失物招领WebSocket在线聊天研发记录

记录时间：22届TikciNet失物招领重构研发 2024.07.01~研发完成

记录原因：WebSocket在日常研发使用的频率自然是没有Http 和 Https用的多，相关研发人员在websocket的研发经验上自然也没那么充足，工作室以往的项目中暂时没看见多少WebSocket相关的技术使用和文档参考。记录此次文档，方便以后自我复习，也方便以后工作室若有WebSocket相关的研发，希望能起到一定的参考作用。

文档几乎都是自己找资料and结合自己的研发过程记录，若有不对请各位纠正

WebSocket简介

WebSocket是一种在网络通信中的协议，它是独立于HTTP协议的。该协议基于TCP/IP协议，可以提供双向通讯并保有状态。这意味着客户端和服务器可以进行实时响应，并且这种响应是双向的。WebSocket协议端口通常是80，443。

WebSocket的出现使得浏览器具备了实时双向通信的能力。与HTTP这种非持久单向响应应答的协议相比，WebSocket是一个持久化的协议。举例来说，即使在关闭网页或者浏览器后，WebSocket的连接仍然保持，用户也可以继续接收到服务器的消息。它允许服务端主动发送信息给客户端，是实现推送（Push）技术的一种非常流行的解决方案。

此外，要建立WebSocket连接，需要浏览器和服务器握手进行建立连接。一旦连接建立，WebSocket可以在浏览器和服务器之间双向发送或接受信息。总的来说，WebSocket提供了一个高效、实时的双向通信方案。

优点：

1. **全双工通信：** WebSocket一旦建立连接，服务器和客户端就可以互相发送消息，直到任意一方关闭连接。
2. **节省资源：** 相比于HTTP轮询，WebSocket在建立连接后不需要频繁地打开和关闭连接，节省了带宽和服务器的资源。
3. **低延迟：** 由于是持久连接，WebSocket在传输数据时的延迟较低，适合需要快速交互的应用，如在线聊天室。
4. **标准支持：** WebSocket是W3C标准，得到了主流浏览器的支持。

缺点：

1. **跨代理问题：** 在某些网络环境下，WebSocket可能会受到代理服务器的影响。
2. **兼容性：** 尽管主流浏览器支持WebSocket，但在一些旧版本浏览器中可能需要回退到轮询等机制。

技术研发记录正文

技术选择

首先是一开始的技术选型，后端选择是的是纯粹的WebSocket，而不是采用Netty整合WebSocket的形式，**第一是因为Netty相对来说学习线路比WebSocket更加复杂一点，我相信接触过WebSocket的人应该比Netty多，方便更多人能了解，学习，参与到这篇记录中；第二是纯粹的WebSocket自然是更加简单，没那么底层的去控制和管理网络资源，适合快速开发**

需求分析

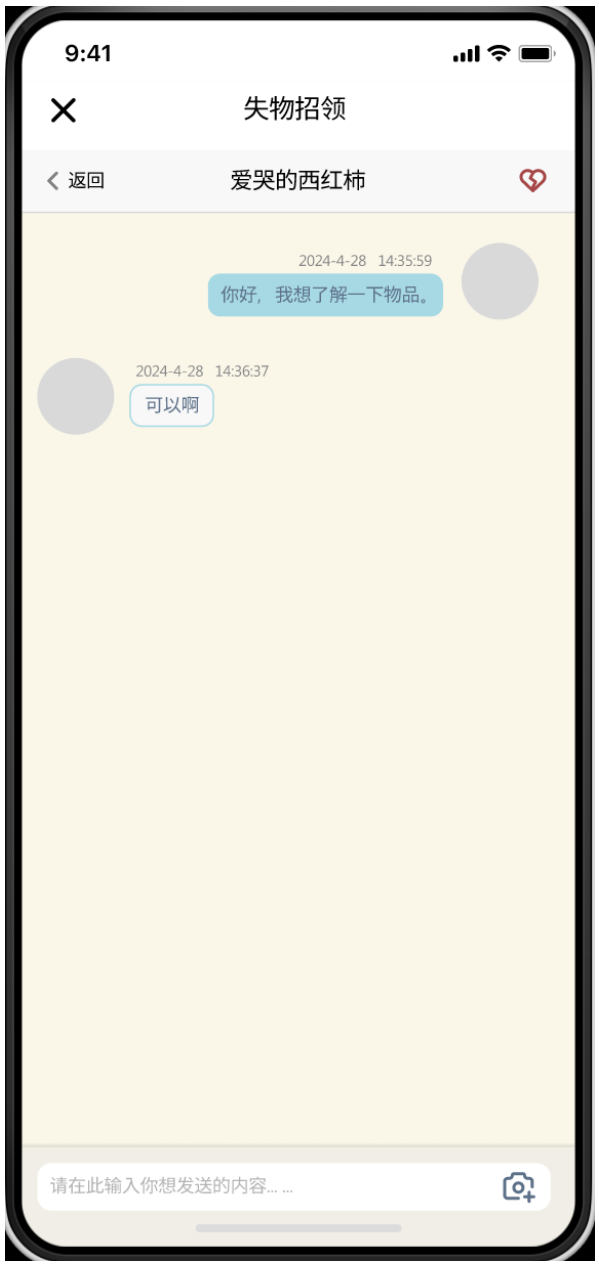
最重要的就是聊天，但是在聊天基础上，我们还需要一些拓展的功能，让其更加功能丰富：

