33 Mock客户端实现，四个人来一局

更新时间：2020-08-24 10:39:55

> 上天赋予的生命，就是要为人类的繁荣和平和幸福而奉献。——松下幸之助

前言

你好，我是彤哥。

上一节，我们一起学习了游戏线程池的设计，并实现了主要的业务逻辑，至此，服务端算是实现完毕了。

然而，没有客户端的服务端是不完美的，因此，本节，我们将一起实现一个模拟客户端，用它来进行调试。

最后，我们也会启动四个客户端真正地打一局。

OK，进入今天的学习吧。

Mock 客户端实现

实现前

首先，我们分析一下一个 Mock 客户端它应该具备的基本功能：

1. 对于服务端返回的消息，能够友好地显示，何为友好地显示？即不要直接显示消息本身，应该做一些格式化。
2. 对于需要用户操作的内容，不能太复杂，最好能使用 1、2、3、4、5 很简单的数字代替，比如出牌，如果还要输入汉字，就会比较繁琐，交互也不好。

好了，能实现这两个功能基本上差不多了。

接下来，应该以什么样的方式呈现呢？

用命令行显示？

用 java swing 开发一个图形界面？

用客户端语言，比如 lua 或者 javascript，开发一个图形界面？

显然，后面两种方式体验更好，但是，开发成本太高，而且，对于后端程序员，用命令行的方式似乎也不错，因此，我们决定使用命令行的方式。

最后，我们需要调研一下，使用命令行如何进行交互？

在 Java 中，可以使用 `System.in.read()` 读取命令行中的输入，或者使用 Scanner 类对 System.in 进行包装也是可以的，相对来说，使用 Scanner 更简单一些，所以，我们选择使用 Scanner 类来读取用户输入。

好了，实现前我们已经确定了基本功能、呈现方式、交互方式，下面就是真刀真枪地干了。

实现中

对于客户端收到的消息，无疑是要把它渲染出来，并提供一些选项供用户操作，其实，整个过程跟服务端处理消息的逻辑是比较相像的，所以，我们仿造服务端处理消息的过程，抽象出来一个 MahjongRender 接口，不同的消息实现各自的渲染方法：

```
public interface MahjongRender<T extends MahjongMessage> {
    void render(T message);
}
```

比如，对于 HelloResponse 这个消息，我们可以这样来实现：

```
public class HelloResponseRender implements MahjongRender<HelloResponse> {
    @Override
    public void render(HelloResponse message) {
        System.out.println(helloResponse);
    }
}
```

当然，我们还需要维护消息与渲染器之间的关系：

```
public enum MahjongRenderManager {
    HELLO_RESPONSE_RENDER(HelloResponse.class, new HelloResponseRender()),
    ;
    private Class<? extends MahjongMessage> msgType;
    private MahjongRender mahjongRender;

    MahjongRenderManager(Class<? extends MahjongMessage> msgType, MahjongRender mahjongRender) {
        this.msgType = msgType;
        this.mahjongRender = mahjongRender;
    }

    public static MahjongRender choose(MahjongMessage message) {
        // 通过消息寻找它的渲染器
        for (MahjongRenderManager value : MahjongRenderManager.values()) {
            if (value.msgType == message.getClass()) {
                return value.mahjongRender;
            }
        }
        return null;
    }
}
```

同样地， 对于渲染器，我们使用的也是单例模式。

这样，我们就可以在 MahjongClientHandler 中确定哪个消息使用哪个渲染器来处理了：

```
@Slf4j
public class MahjongClientHandler extends SimpleChannelInboundHandler<MahjongProtocol> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MahjongProtocol mahjongProtocol) throws Exception {
        // 协议头
        MahjongProtocolHeader header = mahjongProtocol.getHeader();
        // 协议体
        MahjongMessage message = (MahjongMessage) mahjongProtocol.getBody();
        // 查找渲染器
        MahjongRender mahjongRender = MahjongRenderManager.choose(message);
        if (mahjongRender != null) {
            MahjongContext.currentContext().setCurrentChannel(ctx.channel());
            mahjongRender.render(message);
        } else {
            log.error("not found render, msgType={}", message.getClass().getName());
        }
    }
}
```

为了方便，我又定义了一个类 MockClient，把实际的渲染全部放在了这个类中，比如，对于 HelloResponse，它的渲染器，只做一层简单的转发，实际的工作是在 MockClient 中：

```
public class HelloResponseRender implements MahjongRender<HelloResponse> {
    @Override
    public void render(HelloResponse message) {
        MockClient.helloResponse(message);
    }
}
public class MockClient {

    public static void helloResponse(HelloResponse helloResponse) {
        System.out.println(helloResponse);
    }
}
```

