前言

Java 多线程分类中写了 21 篇多线程的文章，21 篇文章的内容很多，个人认为，学习，内容越多、越杂的知识，越需要进行深刻的总结，这样才能记忆深刻，将知识变成自己的。这篇文章主要是对多线程的问题进行总结的，因此罗列了 40 个多线程的问题。

这些多线程的问题，有些来源于各大网站、有些来源于自己的思考。可能有些问题网上有、可能有些问题对应的答案也有、也可能有些各位网友也都看过，但是本文写作的重心就是**所有的问题都会按照自己的理解回答一遍，不会去看网上的答案**，因此可能有些问题讲的不对，能指正的希望大家不吝指教。

40 个问题汇总

1、多线程有什么用？

一个可能在很多人看来很扯淡的一个问题：我会用多线程就好了，还管它有什么用？在我看来，这个回答更扯淡。所谓"知其然知其所以然"，"会用"只是"知其然"，"为什么用"才是"知其所以然"，只有达到"知其然知其所以然"的程度才可以说是把一个知识点运用自如。OK，下面说说我对这个问题的看法：

（1）发挥多核 CPU 的优势

随着工业的进步，现在的笔记本、台式机乃至商用的应用服务器至少也都是双核的，4 核、8 核甚至 16 核的也都不少见，如果是单线程的程序，那么在双核 CPU 上就浪费了 50%，在 4 核 CPU 上就浪费了 75%。**单核 CPU 上所谓的"多线程"那是假的多线程，同一时间处理器只会处理一段逻辑，只不过线程之间切换得比较快，看着像多个线程"同时"运行罢了**。多核 CPU 上的多线程才是真正的多线程，它能让你的多段逻辑同时工作，多线程，可以真正发挥出多核 CPU 的优势来，达到充分利用 CPU 的目的。

（2）防止阻塞

从程序运行效率的角度来看，单核 CPU 不但不会发挥出多线程的优势，反而会因为在单核 CPU 上运行多线程导致线程上下文的切换，而降低程序整体的效率。但是单核 CPU 我们还是要应用多线程，就是为了防止阻塞。试想，如果单核 CPU 使用单线程，那么只要这个线程阻塞了，比方说远程读取某个数据吧，对端迟迟未返回又没有设置超时时间，那么你的整个程序在数据返回回来之前就停止运行了。多线程可以防止这个问题，多条线程同时运行，哪怕一条线程的代码执行读取数据阻塞，也不会影响其它任务的执行。

（3）便于建模

这是另外一个没有这么明显的优点了。假设有一个大的任务 A，单线程编程，那么就要考虑很多，建立整个程序模型比较麻烦。但是如果把这个大的任务 A 分解成几个小任务，任务 B、任务 C、任务 D，分别建立程序模型，并通过多线程分别运行这几个任务，那就简单很多了。

2、创建线程的方式

比较常见的一个问题了，一般就是两种：

（1）继承 Thread 类

（2）实现 Runnable 接口

至于哪个好，不用说肯定是后者好，因为实现接口的方式比继承类的方式更灵活，也能减少程序之间的耦合度，**面向接口编程**也是设计模式 6 大原则的核心。

3、start()方法和 run()方法的区别

只有调用了 start()方法，才会表现出多线程的特性，不同线程的 run()方法里面的代码交替执行。如果只是调用 run()方法，那么代码还是同步执行的，必须等待一个线程的 run()方法里面的代码全部执行完毕之后，另外一个线程才可以执行其 run()方法里面的代码。

4、Runnable 接口和 Callable 接口的区别

有点深的问题了，也看出一个 Java 程序员学习知识的广度。

Runnable 接口中的 run()方法的返回值是 void，它做的事情只是纯粹地去执行 run()方法中的代码而已；Callable 接口中的 call()方法是有返回值的，是一个泛型，和 Future、FutureTask 配合可以用来获取异步执行的结果。

