**垃圾收集算法**

java语言是一门具有内存动态分配和垃圾收集技术的语言。Java堆中存放着java程序中所有的对象实例，当使用垃圾收集(Garbage Collection, GC)技术回收java堆中的内存空间时，需要考虑3件事情：

* 哪些内存空间需要回收（如何判断对象时垃圾）
* 什么时候回收内存空间（在什么时间回收垃圾对象）
* 如何回收内存空间（采用什么方法回收垃圾对象）

1. 判断对象是否已经死亡

Java堆中存放着java程序中所有的对象实例，垃圾收集器在对堆进行回收前，首先要确定堆中哪些对象是“死去”的，哪些是“活着”的。有两种方法可以用来判断对象是否存活。

* 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，计数器就加1，当引用失效时，计数器就减1。任何时刻计数器都为0的对象是不可能在被使用的，代表这个对象已经死亡，是可以被回收的。

但是引用计数算法无法对象之间循环引用的问题。

class ReferenceObject{

public Object reference = null;

public static void testReference(){

ReferenceObject objA = new ReferenceObject();

ReferenceObject objA = new ReferenceObject();

objA.reference = objB;

objB.reference = objA;

/\*

当objA和objB指向null后，堆中的两个实例对象已经不可能再被访问，按理说这两个对象已经是垃圾了，但是由于它们内部互相有一个指向对方的引用，导致它们的引用计数都不为0，所以也就无法让垃圾回收器回收这部分空间。

\*/

objA = null;

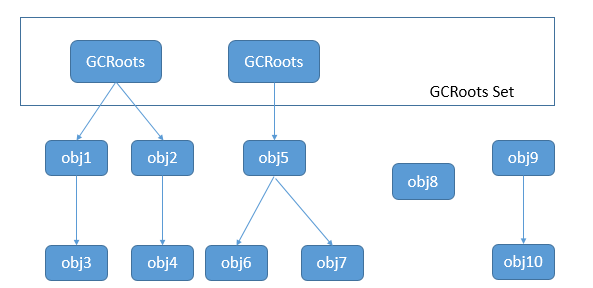
objB = null;

}

}

* 可达性分析算法

通过一系列名为“GC Roots”的对象作为起始点，从这些节点向下搜索，搜索走过的路径称为引用链(Reference Chain)，当一个对象到GC Roots没有任何引用链相连时，则证明此对象是不可用的。



在java中，可以作为GC Roots的对象包括下面几种：

* 虚拟机栈（栈帧中的局部变量区，也叫做局部变量表）中引用的对象。
* 方法区中的类静态属性引用的对象。
* 方法区中常量引用的对象。
* 本地方法栈中JNI(Native方法)引用的对象。

java中的引用类型

由强到弱分为强引用(rong Reference)，软引用(Soft Reference)，弱引用(Weak Reference)和虚引用(Phantom Reference)。

* 强引用(Strong Reference)：就是指在程序代码中普遍存在的，类似Object obj = new Object()这类的引用，只要强引用还存在，垃圾收集器永远不会回收掉被引用的对象。

强引用具备以下三个特点：

* 强引用可以直接访问目标对象；
* 强引用所指对象在任何时候都不会被系统回收。JVM宁愿抛出OOM异常也不回收强引用所指向的对象；
* 强引用可能导致内存泄露；
* 软引用(Soft Reference)：是用来描述一些还有用但并非必须的对象。对于软引用关联着的对象，在系统将要发生内存溢出异常之前，将会把这些对象列进回收范围之中进行第二次回收。如果这次回收还没有足够的内存，才会抛出内存溢出异常。在 JDK 1.2 之后，提供了 SoftReference 类来实现软引用。
* 弱引用(Weak Reference)： 用来描述非必须的对象，但是它的强度比软引用更弱一些，被弱引用关联的对象只能生存到下一次垃圾收集发送之前。当垃圾收集器工作时，无论当前内存是否足够，都会回收掉只被弱引用关联的对象。一旦一个弱引用对象被垃圾回收器回收，便会加入到一个注册引用队列中。

