**Servlet**

**Servlet 与 Servlet 容器**

Servlet其实就是一个**遵循Servlet开发的java类**。Serlvet是**由服务器调用的**，运行在服务器端。

Servlet带给我们最大的作用就是能够**处理浏览器带来HTTP请求，并返回一个响应给浏览器，从而实现浏览器和服务器的交互**。

Sun公司在其API中提供了一个servlet接口，用户若想用发一个动态web资源(即开发一个Java程序向浏览器输出数据)，需要完成以下2个步骤：  
　　**1、编写一个Java类，实现servlet接口。  
　　2、把开发好的Java类部署到web服务器中。**  
　　按照一种约定俗成的称呼习惯，通常我们也把**实现了servlet接口的java程序，称之为Servlet**

Servlet 容器也叫做 Servlet 引擎，是 Web 服务器或应用程序服务器的一部分，用于在发送的请求和响应之上提供网络服务，解码基于 MIME 的请求，格式化基于 MIME 的响应。Servlet 没有 main 方法，不能独立运行，它必须被部署到 Servlet 容器中，由容器来实例化和调用 Servlet 的方法（如 doGet() 和 doPost() 方法），Servlet 容器在 Servlet 的生命周期内包容和管理 Servlet

**Servlet 容器能提供什么？**

上面我们知道了需要由 **Servlet 容器来管理和运行 Servlet** ，但是为什么要这样做呢？使用 Servlet 容器的原因有：

1. **通信支持：**利用容器提供的方法，你能轻松的**让 Servlet 与 web 服务器对话**，**而不用自己建立 serversocket 、监听某个端口、创建流等**。容器知道自己与 web 服务器之间的协议，所以你的 Servlet 不用担心 web 服务器（如Apache）和你自己的 web 代码之间的 API ，只需要考虑如何在 Servlet 中实现业务逻辑（如处理一个订单）。
2. **生命周期管理：** Servlet 容器控制着 Servlet 的生与死，它负责加载类、实例化和初始化 Servlet ，调用 Servlet 方法，以及使 Servlet 实例被垃圾回收，有了 Servlet 容器，你不需要太多的考虑资源管理。
3. **多线程支持：**容器会自动为它所接收的每个 Servlet 请求创建一个新的 java 线程。针对用户的请求，如果 Servlet 已经运行完相应的http服务方法，这个线程就会结束。这并不是说你不需要考虑线程安全性，其实你还会遇到同步问题，不过这样能使你少做很多工作。
4. **声明方式实现安全：**利用 Servlet 容器，你可以使用 xml 部署描述文件来配置和修改安全性，而不必将其硬编码写到 Servlet 类代码中。
5. **JSP支持：** Servlet容器负责将 jsp 代码翻译为真正的 java 代码。

**Servlet 容器也就是指 Web 容器，如 Tomcat、Jboss、Resin、Weblogic 等，它们对 Servlet 进行控制。**

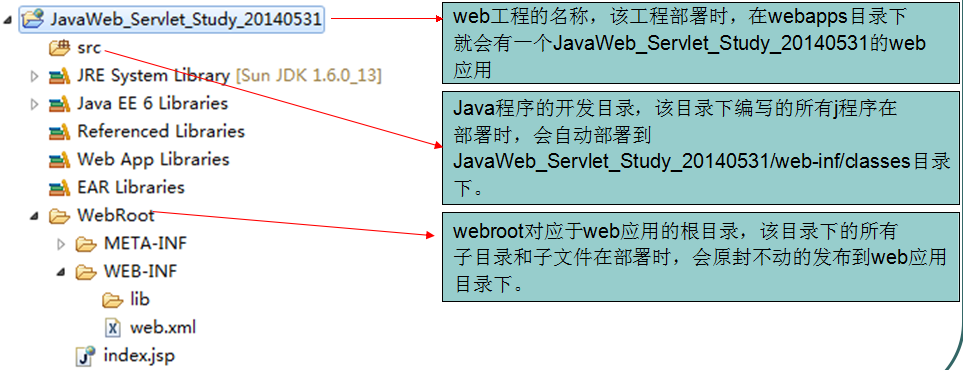
**Servlet生命周期**

1. **加载Servlet并实例化**。Servlet容器负责加载和实例化Servelt。当Servlet容器启动时，或者在容器检测到需要这个Servlet来响应第一个请求时，**创建Servlet实例**。当Servlet容器启动后，Servlet通过**类加载器来加载Servlet类**，加载完成后再new一个Servlet对象来完成实例化，放入实例池。
2. **初始化**。当Servlet被实例化后，容器将**调用init（）方法**，并传递实现**ServletConfig接口**的对象。在init（）方法中，Servlet可以部署描述符中读取配置参数，或者执行任何其他一次性活动。在Servlet的整个生命周期类，init（）方法只被调用一次。
3. **处理服务**。当浏览器**访问Servlet**的时候，Servlet **会调用service()方法处理请求。service()方法检查 HTTP 请求类型（GET、POST、PUT、DELETE 等），并在适当的时候调用 doGet()、doPost()、doPut()，doDelete() 等方法**
4. **销毁**。当Tomcat关闭时或者检测到Servlet要从Tomcat删除的时候会自动调用destroy()方法，**让该实例释放掉所占的资源**。一个Servlet如果长时间不被使用的话，也会被Tomcat自动销毁
5. **卸载**。当Servlet调用完destroy()方法后，等待垃圾回收。如果**有需要再次使用这个Servlet，会重新调用init()方法进行初始化操作**。

* 简单总结：只要访问Servlet，service()就会被调用。init()只有第一次访问Servlet的时候才会被调用。**destroy()只有在Tomcat关闭的时候才会被调用。**

**在MyEclipse中开发Servlet**

在eclipse中新建一个web project工程，eclipse会自动创建下图所示目录结构：

****

**Servlet接口实现类**

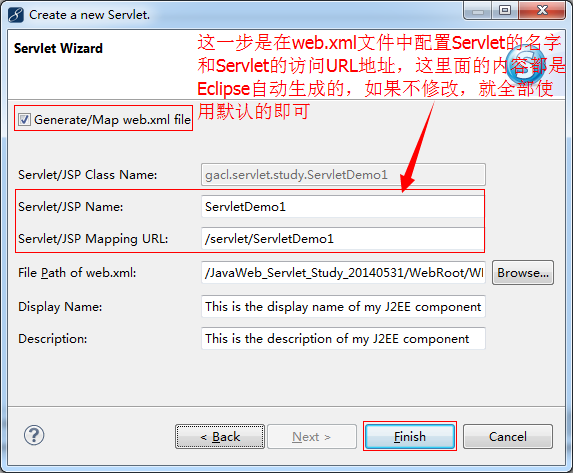
Servlet接口定义了两个默认实现类，分别为**：GenericServlet、HttpServlet。**

**HttpServlet指能够处理HTTP请求的servlet，**它在原有Servlet接口上添加了一些与HTTP协议处理方法，它比Servlet接口的功能更为强大。因此开发人员在编写Servlet时，**通常应继承这个类，而避免直接去实现Servlet接口。**

**Servlet接口有5个方法需要重写**，有init【初始化】，destroy【销毁】,service【服务】,**ServletConfig**【Servlet配置】,getServletInfo【Serlvet信息】。　HttpServlet在实现Servlet接口时，覆写了service方法，该方法体内的代码会自动判断用户的请求方式，如为GET请求，则调用HttpServlet的doGet方法，如为Post请求，则调用doPost方法。**因此，开发人员在编写Servlet时，通常只需要覆写doGet或doPost方法，而不要去覆写service方法。**

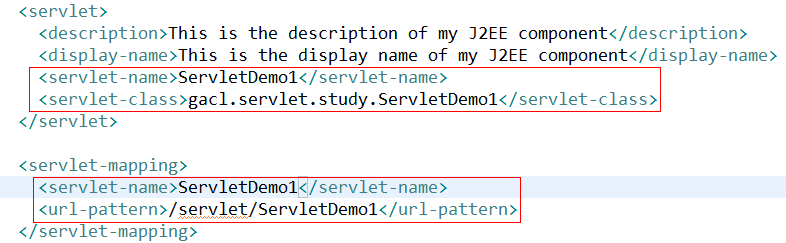
**通过Eclipse创建和编写Servlet**

**注意:若生成方法中的参数是 arg0 或则 arg1 等格式的,原因是还没有关联源代码的问题:**[**关联上 tomcat src 文件即可**](https://jingyan.baidu.com/article/0202781170ffa61bcc9ce5dd.html)

