http://www.jianshu.com/p/50d5c88b272d

JVM中标记垃圾使用的算法是一种根搜索算法。就是从一个叫GC Roots的对象开始，向下搜索，如果一个对象不能达到GC Roots对象的时候，说明它可以被回收了。

JVM中对于被标记为垃圾的对象进行回收时又分为了一下3种算法：

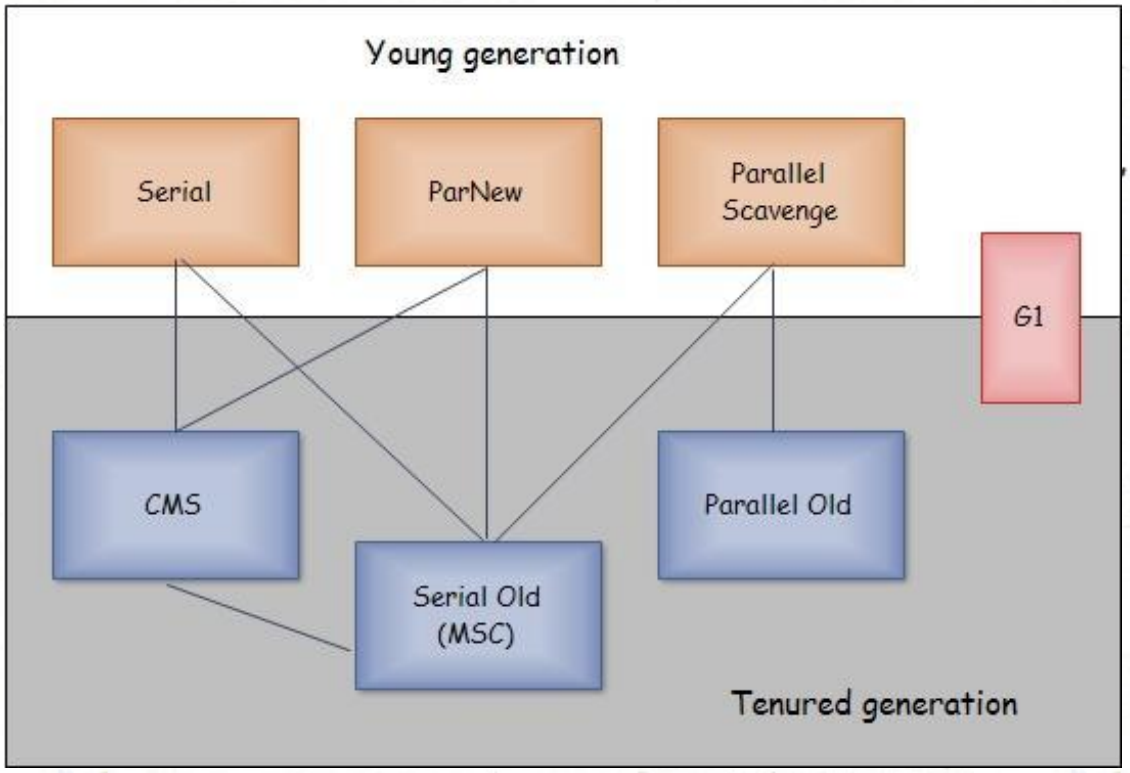
1.标记清除算法，该算法是从根集合扫描整个空间，标记存活的对象，然后在扫描整个空间对没有被标记的对象进行回收，这种算法在存活对象较多时比较高效，但会产生内存碎片。

2.复制算法，该算法是从根集合扫描，并将存活的对象复制到新的空间，这种算法在存活对象少时比较高效。

3.标记整理算法，标记整理算法和标记清除算法一样都会扫描并标记存活对象，在回收未标记对象的同时会整理被标记的对象，解决了内存碎片的问题。

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。

Java虚拟机规范中对垃圾收集器应该如何实现并没有任何规定，因此不同的厂商、不同版本的虚拟机所提供的垃圾收集器都可能会有很大差别，并且一般都会提供参数供用户根据自己的应用特点和要求组合出各个年代所使用的收集器。



