深入理解 Java 垃圾回收机制

一：垃圾回收机制的意义

java 语言中一个显著的特点就是引入了 java 回收机制，是 c++程序员最头疼的内存管理的问题迎刃而解，它使得 java 程序员在编写程序的时候不在考虑内存管理。由于有个垃圾回收机制，java 中的额对象不在有“作用域”的概念，只有对象的引用才有“作用域”。垃圾回收可以有效的防止内存泄露，有效的使用空闲的内存；

内存泄露：指该内存空间使用完毕后未回收，在不涉及复杂数据结构的一般情况下，java 的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度，我们有是也将其称为“对象游离”；

二：垃圾回收机制的算法

java 语言规范没有明确的说明 JVM 使用哪种垃圾回收算法，但是任何一种垃圾回收算法一般要做两件基本事情：（1）发现无用的信息对象；（2）回收将无用对象占用的内存空间。使该空间可被程序再次使用。

1。引用计数法（Reference Counting Collector）

1.1:算法分析：

引用计数算法是垃圾回收器中的早起策略，在这种方法中，堆中的每个对象实例都有一个引用计数器，点一个对象被创建时，且该对象实例分配给一个变量，该变量计数设置为 1 ，当任何其他变量赋值为这个对象的引用时，计数加 1 ，（a=b ，则 b 引用的对象实例计数器+1）但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减 1，任何引用计数器为 0 的对象实例可以当做垃圾收集。 当一个对象的实例被垃圾收集是，它引用的任何对象实例的引用计数器减 1.

一、垃圾回收机制的意义

Java 语言中一个显著的特点就是引入了垃圾回收机制，使 c++程序员最头疼的内存管理的问题迎刃而解，它使得 Java 程序员在编写程序的时候不再需要考虑内存管理。由于有个垃圾回收机制，Java 中的对象不再有“作用域”的概念，只有对象的引用才有“作用域”。垃圾回收可以有效的防止内存泄露，有效的使用空闲的内存。

ps:内存泄露是指该内存空间使用完毕之后未回收，在不涉及复杂数据结构的一般情况下，Java 的内存泄露表现为一个内存对象的生命周期超出了程序需要它的时间长度，我们有时也将其称为“对象游离”。

二、垃圾回收机制中的算法

Java 语言规范没有明确地说明 JVM 使用哪种垃圾回收算法，但是任何一种垃圾回收算法一般要做 2 件基本的事情：（1）发现无用信息对象；（2）回收被无用对象占用的内存空间，使该空间可被程序再次使用。

1.引用计数法(Reference Counting Collector)

1.1 算法分析

引用计数是垃圾收集器中的早期策略。在这种方法中，堆中每个对象实例都有一个引用计数。当一个对象被创建时，且将该对象实例分配给一个变量，该变量计数设置为 1。当任何其它变量被赋值为这个对象的引用时，计数加 1（a = b,则 b 引用的对象实例的计数器+1），但当一个对象实例的某个引用超过了生命周期或者被设置为一个新值时，对象实例的引用计数器减 1。任何引用计数器为 0 的对象实例可以被当作垃圾收集。当一个对象实例被垃圾收集时，它引用的任何对象实例的引用计数器减 1。

1.2 优缺点

优点：

引用计数收集器可以很快的执行，交织在程序运行中。对程序需要不被长时间打断的实时环境比较有利。

缺点：

无法检测出循环引用。如父对象有一个对子对象的引用，子对象反过来引用父对象。这样，他们的引用计数永远不可能为 0.

1.3 引用计数算法无法解决循环引用问题，例如：

```
1   public  class  Main {
2           public  static  void  main(String[] args) {
3                   MyObject object1 = new  MyObject();
4                   MyObject object2 = new  MyObject();
5
6                   object1.object = object2;
7                   object2.object = object1;
8
9                   object1 = null;
10                  object2 = null;
11          }
12  }
```

最后面两句将 object1 和 object2 赋值为 null，也就是说 object1 和 object2 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器都不为 0，那么垃圾收集器就永远不会回收它们。

2.tracing 算法(Tracing Collector) 或 标记-清除算法(mark and sweep)

2.1 根搜索算法

根搜索算法是从离散数学中的图论引入的，程序把所有的引用关系看作一张图，从一个节点 GC ROOT 开始，寻找对应的引用节点，找到这个节点以后，继续寻找这个节点的引用节点，当所有的引用节点寻找完毕之后，剩余的节点则被认为是没有被引用到的节点，即无用的节点。

java 中可作为 GC Root 的对象有

1.虚拟机栈中引用的对象（本地变量表）

2.方法区中静态属性引用的对象

3. 方法区中常量引用的对象

4.本地方法栈中引用的对象（Native 对象）

2.2tracing 算法的示意图

2.3 标记-清除算法分析

标记-清除算法采用从根集合进行扫描，对存活的对象对象标记，标记完毕后，再扫描整个空间中未被标记的对象，进行回收，如上图所示。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，在存活对象比较多的情况下极为高效，但由于标记-清除算法直接回收不存活的对象，因此会造成内存碎片。

3.compacting 算法 或 标记-整理算法

标记-整理算法采用标记-清除算法一样的方式进行对象的标记，但在清除时不同，在回收不存活的对象占用的空间后，会将所有的存活对象往左端空闲空间移动，并更新对应的指针。标记-整理算法是在标记-清除算法的基础上，又进行了对象的移动，因此成本更高，但是却解决了内存碎片的问题。在基于 Compacting 算法的收集器的实现中，一般增加句柄和句柄表。

4.copying 算法(Compacting Collector)

该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收。它开始时把堆分成 一个对象 面和多个空闲面， 程序从对象面为对象分配空间，当对象满了，基于 copying 算法的垃圾 收集就从根集中扫描活动对象，并将每个 活动对象复制到空闲面(使得活动对象所占的内存之间没有空闲洞)，这样空闲面变成了对象面，原来的对象面变成了空闲面，程序会在新的对象面中分配内存。一种典型的基于 coping 算法的垃圾回收是 stop-and-copy 算法，它将堆分成对象面和空闲区域面，在对象面与空闲区域面的切换过程中，程序暂停执行。

5.generation 算法(Generational Collector)

分代的垃圾回收策略，是基于这样一个事实：**不同的对象的生命周期是不一样的**。因此，不同生命周期的对象可以采取不同的回收算法，以便提高回收效率。

年轻代（Young Generation）

1.所有新生成的对象首先都是放在年轻代的。年轻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。

2.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。一个 Eden 区，两个 Survivor 区(一般而言)。大部分对象在 Eden 区中生成。回收时先将 eden 区存活对象复制到一个 survivor0 区，然后清空 eden 区，当这个 survivor0 区也存放满了时，则将 eden 区和 survivor0 区存活对象复制到另一个 survivor1 区，然后清空 eden 和这个 survivor0 区，此时 survivor0 区是空的，然后将 survivor0 区和 survivor1 区交换，即保持 survivor1 区为空， 如此往复。

