https://my.oschina.net/u/1859679/blog/1548866

从这篇开始我们开始探讨一些jvm调优的问题。在jvm调优中一个离不开的重点是垃圾回收，当垃圾回收成为系统达到更高并发量的瓶颈时，我们就需要对jvm中如果进行“自动化”垃圾回收技术实施必要的监控和调节。

这是jvm优化系列第一篇，其他请看以下：

* [jvm优化必知系列——监控工具](https://my.oschina.net/u/1859679/blog/1552290)

对于调优之前，我们必须要了解其运行原理，java 的垃圾收集Garbage Collection 通常被称为“GC”，它诞生于1960年 MIT 的 Lisp 语言，经过半个多世纪，目前已经十分成熟了。因此本篇主要从这三个方面来了解:

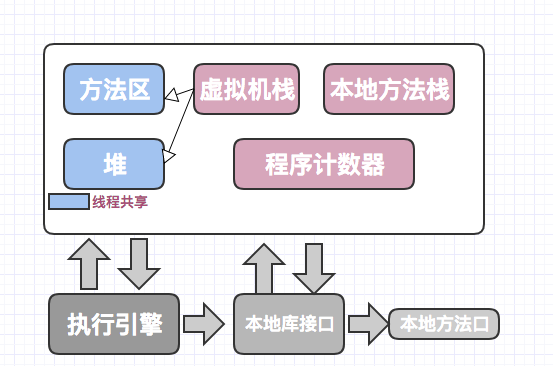
1. 哪些对象需要被回收？

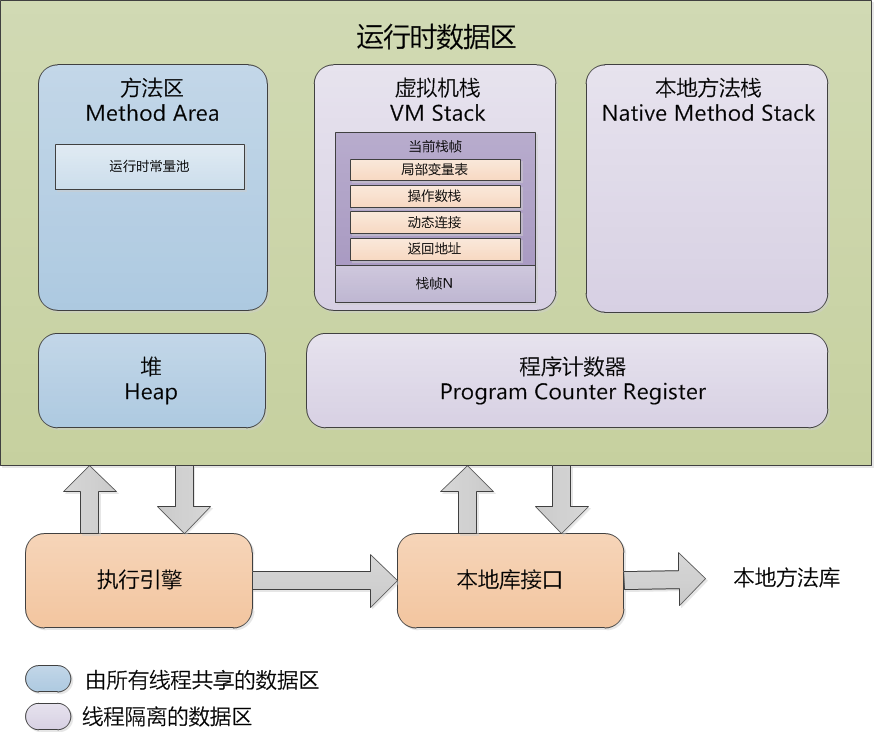
2. 什么时候回收？

3. 如何回收？

**一、谁要被回收**

java虚拟机在执行java程序的过程中会把它所管理的内存划分为若干个不同是数据区域，这些区域有各自各自的用途。主要包含以下几个部分组成：





**1、程序计数器**

程序计数器占用的内存空间我们可以忽略不计，它是每个线程所执行的字节码的行号指示器。

**2、虚拟机栈**

java的虚拟机栈是线程私有的，生命周期和线程相同。它描述的是方法执行的内存模型。同时用于存储局部变量、操作数栈、动态链接、方法出口等。

**3、本地方法栈**

本地方法栈，类似虚拟机栈，它调用的是是native方法。

**4、堆**

堆是jvm中管理内存中最大一块。它是被共享，存放对象实例。也被称为“gc堆”。垃圾回收的主要管理区域

**5、方法区**

方法区也是共享的内存区域。它主要存储已被虚拟机加载的类信息、常量、静态变量、即时编译器（jit）编译后的代码数据。

以上就是jvm在运行时期主要的内存组成，我们看到常见的内存使用不但存在于堆中，还会存在于其他区域，虽然堆的管理对程序的管理至关重要，但我们不能只局限于这一个区域，特别是当出现内存泄露的时候，我们除了要排查堆内存的情况，还得考虑虚拟机栈的以及方法区域的情况。

