1. 一般old GC触发的时候young一般也触发，所以不区分Major 与OldGC，由于Old比较大。
2. 由于串行GC会停掉应用线程，专心gc，所以应用会停摆，但是gc是无效的gc，相当于还是应用崩了，所以不要用串行gc。

# 1.JVM常见启动参数

1. ‐XX:+PrintGCDetails ，打印GC日志详情

2. ‐Xloggc:gc.demo.log gc.demo.log 文件可能在项目的根目录下。

当然， 我们也可以指定GC日志文件存放的绝对路径，比如 ‐Xloggc:/var/log/gc.demo.log 等形式。

1. ‐XX:+PrintGCDateStamps GC日志每一行前面，都打 印了GC发生时的具体时间。如 2019‐12‐15T15:09:59.235‐0800 ，表示的是： 东 8区时间2019年12月15日15:09:59秒.235毫秒 。
2. 指定内存大小

‐Xms512m ‐Xmx512m 表示初始化内存大小=最大堆内存大小

1. 指定垃圾收集器

JDK8时我们可以使用以下几种垃圾收集器

‐XX:+UseSerialGC

‐XX:+UseParallelGC

‐XX:+UseParallelGC ‐XX:+UseParallelOldGC

‐XX:+UseConcMarkSweepGC

‐XX:+UseConcMarkSweepGC ‐XX:+UseParNewGC

‐XX:+UseG1GC

1)使用串行垃圾收集器： ‐XX:+UseSerialGC

2)使用并行垃圾收集器： ‐XX:+UseParallelGC 和 ‐XX:+UseParallelGC ‐XX:+UseParallelOldGC 是等价的, 可以通过GC日志文件中的flags看出

1. 使用CMS垃圾收集器： ‐XX:+UseConcMarkSweepGC 和 ‐XX:+UseParNewGC ‐XX:+UseConcMarkSweepGC 是等价的()。 但如果只指定 ‐XX:+UseParNewGC 参数则老年代GC会使用SerialGC。

使用CMS时，命令行参数中会自动计算出年轻代、老年代的初始值和最大值，以及最大晋升阈值等信息（例如 ‐XX:MaxNewSize=178958336 ‐XX:NewSize=178958336 ‐XX:OldSize=357912576 ）。

1. 使用 G1垃圾收集器： ‐XX:+UseG1GC 。原则上不能指定G1垃圾收集器的年轻代大小，否则不仅是画蛇添足，更是自废武功了。因为G1的回收方式是小批量划定区块（region）进行，可能一次普通GC中既有年轻代又有老年代，可能某个区块一会是老年代，一会又变成年轻代了.

6 ‐XX:+PrintGCApplicationStoppedTime 可以输出每次GC的持续时间和程序暂停时间；

# 2.GC事件类型

## Minor GC

Minor GC 清理的是年轻代，又或者说 Minor GC 就是 年轻代 GC （ Young GC ，简称YGC）。

Minor GC 事件不处理老年代， 所以会把所有从老年代指向年轻代的引用都当做 GC Root 。从年轻代指向老年代的引用则在标记阶段被忽略。

STW，时间较短。

## Major GC 和 Full GC

Major GC处理老年代，Full GC处理整个堆。但是一般不好区分。因为Major GC很多是否都是由MinorGC触发的，所以此时也是Full GC，所以一般将这两个视为相同。

2者都会STW，并且时间较长。

# 解读GC日志

## 垃圾收集器判定：

串行GC中

1)DefNew – 表示垃圾收集器的名称。这个名字表示：年轻代使用的单线程、

标记­复制、STW 垃圾收集器。

1. Tenured 垃圾收集器的名称。老年代使用，表明使用的是单线程的STW垃圾收集器，使用的算法为 标记‐清除‐整理(mark‐sweep‐ compact )

并行GC中

1)PSYoungGen – 垃圾收集器的名称。这个名字表示的是在年轻代中使用的：

并行的 标记‐复制(mark‐copy) ，全线暂停(STW) 垃圾收集器。

2)ParOldGen – 用于清理老年代空间的垃圾收集器类型。在这里使用的是名为 ParOldGen 的垃圾收集器，这是一款并行 STW垃圾收集器，算法为 标记‐清除‐整理(mark‐sweep‐compact) 。

注意：虽然串行GC和并行GC在GC年轻代和老年代的名称不一样，但是算法都是一样的，只是串行和并行的却别。

## Minor Full GC的判定：

GC – 用来区分 Minor GC 还是 Full GC 的标志。 GC 表明这是一次**小型**

**GC**(Minor GC)，即年轻代GC。

Full GC – 完全GC的标志。 Full GC 表明本次GC清理年轻代和老年代，

Ergonomics 是触发GC的原因，表示JVM内部环境认为此时可以进行一次垃圾收集

## 时间消耗计算：

GC事件的持续时间，通过三个部分来衡量：

user 部分表示所有 GC线程消耗的CPU时间；

sys 部分表示系统调用和系统等待事件消耗的时间。

real 则表示应用程序暂停的时间。

1)串行GC中：

[Times: user=0.02 sys=0.01，real=0.03 secs]

因为串行垃圾收集器(Serial Garbage Collector)只使用单个线程，

所以这里 real = user + system,并且由于都是串行执行，所以年轻代和老年代的应用暂停时间计算方法一致。

2)并行GC中

年轻代：

[Times: user=0.04 sys=0.08，real=0.02 secs]

因为并不是所有的操作过程都能全部并行，所以在 Parallel GC 中， real

约等于 user + system /GC线程数 。

老年代：

[Full GC (Ergonomics)

3 [PSYoungGen: 14341K‐>0K(116736K)]

4 [ParOldGen: 327214K‐>242340K(349696K)]

5 341556K‐>242340K(466432K)，

6 [Metaspace: 3322K‐>3322K(1056768K)]，

7 0.0656553 secs]

8 [Times: user=0.30 sys=0.02，real=0.07 secs]

0.065 和 0.07的关系，0.07!=0.3 + 0.2/8

## 内存使用情况：

只有MinorGC时：

[PSYoungGen: 104179K‐>14341K(116736K)]

383933K‐>341556K(466432K)，0.0229343 secs]

104179K‐ >14341K(116736K) 表示GC前后的年轻代使用量，以及年轻代的总大小，简单计算GC后的年轻代使用率 14341K / 116736K = 12%

