数据采集服务

学习目标：
1. 能够了解ETL、大数据和物联网数据采集技术
2. 能够了解Netty网络编程
3. 能够知道Netty网络编程过程和实践
4. 能够知道冷链设备监控中如何使用netty进行数据流转和处理

1 数据采集技术

1.1 ETL概述

ETL是将业务系统的数据经过抽取、清洗转换之后加载到数据仓库的过程，目的是将企业中的分散、零乱、标准不统一的数据整合到一起，为企业的决策提供分析依据。 ETL是BI项目重要的一个环节。 通常情况下，在BI项目中ETL会花掉整个项目至少1/3的时间,ETL设计的好坏直接关接到BI项目的成败。

ETL的设计分三部分：Server2005的SSIS服务、Informatic等，IBM data stage， 开源框架kattle、DataX等技术)实现，一种是SQL方式实现，另外一种是ETL工具和SQL相结合。前两种方法各有各的优缺点，借助工具可以快速的建立起ETL工程，屏蔽了复杂的编码任务，提高了速度，降低了难度，但是缺少灵活性。SQL的方法优点是灵活，提高ETL运行效率，但是编码复杂，对技术要求比较高。第三种是综合了前面二种的优点，会极大地提高ETL的开发速度和效率。

1.1.1 数据抽取

数据抽取需要在调研阶段做大量的工作，首先要搞清楚数据是从几个业务系统中来,各个业务系统的数据库服务器运行什么DBMS,是否存在手工数据，手工数据量有多大，是否存在非结构化的数据等等，当收集完这些信息之后才可以进行数据抽取的设计。

1. 对于与存放DW的数据库系统相同的数据源处理方法

这一类数据源在设计上比较容易。一般情况下，DBMS(SQLServer、Oracle)都会提供数据库链接功能，在DW数据库服务器和原业务系统之间建立直接的链接关系就可以写Select 语句直接访问。

2. 对于与DW数据库系统不同的数据源的处理方法

对于这一类数据源，一般情况下也可以通过ODBC的方式建立数据库链接——如SQL Server和Oracle之间。如果不能建立数据库链接，可以有两种方式完成，一种是通过工具将源数据导出成.txt或者是.xls文件，然后再将这些源系统文件导入到ODS中。另外一种方法是通过程序接口来完成。

3. 对于文件类型数据源(.txt,.xls)，可以培训业务人员利用数据库工具将这些数据导入到指定的数据库，然后从指定的数据库中抽取。或者还可以借助工具实现。

4. 增量更新的问题

对于数据量大的系统，必须考虑增量抽取。一般情况下，业务系统会记录业务发生的时间，我们可以用来做增量的标志,每次抽取之前首先判断ODS中记录最大的时间，然后根据这个时间去业务系统取大于这个时间所有的记录。利用业务系统的时间戳，一般情况下，业务系统没有或者部分有时间戳。

1.1.2 数据清洗

一般情况下，数据仓库分为ODS、DW两部分。通常的做法是从业务系统到ODS做清洗，将脏数据和不完整数据过滤掉，在从ODS到DW的过程中转换，进行一些业务规则的计算和聚合。

1、 数据清洗

数据清洗的任务是过滤那些不符合要求的数据，将过滤的结果交给业务主管部门，确认是否过滤掉还是由业务单位修正之后再进行抽取。

不符合要求的数据主要是有不完整的数据、错误的数据、重复的数据三大类。

(1)不完整的数据：这一类数据主要是一些应该有的信息缺失，如供应商的名称、分公司的名称、客户的区域信息缺失、业务系统中主表与明细表不能匹配等。对于这一类数据过滤出来，按缺失的内容分别写入不同Excel文件向客户提交，要求在规定的时间内补全。补全后才写入数据仓库。

(2)错误的数据：这一类错误产生的原因是业务系统不够健全，在接收输入后没有进行判断直接写入后台数据库造成的，比如数值数据输成全角数字字符、字符串数据后面有一个回车操作、日期格式不正确、日期越界等。这一类数据也要分类，对于类似于全角字符、数据前后有不可见字符的问题，**只能通过写SQL语句的方式找出来，然后要求客户在业务系统修正之后抽取。**日期格式不正确的或者是日期越界的这一类**错误会导致ETL运行失败，**这一类错误需要去业务系统数据库用SQL的方式挑出来，交给业务主管部门要求限期修正，修正之后再抽取。

(3)重复的数据：对于这一类数据——特别是维表中会出现这种情况——将重复数据记录的所有字段导出来，让客户确认并整理。

数据清洗是一个反复的过程，不可能在几天内完成，只有不断的发现问题，解决问题。对于是否过滤，是否修正一般要求客户确认，对于过滤掉的数据，写入Excel文件或者将过滤数据写入数据表，在ETL开发的初期可以每天向业务单位发送过滤数据的邮件，促使他们尽快地修正错误,同时也可以做为将来验证数据的依据。数据清洗需要注意的是不要将有用的数据过滤掉，对于每个过滤规则认真进行验证，并要用户确认。

1.1.3 数据转换

数据转换的任务主要进行不一致的数据转换、数据粒度的转换，以及一些商务规则的计算。

(1)不一致数据转换：这个过程是一个整合的过程，将不同业务系统的相同类型的数据统一，比如同一个供应商在结算系统的编码是XX0001,而在CRM中编码是YY0001，这样在抽取过来之后统一转换成一个编码。

(2)数据粒度的转换：业务系统一般存储非常明细的数据，而数据仓库中数据是用来分析的，不需要非常明细的数据。一般情况下，**会将业务系统数据按照数据仓库粒度进行聚合。**

⑶**商务规则的计算**：不同的企业有不同的业务规则、不同的数据指标，这些指标有的时候不是简单的加加减减就能完成，这个时候需要在ETL中将这些数据指标计算好了之后存储在数据仓库中，以供分析使用。

1.2 大数据采集技术介绍

数据产生的种类很多，并且不同种类的数据产生的方式不同。对于大数据采集系统，主要分为以下三类系统：

1.2.1 系统日志采集系统

许多公司的业务平台每天都会产生大量的日志数据。对于这些日志信息，我们可以得到出很多有价值的数据。通过对这些日志信息进行日志采集、收集，然后进行数据分析，挖掘公司业务平台日志数据中的潜在价值。为公司决策和公司后台服务器平台性能评估提高可靠的数据保证。

系统日志采集系统做的事情就是收集日志数据提供离线和在线的实时分析使用。目前常用的开源日志收集系统有Flume、Scribe等。 Apache Flume是一个分布式、可靠、可用的服务，用于高效地收集、聚合和移动 大量的日志数据，它具有基于流式数据流的简单灵活的架构。其可靠性机制和许多故障转移和恢复机制，使Flume具有强大的容错能力。Scribe是Facebook开源的日志采集系统。Scribe实际上是一个分布式共享队列，它可以从各种数据源上收集日志数据，然后放入它上面的共享队列中。Scribe可以接受thrift client发送过来的数据，将其放入它上面的消息队列中。然后通过消息队列将数据Push到分布式存储系统中，并且由分布式存储系统提供可靠的容错性能。如果最后的分布式存储系统crash时，