知道了要对谁以及那些区域进行内存管理，我还需要知道什么时候对这些区域进行垃圾回收。

**二、什么时候回收**

在垃圾回收之前，我们必须确定的一件事就是对象是否存活？这就牵扯到了判断对象是否存活的算法了。

**引用计数算法**：

给对象中添加一个引用计数器，每当有一个地方引用它时，计数器+1，当引用失效，计数器-1.任何时刻计数器为0的对象就是不可能再被使用的。

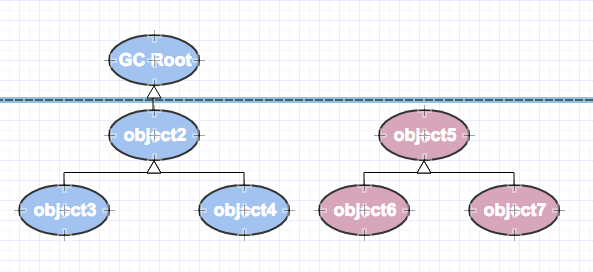
优点：实现简单，判定效率高效，被actionscript3和python中广泛应用。

缺点：无法解决对象之间的相互引用问题。java没有采纳

**可达性分析算法：**

通过一系列称为“GC Roots”的对象作为起始点，从这些节点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GCRoots没有任何引用链相连的时候，则证明此对象是不可用的。

比如如下，右侧的对象是到GCRoot时不可达的，可以判定为可回收对象。



在java中，可以作为GCRoot的对象包括以下几种：

\* 虚拟机栈中引用的对象。

\* 方法区中静态属性引用的对象。

\* 方法区中常量引用的对象。

\* 本地方法中JNI引用的对象。

基于以上，我们可以知道，当当前对象到GCRoot中不可达时候，即会满足被垃圾回收的可能。

那么是不是这些对象就非死不可，也不一定，此时只能宣判它们存在于一种“缓刑”的阶段，要真正的宣告一个对象死亡。至少要经历两次标记：

第一次：对象可达性分析之后，发现没有与GCRoots相连接，此时会被第一次标记并筛选。

第二次：对象没有覆盖finalize（）方法，或者finalize（）方法已经被虚拟机调用过，此时会被认定为没必要执行。

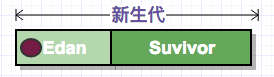
**三、如何回收**

上述的两点讲解之后，我们大概明白了，哪些对象会被回收，以及回收的依据是什么，但回收的这个工作实现起来并不简单，首先它需要扫描所有的对象，鉴别谁能够被回收，其次在扫描期间需要 ”stop the world“ 对象能被冻结，不然你刚扫描，他的引用信息有变化，你就等于白做了。

