目录

[一、为什么使用websocket 1](#_Toc522266258)

[二、客户端（前端）(使用H5的标准来使用websocket) 3](#_Toc522266259)

[1、客户端例子 4](#_Toc522266260)

[三、服务端（后端）简单例子 8](#_Toc522266261)

[1、引用依赖： 8](#_Toc522266262)

[2、启用WebSocket的支持 8](#_Toc522266263)

[3、理解websocket： 9](#_Toc522266264)

[4、WebSocketServer: 9](#_Toc522266265)

[5、Controller层 12](#_Toc522266266)

[运行结果： 13](#_Toc522266267)

[源码：websocket-demo 14](#_Toc522266268)

[四、群聊+显示昵称 14](#_Toc522266269)

[4.1原理 14](#_Toc522266270)

[4.2编码分析 14](#_Toc522266271)

[4.3群聊 功能实现 14](#_Toc522266272)

[4.3.1、引入依赖 14](#_Toc522266273)

[4.3.2、注入ServerEndpointExporter 14](#_Toc522266274)

[4.3.3、websocket的具体实现类 15](#_Toc522266275)

[4.3.4、编写客户端webSocketTest.html(放到static目录下) 17](#_Toc522266276)

[4.3.5测试 19](#_Toc522266277)

[4.4、群聊+昵称 20](#_Toc522266278)

[4.4.1、效果展示： 21](#_Toc522266279)

[4.4.2、对代码进行改造 22](#_Toc522266280)

[4.4.3、测试 25](#_Toc522266281)

[4.5、单聊 25](#_Toc522266282)

[4.5.1、创建消息对象socketMsg 25](#_Toc522266283)

[4.5.2、调整建立连接的方法（MyWebSocket） 26](#_Toc522266284)

[4.5.3、修改连接的方法onOpen（MyWebSocket）: 26](#_Toc522266285)

[4.5.4、修改消息发送的方法 onMessage（MyWebSocket）： 27](#_Toc522266286)

[4.5.5、客户端调整 28](#_Toc522266287)

[4.5.6、测试 29](#_Toc522266288)

# 一、为什么使用websocket

WebSocket的出现是基于Web应用的实时性需要而产生的。这种实时的Web应用大家应该不陌生，在生活中都应该用到过，比如新浪微博的评论、私信的通知，腾讯的WebQQ等。让我们来回顾下实时 Web 应用的窘境吧。

在WebSocket出现之前，一般通过两种方式来实现Web实时用：轮询机制和流技术；其中轮询有不同的轮询，还有一种叫Comet的长轮询。

轮询：这是最早的一种实现实时 Web 应用的方案。客户端以一定的时间间隔向服务端发出请求，以频繁请求的方式来保持客户端和服务器端的同步。这种同步方案的缺点是，当客户端以固定频率向服务 器发起请求的时候，服务器端的数据可能并没有更新，这样会带来很多无谓的网络传输，所以这是一种非常低效的实时方案。

长轮询：是对定时轮询的改进和提高，目地是为了降低无效的网络传输。当服务器端没有数据更新的时候，连接会保持一段时间周期直到数据或状态改变或者 时间过期，通过这种机制来减少无效的客户端和服务器间的交互。当然，如果服务端的数据变更非常频繁的话，这种机制和定时轮询比较起来没有本质上的性能的提 高。

流：常就是在客户端的页面使用一个隐藏的窗口向服务端发出一个长连接的请求。服务器端接到这个请求后作出回应并不断更新连接状态以保证客户端和服务 器端的连接不过期。通过这种机制可以将服务器端的信息源源不断地推向客户端。这种机制在用户体验上有一点问题，需要针对不同的浏览器设计不同的方案来改进 用户体验，同时这种机制在并发比较大的情况下，对服务器端的资源是一个极大的考验。

上述方式其实并不是真正的实时技术，只是使用了一种技巧来实现的模拟实时。在每次客户端和服务器端交互的时候都是一次 HTTP 的请求和应答的过程，而每一次的 HTTP 请求和应答都带有完整的 HTTP 头信息，这就增加了每次传输的数据量。但这些方式最痛苦的是开发人员，因为不论客户端还是服务器端的实现都很复杂，为了模拟比较真实的实时效果，开发人员 往往需要构造两个HTTP连接来模拟客户端和服务器之间的双向通讯，一个连接用来处理客户端到服务器端的数据传输，一个连接用来处理服务器端到客户端的数 据传输，这不可避免地增加了编程实现的复杂度，也增加了服务器端的负载，制约了应用系统的扩展性。

