Netty通信技术进阶

1、hello word

1.1、服务端

1、创建`NettServer`

```
public class NettServer {

    private static final Logger log =
LoggerFactory.getLogger(NettServer.class);

    public static void main(String[] args) {
        NettServer nettServer = new NettServer();
        nettServer.start(8888);
    }

    public void start(int port) {
        //创建bossGroup workerGroup
        EventLoopGroup boss = new NioEventLoopGroup();
        EventLoopGroup worker = new NioEventLoopGroup();
        try {
            //创建启动类
            ServerBootstrap serverBootstrap = new
ServerBootstrap();
            //配置参数
            serverBootstrap.group(boss,worker)
                    .channel(NioServerSocketChannel.class)
//指定服务端通道，用于接收并创建新连接
                    .handler(new
LoggingHandler(LogLevel.DEBUG)) // 给boss group 配置handler
                    .childHandler(new
ChannelInitializer<SocketChannel>() {

                        //每个客户端channel初始化时都会执行该
方法来配置该channel的相关handler
                        @Override
```

```
                        protected void initChannel(SocketChannel ch) throws Exception {
                            //获取与该channel绑定的pipeline
                            ChannelPipeline pipeline = ch.pipeline();
                            //像pipeline中添加handler
                            pipeline.addLast(new ServerInboundHandler1());
                            pipeline.addLast(new ServerInboundHandler2());
                        }
                    }); //给worker group配置handler
            //服务端绑定端口启动
            ChannelFuture future = serverBootstrap.bind(port).sync();
            //服务端监听端口关闭
            future.channel().closeFuture().sync();
        } catch (Exception e) {
            log.error("netty server error , {}",e.getMessage());
        } finally {
            //优雅关闭 boss worker
            boss.shutdownGracefully();
            worker.shutdownGracefully();
        }
    }
}
```

2、创建inbound类型的Handler1

```
/**
 * @description
 * @author: ts
 * @create:2021-04-22 09:40
 */
public class ServerInboundHandler1 extends ChannelInboundHandlerAdapter {
    private static final Logger log = LoggerFactory.getLogger(ServerInboundHandler1.class);
    /**
     * 通道准备就绪时
     * @param ctx
     * @throws Exception
```

```
12          */
13         @Override
14         public void channelActive(ChannelHandlerContext ctx)
throws Exception {
15             log.info("ServerInboundHandler1 channelActive-----
");
16
17             //将事件向下传递
18             ctx.fireChannelActive();
19         }
20
21         /**
22          * 通道有数据可读时
23          * @param ctx
24          * @param msg
25          * @throws Exception
26          */
27         @Override
28         public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
29             log.info("ServerInboundHandler1 channelRead---
-,remoteAddress={}",ctx.channel().remoteAddress());
30             //处理接收的数据
31             ByteBuf buf = (ByteBuf) msg;
32             log.info("ServerInboundHandler1:received client
data = {}",buf.toString(StandardCharsets.UTF_8));
33
34             //将事件消息向下传递
35             ctx.fireChannelRead(msg);
36         }
37
38         /**
39          * 数据读取完毕时
40          * @param ctx
41          * @throws Exception
42          */
43         @Override
44         public void channelReadComplete(ChannelHandlerContext
ctx) throws Exception {
45             log.info("ServerInboundHandler1
channelReadComplete----");
46         }
47
```

```
    /**
     * 发生异常时
     * @param ctx
     * @param cause
     * @throws Exception
     */
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
Throwable cause) throws Exception {
        log.info("ServerInboundHandler1 exceptionCaught---
-,cause={}",cause.getMessage());
    }
}
```

3、创建inbound类型的Handler2

```
/**
 * @description
 * @author: ts
 * @create:2021-04-22 10:55
 */
public class ServerInboundHandler2 extends
ChannelInboundHandlerAdapter {
    private static final Logger log =
LoggerFactory.getLogger(ServerInboundHandler2.class);


    @Override
    public void channelActive(ChannelHandlerContext ctx)
throws Exception {
        log.info("ServerInboundHandler2 channelActive-----
");
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
        log.info("ServerInboundHandler2 channelRead---
-,remoteAddress={}",ctx.channel().remoteAddress());
        //处理接收的数据
        ByteBuf buf = (ByteBuf) msg;
        log.info("ServerInboundHandler2:received client
data = {}",buf.toString(StandardCharsets.UTF_8));
```

```
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
        log.info("ServerInboundHandler2 channelReadComplete----");
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
    }
}
```

4、在ServerInboundHandler1中向外写数据

```
/**
     * 数据读取完毕时
     * @param ctx
     * @throws Exception
     */
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
        log.info("channelReadComplete----");
        //数据读取结束后向客户端写回数据
        /*byte[] data = "hello client , i am server".getBytes(StandardCharsets.UTF_8);
        ByteBuf buffer = Unpooled.buffer(data.length);
        buffer.writeBytes(data);//以bytebuf为中心,看是写到bytebuf中还是从bytebuf中读*/
        ByteBuf buf = Unpooled.copiedBuffer("hello client , i am server",StandardCharsets.UTF_8);
        ctx.writeAndFlush(buf);//通过ctx写，事件会从当前handler向pipeline头部移动
        //ctx.channel().writeAndFlush(buf);//通过Channel写，事件会从通道尾部向头部移动
    }
```

5、创建outbound类型的Handler

```
/**
 * @description
 * @author: ts
 * @create:2021-04-22 09:44
 */
public class ServerOutboundHandler1 extends ChannelOutboundHandlerAdapter {
    private static final Logger log = LoggerFactory.getLogger(ServerOutboundHandler1.class);

    @Override
    public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
        ByteBuf buf = (ByteBuf) msg;
        log.info("OutboundHandler1----server send msg to client,msg ={}",buf.toString(StandardCharsets.UTF_8));

 //ctx.channel().writeAndFlush(Unpooled.copiedBuffer("okokokok".getBytes(StandardCharsets.UTF_8)));//利用channel在outboundhandler中再写数据会引起类似递归的调用，数据再从pipeline尾部流向头部
        super.write(ctx,buf,promise);//将事件向前传播,父类中调用了ctx.write(msg,promise);

 ctx.writeAndFlush(Unpooled.copiedBuffer("nonono".getBytes(StandardCharsets.UTF_8)));//用ctx写数据，代码写到super之后，该事件会流到该handler之前的handler进行处理
    }
}
```

6、像pipeline中添加outbound类型的handler

```
.childHandler(new ChannelInitializer<SocketChannel>() {

    //每个客户端channel初始化时都会执行该方法来配置该channel的相关handler
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
```

```
        //获取与该channel绑定的pipeline
        ChannelPipeline pipeline = ch.pipeline();
        //像pipeline中添加handler
        pipeline.addLast(new ServerOutboundHandler1());

        pipeline.addLast(new ServerInboundHandler1());
        pipeline.addLast(new ServerInboundHandler2());
    }
});
```

1.2、客户端

1、创建`NettyClient`

```
public class NettyClient {

    private static final Logger log =
LoggerFactory.getLogger(NettyClient.class);

    public static void main(String[] args) {
        NettyClient client = new NettyClient();
        client.start("127.0.0.1", 8888);
    }

    public void start(String host,int port) {
        EventLoopGroup group = new NioEventLoopGroup();
        try {
            Bootstrap bootstrap = new Bootstrap();
            bootstrap.group(group)
                    .channel(NioSocketChannel.class)
                    .handler(new
ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void
initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline pipeline =
ch.pipeline();
                            //添加客户端channel对应的handler
                            pipeline.addLast(new
ClientInboundHandler1());
                        }
```

```
                    });
            //连接远程启动
            ChannelFuture future = bootstrap.connect(host,
port).sync();
            //监听通道关闭
            future.channel().closeFuture().sync();
        } catch (Exception e) {
            log.error("netty client error ,msg=
{}",e.getMessage());
        } finally {
            //优雅关闭
            group.shutdownGracefully();
        }
    }
}
```

2、创建`ClientInboundHandler1`

```
public class ClientInboundHandler1 extends
ChannelInboundHandlerAdapter {
    private static final Logger log =
LoggerFactory.getLogger(ClientInboundHandler1.class);
    /**
     * 通道准备就绪
     * @param ctx
     * @throws Exception
     */
    @Override
    public void channelActive(ChannelHandlerContext ctx)
throws Exception {
        log.info("ClientInboundHandler1 channelActive
begin send data");
        //通道准备就绪后开始向服务端发送数据
        ByteBuf buf = Unpooled.copiedBuffer("hello
server,i am client".getBytes(StandardCharsets.UTF_8));
        ctx.writeAndFlush(buf);
    }

    /**
     * 通道有数据可读（服务端返回了数据）
     * @param ctx
```

```
     * @param msg
     * @throws Exception
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
        log.info("ClientInboundHandler1 channelRead");
        ByteBuf buf = (ByteBuf) msg;
        log.info("ClientInboundHandler1: received server
data ={}",buf.toString(StandardCharsets.UTF_8));
    }

    /**
     * 数据读取完毕
     * @param ctx
     * @throws Exception
     */
    @Override
    public void channelReadComplete(ChannelHandlerContext
ctx) throws Exception {
        super.channelReadComplete(ctx);
    }

    /**
     * 产生了异常
     * @param ctx
     * @param cause
     * @throws Exception
     */
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx,
Throwable cause) throws Exception {
        super.exceptionCaught(ctx, cause);
    }
}
```