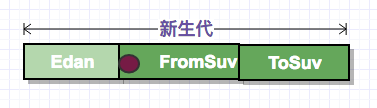
**分代回收**

我们从一个object1来说明其在分代垃圾回收算法中的回收轨迹。

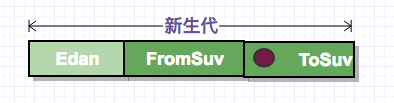
1、object1新建，出生于新生代的Eden区域。



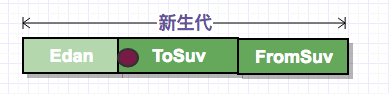
2、minor GC，object1 还存活，移动到Fromsuvivor空间，此时还在新生代。



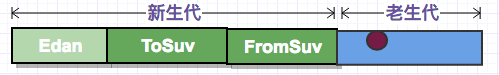
3、minor GC，object1 仍然存活，此时会通过复制算法，将object1移动到ToSuv区域，此时object1的年龄age+1。



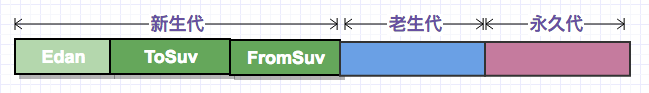
4、minor GC，object1 仍然存活，此时survivor中和object1同龄的对象并没有达到survivor的一半，所以此时通过复制算法，将fromSuv和Tosuv 区域进行互换，存活的对象被移动到了Tosuv。



5、minor GC，object1 仍然存活，此时survivor中和object1同龄的对象已经达到survivor的一半以上（toSuv的区域已经满了），object1被移动到了老年代区域。



6、object1存活一段时间后，发现此时object1不可达GcRoots，而且此时老年代空间比率已经超过了阈值,触发了majorGC（也可以认为是fullGC，但具体需要垃圾收集器来联系），此时object1被回收了。fullGC会触发 stop the world。



在以上的新生代中，我们有提到对象的age，对象存活于survivor状态下，不会立即晋升为老生代对象，以避免给老生代造成过大的影响，它们必须要满足以下条件才可以晋升：

1、minor gc 之后，存活于survivor 区域的对象的age会+1，当超过（默认）15的时候，转移到老年代。

2、动态对象，如果survivor空间中相同年龄所有的对象大小的综合和大于survivor空间的一半，年级大于或等于该年级的对象就可以直接进入老年代。

以上采用分代垃圾收集的思想，对一个对象从存活到死亡所经历的历程。期间，在新生代的时刻，会用到复制算法，在老年代时，有可能会用到标记-清楚算法（mark-sweep）算法或者标记-整理算法，这些都是垃圾回收算法基于不同区域的实现，我们看下这几种回收算法的实现原理。