这其实是很有用的一个特性，因为**多线程相比单线程更难、更复杂的一个重要原因就是因为多线程充满着未知性**，某条线程是否执行了？某条线程执行了多久？某条线程执行的时候我们期望的数据是否已经赋值完毕？无法得知，我们能做的只是等待这条多线程的任务执行完毕而已。而 Callable+Future/FutureTask 却可以获取多线程运行的结果，可以在等待时间太长没获取到需要的数据的情况下取消该线程的任务，真的是非常有用。

5、CyclicBarrier 和 CountDownLatch 的区别

两个看上去有点像的类，都在 java.util.concurrent 下，都可以用来表示代码运行到某个点上，二者的区别在于：

（1）CyclicBarrier 的某个线程运行到某个点上之后，该线程即停止运行，直到所有的线程都到达了这个点，所有线程才重新运行；CountDownLatch 则不是，某线程运行到某个点上之后，只是给某个数值-1 而已，该线程继续运行

（2）CyclicBarrier 只能唤起一个任务，CountDownLatch 可以唤起多个任务

（3）CyclicBarrier 可重用，CountDownLatch 不可重用，计数值为 0 该 CountDownLatch 就不可再用了

6、volatile 关键字的作用

一个非常重要的问题，是每个学习、应用多线程的 Java 程序员都必须掌握的。理解 volatile 关键字的作用的前提是要理解 Java 内存模型，这里就不讲 Java 内存模型了，可以参见第 31 点，volatile 关键字的作用主要有两个：

（1）多线程主要围绕可见性和原子性两个特性而展开，使用 volatile 关键字修饰的变量，保证了其在多线程之间的可见性，即每次读取到 volatile 变量，一定是最新的数据

（2）代码底层执行不像我们看到的高级语言----Java 程序这么简单，它的执行是 **Java 代码-->字节码-->根据字节码执行对应的 C/C++代码-->C/C++代码被编译成汇编语言-->和硬件电路交互**，现实中，为了获取更好的性能 JVM 可能会对指令进行重排序，多线程下可能会出现一些意想不到的问题。使用 volatile 则会对禁止语义重排序，当然这也一定程度上降低了代码执行效率

从实践角度而言，volatile 的一个重要作用就是和 CAS 结合，保证了原子性，详细的可以参见 java.util.concurrent.atomic 包下的类，比如 AtomicInteger。

7、什么是线程安全

又是一个理论的问题，各式各样的答案有很多，我给出一个个人认为解释地最好的：**如果你的代码在多线程下执行和在单线程下执行永远都能获得一样的结果，那么你的代码就是线程安全的**。

这个问题有值得一提的地方，就是线程安全也是有几个级别的：

（1）不可变

像 String、Integer、Long 这些，都是 final 类型的类，任何一个线程都改变不了它们的值，要改变除非新创建一个，因此这些不可变对象不需要任何同步手段就可以直接在多线程环境下使用

（2）绝对线程安全

不管运行时环境如何，调用者都不需要额外的同步措施。要做到这一点通常需要付出许多额外的代价，Java 中标注自己是线程安全的类，实际上绝大多数都不是线程安全的，不过绝对线程安全的类，Java 中也有，比方说 CopyOnWriteArrayList、CopyOnWriteArraySet

（3）相对线程安全

相对线程安全也就是我们通常意义上所说的线程安全，像 Vector 这种，add、remove 方法都是原子操作，不会被打断，但也仅限于此，如果有个线程在遍历某个 Vector、有个线程同时在 add 这个 Vector，99%的情况下都会出现 ConcurrentModificationException，也就是 **fail-fast 机制**。

（4）线程非安全

这个就没什么好说的了，ArrayList、LinkedList、HashMap 等都是线程非安全的类

8、Java 中如何获取到线程 dump 文件

死循环、死锁、阻塞、页面打开慢等问题，打线程 dump 是最好的解决问题的途径。所谓线程 dump 也就是线程堆栈，获取到线程堆栈有两步：

（1）获取到线程的 pid，可以通过使用 jps 命令，在 Linux 环境下还可以使用 ps -ef | grep java

（2）打印线程堆栈，可以通过使用 jstack pid 命令，在 Linux 环境下还可以使用 kill -3 pid
另外提一点，Thread 类提供了一个 getStackTrace()方法也可以用于获取线程堆栈。这是一个实例方法，因此此方法是和具体线程实例绑定的，每次获取获取到的是具体某个线程当前运行的堆栈，