好了，到这里，Mock 客户端实现的基本步骤大家都比较清楚了，下面就是编写各种各样的渲染器，并把它们转发到 MockClient 中就可以了，比如，对于登录，我是这么来实现的。

首先，在客户端启动之后，给服务端发送一条 HelloRequest 的消息，隔 2 秒之后，提示用户登录：

```
public class MockClient {

    private static final String MOCK_USER = "\r\n\r\n[mahjong@mock]$ ";

    // 用于读取用户输入
    private static Scanner scanner = new Scanner(System.in);
    // 当前玩家
    private static Player player;
    // 当前房间
    private static Room room;

    public static void start(Channel channel) {
        MahjongContext.currentContext().setCurrentChannel(channel);
        // 发送hello消息
        HelloRequest helloRequest = new HelloRequest();
        helloRequest.setName("彤哥");
        MessageUtils.sendRequest(helloRequest);

        // 停顿2秒
        LockSupport.parkNanos(TimeUnit.SECONDS.toNanos(2));

        // 请登录
        login();
    }

    public static void helloResponse(HelloResponse helloResponse) {
        System.out.println(helloResponse);
    }

    private static void login() {
        oneOperation("\r\n请输入您的用户名和密码，以空格分隔：", line -> {
            if (!line.contains(" ")) {
                return false;
            }
            String[] arr = line.split(" ");
            // 发送登录消息
            LoginRequest request = new LoginRequest();
            request.setUsername(arr[0]);
            request.setPassword(arr[1]);
            MessageUtils.sendRequest(request);
            return true;
        });
    }

    private static void oneOperation(String tips, Command command) {
        // 打印提示
        System.out.print(tips + MOCK_USER);
        while (scanner.hasNextLine()) {
            // 读取用户输入
            String line = scanner.nextLine();
            // 处理失败，则提示重新输入
            if (!command.run(line)) {
                System.out.println("\r\n错误的输入，请重新输入：");
            } else {
                // 处理成功，跳出循环
                break;
            }
        }
    }

    public static void loginResponse(LoginResponse message) {
        // 登录成功
```

```
        if (message.isResult()) {
            System.out.println("\r\n登录成功，您的信息为：");
            System.out.println(player = message.getPlayer());
            // 登录成功，选择创建房间还是加入房间
            createOrEnterRoom();
        } else {
            // 登录失败
            System.out.println("\r\n登录失败：" + message.getMessage() + "，请您重新登录。");
            login();
        }
    }
}
```

OK，至此，登录的过程就处理完了，登录完成之后，就是提示用户创建房间还是加入房间了，后面都是一样的实现过程，不同之处在于每个消息处理的细节。

经过一番战斗，终于把所有的消息渲染过程都实现了，最后，别忘了维护消息与渲染器的关系：

```
public enum MahjongRenderManager {
    HELLO_RESPONSE_RENDER(HelloResponse.class, new HelloResponseRender()),
    LOGIN_RESPONSE_RENDER(LoginResponse.class, new LoginResponseRender()),
    CREATE_ROOM_RESPONSE_RENDER(CreateRoomResponse.class, new CreateRoomResponseRender()),
    ENTER_ROOM_RESPONSE_RENDER(EnterRoomResponse.class, new EnterRoomResponseRender()),
    ROOM_REFRESH_NOTIFICATION_RENDER(RoomRefreshNotification.class, new RoomRefreshNotificationRender()),
    OPERATION_NOTIFICATION_RENDER(OperationNotification.class, new OperationNotificationRender()),
    OPERATION_RESULT_NOTIFICATION_RENDER(OperationResultNotification.class, new OperationResultNotificationRender()),
    GAME_OVER_NOTIFICATION_RENDER(GameOverNotification.class, new GameOverNotificationRender()),
    SETTLEMENT_NOTIFICATION_RENDER(SettlementNotification.class, new SettlementNotificationRender()),
    ;
}
```

