

首先考虑一个Java服务器实际的需求，然后做开发

\*开发的过程，穿插设计模式和一般的开发之中的运用

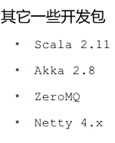
\*对于Java常用的包Spring , Netty, AKKA

引入的目的，服务器端的程序设计和普通Java程序的不一样的地方。就是为了针对**性能**的要求，**系统容错性**的要求以及对于**负载均衡**的要求 【**性能 系统容错 和 负载均衡 三个要求**】



Scala IDE和Eclipse没有区别 就是多了对Scala的支持

最终写出来的程序**一定能在生产服务器运行**

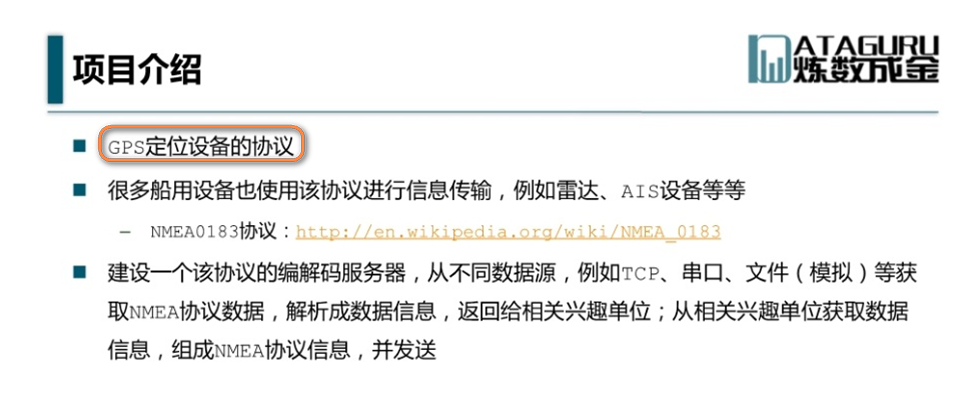


这么课用到了Scala IDE 但是 还是使用Java进行开发，之所以用到Scala就是Akka本身是Scala开发的包

Java中可以调用Akka ---- 最终所有的开发都在java中完成

这些包的下载等到讲解具体内容的时候再去完成

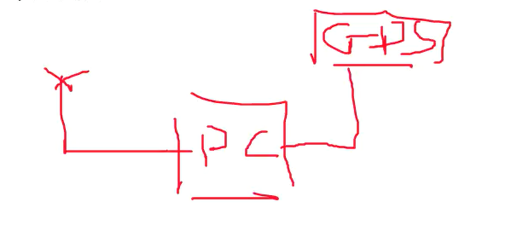
Akka本身是Scala2.1开发的 所以 使用Scala的IDE



另外一个雷达严重问题：波长比较短 造成和探测目标之间有障碍物会挡住雷达的发射波

因为雷达是被动探测的，所以能探测的时间 什么样的时间 多远的距离有一个回波

所以可以使用AIS 就是**一个无线电收发设备 + GPS**



