

点赞再看，养成习惯，微信搜索【三太子敖丙】关注这个互联网苟且偷生的工具人。

本文 **GitHub** <https://github.com/JavaFamily> 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

上次给老公们说过了死循环cpu飙高的排查过程，今天就带着老公们看看堆内存溢出我们一般怎么排查的。

■ cpu100%排查文章

在排查之前，我想jvm的基础知识大家应该都是了解了的吧？

老婆我就是不了解，人家要你说给我听。

行行行，诶真实拿你们没办法，那我就带大家回温一下JVM的内存模型（这玩意跟JAVA内存模型JMM可不一样，不要记错了）

今天我就直说堆，因为溢出是发生在堆中的。

JVM堆内存被分为两部分：年轻代（Young Generation）和老年代（Old Generation）。

年轻代

年轻代是所有新对象产生的地方。当年轻代内存空间被用完时，就会触发垃圾回收。这个垃圾回收叫做Minor GC。年轻代被分为3个部分——Eden区和两个Survivor区。

年轻代空间的要点：

1. 大多数新建的对象都位于Eden区。
2. 当Eden区被对象填满时，就会执行Minor GC，并把所有存活下来的对象转移到其中一个survivor区。
3. Minor GC同样会检查存活下来的对象，并把它们转移到另一个survivor区。这样在一段时间内，总会有一个空的survivor区。
4. 经过多次GC周期后，仍然存活下来的对象会被转移到老年代内存空间，通常这是在年轻代有资格提升到老年代前通过设定年龄阈值来完成的。

老年代

老年代内存里包含了长期存活的对象和经过多次Minor GC后依然存活下来的对象，通常会在老年代内存被占满时进行垃圾回收。

GC种类

Major GC

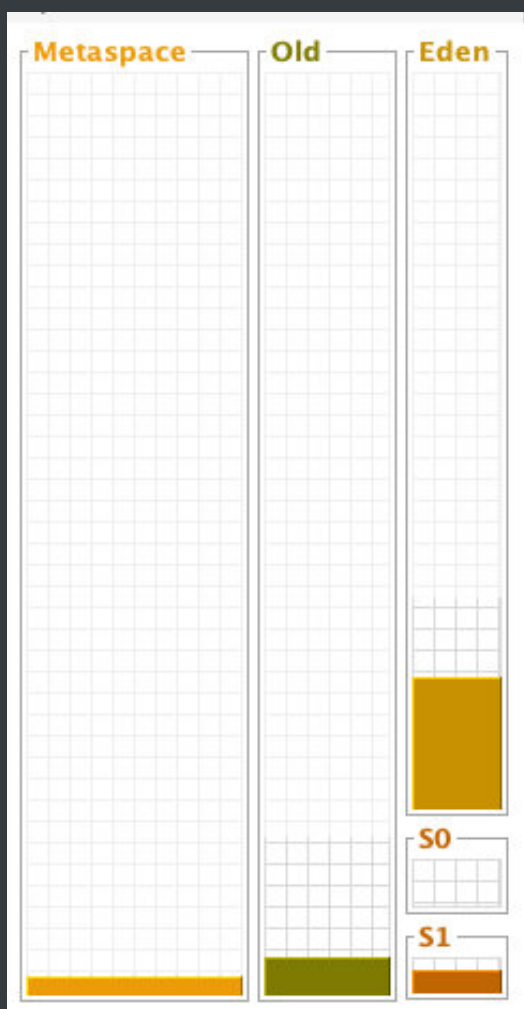
老年代的垃圾收集叫做Major GC，Major GC通常是跟full GC是等价的，收集整个GC堆。

分代GC

1. Young GC：只收集年轻代的GC
2. Old GC：只收集老年代的GC(只有CMS的concurrent collection是这个模式)
3. Mixed GC：收集整个young gen以及部分old gen的GC(只有G1有这个模式)

