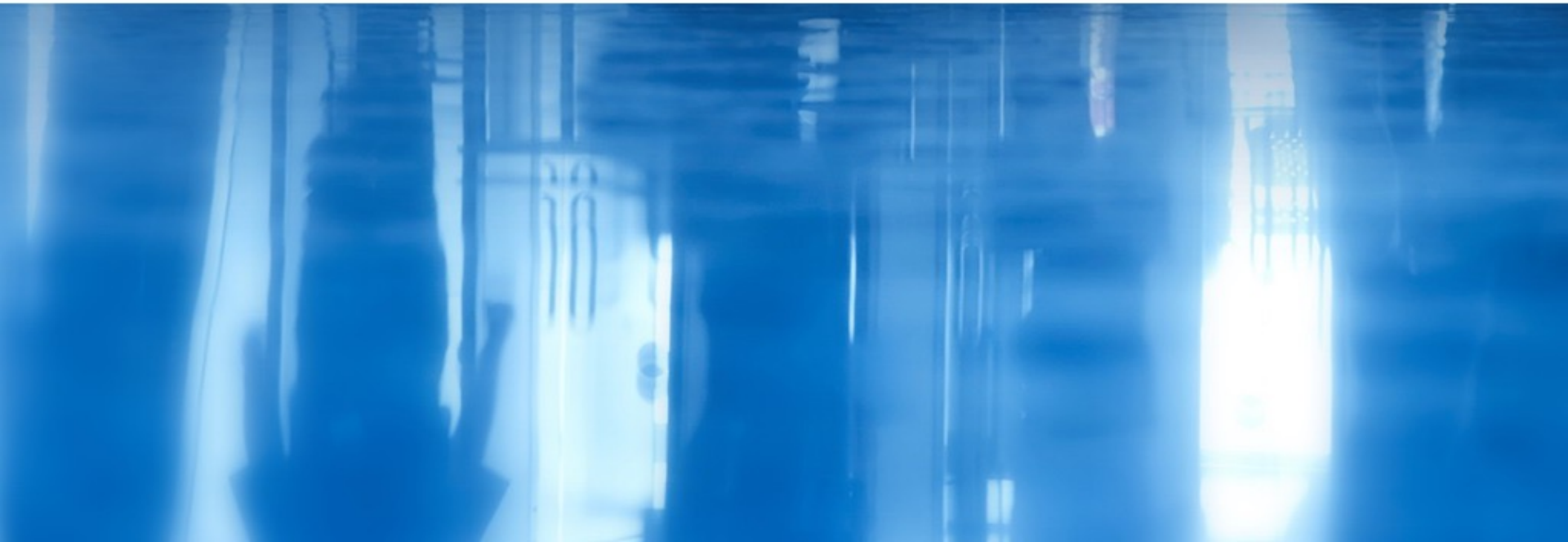
第4章

JSP技术模型



本章内容

- 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

4.1　JSP语法概述

- 在JSP页面中可以包含的元素如表4-1所示。

表4-1　JSP页面元素

JSP页面元素	简要说明	标签语法
声明	声明变量与定义方法	<%! Java 声明　　%>
小脚本	执行业务逻辑的Java代码	<% Java 代码　　%>
表达式	用于在JSP页面输出表达式的值	<%= 表达式　　%>
指令	指定转换时向容器发出的指令	<%@　指令　　%>
动作	向容器提供请求时的指令	<jsp:动作名　　/>
EL表达式	JSP 2.0引进的表达式语言	${applicationScope.email}
注释	用于文档注释	<%-- 任何文本　　--%>
模板文本	HTML标签和文本	同HTML规则

4.1　JSP语法概述

- 下面是一个简单的JSP页面，它输出页面被访问的次数。
- 程序4.1　counter.jsp

```
<%@ page contentType="text/html;charset = UTF-8" %>
<%! int count = 0; %>
<html><body>
 <%
   count++;
 %>
 该页面已被访问<%=count %> 次。
</body></html>
```

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　表达式语言
- 4.1.5　JSP注释

4.1.1 JSP脚本元素

- 在JSP页面中有三种脚本元素：
  - JSP声明
  - JSP小脚本
  - JSP表达式。

1. JSP声明

- JSP声明（declaration）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。下面是JSP声明的一个例子：

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

1. JSP声明

- 注意，由于声明包含的是声明语句，所以每个变量的声明语句必须以分号结束。

  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];
    }
 %>
```

1. JSP声明

- 也可以将上面的两个Java声明语句写在两个JSP声明标签中。

```
<%! String color[] = {"red", "green",
"blue"}; %>
<%!
  String getColor(int i){
      return color[i];
  }
%>
```

2. JSP小脚本

- 小脚本（scriptlets）是嵌入在JSP页面中的Java代码段。小脚本是以“<%”开头，以“%>”结束的标签。例如，程序4.1中的下面一行就是JSP小脚本。

```
<% 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。

2． JSP小脚本

- 由于小脚本可以包含任何Java代码，所以它通常用来在JSP页面嵌入计算逻辑。同时还可以使用小脚本打印HTML模板文本。如下面代码与程序4.1的代码等价：

```
<%@ page contentType="text/html;charset = UTF-8"%>
<%! int count = 0; %>
<%  out.print("<html><body>");
    count++;
    out.print("该页面已被访问" + count +"次。" );
    out.print("</body></html>");
%>
```

2. JSP小脚本

- 这里变量out是一个隐含对象，将在4.5节中讨论out对象。与其他元素不同，小脚本的起始标签“<%”后面没有任何特殊字符，在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，因为它没有使用分号结束。

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

3. JSP表达式

- JSP表达式是以“<%=”开头，以“%>”结束的标签，它作为Java语言表达式的占位符。
- 下面是JSP表达式的例子。

```
<%= count %>
```

- 在页面每次被访问时都要计算表达式，然后将其值嵌入到HTML的输出中。

3. JSP表达式

- 与变量声明不同，表达式不能以分号结束，因此下面的代码是非法的。

```
<%= count; %>
```

- 使用表达式可以向输出流输出任何对象或任何基本数据类型（int、boolean、char等）的值，也可以打印任何算术表达式、布尔表达式或方法调用返回的值。
- 提示：在JSP表达式的百分号和等号之间不能有空格

3. JSP表达式

- 下面代码声明了一些变量并通过表达式输出。
- 程序4.2　expression.jsp

4.1.2 JSP指令

- JSP指令向容器提供关于JSP页面的总体信息。在JSP页面中，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。

4.1.2 JSP指令

- 三种指令的语法格式如下：

```
<%@ page attribute-list %>
<%@ include attribute-list %>
<%@ taglib attribute-list %>
```