3.当 survivor1 区不足以存放 eden 和 survivor0 的存活对象时，就将存活对象直接存放到老年代。若是老年代也满了就会触发一次 Full GC，也就是新生代、老年代都进行回收

4.新生代发生的 GC 也叫做 Minor GC，MinorGC 发生频率比较高(不一定等 Eden 区满了才触发)

年老代（Old Generation）

1.在年轻代中经历了 N 次垃圾回收后仍然存活的对象，就会被放到年老代中。因此，可以认为年老代中存放的都是一些生命周期较长的对象。

2.内存比新生代也大很多(大概比例是 1:2)，当老年代内存满时触发 Major GC 即 Full GC，Full GC 发生频率比较低，老年代对象存活时间比较长，存活率标记高。

持久代（Permanent Generation）

用于存放静态文件，如 Java 类、方法等。持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 Hibernate 等，在这种时候需要设置一个比较大的持久代空间来存放这些运行过程中新增的类。

三.GC（垃圾收集器）

新生代收集器使用的收集器：Serial、PraNew、Parallel Scavenge

老年代收集器使用的收集器：Serial Old、Parallel Old、CMS

Serial 收集器（复制算法)

新生代单线程收集器，标记和清理都是单线程，优点是简单高效。

Serial Old 收集器(标记-整理算法)

老年代单线程收集器，Serial 收集器的老年代版本。

ParNew 收集器(停止-复制算法)

新生代收集器，可以认为是 Serial 收集器的多线程版本,在多核 CPU 环境下有着比 Serial 更好的表现。

Parallel Scavenge 收集器(停止-复制算法)

并行收集器，追求高吞吐量，高效利用 CPU。吞吐量一般为 99%， 吞吐量= 用户线程时间/(用户线程时间+GC 线程时间)。适合后台应用等对交互相应要求不高的场景。

Parallel Old 收集器(停止-复制算法)

Parallel Scavenge 收集器的老年代版本，并行收集器，吞吐量优先

CMS(Concurrent Mark Sweep)收集器（标记-清理算法）

高并发、低停顿，追求最短 GC 回收停顿时间，cpu 占用比较高，响应时间快，停顿时间短，多核 cpu 追求高响应时间的选择

四、GC 的执行机制

由于对象进行了分代处理，因此垃圾回收区域、时间也不一样。GC 有两种类型：Scavenge GC 和 Full GC。

Scavenge GC

一般情况下，当新对象生成，并且在 Eden 申请空间失败时，就会触发 Scavenge GC，对 Eden 区域进行 GC，清除非存活对象，并且把尚且存活的对象移动到 Survivor 区。然后整理 Survivor 的两个区。这种方式的 GC 是对年轻代的 Eden 区进行，不会影响到年老代。因为大部分对象都是从 Eden 区开始的，同时 Eden 区不会分配的很大，所以 Eden 区的 GC 会频繁进行。因而，一般在这里需要使用速度快、效率高的算法，使 Eden 去能尽快空闲出来。

Full GC

对整个堆进行整理，包括 Young、Tenured 和 Perm。Full GC 因为需要对整个堆进行回收，所以比 Scavenge GC 要慢，因此应该尽可能减少 Full GC 的次数。在对 JVM 调优的过程中，很大一部分工作就是对于 FullGC 的调节。有如下原因可能导致 Full GC：

1.年老代（Tenured）被写满

2.持久代（Perm）被写满

3.System.gc()被显示调用

4.上一次 GC 之后 Heap 的各域分配策略动态变化

五、Java 有了 GC 同样会出现内存泄露问题

1.静态集合类像 HashMap、Vector 等的使用最容易出现内存泄露，这些静态变量的生命周期和应用程序一致，所有的对象 Object 也不能被释放，因为他们也将一直被 Vector 等应用着。

```
1  Static Vector v = new  Vector();
2  for  (int  i = 1; i<100; i++)
3  {
4          Object o = new  Object();
5          v.add(o);
6          o = null;
7  }
```

在这个例子中，代码栈中存在 Vector 对象的引用 v 和 Object 对象的引用 o 。在 For 循环中，我们不断的生成新的对象，然后将其添加到 Vector 对象中，之后将 o 引用置空。问题是当 o 引用被置空后，如果发生 GC，我们创建的 Object 对象是否能够被 GC 回收呢？答案是否定的。因为， GC 在跟踪代码栈中的引用时，会发现 v 引用，而继续往下跟踪，就会发现 v 引用指向的内存空间中又存在指向 Object 对象的引用。也就是说尽管 o 引用已经被置空，但是 Object 对象仍然存在其他的引用，是可以被访问到的，所以 GC 无法将其释放掉。如果在此循环之后， Object 对象对程序已经没有任何作用，那么我们就认为此 Java 程序发生了内存泄漏。

2.各种连接，数据库连接，网络连接，IO 连接等没有显示调用 close 关闭，不被 GC 回收导致内存泄露。

3.监听器的使用，在释放对象的同时没有相应删除监听器的时候也可能导致内存泄露。

一.如何确定某个对象是“垃圾”？

在这一小节我们先了解一个最基本的问题：如果确定某个对象是“垃圾”？既然垃圾收集器的任务是回收垃圾对象所占的空间供新的对象使用，那么垃圾收集器如何确定某个对象是“垃圾”？—即通过什么方法判断一个对象可以被回收了。

在 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。那么很显然一个简单的办法就是通过引用计数来判断一个对象是否可以被回收。不失一般性，如果一个对象没有任何引用与之关联，则说明该对象基本不太可能在其他地方被使用到，那么这个对象就成为可被回收的对象了。这种方式成为引用计数法。

这种方式的特点是实现简单，而且效率较高，但是它无法解决循环引用的问题，因此在 Java 中并没有采用这种方式（Python 采用的是引用计数法）。看下面这段代码：

```
1   public class Main {
2           public static void main(String[] args) {
3                   MyObject object1 = new MyObject();
4                   MyObject object2 = new MyObject();
5
6                   object1.object = object2;
7                   object2.object = object1;
8
9                   object1 = null;
10                  object2 = null;
11          }
12  }
13
14  class MyObject{
15          public Object object = null;
16  }
```

最后面两句将 object1 和 object2 赋值为 null，也就是说 object1 和 object2 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。

