**JavaWeb程序运行流程：**

用户发出请求后，会被8080端口的tomcat解析，tomcat会创建Request和Response对象(此时是HttpServletRequest/HttpServletResponse)，然后会去解析web.xml(在web容器中找到对应的Servlet)，找到请求的Servlet，经过listener，filter后到达Servlet，然后将处理的结果返回。

启动web项目后，web容器回去加载web.xml文件，然后创建一个ServletContext，加载顺序

ServletContext中封装了web.xml中的信息

ServletContext--->context-param--->listener--->filter--->servlet

启动web项目就加载Servlet还是第一个访问是加载：Laod-On-start

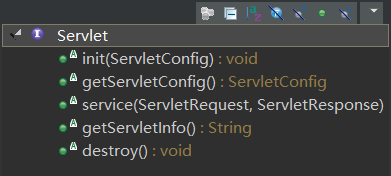
HttpServletRequest/HttpServletResponse在访问时创建，然后传递给Servlet的service方法

当有新请求时，先创建一个HttpServletRequest，然后执行listener，filter，Servlet

**Servlet的继承关系：**

HttpServlet----->GenericServlet----->Servlet

**Servlet**



要想创建servlet就得继承Servlet接口，这样才会被web容器管理

**创建Servlet流程：**

第一次访问此Servlet时，先调用Servlet的构造方法创建一个Servlet的实例，然后调用init()方法，对Servlet进初始化操作，最后调用service方法处理用户的请求，并返回结果，然后Servlet就会被web容器管理，当下次有新的请求访问此Servlet时，就会直接调用service方法，当服务器关闭时，会调用destroy方法。

**ServletConfig：**

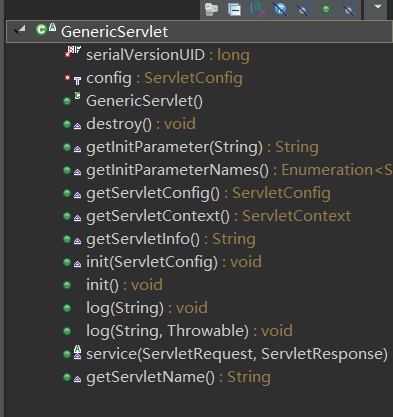
获取用户当前Servlet的参数，每个Servlet都对应着一个ServletConfig

**ServletRequest/ServletResponse：**

当web服务器接收到用户的请求时，会创建一个HttpServletRequest/HttpServletResponse

而HttpServletRequest/HttpServletResponse又继承至ServletRequest/ServletResponse

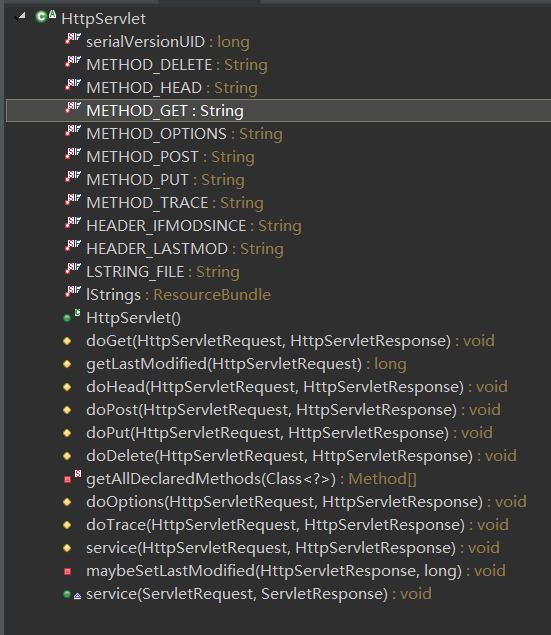
**GenericServlet**



**有个Servlet为什么还要GenericServlet?**

如果继承Servlet话，就需要实现Servlet的所有方法，但是有些方法又用不到，所以创建一个GenericServlet抽象类，这样我们就只要实现想要的方法，而且GenericServlet添加了很多新方法。

**HttpServlet**



HttpServlet接口继承GenericServlet，又添加了很多其他的方法，在

**service**(ServletRequest req, ServletResponse res)中，将ServletRequest/ServletResponse转化为

HttpServletRequest/HttpServletResponse，然后调用**service**(HttpServletRequest req, HttpServletResponse resp)

