http://www.cnblogs.com/smyhvae/p/4744233.html

本文主要内容：

- GC的概念
- GC算法

  引用计数法（无法解决循环引用的问题，不被java采纳）

  根搜索算法(判断对象是否需要回收)

  现代虚拟机中的垃圾搜集算法(垃圾回收器使用的)：

  标记-清除

  复制算法（新生代）

  标记-压缩（老年代）

  分代收集
- Stop-The-World

一、GC的概念：

- GC：Garbage Collection 垃圾收集
- 1960年 Lisp使用了GC
- Java中，**GC的对象是Java堆和方法区**（即永久区）

我们接下来对上面的三句话进行一一的解释：

（1）GC：Garbage Collection 垃圾收集。这里所谓的垃圾指的是**在系统运行过程当中所产生的一些无用的对象，这些对象占据着一定的内存空间，如果长期不被释放，可能导致OOM**。

在C/C++里是由程序猿自己去申请、管理和释放内存空间，因此没有GC的概念。而在Java中，**后台专门有一个专门用于垃圾回收的线程**来进行监控、扫描，自动将一些无用的内存进行释放，这就是垃圾收集的一个基本思想，目的在于防止由程序猿引入的人为的内存泄露。

（2）事实上，GC的历史比Java久远，1960年诞生于MIT的Lisp是第一门真正使用内存动态分配和垃圾收集技术的语言。当Lisp还在胚胎时期时，人们就在思考GC需要完成的3件事情：

哪些内存需要回收？

什么时候回收？

如何回收？

（3）内存区域中的**程序计数器、虚拟机栈、本地方法栈**这3个区域随着线程而生，线程而灭；**栈中的栈帧**随着方法的进入和退出而有条不紊地执行着出栈和入栈的操作，每个栈帧中分配多少内存基本是**在类结构确定下来时就已知的。在这几个区域不需要过多考虑回收的问题，因为方法结束或者线程结束时，内存自然就跟着回收了**。

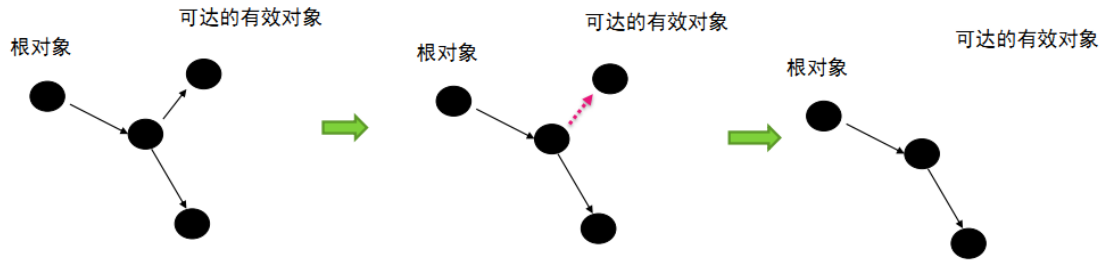
而**Java堆和方法区**则不同，一个接口中的多个实现类需要的内存可能不同，一个方

法中的多个分支需要的内存也可能不一样，我们只有在程序处于运行期间时才能知道会创建哪些对象，这部分内存的分配和回收都是动态的，GC关注的也是这部分内存，后面的文章中如果涉及到"内存"分配与回收也仅指着一部分内存。

二、引用计数算法：（老牌垃圾回收算法。无法处理循环引用，没有被Java采纳）

1、引用计数算法的概念：

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。



2、使用者举例：

引用计数算法的实现简单，判定效率也高，大部分情况下是一个不错的算法。很多地方应用到它。例如：

微软公司的COM技术：Computer Object Model

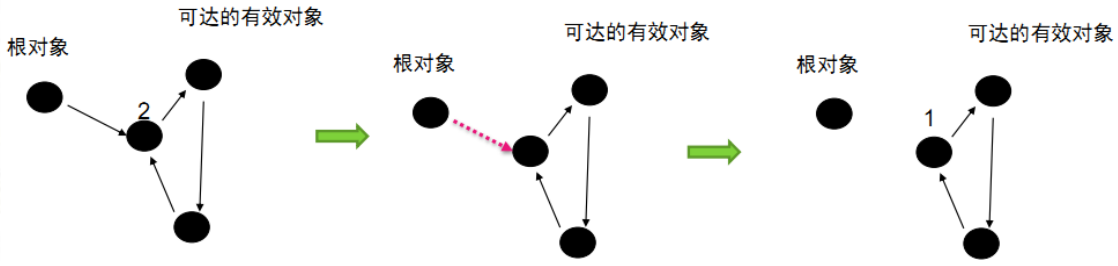
使用ActionScript3的FlashPlayer

Python

但是，主流的java虚拟机并没有选用引用计数算法来管理内存，其中最主要的原因是：它很难解决对象之间相互循环引用的问题。

3、引用计数算法的问题：

- 引用和去引用伴随加法和减法，影响性能
- 致命的缺陷：对于循环引用的对象无法进行回收



上面的3个图中，对于最右边的那张图而言：循环引用的计数器都不为0，但是他们对于根对象都已经不可达了，但是无法释放。

循环引用的代码举例：

```
1 public class Object {
```

```
2
3       Object field = null;
4
5       public static void main(String[] args) {
6           Thread thread = new Thread(new Runnable() {
7               public void run() {
8                   Object objectA = new Object();
9                   Object objectB = new Object();//位置1
10                  objectA.field = objectB;
11                  objectB.field = objectA;//位置2
12                  //to do something
13                  objectA = null;
14                  objectB = null;//位置3
15              }
16          });
17          thread.start();
18          while (true);
19      }
20
21  }
```