3、测试使用`SimpleChannelInboundHandler`

创建客户端`ClientSimpleInboundHandler2`

```
public class ClientSimpleInboundHandler2 extends SimpleChannelInboundHandler<ByteBuf> {
    private static final Logger log = LoggerFactory.getLogger(ClientSimpleInboundHandler2.class);
    @Override
    protected void channelRead0(ChannelHandlerContext ctx, ByteBuf msg) throws Exception {
        log.info("ClientSimpleInboundHandler2 channelRead");
        log.info("ClientSimpleInboundHandler2: received server data ={}",msg.toString(StandardCharsets.UTF_8));
    }
}
```

在客户端`pipeline`中添加该`handler`

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        ChannelPipeline pipeline = ch.pipeline();
        //添加客户端channel对应的handler
        pipeline.addLast(new ClientInboundHandler1());
        pipeline.addLast(new ClientSimpleInboundHandler2());
    }
});
```

注意去：`ClientInboundHandler1`中的`channelRead`方法中讲事件传播到下一个`handler`中。

2、ByteBuf

编写测试类：

```
public class ByteBufTest {

    private static final Logger log = LoggerFactory.getLogger(ByteBufTest.class);

    @Test
```

```
6       public void testRead() {
7           //构造有数据的bytebuf
8           ByteBuf buf = Unpooled.copiedBuffer("hello
    netty".getBytes(StandardCharsets.UTF_8));
9           log.info("bytebuf容量为:{}",buf.capacity());
10          log.info("bytebuf可读容量为:
    {}",buf.readableBytes());
11          log.info("bytebuf可写容量为:
    {}",buf.writableBytes());
12          //按字节读取
13          while (buf.isReadable()) {
14              log.info(""+buf.readByte());
15          }
16          //通过下标读取
17          for (int i=0;i<buf.readableBytes();i++) {
18              log.info(""+buf.getByte(i));//此种方式不会变更
    readerIndex,即后续还可继续读取
19          }
20          //直接读到字节数组
21          byte[] bytes = new byte[buf.readableBytes()];
22          buf.readBytes(bytes);
23          log.info("---"+new String(bytes));
24      }
25
26
27      @Test
28      public void testWrite() {
29          //准备好要写入的数据
30          byte[] bytes = "hello
    netty".getBytes(StandardCharsets.UTF_8);
31
32          //构造bytebuf,给定初始容量和最大容量
33          ByteBuf buf = Unpooled.buffer(10, 1024);
34
35          //每次写1个字节进去
36          for (int i=0;i<100;i++) {
37              buf.writeByte(i);
38          }
39          log.info("bytebuf容量为:{}",buf.capacity());
40          log.info("bytebuf可读容量为:
    {}",buf.readableBytes());
41          log.info("bytebuf可写容量为:
    {}",buf.writableBytes());
```

```
42          log.info("-------------------华丽的分割线----------
------------");
43
44          //直接将bytes中的数据写入bytebuf
45          buf.writeBytes(bytes);
46          log.info("bytebuf容量为:{}",buf.capacity());
47          log.info("bytebuf可读容量为:
{}",buf.readableBytes());
48          log.info("bytebuf可写容量为:
{}",buf.writableBytes());
49      }
50
51      @Test
52      public void testDiscard() {
53          //创建带数据的bytebuf
54          ByteBuf buf = Unpooled.copiedBuffer("hello
netty".getBytes(StandardCharsets.UTF_8));
55          log.info("bytebuf容量为:{}",buf.capacity());
56          log.info("bytebuf可读容量为:
{}",buf.readableBytes());
57          log.info("bytebuf可写容量为:
{}",buf.writableBytes());
58          log.info("-------------------华丽的分割线----------
------------");
59          for (int i=0;i<5;i++) {
60              buf.readByte();
61          }
62          log.info("bytebuf容量为:{}",buf.capacity());
63          log.info("bytebuf可读容量为:
{}",buf.readableBytes());
64          log.info("bytebuf可写容量为:
{}",buf.writableBytes());
65          log.info("-------------------华丽的分割线----------
------------");
66          //丢弃已读取的字节空间---可写空间变多
67          buf.discardReadBytes();
68          log.info("bytebuf容量为:{}",buf.capacity());
69          log.info("bytebuf可读容量为:
{}",buf.readableBytes());
70          log.info("bytebuf可写容量为:
{}",buf.writableBytes());
71          log.info("-------------------华丽的分割线----------
------------");
```

```
72          //clear 将readerIndex=writerIndex=0;只是指针变化,数据并没有清除,支持继续写入
73          buf.clear();
74          log.info("bytebuf容量为:{}",buf.capacity());
75          log.info("bytebuf可读容量为:{}",buf.readableBytes());
76          log.info("bytebuf可写容量为:{}",buf.writableBytes());
77          log.info("--------------------华丽的分割线----------");
78          buf.writeBytes("hello netty".getBytes(StandardCharsets.UTF_8));
79          log.info("bytebuf容量为:{}",buf.capacity());
80          log.info("bytebuf可读容量为:{}",buf.readableBytes());
81          log.info("bytebuf可写容量为:{}",buf.writableBytes());
82          log.info("--------------------华丽的分割线----------");
83
84          //释放消息
85          //buf.release();
86          ReferenceCountUtil.release(buf);//里面的数据已经被释放了 readerIndex=0,writerIndex在最后,数组容量为0
87          log.info("bytebuf容量为:{}",buf.capacity());
88          log.info("bytebuf可读容量为:{}",buf.readableBytes());//此时已经没有数据了
89          log.info("bytebuf可写容量为:{}",buf.writableBytes());
90      }
91
92      @Test
93      public void testWrap() {
94          byte[] bytes = "123456".getBytes(StandardCharsets.UTF_8);
95          ByteBuf buf = Unpooled.wrappedBuffer(bytes);
96          bytes[0] = 1;
97          //写入
98          log.info("原始数据:{}",new String(bytes,StandardCharsets.UTF_8));
99          log.info("buf中的数据:{}",buf.toString(StandardCharsets.UTF_8));
```

```
100         log.info("-------------------华丽的分割线---------
    -----------");
101
102         byte[] bytes1 =
    "hello".getBytes(StandardCharsets.UTF_8);
103         ByteBuf buf1 = Unpooled.copiedBuffer(bytes1);
104         bytes1[0] = 1;
105         //写入
106         log.info("原始数据:{}",new
    String(bytes1,StandardCharsets.UTF_8));
107         log.info("buf中的数据:
    {}",buf1.toString(StandardCharsets.UTF_8));
108     }
109
110      @Test
111     public void testUnpooledByteBuf() {
112         ByteBuf buf = new
    UnpooledDirectByteBuf(UnpooledByteBufAllocator.DEFAULT,10
    0,1024);
113
     buf.writeBytes("hello".getBytes(StandardCharsets.UTF_8))
    ;
114         log.info("---buf的类型:
    {}",buf.getClass().getTypeName());
115
116         ByteBuf buf1 = new
    UnpooledHeapByteBuf(UnpooledByteBufAllocator.DEFAULT,100,
    1024);
117         buf1.writeBytes("hello
    word".getBytes(StandardCharsets.UTF_8));
118         log.info("---buf1的类型是:
    {}",buf1.getClass().getTypeName());
119     }
120 }
```

3、JDK Future

```
1 public class JdkFutureTest {
2
```

```
    private static final Logger log =
LoggerFactory.getLogger(JdkFutureTest.class);

    @Test
    public void testFuture() throws InterruptedException
{
        ExecutorService executorService =
Executors.newFixedThreadPool(1);
        log.info("-------主线程提交runnable任务-------");
        // 提交runnable类型的任务,任务执行完毕后提交者无法获取
执行结果,没有返回
        executorService.execute(new Runnable() {
            @Override
            public void run() {
                log.info("---异步线程---执行任务");
                try {
                    TimeUnit.SECONDS.sleep(3);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }
        });
        log.info("-------主线程继续执行-------");
        log.info("----------------------华丽的分割线------
------------------");

        log.info("-------主线程提交callable任务---第一种写
法-------");
        //提交callable任务，是有返回的，提交者可以获取异步线程
执行结果
        Future<String> future =
executorService.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
               log.info("线程池中的异步线程正在执行任务-----
");
                TimeUnit.SECONDS.sleep(3);
                log.info("异步线程执行完毕,返回结果");
                 return "async task result";
            }
        });
        //提交者阻塞等待获取异步线程执行结果
        try {
```

