垃圾收集（GC）主要针对**Java堆**和**方法区**，解决三个问题：

哪些内存需要回收？

什么时候回收？

怎么回收？

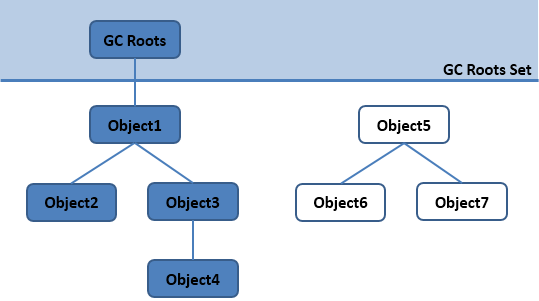
1. 对象存活判定算法
2. 引用计数算法

算法过程：给对象添加引用计数器，每有一个地方引用它时计数器加1；引用失效时计数器减1；计数器为0时对象就是不可能再被使用的（找不到了）。

优缺点：实现简单，判定效率高（如Python、FlashPlayer）；但难以解决对象之间循环引用的问题。

1. 可达性分析算法

可达性分析（tracing GC）判定对象是否存活，算法过程：通过一系列称为“GC Roots”的对象作为起点往下搜索，搜索走过的路径为引用链（Reference Chain），当一个对象到GC Roots没有任何引用链时，此对象不可用，如图。

（object5、6、7不可用）

GC Roots：

它不是对象图里的对象，不会被对象图内的对象所引用，避免了循环引用问题。它不是一组对象，通常是一组特别管理的**指向引用类型对象的指针**，因此只有引用类型的变量才可能是Roots，值类型的变量不可能是Roots。

如何选择GC Roots？

Tracing GC必须选当前存活的对象集为Roots：虚拟机栈、本地方法栈、方法区不被GC管理（GC主要管Java堆），所以从这些区域选Roots。虚拟机栈和本地方法栈都是线程私有的内存区域，线程没终止，它们所引用的对象就存活。因此，可能作为GC Roots的对象包括：

* 虚拟机栈（栈帧中的局部变量表LVT）中引用的对象
* 方法区中类静态属性引用的对象
* 方法区中常量引用的对象
* 本地方法栈中JNI（即Native方法）引用的对象

1. 两次标记与finalize( )方法

可达性分析算法中不可达的对象也不一定会死亡，宣告对象的死亡至少要经历两次标记过程：

对象可达性分析后发现没有与GC Roots相连的引用链，会被第一次标记并进行一次筛选，筛选条件是此对象是否有必要执行finalize( )方法（对象没有覆盖或者已经被调用过finalize( )方法，则没有必要执行）。

如果有必要执行finalize( )方法（只会执行一次），则将对象放入F-Queue，虚拟机会用一个低优先级的finalizer线程去执行它。稍后GC将对F-Queue 队列中的对象进行第二次小规模的标记。期间对象若能重新与引用链上的任何一个对象建立关联即可拯救自己。（最好不用此方法）

1. 回收方法区

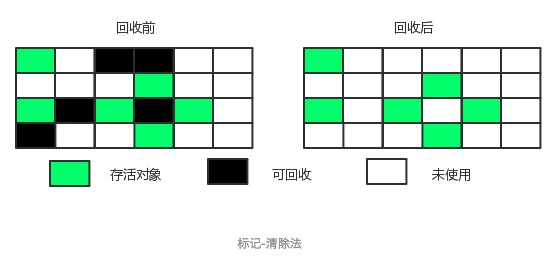
JDK1.8中抛弃了永久代，用元空间作为方法区的实现。元空间内存管理由元空间虚拟机来完成。（每一个类加载器的存储区域都称作一个元空间，所有的元空间合在一起就是我们一直说的元空间。）

1. 垃圾收集算法
2. 标记-清除（Mark-Sweep）算法

过程：标记出所有需要回收的对象，标记完成后统一回收。

缺点：

* 空间问题：标记清除后会产生不连续的内存碎片，可能无法找到足够的连续内存而提前触发另一次GC。
* 效率问题：标记和清除效率都不高，内存碎片会使操作更费时。



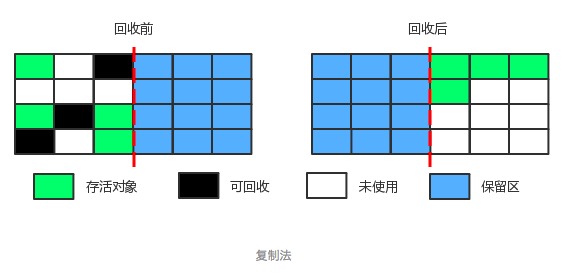
1. 复制（Copying）算法（适用新生代）

**过程：**将可用内存分成两块，每次只使用其中的一块。当一块用完了就将存活对象复制到另一块上，然后清理已用块的内存空间。

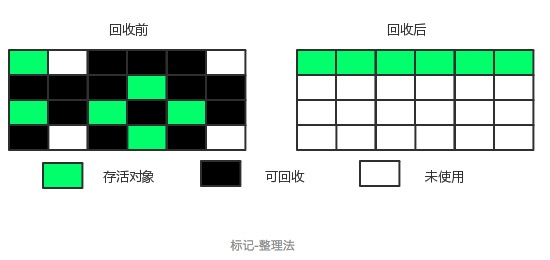
**缺点：**内存分配时不用考虑内存碎片等问题，只需要移动堆顶指针，按顺序分配内存。但是以**内存缩小**作为代价，且在对象存活率高时复制操作变多。