在GC之前，堆内存总使用量为 383933K ，其中年轻代为 104179K ，那么可以算出老年代使用量为 279754K 。

在此次GC完成后，年轻代使用量减少了 104179K ‐ 14341K = 89838K ，总的堆内存使用量减少了 383933K‐341556K = 42377K 。

那么我们可以计算出有 89838K ‐ 42377K = 47461K 的对象从年轻代提升到老年代。老年代的使用量为： 341556K ‐ 14341K = 327215K 。

老年代的大小为： 466432K ‐ 116736K = 349696K ，使用率为： 327215K/349696K=93% ，基本上快满了。

总结：年轻代GC，我们可以关注暂停时间，以及GC后的内存使用率是否正常， 但不用特别关注GC前的使用量，而且只要业务在运行，年轻代的对象分配就少不了，回收量也就不会少。

Full GC时：

[Full GC (Ergonomics)

[PSYoungGen: 14341K‐>0K(116736K)]

[ParOldGen: 327214K‐>242340K(349696K)]

341556K‐>242340K(466432K)，

[Metaspace: 3322K‐>3322K(1056768K)]，

0.0656553 secs]

327214K‐>242340K(349696K)] – 在GC 前后**老年代**内存的使用情况以及老年代空间大小。简单计算一下，GC之前，老年代使用率为 327214K/349696K = 93% ，GC后老年代使用率 242340K /349696K = 69% ，

341556K‐>242340K(466432K) – 在垃圾收集之前和之后**堆内存**的使用情

况，以及可用堆内存的总容量。简单分析可知，GC之前堆内存使用率为:

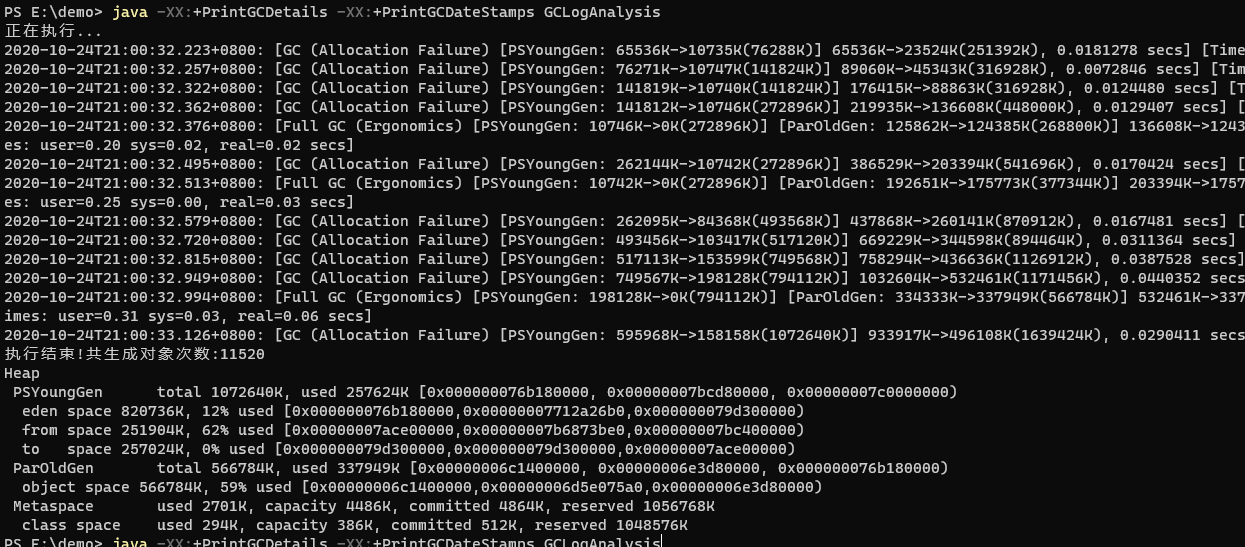
341556K/ 466432K = 73% ，GC之后堆内存的使用率为： 242340K /

