# JVM垃圾回收机制和垃圾收集器详解

## 一：概述

说起垃圾回收（Garbage Collection，GC），很多人就会自然而然地把它和Java联系起来。在Java中，程序员不需要去关心内存动态分配和垃圾回收的问题，顾名思义，垃圾回收就是释放垃圾占用的空间，这一切都交给了JVM来处理。本文主要解答三个问题：

1、哪些内存需要回收？（对象是否可以被回收的两种经典算法: 引用计数法 和 可达性分析算法）   
2、如何回收？（三种经典垃圾回收算法(标记清除算法、复制算法、标记整理算法)及分代收集算法）   
3、使用什么工具回收？（垃圾收集器）

在探讨Java垃圾回收机制之前，我们首先应该记住一个单词：Stop-the-World。Stop-the-world意味着 JVM由于要执行GC而停止了应用程序的执行，并且这种情形会在任何一种GC算法中发生。当Stop-the-world发生时，除了GC所需的线程以外，所有线程都处于等待状态直到GC任务完成。事实上，GC优化很多时候就是指减少Stop-the-world发生的时间，从而使系统具有高吞吐 、低停顿的特点。

## 二、JVM垃圾判定算法

**常用的垃圾判定算法包括：引用计数算法，可达性分析算法。**

### 1、引用计数算法

引用计数算法是通过判断对象的引用数量来决定对象是否可以被回收。

java中是通过引用来和对象进行关联的，也就是说如果要操作对象，必须通过引用来进行。给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时，计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的，即表示该对象可视为”垃圾“被回收。

引用计数器算法实现简单，效率高；但是不能解决循环引用问问题（A 对象引用B 对象，B 对象又引用A 对象，但是A,B 对象已不被任何其他对象引用），同时每次计数器的增加和减少都带来了很多额外的开销，所以在JDK1.1 之后，这个算法已经不再使用了。代码：

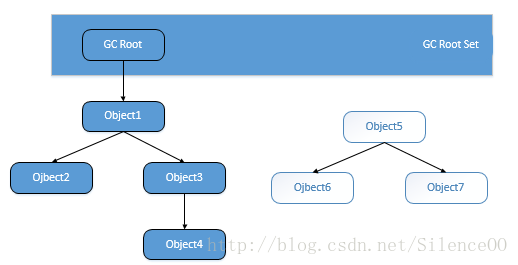
public class Main {
  
 public static void main(String[] args) {
  
 MyTest test1 = new MyTest();
  
 MyTest test2 = new MyTest();
  
   
 test1.obj = test2;
  
 test2.obj = test1;//test1与test2存在相互引用
  
   
 test1 = null;
  
 test2 = null;
  
   
 System.gc();//回收
  
 }
  
}
  
   
class MyTest{
  
 public Object obj = null;
  
}

虽然最后将test1和test2赋值为null，也就是说test1和test2指向的对象已经不可能再被访问，但是由于它们互相引用对方，导致它们的引用计数都不为0，那么垃圾收集器就永远不会回收它们。运行程序，从内存分析看到，事实上这两个对象的内存被回收，这也说明了当前主流的JVM都不是采用的引用计数器算法作为垃圾判定算法的。

### 2、可达性分析算法（根搜索算法）

可达性分析算法是通过判断对象的引用链是否可达来决定对象是否可以被回收。

根搜索算法是通过一些“GC Roots”对象作为起点，从这些节点开始往下搜索，搜索通过的路径成为引用链   
（Reference Chain），当一个对象没有被GC Roots 的引用链连接的时候，说明这个对象是不可用的。



#### GC Roots 对象包括：

##### a) 虚拟机栈（栈帧中的本地变量表）中的引用的对象。

##### b) 方法区域中的类静态属性引用的对象。

##### c) 方法区域中常量引用的对象。

##### d) 本地方法栈中JNI（即一般说的Native方法）的引用的对象。

在可达性分析算法中，不可达的对象，也并非是“非死不可”的，这时候它们暂时处于“缓刑”阶段，要真正宣告一个对象死亡，至少要经历两次标记过程：

如果对象在进行可达性分析后发现没有与GC Roots相连接的引用链，那它将会被第一次标记并且进行一次筛选，筛选的条件是此对象是否需要执行finalize()方法。当对象没有覆盖finalize()方法，或者finalize()方法已经被虚拟机调用过，虚拟机将这两种情况都视为“不需要要执行”。注意任何对象的finalize()方法只会被系统自动执行1次。

