

自定义协议

消息结构设计

开发实现

1. 消息结构
2. 消息协议常量
3. 消息协议字段枚举
 - 3.1. 序列化方式枚举
 - 3.2. 消息类型枚举
 - 3.3. 响应状态
4. 网络传输
5. 编码/解码器
 - 5.1. 使用流程
 - 5.2. 编码器
 - 5.3. 解码器
 - 5.4. 请求处理器
 - 5.4.1. 使用流程
 - 5.5. 请求发送
6. 半包/粘包问题
 - 6.1.1. 半包
 - 6.1.2. 粘包
 - 6.1.3. 使用vertx解决半包粘包

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

```
1 package com.yupi.yurpc.protocol;
2
3 import lombok.AllArgsConstructor;
4 import lombok.Data;
5 import lombok.NoArgsConstructor;
6
7 /**
8  * 协议消息结构
9  *
10  * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
11  * @from <a href="https://yupi.icu">编程导航学习圈</a>
12  * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
13  */
14 @Data
15 @AllArgsConstructor
16 @NoArgsConstructor
17 public class ProtocolMessage<T> {
18
19     /**
20      * 消息头
21      */
22     private Header header;
23
24     /**
25      * 消息体（请求或响应对象）
26      */
27     private T body;
28
29     /**
30      * 协议消息头
31      */
32     @Data
33     public static class Header {
34
35         /**
36          * 魔数，保证安全性
37          */
38         private byte magic;
39
40         /**
41          * 版本号
42          */
43         private byte version;
44     }
45 }
```

```
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

```
1 package com.yupi.yurpc.protocol;
2
3 /**
4  * 协议常量
5  *
6  * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
7  * @from <a href="https://yupi.icu">编程导航学习圈</a>
8  * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
9  */
10 public interface ProtocolConstant {
11
12     /**
13      * 消息头长度
14      */
15     int MESSAGE_HEADER_LENGTH = 17;
16
17     /**
18      * 协议魔数
19      */
20     byte PROTOCOL_MAGIC = 0x1;
21
22     /**
23      * 协议版本号
24      */
25     byte PROTOCOL_VERSION = 0x1;
26 }
27
```

3. 消息协议字段枚举

3.1. 序列化方式枚举

```
1 package com.yupi.yurpc.protocol;
2
3 import cn.hutool.core.util.ObjectUtil;
4 import lombok.Getter;
5
6 import java.util.Arrays;
7 import java.util.List;
8 import java.util.stream.Collectors;
9
10 /**
11  * 协议消息的序列化器枚举
12  *
13  * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
14  * @from <a href="https://yupi.icu">编程导航学习圈</a>
15  * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
16  */
17 @Getter
18 public enum ProtocolMessageSerializerEnum {
19
20     JDK(0, "jdk"),
21     JSON(1, "json"),
22     KRYO(2, "kryo"),
23     HESSIAN(3, "hessian");
24
25     private final int key;
26
27     private final String value;
28
29     ProtocolMessageSerializerEnum(int key, String value) {
30         this.key = key;
31         this.value = value;
32     }
33
34     /**
35      * 获取值列表
36      *
37      * @return
38      */
39     public static List<String> getValues() {
40         return Arrays.stream(values()).map(item -> item.value).collect(Collectors.toList());
41     }
42
43     /**
```

```

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
51 zerEnum.values()) {
52              if (anEnum.key == key) {
53                  return anEnum;
54              }
55          }
56          return null;
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
66 e) {
67          if (ObjectUtil.isEmpty(value)) {
68              return null;
69          }
70          for (ProtocolMessageSerializerEnum anEnum : ProtocolMessageSeriali
71 zerEnum.values()) {
72              if (anEnum.value.equals(value)) {
73                  return anEnum;
74              }
75          }
76          return null;
77      }

```

3.2. 消息类型枚举