Scribe中的消息队列还可以提供容错能力，它会还日志数据写到本地磁盘中。Scribe支持持久化的消息队列，来提供日志收集系统的容错能力。

1.2.2 网络数据采集系统

通过网络爬虫和一些网站平台提供的公共API(如Twitter和新浪微博API)等方式从网站上获取数据。这样就可以将非结构化数据和半结构化数据的网页数据从网页中提取出来。并将其提取、清洗、转换成结构化的数据，将其存储为统一的本地文件数据。

目前常用的网页爬虫系统有Apache Nutch、Crawler4j、Scrapy等框架。

Apache Nutch是一个高度可扩展和可伸缩性的分布式爬虫框架。Apache通过分布式抓取网页数据，并且由Hadoop支持，通过提交MapReduce任务来抓取网页数据，并可以将网页数据存储在HDFS分布式文件系统中。Nutch可以进行分布式多任务进行爬取数据，存储和索引。由于多个机器并行做爬取任务，Nutch利用多个机器充分利用机器的计算资源和存储能力，大大提高系统爬取数据能力。

Crawler4j、Scrapy都是一个爬虫框架，提供给开发人员便利的爬虫API接口。开发人员只需要关心爬虫API接口的实现，不需要关心具体框架怎么爬取数据。Crawler4j、Scrapy框架大大降低了开发人员开发速率，开发人员可以很快的完成一个爬虫系统的开发。

1.2.3 数据库采集系统

一些企业会使用传统的关系型数据库MySQL和Oracle等来存储数据。除此之外，Redis和MongoDB这样的NoSQL数据库也常用于数据的采集。企业每时每刻产生的业务数据，以数据库一行记录形式被直接写入到数据库中。

通过数据库采集系统直接与企业业务后台服务器结合，将企业业务后台每时每刻都在产生大量的业务记录写入到数据库中，最后由特定的处理分许系统进行系统分析。

1.3 物联网数据采集技术介绍

随着物联网工业级采集器领域的逐步扩大，物的数量呈几何级增长，而物联网的信息也必然呈爆炸性增长，随之而来的访问量定然空前高涨。

架构设计：

在实时监控、数据采集等通信类系统中，通常的设计都是：将数据采集或者与底层逻辑单元（比如：底层的软件子系统、硬件终端、远程设备）通信的逻辑功能独立封装在一个子系统中，实现基础通信收发、通信方式分化、通信流程控制、底层协议规整、基础数据整合等网络通信、数据采集职责。

业务数据接口、外部系统接口、业务调度核心类、界面接口定制类、任务队列管理类、采集调度控制类、采集业务类、业务状态机类、采集方式类、通信实现类和通信协议类；

实现步骤：该方法具体包含以下步骤：

步骤一：使用者实现外部系统接口，制定通信协议、通信方式实现与外部系统信息交互；

外部系统通过外部系统接口向业务调度核心类发起通信命令、操控底层设备、实时提取设备状态；

步骤二：外部系统的命令请求通过外部系统接口转入到业务调度核心类，业务调度核心类将命令请求存入命令队列中执行；采集到业务数据后，调用业务数据接口将业务数据返回到业务调度核心类，之后，业务调度核心类调用业务数据接口或者外部系统接口将业务数据反馈到更上层类；

步骤三：任务队列管理；

采集调度控制类自动检测是否有新命令请求加入命令队列，当检测到新命令请求后立即中断通信握手执行请求，执行成功后从命令队列中删除该命令；

步骤四：采集调度控制类管理、协调其下的采集业务类、通信实现类、业务状态机类和通信协议类，完成所有的通信控制及数据采集功能；

采集调度控制类通过调用任务接口获取采集指令；之后，调用业务数据接口（业务接口由“采集业务类”实现，在具体使用中由框架使用者根据自己的业务采集需求开发），获取具体的通信指令，根据通信指令调用正确的协议接口（协议接口由“通信协议类”实现，在具体使用中由使用者根据自己的通信协议需求开发）获得通信帧，启动业务状态机类开始本次采集任务的执行；

步骤五：封装当前系统的具体采集业务对象，为通用的采集调度控制类定制具体的采集任务，把上层的抽象任务细化成具体的通信帧和通信控制步骤；

步骤六：根据采集业务状态的控制、转换需求，框架使用者定制开发，实现状态机接口，用于通信链路的通断控制、数据收发、忙闲标识及转换业务状态机类逻辑；

步骤七：采集方式类封装具体的串口、TCP/IP、语音卡通信采集类，实现具体的通信方式控制及通用的数据收发接口；

步骤八：通信协议类封装系统中软件与底层软件子系统、硬件设备、远程终端的通信协议。

核心组件介绍

1.业务数据接口

以统一的方式，输出本框架按配置的通信实现类、通信协议类、采集业务类所采集到的数据。框架使用者实现此接口的方法可以继续分析、处理、存储、展现业务数据。

2.外部系统接口

外部系统的接口，属于框架设计预留接口。框架使用者可以实现此接口，定制通信协议、通信方式实现与外部系统信息交互。外围系统通过此接口向业务调度核心类发起通信命令、操控底层设备、实时提取设备状态等业务请求。

3.业务调度核心类

采集子系统的业务调度核心类和业务请求中转站。外部系统的命令请求通过外部系统接口转入到业务调度核心类，业务调度核心类将命令请求存入命令队列中执行；采集到数据之后，调用数据接口的方法将数据返回到业务调度核心类，之后，业务调度核心类调用业务数据接口或者外部系统接口将业务数据反馈到更上层类。

4.任务队列管理类

下行任务信息缓存类，业务调度核心类向其中增加命令请求；采集调度控制器自动检测是否有新命令请求，当检测到后立即“中断”通信握手，执行请求，执行成功之后，从队列中删除该命令。

5.采集调度控制类

管理、协调其下的采集业务类、通信实现类、业务状态机类、通信协议类等模块，完成所有的通信控制及数据采集功能。通过调用任务接口获取采集指令；之后，调用业务接口（业务接口由“采集业务类”实现，在具体使用中由框架使用者根据自己的业务采集需求开发），获取具体的通信指令；根据通信指令调用正确的协议接口（协议接口由“通信协议类”实现，在具体使用中由框架使用者根据自己的通信协议需求开发）获得通信帧；最后，启动状态机开始本次采集任务的执行。

