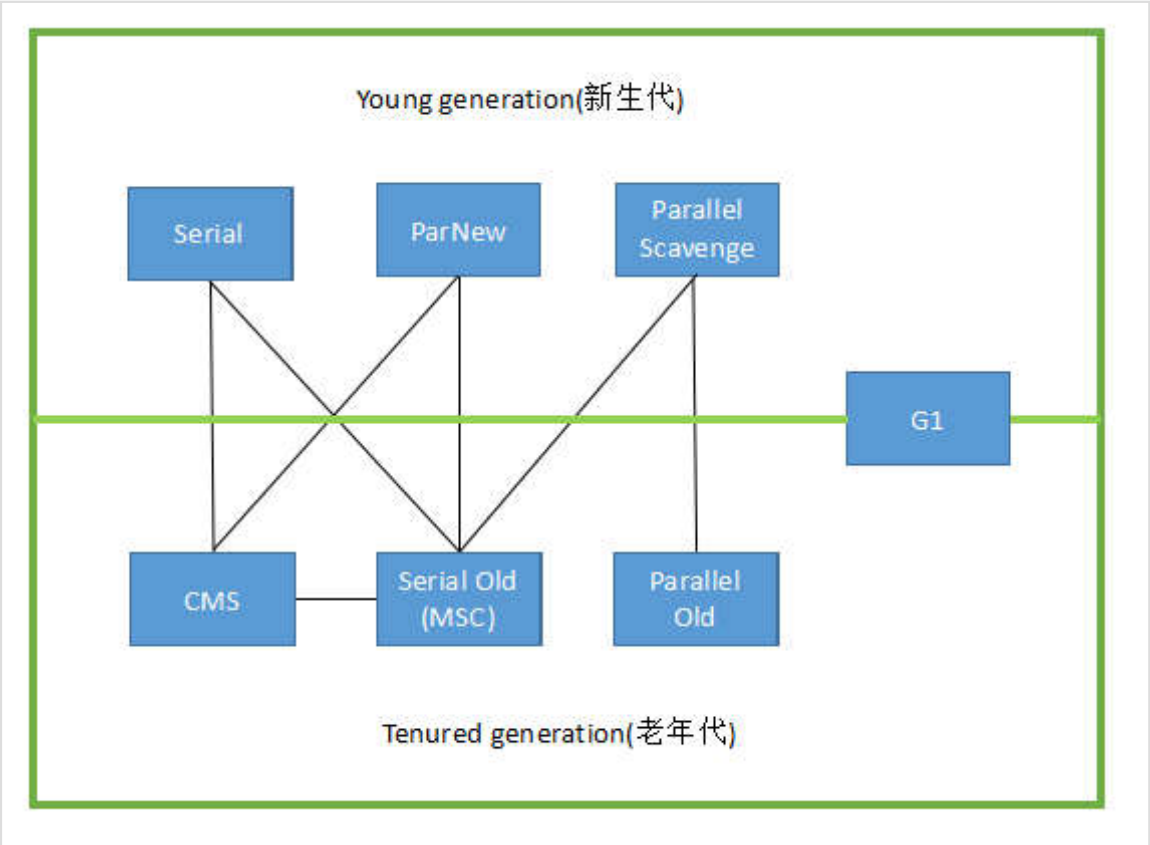
jvm垃圾回收器

2018-07-02 22:00:20 | 2018-08-05 17:58:26 | jvm

概述

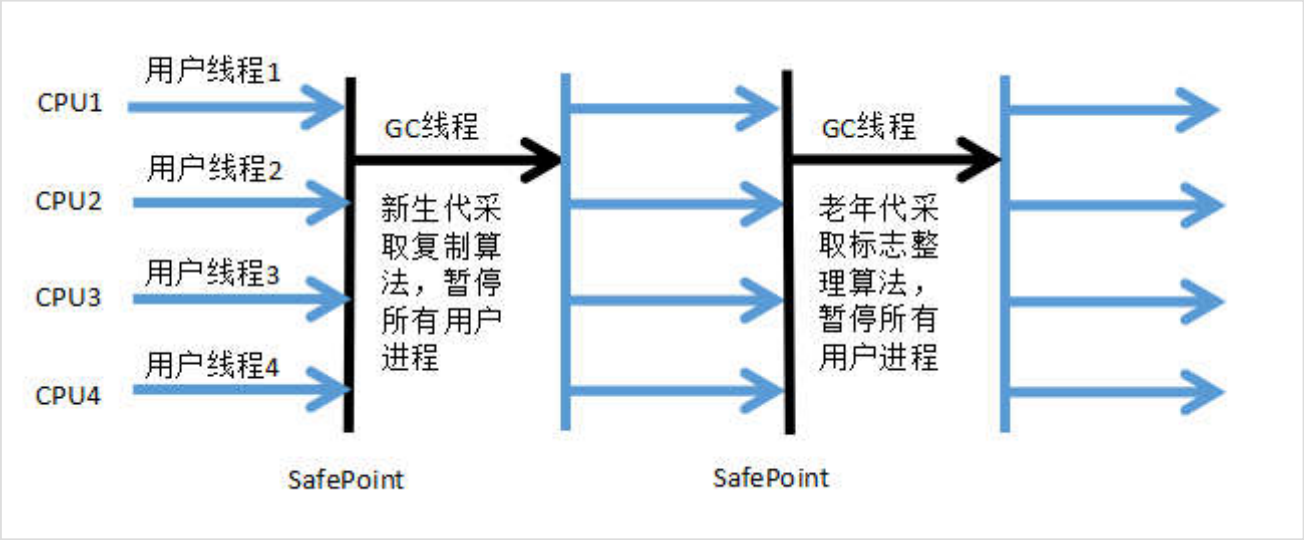
java虚拟机的垃圾回收器多种多样，因不同的厂商、版本而不同。本文主要介绍在HotSpot虚拟机中常用的几种垃圾回收器：Serial、ParNew、Parallel Scavenge、CMS、Serial Old、Parallel Old、G1。这几种垃圾回收器关系如下图所示，其中连线表示两个垃圾回收器可以结合使用。



