Dépannage de l'utilisation élevée du CPU dans la pile Java

Contenu

Introduction

Ce document décrit la pile Java (Jstack) et comment l'utiliser afin de déterminer la cause première d'une utilisation élevée du CPU dans Cisco Policy Suite (CPS).

Dépannage avec Jstack

Qu'est-ce que Jstack ?

Jstack prend un vidage mémoire d'un processus Java en cours d'exécution (dans CPS, QNS est un processus Java). Jstack possède tous les détails de ce processus Java, tels que les threads/applications et la fonctionnalité de chaque thread.

Pourquoi as-tu besoin de Jstack ?

Jstack fournit la trace Jstack afin que les ingénieurs et les développeurs puissent connaître l'état de chaque thread.

La commande Linux utilisée pour obtenir la trace Jstack du processus Java est la suivante :

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

L'emplacement du processus Jstack dans chaque version de CPS (précédemment appelée Quantum Policy Suite (QPS)) est '/usr/java/jdk1.7.0_10/bin/' où 'jdk1.7.0_10' est la version de Java et la version de Java peut différer dans chaque système.

Vous pouvez également entrer une commande Linux afin de trouver le chemin exact du processus Jstack :

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

Jstack est expliqué ici afin de vous familiariser avec les étapes de dépannage des problèmes d'utilisation élevée du CPU à cause du processus Java. Dans les cas d'utilisation élevée du CPU, vous apprenez généralement qu'un processus Java utilise le CPU élevé du système.

Procédure

Étape 1 : Entrez la commande **top** Linux afin de déterminer quel processus consomme un CPU élevé à partir de la machine virtuelle (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
```

Àpartir de cette sortie, retirez les processus qui consomment plus de %CPU. Dans ce cas, Java prend 5,9 %, mais il peut consommer plus de CPU comme plus de 40 %, 100 %, 200 %, 300 %, 400 %, etc.

Étape 2 : Si un processus Java consomme un CPU élevé, entrez l'une de ces commandes afin de déterminer quel thread consomme autant :

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

OU

```
# ps -C
```

Par exemple, cet affichage montre que le processus Java consomme un CPU élevé (+40%) ainsi que les threads du processus Java responsables de la haute utilisation.

```
<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
```

Qu'est-ce qu'un fil ?

Supposons que vous ayez une application (c'est-à-dire un seul processus en cours d'exécution) dans le système. Cependant, pour exécuter de nombreuses tâches, vous devez créer de nombreux processus et chaque processus crée de nombreux threads. Certains threads peuvent être des lecteurs, des rédacteurs et des objectifs différents tels que la création d'enregistrements détaillés des appels (CDR), etc.

Dans l'exemple précédent, l'ID de processus Java (par exemple, 28026) a plusieurs threads qui incluent 30915, 30916, 30927 et bien d'autres.

> **Note**: L'ID de thread (TID) est au format décimal.

Étape 3 : Vérifiez la fonctionnalité des threads Java qui consomment le CPU élevé.

Entrez ces commandes Linux afin d'obtenir la trace complète de Jstack. ID de processus est le PID Java, par exemple 28026, comme indiqué dans la sortie précédente.

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

# jstack <process ID>
```

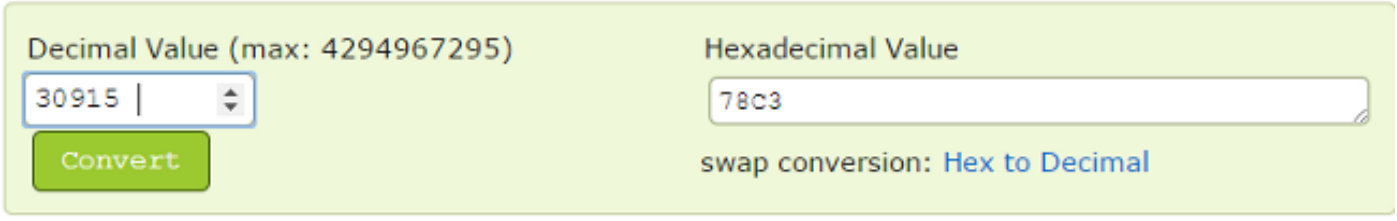
La sortie de la commande précédente ressemble à :

```
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)
```

Vous devez maintenant déterminer quel thread du processus Java est responsable de l'utilisation élevée du CPU.

Àtitre d'exemple, examinez le TID 30915 comme indiqué à l'étape 2. Vous devez convertir le TID au format décimal en format hexadécimal, car dans la trace Jstack, vous ne pouvez trouver que la forme hexadécimale. Utilisez ce convertisseur afin de convertir le format décimal au format hexadécimal.



Comme vous pouvez le voir à l'étape 3, la deuxième moitié de la trace Jstack est le thread qui est l'un des threads responsables derrière l'utilisation élevée du CPU. Lorsque vous trouvez le 78C3 (format hexadécimal) dans la trace Jstack, vous ne trouverez ce thread que 'nid=0x78c3'. Par conséquent, vous pouvez trouver tous les threads de ce processus Java qui sont responsables d'une consommation élevée de CPU.

> **Note**: Vous n'avez pas besoin de vous concentrer sur l'état du thread pour l'instant. Par souci d'intérêt, certains états des threads tels que Runnable, Blocked, Timed_Waiting et Waiting ont été vus.

Toutes les informations précédentes aident CPS et d'autres développeurs technologiques à vous aider à atteindre la cause première du problème d'utilisation élevée du CPU dans le système/la machine virtuelle. Capturez les informations précédemment mentionnées au moment où le problème apparaît. Une fois que l'utilisation du CPU est revenue à la normale, les threads qui ont causé le problème de CPU élevé ne peuvent pas être déterminés.

Les journaux CPS doivent également être capturés. Voici la liste des journaux CPS de la machine virtuelle 'PCRFclient01' sous le chemin '/var/log/broadhop' :

- moteur consolidé
- Consolidated-qns

Obtenez également la sortie de ces scripts et commandes à partir de la machine virtuelle PCRFclient01 :

- # diagnostics.sh (Ce script peut ne pas s'exécuter sur des versions plus anciennes de CPS, telles que QNS 5.1 et QNS 5.2.)
- # df -kh
- # haut