# 一 GC的概念

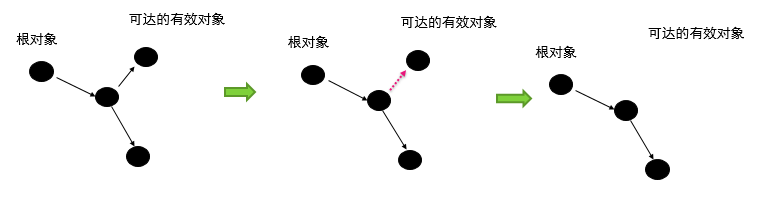
* Garbage Collection 垃圾收集
* 1960年 List 使用了GC
* Java中，GC的对象是堆空间和永久区

# 二 GC算法

## 2.1 引用计数法

* 老牌垃圾回收算法
* 通过引用计算来回收垃圾
* 使用者 COM,ActionScript3,Python

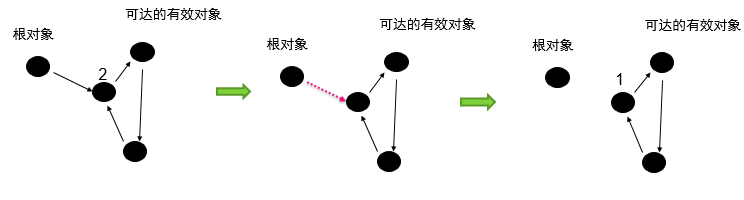
算法描述: 引用计数器的实现很简单，对于一个对象A，只要有任何一个对象引用了A，则A的引用计数器就加1，当引用失效时，引用计数器就减1。只要对象A的引用计数器的值为0，则对象A就不可能再被使用。



* 引用计数法的问题

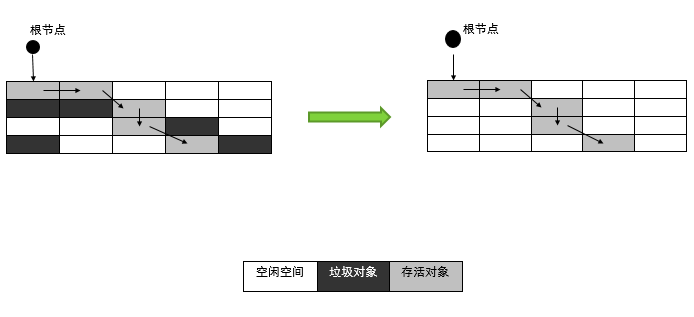
1 引用和去引用伴随加法和减法，影响性能

2 很难处理循环引用



## 2.2 标记清除

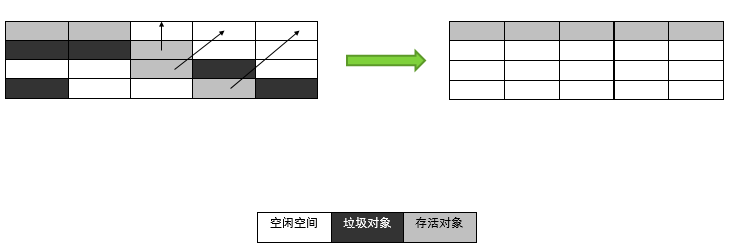
* 标记-清除算法是现代垃圾回收算法的思想基础。标记-清除算法将垃圾回收分为两个阶段：标记阶段和清除阶段。一种可行的实现是，在标记阶段，首先通过根节点，标记所有从根节点开始的可达对象。因此，未被标记的对象就是未被引用的垃圾对象。然后，在清除阶段，清除所有未被标记的对象。



如上图所示，从根节点开始被标记的对象都是存活的对象，未被标记的空白对象我们可以看做是空的对象，黑色部分是没有被标记的垃圾，也就是将要被回收的对象。

## 2.3 标记压缩

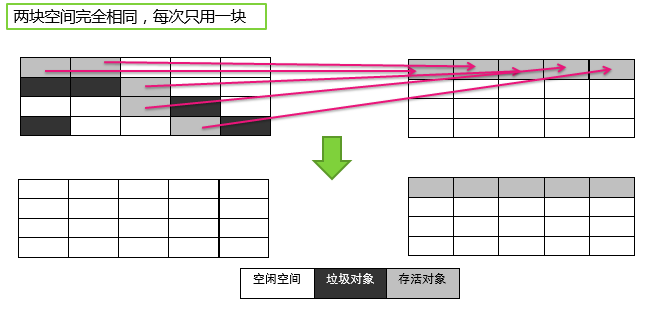
标记-压缩算法适合用于存活对象较多的场合，如老年代。它在标记-清除算法的基础上做了一些优化。和标记-清除算法一样，标记-压缩算法也首先需要从根节点开始，对所有可达对象做一次标记。但之后，它并不简单的清理未标记的对象，而是将所有的存活对象压缩到内存的一端。之后，清理边界外所有的空间。

