# JavaWeb

## http协议

协议指的是双方在交互、通讯的时候， 遵守的一种规范、规则。

http协议是针对网络上的客户端与服务端在执行http请求时遵守的一种规范。即规定客户端在访问服务端时，要带上哪些东西；服务端返回数据时，也要带上什么东西。

HTTP是一个属于应用层的面向对象的协议，由于其简捷、快速的方式，适用于分布式超媒体信息系统。

**HTTP协议的主要特点可概括如下：**

1.支持客户/服务器模式。

2.简单快速：客户向服务器请求服务时，只需传送请求方法和路径。请求方法常用的有GET、HEAD、POST等。每种方法规定了客户与服务器联系的类型不同。由于HTTP协议简单，使得HTTP服务器的程序规模小，因而通信速度很快。

3.灵活：HTTP允许传输任意类型的数据对象。正在传输的类型由Content-Type加以标记。

4.无连接：无连接的含义是限制每次连接只处理一个请求。服务器处理完客户的请求，并收到客户的应答后，即断开连接。采用这种方式可以节省传输时间。

5.无状态：HTTP协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。另一方面，在服务器不需要先前信息时它的应答就较快。

### http的1.0版本

请求数据，服务器返回后，将会断开连接。

### http的1.1版本

请求数据，服务器返回后，连接还会保持着。除非服务器或客户端关掉。有一定的时间限制，如果都空着这个连接，那么后面会自己断掉。

http（超文本传输协议）是一个基于请求与响应模式的、无状态的、应用层的协议，常基于TCP的连接方式，HTTP1.1版本中给出一种持续连接的机制，绝大多数的Web开发，都是构建在HTTP协议之上的Web应用。

### URL

HTTP URL（**Uniform Resource Locator**,统一资源定位符）(URL是一种特殊类型的URI，包含了用于查找某个资源的足够的信息)的格式如下：

http://host[":"port][abs\_path]

http表示要通过HTTP协议来定位网络资源；

host表示合法的Internet主机域名或者IP地址；

port指定一个端口号，为空则使用缺省端口80；

abs\_path指定请求资源的URI；

如果URL中没有给出abs\_path，那么当它作为请求URI时，必须以“/”的形式给出，通常这个工作浏览器自动帮我们完成。

在浏览器地址栏中输入：www.guet.edu.cn时，浏览器自动转换成：http://www.guet.edu.cn/。

## 请求

一个请求中包含请求行、请求头、请求体。

### 请求组成部分

#### 请求行

请求行以一个方法符号开头，以空格分开，后面跟着请求的URI和协议的版本，格式如下：

Method Request-URI HTTP-Version CRLF

Method表示请求方法；

Request-URI是一个统一资源标识符；

HTTP-Version表示请求的HTTP协议版本；

CRLF表示回车和换行（除了作为结尾的CRLF外，不允许出现单独的CR或LF字符）。

#### 请求头

Accept: 客户端向服务器端表示，我能支持什么类型的数据。

Referer ： 真正请求的地址路径，全路径

Accept-Language: 支持语言格式

User-Agent: 用户代理 向服务器表明，当前来访的客户端信息。

Content-Type： 提交的数据类型。经过urlencoding编码的form表单的数据

Accept-Encoding： gzip, deflate ： 压缩算法 。

Host ： 主机地址

Content-Length： 数据长度

Connection : Keep-Alive 保持连接

Cache-Control ： 对缓存的操作

#### 请求体

请求体中的内容是浏览器真正发送给服务器的数据，发送的数据呈现的是key=value ,如果存在多个数据，那么使用&firstname=zhang&lastname=sansan的方式进行连接。

### 请求方式

GET 请求获取Request-URI所标识的资源，会在地址栏后面拼接数据，所以有安全隐患。 一般从服务器获取数据，并且客户端也不用提交上面数据的时候，可以使用GET。能够带的数据有限， 1kb大小。

POST 在Request-URI所标识的资源后附加新的数据，数据是以流的方式写过去，不会在地址栏上面显示。 现在一般提交数据到服务器使用的都是POST。以流的方式写数据，所以数据没有大小限制。

HEAD 请求获取由Request-URI所标识的资源的响应消息报头

PUT 请求服务器存储一个资源，并用Request-URI作为其标识

DELETE 请求服务器删除Request-URI所标识的资源

TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断

CONNECT 保留将来使用

OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求

应用举例：

GET方法：在浏览器的地址栏中输入网址的方式访问网页时，浏览器采用GET方法向服务器获取资源，

eg:GET /form.html HTTP/1.1 (CRLF)

POST方法要求被请求服务器接受附在请求后面的数据，常用于提交表单。

eg：POST /reg.jsp HTTP/ (CRLF)

Accept:image/gif,image/x-xbit,... (CRLF)

...

HOST:www.guet.edu.cn (CRLF)

Content-Length:22 (CRLF)

Connection:Keep-Alive (CRLF)

Cache-Control:no-cache (CRLF)

(CRLF) //该CRLF表示消息报头已经结束，在此之前为消息报头

user=jeffrey&pwd=1234 //此行以下为提交的数据

HEAD方法与GET方法几乎是一样的，对于HEAD请求的回应部分来说，它的HTTP头部中包含的信息与通过GET请求所得到的信息是相同的。利用这个方法，不必传输整个资源内容，就可以得到Request-URI所标识的资源的信息。该方法常用于测试超链接的有效性，是否可以访问，以及最近是否更新。

## 响应

在接收和解释请求消息后，服务器返回一个HTTP响应消息，一个响应中包含响应行 、 响应头 、响应体。

### 响应行

HTTP/1.1 200 OK

Server: Apache-Coyote/1.1

Content-Type: text/html;charset=ISO-8859-1

Content-Length: 673

Date: Fri, 17 Feb 2017 02:53:02 GMT

HTTP-Version Status-Code Reason-Phrase CRLF

HTTP-Version表示服务器HTTP协议的版本；

Status-Code表示服务器发回的响应状态代码；

Reason-Phrase表示状态代码的文本描述。

#### 状态码

状态代码有三位数字组成，第一个数字定义了响应的类别，且有五种可能取值：

1xx：指示信息--表示请求已接收，继续处理

2xx：成功--表示请求已被成功接收、理解、接受

3xx：重定向--要完成请求必须进行更进一步的操作

4xx：客户端错误--请求有语法错误或请求无法实现

5xx：服务器端错误--服务器未能实现合法的请求

**常见状态代码、状态描述、说明：**

200 OK //客户端请求成功

400 Bad Request //客户端请求有语法错误，不能被服务器所理解

401 Unauthorized //请求未经授权，这个状态代码必须和WWW-Authenticate报头域一起使用

403 Forbidden //服务器收到请求，但是拒绝提供服务

404 Not Found //请求资源不存在，eg：输入了错误的URL

500 Internal Server Error //服务器发生不可预期的错误

503 Server Unavailable //服务器当前不能处理客户端的请求，一段时间后可能恢复正常

### 响应头

Server: 服务器是哪一种类型

Content-Type ： 服务器返回给客户端你的内容类型

Content-Length ： 返回的数据长度

Date ： 通讯的日期，响应的时间

### 响应体

响应体就是服务器返回的资源的内容。

## Web资源

在http协议当中，规定了请求和响应双方，客户端和服务器端与web相关的资源。

### 静态资源

html、js、css、jpg、gif等

### 动态资源

servlet、jsp等

## 消息报头

HTTP消息由客户端到服务器的请求和服务器到客户端的响应组成。请求消息和响应消息都是由开始行（对于请求消息，开始行就是请求行，对于响应消息，开始行就是状态行），消息报头（可选），空行（只有CRLF的行），消息正文（可选）组成。

