1、JVM调优目标：使用较小的内存占用来获得较高的吞吐量或者较低的延迟。

程序在上线前的测试或运行中有时会出现一些大大小小的JVM问题，比如cpu load过高、请求延迟、tps降低等，甚至出现内存泄漏（每次垃圾收集使用的时间越来越长，垃圾收集频率越来越高，每次垃圾收集清理掉的垃圾数据越来越少）、内存溢出导致系统崩溃，因此需要对JVM进行调优，使得程序在正常运行的前提下，获得更高的用户体验和运行效率。

这里有几个比较重要的指标：

- 内存占用：程序正常运行需要的内存大小。
- 延迟：由于垃圾收集而引起的程序停顿时间。
- 吞吐量：用户程序运行时间占用户程序和垃圾收集占用总时间的比值。

当然，和CAP原则一样，同时满足一个程序内存占用小、延迟低、高吞吐量是不可能的，程序的目标不同，调优时所考虑的方向也不同，在调优之前，必须要结合实际场景，有明确的的优化目标，找到性能瓶颈，对瓶颈有针对性的优化，最后进行测试，通过各种监控工具确认调优后的结果是否符合目标。

2、JVM调优工具

（1）调优可以依赖、参考的数据有系统运行日志、堆栈错误信息、gc日志、线程快照、堆转储快照等。

①系统运行日志：系统运行日志就是在程序代码中打印出的日志，描述了代码级别的系统运行轨迹（执行的方法、入参、返回值等），一般系统出现问题，系统运行日志是首先要查看的日志。

②堆栈错误信息：当系统出现异常后，可以根据堆栈信息初步定位问题所在，比如根据“java.lang.OutOfMemoryError: Java heap space”可以判断是堆内存溢出；根据“java.lang.StackOverflowError”可以判断是栈溢出；根据“java.lang.OutOfMemoryError: PermGen space”可以判断是方法区溢出等。

③GC日志：程序启动时用 -XX:+PrintGCDetails 和 -Xloggc:/data/jvm/gc.log 可以在程序运行时把gc的详细过程记录下来，或者直接配置“-verbose:gc”参数把gc日志打印到控制台，通过记录的gc日志可以分析每块内存区域gc的频率、时间等，从而发现问题，进行有针对性的优化。
比如如下一段GC日志：

```
2018-08-02T14:39:11.560-0800: 10.171: [GC [PSYoungGen: 30128K->4091K(30208K)] 51092K->50790K(98816K), 0.0140970 secs] [Times: user=0.02 sys=0.03, real=0.01 secs]

2018-08-02T14:39:11.574-0800: 10.185: [Full GC [PSYoungGen: 4091K->0K(30208K)] [ParOldGen: 46698K->50669K(68608K)] 50790K->50669K(98816K) [PSPermGen: 2635K->2634K(21504K)], 0.0160030 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]

2018-08-02T14:39:14.045-0800: 12.656: [GC [PSYoungGen: 14097K->4064K(30208K)] 64766K->64536K(98816K), 0.0117690 secs] [Times: user=0.02 sys=0.01, real=0.01 secs]

2018-08-02T14:39:14.057-0800: 12.668: [Full GC [PSYoungGen: 4064K->0K(30208K)] [ParOldGen: 60471K->401K(68608K)] 64536K->401K(98816K) [PSPermGen: 2634K->2634K(21504K)], 0.0102020 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
```

上面一共是4条GC日志，来看第一行日志，“2018-08-02T14:39:11.560-0800”是精确到了毫秒级别的UTC通用标准时间格式，配置了“-XX:+PrintGCDateStamps”这个参数可以跟随gc日志打印出这种时间戳，“10.171”是从JVM启动到发生gc经过的秒数。第一行日志正文开头的“[GC”说明这次GC没有发生Stop-The-World（用户线程停顿），第二行日志正文开头的“[Full GC”说明这次GC发生了Stop-The-World，所以说，[GC和[Full GC跟新生代和老年代没关系，和垃圾收集器的类型有关系，如果直接调用System.gc()，将显示[Full GC(System)。接下来的“[PSYoungGen”、“[ParOldGen”表示GC发生的区域，具体显示什么名字也跟垃圾收集器有关，比如这里的“[PSYoungGen”表示Parallel Scavenge收集器，