OK，在所有消息都渲染完毕之后，是时候来打一局了。

实现后

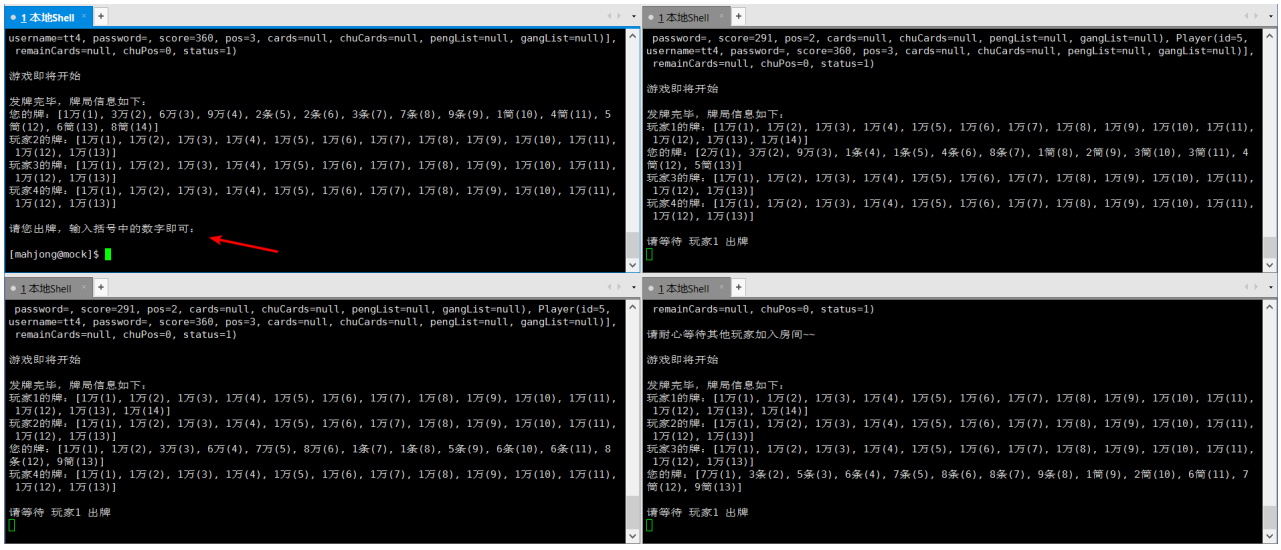
为了能在 XSHELL 中进行调试（cmd 也是可以的），可以添加 spring-boot 的插件，将程序打包成 jar 包，当然，我们这里只需要打包客户端的代码就可以了，服务端还是在 IDEA 中启动：

```
<properties>
    <start-class>com.imooc.netty.mahjong.client.MahjongClient</start-class>
</properties>
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
            <configuration>
                <mainClass>${start-class}</mainClass>
            </configuration>
            <executions>
                <execution>
                    <goals>
                        <goal>repackage</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

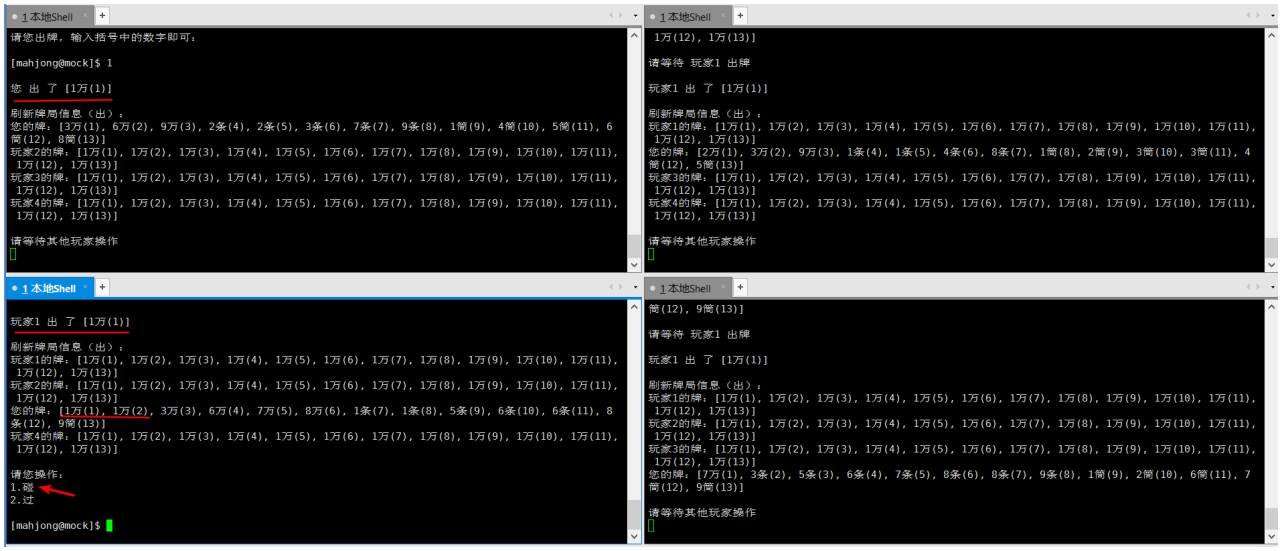
好了，先启动一个客户端来看看效果：