上方代码看起来有点刻意为之，但其实在实际编程过程当中，是经常出现的，比如两个一对一关系的数据库对象，各自保持着对方的引用。最后一个无限循环只是为了保持JVM不退出，没什么实际意义。

代码解释：

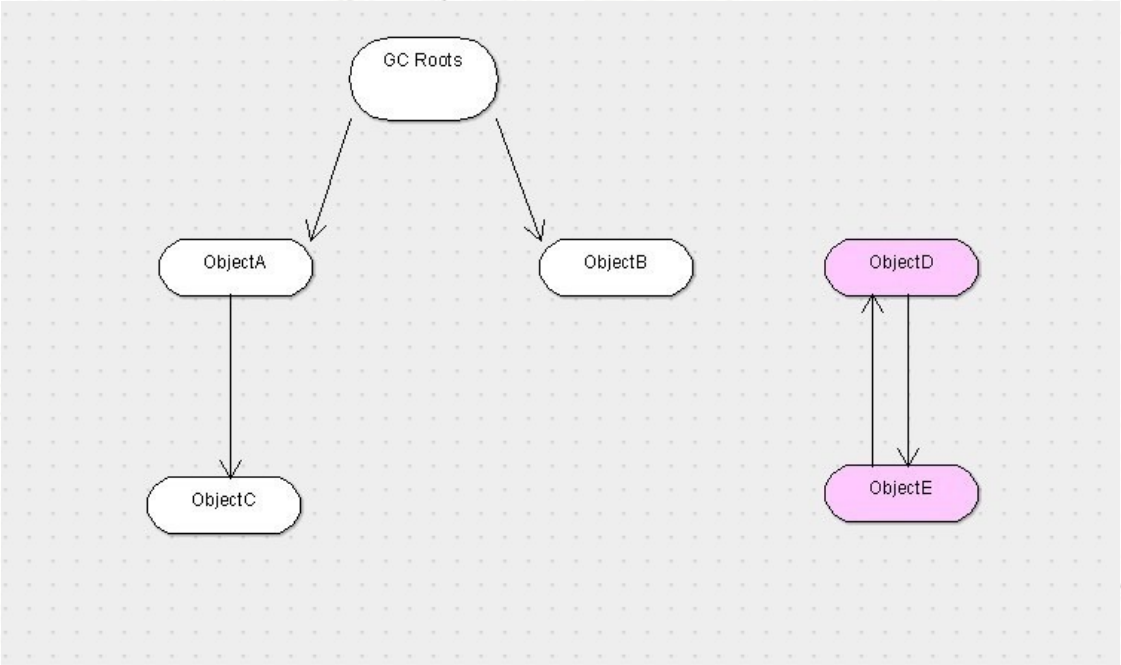
代码中标注了1、2、3三个数字，当位置1的语句执行完以后，两个对象的引用计数全部为1。当位置2的语句执行完以后，两个对象的引用计数就全部变成了2。当位置3的语句执行完以后，也就是将二者全部归为空值以后，二者的引用计数仍然为1。根据引用计数算法的回收规则，引用计数没有归0的时候是不会被回收的。

对于我们现在使用的GC来说，当thread线程运行结束后，会将objectA和objectB全部作为待回收的对象。而如**果我们的GC采用上面所说的引用计数算法，则这两个对象永远不会被回收**，即便我们在使用后显示的将对象归为空值也毫无作用。

三、根搜索算法：

1、根搜索算法的概念：

由于引用计数算法的缺陷，所以JVM一般会采用一种新的算法，叫做**根搜索算法**。它的处理方式就是，**设立若干种根对象，当任何一个根对象到某一个对象均不可达时，则认为这个对象是可以被回收的**。



如上图所示，ObjectD和ObjectE是互相关联的，但是由于GC roots到这两个对象不可达，所以最终D和E还是会被当做GC的对象，上图若是采用引用计数法，则A-E五个对象都不会被回收。

2、可达性分析：

我们刚刚提到，设立若干种根对象，当任何一个根对象到某一个对象均不可达时，则认为这个对象是可以被回收的。我们在后面介绍标记-清理算法/标记整理算法时，也会一直强调从根节点开始，对所有可达对象做一次标记，那什么叫做可达呢？这里解释如下：

可达性分析：

从根（GC Roots）的对象作为起始点，开始向下搜索，搜索所走过的路径称为"引用链"，当一个对象到GC Roots没有任何引用链相连（用图论的概念来讲，就是从GC Roots到这个对象不可达）时，则证明此对象是不可用的。

