# 并发编程：

## LongAdder（java.util.concurrent.atomic.LongAccumulator）

实现思路：出现冲突时，将数值拆分为一个个cell，直到没有冲突为止。

## ConcurrentLinkedQueue

CAS算法实现的非阻塞队列（基于链表实现的单向FIFO队列）。区别于ConcurrentLinkedDeque(双端队列)

## CopyOnWriteArrayList

写时复制，读无需同步。具体通过使用volatile 保证数组内容的可见性，那么并发读时就不会发生线程安全问题，因此读时不需要同步操作，因此在CopyOnWriteArrayList底层实现维护了一个volatile 修饰的Object类型数组，在写时，复制一份数据然后加锁（ReentrantLock）进行操作，但是复制操作开销较大，因此适合迭代（读）操作多的情景。比如分发通知时需迭代已经注册了该通知监听器的操作类。（mir实现：拍卖物品记录保存）

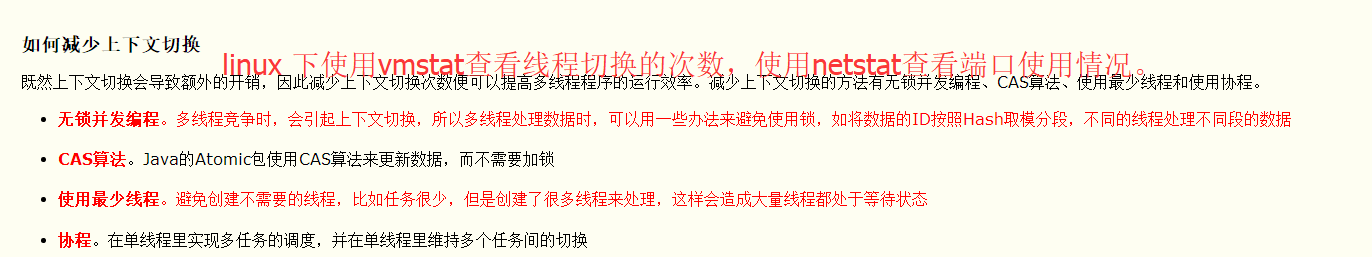
Ps: 1、允许null值。

## ConcurrentHashMap

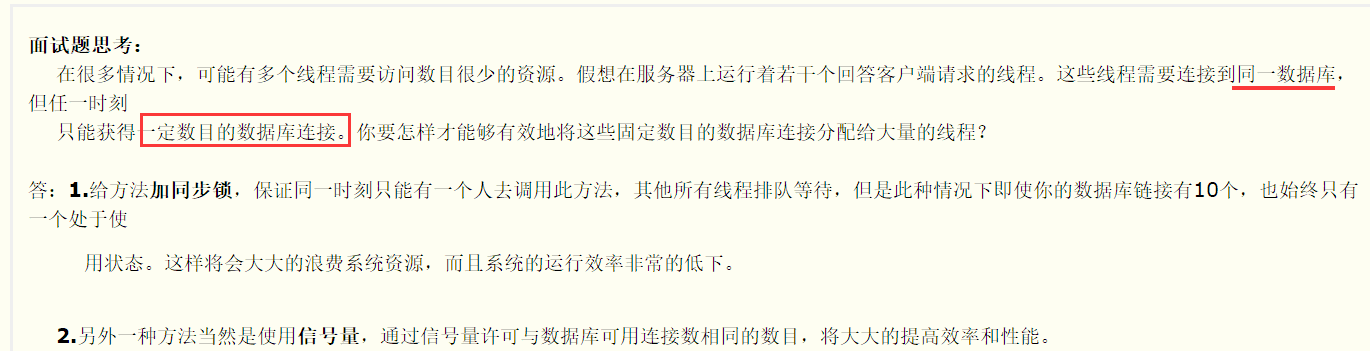
<http://www.importnew.com/22007.html>

<https://my.oschina.net/hosee/blog/639352>

## 5、线程切换



## 6、信号量



## AQS

### AbstractQueuedSynchronizer

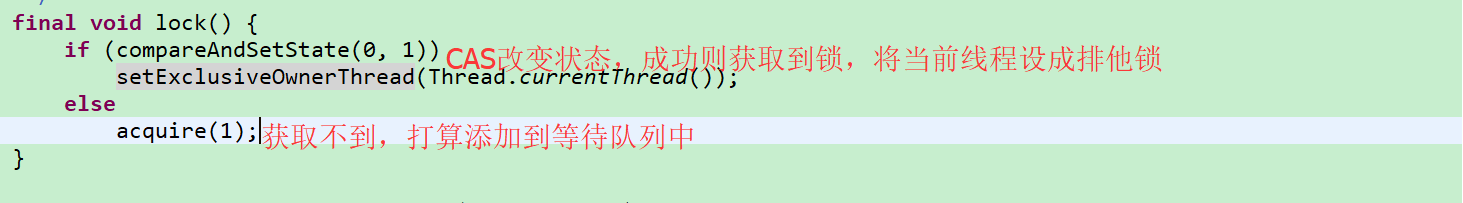
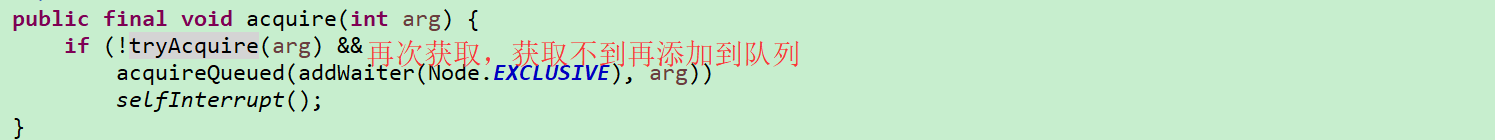
<https://www.jianshu.com/p/df0d7d6571de>