9、一个线程如果出现了运行时异常会怎么样
如果这个异常没有被捕获的话，这个线程就停止执行了。另外重要的一点是：**如果这个线程持有某个某个对象的监视器，那么这个对象监视器会被立即释放**

10、如何在两个线程之间共享数据
通过在线程之间共享对象就可以了，然后通过 wait/notify/notifyAll、await/signal/signalAll 进行唤起和等待，比方说阻塞队列 BlockingQueue 就是为线程之间共享数据而设计的

11、sleep 方法和 wait 方法有什么区别

这个问题常问，sleep 方法和 wait 方法都可以用来放弃 CPU 一定的时间，不同点在于如果线程持有某个对象的监视器，sleep 方法不会放弃这个对象的监视器，wait 方法会放弃这个对象的监视器

12、生产者消费者模型的作用是什么
这个问题很理论，但是很重要：
（1）**通过平衡生产者的生产能力和消费者的消费能力来提升整个系统的运行效率**，这是生产者消费者模型最重要的作用

（2）解耦，这是生产者消费者模型附带的作用，解耦意味着生产者和消费者之间的联系少，联系越少越可以独自发展而不需要收到相互的制约

13、ThreadLocal 有什么用
简单说 ThreadLocal 就是一种以**空间换时间**的做法，在每个 Thread 里面维护了一个以开地址法实现的 ThreadLocal.ThreadLocalMap，把数据进行隔离，数据不共享，自然就没有线程安全方面的问题了

14、为什么 wait()方法和 notify()/notifyAll()方法要在同步块中被调用
这是 JDK 强制的，wait()方法和 notify()/notifyAll()方法在调用前都必须先获得对象的锁

15、wait()方法和 notify()/notifyAll()方法在放弃对象监视器时有什么区别
wait()方法和 notify()/notifyAll()方法在放弃对象监视器的时候的区别在于：**wait()方法立即释放对象监视器，notify()/notifyAll()方法则会等待线程剩余代码执行完毕才会放弃对象监视器**。

16、为什么要使用线程池
避免频繁地创建和销毁线程，达到线程对象的重用。另外，使用线程池还可以根据项目灵活地控制并发的数目。

17、怎么检测一个线程是否持有对象监视器
我也是在网上看到一道多线程面试题才知道有方法可以判断某个线程是否持有对象监视器：Thread 类提供了一个 holdsLock(Object obj)方法，当且仅当对象 obj 的监视器被某条线程持有的时候才会返回 true，注意这是一个 static 方法，这意味着**"某条线程"指的是当前线程**。

18、synchronized 和 ReentrantLock 的区别

synchronized 是和 if、else、for、while 一样的关键字，ReentrantLock 是类，这是二者的本质区别。既然 ReentrantLock 是类，那么它就提供了比 synchronized 更多更灵活的特性，可以被继承、可以有方法、可以有各种各样的类变量，ReentrantLock 比 synchronized 的扩展性体现在几点上：

（1）ReentrantLock 可以对获取锁的等待时间进行设置，这样就避免了死锁

（2）ReentrantLock 可以获取各种锁的信息

（3）ReentrantLock 可以灵活地实现多路通知

另外，二者的锁机制其实也是不一样的。ReentrantLock 底层调用的是 Unsafe 的 park 方法加锁，synchronized 操作的应该是对象头中 mark word，这点我不能确定。

19、ConcurrentHashMap 的并发度是什么

ConcurrentHashMap 的并发度就是 segment 的大小，默认为 16，这意味着最多同时可以有 16 条线程操作 ConcurrentHashMap，这也是 ConcurrentHashMap 对 Hashtable 的最大优势，任何情况下，Hashtable 能同时有两条线程获取 Hashtable 中的数据吗？