Full GC

Full GC定义是相对明确的，就是针对整个新生代、老年代、元空间（metaspace，java8以上版本取代perm gen）的全局范围的GC。



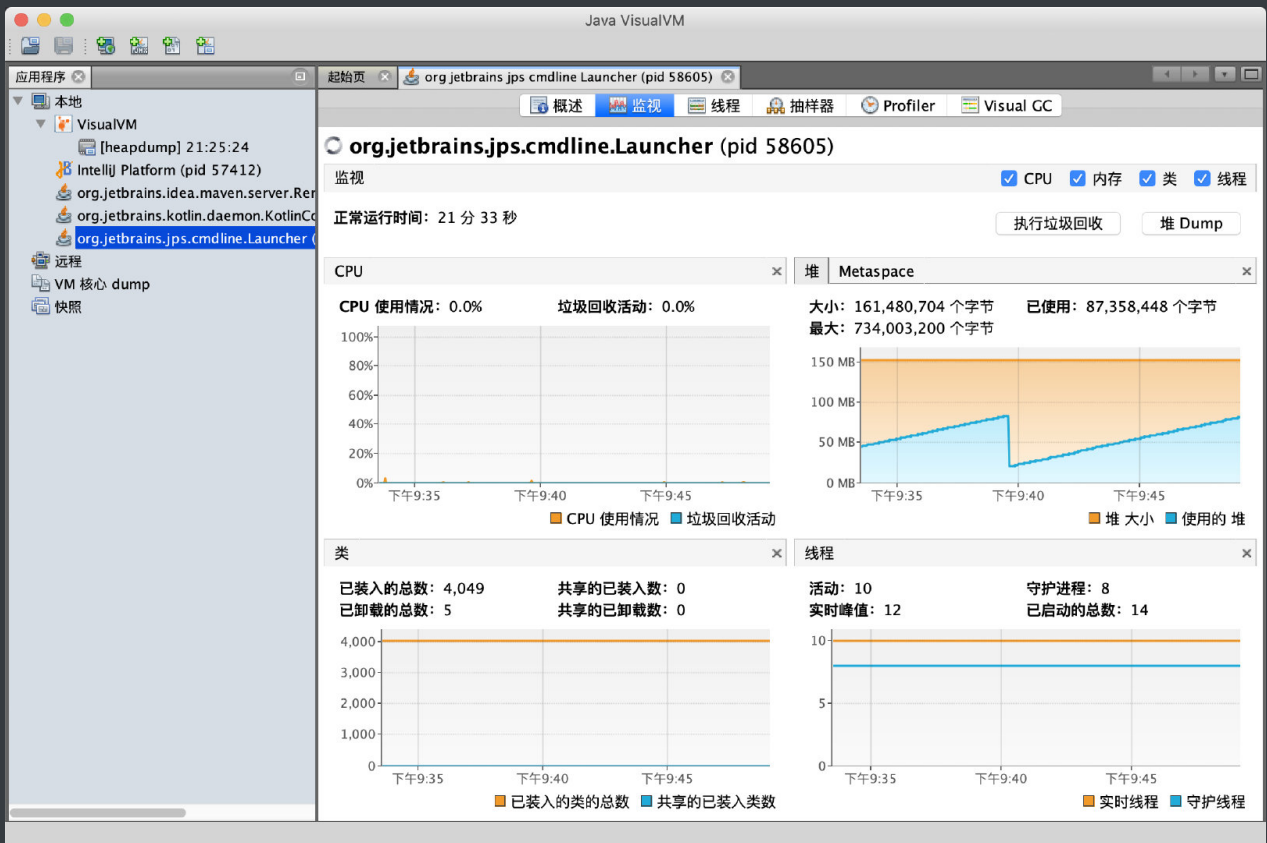
老公们可以从上图看到年轻代分为了一个Eden区和两个survivor区（S1，S2），survivor区同一时间只会有一个满一个空，交替的。

然后就是GC到一定的阈值到老年代，今天不讲永久代所以忽略Mataspace。

老婆：那怎么分析呢？

今天我就用一个JDK自带的工具jvisualvm来给大家演示一波怎么操作的，因为这玩意谁都有，你去命令行敲一下jvisualvm就出来了（Mac是这样的，不知道Windows是怎么样子的）。

