Netty

介绍

Netty 是一个高性能、异步事件驱动的网络应用框架，用于快速开发可维护的高性能协议服务器和客户端。

用来网络通信，Netty 的核心优势在于它能够帮助开发者快速高效地构建网络应用，同时保持高性能和高可靠性。它的设计考虑到了易用性和灵活性，允许开发者专注于业务逻辑的实现，而不是底层的网络编程细节。

使用

1.导入依赖

```
<dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>4.1.111.Final</version>
        </dependency>
```

ByteBuf介绍

Netty并没有使用NIO中提供的ByteBuffer来进行数据装载，而是自行定义了一个ByteBuf类。

那么这个类相比NIO中的ByteBuffer有什么不同之处呢?

- 写操作完成后无需进行`flip()`翻转。
- 具有比ByteBuffer更快的响应速度。
- 动态扩容。

其实就是一个缓冲区的操作工具，**在ByteBuf中，有读写两个指针，类似与d组的双指针，一个先写，一个再读，满足读<=写，都是依次往后。**，缓冲池也类似于数组

我们来实际使用一下看看:

写

```
public static void main(String[] args) {
    //创建一个初始容量为10的ByteBuf缓冲区，这里的Unpooled是用于快速生成ByteBuf的工具类
    //至于为啥叫Unpooled是池化的意思，ByteBuf有池化和非池化两种，区别在于对内存的复用，我们之后再讨论，这里是非池化的
   ByteBuf buf= Unpooled.buffer(10);//传入初始化的缓存池大小，单位为字节，类比为字节数组
        System.out.println("初始状态："+ Arrays.toString(buf.array()));
        buf.writeInt(1);//往缓冲池写入1
        buf.writeInt(2);
    System.out.println("写入Int后："+Arrays.toString(buf.array()));


    //写字符串

}
```

因为一个int占四字节，

```
初始状态：[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
初始状态：[0, 0, 0, 1, 0, 0, 0, 2, 0, 0]
```

读

```
//读
int t=buf.readInt();//读指针从0位置开始读一个int数据，然后读指针向后移
        System.out.println(t+" "+ Arrays.toString(buf.array()));
        int t2=buf.readInt();
        System.out.println(t2+" "+ Arrays.toString(buf.array()));
```

```
初始状态：[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
初始状态：[0, 0, 0, 1, 0, 0, 0, 2, 0, 0]
1 [0, 0, 0, 1, 0, 0, 0, 2, 0, 0]
2 [0, 0, 0, 1, 0, 0, 0, 2, 0, 0]
```

```
//丢弃已经读过的位置，比如说我上面只读一次，那么
buf.discardReadBytes();//会重置读指针为0，而且在读指针没舍弃的位置向前覆盖删除的位数k，也就是从0开始复制没舍弃位置之后的k位，同时写指针也往前退k位，就是原来读指针的位置，
//其实就是相当于把删除的位移到数组末尾，继续可以写而已，因为读<=写
```

```
1 [0, 0, 0, 1, 0, 0, 0, 2, 0, 0]
[0, 0, 0, 2, 0, 0, 0, 2, 0, 0]
```

```
buf.clear();     //清空操作，清空之后读写指针都归零，数据还在，但是不可能读到，因为读指针不能超过写指针
    System.out.println("清空之后："+Arrays.toString(buf.array()));
```

```
//获取当前读指针的位置
        //首先进行划分，即把已经读过的忽略，然后再对右边的求首位置
        ByteBuf slice = buf.slice();//从当前读指针划分
        System.out.println(slice.arrayOffset());
```

```
//我们也可以将一个byte[]直接包装进缓冲区（和NIO是一样的）不过写指针的值一开始就跑到最后去了，但是这玩意不是只读的，在清空之后依然可以写，跟nio不一样
    ByteBuf buf = Unpooled.wrappedBuffer("abcdefg".getBytes());
    //除了包装，也可以复制数据，copiedBuffer()会完完整整将数据拷贝到一个新的缓冲区中
```

动态扩容

```
//在生成时指定容量为10
    ByteBuf buf = Unpooled.buffer(10);
    System.out.println(buf.capacity());//获取当前缓冲区的容量
    buf.writeCharSequence("卢本伟牛逼！", StandardCharsets.UTF_8);//往缓冲区写入字符串,设置编码格式
//在写入的东西超出容量就会宽容，从64字节开始，每超都*2，下次超扩成128
    System.out.println(buf.capacity());

//在生成时指定maxCapacity也为10
    ByteBuf buf = Unpooled.buffer(10,10);//指定最大扩容的上限，超出报错
```

缓冲区实现模式

我们接着来看一下缓冲区的三种实现模式：堆缓冲区模式、直接缓冲区模式、复合缓冲区模式。

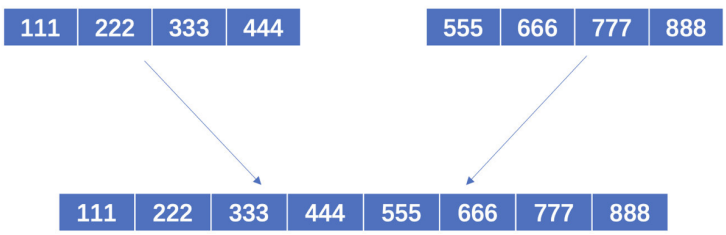
堆缓冲区（数组实现，上面的）和直接缓冲区（堆外内存实现）不用多说，前面我们在NIO中已经了解过了，我们要创建一个直接缓冲区也很简单，直接调用：

```
public static void main(String[] args) {
    ByteBuf buf = Unpooled.directBuffer(10);
    System.out.println(Arrays.toString(buf.array()));
}
```

同样的不能直接拿到数组，因为底层压根不是数组实现的：

```
10
Exception in thread "main" java.lang.IndexOutOfBoundsException Create breakpoint : writerIndex(0) + minWritableBytes(18) exceeds maxCapacity(10): UnpooledByteBufAllocator$Instr
    at io.netty.buffer.AbstractByteBuf.ensureWritable0(AbstractByteBuf.java:294)
    at io.netty.buffer.AbstractByteBuf.setCharSequence0(AbstractByteBuf.java:702)
    at io.netty.buffer.AbstractByteBuf.writeCharSequence(AbstractByteBuf.java:1187)
    at org.example.Main.main(Main.java:13)
```

我们来看看复合模式，复合模式可以任意地拼凑组合其他缓冲区，比如我们可以：



这样，如果我们想要对两个缓冲区组合的内容进行操作，我们就不用再单独创建一个新的缓冲区了，而是直接将其进行拼接操作，相当于是作为多个缓冲区组合的视图。

```java
//创建一个复合缓冲区
CompositeByteBuf buf = Unpooled.compositeBuffer();
buf.addComponent(Unpooled.copiedBuffer("abc".getBytes()));
buf.addComponent(Unpooled.copiedBuffer("def".getBytes()));

for (int i = 0; i < buf.capacity(); i++) {
    //get方法也是使用读指针的，读指针也会往右移
    System.out.println((char) buf.getByte(i));
}
```

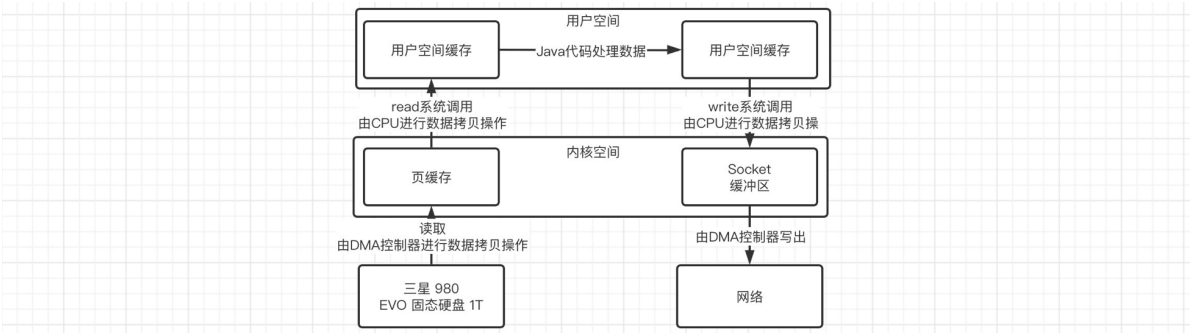
可以看到我们也可以正常操作组合后的缓冲区。

池化缓冲区

```java
//获取池化的工具类
        ByteBufAllocator allocator = PooledByteBufAllocator.DEFAULT;
        //创建池化缓冲区
        ByteBuf buf = allocator.buffer(10);
        buf.writeInt(1);
        buf.release();//释放
        //再获取
        ByteBuf buf2 = allocator.buffer(10);
        System.out.println(buf2 == buf);//结果是true
```

零拷贝

零拷贝是一种I/O操作优化技术，可以快速高效地将数据从文件系统移动到网络接口。传统的拷贝需要2次的cpu拷贝和2次DMA拷贝。我们需要让操作系统帮助我们实现拷贝，首先需要将硬盘的数据拷贝到内核缓存中，再由cpu拷贝到用户进程空间，然后写入Socket缓存，最后才是往网络中发送。

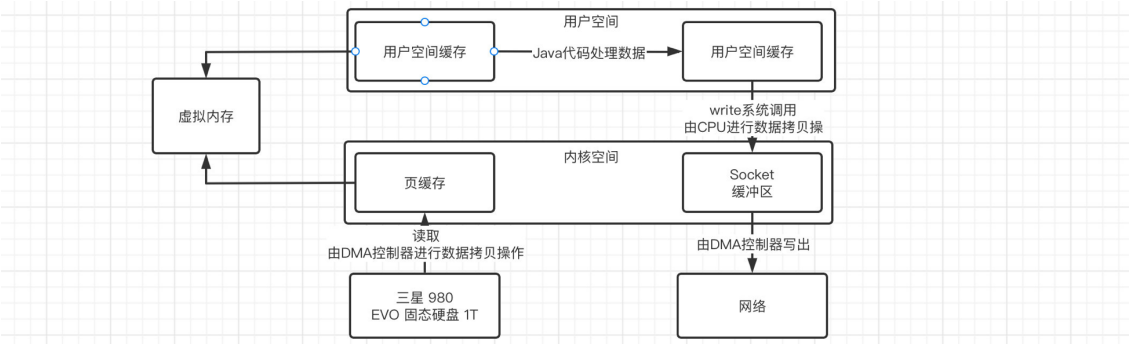


而零拷贝本质上就是减少cpu/DMA的拷贝次数

一般有三种实现方案

1. 使用虚拟内存

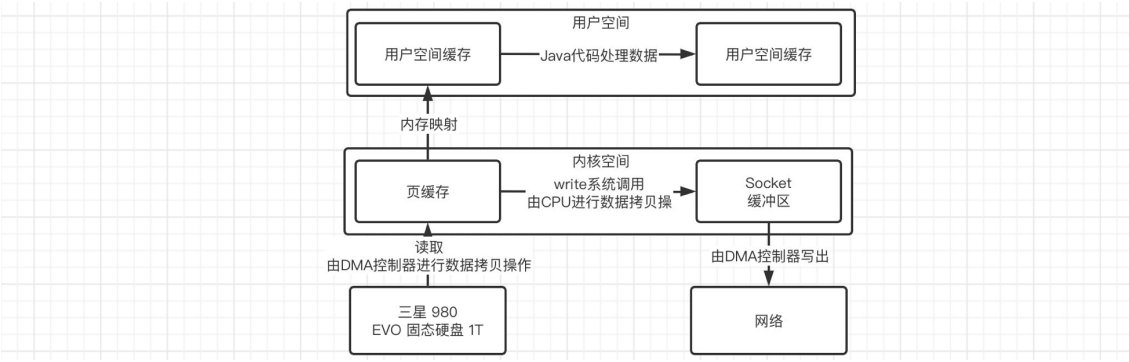
   现在的操作系统基本都是支持虚拟内存的，我们可以让内核空间和用户空间的虚拟地址指向同一个物理地址，这样就相当于是直接共用了这一块区域，也就谈不上拷贝操作了：这样可以省略了一次cpu拷贝



2. 使用mmap/write内存映射

   实际上这种方式就是将内核空间中的缓存直接映射到用户空间缓存，比如NIO中使用的MappedByteBuffer，就是直接作为映射存在，当我们需要将数据发送到Socket缓冲区时，直接在内核空间中进行操作就行了：

   这个减少了两次cpu的拷贝

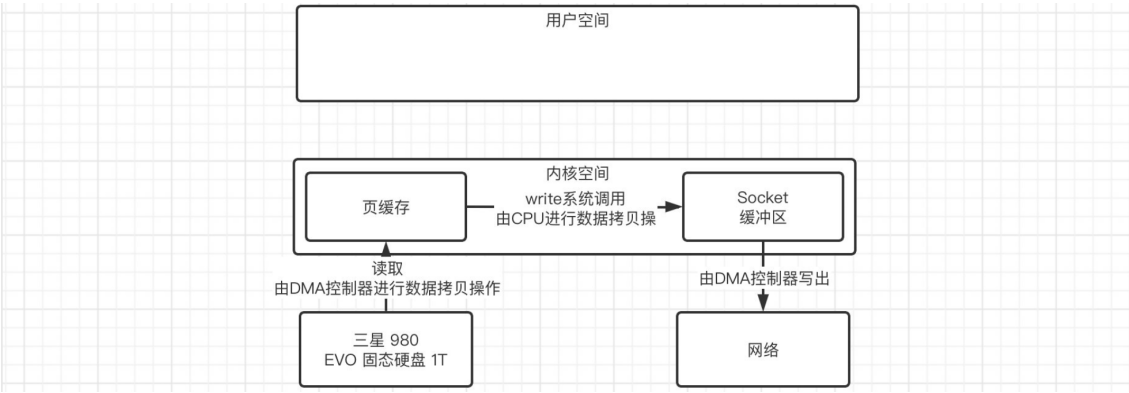


   不过这样还是会出现用户态和内核态的切换，我们得再优化优化。

3. 使用sendfile方式

   在Linux2.1开始，引入了sendfile方式来简化操作，我们可以直接告诉内核要把哪个文件数据拷贝拷贝到Socket缓存上，直接在内核空间中一步到位：

   同样减少了两次cpu拷贝，但同时不需要额外的用户态跟内核态的切换

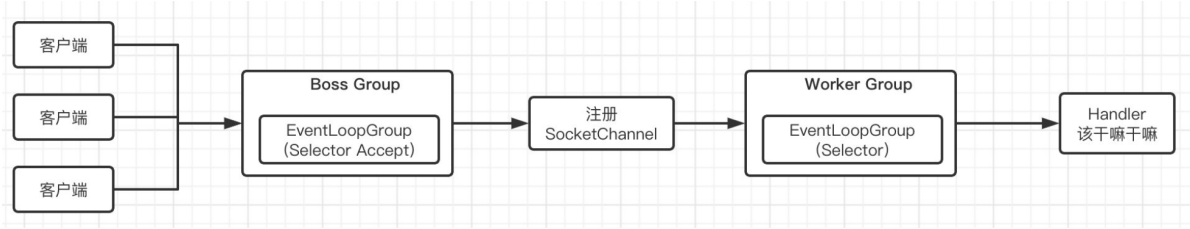


   比如NIO中使用的`transferTo()`方法，就是利用了这种机制来实现零拷贝的。

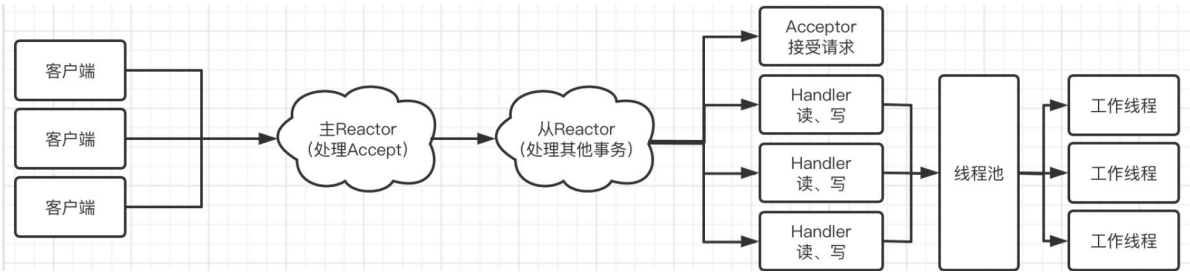
netty的工作模型

Netty是以主从Reactor多线程模型为基础，构建出了一套高效的工作模型。

大致工作模型图如下：



可以看到跟主从Reactor多线程模型非常类似：



所有的客户端需要连接到主Reactor完成Accept操作后，其他的操作由从Reactor去完成，这里也是差不多的思想，但是它进行了一些改进，我们来看一下它的设计：

