# Web基础学习

## 1.Servlet

### 什么是servlet

* 从广义上来讲，Servlet规范是Sun公司制定的一套技术标准，包含与Web应用相关的一系列接口，是Web应用实现方式的宏观解决方案。而具体的Servlet容器负责提供标准的实现。
* 从狭义上来讲，Servlet指的是javax.servlet.Servlet接口及其子接口，也可以指实现了Servlet接口的实现类。
* Servlet作为服务器端的一个组件，它的本意是“服务器端的小程序”。Servlet的实现对象由Servlet容器负责创建；Servlet的方法由容器在特定情况下调用；Servlet容器会在Web应用卸载时销毁Servlet对象的实例。

### Servlet的作用

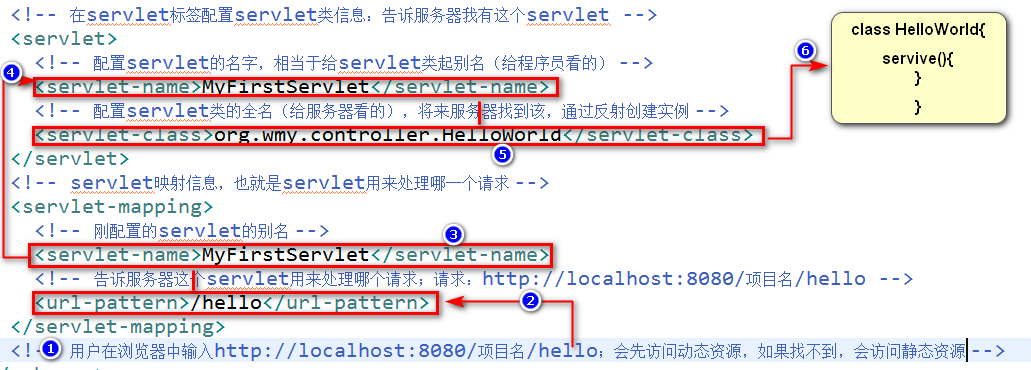
* 接收请求
* 处理请求
* 完成响应

Servlet是运行在Web服务器上的小型java程序。通过Http协议接收和响应来自web客户端的请求。

### HelloWorld

1. 创建自己的HelloServlet，实现Servlet接口
2. 实现servert方法
3. 在web.xml中配置server信息

流程如下：



匹配的优先级是：先匹配动态资源，在匹配静态资源；（老师强调了）

### Servlet的生命周期

Servlet 是跑在Tomcat 服务器上的；Tomcat服务器是servlet 的容器。

Servlet 的生命周期：servlet 从创建到销毁的过程

|  |
| --- |
| **package** org.wmy.controller;  **import** java.io.IOException;  **import** javax.servlet.Servlet;  **import** javax.servlet.ServletConfig;  **import** javax.servlet.ServletException;  **import** javax.servlet.ServletRequest;  **import** javax.servlet.ServletResponse;  **public** **class** HelloServlet **implements** Servlet{  @Override  **public** **void** destroy() {  // **TODO** Auto-generated method stub  }  @Override  **public** ServletConfig getServletConfig() {  // **TODO** Auto-generated method stub  **return** **null**;  }  /\*  \* 获得servlet的描述信息，该方法不常用，一般返回类名  \*/  @Override  **public** String getServletInfo() {  **return** "HelloWorld";  }  @Override  **public** **void** init(ServletConfig arg0) **throws** ServletException {  // **TODO** Auto-generated method stub    }  @Override  **public** **void** service(ServletRequest arg0, ServletResponse arg1) **throws** ServletException, IOException {  // **TODO** Auto-generated method stub    }  } |

当第一次访问HelloServlet时：

1. 创建一个Servlet对象（调用一次）
2. 调用Init方法（调用一次） 该方法初始化servlet
3. 调用Service方法 整个运行期间只创建了一个servlet对象（单实例，多线程运行）

以后的请求：

1. 只调用service方法

当项目从服务器上卸载或服务器停止时：

1. 服务器会调用Destroy()方法（servlet的销毁方法，清理善后）

### ServletConfig

在web.xml中配置的如下信息，会被服务器封装成ServletConfig对象，在初始化的时候传递到init函数

|  |
| --- |
| <servlet>  <servlet-name>MyFirstServlet</servlet-name>  <servlet-class>org.wmy.controller.HelloWorld</servlet-class>  <!-- 配置servlet的初始化参数 -->  <init-param>  <param-name>user</param-name>  <param-value>wangmingyang</param-value>  </init-param>  </servlet>  <servlet-mapping>  <servlet-name>MyFirstServlet</servlet-name>  <url-pattern>/hello</url-pattern>  </servlet-mapping> |

Servletconfig的功能：

* 获取servlet的别名
* 获取servlet 的初始化参数
* 获取ServletContext对象，代表当前servlet的上下文

|  |
| --- |
| **public** **class** HelloWorld **implements** Servlet{    ServletConfig servletConfig=**null**;  /\*  \* 该方法由服务器调用，config由服务器传递进去  \* ServletConfig：封装了servlet配置信息的对象，一个Servlet对应一个ServletConfig；  \* 封装的是当前 servlet 的配置信息  \*/  @Override  **public** **void** init(ServletConfig config) **throws** ServletException {  **this**.servletConfig=config;  }  @Override  **public** **void** service(ServletRequest req, ServletResponse res) **throws** ServletException, IOException {  //1.获取servlet的别名  String servletName = servletConfig.getServletName();  System.***out***.println(servletName);    //2.获取servlet 的初始化参数  String initParameter = servletConfig.getInitParameter("user");  System.***out***.println(initParameter);    //3.获取ServletContext对象，代表当前servlet的上下文  //代表我当前web项目信息(一个项目唯一一个 ServletContext)  ServletContext context = servletConfig.getServletContext();  }  } |

### ServletContext

研究ServletContext对象

1.一个web 应用对应一个ServletContext，代表整个web项目

2.功能

1）可以获取web项目的配置信息，获取web项目的初始化参数

2）获取web的项目路径

3）获取资源的真实路径

虚拟路径：网络访问使用虚拟路径，每一个虚拟路径对应一个实际的资源

静态资源（文件的形式），动态资源（只是启动一段程序代码）

真实路径：文件在磁盘中存储的路径

4）可以作为最大的域对象（application）共享数据(有4个域对象)

