前置启动程序

事先启动一个web应用程序，用jps查看其进程id，接着用各种jdk自带命令优化应用

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

```
D:\>jps
14660 jar
22636 Jps

D:\>jmap -histo 14660 > ./log.txt
```

```
jmap -histo 14660 #查看历史生成的实例
jmap -histo:live 14660 #查看当前存活的实例，执行过程中可能会触发一次full gc
```

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

```
D:\>jmap -heap 14660
Attaching to process ID 14660, 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              = 4265607168 (4068.0MB)
   NewSize                  = 89128960 (85.0MB)
   MaxNewSize               = 1421869056 (1356.0MB)
   OldSize                  = 179306496 (171.0MB)
   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 = 839385088 (800.5MB)
   used     = 55963224 (53.370689392089844MB)
   free     = 783421864 (747.1293106079102MB)
   6.6671691932654396% used
From Space:
   capacity = 8388608 (8.0MB)
   used     = 8363072 (7.97564697265625MB)
   free     = 25536 (0.02435302734375MB)
   99.69558715820312% used
To Space:
   capacity = 12582912 (12.0MB)
   used     = 0 (0.0MB)
   free     = 12582912 (12.0MB)
   0.0% used
PS Old Generation
   capacity = 131072000 (125.0MB)
   used     = 28763248 (27.430770874023438MB)
   free     = 102308752 (97.56922912597656MB)
   21.94461669921875% used

23750 interned Strings occupying 2918552 bytes.
```

堆内存dump

```
jmap -dump:format=b,file=eureka.hprof 14660
```

```
D:\>jmap -dump:format=b,file=eureka.hprof 14660
Dumping heap to D:\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)

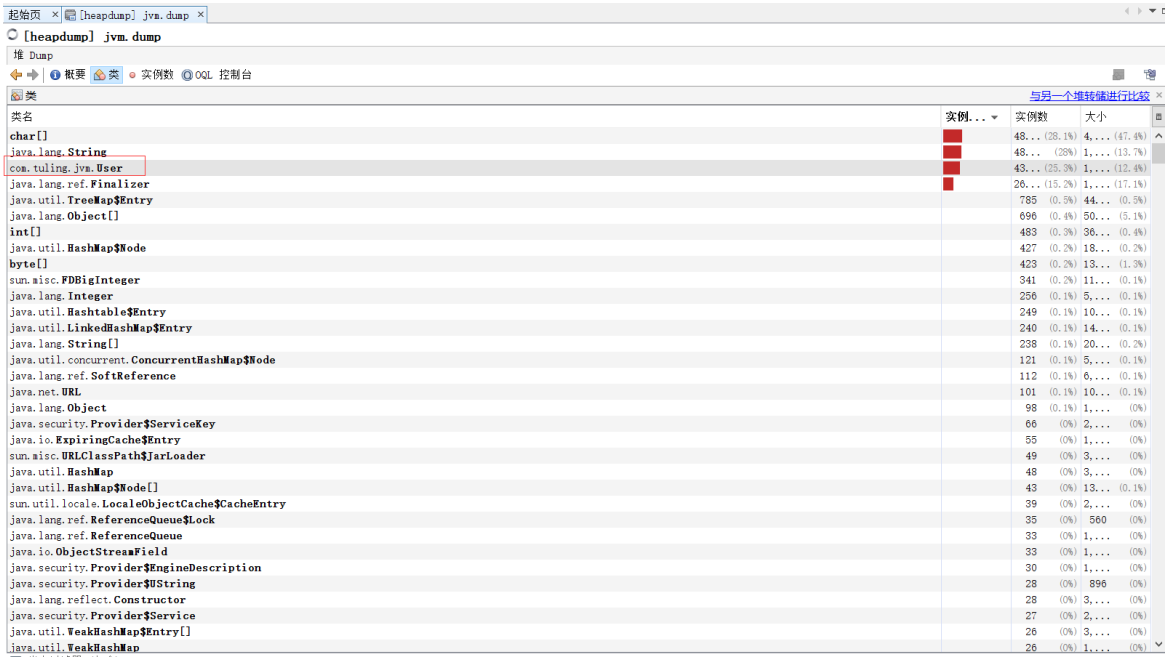
1. -XX:+HeapDumpOnOutOfMemoryError
2. -XX:HeapDumpPath=./ (路径)

示例代码：

```
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) {
```

```
12  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加进程id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 [0x000000002047f0
   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 [0x000000002037f0
   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

