



# 面试官问：平时碰到系统CPU飙高和频繁GC，你会怎么排查？



占小狼 关注

45 2019.08.17 15:16:41 字数 4,269 阅读 13,878

处理过线上问题的同学基本上都会遇到系统突然运行缓慢，CPU 100%，以及Full GC次数过多的问题。当然，这些问题的最终导致的直观现象就是系统运行缓慢，并且有大量的报警。本文主要针对系统运行缓慢这一问题，提供该问题的排查思路，从而定位出问题的代码点，进而提供解决该问题的思路。

对于线上系统突然产生的运行缓慢问题，如果该问题导致线上系统不可用，那么首先需要做的就是，导出jstack和内存信息，然后重启系统，尽快保证系统的可用性。这种情况可能的原因主要有两种：

- 代码中某个位置读取数据量较大，导致系统内存耗尽，从而导致Full GC次数过多，系统缓慢；
- 代码中有比较耗CPU的操作，导致CPU过高，系统运行缓慢；

相对来说，这是出现频率最高的两种线上问题，而且它们会直接导致系统不可用。另外有几种情况也会导致某个功能运行缓慢，但是不至于导致系统不可用：

- 代码某个位置有阻塞性的操作，导致该功能调用整体比较耗时，但出现是比较随机的；
- 某个线程由于某种原因而进入WAITING状态，此时该功能整体不可用，但是无法复现；
- 由于锁使用不当，导致多个线程进入死锁状态，从而导致系统整体比较缓慢。

对于这三种情况，通过查看CPU和系统内存情况是无法查看出具体问题的，因为它们相对来说都是具有一定阻塞性操作，CPU和系统内存使用情况都不高，但是功能却很慢。下面我们就通过查看系统日志来一步一步甄别上述几种问题。

## 1. Full GC次数过多

相对来说，这种情况是最容易出现的，尤其是新功能上线时。对于Full GC较多的情况，其主要有如下两个特征：

- 线上多个线程的CPU都超过了100%，通过jstack命令可以看到这些线程主要是垃圾回收线程
- 通过jstat命令监控GC情况，可以看到Full GC次数非常多，并且次数在不断增加。

首先我们可以使用top命令查看系统CPU的占用情况，如下是系统CPU较高的一个示例：

```
1 |
2 | top - 08:31:10 up 30 min,  0 users,  load average: 0.73, 0.58, 0.34
3 | KiB Mem:  2046460 total,  1923864 used,   122596 free,    14388 buffers
4 | KiB Swap: 1048572 total,      0 used, 1048572 free. 1192352 cached Mem
5 |
```



占小狼

总资产257

关注

Stack Overflow上200万阅读量的一个提问：Java中怎么快速把...

阅读 4,357

阿里问题定位神器 Arthas 的骚操作，定位线上BUG，超给力

阅读 4,594

### 推荐阅读

top+jstack查找线上CPU占用最高的线程

阅读 332

线上cpu满载排查过程

阅读 257

cpu高的怎么回事(二)

阅读 173

JAVA线上故障排查全套路

阅读 253

JVM——JVM调优

阅读 830



可以看到，有一个Java程序此时CPU占用量达到了98.8%，此时我们可以复制该进程id9，并且使用如下命令查看呢该进程的各个线程运行情况：

```
1 |
2 | top -Hp 9
3 |
```

该进程下的各个线程运行情况如下：

```
1 | top - 08:31:16 up 30 min, 0 users, load average: 0.75, 0.59, 0.35
2 | Threads: 11 total, 1 running, 10 sleeping, 0 stopped, 0 zombie
3 | %Cpu(s): 3.5 us, 0.6 sy, 0.0 ni, 95.9 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
4 | KiB Mem: 2046460 total, 1924856 used, 121604 free, 14396 buffers
5 | KiB Swap: 1048572 total, 0 used, 1048572 free. 1192532 cached Mem
6 |
7 | PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ COMMAND
8 | 10 root 20 0 2557160 289824 15872 R 79.3 14.2 0:41.49 java
9 | 11 root 20 0 2557160 289824 15872 S 13.2 14.2 0:06.78 java
10 |
```