1：进行websocket连接前需要进行身份校验 自然是不能允许非学校允许用户参与到聊天中

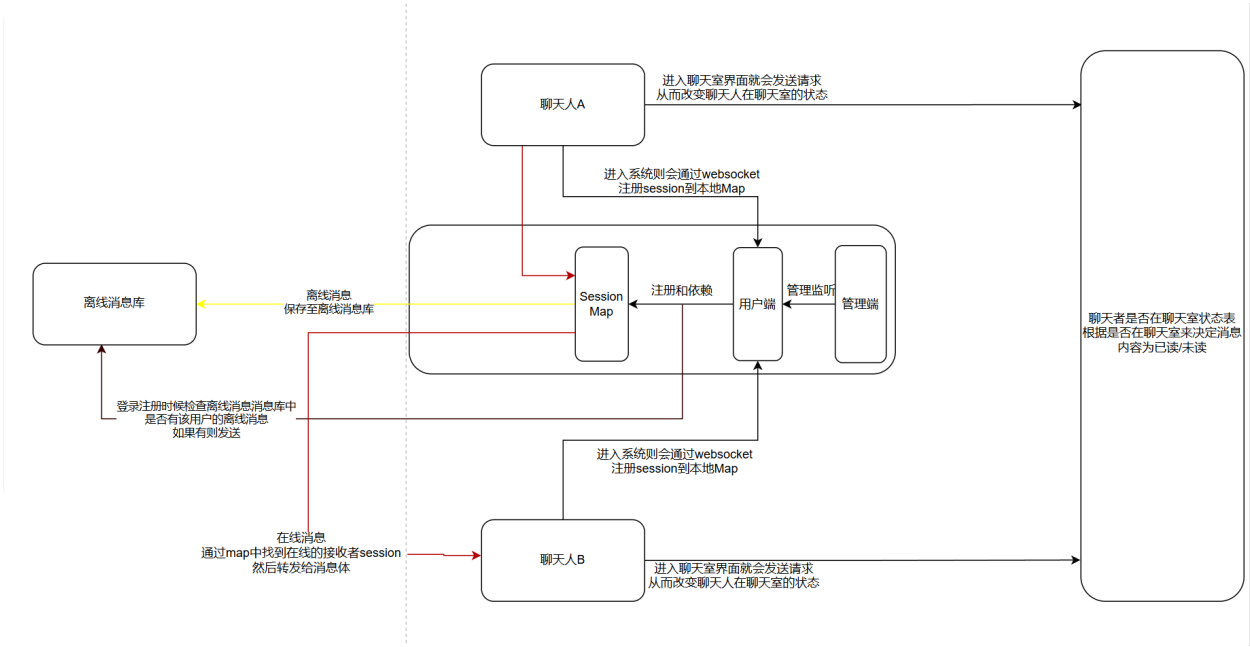
2：聊天与聊天之间自然需要隔离

3：聊天内容准确发送，保留

4：产品需要，做出已读，未读状态区分；管理端有对话管理功能，所以对于对话需要有记录和区分，然后管理端需要能主动推送提醒/警告/关闭/封禁的消息通知功能



基础聊天模块架构（未加入黑名单检测；封禁账号检测；被禁对话检测）



研发记录

（因为是研发记录 可能随时有个想法就出来了 记录顺序上可能有点乱）

依赖问题和连接问题

首先是基础websocket服务的搭建，第一个问题就来了：

首先是导入依赖，然后要记得要**配置类,配置类，配置类**

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

```
@Configuration
@EnableWebSocket  // 这个可以写道启动类上
public class WebSocketConfig {
    @Bean
    public ServerEndpointExporter serverEndpointExporter() {
        return new ServerEndpointExporter();
    }
}
```