```
1 package com.yupi.yurpc.protocol;
2
3 import lombok.Getter;
4
5 /**
6  * 协议消息的类型枚举
7  *
8  * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
9  * @from <a href="https://yupi.icu">编程导航学习圈</a>
10  * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
11  */
12 @Getter
13 public enum ProtocolMessageTypeEnum {
14
15     REQUEST(0),
16     RESPONSE(1),
17     HEART_BEAT(2),
18     OTHERS(3);
19
20     private final int key;
21
22     ProtocolMessageTypeEnum(int key) {
23         this.key = key;
24     }
25
26     /**
27      * 根据 key 获取枚举
28      *
29      * @param key
30      * @return
31      */
32     public static ProtocolMessageTypeEnum getEnumByKey(int key) {
33         for (ProtocolMessageTypeEnum anEnum : ProtocolMessageTypeEnum.values()) {
34             if (anEnum.key == key) {
35                 return anEnum;
36             }
37         }
38         return null;
39     }
40 }
41
```


3.3. 响应状态

```
1 package com.yupi.yurpc.protocol;
2
3 import lombok.Getter;
4
5 /**
6  * 协议消息的状态枚举
7  *
8  * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
9  * @from <a href="https://yupi.icu">编程导航学习圈</a>
10  * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
11  */
12 @Getter
13 public enum ProtocolMessageStatusEnum {
14
15     OK("ok", 20),
16     BAD_REQUEST("badRequest", 40),
17     BAD_RESPONSE("badResponse", 50);
18
19     private final String text;
20
21     private final int value;
22
23     ProtocolMessageStatusEnum(String text, int value) {
24         this.text = text;
25         this.value = value;
26     }
27
28     /**
29      * 根据 value 获取枚举
30      *
31      * @param value
32      * @return
33      */
34     public static ProtocolMessageStatusEnum getEnumByValue(int value) {
35         for (ProtocolMessageStatusEnum anEnum : ProtocolMessageStatusEnum.
values()) {
36             if (anEnum.value == value) {
37                 return anEnum;
38             }
39         }
40         return null;
41     }
42 }
43
```

4. 网络传输

使用Vertx的TCP传输

```
1 package com.yupi.yurpc.server.tcp;
2
3 import com.yupi.yurpc.server.HttpServer;
4 import io.vertx.core.Vertx;
5 import io.vertx.core.buffer.Buffer;
6 import io.vertx.core.net.NetServer;
7
8 public class VertxTcpServer implements HttpServer {
9
10     private byte[] handleRequest(byte[] requestData) {
11         // 在这里编写处理请求的逻辑, 根据 requestData 构造响应数据并返回
12         // 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
13         return "Hello, client!".getBytes();
14     }
15
16     @Override
17     public void doStart(int port) {
18         // 创建 Vert.x 实例
19         Vertx vertx = Vertx.vertx();
20
21         // 创建 TCP 服务器
22         NetServer server = vertx.createNetServer();
23
24         // 处理请求
25         server.connectHandler(socket -> {
26             // 处理连接
27             socket.handler(buffer -> {
28                 // 处理接收到的字节数组
29                 byte[] requestData = buffer.getBytes();
30                 // 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
31                 // 应等
32                 byte[] responseData = handleRequest(requestData);
33                 // 发送响应
34                 socket.write(Buffer.buffer(responseData));
35             });
36         });
37
38         // 启动 TCP 服务器并监听指定端口
39         server.listen(port, result -> {
40             if (result.succeeded()) {
41                 System.out.println("TCP server started on port " + port);
42             } else {
43                 System.err.println("Failed to start TCP server: " + result.cause());
44             }
45         });
46     }
47 }
```

