WebSocket实现聊天室

# 课程介绍

本课程中，将使用Java技术快速实现一个简单的在线聊天室系统，该系统具备很强的扩展性，可以根据业务需要，制作出聊吧、在线客服系统、Web版的微信QQ即使通信系统等。使用的都是比较流行的技术，采用积木式的编程思路，不管你原来的技术基础如何，都可以快速上手。 当然，由于时间关系，一些技术只能是抛砖引玉，若要更加灵活的运用，还是需要深入学习，欢迎大家交流。

Web领域的实时推送技术，也被称作Realtime技术。这种技术要达到的目的是让用户不需要刷新浏览器就可以获得实时更新。它有着广泛的应用场景，比如在线聊天室、在线客服系统、评论系统、WebIM等。

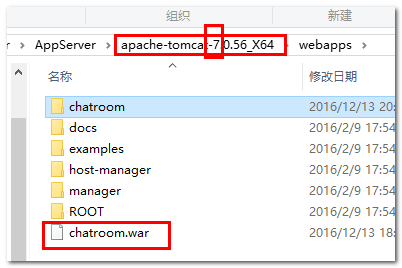
有一定Java技术基础的同学。

没有基础的同学？

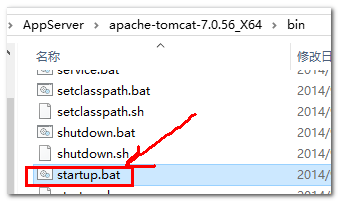
搭积木一样，也可以弄一个聊天室出来。

# 项目运行演示

将war部署到tomcat中（将复制到）



启动tomcat即可：



# 核心技术介绍

## WebSocket协议概述

WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信(full-duplex)。一开始的握手需要借助HTTP请求完成。

WebSocket是真正实现了全双工通信的服务器向客户端推的互联网技术。

它是一种在单个TCP连接上进行全双工通讯协议。Websocket通信协议与2011年倍IETF定为标准RFC 6455，Websocket API被W3C定为标准。

全双工和单工的区别（了解）：

|  |
| --- |
| * 全双工（Full Duplex）是通讯传输的一个术语。通信允许数据在两个方向上同时传输，它在能力上相当于两个单工通信方式的结合。全双工指可以同时（瞬时）进行信号的双向传输（A→B且B→A）。指A→B的同时B→A，是瞬时同步的。 * 单工、半双工（Half Duplex），所谓半双工就是指一个时间段内只有一个动作发生，举个简单例子，一条窄窄的马路，同时只能有一辆车通过，当目前有两辆车对开，这种情况下就只能一辆先过，等到头儿后另一辆再开，这个例子就形象的说明了半双工的原理。早期的对讲机、以及早期集线器等设备都是基于半双工的产品。随着技术的不断进步，半双工会逐渐退出历史舞台。 |

推的技术和拉的技术（了解）：

|  |
| --- |
| 推送（PUSH）技术是一种建立在客户服务器上的机制，就是由服务器主动将信息发往客户端的技术。就像是广播电台播音。  同传统的拉（PULL）技术相比，最主要的区别在于推送（PUSH）技术是由服务器主动向客户机发送信息，而拉（PULL）技术则是由客户机主动请求信息。PUSH技术的优势在于信息的主动性和及时性。  简单的说，相对于服务端：拉的技术是被动向客户端提供数据，推的技术是主动向客户端提供数据。 |

互联网技术（了解）：

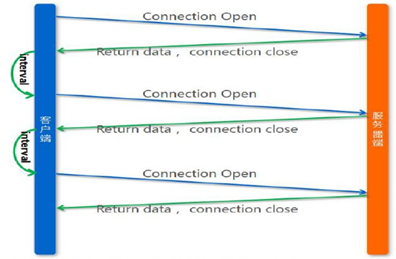
|  |
| --- |
| 互联网技术定义：互联网技术指在计算机技术的基础上开发建立的一种信息技术（直译：Internet Technology；简称：IT）。  该技术把互联网上分散的资源融为有机整体，实现资源的全面共享和有机协作，使人们能够透明地使用资源的整体能力并按需获取信息。 |

