Java устранения неполадок складывает высокую загрузку ЦП

Содержание

Введение

Этот документ описывает Стек Java (Jstack) и как использовать его для определения основной причины высокой загрузки ЦП в Комплекте политики Cisco (CPS).

Устранение неполадок с Jstack

Что такое Jstack?

Jstack берет дамп памяти рабочего процесса Java (в CPS, QNS является процессом Java). Jstack имеет все подробности того процесса Java, такие как потоки/приложения и функциональность каждого потока.

Почему вам нужен Jstack?

Jstack предоставляет трассировку Jstack так, чтобы инженеры и разработчики могли узнать состояние каждого потока.

Команда Linux, используемая для получения трассировки Jstack процесса Java:

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

Местоположение процесса Jstack в каждом CPS (ранее известный как Комплект политики Quantum (QPS)) версия является '/usr/java/jdk1.7.0_10/bin /' где 'jdk1.7.0_10' является версией Java, и версия Java может отличаться по каждой системе.

Можно также ввести команду Linux для обнаружения точного пути процесса Jstack:

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

Jstack объясняют здесь, чтобы заставить вас знакомый с шагами решать проблемы высокой загрузки ЦП из-за процесса Java. В высокой загрузке ЦП заключает вас в корпус, обычно узнают, что процесс Java использует высокую загрузку CPU от системы.

Процедура

Шаг 1: Введите **главную** команду Linux для определения, какой процесс использует высокую загрузку CPU от виртуальной машины (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
```

От этих выходных данных выньте процессы , которые используют больше %CPU. Здесь, Java берет 5.9%, но он может использовать больше ЦП, такого как больше чем 40%, 100%, 200%, 300%, 400%, и так далее.

Шаг 2: Если процесс Java использует высокую загрузку CPU, введите одну из этих команд для обнаружения, который поток использует сколько:

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

Или

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

Как пример, этот показ показывает, что процесс Java использует высокую загрузку CPU (+40%), а также потоки процесса Java, ответственного за высокий коэффициент использования.

```
<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
```

Что такое поток?

Предположим, что у вас есть приложение (т.е. одиночный рабочий процесс) в системе. Однако для выполнения многих задач, вы требуете, чтобы много процессов были созданы, и каждый процесс создает много потоков. Некоторые потоки могли быть читателем, писателем и другими целями, такими как создание Подробной записи о вызове (CDR) и так далее.

В предыдущем примере ID процесса Java (например, 28026) имеет многопроцесного, который включает 30915, 30916, 30927 и еще много.

> **Примечание:** Идентификатор потока (TID) находится в десятичном формате.

Шаг 3: Проверьте функциональность потоков Java, которые используют высокую загрузку CPU.

Введите эти команды Linux для получения завершенной трассировки Jstack. ID процесса является PID Java, например 28026 как показано в предыдущих выходных данных.

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

# jstack <process ID>
```

Выходные данные предыдущей команды похожи:

```
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
```

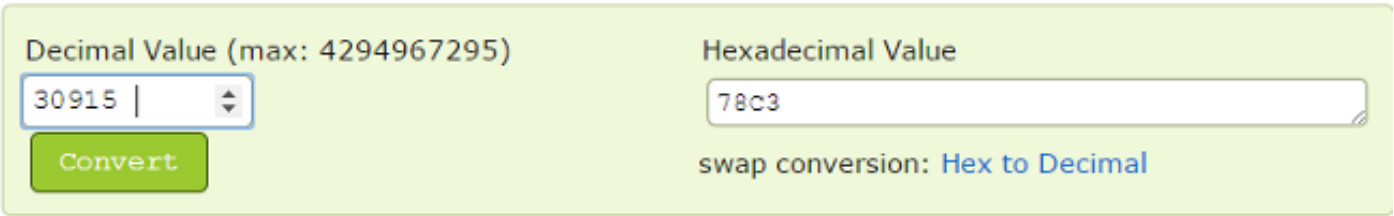
```
(SynchronousQueue.java:359)
at java.util.concurrent.SynchronousQueue.poll(SynchronousQueue.java:942)
at java.util.concurrent.ThreadPoolExecutor.getTask(ThreadPoolExecutor.java:1068)
at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1130)
at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:615)
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)
```

Теперь необходимо определить, какой поток процесса Java ответственен за высокую загрузку ЦП.

Как пример, посмотрите на TID 30915, как упомянуто в Шаге 2. Необходимо преобразовать TID в десятичном числе к шестнадцатеричному формату, потому что в трассировке Jstack, можно только найти шестнадцатеричную форму. Используйте этот преобразователь для преобразования десятичного формата в шестнадцатеричный формат.



Как вы можете видеть в Шаге 3, вторая половина трассировки Jstack является потоком, который является одним из ответственных потоков позади высокой загрузки ЦП. Когда вы найдете 78C3 (шестнадцатеричный формат) в трассировке Jstack, тогда вы только найдете этот поток как 'nid=0x78c3'. Следовательно, можно найти все потоки того процесса Java, которые ответственны за потребление высокой загрузки CPU.

> **Примечание:** Вы не должны фокусироваться на состоянии потока на данный момент. Как интересное место, были замечены некоторые состояния потоков как Выполнимый, Заблокированный, Timed_Waiting и Ожидание.

Вся предыдущая информация помогает CPS, и другие разработчики технологии помогают вам добираться до основной причины проблемы высокой загрузки ЦП в системе/VM. Перехватите ранее упомянутую информацию в то время, когда появляется проблема. Как только загрузка ЦПУ вернулась к обычному тогда потоки, которые вызвали проблему высокой загрузки CPU, не может быть определен.

Журналы CPS должны быть перехвачены также. Вот список журналов CPS от 'PCRFclient01' VM под путем '/var/log/broadhop':

- **объединенный механизм**
- **объединенный-qns**

Кроме того, получите выходные данные этих сценариев и команд от PCRFclient01 VM:

- **# diagnostics.sh** (Этот сценарий не мог бы работать на более старых версиях CPS, таких как QNS 5.1 и QNS 5.2.)
- **# df-kh**
- **Вершина #**