20、ReadWriteLock 是什么

首先明确一下，不是说 ReentrantLock 不好，只是 ReentrantLock 某些时候有局限。如果使用 ReentrantLock，可能本身是为了防止线程 A 在写数据、线程 B 在读数据造成的数据不一致，但这样，如果线程 C 在读数据、线程 D 也在读数据，读数据是不会改变数据的，没有必要加锁，但是还是加锁了，降低了程序的性能。

因为这个，才诞生了读写锁 ReadWriteLock。ReadWriteLock 是一个读写锁接口，ReentrantReadWriteLock 是 ReadWriteLock 接口的一个具体实现，实现了读写的分离，**读锁是共享的，写锁是独占的**，读和读之间不会互斥，读和写、写和读、写和写之间才会互斥，提升了读写的性能。

21、FutureTask 是什么

这个其实前面有提到过，FutureTask 表示一个异步运算的任务。FutureTask 里面可以传入一个 Callable 的具体实现类，可以对这个异步运算的任务的结果进行等待获取、判断是否已经完成、取消任务等操作。当然，由于 FutureTask 也是 Runnable 接口的实现类，所以 FutureTask 也可以放入线程池中。

22、Linux 环境下如何查找哪个线程使用 CPU 最长

这是一个比较偏实践的问题，这种问题我觉得挺有意义的。可以这么做：

（1）获取项目的 pid，jps 或者 ps -ef | grep java，这个前面有讲过

（2）top -H -p pid，顺序不能改变

这样就可以打印出当前的项目，每条线程占用 CPU 时间的百分比。注意这里打出的是 LWP，也就是操作系统原生线程的线程号，我笔记本山没有部署 Linux 环境下的 Java 工程，因此没有办法截图演示，网友朋友们如果公司是使用 Linux 环境部署项目的话，可以尝试一下。

使用"top -H -p pid"+"jps pid"可以很容易地找到某条占用 CPU 高的线程的线程堆栈，从而定位占用 CPU 高的原因，一般是因为不当的代码操作导致了死循环。

最后提一点，"top -H -p pid"打出来的 LWP 是十进制的，"jps pid"打出来的本地线程号是十六进制的，转换一下，就能定位到占用 CPU 高的线程的当前线程堆栈了。

23、Java 编程写一个会导致死锁的程序

第一次看到这个题目，觉得这是一个非常好的问题。很多人都知道死锁是怎么一回事儿：线程 A 和线程 B 相互等待对方持有的锁导致程序无限死循环下去。当然也仅限于此了，问一下怎么写一个死锁的程序就不知道了，这种情况说白了就是不懂什么是死锁，懂一个理论就完事儿了，实践中碰到死锁的问题基本上是看不出来的。

真正理解什么是死锁，这个问题其实不难，几个步骤：
（1）两个线程里面分别持有两个 Object 对象：lock1 和 lock2。这两个 lock 作为同步代码块的锁；
（2）线程 1 的 run()方法中同步代码块先获取 lock1 的对象锁，Thread.sleep(xxx)，时间不需要太多，50 毫秒差不多了，然后接着获取 lock2 的对象锁。这么做主要是为了防止线程 1 启动一下子就连续获得了 lock1 和 lock2 两个对象的对象锁
（3）线程 2 的 run)(方法中同步代码块先获取 lock2 的对象锁，接着获取 lock1 的对象锁，当然这时 lock1 的对象锁已经被线程 1 锁持有，线程 2 肯定是要等待线程 1 释放 lock1 的对象锁的
这样，线程 1"睡觉"睡完，线程 2 已经获取了 lock2 的对象锁了，线程 1 此时尝试获取 lock2 的对象锁，便被阻塞，此时一个死锁就形成了。代码就不写了，占的篇幅有点多，Java 多线程 7：死锁这篇文章里面有，就是上面步骤的代码实现。

