JVM 调优系列之监控工具

2017-10-23 wier 开源中国

> 摘要: 项目部署线上之后，我们该如何基于监控工具来快速定位问题....

通过上一篇的jvm垃圾回收知识，我们了解了jvm对内存分配以及垃圾回收是怎么来处理的。理论是指导实践的工具，有了理论指导，定位问题的时候，知识和经验是关键基础，数据可以为我们提供依据。

在常见的线上问题时候，我们多数会遇到以下问题：

> - 内存泄露
> - 某个进程突然cpu飙升
> - 线程死锁
> - 响应变慢...等等其他问题。

如果遇到了以上这种问题，在线下可以有各种本地工具支持查看，但到线上了，就没有这么多的本地调试工具支持，我们该如何基于监控工具来进行定位问题?

我们一般会基于数据收集来定位，而数据的收集离不开监控工具的处理，比如：运行日志、异常堆栈、GC日志、线程快照、堆快照等。经常使用恰当的分析和监控工具可以加快我们的分析数据、定位解决问题的速度。以下我们将会详细介绍。

JVM 常见监控工具&指令

jps：JVM 进程状况工具

```
jps [options] [hostid]
```

如果不指定hostid就默认为当前主机或服务器。

命令行参数选项说明如下：

```
-q 不输出类名、Jar名和传入main方法的参数

- l 输出main类或Jar的全限名

-m 输出传入main方法的参数

- v 输出传入JVM的参数
```

例如:

```
[root@localhost ~]# jps -l
13508 /opt/jetty_tx/start.jar
14691 /opt/jetty0/start.jar
25667 org.eclipse.jetty.xml.XmlConfiguration
8390 /opt/jetty1/start.jar
30240 /opt/jetty3/start.jar
24740 org.eclipse.jetty.xml.XmlConfiguration
8416 org.eclipse.jetty.xml.XmlConfiguration
14708 org.eclipse.jetty.xml.XmlConfiguration
13525 org.eclipse.jetty.xml.XmlConfiguration
8967 sun.tools.jps.Jps
30257 org.eclipse.jetty.xml.XmlConfiguration
2604 jenkins.war
24723 /opt/jetty_meta_new/start.jar
25650 /opt/jetty_meta_tw/start.jar
```

jstat：JVM 统计信息监控工具

jstat 是用于见识虚拟机各种运行状态信息的命令行工具。它可以显示本地或者远程虚拟机进程中的类装载、内存、垃圾收集、jit编译等运行数据，它是线上定位jvm性能的首选工具。

命令格式:

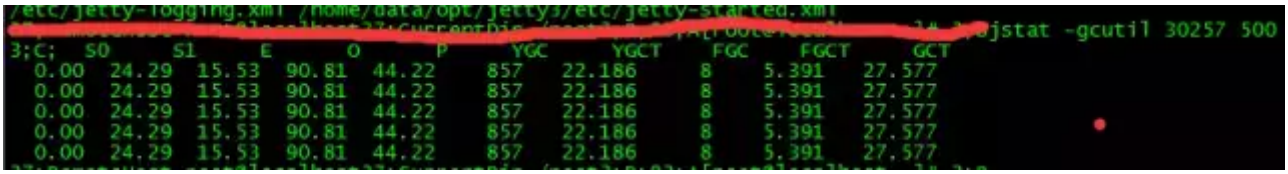
```
jstat [ generalOption | outputOptions vmid [interval[s|ms] [count]] ]

generalOption - 单个的常用的命令行选项，如-help, -options, 或 -version。

outputOptions ——个或多个输出选项，由单个的statOption选项组成，可以和-t, -h, and -J等选项配合使用。
```

