查看默认GC回收器

```
java -XX:+PrintCommandLineFlags -version
-XX:InitialHeapSize=536870912 -XX:MaxHeapSize=8589934592
-XX:+PrintCommandLineFlags -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -XX:+UseParallelG
java version "1.8.0_271"
Java(TM) SE Runtime Environment (build 1.8.0_271-b09)
Java HotSpot(TM) 64-Bit Server VM (build 25.271-b09, mixed mode)
```

```
各个版本默认的垃圾回收器:
jdk1.8以及jdk1.8+    是ParallelGC
jdk1.9默认是G1回收机制
jdk11+       zgc默认回收机制
```

```
-XX:+PrintGC 输出GC日志
-XX:+PrintGCDetails 输出GC的详细日志
-XX:+PrintGCTimeStamps 输出GC的时间戳（以基准时间的形式）
-XX:+PrintGCDateStamps 输出GC的时间戳（以日期的形式，如 2013-05-04T21:53:59.234+0800）
-XX:+PrintHeapAtGC 在进行GC的前后打印出堆的信息
-Xloggc:../logs/gc.log 日志文件的输出路径
-XX:-UseAdaptiveSizePolicy 禁用动态调整，使SurvivorRatio可以起作用
-XX:SurvivorRatio=8 设置Eden:Survivior=8
-XX:NewSize=10M -XX:MaxNewSize=10M 设置整个新生代的大小为10M
 -XX:MaxGCPauseMillis=n: 最大GC停顿时间,毫秒值 (G1)
 -XX:InitatingHeapOccupancyPercent=n: 当堆空间占用到n兆时就触发GC(45)
```

```
UseParNewGC
默认回收组合: ParNew+SerialGC 在Jdk1.8版本中使用,
会提示: Java HotSpot(TM) 64-Bit Server VM warning:
Using the ParNew young collector with the Serial old collector
is deprecated and will likely be removed in a future release
```

```
UseSerialGC
默认回收组合: Serial+SerialOld组合
```

```
UseConcMarkSweepGC
默认回收组合: 使用ParNew+CMS+Serial Old组合并发收集，优先使用ParNew+CMS，当用户线程内存不足时，采用备用方案Seria
```

```
java -XX:+UseSerialGC -XX:+PrintGCDetails -Xms256m -Xmx256m GCLogAnalysis
进行10次youngGC后，就一直执行FullFGC，最后OOM
java -XX:+UseSerialGC -XX:+PrintGCDetails -Xms512m -Xmx512m GCLogAnalysis
执行15次youngGC后，执行了一次FullGC，然后YoungGC5次后，执行了5次左右的FullGC，执行结束!共生成对象次数:12790 gc停
java -XX:+UseSerialGC -XX:+PrintGCDetails -Xms1g -Xmx1g GCLogAnalysis
一直执行的是youngGC。执行15次YoungGC后，执行结束!共生成对象次数:15946 gc停顿时间平均: 0.0368235sec
java -XX:+UseSerialGC -XX:+PrintGCDetails -Xms2g -Xmx2g GCLogAnalysis
一直执行的是youngGC。执行7次YoungGC后，执行结束!共生成对象次数:14804 gc停顿时间平均: 0.06sec


 java -XX:+UseParallelGC -XX:+PrintGCDetails -Xms256m -Xmx256m GCLogAnalysis
 进行10次younggc后，执行了1次FullGC后，又执行了两次younggc后，一直执行fullGC，最后oom。 fullgc停顿时间0.1sec
 java -XX:+UseParallelGC -XX:+PrintGCDetails -Xms512m -Xmx512m GCLogAnalysis
```

```
进行11次youngGC后，执行了1次fullGC后，执行4次younggc，再次执行fullgc，fullgc和（两到三次）younggc交替执行一段时
java -XX:+UseParallelGC -XX:+PrintGCDetails -Xms1g -Xmx1g GCLogAnalysis
进行11次youngGC后，执行了1次fullGC后，执行4次younggc，再次执行fullgc，fullgc和（七到9次左右）younggc交替执行一
java -XX:+UseParallelGC -XX:+PrintGCDetails -Xms2g -Xmx2g GCLogAnalysis
只进行了younggc，执行11次younggc，执行结束!共生成对象次数:17182


java -XX:+UseConcMarkSweepGC -XX:+PrintGCDetails -Xms512m -Xmx512m GCLogAnalysis
先经历大约4次的ParNew（标记复制算法）
完整的阶段:
[GC (CMS Initial Mark) [1 CMS-initial-mark: 164004K(174784K)] 172851K(253440K), 0.0001483 secs] [Ti
（初始化标记）标记老年代中根对象，以及被年轻代中存活引用的对象。
[CMS-concurrent-mark-start]（开始并发标记）从initial mark 标记的根对象开始，标记存活的对象，并发标记，与应用线
[CMS-concurrent-mark: 0.001（花费的cpu时间）/0.001（花费的系统时间） secs] [Times: user=0.00 sys=0.00, r
[CMS-concurrent-preclean-start]（并发与处理阶段 开始）
是与应用线程并发执行的，不用暂停应用线程。由于上一节阶段的并发标记，也是与应用线程同步进行的，所以对象的引用关系可能会