- Netty 抽象出两组线程池BossGroup和WorkerGroup，BossGroup专门负责接受客户端的连接，WorkerGroup专门负读写，就像我们前面说的主从Reactor一样。
- 无论是BossGroup还是WorkerGroup，都是使用EventLoop（事件循环，很多系统都采用了事件循环机制，比如前端框架Node.js，事件循环顾名思义，就是一个循环，不断地进行事件通知）来进行事件监听的，整个Netty也是使用事件驱动来运作的，比如当客户端已经准备好读写、连接建立时，都会进行事件通知，说白了就像我们之前写NIO多路复用那样，只不过这里换成EventLoop了而已，它已经帮助我们封装好了一些常用操作，而且我们可以自己添加一些额外的任务，如果有多个EventLoop，会存放在EventLoopGroup中，EventLoopGroup就是BossGroup和WorkerGroup的具体实现。
- 在BossGroup之后，会正常将SocketChannel绑定到WorkerGroup中的其中一个EventLoop上，进行后续的读写操作监听。

简单来说：Netty会创建两个进程，一个boss进程，一个work进程，这两个进程都有其线程池，boss进程的一个线程就对应一个客户端，boss进程专门负责接收客户端的连接，比如客户端a发送请求，boss进程的线程b监听这个请求，然后通知给work进程的线程从来识别具体的操作，操作是交给handler，work进程的线程c也要监听boss进程的线程b是否发送了通知。

创建netty服务器

```
 //根据原理，我们需要两个进程boss,work
        //使用NioEventLoopGroup实现类即可
        EventLoopGroup bossGroup = new NioEventLoopGroup(), workerGroup = new
NioEventLoopGroup();
        //创建服务端启动引导类,其实最本质就是类似创建一个ServerSocket，不过比其nb
        ServerBootstrap bootstrap = new ServerBootstrap();
        //配置netty服务器的配置
        bootstrap.group(bossGroup, workerGroup)//指定主从进程
                .channel(NioServerSocketChannel.class)//指定为NIO的
ServerSocketChannel
                .childHandler(new ChannelInitializer<SocketChannel>() {    //注意，
这里的SocketChannel是Netty包下的
                    @Override
                    protected void initChannel(SocketChannel channel) {//激活的流
水线操作
```

```
                        //获取流水线，当我们需要处理客户端的数据时，实际上是像流水线一样在处
理，这个流水线上可以有很多Handler
                        channel.pipeline().addLast(new
ChannelInboundHandlerAdapter(){    //在流水线最后添加一个Handler，这里使用
ChannelInboundHandlerAdapter，
                            @Override
                            public void channelRead(ChannelHandlerContext ctx,
Object msg) {  //ctx是上下文，msg是收到的消息，默认以ByteBuf形式（也可以是其他形式）
                                ByteBuf buf = (ByteBuf) msg;    //类型转换一下

 System.out.println(Thread.currentThread().getName()+" >> 接收到客户端发送的数
据："+buf.toString(StandardCharsets.UTF_8));//转string指定格式
                                //通过上下文可以直接发送数据回去，注意要writeAndFlush才
能让客户端立即收到
                                ctx.writeAndFlush(Unpooled.wrappedBuffer("已收
到！".getBytes()));//返回一个ByteBuf的对象给客户端，不然没有数据返回！！！
                            }
                        });
                    }
                });
        //最后绑定端口，启动
        bootstrap.bind(8080);
```

对应的客户端

```
 //创建一个新的SocketChannel，一会通过通道进行通信,使用的是nio的创建
        try (SocketChannel channel = SocketChannel.open(new
InetSocketAddress("localhost", 8080));
             Scanner scanner = new Scanner(System.in)){
            System.out.println("已连接到服务端！");
            while (true) {    //咱给它套个无限循环，这样就能一直发消息了
                System.out.println("请输入要发送给服务端的内容：");
                String text = scanner.nextLine();
                if(text.isEmpty()) continue;
                //直接向通道中写入数据，真舒服
                channel.write(ByteBuffer.wrap(text.getBytes()));
                System.out.println("已发送！");
                ByteBuffer buffer = ByteBuffer.allocate(128);
                channel.read(buffer);    //直接从通道中读取数据
                buffer.flip();
                System.out.println("收到服务器返回："+new String(buffer.array(), 0,
buffer.remaining()));
            }
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
```

Channel详解

就是通信通道。ChannelInboundHandlerAdapter()是入站的操作

`channel.pipeline().addLast(new ChannelInboundHandlerAdapter()`上面的流水线这个类中有很多操作，比如`channelRead`接收到的执行操作，重写`exceptionCaught`代表发送异常的处理操作，此外还有很多，比如

就是流水线，可以添加addLast多个`new ChannelInboundHandlerAdapter()`处理器，然后重写不同的方法获得不同的操作，接到的操作，发送异常的操作等等

对于没重写的方法，就默认传递给下一个处理器

```
  channel.pipeline()   //直接获取pipeline，然后添加两个Handler，注意顺序
                .addLast(new ChannelInboundHandlerAdapter(){   //第一个用于处理消息
接收
                    @Override//流水线上的处理器1
                    public void channelRead(ChannelHandlerContext ctx, Object
msg) throws Exception {
                        ByteBuf buf = (ByteBuf) msg;
                        System.out.println("接收到客户端发送的数
据："+buf.toString(StandardCharsets.UTF_8));
                        throw new RuntimeException("我是异常");
                    }
                })
      //处理器2
                .addLast(new ChannelInboundHandlerAdapter(){   //第二个用于处理异常
                    @Override
                    public void exceptionCaught(ChannelHandlerContext ctx,
Throwable cause) throws Exception {
                        System.out.println("我是异常处理："+cause);
                    }
                });
```

相反的：ChannelOutboundHandlerAdapter是出站操作

就是有请求来，入站，执行入站的流水线操作，再出站，执行出站的流水线操作

注意：出站流水线要写在入站的上面（第一次往下只会执行入站的操作），然后当入站操作结束就往上找出站的操作，不是往下

或者

```
//ctx.channel().writeAndFlush("啊对对对"); 或是通过Channel进行write也可以
//入站操作最后使用上面这个操作返回数据，
```

```
@Override
protected void initChannel(SocketChannel channel) {
    channel.pipeline()
            .addLast(new ChannelOutboundHandlerAdapter(){
              //注意出栈站操作应该在入站操作的前面，当我们使用ChannelHandlerContext的
write方法时，是从流水线的当前位置倒着往前找下一个ChannelOutboundHandlerAdapter，而我们之前
使用的ChannelInboundHandlerAdapter是从前往后找下一个，如果我们使用的是Channel的write方
法，那么会从整个流水线的最后开始倒着往前找ChannelOutboundHandlerAdapter，一定要注意顺序。
                @Override
                public void write(ChannelHandlerContext ctx, Object msg,
ChannelPromise promise) throws Exception {  //当执行write操作时，会
                    System.out.println(msg);   //write的是啥，这里就是是啥
                    //我们将其转换为ByteBuf，这样才能发送回客户端

 ctx.writeAndFlush(Unpooled.wrappedBuffer(msg.toString().getBytes()));
                }
            })
```

```
            .addLast(new ChannelInboundHandlerAdapter(){
                @Override
                public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
                    ByteBuf buf = (ByteBuf) msg;
                    System.out.println("1接收到客户端发送的数
据："+buf.toString(StandardCharsets.UTF_8));
                    ctx.fireChannelRead(msg);
                }
            })
            .addLast(new ChannelInboundHandlerAdapter(){
                @Override
                public void channelRead(ChannelHandlerContext ctx, Object msg)
throws Exception {
                    ByteBuf buf = (ByteBuf) msg;
                    System.out.println("2接收到客户端发送的数
据："+buf.toString(StandardCharsets.UTF_8));
                    ctx.writeAndFlush("不会吧不会吧，不会还有人都看到这里了还没三连吧");
  //这里可以write任何对象
                    //ctx.channel().writeAndFlush("啊对对对"); 或是通过Channel进行
write也可以
                }
            });
}
```

看这里，直接应用就行

Netty1对1聊天

原理：创建一个简单的 Netty 服务器，处理来自客户端的消息，并将其转发给另一个特定的客户端。假设是根据用户名来实现转发到特定的客户端。这就需要在服务器维护一个map用来映射用户名跟Channel

实现

1.导入netty的依赖

```
<dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>4.1.111.Final</version>
        </dependency>
```

2.创建Netty服务器

2.1编写服务器启动类

```
/**
 * 服务器启动类
 */

public class ChatServer {
    public static void main(String[] args) {
        new ChatServer(8080).start();//启动服务
    }
```

```
    private int port;//启动的端口号
    public ChatServer(int port) {
        this.port = port;
    }
    //配置服务器
    public void start() {
        //根据原理，我们需要两个进程boss,work
        //使用NioEventLoopGroup实现类即可
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        // 创建服务器端的启动对象
        ServerBootstrap bootstrap = new ServerBootstrap();
        //配置netty服务器的配置
        bootstrap.group(bossGroup, workerGroup)//指定主从进程
                .channel(NioServerSocketChannel.class)//指定为NIO的
ServerSocketChannel
                .childHandler(new ChannelInitializer<SocketChannel>(){// 创建通道
初始化对象
                    //注意，这里的SocketChannel是Netty包下的

                    @Override
                    protected void initChannel(SocketChannel socketChannel)
throws Exception {
                        //流水线操作
                        // 添加字符串解码器和编码器,方便消息直接支持String
                        socketChannel.pipeline().addLast(new StringDecoder());
                        socketChannel.pipeline().addLast(new StringEncoder());
                        // 添加自定义的服务器处理器
                        socketChannel.pipeline().addLast(chatServerHandler);
                    }
                });
        //最后绑定端口，启动
        bootstrap.bind(port);

    }
}
```

2.2编写自定义的服务器处理器

在其中维护一个Hashmap映射客户端的用户名跟Channel，要继承SimpleChannelInboundHandler，实现其方法，方法就是当有消息来时就触发

1. `channelActive` **方法**：
    - 当一个新的客户端连接到服务器时的操作
2. `channelRead0` **方法**：
    - 有客户端发送的消息的操作
3. `channelInactive` **方法**：
    - 当一个客户端断开连接时的操作
4. `exceptionCaught` **方法**：
    - 处理异常，打印堆栈信息。
    - 关闭通道，防止资源泄露。
    - 记录异常信息，便于调试和日志记录。

```
/**
 * 自定义的服务器处理类
```

```
 *
 */
public class chatServerHandler extends SimpleChannelInboundHandler<String> {
    // 存储所有客户端的映射表
    private static final HashMap<String, Channel> userChannels = new HashMap<>
();
    // 存储双向map,方便在发送信息时快速知道是哪个用户发送的信息,以及快速删除
    private static final HashMap<Channel,String> channelUsers = new HashMap<>();
    // 存储所有连接的通道,方便广播消息
    private static final ChannelGroup channels = new
DefaultChannelGroup(GlobalEventExecutor.INSTANCE);

    @Override//重写客户端连接时的操作
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        // 当一个新客户端连接时，将其加入通道组
        channels.add(ctx.channel());
        System.out.println("新用户："+ctx.channel().remoteAddress());
        ctx.channel().writeAndFlush("成功连接！");
        //调用read方法，使得channelRead0方法被激活
        ctx.read();
    }
    @Override//重写客户端断开连接的操作
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        // 当一个客户端断开连接时，将其从双向映射表和通道组中移除
        channels.remove(ctx.channel());
        String userName=channelUsers.remove(ctx.channel());
        userChannels.remove(userName);
    }

    @Override//当客户端发送消息的处理操作
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
String s) throws Exception {
        //如果是注册消息就往hashmap中加入
        if(s.startsWith("REGISTER:")){
            String userName = s.split(":")[1].trim();//拿到用户名
            //往双向map中存放
            userChannels.put(userName, channelHandlerContext.channel());
            channelUsers.put(channelHandlerContext.channel(),userName);
            channelHandlerContext.writeAndFlush("注册成功->"+userName);
        }
        else {//正常的转发消息,消息格式 "userName:message"
            //调用方法解析消息的用户名
            String[] split = s.split(":",2);//最多返回两个，即只进行一次切割
            if(split.length == 2){
                String userName = split[0].trim();
                String message = split[1];
                if(userChannels.containsKey(userName)){//在map中存在目标用户就发
                    Channel channel = userChannels.get(userName);
                    //向目标发送消息
                   channel.writeAndFlush("From " +
channelUsers.get(channelHandlerContext.channel()) + ": " + message);
                   //返回响应信息
                    channelHandlerContext.channel().writeAndFlush("发送成功");
                }
                else {
                    channelHandlerContext.writeAndFlush("对方不存在或者已离线");
                }
            }
```

```
            else {
                channelHandlerContext.writeAndFlush("消息格式有问题！");
            }

        }
    }
    @Override//重写发生异常时的exceptionCaught，处理异常
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        // 处理异常，打印堆栈信息并关闭通道
        // 记录异常信息
        System.err.println("Exception caught: " + cause.getMessage());
        cause.printStackTrace();
        ctx.close();
    }
}
```

2.3最后新建一个启动入口creatServer

```
/**
 * 建立netty服务器
 */
public class creatServer {
    public static void main(String[] args) {
        new ChatServer(8080).start();//启动服务
    }
}
```

3.创建客户端

3.1创建客户端启动类

服务器创建的是 `ServerBootstrap bootstrap = new ServerBootstrap();`，对应的客户端创建是 `Bootstrap strap = new Bootstrap();//创建客户端,注意是netty包下的`

跟服务器创建差不多流程

```
/**
 * 客户端启动类
 */
public class ChatClient {
    //启动
    public static void main(String[] args) {
        try {
            new ChatClient("localhost",8080,"user1").start();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    //配置
    private final String host;//服务器ip
    private final int port;//端口
    private final String username;//自己的用户名
```

```
    public ChatClient(String host, int port, String username) {
        this.host = host;
        this.port = port;
        this.username = username;
    }
    public void start() throws IOException, InterruptedException {
        // 创建线程组
        EventLoopGroup group = new NioEventLoopGroup();
            Bootstrap strap = new Bootstrap();//创建客户端,注意是netty包下的
            //简单配置
            strap.group(group) // 设置线程组
                    .channel(NioSocketChannel.class) // 使用 NioSocketChannel 作为
客户端的通道实现,跟服务器保持一致
                    .handler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel socketChannel)
throws Exception {
                            //流水线操作
                            // 添加字符串解码器和编码器,方便消息直接支持String
                            socketChannel.pipeline().addLast(new
StringDecoder());
                            socketChannel.pipeline().addLast(new
StringEncoder());
                            //自定义的处理类，目前只需要打印过来的消息就行
                            socketChannel.pipeline().addLast(new
ChatClienthandler());
                        }
                    });
            //连接服务器,注意添加sync()来保持同步，等连接成功再发送注册信息
            ChannelFuture connect = strap.connect(host, port).sync();
            //发送注册消息
            connect.channel().writeAndFlush("REGISTER:"+username);
            //发送其他消息
            Scanner in=new Scanner(System.in);
            while(true) {
                String msg=in.nextLine();
                if(msg.equals("exit")) {//退出
                    connect.channel().close();
                    break;
                }
                connect.channel().writeAndFlush(msg);
            }
    }
}
```

3.2创建客户端自定义处理器

```
/**
 * 自定义的客户端处理类
 */
public class ChatClienthandler extends SimpleChannelInboundHandler<String> {
    @Override//当有消息来时触发，跟服务器一样
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
String s) {
            System.out.println("服务器消息："+s);
    }
```

```
    @Override//重写发生异常时的exceptionCaught，处理异常
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        // 处理异常，打印堆栈信息并关闭通道
        // 记录异常信息
        System.err.println("Exception caught: " + cause.getMessage());
        cause.printStackTrace();
        ctx.close();
    }
}
```

Netty实现聊天

一对一聊天

原理：创建一个简单的 Netty 服务器，处理来自客户端的消息，并将其转发给另一个特定的客户端。假设是根据用户名来实现转发到特定的客户端。这就需要在服务器维护一个map用来映射用户名跟Channel

多人聊天

原理：在服务器维护两个map，其中`userGroups`存储用户和其加入的群组列表的映射，`groupUsers`存储每个群组的包含用户的映射。当想实现多人聊天，只需要告诉服务器要往用户所在的哪个群组里面发送消息，服务器遍历群组的用户分别发送消息。

消息格式

假设消息的格式为 " 说明:目标:message" ，在**chatServerHandler类中可以修改格式**

那么有以下信息表格

消息格式	说明
REGISTER:username:client_type	注册信息，传递用户名,客户端类型
JOIN:group	加入群组，将该客户端对应的用户加入群组
Private:username:message	一对一的信息发送，中间为目标用户名
Group:groupname:message	多人聊天的信息发送，中间为目标群组

