

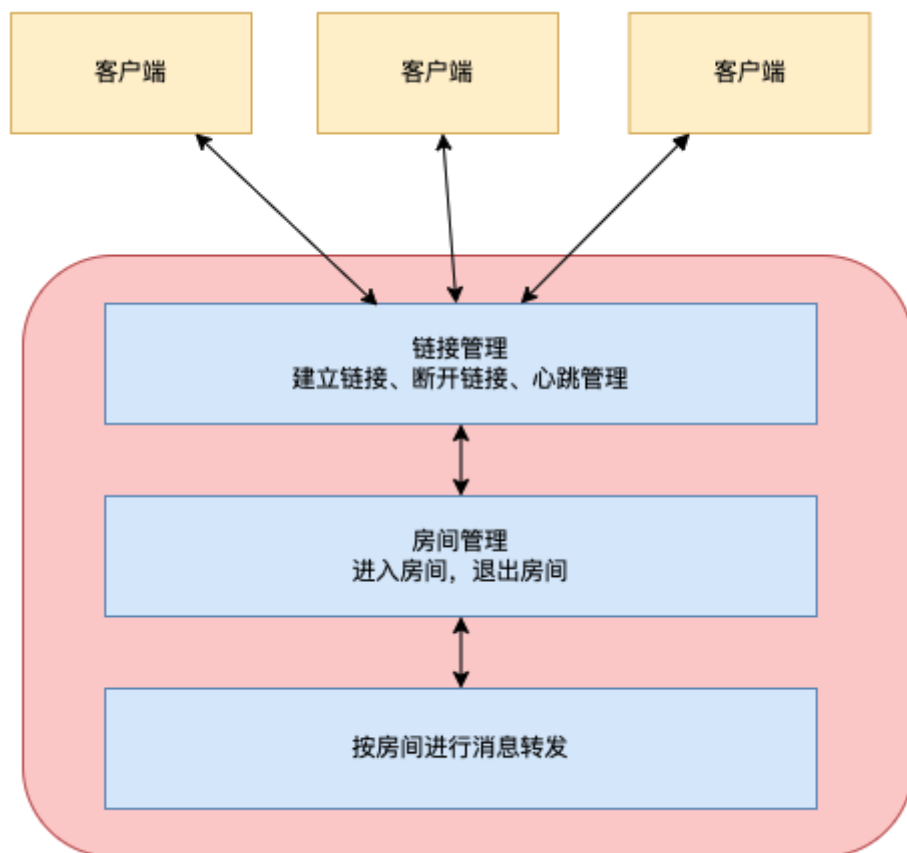
- 一、WebSocket方案设计与技术选型
 - 直播场景的IM系统应该如何设计
- 二、基于Springboot快速实现WebSocket
 - 服务端：基于Springboot实现
 - 客户端：html浏览器支持
- 三、增加消息转发功能
- 四、增加登录用户识别功能
- 五、增加基于房间的消息路由功能
- 六、增加心跳管理功能

二、websocket实战

阶段目标：完成WebSocket基础的后端技术方案演练

一、WebSocket方案设计与技术选型

直播场景的IM系统应该如何设计



二、基于Springboot快速实现WebSocket

服务端：基于Springboot实现

pom.xml核心依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-websocket</artifactId>
</dependency>
<!-- JSON解析 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>fastjson</artifactId>
    <version>${fastjson.version}</version>
</dependency>

<fastjson.version>2.0.19</fastjson.version>
```

核心配置类

```
/** 开启WebSocket支持
 * Author: roy
 * Description:
 */
@Configuration
@EnableWebSocket
public class WebSocketConfig {

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

简单配置文件

```
server:
  port: 8989
spring:
  application:
    name: tl-live-im
```

声明Websocket接口

```
@Component
@ServerEndpoint("/chat")
public class ChatWebSocketController {
    Logger logger = LoggerFactory.getLogger(ChatWebSocketController.class);

    @OnOpen
    public void onOpen(Session session){
        logger.info("建立链接"+session.getId());
    }

    @OnClose
    public void onClose(){
        logger.info("关闭链接");
    }

    @OnError
    public void OnError(Session session, Throwable error){
        logger.info("链接出错"+error.getMessage());
    }

    @OnMessage
    public void onMessage(String message, Session session) throws IOException {
        logger.info("收到消息"+message);
        session.getBasicRemote().sendText(message);
    }
}
```

