# JVM底层TLAB和PLAN

TLAB：新生代 线程独享 的堆区，最核心知识点refill

PLAB：老年代 线程独享 的堆区

http://note.youdao.com/noteshare?id=c5e22df47097a4f6f0fe0fa9e822929d&sub=513B24207540456B8B50C3D02CC82297

# OOM

Out of memory 内存泄露

（1）为什么会发生内存泄露？

回收的速度比不上用的速度

（2）JVM内存模型那几个区域会发生OOM？

1. 堆区
2. 方法区
3. 栈区

# gc日志

方法区、虚拟机栈分析gc日志，堆区分析dump文件

## 方法区

JDK8用的是元空间，使用Parallel回收器来进行垃圾回收，新生代使用Parallel Scavenge垃圾回收器，老年代使用的是Parallel Old垃圾回收器

1、模拟OOM，上代码

第23行的cache如何理解？设置为true、false结果会有何不同？

根据类的全限定名做了缓存，缓存到instanceKlass

import net.sf.cglib.proxy.Enhancer;

import net.sf.cglib.proxy.MethodInterceptor;

import net.sf.cglib.proxy.MethodProxy;

import java.lang.reflect.Method;

public class MetaspaceOverFlowTest {

/\*\*

\* 通过CGLIB模拟向元空间写入数据，CGLIB可以直接操作字节码解析器

\*/

public static void main(String[] args) {

while (true) {

try {

Thread.sleep(10);

} catch (InterruptedException e) {

e.printStackTrace();

}

Enhancer enhancer = new Enhancer();

enhancer.setSuperclass(MetaspaceOverFlowTest.class);

enhancer.setUseCache(false);

enhancer.setCallback(new MethodInterceptor() {

public Object intercept(Object obj, Method method, Object[] args, MethodProxy proxy) throws Throwable {

return proxy.invokeSuper(obj, args);

}

});

System.out.println("running...");

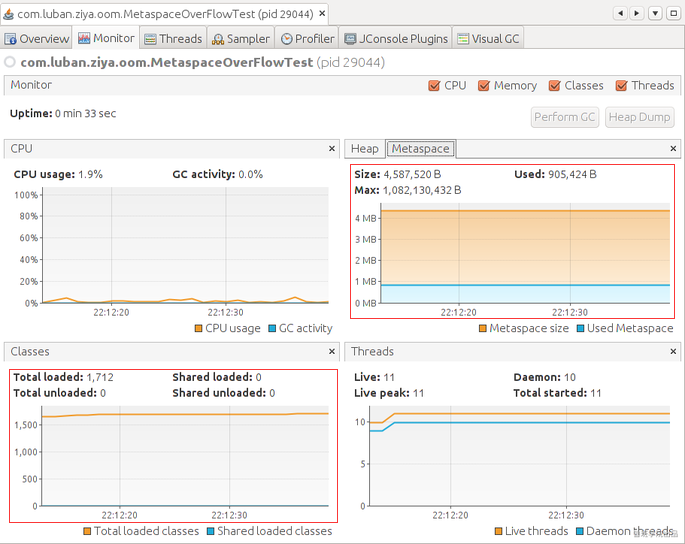
enhancer.create();

}

}

}

2、如何查看（visualVM、arthas）

主要看元空间和类信息

3、gc日志

[GC (Metadata GC Threshold) [PSYoungGen: 45765K->3280K(86528K)] 45781K->3304K(193024K), 0.0367573 secs] [Times: user=0.09 sys=0.00, real=0.04 secs]

[Full GC (Metadata GC Threshold) [PSYoungGen: 3280K->0K(86528K)] [ParOldGen: 24K->3222K(72704K)] 3304K->3222K(159232K), [Metaspace: 9726K->9726K(1058816K)], 0.2736778 secs] [Times: user=0.54 sys=0.02, real=0.27 secs]

