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

```
# 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:+PrintGCDateStamps
-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 }
```

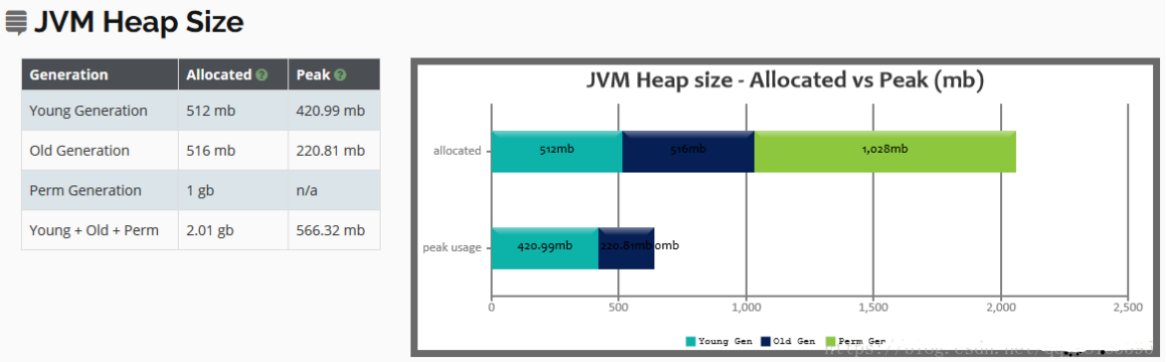
CMS

```
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
```

