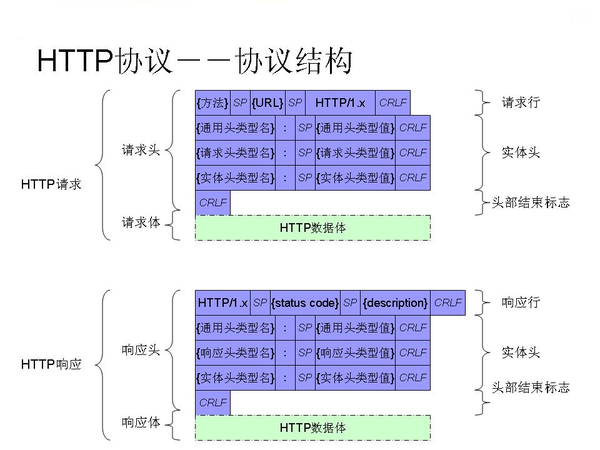
## http简介

HTTP协议（Hyper Text Transfer Protocol超文本传输协议）是用于从万维网（WWW:World Wide Web ）服务器传输超文本到本地浏览器的传送协议。浏览器客户端和服务器端之间数据传输的格式规范。

### http协议内容



### 请求和响应过程

发送http请求报文，web容器如tomcat根据http协议将报文解析并生成HttpServletRequest对象和HttpServletResponse对象传入到web应用中，web服务器根据具体业务对传入的对象进行处理。Request代表请求，Response代表相应。（怎么传入传出的用的是soket网络通信）

## http请求

|  |
| --- |
| GET /day09/hello HTTP/1.1 -请求行  Host: localhost:8080 --请求头（多个key-value对象）  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  Accept-Language: zh-cn,en-us;q=0.8,zh;q=0.5,en;q=0.3  Accept-Encoding: gzip, deflate  Connection: keep-alive  --一个空行  name=eric&password=123456 --（可选）实体内容 |

### 请求行

GET /day09/hello HTTP/1.1

#### http协议版本

http1.0：当前浏览器客户端与服务器端建立连接之后，只能发送一次请求，一次请求之后连接关闭。

http1.1：当前浏览器客户端与服务器端建立连接之后，可以在一次连接中发送多次请求。（基本都使用1.1）

#### 请求资源

URL: 统一资源定位符。http://localhost:8080/day09/testImg.html。只能定位互联网资源。是URI的子集。

URI： 统一资源标记符。/day09/hello。用于标记任何资源。可以是本地文件系统，局域网的资源（//192.168.14.10/myweb/index.html）， 可以是互联网。

#### 请求方式

根据HTTP标准，HTTP请求可以使用多种请求方法。

HTTP1.0定义了三种请求方法： GET, POST 和 HEAD方法。

HTTP1.1新增了五种请求方法：OPTIONS, PUT, DELETE, TRACE 和 CONNECT 方法。

|  |  |  |
| --- | --- | --- |
| 序号 | 方法 | 描述 |
| 1 | GET | 请求指定的页面信息，并返回实体主体。 |
| 2 | HEAD | 类似于get请求，只不过返回的响应中没有具体的内容，用于获取报头 |
| 3 | POST | 向指定资源提交数据进行处理请求（例如提交表单或者上传文件）。数据被包含在请求体中。POST请求可能会导致新的资源的建立和/或已有资源的修改。 |
| 4 | PUT | 从客户端向服务器传送的数据取代指定的文档的内容。 |
| 5 | DELETE | 请求服务器删除指定的页面。 |
| 6 | CONNECT | HTTP/1.1协议中预留给能够将连接改为管道方式的代理服务器。 |
| 7 | OPTIONS | 允许客户端查看服务器的性能。 |
| 8 | TRACE | 回显服务器收到的请求，主要用于测试或诊断。 |

常用的请求方式： GET 和 POST

GET vs POST 区别

GET方式提交：

* 地址栏（URI）会跟上参数数据。以？开头，多个参数之间以&分割。
* GET提交参数数据有限制，不超过1KB。
* GET方式不适合提交敏感密码。
* 浏览器直接访问的请求，默认提交方式是GET方式

|  |
| --- |
| GET /day09/testMethod.html?name=eric&password=123456 HTTP/1.1  Host: localhost:8080  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  Accept-Language: zh-cn,en-us;q=0.8,zh;q=0.5,en;q=0.3  Accept-Encoding: gzip, deflate  Referer: http://localhost:8080/day09/testMethod.html  Connection: keep-alive |