操作界面：



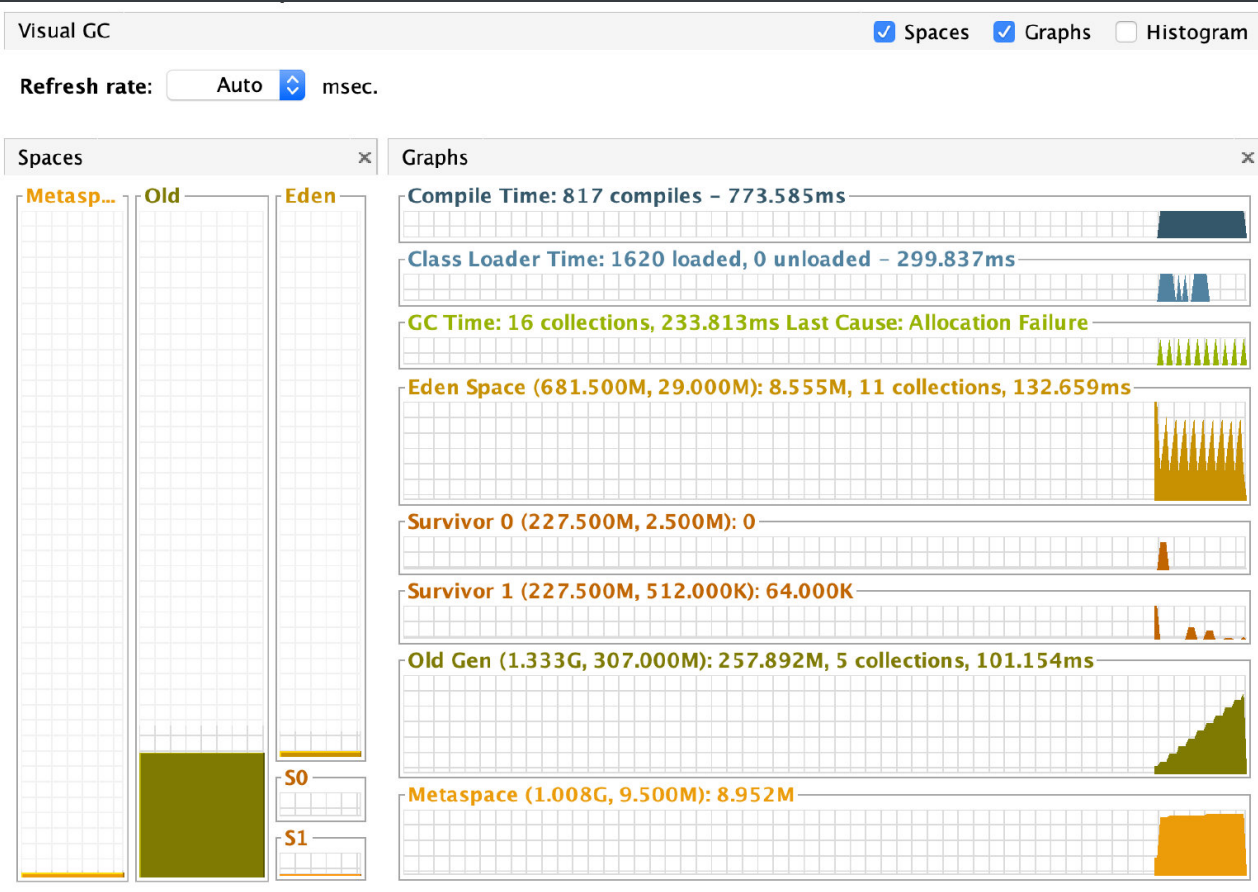
一般什么情况可能是出现了溢出呢？

超时，不进行服务，服务挂掉，接口不在服务这样的异常问题。

那模拟也很简单，我写个循环一直往List丢数据，不使用list就能看到现象了

```
private static final Integer K = 1024;|
public static void main(String[] args) {
    int size=K * K *8;
    List<byte[] > list =new ArrayList<byte[]>();
    for (int i=0;i<K;i++){
        System.out.println("JVM 写入数据"+(i+1)+"M");
        try {
            Thread.sleep( millis: 1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        list.add(new byte[size]);
    }
}
```

老公们可以看到图形化界面还是很清晰明了的，这个是Visual GC的插件



大家点击菜单栏的插件，然后安装就好了，安装完了记得点击激活。

更新可用插件 (17)已下载已安装 (1)设置

检查最新版本(F)

