# Ajax技术

## 1.1 同步和异步

**同步：**提交请求->等待服务器处理->服务器响应->客户端页面载入 这个期间客户端浏览器不能干任何事

**异步:** 请求通过事件触发->服务器处理（这时浏览器仍然可以干其他事情）->服务器响应->部分更新网页(不重新加载)

## 1.2 XMLHttpRequest对象

XMLHttpRequest对象是Ajax的一个关键对象，它被用来和后台进行数据交换，它能够：

* 在不重新加载页面的情况下更新网页
* 在页面已加载后从服务器请求数据
* 在页面已加载后从服务器接收数据
* 在后台向服务器发送数据

### 1.2.1 HTTP请求和响应

http是一个无状态的协议，即服务器端不保留链接的相关信息，再次链接必须重新发送相关信息

**1.HTTP请求格式：**

**(1)请求行**：请求行是请求消息的第一行，用于区分是请求报文还是相应报文，由三部分组成：分别是请求方法（GET/POST/DELETE/PUT/HEAD）、请求资源的URI路径、HTTP的版本号，用空格分离，末尾有回车换行符

**GET方法：用于信息获取，对发送信息数量有限制，**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

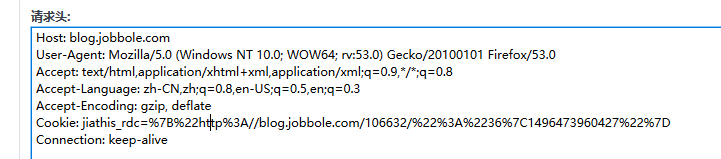
eg: GET /test/demo\_form.asp?name=DJ&password=123456 HTTP/1.1 (CRLF)

注意：get方法发送的信息(参数)都通过URL传递，因此对所有人可见，不安全

**POST方法：用于修改服务器上的资源，不限制发送信息数量，**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，有请求体的HTTP请求一般使用post而不是get。

eg：GET /index.html HTTP/1.1

**(2)请求头：**用于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，请求头发送结束后浏览器会发送一段空行告诉服务器它已经结束了请求头的发送。eg:



**(3)请求体：**

请求体是客户端发给服务端的请求数据，这部分数据并不是每个请求必须的，比如提交表单时或查询字符串等就要发送请求体，有该部分应使用post方法，避免使用get方法，eg：



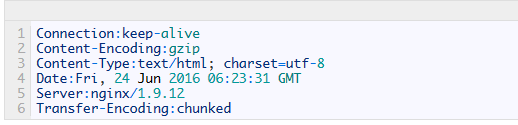
**2.HTTP响应格式：**

**(1)状态行：**状态行位于相应消息的第一行，有HTTP协议版本号，状态码和状态代码文本说明三部分构成。比如notFound404就是状态码的一种，eg：

**HTTP/1.1 200 OK**

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

**(2)响应头：**响应头是服务器传递给客户端用于说明服务器的一些信息，以及将来继续访问该资源时的策略。eg：



**(3)响应体**：响应体是服务端返回给客户端的HTML文本内容，或者其他格式的数据，比如：视频流、图片或者音频数据。

**3.HTTP连接建立过程**

一个完整的HTTP请求过程，通常有以下7个步骤：

1. 建立TCP连接(三次握手过程建立TCP连接)
2. 浏览器向web服务器发送HTTP请求命令(即请求行)
3. 浏览器向web服务器发送HTTP请求头信息，发送完成后浏览器发送一空白行来通知服务器，它已经结束了该头信息的发送。
4. web服务器处理并应答(发送应答头)
5. web服务器发送应答b头信息，发送完成后浏览器同样发送一空白行来通知服务器，它已经结束了该头信息的发送。
6. web服务器向浏览器发送数据
7. 发送完毕后，除非在头信息设置了Connection:keep-alive ，否则将关闭TCP连接

### 1.2.2 XMLHttpRequest发送HTTP请求

**1.open()方法**

**语法：**

**open(method,url,async)**

**参数method表示发送的http请求使用的请求方法，url是发送请求的地址，可以使用相对应文档的地址，也可以使用绝对地址，async表示该http请求是同步的还是异步的，异步将其设置为true，同步设置为false，默认为true，因此一般省略第三个参数**

