

- 一、实现前端消息推送效果
- 二、基于微服务体系进行扩展
 - 1、基于微服务体系增加IM服务水平扩展功能
 - 2、基于Dubbo对外暴露微服务

三、直播间集成IM服务

阶段目标：将IM后端服务接入到仿抖音直播项目中

一、实现前端消息推送效果

核心问题：何时建立链接？何时断开链接？

进入不同直播间需要复用socket链接。所以，不能在进入直播间的时候建立websocket链接。而应该将socket链接缓存起来，尽量复用。进入直播页面时如果没有建立链接，就建立新链接。

websocket链接不能退出直播间时关闭，而应该等待超时自动关闭。

业务实现注意点：

1、建立websocket.js组件，单独负责维护websocket链接。

处理UserId：如果用户已经登录，返回登录用户ID。如果没有登录，随机生成一个字符串型的UserId

收到消息后，通过mitt的事件机制，将消息通知前端VUE。

```
npm install --save mitt
```

Session超时处理：在直播间页面，定时往后端发送心跳消息，进行保活。

2、进入直播间后，要拿到Websocket链接，并且立即发送一条进入房间的消息。这个时候要注意，前端建立socket链接也是需要时间的，如果拿到socket后，立即发送，此时链接还没建立完成，发消息是发不成功的。所以要做个延迟任务，等socket链接建立成功后再次发送进入房间的消息。

很多大型平台会在这个延迟时间内，通过HTTP请求去获取历史聊天信息、以及直播间公告等，进一步提升用户体验。

后端要做处理，之前设定未登录用户不允许发送消息。要改为只是不允许发送聊天消息，其他消息允许发送。

3、增加送礼物消息类型。

4、用户体系的BUG：用户登录状态只保存到Pinia，页面只要一刷新，或者弹出一个新的页面，登录状态就没了。用户登录状态还是要写入到cookie中。

有个Pinia插件据说可以做这个事情。 `npm install pinia-plugin-storage` 他的做法是把持久化数据保存到localStorage中。

```
# 在加载pinia时使用这个组件
import piniaPluginStorage from "pinia-plugin-storage"

const store = createPinia()
store.use(piniaPluginStorage)
app.use(store)

# 创建store时声明storage
import { defineStore } from 'pinia'

export const useMyStore = defineStore({
  id: 'myStore',
  state: () => ({
    data: null
  }),
  storage: {
    enabled: true //默认是false。
  }
})

# 具体配置项详见 https://www.npmjs.com/package/pinia-plugin-storage
# 使用这个插件时要注意，每个浏览器支持的localStorage大小是有限的，注意不要存入太多信息。而且要注意客户端数据的安全性。
```

二、基于微服务体系进行扩展

1、基于微服务体系增加IM服务水平扩展功能

问题1：前端直接指定IM服务器的地址，不灵活。IM服务也不具备高可用

解决思路：将IM服务注册到Nacos。在api模块中增加一个接口/im/getIMServer，读取nacos上IM服务的注册地址列表，然后通过负载均衡算法选择一个地址返回给前端。

前端不再直接指定IM服务地址，而是从API模块的接口中获取一个IM服务地址，建立链接。这样前端就不再需要知道IM服务的地址，依然只需要与API模块打交道。

如果所有IM服务都没有注册到Nacos上，这种情况会带来很多问题，不太好兼容。但是这种情况，在大型线上项目中，完全可以通过运维手段避免，不一定非要从技术上解决。

问题2：IM服务的接口地址是开放的，任何人都可以建立websocket，不安全

解决思路：在IM的Websocket握手协议中增加身份认证功能。-- 参考用户体系中的身份认证机制。

在API模块的getWsServer接口中，除了返回一个IM服务的Websocket服务端地址外，还生成一个随机Token，返回给前端。并以Token为Key，存入到Redis中。Value可以用UserId。