安装	名称	类别	源
<input checked="" type="checkbox"/>	VisualVM-Glassfish	Application Se...	社区
<input type="checkbox"/>	VisualVM-Extensions	Platform	社区
<input type="checkbox"/>	Startup Profiler	Profiling	社区
<input type="checkbox"/>	BTrace Workbench	Profiling	社区
<input type="checkbox"/>	VisualVM-Security	Security	社区
<input type="checkbox"/>	VisualVM-BufferMonitor	Tools	社区
<input type="checkbox"/>	Threads Inspector	Tools	社区
<input type="checkbox"/>	VisualVM-JConsole	Tools	社区
<input type="checkbox"/>	VisualVM-MBeans	Tools	社区
<input type="checkbox"/>	KillApplication	Tools	社区
<input type="checkbox"/>	Tracer-jvmstat Probes	Tracer	社区
<input type="checkbox"/>	Tracer-Monitor Probes	Tracer	社区
<input type="checkbox"/>	Tracer-Swing Probes	Tracer	社区
<input type="checkbox"/>	Tracer-IO Probes	Tracer	社区
<input type="checkbox"/>	Tracer-Collections Probes	Tracer	社区
<input type="checkbox"/>	Tracer-JVM Probes	Tracer	社区
<input type="checkbox"/>	OQL Syntax Support	UI	社区

安装(I)

VisualVM-Glassfish

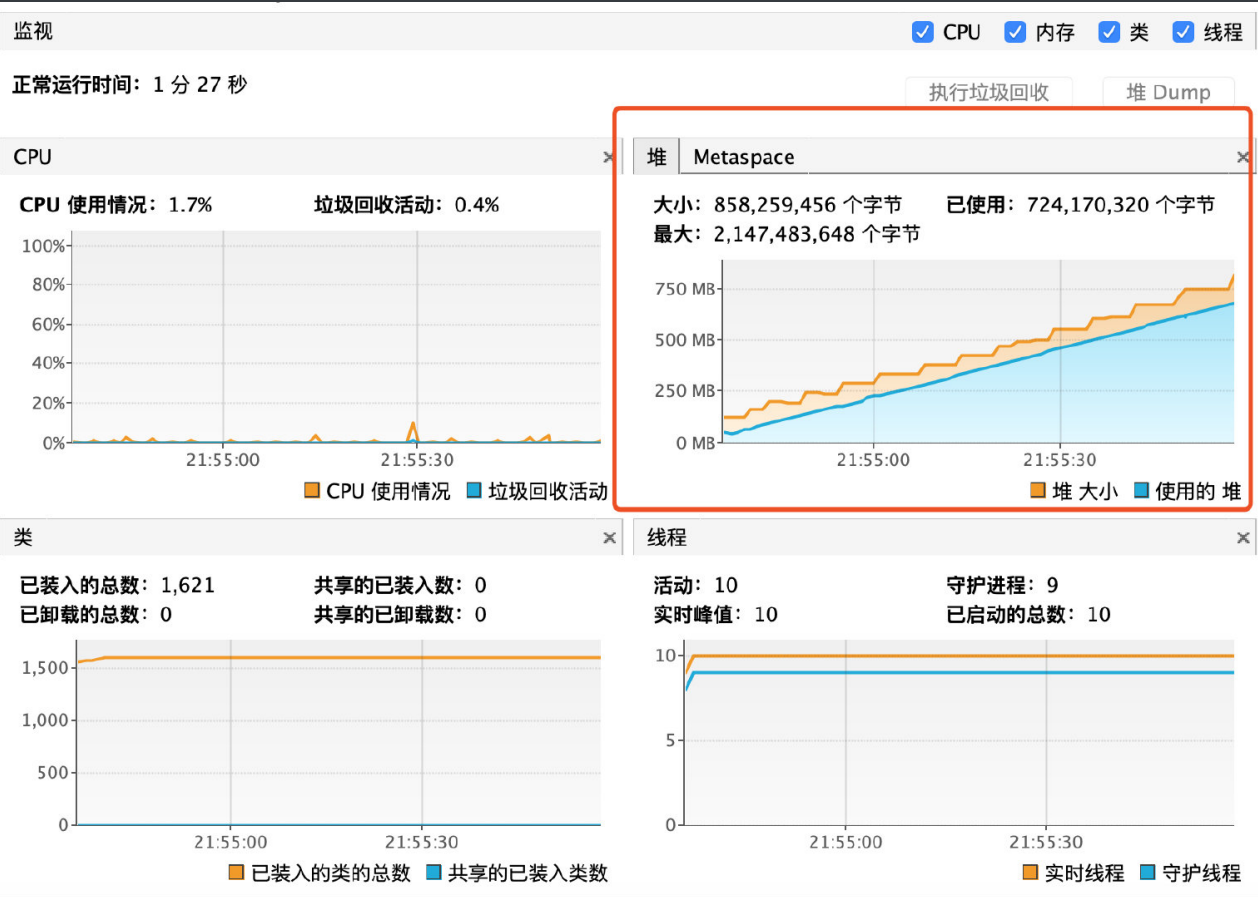
社区提供的插件

版本: 1.5
作者: Jaroslav Bachorik
日期: 16-11-7
源: Java VisualVM 插件中心
主页: <https://visualvm.github.io>