**2.setRequestHeader()方法**

**语法：**

**setRequestHeader(name,value)**

**通过该方法可以修改HTTP请求的请求头，name是请求头的某个属性名，value是要修改的值**

**注意：该方法一定要写在open()和send()中间，否则将抛出异常**

**eg：**var xhr=new XMLHttpRequest();

xhr.open(“POST”,”post.php”);

**xhr.setRequestHeader(“Content-type”,”application/x-www-form-urlencoded”);//告诉服务器发送的是表单**

xhr.send(“name=DJ&password=123”);

**3.send()方法**

**语法：**

**send(string)**

**通过该方法发送请求体，string是请求体的内容，get方法一般没有请求体，因此使用get方法open的XMLHttpRequest调用send方法时一般将参数设为null，因为get方法通过url传递参数而不是请求体**

**eg：**var xhr1=new XMLHttpRequest();

xhr1.open(“GET”,”get.php?name=DJ&password=123”,true);

xhr1.send(null);

var xhr2=new XMLHttpRequest();

xhr2.open(“POST”,”post.php”);

xhr2.send(“name=DJ&password=123”);

### XMLHttpRequest获取响应

**1.responseText属性以字符串形式返回http响应数据，“responseXML”属性以xml文档形式返回http响应数据**

**2.status属性以数字形式返回http响应的状态码，如：200(代表ok)，404(代表notfound)，statusText属性以文本形式返回http响应的状态码，如：ok(代表200)，Not Found (代表404)**

**3.getAllResponseHeader()方法获取所有的响应报头，getResponseHeader()查询响应报头中某个字段的值**

**4.readyState属性的变化代表服务器响应的变化**

**当readyState=0时：请求未初始化，open()未调用【初始状态】**

**当readyState=1时：服务器连接已建立，准备发送请求，open()已调用，并已调用send()方法发送请求体【第1、2、3步】**

**当readyState=2时：请求已发送完毕并被服务器接收，但还未接收到响应**

**当readyState=3时：数据传输中，已接收到响应头，但还未完全接收到响应体【第4、5步】**

**当readyState=4时：响应数据传输完成，此时使用responseBody和responseText等不会出错**

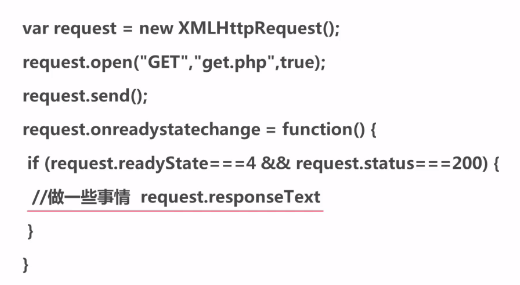
**5.onreadystatechange事件可以监听readyState属性的改变，eg：**

var xhr=new XMLHttpRequest();

xhr.onreadystatechange=function(){

alert("readyState已改变，当前值为"+xhr.readyState);

}



## 1.3 json格式

### 1.3.1 json概念

JSON是JavaScript对象表示法(JavaScript Object Notation)，它是存储和交换信息的语法，通过键值对的方式进行组织，易于解析和阅读，同时，json具有跨平台的特性，无论后台是什么语言，只要按照json的语法规则来，就能正常解析

**JSON和XML的区别：**

**相比于XML，JSON具有短小、灵活的特点，并且读写较快，JSON可以通过JavaScript的内置方法直接转化为对象**

### 1.3.2 json语法规则

**语法：**

**{“name”:”value”}**

**注意：name和value都必须加上””。**

**eg：**

{//json对象

"staff ": [//json内部的键值对

{"name": "DJ","age": 70},//json对象

{"name": "张三","age": 60}, //json对象

{"name": "李四","age": 50}//json对象

]

}

### 1.3.3 json的解析、格式化和校验

在JavaScript中，解析json有两种方法：eval和JSON.parse

**1.eval()解析**

通过eval()解析不会查看json格式是否合法，而且json字符串中若有JavaScript方法也会被执行，十分危险，尤其是使用eval()执行第三方json代码，可能会执行一些恶意代码，除非能够确定eval的参数可控，否则应避免使用eval()

eg：

**2.** **JSON.parse()解析**

JSON.parse()会给程序员预报json格式的错误，并且不会执行json中的方法(直接报错)，相对安全，推荐使用

注意：json格式写起来较为复杂，容易出错，推荐一款在线json校验工具JSONLint

eg:

JSON.parse(xhr.responseText);

## 1.4 jQuery中的AJAX

**语法：**

jQuery.ajax([settingObject]);

type指定http请求类型，”POST”或”GET”，默认为”GET”

url是发送的后台地址

data是发送给后台的数据

dataType是发送给后台的数据的类型，一般采用JSON字符串

success请求成功的回调函数，传入返回数据(是已经处理好的json格式; )以及包含成功代码的字符串，该方法有两个参数方法有两个参数：服务器返回数据，返回状态

function (data, textStatus)

error请求失败的回调函数，传入XMLHttpRequest对象，参数有以下三个：XMLHttpRequest 对象、错误信息、（可选）捕获的错误对象。

error：function (XMLHttpRequest, textStatus, errorThrown)

**注意：jQuery的验证插件  jquery.validate.min.js有bug，会导致返回的json字符出现问题无法解析，error函数抛出textStatus为parsererror的异常，只需要将dataType设置为dataType**:**'text json'即可解决问题**

**eg:**

function submitForm() {

var value = $("input:first").val();

var data;

data = ("name=" + value + "&");

value = $("input:eq(1)").val();

data += ("number=" + value + "&");

value = $("input:eq(2)").val();

data += ("sex=" + value + "&");

value = $("input:eq(3)").val();

data += ("job=" + value);

$.ajax({

type: "POST",

url: url,

dataType: "json",

data: data,

success: function(responseData) {

var success;

var msg;

var showText;

var success = responseData.success;

var msg = responseData.msg;

if (success) {

showText = msg;

} else {

showText = "ERROR!" + msg;

}

$("#showResult1").text(showText);

},

error: function() {

/\* Act on the event \*/

$("#showResult1").text(xhr.status + " " + xhr.statusText);

}

})

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

document.getElementById("a").onclick = function() {

$.ajax({

type: "GET",

url: "server.php" + "?number=" + $("input[name='number']:eq(1)").val(),

data: null,

dataType: "json",

success: function(responseData) {

var id;

var name;

var sex;

var job;

var success = responseData.success;

var msg = responseData.msg;

var JqSelecterStr = "#showResult tr:eq(1) td:eq("; //jQuery选择器字符串

if (success) {

//显示表格

$("#showResult").css("display", "block");

//处理字符串

msg = msg.slice(msg.indexOf("：") + 1, msg.length); //将头部的"找到员工"去掉

var valueArr = new Array();

var keyValueArr = msg.split('，'); //根据,分割字符为“属性-值”数组

for (i = 0; i < keyValueArr.length; i++) {

valueArr[i] = (keyValueArr[i].split("："))[1];

//将获取的值添加进表中

JqSelecterStr += i + ")";

$(JqSelecterStr).text(valueArr[i]);

JqSelecterStr = "#showResult tr:eq(1) td:eq(";

}

$("#showLabel").text("找到员工！");

} else {

$("#showLabel").text("没有找到员工");

$("#showResult").css("display", "none");

}

},

error: function() {

$("#showResult").text(xhr.status + " " + xhr.statusText);

}

})

}

## 1.5 跨域



abc.com是顶级域名，www.abc.com或mail.ahc.com是二级域名，是子域名的一种

一般来说端口号是80，如果端口号是80即可省略不写，当协议，子域名，主域名，端口号中任意一个不相同时，两个域名地址就分属不同的域名，不同域之间相互请求资源称为跨域请求，JavaScript禁止跨域请求

localhost和127.0.0.1虽然都指向本机,但也属于跨域，在1.3的例子中，将查询页面的url变量值都改成http://127.0.0.1:8080/AJAXSimple/server.php，通过localhost访问前端查询页面，此时查询前端页面在web服务器上(localhost)，后台server.php也在web服务器上(127.0.0.1)二者之间调用服务(查询页面.html想调用sever.php的服务)就属于跨域