HotSpot虚拟机的垃圾回收器

图中展示了7种作用于不同分代的收集器，如果两个收集器之间存在连线，就说明它们可以搭配使用。虚拟机所处的区域，则表示它是属于新生代收集器还是老年代收集器。

概念理解

1. 并发和并行

这两个名词都是并发编程中的概念，在谈论垃圾收集器的上下文语境中，它们可以解释如下。

- 并行（**Parallel**）：指多条垃圾收集线程并行工作，但此时用户线程仍然处于等待状态。
- 并发（**Concurrent**）：指用户线程与垃圾收集线程同时执行（但不一定是并行的，可能会交替执行），用户程序在继续运行，而垃圾收集程序运行于另一个CPU上。

2. Minor GC 和 Full GC

- 新生代**GC**（**Minor GC**）：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，

一般回收速度也比较快。

- 老年代GC（**Major GC / Full GC**）：指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。

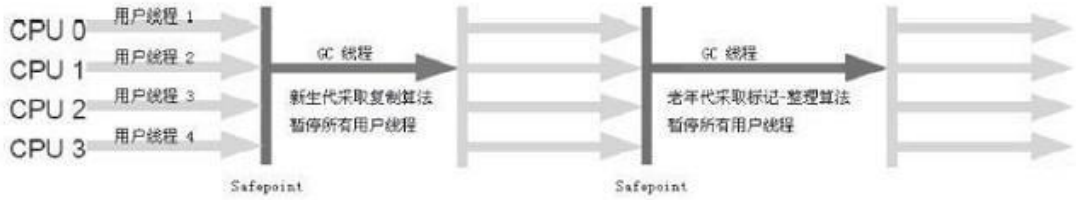
3. 吞吐量

吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量 = 运行用户代码时间 /（运行用户代码时间 + 垃圾收集时间）。

虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

一、Serial收集器(复制算法)

Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK 1.3.1之前）是虚拟机新生代收集的唯一选择。



Serial / Serial Old 收集器运行示意图

1. 特性：

这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。`Stop The World`

2. 应用场景：

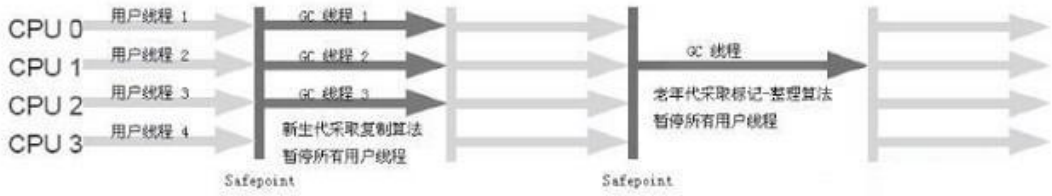
Serial收集器是虚拟机运行在Client模式下的默认新生代收集器。

3. 优势：

简单而高效（与其他收集器的单线程比），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可以获得

最高的单线程收集效率。

二、ParNew收集器(复制算法)



ParNew / Serial Old 收集器运行示意图

1. 特性：

ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。

2. 应用场景：

ParNew收集器是许多运行在Server模式下的虚拟机中首选的新生代收集器。

很重要的原因是：除了Serial收集器外，目前只有它能与CMS收集器配合工作。

在JDK 1.5时期，HotSpot推出了一款在强交互应用中几乎可认为有划时代意义的垃圾收集器——CMS收集器，这款收集器是HotSpot虚拟机中第一款真正意义上的并发收集器，它第一次实现了让垃圾收集线程与用户线程同时工作。

不幸的是，CMS作为老年代的收集器，却无法与JDK 1.4.0中已经存在的新生代收集器Parallel Scavenge配合工作，所以在JDK 1.5中使用CMS来收集老年代的时候，新生代只能选择ParNew或者Serial收集器中的一个。

3. Serial收集器 VS ParNew收集器：

ParNew收集器在单CPU的环境中绝对不会有比Serial收集器更好的效果，甚至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Serial收集器。

