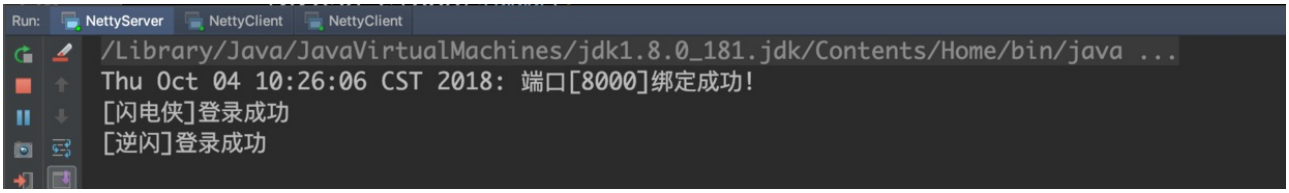
# 实战：客户端互聊原理与实现

> 之前写过一篇非严肃的[微信单聊原理(https://mp.weixin.qq.com/s?__biz=MzI1OTUzMTQyMA==&mid=2247484094&idx=1](https://mp.weixin.qq.com/s?__biz=MzI1OTUzMTQyMA==&mid=2247484094&idx=1)得到广大网友的一致好评，有很多读者留言问我如何使用 Netty 来具体实现这个逻辑，学完本小节，你会发现其实很简单。

在开始本小节之前，我们先来看一下本小节学完之后，单聊的实现的效果是什么样的?

