

## 39 | Tomcat进程占用CPU过高怎么办？

2019-08-10 李号双

深入拆解Tomcat & Jetty

[进入课程 >](#)



讲述：李号双

时长 06:30 大小 5.97M



在性能优化这个主题里，前面我们聊过了 Tomcat 的内存问题和网络相关的问题，接下来我们看一下 CPU 的问题。CPU 资源经常会成为系统性能的一个瓶颈，这其中的原因是多方面的，可能是内存泄露导致频繁 GC，进而引起 CPU 使用率过高；又可能是代码中的 Bug 创建了大量的线程，导致 CPU 上下文切换开销。

今天我们就来聊聊 Tomcat 进程的 CPU 使用率过高怎么办，以及怎样一步一步找到问题的根因。

### “Java 进程 CPU 使用率高”的解决思路是什么？

通常我们所说的 CPU 使用率过高，这里面其实隐含着一个用来比较高与低的基准值，比如 JVM 在峰值负载下的平均 CPU 利用率为 40%，如果 CPU 使用率飙到 80% 就可以被认为


是不正常的。

典型的 JVM 进程包含多个 Java 线程，其中一些在等待工作，另一些则正在执行任务。在单个 Java 程序的情况下，线程数可以非常低，而对于处理大量并发事务的互联网后台来说，线程数可能会比较高。

对于 CPU 的问题，最重要的是要找到是**哪些线程在消耗 CPU**，通过线程栈定位到问题代码；如果没有找到个别线程的 CPU 使用率特别高，我们要怀疑到是不是线程上下文切换导致了 CPU 使用率过高。下面我们通过一个实例来学习 CPU 问题定位的过程。

## 定位高 CPU 使用率的线程和代码

1. 写一个模拟程序来模拟 CPU 使用率过高的问题，这个程序会在线程池中创建 4096 个线程。代码如下：

 复制代码

```
1 @SpringBootApplication
2 @EnableScheduling
3 public class DemoApplication {
4
5     // 创建线程池，其中有 4096 个线程。
6     private ExecutorService executor = Executors.newFixedThreadPool(4096);
7     // 全局变量，访问它需要加锁。
8     private int count;
9
10    // 以固定的速率向线程池中加入任务
11    @Scheduled(fixedRate = 10)
12    public void lockContention() {
13        IntStream.range(0, 1000000)
14            .forEach(i -> executor.submit(this::incrementSync));
15    }
16
17    // 具体任务，就是将 count 数加一
18    private synchronized void incrementSync() {
19        count = (count + 1) % 10000000;
20    }
21
22    public static void main(String[] args) {
23        SpringApplication.run(DemoApplication.class, args);
24    }
25
26 }
```

2. 在 Linux 环境下启动程序：

复制代码

```
1 java -Xss256k -jar demo-0.0.1-SNAPSHOT.jar
```

请注意，这里我将线程栈大小指定为 256KB。对于测试程序来说，操作系统默认值 8192KB 过大，因为我们需要创建 4096 个线程。

3. 使用 top 命令，我们看到 Java 进程的 CPU 使用率达到了 262.3%，注意到进程 ID 是 4361。

```
top - 21:41:50 up 392 days, 19:02, 1 user, load average: 2.05, 1.81, 1.23
Tasks: 145 total, 1 running, 144 sleeping, 0 stopped, 0 zombie
%Cpu(s): 68.2 us, 3.6 sy, 0.0 ni, 27.9 id, 0.0 wa, 0.0 hi, 0.1 si, 0.2 st
KiB Mem: 8175216 total, 5704800 used, 2470416 free, 192884 buffers
KiB Swap: 0 total, 0 used, 0 free. 2732192 cached Mem

  PID USER      PR  NI   VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 4361 haosli    20   0 6833676 1.181g 28120 S 262.3 15.1  11:37.91 java
29877 cronus    20   0 3073168 151592 8884 S 24.9 1.9   7709:57 python
    7 root      20   0     0     0     0 S  0.3 0.0   1037:59 rcu_sched
10352 haosli    20   0   28232   3140  2664 R  0.3 0.0    0:00.09 top
13579 mongodb    20   0 1691948 83104 37232 S  0.3 1.0   83:52.89 mongod
15872 haosli    20   0 5828436 399356 31388 S  0.3 4.9   10:34.44 java
    1 root      20   0   29272   4000  2608 S  0.0 0.0    4:01.32 init
    2 root      20   0     0     0     0 S  0.0 0.0    0:03.15 kthreadd
```

