

# JSP 服务器端页面技术

## 今日内容介绍

- ◆ 案例：显示商品信息

## 今日内容学习目标

- ◆ 简单阐述 JSP 的执行原理
- ◆ 会使用 page 执行处理 JSP 编码问题
- ◆ 会使用 page 执行导入 jsp 需要使用的类
- ◆ 列举 JSP 内置对象
- ◆ 列举 JSP 四大作用域
- ◆ 比较静态包含和动态包含

## 第1章 案例一：在 JSP 的页面中显示商品的信息.

### 1. 需求：

数据库中存放了很多商品信息，现在将商品的信息全部显示到页面。

## 2. 相关知识点：

### 1.2.1 JSP 概述

#### 1.2.1.1 什么是 JSP

JSP 全名是 Java Server Pages，它是建立在 Servlet 规范之上的动态网页开发技术。在 JSP 文件中，HTML 代码与 Java 代码共同存在，其中，HTML 代码用来实现网页中静态内容的显示，Java 代码用来实现网页中动态内容的显示。为了与传统 HTML 有所区别，JSP 文件的扩展名为.jsp。

JSP 技术所开发的 Web 应用程序是基于 Java 的，它可以用一种简捷而快速的方法从 Java 程序生成 Web 页面，其使用上具有如下几点特征：

- 跨平台：由于 JSP 是基于 Java 语言的，它可以使用 Java API，所以它也是跨平台的，可以应用于不同的系统中，如 Windows、Linux 等。当从一个平台移植到另一个平台时，JSP 和 JavaBean 的代码并不需要重新编译，这是因为 Java 的字节码是与平台无关的，这也应验了 Java 语言“一次编译，到处运行”的特点。
- 业务代码相分离：在使用 JSP 技术开发 Web 应用时，可以将界面的开发与应用程序的开发分离开。开发人员使用 HTML 来设计界面，使用 JSP 标签和脚本来动态生成页面上的内容。在服务器端，JSP 引擎（或容器，本书中指 Tomcat）负责解析 JSP 标签和脚本程序，生成所请求的内容，并将执行结果以 HTML 页面的形式返回到浏览器。
- 组件重用：JSP 中可以使用 JavaBean 编写业务组件，也就是使用一个 JavaBean 类封装业务处理代码或者作为一个数据存储模型，在 JSP 页面中，甚至在整个项目中，都可以重复使用这个 JavaBean，同时，JavaBean 也可以应用到其他 Java 应用程序中。
- 预编译：预编译就是在用户第一次通过浏览器访问 JSP 页面时，服务器将对 JSP 页面代码进行编译，并且仅执行一次编译。编译好的代码将被保存，在用户下一次访问时，会直接执行编译好的代码。这样不仅节约了服务器的 CPU 资源，还大大的提升了客户端的访问速度。

#### 1.2.1.2 编写第一个 JSP 文件

在 Eclipse 中，创建一个名称为 day17 的 Web 项目，然后右击 WebContent 目录 → 【new】→【Other】，在弹出的窗口中找到 JSP 文件，如图 1-1 所示。

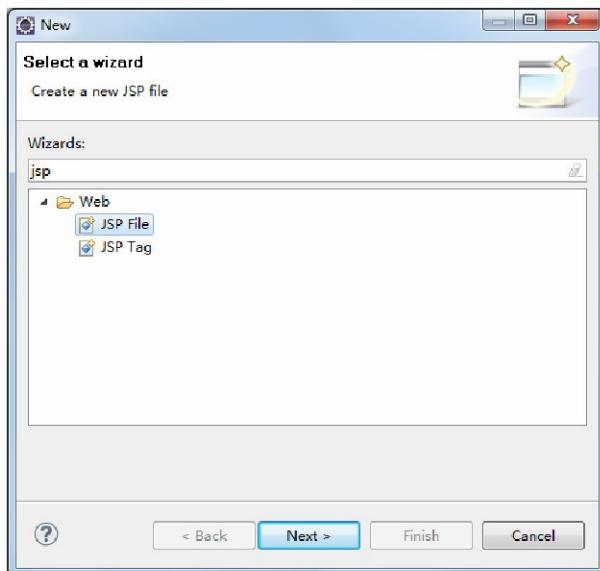


图1-1 创建 JSP 文件

在图 1-1 中，选择 JSP File 后，点击【Next】按钮，在新窗口的 File name 文本框中填写 JSP 文件名称 HelloWorld，如图 1-2 所示。

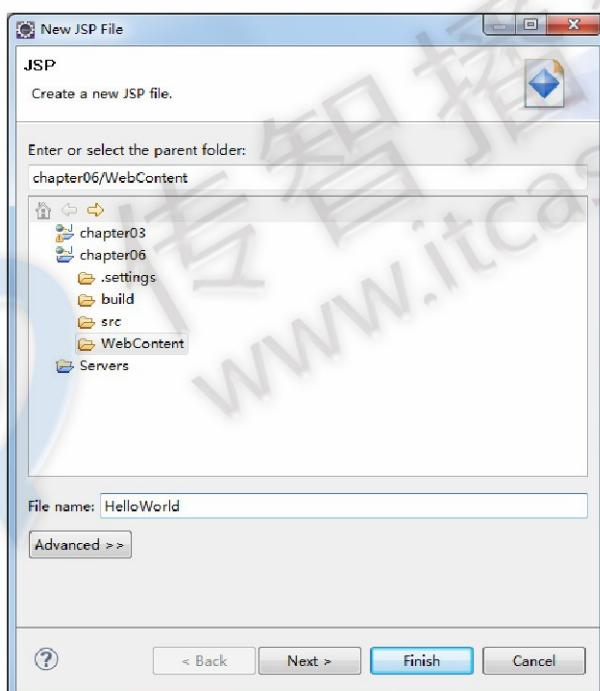


图1-2 命名文件

填写完图 1-2 中 JSP 文件名称后，点击【Next】按钮，进入选择模板窗口，此处采用默认设置，如图 1-3 所示。

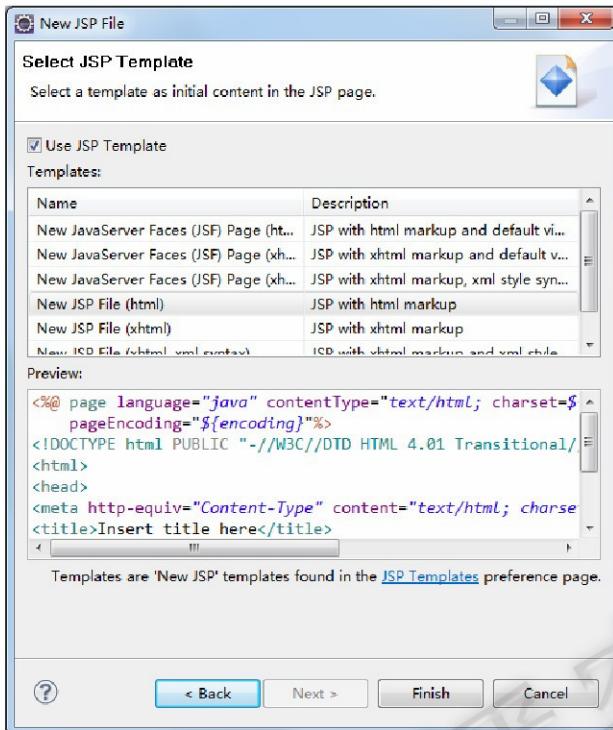


图1-3 选择模板窗口

点击图 1-3 中的【Finish】按钮后，第一个 JSP 文件就创建成功了。创建后的 JSP 文件代码如图 1-4 所示。

```

1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2     pageEncoding="UTF-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4             "http://www.w3.org/TR/html4/loose.dtd">
5<html>
6<head>
7 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8 <title>Insert title here</title>
9 </head>
10<body>
11
12</body>
13</html>

