**Web高级知识-跨域&XSS&CSRF解决方案**

# http长连接与短连接

## HTTP协议与TCP/IP协议的关系

HTTP的长连接和短连接本质上是TCP长连接和短连接。HTTP属于应用层协议，在传输层使用TCP协议，在网络层使用IP协议。IP协议主要解决网络路由和寻址问题，TCP协议主要解决如何在IP层之上可靠的传递数据包，使在网络上的另一端收到发端发出的所有包，并且顺序与发出顺序一致。TCP有可靠，面向连接的特点。

## 如何理解HTTP协议是无状态的

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。也就是说，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）。

## 什么是长连接、短连接？

在HTTP/1.0中，默认使用的是短连接。也就是说，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

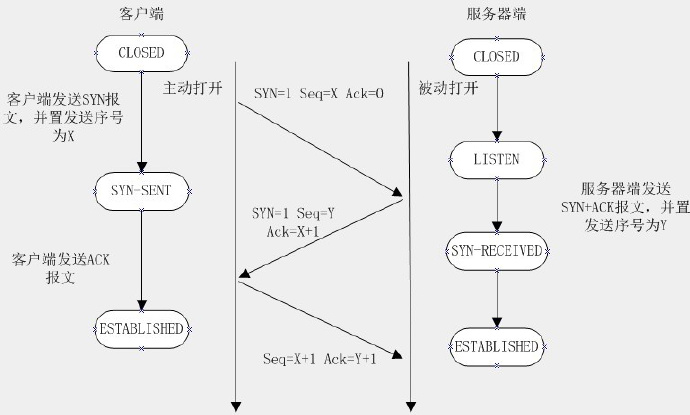
在使用长连接的情况下，当一个网页打开完成后，客户端和服务器之间用于传输HTTP数据的 TCP连接不会关闭，如果客户端再次访问这个服务器上的网页，会继续使用这一条已经建立的连接。Keep-Alive不会永久保持连接，它有一个保持时间，可以在不同的服务器软件（如Apache）中设定这个时间。实现长连接要客户端和服务端都支持长连接。

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。

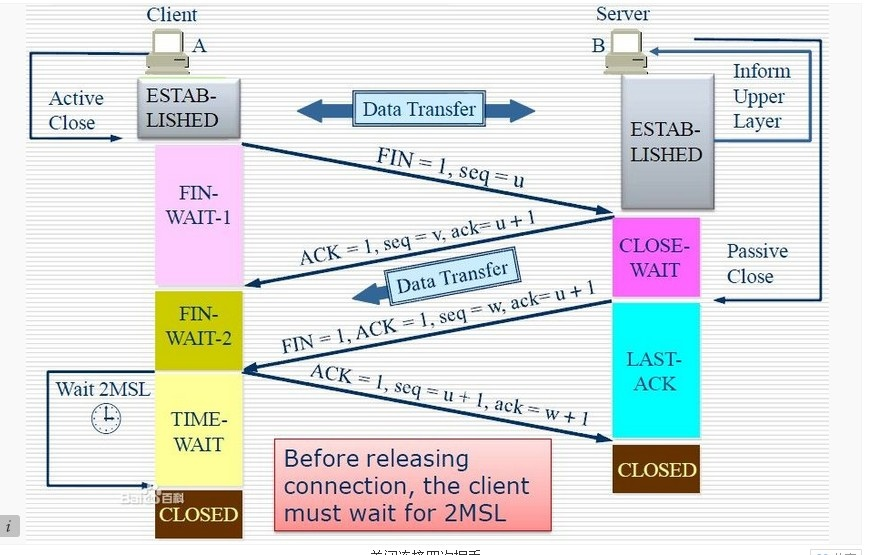
### TCP连接

当网络通信时采用TCP协议时，在真正的读写操作之前，server与client之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接 时它们可以释放这个连接，连接的建立是需要三次握手的，而释放则需要4次握手，所以说每个连接的建立都是需要资源消耗和时间消耗的

经典的三次握手示意图：



经典的四次握手关闭图：



### TCP短连接

我们模拟一下TCP短连接的情况，client向server发起连接请求，server接到请求，然后双方建立连接。client向server 发送消息，server回应client，然后一次读写就完成了，这时候双方任何一个都可以发起close操作，不过一般都是client先发起 close操作。为什么呢，一般的server不会回复完client后立即关闭连接的，当然不排除有特殊的情况。从上面的描述看，短连接一般只会在 client/server间传递一次读写操作