4. 接着我们用更精细化的 top 命令查看这个 Java 进程中各线程使用 CPU 的情况：

复制代码


```
1 #top -H -p 4361
```

```
top - 21:47:33 up 392 days, 19:07, 1 user, load average: 1.55, 1.83, 1.44
Threads: 4137 total, 2 running, 4135 sleeping, 0 stopped, 0 zombie
%Cpu(s): 29.8 us, 7.5 sy, 0.0 ni, 62.4 id, 0.0 wa, 0.0 hi, 0.2 si, 0.1 st
KiB Mem: 8175216 total, 6405212 used, 1770004 free, 192896 buffers
KiB Swap: 0 total, 0 used, 0 free. 2741060 cached Mem
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
4387	haosli	20	0	6834084	1.829g	28120	S	42.5	23.5	3:42.95	scheduling-1
4363	haosli	20	0	6834084	1.829g	28120	S	1.0	23.5	1:08.56	GC Thread#0
4368	haosli	20	0	6834084	1.829g	28120	S	1.0	23.5	0:06.17	VM Thread
4379	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	1:08.49	GC Thread#1
4380	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	1:08.26	GC Thread#2
4382	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	1:08.36	GC Thread#3
4799	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.19	pool-1-thread-3
4900	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.17	pool-1-thread-5
4952	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.12	pool-1-thread-5
5002	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.12	pool-1-thread-6
6211	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.11	pool-1-thread-1
6631	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.14	pool-1-thread-2
6847	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.13	pool-1-thread-2
7348	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.12	pool-1-thread-2
7738	haosli	20	0	6834084	1.829g	28120	S	0.7	23.5	0:00.13	pool-1-thread-3
4367	haosli	20	0	6834084	1.829g	28120	S	0.3	23.5	0:00.61	G1 Young RemSet
4428	haosli	20	0	6834084	1.829g	28120	S	0.3	23.5	0:00.13	pool-1-thread-4
4447	haosli	20	0	6834084	1.829g	28120	S	0.3	23.5	0:00.13	pool-1-thread-6


从图上我们可以看到，有个叫“scheduling-1”的线程占用了较多的CPU，达到了42.5%。因此下一步我们要找出这个线程在做什么事情。

5. 为了找出线程在做什么事情，我们需要用 jstack 命令生成线程快照，具体方法是：

 复制代码

```
1 jstack 4361
```

jstack 的输出比较大，你可以将输出写入文件：

 复制代码

```
1 jstack 4361 > 4361.log
```

然后我们打开 4361.log，定位到第 4 步中找到的名为“scheduling-1”的线程，发现它的线程栈如下：