基于上述弊端，实现Web实时应用的技术出现了，WebSocket通过浏览器提供的API真正实现了具备像C/S架构下的桌面系统的实时通讯能 力。其原理是使用JavaScript调用浏览器的API发出一个WebSocket请求至服务器，经过一次握手，和服务器建立了TCP通讯，因为它本质 上是一个TCP连接，所以数据传输的稳定性强和数据传输量比较小。

WebSocket 协议

WebSocket 协议本质上是一个基于 TCP 的协议。为了建立一个 WebSocket 连接，客户端浏览器首先要向服务器发起一个 HTTP 请求，这个请求和通常的 HTTP 请求不同，包含了一些附加头信息，其中附加头信息”Upgrade: WebSocket”表明这是一个申请协议升级的 HTTP 请求，服务器端解析这些附加的头信息然后产生应答信息返回给客户端，客户端和服务器端的 WebSocket 连接就建立起来了，双方就可以通过这个连接通道自由的传递信息，并且这个连接会持续存在直到客户端或者服务器端的某一方主动的关闭连接。

# 二、客户端（前端）(使用H5的标准来使用websocket)

握手协议，客户端是由浏览器提供了API，所以只要使用JavaScript来简单调用即可

WebSocket JavaScript 接口定义:

|  |
| --- |
| [Constructor(in DOMString url, optional in DOMString protocol)]  interface WebSocket {  readonly attribute DOMString URL;  // ready state  const unsigned short CONNECTING = 0;  const unsigned short OPEN = 1;  const unsigned short CLOSED = 2;  readonly attribute unsigned short readyState;  readonly attribute unsigned long bufferedAmount;    // networking  attribute Function onopen;  attribute Function onmessage;  attribute Function onclose;  boolean send(in DOMString data);  void close();  };  WebSocket implements EventTarget; |

简单了解下上面接口方法和属性：

|  |
| --- |
| readyState表示连接有四种状态：  CONNECTING (0)：表示还没建立连接；  OPEN (1)： 已经建立连接，可以进行通讯；  CLOSING (2)：通过关闭握手，正在关闭连接；  CLOSED (3)：连接已经关闭或无法打开；  url是代表 WebSocket 服务器的网络地址，协议通常是”ws”或“wss(加密通信)”,send 方法就是发送数据到服务器端；  close 方法就是关闭连接；  onopen连接建立，即握手成功触发的事件；  onmessage收到服务器消息时触发的事件；  onerror异常触发的事件；  onclose关闭连接触发的事件； |

JavaScript调用浏览器接口实例如下：

html:

|  |
| --- |
| var wsServer = 'ws://localhost:8888/Demo'; //服务器地址  var websocket = new WebSocket(wsServer); //创建WebSocket对象  websocket.send("hello");//向服务器发送消息  alert(websocket.readyState);//查看websocket当前状态  websocket.onopen = function (evt) {  //已经建立连接  };  websocket.onclose = function (evt) {  //已经关闭连接  };  websocket.onmessage = function (evt) {  //收到服务器消息，使用evt.data提取  };  websocket.onerror = function (evt) {  //产生异常  }; |

## 1、客户端例子

我这里用2个页面来模拟2个用户

index.html :用户11

|  |
| --- |
| <!DOCTYPE HTML> <html> <head>  <title>My WebSocket</title> </head>  <body> Welcome<br/>  <input id="text" type="text" /> <button onclick="send()">Send</button> <button onclick="closeWebSocket()">Close</button>  <div id="message"> </div> </body>  <script type="text/javascript">  **var** websocket = **null**;   **var** sid=11;//用户id   //判断当前浏览器是否支持WebSocket  **if**('WebSocket' in window){  websocket = **new** WebSocket("ws://localhost:8084/websocket/"+sid);//创建WebSocket对象  //websocket = new WebSocket("${basePath}websocket/${cid}".replace("http","ws"));  }  **else**{  alert('Not support websocket')  }   //连接发生错误的回调方法  websocket.onerror = **function**(){  setMessageInnerHTML("error");  };   //连接成功建立的回调方法  websocket.onopen = **function**(event){  setMessageInnerHTML("open");  }   //接收到消息的回调方法  websocket.onmessage = **function**(event){  setMessageInnerHTML(event.data);  }   //连接关闭的回调方法  websocket.onclose = **function**(){  setMessageInnerHTML("close");  }   //监听窗口关闭事件，当窗口关闭时，主动去关闭websocket连接，防止连接还没断开就关闭窗口，server端会抛异常。  window.onbeforeunload = **function**(){  websocket.close();  }   //将消息显示在网页上  **function** setMessageInnerHTML(innerHTML){  document.getElementById('message').innerHTML += innerHTML + '<br/>';  }   //关闭连接  **function** closeWebSocket(){  websocket.close();  }   //发送消息  **function** send(){  **var** message = document.getElementById('text').value;  websocket.send(message);  }  </script> </html> |