然后是websocket的主体代码 下面是网上找的案例结构

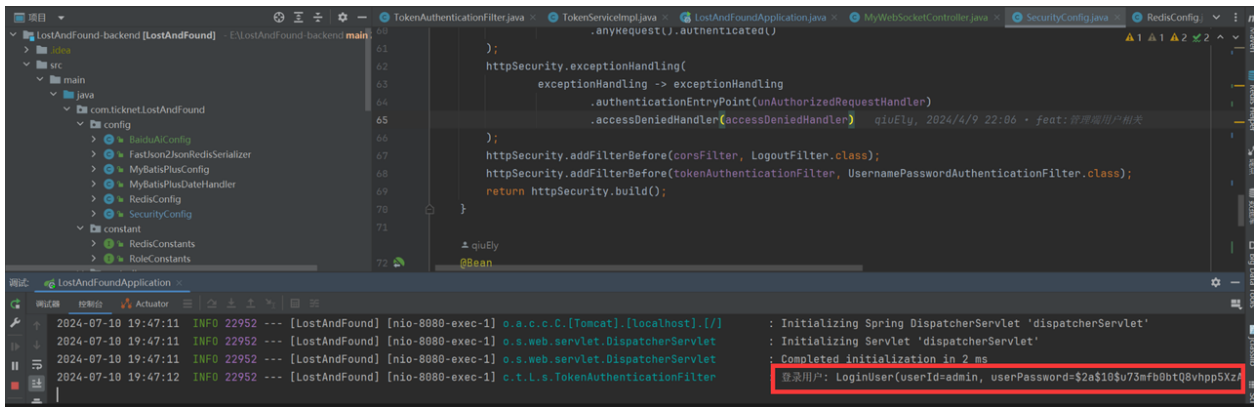
```
package com.loit.park.common.websocket;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import javax.websocket.*;
import javax.websocket.server.PathParam;
import javax.websocket.server.ServerEndpoint;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.CopyOnWriteArraySet;

@Component
@ServerEndpoint("/websocket/{userId}")
```

```java
public class WebSocketServer {
    /**
     * 日志工具
     */
    private Logger logger = LoggerFactory.getLogger(this.getClass());
    /**
     * 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;
    /**
     * 用户id
     */
    private String userId;
    /**
     * 用来存放每个客户端对应的MyWebSocket对象
     */
    private static CopyOnWriteArraySet<WebSocketServer> webSockets = new CopyOnWriteArraySet<>();
    /**
     * 用来存在线连接用户信息
     */
    private static ConcurrentHashMap<String, Session> sessionPool = new ConcurrentHashMap<String, Session>();

    /**
     * 链接成功调用的方法
     */
    @OnOpen
    public void onOpen(Session session, @PathParam(value = "userId") String userId) {

    }

    /**
     * 链接关闭调用的方法
     */
    @OnClose
    public void onClose() {

    }

    /**
     * 收到客户端消息后调用的方法
     */
    @OnMessage
    public void onMessage(String message) {
        logger.info("【websocket消息】收到客户端消息:" + message);
    }

    /**
     * 发送错误时的处理
     *
     * @param session
     * @param error2
     */
    @OnError
    public void onError(Session session, Throwable error) {
        logger.error("用户错误,原因:" + error.getMessage());
        error.printStackTrace();
    }

```

```
73      /**
74       * 此为广播消息
75       */
76      public void sendAllMessage(String message) {
77      }
78
79      /**
80       * 此为单点消息
81       */
82      public void sendOneMessage(String userId, String message) {
83      }
84
85      /**
86       * 此为单点消息(多人)
87       */
88      public void sendMoreMessage(String[] userIds, String message)
    {
89      }
90  }
91
```

第一版的尝试没有中就是忘记写配置类，出现了：websocket的请求能被过滤器过滤到（身份验证采用的Spring Security），但是连接不上，因为websocket本质上还是Http请求，所以最终在Security中走的应该还是Http请求，所以会发现过滤器中依旧能过滤并且获取到其中的头信息token并且能正确身份校验

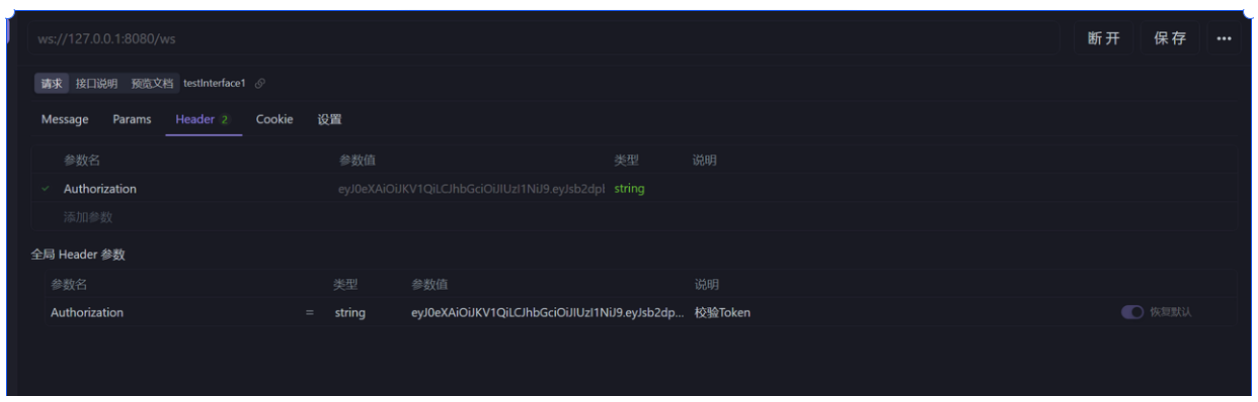
（从这里也能注意到 我们在Spring Security中虽然配置了 /websocket 的放行 但是实际上还是走了过滤器链 这是security过滤器链执行逻辑决定的 security的对于特定URL的放行一般是在最后一个过滤器FilterSecurityInterceptor中执行的 所以其实从某种角度来说任何一个请求都走了我们的token校验过滤器）



```java
@Configuration
@EnableWebSecurity
public class SecurityConfig implements WebSocketMessageBrokerConfigurer {
    @Resource
    private TokenAuthenticationFilter tokenAuthenticationFilter;
    @Resource
    private UnAuthorizedRequestHandler unAuthorizedRequestHandler;
    @Resource
    private CORSFilter corsFilter;
    @Resource
    private AccessDeniedHandlerImpl accessDeniedHandler;
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
        httpSecurity.csrf(AbstractHttpConfigurer::disable)
                .cors(AbstractHttpConfigurer::disable)
                .sessionManagement(
                        sessionManagement -> sessionManagement.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                );
        httpSecurity.authorizeHttpRequests(
                authorizeRequests -> authorizeRequests
                        .requestMatchers("/websocket").anonymous()
                        .requestMatchers("/test/baidu/text","/test/baidu/img","/test/fakerName","/test/ocr/bankcard","/test/ocr/idcard","/user/login","u
                        .requestMatchers(HttpMethod.OPTIONS).permitAll()
                        .requestMatchers("/admin/**").hasRole("ADMIN")
                        .requestMatchers("/user/**").hasRole("USER")
                        .anyRequest().authenticated()
        );
        httpSecurity.exceptionHandling(
                exceptionHandling -> exceptionHandling
                        .authenticationEntryPoint(unAuthorizedRequestHandler)
                        .accessDeniedHandler(accessDeniedHandler)
        );
        httpSecurity.addFilterBefore(corsFilter, LogoutFilter.class);
        httpSecurity.addFilterBefore(tokenAuthenticationFilter, UsernamePasswordAuthenticationFilter.class);
```

```java
@Component   You, Yesterday · Uncommitted changes
@ServerEndpoint(value = "/websocket")
public class MyWebSocketController {

    private volatile static List<Session> sessions = Collections.synchronizedList(new ArrayList());
    //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;
    //设置为静态的 公用一个消息map ConcurrentMap为线程安全的map  HashMap不安全
    private static ConcurrentMap<String, Map<String, List<Object>>> messageMap=new ConcurrentHashMap<>();
    /**
     * /**
     * 连接建立成功调用的方法
     * @param session 可选的参数，session为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
     * @param userId 用户id
     * @throws Exception
     */
    @OnOpen
    public void onOpen(Session session) throws Exception{
        System.out.println("开始");
        this.session = session;
        // sessions.add((Session) this);
        String key="";//当前用户id
        String objectUserId="";//对象id
        if("".equals(session.getQueryString()) || session.getQueryString()==null) {

            key= "";
        }else {
            String keString=session.getQueryString();
            if(keString.length()>1) {
                key=keString.split( regex: ",")[0];
                objectUserId=keString.split( regex: ",")[1];
            }else {
```