6.采集业务类

封装当前系统的具体采集业务对象，为通用的采集调度控制类定制具体的采集任务。本质就是：把上层的抽象任务细化成具体的通信帧和通信控制步骤、是一个简单的工作流定制器。

7.业务状态机类

实现状态机接口，根据采集业务状态的控制、转换需求，框架使用者定制开发，主要用于通信链路的通断控制、数据收发、忙闲标识及转换等业务状态机逻辑。

业务状态机类包括：车辆信息，工作参数信息；省市区信息；经纬度加密；数据存储。车辆信息：采集服务器定时获取车辆信息；将收到的终端数据存储到对应的轨迹表。工作参数：采集服务器定时获取工作参数，用于收到终端工作参数数据的解析。省市区数据：根据定位数据中的经纬度，解析所在的省市区数据，并存储到数据库。经纬度加密：对定位数据中的经纬度进行加密，然后再存储到数据库。数据存储：按照车辆分表，存储车辆轨迹数据。

终端数据解析后，根据位置数据中的经纬度获取对应的省市区信息，并对经纬度进行加密；将处理之后的数据持久化，并根据服务器配置定时更新车辆当前位置数据。由于终端轨迹数据量庞大，采用多线程解析数据和多线程存储数据，提高数据的存储解析效率。

8.采集方式类

封装具体的串口、TCP/IP、语音卡等通信采集类，实现具体的通信方式控制及通用的数据收发接口。

9.通信协议类

封装系统中软件与底层软件子系统、硬件设备、远程终端的通信协议。

2. Netty网络编程

2.1 Netty介绍

2.1.1 网络编程原理

IO在计算机中指Input/Output，也就是输入和输出。由于程序和运行时数据是在内存中驻留，由CPU这个超快的计算核心来执行，涉及到数据交换的地方，通常是磁盘、网络等，就需要IO接口。

比如你打开浏览器，访问某个网站，浏览器这个程序就需要通过网络IO获取网站的网页。浏览器首先会发送数据给网站服务器，告诉它我想要首页的HTML，这个动作是往外发数据，叫Output，随后网站服务器把网页发过来，这个动作是从外面接收数据，叫Input。通常，程序完成IO操作会有Input和Output两个数据流。当然也有只用一个的情况，比如，从磁盘读取文件到内存，就只有Input操作，反过来，把数据写到磁盘文件里，就只是一个Output操作。

我们回顾一下传统的HTTP服务器的原理

1. 创建一个ServerSocket，监听并绑定一个端口
2. 一系列客户端来请求这个端口
3. 服务器使用Accept，获得一个来自客户端的Socket连接对象
4. 启动一个新线程处理连接
5.
    1. 读Socket，得到字节流
    2. 解码协议，得到Http请求对象
    3. 处理Http请求，得到一个结果，封装成一个HttpResponse对象
    4. 编码协议，将结果序列化字节流
    5. 写Socket，将字节流发给客户端
6. 继续循环步骤3

HTTP服务器之所以称为**HTTP**服务器，是因为编码解码协议是HTTP协议，如果协议是Redis协议，那它就成了Redis服务器，如果协议是WebSocket，那它就成了WebSocket服务器，等等。

我们可以使用BIO、NIO、Netty等编程框架来实现自己的特定协议的服务器或进行IO网络编程。

但是使用JDK 原生NIO进行网络编程有很多问题，包括：

1）NIO 的类库和 API 繁杂，使用麻烦：你需要熟练掌握 Selector、ServerSocketChannel、SocketChannel、ByteBuffer 等。

2）需要具备其他的额外技能做铺垫：例如熟悉 Java 多线程编程，因为 NIO 编程涉及到 Reactor 模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的 NIO 程序。

3）可靠性能力补齐，开发工作量和难度都非常大：例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等等。NIO 编程的特点是功能开发相对容易，但是可靠性能力补齐工作量和难度都非常大。

4）JDK NIO 的 Bug：例如Epoll Bug，它会导致 Selector 空轮询，最终导致 CPU 100%。官方声称在 JDK 1.6 版本的 update 18 修复了该问题，但是直到 JDK 1.7 版本该问题仍旧存在，只不过该 Bug 发生概率降低了一些而已，它并没有被根本解决。

Netty 对 JDK 自带的 NIO 的 API 进行了封装，解决了上述问题。

2.1.3 Netty的主要特点

1）设计优雅：适用于各种传输类型的统一 API 阻塞和非阻塞 Socket；基于灵活且可扩展的事件模型，可以清晰地分离关注点；高度可定制的线程模型 - 单线程，一个或多个线程池；真正的无连接数据报套接字支持（自 3.1 起）。

2）使用方便：详细记录的 Javadoc，用户指南和示例；没有其他依赖项，JDK 5（Netty 3.x）或 6（Netty 4.x）就足够了。

3）高性能、吞吐量更高：延迟更低；减少资源消耗；最小化不必要的内存复制。

4）安全：完整的 SSL/TLS 和 StartTLS 支持。

5）社区活跃、不断更新：社区活跃，版本迭代周期短，发现的 Bug 可以被及时修复，同时，更多的新功能会被加入。

2.1.4 Netty 常见应用场景

1）互联网行业：在分布式系统中，各个节点之间需要远程服务调用，高性能的 RPC 框架必不可少，Netty 作为异步高性能的通信框架，往往作为基础通信组件被这些 RPC 框架使用。典型的应用有：阿里分布式服务框架 Dubbo 的 RPC 框架使用 Dubbo 协议进行节点间通信，Dubbo 协议默认使用 Netty 作为基础通信组件，用于实现各进程节点之间的内部通信。

2）游戏行业：无论是手游服务端还是大型的网络游戏，Java 语言得到了越来越广泛的应用。Netty 作为高性能的基础通信组件，它本身提供了 TCP/UDP 和 HTTP 协议栈。

非常方便定制和开发私有协议栈，账号登录服务器，地图服务器之间可以方便的通过 Netty 进行高性能的通信。

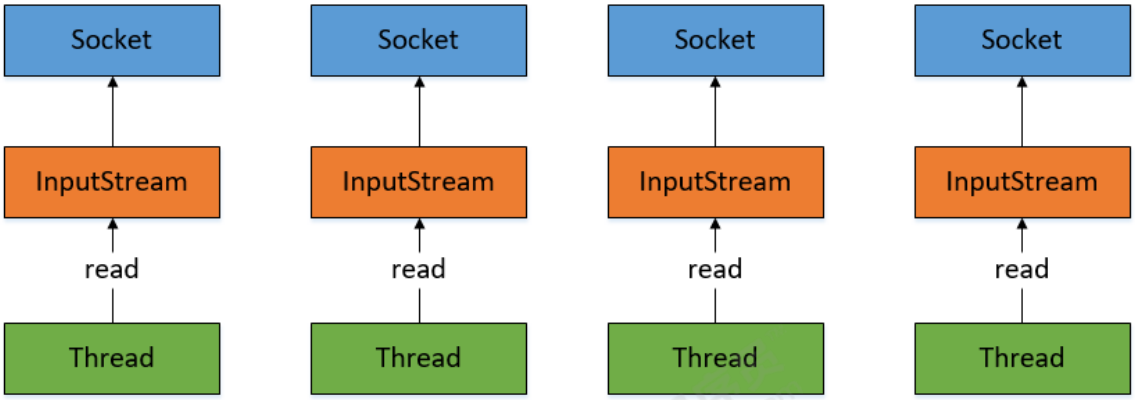
3）大数据领域：经典的 Hadoop 的高性能通信和序列化组件 Avro 的 RPC 框架，默认采用 Netty 进行跨界点通信，它的 Netty Service 基于 Netty 框架二次封装实现。

