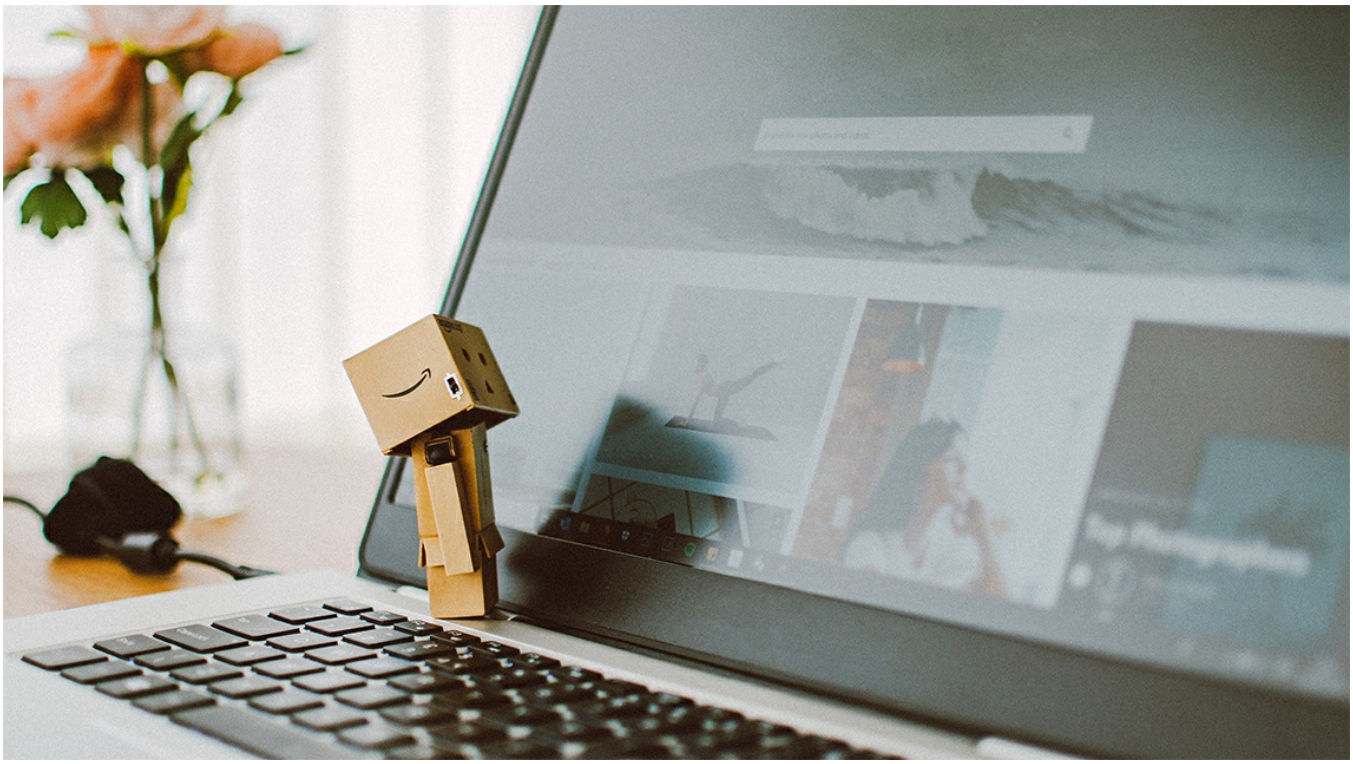
15 | 多线程调优（上）：哪些操作导致了上下文切换？

2019-06-22 刘超



你好，我是刘超。

我们常说“实践是检验真理的唯一标准”，这句话不光在社会发展中可行，在技术学习中也同样适用。

记得我刚入职上家公司的时候，恰好赶上了一次抢购活动。这是系统重构上线后经历的第一次高并发考验，如期出现了大量超时报警，不过比我预料的要好一点，起码没有挂掉重启。

通过工具分析，我发现 cs（上下文切换每秒次数）指标已经接近了 60w，平时的话最高5w。再通过日志分析，我发现了大量带有 wait() 的 Exception，由此初步怀疑是大量线程处理不及时导致的，进一步锁定问题是连接池大小设置不合理。后来我就模拟了生产环境配置，对连接数压测进行调节，降低最大线程数，最后系统的性能就上去了。

