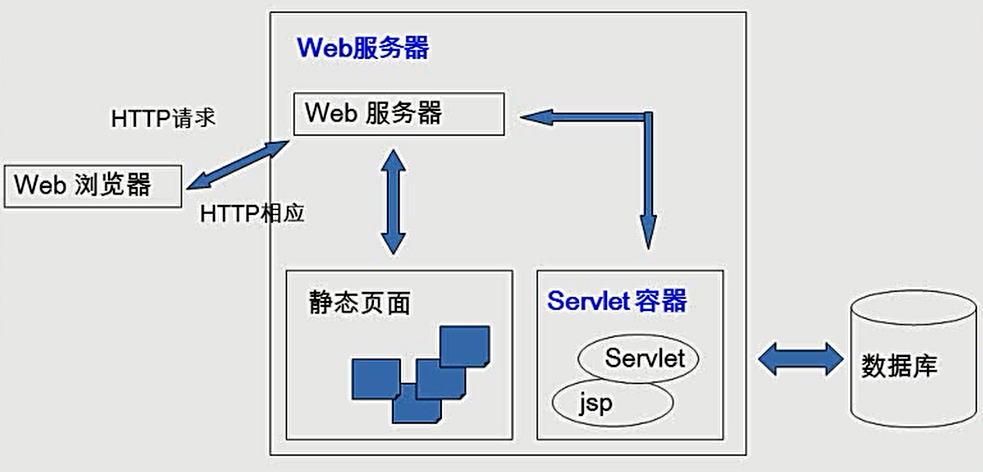
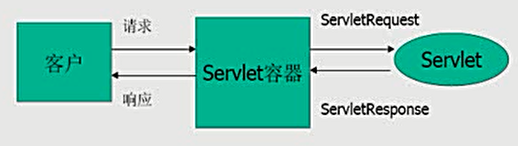
# JavaWeb

1. **Servlet、Servlet容器的作用**



* **Servlet容器：**Tomcat、Resin、J2EE服务器，为JavaWeb应用运行提供环境，负责管理Servlet、JSP的生命周期，和共享数据，负责Servlet和客户的通信以及调用Servlet方法（也称为JavaWeb应用容器）

**Servlet容器响应过程：**



* **Servlet：**功能：
* 创建、返回基于客户端请求的动态HTML页面
* 创建可嵌入到HTML页面的部分HTML页面片段
* 和其他服务器资源通信：数据库、Java应用等

1. **Servlet：**生命周期，**以下方法由Servlet容器负责调用，不需要手动调用。**

* **构造器：**在第一次请求Servlet时，构造器只被调用一次，因此，servlet是单实例的。
* **init方法**：在创建好实例后立即被调用（只被调用一次），用于初始化当前的Servlet
* **service方法**：每次请求都会调用service方法，用于实际的请求响应。
* **destory方法：**在Servlet所在的Web应用被卸载前调用。释放当前Servlet所占用的资源。

1. **Servlet中几个重要的接口方法：**

* **getInitParameterNames：**

获取web.xml中配置的**所有**初始化参数名

* **getInitParameter**

根据参数名，获取**特定**的参数值（需要用到上面的方法获取参数名）

* **getServletName**

获取当前servlet的名字

* **getServletContext：**

**每个servlet都有这个对象，代表当前web应用，可从中获取应用的各种信息**。该对象包含在：**ServletConfig**

* **getRealPath(String path)：**获取文件在服务器上的绝对路径（在out文件下），不是部署前的路径，用于文件上传、下载。
* **getContextPath()：**获取Web应用的名称
* **getResourceAsStream(String path)**：获取某个文件的输入流，path的‘/’为相对于当前的根目录
* **示例代码：**

public void init(ServletConfig servletConfig) throws ServletException {  
 System.*out*.println("servlet init ...");  
  
 **/\*\*  
 \* 使用ServletConfig，获取web.xml中配置的参数  
 \*/**  
 String user = **servletConfig.getInitParameter("user"); //web.xml配置的user参数**  
 System.*out*.println("user = " + user);  
 Enumeration<String> params = **servletConfig.getInitParameterNames();** while (params.hasMoreElements()){  
 String name = params.**nextElement**();  
 String value = servletConfig.**getInitParameter**(name);  
 System.*out*.println("paraName = " + name + " , value = " + value);  
 }  
 **/\*\*  
 \* 使用ServletContext对象  
 \*/**  
 **ServletContext** servletContext = **servletConfig.getServletContext();** String realPath = servletContext.**getRealPath**("/index.jsp");  
 System.*out*.println("realPath = " + realPath);  
  
}

1. **web.xml：配置Servlet的文件**

<servlet>  
  **<!-- Servlet注册名字 -->**  
 <servlet-name>helloServlet</servlet-name>  
  **<!-- Servlet所在的全类名 -->**  
 <servlet-class>Servlet.HelloWorld</servlet-class>  
 **<!-- 配置Servlet初始化参数 -->**  
 <init-param>  
 <param-name>user</param-name>  
 <param-value>root</param-value>  
 </init-param>  
 <init-param>  
 <param-name>password</param-name>  
 <param-value>1230re0321re</param-value>  
 </init-param>  
 **<!-- 指定Servlet创建时机：数字越小，越先创建 -->**  
 <load-on-startup>1</load-on-startup>  
</servlet>  
  
**<!-- 映射Servlet -->**  
<servlet-mapping>  
 **<!-- 对应上面的Servlet注册名字 -->**  
 <servlet-name>helloServlet</servlet-name>  
 **<!-- 自定义浏览器访问该Servlet的URL ：/代表当前的Web根目录 -->**  
 <url-pattern>/helloServlet</url-pattern>  
 **<!-- ‘\*’：任意URl均可访问该Servlet -->**  
 <url-pattern>/\*</url-pattern>  
</servlet-mapping>

**<!-- login的Servlet ：配置form action=”xx”的请求，指向特定的Servlet -->**  
<servlet>  
 <servlet-name>loginServlet</servlet-name>  
 <servlet-class>Servlet.LoginServlet</servlet-class>  
</servlet>  
  
<servlet-mapping>

**<!-- login.html请求"loginServlet"，对应的Servlet名 -->**  
 <servlet-name>loginServlet</servlet-name>  
 **<!-- 填写login.html的请求名：<form action="loginServlet" method="psot"> -->**  
 <url-pattern>**/loginServlet**</url-pattern>  
</servlet-mapping>

1. **HTTP协议中的GET、POST请求**

* **GET**

传输数据量有限制，一般在1KB之内，适用于浏览器向服务器**获取数据**时使用。在URL中可以看到提交的数据。

**参数放置的位置：**在url后面，**例如**：

[http://www.baidu.com/index.jsp？**name=root&password=123**](http://www.baidu.com/index.jsp？name=root&password=123)

* **POST**

传输的数据量比GET大（**没有限制**），适合用于向服务器**提交数据**时使用。

一般，**只有在表单中才会使用POST，其余的请求都是GET。**

**参数存放的位置：**用Form表单存放数据，将各个表单字段元素和数据作为HTTP消息实体内容（放在HTML HEADER内），发送给服务器。**例如：**

**POST:** /counter.jsp HTTP/1.1

**referer:**

**content-type:**

**…**

**name=root&password=123 //传递的参数**

1. **JAVA的Servlet接口、类的使用：**

**Servlet对象说明：**该对象在Servlet容器启动时被初始化，第一次请求时，由Servlet容器将其序列化，每一个请求是一个线程而不是进程

* **Servlet接口：**

定义基本的Servlet方法：

init()、destory()、getServletConfig()、service()----用于处理用户请求。。。

* **ServletConfig接口：**

**封装里Servlet的配置信息，每一个Servlet都有一个唯一的ServletConfig对象。**在初始化Servlet时，向Servlet传递信息，例如：

getServletName()：返回Servlet示例名称；

getServletContext()：返回ServletContext的引用 ；

* **ServletContext接口：**
* 每一个虚拟机都会有一个ServletContext，是Web服务器种的一个已知路径的根
* **作用：**

1. **获取Web应用在web.xml种的全局初始化参数：**

**eg：**

**web.xml中：**

<context-param>

<param-name>username</param-name>

<param-value>chris</param-value>

</context-param>

**xx.java中：**获取这些初始化参数

//获得ServletContext对象

ServletContext servletContext = this.getServletContext();