为了解决这个问题，在 Java 中采取了 可达性分析法。该方法的基本思想是通过一系列的“GC Roots”对象作为起点进行搜索，如果在“GC Roots”和一个对象之间没有可达路径，则称该对象是不可达的，不过要注意的是被判定为不可达的对象不一定就会成为可回收对象。被判定为不可达的对象要成为可回收对象必须至少经历两次标记过程，如果在这两次标记过程中仍然没有逃脱成为可回收对象的可能性，则基本上就真的成为可回收对象了。

至于可达性分析法具体是如何操作的我暂时也没有看得很明白，如果有哪位朋友比较清楚的话请不吝指教。

下面来看个例子：

```
1  Object aobj = new Object ( ) ;
2  Object bobj = new Object ( ) ;
3  Object cobj = new Object ( ) ;
4  aobj = bobj;
5  aobj = cobj;
6  cobj = null;
7  aobj = null;
```

第几行有可能会使得某个对象成为可回收对象？第 7 行的代码会导致有对象会成为可回收对象。至于为什么留给读者自己思考。

再看一个例子：

```
1  String str = new String("hello");
2  SoftReference<String> sr = new SoftReference<String>(new String("java"));
3  WeakReference<String> wr = new WeakReference<String>(new String("world"));
```

这三句哪句会使得 String 对象成为可回收对象？第 2 句和第 3 句，第 2 句在内存不足的情况下会将 String 对象判定为可回收对象，第 3 句无论什么情况下 String 对象都会被判定为可回收对象。

最后总结一下平常遇到的比较常见的将对象判定为可回收对象的情况：

1）显示地将某个引用赋值为 null 或者将已经指向某个对象的引用指向新的对象，比如下面的代码：

```
1  Object obj = new Object();
2  obj = null;
3  Object obj1 = new Object();
4  Object obj2 = new Object();
5  obj1 = obj2;
```

2）局部引用所指向的对象，比如下面这段代码：

```
1  void fun() {
2
3  .....
4          for(int i=0;i<10;i++) {
5                  Object obj = new Object();
6                  System.out.println(obj.getClass());
7          }
8  }
```

循环每执行完一次，生成的 Object 对象都会成为可回收的对象。

3）只有弱引用与其关联的对象，比如：

```
1  WeakReference<String> wr = new WeakReference<String>(new String("world"));
```

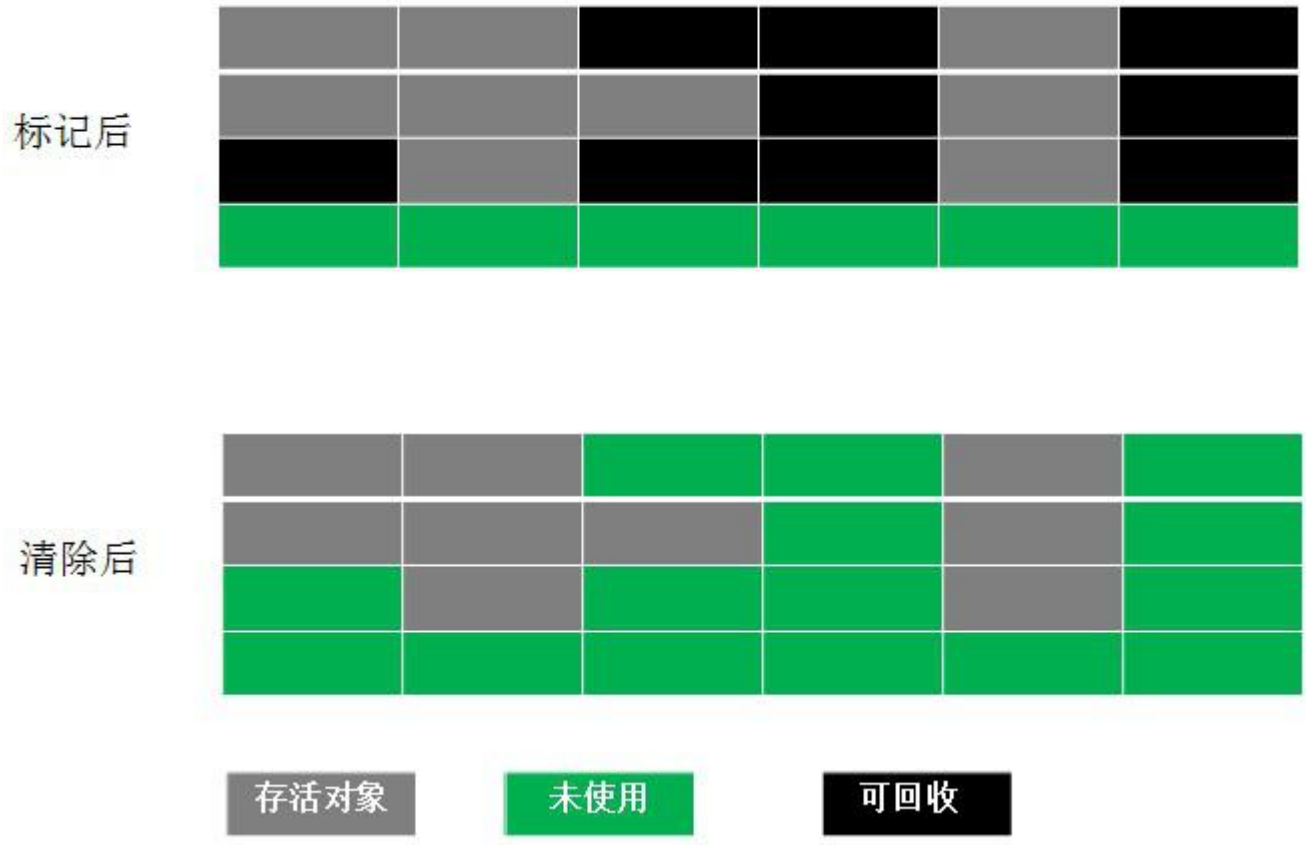
二.典型的垃圾收集算法

在确定了哪些垃圾可以被回收后，垃圾收集器要做的事情就是开始进行垃圾回收，但是这里面涉及到一个问题是：如何高效地进行垃圾回收。由于 Java 虚拟机规范并没有对如何实现

