1．垃圾收集算法的核心思想  
Java语言建立了垃圾收集机制，用以跟踪正在使用的对象和发现并回收不再使用（引用）的对象。该机制可以有效防范动态内存分配中可能发生的两个危 险：因内存垃圾过多而引发的内存耗尽，以及不恰当的内存释放所造成的内存非法引用。  
垃 圾收集算法的核心思想是：对虚拟机可用内存空间，即堆空间中的对象进行识别，如果对象正在被引用，那么称其为存活对象，反之，如果对象不再被引 用，则为 垃圾对象，可以回收其占据的空间，用于再分配。垃圾收集算法的选择和垃圾收集系统参数的合理调节直接影响着系统性能，因此需要开发人员做比较深入 的了 解。  
2．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。  
②Java 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。  
3．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（详见blog另一篇文章JAVA中String与 StringBuffer）  
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。  
4．gc与finalize方法  
⑴gc方法请求垃圾回收  
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。需要注意的是，调用 System.gc()也仅 仅是一个请求。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。  
⑵finalize方法透视垃圾收集器的运行  
在JVM垃圾收集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化该对象释放资源，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。  
因此，当对象即将被销毁时，有时需要做一些善后工作。可以把这些操作写在finalize()方法里。  
protected void finalize()  
{  
// finalization code here  
}  
⑶代码示例  
class Garbage  
{  
    int index;  
    static int count;  
    Garbage()  
    {  
        count++;  
        System.out.println("object "+count+" construct");  
        setID(count);  
    }  
     
    void setID(int id)  
    {  
        index=id;  
    }  
     
    protected void finalize() //重写finalize方法  
    {  
        System.out.println("object "+index+" is reclaimed");  
    }  
     
    public static void main(String[] args)  
    {  
        new Garbage();  
        new Garbage();  
        new Garbage();  
        new Garbage();  
        System.gc(); //请求运行垃圾收集器  
    }  
}  
5．Java 内存泄漏  
由 于采用了垃圾回收机制，任何不可达对象（对象不再被引用）都可以由垃圾收集线程回收。因此通常说的Java 内存泄漏其实是指无意识的、非故意的对象引 用，或者无意识的对象保持。无意识的对象引用是指代码的开发人员本来已经对对象使用完毕，却因为编码的错误而意 外地保存了对该对象的引用（这个引用的存 在并不是编码人员的主观意愿），从而使得该对象一直无法被垃圾回收器回收掉，这种本来以为可以释放掉的却最终未能 被释放的空间可以认为是被“泄漏了”。  
考 虑下面的程序,在ObjStack类中,使用push和pop方法来管理堆栈中的对象。两个方法中的索引(index)用于指示堆栈中下一个可用 位置。 push方法存储对新对象的引用并增加索引值,而pop方法减小索引值并返回堆栈最上面的元素。在main方法中,创建了容量为64的栈,并64次 调用 push方法向它添加对象,此时index的值为64,随后又32次调用pop方法,则index的值变为32,出栈意味着在堆栈中的空间应该被收 集。 但事实上,pop方法只是减小了索引值,堆栈仍然保持着对那些对象的引用。故32个无用对象不会被GC回收,造成了内存渗漏。  
  
public class ObjStack {  
private Object[] stack;  
private int index;  
ObjStack(int indexcount) {  
stack = new Object[indexcount];  
index = 0;  
}  
  
public void push(Object obj) {  
stack[index] = obj;  
index++;  
}  
public Object pop() {  
index--;  
return stack[index];  
}  
}  
  
public class Pushpop {  
public static void main(String[] args) {  
int i = 0;  
Object tempobj;  
ObjStack stack1 = new ObjStack(64);//new一个ObjStack对象，并调用有参构造函数。分配stack Obj数组的空间大小为64，可以存64个对象，从0开始存储。  
while (i < 64)  
{  
   tempobj = new Object();//循环new Obj对象，把每次循环的对象一一存放在stack Obj数组中。  
   stack1.push(tempobj);  
   i++;  
   System.out.println("第" + i + "次进栈" + "\t");  
}  
while (i > 32)  
{  
   tempobj = stack1.pop();//这里造成了空间的浪费。  
   //正确的pop方法可改成如下所指示,当引用被返回后,堆栈删除对他们的引用,因此垃圾收集器在以后可以回收他们。  
   /\*  
    \* public Object pop() {index - -;Object temp = stack [index];stack [index]=null;return temp;}  
    \*/  
   i--;  
   System.out.println("第" + (64 - i) + "次出栈" + "\t");  
}  
}  
}  
  