### 1.5.1 代理跨域

通过后台代理，可以解决跨域问题，比如某个上海的服务器有某个服务[www.shanghai.com/server.php](http://www.shanghai.com/server.php),访问某北京服务器的前端页面想要调用上海服务器的这个服务，可以通过北京的服务器进行后台代理，[www.beijing.com/proxy-shanghaiSever.php](http://www.beijing.com/proxy-shanghaiSever.php)，通过访问该代理文件来获取响应服务，再由该代理服务器将响应发送给前台

### 1.5.2 JSONP

JSONP可通过前端技术解决跨域访问问题，但它**只能用于GET请求**

JSONP是一种使用JSON数据的方式，返回的不是JSON对象，是包含JSON对象的javaScript脚本。即在跨域的服务端生成JSON数据，然后包装成script脚本回传，然后在本页面引入，就能实现跨域访问

jQuery使用jsonp的方法十分简单，只需要将$.ajax({})中的dataType属性设为”jsonp”，给jsonp取一个名字,jsonp:”随便取名字”,即可(后台需要通过jsnop的名字获取jsonp的值，然后加在响应体前面，并将原本的响应体用括号括起来)，eg：

type: "GET",

url: "server.php" + "?number=" + $("input[name='number']:eq(1)").val(),

data: null,

dataType: "jsonp",

jsonp:"MyJsonp",

请求头给我们添加了参数MyJsonp

localhost:8080/AJAXSimple/server.php?number=103&MyJsonp=jQuery32108802286077924375\_1497086355815&\_=1497086355817

返回体为：

jQuery32108802286077924375\_1497086355815({"success":true,"msg":"找到员工：员工编号：103，员工姓名：黄蓉，员工性别：女，员工职位：产品

经理"})

### 1.5.2 XHR2

XML5提供了xmlHttpRequest level2(简称xhr2)实现了跨域访问，但缺点是IE10以下不兼容

方法非常简单，只需要在被调用端服务的php代码最上面添加

header(“Access-Control-Allow-Origin:\*”);//\*代表所有域名都能访问，也可以设置可访问域名

header(“Access-Control-Allow-Methods:POST,GET”)

### 1.5.2FormData

XMLHttpRequest Level 2添加了一个新的接口FormData.利用FormData对象,我们可以通过JavaScript用一些键值对来模拟一系列表单控件,我们还可以使用XMLHttpRequest的send()方法来异步的提交这个”表单”.比起普通的ajax,使用FormData的最大优点就是我们可以异步上传一个二进制文件.

所有主流浏览器的较新版本都已经支持这个对象了，比如Chrome 7+、Firefox 4+、IE 10+、Opera 12+、Safari 5+。之前都是用原生js的XMLHttpRequest写的请求

XMLHttpRequest方式

xhr.open("POST", uri, true);

xhr.onreadystatechange = function() {

if (xhr.readyState == 4 && xhr.status == 200) {

// Handle response.

alert(xhr.responseText); // handle response.

}

};

fd.append('myFile', file);

// Initiate a multipart/form-data upload

xhr.send(fd);

其实jquery的ajax也可以支持到的，关键是设置：processData 和 contentType 。

ajax方式

var formData = new FormData();

var name = $("input").val();

formData.append("file",$("#upload")[0].files[0]);

formData.append("name",name);

$.ajax({

url : Url,

type : 'POST',

data : formData,

// 告诉jQuery不要去处理发送的数据

processData : false,

// 告诉jQuery不要去设置Content-Type请求头

contentType : false,

beforeSend:function(){

console.log("正在进行，请稍候");

},

success : function(responseStr) {

if(responseStr.status===0){

console.log("成功"+responseStr);

}else{

console.log("失败");

}

},

error : function(responseStr) {

console.log("error");

}

});

# WebSocket技术

## 2.1 WebSocket概述

### 2.1.1 常见概念

首先来看一组常见的概念

**单工：**单工数据传输只支持数据在一个方向上传输；在同一时间只有一方能接受或发送信息，不能实现双向通信，举例：电视，广播。

**半双工：**半双工数据传输允许数据在两个方向上传输,但是,在某一时刻,只允许数据在一个方向上传输,它实际上是一种切换方向的单工通信；在同一时间只可以有一方接受或发送信息，可以实现双向通信。举例：对讲机。

**全双工：**全双工数据通信允许数据同时在两个方向上传输,因此,全双工通信是两个单工通信方式的结合,它要求发送设备和接收设备都有独立的接收和发送能力；在同一时间可以同时接受和发送信息，实现双向通信，举例：电话通信。

**轮询：**客户端和服务器之间会一直进行HTTP的连接和释放，每隔一段时间client就询问一次server，一个接受，一个发送，不管server有没有消息，在收到client的请求后都要立即返回。而且每次发送请求都会有Http的Header，会很耗流量，也会消耗CPU的利用率