```

"scheduling-1" #17 prio=5 os_prio=0 cpu=198267.66ms elapsed=568.21s tid=0x00007fc69cbac000 nid=0x1123 waiting on condition [0x00007fc6329b2000]
java.lang.Thread.State: WAITING (parking)
  at jdk.internal.misc.Unsafe.park(java.base@12.0.1/Native Method)
  - parking to wait for <0x0000000083e11020> (a java.util.concurrent.locks.ReentrantLock$NonfairSync)
  at java.util.concurrent.locks.LockSupport.park(java.base@12.0.1/LockSupport.java:194)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.parkAndCheckInterrupt(java.base@12.0.1/AbstractQueuedSynchronizer.java:885)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquireQueued(java.base@12.0.1/AbstractQueuedSynchronizer.java:917)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.acquire(java.base@12.0.1/AbstractQueuedSynchronizer.java:1240)
  at java.util.concurrent.locks.ReentrantLock.lock(java.base@12.0.1/ReentrantLock.java:267)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer.signalNotEmpty(java.base@12.0.1/LinkedBlockingQueue.java:173)
  at java.util.concurrent.LinkedBlockingQueue.offer(java.base@12.0.1/LinkedBlockingQueue.java:421)
  at java.util.concurrent.ThreadPoolExecutor.execute(java.base@12.0.1/ThreadPoolExecutor.java:1347)
  at java.util.concurrent.AbstractExecutorService.submit(java.base@12.0.1/AbstractExecutorService.java:118)
  at com.example.tomcat.test.demo.DemoApplication.lambda$lockContention$0(DemoApplication.java:27)
  at com.example.tomcat.test.demo.DemoApplication$$Lambda$617/0x0000000080155040.accept(Unknown Source)
  at java.util.stream.Streams$RangeIntSplitter.forEachRemaining(java.base@12.0.1/Streams.java:104)
  at java.util.stream.IntPipeline$Head.forEach(java.base@12.0.1/IntPipeline.java:593)
  at com.example.tomcat.test.demo.DemoApplication.lockContention(DemoApplication.java:27)
  at jdk.internal.reflect.GeneratedMethodAccessor30.invoke(Unknown Source)
  at jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(java.base@12.0.1/DelegatingMethodAccessorImpl.java:43)
  at java.lang.reflect.Method.invoke(java.base@12.0.1/Method.java:567)
  at org.springframework.scheduling.support.ScheduledMethodRunnable.run(ScheduledMethodRunnable.java:84)
  at org.springframework.scheduling.support.DelegatingErrorHandlingRunnable.run(DelegatingErrorHandlingRunnable.java:54)
  at java.util.concurrent.Executors$RunnableAdapter.call(java.base@12.0.1/Executors.java:515)
  at java.util.concurrent.FutureTask.runAndReset(java.base@12.0.1/FutureTask.java:305)
  at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(java.base@12.0.1/ScheduledThreadPoolExecutor.java:305)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(java.base@12.0.1/ThreadPoolExecutor.java:1128)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(java.base@12.0.1/ThreadPoolExecutor.java:628)
  at java.lang.Thread.run(java.base@12.0.1/Thread.java:835)

```

从线程栈中我们看到了AbstractExecutorService.submit这个函数调用，说明它是Spring Boot 启动的周期性任务线程，向线程池中提交任务，这个线程消耗了大量 CPU。

## 进一步分析上下文切换开销

一般来说，通过上面的过程，我们就能定位到大量消耗 CPU 的线程以及有问题的代码，比如死循环。但是对于这个实例的问题，你是否发现这样一个情况：Java 进程占用的 CPU 是 262.3%，而“scheduling-1”线程只占用了 42.5% 的 CPU，那还有将近 220% 的 CPU 被谁占用了呢？

不知道你注意到没有，我们在第 4 步用 `top -H -p 4361` 命令看到的线程列表中还有许多名为“pool-1-thread-x”的线程，它们单个的 CPU 使用率不高，但是似乎数量比较多。你可能已经猜到，这些就是线程池中干活的线程。那剩下的 220% 的 CPU 是不是被这些线程消耗了呢？

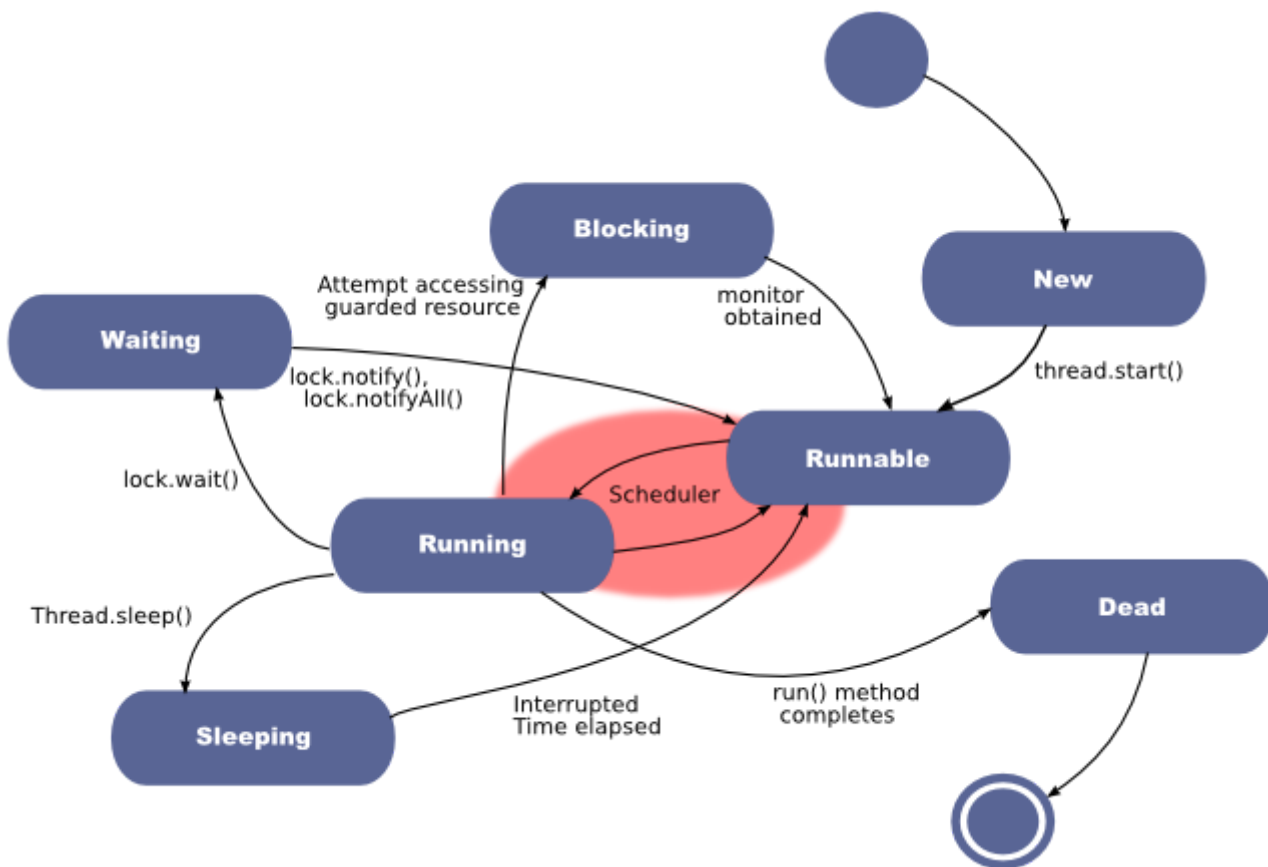
要弄清楚这个问题，我们还需要看 jstack 的输出结果，主要是看这些线程池中的线程是不是真的在干活，还是在“休息”呢？