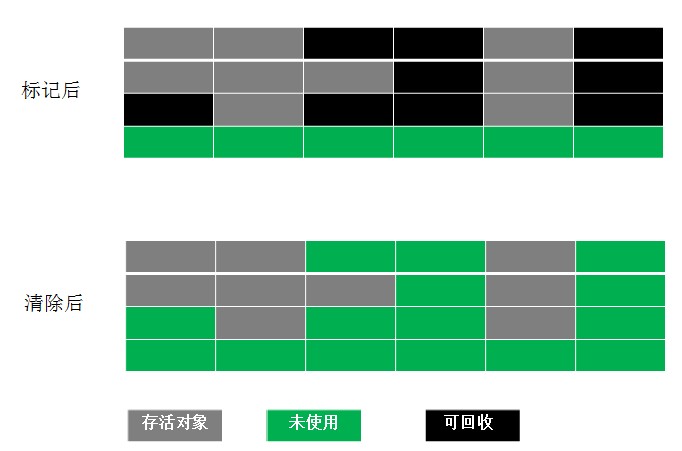
**垃圾回收算法**

**标记清除法（Mark-Sweep）**

标记清除法是垃圾回收算法的思想基础。标记清除算法将垃圾分为两个阶段：标记阶段和清除阶段。

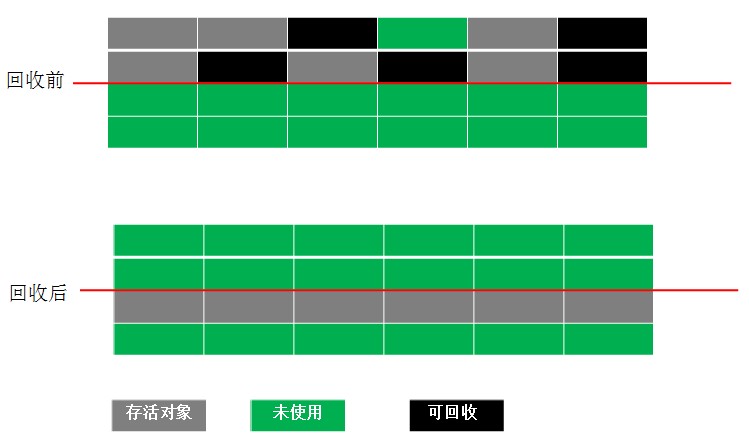
标记阶段，通过根节点，标记所有从根节点开始的可达对象，未标记过的对象就是未被引用的垃圾对象。

