1.Java 中是值传递还是引用传递？

但是传引用的**错觉**是如何造成的呢？在运行栈中，基本类型和引用的处理是一样的，都是传值，所以，如果是传引用的方法调用，也同时可以理解为“传引用值”的传值调用，即引用的处理跟基本类型是完全一样的。但是当进入被调用方法时，**被传递的这个引用的值，被程序解释（或者查找）到堆中的对象，这个时候才对应到真正的对象**。如果此时进行修改，修改的是引用对应的对象，而不是引用本身，即：修改的是堆中的数据。所以这个修改是可以保持的了。

对象，从某种意义上说，是由基本类型组成的。可以把一个对象看作为一棵树，对象的属性如果还是对象，则还是一颗树（即非叶子节点），基本类型则为树的叶子节点。程序参数传递时，被传递的值本身都是不能进行修改的，但是，如果这个值是一个非叶子节点（即一个对象引用），则可以修改这个节点下面的所有内容。

2.引用类型

对象引用类型分为**强引用、软引用、弱引用和虚引用**。

强引用:就是我们一般声明对象是时虚拟机生成的引用，强引用环境下，垃圾回收时需要严格判断当前对象是否被强引用，如果被强引用，则不会被垃圾回收

软引用:软引用一般被做为缓存来使用。与强引用的区别是，软引用在垃圾回收时，虚拟机会根据当前系统的剩余内存来决定是否对软引用进行回收。如果剩余内存比较紧张，则虚拟机会回收软引用所引用的空间；如果剩余内存相对富裕，则不会进行回收。换句话说，虚拟机在发生 OutOfMemory 时，肯定是没有软引用存在的。

弱引用:弱引用与软引用类似，都是作为缓存来使用。但与软引用不同，弱引用在进行垃圾回收时，是一定会被回收掉的，因此其生命周期只存在于一个垃圾回收周期内。

强引用不用说，我们系统一般在使用时都是用的强引用。而“软引用”和“弱引用”比较少见。他们一般被作为缓存使用，而且一般是在内存大小比较受限的情况下做为缓存。因为如果内存足够大的话，可以直接使用强引用作为缓存即可，同时可控性更高。因而，他们常见的是被使用在桌面应用系统的缓存。