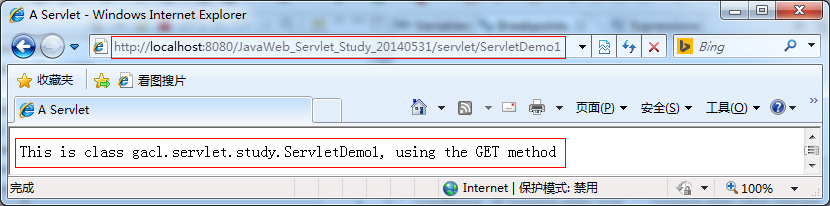
****

这样，我们就通过Eclipse帮我们创建好一个名字为ServletDemo1的Servlet，创建好的ServletDemo01。

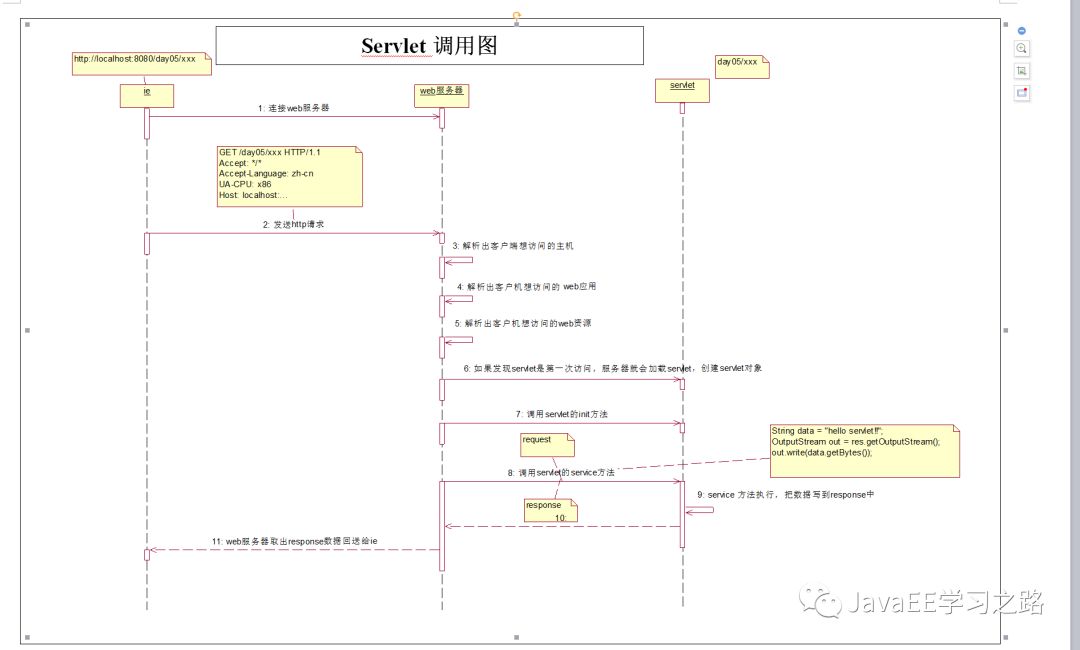
**配置xml文件，光写了Servlet是不行的，Tomcat还要知道浏览器怎么访问这个Servlet。**web.xml文件中也多了<servlet></servlet>和<servlet-mapping></servlet-mapping>两对标签，这两对标签是配置ServletDemo1的，如下图所示：

****

然后我们就可以通过浏览器访问ServletDemo1这个Servlet，如下图所示：

****

**Servlet的调用图**



### Servlet 请求过程

**Servlet 的请求流程**

1. **浏览器发出请求：**http://localhost:80/xxx1/xxx2 （80端口可以默认不写，因为这是http协议默认的端口，平时我们访问https://www.baidu.com/ 时其实访问的是https://www.baidu.com:80/）
2. **服务器解析请求信息：**
   * **http:**协议名称
   * **localhost:**访问的是互联网中的**哪一台计算机**
   * **80:**从主机当中找到**对应 80 端口的程序**（**这里即为 Tomcat 服务器**）
   * **/xxx1:**当前项目的**上下文路径** （即在 server.xml 中配置主机时配置的 **path属性**）
   * **/xxx2:**当前**请求的资源名**
3. **解析** Tomcat 服务器根目录下的 **/config/server.xml** 文件：  
   <Context docBase="D:\javaPros\test\webapp" path="xxx1" />  
   判断哪一个<Context />元素的 **path属性** 属性为 xxx1
   * 若找不到，则返回 **404错误**
   * 若找到了，则解析该<Context />元素，得到docBase属性，获取当前访问 Web 项目的跟的绝对路径：D:\javaPros\test\webapp
4. 从D:\javaPros\test\webapp下的 **WEB-INF** 下找到 **web.xml** 文件  
   判断 **web.xml** 文件中是否有 <url-pattern> 的文本内容为 **/xxx2**
   * 若找不到，则返回 **404错误**
   * 若找到了，则继续**获取该资源对应 Servlet 类的全限名称：** xxx.xxx
5. 判断 **Servlet 实例缓存池** 中是否有 xxx.xxx 的对象

Map<String,Servlet> cache = ......(Tomcat提供的);

key:存Servlet类的全限定名称

value:该Servlet类的对象.

Servlet obj = cache.get("xxx.xxx");

if(obj==null){

//Servlet实例缓存中没有该类的对象,第一次.

GOTO 6:

}else{

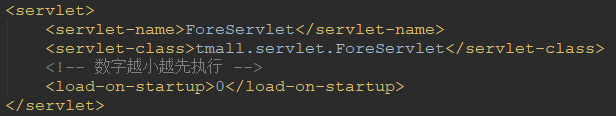
//有对象,非第一次.

GOTO 8:

}

}

1. **使用反射**调用构造器，**创建对应的对象**obj = Class.forName("xxx.xxx").newInstance();把当前创建的 **Servlet 对象**，存放在缓存之中，**供给下一次的使用.**cache.put("xxx.xxx",obj);
2. 创建 **ServletConfig 对象**，并调用 **init()** 方法  
   obj.init(config);

关于 **Servlet 的初始化操作，**如果初始化操作非常的耗时，那么第一个请求的用户的用户体验就非常差。  
**解决思路：**将初始化操作**向前移,**在服务器启动的时候执行 Servlet 的初始化  


1. 创建 **ServletRequest 对象和 ServletResponse 对象**，并调用 **service()**方法  
   obj.service(req,resp);
2. 在 **service()** 方法中对浏览器作出响应操作。

**Servlet开发注意细节**

### GET 和 POST 的区别

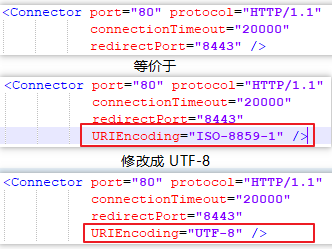
要知道，GET 和 POST 都是请求方式

* **GET：**  
  浏览器器地址栏：http://localhost/test.html**?name=wmyskxz&sex=male**  
  这里提交了两个参数，一个是name属性值为wmyskxz，另一个是sex属性值为male，这是一种直接的请求方式，在请求资源后面跟上 **?** 符号与参数连接，其他的参数使用 **&** 符号连接。
* **缺点：**  
  1.暴露请求信息，**不安全**  
  2.请求信息不能超过**1kb**，可传输的信息有限，不能上传图片
* **POST：**  
  浏览器地址栏：http://localhost/test.html#
* **优点：**  
  1.隐藏了请求信息，**较安全**（但仍可以通过相关工具访问到数据）  
  2.POST 方式**没有限制**请求的数据大小，可以做图片的上传

但并不是所有的数据都需要使用 POST 请求来完成，事实上，GET 请求方式会比 POST 请求更快，当数据小并且安全性要求不是那么高的时候，GET 仍然是很好的选择.(并且 GET 相较 POST 简单)

### 请求中文乱码的处理

在 **Tomcat 服务器**中，接受请求的时候，**默认的编码方式为 ISO-8859-1**，而该编码方式只占一个字节，不支持中文（两个字节），所以当我们做请求的时候，会出现乱码的问题

* **解决方案：**  
  1.对乱码使用 **ISO-8859-1** 解码，转换成**byte数组**，恢复为二进制  
  byte[] data = name.getBytes("ISO-8859-1");  
  2.对byte数组重新进行 UTF-8 编码：  
  name = new String(data,"UTF-8");  
  但是这样会出现一个问题，那就是当表单数据太多的时候，这样反复解码-编码，会很繁琐。
* **终极解决方案：**  
  **1.对于 POST 请求：**  
  设置请求的编码方式：request.setCharacterEncoding("UTF-8");  
  **注意：**必须在获取**第一个参数之前设置**，并且该方式**只对 POST 方式有效。**  
  **2.对于 GET 请求：**  
  重新设置 Tomcat 的编码方式，修改 Tomcat 的配置文件:  
  Tomcat根目录/conf/server.xml(修改端口的那一行)  
  

### Servlet访问URL映射配置

由于客户端是通过URL地址访问web服务器中的资源，所以Servlet程序若想被外界访问，必须把servlet程序映射到一个URL地址上，这个工作**在web.xml文件中使用<servlet>元素和<servlet-mapping>元素完成**。  
　　<servlet>元素用于注册Servlet，它包含有两个主要的子元素：<servlet-name>和<servlet-class>，分别用于设置Servlet的**注册名称**和Servlet的**完整类名**。   
 <servlet-mapping>元素用于映射一个已注册的Servlet的一个对外访问路径，它包含有两个子元素：<servlet-name>和<url-pattern>，分别用于指定Servlet的**注册名称**和Servlet的**对外访问路径**。

<servlet>

<servlet-name>ServletDemo1</servlet-name>

<servlet-class>gacl.servlet.study.ServletDemo1</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>ServletDemo1</servlet-name>

<url-pattern>**/servlet/ServletDemo1**</url-pattern>

</servlet-mapping>

**同一个Servlet可以被映射到多个URL**上，即多个<servlet-mapping>元素的<servlet-name>子元素的设置值可以是同一个Servlet的注册名。 例如：

<servlet>

<servlet-name>ServletDemo1</servlet-name>

<servlet-class>gacl.servlet.study.ServletDemo1</servlet-class>

</servlet>

<servlet-mapping>

<servlet-name>ServletDemo1</servlet-name>

<url-pattern>**/servlet/ServletDemo1**</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>ServletDemo1</servlet-name>

<url-pattern>/**1.htm**</url-pattern>

</servlet-mapping>

<servlet-mapping>

<servlet-name>ServletDemo1</servlet-name>

<url-pattern>**/2.jsp**</url-pattern>

</servlet-mapping>

等。。。。。。

通过上面的配置，当**我们想访问名称是ServletDemo1的Servlet**，可以使用如下的几个地址去访问：

　　http://localhost:8080/JavaWeb\_Servlet\_Study\_20140531**/servlet/ServletDemo1**

　　http://localhost:8080/JavaWeb\_Servlet\_Study\_20140531**/1.htm**

　　http://localhost:8080/JavaWeb\_Servlet\_Study\_20140531**/2.jsp**

**ServletDemo1被映射到了多个URL上。**

### Servlet访问URL使用\*通配符映射

通配符有两种格式：

1. **\*.扩展名；\*可以匹配任意的字符**
2. **正斜杠（/）开头并以“/\*”结尾。**
3. **\*.do**

**\*.jsp匹配扩展名为.jsp的**

**/\*匹配所有**

如果*.*扩展名和正斜杠（/）开头并以“/”结尾两种通配符同时出现，匹配的是哪一个呢？

* **看谁的匹配度高，谁就被选择**
* **\*.扩展名的优先级最低**
* **按照 “ 精确路径 > 最长路径>扩展名”这样的优先级匹配到对应的servlet**