```
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

```
1 package com.yupi.yurpc.protocol;
2
3 import com.yupi.yurpc.serializer.Serializer;
4 import com.yupi.yurpc.serializer.SerializerFactory;
5 import io.vertx.core.buffer.Buffer;
6
7 import java.io.IOException;
8
9 public class ProtocolMessageEncoder {
10
11     /**
12      * 编码
13      *
14      * @param protocolMessage
15      * @return
16      * @throws IOException
17      */
18     public static Buffer encode(ProtocolMessage<?> protocolMessage) throws
        IOException {
19         if (protocolMessage == null || protocolMessage.getHeader() == null) {
20             return Buffer.buffer();
21         }
22         ProtocolMessage.Header header = protocolMessage.getHeader();
23         // 依次向缓冲区写入字节
24         Buffer buffer = Buffer.buffer();
25         buffer.appendByte(header.getMagic());
26         buffer.appendByte(header.getVersion());
27         buffer.appendByte(header.getSerializer());
28         buffer.appendByte(header.getType());
29         buffer.appendByte(header.getStatus());
30         buffer.appendLong(header.getRequestId());
31         // 获取序列化器
32         ProtocolMessageSerializerEnum serializerEnum = ProtocolMessageSerializerEnum.getEnumByKey(header.getSerializer());
33         if (serializerEnum == null) {
34             throw new RuntimeException("序列化协议不存在");
35         }
36         Serializer serializer = SerializerFactory.getInstance(serializerEnum.getValue());
37         byte[] bodyBytes = serializer.serialize(protocolMessage.getBody());
38         // 写入 body 长度和数据
39         buffer.appendInt(bodyBytes.length);
```

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

5.3. 解码器

```
1 package com.yupi.yurpc.protocol;
2
3 import com.yupi.yurpc.serializer.Serializer;
4 import com.yupi.yurpc.serializer.SerializerFactory;
5 import io.vertx.core.buffer.Buffer;
6
7 import java.io.IOException;
8
9 public class ProtocolMessageEncoder {
10
11     /**
12      * 编码
13      *
14      * @param protocolMessage
15      * @return
16      * @throws IOException
17      */
18     public static Buffer encode(ProtocolMessage<?> protocolMessage) throws
19         IOException {
20         if (protocolMessage == null || protocolMessage.getHeader() == null) {
21             return Buffer.buffer();
22         }
23         ProtocolMessage.Header header = protocolMessage.getHeader();
24         // 依次向缓冲区写入字节
25         Buffer buffer = Buffer.buffer();
26         buffer.appendByte(header.getMagic());
27         buffer.appendByte(header.getVersion());
28         buffer.appendByte(header.getSerializer());
29         buffer.appendByte(header.getType());
30         buffer.appendByte(header.getStatus());
31         buffer.appendLong(header.getRequestId());
32         // 获取序列化器
33         ProtocolMessageSerializerEnum serializerEnum = ProtocolMessageSerializerEnum.getEnumByKey(header.getSerializer());
34         if (serializerEnum == null) {
35             throw new RuntimeException("序列化协议不存在");
36         }
37         Serializer serializer = SerializerFactory.getInstance(serializerEnum.getValue());
38         byte[] bodyBytes = serializer.serialize(protocolMessage.getBody());
39         // 写入 body 长度和数据
40         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rpcrequest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

```
1  package com.yupi.yurpc.server.tcp;
2
3  import com.yupi.yurpc.model.RpcRequest;
4  import com.yupi.yurpc.model.RpcResponse;
5  import com.yupi.yurpc.protocol.ProtocolMessage;
6  import com.yupi.yurpc.protocol.ProtocolMessageDecoder;
7  import com.yupi.yurpc.protocol.ProtocolMessageEncoder;
8  import com.yupi.yurpc.protocol.ProtocolMessageTypeEnum;
9  import com.yupi.yurpc.registry.LocalRegistry;
10 import io.vertx.core.Handler;
11 import io.vertx.core.buffer.Buffer;
12 import io.vertx.core.net.NetSocket;
13
14 import java.io.IOException;
15 import java.lang.reflect.Method;
16
17 public class TcpServerHandler implements Handler<NetSocket> {
18
19     @Override
20     public void handle(NetSocket netSocket) {
21         // 处理连接
22         netSocket.handler(buffer -> {
23             // 接受请求，解码
24             ProtocolMessage<RpcRequest> protocolMessage;
25             try {
26                 protocolMessage = (ProtocolMessage<RpcRequest>) ProtocolMessageDecoder.decode(buffer);
27             } catch (IOException e) {
28                 throw new RuntimeException("协议消息解码错误");
29             }
30             RpcRequest rpcRequest = protocolMessage.getBody();
31
32             // 处理请求
33             // 构造响应结果对象
34             RpcResponse rpcResponse = new RpcResponse();
35             try {
36                 // 获取要调用的服务实现类，通过反射调用
37                 Class<?> implClass = LocalRegistry.get(rpcRequest.getServiceName());
38                 Method method = implClass.getMethod(rpcRequest.getMethodName(), rpcRequest.getParameterTypes());
39                 Object result = method.invoke(implClass.newInstance(), rpcRequest.getArgs());
40                 // 封装返回结果
```