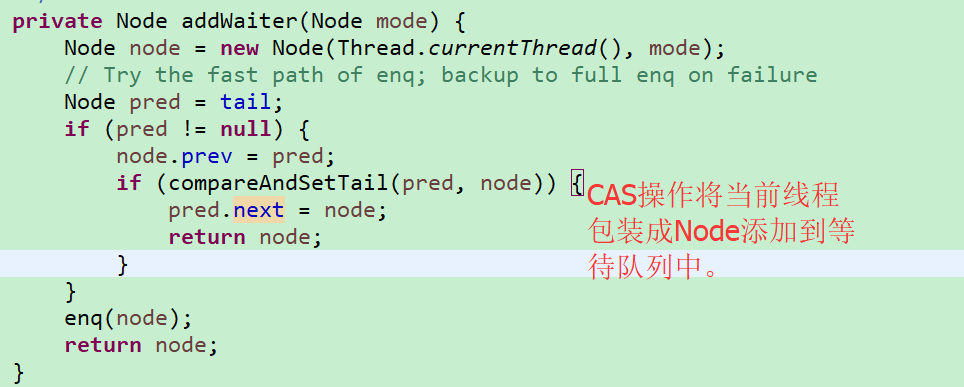
### 1、reentrantLock

#### 实现原理（源码分析）

通过**CAS**操作维护一个**锁的状态**，0表示锁没被获取，1表示获取到锁。通过这个状态的来判断是否可以获得锁，另外需要维护一个**等待队列**，用来保存当前没有获取到锁的线程，线程进入到队列后进行挂起**park（）**操作，当获取到锁的线程执行完，会针对一个线程进行**unpark()**操作。