```

图1-4 HelloWorld.jsp

从图 1-4 中可以看出，新创建的 JSP 文件与传统的 HTML 文件几乎没有什么区别，唯一的区别是默认创建时，页面代码最上方多了一条 page 指令，并且该文件的后缀名是 jsp，而不是 html，关于 page 指令会在 1.3 节中详细讲解，此处了解即可。JSP 文件必须发布到 Web 容器中的某个 Web 应用中才能查看出效果。在 HelloWorld.jsp 的<body>元素内添加上文字“My First JSP”并保存后，将 day13 项目发布到 Tomcat 中并启动项目，在浏览器地址栏中输入地址“<http://localhost:8080/day13>HelloWorld.jsp>”，此时浏览器的显示效果如图 1-5 所示。

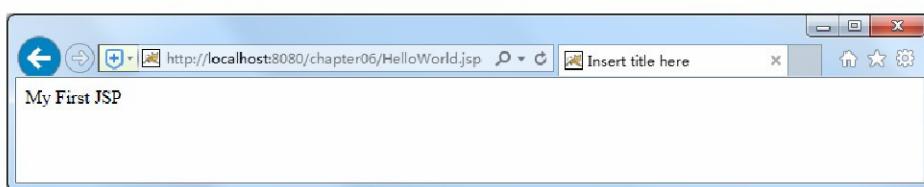


图1-5 HelloWorld.jsp 文件显示效果

从图 1-5 中可以看出，HelloWorld.jsp 中添加的内容已被显示出来，这说明了 HTML 中的元素可

以被 JSP 容器所解析。实际上，JSP 只是在原有的 HTML 文件中加入了一些具有 Java 特点的代码，这些代码具有其独有的特点，称为 JSP 的语法元素。

### 1.2.1.3 JSP 运行原理

JSP 的工作模式是请求/响应模式，客户端首先发出 HTTP 请求，JSP 程序收到请求后进行处理并返回处理结果。在一个 JSP 文件第一次被请求时，JSP 引擎（容器）把该 JSP 文件转换成为一个 Servlet，而这个引擎本身也是一个 Servlet。JSP 的运行过程如图 1-6 所示。

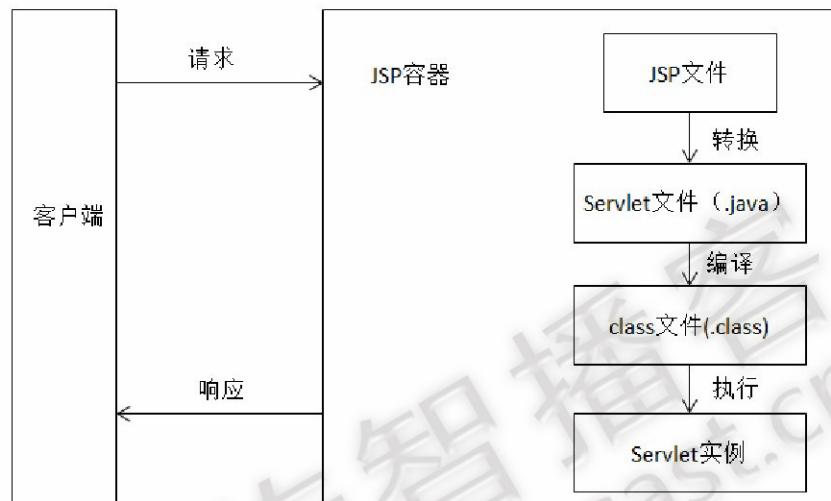


图 1-6 JSP 的运行原理

JSP 的运行过程具体如下：

- (1) 客户端发出请求，请求访问 JSP 文件。
- (2) JSP 容器先将 JSP 文件转换成一个 Java 源文件（Java Servlet 源程序），在转换过程中，如果发现 JSP 文件中存在任何语法错误，则中断转换过程，并向服务端和客户端返回出错信息。
- (3) 如果转换成功，则 JSP 容器将生成的 Java 源文件编译成相应的字节码文件\*.class。该 class 文件就是一个 Servlet，Servlet 容器会像处理其他 Servlet 一样来处理它。

为了使同学们更容易理解 JSP 的运行原理，接下来简单介绍分析一下 JSP 所生成的 Servlet 代码。

以 HelloWorld.jsp 为例，当用户第一次访问 HelloWorld.jsp 页面时，该页面会先被 JSP 容器转换为一个名称为 HelloWorld\_jsp.java 的源文件，然后将源文件编译为一个名称为 HelloWorld\_jsp.class 字节码文件。如果项目发布在 Tomcat 的 webapps 目录中，源文件和.class 文件可以在“Tomcat 安装目录/work/Catalina/localhost/项目名/org/apache/jsp”下找到，如图 1-7 所示。

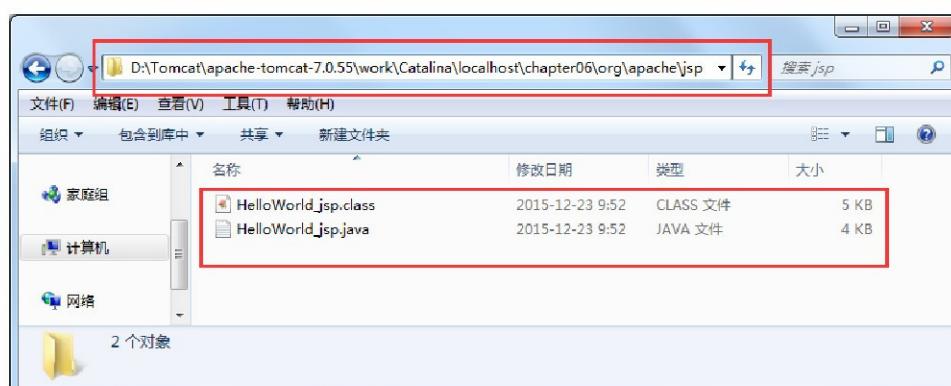


图1-7 JSP 文件编译后的文件

在图 1-7 中，地址栏中的路径多出了 org\apache\jsp，这是由于 JSP 文件转换成类文件时会带有包名，该包名为 org.apache.jsp。从图中还可以看出，HelloWorld.jsp 已被转换为源文件和.class 文件。打开 HelloWorld\_jsp.java 文件，可查看转换后的源代码，其主要代码如下所示。（以下代码可以快速浏览，确定父类和方法名即可）

```
package org.apache.jsp;
...
public final class HelloWorld_jsp
    extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent {
...
    public void _jspInit() {
        _el_expressionfactory = _jspxFactory.getJspApplicationContext(
            getServletConfig().getServletContext()).getExpressionFactory();
        _jsp_instancemanager = org.apache.jasper.runtime
            .InstanceManagerFactory.getInstanceManager(getServletConfig());
    }

    public void _jspDestroy() {
    }

    public void _jspService(final javax.servlet.http.HttpServletRequest request,
        final javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {
        final javax.servlet.jsp.PageContext pageContext;
        javax.servlet.http.HttpSession session = null;
        final javax.servlet.ServletContext application;
        final javax.servlet.ServletConfig config;
        javax.servlet.jsp.JspWriter out = null;
        final java.lang.Object page = this;
        javax.servlet.jsp.JspWriter _jspx_out = null;
        javax.servlet.jsp.PageContext _jspx_page_context = null;
        try {
            response.setContentType("text/html; charset=UTF-8");
            pageContext = _jspxFactory.getPageContext(this, request, response,
                null, true, 8192, true);
            _jspx_page_context = pageContext;
            application = pageContext.getServletContext();
            config = pageContext.getServletConfig();
            session = pageContext.getSession();
            out = pageContext.getOut();
            _jspx_out = out;
            out.write("\r\n");
            out.write("<!DOCTYPE html PUBLIC \"-//W3C//DTD HTML 4.01 Transitional//EN\"");

