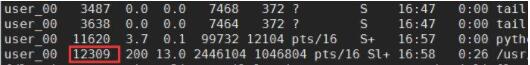
问题出现：现网CPU飙高，Full GC告警

CGI 服务发布到现网后，现网机器出现了Full GC告警，同时CPU飙高99%。在优先恢复现网服务正常后，开始着手定位Full GC的问题。在现场只能够抓到四个GC线程占用了很高的CPU，无法抓到引发Full GC的线程。查看了服务故障期间的错误日志，发现更多的是由于Full GC引起的问题服务异常日志，无法确定Full GC的根源。为了查找问题的根源，只能从发布本身入手去查问题，发现一次bugfix的提交，有可能触发一个死循环逻辑：  
for(int i = 1 ;i <= totalPage ;i++) {  
String path = path\_prefix + "?cmd=txt\_preview&page=" + String.valueOf(i) + "&sign=" + fileSignature;  
url\_list.add(path);  
}  
  
循环中的参数 totalPage 为 long 类型，由一个外部参数进行赋值。当外部参数非常大，超过 int 的最大值时，i递增到int的最大值后，i++ 会发生翻转，变成一个负数，从而使 for 会进入死循环。利用下面这段代码可以试验：  
public static void main(String[] args) {  
long totalPage = Long.MAX\_VALUE;  
for(int i = 0 ;i<totalPage;i++){  
if(i<0){  
System.out.println(i);  
}  
}  
}  
  
通过日志，发现外部确实传递了一个非常大的参数：

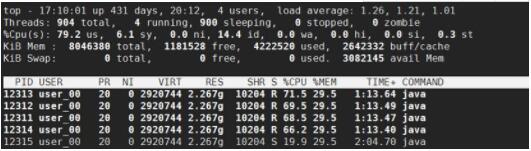
http://imgsrc.baidu.com/forum/w%3D580/sign=1883f2b49d529822053339cbe7cb7b3b/cfa13e9b033b5bb5974aee8d3bd3d539b700bcbe.jpg

确认了当命中逻辑的时候，会进入一个死循环。在循环中不断进行字符串的拼接与list的Add操作，很快就会耗尽JVM堆内存导致Full GC。经过测算，实际上并不需要死循环，只要是一个比较大的循环，就能够引发Full GC。对totlePage的大小做了限定后，发布了新版本，没有再出现Full GC的问题。  
现场还原：重现问题，探索定位思路  
回顾排查问题的过程并不高效，最开始怀疑过是否是打包有问题或使用的jdk版本不对，花了较多的时间确认打包问题。另一方面，发布带出的代码较多，通过重复review代码无法很快锁定问题。为了探索一种更有效的问题定位方法，我将有问题的代码重新部署到机器上，手动构造请求触发bug，探索定位此类问题的通用思路。

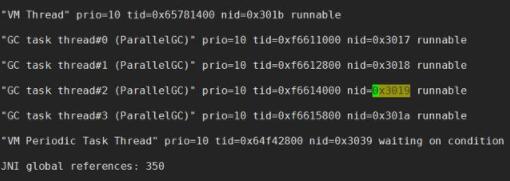
如何获取占用CPU最高的线程id？  
2）可以登上机器，确认下是什么线程使 CPU 飙高。先ps查看 Java 进程的 PID：



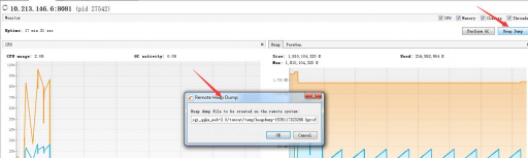
拿到进程 pid 后，可以使用 top 命令，来看是什么线程占用了 CPU。  
top -p 12309 -H  
  
-p 用于指定进程，-H 用于获取每个线程的信息，从 top 输出的内容，可以看到有四个线程占用了非常高的 CPU：



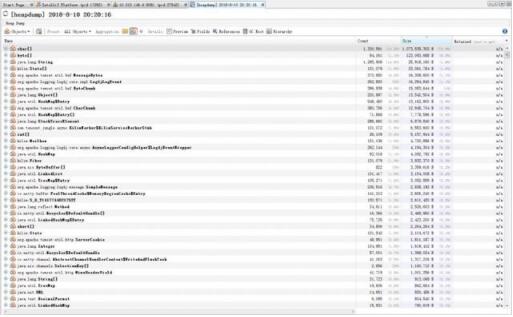
到这里可以拿到12313、12312、12311、12314这四个线程id。为了确定这些是什么线程，需要使用 jstack 命令来查看这几个是什么线程。  
高占用CPU的是什么线程？  
3) jstack 是 Java 虚拟机自带的一种堆栈跟踪工具，用于打印出给定的 Java 进程 ID 或 core file 或远程调试服务的 Java 堆栈信息。使用下面命令，将 Java 进程的堆栈信息打印到文件中：  
jstack -l 12309 > stack.log  
  
在线程堆栈信息中，线程 id 是使用十六进制来表示的。将上面四个四个线程 id 转换为16进制，分别是0X3019、0X3018、0×3017、0x301A。在 stack.log 中可以找到这几个线程：



到这里可以确定的是，死循环引发了Full GC,四个GC线程一直尝试着回收内存，这四个线程将CPU占满。

是哪些对象占用了内存？  
4）Full GC、OOM、CPU 被占满的问题都得到了解答。那么再次遇到类似的线上问题时，如何确定或者缩小问题范围，找到导致问题的代码呢？这时候需要进一步观察的是 Java 堆内存的信息，查看是什么对象占用了内存。可以使用上文提到的 VisualVM 来生成 headdump 文件：

也可以在机器上使用 jmap 命令来生成 head dump 文件。  
jmap -dump:live,format=b,file=headInfo.hprof 12309  
  
live 这个参数表示我们需要抓取的是目前在生命周期内的内存对象，也就是说 GC 收不走的对象，在这种场景下，我们需要的就是这些内存的信息。生成了 hprof 文件后，可以拉回到本地，使用 VisualVM 来打开它进行分析。打开后可以看到：



从信息中可以看到，字符串 char[] 在占了内存的73%，因此可以确定的是内存泄漏与字符串有关。通常生成的 headdump 文件会很大，也可以使用下面的命令，来查看占用内存最多的类型：  
jmap -histo 12309 > heap.log  
  
输出内容如下：