2.2 Netty架构设计

Netty 作为异步事件驱动的网络，高性能之处主要来自于其 **I/O 模型**和**线程处理模型**，前者决定如何收发数据，后者决定如何处理数据。

2.2.1 I/O模型

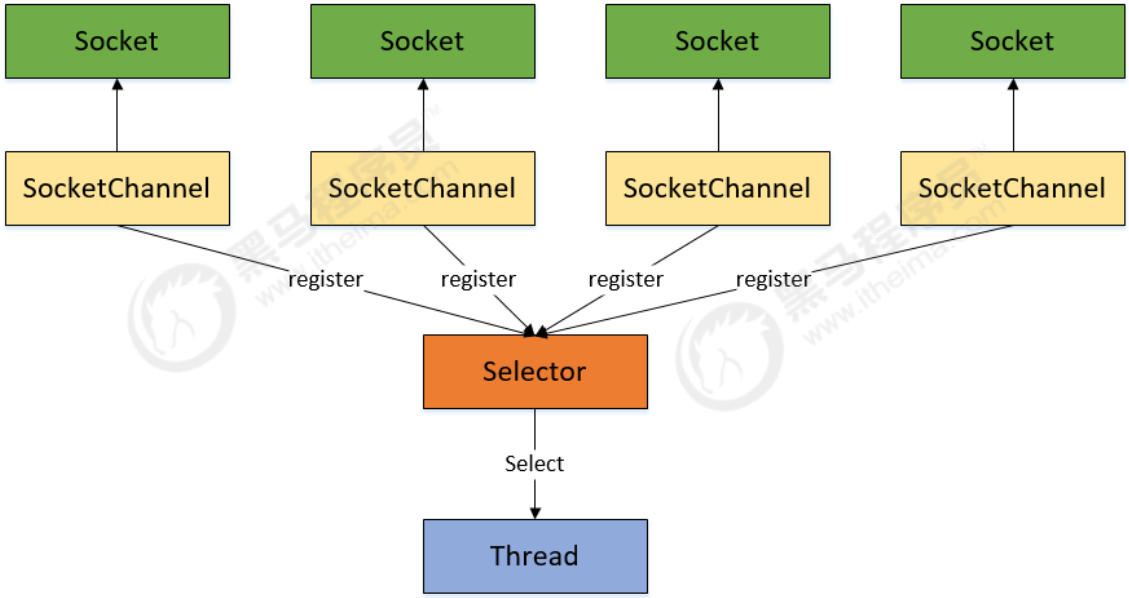
2.2.1.1 阻塞模型



特点如下：

- 每个请求都需要独立的线程完成数据 Read，业务处理，数据 Write 的完整操作问题。
- 当并发数较大时，需要创建大量线程来处理连接，系统资源占用较大。
- 连接建立后，如果当前线程暂时没有数据可读，则线程就阻塞在 Read 操作上，造成线程资源浪费。

2.2.1.2 复用模型



Netty特点：

Netty 的 IO 线程 NioEventLoop 由于聚合了多路复用器 Selector，可以同时并发处理成百上千个客户端连接。当线程从某客户端 Socket 通道进行读写数据时，若没有数据可用时，该线程可以进行其他任务。线程通常将非阻塞 IO 的空闲时间用于在其他通道上执行 IO 操作，所以单独的线程可以管理多个输入和输出通道。由于读写操作都是非阻塞的，这就可以充分提升 IO 线程的运行效率，避免由于频繁 I/O 阻塞导致的线程挂起。一个 I/O 线程可以并发处理 N 个客户端连接和读写操作，这从根本上解决了传统同步阻塞 I/O 一连接一线程模型，架构的性能、弹性伸缩能力和可靠性都得到了极大的提升。

传统的 I/O 是面向字节流或字符流的，以流式的方式顺序地从一个 Stream 中读取一个或多个字节, 因此也就不能随意改变读取指针的位置。在 NIO 中，抛弃了传统的 I/O 流，而是引入了 Channel 和 Buffer 的概念。在 NIO 中，只能从 Channel 中读取数据到 Buffer 中或将数据从 Buffer 中写入到 Channel。基于 Buffer 操作不像传统 IO 的顺序操作，NIO 中可以随意地读取任意位置的数据。