从实践中总结经验，我知道了在并发程序中，并不是启动更多的线程就能让程序最大限度地并发执行。线程数量设置太小，会导致程序不能充分地利用系统资源；线程数量设置太大，又可能带来资源的过度竞争，导致上下文切换带来额外的系统开销。

你看，其实很多经验就是这么一点点积累的。那么今天，我就想和你分享下“上下文切换”的相关内容，希望也能让你有所收获。

初识上下文切换

我们首先得明白，上下文切换到底是什么。

其实在单个处理器的时期，操作系统就能处理多线程并发任务。处理器给每个线程分配 CPU 时间片（Time Slice），线程在分配获得的时间片内执行任务。

CPU 时间片是 CPU 分配给每个线程执行的时间段，一般为几十毫秒。在这么短的时间内线程互相切换，我们根本感觉不到，所以看上去就好像是同时进行的一样。

时间片决定了一个线程可以连续占用处理器运行的时长。当一个线程的时间片用完了，或者因自身原因被迫暂停运行了，这个时候，另外一个线程（可以是同一个线程或者其它进程的线程）就会被操作系统选中，来占用处理器。这种一个线程被暂停剥夺使用权，另外一个线程被选中开始或者继续运行的过程就叫做上下文切换（Context Switch）。

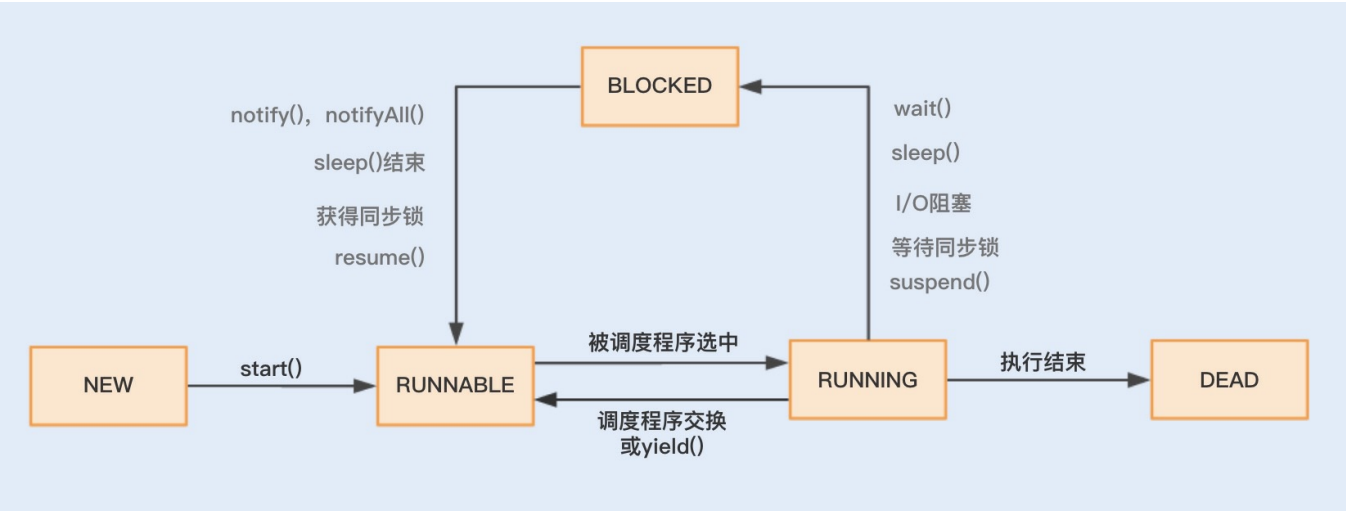
具体来说，一个线程被剥夺处理器的使用权而被暂停运行，就是“切出”；一个线程被选中占用处理器开始或者继续运行，就是“切入”。在这种切出切入的过程中，操作系统需要保存和恢复相应的进度信息，这个进度信息就是“上下文”了。

