Java 스택 높은 CPU 사용률 문제 해결

목차

소개

이 문서에서는 CPS(Cisco Policy Suite)에서 CPU 사용률이 높은 근본 원인을 확인하기 위해 Jstack(Java Stack) 및 이 JSTACK을 사용하는 방법에 대해 설명합니다.

Jstack 문제 해결

Jstack이란?

Jstack은 실행 중인 Java 프로세스의 메모리 덤프를 사용합니다(CPS에서 QNS는 Java 프로세스). Jstack에는 스레드/애플리케이션, 각 스레드의 기능 등 해당 Java 프로세스의 모든 세부 정보가 있습니다.

Jstack이 필요한 이유는 무엇입니까?

Jstack은 엔지니어와 개발자가 각 스레드의 상태를 알 수 있도록 Jstack 추적을 제공합니다.

Java 프로세스의 Jstack 추적을 얻는 데 사용되는 Linux 명령은 다음과 같습니다.

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

모든 CPS(이전의 Quantum Policy Suite(QPS)) 버전에서 Jstack 프로세스의 위치는 '/usr/java/jdk1.7.0_10/bin/'입니다. 여기서 'jdk1.7.0_10'은 Java 버전이고 Java 버전은 모든 시스템에서 다를 수 있습니다.

Jstack 프로세스의 정확한 경로를 찾기 위해 Linux 명령을 입력할 수도 있습니다.

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

Jstack은 Java 프로세스 때문에 CPU 사용률이 높은 문제를 해결하는 단계를 익히기 위해 여기에서

설명합니다.CPU 사용률이 높은 경우에는 일반적으로 Java 프로세스가 시스템의 높은 CPU를 사용한다는 사실을 알게 됩니다.

절차

1단계:가상 머신(VM)에서 높은 CPU를 사용하는 프로세스를 확인하려면 **top** Linux 명령을 입력합니다.

```
[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
```

이 출력에서 %CPU를 더 많이 사용하는 프로세스를 제거합니다.여기서 Java는 5.9%를 사용하지만 40%, 100%, 200%, 300%, 400% 등 더 많은 CPU를 사용할 수 있습니다.

2단계: Java 프로세스에서 높은 CPU를 사용하는 경우 다음 명령 중 하나를 입력하여 어느 스레드가 어느 정도의 양을 소비하는지 확인합니다.

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

또는

```
# ps -C
```

예를 들어, 이 화면에서는 Java 프로세스가 높은 CPU(+40%)를 소비할 뿐만 아니라 높은 활용률을 책임지는 Java 프로세스의 스레드를 사용한다는 것을 보여줍니다.

```
<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
```

실이란 무엇입니까?

시스템 내에 응용 프로그램(즉, 단일 실행 프로세스)이 있다고 가정합니다.그러나 많은 작업을 수행하려면 많은 프로세스를 만들어야 하고 각 프로세스에서는 많은 스레드를 만듭니다.스레드 중 일부는 읽기, 쓰기, CDR(Call Detail Record) 만들기 등의 다른 용도가 될 수 있습니다.

이전 예에서 Java 프로세스 ID(예: 28026)에는 30915, 30916, 30927 등 여러 스레드가 포함되어 있습니다.

참고:스레드 ID(TID)는 10진수 형식입니다.

3단계:높은 CPU를 사용하는 Java 스레드의 기능을 확인합니다.

전체 Jstack 추적을 얻으려면 다음 Linux 명령을 입력합니다.프로세스 ID는 Java PID입니다(예: 이전 출력에 표시된 28026).

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

# jstack <process ID>
```

이전 명령의 출력은 다음과 같습니다.

```
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)
```

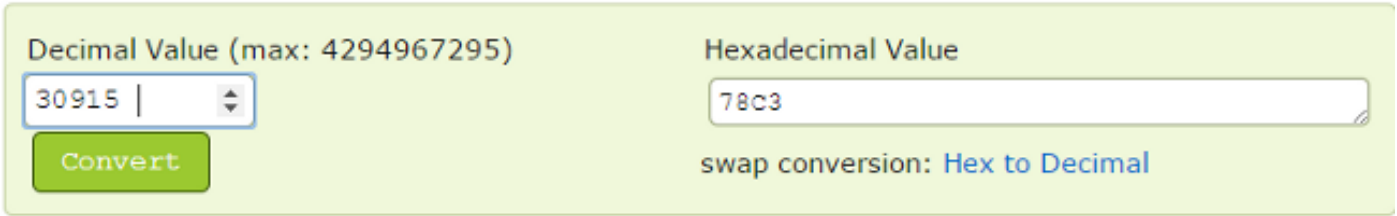
```
at java.util.concurrent.locks.LockSupport.parkNanos(LockSupport.java:226)
at java.util.concurrent.SynchronousQueue$TransferStack.awaitFulfill
(SynchronousQueue.java:460)
at java.util.concurrent.SynchronousQueue$TransferStack.transfer
(SynchronousQueue.java:359)
at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:942)
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)
```

이제 CPU 사용률이 높은 Java 프로세스의 스레드를 결정해야 합니다.

예를 들어, 2단계에서 설명한 것처럼 TID 30915를 살펴봅니다. TID를 16진수 형식으로 변환해야 합니다. Jstack 추적에서는 16진수 형식만 찾을 수 있기 때문입니다.10진수 형식을 16진수 형식으로 변환하려면 이 변환기를 사용합니다.



3단계에서 볼 수 있듯이 Jstack 추적의 후반부는 높은 CPU 사용률 뒤에 있는 책임 스레드 중 하나인 스레드입니다.Jstack 추적에서 78C3(16진수 형식)을 찾으면 이 스레드는 'nid=0x78c3'로만 표시됩니다. 따라서 CPU 사용량이 많은 Java 프로세스의 모든 스레드를 찾을 수 있습니다.

> **참고:**지금은 스레드 상태에 집중할 필요가 없습니다.관심 사항으로 Runnable, Blocked, Timed_Waiting 및 Waiting과 같은 스레드의 일부 상태가 표시되었습니다.

이전 정보는 CPS 및 기타 기술 개발자가 시스템/VM에서 CPU 사용률 문제가 발생하는 근본 원인을 파악하는 데 도움이 됩니다.문제가 나타날 때 앞서 언급한 정보를 캡처합니다.CPU 사용률이 정상으로 돌아가면 높은 CPU 문제를 일으킨 스레드를 확인할 수 없습니다.

CPS 로그도 캡처해야 합니다.다음은 '/var/log/broadhop' 경로 아래의 'PCRFclient01' VM의 CPS 로그 목록입니다.

- **통합 엔진**
- **통합 qns**

또한 PCRFclient01 VM에서 다음 스크립트 및 명령의 출력을 가져옵니다.

- **# diagnostics.sh**(이 스크립트는 QNS 5.1 및 QNS 5.2와 같은 이전 버전의 CPS에서 실행되지 않을 수 있습니다.)
- **df -kh**
- **상위 항목**