4、调优参数

-XX:MetaspaceSize=10m

-XX:MaxMetaspaceSize=10m

5、调优原则

1、最大、最小设置成一样大

2、程序运行起来后，通过visualVM、arthas查看占用了多少内存，向上调优，预留20%以上的空间

## 堆区

1、模拟OOM，上代码

import java.util.ArrayList;

import java.util.List;

public class HeapOverFlowTest1 {

int[] intArr = new int[10];

public static void main(String[] args) {

List<HeapOverFlowTest1> objs = new ArrayList<>();

for (;;) {

try {

Thread.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

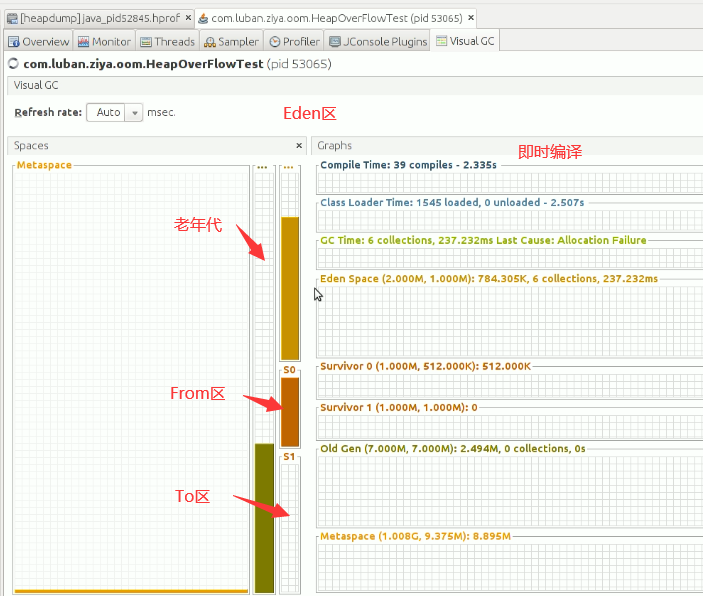
objs.add(new HeapOverFlowTest1());

}

}

}

2、查看（visualVM、arthas）



3、gc日志

[GC (Allocation Failure) [PSYoungGen: 1344K->320K(2048K)] 7894K->7118K(9216K), 0.0071516 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]

[GC类型 (GC原因) [新生代垃圾收集器: gc前新生代的内存使用情况->gc后新生代的内存使用情况(新生代总内存)] gc前堆内存的使用情况->gc后堆内存的使用情况(堆总内存), gc耗时] [Times: gc阶段用户空间耗时 gc阶段内核空间耗时, gc阶段实际耗时]

[Full GC (Ergonomics) [PSYoungGen: 320K->0K(2048K)] [ParOldGen: 6798K->5930K(7168K)] 7118K->5930K(9216K), [Metaspace: 9296K->9233K(1058816K)], 0.6733958 secs] [Times: user=1.76 sys=0.00, real=0.68 secs]

[GC类型 (GC原因) [新生代垃圾收集器: gc前新生代的内存使用情况->gc后新生代的内存使用情况(新生代总内存)] [老年代垃圾收集器: gc前老年代的内存使用情况->gc后老年代的内存使用情况(新生代总内存)] gc前堆内存的使用情况->gc后堆内存的使用情况(堆总内存), [Metaspace: gc前元空间的内存使用情况->gc后元空间的内存使用情况(元空间总内存)], gc耗时] [Times: gc阶段用户空间耗时 gc阶段内核空间耗时, gc阶段实际耗时]

