

1、并行 GC

1.1、默认参数执行 `java -XX:+PrintGCDetails GCLogAnalysis`

默认参数下，分配的堆内存是物理内存的 1/4。

- 随着 GC 次数的增加，年轻代、老年代占用的大小不断增加（两者之和为堆的大小）；
- 每次 Young GC 会有一部分对象进入老年区，可以从下面的日志得到

```
[GC (Allocation Failure) [PSYoungGen: 32879K->5094K(38400K)] 32879K->10641K(125952K), 0.0032114 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 38374K->5112K(71680K)] 43921K->20523K(159232K), 0.0035912 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 71672K->5106K(71680K)] 87083K->38776K(159232K), 0.0063285 secs] [Times: user=0.02 sys=0.03, real=0.02 secs]
[GC (Allocation Failure) [PSYoungGen: 71666K->5118K(138240K)] 105336K->61840K(225792K), 0.0071410 secs] [Times: user=0.00 sys=0.06, real=0.02 secs]
[GC (Allocation Failure) [PSYoungGen: 138153K->5118K(138240K)] 194874K->102649K(236032K), 0.0097208 secs] [Times: user=0.03 sys=0.03, real=0.02 secs]
[Full GC (Ergonomics) [PSYoungGen: 5118K->0K(138240K)] [ParOldGen: 97530K->92659K(178176K)] 102649K->92659K(316416K),
```

比如对于第一行，进入老年代的大小为： $32879K - 5094K - (32879K - 10641K) = 5547$ ；前 5 行的计算结果正好等于第 6 行在 Full GC 时打印的老年区的内存使用量。

在刚开始运行的时候，没有使用到 OLD 区，所以此时堆的大小就是 OLD 去的大小，见上表第一行。

- 每次 Full GC 可以将年轻代内存清空，但不是每次都会：

```
[Full GC (Ergonomics) [PSYoungGen: 5118K->0K(138240K)] [ParOldGen: 97530K->92659K(178176K)] 102649K->92659K(316416K)
[Full GC (Ergonomics) [PSYoungGen: 59903K->6087K(304128K)] [ParOldGen: 148746K->177833K(291840K)] 208650K->183920K(595968K)
[Full GC (Ergonomics) [PSYoungGen: 109052K->0K(505856K)] [ParOldGen: 236541K->274888K(412160K)] 345593K->274888K(918016K)
[Full GC (Ergonomics) [PSYoungGen: 116483K->0K(482304K)] [ParOldGen: 352421K->321757K(491520K)] 468905K->321757K(973824K)
[Full GC (Ergonomics) [PSYoungGen: 94918K->0K(512512K)] [ParOldGen: 405734K->341095K(540672K)] 500653K->341095K(1053184K)
```

- 一般来说，Full GC 的速度都比 Young GC 慢（前者比后者大嘛）；
- 随着 GC 的进行，两个内存区的使用量和容量都在增大。
- Young GC 的过程中，不会改变老年代的容量

```
[Full GC (Ergonomics) [PSYoungGen: 94918K->0K(512512K)] [ParOldGen: 405734K->341095K(540672K)] 500653K->341095K(1053184K)
```

```
[GC (Allocation Failure) [PSYoungGen: 333824K->92059K(502784K)] 674919K->433154K(1043456K), 0.0123865 secs]
```

```
[GC (Allocation Failure) [PSYoungGen: 428443K->103175K(508928K)] 769538K->515942K(1049600K), 0.0229459 secs]
```

```
执行结束!共生成对象次数:13487
```

