**Java垃圾回收机制详解**

垃圾回收机制是 Java 非常重要的特性之一，也是面试题的常客。它让开发者无需关注空间的创建和释放，而是以守护进程的形式在后台自动回收垃圾。这样做不仅提高了开发效率，更改善了内存的使用状况。

今天本文来对垃圾回收机制进行讲解，主要涉及下面几个问题：

1、什么是堆内存？

2、什么是垃圾？

3、有哪些方法回收这些垃圾？

4、什么是分代回收机制？

**1、什么是 Java 堆内存**

堆是在 JVM 启动时创建的，主要用来维护运行时数据，如运行过程中创建的对象和数组都是基于这块内存空间。Java 堆是非常重要的元素，如果我们动态创建的对象没有得到及时回收，持续堆积，最后会导致堆空间被占满，内存溢出。

因此，Java 提供了一种垃圾回收机制，在后台创建一个守护进程。该进程会在内存紧张的时候自动跳出来，把堆空间的垃圾全部进行回收，从而保证程序的正常运行。

**2、那什么是垃圾呢？**

所谓“垃圾”，就是指所有不再存活的对象。常见的判断是否存活有两种方法：引用计数法和可达性分析。

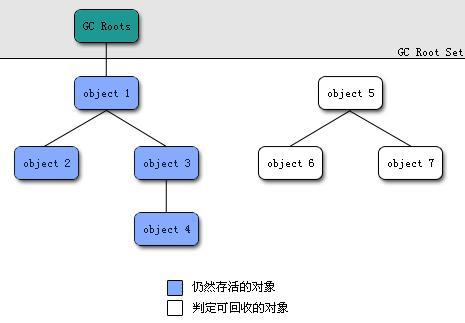
**（1）、引用计数法**

为每一个创建的对象分配一个引用计数器，用来存储该对象被引用的个数。当该个数为零，意味着没有人再使用这个对象，可以认为“对象死亡”。但是，这种方案存在严重的问题，就是无法检测“循环引用”：当两个对象互相引用，即使它俩都不被外界任何东西引用，它俩的计数都不为零，因此永远不会被回收。而实际上对于开发者而言，这两个对象已经完全没有用处了。因此，Java 里没有采用这样的方案来判定对象的“存活性”。

**（2）、可达性分析**

这种方案是目前主流语言里采用的对象存活性判断方案。基本思路是把所有引用的对象想象成一棵树，从树的根结点 GC Roots 出发，持续遍历找出所有连接的树枝对象，这些对象则被称为“可达”对象，或称“存活”对象。其余的对象则被视为“死亡”的“不可达”对象，或称“垃圾”。

参考下图，object5,object6和object7便是不可达对象，视为“死亡状态”，应该被垃圾回收器回收。



**GC Roots 究竟指谁呢？**

我们可以猜测，GC Roots 本身一定是可达的，这样从它们出发遍历到的对象才能保证一定可达。那么，Java 里有哪些对象是一定可达呢？主要有以下四种：

* 虚拟机栈（帧栈中的本地变量表）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。
* 本地方法栈中JNI引用的对象。

不少读者可能对这些 GC Roots 似懂非懂，这涉及到 JVM 本身的内存结构等等，未来的文章会再做深入讲解。这里只要知道有这么几种类型的 GC Roots，每次垃圾回收器会从这些根结点开始遍历寻找所有可达节点。

**3、有哪些方式来回收这些垃圾呢？**

上面已经知道，所有GC Roots不可达的对象都称为垃圾，参考下图，黑色的表示垃圾，灰色表示存活对象，绿色表示空白空间。



那么，我们如何来回收这些垃圾呢？

**标记－清理**

第一步，所谓“标记”就是利用可达性遍历堆内存，把“存活”对象和“垃圾”对象进行标记，得到的结果如上图；

第二步，既然“垃圾”已经标记好了，那我们再遍历一遍，把所有“垃圾”对象所占的空间直接清空即可。

结果如下：



这便是标记－清理方案，简单方便，但是容易产生内存碎片。

**标记－整理**

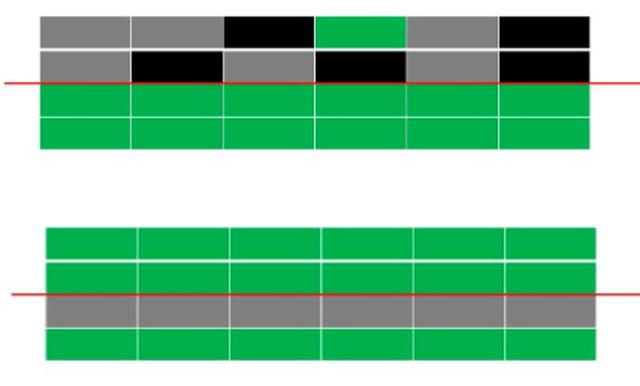
既然上面的方法会产生内存碎片，那好，我在清理的时候，把所有存活对象扎堆到同一个地方，让它们待在一起，这样就没有内存碎片了。结果如下：