垃圾收集器做出明确的规定，因此各个厂商的虚拟机可以采用不同的方式来实现垃圾收集器，所以在此只讨论几种常见的垃圾收集算法的核心思想。

1.Mark-Sweep（标记-清除）算法

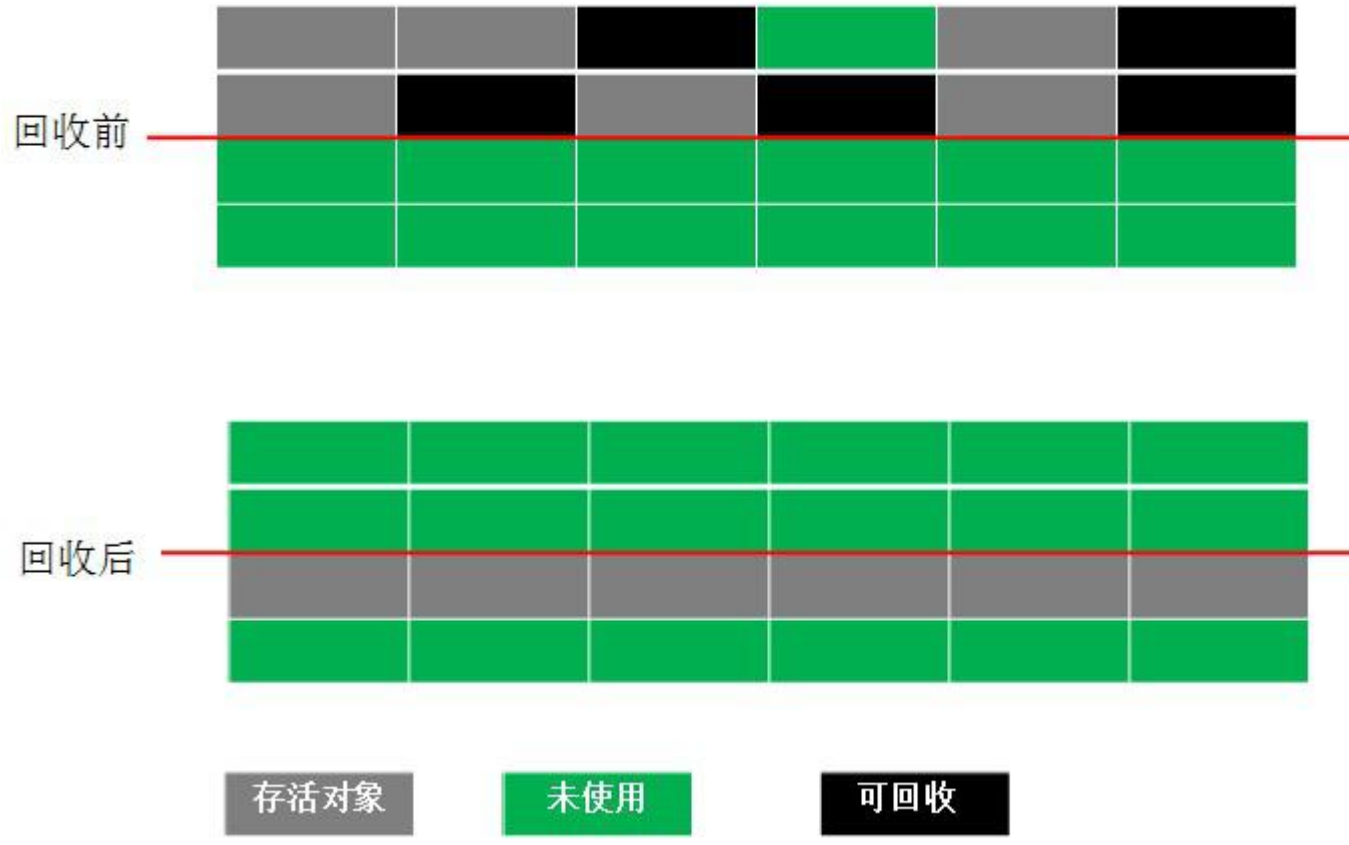
这是最基础的垃圾回收算法，之所以说它是最基础的是因为它最容易实现，思想也是最简单的。标记-清除算法分为两个阶段：标记阶段和清除阶段。标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。具体过程如下图所示：



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.Copying（复制）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：