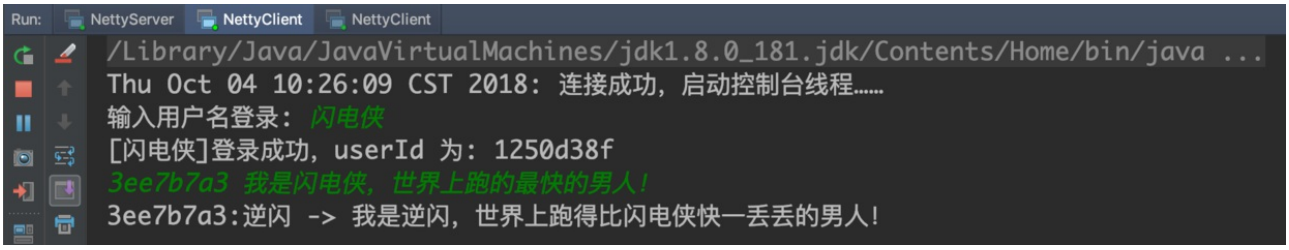
## 1. 最终效果

> 服务端

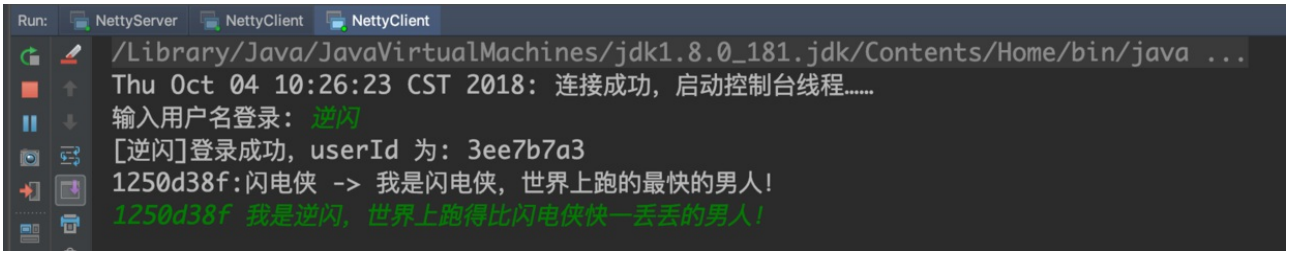



服务端启动之后，两个客户端陆续登录

> 客户端 1




客户端 2



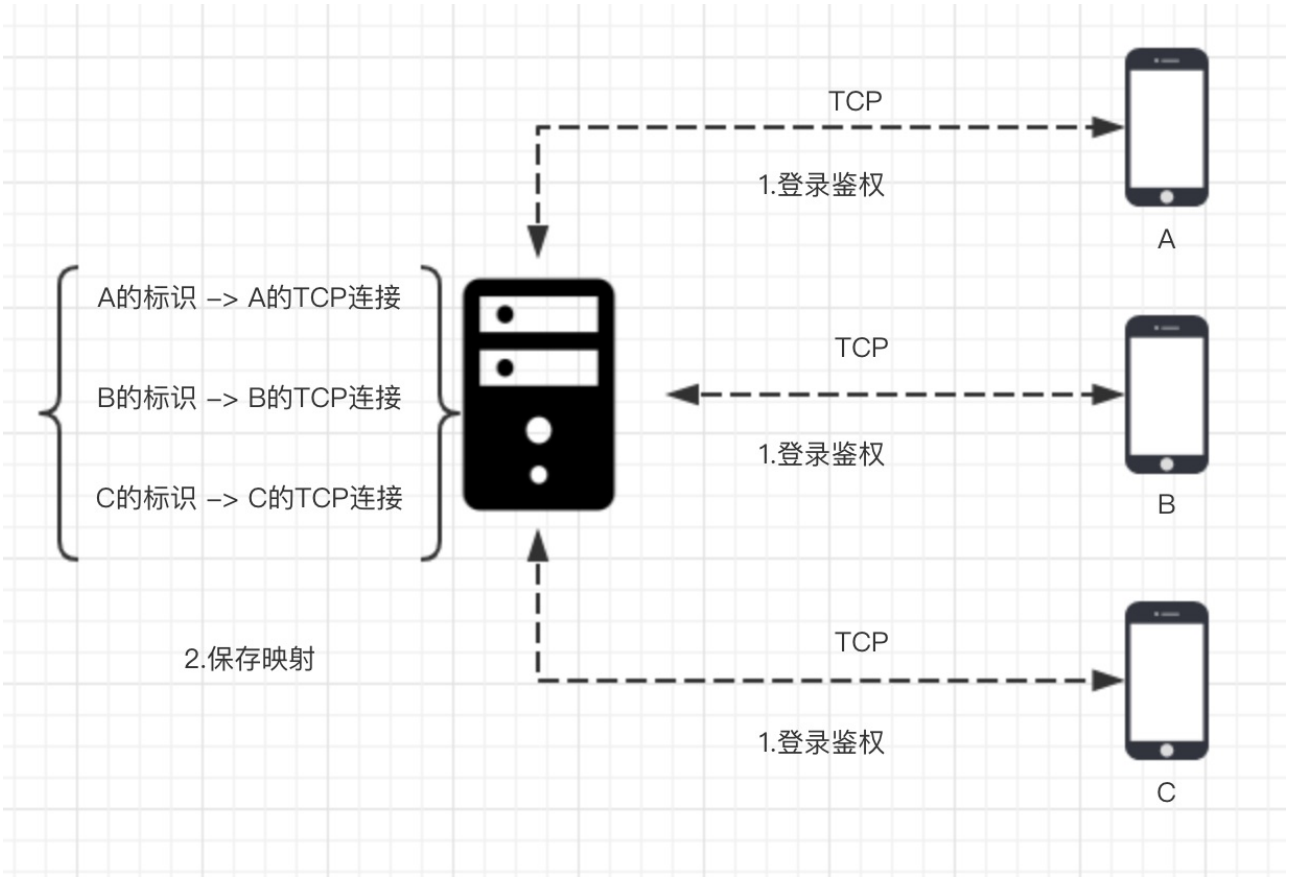

1. 客户端启动之后，我们在控制台输入用户名，服务端随机分配一个 userId 给客户端，这里我们省去了通过账号密码注册的过程，userId 就在服务端随机生成了，生产环境中可能会持久化在数据库，然后每次通过账号密码去“捞”。

2. 当有两个客户端登录成功之后，在控制台输入userId + 空格 + 消息，这里的 userId 是消息接收方的标识， 消息接收方的控制台接着就会显示另外一个客户端发来的消息。

一对一单聊的本质其实就这么简单，稍加改动其实就可以用在生产环境下，下面，我们就来一起学习一下如何实现控制台一对一单聊

# 2. 一对一单聊原理

一对一单聊的原理我们在 [仿微信 IM 系统简介 (https://juejin.im/book/5b4bc28bf265da0f60130116/sectior](https://juejin.im/book/5b4bc28bf265da0f60130116/sectior) 已经学习过，我们再来重温一下




1. 如上图，A 要和 B 聊天，首先 A 和 B 需要与服务器建立连接，然后进行一次登录流程，服务端保存用户标识和 TCP 连接的映射关系。
2. A 发消息给 B，首先需要将带有 B 标识的消息数据包发送到服务器，然后服务器从消息数据包中拿到 B 的标识，找到对应的 B 的连接，将消息发送给 B。

