Servlet常用对象API详解

# ServletConfig对象

## 怎样获取这个对象？

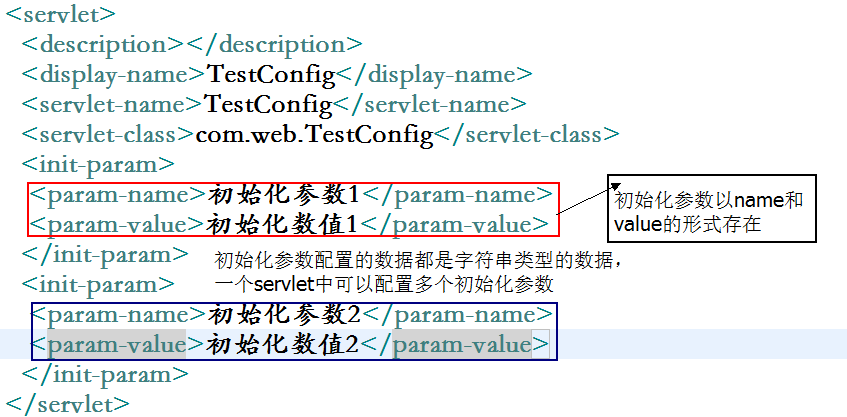
在servlet初始化的时候这个对象就被服务器传递给了servlet，所以可以从servlet中获取。

|  |
| --- |
| ServletConfig config = this.getServletConfig(); |

## 常用API

### 初始化参数的配置

**在WEB-INF下的web.xml文件中配置**



### 获取初始化参数-------获取指定名称的一个参数

|  |
| --- |
| //获取指定名称的初始化参数  String value1 = config.getInitParameter("初始化参数1"); |

### 获取初始化参数--------获取所配置的所有名称

|  |
| --- |
| // 获取当前servlet中所配置的所有的初始化参数名称  Enumeration<String> names = config.getInitParameterNames();  **while** (names.hasMoreElements()) {  // 获取的初始化参数的名字顺序不确定  String name = names.nextElement();  // 获取所对应的初始化参数名称  String value = config.getInitParameter(name);  System.***out***.println(name + ":" + value);  } |

### 获取当前servlet的名字

|  |
| --- |
| String servletName = config.getServletName(); |

### 获取ServletContext对象

|  |
| --- |
| ServletContext context = this.getServletConfig().getServletContext(); |

# ServletContext对象

## 怎样获取这个对象？

**1 通过ServletConfig对象获取**

|  |
| --- |
| ServletContext context = this.getServletConfig().getServletContext(); |

**2 整个程序应用一个**ServletContext对象,那么当前servlet也能获取这个对象

|  |
| --- |
| ServletContext context = this.getServletContext(); |

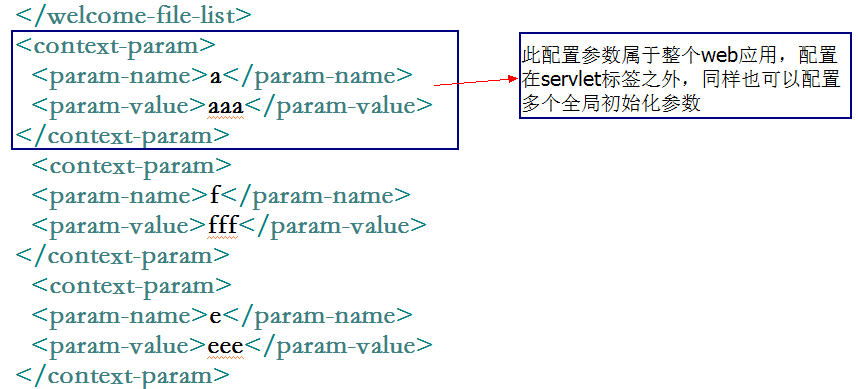
## 常用API

### 获取全局初始化参数

servletContext对象是Servlet三大域对象之一，每个Web应用程序都拥有一个ServletContext对象，该对象是Web应用程序的全局对象或者上下文。Tomcat服务器在启动时，会自动创建一个ServletContext对象，在关闭时，会自动销毁这个ServletContext对象。每个Web应用程序只拥有一个ServletContext对象，ServletContext对象可以在整个Web应用中共享数据资源。

#### 全局初始化参数的配置

**在WEB-INF下的web.xml文件中配置**



#### 获取全局初始化参数-------获取指定名称的参数

和config对象获取局部初始化参数的用法是一样的 ，区别在于：所有的servlet都能获取所有配置的初始化参数 。

|  |
| --- |
| String value1 = context.getInitParameter("a"); |

#### 获取全局初始化参数-------获取所配置的所有参数

|  |
| --- |
| // 获取所配置的所有的全局初始化参数名称  Enumeration<String> names = context.getInitParameterNames();  **while** (names.hasMoreElements()) {  // 获取全局初始化参数的名字顺序不确定  String name = names.nextElement();  // 获取所对应的全局初始化参数名称  String value = context.getInitParameter(name);  System.***out***.println(name + ":" + value);  } |

### 整个应用的数据共享（application域）

Application域中的数据会被整个web应用程序所共用。所有的servlet都能获取application域中的数据 。

**1向域中存数据**

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | void | **[setAttribute](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletContext.html" \l "setAttribute(java.lang.String, java.lang.Object))**(java.lang.String name, java.lang.Object object) | |

