IO:

用户线程与内核线程中间有一个buffer,这个buffer既可以是用户态,也可以是内核态;

阻塞与非阻塞(IO请求:将数据写入到buffer或者从buffe中读取数据,即用户线程的Read/Write函数):当用户线程从buffer中读取数据,如果buffer中没有数据,读会阻塞直到有数据,往buffer中写入数据,如果buffer已满,写操作也会阻塞;

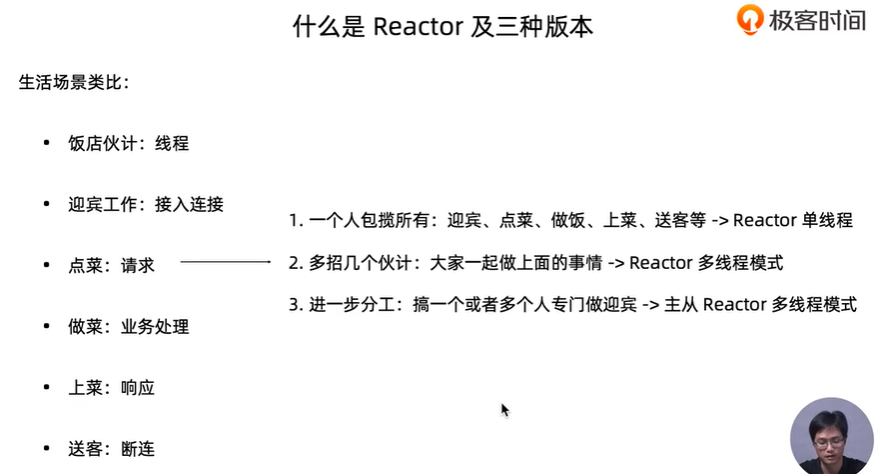
异步与同步:

同步: 当内核中的数据就绪后,需要用户线程自己去内核中读取;

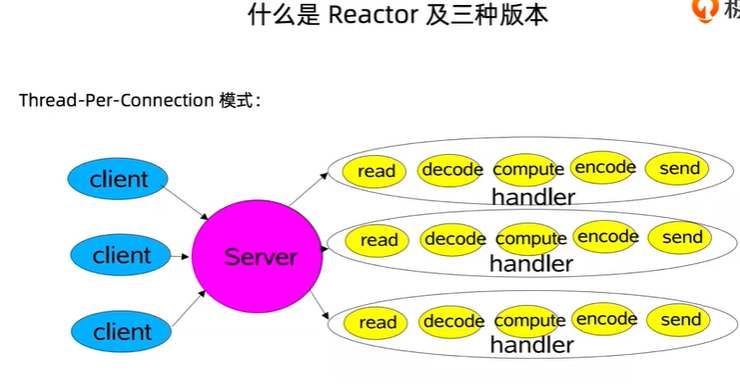
异步: [用户态buffer]数据就绪后直接由OS回调给用户进程是异步;

NIO(同步非阻塞IO): netty

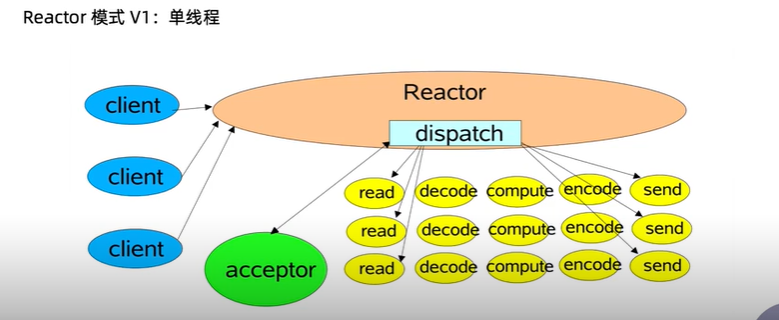
Reactor模式:

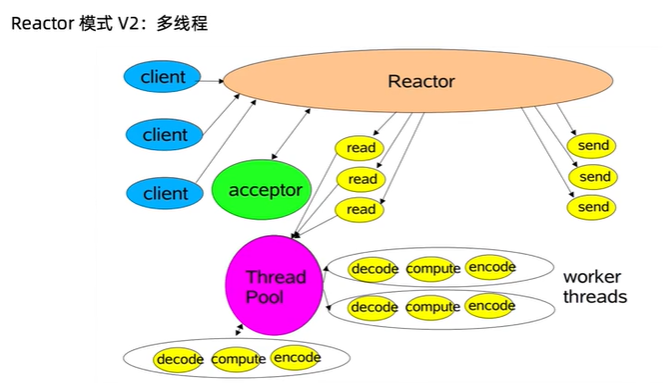


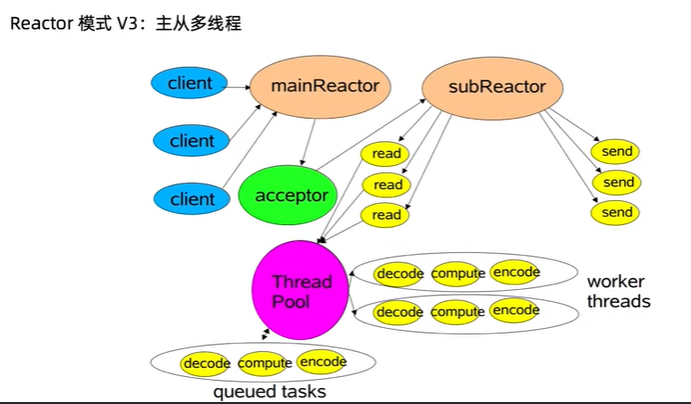




Handler[一个线程]负责数据的处理, 可以多个Handler分工协作,一个Handler处理一个具体的工作,多个Handler组成一个完成的处理链路,完成整个处理工作;









**TCP粘包和半包**

粘包:数据量太小,

半包: 数据量太大,需要拆包,分多次传输;

粘包的主要原因: 发送方每次写入数据<套接字缓冲区大小;(网卡一般不会立马发送)

接收方读取套接字缓冲区数据不够及时;

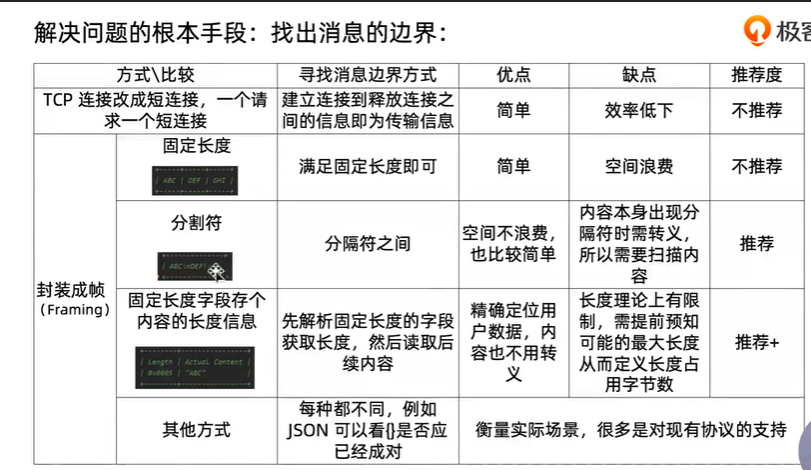
半包的主要原因: 发送方写入数据 > 套接字缓冲区大小;

发送的数据大于协议的MTU(Maximum Transmission Unit: 最大传输单元),必须拆包;

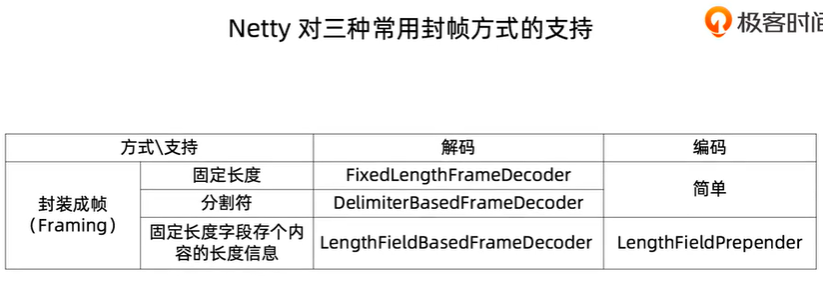
根本原因:TCP是流式协议,消息无边界;UDP像邮寄的包裹,虽然一次传输多个,但是包裹都有”界限”,一个一个接受,所以无粘包,半包问题;

从收发角度看粘包和半包: 一个发送可能被多次接收,多个发送可能被一次接收;

从传输角度看粘包和半包: 一个发送可能包含多个传输包,多个发送可能组成一个传输包;



