30 协议实现，双端打通

更新时间：2020-08-20 09:38:01

> 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，我们已经从技术选型、领域模型设计、接口设计、部署架构设计等各个方面将系统规划好了。

但是，没有实现的系统是不完美的，毕竟，我们不是 PPT 架构师。

本节，我们就基于系统设计来实现我们的系统，对于本次实战项目的实现，我想通过下面四个部分来讲解：

1. 协议实现，双端打通：根据技术选型的设计，将协议和编解码实现，并打通客户端和服务端，这样在编写代码的过程中随时可以进行一些简单的调试。
2. 领域模型实现：主要是参考领域模型的设计，将这些领域模型都定义好一个个的 Java 对象，同时，我也会把消息的定义也放在这一节，当然了，我们这里使用的也是贫血模型，为什么使用贫血模型呢？彼时，我会详细说明。
3. 业务逻辑实现：主要是服务端如何对这些消息进行处理，如何设计线程模型，加速服务端处理等。
4. Mock 客户端实现：编写一个 Mock 客户端，并自测，四个客户端一桌麻将，妥妥的。

为了减少一节的内容过大，我会将实现分成四个小节，分别对应上面的内容。

好了，下面正式进入今天的学习 —— 协议实现，双端打通。

协议实现

上一节，我们已经将协议规划好了，协议分为 Header 和 Body，Header 中主要存储版本号、请求 ID、命令字，这里的命令字又可以称为操作码或者序列化类型等，主要是用来反查 Body 的真正类型，对于每一条消息，它们的命令字必须保证唯一性。

因此，我们可以定义协议如下：

```
@Data
public final class MahjongProtocol {
    /**
     * 协议头
     */
    private MahjongProtocolHeader header;
    /**
     * 协议体
     */
    private MahjongProtocolBody body;
}
@Data
public final class MahjongProtocolHeader {
    /**
     * 版本号
     */
    private int version;
    /**
     * 命令字
     */
    private int cmd;
    /**
     * 请求ID
     */
    private int reqId;
}
interface MahjongProtocolBody {
}
public interface MahjongMessage extends MahjongProtocolBody {
}
```

> Mahjong，麻将的意思。

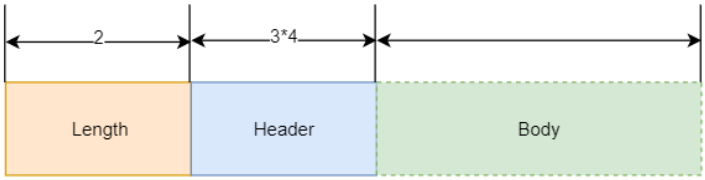
可以看到，对于协议本身和协议头，我们定义为 final 类型，而对于协议体，我们定义为接口，协议体为什么要定义为接口呢？

其实，这里的协议体，就是我们所说的消息，所有的消息都应该实现该接口，这么做的目的是为了实现统一处理，其实，这是运用了多态的思想。

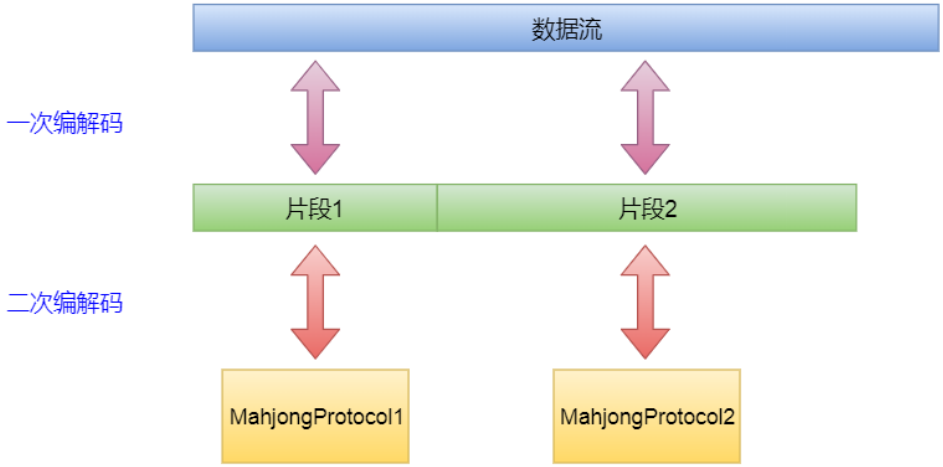
为了从命名上更直观，我又定义了一个接口 `MahjongMessage` 继承自 `MahjongProtocolBody`，这样所有的消息都实现自 `MahjongMessage`，更清晰。

