35 如何支持WebSocket

更新时间：2020-08-25 11:28:12

> 天才就是这样，终身努力，便成天才。——门捷列夫

前言

你好，我是彤哥。

上一节，我们从性能的角度一起学习了如何支持 Protobuf，并对整个实战项目进行了改造，工作量还是有点大的。

本节，我们将从扩展性的角度一起学习如何支持 WebSocket 协议，什么是 WebSocket 协议？为什么要使用 WebSocket 协议呢？扩展性体现在哪里呢？对于项目又该如何改造呢？

让我们带着这些问题进入今天的学习吧。

WebSocket 协议是什么？

WebSocket，是 HTML5 开始提供的一种在单个 TCP 连接上进行全双工通信的协议，它的建立连接的过程是基于 Http 协议的，需要客户端发送一个 Http 请求，并携带升级为 WebSocket 协议的头，服务端收到这个请求会尝试把这个 Http 请求升级为 WebSocket 请求，并返回给客户端类似的报文。客户端与服务端之间只有这一条请求是 Http 的，之后的通信都是基于 WebSocket 协议的了。

客户端发送的报文：

```
GET http://127.0.0.1:8080/websocket HTTP/1.1
Host: 127.0.0.1:8080
Upgrade: websocket
Connection: Upgrade
Pragma: no-cache
Cache-Control: no-cache
User-Agent: Mozilla/5.0 (Windows NT 6.3; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/78.0.3904.108 Safari/537.36
Origin: http://127.0.0.1:8080
Sec-WebSocket-Version: 13
Accept-Encoding: gzip, deflate, br
Accept-Language: zh-CN,zh;q=0.9
Sec-WebSocket-Key: VPGPDF5FOFK2wkrmTxNlgg==
Sec-WebSocket-Extensions: permessage-deflate; client_max_window_bits
```

服务端返回的报文：

```
HTTP/1.1 101 Switching Protocols
upgrade: websocket
connection: upgrade
sec-websocket-accept: Knk4EsfLZFCIttKFPqd98QXUagU=
sec-websocket-extensions: permessage-deflate
```

相比较于传统的 Http 协议，WebSocket 协议具有以下优势：

1. 是一种长连接，建立一次连接可以复用；
2. 全双工，服务端可以主动向客户端推送消息；
3. header 很小，不像 Http 请求一次需要传输大量的 header 头信息；
4. 可以支持二进制传输，不像 Http 协议是基于文本的；
5. 实时性更好，服务端可以主动推送，因此，客户端不需要轮询；

问题来了，WebSocket 虽然好，那么，我们是否一定要使用它呢？

不一定，看具体的业务场景，比如，做一个后端 RPC 框架，它完全不会跟 Web 打交道，那就可以不用考虑。如果你的应用是需要跟前端（安卓、IOS、Web、小程序）有交互的，那我建议你前期就把 WebSocket 协议考虑在内，毕竟谁也无法预测哪天是不是就要把安卓改成小程序呢，比如，同花顺（炒股软件），它的数据实时更新都是服务端主动推送给客户端的，前期它只有 APP 端，后面有了小程序端，如果前期不支持 WebSocket 协议，后期升级风险就很大。

对于，我们的麻将实战项目也是一样，谁也无法预测老板哪天是不是就要做小程序端的麻将了，所以，我们前期就支持 WebSocket 协议会比较妥当。

Netty 如何使用 WebSocket 协议？

Netty 天然就是支持 WebSocket 协议的，WebSocket 的相关处理位于 `netty-codec-http` 工程下面的 `io.netty.handler.codec.http.websocketx` 包中，那么，怎么在 Netty 中使用 WebSocket 协议呢？

