

## 18 | 如何设置线程池大小？

2019-06-29 刘超

Java性能调优实战

[进入课程 >](#)



讲述：李良

时长 11:05 大小 10.16M



你好，我是刘超。

还记得我在 16 讲中说过“线程池的线程数量设置过多会导致线程竞争激烈”吗？今天再补一句，如果线程数量设置过少的话，还会导致系统无法充分利用计算机资源。那么如何设置才不会影响系统性能呢？

其实线程池的设置是有方法的，不是凭借简单的估算来决定的。今天我们就来看看究竟有哪些计算方法可以复用，线程池中各个参数之间又存在怎样的关系。

### 线程池原理

开始优化之前，我们先来看看线程池的实现原理，有助于你更好地理解后面的内容。

在 HotSpot VM 的线程模型中，Java 线程被一对一映射为内核线程。Java 在使用线程执行程序时，需要创建一个内核线程；当该 Java 线程被终止时，这个内核线程也会被回收。因此 Java 线程的创建与销毁将会消耗一定的计算机资源，从而增加系统的性能开销。

除此之外，大量创建线程同样会给系统带来性能问题，因为内存和 CPU 资源都将被线程抢占，如果处理不当，就会发生内存溢出、CPU 使用率超负荷等问题。

为了解决上述两类问题，Java 提供了线程池概念，对于频繁创建线程的业务场景，线程池可以创建固定的线程数量，并且在操作系统底层，轻量级进程将会把这些线程映射到内核。

线程池可以提高线程复用，又可以固定最大线程使用量，防止无限制地创建线程。当程序提交一个任务需要一个线程时，会去线程池中查找是否有空闲的线程，若有，则直接使用线程池中的线程工作，若没有，会去判断当前已创建的线程数量是否超过最大线程数量，如未超过，则创建新线程，如已超过，则进行排队等待或者直接抛出异常。

## 线程池框架 Executor

Java 最开始提供了 ThreadPool 实现了线程池，为了更好地实现用户级的线程调度，更有效地帮助开发人员进行多线程开发，Java 提供了一套 Executor 框架。

这个框架中包括了 ScheduledThreadPoolExecutor 和 ThreadPoolExecutor 两个核心线程池。前者是用来定时执行任务，后者是用来执行被提交的任务。鉴于这两个线程池的核心原理是一样的，下面我们就重点看看 ThreadPoolExecutor 类是如何实现线程池的。

Executors 实现了以下四种类型的 ThreadPoolExecutor：

类型	特性
newCachedThreadPool	线程池的大小不固定，可灵活回收空闲线程，若无可回收，则新建线程
newFixedThreadPool	固定大小的线程池，当有新的任务提交，线程池中如果有空闲线程，则立即执行，否则新的任务会被缓存在一个任务队列中，等待线程池释放空闲线程
newScheduledThreadPool	定时线程池，支持定时及周期性任务执行
newSingleThreadExecutor	只创建一个线程，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序 (FIFO-LIFO-优先级) 执行

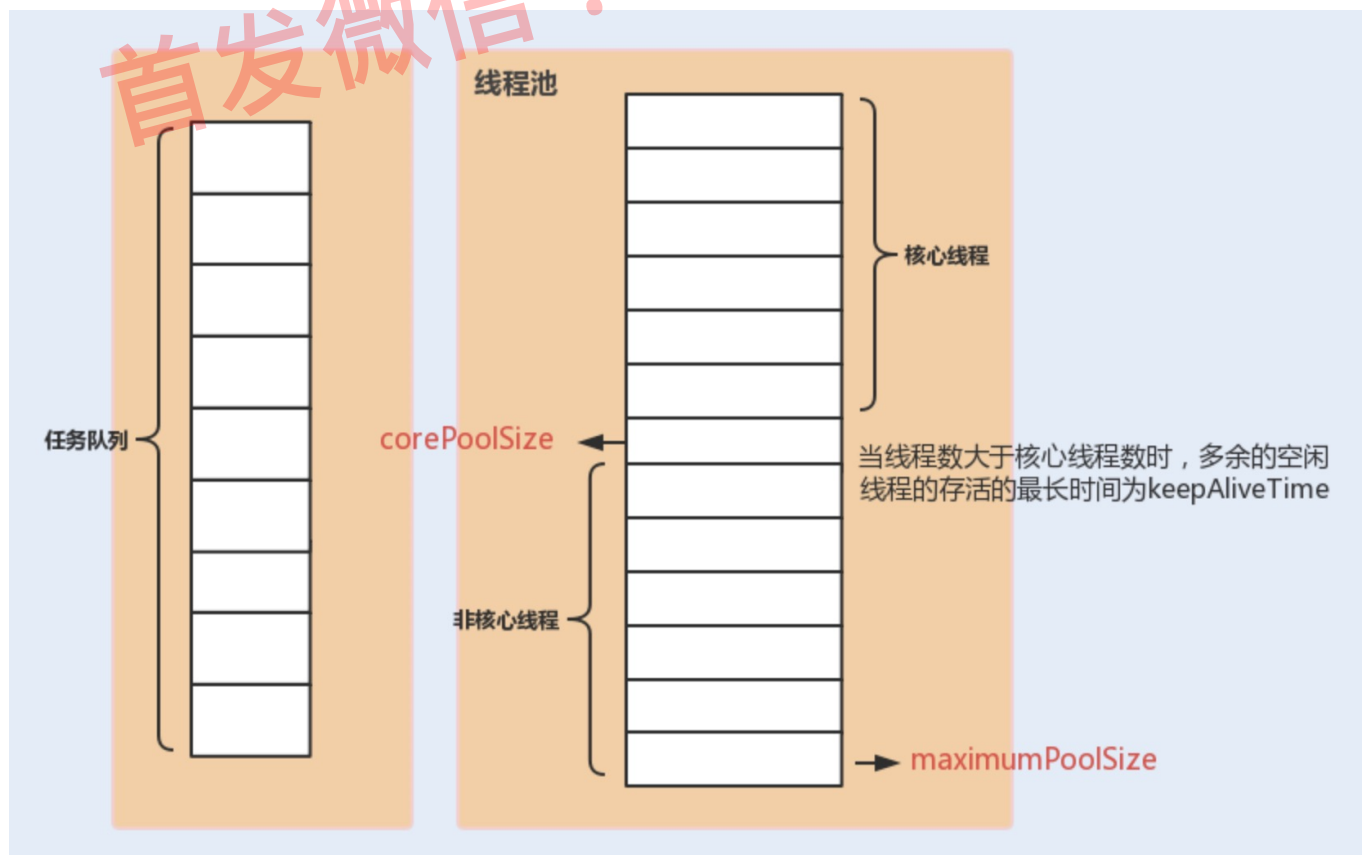
Executors 利用工厂模式实现的四种线程池，我们在使用的时候需要结合生产环境下的实际场景。不过我不太推荐使用它们，因为选择使用 Executors 提供的工厂类，将会忽略很多线程池的参数设置，工厂类一旦选择设置默认参数，就很容易导致无法调优参数设置，从而产生性能问题或者资源浪费。

