

01、高cpu占用

1、代码

```
public class FindJavaThreadInTaskManager {  
    public static void main(String[] args) {  
        Thread thread = new Thread(new Worker());  
        thread.start();  
    }  
    static class Worker implements Runnable {  
        @Override  
        public void run() {  
            while (true) {  
                System.out.println("Thread Name:" + Thread.currentThread().getName());  
            }  
        }  
    }  
}
```

2、运行

java -jar jstacktopcpu.jar

3、top

选择大写P，进程按照cpu排序，找到最耗内存的java进程。

4、获取线程号

ps p 28348 -L -o pcpu,pid,tid,time,tname,cmd

5、将获取的线程号（十进制数）转换成十六进制,此处为0xb46

printf "%x\n" 28376

0x6ed8

6、查看进程PID为28348 中

nid为0x6ed8的线程信息。

命令：

jstack -l 28348 >28348.jstack

完整文档:

java程序CPU利用率高怎么办

请jstack神器来帮忙

本文介绍Linux环境下使用jstack定位问题的秘笈

工具/原料

- Linux
- java
- thread
- jstack
- top
- ps

- printf
- Runnable

方法/步骤

1. 一个CPU密集型线程的demo：

package chapter1;

public class FindJavaThreadInTaskManager {

public static void main(String[] args) {

Thread thread = new Thread(new Worker());

thread.start();

}

static class Worker implements Runnable {

@Override

public void run() {

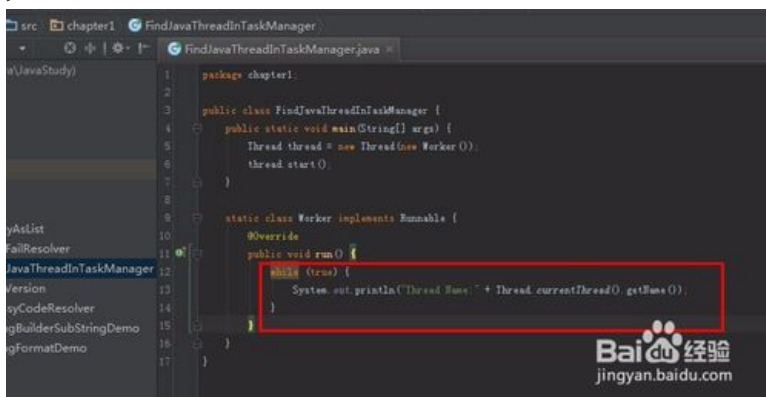
while (true) {

System.out.println("Thread Name:" + Thread.currentThread().getName());

