内存申请

java一般内存申请有两种：静态内存和动态内存。编译时就能够确定的内存就是静态内存，即内存是固定的，系统一次性分配，比如基本类型变量；动态内存分配就是在程序执行时才知道要分配的存储空间大小，比如java对象的内存空间。java栈、程序计数器、本地方法栈都是线程私有的，线程生就生，线程灭就灭，栈中的栈帧随着方法的结束也会撤销，内存自然就跟着回收了。所以这几个区域的内存分配与回收是确定的，我们不需要管的。但是java堆则不一样，我们只有在程序运行期间才知道会创建哪些对象，所以这部分内存的分配和回收都是动态的。一般我们所说的垃圾回收也是针对的这一部分。

总之Stack的内存管理是顺序分配的，而且定长，不存在内存回收问题；而Heap 则是为java对象的实例随机分配内存，不定长度，所以存在内存分配和回收的问题；

将对象按其生命周期的不同划分成：年轻代(Young Generation)、年老代(Old Generation)、持久代(Permanent Generation)。
年轻代：是所有新对象产生的地方。年轻代被分为3个部分——Enden区和两个Survivor区（From和to）。当Eden区被对象填满时，就会执行Minor GC（清理年轻代区域）。并把所有存活下来的对象（from区和Enden区）转移到其中一个survivor区（to区）。在一段时间内，总会有一个空的survivor区。经过多次GC周期后，仍然存活下来的对象会被转移到年老代内存空间。或当“年轻代”区域存放满对象后，就将对象存放到年老代区域。

年老代：在年轻代中经历了N（默认15）次回收后仍然没有被清除的对象，就会被放到年老代中，都是生命周期较长的对象。通常会在老年代

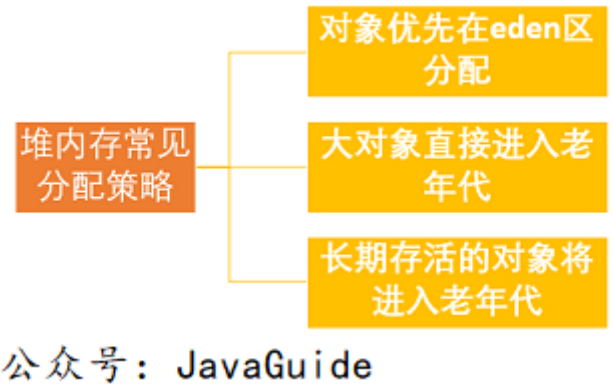
内存被占满时将会触发Major GC（清理老年代区域）和Full GC（成本高），回收整个堆内存（包括年轻代和年老代）。

导致Full GC：

1. 年老代(Tenured)被写满
2. 持久代(Perm)被写满
3. System.gc()被显式调用（程序建议GC启动，不是调用GC）
4. 上一次GC之后Heap的各域分配策略动态变化

持久代：用于存放静态文件，比如java类信息、方法等。持久代对垃圾回收没有显著的影响。jdk1.7已经去永久代了，永久代是当jvm启动时就存放的JDK自身的类和接口数据，关闭则释放。

对象分配规则



对象优先分配在Eden区，如果Eden区没有足够的空间时，虚拟机执行一次Minor GC。

- 新生代 GC（Minor GC）:指发生新生代的的垃圾收集动作，Minor GC 非常频繁，回收速度一般也比较快。
- 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（并非绝对），Major GC 的速度一般会比 Minor GC 的慢 10 倍以上

大对象直接进入老年代。大对象就是需要大量连续内存空间的对象。eg.字符串、数组。目的是避免在Eden区和两个Survivor区之间发生大量的内存拷贝（新生代采用复制算法收集内存）。避免为大对象分配内存时由于分配担保机制带来的复制而降低效率。

