# 新版采集程序设计文档

# 前言 由于程序较为依赖协议文档，建议结合相关协议文档进行阅读。本文档通俗易懂，相信耐心读完后会对整个采集程序的框架有一个大致的理解，从而更好地在其基础上修复尚未发现的bug，或进行二次开发。

## 背景与目的

每个集中器通过采集器与成百上千的水表进行交互，包括向水表发送开关阀指令，获取与存储水表的读数与状态。智能抄表系统是实现了一个网站，通过网页能够向集中器下发采集、读取、下载档案、控制水表开关阀等各种命令，并能做各种查询，而负责与集中器直接通讯的采集程序就是其中关键的一环，至于另一部分——web程序则在于交互、查询和展示。

简单来讲，采集程序的功能就是根据特定的协议，通过socket编程完成与大量集中器的数据交互，它能支持高并发，并能对硬件的不稳定通信（比如命令执行过程中掉线）提供一定的处理方案。程序采用的主要框架有用于网络通信的Netty，用于与数据库交互的MyBatis，以及用于定时任务的Quartz等，但这些只是工具，在使用方面通过官方文档或相关书籍较容易掌握，因此本文只讲相关要点，不会过多赘述框架的原理。本文档将重点从设计思路的角度，详细描述程序的架构，为阅读该程序或将在其基础上进行二次开发的技术人员提供帮助。

## 程序结构

### 2.1 五大线程与主方法

程序的主要子线程有以下5个：服务端数据接收与处理类(CollectServer)、命令获取类(CommandFetcher)、命令处理类(CommandExecutor)、状态监视类(ConditionMonitor)、定时任务处理类(ScheduleTask),图2-1展示了采集程序文件结构与核心文件夹，这几个部分以独立的类被放置在文件夹“core”下。图2-2则展示了采集程序的整体设计架构。下面通过简单介绍下这几个线程来描述程序的整体设计。

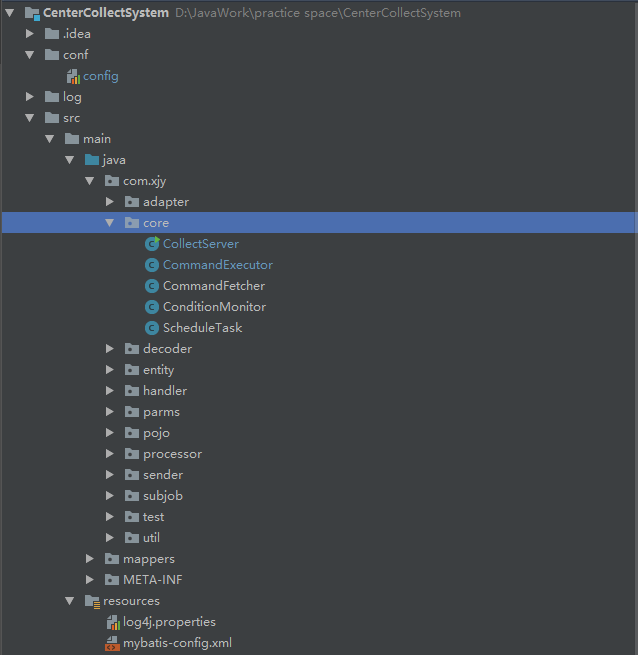


图2-1 采集程序文件结构与核心文件夹



图2-2 采集程序整体设计架构图

首先，CollectServer 即Socket的服务端监听与数据接收程序，是整个程序最核心的部分，通过Netty来实现非阻塞式的高并发通信，完成了字节流的解码和数据处理的工作，这部分也将在之后重点详述。

CommandFetcher和CommandExecutor较为简单，它们都是线程类，实现了Runnable接口。通俗地描述它们的功能就是一个负责生产，一个负责消费。

命令获取线程CommandFetcher不断从数据库中查找每个集中器的待执行命令，并添加到每个集中器的命令执行队列中。这些命令是操作人员在点击网页的相关按钮时，由web程序写入到数据库的t\_TempCommandQueue表中的（这里需要提一下，原来的t\_PlanCommandQueue计划命令表，在新版程序中已经不再使用）。

命令执行线程CommandExecutor则针对每个集中器，不断地从它的命令队列中取出待执行命令进而执行。值得一提的是，存储命令的数据结构采用了ConcurrentLinkedQueue<Command>,它是支持高并发的队列结构，能很好地满足生产者-消费者模式，当然这里对命令存储个数不设限，所以也不算严格意义上的生产者-消费者模式，实现起来更为简单。

状态监视模块ConditionMonitor又是一个线程类，它每隔一段时间检查每个集中器的状态，通过Netty中Channel.isActive()的返回值判断连接是否断开，随后将掉线的集中器的状态进行修改。

定时任务处理模块[[1]](#【1】)ScheduleTask利用Quartz实现定时任务的处理，具体执行的是TimingCollect任务，即定时采集。TimingCollect这个类实现了Quartz提供的Job接口，它通过解析t\_readScheme表中的相关参数，确定每个集中器在当前的时段是否应该执行采集任务，如果是，生成一个新的采集命令插入到t\_tempCommandQueue表中。