4、调优参数

-Xms10m -Xmx10m

5、调优原则

（1）预留30%以上的空间

（2）周期性看日志，重点关注full gc频率

6、内存溢出错误

gc overhead limit exceeded：频繁full gc导致的

7、堆内存溢出时，保存内存快照，自动生成dump文件

-XX:+HeapDumpOnOutOfMemoryError

8、dump文件分析工具

Visualvm

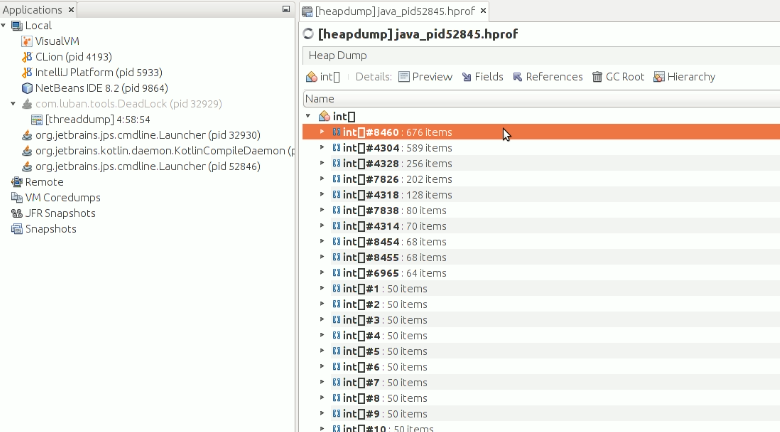
PerfMa的Xelephant：网站https://www.perfma.com/

这2个工具都是Java Agent实现

Visualvm使用：

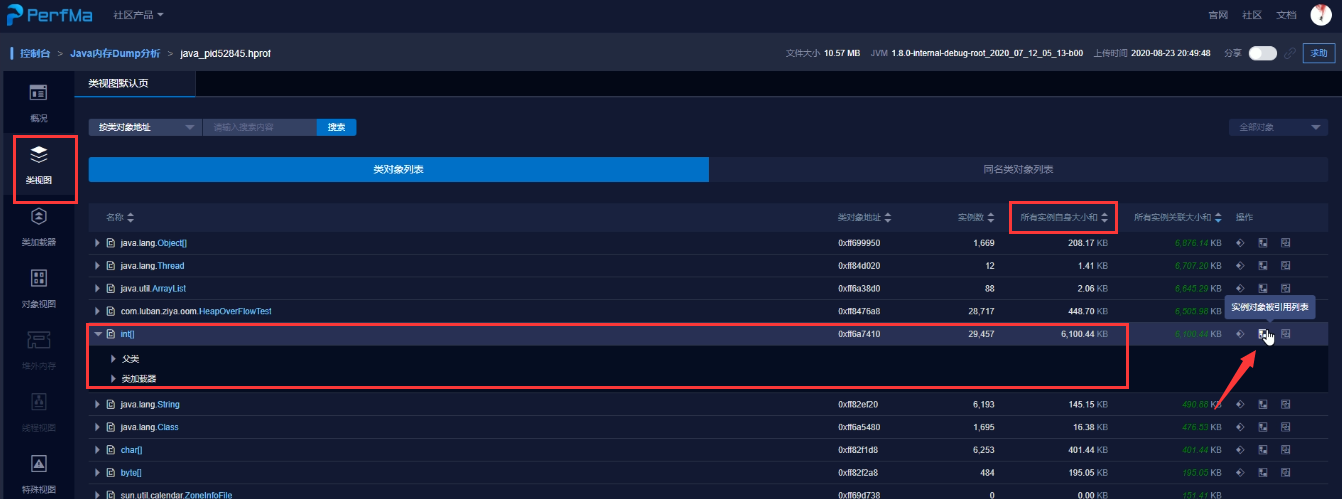
1. File – Load 加载dump文件
2. Classes by Size of instances找到占用空间最大实例
3. 分析被引用列表

