jsp

运行:<http://localhost/ch04/area.jsp>



## 2. 小脚本的顺序

➤ 由于小脚本将被转换成Servlet的\_jspService()方法的一部分，因此小脚本中声明的变量成为该方法的局部变量，故它们**出现的顺序是重要的**。

```
<html> <body>
```

```
<% String s2 = "world"; %>
```

```
<% String s = s1+s2; %>
```

```
<%! String s1 = "hello"; %>
```

```
<% out.print(s); %>
```

```
</body> </html>
```

### 3.变量的初始化

➤在Java语言中，实例变量被自动初始化为默认值，而局部变量使用之前必须明确赋值。

```
<html>
```

```
<body>
```

```
<%! int i; %>
```

```
<% int j = 0; %>
```

```
The value of i is <%= ++i %> <br>
```

```
The value of j is <%= ++j %> <br>
```

```
</body>
```

```
</html>
```

在JSP声明中声明的变量被初始化为默认值，而在JSP小脚本中声明的变量使用之前必须明确初始。

➤ **问题：**如果多次访问上面页面，i的值输出结果怎样？j的值输出结果怎样？原因是什么？

答：i的值将**每次增1**，**输出一个新值**，而j的值总是输出**1**。

原因如下：

- ① **实例变量**是在容器实例化Servlet时被创建的并只被**初始化一次**，因此在JSP声明中声明的变量在多个请求中一直**保持**它们的值。
- ② 而**局部变量**对每个请求都创建和销毁一次，因此在小脚本中声明的变量在多个请求中**不保持其值**，而是在JSP容器每次调用\_jspService()时被**重新初始化**

## 4.4.2 使用条件和循环语句

---

### 1、脚本代码使用条件语句

<%

```
String username =  
request.getParameter('username');  
String password =  
    request.getParameter('password');  
boolean isLoggedIn = false;  
if(username.equals('admin')&&  
password.equals('admin'))  
isLoggedIn = true;
```



**else**

**isLoggedIn = false;**

**if(isLoggedIn)**

**out.print("<h3>欢迎你,"+username+"访问  
该页面！ </h3>");**

**else{**

**out.println("你还没有登录！ <br>");**

**out.println("<a href='login.jsp'>登录</a>");**

**}**

**%>**

## 2、条件语句跨越多个小脚本

<%

if (isLoggedIn) {

%>

<h3>欢迎你, <%=username%> 访问该页面! </h3>

这里可包含其他HTML代码!

<%

} else {

%>

你还没有登录! <br>

<a href="login.jsp">登录</a>

这里可包含其他HTML代码!

<%

}

%>

➤ 注意忽略大括号将会产生编译错误。

```
<% if (isLoggedIn) %>
```

```
<h3>欢迎你, <%=username%> 访问该页面  
! </h3>
```

将被转换成:

```
if (isLoggedIn)
```

```
out.write("欢迎你, ");
```

```
out.print(username);
```

```
out.write("访问该页面! </h3>");
```



### 3、循环语句也跨越多段小脚本。

下面例子使用循环计算并输出100以内的素数。

#### 程序4.8 prime.jsp

运行: <http://localhost/ch04/prime.jsp>

- 上述代码使用两段小脚本把HTML代码包含在循环中，然后使用JSP表达式输出素数n，之后输出两个空格。

### 4.4.3 请求时属性表达式的使用

- JSP表达式并不总是写到页面的输出流中，它们也可以用来向动作传递属性值：

```
<% String pageURL = "copyright.html"; %>
```

```
<jsp:include page="<%= pageURL %>" />
```

- 在该例中，JSP表达式<%= pageURL %>的值并不发送到输出流，而是在请求时计算出该值，然后将它赋给<jsp:include>动作的page属性。

- 这种向动作传递一个属性值使用的表达式称为请求时属性表达式。

注意：提供请求时属性值的机制**不能用在指令中**。

➤ 因为指令具有转换时的语义，即容器仅在页面转换期间使用指令。

➤ 例如：

`<%! String pageURL = "copyright.html"; %>`

**非 法 的 ×**

`<% @ include file="'<%= pageURL %>'" %>`

## 4.5 JSP隐含变量

```
<html><body>
```

```
<%
```

```
    out.print("<h1>Hello World! </h1>");
```

```
%>
```

```
</body></html>
```

- 在JSP页面的**转换阶段**，容器在\_jspService()方法中**声明并初始化**一些变量，我们可以在JSP页面中**直接使用**这些变量，例如：
- out一般被称为**隐含变量**，也被叫做**隐含对象**

➤ 例： [程序4.3 todayDate.jsp](#)

转换后代码： [程序4.9 todayDate\\_jsp.java](#)

可以看到在\_jspService中声明了8个变量。

➤ 例： [程序4.6 errorHandler.jsp](#)

错误处理页面，即页面中包含下面的page指令。

```
<%@ page isErrorPage="true" %>
```

转换后代码： [errorHandler\\_jsp.java](#)

则页面实现类中自动声明一个exception隐含变量，

```
Throwable exception =  
(Throwable) request.getAttribute  
("javax.servlet.jsp.jspException");
```

隐含变量	类或接口	说明
<b>application</b>	<b>javax.servlet.Servlet Context接口</b>	引用Web应用程序上下文
<b>session</b>	<b>javax.servlet.http.Http Session接口</b>	引用用户会话
<b>request</b>	<b>javax.servlet.http.Http ServletRequest接口</b>	引用页面的当前请求对象
<b>response</b>	<b>javax.servlet.http.Http ServletResponse接口</b>	用来向客户发送一个响应