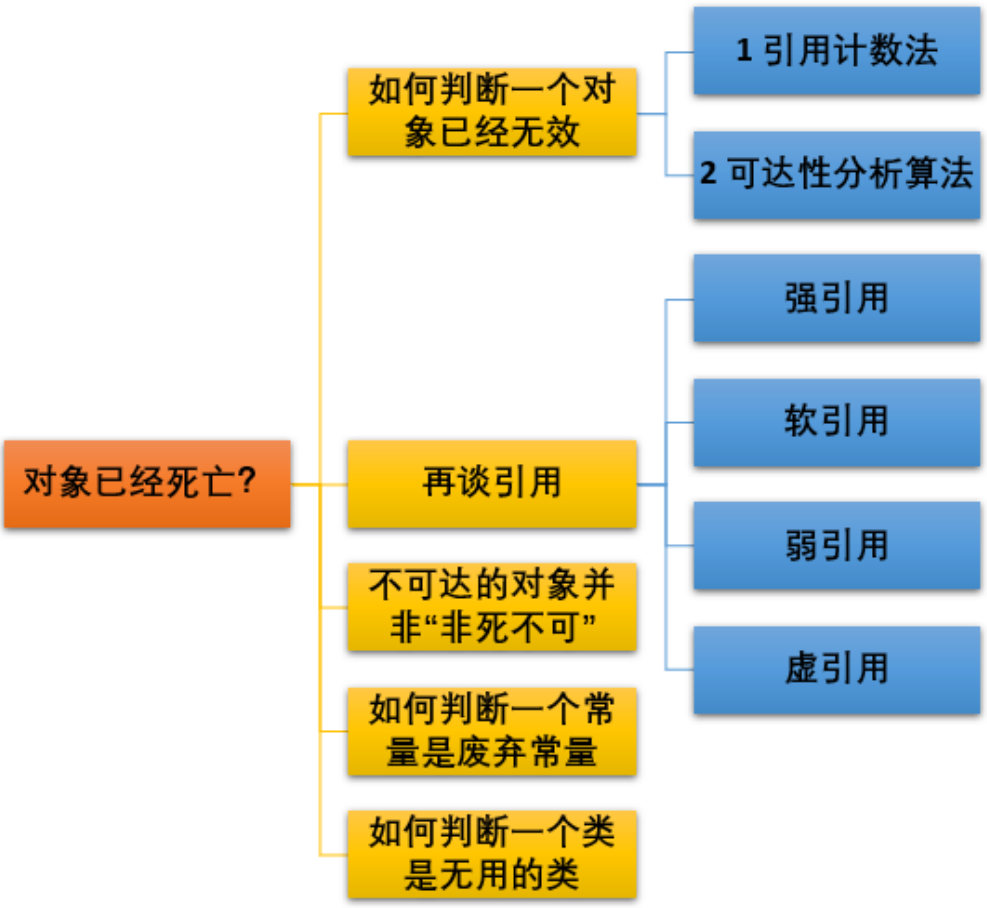
- 空间分配担保。每次进行Minor GC时，JVM会计算Survivor区移至老年区的对象的平均大小，如果这个值大于老年区的剩余值大小则进行一次Full GC，如果小于检查HandlePromotionFailure设置，如果true则只进行Monitor GC,如果false则进行Full GC。

长期存活的对象进入老年代。虚拟机为每个对象定义了一个年龄计数器，如果对象经过了1次Minor GC那么对象会进入Survivor区，之后每经过一次Minor GC那么对象的年龄加1，直到达到阀值15岁对象进入老年区。可以通过参数 -XX:MaxTenuringThreshold 来设置。