- 在上面的指令标签中，attribute-list表示一个或多个针对指令的属性/值对，多个属性之间用空格分隔。

1. page指令

- page指令通知容器关于JSP页面的总体特性。
- 例如，下面的page指令通知容器页面输出的内容类型和使用的字符集。

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

2. include指令

- **include指令**实现把另一个文件（HTML、JSP等）的内容包含到当前页面中。下面是include指令的一个例子

```
<%@ 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，下面是taglib指令的例子。

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

3. taglib指令

- 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

- JSP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 />

- 动作是一种标签，在动作标签中，prefix为前缀名，actionName为动作名，attribute-list表示针对该动作的一个或多个属性/值对。

4.1.3 JSP动作

- 在JSP页面中可以使用三种动作：
  - JSP标准动作
  - 标准标签库（JSTL）中的动作
  - 用户自定义动作
- 例如，下面一行指示容器把另一个JSP页面copyright.jsp的输出包含在当前JSP页面的输出中：

```
<jsp:include page="copyright.jsp" />
```

4.1.3 JSP动作

下面是常用的JSP标准动作：

jsp:include，在当前页面中包含另一个页面的输出。

jsp:forward，将请求转发到指定的页面。

jsp:useBean，查找或创建一个JavaBeans对象。

jsp:setProperty，设置JavaBeans对象的属性值。

jsp:getProperty，返回JavaBeans对象的属性值。

jsp:plugin，在JSP页面中嵌入一个插件（如Applet）。

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，它是一种可以在JSP页面中使用的简洁的数据访问语言。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。

```
${param.userName}
```

该EL显示请求参数userName的值。

4.1.5 JSP注释

- JSP注释的格式为：

<%-- 这里是JSP注释内容 --%>

- JSP注释是以“<%--”开头，以“--%>”结束的标签。注释不影响JSP页面的输出，但它对用户理解代码很有帮助。
- Web容器在输出JSP页面时去掉JSP注释内容，所以在调试JSP页面时可以将JSP页面中一大块内容注释掉，包括嵌套的HTML和其他JSP标签。

4.1.5 JSP注释

- 还可以在小脚本或声明中使用一般的Java风格的注释，也可以在页面的HTML部分使用HTML风格的注释，如下所示：

```
<% // 这里是Java 注释 %>
<!-- 这里是HTML 注释 -->
```

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

- 下面的JSP页面todayDate.jsp用于显示当前日期。下面以该页面为例说明JSP页面的生命周期的7个阶段。
- 程序4.3 todayDate.jsp
- 当客户通过 http://localhost:8080/helloweb/todayDate.jsp
- 首次访问该页面时，Web容器执行该JSP页面要经过7个生命周期阶段，如图4-1所示。

4.2.2 JSP生命周期阶段

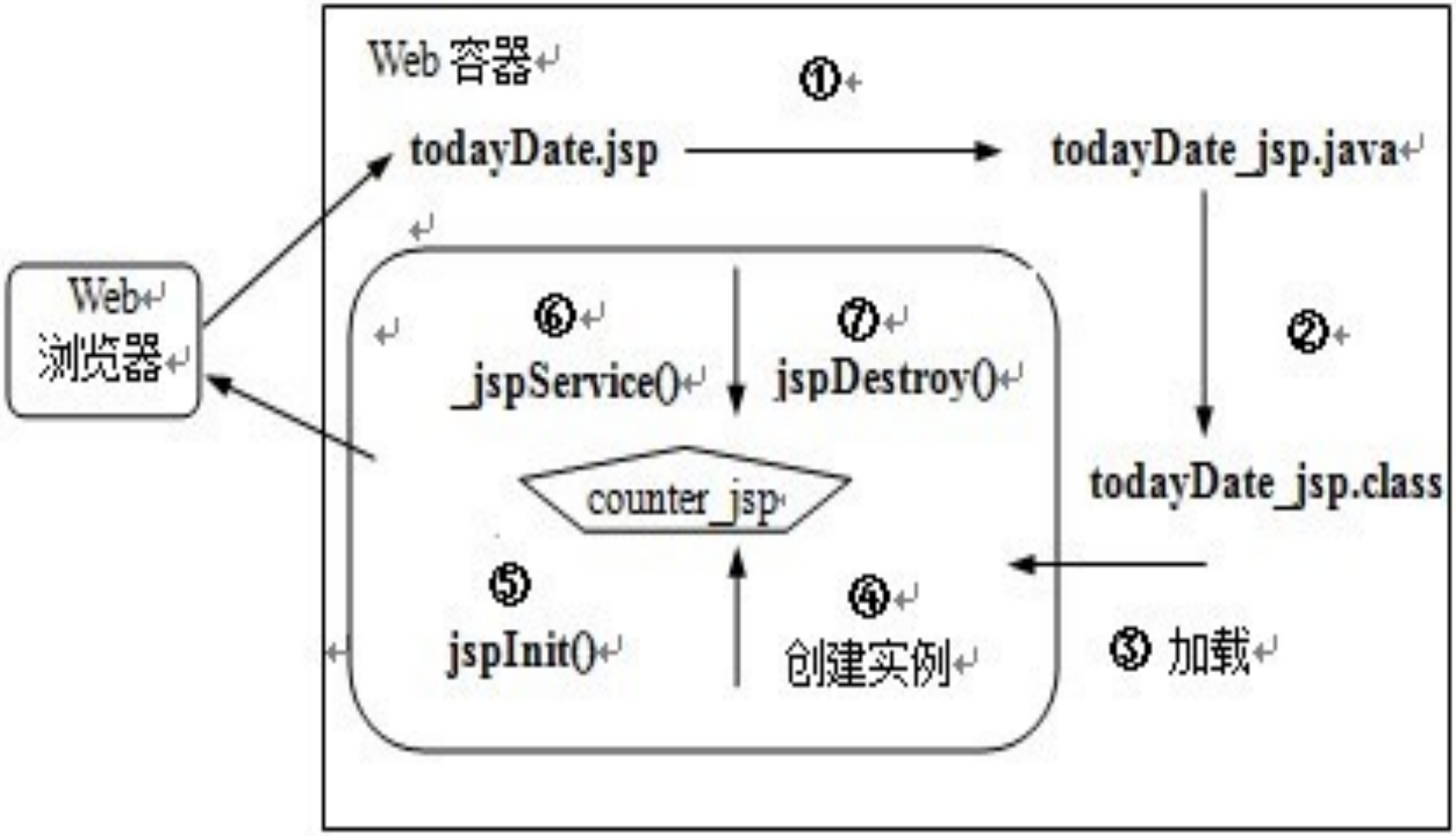


图 4-1 JSP 页面生命周期阶段

4.2.2 JSP生命周期阶段

- JSP生命周期的前四个阶段将JSP页面转换成一个Servlet类并装载和创建该类实例，后三个阶段初始化、提供服务和销毁阶段。

表4-2按生命周期的顺序列出了每个阶段及说明。

表4-2　JSP页面生命周期阶段

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

1. 转换阶段

- Web容器读取JSP页面对其解析，并将其转换成Java源代码。JSP文件中的元素都转换成页面实现类的成员。在这个阶段，容器将检查JSP页面中标签的语法，如果发现错误将不能转换。例如，下面的指令就是非法的，因为在Page中使用了大写字母P，这在转换阶段被捕获。

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

- 除了检查语法外，容器还将执行其他有效性检查，其中一些涉及验证：

1. 转换阶段

- 一旦验证完成，Web容器将JSP页面转换成Java源文件，它实际是一个Servlet，该文件存放在<*tomcat-install*>\work\Catalina\localhost\helloweb\org\apache\jsp目录中。

1. 转换阶段

- JspPage接口只声明了两个方法：jspInit()和jspDestroy()。所有的JSP页面都应该实现这两个方法。HttpJspPage接口中声明了一个方法：_jspService()。下面是这三个JSP方法的格式：

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

1. 转换阶段

- 每个容器销售商都提供一个特定的类作为页面实现类的基类。在Tomcat中，JSP页面转换的类就继承了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。

2. 编译阶段

- 在将JSP页面转换成Java文件后，Web容器调用Java编译器javac编译该文件。在编译阶段，编译器将检查在声明中、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 %>
```