466432K = 52% 。

总结: Full GC时我们更关注老年代的使用量有没有下降，以及下降了多少。如果FullGC之后内存不怎么下降，使用率还很高，那就说明系统有问题了。

jdk8默认执行情况如下：

java -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis



通过PSYoungGen 及 ParOldGen得知，默认使用的是并行GC。

并且堆内存开始分配的内存较小250m，在使用过程中自动扩容到1640m。

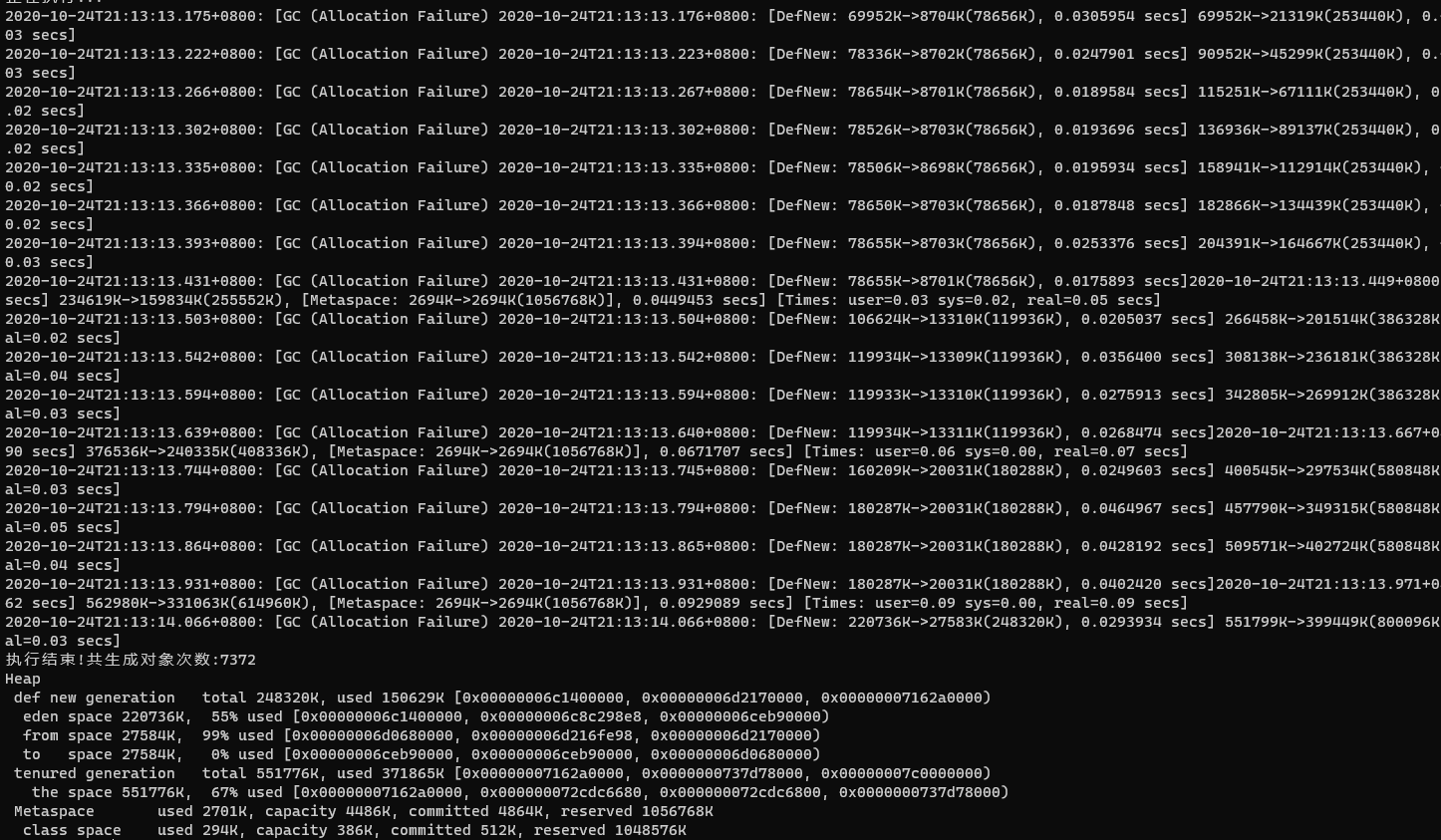
# 串行GC案例

## 没有指定Xmx Xms

### 指令

java -XX:+UseSerialGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

|  |  |  |  |
| --- | --- | --- | --- |
| Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 17 | 3 | 20~40 | 20~50 |

第一条GC日志如下：

2020-10-24T21:13:13.175+0800: [GC (Allocation Failure) 2020-10-24T21:13:13.176+0800: [DefNew: 69952K->8704K(78656K), 0.0305954 secs] 69952K->21319K(253440K), 0.0311253 secs] [Times: user=0.03 sys=0.00, real=0.03 secs]

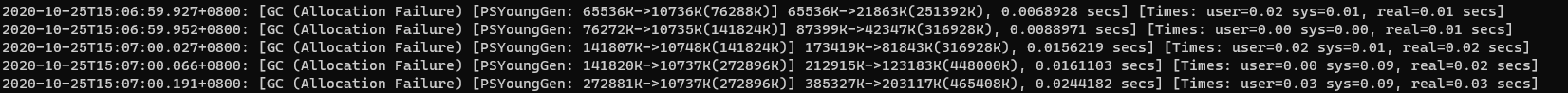
耗时30ms，时间较短，可以接受。

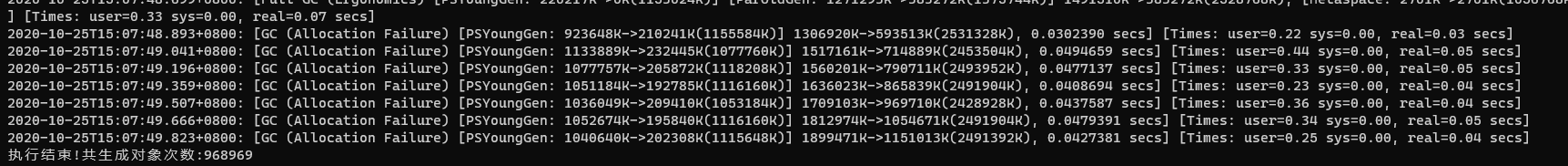
GC表示只有发生在年轻代，69952K->8704K(78656K)分别表示Young区GC前后内存(Young区当前可用内存)，69952K->21319K(253440K)分别表示整个堆GC前后内存(堆总共可用内存)

问题：Young及堆可用内存为什么会增长？且初始值不是2:1的关系???

回答：由于没有指定Xmx，JVM会根据heap使用情况逐渐多分配heap可使用内存，直到达到上限Xmx，这是一个动态调整的过程，并且Young:Old也不会满足2:1的比例。

