垃圾收集器与内存分配策略

概述

垃圾收集（GC）关注的是JAVA堆中内存的自动回收问题：

1. 哪些内存需要回收
2. 什么时候回收
3. 如何回收 内存分配时指如何决定新建对象生成的位置，特别是当同时进行GC时。

哪些内存需要回收?

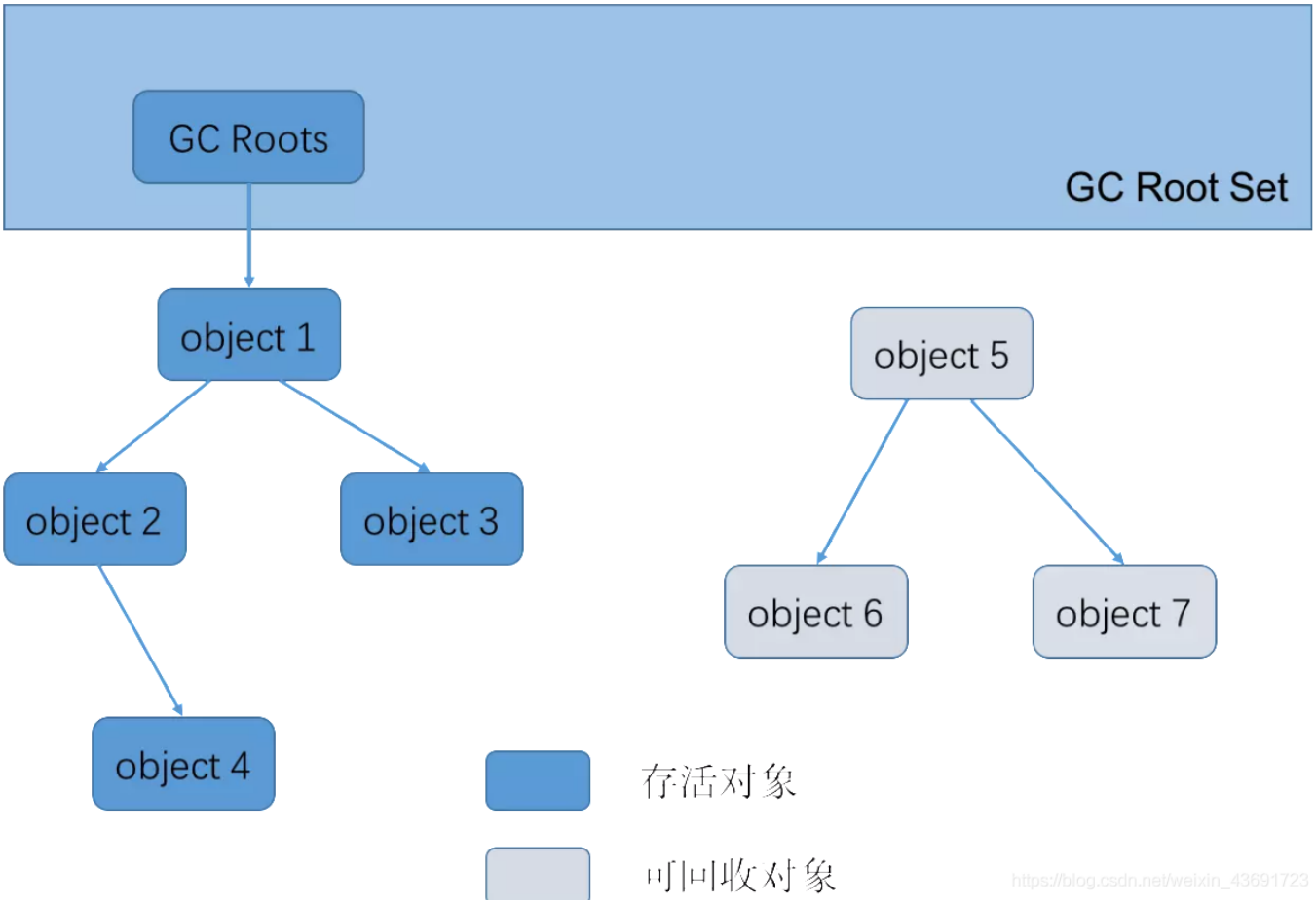
垃圾收集器回收的时候第一件事情就是确认哪些对象还存活着，哪些已经死去（即对象不会再被使用）。

引用计数法

类似C++智能指针的实现方式，为每个对象维护一个计数器，当对象被引用的时候，计数器加一，计数器为0的时候代表对象已死。优势在于实现简单，劣势在于难以解决循环引用等问题。

可达性分析算法

通过一系列的 'GC Roots' 的对象作为起始点，从这些节点出发所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连的时候说明对象不可用。



可作为GC Roots的对象：

1. 虚拟机栈 (栈帧中的本地变量表) 中引用的对象

2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 本地方法栈中 JNI(即一般说的 Native 方法) 引用的对象
5. 等等

JAVA中的引用

不同的对象引用类型， GC 会采用不同的方法进行回收，JVM 对象的引用分为了四种类型：

1. 强引用：类似于 Object obj = new Object(); 创建的，只要强引用在就不回收。
2. 软引用：SoftReference 类实现软引用。在系统要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行二次回收。
3. 弱引用：WeakReference 类实现弱引用。对象只能生存到下一次垃圾收集之前。在垃圾收集器工作时，无论内存是否足够都会回收掉只被弱引用关联的对象。
4. 虚引用：PhantomReference 类实现虚引用。无法通过虚引用获取一个对象的实例，为一个对象设置虚引用关联的唯一目的就是能在这个对象被收集器回收时收到一个系统通知。 无论引用计数算法还是可达性分析算法都是基于强引用而言的。