（这里采用的是APIFox模拟的websocket请求，能携带我们的全局参数头，但是查询和询问相关前端人员，但websocket 不支持自定义 `Headers` 头，所以真正研发中这一步行不通，后续还需要修改，这里有关于后续的身份校验问题，在后面会再讲解）



然后为了解决上述问题，发现配置类没有配置，所以开始补上配置类，然后就出现了依赖报错

```
Error creating bean with name 'serverEndpointExporter' defined in class path resource [org/springblade/lab/external/webScoket/WebSocketConfig.class]: Invocation of init method failed; nested exception is java.lang.IllegalStateException: javax.websocket.server.ServerContainer not available
```

发现是`ServerEndpointExporter()`这个类自动注入失败，这是因为依赖冲突导致实现类有多个，不知道注入哪一个导致为null，解决方案参考阿里云社区
https://developer.aliyun.com/article/1430614

登录校验问题

根据上述的情况，我们知道websokcet的头信息不是放在我们的"Authorization"头中，但是因为是用的APIFOX测试，一开始并没针对这个做思考，直接用现成的全局参数进行测试也方便。然后我想到了如下的解决方案：

方案一（利用上下文）

每一个websocket连接其实我们关注的是session信息，对于每一个websocket的端点，`ServerEndpointConfig`的`getUserProperties()`方法返回一个`Map<String, Object>`，它允许你在WebSocket会话期间存储用户特定的属性，也就是可以理解为这个Map是每一个连接websocket的session的一个上下文信息的感觉。我们对于每一个session可以获取到这个Map，从而从Map中拿取和放入信息，那么就可以从Map中放入token方便后续校验

`modifyHandshake`方法是在WebSocket握手阶段执行的，这个阶段发生在WebSocket连接实际建立之前。

```
@Configuration
public class WebSocketConfig extends
ServerEndpointConfig.Configurator{
        @Bean
        public ServerEndpointExporter serverEndpointExporter() {
            return new ServerEndpointExporter();
        }
        @Override
        public void modifyHandshake(ServerEndpointConfig sec,
HandshakeRequest request, HandshakeResponse response) {
            HttpSession httpSession = (HttpSession)
request.getHttpSession();
            if (httpSession != null) {
                // 读取session域中存储的数据
                sec.getUserProperties().put("sessionid",
httpSession.getId());
                Map<String, List<String>> headers =
request.getHeaders();
                List<String> authorizations =
headers.get("Authorization");
                if (authorizations != null &&
!authorizations.isEmpty()) {
                    // 假设只有一个Authorization头
                    String token = authorizations.get(0);
                    // 将token存储在用户属性中 方便后续进行校验
                    sec.getUserProperties().put("token", token);
                }
            }
            super.modifyHandshake(sec, request, response);
        }
    }
```

