

# 实操分析GC日志总结

## 1. GCLogAnalysis 文件分析

- 使用如下命令，通过windows 使用powershell安装 superbenmarker

```
1. Set-ExecutionPolicy Bypass -Scope Process -Force;
[System.Net.ServicePointManager]::SecurityProtocol =
[System.Net.ServicePointManager]::SecurityProtocol -bor 3072; iex ((New-
Object
System.Net.WebClient).DownloadString('https://chocolatey.org/install.ps1'))

2.choco install superbenchmarker
```

- 使用如下命令编译现成的 GCLogAnalysis.java 文件

```
javac GCLogAnalysis.java
```

- 执行如下命令（此处命令需要指定编码格式为utf-8，否则会导致打印出来的日志乱码），打印GC日志：

```
java -XX:+PrintGCDetails -Dfile.encoding=utf-8 GCLogAnalysis
```

如需输出GC日志到文件中，则可以使用如下命令：

```
java -Xloggc:gc.demo.log -XX:+PrintGCDetails -XX:+PrintGCDateStamps -
Dfile.encoding=utf-8 GCLogAnalysis
```

## Parallel GC日志分析

我用的是JDK8环境，从输出的文件中可以看到，命令行启动参数使用的默认GC是**并行GC**

```
gc_demo.log
1 Java HotSpot(TM) 64-Bit Server VM (25.161-b12) for windows-amd64 JRE (1.8.0_161-b12), built on Dec 19 2017 17:52:25 by "java_re" with MS VC++ 10.0 (VS2010)
2 Memory: 4k page, physical 8295652k(1738976k free), swap 13328008k(2310896k free)
3 Commandline flags: -XX:InitialHeapSize=132730432 -XX:MaxHeapSize=2123686912 -XX:+PrintGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps
4 -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:-UseLargePagesIndividualAllocation -XX:+UseParallelGC 32944K->11059K(125952K), 0.0040531 secs] [Times: user=0.00
5 sys=0.00, real=0.00 secs]
6 2020-10-28T17:41:17.003+0800: 0.121: [GC (Allocation Failure) [PSYoungGen: 32944K->5109K(38400K)] 32944K->11059K(125952K), 0.0040531 secs] [Times: user=0.00
7 sys=0.00, real=0.01 secs]
8 2020-10-28T17:41:17.019+0800: 0.137: [GC (Allocation Failure) [PSYoungGen: 38384K->5104K(71680K)] 44334K->22306K(159232K), 0.0044230 secs] [Times: user=0.00
9 sys=0.00, real=0.00 secs]
10 2020-10-28T17:41:17.048+0800: 0.166: [GC (Allocation Failure) [PSYoungGen: 71664K->5109K(71680K)] 88866K->43949K(159232K), 0.0069393 secs] [Times: user=0.00
11 sys=0.00, real=0.01 secs]
12 2020-10-28T17:41:17.066+0800: 0.183: [GC (Allocation Failure) [PSYoungGen: 71567K->5109K(138240K)] 110407K->67885K(225792K), 0.0072928 secs] [Times: user=0.02
13 sys=0.11, real=0.01 secs]
14 2020-10-28T17:41:17.079+0800: 0.191: [Full GC (Ergonomics) [PSYoungGen: 5109K->0K(138240K)] [ParOldGen: 62776K->64629K(138240K)] 67885K->64629K(276480K),
15 [Metaspace: 2812K->2812K(1056768K)], 0.0176626 secs] [Times: user=0.13 sys=0.00, real=0.02 secs]
16 2020-10-28T17:41:17.137+0800: 0.255: [GC (Allocation Failure) [PSYoungGen: 133120K->5113K(138240K)] 197749K->111759K(276480K), 0.0162929 secs] [Times: user=0.03
17 sys=0.08, real=0.02 secs]