这里我建议你使用 `ThreadPoolExecutor` 自我定制一套线程池。进入四种工厂类后，我们可以发现除了 `newScheduledThreadPool` 类，其它类均使用了 `ThreadPoolExecutor` 类进行实现，你可以通过以下代码简单看下该方法：

复制代码

```
1 public ThreadPoolExecutor(int corePoolSize, // 线程池的核心线程数量
2                           int maximumPoolSize, // 线程池的最大线程数
3                           long keepAliveTime, // 当线程数大于核心线程数时，多余的空闲线程
4                           TimeUnit unit, // 时间单位
5                           BlockingQueue<Runnable> workQueue, // 任务队列，用来储存等待
6                           ThreadFactory threadFactory, // 线程工厂，用来创建线程，一般默
7                           RejectedExecutionHandler handler) // 拒绝策略，当提交的任务
```

我们还可以通过下面这张图来了解下线程池中各个参数的相互关系：



通过上图，我们发现线程池有两个线程数的设置，一个为核心线程数，一个为最大线程数。在创建完线程池之后，默认情况下，线程池中并没有任何线程，等到有任务来才创建线程去执行任务。

但有一种情况排除在外，就是调用 `prestartAllCoreThreads()` 或者 `prestartCoreThread()` 方法的话，可以提前创建等于核心线程数的线程数量，这种方式被称为预热，在抢购系统中就经常被用到。

当创建的线程数等于 `corePoolSize` 时，提交的任务会被加入到设置的阻塞队列中。当队列满了，会创建线程执行任务，直到线程池中的数量等于 `maximumPoolSize`。

