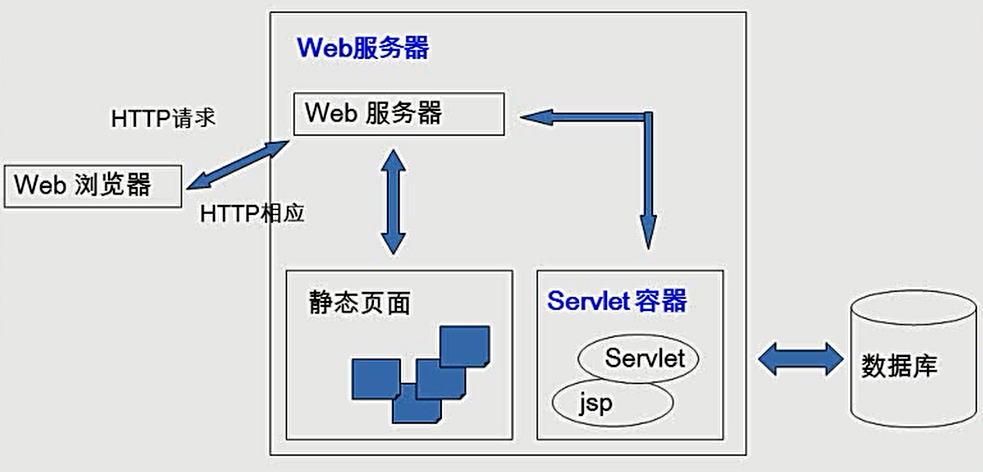
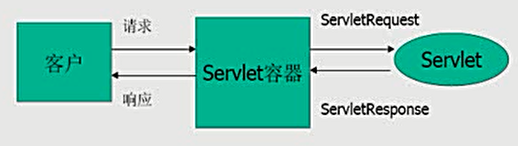
# JavaWeb

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，获取对应的参数值（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**。该对象包含于：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。

**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，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，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>

**.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方法进行相应的除磷。

* **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.

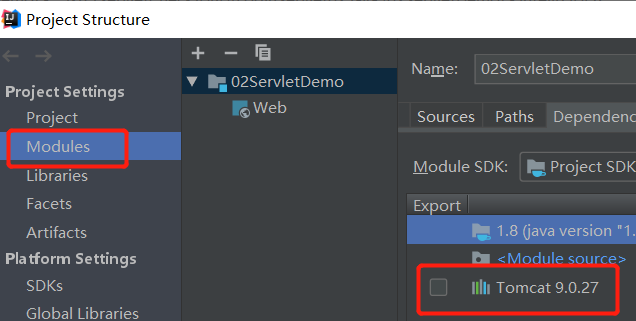
<servlet-mapping>  
 <servlet-name>login</servlet-name>

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

</servlet-mapping>

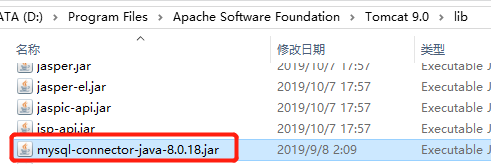
1. **关于Tomcat使用MySQL数据库的注意事项：**

* **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

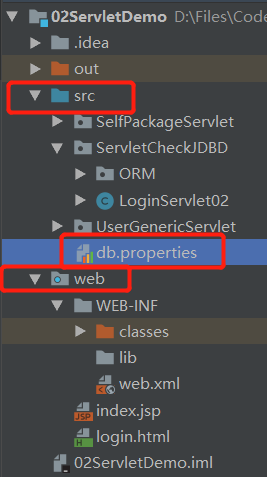


* **关于资源文件.properties的使用：**

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，**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





# 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得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：**

**本质上，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。

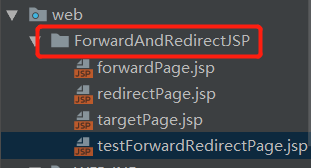
**intelliJ IDEA编译.jsp页面成.java的Servlet文件存放位置：**