HTTP消息报头包括普通报头、请求报头、响应报头、实体报头。

每一个报头域都是由名字+“：”+空格+值 组成，消息报头域的名字是大小写无关的。

### 普通报头

在普通报头中，有少数报头域用于所有的请求和响应消息，但并不用于被传输的实体，只用于传输的消息。

eg：

Cache-Control 用于指定缓存指令，缓存指令是单向的（响应中出现的缓存指令在请求中未必会出现），且是独立的（一个消息的缓存指令不会影响另一个消息处理的缓存机制），HTTP1.0使用的类似的报头域为Pragma。

请求时的缓存指令包括：no-cache（用于指示请求或响应消息不能缓存）、no-store、max-age、max-stale、min-fresh、only-if-cached;

响应时的缓存指令包括：public、private、no-cache、no-store、no-transform、must-revalidate、proxy-revalidate、max-age、s-maxage.

eg：为了指示IE浏览器（客户端）不要缓存页面，服务器端的JSP程序可以编写如下：response.sehHeader("Cache-Control","no-cache");

//response.setHeader("Pragma","no-cache");作用相当于上述代码，通常两者//合用

这句代码将在发送的响应消息中设置普通报头域：Cache-Control:no-cache

Date普通报头域表示消息产生的日期和时间

Connection普通报头域允许发送指定连接的选项。例如指定连接是连续，或者指定“close”选项，通知服务器，在响应完成后，关闭连接

### 请求报头

请求报头允许客户端向服务器端传递请求的附加信息以及客户端自身的信息。

常用的请求报头

Accept

Accept请求报头域用于指定客户端接受哪些类型的信息。eg：Accept：image/gif，表明客户端希望接受GIF图象格式的资源；Accept：text/html，表明客户端希望接受html文本。

Accept-Charset

Accept-Charset请求报头域用于指定客户端接受的字符集。eg：Accept-Charset:iso-8859-1,gb2312.如果在请求消息中没有设置这个域，缺省是任何字符集都可以接受。

Accept-Encoding

Accept-Encoding请求报头域类似于Accept，但是它是用于指定可接受的内容编码。eg：Accept-Encoding:gzip.deflate.如果请求消息中没有设置这个域服务器假定客户端对各种内容编码都可以接受。

Accept-Language

Accept-Language请求报头域类似于Accept，但是它是用于指定一种自然语言。eg：Accept-Language:zh-cn.如果请求消息中没有设置这个报头域，服务器假定客户端对各种语言都可以接受。

Authorization

Authorization请求报头域主要用于证明客户端有权查看某个资源。当浏览器访问一个页面时，如果收到服务器的响应代码为401（未授权），可以发送一个包含Authorization请求报头域的请求，要求服务器对其进行验证。

Host（发送请求时，该报头域是必需的）

Host请求报头域主要用于指定被请求资源的Internet主机和端口号，它通常从HTTP URL中提取出来的，eg：

我们在浏览器中输入：http://www.guet.edu.cn/index.html

浏览器发送的请求消息中，就会包含Host请求报头域，如下：

Host：www.guet.edu.cn

此处使用缺省端口号80，若指定了端口号，则变成：Host：www.guet.edu.cn:指定端口号

User-Agent

我们上网登陆论坛的时候，往往会看到一些欢迎信息，其中列出了你的操作系统的名称和版本，你所使用的浏览器的名称和版本，这往往让很多人感到很神奇，实际上，服务器应用程序就是从User-Agent这个请求报头域中获取到这些信息。User-Agent请求报头域允许客户端将它的操作系统、浏览器和其它属性告诉服务器。不过，这个报头域不是必需的，如果我们自己编写一个浏览器，不使用User-Agent请求报头域，那么服务器端就无法得知我们的信息了。

请求报头举例：

GET /form.html HTTP/1.1 (CRLF)

Accept:image/gif,image/x-xbitmap,image/jpeg,application/x-shockwave-flash,application/vnd.ms-excel,application/vnd.ms-powerpoint,application/msword,\*/\* (CRLF)

Accept-Language:zh-cn (CRLF)

Accept-Encoding:gzip,deflate (CRLF)

If-Modified-Since:Wed,05 Jan 2007 11:21:25 GMT (CRLF)

If-None-Match:W/"80b1a4c018f3c41:8317" (CRLF)

User-Agent:Mozilla/4.0(compatible;MSIE6.0;Windows NT 5.0) (CRLF)

Host:www.guet.edu.cn (CRLF)

Connection:Keep-Alive (CRLF)

(CRLF)

### 响应报头

响应报头允许服务器传递不能放在状态行中的附加响应信息，以及关于服务器的信息和对Request-URI所标识的资源进行下一步访问的信息。

常用的响应报头

Location

Location响应报头域用于重定向接受者到一个新的位置。Location响应报头域常用在更换域名的时候。

Server

Server响应报头域包含了服务器用来处理请求的软件信息。与User-Agent请求报头域是相对应的。下面是

Server响应报头域的一个例子：

Server：Apache-Coyote/1.1

WWW-Authenticate

WWW-Authenticate响应报头域必须被包含在401（未授权的）响应消息中，客户端收到401响应消息时候，并发送Authorization报头域请求服务器对其进行验证时，服务端响应报头就包含该报头域。

eg：WWW-Authenticate:Basic realm="Basic Auth Test!" //可以看出服务器对请求资源采用的是基本验证机制。

### 实体报头

请求和响应消息都可以传送一个实体。一个实体由实体报头域和实体正文组成，但并不是说实体报头域和实体正文要在一起发送，可以只发送实体报头域。实体报头定义了关于实体正文（eg：有无实体正文）和请求所标识的资源的元信息。

常用的实体报头

Content-Encoding

Content-Encoding实体报头域被用作媒体类型的修饰符，它的值指示了已经被应用到实体正文的附加内容的编码，因而要获得Content-Type报头域中所引用的媒体类型，必须采用相应的解码机制。Content-Encoding这样用于记录文档的压缩方法，eg：Content-Encoding：gzip