好了，协议定义完毕，下面我们来一起看看如何对这个协议进行编解码。

编解码实现



我们一再强调，编解码分成一次编解码和二次编解码，一次编解码是对粘包半包的处理，将字节流分割成一个一个的片段，二次编解码是将这些片段再转换成 Java 对象。



对于本次实战项目，最终转换成的 Java 对象就是 MahjongProtocol 对象。

首先，我们来看一次编解码该如何编写，前面我们已经说过了，我们使用的是 长度 + 内容 的方式来进行一次编解码，在 Netty 中，对于这种方式也提供了支持，就是 LengthFieldPrepender 和 LengthFieldBasedFrameDecoder 类，前者负责编码，后者负责解码，所以，我们只需要继承这两个类就可以轻松地实现一次编解码的工作了，下面我直接给出代码：

```
public class MahjongFrameEncoder extends LengthFieldPrepender {
    public MahjongFrameEncoder() {
        // 长度字段占用两个字节
        super(2);
    }
}
public class MahjongFrameDecoder extends LengthFieldBasedFrameDecoder {
    public MahjongFrameDecoder() {
        // 2个字节表示最多可传输65535个字节的内容
        super(65535, 0, 2, 0, 2);
    }
}
```

是不是非常简单？没错，就是这么简单。

对于解码器，在经历过一次解码之后，拿到的就是一个一个完整的 MahjongProtocol 对象的字节流了，对于这个字节流，我们再编写相应的解码器将其转换成 MahjongProtocol 对象即可。

对于编码器，整个过程正好是反过来的，先通过二次编码器，将 MahjongProtocol 对象转换成字节流，再通过一次编码器将这个字节流添加上长度字段，然后再发送出去，在发送的时候，它可能会跟其它的字节流合并在一起发送出去，对方拿到这串字节流，根据一次解码器再进行解码就可以了。

那么，二次编解码该如何编写呢？

因为，我们上面定义的协议的 body 是需要先解出 header 中的 cmd，根据 cmd 找到 body 的类型，才能解出 body，所以，在编码的时候也是一样，我们先编码 header，header 中就三个字段，我们手动编码即可，对于 body，我们第一个版本简单点，使用 JSON 序列化成字节流，所以，我们的编码器看起来像下面这样：

```
public class MahjongProtocolEncoder extends MessageToMessageEncoder<MahjongProtocol> {

    @Override
    protected void encode(ChannelHandlerContext ctx, MahjongProtocol mahjongProtocol, List<Object> out) throws Exception {
        // 调用分配器分配一个ByteBuf
        ByteBuf buffer = ctx.alloc().buffer();
        // 调用的协议的编码方法
        mahjongProtocol.encode(buffer);
        // 添加到out中
        out.add(buffer);
    }
}
public final class MahjongProtocol {
    // 协议的编码方法
    public void encode(ByteBuf buffer) {
        // 调用header的编码方法
        header.encode(buffer);
        // 将body序列化成字节流写入到buffer中
        buffer.writeBytes(JSON.toJSONString(body).getBytes(StandardCharsets.UTF_8));
    }
}
public final class MahjongProtocolHeader {
    // header的编码方法
    public void encode(ByteBuf buffer) {
        // 写入版本号
        buffer.writeInt(version);
        // 写入命令字
        buffer.writeInt(cmd);
        // 写入请求ID
        buffer.writeInt(reqId);
    }
}
```

这里，你可能不禁要问：为什么 header 单独写一个编码方法，而 body 不也写一个编码方法呢？

那是因为 header 类型只有一个，而 body 的类型是有很多个的，如果在 MahjongProtocolBody 接口中定义一个 encode () 方法，那么，所有的消息类型都要实现这个方法，而它们中的内容并没有什么差别。但是，header 就不一样了，如果后面 header 增加新的内容，把编码方法放在 MahjongProtocolHeader 类中，就只需要修改这一个类就够了，如果把 header 的编码方法去掉，放到 MahjongProtocol 类中，也是可以的，只是后面增加新的字段，要同时修改两个类才可以，就不够单纯了。