> 说明：此图参考《深入理解Java虚拟机：JVM高级特性与最佳实践》画出

Serial回收器

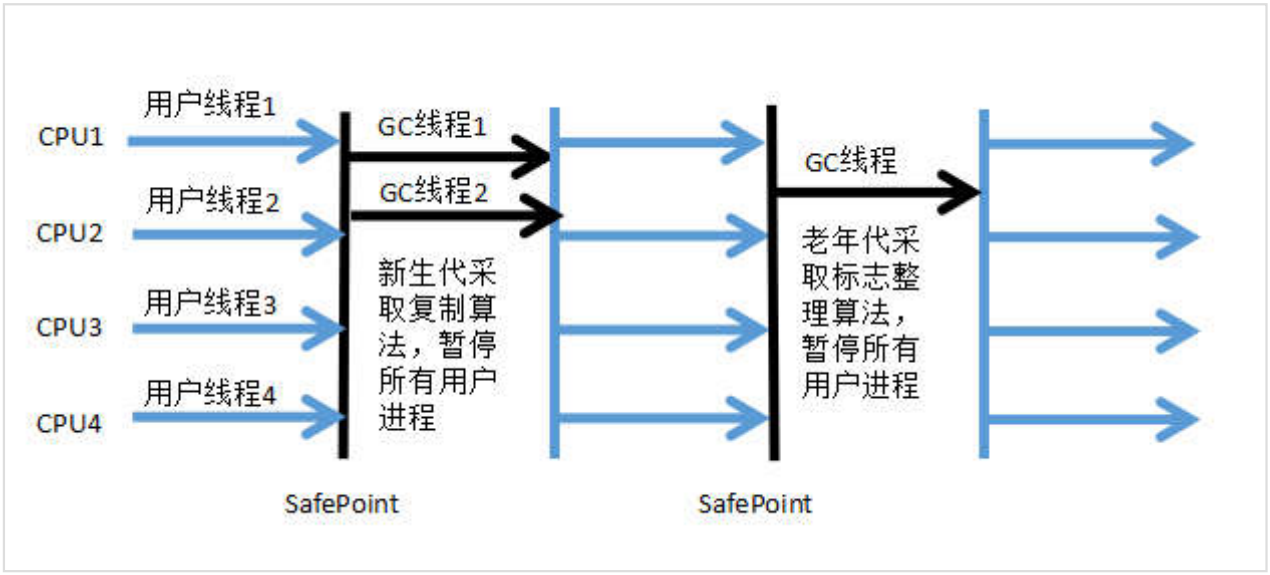
Serial收集器是最基本也是发展历史最悠久的回收器。它相比其他回收器而言显得简单而高效，是虚拟机运行在Client模式下的默认新生代回收器。这是一个单线程的收集器，其在进行垃圾回收时，必须暂停其他所有的工作线程（Stop The World）直到收集结束。其运行过程大致如下图所示