finalize（）相关

回收方法区

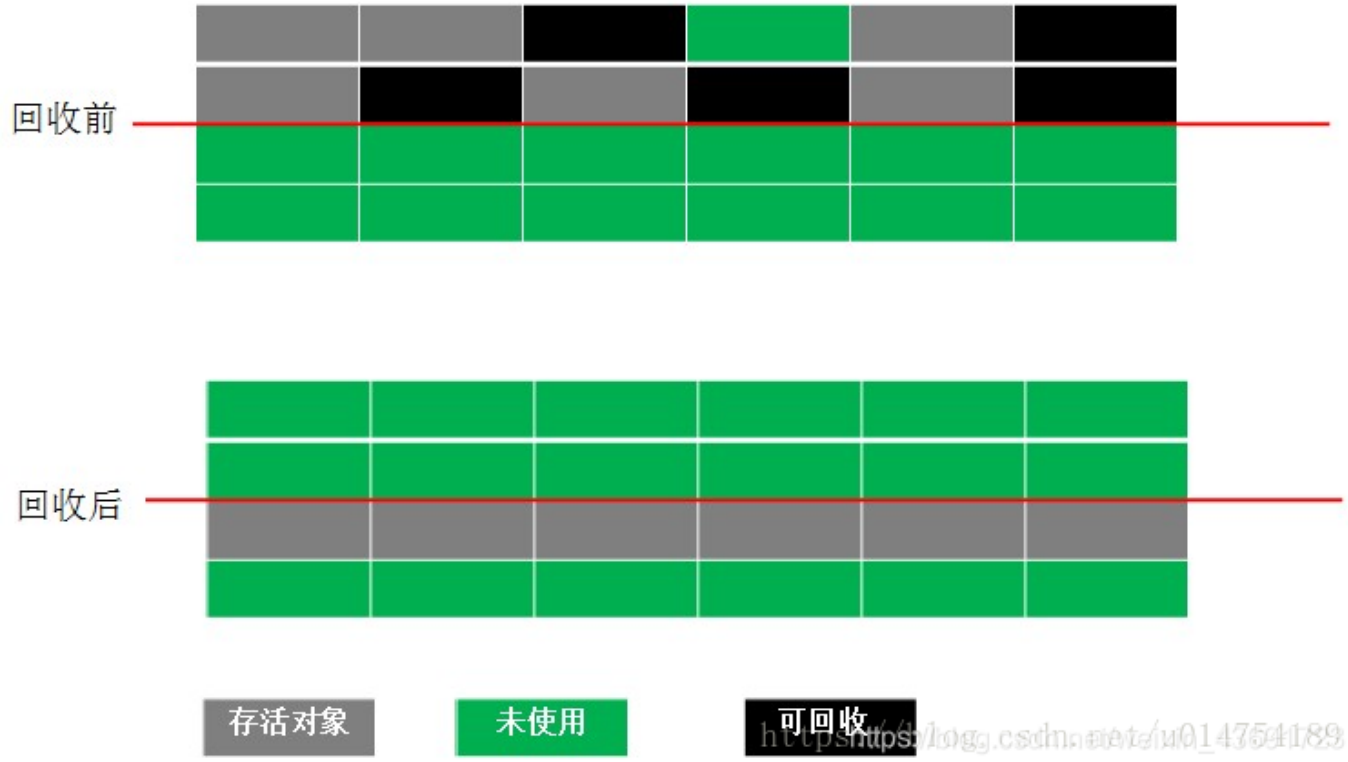
垃圾回收算法

分代收集理论

弱分代假说：大部分对象朝生夕死，保存在新生代，频繁进行GC 强分代假说：熬过越多次垃圾收集过程中的对象就越难以消亡，熬过一定次数GC的对象放入老年代

标记-清除算法

标记—清除算法是最基础的收集算法，它分为 “标记” 和“清除”两个阶段：首先标记出所需回收的对象，在标记完成后统一回收掉所有被标记的对象，它的标记过程其实就是前面的可达性分析算法中判定垃圾对象的标记过程。标记—清除算法的执行情况如下图所示：

