二

18 如何设置线程池大小?

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：

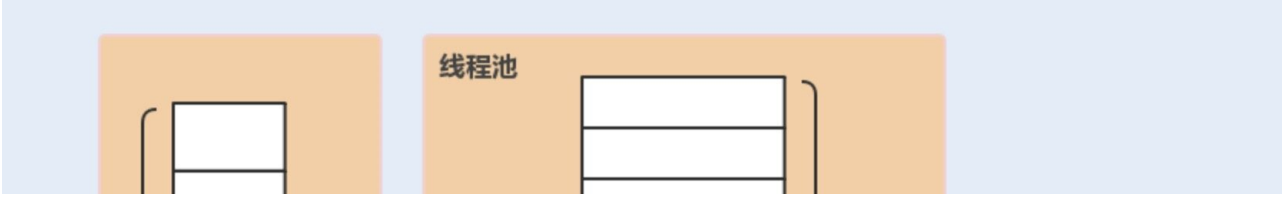
类型	特性
newCachedThreadPool	线程池的大小不固定，可灵活回收空闲线程，若无可回收，则新建线程
newFixedThreadPool	固定大小的线程池，当有新的任务提交，线程池中如果有空闲线程，则立即执行，否则新的任务会被缓存在一个任务队列中，等待线程池释放空闲线程
newScheduledThreadPool	定时线程池，支持定时及周期性任务执行
newSingleThreadExecutor	只创建一个线程，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序 (FIFO-LIFO-优先级)执行

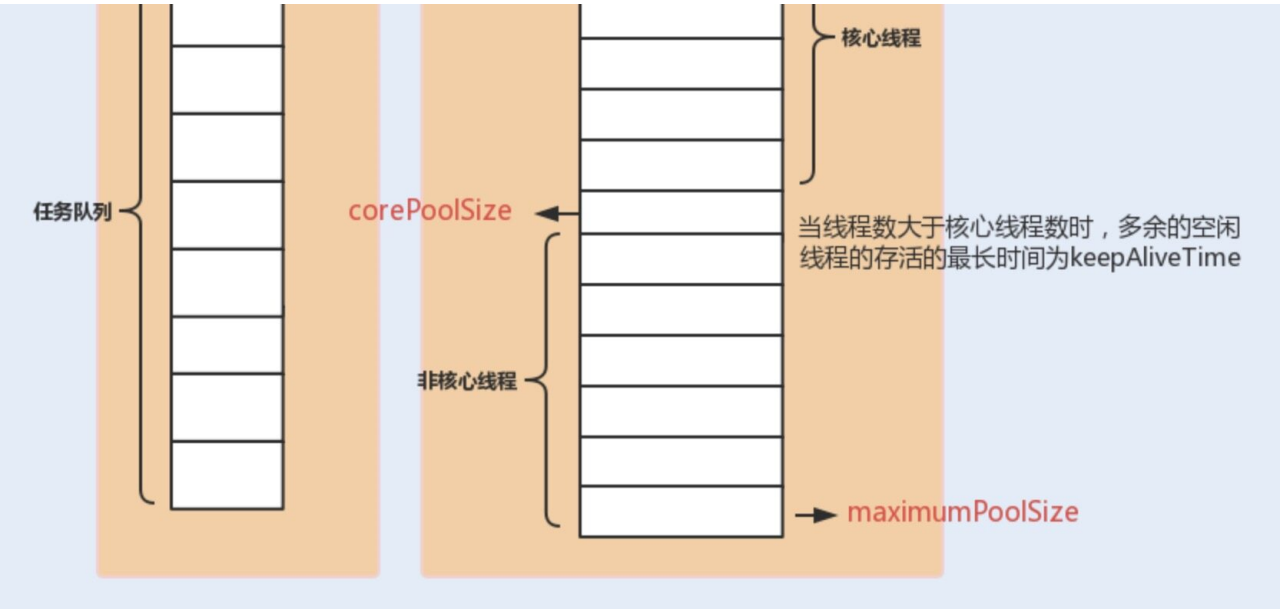
Executors 利用工厂模式实现的四种线程池，我们在使用的时候需要结合生产环境下的实际场景。不过我不太推荐使用它们，因为选择使用 Executors 提供的工厂类，将会忽略很多线程池的参数设置，工厂类一旦选择设置默认参数，就很容易导致无法调优参数设置，从而产生性能问题或者资源浪费。