对此，我们就有了身份上下文的token信息，我们知道，websocket连接成功后都会触发@onOpen方法，那么我们就可以在这个方法的开始的时候拿到token信息进行处理。

这个时候就引出另外一个问题 需要在websocket端点中进行身份校验，自然是能利用我们之前写好的校验相关的服务是最好的，自然就需要自动注入使用@Autowried 或者@Resource 使用后你就会发现 这些对象全部注入失败，运行爆空指针异常。

这是因为我们的Bean是在Spring的环境下，也就是Spring管理的，但是因为`@ServerEndpoint`是由Java WebSocket API（JSR-356）定义的，而该API的端点实例是由WebSocket容器（如Tomcat、Jetty）直接创建的，不是由Spring容器管理的。所以其实我们无法获取到这个Bean。

```
@Component
@Slf4j
@ServerEndpoint(value = "/websocket",configurator = WebSocketConfig.class)
public class MyWebSocketController {

    // redis操作缓存
    private RedisCache redisCache;

    // 消息记录
    private MessageService messageService;

    // 聊天状态
    private ChatStateService chatStateService;

    private final static String LOGIN_USER_KEY = "login_user_key";

    //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;

    private LoginUser loginUser;
```

这一点问题说大不大 说小不小，无法自动注入我们可以尝试原始的new对象，但是对于一般的service还好，对于RedisTemplate这种你还需要单独new的时候指定配置连接信息，序列化和反序列化极其麻烦，还有更多对象中还依赖自动注入的，那些就会更加麻烦。

所以为了解决这个问题，**利用spring的上下文，在spring的上下文信息中可以根据类名/类获取对应Bean**，在Spring容器中轻松获取，所以我们可以写一个工具类专门用于websocket端点中获取Bean对象（类对象是可以被导入的 工具类中使用static标记获取到的applicationContext 使得其在websocket环境中也能使用）

```
/**
 * 获取spring容器
 * 当一个类实现了这个接口ApplicationContextAware之后，这个类就可以方便获得ApplicationContext中的所有bean。
 * 换句话说，这个类可以直接获取spring配置文件中所有有引用到的bean对象
 * 前提条件需作为一个普通的bean在spring的配置文件中进行注册
 */
@Component
public class SpringCtxUtils implements ApplicationContextAware {

    private static ApplicationContext applicationContext;

    @Override
    public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
        SpringCtxUtils.applicationContext = applicationContext;
    }


    public static <T> T getBean(Class<T> type) {
        try {
            return applicationContext.getBean(type);
        } catch (NoUniqueBeanDefinitionException e) {   //出现多个，选第一个
```

```
            String beanName =
applicationContext.getBeanNamesForType(type)[0];
            return applicationContext.getBean(beanName, type);
        }
    }

    public static <T> T getBean(String beanName, Class<T> type) {
        return applicationContext.getBean(beanName, type);
    }
}


@OnOpen
    public void onOpen(Session session) throws Exception{
            log.info("连接.....进行登录校验");
            // 初始化成员
            this.session=session;

this.redisCache=SpringCtxUtils.getBean(RedisCache.class);

this.messageService=SpringCtxUtils.getBean(MessageService.class);

this.chatStateService=SpringCtxUtils.getBean(ChatStateService.clas
s);
            // 身份校验
            Map<String,Object> sessionMap =
session.getUserProperties();
            if(MapUtil.isEmpty(sessionMap) ||
sessionMap.get("token")==null){
                log.error("WebSocket token 校验失败");
                session.close(new
CloseReason(CloseReason.CloseCodes.VIOLATED_POLICY,"token校验失
败"));
                return;
            }
            JWT jwt =
JWTUtil.parseToken(sessionMap.get("token").toString());
            String uuid = (String) jwt.getPayload(LOGIN_USER_KEY);
            this.loginUser =
redisCache.getCache(RedisConstants.USER_LOGIN_KEY + uuid);
            if(loginUser==null){
                log.error("WebSocket token 校验失败");
                session.close(new
CloseReason(CloseReason.CloseCodes.VIOLATED_POLICY,"token校验失
败"));
            }
            ......剩下的逻辑处理......
        }
```

到这里，登录校验方案一算是完成了，但是大伙也明显看的出缺点：**这玩意儿就是个假校验，先连接再校验，校验不通过服务端手动断开连接，虽然你不能说他没用吧，但是确实显得就比较辣鸡**

再者，**我们在conifg配置类中的时候modifyHandshake方法中已经拿到token了对吧，其实直接可以在modifyHandshake方法中就进行了校验了（没去尝试了，有兴趣的可以去试试）**，这样子相对来说比目前这个方法好那么点，起码确实是先校验再连接了。

然后我觉得最不好的就是和Spring管理不在一个环境中，每一个原本能自动注入的都需要调用工具类去获取，这样操作起来不方便，而且没在Spring环境中也不方便管理，所以就开始找方法解决这个问题，方案二应运而生。

方案二（自定义拦截和校验）

利用websocket的配置类，我们不采用之前的直接使用 **`ServerEndpoint` 方式，**而是直接用register注册方法，自定义websocketHandler也就是websocket的处理方法，因为config配置类肯定是spring加载注册管理的，所以我们对应注入注册的websocketHandler自然也就是在Spring的管理当中了，并且在使用的时候同时发现可以注册拦截器