}

}

}



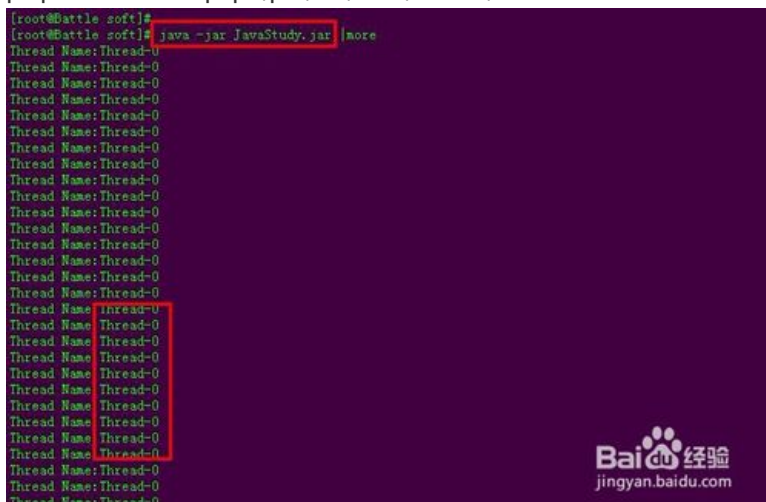
2. 将上述代码打成Jar。

在Linux上执行上述代码

命令：

java -jar jstack.jar

ps p 27892 -L -o pcpu,pid,tid,time,tname,cmd



3. 找到CPU利用率持续比较高的进程，获取进程号，此处PID为 27892

命令：

top

```
top 0.01s user 0.01s system 0.00s idle 99.98%
  PID TID          PPID    CPU   TIME     TTY      CMD
  3036 3036          3036    0.0   00:00:00 pts/1    java -jar JavaStudy.jar
  3036 3037          3036    0.1   00:00:00 pts/1    java -jar JavaStudy.jar
  3036 3038          3036    2.1   00:00:07 pts/1    java -jar JavaStudy.jar
  3036 3039          3036    0.0   00:00:00 pts/1    java -jar JavaStudy.jar
  3036 3040          3036    0.0   00:00:00 pts/1    java -jar JavaStudy.jar
  3036 3041          3036    0.0   00:00:00 pts/1    java -jar JavaStudy.jar
  3036 3042          3036    0.1   00:00:00 pts/1    java -jar JavaStudy.jar
  3036 3043          3036    0.0   00:00:00 pts/1    java -jar JavaStudy.jar
  3036 3044          3036    0.0   00:00:00 pts/1    java -jar JavaStudy.jar
  3036 3045          3036    0.7   00:00:02 pts/1    java -jar JavaStudy.jar
  3036 3046          3036   62.2  00:03:22 pts/1    java -jar JavaStudy.jar
```

4. 找到上述进程中，CPU利用率比较高的线程号TID（十进制数），此处为3046

命令：

ps p 27892 -L -o pcpu,pid,tid,time,tname,cmd

```
[root@Battle soft]# ps p 27892 -L -o pcpu,pid,tid,time,tname,cmd
%CPU    PID     TID      TIME    TTY      CMD
  0.0    3036    3036    00:00:00 pts/1    java -jar JavaStudy.jar
  0.1    3036    3037    00:00:00 pts/1    java -jar JavaStudy.jar
  2.1    3036    3038    00:00:07 pts/1    java -jar JavaStudy.jar
  0.0    3036    3039    00:00:00 pts/1    java -jar JavaStudy.jar
  0.0    3036    3040    00:00:00 pts/1    java -jar JavaStudy.jar
  0.0    3036    3041    00:00:00 pts/1    java -jar JavaStudy.jar
  0.1    3036    3042    00:00:00 pts/1    java -jar JavaStudy.jar
  0.0    3036    3043    00:00:00 pts/1    java -jar JavaStudy.jar
  0.0    3036    3044    00:00:00 pts/1    java -jar JavaStudy.jar
  0.7    3036    3045    00:00:02 pts/1    java -jar JavaStudy.jar
 62.2    3036    3046    00:03:22 pts/1    java -jar JavaStudy.jar
```

5. 将获取的线程号（十进制数）转换成十六进制，此处为0xb46

命令：

printf "%x\n" 27892

```
[root@Battle soft]# printf "%x\n" 27892
be6
```

6. 查看进程PID为3036中

nid为0x6cf4的线程信息。

命令：

jstack -l 27892

```
[root@battle soft]#
[root@battle soft]#
[root@battle soft]# /opt/jdk1.8.0_73/bin/jstack -l 3036|more
2016-03-12 00:19:03
Full thread dump OpenJDK 64-Bit Server VM (25.71-b15 mixed mode):

"Attach Listener" #10 daemon prio=0 os_prio=0 tid=0x00007fed6c001000 nid=0xc02 waiting on condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

    Locked ownable synchronizers:
      - None

"DestroyJavaVM" #9 prio=0 os_prio=0 tid=0x00007fed94008800 nid=0xbdd waiting on condition [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

    Locked ownable synchronizers:
      - None

Thread-0" #8 prio=5 os_prio=0 tid=0x00007fed94114000 nid=0xbef runnable [0x00007fed9903b000]
  java.lang.Thread.State: RUNNABLE
    at java.util.Arrays.copyOfRange(Arrays.java:3604)
    at java.lang.String.<init>(String.java:207)
    at java.lang.StringBuilder.<init>(StringBuilder.java:407)
    at chapter1.FindJavaThreadInTaskManager$Worker.run(FindJavaThreadInTaskManager.java:13)
    at java.lang.Thread.run(Thread.java:745)

    Locked ownable synchronizers:
      - None

"Service Thread" #7 daemon prio=0 os_prio=0 tid=0x00007fed940e9800 nid=0xbef runnable [0x0000000000000000]
  java.lang.Thread.State: RUNNABLE

    Locked ownable synchronizers:
      - None
```

7. 总结:

可以看到jstack命令的输出结果是相当准确的:

显示耗CPU比较高的代码与实际情况相同, 都是第13行。

放心的用吧。

Enjoy yourself!

```
FindJavaThreadInTaskManager.java x
1 package chapter1;
2
3 public class FindJavaThreadInTaskManager {
4     public static void main(String[] args) {
5         Thread thread = new Thread(new Worker());
6         thread.start();
7     }
8
9     static class Worker implements Runnable {
10         @Override
11         public void run() {
12             while (true) {
13                 System.out.println("Thread Name:" + Thread.currentThread().getName());
14             }
15         }
16     }
17 }
```