lock()方法的实现：



添加进去之后还会再次尝试获取，实在获取不到就执行park（）操作。

#### 与synchronized区别

**锁的实现**：re基于JDK实现，通过CAS算法思想去获取锁，避免线程进入内核态阻塞挂起。sy通过JVM实现。

**性能的区别**：sy引入偏向锁，自旋锁优化后（主要是借助CAS算法思想，将加锁在用户态就解决，避免进入内核态的线程阻塞），性能与re差不多。官方推荐sy，因为简单。

**功能的区别**：便利性：sy更便利，且由jvm来管理锁，re需要手动加锁，解锁，因此避免死锁，用finally保证。

锁的粒度：re粒度更细。

调试便利性:sy可以生成线程转储信息，更容易定位问题。

**异常的区别**：有异常时，re不会自动释放锁，所以使用时需要用finally来unlock，sy会自动释放锁。

**Re独有的功能：**

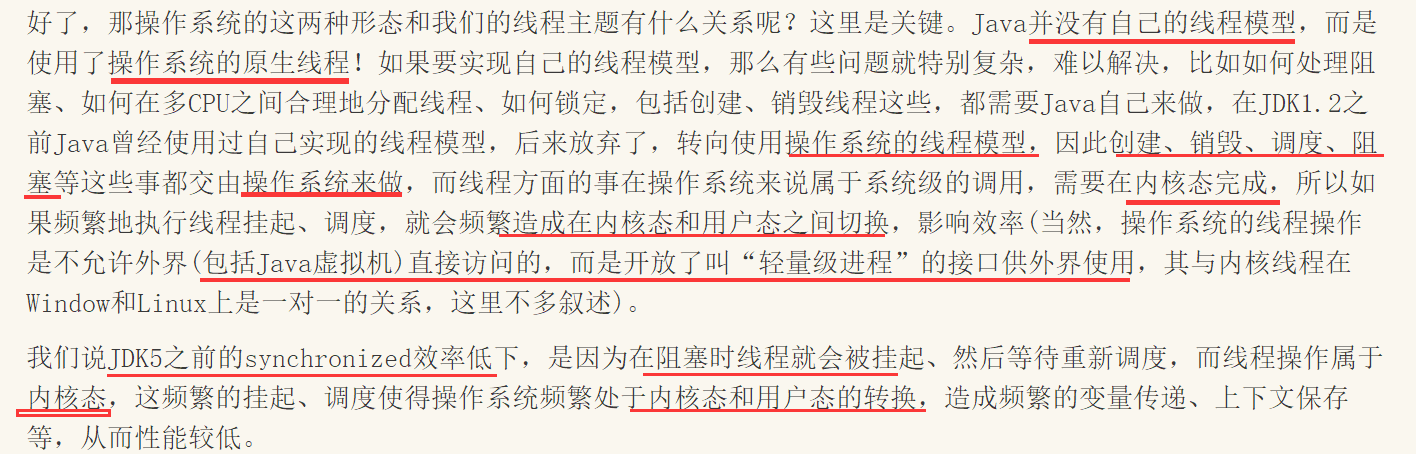
可设置公平锁和非公平锁。

可中断线程。

可限制锁的获取时间。

可使用Conditions分组唤醒线程。

#### 用户态与内核态



#### 公平与非公平锁

当竞争激烈时，且锁的持有时间相对较短，或者请求锁的时间较短，那么应该使用非公平锁，因为唤醒一个线程会有延迟，所以在唤醒一个线程B时，C刚好过来就可以获取到锁，可以提高吞吐量。相反，锁的持有时间相对较长，或者请求锁的时间较长时，使用公平锁。

### Semaphore

### CountDownLatch

### CycliBarrier

与countDownLatch的区别：

CountDownLatch强调的是一个线程等待其他线程达到某个状态后再一起执行，内部做减法操作，且countDownlatch减到0时是不可以重新计算的。

cycliBarrier强调的是多个线程之间互相等待达到某个状态，内部做加法操作，当一次条件满足后，会重置计数器，可循环执行。

### ReadWriteLock

### LockSupport

内部实现类似信号量，当挂起线程时获取许可（减少一个许可），唤醒线程时返还一个许可。因为不会出现线程冻结（suppend（）与resume（）有可能发生线程冻结，resume（）发生在suppend之前会冻结）

# 缓存

## Guava 的cache

1. 类似于ConcurrentHashMap的实现。
2. <https://blog.csdn.net/guozebo/article/details/51590517>
3. <http://ifeve.com/google-guava-cachesexplained/>
4. newcommondata在上线时缓存到ConcurrentMap中。

# 分布式微服务

## rpc协议与restfulapi

<https://blog.csdn.net/douliw/article/details/52592188>

接口调用通常包含两个部分，序列化协议（xml,json,text,protobuf..）和通信协议(http,soap,websocket)

## rpc原理

<https://blog.csdn.net/u013521220/article/details/70157956>

1. dubbo框架的多种协议（默认dubbo协议）

<https://blog.csdn.net/fuyuwei2015/article/details/72848310>

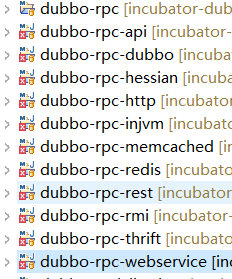


图1 dubbo 提供的rpc调用协议

# Java反射

## 1、JDK自带反射

<https://blog.csdn.net/sinat_38259539/article/details/71799078>

## javassit动态编程实现反射

<https://www.cnblogs.com/duanxz/p/3571217.html>

# Dubbo

## 注意：

本篇讲的是本地暴露,面试主要问得多是远程暴露,因为远程暴露涉及到了zookeeper和netty.比如和zookeeper相关的最常问的两道题是服务提供者能实现失效踢出是根据什么原理?,dubbo中zookeeper做注册中心,如果注册中心集群都挂掉,那发布者和订阅者还能通信吗?.当然如果这两个你答得好,还会往下问,比如zookeeper实现分布式锁,zookeeper选举机制等等,netty能问得可就很多了,AIO BIO NIO,线程模型等等,netty如果要问的话一般要问得比zookeeper要深一些.

## dubbo涉及技术

**代理（Proxy：javassist等）**

**反射（Invoke）**

**协议（Protocol：DubboProtocl等）**

**序列化（Hession等）**

**NIO（netty,mina）**

**SPI（java spi）**

**装饰器模式（wrapper）**

**观察者模式（订阅和监听）**

**spring自定义标签（容器启动时bean的解析）**

## 观察者模式与发布订阅模式的区别

<https://www.sohu.com/a/207062452_464084>

一句话：观察者模式的被观察者（Subject）是知道它的观察者（Obsevers)的存在，其内部维护这个列表，消息是直接从subject发送到obsever的，两者紧耦合。而发布订阅模式中，发布者不知道订阅者的存在，两者的消息传输是通过消息代理来实现的（比如消息队列），两者是松耦合的。

## Dubbo Inteview

### 什么是服务本地暴露与远程暴露？区别是什么？

有时一个发布者也会是消费者，需要调用自己的服务，那这就不需要在通过网络来访问，将服务暴露在JVM中为本地暴露，本地暴露的服务url以“injvm”开头，该常量在Constants类中。远程暴露通过把服务的IP地址，端口暴露给远程客户端，访问通过网络通信访问。

### 服务暴露过程中做了什么？

1. 暴露本地服务，2、暴露远程服务，3、启动netty监听，4、连接到zookeeper，

5、在zookeeper上注册，6、监听zookeeper。

### 3、**服务提供者能实现失效踢出是根据什么原理?**

### 4、dubbo线程模型

提供了五种线程模型，由于dubbo默认使用netty实现通信，所以线程模型跟netty差不多。参考netty线程模型及https://blog.csdn.net/manzhizhen/article/details/73436619