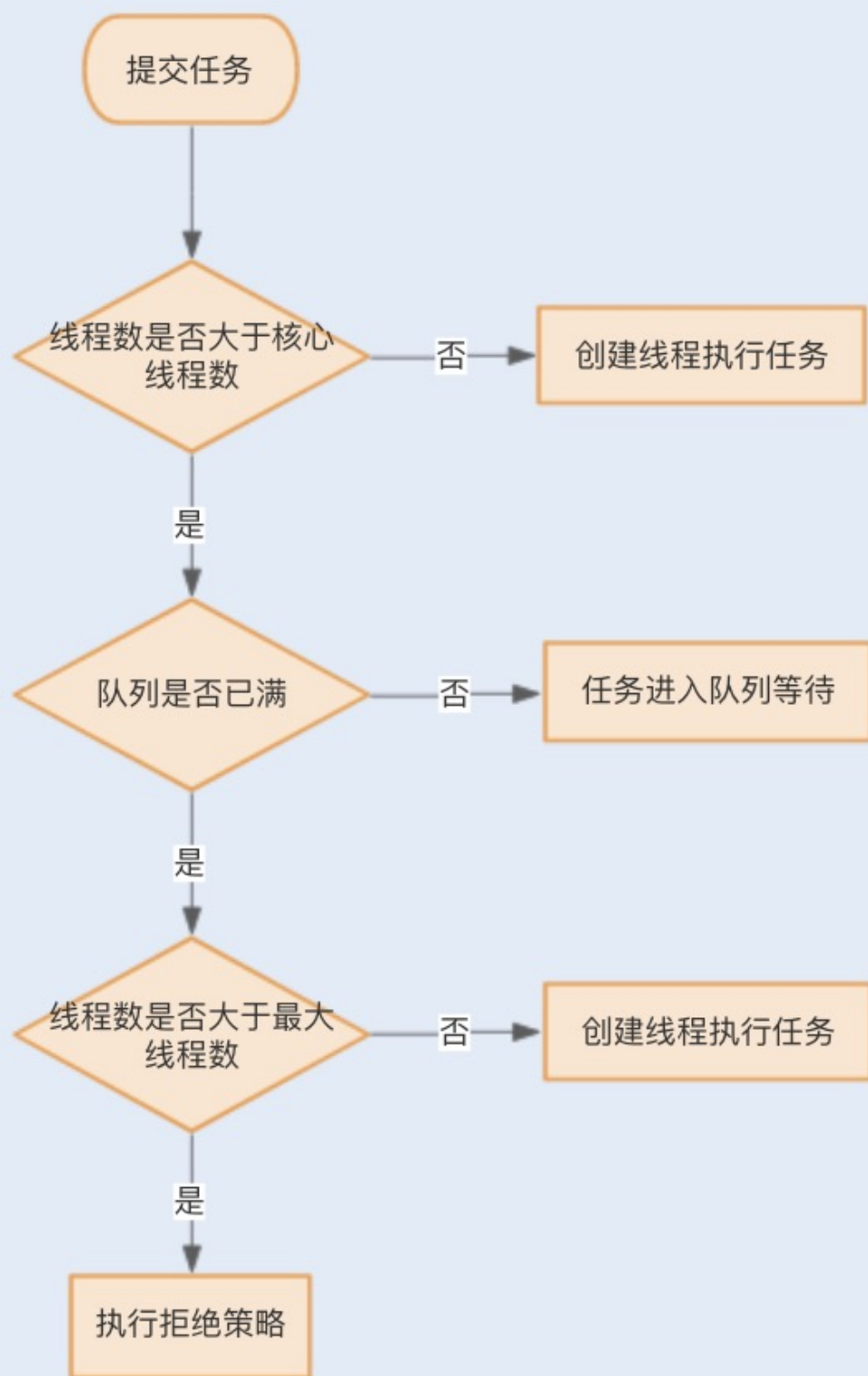
当线程数量已经等于 `maximumPoolSize` 时，新提交的任务无法加入到等待队列，也无法创建非核心线程直接执行，我们又没有为线程池设置拒绝策略，这时线程池就会抛出 `RejectedExecutionException` 异常，即线程池拒绝接受这个任务。

当线程池中创建的线程数量超过设置的 `corePoolSize`，在某些线程处理完任务后，如果等待 `keepAliveTime` 时间后仍然没有新的任务分配给它，那么这个线程将会被回收。线程池回收线程时，会对所谓的“核心线程”和“非核心线程”一视同仁，直到线程池中线程的数量等于设置的 `corePoolSize` 参数，回收过程才会停止。

即使是 `corePoolSize` 线程，在一些非核心业务的线程池中，如果长时间地占用线程数量，也可能会影响到核心业务的线程池，这个时候就需要把没有分配任务的线程回收掉。

我们可以通过 `allowCoreThreadTimeOut` 设置项要求线程池：将包括“核心线程”在内的，没有任务分配的所有线程，在等待 `keepAliveTime` 时间后全部回收掉。