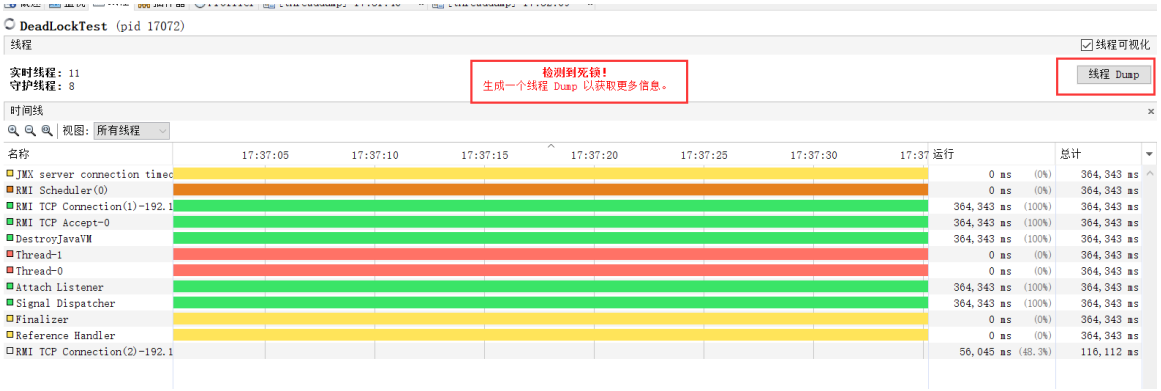
tid=0x000000001fa9e000 线程id

nid=0x2d64 线程对应的本地线程标识nid

java.lang.Thread.State: BLOCKED 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=8888 -Djava.rmi.server.hostname=192.168.65.60 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false -jar microservice-eureka-server.jar
```

PS：

-Dcom.sun.management.jmxremote.port 为远程机器的JMX端口

-Djava.rmi.server.hostname 为远程机器IP

tomcat的JMX配置：在catalina.sh文件里的最后一个JAVA_OPTS的赋值语句下一行增加如下配置行

```
JAVA_OPTS="$JAVA_OPTS -Dcom.sun.management.jmxremote.port=8888 -Djava.rmi.server.hostname=192.168.50.60 -Dcom.sun.management.jmxremote.ssl=false -Dcom.sun.management.jmxremote.authenticate=false"
```

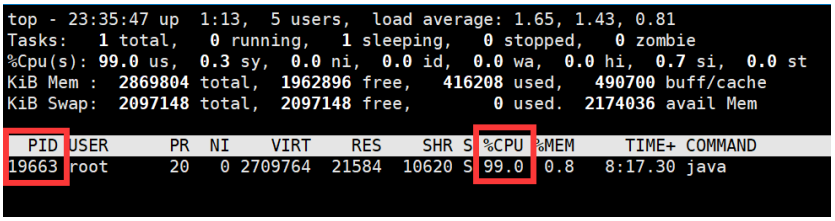
连接时确认下端口是否通畅，可以临时关闭下防火墙

```
systemctl stop firewalld #临时关闭防火墙
```

jstack找出占用cpu最高的线程堆栈信息

```
package com.tuling.jvm;

/**
 * 运行此代码，cpu会飙高
 */
public class Math {

 public static final int initData = 666;
 public static User user = new User();

 public int compute() { //一个方法对应一块栈帧内存区域
 int a = 1;
 int b = 2;
 int c = (a + b) * 10;
 return c;
 }

 public static void main(String[] args) {
 Math math = new Math();
 while (true){
 math.compute();
 }
 }
}
```

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19663



2，按H，获取每个线程的内存情况

```
top - 23:36:24 up  1:13,  5 users,  load average: 1.77, 1.49, 0.85
Threads:  11 total,   1 running,  10 sleeping,   0 stopped,   0 zombie
%Cpu(s): 99.7 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.3 si,  0.0 st
KiB Mem :  2869804 total,  1962896 free,   416208 used,   490700 buff/cache
KiB Swap:  2097148 total,  2097148 free,        0 used.  2174036 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S %CPU %MEM     TIME+ COMMAND
19664 root      20   0 2709764  21584  10620 R 99.0  0.8   8:53.66 java
19663 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.00 java
19665 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.01 java
19666 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.00 java
19667 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.00 java
19668 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.00 java
19669 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.00 java
19670 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.00 java
19671 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.00 java
19672 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.19 java
19777 root      20   0 2709764  21584  10620 S  0.0  0.8   0:00.00 java
```

3，找到内存和cpu占用最高的线程tid，比如19664

4，转为十六进制得到 0x4cd0，此为线程id的十六进制表示

5，执行 jstack 19663|grep -A 10 4cd0，得到线程堆栈信息中 4cd0 这个线程所在行的后面10行，从堆栈中可以发现导致cpu飙高的调用方法

```
[root@localhost ~]# jstack 19663|grep -A 10 4cd0
"main" #1 prio=5 os_prio=0 tid=0x00007fbb30009800 nid=0x4cd0 runnable [0x00007fbb380e5000]
   java.lang.Thread.State: RUNNABLE
        at com.tuling.jvm.Math.main(Math.java:22)

"VM Thread" os_prio=0 tid=0x00007fbb3006e000 nid=0x4cd1 runnable

"VM Periodic Task Thread" os_prio=0 tid=0x00007fbb300b7000 nid=0x4cd8 waiting on condition