**软引用、弱引用都非常适合来保存那些可有可无的缓存数据。如果这么做，当系统内存不足时，这些缓存数据会被回收，不会导致内存溢出。而当内存资源充足时，这些缓存数据又可以存在相当长的时间，从而起到加速系统的作用。**

* 虚引用(Phantom Reference)：虚引用也称为幽灵引用或者幻影引用，它是最弱的一种引用关系。一个持有虚引用的对象，和没有引用几乎是一样的，随时都有可能被垃圾回收器回收。当试图通过虚引用的get()方法取得强引用时，总是会失败。并且，虚引用必须和引用队列一起使用，它的作用在于跟踪垃圾回收过程。

垃圾回收过程中的步骤

对于可达性分析算法中不可达的对象，不一定是“非死不可”的，要确定一个对象是否是死亡对象，要经历两次标记阶段。

1)、如果对象在进行可达性分析以后发现没有与GC Roots相连的引用链，则该对象被第一次标记然后对该对象进行一次筛选，筛选条件是是否有必要执行该对象的finalize方法，若对象没有覆盖finalize方法或者该finalize方法是否已经被虚拟机执行过了，则均视作不必要执行该对象的finalize方法，则判定对象已经死亡，该对象在下个回收周期将会被回收。反之，若对象覆盖了finalize方法并且该finalize方法并没有被执行过，那么，这个对象会被放置在一个叫F-Queue的队列中，之后会由虚拟机自动建立的，低优先级的Finalizer线程去执行。

2)、对F-Queue中对象进行第二次标记，如果对象在finalize方法中拯救了自己，即关联上了GC Roots引用链，如把this关键字赋值给其他变量，那么在第二次标记的时候该对象将从“即将回收”的集合中移除，如果对象还是没有拯救自己，那就会被回收。但是，一个对象只能拯救自己一次，第二次就被回收了。

* 简单来说，因为有些对象在被垃圾收集器回收之前需要做一些必要的清理工作，那么这些对象的finalize()是被重写了的，在垃圾回收时，可达性算法判断出一个对象已经没有和任何GC Roots相关联时，先对该对象进行标记，然后对该对象进行一次筛选，筛选条件是是否有必要执行该对象的finalize()方法，

如果该对象的finalize()没有被重写或者是重写但已经调用过了，则直接判定该对象是一个死亡对象了，随后会被收集。

如果该对象的finalize()被重写了并且该方法还没有被执行，说明该对象在死亡之前还需要做一些必要的清理工作，所以，就把该对象加入到F-Queue队列中，让别的线程去完成清理工作。

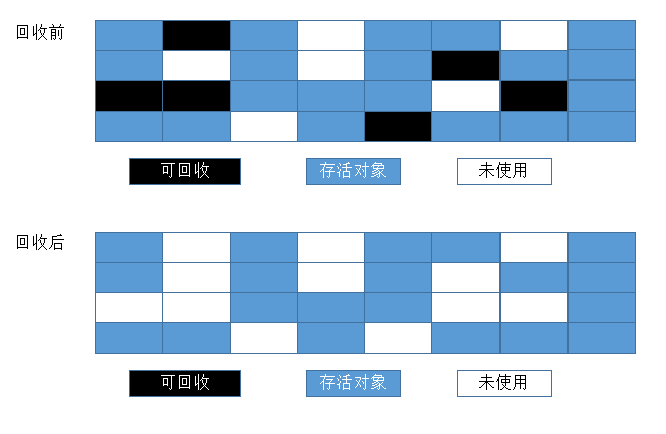
* **finalize()方法也叫收尾方法。**

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①进行一些必要的清理工作（对垃圾回收器不能处理的特殊情况进行处理）②也有可能使该对象重新被引用，对象复活。每个对象的finalize()方法只能被执行一次，第二次就会直接跳过finalize()方法，这就是为了防止出现对象无限复活，内存空间只增不减。

2、垃圾收集算法

* 标记-清除算法(Mark-Sweep)

