JAVA的垃圾回收机制

 java7之前，方法区位于永久代(PermGen)，永久代和堆相互隔离，永久代的大小在启动JVM时可以设置一个固定值，不可变； java7中，**static变量从永久代移到堆中；** java8中，取消永久代，方法存放于元空间(Metaspace)，元空间仍然与堆不相连，但与堆共享物理内存，逻辑上可认为在堆中

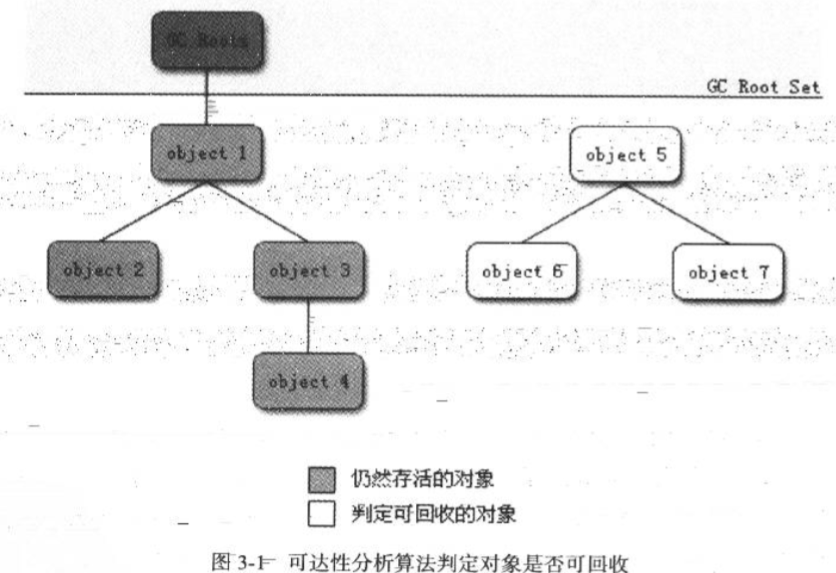
GC通过确定对象是否被活动对象引用来确定是否收集该对象。

JVM内存由几个部分组成：堆、方法区、栈、程序计数器、本地方法栈  
    JVM垃圾回收仅针对公共内存区域，即：堆和方法区进行，因为只有这两个区域在运行时才能知道需要创建些对象，其内存分配和回收都是动态的。

**哪些对象需要回收的对象?**

在进行垃圾回收前，需要知道哪些对象已经没用可以回收了。在objc中，使用ARC自动引用计数，当一个对象ARC为降为0时，认为该对象不在使用。引用技术算法容易出现两个对象循环引用，导致无法正确回收的问题。

在Java中，使用**可达性分析技术**实现。算法的基本思路是：通过**一系列成为”GC Roots”的对象**作为起始点，从这些节点向下搜索，搜索所走过的路称为引用链。如果一个对象到GC Roots不存在任何可达路径，则认为该对象不可用。



在Java中，GC Roots包括：

* **栈帧中局部变量表的引用对象**
* 方法区静态字段引用对象
* 方法区常量引用对象
* 本地方法栈JNI Native方法引用对象

**1.2垃圾回收**  
        要执行垃圾回收，关键在于两点，一是检测垃圾对象，二是释放垃圾对象所占用的空间。  
        1.检测垃圾对象  
            1）引用计数法：（不用的方法）  
                对于一个对象A，只要有任何一个对象引用了A，则A的引用计数器就加1，当引用失效时，引用计数器就减1.  
                只要对象A的引用计数器的值为0，则对象A 就不可能再被使用。  
                实现时只需要为每个对象配置一个整型的计数器即可。  
                但是引用计数器有一个严重的问题，即**无法处理循环引用的情况。**  
                对象A中含有对象B的引用，对象B中含有对象A的引用。此时，对象A和B的引用计数器都不为0.但是在系统中却不存在任何第3个对象引用了A或B。  
                即，A和B是应该被回收的垃圾对象，但由于垃圾对象间相互引用，从而垃圾回收器无法识别**，引起内存泄漏**。  
       2）可达性分析：  
                引用计数法无法检测对象之间相互循环引用的问题，所以基本不用。垃圾回收中检测垃圾对象主要还是“可达性分析”法。  
                可达性分析算法：  
                    通过一系列名为“GC Root”的对象作为起点，从这些节点向下搜索，搜索所走过的路径称为引用链（Reference Chain），  
                    当一个对象到GC Root没有任何引用链相连时，则该对象不可达，该对象是不可使用的，垃圾收集器将回收其所占的内存。  
                    所以JVM判断对象需要存活的原则是：能够被一个根对象到达的对象。  
                    可达：对象A引用了对象B，即A到B可达。  
                      