JNI global references: 5
```

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

```
D:\>jinfo -flags 14124
Attaching to process ID 14124, please wait...
Debugger attached successfully.
Server compiler detected.
JVM version is 25.45-b02
Non-default VM flags: -XX:CICompilerCount=4 -XX:InitialHeapSize=10485760 -XX:MaxHeapSize=10485760 -XX:MaxNewSize=314572
-XX:MinHeapDeltaBytes=524288 -XX:NewSize=3145728 -XX:OldSize=7340032 -XX:+PrintGCDetails -XX:+UseCompressedClassPointer
s -XX:+UseCompressedOops -XX:+UseFastUnorderedTimeStamps -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
Command line:  -Xms10M -Xmx10M -XX:+PrintGCDetails -javaagent:D:\dev\IntelliJ IDEA 2018.3.2\lib\idea_rt.jar=51878:D:\dev
\IntelliJ IDEA 2018.3.2\bin -Dfile.encoding=UTF-8
```

查看java系统参数

```
D:\>jinfo -sysprops 14124
Attaching to process ID 14124, 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 = ;
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:\ideaProjects
java.vm.specification.name = Java Virtual Machine Specification
java.runtime.version = 1.8.0_45-b14
java.awt.graphicsenv = sun.awt.Win32GraphicsEnvironment
os.arch = amd64
java.endorsed.dirs = D:\dev\Java\jdk1.8.0_45\jre\lib\endorsed
```

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    0.612
```

- S0C：第一个幸存区的大小，单位KB
- 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     S0C   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       S0CMX     S0C     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。

阿里巴巴Arthas详解

Arthas 是 Alibaba 在 2018 年 9 月开源的 **Java 诊断**工具。支持 JDK6+， 采用命令行交互模式，可以方便的定位和诊断线上程序运行问题。**Arthas** 官方文档十分详细，详见：*https://alibaba.github.io/arthas*

Arthas使用场景

得益于 **Arthas** 强大且丰富的功能，让 **Arthas** 能做的事情超乎想象。下面仅仅列举几项常见的使用情况，更多的使用场景可以在熟悉了 **Arthas** 之后自行探索。

1. 是否有一个全局视角来查看系统的运行状况?
2. 为什么 CPU 又升高了，到底是哪里占用了 CPU ?
3. 运行的多线程有死锁吗？有阻塞吗?
4. 程序运行耗时很长，是哪里耗时比较长呢？如何监测呢?
5. 这个类从哪个 jar 包加载的？为什么会报各种类相关的 Exception?
6. 我改的代码为什么没有执行到？难道是我没 commit？分支搞错了?
7. 遇到问题无法在线上 debug，难道只能通过加日志再重新发布吗?
8. 有什么办法可以监控到 JVM 的实时运行状态?

Arthas使用

```
# github下载arthas
wget https://alibaba.github.io/arthas/arthas-boot.jar
# 或者 Gitee 下载
wget https://arthas.gitee.io/arthas-boot.jar
```

用java -jar运行即可，可以识别机器上所有Java进程(我们这里之前已经运行了一个Arthas测试程序，代码见下方)

```
[root@localhost local]# java -jar arthas-boot.jar
[INFO] arthas-boot version: 3.3.3
[INFO] Found existing java process, please choose one and input the serial number of the process, eg : 1. Then hit ENTER.
* [1]: 22964 com.tuling.jvm.Arthas
```

```
package com.tuling.jvm;

import java.util.HashSet;

public class Arthas {

 private static HashSet hashSet = new HashSet();

 public static void main(String[] args) {
  // 模拟 CPU 过高
  cpuHigh();
  // 模拟线程死锁
  deadThread();
  // 不断的向 hashSet 集合增加数据
  addHashSetThread();
  }

  /**
  * 不断的向 hashSet 集合添加数据
  */
  public static void addHashSetThread() {
  // 初始化常量
  new Thread(() -> {
  int count = 0;
  while (true) {
  try {
  hashSet.add("count" + count);
  Thread.sleep(1000);
  count++;
  } catch (InterruptedException e) {
  e.printStackTrace();
```

```
    }
    }
    }).start();
    }

    public static void cpuHigh() {
    new Thread(() -> {
    while (true) {

    }
    }).start();
    }

    /**
    * 死锁
    */
    private static void deadThread() {
    /** 创建资源 */
    Object resourceA = new Object();
    Object resourceB = new Object();
    // 创建线程
    Thread threadA = new Thread(() -> {
    synchronized (resourceA) {
    System.out.println(Thread.currentThread() + " get ResourceA");
    try {
    Thread.sleep(1000);
    } catch (InterruptedException e) {
    e.printStackTrace();
    }
    System.out.println(Thread.currentThread() + "waiting get resourceB");
    synchronized (resourceB) {
    System.out.println(Thread.currentThread() + " get resourceB");
    }
    }
    });

    Thread threadB = new Thread(() -> {
    synchronized (resourceB) {
    System.out.println(Thread.currentThread() + " get ResourceB");
    try {
    Thread.sleep(1000);
    } catch (InterruptedException e) {
    e.printStackTrace();
    }
    System.out.println(Thread.currentThread() + "waiting get resourceA");
    synchronized (resourceA) {
    System.out.println(Thread.currentThread() + " get resourceA");
    }
    }
    });
    threadA.start();
    threadB.start();
    }
}
```