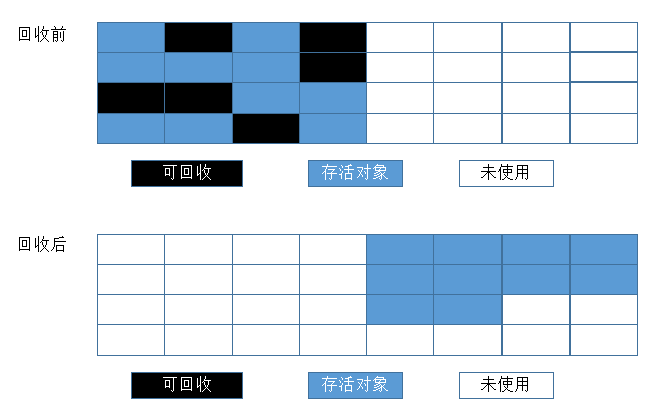
分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，标记完成后统一回收所有被标记的对象。



不足之处：

* 效率角度：标记和清除两个过程的效率都不高；
* 空间角度：标记清除后会产生大量不连续的内存碎片，内存碎片太多可能会导致以后程序运行过程中在需要分配较大对象时，无法找到足够的连续内存而不得不提前触发一次垃圾收集动作。
* 复制算法

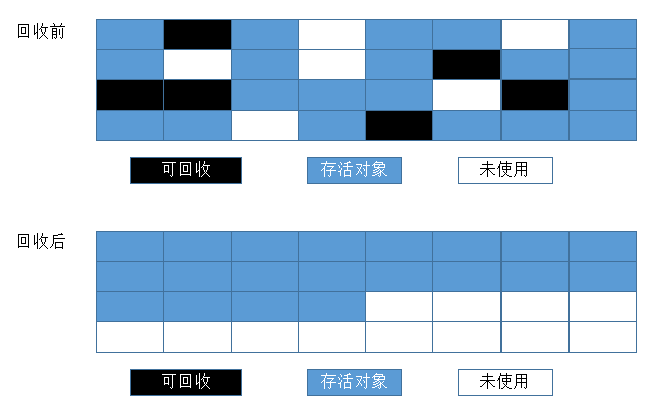
复制算法是为了解决效率问题而出现的，它将可用的内存分为两块，每次只用其中一块，当这一块内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已经使用过的内存空间一次性清理掉。这样每次只需要对整个半区进行内存回收，内存分配时也不需要考虑内存碎片等复杂情况，只需要移动指针，按照顺序分配即可。



不足之处：内存缩小为原来的一半，空间利用率不高，但是，大多数虚拟机用这种方法来收集新生代，不过比例不是1:1，新生代的内存被划分为一块较大的Eden空间和两块较小的Survivor空间，空间占比为8:1:1。每次使用Eden和其中一块Survivor来执行复制算法。

* 标记-整理算法(Mark-Compact)

复制算法在对象存活率较高的场景下要进行大量的复制操作，效率很低。老年代都是不易被回收的对象，对象存活率高，因此一般不能直接选用复制算法。根据老年代的特点，采用标记-整理算法，过程与标记-清除算法一样，不过不是直接对可回收对象进行清理，而是让所有存活对象都向一端移动，然后直接清理掉边界以外的内存。