```

```

\"http://www.w3.org/TR/html4/loose.dtd\">>\r\n");
    out.write("<html>\r\n");
    out.write("<head>\r\n");
    out.write("<meta http-equiv=\"Content-Type\" content=\"text/html;
charset=UTF-8\>\r\n");
    out.write("<title>Insert title here</title>\r\n");
    out.write("</head>\r\n");
    out.write("<body>\r\n");
    out.write("    My First JSP\r\n");
    out.write("</body>\r\n");
    out.write("</html>");
} catch (java.lang.Throwable t) {
    if (!(t instanceof javax.servlet.jsp.SkipPageException)){
        out = _jspx_out;
        if (out != null && out.getBufferSize() != 0)
            try {
                if (response.isCommitted()) {
                    out.flush();
                } else {
                    out.clearBuffer();
                }
            } catch (java.io.IOException e) {}
        if (_jspx_page_context != null) _jspx_page_context.handlePageException(t);
        else throw new ServletException(t);
    }
} finally {
    _jspxFactory.releasePageContext(_jspx_page_context);
}
}
}

```

从上面的代码可以看出，HelloWorld.jsp 文件转换后的源文件没有实现 Servlet 接口，但继承了 org.apache.jasper.runtime.HttpJspBase 类。在 Tomcat 源文件中查看 HttpJspBase 类的源代码，具体如下所示：（以下代码可以快速阅读，确定父类即可）

```

package org.apache.jasper.runtime;
...
public abstract class HttpJspBase extends HttpServlet implements HttpJspPage {
    private static final long serialVersionUID = 1L;
    protected HttpJspBase() {
    }
    @Override
    public final void init(ServletConfig config)
        throws ServletException
    {
        super.init(config);
    }
}

```

```
    jspInit();
    _jspInit();
}

@Override
public String getServletInfo() {
    return Localizer.getMessage("jsp.engine.info");
}

@Override
public final void destroy() {
    jspDestroy();
    _jspDestroy();
}

@Override
public final void service(HttpServletRequest request,
                           HttpServletResponse response)
    throws ServletException, IOException
{
    _jspService(request, response);
}

@Override
public void jspInit() {
}

public void _jspInit() {
}

@Override
public void jspDestroy() {
}

protected void _jspDestroy() {
}

@Override
public abstract void _jspService(HttpServletRequest request,
                                 HttpServletResponse response)
    throws ServletException, IOException;
}
```

从 HttpJspBase 源代码中可以看出，HttpJspBase 类是 HttpServlet 的一个子类，由此可见，HelloWorld\_jsp 类就是一个 Servlet。结论：JSP 就是 Servlet。

## 1.2.2 JSP 基本语法

在 JSP 文件中可以嵌套很多内容，例如，JSP 的脚本元素和注释等，这些内容的编写都需要遵循一定的语法规则，接下来，本节将针对这些语法进行详细的讲解。

### 1.2.2.1 JSP 脚本元素

JSP 脚本元素是指嵌套在<%和%>之中的一条或多条 Java 程序代码。通过 JSP 脚本元素可以将 Java 代码嵌入 HTML 页面中，所有可执行的 Java 代码，都可以通过 JSP 脚本来执行。

JSP 脚本元素主要包含如下三种类型：

- JSP Scriptlets
- JSP 声明语句
- JSP 表达式

#### 1. JSP Scriptlets

JSP Scriptlets 是一段代码段。当需要使用 Java 实现一些复杂操作或控制时，可以使用它。JSP Scriptlets 的语法格式如下所示：

```
<% java 代码（变量、方法、语句等）%>
```

在 JSP Scriptlets 中声明的变量是 JSP 页面的局部变量，调用 JSP Scriptlets 时，会为局部变量分配内存空间，调用结束后，释放局部变量占有的内存空间。

#### 2. JSP 声明语句

JSP 的声明语句用于声明变量和方法，它以“<%!”开始，以“%>”结束，其语法格式如下所示：

```
<%!
    定义的变量或方法等
%>
```

在上述语法格式中，被声明的 Java 代码将被编译到 Servlet 的 \_jspService() 方法之外，即在 JSP 声明语句中定义的都是成员方法、成员变量、静态方法、静态变量、静态代码块等。在 JSP 声明语句中声明的方法在整个 JSP 页面内有效。

在一个 JSP 页面中可以有多个 JSP 声明语句，单个声明中的 Java 语句可以是不完整的，但是多个声明组合后的结果必须是完整的 Java 语句。接下来，通过一个案例来演示 JSP Scriptlets 和声明语句的使用。

在 day12 项目的 WebContent 目录下创建一个名称为 example01.jsp 的文件，在该文件中编写声明语句，如文件 1-1 所示。

文件 1-1 example01.jsp

```

1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2     pageEncoding="UTF-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4     "http://www.w3.org/TR/html4/loose.dtd">
5 <html>
6 <head>
7 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8 <title>JSP 声明语句</title>
9 </head>
10 <%!
11     int a = 1, b = 2; // 定义两个变量 a,b
12 %>
13 <%!
14     public String print() { // 定义 print 方法

```

```

15         String str = "itcast"; //方法内定义的变量 str
16         return str;
17     }
18 %>
19 <body>
20     <%
21         out.println(a + b); //输出两个变量的和
22     %>
23     <br>
24     <%
25         out.println(print());//调用 print()方法，输出其返回值
26     %>
27 </body>
28 </html>

```

在文件 1-1 中，首先使用`<%!`和`%>`定义了两个变量 a、b，以及 print()方法，然后使用了`<%`和`%>`输出了两个常量的和，以及 print()方法中的返回信息。启动项目后，在浏览器地址栏中输入地址“<http://localhost:8080/day17/example01.jsp>”访问 example01.jsp 文件，显示效果如图 1-8 所示。



图1-8 example01.jsp 的执行结果

从图 1-8 中可以看到，浏览器中已经显示出了相应的结果。

需要注意的是，`<%!`和`%>`里面定义的变量是成员变量，方法是全局的方法，此处只是声明，也就是定义，变量或方法都没有被调用。`<%`和`%>`里面定义的是局部变量，不能定义方法（Java 方法中不能再嵌套定义方法），代码块操作可以将结果输出到浏览器。总之，`<%!`和`%>`是用来定义成员变量属性和方法的，`<%`和`%>`主要是用来输出内容的，因此如果涉及到了成员变量的操作，那么就应该使用`<%!`和`%>`，而如果是涉及到了输出内容的时候，就使用`<%`和`%>`。

