Jmap

此命令可以用来查看内存信息。

实例个数以及占用内存大小

```
C:\Users\39497>jmap -histo 13988 > ./log.txt
```

打开log.txt，文件内容如下：

```
 num     #instances         #bytes  class name
----------------------------------------------
   1:       6027049      190824296  [Ljava.lang.Object;
   2:       2481762       99270480  java.util.TreeMap$Entry
   3:       2409493       77103776  java.io.ObjectStreamClass$WeakClassKey
   4:         31998       43117528  [I
   5:        350162       35275656  [C
   6:        409782       19669536  java.util.TreeMap
   7:        440067       14082144  java.util.TreeMap$KeyIterator
   8:        331461       10606752  java.lang.StackTraceElement
   9:        422453       10138872  java.io.SerialCallbackContext
  10:        409605        9830520  javax.management.openmbean.CompositeDataSupport
  11:        305940        7342560  java.lang.Long
  12:         40361        7288320  [B
  13:        427757        6844112  java.lang.Boolean
  14:        409733        6555728  java.util.TreeMap$EntrySet
  15:        409613        6553808  java.util.TreeMap$KeySet
  16:        363097        5809552  java.lang.Integer
  17:        216844        5204256  java.lang.String
  18:         47530        4562880  java.lang.management.ThreadInfo
  19:         86963        4174224  java.util.HashMap
  20:        143948        3966488  [Ljavax.management.openmbean.CompositeData;
  21:         35537        2285448  [Ljava.util.Hashtable$Entry;
  22:         70653        2260896  java.util.Vector
  23:         47533        2228296  [Ljava.lang.StackTraceElement;
  24:         18113        1883752  java.io.ObjectStreamClass
  25:         35515        1704720  java.util.Hashtable
  26:         64980        1559520  java.lang.StringBuilder
  27:         36769        1470760  java.security.ProtectionDomain
  28:         10312        1143696  java.lang.Class
  29:         35399        1132768  java.security.CodeSource
  30:         16871        1046856  [Ljava.util.HashMap$Node;
  31:         31493        1007776  java.util.concurrent.ConcurrentHashMap$Node
  32:         11725         844200  javax.management.remote.rmi.RMIConnectionImpl$CombinedClassLoader
  33:         11725         844200  javax.management.remote.rmi.RMIConnectionImpl$CombinedClassLoader$ClassLoaderWrapper
  34:         11724         844128  com.sun.jmx.remote.util.OrderClassLoaders
  35:          8650         761200  java.lang.reflect.Method
```

- num：序号
- instances：实例数量
- bytes：占用空间大小
- class name：类名称，[C is a char[]，[S is a short[]，[I is a int[]，[B is a byte[]，[[I is a int[][]

堆信息

```
C:\Users\39497>jmap -heap 13988
Attaching to process ID 13988, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.45-b02

using thread-local object allocation.
Parallel GC with 8 thread(s)

Heap Configuration:
   MinHeapFreeRatio         = 0
   MaxHeapFreeRatio         = 100
   MaxHeapSize              = 2118123520 (2020.0MB)
   NewSize                  = 44564480 (42.5MB)
   MaxNewSize               = 705691648 (673.0MB)
   OldSize                  = 89653248 (85.5MB)
   NewRatio                 = 2
   SurvivorRatio            = 8
   MetaspaceSize            = 21807104 (20.796875MB)
   CompressedClassSpaceSize = 1073741824 (1024.0MB)
   MaxMetaspaceSize         = 17592186044415 MB
   G1HeapRegionSize         = 0 (0.0MB)

Heap Usage:
PS Young Generation
Eden Space:
   capacity = 632291328 (603.0MB)
   used     = 380963288 (363.31490325927734MB)
   free     = 251328040 (239.68509674072266MB)
   60.25122773785694% used
From Space:
   capacity = 20971520 (20.0MB)
   used     = 20971520 (20.0MB)
   free     = 0 (0.0MB)
   100.0% used
To Space:
   capacity = 24641536 (23.5MB)
   used     = 0 (0.0MB)
   free     = 24641536 (23.5MB)
   0.0% used
PS Old Generation
   capacity = 191889408 (183.0MB)
   used     = 21845248 (20.833251953125MB)
   free     = 170044160 (162.166748046875MB)
   11.384290684767759% used

20372 interned Strings occupying 2651120 bytes.
```

堆内存dump

```
C:\Users\39497>jmap -dump:format=b,file=eureka.hprof 13988
Dumping heap to C:\Users\39497\eureka.hprof ...
Heap dump file created
```