3、根（GC Roots）：

说到GC roots（GC根），在JAVA语言中，可以当做GC roots的对象有以下几种：

1、栈（栈帧中的本地变量表）中引用的对象。

2、方法区中的静态成员。

3、方法区中的常量引用的对象（全局变量）

4、本地方法栈中**JNI**（一般说的**Native**方法）引用的对象。

注：第一和第四种都是指的方法的本地变量表，第二种表达的意思比较清晰，第三种主要指的是声明为final的常量值。

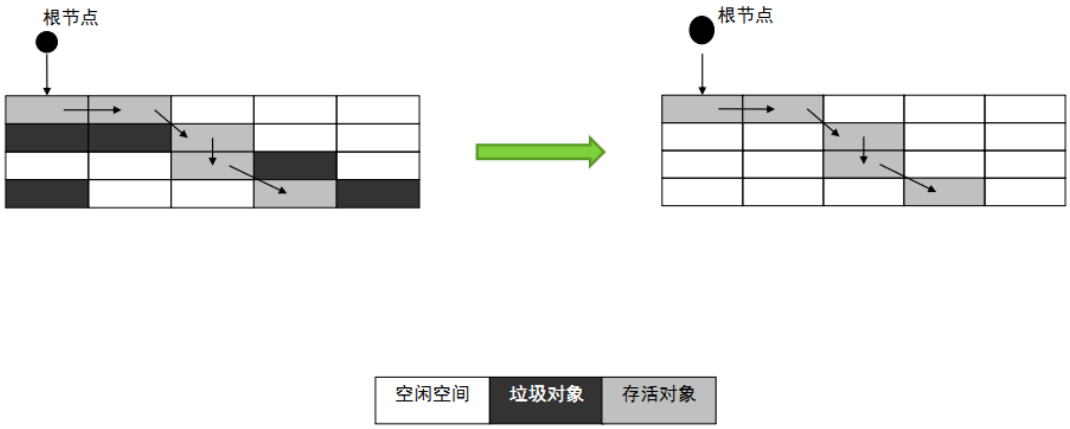
在根搜索算法的基础上，现代虚拟机的实现当中，垃圾搜集的算法主要有三种，分别

是**标记-清除算法、复制算法、标记-整理**算法。这三种算法都扩充了根搜索算法，不过它们理解起来还是非常好理解的。

四、标记-清除算法：

1、标记清除算法的概念：

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。



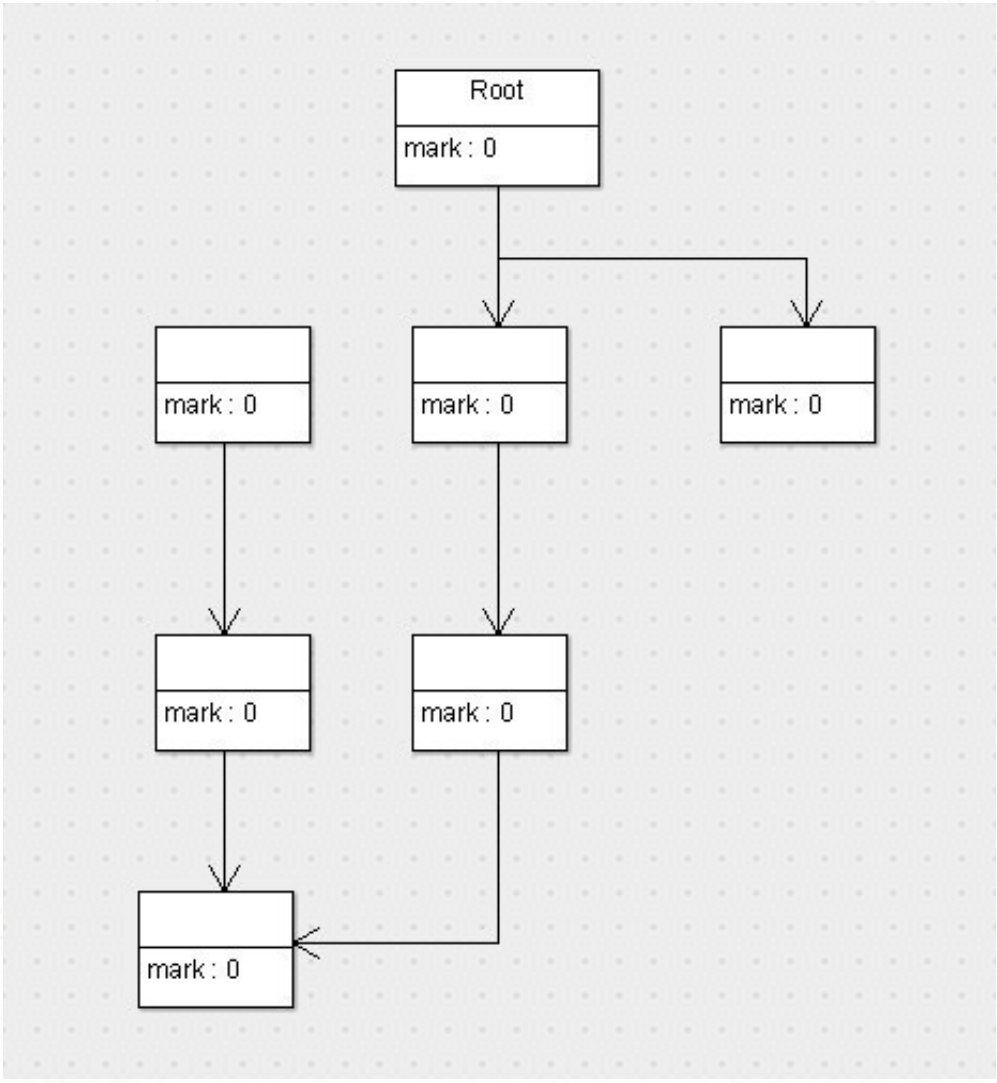
2、标记-清除算法详解：

它的做法是当堆中的有效内存空间（available memory）被耗尽的时候，就会停止整个程序（也被成为stop the world），然后进行两项工作，第一项则是标记，第二项则是清除。