## WebSocket协议的优越性

以前不管使用HTTP轮询或使用TCP长连接等方式制作在线聊天系统，都有天然缺陷，随着Html5的兴起，其中有一个新的协议WebSocket protocol，可实现浏览器与服务器全双工通信(full-duplex)，它可以做到：浏览器和服务器只需要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道。两者之间就直接可以数据互相传送。这个新的协议的特点正好适合这种在线即时通信。

传统的Http协议实现方式：

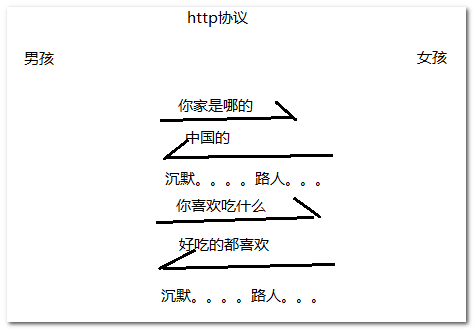
http协议可以多次请求，因为每次请求之后，都会关闭链接，下次重新请求数据，需要再次打开链接。



说明：

1. 基于polling（轮询）技术：以频繁请求方式来保持客户端和服务端的同步
2. 问题：客户端的频繁的请求，服务端的数据无变化，造成通信低效

图解：



传统socket技术：

长连接

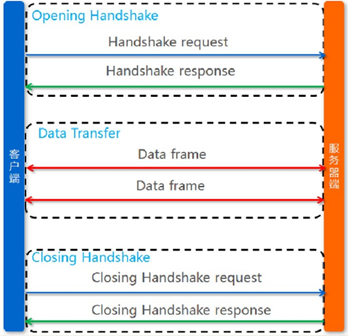
客户端 --（先连接上去）----- 服务端

好处：可以实现客户端和服务端双向通信

缺点：如果大家都不说话，是不是资源就浪费了

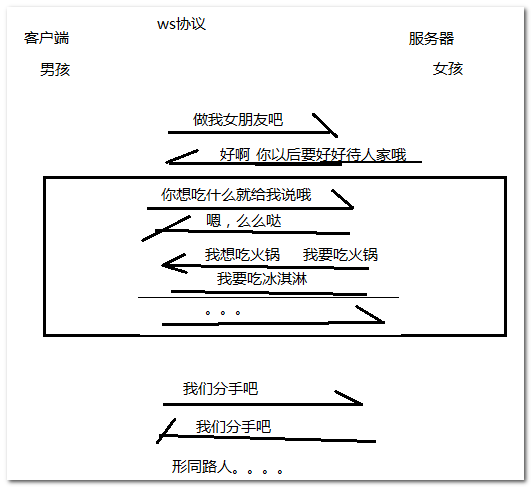
WebSocket协议实现方式：

它是一种长链接，只能通过一次请求来初始化链接，然后所有的请求和响应都是通过这个TCP链接进行通讯，这意味着它是一种基于事件驱动，异步的消息机制



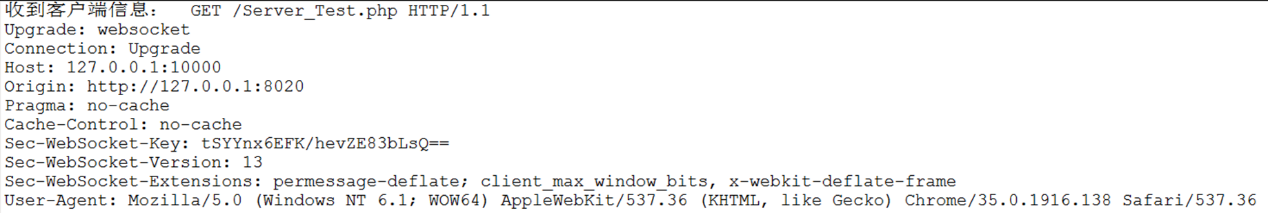
说明：原理和TCP一样，只需做一个握手动作，就可以形成一条快速通道。

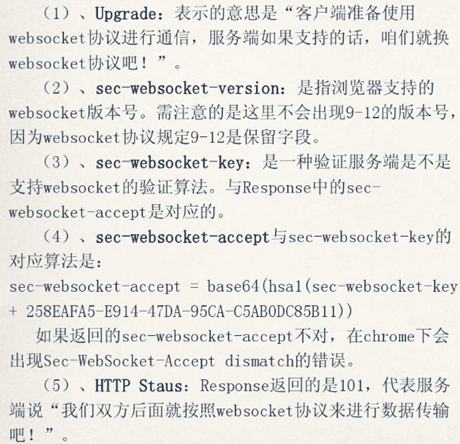
图解：



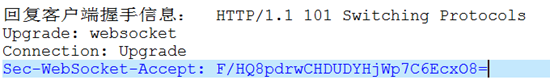
详细的通信过程：（学完再看）

1）客户端发起http请求，附加头信息为：“Upgrade Websocket”





2）服务端解析，并返回握手信息，从而建立连接



3）传输数据（双向）

https://pic3.zhimg.com/v2-743a9e1ad8af5895fa5dd7ff7a97a60a_b.png

4）客户端或服务端主动断开连接。客户端主动断开：客户端发起http请求，请求断开连接，服务端收到消息后断开WebSocket连接；服务端主动断开：直接断开WebSocket连接，客户端的API会立刻得知。

websocket的优越性不言自明，长连接的连接资源（线程资源）随着连接数量的增多，必会耗尽，客户端轮询会给服

务器造成很大的压力，而websocket是在物理层非网络层建立一条客户端至服务器的长连接，以此来保证服务器向客