清除阶段，清除所有未被标记的对象。

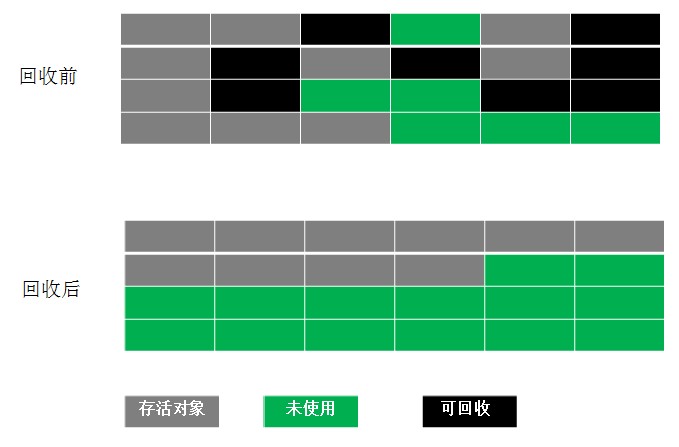


**复制算法（Copying）**

复制算法是，将原有的内存空间分为两块，每次只使用其中一块，在垃圾回收时，将正在适用的内存中存活对象复制到未使用的内存块，然后清除使用的内存块中所有的对象。



**标记压缩算法（Mark-Compact）**



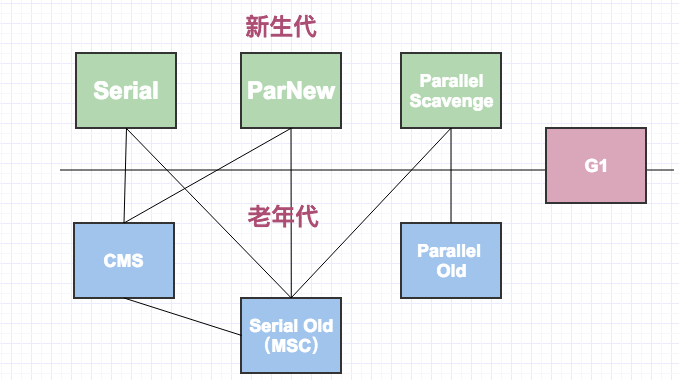
标记压缩算法是一种老年代的回收算法。

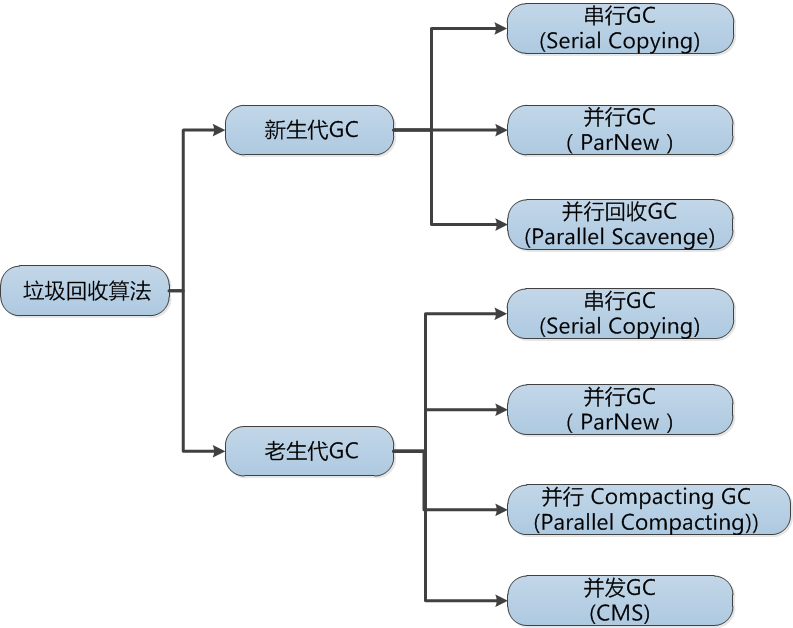
标记阶段和标记清除算法一致，对可达对象做一次标记。

清理阶段，为了避免内存碎片产生，将所有的存活对象压缩到内存的一端。

**四、垃圾收集器**

垃圾收集器是内存回收的具体实现，不同的厂商提供的垃圾收集器有很大的差别，一般的垃圾收集器都会作用于不同的分代，需要搭配使用。以下是各种垃圾收集器的组合方式：





各种组合的优缺点：

|  |  |  |  |
| --- | --- | --- | --- |
|  | 新生代GC策略 | 年老代GC策略 | 说明 |
| 组合1 | Serial | Serial Old | Serial和Serial Old都是单线程进行GC，特点就是GC时暂停所有应用线程。 |
| 组合2 | Serial | CMS+Serial Old | CMS（Concurrent Mark Sweep）是并发GC，实现GC线程和应用线程并发工作，不需要暂停所有应用线程。另外，当CMS进行GC失败时，会自动使用Serial Old策略进行GC。 |
| 组合3 | ParNew | CMS | 使用-XX:+UseParNewGC选项来开启。ParNew是Serial的并行版本，可以指定GC线程数，默认GC线程数为CPU的数量。可以使用-XX:ParallelGCThreads选项指定GC的线程数。  如果指定了选项-XX:+UseConcMarkSweepGC选项，则新生代默认使用ParNew GC策略。 |
| 组合4 | ParNew | Serial Old | 使用-XX:+UseParNewGC选项来开启。新生代使用ParNew GC策略，年老代默认使用Serial Old GC策略。 |
| 组合5 | Parallel Scavenge | Serial Old | Parallel Scavenge策略主要是关注一个可控的吞吐量：应用程序运行时间 / (应用程序运行时间 + GC时间)，可见这会使得CPU的利用率尽可能的高，适用于后台持久运行的应用程序，而不适用于交互较多的应用程序。 |
| 组合6 | Parallel Scavenge | Parallel Old | Parallel Old是Serial Old的并行版本 |
| 组合7 | G1GC | G1GC | -XX:+UnlockExperimentalVMOptions -XX:+UseG1GC        #开启  -XX:MaxGCPauseMillis =50                  #暂停时间目标  -XX:GCPauseIntervalMillis =200          #暂停间隔目标  -XX:+G1YoungGenSize=512m            #年轻代大小  -XX:SurvivorRatio=6                            #幸存区比例 |

以上优缺点来自：<http://www.importnew.com/23752.html>