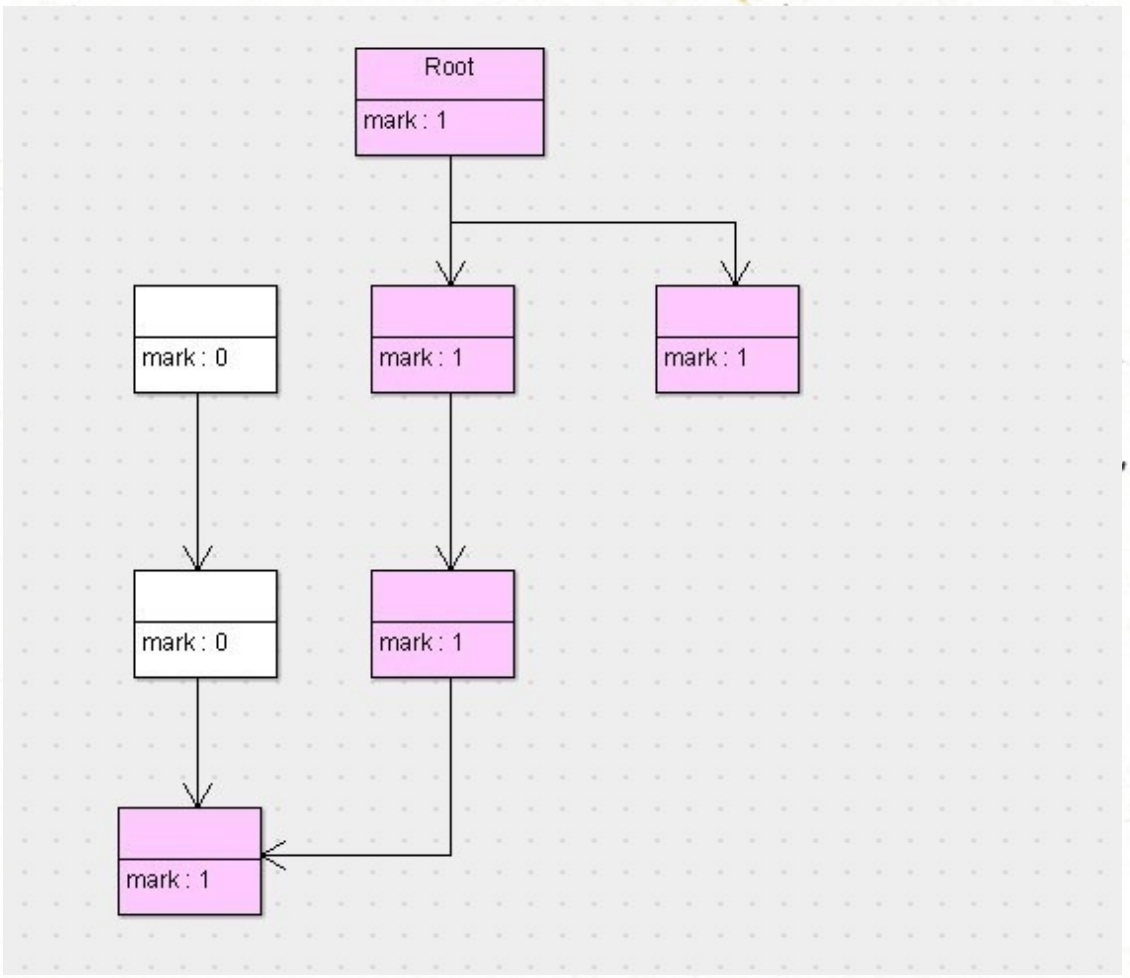
- 标记：标记的过程其实就是，遍历所有的GC Roots，然后将所有GC Roots可达的对象标记为存活的对象。
- 清除：清除的过程将遍历堆中所有的对象，将没有标记的对象全部清除掉。

也就是说，**就是当程序运行期间，若可以使用的内存被耗尽的时候，GC线程就会被触发并将程序暂停，随后将依旧存活的对象标记一遍，最终再将堆中所有没被标记的对象全部清除掉，接下来便让程序恢复运行。**

