**Request**

**请求行** 请求方式 请求资源路径 HTTP协议及版本

String getMethod() 获取请求方式: GET POST

String getContextPath() 获取虚拟目录(项目访问路径): /request-demo

StringBuffer getRequestURL() 获取URL: http://localhost:8080/request-demo/req1

StringBuffer getRequestURI() 获取URI: /request-demo/req1

String getQueryString() 获取请求参数: username=zhangsan&password=123

**请求头** key: value

String getHeader(String name) 所以根据请求头名称获取对应值

客户端浏览器的版本信息 user-agent

**请求体**

前端发送字节数据，如文件 ServletInputStream getInputStream() 获取字节输入流，

如果前端发送的是纯文本数据 BufferedReader getReader()

PS：BufferedReader流是通过request对象来获取的，当请求完成后request对象就会被销毁， request对象被销毁后，BufferedReader流就会自动关闭，无需手动关闭

对于请求参数的获取,常用的有以下两种:

GET方式: String getQueryString() POST方式: BufferedReader getReader();

获取所有参数Map集合 Map<String,String[]> getParameterMap()

根据名称获取参数值（数组） String[] getParameterValues(String name)

根据名称获取参数值(单个值) String getParameter(String name)

POST 请求乱码

POST请求参数是通过request的getReader()来获取流中的数据，TOMCAT在获取流的时候采用的编码是ISO-8859-1

解决： 把TOMCAT在获取流数据之前的编码设置为UTF-8 request.setCharacterEncoding("UTF-8")

GET 请求乱码

浏览器把中文参数按照UTF-8进行URL编码Tomcat对获取到的内容进行了ISO-8859-1的URL解码

URL编码

String encode = URLEncoder.encode(username, "utf-8");

System.out.println(encode); //打印:%E5%BC%A0%E4%B8%89

URL解码

String decode = URLDecoder.decode(encode, "utf-8"); //打印:张三

String decode = URLDecoder.decode(encode, "ISO-8859-1"); //打印:`å¼ ä¸ `

解决： 先把ISO-8859-1转化成字节，再把他转成字符串

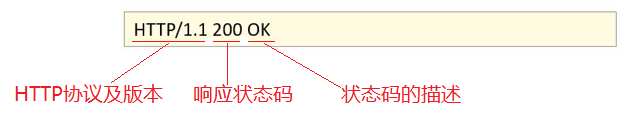
// byte[] bytes = username.getBytes(StandardCharsets.ISO\_8859\_1);

// username = new String(bytes, StandardCharsets.UTF\_8);

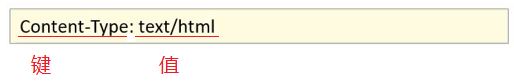
username = new

String(username.getBytes(StandardCharsets.ISO\_8859\_1),StandardCharsets.UTF\_8)

Response设置响应数据

**响应行**

设置响应状态码: void setStatus(int sc);



**响应头**

设置响应头键值对： void setHeader(String name,String value);

**响应体**

对于响应体，是通过字符、字节输出流的方式往浏览器写，

获取字符输出流: PrintWriter getWriter();

获取字节输出流: ServletOutputStream getOutputStream();

**请求转发forward**

req.getRequestDispatcher("资源B路径").forward(req,resp);

**Respones请求重定向**

Response重定向(redirect):一种资源跳转方式。

(1)浏览器发送请求给服务器，服务器中对应的资源A接收到请求

(2)资源A无法处理该请求，给浏览器响应302状态码+location的一个访问资源B的路径

(3)浏览器接收到302状态码就会重新发送请求到location对应的访问地址去访问资源B

(4)资源B接收到请求后进行处理并最终给浏览器响应结果，这整个过程就叫重定向

// resp.setStatus(302); resp.setHeader("location","资源B的访问路径");

resposne.sendRedirect("/request-demo/resp2");

共享数据:

