**线程池原理**

# 线程池高频重点

1. 为什么使用线程池，优势是什么？
2. 线程池如何使用？（架构说明、编码实现、ThreadPoolExecutor）
3. 线程池的几个重要参数介绍
4. 说说线程池的底层工作原理
5. 线程池用过吗？生产上你如何设置合理参数？

# 为什么使用线程池

## 1、Java线程的理解

线程是调度CPU的最小单元，也叫轻量级进程LWP（Light Weight Process）

线程模型分类：

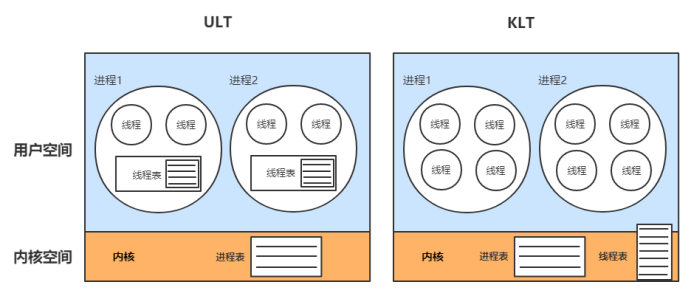
* 用户级线程（User-Level Thread，简称ULT）

用户程序实现，不依赖操作系统核心，应用提供创建、同步、调度、和管理线程的函数来控制用户线程。不需要用户态/内核态切换，速度快。内核对ULT无感知。

* 内核级线程（User-Level Thread，简称KLT）

系统内核管理线程（KLT），内核保存线程的状态和上下文信息，线程阻塞不会引起进程阻塞。线程的创建、调度和管理由内核完成，效率比ULT要慢，比进程操作快。

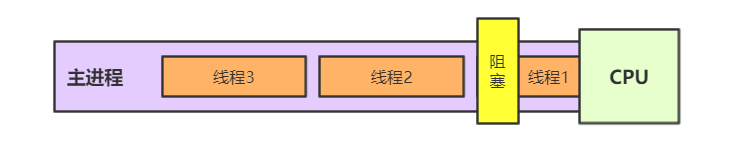
## 2、用户态到内核态的切换



操作系统分为**内核空间**和**用户空间。**当进程运行在内核空间时就处于内核态，当进程运行在用户空间时就处于用户态。为了安全，它们是隔离的，即使用户的程序崩溃了，内核也不受影响。

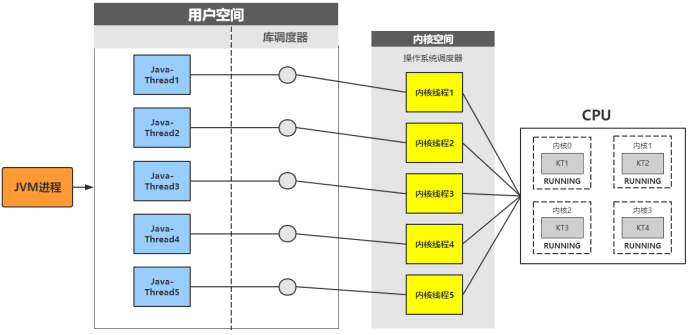
Kernel space 可以执行任意命令，调用系统的一切资源；User space 只能执行简单的运算，不能直接调用系统资源，必须通过系统接口（又称 system call），才能向内核发出指令。只有内核空间才能拥有CPU的最高特权级别去操作CPU。

**用户线程（ULT）**：是由用户空间里面的用户进程自己创建的线程，并且由用户进程自己维护，进程里面创建的所有线程并没有CPU的使用权限，只有内核才有资格分配CPU的时间片，用户线程是依托主进程去执行的，主进程的所有线程在一条线上执行，进程内如果有线程切换，容易引起阻塞，阻塞的话整个进程也就阻塞了。