简单实现后，就可以使用Postman之类的工具进行测试

注意：这个Controller如果要注入Spring容器内的组件，是不能使用@Resource或者@Autowired直接注入的。这是因为Websocket是针对每个前端的websocket链接开启一个单独的线程，也就是说ChatWebSocketController是多例的。这与Spring容器要求的单例是冲突的。

如果要注入Spring容器内的组件，可以引入一个单独的工具类进行。

```
@Component
public class SpringContextUtil implements ApplicationContextAware {
    private static ApplicationContext applicationContext;

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

    public ApplicationContext getApplicationContext(){
        return applicationContext;
    }

    public static Object getBean(String beanName){
        return applicationContext.getBean(beanName);
    }

    public static <T> T getBean(Class<T> clazz){
        return (T)applicationContext.getBean(clazz);
    }
}
```

客户端：html浏览器支持

核心js方法

```

var socket;
if(!window.WebSocket){
    alert("浏览器不支持websocket");
}
function connect() {
    var websocketUrl = "ws://localhost:8989/chat";
    var userId = document.getElementById("userId");
    if(userId.value){
        websocketUrl += "?userId="+userId.value;
    }
    socket = new WebSocket(websocketUrl);
    socket.onmessage = function (event) {
        var rt = document.getElementById("responseText");
        rt.value = rt.value + "\n"+event.data;
    }
    socket.onopen=function (event) {
        var rt = document.getElementById("responseText");
        rt.value = "建立WebSocket连接....";
    }
    socket.onclose=function (event) {
        var rt = document.getElementById("responseText");
        rt.value = rt.value + "\n 连接已关闭";
    }
}

function send(message){
    if(!window.socket){
        alert("还没有建立连接。")
        return;
    }
    if(socket.readyState == WebSocket.OPEN){
        socket.send(message)
    }else{
        alert("没有建立连接");
    }
}

function disconnect() {
    if(window.socket){
        socket.close();
        socket = null;
    }
}

```

三、增加消息转发功能

一个Session代表了服务端与客户端建立的一个长连接。将Session保存下来，就可以实现长连接复用，从而实现消息转发。

通过一个全局容器将所有客户端的Session保存下来，就可以实现消息转发。

```

package com.tl.im.server.manager;

import io.micrometer.common.util.StringUtils;
import jakarta.websocket.Session;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.socket.WebSocketSession;

import java.util.List;
import java.util.Map;
import java.util.Optional;
import java.util.concurrent.ConcurrentHashMap;

/**
 * Author: roy
 * Description: 连接管理器
 */
public class ConnectionManager {

    private static final Logger logger = LoggerFactory.getLogger(ConnectionManager.class);

    private static final Map<String, Session> CHANNEL_CONTAINER = new ConcurrentHashMap<>
();

    public static boolean register(String sessionId, Session session){
        Session addedSession = CHANNEL_CONTAINER.putIfAbsent(sessionId, session);
        if(null != addedSession){
            logger.warn("sessionId:{} 已存在, 不允许重复注册", sessionId);
            return false;
        }
        return true;
    }

    public static Optional<Session> getSession(String sessionId){
        if(StringUtils.isBlank(sessionId)){
            return Optional.empty();
        }
        return Optional.ofNullable(CHANNEL_CONTAINER.get(sessionId));
    }

    public static void cancel(String sessionId, Session session){
        Optional<Session> optConn = getSession(sessionId);
        if (optConn.isPresent()) {
            if (optConn.get().getId().equals(session.getId())) {
                CHANNEL_CONTAINER.remove(sessionId);
                logger.debug("清理路由成功,sessionId=>{}", sessionId);
            }
        }
    }

    public static List<Session> getAllSession(){
        return CHANNEL_CONTAINER.values().stream().toList();
    }
}

