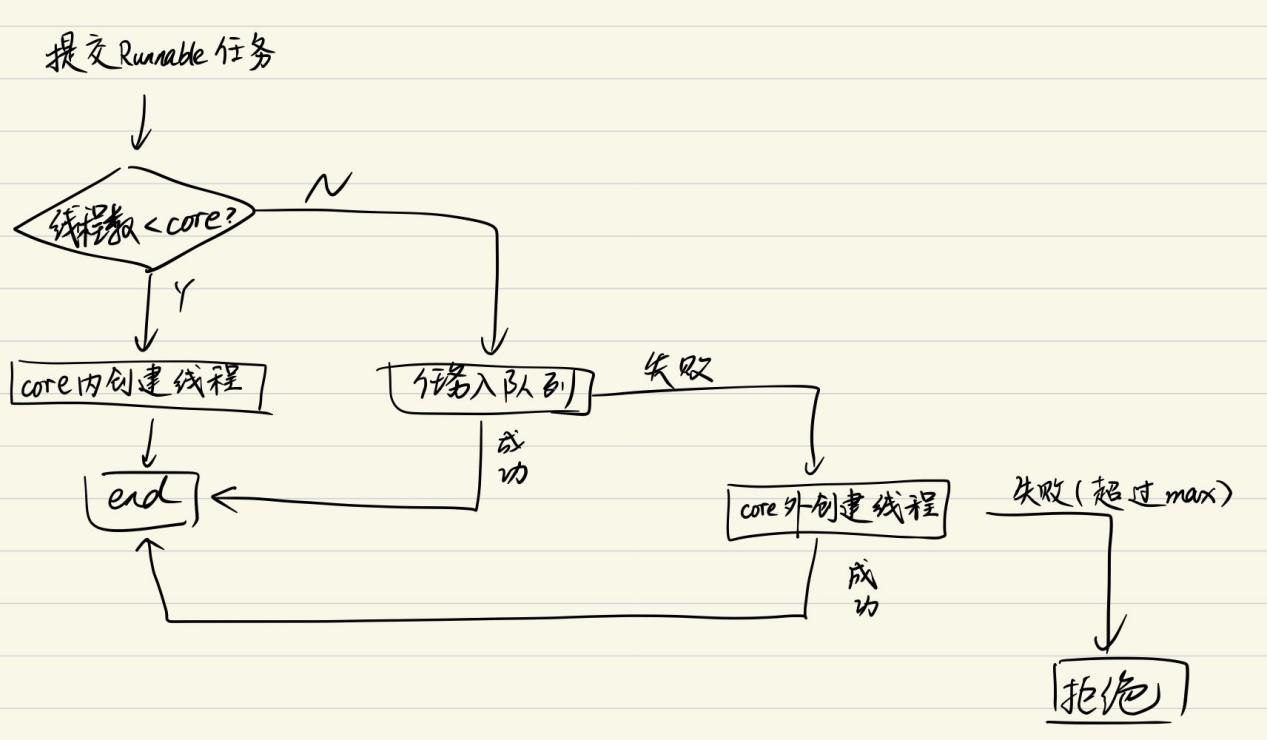