2.2.2 线程处理模型

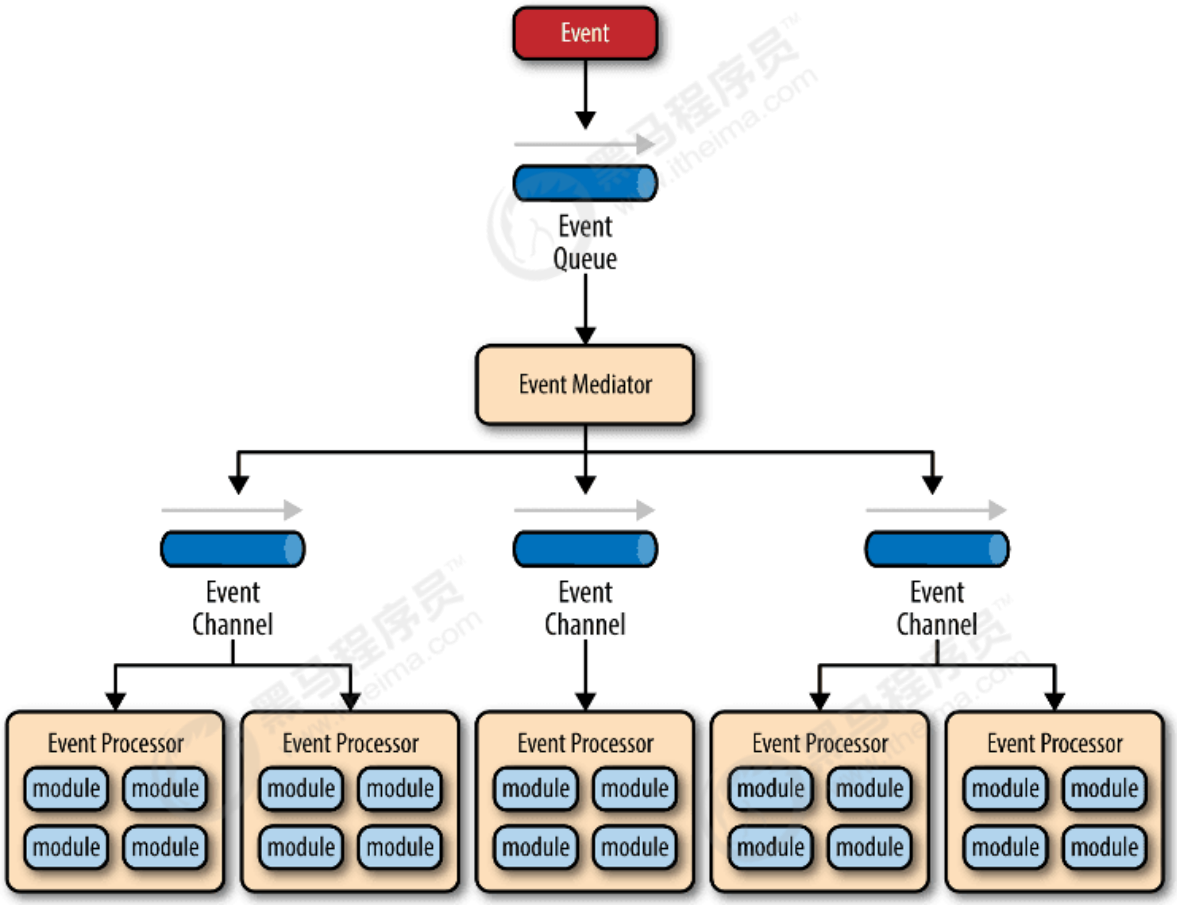
数据报如何读取？读取之后的编解码在哪个线程进行，编解码后的消息如何派发，线程模型的不同，对性能的影响也非常大。

2.2.1 事件驱动模型

通常，我们设计一个事件处理模型的程序有两种思路：

1）轮询方式：线程不断轮询访问相关事件发生源有没有发生事件，有发生事件就调用事件处理逻辑；

2）事件驱动方式：发生事件，主线程把事件放入事件队列，在另外线程不断循环消费事件列表中的事件，调用事件对应的处理逻辑处理事件。事件驱动方式也被称为消息通知方式，其实是设计模式中观察者模式的思路。



主要包括 4 个基本组件：

1）事件队列（event queue）：接收事件的入口，存储待处理事件；

2）分发器（event mediator）：将不同的事件分发到不同的逻辑处理单元；

3）事件通道（event channel）：分发器与处理器之间的联系渠道；

4）事件处理器（event processor）：实现业务逻辑，处理完成后会发出事件，触发下一步操作。

2.2.2 Reactor 线程模型

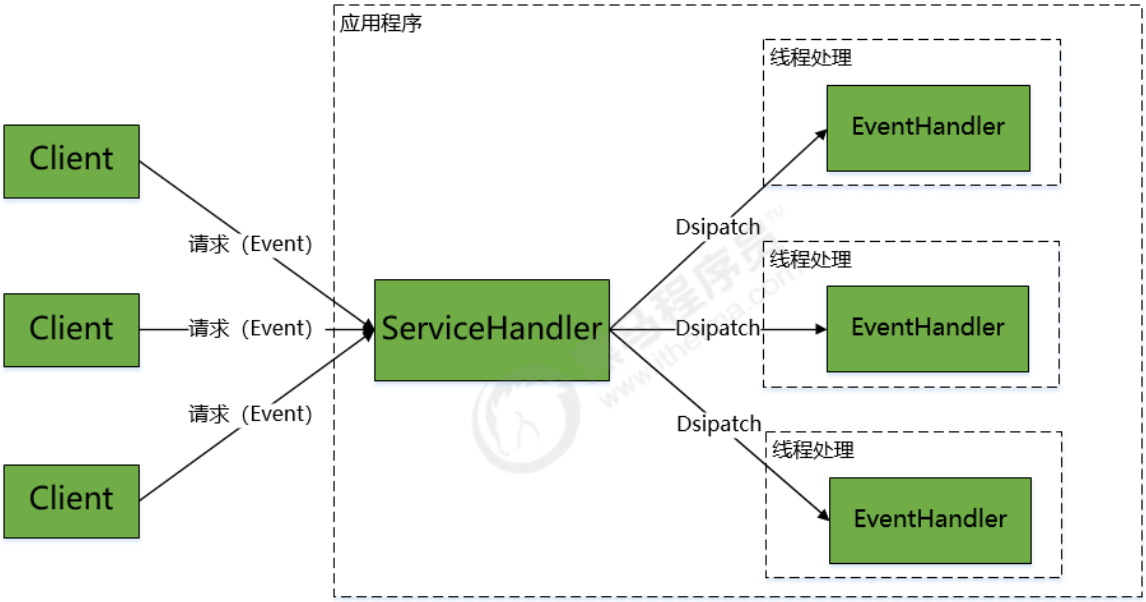
Reactor 是反应堆的意思，Reactor 模型是指通过一个或多个输入同时传递给服务处理器的事件驱动处理模式。

服务端程序处理传入多路请求，并将它们同步分派给请求对应的处理线程，Reactor 模式也叫 Dispatcher 模式，即 I/O 多了复用统一监听事件，收到事件后分发(Dispatch 给某线程)，是编写高性能网络服务器的必备技术之一。

Reactor 模型中有 2 个关键组成：

1）Reactor：Reactor 在一个单独的线程中运行，负责监听和分发事件，分发给适当的处理程序来对 IO 事件做出反应。它就像公司的电话接线员，它接听来自客户的电话并将线路转移到适当的联系人；

2）Handlers：处理程序执行 I/O 事件要完成的实际事件，类似于客户想要与之交谈的公司中的实际官员。Reactor 通过调度适当的处理程序来响应 I/O 事件，处理程序执行非阻塞操作。



取决于 Reactor 的数量和 Hanndler 线程数量的不同，Reactor 模型有 3 个变种：

1）单 Reactor 单线程；

2）单 Reactor 多线程；

3）主从 Reactor 多线程。

2.2.3 Netty 线程模型

Netty 主要基于主从 Reactors 多线程模型做了一定的修改，其中主从 Reactor 多线程模型有多个 Reactor：

1）MainReactor 负责客户端的连接请求，并将请求转交给 SubReactor；

2）SubReactor 负责相应通道的 IO 读写请求；

3）非 IO 请求（具体逻辑处理）的任务则会直接写入队列，等待 worker threads 进行处理。

img

2.3 核心类

1. **Bootstrap/ServerBootstrap**

Bootstrap 意思是引导，一个 Netty 应用通常由一个 Bootstrap 开始，主要作用是配置整个 Netty 程序，串联各个组件，Netty 中 Bootstrap 类是客户端程序的启动引导类，ServerBootstrap 是服务端启动引导类。