**2在域中取数据**

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | **java.lang.Object** | **[getAttribute](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletContext.html" \l "getAttribute(java.lang.String))(java.lang.String name)** | |
| |  |  | | --- | --- | | java.util.Enumeration | **[getAttributeNames](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletContext.html" \l "getAttributeNames())**() | |

**3 删除域中的指定数据**

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | void | **[removeAttribute](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletContext.html" \l "removeAttribute(java.lang.String))**(java.lang.String name) | |

### 获取web应用中文件的真实路径

**获取文件的真实路径 从而读取文件的内容**

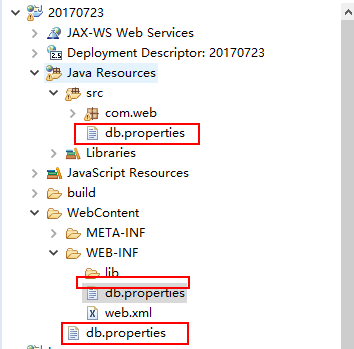
|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | java.lang.String | **[getRealPath](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletContext.html" \l "getRealPath(java.lang.String))**(java.lang.String path) | |

Web应用程序中的文件存在于三个位置：

第一种 src目录下的文件 /WEB-INF/classes/db.properties(src目录下的东西都发布在/WEB-INF/classes/目录下面)

第二种WebContent目录下的文件/db.properties(默认在根目录下)

第三种 WEB-INF目录下的文件/WEB-INF/db.properties



|  |
| --- |
| String realPath = context.getRealPath("/WEB-INF/db.properties");  FileInputStream inputStream = **new** FileInputStream(**new** File(realPath));  Properties p = **new** Properties();  p.load(inputStream); |

### 读取web资源文件

**ServletContext接口通过下列方法提供对web应用组成的静态内容文档层级的直接访问，包括HTML，GIF和JPEG文件：**

**getResource和getResourceAsStream方法以“/”开头的字符串为参数，它指定上下文根路径的资源相对路径。文档的层级可能存在于服务器的文件系统**

|  |  |
| --- | --- |
| java.net.URL | **[getResource](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletContext.html" \l "getResource(java.lang.String))**(java.lang.String path) |
| java.io.InputStream | **[getResourceAsStream](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletContext.html" \l "getResourceAsStream(java.lang.String))**(java.lang.String path) |

**Web应用中利用context对象总结读取资源文件的三种方法**

|  |
| --- |
| //第一种方式  URL url = context.getResource("WEB-INF/classes/db.properties");  InputStream is =  url.openStream();  //第二种方式   /\*读取db.properties文件\*/  String path =context.getRealPath("WEB-INF/classes/db.properties");   /\*根据文件的路径 构建文件对象\*/  File file = new File(path);   /\*根据file文件对象 创建输入流\*/  InputStream is = new FileInputStream(file);  //第三种方式 直接可以获取文件的输入流  InputStream is = context.getResourceAsStream("WEB-INF/classes/db.properties ") |

### 实现转发

|  |  |
| --- | --- |
| [RequestDispatcher](mk:@MSITStore:C:/jsdk23/javax/servlet/RequestDispatcher.html) | **[getRequestDispatcher](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletContext.html" \l "getRequestDispatcher(java.lang.String))**(java.lang.String path) |

先获取转发器---> 转发（携带数据）

|  |
| --- |
| context.getRequestDispatcher("/Servlet").forward(request, response); |

# HttpServletRrquest对象

  当客户端向服务器端发送请求时，服务器为本次请求创建request对象，并在调用Servlet的service方法时，将该对象传递给service方法。Request对象中封装了客户端发送过来的所有的请求数据。

## 常用的API

### 获取请求头信息

获取常用信息

2 String **getMethod（）**：获取客户端请求方式。例如：get或post

获取请求头信息

java.lang.String **getHeader(java.lang.String name)** 获取指定的请求头

java.util.Enumeration **getHeaderNames()** 获取此次请求的所有请求头的名字

java.util.Enumeration **getHeaders(java.lang.String name)**  获取指定请求头中多值

### 获取请求参数

通过request对象获取的页面表单传递的参数返回的都是字符串类型的数据，如果要将页面传递的数据封装到JavaBean中，必须进行数据格式的转换，但是使用阿帕奇提供的工具类只要保证参数的key和实体类中的属性名称对应一直则可以自动封装。

* getParameter(String)方法(常用)

|  |
| --- |
| String name = request.getParameter("name");//获取制定参数名称的数值 |

* getParameterValues(String name)方法(常用)

|  |
| --- |
| String[] hobby = request.getParameterValues("hobby");//获取像爱好这种一个key对应多个值得情况（复选框） |

* getParameterNames()方法(不常用)获取所有参数的名称
* getParameterMap()方法(编写框架时常用)次方法可以获取所有的请求数据）

|  |
| --- |
| Map<String, String[]> map = request.getParameterMap(); |

**结合工具类完成数据的自动封装**

**两个jar包缺一不可**

****

|  |
| --- |
| **User user = new User();**  **BeanUtils.populate(user, request.getParameterMap());** |