2. 编译阶段

- 可能注意到，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。

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。

4. 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

4.2.3 JSP生命周期方法示例

- 当Web容器首次装入页面时，它将调用jspInit()。在JSP页面的生命周期中，count变量可能被多次访问，每次都将执行_jspService()。由于小脚本<% count++; %>变成_jspService()的一部分，count++每次都会被执行使计数器增1。最后，当页面被销毁时，容器调用jspDestroy()。

  注意，_jspService()不能被覆盖。

4.2.4 理解页面转换过程

- JSP页面生命周期的第一阶段是转换阶段，在该阶段JSP页面被转换成包含相应Servlet的Java文件。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。

4.2.4 理解页面转换过程

- 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

- 在JSP页面中可以使用<%@ include … %>指令把另一个文件（如JSP页面、HTML页面等）的内容包含到当前页面中。容器在为当前JSP页面产生Java代码时，它也把被包含的文件的内容插入到产生的页面实现类中。
- 这些被转换成单个页面实现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在的页面。

4.2.5 理解转换单元

- 关于转换单元，请记住下面要点：
  - page指令影响整个转换单元。有些指令通知容器关于页面的总体性质，例如，page指令的contentType属性指定响应的内容类型，session属性指定页面是否参加HTTP会话。
  - 在一个转换单元中一个变量不能多次声明。例如，如果一个变量已经在主页面中声明，它就不能在被包含的页面中声明。
  - 在一个转换单元中不能使用<jsp:useBean>动作对一个bean声明两次。
  - http://blog.csdn.net/javaxiaochouyu/article/details/6306200

4.3 理解page指令属性

- page指令用于告诉容器关于JSP页面的总体特性，该指令适用于整个转换单元而不仅仅是它所声明的页面。表4-3描述了page指令的常用的属性。

属性名	说明	默认值
import	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.*;
contentType	指定输出的内容类型和字符集	text/html; charset=ISO-8859-1
pageEncoding	指定JSP文件的字符编码	ISO-8859-1
session	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
buffer	指定输出缓冲区的大小	与实现有关
autoFlush	指定是否当缓冲区满时自动刷新	true
info	关于JSP页面的任何文本信息	与实现有关
isThreadSafe	指定页面是否同时为多个请求服务	true
isELIgnored	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

- import属性的功能类似于Java程序的import语句，它是将import属性值指定的类导入到页面中。在转换阶段，容器对使用import属性声明的每个包都转换成页面实现类的一个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：

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

4.3.1 import属性

- 为了增强代码可读性也可以使用多个page指令，如上面的page指令也可以写成：

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

4.3.1 import属性

- 由于在Java程序中import语句的顺序是没有关系的，因此这里import属性的顺序也没有关系。
- 容器总是导入java.lang.*、javax.servlet.*、javax.servlet.http.*和javax.servlet.jsp.* 包，所以不必明确地导入它们。

4.3.2 contentType和pageEncoding属性

- contentType属性用来指定JSP页面输出的MIME类型和字符集，MIME类型的默认值是text/html，字符集的默认值是ISO-8859-1。MIME类型和字符集之间用分号分隔，如下所示。

```
<%@ page contentType="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" %>
```

4.3.2 contentType和pageEncoding属性

- pageEncoding属性指定JSP页面的字符编码，它的默认值为ISO-8859-1。如果设置了该属性，则JSP页面使用该属性设置的字符集编码；如果没有设置这个属性，则JSP页面使用contentType属性指定的字符集。如果页面中含有中文，应该将该属性值指定为UTF-8，如下：

```
<%@ 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（将在4.5节学习更多的隐含变量）。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

- 在页面执行过程中，嵌入在页面中的Java代码可能抛出异常。与一般的Java程序一样，在JSP页面中也可以使用try-catch块处理异常。然而，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。
- 在该方法中，JSP页面使用page指令的errorPage属性将异常代理给另一个包含错误处理代码的JSP页面。

4.3.4 errorPage与isErrorPage属性

- 程序4.5的helloUser.jsp页面中，errorHandler.jsp被指定为错误处理页面。
- 程序4.5 helloUser.jsp
- 对该JSP页面的请求如果指定了name请求参数值，该页面将正常输出，如果没有指定name请求参数值，将抛出一个异常，但它本身并没有捕获异常，而是通过errorPage属性指示容器将错误处理代理给页面errorHandler.jsp。

4.3.4 errorPage与isErrorPage属性

- errorPage的属性值不必是JSP页面，它也可以是静态的HTML页面，例如：

```
<%@ page
errorPage="errorHandler.html" %>
```

- 显然，在errorHandler.html文件中不能编写小脚本或表达式产生动态信息。

4.3.4 errorPage与isErrorPage属性

- isErrorPage属性指定当前页面是否作为其他JSP页面的错误处理页面。isErrorPage属性的默认值为false。如上例使用的errorHandler.jsp页面中该属性必须明确设置为true，如下所示。
- 程序4.6 errorHandler.jsp

4.3.4 errorPage与isErrorPage属性

- 在这种情况下，容器在页面实现类中声明一个名为exception的隐含变量。
- 注意，该页面仅从异常对象中检索信息并产生适当的错误消息。因为该页面没有实现任何业务逻辑，所以可以被不同的JSP页面重用。
- 一般来说，为所有的JSP页面中指定一个错误页面是一个良好的编程习惯，这可以防止在客户端显示不希望的错误消息。

4.3.4 errorPage与isErrorPage属性

- 注意：如果不带参数请求helloUser.jsp页面，浏览器可能显示“无法显示网页”的页面。
- 此时，可以打开“工具”菜单的“Internet选项”对话框，在“高级”选项卡中，将“浏览”组中的“显示友好HTTP错误信息”的复选框取消，再重新访问页面，则显示JSP页面中指定的错误页面。

4.4 JSP脚本元素

- 由于声明、小脚本和表达式允许在页面中编写脚本语言代码，所以这些元素统称为脚本元素。
- JSP页面使用Java语言作为脚本语言，因此脚本元素中代码的编译和运行行为受到Java编程语言的控制。

4.4.1　变量的声明及顺序

1. 声明的顺序

- 因为在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说明了这一点。

  程序4.7　area.jsp

- 在该例中，尽管半径r、求面积的方法area()是在使用之后定义的，但不仅能够转换、编译和运行，其运行结果如图4-2所示。

2. 小脚本的顺序

- 由于小脚本被转换成页面实现类的_jspService()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，故它们出现的顺序很重要，下面代码说明了这一点。

