

上图展示了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。

**相关概念**

**并行和并发**

* **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在另一个CPU上。

**吞吐量（Throughput）**

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）。**

假设虚拟机总共运行了100分钟，其中垃圾收集花掉1分钟，那吞吐量就是99%。

**Minor GC 和 Full GC**

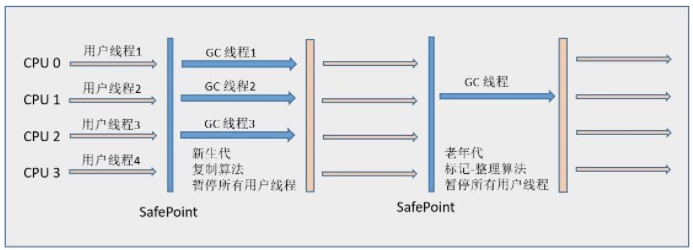
* **新生代GC（Minor GC）**：指发生在新生代的垃圾收集动作，因为Java对象大多都具备朝生夕灭的特性，所以Minor GC非常频繁，一般回收速度也比较快。具体原理见上一篇文章。
* **老年代GC（Major GC / Full GC）**：指发生在老年代的GC，出现了Major GC，经常会伴随至少一次的Minor GC（但非绝对的，在Parallel Scavenge收集器的收集策略里就有直接进行Major GC的策略选择过程）。Major GC的速度一般会比Minor GC慢10倍以上。

# 新生代收集器

#### Serial收集器（串行，复制算法）

**Serial（串行）**收集器是最基本、发展历史最悠久的收集器，它是采用**复制算法**的**新生代收集器**，曾经（JDK 1.3.1之前）是虚拟机**新生代**收集的唯一选择。它是一个单线程收集器，只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是**它在进行垃圾收集时，必须暂停其他所有的工作线程，直至Serial收集器收集结束为止（“Stop The World”）**。这项工作是由虚拟机在后台自动发起和自动完成的，在用户不可见的情况下把用户正常工作的线程全部停掉，这对很多应用来说是难以接收的。

下图展示了Serial 收集器（老年代采用Serial Old收集器）的运行过程：



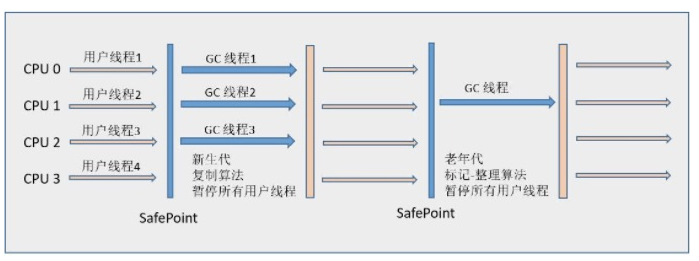
为了消除或减少工作线程因内存回收而导致的停顿，HotSpot虚拟机开发团队在JDK 1.3之后的Java发展历程中研发出了各种其他的优秀收集器，这些将在稍后介绍。但是这些收集器的诞生并不意味着Serial收集器已经“老而无用”，实际上到现在为止，它依然是H**otSpot虚拟机运行在Client模式下的默认的新生代收集器**。它也有着优于其他收集器的地方：**简单而高效（与其他收集器的单线程相比），对于限定单个CPU的环境来说，Serial收集器由于没有线程交互的开销，专心做垃圾收集自然可以获得更高的单线程收集效率。**

在用户的桌面应用场景中，分配给虚拟机管理的内存一般不会很大，收集几十兆甚至一两百兆的新生代（仅仅是新生代使用的内存，桌面应用基本不会再大了），停顿时间完全可以控制在几十毫秒最多一百毫秒以内，只要不频繁发生，这点停顿时间可以接收。所以，Serial收集器对于运行在Client模式下的虚拟机来说是一个很好的选择

#### ParNew 收集器（并行，复制算法）

**ParNew**收集器就是Serial收集器的多线程版本，它也是一个**新生代收集器**。除了使用多线程进行垃圾收集外，其余行为包括Serial收集器可用的所有控制参数、收集算法（复制算法）、Stop The World、对象分配规则、回收策略等与Serial收集器完全相同，两者共用了相当多的代码。

ParNew收集器的工作过程如下图（老年代采用Serial Old收集器）：



ParNew收集器除了使用多线程收集外，其他与Serial收集器相比并无太多创新之处，但它却是许多运行在Server模式下的虚拟机中首选的新生代收集器，其中有一个与性能无关的重要原因是，**除了Serial收集器外，目前只有它能和CMS收集器（Concurrent Mark Sweep）配合工作**，CMS收集器是JDK 1.5推出的一个具有划时代意义的收集器，具体内容将在稍后进行介绍。