```
            log.info("-------主线程等待callable任务返回结果-------");
            String result = future.get();
            log.info("提交者线程中获取到的异步线程执行结果是:{}",result);
        } catch (ExecutionException e) {
            log.error("提交者阻塞等待异步线程的执行结果异常,{}",e.getMessage());
        }

        log.info("----------------------华丽的分割线----------------------");

        log.info("-------主线程提交callable任务---第二种写法-------");
        FutureTask<String> task = new FutureTask<String>(new Callable<String>() {
            @Override
            public String call() throws Exception {
                log.info("线程池中的异步线程正在执行任务-----");
                TimeUnit.SECONDS.sleep(3);
                log.info("异步线程执行完毕,返回结果");
                return "async task result2";
            }
        });
        executorService.submit(task);
        //提交者阻塞等待获取异步线程执行结果
        try {
            log.info("-------主线程等待callable任务返回结果-------");
            String result = task.get();
            log.info("提交者线程中获取到的异步线程执行结果是:{}",result);
        } catch (ExecutionException e) {
            log.error("提交者阻塞等待异步线程的执行结果异常,{}",e.getMessage());
        }
    }

    /**
```

```
68      * 通过callable+future，虽然可以在提交线程中拿到异步线程的执行结果，但它并不是真正的异步，没有实现回调，提交线程中仍然需要通过get()
69      * 阻塞等待,所以在Java8 中又新增了一个真正的异步函数：CompletableFuture
70      *
71      * Java 8 中新增加了一个类：CompletableFuture，它提供了非常强大的 Future 的扩展功能，最重要的是实现了回调的功能
72      */
73
74     @Test
75     public void testCompletableFuture() throws InterruptedException {
76         /**
77          * 异步非阻塞,执行无返回值任务,
78          * public static CompletableFuture<Void> runAsync(Runnable runnable){..}
79          *
80          * public static CompletableFuture<Void> runAsync(Runnable runnable,Executor executor){..}
81          * 同时支持传入自定义的线程池，如果不传入线程池的话默认是使用 ForkJoinPool.commonPool()作为它的线程池执行异步代码
82          *
83          */
84         CompletableFuture.runAsync(new Runnable() {
85             @Override
86             public void run() {
87                 log.info("---开始异步线程执行任务----");
88                 try {
89                     TimeUnit.SECONDS.sleep(3);
90                 } catch (InterruptedException e) {
91                     e.printStackTrace();
92                 }
93                 log.info("---异步线程执行任务结束----");
94             }
95         });
96         log.info("---主线程----");
97         TimeUnit.SECONDS.sleep(5);
98     }
99
100
101    @Test
```

```
    public void testCompletableSupplyAsync() throws InterruptedException {
        /**
         * 异步非阻塞，执行有返回值任务，
         * public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier){..}
         * public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier,Executor executor){..}
         * 同时支持传入自定义的线程池，如果不传入线程池的话默认是使用 ForkJoinPool.commonPool()作为它的线程池执行异步代码
         */
        //使用自定义的线程池
        Executor executor = Executors.newFixedThreadPool(10);
        CompletableFuture<String> cf = CompletableFuture.supplyAsync(new Supplier<String>() {
            @Override
            public String get() {
                log.info("---开始异步线程执行任务----");
                try {
                    TimeUnit.SECONDS.sleep(3);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                log.info("---异步线程执行任务结束----");
                return "CompletableFuture returned result";
            }
        }, executor);

        // 设置回调（监听）如果执行成功:
        cf.thenAccept(new Consumer<String>() {
            @Override
            public void accept(String s) {
                log.info("异步通知的结果是:{}",s);
            }
        }).exceptionally((e)->{
            log.info("异步执行产生了异常,异常信息是:{}",e.getMessage());
            return null;
        });
        log.info("----主线程无需阻塞等待,继续执行其他业务");
        TimeUnit.SECONDS.sleep(10);
```

```
137     }
138
139     /**
140      * CompletableFuture的优点是：
141      *
142      * 1、异步任务结束时，会自动回调某个对象的方法；
143      * 2、异步任务出错时，会自动回调某个对象的方法；
144      * 3、主线程设置好回调后，不再关心异步任务的执行。
145      * 另外注意CompletableFuture 某些系列方法的命名规则：
146      *  xxx()：表示该方法将继续在已有的线程中执行；
147      *  xxxAsync()：表示该方法在另外的线程池中执行(特别针对的是指定了自定义线程池后,xxxAsyn()方法会在CompletableFuture内部默认的ForkJoinPool中执行)
148      *
149      *  如果只是实现了异步回调机制，我们还看不出CompletableFuture相比Future的优势。
150      *  CompletableFuture更强大的功能是，多个CompletableFuture可以串行执行，可以并行执行，合并两个异步任务，下一个依赖上一个的结果等等，
151      *
152      */
153
154     @Test
155     public void testSerialize() throws InterruptedException {
156         //测试多个CompletableFuture串行执行
157         //第一个任务
158         CompletableFuture<String> cf = CompletableFuture.supplyAsync(() -> {
159             log.info("---第一个异步线程执行任务----,time={}", LocalDateTime.now().toString());
160             try {
161                 TimeUnit.SECONDS.sleep(3);
162             } catch (InterruptedException e) {
163                 e.printStackTrace();
164             }
165             log.info("---第一个异步线程执行任务结束----,time={}", LocalDateTime.now().toString());
166             return "hello";
167         });
168         //第一个任务成功后继续执行下一个任务
169         CompletableFuture<String> cf2 = cf.thenApply((pre) -> {
```

```
            log.info("---第二个异步线程执行任务,接收到的第一个异步线程的执行结果是:{},time={}", pre, LocalDateTime.now().toString());
            return pre + " word";
        });

        //设置回调
        cf2.thenAccept((result)->{
            log.info("两个异步任务总的结果是:{}",result);
        }).exceptionally((e)->{
            log.info("异步执行产生了异常,异常信息是:{}",e.getMessage());
            return null;
        });

        log.info("----主线程无需阻塞等待,继续执行其他业务");
        TimeUnit.SECONDS.sleep(10);
    }


    @Test
    public void testAnyOf() throws InterruptedException {
        //测试多个CompletableFuture 并行执行; 只要其中任意一个返回就可以继续往下执行
        //第一个任务
        CompletableFuture<String> cf1 = CompletableFuture.supplyAsync(() -> {
            log.info("---第一个异步线程执行任务开始---,time={}", LocalDateTime.now().toString());
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.info("---第一个异步线程执行任务结束---,time={}", LocalDateTime.now().toString());
            return "hello";
        });
        //第二个任务
        CompletableFuture<String> cf2 = CompletableFuture.supplyAsync(() -> {
            log.info("---第二个异步线程执行任务开始---,time={}", LocalDateTime.now().toString());
```

```
            try {
                TimeUnit.SECONDS.sleep(3);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            log.info("---第二个异步线程执行任务结束---
-,time={}", LocalDateTime.now().toString());
            return " word ";
        });

        /**
         * 将两个CompletableFuture合并为一个新的
CompletableFuture
         * anyOf()和allOf()用于并行化多个
CompletableFuture。
         * anyof:两个任务任意一个返回即产生回调
         */
        CompletableFuture<Object> cf3 =
CompletableFuture.anyOf(cf1, cf2);
        //设置回调
        cf3.thenAccept((result)->{
            log.info("两个异步任务的结果是:{}",result);
        }).exceptionally((e)->{
            log.info("异步执行产生了异常,异常信息是:
{}",e.getMessage());
            return null;
        });


        log.info("----主线程无需阻塞等待,继续执行其他业务");
        TimeUnit.SECONDS.sleep(10);
    }


    @Test
    public void testCombine() throws InterruptedException
{
        /**
         * 如果有两个任务需要异步执行，且后面需要对这两个任务的
结果进行合并处理，CompletableFuture 也支持这种处理：
         */
        Executor executor =
Executors.newFixedThreadPool(10);
```

```
239         //第一个任务
240         CompletableFuture<String> cf1 =
CompletableFuture.supplyAsync(() -> {
241             log.info("---第一个异步线程执行任务开始---
-,time={}", LocalDateTime.now().toString());
242             try {
243                 TimeUnit.SECONDS.sleep(3);
244             } catch (InterruptedException e) {
245                 e.printStackTrace();
246             }
247             log.info("---第一个异步线程执行任务结束---
-,time={}", LocalDateTime.now().toString());
248             return "hello";
249         },executor);
250         //第二个任务
251         CompletableFuture<String> cf2 =
CompletableFuture.supplyAsync(() -> {
252             log.info("---第二个异步线程执行任务开始---
-,time={}", LocalDateTime.now().toString());
253             try {
254                 TimeUnit.SECONDS.sleep(3);
255             } catch (InterruptedException e) {
256                 e.printStackTrace();
257             }
258             log.info("---第二个异步线程执行任务结束---
-,time={}", LocalDateTime.now().toString());
259             return " word ";
260         },executor);
261
262         //对两个任务的结果进行合并
263         CompletableFuture<String> cf3 =
cf1.thenCombineAsync(cf2, (task1, task2) -> {
264             log.info("两个异步任务的结果分别是{}:{}", task1,
task2);
265             return task1 + "******" + task2;
266         });
267         //设置回调
268         cf3.thenAcceptAsync((totalResult)->{
269             log.info("两个异步任务合并后的结果是:
{}",totalResult);
270         }).exceptionally((e)->{
271             log.info("异步执行产生了异常,异常信息是:
{}",e.getMessage());
```