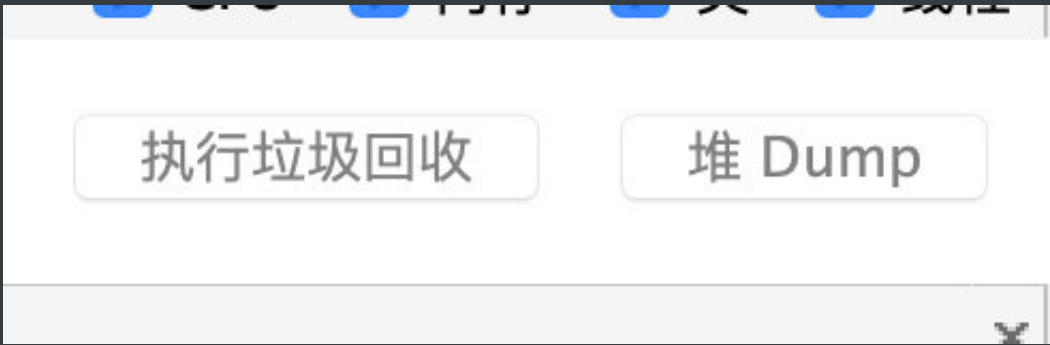
插件描述

A sample plugin giving an overview of advance of VisualVM. Enhances monitoring of GlassFish adding specialized overview, new tab for monitoring the ability to visually select and monitor any o applications