```

41         rpcResponse.setData(result);
42         rpcResponse.setDataType(method.getReturnType());
43         rpcResponse.setMessage("ok");
44     } catch (Exception e) {
45         e.printStackTrace();
46         rpcResponse.setMessage(e.getMessage());
47         rpcResponse.setException(e);
48     }
49
50     // 发送响应, 编码
51     ProtocolMessage.Header header = protocolMessage.getHeader();
52     header.setType((byte) ProtocolMessageTypeEnum.RESPONSE.getKey
53         ());
54     ProtocolMessage<RpcResponse> responseProtocolMessage = new Pro
55         tocolMessage<>(header, rpcResponse);
56     try {
57         Buffer encode = ProtocolMessageEncoder.encode(responseProt
58             ocolMessage);
59         netSocket.write(encode);
60     } catch (IOException e) {
61         throw new RuntimeException("协议消息编码错误");
62     }
63 }

```

5.5. 请求发送

```
1  package com.yupi.yurpc.proxy;
2
3  import cn.hutool.core.collection.CollUtil;
4  import cn.hutool.core.util.IdUtil;
5  import cn.hutool.http.HttpRequest;
6  import cn.hutool.http.HttpResponse;
7  import com.yupi.yurpc.RpcApplication;
8  import com.yupi.yurpc.config.RpcConfig;
9  import com.yupi.yurpc.constant.RpcConstant;
10 import com.yupi.yurpc.model.RpcRequest;
11 import com.yupi.yurpc.model.RpcResponse;
12 import com.yupi.yurpc.model.ServiceMetaInfo;
13 import com.yupi.yurpc.protocol.*;
14 import com.yupi.yurpc.registry.Registry;
15 import com.yupi.yurpc.registry.RegistryFactory;
16 import com.yupi.yurpc.serializer.Serializer;
17 import com.yupi.yurpc.serializer.SerializerFactory;
18 import io.vertx.core.Future;
19 import io.vertx.core.Vertx;
20 import io.vertx.core.buffer.Buffer;
21 import io.vertx.core.net.NetClient;
22 import io.vertx.core.net.NetSocket;
23 import io.vertx.core.net.SocketAddress;
24
25 import java.io.IOException;
26 import java.lang.reflect.InvocationHandler;
27 import java.lang.reflect.Method;
28 import java.util.List;
29 import java.util.concurrent.CompletableFuture;
30 import java.util.concurrent.CountDownLatch;
31
32 /**
33  * 服务代理 (JDK 动态代理)
34  *
35  * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
36  * @learn <a href="https://codefather.cn">编程宝典</a>
37  * @from <a href="https://yupi.icu">编程导航知识星球</a>
38  */
39 public class ServiceProxy implements InvocationHandler {
40
41     /**
42      * 调用代理
43      *
44      * @return
```

```

45     * @throws Throwable
46     */
47     @Override
48     public Object invoke(Object proxy, Method method, Object[] args) thro
ws Throwable {
49         // 指定序列化器
50         final Serializer serializer = SerializerFactory.getInstance(RpcAp
plication.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.get
RegistryConfig().getRegistry());
66             ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
67             serviceMetaInfo.setServiceName(serviceName);
68             serviceMetaInfo.setServiceVersion(RpcConstant.DEFAULT_SERVICE
_VERSION);
69             List<ServiceMetaInfo> serviceMetaInfoList = registry.serviceDi
scovery(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 Completab
leFuture<>();
78             netClient.connect(selectedServiceMetaInfo.getServicePort(), s
electServiceMetaInfo.getServiceHost(),
79             result -> {
80                 if (result.succeeded()) {
81                     System.out.println("Connected to TCP serve
r");
82

```