Content-Language

Content-Language实体报头域描述了资源所用的自然语言。没有设置该域则认为实体内容将提供给所有的语言阅读

者。eg：Content-Language:da

Content-Length

Content-Length实体报头域用于指明实体正文的长度，以字节方式存储的十进制数字来表示。

Content-Type

Content-Type实体报头域用语指明发送给接收者的实体正文的媒体类型。eg：

Content-Type:text/html;charset=ISO-8859-1

Content-Type:text/html;charset=GB2312

Last-Modified

Last-Modified实体报头域用于指示资源的最后修改日期和时间。

Expires

Expires实体报头域给出响应过期的日期和时间。为了让代理服务器或浏览器在一段时间以后更新缓存中(再次访问曾访问过的页面时，直接从缓存中加载，缩短响应时间和降低服务器负载)的页面，我们可以使用Expires实体报头域指定页面过期的时间。eg：Expires：Thu，15 Sep 2006 16:23:12 GMT

HTTP1.1的客户端和缓存必须将其他非法的日期格式（包括0）看作已经过期。eg：为了让浏览器不要缓存页面，我们也可以利用Expires实体报头域，设置为0，jsp中程序如下：response.setDateHeader("Expires","0");

# Servlet

## 概述

Servlet是JavaWeb三大组件之一。JavaWeb的三大组件分别是：Servlet程序、Filter过滤器、Listener监听器。Servlet是一个接口，是JavaEE规范之一。Servlet程序是运行在服务器的上的java程序，通过http协议接收处理客户端发送过来的请求，并回传数据。

自定义servlet更多的是配合动态资源运行，静态资源也需要使用到servlet，Tomcat里已定义一个DefaultServlet用来处理静态资源。

Servlet可以理解为一个对数据进行操作的中转站，页面和页面之间进行跳转时，可以通过Servlet获取数据并传递到新页面，返回原页面时也可以通过Servlet中转并携带新的数据到原页面。

## 配置Servlet

### 实现Servlet接口

|  |
| --- |
| public class ServletImpl implements Servlet {  public void init(ServletConfig servletConfig){ }  public ServletConfig getServletConfig() { return null; }  public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {}  public String getServletInfo() {return null;}  public void destroy() { } } |

### 配置XML文件

|  |
| --- |
| <!-- servlet标签配置servlet程序 --> <servlet>  <!-- servlet-name标签是给servlet程序配置一个别名 -->  <servlet-name>ServletImpl</servlet-name>  <!-- servlet-class标签配置全类名 -->  <servlet-class>com.web.servlet.ServletImpl</servlet-class> </servlet> <!-- servlet-mapping标签配置访问地址 --> <servlet-mapping>  <!-- servlet-name是告诉Tomcat服务器，我当前配置的地址给谁用 -->  <servlet-name>ServletImpl</servlet-name>  <!-- url-pattern标签配置访问地址  / 被服务器解析得到：http://ip:port/工程名/  /servletImpl 得到：http://ip:port/工程名/servletImpl-->  <url-pattern>/servletImpl</url-pattern> </servlet-mapping> |

**注意事项**

配置XML文件时如果两个servlet-name标签中的名称不同，则会导致Tomcat服务器无法启动，并且在控制台输出IllegalArgumentException。

配置XML文件时如果servlet-class标签中的全类名配置错误，则会发生ClassNotFoundException。

配置XML文件时url-pattern标签中的名称可以任意输入，在访问时将该名称置于url中的工程名后即可。但是要**注意以斜杠开头**，否则会发生IllegalArgumentException。

### 定位Servlet

对Servlet的定位是通过url地址进行的，URL地址：http://localhost:8080/JavaWeb/servletImpl，其中http表示请求的协议，localhost表示请求访问的服务器的IP地址，8080表示请求通信的程序的端口号，此处表示是tomcat，JavaWeb表示工程名，定位到tomcat上部署的对应的工程，servletImpl表示访问的目标资源。

在web.xml配置文件中，url-pattern表示的是请求对应的目标资源，然后通过servlet-mappig中的servlet-name中的值来找到servlet中对应的servlet-name的值，然后再定位到对应的servlet-class中的Java类，最后调用该类中的service方法处理请求。

### Servlet生命周期

servlet生命周期指的是一个servlet从创建到销毁的一段事件，而这段事件中调用的那些方法就是生命周期方法。

doGet 和 doPost不算生命周期方法，所谓的生命周期方法是指，从对象的创建到销毁一定会执行的方法， 但是这两个方法，不一定会执行。

**Servlet的构造器**  在第一次访问服务器中Servlet程序时调用创建Servlet对象

**init初始化方法** 在servlet对象被创建后立即调用，一次生命周期只会被调用一次

默认情况下，只有在初次访问servlet的时候，才会执行init方法。有时在这个方法里面执行一些初始化工作，甚至是做一些比较耗时的逻辑。这时可能会在init方法中逗留太久的时间，可以在配置时使用load-on-startup元素来指定，给定的数字越小，启动的时机就越早，一般不写负数，从1开始即可，让这个初始化的时机提前一点。

**service方法**  在每次访问Servlet程序被调用，用来处理请求

**destroy方法**  在web工程正常停止时会被调用，如果以暴力方式停止则不会被调用

### Get和Post分发处理

|  |
| --- |
| public void service(ServletRequest servletRequest, ServletResponse servletResponse) throws ServletException, IOException {  HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;  /\* service方法不仅要处理get请求，还要处理post请求  往往get请求和post请求可能要处理不同的业务 \*/  String requestMethod = httpServletRequest.getMethod();  //GET和POST字符串必须要全大写，因为返回的字符串也是大写  if("POST".equals(requestMethod)){  System.*err*.println("this is post request");  }else if("GET".equals(requestMethod)){  System.*err*.println("this is get request");  } } |

## Servlet相关类

### 继承实现关系

|  |
| --- |
| public interface Servlet {}  public abstract class GenericServlet implements Servlet, ServletConfig, Serializable {}  public abstract class HttpServlet extends GenericServlet {}  public class HttpServletSub extends HttpServlet {} |

Servlet接口是JavaWeb程序中的一个组件，描述了Servlet实例需要实现的方法。

GenericServlet抽象类实现了Servlet接口，重写了许多方法。

HttpServlet抽象类继承了GenericServlet抽象类，是具体HTTP请求的servlet分支，在其service()方法中调用getMethod()方法，根据不同的请求方式将其分发到不同的请求方法：doGet()和doPost()。

HttpServletSub为自定义servlet类，用于实现业务功能，只需要重写doGet()和doPost()实现业务功能。

### HttpServlet类

在实际的开发当中，我们一般不会直接实现Servlet接口。而是通过继承HttpServlet类来实现Servlet程序。在doGet方法中调用doPost方法，并不会导致请求方式发生变化，仍然是get请求；反之亦然。

编写一个类去继承HttpServlet程序

重写doGet或doPost方法

在web.xml中去配置访问地址

|  |
| --- |
| //doGet方法专门处理get请求， 在get请求进来的时候会被自动调用 protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  System.*err*.println("this is doGet");  } //doPost方法专门处理post请求，在post请求进来的时候会自动调用 protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  System.*err*.println("this is doPost"); } |