选择进程序号1，进入进程信息操作

```
[root@localhost local]# java -jar arthas-boot.jar
[INFO] arthas-boot version: 3.3.3
[INFO] Found existing java process, please choose one and input the serial number of the process, eg : 1. Then hit ENTER.
* [1]: 22964 com.tuling.jvm.Arthas
1
[INFO] arthas home: /root/.arthas/lib/3.3.6/arthas
[INFO] Try to attach process 22964
[INFO] Attach process 22964 success.
[INFO] arthas-client connect 127.0.0.1 3658
  ,---.  ,------. ,--------.,--.  ,--.  ,---.   ,---.
 /  O  \ |  .--. ''--.  .--'|  '--'  | /  O  \ '   .-'
|  .-.  ||  '--'.'    |  |   |  .--.  ||  .-.  |`.  `-.
|  | |  ||  |\  \     |  |   |  |  |  ||  | |  |.-'    |
`--' `--'`--' '--'    `--'   `--'  `--'`--' `--'`-----'


wiki      https://alibaba.github.io/arthas
tutorials https://alibaba.github.io/arthas/arthas-tutorials
version   3.3.6
pid       22964
time      2020-07-03 18:31:10

[arthas@22964]$
```

输入**dashboard**可以查看整个进程的运行情况，线程、内存、GC、运行环境信息：

```
[arthas@22964]$ dashboard
ID         NAME                                  GROUP          PRIORITY   STATE        %CPU   TIME     INTERRUPTED  DAEMON
8          Thread-0                              main           5          RUNNABLE     97     6:53     false        false
24         Timer-for-arthas-dashboard-bfbd5096-4f system         10         RUNNABLE     2      0:0      false        true
13         Attach Listener                       system         9          RUNNABLE     0      0:0      false        true
12         DestroyJavaVM                         main           5          RUNNABLE     0      0:0      false        false
3          Finalizer                             system         8          WAITING      0      0:0      false        true
2          Reference Handler                     system         10         WAITING      0      0:0      false        true
4          Signal Dispatcher                     system         9          RUNNABLE     0      0:0      false        true
9          Thread-1                              main           5          BLOCKED      0      0:0      false        false
10         Thread-2                              main           5          BLOCKED      0      0:0      false        false
11         Thread-3                              main           5          TIMED_WAITIN 0      0:0      false        false
18         arthas-shell-server                   system         9          TIMED_WAITIN 0      0:0      false        true
19         arthas-shell-server                   system         9          TIMED_WAITIN 0      0:0      false        true
15         arthas-timer                          system         9          WAITING      0      0:0      false        true
Memory                  used     total    max      usage    GC
heap                    17M      42M      678M     2.53%    gc.copy.count                    4
eden_space              1M       11M      187M     0.68%    gc.copy.time(ms)                 836
survivor_space          1M       1M       23M      6.15%    gc.marksweepcompact.count        0
tenured_gen             14M      29M      468M     3.12%    gc.marksweepcompact.time(ms)     0
nonheap                 19M      20M      -1       96.46%
code_cache              3M       3M       240M     1.31%
Runtime
os.name                                                     Linux
os.version                                                  3.10.0-957.10.1.el7.x86_64
java.version                                                1.8.0_201
java.home                                                   /usr/local/jdk1.8.0_201/jre
systemload.average                                          1.31
processors                                                  1
uptime                                                      509s
```

输入**thread**可以查看线程详细情况

```
[arthas@22964]$ thread
Threads Total: 17, NEW: 0, RUNNABLE: 8, BLOCKED: 2, WAITING: 4, TIMED_WAITING: 3, TERMINATED: 0
ID         NAME                          GROUP          PRIORITY   STATE        %CPU   TIME     INTERRUPTED  DAEMON
8          Thread-0                      main           5          RUNNABLE     99     10:4     false        false
13         Attach Listener               system         9          RUNNABLE     0      0:0      false        true
12         DestroyJavaVM                 main           5          RUNNABLE     0      0:0      false        false
3          Finalizer                     system         8          WAITING      0      0:0      false        true
2          Reference Handler             system         10         WAITING      0      0:0      false        true
4          Signal Dispatcher             system         9          RUNNABLE     0      0:0      false        true
9          Thread-1                      main           5          BLOCKED      0      0:0      false        false
10         Thread-2                      main           5          BLOCKED      0      0:0      false        false
11         Thread-3                      main           5          TIMED_WAITIN 0      0:0      false        false
18         arthas-shell-server           system         9          TIMED_WAITIN 0      0:0      false        true
19         arthas-shell-server           system         9          TIMED_WAITIN 0      0:0      false        true
15         arthas-timer                  system         9          WAITING      0      0:0      false        true
23         as-command-execute-daemon     system         10         RUNNABLE     0      0:0      false        true
16         nioEventLoopGroup-3-1         system         10         RUNNABLE     0      0:0      false        false
22         nioEventLoopGroup-3-2         system         10         RUNNABLE     0      0:1      false        false
17         nioEventLoopGroup-4-1         system         10         RUNNABLE     0      0:0      false        false
20         pool-1-thread-1               system         5          WAITING      0      0:0      false        false
Affect(row-cnt:0) cost in 121 ms.
```