二次编码器的编写相对来说比较简单一些，可以看到，这里并没有太关心 body 的类型，解码器就不一样了，请看：

```
public class MahjongProtocolDecoder extends MessageToMessageDecoder<ByteBuf> {
    @Override
    protected void decode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws Exception {
        // 创建一个协议对象
        MahjongProtocol mahjongProtocol = new MahjongProtocol();
        // 同样委托给协议自己去解码
        mahjongProtocol.decode(msg);
        // 添加到out中
        out.add(mahjongProtocol);
    }
}
public final class MahjongProtocol {
    public void decode(ByteBuf msg) {
        MahjongProtocolHeader header = new MahjongProtocolHeader();
        // 解码header
        header.decode(msg);
        this.header = header;

        // 命令字
        int cmd = header.getCmd();
        // 根据命令字获取body的真实类型
        Class<? extends MahjongProtocolBody> bodyType = getBodyTypeByCmd(cmd);
        this.body = JSON.parseObject(msg.toString(StandardCharsets.UTF_8), bodyType);
    }

    private Class<? extends MahjongProtocolBody> getBodyTypeByCmd(int cmd) {
        // todo 这里该如何写？
        return null;
    }
}
public final class MahjongProtocolHeader {
    public void decode(ByteBuf msg) {
        // 读取一个int赋值给version
        version = msg.readInt();
        // 读取一个int赋值给cmd
        cmd = msg.readInt();
        // 读取一个int赋值给请求ID
        reqId = msg.readInt();
    }
}
```

二次解码的过程相对来说要复杂一些，先解出 header，从 header 中取出 cmd，根据 cmd 找到正确的 body 类型，再使用 JSON 反序列化为 body 对象，这里的难点在于如何根据 cmd 的值找到正确的 body 类型，我提供以下几种思路：

1. 使用 Spring 容器来管理这些消息类型；
2. 使用枚举类型来管理这些消息类型；
3. 使用一个全局 Map 来管理这些消息类型；

使用 Spring 容器的话相对来说要方便一些，不过要编写自定义的 BeanPostProcessor 或者 BeanFactoryPostProcessor 来处理 cmd 和消息类型之间的映射关系，容错率也相对高一些，不过要引入 Spring，不在本课程的讨论范围之内。

使用全局 Map 的话，何时初始化这个 Map，怎么初始化这个 Map，是个头疼的问题。

使用枚举类型的话，每次添加一个新的消息，都要记得在枚举类中添加一条记录，相对来说有点麻烦，不过好在也比较简单，也不用依赖其它组件，所以，我们这里暂且使用枚举这种方式来管理这些消息类型，请看：

```
@Slf4j
@Getter
public enum MessageManager {
    HELLO_REQUEST(1, HelloRequest.class),
    ;

    private int cmd;
    private Class<? extends MahjongMessage> msgType;

    MessageManager(int cmd, Class<? extends MahjongMessage> msgType) {
        this.cmd = cmd;
        this.msgType = msgType;
    }

    public static Class<? extends MahjongMessage> getMsgTypeByCmd(int cmd) {
        for (MessageManager value : MessageManager.values()) {
            if (value.cmd == cmd) {
                return value.msgType;
            }
        }
        log.error("error cmd: {}", cmd);
        throw new RuntimeException("error cmd:" + cmd);
    }
}
```

在这里，我们定义了一个 HelloRequest 的消息，把它添加到枚举中，并给它分配一个 cmd，这样我们就可以根据 cmd 取出消息的类型了，这种方式的缺点有两个：一是新增的消息要记得在枚举中添加一次，二是 cmd 千万不能重复。

此时，我们再回过头去修改 MahjongProtocol 中的解码方法：

```
public final class MahjongProtocol {
    public void decode(ByteBuf msg) {
        MahjongProtocolHeader header = new MahjongProtocolHeader();
        // 解码header
        header.decode(msg);
        this.header = header;

        // 命令字
        int cmd = header.getCmd();
        // 根据命令字获取body的真实类型
        Class<? extends MahjongProtocolBody> bodyType = getBodyTypeByCmd(cmd);
        this.body = JSON.parseObject(msg.toString(StandardCharsets.UTF_8), bodyType);
    }

    private Class<? extends MahjongProtocolBody> getBodyTypeByCmd(int cmd) {
        // 从MessageManager中获取
        return MessageManager.getMsgTypeByCmd(cmd);
    }
}
```

