# **[深入探究jvm之GC的算法及种类](https://www.cnblogs.com/liuyk-code/p/10289949.html)**

https://www.cnblogs.com/liuyk-code/p/10289949.html

一、GC基本概念

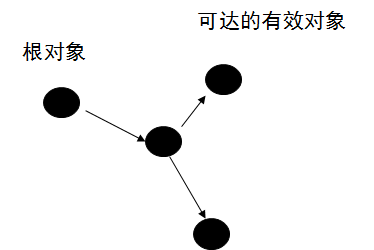
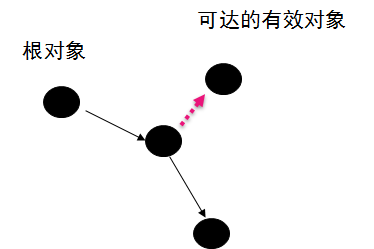
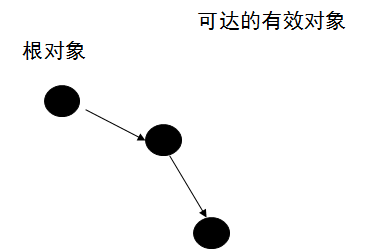
　　GC(Garbage Collection)垃圾收集，1960年最早在List中使用。在Java中GC回收的对象是堆空间和永久区，可以有效避免程序员人为造成内存泄漏问题。将堆空间和永久区没有作用的对象进行释放和回收。

二、GC算法

1、引用计数法：

　　是一种老牌的垃圾回收算法，通过引用计算来回收垃圾，被COM、ActionScript3、Python所使用。

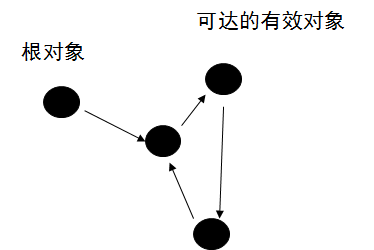
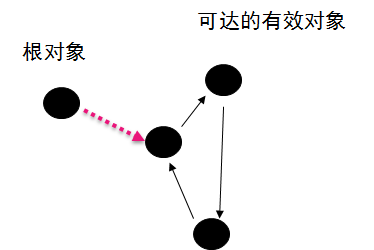
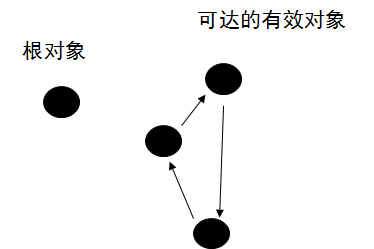
　　引用计数法的实现很简单，对于一个对象A，只要有任何一个对象引用了A，那么A的引用计数器就会+1，当引用失效时，引用计数器就会-1。只要对象A的引用计数器的值为0，那么对象A 就不可能再被使用。

IMG_257IMG_259

　　引用计数法存在的问题：

　　1）引用和去引用伴随着加法，程序运行时随时都发生着引用和去引用，影响性能；

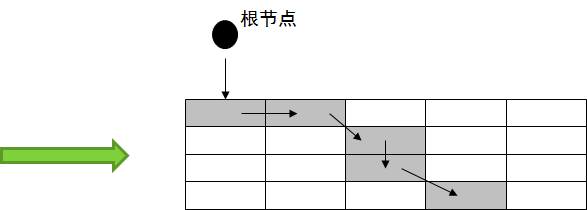
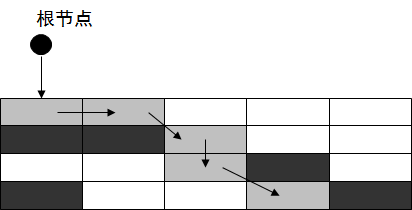
　　2）很难处理循环引用的问题，如下图：

IMG_262IMG_264

 　　中间节点未被根节点引用，但经过一次循环后引用计数为2，仍然不会被清除，实际上应该被清除。

2、标记-清除法：

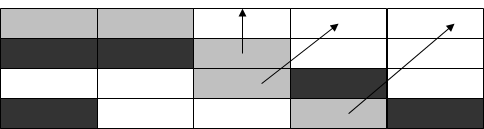
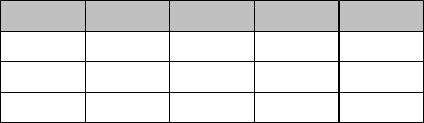
　　标记-清除算法是现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此未被标记的对象就是未被引用的垃圾对象。然后在清除阶段，清除所有未被标记的对象。