Heap

```
PSYoungGen      total 508928K, used 197782K [0x00000000d5d00000, 0x0000000100000000, 0x0000000100000000)
```

```
  eden space 336384K, 28% used [0x00000000d5d00000,0x00000000db963950,0x00000000ea580000)
```

```
  from space 172544K, 59% used [0x00000000ea580000,0x00000000f0a41f90,0x00000000f4e00000)
```

```
to   space 173056K, 0% used [0x00000000f5700000,0x00000000f5700000,0x0000000100000000)
```

```
ParOldGen      total 540672K, used 412766K [0x0000000081600000, 0x00000000a2600000, 0x00000000d5d00000)
```

```
  object space 540672K, 76% used [0x0000000081600000,0x000000009a917a70,0x00000000a2600000)
```

```
Metaspace      used 2643K, capacity 4486K, committed 4864K, reserved 1056768K
```

```
  class space   used 291K, capacity 386K, committed 512K, reserved 1048576K
```

上表中，最后一次 Full GC 时老年代的大小和经过两次 Young GC 后程序结束运行时的老年代大小一样；而年轻代的大小已经增大了。打印出的 Young 区的大小是 eden + from 的大小。

- Meta 区的容量和使用量没有变化。

1.2、模拟 OOM: java -Xmx128m -XX:+PrintGCDetails GCLogAnalysis

此时内存配置过小，很容易内存溢出：

```
[GC (Allocation Failure) [PSYoungGen: 32608K->5089K(38400K)] 32608K->13703K(125952K), 0.0044379 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
```

```
[GC (Allocation Failure) [PSYoungGen: 38251K->5112K(38400K)] 46865K->27380K(125952K), 0.0051897 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 38150K->5108K(38400K)] 60418K->41056K(125952K), 0.0057233 secs] [Times: user=0.06 sys=0.00, real=0.02 secs]
[GC (Allocation Failure) [PSYoungGen: 38286K->5115K(38400K)] 74235K->54274K(125952K), 0.0063850 secs] [Times: user=0.00 sys=0.00, real=0.02 secs]
[GC (Allocation Failure) [PSYoungGen: 38395K->5116K(38400K)] 87554K->65459K(125952K), 0.0043352 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 38330K->5113K(19968K)] 98673K->77000K(107520K), 0.0049580 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [PSYoungGen: 19961K->9608K(29184K)] 91848K->83455K(116736K), 0.0029665 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[Full GC (Ergonomics) [PSYoungGen: 9608K->0K(29184K)] [ParOldGen: 73847K->74420K(87552K)] 83455K->74420K(116736K),
..... #全是 Full GC
[Full GC (Ergonomics) [PSYoungGen: 14742K->14579K(29184K)] [ParOldGen: 87400K->87400K(87552K)] 102142K->101980K(116736K),
[Full GC (Ergonomics) [PSYoungGen: 14757K->14757K(29184K)] [ParOldGen: 87400K->87112K(87552K)] 102158K->101870K(116736K)
[Full GC (Ergonomics) [PSYoungGen: 14757K->14757K(29184K)] [ParOldGen: 87249K->87112K(87552K)] 102007K->101870K(116736K)
[Full GC (Allocation Failure) [PSYoungGen: 14757K->14757K(29184K)] [ParOldGen: 87112K->87093K(87552K)] 101870K->101850K(116736K),
```

上表中，最后几次 Full GC 已经没有内存可以回收，所以最后一次内存分配失败，导致 OOM。

1.3、分配不同大小的内存，运行程序

命令：`java -Xmx512m -XX:+PrintGCDetails GCLogAnalysis`

运行结果汇总

内存大小	总 GC 次数	Full GC 次数	分配的对象数	FULLGC 时长 MS
512m	44	10	8888	30-60
1024m	30	7	11278	20-60
2048m	22	6	14000	20-50
4096m	19	5	14397	20-60

1.4、GC 内容

- 对于 Minor GC，只清理 YOUNG 区，将 Eden 区和 S0 区的存活对象拷贝到 S1 区，并将部分对象晋升到老年区，参见 1.1 节第 2 条描述；
- 对于 Major GC，对新生代进行清理（在内存充足的情况下有可能将其清空），也对老年区进行整理操作。比如：Full GC (Ergonomics) [PSYoungGen: 21940K->0K(116736K)] [ParOldGen: 297023K->274367K(349696K)] 318963K->274367K(466432K)；或者 [PSYoungGen: 14724K->288K(29184K)] [ParOldGen: 84081K->87408K(87552K)] 98805K->87696K(116736K)。