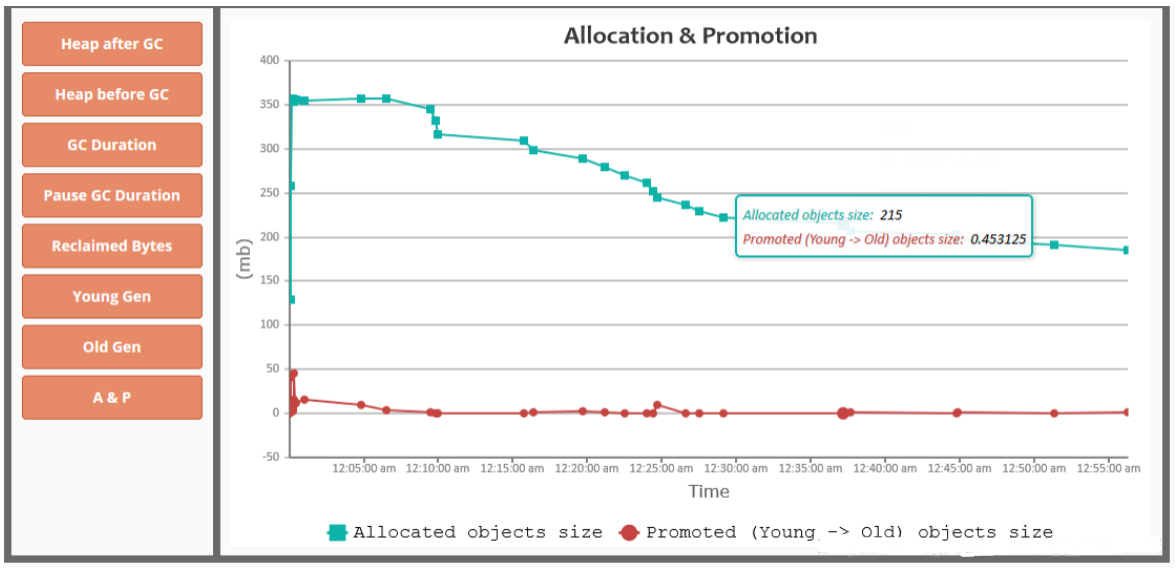
G1

```
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

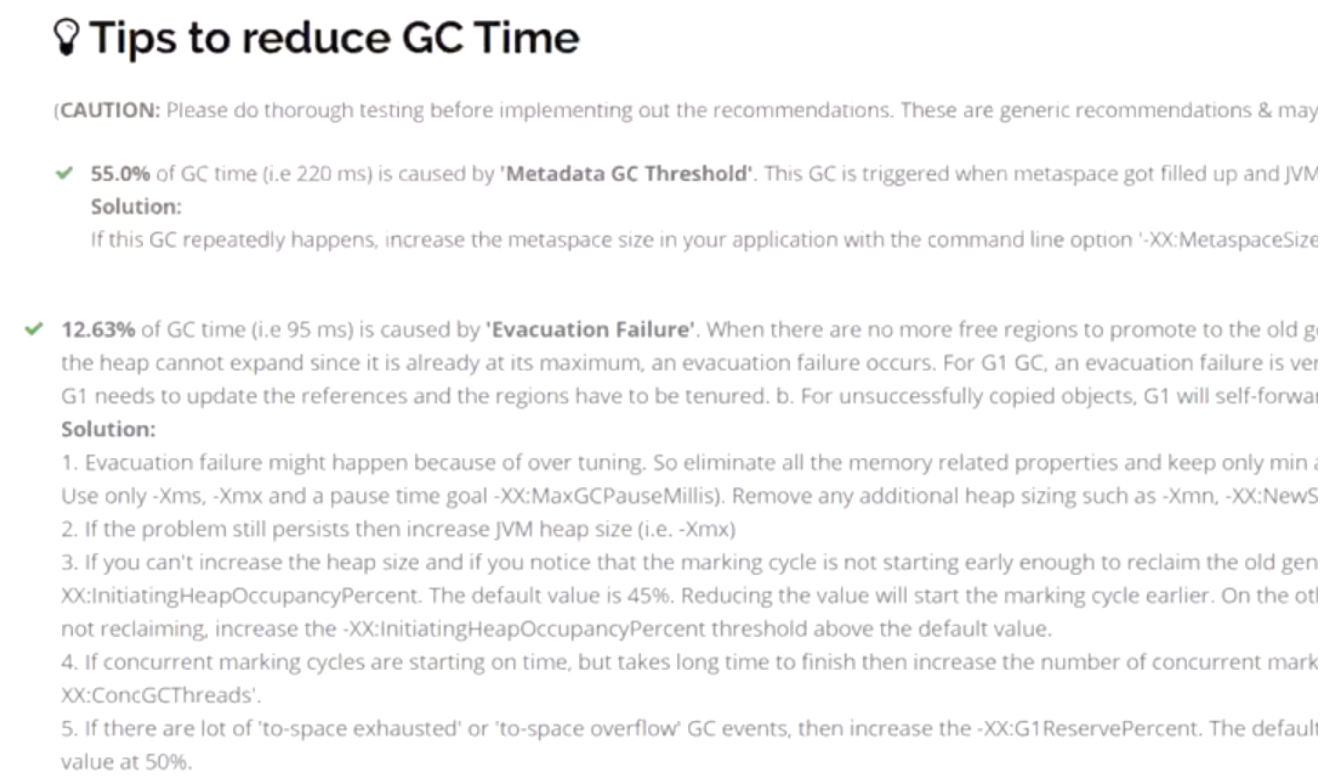


上图我们可以看到年轻代，老年代，以及永久代的内存分配，和最大使用情况。



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

Class常量池与运行时常量池

Class常量池可以理解为是Class文件中的资源仓库。 Class文件中除了包含类的版本、字段、方法、接口等描述信息外，还有一项信息就是**常量池(constant pool table)**，用于存放编译期生成的各种**字面量(Literal)和符号引用(Symbolic References)**。

一个class文件的16进制大体结构如下图：

Math.class

```
1  cafe babe 0000 0034 0042 0a00 0c00 2c0a
2  002d 002e 0900 2f00 300a 0031 0032 0700
3  330a 0005 002c 0a00 0500 3409 0005 0035
4  0700 360a 0009 002c 0900 0500 3707 0038
5  0100 0869 6e69 7444 6174 6101 0001 4901
6  0004 7573 6572 0100 154c 636f 6d2f 7475
7  6c69 6e67 2f6a 766d 2f55 7365 723b 0100
8  063c 696e 6974 3e01 0003 2829 5601 0004
9  436f 6465 0100 0f4c 696e 654e 756d 6265
0  7254 6162 6c65 0100 124c 6f63 616c 5661
1  7269 6162 6c65 5461 626c 6501 0004 7468
2  6973 0100 154c 636f 6d2f 7475 6c69 6e67
3  2f6a 766d 2f4d 6174 683b 0100 0763 6f6d
4  7075 7465 0100 0328 2949 0100 0161 0100
5  0162 0100 0163 0100 046d 6169 6e01 0016
6  285b 4c6a 6176 612f 6c61 6e67 2f53 7472
7  696e 673b 2956 0100 0461 7267 7301 0013
8  5b4c 6a61 7661 2f6c 616e 672f 5374 7269
```

对应的含义如下，细节可以查下oracle官方文档

