JSP常见面试题

jsp静态包含和动态包含的区别

> jsp静态包含和动态包含的区别

- **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**
- inclue指令也是做这样的事情，我们来试验一下吧！
- 这是页头

```
<%@ page contentType="text/html;charset=UTF-8" language="java"   %>
<html>
    <head>
        <title>页头</title>
    </head>
    <body>
    我是页头
    <br>
    <br>
    <br>
    </body>
</html>
```

- 这是页尾

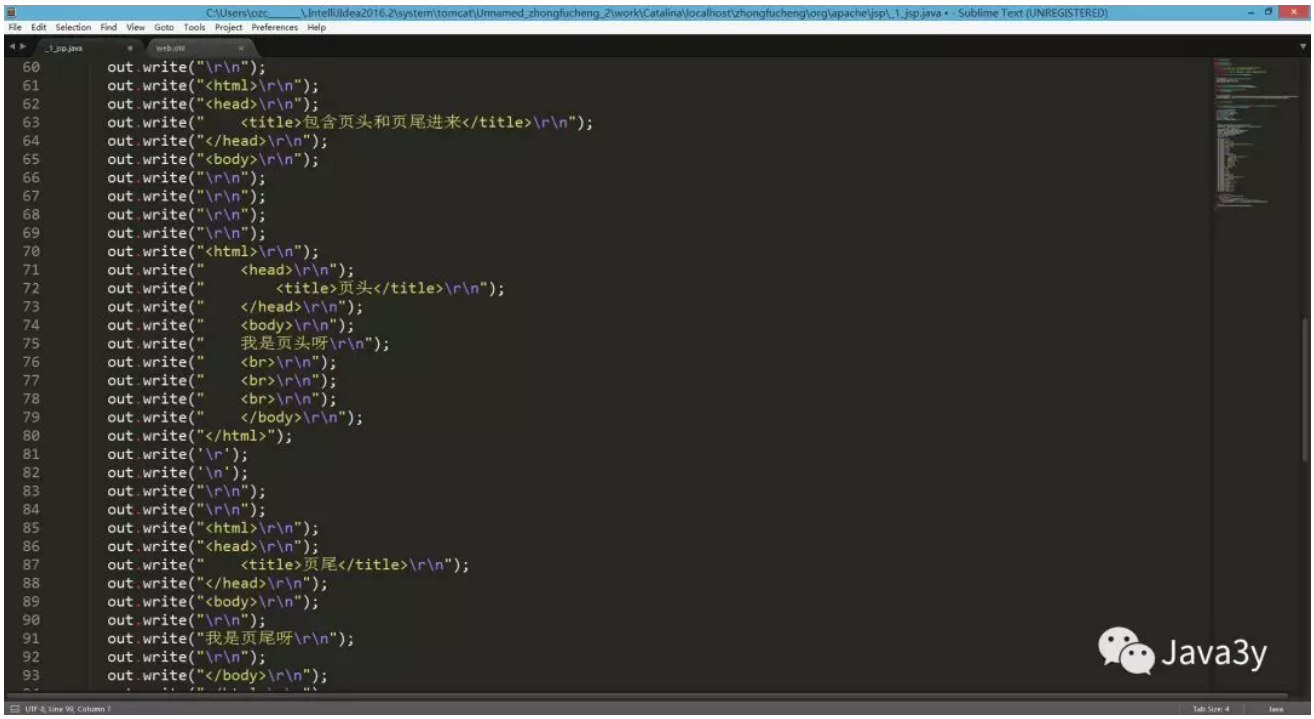
```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>页尾</title>
</head>
<body>

我是页尾

</body>
</html>
```

- **在1.jsp中把页头和页尾包含进来**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>包含页头和页尾进来</title>
</head>
<body>


<%@include file="head.jsp" %>
<%@include file="foot.jsp" %>
</body>
</html>
```

- 访问1.jsp

- **include指令是静态包含。**静态包含的意思就是：**把文件的代码内容都包含进来，再编译！**，看一下jsp的源代码就知道了！



- 上面已经提及到了，**include指令是静态包含，include行为是动态包含。**其实include行为就是封装了request.getRequestDispatcher(String url).include(request,response)

- include行为语法是这个样子的

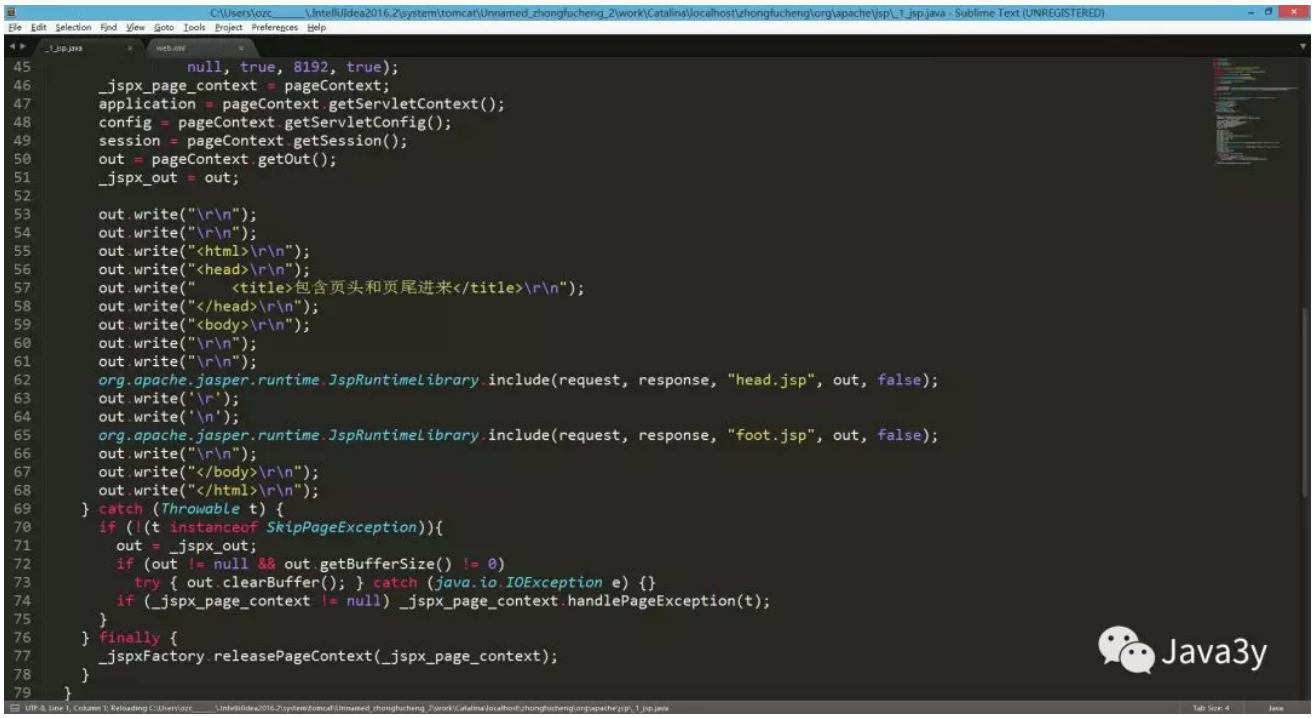
```
<jsp:include page=""/>
```

- 我们先来使用一下把，**在1.jsp页面中也将页头和页尾包含进来。**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>包含页头和页尾进来</title>
</head>
<body>
    <jsp:include page="head.jsp"/>
    <jsp:include page="foot.jsp"/>
</body>
</html>
```