2. **Selector**

Netty 基于 Selector 对象实现 I/O 多路复用，通过 Selector 一个线程可以监听多个连接的 Channel 事件。

当向一个 Selector 中注册 Channel 后，Selector 内部的机制就可以自动不断地查询(Select) 这些注册的 Channel 是否有已就绪的 I/O 事件（例如可读，可写，网络连接完成等），这样程序就可以很简单地使用一个线程高效地管理多个 Channel 。

3. **Channel**

Netty 网络通信的组件，能够用于执行网络 I/O 操作。Channel 为用户提供：

1）当前网络连接的通道的状态（例如是否打开？是否已连接？）

2）网络连接的配置参数（例如接收缓冲区大小）

3）提供异步的网络 I/O 操作(如建立连接，读写，绑定端口)，异步调用意味着任何 I/O 调用都将立即返回，并且不保证在调用结束时所请求的 I/O 操作已完成。

4）调用立即返回一个 ChannelFuture 实例，通过注册监听器到 ChannelFuture 上，可以 I/O 操作成功、失败或取消时回调通知调用方。

5）支持关联 I/O 操作与对应的处理程序。

不同协议、不同的阻塞类型的连接都有不同的 Channel 类型与之对应。

常用的 Channel 类型：

- NioSocketChannel，异步的客户端 TCP Socket 连接。
- NioServerSocketChannel，异步的服务器端 TCP Socket 连接。
- NioDatagramChannel，异步的 UDP 连接。
- NioSctpChannel，异步的客户端 Sctp 连接。
- NioSctpServerChannel，异步的 Sctp 服务器端连接，这些通道涵盖了 UDP 和 TCP 网络 IO 以及文件 IO。

4. **NioEventLoop/NioEventLoopGroup**

NioEventLoop 中维护了一个线程和任务队列，支持异步提交执行任务，线程启动时会调用 NioEventLoop 的 run 方法，执行 I/O 任务和非 I/O 任务：

- I/O 任务，即 selectionKey 中 ready 的事件，如 accept、connect、read、write 等，由 processSelectedKeys 方法触发。
- 非 IO 任务，添加到 taskQueue 中的任务，如 register0、bind0 等任务，由 runAllTasks 方法触发。

两种任务的执行时间比由变量 ioRatio 控制，默认为 50，则表示允许非 IO 任务执行的时间与 IO 任务的执行时间相等。

NioEventLoopGroup，主要管理 eventLoop 的生命周期，可以理解为一个线程池，内部维护了一组线程，每个线程(NioEventLoop)负责处理多个 Channel 上的事件，而一个 Channel 只对应于一个线程。

5. **ChannelHandler**

ChannelHandler 是一个接口，处理 I/O 事件或拦截 I/O 操作，并将其转发到其 ChannelPipeline(业务处理链)中的下一个处理程序。

6. **Future、ChannelFuture**

Netty 中所有的 IO 操作都是异步的，不能立刻得知消息是否被正确处理。

但是可以过一会等它执行完成或者直接注册一个监听，具体的实现就是通过 Future 和 ChannelFutures，他们可以注册一个监听，当操作执行成功或失败时监听会自动触发注册的监听事件。

2.4 开发步骤

2.4.1. 服务端处理步骤

1. 定义线程组
2. 定义启动服务类
3. 初始化事件处理器
4. 定义事件处理器的处理逻辑
5. 绑定端口、启动服务

2.3.2. 客户端处理步骤

1. 定义线程组
2. 定义启动类
3. 初始化事件处理器
4. 定义事件处理器的处理逻辑
5. 绑定端口、启动客户端

Netty客户端和服务端的开发过程非常相似，但也有差异：

相同点：

1. 都是EventLoopGroup和NioEventLoopGroup。
2. 都有group，channel，handler（ChannelInitializer方法）

不同点：

区别点	客户端	服务端
EventLoopGroup个数	1	2
工厂类	NioSocketChannel	NioServerSocketChannel
启动类	Bootstrap	ServerBootstrap
绑定方法	connect	bind

2.4 代码示例

服务端

```
/**
 * 开启端口、接收数据
 */
public class Server {
    private int port;
    public Server(int port){
        this.port = port;
    }

    public void run() throws Exception{
        //1. 定义线程组
        //定义接收请求的线程组
        EventLoopGroup boss = new NioEventLoopGroup();
        //定义具体处理工作的线程组
        EventLoopGroup worker = new NioEventLoopGroup();

        //2. 定义启动引导类
        ServerBootstrap bootstrap = new ServerBootstrap();

        //3. 设置启动类的属性、绑定处理程序
```

```
                bootstrap.group(boss, worker)
                  .channel(NioServerSocketChannel.class)
                  .childHandler(new ChannelInitializer<SocketChannel>() {

                      @Override
                      protected void initChannel(SocketChannel socketChannel)
throws Exception {
                          //注册处理程序
                          socketChannel.pipeline().addLast(new ServerHandler());
                      }
                  })
                  .option(ChannelOption.SO_BACKLOG, 128)
                  .childOption(ChannelOption.SO_KEEPALIVE, true);

          //4. 绑定端口
          try{
              ChannelFuture f = bootstrap.bind(port).sync();
              f.channel().closeFuture().sync();
          }
          finally {
              boss.shutdownGracefully();
              worker.shutdownGracefully();
          }
      }

      public static void main(String[] args) throws Exception{
          //5. 启动服务
          new Server(10010).run();
      }
}
```

客户端

```
public class Client {
    //端口、ip
    private  int port;
    private String ip;

    public Client(String ip, int port){
        this.ip = ip;
        this.port = port;
    }

    /**
     * 客户端的连接程序
     */
    public void connect(){
        //1. 定义线程池
        EventLoopGroup worker = new NioEventLoopGroup();

        //2. 定义启动引导类
        Bootstrap bootstrap = new Bootstrap();

        //3. 配置启动类的属性
        bootstrap.group(worker)
                .channel(NioSocketChannel.class)
                .option(ChannelOption.SO_KEEPALIVE, true)
```