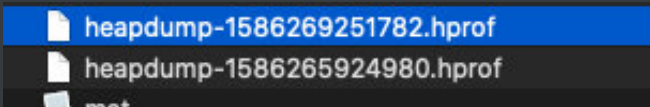
可以看到不释放，堆空间就一直上去，直到OOM（out of memory）



这个时候我们就dump下来堆信息看看

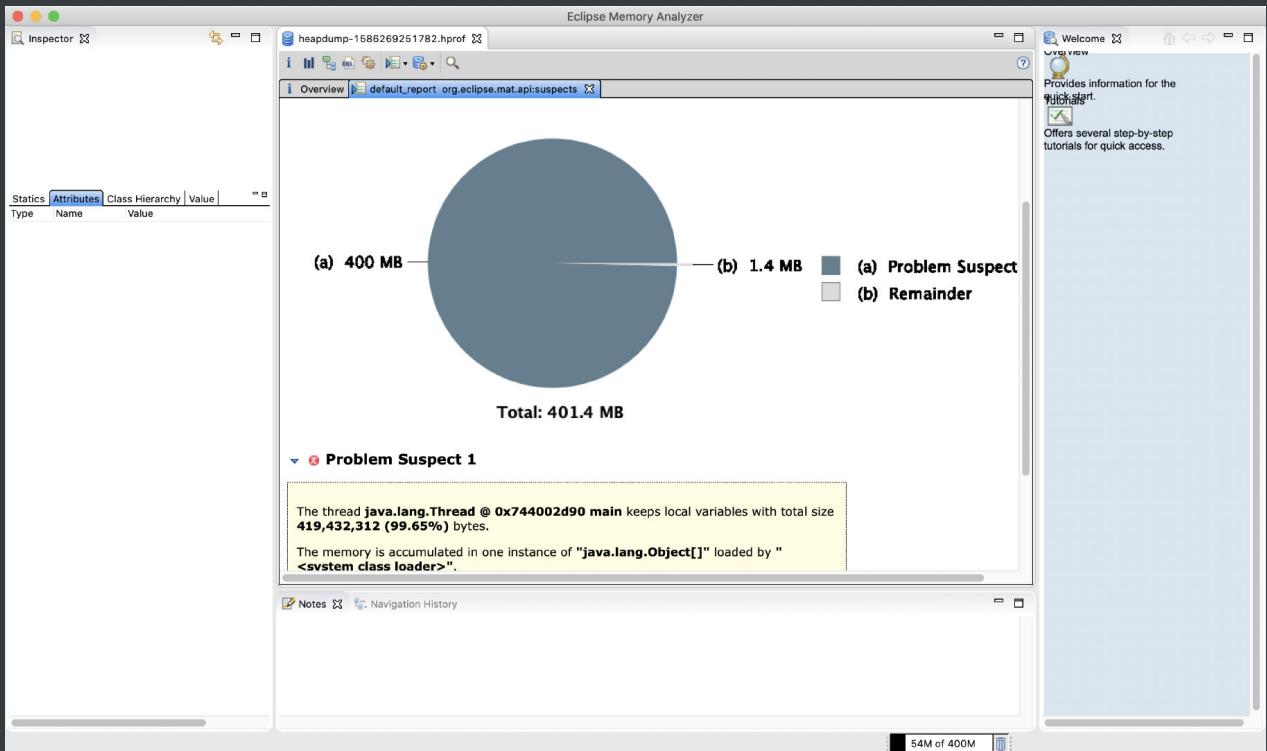


会dump出一个这样的hprof快照文件，可以用jvisualvm本身的系统去分析，我这里推荐MAT吧，因为我习惯这个了。



MAT：[下载地址](#)

下来好了我们可以看到mat已经分析了我们的文件



你看他就是个暖男，都帮我们分析出来了一个问题，我们点进去看看

Total: 401.4 MB

Problem Suspect 1

The thread `java.lang.Thread @ 0x744002d90 main` keeps local variables with total size **419,432,312 (99.65%)** bytes.

The memory is accumulated in one instance of `"java.lang.Object[]"` loaded by `"<system class loader>"`.

The stacktrace of this Thread is available. [See stacktrace.](#)

Keywords
`java.lang.Object[]`
[Details >](#)

[Table Of Contents](#) Created by **Eclipse Memory Analyzer**

他发现了是ArrayList的问题了，我们再往下看

The thread **java.lang.Thread @ 0x744002d90 main** keeps local variables with total size **419,432,312 (99.65%)** bytes.

The memory is accumulated in one instance of "**java.lang.Object[]**" loaded by "**<system class loader>**".

The stacktrace of this Thread is available. [See stacktrace.](#)

Keywords

java.lang.Object[]

▼ Shortest Paths To the Accumulation Point


Class Name	Shallow Heap	Retained Heap
 java.lang.Object[73] @ 0x7591e4090		
 elementData java.util.ArrayList @ 0x7440147f0	24	419,431,536
 <Java Local> java.lang.Thread @ 0x744002d90 main JNI Global, Thread	120	419,432,312

就是我们刚才没释放的ArrayList

▼ Accumulated Objects in Dominator Tree

看到了嘛，具体代码的位置都帮我们定位好了，那排查也就是手到擒来的事情了。

▼ Thread Properties

Object / Stack Frame	java.lang.Thread @ 0x744002d90
Name	main
Shallow Heap	120
Retained Heap	419,432,312
Context Class Loader	sun.misc.Launcher\$AppClassLoader @ 0x744003ff8
Is Daemon	false
 Total: 6 entries	

堆大小

▼ Thread Stack

```
main
  at java.lang.Thread.sleep(J)V (Native Method)
  at redis.RedisLock.main([Ljava/lang/String;)V (RedisLock.java:20)
```

这里给到代码位置了

延伸点

上面我们使用工具jump了，那怎么去服务器上jump呢？

```
jmap -dump:format=b,file=<dumpfile.hprof> <pid>
```

有老公可能问了，不是所有的故障当时我们都在场的，无法及时jump，那也简单

```
-XX:+HeapDumpOnOutOfMemoryError
```

配置这玩意之后，oom的时候会自动jump的，到时候拿快照分析一波就好了。

MAT的功能还有很多的，百度谷歌太多工具文了，我就不做重复的工作了，比如还可以排查对象的强弱引用，还可以查看引用链等等。

还有个只写这么点的原因是有点晚了，顶不住了，最近不拍视频也是因为事情多了，有点小忙，希望老公们体谅，对了Redis的分布式锁已经在安排的路上了。

我是敖丙，一个在互联网苟且偷生的工具人。

最好的关系是互相成就，老公们的「三连」就是丙丙创作的最大动力，我们下期见！

注：如果本篇博客有任何错误和建议，欢迎老公们留言，老公你快说句话啊！

文章持续更新，可以微信搜索「三太子敖丙」第一时间阅读，回复【资料】【面试】【简历】有我准备的一线大厂面试资料和简历模板，本文 **GitHub** <https://github.com/JavaFamily> 已经收录，有大厂面试完整考点，欢迎Star。



你知道的越多，你不知道的越多