也可以设置内存溢出自动导出dump文件(内存很大的时候，可能会导不出来)

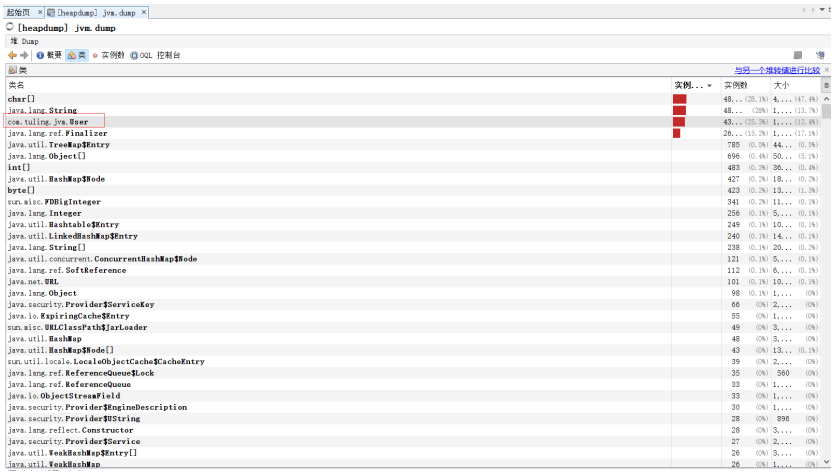
1. -XX:+HeapDumpOnOutOfMemoryError
2. -XX:HeapDumpPath=./ (路径)

示例代码：

```java
public class OOMTest {

 public static List<Object> list = new ArrayList<>();

 // JVM设置
 // -Xms10M -Xmx10M -XX:+PrintGCDetails -XX:+HeapDumpOnOutOfMemoryError -XX:HeapDumpPath=D:\jvm.dump
 public static void main(String[] args) {
 List<Object> list = new ArrayList<>();
 int i = 0;
  int j = 0;
  while (true) {
  list.add(new User(i++, UUID.randomUUID().toString()));
```

```
13   new User(j--, UUID.randomUUID().toString());
14   }
15   }
16 }
```

可以用jvisualvm命令工具导入该dump文件分析



Jstack

用jstack查找死锁，见如下示例

```
1  public class DeadLockTest {
2
3   private static Object lock1 = new Object();
4   private static Object lock2 = new Object();
5
6   public static void main(String[] args) {
7   new Thread(() -> {
8   synchronized (lock1) {
9   try {
10   System.out.println("thread1 begin");
11   Thread.sleep(5000);
12   } catch (InterruptedException e) {
13   }
14   synchronized (lock2) {
15   System.out.println("thread1 end");
16   }
17   }
18   }).start();
19
20   new Thread(() -> {
21   synchronized (lock2) {
22   try {
23   System.out.println("thread2 begin");
24   Thread.sleep(5000);
25   } catch (InterruptedException e) {
26   }
27   synchronized (lock1) {
28   System.out.println("thread2 end");
29   }
30   }
31   }).start();
32
33   System.out.println("main thread end");
34   }
35 }
```

```
"Thread-1" #13 prio=5 os_prio=0 tid=0x000000001fa9e000 nid=0x2d64 waiting for monitor entry [0x000000002047f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at com.tuling.jvm.DeadLockTest.lambda$main$1(DeadLockTest.java:34)
        - waiting to lock <0x000000076b6ef868> (a java.lang.Object)
        - locked <0x000000076b6ef878> (a java.lang.Object)
        at com.tuling.jvm.DeadLockTest$$Lambda$2/1480010240.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:745)

"Thread-0" #12 prio=5 os_prio=0 tid=0x000000001fa99000 nid=0x3d94 waiting for monitor entry [0x000000002037f000]
   java.lang.Thread.State: BLOCKED (on object monitor)
        at com.tuling.jvm.DeadLockTest.lambda$main$0(DeadLockTest.java:21)
        - waiting to lock <0x000000076b6ef878> (a java.lang.Object)
        - locked <0x000000076b6ef868> (a java.lang.Object)
        at com.tuling.jvm.DeadLockTest$$Lambda$1/2074407503.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:745)
```