动态判断对象的年龄。如果Survivor区中相同年龄的所有对象大小的总和大于Survivor空间的一半，年龄大于或等于该年龄的对象直接进入老年代。

垃圾回收

垃圾收集器一般必须完成两件事：检测出垃圾；回收垃圾。

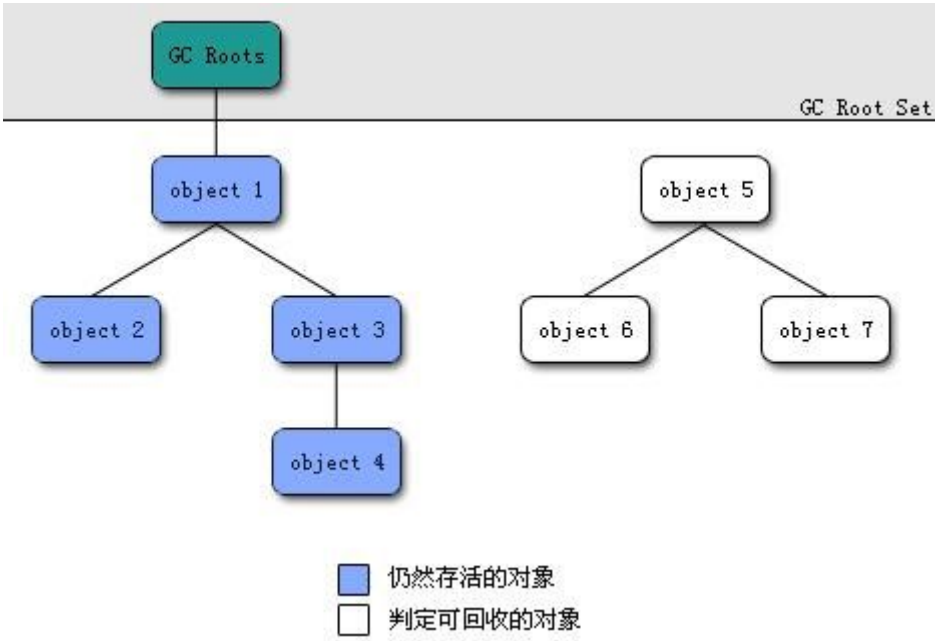


检测垃圾：

1. **引用计数法**：给一个对象添加引用计数器，每当有个地方引用它，计数器就加1；引用失效就减1。

问题：如果我有两个对象A和B，互相引用，除此之外，没有其他任何对象引用它们，实际上这两个对象已经无法访问，即是我们说的垃圾对象。但是互相引用，计数不为0，导致无法回收。

2. **可达性分析算法**：以根集对象GC ROOT为起始点进行搜索，如果有对象不可达的话，即是垃圾对象。这里的根集一般包括java栈中引用的对象、方法区常良池中引用的对象本地方法中引用的对象等。



释放对象的根本原则就是对象不会再被使用：

- 给对象赋予了空值null，之后再没有调用过。
- 另一个是给对象赋予了新值，这样重新分配了内存空间。

“无用的类”可以被回收，但不一定 ：

- 该类所有的实例都已经被回收，也就是 Java 堆中不存在该类的任何实例。
- 加载该类的 ClassLoader 已经被回收。
- 该类对应的 java.lang.Class 对象没有在任何地方被引用，无法在任何地方通过反射访问该类的方法。