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，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。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很有好处的。它默认开启的收集线程数与CPU的数量相同，在CPU非常多的情况下可使用**-XX:ParallerGCThreads**参数设置。

#### Parallel Scavenge 收集器（并行，复制算法）

**Parallel Scavenge**收集器也是一个**并行**的**多线程新生代**收集器，它也使用**复制算法**。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收集器的目标是**达到一个可控制的吞吐量（Throughput）**。

**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**。

Parallel Scavenge收集器除了会显而易见地提供可以精确控制吞吐量的参数，还提供了一个参数**-XX:+UseAdaptiveSizePolicy**，这是一个开关参数，打开参数后，就不需要手工指定新生代的大小（-Xmn）、Eden和Survivor区的比例（-XX:SurvivorRatio）、晋升老年代对象年龄（-XX:PretenureSizeThreshold）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，这种方式称为**GC自适应的调节策略（GC Ergonomics）**。自适应调节策略也是Parallel Scavenge收集器与ParNew收集器的一个重要区别。

另外值得注意的一点是，Parallel Scavenge收集器无法与CMS收集器配合使用，所以在JDK 1.6推出Parallel Old之前，如果新生代选择Parallel Scavenge收集器，老年代只有Serial Old收集器能与之配合使用。

**老年代收集器**

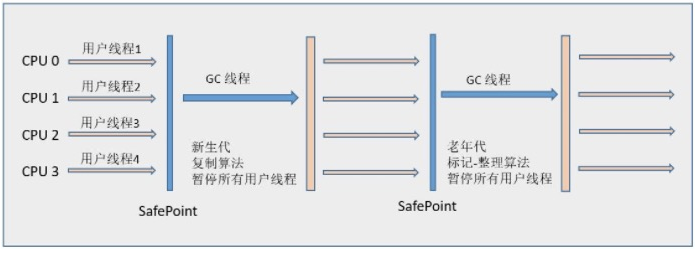
**Serial Old收集器(串行，标记-整理)**

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”（Mark-Compact）**算法。

此收集器的主要意义也是在于给Client模式下的虚拟机使用。如果在Server模式下，它还有两大用途：

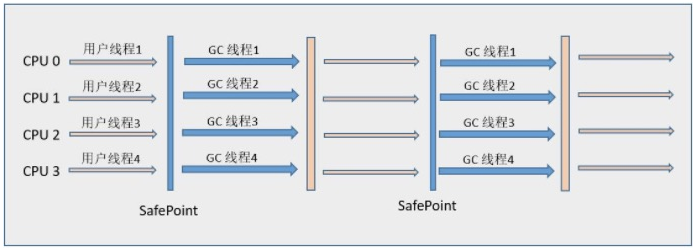
* 在JDK1.5 以及之前版本（Parallel Old诞生以前）中与Parallel Scavenge收集器搭配使用。
* 作为CMS收集器的后备预案，在并发收集发生**Concurrent Mode Failure**时使用。

它的工作流程与Serial收集器相同，这里再次给出Serial/Serial Old配合使用的工作流程图：



#### Parallel Old收集器（并行，标记-整理）

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。前面已经提到过，这个收集器是在JDK 1.6中才开始提供的，在此之前，如果新生代选择了Parallel Scavenge收集器，老年代除了Serial Old以外别无选择，所以在Parallel Old诞生以后，**“吞吐量优先”收集器**终于有了比较名副其实的应用组合，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。Parallel Old收集器的工作流程与Parallel Scavenge相同，这里给出Parallel Scavenge/Parallel Old收集器配合使用的流程图：



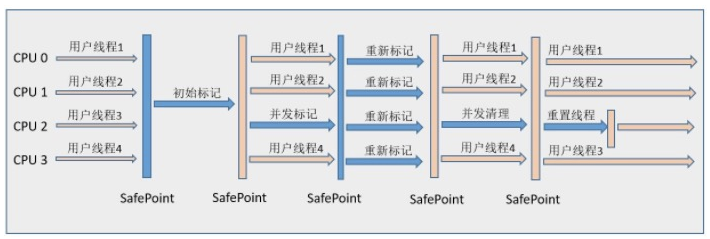
**CMS收集器(基本并发，标记-清除)**

**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，它非常符合那些集中在互联网站或者B/S系统的服务端上的Java应用，这些应用都非常重视服务的响应速度。从名字上（“Mark Sweep”）就可以看出它是基于**“标记-清除”**算法实现的。

