链滴

后端 | 基于对象传输的 Netty 初体验

作者：z875479694h

一.前言

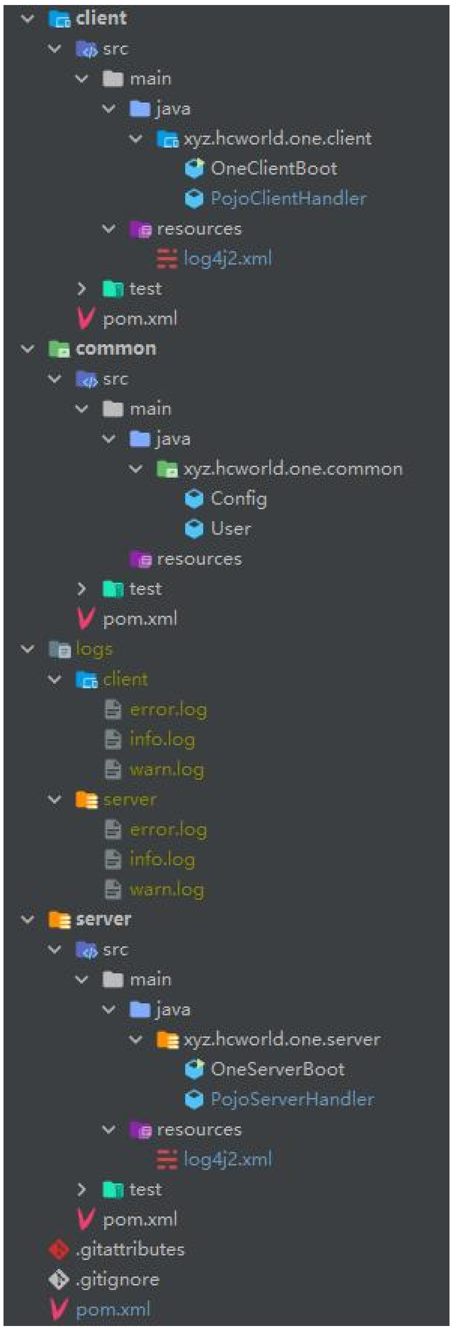
这是一个基于Netty进行对象传输的极简教程，笔者使用到的技术及版本如下：

Netty-All 4.1.70.Final

Log4J 2.14.1

笔者从知乎中的netty的教程学完后做的笔记，好记性不如烂笔头，记起来偶尔翻一番，总会有新收
。

二.项目目录结构


```
client
  src
    main
      java
        xyz.hcworld.one.client
          OneClientBoot
          PojoClientHandler
      resources
        log4j2.xml
    test
  pom.xml
common
  src
    main
      java
        xyz.hcworld.one.common
          Config
          User
      resources
    test
  pom.xml
logs
  client
    error.log
    info.log
    warn.log
  server
    error.log
    info.log
    warn.log
server
  src
    main
      java
        xyz.hcworld.one.server
          OneServerBoot
          PojoServerHandler
      resources
        log4j2.xml
    test
  pom.xml
.gitattributes
.gitignore
pom.xml
```

1. common：服务端与客户端共同使用的配置文件及User对象
2. server：服务端负责接收客户端传来的Object数据并响应
3. client：客户端负责主动向服务器发送Objcet数据并接收服务端反馈的数据

三.pom文件

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>xyz.hcworld</groupId>
    <artifactId>netty-one</artifactId>
    <packaging>pom</packaging>
    <version>0.0.1</version>
    <modules>
        <module>server</module>
        <module>client</module>
        <module>common</module>
    </modules>

    <properties>
        <maven.compiler.source>8</maven.compiler.source>
        <maven.compiler.target>8</maven.compiler.target>
        <lombok.version>1.18.20</lombok.version>
        <netty.version>4.1.70.Final</netty.version>
        <log4j.version>2.14.1</log4j.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>${lombok.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>io.netty</groupId>
            <artifactId>netty-all</artifactId>
            <version>${netty.version}</version>
        </dependency>
        <dependency>
            <groupId>org.apache.logging.log4j</groupId>
            <artifactId>log4j-core</artifactId>
            <version>${log4j.version}</version>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.1</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                    <encoding>UTF-8</encoding>
                </configuration>
            </plugin>
        </plugins>
    </build>