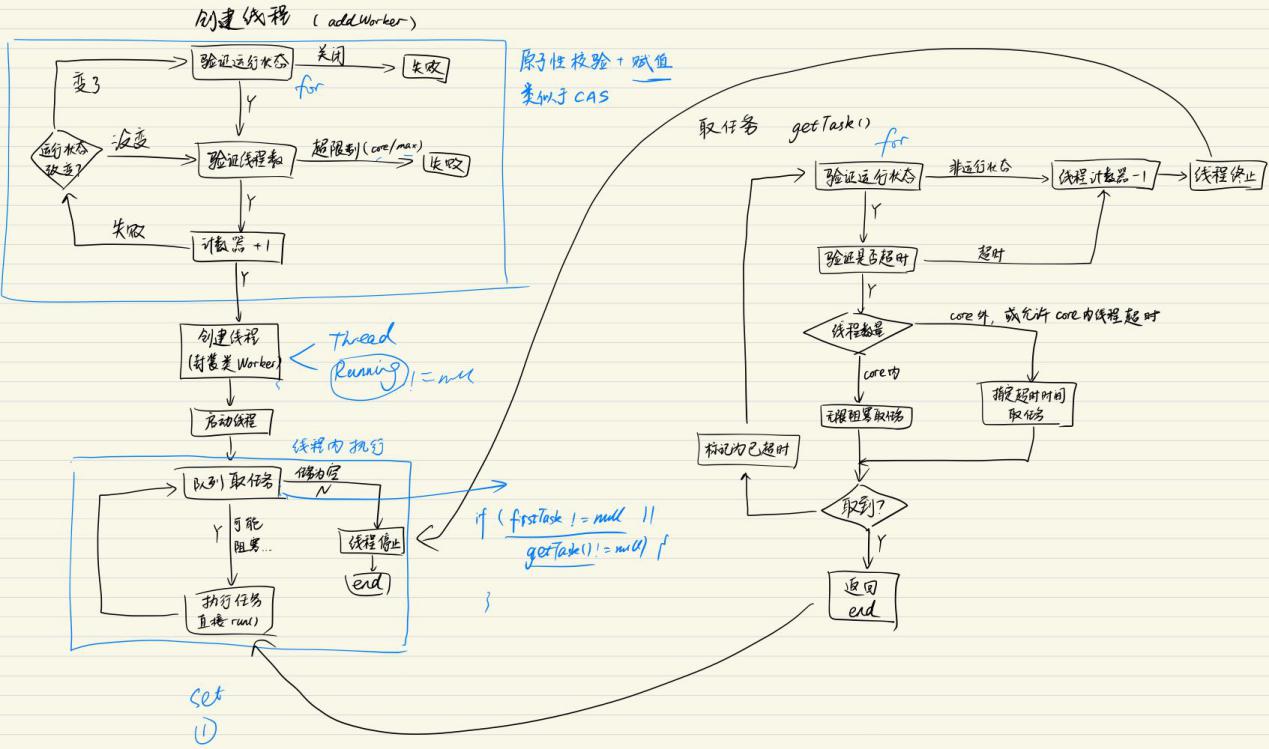
# 线程池原理





