



Java性能调优实战
刘超
金山软件西山居技术经理

[查看详情](#)

5970 人已学习

18 | 如何设置线程池大小?

19 | 如何用协程来优化多线程业务?

答疑课堂：模块三热点问题解答

加餐 | 什么是数据的强、弱一致性?

模块四 · JVM性能监测及调优
(6讲)

答疑课堂：模块三热点问题解答

刘超 2019-07-04



00:00

讲述：李良 大小：13.13M

07:10

你好，我是刘超。

不知不觉“多线程性能优化”已经讲完了，今天这讲我来解答下各位同学在这个模块集中提出的两大问题，第一个是有关监测上下文切换异常的命令排查工具，第二个是有关 blockingQueue 的内容。

也欢迎你积极留言给我，让我知晓你想了解的内容，或者说出你的困惑，我们共同探讨。下面我就直接切入今天的主题了。

使用系统命令查看上下文切换

在第 15 讲中我提到了上下文切换，其中有用到一些工具进行监测，由于篇幅关系就没有详细介绍，今天我就补充总结几个常用的工具给你。

1. Linux 命令行工具之 vmstat 命令

vmstat 是一款指定采样周期和次数的功能性监测工具，我们可以使用它监控进程上下文切换的情况。

```
[root@localhost conf]# vmstat 1 3
procs-----memory-----swap-----io-----system-----cpu-----
b  swpd  free  buff  cache   si   so    bi    bo    in   cs   us   sy   id   wa   st
1  0 500040 2259324 156404 1634120 0  0  0  0  2  0  0  0  0  100  0  0
0  0 500040 2259184 156404 1634120 0  0  0  0  0 454 923 0  0 100  0  0
0  0 500040 2259076 156404 1634120 0  0  0  0 20 469 999 0  0 100  0  0
```

vmstat 1 3 命令行代表每秒收集一次性能指标，总共获取 3 次。以下为上图中各个性能指标的注释：

- procs
 - r: 等待运行的进程数
 - b: 处于非中断睡眠状态的进程数
- memory
 - swpd: 虚拟内存使用情况
 - free: 空闲的内存
 - buff: 用来作为缓冲的内存在数
 - cache: 缓存大小
- swap
 - si: 从磁盘交换到内存的交换页数数量
 - so: 从内存交换到磁盘的交换页数数量
- io

bi: 发送到块设备的块数
bo: 从块设备接收到的块数

• system

in: 每秒中断数
cs: 每秒上下文切换次数

• cpu

us: 用户 CPU 使用时间
sy: 内核 CPU 系统使用时间
id: 空闲时间
wa: 等待 I/O 时间
st: 运行虚拟机窃取的时间

2. Linux 命令行工具之 pidstat 命令

我们通过上述的 vmstat 命令只能观察到哪个进程的上下文切换出现了异常，那如果是要查看哪个线程的上下文出现了异常呢？

pidstat 命令就可以帮助我们监测到具体线程的上下文切换。pidstat 是 Sysstat 中一个组件，也是一款功能强大的性能监测工具。我们可以通过命令 yum install sysstat 安装该监控组件。

通过 pidstat -help 命令，我们可以查看到有以下几个常用参数可以监测线程的性能：

```
[root@localhost conf]# pidstat -help
usage: pidstat [ options ] [ <interval> ] [ <count> ] ]
options are:
-d [ -h ] [ -i ] [ -l ] [ -r ] [ -s ] [ -t ] [ -u [ <username> ] ] [ -u ]
-v [ -w ] [ -c <command> ] [ -p [ <pid> [ ... ] | SELF | ALL ] ]
-v [ TASK CHILD | ALL ] ]
```

常用参数：

- -u: 默认参数，显示各个进程的 cpu 使用情况；
- -r: 显示各个进程的内存使用情况；
- -d: 显示各个进程的 I/O 使用情况；
- -w: 显示每个进程的上下文切换情况；
- -p: 指定进程号；
- -t: 显示进程中线程的统计信息

首先，通过 pidstat -w -p pid 命令行，我们可以查看到进程的上下文切换：

```
[root@localhost ~]# pidstat -w -p 16079
Linux 3.10.0-514.el7.x86_64 (localhost) 07/02/2019 _x86_64_ (4 CPU)
02:12:34 PM UID PID cswch/s nvcsch/s Command
02:12:34 PM 0 16079 0.00 0.00 java
```