18 2020-10-28T17:41:17.154+0800: 0.272: [Full GC (Ergonomics) [PSYoungGen: 5113K->0K(138240K)] [ParOldGen: 106645K->105005K(212992K)] 111759K->105005K(351232K),
19 [Metaspace: 2812K->2812K(1056768K)], 0.0229034 secs] [Times: user=0.13 sys=0.00, real=0.02 secs]
20 2020-10-28T17:41:17.197+0800: 0.315: [GC (Allocation Failure) [PSYoungGen: 133094K->45484K(258560K)] 238100K->150490K(471552K), 0.0149265 secs] [Times: user=0.03
21 sys=0.03, real=0.01 secs]
22 2020-10-28T17:41:17.266+0800: 0.384: [GC (Allocation Failure) [PSYoungGen: 253868K->61439K(269824K)] 358874K->210434K(482816K), 0.0337277 secs] [Times: user=0.06
23 sys=0.17, real=0.03 secs]
24 2020-10-28T17:41:17.330+0800: 0.448: [GC (Allocation Failure) [PSYoungGen: 269823K->95742K(400896K)] 418818K->268664K(613888K), 0.0391715 secs] [Times: user=0.14
25 sys=0.20, real=0.04 secs]
26 2020-10-28T17:41:17.369+0800: 0.487: [Full GC (Ergonomics) [PSYoungGen: 95742K->6530K(400896K)] [ParOldGen: 172921K->212786K(338944K)] 268664K->219316K(739840K),
27 [Metaspace: 2812K->2812K(1056768K)], 0.0654477 secs] [Times: user=0.30 sys=0.06, real=0.07 secs]
28 2020-10-28T17:41:17.484+0800: 0.602: [GC (Allocation Failure) [PSYoungGen: 311682K->98051K(429568K)] 524468K->311003K(768512K), 0.0302416 secs] [Times: user=0.05
29 sys=0.14, real=0.03 secs]
30 2020-10-28T17:41:17.559+0800: 0.677: [GC (Allocation Failure) [PSYoungGen: 402660K->136176K(484352K)] 615612K->383872K(823296K), 0.0667257 secs] [Times: user=0.14
31 sys=0.27, real=0.07 secs]
32 2020-10-28T17:41:17.674+0800: 0.792: [GC (Allocation Failure) [PSYoungGen: 484336K->115122K(463360K)] 732032K->461485K(809984K), 0.0562751 secs] [Times: user=0.22
33 sys=0.19, real=0.06 secs]
34 2020-10-28T17:41:17.730+0800: 0.848: [Full GC (Ergonomics) [PSYoungGen: 115122K->0K(463360K)] [ParOldGen: 346363K->331191K(502272K)] 461485K->331191K(965632K),
35 [Metaspace: 2812K->2812K(1056768K)], 0.0875903 secs] [Times: user=0.50 sys=0.02, real=0.09 secs]
36 2020-10-28T17:41:17.883+0800: 1.001: [GC (Allocation Failure) [PSYoungGen: 347927K->99325K(512000K)] 679119K->430517K(1014272K), 0.0242623 secs] [Times: user=0.13
37 sys=0.00, real=0.02 secs]
38 Heap
39 PSYoungGen      total 512000K, used 422731K [0x00000000d5d00000, 0x0000000100000000, 0x0000000100000000)
40 eden space 332800K, 97% used [0x00000000d5d00000, 0x00000000e9838e0, 0x00000000ea200000)
41 from space 179200K, 55% used [0x00000000f5100000, 0x00000000fb1ff550, 0x0000000100000000)
42 to space 179200K, 0% used [0x00000000ea200000, 0x00000000ea200000, 0x00000000f5100000)
43 ParOldGen       total 502272K, used 331191K [0x0000000081600000, 0x00000000a0800000, 0x00000000d5d00000)
```