**难点总结：**

**页面中form表单和url或者超链接请求servlet传递的任何参数的数据类型都是字符串，request对象接收到请求参数的数据以后想要封装到实体bean中只能手动转换类型。但是使用工具类只要满足属性名称和传递参数的key一致即可。**

Servlet 接收传递过来的json数据，可以使用工具类接收。

### ****获取请求路径****

**Request对象通过以下方法来获取请求路径**。

  （1）String getServerName（）：获取服务器名：localhost

  （2）String getServerPort（）：获取服务器端口号：8080

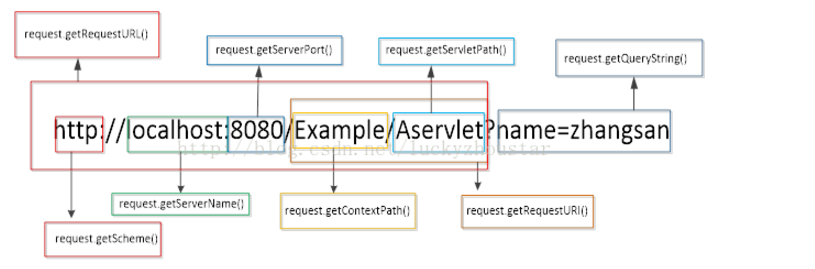
**（3）String getContextPath（）：获取项目名：Test**

  （4）String getServletPath（）：获取Servlet路径：/login

   （5）String getQueryString（）：获取参数部门，即问号后面的部分：username=zhangsan

  （6）String getRequestURL（）：获取请求URL

　（7）getRequestURI方法返回请求行中的资源名部分。  
　（8）getQueryString 方法返回请求行中的参数部分。



### 获取客户机的信息

|  |
| --- |
| String requestUrl = request.getRequestURL().toString();//得到请求的URL地址  String requestUri = request.getRequestURI();//得到请求的资源  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。  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 remoteUser = request.getRemoteUser();  Locale locale = request .getLocale();String string = locale.getDisplayCountry();//中国  String method = request.getMethod();//得到请求URL地址时使用的方法  　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。  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 |

### ****获取当前请求用户的所有cookie和session对象****

获取当前请求用户拥有的所有的cookie

|  |
| --- |
| Cookie[] cookies = request.getCookies(); |

获取当前会话的session

|  |
| --- |
| HttpSession session = request.getSession(); |

### ****作为请求域保存数据****

**相当一个Map容器来存储数据，一般通过转发来实现两个servlet的之间的数据共享**

|  |  |
| --- | --- |
| java.lang.Object | **[getAttribute](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletRequest.html" \l "getAttribute(java.lang.String))**(java.lang.String name)            Returns the value of the named attribute as an Object, or null if no attribute of the given name exists. |
| java.util.Enumeration | **[getAttributeNames](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletRequest.html" \l "getAttributeNames())**()            Returns an Enumeration containing the names of the attributes available to this request. |
| void | **[removeAttribute](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletRequest.html" \l "removeAttribute(java.lang.String))**(java.lang.String name)            Removes an attribute from this request. |
| void | **[setAttribute](mk:@MSITStore:C:/jsdk23/javax/servlet/ServletRequest.html" \l "setAttribute(java.lang.String, java.lang.Object))**(java.lang.String name, java.lang.Object o)            Stores an attribute in this request. |

保存数据的数据类型Object，可以是**任意数据类型**。域中的数据结合jsp可以动态的展示处理 。

### ****实现转发****

请求转发的基本概念

　　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。

**在Servlet中实现请求转发的两种方式：**

**1、通过ServletContext对象**

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

2 reqDispatcher.forward(request, response);