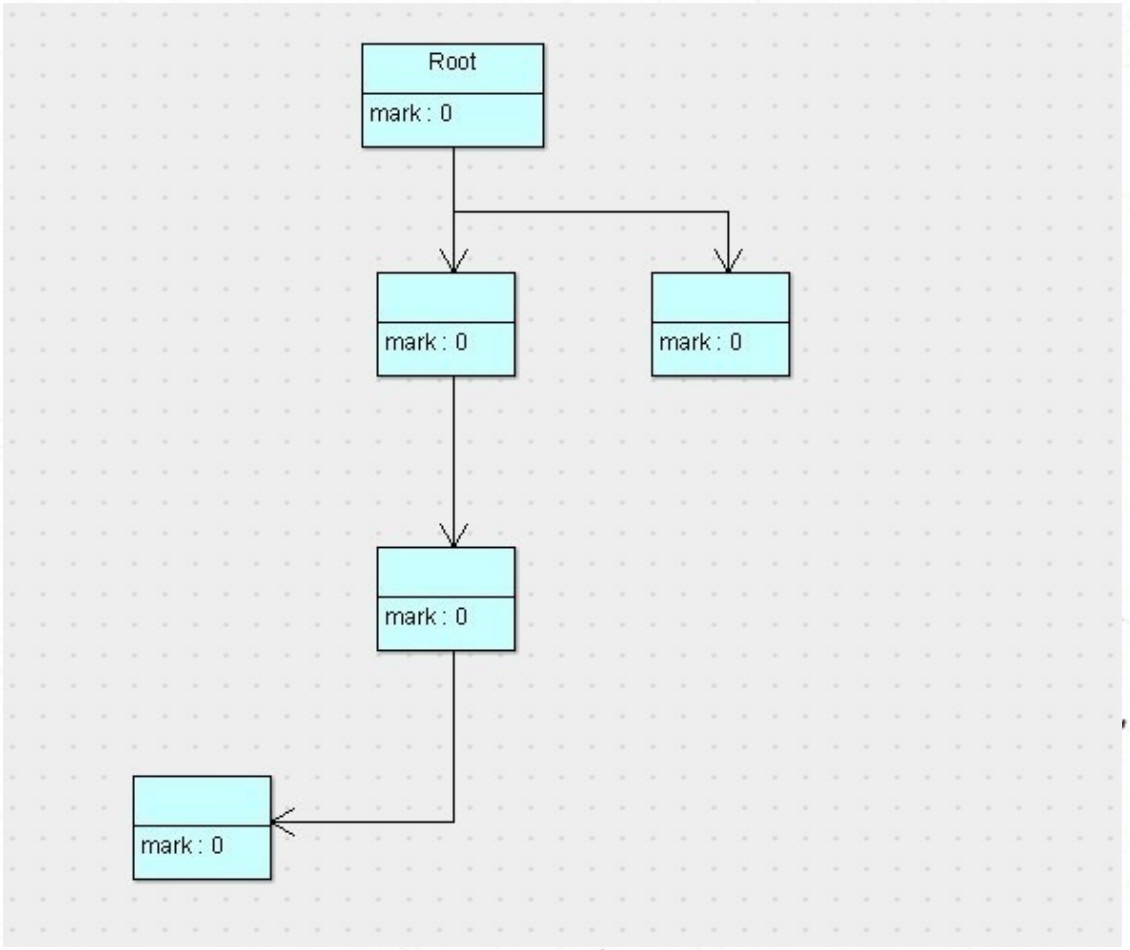
来看下面这张图：



上图代表的是程序运行期间所有对象的状态，**它们的标志位全部是0**（也就是未标记，以下默认0就是未标记，1为已标记），假设这会儿**有效内存空间耗尽**了，JVM将会停止应用程序的运行并开启GC线程，然后开始进行标记工作，按照根搜索算法，标记完以后，对象的状态如下图：



上图中可以看到，按照根搜索算法，所有从root对象可达的对象就被标记为了存活的对象，此时已经完成了第一阶段标记。接下来，就要执行第二阶段清除了，那么清除完以后，剩下的对象以及对象的状态如下图所示：



上图可以看到，没有被标记的对象将会回收清除掉，而被标记的对象将会留下，并且会将标记位重新归0。接下来就不用说了，唤醒停止的程序线程，让程序继续运行即可。

疑问：为什么非要停止程序的运行呢?

答:

这个其实也不难理解，假设我们的程序与GC线程是一起运行的，各位试想这样一种场景。

假设我们刚标记完图中最右边的那个对象，暂且记为A，结果此时在程序当中又new了一个新对象B，且A对象可以到达B对象。但是由于此时A对象已经标记结束，B对象此时的标记位依然是0，因为它错过了标记阶段。因此当接下来轮到清除阶段的时候，新对象B将会被苦逼的清除掉。如此一来，不难想象结果，GC线程将会导致程序无法正常工作。

上面的结果当然令人无法接受，我们刚new了一个对象，结果经过一次GC，忽然变成null了，这还怎么玩?

3、标记-清除算法的缺点:

（1）首先，**它的缺点就是效率比较低（递归与全堆对象遍历）**，导致stop the world的时间比较长，尤其对于交互式的应用程序来说简直是无法接受。试想一下，

如果你玩一个网站，这个网站一个小时就挂五分钟，你还玩吗?