POST方式提交：

* 参数不会跟着URI后面，参数而是跟在请求的实体内容中，多个参数之间以&分割。
* POST提交的参数数据没有限制。
* POST方式提交敏感数据。

|  |
| --- |
| POST /day09/testMethod.html HTTP/1.1  Host: localhost:8080  User-Agent: Mozilla/5.0 (Windows NT 6.1; WOW64; rv:35.0) Gecko/20100101 Firefox/35.0  Accept: text/html,application/xhtml+xml,application/xml;q=0.9,\*/\*;q=0.8  Accept-Language: zh-cn,en-us;q=0.8,zh;q=0.5,en;q=0.3  Accept-Encoding: gzip, deflate  Referer: http://localhost:8080/day09/testMethod.html  Connection: keep-alive  name=eric&password=123456 |

### 请求头

|  |
| --- |
| Accept: text/html,image/\* -- 浏览器接受的数据类型  Accept-Charset: ISO-8859-1 -- 浏览器接受的编码格式  Accept-Encoding: gzip,compress --浏览器接受的数据压缩格式  Accept-Language: en-us,zh- --浏览器接受的语言  Host: www.it315.org:80 --（必须的）当前请求访问的目标地址（主机:端口）  If-Modified-Since: Tue, 11 Jul 2000 18:23:51 GMT --浏览器最后的缓存时间  Referer: http://www.it315.org/index.jsp -- 当前请求来自于哪里  User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0) --浏览器类型  Content-Type: application/x-www-form-urlencoded; charset=UTF-8  Cookie:name=eric -- 浏览器保存的cookie信息，服务端的叫session  Connection: close/Keep-Alive -- 浏览器跟服务器连接状态。close: 连接关闭 keep-alive：保存连接。  Date: Tue, 11 Jul 2000 18:23:51 GMT -- 请求发出的时间 |

#### content-type

定义网络文件的类型和网页的编码，决定浏览器将以什么形式、什么编码读取这个文件，这就是经常看到一些Asp网页点击的结果却是下载到的一个文件或一张图片的原因。

Http Header里的Content-Type一般有这三种：  
application/x-www-form-urlencoded：数据被编码为名称/值对。这是标准的编码格式。  
multipart/form-data： 数据被编码为一条消息，页上的每个控件对应消息中的一个部分。  
text/plain： 数据以纯文本形式(text/json/xml/html)进行编码，其中不含任何控件或格式字符。

form的enctype属性为编码方式，常用有两种：application/x-www-form-urlencoded和multipart/form-data，默认为application/x-www-form-urlencoded。

当action为get时候，浏览器用x-www-form-urlencoded的编码方式把form数据转换成一个字串（name1=value1&name2=value2...），然后把这个字串追加到url后面，用?分割，加载这个新的url。

当action为post时候，浏览器把form数据封装到http body中，然后发送到server。 如果没有type=file的控件，用默认的application/x-www-form-urlencoded就可以了。 但是如果有type=file的话，就要用到multipart/form-data了。

当action为post且Content-Type类型是multipart/form-data，浏览器会把整个表单以控件为单位分割，并为每个部分加上Content-Disposition(form-data或者file),Content-Type(默认为text/plain),name(控件name)等信息，并加上分割符(boundary)。

#### Connection

无状态

HTTP协议是无状态的，指的是协议对于事务处理没有记忆能力，服务器不知道客户端是什么状态。打开一个服务器上的网页和你之前打开这个服务器上的网页之间没有任何联系。HTTP是一个无状态的面向连接的协议，无状态不代表HTTP不能保持TCP连接，更不能代表HTTP使用的是UDP协议（无连接）。

实际中的使用情况：

在web应用中，我们使用http协议，但是我们需要的web是有状态的，因此加入了cookie、session等机制实现有状态的的web

短连接和长连接

在HTTP/1.0中，默认使用的是短连接。浏览器和服务器每进行一次HTTP操作，就建立一次连接，但任务结束就中断连接。如果客户端浏览器访问的某个HTML或其他类型的 Web页中包含有其他的Web资源，如JavaScript文件、图像文件、CSS文件等，当浏览器每遇到这样一个Web资源，就会建立一个HTTP会话。但从 HTTP/1.1起，默认使用长连接，用以保持连接特性。使用长连接的HTTP协议，会在响应头有加入这行代码：Connection:keep-alive在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的 TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接要客户端和服务端都支持长连接。HTTP协议的长连接和短连接，实质上是TCP协议的长连接和短连接。