通过PC输入当前运载的货物 吃水 通过GPS+手工输入的信号 告诉周围的传 我在哪 我的船名 呼号等等

AIS设备有一个明显的信号增强功能 ------ 只要有AIS设备 只要你发送 我就能收到

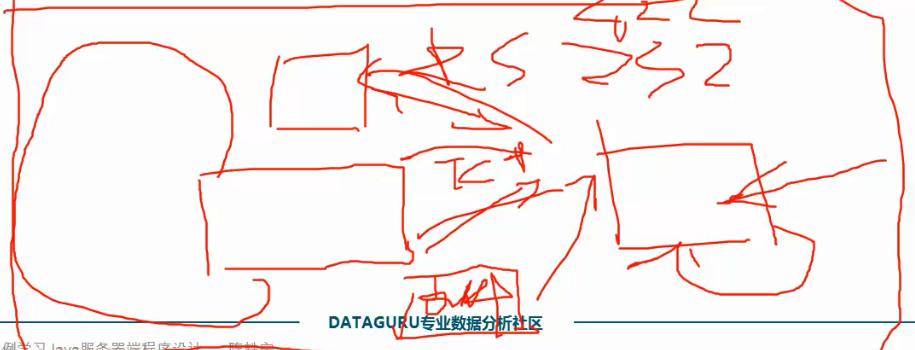
AIS发射的波长相对比较长

现在知道两条船 中间有一个岛屿 ----对于雷达 无法绕过 但是 AIS可以绕射的功能

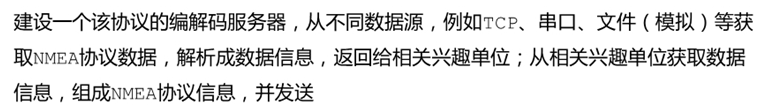
AIS发射波的波长比较长 所以 受到天气的影响也不大

----- 要做的东西 就是：

**现在做的就是通过多个数据源** 来**接受NMEA0183协议的信号** ---- 然后**去解析它** ---- 解析出航速 航向 等等给第三方使用



同时第三方可能会发一个短信过来 我这边组成文本信息之后 在发送出去

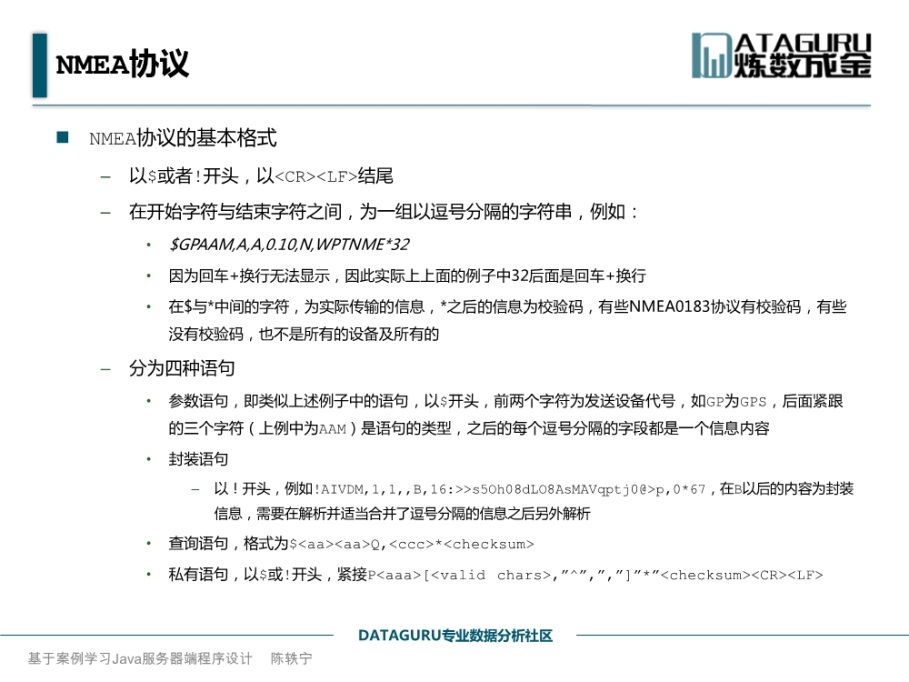
【实际上就是数据的解析和发送】

系统外扩：

传播的经度纬度信息可以入库 --- -mongo可以分析空间关系

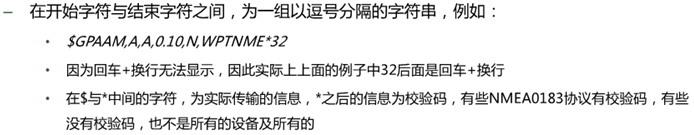
也可以使用Hadoop Spark库等等来进行分析

但是我们这里不考虑入库 ---- 仅仅考虑信息的编码【要做的就是**编解码服务器**】



 结尾的是 回车 十六进制就是0D和换行 0A ----- 每一个都是$开头 0D0A结尾