```

83         result();
84         // 发送数据
85         // 构造消息
86         ProtocolMessage<RpcRequest> protocolMessage
87         = new ProtocolMessage<>();
88         ProtocolMessage.Header header = new ProtocolM
89         essage.Header();
90         header.setMagic(ProtocolConstant.PROTOCOL_MAG
91         IC);
92         header.setVersion(ProtocolConstant.PROTOCOL_V
93         ERSION);
94         header.setSerializer((byte) ProtocolMessageSe
95         rializerEnum.getEnumByValue(RpcApplication.getRpcConfig().getSerializer
96         ().getKey());
97         header.setType((byte) ProtocolMessageTypeEnu
98         m.REQUEST.getKey());
99         header.setRequestId(IdUtil.getSnowflakeNextId
100         ());
101         protocolMessage.setHeader(header);
102         protocolMessage.setBody(rpcRequest);
103         // 编码请求
104         try {
105             Buffer encodeBuffer = ProtocolMessageEnco
106             der.encode(protocolMessage);
107             socket.write(encodeBuffer);
108         } catch (IOException e) {
109             throw new RuntimeException("协议消息编码错
110             误");
111         }
112         // 接收响应
113         socket.handler(buffer -> {
114             try {
115                 ProtocolMessage<RpcResponse> rpcRespo
116                 nseProtocolMessage = (ProtocolMessage<RpcResponse>) ProtocolMessageDecode
117                 r.decode(buffer);
118                 responseFuture.complete(rpcResponsePr
119                 otocolMessage.getBody());
120             } catch (IOException e) {
121                 throw new RuntimeException("协议消息解
122                 码错误");
123             }
124         });
125     } else {
126         System.err.println("Failed to connect to TCP
127         server");

```

```

114         }
115     });
116
117     RpcResponse rpcResponse = responseFuture.get();
118     // 记得关闭连接
119     netClient.close();
120     return rpcResponse.getData();
121 } catch (IOException e) {
122     e.printStackTrace();
123 }
124
125 return null;
126 }

```

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

```
1 package com.yupi.yurpc.server.tcp;
2
3 import com.yupi.yurpc.protocol.ProtocolConstant;
4 import com.yupi.yurpc.server.HttpServer;
5 import io.vertx.core.Handler;
6 import io.vertx.core.Vertx;
7 import io.vertx.core.buffer.Buffer;
8 import io.vertx.core.net.NetServer;
9 import io.vertx.core.parsetools.RecordParser;
10 import lombok.extern.slf4j.Slf4j;
11
12 @Slf4j
13 public class VertxTcpServer implements HttpServer {
14
15     @Override
16     public void doStart(int port) {
17         // 创建 Vert.x 实例
18         Vertx vertx = Vertx.vertx();
19
20         // 创建 TCP 服务器
21         NetServer server = vertx.createNetServer();
22
23         // 处理请求
24         server.connectHandler(socket -> {
25             // 构造 parser
26             RecordParser parser = RecordParser.newFixed(8);
27             parser.setOutput(new Handler<Buffer>() {
28                 // 初始化
29                 int size = -1;
30                 // 一次完整的读取 (头 + 体)
31                 Buffer resultBuffer = Buffer.buffer();
32
33                 @Override
34                 public void handle(Buffer buffer) {
35                     if (-1 == size) {
36                         // 读取消息体长度
37                         size = buffer.getInt(4);
38                         parser.fixedSizeMode(size);
39                         // 写入头信息到结果
40                         resultBuffer.appendBuffer(buffer);
41                     } else {
42                         // 写入体信息到结果
43                         resultBuffer.appendBuffer(buffer);
44                         System.out.println(resultBuffer.toString());
45                     }
46                 }
47             });
48         });
49     }
50 }
```