- 访问1.jsp页面看一下效果：

- 使用jsp行为来包含文件，jsp源文件是这样子的：



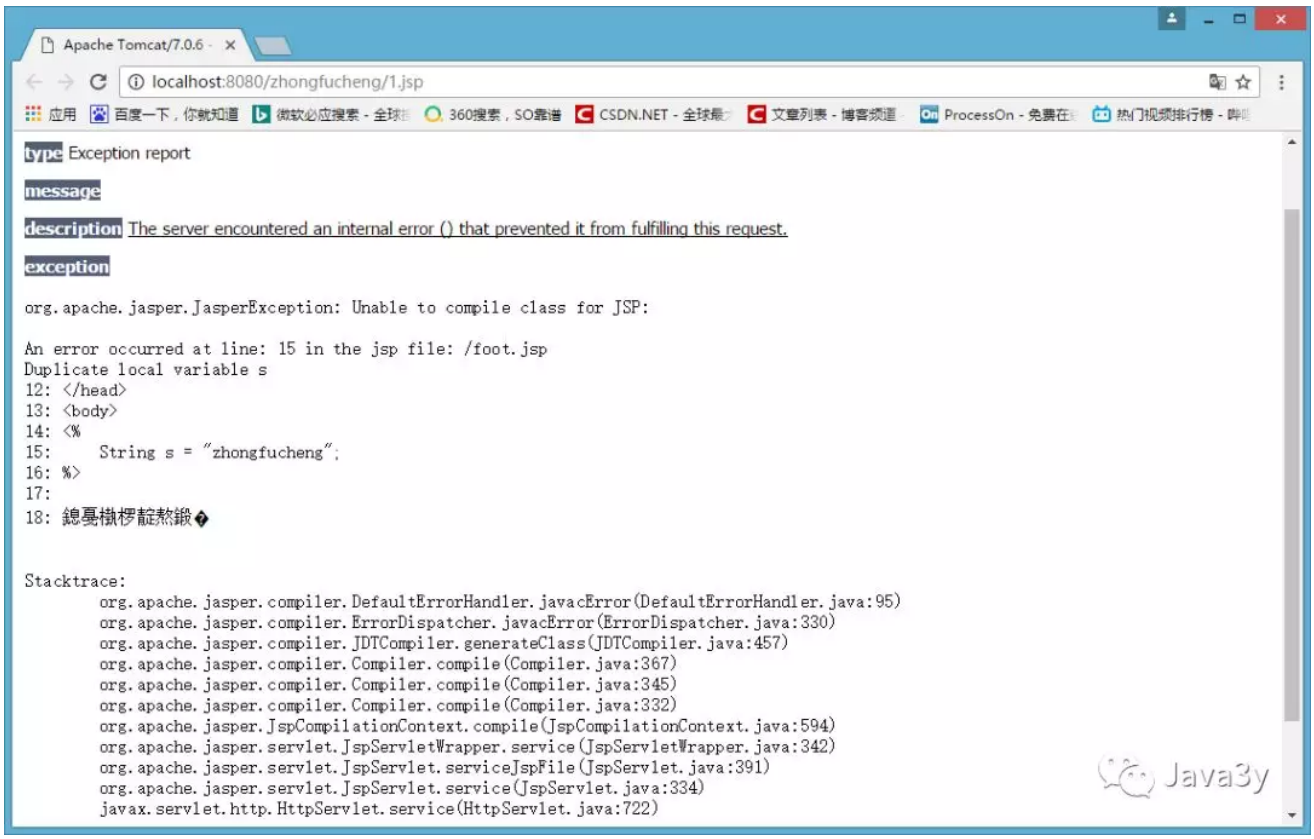
- jsp行为包含文件就是**先编译被包含的页面，再将页面的结果写入到包含的页面中（1.jsp）**
- 当然了，现在有静态包含和动态包含，使用哪一个更好呢？**答案是：动态包含。**
- 动态包含可以**向被包含的页面传递参数（用处不大）**，并且是**分别处理包含页面的（将被包含页面编译后得出的结果再写进包含页面）【如果有相同名称的参数，使用静态包含就会报错！】**！
- 模拟一下场景吧，现在我的头页面有个名为s的字符串变量

```
<%@ page contentType="text/html;charset=UTF-8" language="java"   %>
<html>
    <head>
        <title>页头</title>
    </head>
    <body>

    <%
        String s = "zhongfucheng";
    %>
    我是页头呀
    <br>
    <br>
    <br>
    </body>
</html>
```