那上下文都包括哪些内容呢？具体来说，它包括了寄存器的存储内容以及程序计数器存储的指令内容。CPU 寄存器负责存储已经、正在和将要执行的任务，程序计数器负责存储CPU 正在执行的指令位置以及即将执行的下一条指令的位置。

在当前 CPU 数量远远不止一个的情况下，操作系统将 CPU 轮流分配给线程任务，此时的上下文切换就变得更加频繁了，并且存在跨 CPU 上下文切换，比起单核上下文切换，跨核切换更加昂贵。

多线程上下文切换诱因

在操作系统中，上下文切换的类型还可以分为进程间的上下文切换和线程间的上下文切换。而在多线程编程中，我们主要面对的就是线程间的上下文切换导致的性能问题，下面我们就重点看看究竟是什么原因导致了多线程的上下文切换。开始之前，先看下Java 线程的生命周期状态。



结合图示可知，线程主要有“新建”（NEW）、“就绪”（RUNNABLE）、“运行”（RUNNING）、

“阻塞”（BLOCKED）、“死亡”（DEAD）五种状态。

在这个运行过程中，线程由RUNNABLE转为非RUNNABLE的过程就是线程上下文切换。

一个线程的状态由 RUNNING 转为 BLOCKED，再由 BLOCKED 转为 RUNNABLE，然后再被调度器选中执行，这就是一个上下文切换的过程。

当一个线程从 RUNNING 状态转为 BLOCKED 状态时，我们称为一个线程的暂停，线程暂停被切出之后，操作系统会保存相应的上下文，以便这个线程稍后再次进入 RUNNABLE 状态时能够在之前执行进度的基础上继续执行。

当一个线程从 BLOCKED 状态进入到 RUNNABLE 状态时，我们称为一个线程的唤醒，此时线程将获取上次保存的上下文继续完成执行。

通过线程的运行状态以及状态间的相互切换，我们可以了解到，多线程的上下文切换实际上就是由多线程两个运行状态的互相切换导致的。

那么在线程运行时，线程状态由 RUNNING 转为 BLOCKED 或者由 BLOCKED 转为 RUNNABLE，这又是什么诱发的呢？

我们可以分两种情况来分析，一种是程序本身触发的切换，这种我们称为自发性上下文切换，另一种是由系统或者虚拟机诱发的非自发性上下文切换。

自发性上下文切换指线程由 Java 程序调用导致切出，在多线程编程中，执行调用以下方法或关键字，常常就会引发自发性上下文切换。

- sleep()
- wait()
- yield()
- join()
- park()
- synchronized
- lock

非自发性上下文切换指线程由于调度器的原因被迫切出。常见的有：线程被分配的时间片用完，虚拟机垃圾回收导致或者执行优先级的问题导致。

这里重点说下“虚拟机垃圾回收为什么会导致上下文切换”。在 Java 虚拟机中，对象的内存都是由虚拟机中的堆分配的，在程序运行过程中，新的对象将不断被创建，如果旧的对象使用后不进行回收，堆内存将很快被耗尽。Java 虚拟机提供了一种回收机制，对创建后不再使用的对象进行回收，从而保证堆内存的可持续性分配。而这种垃圾回收机制的使用有可能会导致 stop-the-world 事件的发生，这其实就是一种线程暂停行为。