输入 **thread加上线程ID** 可以查看线程堆栈

```
[arthas@22964]$ thread 8
"Thread-0" Id=8 RUNNABLE
    at com.tuling.jvm.Arthas.lambda$cpuHigh$1(Arthas.java:39)
    at com.tuling.jvm.Arthas$$Lambda$1/471910020.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:748)
```

输入 **thread -b** 可以查看线程死锁

```
[arthas@22964]$ thread -b
"Thread-2" Id=10 BLOCKED on java.lang.Object@7cea2f35 owned by "Thread-1" Id=9
    at com.tuling.jvm.Arthas.lambda$deadThread$3(Arthas.java:78)
    -  blocked on java.lang.Object@7cea2f35
    -  locked java.lang.Object@17dd6f11 <---- but blocks 1 other threads!
    at com.tuling.jvm.Arthas$$Lambda$3/142257191.run(Unknown Source)
    at java.lang.Thread.run(Thread.java:748)
```

输入 **jad加类的全名** 可以反编译，这样可以方便我们查看线上代码是否是正确的版本

```
[arthas@22964]$ jad com.tuling.jvm.Arthas

ClassLoader:
+-sun.misc.Launcher$AppClassLoader@4e0e2f2a
  +-sun.misc.Launcher$ExtClassLoader@56ef0ed

Location:
/usr/local/

/*
 * Decompiled with CFR.
 */
package com.tuling.jvm;

import java.util.HashSet;

public class Arthas {
    private static HashSet hashSet = new HashSet();

    public static void addHashSetThread() {
        new Thread(() -> {
            int count = 0;
            while (true) {
                try {
                    while (true) {
                        hashSet.add("count" + count);
                        Thread.sleep(1000L);
                        ++count;
                    }
                }
```

使用 **ognl** 命令可以查看线上系统变量的值，甚至可以修改变量的值

```
[arthas@23164]$ ognl @com.tuling.jvm.Arthas@hashSet
@HashSet[
    @String[count69],
    @String[count68],
    @String[count67],
    @String[count130],
    @String[count122],
    @String[count123],
    @String[count120],
    @String[count121],
    @String[count126],
    @String[count127],
    @String[count124],
    @String[count125],
    @String[count66],
    @String[count65],
[arthas@23164]$ ognl '@com.tuling.jvm.Arthas@hashSet.add("test123")'
@Boolean[true]
```

更多命令使用可以用help命令查看，或查看文档：https://alibaba.github.io/arthas/commands.html#arthas

GC日志详解

对于java应用我们可以通过一些配置把程序运行过程中的gc日志全部打印出来，然后分析gc日志得到关键性指标，分析GC原因，调优JVM参数。

打印GC日志方法，在JVM参数里增加参数，%t 代表时间

```
-Xloggc:./gc-%t.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps  -XX:+PrintGCTimeStamps -XX:+PrintGCCause
-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M
```

Tomcat则直接加在JAVA_OPTS变量里。

如何分析GC日志

运行程序加上对应gc日志

```
java -jar -Xloggc:./gc-%t.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+PrintGCCause
```

```
-XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M microservice-eureka-server.jar
```