**内核线程（KLT）**：主进程里面的线程全部是依附于内核，假设一个JVM进程，进程里面的每一个线程都维护在内核空间的线程表。操作系统是知道内核线程的存在，并为其安排时间片，管理与其有关的内核对象。

Java线程模型是依赖于底层操作系统内核级线程去完成的，两者之间是什么关系？1:1的映射关系。



Java线程与系统内核线程的关系

JVM进程里面可以创建大量的线程，本质上只是在JVM进程里面创建了线程栈空间，栈空间里面会有一些栈帧指令，真正的线程需要通过库调度器去调度内核创建内核线程，创建完内核线程之后，才具有竞争CPU的使用权限。

## 3、上下文切换

线程的创建，本质上都是依赖于内核，线程上下文的切换，就会涉及到用户态到内核态的切换，当线程t1任务还没有执行完，时间片已经用完，要切换到线程t2执行的时候，线程t1的中间状态就要刷回主内存，也就是说线程t1的上下文内容（指令，程序指针，中间数据）要经过总线保存到内核栈空间的Tss任务状态段里面，当线程t2执行完时间片，如果线程t1又竞争到CPU的时间片，就要从内核里面加载任务状态到缓存或寄存器里面。

## 4、线程池优势

什么时候使用线程池？

* 单个任务处理时间比较短
* 需要处理的任务数量很大

**线程池优势**

* 重用存在的线程，减少线程创建，消亡的开销，提高性能
* 提高响应速度。当任务到达时，任务可以不需要的等到线程创建就能立即执行。
* 提高线程的可管理性。线程是稀缺资源，如果无限制的创建，不仅会消耗系统资源，还会降低系统的稳定性，使用线程池可以进行统一的分配，调优和监控。

# 线程池怎么使用

## 1、线程的实现方式

Runnable,Thread,Callable

// 实现Runnable接口的类将被Thread执行，表示一个基本的任务

public interface Runnable {

// run方法就是它所有的内容，就是实际执行的任务

public abstract void run();

}

//Callable同样是任务，与Runnable接口的区别在于它接收泛型，同时它执行任务后带有返回内容

public interface Callable<V> {

// 相对于run方法的带有返回值的call方法

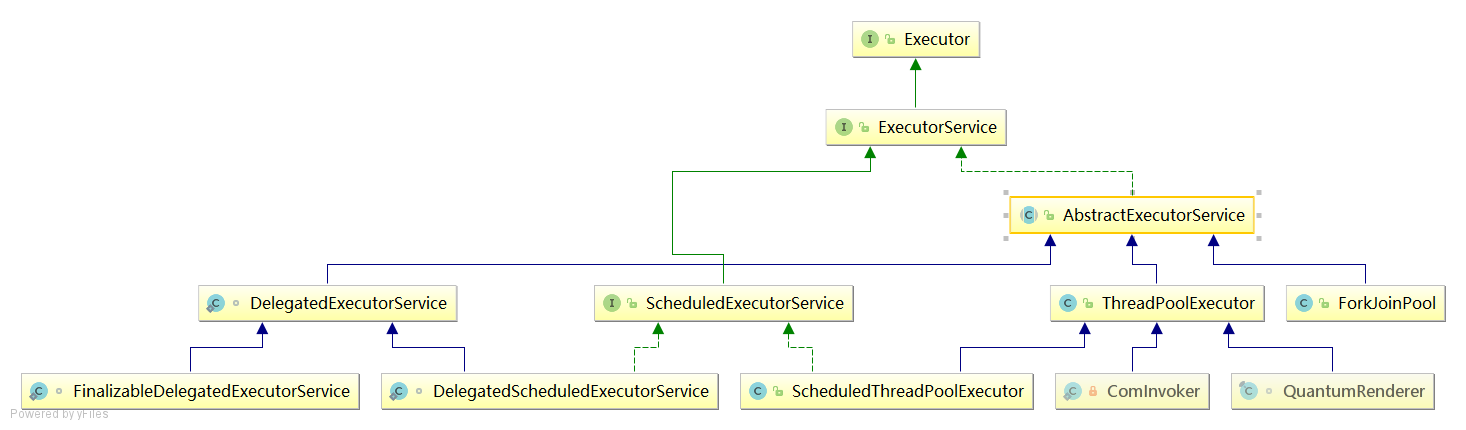
V call() throws Exception;

}

## 2、线程池Executor框架

Java中的线程池是通过Executor框架实现的，该框架中用到了Executor，Executors，ExecutorService，ThreadPoolExecutor这几个类。