jvm采取卡片的方式将发生变化的区域标记为“脏”区。
[CMS-concurrent-preclean: 0.000（花费的cpu时间）/0.000 （花费的系统时间）secs] [Times: user=0.00 sys=0.0
[CMS-concurrent-abortable-preclean-start]（并法与处理阶段。可能会有一些对象从新生代晋升到老年代， 或者有一些对
[CMS-concurrent-abortable-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (CMS Final Remark) （最终标记）本阶段的目标是完成所有老年代的存活对象的标记。是第二次发生STW的阶段。
由于之前清理阶段是并发执行，Gc线程处理速度可能跟不上应用线程的改变，所以需要一次STW来处理复杂的情况。
通常 CMS 会尝试在年轻代尽可能空的情况下执行 Final Remark 阶段，以免连续触发多次 STW 事件。
[YG occupancy: 15817 K (78656 K)][Rescan (parallel) , 0.0001950 secs][weak refs processing, 0.00000
[class unloading, 0.0001367 secs][scrub symbol table, 0.0002230 secs][scrub string table, 0.0000750
[1 CMS-remark: 164004K(174784K)] 179822K(253440K), 0.0006769 secs] [Times: user=0.00 sys=0.00, real
[CMS-concurrent-sweep-start]（并发清除）
[CMS-concurrent-sweep: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[CMS-concurrent-reset-start]（并发重置）
[CMS-concurrent-reset: 0.000/0.000 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
[GC (Allocation Failure) [ParNew (promotion failed): 78654K->78654K(78656K), 0.0072913 secs][CMS: 1


java -XX:+UseG1GC -XX:+PrintGCDetails -Xms1g -Xmx1g GCLogAnalysis
G1的首要目标是为需要大量内存的系统提供一个保证GC低延迟的解决方案
[GC pause (G1 Evacuation Pause) (young), 0.0060223 secs] （程序运行0.0060223后发生一个Evacuation Pause
   [Parallel Time: 5.4 ms, GC Workers: 8]
      [GC Worker Start (ms): Min: 108.8, Avg: 108.9, Max: 108.9, Diff: 0.1]
      [Ext Root Scanning (ms): Min: 0.0, Avg: 0.2, Max: 0.3, Diff: 0.3, Sum: 1.3]   （每个扫描root的线
      [Update RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]   （每个执行更新RS（Remember
         [Processed Buffers: Min: 0, Avg: 0.0, Max: 0, Diff: 0, Sum: 0]     （每个工作线程执行UB（Updat
      [Scan RS (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]     （每个工作线程扫描RS的耗时）
      [Code Root Scanning (ms): Min: 0.0, Avg: 0.0, Max: 0.0, Diff: 0.0, Sum: 0.0]      （每个工作线程
      [Object Copy (ms): Min: 4.7, Avg: 4.9, Max: 5.0, Diff: 0.3, Sum: 38.8]
      [Termination (ms): Min: 0.0, Avg: 0.2, Max: 0.4, Diff: 0.4, Sum: 1.7]       （每个工作线程执行终止
         [Termination Attempts: Min: 1, Avg: 1.0, Max: 1, Diff: 0, Sum: 8]
      [GC Worker Other (ms): Min: 0.0, Avg: 0.0, Max: 0.1, Diff: 0.1, Sum: 0.3]
      [GC Worker Total (ms): Min: 5.2, Avg: 5.3, Max: 5.4, Diff: 0.1, Sum: 42.2]
      [GC Worker End (ms): Min: 114.1, Avg: 114.2, Max: 114.2, Diff: 0.1]
   [Code Root Fixup: 0.0 ms]
   [Code Root Purge: 0.0 ms]
   [Clear CT: 0.1 ms]                （ 清理CT（Card Table）的耗时）
   [Other: 0.5 ms]                   （其他任务（上述未统计的内容）的耗时）
      [Choose CSet: 0.0 ms]
```

```
68        [Ref Proc: 0.1 ms]
69        [Ref Enq: 0.0 ms]
70        [Redirty Cards: 0.1 ms]
71        [Humongous Register: 0.1 ms]
72        [Humongous Reclaim: 0.0 ms]
73        [Free CSet: 0.0 ms]
74     [Eden: 51.0M(51.0M)->0.0B(44.0M) Survivors: 0.0B->7168.0K Heap: 65.8M(1024.0M)->26.5M(1024.0M)]
75   [Times: user=0.01 sys=0.03, real=0.01 secs]
76
```

