自定义协议

消息结构设计

1. **魔数**。作用是安全校验，防止非服务器发来的消息。1字节
2. **版本号**。保证请求和响应版本一致。1字节
3. **序列化方式**。告诉服务端和消费端怎么解析数据。1字节
4. **类型**。标识是请求还是响应。1字节
5. **状态**。如果类型是响应，响应状态是什么。1字节

6. **请求id**。tcp请求是双向的，需要有唯一id追踪每个请求。8字节
7. **请求体长度**。4字节
8. **请求体**

开发实现

1. 消息结构

Java

```java
package com.yupi.yurpc.protocol;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * 协议消息结构
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @from <a href="https://yupi.icu">编程导航学习圈</a>
 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ProtocolMessage<T> {

    /**
     * 消息头
     */
    private Header header;

    /**
     * 消息体（请求或响应对象）
     */
    private T body;

    /**
     * 协议消息头
     */
    @Data
    public static class Header {

        /**
         * 魔数，保证安全性
         */
        private byte magic;

        /**
         * 版本号
         */
        private byte version;

```

```
45          /**
46           * 序列化器
47           */
48          private byte serializer;
49
50          /**
51           * 消息类型（请求 / 响应）
52           */
53          private byte type;
54
55          /**
56           * 状态
57           */
58          private byte status;
59
60          /**
61           * 请求 id
62           */
63          private long requestId;
64
65          /**
66           * 消息体长度
67           */
68          private int bodyLength;
69      }
70
71  }
72
```

2. 消息协议常量

```java
package com.yupi.yurpc.protocol;

/**
 * 协议常量
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @from <a href="https://yupi.icu">编程导航学习圈</a>
 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
 */
public interface ProtocolConstant {

    /**
     * 消息头长度
     */
    int MESSAGE_HEADER_LENGTH = 17;

    /**
     * 协议魔数
     */
    byte PROTOCOL_MAGIC = 0x1;

    /**
     * 协议版本号
     */
    byte PROTOCOL_VERSION = 0x1;
}

```

3. 消息协议字段枚举

3.1. 序列化方式枚举

Java

```java
package com.yupi.yurpc.protocol;

import cn.hutool.core.util.ObjectUtil;
import lombok.Getter;

import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

/**
 * 协议消息的序列化器枚举
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @from <a href="https://yupi.icu">编程导航学习圈</a>
 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
 */
@Getter
public enum ProtocolMessageSerializerEnum {

    JDK(0, "jdk"),
    JSON(1, "json"),
    KRYO(2, "kryo"),
    HESSIAN(3, "hessian");

    private final int key;

    private final String value;

    ProtocolMessageSerializerEnum(int key, String value) {
        this.key = key;
        this.value = value;
    }

    /**
     * 获取值列表
     *
     * @return
     */
    public static List<String> getValues() {
        return Arrays.stream(values()).map(item -> item.value).collect(Collectors.toList());
    }

    /**
```

```
44          * 根据 key 获取枚举
45          *
46          * @param key
47          * @return
48          */
49         public static ProtocolMessageSerializerEnum getEnumByKey(int key) {
50             for (ProtocolMessageSerializerEnum anEnum : ProtocolMessageSeriali
    zerEnum.values()) {
51                 if (anEnum.key == key) {
52                     return anEnum;
53                 }
54             }
55             return null;
56         }
57
58
59         /**
60          * 根据 value 获取枚举
61          *
62          * @param value
63          * @return
64          */
65         public static ProtocolMessageSerializerEnum getEnumByValue(String valu
    e) {
66             if (ObjectUtil.isEmpty(value)) {
67                 return null;
68             }
69             for (ProtocolMessageSerializerEnum anEnum : ProtocolMessageSeriali
    zerEnum.values()) {
70                 if (anEnum.value.equals(value)) {
71                     return anEnum;
72                 }
73             }
74             return null;
75         }
76     }
77
```