**粘包和半包对应编解码**



常用的”二次”编解码方式: java序列化, Marshaling, XML, JSON,MessagePack, Protobuf…

一次解码的结果是字节,需要和项目中所使用的的对象做转化以方便使用,这层解码器可以称为”二次解码器”,相应的,对应的编码器是为了将java对象转化成字节流**方便存储和传输**;

一次解码器: ByteToMessageDecoder

io.netty.buffer.ByteBuf(原始数据流) -> io.netty.buffer.ByteBuf(用户数据)

二次解码器: MessageToMessageDecoder<I>

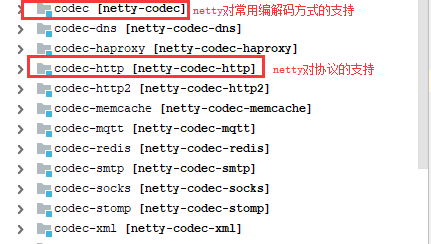
io.netty.buffer.ByteBuf(用户数据) -> java object

选择编解码方式的要点: 编码后占用的空间(IO), 编解码速度(CPU),可读性,多语言支持

Java对象序列化: 将java对象转化成字节数组;

java反序列化(二次解码): 将字节数组转化为java对象;

Netty源码:



Keepalive与Idle监控:

Keepalive: 给对方发送信息确保其在线,如果不在线就断开连接;

TCP keepalive三个核心参数: 多久未收到数据包, 发送探测包的间隙, 发送几次探测包

传输层关注是否”通”,应用层关注是否可服务,类比前面的电话订餐例子,电话能通,不代表有人接;服务器连接在,但是不一定可以提供服务(例如服务不过来等)

TCP层的keepalive默认关闭,且经过路由等中转设备keepalive包可能会被丢弃;

HTTP Keep-Alive: 指的是对长连接和短连接的选择, HTTP/1.1默认长连接,不需要带这个header;

Idle监控(配合Keepalive使用): 有其他数据传输的时候,不发送keepalive,无数据传输超过一定时间,判断为Idle,再发keepalive;

同步问题的核心三要素: 原子性, 可见性(Volatile),有序性(重排序);

内存使用:

目标: 内存占用更少(空间),响应速度快(时间)

对Java而言: 减少Full GC的STW(stop the world)时间;

Netty内存使用技巧-堆外内存

JVM外部: 堆外(off heap) ->受OS管理;但是JVM有对其的引用;

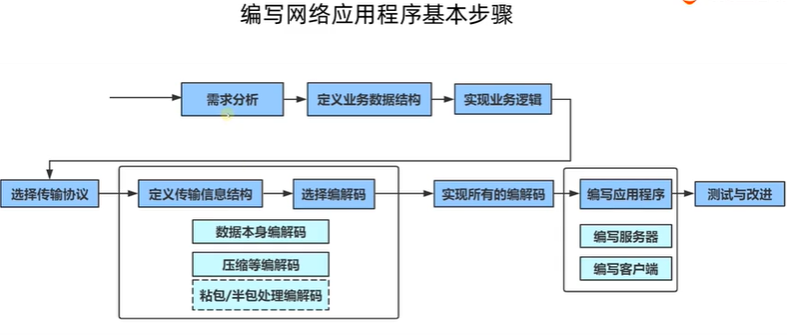
JVM内部: 堆(heap) + 非堆(non heap)

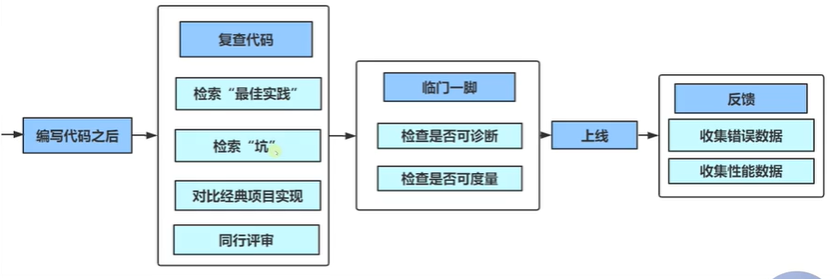
JDK自身的Zero-copy接口;

Netty内存使用技巧-内存池(对象池)