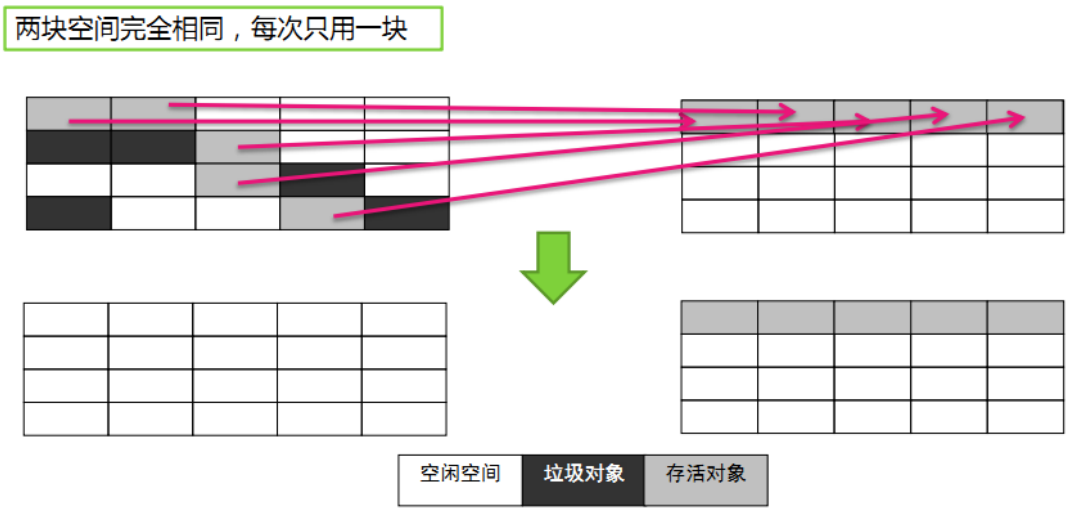
（2）第二点主要的缺点，则是**这种方式清理出来的空闲内存是不连续的**，这点不难理解，我们的死亡对象都是随即的出现在内存的各个角落的，现在把它们清除之后，内存的布局自然会乱七八糟。而为了应付这一点，JVM就不得不维持一个内存的空闲列表，这又是一种开销。而且在分配数组对象的时候，寻找连续的内存空间会不太好找。

五、复制算法：（新生代的GC）

复制算法的概念：

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。

- 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（复制算法**适合做新生代的GC**）



- **复制算法的最大的问题是：空间的浪费**

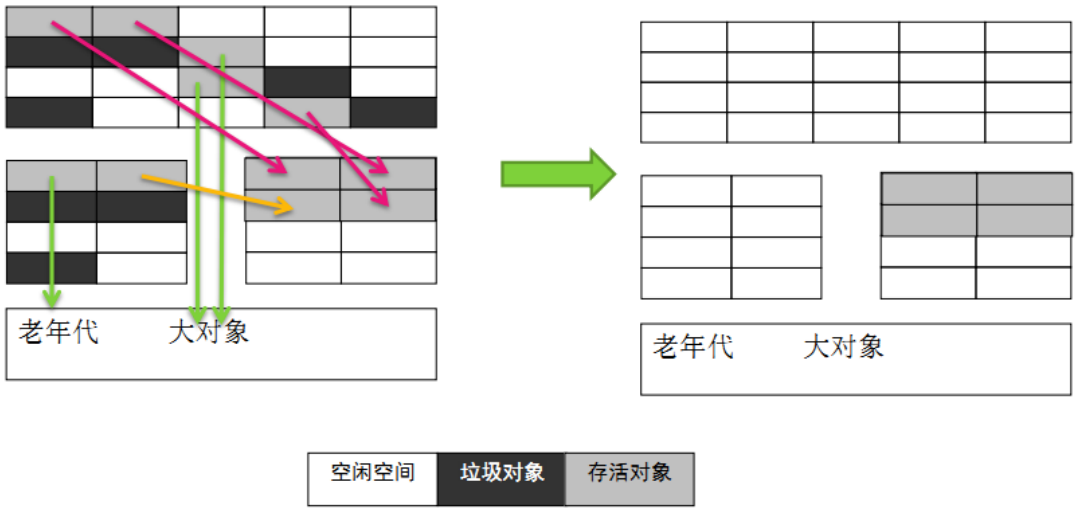
复制算法使得每次都只对整个半区进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半，这个太要命了。

所以从以上描述不难看出，复制算法要想使用，最起码对象的存活率要非常低才行，而且最重要的是，我们必须要克服50%内存的浪费。

现在的商业虚拟机都采用这种收集算法来回收新生代，新生代中的对象98%都是"朝生夕死"的，所以并不需要按照1:1的比例来划分内存空间，而是**将内存分为一块比较大的Eden空间和两块较小的Survivor空间**，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性地复制到另外一块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，也就是说，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%的空间会被浪费。

当然，98%的对象可回收只是一般场景下的数据，我们没有办法保证每次回收都只有不多于10%的对象存活，**当Survivor空间不够用时，需要依赖于老年代进行分配担保，所以大对象直接进入老年代**。整个过程如下图所示：



上图中，绿色箭头的位置代表的是大对象，大对象直接进入老年代。

根据上面的复制算法，现在我们来看下面的这个gc日志的数字，就应该能看得懂了吧：

```
-XX:+PrintGCDetails的输出
-  Heap                    12288K+ 1536K
-  def new generation  total 13824K, used 11223K [0x27e80000, 0x28d80000, 0x28d80000)  （0x28d80000-0x27e80000)/1024/1024=15M
-   eden space 12288K,  91% used [0x27e80000, 0x28975f20, 0x28a80000)
-   from space 1536K,   0% used [0x28a80000, 0x28a80000, 0x28c00000)
-   to   space 1536K,   0% used [0x28c00000, 0x28c00000, 0x28d80000)
-  tenured generation   total 5120K, used 0K [0x28d80000, 0x29280000, 0x34680000)
-    the space 5120K,   0% used [0x28d80000, 0x28d80000, 0x28d80200, 0x29280000)
-  compacting perm gen  total 12288K, used 142K [0x34680000, 0x35280000, 0x38680000)
-    the space 12288K,   1% used [0x34680000, 0x346a3a90, 0x346a3c00, 0x35280000)
-     ro space 10240K,  44% used [0x38680000, 0x38af73f0, 0x38af7400, 0x39080000)
-     rw space 12288K,  52% used [0x39080000, 0x396cdd28, 0x396cde00, 0x39c80000)
```