### 3. JSP 表达式

JSP 表达式（expression）用于将程序数据输出到客户端，它将要输出的变量或者表达式直接封装在以“`<%=`”开头和以“`%>`”结尾的标记中，其基本的语法格式如下所示：

```
<%= expression %>
```

在上述语法格式中，JSP 表达式中的将“`expression`”表达式结果输出到浏览器。例如，对 example01.jsp 文件进行修改，将`<body>`内的脚本元素修改为表达式，具体如下。

```
<%=a+b %><br>
<%=print() %>
```

在浏览器中再次访问 example01.jsp 页面，同样可以正确输出如图 1-8 中的显示结果。需要注意的是：

- “`<%=`” 和 “`%>`” 标记之间插入的是表达式，不能插入语句。
- “`<%=`” 是一个完整的符号，“`<%`” 和 “`=`” 之间不能有空格。
- JSP 表达式中的变量或表达式后面不能有分号（`;`）。

### 1.2.2.2 JSP 注释

同其他各种编程语言一样，JSP 也有自己的注释方式，其基本语法格式如下：

```
<%-- 注释信息 --%>
```

需要注意的是，Tomcat 在将 JSP 页面编译成 Servlet 程序时，会忽略 JSP 页面中被注释的内容，不会将注释信息发送到客户端。接下来，通过一个案例来演示 JSP 注释的使用。

在 day12 项目的 WebContent 目录下创建一个名称为 example02 的 JSP 页面，如文件 1-2 所示。

文件 1-2 example02.jsp

```

1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2      pageEncoding="UTF-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
4           "http://www.w3.org/TR/html4/loose.dtd">
5 <html>
6 <head>
7 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8 <title>JSP 注释</title>
9 </head>
10 <body>
11    <!-- 这个是 HTML 注释 -->
12    <%-- 这个是 JSP 注释 --%>
13 </body>
14 </html>

```

在上述页面代码中，包含 HTML 注释和 JSP 两种注释方式。启动 Tomcat 服务器，在浏览器的地址栏中输入地址“<http://localhost:8080/day17/example02.jsp>”访问 example02.jsp 页面，此时，可以看到 example02.jsp 页面什么都不显示，接下来在打开的页面中点击鼠标右键，在弹出菜单中选择【查看源文件】选项，结果如图 1-9 所示。



```

1
2 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
3           "http://www.w3.org/TR/html4/loose.dtd">
4 <html>
5 <head>
6 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7 <title>JSP 注释</title>
8 </head>
9 <body>
10   <!-- 这个是HTML注释 -->
11   <%-- 这个是 JSP 注释 --%>
12 </body>
13 </html>

```

图 1-9 example02.jsp 的源代码

从图 1-9 中可以看出，JSP 的注释信息没有显示出来，而只显示出了 HTML 注释。这是因为在 Tomcat 编译 JSP 文件时，会将 HTML 注释当成普通文本发送到客户端，而 JSP 页面中格式为“`<%-- 注释信息 --%>`”的内容则会被忽略，不会发送到客户端。

- JSP 注释、Java 注释、HTML 注释对比：

```

<%-- jsp 注释--%>
<% //java 注释 %>
<!-- html 注释-->

```

	jsp 源码	java 源码	html 源码
jsp 注释	有	无	无
java 注释	有	有	无
html 注释	有	有	有

### 1.2.3 JSP 指令

为了设置 JSP 页面中的一些信息，Sun 公司提供了 JSP 指令。JSP 2.0 中共定义了 page、include 和 taglib 三种指令，每种指令都定义了各自的属性。接下来，本节将针对 page 和 include 指令进行详细的讲解。

#### 1.2.3.1 page 指令

在 JSP 页面中，经常需要对页面的某些特性进行描述，例如，页面的编码方式，JSP 页面采用的语言等，这时，可以通过 page 指令来实现。page 指令的具体语法格式如下所示：

```
<%@ page 属性名 1= "属性值 1" 属性名 2= "属性值 2" ...%>
```

在上面的语法格式中，page 用于声明指令名称，属性用来指定 JSP 页面的某些特性。page 指令提供了一系列与 JSP 页面相关的属性，如表 1-1 所示。

表1-1 page 指令的常用属性

属性名称	取值 or 范围	描述
pageEncoding	当前页面	指定页面编码格式
contentType	有效的文档类型	客户端浏览器根据该属性判断文档类型，例如： HTML 格式为 text/html 纯文本格式为 text/plain JPG 图像为 image/jpeg GIF 图像为 image/gif Word 文档为 application/msword
buffer	8kb	jsp 缓存大小
autoFlush	true / false	是否自动刷新
errorCode	某个 JSP 页面的相对路径	指定一个错误页面，如果该 JSP 程序抛出一个未捕捉的异常，则转到 errorCode 指定的页面。errorCode 指定页面的 isErrorPage 属性为 true，且内置的 exception 对象为未捕捉的异常
isErrorPage	true / false	指定该页面是否为错误处理页面，如果为 true，则该 JSP 内置有一个 Exception 对象的 exception，可直接使用。默认情况下，isErrorPage 的值为 false
import	任何包名、类名	指定在 JSP 页面翻译成的 Servlet 源文件中导入的包或类。import 是唯一可以声明多次的 page 指令属性。一个 import 属性可以引用多个类，中间用英文逗号隔开。
language	java	指明解释该 JSP 文件时采用的语言，默认为 Java
session	true、false	指明该 JSP 内是否内置 Session 对象，如果为 true，则说明内置 Session 对象，可以直接使用，否则没有内置 Session 对象。默认情况下，session 属性的值为 true。需要注意的是，JSP 引擎自动导入以下 4 个包： java.lang.* javax.servlet.* javax.servlet.jsp.* javax.servlet.http.*

表 1-1 中列举了 page 指令的常见属性，其中，除了 import 属性外，其他的属性都只能出现一次，否则会编译失败。需要注意的是，page 指令的属性名称都是区分大小写的。

下面列举两个使用 page 指令的示例：

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
           pageEncoding="UTF-8"%>
<%@ page import="java.awt.*" %>
<%@ page import="java.util.*","java.awt.*"%>
```

上面代码中使用了 page 指令的 language、contentType、pageEncoding 和 import 属性。

需要注意的是，page 指令对整个页面都有效，而与其书写的位置无关，但是习惯上把 page 指令写在 JSP 页面的最前面。

### 1.2.3.2 include 指令

在实际开发时，有时需要在 JSP 页面静态包含一个文件，例如 HTML 文件，文本文件等，这时，可以通过 include 指令来实现，include 指令的具体语法格式如下所示：

```
<%@ include file="被包含的文件地址"%>
```

include 指令只有一个 file 属性，该属性用来指定插入到 JSP 页面目标位置的文件资源。

为了使读者更好地理解 include 指令的使用，接下来，通过一个案例来学习 include 指令的具体用法。

在 day12 项目的 WebContent 目录下创建两个 JSP 页面文件 date.jsp 和 include.jsp，在 include.jsp 文件中使用 include 指令将 date.jsp 文件包含其中，具体如文件 1-3 和文件 1-4 所示。

文件1-3 date.jsp

```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"%>
2 <html>
3 <head><title>Insert title here</title>
4 </head>
5 <body>
6 <% out.println(new java.util.Date().toLocaleString());%>
7 </body>
8 </html>
```

文件1-4 include.jsp