```
cafe babe    0000     0034                    0042          0a00 0ceo 2c0a。。
魔数         次版本   主版本(十进制52,JDK1.8)  常量池计数器   常量池数据区与代码区
```

当然我们一般不会去人工解析这种16进制的字节码文件，我们一般可以通过javap命令生成更可读的JVM字节码指令文件：

javap -v Math.class

```
Classfile /D:/ideaProjects/project-all/target/classes/com/tuling/jvm/Math.class
  Last modified 2019-9-22; size 1103 bytes
  MD5 checksum 5e638558e268be4e71e92337b5071405
  Compiled from "Math.java"
public class com.tuling.jvm.Math
  minor version: 0
  major version: 52
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
   #1 = Methodref          #12.#44        // java/lang/Object."<init>":()V
   #2 = Methodref          #45.#46        // java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   #3 = Fieldref           #47.#48        // java/lang/System.out:Ljava/io/PrintStream;
   #4 = Methodref          #49.#50        // java/io/PrintStream.println:(Z)V
   #5 = Class              #51            // com/tuling/jvm/Math
   #6 = Methodref          #5.#44         // com/tuling/jvm/Math."<init>":()V
   #7 = Methodref          #5.#52         // com/tuling/jvm/Math.compute:()I
   #8 = Fieldref           #5.#53         // com/tuling/jvm/Math.initData:I
   #9 = Class              #54            // com/tuling/jvm/User
  #10 = Methodref          #9.#44         // com/tuling/jvm/User."<init>":()V
  #11 = Fieldref           #5.#55         // com/tuling/jvm/Math.user:Lcom/tuling/jvm/User;
  #12 = Class              #56            // java/lang/Object
  #13 = Utf8               initData
  #14 = Utf8               I
  #15 = Utf8               user
  #16 = Utf8               Lcom/tuling/jvm/User;
  #17 = Utf8               <init>
  #18 = Utf8               ()V
  #19 = Utf8               Code
  #20 = Utf8               LineNumberTable
  #21 = Utf8               LocalVariableTable
  #22 = Utf8               this
  #23 = Utf8               Lcom/tuling/jvm/Math;
  #24 = Utf8               compute
  #25 = Utf8               ()I
  #26 = Utf8               a
  #27 = Utf8               b
  #28 = Utf8               c
```

红框标出的就是class常量池信息，常量池中主要存放两大类常量：**字面量和符号引用。**

字面量

字面量就是指由字母、数字等构成的字符串或者数值常量

字面量只可以右值出现，所谓右值是指等号右边的值，如：int a=1 这里的a为左值，1为右值。在这个例子中1就是字面量。

```
1 int a = 1;
2 int b = 2;
3 int c = "abcdefg";
4 int d = "abcdefg";
```

符号引用

符号引用是编译原理中的概念，是相对于直接引用来说的。主要包括了以下三类常量：

- 类和接口的全限定名
- 字段的名称和描述符
- 方法的名称和描述符

上面的a，b就是字段名称，就是一种符号引用，还有Math类常量池里的 Lcom/tuling/jvm/Math 是类的全限定名，main和compute是方法名称，()是一种UTF8格式的描述符，这些都是符号引用。

这些常量池现在是静态信息，只有到运行时被加载到内存后，这些符号才有对应的内存地址信息，这些常量池一旦被装入内存就变成**运行时常量池**，对应的符号引用在程序加载或运行时会被转变为被加载到内存区域的代码的直接引用，也就是我们说的**动态链接了。例如，compute()这个符号引用在运行时就会被转变为compute()方法具体代码在内存中的地址，主要通过对象头里的类型指针去转换直接引用。**

字符串常量池

字符串常量池的设计思想

1. 字符串的分配，和其他的对象分配一样，耗费高昂的时间与空间代价，作为最基础的数据类型，大量频繁的创建字符串，极大程度地影响程序的性能
2. JVM为了提高性能和减少内存开销，在实例化字符串常量的时候进行了一些优化

- 为字符串开辟一个字符串常量池，类似于缓存区
- 创建字符串常量时，首先查询字符串常量池是否存在该字符串
- 存在该字符串，返回引用实例，不存在，实例化该字符串并放入池中

三种字符串操作(Jdk1.7 及以上版本)

- 直接赋值字符串

```
String s = "zhuge"; // s指向常量池中的引用
```

这种方式创建的字符串对象，只会在常量池中。
因为有"zhuge"这个字面量，创建对象s的时候，JVM会先去常量池中通过 equals(key) 方法，判断是否有相同的对象
如果有，则直接返回该对象在常量池中的引用；
如果没有，则会在常量池中创建一个新对象，再返回引用。

- new String();