index2.html ：用户2

|  |
| --- |
| <!DOCTYPE HTML> <html> <head>  <title>My WebSocket</title> </head>  <body> Welcome<br/>  <input id="text" type="text" /> <button onclick="send()">Send</button> <button onclick="closeWebSocket()">Close</button>  <div id="message"> </div> </body>  <script type="text/javascript">  **var** websocket = **null**;   **var** sid=22;//用户id   //判断当前浏览器是否支持WebSocket  **if**('WebSocket' in window){  websocket = **new** WebSocket("ws://localhost:8084/websocket/"+sid);//创建WebSocket对象  //websocket = new WebSocket("${basePath}websocket/${cid}".replace("http","ws"));  }  **else**{  alert('Not support websocket')  }   //连接发生错误的回调方法  websocket.onerror = **function**(){  setMessageInnerHTML("error");  };   //连接成功建立的回调方法  websocket.onopen = **function**(event){  setMessageInnerHTML("open");  }   //接收到消息的回调方法  websocket.onmessage = **function**(event){  setMessageInnerHTML(event.data);  }   //连接关闭的回调方法  websocket.onclose = **function**(){  setMessageInnerHTML("close");  }   //监听窗口关闭事件，当窗口关闭时，主动去关闭websocket连接，防止连接还没断开就关闭窗口，server端会抛异常。  window.onbeforeunload = **function**(){  websocket.close();  }   //将消息显示在网页上  **function** setMessageInnerHTML(innerHTML){  document.getElementById('message').innerHTML += innerHTML + '<br/>';  }   //关闭连接  **function** closeWebSocket(){  websocket.close();  }   //发送消息  **function** send(){  **var** message = document.getElementById('text').value;  websocket.send(message);  }  </script> </html> |

# 三、服务端（后端）简单例子

spring boot+websocket

## 1、引用依赖：

<!--webSocket-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-websocket</artifactId>  
</dependency>

## 2、启用WebSocket的支持

WebSocketConfig：

|  |
| --- |
| @Configuration @EnableWebSocket public class WebSocketConfig implements WebSocketConfigurer {   @Bean  public ServerEndpointExporter serverEndpointExporter(ApplicationContext context) {  return new ServerEndpointExporter();  }  } |