对用户的请求进行处理，判断用户的请求类型(post/get)，然后调用相应的方法进行处理。

Servlet不是线程安全的，在web容器中只有一个Servlet的实例，每次请求都会分配一个新的线程去处理用户的请求，不会等到当前用户请求完后再然其他用户请求，这也是不符合实际的。

**JSP注释**：

<%-- --%> :不可见，这是JSP的注释，JSP的注释不会发送到客户端

<!-- --> :可见，这是HTML的注释，在jsp中，最好使用<%-- --%>，因为<!-- -->对jsp的标签有一定的响应

**JSP脚本：**

<%= %>：脚本中的所有声明在转化为Servlet时，都会转化到\_jspService()方法中，属于局部变量(方法内)

**JSP表达式**：

<%= %>：用于输出变量

**JSP声明：**

<%! %>：用于声明变量和方法，声明中的方法和变量在转化为Servlet后，是成员变量(方法外，属于类)和成员方法

**JSP的编译指令：**

Page指令：

一个JSP页面中可以有个page指令

Include指令：

将外部jsp页面导入当前页面(静态)

Taglib指令：

用于访问标签



**JSP7个动作指令：**

Jsp:forward

<jsp:forward page=”new.jsp”>

转发到新的页面(内部转发)

Jsp:param

和include，forward，plugin指令使用

Jsp:include

动态导入

Jsp:plugin

用的少

Jsp:useBean

创建bean

<jsp:useBean id=”stu” class=”com.student” scope=”session”>

Jsp:setProperty

向bean中添加属性

<jsp:setProperty name=”stu” property=”” value=””>

Jsp:getProperty

获取Bean中的属性

编译指令时通知Servlet引擎处理消息，编译时

动作指令时运行是的动作，运行时

**Servlet的4个域对象：Map结构**

PageContext：

类型：PageContext

但是PageContext只能在JSP页面的中使用，不能在Servlet中使用，在JSP中使用PageContext可以向指定的域对象插入值，并且可以获取其他8个隐含对象，还可以通过findAttribute(String name)全域(page，Request，session，application)查找属性

Request：

类型：HttpServletRequest

Session：

类型：HttpSession

Application

ServletContext的实例，一个web应用存在一个唯一的ServletContext(管理所有Servlet)

作用：

获取整个web应用的初始化参数

获取资源的路径

最大域对象

获取方式：

ServletConfig

HttpSession

this(Servlet)

request

**JSP的9个内置对象：**

在jsp页面可以直接使用的

Application

ServletContext

Config

ServletConfig

Exception

Throwable：错误页面的异常对象

Out

JSPWriter，Servlet中的Response.getWriter

Page

当前页面，就是this

PageContext

当前页的上下文，是一个域对象，可以访问page，Request，session，application中的变量

Request

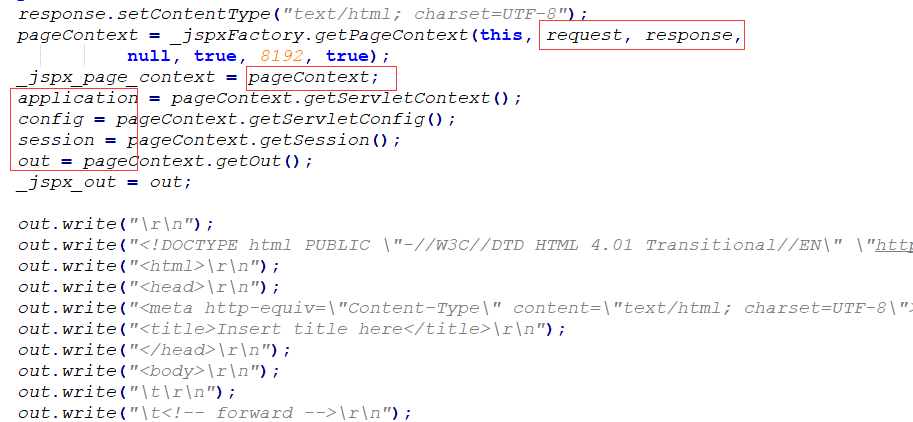
HttpServletRequest

Response

HttpServletResponse

Session

HttpSession



Response，Request是\_jspService(),page,pageCOntext，application，Config，session，out都是局部变量