Serial 收集器

算法:

年轻代采用“复制算法”;

老年代采用“标记-整理算法”单线程收集器，只会使用一个 CPU 或一条收集线程去完成垃圾收集工作

缺点：垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束(又称为「Stop The World」)

优点是简单而高效(与其他收集器的单线程相比)，对于限定单个 CPU 的环境来说，Serial 收集器由于没有线程交互的开销，专心做垃圾收集自然就可以获得最高的单线程收集效率

ParNew 收集器（ParNew 收集器就是 Serial 收集器的多线程版本）

算法:

年轻代采用“复制算法”

只能用于新生代

多线程收集，并行

在多 CPU 环境下，随着 CPU 的数量增加，它对于 GC 时系统资源的有效利用是有益的。它默认开启的收集线程数与 CPU 的数量相同

效率：

由于存在线程交互的开销，该收集器在通过超线程技术实现的两个 CPU 的环境中都不能百分之百地保证可以超越 ParNew 收集器在单 CPU 的环境中绝对不会有比 Serial 收集器有更好的效果

Parallel Scavenge 收集器

算法:年轻代采用“复制算法”;配合收集器:ParallelOldGC，老年代采用“标记-整理算法”。多线程收集器，自适应调节策略（可以关闭） 吞吐量较高的垃圾回收器

Parallel Scavenge 收集器无法与 CMS 收集器配合使用

CMS 收集器

初始标记:

仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要“STW”。

并发标记

进行 GC Roots 追溯所有对象的过程，在整个过程中耗时最长

重新标记

为了修正并发标记期间因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短 。此阶段也需要“STW”

并发清除

特点

并行与并发

Concurrent Mark Sweep，基于“标记-清除”算法实现

各阶段耗时:并发标记/并发清除 > 重新标记 > 初始标记

对 CPU 资源非常敏感

标记-清除算法导致的空间碎片

并发收集、低停顿，因此 CMS 收集器也被称为并发低停顿收集器

无法处理浮动垃圾，可能出现“Concurrent Mode Failure”失败而导致另一次 Full GC 的产生

由于整个过程中耗时最长的并发标记和并发清除过程收集器线程都可以与用户线程一起工作;所以，从总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发执行的。

G1 收集器

初始标记

仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，需要STW

并发标记

进行 GC Roots 追溯所有对象的过程，可与用户程序并发执行

最终标记

修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录

筛选回收

对各个Region的回收价值和成本进行排序，根据用户所期望的 GC 停顿时间来指定回收计划

特点

空间整合:整体上看来是基于“标记-整理”算法实现的，从局部(两个Region)上看来是基于“复制”算法实现的

并行与并发、分代收集、可预测的停顿

G1将整个Java堆(包括新生代、老年代)划分为多个大小相等的内存块(Region)，

每个Region 是逻辑连续的一段内存，在后台维护一个优先列表，每次根据允许的收集时间，优先 回收垃圾最多的区域

每个Region可能是做为eden区也有可能是老年代，region的区的划分是动态的。

Minor GC 与 Full GC 的区别

Minor GC 称为新生代GC，是发生在新生代的垃圾回收动作，Java 对象绝大多数具备朝生夕灭的特性，Minor GC 触发非常频繁，并且回收速度很快。

Full GC 称为老年代GC，发生在老年代，出现了 Full GC，通常会伴随一次 Minor GC。Full GC的速度一般比 Minor GC 慢 10 倍以上。