# Java垃圾回收机制

综合了若干人的blog～

## 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：

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。

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。

参考：

http://blog.sina.com.cn/s/blog_544e54220100dv0q.html

http://java.chinaitlab.com/base/839279_1.html

http://blog.sina.com.cn/s/blog_5ef35aa00100ky9o.html

http://hi.baidu.com/ou_yang_/blog/item/129f1a33486e1dfe1b4cff55.html