上图中可以显然的看出发生了12次minor GC 和 4次Full GC。其中只有在Full GC时才会回收老年代和metaspase进行回收清理。在minor GC时只是回收了年轻代的数据。在每次GC日志中，都可以看到GC消耗的时间信息，其中各个参数的含义为：

**user** 表示GC线程所消耗的总CPU时间， **sys** 表示 操作系统调用和系统等待事件所消耗的时间；  
**real** 则表示应用程序实际暂停的时间。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。

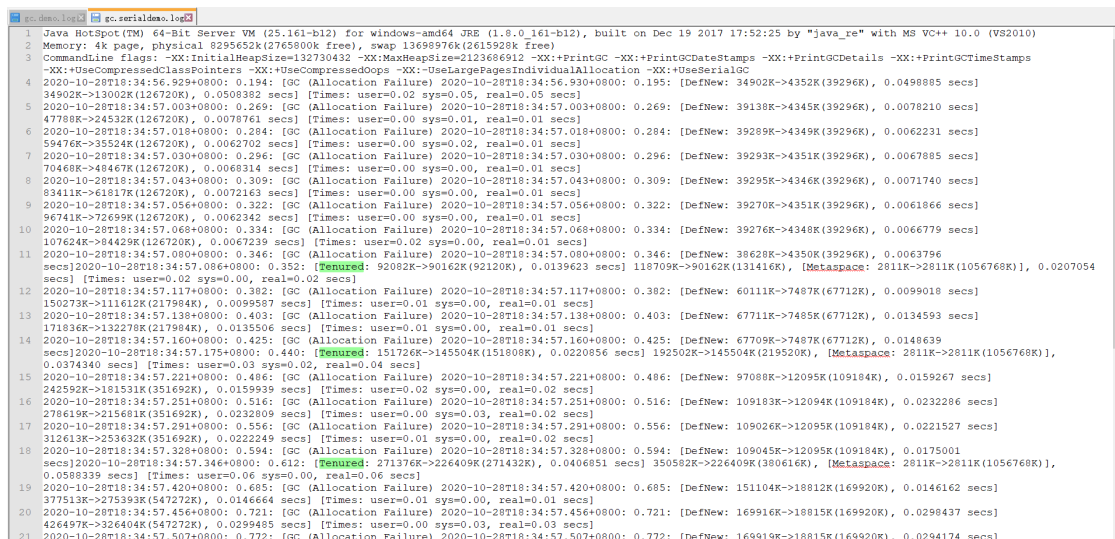
上图中可以看出，年轻代GC的名称为PSYoungGen，它是使用**标记-复制算法**的GC。而老年代GC的名称叫ParOldGen，它是使用**标记-清除-整理算法**的GC。二者不同之处在于，老年代的清除之后会整理内存。另外，无论是minor GC 还是Full GC，都会发生STW。