如何消除内存泄漏  
　 　虽然Java虚拟机(JVM)及其垃圾收集器(garbage collector，GC)负责管理大多数的内存任务，Java软件程序中还是有可能出 现内存泄漏。实际上，这在大型项目中是一个常见的问题。避免内存泄 漏的第一步是要弄清楚它是如何发生的。本文介绍了编写Java代码的一些常见的内存泄 漏陷阱，以及编写不泄漏代码的一些最佳实践。一旦发生了内存泄漏，要 指出造成泄漏的代码是非常困难的。因此本文还介绍了一种新工具，用来诊断泄漏并指出 根本原因。该工具的开销非常小，因此可以使用它来寻找处于生产中的系统 的内存泄漏。  
垃圾收集器的作用  
　　虽然垃圾收集器处理了大多数内 存管理问题，从而使编程人员的生活变得更轻松了，但是编程人员还是可能犯错而导致出现内存问题。简单地说，GC循 环地跟踪所有来自“根”对象（堆栈对 象、静态对象、JNI句柄指向的对象，诸如此类）的引用，并将所有它所能到达的对象标记为活动的。程序只可以操纵这些 对象；其他的对象都被删除了。因为 GC使程序不可能到达已被删除的对象，这么做就是安全的。  
　　虽然内存管理可以说是自动化的，但是这并不能使编程人员免受思考内存管理问题之苦。 例如，分配（以及释放）内存总会有开销，虽然这种开销对编程 人员来说是不可见的。创建了太多对象的程序将会比完成同样的功能而创建的对象却比较少的程序 更慢一些（在其他条件相同的情况下）。  
　　而且，与本文更为密切相关的是，如果忘记“释放”先前分配的内存，就可能造成内存泄漏。如果程序保留对 永远不再使用的对象的引用，这些对象将会 占用并耗尽内存，这是因为自动化的垃圾收集器无法证明这些对象将不再使用。正如我们先前所说的，如果存在一个对 对象的引用，对象就被定义为活动的，因此不 能删除。为了确保能回收对象占用的内存，编程人员必须确保该对象不能到达。这通常是通过将对象字段设置为 null或者从集合(collection)中移 除对象而完成的。但是，注意，当局部变量不再使用时，没有必要将其显式地设置为null。对这些变量的 引用将随着方法的退出而自动清除。  
　　概括地说，这就是内存托管语言中的内存泄漏产生的主要原因：保留下来却永远不再使用的对象引用。  
典型泄漏  
　　既然我们知道了在Java中确实有可能发生内存泄漏，就让我们来看一些典型的内存泄漏及其原因。  
全局集合  
　　在大的应用程序中有某种全局的数据储存库是很常见的，例如一个JNDI树或一个会话表。在这些情况下，必须注意管理储存库的大小。必须有某种机 制从储存库中移除不再需要的数据。  
　　这可能有多种方法，但是最常见的一种是周期性运行的某种清除任务。该任务将验证储存库中的数据，并移除任何不再需要的数据。  
　　另一种管理储存库的方法是使用反向链接(referrer)计数。然后集合负责统计集合中每个入口的反向链接的数目。这要求反向链接告诉集合何 时会退出入口。当反向链接数目为零时，该元素就可以从集合中移除了。  
缓存  
　　缓存是一种数据结构，用于快速查找已经执行的操作的结果。因此，如果一个操作执行起来很慢，对于常用的输入数据，就可以将操作的结果缓存，并在 下次调用该操作时使用缓存的数据。  