"Thread-1" 线程名

prio=5 优先级=5

tid=0x000000001fa9e000 线程id

nid=0x2d64 线程对应的本地线程标识nid

runnable 线程状态

```
Found one Java-level deadlock:
=============================
"Thread-1":
  waiting to lock monitor 0x000000000333a078 (object 0x000000076b6ef868, a java.lang.Object),
  which is held by "Thread-0"
"Thread-0":
  waiting to lock monitor 0x00000000033377e8 (object 0x000000076b6ef878, a java.lang.Object),
  which is held by "Thread-1"

Java stack information for the threads listed above:
===================================================
"Thread-1":
        at com.tuling.jvm.DeadLockTest.lambda$main$1(DeadLockTest.java:34)
        - waiting to lock <0x000000076b6ef868> (a java.lang.Object)
        - locked <0x000000076b6ef878> (a java.lang.Object)
        at com.tuling.jvm.DeadLockTest$$Lambda$2/1480010240.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:745)
"Thread-0":
        at com.tuling.jvm.DeadLockTest.lambda$main$0(DeadLockTest.java:21)
        - waiting to lock <0x000000076b6ef878> (a java.lang.Object)
        - locked <0x000000076b6ef868> (a java.lang.Object)
        at com.tuling.jvm.DeadLockTest$$Lambda$1/2074407503.run(Unknown Source)
        at java.lang.Thread.run(Thread.java:745)

Found 1 deadlock.
```

还可以用jvisualvm自动检测死锁

远程连接jvisualvm

启动普通的jar程序JMX端口配置：

java -Dcom.sun.management.jmxremote.port=8899 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false -jar foo.jar

tomcat的JMX配置

JAVA_OPTS=-Dcom.sun.management.jmxremote.port=8899 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false

jvisualvm远程连接服务需要在远程服务器上配置host(连接ip 主机名)，并且要关闭防火墙

jstack找出占用cpu最高的堆栈信息

1，使用命令top -p <pid> ，显示你的java进程的内存情况，pid是你的java进程号，比如4977

2，按H，获取每个线程的内存情况

3，找到内存和cpu占用最高的线程tid，比如4977

4，转为十六进制得到 0x1371 ,此为线程id的十六进制表示

5，执行 jstack 4977|grep -A 10 1371，得到线程堆栈信息中1371这个线程所在行的后面10行

6，查看对应的堆栈信息找出可能存在问题的代码

Jinfo

查看正在运行的Java应用程序的扩展参数

查看jvm的参数

```
C:\Users\39497>jinfo -flags 30880
Attaching to process ID 30880, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.45-b02
Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=134217728 -XX:MaxHeapSize=2118123520 -
9653248 -XX:+PrintGCDetails -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseFastUnorder
Command line:  -XX:+PrintGCDetails -Dfile.encoding=GBK
```

查看java系统参数

```
C:\Users\39497>jinfo -sysprops 30880
Attaching to process ID 30880, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.45-b02
java.runtime.name = Java(TM) SE Runtime Environment
java.vm.version = 25.45-b02
sun.boot.library.path = D:\dev\Java\jdk1.8.0_45\jre\bin
java.vendor.url = http://java.oracle.com/
java.vm.vendor = Oracle Corporation
path.separator = ;
java.rmi.server.randomIDs = true
file.encoding.pkg = sun.io
java.vm.name = Java HotSpot(TM) 64-Bit Server VM
sun.os.patch.level =
sun.java.launcher = SUN_STANDARD
user.script =
user.country = CN
user.dir = D:\workspace\JVM
java.vm.specification.name = Java Virtual Machine Specification
java.runtime.version = 1.8.0_45-b14
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment
```

Jstat

jstat命令可以查看堆内存各部分的使用量，以及加载类的数量。命令的格式如下：

jstat [-命令选项] [vmid] [间隔时间(毫秒)] [查询次数]

注意：使用的jdk版本是jdk8.

垃圾回收统计

jstat -gc pid 最常用，可以评估程序内存使用及GC压力整体情况

```
C:\Users\39497>jstat -gc 13988
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
8704.0 13312.0 2592.0  0.0    593408.0 545245.5  187392.0    21205.2   50088.0 48890.5 6568.0 6291.8     28    0.207   5      0.405    (
```

- S0C：第一个幸存区的大小
- S1C：第二个幸存区的大小
- S0U：第一个幸存区的使用大小
- S1U：第二个幸存区的使用大小
- EC：伊甸园区的大小
- EU：伊甸园区的使用大小
- OC：老年代大小
- OU：老年代使用大小
- MC：方法区大小(元空间)
- MU：方法区使用大小
- CCSC:压缩类空间大小
- CCSU:压缩类空间使用大小
- YGC：年轻代垃圾回收次数
- YGCT：年轻代垃圾回收消耗时间，单位s

- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间，单位s
- GCT：垃圾回收消耗总时间，单位s

堆内存统计

```
C:\Users\39497>jstat -gccapacity 13988
 NGCMN    NGCMX     NGC     SOC   S1C       EC      OGCMN      OGCMX       OGC         OC       MCMN     MCMX      MC     CCSMN    CCSMX     CCSC    YGC    FGC
 43520.0 689152.0 641536.0 13824.0 12800.0 604672.0     87552.0  1379328.0   187392.0   187392.0       0.0 1093632.0  50088.0       0.0 1048576.0   6568.0     31     5
```

- NGCMN：新生代最小容量
- NGCMX：新生代最大容量
- NGC：当前新生代容量
- S0C：第一个幸存区大小
- S1C：第二个幸存区的大小
- EC：伊甸园区的大小
- OGCMN：老年代最小容量
- OGCMX：老年代最大容量
- OGC：当前老年代大小
- OC:当前老年代大小
- MCMN:最小元数据容量
- MCMX：最大元数据容量
- MC：当前元数据空间大小
- CCSMN：最小压缩类空间大小
- CCSMX：最大压缩类空间大小
- CCSC：当前压缩类空间大小
- YGC：年轻代gc次数
- FGC：老年代GC次数

新生代垃圾回收统计

```
C:\Users\39497>jstat -gcnew 13988
 S0C     S1C     S0U     S1U   TT MTT  DSS       EC        EU      YGC     YGCT
13824.0 15360.0 12704.0     0.0 15  15 15360.0 612352.0    3237.1     32     0.287
```

- S0C：第一个幸存区的大小
- S1C：第二个幸存区的大小
- S0U：第一个幸存区的使用大小
- S1U：第二个幸存区的使用大小
- TT:对象在新生代存活的次数
- MTT:对象在新生代存活的最大次数
- DSS:期望的幸存区大小
- EC：伊甸园区的大小
- EU：伊甸园区的使用大小
- YGC：年轻代垃圾回收次数
- YGCT：年轻代垃圾回收消耗时间

新生代内存统计

```
C:\Users\39497>jstat -gcnewcapacity 13988
  NGCMN       NGCMX        NGC       SOCMX      SOC       S1CMX      S1C         ECMX        EC       YGC   FGC
   43520.0    689152.0    643584.0  229376.0   13824.0  229376.0   15360.0    688128.0    612352.0     32     5
```

- NGCMN：新生代最小容量
- NGCMX：新生代最大容量
- NGC：当前新生代容量
- S0CMX：最大幸存1区大小
- S0C：当前幸存1区大小
- S1CMX：最大幸存2区大小
- S1C：当前幸存2区大小
- ECMX：最大伊甸园区大小
- EC：当前伊甸园区大小
- YGC：年轻代垃圾回收次数

- FGC：老年代回收次数

老年代垃圾回收统计

```
C:\Users\39497>jstat -gcold 13988
   MC       MU      CCSC     CCSU       OC          OU       YGC    FGC    FGCT     GCT
50088.0  48901.7   6568.0   6291.8   187392.0     21213.2     32      5    0.405    0.692
```

- MC：方法区大小
- MU：方法区使用大小
- CCSC:压缩类空间大小
- CCSU:压缩类空间使用大小
- OC：老年代大小
- OU：老年代使用大小
- YGC：年轻代垃圾回收次数
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

老年代内存统计

```
C:\Users\39497>jstat -gcoldcapacity 13988
   OGCMN       OGCMX        OGC         OC       YGC   FGC    FGCT     GCT
    87552.0   1379328.0    187392.0    187392.0     32     5    0.405    0.692
```

- OGCMN：老年代最小容量
- OGCMX：老年代最大容量
- OGC：当前老年代大小
- OC：老年代大小
- YGC：年轻代垃圾回收次数
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

元数据空间统计

```
C:\Users\39497>jstat -gcmetacapacity 13988
   MCMN       MCMX        MC       CCSMN      CCSMX       CCSC     YGC   FGC    FGCT     GCT
      0.0  1093632.0    50088.0        0.0  1048576.0     6568.0     33     5    0.405    0.730
```

- MCMN:最小元数据容量
- MCMX：最大元数据容量
- MC：当前元数据空间大小
- CCSMN：最小压缩类空间大小
- CCSMX：最大压缩类空间大小
- CCSC：当前压缩类空间大小
- YGC：年轻代垃圾回收次数
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

```
C:\Users\39497>jstat -gcutil 13988
  S0     S1     E      O      M     CCS    YGC     YGCT    FGC    FGCT     GCT
  0.00  99.58  26.25  11.32  97.63  95.80     33    0.325     5    0.405    0.730
```

