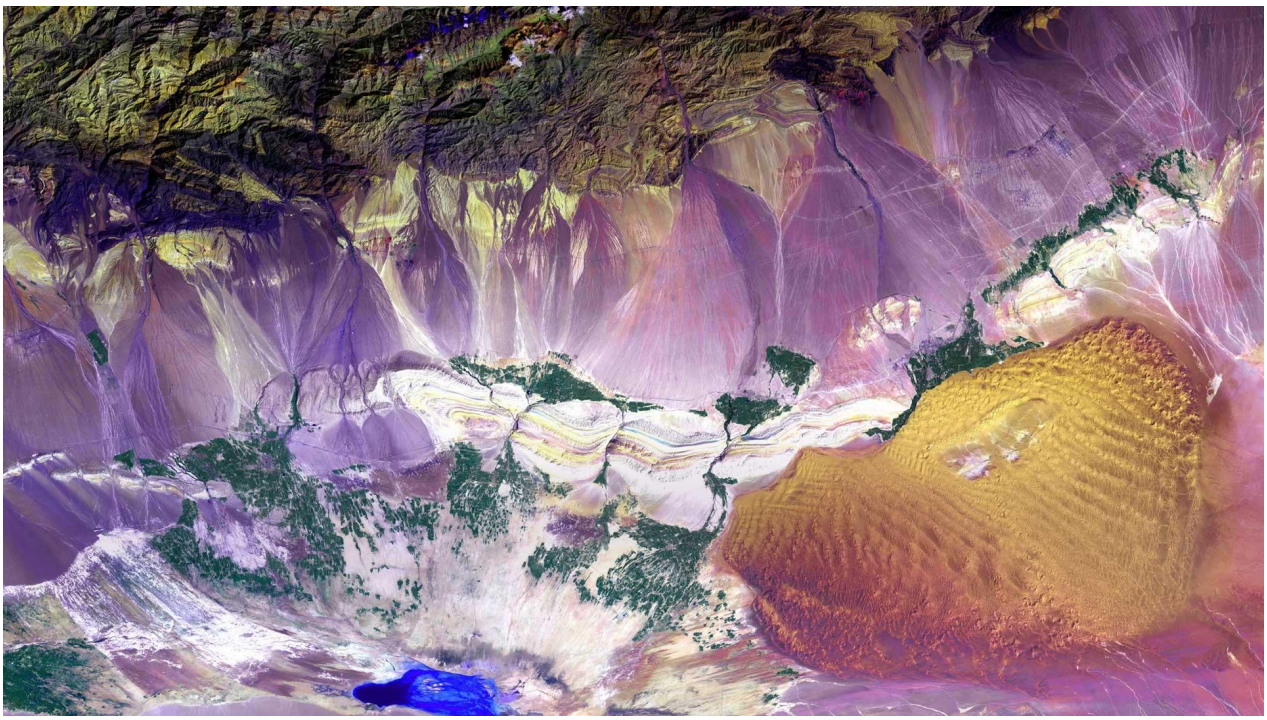
【websocket】spring boot 集成 websocket 的四种方式



集成 websocket 的四种方案

1. 原生注解

pom.xml

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

WebSocketConfig

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.config;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.server.standard.ServerEndpointExporter;

/**
 * @author buhao
 * @version WebSocketConfig.java, v 0.1 2019-10-18 15:45 buhao
 */
@Configuration
@EnableWebSocket
public class WebSocketConfig {

    @Bean
    public ServerEndpointExporter serverEndpoint() {
        return new ServerEndpointExporter();
    }
}
```

说明：

这个配置类很简单，通过这个配置 spring boot 才能去扫描后面的关于 websocket 的注解

WsServerEndpoint

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.ws;

import org.springframework.stereotype.Component;

import javax.websocket.OnClose;
import javax.websocket.OnMessage;
import javax.websocket.OnOpen;
import javax.websocket.Session;
import javax.websocket.server.ServerEndpoint;
import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

/**
 * @author buhao
 * @version WsServerEndpoint.java, v 0.1 2019-10-18 16:06 buhao
 */
@ServerEndpoint("/myWs")
```

```java
@Component
public class WsServerEndpoint {

    /**
     * 连接成功
     *
     * @param session
     */
    @OnOpen
    public void onOpen(Session session) {
        System.out.println("连接成功");
    }

    /**
     * 连接关闭
     *
     * @param session
     */
    @OnClose
    public void onClose(Session session) {
        System.out.println("连接关闭");
    }

    /**
     * 接收到消息
     *
     * @param text
     */
    @OnMessage
    public String onMsg(String text) throws IOException {
        return "servet 发送: " + text;
    }
}
```