我们可以通过下面这张图来了解下线程池的线程分配流程：



## 计算线程数量

了解完线程池的实现原理和框架，我们就可以动手实践优化线程池的设置了。




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。

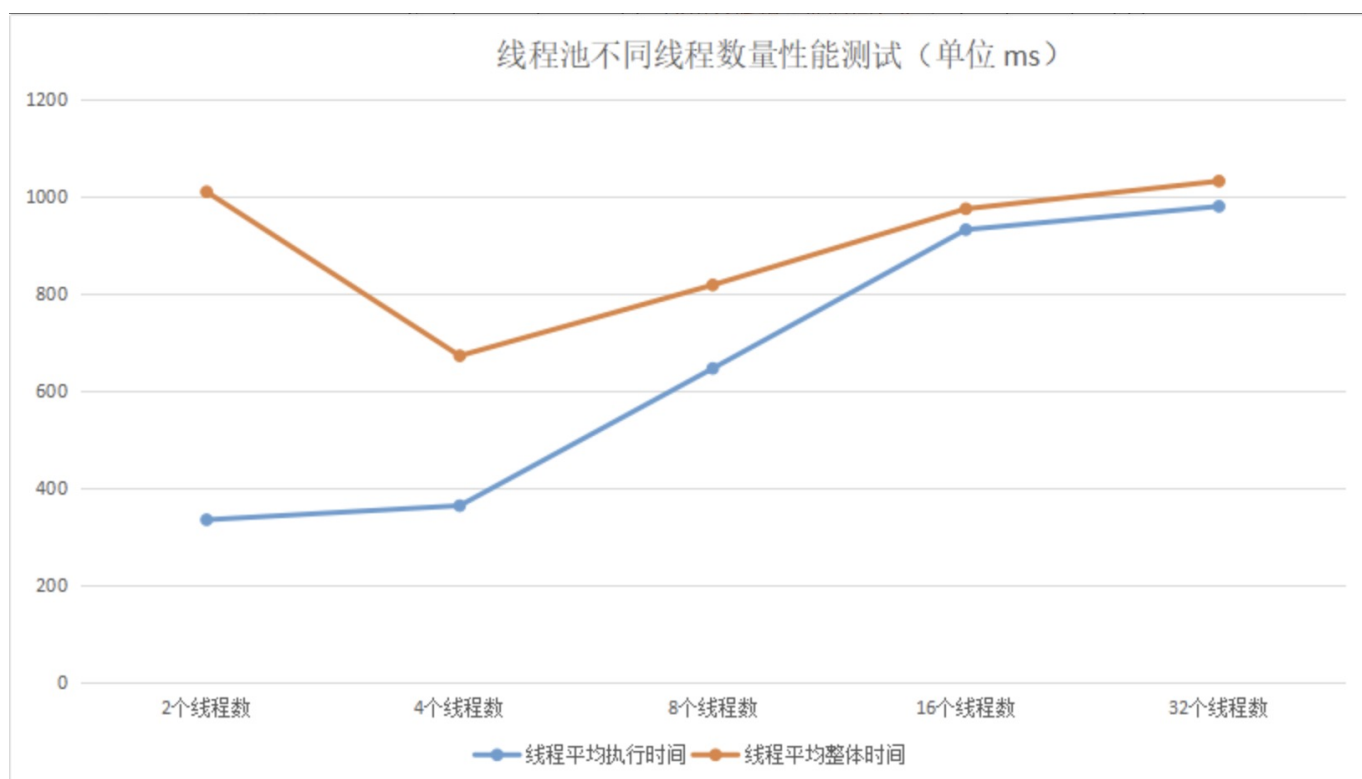
下面我们用一个例子来验证下这个方法的可行性，通过观察 CPU 密集型任务在不同线程数下的性能情况就可以得出结果，你可以点击[Github](#)下载到本地运行测试：

```
1 public class CPUTypeTest implements Runnable {
2
3     // 整体执行时间，包括在队列中等待的时间
4     List<Long> wholeTimeList;
5     // 真正执行时间
6     List<Long> runTimeList;
7
8     private long initStartTime = 0;
9
10    /**
11     * 构造函数
12     * @param runTimeList
13     * @param wholeTimeList
14     */
15    public CPUTypeTest(List<Long> runTimeList, List<Long> wholeTimeList) {
16        initStartTime = System.currentTimeMillis();
17        this.runTimeList = runTimeList;
18        this.wholeTimeList = wholeTimeList;
19    }
20
21    /**
22     * 判断素数
23     * @param number
24     * @return
25     */
26    public boolean isPrime(final int number) {
27        if (number <= 1)
28            return false;
```

 复制代码

```
29
30
31         for (int i = 2; i <= Math.sqrt(number); i++) {
32             if (number % i == 0)
33                 return false;
34         }
35         return true;
36     }
37
38     /**
39     * 计算素数
40     * @param number
41     * @return
42     */
43     public int countPrimes(final int lower, final int upper) {
44         int total = 0;
45         for (int i = lower; i <= upper; i++) {
46             if (isPrime(i))
47                 total++;
48         }
49         return total;
50     }
51
52     public void run() {
53         long start = System.currentTimeMillis();
54         countPrimes(1, 1000000);
55         long end = System.currentTimeMillis();
56
57
58         long wholeTime = end - initStartTime;
59         long runTime = end - start;
60         wholeTimeList.add(wholeTime);
61         runTimeList.add(runTime);
62         System.out.println(" 单个线程花费时间: " + (end - start));
63     }
64 }
```


测试代码在 4 核 intel i5 CPU 机器上的运行时间变化如下：



综上所述：当线程数量太小，同一时间大量请求将被阻塞在线程队列中排队等待执行线程，此时 CPU 没有得到充分利用；当线程数量太大，被创建的执行线程同时在争取 CPU 资源，又会导致大量的上下文切换，从而增加线程的执行时间，影响了整体执行效率。通过测试可知，4~6 个线程数是最合适的。

**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  $2N$ 。

这里我们还是通过一个例子来验证下这个公式是否可以标准化：

 复制代码

```
1 public class IOTypeTest implements Runnable {
2
3     // 整体执行时间，包括在队列中等待的时间
4     Vector<Long> wholeTimeList;
5     // 真正执行时间
6     Vector<Long> runTimeList;
7
8     private long initStartTime = 0;
9
10    /**
11     * 构造函数
12     * @param runTimeList
13     * @param wholeTimeList
```