下图中是我截取的JVM刚启动的一部分GC日志

```
Java HotSpot(TM) 64-Bit Server VM (25.45-b02) for windows-amd64 JRE (1.8.0_45-b14), built on Apr 10 2015 10:34:15 by "java_re" with MS VC++ 1
VS2010)
Memory: 4k page, physical 16658532k(8816064k free), swap 19148900k(7122820k free)
CommandLine flags: -XX:InitialHeapSize=266536512 -XX:MaxHeapSize=4264584192 -XX:+PrintGC -XX:+PrintGCDateStamps -XX:+PrintGCDetails -
XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC
2019-07-03T17:28:24.889+0800: 0.613: [GC (Allocation Failure) [PSYoungGen: 65536K->3872K(76288K)] 65536K->3888K(251392K), 0.0042006 secs] [T
user=0.00 sys=0.00, real=0.00 secs]
2019-07-03T17:28:25.087+0800: 0.811: [GC (Allocation Failure) [PSYoungGen: 69408K->4464K(76288K)] 69424K->4488K(251392K), 0.0044453 secs] [T
user=0.00 sys=0.00, real=0.00 secs]
2019-07-03T17:28:25.277+0800: 1.001: [GC (Allocation Failure) [PSYoungGen: 70000K->4934K(76288K)] 70024K->4966K(251392K), 0.0034056 secs] [T
user=0.00 sys=0.00, real=0.00 secs]
2019-07-03T17:28:25.424+0800: 1.148: [GC (Allocation Failure) [PSYoungGen: 70470K->5168K(141824K)] 70502K->5208K(316928K), 0.0034983 secs] [T
user=0.13 sys=0.00, real=0.00 secs]
2019-07-03T17:28:27.180+0800: 2.904: [GC (Metadata GC Threshold) [PSYoungGen: 54010K->6160K(141824K)] 54050K->6272K(316928K), 0.0049121 secs]
Times: user=0.00 sys=0.00, real=0.00 secs]
2019-07-03T17:28:27.185+0800: 2.909: [Full GC (Metadata GC Threshold) [PSYoungGen: 6160K->0K(141824K)] [ParOldGen: 112K->6056K(95744K)] 6272K
(237568K), [Metaspace: 20516K->20516K(1069056K)], 0.0209707 secs] [Times: user=0.03 sys=0.00, real=0.02 secs]
2019-07-03T17:28:29.831+0800: 5.555: [GC (Allocation Failure) [PSYoungGen: 131072K->2528K(209920K)] 137128K->8592K(305664K), 0.0030923 secs]
Times: user=0.00 sys=0.00, real=0.00 secs]
2019-07-03T17:28:30.268+0800: 5.992: [GC (Allocation Failure) [PSYoungGen: 209888K->5524K(264192K)] 215952K->11596K(359936K), 0.0052478 secs]
Times: user=0.13 sys=0.00, real=0.01 secs]
2019-07-03T17:28:31.086+0800: 6.810: [GC (Allocation Failure) [PSYoungGen: 262548K->7136K(334336K)] 268620K->15752K(430080K), 0.0078223 secs]
Times: user=0.00 sys=0.00, real=0.01 secs]
2019-07-03T17:28:32.062+0800: 7.787: [GC (Metadata GC Threshold) [PSYoungGen: 82699K->6956K(336384K)] 91316K->17421K(432128K), 0.0063670 secs
Times: user=0.13 sys=0.00, real=0.01 secs]
2019-07-03T17:28:32.069+0800: 7.793: [Full GC (Metadata GC Threshold) [PSYoungGen: 6956K->0K(336384K)] [ParOldGen: 10465K->16147K(163840K)] 1
16147K(500224K), [Metaspace: 33864K->33864K(1079296K)], 0.1122566 secs] [Times: user=0.74 sys=0.02, real=0.11 secs]
2019-07-03T17:28:36.475+0800: 12.200: [GC (Allocation Failure) [PSYoungGen: 327168K->7784K(398848K)] 343315K->23939K(562688K), 0.0054645 secs
Times: user=0.00 sys=0.00, real=0.01 secs]
2019-07-03T17:28:39.563+0800: 15.287: [GC (Allocation Failure) [PSYoungGen: 398440K->9716K(444416K)] 414595K->27681K(608256K), 0.0088174 secs
Times: user=0.11 sys=0.02, real=0.01 secs]
2019-07-03T17:28:40.607+0800: 16.331: [GC (Allocation Failure) [PSYoungGen: 444404K->9544K(469504K)] 462369K->33090K(633344K), 0.0106355 secs
Times: user=0.08 sys=0.03, real=0.01 secs]
2019-07-03T17:28:44.479+0800: 20.203: [GC (Allocation Failure) [PSYoungGen: 467272K->11871K(470016K)] 490818K->35426K(633856K), 0.0292316 sec
Times: user=0.20 sys=0.03, real=0.03 secs]
```

我们可以看到图中第一行红框，是项目的配置参数。这里不仅配置了打印GC日志，还有相关的VM内存参数。

第二行红框中的是在这个GC时间点发生GC之后相关GC情况。

1、对于**2.909：** 这是从jvm启动开始计算到这次GC经过的时间，前面还有具体的发生时间日期。

2、Full GC(Metadata GC Threshold)指这是一次full gc，括号里是gc的原因， PSYoungGen是年轻代的GC，ParOldGen是老年代的GC，Metaspace是元空间的GC