说明

这里有几个注解需要注意一下，首先是他们的包都在 **javax.websocket** 下。并不是 spring 提供的，而 jdk 自带的，下面是他们的具体作用。

1. **@ServerEndpoint**
2. 通过这个 spring boot 就可以知道你暴露出去的 ws 应用的路径，有点类似我们经常用的 @RequestMapping。比如你的启动端口是8080，而这个注解的值是ws，那我们就可以通过 ws://127.0.0.1:8080/ws 来连接你的应用
3. **@OnOpen**
4. 当 websocket 建立连接成功后会触发这个注解修饰的方法，注意它有一个 Session 参数
5. **@OnClose**
6. 当 websocket 建立的连接断开后会触发这个注解修饰的方法，注意它有一个 Session 参数
7. **@OnMessage**
8. 当客户端发送消息到服务端时，会触发这个注解修改的方法，它有一个 String 入参表明客户端传

入的值
9. @OnError
10. 当 websocket 建立连接时出现异常会触发这个注解修饰的方法，注意它有一个 Session 参数

另外一点就是服务端如何发送消息给客户端，服务端发送消息必须通过上面说的 Session 类，通常是在@OnOpen 方法中，当连接成功后把 session 存入 Map 的 value，key 是与 session 对应的用户标识，当要发送的时候通过 key 获得 session 再发送，这里可以通过 **session.getBasicRemote().sendText()** 来对客户端发送消息。

2. Spring封装

pom.xml

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

HttpAuthHandler

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.handler;

import cn.coder4j.study.example.websocket.config.WsSessionManager;
import org.springframework.stereotype.Component;
import org.springframework.web.socket.CloseStatus;
import org.springframework.web.socket.TextMessage;
import org.springframework.web.socket.WebSocketSession;
import org.springframework.web.socket.handler.TextWebSocketHandler;

import java.time.LocalDateTime;

/**
 * @author buhao
 * @version MyWSHandler.java, v 0.1 2019-10-17 17:10 buhao
 */
@Component
public class HttpAuthHandler extends TextWebSocketHandler {

    /**
     * socket 建立成功事件
     *
     * @param session
```

```
     * @throws Exception
     */
    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws
Exception {
        Object token = session.getAttributes().get("token");
        if (token != null) {
            // 用户连接成功，放入在线用户缓存
            WsSessionManager.add(token.toString(), session);
        } else {
            throw new RuntimeException("用户登录已经失效!");
        }
    }

    /**
     * 接收消息事件
     *
     * @param session
     * @param message
     * @throws Exception
     */
    @Override
    protected void handleTextMessage(WebSocketSession session, TextMessage
message) throws Exception {
        // 获得客户端传来的消息
        String payload = message.getPayload();
        Object token = session.getAttributes().get("token");
        System.out.println("server 接收到 " + token + " 发送的 " + payload);
        session.sendMessage(new TextMessage("server 发送给 " + token + " 消息 "
+ payload + " " + LocalDateTime.now().toString()));
    }

    /**
     * socket 断开连接时
     *
     * @param session
     * @param status
     * @throws Exception
     */
    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus
status) throws Exception {
        Object token = session.getAttributes().get("token");
        if (token != null) {
            // 用户退出，移除缓存
            WsSessionManager.remove(token.toString());
        }
    }
```