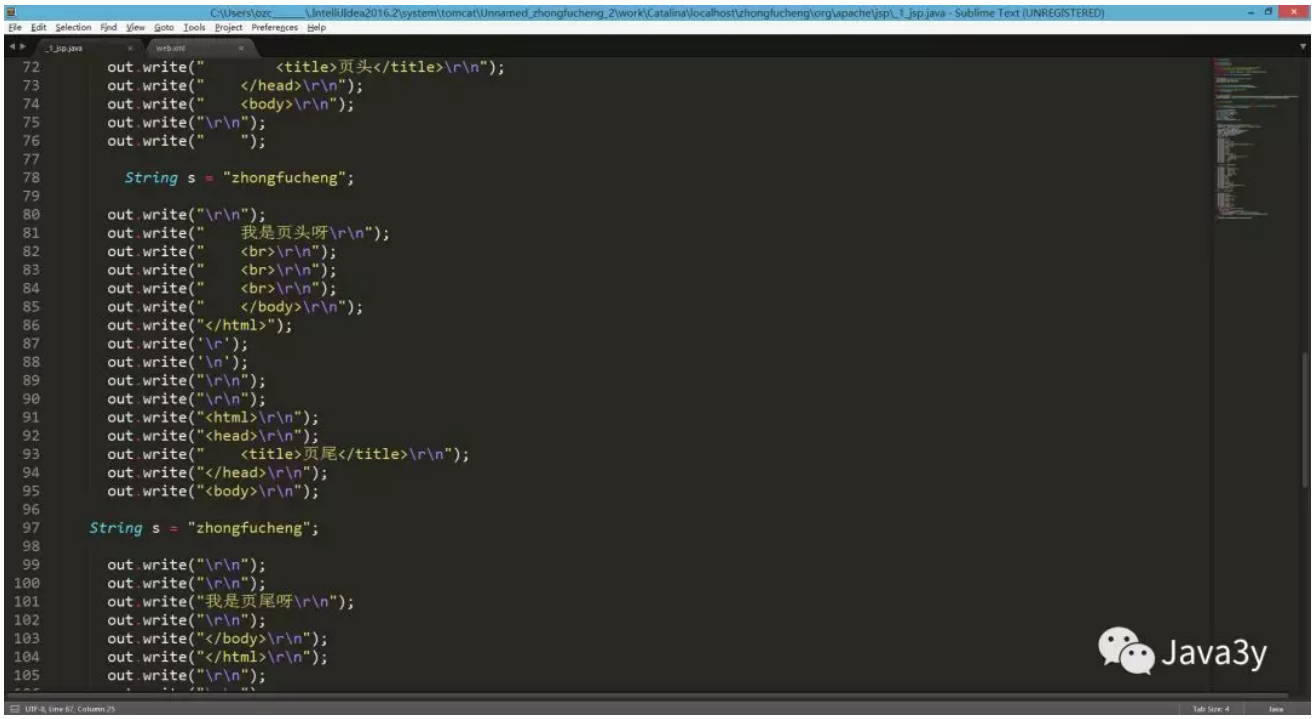
- 我的页尾也有个名为s的字符串变量

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>页尾</title>
</head>
<body>
<%
    String s = "zhongfucheng";
%>