**长轮询：**长轮询是对轮询的改进版，客户端发送HTTP给服务器之后，如果没有新消息，就一直等待。当有新消息的时候，才会返回给客户端。在某种程度上减小了网络带宽和CPU利用率等问题。但是这种方式还是有一种弊端：例如假设服务器端的数据更新速度很快，服务器在传送一个数据包给客户端后必须等待客户端的下一个请求到来，才能传递第二个消息给客户端，这种串行按序访问，效率极其低下。另外，由于http数据包的头部数据量往往很大，但是真正被服务器需要的数据却很少，极度浪费带宽。

**短连接：**HTTP/1.0中默认使用短连接，即客户端和服务器每进行一次HTTP访问，就建立一次连接，访问结束就中断连接。这种端连接在client和server交流频繁时会大量浪费资源

**长连接：**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，客户端再次访问这个服务器时，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。实现长连接需要客户端和服务端都支持长连接。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，但长连接如果长时间不关闭的话，会对server造成较大压力。

### 2.1.2 http协议和webSocket

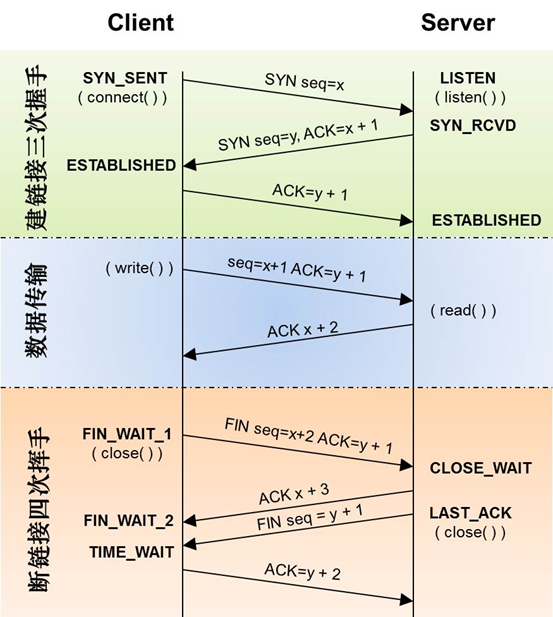
#### 2.1.2.1 http协议的三次握手和四次挥手

HTTP协议，全名超文本传输协议(Hyper Text Transfer Protocol)，是一个基于TCP/IP的应用层(OSI第七层，TCP/IP第五层)的协议，它被用于传递数据（HTML 文件, 图片文件, 查询结果等），是我们最常用的通信协议

客户端和服务器建立HTTP连接，通常是通过三次握手建立的。假设现有一台client希望和一台server通信：

1. 第一次握手：client发送SYN报文到server，并进入**SYN\_SEND**状态，等待server确认
2. 第二次握手：server发送ACK确认报文 和 SYN报文到client，此时server进入**SYN\_RECV**状态
3. 第三次握手：client收到server的SYN+ACK报文，向服务器发送ACK确认报文，此报文发送完毕后，客户端和服务器就进入了**ESTABLISHED**状态，三次握手完成。
4. 第一次挥手：client发送FIN报文到服务器，不再发送数据，主动关闭TCP连接，进入**FIN-WAIT-1**状态
5. 第二次挥手：server收到FIN报文后，发出ACK确认报文，server进入**CLOSE-WAIT**状态，client到server的连接释放。client收到ACK确认报文后，进入**FIN-WAIT-2**状态，继续等待server发出服务器到客户端的FIN释放报文
6. 第三次挥手：server在发送ACK确认报文后，等待自己将剩余的数据发送完毕，然后向client发送FIN释放报文，同时server进入**LAST-ACK**状态
7. 第四次挥手：client收到FIN释放报文后，向server发出ACK确认报文，接着client进入**TIME-WAIT**状态，等待2MSL后，进入**CLOSE**状态