</project>
```

四.log4J配置文件

1.Server端log4J配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL
-->
<!--Configuration后面的status，这个用于设置log4j2自身内部的信息输出，可以不设置，当设置成t
ace时，你会看到log4j2内部各种详细输出-->
<!--monitorInterval：Log4j能够自动检测修改配置 文件和重新配置本身，设置间隔秒数-->
<configuration status="WARN" monitorInterval="30">
  <!--先定义所有的appender-->
  <appenders>
    <!--这个输出控制台的配置-->
    <console name="Console" target="SYSTEM_OUT">
      <!--输出日志的格式-->
      <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>
    </console>
    <!-- 这个会打印出所有的info及以下级别的信息，每次大小超过size，则这size大小的日志会自
存入按年份-月份建立的文件夹下面并进行压缩，作为存档-->
    <RollingFile name="RollingFileInfo" fileName="./logs/server/info.log"
           filePattern="./logs/server/$${date:yyyy-MM}/info-%d{yyyy-MM-dd}-%i.log">
      <!--控制台只输出level及以上级别的信息（onMatch），其他的直接拒绝（onMismatch）-
>
      <ThresholdFilter level="info" onMatch="ACCEPT" onMismatch="DENY"/>
      <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>
      <Policies>
        <TimeBasedTriggeringPolicy/>
        <SizeBasedTriggeringPolicy size="100 MB"/>
      </Policies>
    </RollingFile>
    <RollingFile name="RollingFileWarn" fileName="./logs/server/warn.log"
           filePattern="./logs/server/$${date:yyyy-MM}/warn-%d{yyyy-MM-dd}-%i.log">
      <ThresholdFilter level="warn" onMatch="ACCEPT" onMismatch="DENY"/>
      <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>
      <Policies>
        <TimeBasedTriggeringPolicy/>
        <SizeBasedTriggeringPolicy size="100 MB"/>
      </Policies>
      <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7个文件，这里设
了20 -->
      <DefaultRolloverStrategy max="20"/>
    </RollingFile>
    <RollingFile name="RollingFileError" fileName="./logs/server/error.log"
           filePattern="./logs/server/$${date:yyyy-MM}/error-%d{yyyy-MM-dd}-%i.log">
      <ThresholdFilter level="error" onMatch="ACCEPT" onMismatch="DENY"/>
      <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>
      <Policies>
        <TimeBasedTriggeringPolicy/>
        <SizeBasedTriggeringPolicy size="100 MB"/>
      </Policies>
    </RollingFile>
  </appenders>
  <!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
  <loggers>
    <!--过滤掉spring的一些无用的DEBUG信息-->
```

```
        <logger name="org.springframework" level="INFO"></logger>
        <root level="all">
            <appender-ref ref="Console"/>
            <appender-ref ref="RollingFileInfo"/>
            <appender-ref ref="RollingFileWarn"/>
            <appender-ref ref="RollingFileError"/>
        </root>
    </loggers>
</configuration>
```