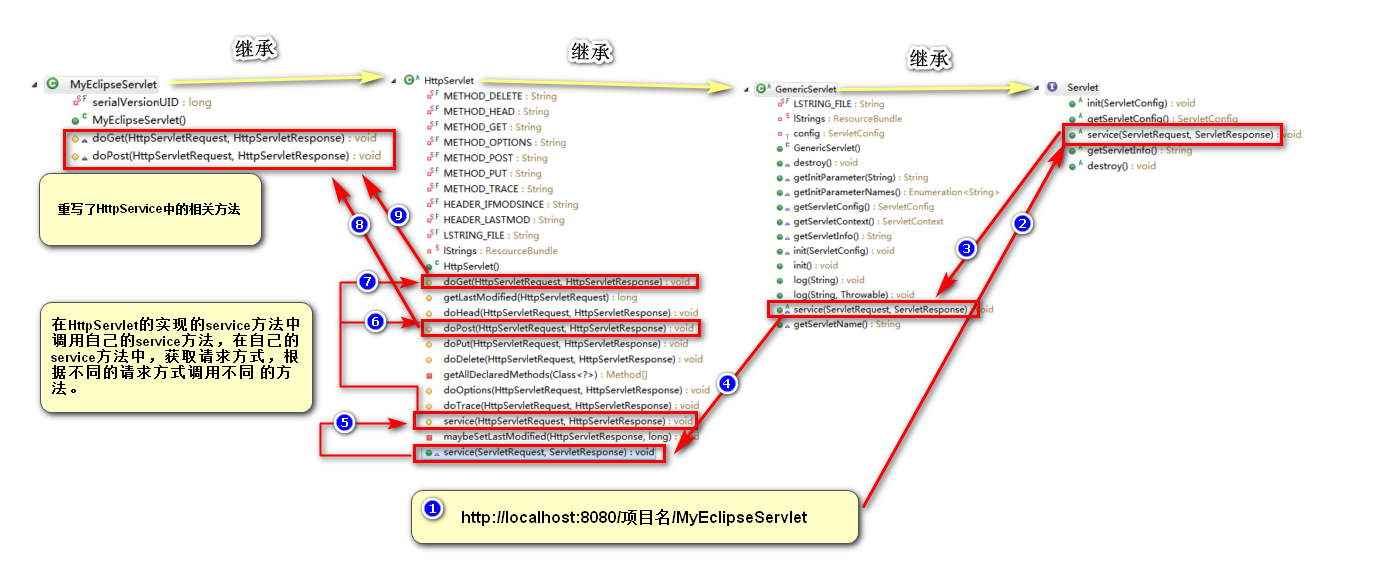
域对象的作用：共享数据

|  |
| --- |
| <!-- 配置 web 的初始化参数 -->  <context-param>  <param-name>user</param-name>  <param-value>wmy</param-value>  </context-param>    <servlet>  <servlet-name>ContextServlet</servlet-name>  <servlet-class>org.wmy.controller.ContextServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>ContextServlet</servlet-name>  <url-pattern>/contextServlet</url-pattern>  </servlet-mapping> |

|  |
| --- |
| **public** **class** ContextServlet **implements** Servlet {  **private** ServletConfig servletConfig;  @Override  **public** ServletConfig getServletConfig() {  **return** servletConfig;  }  @Override  **public** String getServletInfo() {  **return** "ContextServlet";  }  @Override  **public** **void** init(ServletConfig servletConfig)  **throws** ServletException {  **this**.servletConfig=servletConfig;  }  /\*  \* 研究ServletContext对象  \* 1.一个web 应用对应一个ServletContext，代表整个web项目  \* 2.功能  \* 1）可以获取web项目的配置信息，获取web项目的初始化参数  \* 2）获取web的项目路径  \* 3）获取资源的真实路径  \* 虚拟路径：网络访问使用虚拟路径，每一个虚拟路径对应一个实际的资源  \* 静态资源（文件的形式），动态资源（只是启动一段程序代码）  \* 真实路径：文件在磁盘中存储的路径  \* 4）可以作为最大的域对象（application）共享数据(有4个域对象)  \* 域对象的作用：共享数据  \*/  @Override  **public** **void** service(ServletRequest req, ServletResponse res)  **throws** ServletException, IOException {  ServletConfig config = **this**.getServletConfig();  ServletContext context = config.getServletContext();  //获取web项目的初始化参数  String string = context.getInitParameter("user");  System.***out***.println(string);    //获取web的项目路径  String path = context.getContextPath();  System.***out***.println(path);    //获取资源的真实路径  String realPath = context.getRealPath("index.html");  System.***out***.println(realPath);    //可以作为最大的域对象共享数据    }  }  wmy  /01JavaWeb  E:\workspace-sts-3.7.3.RELEASE\.metadata\.plugins\org.eclipse.wst.server.core\tmp0\wtpwebapps\01JavaWeb\index.html |

### 继承HttpServlet实现自己的servlet

继承体系：



### HttpServletResponsed的常见用法

* 1.可以给浏览器响应字符串
* 2.可以重定向到一个页面或其它资源。重定向就是服务器告诉浏览器重新请求别的资源

|  |
| --- |
| **public** **class** MyEclipseServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  //一个请求对应一个响应  //1.可以给浏览器响应字符串  PrintWriter writer = response.getWriter();  writer.write("hello");    //2.可以重定向到一个页面或其它资源。重定向就是服务器告诉浏览器重新请求别的资源  //以下写法是相对路径，代表相对于当前资源；  //当前资源：http://localhost:8080/项目名/myservlet01  //所以，要重定向的资源是http://localhost:8080/项目名/success.html  response.sendRedirect("success.html");//执行完，代表已经对之前的请求响应完毕  }  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  // **TODO** Auto-generated method stub  doGet(request, response);  }  } |

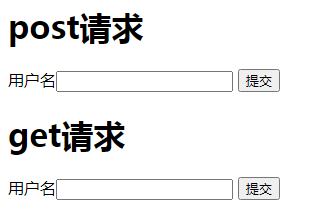
### HttpServletRequest

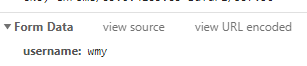
* HttpServletRequest 代表浏览器发送给服务器的请求信息
* http请求包含：请求首行 请求头 请求体（封装的请求数据-post）
* get 请求是将所有的请求数据放到url中