客户端在一开始创建时需要向服务端发送一条消息告知自己的用户名，服务器会将该用户名与该客户端绑定。

具体执行流程

1. **客户端连接到服务器**：
    - 客户端发起连接请求。
    - 服务器接受连接请求，创建一个新的 `Channel` 对象。
    - Netty 调用 `channelActive` 方法，表示新的客户端连接已经建立。
2. **客户端发送注册消息**：
    - 客户端发送一条注册消息，例如 `"REGISTER:user1"`。

- 服务器接收到这条消息，Netty 调用 `channelRead0` 方法来处理这条消息。实现记录通道跟用户名的绑定

3. **客户端发送加入群组消息**：
   - 客户端发送加入群组消息，例如 `"JOIN:world"`。
   - 服务器接收到这些消息，Netty 调用 `channelRead0` 方法来将该客户端对应的用户加入该群组，不存在就创建群组
4. 其他消息

   私发的，服务器就转发给目标用户，群发的服务器就遍历那个群组发送消息

实现

1.导入netty的依赖

```
<dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>4.1.111.Final</version>
        </dependency>
```

2.创建Netty服务器

2.1编写服务器启动类

```
/**
 * 服务器启动类
 */

public class ChatServer {
    public static void main(String[] args) {
        new ChatServer(8080).start();//启动服务
    }
    private final int port;//启动的端口号
    public ChatServer(int port) {
        this.port = port;
    }
    //配置服务器
    public void start() {
        //根据原理，我们需要两个进程boss,work
        //使用NioEventLoopGroup实现类即可
        EventLoopGroup bossGroup = new NioEventLoopGroup();
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        // 创建服务器端的启动对象
        ServerBootstrap bootstrap = new ServerBootstrap();
        //配置netty服务器的配置
        bootstrap.group(bossGroup, workerGroup)//指定主从进程
                .channel(NioServerSocketChannel.class)//指定为NIO的
ServerSocketChannel
                .childHandler(new ChannelInitializer<SocketChannel>(){// 创建通道
初始化对象
                    //注意，这里的SocketChannel是Netty包下的

                    @Override
```

```
                    protected void initChannel(SocketChannel socketChannel)
throws Exception {
                        //流水线操作
                        // 添加字符串解码器和编码器,方便消息直接支持String
                        socketChannel.pipeline().addLast(new StringDecoder());
                        socketChannel.pipeline().addLast(new StringEncoder());
                        //添加
                        // 添加自定义的服务器处理器
                        socketChannel.pipeline().addLast(new
chatServerHandler());
                    }
                })
                .option(ChannelOption.SO_BACKLOG, 128)//设置连接队列的大小
                .childOption(ChannelOption.SO_KEEPALIVE, true);//设置保持连接选项

         //最后绑定端口，启动
        try {
            ChannelFuture bind = bootstrap.bind(port).sync();
            bind.channel().closeFuture().sync();//结束释放资源
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

    }
}
```

2.2编写自定义的服务器处理器

在其中维护一个Hashmap映射客户端的用户名跟Channel，要继承SimpleChannelInboundHandler，实现其方法，方法就是当有消息来时就触发

1. `channelActive` **方法**:
    - 当一个新的客户端连接到服务器时的操作
2. `channelRead0` **方法**:
    - 有客户端发送的消息的操作
3. `channelInactive` **方法**:
    - 当一个客户端断开连接时的操作
4. `exceptionCaught` **方法**:
    - 处理异常，打印堆栈信息。
    - 关闭通道，防止资源泄露。
    - 记录异常信息，便于调试和日志记录。

```
/**
 * 自定义的服务器处理类
 */
public class chatServerHandler extends SimpleChannelInboundHandler<String> {
    // 存储所有客户端的映射表
    private static final HashMap<String, Channel> userChannels = new HashMap<>
();
    // 存储双向map,方便在发送信息时快速知道是哪个用户发送的信息,以及快速删除
    private static final HashMap<Channel,String> channelUsers = new HashMap<>();
    //存储用户跟其加入群组的映射
    private static final HashMap<String, List<String>> userGroups = new
HashMap<>();
```

```
    //存储群组跟其用户的映射
    private static final HashMap< String,List<String>> groupUsers = new
HashMap<>();
    // 存储所有连接的通道,方便广播消息
    private static final ChannelGroup channels = new
DefaultChannelGroup(GlobalEventExecutor.INSTANCE);

    @Override//重写客户端连接时的操作
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        // 当一个新客户端连接时，将其加入通道组
        channels.add(ctx.channel());
        System.out.println("新用户："+ctx.channel().remoteAddress());
        ctx.channel().writeAndFlush("成功连接！");
        //调用read方法，使得channelRead0方法被激活
        ctx.read();
    }
    @Override//重写客户端断开连接的操作
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        // 当一个客户端断开连接时，将其从双向映射表和通道组中移除
        channels.remove(ctx.channel());
        String userName=channelUsers.remove(ctx.channel());
        userChannels.remove(userName);

        //将用户的群组删除以及删除对应群组的这个用户
        List<String>groups=userGroups.remove(userName);
        if(groups!=null){
            for(String group:groups){
                List<String> users=groupUsers.get(group);
                for(String user:users){
                    if(userName.equals(user)){
                        users.remove(user);
                        break;
                    }
                }
            }
        }

    }

    @Override//当客户端发送消息的处理操作
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
String s) throws Exception {
        //如果是注册消息就往hashmap中加入
        if(s.startsWith("REGISTER:")){
            String userName = s.split(":")[1].trim();//拿到用户名
            //往双向map中存放
            userChannels.put(userName, channelHandlerContext.channel());
            channelUsers.put(channelHandlerContext.channel(),userName);
            channelHandlerContext.writeAndFlush("注册成功->"+userName);
        }
        else if(s.startsWith("JOIN:")){//加入群组的消息
            String group = s.split(":")[1].trim();//拿到群组名
            String userName =
channelUsers.get(channelHandlerContext.channel());//拿到用户名
            //加入用户_群组映射
            if(!userGroups.containsKey(userName)){
                List<String> groups=new ArrayList<>();
                groups.add(group);
```

```
                userGroups.put(userName,groups);
            }
            else {
                userGroups.get(userName).add(group);
            }
            //加入群组_用户映射
            if(!groupUsers.containsKey(group)){
                List<String> users=new ArrayList<>();
                users.add(userName);
                groupUsers.put(group,users);
            }
            else {
                groupUsers.get(group).add(userName);
            }
            //返回响应消息
            channelHandlerContext.channel().writeAndFlush("加入"+group+"成功");
        }
        else {//正常的转发消息,根据消息格式不同进行不同操作
            //调用方法解析消息的用户名
            String[] split = s.split(":",3);//最多返回3个，即只进行两次切割
            if(split.length == 3){
                String explain = split[0].trim();//说明
                String target=split[1].trim();//目标
                String message = split[2];//消息
                if(explain.equals("Private")){
                    if(userChannels.containsKey(target)){//在map中存在目标用户就发
                        Channel channel = userChannels.get(target);
                        //向目标发送消息
                        channel.writeAndFlush("From " +
channelUsers.get(channelHandlerContext.channel()) + ": " + message);
                        //返回响应信息

 channelHandlerContext.channel().writeAndFlush("向"+target+"发送消息成功");
                    }
                    else {
                        channelHandlerContext.writeAndFlush("对方不存在或者已离线");
                    }
                }
                else {//群发
                    if(groupUsers.containsKey(target)){//群组要存在
                        List<String> users=groupUsers.get(target);
                        if(!users.isEmpty()){
                            for(String user:users){
                                Channel channel = userChannels.get(user);
                                //向目标发送消息
                                channel.writeAndFlush("From " +
channelUsers.get(channelHandlerContext.channel()) + ": " + message);
                                //返回响应信息

 channelHandlerContext.channel().writeAndFlush("向"+user+"发送消息成功");
                            }
                        }
                    }
                    else {
                        channelHandlerContext.writeAndFlush(target+" 群组不存在");
                    }
                }
```

```
            }
            else {
                channelHandlerContext.writeAndFlush("消息格式有问题！");
            }

        }
    }
    @Override//重写发生异常时的exceptionCaught，处理异常
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)
throws Exception {
        // 处理异常，打印堆栈信息并关闭通道
        // 记录异常信息
        System.err.println("Exception caught: " + cause.getMessage());
        cause.printStackTrace();
        ctx.close();
    }
}
```

2.3最后新建一个启动入口creatServer

```
/**
 * 建立netty服务器
 */
public class creatServer {
    public static void main(String[] args) {
        new ChatServer(8080).start();//启动服务
    }
}
```

3.创建客户端

3.1创建客户端启动类

服务器创建的是 `ServerBootstrap bootstrap = new ServerBootstrap();`，对应的客户端创建是 `Bootstrap strap = new Bootstrap();//创建客户端,注意是netty包下的`

跟服务器创建差不多流程

```
/**
 * 客户端启动类
 */
public class ChatClient {
    //启动
    public static void main(String[] args) {
        try {
            new ChatClient("localhost",8080,"user1").start();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
    //配置
    private final String host;//服务器ip
    private final int port;//端口
```

```
    private final String username;//自己的用户名

    public ChatClient(String host, int port, String username) {
        this.host = host;
        this.port = port;
        this.username = username;
    }
    public void start() throws IOException, InterruptedException {
        // 创建线程组
        EventLoopGroup group = new NioEventLoopGroup();
            Bootstrap strap = new Bootstrap();//创建客户端,注意是netty包下的
            //简单配置
            strap.group(group) // 设置线程组
                    .channel(NioSocketChannel.class) // 使用 NioSocketChannel 作为
客户端的通道实现,跟服务器保持一致
                    .handler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel socketChannel)
throws Exception {
                            //流水线操作
                            // 添加字符串解码器和编码器,方便消息直接支持String
                            socketChannel.pipeline().addLast(new
StringDecoder());
                            socketChannel.pipeline().addLast(new
StringEncoder());
                            //自定义的处理类，目前只需要打印过来的消息就行
                            socketChannel.pipeline().addLast(new
ChatClienthandler());
                        }
                    });
            //连接服务器,注意添加sync()来保持同步，等连接成功再发送注册信息
            ChannelFuture connect = strap.connect(host, port).sync();
            //发送注册消息
            connect.channel().writeAndFlush("REGISTER:"+username);
            //发送其他消息
            Scanner in=new Scanner(System.in);
            while(true) {
                String msg=in.nextLine();
                if(msg.equals("exit")) {//退出
                    connect.channel().close();
                    break;
                }
                connect.channel().writeAndFlush(msg);
            }
    }
}
```

3.2创建客户端自定义处理器

```
/**
 * 自定义的客户端处理类
 */
public class ChatClienthandler extends SimpleChannelInboundHandler<String> {

    @Override//当有消息来时触发，跟服务器一样
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
String s) {
```

```
            System.out.println("服务器消息："+s);
    }

    @Override//重写发生异常时的exceptionCaught，处理异常
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        // 处理异常，打印堆栈信息并关闭通道
        // 记录异常信息
        System.err.println("Exception caught: " + cause.getMessage());
        cause.printStackTrace();
        ctx.close();
    }
}
```

记录用户

为了实现重新连接之后,也能被识别为原来的用户，需要修改服务端的处理类

1.修改断开操作

因为连接时都会发送一条注册信息，这个注册信息存储了的，这就要求在连接断开时，不能删除用户名-通道的映射，但是可以从通道组中删除,因为下次连接的通道肯定不一样了。

并且也不能把用户的群组信息删除，不然每次都要新加群组不符合现实

```
    @Override//重写客户端断开连接的操作
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        /* 当一个客户端断开连接时，为方便下次具有唯一标识的相同用户名登录能被识别为同一个用户
            所以不能将群组信息删除，也不能删除双向map，可以把通道-用户名的删除，因为下一次基本不会是相同的通道连接。
            也将其从通道组删除，
         */
        channels.remove(ctx.channel());
        String userName=channelUsers.remove(ctx.channel());
        userChannels.get(userName).close();//关闭
        channels.writeAndFlush(new TextMessage(userName+"已经下线！"));//广播下线
//
//        //将用户的群组删除以及删除对应群组的这个用户
//        List<String>groups=userGroups.remove(userName);
//        if(groups!=null){
//            for(String group:groups){
//                List<String> users=groupUsers.get(group);
//                for(String user:users){
//                    if(userName.equals(user)){
//                        users.remove(user);
//                        break;
//                    }
//                }
//            }
//        }

    }
```

2.修改连接操作

当有连接来，先判断map中有无这个用户，如果有就是更新map中的通道，没有就加入map

```
//如果是注册消息就往hashmap中加入
        if (s.startsWith("REGISTER:")) {
            String userName = s.split(":")[1].trim();//拿到用户名
            if(userChannels.containsKey(userName)){//用户已经存在就更新双向map中的通道
                Channel channel = userChannels.get(userName);//获取原来的通道
                if(userChannels.get(userName).isActive()){//关闭原来的通道
                    channel.close();
                }
                //更新通道
                userChannels.put(userName, channelHandlerContext.channel());
                channelUsers.put(channelHandlerContext.channel(), userName);
                channelHandlerContext.writeAndFlush(new TextMessage("欢迎上线 !" +
userName));
            }
            else {
                //往双向map中存放
                userChannels.put(userName, channelHandlerContext.channel());
                channelUsers.put(channelHandlerContext.channel(), userName);
                channelHandlerContext.writeAndFlush(new TextMessage("注册成功->" +
userName));
            }
            channels.writeAndFlush(new TextMessage(userName+"已上线! "));//广播上线
            // 再将其加入通道组
            channels.add(channelHandlerContext.channel());

        }
```

Netty实现传输图片

传输图片涉及到二进制数据的处理，而不仅仅是文本消息。这里采用传输byte[]的方式传输图片.**并且假设图片的后缀为jpg**

首先读取用户输入标识图片消息如下

SEND_IMAGE:说明:目标:图片地址	**标识发送图片：私发/群发:target:图片地址**

其中后三部分的消息有以下两种形式

Private:username:image	**一对一的信息发送，中间为目标用户名，后面为图片标识符+图片信息**
Group:groupname:image	多人聊天的信息发送，中间为目标群组

注意：这里会有一个问题，因为前面实现的聊天都是文本，服务端和客户端使用的解码编码器都是关于String的，这里当客户端发送一个`ByteBuf`数据会被解析为String类型，从而解析失败。

解决：使用 自定义的`ByteToMessageDecoder` 和 `MessageToByteEncoder`进行编码和解码

修改编解码格式

1.新建一个通用的消息类Message及其子类

```
/**
 * 通用信息类
 */
public class Message implements Serializable {//实现Serializable 接口表示可序列化
    private static final long serialVersionUID = 1L;//用于在反序列化时验证序列化的对象和当前类是否兼容。
}
```

```
/**
 * 文本消息
 */
@Getter
public class TextMessage extends Message {
    private final String text;
    public TextMessage(String text) {
        this.text = text;
    }
}
```

```
/**
 * 图片消息
 */
@Getter
public class ImageMessage extends Message {
    private final String header;//头部消息
    private final byte[] image;//图片消息
    public  ImageMessage(byte[] image,String header) {
        this.image = image;
        this.header = header;
    }
}
```

2.创建自定义的编码跟解码器`

编码器

```
/**
 * 自定义的编码器,将 Java 对象转换为字节流
 */
public class MyEncoder extends MessageToByteEncoder {
    @Override
    protected void encode(ChannelHandlerContext channelHandlerContext, Object o, ByteBuf byteBuf) throws Exception {
        // 创建一个 ByteArrayOutputStream 来存储对象的字节数据
        try (ByteArrayOutputStream baos = new ByteArrayOutputStream();
             ObjectOutputStream oos = new ObjectOutputStream(baos)) {
            // 将对象写入 ByteArrayOutputStream
            oos.writeObject(o);
            // 将 ByteArrayOutputStream 转换为字节数组
            byte[] bytes = baos.toByteArray();
            // 写入字节数组的长度（4个字节）
```

```
            byteBuf.writeInt(bytes.length);
            // 往ByteBuf写入字节数组
            byteBuf.writeBytes(bytes);
        }
    }
}
```

解码器

```
/**
 * 自定义的解码器，将字节流转化为java对象
 */
public class MyDecoder extends ByteToMessageDecoder {
    @Override
    protected void decode(ChannelHandlerContext channelHandlerContext, ByteBuf
byteBuf, List<Object> list) throws Exception {
        // 检查是否有足够的字节来读取长度字段（4个字节）
        if (byteBuf.readableBytes() < 4) {
            return; // 如果不足4个字节，返回
        }

        // 标记当前的读取位置
        byteBuf.markReaderIndex();
        // 读取长度字段
        int length = byteBuf.readInt();

        // 检查是否有足够的字节来读取整个消息体
        if (byteBuf.readableBytes() < length) {
            // 如果不足，重置读取位置并返回
            byteBuf.resetReaderIndex();
            return;
        }

        // 读取消息体的字节数组
        byte[] data = new byte[length];
        byteBuf.readBytes(data);

        // 将字节数组转换为 Java 对象
        try (ByteArrayInputStream bais = new ByteArrayInputStream(data);
             ObjectInputStream ois = new ObjectInputStream(bais)) {
            Object obj = ois.readObject();
            // 将解码后的对象添加到输出列表中
            list.add(obj);
        }
    }
}
```