## 3、理解websocket：

WebSocket 是什么原理？为什么可以实现持久连接？ - 知乎

<https://www.zhihu.com/question/20215561>

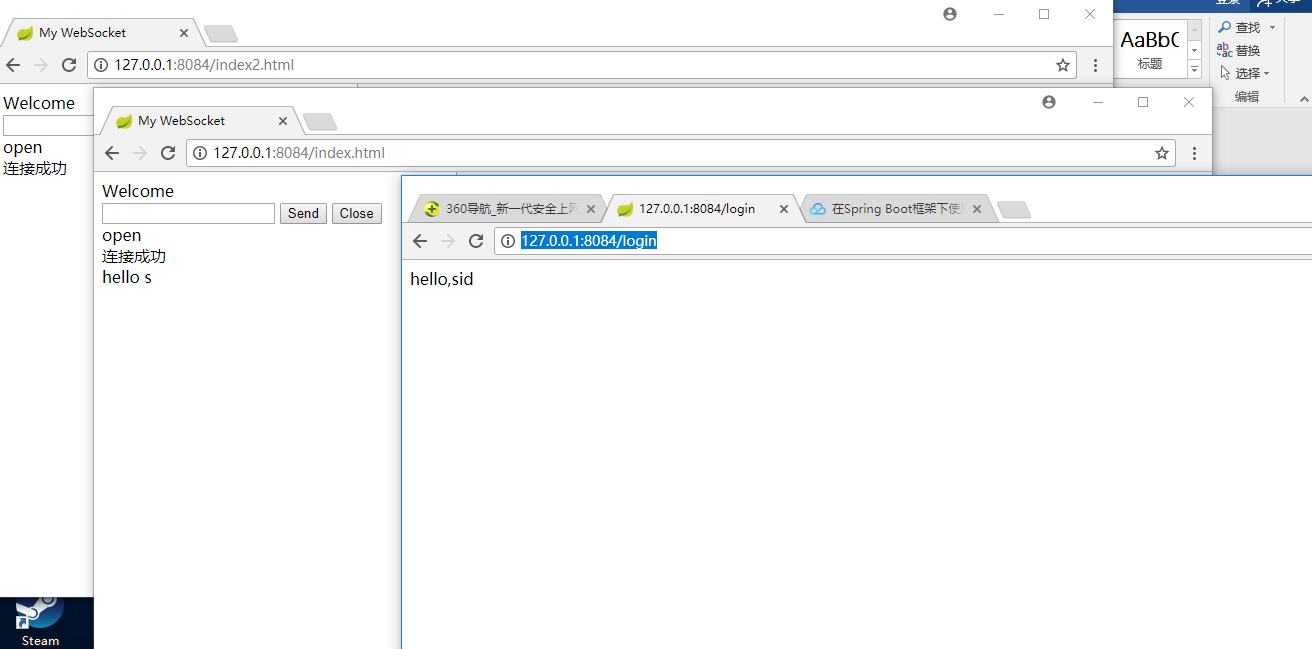
## 4、WebSocketServer:

|  |
| --- |
| package websocket.demo; import org.springframework.stereotype.Component; import javax.websocket.\*; import javax.websocket.server.PathParam; import javax.websocket.server.ServerEndpoint; import java.io.IOException; import java.util.HashMap; import java.util.Map; import java.util.concurrent.CopyOnWriteArraySet;  @ServerEndpoint(value = "/websocket/{sid}") @Component public class WebSocketServer {  //静态变量，用来记录当前在线连接数。应该把它设计成线程安全的。  private static int *onlineCount* = 0;  //concurrent包的线程安全Set，用来存放每个客户端对应的MyWebSocket对象。  private static CopyOnWriteArraySet<WebSocketServer> *webSocketSet* = new CopyOnWriteArraySet<WebSocketServer>();  //通过id 来获取相应的Session（因为Session没有的话，websocket是用的）  private static Map<String,Session> *map*= new HashMap<>();  //与某个客户端的连接会话，需要通过它来给客户端发送数据  private Session session;   //接收sid  private String sid="";   */\*\*  \* 连接建立成功调用的方法\*/* @OnOpen  public void onOpen(Session session,@PathParam("sid") String sid) {  this.session = session;  *webSocketSet*.add(this); //加入set中  *addOnlineCount*(); //在线数加1  *map*.put( sid,session );  System.*out*.println("有新连接加入"+sid+"!,当前在线人数为" + *getOnlineCount*());  try {  sendMessage("连接成功",sid);  } catch (IOException e) {  System.*out*.println("websocket IO异常");  }  }   */\*\*  \* 连接关闭调用的方法  \*/* @OnClose  public void onClose() {  *webSocketSet*.remove(this); //从set中删除  *subOnlineCount*(); //在线数减1  System.*out*.println("有一连接关闭！当前在线人数为" + *getOnlineCount*());  }   */\*\*  \* 收到客户端消息后调用的方法  \*  \** ***@param*** *message 客户端发送过来的消息\*/* @OnMessage  public void onMessage(String message) {   System.*out*.println("来自客户端"+sid+"的消息:" + message);   //群发消息  for (WebSocketServer item : *webSocketSet*) {  try {  item.sendMessage(message,sid);  } catch (IOException e) {  e.printStackTrace();  }  }  }   */\*\*  \* 发生异常时调用  \** ***@param*** *session  \** ***@param*** *error  \*/* @OnError  public void onError(Session session, Throwable error) {  System.*out*.println("发生错误");  error.printStackTrace();  }   */\*\*  \* 用于主动推送信息  \** ***@param*** *message  \** ***@throws*** *IOException  \*/* public void sendMessage(String message,@PathParam("sid") String sid) throws IOException {  Session session = *map*.get( sid );  session.getBasicRemote().sendText(message);  }    */\*\*  \* 群发自定义消息  \* \*/* public static void sendInfo(String message,@PathParam("sid") String sid) throws IOException {  System.*out*.println("推送消息到窗口"+sid+"，推送内容:"+message);  for (WebSocketServer item : *webSocketSet*) {  try {  //item.sendMessage(message);  //这里可以设定只推送给这个sid的，为null则全部推送  if(sid==null) {  item.sendMessage(message,sid);  }else if(item.sid.equals(sid)){  item.sendMessage(message,sid);  }  } catch (IOException e) {  continue;  }  }  }   public static synchronized int getOnlineCount() {  return *onlineCount*;  }   public static synchronized void addOnlineCount() {  WebSocketServer.*onlineCount*++;  }   public static synchronized void subOnlineCount() {  WebSocketServer.*onlineCount*--;  } } |
|  |