“[ParOldGen”表示Serial Old收集器，此外，Serial收集器显示“[DefNew”，ParNew收集器显示“[ParNew”等。再往后的“30128K->4091K(30208K)”表示进行了这次gc后，该区域的内存使用空间由30128K减小到4091K，总内存大小为30208K。每个区域gc描述后面的“51092K->50790K(98816K), 0.0140970 secs”进行了这次垃圾收集后，整个堆内存的内存使用空间由51092K减小到50790K，整个堆内存总空间为98816K，gc耗时0.0140970秒。

④线程快照：顾名思义，根据线程快照可以看到线程在某一时刻的状态，当系统中可能存在请求超时、死循环、死锁等情况是，可以根据线程快照来进一步确定问题。通过执行虚拟机自带的“jstack pid”命令，可以dump出当前进程中线程的快照信息，更详细的使用和分析网上有很多例，这篇文章写到这里已经很长了就不过多叙述了，贴一篇博客供参考：http://www.cnblogs.com/kongzhongqijing/articles/3630264.html

⑤堆转储快照：程序启动时可以使用“-XX:+HeapDumpOnOutOfMemory”和“-XX:HeapDumpPath=/data/jvm/dumpfile.hprof”，当程序发生内存溢出时，把当时的内存快照以文件形式进行转储（也可以直接用jmap命令转储程序运行时任意时刻的内存快照），事后对当时的内存使用情况进行分析。

（2）JVM调优工具

①用 jps（JVM process Status）可以查看虚拟机启动的所有进程、执行主类的全名、JVM启动参数，比如当执行了JPSTest类中的main方法后（main方法持续执行），执行 jps -l可看到下面的JPSTest类的pid为31354，加上-v参数还可以看到JVM启动参数。

```
3265
32914 sun.tools.jps.Jps
31353 org.jetbrains.jps.cmdline.Launcher
31354 com.danny.test.code.jvm.JPSTest380
```

②用jstat（JVM Statistics Monitoring Tool）监视虚拟机信息
jstat -gc pid 500 10：每500毫秒打印一次Java堆状况（各个区的容量、使用容量、gc时间等信息），打印10次

```
S0C    S1C    S0U    S1U      EC       EU        OC         OU       MC     MU
 CCSC   CCSU   YGC     YGCT    FGC    FGCT     GCT
11264.0 11264.0 11202.7  0.0   11776.0   1154.3   68608.0    36238.7     -      -
     -      -        14    0.077   7      0.049    0.126
11264.0 11264.0 11202.7  0.0   11776.0   4037.0   68608.0    36238.7     -      -
     -      -        14    0.077   7      0.049    0.126
11264.0 11264.0 11202.7  0.0   11776.0   6604.5   68608.0    36238.7     -      -
     -      -        14    0.077   7      0.049    0.126
11264.0 11264.0 11202.7  0.0   11776.0   9487.2   68608.0    36238.7     -      -
     -      -        14    0.077   7      0.049    0.126
11264.0 11264.0  0.0    0.0   11776.0   258.1    68608.0    58983.4     -      -
     -      -        15    0.082   8      0.059    0.141
11264.0 11264.0  0.0    0.0   11776.0   3076.8   68608.0    58983.4     -      -
     -      -        15    0.082   8      0.059    0.141
11264.0 11264.0  0.0    0.0   11776.0    0.0     68608.0     390.0      -      -
     -      -        16    0.084   9      0.066    0.149
11264.0 11264.0  0.0    0.0   11776.0    0.0     68608.0     390.0      -      -
     -      -        16    0.084   9      0.066    0.149
11264.0 11264.0  0.0    0.0   11776.0   258.1    68608.0     390.0      -      -
     -      -        16    0.084   9      0.066    0.149
11264.0 11264.0  0.0    0.0   11776.0   3012.8   68608.0     390.0      -      -
     -      -        16    0.084   9      0.066    0.149
```

jstat还可以以其他角度监视各区内存大小、监视类装载信息等，具体可以google jstat的详细用法。

③用jmap（Memory Map for Java）查看堆内存信息
执行jmap -histo pid可以打印出当前堆中所有每个类的实例数量和内存占用，如下，class name是每个类的类名（[B是byte类型，[C是char类型，[I是int类型），bytes是这个类的所有示例占用内存大小，instances是这个类的实例数量：