3.修改客户端和服务端的编解码器为自定义的

服务器

```
 @Override
                    protected void initChannel(SocketChannel socketChannel)
throws Exception {
                        //流水线操作
                        // 添加自定义的编码器和解码器
                        socketChannel.pipeline().addLast(new MyEncoder());
                        socketChannel.pipeline().addLast(new MyDecoder());
                        //添加
                        // 添加自定义的服务器处理器
                        socketChannel.pipeline().addLast(new
chatServerHandler());
                    }
```

客户端

```
 @Override
                        protected void initChannel(SocketChannel socketChannel)
throws Exception {
                            //流水线操作
                            // 添加自定义的编码器和解码器
                            socketChannel.pipeline().addLast(new MyEncoder());
                            socketChannel.pipeline().addLast(new MyDecoder());
                            //自定义的处理类，目前只需要打印过来的消息就行
                            socketChannel.pipeline().addLast(new
ChatClienthandler());
                        }
```

4.修改发送消息和接收消息的处理

使客户端发送java对象

```
//发送文本信息
//发送注册消息,发送TextMessage类
            connect.channel().writeAndFlush(new
TextMessage("REGISTER:"+userName));
//发送正常文本信息
                    connect.channel().writeAndFlush(new TextMessage(msg));
//发送图片消息
connect.channel().writeAndFlush(new ImageMessage(header,image));
```

服务端解析java对象

```
//修改我们的自定义处理器要处理的类型为Message通用消息类
public class chatServerHandler extends SimpleChannelInboundHandler<Message>

//消息处理
        @Override//当客户端发送消息的处理操作
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
Message msg) throws Exception {
        //处理文本消息
        if (msg instanceof TextMessage) {
            //处理文本消息
            dealText(channelHandlerContext, (TextMessage) msg);
        }
        //处理图片数据
```

```
        else if (msg instanceof ImageMessage) {
          //处理图片消息
            dealImage(channelHandlerContext, (ImageMessage) msg);
        }
        //未知数据类型
        else {
            channelHandlerContext.channel().writeAndFlush(new TextMessage("未知数据类型，暂不支持处理!"));
        }
    }
```

实现传输图片

客户端

发送图片

在用户输入数据加一个分支用来判断是图片消息，**假设格式如下**

```
格式 SEND_IMAGE:说明:目标:图片地址
```

然后对输入切割，读取头部信息（说明:目标），读取图片消息转化为byte[]----->封装到ImageMessage----->发送

```
    //发送其他消息
            Scanner in=new Scanner(System.in);
            while(true) {
                String msg=in.nextLine();
                if(msg.equals("exit")) {//退出
                    connect.channel().close();
                    break;
                }
                else if (msg.startsWith("SEND_IMAGE:")) {//发送的是图片时,格式SEND_IMAGE:说明:目标:图片地址
                   clientUtil.sendImage(msg,connect);//调用方法处理图片消息的发送

                }
                else {//发送正常文本信息
                    connect.channel().writeAndFlush(new TextMessage(msg));
                }
            }
```

对应的图片消息处理方法

```
/**
 * 客户端工具类
 */
public class ClientUtil {
    //处理图片消息发送
    public void sendImage(String msg, ChannelFuture connect){
        String[] parts = msg.split(":", 4);
        if (parts.length == 4) {
            String explain = parts[1].trim(); // 说明
```

```
            String target = parts[2].trim();  // 目标
            String imagePath = parts[3].trim();
            //打开图片文件
            try {
                File file = new File(imagePath);
                FileInputStream fis = new FileInputStream(file);
                byte[] imageBytes = new byte[(int) file.length()];
                fis.read(imageBytes);//将图片存储到byte数组中
                fis.close();
                // 将头部消息整合
                String header=explain+":"+target;
                // 计算图片数据的字节长度
                int imageLength = imageBytes.length;
                //封装到ImageMessage
                ImageMessage image = new ImageMessage(imageBytes, header);
                connect.channel().writeAndFlush(image); // 发送图片数据到服务器
                connect.channel().writeAndFlush(new TextMessage("hello"));
                System.out.println("SEND IMAGE成功");
            } catch (IOException e) {
                System.err.println("Failed to read image: " + e.getMessage());
            }
        } else {
            System.out.println("Invalid send image format. Use 'SEND_IMAGE:说明:目
标:图片地址'.");
        }
    }
}
```

接收图片

在客户端自定义处理器里面写，自定义的处理器的处理类型也要是Message

把图片保存到本地，保存路径为`image/username/当前时间.jpg`

```
/**
 * 自定义的客户端处理类
 */
public class ChatClienthandler extends SimpleChannelInboundHandler<Message> {
    private final String userName;//该客户端的用户名
    public ChatClienthandler(String userName) {
        this.userName = userName;
    }
    @Override//当有消息来时触发，跟服务器一样
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
Message s) {
        //处理文本消息
        if (s instanceof TextMessage) {
            String text = ((TextMessage) s).getText();
            System.out.println("服务器消息: "+text);
            //进行处理
        }
        //处理图片数据,保存图片
        else if (s instanceof ImageMessage) {
            //拿到头部信息
            String header=((ImageMessage)s).getHeader();
            //拿到图片信息
            byte[] image=((ImageMessage)s).getImage();
```

```
            // 保存图片到本地,保存位置image/userName/filename.jpg
            String path=new ClientUtil().getTime(userName);//获取保存图片的名字：当前
时间.jpg
            File file = new File(path);
            // 确保文件路径的目录存在
            File parentDir = file.getParentFile();
            if (!parentDir.exists()) {
                boolean created = parentDir.mkdirs();
                if (!created) {
                    System.err.println("Failed to create directory: " +
parentDir.getAbsolutePath());
                    return;
                }
            }
            //保存图片
            try (FileOutputStream fos = new FileOutputStream(file)) {
                fos.write(image);
                System.out.println("Received image from " + header + " and saved
to " + file.getAbsolutePath());
            } catch (IOException e) {
                System.err.println("Failed to save image: " + e.getMessage());
            }
        }
    }
```

对应的工具方法

```
//获取保存图片的路径
    public String getTime(String userName){
        LocalDateTime now = LocalDateTime.now();
        // 格式化日期时间到毫秒,不能用:跟空格
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-
dd_HH-mm-ss_SSS");
        String formattedDateTime = now.format(formatter);
//        System.out.println(formattedDateTime);
        return "image/"+userName+"/"+formattedDateTime+".jpg";//返回保存图片的路径
    }
```

服务器

在自定义处理添加处理图片信息的逻辑，根据头部信息实现图片消息的转发

```
//处理图片消息
    void dealImage(ChannelHandlerContext channelHandlerContext,ImageMessage msg)
{
        //拿到头部信息
        String header=((ImageMessage)msg).getHeader();
        //拿到图片信息
        byte[] image=((ImageMessage)msg).getImage();
        System.out.println("这是图片信息");
        //切割头部信息
        String[] split = header.split(":", 2);
```

```
        if (split.length == 2) {
            String explain = split[0].trim(); // 说明
            String target = split[1].trim();  // 目标
            System.out.println("说明: " + explain + ", target: " + target);//测试

            //私发给target发送数据
            if (explain.equals("Private")) {
                if (userChannels.containsKey(target)) {//在map中存在目标用户就发
                    Channel channel = userChannels.get(target);
                    // 将图片数据发送到目标用户
                    String userName =
channelUsers.get(channelHandlerContext.channel());//获取消息来源用户名
                    ImageMessage imageMessage = new ImageMessage(image,
userName);//重组图片消息类，头部说明消息来源
                    channel.writeAndFlush(imageMessage);//向目标用户发送图片
                    //返回响应信息
                    channelHandlerContext.channel().writeAndFlush(new
TextMessage("向" + target + "发送图片成功"));
                } else {
                    channelHandlerContext.writeAndFlush(new TextMessage("对方不存
在或者已离线"));
                }
            }
            else {//群发
                if (groupUsers.containsKey(target)) {//群组要存在
                    List<String> users = groupUsers.get(target);
                    // 将图片数据发送到目标用户
                    String userName =
channelUsers.get(channelHandlerContext.channel());//获取消息来源用户名
                    ImageMessage imageMessage = new ImageMessage(image,
userName);//重组图片消息类，头部说明消息来源
                    if (!users.isEmpty()) {
                        for (String user : users) {
                            Channel channel = userChannels.get(user);
                            //向目标发送消息
                            channel.writeAndFlush(imageMessage);
                            //返回响应信息
                            channelHandlerContext.channel().writeAndFlush(new
TextMessage("向" + user + "发送消息成功"));
                        }
                    }
                } else {
                    channelHandlerContext.writeAndFlush(new TextMessage(target +
" 群组不存在"));
                }
            }

        }
        else {//信息不全
            channelHandlerContext.writeAndFlush(new TextMessage("头部信息不全"));
        }
    }
```

整合到springboot中

就是使netty的服务器随着springboot项目的运行而运行

但是需要注意的是**netty的处理端口要跟springboot的http请求的端口不一样**，不然就会很麻烦，因为netty使用tcp可以直接传输string等

1.将netty的服务启动类标记为Bean

注意这样就不能带端口号创建，因为没有int的Bean，可以使用获取配置文件中的端口号来实现创建

```
/**
 * 服务器启动类
 */
@Component
public class ChatServer {
    public static void main(String[] args) {
        new ChatServer().start();//启动服务
    }
    @Value("${netty.server.port}")//注意是netty的端口，不是springboot的端口
    private int port;//启动的端口号
```

2.创建一个监听器

创建**监听器**，当sprigboot初始化完毕就调用netty启动类的启动方法启动netty服务，就是把上面的启动入口改一下，注意要在**Bean环境**下运行

```
/**
 * 通过监听器建立netty服务器
 */
@Component//在bean环境下
public class creatServer implements ApplicationListener<ContextRefreshedEvent> {
    //实现ContextRefreshedEvent，就是在监听所有bean都初始化成功之后进行的操作
    @Resource
    private ChatServer chatServer;
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {//监听到时触发的动
作
        chatServer.start();//开启netty服务器
    }
}
```

其他类型的**监听器**

- `ContextStartedEvent`：应用上下文被启动时发布。
- `ContextStoppedEvent`：应用上下文被停止时发布。
- `ContextClosedEvent`：应用上下文被关闭时发布。
- `ContextRefreshedEvent`，就是在监听所有bean都初始化成功之后进行的操作

实现web端利用netty通信

web端就不能使用socket，而应该使用websocket。那么首先对于客户端来说，一般是利用js函数来读取输入框的输入来发送

比如下面，特别需要注意**路径的问题**

重要

这个路径的问题：当前后端在同一个服务器时，通过ip地址访问连接是没问题的。当前后端分离，又需要ssl证书时，就会出错。因为客户端由于在不同机子上无法信任服务器的ssl证书。因为ssl证书是依据域名的，但是通过ip地址连接是无法成功信任的。这时候就需要通过域名去建立websocket连接

```
 //创建一个websocket的连接
    var websocket=new WebSocket("ws://localhost:80/ws");//注意路径一致，跟服务端配置的websocket协议的路径要一致，后面要带/ws，不然就报错连接不上

   // 当 WebSocket 连接成功时触发
    websocket.onopen=function(){
        console.log("连接成功");
    }
    //当接收到信息时触发
    websocket.onmessage=function(data){
        console.log(data.data);//返回的是一个MessageEvent 对象，调用data方法获取文本信息
    }
    //当连接关闭时触发的操作
    websocket.onclose=function(){
        console.log("连接关闭");
    }
    //当发生错误时触发的操作
    websocket.onerror=function(data){
        console.log("发生错误"+data);
    }

    function sendMessage(){
        //获取输入框的输入
        var message=document.getElementById("message").value;
        //检查连接状态并发送信息
        if(websocket.readyState === WebSocket.OPEN){//如果连接状态==打开
            websocket.send(message);//调用方法发送信息
            document.getElementById("message").value="";//清空输入框
        }
    }
```

那么相应的服务端也需要添加一些东西，因为web是通过http来握手连接，使用websocket来发送数据，所以服务端要添加http的编解码器和WebSocket协议处理器

```
 @Override
                    protected void initChannel(SocketChannel socketChannel) throws Exception {
                        //流水线操作
                        socketChannel.pipeline().addLast(new HttpServerCodec());//处理htpp的编解码
                        //将多个htpp请求聚合为一个
                        socketChannel.pipeline().addLast(new HttpObjectAggregator(65536));
```

```
                        socketChannel.pipeline().addLast(new
ChunkedWriteHandler());//添加来支持大文件的传输
                        //websocket协议，使用/ws路径，也就是js中ip地址的前缀，一定要一致
                        socketChannel.pipeline().addLast(new
WebSocketServerProtocolHandler("/ws"));
                        //添加
                        // 添加自定义的服务器处理器
                        socketChannel.pipeline().addLast(new test());
                    }
                })
                .option(ChannelOption.SO_BACKLOG, 128)//设置连接队列的大小
                .childOption(ChannelOption.SO_KEEPALIVE, true);//设置保持连接选项
```

对于的自定义处理器

```
public class test extends SimpleChannelInboundHandler<TextWebSocketFrame> {

    @Override
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
TextWebSocketFrame textWebSocketFrame) throws Exception {
        System.out.println(textWebSocketFrame.text());//以字符串显示,当文本信息时
        channelHandlerContext.writeAndFlush(new TextWebSocketFrame("已经收到"));
    }
}
```

这样就完成了最基本的连接通信。

处理文本/图片信息

在利用websocket进行的通信的，传输的是**TextWebSocketFrame：文本帧数据**

`TextWebSocketFrame` 是Netty中用于表示WebSocket文本帧的类。它主要用于传输文本数据，包括普通的文本消息和Base64编码的图片。所以直接使用`TextWebSocketFrame`来解析就可以天然支持文本跟图片信息。

图片采用分块发送，在服务端再整合，不然图片太大 。那么如何标识信息的结束

假设信息的结束标识为 `:MESSAGE_DATA_END!`,文本跟图片的最后都要带这个标识结束的标识符,18个字符

不然统一用TextWebSocketFrame传输，不知道到哪里结束。

客户端发送

可以有一个输入框跟一个文件选择框，读取分别发送不用类型的信息。

```
//发送文本信息
function sendMessage(){
        //获取输入框的输入
        var message=document.getElementById("message").value;
        //检查连接状态并发送信息
        if(websocket.readyState === WebSocket.OPEN){//如果连接状态==打开
             websocket.send(message+":MESSAGE_DATA_END!");//调用方法发送信息,带上结束
标识
            document.getElementById("message").value="";//清空输入框
        }
    }
```

发送图片数据，采用分块发送，先发送头部信息，再发送前len-1个，最后再带着结束标识符再发送一次

```
  //发送图片信息
    function sendImage() {
        //获取选择的图片
        var input = document.getElementById('image');
        if (input.files.length > 0) {
            var file = input.files[0];//选择第一个选中的图片发送
            // console.log(file);
            var reader = new FileReader();
            //这个一个触发器，当reader的读取完成之后触发的动作
            reader.onloadend = function() {
                if(websocket.readyState === WebSocket.OPEN){//如果连接状态==打开
                    console.log("准备发送")
                    //分块发送
                    var MaxSize=65518;//设置64kB的块大小,注意算上18字节的结束标识符
                    var array=[];   //设置存储分块的数组
                    var len=reader.result.length;
                    for(var i=0;i<len;i+=MaxSize){//切割
                        //切割数据
                        array.push(reader.result.slice(i,i+MaxSize));
                    }
                    //分块发送
                    //先发送头部消息,写死的，可以修改
                    websocket.send("Group:user1:group1:");//头部消息，可以改成读取拼
接
                    var arrayLen=array.length;
                    for(var j=0;j<arrayLen-1;j++){//发送
                        //发送数据
                        websocket.send(array[j]);//base64编码之后发送的也是
TextWebSocketFrame文本帧
                    }
                    //最后带着标识符发送最后一次
                    //发送结束数据
                    websocket.send(array[j]+":MESSAGE_DATA_END!");//base64编码之后
发送的也是TextWebSocketFrame文本帧

                    console.log("发送成功")
                } else {
                    console.log('未连接');
                }
            };
            //开始读取将选择的图片以base64的编码读取到reader中，注意触发器要写在前面！！！
            reader.readAsDataURL(file);
```

```
        } else {
            console.log('没有选择图片');
        }
    }
```