3.2. 消息类型枚举

Java

```java
package com.yupi.yurpc.protocol;

import lombok.Getter;

/**
 * 协议消息的类型枚举
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @from <a href="https://yupi.icu">编程导航学习圈</a>
 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
 */
@Getter
public enum ProtocolMessageTypeEnum {

    REQUEST(0),
    RESPONSE(1),
    HEART_BEAT(2),
    OTHERS(3);

    private final int key;

    ProtocolMessageTypeEnum(int key) {
        this.key = key;
    }

    /**
     * 根据 key 获取枚举
     *
     * @param key
     * @return
     */
    public static ProtocolMessageTypeEnum getEnumByKey(int key) {
        for (ProtocolMessageTypeEnum anEnum : ProtocolMessageTypeEnum.values()) {
            if (anEnum.key == key) {
                return anEnum;
            }
        }
        return null;
    }
}

```

3.3. 响应状态

Java

```java
package com.yupi.yurpc.protocol;

import lombok.Getter;

/**
 * 协议消息的状态枚举
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @from <a href="https://yupi.icu">编程导航学习圈</a>
 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
 */
@Getter
public enum ProtocolMessageStatusEnum {

    OK("ok", 20),
    BAD_REQUEST("badRequest", 40),
    BAD_RESPONSE("badResponse", 50);

    private final String text;

    private final int value;

    ProtocolMessageStatusEnum(String text, int value) {
        this.text = text;
        this.value = value;
    }

    /**
     * 根据 value 获取枚举
     *
     * @param value
     * @return
     */
    public static ProtocolMessageStatusEnum getEnumByValue(int value) {
        for (ProtocolMessageStatusEnum anEnum : ProtocolMessageStatusEnum.values()) {
            if (anEnum.value == value) {
                return anEnum;
            }
        }
        return null;
    }
}

```

4. 网络传输

使用Vertx的TCP传输

Java

```java
package com.yupi.yurpc.server.tcp;

import com.yupi.yurpc.server.HttpServer;
import io.vertx.core.Vertx;
import io.vertx.core.buffer.Buffer;
import io.vertx.core.net.NetServer;

public class VertxTcpServer implements HttpServer {

    private byte[] handleRequest(byte[] requestData) {
        // 在这里编写处理请求的逻辑，根据 requestData 构造响应数据并返回
        // 这里只是一个示例，实际逻辑需要根据具体的业务需求来实现
        return "Hello, client!".getBytes();
    }

    @Override
    public void doStart(int port) {
        // 创建 Vert.x 实例
        Vertx vertx = Vertx.vertx();

        // 创建 TCP 服务器
        NetServer server = vertx.createNetServer();

        // 处理请求
        server.connectHandler(socket -> {
            // 处理连接
            socket.handler(buffer -> {
                // 处理接收到的字节数组
                byte[] requestData = buffer.getBytes();
                // 在这里进行自定义的字节数组处理逻辑，比如解析请求、调用服务、构造响应等
                byte[] responseData = handleRequest(requestData);
                // 发送响应
                socket.write(Buffer.buffer(responseData));
            });
        });

        // 启动 TCP 服务器并监听指定端口
        server.listen(port, result -> {
            if (result.succeeded()) {
                System.out.println("TCP server started on port " + port);
            } else {
                System.err.println("Failed to start TCP server: " + result.cause());
```

```
43                 }
44             });
45         }
46 
47         public static void main(String[] args) {
48             new VertxTcpServer().doStart(8888);
49         }
50     }
51 
```

5. 编码/解码 器

5.1. 使用流程

1. 客户端组装消息协议类，设置各个属性，例如消息头，消息体
2. 然后在编码器里将各个属性设置到Buffer的固定位置，例如魔数是第1个，版本号第2个.等等。。。。
3. 发送请求到服务端，服务端接收到请求
4. 在解码器里按固定格式解析buffer
5. 解析后的报文放到请求处理器中处理
6. 服务端使用编码器对响应进行处理
7. 客户端接收到响应，在解码器里解析响应
8. 解析后的响应使用响应处理器处理