然而，随着可以使用的CPU的数量的增加，它对于GC时系统资源的有效利用还是很有好处的。

三、Parallel Scavenge收集器(复制算法)

1. 特性：

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。

2. 应用场景：

停顿时间越短就越适合需要与用户交互的程序，良好的响应速度能提升用户体验，而高吞吐量则可以高效率地利用CPU时间，尽快完成程序的运算任务，主要适合在后台运算而不需要太多交互的任务。

3. 对比分析：

- **Parallel Scavenge收集器 VS CMS等收集器**：

Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，CMS等收集器的关注点是尽可能地缩短垃圾收集时用户线程的停顿时间，而Parallel Scavenge收集器的目标则是达到一个可控制的吞吐量（Throughput）。

由于与吞吐量关系密切，Parallel Scavenge收集器也经常称为“吞吐量优先”收集器。

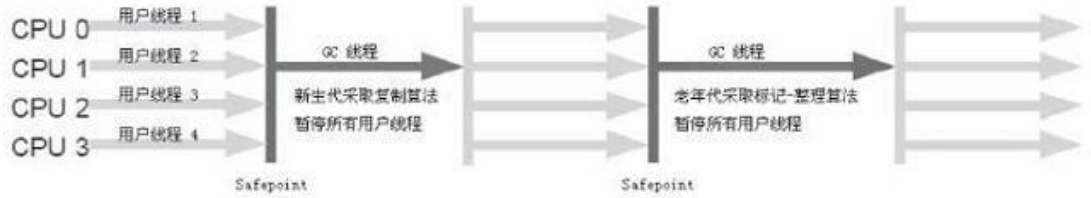
- **Parallel Scavenge收集器 VS ParNew收集器**：

Parallel Scavenge收集器与ParNew收集器的一个重要区别是它具有自适应调节策略。

GC自适应的调节策略：

Parallel Scavenge收集器有一个参数`-XX:+UseAdaptiveSizePolicy`。当这个参数打开之后，就不需要手工指定新生代的大小、Eden与Survivor区的比例、晋升老年代对象年龄等细节参数了，虚拟机会根据当前系统的运行情况收集性能监控信息，动态调整这些参数以提供最合适的停顿时间或者最大的吞吐量，这种调节方式称为GC自适应的调节策略（GC Ergonomics）。

四、Serial Old收集器(标记整理算法)



Serial / Serial Old 收集器运行示意图

1. 特性：

Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用标记－整理算法。

2. 应用场景：

- **Client模式**

Serial Old收集器的主要意义也是在于给Client模式下的虚拟机使用。

- **Server模式**

如果在Server模式下，那么它主要还有两大用途：一种用途是在JDK 1.5以及之前的版本中与Parallel Scavenge收集器搭配使用，另一种用途就是作为CMS收集器的后备预案，在并发收集发生Concurrent Mode Failure时使用。

五、Parallel Old收集器(标记整理算法)

