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

```
-XX:+PrintGCDetails  -XX:+PrintGCTimeStamps  -XX:+PrintGCDateStamps  -Xloggc:./gc.log
```

Tomcat则直接加在JAVA_OPTS变量里。

如何分析GC日志

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 这是具体发生GC的时间点。这是时间戳是从jvm启动开始计算的，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。

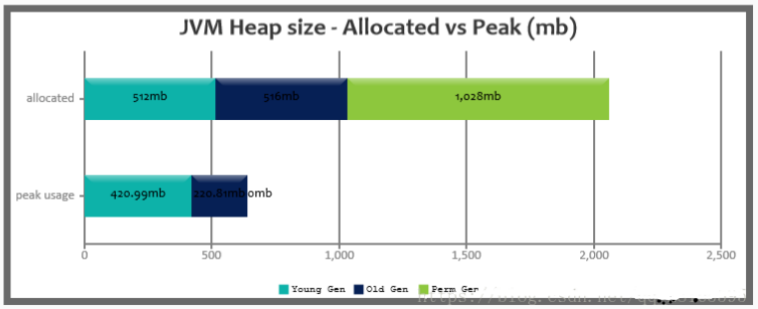
6、20516K->20516K(1069056K)，这三个数字分别对应GC之前占用元空间内存的大小，GC之后元空间内存占用，以及整个元空间内存的大小。

7、0.0209707是该时间点GC总耗费时间。

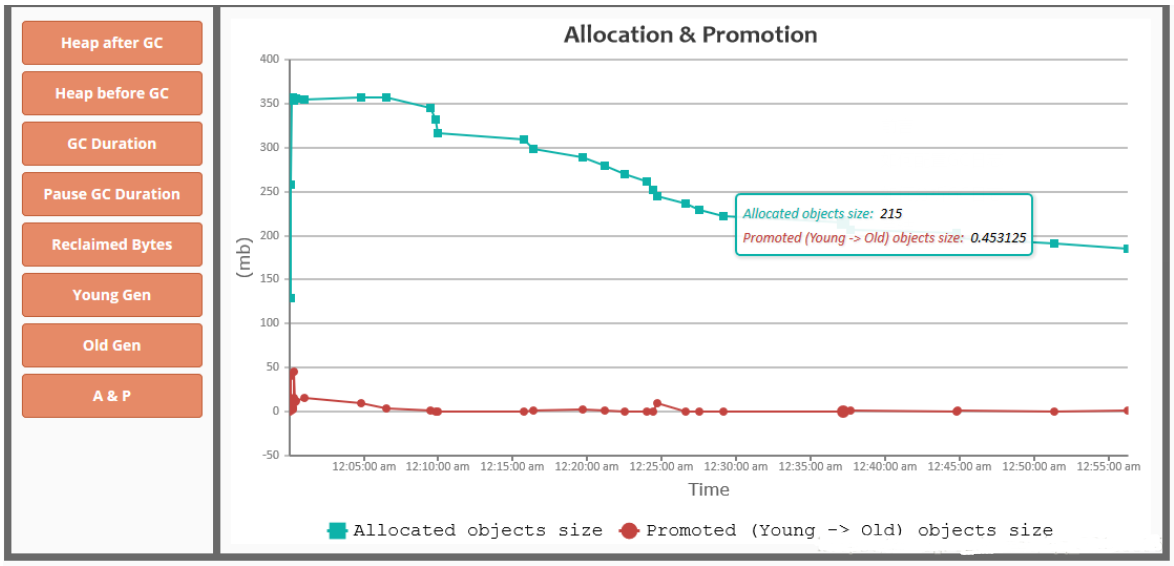
上面的这些参数，能够帮我们查看分析GC的垃圾收集情况。但是如果GC日志很多很多，成千上万行。就算你一目十行，看完了，脑子也是一片空白。所以我们可以借助一些功能来帮助我们分析，这里推荐一个gceasy(https://gceasy.io)，可以上传gc文件，然后他会利用可视化的界面来展现GC情况。具体下图所示

JVM Heap Size

Generation	Allocated	Peak
Young Generation	512 mb	420.99 mb
Old Generation	516 mb	220.81 mb
Perm Generation	1 gb	n/a
Young + Old + Perm	2.01 gb	566.32 mb



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

Tips to reduce GC Time

(**CAUTION:** Please do thorough testing before implementing out the recommendations. These are generic recommendations & may

