Pesquise defeitos a utilização elevada da CPU da pilha das Javas

Índice

Introdução

Este documento descreve a pilha das Javas (Jstack) e como usá-la a fim determinar a causa de raiz da utilização elevada da CPU na série da política de Cisco (CP).

Pesquise defeitos com Jstack

Que é Jstack?

Jstack toma um dump de memória de um processo running das Javas (nos CP, QNS é um processo das Javas). Jstack tem todos os detalhes daquele processo das Javas, tal como linhas/aplicativos e a funcionalidade de cada linha.

Por que você precisa Jstack?

Jstack fornece o traço de Jstack de modo que os coordenadores e os colaboradores possam conhecer o estado de cada linha.

O comando linux usado para obter o traço de Jstack do processo das Javas é:

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

O lugar do processo de Jstack no cada CP (conhecidos previamente como a série da política do quantum (QPS)) a versão é '/usr/java/jdk1.7.0_10/bin/ onde 'jdk1.7.0_10 é a versão de java e a versão de java pode diferir em cada sistema.

Você pode igualmente inscrever um comando linux a fim encontrar o caminho exato do processo de Jstack:

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

Jstack é explicado aqui a fim consegui-lo familiar com as etapas pesquisar defeitos edições da utilização elevada da CPU devido ao processo das Javas. Na utilização elevada da CPU encaixota-o aprendem geralmente que um processo das Javas utiliza a alta utilização da CPU do sistema.

Procedimento

Passo 1: Inscreva o comando linux **superior** a fim determinar que processo consome a alta utilização da CPU 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
```

Desta saída, remova os processos que consomem mais %CPU. Aqui, a Java toma 5.9% mas pode consumir mais CPU como mais de 40%, 100%, 200%, 300%, 400%, e assim por diante.

Passo 2: Se um processo das Javas consome a alta utilização da CPU, incorpore um destes comandos a fim encontrar que a linha consome quanto:

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

OU

```
# ps -C <process ID> -L -o pcpu,cpu,nice,state,cputime,pid,tid | sort
```

Como um exemplo, este indicador mostra que o processo das Javas consome a alta utilização da CPU (+40%) assim como as linhas das Javas processam o responsável para a utilização elevada.

```
<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
```

Que é uma linha?

Supõe que você tem um aplicativo (isto é, um único processo do corredor) dentro do sistema. Contudo, a fim executar muitas tarefas você exige muitos processos ser criado e cada processo cria muitas linhas. Algumas das linhas poderiam ser leitor, escritor, e finalidades diferentes tais como a criação do registro dos destalhes da chamada (CDR) e assim por diante.

No exemplo anterior, o processo ID das Javas (por exemplo, 28026) tem as linhas múltiplas que incluem 30915, 30916, 30927 e muito mais.

> Nota: A linha ID (TID) está no formato decimal.

Passo 3: Verifique a funcionalidade das linhas das Javas que consomem a alta utilização da CPU.

Inscreva estes comandos linux a fim obter o traço completo de Jstack. O processo ID é a Java PID, por exemplo 28026 segundo as indicações da saída precedente.

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

# jstack <process ID>
```

A saída do comando precedente olha como:

```
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)
```

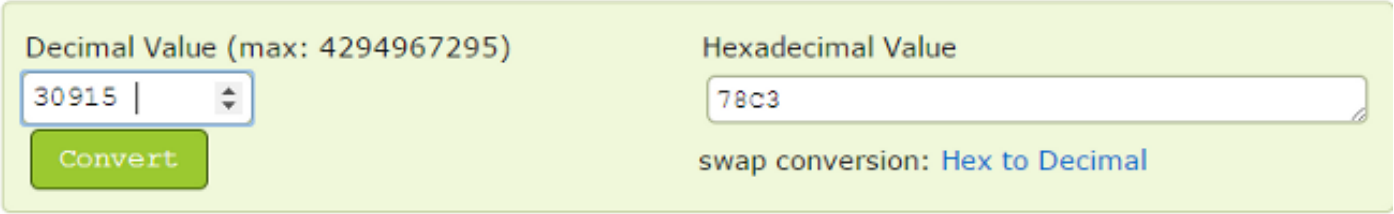
```
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
at java.lang.Thread.run(Thread.java:722)


<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 de determinar que linha do processo das Javas é responsável para a utilização elevada da CPU.

Como um exemplo, olhar em TID 30915 como mencionado em etapa 2. Você precisa de converter o TID no decimal ao formato hexadecimal porque no traço de Jstack, você pode somente encontrar o formulário hexadecimal. Use este conversor a fim converter o formato decimal no formato hexadecimal.



Como você pode ver em etapa 3, a segunda metade do traço de Jstack é a linha que é uma das linhas responsáveis atrás da utilização elevada da CPU. Quando você encontra o 78C3 (formato hexadecimal) no traço de Jstack, a seguir você encontrará somente esta linha como 'nid=0x78c3. Daqui, você pode encontrar todas as linhas daquele processo das Javas que são responsáveis para o consumo da alta utilização da CPU.

> Nota: Você não precisa de centrar-se por agora sobre o estado da linha. Como um ponto do interesse, alguns estados das linhas como praticável, obstruídos, Timed_Waiting, e espera foram considerados.

Todas as ajudas CP da informação anterior e outros colaboradores da tecnologia lhe ajudam a obter à causa de raiz da edição da utilização elevada da CPU no system/VM. Capture a informação previamente mencionada então a edição aparece. Uma vez que a utilização CPU é de volta ao normal então as linhas que causaram a edição da alta utilização da CPU não podem ser determinadas.

Os logs CP precisam de ser capturados também. Está aqui a lista de logs CP do 'PCRFclient01 VM sob o trajeto “/var/log/broadhop”:

- consolidar-motor
- consolidado-qns

Também, obtenha a saída destes scripts e comandos do PCRFclient01 VM:

- # diagnostics.sh (este script não pôde ser executado em umas versões mais velhas dos CP, tais como QNS 5.1 e QNS 5.2.)
- # df - KH
- # parte superior