2.Client端log4J配置文件

```
<?xml version="1.0" encoding="UTF-8"?>
<!--日志级别以及优先级排序: OFF > FATAL > ERROR > WARN > INFO > DEBUG > TRACE > ALL
-->
<!--Configuration后面的status，这个用于设置log4j2自身内部的信息输出，可以不设置，当设置成t
ace时，你会看到log4j2内部各种详细输出-->
<!--monitorInterval：Log4j能够自动检测修改配置 文件和重新配置本身，设置间隔秒数-->
<configuration status="WARN" monitorInterval="30">
    <!--先定义所有的appender-->
    <appenders>
        <!--这个输出控制台的配置-->
        <console name="Console" target="SYSTEM_OUT">
            <!--输出日志的格式-->
            <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>
        </console>
        <!-- 这个会打印出所有的info及以下级别的信息，每次大小超过size，则这size大小的日志会自
存入按年份-月份建立的文件夹下面并进行压缩，作为存档-->
        <RollingFile name="RollingFileInfo" fileName="./logs/client/info.log"
                     filePattern="./logs/client/$${date:yyyy-MM}/info-%d{yyyy-MM-dd}-%i.log">
            <!--控制台只输出level及以上级别的信息（onMatch），其他的直接拒绝（onMismatch）-
>
            <ThresholdFilter level="info" onMatch="ACCEPT" onMismatch="DENY"/>
            <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>
            <Policies>
                <TimeBasedTriggeringPolicy/>
                <SizeBasedTriggeringPolicy size="100 MB"/>
            </Policies>
        </RollingFile>
        <RollingFile name="RollingFileWarn" fileName="./logs/client/warn.log"
                     filePattern="./logs/client/$${date:yyyy-MM}/warn-%d{yyyy-MM-dd}-%i.log">
            <ThresholdFilter level="warn" onMatch="ACCEPT" onMismatch="DENY"/>
            <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>
            <Policies>
                <TimeBasedTriggeringPolicy/>
                <SizeBasedTriggeringPolicy size="100 MB"/>
            </Policies>
            <!-- DefaultRolloverStrategy属性如不设置，则默认为最多同一文件夹下7个文件，这里设
了20 -->
            <DefaultRolloverStrategy max="20"/>
        </RollingFile>
        <RollingFile name="RollingFileError" fileName="./logs/client/error.log"
                     filePattern="./logs/client/$${date:yyyy-MM}/error-%d{yyyy-MM-dd}-%i.log">
            <ThresholdFilter level="error" onMatch="ACCEPT" onMismatch="DENY"/>
            <PatternLayout pattern="[%d{HH:mm:ss:SSS}] [%p] - %l - %m%n"/>
            <Policies>
                <TimeBasedTriggeringPolicy/>
                <SizeBasedTriggeringPolicy size="100 MB"/>
            </Policies>
        </RollingFile>
    </appenders>
    <!--然后定义logger，只有定义了logger并引入的appender，appender才会生效-->
    <loggers>
        <!--过滤掉spring的一些无用的DEBUG信息-->
        <logger name="org.springframework" level="INFO"></logger>
        <root level="all">
            <appender-ref ref="Console"/>
            <appender-ref ref="RollingFileInfo"/>
            <appender-ref ref="RollingFileWarn"/>
            <appender-ref ref="RollingFileError"/>
        </root>
    </loggers>
</configuration>
```

注: log4j配置文件来源于互联网，有兴趣自行了解，此处不多做篇幅描述

五.服务端源码

1.PojoServerHandler 对象消息处理器

```
package xyz.hcworld.one.server;

import io.netty.buffer.ByteBuf;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.CharsetUtil;
import io.netty.util.ReferenceCountUtil;
import lombok.extern.log4j.Log4j2;
import xyz.hcworld.one.common.User;
import io.netty.channel.ChannelHandler.Sharable;
import java.nio.charset.StandardCharsets;
import java.util.ArrayList;
import java.util.List;

import static io.netty.channel.ChannelFutureListener.FIRE_EXCEPTION_ON_FAILURE;

/**
 * 使用对象进行数据传输(服务器)的消息处理器
 * @ClassName: PojoServerHandler
 * @Author: 张冠诚
 * @Date: 2021/9/3 11:13
 * @Version： 1.0
 */
@Sharable
@Log4j2
public class PojoServerHandler extends ChannelInboundHandlerAdapter {
    /**
     * 解码成功的消息处理
     * @param ctx 上下文
     * @param msg 解析后的数据
     */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        if (msg instanceof String) {
            log.info("server收到String对象:{}", msg);
        } else {
            User user = (User) msg;
            log.info("server收到User对象:{}", user.toString());
        }
        ctx.write("收到了！ ");
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    /**
     * 无论是否有完整的消息被解码成功，只要读到消息，都会触发channelReadComplete
     * @param ctx 上下文
     */
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.flush();
    }

    /**
     * 异常处理
     * @param ctx 上下文
     * @param cause 异常消息
     */
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        log.error("出现异常", cause);
        ctx.close();
    }
}
```