　　缓存通常都是以动态方式实现的，其中新的结果是在执行时添加到缓存中的。典型的算法是：  
检查结果是否在缓存中，如果在，就返回结果。   
如果结果不在缓存中，就进行计算。   
将计算出来的结果添加到缓存中，以便以后对该操作的调用可以使用。   
　　该算法的问题（或者说是潜在的内存泄漏）出在最后一步。如果调用该操作时有相当多的不同输入，就将有相当多的结果存储在缓存中。很明显这不是正确的方 法。  
　　为了预防这种具有潜在破坏性的设计，程序必须确保对于缓存所使用的内存容量有一个上限。因此，更好的算法是：  
检查结果是否在缓存中，如果在，就返回结果。   
如果结果不在缓存中，就进行计算。   
如果缓存所占的空间过大，就移除缓存最久的结果。   
将计算出来的结果添加到缓存中，以便以后对该操作的调用可以使用。   
　　通过始终移除缓存最久的结果，我们实际上进行了这样的假设：在将来，比起缓存最久的数据，最近输入的数据更有可能用到。这通常是一个不错的假设。  
　　新算法将确保缓存的容量处于预定义的内存范围之内。确切的范围可能很难计算，因为缓存中的对象在不断变化，而且它们的引用包罗万象。为缓存设置 正确的大小是一项非常复杂的任务，需要将所使用的内存容量与检索数据的速度加以平衡。  
　　解决这个问题的另一种方法是使用java.lang.ref.SoftReference类跟踪缓存中的对象。这种方法保证这些引用能够被移 除，如果虚拟机的内存用尽而需要更多堆的话。  
ClassLoader  
　 　Java ClassLoader结构的使用为内存泄漏提供了许多可乘之机。正是该结构本身的复杂性使ClassLoader在内存泄漏方面存在如此多 的问题。 ClassLoader的特别之处在于它不仅涉及“常规”的对象引用，还涉及元对象引用，比如：字段、方法和类。这意味着只要有对字段、方法、 类或 ClassLoader的对象的引用，ClassLoader就会驻留在JVM中。因为ClassLoader本身可以关联许多类及其静态字段，所 以就有 许多内存被泄漏了。  
确定泄漏的位置  
　　通常发生内存泄漏的第一个迹象是：在应用程序中出现了OutOfMemoryError。 这通常发生在您最不愿意它发生的生产环境中，此时几乎 不能进行调试。有可能是因为测试环境运行应用程序的方式与生产系统不完全相同，因而导致泄漏只出现 在生产中。在这种情况下，需要使用一些开销较低的工具来 监控和查找内存泄漏。还需要能够无需重启系统或修改代码就可以将这些工具连接到正在运行的系统 上。可能最重要的是，当进行分析时，需要能够断开工具而保持 系统不受干扰。  
　　虽然OutOfMemoryError通常都是内存泄漏的信号， 但是也有可能应用程序确实正在使用这么多的内存；对于后者，或者必须增加JVM 可用的堆的数量，或者对应用程序进行某种更改，使它使用较少的内存。但 是，在许多情况下，OutOfMemoryError都是内存泄漏的信号。一种查明 方法是不间断地监控GC的活动，确定内存使用量是否随着时间增加。如 果确实如此，就可能发生了内存泄漏。  
  