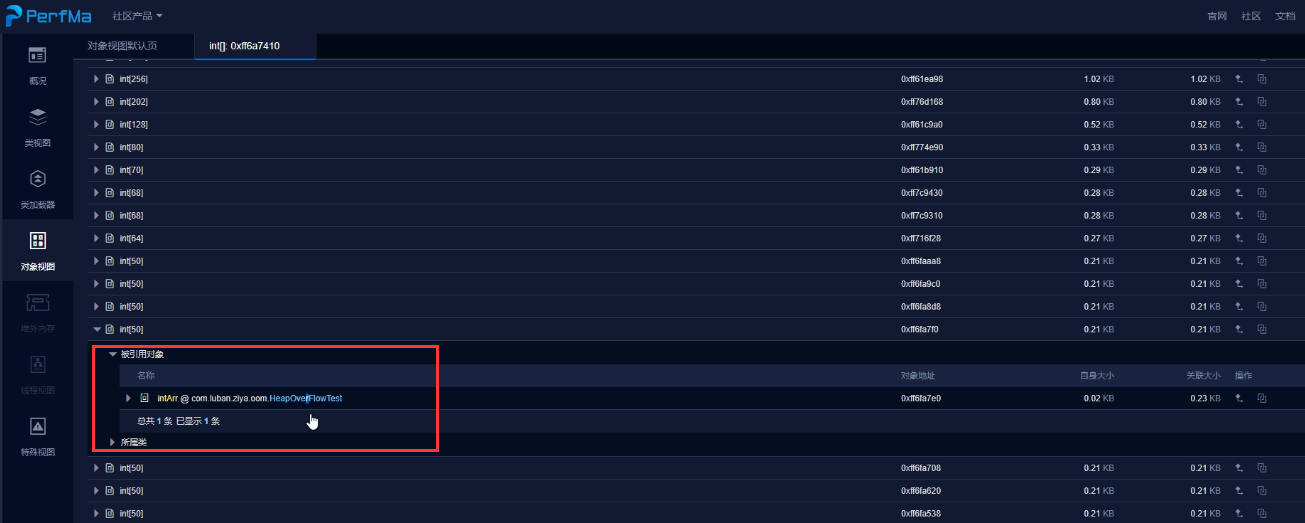




Xelephant使用：

1. 上传本地dump文件，自动生成可视化界面
2. 分析类视图，找到占内存最大的实例
3. 点击实例对象被引用列表
4. 找到用户引用对象的类。注意：有些是用户的，有些是系统的





## 虚拟机栈

1、模拟OOM，上代码

public class StackOverFlowTest {

private int val = 0;

public void test() {

val++;

test();

}

public static void main(String[] args) {

StackOverFlowTest test = new StackOverFlowTest();

try {

test.test();

} catch (Throwable t) {

t.printStackTrace();

System.out.println(test.val);

}

}

}

2、调优参数

-Xmss200k

栈大小相同，栈深度不同，为什么？

# 调优工具

Jps、Jstack、Jstat、Jmap、Jinfo、visualVM、arthas

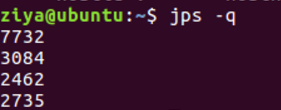
主要是visualVM、arthas，是上面工具的综合体

## Jps

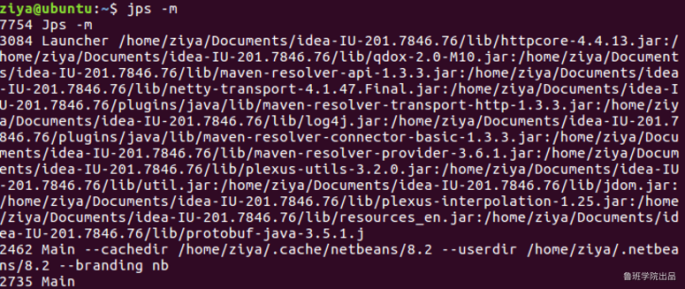
只输出java进程，该工具通过内存映射实现的，文件内容与内存内容50ms同步一次