可以看到，在进程为9的Java程序中各个线程的CPU占用情况，接下来我们可以通过jstack命令查看线程id为10的线程为什么耗费CPU最高。需要注意的是，在jstack命令展示的结果中，线程id都转换成了十六进制形式。可以用如下命令查看转换结果，也可以找一个科学计算器进行转换：

```
1 | root@a39de7e7934b:/# printf "%x\n" 10
2 | a
3 |
```

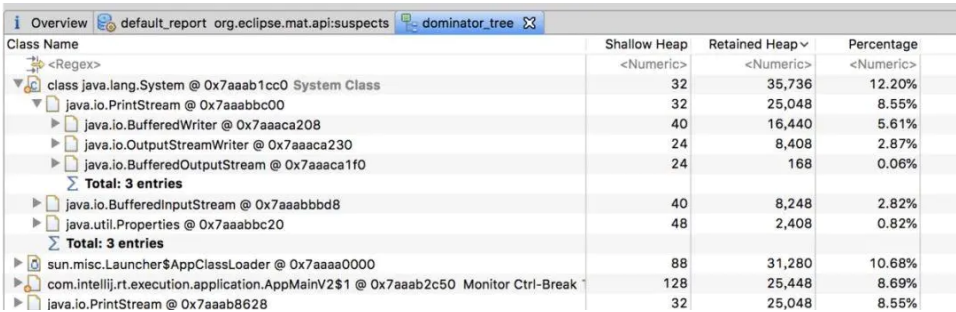
这里打印结果说明该线程在jstack中的展现形式为0xa，通过jstack命令我们可以看到如下信息：

```
1 |
2 | "main" #1 prio=5 os_prio=0 tid=0x00007f8718009800 nid=0xb runnable [0x00007f871fe41000]
3 |   java.lang.Thread.State: RUNNABLE
4 |     at com.aibaobei.chapter2.eg2.UserDemo.main(UserDemo.java:9)
5 |
6 | "VM Thread" os_prio=0 tid=0x00007f871806e000 nid=0xa runnable
7 |
```

这里的VM Thread一行的最后显示nid=0xa，这里nid的意思就是操作系统线程id的意思。而VM Thread指的就是垃圾回收的线程。这里我们基本上可以确定，当前系统缓慢的原因主要是垃圾回收过于频繁，导致GC停顿时间较长。我们通过如下命令可以查看GC的情况：

```
1 | root@8d36124607a0:/# jstat -gcutil 9 1000 10
2 | S0 S1 E O M CCS YGC YGCT FGC FGCT GCT
3 | 0.00 0.00 0.00 75.07 59.09 59.60 3259 0.919 6517 7.715 8.635
4 | 0.00 0.00 0.00 0.08 59.09 59.60 3306 0.930 6611 7.822 8.752
5 | 0.00 0.00 0.00 0.08 59.09 59.60 3351 0.943 6701 7.924 8.867
6 | 0.00 0.00 0.00 0.08 59.09 59.60 3397 0.955 6793 8.029 8.984
7 |
```

导致的内存溢出呢，这个可以dump出内存日志，然后通过eclipse的mat工具进行查看，如下是其展示的一个对象树结构：



Class Name	Shallow Heap	Retained Heap	Percentage
<Regex>	<Numeric>	<Numeric>	<Numeric>
class java.lang.System @ 0x7aaab1cc0 System Class	32	35,736	12.20%
java.io.PrintStream @ 0x7aaabbc00	32	25,048	8.55%
java.io.BufferedWriter @ 0x7aaaca208	40	16,440	5.61%
java.io.OutputStreamWriter @ 0x7aaaca230	24	8,408	2.87%
java.io.BufferedOutputStream @ 0x7aaaca1f0	24	168	0.06%
Total: 3 entries			
java.io.BufferedReader @ 0x7aaabbbd8	40	8,248	2.82%
java.util.Properties @ 0x7aaabbc20	48	2,408	0.82%
Total: 3 entries			
sun.misc.Launcher\$AppClassLoader @ 0x7aaaa0000	88	31,280	10.68%
com.intellij.rt.execution.application.AppMainV2\$1 @ 0x7aaab2c50 Monitor Ctrl-Break	128	25,448	8.69%
java.io.PrintStream @ 0x7aaab8628	32	25,048	8.55%