```
272             return null;
273         });
274 
275 
276         log.info("----主线程无需阻塞等待,继续执行其他业务");
277         TimeUnit.SECONDS.sleep(10);
278     }
279 
280     /**
281      * 常用 API 介绍
282      * 1、拿到上一个任务的结果做后续操作，上一个任务完成后的动作
283      * public CompletableFuture<T>
    whenComplete(BiConsumer<? super T,? super Throwable>
    action)
284      * public CompletableFuture<T>
    whenCompleteAsync(BiConsumer<? super T,? super Throwable>
    action)
285      * public CompletableFuture<T>
    whenCompleteAsync(BiConsumer<? super T,? super Throwable>
    action, Executor executor)
286      * public CompletableFuture<T>
    exceptionally(Function<Throwable,? extends T> fn)
287      * 上面四个方法表示在当前阶段任务完成之后下一步要做什么。
288      *
289      * 2、拿到上一个任务的结果做后续操作，使用 handler 来处理逻
    辑，可以返回与第一阶段处理的返回类型不一样的返回类型。
290      * public <U> CompletableFuture<U>
    handle(BiFunction<? super T,Throwable,? extends U> fn)
291      * public <U> CompletableFuture<U>
    handleAsync(BiFunction<? super T,Throwable,? extends U>
    fn)
292      * public <U> CompletableFuture<U>
    handleAsync(BiFunction<? super T,Throwable,? extends U>
    fn, Executor executor)
293      * Handler 与 whenComplete 的区别是 handler 是可以返回一
    个新的 CompletableFuture 类型的。
294      *
295      * CompletableFuture<Integer> f1 =
    CompletableFuture.supplyAsync(() -> {
296      *     return "hahaha";
297      * }).handle((r, e) -> {
298      *     return 1;
299      * });
```

```
300         *
301         *
302         * 3、拿到上一个任务的结果做后续操作， thenApply方法
303         * public <U> CompletableFuture<U>
    thenApply(Function<? super T,? extends U> fn)
304         * public <U> CompletableFuture<U>
    thenApplyAsync(Function<? super T,? extends U> fn)
305         * public <U> CompletableFuture<U>
    thenApplyAsync(Function<? super T,? extends U> fn,
    Executor executor)
306         * 注意到 thenApply 方法的参数中是没有 Throwable，这就意
    味着如有有异常就会立即失败，不能在处理逻辑内处理。且 thenApply
    返回的也是新的 CompletableFuture。 这就是它与前面两个的区别。
307         *
308         * 4、拿到上一个任务的结果做后续操作，可以不返回任何值，
    thenAccept方法
309         * public CompletableFuture<Void>
    thenAccept(Consumer<? super T> action)
310         * public CompletableFuture<Void>
    thenAcceptAsync(Consumer<? super T> action)
311         * public CompletableFuture<Void>
    thenAcceptAsync(Consumer<? super T> action, Executor
    executor)
312         * 看这里的示例：
313         *
314         * CompletableFuture.supplyAsync(() -> {
315         *   return "result";
316         * }).thenAccept(r -> {
317         *   System.out.println(r);
318         * }).thenAccept(r -> {
319         *   System.out.println(r);
320         * });
321         * 执行完毕是不会返回任何值的。
322         *
323         * CompletableFuture 的特性提现在执行完 runAsync 或者
    supplyAsync 之后的操作上。
324         * CompletableFuture 能够将回调放到与任务不同的线程中执
    行，也能将回调作为继续执行的同步函数，在与任务相同的线程中执行。它
    避免了传统回调最大的问题，那就是能够将控制流分离到不同的事件处理器
    中。
325         *
```

```
326         * 另外当你依赖 CompletableFuture 的计算结果才能进行下一步的时候，无需手动判断当前计算是否完成，可以通过 CompletableFuture 的事件监听自动去完成。
327         *
328         */
329 }
```

4、Netty Future/Promise

```
1  public class NettyFutureTest {
2      private static final Logger log = LoggerFactory.getLogger(NettyFutureTest.class);
3
4
5      @Test
6      public void testFuture() throws InterruptedException, ExecutionException {
7          EventLoopGroup group = new NioEventLoopGroup();
8          Future<String> future = group.submit(new Callable<String>() {
9              @Override
10             public String call() throws Exception {
11                 log.info("---异步线程执行任务开始----,time={}", LocalDateTime.now().toString());
12                 try {
13                     TimeUnit.SECONDS.sleep(3);
14                 } catch (InterruptedException e) {
15                     e.printStackTrace();
16                 }
17                 log.info("---异步线程执行任务结束----,time={}", LocalDateTime.now().toString());
18                 return "hello netty future";
19             }
20         });
21         /*String result = future.get();
22         log.info("----主线程阻塞等待异步线程执行结果:{}",result);*/
23         //设置监听
24         future.addListener(new FutureListener<String>() {
25             @Override
```

```
            public void operationComplete(Future<String>
future) throws Exception {
                log.info("---收到异步线程执行任务结果通知----
执行结果是;{},time={}",future.get(),
LocalDateTime.now().toString());
            }
        });
        log.info("---主线程----");
        TimeUnit.SECONDS.sleep(10);
    }


    @Test
    public void testPromise() throws InterruptedException
{
        EventLoopGroup group = new NioEventLoopGroup();
        Promise promise = new
DefaultPromise(group.next());//promise绑定到eventloop上
        group.submit(()->{
            log.info("---异步线程执行任务开始----,time={}",
LocalDateTime.now().toString());
            try {
                TimeUnit.SECONDS.sleep(3);
                promise.setSuccess("hello netty promise");
                log.info("---异步线程执行任务结束----,time=
{}", LocalDateTime.now().toString());
                return;
            } catch (InterruptedException e) {
                promise.setFailure(e);
            }
        });
        //设置监听回调
        promise.addListener(new FutureListener<String>() {
            @Override
            public void operationComplete(Future<String>
future) throws Exception {
                log.info("----异步任务执行结果:
{}",future.get());
            }
        });
        log.info("---主线程----");
        TimeUnit.SECONDS.sleep(10);
    }
```

```
60
61 }
```

5、粘包，半包

粘包半包案例演示：

0、创建`pojo`

```
1  package com.itheima.netty.pojo;
2
3  import java.io.Serializable;
4
5  /**
6   * @description
7   * @author: ts
8   * @create:2021-04-09 09:41
9   */
10 public class UserInfo implements Serializable {
11
12     private  Integer id;
13     private String name;
14     private Integer age;
15     private String gender;
16     private String address;
17
18     public UserInfo() {
19     }
20
21     public UserInfo(Integer id, String name, Integer age,
   String gender, String address) {
22         this.id = id;
23         this.name = name;
24         this.age = age;
25         this.gender = gender;
26         this.address = address;
27     }
28
29     public Integer getId() {
30         return id;
31     }
```

```
32
33      public void setId(Integer id) {
34          this.id = id;
35      }
36
37      public String getName() {
38          return name;
39      }
40
41      public void setName(String name) {
42          this.name = name;
43      }
44
45      public Integer getAge() {
46          return age;
47      }
48
49      public void setAge(Integer age) {
50          this.age = age;
51      }
52
53      public String getGender() {
54          return gender;
55      }
56
57      public void setGender(String gender) {
58          this.gender = gender;
59      }
60
61      public String getAddress() {
62          return address;
63      }
64
65      public void setAddress(String address) {
66          this.address = address;
67      }
68
69      @Override
70      public String toString() {
71          return "UserInfo{" +
72                  "id=" + id +
73                  ", name='" + name + '\'' +
74                  ", age=" + age +
```

```
                ", gender='" + gender + '\'' +
                ", address='" + address + '\'' +
                '}';
    }
}
```

1、在NettyClient的ClientInboundHandler1中向服务端批量写入数据

```
@Override
public void channelActive(ChannelHandlerContext ctx)
throws Exception {
    log.info("ClientInboundHandler1 channelActive begin
send data");
    //通道准备就绪后开始向服务端发送数据
    /*ByteBuf buf = Unpooled.copiedBuffer("hello server,i
am client".getBytes(StandardCharsets.UTF_8));
        ctx.writeAndFlush(buf);*/
    //批量发送数据
    UserInfo userInfo;
    for (int i=0;i<100;i++) {
      userInfo = new UserInfo(i,"name"+i,i+1,(i%2==0) ?
"男":"女","北京");

 ctx.writeAndFlush(Unpooled.copiedBuffer(userInfo.toString
().getBytes(StandardCharsets.UTF_8)));
    }
}
```

2、NettyServer中创建TcpStickHalfHandler1