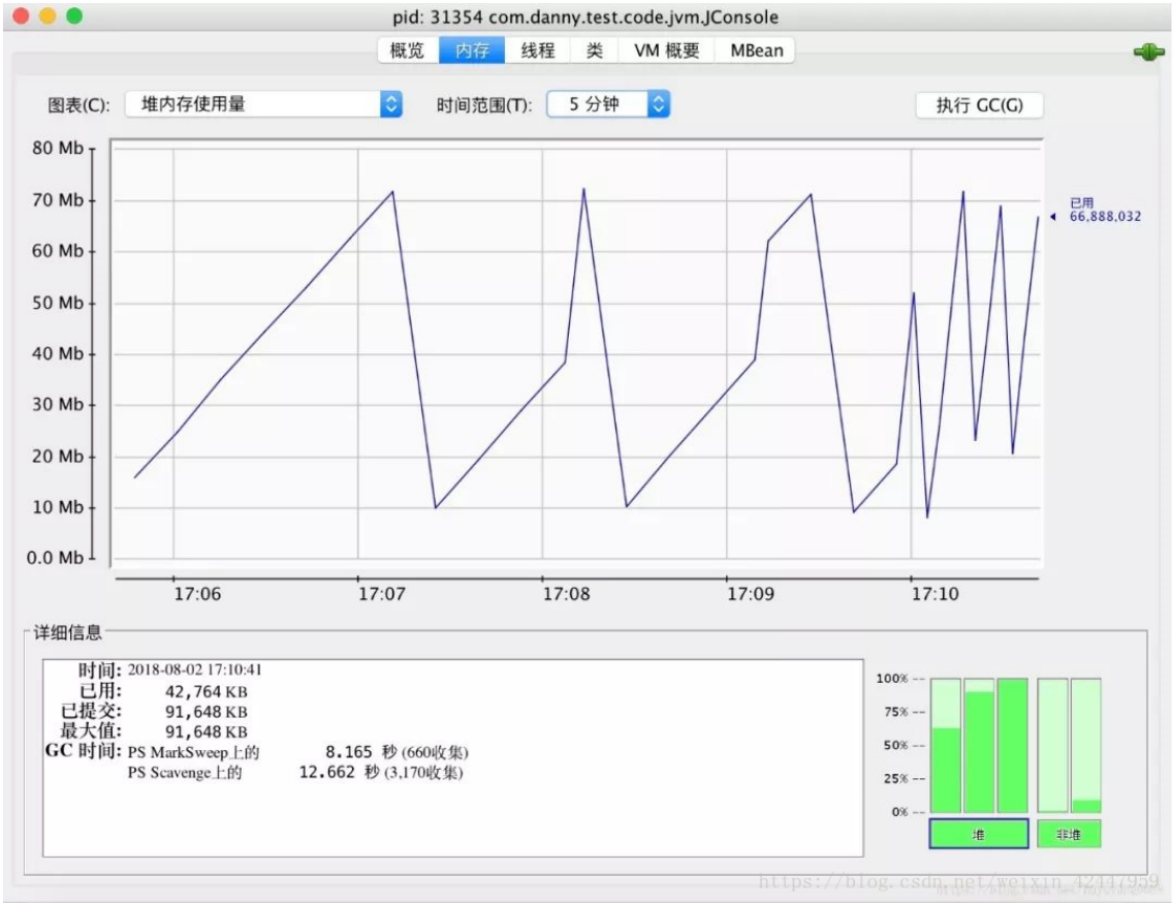
```
num     #instances         #bytes  class name
----------------------------------------------
  1:          2291       29274080  [B
  2:         15252        1961040  <methodKlass>
  3:         15252        1871400  <constMethodKlass>
  4:         18038         721520  java.util.TreeMap$Entry
  5:          6182         530088  [C
  6:         11391         273384  java.lang.Long
  7:          5576         267648  java.util.TreeMap
  8:            50         155872  [I
  9:          6124         146976  java.lang.String
 10:          3330         133200  java.util.LinkedHashMap$Entry
 11:          5544         133056  javax.management.openmbean.CompositeDataSupport
```

执行 jmap -dump 可以转储堆内存快照到指定文件，比如执行 jmap -dump:format=b,file=/data/jvm/dumpfile_jmap.hprof 3361 可以把当前堆内存的快照转储到dumpfile_jmap.hprof文件中，然后可以对内存快照进行分析。