```
<% String s = s1 + s2; %>
<%! String s1 = "hello"; %>
<% String s2 = "world"; %>
<% out.print(s); %>
```

- 该例中，s1是在声明中定义的，它成为页面实现类的成员变量，s与s2是在小脚本中声明的，它们成为_jspService()的局部变量。s2在声明之前使用，因此该代码将不能被编译。

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。因此在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化，请看下面的代码。

```
<%! int i; %>
<% int j; %>
The value of i is <%= ++i %><br>
The value of j is <%= ++ j %><br>
```

- 变量i是使用声明（<%!...%>）声明的，它成为产生的Servlet类的实例变量并被初始化为0。变量j是使用小脚本（<%...%>）声明的，它变成产生的_jspService()的局部变量并没有被初始化。

3. 变量的初始化

- 由于Java要求局部变量在使用之前明确初始化，因此上述代码是非法的且不能编译。
- 需要注意的是，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。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。
- 要使上面的代码能够编译，可以像下面这样初始化变量j。

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

- 如果多次访问上面页面，i的值将每次增1，输出一个新值，而j的值总是输出1。

4.4.2 使用条件和循环语句

- 小脚本用来在JSP页面中嵌入计算逻辑，通常这种逻辑包含条件和循环语句。例如，下面的脚本代码使用了条件语句检查用户的登录状态，并基于该状态显示适当的消息。

```
<%
    String username =
request.getParameter("username");
    String password =
request.getParameter("password");
    boolean isLoggedIn = false;
if(username.equals("admin")&&password.equals("admi
n"))
```

4.4.2 使用条件和循环语句

```
        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>");
    }
%>
```

4.4.2 使用条件和循环语句

- 如果在条件语句中包含大量HTML代码，可以使条件语句跨越JSP页面多个小脚本，从而避免书写多个out.println()语句。

```
<%
    if (isLoggedIn){
%>
<h3>欢迎你，<%=username%> 访问该页面！ </h3>
   这里可包含其他HTML代码！
<%
  }else{
%>
```

4.4.2 使用条件和循环语句

```
 你还没有登录！<br>
<a href="login.jsp">登录</a>
这里可包含其他HTML代码！
<%
 }
%>
```

- 上面的代码段中，if-else语句跨越三段小脚本。运行时，如果变量的值为true，则将第一和第二段小脚本之间的HTML代码包含在输出中，如果该值为false，则将第二和第三段小脚本之间的HTML代码包含在输出中。

4.4.2 使用条件和循环语句

- 注意，大括号的用途是标记Java编程语言的代码块的开始和结束。忽略大括号可能在编译时产生错误，在运行时引起未知行为，例如代码：

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

- 将被转换成：

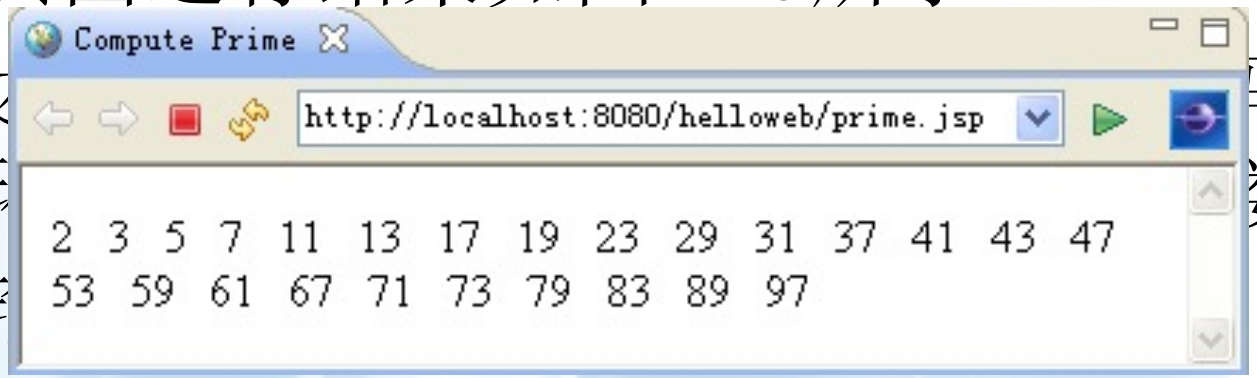
```
if (isLoggedIn)
out.write("欢迎你，");
out.print(username);
out.write("访问该页面！ </h3>");
```

4.4.2 使用条件和循环语句

- 与条件语句一样，循环语句也可以跨越多段小脚本，使常规的HTML代码段处于小脚本之间。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

- 该页面运行结果如图4-3所示。



- 上这 码包含在循环 数n，之后转

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JSP动作传递属性值，例如：

```
<%! String pageURL = "copyright.jsp"; %>
<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。以这种方式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。

4.4.3　请求时属性表达式的使用

- 注意，请求时属性表达式不能用在指令的属性中，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。因此，下例中的指令是非法的：

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

4.5 JSP隐含变量

- 在JSP页面的转换阶段，Web容器在_jspService()中声明并初始化一些变量，可以在JSP页面小脚本中或表达式中直接使用这些变量，例如：

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

- 该段小脚本中使用out对象的print()输出一个字符串。out对象是由容器隐含声明的，所以一般被称为**隐含对象**（implicit objects），这些对象是由容器创建，可象变量一样使用，因此也被叫做**隐含变量**（implicit variables）。

表4-4　JSP页面中可使用的隐含变量

隐含变量	类或接口	说明
application	javax.servlet.ServletContext接口	引用Web应用程序上下文
session	javax.servlet.http.HttpSession接口	引用用户会话
request	javax.servlet.http.HttpServletRequest接口	引用页面的当前请求对象
response	javax.servlet.http.HttpServletResponse接口	用来向客户发送一个响应
out	javax.servlet.jsp.JspWriter类	引用页面输出流
page	java.lang.Object类	引用页面的Servlet实例
pageContex	javax.servlet.jsp.PageContext类	引用页面上下文
config	javax.servlet.ServletConfig接口	引用Servlet的配置对象
exception	java.lang.Throwable类	用来处理错误

4.5 JSP隐含变量

- 以todayDate.jsp页面为例，在转换阶段，Web容器自动将该页面转换成一个名为todayDate_jsp.java的类文件，该文件位于<tomcat-install>\work\Catalina\localhost\helloweb\org\apache\jsp目录中，该文件就是JSP页面实现类。
- 程序4.9 todayDate_jsp.java
- 可以看到在 jspService中声明了8个变量（加粗字体表示的）。

4.5 JSP隐含变量

- 如果一个页面是错误处理页面，即页面中包含下面的page指令。

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

- 则页面实现类中将自动声明一个exception隐含变量，如下所示。

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

下面详细讨论这9个隐含变量的使用。

- 注意，这些隐含变量只能在JSP的小脚本和表达式中使用。

4.5.1 request与response变量

- request是HttpServletRequest类型的隐含变量，response是HttpServletResponse类型的隐含变量，当页面的Servlet向客户提供服务时，它们作为参数传递给_jspService()。
- 在JSP页面中使用它们与在Servlet中使用完全一样，即用来分析请求和发送响应，如下代码所示。