Executor接口是线程池框架中最基础的部分，定义了一个用于执行Runnable的execute方法。



从图中可以看出Executor下有一个重要子接口**ExecutorService**，其中定义了线程池的具体行为：

1，execute（Runnable command）：履行Ruannable类型的任务,

2，submit（task）：可用来提交Callable或Runnable任务，并返回代表此任务的Future对象

3，shutdown（）：在完成已提交的任务后封闭办事，不再接管新任务,

4，shutdownNow（）：停止所有正在履行的任务并封闭办事。

5，isTerminated（）：测试是否所有任务都履行完毕了。

6，isShutdown（）：测试是否该ExecutorService已被关闭。

**Executors**辅助工具类，一般创建简单的线程池都是通过Executors这个类的静态方法创建的。

* Executors.newFixedThreadPool(int nThreads);//创建固定大小的线程池，核心数和最大数是一样的
* Executors.newSingleThreadExecutor();//创建一个单线程的线程池。这个线程池的核心数和最大数都是1，也就是相当于单线程串行执行所有任务.
* Executors.newCachedThreadPool();//创建一个可缓存的线程池。核心数是0,最大数是 Integer.MAX\_VALUE，60秒不执行任务就回收
* Executors.newScheduledThreadPool(int corePoolSize);
* Executors.newWorkStealingPool();//1.8新加的线程池,forkJoinPool 可以根据CPU的核数并行的执行，适合使用在很耗时的操作，可以充分的利用CPU执行任务，任务窃取线程池，不保证执行顺序，适合任务耗时差异较大。

## 3、Java中的ThreadPoolExecutor类

java.uitl.concurrent.ThreadPoolExecutor类是线程池中最核心的一个类，因此如果要透彻地了解Java中的线程池，必须先了解这个类。下面我们来看一下ThreadPoolExecutor类的具体实现源码。

public class ThreadPoolExecutor extends AbstractExecutorService {

.....

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,RejectedExecutionHandler handler);

public ThreadPoolExecutor(int corePoolSize,int maximumPoolSize,long keepAliveTime,TimeUnit unit,

BlockingQueue<Runnable> workQueue,ThreadFactory threadFactory,RejectedExecutionHandler handler);

...

}

### 构造器中7大参数的含义

* corePoolSize

核心池的大小，这个参数跟后面讲述的线程池的实现原理有非常大的关系。在创建了线程池后，默认情况下，线程池中并没有任何线程，而是等待有任务到来才创建线程去执行任务，除非调用了prestartAllCoreThreads()或者prestartCoreThread()方法，从这2个方法的名字就可以看出，是预创建线程的意思，即在没有任务到来之前就创建corePoolSize个线程或者一个线程。默认情况下，在创建了线程池后，线程池中的线程数为0，当有任务来之后，就会创建一个线程去执行任务，当线程池中的线程数目达到corePoolSize后，就会把到达的任务放到缓存队列当中；

* maximumPoolSize

线程池最大线程数，这个参数也是一个非常重要的参数，它表示在线程池中最多能创建多少个线程；

* keepAliveTime

表示线程没有任务执行时最多保持多久时间会终止。默认情况下，只有当线程池中的线程数大于corePoolSize时，keepAliveTime才会起作用，直到线程池中的线程数不大于corePoolSize，即当线程池中的线程数大于corePoolSize时，如果一个线程空闲的时间达到keepAliveTime，则会终止，直到线程池中的线程数不超过corePoolSize。但是如果调用了allowCoreThreadTimeOut(boolean)方法，在线程池中的线程数不大于corePoolSize时，keepAliveTime参数也会起作用，直到线程池中的线程数为0；