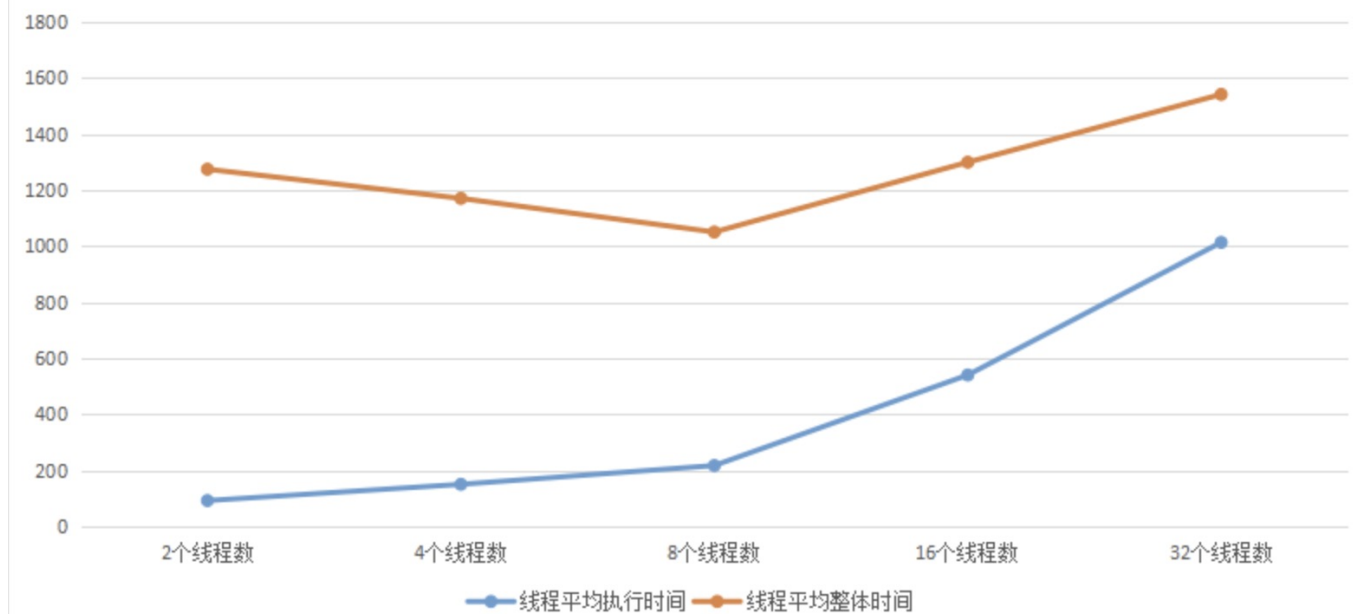
```

14     */
15     public IOTypeTest(Vector<Long> runTimeList, Vector<Long> wholeTimeList) {
16         initStartTime = System.currentTimeMillis();
17         this.runTimeList = runTimeList;
18         this.wholeTimeList = wholeTimeList;
19     }
20
21     /**
22      * IO 操作
23      * @param number
24      * @return
25      * @throws IOException
26      */
27     public void readAndWrite() throws IOException {
28         File sourceFile = new File("D:/test.txt");
29         // 创建输入流
30         BufferedReader input = new BufferedReader(new FileReader(sourceFile));
31         // 读取源文件，写入到新的文件
32         String line = null;
33         while((line = input.readLine()) != null){
34             //System.out.println(line);
35         }
36         // 关闭输入输出流
37         input.close();
38     }
39
40     public void run() {
41         long start = System.currentTimeMillis();
42         try {
43             readAndWrite();
44         } catch (IOException e) {
45             // TODO Auto-generated catch block
46             e.printStackTrace();
47         }
48         long end = System.currentTimeMillis();
49
50
51         long wholeTime = end - initStartTime;
52         long runTime = end - start;
53         wholeTimeList.add(wholeTime);
54         runTimeList.add(runTime);
55         System.out.println(" 单个线程花费时间: " + (end - start));
56     }
57 }

```

备注：由于测试代码读取 2MB 大小的文件，涉及到大内存，所以在运行之前，我们需要调整 JVM 的堆内存空间：-Xms4g -Xmx4g，避免发生频繁的 FullGC，影响测试结果。

