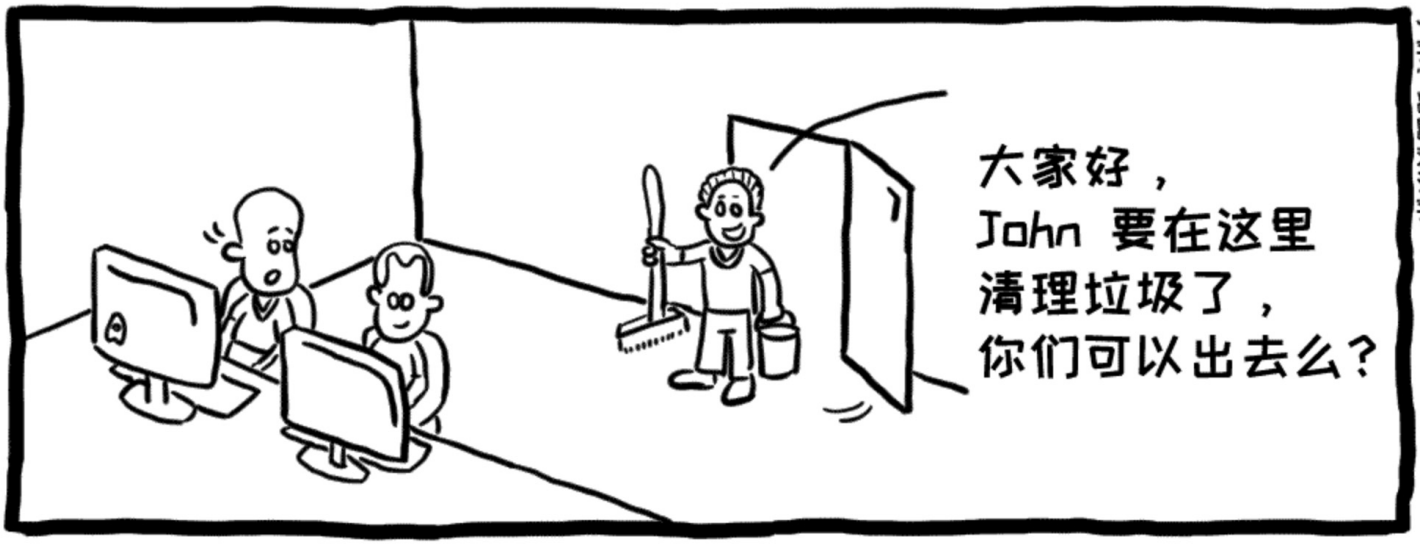
> 作者：率鸽
> 创作日期：2019-08-07
> 专栏地址：【稳定大于一切】



注：本文已同步至InfoQ https://www.infoq.cn/article/ZOYqRI4c-BFKmUBmzmKN

之前上学的时候有这个一个梗，说在食堂里吃饭，吃完把餐盘端走清理的，是C++程序员，吃完直接就走的，是Java程序员。确实，在Java的世界里，似乎我们不用对垃圾回收那么的专注，很多初学者不懂GC，也依然能写出一个能用甚至还不错的程序或系统。但其实这并不代表Java的GC就不重要。相反，它是那么的重要和复杂，以至于出了问题，那些初学者除了打开GC日志，看着一堆0101的天文，啥也做不了。今天我们就从头到尾完整地聊一聊Java的垃圾回收。

目录

什么是垃圾回收

垃圾回收（Garbage Collection，GC），顾名思义就是释放垃圾占用的空间，防止内存泄露。有效的使用可以使用的内存，对内存堆中已经死亡的或者长时间没有使用的对象进行清除和回收。

Java语言出来之前，大家都在拼命的写C或者C++的程序，而此时存在一个很大的矛盾，C++等语言创建对象要不断的去开辟空间，不用的时候又需要不断的去释放控件，既要写构造函数，又要写析构函数，很多时候都在重复的allocated，然后不停的析构。于是，有人就提出，能不能写一段程序实现这块功能，每次创建，

释放控件的时候复用这段代码，而无需重复的书写呢?