长连接和短连接的优点和缺点

长连接可以省去较多的TCP建立和关闭的操作，减少浪费，节约时间。对于频繁请求资源的客户来说，较适用长连接。存活功能的探测周期太长，还有就是它只是探测TCP连接的存活，属于比较斯文的做法，遇到恶意的连接时，保活功能就不够使了。

短连接对于服务器来说管理较为简单，存在的连接都是有用的连接，不需要额外的控制手段。但如果客户请求频繁，将在TCP的建立和关闭操作上浪费时间和带宽。

#### Cookie、session

会话技术(Cookie和Session):从打开浏览器到关闭浏览器的过程

作用：

Cookie和Session是为了在无状态的HTTP协议之上维护会话状态，使得服务器可以知道当前是和哪个客户在打交道，因为HTTP协议是无状态的，即每次用户请求到达服务器时，HTTP服务器并不知道这个用户是谁、是否登录过等。现在的服务器之所以知道我们是否已经登录，是因为服务器在登录时设置了浏览器的Cookie！Session则是借由Cookie而实现的更高层(Cookie的局限：

1）Cookie只能存字符串类型。不能保存对象

2）只能存非中文。

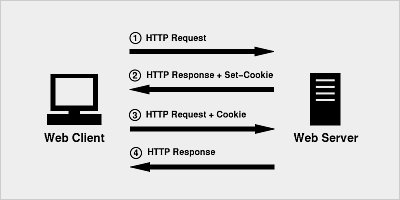
3）1个Cookie的容量不超过4KB。

如果要保存非字符串，超过4kb内容，只能使用session技术)的服务器与浏览器之间的会话,保存在服务器端。

实现机制：

1. 浏览器向某个URL发起HTTP请求，对应的服务器收到该HTTP请求如果是第一次访问创建session对象，给session对象分配一个唯一的ID叫JSESSIONID,把JSESSIONID作为Cookie的值发送给浏览器保存Cookie cookie = new Cookie("JSESSIONID", sessionID),response.addCookie(cookie),返回给浏览器的HTTP响应,在响应头加入Set-Cookie字段，它的值是要设置的Cookie。
2. 浏览器收到来自服务器的HTTP响应。浏览器在响应头中发现Set-Cookie字段，就会将该字段的值保存在内存或者硬盘中。浏览器下次给该服务器发送HTTP请求时， 会将服务器设置的Cookie附加在HTTP请求的头字段Cookie中。

浏览器可以存储多个域名下的Cookie，但只发送当前请求的域名曾经指定的Cookie， 这个域名也可以在Set-Cookie字段中指定。服务器收到这个HTTP请求，发现请求头中有Cookie字段， 便知道之前就和这个用户打过交道了。如果没有则当时第一次请求，继续创建session。过期的Cookie会被浏览器删除，默认的过期时间是用户关闭浏览器时，session默认过期时间是30分钟。



#### Referer

当前请求来自于哪里，可以知道在哪个页面发起的请求，从而防止非法链接(referer)

|  |
| --- |
| **String referer = request.getHeader("referer");**  **System.*out*.println("refer is" + "" + referer);**  **if (referer == null || !referer.contains(request.getServerName())) {**  **request.getRequestDispatcher("/static/error.png").forward(request, response);**  **} else {**  **filterChain.doFilter(request, response);**  **}** |

一般企业级项目都会应用安全框架，如shiro、security等，逻辑和这个差不多。

### 实体内容

只有POST提交的参数会放到实体内容中

### HttpServletRequest

HttpServletRequest对象作用是用于获取请求数据，封装了http发送的请求报文

#### 请求行

request.getMethod(); 请求方式

request.getRequetURI() request.getRequetURL() 请求资源

request.getProtocol() 请求http协议版本

#### 请求头

request.getHeader("名称") 根据请求头获取请求值

request.getHeaderNames() 获取所有的请求头名称

#### 实体内容

request.getInputStream() 获取实体内容数据

### 发起http请求

#### 客户端发送方式

Postmen(谷歌插件)、RestClient、form表单

#### 服务端发送方式

阿帕奇的httpclient、jdk自带的HttpURLConnection

#### 前端ajax发送方式

Asynchronous javascript and xml 异步的 JavaScript 和 XML

|  |
| --- |
| $.ajax({  type : 'post',  dataType : "text",  url : "http://a.a.com/a/FromUserServlet",  data :{“key”:value...},  success : function(msg) {  alert(msg);  }  }); |