<https://www.cnblogs.com/xhj123/p/9095278.html>

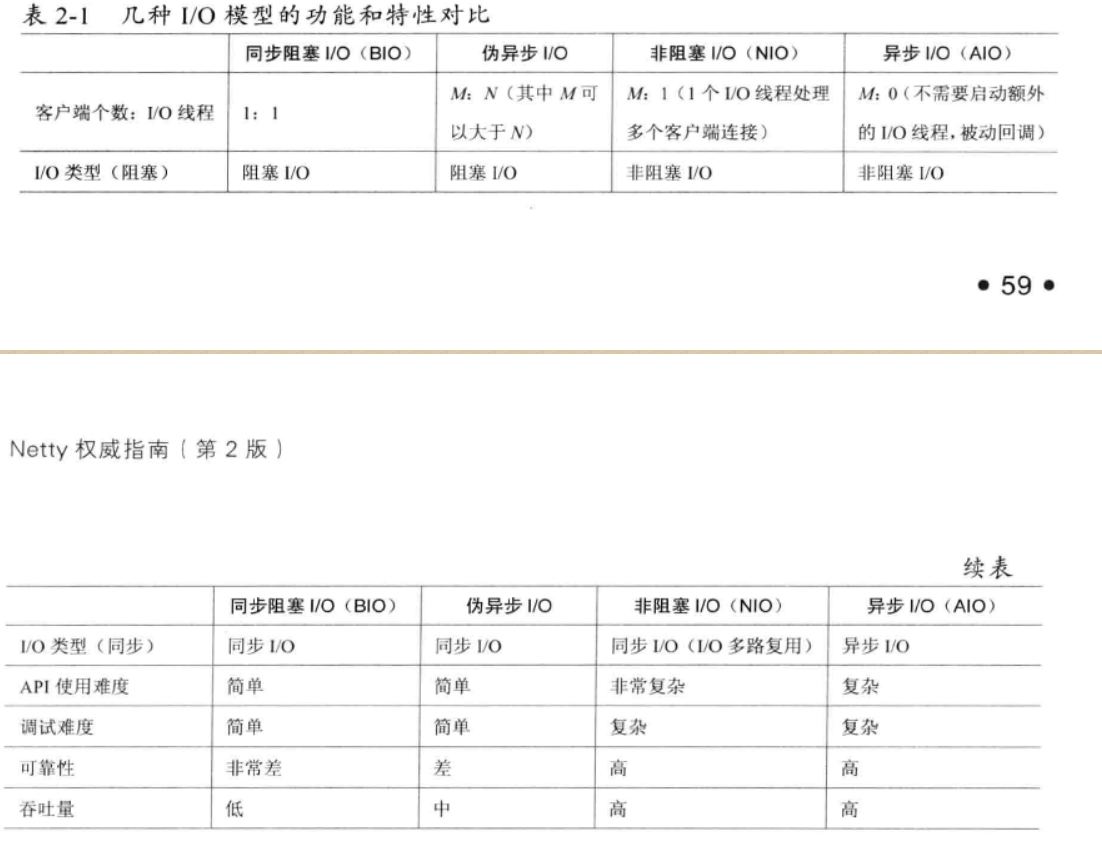
# Zookeeper

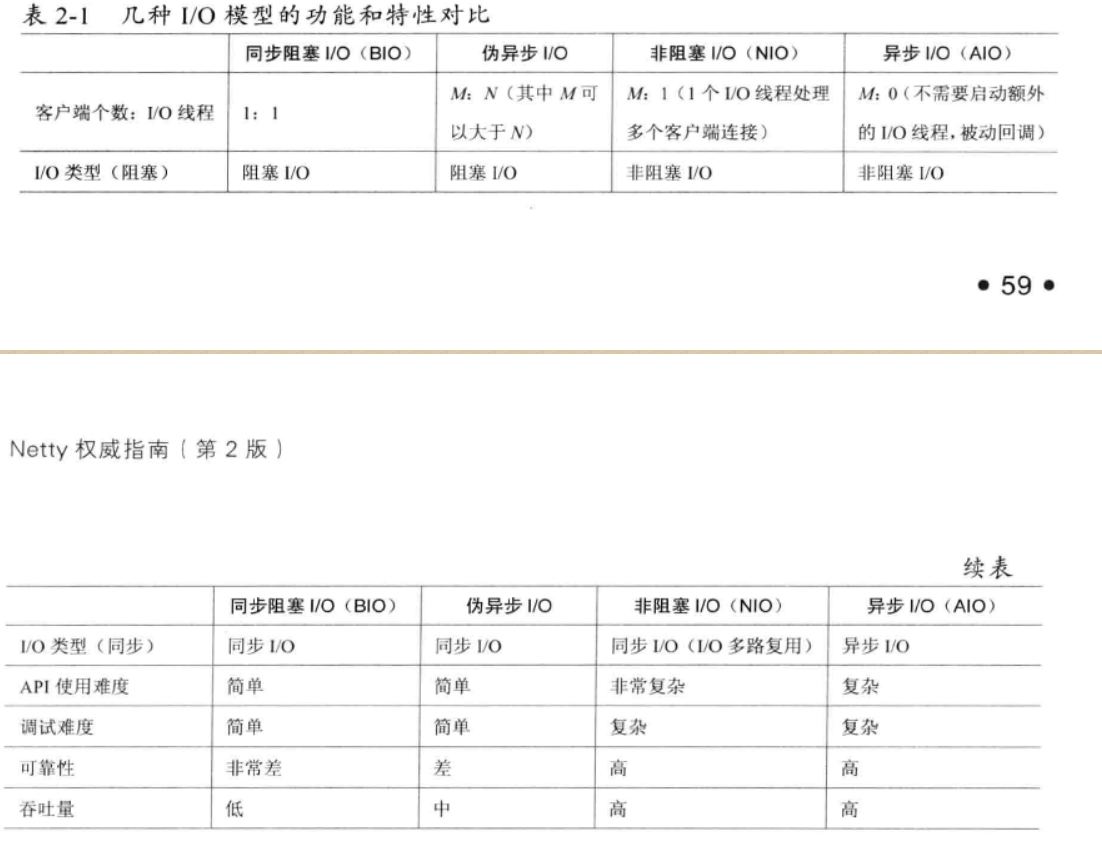