* unit

参数keepAliveTime的时间单位，有7种取值，在TimeUnit类中有7种静态属性：

TimeUnit.DAYS; //天

TimeUnit.HOURS; //小时

TimeUnit.MINUTES; //分钟

TimeUnit.SECONDS; //秒

TimeUnit.MILLISECONDS; //毫秒

TimeUnit.MICROSECONDS; //微妙

TimeUnit.NANOSECONDS; //纳秒

* workQueue

用来保存等待被执行的任务的**阻塞队列**，且任务必须实现Runable接口，在JDK中提供了如下阻塞队列：

1、ArrayBlockingQueue：基于数组结构的**有界**阻塞队列，按FIFO排序任务；

2、LinkedBlockingQuene：基于链表结构的阻塞队列，按FIFO排序任务，吞吐量通常要高于ArrayBlockingQuene；

3、SynchronousQuene：一个不存储元素的阻塞队列，每个插入操作必须等到另一个线程调用移除操作，否则插入操作一直处于阻塞状态，吞吐量通常要高于LinkedBlockingQuene；

4、priorityBlockingQuene：具有优先级的**无界**阻塞队列；

阻塞队列：

在任意时刻，不管并发有多高，永远只有一个线程能够进行队列的入队或出队操作！

线程安全的队列

有界 | 无界

有界：队列有大小。队列满，只能进行出队操作，所有入队操作必须等待，也就是被阻塞；队列空，只能进行入队操作，所有出队的操作必须等待

无界：理论上是无界的，实际上受物理主机内存的大小限制

* threadFactory

它是ThreadFactory类型的变量，用来创建新线程。默认使用Executors.defaultThreadFactory() 来创建线程。使用默认的ThreadFactory来创建线程时，会使新创建的线程具有相同的NORM\_PRIORITY优先级并且是非守护线程，同时也设置了线程的名称。

* handler

线程池的饱和策略，当阻塞队列满了，且没有空闲的工作线程，如果继续提交任务，必须采取一种策略处理该任务，线程池提供了4种策略：

ThreadPoolExecutor.AbortPolicy:丢弃任务并抛出RejectedExecutionException异常。

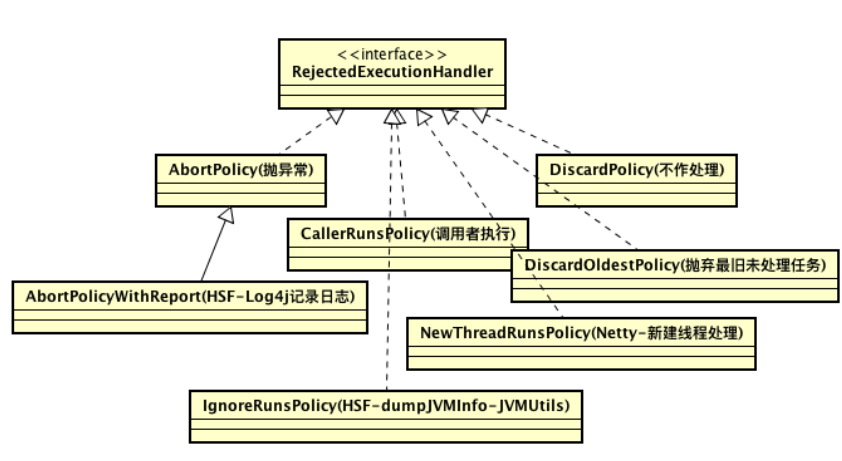
ThreadPoolExecutor.DiscardPolicy：也是丢弃任务，但是不抛出异常。