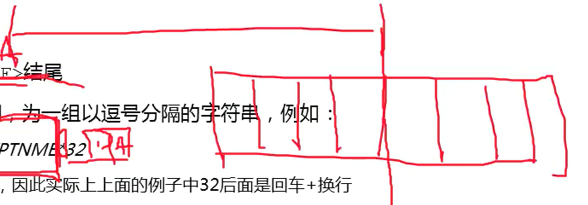
在开始字符和结尾之间 有一个逗号分隔的字符串

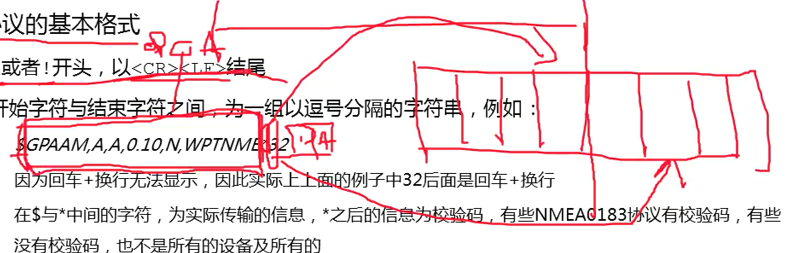


例子中的32实际上有两个字符无法显示 就是回车换行

\*\*校验码是对$和\*之间所有字符按照比特位 bit 进行异或

异或到最后可能是可打印或者不可打印的8 bit字符 ----- 每一个字符拆分成两个4bit的





有校验码的要求去判断一下

不带校验码的 用不着去验证

NMEA的四种语句



前两个字符AI表示发送设备 VDM表示发送设备的语句

封装语句所不同的 普通的语句都是一个字符字符解析出来的

但是 封装语句 是经过一定的内容编码的 --- 会把一堆信息进行编码 --- 编码之后 **每6 bit分一个组** ---- **6 bit对应一个整数 可能是0 1234等**

封装语句真正需要解析的内容 不是那么明确

第三类是查询语句



封装语句 参数语句都不会以Q结尾

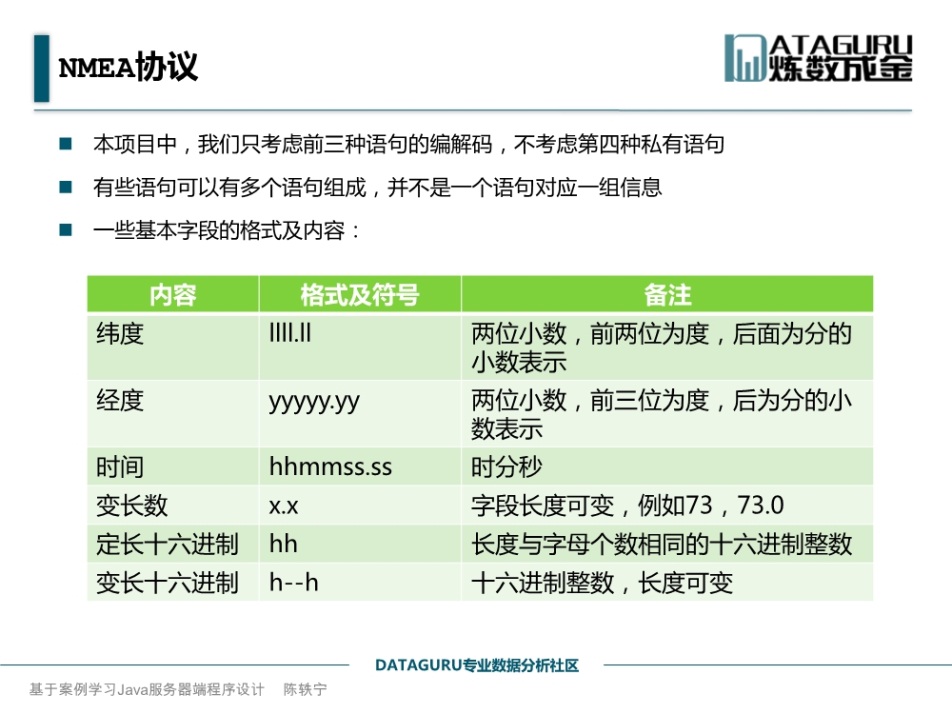
【这几个语句可以后面再听】

最后一个私有语句 不用管太多

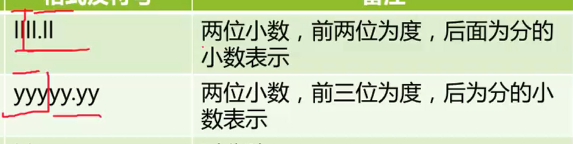


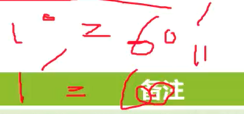
对于项目来说 不关心私有语句 ---- 就是每一个公司对于特定的产品都有自己的私有语句

