排除Java堆栈CPU使用率过高的故障

目录

简介

本文档介绍Java Stack(Jstack)以及如何使用它来确定Cisco Policy Suite(CPS)中CPU使用率较高的根本原因。

使用Jstack排除故障

什么是Jstack?

Jstack获取正在运行的Java进程的内存转储（在CPS中，QNS是Java进程）。 Jstack包含该Java进程的所有详细信息，如线程/应用程序和每个线程的功能。

你为什么需要Jstack?

Jstack提供Jstack跟踪，以便工程师和开发人员能够了解每个线程的状态。

用于获取Java进程的Jstack跟踪的Linux命令是：

```
# jstack <process id of Java process>
```

Jstack进程在每个CPS(以前称为Quantum Policy Suite(QPS))版本中的位置是'/usr/java/jdk1.7.0_10/bin/'，其中'jdk1.7.0_10'是Java的版本，Java的版本在每个系统中都可能不同。

您还可以输入Linux命令以查找Jstack进程的确切路径：

```
# find / -iname jstack
```

此处介绍Jstack，以便您熟悉排除因Java进程而导致的CPU使用率过高问题的步骤。在CPU使用率较高的情况下，您通常会了解到Java进程利用系统的高CPU。

步骤

步骤 1：输入**top** Linux命令，以确定哪个进程消耗了虚拟机(VM)中的高CPU。

```
[root@pcrfclient01 ~]# top
top - 08:36:01 up 221 days, 20:52,  4 users,  load average: 5.86, 3.32, 2.60
Tasks: 1048 total,   1 running, 1037 sleeping,   0 stopped,  10 zombie
Cpu(s): 13.8%us,  4.2%sy,  0.0%ni, 80.0%id,  0.7%wa,  0.2%hi,  1.2%si,  0.0%st
Mem:   5975016k total,  5612888k used,   362128k free,    59776k buffers
Swap:  2097144k total,  1434016k used,   663128k free,   913832k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
14763 root      25   0 10.4g 1.3g 9.8m S  5.9 23.3   5728:23 java
21534 qns       18   0  121m  71m 1460 S  1.7  1.2   6250:45 cisco
 6667 apache    16   0  312m  20m 3984 S  1.3  0.3   0:15.51 httpd
  929 mongod    15   0  572m  97m  71m S  1.0  1.7   1744:19 mongod
14973 root      15   0 13428 2060  940 R  1.0  0.0   0:00.09 top
 4950 apache    16   0  312m  19m 3984 S  0.3  0.3   0:09.06 httpd
11839 apache    16   0  312m  20m 3984 S  0.3  0.3   0:27.41 httpd
12819 apache    16   0  312m  20m 3984 S  0.3  0.3   0:16.89 httpd
    1 root      15   0 10368  628  596 S  0.0  0.0   7:00.45 init
    2 root      RT  -5     0    0    0 S  0.0  0.0   9:12.97 migration/0
```

从此输出中取出消耗%CPU的进程。在这里，Java占5.9%，但它可以消耗更多CPU，例如40%、100%、200%、300%、400%等。

步骤 2：如果Java进程消耗的CPU较多，请输入以下命令之一，以确定哪个线程消耗的CPU:

```
# ps -C java -L -o pcpu,cpu,nice,state,cputime,pid,tid | sort
```

或者

```
# ps -C
```

例如，此显示显示显示Java进程消耗高CPU(+40%)以及Java进程线程导致高利用率。

```
<snip>

0.2 - 0 S 00:17:56 28066 28692
0.2 - 0 S 00:18:12 28111 28622
0.4 - 0 S 00:25:02 28174 28641
0.4 - 0 S 00:25:23 28111 28621
0.4 - 0 S 00:25:55 28066 28691
 43.9 - 0 R 1-20:24:41 28026 30930
 44.2 - 0 R 1-20:41:12 28026 30927
 44.4 - 0 R 1-20:57:44 28026 30916
 44.7 - 0 R 1-21:14:08 28026 30915
 %CPU CPU NI S TIME     PID  TID
```

什么是线？

假设您在系统内有一个应用程序（即单个运行进程）。但是，为了执行许多任务，您需要创建许多进程，每个进程创建许多线程。某些线程可能是读取器、写入器和不同用途，如呼叫详细记录(CDR)创建等。

在上一个示例中，Java进程ID(例如，28026)具有多个线程，包括30915、30916、30927等。

注意：线程ID(TID)采用十进制格式。

步骤 3：检查消耗高CPU的Java线程的功能。

输入以下Linux命令以获取完整的Jstack跟踪。进程ID是Java PID，例如28026，如上一输出所示。

```
# cd  /usr/java/jdk1.7.0_10/bin/

# jstack <process ID>
```