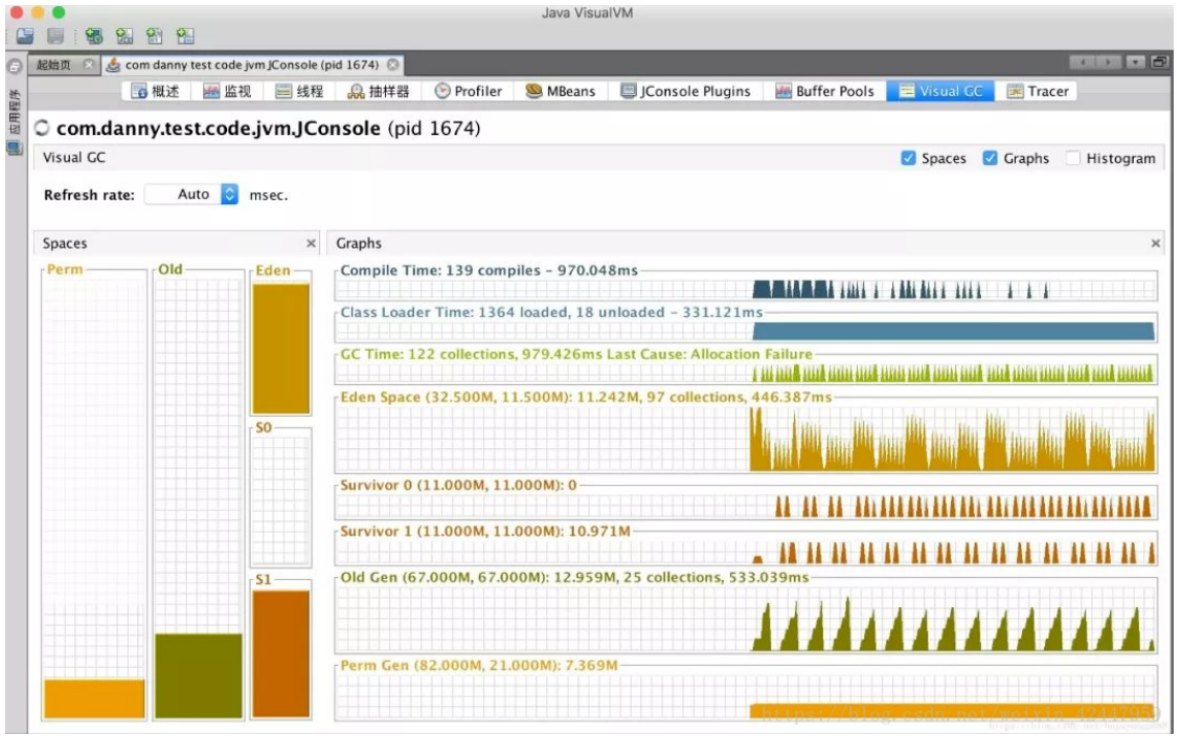
④利用jconsole、jvisualvm分析内存信息(各个区如Eden、Survivor、Old等内存变化情况)，如果查看的是远程服务器的JVM，程序启动需要加上如下参数：

```
"-Dcom.sun.management.jmxremote=true"
"-Djava.rmi.server.hostname=12.34.56.78"
"-Dcom.sun.management.jmxremote.port=18181"
"-Dcom.sun.management.jmxremote.authenticate=false"
"-Dcom.sun.management.jmxremote.ssl=false"
```

下图是jconsole界面，概览选项可以观测堆内存使用量、线程数、类加载数和CPU占用率；内存选项可以查看堆中各个区域的内存使用量和左下角的详细描述（内存大小、GC情况等）；线程选项可以查看当前JVM加载的线程，查看每个线程的堆栈信息，还可以检测死锁；VM概要描述了虚拟机的各种详细参数。（jconsole功能演示）



下图是jvisualvm的界面，功能比jconsole略丰富一些，不过大部分功能都需要安装插件。概述跟jconsole的VM概要差不多，描述的是jvm的详细参数和程序启动参数；监视展示的和jconsole的概览界面差不多（CPU、堆/方法区、类加载、线程）；线程和jconsole的线程界面差不多；抽样器可以展示当前占用内存的类的排行榜及其实例的个数；Visual GC可以更丰富地展示当前各个区域的内存占用大小及历史信息（下图）。（jvisualvm功能演示）