```

四、增加登录用户识别功能

完成了简单的消息转发功能后，还有几个问题

问题1：同一个客户端可以多次重复建立链接

方案：

建立链接时，前端对登录用户传递UserId，未登录用户，生成一个随机字符串代替UserId。以UserId作为客户端的唯一标识，防止同一个客户端反复建立链接。

后端在建立连接时，以UserId为Key，缓存Session。

问题2：无法对客户端身份进行校验。登录用户才允许发送消息。

方案：

前端做控制，只有登录用户才显示发消息的按钮。--不靠谱

后端建立链接时，如果UserId是数字，认为是登录用户，就给Session记录一个登录的状态，再缓存。未登录用户则不记录状态。

消息转发时，判断当前Session是否登录。只有登录后的用户发送的消息才进行转发。变向阻止未登录用户发送消息。

五、增加基于房间的消息路由功能

实际业务当中，并不是将消息转发给所有客户，而是只转发给当前房间的客户。自然要增加一个房间的集合管理。用一个Map结构的数据 ROOM_CONTAINER 就可以保存房间的关系。

问题1：key是roomId，value是什么呢？List<Session> 还是List<UserId>？

如果Value用Session，意味着用户每次切换房间，都需要调整Session集合，性能比较低。但是Value用List<UserId>，切换房间，只需要调整一个字符串集合，会简单很多。

并且，UserID是一个String，就比较容易后续移动到Redis中进行缓存管理。而Session是无法进行序列化的，自然也就无法传到Redis中保存。

问题2：房间ID是在链接路径上传递过来，还是在消息体上传递过来？

房间ID在链接路径上穿过来，意味着用户每次切换房间，都需要重建Session，这肯定不好。所以房间ID应该在消息体上传递过来。

进一步，这也意味着，消息体就不能再是一个简单的字符串，而应该是一个完整的数据结构。包含房间ID、用户ID，消息体，另外，还应该包含业务类型，区分进入房间、在房间内聊天、送礼物、退出房间等业务动作。

接下来需要定义消息的标准格式。这需要在前端与后端之间达成一致。消息的标准分为两个部分，一是消息的协议格式，二是消息的传递方式。

对于消息的协议格式，定义如下：

协议主体：

```
public class GenericMessage implements Serializable {

    /**
     * 消息类型：
     */
    private Integer type;

    /**
     * 房间ID
     */
    private Long roomId;
    /**
     * 发送消息用户
     */
    private Long fromUserId;
    /**
     * 消息体
     */
    private List<MessageBody> body;
    ...
}
```

协议内容：

```
public class MessageBody implements Serializable {

    private Long msgId;

    private String content;

    private Long toUserId;

    private Long fromUserId;

    private String fromUserName;
    ....
}
```

消息协议格式需要根据业务进行定制。这里定制的消息协议格式不光包含当前业务，还预设了一些其他业务，比如单聊。

对于消息的传递形式，在Web场景，数据的传递形式就可以用JSON格式，虽然空间浪费比较多，但是前后端的兼容性比较好。当然，如果是TCP场景，消息双方的业务处理能力都比较强，可以选择使用更紧凑的二进制协议。

接下来就需要前端将消息整合成JSON串往后端发送。而后端也按照JSON格式进行解析。

测试消息格式:

加入房间

```
{"type": 0, "roomId": 9999, "body": [{"content": "4443434343434"}]}
```

退出房间

```
{"type": 1, "roomId": 9999, "body": [{"content": "4443434343434"}]}
```