要发一个很长的文字 需要很多句语句来组成 这个就是要解析的基本的内容









1度=60分

1分=60s

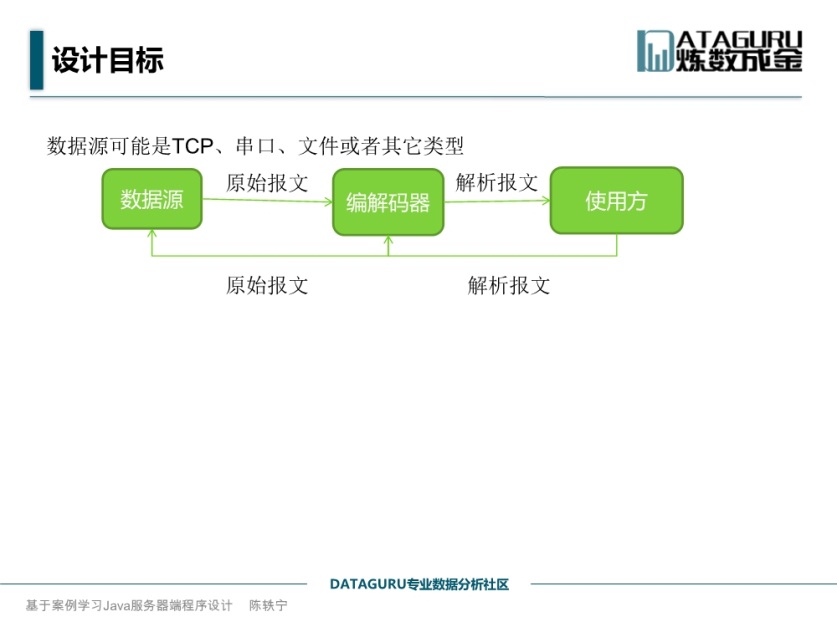
还有一个尝试 就是 1分是1海里



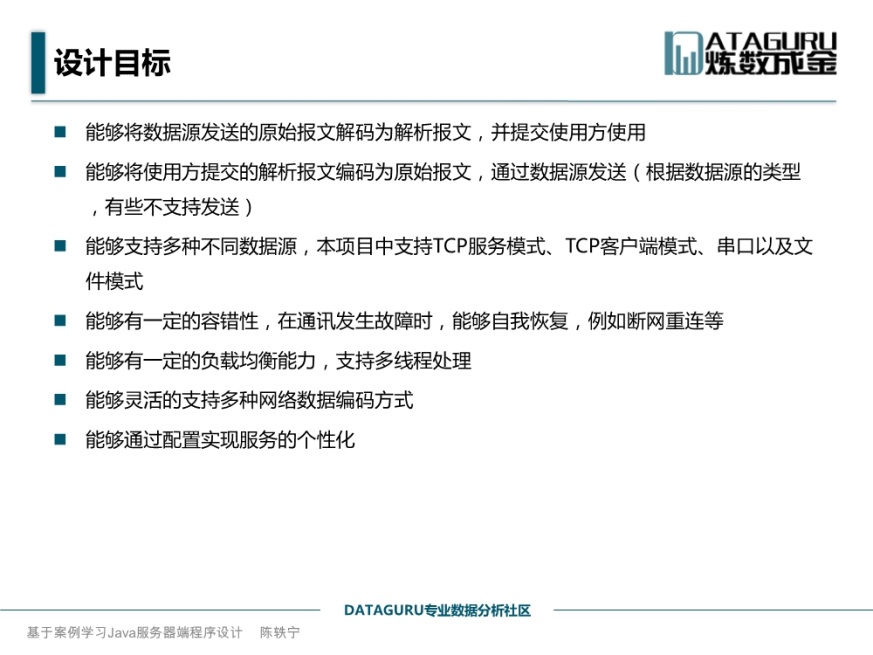


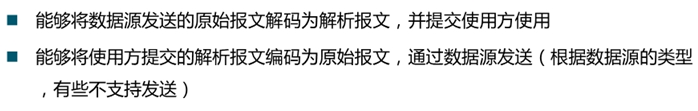


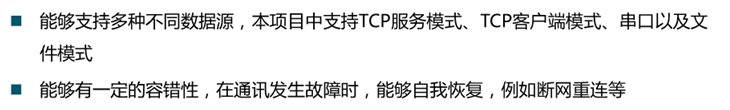
下面是设计目标



原始报文是上面的语句构成的 --- 解析报文是原始报文解析出来的内容



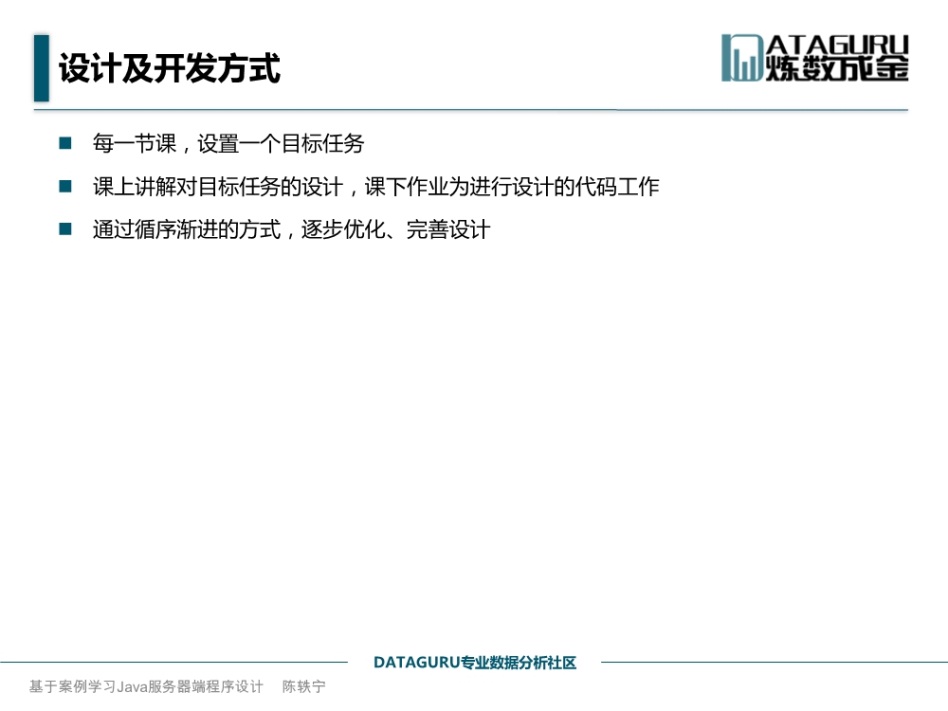
 ---- 编码和解码都有了 双向 ----就是上面的环【图】

 --- 串口很难返回错误信息 打开之后 永远可以往里面写 不管这个串口是否坏掉了

还要有负载均衡能力

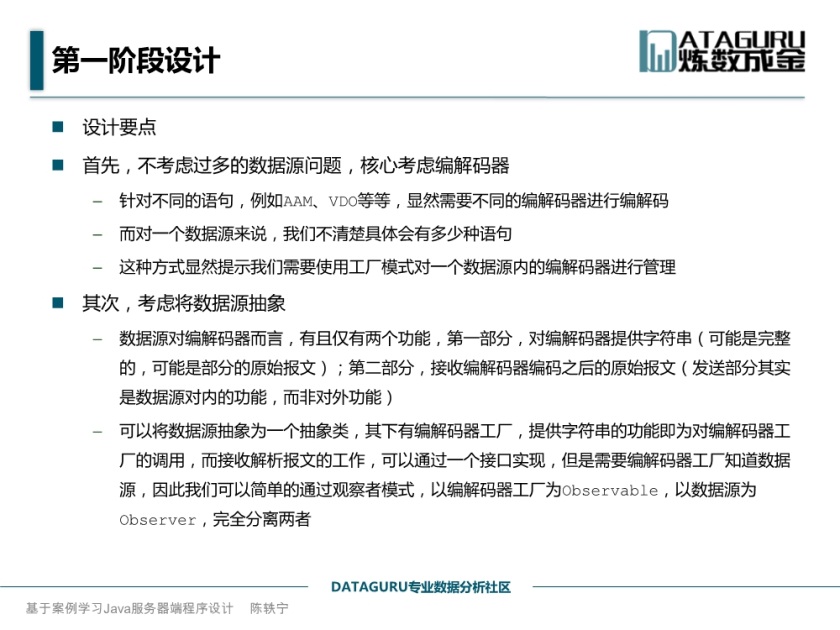