综上，再将 CollectServer中的main方法罗列如下，便很容易理解。它先后开启这几个主要线程，然后让Netty服务端绑定特定端口开始监听。

public static void main(String[] args) {  
 new Thread(new ConditionMonitor()).start();//状态监控线程  
 new Thread(new CommandFetcher()).start();//轮询数据库命令的线程  
 new Thread(new CommandExecutor()).start();//命令执行线程  
 ScheduleTask.*startTimingCollect*();//开启定时采集任务  
 try{  
 new CollectServer().bind(Integer.*parseInt*(Constants.*protocolPort*));  
 }catch (Exception e){  
 e.printStackTrace();  
 System.*out*.println("绑定地址错误！\r\n + IP:"+Constants.*connectServer* + "端口号："+Constants.*protocolPort* );  
 }  
}

注：原来的计划命令由另一个程序——定时服务来执行插入，并且插入的是t\_planCommandQueue表，可能考虑的是低耦合以及定时任务的优先级应该偏低，但是有必要改成现在的处理方式。原因有三：第一，集中器一旦开始执行一个任务，是不允许打断的，并且针对集中器同时生成计划命令和临时命令的可能性极低，与其按优先级处理不如采用先来先服务；第二，网页前端是看不到的t\_planCommandQueue的内容的，就不知道集中器是否在执行定时任务，这是一个弊端；第三，定时采集时常出问题，经常需要两头判断，到底是定时服务没部署，还是采集程序出了问题，这会为程序的维护带来一定的麻烦。因此，定时任务现在也由新版采集程序一并处理，并且不再采用t\_planCommandQueue表。

### 2.2 实体类与存储结构

在对程序的主体处理流程作介绍前，必须先介绍几个主要的实体类，它们都在entity文件夹下，如图2-3所示。

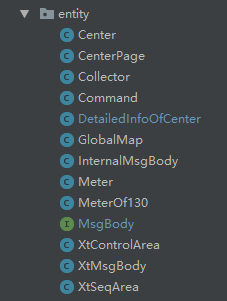


图2-3 实体类

#### 2.2.1 硬件对应实体类Center、Collector和Meter

最关键的类自然是Center类，代表集中器，准确地讲，集中器的准确译文应为Concentrator，但沿用原来程序的命名，使用Center来表示。这个类的所有字段罗列如下：

private String id; //集中器编号  
private Integer dbId;//集中器在数据库中的索引  
private List<Collector> collectors = new ArrayList<>();//采集器列表  
private LocalDateTime heartBeatTime; //最近一次心跳时间  
private LocalDateTime readTime;//最近一次读取（成功）时间  
private ChannelHandlerContext ctx; //绑定ChannelHandler的上下文对象，用于发送数据，检测数据流向  
private String enprNo ; //所属水司  
private DetailedInfoOfCenter information;  
private Command curCommand; // 当前正在执行的命令  
private MsgBody latestMsg; //最近一次发送的命令  
private ConcurrentLinkedQueue<Command> commandQueue = new ConcurrentLinkedQueue<>(); //待执行的命令队列

其中几个比较关键的字段：id表示集中器地址，唯一识别每个集中器；collectors表示集中器下的所有采集器[[2]](#【2】)；curCommand表示当前正在执行的命令，commandQueue表示队列中待执行的命令；ctx是借用了Netty中的ChannelHandlerContext类，即通道上下文，通过它可以获取信道连接的状态和发送数据，每个集中器的ctx都是不同的，是一一对应的关系；MsgBody是笔者自己定义的接口，用于封装数据实体，latestMsg字段记录服务器最近向集中器发送的报文体，用于实现消息重发，可以理解，每当程序向集中器发送一次指令，该字段都会被刷新。

采集器[[2]](#【2】)实体类Collector和水表实体类Meter的字段较为简单，在程序中注释得也较为清晰，分别罗列如下。另外MeterOf130类继承了Meter类，它能表示更多与130协议相关的信息，详见代码与注释。

采集器Collector类字段：

private String id;//采集器地址  
private List<Meter> meters = new ArrayList<>(); //采集器下的所有水表  
private Center center; //所属集中器

水表类Meter类字段：

private String id; //表地址  
private Integer indexNo;//表序号，在130协议中使用  
private Double value = 0.0; //表读数  
private Integer state = 0; //通信状态 0-正常 1-数据读取失败 2 - 采集器失败  
private Integer valveState = 0; //阀门状态 0-无阀 1-开阀 2-关阀  
private Collector collector; //所属采集器  
private Integer collectorIndex; // 针对内部协议：采集器索引，下载档案时用到（动态数据）

注：这里的采集器实际上代表的是采集通道，一个采集器可以通过多个通道对水表数据进行采集，比如采集器实际编号为111111111100，但它有3个通道，可以分别为111111111101、111111111102、111111111103，如果是无线采集器，通常没有多通道，则通道地址只有111111111100，**数据库中的资料以及集中器中的档案资料实际上存储的是采集器的通道编号，**这个细节了解即可，本文接下来将只用采集器来描述。