聊天

```
{"type": 2, "roomId": 9999, "body": [{"content": "4443434343434"}]}
```

在ConnectionManager中增加房间相关的管理服务

```

public class ConnectionManager {

    private static final Logger logger = LoggerFactory.getLogger(ConnectionManager.class);

    //存放房间与用户关系 key为RoomId, value为Set<UserId>
    private static final Map<String, Set<String>> ROOM_CONTAINER = new ConcurrentHashMap<>
();

    /**
     * 进入房间
     *
     * @param roomId 房间ID
     * @param userId 用户ID
     */
    public static void joinRoom(String roomId,String userId){
        ROOM_CONTAINER.computeIfAbsent(roomId, v -> new ConcurrentSkipListSet<>
()).add(userId);
    }

    /**
     * 退出房间
     *
     * @param roomId 房间ID
     * @param userId 用户ID
     */
    public static void exitRoom(String roomId,String userId){
        Set<String> roomUsers = ROOM_CONTAINER.get(roomId);
        if (!CollectionUtils.isEmpty(roomUsers)) {
            roomUsers.remove(userId);
        }
    }

    /**
     * 获取房间内所有用户连接 自己必须在这个房间里
     *
     * @param roomId 房间
     * @param userId 消息发送者ID
     * @return List<Session>
     */
    public static List<Session> getRoomAllUserConnect(String roomId, String userId) {
        Set<String> userSet = ROOM_CONTAINER.get(roomId);
        //房间没人, 返回空
        if (userSet == null || userSet.isEmpty()) {
            return Collections.emptyList();
        }
        //自己不在房间里, 不能向房间里的人发消息, 返回空。
        if(!userSet.contains(userId)){
            return Collections.emptyList();
        }

        LinkedList<Session> resultList = new LinkedList<>();
        userSet.forEach(u -> {
            Optional<Session> optSession = getSession(u);
            optSession.ifPresent(resultList::add);
        });
        return resultList;
    }
}

```

另外增加一个消息分发的服务，统一管理消息按房间分发的业务逻辑。MessageTypeDispatchManager核心代码如下：

```

package com.tl.im.server.manager;

import com.tl.im.server.config.IMConstants;
import com.tl.im.server.protocol.GenericMessage;
import com.tl.im.server.service.MessageHandlerService;
import jakarta.annotation.Resource;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.stereotype.Component;

import java.util.concurrent.Executor;

/**
 * Author: roy
 * Description: 消息分发服务
 */
@Component
public class MessageTypeDispatchManager {

    private Logger logger = LoggerFactory.getLogger(MessageTypeDispatchManager.class);

    @Resource
    private MessageHandlerService messageHandlerService;

    @Resource(name = "asyncExecutor")
    private Executor executor;

    /**
     * 根据UserID, 对消息进行路由。
     * @param userId
     * @param message
     */
    public void messageTypeDispatch(String userId, GenericMessage message) {
        if (message.getType() == null) {
            logger.warn("消息格式异常, 直接丢弃");
            return;
        }
        String roomId = message.getRoomId().toString();
        switch (message.getType()) {
            case IMConstants.MESSAGE_TYPE_JOIN_ROOM://加入房间
                ConnectionManager.joinRoom(roomId, userId);
                logger.info("用户=>{},加入房间=>{}", userId, roomId);
                executor.execute(() -> messageHandlerService.sendIndexMessage(userId,
roomId));
                break;
            case IMConstants.MESSAGE_TYPE_EXIT_ROOM://退出房间
                ConnectionManager.exitRoom(roomId, userId);
                logger.info("用户=>{},退出房间=>{}", userId, roomId);
                break;
            case IMConstants.MESSAGE_TYPE_CHAT://聊天
                executor.execute(() -> messageHandlerService.sendRoomChatMessage(userId,
roomId, message));
                logger.info("用户=>{},房间=>{}, 发送消息=>{}", userId, roomId,message);
                break;
            default:
                logger.warn("消息类型异常,message =>{}",message);
        }
    }
}

```

```
}
```

六、增加心跳管理功能

电话打完一定要主动挂掉，websocket链接也一样。这需要客户端和服务端同时努力。客户端需要尽量引导长期无操作的闲置客户进行一些业务操作，判断websocket连接是否还在用。而服务端就需要监听每个websocket链接有没有业务请求。如果长时间没有业务请求，也就理解为没有心跳了，那么就需要主动关闭websocket。