我是页尾呀

</body>
</html>
```

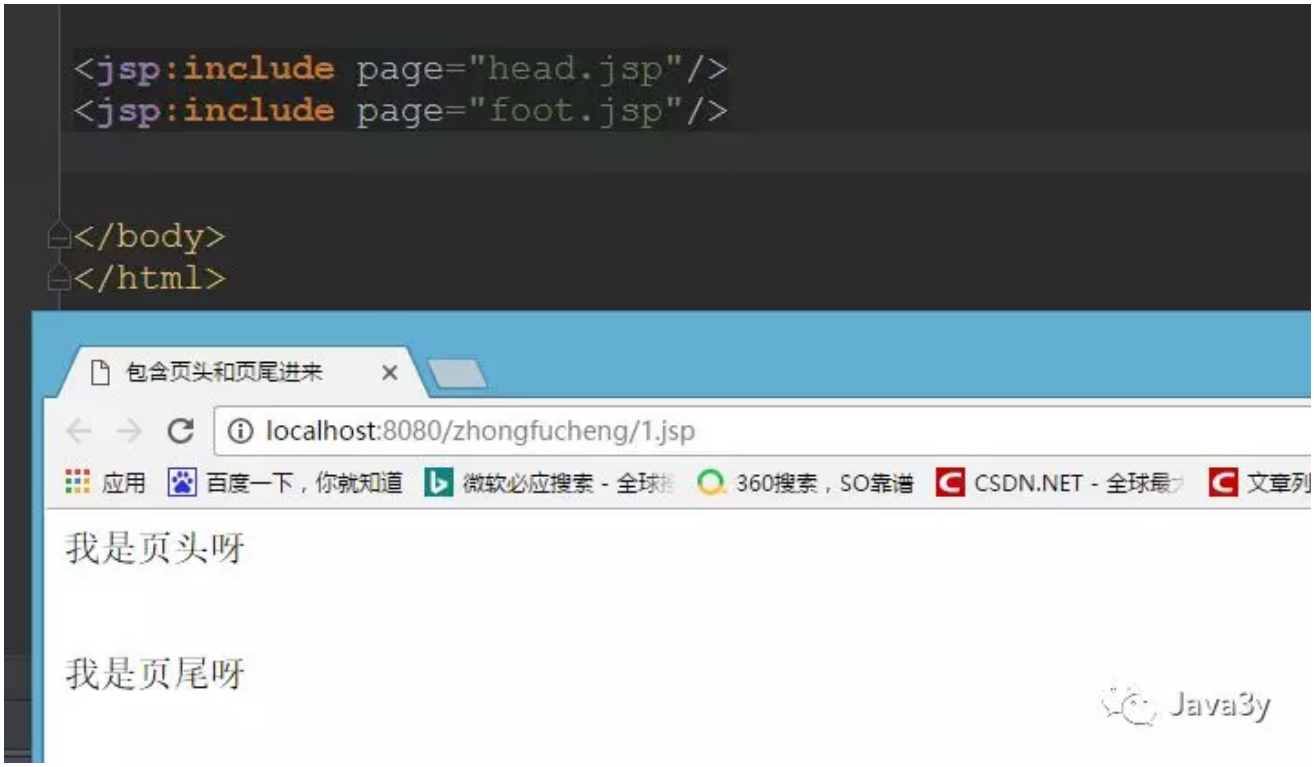
- 现在我使用静态包含看看会发生什么，出现异常了。

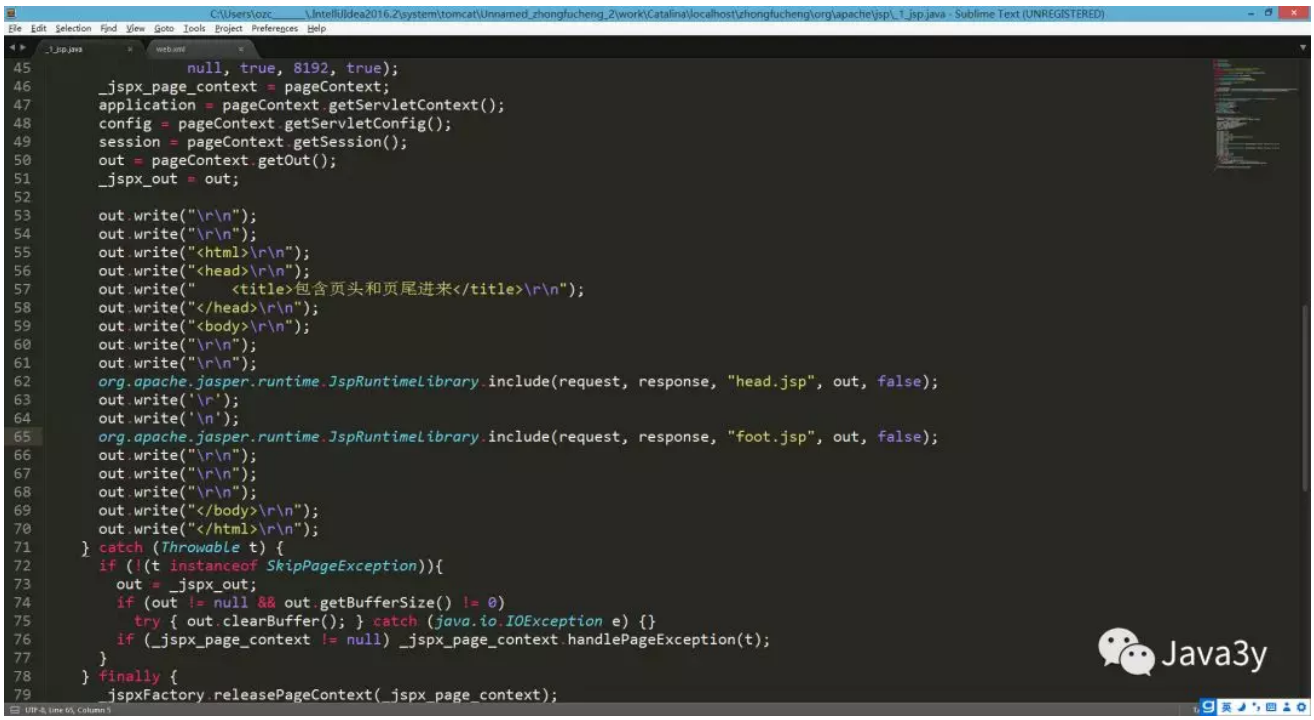


- 出现异常的原因很简单，就是**同一个文件中有两个相同的变量s**



- **使用动态包含就可以避免这种情况**





总结

1. `<%@include file="xxx.jsp"%>`为jsp中的编译指令，**其文件的包含是发生在jsp向servlet转换的时期**，而`<jsp:include page="xxx.jsp">`是jsp中的动作指令，其文件的包含是发生**在编译时期，也就是将java文件编译为class文件的时期**
2. 使用静态包含只会**产生一个class文件，而使用动态包含会产生多个class文件**
3. 使用静态包含，包含页面和被包含页面的request对象为同一对象，因为静态包含只是将被包含的页面的内容复制到包含的页面中去；而动态包含包含页面和被包含页面不是同一个页面，被包含的页面的request对象可以取到的参数范围要相对大些，不仅可以取到传递到包含页面的参数，同样也能取得在包含页面向下传递的参数

jsp有哪些内置对象?作用分别是什么?

> jsp有哪些内置对象?作用分别是什么?

九个内置对象：

- **pageContext**
- page
- **config**
- **request**
- **response**
- **session**
- **application**
- exception
- out

其中，request、response、session、application、config这五个对象和Servlet的API是一样的。这5个对象我就不解释了。

在JSP中，尤其重要的是pageContext对象。

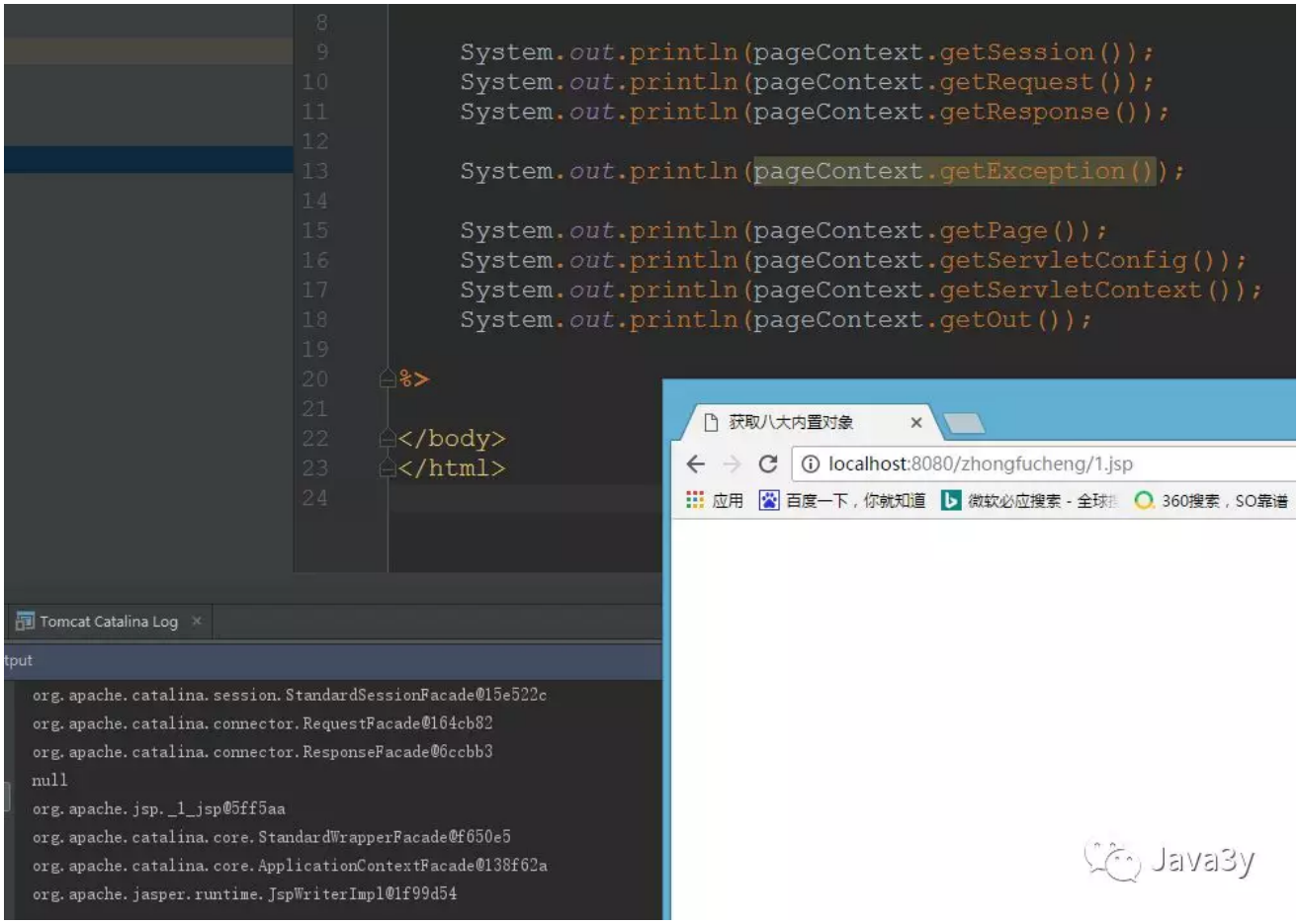
pageContext是内置对象中最重要的一个对象，它代表着JSP页面编译后的内容（也就是JSP页面的运行环境）！

> pageContext对象

- 既然它代表了JSP页面编译后的内容，理所当然的：**它封装了对其他8大内置对象的引用！**，也就是说，**通过pageContext可以获取到其他的8个内置对象！**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>获取八大内置对象</title>
</head>
<body>
<%

    System.out.println(pageContext.getSession());
    System.out.println(pageContext.getRequest());
    System.out.println(pageContext.getResponse());

    System.out.println(pageContext.getException());

    System.out.println(pageContext.getPage());
    System.out.println(pageContext.getServletConfig());
    System.out.println(pageContext.getServletContext());
    System.out.println(pageContext.getOut());

%>

</body>
</html>
```