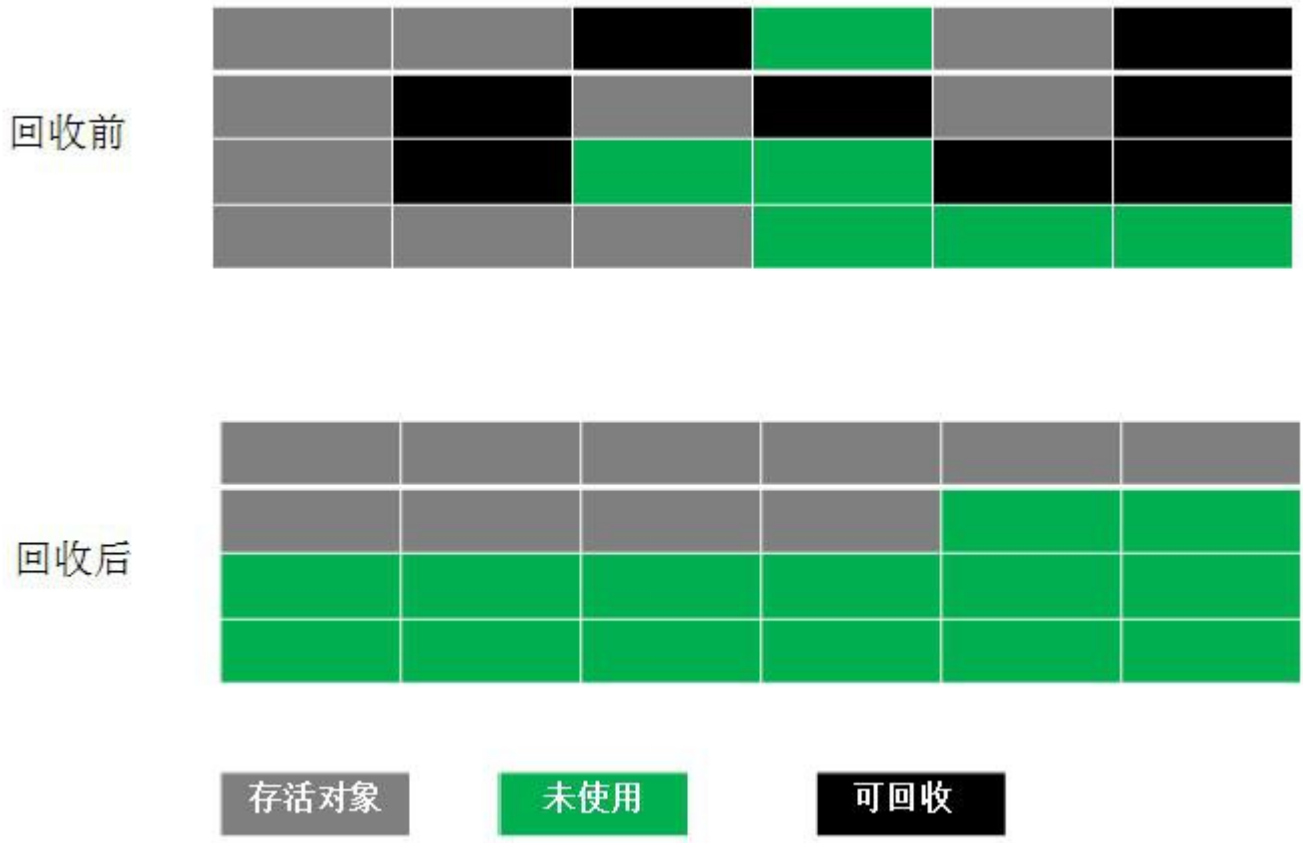


这种算法虽然实现简单，运行高效且不容易产生内存碎片，但是却对内存空间的使用做出了高昂的代价，因为能够使用的内存缩减到原来的一半。

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。

3.Mark-Compact（标记-整理）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：



4.Generational Collection（分代收集）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。

注意，在堆区之外还有一个代就是永久代（Permanet Generation），它用来存储 class 类、常量、方法描述等。对永久代的回收主要回收两部分内容：废弃常量和无用的类。

三.典型的垃圾收集器

垃圾收集算法是 内存回收的理论基础，而垃圾收集器就是内存回收的具体实现。下面介绍一下 HotSpot（JDK 7)虚拟机提供的几种垃圾收集器，用户可以根据自己的需求组合出各个年代使用的收集器。

1.Serial/Serial Old

Serial/Serial Old 收集器是最基本最古老的收集器，它是一个单线程收集器，并且在它进行垃圾收集时，必须暂停所有用户线程。Serial 收集器是针对新生代的收集器，采用的是

Copying 算法 ,Serial Old 收集器是针对老年代的收集器 ,采用的是 Mark-Compact 算法。它的优点是实现简单高效，但是缺点是会给用户带来停顿。

2.ParNew

ParNew 收集器是 Serial 收集器的多线程版本，使用多个线程进行垃圾收集。

3.Parallel Scavenge

Parallel Scavenge 收集器是一个新生代的多线程收集器（并行收集器），它在回收期间不需要暂停其他用户线程，其采用的是 Copying 算法，该收集器与前两个收集器有所不同，它主要是为了达到一个可控的吞吐量。

4.Parallel Old

Parallel Old 是 Parallel Scavenge 收集器的老年代版本（并行收集器），使用多线程和 Mark-Compact 算法。

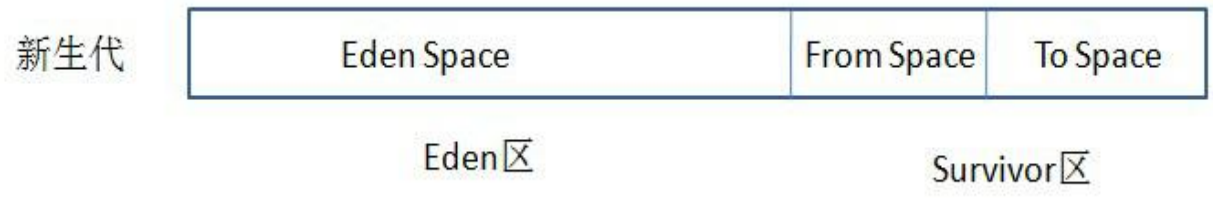
5.CMS

CMS（Current 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，它是一种并发收集器，采用的是 Mark-Sweep 算法。

6.G1

G1 收集器是当今收集器技术发展最前沿的成果，它是一款面向服务端应用的收集器，它能充分利用多 CPU、多核环境。因此它是一款并行与并发收集器，并且它能建立可预测的停顿时间模型。

下面补充一下关于内存分配方面的东西：



对象的内存分配，往大方向上讲就是在堆上分配，对象主要分配在新生代的 Eden Space 和 From Space，少数情况下会直接分配在老年代。如果新生代的 Eden Space 和 From Space 的空间不足，则会发起一次 GC，如果进行了 GC 之后，Eden Space 和 From Space 能够容纳该对象就放在 Eden Space 和 From Space。在 GC 的过程中，会将 Eden Space 和 From Space 中的存活对象移动到 To Space，然后将 Eden Space 和 From Space 进行清理。如果在清理的过程中，To Space 无法足够来存储某个对象，就会将该对象移动到老年代中。在进行了 GC 之后，使用的便是 Eden space 和 To Space 了，下次 GC 时会将存活对象复制到 From Space，如此反复循环。当对象在 Survivor 区躲过一次 GC 的话，其对象年龄便会加 1，默认情况下，如果对象年龄达到 15 岁，就会移动到老年代中。

一般来说，大对象会被直接分配到老年代，所谓的大对象是指需要大量连续存储空间的对象，最常见的一种大对象就是大数组，比如：

```
byte[] data = new byte[4*1024*1024]
```

这种一般会直接在老年代分配存储空间。

当然分配的规则并不是百分之百固定的，这要取决于当前使用的是哪种垃圾收集器组合和JVM 的相关参数。

1. 垃圾回收的意义

在 C++中，对象所占的内存在程序结束运行之前一直被占用，在明确释放之前不能分配给其它对象；而在 **Java** 中，当没有对象引用指向原先分配给某个对象的内存时，该内存便成为垃圾。JVM 的一个系统级线程会自动释放该内存块。垃圾回收意味着程序不再需要的对象是"无用信息"，这些信息将被丢弃。当一个对象不再被引用的时候，内存回收它占领的空间，以便空间被后来的新对象使用。事实上，除了释放没用的对象，垃圾回收也可以清

除内存记录碎片。由于创建对象和垃圾回收器释放丢弃对象所占的内存空间，内存会出现碎片。碎片是分配给对象的内存块之间的空闲内存洞。碎片整理将所占用的堆内存移到堆的一端，JVM 将整理出的内存分配给新的对象。

垃圾回收能自动释放内存空间，减轻编程的负担。这使 Java 虚拟机具有一些优点。首先，它能使编程效率提高。在没有垃圾回收机制的时候，可能要花许多时间来解决一个难懂的存储器问题。在用 Java 语言编程的时候，靠垃圾回收机制可大大缩短时间。其次是它保护程序的完整性， 垃圾回收是 Java 语言安全性策略的一个重要部份。