这样的话，以后如果更换 cmd 与消息类型的映射方式，只需要修改 `getBodyTypeByCmd()` 这个方法就可以了。

好了，到此，协议实现和编解码实现都搞定了，下面就是把服务端和客户端实现，来检验它们的实现是否可行了。

服务端实现

服务端实现就比较简单了，前面我们已经着重分析过了，直接把代码拿过来即可。

```
public class MahjongServer {

    static final int PORT = Integer.parseInt(System.getProperty("port", "8080"));

    public static void main(String[] args) throws Exception {
        // 1. 声明线程池
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            // 2. 服务端引导器
            ServerBootstrap serverBootstrap = new ServerBootstrap();
            // 3. 设置线程池
            serverBootstrap.group(bossGroup, workerGroup)
                    // 4. 设置ServerSocketChannel的类型
                    .channel(NioServerSocketChannel.class)
                    // 5. 设置参数
                    .option(ChannelOption.SO_BACKLOG, 100)
                    // 6. 设置ServerSocketChannel对应的Handler，只能设置一个
                    .handler(new LoggingHandler(LogLevel.INFO))
                    // 7. 设置SocketChannel对应的Handler
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        public void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline p = ch.pipeline();
                            // 打印日志
                            p.addLast(new LoggingHandler(LogLevel.INFO));
                            // 一次编解码器
                            p.addLast(new MahjongFrameDecoder());
                            p.addLast(new MahjongFrameEncoder());
                            // 二次编解码器
                            p.addLast(new MahjongProtocolDecoder());
                            p.addLast(new MahjongProtocolEncoder());
                        }
                    });

            // 8. 绑定端口
            ChannelFuture f = serverBootstrap.bind(PORT).sync();
            // 9. 等待服务端监听端口关闭，这里会阻塞主线程
            f.channel().closeFuture().sync();
        } finally {
            // 10. 优雅地关闭两个线程池
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

客户端实现

客户端的实现是我们之前没有讲过的，我这里先给出代码：

```
@Slf4j
public class MahjongClient {

    static final int PORT = Integer.parseInt(System.getProperty("port", "8080"));

    public static void main(String[] args) throws Exception {
        // 工作线程池
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(workerGroup);
            bootstrap.channel(NioSocketChannel.class);
            bootstrap.handler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel ch) throws Exception {
                    ChannelPipeline pipeline = ch.pipeline();
                    // 打印日志
                    pipeline.addLast(new LoggingHandler(LogLevel.INFO));
                    // 一次编解码器
                    pipeline.addLast(new MahjongFrameDecoder());
                    pipeline.addLast(new MahjongFrameEncoder());
                    // 二次编解码器
                    pipeline.addLast(new MahjongProtocolDecoder());
                    pipeline.addLast(new MahjongProtocolEncoder());
                }
            });

            // 连接到服务端
            ChannelFuture future = bootstrap.connect(new InetSocketAddress(PORT)).sync();

            log.info("connect to server success");

            future.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
        }
    }
}
```

客户端因为不需要像服务端那样去监听网卡，所以，就不需要 ServerSocketChannel 以及 bossGroup 相关的配置了，相信有服务端编码过程的了解，对于这段代码，你一定可以驾轻就熟的，我们就不再赘述了。

双端打通

好了，服务端和客户端的代码都写好了，如何把它们连接起来呢？

其实，直接启动两者的 main () 方法，就可以成功连接起来了：

```
22:38:14 [nioEventLoopGroup-2-1] AbstractInternalLogger: [id: 0x1cafd9a5] REGISTERED
22:38:14 [nioEventLoopGroup-2-1] AbstractInternalLogger: [id: 0x1cafd9a5] CONNECT: 0.0.0.0/0.0.0.0:8080
22:38:14 [main] MahjongClient: connect to server success
22:38:14 [nioEventLoopGroup-2-1] AbstractInternalLogger: [id: 0x1cafd9a5, L:/192.168.175.1:55463 - R:0.0.0.0/0.0.0.0:8080] ACTIVE
```

只不过目前两者还没有任何消息的通信，所以，还看不到任何的效果。

因此，我们还需要定义一对消息，让它们在客户端与服务端之间传递，同时，两端还需要定义各自的 Handler 来处理这一对消息。

这一对消息我们姑且称之为 HelloRequest 和 HelloResponse，HelloRequest 在上面我们已经提起过了，这里给出它的代码：

```
@Data
public class HelloRequest implements MahjongMessage {
    private String name;
}
```

非常简单，HelloResponse 是对 HelloRequest 的回应，我们姑且给它一个 message 的字段：

```
@Data
public class HelloResponse implements MahjongMessage {
    private String message;
}
```

另外，记得在 MessageManager 中添加 HelloResponse 与 cmd 的映射关系：