参数选项：

Option	Displays	Ex
class	用于查看类加载情况的统计	jstat -class pid:显示加载class的数量，及所占空间等信息。
compiler	查看HotSpot中即时编译器编译情况的统计	jstat -compiler pid:显示VM实时编译的数量等信息。
gc	查看JVM中堆的垃圾收集情况的统计	jstat -gc pid:可以显示gc的信息，查看gc的次数，及时间。其中最后五项，分别是young gc的次数，young gc的时间，full gc的次数，full gc的时间，gc的总时间。
gccapacity	查看新生代、老生代及持久代的存储容量情况	jstat -gccapacity:可以显示，VM内存中三代（young,old,perm）对象的使用和占用大小
gccause	查看垃圾收集的统计情况（这个和-gcutil选项一样），如果有发生垃圾收集，它还会显示最后一次及当前正在发生垃圾收集的原因。	jstat -gccause:显示gc原因
gcnew	查看新生代垃圾收集的情况	jstat -gcnew pid:new对象的信息
gcnewcapacity	用于查看新生代的存储容量情况	jstat -gcnewcapacity pid:new对象的信息及其占用量
gcold	用于查看老生代及持久代发生GC的情况	jstat -gcold pid:old对象的信息
gcoldcapacity	用于查看老生代的容量	jstat -gcoldcapacity pid:old对象的信息及其占用量
gcpermcapacity	用于查看持久代的容量	jstat -gcpermcapacity pid: perm对象的信息及其占用量
gcutil	查看新生代、老生代及持代垃圾收集的情况	jstat -util pid:统计gc信息统计
printcompilation	HotSpot编译方法的统计	jstat -printcompilation pid:当前VM执行的信息

例如:

查看 gc 情况执行：jstat-gcutil 27777



jinfo：Java 配置信息

命令格式：

```
jinfo[option] pid
```

比如:获取一些当前进程的jvm运行和启动信息。

jmap：Java 内存映射工具

jmap命令用于生产堆转存快照。打印出某个java进程（使用pid）内存内的，所有‘对象’的情况（如：产生那些对象，及其数量）。

命令格式：

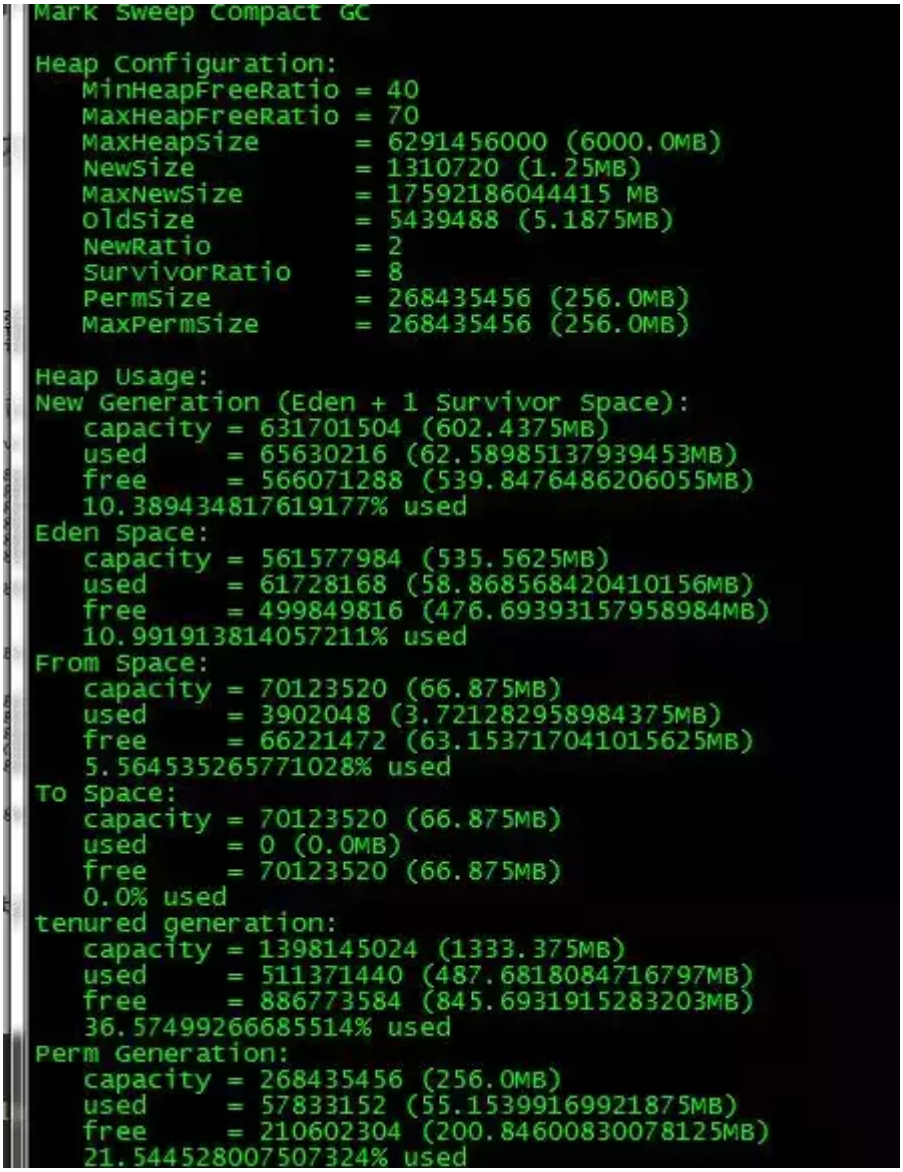
```
jmap [ option ] pid

jmap [ option ] executable core

jmap [ option ] [server-id@]remote-hostname-or-IP
```