//获得初始化参数

String username = servletContext.getInitParameter("username");

1. **在web应用范围内，存取共享数据：**
2. **web应用范围：**

* web应用的生命周期构成的时间段
* web应用的生命周期内所有web组件的集合

1. **基本的应用：**

* setAtrribute(String name，Object obj)：将java对象和属性名绑定，存放到ServletContext中，该java对象用于共享数据；
* getAtrribute(String name)：根据参数名，获取相应的java对象；
* getAtrributeNames()：返回ServletContext中所有的属性名，使用枚举变量返回；
* getRealPath(“文件名”)：获取绝对路径；
* **示例：**

**//1、获取ServletContext对象**

ServletContext servletContext = servletConfig.getServletContext();

**//2、获取该文件“**index.jsp**”的绝对路径**  
String realPath = servletContext.getRealPath("/index.jsp");

**//3、设置contest属性**

servletContext.setAttribute("personName","Jim");

**//4、获取Context属性**

String personName = (String)servletContext.getAttribute("personName");

* **GenericServlet抽象类：**

直接继承：Servlet、ServletConfig、Serializable等接口

**作用：**在编写自己的Servlet时，直接继承该抽象类，可以快速的创建Servlet

* **HttpServlet抽象类：通常使用这个，一般说Servlet也指的是这个**

直接继承**GenericServlet抽象类，**

**作用：**用于快速编写自己的HttpServlet类，方便的获取Http请求进行处理，进而完成用户和程序之间的交互。

**例如：**浏览器的请求数据，对于<form>表单的请求，方法method=GET、POST，提交到对应的Servlet对象时，该Servlet对象调用doGet、doPost方法进行相应的处理。

* **如何自己编写、使用Servlet：**

1. 可以继承HttpServlet或GenericServlet等抽象类，并且重写doPost()、doGet()或service()
2. 在相应的.html或.jsp中，的表单填写相应的URL请求和方法，例如：

<form action = "**loginServlet**" method="post">

1. 在web.xml中配置Servlet：

**（1）首先 ，声明Servlet**

<servlet>

**<!—自定义的Servlet对象的标签名-->**  
 <servlet-name>login</servlet-name>

**<!—Servlet对象的具体位置：包名 + 类名 -->**

<servlet-class>ServletCheckJDBD.LoginServlet02</servlet-class>  
</servlet>

**（2）其次，映射Servlet：**将对应表单URL的请求和对应的Servlet对象绑定.

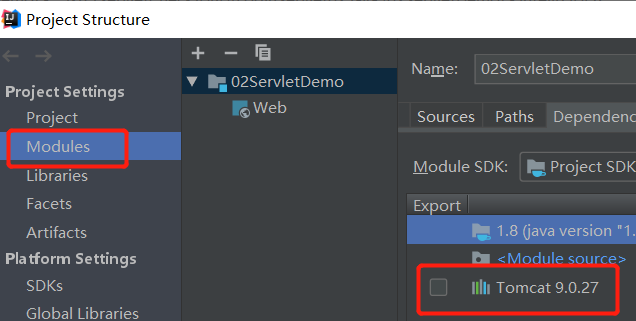
<servlet-mapping>  
 <servlet-name>login</servlet-name>

**<!—表单中的URL请求 -->**  
 <url-pattern>**/loginServlet**</url-pattern>

</servlet-mapping>

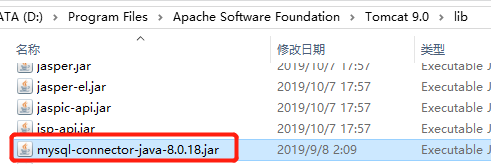
1. **关于Tomcat使用MySQL数据库的注意事项：**

* **IntelliJ IDEA使用Tomcat和MySQL的驱动时：**
* **需要把mysql-connector-java驱动放到Tomcat的安装路径lib内，这样就可以直接使用mysql的驱动了**，不需要在IDEA里面手动添加mysql驱动。因为，IDEA已经引用了Tomcat的库（如下图所示）。



必须这样添加mysql驱动，否则会出现找不到驱动的错误,eg：

java.lang.ClassNotFoundeException:com.mysql.cj.jdbc.Driver



* **关于资源文件.properties的使用：**

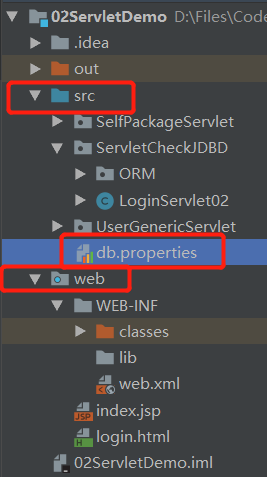
无论放在src还是web文件夹内都可以被识别到：

* **WEB-INF：**是Web应用的安全目录，用户无法使用浏览器直接访问。其中：（**下面的文件、文件夹一定要放在WEB-INF文件夹中**）
* **web.xml**：配置文件。
* **classes：**站点所有的class文件，所有src编译后的文件（前提：手动调整编译后文件的存放路径）。
* **lib：**驱动文件，Tomcat用到的所有驱动文件必须放在/WEB-INF/lib文件夹中，否则会出现找不到.jar文件的情况；

**例如：在Tomcathuan中，使用MySQL数据库：**

1. **Intellij IDEA工程**：必须把对应的数据库连接驱动（mysql-connector-java-8.0.18）放在/WEB-INF/lib文件中。否则一定会报错误： [java.sql.SQLException: No suitable driver](https://www.cnblogs.com/xiqoqu/p/9173885.html)，找不到驱动。
2. **Tomcat服务器安装目录下**：lib/文件夹中，要存放对应的数据库连接驱动：mysql-connector-java-8.0.18

Intellij IDEA中，WEB-INF的目录结构如下：



* **Intellij IDEA Debug调试经验：（和keil的调试很像）**

1. 清空所有断点；
2. 点击debug按钮，等页面加载完毕后，在想要debug的代码处添加断点；
3. 在浏览器上的web页面输入数据后，Intellij IDEA就会自动将代码运行到对应的断点处---接下来就可以debug对应的地方了；



# JSP

1. **什么是JSP：java server page，java服务器网页**

JSP是简洁的Servlet编写技术，将Java和HTML代码混合在一个文件中，网页中要动态产生的内容采用Java编写，固定不变得静态内容采用静态的HTMl编写。

**示例代码：在.jsp文件中编写 JSP表达式**

<body>  
 时间：  
 <%  
 Date date = new Date();  
 *out*.print(date);  
 %>

<body>

1. **JSP的运行原理：**

**JSP本质：JSP就是一个Servlet文件。**

每个JSP 页面在第一次被访问时, JSP 引擎将它翻译成一个 Servlet 源程序, 接着再把这个 Servlet 源程序编译成 Servlet 的 class 类文件.