#### 2.2.2 命令实体类

命令由web部分前端的操作人员产生，采集程序从数据库取出命令，使用Command的类进行封装。2.1中描述的CommandFetcher和CommandExecutor都是针对Command类进行处理的。它的字段罗列如下：

CommandType type= CommandType.*NONE*;//命令类型  
Integer id; //命令对应的数据库中的Id  
Integer state; //0-未分派 1-集中器未连接 2-等待执行 3-执行中 4-执行错误，重试中 5-执行成功 6-执行失败  
String[] args; //命令的相关参数  
LocalDateTime startExecuteTime; //开始执行命令的时刻  
LocalDateTime generateTime; //命令生成时间  
int secondsLimit = 5 \* 60; //允许超时时间，默认5分钟  
boolean suspend = false; //命令在执行中是否中断  
int retryNum = 0; //重试次数  
Object parameter; //自定义参数  
int allowedRetryTimes = 1; //允许重试次数

其中命令类型CommandType是自定义的枚举类，它的实现如下：

public enum CommandType {  
 *NONE*(""), //无命令  
 *READ\_SINGLE\_METER*("000106"), //读取单个表  
 *READ\_ALL\_METERS*("000102"), //读取所有表  
 *READ\_CENTER\_INFO*("000103"),//读取集中器信息  
 *COLLECT\_FOR\_METER*("000105"),//采集单个表  
 *COLLECT\_FOR\_COLLECTOR*("待定"),//针对采集器采集  
 *COLLECT\_FOR\_CENTER*("000101"), //针对集中器采集  
 *OPEN\_VALVE*("000302"), //开阀  
 *CLOSE\_VALVE*("000304"),//关阀  
 *OPEN\_VALVE\_BATCH*("000301"),//批量开阀  
 *CLOSE\_VALVE\_BATCH*("000303"),//批量关阀  
 *CHECK\_CLOCK*("000204"),//校验时钟  
 *WRITE\_INFO*("000208") ; //写入资料到集中器  
 private String value;  
 private CommandType(String value){  
 this.value = value;  
 }  
 public String getValue(){  
 return value;  
 }  
}

每个value值是由最初的设计人员制定的命令码，它们与命令类型一一对应。大概了解了Command的类的结构，就可以简单看看CommandExecutor类中的executeForInternalProtocol()方法，它的主体逻辑很简单，可以参考下面罗列的代码结构。入参是集中器Center对象和当前的命令currentCommand（实际上一个Center参数就可以，因为Center有curCommand这个字段），方法根据当前命令的类型作相应的处理，比如发送给集中器一条指令。

private static void executeForInternalProtocol(Center center, Command currentCommand) {  
 switch (currentCommand.getType()){  
 case *OPEN\_VALVE*://开阀：先打开节点  
 //do something，like sending the command  
 break;  
 case *CLOSE\_VALVE*://关阀：先打开节点  
 //do something  
 break;  
 case *WRITE\_INFO*:

……

#### 2.2.3 数据报文实体类

数据报文实体类用于将字节信息进行封装，它与协议类型密切相关，本程序使用到的是内部协议（千宝通水司对188协议的改版）和130协议（具体协议内容请参考相关文档），所以数据实体类有两个：InternalMsgBody(对应内部协议)和XtMsgBody（对应130协议），它们都实现了MsgBody接口。MsgBody接口只定义了一个方法：toBytes(),用于将数据实体类转为字节数组进行传输。

public interface MsgBody {  
 byte[] toBytes();  
}

下面将以InternalMsgBody为例进行简单介绍。InternalMsgBody的字段罗列如下，包括**数据包的类型、有效数据的字节数组、设备号**等。对于内部协议，每个数据包都会包含设备号，也就是集中器地址，因此通过解析出deviceId，便可知道，消息体来自哪个集中器。

InternalMsgType msgType = InternalMsgType.*INVALID\_PACKAGE*; //3种类型，心跳包，发送数据包，接收数据包  
String deviceId; //11字节的设备号  
int[] effectiveBytes;//有效数据  
int crcCode; //crc校验码

InternalMsgBody的其他主要方法如构造方法和toBytes()方法用于数据解析或封装。比如它的构造方法之一就是接收原始字节数组，分离出设备号、有效数据、校验码这几部分信息，并对相应字段进行赋值。而具体的数据解析方法只需要对MsgBody进行进一步处理即可，这些将在之后介绍。