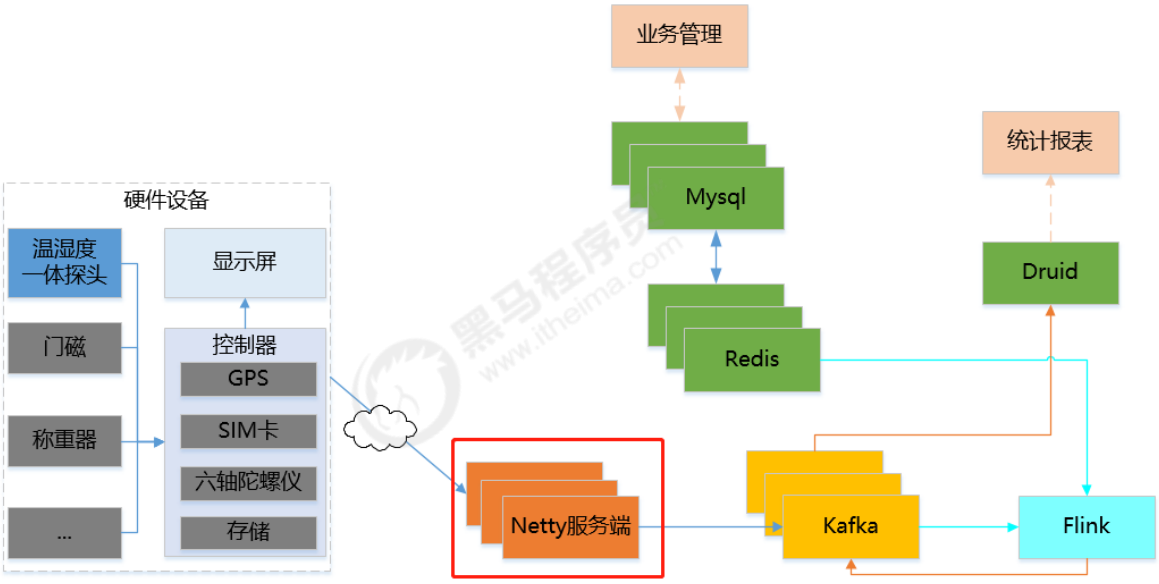
```
                .handler(new ChannelInitializer<SocketChannel>() {

                    @Override
                    protected void initChannel(SocketChannel socketChannel)
throws Exception {
                        //4. 注册处理程序
                        socketChannel.pipeline().addLast(new ClientHandler());
                    }
                });



        //5. 连接服务器
        ChannelFuture future = bootstrap.connect(ip, port);
        future.channel().closeFuture();

    }

    public static void main(String[] args) {
        //启动客户端
        new Client("127.0.0.1", 10010).connect();
    }
}
```

3 设备数据处理

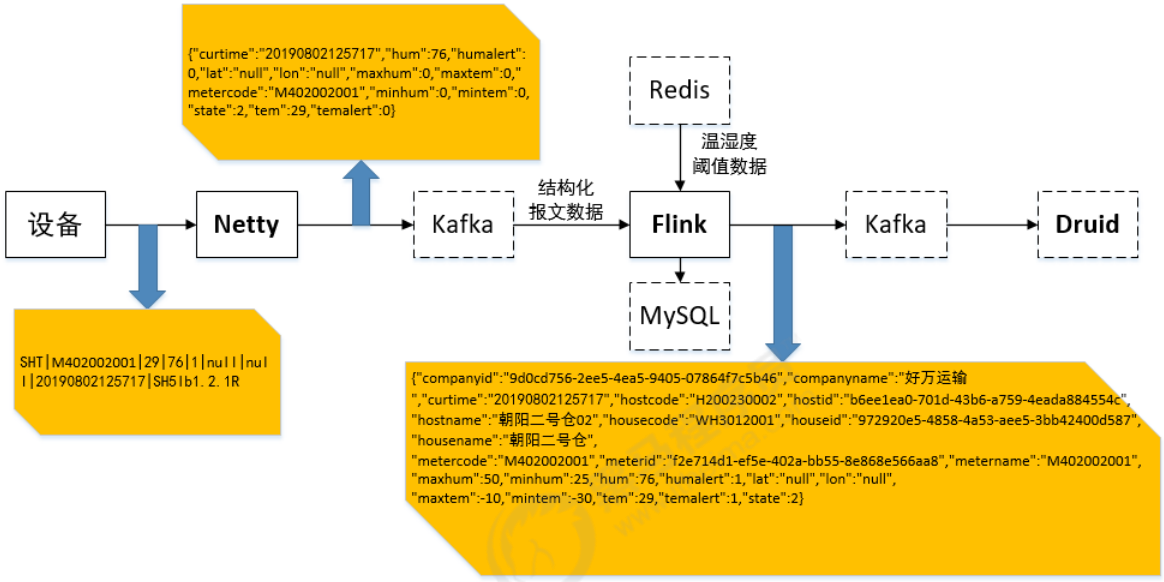
3.1 需求分析

根据业务需要，我们目前需要处理的是仪表设备发送报文消息到服务端，服务端接收到数据进行后续处理。

我们采用的是Netty网络编程技术。



报文格式转换：



3.2 服务端(cold-netty-server)

Netty服务端开发的过程主要有以下几步:

1. 首先定义两个线程组（事件循环组）

```
private EventLoopGroup bossGroup = new NioEventLoopGroup();
private EventLoopGroup workerGroup = new NioEventLoopGroup();
```

实际上一个线程组也能完成所需的功能，不过netty建议我们使用两个线程组，分别具有不同的职责。bossGroup目的是获取客户端连接，连接接收到之后再将连接转发给workerGroup去处理。

2. 定义一个轻量级的启动服务类定义一个轻量级的启动服务类

```
ServerBootstrap bootstrap = new ServerBootstrap();
bootstrap.group(bossGroup, workerGroup);
bootstrap.channel(NioServerSocketChannel.class);
```

3. 设置启动类属性

```
//3. 设置启动类的属性
        bootstrap.group(boss, worker);
        bootstrap.option(ChannelOption.SO_BACKLOG, 128);
        bootstrap.childOption(ChannelOption.SO_KEEPALIVE, true);

        // 设置通道类型
        bootstrap.channel(NioServerSocketChannel.class);
        // 设置处理器
        bootstrap.childHandler(new ChannelInitializer<SocketChannel>() {

                    @Override
                    protected void initChannel(SocketChannel socketChannel)
throws Exception {
                        socketChannel.pipeline().addLast(new
ServerHandler());
                    }
                });
```

4. 创建自定义处理器，通常继承SimpleChannelInboundHandler, 该处理器覆写channelRead方法，该方法负责请求接入，读取客户端请求，发送响应给客户端。