Parallel: 年轻代GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前	<div></div>	<div></div>		<div>79%</div>
GC后			<div></div>	<div>93%</div>

Parallel: Full GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前	<div></div>	<div></div>		<div>93%</div>
GC后				<div>69%</div>

2、串行 GC

命令: `java -XX:+UseSerialGC -Xms512m -Xmx512m -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis`

输出如下:

```
java -XX:+UseSerialGC -Xms512m -Xmx512m -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis
正在执行...
2020-10-27T22:21:36.840-0800: [GC (Allocation Failure) 2020-10-27T22:21:36.840-0800: [DefNew: 139776K->17472K(157248K), 0.0946624 secs]
139776K->46467K(506816K), 0.0947164 secs] [Times: user=0.02 sys=0.02, real=0.09 secs]
2020-10-27T22:21:36.969-0800: [GC (Allocation Failure) 2020-10-27T22:21:36.969-0800: [DefNew: 157248K->17471K(157248K), 0.0995465 secs]
186243K->88831K(506816K), 0.0995927 secs] [Times: user=0.03 sys=0.02, real=0.10 secs]
.....
2020-10-27T22:21:37.398-0800: [GC (Allocation Failure) 2020-10-27T22:21:37.398-0800: [DefNew: 157247K->157247K(157248K), 0.0000186
secs]2020-10-27T22:21:37.398-0800: [Tenured: 299034K->268966K(349568K), 0.0603783 secs] 456281K->268966K(506816K), [Metaspace:
2707K->2707K(1056768K)], 0.0604480 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]
2020-10-27T22:21:37.477-0800: [GC (Allocation Failure) 2020-10-27T22:21:37.477-0800: [DefNew: 139776K->17471K(157248K), 0.0074137 secs]
408742K->314888K(506816K), 0.0074521 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
2020-10-27T22:21:37.510-0800: [GC (Allocation Failure) 2020-10-27T22:21:37.510-0800: [DefNew: 157185K->17471K(157248K), 0.0341385 secs]
454602K->362857K(506816K), 0.0341909 secs] [Times: user=0.02 sys=0.02, real=0.03 secs]
2020-10-27T22:21:37.569-0800: [GC (Allocation Failure) 2020-10-27T22:21:37.569-0800: [DefNew: 157247K->157247K(157248K), 0.0000182
secs]2020-10-27T22:21:37.569-0800: [Tenured: 345385K->312314K(349568K), 0.0518391 secs] 502633K->312314K(506816K), [Metaspace:
2707K->2707K(1056768K)], 0.0519094 secs] [Times: user=0.05 sys=0.00, real=0.06 secs]
2020-10-27T22:21:37.641-0800: [GC (Allocation Failure) 2020-10-27T22:21:37.641-0800: [DefNew: 139776K->139776K(157248K), 0.0000179
```

```
secs]2020-10-27T22:21:37.641-0800: [Tenured: 312314K->324754K(349568K), 0.0553174 secs] 452090K->324754K(506816K), [Metaspace: 2707K->2707K(1056768K)], 0.0553937 secs] [Times: user=0.05 sys=0.00, real=0.05 secs]
```

执行结束!共生成对象次数:6302

Heap

```
def new generation    total 157248K, used 5844K [0x00000007a0000000, 0x00000007aaaa0000, 0x00000007aaaa0000)
  eden space 139776K,    4% used [0x00000007a0000000, 0x00000007a05b5270, 0x00000007a8880000)
  from space 17472K,    0% used [0x00000007a9990000, 0x00000007a9990000, 0x00000007aaaa0000)
  to   space 17472K,    0% used [0x00000007a8880000, 0x00000007a8880000, 0x00000007a9990000)
tenured generation    total 349568K, used 324754K [0x00000007aaaa0000, 0x00000007c0000000, 0x00000007c0000000)
  the space 349568K,   92% used [0x00000007aaaa0000, 0x00000007be7c4818, 0x00000007be7c4a00, 0x00000007c0000000)
```