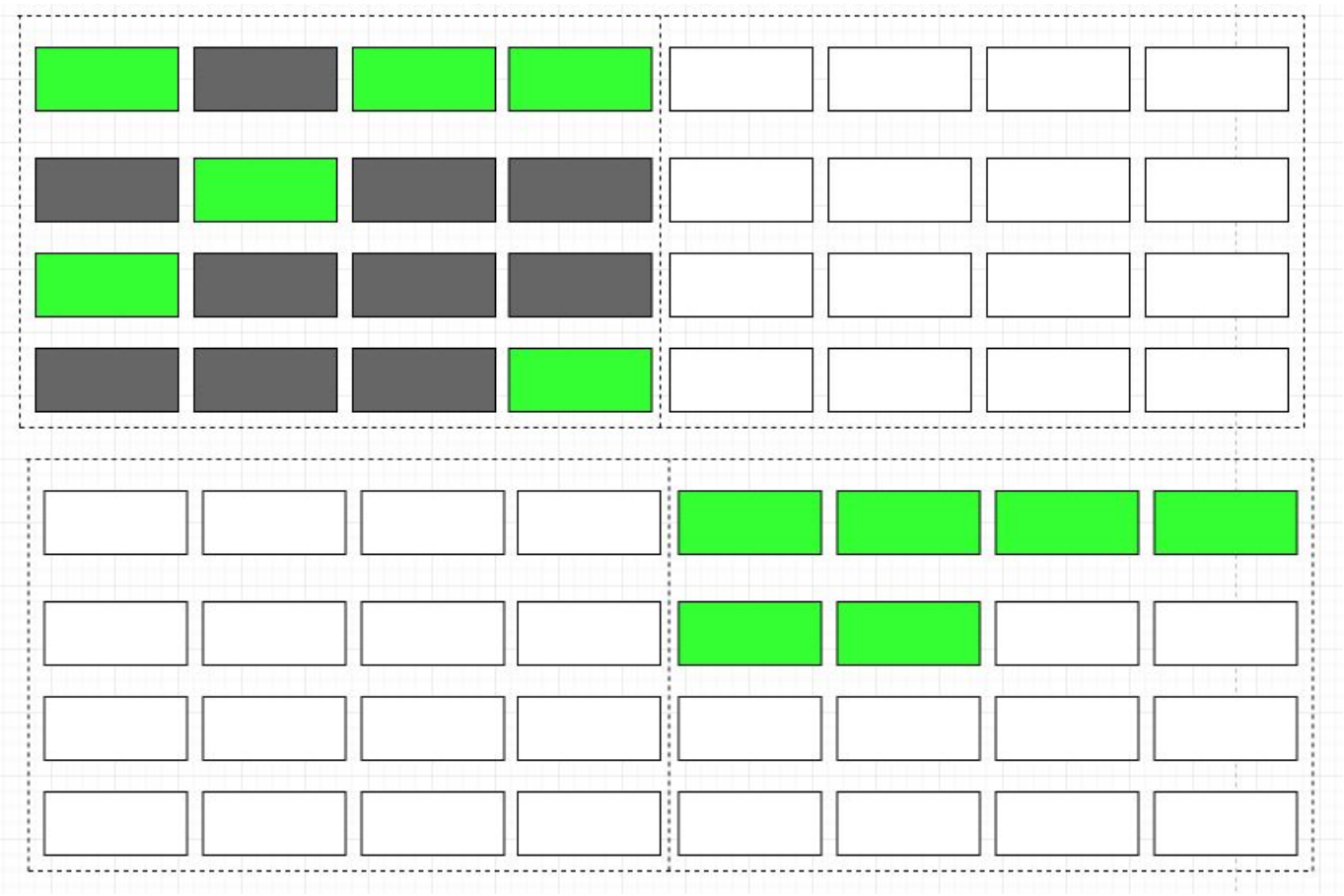


主要缺点：

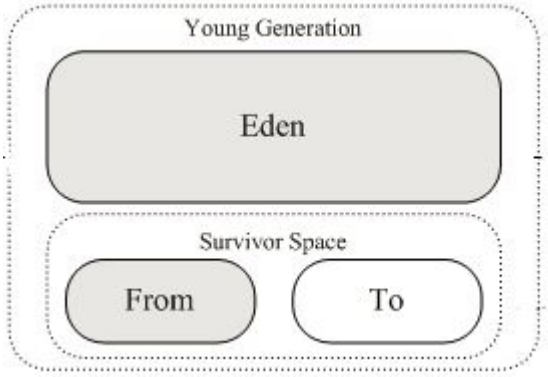
1. 一个是效率问题，标记和清除过程的效率都不高。
2. 内存碎片

标记-复制算法

为了解决 Mark-Sweep 算法的缺陷，Copying 算法就被提了出来。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。



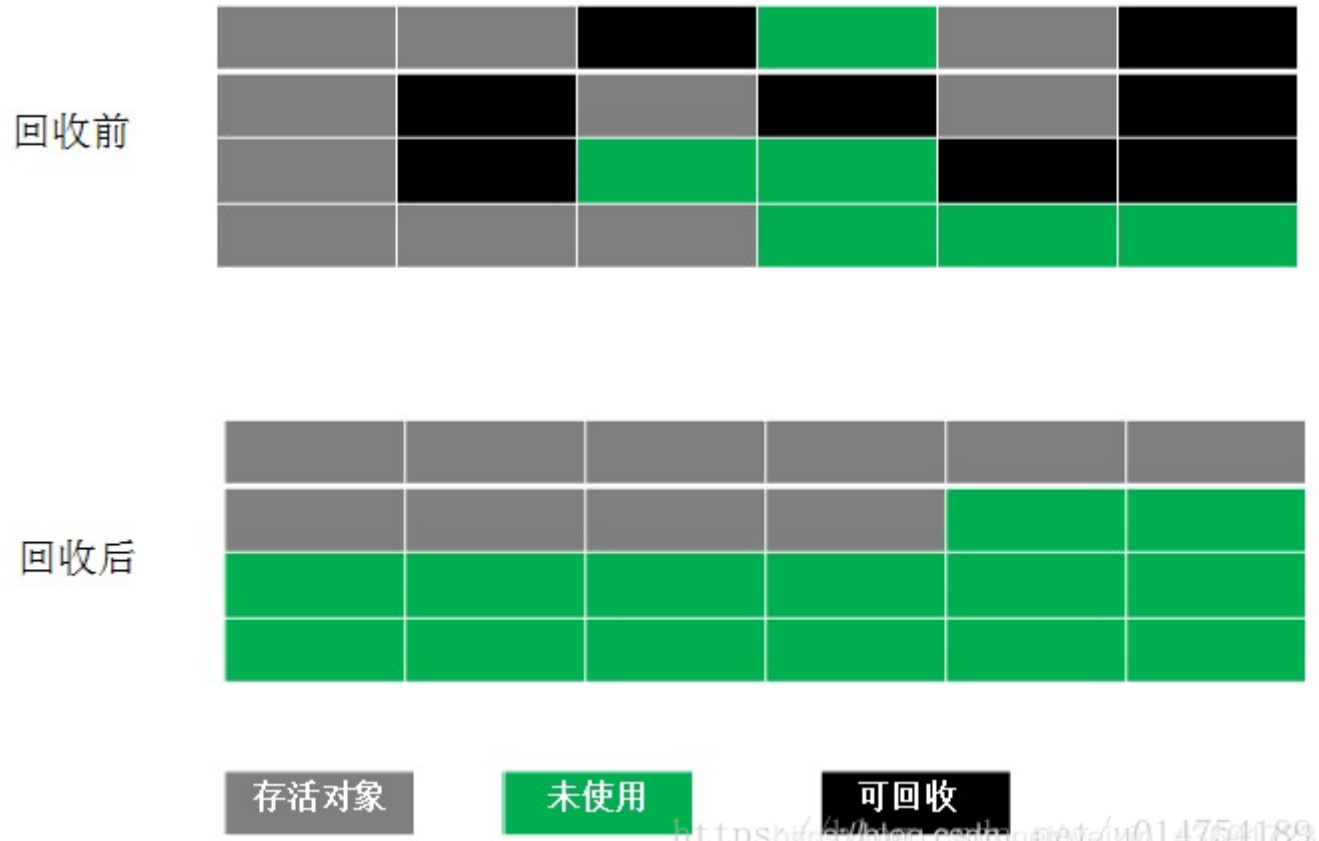
IBM 公司的专门研究表明，新生代中的对象 98% 是 “朝生夕死” 的，所以并不需要按照 1:1 的比例来划分内存空间，而是将内存分为一块较大的 Eden 空间和两块较小的 Survivor 空间，每次使用 Eden 和其中一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性的复制到另外一块 Survivor。当回收时，将 Eden 和 Survivor 中还存活着的对象一次性的复制到另外一块 Survivor 空间上，最后清理掉 Eden 和刚才用过的 Survivor 空间。HotSpot 虚拟机默认 Eden 和 Survivor 的大小比例是 8:1，也就是每次新生代中可用内存为整个新生代容量的 90%（80%+10%），只有 10% 的内存会被“浪费”。



标记-整理算法

为了解决 Copying 算法的缺陷，充分利用内存空间，提出了 Mark-Compact 算法。该算法标记阶段和 Mark-Sweep 一样，但是在完成标记之后，它不是直接清理可回收对象，而是将存活对象都向一端移动，然后清理掉