```
}
```

说明

通过继承 **TextWebSocketHandler** 类并覆盖相应方法，可以对 websocket 的事件进行处理，这里可以同原生注解的那几个注解连起来看

1. **afterConnectionEstablished** 方法是在 socket 连接成功后被触发，同原生注解里的 @OnOpen 功能
2. **afterConnectionClosed** 方法是在 socket 连接关闭后被触发，同原生注解里的 @OnClose 功能
3. **handleTextMessage** 方法是在客户端发送信息时触发，同原生注解里的 @OnMessage 功能

WsSessionManager

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.config;

import lombok.extern.slf4j.Slf4j;
import org.springframework.web.socket.WebSocketSession;

import java.io.IOException;
import java.util.concurrent.ConcurrentHashMap;

/**
 * @author buhao
 * @version WsSessionManager.java, v 0.1 2019-10-22 10:24 buhao
 */
@Slf4j
public class WsSessionManager {
    /**
     * 保存连接 session 的地方
     */
    private static ConcurrentHashMap<String, WebSocketSession> SESSION_POOL =
new ConcurrentHashMap<>();

    /**
     * 添加 session
     *
     * @param key
     */
    public static void add(String key, WebSocketSession session) {
        // 添加 session
        SESSION_POOL.put(key, session);
```

```
    }

    /**
     * 删除 session,会返回删除的 session
     *
     * @param key
     * @return
     */
    public static WebSocketSession remove(String key) {
        // 删除 session
        return SESSION_POOL.remove(key);
    }

    /**
     * 删除并同步关闭连接
     *
     * @param key
     */
    public static void removeAndClose(String key) {
        WebSocketSession session = remove(key);
        if (session != null) {
            try {
                // 关闭连接
                session.close();
            } catch (IOException e) {
                // todo: 关闭出现异常处理
                e.printStackTrace();
            }
        }
    }

    /**
     * 获得 session
     *
     * @param key
     * @return
     */
    public static WebSocketSession get(String key) {
        // 获得 session
        return SESSION_POOL.get(key);
    }
}
```

说明

这里简单通过 **ConcurrentHashMap** 来实现了一个 session 池，用来保存已经登录的 web socket 的 session。前文提过，服务端发送消息给客户端必须要通过这个 session。

MyInterceptor

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.interceptor;

import cn.hutool.core.util.StrUtil;
import cn.hutool.http.HttpUtil;
import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.stereotype.Component;
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.server.HandshakeInterceptor;

import java.util.HashMap;
import java.util.Map;

/**
 * @author buhao
 * @version MyInterceptor.java, v 0.1 2019-10-17 19:21 buhao
 */
@Component
public class MyInterceptor implements HandshakeInterceptor {

    /**
     * 握手前
     *
     * @param request
     * @param response
     * @param wsHandler
     * @param attributes
     * @return
     * @throws Exception
     */
    @Override
    public boolean beforeHandshake(ServerHttpRequest request,
ServerHttpResponse response, WebSocketHandler wsHandler, Map<String, Object>
attributes) throws Exception {
        System.out.println("握手开始");
        // 获得请求参数
        HashMap<String, String> paramMap =
HttpUtil.decodeParamMap(request.getURI().getQuery(), "utf-8");
        String uid = paramMap.get("token");
        if (StrUtil.isNotBlank(uid)) {
            // 放入属性域
            attributes.put("token", uid);
            System.out.println("用户 token " + uid + " 握手成功! ");
```

```
            return true;
        }
        System.out.println("用户登录已失效");
        return false;
    }

    /**
     * 握手后
     *
     * @param request
     * @param response
     * @param wsHandler
     * @param exception
     */
    @Override
    public void afterHandshake(ServerHttpRequest request, ServerHttpResponse
response, WebSocketHandler wsHandler, Exception exception) {
        System.out.println("握手完成");
    }