但是，如果指定了Xmx，那么可用Heap始终都是固定值。如下，可用内存从251m一路增长到2g：



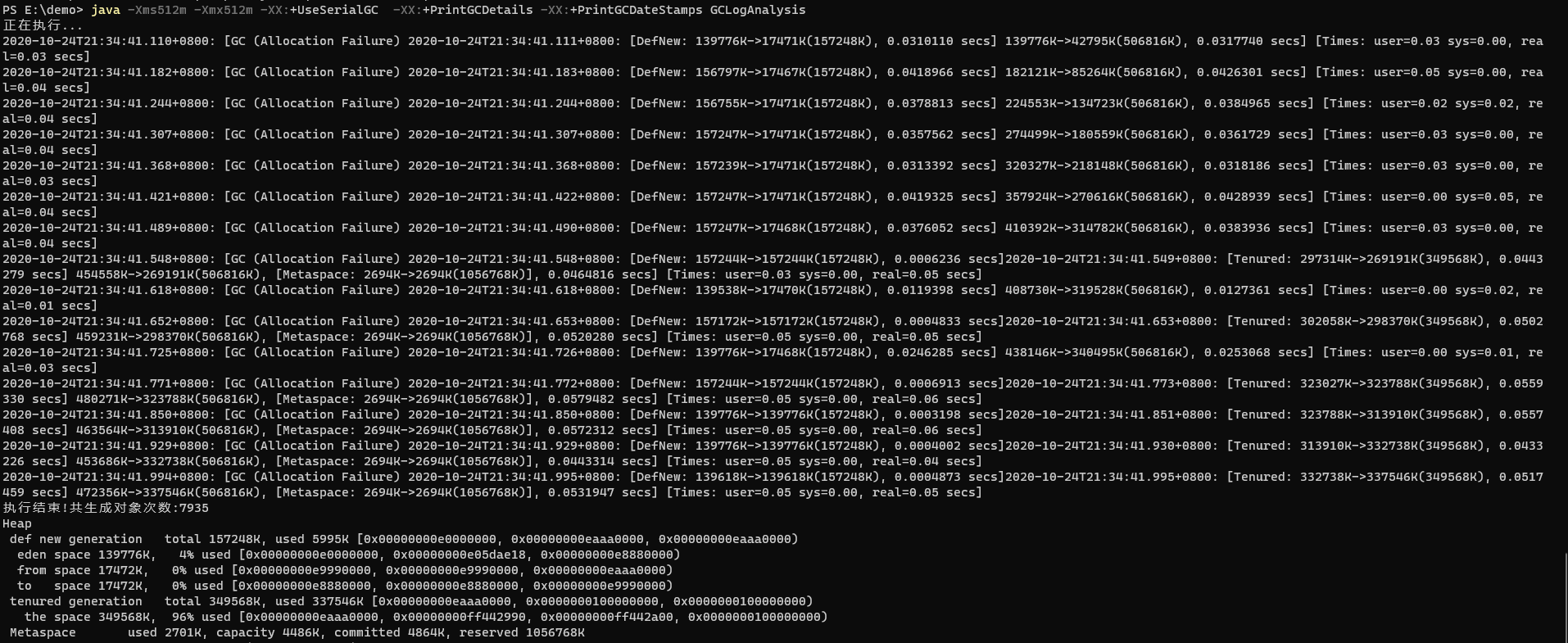


## 指定Xmx512m Xms512m

### 1)指令

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

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

1.Young和heap的可用内存大小没有发生变化，分别是Young=157m,Heap=506m，推出Old=506-157=349m，基本满足2:1的关系。Eden=140m，from=17m，to=17m，基本满足8:1:1关系。

2.Young和Old区前后的内存大小

可以得出前7次YoungGC每次都清理130m左右，但是整个heap的内存却在增加，说明很多对象从Young进入了Old。

后8次Young区大小达到了157m(100%)，GC清理很少几乎为0，Old区大小达到了337m(95%)，OldGC清理的内存也几乎为0。

总结，后期整个heap的内存被消耗完毕，故FullGC频繁，并由于是串行GC，故一直在STW，应用瘫痪，说明heap内存分配过小。

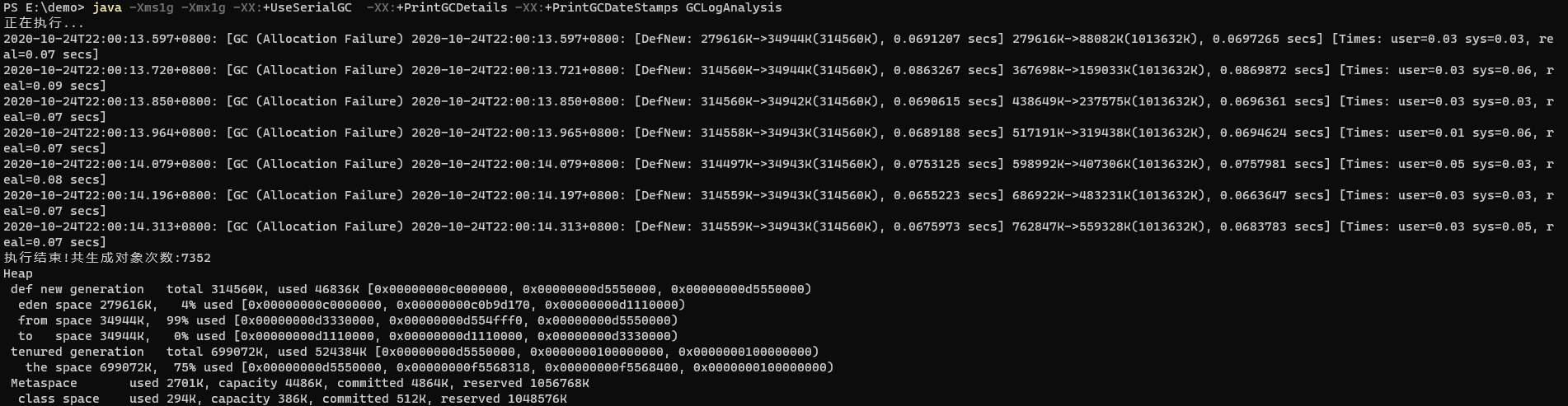
|  |  |  |  |
| --- | --- | --- | --- |
| Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 15 | 6 | 0.4~40 | 45~55ms |

## 指定Xmx1g Xms1g

### 1)指令

java -Xms1g -Xmx1g -XX:+UseSerialGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

1.Young和heap的可用内存大小没有发生变化，分别是Young=315m,Heap=1011m，推出Old=1011-315=696m，基本满足2:1的关系。Eden=280m，from=35m，to=35m，基本满足8:1:1关系。

2.根据Young和Old区Old前后的内存大小

只发生了YoungGC，没有FullGC，并且最后YoungGC后的值为35m(刚好是servivor区的大小)。

Old区，最大为559-35=524m，使用率为50%，所以没有触发Old区的GC，也就没有触发FullGC。

|  |  |  |  |
| --- | --- | --- | --- |
| Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 7 | 0 | 70~90 | / |