- cswch/s: 每秒主动任务上下文切换数量
- nvcsch/s: 每秒被动任务上下文切换数量

之后，通过 pidstat -w -p pid -t 命令行，我们可以查看到具体线程的上下文切换：

```
[root@localhost ~]# pidstat -w -p 16079 -t
Linux 3.10.0-514.el7.x86_64 (localhost) 07/02/2019 _x86_64_ (4 CPU)
02:14:27 PM UID TID cswch/s nvcsch/s Command
02:14:27 PM 0 16079 0.00 0.00 java
02:14:27 PM 0 16080 0.00 0.00 java
02:14:27 PM 0 16081 0.00 0.00 java
02:14:27 PM 0 16082 0.00 0.00 java
02:14:27 PM 0 16083 0.00 0.00 java
02:14:27 PM 0 16084 0.00 0.00 java
02:14:27 PM 0 16085 0.05 0.00 java
02:14:27 PM 0 16086 0.00 0.00 java
02:14:27 PM 0 16087 0.00 0.00 java
02:14:27 PM 0 16088 0.00 0.00 java
02:14:27 PM 0 16089 0.01 0.00 java
02:14:27 PM 0 16090 0.01 0.00 java
02:14:27 PM 0 16091 0.01 0.00 java
02:14:27 PM 0 16092 0.00 0.00 java
02:14:27 PM 0 16093 0.90 0.00 java
02:14:27 PM 0 16094 0.00 0.00 java
02:14:27 PM 0 16099 0.00 0.00 java
02:14:27 PM 0 16100 0.00 0.00 java
02:14:27 PM 0 16116 0.00 0.00 java
02:14:27 PM 0 16117 0.09 0.00 java
02:14:27 PM 0 16118 0.09 0.00 java
02:14:27 PM 0 16119 0.09 0.00 java
02:14:27 PM 0 16120 0.09 0.00 java
02:14:27 PM 0 16121 0.09 0.00 java
02:14:27 PM 0 16122 0.09 0.00 java
02:14:27 PM 0 16123 0.00 0.00 java
02:14:27 PM 0 16124 0.45 0.00 java
02:14:27 PM 0 16126 0.00 0.00 java
02:14:27 PM 0 16129 0.00 0.00 java
02:14:27 PM 0 16131 1.48 0.00 java
```

3. JDK 工具之 jstack 命令

查看具体线程的上下文切换异常，我们还可以使用 jstack 命令查看线程堆栈的运行情况。jstack 是 JDK 自带的线程堆栈分析工具，使用该命令可以查看或导出 Java 应用程序中的线程堆栈信息。

jstack 最常用的功能就是使用 `jstack pid` 命令查看线程堆栈信息，通常是结合 `pidstat -p pid -t` 一起查看具体线程的状态，也经常用来排查一些死锁的异常。

[illegible]

每个线程堆栈的信息中，都可以查看到线程 ID、线程状态（wait、sleep、running 等状态）以及是否持有锁等。

我们可以通过 `jstack 16079 > /usr/dump` 将线程堆栈信息日志 dump 下来，之后打开 dump 文件，通过查看线程的状态变化，就可以找出导致上下文切换异常的具体原因。例如，系统出现了大量处于 BLOCKED 状态的线程，我们就需要立刻分析代码找出原因。

多线程队列

针对这讲的第一个问题，一份上下文切换的命令排查工具就总结完了。下面我来解答第二个问题，是在 17 讲中呼声比较高的有关 blockingQueue 的内容。

在 Java 多线程应用中，特别是在线程池中，队列的使用率非常高。Java 提供的线程安全队列又分为了阻塞队列和非阻塞队列。

1. 阻塞队列

我们先看下阻塞队列。阻塞队列可以很好地支持生产者和消费者模式的相互等待，当队列为空的时候，消费线程会阻塞等待队列不为空；当队列满了的时候，生产线程会阻塞直到队列不满。

在 Java 线程池中，也用到了阻塞队列。当创建的线程数量超过核心线程数时，新建的任务将会被放到阻塞队列中。我们可以根据自己的业务需求来选择使用哪一种阻塞队列，阻塞队列通常包括以下几种：