 比如可能需要更高效的编码方式



每一课的作业把课上说的内容 转化成代码

**提交的内容是源代码 + 单元测试 + 以及对这一课的理解**



第一节课做一个**简单的编解码器** 不要考虑数据源 

处理**参数语句 封装语句** 并且**能够处理查询语句**

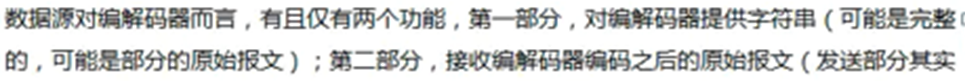
所谓的**处理 在这里就是 编码和解码**

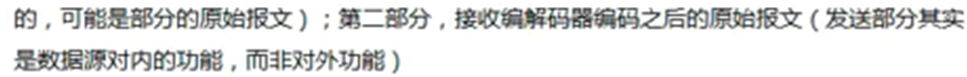
通过单元测试验证

初始设计：

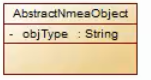
想一下里面的关系：

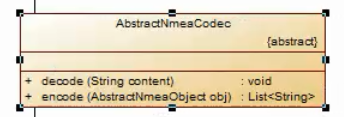
数据源对于编解码器

 --- TCP把包一个来的 不能保证每一个frame过来的是完成的信息



定义解析报文的基类

 抽象的AbstractNmesObject【----就是代表每一种数据源的数据实体】