上方GC日志中，新生代的可用空间是13824K（eden区的12288K+from space的1536K）。而根据内存的地址计算得知，新生代的总空间为15M，而这个15M的空间是 = 13824K +to space 的 1536K。

六、标记-整理算法：（老年代的GC）

引入：

如果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中这种算法。

概念：

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。

- **标记：它的第一个阶段与标记/清除算法是一模一样的，均是遍历GC Roots，然后将存活的对象标记。**
- **整理：移动所有存活的对象，且按照内存地址次序依次排列，然后将末端内存地址以后的内存全部回收。因此，第二阶段才称为整理阶段。**

上图中可以看到，标记的存活对象将会被整理，按照内存地址依次排列，而未被标记的内存会被清理掉。如此一来，当我们需要给新对象分配内存时，JVM只需要持有一个内存的起始地址即可，这比维护一个空闲列表显然少了许多开销。

标记/整理算法不仅可以弥补标记/清除算法当中，内存区域分散的缺点，也消除了复制算法当中，内存减半的高额代价。

- 但是，标记/整理算法唯一的缺点就是效率也不高。

不仅要标记所有存活对象，还要整理所有存活对象的引用地址。从效率上来说，标记/整理算法要低于复制算法。

标记-清除算法、复制算法、标记整理算法的总结：

三个算法都基于根搜索算法去判断一个对象是否应该被回收，而支撑根搜索算法可以正常工作的理论依据，就是语法中变量作用域的相关内容。因此，要想防止内存泄露，最根本的办法就是掌握好变量作用域，而不应该使用C/C++式内存管理方式。

在GC线程开启时，或者说GC过程开始时，它们都要暂停应用程序（stop the world）。

它们的区别如下：（>表示前者要优于后者，=表示两者效果一样）

（1）效率：复制算法>标记/整理算法>标记/清除算法（此处的效率只是简单的对比时间复杂度，实际情况不一定如此）。

（2）内存整齐度：复制算法=标记/整理算法>标记/清除算法。
（3）内存利用率：标记/整理算法=标记/清除算法>复制算法。
注1：可以看到标记/清除算法是比较落后的算法了，但是后两种算法却是在此基础上建立的。
注2：时间与空间不可兼得。

七、分代收集算法：（新生代的GC+老年代的GC）
当前商业虚拟机的GC都是采用的"分代收集算法"，这并不是什么新的思想，只是根据对象的存活周期的不同将内存划分为几块儿。一般是把Java堆分为新生代和老年代：短命对象归为新生代，长命对象归为老年代。

- 少量对象存活，适合复制算法：在新生代中，每次GC时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成GC。
- 大量对象存活，适合用标记-清理/标记-整理：在老年代中，因为对象存活率高、没有额外空间对他进行分配担保，就必须使用"标记-清理"/"标记-整理"算法进行GC。

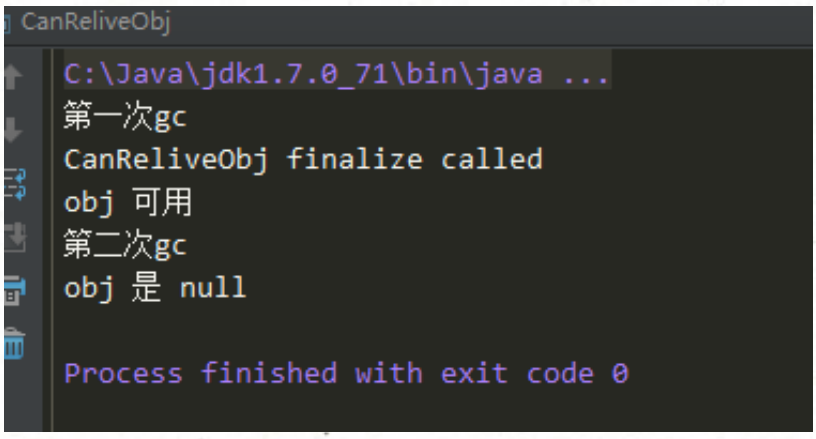
注：**老年代的对象中，有一小部分是因为在新生代回收时，老年代做担保，进来的对象；绝大部分对象是因为很多次GC都没有被回收掉而进入老年代。**

八、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。
其实就是从根节点扫描，只要这个对象在引用链中，那就是可触及的。
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
要被回收。
finalize方法复活对象的代码举例：

```
1 package test03;
2
3 /**
4  * Created by smyhvae on 2015/8/19.
5  */
6 public class CanReliveObj {
7     public static CanReliveObj obj;
8
```

```
 9      //当执行GC时，会执行finalize方法，并且只会执行一次
10      @Override
11      protected void finalize() throws Throwable {
12          super.finalize();
13          System.out.println("CanReliveObj finalize called");
14          obj = this;   //当执行GC时，会执行finalize方法，然后这一行代码的作用是将null的object复活一下，然后变成了可触及性
15      }
16
17      @Override
18      public String toString() {
19          return "I am CanReliveObj";
20      }
21
22      public static void main(String[] args) throws
23              InterruptedException {
24          obj = new CanReliveObj();
25          obj = null;   //可复活
26          System.out.println("第一次gc");
27          System.gc();
28          Thread.sleep(1000);
29          if (obj == null) {
30              System.out.println("obj 是 null");
31          } else {
32              System.out.println("obj 可用");
33          }
34          obj = null;    //不可复活
35          System.out.println("第二次gc");
36          System.gc();
37          Thread.sleep(1000);
38          if (obj == null) {
39              System.out.println("obj 是 null");
40          } else {
41              System.out.println("obj 可用");
42          }
43      }
44  }
```