4.5.1 request与response变量

```
<%
 String remoteAddr = request.getRemoteAddr();
 response.setContentType("text/html;charset=UTF-8");
%>
你的IP地址为：<%=remoteAddr%><br>
你的主机名为：<%=request.getRemoteHost()%>
```

4.5.2 out变量

- out是javax.servlet.jsp.JspWriter类型的隐含变量，打印输出所有的基本数据类型、字符串以及用户定义的对象。可以在小脚本中直接使用它，也可在表达式中间接使用它产生HTML代码，例如：

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

- 对上面两行代码，在页面实现类中都使用out.print()语句输出。

4.5.2 out变量

- 下面的脚本使用out变量的print()打印输出不同类型的数据。

```
<% int anInt = 3;
   Float aFloatObj = new Float(5.6);

   out.print(anInt);
   out.print(anInt > 0);
   out.print(anInt*3.5/100-500);
   out.print(aFloatObj);
   out.print(aFloatObj.floatValue());
   out.print(aFloatObj.toString());
%>
```

4.5.3 application变量

- application是javax.servlet.ServletContext类型的隐含变量，它是JSP页面所在的Web应用程序的上下文的引用（ServletContext接口），下面两段小脚本是等价的。

4.5.3 application变量

```
<%
  String path =
application.getRealPath("/WEB-
INF/counter.db");
  application.log("绝对路径为:"+path);
%>
<%
String path =
getServletContext().getRealPath("/WEB-
INF/counter.db");
getServletContext().log("绝对路径为:"+path);
%>
```

4.5.4 session变量

- session是javax.servlet.http.HttpSession类型的隐含变量，它在JSP页面中表示会话对象。要使用会话对象，必须要求JSP页面参加HTTP会话，即要求将JSP页面的page指令的session属性值设置为true。
- 默认情况下，session属性的值为true。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，如下所示。

```
<%@ page session = "false" %>
<html><body>
    会话ID = <%=session.getId()%>
</body></html>
```

4.5.5 pageContext变量

- pageContext是javax.servlet.jsp.PageContext类型的隐含变量，它是一个页面上下文对象。PageContext类是一个抽象类，容器厂商提供了一个具体子类（如JspContext），它有下面三个作用。

4.5.5 pageContext变量

- （1）存储隐含对象的引用。pageContext对象是作为管理所有在JSP页面中使用的其他对象的一个地方，包括用户定义的和隐含的对象，并且它提供了一个访问方法来检索它们。
- （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()和include()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，它们的格式如下。

4.5.5 pageContext变量

- public void include(String relativeURL)：将另一个资源的输出包含在当前页面的输出中，与RequestDispatcher()接口的include()功能相同。
- public void forward(String relativeURL)：将请求转发到参数指定的资源，与RequestDispatcher接口的forward()功能相同。

4.5.5 pageContext变量

- 例如，从Servlet中将请求转发到另一个资源，需要写下面两行代码。

```
RequestDispatcher rd =
request.getRequestDispatcher("other.jsp");
rd.forward(request, response);
```

- 在JSP页面中，通过使用pageContext变量仅需一行就可以完成上述功能。

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

4.5.6 page变量

- page变量是java.lang.Object类型的对象，声明如下

```
Object page = this;
```

- 它指的是生成的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。可通过DD文件为Servlet传递一组初始化参数，然后在Servlet中使用ServletConfig对象检索这些参数。
- 类似地，也可以为JSP页面传递一组初始化参数，这些参数在JSP页面中可以使用config隐含变量来检索。

4.5.8 exception变量

- exception是java.lang.Throwable类型的隐含变量，它被用来作为其他页面的错误处理器。为使页面能够使用exception变量，必须在page指令中将isErrorPage的属性值设置为true，如下所示。

  程序4.10 errorPage.jsp

- 该页面中，将page指令的isErrorPage属性设置为true，容器明确定义了exception变量。该变量指向使用该页面作为错误处理器的页面抛出的未捕获的java.lang.Throwable对象。

4.6 作用域对象

- 在JSP页面中有四个作用域对象，它们的类型分别是ServletContext、HttpSession、HttpServletRequest和PageContext，这四个作用域分别称为：
  - 应用（application）作用域
  - 会话（session）作用域
  - 请求（request）作用域
  - 页面（page）作用域，
- 如表4-5所示。

4.6 作用域对象

- 在JSP页面中，所有的隐含对象以及用户定义的对象都处于这四种作用域之一，这些作用域定义了对象存在性和从JSP页面和Servlet中的可访问性。
- 应用作用域对象具有最大的访问作用域，页面作用域对象具有最小的访问作用域。

4.6 作用域对象

表4-5 JSP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	application	在整个Web应用程序有效
会话作用域	session	在一个用户会话范围内有效
请求作用域	request	在用户的请求和转发的请求内有效
页面作用域	pageContext	只在当前的页面（转换单元）内有效

4.6.1 应用作用域

- 存储在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。这些对象是通过ServletContext实例作为“属性/值”对维护的。
- 在JSP页面中，该实例可以通过隐含对象application访问。因此，要在应用程序级共享对象，可以使用ServletContext接口的setAttribute()和getAttribute()。

4.6.1 应用作用域

- 例如，在Servlet使用下面代码将对象存储在应用作用域中：

```
Float one = new Float(42.5) ;
ServletContext context = getServletContext();
context.setAttribute("foo", one);
```

在JSP页面中就可使用下面代码访问context中数据：

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

4.6.2 会话作用域

- 存储在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。
- 这些对象是通过HttpSession类的一个实例作为“属性/值”对维护的。在JSP页面中该实例可以通过隐含对象session访问。
- 因此，要在会话级共享对象，可以使用HttpSession接口的setAttribute()和getAttribute()。

4.6.2 会话作用域

- 在购物车应用中，用户的购物车对象就应该存放在会话作用域中，它在整个的用户会话中共享。

```
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

- 存储在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。这些对象是由HttpServletRequest对象作为“属性/值”对维护的。
- 在JSP页面中，该实例是通过隐含对象request的形式被使用的。通常，在Servlet中使用请求对象的setAttribute()将一个对象存储到请求作用域中，然后将请求转发到JSP页面，在JSP页面中通过脚本或EL取出作用域中的对象。

4.6.3 请求作用域

- 例如，下面代码在Servlet中创建一个User对象并存储在请求作用域中，然后将请求转发到valid.jsp页面。

```
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");
    }
%>
```

4.6.3 请求作用域

- 这里，valid.jsp页面根据数据库验证用户信息，最后根据验证处理的结果，或者将对象传输给会话作用域并将请求转发给account.jsp，或者将请求转发给loginError.jsp，它可以使用User对象产生一个适当的响应。

4.6.4 页面作用域

- 存储在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。这些对象是由PageContext抽象类的一个具体子类的一个实例通过属性/值对维护的。
- 在JSP页面中，该实例可以通过隐含对象pageContext访问。
- 为在页面作用域中共享对象，可以使用javax.servlet.jsp.PageContext定义的两个方法，其格式如下：

