一个优秀的Java程序员必须了解GC的工作原理、如何优化GC的性能、如何与GC进行有限的交互，因为有一些应用程序对性能要求较高，例如嵌入式系统、实时系统等，只有全面提升内存的管理效率 ，才能提高整个应用程序的性能。本篇文章首先简单介绍GC的工作原理之后，然后再对GC的几个关键问题进行深入探讨，最后提出一些Java程序设计建议， 从GC角度提高Java程序的性能。

# GC的基本原理

Java的内存管理实际上就是对象的管理，其中包括对象的分配和释放。

对于程序员来说，分配对象使用new关键字；释放对象时，只要将对象所有引用赋值为null，让程序不能够再访问到这个对象，我们称该对象为"不可达的".GC将负责回收所有"不可达"对象的内存空间。

对于GC来说，当程序员创建对象时，GC就开始监控这个对象的地址、大小以及使用情况。通常，GC采用有向图的方式记录和管理堆（heap）中的所有 对象（详见 参考资料1 ）。通过这种方式确定哪些对象是"可达的"，哪些对象是"不可达的".当GC确定一些对象为"不可达"时，GC就有责任回收这些内存空间。但是，为了保证 GC能够在不同平台实现的问题，Java规范对GC的很多行为都没有进行严格的规定。例如，对于采用什么类型的回收算法、什么时候进行回收等重要问题都没 有明确的规定。因此，不同的JVM的实现者往往有不同的实现算法。这也给Java程序员的开发带来行多不确定性。本文研究了几个与GC工作相关的问题，努力减少这种不确定性给Java程序带来的负面影响。

# 增量式GC（ Incremental GC ）

GC在JVM中通常是由一个或一组进程来实现的，它本身也和用户程序一样占用heap空间，运行时也占用CPU.当GC进程运行时，应用程序停止运 行。因此，当GC运行时间较长时，用户能够感到Java程序的停顿，另外一方面，如果GC运行时间太短，则可能对象回收率太低，这意味着还有很多应该回收 的对象没有被回收，仍然占用大量内存。因此，在设计GC的时候，就必须在停顿时间和回收率之间进行权衡。一个好的GC实现允许用户定义自己所需要的设置， 例如有些内存有限有设备，对内存的使用量非常敏感，希望GC能够准确的回收内存，它并不在意程序速度的放慢。另外一些实时网络游戏，就不能够允许程序有长 时间的中断。增量式GC就是通过一定的回收算法，把一个长时间的中断，划分为很多个小的中断，通过这种方式减少GC对用户程序的影响。虽然，增量式GC在 整体性能上可能不如普通GC的效率高，但是它能够减少程序的最长停顿时间。

　　Sun JDK提供的HotSpot JVM就能支持增量式GC.HotSpot JVM缺省GC方式为不使用增量GC，为了启动增量GC，我们必须在运行Java程序时增加-Xincgc的参数。HotSpot JVM增量式GC的实现是采用Train GC算法。它的基本想法就是，将堆中的所有对象按照创建和使用情况进行分组（分层），将使用频繁高和具有相关性的对象放在一队中，随着程序的运行，不断对 组进行调整。当GC运行时，它总是先回收最老的（最近很少访问的）的对象，如果整组都为可回收对象，GC将整组回收。这样，每次GC运行只回收一定比例的 不可达对象，保证程序的顺畅运行。

# 详解finalize函数

finalize是位于Object类的一个方法，该方法的访问修饰符为protected，由于所有类为Object的子类，因此用户类很容易访问 到这个方法。由于，finalize函数没有自动实现链式调用，我们必须手动的实现，因此finalize函数的最后一个语句通常是 super.finalize（）。通过这种方式，我们可以实现从下到上实现finalize的调用，即先释放自己的资源，然后再释放父类的资源。

根据Java语言规范，JVM保证调用finalize函数之前，这个对象是不可达的，但是JVM不保证这个函数一定会被调用。另外，规范还保证finalize函数最多运行一次。

很多Java初学者会认为这个方法类似与C++中的析构函数，将很多对象、资源的释放都放在这一函数里面。其实，这不是一种很好的方式。原因有三，其 一，GC为了能够支持finalize函数，要对覆盖这个函数的对象作很多附加的工作。其二，在finalize运行完成之后，该对象可能变成可达 的，GC还要再检查一次该对象是否是可达的。因此，使用finalize会降低GC的运行性能。其三，由于GC调用finalize的时间是不确定的，因 此通过这种方式释放资源也是不确定的。

通常，finalize用于一些不容易控制、并且非常重要资源的释放，例如一些I/O的操作，数据的连接。这些资源的释放对整个应用程序是非常关键的。在这种情况下，程序员应该以通过程序本身管理（包括释放）这些资源为主，以finalize函数释放资源方式为辅，形成一种双保险的管理机制，而不应该仅仅依靠finalize来释放资源。