 　　灰色对象都是根节点的可达对象，黑色对象未被根节点引用（直接或间接），白色部分为空闲空间。清理阶段会将黑色对象（未被标记）清理掉。

3、标记-压缩法：

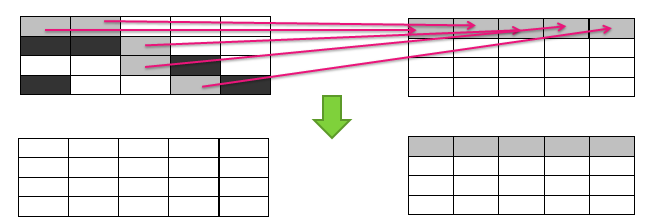
　　标记-压缩算法适合存活对象比较多的场合，如老年代。它在标记-清除算法的基础上做了一些优化。和标记-清除算法一样，标记-压缩算法也首先需要从根节点开始，对所有可达对象做一次标记，但之后，它并不仅仅简单的清理未标记的对象，而是将所有存活的对象压缩到内存的一端。之后清理边界外所有的对象。如下图：

IMG_269

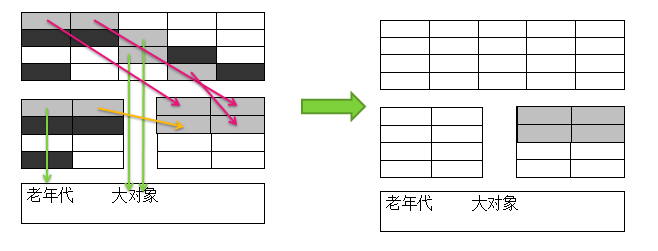
4、复制算法

　　与标记-清除算法相比，复制算法是一种相对高效的回收算法，但不适用于存活对象较多的场合，如老年代。

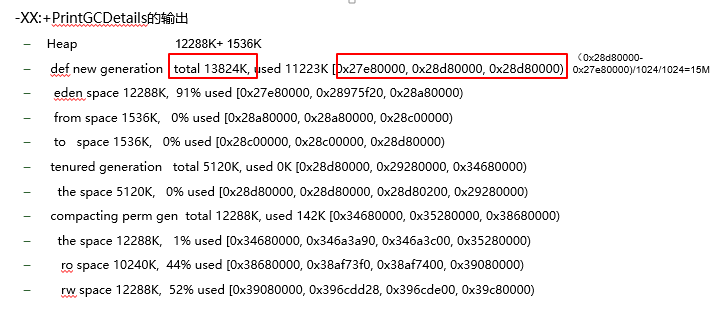
　　它主要实现方案是将原有的内存空间分为两块，每次只使用其中的一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未被使用的内存块中，之后，清除正在使用的内存块中所有对象，交换两个内存块的角色，完成垃圾回收。



　　由此可见，复制算法最大的问题是空间浪费。Java中实际应用做了一些优化（分代思想，下面介绍），示例图如下：



　　新生代总空间为15M，但可用空间只有13824K。其中12288K为eden区空间，主要存放新产生的对象，1536K为新生代两块相同空间（幸存代中from区和to区）中其中一块。



三、分代思想：

　　1、依据对象的存活周期进行分类，短命对象归为新生代，长命对象归为老年代；

　　2、根据不同代的特点，选取合适的GC算法，少量对象存活，适合复制算法；大量对象存活，适合标记清理或者标记压缩。

 四、可触及性：

　　所有的算法需要能够识别一个垃圾对象，因此引入了可触及性的概念。

　　1、可触及的：从根节点可以触及到的对象；

　　2、可复活的：一旦所有引用都被释放，就是可复活状态；因为有可能在finalize()方法中可能复活该对象；

　　3、不可触及的：在finalize()方法后，可能会进入不可触及状态，不可触及的对象不可能被复活，此时可以被GC回收。

　　对象状态转换如下：

　　1）首先定义一个可复活对象，在finalize()方法中复活一个对象：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | public class CanReliveObj {      public static CanReliveObj obj;      @Override      protected void finalize() throws Throwable {          super.finalize();          System.out.println("CanReliveObj finalize called");          obj=this;      }      @Override      public String toString(){          return "I am CanReliveObj";      }<br>} |

　　2）主函数方法如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public static void main(String[] args) throws       InterruptedException{  　　obj=new CanReliveObj();  　　obj=null;   //可复活  　　System.gc();  　　Thread.sleep(1000);  　　if(obj==null){      　　System.out.println("obj 是 null");  　　}else{      　　System.out.println("obj 可用");  　　}  　　System.out.println("第二次gc");  　　obj=null;    //不可复活  　　System.gc();  　　Thread.sleep(1000);  　　if(obj==null){  　　　　System.out.println("obj 是 null");  　　}else{  　　　　System.out.println("obj 可用");  　　}  } |

　　3）运行结果如下：

|  |  |
| --- | --- |
| 1  2  3  4 | CanReliveObj finalize called  obj 可用  第二次gc  obj 是 null |