> 说明：此图参考《深入理解Java虚拟机：JVM高级特性与最佳实践》画出

ParNew回收器

ParNew回收器是Serial回收器的多线程版本，是大部分虚拟机运行在Server模式下的首选新生代回收器，同时也是使用-XX:+UseConcMarkSweepGC选项后的默认的新生代回收器，当然，也可以使用-XX:+UseParNewGC选项来强制启动该回收器。由于这是多线程进行回收，存在线程切换的开销，因此当CPU数量较少时，其性能并不比Serial回收器好，甚至比Serial回收器还差，但是当CPU数量比较多时，其性能远远超过Seral回收器，这也是Serial回收器应用于Client端而ParNew回收器应用于Server端的一个原因。当然，可以通过-XX:ParallelGCThreads参数来限制垃圾回收的线程数。其运行过程大致如下图所示



> 说明：此图参考《深入理解Java虚拟机：JVM高级特性与最佳实践》画出

Parallel Scavenge回收器

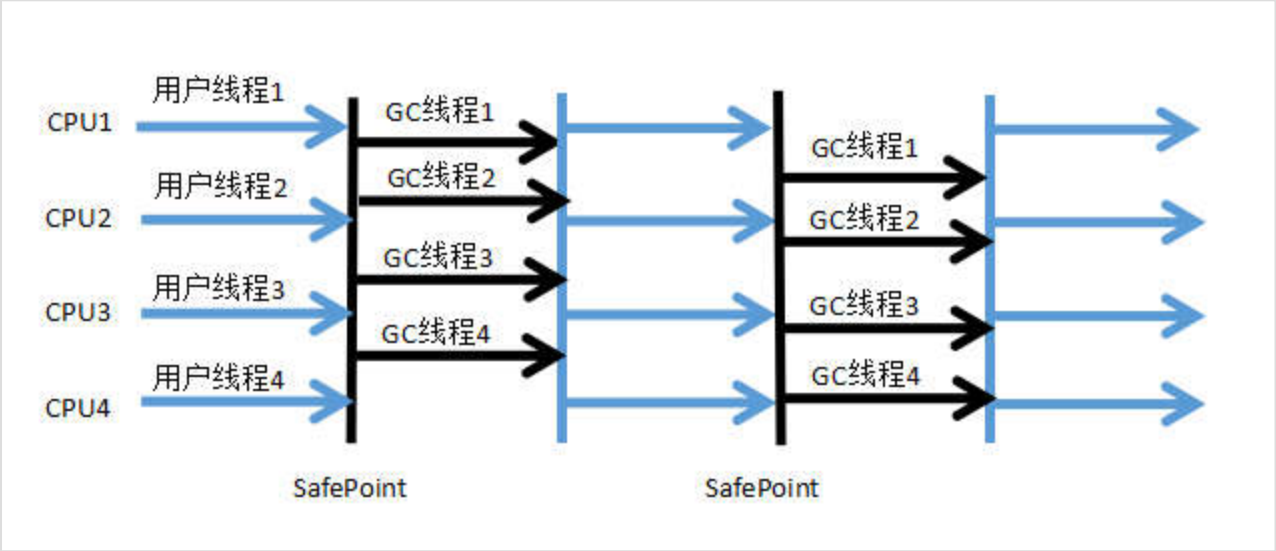
Parallel Scavenge回收器也是一个使用复制算法的新生代的并行多线程的回收器，与ParNew回收器不同的是：ParNew回收器注重的是响应时间，即在垃圾回收过程中用户线程停顿的时间；而Parallel Scavenge回收器注重的是吞吐量，即运行用户的代码消耗的CPU时间占CPU总消耗时间的比率。吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾回收时间）。

