Java垃圾回收机制

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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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
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

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
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
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
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
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
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
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
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
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
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c
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
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，
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
情况，并将选择适当算法的垃圾收集器。