```
public void setAttribute(String name , Object value)
public Object getAttribute(String name )
```

4.6.4 页面作用域

- 下面代码设置一个页面作用域的属性：

```
<% Float one = new Float(42.5);%>
<% pageContext.setAttribute("foo",
one );%>
```

- 下面代码获得一个页面作用域的属性：

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

4.6.4 页面作用域

- PageContext类中还定义了几个常量和其他属性处理方法，使用它们可以方便地处理不同作用域的属性。该类定义的常量有：

public static final int APPLICATION_SCOPE，表示application作用域。

public static final int SESSION_SCOPE，表示session作用域。

4.6.4 页面作用域

public static final int REQUEST_SCOPE，表示request作用域。

public static final int PAGE_SCOPE，表示page作用域。

4.6.4 页面作用域

- 该类定义的方法有：

  public void setAttribute(String name, Object object, int scope)：在指定的作用域中设置属性。

  public Object getAttribute(String name, int scope)：返回在指定作用域中名为name的对象，没有找到则返回null。

4.6.4 页面作用域

public Object findAttribute(String name)：查找指定名称的属性值。查找顺序为页面作用域、请求作用域、会话作用域（若有效）、应用作用域。

public int getAttributesScope(String name)：返回给定属性的作用域。

4.6.4 页面作用域

- 下面是使用pageContext设置和返回属性的示例。
- 使用pageContext设置一个会话作用域的属性：

```
<% Float two = new Float(22.5) ;%>
<% pageContext.setAttribute(
    "foo", two,
PageContext.SESSION_SCOPE ) ;%>
```

4.6.4 页面作用域

- 使用pageContext获得一个会话作用域的属性：

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

- 上面一行等价于：

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

4.6.4 页面作用域

- 使用pageContext获得一个应用作用域的属性：

```
Email is:
<%=pageContext.getAttribute("email",PageContext.APPLICATION_SCOPE)%>
```

- 上述代码等价于：

```
Email is:
<%=application.getAttribute("email")%>
```

- 使用pageContext，即使不知道作用域也可以查找一个属性，例如：

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

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。

4.7.1 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。使用JSP的include指令完成这一功能，它的语法为：

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

- file属性是include指令唯一的属性，它是指被包含的文件。文件使用相对路径指定，相对路径或者以斜杠（/）开头，是相对于Web应用程序文档根目录的路径，或者不以斜杠开头，它是相对于当前JSP文件的路径。被包含的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。

图4-4说明了include指令的工作方式

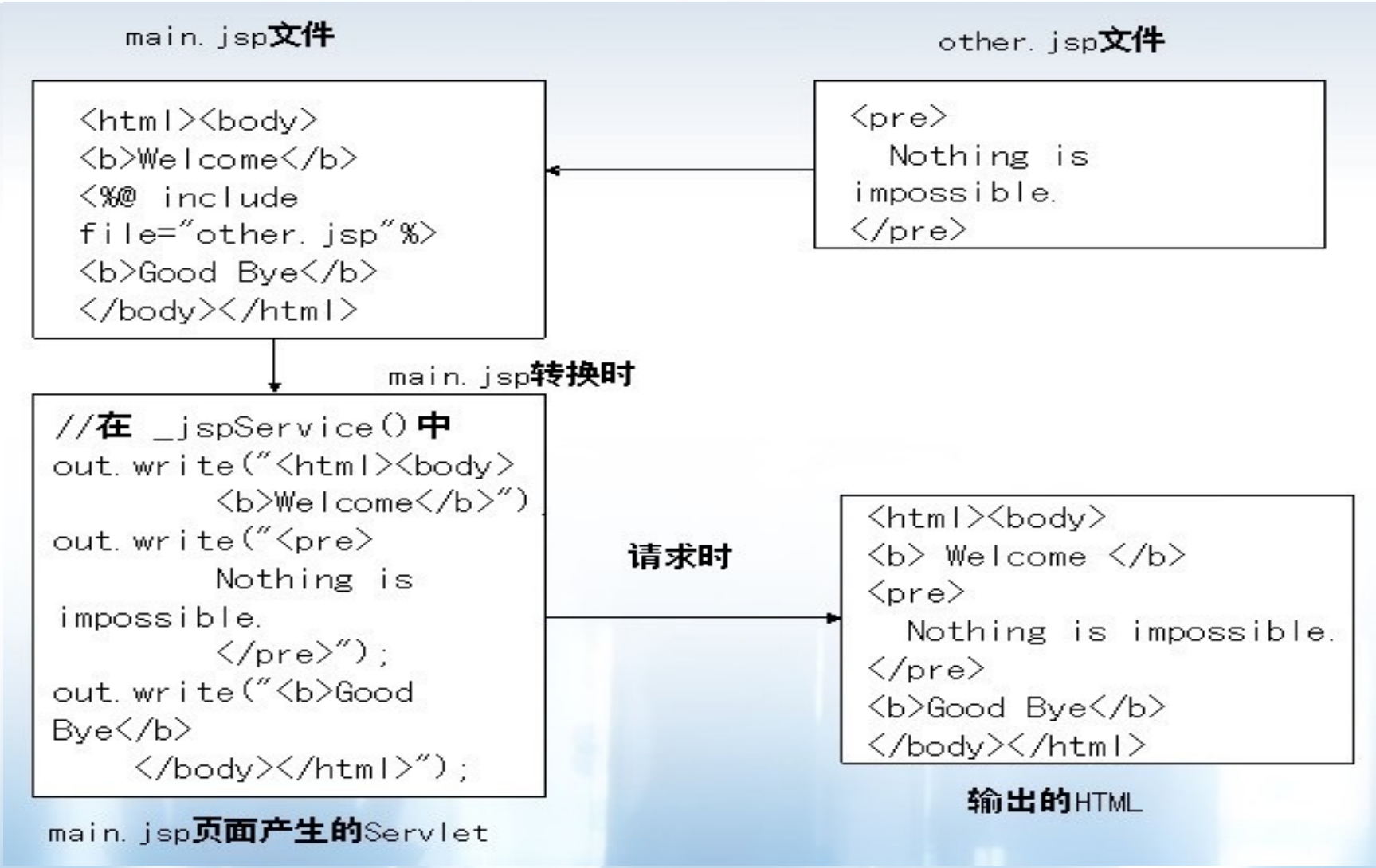


图4-4 使用include指令的静态包含

1. 从被包含页面中访问变量

- 由于被包含JSP页面的代码成为主页面代码的一部分，因此，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变量，如程序4.11所示。

  程序4.11　hello.jsp

- 下面代码是被包含页面response.jsp。

  程序4.12

图 4-5　hello.jsp 的运行结果

2. 静态包含的限制

- 当使用include指令包含一个文件时，需要遵循下列几个规则。

（1）在转换阶段不进行任何处理，这意味着file属性值不能是请求时表达式，因此下面的使用是非法的。

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

2. 静态包含的限制

（2）不能通过file属性值向被包含的页面传递任何参数，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。下面例子中的file属性值是非法的。

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

（3）被包含的页面可能不能单独编译。一般来说，最好避免这种依赖性，而使用隐含变量pageContext共享对象，通过使用pageContext的setAttribute()和getAttribute()实现。

4.7.2 动态包含：include动作

- 动态包含是通过JSP标准动作<jsp:include>实现的。该动作的格式如下：

