

# 线上CPU100%？看看这篇是怎么排查的！

## 前言

作为后端开发工程师,当收到线上服务器CPU负载过高告警时,你会这么做?重启服务,忽略告警?不过在我看来一个合格的工程师是一定要定位到具体问题所在的,从而 fix 它。下面记录一下线上服务器 CPU 负载过高排查过程,把排查流程理清楚,以后遇到问题将会迅速定位到问题所在,快速解决。

## 什么样的场景会导致线上CPU负载过高？

代码层面常见的场景有：

1. 程序陷入死循环，不停地消耗CPU
2. 线程死锁，线程相互等待，导致假死状态，不停地消耗CPU

## 程序死循环场景

这里使用 JAVA 简单模拟程序死循环带来的系统高负载情况，代码如下：

```
1  /**
2   * @program: easywits
3   * @description: 并发下的 HashMap 测试....
4   * @author: zhangshaolin
5   * @create: 2018-12-19 15:27
6   **/
7  public class HashMapMultiThread {
8
9      static Map<String, String> map = new HashMap<>();
10
11     public static class AddThread implements Runnable {
12
13         int start = 0;
14         public AddThread(int start) {
15             this.start = start;
16         }
17         @Override
18         public void run() {
19             // 死循环, 模拟CPU占用过高场景
20             while (true) {
21                 for (int i = start; i < 100000; i += 4) {
22                     map.put(Integer.toString(i), Integer.toBinaryString(i));
23                 }
24             }
25         }
26     }
27 }
```

```

25     }
26     public static void main(String[] args) throws InterruptedException {
27         //线程并发对 HashMap 进行 put 操作 如果一切正常,则得到 map.size() 为10
28
29         //可能的结果:
30         //1. 程序正常, 结果为100000
31         //2. 程序正常, 结果小于100000
32         Thread thread1 = new Thread(new AddThread(0), "myTask-1");
33         Thread thread2 = new Thread(new AddThread(1), "myTask-2");
34         Thread thread3 = new Thread(new AddThread(2), "myTask-3");
35         Thread thread4 = new Thread(new AddThread(3), "myTask-4");
36         thread1.start();
37         thread2.start();
38         thread3.start();
39         thread4.start();
40         thread1.join();
41         thread2.join();
42         thread3.join();
43         thread4.join();
44         System.out.println(map.size());
45     }
46 }
47 }

```

## 线程死锁场景

同样使用 JAVA 程序简单模拟线程死锁场景，代码如下：

```

1  /**
2   * @program: easywits
3   * @description: 死锁 demo ....
4   * 1. 两个线程里面分别持有两个Object对象：lock1和lock2。这两个lock作为同步代码块的锁；
5   * 2. 线程1的run()方法中同步代码块先获取lock1的对象锁，Thread.sleep(xxx)，时间不需要太
6   * 这么做主要是为了防止线程1启动一下子就连续获得了lock1和lock2两个对象的对象锁
7   * 3. 线程2的run()方法中同步代码块先获取lock2的对象锁，接着获取lock1的对象锁，当然这时loc
8   * <p>
9   * 线程1“睡觉”睡完，线程2已经获取了lock2的对象锁了，线程1此时尝试获取lock2的对象锁，便被
10  * @author: zhangshaolin
11  * @create: 2018-12-20 11:33
12  */
13  public class DeadLock {
14
15      static Object lock1 = new Object();
16      static Object lock2 = new Object();
17

