根据CPU核心数确定线程池并发线程数 - 只会一点java - 博客园

cnblogs.com/dennyzhangdd/p/6909771.html

阅读目录

正文

回到顶部

一、抛出问题

关于如何计算并发线程数，一般分两派，来自两本书，且都是好书，到底哪个是对的？问题追踪后，整理如下：

第一派：《Java Concurrency in Practice》即《java并发编程实践》，如下图：

For compute-intensive tasks, an N_{cpu}-processor system usually achieves optimum utilization with a thread pool of N_{cpu} +1 threads. (Even compute-intensive threads occasionally take a page fault or pause for some other reason, so an "extra" runnable thread prevents CPU cycles from going unused when this happens.) For tasks that also include I/O or other blocking operations, you want a larger pool, since not all of the threads will be schedulable at all times. In order to size the pool properly, you must estimate the ratio of waiting time to compute time for your tasks; this estimate need not be precise and can be obtained through pro-filing or instrumentation. Alternatively, the size of the thread pool can be tuned by running the application using several different pool sizes under a benchmark load and observing the level of CPU utilization.

Given these definitions:

N_{cpu} = number of CPUs
U_{cpu} = target CPU utilization, $0 \le U_{cpu} \le 1$
$\frac{W}{C}$ = ratio of wait time to compute time

The optimal pool size for keeping the processors at the desired utilization is:

$$N_{threads} = N_{cpu} * U_{cpu} * \left(1 + \frac{W}{C}\right)$$

You can determine the number of CPUs using `Runtime`:

```
int N_CPUS = Runtime.getRuntime().availableProcessors();
```

Of course, CPU cycles are not the only resource you might want to manage using thread pools. Other resources that can contribute to sizing constraints are memory, file handles, socket handles, and database connections. Calculating pool size constraints for these types of resources is easier: just add up how much of that resource each task requires and divide that into the total quantity available. The result will be an upper bound on the pool size.

When tasks require a pooled resource such as database connections, thread pool size and resource pool size affect each other. If each task requires a connection, the effective size of the thread pool is limited by the connection pool size. Similarly, when the only consumers of connections are pool tasks, the effective size of the connection pool is limited by the thread pool size.

如上图，在《Java Concurrency in Practice》一书中，给出了估算线程池大小的公式：

Nthreads=Ncpu*Ucpu*(1+w/c)，其中

Ncpu=CPU核心数

Ucpu=cpu使用率，0~1

W/C=等待时间与计算时间的比率

第二派：《Programming Concurrency on the JVM Mastering》即《Java 虚拟机并发编程》

1. 确定线程数

为了解决上述难题，我们希望至少可以创建处理器核心数那么多个线程。这就保证了有尽可能多地处理器核心可以投入到解决问题的工作中去。通过下面的代码，我们可以很容易地获取到系统可用的处理器核心数[⊖]：

```
Runtime.getRuntime().availableProcessors();
```

所以，应用程序的最小线程数应该等于可用的处理器核数。如果所有的任务都是计算密集型的，则创建处理器可用核心数那么多个线程就可以了。在这种情况下，创建更多的线程对程序性能而言反而是不利的。因为当有多个任务处于就绪状态时，处理器核心需要在线程间频繁进行上下文切换，而这种切换对程序性能损耗较大。但如果任务都是 IO 密集型的，那么我们就需要开更多的线程来提高性能。

当一个任务执行 IO 操作时，其线程将被阻塞，于是处理器可以立即进行上下文切换以便处理其他就绪线程。如果我们只有处理器可用核心数那么多个线程的话，则即使有待执行的任务也无法处理，因为我们已经拿不出更多的线程供处理器调度了。

如果任务有 50% 的时间处于阻塞状态，则程序所需线程数为处理器可用核心数的两倍。如果任务被阻塞的时间少于 50%，即这些任务是计算密集型的，则程序所需线程数将随之减少，但最少也不应低于处理器的核心数。如果任务被阻塞的时间大于执行时间，即该任务是 IO 密集型的，我们就需要创建比处理器核心数大几倍数量的线程。

我们可以计算出程序所需线程的总数，总结如下：

线程数 =CPU 可用核心数 /(1 – 阻塞系数)，其中阻塞系数的取值在 0 和 1 之间。

计算密集型任务的阻塞系数为 0，而 IO 密集型任务的阻塞系数则接近 1。一个完全阻塞的任务是注定要挂掉的，所以我们无须担心阻塞系数会达到 1。

为了更好地确定程序所需线程数，我们需要知道下面两个关键参数：

- 处理器可用核心数

⊖ availableProcessors() 函数可以获取 JVM 可使用的逻辑处理器数量。

更多资源请访问稀酷客(www.ckook.co

- 任务的阻塞系数

第一个参数很容易确定，我们甚至可以用之前的方法在运行时查到这个值。但确定阻塞系数就稍微困难一些。我们可以先试着猜测，抑或采用一些性能分析工具或 java.lang.management API 来确定线程花在系统 /IO 操作上的时间与 CPU 密集任务所耗时间的比值。