参数选项：

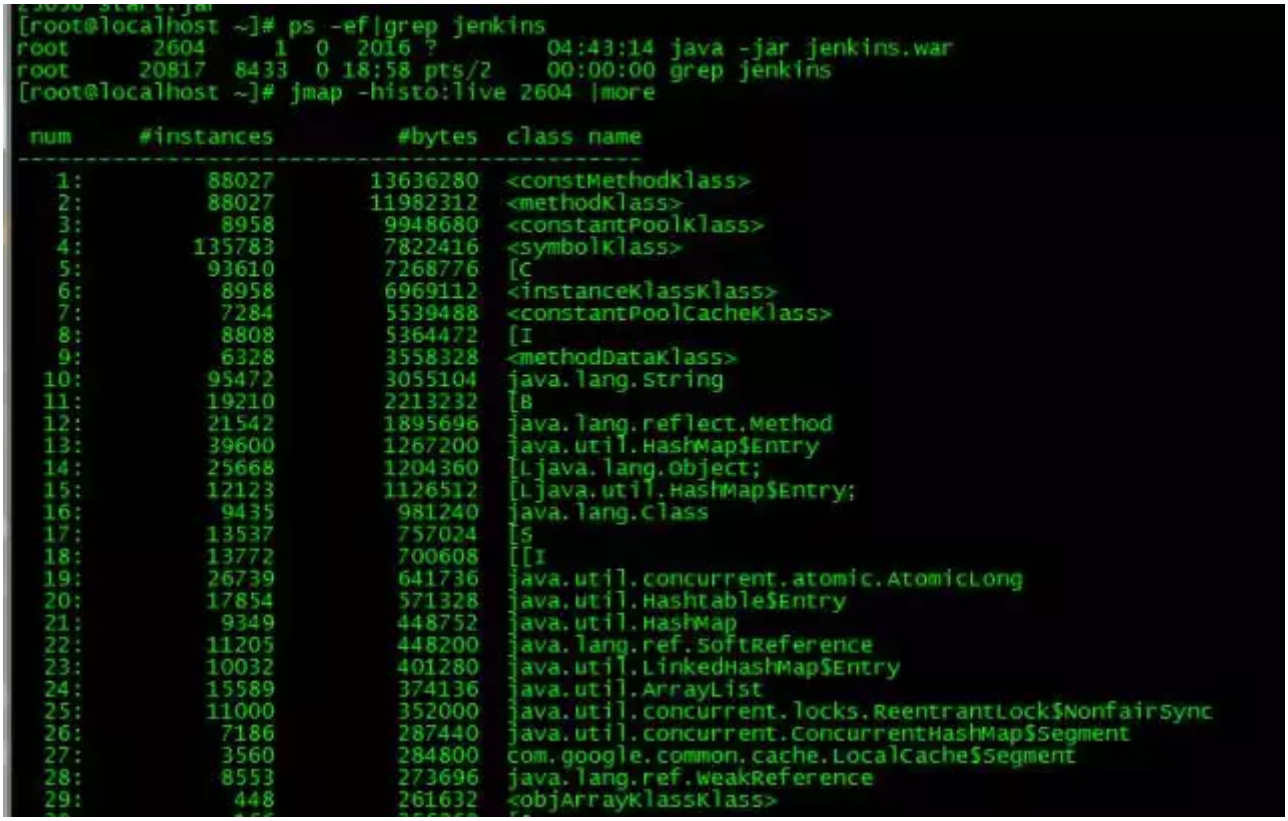
> -dump:[live,]format=b,file=<filename> 使用hprof二进制形式,输出jvm的heap内容到文件=. live子选项是可选的，假如指定live选项,那么只输出活的对象到文件.
>
> -finalizerinfo 打印正等候回收的对象的信息.
>
> -heap 打印heap的概要信息，GC使用的算法，heap的配置及wise heap的使用情况.
>
> -histo[:live] 打印每个class的实例数目,内存占用,类全名信息. VM的内部类名字开头会加上前缀”*”. 如果live子参数加上后,只统计活的对象数量.
>
> -permstat 打印classload和jvm heap长久层的信息. 包含每个classloader的名字,活泼性,地址,父classloader和加载的class数量. 另外,内部String的数量和占用内存数也会打印出来.
>
> -F 强迫.在pid没有相应的时候使用-dump或者-histo参数. 在这个模式下,live子参数无效.
>
> -h | -help 打印辅助信息

