JVM垃圾回收

JVM不但可以管理内存，还可以对内存进行自动垃圾回收。所谓自动垃圾回收就是将JVM堆中的已经不再被使用的对象清理掉，释放宝贵的内存资源。那么要想进行垃圾回收，首先一个问题就是如何知道哪些对象是不再被使用的，可以清理的呢?

JVM通过一种可达性分析算法进行垃圾对象的识别，具体过程是：从线程栈帧中的局部变量，或者是方法区的静态变量出发，将这些变量引用的对象进行标记，然后看这些被标记的对象是否引用了其他对象，继续进行标记，所有被标记过的对象都是被使用的对象，而那些没有被标记的对象就是可回收的垃圾对象了。所以你可以看出来，可达性分析算法其实是一个引用标记算法。

进行完标记以后，JVM就会对垃圾对象占用的内存进行回收，回收主要有三种方法。

第一种方式是清理：将垃圾对象占据的内存清理掉，其实JVM并不会真的将这些垃圾内存进行清理，而是将这些垃圾对象占用的内存空间标记为空闲，记录在一个空闲列表里，当应用程序需要创建新对象的时候，就从空闲列表中找一段空闲内存分配给这个新对象。

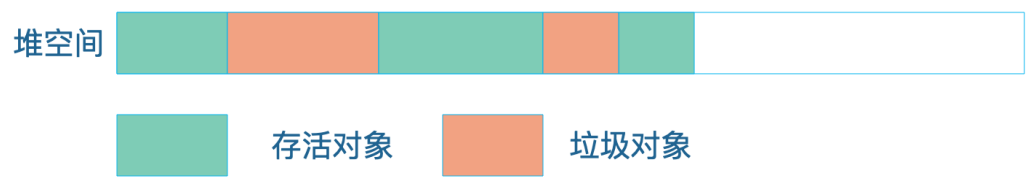
但这样做有一个很明显的缺陷，由于垃圾对象是散落在内存空间各处的，所以标记出来的空闲空间也是不连续的，当应用程序创建一个数组需要申请一段连续的大内存空间时，即使堆空间中有足够的空闲空间，也无法为应用程序分配内存。

第二种方式是压缩：从堆空间的头部开始，将存活的对象拷贝放在一段连续的内存空间中，那么其余的空间就是连续的空闲空间。

第三种方法是复制：将堆空间分成两部分，只在其中一部分创建对象，当这个部分空间用完的时候，将标记过的可用对象复制到另一个空间中。JVM将这两个空间分别命名为from区域和to区域。当对象从from区域复制到to区域后，两个区域交换名称引用，继续在from区域创建对象，直到from区域满。