短连接的优点是：管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段

## TCP长连接

接下来我们再模拟一下长连接的情况，client向server发起连接，server接受client连接，双方建立连接。Client与server完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。

首先说一下TCP/IP详解上讲到的TCP保活功能，保活功能主要为服务器应用提供，服务器应用希望知道客户主机是否崩溃，从而可以代表客户使用资源。如果客户已经消失，使得服务器上保留一个半开放的连接，而服务器又在等待来自客户端的数据，则服务器将应远等待客户端的数据，保活功能就是试图在服务 器端检测到这种半开放的连接。

如果一个给定的连接在两小时内没有任何的动作，则服务器就向客户发一个探测报文段，客户主机必须处于以下4个状态之一：

客户主机依然正常运行，并从服务器可达。客户的TCP响应正常，而服务器也知道对方是正常的，服务器在两小时后将保活定时器复位。

客户主机已经崩溃，并且关闭或者正在重新启动。在任何一种情况下，客户的TCP都没有响应。服务端将不能收到对探测的响应，并在75秒后超时。服务器总共发送10个这样的探测 ，每个间隔75秒。如果服务器没有收到一个响应，它就认为客户主机已经关闭并终止连接。

客户主机崩溃并已经重新启动。服务器将收到一个对其保活探测的响应，这个响应是一个复位，使得服务器终止这个连接。

客户机正常运行，但是服务器不可达，这种情况与2类似，TCP能发现的就是没有收到探查的响应。

## 长连接短连接操作过程

短连接的操作步骤是：

建立连接——数据传输——关闭连接…建立连接——数据传输——关闭连接

长连接的操作步骤是：

建立连接——数据传输…（保持连接）…数据传输——关闭连接

## 长连接是什么时候关闭

1. 响应头Keep-Alive: timeout。这个值能够让一些浏览器主动关闭连接，这样服务器就不必要去关闭连接了。
2. tcp自动探测一次，发现对方关闭，则断开连接

## 长连接和短连接的优点和缺点

由上可以看出，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。不过这里存在一个问题，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。在长连接的应用场景下，client端一般不会主动关闭它们之间的连接，Client与server之间的连接如果一直不关闭的话，会存在一个问题，随着客户端连接越来越多，server早晚有扛不住的时候，这时候server端需要采取一些策略，如关闭一些长时间没有读写事件发生的连接，这样可 以避免一些恶意连接导致server端服务受损；如果条件再允许就可以以客户端机器为颗粒度，限制每个客户端的最大长连接数，这样可以完全避免某个蛋疼的客户端连累后端服务。

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。

长连接和短连接的产生在于client和server采取的关闭策略，具体的应用场景采用具体的策略，没有十全十美的选择，只有合适的选择。

## 什么时候用长连接？什么时候用短连接？

长连接多用于操作频繁，点对点的通讯，而且连接数不能太多情况，。每个TCP连接都需要三步握手，这需要时间，如果每个操作都是先连接，再操作的话那么处理速度会降低很多，所以每个操作完后都不断开，次处理时直接发送数据包就OK了，不用建立TCP连接。例如：数据库的连接用长连接， 如果用短连接频繁的通信会造成socket错误，而且频繁的socket 创建也是对资源的浪费。

而像WEB网站的http服务一般都用短链接，因为长连接对于服务端来说会耗费一定的资源，而像WEB网站这么频繁的成千上万甚至上亿客户端的连接用短连接会更省一些资源，如果用长连接，而且同时有成千上万的用户，如果每个用户都占用一个连接的话，那可想而知吧。所以并发量大，但每个用户无需频繁操作情况下需用短连好。