}
```

说明

通过实现 **HandshakeInterceptor** 接口来定义握手拦截器，注意这里与上面 **Handler** 的事件是不同的，这里是建立握手时的事件，分为握手前与握手后，而 **Handler** 的事件是在握手成功后的基础上建立 socket 的连接。所以在如果把认证放在这个步骤相对来说最节省服务器资源。它主要有两个方法 **beforeHandshake** 与 **afterHandshake**，顾名思义一个在握手前触发，一个在握手后触发。

WebSocketConfig

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.config;

import cn.coder4j.study.example.websocket.handler.HttpAuthHandler;
import cn.coder4j.study.example.websocket.interceptor.MyInterceptor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.socket.config.annotation.EnableWebSocket;
import org.springframework.web.socket.config.annotation.WebSocketConfigurer;
import
org.springframework.web.socket.config.annotation.WebSocketHandlerRegistry;

/**
```

```
 * @author buhao
 * @version WebSocketConfig.java, v 0.1 2019-10-17 15:43 buhao
 */
@Configuration
@EnableWebSocket
public class WebSocketConfig implements WebSocketConfigurer {

    @Autowired
    private HttpAuthHandler httpAuthHandler;
    @Autowired
    private MyInterceptor myInterceptor;

    @Override
    public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry
                .addHandler(httpAuthHandler, "myWS")
                .addInterceptors(myInterceptor)
                .setAllowedOrigins("*");
    }
}
```

说明

通过实现 **WebSocketConfigurer** 类并覆盖相应的方法进行 **websocket** 的配置。我们主要覆盖 **registerWebSocketHandlers** 这个方法。通过向 **WebSocketHandlerRegistry** 设置不同参数来进行配置。其中 **addHandler** 方法添加我们上面的写的 ws 的 handler 处理类，第二个参数是你暴露出的 ws 路径。**addInterceptors** 添加我们写的握手过滤器。**setAllowedOrigins("_") _这个是关闭跨域校验，方便本地调试，线上推荐打开。

3. TIO

pom.xml

```
<dependency>
     <groupId>org.t-io</groupId>
     <artifactId>tio-websocket-spring-boot-starter</artifactId>
     <version>3.5.5.v20191010-RELEASE</version>
</dependency>
```

application.xml

```
tio:
  websocket:
    server:
      port: 8989
```

说明

这里只配置了 ws 的启动端口，还有很多配置，可以通过结尾给的链接去寻找

MyHandler

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.handler;

import org.springframework.stereotype.Component;
import org.tio.core.ChannelContext;
import org.tio.http.common.HttpRequest;
import org.tio.http.common.HttpResponse;
import org.tio.websocket.common.WsRequest;
import org.tio.websocket.server.handler.IWsMsgHandler;

/**
 * @author buhao
 * @version MyHandler.java, v 0.1 2019-10-21 14:39 buhao
 */
@Component
public class MyHandler implements IWsMsgHandler {
    /**
     * 握手
     *
     * @param httpRequest
     * @param httpResponse
     * @param channelContext
     * @return
     * @throws Exception
     */
    @Override
    public HttpResponse handshake(HttpRequest httpRequest, HttpResponse
httpResponse, ChannelContext channelContext) throws Exception {
        return httpResponse;
    }

    /**
     * 握手成功
     *
     * @param httpRequest
     * @param httpResponse
     * @param channelContext
     * @throws Exception
     */
    @Override
```

```
    public void onAfterHandshaked(HttpRequest httpRequest, HttpResponse
httpResponse, ChannelContext channelContext) throws Exception {
        System.out.println("握手成功");
    }

    /**
     * 接收二进制文件
     *
     * @param wsRequest
     * @param bytes
     * @param channelContext
     * @return
     * @throws Exception
     */
    @Override
    public Object onBytes(WsRequest wsRequest, byte[] bytes, ChannelContext
channelContext) throws Exception {
        return null;
    }

    /**
     * 断开连接
     *
     * @param wsRequest
     * @param bytes
     * @param channelContext
     * @return
     * @throws Exception
     */
    @Override
    public Object onClose(WsRequest wsRequest, byte[] bytes, ChannelContext
channelContext) throws Exception {
        System.out.println("关闭连接");
        return null;
    }

    /**
     * 接收消息
     *
     * @param wsRequest
     * @param s
     * @param channelContext
     * @return
     * @throws Exception
     */
    @Override
    public Object onText(WsRequest wsRequest, String s, ChannelContext
channelContext) throws Exception {
        System.out.println("接收文本消息:" + s);
```