#### 2.2.4 全局单例Map

程序包含多个线程，对同一个协议的大量集中器同时发送来的数据进行的处理工作，实际上也是多个线程调用相同的回调函数实现的。在并发的环境下，所有连接上本服务的集中器的性质是全局性的。程序要把控所有集中器的实时状态，并希望随时可以访问指定的集中器，这就需要全局的容器来存储集中器的信息。GlobalMap类中就定义了这样的容器，它采用内部静态类的方式实现了单例模式。

public class GlobalMap {  
 private GlobalMap(){}  
 private static class InnerMap{  
 private static ConcurrentHashMap<String,Center> *map* =

new ConcurrentHashMap<>();  
 }

//获得实时集中器状态表，key为集中器地址  
public static ConcurrentHashMap getMap(){

//getMap方法第一次调用时，InnerMap实例才会被创建  
 return InnerMap.*map*;  
 }

}

程序使用ConcurrentHashMap来存储所有集中器的信息，它是Java中最重要的线程安全的并发容器之一。key为集中器地址，因为根据地址可以唯一识别每个集中器，value为前面介绍过的Center——集中器实体类。针对特定协议的其他全局的信息，GlobalMap中还定义了其他的ConcurrentHashMap,比如InnerInfo内部类中定义的persistentDataOfInternalProtocol,它的key是Center，value是List<CenterPage>。由于内部协议的集中器以“页”的形式组织数据，于是设计CenterPage实体类与之相对应，代表集中器每页的信息，具体的可参考程序与注释。

### 2.3 接收数据处理流程

#### 2.3.1 Netty服务端编程模式

Netty通信模块是整个程序的核心，它是一个良好的NIO框架，封装了完整的功能，使用起来相较NIO更为方便，并避免了NIO中棘手的epoll空轮询的问题。本程序的CollectServer中的bind方法对Netty的使用即遵照Netty的基本使用模式：

EventLoopGroup bossGroup = new NioEventLoopGroup();  
EventLoopGroup workerGroup = new NioEventLoopGroup();  
try{  
 ServerBootstrap b = new ServerBootstrap();//NIO服务端的辅助启动类  
 //通过辅助启动类进行配置  
 b.group(bossGroup,workerGroup)//定义线程组  
 //选择通道类型  
 .channel(NioServerSocketChannel.class) //指定NIO的模式  
 .option(ChannelOption.*SO\_BACKLOG*,1024) //配置tcp缓冲区，对已经建立的连接不影响  
 //绑定事件处理方法  
 .option(ChannelOption.*SO\_SNDBUF*,32 \* 1024)//发送缓冲大小  
 .option(ChannelOption.*SO\_RCVBUF*,32 \* 1024)//接收缓冲大小  
 //.option(ChannelOption.ALLOW\_HALF\_CLOSURE,true)  
 .option(ChannelOption.*SO\_KEEPALIVE*,true)  
 .option(ChannelOption.*SO\_REUSEADDR*,true)  
 .childHandler(new ChildChannelHandler()); // 拿到SocketChannel,交给具体数据的处理类  
 //绑定端口，同步等待成功  
 ChannelFuture f = b.bind(port).sync(); //实际服务程序到此阻塞，保持监听  
 //等待服务器监听端口关闭  
 f.channel().closeFuture().sync();  
} catch (InterruptedException e) {  
 e.printStackTrace();  
}finally {  
 bossGroup.shutdownGracefully();  
 workerGroup.shutdownGracefully();  
}

1. 定义两个线程组**EventLoopGroup** ,其中bossGroup负责接收连接，把连接信息注册到workerGroup上，由workerGroup负责每个通道的连接和数据收发维护。
2. 利用**ServerBootstrap**对通道的一些option参数进行配置，最后通过channelHandler方法指定具体的通道配置类ChildChannelHandler，此后当每个通道接收到数据时，Netty就会回调ChildChannelHandler中已经配置好的ChannelHandler通道处理器进行处理。这也是典型的异步的处理模式。
3. 自实现**ChildChannelHandler**（继承ChannelInitializer）完成对**pipeline**的构造，pipeline采用责任链式处理，在pipeline中按序定义数据的解码器、消息处理器等。如下便是对ChildChannelHandler的定义，它继承了Netty提供的ChannelInitializer，实现初始化通道的方法。程序根据配置文件中的信息获取协议类型，如“XT”表示130协议，则往通道中先后添加自定义的XtProtocolDecoder和XtMessageHandler，前者负责将原始字节流拆分成一个个有效的数据包，后者分析每个数据包的具体信息进行业务处理。内部协议的处理方式与之相似。可以这样想像，Netty中的pipeLine是一个数据传送带，而每个ChannelHandler代表传送过程中要经过的一道道加工流程，本程序的处理类，无论是XtProtocolDecoder或XtMessageHandler,它们的父接口都是ChannelHandler。

private class ChildChannelHandler extends ChannelInitializer<SocketChannel> {  
 protected void initChannel(SocketChannel socketChannel) throws Exception {  
 if(Constants.*protocol* != null && Constants.*protocol*.equals("XT")){//130  
 socketChannel.pipeline().addLast(new XtProtocolDecoder());  
 socketChannel.pipeline().addLast( new XtMessageHandler());  
 }else{//默认内部协议  
 socketChannel.pipeline().addLast( new InternalProtocolDecoder());  
 socketChannel.pipeline().addLast( new InternalMessageHandler());  
 }  
 }