3、 6160K->0K(141824K)，这三个数字分别对应GC之前占用年轻代的大小，GC之后年轻代占用，以及整个年轻代的大小。

4、112K->6056K(95744K)，这三个数字分别对应GC之前占用老年代的大小，GC之后老年代占用，以及整个老年代的大小。

5、6272K->6056K(237568K)，这三个数字分别对应GC之前占用堆内存的大小，GC之后堆内存占用，以及整个堆内存的大小。

6、20516K->20516K(1069056K)，这三个数字分别对应GC之前占用元空间内存的大小，GC之后元空间内存占用，以及整个元空间内存的大小。

7、0.0209707是该时间点GC总耗费时间。

从日志可以发现几次fullgc都是由于元空间不够导致的，所以我们可以将元空间调大点

```
java -jar -Xloggc:./gc-adjust-%t.log -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+PrintGCDetails -XX:+Print
GCDateStamps
-XX:+PrintGCTimeStamps -XX:+PrintGCCause -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M
microservice-eureka-server.jar
```

调整完我们再看下gc日志发现已经没有因为元空间不够导致的fullgc了

对于CMS和G1收集器的日志会有一点不一样，也可以试着打印下对应的gc日志分析下，可以发现gc日志里面的gc步骤跟我们之前讲过的步骤是类似的

```
public class HeapTest {

 byte[] a = new byte[1024 * 100]; //100KB

```

```
5  public static void main(String[] args) throws InterruptedException {
6  ArrayList<HeapTest> heapTests = new ArrayList<>();
7  while (true) {
8  heapTests.add(new HeapTest());
9  Thread.sleep(10);
10  }
11  }
12 }
```

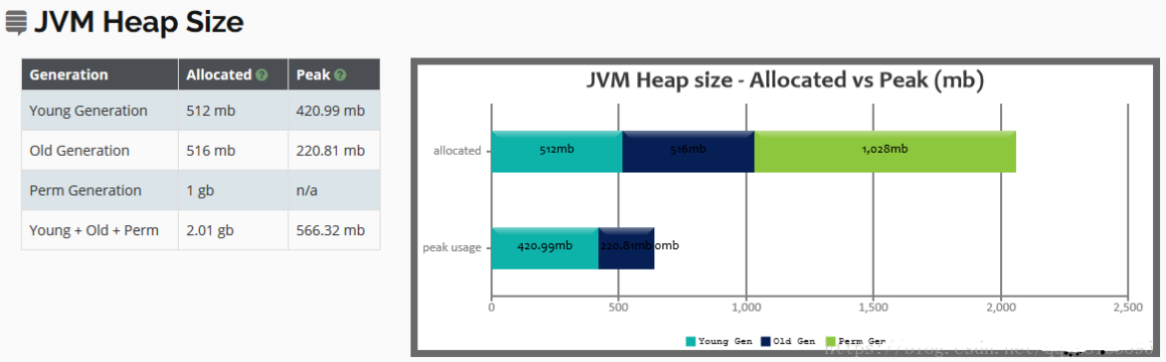
CMS

```
1 -Xloggc:d:/gc-cms-%t.log -Xms50M -Xmx50M -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+PrintGCDetails -XX:+P
rintGCDateStamps
2  -XX:+PrintGCTimeStamps -XX:+PrintGCCause -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M
3  -XX:+UseParNewGC -XX:+UseConcMarkSweepGC
```

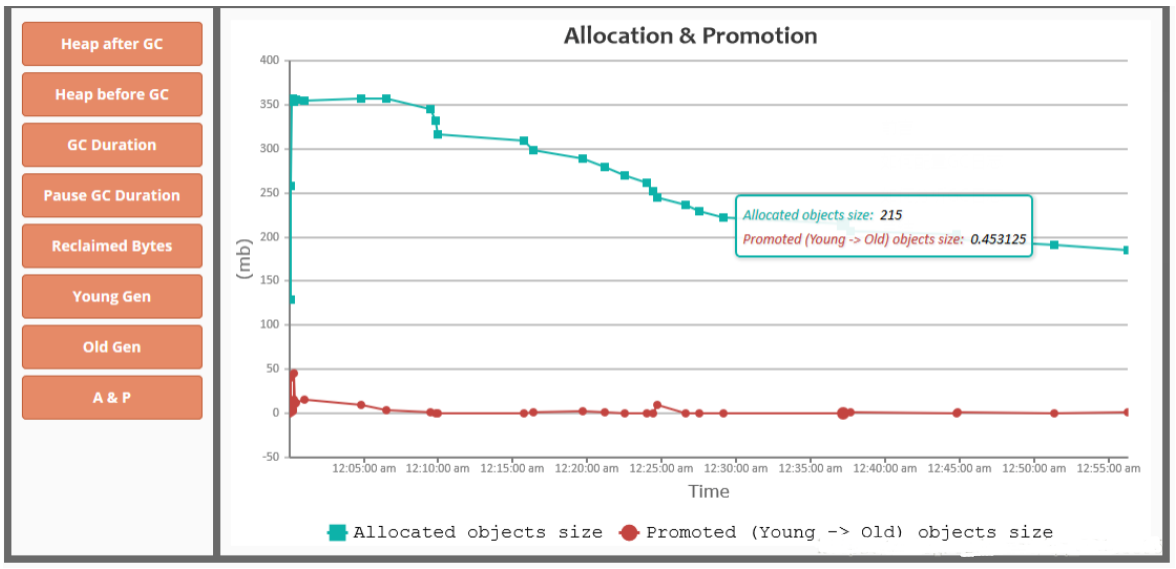
G1

```
1 -Xloggc:d:/gc-g1-%t.log -Xms50M -Xmx50M -XX:MetaspaceSize=256M -XX:MaxMetaspaceSize=256M -XX:+PrintGCDetails -XX:+Pr
intGCDateStamps
2  -XX:+PrintGCTimeStamps -XX:+PrintGCCause -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=10 -XX:GCLogFileSize=100M
-XX:+UseG1GC
```