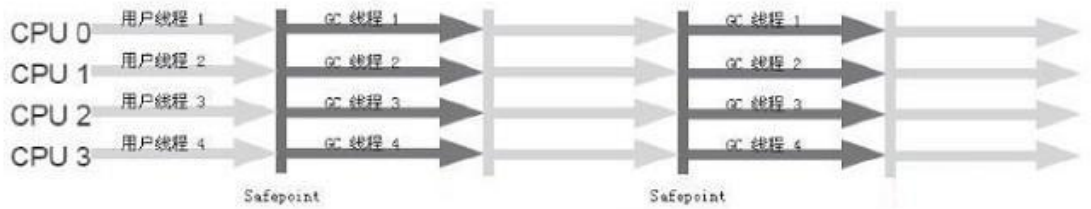


图 3-9　Parallel Scavenge / Parallel Old 收集器运行示意图

1. 特性：

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记－整理”算法。

2. 应用场景：

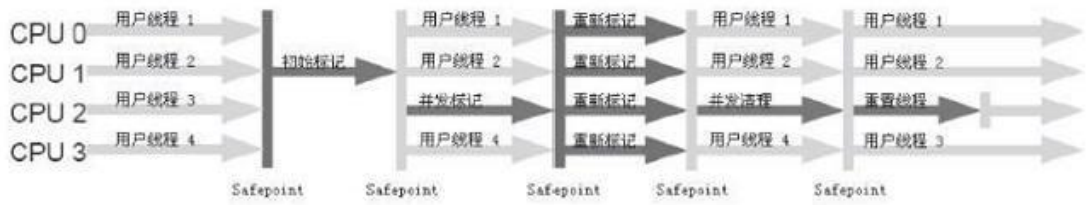
在注重吞吐量以及CPU资源敏感的场合，都可以优先考虑Parallel Scavenge加Parallel Old收集器。

这个收集器是在JDK 1.6中才开始提供的，在此之前，新生代的Parallel Scavenge收集器一直处于比较尴尬的状态。原因是，如果新生代选择了Parallel Scavenge收集器，老年代除了Serial Old收集器外别无选择（Parallel Scavenge收集器无法与CMS收集器配合工作）。由于老年代Serial Old收集器在服务端应用性能上的“拖累”，使用了Parallel Scavenge收集器也未必能在整体应用上获得吞吐量最大化的效果，由于单线程的老年代收集中无法充分利用服务器多CPU的处理能力，在老年代很大而且硬件比较高级的环境中，这种组合的吞吐量甚至还不一定有ParNew加CMS的组合“给力”。直到Parallel Old收集器出现后，“吞吐量优先”收集器终于有了比较名副其实的应用组合。

六、CMS收集器(标记清除算法)

1. 特性：

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。目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。CMS收集器就非常符合这类应用的需求。



Concurrent Mark Sweep 收集器运行示意图

CMS收集器是基于“标记—清除”算法实现的，它的运作过程相对于前面几种收集器来说更复杂一些，整个过程分为4个步骤：

- **初始标记（CMS initial mark）**

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“Stop The World”。

- **并发标记（CMS concurrent mark）**

并发标记阶段就是进行GC Roots Tracing的过程。

- **重新标记（CMS remark）**

重新标记阶段是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，仍然需要“Stop The World”。

- **并发清除（CMS concurrent sweep）**

并发清除阶段会清除对象。

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。

2. 优点：

CMS是一款优秀的收集器，它的主要优点在名字上已经体现出来了：并发收集、低停顿。

3. 缺点：

- **CMS收集器对CPU资源非常敏感**

其实，面向并发设计的程序都对CPU资源比较敏感。在并发阶段，它虽然不会导致用户线程停顿，但是会因为占用了一部分线程（或者说CPU资源）而导致应用程序变慢，总吞吐量会降低。

CMS默认启动的回收线程数是（CPU数量+3）/ 4，也就是当CPU在4个以上时，并发回收时垃圾收集线程不少于25%的CPU资源，并且随着CPU数量的增加而下降。但是当CPU不足4个（譬如2个）时，CMS对用户程序的影响就可能变得很大。

- **CMS收集器无法处理浮动垃圾**

CMS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CMS并发清理阶段用户线程还在运行着，伴随程序运行自然就还会有

新的垃圾不断产生，这一部分垃圾出现在标记过程之后，CMS无法在当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就称为“浮动垃圾”。