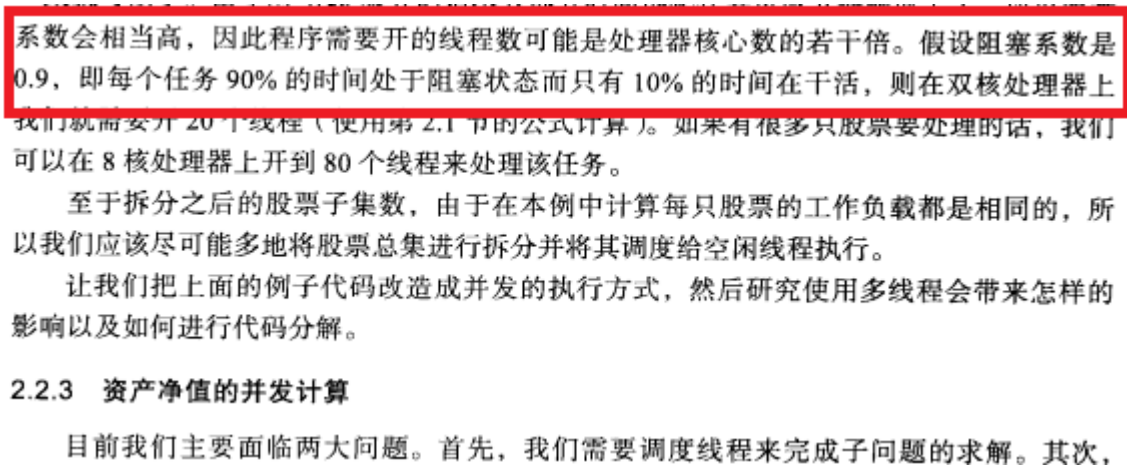
线程数=Ncpu/（1-阻塞系数）

回到顶部

二、分析

对于派系一，假设cpu100%运转，即撇开CPU使用率这个因素，线程数=Ncpu*(1+w/c)。

现在假设将派系二的公式等于派系一公式，即Ncpu/（1-阻塞系数）=Ncpu*(1+w/c),===》阻塞系数=w/(w+c)，即阻塞系数=阻塞时间/（阻塞时间+计算时间），这个结论在派系二后续中得到应征，如下图：


系数会相当高，因此程序需要开的线程数可能是处理器核心数的若干倍。假设阻塞系数是0.9，即每个任务90%的时间处于阻塞状态而只有10%的时间在干活，则在双核处理器上
我们就需要开20个线程（使用第2.1节的公式计算）。如果有很多只股票要处理的话，我们可以在8核处理器上开到80个线程来处理该任务。

至于拆分之后的股票子集数，由于在本例中计算每只股票的工作负载都是相同的，所以我们应该尽可能多地将股票总集进行拆分并将其调度给空闲线程执行。

让我们把上面的例子代码改造成并发的执行方式，然后研究使用多线程会带来怎样的影响以及如何进行代码分解。

2.2.3 资产净值的并发计算

目前我们主要面临两大问题。首先，我们需要调度线程来完成子问题的求解。其次，

由此可见，派系一和派系二其实是一个公式.....这样我就放心了.....

回到顶部

三、实际应用

那么实际使用中并发线程数如何设置呢？分析如下（我们以派系一公式为例）：

Nthreads=Ncpu*(1+w/c)

IO密集型：一般情况下，如果存在IO，那么肯定w/c>1（阻塞耗时一般都是计算耗时的很多倍）,但是需要考虑系统内存有限（每开启一个线程都需要内存空间），这里需要上服务器测试具体多少个线程数适合（CPU占比、线程数、总耗时、内存消耗）。如果不想去测试，保守点取1即，Nthreads=Ncpu*(1+1)=2Ncpu。这样设置一般都OK。

计算密集型：假设没有等待w=0，则W/C=0. Nthreads=Ncpu。

至此结论就是：

IO密集型=2Ncpu（可以测试后自己控制大小，2Ncpu一般没问题）（常出现于线程中：数据库数据交互、文件上传下载、网络数据传输等等）

计算密集型=Ncpu（常出现于线程中：复杂算法）

java中：Ncpu=`Runtime.getRuntime().availableProcessors()`

==========此处可略过==========

当然派系一种《Java Concurrency in Practice》还有一种说法，

mostly computation, I/O, or some combination? Do they require a scarce resource, such as a JDBC connection? If you have different categories of tasks with very different behaviors, consider using multiple thread pools so each can be tuned according to its workload.

For compute-intensive tasks, an N_{cpu}-processor system usually achieves optimum utilization with a thread pool of N_{cpu} +1 threads. (Even compute-intensive threads occasionally take a page fault or pause for some other reason, so an "extra" runnable thread prevents CPU cycles from going unused when this happens.) For tasks that also include I/O or other blocking operations, you want a larger pool, since not all of the threads will be schedulable at all times. In order to size the pool properly, you must estimate the ratio of waiting time to compute time for your tasks; this estimate need not be precise and can be obtained through pro-filing or instrumentation. Alternatively, the size of the thread pool can be tuned by running the application using several different pool sizes under a benchmark load and observing the level of CPU utilization.

即对于计算密集型的任务，在拥有N个处理器的系统上，当线程池的大小为N+1时，通常能实现最优的效率。(即使当计算密集型的线程偶尔由于缺失故障或者其他原因而暂停时，这个额外的线程也能确保CPU的时钟周期不会被浪费。)

即，计算密集型=Ncpu+1，但是这种做法导致的多一个cpu上下文切换是否值得，这里不考虑。读者可自己考量。

==========

回到顶部

四、总结：

选择线程池并发线程数的因素很多：任务类型、内存等线程中使用到所有资源都需要考虑。本文经过对现有文献的分析论证，得出结论，并给出了实际应用公式，实乃工程师之福利，技术之典范.....