```
9 <%@ page language="java" contentType="text/html; charset=UTF-8"%>
10 <html>
11 <head>
12 <title>欢迎你</title>
13 </head>
14 <body>
15     欢迎你，现在的时间是：
16     <%@ include file="/date.jsp"%>
17 </body>
18 </html>
```

启动 tomcat 服务器，在浏览器中访问地址“<http://localhost:8080/day17/include.jsp>”，浏览器的显示结果如图 1-10 所示。

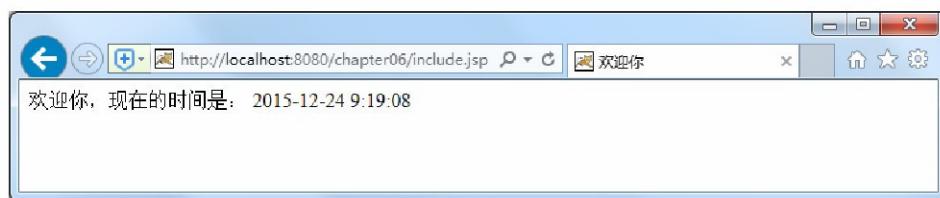


图1-10 运行结果

从图 1-10 中可以看出，date.jsp 文件中用于输出当前日期的语句已显示出来，这说明 include 指令成功地将 date.jsp 文件中的代码合并到了 include.jsp 文件中。

关于 include 指令的具体应用，有很多问题需要注意，接下来，将这些问题进行列举，具体如下：

(1) 被引入的文件必须遵循 JSP 语法，其中的内容可以包含静态 HTML、JSP 脚本元素和 JSP 指令等普通 JSP 页面所具有的一切内容。

(2) 除了指令元素之外，被引入的文件中的其他元素都被转换成相应的 Java 源代码，然后插入进当前 JSP 页面所翻译成的 Servlet 源文件中，插入位置与 include 指令在当前 JSP 页面中的位置保持一致。

### 1.2.3.3 taglib 指令

用于页面中引入标签库的，这个指令会在后面介绍 JSTL 的时候讲解。

### 1.2.4 JSP 内置对象

#### 1.2.4.1 内置对象的概述

在 JSP 页面中，有一些对象需要频繁使用，如果每次都重新创建这些对象则会非常麻烦。为了简化 Web 应用程序的开发，JSP2.0 规范中提供了 9 个隐式（内置）对象，它们是 JSP 默认创建的，可以直接在 JSP 页面中使用。这 9 个隐式对象的名称、类型和描述如表 1-2 所示。

表1-2 JSP 隐式对象

名称	类型	描述
out	javax.servlet.jsp.JspWriter	用于页面输出
request	javax.servlet.http.HttpServletRequest	得到用户请求信息，
response	javax.servlet.http.HttpServletResponse	服务器向客户端的回应信息
config	javax.servlet.ServletConfig	服务器配置，可以取得初始化参数
session	javax.servlet.http.HttpSession	用来保存用户的信息
application	javax.servlet.ServletContext	所有用户的共享信息
page	java.lang.Object	指当前页面转换后的 Servlet 类的实例
pageContext	javax.servlet.jsp.PageContext	JSP 的页面容器
exception	java.lang.Throwable	表示 JSP 页面所发生的异常，在错误页中才起作用

在表 1-2 中，列举了 JSP 的 9 个隐式对象及它们各自对应的类型。其中，由于 request、response、config、session 和 application 所属的类及其用法在前面的章节都已经讲解过，而 page 对象在 JSP 页面中很少被用到。因此，在下面几个小节中，将针对 out 和 pageContext 对象进行详细的讲解。

### 1.2.4.2 out 对象

在 JSP 页面中，经常需要向客户端发送文本内容，这时，可以使用 out 对象来实现。out 对象是 javax.servlet.jsp.JspWriter 类的实例对象，它的作用与 `ServletResponse.getWriter()` 方法返回的 `PrintWriter` 对象非常相似，都是用来向客户端发送文本形式的实体内容。不同的是，out 对象的类型为 `JspWriter`，它相当于一种带缓存功能的 `PrintWriter`。接下来，通过一张图来描述 JSP 页面的 out 对象与 Servlet 引擎提供的缓冲区之间的工作关系，具体如图 1-11 所示。

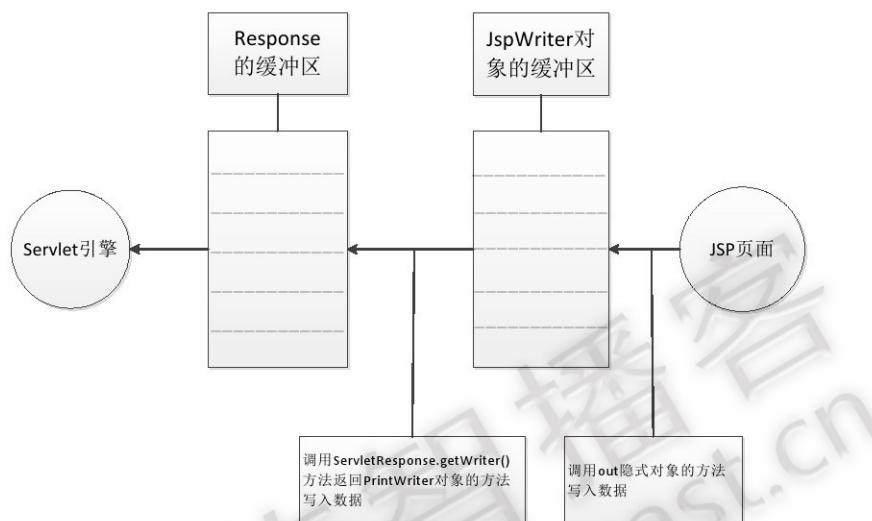


图1-11 out 对象与 Servlet 引擎的关系

从图 1-11 可以看出，在 JSP 页面中，通过 out 隐式对象写入数据相当于将数据插入到 `JspWriter` 对象的缓冲区中，只有调用了 `ServletResponse.getWriter()` 方法，缓冲区中的数据才能真正写入到 Servlet 引擎所提供的缓冲区中。为了验证上述说法是否正确，接下来，通过一个具体的案例来演示 out 对象的使用。

在 day12 项目的 WebContent 目录下创建一个名称为 out 的 JSP 页面，如文件 1-5 所示。

文件1-5 out.jsp