```
public class TcpStickHalfHandler1 extends ChannelInboundHandlerAdapter {
    private static final Logger log = LoggerFactory.getLogger(TcpStickHalfHandler1.class);
    int count =0;
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        ByteBuf buf = (ByteBuf) msg;
        count++;
        log.info("---服务端收到的第{}个数据:{}",count,buf.toString(StandardCharsets.UTF_8));
    }
}
```

3、在`NettyServer`的`initChannel`中添加该`handler`

```
.childHandler(new ChannelInitializer<SocketChannel>() {
    //每个客户端channel初始化时都会执行该方法来配置该channel的相关handler
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        //获取与该channel绑定的pipeline
        ChannelPipeline pipeline = ch.pipeline();
        //测试tcp stick or half pack
        pipeline.addLast(new TcpStickHalfHandler1());
    }
});
```

启动测试

6、codec

string编解码

1、服务端添加编解码器

```
@Override
protected void initChannel(SocketChannel ch) throws
Exception {
    //获取与该channel绑定的pipeline
    ChannelPipeline pipeline = ch.pipeline();

    //测试tcp stick or half pack
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new StringEncoder());
    pipeline.addLast(new
LengthFieldBasedFrameDecoder(65536,0,2,0,2));
    pipeline.addLast(new StringDecoder());
    pipeline.addLast(new TcpStickHalfHandler1());
}
```

2、服务端TcpStickHalfHandler1直接接收string类型的数据

```
public class TcpStickHalfHandler1 extends
ChannelInboundHandlerAdapter {
    private static final Logger log =
LoggerFactory.getLogger(TcpStickHalfHandler1.class);
    int count =0;
   /* @Override
    public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
        ByteBuf buf = (ByteBuf) msg;
        count++;
        log.info("---服务端收到的第{}个数据:
{}",count,buf.toString(StandardCharsets.UTF_8));
    }*/
   @Override
    public void channelRead(ChannelHandlerContext ctx,
Object msg) throws Exception {
        String message = (String) msg;
        count++;
        log.info("---服务端收到的第{}个数据:
{}",count,message);
    }
}
```

3、客户端添加string编解码器

```
@Override
protected void initChannel(SocketChannel ch) throws
Exception {
    ChannelPipeline pipeline = ch.pipeline();
    //添加编码器
    //lengthFieldLength is 1,2,3,4,8
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new StringEncoder());
    pipeline.addLast(new
LengthFieldBasedFrameDecoder(65536,0,2,0,2));
    pipeline.addLast(new StringDecoder());
    //添加客户端channel对应的handler
    pipeline.addLast(new ClientInboundHandler1());
    pipeline.addLast(new ClientSimpleInboundHandler2());
}
```

4、客户端直接发送string数据

```
/**
     * 通道准备就绪
     * @param ctx
     * @throws Exception
     */
@Override
public void channelActive(ChannelHandlerContext ctx)
throws Exception {
    log.info("ClientInboundHandler1 channelActive begin
send data");
    //批量发送数据
    UserInfo userInfo;
    for (int i=0;i<100;i++) {
        userInfo = new UserInfo(i,"name"+i,i+1,(i%2==0) ?
"男":"女","北京");
        ctx.writeAndFlush(userInfo.toString());
    }
}
```

5、测试

protobuf编解码

1、下载protobuf编辑器：https://github.com/protocolbuffers/protobuf/releases

2、编写`.proto`文件

```
syntax = "proto3";

option java_package = "com.itheima.netty.pojo";
option java_outer_classname = "MessageProto";
message Message {
  string id = 1;
  string content = 2;
}
```

3、执行命令生成`pojo`

```
protoc.exe --java_out=./ Message.proto
```

4、将生成的`pojo`拷贝到项目中

5、服务端添加对应的编解码器

```
rotected void initChannel(SocketChannel ch) throws Exception {
    //获取与该channel绑定的pipeline
    ChannelPipeline pipeline = ch.pipeline();
    //测试tcp stick or half pack
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new ProtobufEncoder());
    pipeline.addLast(new LengthFieldBasedFrameDecoder(65536,0,2,0,2));
    pipeline.addLast(new ProtobufDecoder(MessageProto.Message.getDefaultInstance()));
    pipeline.addLast(new TcpStickHalfHandler1());
}
```

6、服务端`TcpStickHalfHandler1`接收对应的数据

```
public class TcpStickHalfHandler1 extends ChannelInboundHandlerAdapter {
    private static final Logger log = LoggerFactory.getLogger(TcpStickHalfHandler1.class);
    int count =0;

   @Override
   public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        MessageProto.Message message = (MessageProto.Message) msg;
        count++;
        log.info("---服务端收到的第{}个数据:{}",count,message);
    }
}
```

也可以继承`SimpleChannelInboundHandler`指定泛型，这样就不用自己强转了

7、客户端添加对应的编解码器

```
protected void initChannel(SocketChannel ch) throws Exception {
    ChannelPipeline pipeline = ch.pipeline();
    //添加编码器
    //lengthFieldLength is 1,2,3,4,8
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new ProtobufEncoder());
    pipeline.addLast(new LengthFieldBasedFrameDecoder(65536,0,2,0,2));
    pipeline.addLast(new ProtobufDecoder(MessageProto.Message.getDefaultInstance()));
    //添加客户端channel对应的handler
    pipeline.addLast(new ClientInboundHandler1());
    pipeline.addLast(new ClientSimpleInboundHandler2());
}
```

8、在客户端`ClientInboundHandler1`中发送数据

```
@Override
public void channelActive(ChannelHandlerContext ctx)
throws Exception {
    log.info("ClientInboundHandler1 channelActive begin
send data");
    //批量发送数据
    MessageProto.Message message;
    for (int i=0;i<100;i++) {
        message=
MessageProto.Message.newBuilder().setId("message" +
i).setContent("hello protobuf").build();
        ctx.writeAndFlush(message);
    }
}
```

启动测试

protostuff编解码

1、编写`ProtostuffUtil`工具

```
package com.itheima.netty.util;

import com.sun.org.slf4j.internal.Logger;
import com.sun.org.slf4j.internal.LoggerFactory;
import io.protostuff.LinkedBuffer;
import io.protostuff.ProtostuffIOUtil;
import io.protostuff.Schema;
import io.protostuff.runtime.RuntimeSchema;

import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

/**
 * @description
 * @author: ts
 * @create:2021-04-08 10:31
 */
public class ProtostuffUtil {

```

```
20      private static final Logger log =
LoggerFactory.getLogger(ProtostuffUtil.class);
21
22      //存储因为无法直接序列化/反序列化 而需要被包装的类型Class
23      private static final Set<Class<?>> WRAPPER_SET = new
HashSet<Class<?>>();
24
25      static {
26          WRAPPER_SET.add(List.class);
27          WRAPPER_SET.add(ArrayList.class);
28          WRAPPER_SET.add(CopyOnWriteArrayList.class);
29          WRAPPER_SET.add(LinkedList.class);
30          WRAPPER_SET.add(Stack.class);
31          WRAPPER_SET.add(Vector.class);
32          WRAPPER_SET.add(Map.class);
33          WRAPPER_SET.add(HashMap.class);
34          WRAPPER_SET.add(TreeMap.class);
35          WRAPPER_SET.add(LinkedHashMap.class);
36          WRAPPER_SET.add(Hashtable.class);
37          WRAPPER_SET.add(SortedMap.class);
38          WRAPPER_SET.add(Object.class);
39      }
40
41      //注册需要使用包装类进行序列化的Class对象
42      public static void registerWrapperClass(Class<?>
clazz) {
43          WRAPPER_SET.add(clazz);
44      }
45
46      /**
47       * 将对象序列化为字节数组
48       * @param t
49       * @param useWrapper 为true完全使用包装模式 为false则选
择性的使用包装模式
50       * @param <T>
51       * @return
52       */
53      public static <T> byte[] serialize(T t,boolean
useWrapper) {
54          Object serializerObj = t;
55          if (useWrapper) {
56              serializerObj =
SerializeDeserializeWrapper.build(t);
```

```
57          }
58          return serialize(serializerObj);
59      }
60
61      /**
62       * 将对象序列化为字节数组
63       * @param t
64       * @param <T>
65       * @return
66       */
67      public static <T> byte[] serialize(T t) {
68          //获取序列化对象的class
69          Class<T> clazz = (Class<T>) t.getClass();
70          Object serializerObj = t;
71          if (WRAPPER_SET.contains(clazz)) {
72              serializerObj =
SerializeDeserializeWrapper.build(t);//将原始序列化对象进行
包装
73          }
74          return doSerialize(serializerObj);
75      }
76
77
78      /**
79       * 执行序列化
80       * @param t
81       * @param <T>
82       * @return
83       */
84      public static <T> byte[] doSerialize(T t) {
85          //获取序列化对象的class
86          Class<T> clazz = (Class<T>) t.getClass();
87          //获取Schema
88          // RuntimeSchema<T> schema =
RuntimeSchema.createFrom(clazz);//根据给定的class创建schema
89          /**
90           * this is lazily created and cached by
RuntimeSchema
91           * so its safe to call RuntimeSchema.getSchema()
over and over The getSchema method is also thread-safe
92           */
93          Schema<T> schema =
RuntimeSchema.getSchema(clazz);//内部有缓存机制
```