上面的这些参数，能够帮我们查看分析GC的垃圾收集情况。但是如果GC日志很多很多，成千上万行。就算你一目十行，看完了，脑子也是一片空白。所以我们可以借助一些功能来帮助我们分析，这里推荐一个gceasy(https://gceasy.io)，可以上传gc文件，然后他会利用可视化的界面来展现GC情况。具体下图所示

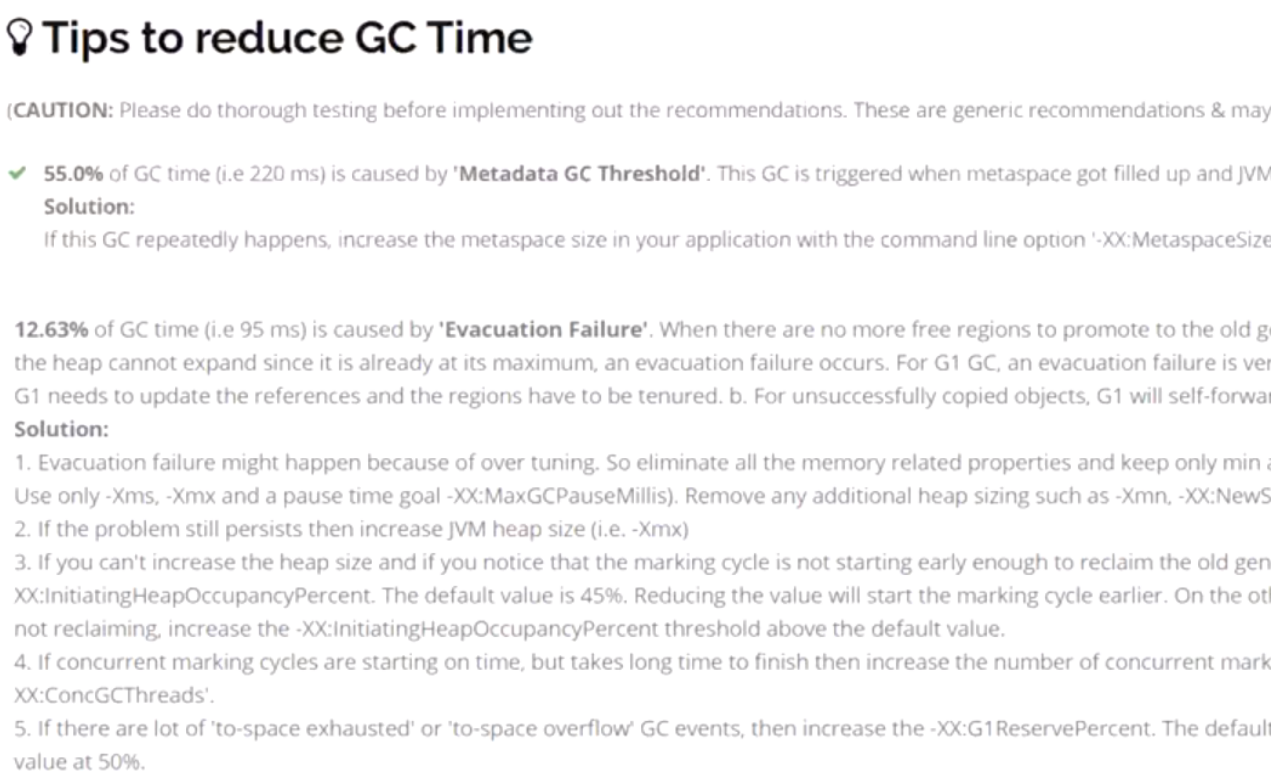


上图我们可以看到年轻代，老年代，以及永久代的内存分配，和最大使用情况。



上图我们可以看到堆内存在GC之前和之后的变化，以及其他信息。

这个工具还提供基于机器学习的JVM智能优化建议，当然现在这个功能需要付费



JVM参数汇总查看命令

java -XX:+PrintFlagsInitial 表示打印出所有参数选项的默认值

java -XX:+PrintFlagsFinal 表示打印出所有参数选项在运行程序时生效的值