发现上下文切换

我们总说上下文切换会带来系统开销，那它带来的性能问题是不是真有这么糟糕呢？我们又该怎么去监测到上下文切换？上下文切换到底开销在哪些环节？接下来我将给出一段代码，来对比串联执行和并发执行的速度，然后一一解答这些问题。

```
public class DemoApplication {
    public static void main(String[] args) {
        //运行多线程
        MultiThreadTester test1 = new MultiThreadTester();
        test1.Start();
        //运行单线程
        SerialTester test2 = new SerialTester();
        test2.Start();
    }


    static class MultiThreadTester extends ThreadContextSwitchTester {
        @Override
        public void Start() {
            long start = System.currentTimeMillis();
            MyRunnable myRunnable1 = new MyRunnable();
            Thread[] threads = new Thread[4];
            //创建多个线程
            for (int i = 0; i < 4; i++) {
                threads[i] = new Thread(myRunnable1);
                threads[i].start();
            }
            for (int i = 0; i < 4; i++) {
                try {
                    //等待一起运行完
                    threads[i].join();
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
            long end = System.currentTimeMillis();
```

```
            System.out.println("multi thread exce time: " + (end - start) + "s");
            System.out.println("counter: " + counter);
        }
        // 创建一个实现Runnable的类
        class MyRunnable implements Runnable {
            public void run() {
                while (counter < 100000000) {
                    synchronized (this) {
                        if(counter < 100000000) {
                            increaseCounter();
                        }

                    }
                }
            }
        }
    }

    //创建一个单线程
    static class SerialTester extends ThreadContextSwitchTester{
        @Override
        public void Start() {
            long start = System.currentTimeMillis();
            for (long i = 0; i < count; i++) {
                increaseCounter();
            }
            long end = System.currentTimeMillis();
            System.out.println("serial exec time: " + (end - start) + "s");
            System.out.println("counter: " + counter);
        }
    }

    //父类
    static abstract class ThreadContextSwitchTester {
        public static final int count = 100000000;
        public volatile int counter = 0;
        public int getCount() {
            return this.counter;
```

```
            return this.counter;
        }
        public void increaseCounter() {

            this.counter += 1;
        }
        public abstract void Start();
    }
}
```

执行之后，看一下两者的时间测试结果：

```
<terminated> DemoApplication [Java Application] C:\Program Files\Java\jre1.8.0_201\bin\javaw.exe (2019年3月2日 下午3:36:10)
multi thread exce time: 5234s
counter: 100000000
serial exec time: 858s
counter: 100000000
```

通过数据对比我们可以看到：串联的执行速度比并发的执行速度要快。这就是因为线程的上下文切换导致了额外的开销，使用 Synchronized 锁关键字，导致了资源竞争，从而引起了上下文切换，但即使不使用 Synchronized 锁关键字，并发的执行速度也无法超越串联的执行速度，这是因为多线程同样存在着上下文切换。Redis、NodeJS的设计就很好地体现了单线程串行的优势。

在 Linux 系统下，可以使用 Linux 内核提供的 vmstat 命令，来监视 Java 程序运行过程中系统的上下文切换频率，cs如下图所示：

```
[root@localhost ~]# vmstat 2
procs -----------memory---------- ---swap-- -----io---- -system-- ------cpu-----
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  0      0 2130600 247876 5004968    0    0     0     1    0    1  0  0 100  0  0
 0  0      0 2130352 247876 5004968    0    0     0     0  317  653  0  0 99  0  0
 2  0      0 2106740 247876 5004968    0    0     0     2 1818 5202  9  1 90  0  0
 2  0      0 2106848 247876 5005004    0    0     0     0 13664 42153 35  5 59  0  0
 1  0      0 2106848 247876 5005004    0    0     0     0 16576 51599 33  6 61  0  0
 0  0      0 2130536 247876 5004972    0    0     0    36  511  635  5  0 94  0  0
 0  0      0 2130536 247876 5004972    0    0     0     0  254  579  0  0 100  0  0
 0  0      0 2130536 247876 5004972    0    0     0     0  254  569  0  0 100  0  0
```