作用：

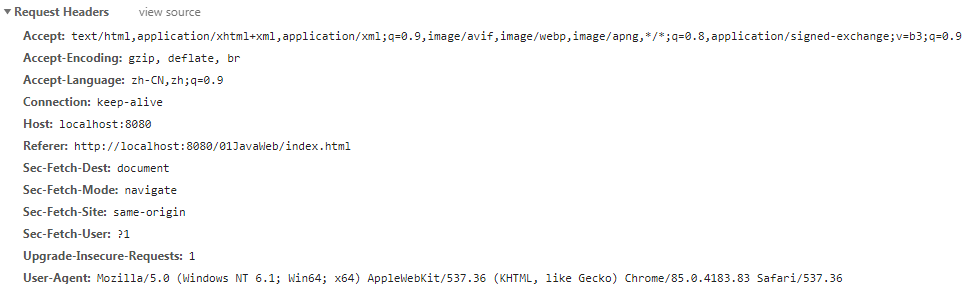
1. 获得请求数据 (get放在url后面；post放在请求体里，无论哪种方式都能获得)
2. 获得请求头
3. 转发一个页面/资源
4. 作为域对象共享数据

|  |
| --- |
| <body>  <h1>post请求</h1>  <form action=*"MyEclipse"* method=*"post"*>  用户名<input type=*"text"* name=*"username"*/>  <input type=*"submit"*/>  </form>    <h1>get请求</h1>  <form action=*"MyEclipse"* method=*"get"*>  用户名<input type=*"text"* name=*"username"*/>  <input type=*"submit"*/>  </form>    </body> |
| **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  //1.获得请求数据 (get放在url后面；post放在请求体里，无论哪种方式都能获得)  String username = request.getParameter("username");  //如果一个key有多个值，比如复选框等，使用如下方式  String[] values = request.getParameterValues("aihao");    //2.获得请求头  String header = request.getHeader("User-Agent");    //3.转发一个页面/资源  //先获取一个转发器  RequestDispatcher dispatcher = request.getRequestDispatcher("success.html");  //将请求转发出去  dispatcher.forward(request, response);    //4，作为域对象共享数据  } |









### 转发和重定向的区别：

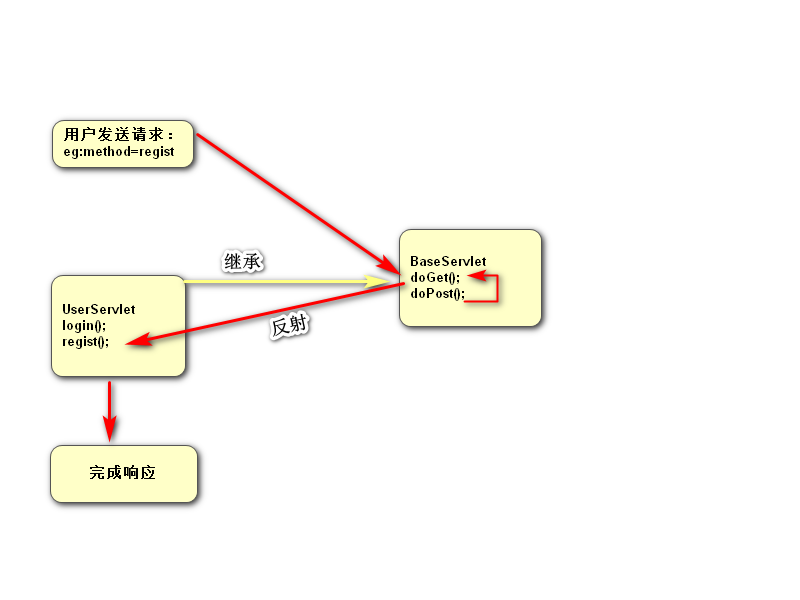
|  |  |  |
| --- | --- | --- |
|  | **转发** | **重定向** |
| **浏览器地址栏** | 不会变化 | 会 |
| **Request** | 同一个请求 | 两次请求 |
| **API** | Request对象 | Response对象 |
| **位置** | 服务器内部完成 | 浏览器完成 |
| **WEB-INF** | 可以访问 | 不能访问 |
| **共享请求域数据** | 可以共享 | 不可以共享 |
| **目标资源** | 必须是当前Web应用中的资源 | 不局限于当前Web应用 |

## BaseServlet

BaseServlet的作用是，利用反射动态的调用方法，处理跟某一模块相关的逻辑，这样只需要在具体的servlet中编写业务；另外，该方式减少了servlet的数量，使得程序模块化。

利用反射动态调用

|  |
| --- |
| **public** **class** BaseServlet **extends** HttpServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;      **protected** **void** doGet(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  String method = request.getParameter("method");  **try** {  Method invokedMethod = **this**.getClass().getDeclaredMethod(method,  HttpServletRequest.**class**, HttpServletResponse.**class**);  //把方法权限放大  invokedMethod.setAccessible(**true**);  //对象；参数  invokedMethod.invoke(**this**, request,response);  } **catch** (Exception e) {  e.printStackTrace();  }  }  **protected** **void** doPost(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  doGet(request, response);  }  }  **public** **class** UserServlet **extends** BaseServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    //处理用户注册的  **protected** **void** regist(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  //业务逻辑......  }  //处理用户登录的  **protected** **void** login(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  //业务逻辑......  }  } |



## Cookie

### 什么是cookie？

* 服务器发给浏览器保存的数据。浏览器保存相关数据
* Http是无状态协议：

无状态：服务器无法分辨每次的请求来自哪个客户端

如果客户端来的时候能带上类似会员卡的东西，记录有卡号。就可以辨别了。

1. 创建

Cookie创建发送给浏览器后，响应头会有一行：Set-Cookie:username=wmy

该响应头就是命令浏览器保存一个cookie：username=wmy

浏览器保存的这个字段，默认在一个会话期间只要访问这个项目，该cookie都在

保存在浏览器中的cookie ，以后只要是请求当前项目下的资源都会携带（一个session期间）

1. 获取

Cookie[] cookies = request.getCookies();

1. 删除；cookie 的生命周期
2. 默认cookie是在会话期间有效（浏览器一直不关）
3. Cookie可以修改默认的存活时间

cookie.serMaxAge(0);