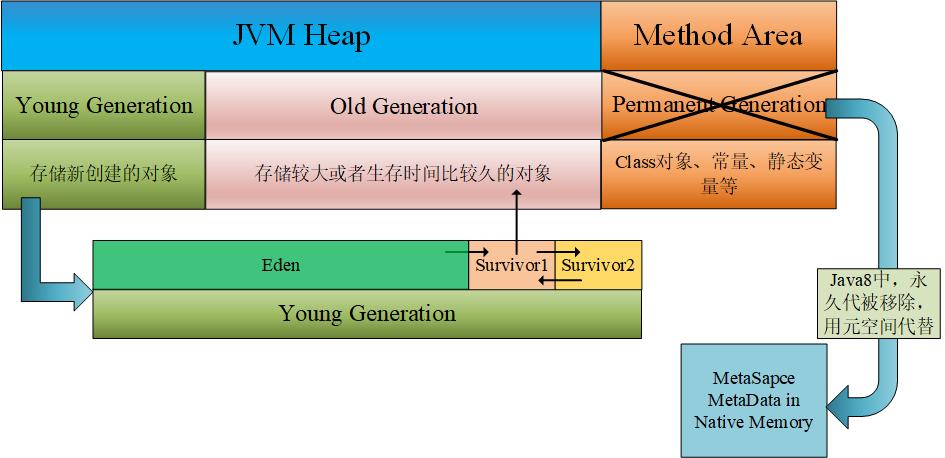


分代收集算法

现代商用虚拟机基本都采用分代收集算法来进行垃圾回收。，根据各个区域的特点采用不同的收集算法。大批对象死亡，少量对象存活的（新生代）采用复制算法；对象存活率高、没有额外空间进行分配担保的（老年代），采用标记-清理算法或者标记-整理算法。

1. Java Heap划分

根据对象存活的生命周期将内存划分为若干个不同的区域，一般将堆区分为新生代(Young Generation)和老年代(Old Generation)，堆区之外还有一个永久代(Permanent Generation)



* 新生代：主要用来存放新创建的对象，一般占据堆区的1/3空间。由于频繁创建对象，所以新生代会频繁触发MinorGC进行垃圾回收。新生代有划分为3个区域：Eden、SurvivorTo和SurvivorFrom。
* Eden区：Java新对象的出生地（如果新创建的对象过大，则直接分配到老年代），当Eden区域内存不够时就会触发MinorGC，对新生代进行一次垃圾回收。
* SurvivorTo：保留上一次MinorGC过程中的幸存者。
* SurvivorFrom：上一次MinorGC的幸存者，作为这一次MinorGC的被扫描区域。
* MinorGC过程：GC开始时，对象只会存在于Eden区和SurvivorFrom区，SurvivorTo区是空的。GC进行时，Eden区中存活的所有对象会被复制到SurvivorTo区，而在SurvivorFrom区中的对象会根据年龄大小决定去向，年龄达到阈值的被转移到老年代中，年龄不够的会被移动到SurvivorTo区。然后清空Eden区和SurvivorFrom区，接着SurvivorFrom和SurvivorTo交换角色。如果SurvivorTo区没有足够的空间存放GC以后的存活对象，就将存活对象直接放到老年代中。如果老年代也满了就会触发一次FullGC。
* 老年代：在新生代中经历了N次垃圾回收后仍然存活的对象，就会被放到老年代中，所以，老年代主要存放程序中生命周期比较长的对象。
* FullGC过程：老年代的对象都比较稳定，所以FullGC不会频繁执行，在进行一次MinorGC后，如果有新生代的对象进入老年代导致空间不够才会触发FullGC。如果无法找到足够大的连续空间分配给新创建的较大的对象也会触发FullGC。FullGC采用标记—清除算法：首先扫描一次所有老年代，标记出存活的对象，然后回收没有标记的对象。
* 永久代：指内存的永久保存区域，主要存放类的信息、常量池、方法代码等,Class在被加载的时候被放入永久区域. 它和存放实例对象的区域不同,GC不会在主程序运行期对永久区域进行清理。所以这也导致了永久代的区域会随着加载的Class的增多而胀满，最终抛出OOM异常。
* 永久代与元空间

方法区是JVM的规范，永久代是方法区的一种实现，只有HotSpot中有永久代的概念，其它JVM中不存在永久代。在java8中，永久代已经被移除，被“元数据区”(元空间)代替，元空间和永久代的本质类似，都是JVM规范中方法区的实现。不过区别在于：元空间不在虚拟机中，而是使用本地内存(Native Memory)。因此，元空间的大小仅受本地内存限制。类的元数据放入本地内存，字符串常量池和类的静态变量放入java堆中，这样可以加载类的元数据不是由虚拟机绝对，而是系统决定。

* 为什么要将永久代替换为元空间
  + 字符串存在永久代中，容易出现性能问题和内存溢出。
  + 类及方法的信息等比较难确定其大小，因此对于永久代的大小指定比较困难，太小容易出现永久代溢出，太大则容易导致老年代溢出。
  + 永久代会为 GC 带来不必要的复杂度，并且回收效率偏低。