```
94          /**
95           * Re-use (manage) this buffer to avoid
allocating on every serialization
96           */
97          LinkedBuffer buffer =
LinkedBuffer.allocate(LinkedBuffer.DEFAULT_BUFFER_SIZE);
98          byte[] protostuff = null;
99          try {
100             protostuff = ProtostuffIOUtil.toByteArray(t,
schema, buffer);
101         } catch (Exception e){
102             log.error("protostuff serialize error,
{}",e.getMessage());
103         }finally {
104             buffer.clear();
105         }
106         return protostuff;
107     }
108
109
110     /**
111      * 反序列化
112      * @param data
113      * @param clazz
114      * @param <T>
115      * @return
116      */
117     public static <T> T deserialize(byte[] data,Class<T>
clazz) {
118         //判断是否经过包装
119         if (WRAPPER_SET.contains(clazz)) {
120             SerializeDeserializeWrapper<T> wrapper = new
SerializeDeserializeWrapper<T>();
121
ProtostuffIOUtil.mergeFrom(data,wrapper,RuntimeSchema.get
Schema(SerializeDeserializeWrapper.class));
122             return wrapper.getData();
123         }else {
124             Schema<T> schema =
RuntimeSchema.getSchema(clazz);
125             T newMessage = schema.newMessage();
126
ProtostuffIOUtil.mergeFrom(data,newMessage,schema);
```

```
            return newMessage;
        }
    }


    private static class SerializeDeserializeWrapper<T> {
        //被包装的数据
        T data;

        public static <T> SerializeDeserializeWrapper<T>
build(T data){
            SerializeDeserializeWrapper<T> wrapper = new
SerializeDeserializeWrapper<T>();
            wrapper.setData(data);
            return wrapper;
        }

        public T getData() {
            return data;
        }

        public void setData(T data) {
            this.data = data;
        }
    }
}
```

2、编写编码器和解码器

```
/**
 * @description
 * @author: ts
 * @create:2021-04-30 21:20
 */
public class ProtostuffDecoder  extends
MessageToMessageDecoder<ByteBuf> {
    private static final Logger log =
LoggerFactory.getLogger(ProtostuffDecoder.class);

    @Override
```

```
    protected void decode(ChannelHandlerContext ctx, ByteBuf msg, List<Object> out) throws Exception {
        try {
            int length = msg.readableBytes();
            byte[] bytes = new byte[length];
            msg.readBytes(bytes);
            UserInfo userInfo = ProtostuffUtil.deserialize(bytes, UserInfo.class);
            out.add(userInfo);
        } catch (Exception e) {
            log.error("protostuff decode error,msg={}",e.getMessage());
            throw new RuntimeException(e);
        }
    }
}
```

编码器

```
/**
 * @description
 * @author: ts
 * @create:2021-04-30 21:26
 */
public class ProtostuffEncoder extends MessageToMessageEncoder<UserInfo> {

    private static final Logger log = LoggerFactory.getLogger(ProtostuffEncoder.class);

    @Override
    protected void encode(ChannelHandlerContext ctx, UserInfo msg, List<Object> out) throws Exception {
        try {
            byte[] bytes = ProtostuffUtil.serialize(msg);
            ByteBuf buf = Unpooled.wrappedBuffer(bytes);
            out.add(buf);
        } catch (Exception e) {
            log.error("protostuff encode error,msg={}",e.getMessage());
            throw new RuntimeException(e);
```

```
        }
    }
}
```

3、服务端`TcpStickHalfHandler1`接收数据

```
public class TcpStickHalfHandler1 extends ChannelInboundHandlerAdapter {
    private static final Logger log = LoggerFactory.getLogger(TcpStickHalfHandler1.class);
    int count =0;
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        UserInfo userInfo = (UserInfo) msg;//直接用UserInfo接收
        count++;
        log.info("---服务端收到的第{}个数据:{}",count,userInfo);
    }
}
```

4、客户端`ClientInboundHandler1`直接发送`UserInfo`对象

```
@Override
public void channelActive(ChannelHandlerContext ctx) throws Exception {
    log.info("ClientInboundHandler1 channelActive begin send data");
    //批量发送数据
    UserInfo userInfo;
    for (int i=0;i<100;i++) {
        userInfo = new UserInfo(i,"name"+i,i+1,(i%2==0) ? "男":"女","北京");
        ctx.writeAndFlush(userInfo);
    }
}
```

5、服务端添加对应的编解码器

```
protected void initChannel(SocketChannel ch) throws
Exception {
    //获取与该channel绑定的pipeline
    ChannelPipeline pipeline = ch.pipeline();
    //测试tcp stick or half pack
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new ProtostuffEncoder());

    pipeline.addLast(new
LengthFieldBasedFrameDecoder(65536,0,2,0,2));
    pipeline.addLast(new ProtostuffDecoder());
    pipeline.addLast(new TcpStickHalfHandler1());
}
```

6、客户端添加对应的编解码器

```
protected void initChannel(SocketChannel ch) throws
Exception {
    ChannelPipeline pipeline = ch.pipeline();
    //添加编码器
    //lengthFieldLength is 1,2,3,4,8
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new ProtostuffEncoder());
    pipeline.addLast(new
LengthFieldBasedFrameDecoder(65536,0,2,0,2));
    pipeline.addLast(new ProtostuffDecoder());
    //添加客户端channel对应的handler
    pipeline.addLast(new ClientInboundHandler1());
    pipeline.addLast(new ClientSimpleInboundHandler2());
}
```

启动测试

扩展：对于protobuf相关编解码，netty也提供了对应的1次编解码器，可以替换

```
protected void initChannel(SocketChannel ch) throws
Exception {
    //获取与该channel绑定的pipeline
```

```
    ChannelPipeline pipeline = ch.pipeline();

    //测试tcp stick or half pack
    //pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new ProtobufVarint32LengthFieldPrepender());
    pipeline.addLast(new ProtostuffEncoder());

    //pipeline.addLast(new LengthFieldBasedFrameDecoder(65536,0,2,0,2));
    pipeline.addLast(new ProtobufVarint32FrameDecoder());
    pipeline.addLast(new ProtostuffDecoder());
    pipeline.addLast(new TcpStickHalfHandler1());
}
```

http编解码

基于Netty实现一个HTTP服务器

1、在服务端添加`http`协议相关编解码器

```
protected void initChannel(SocketChannel ch) throws Exception {
    //获取与该channel绑定的pipeline
    ChannelPipeline pipeline = ch.pipeline();

    //测试 http service
    pipeline.addLast(new HttpResponseEncoder());

    pipeline.addLast(new HttpRequestDecoder());
    //文件上传需要设置大点儿 单位是字节
    pipeline.addLast(new HttpObjectAggregator(1024*1024*8));
    pipeline.addLast(new MyHttpServerHandler());
}
```

2、编写`MyHttpServerHandler`

```
package com.itheima.netty.handler;
```

```
2
3  import com.alibaba.fastjson.JSONObject;
4  import io.netty.buffer.ByteBuf;
5  import io.netty.buffer.Unpooled;
6  import io.netty.channel.ChannelFutureListener;
7  import io.netty.channel.ChannelHandlerContext;
8  import io.netty.channel.SimpleChannelInboundHandler;
9  import io.netty.handler.codec.http.*;
10 import io.netty.handler.codec.http.multipart.*;
11 import org.slf4j.Logger;
12 import org.slf4j.LoggerFactory;
13
14 import java.io.File;
15 import java.io.IOException;
16 import java.nio.charset.StandardCharsets;
17 import java.util.List;
18 import java.util.Map;
19
20 /**
21  * @description
22  * @author: ts
23  * @create:2021-04-30 23:43
24  */
25 public class MyHttpServerHandler extends
   SimpleChannelInboundHandler<FullHttpRequest> {
26     private static final Logger log =
   LoggerFactory.getLogger(MyHttpServerHandler.class);
27
28     private static final HttpDataFactory
   HTTP_DATA_FACTORY = new
   DefaultHttpDataFactory(DefaultHttpDataFactory.MAXSIZE);
29
30     static {
31         DiskFileUpload.baseDirectory =
   "/opt/netty/fileupload";
32     }
33
34     @Override
35     protected void channelRead0(ChannelHandlerContext
   ctx, FullHttpRequest fullHttpRequest) throws Exception {
36         //HttpRequest request = fullHttpRequest;
37         //String uri = fullHttpRequest.uri();
38         //获取method
```