Set-Cookie: username=wmy; Expires=Thu, 01-Jan-1970 00:00:10 GMT

1. 持久化Cookie

cookie.setMaxAge(60\*60)：表示cookie一小时后才会过期，无论浏览器关闭与否（浏览 器自己删除）

1. 设置cookie路径：告诉浏览器访问那些资源会携带这个cookie

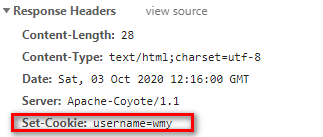
默认情况下，访问当前项目的所有资源都会携带cookie

1. 修改cookie

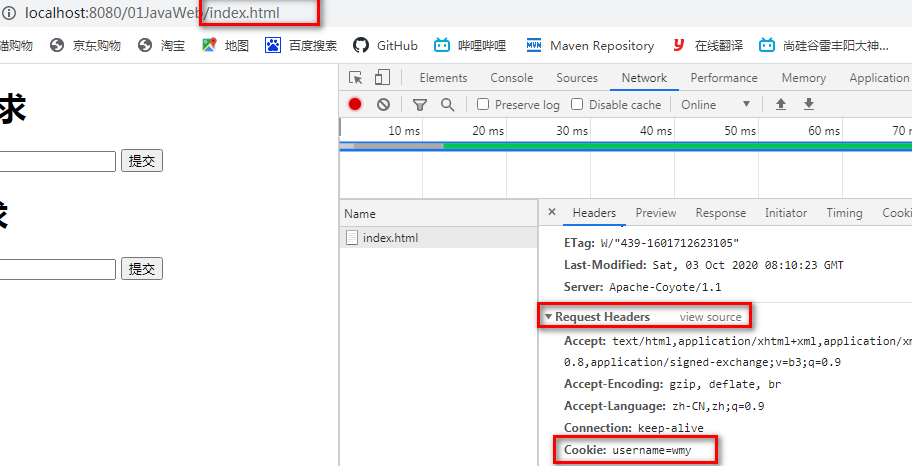
浏览器根据cookie的name 识别唯一的cookie；

因此，可以根据同名的cookie对cookie进行修改，删除等操作

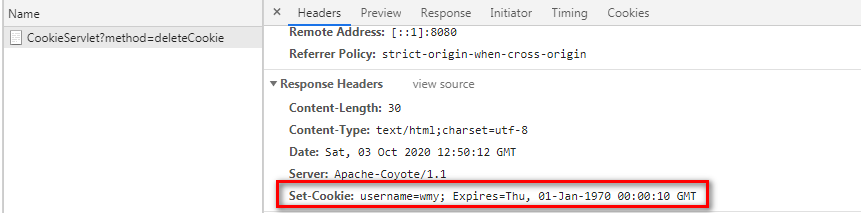
创建cookie







删除cookie

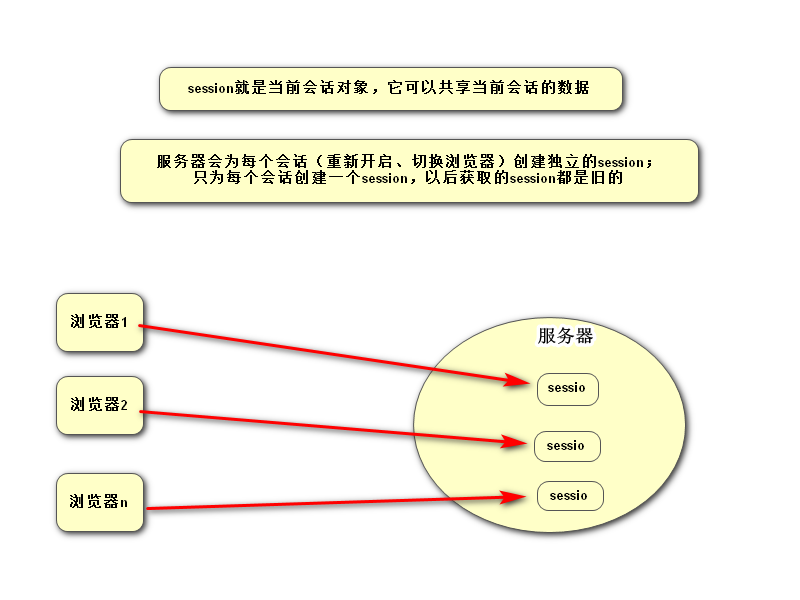


|  |
| --- |
| **public** **class** CookieServlet **extends** BaseServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;  //创建cookie  **protected** **void** createCookie(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  Cookie cookie = **new** Cookie("username", "wmy");  response.addCookie(cookie);  response.getWriter().write("cookie 给你了，，，，");  }  //获取Cookie  **protected** **void** getCookie(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  Cookie[] cookies = request.getCookies();  **for**(Cookie cookie:cookies){  String name = cookie.getName();  String value = cookie.getValue();  System.***out***.println(name+":"+value);  }  }  //删除Cookie  **protected** **void** deleteCookie(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  Cookie[] cookies = request.getCookies();  Cookie c=**null**;  **for**(Cookie cookie:cookies){  **if**("username".equals(cookie.getName())){  c=cookie;  }  }  //负数 ：不保存cookie，即使发给浏览器也不保存（这个功能大部分浏览器设置后不会有效果，还是保存）  //0：表示删除cookie  //正数：表示cookie的最大生存时间，单位秒  c.setMaxAge(0);  response.addCookie(c);  response.getWriter().write("username 的cookie已经删除");  }  //给Cookie设置路径，使得访问指定路径下的资源才会携带cookie  //默认情况下，访问当前项目的所有资源都会携带cookie  **protected** **void** setPath(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  Cookie cookie = **new** Cookie("mycookie", "path\_cookie");    //表示访问hello下的资源，cookie才会被带上；此处的"/"代表服务器的根目录  cookie.setPath("/hello");  response.addCookie(cookie);  response.getWriter().write("cookie的路径修改了");  }    //修改cookie  **protected** **void** updateCookie(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  Cookie cookie = **new** Cookie("username", "wmy01");  response.addCookie(cookie);  response.getWriter().write("cookie 修改了，，，，");  }  } |