5.2. 编码器

Java

```java
package com.yupi.yurpc.protocol;

import com.yupi.yurpc.serializer.Serializer;
import com.yupi.yurpc.serializer.SerializerFactory;
import io.vertx.core.buffer.Buffer;

import java.io.IOException;

public class ProtocolMessageEncoder {

    /**
     * 编码
     *
     * @param protocolMessage
     * @return
     * @throws IOException
     */
    public static Buffer encode(ProtocolMessage<?> protocolMessage) throws IOException {
        if (protocolMessage == null || protocolMessage.getHeader() == null) {
            return Buffer.buffer();
        }
        ProtocolMessage.Header header = protocolMessage.getHeader();
        // 依次向缓冲区写入字节
        Buffer buffer = Buffer.buffer();
        buffer.appendByte(header.getMagic());
        buffer.appendByte(header.getVersion());
        buffer.appendByte(header.getSerializer());
        buffer.appendByte(header.getType());
        buffer.appendByte(header.getStatus());
        buffer.appendLong(header.getRequestId());
        // 获取序列化器
        ProtocolMessageSerializerEnum serializerEnum = ProtocolMessageSerializerEnum.getEnumByKey(header.getSerializer());
        if (serializerEnum == null) {
            throw new RuntimeException("序列化协议不存在");
        }
        Serializer serializer = SerializerFactory.getInstance(serializerEnum.getValue());
        byte[] bodyBytes = serializer.serialize(protocolMessage.getBody());
        // 写入 body 长度和数据
        buffer.appendInt(bodyBytes.length);
```

```
40            buffer.appendBytes(bodyBytes);
41            return buffer;
42        }
43    }
44
```

5.3. 解码器

Java

```java
package com.yupi.yurpc.protocol;

import com.yupi.yurpc.serializer.Serializer;
import com.yupi.yurpc.serializer.SerializerFactory;
import io.vertx.core.buffer.Buffer;

import java.io.IOException;

public class ProtocolMessageEncoder {

    /**
     * 编码
     *
     * @param protocolMessage
     * @return
     * @throws IOException
     */
    public static Buffer encode(ProtocolMessage<?> protocolMessage) throws IOException {
        if (protocolMessage == null || protocolMessage.getHeader() == null) {
            return Buffer.buffer();
        }
        ProtocolMessage.Header header = protocolMessage.getHeader();
        // 依次向缓冲区写入字节
        Buffer buffer = Buffer.buffer();
        buffer.appendByte(header.getMagic());
        buffer.appendByte(header.getVersion());
        buffer.appendByte(header.getSerializer());
        buffer.appendByte(header.getType());
        buffer.appendByte(header.getStatus());
        buffer.appendLong(header.getRequestId());
        // 获取序列化器
        ProtocolMessageSerializerEnum serializerEnum = ProtocolMessageSerializerEnum.getEnumByKey(header.getSerializer());
        if (serializerEnum == null) {
            throw new RuntimeException("序列化协议不存在");
        }
        Serializer serializer = SerializerFactory.getInstance(serializerEnum.getValue());
        byte[] bodyBytes = serializer.serialize(protocolMessage.getBody());
        // 写入 body 长度和数据
        buffer.appendInt(bodyBytes.length);
```

```
40            buffer.appendBytes(bodyBytes);
41            return buffer;
42        }
43    }
44
```

5.4. 请求处理器

5.4.1. 使用流程

1. 将接收的请求解码后
2. 获取rpcrequest
3. 从rpcequest中获取需要的方法，使用反射调用方法，获取结果
4. 将结果封装
5. 进行编码并发送