## SerialGC日志分析

- 使用如下命令，产生串行GC日志文件：

```
java -Xloggc:gc.serialdemo.log -XX:+UseSerialGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Dfile.encoding=utf-8 GCLogAnalysis
```

此时日志文件与上面的并行GC日志文件有很大的不同：



```
1 Java HotSpot(TM) 64-Bit Server VM (25.161-b12) for windows-amd64 JRE (1.8.0_161-b12), built on Dec 19 2017 17:52:25 by "java_re" with MS VC++ 10.0 (VS2010)
2 Memory: 4k page, physical 8295652K(2765900k free), swap 13698976K(2615928k free)
3 Commandline flags: -XX:InitialHeapSize=132730432 -XX:MaxHeapSize=2123666912 -XX:+PrintGC -XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+PrintGCTimeStamps
4 -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:UseLargePagesIndividualAllocation -XX:+UseSerialGC
5 2020-10-28T18:34:56.929+0800: 0.194: [GC (Allocation Failure) 2020-10-28T18:34:56.930+0800: 0.195: [DefNew: 34902K->4352K(39296K), 0.0498885 secs]
6 34902K->13002K(126720K), 0.0508982 secs] [Times: user=0.02 sys=0.05, real=0.05 secs]
7 2020-10-28T18:34:57.003+0800: 0.269: [GC (Allocation Failure) 2020-10-28T18:34:57.003+0800: 0.269: [DefNew: 39138K->4345K(39296K), 0.0078210 secs]
8 47788K->24532K(126720K), 0.0078761 secs] [Times: user=0.00 sys=0.01, real=0.01 secs]
9 2020-10-28T18:34:57.018+0800: 0.284: [GC (Allocation Failure) 2020-10-28T18:34:57.018+0800: 0.284: [DefNew: 39289K->4349K(39296K), 0.0062231 secs]
10 59476K->35524K(126720K), 0.0062702 secs] [Times: user=0.00 sys=0.02, real=0.01 secs]
11 2020-10-28T18:34:57.030+0800: 0.296: [GC (Allocation Failure) 2020-10-28T18:34:57.030+0800: 0.296: [DefNew: 39293K->4351K(39296K), 0.0067895 secs]
12 70468K->48467K(126720K), 0.0068314 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
13 2020-10-28T18:34:57.043+0800: 0.309: [GC (Allocation Failure) 2020-10-28T18:34:57.043+0800: 0.309: [DefNew: 39295K->4346K(39296K), 0.0071740 secs]
14 83411K->61817K(126720K), 0.0072163 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
15 2020-10-28T18:34:57.056+0800: 0.322: [GC (Allocation Failure) 2020-10-28T18:34:57.056+0800: 0.322: [DefNew: 39270K->4351K(39296K), 0.0061866 secs]
16 96741K->72699K(126720K), 0.0062342 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
17 2020-10-28T18:34:57.068+0800: 0.334: [GC (Allocation Failure) 2020-10-28T18:34:57.068+0800: 0.334: [DefNew: 39276K->4348K(39296K), 0.0066779 secs]
18 107624K->84429K(126720K), 0.0067239 secs] [Times: user=0.02 sys=0.00, real=0.01 secs]
19 2020-10-28T18:34:57.080+0800: 0.346: [GC (Allocation Failure) 2020-10-28T18:34:57.080+0800: 0.346: [DefNew: 38628K->4350K(39296K), 0.0063796 secs]
20 2020-10-28T18:34:57.086+0800: 0.352: [Tenured: 92082K->90162K(92120K), 0.0139623 secs] 118709K->90162K(131416K), [Metaspace: 2811K->2811K(1056768K)], 0.0207054 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
21 2020-10-28T18:34:57.117+0800: 0.382: [GC (Allocation Failure) 2020-10-28T18:34:57.117+0800: 0.382: [DefNew: 60111K->7487K(67712K), 0.0099018 secs]