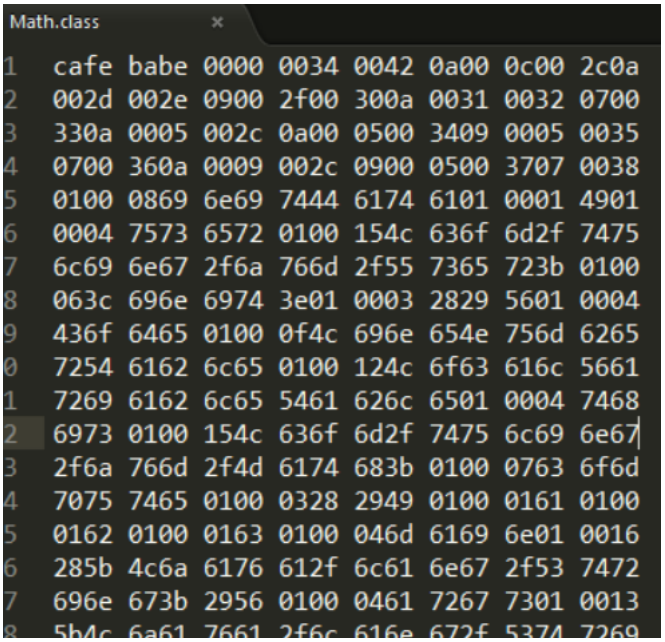
✔ **55.0%** of GC time (i.e 220 ms) is caused by '**Metadata GC Threshold**'. This GC is triggered when metaspace got filled up and JVM
Solution:
If this GC repeatedly happens, increase the metaspace size in your application with the command line option '-XX:MetaspaceSize

✔ **12.63%** of GC time (i.e 95 ms) is caused by '**Evacuation Failure**'. When there are no more free regions to promote to the old g
the heap cannot expand since it is already at its maximum, an evacuation failure occurs. For G1 GC, an evacuation failure is ver
G1 needs to update the references and the regions have to be tenured. b. For unsuccessfully copied objects, G1 will self-forwar
Solution:
1. Evacuation failure might happen because of over tuning. So eliminate all the memory related properties and keep only min a
Use only -Xms, -Xmx and a pause time goal -XX:MaxGCPauseMillis). Remove any additional heap sizing such as -Xmn, -XX:NewS
2. If the problem still persists then increase JVM heap size (i.e. -Xmx)
3. If you can't increase the heap size and if you notice that the marking cycle is not starting early enough to reclaim the old gen
XX:InitiatingHeapOccupancyPercent. The default value is 45%. Reducing the value will start the marking cycle earlier. On the otl
not reclaiming, increase the -XX:InitiatingHeapOccupancyPercent threshold above the default value.
4. If concurrent marking cycles are starting on time, but takes long time to finish then increase the number of concurrent mark
XX:ConcGCThreads'.
5. If there are lot of 'to-space exhausted' or 'to-space overflow' GC events, then increase the -XX:G1ReservePercent. The default
value at 50%.

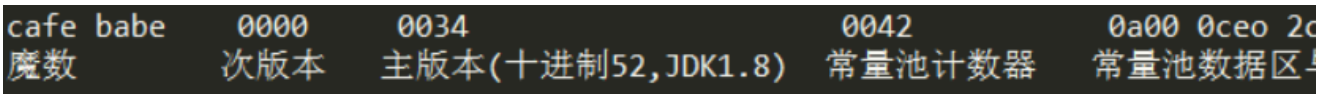
Class常量池

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)，用于存放编译器生成的各种**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：