2.OneServerBoot 服务端启动器

```
package xyz.hcworld.one.server;

import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioServerSocketChannel;
import io.netty.handler.codec.serialization.ClassResolvers;
import io.netty.handler.codec.serialization.ObjectDecoder;
import io.netty.handler.codec.serialization.ObjectEncoder;
import xyz.hcworld.one.common.Config;

/**
 * 服务器端启动类
 * @ClassName: OneServerBoot
 * @Author: 张冠诚
 * @Date: 2021/8/31 9:38
 * @Version：  1.0
 */
public class OneServerBoot {

    public static void main(String[] args) {
        //建立两个EventloopGroup用来处理连接和消息
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            ServerBootstrap b = new ServerBootstrap();
            b.group(bossGroup, workerGroup)
                    .channel(NioServerSocketChannel.class)
                    .localAddress(Config.PORT)
                    // tcp/ip协议listen函数中的backlog参数（队列）,等待连接池的大小
                    .option(ChannelOption.SO_BACKLOG, 100)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        public void initChannel(SocketChannel ch) throws Exception {
                            ChannelPipeline p = ch.pipeline();
                            p.addLast(
                                    // 添加Object的序列化及反序列化encoder和decoder（用的java自动的（
）序列化，性能较差）
                                    new ObjectEncoder(),
                                    new ObjectDecoder(ClassResolvers.cacheDisabled(null)),
                                    // 自定义消息处理器
                                    new PojoServerHandler());
                        }
                    })
                    .option(ChannelOption.SO_BACKLOG, 128)
                    .childOption(ChannelOption.SO_KEEPALIVE, true);
            // 绑定端口并开始接收连接
            ChannelFuture f = b.bind().sync();
            // 等待所有的socket都关闭。
            f.channel().closeFuture().sync();
        }catch (Exception e){
            System.out.println(e.getMessage());
        }
    }
}
```

六.客户端源码

1.PojoClientHandler 消息处理器

```
package xyz.hcworld.one.client;
```

```
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelHandler.Sharable;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.handler.timeout.IdleState;
import io.netty.handler.timeout.IdleStateEvent;
import lombok.extern.log4j.Log4j2;
import xyz.hcworld.one.common.Config;
import xyz.hcworld.one.common.User;

import java.util.concurrent.TimeUnit;

import static io.netty.channel.ChannelFutureListener.FIRE_EXCEPTION_ON_FAILURE;

/**
 * 使用对象进行数据传输（客户端）
 *
 * @ClassName: PojoClientHandler
 * @Author: 张冠诚
 * @Date: 2021/9/3 11:17
 * @Version：  1.0
 */
@Sharable
@Log4j2
public class PojoClientHandler extends ChannelInboundHandlerAdapter {
    int i = 0;

    long startTime = -1;


    @Override
    public void channelActive(ChannelHandlerContext ctx) {
        if (startTime < 0) {
            startTime = System.currentTimeMillis();
        }
        println("连接到: " + ctx.channel().remoteAddress());
        // 在channel active的时候发送消息
        ChannelFuture future = ctx.writeAndFlush("对象传输");
        // 将ChannelFuture中的Throwable转发到ChannelPipeline中。
        future.addListener(FIRE_EXCEPTION_ON_FAILURE);
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        // 将消息写回channel
        log.info("客户端收到对象:{}", msg);
        User user = new User();
        user.setUsername("测试账号：" + i);
        user.setPassword("测试密码：" + i);
        i++;
        if (i<=10){
            ctx.writeAndFlush(user);
        }
    }


    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
        // 异常处理
        log.error("出现异常", cause);
        ctx.close();
    }

    @Override
    public void channelInactive(final ChannelHandlerContext ctx) {
        println("连接断开:" + ctx.channel().remoteAddress());
    }

    @Override
    public void userEventTriggered(ChannelHandlerContext ctx, Object evt) {
        if (!(evt instanceof IdleStateEvent)) {
            return;
        }
        IdleStateEvent e = (IdleStateEvent) evt;
        if (e.state() == IdleState.READER_IDLE) {
            // 在Idle状态
            println("Idle状态，关闭连接");
            ctx.close();
        }
    }

    @Override
    public void channelUnregistered(final ChannelHandlerContext ctx) throws Exception {
        println("sleep:" + OneClientBoot.RECONNECT_DELAY + 's');
        i=0;
        ctx.channel().eventLoop().schedule(() -> {
            println("重连接: " + Config.HOST + ':' + Config.PORT);
            OneClientBoot.connect();
        }, OneClientBoot.RECONNECT_DELAY, TimeUnit.SECONDS);

    }

    void println(String msg) {
        if (startTime < 0) {
            log.error("服务下线:{}", msg);

        } else {
            log.error("服务运行时间:{},{}", (System.currentTimeMillis() - startTime) / 1000, msg);
        }
    }
}
```