**Page和PageContext的区别**

Page就是当前页面转化后的Servlet类，只能访问当前Servlet的属性，用的很少，而PageCOntext可以获取页面的上下文相关的信息，获取ServletContext，ServletConfig等其他隐含对象。

Page只能获取自己页面的数据，而PageContext还可以获取其他隐含对象，在JSP中还可以获取page，request，response，application中的属性

**内置对象和域对象的区别**

内置对象在JSP中使用，不需要创建就可以直接使用(JSP就是Servlet)，而域对象需要Servlet自己获取

内置对象的使用需要结合JSP脚本使用，所以用得很少

域对象是在Servlet中使用的(PageContext除外)，隐含对象可以直接在JSP的使用，无需创建，在Servlet中也可以使用隐含对象，但是要创建获取

**JSP/Servlet**

JSP会web容器转化为Servlet，在JSP被转化为Servlet的.java的源文件中可以看到，向客户端输出页面的内容的都是在\_jspService(HttpServletRequest request, HttpServletResponse response)方法中

**EL表达式**

EL是JSP内置的表达式语言，用以访问页面的上下文以及不同作用域中的对象 ，取得对象属性的值，或执行简单的运算或判断操作。EL在得到某个数据时，会自动进行数据类型的转换，格式：${expression} ${scope.name}，当EL表单式未找到属性时，不会像页面输出NULL，而JSP的表达式会输出NULL，不友好

EL在获取未指定域的属性时的加载顺序：pageScope、requestScope、sessionScope、applicationScope

EL存在11个隐含对象(Map类型)，可以直接在EL表达式获取

PageContext：和JSP的PageContext类似

请求域：主要获取域对象的属性

pageScope

requestScope

sessionScope

applicationScope

请求参数：

Param：获取请求参数

ParamValues：获取请求参数集

其他：

Header，headerValues，initParam，cookie

**JSTL：**

EL表达式只能做获取域中的数据和一些基本的运算，功能不完善，所以引入JSTL第三方标签，需要导包，JSTL可以遍历集合，添加属性的操作，JSTL还可以声明函数等功能更加丰富。

JSTL默认是从page，request，session，application这四个Scope逐次查找

request.getsession(true)和request.getsession(false)的区别在于：

request.getsession(true)判断有session就得到当前的session没有则创建一个新的，

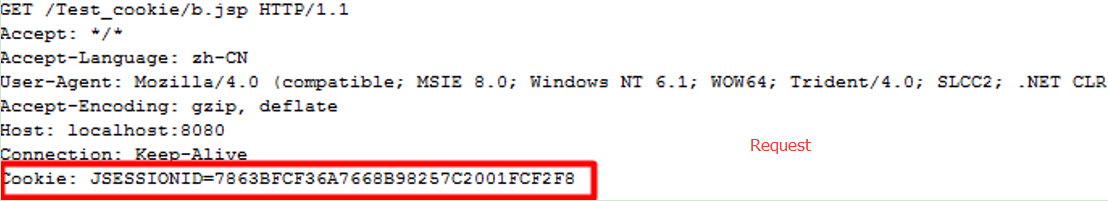
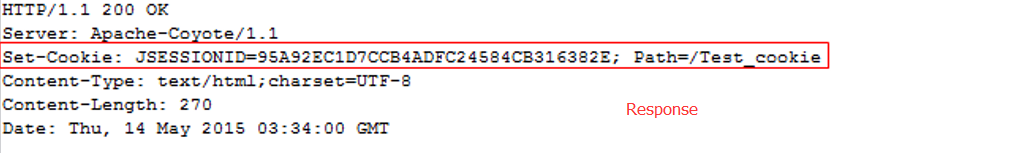
request.getsession(false)是有则得到这个session没有则返回null

request，session，application中保存数据的方式都是ConcurrentHashMap

**Cookie/Session：**

Http请求是无状态纯文本的，服务器不能简单的通过HTTP协议来区分多次请求是否发自同一个用户

当用户请求时，服务器会为当前用户创建一个Cookie，然后为Cookie赋值一个JSESSIONID(唯一的)，然后服务器通过Response将Cookie返回给浏览器，当浏览器再次请求时，Request会将JSESSIONID传递给服务器，这样服务器和浏览器就建立了一个长连接