**2、通过request对象**

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。

**转发的特点：（307）**

**1 服务器内部的行为**

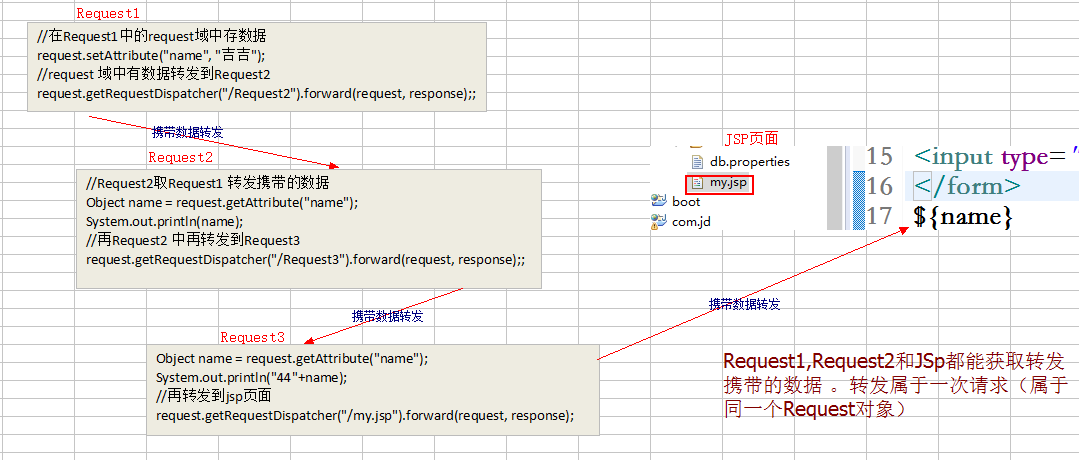
**2 属于同一次请求**

**3地址栏不会发生改变**

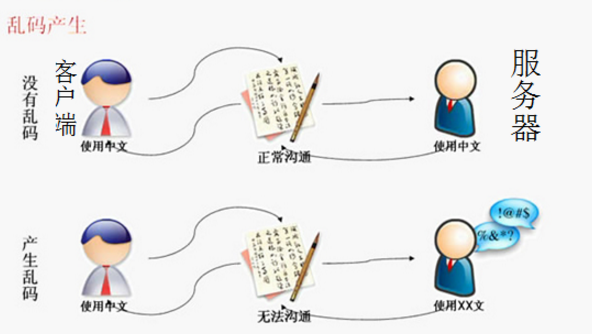
**4 转发是携带request对象中的数据**

**5转发可以直接转发到WEB-INF下面的资源（可以访问）**

**转发实际上属于一次请求，转发后的request还是原来的那个request，转发就可以形成一条转发链，从而实现数据的传递，由于request域中的数据类型任意，所以传递的数据类型就任意包括实体bean，集合。**



### 乱码问题



**产生乱码的原因**

**就是因为服务器和客户端沟通的编码不一致造成的，因此解决的办法是：在客户端和服务器之间设置一个统一的编码，之后就按照此编码进行数据的传输和接收。**

**1 Post方式解决乱码**

客户端是以UTF-8编码传输数据到服务器端的，所以需要设置服务器端以UTF-8的编码进行接收，否则对于中文数据就会产生乱码

|  |
| --- |
| **request.setCharacterEncoding("UTF-8");** |

**2 Get方式解决乱码**

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")** |

**3 超链接传参数解决乱码-----和get请求一样**

|  |
| --- |
| <a href="${pageContext.request.contextPath}/my/s/**?name=张三**">点击</a> |

**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**

|  |
| --- |
| <ahref="${pageContext.request.contextPath}//my/s/**?name=**=**<%=URLEncoder.encode("张三", "UTF-8")**%>">点击</a> |

# HttpServletResponse对象

**和request对象一样在servlet创建的时候这个对象就会被放在service方法的参数中 ，主要用来响应用户的请求 。**

## 常用API

### 操作响应内容

#### 向页面响应字符流（文本）

|  |
| --- |
| PrintWriter writer = response.getWriter();  writer.println("Hello1 ，吉吉 and 毛毛 ......");//不是换行  writer.print("Hello2 ，吉吉 and 毛毛 ......");  writer.write("最好只输出字符串"); |

共同点：两者都不刷新页面，只在原来的页面写数据.最终都是重写了抽象类Writer里面的write方法.   
print方法可以将各种类型的数据转换成字符串的形式输出。重载的write方法只能输出字符、字符数组、字符串等与字符相关的数据。

#### 向页面响应字节流

|  |
| --- |
| ServletOutputStream out = response.getOutputStream(); |

**使用字节流响应页面图片例子**

|  |
| --- |
| //响应字节流-----节流和字符流不能同时使用它们是互斥的  String realPath = **this**.getServletContext().getRealPath("/b.jpg");  FileInputStream in = **new** FileInputStream(**new** File(realPath));  response.setContentType("image/jpg");  **byte**[] b = **new** **byte**[1024];  **int** len = 0 ;  **while**((len=in.read(b))!=-1){  //响应字节流  response.getOutputStream().write(b, 0, len); |

节流和字符流不能同时使用它们是互斥的

|  |
| --- |
| Serlvet的service方法结束后，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。 |

### 操作响应元信息

#### response.setContentType()

**response.setContentType(MIME)的作用是使客户端浏览器，区分不同种类的数据，并根据不同的MIME调用浏览器内不同的程序嵌入模块来处理相应的数据。**

|  |
| --- |
| 例如web浏览器就是通过MIME类型来判断文件是GIF图片。通过MIME类型来处理json字符串。 Tomcat的安装目录\conf\web.xml 中就定义了大量MIME类型 ，可以参考。 response.setContentType("text/html; charset=utf-8"); html .setContentType("text/plain; charset=utf-8"); 文本 text/javascript json数据 application/xml  xml数据 |

这 个方法设置发送到客户端的响应的内容类型，此时响应还没有提交。给出的内容类型可以包括字符编码说明，例 如：**text/html;charset=UTF-8**.如果该方法在getWriter()方法被调用之前调用，那么响应的字符编码将仅从给出的内容类型 中设置。该方法如果在getWriter()方法被调用之后或者在被提交之后调用，将不会设置响应的字符编码，在使用http协议的情况中，该方法设 置 Content-type实体报头。  
一般在Servlet中，习惯性的会首先设置请求以及响应的内容类型以及编码方式：  
response.setContentType("text/html;charset=UTF-8");  
request.setCharacterEncoding("UTF-8");

**response.setContentType()的String参数及对应类型**

|  |
| --- |
| <option   value="image/bmp">BMP</option>    <option   value="image/gif">GIF</option>    <option   value="image/jpeg">JPEG</option>    <option   value="image/tiff">TIFF</option>    <option   value="image/x-dcx">DCX</option>    <option   value="image/x-pcx">PCX</option>    <option   value="text/html">HTML</option>    <option   value="text/plain">TXT</option>    <option   value="text/xml">XML</option>    <option   value="application/afp">AFP</option>    <option   value="application/pdf">PDF</option>    <option   value="application/rtf">RTF</option>    <option   value="application/msword">MSWORD</option>    <option   value="application/vnd.ms-excel">MSEXCEL</option>    <option   value="application/vnd.ms-powerpoint">MSPOWERPOINT</option>    <option   value="application/wordperfect5.1">WORDPERFECT</option>    <option   value="application/vnd.lotus-wordpro">WORDPRO</option>    <option   value="application/vnd.visio">VISIO</option>    <option   value="application/vnd.framemaker">FRAMEMAKER</option>    <option   value="application/vnd.lotus-1-2-3">LOTUS123</option> |