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：

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

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。

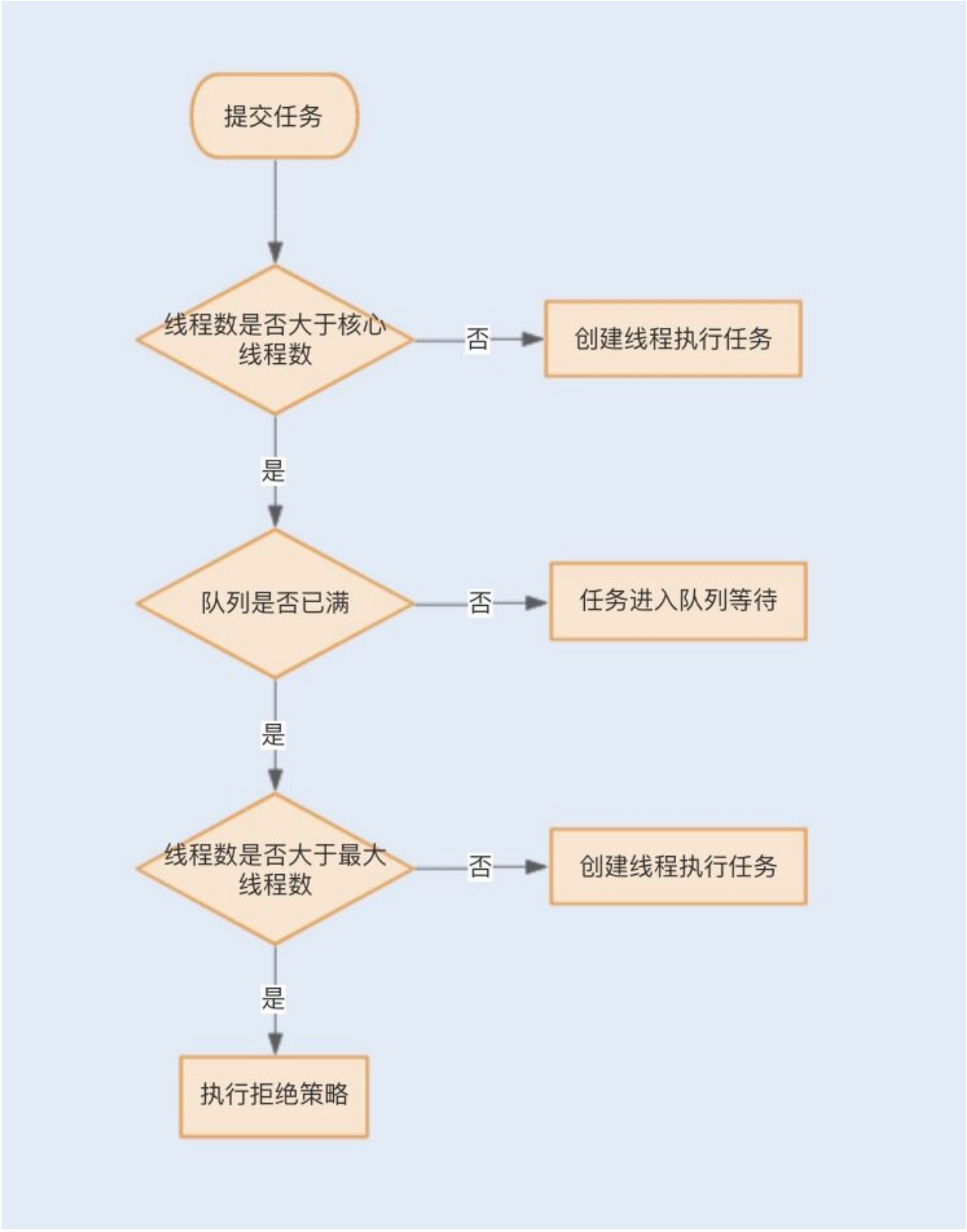
当线程数量已经等于 maximumPoolSize 时， 新提交的任务无法加入到等待队列，也无法创建非核心线程直接执行，我们又没有为线程池设置拒绝策略，这时线程池就会抛出 RejectedExecutionException 异常，即线程池拒绝接受这个任务。