总结，heap=1g，能够满足内存需求，只发生了MinorGC，但是GC时间较长达到了70ms，还有优化的空间，可以考虑使用并行GC。

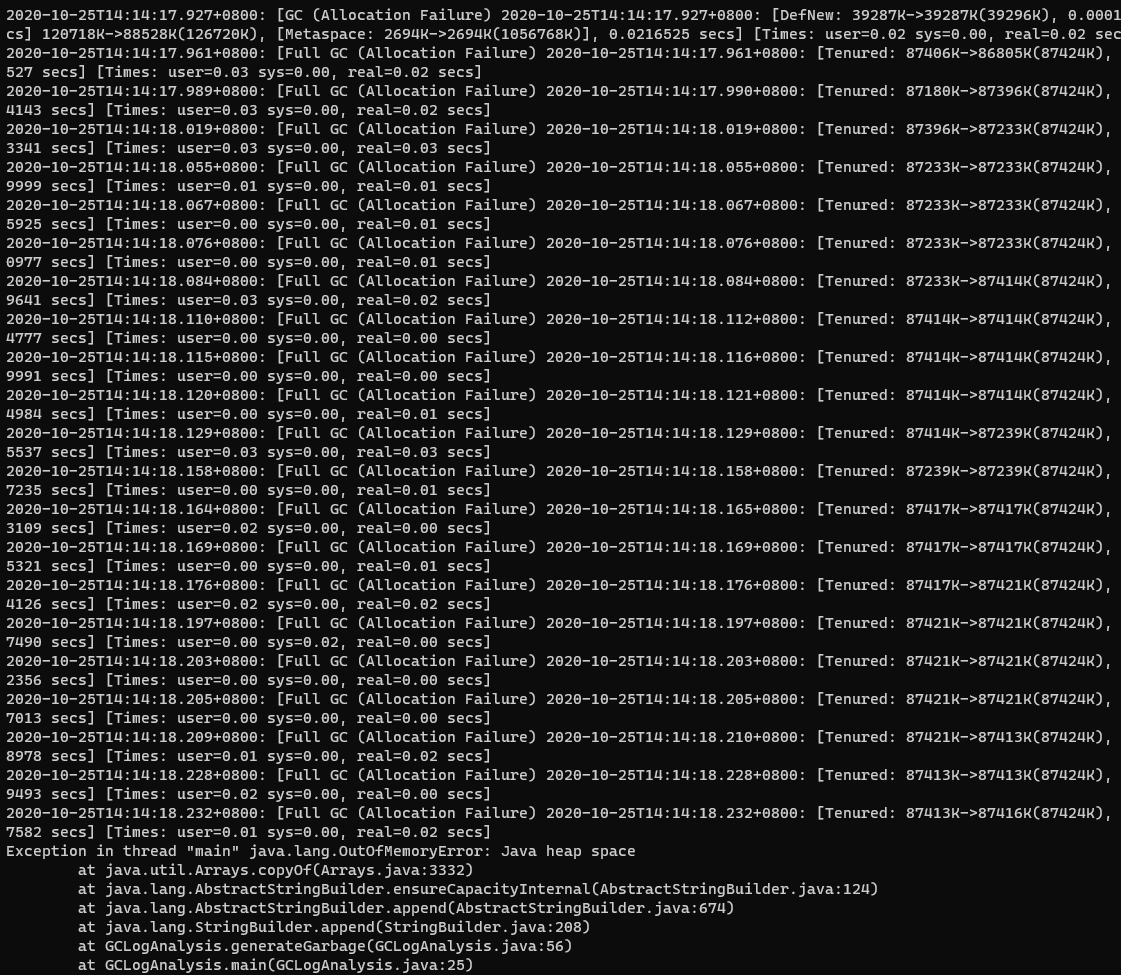
## 指定Xmx128m Xms128m

### 1)指令

java -Xms128m -Xmx128m -XX:+UseSerialGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情





### 3)OOM情况

在多次Full GC之后出现OOM情况

### 4)日志分析

1.Young和heap的可用内存大小没有发生变化，分别是Young=40m,Heap=126m，推出Old=126-40=86m，基本满足2:1的关系。Eden=35m，from=4m，to=4m，基本满足8:1:1关系。

2.根据Young和Old区Old前后的内存大小

前8次只有MinorGC，并且时间较短，这是因为Young内存较少且清理后占用空间为15%，但是整个Heap堆的内存一直在增加，说明Old区一直在增大。

后22次的fullGC，只触发了Old区的GC(此处可以得知Old GC也可以称为FullGC)，Old区使用率达到87m/87m=100%，导致Old GC触发频繁。但是GC清理对象几乎没有，最后导致了OOM。

|  |  |  |  |
| --- | --- | --- | --- |
| Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 8 | 22 | 9~17 | 3~30 |

总结，heap=126m，GC频繁，并且整个heap最后使用率也达到了100%，导致OOM；需要增加Heap堆大小。

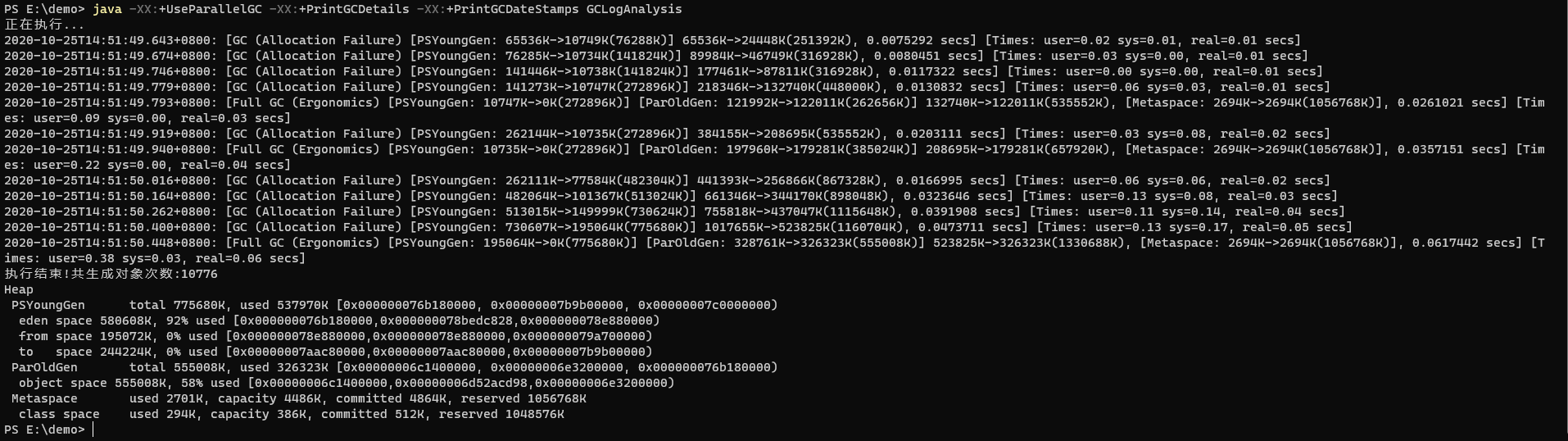
# 5.并行GC

## 1.没有指定Xmx Xms

### 1)指令

java -XX:+UseParallelGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

|  |  |  |  |
| --- | --- | --- | --- |
| Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 9 | 3 | 8~40 | 30~60 |

1.第一条GC日志如下：

2020-10-25T14:51:49.643+0800: [GC (Allocation Failure) [PSYoungGen: 65536K->10749K(76288K)] 65536K->24448K(251392K), 0.0075292 secs] [Times: user=0.02 sys=0.01, real=0.01 secs]

2.问题：Young及堆可用内存为什么会增长？且初始值不是2:1的关系???