1960年基于MIT的Lisp首先提出了垃圾回收的概念，而这时Java还没有出世呢！所以实际上GC并不是Java的专利，GC的历史远远大于Java的历史！

怎么定义垃圾

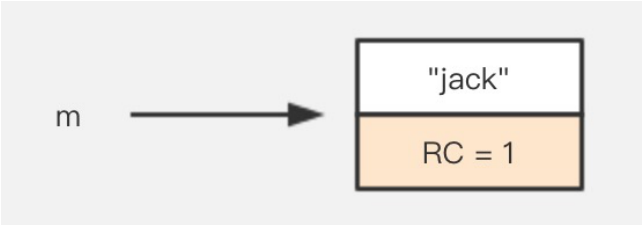
既然我们要做垃圾回收，首先我们得搞清楚垃圾的定义是什么，哪些内存是需要回收的。

引用计数算法

引用计数算法（Reachability Counting）是通过在对象头中分配一个空间来保存该对象被引用的次数（Reference Count）。如果该对象被其它对象引用，则它的引用计数加1，如果删除对该对象的引用，那么它的引用计数就减1，当该对象的引用计数为0时，那么该对象就会被回收。

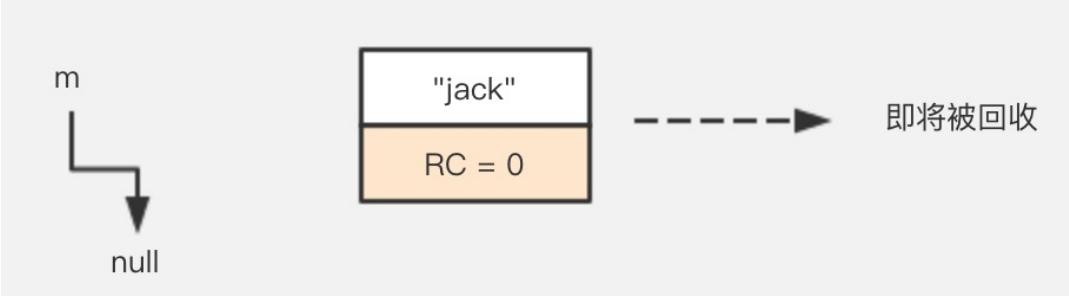
```
String m = new String("jack");
```

先创建一个字符串，这时候"jack"有一个引用，就是m。



然后将m置为null，这时候"jack"的引用次数就等于0了，在引用计数算法中，意味着这块内容就需要被回收了。

```
m = null;
```



引用计数算法是将垃圾回收分摊到整个应用程序的运行当中了，而不是在进行垃圾收集时，要挂起整个应用的运行，直到对堆中所有对象的处理都结束。因此，采用引用计数的垃圾收集不属于严格意义上的"Stop-The-World"的垃圾收集机制。

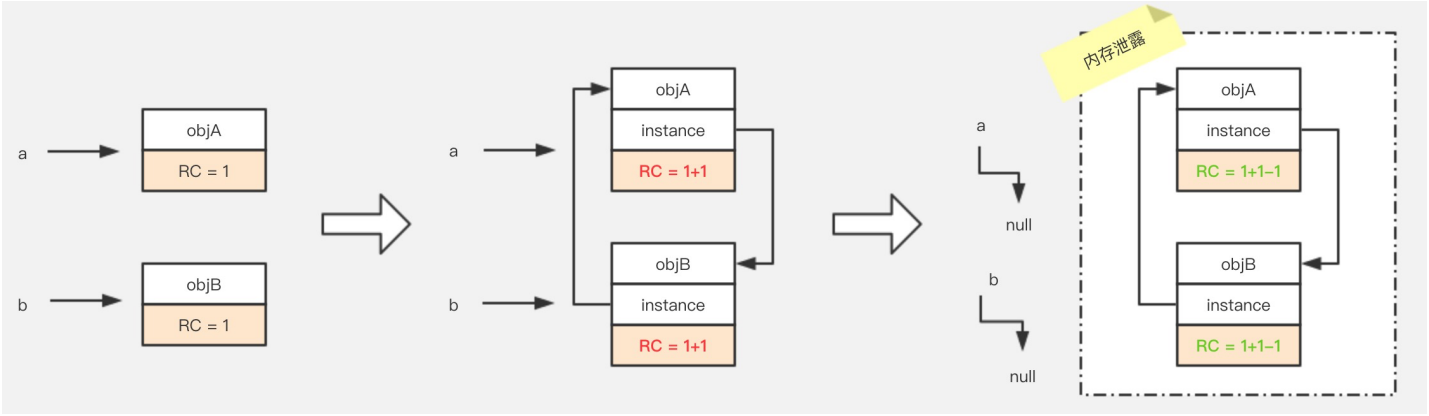
看似很美好，但我们知道JVM的垃圾回收就是"Stop-The-World"的，那是什么原因导致我们最终放弃了引用计数算法呢？看下面的例子。

```
public class ReferenceCountingGC {

    public Object instance;

    public ReferenceCountingGC(String name){}
}

public static void testGC(){

    ReferenceCountingGC a = new ReferenceCountingGC("objA");
    ReferenceCountingGC b = new ReferenceCountingGC("objB");

    a.instance = b;
    b.instance = a;

    a = null;
    b = null;
}
```