I/O密集型线程池不同线程数量性能测试（单位 ms）



通过测试结果，我们可以看到每个线程所花费的时间。当线程数量在 8 时，线程平均执行时间是最佳的，这个线程数量和我们的计算公式所得的结果就差不多。

看完以上两种情况下的线程计算方法，你可能还想说，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：

复制代码

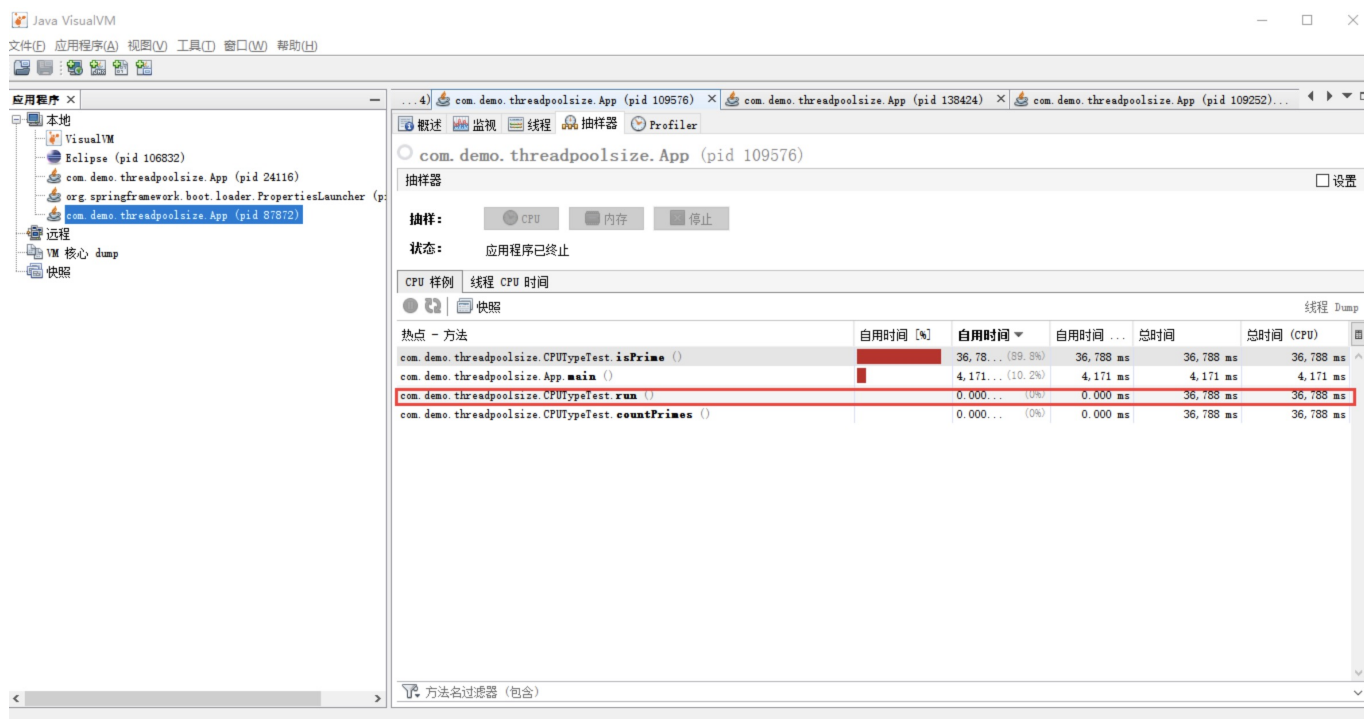
```
1 线程数 = N (CPU 核数) * (1 + WT (线程等待时间) / ST (线程时间运行时间))
```

我们可以通过 JDK 自带的工具 VisualVM 来查看 WT/ST 比例，以下例子是基于运行纯 CPU 运算的例子，我们可以看到：

复制代码

```
1 WT (线程等待时间) = 36788ms [线程运行总时间] - 36788ms [ST (线程时间运行时间)] = 0
2 线程数 = N (CPU 核数) * (1 + 0 [WT (线程等待时间)] / 36788ms [ST (线程时间运行时间)]) = N (CPU ;
```

这跟我们之前通过 CPU 密集型的计算公式  $N+1$  所得出的结果差不多。



综合来看，我们可以根据自己的业务场景，从“ $N+1$ ”和“ $2N$ ”两个公式中选出一个适合的，计算出一个大概的线程数量，之后通过实际压测，逐渐往“增大线程数量”和“减小线程数量”这两个方向调整，然后观察整体的处理时间变化，最终确定一个具体的线程数量。

## 总结

今天我们主要学习了线程池的实现原理，Java 线程的创建和消耗会给系统带来性能开销，因此 Java 提供了线程池来复用线程，提高程序的并发效率。

Java 通过用户线程与内核线程结合的 1:1 线程模型来实现，Java 将线程的调度和管理设置在了用户态，提供了一套 Executor 框架来帮助开发人员提高效率。Executor 框架不仅包括了线程池的管理，还提供了线程工厂、队列以及拒绝策略等，可以说 Executor 框架为并发编程提供了一个完善的架构体系。

在不同的业务场景以及不同配置的部署机器中，线程池的线程数量设置是不一样的。其设置不宜过大，也不宜过小，要根据具体情况，计算出一个大概的数值，再通过实际的性能测试，计算出一个合理的线程数量。

我们要提高线程池的处理能力，一定要先保证一个合理的线程数量，也就是保证 CPU 处理线程的最大化。在此前提下，我们再增大线程池队列，通过队列将来不及处理的线程缓存起

来。在设置缓存队列时，我们要尽量使用一个有界队列，以防因队列过大而导致的内存溢出问题。

## 思考题

在程序中，除了并行段代码，还有串行段代码。那么当程序同时存在串行和并行操作时，优化并行操作是不是优化系统的关键呢？

期待在留言区看到你的见解。也欢迎你点击“请朋友读”，把今天的内容分享给身边的朋友，邀请他一起讨论。

 极客时间

# Java 性能调优实战

覆盖 80% 以上 Java 应用调优场景

刘超  
金山软件西山居技术经理



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 17 | 并发容器的使用：识别不同场景下最优容器

## 精选留言 (16)

 写留言



QQ怪

2019-06-29

老师，我想问下生产环境情况下很少碰到单个java进程在一台主机中运行，大部分肯定是多个进程同时运行，不如docker技术，都是共享同一套硬件，那这套计算方程式是不是不适用了？