### ServletConfig类

ServletConfig类是由Tomcat来进行创建的。每个Servlet程序都有一个对应的ServletConfig对象实例。它是Servlet程序的配置信息类，可以通过ServletConfig类得到Servlet程序的配置信息。

该类一般用于参数传递给重写的init方法，需要注意的是在重写init方法时，一定要在第一行位置使用super关键字调用父类的init方法，并将该类的对象作为参数传递给父类的init方法。这样父类才能保存ServletConfig的对象实例，使该对象实例可以在子类的其他位置如doGet和doPost中被调用。

1. 获取Servlet程序的别名
2. 获取servlet程序的初始化参数 init-param（有用）
3. 获取ServletContext对象

HttpServletSub.java

|  |
| --- |
| public void init(ServletConfig config) throws ServletException {  super.init(config);  //获取ServletContext对象  System.*err*.println(config.getServletContext().);  //获取servlet的名称  System.*err*.println("servlet-name:"+config.getServletName);  //获取所有servlet程序的初始化参数  Enumeration<String> initParameterNames = config.getInitParameterNames();  while(initParameterNames.hasMoreElements()){  String element = initParameterNames.nextElement();  System.*err*.println("param-name:"+element);  //获取servlet程序的初始化参数的值  System.*err*.println("param-value:"+config.getInitParameter(element));  } } |

web.xml

|  |
| --- |
| <servlet>  <servlet-name>HttpServletSub</servlet-name>  <servlet-class>com.web.servlet.HttpServletSub</servlet-class>  <!-- init-param表示初始化参数，其位置必须位于servlet-class标签后 -->  <init-param>  <!-- param-name配置参数名 -->  <param-name>diy\_init\_param</param-name>  <!-- 参数值 -->  <param-value>diy\_param\_value</param-value>  </init-param>  <!-- 可以配置多个初始化参数 -->  <init-param>  <param-name>diy\_init\_param2</param-name>  <param-value>diy\_param\_value2</param-value>  </init-param> </servlet> <servlet-mapping>  <servlet-name>HttpServletSub</servlet-name>  <url-pattern>/httpServletSub</url-pattern> </servlet-mapping> |

### ServletContext类

ServletContext表示上下文对象，是一个接口，其对象代表一个web工程。一个web工程，只会有一个ServletContext对象实例（Tomcat服务器负责管理）。servletContext对象在web工程启动的时候创建，在web工程停止的销毁。ServletContext对象是一个域对象。

域对象是可以像Map一样存取数据的对象，域指的是作用域，指的是保存到ServletContext域中数据的有效操作范围，是整个web工程。

保存数据 获取数据 删除数据

Map对象 put get remove

ServletContext对象 setAttribute getAttribute removeAttribute

|  |
| --- |
| protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  ServletContext servletContext = getServletContext();  // 获取在web.xml中配置的上下文参数context-param  Enumeration<String> initParameterNames = servletContext.getInitParameterNames();  while (initParameterNames.hasMoreElements()){  String nextElement = initParameterNames.nextElement();  System.*err*.println("param-name:"+nextElement);  System.*err*.println("param-value:"+servletContext.getInitParameter(nextElement));  }  //获取工程部署在服务器上的相对路径  System.*err*.println(servletContext.getContextPath());  //获取工程在硬盘中的真实绝对路径  System.*err*.println(servletContext.getRealPath("/"));  } |

|  |
| --- |
| <context-param>  <param-name>context\_name</param-name>  <param-value>context\_value</param-value> </context-param> <context-param>  <param-name>context\_name2</param-name>  <param-value>context\_value2</param-value> </context-param> |

**像Map一样存储数据**

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  //在一个工程中只有一个ServletContext对象，其是单例的  ServletContext servletContext = getServletContext();  System.*err*.println("old:"+servletContext.getAttribute("context"));  //在所有位置获取或设置Attribute都可以共享  servletContext.setAttribute("context", "newValue");  System.*err*.println("new:"+servletContext.getAttribute("context")); } |

### HttpServletRequest类

HttpServletRequest类由Tomcat负责创建。每次只要有请求进来的时候，Tomcat就会创建一个HttpServletRequest对象，并把所有的请求的http协议内容封闭进去，并传递给Servlet程序使用。

学习HttpServletRequest对象，主要是从这个对象中获取到请求过来的信息。

|  |  |
| --- | --- |
| **HttpServletRequest类方法摘要** | |
| String | **getRequestURI**()获取请求的资源路径 |
| StringBuffer | **getRequestURL**()获取请求的统一资源定位符（绝对路径） |
| String | **getRemoteHost**()获取客户端ip地址 |
| String | **getHeader**()获取请求头信息 |
| String | **getParameter**()获取请求的参数值 |
| String[] | **getParameterValues**()获取请求的参数值（多个值的时候使用） |
| String | **getMethod**()获取请求的方式 GET或POST |
| void | **setAttribute**(key,value) 设置域数据 |
| Object | **getAttribute**(key) 获取域数据 |
| RequestDispatcher | **getRequestDispatcher**()获取请求转发对象 |

setAttribute(String name,Object)：设置名字为name的request的参数值

getAttribute(String name)：返回由name指定的属性值

getAttributeNames()：返回request对象所有属性的名字集合，结果是一个枚举的实例

getCookies()：返回客户端的所有Cookie对象，结果是一个Cookie数组

getCharacterEncoding()：返回请求中的字符编码方式

getContentLength()：返回请求的Body的长度

getHeader(String name)：获得HTTP协议定义的文件头信息

getHeaders(String name)：返回指定名字的request Header的所有值，结果是一个枚举的实例

getHeaderNames()：返回所以request Header的名字，结果是一个枚举的实例

getInputStream()：返回请求的输入流，用于获得请求中的数据

getMethod()：获得客户端向服务器端传送数据的方法

getParameter(String name)：获得客户端传送给服务器端的有name指定的参数值

getParameterNames()：获得客户端传送给服务器端的所有参数的名字，结果是一个枚举的实例

getParametervalues(String name)：获得有name指定的参数的所有值

getProtocol()：获取客户端向服务器端传送数据所依据的协议名称

getQueryString()：获得查询字符串

getRequestURI()：获取发出请求字符串的客户端地址

getRemoteAddr()：获取客户端的IP地址

getRemoteHost()：获取客户端的名字

getSession([Boolean create])：返回和请求相关Session

getServerName()：获取服务器的名字

getServletPath()：获取客户端所请求的脚本文件的路径

getServerPort()：获取服务器的端口号

removeAttribute(String name)：删除请求中的一个属性

**获取请求信息**

|  |
| --- |
| protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  //获取请求的资源路径  System.*err*.println(req.getRequestURI());  //获取请求URL地址  System.*err*.println(req.getRequestURL());  System.*err*.println(req.getRemoteHost()+":"  +req.getRemotePort()+":"  +req.getRemoteAddr());  //获取所有请求头的名称  Enumeration<String> headerNames = req.getHeaderNames();  while (headerNames.hasMoreElements()){  String element = headerNames.nextElement();  System.*err*.print(element + " : ");  //获取指定请求头的值  System.*err*.println(req.getHeader(element));  }  //获取请求方式  System.*err*.println(req.getMethod()); } |