- 看下效果：



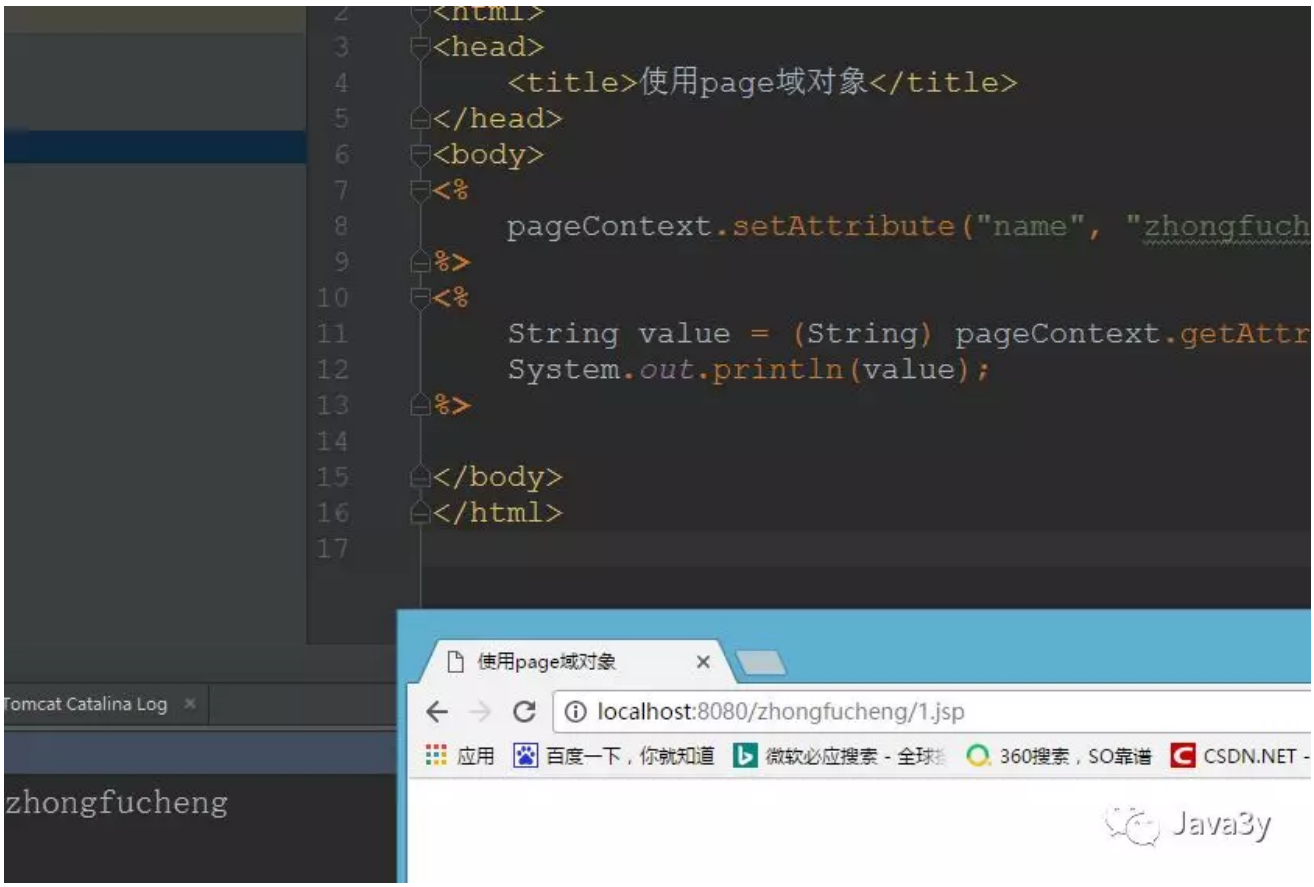
> pageContext作为域对象

- 类似于request，session，ServletContext作为域对象而言**都有以下三个方法**：
- - **setAttribute(String name,Objcet o)**
  - **getAttribute(String name)**
  - **removeAttribute(String name)**
- 当然了，pageContext也不例外，**pageContext也有这三个方法**！
- pageContext本质上代表的是当前JSP页面编译后的内容，**作为域对象而言，它就代表着当前JSP页面（也就是page）**！也就是说：**pageContext域对象只在page范围内有效，超出了page范围就无效了**！
- 首先来看看在**page范围内能不能使用**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>使用page域对象</title>
</head>
<body>
<%
    pageContext.setAttribute("name", "zhongfucheng");
%>
<%
    String value = (String) pageContext.getAttribute("name");
    System.out.println(value);
%>
```

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

- 效果如下：



- 我们现在来试验一下**是不是超出了page范围就无效了！**
- **在2.jsp中request域对象设置属性**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>request域对象设置属性</title>
</head>
<body>
<%
    //这是request域对象保存的内容
    request.setAttribute("name","zhongfucheng");
%>

<%--跳转到1.jsp中--%>

<jsp:forward page="1.jsp"/>

</body>
</html>
```