#### response.setStatus();  操作响应行信息

**通过设置响应头信息的location和状态码302实现重定向（原理）**

|  |
| --- |
| response.setStatus(302);  response.setHeader("location", "http://localhost:8080/20170723/my.jsp"); |

**其中常用的状态码的含义：**

|  |
| --- |
| 200：  表示请求处理完成。  302：  表示请求需要进一步细化。通常该状态码和location响应头结合使用。  404： 表示客户端错误，找不到资源。  500：  表示服务器错误。  其中不常用的也在此告诉大家：  |"100" ; 10.1.1节: 继续  |"101" ; 10.1.2节: 转换协议  |"200" ; 10.2.1节: OK  |"201" ; 10.2.2节: 已创建  |"202" ; 10.2.3节: 接受  |"203" ; 10.2.4节: 非权威信息  |"204" ; 10.2.5节: 无内容  |"205" ; 10.2.6节: 重置内容  |"206" ; 10.2.7节: 部分内容  |"300" ; 10.3.1节: 多个选择  |"301" ; 10.3.2节: 永久移动  |"302" ; 10.3.3节: 发现  |"303" ; 10.3.4节: 见其它  |"304" ; 10.3.5节: 没有被改变  |"305" ; 10.3.6节: 使用代理  |"307" ; 10.3.8节 临时重发  |"400" ; 10.4.1节: 坏请求  |"401" ; 10.4.2节: 未授权的  |"402" ; 10.4.3节: 必要的支付  |"403" ; 10.4.4节: 禁用  |"404" ; 10.4.5节: 没有找到  |"405" ; 10.4.6节: 方式不被允许  |"406" ; 10.4.7节: 不接受的  |"407" ; 10.4.8节: 需要代理验证  |"408" ; 10.4.9节： 请求超时  |"409" ; 10.4.10节； 冲突  |"410" ; 10.4.11节： 不存在  |"411" ; 10.4.12节： 长度必需  |"412" ； 10.4.13节；先决条件失败  |"413" ; 10.4.14节： 请求实体太大  |"414" ; 10.4.15节； 请求URI太大  |"415" ; 10.4.16节： 不被支持的媒体类型  |"416" ； 10.4.17节： 请求的范围不满足  |"417" ; 10.4.18节： 期望失败  |"500" ; 10.5.1节: 服务器内部错误  |"501" ; 10.5.2节: 不能实现  |"502" ; 10.5.3节: 坏网关  |"503" ; 10.5.4节: 服务不能获得  |"504" ; 10.5.5节: 网关超时  |"505" ; 10.5.6节: HTTP版本不支持 |

#### response.setHeader(name,value); 操作头信息

|  |  |
| --- | --- |
| **响应头** | **说明** |
| Allow | 服务器支持哪些请求方法（如GET、POST等）。 |
| Content-Encoding | 文档的编码（Encode）方法。只有在解码之后才可以得到Content-Type头指定的内容类型。利用gzip压缩文档能够显著地减少HTML文档的下载时间。Java的GZIPOutputStream可以很方便地进行gzip压缩，但只有Unix上的Netscape和Windows上的IE 4、IE 5才支持它。因此，Servlet应该通过查看Accept-Encoding头（即request.getHeader("Accept-Encoding")）检查浏览器是否支持gzip，为支持gzip的浏览器返回经gzip压缩的HTML页面，为其他浏览器返回普通页面。 |
| Content-Length | 表示内容长度。只有当浏览器使用持久HTTP连接时才需要这个数据。如果你想要利用持久连接的优势，可以把输出文档写入 ByteArrayOutputStream，完成后查看其大小，然后把该值放入Content-Length头，最后通过byteArrayStream.writeTo(response.getOutputStream()发送内容。 |
| Content-Type | 表示后面的文档属于什么MIME类型。Servlet默认为text/plain，但通常需要显式地指定为text/html。由于经常要设置Content-Type，因此HttpServletResponse提供了一个专用的方法setContentType。 |
| Date | 当前的GMT时间。你可以用setDateHeader来设置这个头以避免转换时间格式的麻烦。 |
| Expires | 应该在什么时候认为文档已经过期，从而不再缓存它？ |
| Last-Modified | 文档的最后改动时间。客户可以通过If-Modified-Since请求头提供一个日期，该请求将被视为一个条件GET，只有改动时间迟于指定时间的文档才会返回，否则返回一个304（Not Modified）状态。Last-Modified也可用setDateHeader方法来设置。 |
| Location | 表示客户应当到哪里去提取文档。Location通常不是直接设置的，而是通过HttpServletResponse的sendRedirect方法，该方法同时设置状态代码为302。 |
| Refresh | 表示浏览器应该在多少时间之后刷新文档，以秒计。除了刷新当前文档之外，你还可以通过setHeader("Refresh", "5; URL=http://host/path")让浏览器读取指定的页面。  注意这种功能通常是通过设置HTML页面HEAD区的＜META HTTP-EQUIV="Refresh" CONTENT="5;URL=http://host/path"＞实现，这是因为，自动刷新或重定向对于那些不能使用CGI或Servlet的HTML编写者十分重要。但是，对于Servlet来说，直接设置Refresh头更加方便。   注意Refresh的意义是"N秒之后刷新本页面或访问指定页面"，而不是"每隔N秒刷新本页面或访问指定页面"。因此，连续刷新要求每次都发送一个Refresh头，而发送204状态代码则可以阻止浏览器继续刷新，不管是使用Refresh头还是＜META HTTP-EQUIV="Refresh" ...＞。   注意Refresh头不属于HTTP 1.1正式规范的一部分，而是一个扩展，但Netscape和IE都支持它。 |
| Server | 服务器名字。Servlet一般不设置这个值，而是由Web服务器自己设置。 |
| Set-Cookie | 设置和页面关联的Cookie。Servlet不应使用response.setHeader("Set-Cookie", ...)，而是应使用HttpServletResponse提供的专用方法addCookie。参见下文有关Cookie设置的讨论。 |
| WWW-Authenticate | 客户应该在Authorization头中提供什么类型的授权信息？在包含401（Unauthorized）状态行的应答中这个头是必需的。例如，response.setHeader("WWW-Authenticate", "BASIC realm=＼"executives＼"")。  注意Servlet一般不进行这方面的处理，而是让Web服务器的专门机制来控制受密码保护页面的访问（例如.htaccess）。 |