```

19 <%@ page language="java" contentType="text/html; charset=UTF-8"%>
20 <html>
21 <head>
22 <title>Insert title here</title>
23 </head>
24 <body>
25 <%
26     out.println("first line<br>");
27     response.getWriter().println("second line<br>");
28 %>
29 </body>
30 </html>

```

启动 Tomcat 服务器，在浏览器地址栏中访问 “<http://localhost:8080/day12/out.jsp>”，浏览器的显示结果如图 1-12 所示。



图1-12 运行结果

从图 1-12 中可以看出，尽管 `out.println` 语句位于 `response.getWriter().println` 语句之前，但它的输出内容却在后面。由此可以说明，`out` 对象通过 `print` 语句写入数据后，直到整个 JSP 页面结束，`out` 对象中输入缓冲区的数据（即：first line）才真正写入到 Servlet 引擎提供的缓冲区中，而 `response.getWriter().println` 语句则是直接把内容（即：second line）写入 Servlet 引擎提供的缓冲区中，Servlet 引擎按照缓冲区中的数据存放顺序输出内容。

### 1.2.4.3 pageContext 对象

在 JSP 页面中，使用 `pageContext` 对象可以获取 JSP 的其他 8 个隐式对象。`pageContext` 对象是 `javax.servlet.jsp.PageContext` 类的实例对象，它代表当前 JSP 页面的运行环境，并提供了一系列用于获取其他隐式对象的方法。`pageContext` 对象获取隐式对象的方法如表 1-3 所示。

表1-3 pageContext 获取隐式对象的方法

方法名	功能描述
<code>JspWriter getOut()</code>	用于获取 <code>out</code> 隐式对象
<code>Object getPage()</code>	用于获取 <code>page</code> 隐式对象
<code>ServletRequest getRequest()</code>	用于获取 <code>request</code> 隐式对象
<code>ServletResponse getResponse()</code>	用于获取 <code>response</code> 隐式对象
<code>HttpSession getSession()</code>	用于获取 <code>session</code> 隐式对象
<code>Exception getException()</code>	用于获取 <code>exception</code> 隐式对象
<code>ServletConfig getServletConfig()</code>	用于获取 <code>config</code> 隐式对象
<code>ServletContext getServletContext()</code>	用于获取 <code>application</code> 隐式对象

表 1-3 中列举了 `pageContext` 获取其他隐式对象的方法，这样，当传递一个 `pageContext` 对象后，就可以通过这些方法轻松地获取到其他 8 个隐式对象了。

`pageContext` 对象不仅提供了获取隐式对象的方法，还提供了存储数据的功能。`pageContext` 对象存储数据是通过操作属性来实现的，表 1-4 列举了 `pageContext` 操作属性的一系列方法，具体如下：

表1-4 pageContext 操作属性的相关方法

方法名称	功能描述
<code>void setAttribute(String name, Object value, int scope)</code>	用于设置 <code>pageContext</code> 对象的属性
<code>Object getAttribute(String name, int scope)</code>	用于获取 <code>pageContext</code> 对象的属性
<code>void removeAttribute(String name, int scope)</code>	删除指定范围内名称为 <code>name</code> 的属性
<code>void removeAttribute(String name)</code>	删除所有范围内名称为 <code>name</code> 的属性
<code>Object findAttribute(String name)</code>	从 4 个域对象中查找名称为 <code>name</code> 的属性

表 1-4 列举了 `pageContext` 对象操作属性的相关方法，其中，参数 `name` 指定的是属性名称，参数 `scope` 指定的是属性的作用范围。`pageContext` 对象的作用范围有 4 个值，具体如下：

- `PageContext.PAGE_SCOPE`: 表示页面范围
- `PageContext.REQUEST_SCOPE`: 表示请求范围

- PageContext.SESSION\_SCOPE：表示会话范围
- PageContext.APPLICATION\_SCOPE：表示 Web 应用程序范围

需要注意的是，当使用 `findAttribute()`方法查找名称为 name 的属性时，会按照 page、request、session 和 application 的顺序依次进行查找，如果找到，则返回属性的名称，否则返回 null。接下来，通过一个案例来演示 `pageContext` 对象的使用。

在 day12 项目的 WebContent 目录下创建一个名称为 `pageContext.jsp` 的页面，编辑后如文件 1-7 所示。

文件1-6 pageContext.jsp

```

31 <%@ page language="java" contentType="text/html; charset=UTF-8"%>
32 <html>
33 <head>
34 <title>pageContext</title>
35 </head>
36 <body>
37     <%
38         //获取 request 对象
39         HttpServletRequest req = (HttpServletRequest) pageContext
40             .getRequest();
41         //设置 page 范围内属性
42         pageContext.setAttribute("str", "Java", pageContext.PAGE_SCOPE);
43         //设置 request 范围内属性
44         req.setAttribute("str", "Java Web");
45         //获得的 page 范围属性
46         String str1 = (String)pageContext.getAttribute("str",
47                         pageContext.PAGE_SCOPE);
48         //获得的 request 范围属性
49         String str2 = (String)pageContext.getAttribute("str",
50                         pageContext.REQUEST_SCOPE);
51     %>
52     <%="page 范围: "+str1 %><br>
53     <%="request 范围: "+str2 %><br>
54 </body>
55 </html>
```

在上述代码中，首先使用 `pageContext` 获取了 `request` 对象，并设置 `page` 范围内属性；然后使用获取的 `request` 对象设置了 `request` 范围内属性，接下来使用 `pageContext` 对象获得 `page` 和 `request` 范围内的相应属性，最后使用 JSP 表达式输出数据。

启动 Tomcat 服务器，在浏览器的地址栏中输入地址“`http://localhost:8080/day17/pageContext.jsp`”访问 `pageContext.jsp` 页面，浏览器显示的结果如图 1-13 所示。



图1-13 运行结果

从图 1-13 的显示结果可以看出，通过 pageContext 对象可以获取到 request 对象，并且还可以获取不同范围内的属性。

### 1.2.5 JSP 的四个域范围：

PageContext 常量名	描述	作用域名称	域对象类型
PageScope	当前页面中有效	pageContext	PageContext
RequestScope	一次请求范围	request	HttpServletRequest
SessionScope	一次会话范围	session	HttpSession
ApplicationScope	应用范围	application	ServletContext

- **page:** 表示当前页，通常没用。jsp 标签底层使用。
- **request:** 表示一次请求。通常一次请求就一个页面，但如果使用请求转发，可以涉及多个页面。
- **session:** 表示一次会话。可以在多次请求之间共享数据。
- **application:** 表示一个 web 应用(项目)。可以整个 web 项目共享，多次会话共享数据。

### 1.2.6 JSP 动作元素(了解)

JSP 动作元素用来控制 JSP 的行为，执行一些常用的 JSP 页面动作。通过动作元素可以实现使用多行 Java 代码能够实现的效果，如包含页面文件，实现请求转发等。

#### 1.2.6.1 <jsp:include>动作元素

在 JSP 页面中，为了把其他资源的输出内容插入到当前 JSP 页面的输出内容中，JSP 技术提供了<jsp:include>动作元素，<jsp:include>动作元素的具体语法格式如下所示：

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

在上述语法格式中，page 属性用于指定被引入资源的相对路径，flush 属性用于指定是否将当前页面的输出内容刷新到客户端，默认情况下，flush 属性的值为 false。

<jsp:include>包含的原理是将被包含的页面编译处理后将结果包含在页面中。当浏览器第一次请求一个使用<jsp:include>包含其他页面的页面时，Web 容器首先会编译被包含的页面，然后将编译处理后的返回结果包含在页面中，之后编译包含页面，最后将两个页面组合的结果回应给浏览器。为了使读者更好地理解<jsp:include>动作元素，接下来，通过一个案例来演示<jsp:include>动作元素的使用，具体如下：

(1) 在 day12 项目的 WebContent 目录下编写两个 JSP 文件，分别是 included.jsp 和 dynamicInclude.jsp。其中 dynamicInclude.jsp 页面用于引入 included.jsp 页面。included.jsp 作为被引入的文件，让它暂停 5 秒钟后才输出内容，这样，可以方便测试<jsp:include>标签的 flush 属性。included.jsp 的具体代码如文件 1-10 所示，dynamicInclude.jsp 具体代码如文件 1-11 所示。

文件 1-7 included.jsp

