# [JSP](https://www.cnblogs.com/xdp-gacl/p/3764991.html)

## JSP原理

JSP这门技术的最大的特点在于，写jsp就像在写html，但它相比html而言，html只能为用户提供静态数据，而Jsp技术允许在页面中嵌套java代码，为用户提供动态数据。

JSP是为了简化Servlet的工作出现的替代品，Servlet输出HTML非常困难，JSP就是替代Servlet输出HTML的。

Tomcat访问任何的资源都是在访问Servlet！，当然了，JSP也不例外！JSP本身就是一种Servlet。为什么我说JSP本身就是一种Servlet呢？**其实JSP在第一次被访问的时候会被编译为HttpJspPage类（该类是HttpServlet的一个子类）**

**地址：**

<%

String path = request.getContextPath();

String basePath = request.getScheme()+"://"+request.getServerName()+":"+request.getServerPort()+path+"/";

%>

**1.编译过程是这样子的：**浏览器第一次请求1.jsp时，Tomcat会将1.jsp转化成**1\_jsp.java这么一个类**，并将该文件编译成class文件。编译完毕后再运行class文件来响应浏览器的请求。

2.以后访问1.jsp就不再重新编译jsp文件了，直接**调用class文件来响应浏览器**。当然了，如果Tomcat检测到JSP页面改动了的话，会重新编译的。

3.有人可能也会问：**JSP页面的代码服务器是怎么执行的**？再看回1\_jsp.java文件，**java代码就直接在类中的service()中。**

**4.JSP比Servlet更方便更简单的一个重要原因就是：内置了9个对象！**

PageContext pageContext;

HttpSession session;

**ServletContext** application;

ServletConfig config;

JspWriter out;

Object page = this;

HttpServletRequest request,

HttpServletResponse response

其中page对象，request和response已经完成了实例化，而其它5个没有实例化的对象。

### out对象

out对象是通过调用**pageContext对象的getOut方法返回的**，其作用和用法与ServletResponse.getWriter方法返回的PrintWriter对象非常相似。   
　　JSP页面中的out对象的类型为JspWriter，**JspWriter相当于一种带缓存功能的PrintWriter，**设置JSP页面的page指令的buffer属性可以调整它的缓存大小，甚至关闭它的缓存。   
　　**只有向out对象中写入了内容，且满足如下任何一个条件时，out对象才去调用ServletResponse.getWriter方法，并通过该方法返回的PrintWriter对象将out对象的缓冲区中的内容真正写入到Servlet引擎提供的缓冲区中**：

* 设置page指令的buffer属性关闭了out对象的缓存功能
* out对象的缓冲区已满
* 整个JSP页面结束

### pageContext对象

pageContext对象是JSP技术中最重要的一个对象，它代表**JSP页面的运行环境，**这个对象不仅**封装了对其它8大隐式对象的引用**，它自身还是一个域对象(容器)，可以用来保存数据。并且，这个对象还封装了web开发中经常涉及到的一些常用操作，例如引入和跳转其它资源、检索其它域对象中的属性等。

**pageContext对象可以作为容器**来使用，因此可以将一些数据存储在pageContext对象中。

1 public void setAttribute(java.lang.String name,java.lang.Object value)

2 public java.lang.Object getAttribute(java.lang.String name)

3 public void removeAttribute(java.lang.String name)

**findAttribute方法**按照查找顺序"page→request→session→application"在这四个对象中去查找，只要找到了就返回属性值，如果四个对象都没有找到要查找的属性，则返回一个null。

**围绕自定义标签**写一个java类，jsp引擎在执行自定义标签的时候就会调用围绕自定义标签写的那个java类，在调用java类的时候就会把pageContext对象传递给这个java类，由于pageContext对象封装了对其它8大隐式对象的引用，因此在这个java类中就可以使用jsp页面中的8大隐式对象(request，response，config，application，exception，Session，page，out)了，**pageContext对象在jsp自定义标签开发中特别重要。**