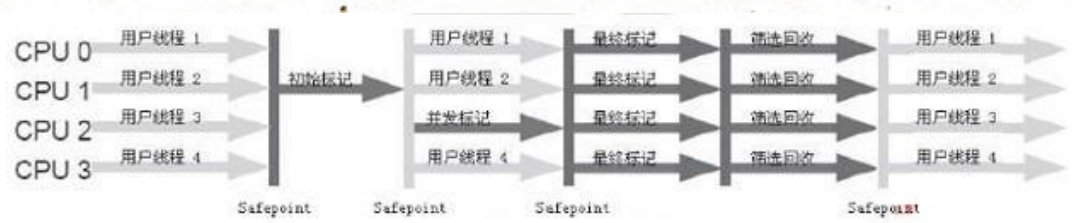
也是由于在垃圾收集阶段用户线程还需要运行，那也就还需要预留有足够的内存空间给用户线程使用，因此CMS收集器不能像其他收集器那样等到老年代几乎完全被填满了再进行收集，需要预留一部分空间提供并发收集时的程序运作使用。要是CMS运行期间预留的内存无法满足程序需要，就会出现一次“Concurrent Mode Failure”失败，这时虚拟机将启动后备预案：临时启用Serial Old收集器来重新进行老年代的垃圾收集，这样停顿时间就很长了。

- **CMS收集器会产生大量空间碎片**

CMS是一款基于“标记—清除”算法实现的收集器，这意味着收集结束时会有大量空间碎片产生。

空间碎片过多时，将会给大对象分配带来很大麻烦，往往会出现老年代还有很大空间剩余，但是无法找到足够大的连续空间来分配当前对象，不得不提前触发一次Full GC。

七、G1收集器(复制算法和标记整理算法)



G1 收集器运行示意图

1. 特性：

G1（Garbage-First）是一款面向服务端应用的垃圾收集器。HotSpot开发团队赋予它的使命是未来可以替换掉JDK 1.5中发布的CMS收集器。与其他GC收集器相比，G1具备如下特点。

- **并行与并发**

G1能充分利用多CPU、多核环境下的硬件优势，使用多个CPU来缩短Stop-The-World停顿的时间，部分其他收集器原本需要停顿Java线程执行的GC动作，G1收集器仍然可以通过并发的方式让Java程序继续执行。

- 分代收集

与其他收集器一样，分代概念在G1中依然得以保留。虽然G1可以不需要其他收集器配合就能独立管理整个GC堆，但它能够采用不同的方式去处理新创建的对象和已经存活了一段时间、熬过多次GC的旧对象以获取更好的收集效果。

- 空间整合

与CMS的“标记－清理”算法不同，G1从整体来看是基于“标记－整理”算法实现的收集器，从局部（两个Region之间）上来看是基于“复制”算法实现的，但无论如何，这两种算法都意味着G1运作期间不会产生内存空间碎片，收集后能提供规整的可用内存。这种特性有利于程序长时间运行，分配大对象时不会因为无法找到连续内存空间而提前触发下一次GC。

- 可预测的停顿

这是G1相对于CMS的另一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了追求低停顿外，还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，消耗在垃圾收集上的时间不得超过N毫秒。

在G1之前的其他收集器进行收集的范围都是整个新生代或者老年代，而G1不再是这样。使用G1收集器时，Java堆的内存布局就与其他收集器有很大差别，它将整个Java堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔离的了，它们都是一部分Region（不需要连续）的集合。

G1收集器之所以能建立可预测的停顿时间模型，是因为它可以有计划地避免在整个Java堆中进行全区域的垃圾收集。G1跟踪各个Region里面的垃圾堆积的价值大小（回收所获得的空间大小以及回收所需时间的经验值），在后台维护一个优先列表，每次根据允许的收集时间，优先回收价值最大

的Region（这也就是Garbage-First名称的来由）。这种使用Region划分内存空间以及有优先级的区域回收方式，保证了G1收集器在有限的时间内可以获取尽可能高的收集效率。

2. 执行过程：

G1收集器的运作大致可划分为以下几个步骤：

- **初始标记（Initial Marking）**

初始标记阶段仅仅只是标记一下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中创建新对象，这阶段需要停顿线程，但耗时很短。