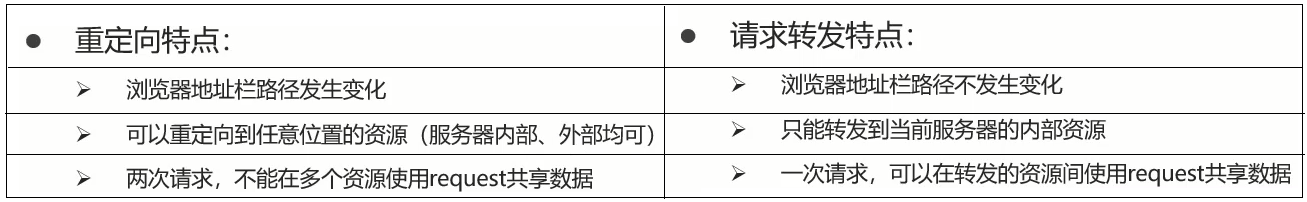
重定向：不方便在重定向过程中共享数据，因为它是两个独立的请求。通常需要使用会话（Session）或URL参数来传递数据。

请求转发：非常容易在请求转发过程中共享数据，因为资源之间可以直接共享相同的请求对象、请求属性和会话数据。

性能开销:

重定向：通常比请求转发更耗性能，因为它涉及两个独立的请求-响应周期。

请求转发：性能开销较小，因为它只涉及一个请求-响应周期。

对于转发来说，在服务端进行，不要加虚拟目录

对于重定向来说，路径最终是由浏览器来发送请求，就需要添加虚拟目录

浏览器使用:需要加虚拟目录(项目访问路径) 服务端使用:不需要加虚拟目录

**Response响应字符数据**

通过Response对象获取字符输出流： PrintWriter writer = resp.getWriter();

通过字符输出流写数据: writer.write("aaa");

设置响应的数据格式及数据的编码 可以识别html5语句

response.setContentType("text/html;charset=utf-8");

writer.write(<h1>"你好"<h1>);

**Response响应字节数据**

读取文件

FileInputStream fis = new FileInputStream("d://a.jpg");

获取response字节输出流

ServletOutputStream os = response.getOutputStream();

完成流的copy

byte[] buff = new byte[1024];

int len = 0;

while ((len = fis.read(buff))!= -1){

os.write(buff,0,len);

}

fis.close();

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.6</version>

</dependency>

fis:输入流

os:输出流

IOUtils.copy(fis,os);

//1. 读取文件

FileInputStream fis = new FileInputStream("d://a.jpg");

//2. 获取response字节输出流

ServletOutputStream os = response.getOutputStream();

//3. 完成流的copy

IOUtils.copy(fis,os);

fis.close();

**JSP（本质上就是个servlet）**

导maven,不导好像也没事

<dependency>

<groupId>javax.servlet</groupId>

<artifactId>javax.servlet-api</artifactId>

<version>3.1.0</version>

<scope>provided</scope>

</dependency>

**JSP脚本**

<%...%>：内容会直接放到\_jspService()方法之中

<%=…%>：内容会放到out.print()中，作为out.print()的参数

<%!…%>：内容会放到\_jspService()方法之外，被类直接包含

<%

for (int i = 0; i < brands.size(); i++) {

//获取集合中的 每一个 Brand 对象

Brand brand = brands.get(i);

%>

<tr align="center">

<td><%=brand.getId()%></td>

<td><%=brand.getBrandName()%></td>

<td><%=brand.getCompanyName()%></td>

<td><%=brand.getOrdered()%></td>

<td><%=brand.getDescription()%></td>

<td><%=brand.getStatus() == 1 ? "启用":"禁用"%></td>

<td><a href="#">修改</a> <a href="#">删除</a></td>

</tr>

<%

}

%>

Cookie是存储在浏览器端 Session是存储 在服务器端

**Cookie**

创建Cookie设置数据 Cookie cookie = new Cookie("key","value");

发送Cookie到客户端,用response对象 response.addCookie(cookie);

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

cookie.getName(); cookie.getValue();

设置Cookie存活时间 setMaxAge(int seconds)

1.正数：将Cookie写入浏览器所在电脑的硬盘，持久化存储。到时间自动删除

2.负数：默认值，Cookie在当前浏览器内存中，当浏览器关闭，则Cookie被销毁

3.零：删除对应Cookie

设置Cookie 存储中文

String value = "张三";

//对中文进行URL编码

value = URLEncoder.encode(value, "UTF-8");

System.out.println("存储数据："+value);