```
String s1 = new String("zhuge"); // s1指向内存中的对象引用
```

这种方式会保证字符串常量池和堆中都有这个对象，没有就创建，最后返回堆内存中的对象引用。
步骤大致如下：
因为有"zhuge"这个字面量，所以会先检查字符串常量池中是否存在字符串"zhuge"
不存在，先在字符串常量池里创建一个字符串对象；再去内存中创建一个字符串对象"zhuge"；
存在的话，就直接去堆内存中创建一个字符串对象"zhuge"；
最后，将内存中的引用返回。

- intern方法

```
String s1 = new String("zhuge");
String s2 = s1.intern();

System.out.println(s1 == s2); //false
```

String中的intern方法是一个 native 的方法，当调用 intern方法时，如果池已经包含一个等于此String对象的字符串（用equals(oject)方法确定），则返回池中的字符串。**否则，将intern返回的引用指向当前字符串 s1(jdk1.6版本需要将s1 复制到字符串常量池里)**。

字符串常量池位置

Jdk1.6及之前： 有永久代, 运行时常量池在永久代，运行时常量池包含字符串常量池
Jdk1.7：有永久代，但已经逐步“去永久代”，字符串常量池从永久代里的运行时常量池分离到堆里
Jdk1.8及之后： 无永久代，运行时常量池在元空间，字符串常量池里依然在堆里
用一个程序证明下字符串常量池在哪里：

```
/**
 * jdk6: -Xms6M -Xmx6M -XX:PermSize=6M -XX:MaxPermSize=6M
 * jdk8: -Xms6M -Xmx6M -XX:MetaspaceSize=6M -XX:MaxMetaspaceSize=6M
 */
public class RuntimeConstantPoolOOM{
 public static void main(String[] args) {
 ArrayList<String> list = new ArrayList<String>();
 for (int i = 0; i < 10000000; i++) {
 String str = String.valueOf(i).intern();
  list.add(str);
  }
  }
}
```

```
14
15 运行结果：
16 jdk7及以上：Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
17 jdk6：Exception in thread "main" java.lang.OutOfMemoryError: PermGen space
```

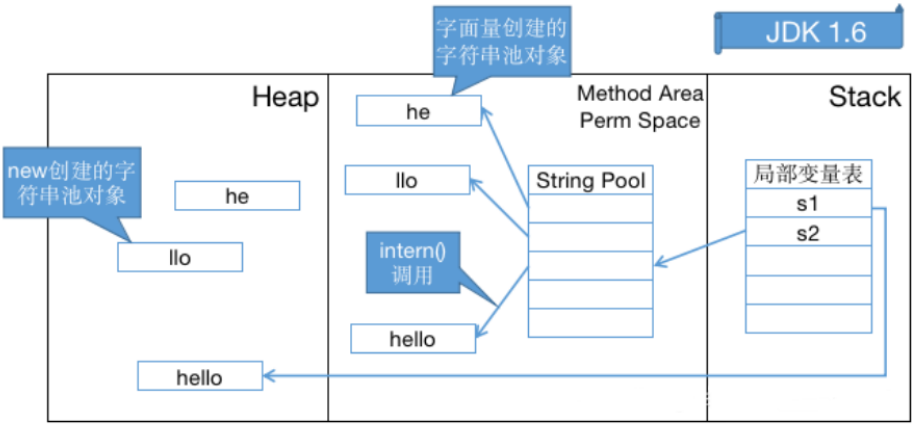
字符串常量池设计原理

字符串常量池底层是hotspot的C++实现的，底层类似一个 HashTable， 保存的本质上是字符串对象的引用。看一道比较常见的面试题，下面的代码创建了多少个 String 对象?

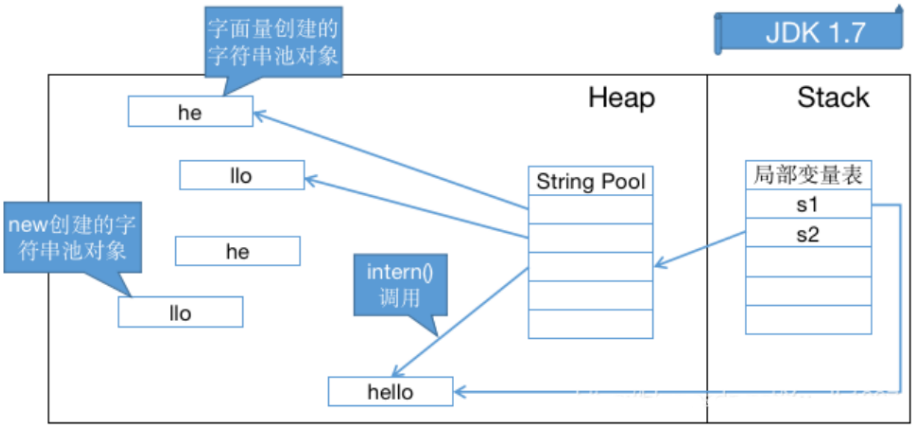
```
1 String s1 = new String("he") + new String("llo");
2 String s2 = s1.intern();
3
4 System.out.println(s1 == s2);
5 // 在 JDK 1.6 下输出是 false，创建了 6 个对象
6 // 在 JDK 1.7 及以上的版本输出是 true，创建了 5 个对象
7 // 当然我们这里没有考虑GC，但这些对象确实存在或存在过
```

为什么输出会有这些变化呢？主要还是字符串池从永久代中脱离、移入堆区的原因， intern() 方法也相应发生了变化：

1、在 JDK 1.6 中，调用 intern() 首先会在字符串池中寻找 equal() 相等的字符串，假如字符串存在就返回该字符串在字符串池中的引用；假如字符串不存在，虚拟机会重新在永久代上创建一个实例，将 StringTable 的一个表项指向这个新创建的实例。



2、在 JDK 1.7 (及以上版本)中，由于字符串池不在永久代了，intern() 做了一些修改，更方便地利用堆中的对象。字符串存在时和 JDK 1.6一样，但是字符串不存在时不再需要重新创建实例，可以直接指向堆上的实例。



由上面两个图，也不难理解为什么 JDK 1.6 字符串池溢出会抛出 OutOfMemoryError: PermGen space ，而在 JDK 1.7 及以上版本抛出 OutOfMemoryError: Java heap space 。

String常量池问题的几个例子

示例1：