```
39          HttpMethod method = fullHttpRequest.method();
40          //根据method解析参数，封装数据，
41          if (HttpMethod.GET.equals(method)) {
42              parseGet(fullHttpRequest);
43          }else if (HttpMethod.POST.equals(method)) {
44              parsePost(fullHttpRequest);
45          }else {
46              log.error("{} method is not supported ,please change http method for get or post!");
47          }
48          //service
49          //response client
50          StringBuilder sb = new StringBuilder();
51          sb.append("<html>");
52          sb.append("<head>");
53          sb.append("</head>");
54          sb.append("<body>");
55          sb.append("<h3>success</h3>");
56          sb.append("</body>");
57          sb.append("</html>");
58
 writeResponse(ctx,fullHttpRequest,HttpResponseStatus.OK,sb.toString());
59      }
60
61      private void writeResponse(ChannelHandlerContext ctx, FullHttpRequest fullHttpRequest, HttpResponseStatus status, String msg) {
62          FullHttpResponse response = new DefaultFullHttpResponse(HttpVersion.HTTP_1_1,status);
63
 response.content().writeBytes(msg.getBytes(StandardCharsets.UTF_8));
64
 response.headers().set(HttpHeaderNames.CONTENT_TYPE,"text/html;charset=utf-8");
65
 HttpUtil.setContentLength(response,response.content().readableBytes());
66          boolean keepAlive = HttpUtil.isKeepAlive(fullHttpRequest);
67          if (keepAlive) {
```

```
68
 response.headers().set(HttpHeaderNames.CONNECTION,"keep-
alive");
69              ctx.writeAndFlush(response);
70          }else {
71
 ctx.writeAndFlush(response).addListener(ChannelFutureLis
tener.CLOSE);
72          }
73      }
74
75      private void parsePost(FullHttpRequest
fullHttpRequest) {
76          //获取content-type
77          String contentType =
getContentType(fullHttpRequest);
78          switch (contentType) {
79              case "application/json":
80                  parseJson(fullHttpRequest.content());
81                  break;
82              case "application/x-www-form-urlencoded":
83                  parseForm(fullHttpRequest);
84                  break;
85              case "multipart/form-data":
86                  parseMultipart(fullHttpRequest);
87                  break;
88              default:
89          }
90
91      }
92
93      private void parseMultipart(FullHttpRequest
fullHttpRequest) {
94          HttpPostRequestDecoder postRequestDecoder = new
HttpPostRequestDecoder(HTTP_DATA_FACTORY,fullHttpRequest)
;
95          //判断是否是multipart
96          if (postRequestDecoder.isMultipart()) {
97          //获取 body中的数据
98          List<InterfaceHttpData> bodyHttpDatas =
postRequestDecoder.getBodyHttpDatas();
99          bodyHttpDatas.forEach(dataItem ->{
100             //获取数据项的类型
```

```
            InterfaceHttpData.HttpDataType dataType =
dataItem.getHttpDataType();
            //判断是普通表达项还是文件上传项
            if
(dataType.equals(InterfaceHttpData.HttpDataType.Attribute
)) {
                //普通表单项 直接获取数据
                Attribute attribute = (Attribute)
dataItem;
                try {
                    log.info("表单项名称:{},表单项值:
{}",attribute.getName(),attribute.getValue());
                } catch (IOException e) {
                    log.error("获取表单项数据错误,msg=
{}",e.getMessage());
                }

            }else if
(dataType.equals(InterfaceHttpData.HttpDataType.FileUploa
d)) {
                //文件上传项 处理待上传的数据
                FileUpload fileUpload = (FileUpload)
dataItem;
                //获取原始文件名称
                String filename =
fileUpload.getFilename();
                //获取表单name属性
                String name = fileUpload.getName();
                log.info("文件名称:{},表单项名称:
{}",filename,name);
                //将文件数据保存到磁盘
                if (fileUpload.isCompleted()) {
                    try {
                        String path =
DiskFileUpload.baseDirectory + File.separator + filename;
                        //File file =
fileUpload.getFile();
                        fileUpload.renameTo(new
File(path));
                    } catch (IOException e) {
                        log.error("文件转存失败,msg=
{}",e.getMessage());
                    }
```

```
            }
        }else {

            }
        });
        }
    }

    private void parseForm(FullHttpRequest
fullHttpRequest) {
        //post请求时uri中也可能携带参数
        parseKVstr(fullHttpRequest.uri(),true);
        //解析请求体中的表单数据
        parseFormData(fullHttpRequest.content());
    }

    private void parseFormData(ByteBuf body) {
        String bodystr =
body.toString(StandardCharsets.UTF_8);
        parseKVstr(bodystr,false);
    }

    private void parseJson(ByteBuf jsonbody) {
        String jsonstr =
jsonbody.toString(StandardCharsets.UTF_8);
        //使用json工具反序列化
        JSONObject jsonObject =
JSONObject.parseObject(jsonstr);
        //打印 json数据
        jsonObject.entrySet().stream().forEach(entry ->{
            log.info("json key={},json value=
{}",entry.getKey(),entry.getValue());
        });
    }

    private String getContentType(FullHttpRequest
request) {
        HttpHeaders headers = request.headers();
        String contentType =
headers.get(HttpHeaderNames.CONTENT_TYPE);//
text/plain;charset=UTF-8
```

```
        //List<String> acceptEncoding = headers.getAll(HttpHeaderNames.ACCEPT_ENCODING);//accept-encoding:gzip, deflate, br
        return contentType.split(";")[0];
    }

    private void parseGet(FullHttpRequest request) {
        //通过uri解析请求参数
        parseKVstr(request.uri(),true);
    }

    private void parseKVstr(String str,boolean hasPath) {
        //通过QueryStringDecoder解析kv字符串
        QueryStringDecoder qsd = new QueryStringDecoder(str, StandardCharsets.UTF_8,hasPath);////get请求的uri是: path?k=v
        Map<String, List<String>> parameters = qsd.parameters();
        //封装参数，执行业务 此处打印即可
        if (parameters!=null && parameters.size() > 0) {
            parameters.entrySet().stream().forEach(entry->{
                log.info("参数名:{},参数值:{}",entry.getKey(),entry.getValue());
            });
        }
    }


    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
        //ctx.flush();
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception {
        log.error("MyHttpServerHandler Exception,{}",cause.getMessage());
    }
}
```

```
194
```

3、编写测试页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>测试 netty http server</title>
</head>
<body>
    <form action="http://localhost:8888/test"
method="post" enctype="multipart/form-data">
        <label for="username">用户名:</label><input
id="username" type="text" name="username"><br/>
        <label for="password">密码:</label><input
id="password" type="password" name="password"><br/>
        <label for="email">邮箱:</label><input id="email"
type="email" name="email"><br/>
        <label for="address">地址:</label><input
id="address" type="text" name="address"><br/>
        <label for="pic">选择文件:</label><input id="pic"
type="file" name="pic"><br/>

        <input type="submit" value="提交">
    </form>
</body>
</html>
```

启动测试

websocke通信

1、服务端添加对应的编解码器

```
.childHandler(new ChannelInitializer<SocketChannel>() {
    //每个客户端channel初始化时都会执行该方法来配置该channel的相
关handler
```

```
    @Override
    protected void initChannel(SocketChannel ch) throws Exception {
        //获取与该channel绑定的pipeline
        ChannelPipeline pipeline = ch.pipeline();
        //基于netty开发websocket 服务端
        pipeline.addLast(new HttpServerCodec());//HttpServerCodec = HttpRequestDecoder + HttpResponseEncoder
        pipeline.addLast(new HttpObjectAggregator(1024*1024*8));
        //pipeline.addLast(new ChunkedWriteHandler());
        //添加业务处理器
        pipeline.addLast(new MyWebSocketServerHandler());

    }
});
```

2、服务端编写`MyWebSocketServerHandler`

```
package com.itheima.netty.handler;

import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.SimpleChannelInboundHandler;
import io.netty.handler.codec.http.*;
import io.netty.handler.codec.http.websocketx.*;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.nio.charset.StandardCharsets;
import java.util.Date;
import java.util.concurrent.TimeUnit;

/**
 * @description
 * @author: ts
 * @create:2021-04-12 20:44
```

```
 */
public class MyWebSocketServerHandler extends SimpleChannelInboundHandler<Object> {

    private static final Logger log = LoggerFactory.getLogger(MyWebSocketServerHandler.class);

    private WebSocketServerHandshaker serverHandshaker;



    @Override
    protected void channelRead0(ChannelHandlerContext ctx, Object msg) throws Exception {
        //判断是http握手请求还是websocket请求
        if (msg instanceof FullHttpRequest) {
            boolean handShaker = handleHttpRequest(ctx, (FullHttpRequest) msg);
            if (handShaker) {
                //握手成功后 服务端主动推送消息，每隔5s推送一次
                new Thread(()->{
                    while (true) {
                        try {

ctx.channel().writeAndFlush(new TextWebSocketFrame("你好,这是服务器主动推送回来的数据,当前时间为:"+new Date().toString()));
                            TimeUnit.SECONDS.sleep(5);
                        } catch (InterruptedException e) {
                            log.error("push msg exception,{}",e.getMessage());
                        }
                    }
                }).start();
            }
        }else if (msg instanceof WebSocketFrame) {
            handleWebSocketFrame(ctx, (WebSocketFrame)msg);
        }
    }