## Session

服务器端保存数据的技术；域对象（在整个会话期间保存的数据在任意资源都可以取到）

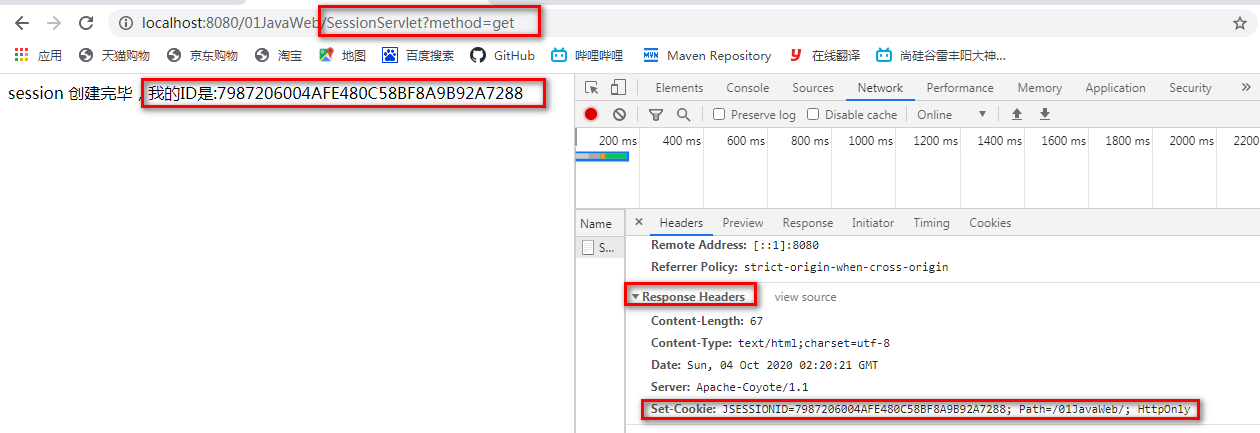
### Session保存数据共享数据



第一次请求服务器创建session时，服务器的响应头会让浏览器保存cookie（JSESSIONID）；

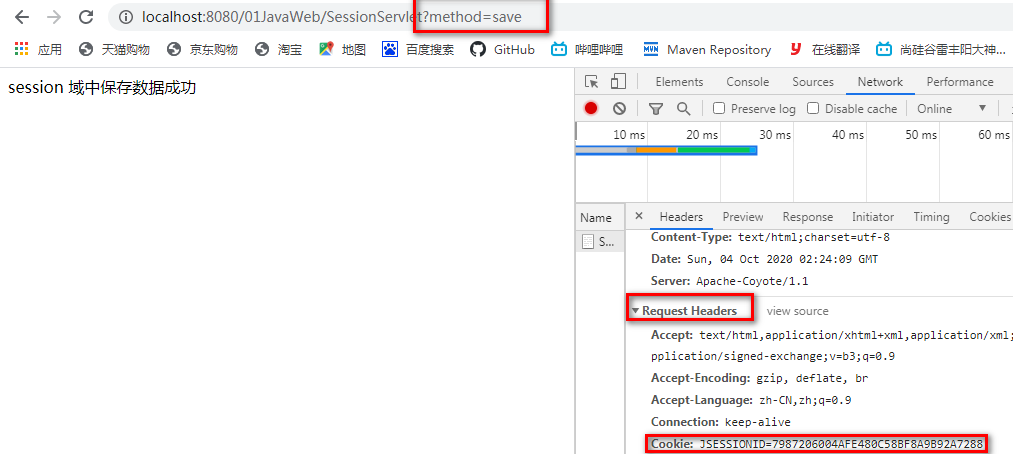
服务器以后会根据该cookie判断，浏览器的请求属于那个session

|  |
| --- |
| //创建session  **protected** **void** get(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {  //获取session对象  HttpSession session = request.getSession();  //判断session是否是新建的  **boolean** new1 = session.isNew();  //表示创建的session的唯一标识  String id = session.getId();  response.getWriter().write("session 创建完毕，我的ID是"+":"+id);  } |



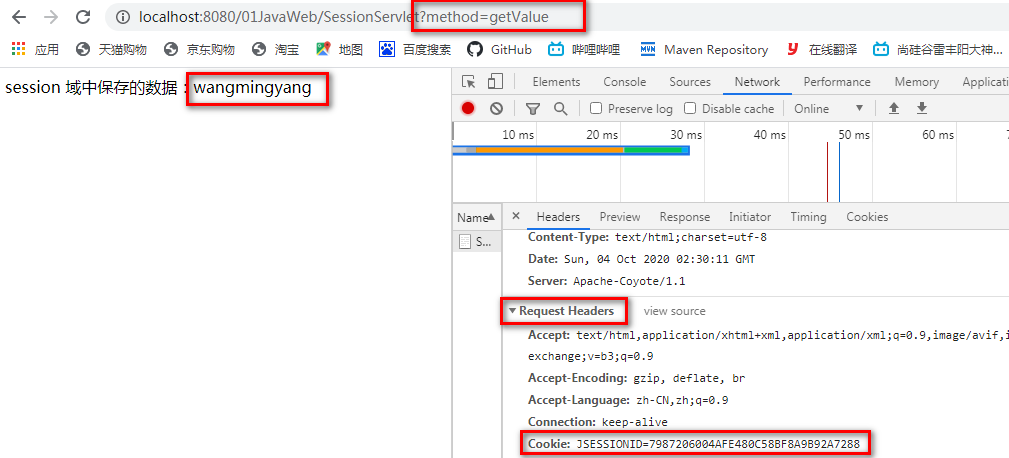
第二次请求时，浏览器的响应头中会带上cookie

|  |
| --- |
| //给session中保存数据  **protected** **void** save(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {    HttpSession session = request.getSession();  session.setAttribute("username", "wangmingyang");  response.getWriter().write("session 域中保存数据成功");  } |



客户端请求服务器时，会带上cookie，服务器依据该cookie，就知道要到那个session中获得保存的数据

|  |
| --- |
| //获取session中保存的数据  **protected** **void** getValue(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {    HttpSession session = request.getSession();  String attribute = (String) session.getAttribute("username");  response.getWriter().write("session 域中保存的数据："+attribute);  } |



### Session的生命周期

创建：第一次用session会创建一个新的session

对于html(使用servletrequest时)

对于jsp(已经内置session获取好了)

销毁：

* 默认30分钟后session销毁
* 收到设置session多少秒后过期session.setMaxInactiveInterval(3);
* 使session立即失效session.invalidate();

总结：新的会话获取的session是新的，和cookie的生命周期有关系。浏览器关闭打开开启新会话并不会把之前的session销毁，只是找不到之前的session了；session在服务器中有很多的垃圾。

|  |
| --- |
| //获取session的最大存活时间 单位秒  **protected** **void** timeToLive(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {    HttpSession session = request.getSession();  /\*\*  \* 获取session的最大存活时间 单位秒  \* session 默认存活时间1800（30分钟）  \* 为什么开启新的会话后会返回新的session？  \* 因为获取session是根据cookie带来的JSESSIONID来获取，  \* cookie默认关闭浏览器就没了；  \* 再来获取session时，由于没有cookie，所以服务器会返回新的session，  \* 但是旧的session还在，只是找不到了。  \*/  **int** maxInactiveInterval = session.getMaxInactiveInterval();  response.getWriter().write("session 最大存活时间："+maxInactiveInterval);  }  //修改session的失效时间  **protected** **void** update(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {    HttpSession session = request.getSession();  //传入负数：代表永不过期；传入正数：代表多少秒后过期（距离最后一次使用session的时间）  session.setMaxInactiveInterval(3);  response.getWriter().write("session 将于3秒后过期");  } |