如果这个对象被判定为需要执行finalize()方法，那么这个对象将会放置在一个叫做F-Queue的队列之中，并在稍后由一个由虚拟机自动建立的、低优先级的Finalizer线程去执行它。这里所谓的“执行”是指虚拟机会触发这个方法，但并不承诺会等待它运行结束，这样做的原因是，如果一个对象在finalize()方法中执行缓慢，或者发生了死循环，将很可能会导致F-Queue队列中其他对象永久处于等待，甚至导致整个内存回收系统崩溃。因此调用finalize()方法不代表该方法中代码能够完全被执行。

finalize()方法是对象逃脱死亡命运的最后一次机会，稍后GC将对F-Queue中的对象进行第二次小规模的标记，如果对象要在finalize()中成功拯救自己——只要重新与引用链上的任何一个对象建立关联即可，譬如把自己（this关键字）赋值给某个类变量或者对象的成员变量，那在第二次标记时它将被移除出“即将回收”的集合；如果对象这时候还没有逃脱，那基本上它就真的被回收了。从如下代码中我们可以看到一个对象的finalize()被执行，但是它仍然可以存活。

/\*\*
  
 \* 此代码演示了两点：
  
 \* 1.对象可以在被GC时自我拯救。
  
 \* 2.这种自救的机会只有一次，因为一个对象的finalize()方法最多只会被系统自动调用一次
  
 \*/
  
public class FinalizeEscapeGC {
  
   
 public static FinalizeEscapeGC SAVE\_HOOK = null;
  
   
 public void isAlive() {
  
 System.out.println("yes, i am still alive :)");
  
 }
  
   
 @Override
  
 protected void finalize() throws Throwable {
  
 super.finalize();
  
 System.out.println("finalize mehtod executed!");
  
 FinalizeEscapeGC.SAVE\_HOOK = this;
  
 }
  
   
 public static void main(String[] args) throws Throwable {
  
 SAVE\_HOOK = new FinalizeEscapeGC();
  
   
 //对象第一次成功拯救自己
  
 SAVE\_HOOK = null;
  
 System.gc();
  
 //因为finalize方法优先级很低，所以暂停0.5秒以等待它
  
 Thread.sleep(500);
  
 if (SAVE\_HOOK != null) {
  
 SAVE\_HOOK.isAlive();
  
 } else {
  
 System.out.println("no, i am dead :(");
  
 }
  
   
 //下面这段代码与上面的完全相同，但是这次自救却失败了
  
 SAVE\_HOOK = null;
  
 System.gc();
  
 //因为finalize方法优先级很低，所以暂停0.5秒以等待它
  
 Thread.sleep(500);
  
 if (SAVE\_HOOK != null) {
  
 SAVE\_HOOK.isAlive();
  
 } else {
  
 System.out.println("no, i am dead :(");
  
 }
  
 }
  
}

运行结果：

finalize mehtod executed!
  
yes, i am still alive :)
  
no, i am dead :(

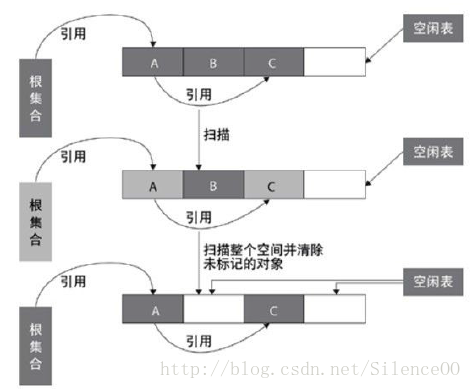
从运行结果可以看出，SAVE\_HOOK对象的finalize()方法确实被GC收集器调用过，且在被收集前成功逃脱了。   
另外一个值得注意的地方是，代码中有两段完全一样的代码片段，执行结果却是一次逃脱成功，一次失败，这是因为任何一个对象的finalize()方法都只会被系统自动调用一次，如果对象面临下一次回收，它的finalize()方法不会被再次执行，因此第二段代码的自救行动失败了。

## 三、JVM垃圾回收算法

常用的垃圾回收算法包括：标记-清除算法，复制算法，标记-整理算法，分代收集算法

### 1、标记—清除算法（Mark-Sweep）（DVM 使用的算法）

标记—清除算法包括两个阶段：“标记”和“清除”。在标记阶段，确定所有要回收的对象，并做标记。清除阶段紧随标记阶段，将标记阶段确定不可用的对象清除。标记—清除算法是基础的收集算法。