隐含变量	类或接口	说明
<b>out</b>	<b>javax.servlet.jsp.JspWriter</b> 类	引用页面输出流
<b>page</b>	<b>java.lang.Object</b> 类	引用页面的Servlet实例
<b>pageContext</b>	<b>javax.servlet.jsp.PageContext</b> 类	引用页面上下文
<b>config</b>	<b>javax.servlet.ServletConfig</b> 接口	引用Servlet的配置对象
<b>exception</b>	<b>java.lang.Throwable</b> 类	用来处理错误

## 4.5.1 request与response变量

- **request**和**response**分别是**HttpServletRequest**和**HttpServletResponse**类型的隐含变量
- 它们作为**参数**传递给**\_jspService()**方法的

```
<%
```

```
    String remoteAddr = request.getRemoteAddr();  
    response.setContentType("text/html;charset=ISO-  
    8859-1");
```

```
%>
```

```
<html><body>
```

```
    Hi! Your IP address is <%=remoteAddr%>
```

```
</body></html>
```



## 4.5.2 out变量

---

➤ 是`javax.servlet.jsp.JspWriter`类型的隐含变量

➤ 例如:

```
<% out.print("Hello World!"); %>
```

```
<%= "Hello User!" %>
```

➤ 产生的Servlet代码都使用out变量打印出值

```
public void _jspService(...){
```

```
//其他代码
```

```
out.print("Hello World!");
```

```
out.print("Hello User!");
```

```
}
```



## 4.5.3 application变量

➤ application是**javax.servlet.ServletContext**类型的隐含变量。下面两段小脚本是等价的：

```
<%  
String path = application.getRealPath('/WEB-INF/counter.db');  
application.log('Using: '+path);  
%>
```

```
<%  
String path =  
    getServletContext().getRealPath('/WEB-INF/counter.db');  
getServletContext().log('Using: '+path);  
%>
```

## 4.5.4 session变量

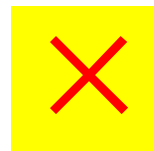
- session是**javax.servlet.http.HttpSession**类型的隐含变量。
- 如果page指令的session属性设置为**true**，那么在页面转换的类中就会声明一个**session**隐含变量。
- 如果明确将session属性设置为**false**，容器将**不会声明该变量**，对该变量的使用将产生错误,例如:

```
<%@ page session = "false" %>
```

```
<html><body>
```

```
Session ID = <%=session.getId()%>
```

```
</body></html>
```



## 4.5.5 pageContext变量

---

- pageContext是`javax.servlet.jsp.PageContext`类型的隐含变量。
- 它有三个作用：
  - (1) 存储隐含对象的引用。  
session、application、config与out这些隐含变量是调用pageContext对象的相应方法得到的。
  - (2) 提供了在不同作用域内返回或设置属性的方便的方法。



(3) 提供了**forward()**方法和**include()**方法实现将请求**转发**到另一个资源和将一个资源的输出**包含**到当前页面中的功能。

```
public void include(String relativeURL)
```

```
public void forward(String relativeURL)
```

➤ 例如，从Servlet中将请求转发到另一个资源：

```
RequestDispatcher view =
```

```
request.getRequestDispatcher("other.jsp");
```

```
view.forward(request, response);
```

➤ 在JSP页面中，通过使用pageContext变量可以完成上述功能：

```
pageContext.forward("other.jsp");
```

## 4.5.6 page变量

---

- page变量是`java.lang.Object`类型的对象，它指的是生成的Servlet实例，声明如下：

`Object page = this;`

`//this` 指当前Servlet实例

- 该变量很少被使用。



## 4.5.7 config变量

---

➤ config是**javax.servlet.ServletConfig**类型的隐含变量。

例：DD文件为JSP页面传递一组初始化参数。

**<servlet>**

**<servlet-name>InitTestServlet</servlet-name>**

**<jsp-file> /jsp/initTest.jsp </jsp-file>**

**<init-param>**

**<param-name>email</param-name>**

**<param-value>smith@yahoo.com.cn</param-value>**

**</init-param>**

**</servlet>**



**<html> <body>**

**Servlet Name =**

**<%=config.getServletName()%><br>**

**Email Address =**

**<%=config.getInitParameter("email")%>**

**</body> </html>**

➤ 访问该页面，它将输出下面的结果：

**Servlet Name = InitTestServlet**

**Email Address= smith@yahoo.com.cn**



## 4.5.8 exception变量

---

- exception是**java.lang.Throwable**类型的隐含变量，
- 在页面的page指令中将**isErrorPage**的属性值**设置为true**


```
<% @ page isErrorPage='true' %>
```

```
<html><body>
```

页面发生了下面错误：

```
<%=exception.toString()%>
```

```
</body></html>
```



## 4.6 作用域对象

作用域名	对应的对象	存在性和可访问性
应用作用域	<b>application</b>	在整个Web应用程序有效
会话作用域	<b>session</b>	在一个用户会话范围内有效
请求作用域	<b>request</b>	在用户请求和转发的请求内有效
页面作用域	<b>pageContext</b>	只在当前页面(转换单元内)有效

## 4.6.1 应用作用域

- 属于应用作用域的对象可被**Web应用程序的所有组件共享**,并在应用程序生命期内都可以访问。

```
<%
```

```
Float one = new Float(42.5) ;
```

```
application.setAttribute("foo", one);
```

```
%>
```

```
<%=application.getAttribute("foo") %>
```

上述代码相当于在Servlet中的下列代码:

```
getServletContext().setAttribute("foo", one);
```

```
out.println(getServletContext().getAttribute("foo"));
```

## 4.6.2 会话作用域

- 属于会话作用域的对象可以被属于一个用户会话的所有请求共享并只能在会话有效时才可被访问。

```
HttpSession session =  
    request.getSession(true);  
ShoppingCart cart =  
(ShoppingCart)session.getAttribute("cart");  
if (cart == null) {  
    cart = new ShoppingCart();  
    // 将购物车存储到会话对象中  
    session.setAttribute("cart", cart);  
}
```