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**

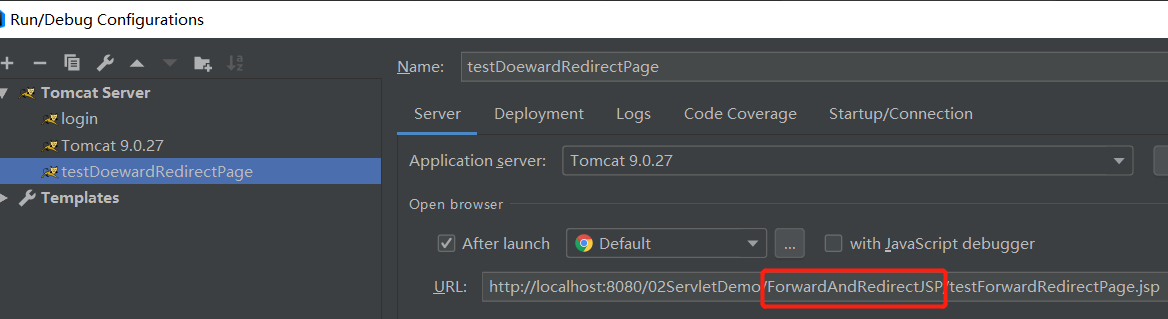
1. **JSP文件的存放位置：**

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，默认打开这个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



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**

%>

</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：**
* 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/servlet/

* 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);

* **重定向：**
* 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");

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**，将其他JSP页面完全导入当前JSP，并**生成一个Servlet**。

**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。

* **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

* **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**



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**

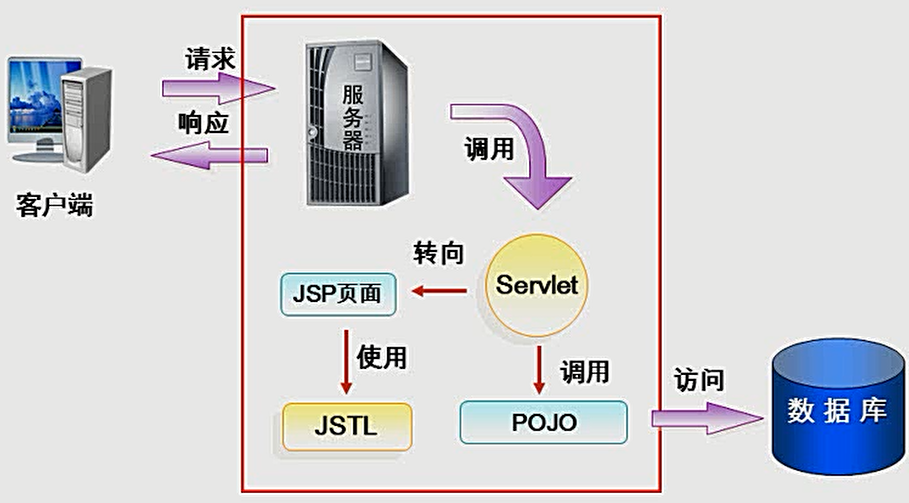
* **MVC：**

**M**：Dao（data access object），数据处理模型

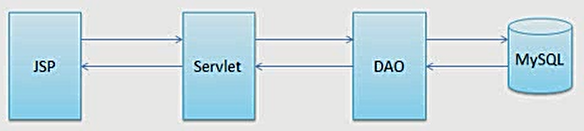
**V**：JSP页面显示

**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：
* 页面需要从request中获取数据时，使用***转发***
* 不需要从request中获取任何数据时，使用***重定向***，**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，

**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.xml配置文件中：**

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

…

</named-config》

* **ResultSetHandler**：结果集封装器，罗列常用的接口：

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 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
2. Session

* **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);  
%>