JVM参数

3.1.1 标准参数

```
-version
-help
-server
-cp
```

```
[root@localhost ~]# java -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
```

3.1.2 -X参数

> 非标准参数，也就是在JDK各个版本中可能会变动

```
-Xint      解释执行
-Xcomp     第一次使用就编译成本地代码
-Xmixed    混合模式，JVM自己来决定
```

```
[root@localhost ~]# java -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
[root@localhost ~]# java -Xint -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, interpreted mode)
[root@localhost ~]# java -Xcomp -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, compiled mode)
[root@localhost ~]# java -Xmixed -version
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
```

3.1.3 -XX参数

> 使用得最多的参数类型
>
> 非标准化参数，相对不稳定，主要用于JVM调优和Debug

```
a.Boolean类型
格式：-XX:[+-]<name>              +或-表示启用或者禁用name属性
比如：-XX:+UseConcMarkSweepGC     表示启用CMS类型的垃圾回收器
      -XX:+UseG1GC                表示启用G1类型的垃圾回收器
b.非Boolean类型
格式：-XX<name>=<value>表示name属性的值是value
比如：-XX:MaxGCPauseMillis=500
```

3.1.4 其他参数

```
-Xms1000M等价于-XX:InitialHeapSize=1000M
-Xmx1000M等价于-XX:MaxHeapSize=1000M
-Xss100等价于-XX:ThreadStackSize=100
```

> 所以这块也相当于是-XX类型的参数

3.1.5 查看参数

> java -XX:+PrintFlagsFinal -version > flags.txt

```
[root@localhost bin]# java -XX:+PrintFlagsFinal -version > flags.txt
java version "1.8.0_191"
Java(TM) SE Runtime Environment (build 1.8.0_191-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.191-b12, mixed mode)
[root@localhost bin]# sz flags.txt
```

```
[Global flags]
     intx ActiveProcessorCount                      = -1                                   {product}
    uintx AdaptiveSizeDecrementScaleFactor          = 4                                    {product}
    uintx AdaptiveSizeMajorGCDecayTimeScale         = 10                                   {product}
    uintx AdaptiveSizePausePolicy                   = 0                                    {product}
    uintx AdaptiveSizePolicyCollectionCostMargin    = 50                                   {product}
    uintx AdaptiveSizePolicyInitializingSteps       = 20                                   {product}
    uintx AdaptiveSizePolicyOutputInterval          = 0                                    {product}
    uintx AdaptiveSizePolicyWeight                  = 10                                   {product}
    uintx AdaptiveSizeThroughPutPolicy              = 0                                    {product}
    uintx AdaptiveTimeWeight                        = 25                                   {product}
     bool AdjustConcurrency                         = false                                {product}
     bool AggressiveHeap                            = false                                {product}
     bool AggressiveOpts                            = false                                {product}
```

> 值得注意的是"="表示默认值，":="表示被用户或JVM修改后的值
> 要想查看某个进程具体参数的值，可以使用jinfo，这块后面聊
> 一般要设置参数，可以先查看一下当前参数是什么，然后进行修改

3.1.6 设置参数的常见方式

- 开发工具中设置比如IDEA，eclipse
- 运行jar包的时候:java -XX:+UseG1GC xxx.jar
- web容器比如tomcat，可以在脚本中的进行设置
- 通过jinfo实时调整某个java进程的参数(参数只有被标记为manageable的flags可以被实时修改)

3.1.7 实践和单位换算

```
1Byte(字节)=8bit(位)
1KB=1024Byte(字节)
1MB=1024KB
1GB=1024MB
1TB=1024GB
```

```
(1)设置堆内存大小和参数打印
-Xmx100M -Xms100M -XX:+PrintFlagsFinal
(2)查询+PrintFlagsFinal的值
:=true
(3)查询堆内存大小MaxHeapSize
:= 104857600
(4)换算
104857600(Byte)/1024=102400(KB)
102400(KB)/1024=100(MB)
(5)结论
104857600是字节单位
```