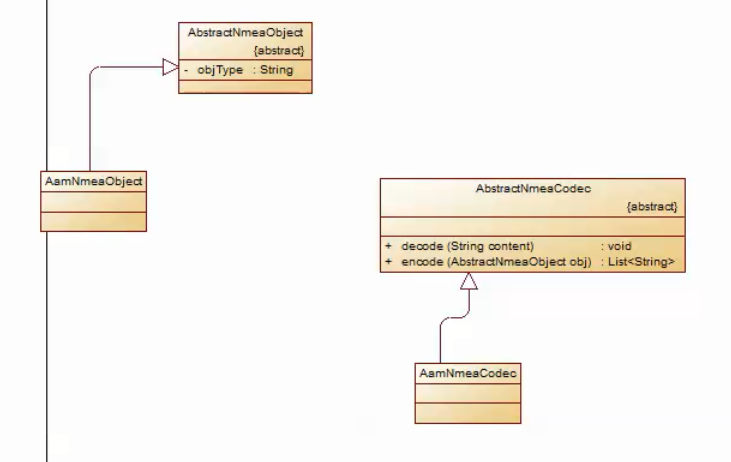


编解码器来说 应该支持两个方法 一个decode 通过String去解码

一个encode 给一个AbstractNmesObject 就应该把里面的信息编码成字符串数组

每个对象不一定是一个语句 所以范湖的是一个List<String>

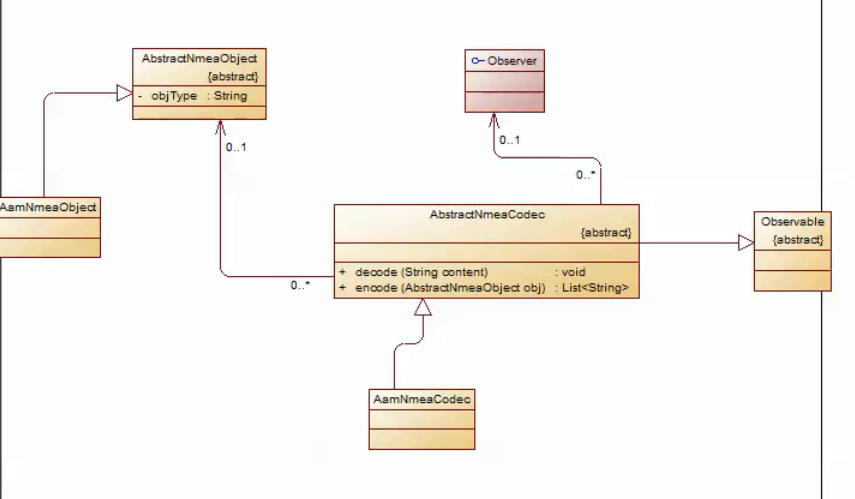
现在对AbstractNmesObject对象进行扩展



这里面decode为什么没有返回一个？ 可以返回一个 后面返回一个 还不如

现在把AbstractNmesCodec变成Observable ----- 然后给Observer去观察

OO模式中 观察者模式起的作用是一个回调作用 就是不知道被观察的对象何时产生预期的结果 ---- 比如**多线程环境中 四五条线程中任何几条出结果** --- 不知道什么时候 ---- 最好**使用观察者模式 ----- 就是对这个东西感兴趣 你什么时候扔给我**