image

经过mat工具分析之后，我们基本上就能确定内存中主要是哪个对象比较消耗内存，然后找到该对象的创建位置，进行处理即可。这里的主要是PrintStream最多，但是我们也可以看到，其内存消耗量只有12.2%。也就是说，其还不足以导致大量的Full GC，此时我们需要考虑另外一种情况，就是代码或者第三方依赖的包中有显示的System.gc()调用。这种情况我们查看dump内存得到的文件即可判断，因为会打印GC原因：

```
1 |
2 | [Full GC (System.gc()) [Tenured: 262546K->262546K(349568K), 0.0014879 secs] 262546K->2
3 | [GC (Allocation Failure) [DefNew: 2795K->0K(157248K), 0.0001504 secs][Tenured: 262546K
4 |
```

比如这里第一次GC是由于System.gc()的显示调用导致的，而第二次GC则是JVM主动发起的。总结来说，对于Full GC次数过多，主要有以下两种原因：

- 代码中一次获取了大量的对象，导致内存溢出，此时可以通过eclipse的mat工具查看内存中有哪些对象比较多；
- 内存占用不高，但是Full GC次数还是比较多，此时可能是显示的 `System.gc()` 调用导致GC次数过多，这可以通过添加 `-XX:+DisableExplicitGC` 来禁用JVM对显示GC的响应。

## 2. CPU过高

在前面第一点中，我们讲到，CPU过高可能是系统频繁的进行Full GC，导致系统缓慢。而我们平常也肯能遇到比较耗时的计算，导致CPU过高的情况，此时查看方式其实与上面的非常类似。首先我们通过top命令查看当前CPU消耗过高的进程是哪个，从而得到进程id；然后通过top -Hp <pid>来查看该进程中有哪些线程CPU过高，一般超过80%就是比较高的，80%左右是合理情况。这样我们就能得到CPU消耗比较高的线程id。接着通过该线程id的十六进制表示在jstack日志中查看当前线程具体的堆栈信息。

在这里我们就可以区分导致CPU过高的原因具体是Full GC次数过多还是代码中有比较耗时的计算了。如果是Full GC次数过多，那么通过jstack得到的线程信息会是类似于VM Thread之类的线程，而如果是代码中有比较耗时的计算，那么我们得到的就是一个线程的具体堆栈信息。如下是一个代码中有比较耗时的计算，导致CPU过高的线程信息：

```
at java.io.BufferedOutputStream.flush(BufferedOutputStream.java:140)
- locked <0x00000006c0137020> (a java.io.BufferedOutputStream)
at java.io.PrintStream.write(PrintStream.java:482)
- locked <0x00000006c0137000> (a java.io.PrintStream)
at sun.nio.cs.StreamEncoder.writeBytes(StreamEncoder.java:221)
at sun.nio.cs.StreamEncoder.implFlushBuffer(StreamEncoder.java:291)
at sun.nio.cs.StreamEncoder.flushBuffer(StreamEncoder.java:104)
- locked <0x00000006c0137140> (a java.io.OutputStreamWriter)
at java.io.OutputStreamWriter.flushBuffer(OutputStreamWriter.java:185)
at java.io.PrintStream.newLine(PrintStream.java:546)
- locked <0x00000006c0137000> (a java.io.PrintStream)
at java.io.PrintStream.println(PrintStream.java:807)
- locked <0x00000006c0137000> (a java.io.PrintStream)
at com.aibaobei.user.controller.UserController.compute(UserController.java:34)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)
at java.lang.reflect.Method.invoke(Method.java:498)
at org.springframework.web.method.support.InvocableHandlerMethod.doInvoke(InvocableHandlerMethod.java:189)
at org.springframework.web.method.support.InvocableHandlerMethod.invokeForRequest(InvocableHandlerMethod.java:138)
at org.springframework.web.servlet.mvc.method.annotation.ServletInvocableHandlerMethod.invokeAndHandle(ServletInvocableHandlerMethod.java:102)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.invokeHandlerMethod(RequestMappingHandlerAdapter.java:895)
at org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter.handleInternal(RequestMappingHandlerAdapter.java:800)
at org.springframework.web.servlet.mvc.method.AbstractHandlerMethodAdapter.handle(AbstractHandlerMethodAdapter.java:87)
at org.springframework.web.servlet.DispatcherServlet.doDispatch(DispatcherServlet.java:1838)
at org.springframework.web.servlet.DispatcherServlet.doService(DispatcherServlet.java:942)
```