22 150279K->111612K(121798K), 0.0099587 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
23 2020-10-28T18:34:57.138+0800: 0.403: [GC (Allocation Failure) 2020-10-28T18:34:57.138+0800: 0.403: [DefNew: 67711K->7485K(67712K), 0.0134593 secs]
24 171836K->132278K(121798K), 0.0135506 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
25 2020-10-28T18:34:57.160+0800: 0.425: [GC (Allocation Failure) 2020-10-28T18:34:57.160+0800: 0.425: [DefNew: 67709K->7487K(67712K), 0.0148639 secs]
26 2020-10-28T18:34:57.160+0800: 0.440: [Tenured: 151726K->145504K(151808K), 0.0220856 secs] 192502K->145504K(219520K), [Metaspace: 2811K->2811K(1056768K)], 0.0374340 secs] [Times: user=0.03 sys=0.02, real=0.04 secs]
27 2020-10-28T18:34:57.221+0800: 0.486: [GC (Allocation Failure) 2020-10-28T18:34:57.221+0800: 0.486: [DefNew: 97088K->12095K(109184K), 0.0159267 secs]
28 242592K->181531K(351692K), 0.0159939 secs] [Times: user=0.02 sys=0.00, real=0.02 secs]
29 2020-10-28T18:34:57.251+0800: 0.516: [GC (Allocation Failure) 2020-10-28T18:34:57.251+0800: 0.516: [DefNew: 109183K->12094K(109184K), 0.0232286 secs]
30 278619K->215681K(351692K), 0.0232809 secs] [Times: user=0.00 sys=0.03, real=0.02 secs]
31 2020-10-28T18:34:57.291+0800: 0.556: [GC (Allocation Failure) 2020-10-28T18:34:57.291+0800: 0.556: [DefNew: 109026K->12095K(109184K), 0.0221527 secs]
32 312613K->253632K(351692K), 0.0222249 secs] [Times: user=0.01 sys=0.00, real=0.02 secs]
33 2020-10-28T18:34:57.328+0800: 0.594: [GC (Allocation Failure) 2020-10-28T18:34:57.328+0800: 0.594: [DefNew: 109045K->12095K(109184K), 0.0175001 secs]
34 2020-10-28T18:34:57.346+0800: 0.612: [Tenured: 271376K->226409K(271432K), 0.0406851 secs] 350582K->226409K(380616K), [Metaspace: 2811K->2811K(1056768K)], 0.0588339 secs] [Times: user=0.06 sys=0.00, real=0.06 secs]
35 2020-10-28T18:34:57.420+0800: 0.685: [GC (Allocation Failure) 2020-10-28T18:34:57.420+0800: 0.685: [DefNew: 151104K->18812K(169920K), 0.0146162 secs]
36 377513K->275393K(547272K), 0.0146664 secs] [Times: user=0.01 sys=0.00, real=0.01 secs]
37 2020-10-28T18:34:57.456+0800: 0.721: [GC (Allocation Failure) 2020-10-28T18:34:57.456+0800: 0.721: [DefNew: 169916K->18815K(169920K), 0.0298437 secs]
38 426497K->326404K(547272K), 0.0299485 secs] [Times: user=0.00 sys=0.03, real=0.03 secs]
39 2020-10-28T18:34:57.507+0800: 0.772: [GC (Allocation Failure) 2020-10-28T18:34:57.507+0800: 0.772: [DefNew: 169919K->18815K(169920K), 0.0294174 secs]
```

串行GC的日志主要是两种，一种是类似并行GC时的minor GC，只清理了年轻代的内存。它的年轻代GC名称叫做DefNew，是一种使用**标记-复制算法**的单线程回收器。

另一种是类似并行GC的Full GC，它清理了整个堆的内存，它的老年代GC名称是Tenured，是一种**标记-清除-整理**的单线程回收器

由于是单线程回收器，因此它的耗时时间**real=user+sys**，同时不管是年轻代还是老年代在垃圾回收时，都会产生STW。

## CMSGC日志分析

- 使用如下命令生成CMSGC日志文件：

```
java -Xloggc:gc.cmsdemo.log -XX:+UseConcMarkSweepGC -XX:+PrintGCDetails -XX:+PrintGCDateStamps -Dfile.encoding=utf-8 GCLogAnalysis
```