端边界以外的内存。



分代收集算法

当前商业虚拟机的垃圾收集 都采用分代收集，它根据对象的存活周期的不同将内存划分为几块，一般是把 Java 堆分为新生代和老年代。 在新生代中，每次垃圾收集时都会发现有大量对象死去，只有少量存活，因此可选用复制算法来完成收集， 而老年代中**因为对象存活率高、没有额外空间对它进行分配担保**，就必须使用标记—清除算法或标记—整理算法来进行回收。

1. 年轻代的垃圾收集算法 在年轻代中 jvm 使用的是复制算法，年轻代分三个区。 一个 Eden 区，两个 Survivor 区 (一般而言)。大部分对象在 Eden 区中生成。当 Eden 区满时，还存活的对象将被复制到 Survivor 区（两个中的一个），当这个 Survivor 区满时，此区的存活对象将被复制到另外一个 Survivor 区，当另外一个 Survivor 区也满了的时候，从第一个 Survivor 区复制过来的并且此时还存活的对象，将被复制到 “年老区 (Tenured)”。需要注意，Survivor 的两个区是对称的，没先后关系，所以同一个区中可能同时存在从 Eden 复制过来对象，和从前一个 Survivor 复制过来的对象，而复制到年老区的只有从第一个 Survivor 区过来的对象。而且，Survivor 区总有一个是空的。 当 survivor1 区不足以存放 eden 和 survivor0 的存活对象时，就将存活对象直接存放到老年代**担保机制。**

   新生代 GC（Minor GC）：发生在新生代的垃圾收集动作，因为 Java 对象大多都具有朝生夕灭的特性，因此 Minor GC 非常频繁，一般回收速度也比较快。

2. 老年代的回收算法 老年代的特点是每次回收都只回收少量对象，一般使用的是 Mark-Compact（标记 - 整理）算法。 在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。 内存比新生代也大很多 (大概比例是 1:2)，当老年代内存满时触发 Major GC 或 Full GC，Full GC 发生频率比较低，老年代对象存活时间比较长，存活率标记高。

老年代 GC（Major GC/Full GC）：发生在老年代的 GC，出现了 Major GC，经常会伴随至少一次 Minor GC。由于老年代中的对象生命周期比较长，因此 Major GC 并不频繁，一般都是等待老年代满了后才进行 Full GC，而且其速度一般会比 Minor GC 慢 10 倍以上。另外，如果分配了 Direct Memory，在老年代中进行 Full GC 时，会顺便清理掉 Direct Memory 中的废弃对象。

垃圾回收的时机

1. 当 Eden 区或者 Servior 区不够用了
2. 老年代空间不够用了
3. 方法区空间不够用了
4. 手动回收，System.gc()（不建议使用）

HotSpot的算法细节实现

根节点枚举

枚举根节点：可作为GC Roots的节点主要在全局性的引用（例如常量或类静态属性）与执行上下文（例如栈帧中的本地变量表）中。执行根节点枚举需要stop the world，即不允许和用户线程并发。

安全点和安全区域

1. 安全点：安全点是在程序执行期间的所有GC Root已知并且所有堆对象的内容一致的点。所有线程必须在GC运行之前在安全点阻塞。
2. 安全区域：在一段代码片段中，引用关系不会发生变化。在这个区域中的任意地方开始GC都是安全的。主要是因为有的线程挂起，无法到达安全点。

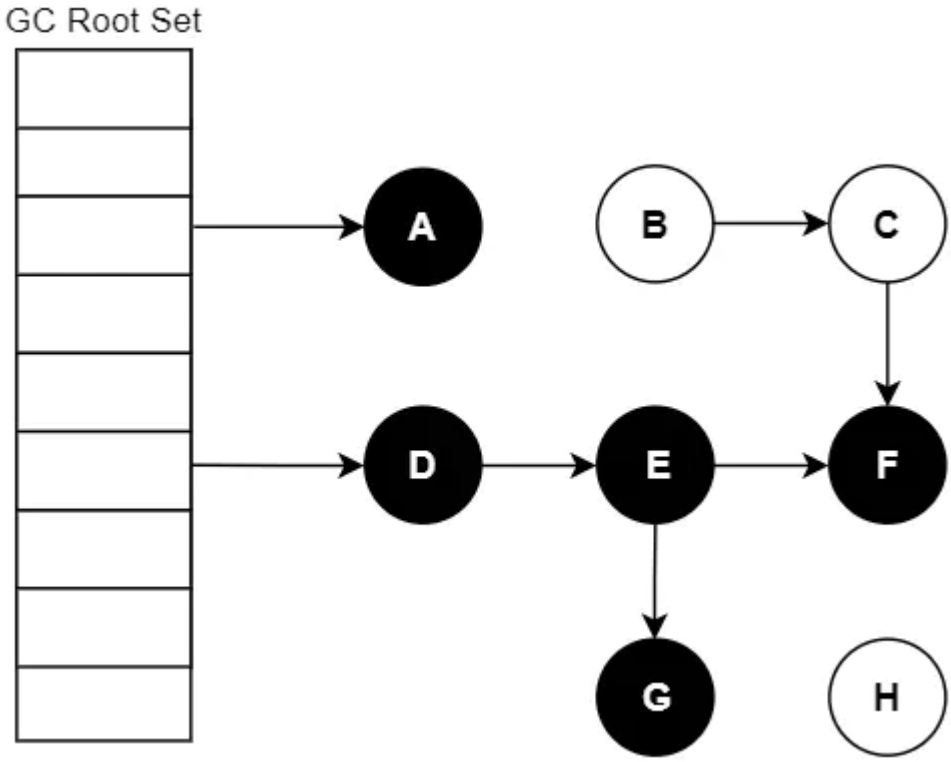
记忆集与卡表

避免全堆都作为GC ROOT 记忆集：记忆集是一种用于记录从非收集区域指向收集区域的指针集合的数据结构。最简单的实现可以用一个对象数组来实现，数组中存着老年代中持有新生代引用的对象。 卡表：卡表是一种高效的记忆集实现，每个记录精确到一块内存区域，该区域中含有跨代指针。最简单的实现是字节数组：crad_table[this address >> 9] = 0; 其中this address就是存在跨代引用的老年代对象地址。首先，计算对象引用所在卡页的卡表索引号。将地址右移9位，相当于用地址除以512（2的9次方）。可以这么理解，假设卡表卡页的起始地址为0，那么卡表项0、1、2对应的卡页起始地址分别为0、512、1024（卡表项索引号乘以卡页512字节）。