}

图2-4则表示了处理类与ChannelHandler接口的继承关系。图2-5具体地演示了每个通道数据的处理流程的结构。



图2-4 ChannelHandler的继承关系



图2-5 Netty中的pipeLine和ChannelHandlers

#### 2.3.2 自定义解码器

Netty框架为很多协议的通信提供了几种典型解码器的支持，进而解决数据的分包粘包问题，如LineBasedFrameDecoder和DelimiterBasedFrameDecoder等，前者使用行尾控制字符(\n或\r\n)作为分隔符来解析数据，后者则更为强大，可以自定义分隔符来解析数据。然而它们都不能很好地适用于130协议或内部协议。首先，130协议规定报文尾为0x16，但它很容易在数据中也出现，所以制定0x16不可行，另外报文头是0x68 + L + L + 0x68,它涉及到了变量——也就是报文长度，也不好作为分隔符。至于内部协议，它的报文头是0x7B010016，是4字节的固定值，乍看好像可行，但若将其作为分隔符，有这样的问题存在：

当操作人员发送读取命令，只读取1页的数据，集中器迅速返回了一个数据包，放入Netty的数据缓冲区，然而，由于下一个报文头要经过1分钟才能到达（心跳包间隔往往是1分钟），以至于ChannelHandler要等待一分钟才能处理从缓冲区中弹出的数据，这就造成了读数据的低效！因此，无论是内部协议还是130协议，本程序都采用自定义的解码器进行处理，本节将对其处理逻辑进行大概介绍，具体请看程序源码。

##### 2.3.2.1 内部协议解码器处理逻辑

内部协议的设计并不是很好，它的心跳包没有报文尾，数据包以0x45结尾，此外报文中没有报文长度的信息，这些信息在做解码操作时需要注意。InternalProtocolDecoder继承了Netty提供的ByteToMessageDecoder类，实现这个父类提供的decode方法即可。以下是decode方法的签名：

protected void decode(ChannelHandlerContext ctx, ByteBuf in, List<Object> out)

其中，in是接收数据缓冲区的内容，ctx是整个通道上下文信息，out是数据处理后的结果存放容器。

解码流程如下（可结合协议文档理解）：

（1）判断数据缓冲区ByteBuf中的可读字节大小，如果小于一个心跳包长度，则返回不做处理；

（2）使用findHead方法寻找ByteBuf中的所有报文头所在的位置，存放在一个List中；

（3）对于List中存放的报文头位置，如果相邻两个报文头相距为心跳包的长度，则弹出前一个报文头到后一个报文头之间的数据；如果后一个报文头的前一个字节为0x45（或前两个字节为0x45 0x0d，这是部分无线集中器烧录程序的问题，实际测试中发现），则表明是一个数据包，于是弹出前一个报文头到后一个报文头之间的数据；如果两个报文头相距小于心跳包的长度，或**两个报文头相距超过心跳包长度，前一段字节却不是以0x45或0x450d结尾的，将其判断为一个无效包，并打印出来**。其实对于超过心跳包长度的情况，可以从后一个报文头往前遍历找第一个0x45的位置，但麻烦且没有必要（当然这样改更好），因为基本不可能出现0x45不在前两个字节的情况，即便出现了也能及时打印发现，像0x450d这种个别集中器的情况就是这样发现的。而存在一个指令0x454545（EEE代表打开采集通道失败），如果来一个非完整的这个指令的数据包（概率极低），因为有0x45，也会被当做完整包弹出。

（4）弹出的数据作为InternalMsgBody构造函数的入参，构造出一个InternalMsgBody（内部协议报文实体类）的对象，放入out列表中，传给下一个处理器，也就是handler文件夹下的业务处理类InternalMessageHandler。

##### 2.3.2.2 130协议解码器处理逻辑

130协议的解码器类叫XtProtocolDecoder，与InternalProtocolDecoder一样位于decoder文件夹下,它也继承了Netty提供的ByteToMessageDecoder类，实现了decode方法。因为130协议设计相比与内部协议更为合理，解码起来也更为方便。

130协议解码流程（可结合协议文档理解）：

1. ByteBuf中的可读字节小于包的最小长度20，不作处理马上返回；
2. 找到这样一个报文段,data[i]处为0x68且data[i+5]处也为0x68,初步判定这一段为报文头(也可再进一步判断两个L是相同的)；
3. 根据data[i+1]和data[i+2]解析报文长度，通过和校验后提取出有效字节数组effectiveData；
4. 根据effectiveData构造XtMsgBody（130协议报文实体类），放入out列表中，传给下一个ChannelHandler，也就是handler文件夹下的XtMessageHander进行进一步的业务处理。

#### 2.3.3 业务消息处理器

整个程序的报文实体处理器有两个，分别是InternalMessageHandler和XtMessageHandler，它们都位于handler文件夹下，也都继承了Netty提供的ChannelHandlerAdapter类，是业务处理的核心所在。它们的处理思路其实很简单，**即判断数据包的类型，然后根据不同类型的数据包调用不同的方法进行数据处理**，因为具体实现细节与协议息息相关，本节不会过多详述，请读者通过源代码去深入了解。当然，了解基本结构即可做程序的扩展工作，不必求甚解。

##### 2.3.3.1 内部协议业务消息处理器

InternalMessageHandler继承了ChannelHandlerAdapter，处理数据逻辑在方法channelRead中实现，channelRead的方法签名如下：

public void channelRead(ChannelHandlerContext ctx, Object msg)