解答：这是JVM动态分配的过程，同串行。

## 2.指定Xmx512m Xms512m

### 1)指令

java -XX:+UseParallelGC -Xmx512m -Xms512m -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

GC时间与次数

|  |  |  |  |
| --- | --- | --- | --- |
| Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 29 | 6 | 10以下 | 45~60ms |

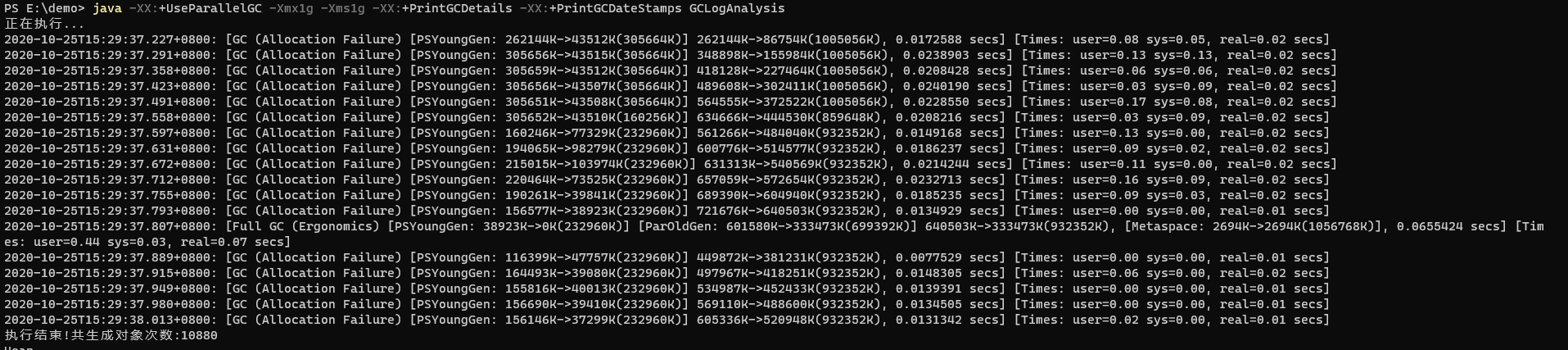
总结：与512的串行相比，由于YoungGC耗时明显较小(串行为20~40ms)，所以单位时间内可以GC的次数更多。并行GC的抖动更小，更推荐。

## 3.指定Xmx1g Xms1g

### 1)指令

java -XX:+UseParallelGC -Xmx1g -Xms1g -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

Young在50%以上会触发MinorGC，Old在95%以上会触发FullGC。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模式 | Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 并行 | 17 | 1 | 10~20 | 70 |
| 串行 | 7 | 0 | 70~90 | / |

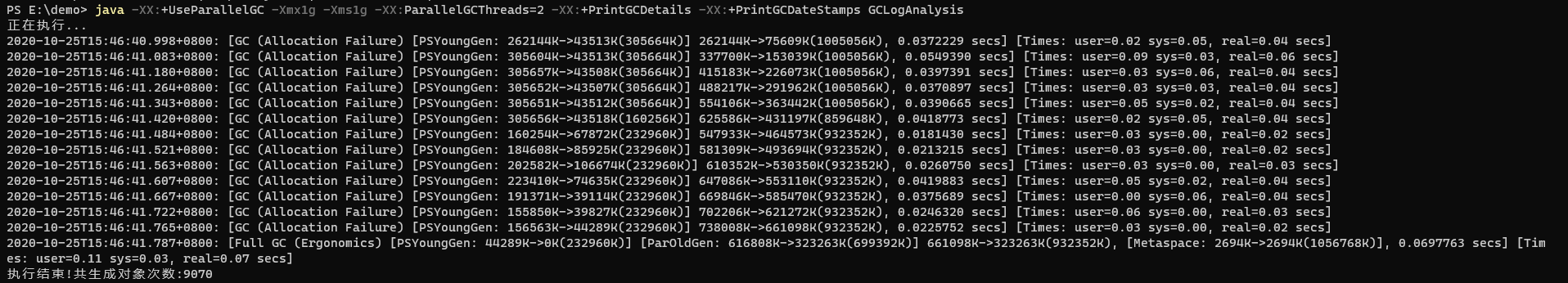
由上表格得知，串行虽然次数少，但是每次耗时远高于并行。对于应用来说，更加推荐用并行。

## 4.指定Xmx1g Xms1g -XX:ParallelGCThreads=2

### 1)指令

java -XX:+UseParallelGC -Xmx1g -Xms1g -XX:ParallelGCThreads=2-XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

Young在50%以上会触发MinorGC，Old在95%以上会触发FullGC。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模式 | Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 并行(默认8核心) | 17 | 1 | 10~20 | 70 |
| 并行(指定2核心) | 13 | 1 | 20~45 | 70 |
| 串行 | 7 | 0 | 70~90 | / |

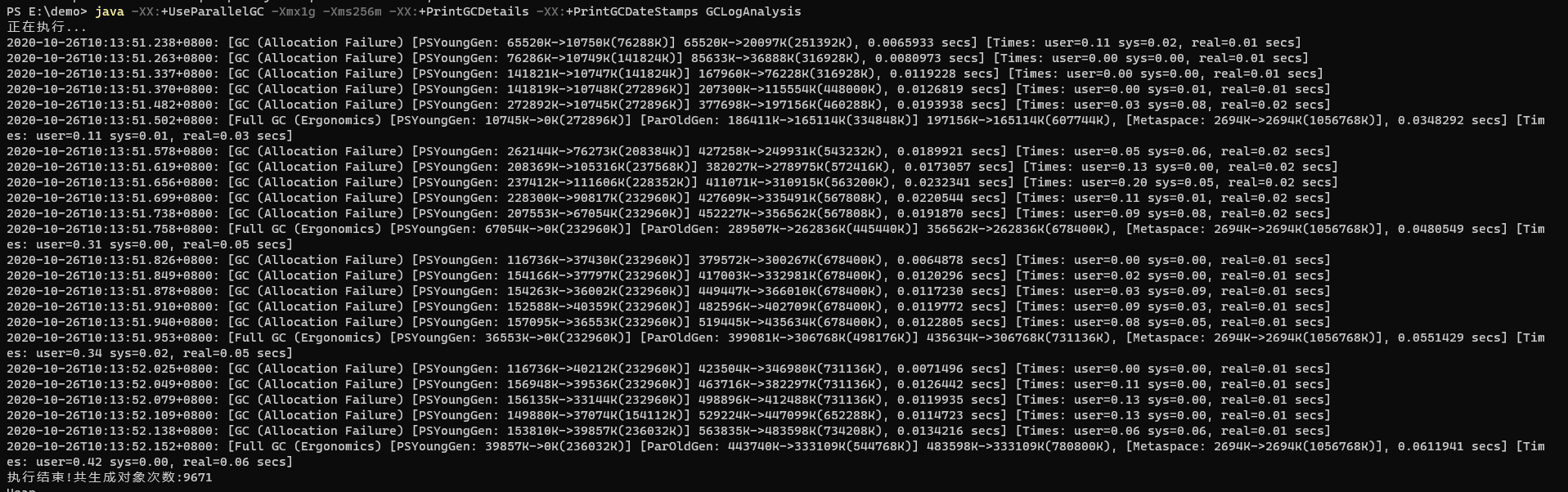
由上表格得知，并行GC指定的线程数越多(<=核心数)时，GC时间更短，GC更频繁，系统延迟越低。