小结：为什么最后要等待2MSL呢？因为在client发出最后的ACK报文后，该ACK报文可能丢失，Server如果没有收到ACK，将不断重复发送FIN片段，所以Client必须在确认Server接收到了该ACK后才能关闭。并且，2MSL后，四次挥手所产生的所有报文都会超时，这样如果立即进行下一次HTTP连接，那么下次连接的三次握手将不会被四次挥手中的报文干扰。



#### 2.1.2.2 webSocket概述

HTTP解决了互联网客户端和服务器端的通信问题。但渐渐的，其缺陷也暴露出来：**通信只能由客户端发起**

这种单向发起请求的特点，使得客户端无法通过HTTP及时获知服务端的最新数据(消息)。最开始，大家通过轮询来解决问题，但后来发现，轮询效率过于低下，出现了长轮询，渐渐地，长轮询也无法满足需求，webSocket应运而生。

WebSocket是一种全双工协议，client和server只需要完成一次握手，两者之间就直接可以创建持久性的连接，并进行双向数据传输，即server 与 client 都能主动向对方发送或接收数据，直到有一方断开连接

1.首先，client 发起 WebSocket 连接，报文类似于 HTTP，但主要有几点不一样的地方：

**Upgrade: websocket"：** 表明这是一个 WebSocket 类型请求，意在告诉 server 需要将通信协议切换到 WebSocket

**"Sec-WebSocket-Key: \*":** 是 client 发送的一个 base64 编码的密文，要求 server 必须返回一个对应加密的 "Sec-WebSocket-Accept" 应答，否则 client 会抛出 "Error during WebSocket handshake" 错误，并关闭连接

2.server 收到报文后，如果支持 WebSocket 协议，那么就会将自己的通信协议切换到 WebSocket，返回以下信息：

**"HTTP/1.1 101 WebSocket Protocol Handshake"：**返回的状态码为 101，表示同意 client 的协议转换请求

**"Upgrade: websocket"**

**"Connection: Upgrade"**

**"Sec-WebSocket-Accept: \*"**

**...**

### 2.1.3 spring整合webSocket

#### 2.1.3.1 添加jar包

<!-- spring websocket-->

<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-messaging</artifactId>  
 <version>${spring.version}</version>  
</dependency>  
<dependency>  
 <groupId>org.springframework</groupId>  
 <artifactId>spring-websocket</artifactId>  
 <version>${spring.version}</version>  
</dependency>

#### 2.1.3.2 添加webSocket相关配置

然后在spring配置文件中添加如下配置：

<!--webSocket配置-->

<websocket:handlers>

<!—webSocket客户端的连接路径是 网址+/mySocket ,访问该路径的webSocket请求将交由springWebSocketHandler 这个bean来处理-->  
 <websocket:mapping path="/mySocket" handler="springWebSocketHandler" />  
 <!—设置WebSocket拦截器,在握手之前 或 握手之后拦截webSocket请求，并对其进行处理-->

<websocket:handshake-interceptors>  
 <bean class="com.test.common.SpringWebSocketHandlerInterceptor" />  
 </websocket:handshake-interceptors>  
</websocket:handlers>

#### 2.1.3.3 编写拦截器类

接着编写拦截器类（对于下面两个方法，前一个方法将在握手前执行，第二个方法将在握手之后执行，握手结束后，才会去访问Handler类处理请求），其中参数**WebSocketHandler wsHandler**代表handler处理类，**Map<String, Object> map**则代表webSocketSession携带的参数，这里不会自动将session携带的参数拷贝到webSocketSeesion，需要我们手动拷贝，本例就是将**username**从session中拷贝到webSocketSeesion中

package com.test.common;

import java.util.Map;  
import javax.servlet.http.HttpServletRequest;  
import javax.servlet.http.HttpSession;  
import org.springframework.http.server.ServerHttpRequest;  
import org.springframework.http.server.ServerHttpResponse;  
import org.springframework.http.server.ServletServerHttpRequest;  
import org.springframework.web.socket.WebSocketHandler;  
import org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor;  
  