这两种方案适合存活对象多，垃圾少的情况，它只需要清理掉少量的垃圾，然后挪动下存活对象就可以了。

**复制**

这种方法比较粗暴，直接把堆内存分成两部分，一段时间内只允许在其中一块内存上进行分配，当这块内存被分配完后，则执行垃圾回收，把所有存活对象全部复制到另一块内存上，当前内存则直接全部清空。参考下图：



起初时只使用上面部分的内存，直到内存使用完毕，才进行垃圾回收，把所有存活对象搬到下半部分，并把上半部分进行清空。

这种做法不容易产生碎片，也简单粗暴；但是，它意味着你在一段时间内只能使用一部分的内存，超过这部分内存的话就意味着堆内存里频繁的复制清空。

这种方案适合存活对象少，垃圾多的情况，这样在复制时就不需要复制多少对象过去，多数垃圾直接被清空处理。

**4、Java 的分代回收机制**

上面我们看到有至少三种方法来回收内存，那么 Java 里是如何选择利用这三种回收算法呢？是只用一种还是三种都用呢？

**Java 的堆结构**

在选择回收算法前，我们先来看一下 Java 堆的结构。

一块 Java 堆空间一般分成三部分，这三部分用来存储三类数据：

* 刚刚创建的对象。在代码运行时会持续不断地创造新的对象，这些新创建的对象会被统一放在一起。因为有很多局部变量等在新创建后很快会变成不可达的对象，快速死去，因此这块区域的特点是存活对象少，垃圾多。形象点描述这块区域为：新生代；
* 存活了一段时间的对象。这些对象早早就被创建了，而且一直活了下来。我们把这些存活时间较长的对象放在一起，它们的特点是存活对象多，垃圾少。形象点描述这块区域为：老年代；
* 永久存在的对象。比如一些静态文件，这些对象的特点是不需要垃圾回收，永远存活。形象点描述这块区域为：永久代。（不过在 Java 8 里已经把永久代删除了，把这块内存空间给了元空间，后续文章再讲解。）

也就是说，常规的 Java 堆至少包括了 新生代 和 老年代 两块内存区域，而且这两块区域有很明显的特征：

* 新生代：存活对象少、垃圾多
* 老年代：存活对象多、垃圾少

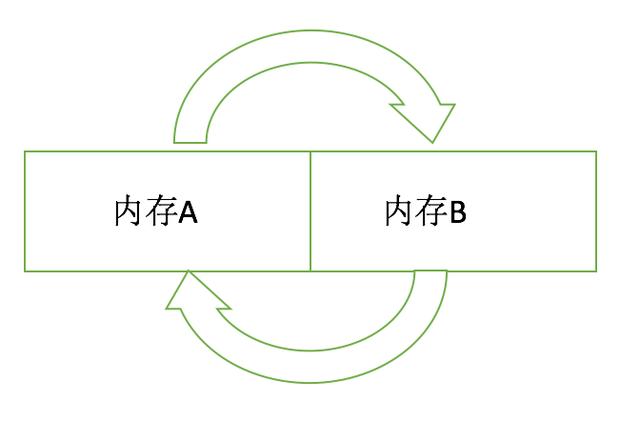
结合新生代／老年代的存活对象特点和之前提过的几种垃圾回收算法，可以得到如下的回收方案：

**新生代－复制回收机制**

对于新生代区域，由于每次 GC 都会有大量新对象死去，只有少量存活。因此采用复制回收算法，GC 时把少量的存活对象复制过去即可。 那么如何设计这个复制算法比较好呢？有以下几种方式：