```

```
    /**
     * 接收到的消息是已经解码的WebSocketFrame消息
     * @param ctx
     * @param frame
     */
    private void handleWebSocketFrame(ChannelHandlerContext ctx, WebSocketFrame frame) {
        // 判断链路消息类型
        if (frame instanceof CloseWebSocketFrame) { // 关闭链路指令
            serverHandshaker.close(ctx.channel(), ((CloseWebSocketFrame) frame).retain());
            return;
        }
        if (frame instanceof PingWebSocketFrame) { // 维持链路的ping 指令
            ctx.channel().writeAndFlush(new PongWebSocketFrame(frame.content().retain()));
            return;
        }
        if (frame instanceof TextWebSocketFrame) { // 普通文本消息
            TextWebSocketFrame textFrame = (TextWebSocketFrame) frame;
            String message = textFrame.text();
            log.info("receive text msg is {}",message);
            //构造返回
            ctx.channel().writeAndFlush(new TextWebSocketFrame("你好,欢迎使用netty websocket 服务,当前时间为:"+new Date().toString()));
            return;
        }
        if (frame instanceof BinaryWebSocketFrame) { //二进制消息
            log.info("frame is binarywebsocketframe");
        }

    }

    private boolean handleHttpRequest(ChannelHandlerContext ctx, FullHttpRequest fullHttpRequest) {
```

```
        //先判断是否解码成功，
        if (!fullHttpRequest.decoderResult().isSuccess()) {
            sendHttpResponse(ctx,fullHttpRequest,new DefaultFullHttpResponse(HttpVersion.HTTP_1_1, HttpResponseStatus.BAD_REQUEST));
            return false;
        }
        // 然后判断是否要建立websocket连接
        //构造握手工厂 创建握手处理类，并且构造握手响应给客户端
        WebSocketServerHandshakerFactory serverHandshakerFactory = new WebSocketServerHandshakerFactory("ws://localhost:8888/mywebsocket",null,false);
        if (!fullHttpRequest.headers().contains(HttpHeaderNames.UPGRADE,"websocket",true)) {
            WebSocketServerHandshakerFactory.sendUnsupportedVersionResponse(ctx.channel());
            return false;
        }
        serverHandshaker = serverHandshakerFactory.newHandshaker(fullHttpRequest);
        serverHandshaker.handshake(ctx.channel(),fullHttpRequest);
        return true;
    }

    private void sendHttpResponse(ChannelHandlerContext ctx, FullHttpRequest request, FullHttpResponse response) {
        if (!response.status().equals(HttpResponseStatus.OK)) {
            ByteBuf byteBuf = Unpooled.wrappedBuffer("error".getBytes(StandardCharsets.UTF_8));
            response.content().writeBytes(byteBuf);
            byteBuf.release();
        HttpUtil.setContentLength(response,response.content().readableBytes());
```

```
        }
        ChannelFuture future =
ctx.channel().writeAndFlush(response);
        if (!HttpUtil.isKeepAlive(request) ||
!response.status().equals(HttpResponseStatus.OK) ) {

future.addListener(ChannelFutureListener.CLOSE);
        }

    }


    @Override
    public void channelReadComplete(ChannelHandlerContext
ctx) throws Exception {
        ctx.flush();
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext
ctx, Throwable cause) throws Exception {
        log.error("server error,msg is
{}",cause.getMessage());
        ctx.close();
    }
}

```

3、编写html页面

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>test netty websocket </title>
</head>
<body>
<br>
<script type="text/javascript">
    var socket;
    if(!window.WebSocket){
```

```
        window.WebSocket=window.MozWebSocket;
    }
    if(window.WebSocket){
        socket=new
WebSocket("ws://localhost:8888/mywebSocket");
        socket.onmessage=function(event){
            var
ta=document.getElementById('responseText');
            ta.value="";
            ta.value=event.data;
        };
        socket.onopen=function(event){
            var
ta=document.getElementById('responseText');
            ta.value='打开WebSocket服务器正常，浏览器支持
WebSocket! ';
        };
        socket.onclose=function(event){
            var
ta=document.getElementById('responseText');
            ta.value='';
            ta.value="WebSocket 关闭! ";
        };
    }else{
        alert("抱歉，您的浏览器不支持WebSocket协议! ");
    }
    function send(message){
        if(!window.WebSocket){
            return;
        }
        if(socket!=null){
            socket.send(message);
        }else{
            alert("WebSocket连接没有建立成功，请刷新页面! ");
        }
        /* if(socket.readyState==WebSocket.open){
            socket.send(message);
        }else{
            alert("WebSocket连接没有建立成功! ");
        } */
    }
</script>
<form onsubmit="return false;">
```

```
    <input type="text" name="message" value="Netty WebSocket实战"/>
    <br><br>
    <input type="button" value="发送WebSocket请求消息" onclick="send(this.form.message.value)"/>
    <hr color="blue"/>
    <h3>服务端返回的应答消息</h3>
    <textarea id="responseText" style="width:500px;height:300px;"></textarea>
</form>
</body>
</html>
```

启动测试!

7、keepavlie + idle

1、将`NettyClient`和`NettyServer`拷贝一份得到`NettyClientV2`和`NettyServerV2`

2、初始化服务端和客户端的`pipeline`

```
//Server端
protected void initChannel(SocketChannel ch) throws Exception {
    //获取与该channel绑定的pipeline
    ChannelPipeline pipeline = ch.pipeline();
    pipeline.addLast(new LoggingHandler(LogLevel.INFO));
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new StringEncoder());
    pipeline.addLast(new LengthFieldBasedFrameDecoder(1024,0,2,0,2));
    pipeline.addLast(new StringDecoder());
}
```

```
//Client端
protected void initChannel(SocketChannel ch) throws
Exception {
    ChannelPipeline pipeline = ch.pipeline();
    pipeline.addLast(new LoggingHandler(LogLevel.INFO));
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new StringEncoder());
    pipeline.addLast(new
LengthFieldBasedFrameDecoder(1024,0,2,0,2));
    pipeline.addLast(new StringDecoder());
}
```

3、服务端编写用于idle监测的handler

```
@Slf4j
public class ServerReadIdleCheckHandler extends
IdleStateHandler {

    public ServerReadIdleCheckHandler() {
        super(10, 0, 0, TimeUnit.SECONDS);
    }

    @Override
    protected void channelIdle(ChannelHandlerContext ctx,
IdleStateEvent evt) throws Exception {
        log.info("server channel idle----");
        if (evt ==
IdleStateEvent.FIRST_READER_IDLE_STATE_EVENT) {
            ctx.close();
            log.info("server read idle , close
channel.....");
            return;
        }
        super.channelIdle(ctx, evt);
    }
}
```

4、服务端添加`ServerIdleCheckHandler`到`pipeline`中

```
protected void initChannel(SocketChannel ch) throws Exception {
    //获取与该channel绑定的pipeline
    ChannelPipeline pipeline = ch.pipeline();
    pipeline.addLast(new LoggingHandler(LogLevel.INFO));
    pipeline.addLast(new ServerReadIdleCheckHandler());//添加ServerReadIdleCheckHandler
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new StringEncoder());
    pipeline.addLast(new LengthFieldBasedFrameDecoder(1024,0,2,0,2));
    pipeline.addLast(new StringDecoder());
}
```

先启动测试，查看idle监测的效果

5、客户端完成，5s的write监测，超过5s不发送数据，就发送一个keepalive消息，避免被服务端断掉连接，故，编写客户端的idlehandler

```
@Slf4j
public class ClientWriteCheckIdleHandler extends IdleStateHandler {
    public ClientWriteCheckIdleHandler() {
        super(0, 5, 0, TimeUnit.SECONDS);
    }

    //也可在channelIdle方法中直接处理
}
```

```
@Slf4j
@ChannelHandler.Sharable
public class KeepaliveHandler extends ChannelDuplexHandler {

    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) throws Exception {
        if (evt == IdleStateEvent.FIRST_WRITER_IDLE_STATE_EVENT) {
```

```
            log.info("client write idle,so send keepalive msg to server");
            ctx.writeAndFlush("this is keepalive msg");
        }

        super.userEventTriggered(ctx, evt);
    }
}
```

6、客户端向pipeline中添加handler

```
protected void initChannel(SocketChannel ch) throws Exception {
    ChannelPipeline pipeline = ch.pipeline();
    pipeline.addLast(new LoggingHandler(LogLevel.INFO));
    pipeline.addLast(new ClientWriteCheckIdleHandler());//write idle监测
    pipeline.addLast(new LengthFieldPrepender(2));
    pipeline.addLast(new StringEncoder());

    pipeline.addLast(new KeepaliveHandler());//发送keepalvie消息
    pipeline.addLast(new LengthFieldBasedFrameDecoder(1024,0,2,0,2));
    pipeline.addLast(new StringDecoder());
}
```

启动测试!