24、怎么唤醒一个阻塞的线程
如果线程是因为调用了 wait()、sleep()或者 join()方法而导致的阻塞，可以中断线程，并且通过抛出 InterruptedException 来唤醒它；如果线程遇到了 IO 阻塞，无能为力，因为 IO 是操作系统实现的，Java 代码并没有办法直接接触到操作系统。

25、不可变对象对多线程有什么帮助
前面有提到过的一个问题，不可变对象保证了对象的内存可见性，对不可变对象的读取不需要进行额外的同步手段，提升了代码执行效率。

26、什么是多线程的上下文切换
多线程的上下文切换是指 CPU 控制权由一个已经正在运行的线程切换到另外一个就绪并等待获取 CPU 执行权的线程的过程。

27、如果你提交任务时，线程池队列已满，这时会发生什么
如果你使用的 LinkedBlockingQueue，也就是无界队列的话，没关系，继续添加任务到阻塞队列中等待执行，因为 LinkedBlockingQueue 可以近乎认为是一个无穷大的队列，可以无限存放任务；如果你使用的是有界队列比方说 ArrayBlockingQueue 的话，任务首先会被添加到 ArrayBlockingQueue 中，ArrayBlockingQueue 满了，则会使用拒绝策略 RejectedExecutionHandler 处理满了的任务，默认是 AbortPolicy。

28、Java 中用到的线程调度算法是什么
抢占式。一个线程用完 CPU 之后，操作系统会根据线程优先级、线程饥饿情况等数据算出一个总的优先级并分配下一个时间片给某个线程执行。

29、Thread.sleep(0)的作用是什么
这个问题和上面那个问题是相关的，我就连在一起了。由于 Java 采用抢占式的线程调度算法，因此可能会出现某条线程常常获取到 CPU 控制权的情况，为了让某些优先级比较低的线程也能获取到 CPU 控制权，可以使用 Thread.sleep(0)手动触发一次操作系统分配时间片的操作，这也是平衡 CPU 控制权的一种操作。

30、什么是自旋
很多 synchronized 里面的代码只是一些很简单的代码，执行时间非常快，此时等待的线程都加锁可能是一种不太值得的操作，因为线程阻塞涉及到用户态和内核态切换的问题。既然 synchronized 里面的代码执行得非常快，不妨让等待锁的线程不要被阻塞，而是在 synchronized 的边界做忙循环，这就是自旋。如果做了多次忙循环发现还没有获得锁，再阻塞，这样可能是一种更好的策略。

31、什么是 Java 内存模型

Java 内存模型定义了一种多线程访问 Java 内存的规范。Java 内存模型要完整讲不是这里几句话能说清楚的，我简单总结一下 Java 内存模型的几部分内容：

（1）Java 内存模型将内存分为了**主内存和工作内存**。类的状态，也就是类之间共享的变量，是存储在主内存中的，每次 Java 线程用到这些主内存中的变量的时候，会读一次主内存中的变量，并让这些内存在自己的工作内存中有一份拷贝，运行自己线程代码的时候，用到这些变量，操作的都是自己工作内存中的那一份。在线程代码执行完毕之后，会将最新的值更新到主内存中去

（2）定义了几个原子操作，用于操作主内存和工作内存中的变量

（3）定义了 volatile 变量的使用规则

（4）happens-before，即先行发生原则，定义了操作 A 必然先行发生于操作 B 的一些规则，比如在同一个线程内控制流前面的代码一定先行发生于控制流后面的代码、一个释放锁 unlock 的动作一定先行发生于后面对于同一个锁进行锁定 lock 的动作等等，只要符合这些规则，则不需要额外做同步措施，如果某段代码不符合所有的 happens-before 规则，则这段代码一定是线程非安全的