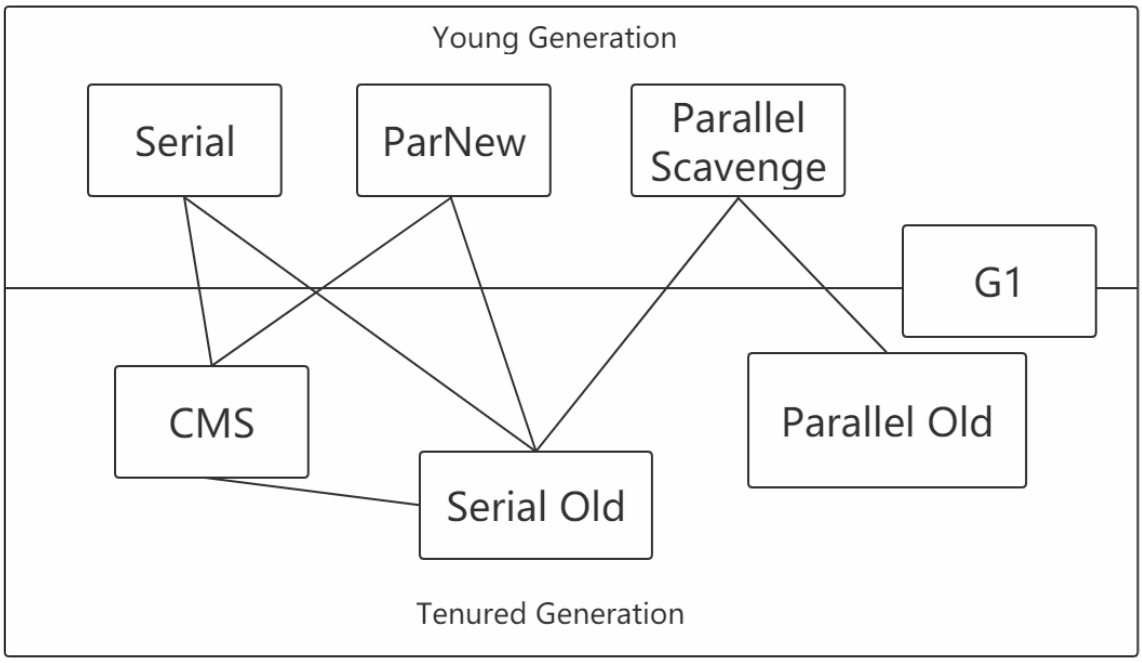
3.1.8 常用参数含义

参数	含义	说明
-XX:CICompilerCount=3	最大并行编译数	如果设置大于1，虽然编译速度会提高，但是同样影响系统稳定性，会增加VM崩溃的可能
-XX:InitialHeapSize=100M	初始化堆大小	简写-Xms100M
-XX:MaxHeapSize=100M	最大堆大小	简写-Xms100M
-XX:NewSize=20M	设置年轻代的大小	
-XX:MaxNewSize=50M	年轻代最大大小	
-XX:OldSize=50M	设置老年代大小	
-XX:MetaspaceSize=50M	设置方法区大小	
-XX:MaxMetaspaceSize=50M	方法区最大大小	
-XX:+UseParallelGC	使用UseParallelGC	新生代，吞吐量优先
-XX:+UseParallelOldGC	使用UseParallelOldGC	老年代，吞吐量优先
-XX:+UseConcMarkSweepGC	使用CMS	老年代，停顿时间优先
-XX:+UseG1GC	使用G1GC	新生代，老年代，停顿时间优先
-XX:NewRatio	新老生代的比值	比如-XX:Ratio=4，则表示新生代:老年代=1:4，也就是新生代占整个堆内存的1/5
-XX:SurvivorRatio	两个S区和Eden区的比值	比如-XX:SurvivorRatio=8，也就是(S0+S1):Eden=2:8，也就是一个S占整个新生代的1/10
-XX:+HeapDumpOnOutOfMemoryError	启动堆内存溢出打印	当JVM堆内存发生溢出时，也就是OOM，自动生成dump文件
-XX:HeapDumpPath=heap.hprof	指定堆内存溢出打印目录	表示在当前目录生成一个heap.hprof文件
-XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -Xloggc:g1-gc.log	打印出GC日志	可以使用不同的垃圾收集器，对比查看GC情况
-Xss128k	设置每个线程的堆栈大小	经验值是3000-5000最佳
-XX:MaxTenuringThreshold=6	提升年老代的最大临界值	默认值为 15
-XX:InitiatingHeapOccupancyPercent	启动并发GC周期时堆内存使用占比	G1之类的垃圾收集器用它来触发并发GC周期,基于整个堆的使用率,而不只是某一代内存的使用比. 值为 0 则表示”一直执行GC循环”. 默认值为 45.
-XX:G1HeapWastePercent	允许的浪费堆空间的占比	默认是10%，如果并发标记可回收的空间小于10%,则不会触发MixedGC。
-XX:MaxGCPauseMillis=200ms	G1最大停顿时间	暂停时间不能太小，太小的话就会导致出现G1跟不上垃圾产生的速度。最终退化成Full GC。所以对这个参数的调优是一个持续的过程，逐步调整到最佳状态。