# 线程池参数最优配置

## 从CPU密集型程序开始

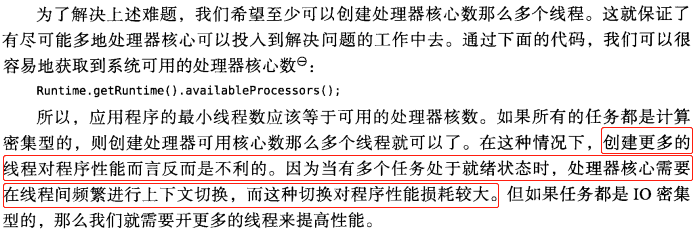
### 先推理

首先，举个例子：多线程计算2-20000的全部素数。

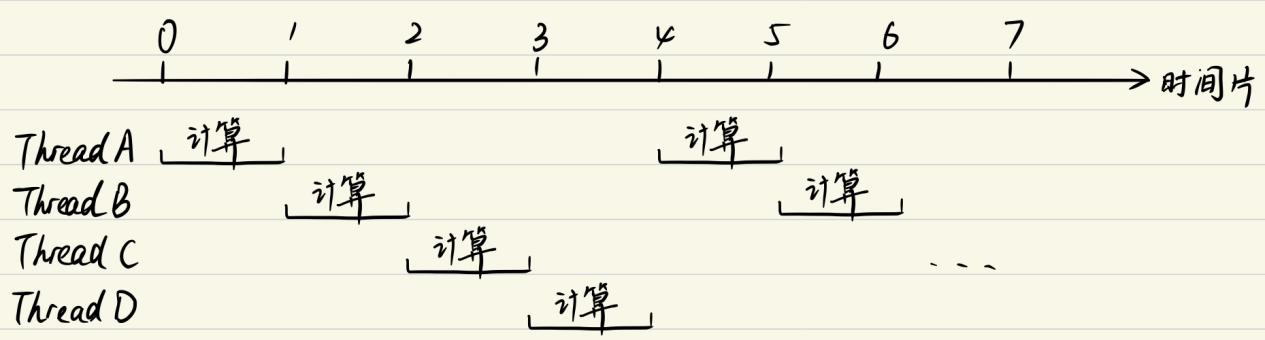
可以确定，这是一个CPU（计算）密集型的程序，处理速度完全取决于CPU的运算速度，与外界因素无关。

假如CPU核数 = 1，那么无论单线程还是多线程，全部运算量都要怼到这个CPU上，速度相同。而多线程还会增加CPU切换线程上下文的消耗，反而更慢。

假如CPU核数 = 4，那么线程数设为4的性能最高。若小于4，则会有CPU处于空闲状态；若大于4，仍旧是所有CPU全速运行，直到停止，所以对程序性能并没有提升，反而会因为切换上下文对性能有所消耗。



同时用4个线程来处理，那么一个CPU上的时间片分配，大概是这样：

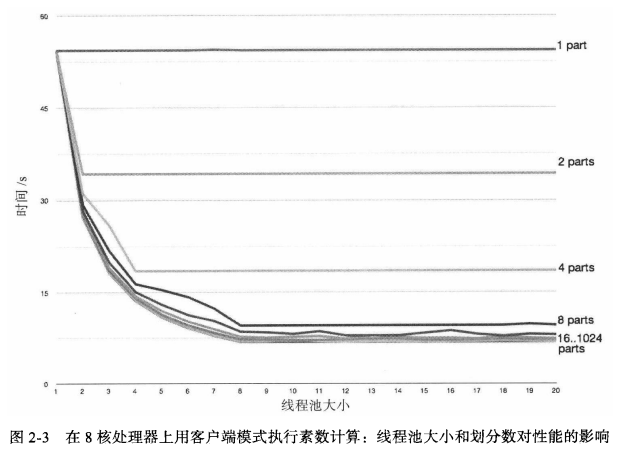


结论：CPU（计算）密集型的程序，线程数设置为CPU核数即可。

补充：《Java并发编程实战》8.2节认为，CPU密集型程序，线程数可以设置为CPU核数+1，这样即使当线程偶尔由于页缺失故障或其他原因而暂停时，这个“额外”的线程也能确保CPU的时钟周期不会被浪费。