然后再由WEB容器（Servlet引擎）像调用普通Servlet程序一样的方式来装载和解释执行这个由JSP页面翻译成的Servlet程序。

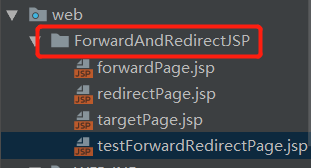
**intelliJ IDEA编译.jsp页面成.java的Servlet文件存放位置：**

C:\Users\Master\.IntelliJIdea2019.1\system\tomcat\**工程名** \work\Catalina\localhost\**工程名**\org\apache\jsp\**jsp页面名\_jsp.java**

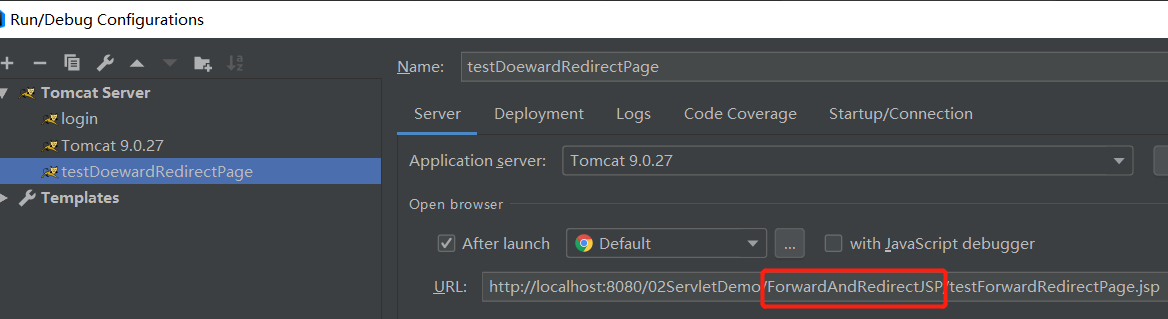
1. **JSP文件的存放位置：**

除了WEB-INF之外得任何目录文件中。但是如果是自己新建文件夹存放.jsp文件，想要在运行程序时，默认打开指定的JSP页面，需要在Tomcat配置中，添加这个jsp文件所在的文件夹名称：

**例如：自己新建文件夹：ForwardAndRedirectJSP**



修改Tomcat配置的路径，添加上jsp文件所在的文件夹名



**web.xml配置Servlet，需添加对应请求的jsp所在的目录：**/app/forServlet

（**注意**：必须把java源文件编译后的字节码文件（.class）放在/WEB-INF/classes目录内）

1. **JSP页面一共由9个隐含对象：在\_jspService函数中**

* **pageContext：**

页面的上下文，是PageContext的一个对象，可以从中获取其他剩余的8个对象，页面信息；

**作用范围：**当前的JSP页面

* **request：**

HttpServletRequest的一个对象；

**作用范围：**同一个请求

* **session：**

代表浏览器和服务器的一次会话，是HttpSession的一个对象；

**作用范围：**限于一次会话：浏览器打开🡪关闭，之间不会失效

* **application：**

代表当前的WEB应用，是ServletContext的一个对象；

**作用范围：**当前的WEB应用，范围最大，只要在任意地方设置属性，其他的JSP、Servlet都可以获取。

**（上面的4个：作用范围依次递增，application最大）**

* **out：**

JspWriter对象，用于输出，调用out.print()方法，可以输出数据到浏览器；

* **page：**

指向当前JSP对应的Servlet对象的引用，类型为Object，只能调用Object类的方法（基本不会使用）

* **exception：**

在声明page指令的isErrorPage=”true”时才能使用，用于异步处理Servlet请求，防止线程堵塞。

* **response：**

HttpServletResponse的对象（基本不会使用）

* **config：**

当前JSP对应的Servlet的ServletConfig对象，只有在需要访问JSP配置的初始化参数时，才会使用，需要通过地址映射才能使用。

（基本不会使用）

* **示例代码**：在.jsp文件中

<body>  
 <%

ServletRequest req = **pageContext**.getRequest(); **//获取请求**

**request**.setAttribute("requestAttr", "request"); **//设置属性，属性名：requestAttr**

System.*out*.println(**session**.getId()); **//获取窗口的ID**

System.*out*.println(**application**.getInitParameter("password")); **//获取初始化参数password**

System.*out*.println(**config**.getInitParameter("test")); **//获取初始化参数password**

**out**.println(**request**.getParameter("name"));  **//在浏览器输出数据**

%>  **//使用绝对路径访问JSP**

<a href**="${pageContext.request.contextPath}/**art/article.jsp">article</a>

</body>

1. **JSP的注释格式：**

* **JSP：**<%-- JSP注释 --%>，使用该注释后，不能运行被注释的JAVA代码
* **HTML：**<!—HTML注释 -->，使用注释后，可以运行被注释的JAVA代码

1. **请求转发和请求重定向：**

* **转发：（forward）**
* 转发后，浏览器上的URL地址，**还是转发前的页面地址**；
* 最终的Servlet中，request对象和转发前的时同一个；
* 只能转发给当前的WEB应用资源；
* **/：**代表当前**WEB应用**的根目录：**即Tomcat配置中的默认路径**

**例如：**http://localhost:8080/war\_explore/

* Tomcat服务器下，WEB-INF文件夹内的文件，无法在浏览器种输入地址的方式来访问，但是可以通过请求转发来访问，**路径为：相对路径**

**eg：在.java的Servlet中**

req.**getRequestDispatcher**("**/**index.jsp").forward(req,resp);

**在JSP中获取当前web应用根目录：/war\_explore/**

${pageContext.request.contextPath}

* **重定向：（redirect）**
* 重定向后，浏览器上的URL地址，**变成重定向后的页面地址；**
* 最终的Servlet中，request对象和重定向前不是同一个；
* 可以重定向到任意资源；
* **/**：代表当前**WEB站点**的根目录：

**例如：**<http://localhost:8080/>

**eg：在.java的Servlet中**

resp.**sendRedirect**("index.jsp");

* **转发 、 重定向的区别：（浏览器上的地址显示不同）**

1. **转发forward：**

服务器直接访问目标地址的URL，将URL的内容读取出来，然后发送给浏览器，浏览器并不知道服务器发送的内容来自哪里，因此浏览器上的地址还是原来的地址；

1. **重定向redirect：**

服务器发送一个状态码，告诉浏览器重新新请求去某个地址，因此浏览器上的地址是新的地址；

1. **JSP指令：**

* JSP指令的作用：不会产生任何的输出效果，只是用于告诉引擎如何处理当前的JSP页面；
* **JSP指令一共由3种：**
* **page：**

用于定义JSP页面的属性，无论page指令放在哪，该指令都能够作用于整个JSP页面。一般来说page指令放在JSP页面的最前面；

**常用属性如下：**

* **import：**指定该JSP对应的Servlet需要导入的类；

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

* **session：**指定当前session变量是否可用，即：访问当前JSP页面是否需要生成HttpSession对象。取值true、false。
* **errorPage：**

指定当前页面出现错误时，对应的处理页面

配置errorPage页面的方式：

1. **方式一：**

在可能出现异常的页面配置：

<%@ **page** language="java" contentType="text/html; **errorPage**="**/**error.jsp"%>

这里的/，标识当前WEB应用的根目录。

**②方式二：**

在web.xml中配置：

<error-page>  
 **<!-- 指定出错的代码: 404 没有指定的资源, 500 内部错误. -->** <error-code>404</error-code>  
 **<!—错误页面的位置 ：因为error.jsp不希望被用户通过浏览器直接访问到-->**  
 <**location**>**/WEB-INF/**error.jsp</location>  
</error-page>

* **isErrorPage：**

说明当前页面是否为错误处理页面

<%@ **page** language="java" contentType="text/html; **isErrorPage**="true"%>

* **contentType：**

指定当前JSP页面的响应类型，实际上调用的是：response.setContentType(“text/html; charset=UTF-8”);

一般来说，这个值都为text/html; charset=UTF-8，表示，即是这个页面为JSP，但最终用户访问的还是HTML格式的文件，编码为：UTF-8

* **pageEncoding：**

指定当前JSP页面的字符编码，一般来说，和contentType中的charset一样。

* **isELIgnored：**

指定当前JSP页面是否使用EL表达式，通常为：false

* **include：**

用于将其他JSP文件的**内容合并**到当前JSP页面中，也就是所谓的**静态引入（静态包含）**，将其他JSP页面完全导入当前JSP，将两个页面合并，**生成一个Servlet，编译出一个.class文件**。

**例如：**

<%@ **include** file="b.jsp" %>

这里的file值，**必须使用相对路径**，若以/开头，则表示当前WEB应用的根目录。

与之对应为“**动态包含**”，使用<jsp:include page=”xx”>，向被包含页面传递参数，**编译出两个.class文件**。

例如：

**<jsp:include page="...">  
<jsp:param name="..." value="..." />**

**</jsp:include>**

* **taglib：**

声明该页面所使用的标签库 ，并引用标签库，指定标签的前缀。

例如：使用fmt、jstl等标签库时，需使用taglib引入、分配别名

* **动标识：**
* **<jsp:include>**

和include指令不同，该指令生成的是**两个Servlet文件**，然后将目标文件包含到当前的JSP页面中，称为：**动态引入**

**eg：**

<**jsp:include** page="b.jsp"></**jsp:include**>

* **<jsp:forward>**

将请求转发到其他的Web资源

eg：

<**jsp:forward** page="/include/b.jsp"></**jsp:forward**>

* **<jsp:param>**

传递参数

* **示例代码：**使用**<jsp:forward>**和**<jsp:param>给b.jsp传递参数**

**当前JSP页面：**

<**jsp:forward** page="/include/b.jsp">  
 <**jsp:param** value="chris" name="username"/>  
</**jsp:forward**>  
**或者**  
<**jsp:include** page="/include/b.jsp">  
 <**jsp:param** value="chris" name="username"/>  
</**jsp:include**>