参数	含义	说明
-XX:ConcGCThreads=n	并发垃圾收集器使用的线程数量	默认值随JVM运行的平台不同而不同
-XX:G1MixedGCLiveThresholdPercent=65	混合垃圾回收周期中要包括的旧区域设置占用率阈值	默认占用率为 65%
-XX:G1MixedGCCountTarget=8	设置标记周期完成后，对存活数据上限为G1MixedGCLIveThresholdPercent的旧区域执行混合垃圾回收的目标次数	默认8次混合垃圾回收，混合回收的目标是要控制在此目标次数以内
-XX:G1OldCSetRegionThresholdPercent=1	描述Mixed GC时，Old Region被加入到CSet中	默认情况下，G1只把10%的Old Region加入到CSet中

垃圾收集器

> 如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

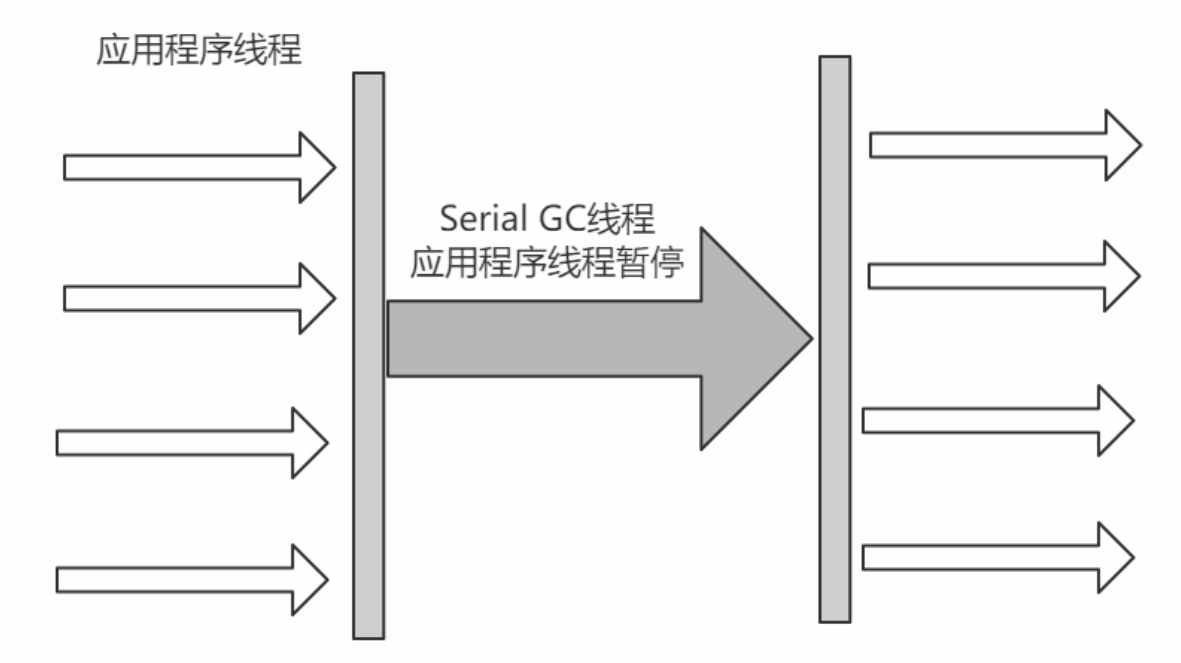


2.5.5.1 Serial

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK1.3.1之前）是虚拟机新生代收集的唯一选择。

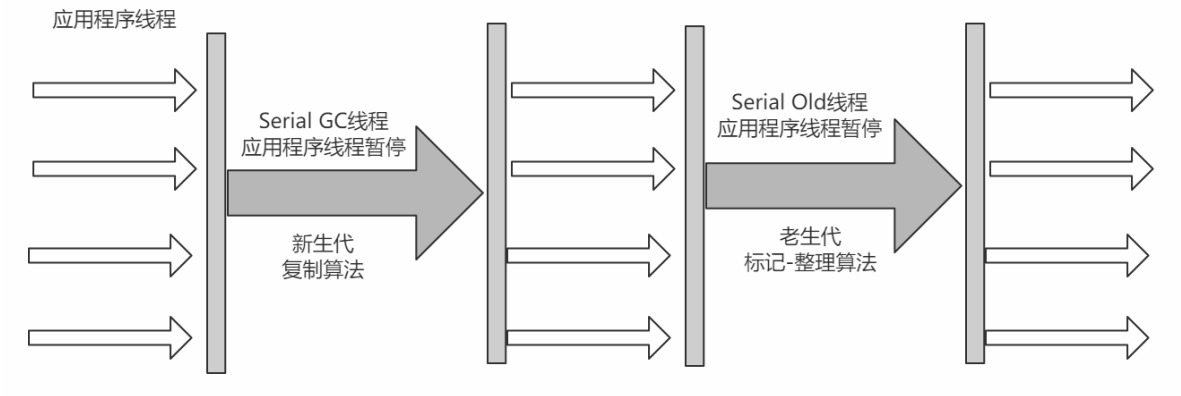
它是一种单线程收集器，不仅仅意味着它只会使用一个CPU或者一条收集线程去完成垃圾收集工作，更重要的是其在进行垃圾收集的时候需要暂停其他线程。

```
优点：简单高效，拥有很高的单线程收集效率
缺点：收集过程需要暂停所有线程
算法：复制算法
适用范围：新生代
应用：Client模式下的默认新生代收集器
```