#### 区别

普通的B/S模式就是同步，而ajax请求默认是异步的，打断点测试时可能不是按顺序从上往下走返回成功或失败的方法。

浏览器请求会自动处理相应，ajax需要自定义失败和成功的处理

### Http重复提交

#### 场景

场景一：网络延时

在平时开发中，如果网速比较慢的情况下，用户提交表单后，发现服务器半天都没有响应，那么用户可能会以为是自己没有提交表单，就会再点击提交按钮重复提交表单，我们在开发中必须防止表单重复提交。

场景二：重新刷新

表单提交后用户点击【刷新】按钮导致表单重复提交

场景三：点击浏览器的【后退】按钮回退到表单页面后进行再次提交

用户提交表单后，点击浏览器的【后退】按钮回退到表单页面后进行再次提交

#### 解决方案

前段javascript 解决

既然存在上述所说的表单重复提交问题，那么我们就要想办法解决，比较常用的方法是采用JavaScript来防止表单重复提交，具体做

|  |
| --- |
| var isFlag = false; //表单是否已经提交标识，默认为false  function submitFlag() {  if (!isFlag) {  isFlag = true;  return true;  } else {  return false;  } |

除了用这种方式之外，经常见的另一种方式就是表单提交之后，将提交按钮设置为不可用，让用户没有机会点击第二次提交按钮，代码如下：

|  |
| --- |
| function dosubmit(){  //获取表单提交按钮  var btnSubmit = document.getElementById("submit");  //将表单提交按钮设置为不可用，这样就可以避免用户再次点击提交按钮  btnSubmit.disabled= "disabled";  //返回true让表单可以正常提交  return true;  } |

后端服务解决

对于【场景二】和【场景三】导致表单重复提交的问题，既然客户端无法解决，那么就在服务器端解决，在服务器端解决就需要用到session了。

具体的做法：  
1、获取用户填写用户名和密码的页面时向后台发送一次请求，这时后台会生成唯一的随机标识号，专业术语称为Token(令牌)。

2、将Token发送到客户端的Form表单中，在Form表单中使用隐藏域来存储这个Token，表单提交的时候连同这个Token一起提交到服务器端。

3、服务器端判断客户端提交上来的Token与服务器端生成的Token是否一致，如果不一致，那就是重复提交了，此时服务器端就可以不处理重复提交的表单。如果相同则处理表单提交，处理完后清除当前用户的Session域中存储的标识号。

## http响应

|  |
| --- |
| HTTP/1.1 200 OK --响应行  Server: Apache-Coyote/1.1 --响应头（key-vaule）  Content-Length: 24  Date: Fri, 30 Jan 2015 01:54:57 GMT  --一个空行  ＜html＞  ＜head＞  ＜title＞Wrox Homepage＜/title＞  ＜/head＞  ＜body＞ --实体内容  ＜!-- body goes here --＞  ＜/body＞  ＜/html＞ |

### 响应行

#### http协议版本

#### 状态码

相应状态码分类

|  |  |
| --- | --- |
| 1\*\* | 信息，服务器收到请求，需要请求者继续执行操作 |
| 2\*\* | 成功，操作被成功接收并处理 |
| 3\*\* | 重定向，需要进一步的操作以完成请求 |
| 4\*\* | 客户端错误，请求包含语法错误或无法完成请求 |
| 5\*\* | 服务器错误，服务器在处理请求的过程中发生了错误 |

### 常见的响应头

|  |
| --- |
| Location: http://www.it315.org/index.jsp -表示重定向的地址，该头和302的状态码一起使用。  Server:apache tomcat ---表示服务器的类型  Content-Encoding: gzip -- 表示服务器发送给浏览器的数据压缩类型  Content-Length: 80 --表示服务器发送给浏览器的数据长度  Content-Language: zh-cn --表示服务器支持的语言  Content-Type: text/html; charset=GB2312 --表示服务器发送给浏览器的数据类型及内容编码  Last-Modified: Tue, 11 Jul 2000 18:23:51 GMT --表示服务器资源的最后修改时间  Refresh: 1;url=http://www.it315.org --表示定时刷新  Content-Disposition: attachment; filename=aaa.zip --表示告诉浏览器以下载方式打开资源（下载文件时用到）  Transfer-Encoding: chunked  Set-Cookie:SS=Q0=5Lb\_nQ; path=/search --表示服务器发送给浏览器的cookie信息（会话管理用到）  Expires: -1 --表示通知浏览器不进行缓存  Cache-Control: no-cache  Pragma: no-cache  Connection: close/Keep-Alive --表示服务器和浏览器的连接状态。close：关闭连接 keep-alive:保存连接 |