写屏障

写屏障：类似AOP，对对象更改后更新卡表

并发的可达性分析：三色理论，增量更新和SATB



要找出存活对象，根据可达性分析，从 GC Roots 开始进行遍历访问，可达的则为存活对象,我们把遍历对象图过程中遇到的对象，按"是否访问过"这个条件标记成以下三种颜色：

- 白色：尚未访问过。
- 黑色：本对象已访问过，而且本对象引用到的其他对象也全部访问过了。
- 灰色：本对象已访问过，但是本对象引用到的其他对象尚未全部访问完。全部访问后，会转换为黑色。

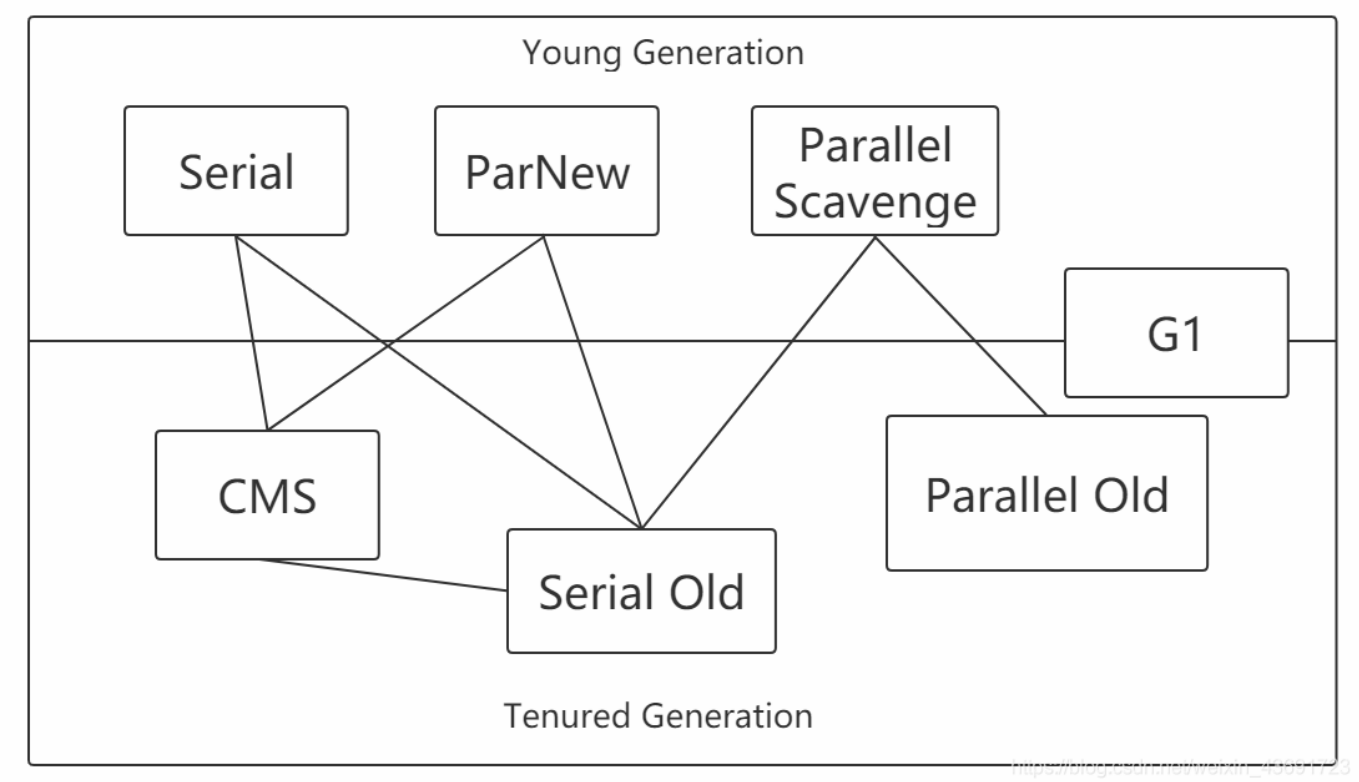
具体过程如下：

1. 初始时，所有对象都在【白色集合】中；
2. 将 GC Roots 直接引用到的对象挪到 【灰色集合】中；
3. 从灰色集合中获取对象：
    1. 将本对象引用到的其他对象全部挪到 【灰色集合】中；
    2. 将本对象挪到【黑色集合】里面。
4. 重复步骤3，直至【灰色集合】为空时结束。
5. 结束后，仍在【白色集合】的对象即为 GC Roots 不可达，可以进行回收。 当 Stop The World （以下简称 STW）时，对象间的引用是不会发生变化的，可以轻松完成标记。 而当需要支持并发标记时，即标记期间应用线程还在继续跑，对象间的引用可能发生变化，多标和漏标的情况就有可能发生。 **漏标** 当已经标记为白色对象（垃圾对象）时，此时程序运行又让他和其他黑色（存活）对象产生引用，那么该对象最终也应该是黑色（存活）对象，如果此时垃圾回收器标记完回收后，会出现对象丢失，这样就引起程序问题。 出现对象丢失的必要条件是（在垃圾回收器标记进行时出现的改变）：
6. 重新建立了一条或多条黑色对象到白色对象的新引用。
7. 删除了灰色对象到白色对象的直接或间接引用 为了防止这种情况的出现，上边说的必要条件中的一个处理掉即可避免对象误删除； 当黑色对象直接引用了一个白色对象后，我们就将这个黑色对象记录下来，在扫描完成后，重新对这个黑色对象扫描,这个就是增量更新（Incremental Update） 当删除了灰色对象