**b.jsp页面**：获取传递来的参数

request.**getParameter**("username")

1. **JSP标准标签库：（JSTL）**

* **<c:forEach>、<c:forTokens>**

**用于循环遍历**

**<c:forEach>：更加通用，遍历集合对象，eg：**

**<c:forEach items**="遍历**对象**" **begin**="开始元素" **end**="最后元素" **step**=步长 **var**="当前变量">

**<c:forTokens>：将指定字符串分割为数组，然后遍历**

* **JSP的标签使用：混合EL表达语言**

<jsp:text>

Box Perimeter is: ${2\*box.width + 2\*box.height}

</jsp:text>



1. **EL表达语言**

* **基本概念：**
* EL（Expression Language），为了简化JSP编写，以${}为标志。需要结合JSTL标签库使用（<c:forEach items=“”>、<c:if test=“”>）、…

**例如：**${sessionScope.user.name}，从session范围中获取user的name，等同于如下JSP Script中编写的代码：

User user = (User)session.getAttribute(“user”);

String name = user.getName();

* **EL的.和[]运算符：**

这两者是一样的，可以混用。

eg：

(1)下面两意思一样

${sessionScope.user.name}、${sessionScope.user[“name”]}

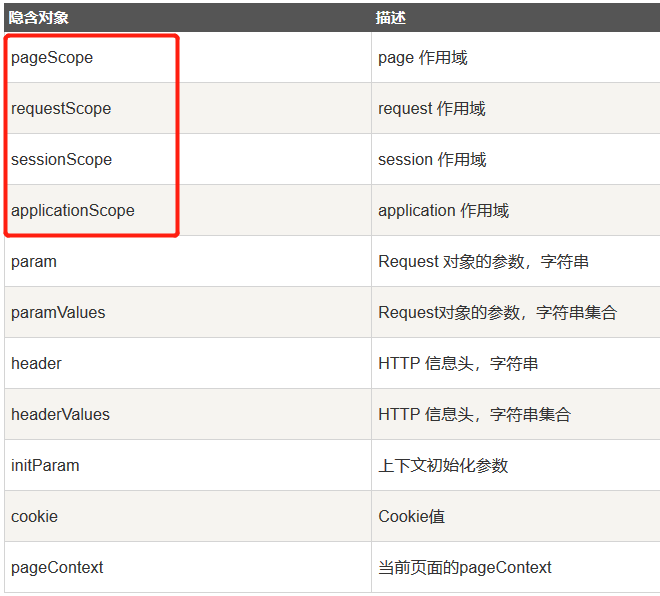
(2)存取特殊字符

${user[“my-name”]}

* **EL隐含对象：**

在未指定范围时，默认先从page范围开始查找，找不到则依次遍历request、session、application范围。若都未找到，jsp页面不会显示null，而是显示空白。

**eg：**${username}，查找、获取username变量值



**获取参数的常用方法：（如下两种表达式等价）**

1. 获取参数名：request.getParameter(String name)

${param.name}

1. 获取参数值：request.getParameterValues（String name）

${paramValues.name}

* **EL的条件运算符：**

${A ? B : C}



1. **中文乱码问题：**

* 在JSP页面上输入中文,请求页面后不出现乱码:

保证：（1）contentType="text/html; charset=UTF-8",

（2）pageEncoding="UTF-8" charset

（3）pageEncoding

这三者的编码一致, 且都支持中文.一般取值为UTF-8。

还需保证**浏览器**的显示的**字符编码**也和请求的 JSP 页面的编码一致.

* **中文**参数传输过程的编码：默认参数在传输过程中使用的编码为 **ISO-8859-1**
* **接收参数的页面处理：**
* **POST：**

只要在获取请求信息之前(调用request.getParameter 或 request.getReader前),

调用request.**setCharacterEncoding**("UTF-8") 即可.

* **GET：**

上面的方式对于 GET 无效. 需要修改 Tomcat 的 server.xml 文件才能实现.

# MVC

1. **MVC设计模式：model-view-controller，模型视图控制器**

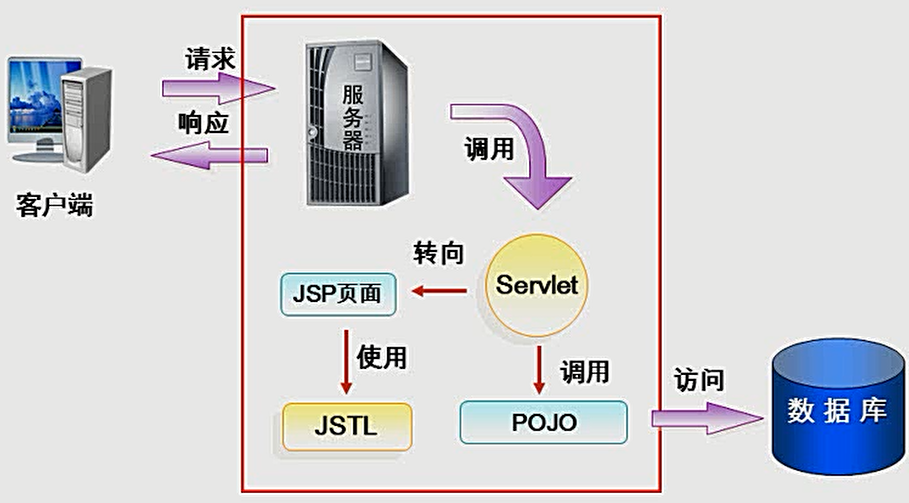
* **MVC：**

**M**：Dao（data access object），数据处理模型

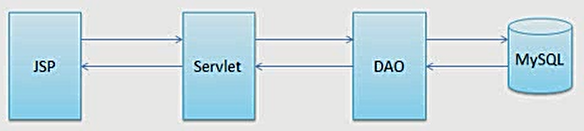
**V**：JSP页面显示

**C**：Servlet处理数据，**流程如下：**

请求->获取请求参数->调用Dao方法->返回Dao方法值->转发、重定向页面



访问的顺序如下：只能够一层一层的递进访问，不能够跨层访问



**开发时：**编写的流程，MySQL -> DAO -> Servlet -> JSP

* 关于什么时候使用**转发** 和 **重定向**：
* 使用***转发：***页面需要从request中获取数据。刷新页面之后，URL地址不会改变；
* 使用***重定向：***不需要从request中获取任何数据。**可以防止表单被重复提交(在刷新之后，URL地址已经改变，不再是原来的地址，因此不会出现表单的重复提交问题)**。
* **在MVC中常用的模块**：

1. **c3p0连接池**：

* 开源的JDBC连接池，实现数据源和JNDI的绑定。
* **相对于JDBC的好处：**

**连接池的数据库连接可以重用，避免频繁的创建连接**，减少资源浪费，提升系统运行的稳定性，可以限制数据库连接数量，统一管理连接，避免数据库连接泄露。即：建立一个缓冲池，预先放入一定数量的连接，使用时从中取出即可

* **c3p0使用配置文件配置：**文件名**c3p0-config.xml**，存放在src根目录，且**名称不能被修改**。该配置文件，配置：登录名、密码、URL、数据库驱动…等，需要结合mysql-connector-java、DBUtils使用。

1. **mysql-connector-java**：

开源的数据库驱动

1. **DBUtils：**Apache提供的数据库操作，类似于Spring的JDBCTemplate，将JDBCD进行封装，简化JDBC的编码量，提供：创建连接、查询、释放资源等，可以替换hibernate。

* **DBUtils使用简介：**
* **QueryRunner**：**核心部分**，可以使用SQL语句进行查询，通常结合**ResultSetHandler**一起使用（结果集封装器）。