ThreadPoolExecutor.DiscardOldestPolicy：丢弃队列最前面的任务，然后重新尝试执行任务（重复此过程）

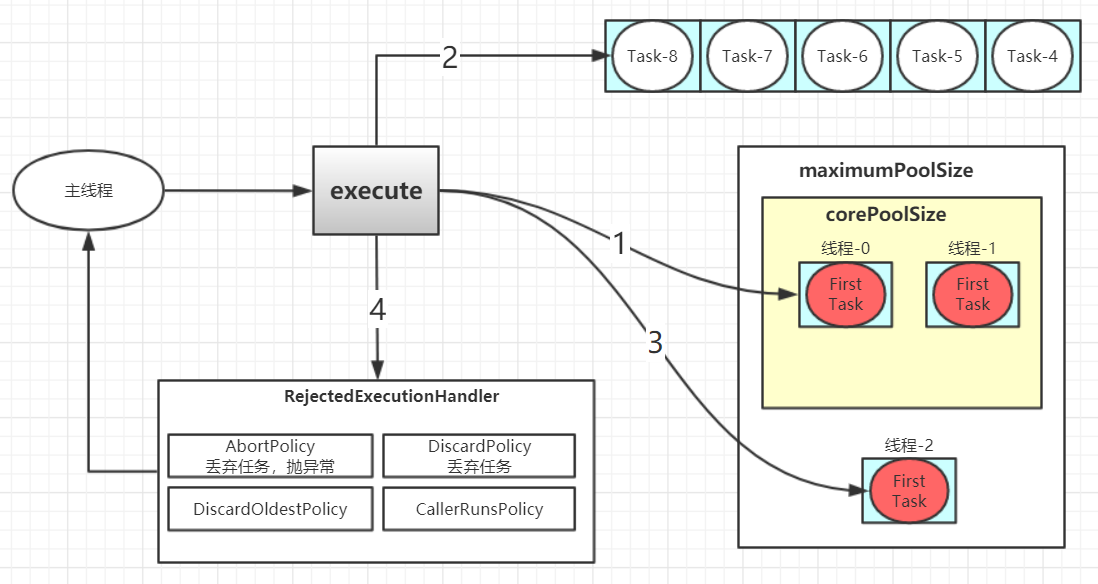
ThreadPoolExecutor.CallerRunsPolicy：“调用者运行”一种调节机制，该策略既不会抛弃任务，也不会抛出异常，而是将某些任务回退到调用者（由调用线程处理该任务），从而降低新任务的流量

* 上面的4种策略都是ThreadPoolExecutor的内部类。

当然也可以根据应用场景实现RejectedExecutionHandler接口，自定义饱和策略，如记录日志或持久化存储不能处理的任务。



# 深入剖析线程池实现原理



## 线程池执行过程

1. 当N多任务提交过来，线程池只接收固定的任务，必须要是实现Runnable接口，或者Callable接口的任务；
2. 任务通过pool.execute()方法丢到线程池里面去，然后由线程池自行决定怎么去调用
3. 以上面示例为例：
   1. 首先来了两个任务，线程池会创建两个核心线程去执行这两个任务
   2. 当来第三个任务的时候，核心线程池已经满了
   3. 第三个任务会被放到阻塞队列里面去，直到把队列放满
   4. 当阻塞队列放满了之后，就会创建非核心线程执行任务
   5. 当核心线程，阻塞队列，非核心线程都满了之后，就会触发拒绝策略
   6. 线程池已经默认定义了四种拒绝策略，也可以自己去扩充自定义拒绝策略