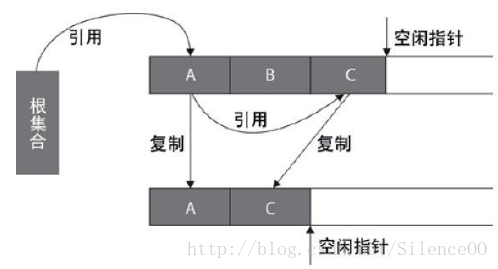
标记-清除算法的缺点有两个：

效率问题：标记和清除两个过程的效率都不高;

空间问题：标记-清除算法不需要进行对象的移动，并且仅对不存活的对象进行处理，因此标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致以后在程序运行过程中需要分配较大对象时，无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

### 2、复制算法（Copying）

复制算法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，然后再把已使用过的内存空间一次清理掉。这种算法适用于对象存活率低的场景，比如新生代。这样使得每次都是对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。



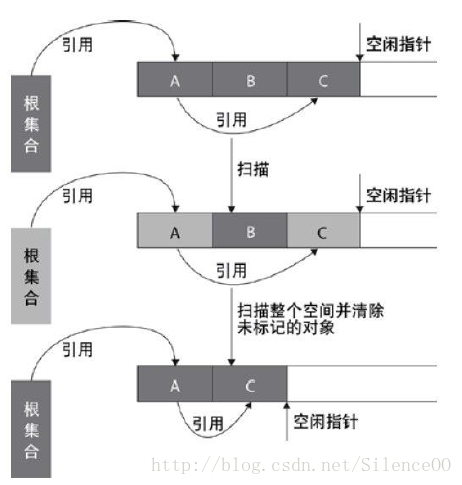
事实上，现在商用的虚拟机都采用这种算法来回收新生代。因为研究发现，新生代中的对象每次回收都基本上只有10%左右的对象存活，所以需要复制的对象很少，效率还不错。

实践中会将新生代内存分为一块较大的Eden空间和两块较小的Survivor空间 (如下图所示)，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地复制到另外一块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空间为整个新生代容量的90% ( 80%+10% )，只有10% 的内存会被“浪费”。

复制收集算法在对象存活率较高时就要进行较多的复制操作，效率将会变低。更关键的是，如果不想浪费50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

### 3、标记—整理算法（Mark-Compact）

标记—整理算法和标记—清除算法一样，但是标记—整理算法不是把存活对象复制到另一块内存，而是把存活对象往内存的一端移动，然后直接回收边界以外的内存。标记—整理算法提高了内存的利用率，并且它适合在收集对象存活时间较长的老年代。



标记整理算法与标记清除算法最显著的区别是：标记清除算法不进行对象的移动，并且仅对不存活的对象进行处理；而标记整理算法会将所有的存活对象移动到一端，并对不存活对象进行处理，因此其不会产生内存碎片。

### 4、分代收集（Generational Collection）

分代收集是基于这样一种事实：不同的对象的生命周期(存活情况)是不一样的，而不同生命周期的对象位于堆中不同的区域，因此对堆内存不同区域采用不同的策略进行回收可以提高 JVM 的执行效率。

当代商用虚拟机使用的都是分代收集算法：新生代对象存活率低，就采用复制算法，只需要付出少量存活对象的复制成本就可以完成收集；老年代存活率高，没有额外空间对它进行分配担保，就用“标记—清理”或者“标记—整理”算法。Java堆内存一般可以分为新生代、老年代和永久代三个模块。

#### (1). 新生代（Young Generation）

新生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，一般情况下，所有新生成的对象首先都是放在新生代的。新生代内存按照 8:1:1 的比例分为一个eden区和两个survivor(survivor0，survivor1)区，大部分对象在Eden区中生成。在进行垃圾回收时，先将eden区存活对象复制到survivor0区，然后清空eden区，当这个survivor0区也满了时，则将eden区和survivor0区存活对象复制到survivor1区，然后清空eden和这个survivor0区，此时survivor0区是空的，然后交换survivor0区和survivor1区的角色（即下次垃圾回收时会扫描Eden区和survivor1区），即保持survivor0区为空，如此往复。特别地，当survivor1区也不足以存放eden区和survivor0区的存活对象时，就将存活对象直接存放到老年代。如果老年代也满了，就会触发一次FullGC，也就是新生代、老年代都进行回收。注意，新生代发生的GC也叫做MinorGC，MinorGC发生频率比较高，不一定等 Eden区满了才触发。