## 指定Xmx1g Xms256m

### 1)指令

java -XX:+UseParallelGC -Xmx1g -Xms256m -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

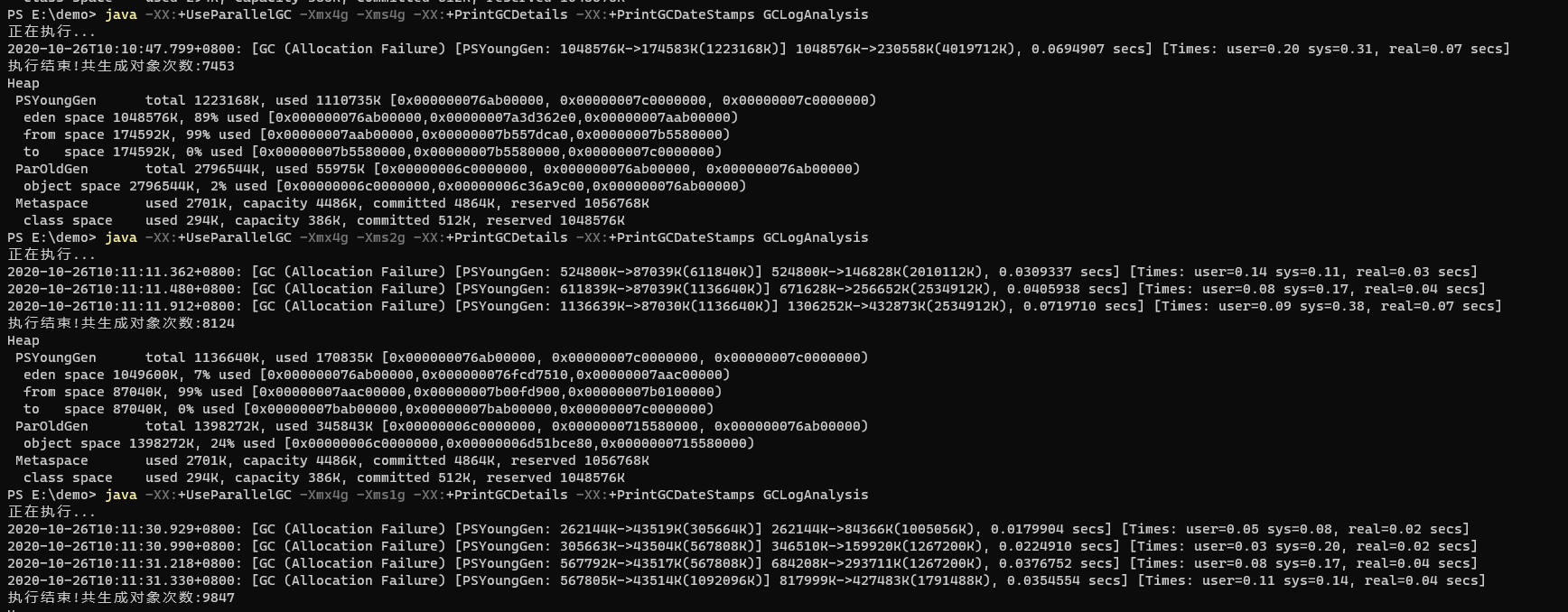
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 参数模式 | Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| Xmx=Xms=1g | 17 | 1 | 10~20 | 70 |
| Xms=256m | 20 | 4 | 10~20 | 60 |