**pageContext还可以封装了访问其他域的方法！**

* 上面的pageContext默认是page范围的，但pageContext对象重载了set、get、removeAttribute这三个方法
  + getAttribute(String name,int scope)
  + setAttribute(String name,Object value,int scope)
  + removeAttribute(String name,int scope)
* PageContext.APPLICATION\_SCOPE
* PageContext.SESSION\_SCOPE
* PageContext.REQUEST\_SCOPE
* PageContext.PAGE\_SCOPE

**使用pageContext的forward方法跳转到其他页面**

**PageContext类中定义了一个forward方法和两个include方法来分别简化和替代RequestDispatcher.forward方法和include方法。**

* pageContext.forward(String url)
* pageContext.include(String url)

pageContext.forward("/pageContextDemo05.jsp");

这种写法是用来简化和替代pageContext.getRequest().getRequestDispatcher("/pageContextDemo05.jsp").forward(request, response);

在开发中，要想从一个Jsp页面采用服务器端跳转的方式跳转到另一个Jsp页面，那么一般会使用**<jsp:forward>标签，**<jsp:forward>标签用于把请求转发给另外一个资源。

在实际开发中，使用pageContext的include方法引入页面这种做法也很少用，一般都使用**jsp:include标签**引入资源，因此这种做法了解一下即可。

## JSP的语法

JSP代码可以分为两部分：

1. **模板数据：就是HTML代码**
2. **元素：JSP页面中的java代码、JSP指令、JSP标签**

**JSP表达式**

语法：**<%= 变量或表达式 %>；**JSP脚本表达式中的变量或表达式后面不能有分号（;）。

**JSP的脚本**

就是JSP页面中的java代码，**也叫做scriptlet**。scriptlet标签：<jsp:scriptlet>，更美观。

**在<% %>中可以定义变量、编写语句，不能定义方法。**

**JSP声明**

JSP页面中编写的所有代码，默认会翻译到servlet的service方法中， 而Jsp**声明中的java代码被翻译到\_jspService方法的外面**。语法：  
　　　　<%！   
    　　　　java代码  
　　　　%>

　　所以，JSP声明可用于定义JSP页面转换成的Servlet程序的**静态代码块、成员变量和方法 。**

**JspInit相同的名字，覆盖方法**

　　多个静态代码块、变量和函数可以定义在一个JSP声明中，也可以分别单独定义在多个JSP声明中。

**JSP注释**

显式注释：直接使用**HTML**风格的注释：**<!- - 注释内容- ->**

隐式注释：直接使用**JAVA**的注释：**//**、**/\*……\*/**

<%--这是JSP注释--%>

## JSP指令

### page指令

<%@指令  属性名="值"  %>

**page指令用于定义JSP页面的各种属性**

<%@ page

[ language="java" ]

[ extends="package.class" ]

[ import="{package.class | package.\*}, ..." ]

[ session="true | false" ]

[ buffer="none | 8kb | sizekb" ]

[ autoFlush="true | false" ]

[ isThreadSafe="true | false" ]

[ info="text" ]

[ errorPage="relative\_url" ]

[ isErrorPage="true | false" ]

[ contentType="mimeType [ ;charset=characterSet ]" | "text/html ; charset=ISO-8859-1" ]

[ pageEncoding="characterSet | ISO-8859-1" ]

[ isELIgnored="true | false" ]

%>

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

在**conf/web.xml**文件中可以查询出来，对应所有的contentType

page指令errorPage和isErrorPage这两个属性，通过page指令来实现**跳转到友好提示页面上**！（**属于是服务器跳转**）但是是单个页面

**page指令的errorPage属性**

* errorPage属性的设置值必须使用**相对路径，如果以“/”开头**，表示相对于当前Web应用程序的根目录(注意不是站点根目录)，否则，表示相对于当前页面
* 可以在**web.xml文件**中使用<error-page>元素为整个Web应用程序设置错误处理页面。
* <error-page>元素有3个子元素，<error-code>、<exception-type>、<location>
* <error-code>子元素指定错误的状态码，例如：<error-code>404</error-code>
* <exception-type>子元素指定异常类的完全限定名，例如：<exception-type>java.lang.ArithmeticException</exception-type>
* <location>子元素指定以“/”开头的**错误处理页面的路径**，例如：<location>/ErrorPage/404Error.jsp</location>// ErrorPage包名
* 如果设置了某个JSP页面的errorPage属性，那么在web.xml文件中设置的错误处理将不对该页面起作用。