## 实现原理

ZAB（原子消息广播），paxos算法。

# Netty

## BIO,NIO,AIO工作原理，I/O模型，对比。





## netty线程模型

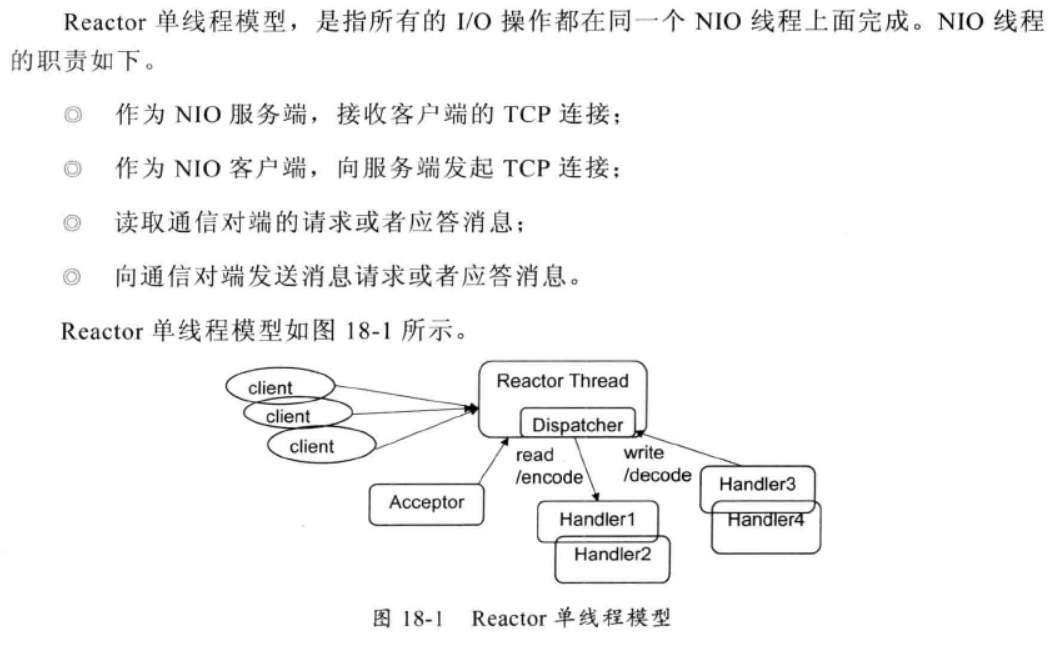
常见写法：