用户也可以像Cookie中添加其他对象，服务器也会将这些对象返回给浏览器，但是如果过多的信息都存放在Cookie中，这样每次传输的数据就很多(因为每次服务器和浏览器的通信时都会将Cookie中的数据传输)，所以需要创建Session

浏览器是如何判断访问一个网址时，该携带哪个Cookie的，因为在本地中可能缓存很多的Cookie，在Response返回的Cookie中，JSESSIONID对应着一个Path，当请求指定Path时，将会携带对应的JSESSIONID给服务器

Session在request.getSession()的时候创建，session对应着JSESSIONID，所以每个用户在服务器存在唯一的session，session是存放在服务器的，用户可以将数据存放在session中，当浏览器与服务器断开后，session中的数据将丢失

**Session的钝化/活化：**

当服务器停止时，可以将Session序列化到硬盘上(对象必须实现java.io.Serializable接口)，当服务器启动时，会将钝化的session活化(反序列化)到内存中

当服务器中的Session的过多时，但是有些又未使用时，可以将Session钝化到硬盘中

**区别和联系：**

Session是基于Cookie的，Cookie是将信息保存到客户端的，可以设置过期时间，而Session是保存在浏览器中。

**URL的重写**

如果用户禁用Cookie，那么就不能获取JSESSIONID，也不能获取Session，可以将用户JSESSIONID放在地址后面，但是这种方法太不安全

为什么有些项目中会出现JSESSIONID，通过标签使用redirect(重定向)的时候，当检测不到cookie时，会自动生成sessionid，换用Response.sendRedirect的方式重定向后即可

**表单的重复提交：**

**情况：**

提交表单后，多次刷新页面

提交表单时，多次提交(网速差的情况下，不知道是否提交成功，多次点提交)