### 命令

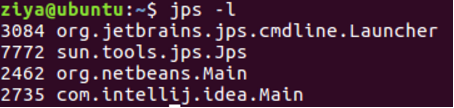
-q：只显示Java进程的ID



-m：输出Java进程的ID + main函数所在类的名词 + 传递给main函数的参数

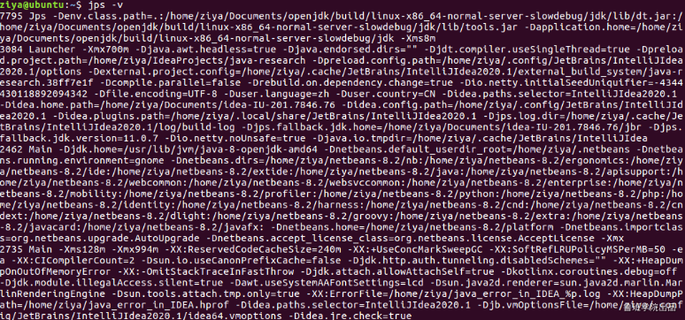


-l：输出Java进程的ID + main函数所在类的全限定名（包名 + 类名）



-v：输出Java进程的ID + main函数所在类的名称 + 传递给JVM的参数

应用：可通过此方式快速查看JVM参数是否设置成功



-V、hostid基本用不到，这里就不做介绍了，感兴趣的同学可以自行百度学习。

### 源码在哪

\openjdk\jdk\src\share\classes\sun\tools\jps\

纯Java编写的

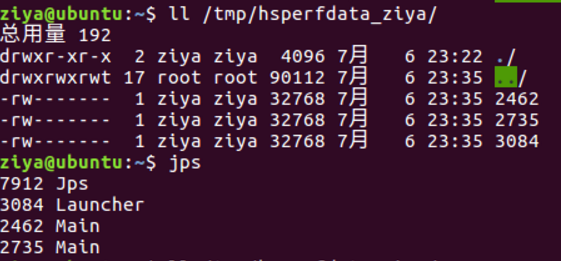
### 如何识别Java进程

jps输出的信息全是Java进程的信息，是如何做到的？

Java进程在创建的时候，会生成相应的文件，进程相关的信息会写入该文件中。

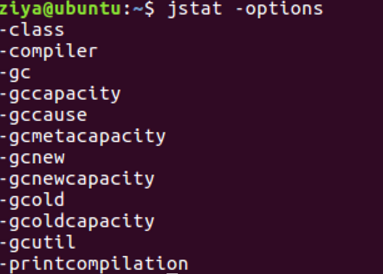
Windows下默认理解是C:\Users\username\AppData\Local\Temp\hsperfdata\_username，

Linux下默认路径是/tmp/hsperfdata\_username



## jstate

Hotspot自带的工具，通过该工具可实时了解某个进程的class、compile、gc、memory的相关信息。具体可通过该工具查看哪些信息可通过jstat -options查看