#### (2). 老年代（Old Generation）

老年代存放的都是一些生命周期较长的对象，就像上面所叙述的那样，在新生代中经历了N次垃圾回收后仍然存活的对象就会被放到老年代中。此外，老年代的内存也比新生代大很多(大概比例是1:2)，当老年代满时会触发Major GC(Full GC)，老年代对象存活时间比较长，因此FullGC发生的频率比较低。

#### (3). 永久代（Permanent Generation）

永久代主要用于存放静态文件，如Java类、方法等。永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，例如使用反射、动态代理、CGLib等bytecode框架时，在这种时候需要设置一个比较大的永久代空间来存放这些运行过程中新增的类。

#### 垃圾回收有两种类型，Minor GC 和 Full GC。

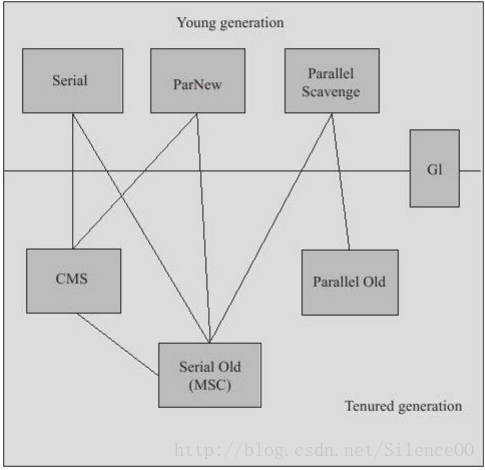
Minor GC：对新生代进行回收，不会影响到年老代。因为新生代的 Java 对象大多死亡频繁，所以 Minor GC 非常频繁，一般在这里使用速度快、效率高的算法，使垃圾回收能尽快完成。

Full GC：也叫 Major GC，对整个堆进行回收，包括新生代和老年代。由于Full GC需要对整个堆进行回收，所以比Minor GC要慢，因此应该尽可能减少Full GC的次数，导致Full GC的原因包括：老年代被写满、永久代（Perm）被写满和System.gc()被显式调用等。

## 四、垃圾收集器

如果说垃圾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。

上面说过，各个平台虚拟机对内存的操作各不相同，因此本章所讲的收集器是基于JDK1.7Update14之后的HotSpot虚拟机。这个虚拟机包含的所有收集器如图：

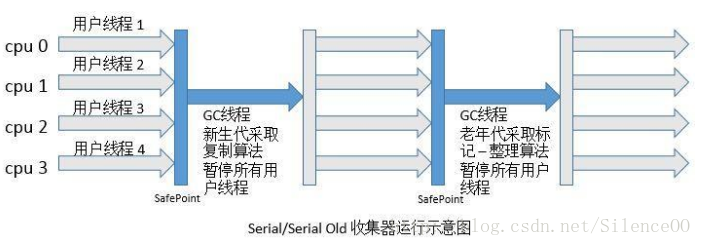


其中用于回收新生代的收集器包括Serial、PraNew、Parallel Scavenge，回收老年代的收集器包括Serial Old、Parallel Old、CMS，还有用于回收整个Java堆的G1收集器。

### 1、Serial收集器（复制算法)

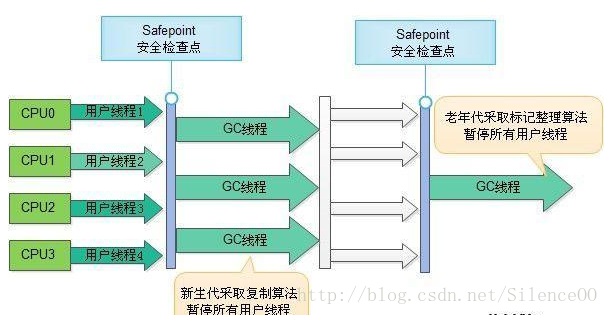
新生代单线程收集器，优点是简单高效。Serial收集器是最基本、发展历史最悠久的收集器，曾经（在JDK 1.3.1之前）是虚拟机新生代收集的唯一选择。

大家看名字就会知道，这个收集器是一个单线程的收集器，但它的“单线程”的意义并不仅仅说明它只会使用一个CPU或一条收集线程去完成垃圾收集工作，更重要的是在它进行垃圾收集时，必须暂停其他所有的工作线程，直到它收集结束。