其日志文件如下：

```
1 Java HotSpot(TM) 64-Bit Server VM (25.161-b12) for windows-amd64 JRE (1.8.0_161-b12), built on Dec 19 2017 17:52:25 by "java_re" with MS VC++ 10.0 (VS2010)
2 Memory: 4k page, physical 8295652k (2071244k free), swap 13689976k (2330068k free)
3 Commandline flags: -XX:InitialHeapSize=132730432 -XX:MaxHeapSize=2123666912 -XX:MaxNewSize=697933824 -XX:MaxTenuringThreshold=6 -XX:OldPLABSize=16 -XX:+PrintGC
-XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseConcMarkSweepGC
-XX:-UseLargePagesIndividualAllocation -XX:+UseParNewGC
4 2020-10-28T20:15:45.030+0800: 0.169: [GC (Allocation Failure) 2020-10-28T20:15:45.030+0800: 0.169: [ParNew: 34891K->4342K(39296K), 0.0046050 secs]
34891K->1333K(126720K), 0.0047960 secs] [Times: user=0.00 sys=0.00, real=0.01 secs]
5 2020-10-28T20:15:45.048+0800: 0.187: [GC (Allocation Failure) 2020-10-28T20:15:45.048+0800: 0.187: [ParNew: 39232K->4352K(39296K), 0.0084013 secs]
48225K->29600K(126720K), 0.0086633 secs] [Times: user=0.08 sys=0.03, real=0.01 secs]
6 2020-10-28T20:15:45.066+0800: 0.205: [GC (Allocation Failure) 2020-10-28T20:15:45.066+0800: 0.205: [ParNew: 39296K->4352K(39296K), 0.0081284 secs]
45444K->42738K(126720K), 0.0081984 secs] [Times: user=0.08 sys=0.03, real=0.01 secs]
7 2020-10-28T20:15:45.081+0800: 0.220: [GC (Allocation Failure) 2020-10-28T20:15:45.081+0800: 0.220: [ParNew: 39296K->4330K(39296K), 0.0081368 secs]
77679K->55097K(126720K), 0.0082010 secs] [Times: user=0.13 sys=0.00, real=0.01 secs]
8 2020-10-28T20:15:45.090+0800: 0.228: [GC (CMS Initial Mark) [1 CMS-initial-mark: 50766K(87424K)] 55237K(126720K), 0.0001656 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
9 2020-10-28T20:15:45.090+0800: 0.229: [CMS-concurrent-mark-start]
10 2020-10-28T20:15:45.091+0800: 0.230: [CMS-concurrent-mark: 0.001/0.001 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
11 2020-10-28T20:15:45.091+0800: 0.230: [CMS-concurrent-preclean-start]
12 2020-10-28T20:15:45.091+0800: 0.230: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
13 2020-10-28T20:15:45.091+0800: 0.230: [CMS-concurrent-shortable-preclean-start]
14 2020-10-28T20:15:45.095+0800: 0.234: [GC (Allocation Failure) 2020-10-28T20:15:45.095+0800: 0.234: [ParNew: 38892K->4343K(39296K), 0.0074564 secs]
89659K->67677K(126720K), 0.0075041 secs] [Times: user=0.11 sys=0.02, real=0.01 secs]
15 2020-10-28T20:15:45.109+0800: 0.247: [GC (Allocation Failure) 2020-10-28T20:15:45.109+0800: 0.247: [ParNew: 39287K->4337K(39296K), 0.0173875 secs]
10262K->77044K(126720K), 0.0174535 secs] [Times: user=0.11 sys=0.00, real=0.02 secs]
16 2020-10-28T20:15:45.132+0800: 0.271: [GC (Allocation Failure) 2020-10-28T20:15:45.132+0800: 0.271: [ParNew: 39016K->4343K(39296K), 0.0293211 secs]
11172K->90179K(126720K), 0.0295248 secs] [Times: user=0.19 sys=0.03, real=0.03 secs]
17 2020-10-28T20:15:45.170+0800: 0.308: [GC (Allocation Failure) 2020-10-28T20:15:45.170+0800: 0.308: [ParNew: 39287K->4348K(39296K), 0.0196417 secs]
12512K->102137K(137724K), 0.0197106 secs] [Times: user=0.11 sys=0.00, real=0.02 secs]
18 2020-10-28T20:15:45.195+0800: 0.334: [GC (Allocation Failure) 2020-10-28T20:15:45.195+0800: 0.334: [ParNew: 39292K->4348K(39296K), 0.0083442 secs]
137071K->115691K(151156K), 0.0083989 secs] [Times: user=0.03 sys=0.00, real=0.01 secs]
19 2020-10-28T20:15:45.210+0800: 0.349: [GC (Allocation Failure) 2020-10-28T20:15:45.210+0800: 0.349: [ParNew: 39292K->4346K(39296K), 0.0076887 secs]
150635K->117571K(162800K), 0.0077468 secs] [Times: user=0.13 sys=0.00, real=0.01 secs]
20 2020-10-28T20:15:45.223+0800: 0.362: [GC (Allocation Failure) 2020-10-28T20:15:45.223+0800: 0.362: [ParNew: 39290K->4351K(39296K), 0.0118021 secs]
162355K->144469K(180036K), 0.0118577 secs] [Times: user=0.11 sys=0.02, real=0.01 secs]
21 2020-10-28T20:15:45.241+0800: 0.380: [GC (Allocation Failure) 2020-10-28T20:15:45.241+0800: 0.380: [ParNew: 39295K->4332K(39296K), 0.0085161 secs]
179413K->157813K(215520K), 0.0085681 secs] [Times: user=0.11 sys=0.00, real=0.01 secs]
22 2020-10-28T20:15:45.255+0800: 0.394: [GC (Allocation Failure) 2020-10-28T20:15:45.255+0800: 0.394: [ParNew: 39090K->4349K(39296K), 0.0077923 secs]
192444K->169745K(205380K), 0.0079516 secs] [Times: user=0.13 sys=0.00, real=0.01 secs]
23 2020-10-28T20:15:45.269+0800: 0.408: [GC (Allocation Failure) 2020-10-28T20:15:45.269+0800: 0.408: [ParNew: 39102K->4350K(39296K), 0.0068402 secs]
204497K->179828K(215520K), 0.0069002 secs] [Times: user=0.09 sys=0.03, real=0.01 secs]
24 2020-10-28T20:15:45.282+0800: 0.420: [GC (Allocation Failure) 2020-10-28T20:15:45.282+0800: 0.420: [ParNew: 39170K->4351K(39296K), 0.0083724 secs]
214712K->192744K(228336K), 0.0084326 secs] [Times: user=0.08 sys=0.03, real=0.01 secs]
```