表单提交成功以后，直接点击浏览器上回退按钮，不刷新页面，然后点击提交按钮再次提交表单

**解决方式：**

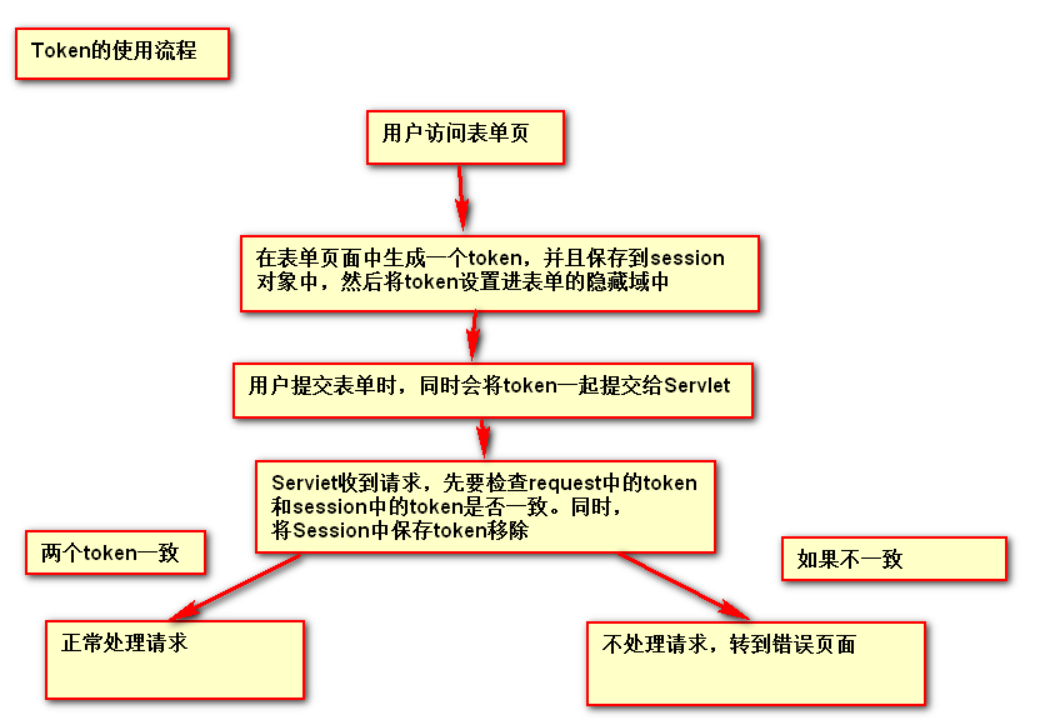
使用token

token由服务器来创建，并交给浏览器，浏览器在向服务器发送请求时需要带着这个token，服务器处理请求前检查token是否正确，如果正确，则正常处理，token一般使用一个唯一的标识

**危害：**

向数据库中插入大量的重复且没有意义的数据，占用服务器的资源。

处理请求服务器并没有检查请求是否为重复的请求，导致恶意的攻击。



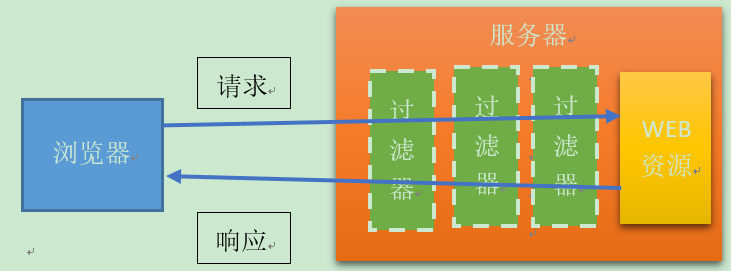
当第二次提交时，token还是原来的token，但是session中已经没有token了。

**Filter：**

Filter和Servlet的创建方式，声明周期，使用都差不多，只是Filter在Servlet执行之前执行，在Servlet前可以放多个Filter(执行顺序根据在web.xml的顺序)，Filter可以根据需求是执行下一个Filter/Servlet，还是直接返回响应(说明HttpServletRequest/HttpServletResponse是在此前就创建了)，返回响应时，任然会倒序执行Filter

不同于Servlet，Filter可以根据请求的类型(直接访问/转发访问/动态包含/异常处理)来判断是否执行Filter，默认是直接访问时就执行

<dispatcher>REQUEST/ FORWARD/ INCLUDE / ERROR </dispatcher>



**Listener：**

用于监听web程序中的事件，例如ServletContext，HttpSession，HttpServletRequest的创建，修改，删除。

**ServletContextListener(接口)**

用于监听ServletContext对象的创建，删除

public void contextInitialized ( ServletContextEvent sce ); ServletContext创建时调用

public void contextDestroyed ( ServletContextEvent sce ); ServletContext销毁时调用

ServletContextEvent

public ServletContext getServletContext ()

**ServletRequestListener(接口)**

监听ServletRequest对象的创建与销毁

public void requestInitialized ( ServletRequestEvent sre ); ServletRequest创建时调用

public void requestDestroyed ( ServletRequestEvent sre ); ServletRequest销毁时调用

ServletRequestEvent对象

public ServletRequest getServletRequest ();

public ServletContext getServletContext ();

每种监听器对应一种事件，比如这里的ServletRequestEvent就是一个事件，而事件被对应的监听器监听，当事件改变时，触发对应的监听器 事件对应监听器

**HttpSessionListener(接口)**

监听HttpSession对象的创建与销毁

public void sessionCreated ( HttpSessionEvent se ); HttpSession对象创建时调用

public void sessionDestroyed ( HttpSessionEvent se ); HttpSession对象销毁时调用

HttpSessionEvent对象

public HttpSession getSession ()



服务器启动时调用contextInitialized()方法，服务器停止时调用contextDestroyed()方法

**监听属性变化的监听器：**

**ServletContextAttributeListener：**

监听ServletContext中属性的创建、修改和销毁

public void attributeAdded(ServletContextAttributeEvent scab);

public void attributeRemoved(ServletContextAttributeEvent scab);

public void attributeReplaced(ServletContextAttributeEvent scab);

ServletContextAttributeEvent对象

public String getName() 获取修改或添加的属性名

public Object getValue() 获取被修改或添加的属性值

public ServletContext getServletContext ()