**可以在web.xml文件中全局设置错误页**

**只要发生了404错误或者空指针异常的错误都会跳转到error.jsp页面上**

**web.xml**

<error-page>  
    <error-code>404</error-code>  
    <location> /ErrorPage/404Error.jsp</location>  
</error-page>  
  
<error-page>  
    <exception-type>java.lang.NullPointerException</exception-type>  
    <location> /ErrorPage/404Error.jsp</location>  
</error-page>

**404Error.jsp代码**

<%@ page language="java" import="java.util.\*" pageEncoding="UTF-8"%>

<html>

<head>

<title>404错误友好提示页面</title>

**<!-- 3秒钟后自动跳转回首页 -->**

**<meta http-equiv="refresh" content="3;url=${pageContext.request.contextPath}/index.jsp">**

</head>

<body>

<img alt="对不起，你要访问的页面没有找到，请联系管理员处理!"

src="${pageContext.request.contextPath}/img/404Error.png"/><br/>

3秒钟后自动跳转回首页，如果没有跳转，请点击<a href="${pageContext.request.contextPath}/index.jsp">这里</a>

</body>

</html>

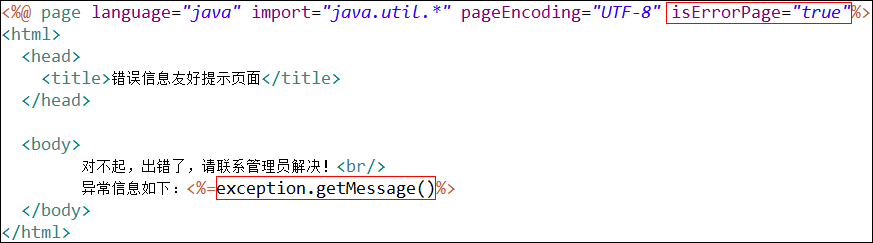
**随便输个资源进行，会发生发404错误的，跳转到错误页面**

这里需要注意的是，**如果错误页面比较小**，那么当访问服务器上不存在的web资源或者访问服务器出错时在IE浏览器下是无法跳转到错误页面的，显示的是ie自己的错误页面，而在火狐和google浏览器下(其他浏览器没有测试过)是不存在注意的问题的。

**不修改IE浏览器的设置下确保定制的错误页面的大小>1024字节**

Jsp有9大内置对象，而一般情况下exception对象在Jsp页面中是获取不到的，只有设置page指令的**isErrorPage**属性为**"true"**来显式声明Jsp页面是一个错误处理页面之后才能够在Jsp页面中使用exception对象。

**isErrorPage="true"；**

将error.jsp页面显式**声明为错误处理页面**后，有什么好处呢，好处就是Jsp引擎在将jsp页面翻译成Servlet的时候，**在Servlet的 \_jspService方法中会声明一个exception对象**，然后将运行jsp出错的**异常信息存储到exception对象**中

### include指令

在讲解request对象的时候，我们曾经使用过request.getRequestDispatcher(String url).include(request,response)来对页头和页尾面进行包含

在JSP中对于包含有两种语句形式：

1. @include指令
2. <jsp:include>指令

@include可以包含任意的文件，当然，只是把文件的内容包含进来。

include指令用于引入其它JSP页面，如果使用include指令引入了其它JSP页面，那么JSP引擎将**把这两个JSP翻译成一个servlet。**所以include指令引入通常也称之为静态引入。

**把文件的代码内容都包含进来，再编译！**

语法：**<%@ include file="relativeURL"%>**，其中的file属性用于指定被引入文件的路径。路径以“/”开头，表示代表当前web应用。