```
        return "success";
    }
}
```

说明

这个同上个例子中的 handler 很像，也是通过实现接口覆盖方法来进行事件处理，实现的接口是**IWsMsgHandler**，它的方法功能如下

1. handshake
2. 在握手的时候触发
3. onAfterHandshaked
4. 在握手成功后触发
5. onBytes
6. 客户端发送二进制消息触发
7. onClose
8. 客户端关闭连接时触发
9. onText
10. 客户端发送文本消息触发

StudyWebsocketExampleApplication

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */

package cn.coder4j.study.example.websocket;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.tio.websocket.starter.EnableTioWebSocketServer;

@SpringBootApplication
@EnableTioWebSocketServer
public class StudyWebsocketExampleApplication {

    public static void main(String[] args) {
        SpringApplication.run(StudyWebsocketExampleApplication.class, args);
    }
}
```

说明

这个类的名称不重要，它其实是你的 spring boot 启动类，只要记得加上**@EnableTioWebSocketServer**注解就可以了

STOMP

pom.xml

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

WebSocketConfig

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.config;

import org.springframework.context.annotation.Configuration;
import org.springframework.messaging.simp.config.MessageBrokerRegistry;
import
org.springframework.web.socket.config.annotation.EnableWebSocketMessageBroker;
import org.springframework.web.socket.config.annotation.StompEndpointRegistry;
import
org.springframework.web.socket.config.annotation.WebSocketMessageBrokerConfigu
rer;

/**
 * @author buhao
 * @version WebSocketConfig.java, v 0.1 2019-10-21 16:32 buhao
 */
@Configuration
@EnableWebSocketMessageBroker
public class WebSocketConfig implements WebSocketMessageBrokerConfigurer {

    @Override
    public void registerStompEndpoints(StompEndpointRegistry registry) {
        // 配置客户端尝试连接地址
        registry.addEndpoint("/ws").setAllowedOrigins("*").withSockJS();
    }

    @Override
    public void configureMessageBroker(MessageBrokerRegistry registry) {
        // 设置广播节点
        registry.enableSimpleBroker("/topic", "/user");
        // 客户端向服务端发送消息需有/app 前缀
        registry.setApplicationDestinationPrefixes("/app");
```

```
        // 指定用户发送（一对一）的前缀 /user/
        registry.setUserDestinationPrefix("/user/");
    }
}
```

说明

1. 通过实现 **WebSocketMessageBrokerConfigurer** 接口和加上**@EnableWebSocketMessageBroker**来进行 stomp 的配置与注解扫描。
2. 其中覆盖 **registerStompEndpoints** 方法来设置暴露的 stomp 的路径，其它一些跨域、客户端之类的设置。
3. 覆盖 **configureMessageBroker** 方法来进行节点的配置。
4. 其中 **enableSimpleBroker** 配置的广播节点，也就是服务端发送消息，客户端订阅就能接收消息的节点。
5. 覆盖**setApplicationDestinationPrefixes** 方法，设置客户端向服务端发送消息的节点。
6. 覆盖 **setUserDestinationPrefix** 方法，设置一对一通信的节点。

WSController

```
/*
 * *
 *  * blog.coder4j.cn
 *  * Copyright (C) 2016-2019 All Rights Reserved.
 *
 */
package cn.coder4j.study.example.websocket.controller;

import cn.coder4j.study.example.websocket.model.RequestMessage;
import cn.coder4j.study.example.websocket.model.ResponseMessage;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.messaging.handler.annotation.MessageMapping;
import org.springframework.messaging.handler.annotation.SendTo;
import org.springframework.messaging.simp.SimpMessagingTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.ResponseBody;

/**
 * @author buhao
 * @version WSController.java, v 0.1 2019-10-21 17:22 buhao
 */
@Controller
public class WSController {

    @Autowired
    private SimpMessagingTemplate simpMessagingTemplate;

    @MessageMapping("/hello")
    @SendTo("/topic/hello")
```

```
    public ResponseMessage hello(RequestMessage requestMessage) {
        System.out.println("接收消息：" + requestMessage);
        return new ResponseMessage("服务端接收到你发的：" + requestMessage);
    }