**获取请求参数**

|  |
| --- |
| <form action="http://localhost:8080/JavaWeb/httpServletSub" method="post">  用户名:<input type="text" name="username"/><br/>  密码:<input type="password" name="password"/><br/>  兴趣爱好：  <input type="checkbox" name="hobby" value="java"/>Java  <input type="checkbox" name="hobby" value="cpp"/>C++  <input type="checkbox" name="hobby" value="js"/>JavaScript  <br/>  <input type="submit" /> </form> |

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  //获取所有的请求参数名  Enumeration<String> parameterNames = req.getParameterNames();  while (parameterNames.hasMoreElements()){  String element = parameterNames.nextElement();  System.*err*.print(element + ":" );  //如果请求参数的值有多个，则使用getParameterValues  if("hobby".equals(element)){  String[] parameterValues = req.getParameterValues(element);  System.*err*.println(Arrays.*asList*(parameterValues));  }else {  //如果请求参数的值只有一个，则使用getParameter  System.*err*.println(req.getParameter(element));  }  } } |

#### 请求中文乱码

**通用解决方法**

这种方法可以在Get和Post请求中使用(POST请求有更好的解决方法)。

先使用getBytes方法将字符串转换为字符数组，并使用iso-8859-1编码集作为参数；

再使用String类的构造器将字符数组转换为字符串，并用utf-8编码集作为参数。

|  |
| --- |
| String parameter = req.getParameter(element); byte[] bytes = parameter.getBytes(StandardCharsets.*ISO\_8859\_1*); parameter = new String(bytes, StandardCharsets.*UTF\_8*); |

**解决Get请求乱码方法**

找到Tomcat的配置文件server.xml，找到Connector标签，添加URIEncoding=”UTF-8”

|  |
| --- |
| <Connector URIEncoding="UTF-8"   connectionTimeout="20000"   port="8080"   protocol="HTTP/1.1"   redirectPort="8443"/> |

**解决Post请求乱码方法**

使用setCharacterEncoding()方法传入需要设置的编码集作为参数。这种方法使用时需要注意必须要位于getParameter之前使用，因此建议将该方法位于doPost方法的方法体的第一行。

|  |
| --- |
| //设置请求体的字符集，必须在getParameter之前调用才有效 req.setCharacterEncoding("UTF-8"); |

#### 请求转发

请求转发指的是服务器从一个资源跳转到另一个资源的行为。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 浏览器 |  | Servlet1(服务器) |  | Servlet2(服务器) |
| http://ip:port/WebName/servlet1 | -> | 业务1 | -> | 检查业务1是否完成，完成后执行业务2 |
|  | <- |  | <- | 业务2完成后返回浏览器 |

**特点**

浏览器地址栏不变；所有资源的执行都在一次请求中；共享request域数据；可以转发到WEB-INF目录下的资源；不能跳转到工程以外的资源。

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  req.getRequestDispatcher("/WEB-INF/pageOne.jsp").forward(req,resp); } |

**Web开发中/斜杆**

在Web中路径名如果以/开头，则该/斜杠也表示一个绝对路径，而且根据位置的不同而具有不同的意义。

**服务器解析**

/被服务器解析时，表示的是<http://ip:port/工程名/> ，即映射到代码的WebContent目录，示例如下：

1、<url-pattern>/servlet1</url-pattern>

2、ServletContext.getRealPath(“/”);

3、Request.getRequestDispatcher(“/”)

**浏览器解析**

/被浏览器解析（html页面）时，表示的是http;//ip:port/。

在response.sendRedirct(“/”)重定向中是从服务器中将/交给浏览器去解析，因此得到的是http://ip:port/。

### HttpServletResponse类

每次请求进来，Tomcat服务器除了会创建一个Request对象之外，还会创建一个Response对象。Request是请求的信息。Response是所有返回给客户端的信息。所有返回给客户端的信息，都可以通过Response对象来进行设置。

**输出流**

字节流：getOutputStream() 一般用于下载

字符流：getWriter() 一般用于回传字符串

两个流不能同时使用，一次只能使用一个，否则会抛出IllegalStateException。

**回传数据**

通过获取到的流的write方法，即response.getWriter().write(“hello!!!”);

#### 响应中文乱码

**方案一**

以字符流输出

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  // 设置服务器的字符集为UTF-8  resp.setCharacterEncoding("UTF-8");  // 设置响应头，通知浏览器支持UTF-8查看  resp.setHeader("Content-Type", "text/html;charset=UTF-8");  resp.getWriter().write("我是中文"); } |

**方案二**

以字节流输出

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  // 设置响应头，通知浏览器支持UTF-8查看  resp.setHeader("Content-Type", "text/html;charset=UTF-8");  resp.getOutputStream().write("我是中文"); } |

**方案三**

无论字节流还是 字符流都可以用如下方式

|  |
| --- |
| protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  // 同时设置服务器和客户端的字符集，要求在获取流之前调用  resp.setContentType("text/html; charset=UTF-8");  ... } |

#### 重定向

**步骤**

浏览器通过URL地址访问到服务器中的老程序，此时老程序随着项目的更新迭代，不能完成需求，有新的接口取代。

老程序回传响应状态码302和通过响应头Location回传新请求地址给浏览器。

浏览器接收到响应状态码得知需要访问新程序，然后根据新请求地址访问新程序。

新程序接收到访问请求，完成业务并返回浏览器需要的数据，请求重定向完成。

**特点**

浏览器地址栏发生变化；请求重定向一共会有两次请求，request对象会发生变化；不能共享request域中的数据；不能重定向到WEB-INF下的资源；可以重定向到工程项目外的资源。

方案一

|  |
| --- |
| protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  //设置响应状态码  resp.setStatus(302);  // 设置响应头  resp.setHeader("Location", "http://www.baidu.com"); } |

方案二

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  resp.sendRedirect("http://www.baidu.com"); } |

#### 重定向和请求转发的区别

序号 问题 重定向 请求转发

1 第二次请求谁请求的？ 浏览器 服务器

2 浏览器发送了几次请求？ 2次以上 1次

3 servlet可以共享request吗？ 不可以 可以

4 地址栏是否发生改变？ 是 不是

5 浏览器地址栏显示的哪一次访问地址？ 最后一次 第一次

6 可以跳转到什么资源？ 任意资源 项目内部

7 第二次的请求路径是？ 绝对路径 内部路径

### ServletContextListener

ServletContextListener监听器，监听ServletContext对象的创建和销毁。

#### 创建ServletContextListener监听器

编写一个类去实现ServletContextListener接口（还要实现它的两个方法）。

|  |
| --- |
| public class ServletContextListenerImpl implements ServletContextListener {  @Override  public void contextInitialized(ServletContextEvent sce) {  System.*err*.println("ServletContext Object is initialized");  }  @Override  public void contextDestroyed(ServletContextEvent sce) {  System.*err*.println("ServletContext Object is destroyed");  } } |