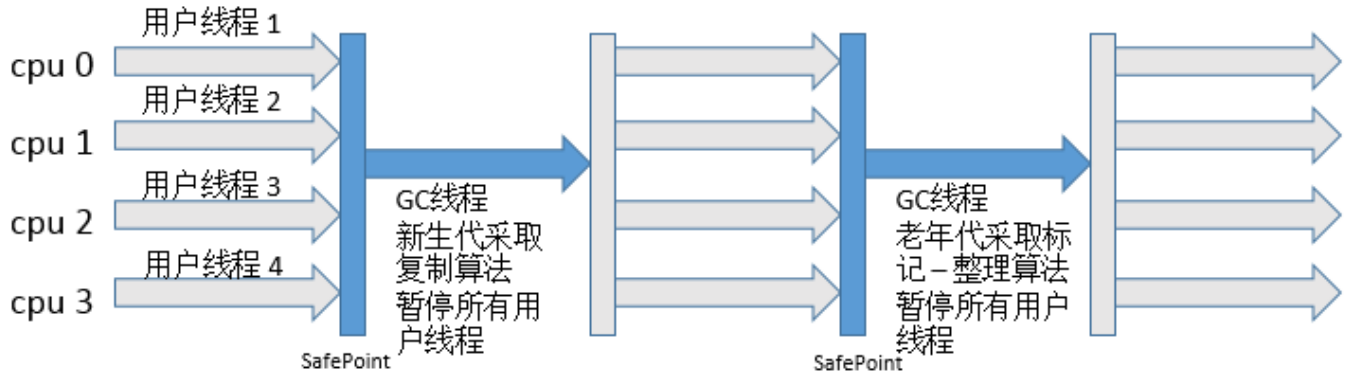
到白色对象的直接或间接引用后，就将这个灰色对象记录下来，再以此灰色对象为根，重新扫描一次。这个就是原始快照（Snapshot At TheBeginning，SATB），相当于不删除 不管是增量更新或者SATB，都是基于写屏障实现，并且这个过程都是要STW的，不过相较于扫描整个引用图，代价要小很多。 参考：https://www.cnblogs.com/jmcui/p/14165601.html

垃圾回收器

垃圾收集算法是内存回收的理论，而垃圾回收器是内存回收的实践。 下图连线表示可以搭配使用。



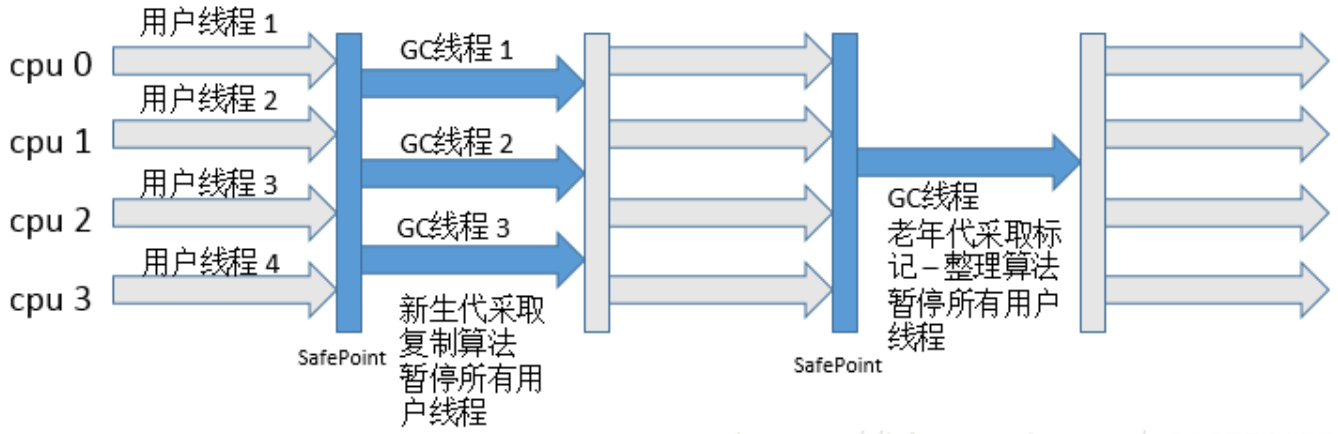
Serial 收集器 和Serial Old收集器



Serial/Serial Old 收集器运行示意图

这是一个新生代单线程收集器，标记和清理都是单线程。意味着它只会使用一个 CPU 或一条收集线程去完成收集工作，并且在进行垃圾回收时必须暂停其它所有的工作线程直到收集结束。简单而高效，但是STW时间较长。

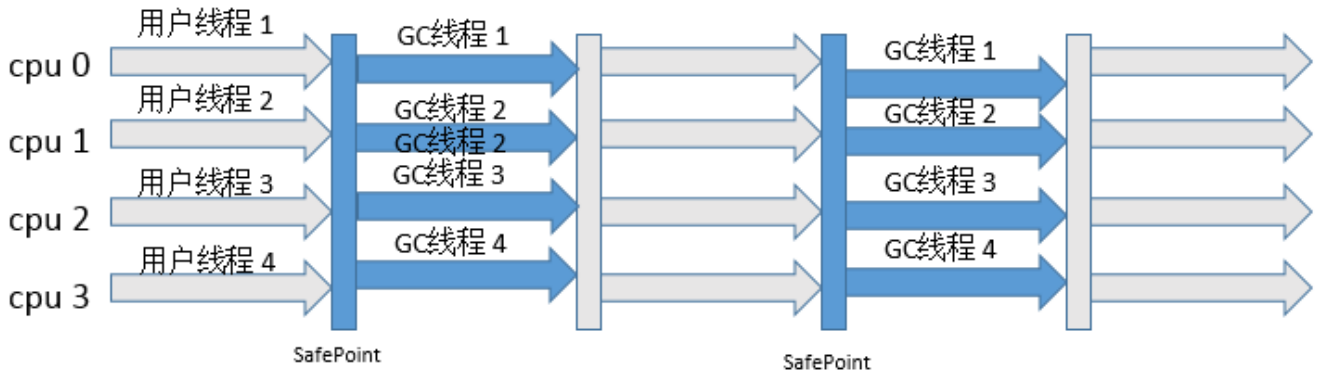
ParNew收集器



ParNew /Serial Old 收集器运行示意图

新生代收集器，可以认为是 Serial 收集器的多线程版本，使用多个线程进行垃圾收集, 在多核 CPU 环境下有着比 Serial 更好的表现。是 Server 模式下的虚拟机首选的新生代收集器，其中有一个很重要的和性能无关的原因是，除了 Serial 收集器外，目前只有它能与 CMS 收集器配合工作。

Parallel Scavenge 收集器和Parallel Old 收集器



Parallel Scavenge/Parallel Old 收集器运行示意图

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。适合不需太多交互的后台程序。 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%。 延迟是STW时间。