  
  
非常多人在谈论内存泄露问题，当然对于c/c++来说，这个应该是老掉牙的问题，不过非常多Java人员也越来越多得讨论这个问题，我这里写个小结，希望 对大家有一定的参考价值。   
  
　　内存泄漏的慨念   
  
　　1.c/c++是程式员自己管理内存，Java内存是由GC自动回收的。   
  
　　我虽然不是非常熟悉C++，不过这个应该没有犯常识性错误吧。   
  
　　2.什么是内存泄露?   
  
　　内存泄露是指系统中存在无法回收的内存，有时候会造成内存不足或系统崩溃。   
  
　　在C/C++中分配了内存不释放的情况就是内存泄露。   
  
　　3.Java存在内存泄露   
  
　　我们必须先承认这个，才能接着讨论。虽然Java存在内存泄露，不过基本上不用非常关心他，特别是那些对代码本身就不讲究的就更不要去关心这个了。   
  
　　Java中的内存泄露当然是指：存在无用不过垃圾回收器无法回收的对象。而且即使有内存泄露问题存在，也不一定会表现出来。   
  
　　4.Java中参数都是传值的。   
  
　　对于基本类型，大家基本上没有异议，不过对于引用类型我们也不能有异议。   
  
　　Java内存泄露情况   
  
　　JVM回收算法是非常复杂的，我也不知道他们怎么实现的，不过我只知道他们要实现的就是：对于没有被引用的对象是能回收的。所以你要造成内存泄露就要 做到： 中国.网管联盟   
  
　　持有对无用对象的引用!   
  
　　不要以为这个非常容易做到，既然无用，你怎么还会持有他的引用? 既然你还持有他，他怎么会是无用的呢?   
  
　　我实在想不到比那个堆栈更经典的例子了,以致于我还要引用别人的例子，下面的例子不是我想到的，是书上看到的，当然如果没有在书上看到，可能过一段时 间我自己也想的到，可是那时我说是我自己想到的也没有人相信的。   
  
public class Stack {   
　private Object[] elements=new Object[10];   
　private int size = 0;   
  
　public void push(Object e){   
　　ensureCapacity();   
　　elements[size++] = e;   
　}   
  
　public Object pop(){   
　　if( size == 0)   
　　　throw new EmptyStackException();   
　　　return elements[--size];   
　}   
  
private void ensureCapacity(){   
　if(elements.length == size){   
　　Object[] oldElements = elements;   
　　elements = new Object[2 \* elements.length+1];   
　　System.arraycopy(oldElements,0, elements, 0, size);   
　}   
}   
}   
  
　　上面的原理应该非常简单，如果堆栈加了10个元素，然后全部弹出来，虽然堆栈是空的，没有我们要的东西，不过这是个对象是无法回收的，这个才符合了内 存泄露的两个条件：无用，无法回收。 中国网管联盟   
  
　　不过就是存在这样的东西也不一定会导致什么样的后果，如果这个堆栈用的比较少，也就浪费了几个K内存而已，反正我们的内存都上G了，哪里会有什么影 响，再说这个东西非常快就会被回收的，有什么关系。下面看两个例子。   
  
　　例子1   
  
public class Bad{   
　public static Stack s=Stack();   
　　static{   
　　　s.push(new Object());   
　　　s.pop(); //这里有一个对象发生内存泄露   
　　　s.push(new Object()); //上面的对象能被回收了，等于是自愈了   
　　}   
}   
  
　 　因为是static，就一直存在到程式退出，不过我们也能看到他有自愈功能，就是说如果你的Stack最多有100个对象，那么最多也就只有100 个 对象无法被回收其实这个应该非常容易理解，Stack内部持有100个引用，最坏的情况就是他们都是无用的，因为我们一旦放新的进取，以前的引用自然 消 失！   
  
　　例子2   
  
public class NotTooBad{   
　public void doSomething(){   
　　Stack s=new Stack();   
　　s.push(new Object());   
　　//other code   
　　s.pop();//这里同样导致对象无法回收,内存泄露.   
　}//退出方法,s自动无效,s能被回收,Stack内部的引用自然没了,所以 www\_bitscn\_com   
　//这里也能自愈,而且能说这个方法不存在内存泄露问题,不过是晚一点   
　//交给GC而已,因为他是封闭的,对外不开放,能说上面的代码99.9999%的   
　//情况是不会造成所有影响的,当然你写这样的代码不会有什么坏的影响,不过   
　//绝对能说是垃圾代码!没有矛盾吧,我在里面加一个空的for循环也不会有   
　//什么太大的影响吧,你会这么做吗?   
}   
  
　 　上面两个例子都不过是小打小闹,不过C/C++中的内存泄露就不是Bad了,而是Worst了。他们如果一处没有回收就永远无法回收,频繁的调用这 个 方法内存不就用光了!因为Java更有自愈功能(我自己起的名字,还没申请专利),所以Java的内存泄露问题几乎能忽略了,不过知道的人就不要犯 了。   
  
　　不知者无罪!Java存在内存泄露,不过也不要夸大其辞。如果你对Java都不是非常熟,你根本就不用关心这个,我说过你无意中写出内存泄露的例子就 像你中一千万相同概率小,开玩笑了,其实应该是小的多的多!   
  