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。

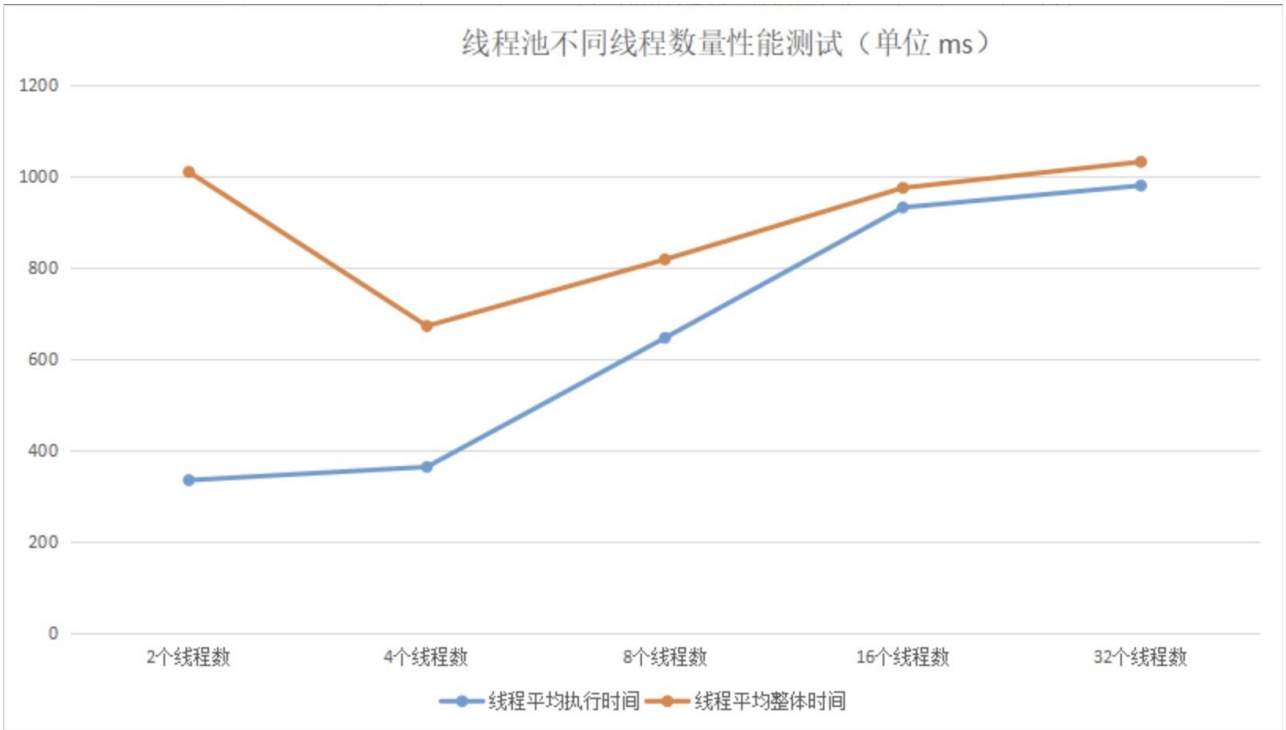
CPU 密集型任务：这种任务消耗的主要是 CPU 资源，可以将线程数设置为 N（CPU 核心数）+1，比 CPU 核心数多出来的一个线程是为了防止线程偶发的缺页中断，或者其它原因导致的任务暂停而带来的影响。一旦任务暂停，CPU 就会处于空闲状态，而在这种情况下多出来的一个线程就可以充分利用 CPU 的空闲时间。