image

这里可以看到，在请求UserController的时候，由于该Controller进行了一个比较耗时的调用，导致该线程的CPU一直处于100%。我们可以根据堆栈信息，直接定位到UserController的34行，查看代码中具体是什么原因导致计算量如此之高。

### 3. 不定期出现的接口耗时现象

对于这种情况，比较典型的例子就是，我们某个接口访问经常需要2~3s才能返回。这是比较麻烦的一种情况，因为一般来说，其消耗的CPU不多，而且占用的内存也不高，也就是说，我们通过上述两种方式进行排查是无法解决这种问题的。而且由于这样的接口耗时比较大的问题是不定时出现的，这就导致了我们在通过jstack命令即使得到了线程访问的堆栈信息，我们也没法判断具体哪个线程是正在执行比较耗时操作的线程。

对于不定时出现的接口耗时比较严重的问题，我们的定位思路基本如下：首先找到该接口，通过压测工具不断加大访问力度，如果说该接口中有某个位置是比较耗时的，由于我们的访问的频率非常高，那么大多数的线程最终都将阻塞于该阻塞点，这样通过多个线程具有相同的堆栈日志，我们基本上就可以定位到该接口中比较耗时的代码的位置。如下是一个代码中有比较耗时的阻塞操作通过压测工具得到的线程堆栈日志：

```
1
2 "http-nio-8080-exec-2" #29 daemon prio=5 os_prio=31 tid=0x00007fd08cb26000 nid=0x9603
3   java.lang.Thread.State: TIMED_WAITING (sleeping)
4     at java.lang.Thread.sleep(Native Method)
5     at java.lang.Thread.sleep(Thread.java:340)
6     at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
7     at com.aibaobei.user.controller.UserController.detail(UserController.java:18)
8
9 "http-nio-8080-exec-3" #30 daemon prio=5 os_prio=31 tid=0x00007fd08cb27000 nid=0x6203
10   java.lang.Thread.State: TIMED_WAITING (sleeping)
11     at java.lang.Thread.sleep(Native Method)
12     at java.lang.Thread.sleep(Thread.java:340)
13     at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
14     at com.aibaobei.user.controller.UserController.detail(UserController.java:18)
15
16 "http-nio-8080-exec-4" #31 daemon prio=5 os_prio=31 tid=0x00007fd08d0fa000 nid=0x6403
17   java.lang.Thread.State: TIMED_WAITING (sleeping)
18     at java.lang.Thread.sleep(Native Method)
19     at java.lang.Thread.sleep(Thread.java:340)
20     at java.util.concurrent.TimeUnit.sleep(TimeUnit.java:386)
21     at com.aibaobei.user.controller.UserController.detail(UserController.java:18)
22
```