## 4.6.3 请求作用域

➤ 属于请求作用域的对象可以被**处理同一个请求的所有组件共享**并仅在该请求被服务期间可被访问。

```
User user = new User();  
user.setName(request.getParameter("name"));  
user.setPassword(request.getParameter("password"))  
request.setAttribute("user", user);  
RequestDispatcher rd =  
request.getRequestDispatcher("/valid.jsp");  
rd.forward(request,response);
```

## 4.6.3 请求作用域

---

下面是valid.jsp文件:

```
<% User user = (User) request.getAttribute("user");  
    if (isValid(user)){  
        request.removeAttribute("user");  
        session.setAttribute("user",user);  
        pageContext.forward("account.jsp");  
    }else{  
        pageContext.forward("loginError.jsp");  
    }  
%>
```

## 4.6.4 页面作用域

---

- 属于页面作用域的对象只能在它们所定义的**转换单元**中被访问。它们不能存在于一个转换单元的单个请求处理之外。
- 在JSP页面中，该实例可以通过隐含对象**pageContext**访问。

```
<% Float one = new Float(42.5) ;%>
```

```
<% pageContext.setAttribute('foo',  
    one ) ;%>
```

```
<%= pageContext.getAttribute('foo' )%>
```

➤ **PageContext**类还定义了几个**常量**和其他属性处理**方法**，使用它们可以方便的处理**不同作用域的属性**。

➤ 定义的常量有：

**public static final int APPLICATION\_SCOPE**

**public static final int SESSION\_SCOPE**

**public static final int REQUEST\_SCOPE**

**public static final int PAGE\_SCOPE**



➤ 定义的方法有：

**public void setAttribute**

**(String name, Object object, int scope)**

**public Object getAttribute**

**(String name, int scope)**

**public void removeAttribute**

**(String name, int scope)**

**public Object findAttribute(String name)**

**public int getAttributesScope(String name)**

```
<% Float two = new Float(22.5) ;%>
```

```
<% pageContext.setAttribute('foo', two,  
    PageContext.SESSION_SCOPE ) ;%>
```

```
<%= pageContext.getAttribute('foo',  
    PageContext.SESSION_SCOPE ) ;%>
```

```
<%=pageContext.getAttribute('email',  
    PageContext.APPLICATION_SCOPE)%>
```

```
<%= pageContext.findAttribute('foo')%>
```

# 作用域对象举例

---

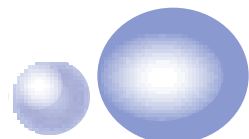
程序举例:[domaintest.jsp](#)

程序演示:

<http://localhost/ch04/domaintest.jsp>



## 4.7 JSP组件包含

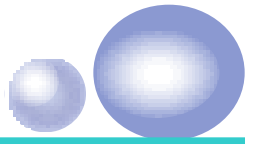


- 代码的**可重用性**是软件开发的一个重要原则。使用可重用的组件可提高应用程序的生产率和可维护性。
- JSP规范定义了一些允许重用Web组件的机制，其中包括在JSP页面中**包含**另一个Web组件的**内容或输出**。
- 这可通过两种方式之一实现：**静态包含或动态包含**。



## 4.7 JSP组件包含

---



**4.7.1 静态包含: include指令**

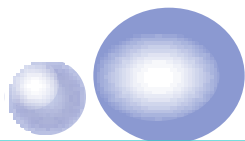
**4.7.2 动态包含: include动作**

**4.7.3 使用<jsp:forward>动作**

**4.7.4 实例: 使用包含设计页面布局**



## 4.7.1 静态包含：include指令



- 静态包含是在JSP页面转换阶段将另一个文件的内容包含到当前JSP页面中产生一个servlet。
- 静态包含使用include指令，语法：  
`<%@ include file="relativeURL" %>`
- 文件可以是任何基于文本的文件，如HTML、JSP、XML文件，甚至是简单的.txt文件。
- 文件使用相对路径指定，相对路径或者以斜杠 (/) 或者不以斜杠开头。
- 例如： `<%@ include file="other.jsp" %>`



## ❖ include指令的工作方式

main.jsp文件

```
<html><body>
<b>Welcome</b>
<%@ include file="other.jsp"%>
<b>Good Bye</b>
</body></html>
```

other.jsp文件

```
<pre>
Once upon a time
</pre>
```

↓ main.jsp转换时

```
//在 _jspService()中
out.write("<html><body>
>
<b>Welcome</b>");
out.write("<pre>
Once upon a time
</pre>");
out.write("
<b>Good Bye</b>
</body></html>");
```

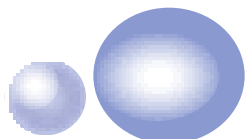
请求时

```
<html><body>
<b> Welcome </b>
<pre>
Once upon a time
</pre>
<b>Goog Bye</b>
</body></html>
```

main.jsp产生的Servlet

HTML的输出

# 1、从被包含的页面中访问变量



❖ 每个页面都可以**访问**在另一个页面中定义的**变量**。它们也**共享**所有的**隐含对象**。


❖ 程序4.11 hello.jsp

❖ 程序4.12 response.jsp

❖ 运行: <http://localhost/ch04/hello.jsp>

A screenshot of a web browser window. The title bar says "Hello". The page content includes a small penguin character (Duke) with its hand raised, followed by the text "My name is Duke. What is yours?". Below this is a text input field, a "提交" (Submit) button, and a "重置" (Reset) button. At the bottom, the text "Hello, qq!" is displayed in blue.

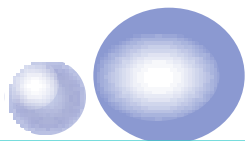
Hello

 My name is Duke. What is yours?

Hello, qq!