作者回复: 适用的，多个进程大部分时间不一定是重合运行的。但具体情况需要具体定，所以最终还是以压测调出来的线程数为准。



1



nico

2019-06-29

老师，请教个问题，生产环境有些应用是混部的，即一个虚拟机上跑很多个java程序，这个时候估算一个程序中的线程池的线程数，是不是就不合理了？这个要怎么估算合理的线程池配置？还有就是即使是单实例部署，cpu资源是机器内共用的，不可能只分配给java线程，这个要怎么考虑？

作者回复: 我们先考虑单个环境，再去调优复杂环境。大多情况下，重要的服务会单独部署，尽量减少重要业务的相互影响。如果是核心业务冗余在了一个服务上，建议拆分之后分别部署。

非核心业务，很多业务处理可能是在不同时间点，彼此相互不影响，所以不用过多考虑混合部署服务的情况。



1



张学磊

2019-06-29

老师，线程池流程图提交任务后的第一个节点应该是线程数是否大于核心线程数，如果是再判断队列是否已满，否则直接创建新线程。

思考题，个人觉得线性执行的代码会成为影响性能的关键，应尽量减少执行时间，比如减少锁持有时间，这样才能达到最大程度的并发。

作者回复: 感谢学磊童鞋的提醒，已修正。

答案正确！



1



赵衍

2019-06-30

老师好！关于线程池我有一个问题一直不明白，在线程池达到了核心线程数，等待队列没满的这段时间，新的任务会被加入到等待队列。而当等待队列满了之后，最大线程数没满的这段时间，线程池会为新的任务直接创建线程。那岂不是说，我后来的任务反而比先到的任务更早被分配到线程的资源？这是不是有点不太合理呢？



飞翔

2019-06-30

话说N+1和2N 是指的核心线程数嘛？那队列和最大线程数怎么设置呀

作者回复: 在一些非核心业务，我们可以将核心线程数设置小一些，最大线程数量设置为计算线程数量。在一些核心业务中，两者可以设置一样。阻塞队列可以根据具体业务场景设置，如果线程处理业务非常迅速，我们可以考虑将阻塞队列设置大一些，处理的请求吞吐量会大些；如果线程处理业务非常耗时，阻塞队列设置小些，防止请求在阻塞队列中等待过长时间而导致请求已超时。



WL

2019-06-29

有三个问题请教一下老师:

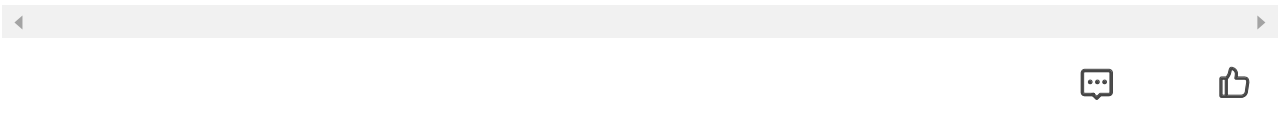
1. 守护线程也会映射到CPU的线程吗, 我还是没太理解在计算线程数量的时候为啥不考虑守护线程呢, 是因为守护线程运行状态是要么一直运行, 要么一直等待这样不会变化的情况所以不考虑吗? 为啥守护线程的状态一直不变化呢, 它们不会出让CPU吗?
2. 文章中讲的主要是核心线程的数量如何配置, 最大线程数量和阻塞队列长度应该如何确...

作者回复: 1、守护线程也是JVM创建的线程，这块我没有具体看到源码。我们一般不会使用守护线程处理业务，我们这里讨论的是处理业务的线程数量。

2、在一些非核心业务，我们可以将核心线程数设置小一些，最大线程数量设置为计算线程数量。在一些核心业务中，两者可以设置一样。阻塞队列可以根据具体业务场景设置，如果线程处理业务非常迅速，我们可以考虑将阻塞队列设置大一些，处理的请求吞吐量会大些；如果线程处理业务非常耗时，阻塞队列设置小些，防止请求在阻塞队列中等待过长时间而导致请求已超时。



3、休眠对应的sleep操作，等待对应的wait，驻留对应的线程池里的空闲线程，监视对应的synchronized阻塞

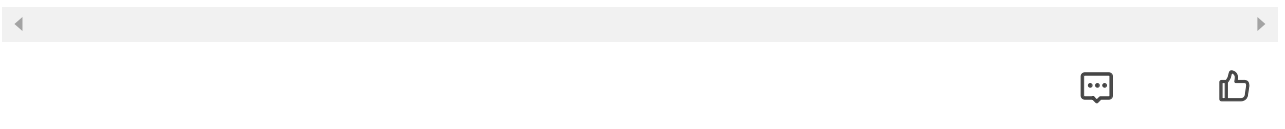


**杨俊**

2019-06-29

老是我可以这样理解吗，阻塞队列满了之后创建的是非核心线程而不是核心线程，非核心线程执行完可能空闲一段时间就会被销毁，除非达到最大线程数，否则当阻塞时候有任务就会创建非核心线程，那是不是有可能这个非核心线程会比阻塞队列里面的先执行任务呢