上面的触发器和读取的操作不能互换，因为`reader.readAsDataURL(file);`是一个异步方法，在读取完成之后就会立马触发`onloadend`操作，如果`onloadend`操作写在读取之后，可能还没注册好那边已经触发了`onloadend`操作，导致发送的总是空值。

客户端接收

判断类型的方法：图片的编码以`"data:image/"`开头，是图片就将图片展示到页面上

```
    //当接收到信息时触发
    websocket.onmessage=function(data){
        if (data.data.startsWith("data:image/")) { //处理图片信息
            var imgElement = document.getElementById('getimage');
            imgElement.src =data.data;//显示图片
            console.log('显示成功')
        }
        else {//处理文本信息
            console.log(data.data);//返回的是一个MessageEvent 对象，调用data方法获取文本信息
        }

    }
```

服务端处理器

接收的可能是文本/图片信息，都是**TextWebSocketFrame**类型，用Stringbuffer来当缓冲区,每次请求的数据都拼接到缓冲区，直到接收到`MESSAGE_DATA_END!`结束标识符结束的数据，就向客户端回显总信息。这样文本，图片都可以

```
public class test extends SimpleChannelInboundHandler<TextWebSocketFrame> {

    private StringBuffer sb = new StringBuffer();//缓冲区
    @Override
    protected void channelRead0(ChannelHandlerContext channelHandlerContext, TextWebSocketFrame textWebSocketFrame) throws Exception {
        System.out.println(textWebSocketFrame.text());//以字符串显示,当文本信息时
        String text = textWebSocketFrame.text();

        //处理具有结束标识的信息
        if(text.endsWith(":MESSAGE_DATA_END!")){
            //切割结束标识
            sb.append(text, 0, text.length()-18);
            //往客户端回显信息
            String result=sb.toString();
            System.out.println(result);
            channelHandlerContext.writeAndFlush(new TextWebSocketFrame(result));
            sb.setLength(0);//清空缓存
        }
        else {//往缓冲区扔数据
```

```
            sb.append(text);
        }
    }
}
```

整合到springboot中

就是使netty的服务器随着springboot项目的运行而运行

但是需要注意的是**netty的处理端口要跟springboot的http请求的端口不一样**，不然就会很麻烦，因为netty使用tcp可以直接传输string等

1.将netty的服务启动类标记为Bean

注意这样就不能带端口号创建，因为没有int的Bean，可以使用获取配置文件中的端口号来实现创建

```
/**
 * 服务器启动类
 */
@Component
public class ChatServer {
    public static void main(String[] args) {
        new ChatServer().start();//启动服务
    }
    @Value("${netty.server.port}")//注意是netty的端口，不是springboot的端口
    private int port;//启动的端口号
```

2.创建一个监听器

创建**监听器**，当sprigboot初始化完毕就调用netty启动类的启动方法启动netty服务，就是把上面的启动入口改一下，注意要在**Bean环境**下运行

```
/**
 * 通过监听器建立netty服务器
 */
@Component//在bean环境下
public class creatServer implements ApplicationListener<ContextRefreshedEvent> {
    //实现ContextRefreshedEvent，就是在监听所有bean都初始化成功之后进行的操作
    @Resource
    private ChatServer chatServer;
    @Override
    public void onApplicationEvent(ContextRefreshedEvent event) {//监听到时触发的动作
        chatServer.start();//开启netty服务器
    }
}
```

其他类型的**监听器**

- `ContextStartedEvent`：应用上下文被启动时发布。
- `ContextStoppedEvent`：应用上下文被停止时发布。
- `ContextClosedEvent`：应用上下文被关闭时发布。
- `ContextRefreshedEvent`，就是在监听所有bean都初始化成功之后进行的操作

加上数据库

使用mysql+mybatisplus

1.导入依赖

```
  <dependency>
            <groupId>com.baomidou</groupId>
            <artifactId>mybatis-plus-spring-boot3-starter</artifactId>
            <version>3.5.7</version>
        </dependency>
```

2.配置数据源

```
spring:
  #数据库连接
  datasource:
    url: jdbc:mysql://localhost:3306/lms
    username: root
    password: 18977474527ai
    driver-class-name: com.mysql.cj.jdbc.Driver
```

3.导入生成代码的依赖**

```
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-generator</artifactId>
    <version>3.5.7</version>
</dependency>
<dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity-engine-core</artifactId>
    <version>2.3</version>
</dependency>
```

4.编写代码生成脚本

比如说在测试类里面编写以下生成代码的代码

```
  @Resource
    DataSource dataSource;//拿到配置信息的数据源

    @Test
    void contextLoads() {
        FastAutoGenerator
                //首先使用create来配置数据库链接信息
                .create(new DataSourceConfig.Builder(dataSource))
                //全局配置，比如作者信息
                .globalConfig(builder -> {
                    builder.author("lzj");              //作者信息，一会会变成注释
                    builder.commentDate("2024-10-15");  //日期信息，一会会变成注释
```

```
                    builder.outputDir("src/main/java"); //输出目录设置为当前项目的目
录
                })
                //打包配置
                //打包设置，这里设置一下包名就行，注意跟我们项目包名设置为一致的
                .packageConfig(builder -> builder.parent("org.example.webchat"))
                .strategyConfig(builder -> {
                    //设置为所有Mapper添加@Mapper注解
                    builder
                            .mapperBuilder()
                            .mapperAnnotation(Mapper.class)//ibatis包里面的mapper
                            .build();
                })
                .execute();
    }
```

成功建立了控制器，实体类跟表的映射，业务方法，对应的mapper方法

5.根据项目规范调整结构

表结构

设计用户表结构如下

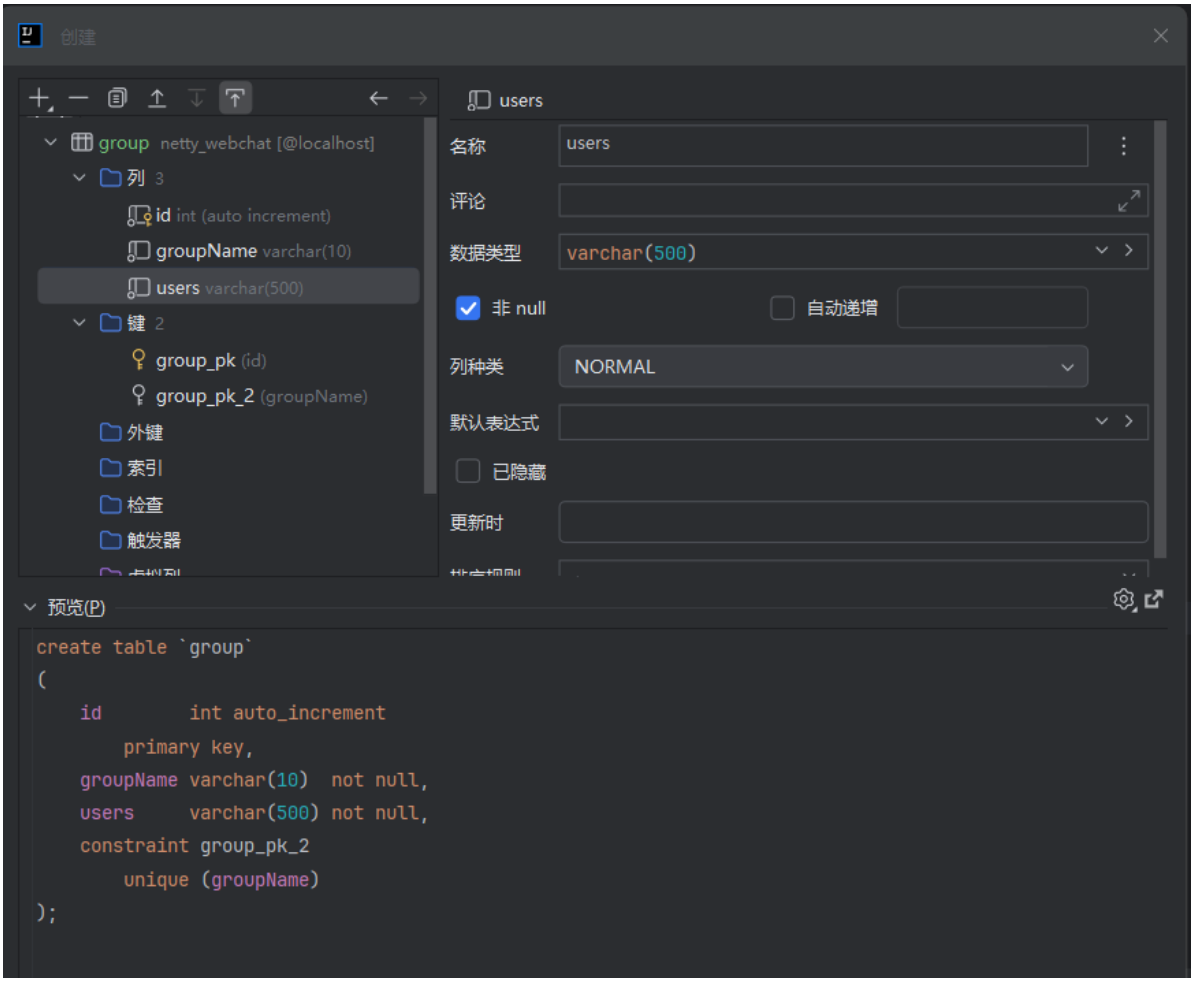
用户表user

id（主键）	用户名（唯一健）	websocket的唯一uid标识（唯一键）	加入的群组
			group1,group2的形式拼接

群组表grouped

id（主键）	群组名(唯一键)	拥有的用户
		user1+uid,user2+uid的形式拼接

比如



实现私发跟群聊

跟正常的socket逻辑一样，存储一个map映射用户名跟通道，还有记录用户-群组，群组-其拥有的用户，来实现根据用户名来私发跟根据群组来实现群发的功能。

但是这里的**用户名跟通道的映射**要持久化存储到数据库中,还有群组的详细信息也要持久化存储

但是**websocket不是基本的数据类型,不能直接放进数据库中**，可以在数据库存储用户名-通道标识uid，然后内存使用一个map（可以考虑redis，这里先使用map实现）维护uid跟对应通道的映射关系。注意当用户重新连接要更新内存中的uid对应的通道

消息格式

假设消息的格式如下，　　**可修改**

那么有以下信息表格，，　　message包括文本跟图片信息

web端发送

消息格式	说明
REGISTER:username	注册信息，传递用户名
JOIN:group:username	加入群组，将该用户加入群组
Private:userId:targetId:message	一对一的信息发送，username为自己的用户名，target为目标用户
Group:username:groupname:message	多人聊天的信息发送，中间为发送者：目标群组
:MESSAGE_DATA_END!	结束标识，图片文本信息都要携带

服务端响应信息格式

from 来源:message	来源:具体信息

客户端在一开始创建时需要向服务端发送一条消息告知自己的用户名

web端发送

为了能记录用户名，把websocket的连接放到一个函数中,这样就可以读取用户名连接了

```
// 当 WebSocket 连接成功时触发
    websocket.onopen=  function(){
        console.log("连接成功");
        var name=document.getElementById('name').value;
        websocket.send("REGISTER:"+name+":MESSAGE_DATA_END!");//发送注册信息,带上结束标识
        console.log("REGISTER:"+name+":MESSAGE_DATA_END!")
    }
```

其他信息要拼接头部信息发送:　　**现在直接输入带着头部信息，后面可以改**

```
//发送文本信息
    function sendMessage(){
        //获取输入框的输入
        var message=document.getElementById("message").value;
        if(message.length>0){
            //检查连接状态并发送信息
            if(websocket.readyState === WebSocket.OPEN){//如果连接状态==打开
                websocket.send(message+":MESSAGE_DATA_END!");//调用方法发送信息,带上结束标识
                document.getElementById("message").value="";//清空输入框
            }
        }
        else {
            console.log('没有输入信息')
        }
    }
```

图片信息的发送就是在分块之前先发送一条说明头部信息的信息

```
 //分块发送
                    //先发送头部消息,写死的，可以修改
                    websocket.send("Private:user1:user1:");//头部消息，可以改成读取拼
接
                    var arrayLen=array.length;
                    for(var j=0;j<arrayLen-1;j++){//发送
                        //发送数据
                        websocket.send(array[j]);//base64编码之后发送的也是
TextWebSocketFrame文本帧
                    }
                    //最后带着标识符发送最后一次
                    //发送结束数据
                    websocket.send(array[j]+":MESSAGE_DATA_END!");//base64编码之后
发送的也是TextWebSocketFrame文本帧
```

web端接收

因为服务端发送过来的消息格式是`from 用户:消息`,所以要先切割消息，分别展示，

```
 //当接收到信息时触发
    websocket.onmessage=function(data){
        //切割消息
        var i=data.data.indexOf(":");
        if(i!==-1){
            var header=data.data.substring(0, i);//来源消息
            var message=data.data.substring(i+1);//具体消息,可能是图片，文本
            console.log(header);//先说明来源
            if (message.startsWith("data:image/")) { //处理图片信息
                var imgElement = document.getElementById('getimage');
                imgElement.src =message;//显示图片
                console.log('显示成功')
            }
            else {//处理文本信息
                console.log(message);//返回的是一个MessageEvent 对象，调用data方法获取
文本信息
            }
        }
        else {
            console.log("响应信息有问题");
        }

    }
```

服务端处理

这里有个问题：`ChannelPipelineException`异常，使用原型bean+从上下文获取处理器就可以解决

对于信息就提取头部信息,根据消息格式分别处理

```
//处理器
@Component
```

```
@Scope("prototype")//原型，声明为每次需要都创建新的
public class Handler extends SimpleChannelInboundHandler<TextWebSocketFrame> {
    // 存储所有uid跟通道的映射表
    private static final HashMap<String, Channel> uidChannels = new HashMap<>();

    private StringBuffer sb = new StringBuffer();//缓冲区

    @Override//有消息时的处理操作
    protected void channelRead0(ChannelHandlerContext channelHandlerContext,
TextWebSocketFrame textWebSocketFrame) throws Exception {
        System.out.println(textWebSocketFrame.text());//以字符串显示,当文本信息时
        String text = textWebSocketFrame.text();

        //处理具有结束标识的信息
        if(text.endsWith(":MESSAGE_DATA_END!")){
            //切割结束标识
            sb.append(text, 0, text.length()-18);
            //获取总信息
            String result=sb.toString();
            //分情况进行信息处理
            dealMessage(result,channelHandlerContext);
            //
          //  channelHandlerContext.writeAndFlush(new
TextWebSocketFrame(result));
            sb.setLength(0);//清空缓存
        }
        else {//往缓冲区扔数据
            sb.append(text);
        }
    }
    //断开连接就从map中删除
    @Override//重写客户端断开连接的操作
    public void channelInactive(ChannelHandlerContext ctx) throws Exception {
        //遍历map删除用户
        for(Map.Entry<String, Channel> entry: uidChannels.entrySet()){
            if(Objects.equals(entry.getValue(), ctx.channel())){
                uidChannels.remove(entry.getKey());
                break;
            }
        }
    }
```