**Servlet映射的URL可以使用通配符和Servlet可以被映射到多个URL上的作用：**

1. **隐藏网站是用什么编程语言写的【.php,.net,.asp实际上访问的都是同一个资源】**
2. **用特定的后缀声明版权【公司缩写】**

### Servlet与普通Java类的区别

Servlet是一个**供其他Java程序（Servlet引擎）调用的Java类**，它不能独立运行，它的**运行完全由Servlet引擎来控制和调度。**

　**Servlet是单例的**：针对客户端的多次Servlet请求，通常情况下，**服务器只会创建一个Servlet实例对象，**也就是说**Servlet实例对象一旦创建，它就会驻留在内存中，为后续的其它请求服务，直至web容器退出，servlet实例对象才会销毁。**  
　　在Servlet的整个生命周期内，Servlet的init方法只被调用一次。而对一个Servlet的每次访问请求都导致Servlet引擎调用一次servlet的service方法。**每次访问请求，Servlet引擎都会创建一个新的HttpServletRequest请求对象和一个新的HttpServletResponse响应对象**，然后将这两个对象作为参数传递给它调用的Servlet的service()方法，**service方法再根据请求方式分别调用doXXX方法。**

　如果在<servlet>元素中配置了一个**<load-on-startup>元素**，那么WEB应用程序在启动时，就会**装载并创建Servlet的实例对象、以及调用Servlet实例对象的init()方法。**  
    举例：

<servlet>

<servlet-name>ServletDemo1</servlet-name>

<servlet-class>gacl.servlet.study.ServletDemo1</servlet-class>

<l**oad-on-startup>**1</load-on-startup>

</servlet>

**用途**：为web应用写一个**InitServlet**，这个servlet配置为启动时装载，为整个web应用创建必要的数据库表和数据。

完成一些定时的任务【定时写日志，定时备份数据】

### 缺省Servlet

当你启动Tomcat，你在网址上输入http://localhost:8080。为什么会出现Tomcat小猫的页面？这是由**缺省Servlet为你服务的**！

<servlet>

<servlet-name>ServletDemo2</servlet-name>

<servlet-class>gacl.servlet.study.ServletDemo2</servlet-class>

</servlet>

<!-- **将ServletDemo2配置成缺省Servlet** -->

<servlet-mapping>

<servlet-name>ServletDemo2</servlet-name>

<url-pattern>/</url-pattern>

</servlet-mapping>

当访问不存在的Servlet时，就**使用配置的默认Servlet进行处理。**

在<tomcat的安装目录>\**conf\web.xml**文件中，**注册了一个名称为org.apache.catalina.servlets.DefaultServlet的Servlet**，并将这个Servlet设置为了**缺省Servlet。**

* 什么叫做缺省Servlet？凡是**在web.xml文件中找不到匹配的<servlet-mapping>元素的URL**，它们的访问**请求都将交给缺省Servlet处理**，也就是说，**缺省Servlet用于处理所有其他Servlet都不处理的访问请求**
* 既然我说了在web访问任何资源都是在访问Servlet，那么**我访问静态资源【本地图片，本地HTML文件】也是在访问这个缺省Servlet【DefaultServlet】**
* 证实一下：当我没有手工配置缺省Servlet的时候，**访问本地图片是可以访问得到的**

现在我自己配置一个缺省Servlet**Demo1就是我手工配置的缺省Servlet，覆盖掉web.xml配置的缺省Servlet**

### Servlet的线程安全问题

当**多个客户端并发访问同一个Servlet**时，web服务器会为每一个客户端的**访问请求创建一个线程，并在这个线程上调用Servlet的service方法**，因此service方法内如果**访问了同一个资源的话，就有可能引发线程安全问题**。

**Servlet对象是单实例多线程，Servlet不是线程安全的。**

不存在线程安全问题的代码：

* public class ServletDemo3 extends HttpServlet {

public void **doGet**(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

/\*\*

\* 当多线程并发访问这个方法里面的代码时，会存在线程安全问题吗

\* i变量被多个线程并发访问，但是没有线程安全问题，因为i是doGet方法里面的局部变量，

\* 当有多个线程并发访问doGet方法时，每一个线程里面都有自己的i变量，

\* 各个线程操作的都是自己的i变量，所以不存在线程安全问题

\* 多线程并发访问某一个方法的时候，如果在方法内部定义了一些资源(变量，集合,文件等)

\* 那么每一个线程都有这些东西，所以就不存在线程安全问题了

\*/

int i=1;

i++;

response.getWriter().write(i);

}

/\*\* **把i定义成全局变量，**当多个线程并发访问变量i时，就会存在线程安全问题了，如下图所示：同时开启两个浏览器模拟并发访问同一个Servlet，本来正常来说，第一个浏览器应该看到2，而第二个浏览器应该看到3的，结果两个浏览器都看到了3，这就不正常。\*/

* **synchronized (this)** {//在java中，每一个对象都有一把锁，这里的this指的就是Servlet对象

**原则：**

* 线程安全问题只存在**多个线程并发操作同一个资源的情况下**，所以在编写Servlet的时候，如果并发访问某一个资源，**加同步机制synchronized (对象){}**
* 如果一个变量**不需要共享**，则**直接在 doGet() 或者 doPost()定义**.这样不会存在线程安全问题

针对Servlet的线程安全问题，Sun公司是提供有解决方案的：**让Servlet去实现一个SingleThreadModel接口，那么Servlet引擎将以单线程模式来调用其service方法。**  
　　查看Sevlet的API可以看到，SingleThreadModel接口中没有定义任何方法和常量，**在Java中，把没有定义任何方法和常量的接口称之为标记接口**，经常看到的一个最典型的标记接口就是"**Serializable**"，这个接口也是没有定义任何方法和常量的，标记接口在Java中有什么用呢？主要作用就是给某个对象打上一个标志，告诉JVM，这个对象可以做什么，比如实现了"**Serializable**"接口的类的对象就可以被序列化，还有一个"Cloneable"接口，这个也是一个标记接口，在默认情况下，Java中的对象是不允许被克隆的，就像现实生活中的人一样，不允许克隆，但是只要实现了"Cloneable"接口，那么对象就可以被克隆了。

让**Servlet实现了SingleThreadModel接口，**只要在Servlet类的定义中增加实现SingleThreadModel接口的声明即可。    
　　**对于实现了SingleThreadModel接口的Servlet，Servlet引擎仍然支持对该Servlet的多线程并发访问，其采用的方式是产生多个Servlet实例对象，并发的每个线程分别调用一个独立的Servlet实例对象**。  
　　实现SingleThreadModel接口并不能真正解决Servlet的线程安全问题，因为Servlet引擎**会创建多个Servlet实例对象**，而真正意义上解决多线程安全问题是指一个Servlet实例对象被多个线程同时调用的问题。事实上，在Servlet API 2.4中，已经将SingleThreadModel标记为Deprecated（过时的）

**ServletConfig对象**

### 配置Servlet初始化参数

在Servlet的配置文件web.xml中，可以使用一个或多个<init-param>标签为servlet配置一些初始化参数。

**能够让你的程序更加灵活**【更换需求，更改配置文件web.xml即可，程序代码不用改】

<servlet>

<servlet-name>ServletConfigDemo1</servlet-name>

<servlet-class>cn.hgd.ServletConfigDemo1</servlet-class>

<!--配置ServletConfigDemo1的初始化参数 -->

<init-param>

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

<param-value>gacl</param-value>

</init-param>

<init-param>

<param-name>password</param-name>

<param-value>123</param-value>

</init-param>

<init-param>

<param-name>charset</param-name>

<param-value>UTF-8</param-value>

</init-param>

</servlet>

**通过ServletConfig获取Servlet（web.xml文件配置的参数）的初始化参数**

当servlet配置了初始化参数后，**web容器在创建servlet实例对象时，会自动将这些初始化参数封装到ServletConfig对象中，并在调用servlet的init方法时，将ServletConfig对象传递给servlet**。进而，我们通过ServletConfig对象就可以得到当前servlet的初始化参数信息。

**public class ServletConfigDemo1 extends HttpServlet {**

/\*\*

\* 定义ServletConfig对象来接收配置的初始化参数

\*/

**private ServletConfig config;**

/\*\*

\* 当servlet配置了初始化参数后，web容器在创建servlet实例对象时，

\* 会自动将这些初始化参数封装到ServletConfig对象中，并在调用servlet的init方法时，

\* 将ServletConfig对象传递给servlet。进而，程序员通过ServletConfig对象就可以

\* 得到当前servlet的初始化参数信息。

\*/

@Override

public void init(ServletConfig config) throws ServletException {

**this.config = config;**

}

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

//获取在web.xml中配置的初始化参数

**String paramVal = this.config.getInitParameter("name");//获取指定初始化参数**

response.getWriter().print(paramVal);

* **//获取所有的初始化参数**

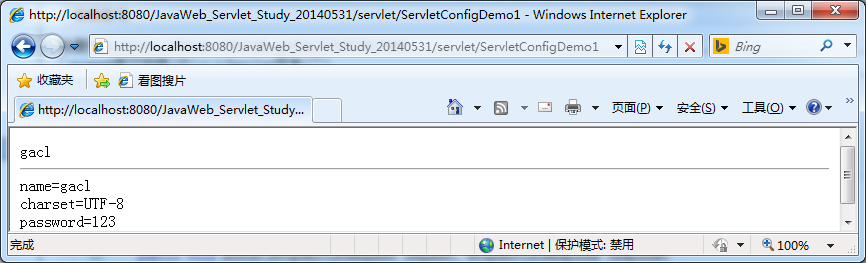
Enumeration<String> e = config.getInitParameterNames();//迭代器

while(e.hasMoreElements()){

String name = e.nextElement();

String value = config.getInitParameter(name);

response.getWriter().print(name + "=" + value + "<br/>")；//浏览器加"<br/>"，换行



### 应用情况：

1. 得到字符编码
2. 链接数据库，可以修改
3. 获得配置文件

**ServletContext对象**

当Tomcat启动的时候，就会创建一个**ServletContext对象**。它**代表着当前web站点**

**ServletConfig对象中维护了ServletContext对象的引用**，开发人员在编写servlet时，可以**通过ServletConfig.getServletContext方法获得ServletContext对象。**  
　　由于一个WEB应用中的所有Servlet共享同一个ServletContext对象，因此Servlet对象之间可以通过ServletContext对象来实现通讯。ServletContext对象通常也被称之为**context域对象。其实域对象可以简单理解成一个容器【类似于Map集合】**