- S0：幸存1区当前使用比例
- S1：幸存2区当前使用比例
- E：伊甸园区使用比例
- O：老年代使用比例
- M：元数据区使用比例
- CCS：压缩使用比例
- YGC：年轻代垃圾回收次数
- FGC：老年代垃圾回收次数
- FGCT：老年代垃圾回收消耗时间
- GCT：垃圾回收消耗总时间

JVM运行情况预估

用 jstat gc -pid 命令可以计算出如下一些关键数据，有了这些数据就可以采用之前介绍过的优化思路，先给自己的系统设置一些初始性的JVM参数，比如堆内存大小，年轻代大小，Eden和Survivor的比例，老年代的大小，大对象的阈值，大龄对象进入老年代的阈值等。

年轻代对象增长的速率

可以执行命令 jstat -gc pid 1000 10 (每隔1秒执行1次命令，共执行10次)，通过观察EU(eden区的使用)来估算每秒eden大概新增多少对象，如果系统负载不高，可以把频率1秒换成1分钟，甚至10分钟来观察整体情况。注意，一般系统可能有高峰期和日常期，所以需要在不同的时间分别估算不同情况下对象增长速率。

Young GC的触发频率和每次耗时

知道年轻代对象增长速率我们就能推根据eden区的大小推算出Young GC大概多久触发一次，Young GC的平均耗时可以通过 YGCT/YGC 公式算出，根据结果我们大概就能知道**系统大概多久会因为Young GC的执行而卡顿多久。**

每次Young GC后有多少对象存活和进入老年代

这个因为之前已经大概知道Young GC的频率，假设是每5分钟一次，那么可以执行命令 jstat -gc pid 300000 10 ，观察每次结果eden，survivor和老年代使用的变化情况，在每次gc后eden区使用一般会大幅减少，survivor和老年代都有可能增长，这些增长的对象就是每次Young GC后存活的对象，同时还可以看出每次Young GC后进去老年代大概多少对象，从而可以推算出**老年代对象增长速率。**

Full GC的触发频率和每次耗时

知道了老年代对象的增长速率就可以推算出Full GC的触发频率了，Full GC的每次耗时可以用公式 FGCT/FGC 计算得出。

优化思路其实简单来说就是尽量让每次Young GC后的存活对象小于Survivor区域的50%，都留存在年轻代里。尽量别让对象进入老年代。尽量减少Full GC的频率，避免频繁Full GC对JVM性能的影响。

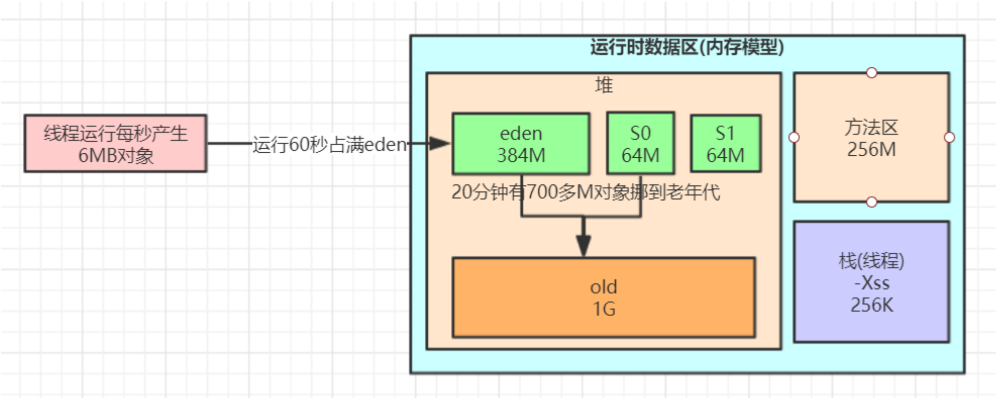
系统频繁Full GC导致系统卡顿是怎么回事

- 机器配置：2核4G
- JVM内存大小：2G
- 系统运行时间：7天
- 期间发生的Full GC次数和耗时：500多次，200多秒
- 期间发生的Young GC次数和耗时：1万多次，500多秒

大致算下来每天会发生70多次Full GC，平均每小时3次，每次Full GC在400毫秒左右；
每天会发生1000多次Young GC，每分钟会发生1次，每次Young GC在50毫秒左右。

JVM参数设置如下：

```
-Xms1536M -Xmx1536M -Xmn512M -Xss256K -XX:SurvivorRatio=6 -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=75 -XX:+UseCMSInitiatingOccupancyOnly
```