Netty 都替你想好了，在 `netty-example` 工程有现成的案例，不过，那个案例还是有点小复杂，我简化了一下，以下是服务端的代码：

```
public class WebSocketServer {

    private static final String WEBSOCKET_PATH = "/websocket";
    private static final int PORT = 8080;

    public static void main(String[] args) throws InterruptedException {
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .handler(new LoggingHandler(LogLevel.INFO))
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline pipeline = ch.pipeline();
                            // 添加Http协议编解码器、处理器
                            pipeline.addLast(new HttpServerCodec());
                            pipeline.addLast(new HttpObjectAggregator(65536));
                            // 添加WebSocket处理器
                            pipeline.addLast(new WebSocketServerCompressionHandler());
                            pipeline.addLast(new WebSocketServerProtocolHandler(WEBSOCKET_PATH, null, true));
                            pipeline.addLast(new WebSocketFrameHandler());
                        }
                    });

            Channel ch = b.bind(PORT).sync().channel();
            ch.closeFuture().sync();
        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}

class WebSocketFrameHandler extends SimpleChannelInboundHandler<WebSocketFrame> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, WebSocketFrame frame) throws Exception {
        if (frame instanceof TextWebSocketFrame) {
            String request = ((TextWebSocketFrame) frame).text();
            // 回写给客户端，转换成大写
            ctx.channel().writeAndFlush(new TextWebSocketFrame(request.toUpperCase(Locale.US)));
        } else {
            String message = "unsupported frame type: " + frame.getClass().getName();
            throw new UnsupportedOperationException(message);
        }
    }
}
```

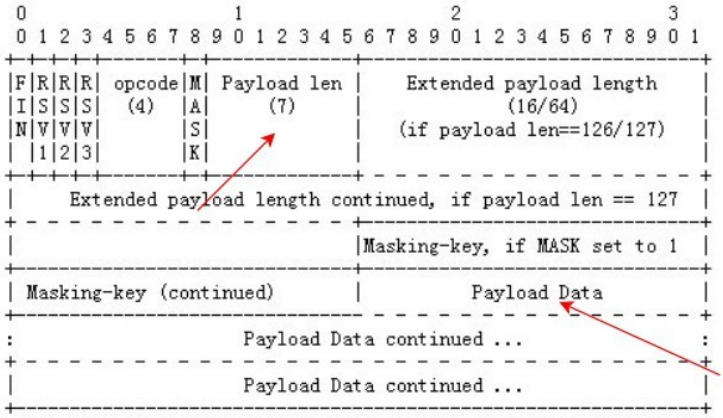
是不是超级简单，只需要添加 5 个 Handler 就可以了，让我们来测试一下。

打开百度，搜索 “websocket 在线测试”，会出来很多结果，随便打开一个，输入地址“ws://localhost:8080/websocket”，连接到服务器，在下面的框框中输入任意内容，发送后服务端都会返回一个转换成大写的响应返回，说明服务端是 OK 的。

使用就是如此简单，那么，对于我们的实战项目该如何改造呢？

项目如何改造？

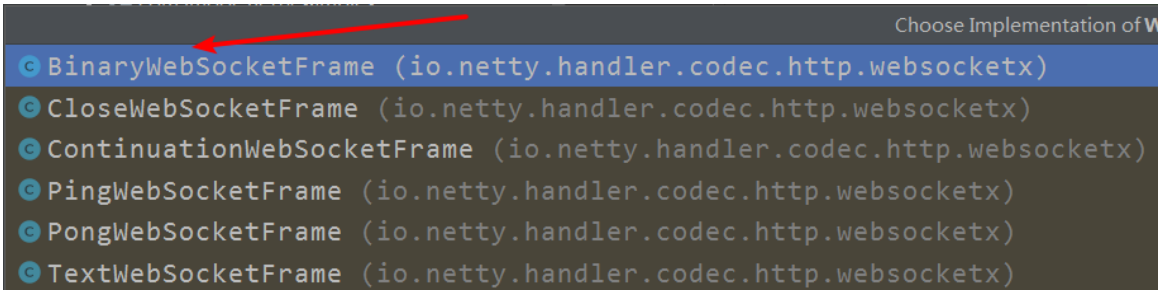
了解项目改造之前，我们有必要先了解一下 WebSocket 协议的报文结构：



可以看到，使用 WebSocket 传输的时候它本身是会指定传输内容的大小的，所以，对于一次编解码，我们直接交给 WebSocket 就可以了，通过上面的案例也可以发现，通过 WebSocket 接收到的内容最终会被转化为 TextWebSocketFrame，我们直接从 TextWebSocketFrame 里面拿解码之后的内容就可以了。