对应的分类消息处理

```
  @Resource
    IUserService userService;//业务接口类
    @Resource
    IGroupedService groupService;

   //处理信息
    void dealMessage(String s,ChannelHandlerContext channelHandlerContext){
        //如果是注册消息就往hashmap中加入
        if (s.startsWith("REGISTER:")) {
            String userName = s.split(":")[1].trim();//拿到用户名
            //查询在数据库中是否存在
            User byName = userService.findByName(userName);
            if(byName==null){//不存在
```

```
                //生成唯一的uid,可以调用工具类，这里暂时先实现
                Random r = new Random();
                int i = r.nextInt(1000);
                //将uid跟通道建立映射

 uidChannels.put(String.valueOf(i),channelHandlerContext.channel());
                //往数据库中存储用户名跟uid的映射
                int result=userService.addUser(userName,String.valueOf(i));
                if(result==1){
                    System.out.println("新加记录成功");
                    channelHandlerContext.writeAndFlush(new
TextWebSocketFrame("from 服务器:注册成功"));
                }
                else {
                    System.out.println("新加记录失败");

                }
            }
            else { //如果存在就更新内存中map中的通道
                //更新uid跟通道建立映射

 uidChannels.put(byName.getWebSocketUid(),channelHandlerContext.channel());
                channelHandlerContext.writeAndFlush(new TextWebSocketFrame("from
服务器:"+"欢迎登录 "+userName));//返回响应
            }
        }else if (s.startsWith("JOIN:")) {//加入群组的消息,格式user1+uid,user2+uid的
形式拼接
            String group = s.split(":")[1].trim();//拿到群组名
            String userName = s.split(":")[2].trim();//拿到用户名
            //先查询有无该群组
            Grouped group1=groupService.getGroupByName(group);
            if(group1==null){//不存在该群组就新加
                //先获取该用户对于的uid
                String uid=userService.findByName(userName).getWebSocketUid();
                int result=groupService.addGroup(group,userName,uid);//新增群组
                if(result==1){
                    System.out.println("新加群组成功");
                    channelHandlerContext.writeAndFlush(new
TextWebSocketFrame("from 服务器:"+"加入群组"+group+"成功"));//返回响应
                }
                else {
                    System.out.println("新加群组失败");
                }
            }
            else {//往已经存在的群组添加用户
                //先获取该用户对于的uid
                String uid=userService.findByName(userName).getWebSocketUid();
                int result=groupService.addUser(group,userName,uid);//添加成员
                if(result==1){
                    System.out.println("新加成员成功");
                    channelHandlerContext.writeAndFlush(new
TextWebSocketFrame("from 服务器:"+"加入群组"+group+"成功"));//返回响应
                }
                else {
                    System.out.println("新加成员失败");
                }
            }
            //更新用户表里面的已经加入的群组信息
```

```
            userService.updateGroup(group,userName);

        }else {//正常的私发或者群发消息,根据消息格式不同进行不同操作
            //调用方法解析消息的用户名
            String[] split = s.split(":", 4);//最多返回4个，即只进行三次切割
            if (split.length == 4) {
                String explain = split[0].trim();//说明
                String userName = split[1].trim();//来源
                String target = split[2].trim();//目标
                String message = split[3];//消息
                if (explain.equals("Private")) {//私发
                    //查找数据库获取target的uid,再根据uid获取通道发送
                    User byName = userService.findByName(target);
                    String uid=byName.getWebSocketUid();
                    Channel channel = uidChannels.get(uid);//获取对应的通道
                    //转发消息
                    channel.writeAndFlush(new TextWebSocketFrame("From
"+userName+":"+message));
                    //响应成功
                    channelHandlerContext.writeAndFlush(new
TextWebSocketFrame("向"+target+":"+"发送信息成功"));
                } else {//群发
                    //先判断该用户有无加入该群组,需要吗?
                    //往该群组里面的所有用户传输消息
                   String[] uids=groupService.getUsers(target); //根据群组名字获取用
户名，uid的数组
                    //遍历uids给每个用户发送消息
                    for (int i = 0; i < uids.length; i++) {
                        String uid = uids[i].split("\\+")[1];//用户的uid
                        String name=uids[i].split("\\+")[0];//用户名
                        if(uidChannels.containsKey(uid)){//往在线的用户发送信息
                            uidChannels.get(uid).writeAndFlush(new
TextWebSocketFrame("From "+userName+":"+message));
                        }
                        else {//对方不在线
                            channelHandlerContext.writeAndFlush(new
TextWebSocketFrame(name+" 不在线"));
                        }

                    }
                }
            }
            else {
                channelHandlerContext.writeAndFlush(new TextWebSocketFrame("消息格
式有问题！"));
            }
        }
    }
```

对应的业务方法跟实现类也要写好

User表

```
public interface IUserService extends IService<User> {
    //查询该用户名是否存在
    User findByName(String name);
    //往用户表里面新增记录
    int addUser(String name,String uid);

    //更新用户表的群组信息,根据用户名更新
    int updateGroup(String group,String name);
}
```

```
@Service
public class UserServiceImpl extends ServiceImpl<UserMapper, User> implements
IUserService {

    @Resource
    private UserMapper userMapper;

    @Override
    public User findByName(String name) {
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("UserName", name)
                .select("WebSocketUid");
        return userMapper.selectOne(queryWrapper);
    }

    @Override
    public int addUser(String name, String uid) {
        System.out.println("新增用户");
        User user = new User();
        user.setUserName(name);
        user.setWebSocketUid(uid);
        user.setJoinGroups("");
        return userMapper.insert(user);//插入
    }

    @Override
    public int updateGroup(String group,String name) {
        QueryWrapper<User> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("UserName", name);
        User user = userMapper.selectOne(queryWrapper);//获取要更新的实体类
        //更新群组信息
        String s;
        if(user.getJoinGroups().isEmpty()){
            s=group;
        }
        else {
            s=user.getJoinGroups()+","+group;
        }

        user.setJoinGroups(s);//更新群组信息
        return userMapper.updateById(user);//更新
    }
}
```

grouped表

```java
public interface IGroupedService extends IService<Grouped> {
    //查询有无该群组
    Grouped getGroupByName(String groupName);

    //新增群组,格式user1+uid,user2+uid的形式拼接
    int addGroup(String groupName,String username,String uid);

    //往已经有的群组里面新增用户
    int addUser(String groupName,String users,String uid);

    //根据群组名字获取对应的用户列表
    String[] getUsers(String groupName);
}
```

```java
@Service
public class GroupServiceImpl extends ServiceImpl<GroupedMapper, Grouped>
implements IGroupedService {

    @Resource
    private GroupedMapper groupMapper;
    @Override
    public Grouped getGroupByName(String name) {
        QueryWrapper<Grouped> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("groupName", name)
                .select("groupName");
        return groupMapper.selectOne(queryWrapper);
    }

    @Override
    public int addGroup(String groupName, String username, String uid) {
        //在用户的列表拼接用户跟uid
        String users = username + "+" + uid;
        Grouped group = new Grouped();
        group.setGroupName(groupName);
        group.setUsers(users);
        return groupMapper.insert(group);
    }

    @Override
    public int addUser(String groupName, String username,String uid) {
        QueryWrapper<Grouped> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("groupName", groupName);
        Grouped group = groupMapper.selectOne(queryWrapper);
        //在用户的列表拼接用户跟uid
        String users = group.getUsers() + "," + username + "+" + uid;
        group.setUsers(users);
        //更新表
        return groupMapper.updateById(group);
    }

    @Override
    public String[] getUsers(String groupName) {
        QueryWrapper<Grouped> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("groupName", groupName);
        Grouped group = groupMapper.selectOne(queryWrapper);
        //切割获取uid列表
```

```
        String users = group.getUsers();
        String[] split = users.split(",");
        return split;
    }

}
```

使用

1.启动springboot项目，就启动了netty服务器

2.在web连接，主要是靠js函数websocket,实现tcp连接

3.根据消息格式发送消息就行

实现历史聊天的存储

最基本的要知道谁发的，跟谁接收的，就可以区分消息了。

使用当下比较流行的单表结构来存储，方便信息的集中管理跟一致性。设计表结构如下

表结构

消息主表（chat_messages）

字段	类型	说明
id	bigint	主键
sender_id	bigint	标识是哪个用户发送的消息
group_id	bigint	标识属于哪个群组的信息，私发为空
send_at	TIMESTAMP	标识信息发送的时间戳,以 `YYYY-MM-DD HH:MM:SS` 为格式
message	TEXT	具体的消息内容
message_type	ENUM	标识消息的类型，如果是图片等上面message可以放地址，不用放base64编码的太长
is_read	boolean	标识该消息是否已读

接收者表（message_recipients）

之所以区分出来接收者表,是因为如果是群聊消息，那么一个人发送，其他人接收。如果把接收者id也当一个属性放在消息主表里面，就会造成除了接收者不同的相同消息的大量重复存储。所以单独把接收者表拎出来，只需要存储id映射，减少空间损耗。

字段	类型	说明
id	bigint	主键
message_id	bigint	消息主表的键,标识该条消息，注意不能为外键，因为下面利用分区来优化，mysql的分区不允许有外键
receiver_id	bigint	标识该条消息的接收者id
is_read	boolean	标识是否已读

其实这样也有不好的地方，每当群组消息多一条，那么接收者表就要多插入好多条数据。但是如果把接收者用列表存储，也不好。当我想查询一个人接收到的历史数据时不好操作。

利用分区来优化性能

暂时先采取时间范围来分区,注意这里要注意两个问题，在下面遇到问题中，要注意分区条件要确定，不能是变量，分区的列得是主键的一部分。

```
 #指定范围(range)作为分区,PARTITION是关键字
    PARTITION BY RANGE (UNIX_TIMESTAMP(sent_at)) (
        #具体的分区PARTITION,按月份来分区,less then表示少于这个分区时间的消息都会被存储在这个分区
        PARTITION p2024_10 VALUES LESS THAN (UNIX_TIMESTAMP('2024-10-01')),
        PARTITION p2024_11 VALUES LESS THAN (UNIX_TIMESTAMP('2024-11-01')),
        PARTITION p2024_12 VALUES LESS THAN (UNIX_TIMESTAMP('2024-12-01')),
        PARTITION p2025_01 VALUES LESS THAN (UNIX_TIMESTAMP('2025-01-01')),
        PARTITION p_future VALUES LESS THAN MAXVALUE
        );
```

实现

消息主表

```
CREATE TABLE chat_messages (
                               id BIGINT AUTO_INCREMENT,
                               sender_id BIGINT NOT NULL,
                               group_id BIGINT,
                               message TEXT NOT NULL,
                               message_type ENUM('TEXT', 'IMAGE', 'FILE') NOT NULL,
                               sent_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,#不指定默认当前时间戳，CURRENT_TIMESTAMP为获取当前时间戳
                               is_read BOOLEAN DEFAULT FALSE,
                                PRIMARY KEY (id, sent_at)#注意把要依赖分区的列放主键中，这是限制
)
    #指定范围(range)作为分区
    PARTITION BY RANGE (UNIX_TIMESTAMP(sent_at)) (
        #具体的分区PARTITION,按月份来分区,less then表示少于这个分区时间的消息都会被存储在这个分区
        PARTITION p2024_10 VALUES LESS THAN (UNIX_TIMESTAMP('2024-10-01')),
```

```
        PARTITION p2024_11 VALUES LESS THAN (UNIX_TIMESTAMP('2024-11-01')),
        PARTITION p2024_12 VALUES LESS THAN (UNIX_TIMESTAMP('2024-12-01')),
        PARTITION p2025_01 VALUES LESS THAN (UNIX_TIMESTAMP('2025-01-01')),
        PARTITION p_future VALUES LESS THAN MAXVALUE
        );
```

接收者表

```
CREATE TABLE message_recipients (
                                    id BIGINT AUTO_INCREMENT ,
                                    message_id BIGINT NOT NULL,
                                    user_id BIGINT NOT NULL,
                                    is_read BOOLEAN DEFAULT FALSE,
                                    PRIMARY KEY (id)
);
```

整合

在服务器接收到消息进行转发时，先存储信息到这两个表中。

因为是后面加的表，所以要先编写实体类+mapper+service+impl，，可以**新建项目生成移过来**

注意：因为消息主表要分区所以，send_at也是主键的一部分，自动生成的代码会报错，因为mybatisplus默认只支持单主键。但是好像我把send_at直接改为普通列也没问题，哈哈哈。

实现：在服务器接收消息时把消息也存储到这两张表中，

```
                if (explain.equals("Private")) {//私发
                    //保存该条消息,文本形式
                    long
messageId=perserveMessage_Private(userName,message,"TEXT");
                    //往接收者表中添加数据
                    int userId=userService.getId(target);
                    addMessageRecipients(messageId,userId);
                  //转发消息的其他操作

                } else {//群发
                    //保存该条消息,文本形式
                    long
messageId=perserveMessage_Group(userName,target,message,"TEXT");

                    //先判断该用户有无加入该群组,需要吗?
                    //往该群组里面的所有用户传输消息
                   String[] uids=groupService.getUsers(target); //根据群组名字获取用
户名，uid的数组
                    //遍历uids给每个用户发送消息
                    for (int i = 0; i < uids.length; i++) {
                        String uid = uids[i].split("\\+")[1];//用户的uid
                        String name=uids[i].split("\\+")[0];//用户名

                        //往接收者表中添加数据
                        int userId=userService.getId(name);
```

```
                        addMessageRecipients(messageId,userId);

                    //转发消息的其他操作

                }
            }
```

对应的方法：业务方法跟实现类也要写好

```
 @Resource
    IChatMessagesService chatMessagesService;
    @Resource
    IMessageRecipientsService messageRecipientsService;

    //保存历史信息的方法
    //私发的保存,返回新加的消息主表id:发生者，消息，消息类型
    long perserveMessage_Private(String name,String message,String message_type)
{
        //获取来源跟目标的id
        int name_id=userService.getId(name);
        ChatMessages chat=new ChatMessages();
        //私发的对于消息主表只需要发送者id，消息跟类型就行，群组的id为空
        chat.setSenderId((long) name_id);
        chat.setMessage(message);
        chat.setMessageType(message_type);
        chatMessagesService.addChatMessage(chat);//添加
        return chat.getId();
        //获取消息主表的id，mybatisplus会回填到对象中
    }
    //群发的保存:发生者，发送的群组，消息，消息类型
    long perserveMessage_Group(String name,String target ,String message,String
message_type){
        //获取来源的id
        int name_id=userService.getId(name);
        //获取群组的id
        int group_id=groupService.getId(target);
        ChatMessages chat=new ChatMessages();
        //群发的对于消息主表只需要发送者id，消息跟类型，群组的id
        chat.setSenderId((long) name_id);
        chat.setMessage(message);
        chat.setMessageType(message_type);
        chat.setGroupId((long) group_id);
        chatMessagesService.addChatMessage(chat);//添加
        return chat.getId();
    }
    //接收者表添加记录:消息主键，接收者id
    void addMessageRecipients(long messageId,int userId){
        MessageRecipients mr=new MessageRecipients();
        mr.setMessageId(messageId);
        mr.setUserId((long) userId);
        messageRecipientsService.addIMessageRecipients(mr);//添加
    }
```

测试成功！

WHERE ORDER BY

	id	sender_id	group_id	message	message_type	sent_at
1	1	1	<null>	hello	TEXT	2024-10-21 13:52:11
2	3	13	<null>	message	TEXT	2024-10-22 09:27:08
3	4	13	<null>	tmd	TEXT	2024-10-22 09:29:07
4	2	2	<null>	你好	TEXT	2024-11-23 13:53:09

图片信息的保存

如果直接在消息表中保存base64编码的字符串图片信息，太大了。所以在服务器接收到图片信息时，分用户建立目录先把图片保存下来，在消息表中保存图片的路径作为消息主体就行。

保存图片

要保存这个文件的路径，path设为类变量

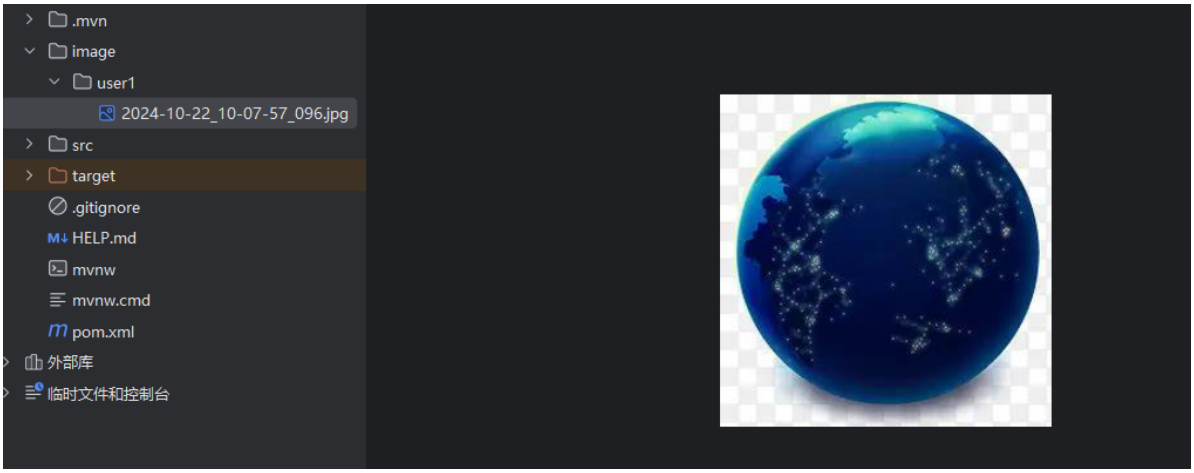
```
//如果是图片消息先保存下来，目录结构为image/username/图片
                if (message.startsWith("data:image/")) {
                    int num = message.length() % 4;

                    // 解码Base64字符串,要删掉""的前缀,一共23
个字符，从第23位置切割就行
                    byte[] imageBytes =
Base64.getDecoder().decode(message.substring(23));
                    // 保存图片到本地,保存位置image/userName/filename.jpg
                    path = getPath(userName);//获取保存图片的名字：当前时间.jpg
                    File file = new File(path);
                    // 确保文件路径的目录存在
                    File parentDir = file.getParentFile();
                    if (!parentDir.exists()) {
                        boolean created = parentDir.mkdirs();
                        if (!created) {
                            System.err.println("Failed to create directory: " +
parentDir.getAbsolutePath());
                            return;
                        }
                    }
                    // 将解码后的字节写入文件
                    try {
                        Files.write(Paths.get(path), imageBytes);
                    } catch (IOException e) {
                        throw new RuntimeException(e);
                    }
                }
//进行具体的消息处理
```