在前端与IM模块建立握手协议时，补充传入Token。IM模块去Redis中验证Token是否合法。合法则允许建立链接，不合法就拒绝链接。

核心代码：

```

@RestController
@RequestMapping("/im")
public class IMController {

    @Value("${tllive.im.instance}")
    private String imInstance;
    @Resource
    private DiscoveryClient discoveryClient;

    @Resource
    private IMTokenService imTokenService;

    /**
     * 获取IM服务器地址
     * @return
     */
    @PostMapping("/getIMServer")
    public WebResDTO getIMServer(String userId){
        List<ServiceInstance> instances = discoveryClient.getInstances(imInstance);
        if(instances.size() == 0){
            return new WebResDTO(WebResDTO.ERROR_CODE,"IM服务未启动");
        }else{
            // items.get(ThreadLocalRandom.current().nextInt(items.size()))
            ServiceInstance instanceToChoose = instances.get(ThreadLocalRandom.current().n
extInt(instances.size()));
            //ws://localhost:8989/chat/1
            var instanceUrl = "ws://" + instanceToChoose.getHost() + ":" + instanceToChoose.getP
ort() + "/chat/" + userId;
            var imToken = imTokenService.generateIMToken(userId);
            Map<String ,Object > res = new HashMap<>();
            res.put("imToken",imToken);
            res.put("url",instanceUrl);
            return new WebResDTO(WebResDTO.SUCCESS_CODE,res);
        }
    }
}

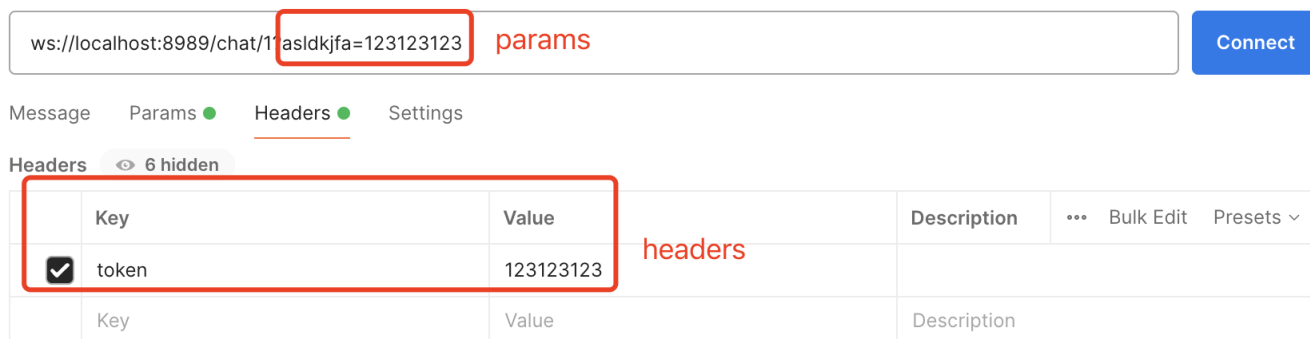
```