我们需要注意第14行的注释。一开始，我们在第25行将obj设置为null，然后执行一次GC，本以为obj会被回收掉，其实并没有，因为GC的时候会调用11行的finalize方法，然后obj在第14行被复活了。紧接着又在第34行设置obj设置为null，然后执行一次GC，此时obj就被回收掉了，因为finalize方法只会执行一次。



finalize方法的使用总结：

- 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，自然也就不知道finalize方法什么时候执行

- 如果要使用finalize去释放资源，我们可以使用try-catch-finally来替代它

九、Stop-The-World：

1、Stop-The-World概念：

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

多半情况下是由于**GC**引起。

少数情况下由其他情况下引起，如：Dump线程、死锁检查、堆Dump。

2、GC时为什么会有全局停顿?

（1）避免无法彻底清理干净

打个比方：类比在聚会，突然GC要过来打扫房间，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。

况且，如果没有全局停顿，会给GC线程造成很大的负担，GC算法的难度也会增加，GC很难去判断哪些是垃圾。

（2）GC的工作必须在一个能确保一致性的快照中进行。

这里的一致性的意思是：在整个分析期间整个执行系统看起来就像被冻结在某个时间点上，不可以出现分析过程中对象引用关系还在不断变化的情况，该点不满足的话分析结果的准确性无法得到保证。

这点是导致GC进行时必须停顿所有Java执行线程的其中一个重要原因。

3、Stop-The-World的危害：

长时间服务停止，没有响应（将用户正常工作的线程全部暂停掉）

遇到HA系统，可能引起主备切换，严重危害生产环境。

备注：HA：High Available, 高可用性集群。

比如上面的这主机和备机：现在是主机在工作，此时如果主机正在GC造成长时间停顿，那么备机就会监测到主机没有工作，于是备机开始工作了；但是主机不工作只是暂时的，当GC结束之后，主机又开始工作了，那么这样的话，主机和备机就同时工作了。**主机和备机同时工作其实是非常危险**的，很有可能会导致应用程序不一致、不能提供正常的服务等，进而影响生产环境。

代码举例：

（1）打印日志的代码：（每隔100ms打印一条）

```
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){

        }
    }
}
```

上方代码中，是负责打印日志的代码，每隔100ms打印一条，并计算打印的时间。

（2）工作线程的代码：（工作线程，专门用来消耗内存）

```
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){    //如果map消耗的内存消耗大于450时，那就清理内存
                    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();
        }
    }
}
```

然后，我们设置gc的参数为：

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

打印日志如下：

- 预期，应该是每秒中有10条输出

```
time:2018
time:2121
time:2221
time:2325
time:2425
time:2527
time:2631
time:2731
time:2834
time:2935
time:3035
time:3153
time:3504
time:4218
======before clean map=======:921765
time:4349
time:4450
time:4551
```

```
3.292: [GC3.292: [DefNew: 959K->63K(960K), 0.0024260 secs] 523578K->523298K(524224K), 0.0024879 secs] [Times: user=0.02 sys=0.00, real=0.00 secs]
3.296: [GC3.296: [DefNew: 959K->959K(960K), 0.0000123 secs]3.296: [Tenured: 523235K->523263K(523264K), 0.2820915 secs] 524195K->523870K(524224K), [Perm : 147K->147K(12288K)], 0.2821730 secs] [Times: user=0.26 sys=0.00, real=0.28 secs]
3.579: [Full GC3.579: [Tenured: 523263K->523263K(523264K), 0.2846036 secs] 524159K->524042K(524224K), [Perm : 147K->147K(12288K)], 0.2846745 secs] [Times: user=0.28 sys=0.00, real=0.28 secs]
3.863: [Full GC3.863: [Tenured: 523263K->515818K(523264K), 0.4282780 secs] 524042K->515818K(524224K), [Perm : 147K->147K(12288K)], 0.4283353 secs] [Times: user=0.42 sys=0.00, real=0.43 secs]
4.293: [GC4.293: [DefNew: 896K->64K(960K), 0.0017584 secs] 516716K->516554K(524224K), 0.0018346 secs] [Times: user=0.00 sys=0.00, real=0.00 secs]
……省略若干……
4.345: [GC4.345: [DefNew: 960K->960K(960K), 0.0000156 secs]4.345: [Tenured: 522929K->12436K(523264K), 0.0781624 secs] 523889K->12436K(524224K), [Perm : 147K->147K(12288K)], 0.0782611 secs] [Times: user=0.08 sys=0.00, real=0.08 secs]
```

上图中，红色字体代表的正在GC。按道理来说，应该是每隔100ms会打印输出一条日志，但是当执行GC的时候，会出现全局停顿的情况，导致没有按时输出。