```

"pool-1-thread-2" #19 prio=5 os_prio=0 cpu=110.27ms elapsed=568.21s tid=0x00007fc618004000 nid=0x1125 waiting on condition [0x00007fc632930000]
java.lang.Thread.State: WAITING (parking)
  at jdk.internal.misc.Unsafe.park(java.base@12.0.1/Native Method)
  - parking to wait for <0x0000000083dade18> (a java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject)
  at java.util.concurrent.locks.LockSupport.park(java.base@12.0.1/LockSupport.java:194)
  at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.await(java.base@12.0.1/AbstractQueuedSynchronizer.java:2081)
  at java.util.concurrent.LinkedBlockingQueue.take(java.base@12.0.1/LinkedBlockingQueue.java:433)
  at java.util.concurrent.ThreadPoolExecutor.getTask(java.base@12.0.1/ThreadPoolExecutor.java:1054)
  at java.util.concurrent.ThreadPoolExecutor.runWorker(java.base@12.0.1/ThreadPoolExecutor.java:1114)
  at java.util.concurrent.ThreadPoolExecutor$Worker.run(java.base@12.0.1/ThreadPoolExecutor.java:628)
  at java.lang.Thread.run(java.base@12.0.1/Thread.java:835)

```

通过上面的图我们发现这些 “pool-1-thread-x” 线程基本都处于 WAITING 的状态，那什么是 WAITING 状态呢？或者说 Java 线程都有哪些状态呢？你可以通过下面的图来理解一下：



从图上我们看到 “Blocking” 和 “Waiting” 是两个不同的状态，我们要注意它们的区别：

Blocking 指的是一个线程因为等待临界区的锁（Lock 或者 synchronized 关键字）而被阻塞的状态，请你注意的是处于这个状态的线程**还没有拿到锁**。

Waiting 指的是一个线程拿到了锁，但是需要等待其他线程执行某些操作。比如调用了 Object.wait、Thread.join 或者 LockSupport.park 方法时，进入 Waiting 状态。**前提是这个线程已经拿到锁了**，并且在进入 Waiting 状态前，操作系统层面会自动释放锁，当等待条件满足，外部调用了 Object.notify 或者 LockSupport.unpark 方法，线程会重新竞争锁，成功获得锁后才能进入到 Runnable 状态继续执行。

回到我们的 “pool-1-thread-x” 线程，这些线程都处在 “Waiting” 状态，从线程栈我们看到，这些线程 “等待” 在 getTask 方法调用上，线程尝试从线程池的队列中取任务，但是队列为空，所以通过 LockSupport.park 调用进到了 “Waiting” 状态。那 “pool-1-thread-x” 线程有多少个呢？通过下面这个命令来统计一下，结果是 4096，正好跟线程池中的线程数相等。