获取当前WEB应用的ServletContext对象



**HttpSessionAttributeListener**

**ServletRequestAttributeListener：使用方法类似**

**HttpSessionBindingListener**

监听某个对象在session域中的创建与移除。

public void valueBound(HttpSessionBindingEvent event)

该类的实例被放到Session域中时调用，当该对象放入session中时调用

public void valueUnbound(HttpSessionBindingEvent event)

该类的实例从Session中移除时调用

HttpSessionBindingEvent对象

public HttpSession getSession ()

public String getName() 获取操作的属性名

public Object getValue() 获取操作的属性值

**HttpSessionActivationListener**

void sessionDidActivate(HttpSessionEvent se)

当前对象和session一起被活化到内存时调用

void sessionWillPassivate(HttpSessionEvent se)

当前对象和session一起钝化到硬盘时调用

监听器监听的事件在监听器创建之前已存在

**Ajax**

异步向服务器发送Web请求

XMLHttpRequest：用于封装请求报文

创建方式

var xhr = new XMLHttpRequest() 目前主流浏览器都支持

var xhr = new ActiveXObject("Msxml2.XMLHTTP") IE6支持的方式

var xhr = new ActiveXObject("Microsoft.XMLHTTP") IE5.5一下支持的方式

使用：

var xhr = new XMLHttpRequest()；

xhr. open(method,url,async); 建立连接

method：请求方式(post/get)

url：清单地址

async：是否是异步请求

xhr. send(string); 发送请求

String：请求参数

Xhr. setRequestHeader(header,value)：设置请求头参数

Xhr.onreadystatechange =function(){

当readyState属性变化时会调用此函数(对服务器返回的结果进行响应)

Xhr.readyState;

描述XMLHttpRequest的状态

0 ：请求尚未初始化，open()尚未被调用

1 ：服务器连接已建立，send()尚未被调用

2 ：请求已接收，服务器尚未响应

3 ：请求已处理，正在接收服务器发送的响应

4 ：请求已处理完毕，且响应已就绪。

Xhr.status;

请求的响应码

200响应成功

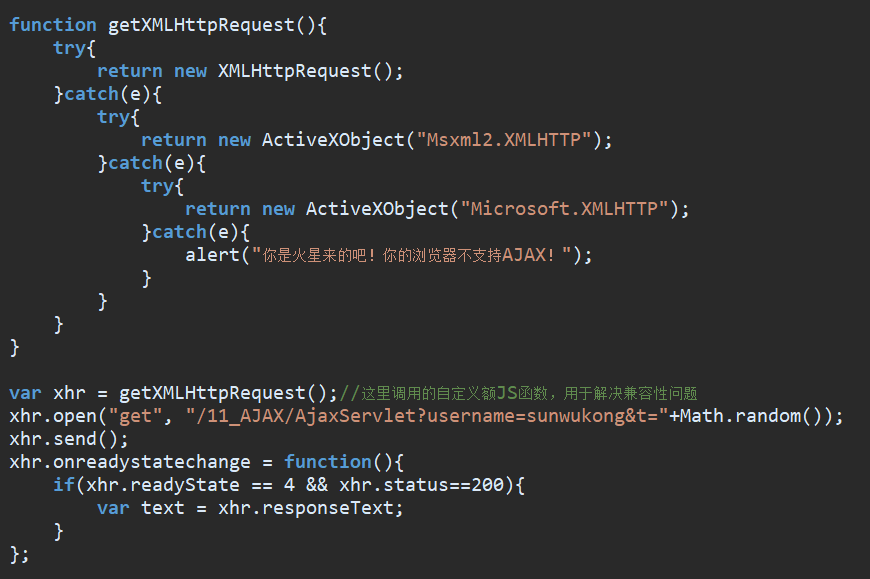
404 页面未找到

500 服务器内部错误

Xhr.responseTest;

获得字符串形式的响应数据

}



在Tomcat的lib目录下的存在Servlet相关的API

get方式只能支持ASCII字符，向服务器传的中文字符可能会乱码。

post支持标准字符集，可以正确传递中文字符。

get请求可以被缓存，可以被收藏为书签，但 post 不行。

get请求会保留在浏览器的历史记录中，post 不会。