其中ctx之前已经介绍过，是通道上下文信息，通过它可以检测Socket的连接状态、发送数据等，msg实际上就是上一个ChannelHandler带过来的信息，本程序中自然就是InternalMessageBody。因此不难理解程序的第一行：

XtMsgBody msgBody = (XtMsgBody)msg;

即将msg强转为InternalMsgBody。

内部协议业务消息处理逻辑：

1. 提取出InternalMsgBody中的设备地址信息，即集中器编号address，如果它不存在于全局的map中，说明是此集中器的第一次上线，于是根据地址address和ctx构造Center类，并将<address,center>作为键值对放入map中，此时数据库中有此集中器的命令是正在执行的，全部将其置为成功，避免程序重启时对后续操作人员发布的命令的执行造成影响。如果address已经存在于map中，更新其Center对象的ctx，并判断它是否有命令处于挂起状态（即命令执行到一半出现掉线的情况），如果有且未超过命令的允许重试次数，重发上一条命令。随后，更新集中器的最近心跳时间，同步更新数据库中集中器的心跳时间字段。
2. 当数据包不是心跳包时，根据不同的数据包类型，调用不同的方法进行处理。内部协议中，集中器数据包会包含连续3字节的指令码，消息处理器根据它判断指令类型，它们大多是集中器的回复，包括读数据回复、下载档案回复、开关阀回复等，比如读回复的指令码就是“RRR”的ASCII码。如下的程序片段便表示了不同消息处理的简单结构。

if(instruction.equals(InternalOrders.*READ*)){ //读取指令,按页解析读数  
 if(currentCenter.getDbId() == null) DBUtil.*preprocessOfRead*(currentCenter);//针对定时读取尚未初始化的情况  
 InternalMsgProcessor.*readProcessor*(currentCenter,msgBody);  
}

else if(instruction.equals(InternalOrders.*COLLECT*)){//采集指令，说明采集器已经开始采集  
 LogUtil.*DataMessageLog*(InternalMessageHandler.class,"集中器已经开始采集！");  
}  
else if(instruction.equals(InternalOrders.*DOWNLOAD*)){  
 //下载档案命令的处理器，先读取页数，是最后一页的话命令成功结束  
 InternalMsgProcessor.*writeProcessor*(currentCenter,msgBody);  
}

具体的命令处理流程请参照协议阅读源代码。处理方式往往是调用processor包下的InternalMsgProcessor辅助类中的方法，或是直接调用sender包下的InternalProtocolSenderHelper进行消息的发送。这里不过多赘述。

##### 2.3.3.2 130协议业务消息处理器

130协议的业务消息处理类是XtMessageHandler,它与InternalMessageHandler类似，同样继承了ChannelHandlerAdapter，并重写channelRead方法来做业务消息的处理。

public void channelRead(ChannelHandlerContext ctx, Object msg)

这里的msg就变成了XtMsgBody，也就是130协议的报文实体类。于是第一行程序如下，也就是将msg先强转为XtMsgBody。

XtMsgBody msgBody = (XtMsgBody)msg;

130协议业务消息处理逻辑：

1. 提取出XtMsgBody中的设备地址信息，即集中器编号address，如果它不存在于全局的map中，说明是此集中器的第一次上线，于是根据地址address和ctx构造Center类，并将<address,center>作为键值对放入map中，此时数据库中有此集中器的命令是正在执行的，全部将其置为成功，避免程序重启时对后续操作人员发布的命令的执行造成影响。如果address已经存在于map中，更新其Center对象的ctx，并判断它是否有命令处于挂起状态（即命令执行到一半出现掉线的情况），如果有且未超过命令的允许重试次数，重发上一条命令。随后，更新集中器的最近心跳时间，同步更新数据库中集中器的心跳时间字段。
2. 130协议的数据包类型由AFN功能码和Fn确定，XTParams类中定义了130协议相关的常数，其字段含义已经明确地由名称表达，比如：

public static final int *AFN\_GET\_PARAM* = 0X8A; //F1读取档案

当AFN=0x8A时表示获取集中器参数相关的指令，而Fn = 1时表示读取档案。现截取部分消息处理的程序段加深理解：

if(AFN == XTParams.*AFN\_LINK\_DETECTION*){//链路检测数据，如心跳包  
 //回复心跳包，也可不回复  
} else if(AFN == XTParams.*AFN\_GET\_REALTIME\_DATA*){ //请求实时数据报文

//做抄表数据的处理工作  
} else if(AFN == XTParams.*AFN\_CONFIRM\_OR\_DENY*){//设置参数报文返回确认或否认，比如下载档案到集中器  
 if(command.getType() == CommandType.*WRITE\_INFO* ){//如果命令是下载档案  
 if(msgBody.getC() == 0x80) //成功  
 else if(msgBody.getC() == 0x89)//失败  
 }  
}else if(AFN == XTParams.*AFN\_GET\_PARAM*){//获取参数报文（如读取档案）  
 if(command.getType() == CommandType.*READ\_CENTER\_INFO*){  
 //读取集中器信息，写到日志  
 }  
}