## 长连接与短连接应用场景

长连接应用于场景 客户端（移动App）与服务器消息推送、RPC远程调用

# http请求工具

## 客户端模拟http请求工具

Postmen(谷歌插件)、RestClient

## 服务器模拟http请求工具

httpclient、HttpURLConnection

httpCient请求代码

|  |
| --- |
| **/\*\***  **\* 发送 post请求访问本地应用并根据传递参数不同返回不同结果**  **\*/**  **public void post() {**  **// 创建默认的httpClient实例.**  **CloseableHttpClient httpclient = HttpClients.*createDefault*();**  **// 创建httppost**  **HttpPost httppost = new HttpPost("http://localhost:8080/myDemo/Ajax/serivceJ.action");**  **// 创建参数队列**  **List<NameValuePair> formparams = new ArrayList<NameValuePair>();**  **formparams.add(new BasicNameValuePair("type", "house"));**  **UrlEncodedFormEntity uefEntity;**  **try {**  **uefEntity = new UrlEncodedFormEntity(formparams, "UTF-8");**  **httppost.setEntity(uefEntity);**  **System.*out*.println("executing request " + httppost.getURI());**  **CloseableHttpResponse response = httpclient.execute(httppost);**  **try {**  **HttpEntity entity = response.getEntity();**  **if (entity != null) {**  **System.*out*.println("--------------------------------------");**  **System.*out*.println("Response content: " + EntityUtils.*toString*(entity, "UTF-8"));**  **System.*out*.println("--------------------------------------");**  **}**  **} finally {**  **response.close();**  **}**  **} catch (ClientProtocolException e) {**  **e.printStackTrace();**  **} catch (UnsupportedEncodingException e1) {**  **e1.printStackTrace();**  **} catch (IOException e) {**  **e.printStackTrace();**  **} finally {**  **// 关闭连接,释放资源**  **try {**  **httpclient.close();**  **} catch (IOException e) {**  **e.printStackTrace();**  **}**  **}**  **}**  **public** **static** **void** get() {  CloseableHttpClient httpclient = HttpClients.*createDefault*();  //请求超时  httpclient.~~getParams~~().~~setParameter~~(~~CoreConnectionPNames~~.~~CONNECTION\_TIMEOUT~~, 60000);  //读取超时  httpclient.~~getParams~~().~~setParameter~~(~~CoreConnectionPNames~~.~~SO\_TIMEOUT~~, 60000);  **try** {  // 创建httpget.  HttpGet httpget = **new** HttpGet("http://www.baidu.com/");  System.***out***.println("executing request " + httpget.getURI());  // 执行get请求.  CloseableHttpResponse response = httpclient.execute(httpget);  **try** {  // 获取响应实体  HttpEntity entity = response.getEntity();  System.***out***.println("--------------------------------------");  // 打印响应状态  System.***out***.println(response.getStatusLine());  **if** (entity != **null**) {  // 打印响应内容长度  System.***out***.println("Response content length: " + entity.getContentLength());  // 打印响应内容  System.***out***.println("Response content: " + EntityUtils.*toString*(entity));  }  System.***out***.println("------------------------------------");  } **finally** {  response.close();  }  } **catch** (ClientProtocolException e) {  e.printStackTrace();  } **catch** (ParseException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  } **finally** {  // 关闭连接,释放资源  **try** {  httpclient.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

## 前端ajax请求

|  |
| --- |
| $.ajax({  type : 'post',  dataType : "text",  url : "http://a.a.com/a/FromUserServlet",  data : "userName=余胜军&userAge=19",  success : **function**(msg) {  alert(msg);  }  }); |

# 跨域实战解决方案

跨域原因产生：在当前域名请求网站中，默认不允许通过ajax请求发送其他域名。

## 服务器端代码

|  |
| --- |
| @WebServlet("/FromServlet")  public class FromServlet extends HttpServlet {  @Override  protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  String userName = req.getParameter("userName");  JSONObject jsonObject = new JSONObject();  jsonObject.put("userName", userName);  resp.getWriter().println(jsonObject.toJSONString());  }  } |

## 前端代码

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>**  **<title>B网站访问</title>**  **</head>**  **<script type=*"text/javascript"***  **src=*"http://www.itmayiedu.com/static/common/jquery-1.7.2.min.js?t=2017-07-27"*></script>**  **<script type=*"text/javascript"*>**  **$(document).ready(function() {**  **$.ajax({**  **type : "GET",**  **async : false,**  **url : "http://a.a.com/a/FromServlet?userName=644064",**  **dataType : "json",**  **success : function(data) {**  **alert(data["userName"]);**  **},**  **error : function() {**  **alert('fail');**  **}**  **});**  **});**  **</script>**  **<body>**  **<img alt=*""* src=*"http://a.a.com/a/imgs/log.png"*>**  **</body>**  **</html></script>**  **<body>**  **<img alt=*""* src=*"http://a.a.com/a/imgs/log.png"*>**  **</body>**  **</html>** |

XMLHttpRequest cannot load 跨域问题解决办法

## 解决方案

### 使用后台response添加header

后台response添加header，response.setHeader("Access-Control-Allow-Origin", "\*"); 支持所有网站