```
<jsp:include page=“relativeURL”
        flush="true|false" />
```

- 这里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JSP页面、Servlet等。可选的flush属性是指在将控制转向被包含页面之前是否刷新主页面。flush属性的默认值为false。

4.7.2 动态包含：include动作

- page属性的值可以是请求时表达式，例如：

```
<%! String pageURL = "other.jsp"; %>
<jsp:include page="<%= pageURL %>" />
```

- 图4-6说明了<jsp:include>的工作方式。

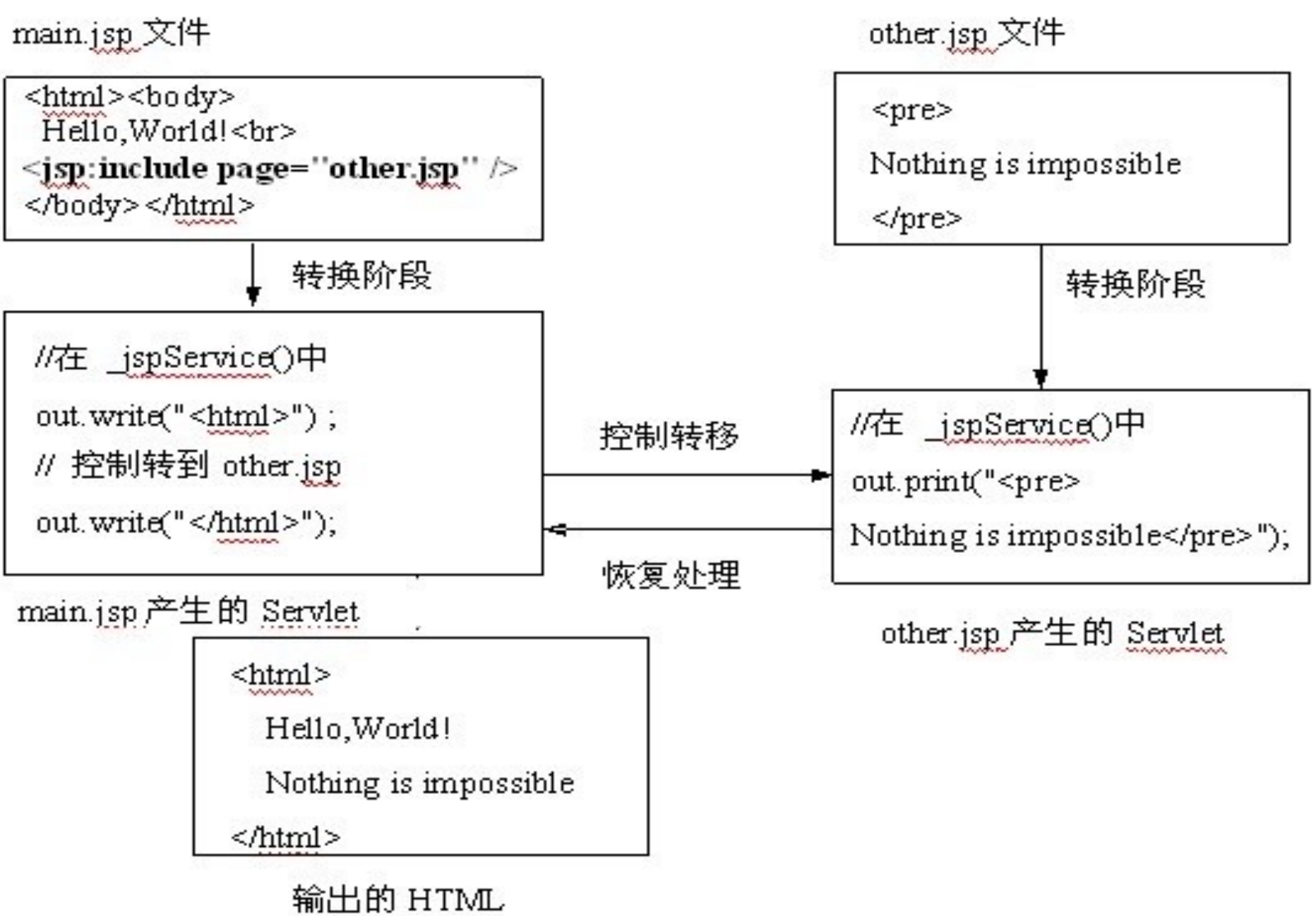


图 4-6 使用 include 动作的动态包含

4.7.2 动态包含：include动作

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

【结构1】

```
<%
RequestDispatcher rd =
 request.getRequestDispatcher("ot
 her.jsp");
 rd.include(request, response);
%>
```

4.7.2 动态包含：include动作

【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 />向被包含的页面传递参数。下面的代码向somePage.jsp页面传递两个参数：

```
<jsp:include page="somePage.jsp">
    <jsp:param name=“name1”
               value="value1" />
    <jsp:param name=“name2”
               value="value2" />
</jsp:include>
```

1. 使用<jsp:param>传递参数

- 在<jsp:include>元素中可以嵌入任意多的<jsp:param>元素。value的属性值也可以像下面这样使用请求时属性表达式来指定。

```
<jsp:include page="somePage.jsp">
        <jsp:param name="name1"
            value="<%= someExpr1 %>" />
        <jsp:param name="name2"
            value="<%= someExpr2 %>" />
</jsp:include>
```

1. 使用<jsp:param>传递参数

- 通过<jsp:param>动作传递的“名/值”对保存在request对象中并只能由被包含的组件使用，在被包含的页面中使用request隐含对象的getParameter()获得传递来的参数。
- 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。
- 上面的例子使用的是<jsp:include>动作，但这里的讨论也适用于<jsp:forward>动作。

2. 与动态包含的组件共享对象

- 被包含的页面是单独执行的，因此它们不能共享在主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。下面程序说明了这一点。

程序4.13　hello2.jsp

程序4.13产生的输出结果与程序4.11相同，但它使用了动态包含而不是静态包含。主页面hello2.jsp通过调用request.setAttribute()把userName对象添加到请求作用域中，然后，被包含的页面response2.jsp通过调用request.getAttribute()检索该对象并使

程序4.14　response2.jsp

这里，在hello2.jsp文件中的隐含变量request与response2.jsp文件中的隐含变量request是请求作用域内的同一个对象。对<jsp:forward>动作可以使用相同的机制。

除request对象外，还可以使用隐含变量session和application在被包含的页面中共享对象，但它们的作用域是不同的。例如，如果使用application代替request，那么username对象就可被多个客户使用。

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。它与<jsp:include>动作的不同之处在于，当转发到的页面处理完输出后，并不将控制转回主页面。使用<jsp:forward>动作，主页面也不能包含任何输出。

4.7.3 使用<jsp:forward>动作

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

【结构1】

