Java スタック CPU 高使用率のトラブルシューティング

内容

概要

このドキュメントでは、Cisco Policy Suite(CPS)でCPU使用率が高くなる根本原因を特定するためにJava Stack(Jstack)を使用する方法について説明します。

Jstackによるトラブルシューティング

Jstackとは？

Jstackは、実行中のJavaプロセスのメモリダンプを取得します（CPSでは、QNSはJavaプロセスです）。 Jstackには、スレッド/アプリケーションや各スレッドの機能など、そのJavaプロセスの詳細がすべて含まれています。

なぜJstackが必要なの？

Jstackは、エンジニアや開発者が各スレッドの状態を知ることができるようにJstackトレースを提供します。

JavaプロセスのJstackトレースを取得するために使用されるLinuxコマンドは次のとおりです。

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

各CPS(以前のQuantum Policy Suite(QPS)バージョン)でのJstackプロセスの場所は、「/usr/java/jdk1.7.0_10/bin/」です。「jdk1.7.0_10」はJavaのバージョンで、Javaのバージョンはシステムごとに異なります。

Linuxコマンドを入力して、Jstackプロセスの正確なパスを見つけることもできます。

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

Javaプロセスが原因でCPU使用率が高くなる問題をトラブルシューティングする手順を理解するために、ここで説明します。CPU使用率が高い場合、一般的に、Javaプロセスがシステムから高いCPUを使用していることを学習します。

手順

ステップ 1：仮想マシン(VM)**から**どのプロセスが高いCPUを消費しているかを判断するには、top Linuxコマンドを入力します。

```
[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
```

この出力から、%CPUを消費するプロセスを取り出します。この場合、Javaは5.9 %を要しますが、40 %、100 %、200 %、300 %、400 %などのCPUを多く消費する可能性があります。

ステップ 2：Javaプロセスが高いCPUを消費する場合は、次のコマンドのいずれかを入力して、どのスレッドがどの程度のCPUを消費しているかを調べます。

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

または

```
# ps -C
```

例として、この表示は、Javaプロセスが高いCPU使用率(+40%)と、高い使用率を担当するJavaプロセスのスレッドを消費していることを示しています。

```
<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
```

スレッドとは

システム内にアプリケーション (つまり、単一の実行プロセス) があるとします。ただし、多くのタスクを実行するには、多くのプロセスを作成する必要があり、各プロセスは多数のスレッドを作成します。一部のスレッドは、リーダー、ライター、およびコール詳細レコード(CDR)の作成などのさまざまな目的である可能性があります。

前の例では、JavaプロセスID (28026など) に30915、30916、30927などの複数のスレッドがあります。

注：スレッドID(TID)は10進形式です。

ステップ 3：高CPUを消費するJavaスレッドの機能をチェックします。

完全なJstackトレースを取得するには、次のLinuxコマンドを入力します。プロセスIDはJava PIDです。たとえば、前の出力に示すように28026です。

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

# jstack <process ID>
```

前のコマンドの出力は次のようになります。

```
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)
```

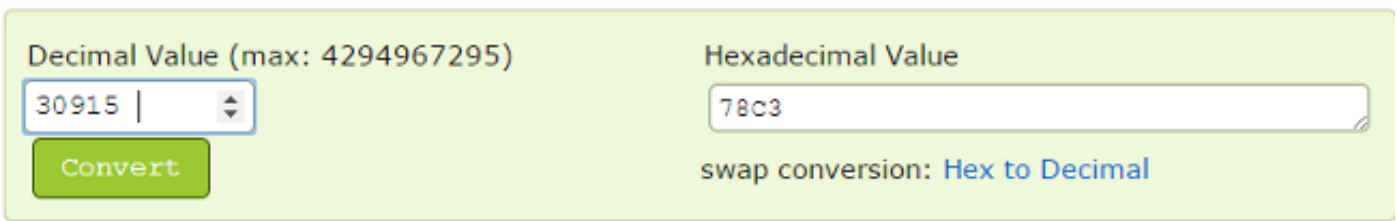
```
"ActiveMQ BrokerService[localhost] Task-4668" daemon prio=10 tid=0x00002aaab4b55800
nid=0xa0f waiting on condition [0x0000000043a1d000]
java.lang.Thread.State: TIMED_WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x00000000c2c07298>
(a java.util.concurrent.SynchronousQueue$TransferStack)
at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
at java.util.concurrent.SynchronousQueue$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)
```

次に、Javaプロセスのどのスレッドが高いCPU使用率を引き起こしているかを判断する必要があります。

例として、ステップ2で説明したTID 30915を見てください。Jstackトレースでは16進数形式しか見つからないため、10進数形式のTIDを16進数形式に変換する必要があります。10進形式を16進形式に変換するには、このコンバータを使用します。



ステップ3で確認できるように、Jstackトレースの後半は、高いCPU使用率の背後にある責任のあるスレッドの1つであるスレッドです。Jstackトレースで78C3（16進形式）が見つかると、この

スレッドは「nid=0x78c3」としてのみ見つかります。 したがって、高いCPU消費を引き起こすJavaプロセスのすべてのスレッドを見つけることができます。

> **注**：現時点では、スレッドの状態に集中する必要はありません。関心のポイントとして、Runnable、Blocked、Timed_Waiting、Waitingなどのスレッドの状態が確認されています。

上記のすべての情報は、CPSおよびその他のテクノロジー開発者が、システム/VMの高いCPU使用率の問題の根本原因を特定するのに役立ちます。問題が表示された時点で、前述の情報をキャプチャします。CPU使用率が正常に戻ると、CPU高使用率の問題の原因となったスレッドを特定できません。

CPSログもキャプチャする必要があります。パス'/var/log/broadhop'の「PCRFclient01」VMからのCPSログのリストを次に示します。

- 統合エンジン
- consolidated-qns

また、PCRFclient01 VMから次のスクリプトとコマンドの出力を取得します。

- # diagnostics.sh（このスクリプトは、QNS 5.1やQNS 5.2などの古いバージョンのCPSでは実行されない場合があります）。
- # df - kh
- # top