大家可以结合对象挪动到老年代那些规则推理下我们这个程序可能存在的一些问题

为了给大家看效果，我模拟了一个示例程序，打印了jstat的结果如下：

```
C:\Users\zhuge>jstat -gc 18424 2000 10000
 S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU    CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
65536.0 65536.0  0.0    0.0   393216.0 369692.4 1048576.0    0.0     4480.0 766.2  384.0   75.8       0    0.000   0      0.000   0.000
65536.0 65536.0  0.0    0.0   393216.0 369692.4 1048576.0    0.0     4480.0 766.2  384.0   75.8       0    0.000   0      0.000   0.000
65536.0 65536.0  0.0    0.0   393216.0 369692.4 1048576.0    0.0     4480.0 766.2  384.0   75.8       0    0.000   0      0.000   0.000
65536.0 65536.0  0.0    0.0   393216.0 369692.4 1048576.0    0.0     4480.0 766.2  384.0   75.8       0    0.000   0      0.000   0.000
65536.0 65536.0  0.0    0.0   393216.0 369692.4 1048576.0    0.0     4480.0 766.2  384.0   75.8       0    0.000   0      0.000   0.000
65536.0 65536.0  0.0    0.0   393216.0 369692.4 1048576.0    0.0     4480.0 766.2  384.0   75.8       0    0.000   0      0.000   0.000
65536.0 65536.0  0.0    0.0   393216.0 369692.4 1048576.0    0.0     4480.0 766.2  384.0   75.8       0    0.000   0      0.000   0.000
65536.0 65536.0  0.0    0.0   393216.0 369692.4 1048576.0    0.0     4480.0 766.2  384.0   75.8       0    0.000   0      0.000   0.000
65536.0 65536.0 65536.0  0.0    393216.0 130287.8 1048576.0   342209.8  36864.0 35131.6 4864.0 4538.9      2    0.288   0      0.000   0.288
65536.0 65536.0 65536.0  0.0    393216.0 130287.9 1048576.0   342209.8  36864.0 35131.6 4864.0 4538.9      2    0.288   0      0.000   0.288
65536.0 65536.0 65536.0 65536.0 393216.0    0.0    1048576.0   701558.1  37248.0 35432.1 4992.0 4575.5      3    0.442   0      0.000   0.442
65536.0 65536.0  0.0   65536.0 393216.0 270793.6 1048576.0   701558.1  37248.0 35432.1 4992.0 4575.5      3    0.442   0      0.000   0.442
65536.0 65536.0  0.0   65536.0 393216.0 270793.6 1048576.0   701558.1  37248.0 35432.1 4992.0 4575.5      3    0.442   0      0.000   0.442
65536.0 65536.0 65536.0  0.0    393216.0 11306.5  1048576.0   450265.2  37248.0 35434.8 4992.0 4574.9      5    0.510   1      0.148   0.658
65536.0 65536.0 65536.0  0.0    393216.0 12386.5  1048576.0   450265.2  37248.0 35434.8 4992.0 4574.9      5    0.510   1      0.148   0.658
65536.0 65536.0 65536.0  0.0    393216.0 12386.7  1048576.0   450265.2  37248.0 35434.8 4992.0 4574.9      5    0.510   1      0.148   0.658
65536.0 65536.0  0.0   65536.0 393216.0 144049.2 1048576.0   818634.4  37248.0 35434.8 4992.0 4574.9      6    0.597   2      0.149   0.746
65536.0 65536.0  0.0   65536.0 393216.0 144049.2 1048576.0   818634.4  37248.0 35434.8 4992.0 4574.9      6    0.597   2      0.149   0.746
65536.0 65536.0 65536.0  0.0    393216.0 270609.9 1048576.0   491866.8  37248.0 35435.4 4992.0 4575.4      7    0.644   3      0.193   0.838
65536.0 65536.0 65536.0  0.0    393216.0 271066.4 1048576.0   491866.8  37248.0 35435.4 4992.0 4575.4      7    0.644   3      0.193   0.838
65536.0 65536.0 65536.0  0.0    393216.0 271066.6 1048576.0   491866.8  37248.0 35435.4 4992.0 4575.4      7    0.644   3      0.193   0.838
65536.0 65536.0 65536.0  0.0    393216.0  7738.1  1048576.0   926934.9  37248.0 35435.4 4992.0 4575.4      9    0.764   4      0.194   0.958