**ServletContext有什么用？**

1. ServletContext既然代表着当前web站点，那么**所有Servlet都共享着一个ServletContext对象**，所以**Servlet之间可以通过ServletContext实现通讯**。
2. ServletConfig获取的是配置的是单个Servlet的参数信息，**ServletContext可以获取的是配置整个web站点的参数信息**
3. **利用ServletContext读取web站点的资源文件**
4. 实现Servlet的转发【用ServletContext转发不多，主要用request转发】

**ServletContext的应用**

### 多个Servlet通过ServletContext对象实现数据共享

**实现Servlet之间通讯就要用到ServletContext的setAttribute(String name,Object obj)方法**

* **public class ServletContextDemo1 extends HttpServlet {**

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

**String data = "xdp\_gacl";**

/\*\*

\* ServletConfig对象中维护了ServletContext对象的引用，开发人员在编写servlet时，

\* 可以通过ServletConfig.getServletContext方法获得ServletContext对象。

\*/

**ServletContext context = this.getServletConfig().getServletContext();//获得ServletContext对象**

**context.setAttribute("data", data); //将data存储到ServletContext对象中**

}

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

doGet(request, response);

* **public class ServletContextDemo2 extends HttpServlet {**

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

**ServletContext context = this.getServletContext();**

**String data = (String) context.getAttribute("data");**//从ServletContext对象中取出数据

response.getWriter().print("data="+data);

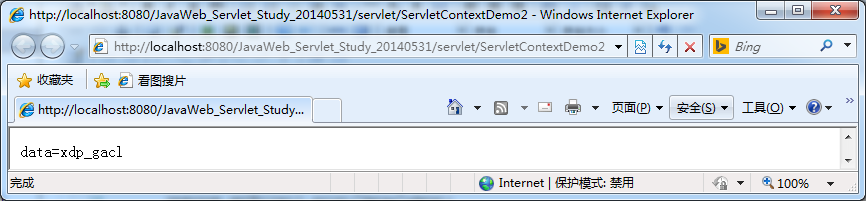
}

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

doGet(request, response);

先运行ServletContextDemo1，将数据data存储到ServletContext对象中，然后运行ServletContextDemo2就可以从ServletContext对象中取出数据了，这样就实现了数据共享，如下图所示：



### 获取web站点配置的信息

如果我想要让所有的Servlet都能够获取到连接数据库的信息，不可能在web.xml文件中每个Servlet中都配置一下，这样代码量太大了！并且会显得非常啰嗦冗余。

在web.xml文件中使用<context-param>标签配置WEB应用的初始化参数

<!-- 配置WEB应用的初始化参数 -->

<context-param>

<param-name>url</param-name>

<param-value>jdbc:mysql://localhost:3306/test</param-value>

</context-param>

public class ServletContextDemo3 extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

**ServletContext context = this.getServletContext();**

//获取整个web站点的初始化参数

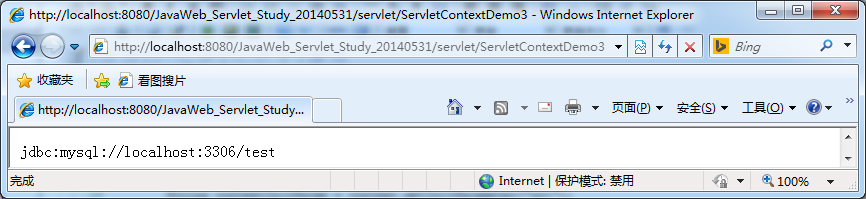
**String contextInitParam = context.getInitParameter(**"url");

response.getWriter().print(contextInitParam);

public void doPost(HttpServletRequest request, HttpServletResponse response)

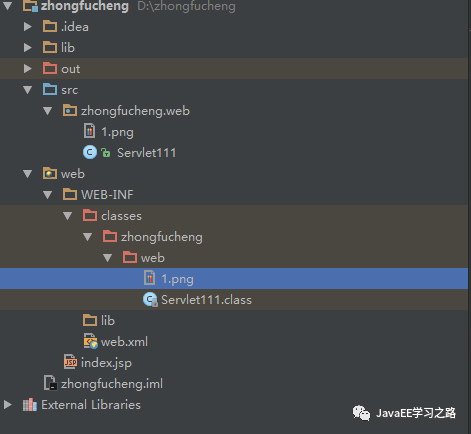
throws ServletException, IOException {

doGet(request, response);



### 利用ServletContext对象读取资源文件

* 根据web的目录规范，**Servlet编译后的class文件是存放在WEB-INF\classes文件夹中的**。我们知道了**要进入classes目录中读取文件**。



FileInputStream fileInputStream = **new** FileInputStream("D:\\zhongfucheng\\web\\WEB-INF\\classes\\zhongfucheng\\web\\1.png");System.out.println(fileInputStream);

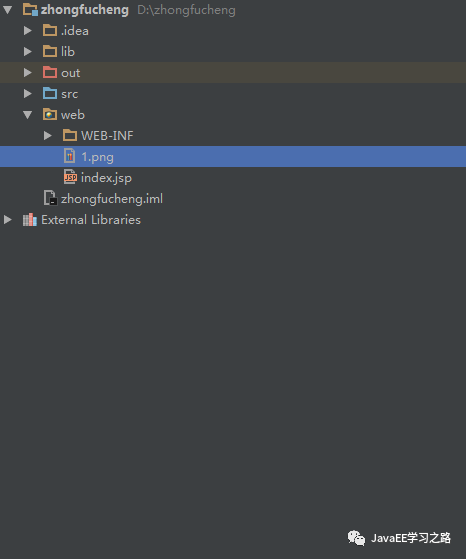
* 但是现在问题又来了，**我读取文件的时候都要写上绝对路径，这样太不灵活了**。试想一下，如果我将该**读取文件的模块移到其他的web站点上**，我的代码就又要修改了【因为web站点的名字不一样】。
* 我们通过**ServletContext读取就可以避免修改代码的情况**，因为ServletContext对象是根据当前web站点而生成的
* //获取到ServletContext对象

ServletContext servletContext = this.getServletContext();

//调用ServletContext方法获取到读取文件的流

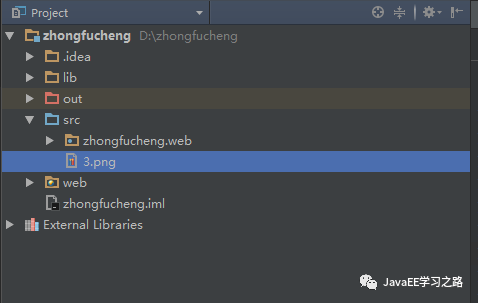
InputStream inputStream = servletContext.getResourceAsStream("/WEB-INF/classes/zhongfucheng/web/1.png");

* 如果我的**文件放在web目录下**，那么就简单得多了！,**直接通过文件名称就能获取**



InputStream inputStream = servletContext.getResourceAsStream("1.png");

* 通过**类装载器读取资源文件**。



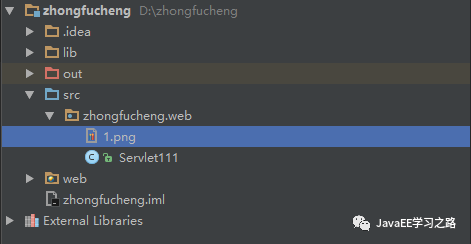
* 我的文件放在了**src目录下【也叫做类目录】**

ClassLoader classLoader = Servlet111.class.getClassLoader();

//通过类装载器获取到读取文件流

InputStream inputStream = classLoader.getResourceAsStream("3.png");

* 我的文件放在了**src目录下的包下**



ClassLoader Loader = Servlet111.class.getClassLoader()  
 //通过类装载器获取到读取文件流

InputStream inputStream = Loader.getResourceAsStream("/zhongfucheng/web/1.png");

**原则：如果文件太大，就不能用类装载器的方式去读取，会导致内存溢出**

/\*\*

\* 通过类装载器读取资源文件的注意事项:**不适合装载大文件，否则会导致jvm内存溢出**

**\*/**

public void test3() {

/\*\*

\* 01.avi是一个150多M的文件，使用类加载器去读取这个大文件时会导致内存溢出：

\* java.lang.OutOfMemoryError: Java heap space

\*/

InputStream in = ServletContextDemo7.class.getClassLoader().getResourceAsStream("01.avi");

System.out.println(in);

}

/\*\*

\* 读取01.avi,并拷贝到e:\根目录下

\* **01.avi文件太大，只能用servletContext去读取**

\* @throws IOException

\*/

public void test4() throws IOException {

// path=G:\Java学习视频\JavaWeb学习视频\JavaWeb\day05视频\01.avi

// path=01.avi

String path = this.getServletContext().getRealPath("/WEB-INF/classes/01.avi");

/\*\*

\* path.lastIndexOf("\\") + 1是一个非常绝妙的写法

\*/

**String filename = path.substring(path.lastIndexOf("\\") + 1);//获取文件名**

InputStream in = this.getServletContext().getResourceAsStream("/WEB-INF/classes/01.avi");

byte buffer[] = new byte[1024];

int len = 0;

OutputStream out = new FileOutputStream("e:\\" + filename);

while ((len = in.read(buffer)) > 0) {

out.write(buffer, 0, len);

}

out.close();

in.close();

}

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

this.doGet(request, response);

}

### 请求转发

**public class ServletContextDemo4 extends HttpServlet {**

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String data = "<h1><font color='red'>abcdefghjkl</font></h1>";

response.getOutputStream().write(data.getBytes());

ServletContext context = this.getServletContext();//获取ServletContext对象

**RequestDispatcher rd = context.getRequestDispatcher("/servlet/ServletContextDemo5");//获取请求转发对象(RequestDispatcher)**

**rd.forward(request, response);//调用forward方法实现请求转发**

**}**

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

}

}

**public class ServletContextDemo5 extends HttpServlet {**

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

response.getOutputStream().write("servletDemo5".getBytes());

