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：

```
/**
 * @program: easywits
 * @description: 并发下的 HashMap 测试....
 * @author: zhangshaolin
 * @create: 2018-12-19 15:27
 **/
public class HashMapMultiThread {

    static Map<String, String> map = new HashMap<>();

    public static class AddThread implements Runnable {

        int start = 0;
        public AddThread(int start) {
            this.start = start;
        }
        @Override
        public void run() {
            //死循环,模拟CPU占用过高场景
            while (true) {
                for (int i = start; i < 100000; i += 4) {
                    map.put(Integer.toString(i), Integer.toBinaryString(i));
                }
            }
```

```
        }
        public static void main(String[] args) throws InterruptedException {
            //线程并发对 HashMap 进行 put 操作  如果一切正常,则得到 map.size() 为10

            //可能的结果:
            //1. 程序正常,结果为100000
            //2. 程序正常,结果小于100000
            Thread thread1 = new Thread(new AddThread(0), "myTask-1");
            Thread thread2 = new Thread(new AddThread(1), "myTask-2");
            Thread thread3 = new Thread(new AddThread(2), "myTask-3");
            Thread thread4 = new Thread(new AddThread(3), "myTask-4");
            thread1.start();
            thread2.start();
            thread3.start();
            thread4.start();
            thread1.join();
            thread2.join();
            thread3.join();
            thread4.join();
            System.out.println(map.size());
        }
    }
}
```

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：

```
/**
 * @program: easywits
 * @description: 死锁 demo ....
 * 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；
 * 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
 * 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
 * 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lo
 * <p>
 * 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
 * @author: zhangshaolin
 * @create: 2018-12-20 11:33
 **/
public class DeadLock {

    static Object lock1 = new Object();
    static Object lock2 = new Object();

```

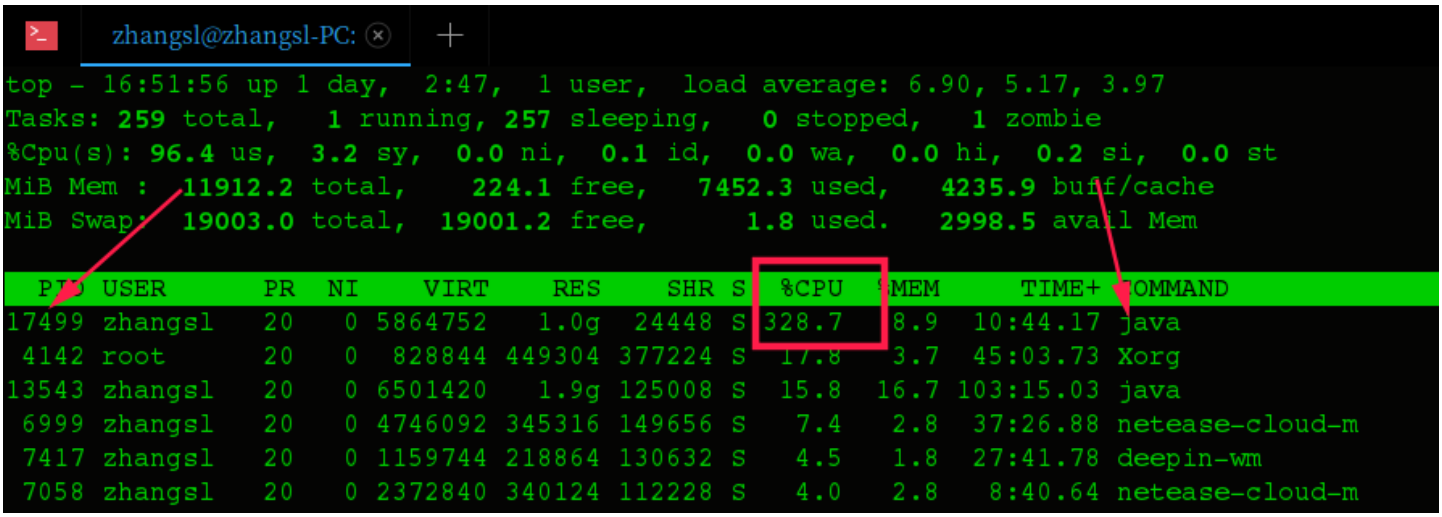
```
    public static class Task1 implements Runnable {

        @Override
        public void run() {
            synchronized (lock1) {
                System.out.println(Thread.currentThread().getName() + " 获得了

                try {
                    Thread.sleep(50);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                synchronized (lock2) {
                    System.out.println(Thread.currentThread().getName() + " 获
                }
            }
        }
    }

    public static class Task2 implements Runnable {

        @Override
        public void run() {
            synchronized (lock2) {
                System.out.println(Thread.currentThread().getName() + " 获得了

                synchronized (lock1) {
                    System.out.println(Thread.currentThread().getName() + " 获
                }
            }
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread1 = new Thread(new Task1(), "task-1");
        Thread thread2 = new Thread(new Task2(), "task-2");
        thread1.start();
        thread2.start();

        thread1.join();
        thread2.join();
        System.out.println(Thread.currentThread().getName() + " 执行结束!");
    }
}
```