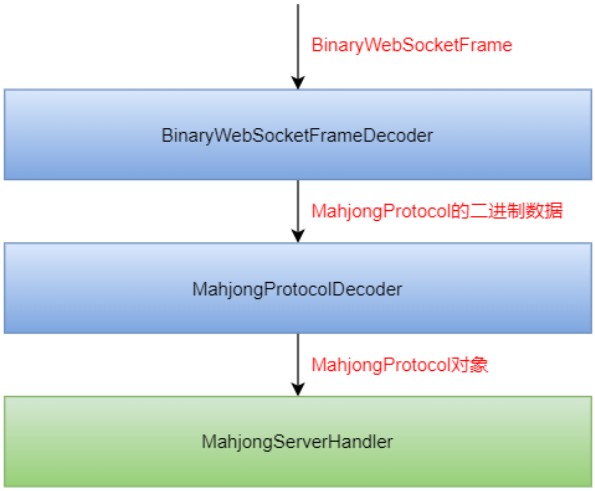
不过，我们的情况似乎不太一样，TextWebSocketFrame 只是用来传输文本，如果是用之前的 JSON 序列化的话，使用 TextWebSocketFrame 问题不大，但是，我们已经改成 Protobuf 了，它是一种二进制的序列化方式，所以，WebSocket 能不能支持二进制传输呢？

答案是肯定的，找到 TextWebSocketFrame 的父类， 在 IDEA 中按 CTRL+ALT+B 查看其所有子类：



可以看到，第一个就是二进制的帧协议，使用它就完事了。

好了，我们先从服务端开始，以服务端接收请求为例，它收到请求之后会解析出 BinaryWebSocketFrame，它里面存储的内容就是 MahjongProtocol 编码之后的二进制数据，所以，在这之后，还需要对 MahjongProtocol 的二进制解码，也就是二次解码，这个可以复用之前的 MahjongProtocolDecoder，后面的过程就都一样的。



所以，我们应该把从 BinaryWebSocketFrame 中提取出 MahjongProtocol 二进制的过程定义为一个解码器，而不是像上面简单案例中一样定义为 Handler，我们姑且叫这个解码器为 BinaryWebSocketFrameDecoder：

```
public class BinaryWebSocketFrameDecoder extends MessageToMessageDecoder<BinaryWebSocketFrame> {

    @Override
    protected void decode(ChannelHandlerContext ctx, BinaryWebSocketFrame frame, List<Object> out) throws Exception {
        out.add(frame.content());
    }
}
```

它的实现很简单，就是从 BinaryWebSocketFrame 拿出二进制数据往后传递。

如果把 MahjongProtocolDecoder 和 BinaryWebSocketFrameDecoder 合并成一个可不可以呢？

也是可以的，只是不建议，这样分层比较清晰，每个 Decoder 都有自己的职责，组合起来也比较方便，对原有代码也没有入侵。

编码的过程正好是反过来的，让我们定义一个 BinaryWebSocketFrameEncoder，它的职责是把 MahjongProtocol 的二进制数据转换成 BinaryWebSocketFrame，发送出去：

```
public class BinaryWebSocketFrameEncoder extends MessageToMessageEncoder<ByteBuf> {
    @Override
    protected void encode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws Exception {
        BinaryWebSocketFrame binaryWebSocketFrame = new BinaryWebSocketFrame(msg);
        out.add(binaryWebSocketFrame);
    }
}
```

好了， 这样服务端基本上就 OK 了，再把 WebSocket 的那几个常规处理器加入到 Pipeline 中就可以了：

```
ChannelPipeline p = ch.pipeline();
// 打印日志
p.addLast(new LoggingHandler(LogLevel.INFO));

// 添加Http协议编解码器、处理器
p.addLast(new HttpServerCodec());
p.addLast(new HttpObjectAggregator(65536));
// 添加WebSocket处理器
p.addLast(new WebSocketServerCompressionHandler());
p.addLast(new WebSocketServerProtocolHandler(WEBSOCKET_PATH, null, true));
// websocket编解码器
p.addLast(new BinaryWebSocketFrameDecoder());
p.addLast(new BinaryWebSocketFrameEncoder());

// 二次编解码器
p.addLast(new MahjongProtocolDecoder());
p.addLast(new MahjongProtocolEncoder());
// 处理器
p.addLast(new MahjongServerHandler());
```