为什么说是实时呢，因为底层实现是mmap，及内存映射文件

### jstat输出的这些值从哪来的

PerfData文件

Windows下默认理解是C:\Users\username\AppData\Local\Temp\hsperfdata\_username

Linux下默认路径是/tmp/hsperfdata\_username

### PerfData文件

**1、文件创建**

取决于两个参数

-XX:-/+UsePerfData

默认是开启的

关闭方式：-XX:-UsePerfData。如果关闭了，就不会创建PerfData文件

-XX:-/+PerfDisableSharedMem（禁用共享内存）

默认是关闭的，即支持内存共享。如果禁用了，依赖于PerfData文件的工具就无法正常工作了

**2、文件删除**

默认情况下随Java进程的结束而销毁

**3、文件更新**

-XX:PerfDataSamplingInterval = 50ms

即内存与PerfData文件的数据延迟为50ms

### 源码在哪

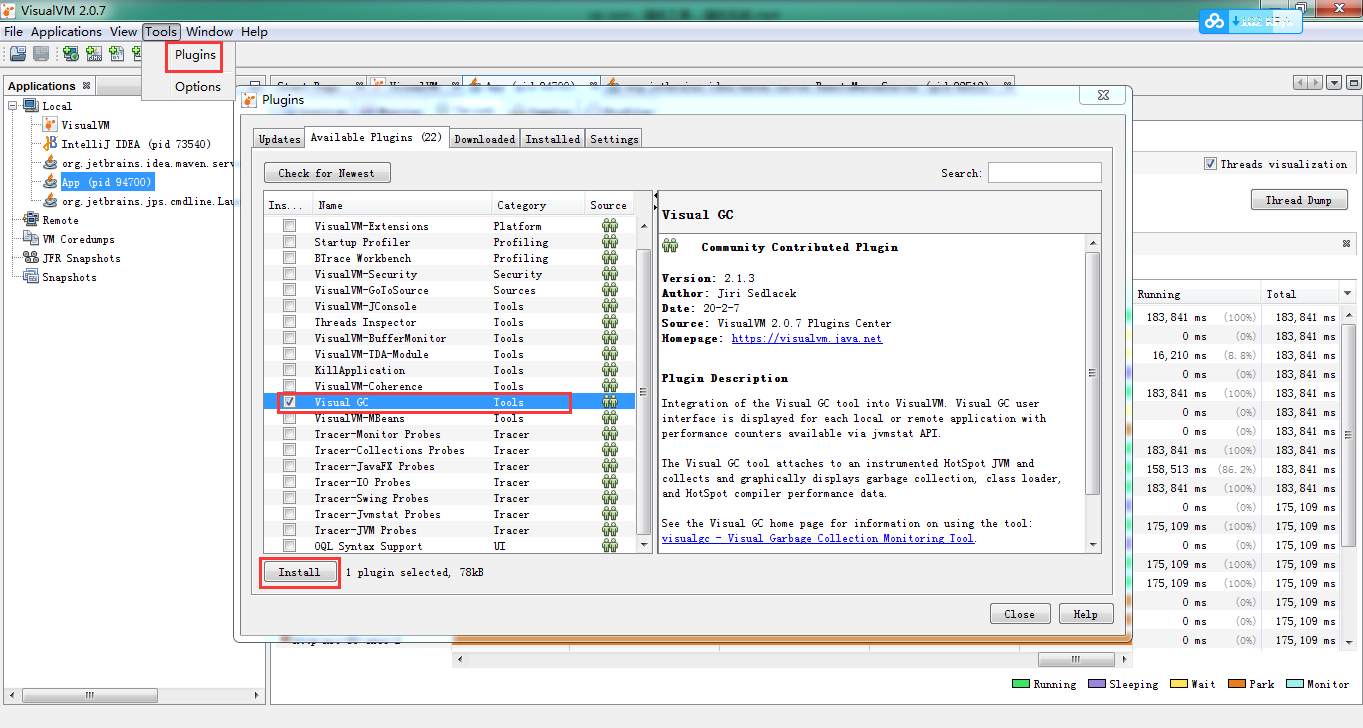
纯Java编写

\openjdk\jdk\src\share\classes\sun\tools\jstat\Jstat.java

## VisualVM



VisualGC插件下载



# Java Agent

Java Agent是一个技术，也就是一个jar包，只是启动方式和普通Jar包有所不同，对于普通的Jar包，通过指定类的main函数进行启动，但是Java Agent并不能单独启动，必须依附在一个Java应用程序运行，有点像寄生虫的感觉，它的底层用C++的JVMTI Agent实现的。