### Cookie被禁用导致session失效

解决方法：使用url重写

当cookie被禁用，可以使用url重写（把要访问的地址动态的加上jsessionid）

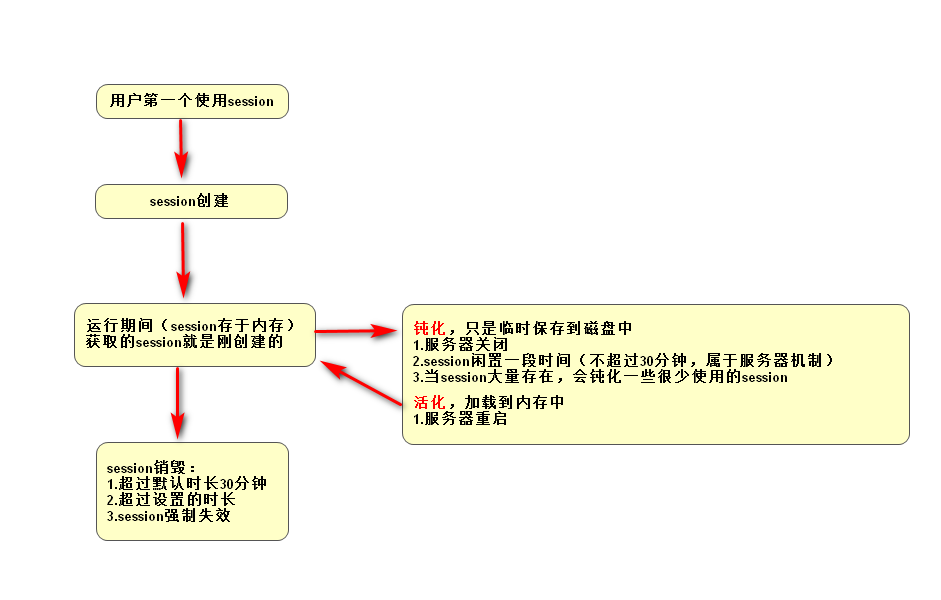
### Session 的活化和钝化

现象：服务器关闭重新启动，只要浏览器没关，还是能获取到session里面的内容

钝化：服务器关闭以后，会将session（序列化）保存在硬盘中；可以在work/当前目录下观察到保存的session.ser

活化：服务器再启动时，会把之前序列化好的文件（session.ser）加载进来；session.ser中包含了session域中的所有内容。

注意：在session域中存放对象时，如果该对象的类没有实现序列化接口，则该对象不会保存到session.ser文件；



Session的生命周期图

### Session在关闭浏览器后，继续在浏览器端保持

|  |
| --- |
| //关闭浏览器后，保持session  //思路：将jsessionid这个cookie进行持久化  **protected** **void** presist(HttpServletRequest request, HttpServletResponse response) **throws** ServletException, IOException {    HttpSession session = request.getSession();  String id = session.getId();  Cookie cookie = **new** Cookie("JSESSIONID", id);  //设置cookie的持久化时间  cookie.setMaxAge(60\*60);  //如果浏览器中存在同名的cookie，只会更新  response.addCookie(cookie);  response.getWriter().write("session 已经保持住了");  } |

## Filter



Filter拦截JSP页面

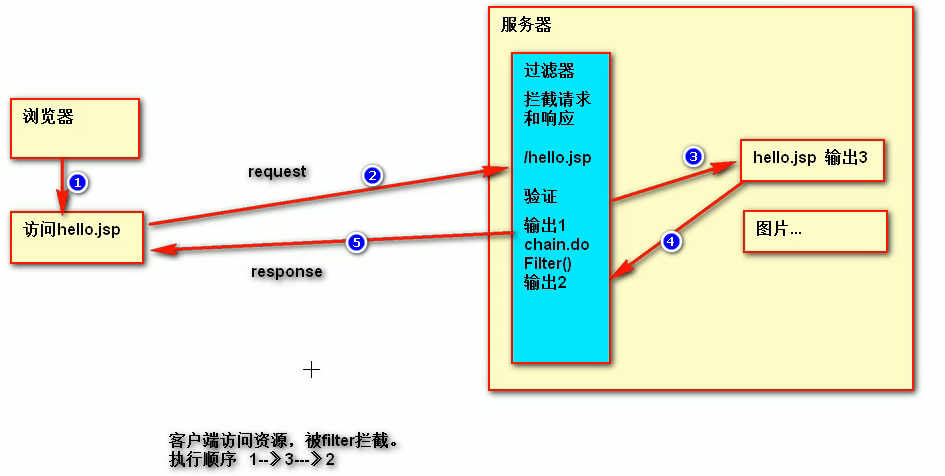
|  |
| --- |
| <!-- 配置filter的信息 -->  <filter>  <!-- 给filter起别名 -->  <filter-name>HelloFilter</filter-name>  <!-- filter 的全类名 -->  <filter-class>org.wmy.filter.HelloFilter</filter-class>  </filter>    <!-- 配置filter 的映射信息 -->  <filter-mapping>  <!-- 指定filter -->  <filter-name>HelloFilter</filter-name>  <!-- 过滤哪些请求 ，表示访问hello.jsp的时候会被过滤器拦截-->  <url-pattern>/hello.jsp</url-pattern>  </filter-mapping> |

|  |
| --- |
| **package** org.wmy.filter;  **import** java.io.IOException;  **import** javax.servlet.Filter;  **import** javax.servlet.FilterChain;  **import** javax.servlet.FilterConfig;  **import** javax.servlet.ServletException;  **import** javax.servlet.ServletRequest;  **import** javax.servlet.ServletResponse;  /\*\*  \* 创建的该过滤器，是用来过滤所有请求hello.jsp页面的  \*  \* 服务端的3大组件，是运行在服务器上的，服务器调用，因此需要配置  \* 1.写一个实现类  \* 2.在服务器配置filter要过滤哪些请求(在web.xml中)  \*  \*/  **public** **class** HelloFilter **implements** Filter{  @Override  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  // **TODO** Auto-generated method stub    }  //执行过滤方法  @Override  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {    System.***out***.println("我是hello.jsp的过滤器");  //通过请求参数判断是否放行  String money = request.getParameter("money");  **if**(money!=**null**){  //带钱了，将请求放行  System.***out***.println("我是请求放行之前1...");  //chain就是用来放行请求，只要不显示的调用放行方法，请求就不会被放行  chain.doFilter(request, response);  System.***out***.println("我是请求放行之后2...");  }**else**{  //没带钱直接返回  response.getWriter().write("no money,no mm");  }  }  @Override  **public** **void** destroy() {  // **TODO** Auto-generated method stub  }  } |

Filter 拦截Servlet

|  |
| --- |
| <servlet>  <servlet-name>UserServlet</servlet-name>  <servlet-class>org.wmy.controller.UserServlet</servlet-class>  </servlet>  <servlet-mapping>  <servlet-name>UserServlet</servlet-name>  <url-pattern>/UserServlet</url-pattern>  </servlet-mapping> |
| **public** **class** UserServlet **extends** BaseServlet {  **private** **static** **final** **long** ***serialVersionUID*** = 1L;    //处理用户注册的  **protected** **void** regist(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  //业务逻辑......  }  //处理用户登录的  **protected** **void** login(HttpServletRequest request, HttpServletResponse response)  **throws** ServletException, IOException {  //业务逻辑......  }  } |
| <filter>  <filter-name>UserFilter</filter-name>  <filter-class>org.wmy.controller.UserFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>UserFilter</filter-name>  <url-pattern>/UserServlet</url-pattern>  </filter-mapping>  这样就可以拦截访问UserServlet（所有方法）的所有请求 |