`websocketHandler` 是定义的WebSocket处理器（可以理解为上面的加了@ `ServerEndpoint` 注解的那个websocket主处理类），它处理客户端发送的消息和事件。 `authInterceptor` 是一个拦截器，可以用它在WebSocket连接建立之前进行身份验证。 `"/ws"` 是WebSocket端点的路径，客户端将使用这个路径来与服务器建立WebSocket连接。最后， `"*"` 允许来自任何源的WebSocket连接。

```
@Configuration
public class WebSocketConfig extends
ServerEndpointConfig.Configurator implements WebSocketConfigurer {

    @Autowired
    private WebsocketHandler websocketHandler;
    @Autowired
    private AuthInterceptor authInterceptor;

    @Override
    public void
registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
        registry.addHandler(websocketHandler, "/ws")
                .addInterceptors(authInterceptor)
                .setAllowedOriginPatterns("*")
                ;
    }
}
```

那么这就好办了，全部自定义和自动注入，自然都在Spring的环境中，下面是websocketHandler和authInterceptor类的代码

```
package com.ticknet.LostAndFound.websocket;

import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;
import org.springframework.web.socket.CloseStatus;
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.WebSocketMessage;
import org.springframework.web.socket.WebSocketSession;

/**
 * @Author Created by 黄振伟
 *具体每个方法的实现请看项目代码
 **/
@Component
@Slf4j
public class WebsocketHandler implements WebSocketHandler {
    @Autowired
    private MyWebSocketController myWebSocketController;

    /**
     * 对应原本的 onOpen
```

```
23          * @param session
24          * @throws Exception
25          */
26         @Override
27         public void afterConnectionEstablished(WebSocketSession
    session) throws Exception {
28             myWebSocketController.onOpen(session);
29         }
30
31         /**
32          *相当于原本的 onMessage
33          * @param session
34          * @param message
35          * @throws Exception
36          */
37         @Override
38         public void handleMessage(WebSocketSession session,
    WebSocketMessage<?> message) throws Exception {
39             myWebSocketController.onMessage(session,message);
40         }
41
42         /**
43          * 相当于原本的 onError
44          * @param session
45          * @param exception
46          * @throws Exception
47          */
48         @Override
49         public void handleTransportError(WebSocketSession session,
    Throwable exception) throws Exception {
50             myWebSocketController.onError(session,exception);
51         }
52
53         /**
54          * 对应原本的 onClose
55          * @param session
56          * @param closeStatus
57          * @throws Exception
58          */
59         @Override
60         public void afterConnectionClosed(WebSocketSession session,
    CloseStatus closeStatus) throws Exception {
61
    myWebSocketController.afterConnectionClosed(session,closeStatus);
62         }
63
64
65         @Override
66         public boolean supportsPartialMessages() {
67             return myWebSocketController.supportsPartialMessages();
68         }
69     }
```