垃圾回收的一个潜在的缺点是它的开销影响程序性能。Java 虚拟机必须追踪运行程序中有用的对象，而且最终释放没用的对象。这一个过程需要花费处理器的时间。其次垃圾回收算法的不完备性，早先采用的某些垃圾回收算法就不能保证 100%收集到所有的废弃内存。当然随着垃圾回收算法的不断改进以及软硬件运行效率的不断提升，这些问题都可以迎刃而解。

2. 垃圾收集的算法分析

Java 语言规范没有明确地说明 JVM 使用哪种垃圾回收算法，但是任何一种垃圾回收算法一般要做 2 件基本的事情：（1）发现无用信息对象；（2）回收被无用对象占用的内存空间，使该空间可被程序再次使用。

大多数垃圾回收算法使用了根集(root set)这个概念；所谓根集就是正在执行的 Java 程序可以访问的引用变量的集合(包括局部变量、参数、类变量)，程序可以使用引用变量访问对象的属性和调用对象的方法。垃圾回收首先需要确定从根开始哪些是可达的和哪些是不可达的，从根集可达的对象都是活动对象，它们不能作为垃圾被回收，这也包括从根集间接可达的对象。而根集通过任意路径不可达的对象符合垃圾收集的条件，应该被回收。下面介绍几个常用的算法。

2.1. 引用计数法(Reference Counting Collector)

引用计数法是唯一没有使用根集的垃圾回收的法，该算法使用引用计数器来区分存活对象和不再使用的对象。一般来说，堆中的每个对象对应一个引用计数器。当每一次创建一个对象并赋给一个变量时，引用计数器置为 1。当对象被赋给任意变量时，引用计数器每次加 1 当对象出了作用域后(该对象丢弃不再使用)，引用计数器减 1，一旦引用计数器为 0，对象就满足了垃圾收集的条件。

基于引用计数器的垃圾收集器运行较快，不会长时间中断程序执行，适宜地必须实时运行的程序。但引用计数器增加了程序执行的开销，因为每次对象赋给新的变量，计数器加 1，而每次现有对象出了作用域生，计数器减 1。

2.2. tracing 算法(Tracing Collector)

tracing 算法是为了解决引用计数法的问题而提出，它使用了根集的概念。基于 tracing 算法的垃圾收集器从根集开始扫描，识别出哪些对象可达，哪些对象不可达，并用某种方式标记可达对象，例如对每个可达对象设置一个或多个位。在扫描识别过程中，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.

2.3. compacting 算法(Compacting Collector)

为了解决堆碎片问题，基于 tracing 的垃圾回收吸收了 Compacting 算法的思想，在清除的过程中，算法将所有的对象移到堆的一端，堆的另一端就变成了一个相邻的空闲内存区，收集器会对它移动的所有对象的所有引用进行更新，使得这些引用在新的位置能识别原来的对象。在基于 Compacting 算法的收集器的实现中，一般增加句柄和句柄表。

2.4. copying 算法(Coping Collector)

该算法的提出是为了克服句柄的开销和解决堆碎片的垃圾回收。它开始时把堆分成一个

对象区和多个空闲区，程序从对象区为对象分配空间，当对象满了，基于 coping 算法的垃圾回收就从根集中扫描活动对象，并将每个活动对象复制到空闲区(使得活动对象所占的内存之间没有空闲间隔)，这样空闲区变成了对象区，原来的对象区变成了空闲区，程序会在新的对象区中分配内存。

一种典型的基于 coping 算法的垃圾回收是 stop-and-copy 算法，它将堆分成对象区和空闲区域区，在对象区与空闲区域的切换过程中，程序暂停执行。

2.5. generation 算法(Generational Collector)

stop-and-copy 垃圾收集器的一个缺陷是收集器必须复制所有的活动对象，这增加了程序等待时间，这是 coping 算法低效的原因。在程序设计中有这样的规律：多数对象存在的时间比较短，少数的存在时间比较长。因此，generation 算法将堆分成两个或多个，每个子堆作为对象的一代 (generation)。由于多数对象存在的时间比较短，随着程序丢弃不使用的对象，垃圾收集器将从最年轻的子堆中收集这些对象。在分代式的垃圾收集器运行后，上次运行存活下来的对象移到下一最高代的子堆中，由于老一代的子堆不会经常被回收，因而节省了时间。

2.6. adaptive 算法(Adaptive Collector)

在特定的情况下，一些垃圾收集算法会优于其它算法。基于 Adaptive 算法的垃圾收集器就是监控当前堆的使用情况，并将选择适当算法的垃圾收集器。

3. System.gc()方法

命令行参数透视垃圾收集器的运行

使用 System.gc()可以不管 JVM 使用的是哪一种垃圾回收的算法，都可以请求 Java 的垃圾回收。在命令行中有一个参数-verbosegc 可以查看 Java 使用的堆内存的情况，它的格式如下：

java -verbosegc classfile

可以看个例子：

[java] view plain copy

```java
class TestGC
{
    public static void main(String[] args)
    {
        new TestGC();
        System.gc();
        System.runFinalization();
     }
}
```

在这个例子中，一个新的对象被创建，由于它没有使用，所以该对象迅速地变为不可达，程序编译后，执行命令： java -verbosegc TestGC 后结果为：

[Full GC 168K->97K(1984K)， 0.0253873 secs]

机器的环境为，Windows 2000 + JDK1.3.1，箭头前后的数据 168K 和 97K 分别表示垃圾收集 GC 前后所有存活对象使用的内存容量，说明有 168K-97K=71K 的对象容量被回收，括号内的数据 1984K 为堆内存的总容量，收集所需要的时间是 0.0253873 秒（这个时间在每次执行的时候会有所不同）。

需要注意的是，调用 System.gc()也仅仅是一个请求(建议)。JVM 接受这个消息后，并不是立即做垃圾回收，而只是对几个垃圾回收算法做了加权，使垃圾回收操作容易发生，或提早发生，或回收较多而已。

4. finalize()方法

在 JVM 垃圾回收器收集一个对象之前，一般要求程序调用适当的方法释放资源，但在没有明确释放资源的情况下，Java 提供了缺省机制来终止该对象心释放资源，这个方法就是 finalize（）。它的原型为：

protected void finalize() throws Throwable

在 finalize()方法返回之后，对象消失，垃圾收集开始执行。原型中的 throws Throwable 表示它可以抛出任何类型的异常。

之所以要使用 finalize()，是存在着垃圾回收器不能处理的特殊情况。假定你的对象（并非使用 new 方法）获得了一块“特殊”的内存区域，由于垃圾回收器只知道那些显示地经由 new 分配的内存空间，所以它不知道该如何释放这块“特殊”的内存区域，那么这个时候 java 允许在类中定义一个由 finalize()方法。