- **企图在1.jsp中pageContext取出request存进去的属性**

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>在page域对象获取属性</title>
</head>
<body>

<%
    //企图获取request域对象存进的属性
    String value = (String) pageContext.getAttribute("name");
    System.out.println(value);
%>

</body>
</html>
```

- 效果如下：

- pageContext本质上代表着编译后JSP的内容，**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)**
- **多了一个设置域范围的一个参数，如果不指定默认就是page。**当然了，**pageContext把request、session、application、page这几个域对象封装着了静态变量供我们使用。**
- - **PageContext.APPLICATION_SCOPE**
  - **PageContext.SESSION_SCOPE**
  - **PageContext.REQUEST_SCOPE**
  - **PageContext.PAGE_SCOPE**
- 刚才我们没有使用重载方法的时候，使用pageContext是无法获取到request域对象设置的属性的。**现在我们使用重载后的方法看一下能不能获取得到**！

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>在page域对象获取request域对象的属性</title>
</head>
<body>

<%
    //使用重载的方法获取request域对象的属性
    String value = (String) pageContext.getAttribute("name",pageContext.REQUEST_SCOPE);
    System.out.println(value);
%>

</body>
```

```
</html>
```

- 效果：

- pageContexst还有这么一个方法：
- - **findAttribute(String name)**
- **该方法会查找各个域的属性，从小到大开始寻找！也就是page—>request->session->application。**
- 我们用此方法看能不能查找出request域对象的属性吧！

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
    <title>使用findAttribute</title>
</head>
<body>

<%

    //使用findAttribute查找2.jsp中request域对象的属性
    String value = (String) pageContext.findAttribute("name");
    System.out.println(value);
%>

</body>
</html>
```

- 效果如下：

> out对象：

- **out对象用于向浏览器输出数据，与之对应的是Servlet的PrintWriter对象。**然而这个out对象的类型并不是PrintWriter，是JspWriter

- 我们可以简单理解为：**JspWriter就是带缓存的PrintWrieter。**
- out对象的原理如下：

- 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**
- **一般我们在JSP页面输出都是用表达式（<%=%>），所以out对象用得并不是很多**！

> page对象

内置对象page是HttpJasPage对象，其实page对象代表的就是当前JSP页面，是当前JSP编译后的Servlet类的对象。也就是说：page对象相当于普通java类的this

> exception对象

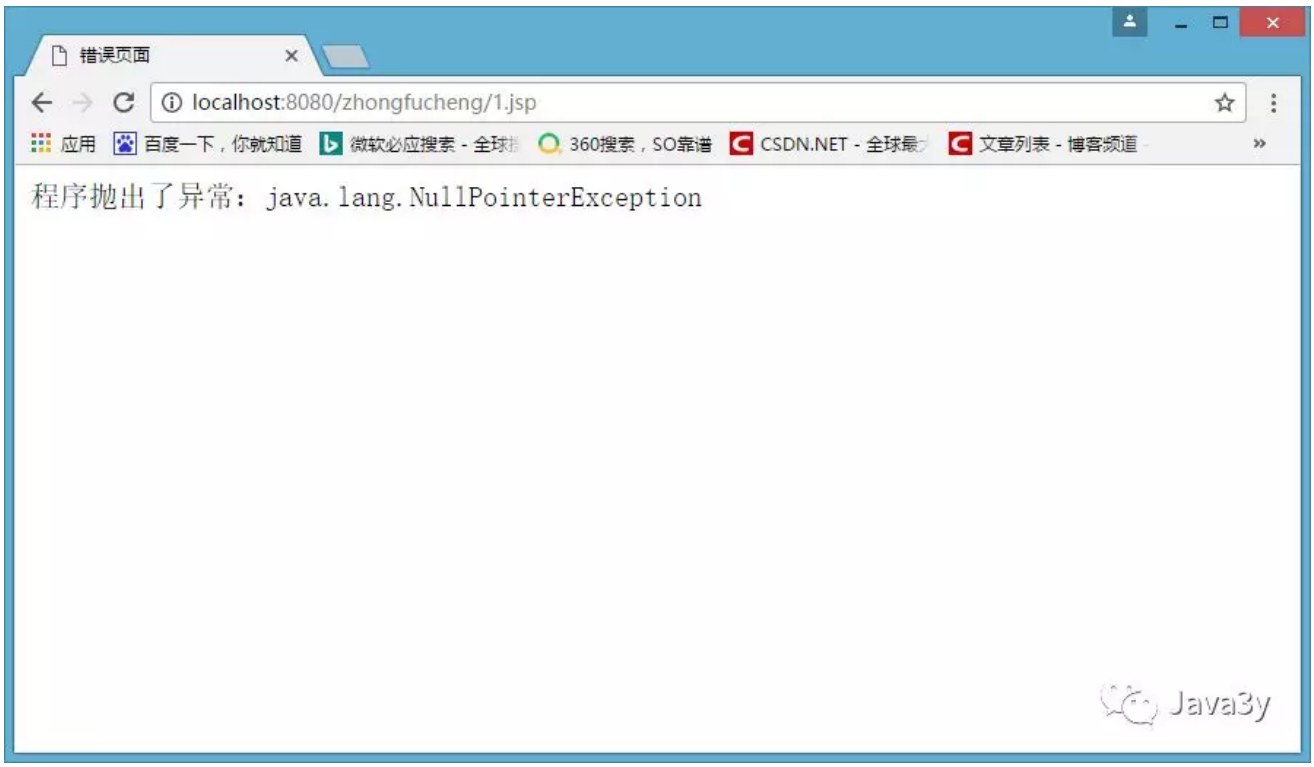
- **内置对象exception是java.lang.Exception类的对象，exception封装了JSP页面抛出的异常信息。**
  exception经常被用来处理错误页面
- 前面我们已**经讲过了怎么设置错误页面了**，下面我们就来简单使用一下exception对象吧
- 1.jsp页面

```
<%@ page contentType="text/html;charset=UTF-8" language="java" errorPage="error.jsp" %>

<html>
<head>
    <title></title>
</head>
<body>

<%--模拟空指针异常的错误--%>
<%

    String sss = null;
    sss.length();
%>

</body>
</html>
```

- error.jsp页面

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

<html>
<head>
    <title>错误页面</title>
</head>
<body>

<%
    out.println("程序抛出了异常：" + exception);
%>

</body>
</html>
```

- 效果：



总结

1. request 用户端请求，此请求会包含来自GET/POST请求的参数
2. response 网页传回用户端的回应
3. pageContext 网页的属性是在这里管理，代表的编译后JSP内容
4. session 与请求有关的会话期
5. application servlet 正在执行的内容
6. out 用来传送回应的输出
7. config servlet的构架部件
8. page JSP网页本身
9. exception 针对错误网页，未捕捉的例外

jsp和servlet的区别、共同点、各自应用的范围?