到web.xml中去配置监听器

|  |
| --- |
| <listener>  <listener-class>com.web.listener.ServletContextListenerImpl</listener-class> </listener> |

## Cookie

http的请求是无状态。 客户端与服务器在通讯的时候，是无状态的，其实就是客户端在第二次来访的时候，服务器根本就不知道这个客户端以前有没有来访问过。 为了更好的用户体验，更好的交互 [自动登录]，其实从公司层面讲，就是为了更好的收集用户习惯[大数据]。

Cookie是由服务器通知浏览器保存键值对的一种技术，每组键值对大小不能超过4kb。当客户端保存了cookie(键值对)之后，每次请求都会把cookie发送给服务器。Cookie的键和值不能直接的支持中文（空格、方括号、圆括号、等号、逗号、双引号、斜杠、问号、at 符号、冒号和分号。空值在所有浏览器上的行为不一定相同）。

### 创建和获取

在servlet中使用new和Cookie的构造器创建一个Cookie对象，调用response.addCookie(Cookie cookie)将cookie对象添加到响应头Set-Cookie，并以Set-Cookie：key1=value1的形式传递给浏览器，浏览器收到响应后，先检查该cookie是否已存在，如果不存在就创建，如果存在就修改覆盖。

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  Cookie cookie = new Cookie("key", "value");  //cookie对象一定要添加到响应头中，否则不会生效  resp.addCookie(cookie); } |

可以通过调用request.getCookies()方法来获取cookie，该方法返回的是一个cookies对象数组，该数组存储了全部的cookie信息。如果想要获取指定的cookie信息，可以使用for-each循环来获取。

|  |
| --- |
| protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  Cookie[] cookies = req.getCookies();  for (Cookie cookie : cookies) {  System.*err*.println(cookie.getName()+":"+cookie.getValue());  } } |

### 修改cookie

修改cookie的值有两种方式，分别是新建cookie覆盖原先的cookie和调用cookie的setValue方法设置该cookie的值。

**覆盖方式修改**

新建一个Cookie对象，然后在构造器中的key与要覆盖的cookie的key相同，value赋予新值，然后调用response.addCookie(cookie)通知浏览器保存修改。

**setValue方法设置**

调用request.getCookies()获取全部的cookie并遍历查找需要修改的cookie对象，调用该对象的setValue方法设置新的值到该cookie对象中，调用response.addCookie(cookie)通知浏览器保存修改。

|  |
| --- |
| protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  Cookie[] cookies = req.getCookies();  for (Cookie cookie : cookies) {  cookie.setValue("newValue"+new Date());  resp.addCookie(cookie);  } } |

### 生命周期

cookie对象有其生命周期，并且可以用setMaxAge(int expiry)来设置，其中expiry默认值为-1。如果传入的参数expiry为正数，则表示该cookie在指定的秒数后过期被删除；如果expiry为0则表示浏览器接收到响应后马上删除该cookie，不用等浏览器关闭；如果expiry为负数则表示浏览器关闭后就删除该cookie。

|  |
| --- |
| //设置该cookie的有效期为60\*60秒 cookie.setMaxAge(60\*60); //设置该cookie在浏览器接收到响应后立刻删除 cookie.setMaxAge(0); //设置该cookie在浏览器退出时删除 cookie.setMaxAge(-1); |

### 有效路径path的设置

cookie中的path属性可以有效的过滤哪些请求下哪些cookie可以不用发送给服务器，可以节省流量费用。cookie的path属性的默认值是当前工程路径：/工程名，可以通过cookie对象的setPath方法来设置cookie的有效路径。

|  |
| --- |
| cookie2.setPath(req.getContextPath()+"/nothing"); |

ACookie path=/day13 表示只有访问地址为http://ip:port/day13/任意路径 这个cookie都会发送给服务器。

BCookie path=/day13/abc 表示只有访问地址为http://ip:port/day13/abc/任意路径 才会把这个BCookie发送给服务器。

举例：

ACookie：path=/day13 BCookie：path=/day13/abc

请求地址为：http://ip:port/day13/c.html ACookie会发送给服务器 BCookie不会发送给服务器

请求地址为：http://ip:port/day13/abc/c.html ACookie 会发送给服务器 BCookie也会发送给服务器

### Cookie相关方法

setDomain(String domain)用于指定只有请求了指定的域名，才会带上该cookie。

### 实际开发

在实际开发中，cookie一般用于保存用户登陆信息，避免用户每一次登陆都需要输入用户名等信息，优化用户体验。

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  String username = req.getParameter("username");  String password = req.getParameter("password");  if("admin".equals(username) && "admin".equals(password)){  Cookie cookie = new Cookie("username", username);  //设置cookie存活时间为一个星期，该时间段内用户重复登陆都不用再次输入用户名  cookie.setMaxAge(60 \* 60 \* 24 \* 7);  resp.addCookie(cookie);  System.*out*.println("登陆成功");  }else{  System.*out*.println("登陆失败");  } } |

|  |
| --- |
| <%-- 在form标签中，action属性值里不要追加请求的参数 --%> <%-- 如果要添加请求的参数，可以使用隐藏域 --%> <form action="/JavaWeb/httpServletSub" method="post">  <input type="hidden" name="action" value="login"/>  <label for="username">  <%-- 将用户名输入栏的默认值设置为cookie中存储的username的值，  避免用户再次输入，优化用户体验 --%>  username:<input id="username"  type="text"  name="username"  value="${cookie.username.value}"/><br/>  </label>  <label for="password">  password:<input id="password"  type="password"  name="password"><br/>  </label>  <input type="submit"/> </form> |

## Session

是用来维护客户端和服务器之间关联的一种技术。一个Session会话对象维护一个客户端和服务器之间的关联，常将用户登录的信息保存到Session域中。

Cookie是把用户的信息（程序需要的信息）保存到客户端。Session是把用户的信息（程序需要的信息）保存到服务器。

由于Cookie会保存在客户端上，所以有安全隐患问题。 还有一个问题， Cookie的大小与个数有限制。 Session的出现就是为了解决这个问题。

### 创建和获取

创建和获取Session对象的方法都是request.getSession()方法，第一次调用该方法是创建Session对象，第二次调用该方法是获取Session对象。

Session对象的isNew()方法可以用来判断该会话对象是否是刚创建的会话，如果返回true则表明是刚创建的会话对象，如果返回false则表明是获取的会话对象。