## 4. 某个线程进入WAITING状态

对于这种情况，这是比较罕见的一种情况，但是也是有可能出现的，而且由于其具有一定的“不可复现性”，因而在排查的时候是非常难以发现的。笔者曾经就遇到过类似的这种情况，具体的场景是，在使用CountDownLatch时，由于需要每一个并行的任务都执行完成之后才会唤醒主线程往下执行。而当时我们是通过CountDownLatch控制多个线程连接并导出用户的gmail邮箱数据，这其中有一个线程连接上了用户邮箱，但是连接被服务器挂起了，导致该线程一直在等待服务器的响应。最终导致我们的主线程和其余几个线程都处于WAITING状态。

对于这样的问题，查看过jstack日志的读者应该都知道，正常情况下，线上大多数线程都是处于TIMED\_WAITING状态，而我们这里出问题的线程所处的状态与其是一模一样的，这就非常容易混淆我们的判断。解决这个问题的思路主要如下：

- 通过grep在jstack日志中找出所有的处于 **TIMED\_WAITING** 状态的线程，将其导出到某个文件中，如a1.log，如下是一个导出的日志文件示例：

```
1 | "Attach Listener" #13 daemon prio=9 os_prio=31 tid=0x00007fe690064000 nid=0xd07 waitin
2 | "DestroyJavaVM" #12 prio=5 os_prio=31 tid=0x00007fe690066000 nid=0x2603 waiting on con
3 | "Thread-0" #11 prio=5 os_prio=31 tid=0x00007fe690065000 nid=0x5a03 waiting on conditio
4 | "C1 CompilerThread3" #9 daemon prio=9 os_prio=31 tid=0x00007fe68c00a000 nid=0xa903 wai
5 |
```

- 等待一段时间之后，比如10s，再次对jstack日志进行grep，将其导出到另一个文件，如a2.log，结果如下所示：

```
1 |
2 | "DestroyJavaVM" #12 prio=5 os_prio=31 tid=0x00007fe690066000 nid=0x2603 waiting on con
3 | "Thread-0" #11 prio=5 os_prio=31 tid=0x00007fe690065000 nid=0x5a03 waiting on conditio
4 | "VM Periodic Task Thread" os_prio=31 tid=0x00007fe68d114000 nid=0xa803 waiting on cond
```

- 重复步骤2，待导出34个文件之后，我们对导出的文件进行对比，找出其中在这几个文件中一直都存在的用户线程，这个线程基本上就可以确认是包含了处于等待状态有问题的线程。因为正常的请求线程是不会在20<sup>30</sup>s之后还是处于等待状态的。
- 经过排查得到这些线程之后，我们可以继续对其堆栈信息进行排查，如果该线程本身就应该处于等待状态，比如用户创建的线程池中处于空闲状态的线程，那么这种线程的堆栈信息中是不会包含用户自定义的类的。这些都可以排除掉，而剩下的线程基本上就可以确认是我们要找的有问题的线程。通过其堆栈信息，我们就可以得出具体是在哪个位置的代码导致该线程处于等待状态了。

这里需要说明的是，我们在判断是否为用户线程时，可以通过线程最前面的线程名来判断，因为一般的框架的线程命名都是非常规范的，我们通过线程名就可以直接判断得出该线程是某些框架中的线程，这种线程基本上可以排除掉。而剩余的，比如上面的Thread-0，以及我们可以辨别的自定义线程名，这些都是我们需要排查的对象。

经过上面的方式进行排查之后，我们基本上就可以得出这里的Thread-0就是我们要找的线程，通过查看其堆栈信息，我们就可以得到具体是在哪个位置导致其处于等待状态了。如下示例中