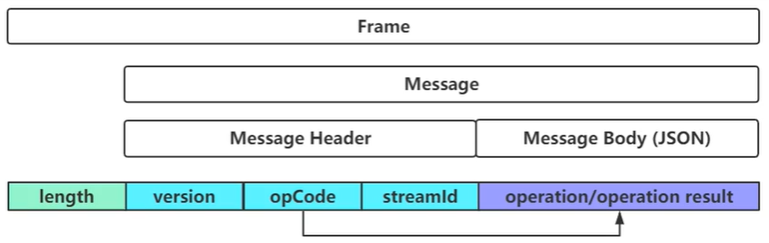
如何实现对象池: 开源实现(Apache Commons Pool), Netty轻量级对象池实现: io.netty.util.Recycler;

堆内,堆外可以和池化,非池化自由组合;





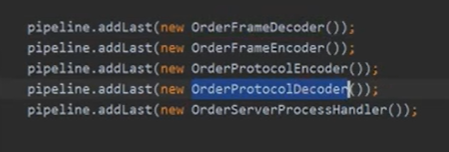
**Message的定义 \*\*\*\*\***



响应分发:

编解码处理器的顺序问题:

一次解码 -> 二次解码 -> 业务处理 -> 二次编码 ->一次编码



**@ChannelHandler.Sharable:** (SocketChannel)每一个连接都有一个ChannelPipeline, ChannelPipeline中的Handle实例既可以是多个ChannelPipeline共享的,也可以是每个ChannelPipeline对应一个Handle实例; 是否需要共享取决于是否存在并发问题,如果不确定可以不共享,也就是浪费了点资源;

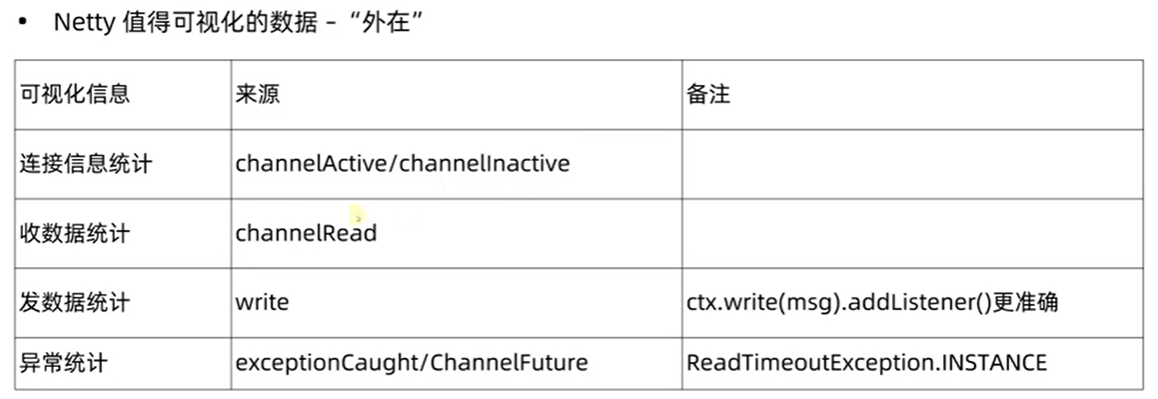
Netty支持的系统参数:(option:参数设置)