### 2、ParNew收集器 (复制算法)

新生代并行收集器，ParNew收集器其实就是Serial收集器的多线程版本，除了使用多条线程进行垃圾收集之外，其余行为包括Serial收集器可用的所有控制参数（例如：-XX：SurvivorRatio、-XX：PretenureSizeThreshold、-XX：HandlePromotionFailure等）、收集算法、Stop The World、对象分配规则、回收策略等都与Serial收集器完全一样，在实现上，这两种收集器也共用了相当多的代码。



### 3、Parallel Scavenge收集器(复制算法)

新生代并行收集器，追求高吞吐量，高效利用 CPU。



Parallel Scavenge收集器的特点是它的关注点与其他收集器不同，Parallel Scavenge收集器的目标是达到一个可控制的吞吐量（Throughput）。所谓吞吐量就是CPU用于运行用户代码的时间与CPU总消耗时间的比值，即吞吐量=运行用户代码时间/（运行用户代码时间+垃圾收集时间）。由于与吞吐量关系密切，Parallel Scavenge收集器也经常称为“吞吐量优先”收集器。

### 4、Serial Old收集器(标记-整理算法)

老年代单线程收集器，Serial Old是Serial收集器的老年代版本，它同样是一个单线程收集器，使用“标记-整理”算法。这个收集器的主要意义也是在于给Client模式下的虚拟机使用。如果在Server模式下，那么它主要还有两大用途：一种用途是在JDK 1.5以及之前的版本中与Parallel Scavenge收集器搭配使用[1]，另一种用途就是作为CMS收集器的后备预案，在并发收集发生ConcurrentMode Failure时使用。

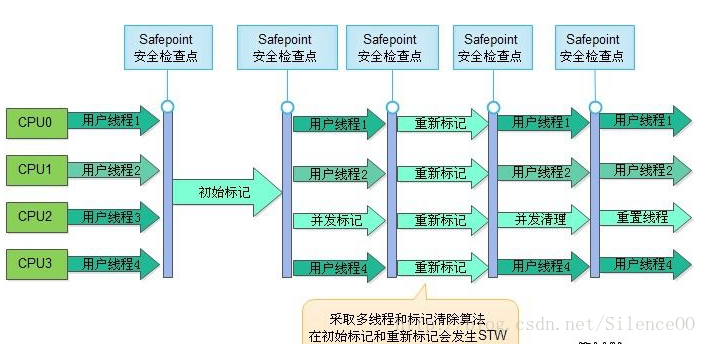
### 5、Parallel Old收集器 (标记-整理算法)

老年代并行收集器，吞吐量优先

Parallel Old是Parallel Scavenge收集器的老年代版本，使用多线程和“标记-整理”算法。这个收集器是在JDK 1.6中才开始提供的。

### 6、CMS收集器（标记-清除算法）

老年代并行收集器，以获取最短回收停顿时间为目标的收集器，具有高并发、低停顿的特点，追求最短GC回收停顿时间。



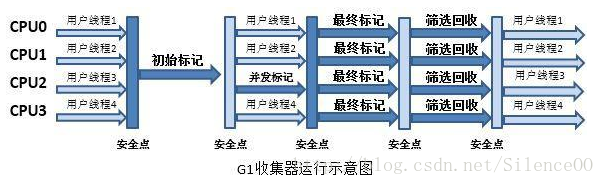
CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。

目前很大一部分的Java应用集中在互联网站或者B/S系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。   
运作过程分为4个步骤，包括：   
a)初始标记（CMS initial mark）   
b)并发标记（CMS concurrent mark）   
c)重新标记（CMS remark）   
d)并发清除（CMS concurrent sweep）

CMS收集器存在3个缺点：   
1 对CPU资源敏感。一般并发执行的程序对CPU数量都是比较敏感的   
2 无法处理浮动垃圾。在并发清理阶段用户线程还在执行，这时产生的垃圾无法清理。   
3 由于标记-清除算法产生大量的空间碎片。

### 7、G1收集器 (标记-整理算法)

Java堆并行收集器，G1收集器是JDK1.7提供的一个新收集器，G1收集器基于“标记-整理”算法实现，也就是说不会产生内存碎片。此外，G1收集器不同于之前的收集器的一个重要特点是：G1回收的范围是整个Java堆(包括新生代，老年代)，而前六种收集器回收的范围仅限于新生代或老年代。



G1是一款面向服务端应用的垃圾收集器。   
G1收集器的运作大致可划分为以下几个步骤：

a)初始标记（Initial Marking）   
b)并发标记（Concurrent Marking）   
c)最终标记（Final Marking）   
d)筛选回收（Live Data Counting and Evacuation）