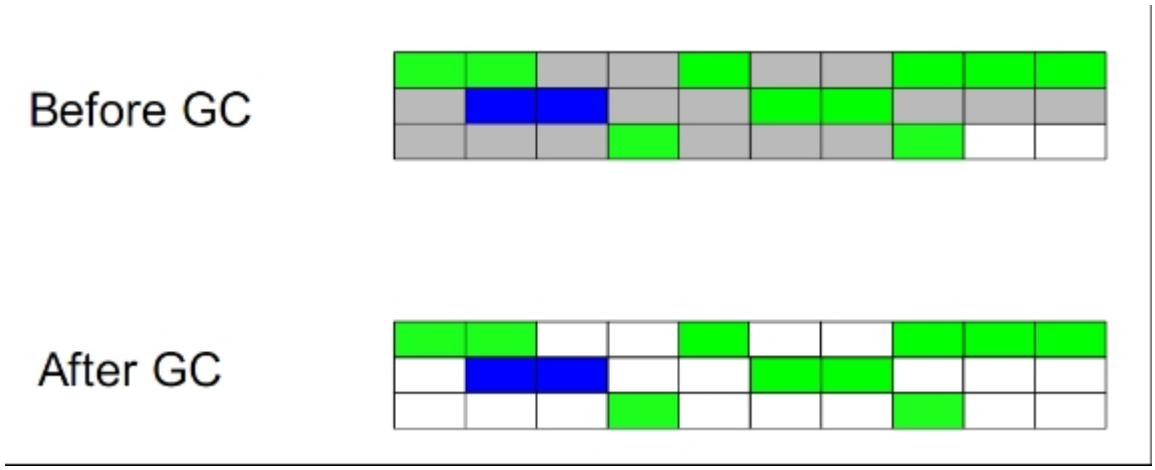
3.基本垃圾回收算法

3.1 按照基本回收策略分

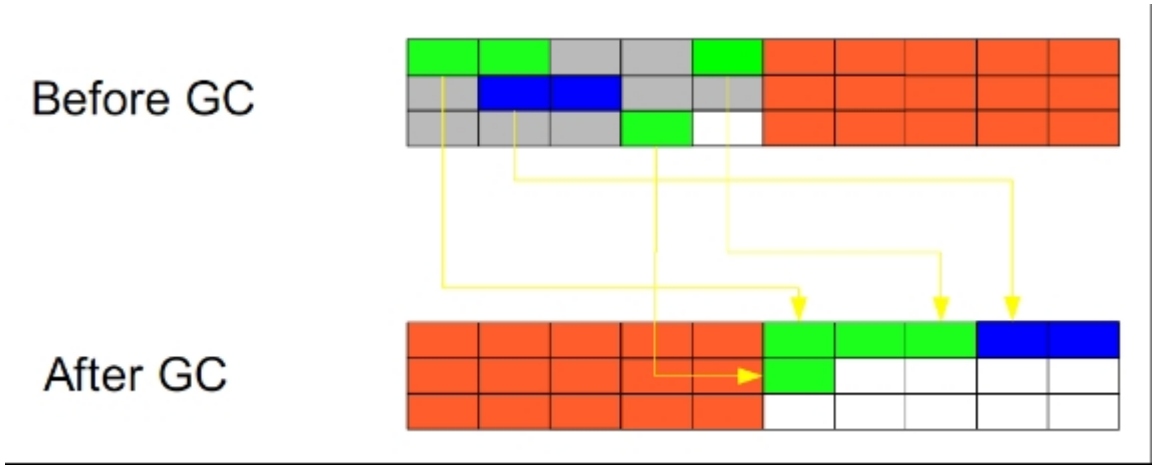
3.1.1 引用计数（Reference Counting）:

比较古老的回收算法。原理是此对象有一个引用，即增加一个计数，删除一个引用则减少一 个计数。垃圾回收时，只用收集计数为 0 的对象。此算法最致命的是无法处理循环引用的问题。

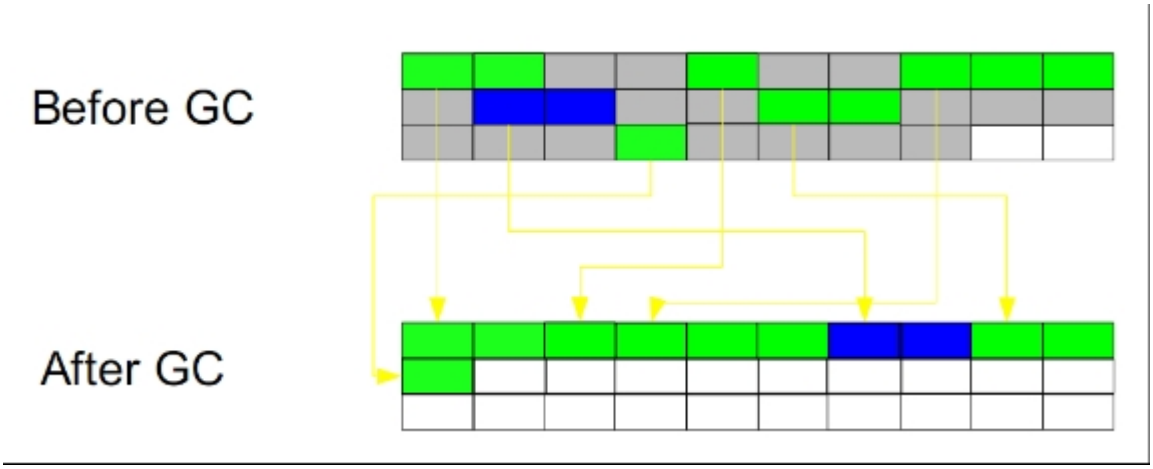
3.1.2 可达性分析清理



标记-清除（Mark-Sweep）:此算法执行分两阶段。第一阶段从**引用根节点开始标记**所有被引用的对象，第二阶段**遍历整个堆**，把未标记的**对象清除**。此算法需要**暂停**整个**应用**，同时，会**产生内存碎片**。