2.5.5.2 Serial Old

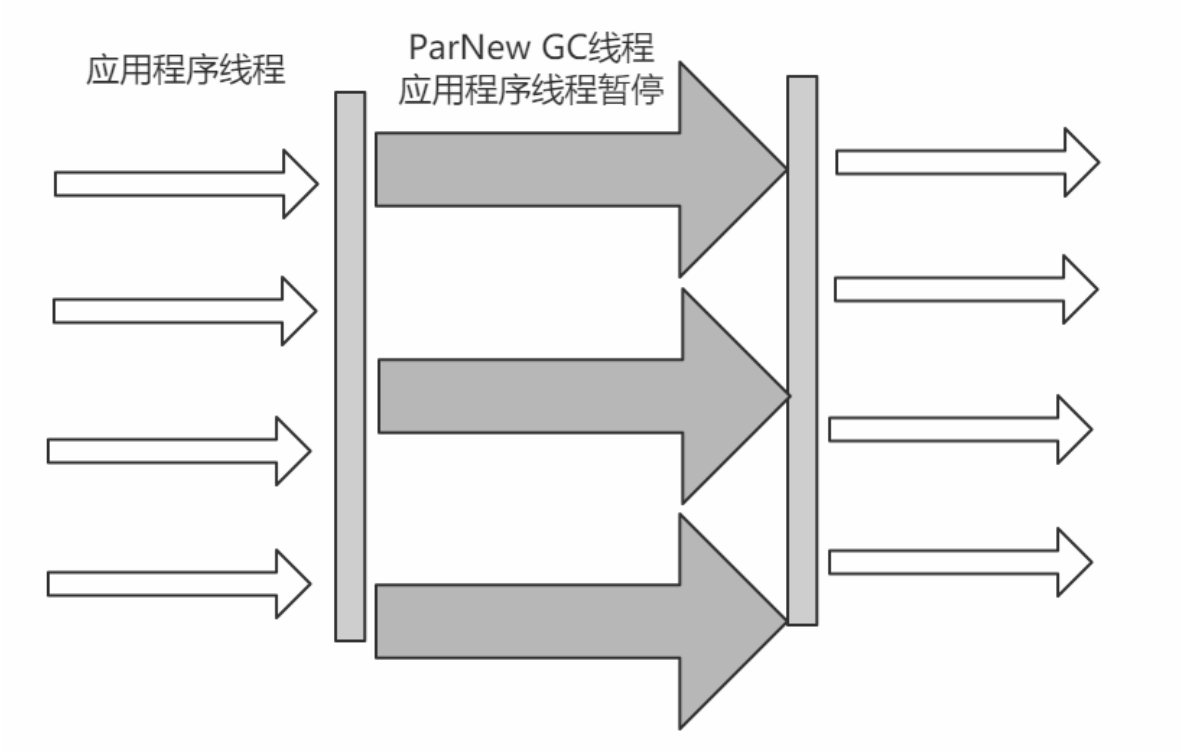
Serial Old收集器是Serial收集器的老年代版本，也是一个单线程收集器，不同的是采用"**标记-整理算法**"，运行过程和Serial收集器一样。



2.5.5.3 ParNew

可以把这个收集器理解为Serial收集器的多线程版本。

```
优点：在多CPU时，比Serial效率高。
缺点：收集过程暂停所有应用程序线程，单CPU时比Serial效率差。
算法：复制算法
适用范围：新生代
应用：运行在Server模式下的虚拟机中首选的新生代收集器
```