户端的即时推送，既不耗费线程资源，又不会不断向服务器轮询请求。

webscoket和传统http协议的区别

传统的http请求的的rest风格现在很流行，它的作用就是通过rest风格，能够把不同路径的请求映射到同一个方法进行处理。http协议可以多次请求，因为每次请求之后，都会关闭链接，下次重新请求数据，需要再次打开链接。而对于webscoket来说，它是一种长链接，只能通过一次请求来初始化链接，然后所有的请求和响应都是通过这个TCP链接进行通讯，这意味着它是一种基于事件驱动，异步的消息机制，和JMS, AMQP等消息机制的应用差不多。

以前不管使用HTTP轮询或使用TCP长连接等方式制作在线聊天系统，都有天然缺陷，随着Html5的兴起，其中有一个新的协议WebSocket protocol，可实现浏览器与服务器全双工通信(full-duplex)，它可以做到：浏览器和服务器只需要做一个握手的动作，然后，浏览器和服务器之间就形成了一条快速通道。两者之间就直接可以数据互相传送。这个新的协议的特点正好适合这种在线即时通信。

传统的客户端和服务端通信方式：

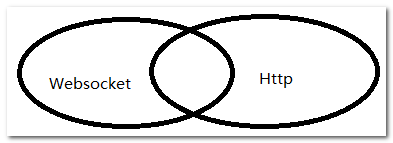
WebSocket的客户端和服务端通信方式：



说明：

http协议是一种应用层协议，已经定义了请求的格式，例如请求的头部的关键字，还有也定义了，服务器响应数据的格式，它对请求和响应的数据格式做了规范，而websocket协议不同，websocket协议还不够详细，它没有规定请求和接收的数据格式，例如，浏览器想向服务器请求进行socket通讯，但服务器不知道是否要进行socketon通讯，由于这个原因，websocket就定义了一个子协议，也就是浏览器客户端和服务器在请求握手的时候，他们能根据头部的Sec-WebSocket-Protocol，决定是否要进行websocket通讯。当然子协议的使用不是必须的，但是如果不使用子协议，那就必须自己定义一种请求和接收的数据格式规范，然后客户端和服务器都使用这种规范来进行通讯。

WebSocket协议和Http协议的关系：



## 客户端-浏览器的支持

WebSocket通信的客户端使用的是浏览器，客户端操作的API是HTML5中新增的API，使用这些API可以让客户端（浏览器）和服务端（服务器）进行全双工的通讯。

支持的浏览器如下：

|  |  |
| --- | --- |
| 浏览器类型 | 浏览器版本 |
| Chrome | Supported in version 4+ |
| Firefox | Supported in version 4+ |
| Internet Explorer | Supported in version 10+ |
| Opera | Supported in version 10+ |
| Safari | Supported in version 5+ |