> jsp和servlet的区别、共同点、各自应用的范围?

1. JSP是Servlet技术的扩展，**本质上就是Servlet的简易方式。**JSP编译后是"类servlet"。
2. Servlet和JSP最主要的不同点在于：Servlet的应用逻辑是在Java文件中，并且完全从表示层中的HTML里分离开来。**而JSP的情况是Java和HTML可以组合成一个扩展名为.jsp的文件。**
3. **JSP侧重于视图，Servlet主要用于控制逻辑。**

属性作用域范围

> 属性作用域范围

1. **page【只在一个页面中保存属性，跳转页面无效】**
2. **requet【只在一次请求中保存属性，服务器跳转有效，浏览器跳转无效】**
3. **session【在一个会话范围中保存属性，无论何种跳转均有效，关闭浏览器后无效】**
4. **application【在整个服务器中保存，所有用户都可以使用】**

应用场景：

1. request：如果客户向服务器发请求，产生的数据，**用户看完就没用了，**像这样的数据就存在request域,像新闻数据，属于用户看完就没用的
2. session：如果客户向服务器发请求，产生的数据，**用户用完了等一会儿还有用**，像这样的数据就存在session域中，像购物数据，用户需要看到自己购物信息，并且等一会儿，还要用这个购物数据结帐
3. servletContext：如果客户向服务器发请求，产生的数据，**用户用完了，还要给其它用户用**，像这样的数据就存在servletContext域中，像聊天数据

写出5种JSTL常用标签

> 写出5种JSTL常用标签

```
<c:if>, <c:item>, <c:foreach>, <c:out>, <c:set>
```

写一个自定义标签要继承什么类

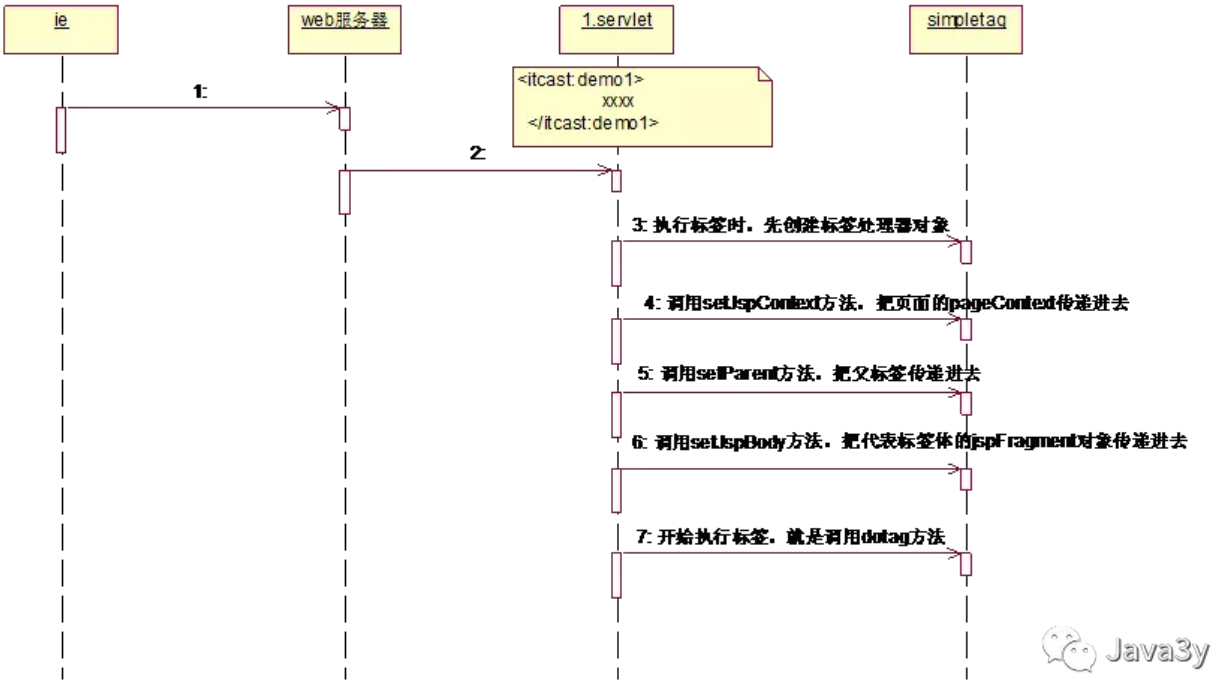
> 写一个自定义标签要继承什么类

我们可以有两种方式来实现自定义标签：

- 传统方式，实现Tag接口(老方法)
- **简单方式，继承SimpleTagSupport类**

SimpleTagSupport类的执行顺序(原理)：

- **①WEB容器调用标签处理器对象的setJspContext方法，将代表JSP页面的pageContext对象传递给标签处理器对象**
- **②WEB容器调用标签处理器对象的setParent方法，将父标签处理器对象传递给这个标签处理器对象。【注意，只有在标签存在父标签的情况下，WEB容器才会调用这个方法】**
- **③如果调用标签时设置了属性，容器将调用每个属性对应的setter方法把属性值传递给标签处理器对象。如果标签的属性值是EL表达式或脚本表达式，则WEB容器首先计算表达式的值，然后把值传递给标签处理器对象。**
- **④如果简单标签有标签体，容器将调用setJspBody方法把代表标签体的JspFragment对象传递进来**
- **⑤执行标签时：容器调用标签处理器的doTag()方法，开发人员在方法体内通过操作JspFragment对象，就可以实现是否执行、迭代、修改标签体的目的。**



总结

SimpleTagSupport，一般调用doTag方法或者实现SimpleTag接口

JSP是如何被执行的？执行效率比SERVLET低吗？

> JSP是如何被执行的？执行效率比SERVLET低吗？

- **当客户端向一个jsp页面发送请求时，Web Container将jsp转化成servlet的源代码（只在第一次请求时），然后编译转化后的servlet并加载到内存中执行，执行的结果response到客户端**
- **jsp只在第一次执行的时候会转化成servlet，以后每次执行，web容器都是直接执行编译后的servlet，所以jsp和servlet只是在第一次执行的时候不一样，jsp慢一点，以后的执行都是相同的**