**思路1. 把内存均分成 1:1 两等份**

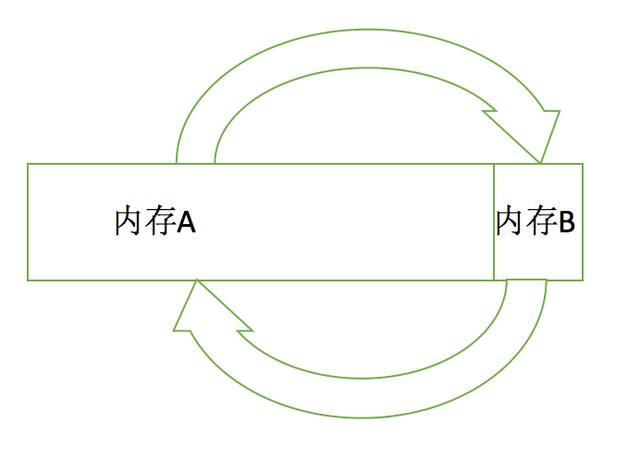
如下图拆分内存。



每次只使用一半的内存，当这一半满了后，就进行垃圾回收，把存活的对象直接复制到另一半内存，并清空当前一半的内存。

这种分法的缺陷是相当于只有一半的可用内存，对于新生代而言，新对象持续不断地被创建，如果只有一半可用内存，那显然要持续不断地进行垃圾回收工作，反而影响到了正常程序的运行，得不偿失。

**思路2. 把内存按 9:1 分**：既然上面的分法导致可用内存只剩一半，那么我做些调整，把 1:1变成9:1，

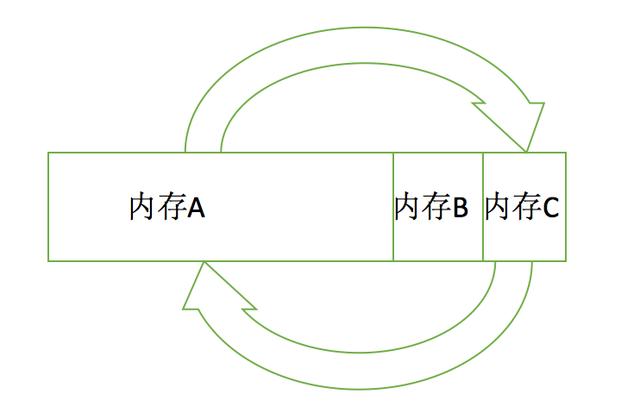


最开始在 9 的内存区使用，当 9 快要满时，执行复制回收，把 9 内仍然存活的对象复制到 1 区，并清空 9区。

这样看起来是比上面的方法好了，但是它存在比较严重的问题： 当我们把 9 区存活对象复制到 1 区时，由于内存空间比例相差比较大，所以很有可能 1 区放不满，此时就不得不把对象移到 老年区。而这就意味着，可能会有一部分 并不老 的 9 区对象由于 1 区放不下了而被放到了 老年区，可想而知，这破坏了 老年区 的规则。或者说，一定程度上的 老年区 并不一定全是 老年对象。

那应该如何才能把真正比较 老 的对象挪到 老年区 呢？

**思路3. 把内存按 8:1:1 分**



既然 9:1 有可能把年轻对象放到 老年区，那就换成 8:1:1，依次取名为 Eden、Survivor A、Survivor B区，其中Eden意为伊甸园，形容有很多新生对象在里面创建；Survivor区则为幸存者，即经历 GC 后仍然存活下来的对象。

工作原理如下：

1. 首先，Eden区最大，对外提供堆内存。当 Eden 区快要满了，则进行 Minor GC，把存活对象放入Survivor A区，清空 Eden 区；
2. Eden区被清空后，继续对外提供堆内存；
3. 当Eden区再次被填满，此时对Eden区和Survivor A区同时进行 Minor GC，把存活对象放入Survivor B区，同时清空Eden 区和Survivor A区；
4. Eden区继续对外提供堆内存，并重复上述过程，即在Eden区填满后，把Eden区和某个Survivor区的存活对象放到另一个Survivor区；
5. 当某个Survivor区被填满，且仍有对象未被复制完毕时，或者某些对象在反复Survive 15 次左右时，则把这部分剩余对象放到Old区；
6. 当 Old 区也被填满时，进行 Major GC，对 Old 区进行垃圾回收。

注意，在真实的 JVM 环境里，可以通过参数 SurvivorRatio 手动配置Eden区和单个Survivor区的比例，默认为8。

那么，所谓的 Old 区垃圾回收，或称Major GC，应该如何执行呢？

**老年代－标记整理回收机制**

根据上面我们知道，老年代一般存放的是存活时间较久的对象，所以每一次 GC 时，存活对象比较较大，也就是说每次只有少部分对象被回收。

因此，根据不同回收机制的特点，这里选择存活对象多，垃圾少的标记整理回收机制，仅仅通过少量地移动对象就能清理垃圾，而且不存在内存碎片化。

至此，我们已经了解了 Java 堆内存的分代原理，并了解了不同代根据各自特点采用了不同的回收机制，即新生代采用回收机制，老年代采用标记整理机制。

**小结**

垃圾回收是 Java 非常重要的特性，也是高级 Java 工程师的必经之路。