垃圾收集算法：

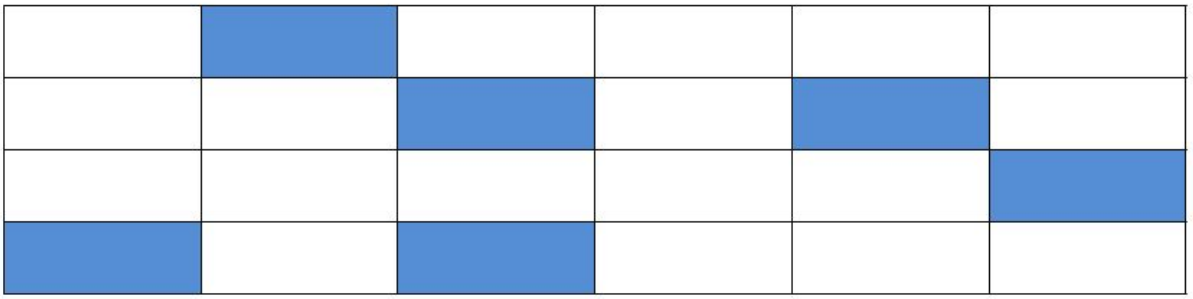
1. 标记-清除（Mark-sweep）

首先标记出所有需要回收的对象，在标记完成后统一回收所有被标记的对象。

不足：效率低；标记清除之后会产生大量碎片；

内存整理前

内存整理后

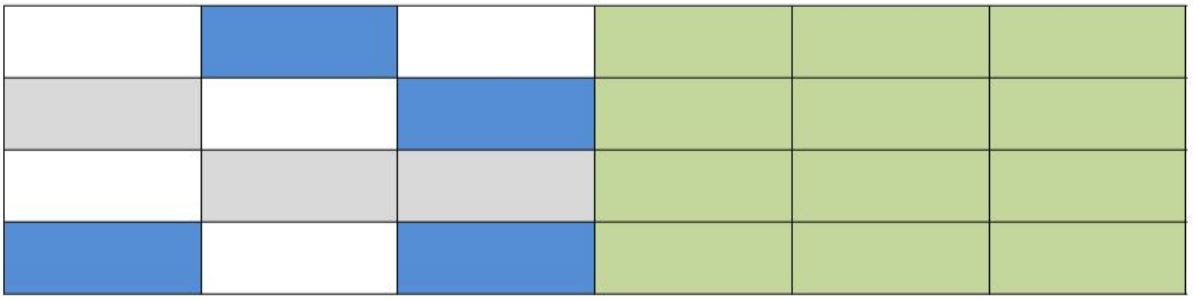


2. 复制（Copying）

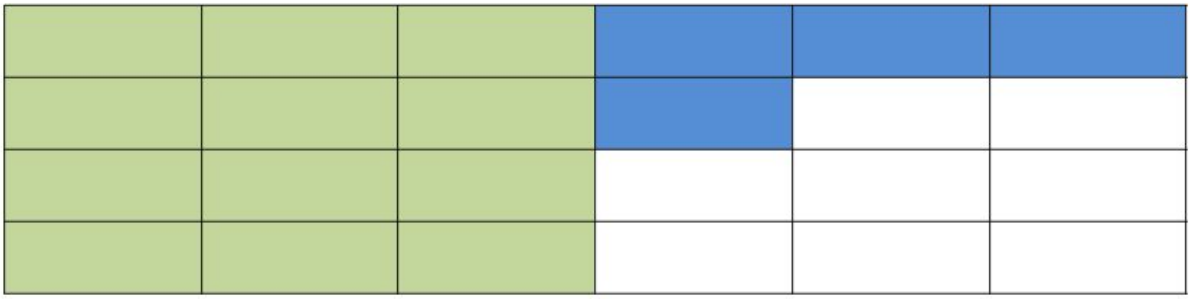
将内存分为大小相同的两块，每次使用其中的一块。当这一块的内存使用完后，就将还存活的对象复制到另一块去，然后再把使用的空间一次清理掉。这样就使每次的内存回收都是对内存区间的一半进行回收。