Session对象的id属性可用用于区别每一个会话，可以通过Session对象的getID()方法来获取该属性。

Session与数据可以通过Session对象的setAttribute()和getAttribute()对象来存取。

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  HttpSession session = req.getSession();  boolean aNew = session.isNew();  String sessionId = session.getId();  System.*err*.println(aNew + " | " + sessionId);  int i = (int)(Math.*random*() \* 100);  session.setAttribute("value", i);  Object value = session.getAttribute("value");  resp.getWriter().write("i = " + i + "value = " + value); } |

### 生命周期

Session对象和Cookie对象一样有生命周期的概念，可以通过其getMaxInactiveInterval()方法来获取当前Session的超时时间，也可以通过setMaxInactiveInterval(int second)来单独设置当前session的超时时间，传入的参数为-1时表示该session永远不超时。Session的超时指的是客户端和服务器之间两次请求的间隔时长。

Session对象的invalidate()方法可以使当前Session对象马上无效并被销毁。

|  |
| --- |
| protected void deleteNow(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  // session马上无效，保存在里面的数据，也跟着销毁了  request.getSession().invalidate();  response.getWriter().write("当前Session已经被销毁！！！"); } protected void life3Second(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  // 当前Session3秒后超时（超时之后Session就不可用，只能创建新的。）  request.getSession().setMaxInactiveInterval(3);  response.getWriter().write("Session已经被设置为3秒后超时"); } protected void defaultLife(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  // 返回当前Session的超时时间（以秒为单位）  int defaultLife = request.getSession().getMaxInactiveInterval();  response.getWriter().write("Session的默认超时时间为：" + defaultLife + " 秒"); } |

除了通过方法来设置session的超时时间，也可以通过修改配置文件的方式来设置session的超时时间。Tomcat服务器默认的session超时时间为30分钟，在tomcat服务器的配置文件web.xml中修改配置：

|  |
| --- |
| <!-- ==================== Default Session Configuration ================= --> <!-- You can set the default session timeout (in minutes) for all newly --> <!-- created sessions by modifying the value below. --> <session-config>  <session-timeout>30</session-timeout> </session-config> |

除了上述方式，还可以在当前web工程的配置文件web.xml中单独设置session的超时时间：

|  |
| --- |
| <!-- 单独设置你的web工程。所有Session默认的超时时间为30分钟 --> <session-config>  <session-timeout>30</session-timeout> </session-config> |

### B/S关联

Session之所以能够解决客户端和服务器之间关联，是通过Cookie技术来实现的，这个Cookie默认的存活时间为Session，也就是浏览器关闭Cookie就会被删除。

**关联步骤如下：**

1、browser第一次发起请求，创建session对象，此时server中request第一次调用getSession()方法创建session返回

2、在响应中通过cookie技术将创建的session对象的id属性返回给browser，browser接收到响应之后会生成一个名为JSESSIONID的cookie用来保存session对象的id属性；

JSESSIONID 86B5404409C3F642CA578F6295126FBE

3、之后browser每次请求都会在请求头中携带JSESSIONID通知server当前session的id，server中request调用getSession()方法在server中寻找对应id的session返回，如果没有则创建一个新的session对象；

4、如果在发送请求前将JSESSIONID删除，则请求时因为没有该cookie，server中request调用getSession()方法会因为没有获取到session的id信息而直接创建一个新的session对象返回。

要想在浏览器关闭之后同样可以关联上之前创建的Session对象的关键就是让JSESSIONID的cookie在浏览器关闭后不被销毁，也就是修改key是JSESSIONID的Cookie的存活时间，让这个Cookie在浏览器关闭后不被销毁。

Cookie: 前端技术，服务器将数据设置到响应头中交给浏览器，浏览器会保存起来。默认保存一次会话。

Session：后端技术，服务器中的一个对象。保存在服务器中，当浏览器需要使用会话对象时，服务器会创建该对象并交给一个JSESSIONID的cookie给浏览器，浏览器访问服务器只要携带此cookie就能操作服务器中对应的session对象的了。

Session技术离不开cookie，但是它俩不是同一个概念。

### Session相关方法

getId()得到session的id

setAttribute(String name,String value)存储值

getAttribute(String name)获取指定Key的属性值

removeAttribute(String name)移除指定Key即其值

invalidate()强制销毁session

# Listener

Listener监听器是JavaWeb的三大组件之一，监听器的作用：监听某个事物的变化，然后反馈给用户（回调方法）。监听器都是接口。

## ServletContextListener

（一般可以用来完成自定义初始化工作或任务调度）

servletContext创建：启动服务器的时候

servletContext销毁：从服务器移除项目

## ServletRequestListener

request创建：访问服务器上的任意资源都会有请求出现

request销毁：服务器对这次请求作出了响应

## HttpSessionListener

（一般可以用来统计在线人数）

session创建：只要调用getSession()就会创建

session销毁：超时30分钟或关闭服务器

## HttpSessionBindingListener

监听对象与session绑定与解除绑定的动作，让JavaBean实现该接口即可。

## HttpSessionActivationListener

监听现在session的值是钝化（序列化）还是活化（反序列化）的动作。

钝化指的是将内存中的数据存储到硬盘上，活化指的是将硬盘中的数据读取到内存中；其意义在于session中的值可能会很多，并且有很长一段时间不使用这个内存中的值，那么可以考虑把session的值存储到硬盘上（钝化），等下一次在使用的时候，在从硬盘上提取出来（活化）。

配置session在一定时间内钝化的方式有如下三种：

1、在tomcat目录下的conf/context.xml中配置，该配置对所有运行在这个服务器上的项目有效；

2、在conf/Catalina/localhost/context.xml中配置，该配置对localhost生效

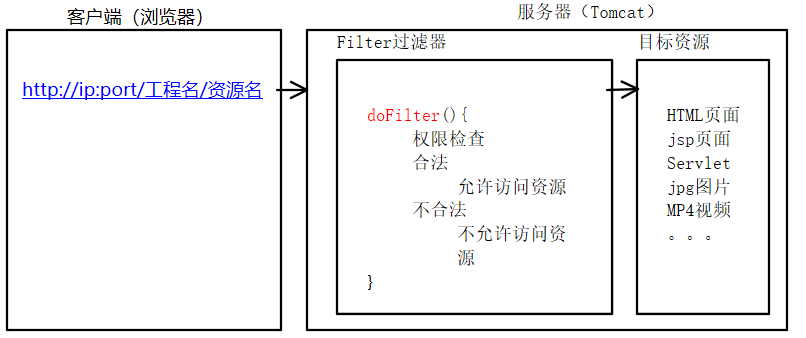
3、在web工程中的META-INF/context.xml中配置，该配置只对当前工程有效

配置时maxIdleSwap表示一分钟不用就钝化，directory表示钝化后文件存放的目录位置。

# Filter

Filter过滤器是JavaWeb三大组件之一。Filter过滤器主要的作用是拦截请求，过滤响应。拦截请求的作用有权限检查、日志处理、敏感词过滤、统一编码和自动登陆等。