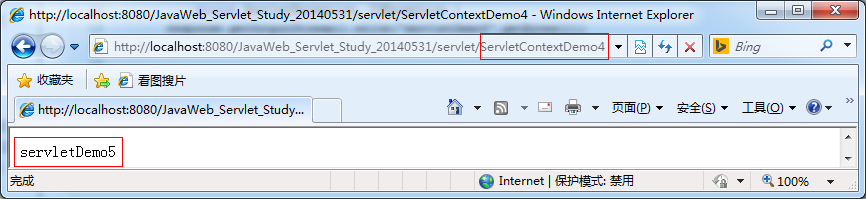
}

public void doPost(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

this.doGet(request, response);

}



**访问的是ServletContextDemo4，浏览器显示的却是ServletContextDemo5的内容**，这就是使用ServletContext实现了请求转发

### 在客户端缓存Servlet的输出

对于不经常变化的数据，在servlet中可以为其设置合理的缓存时间值，以避免浏览器频繁向服务器发送请求，提升服务器的性能。

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String data = "abcddfwerwesfasfsadf";

/\*\*

\* 设置数据合理的缓存时间值，以避免浏览器频繁向服务器发送请求，提升服务器的性能

\* 这里是将数据的缓存时间设置为1天

\*/

**response.setDateHeader("expires",System.currentTimeMillis() + 24 \* 3600 \* 1000);**

response.getOutputStream().write(data.getBytes());

}

# response

概念

Web服务器收到客户端的http请求，**会针对每一次请求**，分别创建一个用于代表请求的request对象、和代表响应的response对象。  
 request和response对象即然代表请求和响应，那我们要获取客户机提交过来的数据，只需要找request对象就行了。要向客户机输出数据，只需要找response对象就行了。

**HttpServletResponse对象**代表服务器的响应。这个对象中封装了向客户端发送**数据、发送响应头，发送响应状态码的方法**。

**HttpServletResponse对象常见应用**

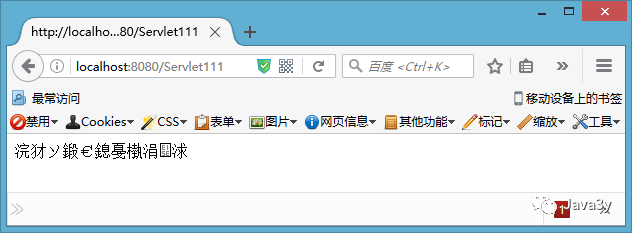
### 使用OutputStream流输出

**中文注意问题：**

为什么使用**write()方法能够正常向浏览器输出中文**呢？"你好呀我是中国".getBytes()这句代码在**转成byte[]数组的时候默认查的是gb2312编码**，而**"你好呀我是中国"支持gb2312编码**，所以可以正常显示出来。

为什么它变成了乱码呢？原因是这样的：我在向服务器输出的**中文是UTF-8编码的，而浏览器采用的是GBK，GBK想显示UTF-8的中文数据，不乱码才怪呢**！

**在服务器端，数据是以哪个码表输出的，就要控制客户端浏览器以相应码表打开**，比如：**outputStream.write("中国".getBytes("UTF-8"))**;使用OutputStream流向客户端浏览器输中文，**以UTF-8的编码进行输出**，此时就要**控制客户端浏览器以UTF-8的编码打开**，中文乱码。**通过设置响应头**控制浏览器行为：**response.setHeader("content-type", "text/html;charset=UTF-8");**通过设置响应头控制浏览器以UTF-8的编码显示数据。



**HTTP响应有对浏览器说明回送数据是什么类型的消息头**，那么HttpServletResponse对象就应该有相对应的方法告诉浏览器回送数据编码格式是什么。找**设置消息头的方法。**

**优先级高**

* **浏览器在显示数据时，自动把页面的编码格式置换成UTF-8**，乱码问题也解决了

String data = "中国";

OutputStream outputStream = response.getOutputStream();

response.setHeader("content-type", "text/html;charset=UTF-8");//通过设置响应头控制浏览器以UTF-8的编码显示数据，如果不加这句话，那么浏览器显示的将是乱码

/\*\*

\* data.getBytes()是一个将**字符转换成字节数组**的过程，一定会去查码表，

\* 如果是中文的操作系统环境，默认就是查找查GB2312的码表，

\* 字符转换成字节数组的过程就是将中文字符转换成GB2312的码表对应数字

* 另外，除了使用HttpServletResponse对象设置消息头的方法，我可以**使用html的标签模拟一个http消息头**

 //获取到servletOutputStream对象  
        ServletOutputStream servletOutputStream = response.getOutputStream();  
        //使用meta标签模拟http消息头，告诉浏览器回送数据的编码和格式  
        servletOutputStream.write("<meta http-equiv='content-type' content='text/html;charset=UTF-8'>".getBytes());  
        servletOutputStream.write("我是中国".getBytes("UTF-8"));

### 调用getWriter()方法向浏览器输出数据

对于getWriter()方法而言，是Writer的子类，那么**只能向浏览器输出字符数据，不能输出二进制数据**

/***/原本是ISO 8859-1的编码\*国外***，我设置成UTF-8

**response.setCharacterEncoding("UTF-8");**//设置将字符以"UTF-8"编码输出到客户端浏览器

/\*\* **PrintWriter out = response.getWriter();这句代码必须放在response.setCharacterEncoding("UTF-8");之后**

\* 否则response.setCharacterEncoding("UTF-8")这行代码的设置将无效，浏览器显示的时候还是乱码

\*/

**PrintWriter out = response.getWriter();**//获取PrintWriter输出流

printWriter.write("中国！");

**乱码问题还没有解决？**细心的朋友会发现，我只是在**中文转换的时候把码表设置成UTF-8**，但是**浏览器未必是使用UTF-8码表来显示数据的呀**

**不仅设置浏览器用UTF-8显示数据，内部还把中文转码的码表设置成UTF-8了**

   //设置浏览器用UTF-8编码显示数据，  
        **response.setContentType("text/html;charset=UTF-8")；**  
        //获取到printWriter对象  
        PrintWriter printWriter = response.getWriter();  
        printWriter.write("看完博客点赞！");

response.setHeader("content-type", "text/html;charset=UTF-8");

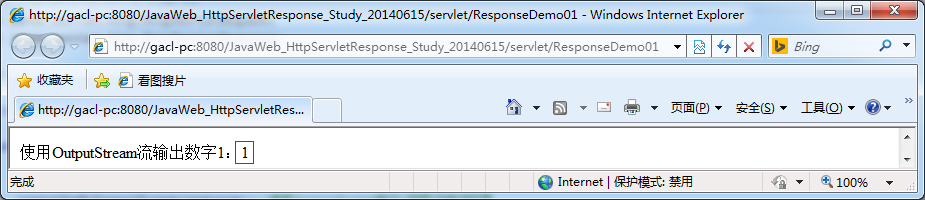
OutputStream outputStream = response.getOutputStream();

outputStream.write("使用OutputStream流输出数字1：".getBytes("UTF-8"));

//outputStream.write(1);

outputStream.write((1+"").getBytes());

**1+""**这一步是将数字1和一个空字符串相加，这样处理之后，数字1就变成了字符串1了，然后再将字符串1转换成字节数组使用OutputStream进行输出，此时看到的结果如下



这次可以看到输出来的1了，这说明了一个问题：**在开发过程中，如果希望服务器输出什么浏览器就能看到什么，那么在服务器端都要以字符串的形式进行输出**。

**getWriter和getOutputStream细节**

1. **getWriter()和getOutputStream()两个方法不能同时调用。如果同时调用就会出现异常**
2. **Servlet程序向ServletOutputStream或PrintWriter对象中写入的数据将被Servlet引擎从response里面获取，Servlet引擎将这些数据当作响应消息的正文，然后再与响应状态行和各响应头组合后输出到客户端。**
3. **Servlet的serice()方法结束后【也就是doPost()或者doGet()结束后】，Servlet引擎将检查getWriter或getOutputStream方法返回的输出流对象是否已经调用过close方法，如果没有，Servlet引擎将调用close方法关闭该输出流对象.**

### 文件下载

文件下载注意事项：**编写文件下载功能时推荐使用OutputStream流，避免使用PrintWriter流，因为OutputStream流是字节流，可以处理任意类型的数据，而PrintWriter流是字符流，只能处理字符数据，如果用字符流处理字节数据，会导致数据丢失。**

文件下载功能是web开发中经常使用到的功能，使用HttpServletResponse对象就可以实现文件的下载

**文件下载功能的实现思路**：

　　1.获取要下载的文件的绝对路径

　　2.获取要下载的文件名

　　3.设置content-disposition响应头控制浏览器以下载的形式打开文件

　　4.获取要下载的文件输入流

　　5.创建数据缓冲区

　　6.通过response对象获取OutputStream流

　　7.将FileInputStream流写入到buffer缓冲区

8.使用OutputStream将缓冲区的数据输出到客户端浏览器

public class ResponseDemo02 extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

downloadFileByOutputStream(response);//下载文件，通过OutputStream流

}

/\*\*

\* 下载文件，通过OutputStream流

\* @param response

\* @throws FileNotFoundException

\* @throws IOException

\*/

private void downloadFileByOutputStream(HttpServletResponse response)

throws FileNotFoundException, IOException {

//1.获取要下载的文件的绝对路径

String realPath = this.getServletContext().getRealPath("/download/1.JPG");

//2.获取要下载的文件名

String fileName = realPath.substring(realPath.lastIndexOf("\\")+1);

//3.设置content-disposition响应头控制浏览器以下载的形式打开文件

response.setHeader("content-disposition","attachment;filename="+fileName);

//4.获取要下载的文件输入流

InputStream in = new FileInputStream(realPath);

int len = 0;

//5.创建数据缓冲区

byte[] buffer = new byte[1024];

//6.通过response对象获取OutputStream流

 ServletOutputStream out = response.getOutputStream();

//7.将FileInputStream流写入到buffer缓冲区

while ((len = in.read(buffer)) > 0) {

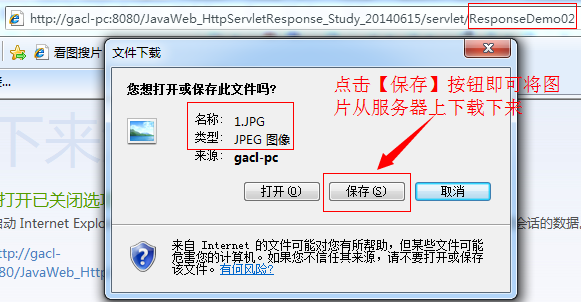
//8.使用OutputStream将缓冲区的数据输出到客户端浏览器

out.write(buffer,0,len);