三秒刷新页面

|  |
| --- |
| response.setHeader("Refresh", "3;http://localhost:8080/20170723/my.jsp"); |

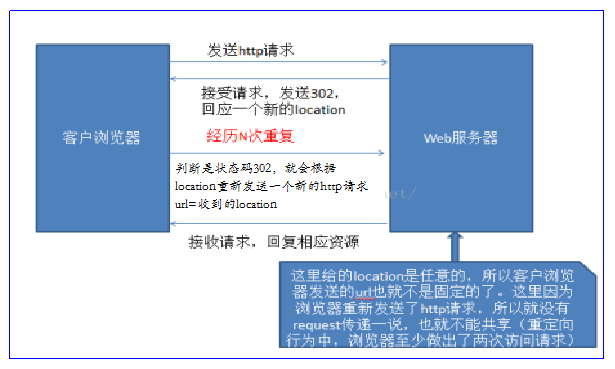
### 向特定的浏览器客户端发送Cookie

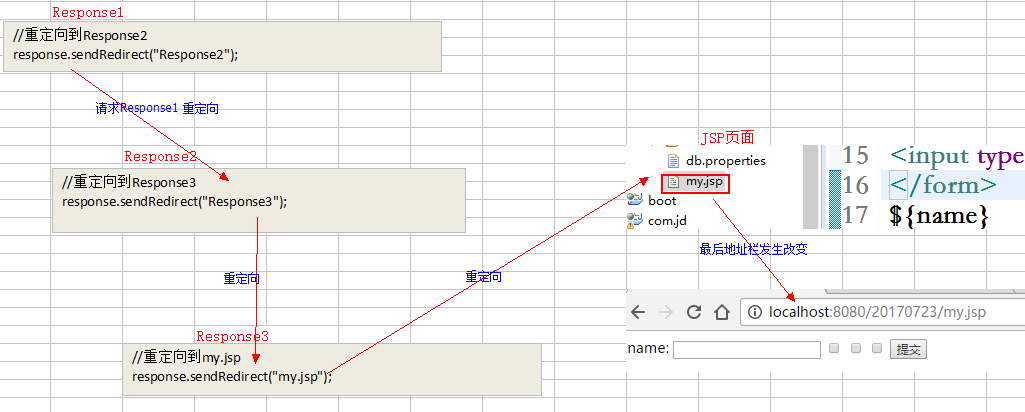
|  |
| --- |
| response.addCookie(new Cookie("","")); |

### 重定向

重定向特点：

* 客户端发送了两次请求,不会携带request对象中的数据
* 地址栏会发生改变
* 不能访问WEB-INF目录里的资源





**各种转发路径总结**

|  |
| --- |
| response.sendRedirect("/login.jsp")-->代表localhost:8080/login.jsp  response.sendRedirect("login.jsp")-->代表localhost:8080/Admin/login.jsp  response.sendRedirect("MyServlet")-->代表localhost:8080/Admin/MyServlet  response.sendRedirect("/MyServlet")-->代表localhost:8080/MyServlet  response.sendRedirect("/Exercise\_Jsp/login.jsp")-->代表localhost:8080/Exercise\_Jsp/login.jsp  response.sendRedirect("/Exercise\_Jsp/MyServlet")--代表localhost:8080/Exercise\_Jsp/MyServlet |