## 源码分析

### 1、线程池生命状态

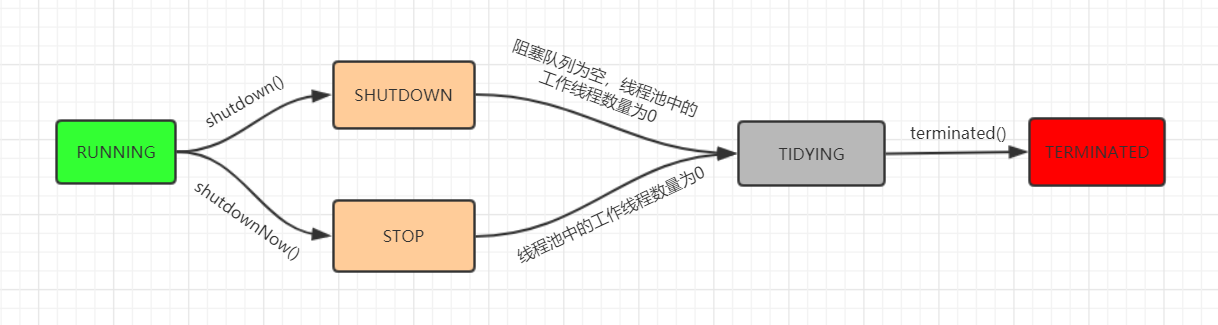
Running ：能接收新任务，以及处理已经添加的任务

Shutdown：不接受新任务，可以处理已经添加的任务

Stop：不接收新任务，不处理已经添加的任务，并且中断中断正在处理的任务

Tidying：所有的任务已经终止，ctl记录的任务数量为“0”（ctl负责记录线程池的运行状态与活动线程数）

Terminated：线程池彻底终止，则线程池转化为terminated状态



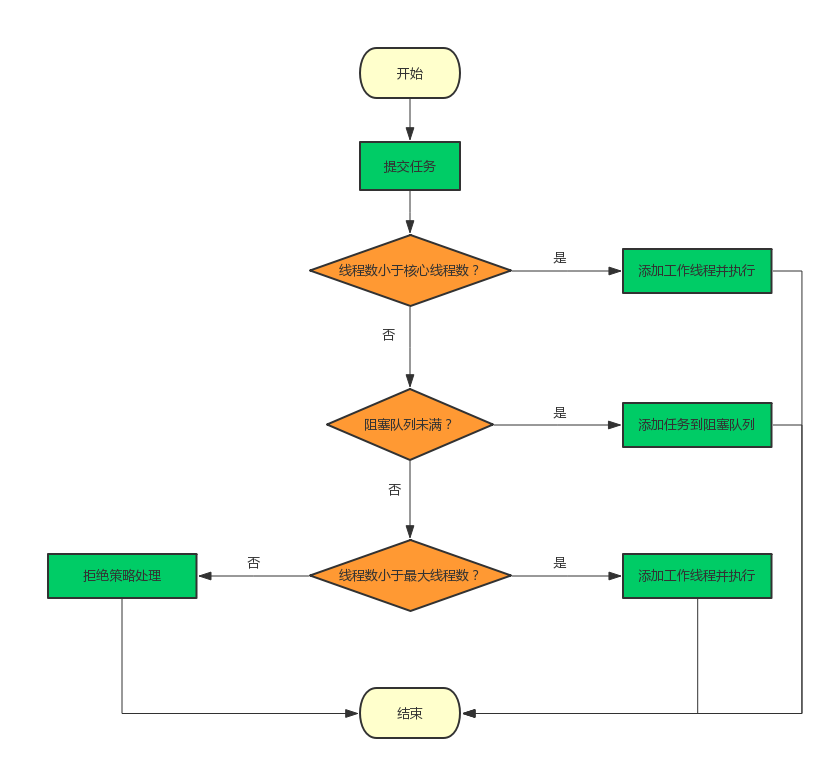
### 2、任务执行：execute（）

**源码分析**

简单来说，在执行execute()方法时如果状态一直是RUNNING时，的执行过程如下：

1. 如果workerCount < corePoolSize，则创建并启动一个线程来执行新提交的任务；
2. 如果workerCount >= corePoolSize，且线程池内的阻塞队列未满，则将任务添加到该阻塞队列中；
3. 如果workerCount >= corePoolSize && workerCount < maximumPoolSize，且线程池内的阻塞队列已满，则创建并启动一个线程来执行新提交的任务；
4. 如果workerCount >= maximumPoolSize，并且线程池内的阻塞队列已满, 则根据拒绝策略来处理该任务, 默认的处理方式是直接抛异常。

execute方法执行流程如下：



# 如何合理配置线程池的大小

我们知道，环境具有多变性，**设置一个绝对精准的线程数其实是不大可能的**，但我们可以通过一些实际操作因素来计算出一个合理的线程数，避免由于线程池设置不合理而导致的性能问题。下面我们就来看看具体的计算方法。

一般多线程执行的任务类型可以分为 CPU 密集型和 I/O 密集型，根据不同的任务类型，我们计算线程数的方法也不一样。

**CPU 密集型任务**：这种任务消耗的主要是 CPU 资源，可以将线程数设置为 **N（CPU 核心数）+1**，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。

**I/O 密集型任务**：这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以**多配置一些线程**，具体的计算方法是 2N。

在平常的应用场景中，我们常常遇不到这两种极端情况，那么碰上一些常规的业务操作，比如，通过一个线程池实现向用户定时推送消息的业务，我们又该如何设置线程池的数量呢？

此时我们可以参考以下公式来计算线程数：

**线程数=N（CPU核数）\*（1+WT（线程等待时间）/ST（线程时间运行时间））**