从上图可以看出，CMS GC只发生了一次，其他的都是年轻代的GC。当我把内存配置为256M时，则发生了十几次。从上图，可以发现其年轻代GC名称为ParNew，同时采用**标记-复制**算法。而其老年代采用的是**并发标记-清除**算法。同时在CMSGC发生时，经历了6个步骤：

1. Initial Mark(初始标记)，会伴随STW
2. Concurrent Mark (并发标记)
3. Concurrent PreClean (并发预清理)
4. Final Mark (最终标记)
5. Concurrent Sweep (并发清除)
6. Concurrent Reset (并发重置)

上图可以看出CMS的暂停时间很短，这也符合它设计的目标：

1. 不对老年代进行整理，使用空闲列表管理内存回收（也因此而产生了很多老年代碎片）
2. 在标记-清除时，大部分工作和应用线程一起并发执行

## G1日志分析

- 使用如下命令生成G1GC日志文件：

```
java -Xloggc:gc.cmsdemo.log -XX:+UseG1GC -XX:+PrintGC -XX:+PrintGCDateStamps
-dfile.encoding=utf-8 GCLogAnalysis
```

其日志文件如下：

```
1 Java HotSpot(TM) 64-Bit Server VM (25.161-b12) for windows-amd64 JRE (1.8.0_161-b12), built on Dec 19 2017 17:52:25 by "java_re" with MS VC++ 10.0 (VS2010)
2 Memory: 4k page, physical 8295652k (1861560k free), swap 14753516k (2802292k free)
3 Commandline flags: -XX:InitialHeapSize=134217728 -XX:MaxHeapSize=536870912 -XX:+PrintGC -XX:+PrintGCDateStamps -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers
-XX:+PrintGCDateStamps -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseConcMarkSweepGC
-XX:-UseLargePagesIndividualAllocation -XX:+UseParNewGC
4 2020-10-28T21:02:57.094+0800: 0.171: [GC pause (G1 Evacuation Pause) (young) 29M->9708K(128M), 0.0109979 secs]
5 2020-10-28T21:02:57.130+0800: 0.207: [GC pause (G1 Evacuation Pause) (young) 85M->47M(205M), 0.0167032 secs]
6 2020-10-28T21:02:57.189+0800: 0.265: [GC pause (G1 Evacuation Pause) (young) 161M->87M(280M), 0.0098701 secs]
8 2020-10-28T21:02:57.227+0800: 0.304: [GC pause (G1 Evacuation Pause) (young) 165M->107M(327M), 0.0071767 secs]
9 2020-10-28T21:02:57.262+0800: 0.339: [GC pause (G1 Humongous Allocation) (young) (initial-mark) 186M->128M(364M), 0.0062496 secs]
10 2020-10-28T21:02:57.269+0800: 0.345: [GC concurrent-root-region-scan-start]
11 2020-10-28T21:02:57.269+0800: 0.346: [GC concurrent-root-region-scan-end, 0.0001641 secs]
12 2020-10-28T21:02:57.269+0800: 0.346: [GC concurrent-mark-start]
13 2020-10-28T21:02:57.270+0800: 0.347: [GC concurrent-mark-end, 0.0011362 secs]
14 2020-10-28T21:02:57.270+0800: 0.347: [GC remark, 0.0018311 secs]
15 2020-10-28T21:02:57.273+0800: 0.350: [GC cleanup 132M->132M(364M), 0.0048585 secs]
16 2020-10-28T21:02:57.322+0800: 0.398: [GC pause (G1 Evacuation Pause) (young) 301M->193M(425M), 0.0243899 secs]
17 2020-10-28T21:02:57.347+0800: 0.423: [GC pause (G1 Humongous Allocation) (young) (initial-mark) 184M->182M(443M), 0.0051541 secs]
18 2020-10-28T21:02:57.352+0800: 0.429: [GC concurrent-root-region-scan-start]
19 2020-10-28T21:02:57.352+0800: 0.429: [GC concurrent-root-region-scan-end, 0.0000063 secs]
20 2020-10-28T21:02:57.352+0800: 0.429: [GC concurrent-mark-start]
21 2020-10-28T21:02:57.353+0800: 0.429: [GC concurrent-mark-end, 0.0013773 secs]
22 2020-10-28T21:02:57.354+0800: 0.430: [GC remark, 0.0030136 secs]
23 2020-10-28T21:02:57.358+0800: 0.434: [GC cleanup 185M->185M(443M), 0.0047501 secs]
24 2020-10-28T21:02:57.429+0800: 0.506: [GC pause (G1 Evacuation Pause) (young) 366M->238M(470M), 0.0131709 secs]
25 2020-10-28T21:02:57.444+0800: 0.521: [GC pause (G1 Evacuation Pause) (mixed) 243M->238M(479M), 0.0054614 secs]
26 2020-10-28T21:02:57.452+0800: 0.529: [GC pause (G1 Humongous Allocation) (young) (initial-mark) 242M->235M(486M), 0.0019215 secs]
27 2020-10-28T21:02:57.454+0800: 0.531: [GC concurrent-root-region-scan-start]
28 2020-10-28T21:02:57.454+0800: 0.531: [GC concurrent-root-region-scan-end, 0.0001079 secs]
29 2020-10-28T21:02:57.454+0800: 0.531: [GC concurrent-mark-start]
30 2020-10-28T21:02:57.456+0800: 0.532: [GC concurrent-mark-end, 0.0017379 secs]
31 2020-10-28T21:02:57.456+0800: 0.533: [GC remark, 0.0041230 secs]
32 2020-10-28T21:02:57.461+0800: 0.538: [GC cleanup 243M->243M(486M), 0.0059078 secs]
33 2020-10-28T21:02:57.502+0800: 0.579: [GC pause (G1 Evacuation Pause) (young) 405M->283M(502M), 0.0100157 secs]
34 2020-10-28T21:02:57.513+0800: 0.592: [GC pause (G1 Evacuation Pause) (mixed) 292M->263M(504M), 0.0074399 secs]
35 2020-10-28T21:02:57.523+0800: 0.600: [GC pause (G1 Humongous Allocation) (young) (initial-mark) 264M->263M(506M), 0.0012611 secs]
36 2020-10-28T21:02:57.524+0800: 0.601: [GC concurrent-root-region-scan-start]
37 2020-10-28T21:02:57.524+0800: 0.601: [GC concurrent-root-region-scan-end, 0.0001130 secs]
38 2020-10-28T21:02:57.524+0800: 0.601: [GC concurrent-mark-start]
39 2020-10-28T21:02:57.524+0800: 0.603: [GC concurrent-mark-end, 0.0017528 secs]
40 2020-10-28T21:02:57.526+0800: 0.603: [GC remark, 0.0036764 secs]
41 2020-10-28T21:02:57.531+0800: 0.607: [GC cleanup 268M->268M(506M), 0.0039996 secs]
```

通过上图可以看出，其与CMSGC产生的日志有些相似之处，可以清楚地看到GC的各个阶段

1. Evacuation Pause : young（纯年轻代模式转移暂停）

2. Concurrent Marking (并发标记)
  - Initial Mark (初始标记)
  - Root Region Scan (Root区扫描)
  - Concurrent Mark (并发标记)
  - Remark (再次标记)
  - CleanUp (清理)
3. Evacuation Pause (转移暂停: 混合模式)
4. Full GC (Allocation Failure)