   以下代码演示的是接收到客户端提交的请求后，将报文转换成json对象，发送到kafka队列中。

```
package com.itheima.cold.netty.service;

import com.alibaba.fastjson.JSON;
import com.itheima.cold.netty.common.SpringContextUtils;
import com.itheima.cold.netty.entity.MessageEntity;
import io.netty.buffer.ByteBuf;
import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.ReferenceCountUtil;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import java.util.Date;

/**
 * 硬件报文处理,数据报文:
 * SHT|HE00120931|29|76|1|null|null|20190802125717|SH5lb1.2.1R
 */
public class ServerHandler extends ChannelInboundHandlerAdapter {
    private static final Logger logger =
LoggerFactory.getLogger(ServerHandler.class);

    public final static String MSG_TOPIC = "device_msg_topic";

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg){
        logger.info("ServerHandler.channelRead()");
        ByteBuf in = (ByteBuf) msg;
        try {
            //接收报文
            if(in.readableBytes()<=0){
                return;
            }
            byte[] req = new byte[in.readableBytes()];
            in.readBytes(req);
            String body = new String(req, "UTF-8");
            logger.info("报文内容:{}", body);

            //解析报文
            MessageEntity message = this.parseMessage(body);
            String jsonString = JSON.toJSONString(message);
            this.logger.info("报文解析结果:{}",jsonString);

            //发送至kafka队列
            KafkaSender sender =
(KafkaSender)SpringContextUtils.getBean("kafkaSender");
            sender.send(MSG_TOPIC, jsonString);
            logger.info("增加报文-仪表号:{}内容:{}", message.getMetercode(), body);


            //向硬件回送响应
```

```
            String responseText = "SHT|true|\0";
            ByteBuf buf = ctx.alloc().buffer();
            if (responseText.length() > 0) {
                buf.writeBytes(responseText.getBytes());
                ctx.writeAndFlush(buf);
                if (null != buf){
                    buf.resetWriterIndex();
                }
            }

        }catch(Exception e){
            e.printStackTrace();
            logger.error(e.getMessage());
        }
        finally {
            //使用完ByteBuf之后，需要主动去释放资源，否则，资源一直在内存中加载，容易造成内
存泄漏
            ReferenceCountUtil.release(msg);
        }
        if (null != in){
            //把当前的写指针 writerIndex 恢复到之前保存的 markedWriterIndex值
            in.resetWriterIndex();
        }
    }

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
        // 写一个空的buf，并刷新写出区域。完成后关闭sock channel连接。

ctx.writeAndFlush(Unpooled.EMPTY_BUFFER).addListener(ChannelFutureListener.CLOSE
);
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)  {
        // 关闭发生异常的连接
        ctx.close();
    }

    /**
     * 解析报文
     */
    private MessageEntity parseMessage(String body){
        String[] arrBody = body.split("\\|");
        logger.info("报文字段数量:{}", arrBody.length );

        MessageEntity msg = new MessageEntity();
        msg.setMetercode(arrBody[1]);
        msg.setTem(Integer.valueOf(arrBody[2]));
        msg.setHum(Integer.valueOf(arrBody[3]));
        msg.setState(Integer.valueOf(arrBody[4]));
        msg.setLon(arrBody[5]);
        msg.setLat(arrBody[6]);

        java.text.DateFormat format_date = new
java.text.SimpleDateFormat("yyyyMMddHHmmss");
        String sysdate = format_date.format(new Date());
        msg.setCurtime(sysdate);
```

```
        return msg;
    }
}
```

5. 绑定端口、启动服务

```
try{
            // 绑定端口、启动服务
            ChannelFuture f = bootstrap.bind(10010).sync();
            f.channel().closeFuture().sync();
        }
        finally {
            boss.shutdownGracefully();
            worker.shutdownGracefully();
        }
```

3.3 客户端程序(cold-jobs)

在本项目中，我们使用定时任务，定时执行netty客户端程序，该程序的业务功能是：

模拟报文消息，将消息发送到netty服务器。

客户端核心代码：

```
private void connect(final String msg) throws Exception{

        String host = YmlUtil.getValue("netty.host").toString();

        int port = Integer.parseInt(YmlUtil.getValue("netty.port").toString());

    EventLoopGroup group = new NioEventLoopGroup();

    Bootstrap bootstrap = new Bootstrap();

    bootstrap.group(group);

    bootstrap.channel(NioSocketChannel.class).option(ChannelOption.TCP_NODELAY,
true);

    bootstrap.handler(

            new ChannelInitializer<SocketChannel>() {

                @Override

                public void initChannel(SocketChannel ch) throws Exception {

                    ChannelPipeline p = ch.pipeline();

                    p.addLast("decoder", new StringDecoder());

                    p.addLast("encoder", new StringEncoder());
```

```
                    p.addLast(new ClientDeviceHandler(msg));

                }

            }

    );

    try {

        //发起异步连接操作(异步连接服务器)

        ChannelFuture channelFuture = bootstrap.connect(host, port).sync();

        channel = channelFuture.channel();

        //异步等待关闭连接channel

        channel.closeFuture().sync();

    } catch (InterruptedException e) {

        logger.error(e.getMessage());

        e.printStackTrace();

    }finally{

        group.shutdownGracefully(); //释放线程池资源

    }

}
```

ClientDeviceHandler是具体处理的类，继承了ChannelInboundHandlerAdapter：

```
package com.itcast.cold.jobs.service;

import io.netty.buffer.Unpooled;
import io.netty.channel.ChannelFutureListener;
import io.netty.channel.ChannelHandlerContext;
import io.netty.channel.ChannelInboundHandlerAdapter;
import io.netty.util.CharsetUtil;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

//模拟硬件:发送硬件报文
//SHT|10666730220|00|aaaaaa|2|8|35|75|50|875|100|10000000|55085|26|34101|460|0|2
0190408000000|IMEI|864121016526443|SH5lb1.2.1R|

public class ClientDeviceHandler extends ChannelInboundHandlerAdapter {
    private static final Logger logger =
LoggerFactory.getLogger(ClientDeviceHandler.class);
    private String msg;
```

```
public ClientDeviceHandler(final String msg){
    this.msg=msg;
}

@Override
public void channelActive(ChannelHandlerContext ctx){
    //logger.info("ClientDeviceHandler.channelActive()");
    //String msg =
"SHT|10666730220|00|aaaaaa|2|8|35|75|50|875|100|10000000|55085|26|34101|460|0|20
190510123000|IMEI|864121016526443|SH5lb1.2.1R|";
    ctx.writeAndFlush(Unpooled.copiedBuffer(msg, CharsetUtil.UTF_8));
}

@Override
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    //logger.info("ClientDeviceHandler.channelRead()");
    this.logger.info("平台响应:{}",msg);
}

@Override
public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
    //logger.info("ClientDeviceHandler.channelReadComplete()");
    // 写一个空buf，并刷新,完成后关闭sock channel连接。

ctx.writeAndFlush(Unpooled.EMPTY_BUFFER).addListener(ChannelFutureListener.CLOSE
);
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception {
    //logger.info("ClientDeviceHandler.exceptionCaught()");
    cause.printStackTrace();
    ctx.close(); // 关闭发生异常的连接
}

}
```

4、总结

通过今天的学习，相信大家对ETL、大数据采集、物联网数据采集等业务和技术有了一定的认识。

同时，应该能够使用netty进行网络编程。