客户端发出请求，先经过过滤器， 如果过滤器放行，那么才能到servlet。如果有多个过滤器， 那么他们会按照注册的映射顺序 来 排队。 只要有一个过滤器， 不放行，那么后面排队的过滤器以及咱们的servlet都不会收到请求。



## 实现步骤

创建一个类实现Filter接口

实现Filter接口后需要实现init、doFilter和destroy方法，分别表示初始化方法，过滤方法和销毁方法。

|  |
| --- |
| public class MyFilter implements Filter {  @Override  public void init(FilterConfig filterConfig) throws ServletException {}  @Override  public void doFilter(ServletRequest sreq, ServletResponse resp, FilterChain filterChain) throws IOException, ServletException {}  @Override  public void destroy() {} } |

实现doFilter拦截方法

过滤器中，FilterChain类的doFilter方法表示用户可以访问资源，如果不调用该方法则用户无法下一步访问。

|  |
| --- |
| public void doFilter(ServletRequest sreq, ServletResponse resp, FilterChain filterChain) throws IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) sreq;  HttpSession session = req.getSession();  System.*err*.println(session);  Object user = session.getAttribute("username");  System.*err*.println("filter:"+user);  if(user == null){  sreq.getRequestDispatcher("/index.jsp").forward(sreq, resp);  }else{  filterChain.doFilter(req, resp);  } } |

在web.xml文件中配置拦截路径

在配置文件中配置过滤器的路径和拦截资源。

|  |
| --- |
| <filter>  <filter-name>MyFilter</filter-name>  <filter-class>com.web.filter.MyFilter</filter-class> </filter> <filter-mapping>  <filter-name>MyFilter</filter-name>  <url-pattern>/pageTwo.jsp</url-pattern> </filter-mapping> |

**实例**

|  |
| --- |
| protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  String username = req.getParameter("username");  HttpSession session = req.getSession();  System.*err*.println("servlet:"+username);  session.setAttribute("username", username);  req.getRequestDispatcher("/pageTwo.jsp").forward(req,resp); } |

## 生命周期

1、执行constructor方法

2、执行init初始化方法，init方法的参数 FilterConfig ,可以用于获取filter在注册的名字以及初始化参数。其实这里的设计的初衷与ServletConfig是一样的。

3、执行doFilter过滤器方法

4、执行destroy销毁方法

以上第一、第二步在创建Filter过滤器时执行，即Web工程启动时；第三步每次拦截到请求时就会调用；第四步在停止Web工程时调用执行。

## FilterConfig类

该类类似于ServletConfig类，可以获取Filter过滤器在web.xml文件中的配置信息，主要有三个作用：

1、获取过滤器的别名，即在xml文件中配置的<**filter-name**>标签中的值

2、获取在xml文件中配置的初始化参数的值，即<**param-value**>标签中的值

3、获取ServletContext域对象

|  |
| --- |
| public void init(FilterConfig config) throws ServletException {  System.*out*.println(config.getFilterName());  System.*out*.println(config.getInitParameter("param"));  System.*out*.println(config.getServletContext()); } |

|  |
| --- |
| <filter>  <filter-name>MyFilter</filter-name>  <filter-class>com.web.filter.MyFilter</filter-class>  <init-param>  <param-name>param</param-name>  <param-value>param\_value</param-value>  </init-param> </filter> |

## FilterChain过滤器链

FilterChain是整个Filter过滤器的调用者。Filter与Filter之间的传递，或者Filter与请求资源之间的传递都靠FilterChain.doFilter方法。一般Filter.doFilter中的代码分为三段。

第一段是FilterChain.doFilter之前的代码，一般用来做请求的拦截，检查用户访问的权限，访问日记的记录，参数编码的设置等等操作。

第二段是FilterChain.doFilter方法，此方法可以将代码的执行传递到下一个Filter中，或者是传递到用户最终访问的资源中。

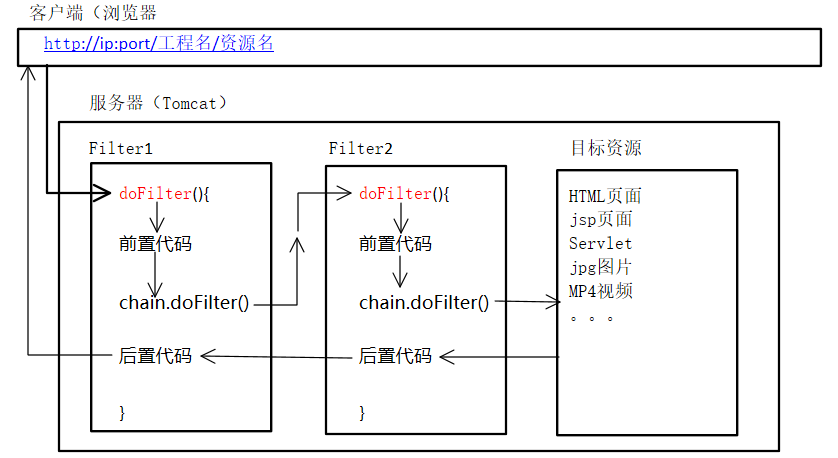
第三段是FilterChain.doFilter之后的代码，主要用过做一些日记操作，一般很少会在第三段中做太多复杂的操作。

在每一个Filter类的doFilter方法中，一定要调用chain.doFilter方法，除非想要阻止用户继续往下面访问，否则一定要调用FilterChain的doFilter方法。

FilterChain的doFilter()方法的作用

1、如果有下一个过滤器，则执行下一个过滤器

2、如果没有下一个过滤器，则执行目标资源代码



**多个过滤器的特点**

1、多个过滤器的执行顺序是其在web.xml文件中从上到下的配置顺序决定的

2、默认情况下，每个过滤器和目标资源使用的请求对象是同一个

3、默认情况下，在同一次请求中，所有代码都默认在同一线程中完成

## 拦截路径

Filter过滤器通过拦截路径的设置，表示如果用户没有通过Filter过滤器，即没有调用FilterChain的doFilter方法，就无法访问到拦截路径下设置的任何资源。

**精确匹配（全路径匹配）**

此时精确匹配到每一个文件，匹配规则要包含文件名及其文件扩展名。

<url-pattern>/index.jsp</url-pattern>这种配置要求访问地址必须是http://ip:port/工程名/index.jsp

**目录匹配（路径匹配、前半段匹配）**

<url-pattern>/img/\*</url-pattern>这种配置要求访问地址必须是http://ip:port/工程名/img/该目录下的所有内容

**后缀名匹配（扩展名匹配）**

<url-pattern>\*.jsp</url-pattern>以上配置要求访问地址必须是要.jsp结尾。才会被拦截到。  
<url-pattern>\*.do</url-pattern>以上配置要求访问地址必须是要.do结尾。才会被拦截到。  
<url-pattern>\*.action</url-pattern>以上配置要求访问地址必须是要.action结尾。才会被拦截到。