**include指令细节注意问题：**

被引入的文件必须遵循JSP语法。

被引入的文件可以使用任意的扩展名，即使其扩展名是html，JSP引擎也会按照处理jsp页面的方式处理它里面的内容，**为了见明知意，JSP规范建议使用.jspf（JSP fragments(片段)）作为静态引入文件的扩展名。**

由于使用include指令将会涉及到2个JSP页面，并会把2个JSP翻译成一个servlet，所以**这2个JSP页面的指令不能冲突**（除了pageEncoding和导包除外）。

新建head.jspf页面和foot.jspf页面，分别作为jsp页面的头部和尾部，存放于**WebRoot下的jspfragments文件夹中**

在WebRoot文件夹下创建一个IncludeTagTest.jsp页面，在IncludeTagTest.jsp页面中使用**@include**指令引入head.jspf页面和foot.jspf页面

<%--使用include标签引入引入其它JSP页面--%>

<%@include file="/jspfragments/head.jspf" %>

<h1>网页主体内容</h1>

<%@include file="/jspfragments/foot.jspf" %>

Tomcat服务器的work\Catalina\localhost\day08 \org\apache\jsp目录下找到IncludeTagTest\_jsp.java

### taglib指令

JSP支持标签技术，要使用标签技术就先得声明**标签库和标签前缀**。taglib指令就是用来指明JSP页面内使用标签库技术。

## JSP行为

（JSP Actions）是**一组JSP内置的标签**

### include行为

其实include行为就是封装了request.getRequestDispatcher(String url).include(request,response)

语法：  
    **<jsp:include page="relativeURL | <%=expression%>" flush="true|false" />**  
　　page属性用于指定被引入资源的相对路径，它也可以通过执行一个表达式来获得。  
　　flush属性指定在插入其他资源的输出内容时，是否先将当前JSP页面的已输出的内容刷新到客户端。

**先编译被包含的页面，再将页面的结果写入到包含的页面中**

不管是<jsp:include>标签，还是include指令，它们都会把两个JSP页面内容合并输出，所以这两个页面不要出现重复的HTML全局架构标签，否则输出给客户端的内容将会是一个格式混乱的HTML文档。

**\*.jspf扩展名文件在jsp:include、@include和c:import中的区别**

JSP规范建议使用.jspf（JSP fragments）作为静态引入文件的扩展名。今天无意中发现，把一个JSP文件命名为jspf扩展名，然后include到另一个jsp文件中的，发现只有用"@include"指令的时候，jspf文件的内容才会被解析并执行其中的jsp指令和tag，而使用"jsp:include"和JSTL的"c:import"都没有用，jspf文件被当作纯文本文件处理了。

**解决办法一：修改web.xml文件，添加对扩展名为\*.jspf文件的映射**

<servlet-mapping>

<servlet-name>jsp</servlet-name>

<url-pattern>\*.jsp</url-pattern>

<!-- 让jspf扩展名同样成为JSP Servlet处理的文件。-->

<url-pattern>\*.jspf</url-pattern>

</servlet-mapping>

**解决办法二：修改Tomcat服务器的web.xml文件，添加对扩展名为\*.jspf文件的映射**

找到tomcat服务器的web.xml文件

找到名字为jsp的那个Servlet

然后根据Servlet名找到对应的servlet-mapping配置

在这里可以看到，名字为jsp的那个Servlet只支持\*.jsp和\*.jspx两种扩展名，因此可以在这个地方添加多一个<url-pattern>\*.jspf</url-pattern>

经过这样的配置之后，Tomcat服务器就可以正常解析和执行\*.jspf文件了。

### forward行为

在讲解request对象的时候，我们使用request.getRequestDispatcher(String url).forward(request,response)进行跳转。**其实forward行为就是对其封装**！

语法：  
    　　**<jsp:forward page="relativeURL | <%=expression%>" />**   
　　page属性用于指定请求转发到的资源的相对路径，它也可以通过执行一个表达式来获得。（**跳转属于服务器端跳转**）