## 总结

- Parallel GC

并行垃圾收集器，其对年轻代采用的是**标记-复制算法**，老年代采用的是**标记-清除-整理算法**，同时在做垃圾回收时会**触发STW**。

它适用于多核服务器，其主要目标是**增加系统吞吐量**(也就是降低GC总体消耗的时间)。为了达成这个目标，会尽量使用尽可能多的CPU资源：

- 在GC事件执行期间，所有 CPU 内核都在并行地清理垃圾，所以暂停时间相对来说更短
- 在两次GC事件中间的间隔期，不会启动GC线程，所以这段时间内不会消耗任何 系统资源

另一方面，因为并行GC的所有阶段都不能中断，所以并行GC很可能会出现长时间的卡顿。长时间卡顿的意思，就是并行GC启动后，一次性完成所有的GC操作，所以单次暂停的时间较长。假如系统延迟是非常重要的性能指标，那么就应该选择其他垃圾收集器

- Serial GC

串行垃圾收集器，其对年轻代采用的是**标记-复制算法**，老年代采用的是**标记-清除-整理算法**，在做垃圾回收时会**触发STW**。

它只适用于一些小内存jvm的情况，而且是单核的CPU比较有用。因为它是单线程做垃圾收集，所以无法发挥出多核CPU的优势，会存在垃圾收集暂停时间长、效率低下的问题。

- CMS GC

CMS垃圾收集器，其对年轻代采用的是**标记-复制算法**，老年代采用的是**并行标记-清除算法**，同时在做回收时**只有在部分阶段会触发STW**。

它可以避免在老年代收集时出现长时间卡顿，因此其比较适合用于一些追求**低延迟**的业务场景。而由于其没有对老年代进行整理，因此也会造成老年代内存产生碎片。

- G1 GC

G1在开始运行时，会调整自己的回收策略和行为，以达到稳定控制暂停时间的目的。在年轻代回收时，会进行转移（也就是拷贝）。而后其标记过程则与CMS类似，同样会经历STW。最终再做清理工作。

## 2. 压测 gateway-server-0.0.1-SNAPSHOT.jar

统一使用50个并发，压了60秒。在8G内存4核的机器上跑出来的结果如下：

最大内存	最小内存	GC名称	结果表现
Xmx512m	Xms512m	并行GC	RPS: 2649.8 (requests/second) Max: 278ms Min: 0ms Avg: 3.2ms
Xmx512m	Xms512m	串行GC	RPS: 3155 (requests/second) Max: 277ms Min: 0ms Avg: 2.8ms
Xmx512m	Xms512m	G1	RPS: 3248.5 (requests/second) Max: 1881ms Min: 0ms Avg: 2.3ms
Xmx512m	Xms512m	CMS	RPS: 3749.4 (requests/second) Max: 300ms Min: 0ms Avg: 1.6ms
Xmx1g	Xms1g	并行GC	RPS: 3808.8 (requests/second) Max: 318ms Min: 0ms Avg: 1.5ms
Xmx1g	Xms1g	串行GC	RPS: 3107.4 (requests/second) Max: 1641ms Min: 0ms Avg: 2.6ms
Xmx1g	Xms1g	G1	RPS: 3424.6 (requests/second) Max: 366ms Min: 0ms Avg: 1.8ms
Xmx1g	Xms1g	CMS	RPS: 3461.7 (requests/second) Max: 334ms Min: 0ms Avg: 2ms
Xmx2g	Xms2g	并行GC	RPS: 3229.5 (requests/second) Max: 287ms Min: 0ms Avg: 2.2ms

最大内存	最小内存	GC名称	结果表现
Xmx2g	Xms2g	串行GC	RPS: 2857.9 (requests/second) Max: 411ms Min: 0ms Avg: 2.6ms
Xmx2g	Xms2g	G1	RPS: 2852.8 (requests/second) Max: 456ms Min: 0ms Avg: 3.3ms
Xmx2g	Xms2g	CMS	RPS: 3231.5 (requests/second) Max: 312ms Min: 0ms Avg: 2.3ms

通过上面的表格可以看出，在内存从512m增大到1g时，不论使用什么GC都会使得性能得到一些提升，但是当内存再升到2g时，则会导致性能反而降低了。另外，在此不同内存的场景下，各GC的表现不一致，在512m时表现最好的是串行回收器，在1g时表现最好的是并行回收器，在2g场景下表现最好的是CMS。