简书

首页

下载APP

IT技术

搜索

关注

beta

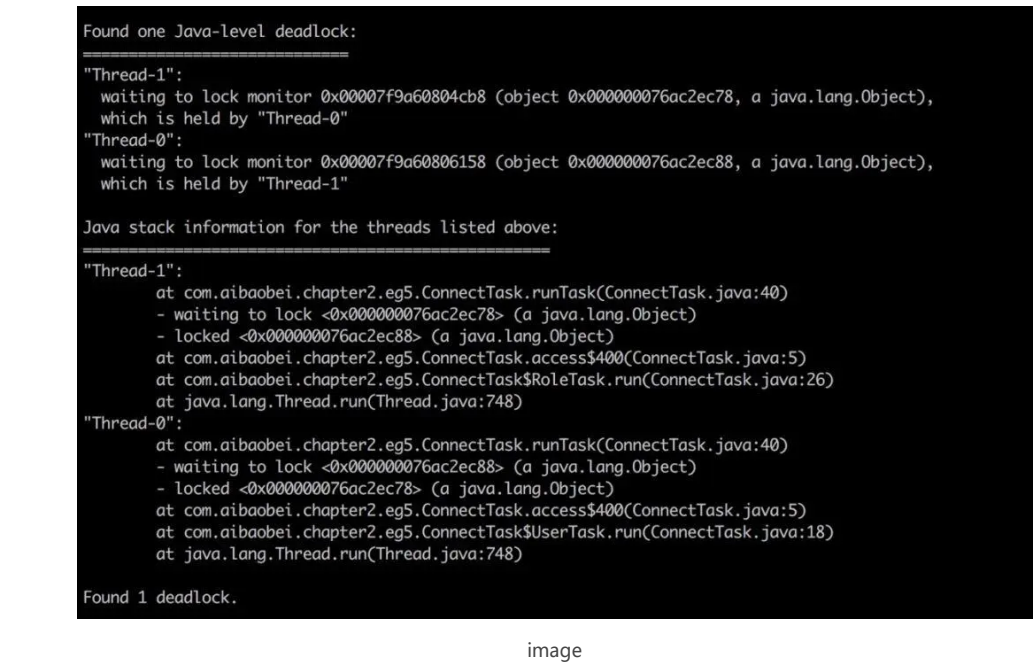
登录

注册

```
3 java.lang.Thread.State: WAITING (parking)
4   at sun.misc.Unsafe.park(Native Method)
5   at java.util.concurrent.locks.LockSupport.park(LockSupport.java:304)
6   at com.aibaobei.chapter2.eg4.SyncTask.lambda$main$0(SyncTask.java:8)
7   at com.aibaobei.chapter2.eg4.SyncTask$Lambda$1/1791741888.run(Unknown Source)
8   at java.lang.Thread.run(Thread.java:748)
9
```

### 5. 死锁

对于死锁，这种情况基本上很容易发现，因为jstack可以帮助我们检查死锁，并且在日志中打印具体的死锁线程信息。如下是一个产生死锁的一个jstack日志示例：



image

可以看到，在jstack日志的底部，其直接帮我们分析了日志中存在哪些死锁，以及每个死锁的线程堆栈信息。这里我们有两个用户线程分别在等待对方释放锁，而被阻塞的位置都是在ConnectTask的第5行，此时我们就可以直接定位到该位置，并且进行代码分析，从而找到产生死锁的原因。

### 6. 小结

本文主要讲解了线上可能出现的五种导致系统缓慢的情况，详细分析了每种情况产生时的现象，已经根据现象我们可以通过哪些方式定位得到是这种原因导致的系统缓慢。简要的说，我们进行线上日志分析时，主要可以分为如下步骤：

- 通过 `top` 命令查看CPU情况，如果CPU比较高，则通过 `top -Hp <pid>` 命令查看当前进程的各个线程运行情况，找出CPU过高的线程之后，将其线程id转换为十六进制的表现形式，然后在jstack日志中查看该线程主要在进行的工作。这里又分为两种情况
- 如果是正常的用户线程，则通过该线程的堆栈信息查看其具体是在哪处用户代码处运行比较消耗CPU；