65536.0 65536.0 65536.0  0.0    393216.0  7738.1  1048576.0   926934.9  37248.0 35435.4 4992.0 4575.4      9    0.764   4      0.194   0.958
65536.0 65536.0 65536.0  0.0    393216.0  7738.1  1048576.0   926934.9  37248.0 35435.4 4992.0 4575.4      9    0.764   4      0.194   0.958
65536.0 65536.0  0.0   65536.0 393216.0 128120.7 1048576.0   828331.2  37248.0 35435.4 4992.0 4575.4     10    0.827   6      0.236   1.063
65536.0 65536.0  0.0   65536.0 393216.0 128120.8 1048576.0   828331.2  37248.0 35435.4 4992.0 4575.4     10    0.827   6      0.236   1.063
65536.0 65536.0 65536.0  0.0    393216.0 251240.7 1048576.0   523973.8  37248.0 35435.7 4992.0 4575.4     11    0.863   7      0.280   1.143
65536.0 65536.0 65536.0  0.0    393216.0 251433.5 1048576.0   523973.8  37248.0 35435.7 4992.0 4575.4     11    0.863   7      0.280   1.143
65536.0 65536.0 65536.0  0.0    393216.0 251433.5 1048576.0   523973.8  37248.0 35435.7 4992.0 4575.4     11    0.863   7      0.280   1.143
65536.0 65536.0  0.0   65536.0 393216.0 368000.3 1048576.0   599799.0  37248.0 35435.7 4992.0 4575.4     12    0.894   7      0.280   1.174
65536.0 65536.0  0.0   65536.0 393216.0 368000.5 1048576.0   599799.0  37248.0 35435.7 4992.0 4575.4     12    0.894   7      0.280   1.174
65536.0 65536.0 57002.3  0.0    393216.0 94366.8  1048576.0   599799.4  37248.0 35435.7 4992.0 4575.4     13    0.902   7      0.280   1.182
65536.0 65536.0  0.0   65536.0 393216.0 96266.8  1048576.0   361632.6  37248.0 35435.7 4992.0 4575.4     14    0.969   9      0.293   1.261
65536.0 65536.0  0.0   65536.0 393216.0 96267.0  1048576.0   361632.6  37248.0 35435.7 4992.0 4575.4     14    0.969   9      0.293   1.261
65536.0 65536.0 65536.0  0.0    393216.0 219672.2 1048576.0   585017.0  37248.0 35435.7 4992.0 4575.4     15    1.020   9      0.293   1.313
65536.0 65536.0 65536.0  0.0    393216.0 219725.3 1048576.0   585017.0  37248.0 35435.7 4992.0 4575.4     15    1.020   9      0.293   1.313
65536.0 65536.0 65536.0  0.0    393216.0 220181.8 1048576.0   585017.0  37248.0 35435.7 4992.0 4575.4     15    1.020   9      0.293   1.313
65536.0 65536.0  0.0   65536.0 393216.0 339756.4 1048576.0   689555.2  37248.0 35435.7 4992.0 4575.4     16    1.046   9      0.293   1.339
65536.0 65536.0  0.0   65536.0 393216.0 339756.6 1048576.0   689555.2  37248.0 35435.7 4992.0 4575.4     16    1.046   9      0.293   1.339
65536.0 65536.0  0.0    0.0   393216.0 72552.7  1048576.0   450062.7  37248.0 35435.7 4992.0 4575.4     18    1.160  10      0.439   1.598
65536.0 65536.0  0.0    0.0   393216.0 73277.2  1048576.0   450062.7  37248.0 35435.7 4992.0 4575.4     18    1.160  10      0.439   1.598
65536.0 65536.0  0.0    0.0   393216.0 73574.0  1048576.0   450062.7  37248.0 35435.7 4992.0 4575.4     18    1.160  10      0.439   1.598
65536.0 65536.0 65536.0  0.0    393216.0 193645.5 1048576.0   764012.4  37248.0 35436.0 4992.0 4575.4     19    1.213  10      0.439   1.651
65536.0 65536.0 65536.0  0.0    393216.0 193645.6 1048576.0   764012.4  37248.0 35436.0 4992.0 4575.4     19    1.213  10      0.439   1.651
65536.0 65536.0 65536.0  0.0    393216.0 193645.8 1048576.0   764012.4  37248.0 35436.0 4992.0 4575.4     19    1.213  10      0.439   1.651
65536.0 65536.0  0.0    0.0   393216.0 316520.6 1048576.0   207954.8  37248.0 35436.0 4992.0 4575.4     20    1.213  11      0.580   1.793
```