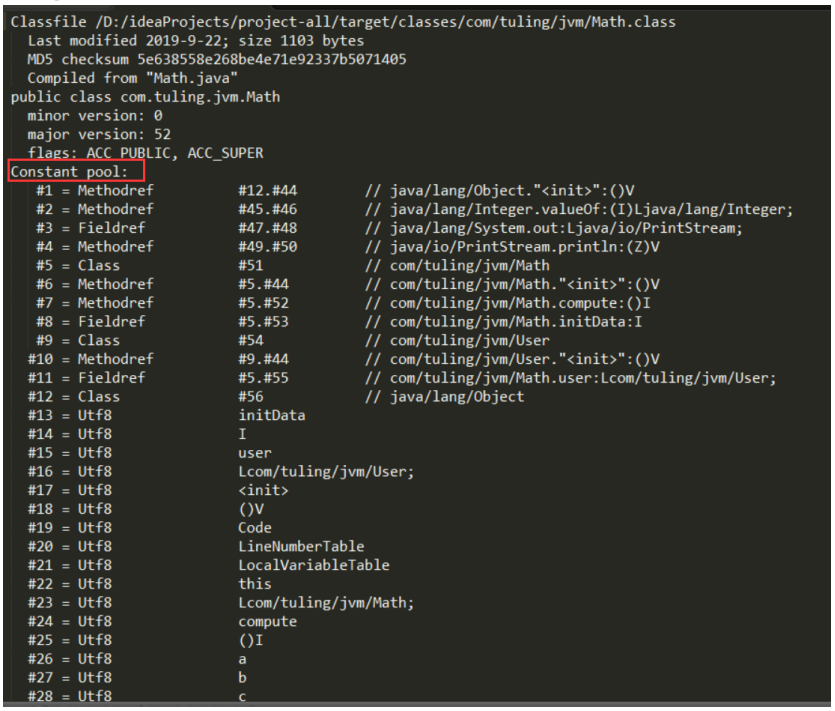


对应的含义如下，细节可以查下oracle官方文档



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



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。

```
int a = 1;
int b = 2;
int c = "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**了，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。**

Jdk1.6及之前： 有永久代, 常量池在方法区

Jdk1.7：有永久代，但已经逐步“去永久代”，常量池在堆

Jdk1.8及之后： 无永久代，常量池在元空间

字符串常量池

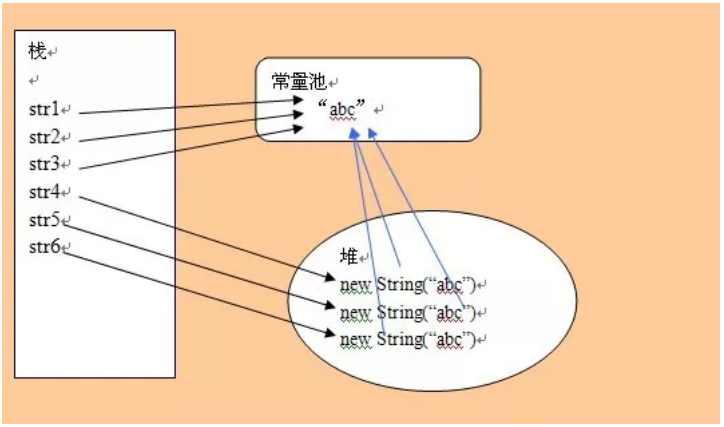
字符串常量池的设计思想

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，首先坚持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

代码示例，一些字符串局部变量操作

```
String str1 = "abc";
String str2 = "abc";
String str3 = "abc";
String str4 = new String("abc");
String str5 = new String("abc");
String str5 = new String("abc");
```



面试题：String str4 = new String(“abc”) 创建多少个对象?

1. 在常量池中查找是否有“abc”对象

- 有则返回对应的引用实例
- 没有则在常量池中创建对应的实例对象

2. 在堆中 new 一个 String("abc") 对象

3. 将对象地址赋值给str4，创建一个引用

所以，常量池中没有“abc”字面量则创建两个对象，否则创建一个对象，以及创建一个引用

根据字面量，往往会提出这样的变式题：

String str1 = new String("A"+"B")；会创建多少个对象?

String str2 = new String("ABC") + "ABC"；会创建多少个对象?

str1：

字符串常量池："A","B","AB"：3个

堆：new String("AB")： 1个

引用： str1 ： 1个

总共： 5个

str2：

字符串常量池："ABC"：1个

堆：new String("ABC")： 1个

引用： str2 ： 1个

总共： 3个

操作字符串常量池的方式

- JVM实例化字符串常量池时

```
String str1 = "hello";
String str2 = "hello";
System.out.println("str1 == str2" : str1 == str2) //true
```

- String.intern()

通过new操作符创建的字符串对象不指向字符串常量池中的任何对象，但是可以通过使用字符串的intern()方法来指向其中的某一个。java.lang.String.intern()返回一个常量池里面的字符串，就是一个在字符串常量池中有了一个入口。如果以前没有在字符串常量池中，那么它就会被添加到里面。

```
String s1 = "Hello";
String s2 = new String("Hello");
String s3 = s2.intern();

System.out.println("s1 == s2? " + (s1 == s2)); // false
System.out.println("s1 == s3? " + (s1 == s3)); // true
```

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，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。

```
public class Test {

 public static void main(String[] args) {
 //5种整形的包装类Byte,Short,Integer,Long,Character的对象，
 //在值小于127时可以使用常量池
 Integer i1 = 127;
 Integer i2 = 127;
 System.out.println(i1 == i2);//输出true

 //值大于127时，不会从常量池中取对象
 Integer i3 = 128;
 Integer i4 = 128;
 System.out.println(i3 == i4);//输出false

 //Boolean类也实现了常量池技术
 Boolean bool1 = true;
 Boolean bool2 = true;
 System.out.println(bool1 == bool2);//输出true

 //浮点类型的包装类没有实现常量池技术
 Double d1 = 1.0;
 Double d2 = 1.0;
 System.out.println(d1 == d2);//输出false
 }
}

```

安全点与安全区域

安全点就是指代码中一些特定的位置,当线程运行到这些位置时它的状态是确定的,这样JVM就可以安全的进行一些操作,比如GC等，所以GC不是想什么时候做就立即触发的，是需要等待所有线程运行到安全点后才能触发。

这些特定的安全点位置主要有以下几种:

1. 方法返回之前
2. 调用某个方法之后
3. 抛出异常的位置
4. 循环的末尾

安全区域又是什么?

Safe Point 是对正在执行的线程设定的。

如果一个线程处于 Sleep 或中断状态，它就不能响应 JVM 的中断请求，再运行到 Safe Point 上。

因此 JVM 引入了 Safe Region。

Safe Region 是指在一段代码片段中，**引用关系不会发生变化。**在这个区域内的任意地方开始 GC 都是安全的。

线程在进入 Safe Region 的时候先标记自己已进入了 Safe Region，等到被唤醒时准备离开 Safe Region 时，先检查能否离开，如果 GC 完成了，那么线程可以离开，否则它必须等待直到收到安全离开的信号为止。