**DBUtils使用示例：**

public class JDBCUtils {  
  **//创建数据源，并初始化时，dataSource只能被初始化一次（静态代码块）**  
 private static DataSource *dataSource* = null;  
 static{

**//c3p0Config：是c3p0-config.xml中配置的**  
 *dataSource* = new ComboPooledDataSource("**c3p0Config**");  
 }  
  **/\* 连接函数\*/**  
 public static Connection connection() {  
 try {  
 return *dataSource*.getConnection();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 System.*out*.println("connection ERROR !");  
 return null;  
 }  
 }  
 **/\* 释放连接对象，不是关闭数据库连接，将该对象状态变更为可用 \*/**  
 public static void releaseConnection(Connection connection){  
 if(connection != null){  
 try {  
 connection.close();  
 } catch (SQLException e) {  
 e.printStackTrace();  
 }  
 }  
 }

}

**c3p0-config.xml配置文件中：**

<named-config name="**c3p0Config**">

…

</named-config>

* **ResultSetHandler**：**结果集封装器**，常用的接口如下：

1. ArrayHandler()：把结果集的第一行数据转成数组，

存入Object[]。

1. ArrayListHandler()：结果集的每一行数据转成一个数组，

存入List中 。

1. BeanHandler(Class<T> type)：结果集的第一行数据，

封装到JavaBean中。

1. BeanListHandler(Class<T> type)：结果集的每行数据，

封装到JavaBean中，并存到List中。

1. ColumnListHandler(String columnName)：将结果集的某一列数据存放到List中。
2. MapHandler()：
3. MapListHandler()：
4. KeyHandler()：
5. ScalarHandler(int columnIndex)：用于保存只有一行一列的结果集。

**示例代码：**

**//查找数据 ：… 表示可以传入任意数量的参数args**

public List<Map<String, Object>> getAll(String sql, Object ... args){  
 Connection connection = null;  
 try {  
 connection = JDBCUtils.*connection*();  
 List<Map<String, Object>> users = queryRunner.**query**(connection, sql, new **MapListHandler**(), args);  
 return users；  
 } catch (SQLException e) {  
 e.printStackTrace();  
 } finally{  
 JDBCUtils.*releaseConnection*(connection);  
 }  
 return null;  
 }

**//测试删除：**

public void testDelete() throws SQLException {

QueryRunner queryRunner = new QueryRunner(JDBCUtils.getDataSource());

String sql = "delete from users where id = ?";

queryRunner.**update**(sql, 3);

}

* **DBUtils**：工具类，提供加载驱动、关闭连接、事务提交、回滚等工具类。

* **资源文件的加载，Web应用启动自动调用初始化Servlet设置属性：**

**1、需要Web应用启动时，自动调用的Servlet：**

/\*\*  
 **\* 在web.xml中进行设置，在WEB应用启动时，自动调用该Servlet，  
 \* 并执行init方法，读取DAOType.properties文件中的type参数**

\*/  
public class InitServlet extends HttpServlet {  
 private static final long *serialVersionUID* = 1L;  
  
 @Override  
 public void init(){  
 **/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 设定默认值：jdbc \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/**  
 EmployeeInfoDAOFactory.*getFactoryInstance*().setType("jdbc");  
  
  **/\*\*\* 读取 properties文件的type参数，来设置CustomerDAOFactory对象 \*\*\*\*/**  
 InputStream **in** = **getServletContext().getResourceAsStream**("/**WEB-INF**/classes/**DAOType.properties**");  
 Properties properties = new **Properties**();  
 try {  
 properties.**load**(in);  
  **//获取 DAOType.properties 设置的 type 属性值**  
 String type = properties.**getProperty**("type");  
  **//赋给了 EmployeeInfoDAOFactory 的 type 属性值**  
 EmployeeInfoDAOFactory.*getFactoryInstance*().setType(type);  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
}

**2、web.xml中的配置：**

**<!-- 自动调用InitServlet -->**  
 <servlet>  
 <servlet-name>InitServlet</servlet-name>  
 <servlet-class>Servlet.**InitServlet**</servlet-class>  
 <**load-on-startup**>**1**</load-on-startup>  
 </servlet>

**3、servlet读取的DAOType.properties配置文件：**

#type=xml  
type=jdbc

# Cookie

1. **Cookie简介**

* HTTP是一种无状态协议，WEB服务器本身不能够识别哪些请求是同一个浏览器发出的，因此浏览器的每一次请求都是孤立的。

* **WEB应用中的会话：**web服务器和浏览器在通信过程中产生的状态信息，使用会话状态，web服务器能够把属于同一会话中的请求和响应关联起来。

会话的标识号：SessionID

* Servlet中有两种机制，进行**会话跟踪**：

1. Cookie：数据保存在客户浏览器上，一般不会超过4k，20个cookie
2. Session：数据保存在服务器上

* **Cookie机制：**

1. 采用客户端保存HTTP状态，在浏览器访问Web服务器时，Web服务器在HTTP响应消息头中，附加**Cookie文件**传送给浏览器
2. 浏览器保存了某个Cookie文件，在每一次访问该Web服务器时，都会在HTTP请求头中，将该Cookie回传给Web服务器。
3. Cookie只能表示一种信息：

至少包含该信息的名称（Name）、数值（Value）

1. Web服务器、浏览器，都可以发送、存储多个Cookie

* **会话Cookie和持久Cookie：**

1. 不设置过期时间，则表示这个Cookie的生命周期为浏览器的会话期间，只要关闭浏览器，这个Cookie就消失；
2. 设置了过期时间，浏览器会把Cookie保存在本地硬盘上，关闭浏览器后，在设定时间内，这些cookie依然存在；

* **浏览器怎么查看当前页面的Cookie信息：**

**有两种方法：**



* **基本应用：**

1. **自动登陆：**在规定时间内存储Cookie的信息，关闭浏览器后，在规定时间内，还是可以不用输入信息，可以登陆。
2. **浏览过的信息提示：**在页面跳转时，将请求参数保存为Cookie，在返回原来页面时，将这些参数进行封装，保存为cookie并返回给原来的页面。

* **Cookie应用示例代码：**

<%

**//获取其他页面传过来的Cookie数据**

Cookie[] cookies = request.**getCookies**();

**//设置Cookie保存数据的最大时间（单位：秒）：0S，即：立即返回**

Cookie tempCookie = null；

tempCookie.**setMaxAge**(0);

**//创建新的Cookie**  
Cookie cookie = new **Cookie**("cookieName", bookName);

**//设置这个Cookie的作用范围：仅在当前目录和其子目录**  
cookie.**setPath**(request.getContextPath());  
**//设置Cookie值：添加到响应中，可用于设置多个Cookie（返回给客户端）**

response.**addCookie**(cookie);  
%>

# Filter过滤器

1. **基本概念**

* **Filter**：是javaWeb的一个重要组件，可以对发送到Servlet或请求响应进行拦截 。使用时需要继承Filetr接口，并且在web.xml中进行配置---和servlet配置基本一样。

**web.xml配置示例：**

<filter>  
 <filter-name>secondFilter</filter-name>  
 <filter-class>javaweb.SecondFilter</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>secondFilter</filter-name>  
 <url-pattern>/test.jsp</url-pattern>  
</filter-mapping>

* **Filter的API说明：**

1. public void **init**(FilterConfig filterConfig){}:

初始化，类似于Servlet中的初始化函数，当Filter对象被Servlet容器加载到当前Web应用中被初始化，且仅仅被初始化一次。

**参数说明：**

* FilterConfig：和 Servlet中的ServletConfig类似；

**使用说明：**

可以在Web.xml中配置该Filter对象的初始化参数，示例代码如下：

<filter>  
 <filter-name>helloFilter</filter-name>  
 <filter-class>javaweb.HelloFilter</filter-class>  
 **<init-param>**  
 <param-name>name</param-name>  
 <param-value>root</param-value>  
 **</init-param>**  
</filter>  
<filter-mapping>  
 <filter-name>helloFilter</filter-name>  
 <url-pattern>/test.jsp</url-pattern>  
</filter-mapping>