Parallel Scavenge回收器提供了以下几个参数用户控制吞吐量

- -XX:UseAdaptiveSizePolicy：此参数是一个开关参数，用于控制是否让虚拟机动态调整相关参数以提供最合适的停顿时间或最大的吞吐量。当此参数打开后，就不需要手工指定新生代的大小（-Xmn）、Eden与Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象大小（-XX:PretenureSizeThreshold）等参数细节。

- -XX:MaxGCPauseMillis：控制最大垃圾回收的停顿时间，停顿时间小是以牺牲新生代的大小为代价换来的。

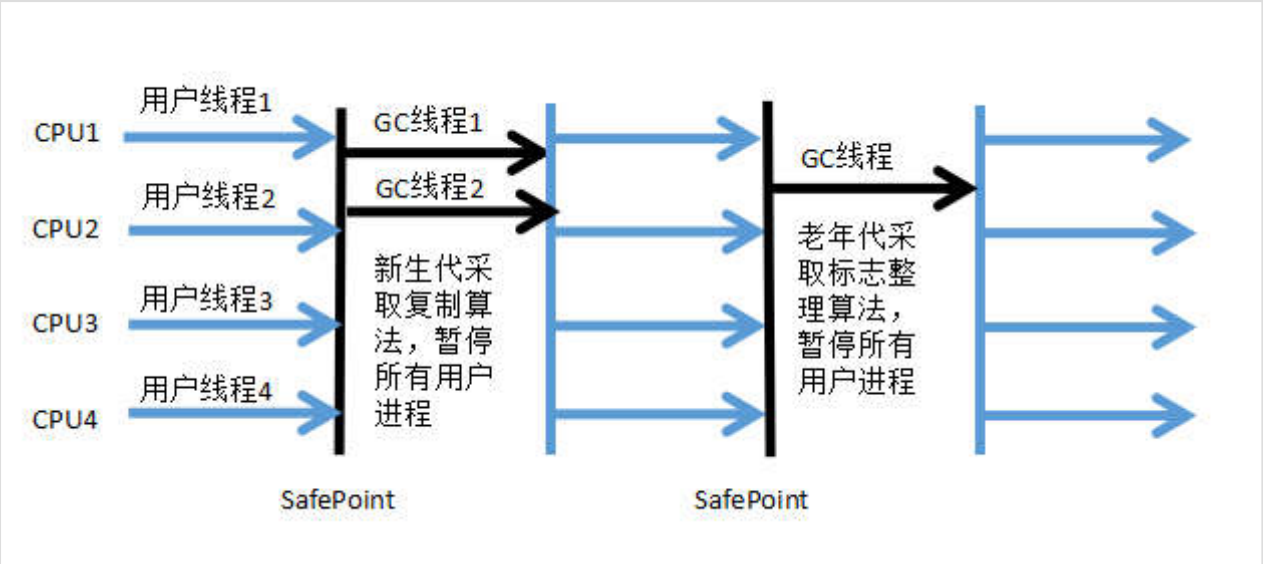
- -XX:GCTimeRatio：一个（0，100）的整数，表示用户线程运行时间与垃圾回收时间的比值，默认值是99，表示允许最大1%（即 1/(1+99)) 的垃圾回收时间。其运行过程图大致如下图所示