问题出现了，Html5 websocket兼容性还不是很好，不是所有的浏览器都支持这些新的API，特别是在IE10以下。

但幸运的是现在绝大多数主流的浏览器都支持这些API，即使不支持的哪些旧的浏览器，也有解决方案。如：

为了处理不同浏览器和浏览器版本的兼容性，spring webscoket基于SockJS protocol提供了一种解决兼容性的方法，在底层屏蔽兼容性的问题，提供统一的，透明的，可理解性的webscoket解决方案。

SockJS 是一个浏览器上运行的 JavaScript 库，如果浏览器不支持 WebSocket，该库可以模拟对 WebSocket 的支持，实现浏览器和 Web 服务器之间低延迟、全双工、跨域的通讯通道。

## 服务端-服务器的支持

本课程是基于Java语言开发的，因此服务器只讨论JEE服务器。

新版本的应用服务器新增了支持的API，如Tomcat 7.0.47+等

# 开发环境搭建-基础项目导入

## 整体框架介绍

服务端：Maven+spring mvc+Spring WebSocket+jQuery+Gson

客户端：html5的WebSocket的api

图解：

## Maven环境的搭建（略）

maven的项目之前一定先配置好maven的环境！（课后自己去弄弄）

引入pom.xml配置：

|  |
| --- |
| <project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>  <modelVersion>4.0.0</modelVersion>  <groupId>cn.itcast.projects</groupId>  <artifactId>chatroomdemo</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>war</packaging>  <name>chatroomdemo</name>  <description>聊天室的demo</description>  <!-- 自定义属性管理 -->  <properties>  <!-- 编译等所有操作使用utf-8编码 -->  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <!-- 统一版本维护管理 -->  <spring.version>4.2.8.RELEASE</spring.version>  <servlet.version>3.1.0</servlet.version>  <jsp.version>2.0</jsp.version>  <gson.version>2.7</gson.version>  <junit.version>4.12</junit.version>  </properties>  <!-- 依赖管理 -->  <dependencies>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-webmvc</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-websocket</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>org.springframework</groupId>  <artifactId>spring-messaging</artifactId>  <version>${spring.version}</version>  </dependency>  <dependency>  <groupId>javax.servlet</groupId>  <artifactId>javax.servlet-api</artifactId>  <version>${servlet.version}</version>  <scope>provided</scope>  </dependency>  <dependency>  <groupId>junit</groupId>  <artifactId>junit</artifactId>  <version>${junit.version}</version>  <scope>test</scope>  </dependency>  <dependency>  <groupId>com.google.code.gson</groupId>  <artifactId>gson</artifactId>  <version>${gson.version}</version>  </dependency>  </dependencies>  <!-- 构建信息管理 -->  <build>  <finalName>chatroom</finalName>  <plugins>  <!-- 编译的jdk版本 -->  <plugin>  <groupId>org.apache.maven.plugins</groupId>  <artifactId>maven-compiler-plugin</artifactId>  <configuration>  <source>1.7</source>  <target>1.7</target>  </configuration>  </plugin>  <plugin>  <groupId>org.apache.tomcat.maven</groupId>  <artifactId>tomcat7-maven-plugin</artifactId>  <version>2.2</version>  <configuration>  <port>8080</port>  <path>/chatroom</path>  <uriEncoding>UTF-8</uriEncoding>  <finalName>chatroom</finalName>  <server>tomcat7</server>  </configuration>  </plugin>  </plugins>  </build>  </project> |

## 基础项目导入

# 项目API讲解

## 客户端的API

客户端如何去连接服务端？

客户端需要主动握手，

需要使用html5的一些代码

|  |
| --- |
| // 创建一个Socket实例（需要浏览器支持）ws：WebSocket协议地址开头  var socket = new WebSocket('ws://localhost:8080');  //下面有几个回调函数，自动调用（什么时候调用？）  // 打开Socket  socket.onopen = function(event) {  //握手成功后，会自动调用该函数  }  // 监听消息：用来获取服务端的消息  socket.onmessage = function(event) {  console.log('Client received a message',event);  };  // 监听Socket的关闭  socket.onclose = function(event) {  console.log('Client notified socket has closed',event);  };  // 关闭Socket....  //socket.close()  }; |