心跳管理是一个业务要求，同样也有很多的实现思路。这里我们选择一个心跳管理器来进行统一管理。

心跳管理器核心逻辑：

```

package com.tl.im.server.manager;

import jakarta.annotation.PreDestroy;
import jakarta.websocket.CloseReason;
import jakarta.websocket.Session;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.ThreadPoolExecutor;
import java.util.concurrent.TimeUnit;

/**
 * Author: roy
 * Description: 链接心跳管理器, 自动关闭空闲链接
 */
@Component
public class ChannelIdleStateManager {

    private Logger logger = LoggerFactory.getLogger(ChannelIdleStateManager.class);
    private static final Map<String, Long> USER_LAST_READ_TIMESTAMP
= new ConcurrentHashMap<>();

    @Value("${tllive.im.heartbeat.timeout.seconds:30}")
    private long READ_TIMEOUT_SECONDS;
    private final ScheduledThreadPoolExecutor scheduledExecutor;

    public ChannelIdleStateManager() {
        int coreCount = Runtime.getRuntime().availableProcessors();
        scheduledExecutor
= new ScheduledThreadPoolExecutor(coreCount, new ThreadPoolExecutor.AbortPolicy());
    }

    public void connect(String userId, Session socketSession) {
        scheduledExecutor.schedule(new ReadTimeOutTask(userId, socketSession),
READ_TIMEOUT_SECONDS, TimeUnit.SECONDS);
        USER_LAST_READ_TIMESTAMP.put(socketSession.getId(), System.nanoTime());
    }

    public void read(Session socketSession){
        if (socketSession.isOpen()){
            USER_LAST_READ_TIMESTAMP.put(socketSession.getId(), System.nanoTime());
            return;
        }
        logger.warn("更新心跳异常, 连接已关闭");
    }

    public void disconnect(Session session) {
        USER_LAST_READ_TIMESTAMP.remove(session.getId());
    }

    @PreDestroy
    public void destroy() {
        scheduledExecutor.shutdown();
    }
}

```

```

private final class ReadTimeOutTask implements Runnable {

    private final String userId;
    private final Session socketSession;

    public ReadTimeOutTask(String id, Session socketSession) {
        this.userId = id;
        this.socketSession = socketSession;
    }

    @Override
    public void run() {
        if (!socketSession.isOpen()) {
            USER_LAST_READ_TIMESTAMP.remove(socketSession.getId());
            ConnectionManager.cancel(userId, socketSession);
            return;
        }
        Long lastReadTime = USER_LAST_READ_TIMESTAMP.get(socketSession.getId());
        if (lastReadTime == null) {
            logger.error("ReadTimeOutTask-中无法找到对应channel的lastReadTime");
            return;
        }
        //按纳秒计算超时时间
        long nextDelay = READ_TIMEOUT_SECONDS*1000000000 - (System.nanoTime() -
lastReadTime);
        if (nextDelay <= 0) {
            try {
                // 空闲超时关闭channel
                if (socketSession.isOpen()) {
                    logger.info("WebSocket通道空闲超时关闭,id:{},socket:{},", userId,
socketSession.getId());
                    socketSession.close(new CloseReason(CloseReason.CloseCodes.NORMAL_
CLOSURE,"心跳超时, 正常关闭链接"));
                }
            } catch (Throwable throwable) {
                logger.error("心跳关闭socket发生异常-task:{},", this, throwable);
            } finally {
                // 确保map中对应的信息一定要被删除, 避免内存泄露
                USER_LAST_READ_TIMESTAMP.remove(socketSession.getId());
                ConnectionManager.cancel(userId, socketSession);
            }
        } else {
            scheduledExecutor.schedule(new ReadTimeOutTask(userId, socketSession),
nextDelay, TimeUnit.NANOSECONDS);
        }
    }
}

```

阶段总结：到这里，这个IM系统已经具备了基础的业务能力。下一步就可以跟项目进行集成了。

当然， 需要注意到， 从一个简单的Websocket到一个IM系统， 对业务的理解能力才是决定程序员技术能力的真正核心。而对于这个IM业务， 随着对业务理解的进一步深入， 也可以变得更为复杂。例如高可用、消息确认与幂等、高并发、安全、限流熔断等， 很多在传统微服务架构下已经解决的问题， 都可以参照微服务的思路逐步添加进来。