排除Java堆疊高CPU使用率故障

目錄

簡介

本檔案介紹Java堆疊(Jstack)以及如何使用它來判斷思科原則套件(CPS)中CPU使用率較高的根本原因。

使用Jstack進行疑難排解

什麼是Jstack?

Jstack獲取正在運行的Java進程的記憶體轉儲（在CPS中，QNS是Java進程）。 Jstack擁有該Java進程的所有詳細資訊，例如執行緒/應用程式和每個執行緒的功能。

你為什麼需要Jstack?

Jstack提供Jstack跟蹤，以便工程師和開發人員能夠瞭解每個執行緒的狀態。

用於獲取Java進程的Jstack跟蹤的Linux命令為：

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

每個CPS(以前稱為Quantum Policy Suite(QPS))版本中Jstack進程的位置是「/usr/java/jdk1.7.0_10/bin/」，其中「jdk1.7.0_10」是Java的版本，Java的版本在每個系統中可以不同。

您還可以輸入Linux命令以尋找Jstack進程的準確路徑：

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

在此介紹Jstack，是為了讓您熟悉由於Java進程而導致的CPU使用率過高問題的故障排除步驟。在CPU使用率較高的情況下，您通常會瞭解到Java進程會利用系統中的CPU使用率。

程式

步驟1:輸入**top** Linux命令，以確定哪個進程消耗虛擬機器(VM)的高的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
```

從此輸出中，取出消耗更多%CPU的進程。在這裡，Java佔用5.9%，但它可以消耗更多的CPU，例如超過40%、100%、200%、300%、400%等。

第2步：如果Java進程佔用了高CPU，請輸入以下命令之一，以找出哪個執行緒佔用了多少：

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

或

```
# ps -C
```

例如，此顯示顯示Java進程消耗高CPU(+40%)以及導致高利用率的Java進程的執行緒。

```
<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
```

什麼是執行緒？

假設系統中有一個應用程式（即單個正在運行的進程）。但是，為了執行許多工，您需要建立許多進程，每個進程都會建立許多執行緒。有些執行緒可以是讀者、編寫者以及不同的用途，例如建立呼叫詳細記錄(CDR)等。

在上一個示例中，Java進程ID(例如28026)有多個執行緒，其中包括30915、30916、30927等等。

附註：執行緒ID(TID)採用十進位制格式。

步驟3:檢查消耗高CPU的Java執行緒的功能。

輸入以下Linux命令可取得完整的Jstack追蹤軌跡。進程ID是Java PID，例如28026如前面的輸出所示。

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

# jstack <process ID>
```

上一個命令的輸出如下：

```
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)
```

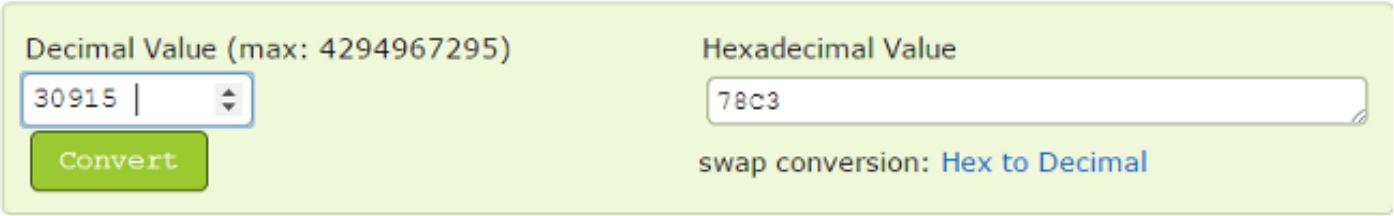
```
at java.lang.Thread.run(Thread.java:722)


<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)
```

現在，您需要確定Java進程的哪個執行緒負責高CPU利用率。

例如，請檢視步驟2中提到的TID 30915。您需要將TID轉換為十進位制格式，因為在Jstack跟蹤中，您只能找到十六進位制格式。使用此轉換器將十進位制格式轉換為十六進位制格式。



正如您在步驟3中所看到的，Jstack跟蹤的後半部分是導致CPU使用率較高的執行緒之一。當您在Jstack跟蹤中找到78C3（十六進位制格式）時，您只會發現此執行緒為「nid=0x78c3」。 因此，您可以找到該Java進程的所有執行緒，它們造成高CPU消耗。

> **附註**：您現在不需要關注執行緒的狀態。作為關注點，已看到Runnable、Blocked、Timed_Waiting和Waiting等執行緒的某些狀態。

所有以前的資訊都幫助CPS和其他技術開發人員幫助您找到系統/VM中CPU使用率較高的根本原因。在問題出現時捕獲先前提及的資訊。一旦CPU使用率恢復正常，就無法確定導致CPU使用率較高的執行緒。

也需要捕獲CPS日誌。以下是路徑「/var/log/broadhop」下「PCRFclient01」虛擬機器的CPS日誌清單：

- consolidated-engine

- consolidated-qns

此外，從PCRFclient01 VM獲取這些指令碼和命令的輸出：

- # diagnostics.sh（此指令碼可能不會在較舊版本的CPS上運行，如QNS 5.1和QNS 5.2。）
- # df -kh
- #頂部