```

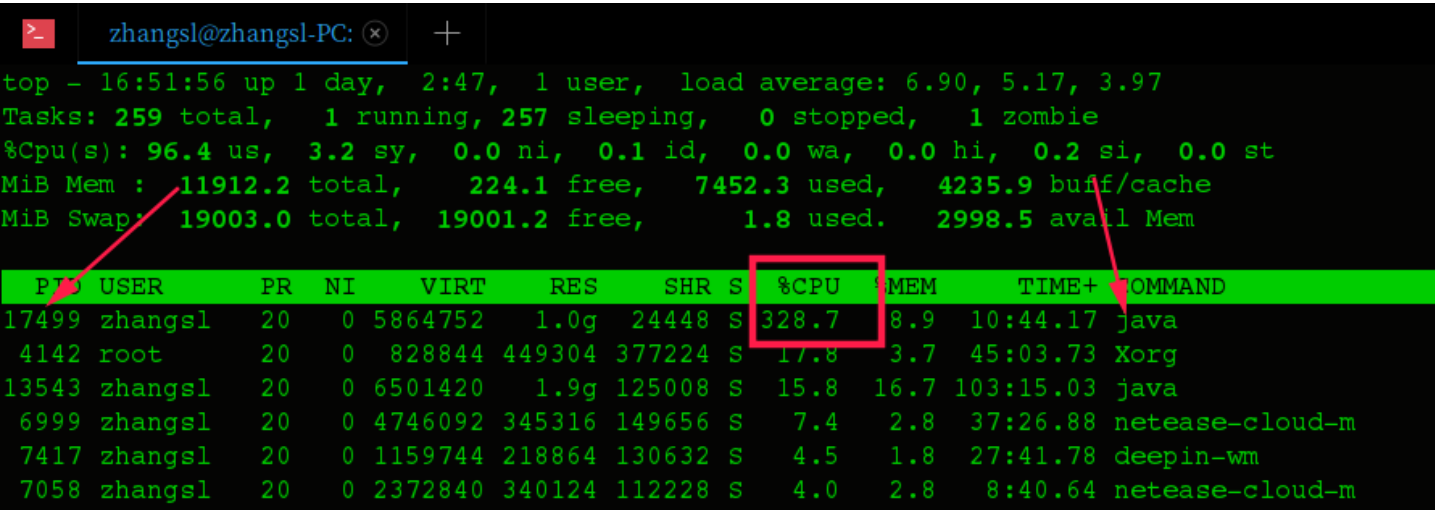
```
18 public static class Task1 implements Runnable {
19
20     @Override
21     public void run() {
22         synchronized (lock1) {
23             System.out.println(Thread.currentThread().getName() + " 获得了
24
25             try {
26                 Thread.sleep(50);
27             } catch (InterruptedException e) {
28                 e.printStackTrace();
29             }
30
31             synchronized (lock2) {
32                 System.out.println(Thread.currentThread().getName() + " 获得了
33             }
34         }
35     }
36 }
37
38 public static class Task2 implements Runnable {
39
40     @Override
41     public void run() {
42         synchronized (lock2) {
43             System.out.println(Thread.currentThread().getName() + " 获得了
44
45             synchronized (lock1) {
46                 System.out.println(Thread.currentThread().getName() + " 获得了
47             }
48         }
49     }
50 }
51
52 public static void main(String[] args) throws InterruptedException {
53     Thread thread1 = new Thread(new Task1(), "task-1");
54     Thread thread2 = new Thread(new Task2(), "task-2");
55     thread1.start();
56     thread2.start();
57
58     thread1.join();
59     thread2.join();
60     System.out.println(Thread.currentThread().getName() + " 执行结束!");
61 }
62 }
```

以上两种场景代码执行后，不出意外，系统CPU负载将会飙升，我的机器，4核CPU已经明显感觉到卡顿了，所以线上应该杜绝出现死循环代码。。

## 使用 top 命令监控当前系统负载情况

执行第一种场景测试代码。

在 linux 命令行键入 top 指令后，就开始实时监控当前系统的负载信息，监控到的负载信息如下图所示：



从图中的监控信息可以快速大致的了解到，PID 为 17499 的进程CPU负载高达 328+%，是一个 JAVA 程序。简单介绍下监控信息如下：

- PID：进程的ID
- USER：进程所有者
- PR：进程的优先级别，越小越优先被执行
- VIRT：进程占用的虚拟内存
- RES：进程占用的物理内存
- SHR：进程使用的共享内存
- S：进程的状态。S表示休眠，R表示正在运行，Z表示僵死状态，N表示该进程优先值为负
- %CPU：进程占用CPU的使用率
- %MEM：进程使用的物理内存和总内存的百分比
- TIME+：该进程启动后占用的总的CPU时间，即占用CPU使用时间的累加值

在监控页面下 按键盘数字 1 可以看到每个CPU的负载情况，如下图：

```
>_ zhangsl@zhangsl-PC: ~ +
top - 16:50:33 up 1 day, 2:46, 1 user, load average: 6.28, 4.59, 3.69
Tasks: 59 total, 1 running, 257 sleeping, 0 stopped, 1 zombie
%Cpu0 96.5 us, 2.9 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.6 si, 0.0 st
%Cpu1 97.6 us, 2.4 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu2 97.0 us, 3.0 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 0.0 si, 0.0 st
%Cpu3 92.9 us, 5.9 sy, 0.0 ni, 0.0 id, 0.0 wa, 0.0 hi, 1.2 si, 0.0 st
MiB Mem : 11912.2 total, 220.8 free, 7466.3 used, 4225.1 buff/cache
MiB Swap: 19003.0 total, 19001.2 free, 1.8 used. 3007.2 avail Mem

  PID USER      PR  NI   VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
17499 zhangsl    20   0 5864752    1.0g  24448 S 338.8   8.9   6:07.04 java
13543 zhangsl    20   0 6501420    1.9g 125008 S  15.9  16.7 103:02.46 java
 4142 root       20   0  826860   447044 374964 S  15.3   3.7  44:51.06 Xorg
 6999 zhangsl    20   0 4745836   344552 149176 S   7.6   2.8  37:20.13 netease-cloud-m
```