实现注意点：

思路清晰之后，还有一些实现上的问题需要解决。如何将这个imToken在websocket链接中传递呢？这就需要对websocket加深了解。

1、websocekt如何传递参数

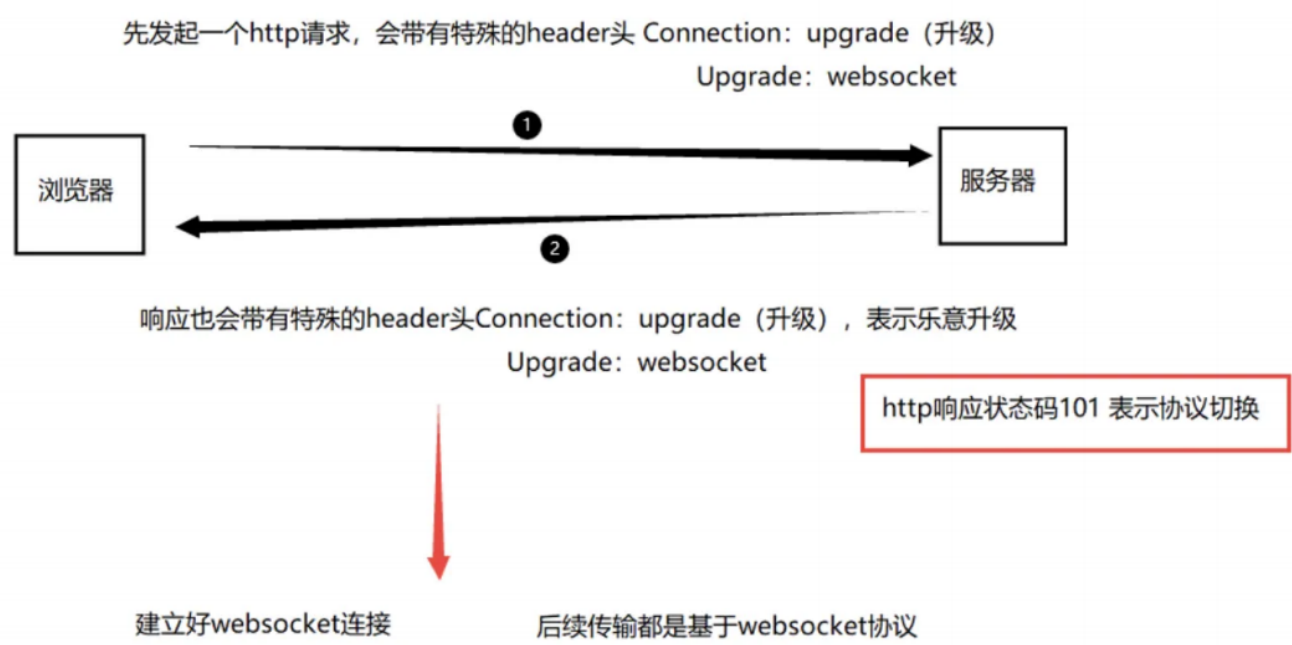
websocket协议传递参数有几种方式，postman中给出了很好的参考



第一种Params传参，在请求路径后面加上? 进行传参。

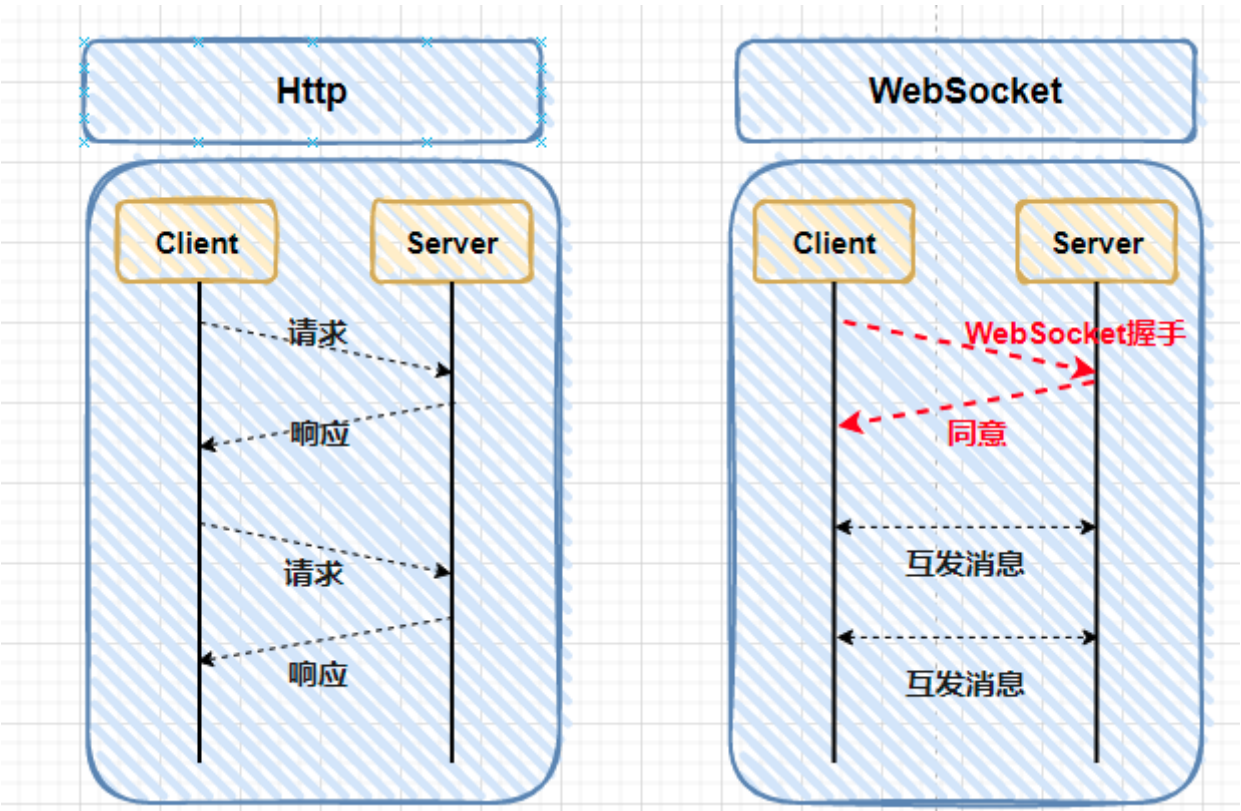
第二种Headers传参，在请求体中添加额外的请求参数。

有没有觉得这两种方式，其实跟HTTP的GET请求是一样的？(无法像POST请求一样添加请求体)其本质还是因为WebSocket的握手请求是通过HTTP协议进行的。只是在建立完连接后，升级成为WS协议。



2、websocket如何验证参数

前端传参后，后端要如何验证这些参数呢？核心思路还是要回到WebSocket的流程当中。



websocket协议本身对消息内容没有做任何限制，也就是说，建立链接后，再要去解析参数，就只能从消息体上下手了，而无法从这些补充的参数下手。因此，在后端要接收自定义参数，还是只能从握手阶段下手。

在SpringBoot集成的Websocket框架中，并没有提供注入自定义握手协议的方式，但是，我们可以从Web请求入手，设定一个Filter对HTTP请求进行过滤。在其中添加自定义的逻辑。

```
@Component
public class WebSocketFilter implements Filter {
    private Logger logger = LoggerFactory.getLogger(WebSocketFilter.class);

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
        FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;
        //解析websocket的header参数
        String token = httpServletRequest.getHeader("token");
        System.out.println("token:"+token);
        //解析websocket的params参数
        Map<String, String[]> parameterMap = httpServletRequest.getParameterMap();
        System.out.println("====parameterMap====");
        parameterMap.forEach((k,v)->{
            System.out.println(k+":"+v[0]);
        });
        System.out.println("====parameterMap====");
        //往下传递请求
        filterChain.doFilter(servletRequest, servletResponse);
    }
}
```

3、通过websocket前端，如何传递参数

在前面总结了websocket两种传递参数的方式。回到我们这个IM系统，应该要用哪种方式来传递Token呢？当然是要选择通过header传参。因为在我们的方案当中，Token是一个很重要的安全参数。如果通过params传参，那就相当于别人同样只要获取到了Websocket的服务地址，就可以建立websocket连接了。Token成了一个掩耳盗铃的参数。而Token放到Header中传参，相比Params就要安全很多。