## 5、Controller层

|  |
| --- |
| @GetMapping("/login") public String login() throws IOException {  String s = "hello s";  String sid = "11";  webSocketServer.sendMessage(s,sid);  return "hello,sid"; } |

# 运行结果：



测试完成

代码参考文档：

<https://my.oschina.net/pierrecai/blog/903158>

<https://blog.csdn.net/moshowgame/article/details/80275084>

<https://blog.csdn.net/u013507760/article/details/53259192>

<https://www.jianshu.com/p/d79bf8174196>

<https://www.jianshu.com/p/0f498adb3820>

@Component

@component （把普通pojo实例化到spring容器中，相当于配置文件中的   
<bean id="" class=""/>）

* 泛指各种组件，就是说当我们的类不属于各种归类的时候（不属于@Controller、@Services等的时候），我们就可以使用@Component来标注这个类。

# 源码：websocket-demo

# 四、群聊+显示昵称

## 4.1原理

看完让你彻底搞懂Websocket原理 - 扶强 - 博客园

<http://www.cnblogs.com/fuqiang88/p/5956363.html>

<http://412887952-qq-com.iteye.com/category/356333?page=2>

## 4.2编码分析

<http://412887952-qq-com.iteye.com/blog/2404269>

## 4.3群聊 功能实现

### 4.3.1、引入依赖

|  |
| --- |
| <!--websocket依赖--> <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-websocket</artifactId> </dependency> |

### 4.3.2、注入ServerEndpointExporter

|  |
| --- |
| */\*\*  \** ***@Auther:*** *zj  \** ***@Date:*** *2018/8/16 17:21  \** ***@Description:*** *编写一个WebSocketConfig配置类，注入对象ServerEndpointExporter，  \* 这个bean会自动注册使用了@ServerEndpoint注解声明的Websocket endpoint  \*/* @Configuration public class WebSocketConfig {  @Bean  public ServerEndpointExporter serverEndpointExporter() {  return new ServerEndpointExporter();  } } |

### 4.3.3、websocket的具体实现类

|  |
| --- |
| */\*\*  \** ***@Auther:*** *zj  \** ***@Date:*** *2018/8/16 17:55  \** ***@Description:*** *websocket的具体实现类  \* 使用springboot的唯一区别是要@Component声明下，而使用独立容器是由容器自己管理websocket的，  \* 但在springboot中连容器都是spring管理的。  虽然@Component默认是单例模式的，但springboot还是会为每个websocket连接初始化一个bean，  所以可以用一个静态set保存起来。  \*/* @ServerEndpoint(value = "/websocket") @Component public class MyWebSocket {   //用来存放每个客户端对应的MyWebSocket对象。  private static CopyOnWriteArraySet<MyWebSocket> *webSocketSet* = new CopyOnWriteArraySet<MyWebSocket>();   //与某个客户端的连接会话，需要通过它来给客户端发送数据  private Session session;   */\*\*  \* 连接建立成功调用的方法  \*/* @OnOpen  public void onOpen(Session session) {  this.session = session;   *webSocketSet*.add(this); //加入set中  System.*out*.println("有新连接加入！当前在线人数为" + *webSocketSet*.size());  this.session.getAsyncRemote().sendText("恭喜您成功连接上WebSocket-->当前在线人数为："+*webSocketSet*.size());  }   */\*\*  \* 连接关闭调用的方法  \*/* @OnClose  public void onClose() {  *webSocketSet*.remove(this); //从set中删除  System.*out*.println("有一连接关闭！当前在线人数为" + *webSocketSet*.size());  }   */\*\*  \* 收到客户端消息后调用的方法  \*  \** ***@param*** *message 客户端发送过来的消息\*/* @OnMessage  public void onMessage(String message, Session session) {  System.*out*.println("来自客户端的消息:" + message);   //群发消息  broadcast(message);  }   */\*\*  \* 发生错误时调用  \*  \*/* @OnError  public void onError(Session session, Throwable error) {  System.*out*.println("发生错误");  error.printStackTrace();  }   */\*\*  \* 群发自定义消息  \* \*/* public void broadcast(String message){  for (MyWebSocket item : *webSocketSet*) {  //同步异步说明参考：http://blog.csdn.net/who\_is\_xiaoming/article/details/53287691  //this.session.getBasicRemote().sendText(message);  item.session.getAsyncRemote().sendText(message);//异步发送消息.  }  } } |