> 说明：此图参考《深入理解Java虚拟机：JVM高级特性与最佳实践》画出

Serial Old回收器

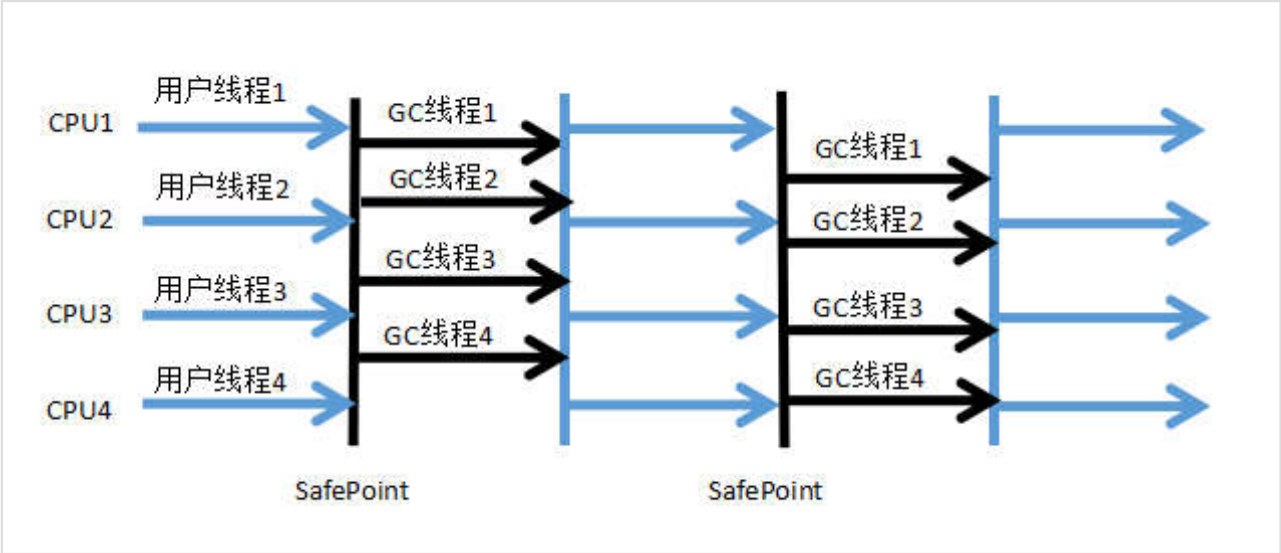
Serial Old回收器是Serial回收器的老年代版本，同样是一个单线程回收器，使用“标记整理”算法。这个回收器的主要意义是给Client模式下的虚拟机使用。如果运行在Server端，还可以作为CMS回收器的后备预案，在并发回收发生Concurrent Mode Failure时使用。其运行过程图大致如下图所示