## 服务端的API

spring WebSocket：jee：WebSocket的封装。

用：只需要知道搭建步骤即可。

|  |
| --- |
| /\*\*  \*  \* 说明：WebScoket配置处理器  \* 把处理器和拦截器注册到spring websocket中  \* **@author** 传智.BoBo老师  \* **@version** 1.0  \* **@date** 2016年10月27日  \*/  @Component("webSocketConfig")  //配置开启WebSocket服务用来接收ws请求  @EnableWebSocket  **public** **class** WebSocketConfig **implements** WebSocketConfigurer {  //注入处理器  @Autowired  **private** ChatWebSocketHandler webSocketHandler;  @Autowired  **private** ChatHandshakeInterceptor chatHandshakeInterceptor;  **public** **void** registerWebSocketHandlers(WebSocketHandlerRegistry registry) {  //添加一个处理器还有定义处理器的处理路径  registry.addHandler(webSocketHandler, "/ws").addInterceptors(chatHandshakeInterceptor);  /\*  \* 在这里我们用到.withSockJS()，SockJS是spring用来处理浏览器对websocket的兼容性，  \* 目前浏览器支持websocket还不是很好，特别是IE11以下.  \* SockJS能根据浏览器能否支持websocket来提供三种方式用于websocket请求，  \* 三种方式分别是 WebSocket, HTTP Streaming以及 HTTP Long Polling  \*/  registry.addHandler(webSocketHandler, "/ws/sockjs").addInterceptors(chatHandshakeInterceptor).withSockJS();  }    } |

|  |
| --- |
| /\*\*  \* websocket的链接建立是基于http握手协议，我们可以添加一个拦截器处理握手之前和握手之后过程  \* **@author** BoBo  \*  \*/  @Component  **public** **class** ChatHandshakeInterceptor **implements** HandshakeInterceptor{  /\*\*  \* 握手之前，若返回false，则不建立链接  \*/  @Override  **public** **boolean** beforeHandshake(ServerHttpRequest request, ServerHttpResponse response, WebSocketHandler wsHandler,  Map<String, Object> attributes) **throws** Exception {  **if** (request **instanceof** ServletServerHttpRequest) {  ServletServerHttpRequest servletRequest = (ServletServerHttpRequest) request;  HttpSession session = servletRequest.getServletRequest().getSession(**false**);  //如果用户已经登录，允许聊天  **if**(session.getAttribute("loginUser")!=**null**){  //获取登录的用户  User loginUser=(User)session.getAttribute("loginUser") ;  //将用户放入socket处理器的会话(WebSocketSession)中  attributes.put("loginUser", loginUser);  System.***out***.println("Websocket:用户[ID:" + (loginUser.getId() + ",Name:"+loginUser.getNickname()+"]要建立连接"));  }**else**{  //用户没有登录，拒绝聊天  //握手失败！  System.***out***.println("--------------握手已失败...");  **return** **false**;  }  }  System.***out***.println("--------------握手开始...");  **return** **true**;  }  /\*\*  \* 握手之后  \*/  @Override  **public** **void** afterHandshake(ServerHttpRequest request, ServerHttpResponse response, WebSocketHandler wsHandler,  Exception exception) {  System.***out***.println("--------------握手成功啦...");  }  } |