```
public enum MessageManager {
    HELLO_REQUEST(1, HelloRequest.class),
    HELLO_RESPONSE(2, HelloResponse.class),
    ;
}
```

好了，两个消息我们也定义好了，下面就是定义两个 Handler 分别用来处理 HelloRequest 和 HelloResponse 了，请看服务端的 Handler：

```
@Slf4j
public class MahjongServerHandler extends SimpleChannelInboundHandler<MahjongProtocol> {

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MahjongProtocol mahjongProtocol) throws Exception {
        // 协议头
        MahjongProtocolHeader header = mahjongProtocol.getHeader();
        // 检查是不是HelloRequest的cmd
        if (header.getCmd() == MessageManager.HELLO_REQUEST.getCmd()) {
            // 强转
            HelloRequest helloRequest = (HelloRequest) mahjongProtocol.getBody();
            // 打印日志
            log.info("receive msg: {}", helloRequest);
            // 获取消息的内容
            String name = helloRequest.getName();
            // 构建响应
            HelloResponse helloResponse = new HelloResponse();
            helloResponse.setMessage("hello " + name);
            // 响应对应的协议头
            MahjongProtocolHeader outHeader = new MahjongProtocolHeader();
            outHeader.setVersion(header.getVersion());
            outHeader.setReqId(header.getReqId());
            outHeader.setCmd(MessageManager.HELLO_RESPONSE.getCmd());
            // 响应对应的协议
            MahjongProtocol out = new MahjongProtocol();
            out.setHeader(outHeader);
            out.setBody(helloResponse);
            // 写出
            ctx.writeAndFlush(out);
        }
    }
}
```

在服务端的 Handler 中，我们接收到请求之后，构造了一个响应并返回，代码看起来稍微啰嗦了一点，是因为，我们二次编解码针对的是 MahjongProtocol，所以，传递到 MahjongServerHandler 中的也是 MahjongProtocol，如果确定后续的业务逻辑处理不会再使用到 header，那么，也可以在这里统一处理 header，往后面传递的时候只传递消息，这一块我们在业务逻辑实现的小节再详细讲解。

再看客户端的 Handler：

```
@Slf4j
public class MahjongClientHandler extends SimpleChannelInboundHandler<MahjongProtocol> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MahjongProtocol mahjongProtocol) throws Exception {
        // 协议头
        MahjongProtocolHeader header = mahjongProtocol.getHeader();
        // 检查是不是HelloResponse的cmd
        if (header.getCmd() == MessageManager.HELLO_RESPONSE.getCmd()) {
            // 强转
            HelloResponse helloResponse = (HelloResponse) mahjongProtocol.getBody();
            // 打印响应
            log.info("receive response: {}", helloResponse);
        }

    }
}
```

客户端接收到消息之后，判断是不是 HelloResponse 的 cmd，然后打印出响应内容。

然后，把这两个 Handler 分别加入到服务端和客户端的 pipeline 中，分别启动服务端和客户端。

效果似乎不是我们预期的那样，仔细检查，发现，没有发送 HelloRequest 的地方呀，那么，在哪里发送 HelloRequest 呢？

无疑，是在客户端，但是，是在客户端的哪里发送比较合适呢？

有两种方式，一种是放到 MahjongClientHandler 的 channelActive () 方法中，一种是在 MahjongClient 的 main () 方法中连接建立之后发送，两种方式的代码分别如下。

放到 MahjongClientHandler 中：

```
public class MahjongClientHandler extends SimpleChannelInboundHandler<MahjongProtocol> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, MahjongProtocol mahjongProtocol) throws Exception {
        // ...省略
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        HelloRequest helloRequest = new HelloRequest();
        helloRequest.setName("tt");

        MahjongProtocolHeader header = new MahjongProtocolHeader();
        header.setVersion(1);
        header.setReqId(1);
        header.setCmd(1);

        MahjongProtocol mahjongProtocol = new MahjongProtocol();
        mahjongProtocol.setHeader(header);
        mahjongProtocol.setBody(helloRequest);

        ctx.writeAndFlush(mahjongProtocol);
    }
}
```

放在 MahjongClient 中：

```
public class MahjongClient {

    public static void main(String[] args) throws Exception {
        try{
            // ... 省略

            // 连接到服务端
            ChannelFuture future = bootstrap.connect(new InetSocketAddress(PORT)).sync();

            log.info("connect to server success");