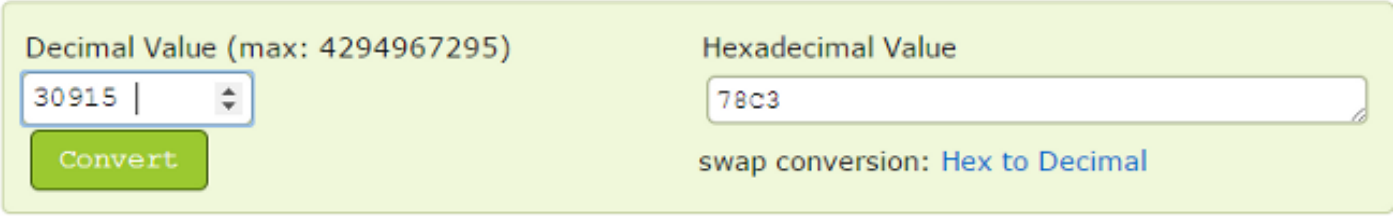
上一命令的输出如下所示：

```
2015-02-04 21:12:21
Full thread dump Java HotSpot(TM) 64-Bit Server VM (23.7-b01 mixed mode):

"Attach Listener" daemon prio=10 tid=0x000000000fb42000 nid=0xc8f waiting on
condition [0x0000000000000000]
java.lang.Thread.State: RUNNABLE

"ActiveMQ BrokerService[localhost] Task-4669" daemon prio=10 tid=0x00002aaab41fb800
nid=0xb24 waiting on condition [0x000000004c9ac000]
java.lang.Thread.State: TIMED_WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x00000000c2c07298>
(a java.util.concurrent.SynchronousQueue$TransferStack)
at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill
(SynchronousQueue.java:460)
at java.util.concurrent.SynchronousQueue$TransferStack.transfer
(SynchronousQueue.java:359)
at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:942)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)

"ActiveMQ BrokerService[localhost] Task-4668" daemon prio=10 tid=0x00002aaab4b55800
nid=0xa0f waiting on condition [0x0000000043a1d000]
java.lang.Thread.State: TIMED_WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x00000000c2c07298>
(a java.util.concurrent.SynchronousQueue$TransferStack)
at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill
(SynchronousQueue.java:460)
at java.util.concurrent.SynchronousQueue$TransferStack.transfer
(SynchronousQueue.java:359)
at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:942)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)
```

```
<snip>


"pool-84-thread-1" prio=10 tid=0x00002aaac45d8000 nid=0x78c3 runnable
[0x000000004c1a4000]
java.lang.Thread.State: RUNNABLE
at sun.nio.ch.IOUtil.drain(Native Method)
at sun.nio.ch.EPollSelectorImpl.doSelect(EPollSelectorImpl.java:92)
- locked <0x00000000c53717d0> (a java.lang.Object)
at sun.nio.ch.SelectorImpl.lockAndDoSelect(SelectorImpl.java:87)
- locked <0x00000000c53717c0> (a sun.nio.ch.Util$2)
- locked <0x00000000c53717b0> (a java.util.Collections$UnmodifiableSet)
- locked <0x00000000c5371590> (a sun.nio.ch.EPollSelectorImpl)
at sun.nio.ch.SelectorImpl.select(SelectorImpl.java:98)
at zmq.Signaler.wait_event(Signaler.java:135)
at zmq.Mailbox.recv(Mailbox.java:104)
at zmq.SocketBase.process_commands(SocketBase.java:793)
at zmq.SocketBase.send(SocketBase.java:635)
at org.zeromq.ZMQ$Socket.send(ZMQ.java:1205)
at org.zeromq.ZMQ$Socket.send(ZMQ.java:1196)
at com.broadhop.utilities.zmq.concurrent.MessageSender.run(MessageSender.java:146)
at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:471)
at java.util.concurrent.FutureTask$Sync.innerRun(FutureTask.java:334)
at java.util.concurrent.FutureTask.run(FutureTask.java:166)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1145)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)
```

现在，您需要确定Java进程的哪个线程是导致CPU使用率较高的原因。

例如，请看步骤2中提到的TID 30915。您需要将十进制格式的TID转换为十六进制格式，因为在Jstack跟踪中，您只能找到十六进制格式。使用此转换器可将十进制格式转换为十六进制格式。



如步骤3所示，Jstack跟踪的后半部分是线程，是CPU使用率较高背后负责的线程之一。在Jstack跟踪中找到78C3（十六进制格式）时，您只会发现此线程为'nid=0x78c3'。 因此，您可以找到该Java进程的所有线程，这些线程导致了高CPU消耗。

> **注意**：您目前无需关注线程的状态。作为关注点，已看到Runnable、Blocked、Timed_Waiting和Waiting等线程的某些状态。

所有以前的信息都有助于CPS和其他技术开发人员帮助您找到系统/VM中CPU使用率过高问题的根本原因。在问题出现时捕获之前提及的信息。一旦CPU使用率恢复正常，则无法确定导致CPU使用率较高的线程。

CPS日志也需要捕获。以下是路径“/var/log/broadhop”下“PCRFclient01”VM的CPS日志列表：

- **整合引擎**
- **consolidated-qns**

此外，从PCRFclient01 VM获取以下脚本和命令的输出：

- # diagnostics.sh（此脚本可能不会在CPS的较旧版本上运行，例如QNS 5.1和QNS 5.2。）
- # df-kh
- #top