                GC Root对象集合：  
                    a：Java虚拟机栈（栈帧中的本地变量表）中的引用对象。（当前栈帧的对象引用）  
                    b：方法区中的类静态属性引用的对象。（static对象引用）  
                    c：方法区中的常量引用的对象。（final对象引用）  
                    d：本地方法栈中JNI本地方法的引用对象

        除了堆之外，**方法区**中的**“废弃常量”和“无用的类”**需要回收以保证永久代不会发生内存溢出，检测方法区垃圾对象的方法：  
                    1、判断废弃常量的方法（不再需要的常量）：

如果常量池中的某个常量没有被任何引用所引用，则该常量是废弃常量  
                    2、判断无用的类（**不再需要的class文件**）：  
       1）该类的所有实例都已经被回收，即Java堆中不存在该类的实例对象  
                        2）加载该类的类加载器已经被回收

   3）该类所对应的java.lang.Class对象没有任何地方被引用，无法在任何地方通过反射机制访问该类的方法

 当持久代（方法区）满时，将触发Full GC，根据以上标准**清除废弃的常量和无用的类。**

**2.5 常用垃圾收集器**

　　　　1) 标记-清除收集器 Mark-Sweep

　　　　2) 复制收集器        Copying

　　　　3) 标记-压缩收集器 Mark-Compact

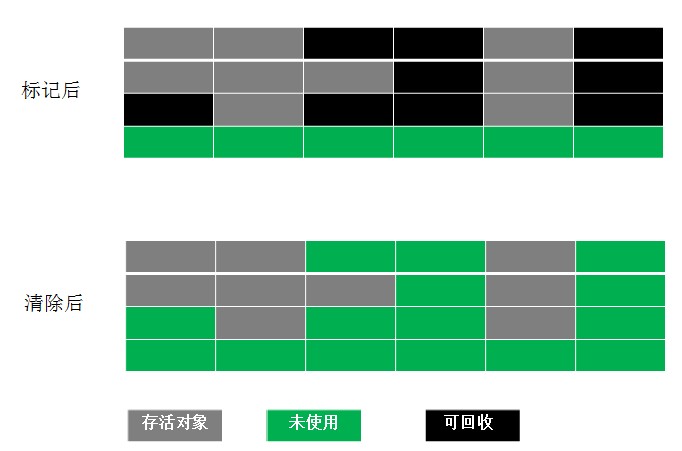
　　　　4) 分代收集器　　　Generational

2.6 垃圾收集算法介绍

　　　　2.6.1 tracing算法

　　　　　　基于tracing算法的垃圾收集也称为标记和清除(mark-and-sweep)垃圾收集器.

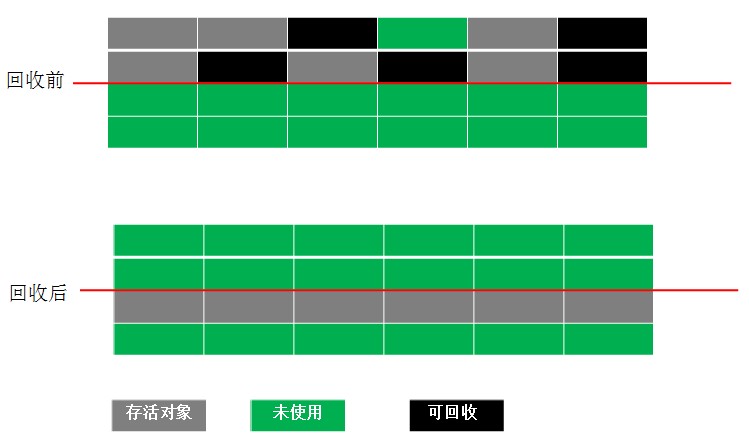
　　　　　　这是最基础的垃圾回收算法，之所以说它是最基础的是因为它最容易实现，思想也是最简单的。标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。具体过程如下图所示：



　　　　　　从图中可以很容易看出标记-清除算法实现起来比较容易，但是有一个比较严重的问题就是容易产生内存碎片**，碎片太多**可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。