作者回复: 最后这个问题我没有太理解。非核心线程执行的任务可能是非阻塞队列的任务？

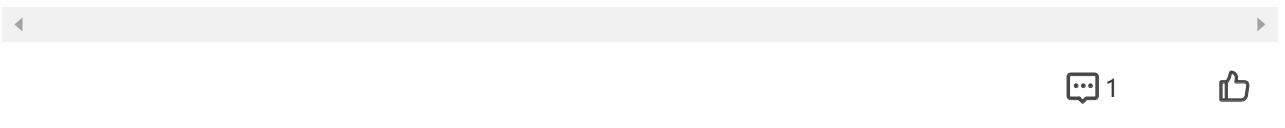


**我期待的** @AR

2019-06-29

刘老师你好，我想问下测试结果的可视化图表是如何制作的？

作者回复: 通过excel制作的



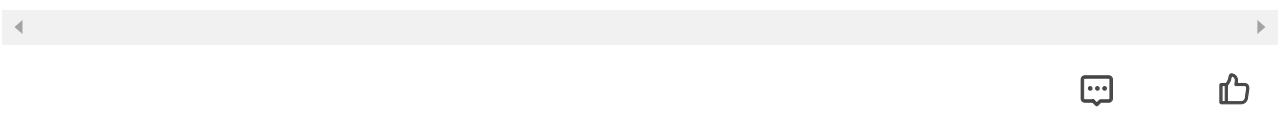
**colin**

2019-06-29

线程分配流程图的第一个分支应该是大于吧

线程池我的理解就是，尽量使用核心线程。能等到核心线程的就不创建非核心线程。有长时间不工作的线程，那么就回收它，保证线程数量在核心线程数量内

作者回复: 是的，已修正。理解到位





许童童

2019-06-29

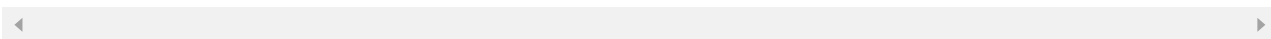
优化并行操作是不是优化系统的关键呢？

可以参考阿姆达尔定律

$$S=1/(1-a+a/n)$$

总之，优化并行操作带来的收益是有上限的。

作者回复: 赞，Amdahl's定律指出优化串行是优化系统性能的关键，我们应该从算法入手，减少程序中串行的部分，而不是增加线程数来提高系统的并发处理能力。

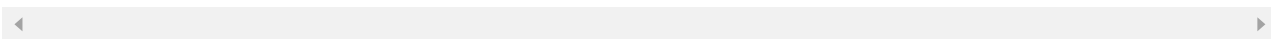


听雨

2019-06-29

老师，程序很快就跑完了，visualvm还没打开程序呢，我看你的图监视的是已终止的程序，那怎么显示已终止程序的信息的呢

作者回复: 用过debug模式，在main函数第一条语句设置一个断点，进入断点后，再去VisualVM操作CPU采样，之后再运行程序。



听雨

2019-06-29

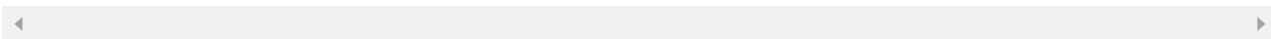
老师，您说的java线程和内核线程一一对应。那我想的是：

- 1.不管java线程是否阻塞，只要没有销毁，对应的内核线程就一直存在，是这样吗？
- 2.如果java线程已经从内核线程拿到数据，但是还没执行完业务流程，现在对应的内核线程明显已经无事可干了，也会一直存在吗？存在的话，略显浪费呀。

请老师解答一下！

作者回复: 1、内核线程一直存在的，是通过调度器进行调度给用户线程使用的。

2、是的，如果大量用户线程创建，长时间不释放，会受限内核线程资源而影响系统性能。





**-W.LI-**

2019-06-29

老师我有个问题想问。

程序里面不止一个线程池啊，然后又有那么多进程线程数这么设真的可以么。

之前做http调用4000请求并发。测试下来开到50个线程时16S完成(最快)4核CPU。调用第三方花费时间差不多一次100ms。按照那个公式不是8个线程最快么。

作者回复: 可以的，这两个公式已经在生产环境中验证过的。



**-W.LI-**

2019-06-29

老师好!能问点操作系统方面的问题么?全忘记了。

内核线程:会被cpu分配执行时间的线程。

用户线程:进程管理的线程，不参与CPU执行时间分配。

Linux操作系统其实只有进程的概念。创建的线程都是fork一个子进程然后，把设置一些线程私有的东西。因为是fork的所以同一个进程的线程之间。进程的内存是可见的。...

作者回复: 这块19讲会帮你解答，建议留意查看。



**Liam**

2019-06-29

请教老师一个问题：

对于应用而言，可能有多种类型的任务要执行，我们是分别创建不同的线程池还是创建一个统一的线程池来控制资源的使用呢？

...

作者回复: 分别创建。

如果过分彼此相互影响，建议拆开服务，分别部署。





木木匠

2019-06-29

思考题:此时优化串行代码是关键，串行代码一般是遇到了锁或阻塞IO，这个时候串行的效率也决定了整个程序的效率。

作者回复: 对的