可以看到开了四个线程，无限循环之后，我的机器中四个核心CPU，每颗负载接近百分百。

## 使用 top 命令监控进程中负载过高的线程

top -H -p pid: 查看指定进程中每个线程的资源占用情况(每条线程占用CPU时间的百分比)，监控结果如下图：

```
>_ zhangsl@zhangsl-PC: ~ +
top - 16:53:32 up 1 day, 2:49, 1 user, load average: 6.39, 5.52, 4.22
Threads: 20 total, 4 running, 16 sleeping, 0 stopped, 0 zombie
%Cpu(s): 96.2 us, 3.3 sy, 0.0 ni, 0.2 id, 0.0 wa, 0.0 hi, 0.3 si, 0.0 st
MiB Mem : 11912.2 total, 227.0 free, 7454.2 used, 4230.9 buff/cache
MiB Swap: 19003.0 total, 19001.2 free, 1.8 used. 3005.4 avail Mem

  PID USER      PR  NI   VIRT    RES    SHR S  %CPU  %MEM    TIME+  COMMAND
17532 zhangsl    20   0 5864752    1.0g  24448 R  91.7   8.9   3:54.39 java
17535 zhangsl    20   0 5864752    1.0g  24448 R  86.7   8.9   3:53.75 java
17533 zhangsl    20   0 5864752    1.0g  24448 R  80.4   8.9   3:53.36 java
17534 zhangsl    20   0 5864752    1.0g  24448 R  70.1   8.9   3:53.33 java
17507 zhangsl    20   0 5864752    1.0g  24448 S   1.3   8.9   0:04.15 java
17509 zhangsl    20   0 5864752    1.0g  24448 S   1.3   8.9   0:04.10 java
17510 zhangsl    20   0 5864752    1.0g  24448 S   1.3   8.9   0:04.16 java
```

以上监控指令输出的指标针对的是某个进程中的线程，从图中看可以快速得出结论：四个 JAVA 线程 CPU负载极高，线程ID分别为: 17532, 17535, 17533, 17534,注意这里打印出来的线程ID为十进制的哦！

## 根据 进程pid && 线程id 查看线程堆栈信息

- `jstack pid`:查看指定进程中线程的堆栈信息，这个命令最终会打印出指定进程的线程堆栈信息，而实际线上情况发生时，我们应当把快速把堆栈信息输出到日志文本中，保留日志信息，然后迅速先重启服务，达到临时缓解服务器压力的目的。
- `jstack 17499 > ./threadDump.log`：将线程堆栈信息输出到当前目录下的 threadDump.log 文件。

**注意：**jstack 打印出的线程id号为十六进制，而 top 命令中打印出来的线程号为十进制，需要进行转换后，定位指定线程的堆栈信息

这里分析日志文件后，过滤出四个线程堆栈信息如下图：

```
"myTask-4" #13 prio=5 os_prio=0 tid=0x00007fca38889000 nid=0x447f runnable [0x00007fca1cdfb000]
java.lang.Thread.State: RUNNABLE
  at java.util.HashMap.put(HashMap.java:612)
  at com.easywits.common.util.concurrent.HashMapMultiThread$AddThread.run(HashMapMultiThread.java:29)
  at java.lang.Thread.run(Thread.java:748)

"myTask-3" #12 prio=5 os_prio=0 tid=0x00007fca38887000 nid=0x447e runnable [0x00007fca1cefc000]
java.lang.Thread.State: RUNNABLE
  at java.util.HashMap.put(HashMap.java:612)
  at com.easywits.common.util.concurrent.HashMapMultiThread$AddThread.run(HashMapMultiThread.java:29)
  at java.lang.Thread.run(Thread.java:748)

"myTask-2" #11 prio=5 os_prio=0 tid=0x00007fca38885000 nid=0x447d runnable [0x00007fca1cffd000]
java.lang.Thread.State: RUNNABLE
  at java.util.HashMap.put(HashMap.java:612)
  at com.easywits.common.util.concurrent.HashMapMultiThread$AddThread.run(HashMapMultiThread.java:29)
  at java.lang.Thread.run(Thread.java:748)

"myTask-1" #10 prio=5 os_prio=0 tid=0x00007fca38884000 nid=0x447c runnable [0x00007fca241cd000]
java.lang.Thread.State: RUNNABLE
  at java.util.HashMap.put(HashMap.java:612)
  at com.easywits.common.util.concurrent.HashMapMultiThread$AddThread.run(HashMapMultiThread.java:29)
  at java.lang.Thread.run(Thread.java:748)
```

从这四个线程执行的堆栈信息，很明显的看出：导致CPU飙升的程序正在执行 HashMap 的 put 操作。