　　下面给出一个例子说明，finalize函数被调用以后，仍然可能是可达的，同时也可说明一个对象的finalize只可能运行一次。   
  
class MyObject{   
　　 Test main; //记录Test对象，在finalize中时用于恢复可达性   
　　 public MyObject(Test t)   
　　 {   
　　 main=t; //保存Test 对象   
　　 }   
　　 protected void finalize()   
　　 {   
　　 main.ref=this;// 恢复本对象，让本对象可达   
　　 System.out.println("This is finalize");//用于测试finalize只运行一次   
　　 }   
　　}   
  
　　class Test {   
　　 MyObject ref;   
　　 public static void main(String[] args) {   
　　 Test test=new Test();   
　　 test.ref=new MyObject(test);   
　　 test.ref=null; //MyObject对象为不可达对象，finalize将被调用   
　　 System.gc();   
　　 if (test.ref!=null) System.out.println("My Object还活着");   
　　 }   
　　}   
  
　　运行结果：   
  
This is finalize   
MyObject还活着   
  
　　此例子中，需要注意的是虽然MyObject对象在finalize中变成可达对象，但是下次回收时候，finalize却不再被调用，因为finalize函数最多只调用一次。

程序如何与GC进行交互   
  
　　Java2增强了内存管理功能， 增加了一个java.lang.ref包，其中定义了三种引用类。这三种引用类分别为SoftReference、WeakReference和 PhantomReference.通过使用这些引用类，程序员可以在一定程度与GC进行交互，以便改善GC的工作效率。这些引用类的引用强度介于可达对 象和不可达对象之间。   
  
　　创建一个引用对象也非常容易，例如如果你需要创建一个Soft Reference对象，那么首先创建一个对象，并采用普通引用方式（可达对象）；然后再创建一个SoftReference引用该对象；最后将普通引用 设置为null.通过这种方式，这个对象就只有一个Soft Reference引用。同时，我们称这个对象为Soft Reference 对象。   
  
　　Soft Reference的主要特点是据有较强的引用功能。只有当内存不够的时候，才进行回收这类内存，因此在内存足够的时候，它们通常不被回收。另外，这些引 用对象还能保证在Java抛出OutOfMemory 异常之前，被设置为null.它可以用于实现一些常用图片的缓存，实现Cache的功能，保证最大限度的使用内存而不引起OutOfMemory.以下给 出这种引用类型的使用伪代码；   
  
//申请一个图像对象   
　　Image image=new Image();//创建Image对象   
　　…   
　　//使用 image   
　　…   
　　//使用完了image，将它设置为soft 引用类型，并且释放强引用；   
　　SoftReference sr=new SoftReference(image);   
　　image=null;   
　　 …   
　　 //下次使用时   
　　 if (sr!=null) image=sr.get();   
　　 else{   
　　 //由于GC由于低内存，已释放image，因此需要重新装载；   
　　 image=new Image();   
　　sr=new SoftReference(image);   
　　}   
  
　　Weak引用对象与Soft引用对象的最大不同就在于：GC在进行回收时，需要通过算法检查是否回收Soft引用对象，而对于Weak引用对象，GC 总是进行回收。Weak引用对象更容易、更快被GC回收。虽然，GC在运行时一定回收Weak对象，但是复杂关系的Weak对象群常常需要好几次GC的运 行才能完成。Weak引用对象常常用于Map结构中，引用数据量较大的对象，一旦该对象的强引用为null时，GC能够快速地回收该对象空间。   
  
　　Phantom引用的用途较少，主要用于辅助finalize函数的使用。Phantom对象指一些对象，它们执行完了finalize函数，并为不 可达对象，但是它们还没有被GC回收。这种对象可以辅助finalize进行一些后期的回收工作，我们通过覆盖Reference的clear（）方法， 增强资源回收机制的灵活性。

# 一些Java编码的建议

根据GC的工作原理，我们可以通过一些技巧和方式，让GC运行更加有效率，更加符合应用程序的要求。以下就是一些程序设计的几点建议。   
  
　　1.最基本的建议就是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.我们在使用这种方式时候，必须特别注意一些复杂的对象图，例如数组，队列，树，图等，这些对象之间有相互引用关系较为复杂。对于这类对象，GC 回收它们一般效率较低。如果程序允许，尽早将不用的引用对象赋为null.这样可以加速GC的工作。

　　2.尽量少用finalize函数。finalize函数是Java提供给程序员一个释放对象或资源的机会。但是，它会加大GC的工作量，因此尽量少采用finalize方式回收资源。   
  
　　3.如果需要使用经常使用的图片，可以使用soft应用类型。它可以尽可能将图片保存在内存中，供程序调用，而不引起OutOfMemory.   
  
　　4.注意集合数据类型，包括数组，树，图，链表等数据结构，这些数据结构对GC来说，回收更为复杂。另外，注意一些全局的变量，以及一些静态变量。这些变量往往容易引起悬挂对象（dangling reference），造成内存浪费。   
  
　　5.当程序有一定的等待时间，程序员可以手动执行System.gc（），通知GC运行，但是Java语言规范并不保证GC一定会执行。使用增量式GC可以缩短Java程序的暂停时间。