```
#grep -o 'pool-1-thread' 4361.log | wc -l
4096
```

你可能好奇了，那剩下的 220% 的 CPU 到底被谁消耗了呢？分析到这里，我们应该怀疑 CPU 的上下文切换开销了，因为我们看到 Java 进程中的线程数比较多。下面我们通过 vmstat 命令来查看一下操作系统层面的线程上下文切换活动：

```
#vmstat 1 100
procs -----memory----- ---swap-- -----io----- -system-- -----cpu-----
r  b    swpd   free   buff  cache   si   so    bi   bo    in   cs  us  sy  id  wa  st
2  0      0 3022252 192884 2732052    0    0     0    5     0    0  2  1 98  0  0
3  0      0 3019256 192884 2732052    0    0     0    0 4442 9607 77  2 21  0  0
7  0      0 2945796 192884 2732052    0    0     0    0 4201 8960 78  4 18  0  0
15 0      0 2871700 192884 2732052    0    0     0    0 2735 4165 79  8 13  0  0
2  0      0 2918232 192884 2732060    0    0     0   100 3342 7703 72  8 20  0  0
4  0      0 2818648 192884 2732060    0    0     0   16 4788 10839 68  4 27  0  0
3  0      0 2764576 192884 2732060    0    0     0    0 4282 9042 71  4 26  0  0
4  0      0 2561888 192884 2732060    0    0     0    0 4242 10099 70  5 25  0  0
2  0      0 2408692 192884 2732064    0    0     0    0 4340 9888 71  5 24  0  0
4  2      0 2223656 192884 2732064    0    0     0   64 3803 8438 75  5 19  0  0
1  0      0 2102620 192884 2732068    0    0     0    4 21705 42496 78  7 15  0  0
4  0      0 2157664 192884 2732068    0    0     0    0 3879 7925 88  1 11  0  0
7  0      0 2109328 192884 2732068    0    0     0    0 3928 8490 83  2 15  0  0
4  0      0 2078456 192884 2732068    0    0     0    0 4350 9053 88  3  9  0  0
4  0      0 2076720 192884 2732068    0    0     0    0 4284 10073 69  3 28  0  0
5  0      0 1992536 192884 2732068    0    0     0   16 5172 12311 77  3 20  0  0
```

如果你还不太熟悉 vmstat，可以在[这里](#)学习如何使用 vmstat 和查看结果。其中 cs 那一栏表示线程上下文切换次数，in 表示 CPU 中断次数，我们发现这两个数字非常高，基本证实了我们的猜测，线程上下文切切换消耗了大量 CPU。那么问题来了，具体是哪个进程导致的呢？

我们停止 Spring Boot 测试程序，再次运行 vmstat 命令，会看到 in 和 cs 都大幅下降了，这样就证实了引起线程上下文切换开销的 Java 进程正是 4361。