#### Location

http协议之转发和重定向

重定向：当客户端向服务器访问A资源时，服务器返回一个302代码和一个URL给客户端，客户端见到302，立马就向服务器给的URL进行访问。这里的客户端是浏览器，对浏览器发起的请求结果浏览器会自动处理，如果是ajax的发起的请求，那么对于结果的处理则是用户自定义，当服务端重定向时，此时则不会自动发起重定向的请求，需要自定义处理。在没有登录的情况下发起ajax请求，那么spring security设置重定向地址返回response并不会直接跳到重定向的地址。（ajax请求与form表单请求即浏览器请求的区别）

转发：当客户端向服务器请求访问A资源，服务器将其转发到B这，返回B资源；

首先转发是服务器的操作，而重定向是客户端的操作。转发只有一次请求，tomcat等容器内部跳转不暴漏地址而重定向有两次请求，转发只能在站内转发，重定向可以随意定向。

#### Set-Cookie

服务端验证通过后会在响应头加入Set-Cookie字段，然后下次再发请求的时候，浏览器会自动将cookie附加在HTTP请求的头字段Cookie中，服务端就能知道这个用户已经登录过了

### 响应体

有的是页面有的是数据，主要取决于后台设置。

### HttpServletResponse

HttpServletResponse对象封装响应信息，对响应的处理类

响应行：

response.setStatus() 设置状态码

响应头：

response.setHeader("name","value") 设置响应头

实体内容：

response.getWriter().writer(); 发送字符实体内容

response.getOutputStream().writer() 发送字节实体内容

## https基础

Hypertext Transfer Protocol Secure，超文本传输安全协议 ，SSL+HTTP 协议构建的可进行加密传输、身份认证的网络协议，http 和 https 使用的是完全不同的连接方式，用的端口也不一样，前者是 80，后者是 443。

### 工作原理

我们都知道 HTTPS 能够加密信息，以免敏感信息被第三方获取，所以很多银行网站或电子邮箱等等安全级别较高的服务都会采用 HTTPS 协议。

客户端在使用 HTTPS 方式与 Web 服务器通信时有以下几个步骤，如图所示。



　　（1）客户使用 https 的 URL 访问 Web 服务器，要求与 Web 服务器建立 SSL 连接。

　　（2）Web 服务器收到客户端请求后，会将网站的证书信息（证书中包含公钥）传送一份给客户端。

　　（3）客户端的浏览器与 Web 服务器开始协商 SSL 连接的安全等级，也就是信息加密的等级。

　　（4）客户端的浏览器根据双方同意的安全等级，建立会话密钥，然后利用网站的公钥将会话密钥加密，并传送给网站。

　　（5）Web 服务器利用自己的私钥解密出会话密钥。

　　（6）Web 服务器利用会话密钥加密与客户端之间的通信。

### 优缺点

优点：

* http 是超文本传输协议，信息是明文传输，https 则是具有安全性的 ssl(Secure Sockets Layer [安全套接层](https://baike.baidu.com/item/%E5%AE%89%E5%85%A8%E5%A5%97%E6%8E%A5%E5%B1%82" \t "https://baike.baidu.com/item/ssl/_blank))加密传输协议，比 http 协议安全。

缺点：

* HTTPS 协议握手阶段比较费时，会使页面的加载时间延长近 50%，增加 10% 到 20% 的耗电；
* HTTPS 连接缓存不如 HTTP 高效，会增加数据开销和功耗，甚至已有的安全措施也会因此而受到影响；
* http升级https要申请证书,SSL 证书需要钱，功能越强大的证书费用越高，个人网站、小网站没有必要一般不会用。
* SSL 证书通常需要绑定 IP，不能在同一 IP 上绑定多个域名，IPv4 资源不可能支撑这个消耗。
* HTTPS 协议的加密范围也比较有限，在黑客攻击、拒绝服务攻击、服务器劫持等方面几乎起不到什么作用。最关键的，SSL 证书的信用链体系并不安全，特别是在某些国家可以控制 CA 根证书的情况下，中间人攻击一样可行。

## http同源策略

### 概念

同源的定义

如果两个页面的协议，端口（如果有指定）和主机都相同，则两个页面具有相同的源。我们也可以把它称为“协议/主机/端口 tuple”，或简单地叫做“tuple". ("tuple" ，“元”，是指一些事物组合在一起形成一个整体，比如（1，2）叫二元，（1，2，3）叫三元).