```
String s0="zhuge";
String s1="zhuge";
String s2="zhu" + "ge";
System.out.println( s0==s1 ); //true
System.out.println( s0==s2 ); //true
```

分析：因为例子中的 s0和s1中的”zhuge”都是字符串常量，它们在编译期就被确定了，所以s0==s1为true；
而”zhu”和”ge”也都是字符串常量，当一个字 符串由多个字符串常量连接而成时，它自己肯定也是字符串常量，所以s2也同样在编译期就被优化为一个字符串常量"zhuge"，所以s2也是常量池中” zhuge”的一个引用。所以我们得出s0==s1==s2；

示例2：

```
String s0="zhuge";
String s1=new String("zhuge");
String s2="zhu" + new String("ge");
System.out.println( s0==s1 );    // false
System.out.println( s0==s2 );    // false
System.out.println( s1==s2 );    // false
```

分析：用new String() 创建的字符串不是常量，不能在编译期就确定，所以new String() 创建的字符串不放入常量池中，它们有自己的地址空间。
s0还是常量池 中"zhuge”的引用，s1因为无法在编译期确定，所以是运行时创建的新对象”zhuge”的引用，s2因为有后半部分 new String(”ge”)所以也无法在编译期确定，所以也是一个新创建对象”zhuge”的引用;明白了这些也就知道为何得出此结果了。

示例3：

```
 String a = "a1";
 String b = "a" + 1;
 System.out.println(a == b); // true

 String a = "atrue";
 String b = "a" + "true";
 System.out.println(a == b); // true

 String a = "a3.4";
  String b = "a" + 3.4;
  System.out.println(a == b); // true
```

分析：JVM对于字符串常量的"+"号连接，将在程序编译期，JVM就将常量字符串的"+"连接优化为连接后的值，拿"a" + 1来说，经编译器优化后在class中就已经是a1。在编译期其字符串常量的值就确定下来，故上面程序最终的结果都为true。

示例4：

```
String a = "ab";
String bb = "b";
String b = "a" + bb;

System.out.println(a == b); // false
```

分析：JVM对于字符串引用，由于在字符串的"+"连接中，有字符串引用存在，而引用的值在程序编译期是无法确定的，即"a" + bb无法被编译器优化，只有在程序运行期来动态分配并将连接后的新地址赋给b。所以上面程序的结果也就为false。

示例5：

```
String a = "ab";
final String bb = "b";
String b = "a" + bb;