## 2、静态包含的限制



(1)在转换阶段不进行任何处理，这意味着file属性值**不能是表达式**。

```
<% String myURL ="copyright.html"; %>
```

```
<%@ include file="<%= myURL %>" %>
```

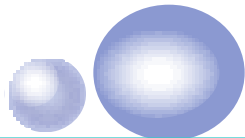


(2)**不能**通过file属性值向被包含的页面**传递参数**,因为请求参数是请求的一个属性，它在转换阶段没有任何意义。

```
<%@ include file="other.jsp?name=honey" %>
```



## 2、静态包含的限制

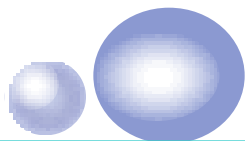


(3)被包含的页面**可能不能单独编译**。

- 如果一个页面使用了另一个页面声明的变量，该页面就不能单独编译。一般来说，最好避免这种相关性，
- 可以使用隐含变量**pageContext**共享对象，通过使用pageContext的**setAttribute()**和**getAttribute()**方法实现



## 4.7.2动态包含： include动作



- ❖ 其语法如下所示：

```
<jsp:include page="relativeURL" flush="true" />
```

- ❖ **page**属性是必须的，其值必须是相对URL，并指向任何静态或动态web组件，包括servlet。也可以是请求时表达式，例如：

```
<% String pageURL = "other.jsp"; %>
```

```
<jsp:include page="<%= pageURL %>" />
```

- ❖ **flush**属性是指在将控制转向被包含页面之前是否刷新当前页面的缓冲区，默认false。



## ❖ jsp:include工作方式

main.jsp文件

```
<html><body>  
  Hello,World!<br>  
  <jsp:include page="other.jsp" />  
</body></html>
```

转换阶段

```
//在 _jspService()中  
out.write("<html>");  
//控制转移到 other.jsp  
out.write("</html>");
```

main.jsp产生的Servlet

```
<html>  
  Hello,World!  
  Nothing is impossible  
</html>
```

输出的HTML

other.jsp文件

```
<pre>  
Nothing is impossible  
</pre>
```

转换阶段

```
//在 _jspService()中  
out.print("<pre>  
Nothing is impossible</pre>")
```

other.jsp产生的Servlet

控制转移

恢复处理

- 在功能上<jsp:include>动作的语义与RequestDispatcher接口的include()的语义相同。
- 下面三个结构是等价的。

### 结构1:

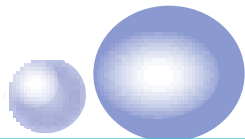
```
<%  
    RequestDispatcher view =  
        request.getRequestDispatcher("other.jsp");  
    view.include(request, response);  
%>
```

### 结构 2:

```
<%  
    pageContext.include("other.jsp");  
%>
```

结构 3: <jsp:include page="other.jsp" flush="true"/>

# 1、使用<jsp:param>传递参数



- 在<jsp:include>动作中可以使用<jsp:param />动作向被包含的页面传递参数。

```
<jsp:include page="somePage.jsp">
```

```
    <jsp:param name="name1" value="value1" />
```

```
    <jsp:param name="name2" value="value2" />
```

```
</jsp:include>
```

- 在被包含的页面中使用request隐含对象的getParameter()方法获得传递来的参数。

```
<%=request.getParameter("name1")>
```



- **value**的属性值可以使用**请求时属性**表达式。

```
<jsp:include page='somePage.jsp'>
```

```
    <jsp:param name='name1'
```

```
        value='<%= someExpr1 %>' />
```

```
    <jsp:param name='name2'
```

```
        value='<%= someExpr2 %>' />
```

```
</jsp:include>
```

- 通过<jsp:param>动作传递的名/值对存在于**request对象**中并只能由**被包含的组件**使用，
- 这些参数的**作用域是被包含的页面**，在被包含的组件完成处理后，容器将从**request对象**中清除这些参数。

## 2、与动态包含的组件共享对象

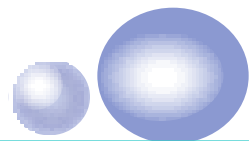
➤ 动态包含的页面是单独执行的，因此它们不能共享在包含页面中定义的变量和方法。然而，它们处理的请求对象是相同的，因此可以共享属于请求作用域的对象。

➤ 例如：[程序4.13 hello2.jsp](#)  
[程序4.14 response2.jsp](#)

➤ 运行：<http://localhost/ch04/hello2.jsp>



## 4.7.3 使用<jsp:forward>动作



- 使用<jsp:forward>动作把请求转发到其他组件，然后由转发到的组件把响应发送给客户
- 该动作的格式为：  

```
<jsp:forward page='relativeURL' />
```
- page属性的值为转发到的组件的相对URL，它可以使用请求时属性表达式。
- 使用<jsp:forward>动作，主页面也不能包含任何输出。



## ❖ jsp:forward工作方式

main.jsp文件

```
//do some processing  
<jsp:forward page="other.jsp" />
```

转换阶段

```
//在 _jspService()中  
pageContext.forward("other.jsp");
```

main.jsp产生的Servlet

处理并转发请求

other.jsp文件

```
<html>  
<%= "Hello!" %>  
</html>
```

转换阶段

```
//在 _jspService()中  
out.write("<html>");  
out.print("Hello!");  
out.write("</html>");
```

other.jsp产生的Servlet

```
<html>  
    Hello,World!  
    Nothing is impossible  
</html>
```

输出的HTML

- 在功能上<jsp:forward>的语义与RequestDispatcher接口的forward()方法的语义相同。
- 下面三个结构是等价的。

### 结构1:

```
<%
```

```
RequestDispatcher view =
```

```
request.getRequestDispatcher("other.jsp");
```

```
view.forward (request, response);
```

```
%>
```