- **ArrayBlockingQueue**：一个基于数组结构实现的有界阻塞队列，按 FIFO（先进先出）原则对元素进行排序，使用 ReentrantLock、Condition 来实现线程安全；
- **LinkedBlockingQueue**：一个基于链表结构实现的阻塞队列，同样按 FIFO（先进先出）原则对元素进行排序，使用 ReentrantLock、Condition 来实现线程安全，吞吐量通常要高于 ArrayBlockingQueue；
- **PriorityBlockingQueue**：一个具有优先级的无限阻塞队列，基于二叉堆结构实现的无界限（最大值 Integer.MAX_VALUE - 8）阻塞队列，队列没有实现排序，但每当有数据变更时，都会将最小或最大的数据放在堆最上面的节点上，该队列也是使用了 ReentrantLock、Condition 实现的线程安全；
- **DelayQueue**：一个支持延时获取元素的无界阻塞队列，基于 PriorityBlockingQueue 扩展实现，与其不同的是实现了 Delay 延时接口；
- **SynchronousQueue**：一个不存储多个元素的阻塞队列，每次进行放入数据时，必须等待相应的消费者取走数据后，才可以再次放入数据，该队列使用了两种模式来管理元素，一种是使用先进先出的队列，一种是使用后先进先出的栈，使用哪种模式可以通过构造函数来指定。

Java 线程池 Executors 还实现了以下四种类型的 ThreadPoolExecutor，分别对应以上队列，详情如下：

线程池类型	实现队列
newCachedThreadPool	SynchronousQueue
newFixedThreadPool	LinkedBlockingQueue
newScheduledThreadPool	DelayQueue
newSingleThreadExecutor	LinkedBlockingQueue

2. 非阻塞队列

我们常用的线程安全的非阻塞队列是 `ConcurrentLinkedQueue`，它是一种无界线程安全队列 (FIFO)，基于链表结构实现，利用 CAS 乐观锁来保证线程安全。

下面我们通过源码来分析下该队列的构造、入列以及出列的具体实现。

构造函数：ConcurrentLinkedQueue 由 head、tail 节点组成，每个节点（Node）由节点元素（item）和指向下一个节点的引用（next）组成，节点与节点之间通过 next 关联，从而组成一张链表结构的队列。在队列初始化时，head 节点存储的元素为空，tail 节点等于 head 节点。

```
1 public ConcurrentLinkedQueue() {
2     head = tail = new Node<E>(null);
3 }
4
5 private static class Node<E> {
6     volatile E item;
7     volatile Node<E> next;
8     .
9     .
10 }
11
```

复制代码

入列：当一个线程入列一个数据时，会将该数据封装成一个 Node 节点，并先获取到队列的队尾节点，当确定此时队尾节点的 next 值为 null 之后，再通过 CAS 将新队尾节点的 next 值设为新节点。此时 p != t，也就是设置 next 值成功，然后再通过 CAS 将队尾节点设置为当前节点即可。

```
1 public boolean offer(E e) {
2     checkNotNull(e);
3     // 创建入队节点
4     final Node<E> newNode = new Node<E>(e);
5     // t, p 为尾节点，默认相等，采用失败即重试的方式，直到入队成功
6     for (Node<E> t = tail, p = t;;) {
7         // 获取队尾节点的下一个节点
8         Node<E> q = p.next;
9         // 如果 q 为 null，则代表 p 就是队尾节点
10        if (q == null) {
11            // 将入列节点设置为当前队尾节点的 next 节点
12            if (p.casNext(null, newNode)) {
13                // 判断 tail 节点和 p 节点距离达到两个节点
14                if (p != t) // hop two nodes at a time
15                    // 如果 tail 不是尾节点则将入队节点设置为 tail。
16                    // 如果失败了，那么说明有其他线程已经把 tail 移动过
17                    casTail(t, newNode); // Failure is OK.
18                return true;
19            }
20        }
21        // 如果 p 节点等于 p 的 next 节点，则说明 p 节点和 q 节点都为空，表示队列刚刚
22        else if (p == q)
23            p = (t != (t = tail)) ? t : head;
24        else
25            // Check for tail updates after two hops.
26            p = (p != t && t != (t = tail)) ? t : q;
27    }
28 }
29
```