|  |
| --- |
| /\*\*  \*  \* 说明：WebSocket处理器  \* **@author** 传智.BoBo老师  \* **@version** 1.0  \* **@date** 2016年10月27日  \*/  @Component("chatWebSocketHandler")  **public** **class** ChatWebSocketHandler **implements** WebSocketHandler {    //在线用户的SOCKETsession(存储了所有的通信通道)  **public** **static** **final** Map<String, WebSocketSession> ***USER\_SOCKETSESSION\_MAP***;    //存储所有的在线用户  **static** {  ***USER\_SOCKETSESSION\_MAP*** = **new** HashMap<String, WebSocketSession>();  }    /\*\*  \* webscoket建立好链接之后的处理函数--连接建立后的准备工作  \*/  @Override  **public** **void** afterConnectionEstablished(WebSocketSession webSocketSession) **throws** Exception {  //将当前的连接的用户会话放入MAP,key是用户编号  User loginUser=(User) webSocketSession.getAttributes().get("loginUser");  ***USER\_SOCKETSESSION\_MAP***.put(loginUser.getId(), webSocketSession);    //群发消息告知大家  Message msg = **new** Message();  msg.setText("风骚的【"+loginUser.getNickname()+"】踩着轻盈的步伐来啦。。。大家欢迎！");  msg.setDate(**new** Date());  //获取所有在线的WebSocketSession对象集合  Set<Entry<String, WebSocketSession>> entrySet = ***USER\_SOCKETSESSION\_MAP***.entrySet();  //将最新的所有的在线人列表放入消息对象的list集合中，用于页面显示  **for** (Entry<String, WebSocketSession> entry : entrySet) {  msg.getUserList().add((User)entry.getValue().getAttributes().get("loginUser"));  }    //将消息转换为json  TextMessage message = **new** TextMessage(GsonUtils.*toJson*(msg));  //群发消息  sendMessageToAll(message);    }  @Override  /\*\*  \* 客户端发送服务器的消息时的处理函数，在这里收到消息之后可以分发消息  \*/  //处理消息：当一个新的WebSocket到达的时候，会被调用（在客户端通过Websocket API发送的消息会经过这里，然后进行相应的处理）  **public** **void** handleMessage(WebSocketSession webSocketSession, WebSocketMessage<?> message) **throws** Exception {  //如果消息没有任何内容，则直接返回  **if**(message.getPayloadLength()==0)**return**;  //反序列化服务端收到的json消息  Message msg = GsonUtils.*fromJson*(message.getPayload().toString(), Message.**class**);  msg.setDate(**new** Date());  //处理html的字符，转义：  String text = msg.getText();  //转换为HTML转义字符表示  String htmlEscapeText = HtmlUtils.*htmlEscape*(text);  msg.setText(htmlEscapeText);  System.***out***.println("消息（可存数据库作为历史记录）:"+message.getPayload().toString());  //判断是群发还是单发  **if**(msg.getTo()==**null**||msg.getTo().equals("-1")){  //群发  sendMessageToAll(**new** TextMessage(GsonUtils.*toJson*(msg)));  }**else**{  //单发  sendMessageToUser(msg.getTo(), **new** TextMessage(GsonUtils.*toJson*(msg)));  }  }  @Override  /\*\*  \* 消息传输过程中出现的异常处理函数  \* 处理传输错误：处理由底层WebSocket消息传输过程中发生的异常  \*/  **public** **void** handleTransportError(WebSocketSession webSocketSession, Throwable exception) **throws** Exception {  // 记录日志，准备关闭连接  System.***out***.println("Websocket异常断开:" + webSocketSession.getId() + "已经关闭");  //一旦发生异常，强制用户下线，关闭session  **if** (webSocketSession.isOpen()) {  webSocketSession.close();  }    //群发消息告知大家  Message msg = **new** Message();  msg.setDate(**new** Date());    //获取异常的用户的会话中的用户编号  User loginUser=(User)webSocketSession.getAttributes().get("loginUser");  //获取所有的用户的会话  Set<Entry<String, WebSocketSession>> entrySet = ***USER\_SOCKETSESSION\_MAP***.entrySet();  //并查找出在线用户的WebSocketSession（会话），将其移除（不再对其发消息了。。）  **for** (Entry<String, WebSocketSession> entry : entrySet) {  **if**(entry.getKey().equals(loginUser.getId())){  msg.setText("万众瞩目的【"+loginUser.getNickname()+"】已经退出。。。！");  //清除在线会话  ***USER\_SOCKETSESSION\_MAP***.remove(entry.getKey());  //记录日志：  System.