}

in.close();

out.close();



**下载中文文件时，需要注意的地方就是中文文件名要使用URLEncoder.encode方法进行编码(URLEncoder.encode(fileName, "字符编码"))，否则会出现文件名乱码。**

**response.setHeader("Content-Disposition", "attachment; filename=" + URLEncoder.encode(fileName, "UTF-8"));**

public class ResponseDemo02 extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

downloadChineseFileByOutputStream(response);//下载中文文件

}

/\*\*

\* 下载中文文件,中文文件下载时，**文件名要经过URL编码，否则会出现文件名乱码**

\* @param response

\* @throws FileNotFoundException

\* @throws IOException

\*/

private void downloadChineseFileByOutputStream(HttpServletResponse response)

throws FileNotFoundException, IOException {

String realPath = this.getServletContext().getRealPath("/download/张家界国家森林公园.JPG");//获取要下载的文件的绝对路径

String fileName = realPath.substring(realPath.lastIndexOf("\\")+1);//获取要下载的文件名

//设置content-disposition响应头控制浏览器以下载的形式打开文件，中文文件名要使用URLEncoder.encode方法进行编码，否则会出现文件名乱码

response.setHeader("content-disposition", "attachment;filename="+URLEncoder.encode(fileName, "UTF-8"));

InputStream in = new FileInputStream(realPath);//获取文件输入流

int len = 0;

byte[] buffer = new byte[1024];

OutputStream out = response.getOutputStream();

while ((len = in.read(buffer)) > 0) {

out.write(buffer,0,len);//将缓冲区的数据输出到客户端浏览器

}

in.close();

}

### 实现自动刷新

**以规定的时间让页面刷新，更新资源**

* 自动刷新，能够实现页面的跳转
* 我们登陆完网站，很多时候都会看见【登陆成功，3秒后自动跳转….】，其实这个就是用Refresh来完成的。

       response.setContentType("text/html;charset=UTF-8");  
        response.getWriter().write("3秒后跳转页面.....");  
        //三秒后跳转到index.jsp页面，**web应用映射路径**我设置成/，url没有写上应用名  
        response.setHeader("Refresh", "3;url='/index.jsp'");

### 设置缓存

**浏览器本身就存在着缓存机制**

* 当我第一次访问index.jsp时，浏览器向服务器发了两次请求【一个是网页的，一个是图片的】
* 当我第二次访问index.jsp的时候，**浏览器将图片缓存起来了！图片不是重新加载的，是从缓存里面取出来的。**
* 像股票类型的网页是不能取缓存的数据的，数据都是要不断更新的。下面我就禁止缓存的功能

 //**浏览器有三消息头设置缓存，为了兼容性！**将三个消息头都设置了  
        response.setDateHeader("Expires", -1);  
        response.setHeader("Cache-Control","no-cache");  
        response.setHeader("Pragma", "no-cache");  
        //这里为了看效果  
        PrintWriter printWriter = response.getWriter();  
        printWriter.print("你好啊" + new Date().toString());

* 当然了，**如果页面有些数据不长期更新，你就将它设置成缓存，这样可以提高服务器的性能**

### 生出随机图片

**在我们登陆的时候经常要写验证码，而那些验证码是一张图片，就是通过HttpServletResponse写给浏览器的。**

生成图片主要用到了一个**BufferedImage**类。

public class ResponseDemo03 extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

**response.setHeader("refresh", "5");//设置refresh响应头控制浏览器每隔5秒钟刷新一次**

//1.在内存中创建一张图片

**BufferedImage image = new BufferedImage(80, 20, BufferedImage.TYPE\_INT\_RGB);**

//2.得到图片

//Graphics g = image.getGraphics();

Graphics2D g = (Graphics2D)image.getGraphics();

**g.setColor(Color.WHITE);//设置图片的背景色**

**g.fillRect(0, 0, 80, 20);//填充背景色，背景改成白色**

//3.向图片上写数据

g.setColor(Color.BLUE);//设置图片上字体的颜色

g.setFont(new Font(null, Font.BOLD, 20));

**g.drawString(makeNum(), 0, 20);**

//4.设置响应头控制浏览器浏览器以图片的方式打开

response.setContentType("image/jpeg");//等同于response.setHeader("Content-Type", "image/jpeg");

//5.设置响应头控制浏览器不缓存图片数据

response.setDateHeader("expries", -1);

response.setHeader("Cache-Control", "no-cache");

response.setHeader("Pragma", "no-cache");

//6.将图片写给浏览器

**ImageIO.write(image, "jpg", response.getOutputStream());}**

/\*\*

\* **生成随机数字**

\* @return

\*/

private String makeNum() {

Random random = new Random();

//这样就会生成0-7位的随机数，现在问题又来了，如果随机数不够7位呢？如果不够7位，我们加到7位就行了

int anInt = random.nextInt(9999999);

//将数字转成是字符串

String num = String.valueOf(anInt);

//判断位数有多少个，不够就加

StringBuffer stringBuffer = new StringBuffer();

for (int i = 0; i < 7 - num.length(); i++) {

stringBuffer.append("0");

}

return stringBuffer.append(num).toString();

}

### 实现数据压缩

* 压缩的原理是什么？我们知道**getOutputStream()和getWriter()都是直接把数据输出给浏览器的**。现在我要做的就是**让数据不直接输出给浏览器，先让我压缩了，再输出给浏览器**。**java提供了GZIP压缩类给我们**
* 就让我们使用GZIP类来对数据压缩吧

 //GZIP的构造方法需要一个**OutputStream子类对象**，究竟哪个对象适合，我们看下write()方法  
        GZIPOutputStream gzipOutputStream = new GZIPOutputStream();  
        //查看了下API，write()接收的是byte[]类型的。  
        gzipOutputStream.write();

* 于是我就在**构造函数上传递个ByteArrayOutputStream给它**

       //既然是byte[]类型，那么我就给他一个ByteArrayOutputStream  
        GZIPOutputStream gzipOutputStream = new GZIPOutputStream(new ByteArrayOutputStream());

* 而**GZIPOutputStream写数据时候，是把数据写到ByteArrayOutputStream上的，等会还要把数据取出来，再写给浏览器**，于是就**不能以匿名内部类的方式给GZIPOutputStream，必须把ByteArrayOutputStream定义出来**，

       //创建GZIPOutputStream对象，给予它ByteArrayOutputStream  
        ByteArrayOutputStream byteArrayOutputStream = new ByteArrayOutputStream();  
        GZIPOutputStream gzipOutputStream = new GZIPOutputStream(byteArrayOutputStream);  
        //GZIP对数据压缩，GZIP写入的数据是保存在byteArrayOutputStream上的  
        gzipOutputStream.write(ss.getBytes());  
        //gzipOutputStream有缓冲，把缓冲清了，并顺便关闭流  
        gzipOutputStream.close();

* 把**压缩后的数据取出来，写给浏览器**

         //将压缩的数据取出来  
        byte[] bytes = byteArrayOutputStream.toByteArray();  
        //将压缩的数据写给浏览器  
        response.getOutputStream().write(bytes);

* 我们来对比一下压缩前的大小和压缩后的大小

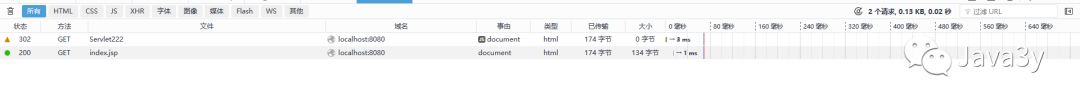


* 数据的确是压缩了，然而，为什么又乱码了啊？很简单，既然**你压缩了数据，你写给浏览器，浏览器是不知道你这是压缩后的数据，它是以正常的方式打开数据的**。这当然造成乱码啦！，现在我要**告诉浏览器我这是压缩数据**

       //告诉浏览器这是gzip压缩的数据  
        response.setHeader("Content-Encoding","gzip");  
        //再将压缩的数据写给浏览器  
        response.getOutputStream().write(bytes);



### 通过response实现请求重定向



请求重定向指：**一个web资源收到客户端请求后，通知客户端去访问另外一个web资源，这称之为请求重定向。**

**应用场景：**用户登陆，用户首先访问登录页面，登录成功后，就会跳转到某个页面，这个过程就是一个请求重定向的过程

实现方式：response.sendRedirect(String location)，调用response对象sendRedirect方法实现请求重定向  
　　sendRedirect内部的实现原理：**使用response设置302状态码和设置location响应头实现重定向**

public class ResponseDemo04 extends HttpServlet {

public void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

/\*\*

\* 1.**调用sendRedirect方法实现请求重定向**

\* sendRedirect方法内部调用了

\* response.setHeader("Location", "/JavaWeb\_HttpServletResponse\_Study\_20140615/index.jsp");

\*/ **response.sendRedirect("/JavaWeb\_HttpServletResponse\_Study\_20140615/index.jsp");**

//**2.使用response设置302状态码和设置location响应头实现重定向实现请求重定向**

* **response.setStatus(HttpServletResponse.SC\_FOUND);**//设置302状态码，等同于**response.setStatus(302);**
* //**response.setHeader("Location", "/JavaWeb\_HttpServletResponse\_Study\_20140615/index.jsp");**}
* **其实sendRedirect()方法就是对setStatus()和setHeader()进行封装**，原理就是setStatus()和setHeader()

### web工程中URL地址的推荐写法

在JavaWeb开发中，只要是写URL地址，那么建议最好以"/"开头，也就是使用绝对路径的方式，那么这个"/"到底代表什么呢？可以用如下的方式来记忆"/"：**如果"/"是给服务器用的，则代表当前的web工程，如果"/"是给浏览器用的，则代表webapps目录。**

**"/"代表当前web工程的常见应用场景**

this.getServletContext().getRealPath("/download/1.JPG");//就是读取web工程下的download文件夹中的1.JPG这个资源

**在服务器端forward到其他页面**

this.getServletContext().getRequestDispatcher("/index.jsp").forward(request, response);