System.out.println(a == b); // true
```

分析：和示例4中唯一不同的是bb字符串加了final修饰，对于final修饰的变量，它在编译时被解析为常量值的一个本地拷贝存储到自己的常量池中或嵌入到它的字节码流中。所以此时的"a" + bb和"a" + "b"效果是一样的。故上面程序的结果为true。

示例6：

```
String a = "ab";
final String bb = getBB();
String b = "a" + bb;

System.out.println(a == b); // false

private static String getBB()
{
 return "b";
 }
```

分析：JVM对于字符串引用bb，它的值在编译期无法确定，只有在程序运行期调用方法后，将方法的返回值和"a"来动态连接并分配地址为b，故上面 程序的结果为false。

关于String是不可变的

通过上面例子可以得出得知：

```
String  s  =  "a" + "b" + "c"; //就等价于String s = "abc";
String  a  =  "a";
String  b  =  "b";
String  c  =  "c";
String  s1  =   a  +  b  +  c;
```

s1 这个就不一样了，可以通过观察其**JVM指令码**发现s1的"+"操作会变成如下操作：

```
StringBuilder temp = new StringBuilder();
temp.append(a).append(b).append(c);
String s = temp.toString();
```

最后再看一个例子：

```
//字符串常量池："计算机"和"技术" 堆内存：str1引用的对象"计算机技术"
//堆内存中还有个StringBuilder的对象，但是会被gc回收，StringBuilder的toString方法会new String()，这个String才是真正返回的对象引用
String str2 = new StringBuilder("计算机").append("技术").toString(); //没有出现"计算机技术"字面量，所以不会在常量池里生成"计算机技术"对象
System.out.println(str2 == str2.intern()); //true
//"计算机技术" 在池中没有，但是在heap中存在，则intern时，会直接返回该heap中的引用

//字符串常量池："ja"和"va" 堆内存：str1引用的对象"java"
//堆内存中还有个StringBuilder的对象，但是会被gc回收，StringBuilder的toString方法会new String()，这个String才是真正返回的对象引用
String str1 = new StringBuilder("ja").append("va").toString(); //没有出现"java"字面量，所以不会在常量池里生成"java"对象
 System.out.println(str1 == str1.intern()); //false
 //java是关键字，在JVM初始化的相关类里肯定早就放进字符串常量池了

 String s1=new String("test");
 System.out.println(s1==s1.intern()); //false
 //"test"作为字面量，放入了池中，而new时s1指向的是heap中新生成的string对象，s1.intern()指向的是"test"字面量之前在池中生成的字符串对象
```

```
16
17 String s2=new StringBuilder("abc").toString();
18 System.out.println(s2==s2.intern()); //false
19 //同上
```

八种基本类型的包装类和对象池

java中基本类型的包装类的大部分都实现了常量池技术(严格来说应该叫**对象池**，在堆上)，这些类是Byte,Short,Integer,Long,Character,Boolean,另外两种浮点数类型的包装类则没有实现。另外Byte,Short,Integer,Long,Character这5种整型的包装类也只是在对应值小于等于127时才可使用对象池，也即对象不负责创建和管理大于127的这些类的对象。因为一般这种比较小的数用到的概率相对较大。

```
1  public class Test {
2
3   public static void main(String[] args) {
4   //5种整形的包装类Byte,Short,Integer,Long,Character的对象，
5   //在值小于127时可以使用对象池
6   Integer i1 = 127; //这种调用底层实际是执行的Integer.valueOf(127)，里面用到了IntegerCache对象池
7   Integer i2 = 127;
8   System.out.println(i1 == i2);//输出true
9
10   //值大于127时，不会从对象池中取对象
11   Integer i3 = 128;
12   Integer i4 = 128;
13   System.out.println(i3 == i4);//输出false
14
15   //用new关键词新生成对象不会使用对象池
16   Integer i5 = new Integer(127);
17   Integer i6 = new Integer(127);
18   System.out.println(i5 == i6);//输出false
19
20   //Boolean类也实现了对象池技术
21   Boolean bool1 = true;
22   Boolean bool2 = true;
23   System.out.println(bool1 == bool2);//输出true
24
25   //浮点类型的包装类没有实现对象池技术
26   Double d1 = 1.0;
27   Double d2 = 1.0;
28   System.out.println(d1 == d2);//输出false
29   }
30 }
```

文档：08-VIP-JVM调优实战及常量池详解

```
1 http://note.youdao.com/noteshare?id=d8d6dc3589ffd9245d97bb7aa91af835&sub=AD9BB50E218B455B9327FC86856F9942
```