要了解更多程序细节需要结合协议文档阅读源代码。

#### 2.3.4 差错处理

在通信过程中，难免会出现集中器掉线的情况，甚至是在命令执行过程中出现掉线，本节简单介绍这些异常情况的处理方式。

对于普通的Socket编程，我们通常需要使用try-catch去主动捕获相关异常，比如客户端突然掉线通常会抛出IOException（远程主机关闭连接），然而Netty对这类异常的处理同样有很好的支持，它能够捕获每个Channel的数据处理过程中产生的各种异常。ChannelHandler接口定义了如下方法：

void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws Exception; //捕获到异常cause时调用

开发人员可以通过实现此方法完成对cause的处理。此外，还有更多接口可实现，

列举其中两个如下：

void disconnect(ChannelHandlerContext ctx, ChannelPromise promise) throws Exception;//断开连接时调用

void channelReadComplete(ChannelHandlerContext ctx) throws Exception;//通道完成读时调用

disconnect方法在客户端断开连接时调用，channelReadComplete方法用于在通道完成读时调用。还用更多的接口方法请参考Netty的ChannelHandler接口源码或文档。

面对命令执行到一半客户端断开连接的场景（通常由于信号不佳或集中器断电），重写ChannelHandlerAdapter的disconnect方法是最好的选择。

重写方式也非常简单，从全局的map中寻找掉线的集中器（根据其ctx域是否相同），如果此集中器当前有命令正在执行，则将此命令的挂起状态设为true，随后打印相关日志，其核心代码如下：

for(Map.Entry<String,Center> entry : map.entrySet()){  
 if(entry.getValue().getCtx() == ctx){  
 centerAddr = entry.getKey();  
 center = entry.getValue();  
 if(center.getCurCommand()!=null && center.getCurCommand().getState()==CommandState.*EXECUTING*){  
 center.getCurCommand().setSuspend(true); //挂起命令  
 LogUtil.*channelLog*(centerAddr,"center disconnected when executing the command --> command "+ center.getCurCommand().getType() + " suspend");  
 }  
 break;  
 }  
}

这样一来，当集中器重新连接上，在2.3.3中提到的业务消息处理器会判断命令是否挂起，并结合重试次数进行命令重发操作。

### 2.4 命令发送流程

命令发送过程实际上就是编码过程，相当于接收数据处理的逆过程。但结合前文内容，本节无需过多阐述，只需简单介绍即可。

在2.2.3节提到了如下的代码结构，即根据命令类型，做不同的事情。

private static void executeForInternalProtocol(Center center, Command currentCommand) {  
 switch (currentCommand.getType()){  
 case *OPEN\_VALVE*://开阀：先打开节点  
 //do something，like sending the command  
 break;  
 case *CLOSE\_VALVE*://关阀：先打开节点  
 //do something  
 break;  
 case *WRITE\_INFO*:

……

下面是130命令（实时抄读集中器所有表）发送在程序中的实际写法，首先设置超时时间（一些命令使用默认的5分钟为超时时间即可），随后设置命令过程中的参数，比如读集中器档案一开始已读的表个数为0，所以把表序号的偏置量设置为0。最后，调用XTProtocolSendHelper类的静态方法readMeters，完成消息的发送。

case *READ\_ALL\_METERS*:  
 *timeLimitSetFor130*(center);  
 currentCommand.setParameter(0); //表序号的偏置量，表示已经读到了哪个表  
 XTProtocolSendHelper.*readMeters*(center,currentCommand);  
 break;

所有发送相关的方法都在sender包下，根据协议类型区分，目前有两个类：

InternalProtocolSendHelper和XTProtocolSendHelper。这两个类中除了一些公用的辅助方法，所有方法的核心都是根据协议和命令类型构造报文实体类，然后调用报文实体类的toBytes方法将其转为字节数组，最后调用writeAndFlush方法进行发送。如下是程序自定义的writeFlush方法：

public static void writeAndFlush(Center center,XtMsgBody msgBody){  
 ByteBuf buf = Unpooled.*copiedBuffer*(msgBody.toBytes());//拷贝字节数组  
 center.getCurCommand().setStartExecuteTime(LocalDateTime.*now*()); //刷新开始执行时间  
 ChannelFuture f = center.getCtx().writeAndFlush(buf);//发送数据  
 *printMsgLog*(msgBody);//打印日志  
 center.setLatestMsg(msgBody);//更新集中器最近发送的一条数据  
}

具体的构造字节数组是与协议息息相关的，结合协议文档和源代码便很容易读懂。

### 2.5 数据访问层

本程序的数据访问层简单使用了MyBatis框架来完成，读者只需简单了解MyBatis的用法便能轻松理解。当然采用原生的JDBC或其他数据库框架都能满足需求。值得一提的是，由于原数据库的表设计的很多字段并没有在实际中使用，并且一些命名也较不规范，为了更好的做逻辑与表数据层的分离，本程序在entity包外特别创建了一个pojo包，下面专门放与数据库表对应的对象。如图2-6所示。