例如：

使用jmap -heap pid查看进程堆内存使用情况，包括使用的GC算法、堆配置参数和各代中堆内存使用情况：



使用jmap -histo[:live] pid查看堆内存中的对象数目、大小统计直方图。



```
[root@localhost ~]# ps -ef|grep jenkins
root      2604     1  0  2016 ?        04:43:14 java -jar jenkins.war
root     20817  8433  0 18:58 pts/2    00:00:00 grep jenkins
[root@localhost ~]# jmap -histo:live 2604 |more

 num     #instances         #bytes  class name
----------------------------------------------
   1:         88027       13636280  <constMethodKlass>
   2:         88027       11982312  <methodKlass>
   3:          8958        9948680  <constantPoolKlass>
   4:        135783        7822416  <symbolKlass>
   5:         93610        7268776  [C
   6:          8958        6969112  <instanceKlassKlass>
   7:          7284        5539488  <constantPoolCacheKlass>
   8:          8808        5364472  [I
   9:          6328        3558328  <methodDataKlass>
  10:         95472        3055104  java.lang.String
  11:         19210        2213232  [B
  12:         21542        1895696  java.lang.reflect.Method
  13:         39600        1267200  java.util.HashMap$Entry
  14:         25668        1204360  [Ljava.lang.Object;
  15:         12123        1126512  [Ljava.util.HashMap$Entry;
  16:          9435         981240  java.lang.Class
  17:         13537         757024  [S
  18:         13772         700608  [[I
  19:         26739         641736  java.util.concurrent.atomic.AtomicLong
  20:         17854         571328  java.util.Hashtable$Entry
  21:          9349         448752  java.util.HashMap
  22:         11205         448200  java.lang.ref.SoftReference
  23:         10032         401280  java.util.LinkedHashMap$Entry
  24:         15589         374136  java.util.ArrayList
  25:         11000         352000  java.util.concurrent.locks.ReentrantLock$NonfairSync
  26:          7186         287440  java.util.concurrent.ConcurrentHashMap$Segment
  27:          3560         284800  com.google.common.cache.LocalCache$Segment
  28:          8553         273696  java.lang.ref.WeakReference
  29:           448         261632  <objArrayKlassKlass>
```

jhat：JVM 堆快照分析工具

jhat 命令与jamp搭配使用，用来分析map生产的堆快存储快照。jhat内置了一个微型http/Html服务器，可以在浏览器找那个查看。

不过建议尽量不用，既然有dumpt文件，可以从生产环境拉取下来，然后通过本地可视化工具来分析，这样既减轻了线上服务器压力，有可以分析的足够详尽(比如 MAT/jprofile/visualVm)等。

jstack：Java 堆栈跟踪工具

jstack用于生成java虚拟机当前时刻的线程快照。线程快照是当前java虚拟机内每一条线程正在执行的方法堆栈的集合，生成线程快照的主要目的是定位线程出现长时间停顿的原因，如线程间死锁、死循环、请求外部资源导致的长时间等待等。