1. public void **doFilter**(ServletRequest req, ServletResponse response, FilterChain chain) {}:

所有需要处理的逻辑代码都在这个函数里面，每次调用拦截时，都会执行该函数。

**参数说明：**

* **FilterChain**：是Filter链，多个Filter可以构成一条Filter链；**Filter链中的Filter对象执行顺序在web.xml中配置（越靠前优先被调用）**；FilterChain对应的方法说明如下：
* **doFilter**(ServletRequest req，ServletResponse resp)：

将请求传给Filter链的下一个Filter对象，若该Filter是最后一个，则将请求发送给目标Servlet或JSP。

1. public void **destory**(){}:

销毁函数，释放该Filter所占用的资源；

1. **Filter使用的示例代码：**

public class UserNameFilter implements Filter {

private **FilterConfig** filterConfig;  
   
 public void init(FilterConfig fConfig) throws ServletException {

**//初始化Filter对象**  
 this.filterConfig = fConfig;  
 }

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {  
 **//获取web.xml中的配置参数**  
 String initUser = filterConfig.getInitParameter("username");  
 String username = request.getParameter("username");  
  **//对比参数是否相同**  
 if(!initUser.equals(username)){  
 request.setAttribute("message", "用户名不正确");  
 request.getRequestDispatcher("/login.jsp").forward(request, response);  
 return;  
 }  
  **//参数相同：放行，访问下一个Filter或JSP**  
 chain.**doFilter**(request, response);  
 }

public void destroy() {  
 }  
}

* **在web.xml中配置访问Filter对象的方法：**

使用<dispatcher>进行配置：指定过滤器所拦截的资源被Servlet容器调用的方式，**包含：**REQUEST、INCLUDE、FORWARD、ERROR，**默认为REQUEST**。仅在请求在满足该元素的配置的参数时可以访问对应的Filter对象。

1. **REQUEST（默认方式）**：

用户直接访问页面时，Web容器会调用该过滤器，使用GET、POST进行访问；

1. **FORWARD**：

仅当使用：RequestDispatcher().forward()、

<jsp:forward page=”/xxx”>，

访问时，web容器会调用该过滤器；

1. **INCLUDE**：

仅当使用：RequestDispatcher().include()、

<jsp:include file=”/xxx”>

访问时，web容器调用该过滤器；

1. **ERROR**：

当出现异常时，该资源使用声明式异常处理机制调用该过滤器。在web.xml中使用<error-page>进行配置、声明；

**示例代码：**

<error-page>  
 <exception-type>java.lang.ArithmeticException</exception-type>  
 <location>/test.jsp</location>  
</error-page>

1. **示例代码：**

jsp中：

<**jsp:forward** page="/test.jsp"></**jsp:forward**>

web.xml中：

<filter>  
 <filter-name>secondFilter</filter-name>  
 <filter-class>javaweb.SecondFilter</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>secondFilter</filter-name>  
 <url-pattern>/test.jsp</url-pattern>  
 <dispatcher>**REQUEST**</dispatcher>  
 <dispatcher>**FORWARD**</dispatcher>  
 <dispatcher>**INCLUDE**</dispatcher>  
 <dispatcher>**ERROR**</dispatcher>  
</filter-mapping>

**<error-page>**  
 <exception-type>java.lang.ArithmeticException</exception-type>  
 <location>/test.jsp</location>  
**</error-page>**

* **HttpServletRequestWrapper**
* **概念：**Servlet API 中提供一个 HttpServletRequestWrapper 类来包装原始的request 对象,HttpServletRequestWrapper类实现 HttpServletRequest 接口中的所有方法, 这些方法的内部实现都是仅仅调用了一下所包装的的 request 对象的对应方法。
* **作用：**用于修改HttpServletRequest的某个方法，开发人员可继承该类，然后对相应的方法进行重写，以达到修改特定的HttpServletRequest方法的目的。（本质上：都是调用原ServletRequest）
* **使用方式：**

需要开发者继承该类，然后进行对应的方法重写。在自己编写的Filter中实例化这个新编写的类、调用该类重写的方法，使用FilterChain.doFilter(request, response)转到对应页面即可。

* **示例代码**：

使用Filter、HttpServletRequestWrapper实现浏览器提交字符的过滤系统；

1. 自己编写HttpFilter，因为**原有的Filter使用的是ServletRequest、ServletResponse，而Http协议需要使用HttpServletRequest、HttpServletReesponse；**

**文件：HttpFilter**

public abstract class HttpFilter **implements Filter** {  
 private FilterConfig filterConfig;  
  
 @Override  
 /\*\*  
 \* 最好不要直接覆盖init方法，使用下面的init来初始化  
 \*/  
 public void init(FilterConfig filterConfig) throws ServletException {  
 this.filterConfig = filterConfig;  
 init();  
 }  
  
 private void init(){}

@Override  
 /\*\*  
  **\* 原来Filter默认使用ServletRequest、ServletResponse，  
 \* Http需要使用的是HttpServletRequest、HttpServletResponse** \* 需要进行相应的修改  
 \*/  
 public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain)  
 throws IOException, ServletException {  
 HttpServletRequest req = (HttpServletRequest) servletRequest;  
 HttpServletResponse resp = (HttpServletResponse) servletResponse;  
 doFilter(req, resp, filterChain);  
 }

**/\* 该抽象方法：\*/**

**/\* 需要在实例化对象后，自己实现，将需要重写的功能写在这里面即可 \*/**  
 public abstract void doFilter(HttpServletRequest req, HttpServletResponse resp, FilterChain filterChain)  
 throws IOException, ServletException;  
 @Override  
 public void destroy() { }  
}

1. **继承HttpServletRequestWrapper，重写getparameter方法**，即：封装自己的类MyHttpServletRequest。实现Servlet容器（Tomcat）可调用的getParameter方法：

**文件：MyHttpServletRequest**

public class MyHttpServletRequest extends **HttpServletRequestWrapper** {  
 public MyHttpServletRequest(HttpServletRequest request) {  
 super(request);  
 }  
 @Override  
  **//重写的API，对request发送过来的特定字符进行过滤、修改，然后返回**  
 public String getParameter(String var){

**//======核心代码======**  
 String **res** = **super.getParameter**(var);  
 if(res != null && res.contains("fuck")){  
 //将fuck字段替换成\*\*\*\*，然后返回  
 res = res.replace("fuck", "\*\*\*\*" );  
 }  
 **return res**;  
 }  
}

1. 编写Filter的Servlet，获取浏览器的请求数据，并且实例化MyHttpServletRequest，调用重写的getParameter，实现字符过滤，然后使用filterChain.doFilter(req,resp)执行下一步；

**文件：ContentFilter（使用该Filter，要在web.xml中配置）**

public class ContentFilter extends **HttpFilter** {  
  
 @Override  
 public void doFilter(HttpServletRequest req, HttpServletResponse resp, FilterChain filterChain)  
 throws IOException, ServletException {  
 **/\*\*\*\*\* 原来默认使用的方法 \*\*\*\*\*\*/** //1. 获取请求 content 参数的值:  
 String content = req.getParameter("content");  
 //2. 把其中 fuck, shit 等字符串替换换为 \*\*\*\*  
 if(content.contains(" fuck ")){ }  
  
 **/\*\*\*现在使用的方法：使用自己修改的HttpServletRequest \*\*\*\*/** **//装饰目前的 HttpServletRequest 对象: 装饰其 getParameter 方法, 而其他方法还和其实现相同.  
 //创建一个类, 该类实现 HttpServletRequest 接口, 把当前 doFilter 中的 request 传入到该类中, 作为  
 //其成员变量, 使用该成员变量去实现接口的全部方法.**

**//======核心代码======使用自己修改的HttpServletRequest**  
 HttpServletRequest **request** = new **MyHttpServletRequest**(**req**);  
  
 **//3. 转到目标页面**  
 **filterChain.doFilter(request, resp)**;  
 }

1. 使用Filter必须在web.xml中进行配置（和servlet差不多）

<filter>  
 <filter-name>ContentFilter</filter-name>  
 <filter-class>Filter.**ContentFilter**</filter-class>  
</filter>  
<filter-mapping>  
 <filter-name>ContentFilter</filter-name>  
 <url-pattern>/contentFilter/contentFilterResult.jsp</url-pattern>  
</filter-mapping>