2.5.5.4 Parallel Scavenge

Parallel Scavenge收集器是一个新生代收集器，它也是使用复制算法的收集器，又是并行的多线程收集器，看上去和ParNew一样，但是Parallel Scanvenge更关注系统的**吞吐量。**

> 吞吐量=运行用户代码的时间/(运行用户代码的时间+垃圾收集时间)
>
> 比如虚拟机总共运行了100分钟，垃圾收集时间用了1分钟，吞吐量=(100-1)/100=99%。
>
> 若吞吐量越大，意味着垃圾收集的时间越短，则用户代码可以充分利用CPU资源，尽快完成程序的运算任务。

```
-XX:MaxGCPauseMillis控制最大的垃圾收集停顿时间，
-XX:GCTimeRatio直接设置吞吐量的大小。
```

2.5.5.5 Parallel Old

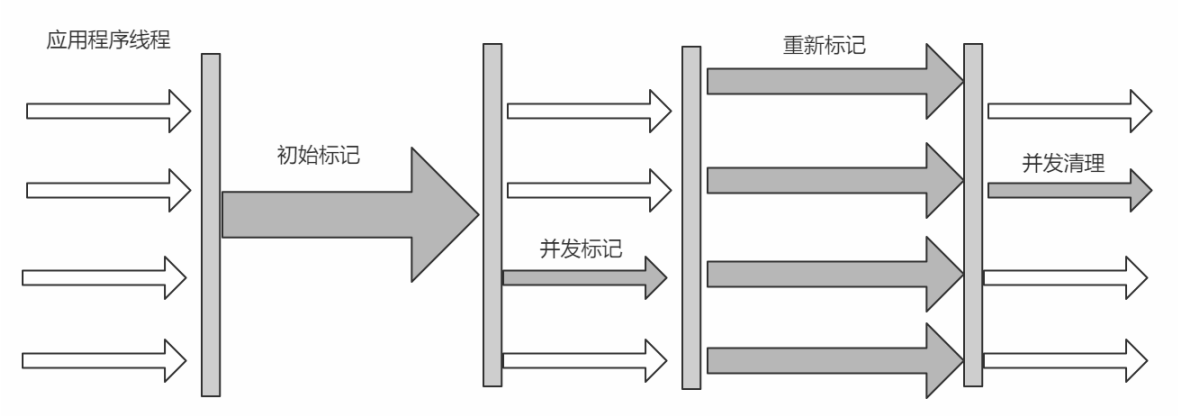
Parallel Old收集器是Parallel Scavenge收集器的老年代版本，使用多线程和**标记-整理算法**进行垃圾回收，也是更加关注系统的**吞吐量。**

2.5.4.6 CMS

> 官网： https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html#concurrent_mark_sweep_cms_collector
>
> CMS(Concurrent Mark Sweep)收集器是一种以获取`最短回收停顿时间`为目标的收集器。
>
> 采用的是"标记-清除算法",整个过程分为4步

```
(1)初始标记 CMS initial mark      标记GC Roots直接关联对象，不用Tracing，速度很快
(2)并发标记 CMS concurrent mark   进行GC Roots Tracing
(3)重新标记 CMS remark            修改并发标记因用户程序变动的内容
(4)并发清除 CMS concurrent sweep 清除不可达对象回收空间，同时有新垃圾产生，留着下次清理称为浮动垃圾
```

> 由于整个过程中，并发标记和并发清除，收集器线程可以与用户线程一起工作，所以总体上来说，CMS收集器的内存回收过程是与用户线程一起并发地执行的。



优点：并发收集、低停顿
缺点：产生大量空间碎片、并发阶段会降低吞吐量，还会并发失败

backgroud模式为正常模式执行上述的CMS GC流程
forefroud模式为Full GC模式

相关参数：

```
//开启CMS垃圾收集器
-XX:+UseConcMarkSweepGC
//默认开启，与-XX:CMSFullGCsBeforeCompaction配合使用
-XX:+UseCMSCompactAtFullCollection
//默认0 几次Full GC后开始整理
-XX:CMSFullGCsBeforeCompaction=0
//辅助CMSInitiatingOccupancyFraction的参数，不然CMSInitiatingOccupancyFraction只会使用一次就恢复自动调整，也就是开启手动调整。
-XX:+UseCMSInitiatingOccupancyOnly
//取值0-100，按百分比回收
-XX:CMSInitiatingOccupancyFraction  默认-1
```

注意：CMS并发GC不是“full GC”。HotSpot VM里对concurrent collection和full collection有明确的区分。所有带有“FullCollection”字样的VM参数都是跟真正的full GC相关，而跟CMS并发GC无关的。

hotspot源码部分我就不截图了，会给到相关的源码包。