排除故障Java斯塔克高CPU利用率

目录

简介

本文在思科策略套件(CPS)描述Java斯塔克(Jstack)和如何使用它为了确定高CPU利用率的根本原因。

排除故障与Jstack

Jstack是什么？

Jstack采取运行Java进程的内存转储(在CPS， QNS是Java进程)。Jstack有那的所有详细信息Java进程，例如线索/应用程序和每个线索的功能。

为什么需要Jstack ？

Jstack提供Jstack trace，以便工程师和开发员能认识每个线索的状态。

用于的linux命令获取Java进程的Jstack trace是：

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

Jstack进程的位置在每CPS的(以前叫作Quantum策略套件(QPS))版本是'jdk1.7.0_10是JAVA版本的'/usr/java/jdk1.7.0_10/bin/，并且JAVA版本在每个系统能有所不同。

您能也输入linux命令为了查找Jstack进程的确切的路径：

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

Jstack解释此处为了使您熟悉步骤排除故障高CPU利用率问题由于Java进程。在高CPU利用率装入您通常学习Java进程使用从系统的高CPU。

步骤

步骤 1：输入**顶部**linux命令为了确定哪进程消耗从虚拟机的高CPU。

```
[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
```

从此输出，请去掉消耗更多%CPU的进程。这里， Java采取5.9%，但是能消耗更多CPU例如超过40%， 100%， 200%， 300%， 400%，等等。

步骤 2：如果Java进程消耗高CPU，请输入线索消耗这些命令的之一为了发现多少：

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

或者

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

为例，此显示显示Java进程消耗高CPU (+40%)以及Java的线索处理负责对高利用率。

```
<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
```

什么是线索？

假设您有一应用程序(即单个运行进程)在系统里面。然而，为了执行许多任务您需要将创建的许多进程，并且每进程创建许多线索。某些线索可能是读者、作家和不同的目的例如呼叫详细信息详情记录(CDR)创建等等。

在前一个示例中， Java进程ID (例如， 28026)有包括30915， 30916， 30927和许多的多个线索。

注意：线索ID (TID)在十进制形式。

步骤 3：检查消耗高CPU Java线索的功能。

输入这些Linux命令为了获取完整Jstack trace。如上一个输出所显示，进程ID是Java PID，例如28026。

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

# jstack <process ID>
```

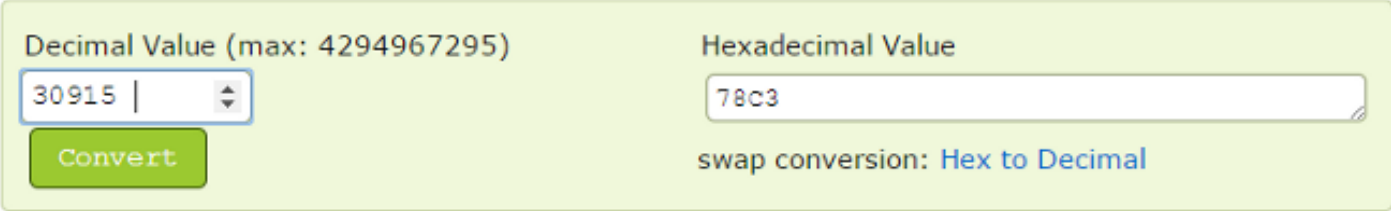
前面的命令的输出看似类似：

```
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)
```

现在您需要确定Java进程的哪个线索对高CPU利用率负责。

为例，看看按照步骤2.所述的TID 30915。因为在Jstack trace，您能只找到十六进制表，您需要转换在十进制的TID到十六进制格式。请使用此转换器为了转换十进制形式到十六进制格式。



正如你在步骤3看到， Jstack trace的第二半是其中一个在高CPU利用率后的负责的线索的线索。当您查找78C3 (十六进制格式)在Jstack trace，然后您只将查找此线索作为'nid=0x78c3。因此，您能找到对高CPU消耗负责那的所有线索Java进程。

> **注意**：您不需要暂时着重线索的状态。作为问题的兴趣，线索的一些状态类似可追捕，阻止，Timed_Waiting和等待看到。

所有上一个信息帮助CPS和其他技术开发员协助解决您达到高CPU利用率问题的根本原因在system/VM的。在问题出现时候，请获取以前被提及的信息。一旦CPU利用率是回到正常然后导致高CPU问题的线索不可能确定。

CPS日志需要捕获。这是CPS日志列表从'PCRFclient01 VM的在路径'/var/log/broadhop下'：

- **统一引擎**
- **统一qns**

并且，请从PCRFclient01 VM得到这些脚本和命令输出：

- **# diagnostics.sh** (此脚本在CPS更旧的版本也许不运行，例如QNS 5.1和QNS 5.2。)
- **# df - kh**
- **#顶部**