如果是监视某个应用的上下文切换，就可以使用 pidstat命令监控指定进程的 Context Switch 上下文切换。

```
[root@localhost ~]# pidstat -w -l -p 998 1 100
Linux 3.10.0-514.el7.x86_64 (localhost.localdomain)     03/03/2019      _x86_64_        (4 CPU)

12:09:34 PM   UID       PID   cswch/s nvcswch/s  Command
12:09:35 PM     0       998      0.00      0.00  -bash
12:09:36 PM     0       998      0.00      0.00  -bash
12:09:37 PM     0       998      0.00      0.00  -bash
12:09:38 PM     0       998      0.00      0.00  -bash
12:09:39 PM     0       998      0.00      0.00  -bash
12:09:40 PM     0       998      0.00      0.00  -bash
12:09:41 PM     0       998      0.00      0.00  -bash
12:09:42 PM     0       998      0.00      0.00  -bash
12:09:43 PM     0       998      0.00      0.00  -bash
12:09:44 PM     0       998      0.00      0.00  -bash
12:09:45 PM     0       998      0.00      0.00  -bash
12:09:46 PM     0       998      0.00      0.00  -bash
12:09:47 PM     0       998      0.00      0.00  -bash
12:09:48 PM     0       998      0.00      0.00  -bash
12:09:49 PM     0       998      0.00      0.00  -bash
12:09:50 PM     0       998      2.00      0.00  -bash
12:09:51 PM     0       998      0.00      0.00  -bash
12:09:52 PM     0       998      0.00      0.00  -bash
```

由于 Windows 没有像 vmstat 这样的工具，在 Windows 下，我们可以使用 Process Explorer，来查看程序执行时，线程间上下文切换的次数。

至于系统开销具体发生在切换过程中的哪些具体环节，总结如下：

- 操作系统保存和恢复上下文；
- 调度器进行线程调度；
- 处理器高速缓存重新加载；
- 上下文切换也可能导致整个高速缓存区被冲刷，从而带来时间开销。

总结

上下文切换就是一个工作的线程被另外一个线程暂停，另外一个线程占用了处理器开始执行任务的过程。系统和 Java 程序自发性以及非自发性的调用操作，就会导致上下文切换，从而带来系统开销。

线程越多，系统的运行速度不一定越快。那么我们平时在并发量比较大的情况下，什么时候用单线程，什么时候用多线程呢？

一般在单个逻辑比较简单，而且速度相对来非常快的情况下，我们可以使用单线程。例如，我们前面讲到的 Redis，从内存中快速读取值，不用考虑 I/O 瓶颈带来的阻塞问题。而在逻辑相对来说很复杂的场景，等待时间相对较长又或者是需要大量计算的场景，我建议使用多线程来提高系统的整体性能。例如，NIO 时期的文件读写操作、图像处理以及大数据分析等。

思考题

以上我们主要讨论的是多线程的上下文切换，前面我讲分类的时候还曾提到了进程间的上下文切换。那么你知道在多线程中使用Synchronized还会发生进程间的上下文切换吗？具体又会发生在哪些环节呢？

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。

精选留言

李博 4

如果Synchronized块中包含io操作或者大量的内存分配时，可能会导致进程IO等待或者内存不足。进一步会导致操作系统进行进程切换，等待系统资源满足时在切换到当前进程。 不知道理解的对不对？

2019-06-22

作者回复

进程上下文切换，是指用户态和内核态的来回切换。我们知道，如果一旦Synchronized锁资源竞争激烈，线程将会被阻塞，阻塞的线程将会从用户态调用内核态，尝试获取mutex，这个过程就是进程上下文切换。