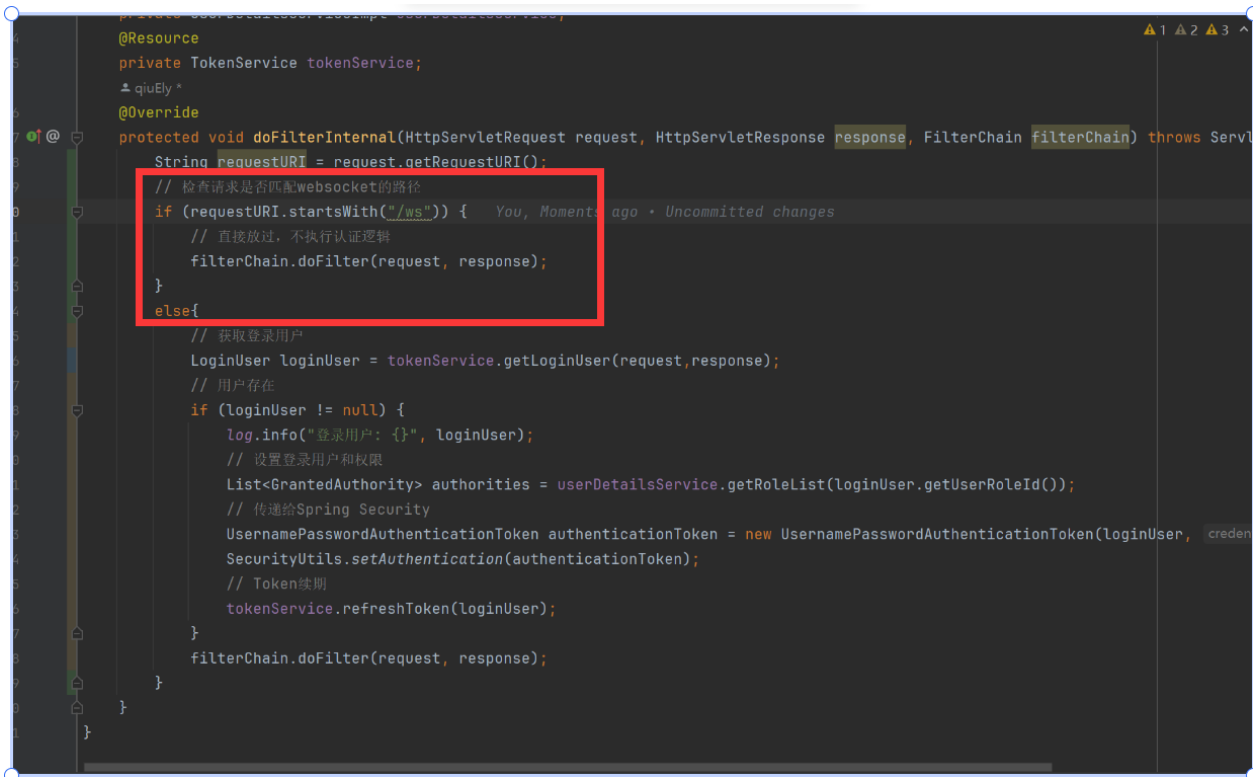
```
1   package com.ticknet.LostAndFound.websocket;
2
3   import cn.hutool.core.util.StrUtil;
4   import cn.hutool.jwt.JWT;
5   import cn.hutool.jwt.JWTUtil;
6   import com.ticknet.LostAndFound.constant.RedisConstants;
```

```java
import com.ticknet.LostAndFound.domain.bo.LoginUser;
import com.ticknet.LostAndFound.service.TokenService;
import com.ticknet.LostAndFound.service.impl.UserDetailsServiceImpl;
import com.ticknet.LostAndFound.util.RedisCache;
import com.ticknet.LostAndFound.util.SecurityUtils;
import jakarta.annotation.Resource;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpHeaders;
import org.springframework.http.server.ServerHttpRequest;
import org.springframework.http.server.ServerHttpResponse;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.stereotype.Component;
import org.springframework.web.socket.WebSocketHandler;
import org.springframework.web.socket.server.HandshakeInterceptor;
import org.springframework.web.socket.server.support.HttpSessionHandshakeInterceptor;

import java.util.List;
import java.util.Map;

/**
 * @Author Created by 黄振伟
 **/
@Component
@Slf4j
public class AuthInterceptor extends HttpSessionHandshakeInterceptor implements HandshakeInterceptor {

    private final static String LOGIN_USER_KEY = "login_user_key";

    @Resource
    private RedisCache redisCache;

    @Resource
    private UserDetailsServiceImpl userDetailsService;

    @Resource
    private TokenService tokenService;

// beforeHandshake方法是建立连接之前会执行的方法  这里直接照搬左伟写的身份校验了
    @Override
    public boolean beforeHandshake(ServerHttpRequest request, ServerHttpResponse response, WebSocketHandler wsHandler, Map<String, Object> attributes) throws Exception {
        HttpHeaders headers = request.getHeaders();
        String token = headers.getFirst("Authorization");
        if(StrUtil.isEmpty(token) || StrUtil.isBlank(token)){
            return false;
        }
        String uuid = getLoginKey(token);
         LoginUser loginUser = redisCache.getCache(RedisConstants.USER_LOGIN_KEY + uuid);
        if (loginUser != null) {
            log.info("登录用户: {}", loginUser);
            // 设置登录用户和权限
```

```
58              List<GrantedAuthority> authorities =
userDetailsService.getRoleList(loginUser.getUserRoleId());
59              // 传递给Spring Security
60              UsernamePasswordAuthenticationToken
authenticationToken = new
UsernamePasswordAuthenticationToken(loginUser, null, authorities);
61              SecurityUtils.setAuthentication(authenticationToken);
62              // Token续期
63              tokenService.refreshToken(loginUser);
64              return true;
65          }else{
66              return super.beforeHandshake(request, response,
wsHandler, attributes);
67          }
68      }
69
70      @Override
71      public void afterHandshake(ServerHttpRequest request,
ServerHttpResponse response, WebSocketHandler wsHandler,
Exception exception) {
72  //          SessionInfoThreadLocal.remove();
73      }
74
75      private String getLoginKey(String token) {
76          JWT jwt = JWTUtil.parseToken(token);
77          return (String) jwt.getPayload(LOGIN_USER_KEY);
78      }
79  }
```

这个方法还存在一个小问题，就是对于/ws这个请求路径，对于http://...../ws 和 ws://...../ws 两种类型的请求，暂时没有找到好方法进行区别，当有一个Controller中存在/ws路径的时候，因为websocket请求是在验证Http请求的时候检测特定的头部信息，如 `Upgrade: websocket` 和 `Connection: Upgrade` 。这些头部信息告诉服务器客户端希望将当前的HTTP连接升级为WebSocket连接，所以一开始（我感觉）不可避免的走Http请求的相关操作，等到服务确认能要升级为WebSocket之后才会使用WebSocket，经过测试证明，发送http://...../ws 和 ws://...../ws都会走到定义到的Controller中而不是WebSocket请求中，所以暂时不知道解决方法（一般来说不会撞的，暂时来看问题不大）

需要注意的是，还是那句话，无论如何都会走原本定义的token校验，所以在原本的token拦截器中，需要对此进行放行或者config中放开，不放开经过测试也行不通（我的理解是：http升级为websocket是在security的过滤器链之后执行的，所以这边必须放行才能保证走完过滤器链不会被拦截，走完过滤器链后才会被升级为ws请求，然后就会跳入上面自定义的AuthInterceptor中 实际上也就是会走两边拦截 http的时候走security的拦截 走完后变成websocket走自定义的AuthInterceptor）