----- 老师说到了一个非常关键的地方 就是 沿着箭头 有访问权 

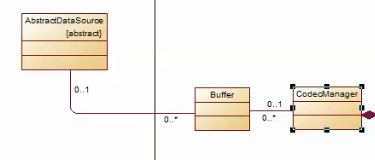
【这样Observable ---- AbstractNmesCodec就知道有哪些观察者 ---- 也就是要注册观察者 --- 因为箭头的方向 是从Observable指向了Observer 这样 实际上就是单向关联 含以上就是Observable可以看到Observer】

这样分离了很多语句的解析

每个语句的解析 就不相关了 ---- 对测试和协同开发都有好处

我们需要一个Buffer ---- TCP不能保障每一个语句是完整的 【所以可能要连续不断写 写出一个完整的语句】可能是一个片段 很长的东西 ---- 每次未必收到的是完整的内容 可能收到任何东西 --- 需要一个buffer 这样所有的abstract data source 都要把数据写到buffer --- 显然每一个具体的data source对应一个写到一个具体的buffer中（混到一起不合适） ---- buffer的任务 就是攒成一个完整的语句 扔给AbstractNmesCodec

\*\*--- 还有一步要做 ---- 需要CodecManager ---- buffer组成语句 但是 不知道是什么语句 ---- 尤其是你的语句来源于网络或者文件的时候 你都不知道这个语句到底是什么 ----- 所以需要一个类似于Dispatcher的东西进行分发

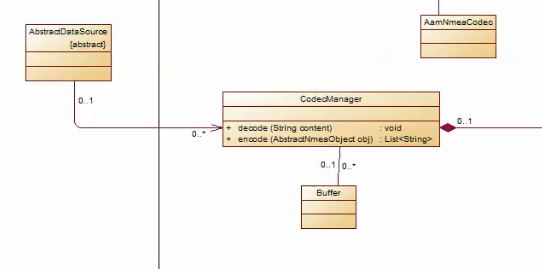


CodecManager就是做这个工作的

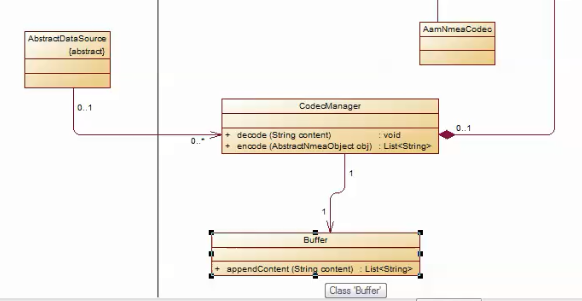
---- 来了一个语句 可能是aam 需要调用aam的codec --- 如果vdo的 那就需要调用vdo的codec

实际上这个时候 做的是分解的工作

---- 第二个就是类似于工厂模式 ---- 就是对于不同的语句进行分类 ---找到不同的codec ----所以 这个Manager实际上就是Codec的工厂【Manager有时候就是工厂 --- 这个很有意思 ---- 这时候的Manger最简单的就是分发】