　　　　2.6.2 Copying算法

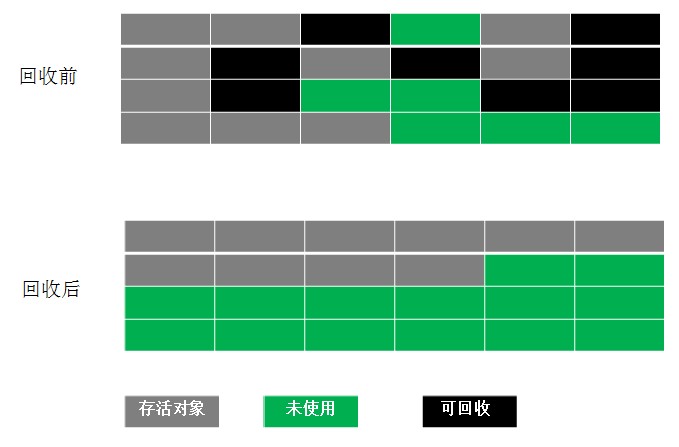
　　　　　　为了解决Mark-Sweep算法的缺陷，Copying算法就被提了出来。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用的内存空间一次清理掉，这样一来就不容易出现内存碎片的问题。具体过程如下图所示：



　　　　　　这种算法虽然实现简单，运行高效且不容易产生内存碎片，但是却对内存空间的使用做出了高昂的代价，因为能够**使用的内存缩减到原来的一半**。很显然，Copying算法的效率跟存活对象的数目多少有很大的关系，如果存活对象很多，那么Copying算法的效率将会大大降低。

　　　　2.6.3 compacting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**。具体过程如下图所示：

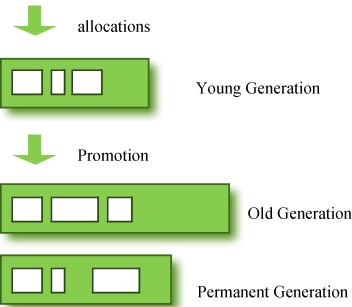


　　　　2.6.4 Generation算法

**分代收集算法是目前大部分JVM的垃圾收集器采用的算法**。它的核心思想是根据对象存活的生命周期将内存划分为若干个不同的区域。一般情况下将堆区划分为老年代（Tenured Generation）和新生代（Young Generation），老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据**不同代的特点采取最适合的收集算法**。

　　　　　　目前大部分垃圾收集器对于**新生代都采取Copying算法**，因为新生代中每次垃圾回收都要回收大部分对象，也就是说需要复制的操作次数较少，但是实际中并不是按照1：1的比例来划分新生代的空间的，一般来说是将新生代划分为**一块较大的Eden空间和两块较小的Survivor空间**，每次使用Eden空间和其中的一块Survivor空间，当进行回收时，将Eden和Survivor中还存活的对象复制到另一块Survivor空间中，然后清理掉Eden和刚才使用过的Survivor空间。

　　　　　　而由于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算法。（标记，放一边，清理掉其它的）**



　　　新年代：新创建的对象都存放在这里。因为大多数对象很快变得不可达，所以大多数对象在年轻代中创建，然后消失。当对象从这块内存区域消失时，我们说发生了一次“**minor GC**”。

　　　　　　老年代：没有变得不可达，存活下来的年轻代对象被复制到这里。这块内存区域一般大于年轻代。因为它更大的规模，GC发生的次数比在年轻代的少。对象从老年代消失时，我们说“**major GC**”（或“**full GC**”）发生了。

　　　　　　上图中的**永久代(permanent generation)**也称为“**方法区(method area)**”，他存储**class对象和字符串常量**。所以这块内存区域**不是永久的存放**从老年代存活下来的对象的。在这块内存中有可能发生垃圾回收。发生在这里垃圾回收也被称为“**major GC**”（或“**full GC**”）。

**Minor GC ，Full GC 触发条件**

**Minor GC触发条件：当Eden区满时，触发Minor GC。**

**Full GC触发条件：**

**（1）调用System.gc时，系统建议执行Full GC，但是不必然执行**

**（2）老年代空间不足**

**（3）方法去空间不足**

**（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存（也就是老年代的空间不足）**

**（5）由Eden区、From Space区向To Space区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小（也就是老年代的空间不足）**