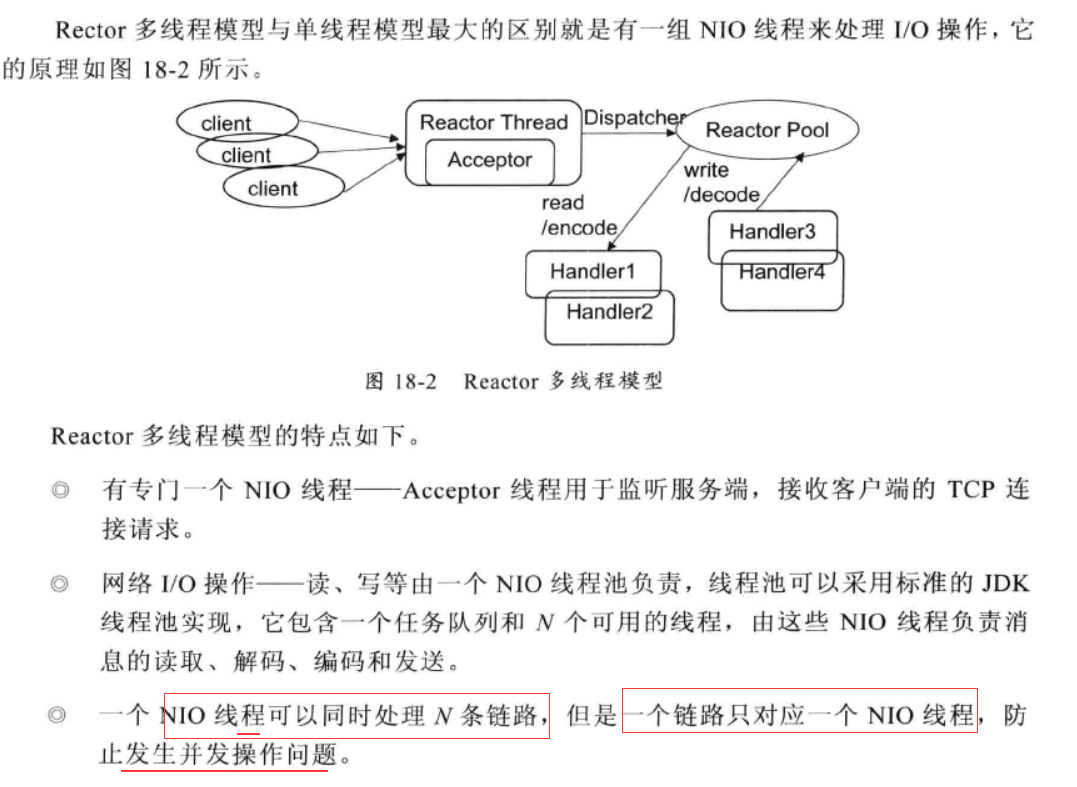
//boss线程监听端口，worker线程负责数据读写

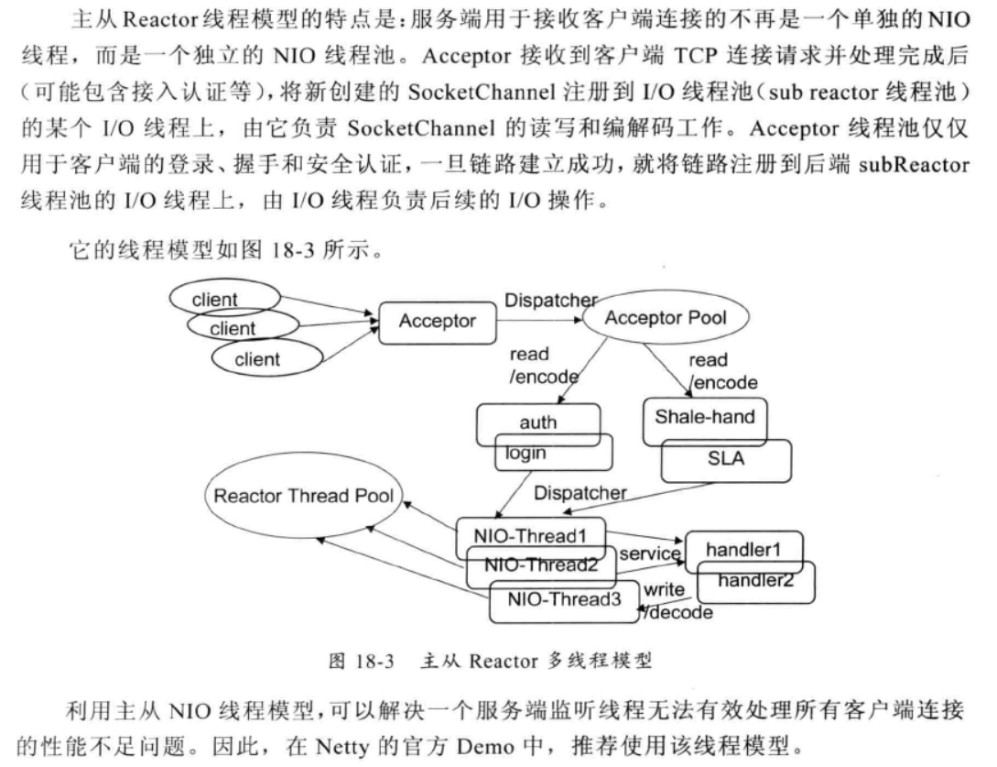
ExecutorService boss = Executors.newCachedThreadPool(1);