```
[G:\workspace\netty-mahjong\netty-mahjong-1.0\target]$ java -jar netty-mahjong-1.0-1.0-SNAPSHOT.jar
08:13:44 [main] MahjongClient: connect to server success
HelloResponse(message=你好，彤哥)

请输入您的用户名和密码，以空格分隔：

[mahjong@mock]$ tt1 aa

登录成功，您的信息为：
Player(id=5, username=tt1, password=, score=889, pos=0, cards=null, chuCards=null, pengList=null, gan
gList=null)

请选择您要进行的操作：
1.创建房间
2.加入房间

[mahjong@mock]$ 1

请输入房间底分以及人数，以空格隔开：

[mahjong@mock]$ 5 4

创建房间成功

房间信息如下：
Room(id=7, maxPlayerNum=4, baseScore=5, players=[Player(id=5, username=tt1, password=, score=889, pos
=0, cards=null, chuCards=null, pengList=null, gangList=null), null, null, null], remainCards=null, ch
uPos=0, status=1)

请耐心等待其他玩家加入房间~~
```

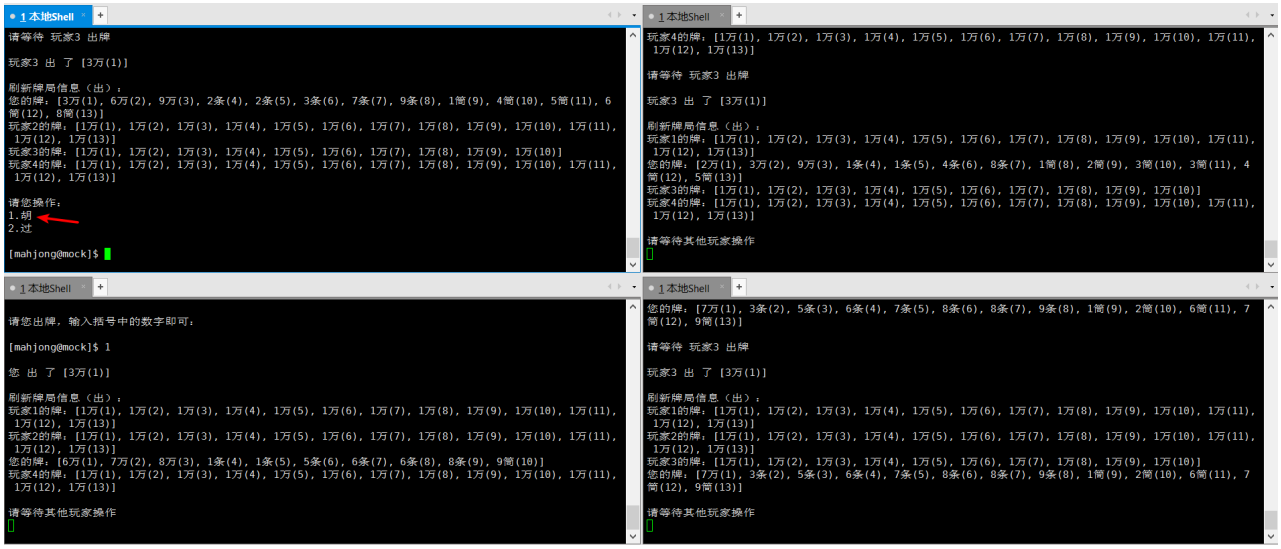
似乎还不错，再启动三个客户端来加入游戏吧：



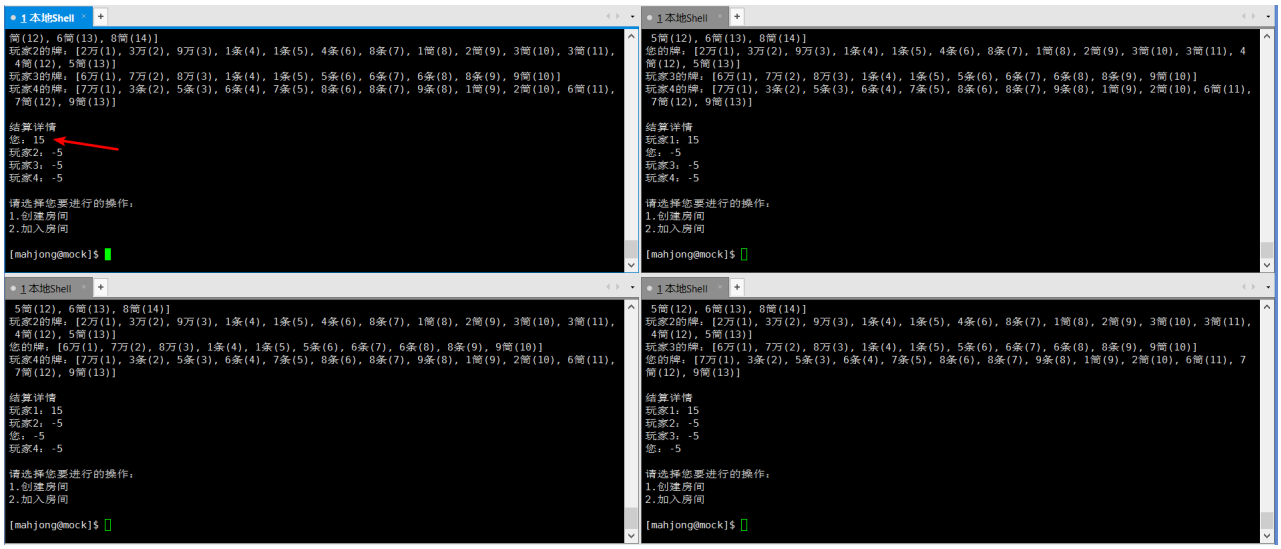
其它玩家的牌全部被隐藏成 1 万了，第一个玩家打个 1 万看看：



正好玩家 3 可以碰，碰一个嘚：



玩家 1 可以胡了，果断胡之，当然，这里的胡牌算法并没有真正的实现，而是使用的随机算法模拟胡牌算法：



玩家 1 赢了 15 分，其他玩家各输了 5 分，游戏结束。

好了，到这里，Mock 客户端就实现完毕了，通过 Mock 客户端，在命令行打牌，终于可以在上班时间愉快地划水了，关键是还不容易被发现，哈哈～～

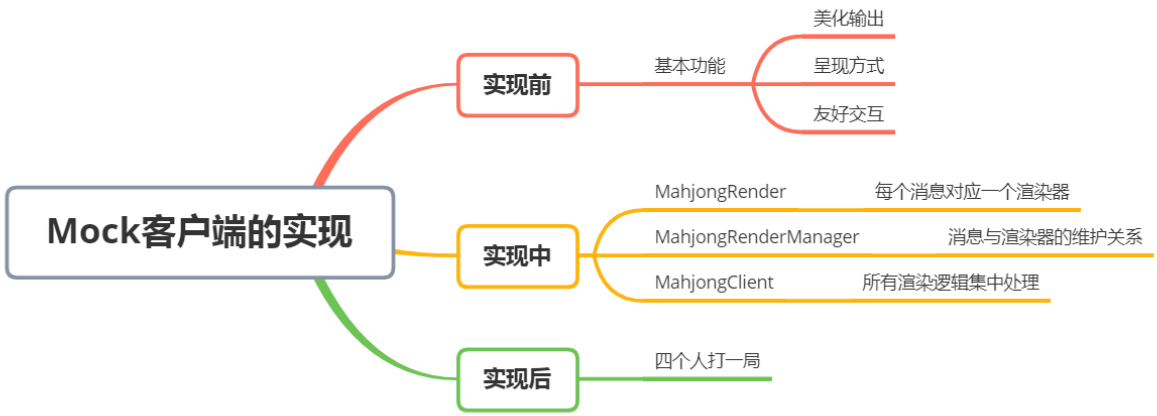
后记

本节，我们一起实现了 Mock 客户端，并通过 Mock 客户端真正地打了一局麻将，到这里， 整个的实战项目也算实现完毕了，但是，体验还不是特别好，比如，打完一局为什么要退出而不是继续游戏，有了前面的学习，我相信这些问题对你来说肯定可以轻松地搞定了。

不过，除了业务逻辑上的毛病，还有一些其他方面的问题，比如安全性、监控、调优等等，这些也是生产环境始终困扰我们的问题，对于这些问题，我们还是要解决的。

好了，下一节我们将进入 “实战进阶” 的篇章，将从优化的角度对本次实战项目再做一次升华，敬请期待。

思维导图



}

32 业务逻辑实现，游戏线程池如何设计？

34 如何支持Protobuf