```
56 <%@ page language="java" contentType="text/html; charset=UTF-8"
57     pageEncoding="UTF-8"%>
58 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
```

```

59                     "http://www.w3.org/TR/html4/loose.dtd">
60 <html>
61 <head>
62 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
63 <title>include</title>
64 </head>
65 <body>
66     <%Thread.sleep(5000);%>
67     included.jsp 内的中文<br>
68 </body>
69 </html>

```

文件1-8 dynamicInclude.jsp

```

70 <%@ page language="java" contentType="text/html; charset=UTF-8"
71     pageEncoding="UTF-8"%>
72 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
73                     "http://www.w3.org/TR/html4/loose.dtd">
74 <html>
75 <head>
76 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
77 <title>dynamicInclude page</title>
78 </head>
79 <body>
80     dynamicInclude.jsp 内的中文
81     <br>
82     <jsp:include page="included.jsp" flush="true" />
83
84 </body>
85 </html>

```

(2) 启动 Tomcat 服务器，访问地址“<http://localhost:8080/day12/dynamicInclude.jsp>”后，发现浏览器首先会显示 dynamicInclude.jsp 页面中的输出内容，等待 5 秒后，才会显示 included.jsp 页面的输出内容。说明被引用的资源 included.jsp 在当前 JSP 页面输出内容后才被调用。其最后显示结果如图 1-14 所示。



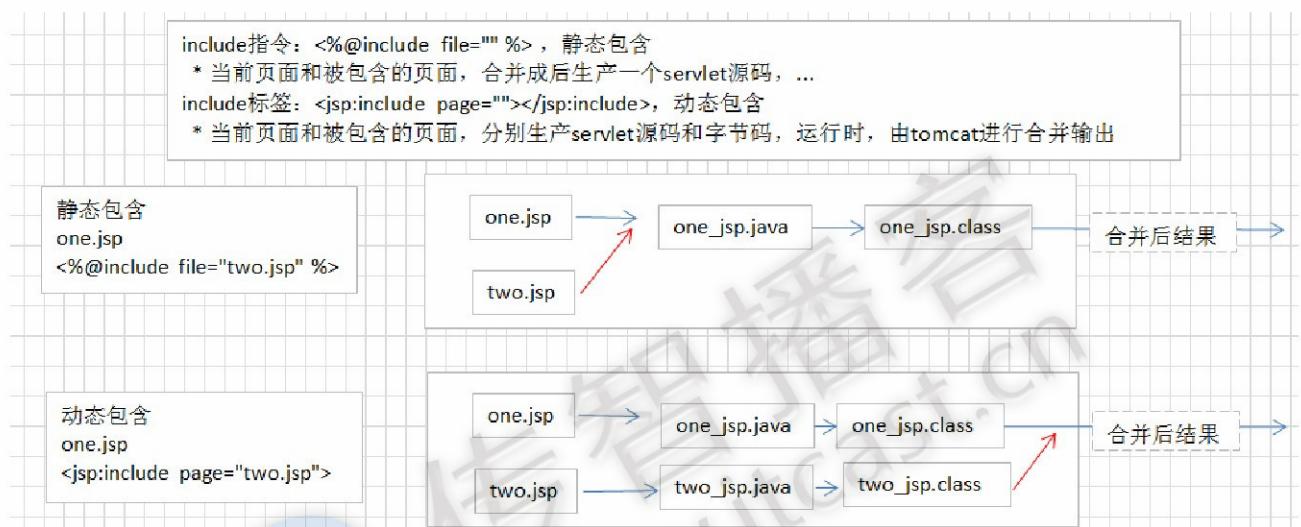
图1-14 dynamicInclude.jsp

(3) 修改 dynamicInclude.jsp 文件，将<jsp:include>动作元素中的 flush 属性设置为 false，刷新浏览器，再次访问地址“<http://localhost:8080/day12/dynamicInclude.jsp>”，这时，浏览器等待 5 秒后，将 dynamicInclude.jsp 和 included.jsp 页面的输出内容同时显示了出来。由此可见，Tomcat 调用被引入的资源 included.jsp 时，并没有将当前 JSP 页面中已输出的内容刷新到客户端。

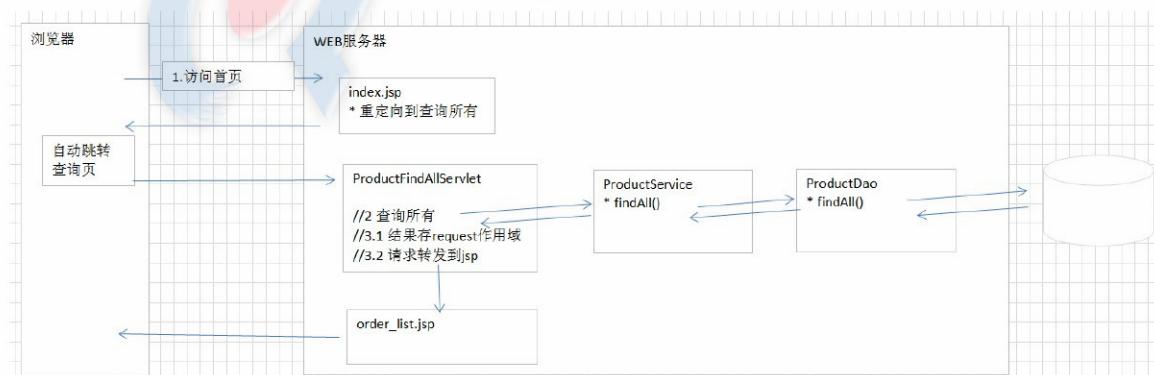
需要注意的是，虽然 include 指令和<jsp:include>标签都能够包含一个文件，但它们之间有很大的区别，具体如下：

- <jsp:include>标签中要引入的资源和当前 JSP 页面是两个彼此独立的执行实体，即被动态引入的资源必须能够被 Web 容器独立执行。而 include 指令只能引入遵循 JSP 格式的文件，被引入文件与当前 JSP 文件需要共同合并才能翻译成一个 Servlet 源文件。
- <jsp:include>标签中引入的资源是在运行时才包含的，而且只包含运行结果。而 include 指令引入的资源是在编译时期包含的，包含的是源代码。
- <jsp:include>标签运行原理与 RequestDispatcher.include()方法类似，即被包含的页面不能改变响应状态码或者设置响应头，而 include 指令没有这方面的限制。

### 1.2.6.2 动态包含和静态包含的区别：



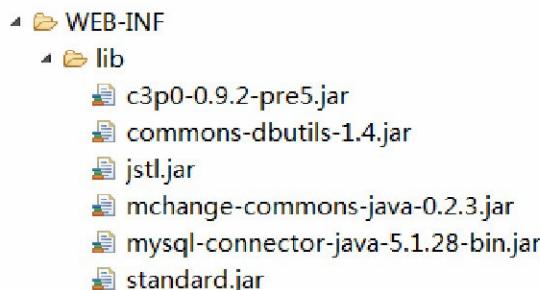
## 3. 案例分析



综合案例需要点击“导航条/分类”进行列表信息的展示，本案为了巩固重定向，在首页编写重定向代码直接跳转到 ProductFindAllServlet 进行查询，在 Servlet 中调用 ProductService 获得所有的查询结果，在 service 中调用 ProductDao 使用 DBUtils 进行查询，当查询完数据，在 JSP 页面进行数据显示。

## 4. 代码实现：

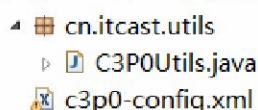
- 步骤 1：创建项目，并导入 jar 包



- 步骤 2：初始化数据库