### <jsp:param>行为

当使用<jsp:include>和<jsp:forward>标签引入或将请求转发给其它资源时，可以使用<jsp:param>标签向这个资源传递参数

语法：

<**jsp:include/jsp:forward** page="relativeURL | <%=expression%>">

<jsp:param name="parameterName" value="parameterValue|<%= expression %>" />

**</jsp:include>**

**使用<jsp:param>标签向被包含的页面传递参数**

**使用<jsp:param>标签向要跳转的页面传递参数**

## 什么是javaBean

**JavaBean是一个遵循特定写法的Java类（也称之为简单java对象--POJO）**，它通常具有如下特点：

* 这个Java类必须具有一个无参的构造函数
* 属性必须私有化。
* 私有化的属性必须通过public类型的方法暴露给其它程序，并且方法的命名也必须遵守一定的命名规范。

**JavaBean的属性可以是任意类型，并且一个JavaBean可以有多个属性**。

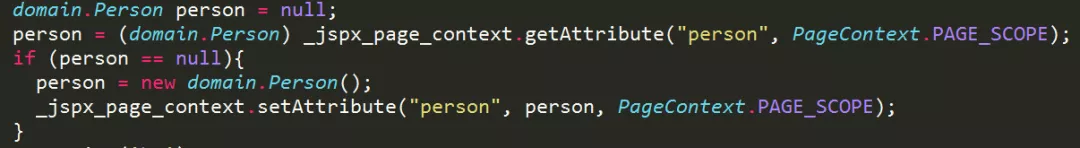
JaveBean你可以理解为一辆货车，在你的java端和web页面进行数据传递的载体，你当然可以每个变量单独传递，或者使用集合传递，但是**javabean可以使你的数据更有可读性，方便开发时明确变量的意义**，也使其他阅读你代码的人能直接你的意图

## 在JSP中使用JavaBean

JSP技术提供了**三个关于JavaBean组件的动作元素**，即JSP行为（标签）

### <jsp:useBean>标签

常用语法：  
   **<jsp:useBean id="beanName" class="package.class"scope="page|request|session|application"/>**  
　　"**id**"属性用于指定JavaBean实例对象的引用名称和其存储在域范围中的名称。  
　　"**class**"属性用于指定JavaBean的完整类名（即必须带有包名）。  
　　"**scope**"属性用于指定JavaBean实例对象所存储的域范围，其取值只能是page、request、session和application等四个值中的一个，其默认值是page。

<jsp:useBean>标签用于在指定的域范围内查找指定名称的JavaBean对象，如果存在则直接返回该JavaBean对象的引用，如果不存在则实例化一个新的JavaBean对象并将它**以指定的名称**存储到指定的域范围中。

通过上面的代码我们也**知道了为什么要有一个无参的构造函数**！**内部在new 对象的时候是没有传递参数进去的！**

**<jsp:useBean>标签的执行原理**："首先在指定的域范围内查找指定名称的JavaBean对象，如果存在则直接返回该JavaBean对象的引用，如果不存在则实例化一个新的JavaBean对象并将它以指定的名称存储到指定的域范围中。

### jsp:setProperty

**语法：**

<jsp:setProerty name="对象名称" property="属性名" param="参数名" or value="值">

**使用jsp:setProperty标签设置person对象的属性值**

<jsp:useBean id="person" class="gacl.javabean.study.Person" scope="page"/>//在jsp中使用**jsp:useBean标签来实例化一个Java类的对象**