Metaspace used 2714K, capacity 4486K, committed 4864K, reserved 1056768K

class space used 297K, capacity 386K, committed 512K, reserved 1048576K

上面的输出中，“DefNew”表示对年轻代进行垃圾回收，“Tenured”表示对老年代进行垃圾回收。

2.1、运行结果汇总

堆内存越大，垃圾回收次数越少，两者成反比关系，下表不再记录。因为换了一台电脑，所以和 1.3 不具备可比性






内存大小	分配的对象数	FULLGC 时长 MS
512m	6302	没有发生 FULLGC，young GC 时长为 60 左右
1024m	8470	没有发生 FULLGC，young GC 时长为 60 左右
2048m	8471	没有发生 FULLGC，young GC 时长为 100 左右
4096m	8478	没有发生 FULLGC，young GC 时长为 140-430

2.2、GC 内容



参见下图。

- 对于 Minor GC, 将 Eden 区和 S0 区的存活对象拷贝到 S1 区, 并将部分对象晋升到老年区, 比如: [DefNew: 39093K->39093K(39296K), 0.0000213 secs]2020-10-27T22:44:31.785-0800: [Tenured: 73472K->83176K(87424K), 0.0187941 secs] 112566K->83176K(126720K);
- 对于 Major GC, 只对老年区进行操作。比如: [Full GC (Allocation Failure) 2020-10-27T22:44:31.849-0800: [Tenured: 87409K->87236K(87424K), 0.0177519 secs] 126703K->102655K(126720K)。

Minor GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				
GC后				

Full GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前	?	?		
GC后	?	?		

2.3、如果不配置 Xms 会发生什么？

第一次 GC 时间会提前。

3、CMS GC

命令：`java -XX:+UseConcMarkSweepGC -Xms512m -Xmx512m -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis;`

和之前的类似，也是先 YOUNG GC，然后 FULL GC。

```
2020-10-27T23:25:58.691-0800: [GC (Allocation Failure) 2020-10-27T23:25:58.691-0800: [ParNew: 139776K->17471K(157248K), 0.0180399 secs]
139776K->44591K(506816K), 0.0180965 secs] [Times: user=0.03 sys=0.04, real=0.01 secs]
2020-10-27T23:25:58.736-0800: [GC (Allocation Failure) 2020-10-27T23:25:58.736-0800: [ParNew: 157247K->17470K(157248K), 0.0255165 secs]
184367K->90336K(506816K), 0.0255971 secs] [Times: user=0.04 sys=0.05, real=0.03 secs]
2020-10-27T23:25:58.791-0800: [GC (Allocation Failure) 2020-10-27T23:25:58.791-0800: [ParNew: 157246K->17470K(157248K), 0.0303479 secs]
230112K->133286K(506816K), 0.0304150 secs] [Times: user=0.09 sys=0.02, real=0.03 secs]
2020-10-27T23:25:58.849-0800: [GC (Allocation Failure) 2020-10-27T23:25:58.849-0800: [ParNew: 157062K->17472K(157248K), 0.0320267 secs]
272877K->180195K(506816K), 0.0320724 secs] [Times: user=0.10 sys=0.02, real=0.04 secs]
2020-10-27T23:25:58.908-0800: [GC (Allocation Failure) 2020-10-27T23:25:58.908-0800: [ParNew: 157248K->17472K(157248K), 0.0416437 secs]
319971K->226650K(506816K), 0.0417022 secs] [Times: user=0.13 sys=0.02, real=0.04 secs]
2020-10-27T23:25:58.949-0800: [GC (CMS Initial Mark) [1 CMS-initial-mark: 209178K(349568K)] 227017K(506816K), 0.0003148 secs] [Times: user=0.00
sys=0.00, real=0.01 secs]
2020-10-27T23:25:58.950-0800: [CMS-concurrent-mark-start]
2020-10-27T23:25:58.959-0800: [CMS-concurrent-mark: 0.009/0.009 secs] [Times: user=0.01 sys=0.00, real=0.00 secs]
2020-10-27T23:25:58.959-0800: [CMS-concurrent-preclean-start]
2020-10-27T23:25:58.960-0800: [CMS-concurrent-preclean: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
2020-10-27T23:25:58.960-0800: [CMS-concurrent-abortable-preclean-start]
2020-10-27T23:25:58.977-0800: [GC (Allocation Failure) 2020-10-27T23:25:58.977-0800: [ParNew: 157248K->17469K(157248K), 0.0337076 secs]
366426K->268720K(506816K), 0.0337515 secs] [Times: user=0.10 sys=0.02, real=0.04 secs]
```