复制代码

出列：首先获取 head 节点，并判断 item 是否为 null，如果为空，则表示已经有一个线程刚刚进行了出列操作，然后更新 head 节点；如果不为空，则使用 CAS 操作将 head 节点设置为 null，CAS 就会成功地直接返回节点元素，否则还是更新 head 节点。

```
1 public E poll() {
2     // 设置起始点
3     restartFromHead:
4     for (;;) {
5         // p 获取 head 节点
6         for (Node<E> h = head, p = h, q;;) {
7             // 获取头节点元素
8             E item = p.item;
9             // 如果头节点元素不为 null，通过 cas 设置 p 节点引用的元素为 null
10            if (item != null && p.casItem(item, null)) {
11                // Successful CAS is the linearization point
12                // for item to be removed from this queue.
13                if (p != h) // hop two nodes at a time
14                    updateHead(h, ((q = p.next) != null) ? q : p);
15                return item;
16            }
17            // 如果 p 节点的下一个节点为 null，则说明这个队列为空，更新 head 结点
18            else if ((q = p.next) == null) {
19
```

复制代码

```
19         updateHead(h, p);
20         return null;
21     }
22     // 节点出队失败, 重新跳到 restartFromHead 来进行出队
23     else if (p == q)
24         continue restartFromHead;
25     else
26         p = q;
27     }
28 }
29 }
30 }
```

ConcurrentLinkedQueue 是基于 CAS 乐观锁实现的, 在并发时的性能要好于其它阻塞队列, 因此很适合作为高并发场景下的排队队列。

今天的答疑就到这里, 如果你还有其它问题, 请在留言区中提出, 我会一一解答。最后欢迎你点击“请朋友读”, 把今天的内容分享给身边的朋友, 邀请他加入讨论。

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。

胡晓

由作者筛选后的优质留言将会公开显示, 欢迎踊跃留言。

Command + Enter 发表

0/2000字

提交留言

精选留言(5)



-W.LI-

老师好!FGC正常情况多久一次比较合适啊?我们项目1.2天一次FGC老年代给了3G年轻代1G想吧年轻代给多点。有个定时任务, 2小时一次用的线程池。给了40个线程并发请求4K次。设置了空闲回收策略回收核心线程。现在就是定时任务, 每次都新建40个线程一张吃老年代内存。不设置回收这些线程不实用的那个吧小时就一直阻塞。怎么处理比较合适

作者回复: GC在核心业务应用服务中越久发生越合适, 且GC的时间不要太长。一般生产环境的FGC几天一次是比较正常的。40个线程是不是设置太大了, 建议调小一些, 当然需要你们具体压测验证下调小后的性能情况。

年轻代可以调大一些, 如果年轻代太小, 当MinorGC时, 发现年轻代依然存活满对象, 新的对象可能将无法放入到年轻代, 则会通过分配担保机制提前转移年轻代的存活对象到老年代中, 这样反而会增加老年代的负担。默认情况下老年代和新生代是2:1。建议没有特殊情况, 不要固定设置老年代和新生代。

2019-07-04



3



张德

老师 Disruptor是不是比ConcurrentLinkedQueue性能更强呢???

2019-07-15



咬你

老师，通过vmstat参数获取的参数，可否结合一些真实场景，分析下什么样的数据范围属于正常范围，出现什么样的参数，我们就需要重点关注

作者回复: 一般系统出现性能瓶颈，可以结果上下文切换指标进行分析。在之前15讲中，我已经通过一个真实案例讲解了，可以参考下，有什么问题欢迎沟通。

2019-07-04



nightmare

性能好是一方面，如果是抢购应用在需要用有界队列

2019-07-04



Liam

我有2个问题想请教老师：

1 系统出现问题时我们一般会首先关注资源的使用情况，什么情况下可能是是上下文切换过多导致的呢？CPU消耗过高？

2 ConcurrentLinkedQueue是非阻塞的，是否意味着它会消耗过多的CPu

作者回复: CPU消耗过高会引起上下文切换的增加，但并不代表这个就不正常了。正常情况下上下文切换在几百到几千，高峰时段会上升至几万，甚至几十万。

如果上下文长时间处于高位，这个时候我们就要注意了，这种情况有可能是某个线程长期占用CPU，例如之前我提到过的正则表达式出现的严重的回溯问题，就会在某一次回溯时，一直占用CPU，CPU的使用率高居不下，会导致上下文切换激增。

另外一种情况，就是之前你们的业务在高峰期出现的上下文切换在某个值，但是在业务迭代之后，高峰期的上下文切换的值异常高于之前的监控值。比如，我之前说的线程大小调整，导致了高峰期的上下文高出了十几倍之多。

ConcurrentLinkedQueue CAS操作会消耗CPU，但会及时释放，这不足以影响到系统的整体性能。

2019-07-04