缺点：需要两倍内存空间。

内存整理前



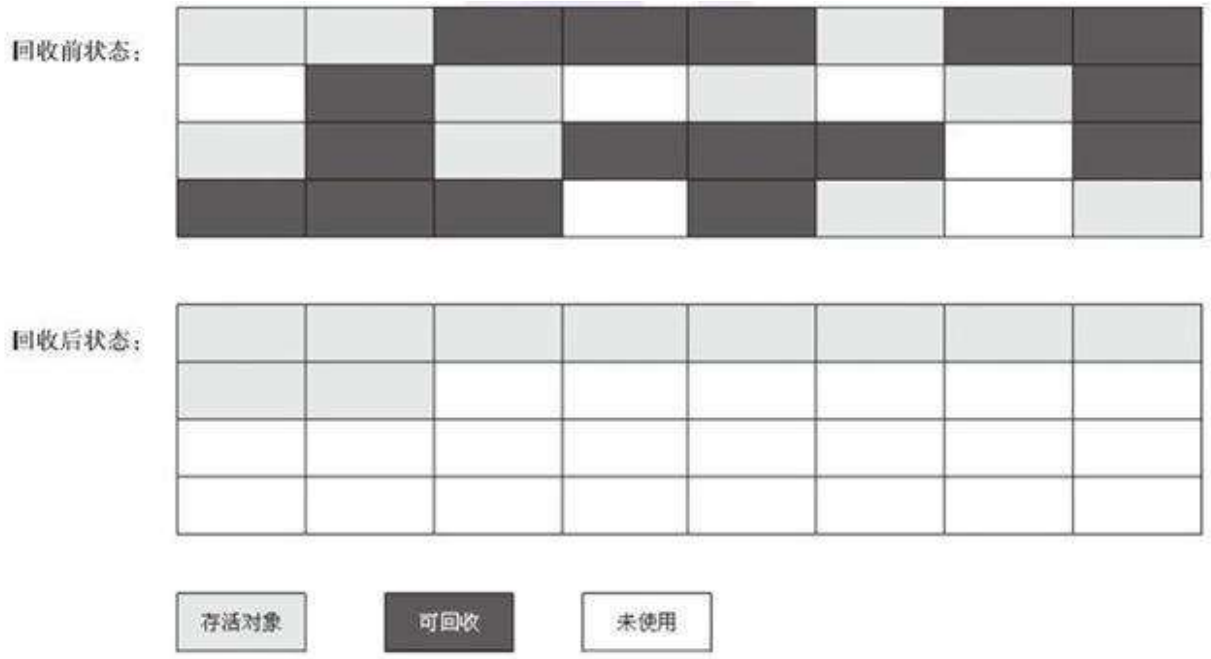
内存整理后



3.标记-整理（Mark-Compact）

标记过程与“标记-清除”算法一样，但后续不是直接对可回收对象回收，而是让所有存活的对象向一端移动，然后直接清理掉端边界以外的内存。

此算法结合了“标记-清除”和“复制”两个算法的优点。



4. 分代收集算法

分代的垃圾回收策略，是基于这样一个事实：不同的对象的生命周期是不一样的。因此，不同生命周期的对象可以采取不同的收集方式，提高回收效率。

eg. 在新生代中，每次收集都会有大量对象死去，所以可以选择复制算法，只需要付出少量对象的复制成本就可以完成每次垃圾收集。而老年代的对象存活几率是比较高的，而且没有额外的空间对它进行分配担保，所以我们必须选择“标记-清除”或“标记-整理”算法进行垃圾收集。

垃圾收集器

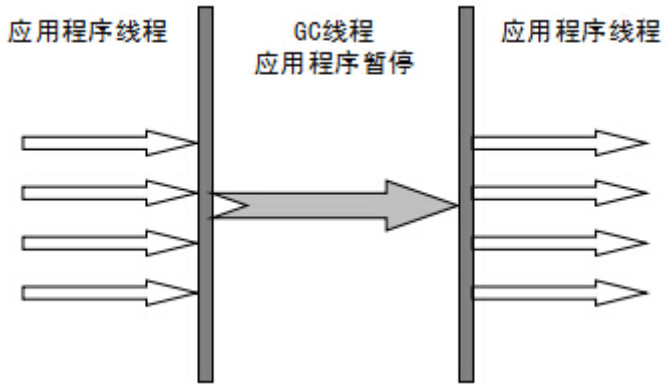
如果说收集算法是内存回收的方法论，那么垃圾收集器就是内存回收的具体实现。

4.1 Serial 收集器

Serial（串行）收集器收集器是最基本、历史最悠久的垃圾收集器了。大家看名字就知道这个收集器是一个单线程收集器了。它的 “单线程” 的意义不仅仅意味着它只会使用一条垃圾收集线程去完成垃圾收集工作，更重要的是它在进行

垃圾收集工作的时候必须暂停其他所有的工作线程（ "Stop The World" ），直到它收集结束。

新生代采用复制算法，老年代采用标记-整理算法。



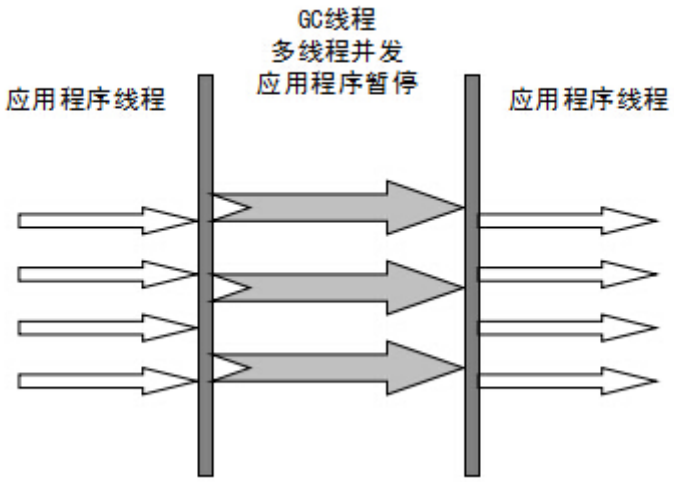
虚拟机的设计者们当然知道 Stop The World 带来的不良用户体验，所以在后续的垃圾收集器设计中停顿时间在不断缩短（仍然还有停顿，寻找最优秀的垃圾收集器的过程仍然在继续）。