如上，对 young GC，使用了 ParNew 算法，对老年代使用 CMS 算法。常见组合如下：

(1)Serial+Serial Old 实现单线程的低延迟 垃圾回收机制;

(2)ParNew+CMS，实现多线程的低延迟垃圾回收机制;

(3)Parallel Scavenge 和 Parallel Scavenge Old，实现多线程的高吞吐量垃圾回收机制;

收集器	串行、并行 or并发	新生代/老年代	算法	目标	适用场景
Serial	串行	新生代	复制算法	响应速度 优先	单CPU环境下的Client模式
Serial Old	串行	老年代	标记-整理	响应速度 优先	单CPU环境下的Client模式、CMS 的后备预案
ParNew	并行	新生代	复制算法	响应速度 优先	多CPU环境时在Server模式下与 CMS配合
Parallel Scavenge	并行	新生代	复制算法	吞吐量优 先	在后台运算而不需要太多交互的任 务
Parallel Old	并行	老年代	标记-整理	吞吐量优 先	在后台运算而不需要太多交互的任 务
CMS	并发	老年代	标记-清除	响应速度 优先	集中在互连网站或B/S系统服务端上 的Java应用
G1	并发	both	标记-整理+复 制算法	响应速度 优先	面向服务端应用，将来替换CMS

3.1、GC 内容


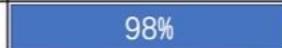


和串行 GC 很像。

- 对于 Minor GC，将 Eden 区和 S0 区的存活对象拷贝到 S1 区，并将部分对象晋升到老年区。
- 对于 Major GC，只对老年区进行操作。和串行 GC 的区别是此时不对内存进行整理。

CMS: 年轻代GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				 87%
GC后				 98%

CMS: Full GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				 98%
GC后				 83%

4、G1 GC

命令: `java -XX:+UseG1GC -Xms512m -Xmx512m -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis` 或者 `java -XX:+UseG1GC -Xms512m`

-Xmx512m -XX:+PrintGC -XX:+PrintGCDateStamps GCLogAnalysis。

GC 内容



和串行 GC 很像。

- 对于 Minor GC，将 Eden 区和 S0 区的存活对象拷贝到 S1 区，并将部分对象晋升到老年区。
- 对于 Major GC，只对老年区进行操作。

G1: 纯年轻代GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				
GC后				

G1: Full GC

内存池	Eden区	存活区S0	存活区S1	老年代
GC前				 78%
GC后				 77%

5、串行、并行、CMS、G1 对比

名称	内存大小	创建对象数量	GC 情况
串行	2g	8300 左右	没有 FULL GC
并行	2g	8000-10000	没有 FULL GC
CMS	2g	8000+	没有 FULL GC
G1	2g	8000 左右	没有 FULL GC
串行	256M	4000 左右	多次 FULL GC
并行	256M	3000 左右	多次 FULL GC
CMS	256M	4000 左右	多次 FULL GC
G1	256M	OOM	OOM
G1	512M	7000 左右	多次 FULL GC