### 结构 2:

```
<%
```

```
pageContext.forward("other.jsp");
```

```
%>
```

结构 3: <jsp:forward page="other.jsp"/>

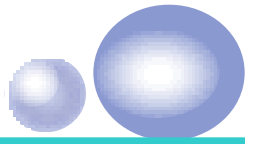
## 4.7.4 实例：使用包含设计页面布局



- 程序4.15 [index.jsp](#)
- 程序4.16 [header.jsp](#)
- 补充程序 [topmenu.jsp](#)
- 程序4.17 [leftmenu.jsp](#)
- 程序4.18 [content.jsp](#)
- 程序4.19 [footer.jsp](#)

运行: <http://localhost/ch04/index.jsp>

## 4.8 JavaBeans



❖ 本节主要内容:

4.8.1 JavaBeans规范

4.8.2 使用<jsp:useBean>动作

4.8.3 使用<jsp:setProperty>动作

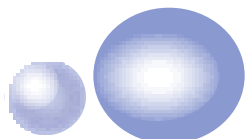
4.8.4 使用<jsp:getProperty>动作

4.8.5 JavaBeans应用示例

4.8.6 实现MVC模式的一般步骤



## 4.8.1 JavaBeans规范



- JavaBeans是用Java语言定义的类，这种类的设计需要遵循JavaBeans规范的有关约定。
- 任何遵循下面三个规范的Java类都可以作为JavaBeans使用。
  - (1) JavaBeans应该是public类，并且具有无参数的public构造方法。
  - (2) JavaBeans类的成员变量一般称为属性（property）。对每个属性访问权限一般定义为private，而不是public。

◆ 注意：属性名必须以小写字母开头



(3) 每个属性通常定义两个**public**方法，一个是 **访问方法 (getter)**，另一个是**修改方法 (setter)**，使用它们访问和修改JavaBeans的属性值。访问方法名应该定义为**getXxx()**，修改方法名应该定义为**setXxx()**。

❖ 例如

```
public String getColor()
```

```
    {return this.color; }
```

```
public void setColor(String color)
```

```
    { this.color=color;}
```



- 当属性是**boolean**类型时，访问器方法应该定义为**isXxx()**形式。
- 例如，假设JavaBean有一个boolean型属性**valid**，则访问器方法应该定义为**public boolean isValid()**。
- 除了访问方法和修改方法外，类中还可以定义其他的方法实现某种业务逻辑。也可以只为某个属性定义访问器方法，这样的属性就是只读属性。
- 例子：[程序4.20 Customer.java](#)

➤ **JavaBean命名惯例:UserBean、AccountBean。**

- 以区别一般的类，这样可以给协作的开发人员清晰的含义。

➤ **JavaBean类存放在**

- **/WEB-INF/classes** 目录中
- **/WEB-INF/lib** 目录中的**JAR**文件中。

➤ **在JSP中使用这些类，可以使用JSP标准动作**

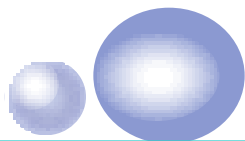
**<jsp:useBean>**创建JavaBeans类的一个实例。

➤ **JavaBeans类的一个实例一般称为一个bean。**

## ❖ JavaBean的优点

- 在JSP页面中使用JavaBeans的代码简洁。
- JavaBeans也有助于增强代码的可重用性。
- 它们是Java语言对象，充分利用该语言的面向对象特征。

## 4.8.2 使用<jsp:useBean>动作



- 在JSP页面中使用JavaBeans主要是通过三个JSP标准动作实现的，它们分别是：

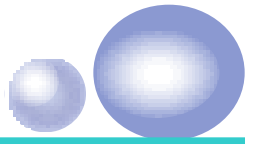
<jsp:useBean>动作

<jsp:setProperty>动作

<jsp:getProperty>动作



# 使用<jsp:useBean>动作

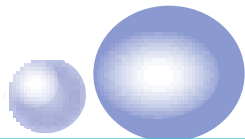


```
<jsp:useBean id="beanName"
    scope="page | request | session | application"
    { class="package.class" |
      type="package.class" |
      class="package.class" type="package.class"
    }
{ /> | >other elements </jsp:useBean> }
```



# 1、属性说明

---



- **id**属性用来唯一标识一个bean实例，该属性是**必须的**。
- 在JSP页面实现类中，id的值被作为Java语言的**变量**。
- 因此可以在JSP页面的表达式和小脚本中**使用**该变量。




# 1、属性说明




- **scope** 属性指定bean实例的**作用域**。与隐含对象类似，JavaBeans在JSP页面中的存在和可访问性是由4个JSP作用域决定的：

**Page   request   session   application**

- 该属性是**可选**的，默认值为**page**作用域。
  - 如果page指令的session属性设置为false，则bean不能在JSP页面中使用session作用域。
- 

# 1、属性说明



- **class**属性指定创建bean实例的**Java**类。
  - 如果容器在指定的作用域中**不能找到**一个现存的bean实例，它将使用**class**属性指定的类**创建**一个bean实例。
  - 如果该类**属于**某个包，则必须指定类的**全名**，如**com.demo.Customer**。
- 

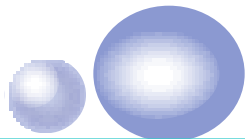


# 1、属性说明



- **type**属性指定由**id**属性声明的变量的类型。
  - 由于该变量是在请求时指向实际的bean实例，其类型必须与bean类的类型相同或者是其超类，或者是一个bean类实现的接口。
  - 同样，如果类或接口属于某个包，需要指定其全名，如**com.demo.Customer**。
- 

## 2、属性的使用



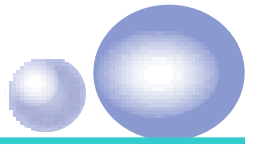
- 在<jsp:useBean>动作的属性中，**id**属性是**必须**的，**scope**属性是**可选**的。**class**和**type****至少**指定一个或**两个同时**指定。