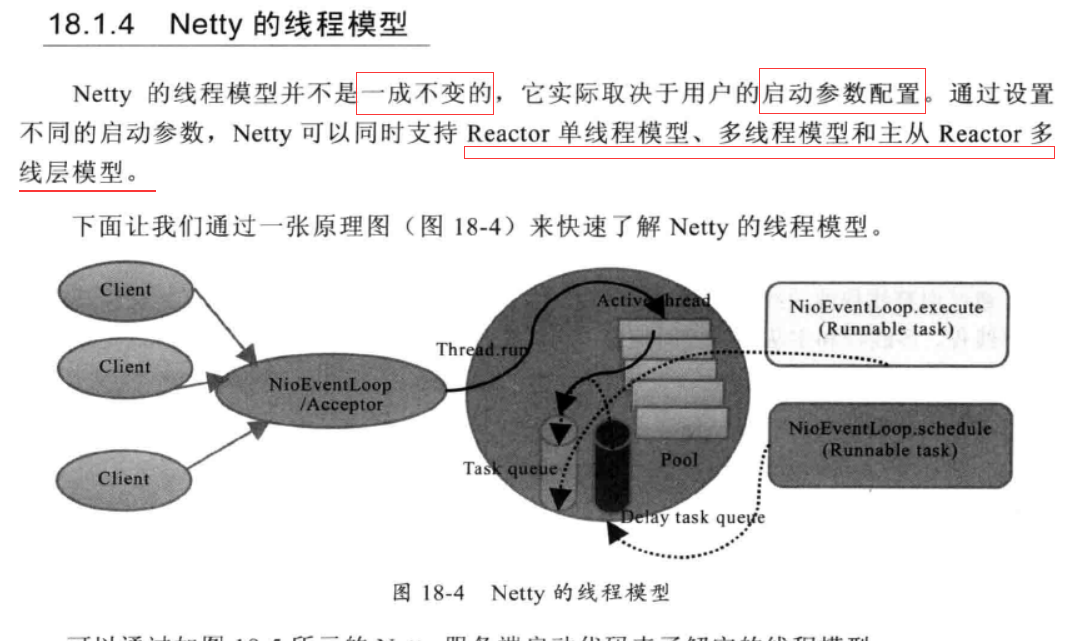
ExecutorService worker = Executors.newCachedThreadPool();

Netty支持单线程模型，多线程模型，主从多线程模型。如下图所示：

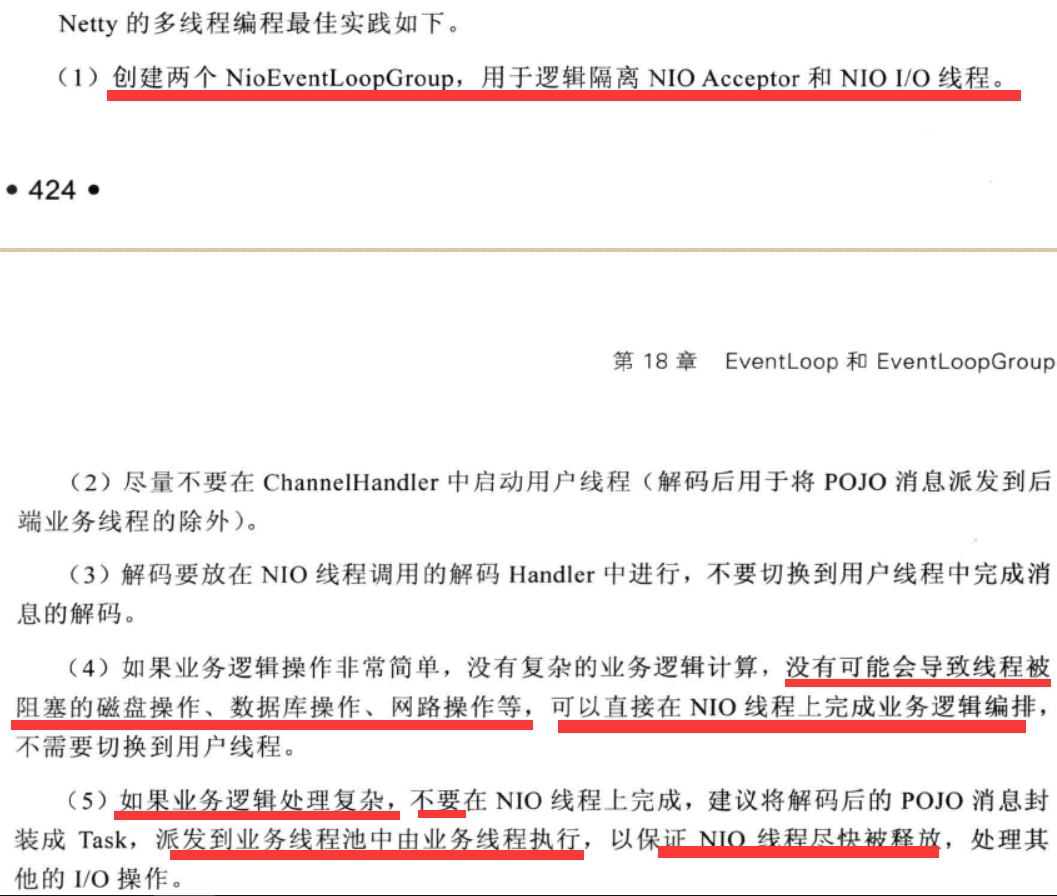
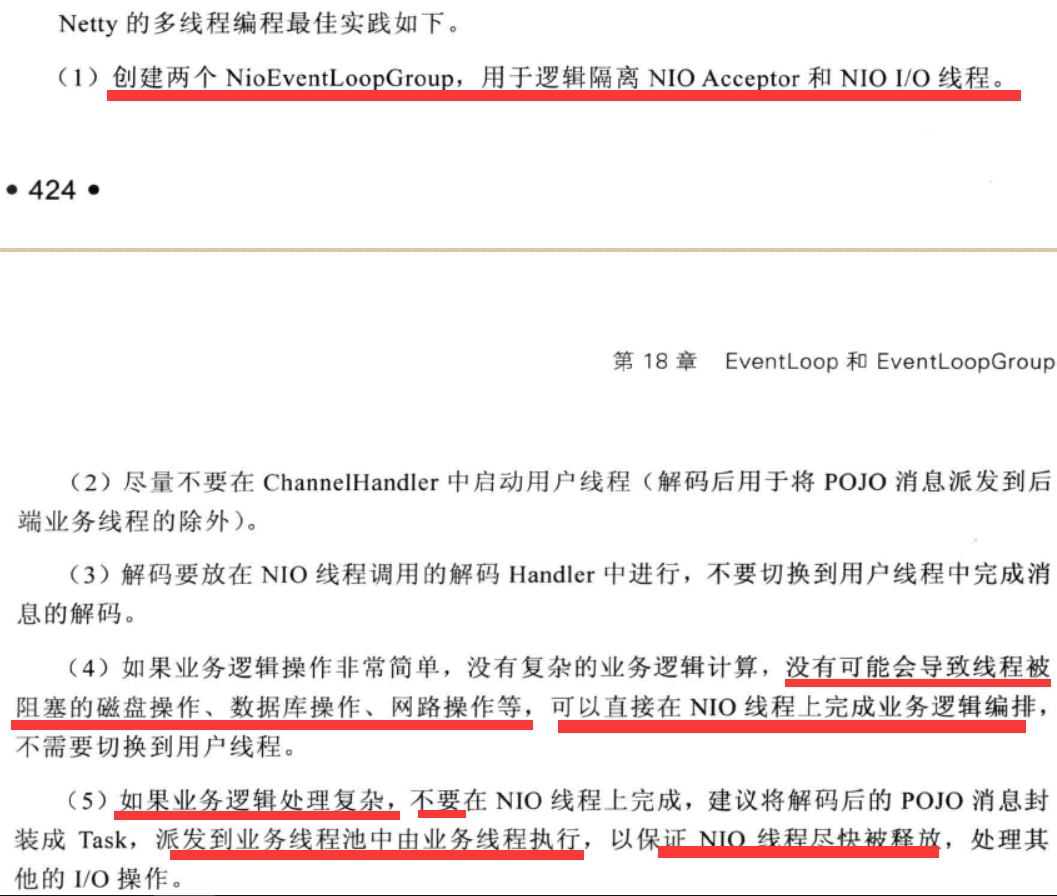