*/\*\*  
 \* WebSocket拦截器----握手之前将登陆用户信息从session设置到WebSocketSession  
 \*/*public class SpringWebSocketHandlerInterceptor extends HttpSessionHandshakeInterceptor {  
 @Override  
 public boolean beforeHandshake(ServerHttpRequest request, ServerHttpResponse response, WebSocketHandler wsHandler, Map<String, Object> map) throws Exception {  
 if (request instanceof ServletServerHttpRequest) {  
 HttpServletRequest servletRequest = ((ServletServerHttpRequest)request).getServletRequest();  
 String username=(String)servletRequest.getSession().getAttribute("username");  
 System.*out*.println("获取session里面的name-------------------"+username);  
 map.put("WEBSOCKET\_USERNAME", username);  
 servletRequest.getSession().setAttribute("WEBSOCKET\_USERNAME", username);  
 }  
 return true;  
 }  
  
 @Override  
 public void afterHandshake(ServerHttpRequest request, ServerHttpResponse response, WebSocketHandler wsHandler, Exception ex) {  
 super.afterHandshake(request, response, wsHandler, ex);  
 }  
}

#### 2.1.3.4 编写webSocket处理类(Handler)

*/\*\**

*\* 处理来自客户端webSocket请求、发送消息到客户端 的类  
 \*/*@Repository  
public class SpringWebSocketHandler extends TextWebSocketHandler {  
 private static final Map<String, WebSocketSession> *users* = new HashMap<String, WebSocketSession>();  
 private String userName = "";  
 private String message = "";  
 // 初次连接成功后执行  
 public void afterConnectionEstablished(WebSocketSession session) throws Exception {  
 userName = (String) session.getAttributes().get("WEBSOCKET\_USERNAME");  
 //以用户名为key，将webSocket存起来  
 *users*.put(userName, session);  
 System.*out*.println("用户" + userName + "连接成功");  
 }  
  
 // 接受消息处理消息  
 public void handleMessage(WebSocketSession webSocketSession, WebSocketMessage<?> webSocketMessage)  
 throws Exception {  
 message+=webSocketMessage.getPayload().toString();  
 this.sendMessageToUsers(new TextMessage(""));  
 }  
  
 public void handleTransportError(WebSocketSession session, Throwable throwable) throws Exception {  
 if (session.isOpen()) {  
 session.close();  
 }  
 System.*out*.println("链接出错，关闭链接......");  
 *users*.remove((String) session.getAttributes().get("WEBSOCKET\_USERNAME"));  
 }  
  
 public void afterConnectionClosed(WebSocketSession session, CloseStatus closeStatus) throws Exception {  
 System.*out*.println("链接关闭......");  
 *users*.remove((String) session.getAttributes().get("WEBSOCKET\_USERNAME"));  
 }  
  
 public boolean supportsPartialMessages() {  
 return false;  
 }  
 */\*\*  
 \* 给单独某个用户发送消息  
 \*/* public void setMessageToUser(String userName,String message) {  
 WebSocketSession user=*users*.get(userName);  
 if(user==null){  
 System.*out*.println("用户不存在");  
 }else {  
 try {  
 if (user.isOpen()) {  
 user.sendMessage(new TextMessage(message));  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 */\*\*  
 \* 给所有在线用户发送消息  
 \** ***@param*** *message  
 \*/* public void sendMessageToUsers(TextMessage message) {  
 for (Map.Entry<String,WebSocketSession> userEntry : *users*.entrySet()) {  
 WebSocketSession user=userEntry.getValue();  
 System.*out*.println("给"+user.getAttributes().get("WEBSOCKET\_USERNAME")+"发送消息");  
 try {  
 if (user.isOpen()) {  
 user.sendMessage(new TextMessage(this.message));  
 }  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

Demo中可以通过定时任务不断向客户端推送消息，客户端的js代码如下：

**var** ws;

//判断当前浏览器是否支持WebSocket  
**if** ('WebSocket' **in** window) {  
 ws = **new** WebSocket("ws://" + document.location.host + "/mySocket");  
} **else** {  
 alert('当前浏览器不支持websocket')  
}  
  
ws.onopen = **function** (evt) {  
 console.log("连接打开 ...");  
 ws.send("这是我发送的消息!");  
};  
  
ws.onmessage = **function** (evt) {  
 console.log("接受信息: " + evt.data);  
};  
  
ws.onclose = **function** (evt) {  
 console.log("连接断开.");  
};