* HttpServletResponseWrapper

和HttpServletRequestWrapper类似。

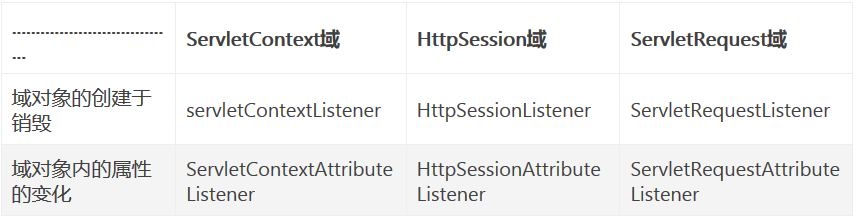
# Listener监听器

1. **基本概念：**

* Listener监听器，用于监听某个对象的状态的组件；

* **监听器的几个概念：**
* 事件源：被监听的对象：session、request、servletContext；
* 监听器：被监听对象状态发生都会触发监听器；
* 注册监听器：将监听器和事件源进行绑定；
* 响应行为：监听器检测到对象状态发生改变时所涉及的功能代码；
* **监听器分类：**按照监听对象可分为3类：

ServletRequest域、ServletContext域、HttpSession域



1. **ServletRequest监听器的作用：**

ServletRequestListener请求监听：每当页面发出请求时启动，并且在请求结束后自动销毁，作用域为：只能对单个Request有效；

1. **ServletContext监听器的作用：（常用）**

ServletContextListener监听器，**随web应用启动而启动**，并且只能初始化一次，随web应用停止而销毁，作用域为：整个Web应用；

**作用：**

① : 用于初始化：初始化对象、数据,加载数据驱动,c3p0连接池的初始化；  
② : 加载初始化的配置文件.properties,如Spring的配置文件；  
③ : 任务调度---定时器---Timer/TimerTask；

1. **HttpSession监听器的作用：**

HttpSessionListener监听器：会话监听，当Request创建session时启动，当session失效、过期时销毁（在浏览器打开—>关闭期间有效）。作用域为：具体的session对象

**作用：**

可用于统计在线人数、自定义session扫描器 ；

1. **ServletRequest域的监听器使用：**

.java文件中：

public class MyServletRequestListener implements ServletRequestListener{  
 @Override  
 public void requestDestroyed(ServletRequestEvent sre) {  
 System.*out*.println("销毁");  
 }  
 @Override  
 public void requestInitialized(ServletRequestEvent sre) {  
 System.*out*.println("创建");  
 }  
}

web.xml中：

<listener>  
 <listener-class>com.MyServletRequestListener </listener-class>  
</listener>

**5、ServletContext域的监听器使用：**

.java文件中：

public class MyServletContextListener implements ServletContextListener{  
 @Override  
 public void contextInitialized(ServletContextEvent sce) {  
 System.*out*.println("context创建了....");  
 }  
 @Override

**// ServletContextEvent：其getServletContext()方法，获取servletContext**  
 public void contextDestroyed(ServletContextEvent sce) {  
 System.*out*.println("context销毁了");  
 }  
}

web.xml中：

<listener>  
 <listener-class>com.MyServletContextListener </listener-class>  
</listener>

**6、HttpSession域的监听器使用：**

.java文件中：

public class MySessionListener implements HttpSessionListener{  
 @Override  
 public void sessionCreated(HttpSessionEvent se) {  
 System.*out*.println("sessionId : " + se.getSession().getId())  
 }  
 @Override  
 public void sessionDestroyed(HttpSessionEvent se)   
 }  
}

web.xml中：

<listener>  
 <listener-class>com.MySessionListener</listener-class>  
</listener>

* **属性操作：上述3个领域的通用方法**：

1、setAttribute(name,value) :添加属性/修改属性的监听器的方法

2、getAttribute(name) ：获取属性的监听方法

3、removeAttribute(name) ：删除属性的监听器的方法

* **session中的对象的几种状态：**

1. 绑定状态 : 就一个对象被放到session域中 ；
2. 解绑状态 : 被绑定的对象从session域中移除了 ；
3. 钝化状态 : 是将session内存中的对象持久化( 序列化)到磁盘 ；
4. 活化状态 : 就是将磁盘上的对象再次恢复到session内存中

* **绑定与解绑监听器使用：**

不需要在web.xml中进行配置。

**//需要被绑定的类**

public class Person implements HttpSessionBindingListener{  
 private String name;  
 public void setName(String name) {  
 this.name = name;  
 }  
 @Override  
 **//绑定**：被绑定时自动调用  
 public void valueBound(HttpSessionBindingEvent event) {  
 System.*out*.println("person被绑定了");  
 Person per = (Person)event.getValue();  
 System.*out*.println(per.getName());  
 }  
 @Override  
 **//解除绑定**：解除绑定时，自动调用  
 public void valueUnbound(HttpSessionBindingEvent event) {  
 System.*out*.println("person被解绑了");  
 }  
}  
**//测试需要被绑定的类**  
public class TestPersonBindingServlet extends HttpServlet {  
 protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {

**//获取session对象**  
 HttpSession session = request.getSession();  
 Person p = new Person();  
 p.setName("思思");  
 **//将person对象绑到session中（实现绑定）**  
 session.setAttribute("person", p);  
 **//将person对象从session中解绑**  
 session.removeAttribute("person");  
 }  
 protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
 doGet(request, response);  
 }  
}

* **活化与钝化监听器：**

1. 活化：从磁盘中读取session对象；

钝化：向磁盘中存入session对象；

1. 使用：通过xxx.xml文件配置，特定对象的钝化时间：该对象多长时间不使用时被钝化。

xxx.xml文件中：

<?xml version="1.0" encoding="UTF-8"?>  
<Context>  
 <!-- **maxIdleSwap:session中的对象多长时间不使用就钝化** -->  
 <!-- **directory:钝化后的对象的文件写到磁盘的哪个目录下 配置钝化的对象文件在 work/catalina/localhost/钝化文件** -->  
 <Manager className="org.apach.catalina.session.PersistentManager" **maxIdleSwap**="1">  
 <Store className="org. apach.catalina.session.FileStore" **directory**="tiantian" />  
 </Manager>  
</Context>

# 文件上传、下载

1. **文件上传**
2. **环境搭建：**

commons-fileupload-1.2.1.jar

commons-io-2.0.jar

1. **服务端：**

* 不能再使用 request.getParameter() 等方式获取请求信息。 因为请求的编码方式已经改为 **multipart/form-data——以二进制的方式来提交请求信息.**

（可以使用输入流的方式来获取. 但不建议这样做.）

* 具体使用 commons-fileupload 组件来完成文件的上传操作.

1. **form表达属性：** enctype

* **application/x-www-form-urlencoded（默认值）**：该编码方案使用有限的字符集，当使用了非字母和数字时，必须用”%HH”代替(H 代表十六进制数字)。对于大容量的二进制数据或包含非 ASCII 字符的文本来说，这种编码不能满足要求。
* **multipart/form-data**：表示表单以二进制传输数据；（文件传采用该编码方式）

1. **文件下载**

# 国际化

1. **基础**

* **国际化和本地化:**

1. **国际化：**软件开发时，让它能支持多个国家和地区的本地化应用。使得应用软件能够适应多个地区的语言和文化风俗习惯
2. **本地化：**一个软件在某个国家或地区使用时，采用该国家或地区的语言，数字，货币，日期等习惯。
3. **本地敏感数据:** 随用户区域信息而变化的数据称为本地信息敏感数据。例如数字，货币, 日期，时间等数据

* **模式字符串：**

使用{}作为占位符，{}中间的数字代表第几个参数，需要结合MessageFormat使用。类似于脚本文件中传入键盘参数的写法；

**eg：**String res = “Date:**{0}**, salary:**{1}**”;

* **相关的 API:**

1. **DateFormat**（常用）和 **SimpleDateFormat**：格式化日期；
2. **NumberFormat**： 格式化数字🡪数字字符串、货币字符串；
3. **MessageFormat**：格式化模式字符串；
4. **ResourceBundle**：根据特定国家、地区，来获取不同.properties资源文件中的数据；
5. **Locale**：获取当前所在的国家、地区，或者式设定的所在国家、地区；

