Risoluzione dei problemi relativi all'utilizzo elevato della CPU nello stack Java

Sommario

Introduzione

Questo documento descrive Java Stack (Jstack) e il suo utilizzo per determinare la causa principale di un elevato utilizzo della CPU in Cisco Policy Suite (CPS).

Risoluzione dei problemi con Jstack

Cos'è Jstack?

Jstack acquisisce un dump della memoria di un processo Java in esecuzione (in CPS, QNS è un processo Java). Jstack dispone di tutti i dettagli del processo Java, quali thread/applicazioni e funzionalità di ogni thread.

Perché hai bisogno di Jstack?

Jstack fornisce la traccia Jstack in modo che ingegneri e sviluppatori possano conoscere lo stato di ogni thread.

Il comando Linux utilizzato per ottenere la traccia Jstack del processo Java è:

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

La posizione del processo Jstack in ogni versione di CPS (precedentemente nota come Quantum Policy Suite (QPS)) è '/usr/java/jdk1.7.0_10/bin/' dove 'jdk1.7.0_10' è la versione di Java e la versione di Java può differire in ogni sistema.

Èpossibile anche immettere un comando Linux per trovare il percorso esatto del processo Jstack:

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

In questa sezione viene illustrato Jstack per acquisire familiarità con le procedure per la risoluzione dei problemi di utilizzo elevato della CPU a causa del processo Java. Nei casi di utilizzo elevato della CPU, in genere si apprende che un processo Java utilizza la CPU elevata del sistema.

Procedura

Passaggio 1: Immettere il comando **top** Linux per determinare quale processo utilizza CPU elevata della macchina virtuale (VM).

```
[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
```

Da questo output, eliminare i processi che utilizzano più %CPU. In questo caso, Java richiede il 5,9% ma può consumare più CPU, ad esempio oltre il 40%, il 100%, il 200%, il 300%, il 400% e così via.

Passaggio 2: Se un processo Java utilizza una CPU elevata, immettere uno di questi comandi per individuare il thread che ne utilizza la quantità:

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

O

```
# ps -C
```

Ad esempio, questa visualizzazione mostra che il processo Java utilizza una CPU elevata (+40%) e i thread del processo Java responsabili dell'elevato utilizzo.

```
<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
```

Cos'è un filo?

Si supponga di disporre di un'applicazione, ovvero di un singolo processo in esecuzione, all'interno del sistema. Tuttavia, per eseguire molte attività è necessario creare molti processi e ogni processo crea molti thread. Alcuni dei thread possono essere di lettura, scrittura e diversi scopi, ad esempio la creazione di Call Detail Record (CDR) e così via.

Nell'esempio precedente, l'ID di processo Java (ad esempio, 28026) dispone di più thread che includono 30915, 30916, 30927 e molti altri.

> **Nota:** L'ID thread (TID) è in formato decimale.

Passaggio 3: Verificare la funzionalità dei thread Java che utilizzano la CPU elevata.

Immettere questi comandi Linux per ottenere la traccia Jstack completa. ID processo è il PID Java, ad esempio 28026 come mostrato nell'output precedente.

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

# jstack <process ID>
```

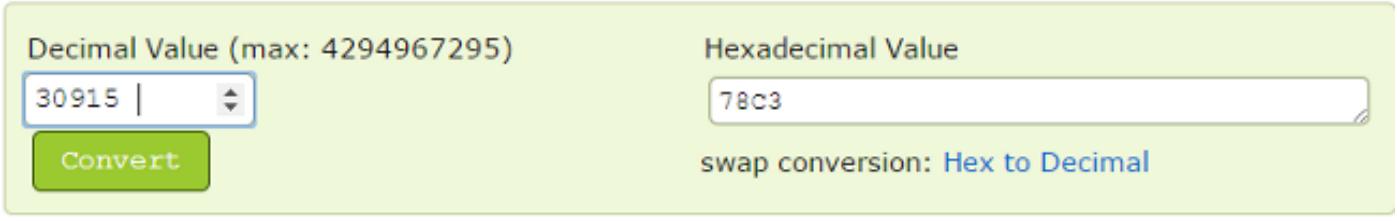
L'output del comando precedente è simile al seguente:

```
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)
```

Ora è necessario determinare quale thread del processo Java è responsabile dell'elevato utilizzo della CPU.

Ad esempio, osservare il TID 30915 come indicato al punto 2. È necessario convertire il TID in formato decimale in esadecimale perché in Jstack trace è possibile trovare solo la forma esadecimale. Utilizzare questo convertitore per convertire il formato decimale nel formato esadecimale.



Come si può vedere al passo 3, la seconda metà della traccia Jstack è il thread che è uno dei thread responsabili dietro l'elevato utilizzo della CPU. Quando si trova il formato 78C3 (formato esadecimale) nell'analisi Jstack, il thread sarà solo 'nid=0x78c3'. Pertanto, è possibile trovare tutti i thread di tale processo Java responsabili di un elevato consumo della CPU.

> **Nota:** Per il momento non è necessario concentrarsi sullo stato del thread. Come punto di interesse, sono stati rilevati alcuni stati dei thread, ad esempio Runnable, Blocked, Timed_Waiting e Waiting.

Tutte le informazioni precedenti consentono a CPS e ad altri sviluppatori di tecnologie di individuare la causa principale del problema di utilizzo elevato della CPU nel sistema/macchina virtuale. Acquisire le informazioni precedentemente menzionate quando viene visualizzato il problema. Quando l'utilizzo della CPU torna alla normalità, non è possibile determinare i thread che hanno causato il problema della CPU.

Ènecessario acquisire anche i log CPS. Di seguito è riportato l'elenco dei log CPS dalla VM 'PCRFclient01' nel percorso '/var/log/broadhop':

- motore consolidato
- consolidate-qns

Inoltre, ottenere l'output di questi script e comandi dalla VM PCRFclient01:

- # diagnostics.sh (Questo script potrebbe non essere eseguito su versioni precedenti di CPS, ad esempio QNS 5.1 e QNS 5.2)
- # df -kh
- N. inizio