特殊的区域例如：1）由于在分配内存的时候可能采用了类似 C 语言的做法，而非 JAVA 的通常 new 做法。这种情况主要发生在 native method 中，比如 native method 调用了 C/C++方法 malloc()函数系列来分配存储空间，但是除非调用 free()函数，否则这些内存空间将不会得到释放，那么这个时候就可能造成内存泄漏。但是由于 free()方法是在 C/C++中的函数，所以 finalize()中可以用本地方法来调用它。以释放这些“特殊”的内存空间。2）又或者打开的文件资源，这些资源不属于垃圾回收器的回收范围。

换言之，finalize()的主要用途是释放一些其他做法开辟的内存空间，以及做一些清理工作。因为在 JAVA 中并没有提够像“析构”函数或者类似概念的函数，要做一些类似清理工作的时候，必须自己动手创建一个执行清理工作的普通方法，也就是 override Object 这个类中的 finalize()方法。例如，假设某一个对象在创建过程中会将自己绘制到屏幕上，如果不是明确地从屏幕上将其擦出，它可能永远都不会被清理。如果在 finalize()加入某一种擦除功能，当 GC 工作时，finalize()得到了调用，图像就会被擦除。要是 GC 没有发生，那么这个图像就会

被一直保存下来。

一旦垃圾回收器准备好释放对象占用的存储空间，首先会去调用 finalize()方法进行一些必要的清理工作。只有到下一次再进行垃圾回收动作的时候，才会真正释放这个对象所占用的内存空间。

在普通的清除工作中，为清除一个对象，那个对象的用户必须在希望进行清除的地点调用一个清除方法。这与 C++"析构函数"的概念稍有抵触。在 C++中，所有对象都会破坏（清除）。或者换句话说，所有对象都"应该"破坏。若将 C++对象创建成一个本地对象，比如在

堆栈中创建（在 Java 中是不可能的，Java 都在堆中），那么清除或破坏工作就会在"结束花括号"所代表的、创建这个对象的作用域的末尾进行。若对象是用 new 创建的（类似于 Java），那么当程序员调用 C++的 delete 命令时（Java 没有这个命令），就会调用相应的析构函数。若程序员忘记了，那么永远不会调用析构函数，我们最终得到的将是一个内存"漏洞"，另外还包括对象的其他部分永远不会得到清除。

相反，Java 不允许我们创建本地（局部）对象--无论如何都要使用 new。但在 Java 中，没有"delete"命令来释放对象，因为垃圾回收器会帮助我们自动释放存储空间。所以如果站在比较简化的立场，我们可以说正是由于存在垃圾回收机制，所以 Java 没有析构函数。然而，随着以后学习的深入，就会知道垃圾收集器的存在并不能完全消除对析构函数的需要，或者说不能消除对析构函数代表的那种机制的需要（原因见下一段。另外 finalize()函数是在垃圾回收器准备释放对象占用的存储空间的时候被调用的，绝对不能直接调用 finalize()，所以应尽量避免用它）。若希望执行除释放存储空间之外的其他某种形式的清除工作，仍然必须调用 Java 中的一个方法。它等价于 C++的析构函数，只是没后者方便。

在 C++中所有的对象运用 delete()一定会被销毁，而 JAVA 里的对象并非总会被垃圾回收器回收。In another word, 1 对象可能不被垃圾回收，2 垃圾回收并不等于“析构”，3 垃圾回收只与内存有关。也就是说，并不是如果一个对象不再被使用，是不是要在 finalize()中释放这个对象中含有的其它对象呢？不是的。因为无论对象是如何创建的，垃圾回收器都会负责释放那些对象占有的内存。

5. 触发主 GC（Garbage Collector）的条件

JVM 进行次 GC 的频率很高,但因为这种 GC 占用时间极短,所以对系统产生的影响不大。更值得关注的是主 GC 的触发条件,因为它对系统影响很明显。总的来说,有两个条件会触发主 GC:

1)当应用程序空闲时,即没有应用线程在运行时,GC 会被调用。因为 GC 在优先级最低的线程中进行,所以当应用忙时,GC 线程就不会被调用,但以下条件除外。

2)Java 堆内存不足时,GC 会被调用。当应用线程在运行,并在运行过程中创建新对象,若这时内存空间不足,JVM 就会强制地调用 GC 线程,以便回收内存用于新的分配。若 GC 一次之后仍不能满足内存分配的要求,JVM 会再进行两次 GC 作进一步的尝试,若仍无法满足要求,则 JVM 将报“out of memory”的错误,Java 应用将停止。

由于是否进行主 GC 由 JVM 根据系统环境决定,而系统环境在不断的变化当中,所以主 GC 的运行具有不确定性,无法预计它何时必然出现,但可以确定的是对一个长期运行的应用来说,其主 GC 是反复进行的。

6. 减少 GC 开销的措施

根据上述 GC 的机制,程序的运行会直接影响系统环境的变化,从而影响 GC 的触发。若不针对 GC 的特点进行设计和编码,就会出现内存驻留等一系列负面影响。为了避免这些影响,基本的原则就是尽可能地减少垃圾和减少 GC 过程中的开销。具体措施包括以下几个方面:

(1)不要显式调用 System.gc()

此函数建议 JVM 进行主 GC,虽然只是建议而非一定,但很多情况下它会触发主 GC,从而增加主 GC 的频率,也即增加了间歇性停顿的次数。

(2)尽量减少临时对象的使用

临时对象在跳出函数调用后,会成为垃圾,少用临时变量就相当于减少了垃圾的产生,从而延长了出现上述第二个触发条件出现的时间,减少了主 GC 的机会。

(3)对象不用时最好显式置为 Null

一般而言,为 Null 的对象都会被作为垃圾处理,所以将不用的对象显式地设为 Null,有利于 GC 收集器判定垃圾,从而提高了 GC 的效率。

(4)尽量使用 StringBuffer,而不用 String 来累加字符串

由于 String 是固定长的字符串对象,累加 String 对象时,并非在一个 String 对象中扩增,而是重新创建新的 String 对象,如 Str5=Str1+Str2+Str3+Str4,这条语句执行过程中会产生多个垃圾对象,因为对次作“+”操作时都必须创建新的 String 对象,但这些过渡对象对系统来说是没有实际意义的,只会增加更多的垃圾。避免这种情况可以改用 StringBuffer 来累加字符串,因 StringBuffer 是可变长的,它在原有基础上进行扩增,不会产生中间对象。