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：

```
public class CPUTypeTest implements Runnable {

        // 整体执行时间，包括在队列中等待的时间
        List<Long> wholeTimeList;
        // 真正执行时间
        List<Long> runTimeList;

        private long initStartTime = 0;

        /**
         * 构造函数
         * @param runTimeList
         * @param wholeTimeList
         */
        public CPUTypeTest(List<Long> runTimeList, List<Long> wholeTimeList) {
                initStartTime = System.currentTimeMillis();
                this.runTimeList = runTimeList;
                this.wholeTimeList = wholeTimeList;
        }

        /**
         * 判断素数
         * @param number
         * @return
         */
        public boolean isPrime(final int number) {
                if (number <= 1)
                        return false;


                for (int i = 2; i <= Math.sqrt(number); i++) {
                        if (number % i == 0)
                                return false;
                }
                return true;
```

```
        }

        /**
         * 計算素数
         * @param number
         * @return
         */
        public int countPrimes(final int lower, final int upper) {
                int total = 0;
                for (int i = lower; i <= upper; i++) {
                        if (isPrime(i))
                                total++;
                }
                return total;
        }

        public void run() {
                long start = System.currentTimeMillis();
                countPrimes(1, 1000000);
                long end = System.currentTimeMillis();


                long wholeTime = end - initStartTime;
                long runTime = end - start;
                wholeTimeList.add(wholeTime);
                runTimeList.add(runTime);
                System.out.println(" 单个线程花费时间： " + (end - start));
        }
}
```

测试代码在 4 核 intel i5 CPU 机器上的运行时间变化如下：



综上可知：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。

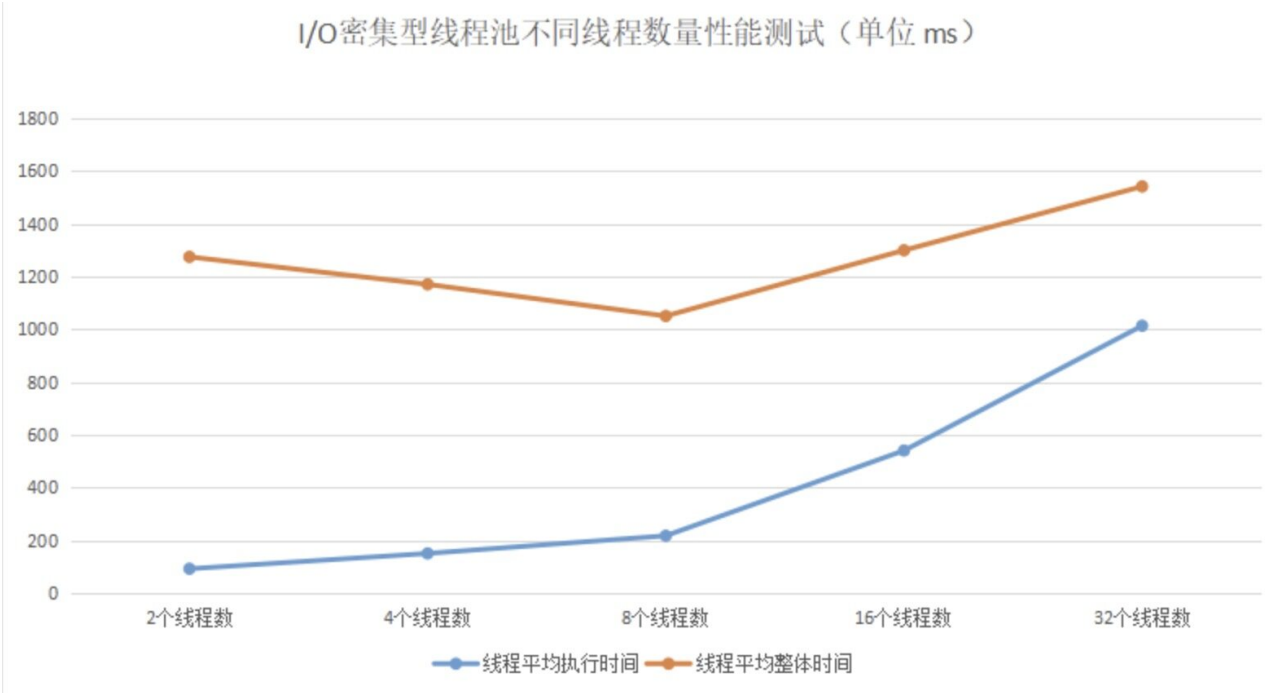
I/O 密集型任务：这种任务应用起来，系统会用大部分的时间来处理 I/O 交互，而线程在处理 I/O 的时间段内不会占用 CPU 来处理，这时就可以将 CPU 交出给其它线程使用。因此在 I/O 密集型任务的应用中，我们可以多配置一些线程，具体的计算方法是 2N。