### 4.3.4、编写客户端webSocketTest.html(放到static目录下)

|  |
| --- |
| <!DOCTYPE HTML> <html> <head>  <meta charset="UTF-8">  <title>My WebSocket</title>   <style>  #message{  margin-top:40px;  border:1px solid gray;  padding:20px;  }  </style>  </head>  <body>   <button onclick="conectWebSocket()">连接WebSocket</button> <button onclick="closeWebSocket()">断开连接</button> <hr />   <br /> 消息：<input id="text" type="text" /> <button onclick="send()">发送消息</button> <div id="message"></div> </body>  <script type="text/javascript">  **var** websocket = **null**;   **function** conectWebSocket(){   //判断当前浏览器是否支持WebSocket  **if** ('WebSocket'in window) {  websocket = **new** WebSocket("ws://localhost:8080/websocket");  } **else** {  alert('Not support websocket')  }    //连接发生错误的回调方法  websocket.onerror = **function**() {  setMessageInnerHTML("error");  };   //连接成功建立的回调方法  websocket.onopen = **function**(event) {  setMessageInnerHTML("Loc MSG: 成功建立连接");  }   //接收到消息的回调方法  websocket.onmessage = **function**(event) {  setMessageInnerHTML(event.data);  }   //连接关闭的回调方法  websocket.onclose = **function**() {  setMessageInnerHTML("Loc MSG:关闭连接");  }   //监听窗口关闭事件，当窗口关闭时，主动去关闭websocket连接，防止连接还没断开就关闭窗口，server端会抛异常。  window.onbeforeunload = **function**() {  websocket.close();  }  }      //将消息显示在网页上  **function** setMessageInnerHTML(innerHTML) {  document.getElementById('message').innerHTML += innerHTML + '<br/>';  }   //关闭连接  **function** closeWebSocket() {  websocket.close();  }   //发送消息  **function** send() {  **var** message = document.getElementById('text').value;  websocket.send(message);  } </script> </html> |

### 4.3.5测试

打开chrome浏览器：

localhost:8080/webSocketTest.html

2）点击【连接WebSocket】,然后就可以发送消息了。



（3）打开另外一个浏览器或者直接打开一个TAB访问地址<http://127.0.0.1:8080/>webSocketTest.html

（4）点击【连接WebSocket】,然后就可以发送消息了。

（5）观察两边的信息打印，看是否可以接收到消息。

参考：<http://412887952-qq-com.iteye.com/blog/2404851>

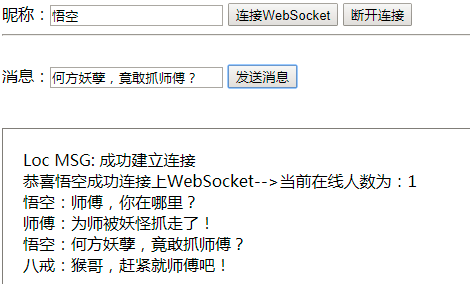
## 4.4、群聊+昵称

上一部分已经可以群聊功能了，但是存在一个问题，就是无法显示消息是谁发送了，这样就会导致消息乱七八糟的无法解读，我们来解决这个问题

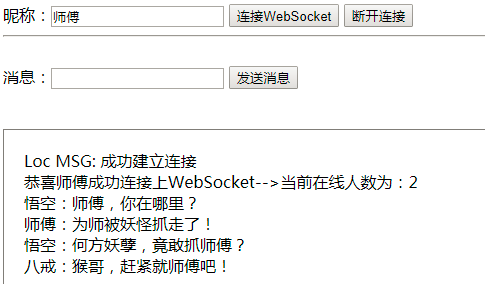
### 4.4.1、效果展示：

**效果展示：**

◊悟空消息：

◊师傅消息：



◊八戒消息：



### 4.4.2、对代码进行改造

1、在连接的时候，让用户输入昵称

|  |
| --- |
| input type="text" id="nickname"/> <button onclick="conectWebSocket()">连接WebSocket</button> |