### 1) 只指定class属性的情况

```
<jsp:useBean id='customer'  
class='com.demo.Customer' />
```



## 2、属性的使用

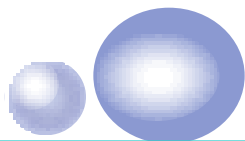


❖该动作与下面的一段代码等价：

```
Customer customer =  
(Customer)pageContext.getAttribute(  
"customer");  
    if (customer == null){  
        customer = new Customer();  
        pageContext.setAttribute(  
            "customer", customer);  
    }
```



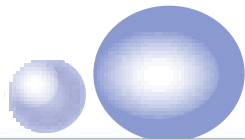
## 2、属性的使用



- ❖ 当JSP页面执行到该动作时，Web容器首先在page作用域中**查找**名为customer的bean**实例**。
- ❖ 如果**找到**就用customer引用**指向它**，如果**找不到**，将使用Customer类**创建**一个**对象**，并用customer引用**指向它**，同时将其作为**属性**添加到page作用域中。
- ❖ 因此该bean只能在它所定义的JSP页面中使用，且只能被该页面创建的请求使用。



## 2、属性的使用



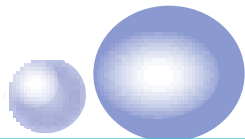
- 下面的动作使用了id、class和scope属性声明一个JavaBeans:

```
<jsp:useBean id="customer"  
class="com.demo.Customer"  
scope="session" />
```

- 该动作与下面的一段代码等价:



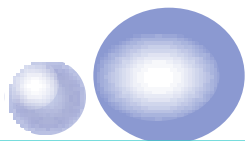
## 2、属性的使用



```
Customer customer =  
(Customer)session.getAttribute("customer");  
if (customer == null){  
    customer = new Customer();  
    session.setAttribute("customer", customer);  
}
```

- 当JSP页面执行到该动作时，容器在会话（**session**）作用域中查找或创建bean实例，并用**customer**引用指向它。

## 2、属性的使用



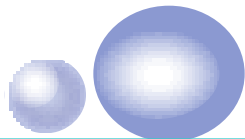
2) 只指定type属性的情况。

➤ 可以使用type属性代替class属性，例如：

```
<jsp:useBean id="customer"  
    type="com.demo.Customer"  
    scope="session" />
```

➤ 该动作在指定作用域中查找类型为Customer的实例，如果找到用customer指向它，如果找不到产生Instantiation异常。因此，使用type属性必须保证bean实例存在。

## 2、属性的使用



3) class属性和type属性的组合。

```
<jsp:useBean id="person"
```

```
    type="com.demo.Person"
```

```
    scope="session"
```

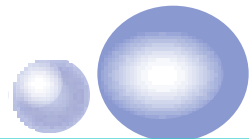
```
    class="com.demo.Customer" />
```

➤ 该动作与下面的一段代码等价：





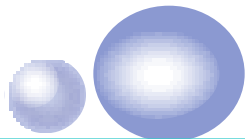
## 2、属性的使用



```
Person person =  
(Person)session.getAttribute("person");  
if (person == null){  
    person = new com.demo.Customer();  
    session.setAttribute("person", person);}
```

- 该动作在指定作用域中查找类型为**Person**的实例，如果找到用**person**指向它，如果找不到就用**class**属性指定的**Customer**类创建一个实例。

## 4.8.3 使用<jsp:setProperty>动作




➤ <jsp:setProperty>动作用来给bean的属性赋新值，它格式为：

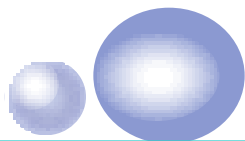
```
<jsp:setProperty name="beanName"  
    { property = "propertyName"  
      value="{string |  
<%=expression%>}" |  
      property = "propertyName"  
      [param="paramName"] |  
      property = "*" } />
```

# 1、属性说明



- ❖ **name属性**用来标识一个**bean实例**，该实例必须是前面使用<jsp:useBean>动作**声明的**，并且name属性值必须与<jsp:useBean>动作中指定的一个id属性值**相同**。该属性是**必须的**。
  - ❖ **property属性**指定要设置值的bean实例的**属性**，容器将根据指定的bean的属性调用适当的**setXxx()**，因此该属性是**必须的**。
- 

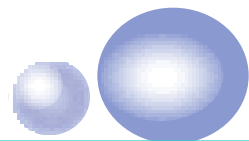
# 1、属性说明



- **value**属性为bean的属性指定**新值**，该属性值可以接受**请求时属性表达式**。
- **param**属性指定请求**参数名**，如果请求中包含指定的参数，那么使用该参数值来设置bean的属性值。
- **value**属性和**param**属性都是**可选**的并且**不能同时使用**。如果这两个属性都没有指定，容器将查找与属性同名的请求参数。



## 2、属性的使用



- 假设已按下面的代码声明了一个bean实例。

```
<jsp:useBean id="customer"  
class="com.demo.Customer" />
```

### 1) 使用value属性

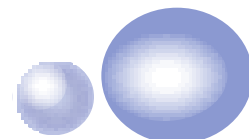
```
<jsp:setProperty name="customer"  
property="custName" value="Mary" />
```

```
<jsp:setProperty name="customer"  
property="email" value="mary@163.com" />
```

```
<jsp:setProperty name="customer"  
property="phone" value="8899123" />
```



## 2、属性的使用



- 假设已按下面的代码声明了一个bean实例。

```
<jsp:useBean id="customer"  
class="com.demo.Customer" />
```