如何避免jsp页面自动生成session对象？为什么要这么做？

> 如何避免jsp页面自动生成session对象？为什么要这么做？

可以使用页面指令显式关掉，代码如下：

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

jsp的缺点？

> jsp的缺点？

- **1）不好调试**
- **2）与其他脚本语言的交互(可读性差)**

说出Servlet和CGI的区别？

> 说出Servlet和CGI的区别？

- Servlet处于服务器进程中，只会有一个servlet实例，**每个请求都会产生一个新的线程，而且servlet实例一般不会销毁**
- **CGI：来一个请求就创建一个进程，用完就销毁，效率低于servlet**

简述JSP的设计模式。

> 简述JSP的设计模式。

在Web开发模式中，有两个主要的开发结构，称为模式一（Mode I）和模式二（Mode II）

首先我们来理清一些概念吧：

- **DAO(Data Access Object)：主要对数据的操作，增加、修改、删除等原子性操作。**
- **Web层：界面+控制器，也就是说JSP【界面】+Servlet【控制器】**
- **Service业务层：将多个原子性的DAO操作进行组合，组合成一个完整的业务逻辑**
- **控制层：主要使用Servlet进行控制**
- **数据访问层：使用DAO、Hibernate、JDBC技术实现对数据的增删改查**
- **JavaBean用于封装数据，处理部分核心逻辑，每一层中都用到！**

模式一指的就是**在开发中将显示层、控制层、数据层的操作统一交给JSP或者JavaBean来进行处理**！

模式一有两种情况：

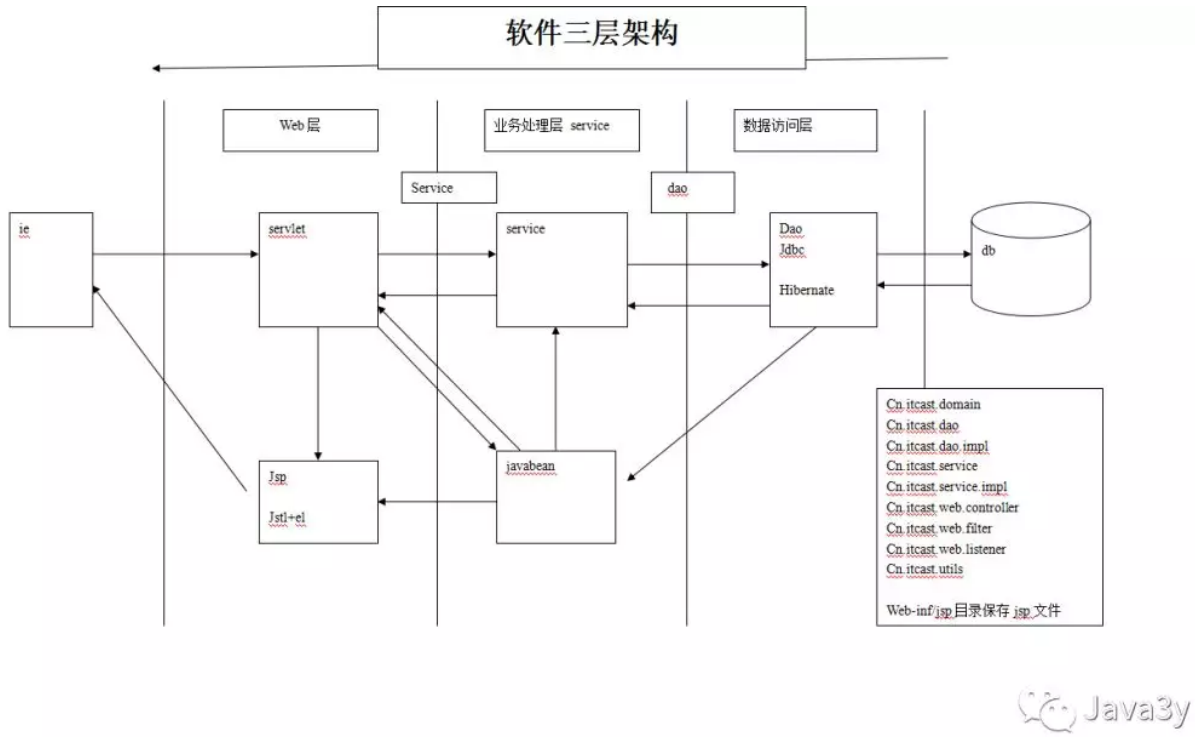
完全使用JSP做开发:

- 优点：
  - **开发速度贼快，只要写JSP就行了，JavaBean和Servlet都不用设计！**
  - **小幅度修改代码方便，直接修改JSP页面交给WEB容器就行了，不像Servlet还要编译成.class文件再交给服务器！【当然了，在ide下开发这个也不算是事】**
- 缺点：
  - **程序的可读性差、复用性低、代码复杂！什么jsp代码、html代码都往上面写，这肯定很难阅读，很难重用！**

使用JSP+JavaBean做开发:

- 优点：
  - **程序的可读性较高，大部分的代码都写在JavaBean上，不会和HTML代码混合在一起，可读性还行的。**
  - **可重复利用高，核心的代码都由JavaBean开发了，JavaBean的设计就是用来重用、封装，大大减少编写重复代码的工作！**
- 缺点：
  - **没有流程控制，程序中的JSP页面都需要检查请求的参数是否正确，异常发生时的处理。显示操作和业务逻辑代码工作会紧密耦合在一起的！日后维护会困难**

Mode II 中所有的开发都是以Servlet为主体展开的，由Servlet接收所有的客户端请求，然后根据请求调用相对应的JavaBean，并所有的显示结果交给JSP完成！，也就是俗称的MVC设计模式！



Java3y

MVC设计模式：

- **显示层（View）：主要负责接受Servlet传递的内容，调用JavaBean，将内容显示给用户**
- **控制层（Controller）：主要负责所有用户的请求参数，判断请求参数是否合法，根据请求的类型调用JavaBean，将最终的处理结果交给显示层显示！**
- **模型层（Mode）：模型层包括了业务层，DAO层。**

总结

- **（1）ModelI，JSP+JavaBean设计模式。**
- **（2）ModelII，MVC设计模式。**