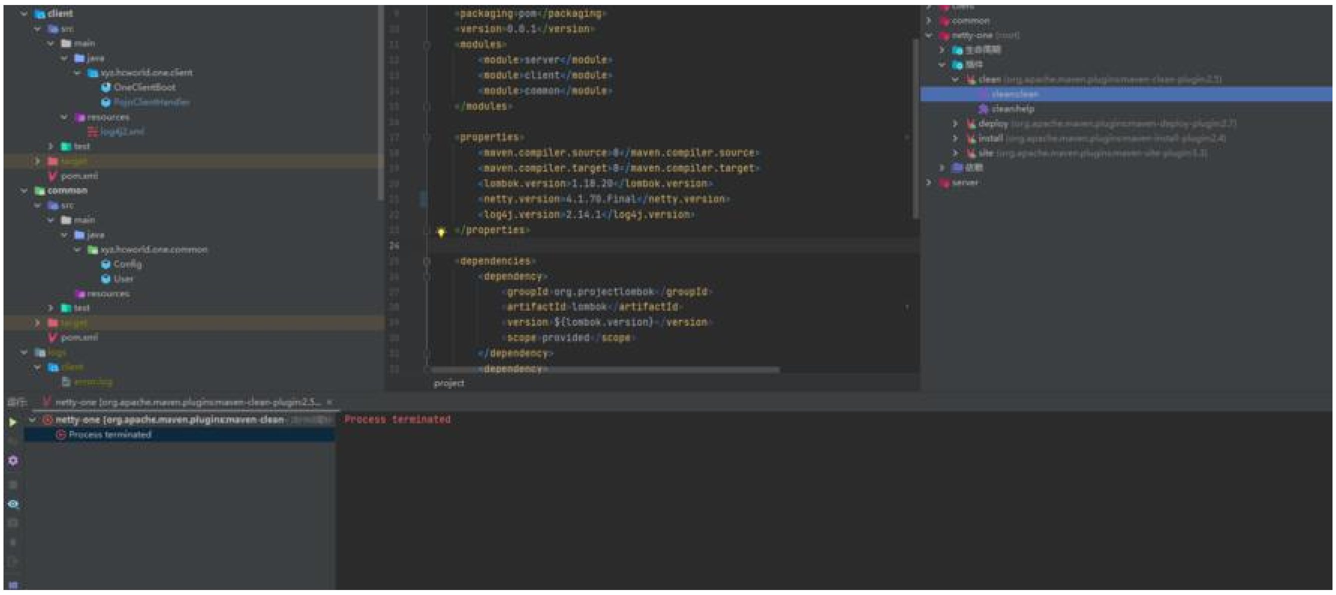
2.OneClientBoot 客户端启动器

```
package xyz.hcworld.one.client;

import io.netty.bootstrap.Bootstrap;
import io.netty.channel.ChannelFuture;
import io.netty.channel.ChannelInitializer;
import io.netty.channel.ChannelPipeline;
import io.netty.channel.EventLoopGroup;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.SocketChannel;
import io.netty.channel.socket.nio.NioSocketChannel;
import io.netty.handler.codec.serialization.ClassResolvers;
import io.netty.handler.codec.serialization.ObjectDecoder;
import io.netty.handler.codec.serialization.ObjectEncoder;
import io.netty.handler.timeout.IdleStateHandler;
import lombok.extern.log4j.Log4j2;
import xyz.hcworld.one.common.Config;

import java.util.concurrent.*;

/**
 * @ClassName: OneClientBoot
 * @Author: 张红尘
 * @Date: 2021/8/31 9:47
 * @Version：  1.0
 */
@Log4j2
public class OneClientBoot extends Thread {


    /**
     * 核心线程池大小
     */
    public static final int CORE_POOL_SIZE = 5;
    /**
     * 最大线程池大小
     */
    public static final int MAX_POOL_SIZE = 10;
    /**
     * 阻塞任务队列大小
     */
    public static final int QUEUE_CAPACITY = 100;
    /**
     * 空闲线程存活时间
     */
    public static final Long KEEP_ALIVE_TIME = 1L;

    /**
     * 在reconnect之前 Sleep 5 秒钟
     */
    static final int RECONNECT_DELAY = Integer.parseInt(System.getProperty("reconnectDelay",
"5"));
    /**
     * 如果在10秒中之内没有任何相应则重连
     */
    private static final int READ_TIMEOUT = Integer.parseInt(System.getProperty("readTimeout"
 "10"));
```

```
private static final PojoClientHandler POJO_CLIENT_HANDLER = new PojoClientHandler();
private static final Bootstrap BS = new Bootstrap();


@Override
public void run() {
    EventLoopGroup group = new NioEventLoopGroup();
    try {

        BS.group(group)
                .channel(NioSocketChannel.class)
                .remoteAddress(Config.HOST, Config.PORT)
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws Exception {
                        ChannelPipeline p = ch.pipeline();
                        p.addLast(
                                // 添加encoder和decoder
                                new ObjectEncoder(),
                                new ObjectDecoder(ClassResolvers.cacheDisabled(null)),
                                //心跳机制
                                new IdleStateHandler(READ_TIMEOUT, 0, 0),
                                POJO_CLIENT_HANDLER);
                    }
                });

        // 连接服务器
        ChannelFuture f = BS.connect().sync();
    } catch (Exception e) {
        log.error("服务下线:{}", e.getMessage());
    }
}

static void connect() {

    BS.connect().addListener(future -> {
        if (future.cause() != null) {
            POJO_CLIENT_HANDLER.startTime = -1;
            POJO_CLIENT_HANDLER.println("建立连接失败: " + future.cause());
        }
    });
}

public static void main(String[] args) {

    //通过ThreadPoolExecutor构造函数自定义参数创建
    ThreadPoolExecutor executor = new ThreadPoolExecutor(
            CORE_POOL_SIZE,
            MAX_POOL_SIZE,
            KEEP_ALIVE_TIME,
            TimeUnit.SECONDS,
            new ArrayBlockingQueue<>(QUEUE_CAPACITY),
            new ThreadPoolExecutor.CallerRunsPolicy());
        // 多线程测试
        for (int i = 0; i < 1; i++) {
            executor.execute(new OneClientBoot());
            try {
                Thread.sleep(400);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }
}
```