接下来回到我们项目当中。前端通过API模块同时获取到WebSocket链接地址和Token之后，在建立WebSocket链接的时候，就要将Token作为一个Header参数，往后端传。这时候会出现一个很严重的问题：
**在HTML中实现的websocket函数中，并没有设定Header参数的方法。 ** 如何传递Headers参数呢？其实办法还是只能从websocket的协议入手。还记得在SpringBoot官方给大家看过的Websocket介绍吗？

A WebSocket interaction begins with an HTTP request that uses the HTTP `Upgrade` header to upgrade or, in this case, to switch to the WebSocket protocol. The following example shows such an interaction:

```
GET /spring-websocket-portfolio/portfolio HTTP/1.1
Host: localhost:8080
Upgrade: websocket ❶
Connection: Upgrade ❷
Sec-WebSocket-Key: Jlc9l9TMkwGhHFD2qnFHltg==
Sec-WebSocket-Protocol: v10.stomp, v11.stomp
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
```

- ❶ The Upgrade header.
- ❷ Using the Upgrade connection.

自定义通信协议

Instead of the usual 200 status code, a server with WebSocket support returns output similar to the following:

```
HTTP/1.1 101 Switching Protocols ❶
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: 1qVdfYHU9hP0L4JYNXF623Gzn0=
Sec-WebSocket-Protocol: v10.stomp
```

- ❶ Protocol switch

After a successful handshake, the TCP socket underlying the HTTP upgrade request remains open for both the client and the server to continue to send and receive messages.

而实际上，前端的WebSocket函数，还可以传递第二个参数。而第二个参数，就是这个Protocol协议。

```
declare var WebSocket: {
  prototype: WebSocket;
  new(url: string | URL, protocols?: string | string[]): WebSocket;
  readonly CONNECTING: 0;
  readonly OPEN: 1;
  readonly CLOSING: 2;
  readonly CLOSED: 3;
};
```

因此，在前端可以这样创建WebSocket协议

```
socket = new WebSocket(websocketUrl, "123");
```

这个123就相当于是一种自定义协议。

在后端，可以通过请求头获取到自定义协议的值。

```

@Component
public class WebSocketFilter implements Filter {
    private Logger logger = LoggerFactory.getLogger(WebSocketFilter.class);

    @Override
    public void doFilter(ServletRequest servletRequest, ServletResponse servletResponse,
        FilterChain filterChain) throws IOException, ServletException {
        HttpServletRequest httpServletRequest = (HttpServletRequest) servletRequest;
        HttpServletResponse httpServletResponse = (HttpServletResponse) servletResponse;
        //解析websocket的自定义协议
        String token = httpServletRequest.getHeader("Sec-WebSocket-Protocol");
        System.out.println("token:"+token);
        //解析websocket中?传递的参数
        Map<String, String[]> parameterMap = httpServletRequest.getParameterMap();
        System.out.println("====parameterMap====");
        parameterMap.forEach((k,v)->{
            System.out.println(k+":"+v[0]);
        });
        System.out.println("====parameterMap====");

        if(StringUtils.hasText(token)){
            //设置自定义协议后，需要将协议往请求端传递，否则websocket协议会报错，无法建立websocket连接
            httpServletResponse.setHeader("Sec-WebSocket-Protocol",token);
            filterChain.doFilter(servletRequest,httpServletResponse);
        }else{
            filterChain.doFilter(servletRequest,servletResponse);
        }
    }
}

```

好了，接下来的事情就简单了。添加上业务逻辑，去Redis中验证Token，就可以完成了。这样Socket链接需要从API模块获取和参数，才可以建立websocket链接。如果不经API模块，即使别人通过网络爬虫抓取到了websocket的地址，也没有用了。

2、基于Dubbo对外暴露微服务

问题：当前IM服务只能接收从直播间页面传递过来的消息，无法接收其他模块传递的消息。而IM业务场景，是需要直播间能够接收外部系统的消息的，比如平台通告、一些活动消息等。

思路：在IM模块中，通过Dubbo框架对微服务体系暴露一个服务，实现从外部往直播间发送消息的功能。