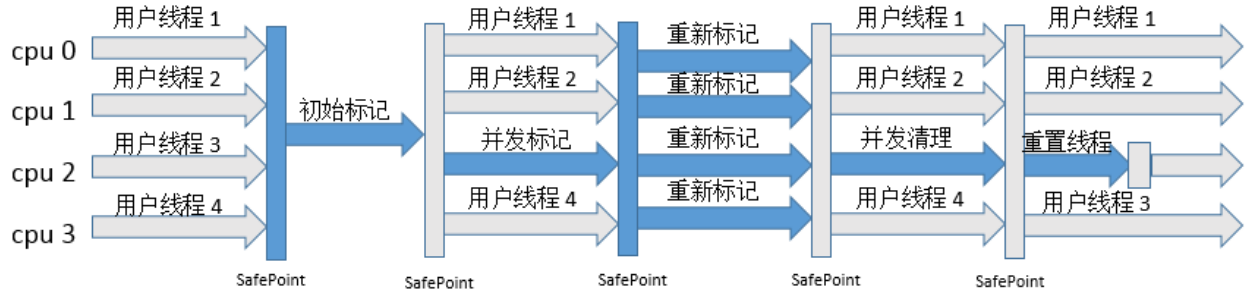
Parallel Scavenge 收集器的老年代版本，并行收集器，吞吐量优先。使用多线程和标记 - 整理（Mark-Compact）算法。

CMS收集器

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（低延迟）。它是一种老年代并发收集器，采用的是 Mark-Sweep 算法。 整个过程分为 4 步：

1. 初始标记，标记 GCRoots 能直接关联到的对象，时间很短。所以这里用的是单线程，会导致停顿（stw，stop the world）。
2. 并发标记，进行 GCRoots Tracing（可达性分析）过程，时间很长。所以用的是多线程。
3. 重新标记，修正并发标记期间的变动部分，时间较长，会导致停顿（stw，stop the world）。使用增量更新方式。

4. 并发清除，回收内存空间，时间很长。因为回收的都是死掉的对象，所以可以直接并发。



Concurrent Mark Sweep收集器运行示意图

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：

5. 对 CPU 资源非常敏感，可能会导致应用程序变慢，吞吐率下降。因为有GC线程和用户线程同时并发，竞争CPU资源。
6. 无法处理浮动垃圾，因为在并发清理阶段用户线程还在运行，自然就会产生新的垃圾，而在此次收集中无法收集他们，只能留到下次收 集，这部分垃圾为浮动垃圾，同时，由于用户线程并发执行，所以需要预留一部分老年代空间提供并发收集时程序运行使用。如果老年代资源不够会启用后被预案：冻结所有用户线程，调用serial old进行GC，效率很低。
7. 内存碎片导致调用full gc

G1收集器

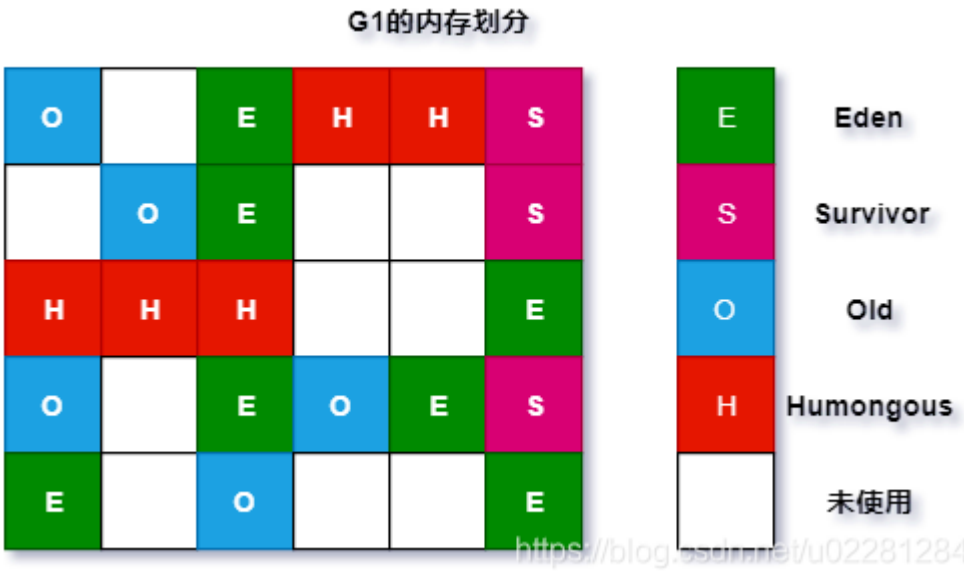
G1收集器的目标是能建立起停顿时间模型，即在一个长度为M毫秒的时间片段内，消耗在垃圾收集上的时间大概率不超过N毫秒这样的目标。

G1的堆内存划分

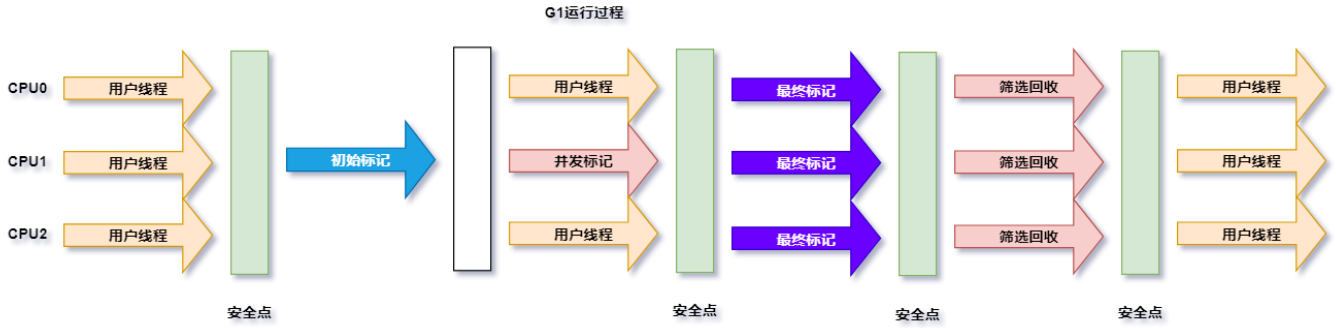
为了实现 STW 的时间可预测，首先要有一个思想上的改变。G1 将堆内存 “化整为零”，将堆内存划分成多个大小相等独立区域（Region），每一个 Region 都可以根据需要，扮演新生代的 Eden 空间、Survivor 空间，或者老年代空间。收集器能够对扮演不同角色的 Region 采用不同的策略去处理，这样无论是新创建的对象还是已经存活了一段时间、熬过多次收集的旧对象都能获取很好的收集效果。Region 可能是 Eden，也有可能是 Survivor，也有可能是 Old，另外 Region 中还有一类特殊的 Humongous 区域，专门用来存储大对象。G1 认为只要大小超过了一个 Region 容量一半的对象即可判定为大对象。每个 Region 的大小可以通过参数-XX:G1HeapRegionSize设定，取值范围为 1MB~32MB，且应为 2 的 N 次幂。而对于那些超过了整个 Region 容量的超级大对象，将会被存放在 N 个连续的 Humongous Region 之中，G1 的进行回收大多数情况下都把 Humongous Region 作为老年代的一部分来进行看待。 G1 在逻辑上还是划分 Eden、Survivor、OLd，但是物理上他们不是连续的。