命令格式：

```
jstack [ option ] pid

jstack [ option ] executable core

jstack [ option ] [server-id@]remote-hostname-or-IP
```

参数：

```
-F当'jstack [-l] pid'没有相应的时候强制打印栈信息

-l长列表. 打印关于锁的附加信息,例如属于java.util.concurrent的ownable synchronizers列表.

-m打印java和native c/c++框架的所有栈信息.

-h | -help打印帮助信息

pid 需要被打印配置信息的java进程id,可以用jps查询.
```

后续的查找耗费最高cpu例子会用到。

可视化工具

对jvm监控的常见可视化工具，除了jdk本身提供的Jconsole和visualVm以外，还有第三方提供的jprofilter，perfino,Yourkit，Perf4j，JProbe，MAT等。这些工具都极大的丰富了我们定位以及优化jvm方式。

这些工具的使用，网上有很多教程提供，这里就不再过多介绍了。对于VisualVm来说，比较推荐使用，它除了对jvm的侵入性比较低以外，还是jdk团队自己开发的，相信以后功能会更加丰富和完善。

jprofilter对于第三方监控工具，提供的功能和可视化最为完善，目前多数ide都支持其插件，对于上线前的调试以及性能调优可以配合使用。

另外对于线上dump的heap信息，应该尽量拉去到线下用于可视化工具来分析，这样分析更详细。如果对于一些紧急的问题，必须需要通过线上监控，可以采用 VisualVm的远程功能来进行，这需要使用tool.jar下的MAT功能。

应用

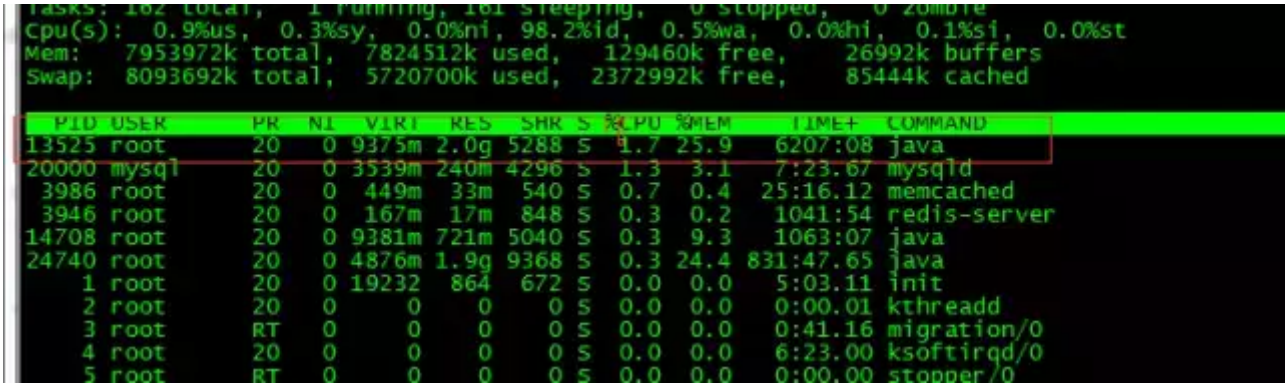
在线上有时候某个时刻，可能会出现应用某个时刻突然cpu飙升的问题。对此我们应该熟悉一些指令，快速排查对应代码。

CPU 飙升

1、找到最耗CPU的进程

指令:

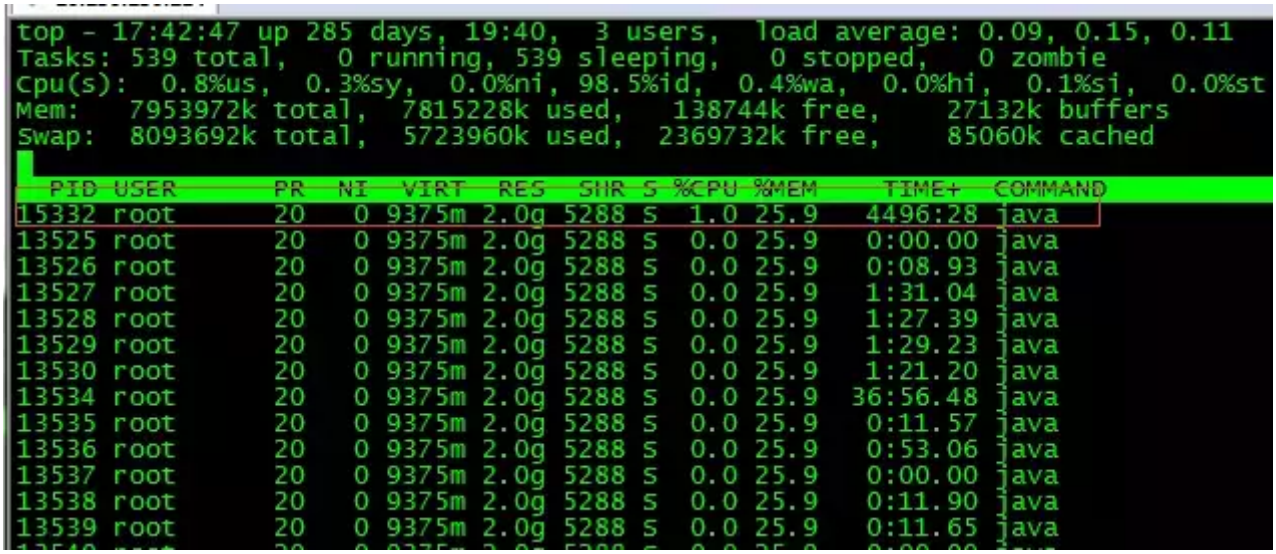
```
top
```



2、找到该进程下最耗费 CPU 的线程

指令:

```
top -Hp pid
```



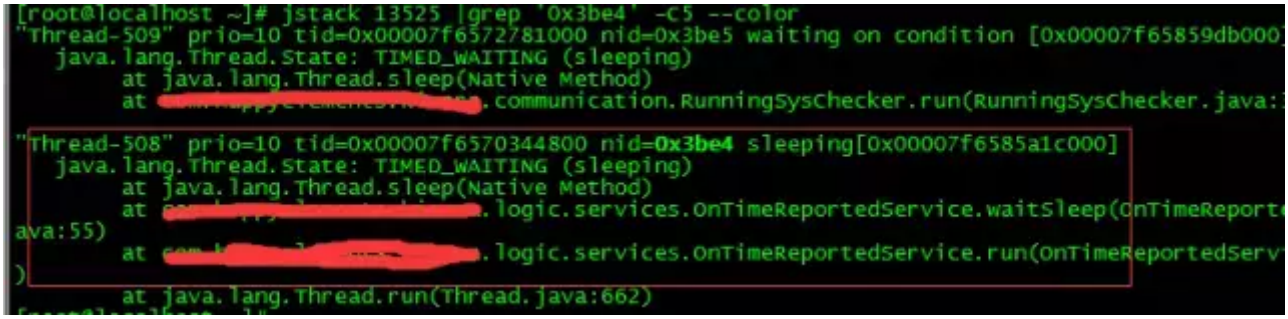
3、转换进制

```
printf “%x\n” 15332 // 转换16进制（转换后为0x3be4）
```

4、过滤指定线程，打印堆栈信息

指令

```
jstack pid |grep 'threadPid'  -C5 --color

jstack 13525 |grep '0x3be4'  -C5 --color  //  打印进程堆栈 并通过线程id，过滤得到线程堆栈信息。
```