无论是转发还是重定向，转发和重定向后面不能再有转发和重定向相关的代码。（**转发和重定向在同一个Servlet中只能有一次。**）

|  |
| --- |
| **怎么选择是重定向还是转发呢？**  通常情况下**转发**更快，而且能保持request内的对象，所以它是第一选择。但是由于在转发之后，浏览器中URL仍然指向开始页面，此时如果重载当前页面，开始页面将会被重新调用。如果你不想看到这样的情况，则选择重定向（**页面刷新问题**）。  重定向：以前的request中存放的变量全部失效，并进入一个新的request作用域。  转发：以前的request中存放的变量不会失效，就像把两个页面拼到了一起。  1、 **如果是web应用之间资源的跳转，必须使用重定向。**  2、 **为了解决页面的刷新问题，必须使用重定向。**  **3、 如果在Servlet中向request对象中存储了一个数据，希望在下一个Servlet/JSP页面中把request对象中的数据取出来，这个时候必须使用转发，因为重定向是两次请求，request不能跨请求传递数据。其它情况都可以使用重定向。** |

【response处理中文乱码】

* 字节流：

设置浏览器默认打开编码

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

中文转成字节数组编码

response.getOutputStream().write("王守义".getBytes("UTF-8"));

* 字符流:

设置response的缓冲区的编码

response.setCharacterEncoding("UTF-8");

设置浏览器默认打开的编码.

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

response.setContentType("text/html;charset=UTF-8");相当于上面两句

# Cookie对象

Cookie对象由服务器的Servlet 创建，并且携带少量的信息随Response对象返回到客户端，由浏览器进行保存 。ookie的值可以唯一地标识客户机。服务器能根据cookie值中的JsessionID来判断一个Cookie来自于哪个用户。**如果浏览器不支持 Cookie 功能时，浏览器会将用户的 SessionCookieName 重写到用户请求的 URL 参数中它的传递格式如 /path/Servlet;name=value;name2=value2? Name3=value3，其中“Servlet；”后面的 K-V 对就是要传递的 Path Parameters，服务器会从这个Path Parameters 中拿到用户配置的SessionCookieName。**

## 常用API

### 创建Cookie

|  |
| --- |
| //创建Cookie对象 构造参数中有两个参数name/value  Cookie cookie = **new** Cookie("", ""); |

### Cookie的基本设置

|  |
| --- |
| //设置cookie的传递路径  cookie.setPath("/");  //设置cookie的最大过期时间  cookie.setMaxAge(60\*60);//int类型  cookie.setValue("newValue");//会覆盖原来数据值  **有set方法就有对应的getting方法** |

**cookie.setPath("")详解 能限制取cookie数据的应用**

|  |
| --- |
| 1. 可在同一应用服务器内共享方法：设置cookie.setPath("/");  本机tomcat/webapp下面有两个应用：webapp\_a和webapp\_b，  1）原来在webapp\_a下面设置的cookie，在webapp\_b下面获取不到，path默认是产生cookie的应用的路径。   2）若在webapp\_a下面设置cookie的时候，增加一条cookie.setPath("/");或者cookie.setPath("/webapp\_b/"); 就可以在webapp\_b下面获取到webapp\_a设置的cookie了。  3）此处的参数，是相对于应用服务器存放应用的文件夹的根目录而言的(比如tomcat下面的webapp)，因此cookie.setPath("/");之后，可以在webapp文件夹下的所有应用共享cookie，而cookie.setPath("/webapp\_b/"); 是指webapp\_a应用设置的cookie只能在webapp\_b应用下获得，即便是产生这个cookie的webapp\_a应用也不可以。  4）设置cookie.setPath("/webapp\_b/jsp")或者cookie.setPath("/webapp\_b/jsp/")的时候，**只有在webapp\_b/jsp下面可以获得cookie**，在webapp\_b下面但是在jsp文件夹外的都不能获得cookie。  5）设置cookie.setPath("/webapp\_b");，是指在webapp\_b下面才可以使用cookie，这样就不可以在产生cookie的应用webapp\_a下面获取cookie了  6）有多条cookie.setPath("XXX");语句的时候，起作用的以最后一条为准。 |

### Cookie不支持中文问题

中文和英文字符不同，中文属于Unicod字符，在内存中站4个字符，而英文属于ASCII字符，内存中只占2个字符。Cookie中使用Unicode字符时需要对Unicode字符进行编码，否则会乱码。编码使用java.net.URLEncoder类的encode（String str,String encoding）方法，解码使用java.net.URLDecoder类的decode(String str,String encoding)方法。

当存储中文的时候要将cookie发送到浏览器时会报数据非法异常，所以要先编码，再解码

**存中文数据---编码**

|  |
| --- |
| String str = "今天要下雨" ;  String nstr = URLEncoder.*encode*(str, "UTF-8");  Cookie cookie = **new** Cookie("name",nstr); |

**取中文数据---解码**

|  |
| --- |
| String value = cookie2.getValue();  String decode = URLDecoder.*decode*(value, "UTF-8");  response.getWriter().write(decode); |

### 将cookie返回到客户端

|  |
| --- |
| response.addCookie(cookie); |

# HttpSession对象

Session是一个接口，由服务器负责生成，将用户的信息保存在服务器。并且为每个用户生成一个唯一的识别的session id来区别用户 。客户端（浏览器）第一次访问servlet（服务器）并且调用request.getsession()时，为每个用户生成一个唯一的用户身份标识sessionId 号，返回给本地的浏览器，当客户端请求的时候携带sessionID，服务器就能知道数据是哪个浏览器的数据 。