G1之前的回收器，都是要么针对整个新生代要么针对整个老年代，那么停顿时间就取决于两个区的垃圾大小，是不可控的。G1回收的基本单位是Region，它可以面向 堆内存任意部分来组成回收集进行回收。衡量标准不

再是它属于那个分代，而是哪块内存中垃圾数量最多，回收收益最大，这就是G1收集器的Mixed GC模式。



G1的运行过程

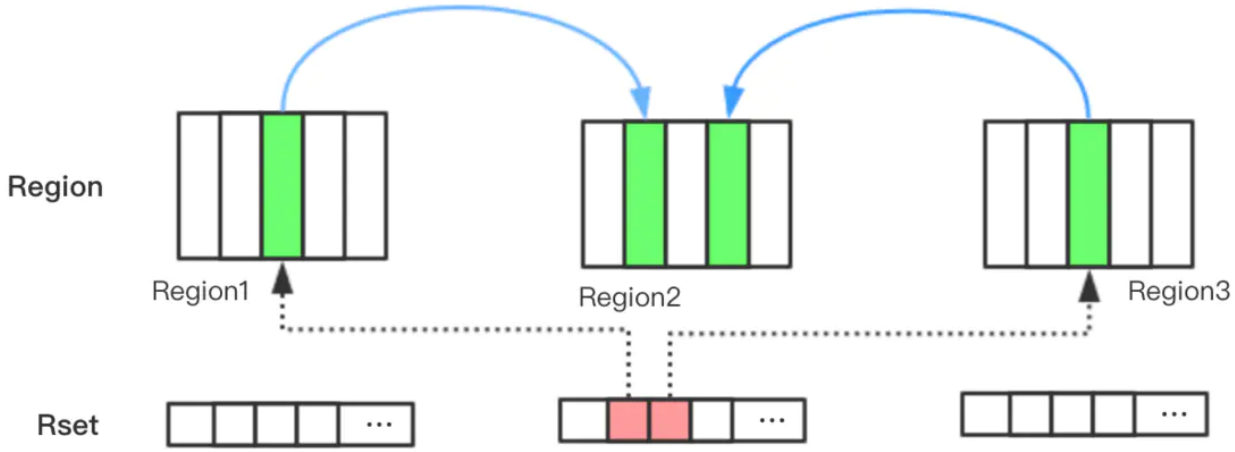


G1 的运行过程与 CMS 大体一致，分为以下四个步骤：

1. 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 指针的值，让下一阶段用户线程并发运行时，能正确地在可用的 Region 中分配新对象。这个阶段需要停顿线程，但耗时很短，而且是借用进行 Minor GC 的时候同步完成的，所以 G1 收集器在这个阶段实际并没有额外的停顿。TAMS 是什么？要达到 GC 与用户线程并发运行，必须要解决回收过程中新对象的分配，所以 G1 为每一个 Region 区域设计了两个名为 TAMS（Top at Mark Start）的指针，从 Region 区域划出一部分空间用于记录并发回收过程中的新对象。这样的对象认为它们是存活的，不纳入垃圾回收范围。
2. 并发标记 (Concurrent Marking)：从 GC Root 开始对堆中对象进行可达性分析，递归扫描整个堆里的对象图，找出要回收的对象，这阶段耗时较长，但可与用户程序并发执行。当对象图扫 描完成以后，并发时有引用变动的对象会产生漏标问题，G1 中会使用 SATB(snapshot-at-the-beginning) 算法来解决，后面会详细介绍。
3. 最终标记 (Final Marking)：对用户线程做一个短暂的暂停，用于处理并发标记阶段仍遗留下来的最后那少量的 SATB 记录 (漏标对象)。
4. 筛选回收 (Live Data Counting and Evacuation)：负责更新 Region 的统计数据，**对各个 Region 的回收价值和成本进行排序，根据用户所期望的停顿时间来制定回收计划，可以自由选择任意多个 Region 构成回收集，然后把决定回收的那一部分 Region 的存活对象复制到空的 Region 中，再清理掉整个旧**

Region 的全部空间。这里的操作涉及存活对象的移动，是必须暂停用户线程，由多个收集器线程并行完成的。

Rset



每个Region初始化时，会初始化一个remembered set（已记忆集合），这个翻译有点拗口，以下简称RSet，该集合用来记录并跟踪其它Region指向该Region中对象的引用，每个Region默认按照512Kb划分成多个Card，所以RSet需要记录的东西应该是 xx Region的 xx Card。 用来解决跨Region引用

Shenandoan收集器和ZGC收集器

参考 https://blog.csdn.net/gaohaicheng123/article/details/106437504 Shenandoan和ZGC都将G1的筛选回收并行化。 筛选回收并行化的核心难点在于对象已经被GC线程复制到其他的Region中去，那么用户线程想要并行访问修改怎么办?

Shenandoan使用转发指针，即在对象头上加了一个新的指针，默认指向自己，如果被移动就指向新对象位置。在使用读屏障增加用户访问对象先访问对象转发指针的操作。并使用CMS保持线程安全。

ZGC使用染色指针，直接将指针用不到的高位上加了几个标志位，线程能通过指针直接判断是否对象移动了，如果是在从ZGC维护的转发表中找到新位置。 ZGC因为修改了指针高位，所以同一个对象可能因为是否移动而具有不同的指针，为了解决这个问题，ZGC使用内存映射将同一个对象的染色指针都映射到同一个位置。