            // 连接建立完成之后发送hello消息给服务端
            HelloRequest helloRequest = new HelloRequest();
            helloRequest.setName("tt");

            MahjongProtocolHeader header = new MahjongProtocolHeader();
            header.setVersion(1);
            header.setReqId(1);
            header.setCmd(1);

            MahjongProtocol mahjongProtocol = new MahjongProtocol();
            mahjongProtocol.setHeader(header);
            mahjongProtocol.setBody(helloRequest);

            future.channel().writeAndFlush(mahjongProtocol);

            future.channel().closeFuture().sync();
        } finally {
            workerGroup.shutdownGracefully();
        }
    }
}
```

两者二选其一即可。

此时，分别启动服务端和客户端，观察控制台日志，可以发现，很顺畅。

服务端日志：

```
//...省略
23:34:00 [nioEventLoopGroup-3-5] MahjongServerHandler: receive msg: HelloRequest(name=tt)
//...省略
```

客户端日志：

```
//...省略
23:34:00 [nioEventLoopGroup-2-1] MahjongClientHandler: receive response: HelloResponse(message=hello tt)
//...省略
```

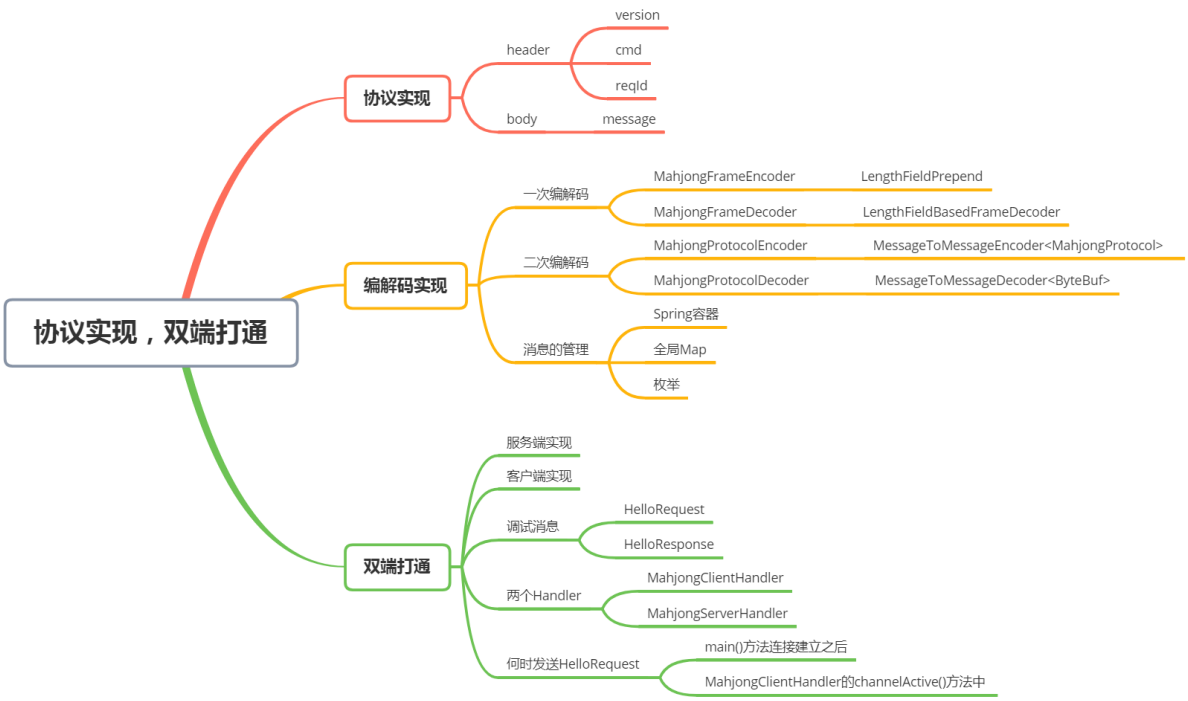
至此，双端已经完全打通。

后记

本节，我们实现了协议及其对应的编解码器，细心的同学会发现，整个过程并没有多少代码，且编解码器在客户端和服务端是通用的，也不用做什么过多的处理，这正是 Netty 的魅力之处，少量的代码就能快速打造一套强力的架构。

下一节，我们将实现系统设计一节中定义的领域模型，彼时，我们会详细讲解为什么我们选择使用贫血模型，敬请期待。

思维导图



}

29 系统设计，前期规划很重要

31 领域模型实现，我又贫血了