复制（Copying）：此算法把内存空间**划为两个相等**的区域，每次**只**使**用**其中**一个**区域。垃圾回收时，遍历当前使用区域，**把正在使用中**的对象**复制**到另外**一个区域**中。次算法每次**只处理正在使用**中的**对象**，因此复制成本比较小，同时复制过去以后还能进行相应的**内存整理**，不会**出现“碎片”问题。**当然，此算法的缺点也是很明显的，就是**需要两倍内存空间**。



标记-整理（Mark-Compact）:此算法**结合**了“标记**-清除**”和“**复制**”两个算法的**优点**。也是分两阶段，第一阶段从根节点开始**标记所有被引用对象**，第二阶段**遍历整个堆，清除标记对象，并**未标记对象并且**把存活对象“压缩”到堆的其中一块**，按**顺序排放**。此算法避免了“标记-清除”的碎片问题，同时也避免了“复制”算法的空间问题。

3.2 按分区对待的方式分

3.2.1 增量收集（Incremental Collecting）:实时垃圾回收算法，即：在应用进行的同时进行垃圾回收。不知道什么原因 JDK5.0 中的收集器没有使用这种算法的。

3.2.2 分代收集（Generational Collecting）:基于对对象生命周期分析后得出的垃圾回收算法。把对象分为年青代、年老代、持久代，对不同生命周期的对象使用不同的算法（上述方式中的一个）进行回收。现在的垃圾回收器（从 J2SE1.2 开始）都是使用此算法的。