> 说明：此图参考《深入理解Java虚拟机：JVM高级特性与最佳实践》画出

Parallel Old回收器

Parallel Old回收器是Parallel Scavenge回收器的老年代版本，使用多线程和“标记整理”算法。其运行过程图大致如下图所示



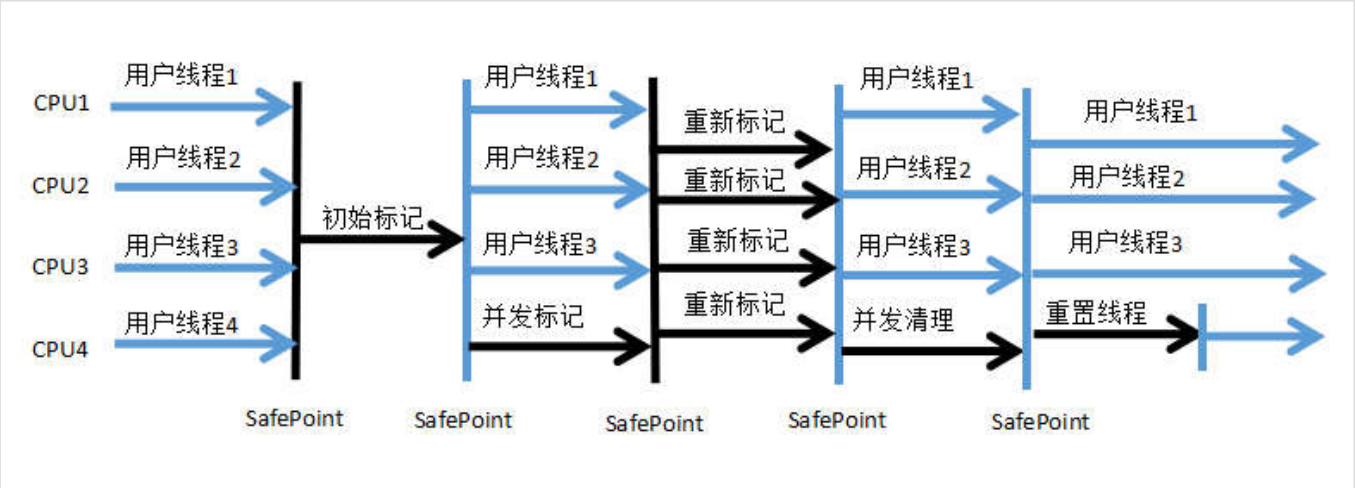
> 说明：此图参考《深入理解Java虚拟机：JVM高级特性与最佳实践》画出

CMS回收器

CMS(Concurrent Mark Sweep)回收器是一种以获取最短回收停顿时间为目标的回收器，也就是说，这是一个注重用户响应时间、注重用户体验的回收器。CMS回收器使用“标记清除”算法，其运行过程主要分成四个步骤：初始标记、并发标记、重新标记、并发清除。

- 初始标记（CMS initial mark）：标记一下GC Roots能直接关联到的对象，速度快，需要暂停用户线程（Stop The World）
- 并发标记（CMS concurrent mark）：根据GC Roots节点查找所有能与之直接或间接关联的对象（GC Roots Tracing），此过程耗时较长，可以与用户线程并发执行
- 重新标记（CMS remark）：修正并发标记期间因用户程序继续运行而导致标记产生变动的对象的标记记录，这个过程一般比初始标记阶段稍微长一点，但是远比并发标记的时间短，此过程也需要暂停用户线程（Stop The World）
- 并发清除（CMS concurrent sweep）：此过程将对象进行回收，可以与用户线程并发执行。

其运行过程图大致如下图所示



> 说明：此图参考《深入理解Java虚拟机：JVM高级特性与最佳实践》画出

CMS是一款优秀的回收器，但是其也存在以下几个缺点

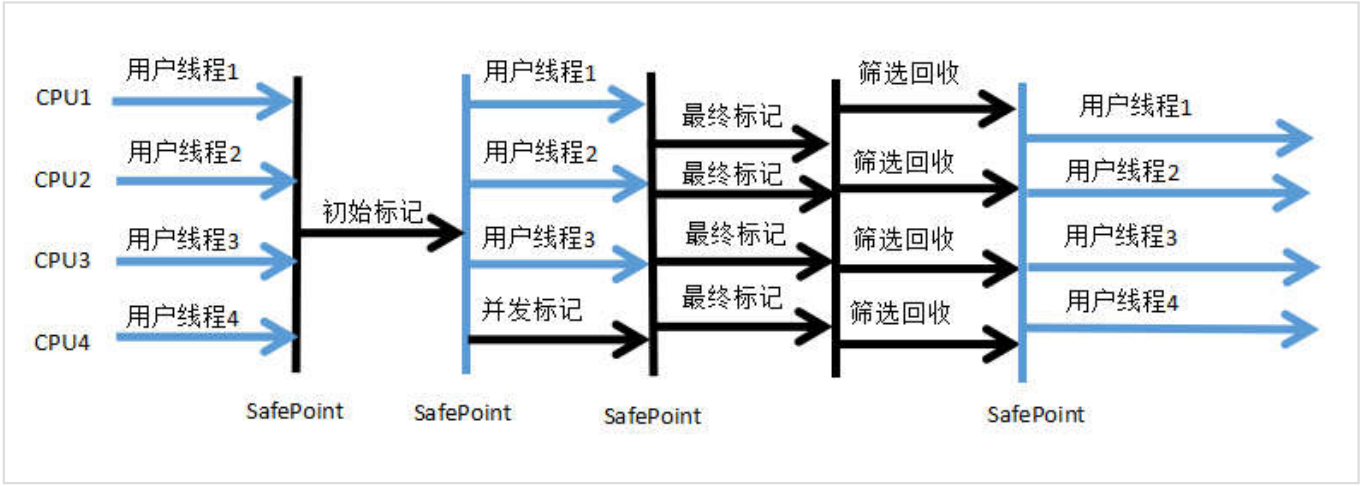
- 对CPU资源非常敏感。因为在并发标记和并发清除这两个阶段是运行用户线程并发执行的，线程之间的切换会导致了用户线程与回收线程争抢CPU资源的情况，如果CPU资源少，则会同时影响用户线程和垃圾回收的运行时间
- CMS回收器无法清理浮动垃圾（Floating Garbage），可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生，此时会临时启用Serial Old回收器重新进行老年代的垃圾回收
- 可能产生大量垃圾碎片，这是因为其使用“标记清除”的回收算法导致的