### Filter的执行流程



执行打印结果



### Filter 的生命周期

从创建到销毁的过程

1. 创建-初始化

服务器启动，并把项目加载进服务器，创建filter对象，并执行初始化；单例多线 程。

1. 每次拦截都执行

doFilter方法。。。。

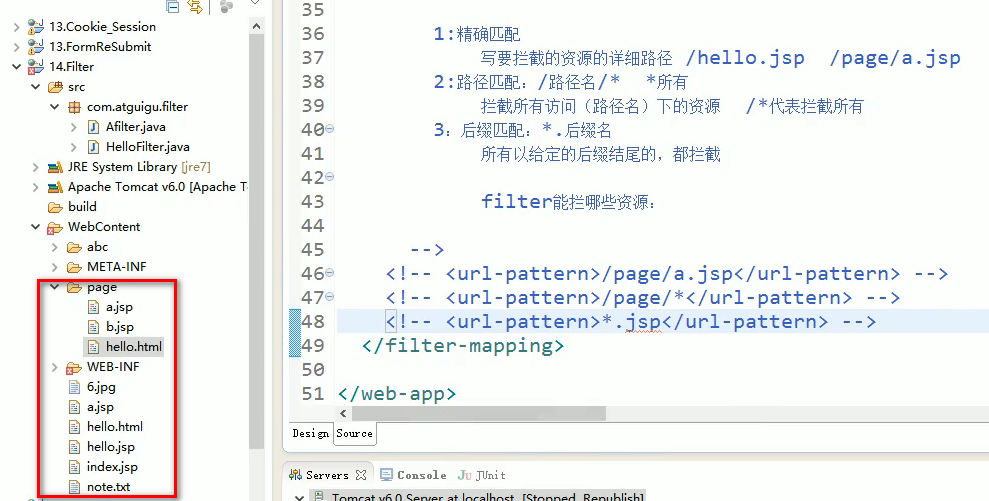
1. 销毁

项目从服务器中卸载执行destroy()方法

### Filter 通过设置url进行匹配拦截

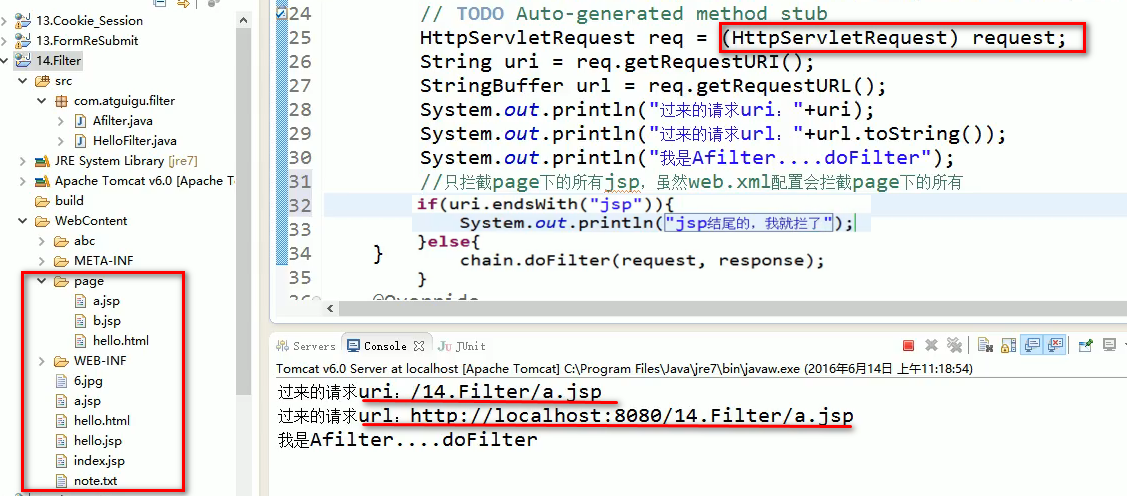
只有如下三种情况

这样写是错误的：<url-pattern>/page/\*.jsp</url-pattern>



### 可以通过request在filter中拦截部分请求

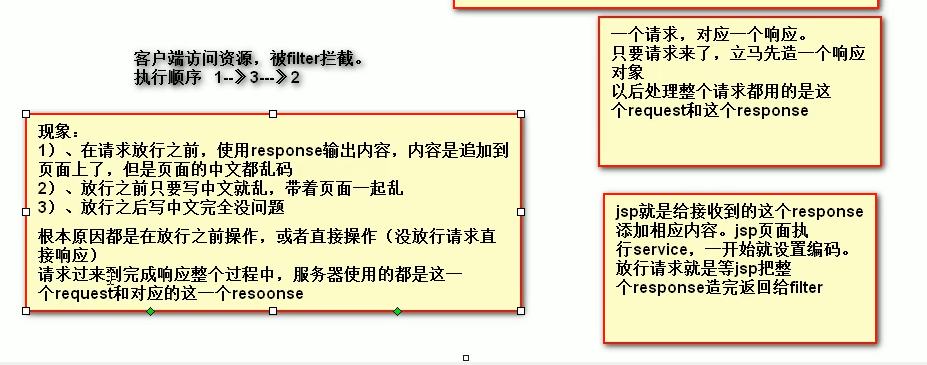




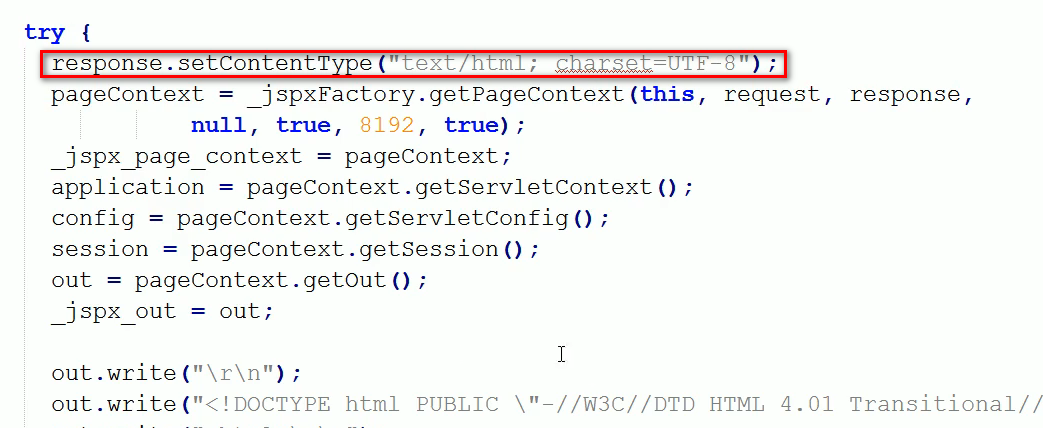
### Filter放行之前写中文乱码

* 解决乱码问题，一定要在写入内容之前进行设置；否则就算再设置也不会起效果（比如，先在doFilter方法中调用response写内容，再放行，尽管JSP中又设置了，但是不会起作用了）；
* 如下是可以进行的，并不会因为先执行response.getWriter().write("你好")，后面的内容就不执行，因为是在同一个response中添加内容。

|  |
| --- |
| @Override  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  //解决乱码，一定要在写入内容之前进行设置  //response.setContentType("text/html;charset=utf-8");    //response.getWriter().write("你好");//坏的  chain.doFilter(request, response);  response.getWriter().write("你好");//好的  } |