```
<%
   RequestDispatcher rd =

request.getRequestDispatcher("other.jsp");
      rd.forward(request, response);
%>
```

4.7.3 使用<jsp:forward>动作

【结构2】

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

【结构3】

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

- 在JSP页面中使用<jsp:forward>标准动作实际上实现的是控制逻辑的转移。在MVC体系结构中，控制逻辑应该由控制器（Servlet）实现而不应该由视图（JSP页面）实现。因此，尽可能不在JSP页面中使用<jsp:forward>动作转发请求。

4.7.4 实例：使用包含设计页面布局

- Web应用程序界面应该具有统一的视觉效果，所有的页面都有同样的整体布局。一种比较典型的布局通常包含标题部分、脚注部分、菜单、广告区和主体实际内容部分。
- 设计这些页面时如果在所有的页面中都复制相同的代码，这不仅不符合模块化设计原则，将来若修改布局也非常麻烦。
- 使用JSP技术提供的include指令（<%@ include…）包含静态文件和include动作（<jsp:include …>）包含动态资源就可以实现一致的页面布局。

4.7.4 实例：使用包含设计页面布局

- 下面的index.jsp页面使用 标签和include指令实现页面布局。

  程序4.15 index.jsp

- 访问该页面，输出结果如图4-7所示。

4.7.4 实例：使用包含设计页面布局

4.7.4　实例：使用包含设计页面布局

- 下面是标题页面header.jsp、左侧菜单页面leftmenu.jsp、主体内容页面content.jsp和页脚页面footer.jsp。

程序4.16　header.jsp

程序4.17　leftmenu.jsp

程序4.18　content.jsp

程序4.19　footer.jsp

4.7.4 实例：使用包含设计页面布局

- 在上面这些被包含的文件中，没有使用<html>和<body>等标签。实际上，它们不是完整的页面，而是页面片段，因此文件名也可以完全不使用.jsp作为扩展名，而可以使用任何的扩展名，如.htmlf或.jspf等。
- 由于被包含的文件是由服务器访问的，因此可以将被包含的文件存放到Web应用程序的WEB-INF目录中，这样可以防止用户直接访问被包含的文件。

4.8 JavaBeans

- JavaBeans是Java平台的组件技术，在Java Web开发中常用JavaBeans来存放数据、封装业务逻辑等，从而很好地实现业务逻辑和表示逻辑的分离，使系统具有更好的健壮性和灵活性。
- 对程序员来说，JavaBeans最大的好处是可以实现代码的重用，另外对程序的易维护性等也有很大的意义。

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。

4.8.1 JavaBeans规范

（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()。

4.8.1 JavaBeans规范

- 例如，假设JavaBeans类中有一个String类型的color属性，
- 下面是访问方法和修改方法的定义：

```
public String getColor(){
    return this.color;
}
public void setColor(String color){
    this.color = color;
}
```

4.8.1 JavaBeans规范

- 除了访问方法和修改方法外，JavaBeans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方法，这样的属性就是只读属性。
- 提示：JavaBeans与EJB不是一回事。
- 下面的Customer类使用三个private属性封装了客户信息，并提供了访问和修改这些信息的方法。

  程序4.20 Customer.java

4.8.1 JavaBeans规范

- 使用JavaBeans的优点是：在JSP页面中使用JavaBeans可使代码更简洁；JavaBeans有助于增强代码的可重用性；它们是Java语言对象，可以充分利用该语言面向对象的特征。

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

4.8.2 使用<jsp:useBean>动作

- <jsp:useBean>动作用来在JSP页面中查找或创建一个bean实例。一般格式如下：

```
<jsp:useBean  id="beanName"
   scope="page|request|session|application"
     {class="package.class" |
       type="package.class" |
     }
 { /> | >其他元素</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。
- 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

- 下面动作仅使用了id和class属性声明一个JavaBeans:

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

- 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

- 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" />
```

- 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。这个过程与下面的代码等价:

2. 属性的使用

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

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。

4.8.3 使用<jsp:setProperty>动作

- <jsp:setProperty>动作用来给bean实例的属性赋值，它的格式如下。

```
<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

- 下面动作将名为customer的custName、email和phone属性值分别设置为"Mary"、"mary@163.com"和"8899123"。

```
 <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。

4.8.4 使用<jsp:getProperty>动作

- <jsp:getProperty>动作检索并向输出流中打印bean的属性值，它的语法非常简单。

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

- 该动作只有两个属性name和property，并且都是必须的。name属性指定bean实例名，property属性指定要输出的属性名。
- 下面的动作指示容器打印customer的email和phone属性值。

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

4.8.5 JavaBeans应用示例

- 下面示例首先在inputCustomer.jsp中输入客户信息，然后将控制转到CustomerServlet，最后将请求转发到displayCustomer.jsp页面。

  程序4.21 inputCustomer.jsp

- 在Servlet代码中创建JavaBeans类的实例，以及如何使用作用域对象共享它们。可以直接在Servlet中使用JavaBeans。并且可以在JSP页面中和Servlet中共享bean实例。

  程序4.22 CustomerServlet.java

- 这个例子说明在Servlet中可以把JavaBeans对象存储到作用域对象中。这里需要注意的是会话作用域对象的访问使用了同步（synchronized）代码块，这是因为

4.8.5 JavaBeans应用示例

HttpSession对象不是线程安全的，其他Servlet和JSP页面可能在多个线程中同时访问或修改这些对象。

- 如果要在JSP页面中使用存储在会话作用域中的bean对象，如下声明即可。

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

- 下面的页面在会话作用域内查找CustomerBean的一个实例并用表格的形式打印出它的属性值。

4.8.5 JavaBeans应用示例

程序4.23 displayCustomer.jsp

- 该页面首先在会话作用域内查找名为customer的bean实例，如果找到将输出bean实例的各属性值，如果找不到将创建一个bean实例并使用同名的请求参数为bean实例的各属性赋值，最后也输出各属性的值。

4.8.6　实现MVC模式的一般步骤

- 使用MVC设计模式开发Web应用程序可采用下面的一般步骤。
- 1. 定义JavaBeans表示数据
- 2. 使用Servlet处理请求
- 3. 填写JavaBeans对象数据
- 4. 结果的存储
- 5. 转发请求到JSP页面
- 6. 从JavaBeans对象中提取数据

1. 定义JavaBeans表示数据

- JavaBeans对象或使用POJO(Plain Old Java Object)对象一般只用来存放数据，JSP页面从JavaBeans对象或POJO中获得数据并显示给用户。

```
public class Customer implements Serializible{
    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。

5. 转发请求到JSP页面

- 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。
- 注意，RequestDispatcher的forward()和响应对象的sendRedirect()完全不同。

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" />
```

6. 从JavaBeans对象中提取数据

- 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。
- 强烈建议：在JSP2.0中应使用表达式语言（EL）输出数据

6. 从JavaBeans对象中提取数据

- 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规范提供了下面三个标准动作：
  - <jsp:useBean>
  - <jsp:setProperty>
  - <jsp:getProperty>

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。