- **并发标记（Concurrent Marking）**

并发标记阶段是从GC Root开始对堆中对象进行可达性分析，找出存活的对象，这阶段耗时较长，但可与用户程序并发执行。

- **最终标记（Final Marking）**

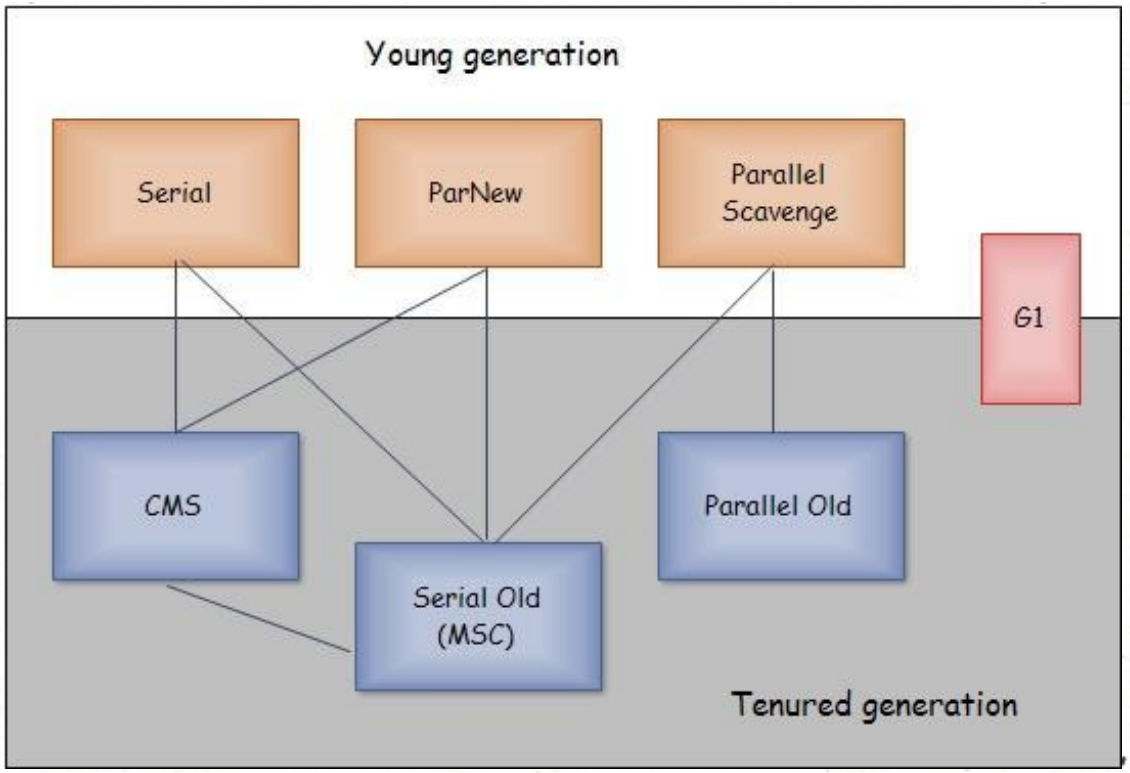
最终标记阶段是为了修正在并发标记期间因用户程序继续运作而导致标记产生变动的那一部分标记记录，虚拟机将这段时间对象变化记录在线程`Remembered Set Logs`里面，最终标记阶段需要把`Remembered Set Logs`的数据合并到`Remembered Set`中，这阶段需要停顿线程，但是可并行执行。

- **筛选回收（Live Data Counting and Evacuation）**

筛选回收阶段首先对各个Region的回收价值和成本进行排序，根据用户所期望的GC停顿时间来制定回收计划，这个阶段其实也可以做到与用户程序一起并发执行，但是因为只回收一部分Region，时间是用户可控制的，而且停顿用户线程将大幅提高收集效率。

八、总结

虽然我们是在对各个收集器进行比较，但并非为了挑选出一个最好的收集器。因为直到现在为止还没有最好的收集器出现，更加没有万能的收集器，所以我们选择的只是对具体应用最合适的收集器。这点不需要多加解释就能证明：如果有一种放之四海皆准、任何场景下都适用的完美收集器存在，那HotSpot虚拟机就没必要实现那么多不同的收集器了。



HotSpot虚拟机的垃圾回收器

推荐阅读：《深入理解Java虚拟机：JVM高级特性与最佳实践》周志明著