客户端请求某个web资源，服务器跳转到另外一个web资源，这个forward也是给服务器用的

**"/"代表webapps目录的常见应用场景**

response.sendRedirect("/JavaWeb\_HttpServletResponse\_Study\_20140615/index.jsp");

服务器发送一个URL地址给浏览器，浏览器拿到URL地址之后，再去请求服务器，所以这个"/"是给浏览器使用的，此时"/"代表的就是webapps目录，"/JavaWeb\_HttpServletResponse\_Study\_20140615/index.jsp"这个地址指的就是"webapps\JavaWeb\_HttpServletResponse\_Study\_20140615\index.jsp"

**response.sendRedirect("/项目名称/文件夹目录/页面");**这种写法是将项目名称写死在程序中的做法,改写成：

**response.sendRedirect(request.getContextPath()+"/index.jsp");**

# Request

## HttpServletRequest介绍

HttpServletRequest对象代表客户端的请求，当客户端通过HTTP协议访问服务器时，HTTP请求头中的所有信息都封装在这个对象中，通过这个对象提供的方法，可以获得客户端请求的所有信息。

## Request常用方法

### 获得客户机信息

**getRequestURL方法返回客户端发出请求时的完整URL。**  
**getRequestURI方法返回请求行中的资源名部分。**  
**getQueryString 方法返回请求行中的参数部分。**  
　　getPathInfo方法返回请求URL中的额外路径信息。额外路径信息是请求URL中的位于Servlet的路径之后和查询参数之前的内容，它以“/”开头。  
　　**getRemoteAddr方法返回发出请求的客户机的IP地址。**  
　　getRemoteHost方法返回发出请求的客户机的完整主机名。  
　　getRemotePort方法返回客户机所使用的网络端口号。  
　　getLocalAddr方法返回WEB服务器的IP地址。  
　　getLocalName方法返回WEB服务器的主机名。

/\*\*

\* 1.获得客户机信息

\*/

String requestUrl = request.getRequestURL().toString();//得到请求的URL地址

String requestUri = request.getRequestURI();//得到请求的资源

String queryString = request.getQueryString();//得到请求的URL地址中附带的参数

String remoteAddr = request.getRemoteAddr();//得到来访者的IP地址

String remoteHost = request.getRemoteHost();

int remotePort = request.getRemotePort();

String remoteUser = request.getRemoteUser();

String method = request.getMethod();//得到请求URL地址时使用的方法

String pathInfo = request.getPathInfo();

String localAddr = request.getLocalAddr();//获取WEB服务器的IP地址

String localName = request.getLocalName();//获取WEB服务器的主机名

response.setCharacterEncoding("UTF-8");//设置将字符以"UTF-8"编码输出到客户端浏览器

//通过设置响应头控制浏览器以UTF-8的编码显示数据，如果不加这句话，那么浏览器显示的将是乱码

response.setHeader("content-type", "text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

out.write（）；

### 获得客户机请求头

getHeader(string name)方法:String   
　　getHeaders(String name)方法:Enumeration   
　　getHeaderNames()方法

response.setCharacterEncoding("UTF-8");//设置将字符以"UTF-8"编码输出到客户端浏览器

//通过设置响应头控制浏览器以UTF-8的编码显示数据

response.setHeader("content-type", "text/html;charset=UTF-8");

PrintWriter out = response.getWriter();

Enumeration<String> reqHeadInfos = request.getHeaderNames();//获取所有的请求头

while (reqHeadInfos.hasMoreElements()) {

String headName = (String) reqHeadInfos.nextElement();

String headValue = request.getHeader(headName);//根据请求头的名字获取对应的请求头的值

out.write(headName+":"+headValue);

out.write("<br/>");

}

String value = request.getHeader("Accept-Encoding");//获取Accept-Encoding请求头对应的值

out.write(value);

相同名字的多个请求头：

Enumeration<String> e = request.getHeaders("Accept-Encoding");

while (e.hasMoreElements()) {

String string = (String) e.nextElement();

System.out.println(string);

### 获得客户机请求参数(客户端提交的数据)

* getParameter(String)方法**(常用)**
* getParameterValues(String name)方法**(常用)**
* getParameterNames()方法(**不常用**)
* getParameterMap()方法**(编写框架时常用)**
  + 1. **form表单**

<body>

<fieldset style="width:500px;">

<legend>Html的Form表单元素</legend>

<!--form表单的action属性规定当提交表单时，向何处发送表单数据，method属性指明表单的提交方式，分为get和post，默认为get-->

<form action="/day6/servlet/RequestDemo4" method="post">

<!--输入文本框，SIZE表示显示长度，maxlength表示最多输入长度-->

编&nbsp;&nbsp;号(文本框)：<input type="text" name="userid" size="2" maxlength="2"><br>

<!--输入文本框，通过value指定其显示的默认值-->

用户名(文本框)：<input type="text" name="username"><br>

<!--密码框，其中所有输入的内容都以密文的形式显示-->

密&nbsp;&nbsp;码(密码框)：

<!--&nbsp;表示的是一个空格-->

<input type="password" name="userpass"><br>

<!--单选按钮，通过checked指定默认选中，名称必须一样，其中value为真正需要的内容-->

性&nbsp;&nbsp;别(单选框)： <input type="radio" name="sex" value="男" checked>男

<input type="radio" name="sex" value="女">女<br>

<!--下拉列表框，通过<option>元素指定下拉的选项，SELECTED默认-->

部&nbsp;&nbsp;门(下拉框)： <select name="dept">

<option value="技术部">技术部</option>

<option value="销售部" SELECTED>销售部</option>

<option value="财务部">财务部</option>

</select><br>

<!--复选框，可以同时选择多个选项，名称必须一样，其中value为真正需要的内容,checked默认-->

兴&nbsp;&nbsp;趣(复选框)： <input type="checkbox" name="inst" value="唱歌">唱歌

<input type="checkbox" name="inst" value="游泳">游泳 <input

type="checkbox" name="inst" value="跳舞">跳舞 <input

type="checkbox" name="inst" value="编程">编程 <input

type="checkbox" name="inst" value="上网">上网 <br>

<!--大文本输入框，宽度为34列，高度为5行-->

说&nbsp;&nbsp;明(文本域)：

<textarea name="note" cols="34" rows="5"></textarea>

<br>

<!--隐藏域，在页面上无法看到，专门用来传递参数或者保存参数-->

<input type="hidden" name="hiddenField" value="hiddenvalue" />

<!--提交表单按钮，当点击提交后，所有填写的表单内容都会被传输到服务器端-->

<input type="submit" value="提交)">

<!--重置表单按钮，当点击重置后，所有表单恢复原始显示内容-->

<input type="reset" value="重置">

</form>

<!--表单结束-->

</fieldset>

</body>

* + 1. 在Form表单中填写数据，然后提交到RequestDemo03这个Servlet进行处理
    2. **在服务器端使用getParameter方法和getParameterValues方法接收表单参数，代码如下：**

/\*\*

\* 编&nbsp;&nbsp;号(文本框)：

<input type="text" name="userid" value="NO." size="2" maxlength="2">

\*/

String userid = request.getParameter("userid");//获取填写的编号，userid是文本框的名字，<input type="text" name="userid">

/\*\*

\* 用户名(文本框)：<input type="text" name="username" value="请输入用户名">

\*/

String username = request.getParameter("username");//获取填写的用户名

/\*\*

\* 密&nbsp;&nbsp;码(密码框)：<input type="password" name="userpass" value="请输入密码">

\*/

String userpass = request.getParameter("userpass");//获取填写的密码

String sex = request.getParameter("sex");//获取选中的性别

String dept = request.getParameter("dept");//获取选中的部门

//获取选中的兴趣，因为可以选中多个值，所以获取到的值是一个字符串数组，因此需要使用getParameterValues方法来获取

String[] insts = request.getParameterValues("inst");

String note = request.getParameter("note");//获取填写的说明信息

String hiddenField = request.getParameter("hiddenField");//获取隐藏域的内容

String instStr="";

/\*\*

\* 获取数组数据的技巧，可以避免insts数组为null时引发的空指针异常错误！

\*/

for (int i = 0; insts!=null && i < insts.length; i++) {

if (i == insts.length-1) {

instStr+=insts[i];

}else {

instStr+=insts[i]+",";

}

}

String htmlStr = "<table>" +

"<tr><td>填写的编号：</td><td>{0}</td></tr>" +

"<tr><td>填写的用户名：</td><td>{1}</td></tr>" +

"<tr><td>填写的密码：</td><td>{2}</td></tr>" +

"<tr><td>选中的性别：</td><td>{3}</td></tr>" +

"<tr><td>选中的部门：</td><td>{4}</td></tr>" +

"<tr><td>选中的兴趣：</td><td>{5}</td></tr>" +

"<tr><td>填写的说明：</td><td>{6}</td></tr>" +

"<tr><td>隐藏域的内容：</td><td>{7}</td></tr>" +

"</table>";

htmlStr = MessageFormat.format(htmlStr,userid,username,userpass,sex,dept,instStr,note,hiddenField);

response.setCharacterEncoding("UTF-8");//设置服务器端以UTF-8编码输出数据到客户端

response.setContentType("text/html;charset=UTF-8");//设置客户端浏览器以UTF-8编码解析数据

response.getWriter().write(htmlStr);//输出htmlStr里面的内容到客户端浏览器显示

* + 1. **在服务器端使用getParameterNames方法接收表单参数，代码如下：**

Enumeration<String> paramNames = request.getParameterNames();//获取所有的参数名

while (paramNames.hasMoreElements()) {

String name = paramNames.nextElement();//得到参数名

String value = request.getParameter(name);//通过参数名获取对应的值

System.out.println(MessageFormat.format("{0}={1}", name,value));

}

**复选框的多个参数接受不全。**

* + 1. **在服务器端使用getParameterMap方法接收表单参数，代码如下：**

//request对象封装的参数是以Map的形式存储的

Map<String, String[]> paramMap = request.getParameterMap();

for(Map.Entry<String, String[]> entry :paramMap.entrySet()){

String paramName = entry.getKey();

String paramValue = "";

String[] paramValueArr = entry.getValue();

for (int i = 0; paramValueArr!=null && i < paramValueArr.length; i++) {

if (i == paramValueArr.length-1) {

paramValue+=paramValueArr[i];