Jsp运行时的service代码

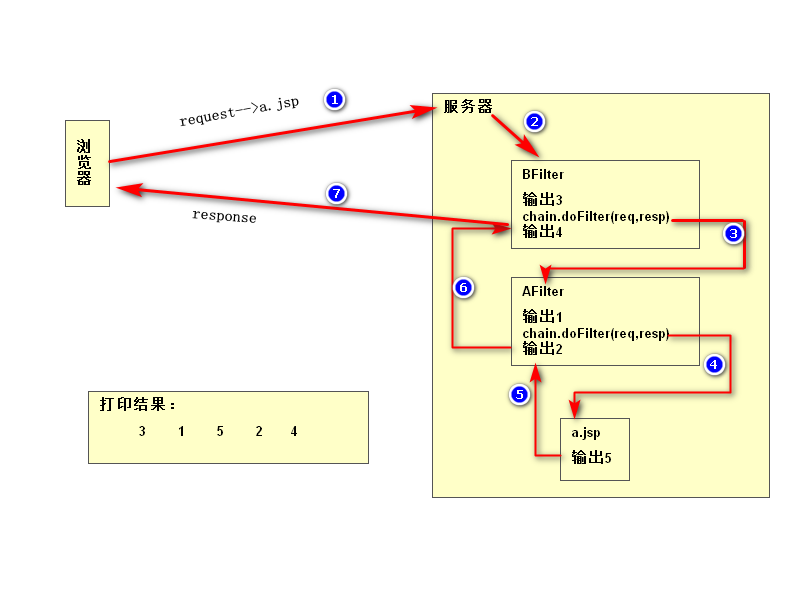


### FilterConfig

|  |
| --- |
| @Override  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  //获得filter别名  String filterName = filterConfig.getFilterName();  //获得filter中的配置参数  String initParameter = filterConfig.getInitParameter("user");  //获得项目的上下文  ServletContext servletContext = filterConfig.getServletContext();  //通过上下文对象获得web.xml中的初始化参数  String initParameter2 = servletContext.getInitParameter("username");  } |

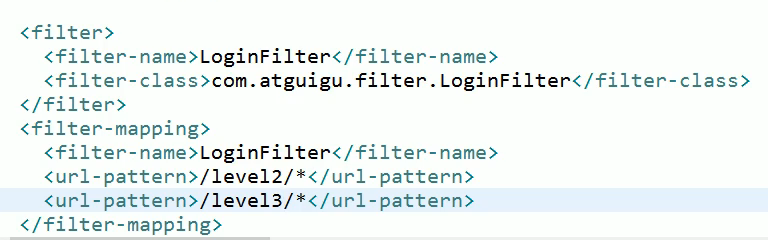
### 多级filter

Filter的执行顺序跟在web.xml中的配置顺序有关，先配置谁，谁先进行拦截。



Filter的登录拦截应用

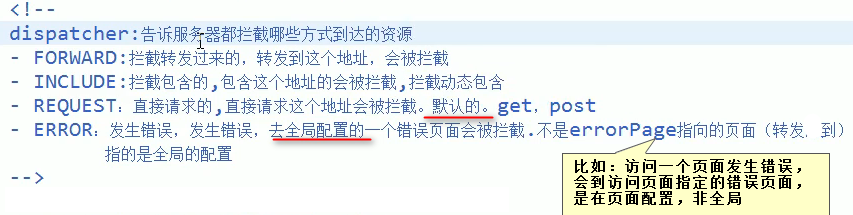
拦截Level2、level3中的所有资源，没有登录，就返回到登录页面。



|  |
| --- |
| @Override  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  //在该方法中要想获得session，必须先强转request  HttpServletRequest req = (HttpServletRequest) request;  HttpSession session = req.getSession();  String attribute = (String) session.getAttribute("user");  **if**(attribute==**null**){  //用户未登录，返回登录页面（index.jsp）  HttpServletResponse resp = (HttpServletResponse) response;  //要想使用sendRedirect，必须先进行强转  resp.sendRedirect(req.getContextPath()+"/index.jsp");  }**else**{  //用户已登录，放行请求  chain.doFilter(request, response);  }  } |

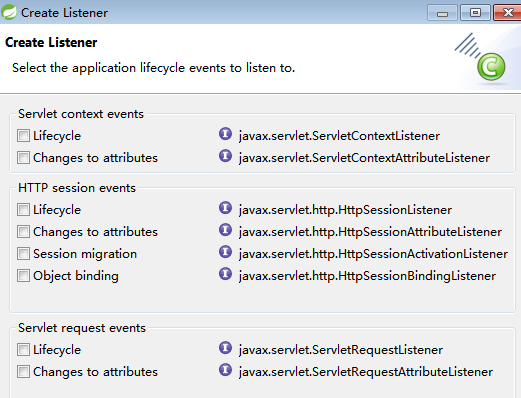
### Filter中配置dispatcher

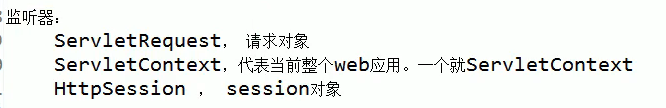
|  |
| --- |
| <error-page>  <error-code>404</error-code>  <location>/a.jsp</location>  </error-page>    <filter>  <filter-name>dispatcherFilter</filter-name>  <filter-class>org.wmy.controller.dispatcherFilter</filter-class>  </filter>  <filter-mapping>  <filter-name>dispatcherFilter</filter-name>  <url-pattern>/a.jsp</url-pattern>  <dispatcher>ERROR</dispatcher>  </filter-mapping> |



## Listenter

### 分类简介





Listener 三大类：

1. 生命周期监听器。监听三个对象的生命周期（创建到销毁）

* ServletRequestListener
  + 发送请求时初始化；请求结束后销毁
* ServletContextListener
  + 项目加载进服务器创建对象，项目卸载销毁
* HttpSessionListener
  + 新会话进来的时候创建对象（第一次使用，对于jsp初始化的时候获取过，html没有），session失效就销毁（强制失效、默认30分钟到时，自定义失效时间到）

1. 属性监听器。监听三个对象。监听域对象中属性的增(setAttribute())、删(removeAttribute())、改(setAttribute())。

* ServletRequestAttributeListener
* ServletContextAttributeListener
* HttpSessionAttributeListener

1. Session固有的监听器

* HttpSessionActivationListener 监听session的活化钝化
* HttpSessionBindingListener 监听一个对象是否绑定到session中（保存到session中）

### 使用

|  |
| --- |
| <listener>  <listener-class>org.wmy.listener.life.RequestLifeListener</listener-class>  </listener> |
| **public** **class** RequestLifeListener **implements** ServletRequestListener{  @Override  **public** **void** requestDestroyed(ServletRequestEvent sre) {  // **TODO** Auto-generated method stub  }  @Override  **public** **void** requestInitialized(ServletRequestEvent sre) {  // **TODO** Auto-generated method stub  }  } |