最后，记得把原来的一次编解码器删除掉，其实，留着问题也不大，只是多此一举，BinaryWebSocketFrame 中存储的将不再是 MahjongProtocol 的二进制数据，而是经过 长度 + 内容法处理之后的二进制数据，因为 WebSocket 已经帮我们处理好了粘包半包的问题，所以，不需要再多此一举了。

服务端改造好了，我们再来看客户端该如何改造呢？

客户端在启动的时候是要发起一次额外的握手请求的，待这次握手完毕之后的处理过程，跟服务端就是完全一样的了，其实，对应的服务端也是有处理握手的过程的，查看源码会发现是在 WebSocketServerProtocolHandler 处理器处理的，但是，客户端主动发起握手并没有现成的处理器可以使用，所以，需要我们自己写一个。

参考官方的使用案例，我写了一个：

```
public class HandshakerClientHandler extends SimpleChannelInboundHandler<Object> {

    // 用于发起握手请求
    private final WebSocketClientHandshaker handshaker;
    // 监听握手请求是否完成
    private ChannelPromise handshakeFuture;

    public HandshakerClientHandler(WebSocketClientHandshaker handshaker) {
        this.handshaker = handshaker;
    }

    public ChannelFuture handshakeFuture() {
        return handshakeFuture;
    }

    @Override
    public void handlerAdded(ChannelHandlerContext ctx) {
        handshakeFuture = ctx.newPromise();
    }

    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        // 连接完成后立马发起握手请求
        handshaker.handshake(ctx.channel());
    }

    @Override
    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws Exception {
        Channel ch = ctx.channel();
        // 第一个请求必然是握手请求
        if (!handshaker.isHandshakeComplete()) {
            try {
                handshaker.finishHandshake(ch, (FullHttpResponse) msg);
                System.out.println("WebSocket Client connected!");
                handshakeFuture.setSuccess();
            } catch (WebSocketHandshakeException e) {
                System.out.println("WebSocket Client failed to connect");
                handshakeFuture.setFailure(e);
            }
            return;
        } else {
            // 如果已经握手完成了，交给后面的处理器进行处理
            ctx.fireChannelRead(msg);
        }
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        cause.printStackTrace();
        if (!handshakeFuture.isDone()) {
            handshakeFuture.setFailure(cause);
        }
        ctx.close();
    }
}
```

比一般的 Handler 稍微复杂一点，首先在连接创建完成之后立马发起握手请求，对于接收到的数据，先判断是不是握手的响应，只有响应成功了，才能视为 WebSocket 连接建立成功了，之后，就跟 Http 没啥关系了。

除了多这么一步操作，其它的跟服务端一模一样，所以，一样是修改客户端的 Pipeline：

```
public class MahjongClient {

    static final String URL = System.getProperty("url", "ws://127.0.0.1:8080/websocket");

    public static void main(String[] args) throws Exception {
        // 工作线程池
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            URI uri = new URI(URL);
            final HandshakerClientHandler handler =
                    new HandshakerClientHandler(
                            WebSocketClientHandshakerFactory.newHandshaker(
                                    uri, WebSocketVersion.V13, null, true, new DefaultHttpHeaders()));

            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 p = ch.pipeline();

                    // 打印日志
                    p.addLast(new LoggingHandler(LogLevel.INFO));

                    p.addLast(new HttpClientCodec());
                    p.addLast(new HttpObjectAggregator(8192));
                    p.addLast(WebSocketClientCompressionHandler.INSTANCE);
                    p.addLast(handler);

                    // websocket编解码器
                    p.addLast(new BinaryWebSocketFrameDecoder());
                    p.addLast(new BinaryWebSocketFrameEncoder());

                    // 二次编解码器
                    p.addLast(new MahjongProtocolDecoder());
                    p.addLast(new MahjongProtocolEncoder());
                    // 处理器
                    p.addLast(new MahjongClientHandler());
                }
            });

            // 连接到服务端
            ChannelFuture future = bootstrap.connect(new InetSocketAddress(uri.getPort())).sync();
            // 等待WebSocket握手完成
            handler.handshakeFuture().sync();

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

            MockClient.start(future.channel());

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

这里要注意的是，需要等待握手完成之后，才能进行后续的操作，也就是 Mock 客户端开始工作。

好了，到这里，基本客户端和服务端都改造完毕了，让我们调试起来看看效果。

先启动服务端，一切正常，再启动客户端，你会发现客户端请求发送失败，而且没有任何报错信息，经过断点跟踪，会发现抛出了一个 IllegalReferenceCountException 的异常，这个异常被设置到了 Promise 中，这个 Promise 又是什么呢？它其实是写数据时返回的一个对象，名叫 ChannelFuture，所以，我们只需要给它添加一个监听器，发送数据失败的时候打印下异常就可以了：