### 使用JSONP

前端代码:

|  |
| --- |
| **$.ajax({**  **type : "POST",**  **async : false,**  **url : "http://a.a.com/a/FromUserServlet?userName=张三",**  **dataType : "jsonp",//数据类型为jsonp**  **jsonp : "jsonpCallback",//服务端用于接收callback调用的function名的参数**  **success : function(data) {**  **alert(data.result);**  **},**  **error : function() {**  **alert('fail');**  **}**  **});** |

#### [jquery 中jsonp的实现原理](http://www.cnblogs.com/wqhwe/p/5816941.html)

|  |
| --- |
| 在同源策略下，在某个服务器下的页面是无法获取到该服务器以外的数据的，即一般的ajax是不能进行跨域请求的。但 img、iframe 、script等标签是个例外，这些标签可以通过src属性请求到其他服务器上的数据。利用 script标签的开放策略，我们可以实现跨域请求数据，当然这需要服务器端的配合。 Jquery中ajax 的核心是通过 XmlHttpRequest获取非本页内容，而jsonp的核心则是动态添加 <script>标签来调用服务器提供的 js脚本。  　　当我们正常地请求一个JSON数据的时候，服务端返回的是一串 JSON类型的数据，而我们使用 JSONP模式来请求数据的时候服务端返回的是一段可执行的 JavaScript代码。因为jsonp 跨域的原理就是用的动态加载 script的src ，所以我们只能把参数通过 url的方式传递,所以jsonp的 type类型只能是get ！  示例：  $.ajax({      url: 'http://192.168.1.114/yii/demos/test.php', //不同的域      type: 'GET', // jsonp模式只有GET 是合法的      data: {          'action': 'aaron'      },      dataType: 'jsonp', // 数据类型      jsonp: 'backfunc', // 指定回调函数名，与服务器端接收的一致，并回传回来  })  其实jquery 内部会转化成  http://192.168.1.114/yii/demos/test.php?backfunc=jQuery2030038573939353227615\_1402643146875&action=aaron  然后动态加载  <script type="text/javascript"src="http://192.168.1.114/yii/demos/test.php?backfunc= jQuery2030038573939353227615\_1402643146875&action=aaron"></script>  然后后端就会执行backfunc(传递参数 )，把数据通过实参的形式发送出去。  　　使用JSONP 模式来请求数据的整个流程：客户端发送一个请求，规定一个可执行的函数名（这里就是 jQuery做了封装的处理，自动帮你生成回调函数并把数据取出来供success属性方法来调用,而不是传递的一个回调句柄），服务器端接受了这个 backfunc函数名，然后把数据通过实参的形式发送出去    （在jquery 源码中， jsonp的实现方式是动态添加<script>标签来调用服务器提供的 js脚本。jquery 会在window对象中加载一个全局的函数，当 <script>代码插入时函数执行，执行完毕后就 <script>会被移除。同时jquery还对非跨域的请求进行了优化，如果这个请求是在同一个域名下那么他就会像正常的 Ajax请求一样工作。） |

后端代码:

|  |
| --- |
| **@WebServlet("/FromUserServlet")**  **public class FromUserServlet extends HttpServlet {**  **@Override**  **protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {**  **doPost(req, resp);**  **}**  **@Override**  **protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {**  **resp.setCharacterEncoding("UTF-8");**  **// resp.setHeader("Access-Control-Allow-Origin", "\*");**  **String userName = req.getParameter("userName");**  **String userAge = req.getParameter("userAge");**  **System.*out*.println(userName + "----" + userAge+"---"+req.getMethod());**  **// JSONObject JSONObject1 = new JSONObject();**  **// JSONObject1.put("success", "添加成功!");**  **// resp.getWriter().write("callbackparam(" + JSONObject1.toJSONString()**  **// + ")");**  **try {**  **resp.setContentType("text/plain");**  **resp.setHeader("Pragma", "No-cache");**  **resp.setHeader("Cache-Control", "no-cache");**  **resp.setDateHeader("Expires", 0);**  **PrintWriter out = resp.getWriter();**  **JSONObject resultJSON = new JSONObject(); // 根据需要拼装json**  **resultJSON.put("result", "content");**  **String jsonpCallback = req.getParameter("jsonpCallback");// 客户端请求参数**  **out.println(jsonpCallback + "(" + resultJSON.toJSONString() + ")");// 返回jsonp格式数据**  **out.flush();**  **out.close();**  **} catch (Exception e) {**  **// TODO: handle exception**  **}**  **}**  **}** |