```
#vmstat 1
procs -----memory----- ---swap-- -----io----- -system-- -----cpu-----
r  b    swpd   free   buff  cache   si   so    bi    bo    in   cs  us  sy  id  wa  st
0  0      0 3964304 192956 2742904    0    0     0     5     0    0   2   1  98   0   0
0  0      0 3964380 192956 2742904    0    0     0     0  401  657   1   0 100   0   0
0  0      0 3964348 192956 2742904    0    0     0     0  387  630   0   0 100   0   0
0  0      0 3964252 192956 2742904    0    0     0     0  382  628   0   0  99   0   0
0  0      0 3964380 192956 2742904    0    0     0     0  389  646   1   0  99   0   0
0  0      0 3964380 192956 2742904    0    0     0    48  430  695   0   0  99   0   0
0  0      0 3964348 192956 2742904    0    0     0     0  423  679   0   0 100   0   0
0  0      0 3964284 192956 2742904    0    0     0     0  400  645   0   0 100   0   0
0  0      0 3964412 192956 2742904    0    0     0     0  370  618   1   0  99   0   0
0  0      0 3964348 192956 2742908    0    0     0     0  374  608   0   0  99   0   0
2  0      0 3948568 192956 2742908    0    0     0    60  636  883  11   4  85   0   0
```

## 本期精华

当我们遇到 CPU 过高的问题时，首先要定位是哪个进程的导致的，之后可以通过 `top -H -p pid` 命令定位到具体的线程。其次还要通 `jstack` 查看线程的状态，看看线程的个数或者线程的状态，如果线程数过多，可以怀疑是线程上下文切换的开销，我们可以通过 `vmstat` 和 `pidstat` 这两个工具进行确认。

## 课后思考

哪些情况可能导致程序中的线程数失控，产生大量线程呢？

不知道今天的内容你消化得如何？如果还有疑问，请大胆的在留言区提问，也欢迎你把你的课后思考和心得记录下来，与我和其他同学一起讨论。如果你觉得今天有所收获，欢迎你把它的分享给你的朋友。



# 深入拆解 Tomcat & Jetty

从源码角度深度探索 Java 中间件

李号双

eBay 技术主管



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。

上一篇 38 | Tomcat拒绝连接原因分析及网络优化

## 精选留言 (9)

写留言



a、

2019-08-10

- 1.使用了Java的newCachedThreadPool，因为最大线程数是int最大值
- 2.自定义线程池最大线程数设置不合理
- 3.线程池的拒绝策略，选择了如果队列满了并且线程达到最大线程数后，提交的任务交给提交任务线程处理

展开 ∨

作者回复: 赞



1



新世界

2019-08-10

线程池和等待队列设置不合理以及拒绝策略设置不合理会导致线程数失控，比如线程池设置小，等到队列也不大，拒绝策略选择用主线程继续执行，瞬间大量请求，会导致等到队列占满，进而用主线程执行任务，导致tomcat线程被打满，线程数失控

展开 ▾



1



802.11

2019-08-10

老师这些都是一些实时的操作，但是大部分情况CPU高的时候并没有及时的在服务器上观察，一旦错过了这个发生的时间点，事后该怎样去判断和定位呢

展开 ▾

作者回复: 这确实是个问题。监控系统一般会将各种指标包括cpu，内存等存下来，但是打印出线程栈，heapdump这些需要手动操作



802.11

2019-08-10

TIMED\_WAITING 是什么意思呢？有什么寓意呢

展开 ▾

作者回复: 根据TCP协议，主动发起关闭的一方，会进入TIME\_WAIT状态



陆离

2019-08-10

容器在启动起来之后就被kill掉的原因有哪些？和CPU过高有关系吗

作者回复: 如何确定是“被kill”呢，或者是异常退出？如果是被kill，看是被谁kill了



许童童

2019-08-10

哪些情况可能导致程序中的线程数失控，产生大量线程呢？

创建线程池时参数是计算出来的，而计算的过程是有bug的，导致结果有问题，从而创建

了大量线程。

这种需要对程序进行测试，线上持续进行性能监控，发现并解决问题。

展开 ▾



a、

2019-08-10

1.使用了Java的newCachedThreadPool，因为最大线程数是int最大值

2.

展开 ▾



-W.LI-

2019-08-10

用线程池创建线程，设置合理的最大线程数。

之前遇见过压测十几个接口200并发下cpu使用率90%可是看了大多都是4%消耗。当时确实发现全部加起来没到90%，最后也没找到原因，就不了了之了。。线上多加了太服务器

展开 ▾



nimil

2019-08-10

👍赞

展开 ▾