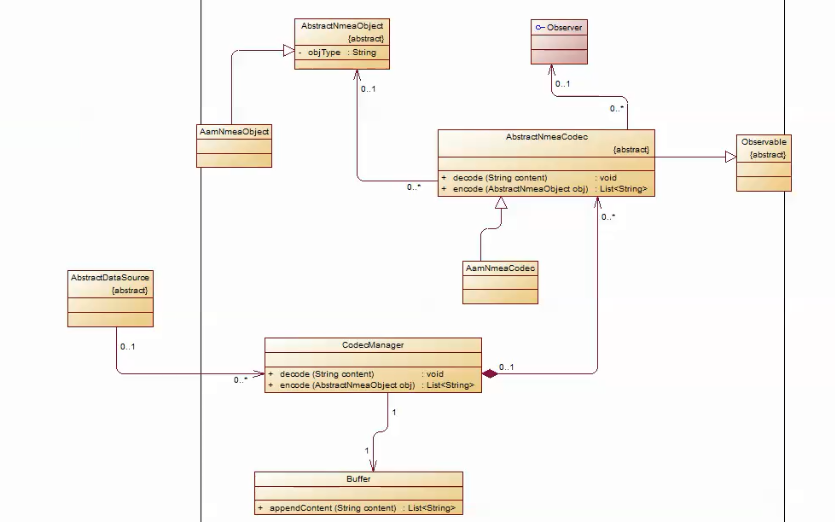


实际上这个decode就是给buffer添加了一个appendContent

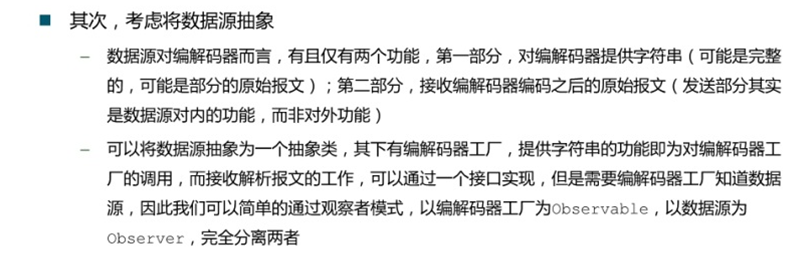


Content可能是完整的 也可能是不完整的 但是返回的List<String>中的每一个String都是完整的

----如何创建工厂 留给大家做



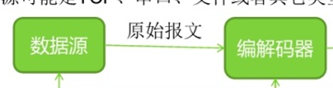
============== 自行分析 ===========



【

\*\*\*数据源对编码器



 这块是数据源对codec ----- ppt上说的 提供字符串给编解码器（完整或者部分的语句的原始报文）

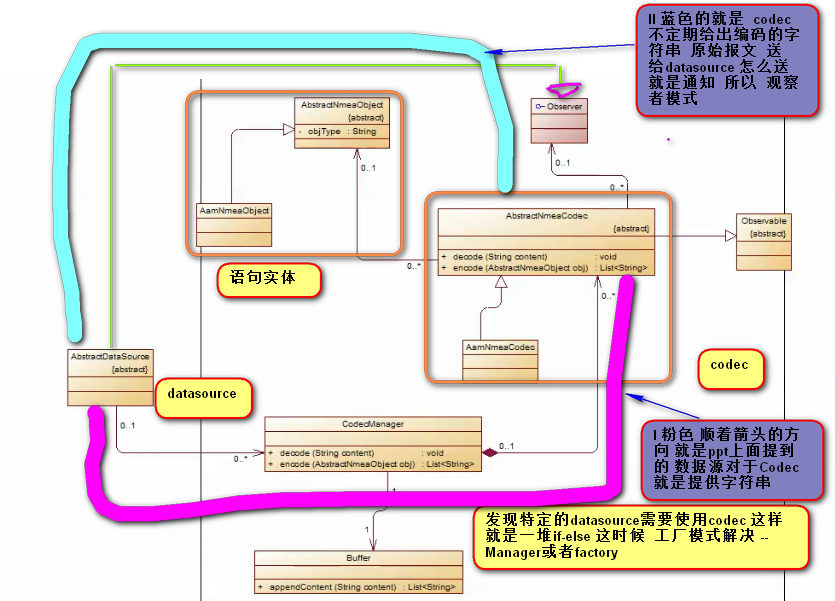
===== 对不同的datasource 需要不同的codec进行处理

第二部分  接收编解码的原始报文

----- 按照这个图 OOP 就有datasource, codec 和User这三个对象

====== codec不定期给datasource提供数据 这样数据源接收数据 ---- 这句话就是ppt上说的 codec factory需要知道DataSource ----- 这样codec工厂就是被观察的对象 Observable 里面注册了Observer --- 自身有这样的成员 不就是知道的含义么 ---- 所以 Observer就是DataSource ---- 一旦codec 编码生成了最后的原始报文 就通知所有的观察者DataSource 然后把生成的数据推送给他们

所以 上面的图 分解

】