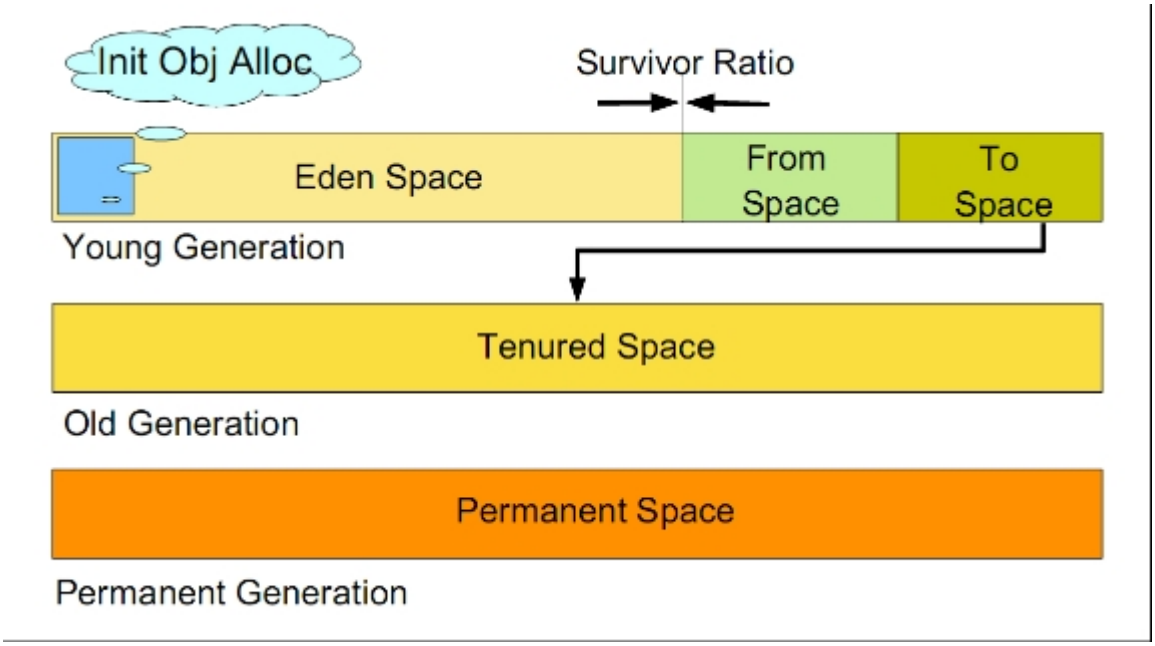
3.3 按系统线程分

3.3.1 串行收集:串行收集使用单线程处理所有垃圾回收工作，因为无需多线程交互，实现容易，而且效率比较高。但是，其局限性也比较明显，即无法使用多处理器的优势，所以此收集适合单处理器机器。当然，此收集器也可以用在小数据量（100M 左右）情况下的多处理器机器上。

3.3.2 并行收集:并行收集使用多线程处理垃圾回收工作，因而速度快，效率高。而且理论上 CPU 数目越多，越能体现出并行收集器的优势。

3.3.3 并发收集:相对于串行收集和并行收集而言，前面两个在进行垃圾回收工作时，需要暂停整个运行环境，而只有垃圾回收程序在运行，因此，系统在垃圾回收时会有明显的暂停，而且暂停时间会因为堆越大而越长。

4.分代处理垃圾



试想，在不进行对象存活时间区分的情况下，每次垃圾回收都是对整个堆空间进行回收，花费时间相对会长，同时，因为每次回收都需要遍历所有存活对象，但实际上，对于生命周期长的对象而言，这种遍历是没有效果的，因为可能进行了很多次遍历，但是他们依旧存在。因此，分代垃圾回收采用分治的思想，进行代的划分，把不同生命周期的对象放在不同代上，不同代上采用最适合它的垃圾回收方式进行回收。

虚拟机中的共划分为三个代：**年轻代**（Young Generation）、**年老点**（Old Generation）和**持久代**（Permanent Generation）。其中持久代主要存放的是 Java 类的类信息，与垃圾收集要收集的 Java 对象关系不大。年轻代和年老代的划分是对垃圾收集影响比较大的。

年轻代:所有新生成的对象首先都是放在年轻代的。年轻代的目标就是尽可能快速的收集掉那些生命周期短的对象。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，当这个 Survivor 去也满了的时候，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“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 去过来的对象。而且，Survivor 区总有一个是空的。同时，根据程序需要，Survivor 区是可以配置为多个的（多于两个），这样可以增加对象在年轻代中的存在时间，减少被放到年老代的可能。

年老代: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。

持久代:用于存放静态文件，如今 Java 类、方法等。持久代对垃圾回收没有显著影响，但是有些应用可能动态生成或者调用一些 class，例如 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。持久代大小通过-XX:MaxPermSize=<N>进行设置。

5.JAVA 中垃圾回收 GC 的类型

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC 有两种类型：**Scavenge GC** 和 **Full GC。**

Scavenge GC： 一般情况下，当新对象生成，并且在 **Eden** 申请空间失败时，就会触发 **Scavenge GC**，对 **Eden 区域进行 GC**，清除非存活对象，并且把尚且存活的对象移动到 Survivor 区。然后整理 Survivor 的两个区。这种方式的 GC 是对年轻代的 Eden 区进行，不会影响到年老代。因为大部分对象都是从 Eden 区开始的，同时 Eden 区不会分配的很大，所以 Eden 区的 GC 会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使 Eden 去能尽快空闲出来。

Full GC： 对整个堆进行整理，包括 Young、Tenured 和 Perm。Full GC 因为需要对整个对进行回收，所以比 Scavenge GC 要慢，因此应该尽可能减少 Full GC 的次数。在对 JVM 调优的过程中，很大一部分工作就是对于 FullGC 的调节。有如下**原因可能导致 Full GC**：· 年老代（Tenured）被写满、持久代（Perm）被写满、System.gc()被显示调用 、上一次 GC 之后 Heap 的各域分配策略动态变化。