Solucionar problemas de alta utilização da CPU da pilha Java

Contents

Introduction

Este documento descreve o Java Stack (Jstack) e como usá-lo para determinar a causa raiz da alta utilização da CPU no Cisco Policy Suite (CPS).

Solucionar problemas com Jstack

O que é o Jstack?

O Jstack usa um dump de memória de um processo Java em execução (no CPS, o QNS é um processo Java). O Jstack tem todos os detalhes desse processo Java, como threads/aplicativos e a funcionalidade de cada thread.

Por que você precisa do Jstack?

O Jstack fornece o rastreamento do Jstack para que engenheiros e desenvolvedores possam conhecer o estado de cada thread.

O comando Linux usado para obter o rastreamento Jstack do processo Java é:

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

O local do processo Jstack em cada versão do CPS (anteriormente conhecido como Quantum Policy Suite (QPS)) é '/usr/java/jdk1.7.0_10/bin/', onde 'jdk1.7.0_10' é a versão do Java e a versão do Java pode ser diferente em cada sistema.

Você também pode inserir um comando Linux para encontrar o caminho exato do processo Jstack:

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

O Jstack é explicado aqui para que você se familiarize com as etapas de solução de problemas de alta utilização da CPU devido ao processo Java. Em casos de alta utilização da CPU, você geralmente aprende que um processo Java utiliza a CPU alta do sistema.

Procedimento

Passo 1: Insira o comando Linux **superior** para determinar qual processo consome CPU alta da máquina virtual (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
```

A partir desta saída, retire os processos que consomem mais %CPU. Aqui, o Java consome 5,9 %, mas pode consumir mais CPU, como mais de 40%, 100%, 200%, 300%, 400%, etc.

Passo 2: Se um processo Java consumir CPU alta, insira um destes comandos para descobrir qual thread consome quanto:

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

OU

```
# ps -C
```

Como exemplo, esta exibição mostra que o processo Java consome alta CPU (+40%), bem como os segmentos do processo Java responsáveis pela alta utilização.

```
<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
```

O que é um fio?

Suponha que você tenha um aplicativo (ou seja, um único processo em execução) dentro do sistema. No entanto, para executar muitas tarefas, você precisa que muitos processos sejam criados e cada processo cria muitos processos. Alguns dos segmentos podem ser leitores, gravadores e finalidades diferentes, como criação do registro de detalhes de chamadas (CDR - Call Detail Record) e assim por diante.

No exemplo anterior, a ID do processo Java (por exemplo, 28026) tem vários segmentos que incluem 30915, 30916, 30927 e muitos outros.

> **Note**: A ID do segmento (TID) está em formato decimal.

Passo 3: Verifique a funcionalidade dos processos Java que consomem a CPU alta.

Insira estes comandos do Linux para obter o rastreamento completo do Jstack. O ID do processo é o PID do Java, por exemplo, 28026, como mostrado na saída anterior.

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

# jstack <process ID>
```

A saída do comando anterior é semelhante a:

```
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)
```

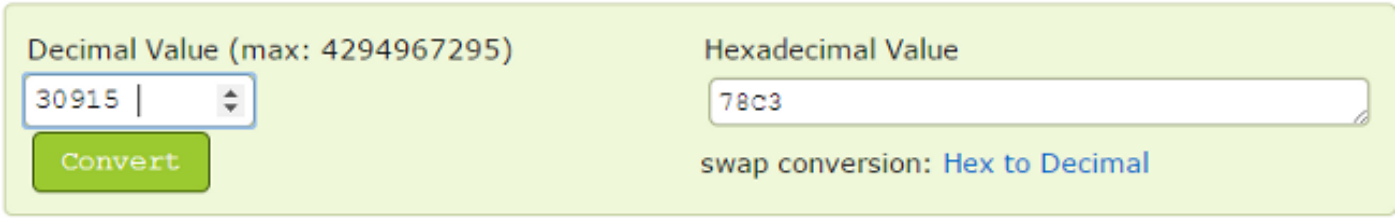
```
"ActiveMQ BrokerService[localhost] Task-4668" daemon prio=10 tid=0x00002aaab4b55800
nid=0xa0f waiting on condition [0x0000000043a1d000]
java.lang.Thread.State: TIMED_WAITING (parking)
at sun.misc.Unsafe.park(Native Method)
- parking to wait for <0x00000000c2c07298>
(a java.util.concurrent.SynchronousQueue$TransferStack)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

Agora você precisa determinar qual thread do processo Java é responsável pela alta utilização da CPU.

Por exemplo, veja o TID 30915 conforme mencionado na Etapa 2. Você precisa converter o TID em formato decimal em formato hexadecimal porque no rastreamento Jstack, você só pode encontrar a forma hexadecimal. Use este conversor para converter o formato decimal no formato hexadecimal.



Como você pode ver na Etapa 3, a segunda metade do rastreamento do Jstack é o segmento que é um dos segmentos responsáveis por trás da alta utilização da CPU. Quando encontrar o 78C3 (formato hexadecimal) no rastreamento Jstack, você encontrará esse thread apenas como 'nid=0x78c3'. Assim, você pode encontrar todos os processos desse processo Java que são responsáveis pelo alto consumo de CPU.

> **Note**: Você não precisa se concentrar no estado do thread por enquanto. Como ponto de interesse, alguns estados dos segmentos como Runnable, Blocked, Timed_Waiting e Waiting foram vistos.

Todas as informações anteriores ajudam o CPS e outros desenvolvedores de tecnologia a ajudarem você a descobrir a causa principal do problema de alta utilização da CPU no sistema/VM. Capture as informações mencionadas anteriormente no momento em que o problema for exibido. Quando a utilização da CPU voltar ao normal, os segmentos que causaram o problema de alta utilização da CPU não poderão ser determinados.

Os registros CPS também precisam ser capturados. Esta é a lista de registros CPS da VM 'PCRFclient01' no caminho '/var/log/broadcast':

- mecanismo consolidado
- consolidado-qns

Além disso, obtenha a saída desses scripts e comandos do PCRFclient01 VM:

- # diagnostics.sh (Este script pode não ser executado em versões mais antigas do CPS, como QNS 5.1 e QNS 5.2.)
- # df -kh
- # superior