2、连接前的校验

|  |
| --- |
| **var** nickname = document.getElementById("nickname").value; **if**(nickname===""){  alert("请输入昵称");  **return**; } |

3、昵称显示方法

首先修改客户端的连接方式：

**Js代码  [收藏代码](javascript:void())**

1. websocket = **new** WebSocket("ws://localhost:8080/websocket/"+nickname);

**Java代码  [收藏代码](javascript:void())**



◊服务端的服务端点发布：

**Java代码  [收藏代码](javascript:void())**

1. @ServerEndpoint(value = "/websocket/{nickname}")
2. @Component
3. **public** **class** MyWebSocket {}

◊服务端获取参数：

       这个在每个方法中都可以获取的，使用@PathParam("nickname") String nickname，比如在连接的时候：

修改MyWebSocket:

添加：

|  |
| --- |
| private String nickname; |

|  |
| --- |
| */\*\*  \* 连接建立成功调用的方法  \*/* @OnOpen public void onOpen(Session session,@PathParam("nickname") String nickname) {  this.session = session;  this.nickname=nickname;   *webSocketSet*.add(this); //加入set中  System.*out*.println("有新连接加入:"+nickname+",当前在线人数为" + *webSocketSet*.size());  this.session.getAsyncRemote().sendText("恭喜您成功连接上WebSocket-->当前在线人数为："+*webSocketSet*.size()); } |

|  |
| --- |
| */\*\*  \* 收到客户端消息后调用的方法  \*  \** ***@param*** *message 客户端发送过来的消息\*/* @OnMessage public void onMessage(String message, Session session,@PathParam("nickname") String nickname) {  System.*out*.println("来自客户端的消息-->"+nickname+": " + message);   //群发消息  broadcast(nickname+": "+message); } |

### 4.4.3、测试



如效果图一样

参考：<http://412887952-qq-com.iteye.com/blog/2404898>

## 4.5、单聊

### 4.5.1、创建消息对象socketMsg

|  |
| --- |
| package jit.hf.agriculture.vo;  */\*\*  \** ***@Auther:*** *zj  \** ***@Date:*** *2018/8/16 23:29  \** ***@Description:这里我们就不能使用简单的文本消息进行消息的发送了，我们使用json进行消息的发送。*** *\* 所以需要先创建一个消息对象，里面包含了消息发送者，消息接受者，消息类型（单聊还是群聊），还是就是消息，如下：  \*/* public class SocketMsg {  private int type; //聊天类型0：群聊，1：单聊.  private String fromUser;//发送者.  private String toUser;//接受者.  private String msg;//消息  public int getType() {  return type;  }  public void setType(int type) {  this.type = type;  }  public String getFromUser() {  return fromUser;  }  public void setFromUser(String fromUser) {  this.fromUser = fromUser;  }  public String getToUser() {  return toUser;  }  public void setToUser(String toUser) {  this.toUser = toUser;  }  public String getMsg() {  return msg;  }  public void setMsg(String msg) {  this.msg = msg;  } } |

### 4.5.2、调整建立连接的方法（MyWebSocket）

这里主要是要使用一个map对象保存频道号和session之前的关系，之后就可以通过频道号获取session，然后使用session进行消息的发送。

|  |
| --- |
| //用来记录sessionId和该session进行绑定 private static Map<String,Session> *map* = new HashMap<String, Session>(); |

### 4.5.3、修改连接的方法onOpen（MyWebSocket）:

在建立连接的时候，就保存频道号（这里使用的是session.getId()作为频道号）和session之间的对应关系：

|  |
| --- |
| */\*\*  \* 连接建立成功调用的方法  \*/* @OnOpen public void onOpen(Session session,@PathParam("nickname") String nickname) {  this.session = session;  this.nickname=nickname;   *map*.put(session.getId(), session);   *webSocketSet*.add(this); //加入set中  System.*out*.println("有新连接加入:"+nickname+",当前在线人数为" + *webSocketSet*.size());  this.session.getAsyncRemote().sendText("恭喜"+nickname+"成功连接上WebSocket(其频道号："+session.getId()+")-->当前在线人数为："+*webSocketSet*.size()); } |