Java

```java
package com.yupi.yurpc.server.tcp;

import com.yupi.yurpc.model.RpcRequest;
import com.yupi.yurpc.model.RpcResponse;
import com.yupi.yurpc.protocol.ProtocolMessage;
import com.yupi.yurpc.protocol.ProtocolMessageDecoder;
import com.yupi.yurpc.protocol.ProtocolMessageEncoder;
import com.yupi.yurpc.protocol.ProtocolMessageTypeEnum;
import com.yupi.yurpc.registry.LocalRegistry;
import io.vertx.core.Handler;
import io.vertx.core.buffer.Buffer;
import io.vertx.core.net.NetSocket;

import java.io.IOException;
import java.lang.reflect.Method;

public class TcpServerHandler implements Handler<NetSocket> {

    @Override
    public void handle(NetSocket netSocket) {
        // 处理连接
        netSocket.handler(buffer -> {
            // 接受请求，解码
            ProtocolMessage<RpcRequest> protocolMessage;
            try {
                protocolMessage = (ProtocolMessage<RpcRequest>) ProtocolMessageDecoder.decode(buffer);
            } catch (IOException e) {
                throw new RuntimeException("协议消息解码错误");
            }
            RpcRequest rpcRequest = protocolMessage.getBody();

            // 处理请求
            // 构造响应结果对象
            RpcResponse rpcResponse = new RpcResponse();
            try {
                // 获取要调用的服务实现类，通过反射调用
                Class<?> implClass = LocalRegistry.get(rpcRequest.getServiceName());
                Method method = implClass.getMethod(rpcRequest.getMethodName(), rpcRequest.getParameterTypes());
                Object result = method.invoke(implClass.newInstance(), rpcRequest.getArgs());
                // 封装返回结果
```

```
                rpcResponse.setData(result);
                rpcResponse.setDataType(method.getReturnType());
                rpcResponse.setMessage("ok");
            } catch (Exception e) {
                e.printStackTrace();
                rpcResponse.setMessage(e.getMessage());
                rpcResponse.setException(e);
            }

            // 发送响应，编码
            ProtocolMessage.Header header = protocolMessage.getHeader();
            header.setType((byte) ProtocolMessageTypeEnum.RESPONSE.getKey
());
            ProtocolMessage<RpcResponse> responseProtocolMessage = new Pro
tocolMessage<>(header, rpcResponse);
            try {
                Buffer encode = ProtocolMessageEncoder.encode(responseProt
ocolMessage);
                netSocket.write(encode);
            } catch (IOException e) {
                throw new RuntimeException("协议消息编码错误");
            }
        });
    }
}