1.可用堆内存从251m逐渐扩容到462m，与我们设置的堆内存大小匹配。

总结：当Xmx！=Xms时，导致JVM会动态扩容Heap可用内存。此时可能导致应用启动时出现OOM。

1. 而且Minor 和 Full GC的次数也明显增多。

3.如果Xmx与Xms的差值过小效果差；差值过大效果更明显，如下：

  
总结：推荐使用Xms=Xmx的方式。

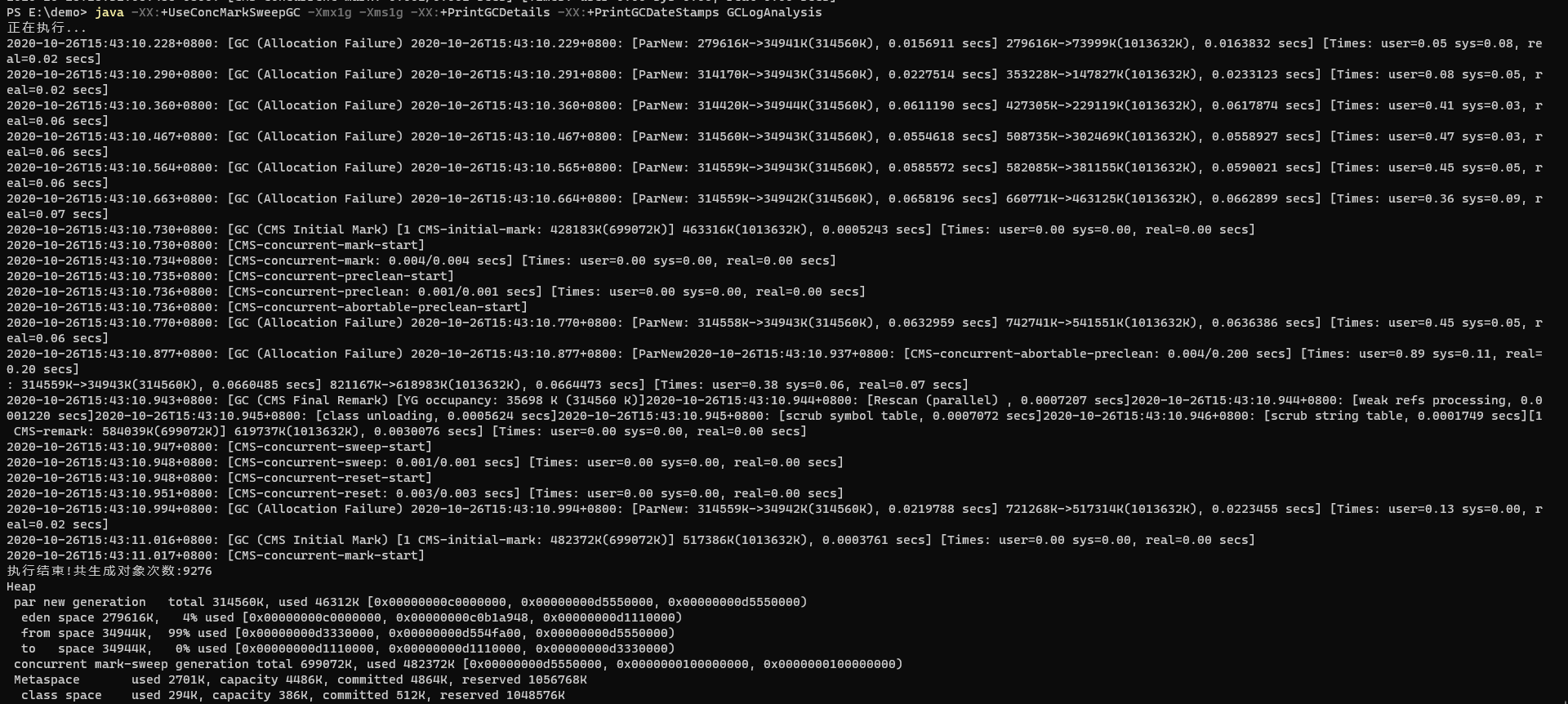
# 6.CMS GC

## 1.指定Xmx1g Xms1g

### 1)指令

java -XX:+UseConcMarkSweepGC -Xmx1g -Xms1g -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情



### 3)OOM情况

无

### 4)日志分析

GC时间与次数

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 模式 | Young GC 次数 | Full GC 次数 | YoungCG时间(ms) | Old GC 时间(ms) |
| 并行 | 17 | 1 | 10~20 | 70 |
| CMS | 7 | 2 | 10~65 |  |

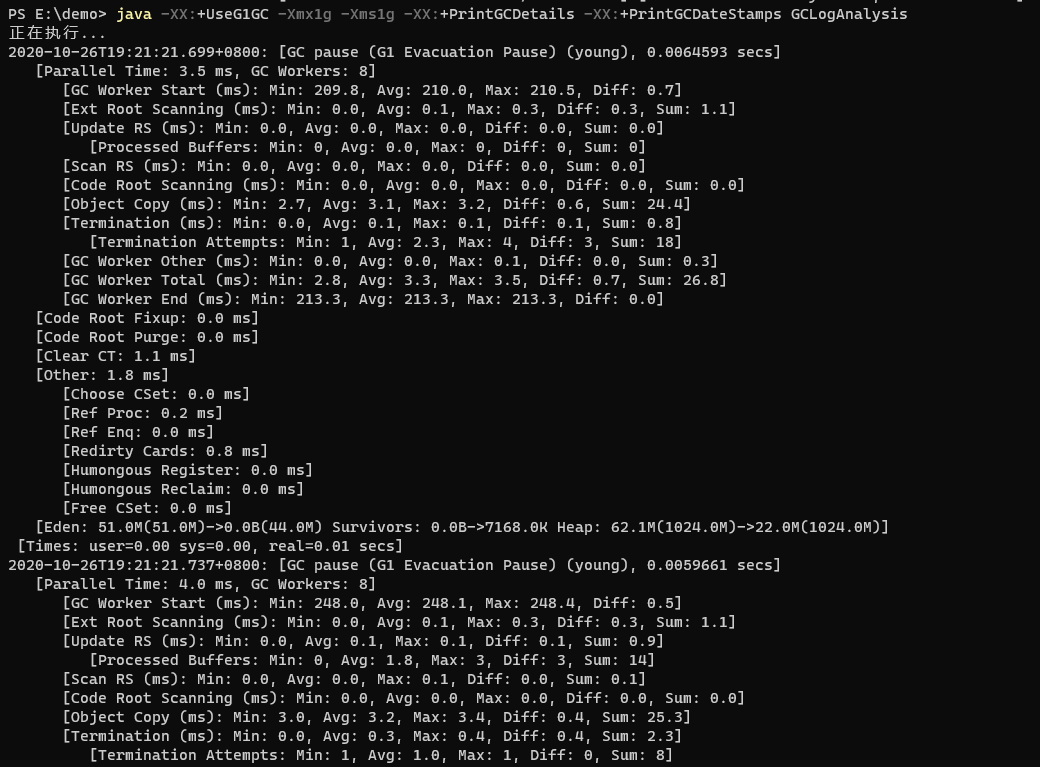
# 7.G1 GC

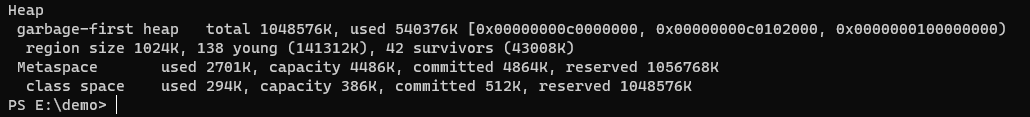
## 1.指定Xmx1g Xms1g

### 1)指令

java -XX:+UseG1GC -Xmx1g -Xms1g -XX:+PrintGCDetails -XX:+PrintGCDateStamps GCLogAnalysis

### 2)日志详情





### 3)OOM情况

无

### 4)日志分析