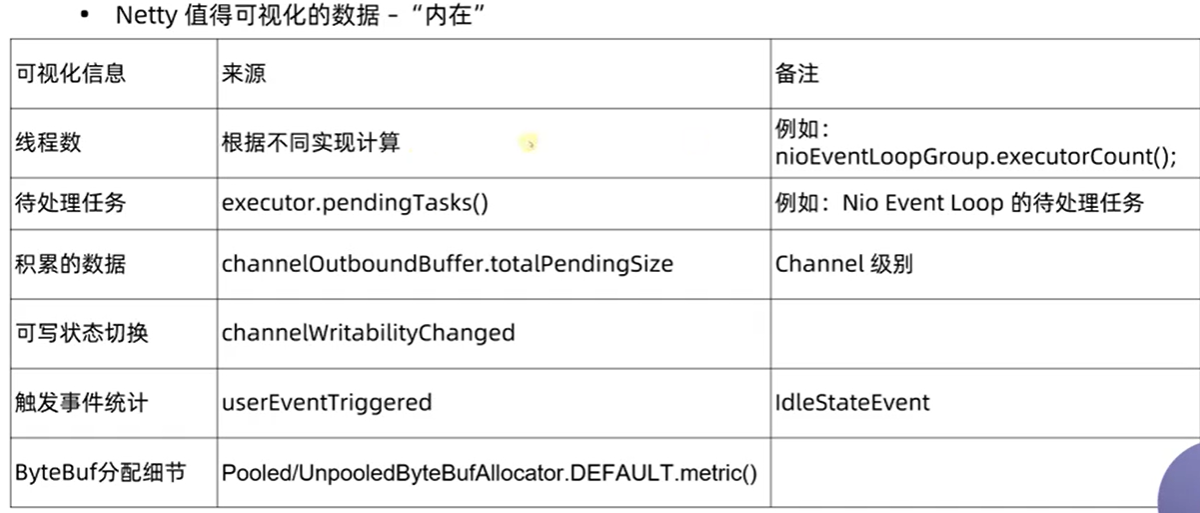
ServerSocketChannel ->.option

socketChannel ->.childOption

Linux系统参数: 进行TCP连接时,系统为每个TCP连接创建一个socket句柄,也就是一个文件句柄,但是Linux对每个进程打开的文件句柄数量做了限制,如果超出: 报错”Too mang open file”;

JMX（Java Management Extensions，即Java管理扩展）是一个为应用程序、设备、系统等[植入](https://baike.baidu.com/item/%E6%A4%8D%E5%85%A5/7958584" \t "_blank)管理功能的框架。





**Netty内存泄漏: -> byteBuf未释放**

**(客户端)Keepalive保持不被断开连接, (服务端)监控idle断开空闲连接;**

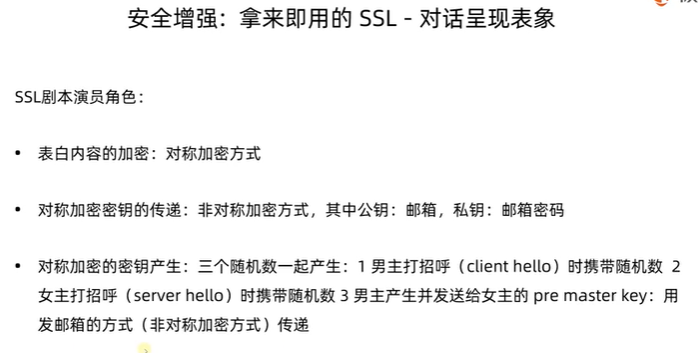
**@ChannelHandler.Sharable:** 如果Handle中没有并发修改共享资源就可以使用这个注解;

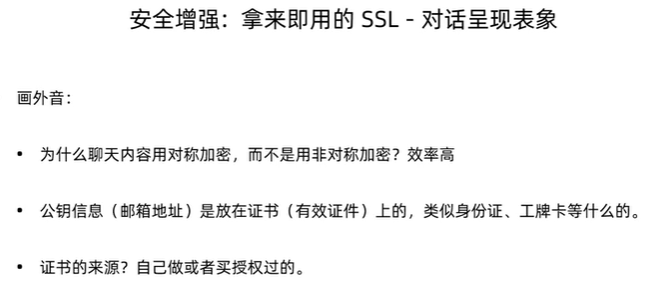
**Handler: 注意排序,从后往前执行;**

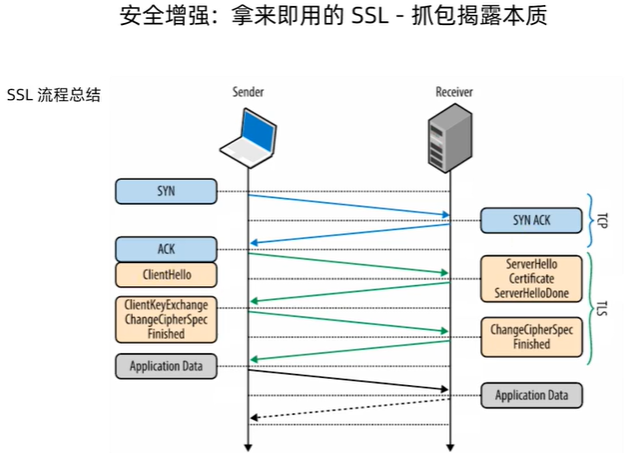
安全增强: 黑白名单(cidrprefix:此参数代表的是网络号),自定义授权, 拿来即用的SSL数据加密;

认证和授权是所有系统都需要的;

SSL/TLS:







**TCP三次握手之后才是SSL/TLS**

**生成对称秘钥: premaster Secret(客户端的预主秘钥) + 客户端随机数 + 服务器随机数;**

**客户端premaster Secret会通过公钥加密传输到服务端;**

**Selector: 多路复/分用器;**

**Cassandra: 面向列存储的NOSQL数据库;**