我们可以看到young gc和full gc都太频繁了，而且看到有大量的对象频繁的被挪动到老年代，这种情况我们可以借助jmap命令大概看下是什么对象

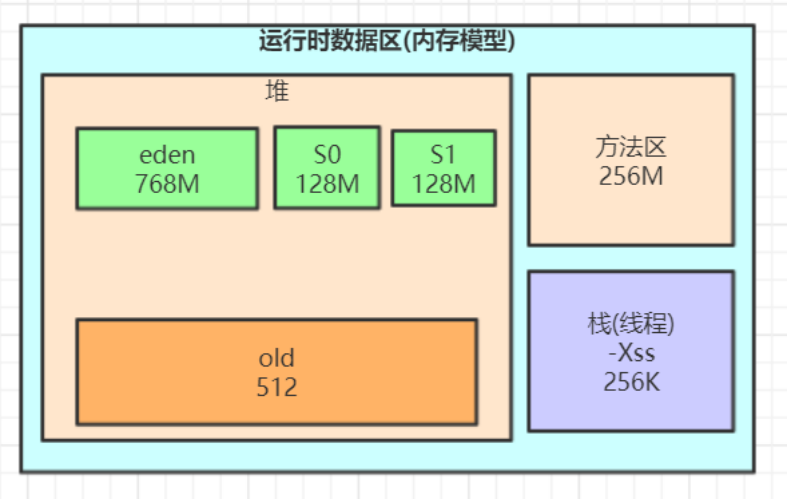
```
C:\Users\zhuge>jmap -histo 18424

 num     #instances         #bytes  class name
----------------------------------------------
   1:          9146      848440368  [B
   2:         73272        8817648  [C
   3:          2804        2656368  [I
   4:         47178        1132272  java.lang.String
   5:          7364         816240  java.lang.Class
   6:         10009         480432  java.nio.HeapCharBuffer
   7:         13862         443584  java.util.concurrent.ConcurrentHashMap$Node
   8:          6697         410424  [Ljava.lang.Object;
   9:          8864         283648  java.util.HashMap$Node
  10:          2883         242936  [Ljava.util.HashMap$Node;
  11:          6006         240240  java.util.LinkedHashMap$Entry
  12:          2684         236192  java.lang.reflect.Method
  13:          7724         185376  com.jvm.User
  14:         10197         163152  java.lang.Object
  15:          2649         148344  java.util.LinkedHashMap
  16:           105         147536  [Ljava.util.concurrent.ConcurrentHashMap$Node;
  17:          1276          75672  [Ljava.lang.String;
  18:          1712          68480  java.lang.ref.SoftReference
  19:          2830          62368  [Ljava.lang.Class;
  20:          1140          54720  java.util.HashMap
  21:          2234          53616  java.util.ArrayList
```

然后就要检查下代码对应的地方，看下是否有问题代码的存在，比如找到了下面的类似代码

```java
import java.util.ArrayList;

@RestController
public class IndexController {

 @RequestMapping("/user/process")
 public String processUserData() throws InterruptedException {
 ArrayList<User> users = queryUsers();

 for (User user: users) {
 //TODO 业务处理
 System.out.println("user:" + user.toString());
 }
 return "end";
 }

 /**
 * 模拟批量查询用户场景
 * @return
 */
 private ArrayList<User> queryUsers() {
 ArrayList<User> users = new ArrayList<>();
 for (int i = 0; i < 5000; i++) {
 users.add(new User(i,"zhuge"));
 }
 return users;
 }
}
```

对于这种业务场景可以先优化下JVM参数：

```
-Xms1536M -Xmx1536M -Xmn1024M -Xss256K -XX:SurvivorRatio=6 -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M
-XX:+UseParNewGC -XX:+UseConcMarkSweepGC -XX:CMSInitiatingOccupancyFraction=92 -XX:+UseCMSInitiatingOccupancyOnly
```



同时，java的代码也是需要优化的，一次查询出500M的对象出来，明显不合适，要根据之前说的各种原则尽量优化到合适的值，尽量消除这种朝生夕死的对象导致的full gc

内存泄露到底是怎么回事

再给大家讲一种情况，一般电商架构可能会使用多级缓存架构，就是redis加上JVM级缓存，大多数同学可能为了图方便对于JVM级缓存就简单使用一个hashmap，于是不断往里面放缓存数据，但是很少考虑这个map的容量问题，结果这个缓存map越来越大，一直占用着老年代的很多空间，时间长了就会导致full gc非常频繁，这就是一种内存泄漏，对于一些老旧数据没有及时清理导致一直占用着宝贵的内存资源，时间长了除了导致full gc，还有可能导致OOM。

这种情况完全可以考虑采用一些成熟的JVM级缓存框架来解决，比如ehcache等自带一些LRU数据淘汰算法的框架来作为JVM级的缓存。