出之后将内存情况放到eclipse的mat工具中进行分析即可得出内存中主要是什么对象比较消耗内存，进而可以处理相关代码；

- 如果通过 `top` 命令看到CPU并不高，并且系统内存占用率也比较低。此时就可以考虑是否是由于另外三种情况导致的问题。具体的可以根据具体情况分析：
- 如果是接口调用比较耗时，并且是不定时出现，则可以通过压测的方式加大阻塞点出现的频率，从而通过 `jstack` 查看堆栈信息，找到阻塞点；
- 如果是某个功能突然出现停滞的状况，这种情况也无法复现，此时可以通过多次导出 `jstack` 日志的方式对比哪些用户线程是一直都处于等待状态，这些线程就是可能存在问题的线程；
- 如果通过 `jstack` 可以查看到死锁状态，则可以检查产生死锁的两个线程的具体阻塞点，从而处理相应的问题。

本文主要是提出了五种常见的导致线上功能缓慢的问题，以及排查思路。当然，线上的问题出现的形式是多种多样的，也不一定局限于这几种情况，如果我们能够仔细分析这些问题出现的场景，就可以根据具体情况具体分析，从而解决相应的问题。

247人点赞 >

java进阶干货

更多精彩内容，就在简书APP




"小礼物走一走，来简书关注我"

赞赏支持

还没有人赞赏，支持一下



占小狼  如果读完觉得有收获的话，欢迎关注我的公众号：占小狼的博客 <a ...  
总资产257 共写了19.6W字 获得17,741个赞 共26,180个粉丝

关注

写下你的评论...

全部评论 14 只看作者

按时间倒序 按时间正序

写下你的评论...

评论14

赞247

赞

回复



茶还是咖啡

2楼 2019.08.21 11:19

感谢分享

赞

回复

上一页

1

2

被以下专题收入，发现更多相似内容

-  面试那些事儿
-  JVM学习
-  嘟嘟程序猿
-  面试
-  Java
-  问题排查
-  性能优化
- 展开更多

推荐阅读

面试被问怎么排查平时遇到的系统CPU飙高和频繁GC，该怎...

处理过线上问题的同学基本上都会遇到系统突然运行缓慢，CPU 100%，以及Full GC次数过多的问题。当然，这些...


 北熊行 阅读 61 评论 0 赞 0

更多精彩内容>

System Class	Shallow HC -<Number>
a208	
7aaca230	
7aaca1f0	
abbb08	
0x7aaaa0000	
MainV281 @ 0x7aaab2c50 Monitor Ctrl-Break	1

平时碰到系统CPU飙高和频繁GC


平时碰到系统CPU飙高和频繁GC，你会怎么排查？处理过线上问题的同学基本上都会遇到系统突然运行缓慢，CPU 10...

 天地征途\_觉醒 阅读 104 评论 0 赞 0

System Class	Shallow HC -<Number>
a208	
7aaca230	
7aaca1f0	
abbb08	
0x7aaaa0000	
MainV281 @ 0x7aaab2c50 Monitor Ctrl-Break	1

阿里美团java面试必问：Full GC频繁，CPU 100%等线上问...

系统运行缓慢，CPU 100%，以及Full GC次数过多问题的排查思路 1. Full GC次数过多 2. CP...

 名猿 阅读 2,656 评论 0 赞 22

System Class	Shallow HC -<Number>
a230	
7aaca1f0	
abbb08	
0x7aaaa0000	
MainV281 @ 0x7aaab2c50 Monitor Ctrl-Break	1

CPU飙高和频繁GC排查经验总结

处理过线上问题的同学基本上都会遇到系统突然运行缓慢，CPU 100%，以及Full GC次数过多的问题。当然，这些...

 步二小哥 阅读 456 评论 1 赞 0

System Class	Shallow HC -<Number>
a208	
7aaca230	
7aaca1f0	
abbb08	
0x7aaaa0000	
MainV281 @ 0x7aaab2c50 Monitor Ctrl-Break	1

写下你的评论...

评论14

赞247