### 1) 使用value属性

- 它们与下面的小脚本代码等价：

```
<%  
customer.setCustName('Mary');  
customer.setEmail('mary@163.com ');  
customer.setPhone('8899123');  
%>
```

## 2、属性的使用




### 2) 使用param属性

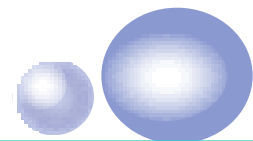
下面的例子中没有指定value属性的值，而是使用param属性指定请求参数名。

```
<jsp:setProperty name='customer '  
property='email' param='myEmail' />
```

```
<jsp:setProperty name='customer'  
property='phone' param='myPhone' />
```



## 2、属性的使用



### 2) 使用param属性

下面的例子中没有指定value属性的值，而是使用param属性指定请求参数名。

➤ 它们与下面的小脚本代码等价：

<%

```
customer.setEmail(request.getParameter(  
"myEmail"));
```

```
customer.setPhone(request.getParameter(  
"myPhone")); %>
```



## 2、属性的使用




### 3) 使用默认参数机制

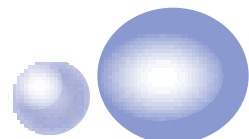
如果请求参数名与bean的属性名匹配，就不必指定param属性或value属性，如下所示。

```
<jsp:setProperty name='customer'  
property='email' />
```

```
<jsp:setProperty name='customer'  
property='phone' />
```



## 2、属性的使用



### 3) 使用默认参数机制

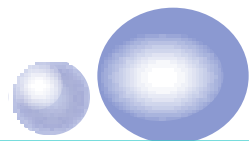
如果请求参数名与bean的属性名匹配，就不必指定param属性或value属性，如下所示。

➤ 它们与下面的小脚本代码等价：

```
<jsp:setProperty name='customer'  
property='email' param='email' />
```

```
<jsp:setProperty name='customer'  
property='phone' param='phone' />
```

## 2、属性的使用



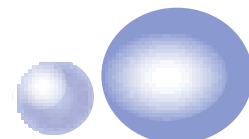
### 4) 在一个动作中设置所有属性

下面是在一个动作中设置bean的所有属性的一个捷径：

```
<jsp:setProperty name="customer"  
property="*" />
```

- 这里，为property的属性值指定"**\***"，它将使用请求参数的每个值为属性赋值，这样，就不用单独为bean的每个属性赋值。

## 2、属性的使用



### 4) 在一个动作中设置所有属性

下面是在一个动作中设置bean的所有属性的一个捷径：

➤ 它们与下面的小脚本代码等价：

**<%**

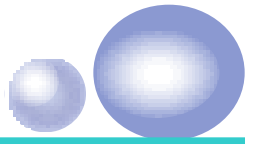
```
customer.setCustName(request.getParameter('custName'));
```

```
customer.setEmail(request.getParameter('email'));
```

```
customer.setPhone(request.getParameter('phone'));
```

**%>**

## 2、属性的使用



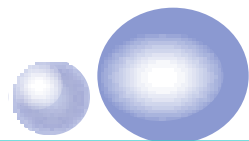
### 5) 有条件设置属性值

**<jsp:setProperty>**动作可以嵌套在**<jsp:UseBean>**动作中。

```
<jsp:useBean id="customer" class="com.demo.  
    Customer" scope="session" >  
    <jsp:setProperty name="customer"  
        property="custName" value="Mary" />  
    <jsp:setProperty name="customer"  
        property="custName" value="Mary" />  
    <jsp:setProperty name="customer"  
        property="custName" value="Mary" />  
</jsp:useBean>
```



## 2、属性的使用



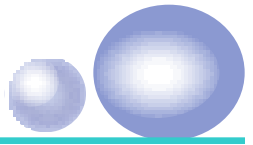
### 5) 有条件设置属性值

**<jsp:setProperty>**动作可以嵌套在**<jsp:UseBean>**动作中。

- 这样使用**<jsp:setProperty>**动作，只有在指定作用域找不到**customer**的**bean**对象时才会执行标签体，否则将不执行标签体。



## 4.8.4 使用<jsp:getProperty>动作



- ❖ <jsp:getProperty>动作用来检索和向输出流中打印bean的属性值，它的语法非常简单：

```
<jsp:getProperty name="beanName"  
                property="propertyName" />
```

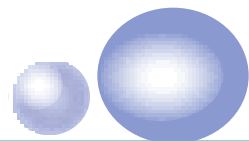
- ❖ 下面的动作指示JSP引擎打印名为customer的bean的email和phone属性值：

```
<jsp:getProperty name="customer"  
                property="email" />
```

```
<jsp:getProperty name="customer"  
                property="phone" />
```



## 4.8.4 使用<jsp:getProperty>动作



❖ <jsp:getProperty>动作用来检索和向输出流中打印bean的属性值，它的语法非常简单：

```
<jsp:getProperty name="beanName"  
                property="propertyName" />
```

➤ 它们与下面的小脚本代码等价：

```
<%
```

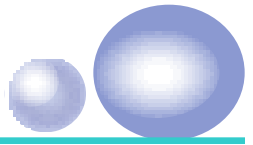
```
    out.print(customer.getEmail());  
    out.print(customer.getPhone());
```

```
%>
```





## 4.8.5 JavaBeans应用示例



### ❖ Model 2体系结构

- 下面示例首先在

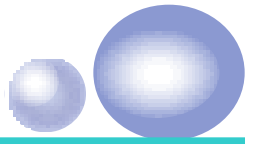
程序4.21 [inputCustomer.jsp](#)

程序4.22 [CustomerServlet.java](#)

程序4.23 [displayCustomer.jsp](#)