```
private static void send(Channel channel, MahjongProtocol mahjongProtocol) {
    if (channel != null && channel.isActive() && channel.isWritable()) {
        ChannelFuture channelFuture = channel.writeAndFlush(mahjongProtocol);
        channelFuture.addListener(future -> {
            if (!future.isSuccess()) {
                log.error("send message error", future.cause());
            }
        });
    } else {
        log.error("channel unavailable, channelId={}, msgType={}", channel.id(), ((MessageLite) mahjongProtocol.getBody()).getClass());
    }
}
```

这个确实有点小复杂，没有前面的源码剖析的基础，这一块也很难定位并完善。

抛出这个异常的原因是 BinaryWebSocketFrame 直接使用了 MahjongProtocol 对应的 ByteBuf，而每一次编解码或者 Handler 处理之后，msg 的引用计数都会减一，导致 ByteBuf 的引用计数减完了，以 BinaryWebSocketFrameEncoder 为例：

```
public class BinaryWebSocketFrameEncoder extends MessageToMessageEncoder<ByteBuf> {
    @Override
    protected void encode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws Exception {
        // 直接使用的msg，这个msg就是MahjongProtocol转换后的二进制数据
        BinaryWebSocketFrame binaryWebSocketFrame = new BinaryWebSocketFrame(msg);
        out.add(binaryWebSocketFrame);
    }
}
public abstract class MessageToMessageEncoder<I> extends ChannelOutboundHandlerAdapter {
    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
        //...省略其他代码
        try {
            // 调用上面的encode()
            encode(ctx, cast, out);
        } finally {
            // 调用完减了一次引用计数
            ReferenceCountUtil.release(cast);
        }
    }
```

默认地，一个 ByteBuf 创建完成之后它只能被使用一次，如果需要重复使用这个 ByteBuf，需要调用 `ReferenceCountUtil.retain(msg);` 方法给它加次数，所以，要解决这个问题也很简单，在 BinaryWebSocketFrameEncoder 中显式地调用一下这个方法：

```
public class BinaryWebSocketFrameEncoder extends MessageToMessageEncoder<ByteBuf> {
    @Override
    protected void encode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws Exception {
        // 显式地增加引用计数
        ReferenceCountUtil.retain(msg);
        // 直接使用的msg，这个msg就是MahjongProtocol转换后的二进制数据
        BinaryWebSocketFrame binaryWebSocketFrame = new BinaryWebSocketFrame(msg);
        out.add(binaryWebSocketFrame);
    }
}
```

同样地，BinaryWebSocketFrameDecoder 以及 HandshakerClientHandler 也有同样的问题，分别调用一下这个方法即可：

```
public class BinaryWebSocketFrameDecoder extends MessageToMessageDecoder<BinaryWebSocketFrame> {