原理掌握之后，接下来我们就来逐个实现这里面的逻辑

# 3. 一对一单聊实现

## 3.1 用户登录状态与 channel 的绑定

我们先来看一下，服务端在单聊实现中是如何处理登录消息的

| LoginRequestHandler.java |
| --- |

```
// 我们略去了非关键部分的代码，详细可以本地更新下代码，切换到本小节名称对应的 git 分支
protected void channelRead0(ChannelHandlerContext ctx, LoginRequestPacket loginRequestPacket) {
    LoginResponsePacket loginResponsePacket = xxx;
    String userId = randomUserId();
    loginResponsePacket.setUserId(userId);
    SessionUtil.bindSession(new Session(userId, loginRequestPacket.getUserName()), ctx.channel());

    // 登录响应

    ctx.channel().writeAndFlush(loginResponsePacket);
}

// 用户断线之后取消绑定
public void channelInactive(ChannelHandlerContext ctx) {
    SessionUtil.unBindSession(ctx.channel());
}
```

登录成功之后，服务端创建一个 Session 对象，这个对象表示用户当前的会话信息，在我们这个应用程序里面，Session 只有两个字段

Session.java

```
public class Session {
    // 用户唯一性标识
    private String userId;
    private String userName;
}
```

实际生产环境中 Session 中的字段可能较多，比如头像 url，年龄，性别等等。

然后，我们调用 SessionUtil.bindSession() 保存用户的会话信息，具体实现如下

SessionUtil.java

```
public class SessionUtil {
    // userId -> channel 的映射
    private static final Map<String, Channel>
userIdChannelMap = new ConcurrentHashMap<>();


    public static void bindSession(Session
session, Channel channel) {
        userIdChannelMap.put(session.getUserId(),
channel);

channel.attr(Attributes.SESSION).set(session);
    }

    public static void unBindSession(Channel
channel) {
        if (hasLogin(channel)) {
```

```
userIdChannelMap.remove(getSession(channel).getUs
erId());

channel.attr(Attributes.SESSION).set(null);
        }
    }

    public static boolean hasLogin(Channel
channel) {

        return
channel.hasAttr(Attributes.SESSION);
    }

    public static Session getSession(Channel
channel) {

        return
channel.attr(Attributes.SESSION).get();
    }

    public static Channel getChannel(String
userId) {

        return userIdChannelMap.get(userId);
    }
}
```

1. SessionUtil 里面维持了一个 useId -> channel 的映射 map，调用 bindSession() 方法的时候，在 map 里面保存这个映射关系，SessionUtil 还提供了 getChannel() 方法，这样就可以通过 userId 拿到对应的 channel。
2. 除了在 map 里面维持映射关系之外，在 bindSession() 方

法中，我们还给 channel 附上了一个属性，这个属性就是当前用户的 Session，我们也提供了 getSession() 方法，非常方便地拿到对应 channel 的会话信息。
3. 这里的 SessionUtil 其实就是前面小节的 LoginUtil，这里重构了一下，其中 hasLogin() 方法，只需要判断当前是否有用户的会话信息即可。
4. 在 LoginRequestHandler 中，我们还重写了 channelInactive() 方法，用户下线之后，我们需要在内存里面自动删除 userId 到 channel 的映射关系，这是通过调用 SessionUtil.unBindSession() 来实现的。

关于用户会话信息的保存的逻辑其实就这么多，总结一点就是：登录的时候保存会话信息，登出的时候删除会话信息，接下来，我们就来实现服务端接收消息并转发的逻辑。

## 3.2 服务端接收消息并转发的实现

我们重新来定义一下客户端发送给服务端的消息的数据包格式

MessageRequestPacket.java

```
public class MessageRequestPacket extends Packet
{
    private String toUserId;
    private String message;
}
```

数据包格式很简单，toUserId 表示要发送给哪个用户，message 表示具体内容，接下来，我们来看一下服务端的消息处理 handler 是如何来处理消息的

MessageRequestHandler.java

```
public class MessageRequestHandler extends
SimpleChannelInboundHandler<MessageRequestPacket>
 {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
MessageRequestPacket messageRequestPacket) {
        // 1.拿到消息发送方的会话信息
        Session session =
SessionUtil.getSession(ctx.channel());

        // 2.通过消息发送方的会话信息构造要发送的消息
        MessageResponsePacket
messageResponsePacket = new
MessageResponsePacket();

messageResponsePacket.setFromUserId(session.getUs
erId());

messageResponsePacket.setFromUserName(session.get
UserName());

messageResponsePacket.setMessage(messageRequestPa
cket.getMessage());

        // 3.拿到消息接收方的 channel
        Channel toUserChannel =
SessionUtil.getChannel(messageRequestPacket.getTo
UserId());
```

```
        // 4.将消息发送给消息接收方
        if (toUserChannel != null &&
SessionUtil.hasLogin(toUserChannel)) {

toUserChannel.writeAndFlush(messageResponsePacket
);
        } else {
            System.err.println("[" +
messageRequestPacket.getToUserId() + "] 不在线，发
送失败!");
        }
    }
}
```