```
#创建数据库和表：
create database day17_db;
#使用
use day17_db;
#创建商品表
CREATE TABLE `product` (
    `pid` varchar(32) primary key,
    `pname` varchar(50) DEFAULT NULL,          #商品名称
    `market_price` double DEFAULT NULL,         #商场价
    `shop_price` double DEFAULT NULL,           #商城价
    `pimage` varchar(200) DEFAULT NULL,          #商品图片路径
    `pdate` date DEFAULT NULL,                  #上架时间
    `is_hot` int(11) DEFAULT NULL,               #是否热门: 0=不热门, 1=热门
    `pdesc` varchar(255) DEFAULT NULL,           #商品描述
    `pflag` int(11) DEFAULT 0,                  #商品标记: 0=未下架(默认值), 1=已经下架
    `cid` varchar(32) DEFAULT NULL              #分类 id
) ;
```

- 步骤 3：导入工具类和 c3p0 配置文件，并修改数据库



```
22      <!-- 命名的配置 -->
23      <named-config name="itheima">
24          <property name="driverClass">com.mysql.jdbc.Driver</property>
25          <property name="jdbcUrl">jdbc:mysql://127.0.0.1:3306/day17_db</property>
26          <property name="user">root</property>
27          <property name="password">1234</property>
```

- 步骤 4: 编写 JavaBean, 并提供相应构造方法, 以及 `toString()`方法

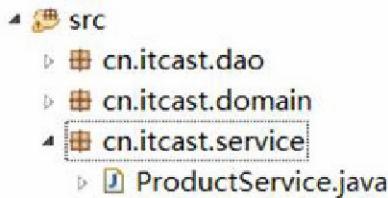
```
public class Product {  
    private String pid;  
    private String pname;  
    private Double market_price;  
  
    private Double shop_price;  
    private String pimage;  
    private Date pdate;  
  
    private Integer is_hot; // 0 不是热门 1: 热门  
    private String pdesc;  
    private Integer pflag; // 0 未下架 1: 已经下架  
  
    private String cid;  
    ...  
}
```

- 步骤 5: dao 层实现

```
src  
  cn.itcast.dao  
    ProductDao.java
```

```
public class ProductDao {  
  
    /**  
     * 查询所有商品  
     * @return  
     */  
    public List<Product> findAll(){  
        try {  
            QueryRunner queryRunner = new QueryRunner(C3P0Utils.getDataSource());  
            String sql = "select * from product";  
            Object[] params = {};  
            return queryRunner.query(sql,  
                new BeanListHandler<Product>(Product.class), params);  
        } catch (Exception e) {  
            throw new RuntimeException(e);  
        }  
    }  
}
```

- 步骤 6: service 层实现



```
public class ProductService {

    /**
     * 查询所有商品
     * @return
     */
    public List<Product> findAll(){
        ProductDao productDao = new ProductDao();
        return productDao.findAll();
    }
}
```

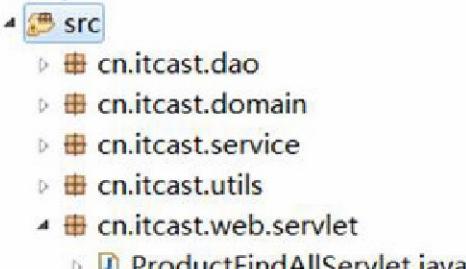
- 步骤 7：首页编写

A screenshot of a JSP file named 'index.jsp'. The code includes several lines of JSP scriptlets and comments for redirection:

```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2     pageEncoding="UTF-8"%>
3 <%-- 重定向到查询所有 servlet --%>
4<%>
5     response.sendRedirect("/day11_findall/productFindAllServlet");
6 %>
7

<%-- 重定向到查询所有 servlet --%>
<%
    response.sendRedirect("/day11_findall/productFindAllServlet");
%>
```

- 步骤 8：servlet 编写



```
<servlet>
    <servlet-name>ProductFindAllServlet</servlet-name>
    <servlet-class>cn.itcast.web.servlet.ProductFindAllServlet</servlet-class>
</servlet>
<servlet-mapping>
```

```
<servlet-name>ProductFindAllServlet</servlet-name>
<url-pattern>/productFindAllServlet</url-pattern>
</servlet-mapping>
```

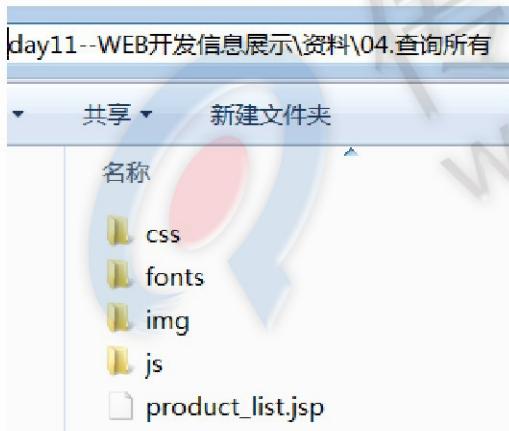
#### 代码实现

```
public class ProductFindAllServlet extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        //1 获得数据，并封装（本案例没有）
        //2 调用 service 进行查询所有
        ProductService productService = new ProductService();
        List<Product> allProduct = productService.findAll();

        //3.1 将查询结果存放在 request 作用域（查询数据当前有效）
        request.setAttribute("allProduct", allProduct);
        //3.2 请求转发到 jsp (jsp 可以使用 request 作用域的数据)
        request.getRequestDispatcher("/product_list.jsp").forward(request, response);

    }
    ...
}
```

- 步骤 9：拷贝 jsp 页面，显示具体信息



```
product_list.jsp
101      <%-- 列表 start --%>
102      <%
103
104      List<Product> allProduct = (List<Product>)request.getAttribute("allProduct");
105      for(Product p : allProduct){
106          out.write("<div class='col-md-2'>");
107          out.write(" <a href='product_info.htm'>");
108          out.write("   <img src='"+p.getPicImage()+"' width='170' height='170' style='display: inline-block;'>");
109          out.write(" </a>");
110          out.write(" <p><a href='product_info.html' style='color:green'>" + p.getPname() + "</a></p>");
111          out.write(" <p><font color='#FF0000'>商城价: &yen;" + p.getShop_price() + "</font></p>");
112          out.write("</div>");
113      }
114      <%>
115      <%-- 列表 end --%>
```

```
<%-- 列表 start --%>
```

```
<%
List<Product> allProduct = (List<Product>)request.getAttribute("allProduct");
for(Product p : allProduct){
    out.write("<div class='col-md-2'>");
    out.write(" <a href='product_info.htm'>");
    out.write("     <img src='"+p.getPicImage()+"' width='170' height='170'
style='display: inline-block;'>");
    out.write(" </a>");
    out.write(" <p><a href='product_info.html'
style='color:green'>" + p.getProductName() + "</a></p>");
    out.write(" <p><font color='#FF0000'> 商城价 : " +
    &yen; + p.getShopPrice() + "</font></p>");
    out.write("</div>");
}
%>
<%-- 列表 end --%>
```

## 第2章 总结