运行： <http://localhost/ch04/inputCustomer.jsp>

## 4.8.5 JavaBeans应用示例



### ❖ Model 1体系结构

❖ 首先在inputCustomer.jsp中输入客户信息，然后将请求转发到displayCustomer.jsp页面进行显示。

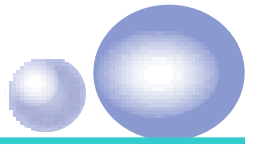
程序： [inputCustomer1.jsp](#)  
[displayCustomer1.jsp](#)

运行：

<http://localhost/ch04/inputCustomer1.jsp>



## 4.8.6 实现MVC模式的一般步骤



### 1. 定义JavaBeans表示数据

❖ JavaBeans对象一般只用来存放数据，JSP页面从JavaBeans对象获得数据并显示给用户。

```
public class Customer implements Serializable{  
    private String customName;  
    private String email;  
    private int age;  
    // 构造方法定义  
    // setter和getter方法定义 }
```



## 2. 使用Servlet处理请求

➤ 在MVC模式中，Servlet实现控制器功能，它从请求中**读取请求信息**（如表单数据）、**创建JavaBeans对象**、**执行业务逻辑**、**访问数据库**等，最后将请求**转发**到视图组件。

➤ Servlet并不创建任何输出，输出由JSP页面实现，因而，在Servlet中并不调用 `response.setContentType()`、`response.getWriter()`或`out.println()`等方法。

### 3.填写JavaBeans对象数据

- 控制器创建JavaBeans对象后需要填写该对象的值。可以通过请求参数值或访问数据库得到有关数据，然后填写到JavaBeans对象属性中。

## 4.结果的存储

- 创建了与请求相关的数据并将数据存储到JavaBeans对象中后，接下来应该将这些bean对象**存储**在JSP页面能够访问的位置。
- 在Servlet中主要可以在三个位置存储JSP页面所需的数据，它们是**HttpServletRequest**对象、**HttpSession**对象和**ServletContext**对象。
- 这些存储位置对应<jsp:useBean>动作**scope**属性的三个非默认值：**request**、**session**和**application**。

## 5. 转发请求到JSP页面

- 在使用请求作用域共享数据时，应该使用 **RequestDispatcher** 对象的 **forward()** 将请求转发到 JSP 页面。
- 获取 **RequestDispatcher** 对象可使用 **请求对象** 的 **getRequestDispatcher()** 或使用 **ServletContext** 对象的 **getRequestDispatcher()** 方法。
- 得到 **RequestDispatcher** 对象后，调用它的 **forward()** 将控制转发到指定的组件。
- 在使用 **会话作用域** 共享数据时，使用响应对象的 **sendRedirect()** 重定向可能更合适。

## 6. 从JavaBeans对象中提取数据

- ❖ 请求到达JSP页面之后，使用<jsp:useBean>和<jsp:getProperty>提取数据。
- ❖ 但应注意，不应在JSP页面中创建对象，创建JavaBeans对象是由Servlet完成的。

- ❖ 为了保证JSP页面不会创建对象，应该使用动作：

```
<jsp:useBean id="customer"  
type="com.demo.Customer" />
```

- ❖ 而不应该使用动作：

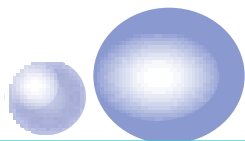
```
<jsp:useBean id="customer"  
class="com.demo.Customer" />
```



- 在JSP页面中也不应该修改对象。因此，只应该使用**<jsp:getProperty>**动作，而不应该使用**<jsp:setProperty>**动作。
- 另外，在**<jsp:useBean>**动作中使用的scope属性值应该与在Servlet中将JavaBeans对象存储的位置**相对应**。
- 如在Servlet中将一个JavaBeans对象存储在**请求作用域**中，在JSP页面中应该使用下面的**<jsp:useBean>**动作获得该JavaBeans对象：

```
<jsp:useBean id="customer"  
type="com.demo.Customer" scope="request">
```

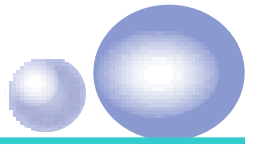
## 4.9 小 结



- ❖ JSP技术的主要目标是实现Web应用的**数据表示和业务逻辑分离**，JSP技术是建立在Servlet技术基础上的，所有的JSP页面最终**都会编译成Servlet代码**。在JSP页面中可以使用**指令、声明、小脚本、表达式、动作以及注释**等语法元素。
- ❖ 一个JSP页面在其生命周期中要经历7个阶段，即**页面转换、页面编译、加载类、创建实例、调用jspInit()、调用\_jspService()和调用jspDestroy()**等。



## 4.9 小 结



❖ JSP页面中可以使用的指令有三种：

**page指令、include指令和taglib指令。**

❖ 在JSP页面中还可以使用9个隐含变量：

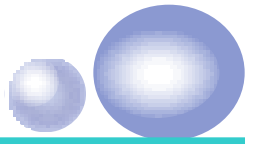
**application、session、request、**

**response、page、pageContext、out、**

**config和exception等。**



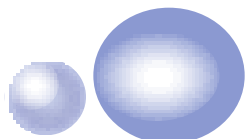
## 4.9 小 结



- ❖ **Java Web**开发中可以有多种方式重用**Web**组件。在**JSP**页面中包含组件的内容或输出实现**Web**组件的重用。有两种实现方式：使用**include**指令的静态包含和使用**<jsp:include>**动作的动态包含。
- ❖ **JavaBeans**是遵循一定规范的**Java**类，它在**JSP**页面中主要用来表示数据。**JSP**规范提供了标准动作：**<jsp:useBean>**



## 4.9 小 结



- ❖ MVC设计模式是Web应用开发中最常使用的设计模式，它将系统中的组件分为**模型、视图和控制器**，实现了业务逻辑和表示逻辑的分离，使用该模式开发的系统具有可维护性和代码重用性。