同源策略

浏览器内发起的ajax（XMLHttpRequest）请求要符合两个URI同源当且仅当它们的协议://host:port相同，当页面所在的服务和发起请求的服务不同源时边产生了跨域问题。

限制有两种表现：一种是限制发起AJAX请求(XMLHttpRequest，Fetch)，另一种是可以发送请求但拦截其他跨站请求的返回结果，这取决于请求是否为简单请求

### 作用

* 防止csrf攻击（Cross-site request forgery跨站请求伪造）。

Csrf攻击步骤：登录受信任网站A，并在本地生成Cookie，在不登出A的情况下，访问危险网站B，B网站内部发起对A网站的访问，浏览器会带上之前生成的Cookie，这样相当于B网站已经登录了A网站。

* 限制的Dom查询

由于没有同源策略的限制，钓鱼网站可以直接拿到别的网站的Dom const iframe = window.frames['yinhang'] const node = iframe.document.getElementById('你输入账号密码的Input')

同源策略是http的一种安全机制，但没有刺不穿的盾，只是攻击的成本和攻击成功后获得的利益成不成正比。

### 跨域问题

在同源策略下的确能防御非法的攻击，但有的正常需求如前后端分离的项目不同源要能正常访问，这就产生了跨域问题。

### 解决方法

* CORS（Cross-origin resource sharing，跨域资源共享）标准的跨域解决方案

CORS背后的基本思想是使用自定义的HTTP头部允许浏览器和服务器相互了解对方，从而决定请求或响应成功与否.

Access-Control-Allow-Origin:指定授权访问的域  
Access-Control-Allow-Methods：授权请求的方法（GET, POST, PUT, DELETE，OPTIONS等)

|  |
| --- |
| **public class** CORSFilter **implements** Filter {  @Override  **public void** doFilter(ServletRequest servletRequest, ServletResponse servletResponse, FilterChain filterChain) **throws** IOException, ServletException {  HttpServletResponse response = (HttpServletResponse) servletResponse;  HttpServletRequest request = (HttpServletRequest)servletRequest;  response.setHeader(**"Access-Control-Allow-Origin"**, \*);  response.setHeader(**"Access-Control-Allow-Methods"**, **"POST, GET, OPTIONS, DELETE"**);  response.setHeader(**"Access-Control-Max-Age"**, **"3600"**);  response.setHeader(**"Access-Control-Allow-Headers"**, **"x-requested-with,Authorization"**);  response.setHeader(**"Access-Control-Allow-Credentials"**, **"true"**);  response.setHeader(**"Access-Control-Allow-Headers"**,**"Content-Type,..."**);//头部类型  String method = request.getMethod();  **if**(method.equalsIgnoreCase(**"OPTIONS"**)){//预请求  response.setStatus(HttpServletResponse.SC\_OK);  servletResponse.getOutputStream().write(**"OPTIONS returns OK"**.getBytes(**"utf-8"**));  }**else**{  filterChain.doFilter(servletRequest, servletResponse);  }  } } |
|  |

* 设置response的header中的Access-Control-Allow-Origin的值。
* 使用接口网关
* 使用nginx转发。
* 使用JSONP

## XSS攻击

### 概念

[跨站脚本攻击](https://baike.baidu.com/item/%E8%B7%A8%E7%AB%99%E8%84%9A%E6%9C%AC%E6%94%BB%E5%87%BB/8186208" \t "https://baike.baidu.com/item/xss/_blank)(Cross Site Scripting)，为了不和[层叠样式表](https://baike.baidu.com/item/%E5%B1%82%E5%8F%A0%E6%A0%B7%E5%BC%8F%E8%A1%A8" \t "https://baike.baidu.com/item/xss/_blank)(Cascading Style Sheets, [CSS](https://baike.baidu.com/item/CSS/5457" \t "https://baike.baidu.com/item/xss/_blank))的缩写混淆，故将跨站脚本攻击缩写为XSS。恶意攻击者往Web页面里插入恶意的Script代码，当用户浏览该页之时，嵌入其中Web里面的Script代码会被执行，从而达到恶意攻击用户的目的。