G1回收器

G1(Garbage-First)回收器是一款面向Server端的垃圾回收器，其不像其他垃圾回收器区分新生代和老年代，而是使用“化整为零”的思想把整个java堆划分成多个大小相等的独立区域（Region），虽然还保留着新生代和老年代的概念，但是新生代和老年代不再是物理隔离的，他们都是一部分Region(不需要连续)的集合。其运行过程主要分成四个步骤：初始标记、并发标记、最终标记、筛选回收。

- 初始标记（Initial Marking）：标记一下GC Roots能直接关联到的对象，并且修改TAMS(Next Top at Mark Start)的值，让下一阶段用户程序并发运行时能在正确可用的Region中创建对象，这阶段需要停顿用户线程（Stop The World），但是耗时短
- 并发标记（Concurrent Marking）：根据GC Roots节点查找所有能与之直接或间接关联的对象（GC Roots Tracing），此过程耗时较长，可以与用户线程并发执行
- 最终标记（Final Marking）：修正并发标记期间因用户程序继续运行而导致标记产生变动的对象的标记记录，此阶段耗时较少，也需要停顿用户线程（Stop The World）
- 筛选回收（Live Data Counting and Evacuation）：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停顿时间制定回收计划并回收对象

其运行过程图大致如下图所示



> 说明：此图参考《深入理解Java虚拟机：JVM高级特性与最佳实践》画出

G1回收器有以下特点

- 并发与并行。G1回收器能充分利用多CPU、多核环境下的硬件优势来缩短Stop-The-World停顿的时间。部分回收器原本需要停顿java线程执行GC动作，但是G1回收器仍然可以通过并发的方式让java程序继续运行
- 分代回收。虽然G1回收器中新生代和老年代的区别不在那么明显，但是G1回收器仍然能够采用不同的方式处理新建的对象和已经存活一段时间的对象
- 空间整合：G1回收器整体上是使用“标记整理”算法，而不是容易产生空间碎片的“标记清除”算法
- 可预测的停顿：G1回收器能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为M毫秒的时间片段内，消耗在垃圾回收上的时间不超过N毫秒，这几乎是实时java(RTSJ)的垃圾回收器的特征。

比较

回收器	适用内存区域	适用B/S范围	回收算法
Serial回收器	新生代	Client	复制
ParNew回收器	新生代	Server	复制
Parallel Scavenge回收器	新生代	Server	复制
Serial Old	老年代	Client、Server	标记整理
Parallel Old	老年代	Server	标记整理
CMS回收器	老年代	Server	标记清除
G1回收器	新生代、老年代	Server	标记整理

参考资料

[1] 周志明，深入理解Java虚拟机：JVM高级特性与最佳实践[M]，北京：机械工业出版社，2013

java # jvm

jvm 垃圾回收算法

jvm 内存分配策略