在保存信息的时候分情况保存，图片就消息主体保存路径

```
                //处理消息
                if (explain.equals("Private")) {//私发
                    //保存该条消息,分类型保存
                    long messageId=0;
                    if(message.startsWith("data:image/")){//图片保存
```

```
messageId=perserveMessage_Private(userName,path,"IMAGE");
                    }
                    else {

messageId=perserveMessage_Private(userName,message,"TEXT");
                    }
                    //往接收者表中添加数据
                    int userId=userService.getId(target);
                    addMessageRecipients(messageId,userId);


                   //转发操作

                } else {//群发,接收的部分不用动
                    //保存该条消息,分类型保存
                    long messageId=0;
                    if(message.startsWith("data:image/")){//图片保存

messageId=perserveMessage_Group(userName,target,path,"IMAGE");
                    }
                    else {

 messageId=perserveMessage_Group(userName,target,message,"TEXT");
                    }

                    //先判断该用户有无加入该群组,需要吗?
                    //往该群组里面的所有用户传输消息
                   String[] uids=groupService.getUsers(target); //根据群组名字获取用户名，uid的数组
                    //遍历uids给每个用户发送消息
                    for (int i = 0; i < uids.length; i++) {
                        String uid = uids[i].split("\\+")[1];//用户的uid
                        String name=uids[i].split("\\+")[0];//用户名

                        //往接收者表中添加数据
                        int userId=userService.getId(name);
                        addMessageRecipients(messageId,userId);

                       //转发操作

                    }
                }
```

测试： 成功！ 到这里已经实现了历史聊天的存储，对于一些检索的功能也支持了，不过现在暂时不写。



离线消息的处理

背景：前面是数据库中存储用户名跟uid的映射，而内存中维护一个map保存uid跟通道的映射。当客户端连接断开时，我从map中删掉了这个uid跟通道的映射。如果在发送消息时在map检索不到该用户的通道消息，说明不在线。那么这时候要发送离线消息。

实现：另外新建一张表存储离线消息，当发送时发现对方不在线，就先把对方的用户名跟message保存到一张表中,当目标对象重新连接之后，就先去检索这个表里面有无未接收的消息，如果有就接收。

因为**历史聊天中已经存储了聊天信息**，所以不用再额外存储。只需要存储消息主表的id跟用户的用户名，uid就行

表结构

设计表如下（**offline_message**）

字段	类型	说明
id	int	主键
message_id	bigint	消息主表的id
username	String	待接收者的名字(因为设定带名字请求)
uid	String	待接收者的uid，用来直接查通道
source	String	该条消息的发送者

```
create table netty_webchat.offline_message
(
    id         int auto_increment
        primary key,
    message_id bigint       not null,
    uid        varchar(255) not null,
    username   varchar(10)  not null,
    sourcename     varchar(10)  not null
);
```

实现

首先要建先编写 实体类+mapper+service+impl，，可以**新建项目生成移过来**

存储

在私发跟群发的转发操作发现不在map中的分支添加处理逻辑

私发的

```
if(channel!=null){
//正常转发
                    }
                    else {//不在线的处理逻辑，存储到离线消息表
                        //往离线消息表添加数据

 offlineMessageService.addOfflineMessage(messageId,target,uid,userName);

                        //返回响应消息
                        channelHandlerContext.writeAndFlush(new
TextWebSocketFrame(target+" 不在线"));
                    }
```

群发的

```
    //遍历uids给每个用户发送消息
                    for (int i = 0; i < uids.length; i++) {
                        String uid = uids[i].split("\\+")[1];//用户的uid
                        String name=uids[i].split("\\+")[0];//用户名

                        //往接收者表中添加数据
                        int userId=userService.getId(name);
                        addMessageRecipients(messageId,userId);

                        if(uidChannels.containsKey(uid)){//往在线的用户发送信息
                            uidChannels.get(uid).writeAndFlush(new
TextWebSocketFrame("From "+userName+":"+message));
                        }
                        else {//对方不在线
                            //往离线消息表添加数据
```

```
 offlineMessageService.addOfflineMessage(messageId,name,uid,userName);
                            //返回响应消息
                            channelHandlerContext.writeAndFlush(new
TextWebSocketFrame(name+" 不在线"));
                        }

                    }
```

测试：完成！

延迟发送

当一个连接进来时先去找离线消息表，如果有未接收的消息就接收。因为每次都会发送一个注册信息，在注册信息那里添加逻辑。

注意图片的返回是一个地址，要取到图片以base64编码返回。

```
if{
//不存在的注册操作
}
 else { //如果存在就更新内存中map中的通道
                //先查询有无未接收的消息
                List<OfflineMessage> list =
offlineMessageService.getOfflineMessageByName(userName);
                //有就接收
                if(list!=null){
                    for (OfflineMessage offlineMessage : list) {
                        //获取消息主体跟消息类型
                        ChatMessages messageById =
chatMessagesService.getMessageById(offlineMessage.getMessageId());
                        //接收离线文本消息
                        if(messageById.getMessageType().equals("TEXT"))
                        {
                            channelHandlerContext.writeAndFlush(
                                    new TextWebSocketFrame("from
"+offlineMessage.getSourcename()+"的离线消息:"+messageById.getMessage()));
                        }
                        else {//接收离线图片消息
                            //根据路径读取图片转化为base64格式发送
                            // 读取图片文件为字节数组
                            Path path = Paths.get(messageById.getMessage());
                            byte[] imageBytes = null;
                            try {
                                imageBytes = Files.readAllBytes(path);
                            } catch (IOException e) {
                                throw new RuntimeException(e);
                            }
                            // 将字节数组编码为Base64字符串
                            String base64Image =
Base64.getEncoder().encodeToString(imageBytes);
                            //经过测试发现，通过这个转化的消息是没有前缀标识，要加
上""，因为在保存图片时就截取了
                            //发送
                            channelHandlerContext.writeAndFlush(
                                    new TextWebSocketFrame("from
"+offlineMessage.getSourcename()
```

```
                                                    +"的离线消
息:"+""+base64Image));
                            }

                        }
                    }

                    //更新uid跟通道建立映射的操作
                }
```

测试：成功！

接收之后删除：接收了离线消息就要同步从离线消息表中删除

```
 //接收之后将该条未接收消息从表中删除

 offlineMessageService.deleteOfflineMessage(offlineMessage.getId());
```

对应的服务接口方法跟实现类

```
/**
 * <p>
 *  服务类
 * </p>
 *
 * @author lzj
 * @since 2024-10-22
 */
public interface IOfflineMessageService extends IService<OfflineMessage> {
    //往离线表添加记录
    int addOfflineMessage(long messageId,String name,String uid,String
sourcename);

    //按待接收的用户名查询有无未接收的消息
    List<OfflineMessage> getOfflineMessageByName(String name);

    //根据id删除该条未接收消息
    int deleteOfflineMessage(int id);

}
```

```
/**
 * <p>
 *  服务实现类
 * </p>
 *
 * @author lzj
 * @since 2024-10-22
 */
@Service
public class OfflineMessageServiceImpl extends ServiceImpl<OfflineMessageMapper,
OfflineMessage> implements IOfflineMessageService {
```

```
    @Resource
    private OfflineMessageMapper offlineMessageMapper;

    @Override
    public int addOfflineMessage(long messageId, String name, String uid,String
sourcename) {
        OfflineMessage offlineMessage = new OfflineMessage();
        offlineMessage.setMessageId(messageId);
        offlineMessage.setUid(uid);
        offlineMessage.setUsername(name);
        offlineMessage.setSourcename(sourcename);
        return   offlineMessageMapper.insert(offlineMessage);//插入

    }

    @Override
    public List<OfflineMessage> getOfflineMessageByName(String name) {
        QueryWrapper<OfflineMessage> queryWrapper = new QueryWrapper<>();
        queryWrapper.eq("username", name);
        return offlineMessageMapper.selectList(queryWrapper);
    }

    @Override
    public int deleteOfflineMessage(int id) {
        return offlineMessageMapper.deleteById(id);
    }
}
```

测试：能正常存储离线消息，能在连接时接收未接收的离线消息并从离线表中删除。

时序数据库

看时序数据库的·笔记

打包设置

新加stop方法，在springboot上下文结束时停止并且释放netty的资源

```
//关闭netty服务的方法,释放资源,不然springboot项目一运行就开启，但是项目关闭这个不关闭.
    public void stop() {

        bind.channel().closeFuture();

        bossGroup.shutdownGracefully();

        workerGroup.shutdownGracefully();

    }
```

另外把start方法改成

音视频聊天

看对应的笔记

遇到问题

问题：在客户端连接之后发送的注册信息没发送出去

解决：因为没加同步sync，连接操作是异步的，可能还没连接好就发送不出去

```
//应该
//连接服务器,注意添加sync()来保持同步，等连接成功再发送注册信息
            ChannelFuture connect = strap.connect(host, port).sync();
            //发送注册消息
            connect.channel().writeAndFlush("REGISTER:"+username);
```

问题：你遇到的 `io.netty.channel.ChannelPipelineException` 异常提示 `org.example.netty_chat.NettyChat.Server.chatServerHandler is not a @Sharable handler, so can't be added or removed multiple times.` 这个错误通常是因为你试图多次将同一个非共享的处理器实例添加到不同的通道管道中。

解决：因为我在服务器的自定义处理器时使用了类变量，但是每个客户端都应该对应有一个处理器实例，不然发送的对象都是同一个了，因为维护的map是静态的，属于类，实例不影响

问题：从错误信息来看，您遇到的是一个 `IndexOutOfBoundsException` 异常，该异常发生在尝试向 `ByteBuf` 中写入数据时，写入的数据量超过了 `ByteBuf` 的最大容量。具体来说，`writerIndex`（当前写指针的位置）加上要写入的最小字节数（`minWritableBytes`）超出了 `ByteBuf` 的最大容量（`maxCapacity`）。

解决:在客户端传输图片时，没有事先计算图片和头部信息的总大小，导致超限

在分配ByteBuf的时候先计算一下总大小就行

```
// 计算头部信息的字节长度
                            String header = explain + ":" + target + ":IMAGE";
                            byte[] headerBytes = header.getBytes();
                            int headerLength = headerBytes.length;
                            // 计算图片数据的字节长度
                            int imageLength = imageBytes.length;
                            // 计算总长度
                            int totalLength = headerLength + imageLength;
                            // 创建一个初始容量为总长度的 ByteBuf
                            ByteBuf buffer = Unpooled.buffer(totalLength+100);
```

问题：客户端发送ByteBuf数据，服务器错误解析为string类型

解决;使用自定义的`ByteToMessageDecoder`和`MessageToByteEncoder`来编解码

问题;客户端发送的图片服务器收不到，原因可能是图片信息Imagemessage利用ByteBuf传输，但是应该使用字节数组传输图片信息

解决：修改ImageMessage类，终于接收到了

```
/**
 * 图片消息
 */
@Getter
public class ImageMessage extends Message {
    private final String header;//头部消息
    private final byte[] image;//图片消息
    public  ImageMessage(byte[] image,String header) {
        this.image = image;
        this.header = header;
    }
}
```

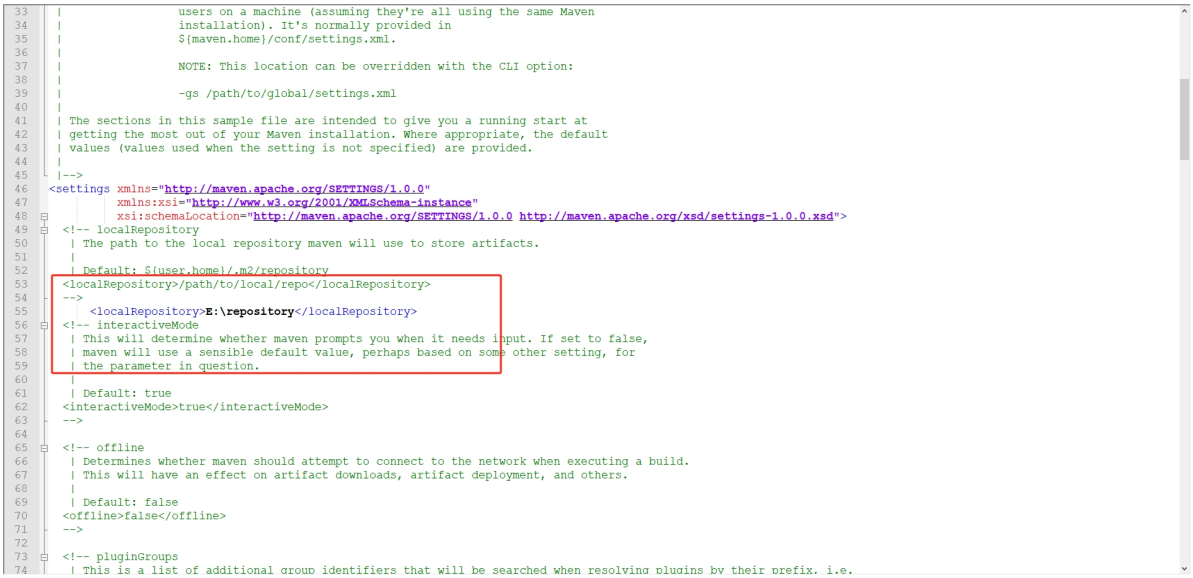
问题：在拉项目时，maven报错，一堆依赖无法解析，原因是我现在的maven版本太高，Maven 3.8.1及更高版本默认禁止了通过HTTP协议访问远程仓库，以提高安全性。Maven现在强制使用HTTPS协议来访问远程仓库。

解决:

1.访问`[Maven – 下载 Apache Maven](https://maven.apache.org/download.cgi)`

下载要求的版本，比如3.8.1

2.在项目里面的conf中的setting替换为以下内容：注意其中存储依赖的位置改为自己的对方存储,可以随便创建，就是存储的位置而已




```
<?xml version="1.0" encoding="UTF-8"?>

<!--
Licensed to the Apache Software Foundation (ASF) under one
or more contributor license agreements.  See the NOTICE file
distributed with this work for additional information
regarding copyright ownership.  The ASF licenses this file
to you under the Apache License, Version 2.0 (the
"License"); you may not use this file except in compliance
with the License.  You may obtain a copy of the License at

    http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing,
software distributed under the License is distributed on an
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY
KIND, either express or implied.  See the License for the
specific language governing permissions and limitations
under the License.
-->

<!--
 | This is the configuration file for Maven. It can be specified at two levels:
 |
 |  1. User Level. This settings.xml file provides configuration for a single
user,
 |                 and is normally provided in ${user.home}/.m2/settings.xml.
 |
 |                 NOTE: This location can be overridden with the CLI option:
 |
 |                 -s /path/to/user/settings.xml
 |
 |  2. Global Level. This settings.xml file provides configuration for all Maven
 |                 users on a machine (assuming they're all using the same Maven
 |                 installation). It's normally provided in
 |                 ${maven.home}/conf/settings.xml.
 |
 |                 NOTE: This location can be overridden with the CLI option:
 |
 |                 -gs /path/to/global/settings.xml
```

```
 |
 | The sections in this sample file are intended to give you a running start at
 | getting the most out of your Maven installation. Where appropriate, the
default
 | values (values used when the setting is not specified) are provided.
 |
 |-->
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
http://maven.apache.org/xsd/settings-1.0.0.xsd">
  <!-- localRepository
   | The path to the local repository maven will use to store artifacts.
   |
   | Default: ${user.home}/.m2/repository
  <localRepository>/path/to/local/repo</localRepository>
  -->
      <localRepository>D:\maven\repository</localRepository>
  <!-- interactiveMode
   | This will determine whether maven prompts you when it needs input. If set
to false,
   | maven will use a sensible default value, perhaps based on some other
setting, for
   | the parameter in question.
   |
   | Default: true
  <interactiveMode>true</interactiveMode>
  -->

  <!-- offline
   | Determines whether maven should attempt to connect to the network when
executing a build.
   | This will have an effect on artifact downloads, artifact deployment, and
others.
   |
   | Default: false
  <offline>false</offline>
  -->

  <!-- pluginGroups
   | This is a list of additional group identifiers that will be searched when
resolving plugins by their prefix, i.e.
   | when invoking a command line like "mvn prefix:goal". Maven will
automatically add the group identifiers
   | "org.apache.maven.plugins" and "org.codehaus.mojo" if these are not already
contained in the list.
   |-->
  <pluginGroups>
    <!-- pluginGroup
     | Specifies a further group identifier to use for plugin lookup.
    <pluginGroup>com.your.plugins</pluginGroup>
    -->
  </pluginGroups>

  <!-- proxies
   | This is a list of proxies which can be used on this machine to connect to
the network.
```