示例

|  |
| --- |
| <!DOCTYPE html>  <html>  <head>  <meta charset="utf-8" />  <title></title>  </head>  <body>  <input type="text" id='liuyan'></input>  <button id="btn\_submit">提交</button>  <div>  <li id="show">  </li>  </div>  <script src="js/jquery-1.9.1.min.js"></script>  <script>  $("#btn\_submit").click(function(){  $("#show").html( $("#liuyan").val())  })  </script>  </body>  </html> |

当在输入框中输入<script>alert("你是sb")</script>后提交会有弹框出来

### 解决方法

前段方式

|  |
| --- |
| <script>  $("#btn\_submit").click(function() {  var value = $("#liuyan").val();  var full = value + '';  var dd = full.replace(/<\/?.+?>/g, "");  var dds = dd.replace(/ /g, "");  $("#show").html(dds)  })  </script> |

后端方式

同过fiter

|  |
| --- |
| /\*\*  \* 拦截防止sql注入、xss注入  \*/  public class XssFilter implements Filter {  /\* (non-Javadoc)  \* @see javax.servlet.Filter#doFilter(javax.servlet.ServletRequest, javax.servlet.ServletResponse, javax.servlet.FilterChain)  \*/  public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain)throws IOException,ServletException {    XssHttpServletRequestWrapper xssRequest = new XssHttpServletRequestWrapper(  (HttpServletRequest) request);  filterChain.doFilter(xssRequest, response);  }  }  /\*\*  \* xss过滤  \*/  public class XssHttpServletRequestWrapper extends HttpServletRequestWrapper {  HttpServletRequest orgRequest = null;  public XssHttpServletRequestWrapper(HttpServletRequest request) {  super(request);  orgRequest = request;  }  /\*\*  \* 覆盖getParameter方法，将参数名和参数值都做xss过滤。  \* 如果需要获得原始的值，则通过super.getParameterValues(name)来获取  \* getParameterNames,getParameterValues和getParameterMap也可能需要覆盖  \*/  @Override  public String getParameter(String name) {  String value = super.getParameter(xssEncode(name));  if (value != null) {  value = xssEncode(value);  }  return value;  }  /\*\*  \* 覆盖getHeader方法，将参数名和参数值都做xss过滤。  \* 如果需要获得原始的值，则通过super.getHeaders(name)来获取  \* getHeaderNames 也可能需要覆盖  \*/  @Override  public String getHeader(String name) {  String value = super.getHeader(xssEncode(name));  if (value != null) {  value = xssEncode(value);  }  return value;  }  /\*\*  \* 将容易引起xss漏洞的半角字符直接替换成全角字符  \* @param s  \* @return  \*/  private static String xssEncode(String s) {  if (s == null || "".equals(s)) {  return s;  }  StringBuilder sb = new StringBuilder(s.length() + 16);  for (int i = 0; i < s.length(); i++) {  char c = s.charAt(i);  switch (c) {  case '>':  sb.append('＞');//全角大于号  break;  case '<':  sb.append('＜');//全角小于号  break;  case '\'':  sb.append('‘');//全角单引号  break;  case '\"':  sb.append('“');//全角双引号  break;  case '&':  sb.append('＆');//全角  break;  case '\\':  sb.append('＼');//全角斜线  break;  case '#':  sb.append('＃');//全角井号  break;  default:  sb.append(c);  break;  }  }  return sb.toString();  }  /\*\*  \* 获取最原始的request  \*  \* @return  \*/  public HttpServletRequest getOrgRequest() {  return orgRequest;  }  /\*\*  \* 获取最原始的request的静态方法  \* @return  \*/  public static HttpServletRequest getOrgRequest(HttpServletRequest req) {  if (req instanceof XssHttpServletRequestWrapper) {  return ((XssHttpServletRequestWrapper) req).getOrgRequest();  }  return req;  }  public static void main(String[] args) {  System.out.println(XssHttpServletRequestWrapper.xssEncode("<script>alert(1)</script>"));  } |

总结：

防止攻击就是防止页面展示<script>alert('sb')</script>类似的字符，不管是前台还是后台都是对输入内容是否含有<script>的检测处理。