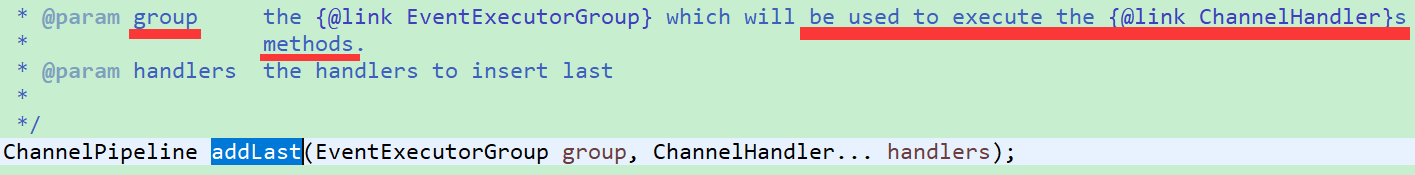




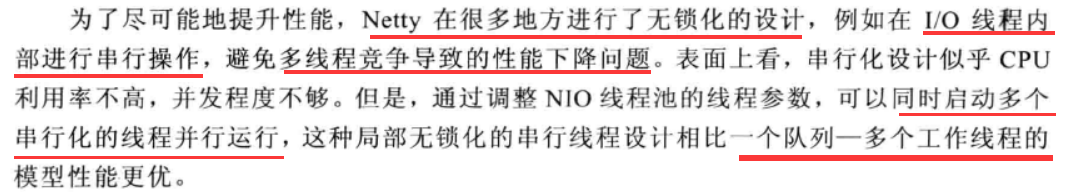
**因此，官方推荐的最佳实践有：**



我们注意到第五点，当业务处理复杂时，将业务放在业务线程池中处理，在netty3中，我们需要自己定义线程池来执行业务，在netty4之后，我们可以使用netty官方的NioEventLoopGroup线程池来处理业务，具体为：



由于NioEventLoopGroup线程池采用无锁化串行操作设计，即：



由此可以参照这个思想写出对请求有序处理的线程池；先说说有序处理的用处：

假如某个操作需要锁住玩家，若线程同时接收到多个锁住玩家的请求时，线程池里的所有线程均被安排去执行这些请求，那么此时线程池里的线程只能等待其他线程处理完释放玩家的锁才能进行操作，其他时间一直被挂起，造成浪费。因此，可以考虑将一个玩家的所有相同请求操作放在一个队列中，指定一个线程来处理，那么就可以实现请求的有序处理。

**霸业里的MessageHandler类就是这样子设计的。(需要看看源码，看看某些数据结构原理，如ConcurrentLinkedQueue的原理)**

此外，dubbo默认使用netty进行网络通信，所以，当消费者发送调用请求过来时，由netty的I/O线程负责处理或者采用I/O线程与业务线程同时处理的方式。对此，dubbo提供了五种线程模型。