**eg：Locale locale = Locale.US; //美国**

* **关于国际化资源文件:i18n.properties（简写：internationalization）**

1. **使用国际化资源文件的要求：**

* **必须有的文件：**基名.properties 文件；

基名\_语言代码\_国家代码.properties 文件；

* **相同的 基名 的资源文件必须有相同的 key；**
* 若资源文件中的参数值涉及中文，需要先将中文转换成ASCII码，再使用（否则会出现乱码）；

1. **properties 文件格式**

基名\_语言代码\_国家代码.properties

**eg：**i18n\_zh\_CN.properties

* **WEB 的国际化：在JSP页面使用**

1. 可以使用request.getLocale()获取 Locale 对象
2. 可以使用JSTL的 fmt 标签完成的国际化.框架也有自己的标签
3. **fmt 标签：**快速实现JSP页面的国际化，代替.java文件中的ResourceBundle、DateFormate、NumberFormat、MessageFormat等等，可以参考 standard-examples.war 中的例子.
4. **需要使用的jar包：jstl.jar：标签<c:**

**standar.jar：标签<fmt:**

**使用标签时，必须在对应页面导入对应标签的uri路径**

* **示例代码：**

1. **JSP中实现国际化：页面语言的变更（常用的方式）**

**<!— 要显示的数据 -->**

<%  
 Date date = new Date();  
 request.setAttribute("date", date);  
 request.setAttribute("salary", 12345.67);  
%>  
**<!— 将请求中的语言参数，设置到session属性 -->**  
<%  
 String code = **request.getParameter**("code"); **//获取页面的请求参数**  
  
 if(code != null){  
 if("en".equals(code)){  
 **session.setAttribute**("locale", Locale.*US*); **//设置session域的属性**  
 }else if("zh".equals(code)){  
 session.setAttribute("locale", Locale.*CHINA*);  
 }  
 }  
%>  
**<!— 使用fmt来显示数据 -->**  
<**c:if** test="${**sessionScope.locale** != null }"> **//获取session设置的属性**  
 **<c:out** value="${sessionScope.locale }"></**c:out**> //显示语言类型  
 <**fmt:setLocale** value="${sessionScope.locale }"/>  
</**c:if**>  
  
<**fmt:setBundle** basename="i18n"/>  
<**fmt:message** key="date"></**fmt:message**>:  
<**fmt:formatDate** value="${date }" dateStyle="FULL"/>,  
<**fmt:message** key="salary"></**fmt:message**>:  
<**fmt:formatNumber** value="${salary }" type="currency"></**fmt:formatNumber**>  
<br><br>  
**<!— 切换页面语言 -->**  
<a href="index.jsp?code=en">English</a>  
<a href="index.jsp?code=zh">中文</a>

1. **.java文件中实现国际化（不常用）**

public void test(){  
 **//1、获取国家、地区：Locale**  
 Locale locale = Locale.*US*;  
 System.*out*.println("locale = " + locale);  
  
 **//2、使用ResourceBundle：根据特定的国家、地区，获取.properties文件**  
 ResourceBundle resourceBundle = ResourceBundle.*getBundle*("i18n", locale);  
 String dateLable = resourceBundle.getString("date");  
 String salaryLable = resourceBundle.getString("salary");  
  
  **//3、模式字符串：数据显示的格式，使用{}占位符，0,1,2,3表示参数的位置**  
 String displayFunction = "{0}:{1} , {2}:{3}";  
 Date date = new Date(); //要显示的时间数据：获取时间  
 Double salary = 123.45; //要显示的工资数据  
  
  **//4、DateFormat：格式化日期**  
 DateFormat dateFormat = DateFormat.*getDateInstance*(DateFormat.*LONG*, locale);  
 String dateStr = dateFormat.***format***(date); //格式化日期  
  
 **//5、NumberFormat：格式化数字-->数字字符串、货币字符串**  
 NumberFormat numberFormat = NumberFormat.***getCurrencyInstance***(locale);

//格式化为货币格式  
 String salaryStr = numberFormat.***format***(salary); //格式化工资  
  
  **//6、MessageFormat:格式化模式字符串**  
 String result = MessageFormat.***format***(displayFunction  
 dateLable, dateStr,  
 salaryLable, salaryStr);  
 System.*out*.println(result);  
}



# 补充内容

1. **JSP中<% %>、<%! %>、<%= %> 、<%@ %>的区别**

* **<% %>：**用于编写脚本片段，其中的内容会被Tomcat服务器翻译为.java中的Servlet的Service()方法中，因此<% %>只能定义局部变量、调用方法，不能定义方法；
* **<%! %>：**用于声明，其中写的内容会被Tomcat服务器翻译成Servlet类，因此<%! %>可以声明方法、属性、全局变量；
* **<%= %>：**JSP表达式，用于将已经声明的变量、表达式输出到网页上；
* **<%@ %>：**页面属性的配置；
* **示例代码:**
* **<% %>：**

<body>

<% for (int i=0;i<3;i++) {%>

out.print(i\*2);

<%} %>

</body>

结果： out.print(i\*2); out.print(i\*2); out.print(i\*2);

* **<%! %>：**

<body>

<%-- 全局变量 -->

<%! int sum=1; %>

<%-- 局部变量 -->

<% int sum=8; %>

<h3>----<% out.print(sum++); %></h3>

<h3>----<%= this.sum %></h3>

</body>

结果：----8

----1

* **<%= %>：**

<%-- 获取属性“requestKey”的值，在JSP页面输出 -->

<%= request.getAttribute("requestKey") %>

1. **JavaWeb工程阅读的步骤：**

表结构 🡪 web.xml 🡪 MVC代码 🡪 DataBase数据库结构阅读

🡪 Spring IOC、AOP 🡪 log日志代码 🡪 阅读工程全部代码

1. **JavaWeb工程编写步骤：**

设计MySQL数据库结构 🡪 编写DAO数据访问对象代码

🡪 编写Servlet 🡪 编写JSP页面

1. **获取请求参数：**

使用方法httpSrvletRequest.getPatamter(“参数名”);

1. **一个Servlet中，根据请求参数的不同，有不同的功能：（重点）**

**/\*\*  
 \* 功能：一个Servlet中，根据不同的请求参数，执行不同的功能  
 \*/**  
protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
 **//获取请求参数：即：方法名称**  
 String **methodName** = req.getParameter("method");  
 System.*out*.println("=============" + methodName);  
  
 try {  
  **//使用反射，来获取当前所在类的方法**  
 Method method = **getClass().getMethod**(methodName,  
 HttpServletRequest.class, HttpServletResponse.class);  
  **//执行方法**  
 method.**invoke**(this, req, resp);  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
}

**//功能一：登陆**

public void login(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {}

**//功能二：注销**

public void logout(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {}

1. **常见状态码：**

|  |  |  |
| --- | --- | --- |
| **状态码** | **类别** | **原因** |
| 1xx | Informational（信息性状态码） | 接受的请求正在处理 |
| 2xx | Success（成功状态码） | 请求正常处理完毕 |
| 3xx | Redirection（重定向） | 需要进行附加操作以完成请求 |
| 4xx | Client error（客户端错误） | 客户端请求出错，服务器无法处理请求 |
| 5xx | Server Error（服务器错误） | 服务器处理请求出错 |

1. 200：ok，从客户端发送给服务器的请求被正常处理、返回；
2. 302：found，临时重定向redirect；
3. 304：服务器资源未变化,简单地说：服务饭返回304状态码时，说明本地的缓存文件是最新的，客户端直接访问本地中的该缓存文件，可以节省带宽；
4. 400：bad request，请求报文中存在语法错误；
5. 403：forbidden，服务器拒绝该访问；
6. 404：Not found，服务器无法找到资源；
7. 500：inter server error，服务器执行请求时发生错误；
8. 502：网关执行请求时，上游服务器接收到无效响应；
9. 503：server unavailable，服务器停机，无法处理请求；
10. 504：网关执行请求时，无法从上游服务器接收到响应；