```

5.5. 请求发送

Java

```java
package com.yupi.yurpc.proxy;

import cn.hutool.core.collection.CollUtil;
import cn.hutool.core.util.IdUtil;
import cn.hutool.http.HttpRequest;
import cn.hutool.http.HttpResponse;
import com.yupi.yurpc.RpcApplication;
import com.yupi.yurpc.config.RpcConfig;
import com.yupi.yurpc.constant.RpcConstant;
import com.yupi.yurpc.model.RpcRequest;
import com.yupi.yurpc.model.RpcResponse;
import com.yupi.yurpc.model.ServiceMetaInfo;
import com.yupi.yurpc.protocol.*;
import com.yupi.yurpc.registry.Registry;
import com.yupi.yurpc.registry.RegistryFactory;
import com.yupi.yurpc.serializer.Serializer;
import com.yupi.yurpc.serializer.SerializerFactory;
import io.vertx.core.Future;
import io.vertx.core.Vertx;
import io.vertx.core.buffer.Buffer;
import io.vertx.core.net.NetClient;
import io.vertx.core.net.NetSocket;
import io.vertx.core.net.SocketAddress;

import java.io.IOException;
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.List;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.CountDownLatch;

/**
 * 服务代理（JDK 动态代理）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class ServiceProxy implements InvocationHandler {

    /**
     * 调用代理
     *
     * @return
```

```
45          * @throws Throwable
46          */
47         @Override
48         public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
49             // 指定序列化器
50             final Serializer serializer = SerializerFactory.getInstance(RpcApplication.getRpcConfig().getSerializer());
51 
52             // 构造请求
53             String serviceName = method.getDeclaringClass().getName();
54             RpcRequest rpcRequest = RpcRequest.builder()
55                     .serviceName(serviceName)
56                     .methodName(method.getName())
57                     .parameterTypes(method.getParameterTypes())
58                     .args(args)
59                     .build();
60             try {
61                 // 序列化
62                 byte[] bodyBytes = serializer.serialize(rpcRequest);
63                 // 从注册中心获取服务提供者请求地址
64                 RpcConfig rpcConfig = RpcApplication.getRpcConfig();
65                 Registry registry = RegistryFactory.getInstance(rpcConfig.getRegistryConfig().getRegistry());
66                 ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
67                 serviceMetaInfo.setServiceName(serviceName);
68                 serviceMetaInfo.setServiceVersion(RpcConstant.DEFAULT_SERVICE_VERSION);
69                 List<ServiceMetaInfo> serviceMetaInfoList = registry.serviceDiscovery(serviceMetaInfo.getServiceKey());
70                 if (CollUtil.isEmpty(serviceMetaInfoList)) {
71                     throw new RuntimeException("暂无服务地址");
72                 }
73                 ServiceMetaInfo selectedServiceMetaInfo = serviceMetaInfoList.get(0);
74                 // 发送 TCP 请求
75                 Vertx vertx = Vertx.vertx();
76                 NetClient netClient = vertx.createNetClient();
77                 CompletableFuture<RpcResponse> responseFuture = new CompletableFuture<>();
78                 netClient.connect(selectedServiceMetaInfo.getServicePort(), selectedServiceMetaInfo.getServiceHost(),
79                         result -> {
80                             if (result.succeeded()) {
81                                 System.out.println("Connected to TCP server");
82 
```

```
                            io.vertx.core.net.NetSocket socket = result.r
83  esult();
84                          // 发送数据
85                          // 构造消息
                            ProtocolMessage<RpcRequest> protocolMessage
86  = new ProtocolMessage<>();
                            ProtocolMessage.Header header = new ProtocolM
87  essage.Header();
                            header.setMagic(ProtocolConstant.PROTOCOL_MAG
88  IC);
                            header.setVersion(ProtocolConstant.PROTOCOL_V
89  ERSION);
                            header.setSerializer((byte) ProtocolMessageSe
    rializerEnum.getEnumByValue(RpcApplication.getRpcConfig().getSerializer
90  ()).getKey());
                            header.setType((byte) ProtocolMessageTypeEnu
91  m.REQUEST.getKey());
                            header.setRequestId(IdUtil.getSnowflakeNextId
92  ());
93                          protocolMessage.setHeader(header);
94                          protocolMessage.setBody(rpcRequest);
95                          // 编码请求
96                          try {
                                Buffer encodeBuffer = ProtocolMessageEnco
97  der.encode(protocolMessage);
98                              socket.write(encodeBuffer);
99                          } catch (IOException e) {
                                throw new RuntimeException("协议消息编码错
100 误");
101                         }
102
103                         // 接收响应
104                         socket.handler(buffer -> {
105                             try {
                                    ProtocolMessage<RpcResponse> rpcRespo
    nseProtocolMessage = (ProtocolMessage<RpcResponse>) ProtocolMessageDecode
106 r.decode(buffer);
                                    responseFuture.complete(rpcResponsePr
107 otocolMessage.getBody());
108                             } catch (IOException e) {
                                    throw new RuntimeException("协议消息解
109 码错误");
110                             }
111                         });
112                     } else {
                            System.err.println("Failed to connect to TCP
113 server");
```

```
                        }
                    });

            RpcResponse rpcResponse = responseFuture.get();
            // 记得关闭连接
            netClient.close();
            return rpcResponse.getData();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;
    }
```