可以看到是一个上报程序，占用过多cpu了（以上例子只为示例，本身耗费cpu并不高）

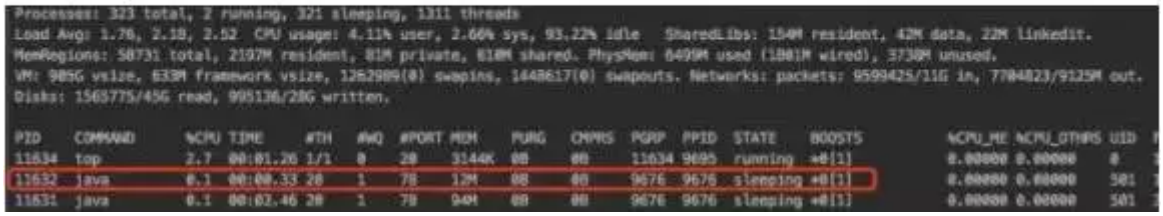
线程死锁

有时候部署场景会有线程死锁的问题发生，但又不常见。此时我们采用jstack查看下一下。比如说我们现在已经有一个线程死锁的程序，导致某些操作waiting中。

1、查找 Java 进程 ID

指令

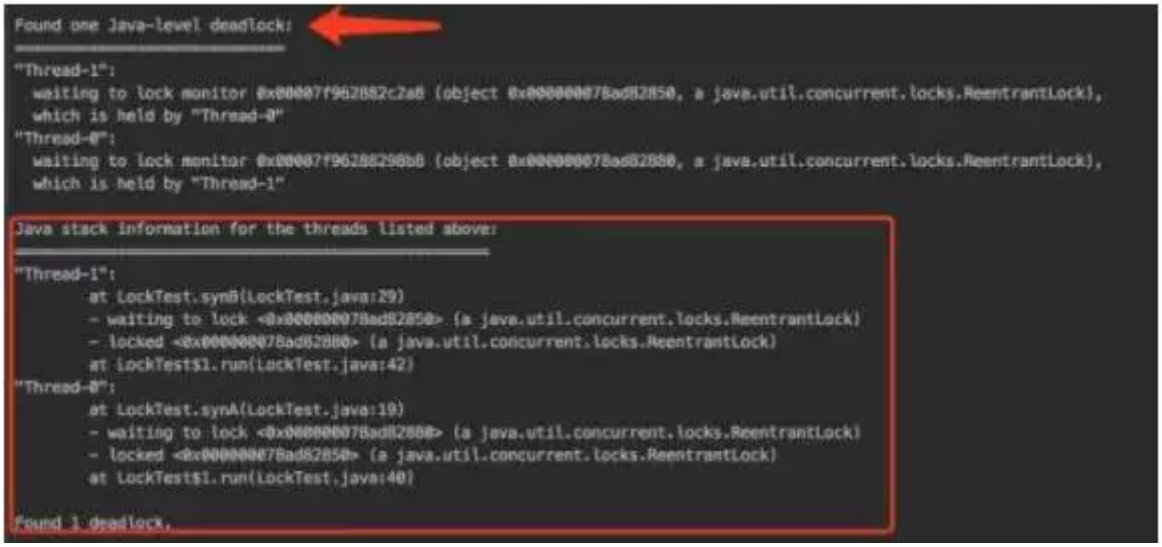
```
top 或者 jps
```



2、查看 Java 进程的线程快照信息

指令

```
jstack -l pid
```



从输出信息可以看到，有一个线程死锁发生，并且指出了那行代码出现的。如此可以快速排查问题。

OOM 内存泄露

Java堆内的OOM异常是实际应用中常见的内存溢出异常。一般我们都是先通过内存映射分析工具（比如MAT）对dump出来的堆转存快照进行分析，确认内存中对象是否出现问题。

当然了出现OOM的原因有很多，并非是堆中申请资源不足一种情况。还有可能是申请太多资源没有释放，或者是频繁频繁申请，系统资源耗尽。针对这三种情况我需要一一排查。

OOM的三种情况:

> 1.申请资源（内存）过小，不够用。
> 2.申请资源太多，没有释放。
> 3.申请资源过多，资源耗尽。比如：线程过多，线程内存过大等。

1、排查申请申请资源问题

```
指令:jmap -heap 11869
```

查看新生代，老生代堆内存的分配大小以及使用情况，看是否本身分配过小。

```
Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 2684354560 (2560.0MB)
   NewSize                  = 55574528 (53.0MB)
   MaxNewSize               = 894435328 (853.0MB)
   OldSize                  = 112197632 (107.0MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 893386752 (852.0MB)
   used     = 625416760 (596.4439010620117MB)
   free     = 267969992 (255.55609893798828MB)
   70.00515270680889% used
From Space:
   capacity = 524288 (0.5MB)
   used     = 0 (0.0MB)
   free     = 524288 (0.5MB)
   0.0% used
To Space:
   capacity = 524288 (0.5MB)
   used     = 0 (0.0MB)
   free     = 524288 (0.5MB)
   0.0% used
PS Old Generation
   capacity = 112197632 (107.0MB)
   used     = 766720 (0.731201171875MB)
   free     = 111430912 (106.268798828125MB)
   0.6833655811915887% used
```

从上述排查，发现程序申请的内存没有问题。

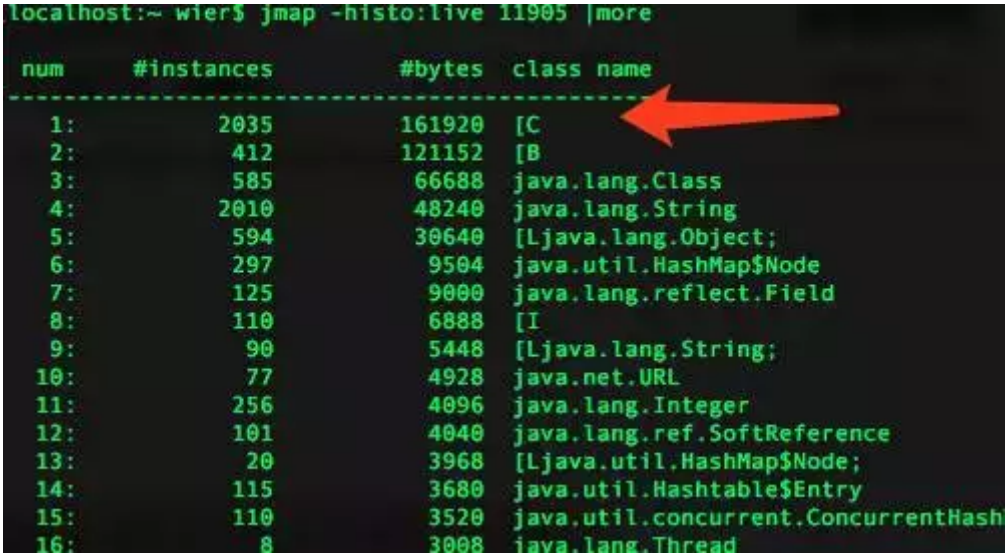
2、排查 gc

特别是fgc情况下，各个分代内存情况。

```
指令:jstat -gcutil 11938 1000 每秒输出一次gc的分代内存分配情况，以及gc时间
```

3、查找最费内存的对象

```
指令: jmap -histo:live 11869 | more
```



上述输出信息中，最大内存对象才161kb,属于正常范围。如果某个对象占用空间很大，比如超过了100Mb，应该着重分析，为何没有释放。

注意，上述指令:

```
jmap -histo:live 11869 | more

执行之后，会造成jvm强制执行一次fgc，在线上不推荐使用，可以采取dump内存快照，线下采用可视化工具进行分析，更加详尽。

jmap -dump:format=b,file=/tmp/dump.dat 11869

或者采用线上运维工具，自动化处理，方便快速定位，遗失出错时间。
```

4、确认资源是否耗尽

> pstree 查看进程线程数量
> netstat 查看网络连接数量

或者采用:

> ll /proc/${PID}/fd | wc -l // 打开的句柄数
> ll /proc/${PID}/task | wc -l （效果等同pstree -p | wc -l ）//打开的线程数

以上就是一些常见的jvm命令应用。

一种工具的应用并非是万能钥匙，包治百病，问题的解决往往是需要多种工具的结合才能更好的定位问题，无论使用何种分析工具，最重要的是熟悉每种工具的优势和劣势。这样才能取长补短，配合使用。

推荐阅读