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。

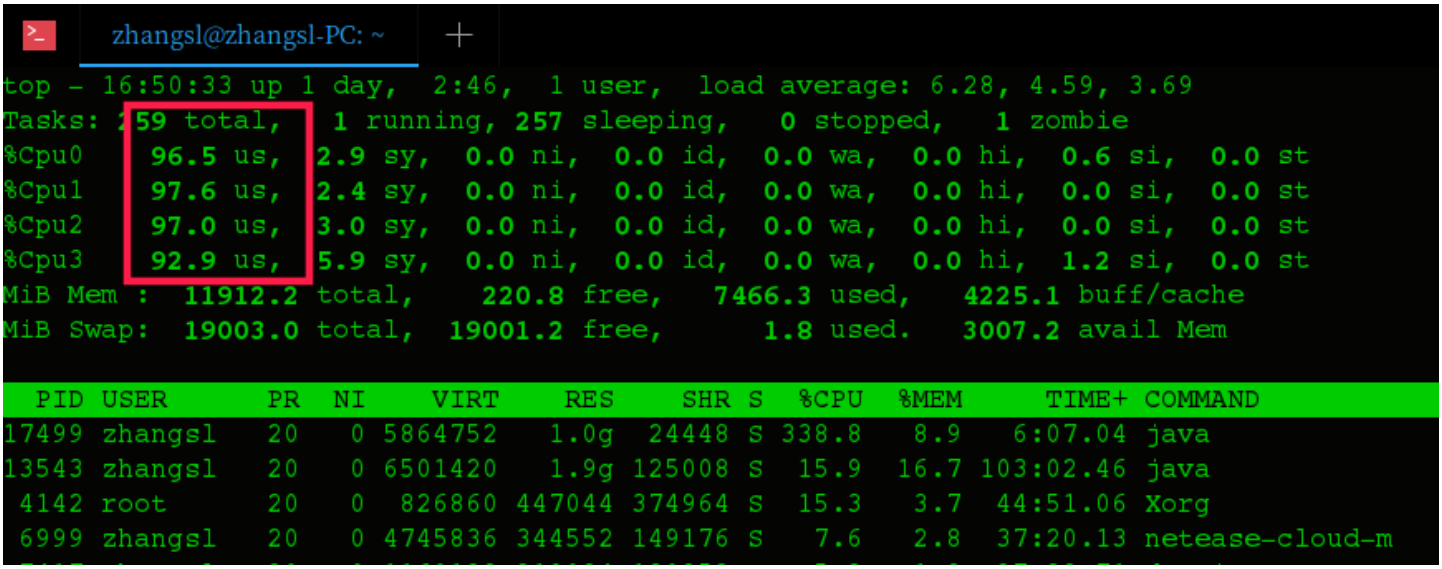
在 linux 命令行键入 top 指令后，就开始实时监控当前系统的负载信息，监控到的负载信息如下图所示：



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

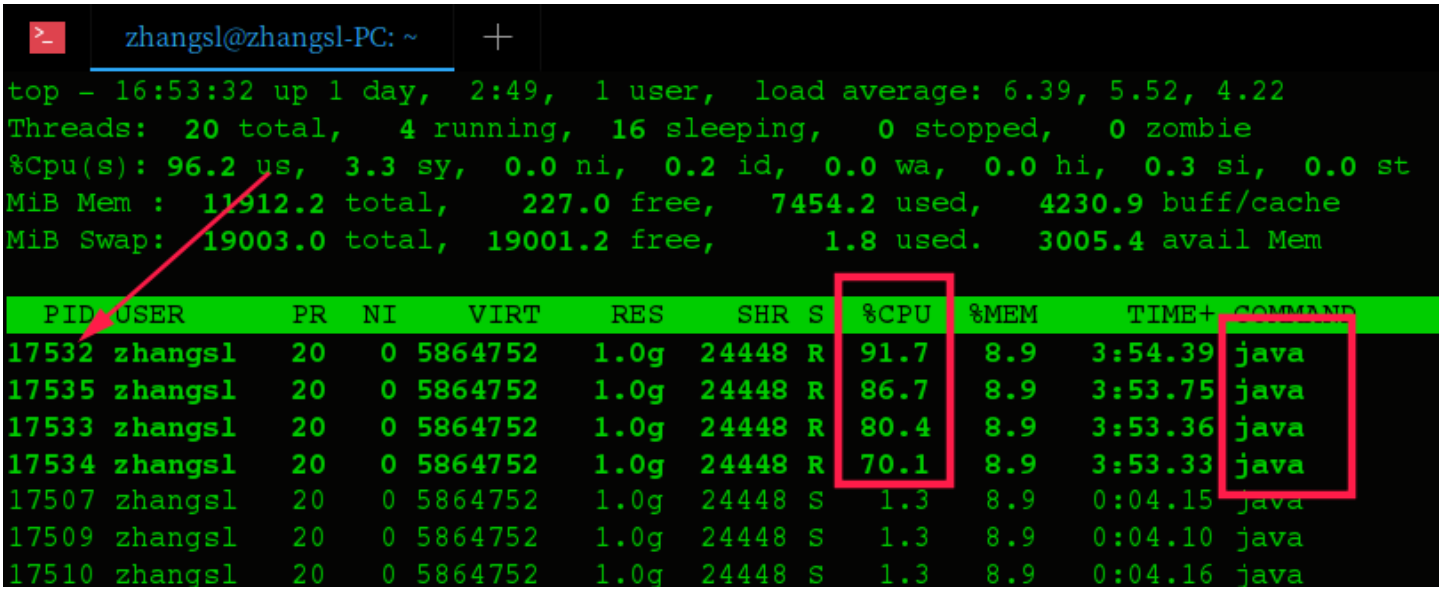
在监控页面下 按键盘数字 1 可以看到每个CPU的负载情况，如下图：



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：



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)                    top命令获取到线程高负载的线程号，转化为十六进制，定位
    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=0x00007fca3888700( 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。