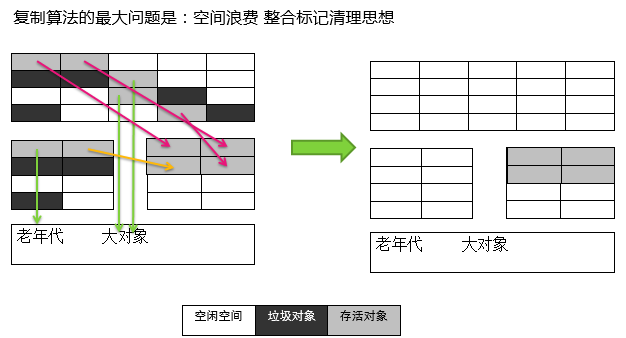


如上图所示，我们把从根节点开始的存活，标记过的对象，压缩到一端以后，再进行垃圾回收。

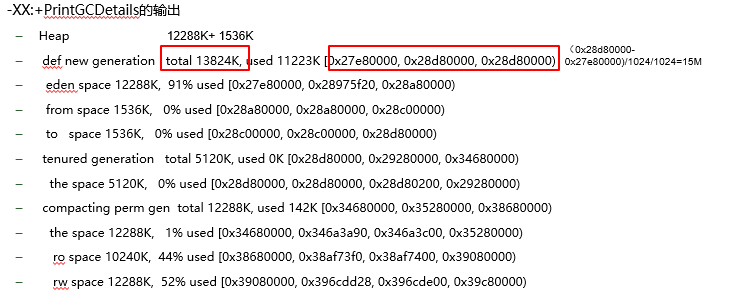
## 2.4 复制算法

与标记-清除算法相比，复制算法是一种相对高效的回收方法，不适用于存活对象较多的场合 如老年代，将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在使用的内存中的存活对象复制到未使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色，完成垃圾回收





如上图所示，由于老年代是比较活跃的对象直接拷贝到s2区不太适合，还有就是大对象直接考过去不利于s2区复制的充分利用，所以首先从eden区把大对象，s1区的幸存者放到老年代，再对eden,s1做回收。



由复制算法可以看出，新生代显示地址是15M，却为什么只有13M,因为s1,s2区是大小相等的，总有一个区是供给给复制算法进行垃圾回收占用的，所以可用的新生代就只能是eden区，与幸存区s1或者s2中的一块。

## 分代思想

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。

根据不同代的特点，选取合适的收集算法

1.少量对象存活，适合复制算法

2.大量对象存活，适合标记清理或者标记压缩

## GC算法总结

* 引用计数

没有被Java采用

* 标记-清除 多用于老年代
* 标记-压缩 多用于老年代
* 复制算法
  + 新生代

# 三 可触及性

所有的算法，需要能够识别一个垃圾对象，因此需要给出一个可触及性的定义。

* 可触及的
  + 从根节点可以触及到这个对象
* 可复活的
  + 一旦所有引用被释放，就是可复活状态
  + 因为在finalize()中可能复活该对象
* 不可触及的
  + 在finalize()后，可能会进入不可触及状态
  + 不可触及的对象不可能复活
  + 可以回收

举例

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";

}

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");

}

}

如上代码段所示，在同一个对象调用第一次GC的时候，可能会由于finalize的方法进行了复活，但是如果说再次调用的时候，清空再被调用的时候可能就无法复活，而进入回收状态了。

经验：避免使用finalize()，操作不慎可能导致错误。

优先级低，何时被调用， 不确定，何时发生GC不确定

可以使用try-catch-finally来替代它.

根

栈中引用的对象

方法区中静态成员或者常量引用的对象（全局对象）

JNI方法栈中引用对象

# 四 Stop-The-World

Java中一种全局暂停的现象,全局停顿，所有Java代码停止，native代码可以执行，但不能和JVM交互,多半由于GC引起,Dump线程,死锁检查,堆Dump。

* GC时为什么会有全局停顿？
  + 类比在聚会时打扫房间，聚会时很乱，又有新的垃圾产生，房间永远打扫不干净，只有让大家停止活动了，才能将房间打扫干净。
* 危害
  + 长时间服务停止，没有响应
  + 遇到HA系统，可能引起主备切换，严重危害生产环境，可能出现两台同时启动。

如下程序例子:

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){

}

}

}

每100毫秒打印一次时间，程序启动以后自动执行不中止。

配置如下参数

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

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){

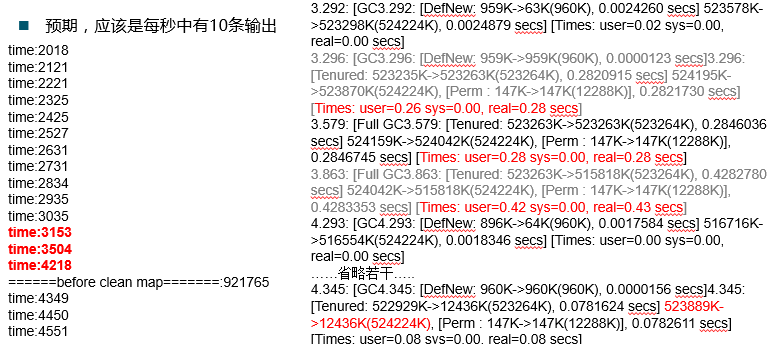
e.printStackTrace();

}

}

}

如上面的代码块所示，每秒产生一个byte数组，直到堆内存大小超过450M的时候清空一下内存。见证奇迹的时刻到了，来看我们的程序是怎么回收的



如上图左所示每100毫秒打印一次，在红色的部分却出现了短暂的jvm停顿，也就是我们说的stop-the-world,为什么会出现这样的原因呢，如右图可以看到，这个过程在执行时间3296的时候产生了一次GC大概时间是280毫秒，在3579时间的时候做了一次GC大概也是280毫秒，

在3863的时候做了一次GC大概用了430毫秒，差不多和左边刚好可以匹配，所以我们说在做GC的时候会stop-the-world。所以提高系统性能我们应该减少GC的执行次数，也就是尽量按官方的推荐去配置jvm参数。