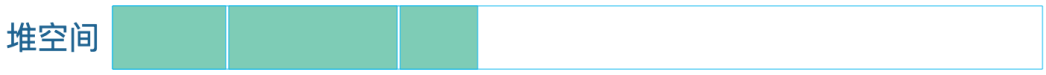
下面这系列图可以让你直观地了解JVM三种不同的垃圾回收机制。

回收前：



清理：

压缩：

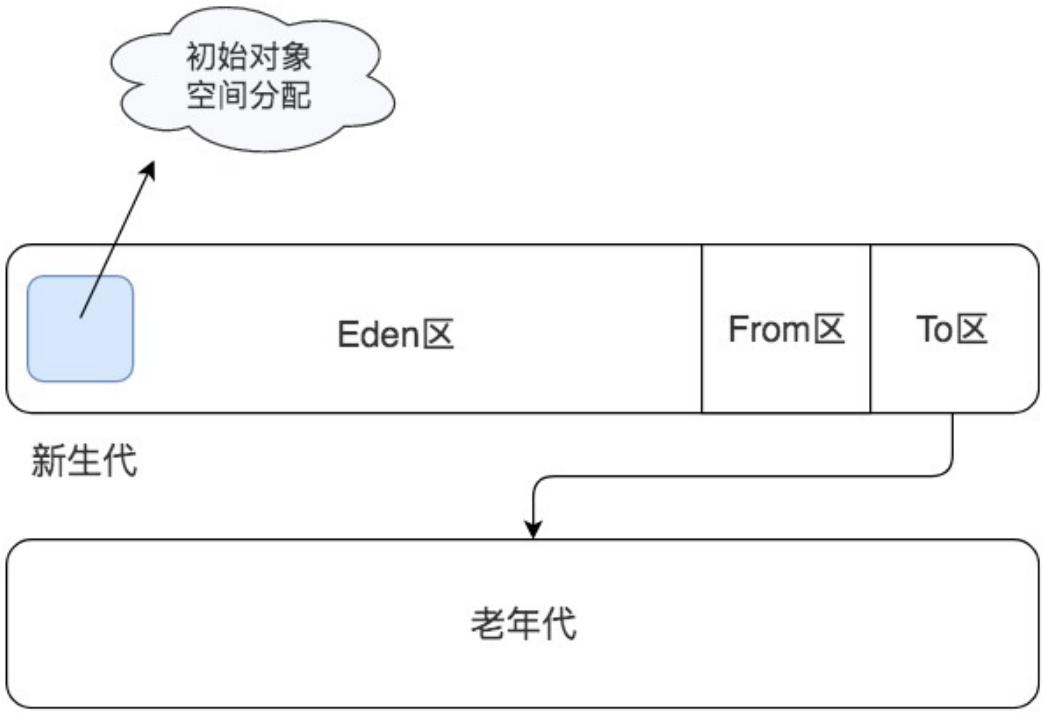


复制：

JVM在具体进行垃圾回收的时候，会进行分代回收。绝大多数的Java对象存活时间都非常短，很多时候就是在一个方法内创建对象，对象引用放在栈中，当方法调用结束，栈帧出栈的时候，这个对象就失去引用了，成为垃圾。针对这种情况，JVM将堆空间分成新生代（young）和老年代（old）两个区域，创建对象的时候，只在新生代创建，当新生代空间不足的时候，只对新生代进行垃圾回收，这样需要处理的内存空间就比较小，垃圾回收速度就比较快。

新生代又分为Eden区、From区和To区三个区域，每次垃圾回收都是扫描Eden区和From区，将存活对象复制到To区，然后交换From区和To区的名称引用，下次垃圾回收的时候继续将存活对象从From区复制到To区。当一个对象经过几次新生代垃圾回收，也就是几次从From区复制到To区以后，依然存活，那么这个对象就会被复制到老年代区域。

当老年代空间已满，也就是无法将新生代中多次复制后依然存活的对象复制进去的时候，就会对新生代和老年代的内存空间进行一次全量垃圾回收，即Full GC。所以根据应用程序的对象存活时间，合理设置老年代和新生代的空间比例对JVM垃圾回收的性能有很大影响，JVM设置老年代新生代比例的参数是–XX:NewRatio。



JVM中，具体执行垃圾回收的垃圾回收器有四种。

第一种是 Serial 串行垃圾回收器，这是JVM早期的垃圾回收器，只有一个线程执行垃圾回收。

第二种是 Parallel 并行垃圾回收器，它启动多线程执行垃圾回收。如果JVM运行在多核CPU上，那么显然并行垃圾回收要比串行垃圾回收效率高。

在串行和并行垃圾回收过程中，当垃圾回收线程工作的时候，必须要停止用户线程的工作，否则可能会导致对象的引用标记错乱，因此垃圾回收过程也被称为stop the world，在用户视角看来，所有的程序都不再执行，整个世界都停止了。

第三种 CMS 并发垃圾回收器，在垃圾回收的某些阶段，垃圾回收线程和用户线程可以并发运行，因此对用户线程的影响较小。Web应用这类对用户响应时间比较敏感的场景，适用CMS垃圾回收器。

最后一种是 G1 垃圾回收器，它将整个堆空间分成多个子区域，然后在这些子区域上各自独立进行垃圾回收，在回收过程中垃圾回收线程和用户线程也是并发运行。G1综合了以前几种垃圾回收器的优势，适用于各种场景，是未来主要的垃圾回收器。