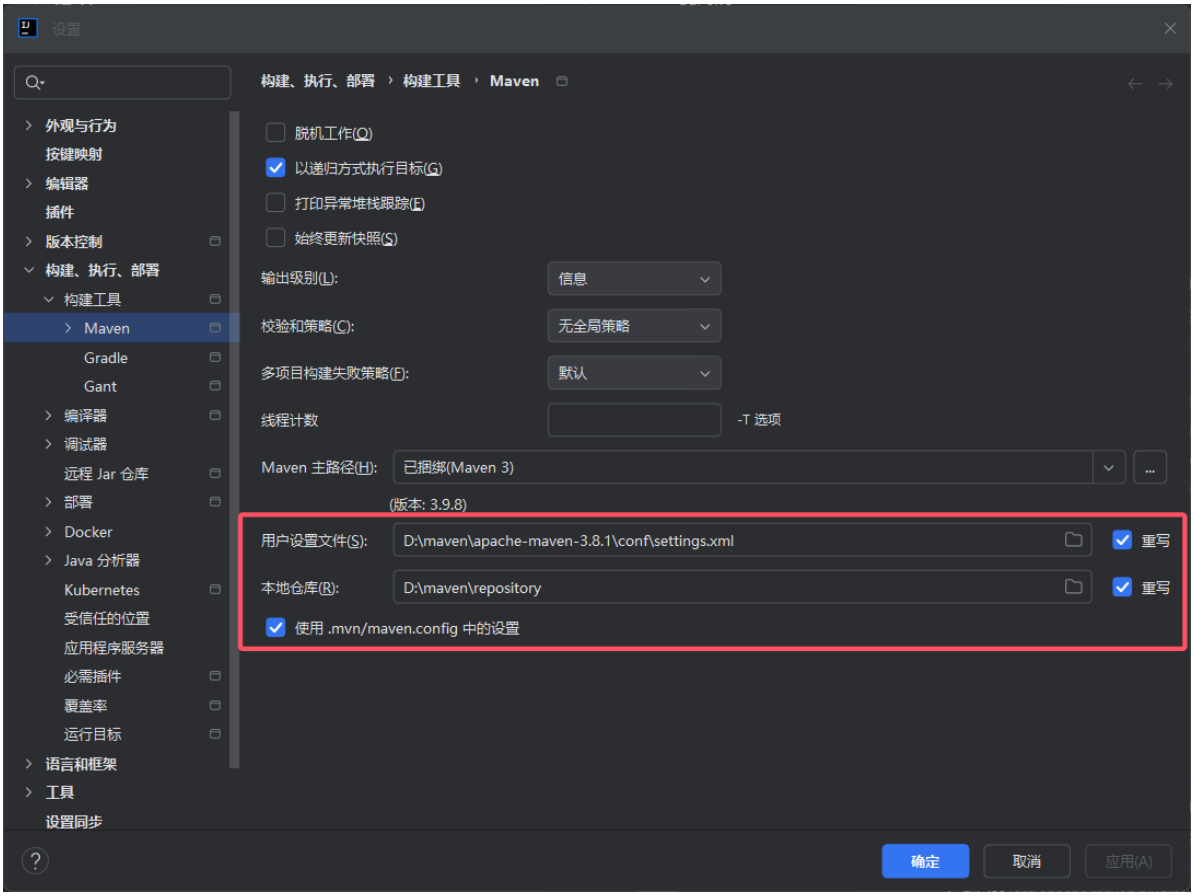
```
   | Unless otherwise specified (by system property or command-line switch), the
first proxy
   | specification in this list marked as active will be used.
   |-->
  <proxies>
    <!-- proxy
     | Specification for one proxy, to be used in connecting to the network.
     |
    <proxy>
      <id>optional</id>
      <active>true</active>
      <protocol>http</protocol>
      <username>proxyuser</username>
      <password>proxypass</password>
      <host>proxy.host.net</host>
      <port>80</port>
      <nonProxyHosts>local.net|some.host.com</nonProxyHosts>
    </proxy>
    -->
  </proxies>

  <!-- servers
   | This is a list of authentication profiles, keyed by the server-id used
within the system.
   | Authentication profiles can be used whenever maven must make a connection
to a remote server.
   |-->
  <servers>
    <!-- server
     | Specifies the authentication information to use when connecting to a
particular server, identified by
     | a unique name within the system (referred to by the 'id' attribute
below).
     |
     | NOTE: You should either specify username/password OR
privateKey/passphrase, since these pairings are
     |       used together.
     |
    <server>
      <id>deploymentRepo</id>
      <username>repouser</username>
      <password>repopwd</password>
    </server>
    -->

    <!-- Another sample, using keys to authenticate.
    <server>
      <id>siteServer</id>
      <privateKey>/path/to/private/key</privateKey>
      <passphrase>optional; leave empty if not used.</passphrase>
    </server>
    -->
  </servers>

  <!-- mirrors
   | This is a list of mirrors to be used in downloading artifacts from remote
repositories.
   |
```

```
   | It works like this: a POM may declare a repository to use in resolving
certain artifacts.
   | However, this repository may have problems with heavy traffic at times, so
people have mirrored
   | it to several places.
   |
   | That repository definition will have a unique id, so we can create a mirror
reference for that
   | repository, to be used as an alternate download site. The mirror site will
be the preferred
   | server for that repository.
   |-->
  <mirrors>
    <!-- mirror
     | Specifies a repository mirror site to use instead of a given repository.
The repository that
     | this mirror serves has an ID that matches the mirrorOf element of this
mirror. IDs are used
     | for inheritance and direct lookup purposes, and must be unique across the
set of mirrors.
     |
    <mirror>
      <id>mirrorId</id>
      <mirrorOf>repositoryId</mirrorOf>
      <name>Human Readable Name for this Mirror.</name>
      <url>http://my.repository.com/repo/path</url>
    </mirror>
     -->
     <mirror>
        <id>nexus-mirror</id>
        <mirrorOf>nexusjdk17</mirrorOf>
        <name>Human Readable Name for this Mirror.</name>
        <url>http://nexus.warmheart.top:9091/repository/maven-public/</url>
    </mirror>
    <mirror>
      <!--This is used to direct the public snapshots repo in the '
          profile below over to a different nexus group -->
      <id>nexus-public-snapshots</id>
      <mirrorOf>public-snapshots,!getui-nexus</mirrorOf>
      <url>http://maven.aliyun.com/nexus/content/repositories/snapshots/</url>
    </mirror>
    <mirror>
        <id>huaweicloud</id>
        <mirrorOf>*</mirrorOf>

<url>https://mirrors.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
    </mirror>
    <mirror>
        <id>huaweicloud</id>
        <mirrorOf>*</mirrorOf>

<url>https://mirrors.huaweicloud.com/repository/maven/huaweicloudsdk/</url>
    </mirror>
  </mirrors>

  <!-- profiles
   | This is a list of profiles which can be activated in a variety of ways, and
which can modify
```

```
   | the build process. Profiles provided in the settings.xml are intended to
provide local machine-
   | specific paths and repository locations which allow the build to work in
the local environment.
   |
   | For example, if you have an integration testing plugin - like cactus - that
needs to know where
   | your Tomcat instance is installed, you can provide a variable here such
that the variable is
   | dereferenced during the build process to configure the cactus plugin.
   |
   | As noted above, profiles can be activated in a variety of ways. One way -
the activeProfiles
   | section of this document (settings.xml) - will be discussed later. Another
way essentially
   | relies on the detection of a system property, either matching a particular
value for the property,
   | or merely testing its existence. Profiles can also be activated by JDK
version prefix, where a
   | value of '1.4' might activate a profile when the build is executed on a JDK
version of '1.4.2_07'.
   | Finally, the list of active profiles can be specified directly from the
command line.
   |
   | NOTE: For profiles defined in the settings.xml, you are restricted to
specifying only artifact
   |       repositories, plugin repositories, and free-form properties to be
used as configuration
   |       variables for plugins in the POM.
   |
   |-->
  <profiles>
    <profile>
        <id>nexusProfile17</id>
        <activation>
            <activeByDefault>true</activeByDefault>
            <jdk>1.8</jdk>
        </activation>

        <repositories>

            <repository>
                <id>nexusjdk17</id>
                <name>nexus-repository</name>
                <url>http://nexus-mirror</url>
                <releases>
                    <enabled>true</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
            </repository>

            <repository>
                <id>aliyunmaven</id>
                <url>http://nexus.warmheart.top:9091/repository/maven-
public/</url>
            </repository>
```

```
            <repository>
                <id>aliyun</id>
                <name>aliyun public</name>
                <url>https://maven.aliyun.com/repository/public</url>
                <releases>
                    <enabled>true</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
            </repository>

        </repositories>

        <pluginRepositories>
            <pluginRepository>
                <id>nexusjdk17</id>
                <name>nexus-plugin-repository</name>
                <url>http://nexus-mirror</url>
                <releases>
                    <enabled>true</enabled>
                </releases>
                <snapshots>
                    <enabled>true</enabled>
                </snapshots>
            </pluginRepository>
        </pluginRepositories>
        <properties>
            <maven.compiler.source>1.8</maven.compiler.source>
            <maven.compiler.target>1.8</maven.compiler.target>
            <maven.compiler.compilerVersion>1.8</maven.compiler.compilerVersion>
        </properties>
    </profile>
  </profiles>

  <!-- activeProfiles
   | List of profiles that are active for all builds.
   |
  <activeProfiles>
    <activeProfile>alwaysActiveProfile</activeProfile>
    <activeProfile>anotherAlwaysActiveProfile</activeProfile>
  </activeProfiles>
  -->
</settings>
```

3.在项目的maven中重写setting跟本地仓库的位置，指向刚才下载修改里面的



问题：wen端跟服务端都配置了。但是就是连接不通。

解决：是客户端路径的问题，因为服务端使用了websocket协议配置：
`socketChannel.pipeline().addLast(new WebSocketServerProtocolHandler("/ws"));`

后面配置了 `/ws`,所以客户端的路径要一致，原本的是`ws://localhost:80/`，现在要在后面添加 `/ws`，最终的为 `"ws://localhost:80/ws"`，卡好久

问题：传输的帧超过限定的65536长度，就是传输的太大了

解决：分块传输再整合

问题：在利用mybatisplus时往数据库新加数据不成功

解决：因为我在用户表中使用了groups的字样，这是一个保留关键字，所以失败修改列名就行

问题：根据提供的错误信息，问题出在 SQL 查询语句中提到的一个字段 `web_socket_uid` 不存在于数据库表 `user` 中。这意味着你在执行查询时尝试选择一个并不存在的列。

解决：**MyBatis-Plus 默认使用下划线命名法（snake_case）来映射数据库列名。**如果你的数据库列名使用驼峰命名法（camelCase），例如 `WebSocketUid`，而实体类中的字段名也是驼峰命名法，MyBatis-Plus 会自动将其转换为下划线命名法 `web_socket_uid`。

可以**显示指定列名**，`@TableField("WebSocketUid")//显示指定，不然会因为驼峰命名失效`：

注意数据库的列名有无多空格，是不是正确

问题：处理器不能被共享，不能频繁添加，因为我服务器处理器类使用了其他的Bean，肯定要把这个处理类也交给springboot管理，但是这样在我使用的时候，注入这个处理器类就是报错：

从错误信息来看，Netty 抛出了 `ChannelPipelineException`，提示 `Server.Handler` 不是一个可共享的处理器（`@Sharable`），因此不能多次添加或删除。

尝试添加`@Scope("prototype")`，使这个处理器每次创建新的，好像也不能解决。

解决：**使用上下文获取处理器实例+在处理器声明创建模式为原型创建，即@Scope("prototype")**

```
 @Autowired
    private  ApplicationContext applicationContext;//通过上下文每次获取新的处理器实例，处理器类也要添加@Scope("prototype")，指定创建模式为原型
```

问题：在创建记录历史记录的表并且指定分区时出错。

原因：我在分区使用了TO_DAYS(sent_at)函数将时间戳转化为日期，但是`TO_DAYS` 函数返回的是一个常量表达式。

在MySQL中，使用分区表时，分区函数中不允许使用常量、随机表达式或与时间区相关的表达式。这是因为分区函数需要能够确定地计算出分区键值，以便在插入和查询时能够正确地分配和定位数据。

```
#指定范围(range)作为分区
    PARTITION BY RANGE (TO_DAYS(sent_at)) (
        #具体的分区PARTITION,按月份来分区,less then表示少于这个分区时间的消息都会被存储在这个分区
        PARTITION p2024_10 VALUES LESS THAN (TO_DAYS('2024-10-01')),
        PARTITION p2024_11 VALUES LESS THAN (TO_DAYS('2024-11-01')),
        PARTITION p2024_12 VALUES LESS THAN (TO_DAYS('2024-12-01')),
        PARTITION p2025_01 VALUES LESS THAN (TO_DAYS('2025-01-01')),
        PARTITION p_future VALUES LESS THAN MAXVALUE#表示其他消息存储的分区,超过上面的时间的所有消息存储位置
        );
```

解决：使用使用 `UNIX_TIMESTAMP` 函数来替代 `TO_DAYS`

```
#指定范围(range)作为分区
    PARTITION BY RANGE (UNIX_TIMESTAMP(sent_at)) (
        #具体的分区PARTITION,按月份来分区,less then表示少于这个分区时间的消息都会被存储在这个分区
        PARTITION p2024_10 VALUES LESS THAN (UNIX_TIMESTAMP('2024-10-01')),
        PARTITION p2024_11 VALUES LESS THAN (UNIX_TIMESTAMP('2024-11-01')),
        PARTITION p2024_12 VALUES LESS THAN (UNIX_TIMESTAMP('2024-12-01')),
        PARTITION p2025_01 VALUES LESS THAN (UNIX_TIMESTAMP('2025-01-01')),
        PARTITION p_future VALUES LESS THAN MAXVALUE
        );
```

问题：A PRIMARY KEY must include all columns in the table's partitioning function (prefixed columns are not considered).

原因：还是sql分区的问题，在mysql中，分区使用到的列一定要包含在主键中，比如使用时间戳来分区，那么时间戳应该跟id一起组成主键。

解决：如上

问题;在创建接收者消息表时，无法将消息主表的id作为外键，

原因：MySQL 不支持在分区表中使用外键

解决：移除外键吧

问题：在记录历史消息的表中为了分区使用了复合主键，但是mybatisplus自动生成的默认单个主键，设置多个就报错。

解决：**设置复合主键类**,等会，好像我把发送时间设置为简单的列也行，哈哈哈

问题：在java中保存base64的图片时出错

原因:在java中，对于base编码的图片，使用`Base64.getDecoder().decode(data);`解码，但是需要注意，要去掉前缀标识，`""`

重要

问题：此站点的连接不安全

localhost 发送了无效的响应。

ERR_SSL_PROTOCOL_ERROR

原因：前端使用https，但是后端没有配置ssl证书.

问题：video.js:346 WebSocket connection to 'wss://47.102.195.246:8443/ws' failed:

再配置ssl证书时遇到，是把后端部署到一个服务器，然后在其他对方启动js文件来连接时遇到。不再是原来的前端页面也在一台服务器上，这样会不会导致跨域问题。看下面**原因2**

WebSocket连接的握手过程是通过HTTP请求完成的。浏览器会发送一个HTTP请求到WebSocket服务器，服务器需要在响应中包含适当的CORS头，以允许跨域请求。

问题：在局域网环境中，在手机上能访问到springboot的页面，但是js静态资源出错。点击按钮无反应。这是为什么。

原因1：如果不是使用norok的链接，而是直接访问主机IP地址，此时是http链接，视频跟音频属于敏感信息，浏览器只会允许在https请求中使用.只能使用ngrok的链接访问。但是此时又导致一个问题，就是混合内容

原因2：可能是跨域问题,(局域网好像又不是)如果不定义自定义的处理器来处理跨域请求，Netty 的 默认 WebSocket 服务器将默认只允许同源请求。这意味着只有在同一域名和端口下的客户端才能成功连接到 WebSocket 服务器。其他域名或端口的客户端将会因为浏览器的同源策略而被阻止连接。

因为在netty的流水线只添加了下面，这个默认只允许同源请求。

```
 //websocket协议，使用/ws路径，也就是js中ip的前缀
                        socketChannel.pipeline().addLast(new
WebSocketServerProtocolHandler("/ws"));
```

问题： `certificate_unknown` **错误**，通常意味着客户端不信任服务器提供的 SSL 证书

原因：这个也会造成连接失败，video.js:346 WebSocket connection to 'wss://47.102.195.246:8443/ws' failed:

解决：因为是直接通过ip地址访问，而ssl证书是依托域名的，所有按照域名进行连接,不能再通过ip地址，因为前后端不是在一个机子上了。

```
websocket=new WebSocket("wss://test-sami-ws.warmheart.top:8443/ws");//公司服务器
```

问题：lombok注解的日志log使用不了

原因：不清楚

解决：多clean几遍就可以顺利运行，没遇到过。

问题：