32、什么是 CAS

CAS，全称为 Compare and Swap，即比较-替换。假设有三个操作数：**内存值 V、旧的预期值 A、要修改的值 B，当且仅当预期值 A 和内存值 V 相同时，才会将内存值修改为 B 并返回 true，否则什么都不做并返回 false**。当然 CAS 一定要 volatile 变量配合，这样才能保证每次拿到的变量是主内存中最新的那个值，否则旧的预期值 A 对某条线程来说，永远是一个不会变的值 A，只要某次 CAS 操作失败，永远都不可能成功。

33、什么是乐观锁和悲观锁

（1）乐观锁：就像它的名字一样，对于并发间操作产生的线程安全问题持乐观状态，乐观锁认为竞争不总是会发生，因此它不需要持有锁，将**比较-替换**这两个动作作为一个原子操作尝试去修改内存中的变量，如果失败则表示发生冲突，那么就应该有相应的重试逻辑。

（2）悲观锁：还是像它的名字一样，对于并发间操作产生的线程安全问题持悲观状态，悲观锁认为竞争总是会发生，因此每次对某资源进行操作时，都会持有一个独占的锁，就像 synchronized，不管三七二十一，直接上了锁就操作资源了。

34、什么是 AQS

简单说一下 AQS，AQS 全称为 AbstractQueuedSychronizer，翻译过来应该是抽象队列同步器。

如果说 java.util.concurrent 的基础是 CAS 的话，那么 AQS 就是整个 Java 并发包的核心了，ReentrantLock、CountDownLatch、Semaphore 等等都用到了它。AQS 实际上以双向队列的形式连接所有的 Entry，比方说 ReentrantLock，所有等待的线程都被放在一个 Entry 中并连成双向队列，前面一个线程使用 ReentrantLock 好了，则双向队列实际上的第一个 Entry 开始运行。

AQS 定义了对双向队列所有的操作，而只开放了 tryLock 和 tryRelease 方法给开发者使用，开发者可以根据自己的实现重写 tryLock 和 tryRelease 方法，以实现自己的并发功能。

35、单例模式的线程安全性

老生常谈的问题了，首先要说的是单例模式的线程安全意味着：**某个类的实例在多线程环境下只会被创建一次出来**。单例模式有很多种的写法，我总结一下：

（1）饿汉式单例模式的写法：线程安全

（2）懒汉式单例模式的写法：非线程安全

（3）双检锁单例模式的写法：线程安全

36、Semaphore 有什么作用

Semaphore 就是一个信号量，它的作用是**限制某段代码块的并发数**。Semaphore 有一个构造函数，可以传入一个 int 型整数 n，表示某段代码最多只有 n 个线程可以访问，如果超出了 n，那么请等待，等到某个线程执行完毕这段代码块，下一个线程再进入。由此可以看出如果 Semaphore 构造函数中传入的 int 型整数 n=1，相当于变成了一个 synchronized 了。

37、Hashtable 的 size()方法中明明只有一条语句"return count"，为什么还要做同步？

这是我之前的一个困惑，不知道大家有没有想过这个问题。某个方法中如果有多条语句，并且都在操作同一个类变量，那么在多线程环境下不加锁，势必会引发线程安全问题，这很好理解，但是 size()方法明明只有一条语句，为什么还要加锁？

关于这个问题，在慢慢地工作、学习中，有了理解，主要原因有两点：

（1）**同一时间只能有一条线程执行固定类的同步方法，但是对于类的非同步方法，可以多条线程同时访问**。所以，这样就有问题了，可能线程 A 在执行 Hashtable 的 put 方法添加数据，线程 B 则可以正常调用 size()方法读取 Hashtable 中当前元素的个数，那读取到的值可能不是最新的，可能线程 A 添加了完了数据，但是没有对 size++，线程 B 就已经读取 size 了，那么对于线程 B 来说读取到的 size 一定是不准确的。而给 size()方法加了同步之后，意味着线程 B 调用 size()方法只有在线程 A 调用 put 方法完毕之后才可以调用，这样就保证了线程安全性