    @Override
    protected void decode(ChannelHandlerContext ctx, BinaryWebSocketFrame frame, List<Object> out) throws Exception {
        ReferenceCountUtil.retain(frame.content());
        out.add(frame.content());
    }
}
public class HandshakerClientHandler extends SimpleChannelInboundHandler<Object> {
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws Exception {
        Channel ch = ctx.channel();
        // 第一个请求必然是握手请求
        if (!handshaker.isHandshakeComplete()) {
            //...省略其他代码
        } else {
            // 如果已经握手完成了，交给后面的处理器进行处理
            ReferenceCountUtil.retain(msg);
            ctx.fireChannelRead(msg);
        }
    }
}
```

修改完毕之后，重启服务端、客户端，发现一切都正常了，我这里就不演示了。

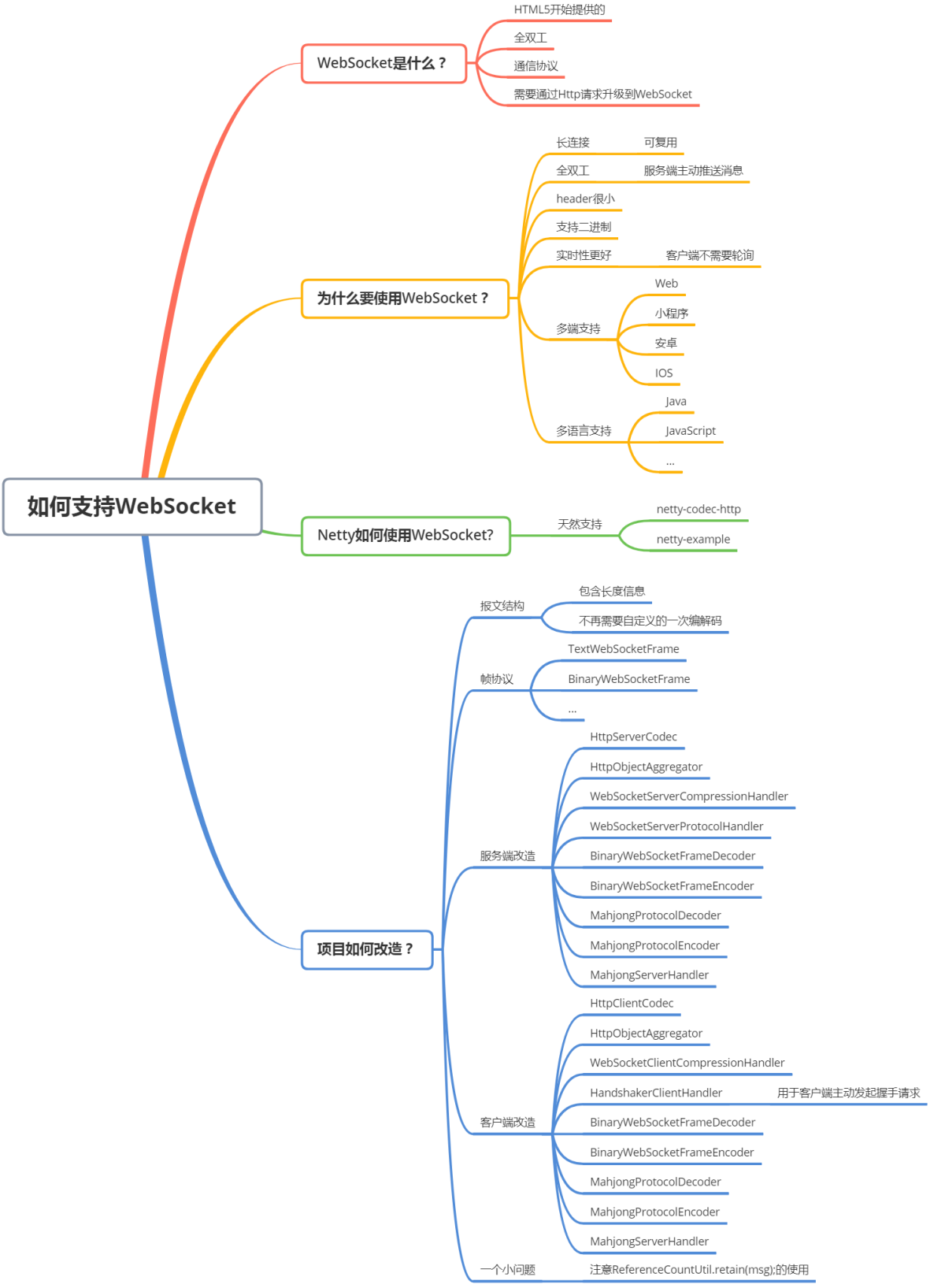
后记

本节，我们一起学习了如何在 Netty 中使用 WebSocket 协议，并将我们的项目改造成了支持 WebSocket，期间还发现了一个小问题。

细心的同学可能会想，这样显式地调用 `ReferenceCountUtil.retain(msg);` 真的好么？会不会有内存泄漏的风险？

这个问题我们后面再详细讨论，下一节，我将从安全性的角度出发，给实战项目添加上安全的外衣，减少被攻击被窃取数据的风险，敬请期待。

思维导图



}

34 如何支持Protobuf

36 如何增加安全性