### 实际效果

下图是：8核处理器，多线程运行找素数程序的效果图。



可以看出：

1. 不管如何划分任务，8核8线程，已经达到性能极限值，更高的线程数不会降低运行时间；
2. 线程数与运行时间成反比：4线程的运行时间是8线程的两倍，符合猜想。

## IO密集型程序

### 先推理

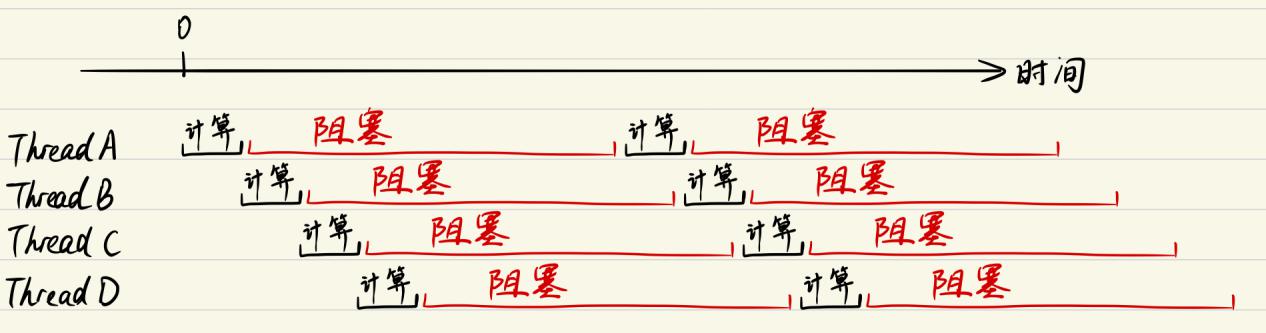
这类程序运行时通常包括：计算+阻塞 两部分。

比如查询SQL（发出查询请求 + 等待SQL执行）、读写文件（把命令提交给OS + 等待OS执行）、web服务调用（发出http/RPC请求 + 等待返回结果）。

首先一个大前提：**当一个线程阻塞时，CPU会立刻终止线程所在的时间片，将运行权交给其他线程。**

再来一个小前提：IO密集型的程序，阻塞时间一般远大于计算时间。

于是，IO密集型程序多线程运行情况如下（下图没有时间片的概念，反正会被CPU提前终止）：



问题来了：从ThreadD第一次计算结束，到ThreadA第一次阻塞终止，中间这段时间CPU是空闲的。四个线程全部阻塞了！

所以，为了最大化利用CPU的计算资源，我们希望所有时刻CPU都是在计算的。于是能想到如下公式：

线程数 = 

这样就会保证，当最后一个线程刚刚发出查询SQL的请求，第一个线程恰好获取到SQL结果，阻塞停止。完美衔接。

完善一下，如果有多个CPU核心，则需要乘以CPU核心数量；如果不希望占用全部CPU资源，还要乘以期望CPU占用率。这样，就得到上次讲的公式：

Nthreads = Ncpu \* Ucpu \* (1 + W/C)

其中：

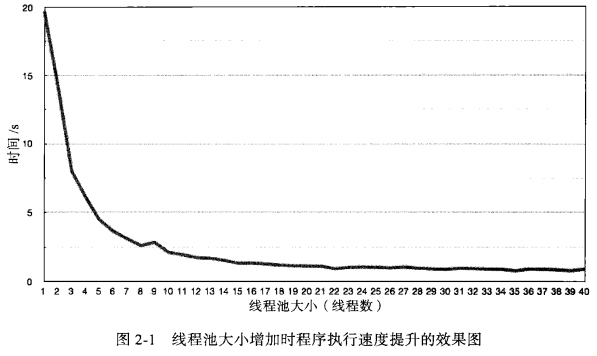
❑Ncpu是处理器的核的数目

❑Ucpu是期望的CPU利用率（该值应该介于0和1之间）

❑W/C是等待时间与计算时间的比率

### 实际效果

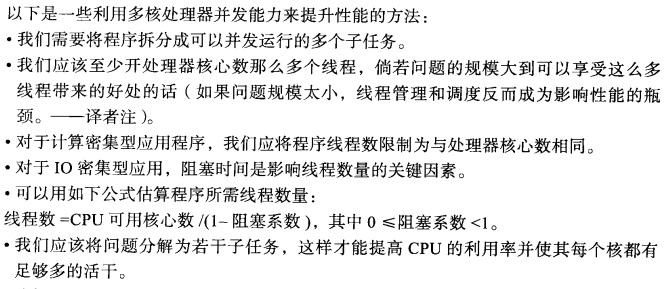
下图的任务数据：2核机器，共40个任务。



线程数=20是一个分界点，20之前只要增加线程数，都会提高性能。20之后基本就是水平线了（因为没有那么多任务去跑，会有空闲线程）。

这种阻塞时间远大于计算时间的任务，就应该线程数越大越好，最好让所有任务一次性发出命令，一起阻塞住。这是最省时间的策略。

## 总结



参考资料：

《Java虚拟机并发编程》第二章

《Java并发编程实战》8.2节