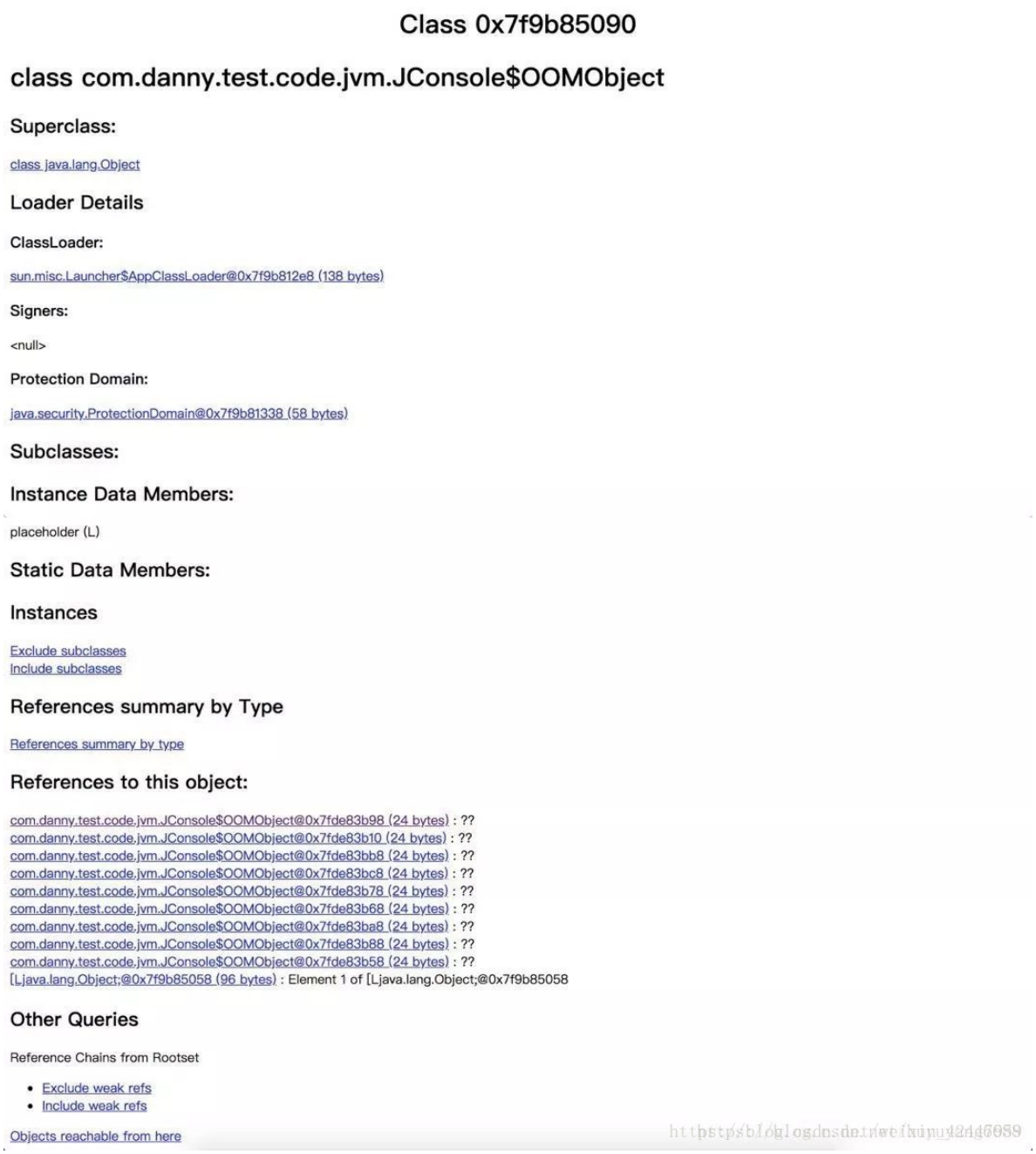


⑤分析堆转储快照

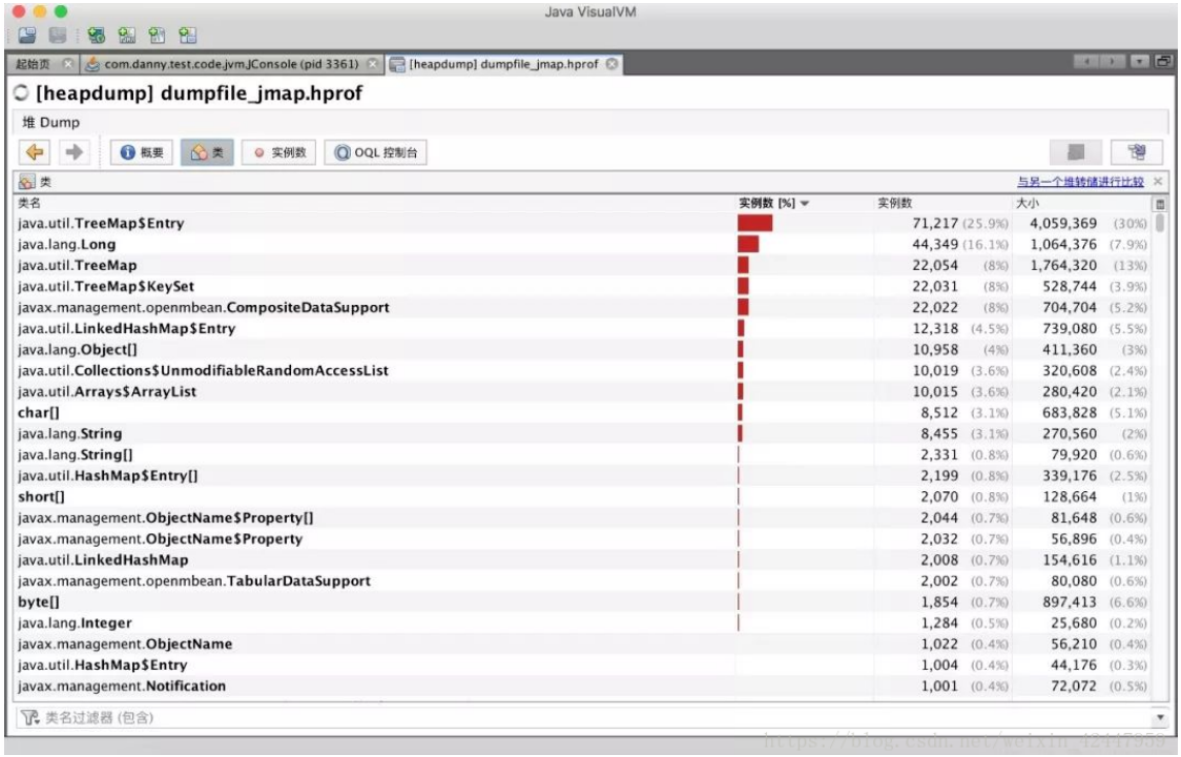
前面说到配置了 “-XX:+HeapDumpOnOutOfMemory” 参数可以在程序发生内存溢出时dump出当前的内存快照，也可以用jmap命令随时dump出当时内存状态的快照信息，dump的内存快照一般是以.hprof为后缀的二进制格式文件。
可以直接用 jhat（JVM Heap Analysis Tool） 命令来分析内存快照，它的本质实际上内嵌了一个微型的服务器，可以通过浏览器来分析对应的内存快照，比如执行 jhat -port 9810 -J-Xmx4G /data/jvm/dumpfile_jmap.hprof 表示以9810端口启动 jhat 内嵌的服务器：

```
Reading from /Users/dannyhoo/data/jvm/dumpfile_jmap.hprof...
Dump file created Fri Aug 03 15:48:27 CST 2018
Snapshot read, resolving...
Resolving 276472 objects...
Chasing references, expect 55
dots.....................................................
Eliminating duplicate
references.....................................................
Snapshot resolved.
Started HTTP server on port 9810
Server is ready.
```