CMS收集器工作的整个流程分为以下4个步骤：

* **初始标记（CMS initial mark）**：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）**：进行**GC Roots Tracing**的过程，在整个过程中耗时最长。
* **重新标记（CMS remark）**：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“Stop The World”。
* **并发清除（CMS concurrent sweep）**

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。通过下图可以比较清楚地看到CMS收集器的运作步骤中并发和需要停顿的时间：



**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**，因此CMS收集器也被称为**并发低停顿收集器（Concurrent Low Pause Collector）**。

**缺点**

* **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个时（比如2个），CMS对用户程序的影响就可能变得很大**，如果本来CPU负载就比较大，还要分出一半的运算能力去执行收集器线程，就可能导致用户程序的执行速度忽然降低了50%，其实也让人无法接受。
* **无法处理浮动垃圾（Floating Garbage）** 可能出现“Concurrent Mode Failure”失败而导致另一次Full GC的产生。**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无法再当次收集中处理掉它们，只好留待下一次GC时再清理掉。这一部分垃圾就被称为**“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。
* **标记-清除算法导致的空间碎片** 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，往往出现老年代空间剩余，但无法找到足够大连续空间来分配当前对象。

**G1收集器**

**G1（Garbage-First）**收集器是当今收集器技术发展最前沿的成果之一，它是一款**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（在比较长期的）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，但它能够采用不同方式去处理新创建的对象和已存活一段时间、熬过多次GC的旧对象来获取更好的收集效果。
* **空间整合** 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。这意味着G1运行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。此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CMS的一大优势，降低停顿时间是G1和CMS共同的关注点，但G1除了降低停顿外，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GC上的时间不得超过N毫秒，这几乎已经是实时Java（RTSJ）的垃圾收集器的特征了。

**横跨整个堆内存**

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生代，而G1不再是这样。G1在使用时，Java堆的内存布局与其他收集器有很大区别，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，而都是一部分Region（不需要连续）的集合**。

**建立可预测的时间模型**

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。

**避免全堆扫描——Remembered Set**

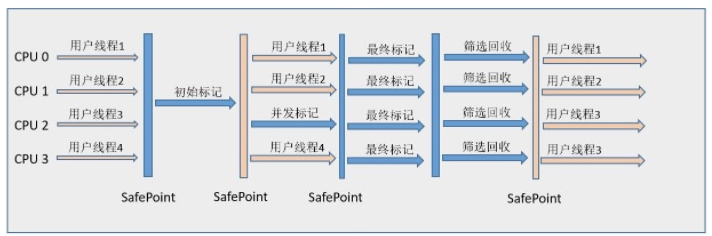
G1把Java堆分为多个Region，就是“化整为零”。但是Region不可能是孤立的，一个对象分配在某个Region中，可以与整个Java堆任意的对象发生引用关系。在做可达性分析确定对象是否存活的时候，需要扫描整个Java堆才能保证准确性，这显然是对GC效率的极大伤害。

为了避免全堆扫描的发生，虚拟机**为G1中每个Region维护了一个与之对应的Remembered Set**。虚拟机发现程序在对Reference类型的数据进行写操作时，会产生一个Write Barrier暂时中断写操作，检查Reference引用的对象是否处于不同的Region之中（在分代的例子中就是检查是否老年代中的对象引用了新生代中的对象），如果是，便通过CardTable**把相关引用信息记录到被引用对象所属的Region的Remembered Set之中**。当进行内存回收时，在GC根节点的枚举范围中加入Remembered Set即可保证不对全堆扫描也不会有遗漏。

如果不计算维护Remembered Set的操作，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）** 仅仅只是标记一下GC Roots 能直接关联到的对象，并且修改**TAMS（Nest Top Mark Start）**的值，让下一阶段用户程序并发运行时，能在正确可以的Region中创建对象，此阶段需要**停顿线程**，但耗时很短。
* **并发标记（Concurrent Marking）** 从GC Root 开始对堆中对象进行**可达性分析**，找到存活对象，此阶段耗时较长，但**可与用户程序并发执行**。
* **最终标记（Final Marking）** 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**线程的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）** 首先对各个Region中的回收价值和成本进行排序，根据用户所期望的GC 停顿是时间来制定回收计划。此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度提高收集效率。

通过下图可以比较清楚地看到G1收集器的运作步骤中并发和需要停顿的阶段（Safepoint处）：



# 总结