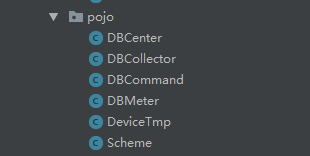


图2-6 数据库表的相关对象

通过adapter包下的适配器类，可以完成表对象到逻辑对象entity的转换。以DBMeter为例：

public static Meter getMeter(DBMeter dbMeter){  
 Meter meter = new Meter();  
 meter.setId(dbMeter.getiAddr());  
 meter.setIndexNo(dbMeter.getIndexNo());  
 meter.setValue(dbMeter.getShowValue());  
 meter.setState(dbMeter.getStatue());  
 return meter;  
}

逻辑层的Meter只需要DBMeter的id、indexNo、value、state等四个字段，用MeterAdapter的getMeter方法即可完成转化。当然，也可以利用MyBatis中的别名，从而避免pojo下的这些类的使用，但需要在配置文件中写额外的语句进行别名配置（也挺麻烦）。不过确实只使用Meter对象来承接表数据会让程序更简单，但由于涉及的表也并不是很多，加个adapter转化的话，笔者个人觉得逻辑层更为清晰些，也算有利有弊吧。

接下来简单讲讲数据访问层的结构。

逻辑层要访问数据，只需调用util包下的DBUtil类的相关静态方法，举例如下：

public class DBUtil {  
 //获取某个集中器的待执行命令队列（这是很频繁的操作）  
 public static List<DBCommand> getCommandByCenterAddress(String address){  
 SqlSession session = MyBatisUtil.*getSqlSessionFactory*().openSession();  
 CommandMapper mapper = session.getMapper(CommandMapper.class);  
 List<DBCommand> DBCommands = mapper.getCommands(address , CommandState.*UN\_ENQUEUED*,"%"+Constants.*connectServer*+":"+Constants.*protocolPort*+"%");  
 session.close();  
 return DBCommands;  
 }

上面列出了获取集中器的待执行命令的操作，完成这几个步骤即可：打开session会话、获取相关Mapper、执行Mapper中定义的相关方法、返回结果。而所有Mapper接口和对应的xml文档，全部在mapper文件夹下（如图2-7），一目了然。另外，整个MyBatis的配置位于resources文件夹下的mybatis-config.xml。

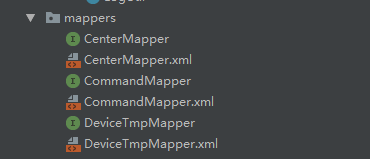


图2-7 MyBatis接口映射配置

## 程序配置与部署

程序的全局配置文件位于conf文件夹下的config文件中，其中的内容和填写方式已用注释的方式明确标明，如图3-1所示。



图3-1 配置文件的内容

实际部署时，可建立一个父文件夹,整个程序利用maven打成的jar包位于和conf文件夹都位于此文件夹下，如图3-2所示。其中conf文件夹需要事先创建，并且在里面建立创建一个config文件（也可直接拷贝idea工作空间下的本程序的config文件），完成对配置参数的填写，而log文件夹由程序自动生成，不必额外创建。

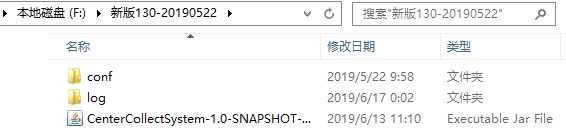


图3-2 程序所在文件夹

运行程序时，可在此文件夹下打开命令行窗口，在控制台输入：

java –jar CenterCollectSystem

按Tab键自动补全，随后回车，程序即可运行。如果出现端口绑定错误的问题，需要先查看该端口是否已经被占用。

## 总结

本文介绍了整个集中器数据采集程序的设计结构和配置与部署方式。目前程序运行较为稳定，没有新的bug出现。如果有水司反映问题，特别是集中器实际在线却在网页版显示“不在线”，优先考虑数据库server表和port表的相关配置是否不全（这需要进入后台管理系统操作），其次考虑采集程序的配置文件是否填写正确，对于这一点，水司的维护人员也比较熟悉了。

回看整个程序，无论是包结构还是逻辑相较于原来的C#旧版程序都更为简单清晰，可读性也更强，但是无论是哪个版本，都需要阅读相关的通讯协议文档以便于理解。

程序借助Netty框架来完成网络编程，它能更好地管理socket底层的连接，让开发人员能将更多精力投入于逻辑层，而无需过多担心超时、断连等差错的处理。但本程序在对Netty的使用方面还处于较为基本的层面，不算特别灵活地应用。如果有二次开发人员熟读Netty源码，大可在此程序的基础上予以进一步的改进。

此外，还有另外一个新天通讯协议，也叫做中原油田协议，仍然需要用旧版采集程序运行，但这只牵涉到两个烧错了程序的集中器，因此新版不实现此协议。新天通讯协议是130的改版，没必要的话无需阅读其文档。如果之后要实现新的协议，只需在源程序的基础上添加即可。

## 参考文献

1.《Netty权威指南》

2. Netty5用户指南<http://ifeve.com/netty5-user-guide/>

## 作者 许加耀 华中科技大学水电学院信息与控制研究所

邮箱：997858235@qq.com

## 2019年6月17日