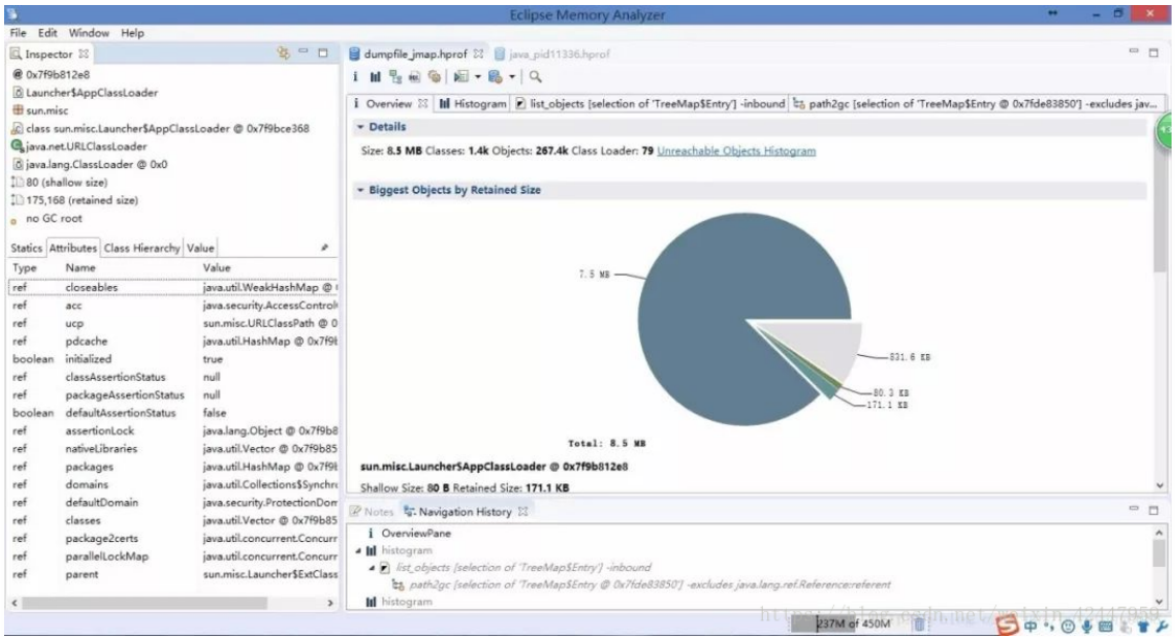
在控制台可以看到服务器启动了，访问 http://127.0.0.1:9810/ 可以看到对快照中的每个类进行分析的结果（界面略low），下图是我随便选择了一个类的信息，有这个类的父类，加载这个类的类加载器和占用的空间大小，下面还有这个类的每个实例（References）及其内存地址和大小，点进去会显示这个实例的一些成员变量等信息：



jvisualvm也可以分析内存快照，在jvisualvm菜单的“文件”-“装入”，选择堆内存快照，快照中的信息就以图形界面展示出来了，如下，主要可以查看每个类占用的空间、实例的数量和实例的详情等：



还有很多分析内存快照的第三方工具，比如eclipse mat，它比jvisualvm功能更专业，出了查看每个类及对应实例占用的空间、数量，还可以查询对象之间的调用链，可以查看某个实例到GC Root之间的链，等等。可以在eclipse中安装mat插件，也可以下载独立的版本（http://www.eclipse.org/mat/downloads.php），我在mac上安装后运行起来老卡死~下面是在windows上的截图（MAT功能演示）：



（3）JVM调优经验

JVM配置方面，一般情况可以先用默认配置（基本的一些初始参数可以保证一般的应用跑的比较稳定了），在测试中根据系统运行状况（会话并发情况、会话时间等），结合gc日志、内存监控、使用的垃圾收集器等进行合理的调整，当老年代内存过小时可能引起频繁Full GC，当内存过大时Full GC时间会特别长。

那么JVM的配置比如新生代、老年代应该配置多大最合适呢？答案是不一定，调优就是找答案的过程，物理内存一定的情况下，新生代设置越大，老年代就越小，Full GC频率就越高，但Full GC时间越短；相反新生代设置越小，老年代就越大，Full GC频率就越低，但每次Full GC消耗的时间越大。建议如下：

- -Xms和-Xmx的值设置成相等，堆大小默认为-Xms指定的大小，默认空闲堆内存小于40%时，JVM会扩大堆到-Xmx指定的大小；空闲堆内存大于70%时，JVM会减小堆到-Xms指定的大小。如果在Full GC后满足不了内存需求会动态调整，这个阶段比较耗费资源。
- 新生代尽量设置大一些，让对象在新生代多存活一段时间，每次Minor GC 都要尽可能多的收集垃圾对象，防止或延迟对象进入老年代的机会，以减少应用程序发生Full GC的频率。
- 老年代如果使用CMS收集器，新生代可以不用太大，因为CMS的并行收集速度也很快，收集过程比较耗时的并发标记和并发清除阶段都可以与用户线程并发执行。
- 方法区大小的设置，1.6之前的需要考虑系统运行时动态增加的常量、静态变量等，1.7只要差不多能装下启动时和后期动态加载的类信息就行。