但是 Serial 收集器有没有优于其他垃圾收集器的地方呢？当然有，它简单而高效（与其他收集器的单线程相比）。Serial 收集器由于没有线程交互的开销，自然可以获得很高的单线程收集效率。Serial 收集器对于运行在 Client 模式下的虚拟机来说是个不错的选择。

4.2 ParNew 收集器

ParNew 收集器其实就是 Serial 收集器的多线程版本，除了使用多线程进行垃圾收集外，其余行为（控制参数、收集算法、回收策略等等）和 Serial 收集器完全一样。

新生代采用复制算法，老年代采用标记-整理算法。



它是许多运行在 Server 模式下的虚拟机的首要选择，除了 Serial 收集器外，只有它能与 CMS 收集器（真正意义上的并发收集器，后面会介绍到）配合工作。

并行和并发概念补充：

- 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，可能会交替执行），用户程序在继续运行，而垃圾收集器运行在另一个 CPU 上。

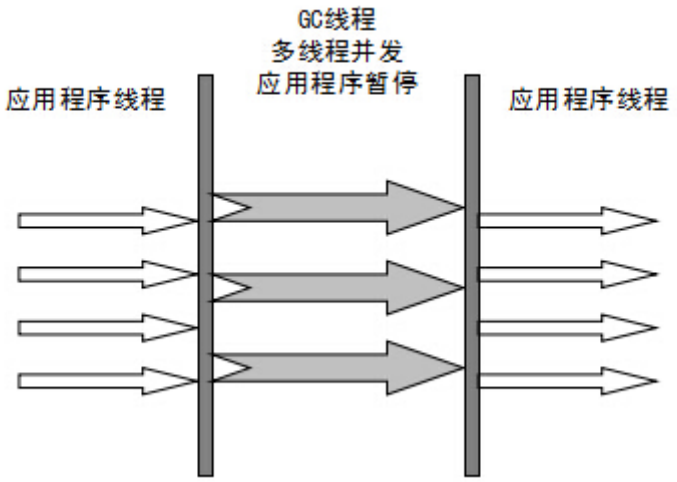
4.3 Parallel Scavenge 收集器

Parallel Scavenge 收集器也是使用复制算法的多线程收集器，它看上去几乎和ParNew都一样。 那么它有什么特别之处呢？

```
-XX:+UseParallelGC

    使用 Parallel 收集器+ 老年代串行

-XX:+UseParallelOldGC

    使用 Parallel 收集器+ 老年代并行
```

Parallel Scavenge 收集器关注点是吞吐量（高效率的利用 CPU）。CMS 等垃圾收集器的关注点更多的是用户线程的停顿时间（提高用户体验）。所谓吞吐量就是 CPU 中用于运行用户代码的时间与 CPU 总消耗时间的比值。 Parallel Scavenge 收集器提供了很多参数供用户找到最合适的停顿时间或最大吞吐量，如果对于收集器运作不太了解的话，手工优化存在的话可以选择把内存管理优化交给虚拟机去完成也是一个不错的选择。

新生代采用复制算法，老年代采用标记-整理算法。



4.4.Serial Old 收集器

Serial 收集器的老年代版本，它同样是一个单线程收集器。它主要有两大用途：一种用途是在 JDK1.5 以及以前的版本中与 Parallel Scavenge 收集器搭配使用，另一种用途是作为 CMS 收集器的后备方案。

4.5 Parallel Old 收集器

Parallel Scavenge 收集器的老年代版本。使用多线程和“标记-整理”算法。在注重吞吐量以及 CPU 资源的场合，都可以优先考虑 Parallel Scavenge 收集器和 Parallel Old 收集器。