　　注意，如果在调用finalize()方法后忘记释放内存，那么可复活对象就会一直存在于堆内存中，很容易造成内存溢出，因此是有风险的。在编码的时候要注意以下几点：

　　1）避免使用finalize()方法，操作不慎可能导致错误；

　　2）优先级很低，因为我们不知道也无法明确GC什么时候发生，使用finalize()方法反而增加了程序的不确定性；

　　3）可以使用try-catch-finally来代替finalize()方法。

五、根对象

　　什么是根对象呢？主要有以下三类：

　　1、栈中引用的对象；

　　2、方法区静态成员或者常量引用的对象（全局变量）；

　　3、JNI方法栈中引用的对象。

六、STOP-THE-WORLD

　　STOP-THE-WORLD是Java中一种全局停顿的现象，此时所有Java代码停止运行，native方法可以运行但是无法与jvm发生交互，发生这种情况多半是由于GC引起的。另外Dump检查、死锁检查、堆Dump也有可能引起。

　　1、GC为什么会引起全局停顿？

　　类比在聚会时打扫卫生，聚会时很乱，又会产生新的垃圾，房间永远不会被打扫干净，只有暂停一下聚会才会将房间打扫干净。

　　2、全局停顿的危害

　　长时间停止服务，没有响应；对于HA系统可能会引起主备切换。

　　3、写一个测试demo验证STOP-THE-WORLD的存在

　　1）声明一个线程，每过1s打印10条记录；

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | public static class PrintThread extends Thread{      public static final long starttime=System.currentTimeMillis();      @Override      public void run(){          try{              while(true){                  long t=System.currentTimeMillis()-starttime;                  System.out.println("time:"+t);                  Thread.sleep(100);              }          }catch(Exception e){            }      }  } |

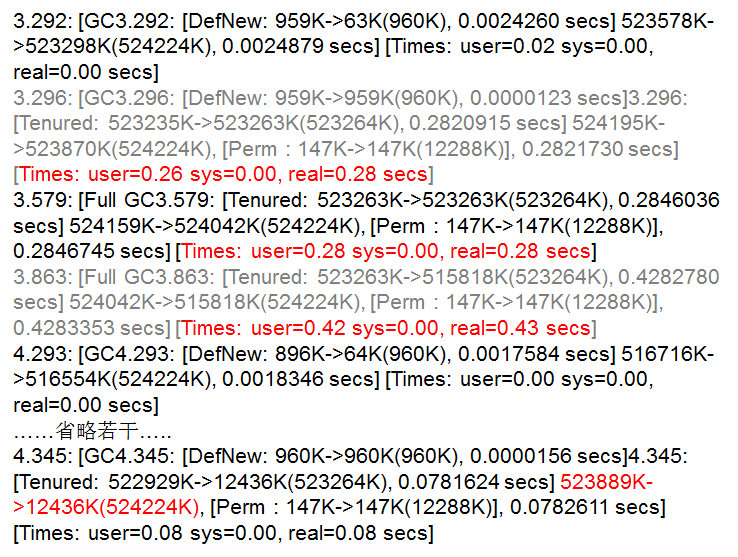
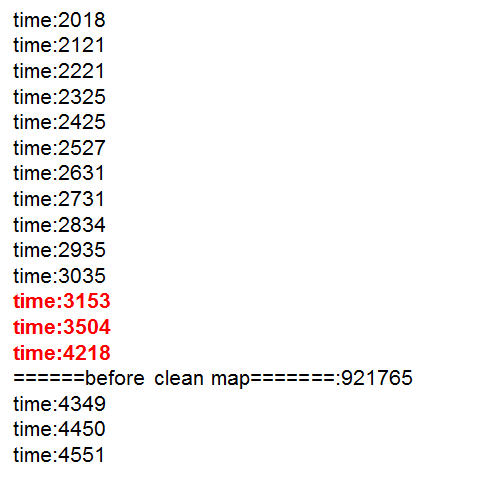
　　2）声明另外一个线程消耗资源，用来触发GC（不断的占用内存，不断的释放变量来触发GC）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21 | public static class MyThread extends Thread{      HashMap<Long,byte[]> map=new HashMap<Long,byte[]>();      @Override      public void run(){          try{              while(true){                  if(map.size()\*512/1024/1024>=450){                      System.out.println(“=====准备清理=====:"+map.size());                      map.clear();                  }                    for(int i=0;i<1024;i++){                      map.put(System.nanoTime(), new byte[512]);                  }                  Thread.sleep(1);              }          }catch(Exception e){              e.printStackTrace();          }      }  } |

　　3）启动JVM的参数为512m堆空间、串行回收器

|  |  |
| --- | --- |
| 1 | -Xmx512M -Xms512M -XX:+UseSerialGC -Xloggc:gc.log -XX:+PrintGCDetails  -Xmn1m -XX:PretenureSizeThreshold=50 -XX:MaxTenuringThreshold=1 |

　　4）观察控制台输出和GC日志，开始是1s打印10条记录，后来产生了全局停顿。



　　GC发生的时间与全局停顿的时间是吻合的。