JSONP的优缺点:

JSONP只支持get请求不支持psot请求

### 后台Http请求转发

使用HttpClinet转发进行转发

### 使用接口网关

使用nginx转发。

### 使用SpringCloud网关

# 表单重复提交解决方案(防止Http重复提交

## 场景模拟

创建一个from.jsp页面

|  |
| --- |
| <%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"*  pageEncoding=*"UTF-8"*%>  <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">  <html>  <head>  <title>Form表单</title>  </head>    <body>  <form action=*"*${pageContext.request.contextPath}*/DoFormServlet"* method=*"post"*>  用户名：<input type=*"text"* name=*"userName"*>  <input type=*"submit"* value=*"提交"* id=*"submit"*>  </form>  </body>  </html> |

DoFormServlet 代码

|  |
| --- |
| **import java.io.IOException;**  **import javax.servlet.ServletException;**  **import javax.servlet.annotation.WebServlet;**  **import javax.servlet.http.HttpServlet;**  **import javax.servlet.http.HttpServletRequest;**  **import javax.servlet.http.HttpServletResponse;**  **@WebServlet("/DoFormServlet")**  **public class DoFormServlet extends HttpServlet {**  **@Override**  **protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {**  **req.setCharacterEncoding("UTF-8");**  **String userName = req.getParameter("userName");**  **try {**  **Thread.*sleep*(300);**  **} catch (Exception e) {**  **// TODO: handle exception**  **}**  **System.*out*.println("往数据库插入数据...."+userName);**  **resp.getWriter().write("success");**  **}**  **}** |

### 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。

### 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

### 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

### 使用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，具体做法如下：

修改form.jsp页面，添加如下的JavaScript代码来防止表单重复提交

代码:

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<title>Form表单</title>**  **<script type=*"text/javascript"*>**  **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;**  **}**  **}**  **</script>**  **</head>**  **<body>**  **<form action=*"*${pageContext.request.contextPath}*/DoFormServlet"***  **method=*"post"* onsubmit="return submitFlag()">**  **用户名：<input type=*"text"* name=*"userName"*> <input type=*"submit"***  **value=*"提交"* id=*"submit"*>**  **</form>**  **</body>**  **</html>** |

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;  } |

### 使用后端提交解决

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：**在服务器端生成一个唯一的随机标识号，专业术语称为Token(令牌)，同时在当前用户的Session域中保存这个Token。然后将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，然后在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。**  
 在下列情况下，服务器程序将拒绝处理用户提交的表单请求：

1. **存储Session域中的Token(令牌)与表单提交的Token(令牌)不同。**
2. **当前用户的Session中不存在Token(令牌)。**
3. **用户提交的表单数据中没有Token(令牌)。**

转发代码:

|  |
| --- |
| **@WebServlet("/ForwardServlet")**  **public class ForwardServlet extends HttpServlet {**  **@Override**  **protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {**  **req.getSession().setAttribute("sesionToken", TokenUtils.*getToken*());**  **req.getRequestDispatcher("form.jsp").forward(req, resp);**  **}**  **}** |

转发页面:

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<title>Form表单</title>**  **</head>**  **<body>**  **<form action=*"*${pageContext.request.contextPath}*/DoFormServlet"***  **method=*"post"* onsubmit="return dosubmit()">**  **<input type=*"hidden"* name=*"token"* value=*"*${sesionToken}*"*> 用户名：<input type=*"text"***  **name=*"userName"*> <input type=*"submit"* value=*"提交"* id=*"submit"*>**  **</form>**  **</body>**  **</html>** |

后端Java代码:

|  |
| --- |
| @WebServlet("/DoFormServlet")  **public** **class** DoFormServlet **extends** HttpServlet {  @Override  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  req.setCharacterEncoding("UTF-8");  **boolean** flag = isFlag(req);  **if** (!flag) {  resp.getWriter().write("已经提交...");  System.***out***.println("数据已经提交了..");  **return**;  }  String userName = req.getParameter("userName");  **try** {  Thread.*sleep*(300);  } **catch** (Exception e) {  // **TODO**: handle exception  }  System.***out***.println("往数据库插入数据...." + userName);  resp.getWriter().write("success");  }  **public** **boolean** isFlag(HttpServletRequest request) {  HttpSession session = request.getSession();  String sesionToken = (String) session.getAttribute("sesionToken");  String token = request.getParameter("token");  **if** (!(token.equals(sesionToken))) {  **return** **false**;  }  session.removeAttribute("sesionToken");  **return** **true**;  }  } |