//将编码后的值存入Cookie中

Cookie cookie = new Cookie("username",value);

//设置存活时间 ，1周 7天

cookie.setMaxAge(60\*60\*24\*7);

//2. 发送Cookie，response

response.addCookie(cookie);

**Session**

获取Session对象 HttpSession session = request.getSession();

Session对象提供的功能

存储数据到 session 域中 void setAttribute(String name, Object o)

根据 key获取值 Object getAttribute(String name)

根据 key，删除该键值对 void removeAttribute(String name)

**Session的钝化和活化**

服务器是正常关闭和启动，session中的数据是可以被保存下来的

钝化：在服务器正常关闭后，Tomcat会自动将Session数据写入硬盘的文件中

钝化的数据路径为:项目目录\target\tomcat\work\Tomcat\localhost\项目名称 \SESSIONS.ser

活化：再次启动服务器后，从文件中加载数据到Session中

数据加载到Session中后，路径中的SESSIONS.ser文件会被删除掉

**Session的销毁**

默认30分钟

<?xml version="1.0" encoding="UTF-8"?>

<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee

http://xmlns.jcp.org/xml/ns/javaee/web-app\_3\_1.xsd"

version="3.1">

<session-config>

<session-timeout>100</session-timeout>

</session-config>

</web-app>

手动销毁 session.invalidate();

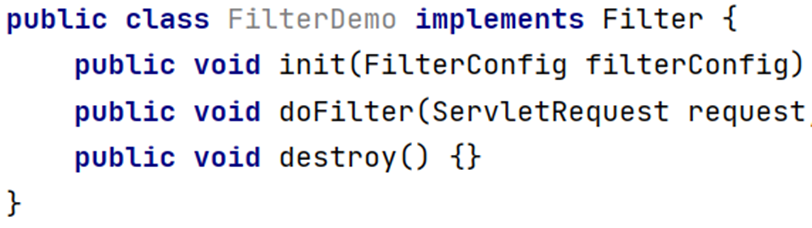
| **特性** | **Cookie** | **Session** |
| --- | --- | --- |
| 存储位置 | 客户端 | 服务端 |
| 安全性 | 不安全 | 安全 |
| 数据大小 | 最大3KB | 无大小限制 |
| 存储时间 | 可以通过setMaxAge()长期存储 | 默认30分钟 |
| 服务器性能 | 不占用服务器资源 | 占用服务器资源 |

| **应用场景** | **使用Cookie**  **的应用场景** | **使用Session**  **的应用场景** |
| --- | --- | --- |
| 购物车 | 是（存储购物车信息） | 否（购物车通常基于用户的会话） |
| 以登录用户的名称展示 | 否（通常需要用户登录信息） | 是（存储用户登录状态和信息） |
| 记住我功能 | 是（用于保持用户登录状态） | 否（通常用于记住登录状态） |
| 验证码 | 否（通常存储在Session中） | 是（存储验证码的临时数据） |

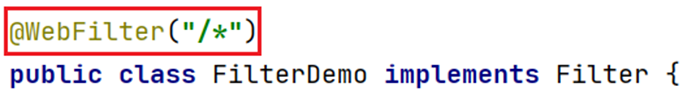
**Filter**

1.2.1 开发步骤

定义类，实现 Filter接口，并重写其所有方法



配置Filter拦截资源的路径：在类上定义 @WebFilter 注解。而注解的 value 属性值 /\* 表示拦截所有的资源



**放行**

chain.doFilter(request,response);

public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain) throws IOException, ServletException {

System.out.println("1.FilterDemo...");

//放行

chain.doFilter(request,response);

}

拦截具体的资源：/index.jsp：只有访问index.jsp时才会被拦截

目录拦截：/user/\*：访问/user下的所有资源，都会被拦截

后缀名拦截：\*.jsp：访问后缀名为jsp的资源，都会被拦截

拦截所有：/\*：访问所有资源，都会被拦截

**过滤器链**

我们现在使用的是注解配置Filter，而这种配置方式的优先级是按照过滤器类名(字符串)的自然排序。

比如有如下两个名称的过滤器 ： BFilterDemo 和 AFilterDemo 。那一定是 AFilterDemo 过滤器先执行。

**Listener**