**Minor GC（新生代GC）：**商业上用复制算法回收新生代，Minor GC回收频繁，速度快。一般分为较大的Eden空间和两块较小的Survivor空间（From Survivor和To Survivor，HotSpot里Eden和Survivor为8：1），回收时将Eden和From中存活的对象复制到To空间上，然后清理Eden和From，下次Minor GC时From和To交换标签。

**分配担保（Handle Promotion）：**当Survivor空间不够用时，这些对象将直接通过分配担保机制进入老年代。



1. 标记-整理（Mark-Compact）算法（适用老年代）

过程：标记后将所有存活对象向一端移动，然后清理掉端边界外的内存。

1. 分代收集（Generational Collection）算法

将Java堆分为新生代和老年代，根据年代特点采用不同收集算法：

* 新生代：对象存活率低，复制内容少。选择复制算法
* 老年代：存活率高，无额外分配担保空间。选择“标记-清除”或“标记-整理”。

1. HotSpot的算法实现（不懂）
2. 枚举根节点

在HotSpot的实现中，使用**准确式GC**（使用准确式内存管理，虚拟机可用知道内存中某个位置的数据具体是什么类型），当GC停顿（Stop The World）时，用OopMap数据结构快速完成GC Roots枚举。

1. 安全点（Safe Point）

HotSpot选择不为每条指令都生成OopMap，而是只在“特定的位置”记录这些信息，这些位置便被称为安全点（Safepoint）。程序执行时并非在所有地方都能停顿下来开始GC，只有在到达安全点时才能暂停。

1. 安全区域（Safe Region）
2. 内存分配策略

分配规则取决于垃圾收集器的具体组合以及虚拟机中与内存相关的参数设置，以Serial/Serial Old收集器为例：

1. 对象优先在Eden区分配

对象一般在Eden区分配，若无足够空间进行分配，虚拟机发起一次Minor GC。

1. 大对象直接进入老年代

大对象（需要大量连续内存，如数组、长字符串）容易导致提前触发GC。为避免在新生代区发生大量内存复制（Eden和Survivor的复制），虚拟机提供参数-XX:PretenureSizeThreshold令大于此设置值的对象直接在老年代分配。

1. 长期存活的对象将进入老年代

为标识对象的年代，虚拟机定义了**对象年龄（Age）计数器**：如果对象在Eden出生，经过第一次Minor GC后仍存活且被Survivor容纳，移入Survivor并将对象年龄设为1，在Survivor区中每熬过一次Minor GC年龄加1，年龄到一定程度时（默认15）就进入老年代。年龄阈值设置参数：-XX:MaxTenuringThreshold。

1. 动态对象年龄判定

并非对象年龄达到阈值才晋升老年代，若Survivor中相同年龄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。

1. 空间分配担保

（当Survivor空间不够放上一次存活的对象时，将这些对象直接进入老年代）

* 在发生Minor GC之前，虚拟机检查老年代最大可用连续空间是否大于新生代所有对象的总空间，如果成立，本次Minor GC确保安全；如果不成立，查看是否允许担保失败（HandlePromotionFailure）。
* 如果空间担保允许失败，检查老年代最大可用连续空间是否大于历次晋升到老年代对象的平均大小。如果大于，尝试进行有风险的Minor GC；如果小于或者不允许担保失败，则进行一次Full GC。

1. Full GC的触发条件

Minor GC：当Eden区空间满时触发，频繁、速度快。

Full GC（Major GC）：常伴随至少一次Minor GC，且速度慢10倍以上，

1. 调用System.gc( )

人为触发，最好不用，通过-XX:+ DisableExplicitGC来禁止调用System.gc( )。

1. 老年代空间不足

大对象&长期存活对象导致空间不足，调优时做到让对象在Minor GC阶段被回收、让对象在新生代多存活一段时间、少创建过大对象。

1. 空间分配担保失败
2. JDK1.7以前的永久代空间不足

1.7以前方法区用永久代实现，空间被占满触发Full GC，用增大空间或者转为使用CMS GC来解决。1.8用元空间（本地内存）作为方法区实现，减少触发。

1. Concurrent Mode Failure

执行CMS GC过程中有对象要放入老年代，空间不足报错并触发Full GC。