Filter 也称之为过滤器，它是 Servlet 技术中最实用的技术，Web 开发人员通过 Filter 技术，对 web 服务器管理的所有 web 资源：例如 Jsp, Servlet, 静态图片文件或静态 html 文件等进行拦截，从而实现一些特殊的功能。例如实现 URL 级别的权限访问控制、过滤敏感词汇、压缩响应信息等一些高级功能。

它主要用于对用户请求进行预处理，也可以对 HttpServletResponse 进行后处理。使用 Filter 的完整流程：Filter 对用户请求进行预处理，接着将请求交给 Servlet 进行处理并生成响应，最后 Filter 再对服务器响应进行后处理。

# 使用Fileter防止XSS攻击

## 什么是XSS攻击？

XSS攻击使用Javascript脚本注入进行攻击

例如在表单中注入: <script>location.href='http://www.itmayiedu.com'</script>

**注意:谷歌浏览器 已经防止了XSS攻击，为了演示效果，最好使用火狐浏览器**

实例:

演示:

代码: fromToXss.jsp

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>**  **<title>Insert title here</title>**  **</head>**  **<body>**  **<form action=*"XssDemo"* method=*"post"*>**  **<input type=*"text"* name=*"userName"*> <input type=*"submit"*>**  **</form>**  **</body>**  **</html>** |

代码: XssDemo

|  |
| --- |
| **import** java.io.IOException;  **import** javax.servlet.ServletException;  **import** javax.servlet.annotation.WebServlet;  **import** javax.servlet.http.HttpServlet;  **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  @WebServlet("/XssDemo")  **public** **class** XssDemo **extends** HttpServlet {  @Override  **protected** **void** doPost(HttpServletRequest req, HttpServletResponse resp) **throws** ServletException, IOException {  String userName = req.getParameter("userName");  req.setAttribute("userName", userName);  req.getRequestDispatcher("showUserName.jsp").forward(req, resp);  }    } |

代码: showUserName.jsp

|  |
| --- |
| **<%@ page language=*"java"* contentType=*"text/html; charset=UTF-8"***  **pageEncoding=*"UTF-8"*%>**  **<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">**  **<html>**  **<head>**  **<meta http-equiv=*"Content-Type"* content=*"text/html; charset=UTF-8"*>**  **<title>Insert title here</title>**  **</head>**  **<body>userName:${userName}**  **</body>**  **</html>** |

## 解決方案

使用Fileter过滤器过滤器注入标签

XSSFilter

|  |
| --- |
| **public** **class** XssFiter **implements** Filter {  **public** **void** init(FilterConfig filterConfig) **throws** ServletException {  }  **public** **void** doFilter(ServletRequest request, ServletResponse response, FilterChain chain)  **throws** IOException, ServletException {  HttpServletRequest req = (HttpServletRequest) request;  XssAndSqlHttpServletRequestWrapper xssRequestWrapper = **new** XssAndSqlHttpServletRequestWrapper(req);  chain.doFilter(xssRequestWrapper, response);  }  **public** **void** destroy() {  }  } |

**XssAndSqlHttpServletRequestWrapper**

|  |
| --- |
| **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletRequestWrapper;  **import** org.apache.commons.lang3.StringEscapeUtils;  **import** org.apache.commons.lang3.StringUtils;  /\*\*  \* 防止XSS攻击  \*/  **public** **class** XssAndSqlHttpServletRequestWrapper **extends** HttpServletRequestWrapper {  HttpServletRequest request;  **public** XssAndSqlHttpServletRequestWrapper(HttpServletRequest request) {  **super**(request);  **this**.request = request;  }  @Override  **public** String getParameter(String name) {  String value = request.getParameter(name);  System.***out***.println("name:" + name + "," + value);  **if** (!StringUtils.*isEmpty*(value)) {  // 转换Html  value = StringEscapeUtils.*escapeHtml4*(value);  }  **return** value;  }  } |

作业:使用Cookie实现自动登录效果。

本节课用到maven坐标

|  |
| --- |
| <!-- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3 -->  <dependency>  <groupId>org.apache.commons</groupId>  <artifactId>commons-lang3</artifactId>  <version>3.4</version>  </dependency> |