<jsp:setProperty property="name" name="person" value="白虎神皇"/>

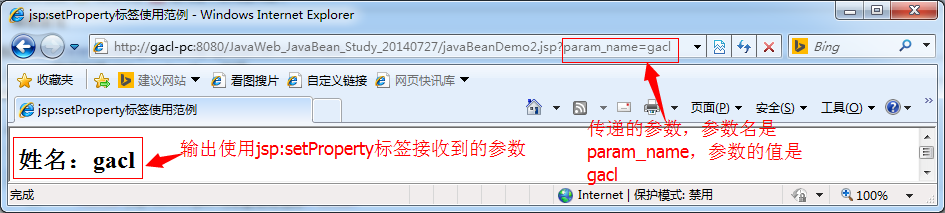
使用jsp:setProperty标签设置person对象的属性值

jsp:setProperty在设置对象的属性值时会自动把字符串转换成8种基本数据类型

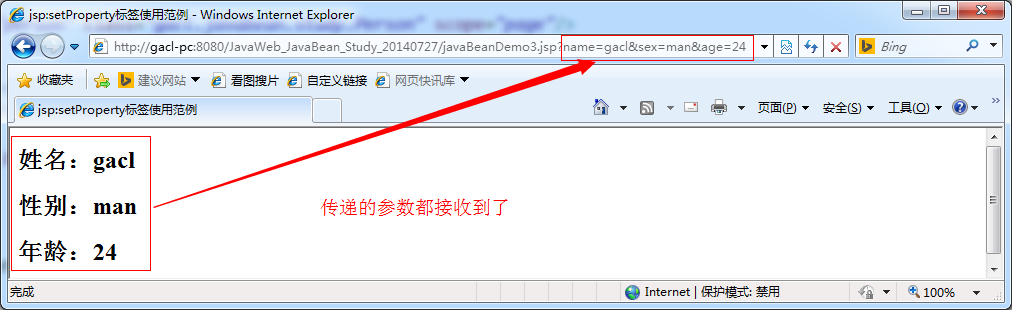
但是jsp:setProperty对于复合数据类型无法自动转换

<jsp:setProperty property="birthday" name="person" value="<%=new Date()%>"/>

**使用请求参数为bean的属性赋值**

<jsp:setProperty property="name" name="person" param="param\_name"/>

**用所有的请求参数为bean的属性赋值**

<jsp:setProperty property="\*" name="person"/>

当**我们没有学习到<jsp:setProperty>时**，我们获**取表单的信息，然后导入到javaBean对象中是这样的一种情况**：

**表单的页面代码**：

<form action="/zhongfucheng/1.jsp" method="post">  
    用户名：<input type="text" name="username">  
    年龄：<input type="text " name="age">  
    <input type="submit" value="提交">  
</form>

**处理表单提交过来数据的jsp的代码**

<jsp:useBean id="person" class="domain.Person" scope="page"/>  
<%--指定属性名称为age--%>  
<jsp:setProperty name="person" property="age"/>  
<%  
    System.out.println(person.getAge());  
%>

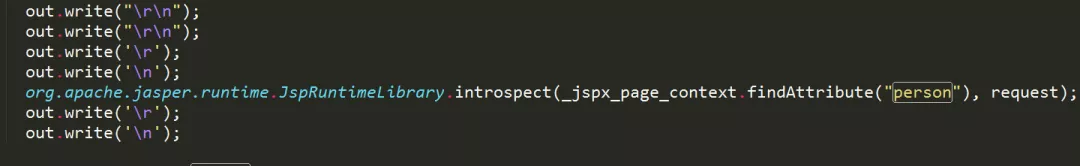
**自动匹配来感受它的强大之处吧**

<jsp:useBean id="person" class="domain.Person" scope="page"/>  
<%--property的值设置为\*就代表自动匹配--%>  
<jsp:setProperty name="person" property="\*"/>  
<%  
    System.out.println(person.getAge());  
    System.out.println(person.getName());  
%>

**只要设置property的值就可以将表单传递过来的数据封装到JavaBean对象中了**！

**JavaBean的属性名称和表单的name属性设置的名称是一模一样的**！

要想能够把表单带过来的数据成功封装到JavaBean对象上， **JavaBean属性名要和表单的name的名称一致**

**至于原理，它是通过反射来做的,调用了内省的方法！**

### <jsp:getProperty>标签

<jsp:getProperty>标签用于读取JavaBean对象的属性，也就是调用JavaBean对象的getter方法，然后将读取的属性值转换成字符串后插入进输出的响应正文中。  
　　语法：  
   <jsp:getProperty name="beanInstanceName" property="PropertyName" />