　　而且即使你有幸写出这样的代码，中奖了!基本上都是一包洗衣粉，不会让你发财，对系统没有什么大的影响。   
  
　　杞人忧天的情况   
  
　　1.无话可说型   
  
Object obj=new Object();   
obj=null;   
//这个完全多此一举,因为退出了作用范围，对象的引用自动消失 中国网管联盟   
//不要在你的程式中出现这样的语句，没有错，不过就是不雅观   
  
　　2.思考不对型   
  
void func(Object o){   
　o=new Object();   
　return   
}   
  
　　当我们知道Java参数是传值，就知道上面的方法什么也没错，就是申请了一个对象然后再丢给GC。因为是传值，这里的o是个调用时候的拷贝,会不会无 法回收？不就是拷贝吗，退出方法什么都没了，这个对象怎么会留的住。   
  
　　3.尽量避免型   
  
class A{   
　B b=new B(this);   
}   
class B{   
　A a;   
　B(A a)   
}   
  
　　这个存在互相引用，可能导致孤岛现象，不过这个不会造成内存泄露不过我自己觉得这个会降低GC的效率，就从我的智力来看，我觉得这种情况比一般情况难 以判断怎么回收!当然GC比我聪明，不过应该也要动一点脑子吧。  
  
  
  
统出现内存泄漏，公司派来个牛人帮忙解决，最终成功搞定   
大学里图论有没有用呢，答案是肯定的，不然就不开这门课了，看java内存泄漏的原因就知道了，对象  
引用出现了循环调用，导致GC不能回收无用对象  
以下是收集的网络信息，转载，记录，以便学习  
Java 的一个重要优点就是通过垃圾收集器（Garbage Collection，GC）自动管理内存的回收，程序员不需要通过调用函数来释放内存。因此，很多 程序员认为Java不存在内存泄漏问题，或者认为即使 有内存泄漏也不是程序的责任，而是GC或 JVM的问题。其实，这种想法是不正确的，因为Java 也存在内存泄露，但它的表现与C++不同。  
在Java中，内存泄漏就是存在一些被分配的对象，这些对象有下面两个特点，首先，这些对象是可达的， 即在有向图中，存在通路可以与其相连；其次， 这些对象是无用的，即程序以后不会再使用这些对象。如果对象满足这两个条件，这些对象就可以判定为Java 中的内存泄漏，这些对象不会被GC所回收，然而 它却占用内存。  
在C++中，内存泄漏的范围更大一些。有些对象被分配了内存空间，然后却不可达，由于C++中没有GC，这些内存将永远收不回来。在Java中，这 些不可达的对象都由GC负责回收，因此不需要考虑这部分的内存泄露。  
注意事项：  
1、最基本的建议是尽早释放无用对象的引用。如：  
……  
A a = new A（）；  
//应用a对象  
a = null； //当使用对象a之后主动将其设置为空  
…  
注：如果a 是方法的返回值，不要做这样的处理，否则你从该方法中得到的返回值永远为空，而且这种错误不易被发现、排除  
2、尽量少用finalize函数。它会加大GC的工作量。  
3、如果需要使用经常用到的图片，可以使用soft应用类型。它尽可能把图片保存在内存中  
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。  
5、尽量避免在类的默认构造器中创建、初始化大量的对象，防止在调用其自类的构造器时造成不必要的内存资源浪费  
6、尽量避免强制系统做垃圾内存的回收，增长系统做垃圾回收的最终时间  
7、尽量避免显式申请数组空间  
8、尽量做远程方法调用类应用开发时使用瞬间值变量，除非远程调用端需要获取该瞬间值变量的值。  
9、尽量在合适的场景下使用对象池技术以提高系统性能。