(5)能用基本类型如 Int,Long,就不用 Integer,Long 对象

基本类型变量占用的内存资源比相应对象占用的少得多,如果没有必要,最好使用基本变量。

(6)尽量少用静态对象变量

静态变量属于全局变量,不会被 GC 回收,它们会一直占用内存。

(7)分散对象创建或删除的时间

集中在短时间内大量创建新对象,特别是大对象,会导致突然需要大量内存,JVM 在面临这种情况时,只能进行主 GC,以回收内存或整合内存碎片,从而增加主 GC 的频率。集中删除对象,道理也是一样的。它使得突然出现了大量的垃圾对象,空闲空间必然减少,从而大大增加了下一次创建新对象时强制主 GC 的机会。

下面这个例子向大家展示了垃圾收集所经历的过程，并对前面的陈述进行了总结。

```
1.  class Chair {
2.      static boolean gcrun = false;
3.      static boolean f = false;
4.      static int created = 0;
5.      static int finalized = 0;
6.      int i;
7.      Chair() {
8.          i = ++created;
9.          if(created == 47)
10.             System.out.println("Created 47");
11.     }
12.     protected void finalize() {
13.         if(!gcrun) {
14.             gcrun = true;
15.             System.out.println("Beginning to finalize after " + created + " C
    hairs have been created");
16.         }
17.         if(i == 47) {
18.             System.out.println("Finalizing Chair #47, " +"Setting flag to sto
    p Chair creation");
19.             f = true;
20.         }
21.         finalized++;
22.         if(finalized >= created)
23.             System.out.println("All " + finalized + " finalized");
24.     }
25. }
26.
27. public class Garbage {
28.     public static void main(String[] args) {
29.     if(args.length == 0) {
30.         System.err.println("Usage: /n" + "<a href="http://lib.csdn.net/base/j
    ava" class='replace_word' title="Java 知识库
    " target='_blank' style='color:#df3434; font-weight:bold;'>Java </a>Garbage
    before/n or:/n" + "java Garbage after");
31.         return;
32.     }
33.     while(!Chair.f) {
34.         new Chair();
35.         new String("To take up space");
36.     }
37.
    System.out.println("After all Chairs have been created:/n" + "total created
    = " + Chair.created +
```

```
38.     ",  total finalized = " + Chair.finalized);
39.     if(args[0].equals("before")) {
40.         System.out.println("gc():");
41.         System.gc();
42.         System.out.println("runFinalization():");
43.         System.runFinalization();
44.     }
45.     System.out.println("bye!");
46.     if(args[0].equals("after"))
47.         System.runFinalizersOnExit(true);
48.     }
49. }
```

上面这个程序创建了许多 Chair 对象，而且在垃圾收集器开始运行后的某些时候，程序会停止创建 Chair。由于垃圾收集器可能在任何时间运行，所以我们不能准确知道它在何时启动。因此，程序用一个名为 gcrun 的标记来指出垃圾收集器是否已经开始运行。利用第二个标记 f，Chair 可告诉 main()它应停止对象的生成。这两个标记都是在 finalize()内部设置的，它调用于垃圾收集期间。另两个 static 变量--created 以及 finalized--分别用于跟踪已创建的对象数量以及垃圾收集器已进行完收尾工作的对象数量。最后，每个 Chair 都有它自己的（非 static）int i，所以能跟踪了解它具体的编号是多少。编号为 47 的 Chair 进行完收尾工作后，标记会设为 true，最终结束 Chair 对象的创建过程。

7. 关于垃圾回收的几点补充

经过上述的说明，可以发现垃圾回收有以下的几个特点：

（1）垃圾收集发生的不可预知性：由于实现了不同的垃圾回收算法和采用了不同的收集机制，所以它有可能是定时发生，有可能是当出现系统空闲 CPU 资源时发生，也有可能是和原始的垃圾收集一样，等到内存消耗出现极限时发生，这与垃圾收集器的选择和具体的设置都有关系。

（2）垃圾收集的精确性：主要包括 2 个方面：（a）垃圾收集器能够精确标记活着的对象；（b）垃圾收集器能够精确地定位对象之间的引用关系。前者是完全地回收所有废弃对象的前提，否则就可能造成内存泄漏。而后者则是实现归并和复制等算法的必要条件。所有不可达对象都能够可靠地得到回收，所有对象都能够重新分配，允许对象的复制和对象内存的缩并，这样就有效地防止内存的支离破碎。

（3）现在有许多种不同的垃圾收集器，每种有其算法且其表现各异，既有当垃圾收集开始时就停止应用程序的运行，又有当垃圾收集开始时也允许应用程序的线程运行，还有在同一时间垃圾收集多线程运行。

（4）垃圾收集的实现和具体的 JVM 以及 JVM 的内存模型有非常紧密的关系。不同的 JVM 可能采用不同的垃圾收集，而 JVM 的内存模型决定着该 JVM 可以采用哪些类型垃圾收集。现在，HotSpot 系列 JVM 中的内存系统都采用先进的面向对象的框架设计，这使得该系列 JVM 都可以采用最先进的垃圾收集。

（5）随着技术的发展，现代垃圾收集技术提供许多可选的垃圾收集器，而且在配置每种收集器的时候又可以设置不同的参数，这就使得根据不同的应用环境获得最优的应用性能成为可能。

针对以上特点，我们在使用的时候要注意：

（1）不要试图去假定垃圾收集发生的时间，这一切都是未知的。比如，方法中的一个临时对象在方法调用完毕后就变成了无用对象，这个时候它的内存就可以被释放。

（2）Java 中提供了一些和垃圾收集打交道的类，而且提供了一种强行执行垃圾收集的方法--调用 System.gc()，但这同样是个不确定的方法。Java 中并不保证每次调用该方法就一定能够启动垃圾收集，它只不过会向 JVM 发出这样一个申请，到底是否真正执行垃圾收集，一切都是个未知数。

（3）挑选适合自己的垃圾收集器。一般来说，如果系统没有特殊和苛刻的性能要求，可以采用 JVM 的缺省选项。否则可以考虑使用有针对性的垃圾收集器，比如增量收集器就比较适合实时性要求较高的系统之中。系统具有较高的配置，有比较多的闲置资源，可以考虑使用并行标记/清除收集器。

（4）关键的也是难把握的问题是内存泄漏。良好的编程习惯和严谨的编程态度永远是最重要的，不要让自己的一个小错误导致内存出现大漏洞。

（5）尽早释放无用对象的引用。大多数程序员在使用临时变量的时候，都是让引用变量在退出活动域(scope)后，自动设置为 null，暗示垃圾收集器来收集该对象，还必须注意该引用的对象是否被监听，如果有，则要去掉监听器，然后再赋空值。