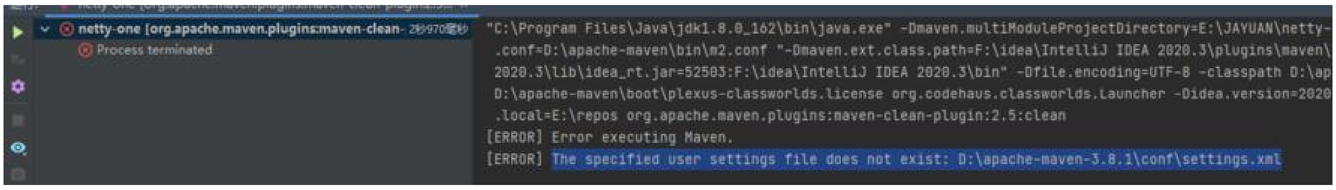
疑难杂症

问题一 使用clean清除编译文件时报Process terminated



```
<packaging>pom</packaging>
<version>0.0.1</version>
<modules>
    <module>server</module>
    <module>client</module>
    <module>common</module>
</modules>

<properties>
    <maven.compiler.source>8</maven.compiler.source>
    <maven.compiler.target>8</maven.compiler.target>
    <lombok.version>1.18.20</lombok.version>
    <netty.version>4.1.70.Final</netty.version>
    <log4j.version>2.14.1</log4j.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombok.version}</version>
        <scope>provided</scope>
    </dependency>
    <dependency>
```

```
Process terminated
```



```
"C:\Program Files\Java\jdk1.8.0_162\bin\java.exe" -Dmaven.multiModuleProjectDirectory=E:\JAYUAN\netty-
 .conf=D:\apache-maven\bin\m2.conf "-Dmaven.ext.class.path=F:\idea\IntelliJ IDEA 2020.3\plugins\maven\
 2020.3\lib\idea_rt.jar=52503:F:\idea\IntelliJ IDEA 2020.3\bin" -Dfile.encoding=UTF-8 -classpath D:\ap
 D:\apache-maven\boot\plexus-classworlds.license org.codehaus.classworlds.Launcher -Didea.version=2020
 .local=E:\repos org.apache.maven.plugins:maven-clean-plugin:2.5:clean
[ERROR] Error executing Maven.
[ERROR] The specified user settings file does not exist: D:\apache-maven-3.8.1\conf\settings.xml
```

解决方案：setting.xml配置的路径错误，修改了setting.xml文件的路径。切setting.xml的缩进不规
。使用VS Code格式化后完成清除

项目下载

Github: https://github.com/z875479694h/netty-one