**获取session----> session由服务器生成**

|  |
| --- |
| HttpSession session = request.getSession(); |

**获取SessionId**

|  |
| --- |
| String id = session.getId(); |

**设置session的最大过期时间**

|  |
| --- |
| session.setMaxInactiveInterval(3600); |

## session域

**Session作为一次会话范围的存储数据的域对象，内部维护一个map容器来存储数据。作用域的作用范围是一次会话 。**

**向session域中存数据**

|  |
| --- |
| session.setAttribute("name", new Object()); |

**在session域中取数据**

|  |
| --- |
| 取指定名称的数据  Object attribute = session.getAttribute("");  取所有数据的key（名称）  Enumeration<String> names = session.getAttributeNames(); |

**会话的作用范围是一次会话，只要访问的一个站点的浏览器没有全部关闭，这次会话就存在，那么session域中的数据就能取出 。**

|  |
| --- |
| long getCreationTime() 返回Session的创建日期。返回类型为long，常被转化为Date类型，例如：Date createTime = new Date(session.getCreationTime()) long getLastAccessedTime() 返回Session的最后活跃时间。返回类型为long |

## Session的持久化

### 为什么需要持久化

客户端访问了某个能开启会话功能的资源， web服务器就会创建一个与该客户端对应的HttpSession对象，每个HttpSession对象都要站用一定的内存空间。如果在某一时间段内访问站点的用户很多，**web服务器内存中就会积累大量的HttpSession对象**，消耗大量的服务器内存，即使用户已经离开或者关闭了浏览器，web服务器仍要保留与之对应的HttpSession对象，在他们超时之前，一直占用web服务器内存资源。

**web服务器通常将那些暂时不活动但未超时的HttpSession对象转移到文件系统或数据库中保存，服务器要使用他们时再将他们从文件系统或数据库中装载入内存，这种技术称为Session的持久化**

Session的持久化是由Session Manager来管理的。Tomcat提供了两个实现类：

org.apache.catalina.session.StandardManager  (默认)

如果突然终止该服务器，则所有会话都将丧失，因为StandardManager没有机会实现存盘处理。

**org.apache.catalina.session.PersistentManager（这个更高级）**

PersistentManager能够把Session对象保存到Session Store中，它提供了比 StandardManager更为灵活的Session管理功能，它具有以下功能：

  对内存中的HttpSession对象进行持久化，把它们保存到Session Store中。

  具有容错功能，可以及时把Session备份到Session Store中，当Tomcat服务器意外关闭后再重启时，可以从Session Store中恢复Session对象。

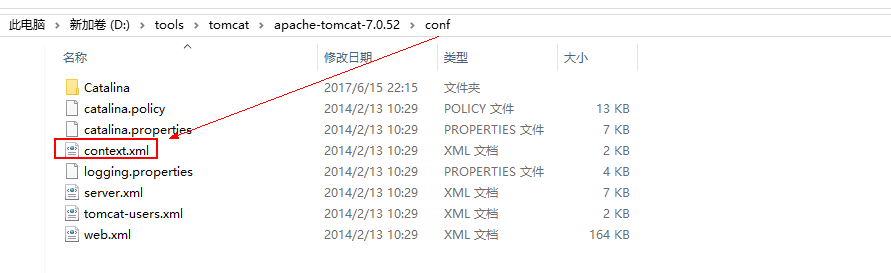
  可以灵活控制在内存中的Session数目，将部分Session转移到Session Store中。

通过使用Store,将内存中的session备份到文件或数据库中。当备份一个session对象时，该session对象会被复制到存储器中，而原对象仍然留在内存中。因此如果服务器崩溃，就可以从存储器中获取活动的session对象。当session被换出时，他会被移动到存储器中，因为当前活动的session对象超过了上限值，或者session对象闲置了过长时间。换出session节省内存空间。

### Session持久化的配置

http://blog.csdn.net/zhaky/article/details/50898496

**保存在服务器的磁盘上**



在<Context>标签里添加

|  |
| --- |
| <Manager className="org.apache.catalina.session.PersistentManager"  debug="0"  saveOnRestart="false"  maxActiveSession="-1"  minIdleSwap="-1"  maxIdleSwap="-1"  maxIdleBackup="-1">  <Store  className="org.apache.catalina.session.FileStore" directory="../session" />  </Manager> |

**参数说明：**

**maxActiveSessions－可处于活动状态的session数，default -1 不限制**

**checkInterval －检查session是否过期的时间间隔，default 60s**

**saveOnRestart－服务器关闭时，是否将所有的session保存到文件中；  
minIdleSwap/maxIdleSwap－session处于不活动状态最短/长时间(s)，sesson对象转移到File Store中；(－1表示没有限制)  
maxIdleBackup－超过这一时间，将session备份。(－1表示没有限制)**

**directory－文件存储位置work\Catalina\host name\web app\session\文件名.session**

**保存在数据库中需要配置store节点**

|  |
| --- |
| <Store  calssName="org.apache.catalina.JDBCStore"  driverName="com.mysql.jdbc.Driver"  connectionURL="jdbc:mysql://localhost/tomsessionDB?user=root&password="  sessionTable="tomcat\_session"  sessionIdCol="session\_id"  sessionDataCol="session\_data"  sessionValidCol="session\_valid"  sessionMaxInactiveCol="max\_inactive"  sessionLastAccessedCol="last\_access"  sessionAppCol="app\_name"  checkInterval="60"  debug="99" /> |