2019-06-23

老杨同志 3

使用Synchronized获得锁失败，进入等待队列会发生上下文切换。如果竞争锁时锁是其他线程的偏向锁，需要降级，这是需要stop the world也会发生上下文切换

2019-06-22

作者回复

理解正确~

2019-06-23

QQ怪 2

老师讲的上下文切换的确干货很多，思考题我觉得应该是使用synchronize导致单线程进行，且

执行方法时间过长，当前进程时间片执行时间结束，导致cpu不得不进行进程间上下文切换。

2019-06-22

作者回复

进程上下文切换，是指用户态和内核态的来回切换。当Synchronized锁资源竞争激烈，线程将会被阻塞，阻塞的线程将会从用户态调用内核态，尝试获取mutex，这个过程是Synchronized锁产生的进程上下文切换。

2019-06-23

Jxin 2

首先，如何决定多线程。这点核心的依据我认为是提高计算机资源使用率。将cpu执行耗时较长比如io操作，跟耗时较短比如纯逻辑计算的业务操作分解开。根据时间比例对应分配操作线程池的线程数。进而保障资源最大化利用，比如耗时较短的业务线程不会空闲。理论上多核的现在，并行逻辑都要比串行逻辑快（并行交集时间，既剩下的时间大于上下文切换和资源合并的时间开销）。其实我觉得还得引入业务价值做考虑，核心业务加大优化力度，边缘业务性能保持在容忍线以上就好，为核心业务让渡资源。最后是思考题。才疏学浅，隐式锁个人认为是为线程价格执行体准备的，不会影响到进程间的切换。但是多进程间用的也是同一台服务器的资源。所以必然也会有上下文切换，而这块都是非自发的。比如cpu时间分片呈现的进程间交替使用cpu，或则进程各自持有的虚拟内存页对实际物理内存的使用。至于文件操作，java发现文件资源被其他进程占用好像是直接报错的，所以没有进程间竞争。但输出设备，打印机音箱这些，它们有多进程轮流共用的现象，感觉起来也有点分片执行，优先调度之类的样子，应该也有竞争。个人认知半猜回复，还望老师指正。搬砖引玉。

2019-06-22

晓杰 1

锁的竞争太激烈会导致锁升级为重量级锁，未抢到锁的线程会进入monitor，而monitor依赖于底层操作系统的mutex lock，获取锁时会发生用户态和内核态之间的切换，所以会发生进程间的上下文切换。

2019-06-25

作者回复

对的

2019-06-26

勿闻轩外香 1

干货满满

2019-06-24

周星星 1

使用Synchronized在锁获取的时候会发生CPU上下文切换，多线程本身也会有上下文切换，这样就会多一次切换，是这样吗？

2019-06-22

作者回复

Synchronized在轻量级锁之前，锁资源竞争产生的是线程上下文切换，一旦升级到重量级锁，就会产生进程上下文切换。

2019-06-23

[] 1

sleep引起上下文切换是指系统调用吗？用户态到内核态的切换。但是这时候线程会从running变成block吗？感觉这个线程没有让出控制吧，跟wait不一样的吧

2019-06-22

作者回复

sleep和wait一样，都会进入阻塞状态，区别是sleep没有释放锁，而wait释放了锁。所以也是一次上下文切换。

2019-06-23

木木匠 1

思考题:多线程会导致线程间的上下文切换，而使用同步锁会导致多线程之间串行化，会增加程序执行时间，而过长的执行时间可能导致分配给本进程的时间片不够用，从而发生进程间的上下文切换。

2019-06-22

K先生 0

老师好，生产环境线程上下文切换次数一般都是28万次正常吗？

2019-07-01

小美 0

转入到runnable状态时就加载上下文了？不应该是running状态时么

2019-06-29

作者回复

加载上下文一般是加载正在运行和将要运行的线程上下文。

2019-06-30

Lost In The Echo。 0

java能支持协成吗？

2019-06-24

作者回复

可以引入第三方框架支持，目前原生java暂时没有支持协程。

2019-06-26

LW 0

看回复老师说锁升级到重量级锁，就会发生进程间切换，这个点能详细讲讲吗？

2019-06-24

作者回复

当升级到重量级锁后，线程竞争锁资源，将会进入等待队列中，并在等待队列中不断尝试获取锁资源。每次去获取锁资源，都需要通过系统调底层操作系统申请获取Mutex Lock，这个过程就是一次用户态和内核态的切换。