    @GetMapping("/sendMsgByUser")
    public @ResponseBody
    Object sendMsgByUser(String token, String msg) {
        simpMessagingTemplate.convertAndSendToUser(token, "/msg", msg);
        return "success";
    }

    @GetMapping("/sendMsgByAll")
    public @ResponseBody
    Object sendMsgByAll(String msg) {
        simpMessagingTemplate.convertAndSend("/topic", msg);
        return "success";
    }

    @GetMapping("/test")
    public String test() {
        return "test-stomp.html";
    }
}
```

说明

1. 通过 @MessageMapping 来暴露节点路径，有点类似 @RequestMapping 。注意这里虽然写的是 hello ，但是我们客户端调用的真正地址是** /app/hello。 **因为我们在上面的 config 里配置了**registry.setApplicationDestinationPrefixes("/app")**。
2. @SendTo 这个注解会把返回值的内容发送给订阅了 **/topic/hello** 的客户端，与之类似的还有一个@SendToUser 只不过他是发送给用户端一对一通信的。这两个注解一般是应答时响应的，如果服务端主动发送消息可以通过 **simpMessagingTemplate**类的**convertAndSend**方法。注意 **simpMessagingTemplate.convertAndSendToUser(token, "/msg", msg)** ，联系到我们上文配置的 **registry.setUserDestinationPrefix("/user/"),**这里客户端订阅的是**/user/{token}/msg**,千万不要搞错。

Session 共享的问题

上面反复提到一个问题就是，服务端如果要主动发送消息给客户端一定要用到 session。而大家都知道的是 session 这个东西是不跨 jvm 的。如果有多台服务器，在 http 请求的情况下，我们可以通过把 session 放入缓存中间件中来共享解决这个问题，通过 spring session 几条配置就解决了。但是 web socket 不可以。他的 session 是不能序列化的，当然这样设计的目的不是为了为难你，而是出于对 http 与 web socket 请求的差异导致的。
目前网上找到的最简单方案就是通过 redis 订阅广播的形式，主要代码跟第二种方式差不多，你要在本地放个 map 保存请求的 session。也就是说每台服务器都会保存与他连接的 session 于本地。然后发消息的地方要修改，并不是现在这样直接发送，而通过 redis 的订阅机制。服务器要发消息的时候，你通过 redis 广播这条消息，所有订阅的服务端都会收到这个消息，然后本地尝试发送。最后肯定只有有这

个对应用户 session 的那台才能发送出去。

如何选择

1. 如果你在使用 tio，那推荐使用 tio 的集成。因为它已经实现了很多功能，包括上面说的通过 redis 的 session 共享，只要加几个配置就可以了。但是 tio 是半开源，文档是需要收费的。如果没有使用，那就忘了他。
2. 如果你的业务要求比较灵活多变，推荐使用前两种，更推荐第二种 Spring 封装的形式。
3. 如果只是简单的服务器双向通信，推荐 stomp 的形式，因为他更容易规范使用。

其它

1. websocket 在线验证

写完服务端代码后想调试，但是不会前端代码怎么办，点这里，这是一个在线的 websocket 客户端，功能完全够我们调试了。

2. stomp 验证

这个没找到在线版的，但是网上有很多 demo 可以下载到本地进行调试，也可以通过后文的连接找到。

3. 另外由于篇幅有限，并不能放上所有代码，但是测试代码全都上传 gitlab，保证可以正常运行，可以在 这里 找到

参考链接

1. SpringBoot 系统 - 集成 WebSocket 实时通信
2. WebSocket 的故事（二）—— Spring 中如何利用 STOMP 快速构建 WebSocket 广播式消息模式
3. SpringBoot集成WebSocket【基于纯H5】进行点对点[一对一]和广播[一对多]实时推送
4. Spring Framework 参考文档（WebSocket STOMP）
5. Spring Boot中使用WebSocket总结（一）：几种实现方式详解
6. Spring Boot 系列 - WebSocket 简单使用
7. tio-websocket-spring-boot-starter