6. 半包/粘包问题

6.1.1. 半包

半包就是接收的数据少了

解决半包办法：在消息头中设置消息体长度，每次获取消息体时，判断长度是否一致，不一致就留着下次读

6.1.2. 粘包

粘包就是接收的数据多了

解决办法：每次获取消息体时，判断长度，如果长了，就按照消息头中的长度读取，剩下的下次读。

6.1.3. 使用vertx解决半包粘包

Java

```java
package com.yupi.yurpc.server.tcp;

import com.yupi.yurpc.protocol.ProtocolConstant;
import com.yupi.yurpc.server.HttpServer;
import io.vertx.core.Handler;
import io.vertx.core.Vertx;
import io.vertx.core.buffer.Buffer;
import io.vertx.core.net.NetServer;
import io.vertx.core.parsetools.RecordParser;
import lombok.extern.slf4j.Slf4j;

@Slf4j
public class VertxTcpServer implements HttpServer {

    @Override
    public void doStart(int port) {
        // 创建 Vert.x 实例
        Vertx vertx = Vertx.vertx();

        // 创建 TCP 服务器
        NetServer server = vertx.createNetServer();

        // 处理请求
        server.connectHandler(socket -> {
            // 构造 parser
            RecordParser parser = RecordParser.newFixed(8);
            parser.setOutput(new Handler<Buffer>() {
                // 初始化
                int size = -1;
                // 一次完整的读取（头 + 体）
                Buffer resultBuffer = Buffer.buffer();

                @Override
                public void handle(Buffer buffer) {
                    if (-1 == size) {
                        // 读取消息体长度
                        size = buffer.getInt(4);
                        parser.fixedSizeMode(size);
                        // 写入头信息到结果
                        resultBuffer.appendBuffer(buffer);
                    } else {
                        // 写入体信息到结果
                        resultBuffer.appendBuffer(buffer);
                        System.out.println(resultBuffer.toString());
```

```
                            // 重置一轮
                            parser.fixedSizeMode(8);
                            size = -1;
                            resultBuffer = Buffer.buffer();
                        }
                    }
                });

                socket.handler(parser);
            });

            // 启动 TCP 服务器并监听指定端口
            server.listen(port, result -> {
                if (result.succeeded()) {
                    log.info("TCP server started on port " + port);
                } else {
                    log.info("Failed to start TCP server: " + result.cause());
                }
            });
        }

        public static void main(String[] args) {
            new VertxTcpServer().doStart(8888);
        }
    }

```

Java

```java
package com.yupi.yurpc.server.tcp;

import io.vertx.core.Vertx;
import io.vertx.core.buffer.Buffer;

public class VertxTcpClient {

    public void start() {
        // 创建 Vert.x 实例
        Vertx vertx = Vertx.vertx();

        vertx.createNetClient().connect(8888, "localhost", result -> {
            if (result.succeeded()) {
                System.out.println("Connected to TCP server");
                io.vertx.core.net.NetSocket socket = result.result();
                for (int i = 0; i < 1000; i++) {
                    // 发送数据
                    Buffer buffer = Buffer.buffer();
                    String str = "Hello, server!Hello, server!Hello, server!Hello, server!";
                    buffer.appendInt(0);
                    buffer.appendInt(str.getBytes().length);
                    buffer.appendBytes(str.getBytes());
                    socket.write(buffer);
                }
                // 接收响应
                socket.handler(buffer -> {
                    System.out.println("Received response from server: " + buffer.toString());
                });
            } else {
                System.err.println("Failed to connect to TCP server");
            }
        });
    }

    public static void main(String[] args) {
        new VertxTcpClient().start();
    }
}

```

Java

```java
package com.yupi.yurpc.server.tcp;

import com.yupi.yurpc.protocol.ProtocolConstant;
import io.vertx.core.Handler;
import io.vertx.core.buffer.Buffer;
import io.vertx.core.parsetools.RecordParser;

/**
 * 装饰者模式（使用 recordParser 对原有的 buffer 处理能力进行增强）
 */
public class TcpBufferHandlerWrapper implements Handler<Buffer> {

    private final RecordParser recordParser;

    public TcpBufferHandlerWrapper(Handler<Buffer> bufferHandler) {
        recordParser = initRecordParser(bufferHandler);
    }

    @Override
    public void handle(Buffer buffer) {
        recordParser.handle(buffer);
    }

    private RecordParser initRecordParser(Handler<Buffer> bufferHandler) {
        // 构造 parser
        RecordParser parser = RecordParser.newFixed(ProtocolConstant.MESSAGE_HEADER_LENGTH);

        parser.setOutput(new Handler<Buffer>() {
            // 初始化
            int size = -1;
            // 一次完整的读取（头 + 体）
            Buffer resultBuffer = Buffer.buffer();

            @Override
            public void handle(Buffer buffer) {
                if (-1 == size) {
                    // 读取消息体长度
                    size = buffer.getInt(13);
                    parser.fixedSizeMode(size);
                    // 写入头信息到结果
                    resultBuffer.appendBuffer(buffer);
                } else {
                    // 写入体信息到结果
```

```
                    resultBuffer.appendBuffer(buffer);
                    // 已拼接为完整 Buffer，执行处理
                    bufferHandler.handle(resultBuffer);
                    // 重置一轮
                    parser.fixedSizeMode(ProtocolConstant.MESSAGE_HEADER_LENGTH);
                    size = -1;
                    resultBuffer = Buffer.buffer();
                }
            }
        });

        return parser;
    }
}
```