2019-06-26

夏天39度

0

Synchronized由自旋锁升级为重量级锁时会发生上下文切换，获取锁之后，cpu不会在释放锁之前切走，老师，我理解的对吗？

2019-06-24

作者回复

Synchronized由自旋后升级为重量级锁，在存在多个线程竞争的情况下会发生上下文切换。

2019-06-26

-W.LI-

0

老师好!看了大牛们的课后习题回答，大概意思就是偏斜锁，轻量级锁这种不涉及进程切换。然后并发严重膨胀为重量级锁了，发生blocked了或者调用wait(),join()方法释放锁资源，就会触发进程切换了。CAS这种乐观锁，不会触发进程上下文切换?LOCK呢?在调用pack()的时候会导致进程切换么?lock()方法直接获取到锁，没有发生寻找安全点的时候是不是就不会触发进程上下文切换了?
纯属瞎猜，希望老师解惑谢谢。

2019-06-23

作者回复

CAS乐观锁只是一个原子操作，为CPU指令实现，所以操作速度非常快，Java是调用C语言中的函数执行CPU指令操作，不需要进入内核或者切换线程。

而lock竞争锁资源是基于用户态完成，所以竞争锁资源时不会发生进程上下文切换。

2019-06-24

nightmare

0

对于思考题，schronzid在激烈竞争的时候，有可能导致运行的进程里面的线程很快用完cpu时间片，而非自发的被切换，还有一种情况就是stop the vm虚拟机暂停，垃圾回收，那么只能其他进程更多的执行。关于文章的知识点最好带有案列，谢谢

2019-06-22

作者回复

之前讲到的锁优化，以及后面要讲的优化线程池，这些都是一些上下文切换的案例。

2019-06-23

汤小高

0

老师能否提供一份全面的如何定位性能方面问题的工具或者命令了，比如操作系统层面的，也就是文章中提到的，或者JAVA工具层面的。能出一篇这种通过相关命令或者工具定位排查问题的案例最好不过了。

2019-06-22

作者回复

好的，后面我总结一份命令排查工具的使用报告给大家。

2019-06-23

Lsoul

0

老师您好，文中最后提及”而在逻辑相对来说很复杂的场景，等待时间相对较长又或者是需要大

量计算的场景，我建议使用多线程来提高系统的整体性能。”等同于io密集处理与cpu密集运算，而个人所理解的是cpu密集运算通常要降低多线程处理，与上文相悖。是否我理解错误。

2019-06-22

作者回复

文章是建议使用多线程处理。我理解你的意思应该是降低多线程数量吧，这是对的。适当的线程数量来处理cpu密集型计算场景，可以充分利用CPU（线程服务都是多核多个CPU）。

2019-06-23

-W.LI-

0

老师好，有个问题线程切换的时候读到一半的数据会怎么处理啊，高速缓存中的，内存中的，等各个地方的数据。是换出保存，分配到时间片后再换回么?

2019-06-22

作者回复

依然在内存中

2019-06-23

Liam

0

老师好，请教一个问题：

1 非自发场景中，cpu time slice 用完后切换线程，此时被暂停线程是什么状态呢？runnable or blocked, 如果是runnable的话，是不是意味着从runnning到runnable也会导致上下文切换

2019-06-22

作者回复

我们可以调用yield，线程可以从runnning到runnable，我们可以手动编程试试，看看cs是不是增加了。

yield也会导致上下文切换的。

2019-06-23