***out***.println("Socket会话已经移除:用户ID" + entry.getKey());  **break**;  }  }    //并查找出在线用户的WebSocketSession（会话），将其移除（不再对其发消息了。。）  **for** (Entry<String, WebSocketSession> entry : entrySet) {  msg.getUserList().add((User)entry.getValue().getAttributes().get("loginUser"));  }    TextMessage message = **new** TextMessage(GsonUtils.*toJson*(msg));  sendMessageToAll(message);    }  @Override  /\*\*  \* websocket链接关闭的回调  \* 连接关闭后：一般是回收资源等  \*/  **public** **void** afterConnectionClosed(WebSocketSession webSocketSession, CloseStatus closeStatus) **throws** Exception {  // 记录日志，准备关闭连接  System.***out***.println("Websocket正常断开:" + webSocketSession.getId() + "已经关闭");    //群发消息告知大家  Message msg = **new** Message();  msg.setDate(**new** Date());    //获取异常的用户的会话中的用户编号  User loginUser=(User)webSocketSession.getAttributes().get("loginUser");  Set<Entry<String, WebSocketSession>> entrySet = ***USER\_SOCKETSESSION\_MAP***.entrySet();  //并查找出在线用户的WebSocketSession（会话），将其移除（不再对其发消息了。。）  **for** (Entry<String, WebSocketSession> entry : entrySet) {  **if**(entry.getKey().equals(loginUser.getId())){  //群发消息告知大家  msg.setText("万众瞩目的【"+loginUser.getNickname()+"】已经有事先走了，大家继续聊...");  //清除在线会话  ***USER\_SOCKETSESSION\_MAP***.remove(entry.getKey());  //记录日志：  System.***out***.println("Socket会话已经移除:用户ID" + entry.getKey());  **break**;  }  }    //并查找出在线用户的WebSocketSession（会话），将其移除（不再对其发消息了。。）  **for** (Entry<String, WebSocketSession> entry : entrySet) {  msg.getUserList().add((User)entry.getValue().getAttributes().get("loginUser"));  }    TextMessage message = **new** TextMessage(GsonUtils.*toJson*(msg));  sendMessageToAll(message);  }  @Override  /\*\*  \* 是否支持处理拆分消息，返回true返回拆分消息  \*/  //是否支持部分消息：如果设置为true，那么一个大的或未知尺寸的消息将会被分割，并会收到多次消息（会通过多次调用方法handleMessage(WebSocketSession, WebSocketMessage). ）  //如果分为多条消息，那么可以通过一个api：org.springframework.web.socket.WebSocketMessage.isLast() 是否是某条消息的最后一部分。  //默认一般为false，消息不分割  **public** **boolean** supportsPartialMessages() {  **return** **false**;  }  /\*\*  \*  \* 说明：给某个人发信息  \* **@param** id  \* **@param** message  \* **@author** 传智.BoBo老师  \* **@throws** IOException  \* **@time**：2016年10月27日 下午10:40:52  \*/  **private** **void** sendMessageToUser(String id, TextMessage message) **throws** IOException{  //获取到要接收消息的用户的session  WebSocketSession webSocketSession = ***USER\_SOCKETSESSION\_MAP***.get(id);  **if** (webSocketSession != **null** && webSocketSession.isOpen()) {  //发送消息  webSocketSession.sendMessage(message);  }  }    /\*\*  \*  \* 说明：群发信息：给所有在线用户发送消息  \* **@author** 传智.BoBo老师  \* **@time**：2016年10月27日 下午10:40:07  \*/  **private** **void** sendMessageToAll(**final** TextMessage message){  //对用户发送的消息内容进行转义    //获取到所有在线用户的SocketSession对象  Set<Entry<String, WebSocketSession>> entrySet = ***USER\_SOCKETSESSION\_MAP***.entrySet();  **for** (Entry<String, WebSocketSession> entry : entrySet) {  //某用户的WebSocketSession  **final** WebSocketSession webSocketSession = entry.getValue();  //判断连接是否仍然打开的  **if**(webSocketSession.isOpen()){  //开启多线程发送消息（效率高）  **new** Thread(**new** Runnable() {  **public** **void** run() {  **try** {  **if** (webSocketSession.isOpen()) {  webSocketSession.sendMessage(message);  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  }).start();    }  }  }    } |

# 同学问题

问题：