1. 服务端在收到客户端发来的消息之后，首先拿到当前用户，也就是消息发送方的会话信息。
2. 拿到消息发送方的会话信息之后，构造一个发送给客户端的消息对象 MessageResponsePacket，填上发送消息方的用户标识、昵称、消息内容。
3. 通过消息接收方的标识拿到对应的 channel。
4. 如果消息接收方当前是登录状态，直接发送，如果不在线，控制台打印出一条警告消息。

这里，服务端的功能相当于消息转发：收到一个客户端的消息之后，构建一条发送给另一个客户端的消息，接着拿到另一个客户端的 channel，然后通过 writeAndFlush() 写出。接下来，我们再来看一下客户端收到消息之后的逻辑处理。

## 3.3 客户端收消息的逻辑处理

MessageResponseHandler.java

```
public class MessageResponseHandler extends
SimpleChannelInboundHandler<MessageResponsePacket
> {
    @Override
    protected void
channelRead0(ChannelHandlerContext ctx,
MessageResponsePacket messageResponsePacket) {
        String fromUserId =
messageResponsePacket.getFromUserId();
        String fromUserName =
messageResponsePacket.getFromUserName();
        System.out.println(fromUserId + ":" +
fromUserName + " -> " + messageResponsePacket
.getMessage());
    }
}
```

客户端收到消息之后，只是把当前消息打印出来，这里把发送方的用户标识打印出来是为了方便我们在控制台回消息的时候，可以直接复制 ^ ^，到了这里，所有的核心逻辑其实已经完成了，我们还差最后一环：在客户端的控制台进行登录和发送消息逻辑。

## 3.4 客户端控制台登录和发送消息

我们回到客户端的启动类，改造一下控制台的逻辑

NettyClient.java

```
private static void startConsoleThread(Channel
channel) {
    Scanner sc = new Scanner(System.in);
    LoginRequestPacket loginRequestPacket = new
```

```
LoginRequestPacket();

    new Thread(() -> {
        while (!Thread.interrupted()) {
            if (!SessionUtil.hasLogin(channel)) {
                System.out.print("输入用户名登录:
");
                String username = sc.nextLine();

loginRequestPacket.setUserName(username);

                // 密码使用默认的

loginRequestPacket.setPassword("pwd");

                // 发送登录数据包

channel.writeAndFlush(loginRequestPacket);
                waitForLoginResponse();
            } else {
                String toUserId = sc.next();
                String message = sc.next();
                channel.writeAndFlush(new
MessageRequestPacket(toUserId, message));
            }
        }
    }).start();
}

private static void waitForLoginResponse() {
    try {
        Thread.sleep(1000);
    } catch (InterruptedException ignored) {
```

```
    }
}
```

我们在客户端启动的时候，起一个线程

1. 如果当前用户还未登录，我们在控制台输入一个用户名，然后构造一个登录数据包发送给服务器，发完之后，我们等待一个超时时间，可以当做是登录逻辑的最大处理时间，这里就简单粗暴点了。
2. 如果当前用户已经是登录状态，我们可以在控制台输入消息接收方的 userId，然后输入一个空格，再输入消息的具体内容，然后，我们就可以构建一个消息数据包，发送到服务端。

关于单聊的原理和实现，这小节到这里就结束了，最后，我们对本小节内容做一下总结。

# 总结

1. 我们定义一个会话类 Session 用户维持用户的登录信息，用户登录的时候绑定 Session 与 channel，用户登出或者断线的时候解绑 Session 与 channel。
2. 服务端处理消息的时候，通过消息接收方的标识，拿到消息接收方的 channel，调用 writeAndFlush() 将消息发送给消息接收方。

# 思考

我们在本小节其实还少了用户登出请求和响应的指令处理，你是否能说出，对登出指令来说，服务端和客户端分别要干哪些事情？是否能够自行实现？

欢迎留言一起讨论，具体实现也会在下小节对应的代码分支上放出，读者可先自行尝试下实现。