}else {

paramValue+=paramValueArr[i]+",";

}

}

System.out.println(MessageFormat.format("{0}={1}", paramName,paramValue));

}

**值接受的全**

## request接收表单提交中文参数乱码问题

### get方式提交中文数据乱码产生的原因和解决办法

对于以get方式传输的数据，**request即使设置了以指定的编码接收数据也是无效的**(至于为什么无效我也没有弄明白)，默认的还是使用ISO8859-1这个字符编码来接收数据，客户端以UTF-8的编码传输数据到服务器端，而服务器端的request对象使用的是ISO8859-1这个字符编码来接收数据，**服务器和客户端沟通的编码不一致因此才会产生中文乱码的。**解决办法：**在接收到数据后，先获取request对象以ISO8859-1字符编码接收到的原始数据的字节数组，然后通过字节数组以指定的编码构建字符串，解决乱码问题。**

String name = request.getParameter("name");//接收数据

name =new String(name.getBytes("ISO8859-1"), "UTF-8") ;//获取request对象以ISO8859-1字符编码接收到的原始数据的字节数组，然后通过字节数组以指定的编码构建字符串，解决乱码问题

System.out.println("name："+name);

### 以超链接形式传递中文参数的乱码问题

客户端想传输数据到服务器，可以通过表单提交的形式，也可以通过超链接后面加参数的形式，例如：

1 <a href="${pageContext.request.contextPath}/servlet/RequestDemo05**?userName=gacl&name=徐达沛**">点击</a>

　　点击超链接，***数据是以get的方式传输到服务器***的，所以接收中文数据时也会产生中文乱码问题，而解决中文乱码问题的方式与上述的以get方式提交表单中文数据乱码处理问题的方式一致，如下所示：

1 String name = request.getParameter("name");

2 name =new String(name.getBytes("ISO8859-1"), "UTF-8");

　　另外，需要提的一点就是**URL地址后面如果跟了中文数据，那么中文参数最好使用URL编码进行处理**，如下所示：

1 <a href="${pageContext.request.contextPath}/servlet/RequestDemo05?userName=gacl&name=**<%=URLEncoder.encode("徐达沛", "UTF-8")**%>"**>点击</a>**

**jsp可以带Java。**

### 提交中文数据乱码问题总结

1、如果提交方式为post，想不乱码，只需要**在服务器端设置request对象的编码**即可，客户端以哪种编码提交的，**服务器端的request对象就以对应的编码接收**，比如客户端是以UTF-8编码提交的，那么服务器端request对象就以UTF-8编码接收(**request.setCharacterEncoding("UTF-8")**)

2、如果提交方式为get，设置request对象的编码是无效的，request对象还是以默认的ISO8859-1编码接收数据，因此要想不乱码，只能在接收到数据后再手工转换，步骤如下：

　　1).获取获取客户端提交上来的数据，得到的是乱码字符串,data="???è?????"

　　 String data = request.getParameter("paramName");

　　2).查找ISO8859-1码表，得到客户机提交的原始数据的字节数组

　　 byte[] source = data.getBytes("ISO8859-1");

　　3).通过字节数组以指定的编码构建字符串，解决乱码

　　 data = new String(source, "UTF-8");

通过字节数组以**指定的编码**构建字符串，这里**指定的编码**是根据客户端那边提交数据时使用的字符编码来定的，如果是GB2312，那么就设置成data = new String(source, "GB2312")，如果是UTF-8，那么就设置成data = new String(source, "UTF-8")

**Post与get**

首先我们来看一下post方法是怎么进行参数传递的。当我们**点击提交按钮的时候，数据封装进了Form Data中**，\*\*http请求中把实体主体带过去了【传输的数据称之为实体主体】，既然request对象封装了http请求，所以**request对象可以解析到发送过来的数据**，于是只要**把编码设置成UTF-8就可以解决乱码问题了**。

而get方式不同，它的**数据是从消息行带过去的，没有封装到request对象里面**，所以使用request设置编码是无效的。

## Request对象实现请求转发

### 请求转发的基本概念

**response的sendRedirect()可以实现重定向。**

**在讲ServletContext的时候，曾经说过Servlet之间可以通过ServletContext实现通讯，ServletContext也能称之为域对象。而request也可以称之为域对象，只不过ServletContext的域是整个web应用，而request的域仅仅代表一次http请求**

**我们可以使用ServletContext和request实现Servlet之间的通讯，那么我们用哪一种呢？一般的原则：可以使用request就尽可能使用request。因为ServletContext代表着整个web应用，使用ServletContext会消耗大量的资源，而request对象会随着请求的结束而结束，资源会被回收。使用request域进行Servlet之间的通讯在开发中是非常频繁的。**

**请求转发：指一个web资源收到客户端请求后，通知服务器去调用另外一个web资源进行处理。**  
　　请求转发的应用场景：MVC设计模式

　　在Servlet中实现请求转发的两种方式：

**1、通过ServletContext的getRequestDispatcher(String path)方法，该方法返回一个RequestDispatcher对象，调用这个对象的forward方法可以实现请求转发。**

**例如：将请求转发的test.jsp页面**

1 RequestDispatcher reqDispatcher =this.getServletContext().**getRequestDispatcher**("/test.jsp");

2 reqDispatcher.forward(request, response);

**2、通过request对象提供的getRequestDispatche(String path)方法，该方法返回一个RequestDispatcher对象，调用这个对象的forward方法可以实现请求转发。**

**例如：将请求转发的test.jsp页面**

1 request.**getRequestDispatcher**("/test.jsp").forward(request, response);

**request对象同时也是一个域对象(Map容器)，开发人员通过request对象在实现转发时，把数据通过request对象带给其它web资源处理。**

**带一个数据，最好用request，ServletContext是全部的，任意覆盖数据。**

String data="大家好，我是孤傲苍狼，我正在总结JavaWeb";

/\*\*

\* 将数据存放到request对象中,此时把request对象当作一个Map容器来使用

\*/

request.setAttribute("data", data);

//客户端访问RequestDemo06这个Servlet后，RequestDemo06通知服务器将请求转发(forward)到test.jsp页面进行处理

request.getRequestDispatcher("/test.jsp").forward(request, response);

**test.jsp：**

<body>

使用普通方式取出存储在request对象中的数据：

<h3 style="color:red;"><%=**(String)request.getAttribute("data")**%></h3>

使用EL表达式取出存储在request对象中的数据：

<h3 style="color:red;">${data}</h3>

</body>

　request对象作为一个域对象(Map容器)使用时，主要是通过以下的四个方法来操作

* setAttribute(String name,Object o)方法，将数据作为request对象的一个属性存放到request对象中，例如：request.setAttribute("data", data);
* getAttribute(String name)方法，获取request对象的name属性的属性值，例如：request.getAttribute("data")
* removeAttribute(String name)方法，移除request对象的name属性，例如：request.removeAttribute("data")
* getAttributeNames方法，获取request对象的所有属性名，返回的是一个，例如：Enumeration<String> attrNames = request.getAttributeNames();

### 请求重定向和请求转发的区别

**请求转发的细节**

1. 也就是说：**不要在转发之前写数据给浏览器。 forward方法将抛出IllegalStateException异常。**
2. 如果在调用forward方法之前向Servlet引擎的缓冲区中写入了内容，**只要写入到缓冲区中的内容还没有被真正输出到客户端**，forward方法就可以被正常执行，**原来写入到输出缓冲区中的内容将被清空**，但是，**已写入到HttpServletResponse对象中的响应头字段信息保持有效**。

**实际发生位置不同，地址栏不同**

　一个web资源收到客户端请求后，**通知服务器去调用另外一个web资源**进行处理，称之为**请求转发/307。**

request.getRequestDispatcher(path).forward(request, response);

*参数：*path，要跳转到的资源路径：**上下文路径 / 资源路径**

　　一个web资源收到客户端请求后，**通知浏览器去访问另外一个web资源**进行处理，称之为请求**重定向/302。**

**response.sendRedirect(String location);**

*参数：*location，转发到的资源路径

* 转发是发生在服务器的
  + **转发是由服务器进行跳转的**，**浏览器的地址栏是没有发生变化的**，在我访问Servlet111的时候，即使跳转到了Servlet222的页面，浏览器的地址还是Servlet111的。也就是说**浏览器是不知道该跳转的动作，转发是对浏览器透明的**。通过上面的转发时序图我们也可以发现，**实现转发只是一次的http请求**，一次转发中request和response对象都是同一个。这也解释了，为什么可以使用**request作为域对象进行Servlet之间的通讯。在请求对象request中，保存的对象对于每个信息资源是共享的。**
* 重定向是发生在浏览器的
  + **重定向是由浏览器进行跳转的**，进行重定向跳转的时候，**浏览器的地址会发生变化的**。曾经介绍过：实现重定向的原理是由response的状态码:**发送302状态码**和Location头组合而实现的。**这是由浏览器进行的页面跳转**实现重定向**会发出两个http请求**，\*\*request域对象是无效的，因为它不是同一个request对象
  + **转发是服务器跳转只能去往当前web应用的资源**

访问 WEB-INF 中的资源。**WEB-INF 文件夹是 Java Web 应用的默认安全目录**，即客户端无法直接访问，只有服务端可以访问的目录。

**如果想在页面中直接访问其中的文件，必须通过web.xml文件对要访问的文件进行相应映射才能访问。**  
注意：在实际的开发中，可以把不希望用户直接访问到（通过浏览器输入地址栏）的网页放在文件夹中通过此方式访问。

* + **重定向是服务器跳转，可以去往任何的资源**

**典型的应用场景：**

1. 转发: 访问 Servlet 处理业务逻辑，然后 forward 到 jsp 显示处理结果，**浏览器里 URL 不变**
2. 重定向: **提交表单，**处理成功后 redirect 到另一个 jsp，防止表单重复提交，浏览器里 URL 变了

### RequestDispatcher再说明

我们在写网页的时候，一般**网页的头部和尾部是不需要改变的。**如果我们**多个地方使用Servlet输出网头和网尾的话，需要把代码重新写一遍**。而使用RequestDispatcher的**include()方法就可以实现包含网头和网尾的效果了**。