（2）**CPU 执行代码，执行的不是 Java 代码，这点很关键，一定得记住**。Java 代码最终是被翻译成汇编代码执行的，汇编代码才是真正可以和硬件电路交互的代码。**即使你看到 Java 代码只有一行，甚至你看到 Java 代码编译之后生成的字节码也只有一行，也不意味着对于底层来说这句语句的操作只有一个**。一句"return count"假设被翻译成了三句汇编语句执行，完全可能执行完第一句，线程就切换了。

38、线程类的构造方法、静态块是被哪个线程调用的

这是一个非常刁钻和狡猾的问题。请记住：线程类的构造方法、静态块是被 new 这个线程类所在的线程所调用的，而 run 方法里面的代码才是被线程自身所调用的。

如果说上面的说法让你感到困惑，那么我举个例子，假设 Thread2 中 new 了 Thread1，main 函数中 new 了 Thread2，那么：

（1）Thread2 的构造方法、静态块是 main 线程调用的，Thread2 的 run()方法是 Thread2 自己调用的

（2）Thread1 的构造方法、静态块是 Thread2 调用的，Thread1 的 run()方法是 Thread1 自己调用的

39、同步方法和同步块，哪个是更好的选择

同步块，这意味着同步块之外的代码是异步执行的，这比同步整个方法更提升代码的效率。请知道一条原则：**同步的范围越小越好**。

借着这一条，我额外提一点，虽说同步的范围越少越好，但是在 Java 虚拟机中还是存在着一种叫做**锁粗化**的优化方法，这种方法就是把同步范围变大。这是有用的，比方说 StringBuffer，它是一个线程安全的类，自然最常用的 append()方法是一个同步方法，我们写代码的时候会反复 append 字符串，这意味着要进行反复的加锁->解锁，这对性能不利，因为这意味着 Java 虚拟机在这条线程上要反复地在内核态和用户态之间进行切换，因此 Java 虚拟机会将多次 append 方法调用的代码进行一个锁粗化的操作，将多次的 append 的操作扩展到 append 方法的头尾，变成一个大的同步块，这样就减少了加锁-->解锁的次数，有效地提升了代码执行的效率。

40、高并发、任务执行时间短的业务怎样使用线程池？并发不高、任务执行时间长的业务怎样使用线程池？并发高、业务执行时间长的业务怎样使用线程池？

这是我在并发编程网上看到的一个问题，把这个问题放在最后一个，希望每个人都能看到并且思考一下，因为这个问题非常好、非常实际、非常专业。关于这个问题，个人看法是：

（1）高并发、任务执行时间短的业务，线程池线程数可以设置为 CPU 核数+1，减少线程上下文的切换

（2）并发不高、任务执行时间长的业务要区分开看：

a）假如是业务时间长集中在 IO 操作上，也就是 IO 密集型的任务，因为 IO 操作并不占用 CPU，所以不要让所有的 CPU 闲下来，可以加大线程池中的线程数目，让 CPU 处理更多的业务

b）假如是业务时间长集中在计算操作上，也就是计算密集型任务，这个就没办法了，和（1）一样吧，线程池中的线程数设置得少一些，减少线程上下文的切换

（3）并发高、业务执行时间长，解决这种类型任务的关键不在于线程池而在于整体架构的设计，看看这些业务里面某些数据是否能做缓存是第一步，增加服务器是第二步，至于线程池的设置，设置参考（2）。最后，业务执行时间长的问题，也可能需要分析一下，看看能不能使用中间件对任务进行拆分和解耦。

==========

我不能保证写的每个地方都是对的，但是至少能保证不复制、不黏贴，保证每一句话、每一行代码都经过了认真的推敲、仔细的斟酌。每一篇文章的背后，希望都能看到自己对于技术、对于生活的态度。

我相信乔布斯说的，只有那些疯狂到认为自己可以改变世界的人才能真正地改变世界。面对压力，我可以挑灯夜战、不眠不休；面对困难，我愿意迎难而上、永不退缩。

其实我想说的是，我只是一个程序员，这就是我现在纯粹人生的全部。

==========