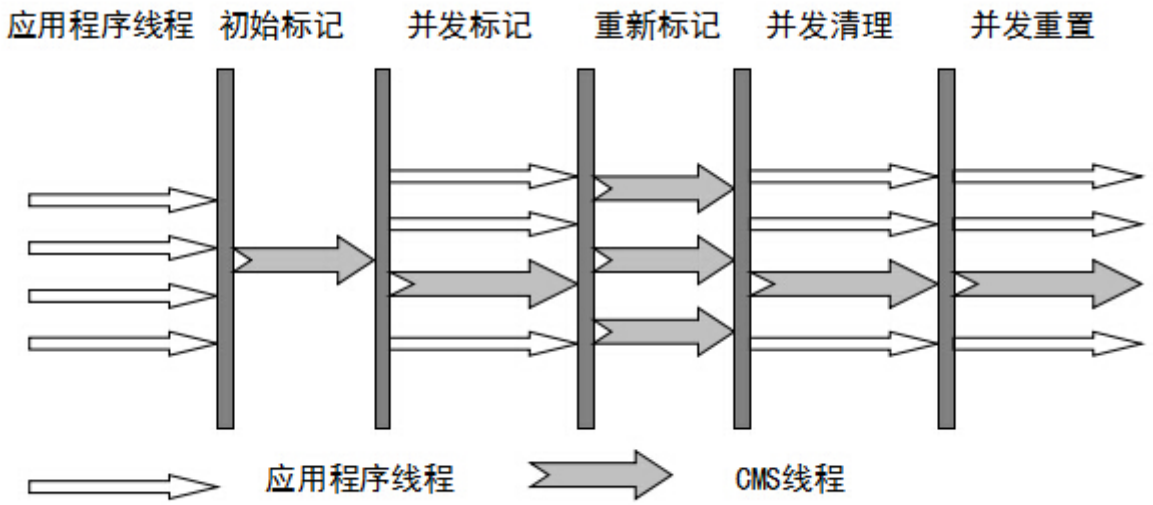
4.6 CMS 收集器

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。它而非常符合在注重用户体验的应用上使用。

CMS（Concurrent Mark Sweep）收集器是 HotSpot 虚拟机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（基本上）同时工作。从名字中的Mark Sweep这两个词可以看出，CMS 收集器是一种 “标记-清除”算法实现的，它的运作过程相比于前面几种垃圾收集器来说更加复杂一些。整个过程分为四个步骤：

- 初始标记： 暂停所有的其他线程，并记录下直接与 root 相连的对象，速度很快 ；
- 并发标记： 同时开启 GC 和用户线程，用一个闭包结构去记录可达对象。但在这个阶段结束，这个闭包结构并不能保证包含当前所有的可达对象。因为用户线程可能会不断的更新引用域，所以 GC 线程无法保证可达性分析的实时性。所以这个算法里会跟踪记录这些发生引用更新的地方。

- 重新标记： 重新标记阶段就是为了修正并发标记期间因为用户程序继续运行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的时间稍长，远远比并发标记阶段时间短
- 并发清除： 开启用户线程，同时 GC 线程开始对为标记的区域做清扫。



从它的名字就可以看出它是一款优秀的垃圾收集器，主要优点：并发收集、低停顿。但是它有下面三个明显的缺点：

- 对 CPU 资源敏感；
- 无法处理浮动垃圾；
- 它使用的回收算法-“标记-清除”算法会导致收集结束时会有大量空间碎片产生。

4.7 G1 收集器

G1 (Garbage-First) 是一款面向服务器的垃圾收集器,主要针对配备多颗处理器及大容量内存的机器. 以极高概率满足 GC 停顿时间要求的同时,还具备高吞吐量性能特征.

被视为 JDK1.7 中 HotSpot 虚拟机的一个重要进化特征。它具备一下特点：

- 并行与并发：G1 能充分利用 CPU、多核环境下的硬件优势，使用多个 CPU（CPU 或者 CPU 核心）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：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，但是还是保留了分代的概念。
- 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；从局部上来看是基于“复制”算法实现的。
- 可预测的停顿：这是 G1 相对于 CMS 的另一个大优势，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。

G1 收集器的运作大致分为以下几个步骤：

- 初始标记
- 并发标记
- 最终标记
- 筛选回收

G1 收集器在后台维护了一个优先列表，每次根据允许的收集时间，优先选择回收价值最大的 Region(这也就是它的名字 Garbage-First 的由来)。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 GF 收集器在有限时间内可以尽可能高的收集效率（把内存化整为零）。