获取处理器个数

Runtime.getRuntime().availableProcessors()

固定的4中线程池

Executors.newSingleThreadExecutor() 只有一个线程的线程池，保证任务顺序执行。

Executors.newScheduledThreadPool(int n)：创建一个定长线程池，支持定时及周期性任务执行

scheduledThreadPool.scheduleAtFixedRate(new Runnable(), 1, 3, TimeUnit.SECONDS); 延迟一秒，每3秒执行一次

Executors.newFixedThreadPool(int n)：创建一个可重用固定个数的线程池，以共享的无界队列方式来运行这些线程

Executors.newCacheThreadPool()：可缓存线程池，先查看池中有没有以前建立的线程，如果有，就直接使用。如果没有，就建一个新的线程加入池中，缓存型池子通常用于执行一些生存期很短的异步型任务

不提倡使用，他们都存在一些弊端都会导致内存溢出oom：

1）FixedThreadPool 和 SingleThreadPool: 允许的请求队列长度为 Integer.MAX\_VALUE，可能会堆积大量的请求，从而导致 OOM。 2）CachedThreadPool 和 ScheduledThreadPool: 允许的创建线程数量为 Integer.MAX\_VALUE，可能会创建大量的线程，从而导致 OOM。

自定义线程池

ThreadPoolExecutor有很多重载的构造参数，下面这个是参数最全的定义。

public ThreadPoolExecutor(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue,  
 ThreadFactory threadFactory) {  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
 threadFactory, *defaultHandler*);  
}

参数最少的定义。

public ThreadPoolExecutor(int corePoolSize,  
 int maximumPoolSize,  
 long keepAliveTime,  
 TimeUnit unit,  
 BlockingQueue<Runnable> workQueue) {  
 this(corePoolSize, maximumPoolSize, keepAliveTime, unit, workQueue,  
 Executors.*defaultThreadFactory*(), *defaultHandler*);  
}

ThreadPoolExecutor threadPoolExecutor = new ThreadPoolExecutor(1000,  
 1000, 60, TimeUnit.*SECONDS*, new LinkedBlockingQueue<>(),  
 new NamedThreadFactory("test-rp-"), new ThreadPoolExecutor.AbortPolicy());

处理任务的优先级为：   
核心线程corePoolSize、任务队列workQueue、最大线程maximumPoolSize，如果三者都满了，使用handler处理被拒绝的任务。   
当线程池中的线程数量大于 corePoolSize时，如果某线程空闲时间超过keepAliveTime，线程将被终止。这样，线程池可以动态的调整池中的线程数。

阻塞队列（BlockingQueue）有三种

**无界队列**

队列大小无限制，常用的为无界的LinkedBlockingQueue，使用该队列做为阻塞队列时要尤其当心，当任务耗时较长时可能会导致大量新任务在队列中堆积最终导致OOM。阅读代码发现，Executors.newFixedThreadPool 采用就是 LinkedBlockingQueue，而楼主踩到的就是这个坑，当QPS很高，发送数据很大，大量的任务被添加到这个无界LinkedBlockingQueue 中，导致cpu和内存飙升服务器挂掉。

**有界队列**

常用的有两类，一类是遵循FIFO原则的队列如ArrayBlockingQueue，另一类是优先级队列如PriorityBlockingQueue。PriorityBlockingQueue中的优先级由任务的Comparator决定。   
使用有界队列时队列大小需和线程池大小互相配合，线程池较小有界队列较大时可减少内存消耗，降低cpu使用率和上下文切换，但是可能会限制系统吞吐量。

在我们的修复方案中，选择的就是这个类型的队列，虽然会有部分任务被丢失，但是我们线上是排序日志搜集任务，所以对部分对丢失是可以容忍的。

**同步移交队列**

如果不希望任务在队列中等待而是希望将任务直接移交给工作线程，可使用SynchronousQueue作为等待队列。SynchronousQueue不是一个真正的队列，而是一种线程之间移交的机制。要将一个元素放入SynchronousQueue中，必须有另一个线程正在等待接收这个元素。只有在使用无界线程池或者有饱和策略时才建议使用该队列。

4种抛弃策略

**AbortPolicy：会抛异常**RejectedExecutionException

## DiscardPolicy：不做任何处理

## DiscardOldestPolicy：抛弃旧的任务。

## CallerRunsPolicy：调用者执行任务。