### 4.5.4、修改消息发送的方法 onMessage（MyWebSocket）：

|  |
| --- |
| */\*\*  \* 收到客户端消息后调用的方法  \*  \** ***@param*** *message 客户端发送过来的消息\*/* @OnMessage  public void onMessage(String message, Session session,@PathParam("nickname") String nickname) {  System.*out*.println("来自客户端的消息-->"+nickname+": " + message);   //从客户端传过来的数据是json数据，所以这里使用jackson进行转换为SocketMsg对象，  // 然后通过socketMsg的type进行判断是单聊还是群聊，进行相应的处理:  ObjectMapper objectMapper = new ObjectMapper();  SocketMsg socketMsg;   try {  socketMsg = objectMapper.readValue(message, SocketMsg.class);  if(socketMsg.getType() == 1){  //单聊.需要找到发送者和接受者.   socketMsg.setFromUser(session.getId());//发送者.  Session fromSession = *map*.get(socketMsg.getFromUser());  Session toSession = *map*.get(socketMsg.getToUser());  //发送给接受者.  if(toSession != null){  //发送给发送者.  fromSession.getAsyncRemote().sendText(nickname+"："+socketMsg.getMsg());  toSession.getAsyncRemote().sendText(nickname+"："+socketMsg.getMsg());  }else{  //发送给发送者.  fromSession.getAsyncRemote().sendText("系统消息：对方不在线或者您输入的频道号不对");  }  }else{  //群发消息  broadcast(nickname+": "+socketMsg.getMsg);  }   } catch (JsonParseException e) {  e.printStackTrace();  } catch (JsonMappingException e) {  e.printStackTrace();  } catch (IOException e) {  e.printStackTrace();  }   // //群发消息 // broadcast(nickname+": "+message);  } |

### 4.5.5、客户端调整

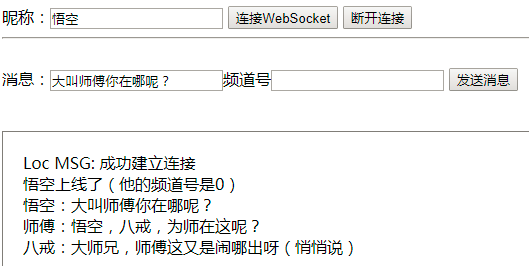
|  |
| --- |
| 消息：<input id="text" type="text" /> 频道号：<input id="toUser" type="text" /> <button onclick="send()">发送消息</button> |

|  |
| --- |
| //发送消息 **function** send() {  //获取输入的文本信息进行发送  **var** message = document.getElementById('text').value;  **var** toUser = document.getElementById('toUser').value;  **var** socketMsg = {msg:message,toUser:toUser};  **if**(toUser == ''){  //群聊.  socketMsg.type = 0;  }**else**{  //单聊.  socketMsg.type = 1;  }   websocket.send(JSON.stringify(socketMsg)); } |

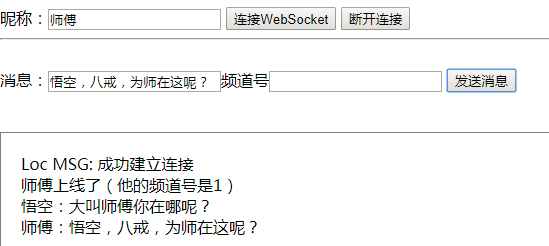
### 4.5.6、测试

与预期一样

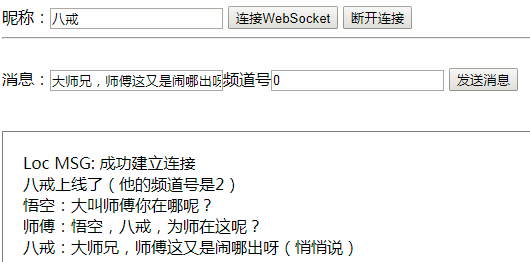
**◊悟空消息：**



**◊师傅消息：**



**◊八戒消息：**



参考：

<http://412887952-qq-com.iteye.com/blog/2405514>