代码实现方面，性能出现问题比如程序等待、内存泄漏除了JVM配置可能存在问题，代码实现上也有很大关系：

- 避免创建过大的对象及数组：过大的对象或数组在新生代没有足够空间容纳时会直接进入老年代，如果是短命的大对象，会提前出发Full GC。
- 避免同时加载大量数据，如一次从数据库中取出大量数据，或者一次从Excel中读取大量记录，可以分批读取，用完尽快清空引用。
- 当集合中有对象的引用，这些对象使用完之后要尽快把集合中的引用清空，这些无用对象尽快回收避免进入老年代。
- 可以在合适的场景（如实现缓存）采用软引用、弱引用，比如用软引用来为ObjectA分配实例：SoftReference objectA=new SoftReference(); 在发生内存溢出前，会将objectA列入回收范围进行二次回收，如果这次回收还没有足够内存，才会抛出内存溢出的异常。
  避免产生死循环，产生死循环后，循环体内可能重复产生大量实例，导致内存空间被迅速占满。
- 尽量避免长时间等待外部资源（数据库、网络、设备资源等）的情况，缩小对象的生命周期，避免进入老年代，如果不能及时返回结果可以适当采用异步处理的方式等。

（4）JVM问题排查记录案例

JVM服务问题排查 https://blog.csdn.net/jacin1/article/details/44837595

次让人难以忘怀的排查频繁Full GC过程 http://caogen81.iteye.com/blog/1513345

线上FullGC频繁的排查 https://blog.csdn.net/wilsonpeng3/article/details/70064336/

【JVM】线上应用故障排查 https://www.cnblogs.com/Dhouse/p/7839810.html

一次JVM中FullGC问题排查过程 http://iamzhongyong.iteye.com/blog/1830265

JVM内存溢出导致的CPU过高问题排查案例 https://blog.csdn.net/nielinqi520/article/details/78455614

一个java内存泄漏的排查案例 https://blog.csdn.net/aasgis6u/article/details/54928744

（5）常用JVM参数参考：

参数	说明	实例
-Xms	初始堆大小，默认物理内存的1/64	-Xms512M
-Xmx	最大堆大小，默认物理内存的1/4	-Xms2G
-Xmn	新生代内存大小，官方推荐为整个堆的3/8	-Xmn512M
-Xss	线程堆栈大小，jdk1.5及之后默认1M，之前默认256k	-Xss512k
-XX:NewRatio=n	设置新生代和年老代的比值。如:为3，表示年轻代与年老代比值为1：3，年轻代占整个年轻代年老代和的1/4	-XX:NewRatio=3

参数	说明	实例
-XX:SurvivorRatio=n	年轻代中Eden区与两个Survivor区的比值。注意Survivor区有两个。如:8，表示Eden：Survivor=8:1:1，一个Survivor区占整个年轻代的1/8	-XX:SurvivorRatio=8
-XX:PermSize=n	永久代初始值，默认为物理内存的1/64	-XX:PermSize=128M
-XX:MaxPermSize=n	永久代最大值，默认为物理内存的1/4	-XX:MaxPermSize=256M
-verbose:class	在控制台打印类加载信息	
-verbose:gc	在控制台打印垃圾回收日志	
-XX:+PrintGC	打印GC日志，内容简单	
-XX:+PrintGCDetails	打印GC日志，内容详细	
-XX:+PrintGCDateStamps	在GC日志中添加时间戳	
-Xloggc:filename	指定gc日志路径	-Xloggc:/data/jvm/gc.log
-XX:+UseSerialGC	年轻代设置串行收集器Serial	
-XX:+UseParallelGC	年轻代设置并行收集器Parallel Scavenge	
-XX:ParallelGCThreads=n	设置Parallel Scavenge收集时使用的CPU数。并行收集线程数。	-XX:ParallelGCThreads=4
-XX:MaxGCPauseMillis=n	设置Parallel Scavenge回收的最大时间(毫秒)	-XX:MaxGCPauseMillis=100
-XX:GCTimeRatio=n	设置Parallel Scavenge垃圾回收时间占程序运行时间的百分比。公式为1/(1+n)	-XX:GCTimeRatio=19
-XX:+UseParallelOldGC	设置老年代为并行收集器ParallelOld收集器	
-XX:+UseConcMarkSweepGC	设置老年代并发收集器CMS	
-XX:+CMSIncrementalMode	设置CMS收集器为增量模式，适用于单CPU情况。	