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：

```
public class IOTypeTest implements Runnable {

        // 整体执行时间，包括在队列中等待的时间
        Vector<Long> wholeTimeList;
        // 真正执行时间
        Vector<Long> runTimeList;

        private long initStartTime = 0;

        /**
         * 构造函数
         * @param runTimeList
         * @param wholeTimeList
         */
        public IOTypeTest(Vector<Long> runTimeList, Vector<Long> wholeTimeList) {
                initStartTime = System.currentTimeMillis();
                this.runTimeList = runTimeList;
                this.wholeTimeList = wholeTimeList;
        }

        /**
         *IO 操作
         * @param number
         * @return
         * @throws IOException
         */
        public void readAndWrite() throws IOException {
                File sourceFile = new File("D:/test.txt");
        // 创建输入流
        BufferedReader input = new BufferedReader(new FileReader(sourceFile));
        // 读取源文件，写入到新的文件
        String line = null;
        while((line = input.readLine()) != null){
            //System.out.println(line);
        }
        // 关闭输入输出流
        input.close();
        }

        public void run() {
                long start = System.currentTimeMillis();
```

```
                try {
                        readAndWrite();
                } catch (IOException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
                long end = System.currentTimeMillis();


                long wholeTime = end - initStartTime;
                long runTime = end - start;
                wholeTimeList.add(wholeTime);
                runTimeList.add(runTime);
                System.out.println(" 单个线程花费时间：" + (end - start));
        }
}
```

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。



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?**

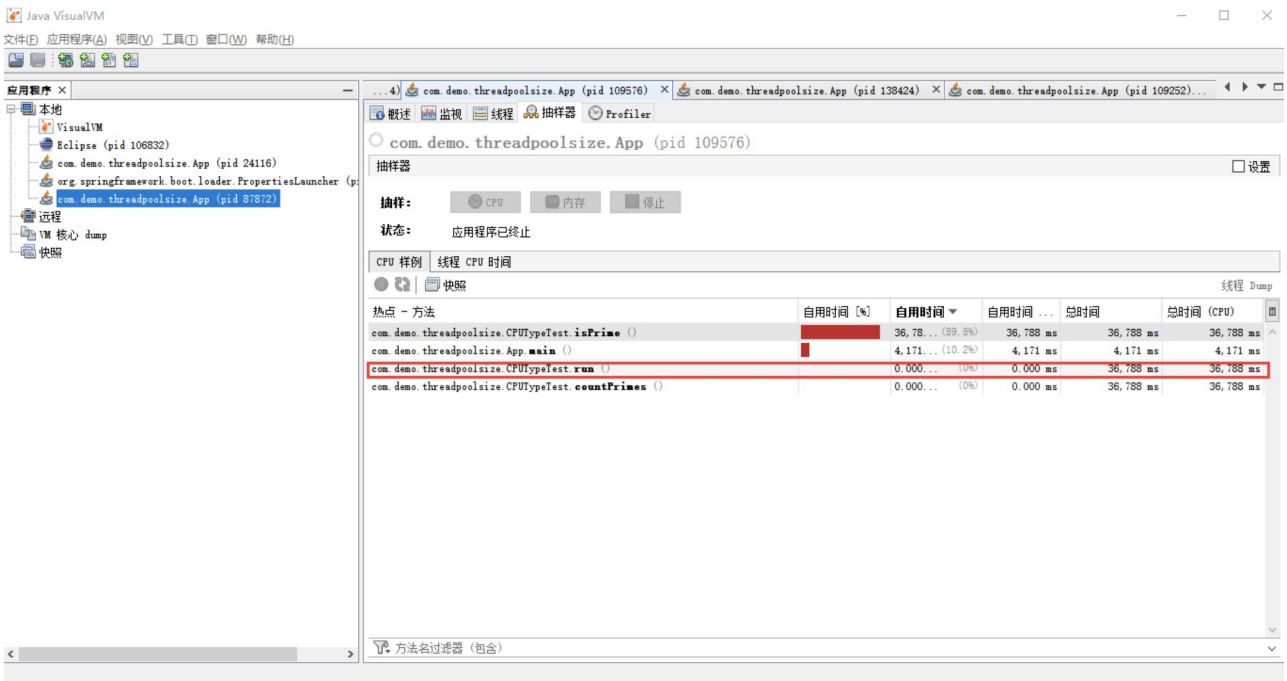
此时我们可以参考以下公式来计算线程数：

```
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：

```
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
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（
```

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？

上一页 下一页