```

45         // 重置一轮
46         parser.fixedSizeMode(8);
47         size = -1;
48         resultBuffer = Buffer.buffer();
49     }
50 }
51 });
52
53     socket.handler(parser);
54 });
55
56 // 启动 TCP 服务器并监听指定端口
57 server.listen(port, result -> {
58     if (result.succeeded()) {
59         log.info("TCP server started on port " + port);
60     } else {
61         log.info("Failed to start TCP server: " + result.cause());
62     }
63 });
64 }
65
66 public static void main(String[] args) {
67     new VertxTcpServer().doStart(8888);
68 }
69 }
70

```

```
1 package com.yupi.yurpc.server.tcp;
2
3 import io.vertx.core.Vertx;
4 import io.vertx.core.buffer.Buffer;
5
6 public class VertxTcpClient {
7
8     public void start() {
9         // 创建 Vert.x 实例
10        Vertx vertx = Vertx.vertx();
11
12        vertx.createNetClient().connect(8888, "localhost", result -> {
13            if (result.succeeded()) {
14                System.out.println("Connected to TCP server");
15                io.vertx.core.net.NetSocket socket = result.result();
16                for (int i = 0; i < 1000; i++) {
17                    // 发送数据
18                    Buffer buffer = Buffer.buffer();
19                    String str = "Hello, server!Hello, server!Hello, serve
20                    r!Hello, server!";
21                    buffer.appendInt(0);
22                    buffer.appendInt(str.getBytes().length);
23                    buffer.appendBytes(str.getBytes());
24                    socket.write(buffer);
25                }
26                // 接收响应
27                socket.handler(buffer -> {
28                    System.out.println("Received response from server: "
29                    + buffer.toString());
30                });
31            } else {
32                System.err.println("Failed to connect to TCP server");
33            }
34        });
35
36        public static void main(String[] args) {
37            new VertxTcpClient().start();
38        }
39    }
```

```
1 package com.yupi.yurpc.server.tcp;
2
3 import com.yupi.yurpc.protocol.ProtocolConstant;
4 import io.vertx.core.Handler;
5 import io.vertx.core.buffer.Buffer;
6 import io.vertx.core.parsetools.RecordParser;
7
8 /**
9  * 装饰者模式 (使用 recordParser 对原有的 buffer 处理能力进行增强)
10 */
11 public class TcpBufferHandlerWrapper implements Handler<Buffer> {
12
13     private final RecordParser recordParser;
14
15     public TcpBufferHandlerWrapper(Handler<Buffer> bufferHandler) {
16         recordParser = initRecordParser(bufferHandler);
17     }
18
19     @Override
20     public void handle(Buffer buffer) {
21         recordParser.handle(buffer);
22     }
23
24     private RecordParser initRecordParser(Handler<Buffer> bufferHandler) {
25         // 构造 parser
26         RecordParser parser = RecordParser.newFixed(ProtocolConstant.MESSA
GE_HEADER_LENGTH);
27
28         parser.setOutput(new Handler<Buffer>() {
29             // 初始化
30             int size = -1;
31             // 一次完整的读取 (头 + 体)
32             Buffer resultBuffer = Buffer.buffer();
33
34             @Override
35             public void handle(Buffer buffer) {
36                 if (-1 == size) {
37                     // 读取消息体长度
38                     size = buffer.getInt(13);
39                     parser.fixedSizeMode(size);
40                     // 写入头信息到结果
41                     resultBuffer.appendBuffer(buffer);
42                 } else {
43                     // 写入体信息到结果
```

```

44         resultBuffer.appendBuffer(buffer);
45         // 已拼接为完整 Buffer, 执行处理
46         bufferHandler.handle(resultBuffer);
47         // 重置一轮
48         parser.fixedSizeMode(ProtocolConstant.MESSAGE_HEADER_L
    ENGTH);
49         size = -1;
50         resultBuffer = Buffer.buffer();
51     }
52 }
53 });
54
55     return parser;
56 }
57 }
58

```