方案三（先下手为强）

说了这么多遍的WebSocket是在Http请求的基础上建立的，所以其实我们最直接的方法就是在原本定义登录校验的基础上进行判断即可，也就是还没升级为websocket请求的时候处于http请求阶段就进行校验，也就是上图的方法中。（之前也写了一部分但是删了，就写个伪代码意思意思一下）

实质上方案三和方案二是差不多的思路，只是方案二将ws的验证分离出去了，没和http的放在一起，我个人喜欢分开一点，后续复习和拓展都好一些。

```
if(是ws请求){
    走ws请求的逻辑
    ws逻辑和http逻辑就唯一的不同就是token拿取地点不同而已
}
```

换header头拿token导致连接自动断开问题

上述的案例代码中，我们都是直接拿的"Authorization"请求头中的信息，但一开始就说到了，这个方案在正式开发中，前端那边是做不到的，所以我们需要从**Sec-WebSocket-Protocol**中拿token

```
String token = headers.getFirst("Authorization");
```

但是当你把"Authorization"换成“Sec-WebSocket-Protocol”，好家伙，立马报错哈哈哈哈，你会发现连接的上但是里面给断了，而且会触发@onClose，直接爽歪歪。

这个是因为WebSocket握手阶段出错：发送了非空“Sec-WebSocket-Protocol”请求头但是响应中没有此字段，在握手请求中，客户端可以通过 `Sec-WebSocket-Protocol` 头来指定期望的子协议。这个头是可选的，但如果客户端发送了这个头，服务器在成功完成握手时应该在响应中也包含这个头，用以确认选择的子协议，我们没有在响应体response中添加这个信息，~~所以理解为你是当websocket接收的，当http返回的，http返回自然不是长连接所以就断开了（个人认为）~~（2024.7.23修改 修改自黄振伟 修改后）websocket是基于HTTP请求的，我们完全可以理解就是建立这个WebSocket请求之前是建立了一个HTTP请求，在这个HTTP请求的基础上进行一次请求-响应即可将HTTP请求协议升级为WebSocket请求，基于已有的HTTP请求，发送一次协议升级的请求但它包含了特殊的头部，如 `Upgrade: websocket` 和 `Connection: Upgrade` ，这些头部表明客户端希望将当前的HTTP连接升级为WebSocket连接，然后

> 这个是因为WebSocket握手阶段出错：…

振伟 黄振伟 2024年7月23日

基于左边的描述，还有些不全，客户端那边可以设置对响应体的逻辑策略，也就是说客户端可以忽略这个 Sec-WebSocket-Protocol 头的回显。因为开发的时候我们都是用的 APIFOX 进行的接口测试，所以对于 APIFOX 而言估计是需要回显的，才会出现左边的相关记录。

客户端的行为取决于其实现和配置。以下是可能的情况：
客户端忽略缺失的头部：如果客户端实现不要求服务器回显 Sec-WebSocket-Protocol 头部，那么即使头部缺失，客户端也可能继续建立 WebSocket 连接并开始通信。
客户端要求头部回显：如果客户

请求到达服务端，服务端接收到后，觉得没问题就会发送响应返回一个状态码为 `101 Switching Protocols` 的HTTP响应。这个响应确认了握手成功，并且告知客户端接下来将使用WebSocket协议进行通信，所以其实在这个时候就在服务端已经确认了WebSocket连接，所以自然就会去触发我们的设置的@onopne注解对应的onopen方法，但是我们的HTTP响应中没有包含对应的 `Sec-WebSocket-Protocol` 导致客户端不同意/不允许这个请求的升级，所以就会进行一定的策略重试/断开，所以导致我们这里后来连接成功后里面就触发了@onclose方法 。

所以只需要在返回的时候添加Header头信息即可，确保客户端成功响应这个升级。

```
1   response.getHeaders().set("Sec-WebSocket-Protocol",token);
```

前端寻主寻物新增导致token不起作用And为何参数不对返回401（黄振伟+郭康乐）

已解决--前端Token对应数据类型不对，导致数据无法解析

端实现期望服务器回显 Sec-WebSocket-Protocol 头部作为握手的一部分，那么客户端可能会认为握手失败，并采取以下行动之一：立即断开连接：客户端可能会认为服务器没有正确处理握手请求，因此它会断开连接。发送错误：客户端可能会在内部记录错误，但不立即断开连接，而是等待进一步的指令或超时。尝试重新握手：在某些情况下，客户端可能会尝试重新发起握手请求，希望这次能收到正确的响应。
客户端逻辑处理：客户端可能会有特定的逻辑来处理这种情况，例如，可能会检查其他头部或响应体来确定是否继续建立连接。
在 WebSocket 协议中，Sec-WebSocket-Protocol 头是可选的，用于客户端和服务器之间协商子协议。如果客户端在请求中发送了这个头部，它期望服务器在响应中也包含这个头部来确认所使用的子协议。如果服务器没有回显这个头部，客户端可能会认为服务器不支持请求的子协议，或者服务器没有正确处理握手请求。
展开