参考文章：

https://blog.csdn.net/qq\_43171869/article/details/83477163

<https://www.jianshu.com/p/f5efc53ced5d>

# 案例

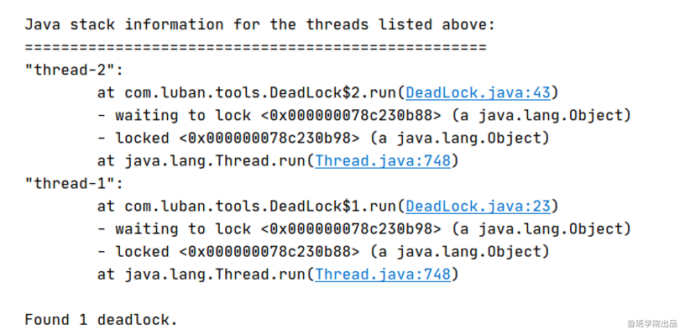
## 1、统计线程数

jstack -l 6972 | grep 'java.lang.Thread.State' | wc -l

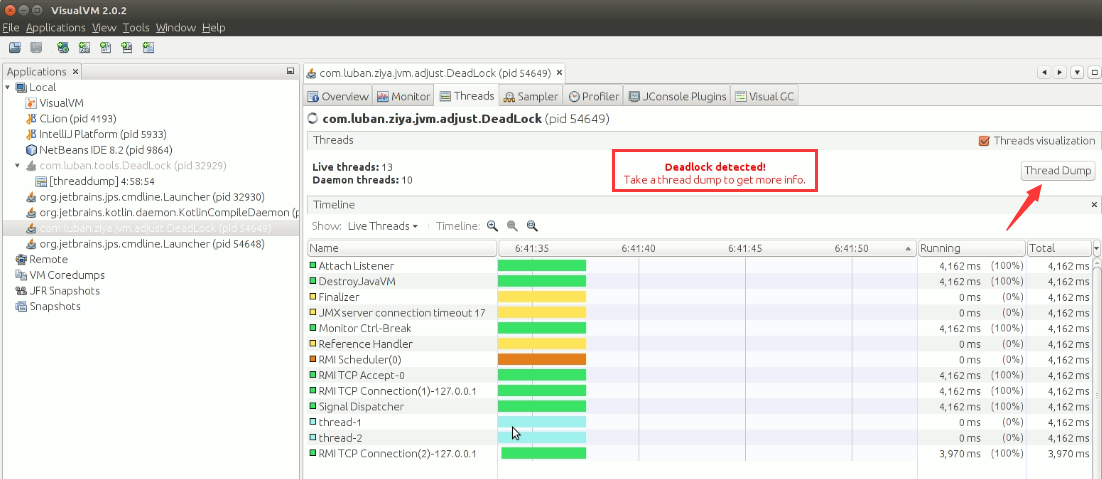
## 2、检测死锁

可使用jstack、jconsle、visualVM

（1）Jstack使用检测死锁



（2）visualVM使用检测死锁

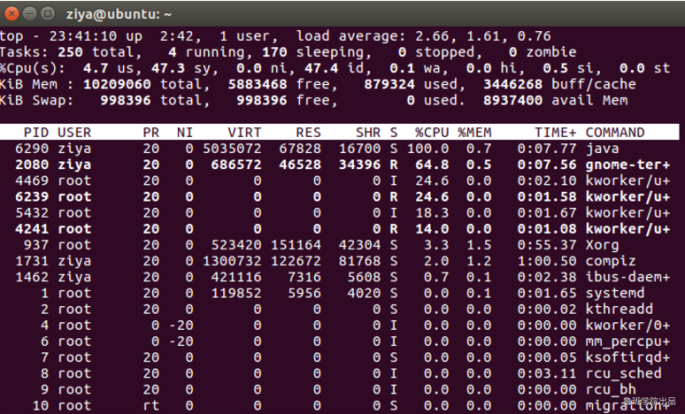


![C:\Users\Administrator\AppData\Roaming\Tencent\Users\243900853\QQ\WinTemp\RichOle\L~](K7I)%FJU095VRY%]BP1.png](data:image/png;base64,)

## 3、CPU占用过高

定位到占用CPU最高的进程：

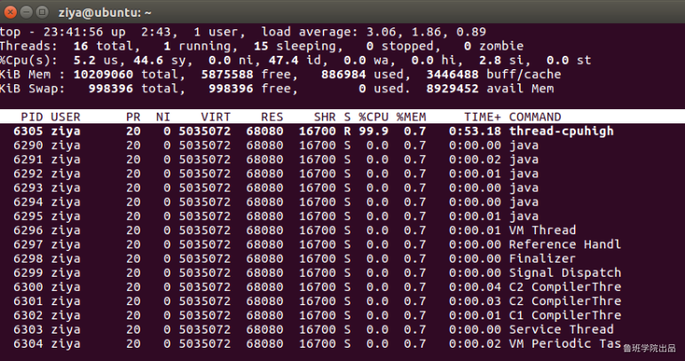
top 找到最高的进程，jps -l 找到进程对应的java全路



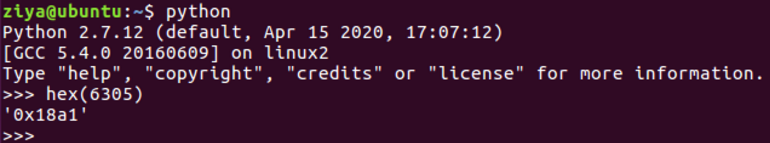
定位到目前占用CPU最高的线程ID

top -H -p 6290

通过COMMAND可以找到线程名称thread-cpuhigh



线程ID由十进制转成十六进制，用Python



定位线程

jstack 6290（进程ID）|grep 18a1（十六进制线程ID） -A 30