1.定义2个对象

2.相互引用

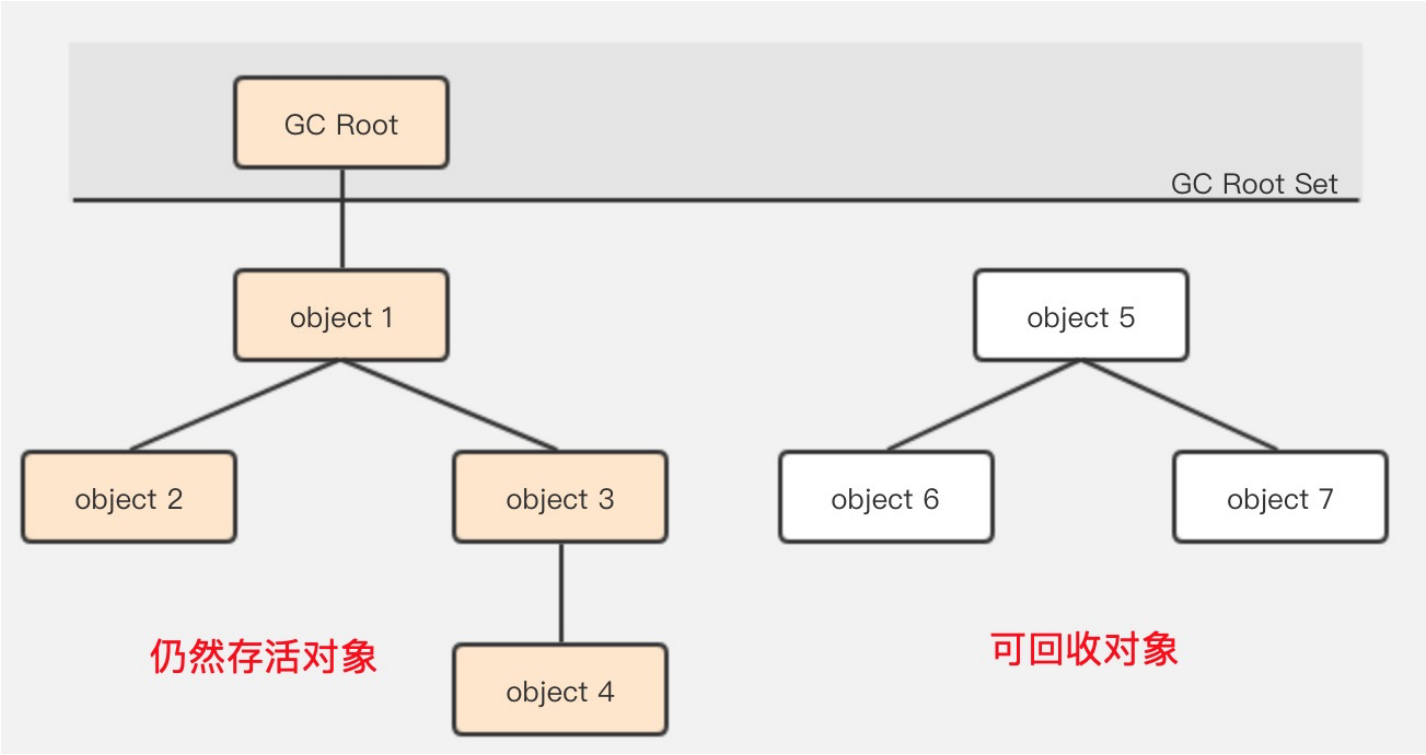
3.置空各自的声明引用



我们可以看到，最后这2个对象已经不可能再被访问了，但由于他们相互引用着对方，导致它们的引用计数永远都不会为0，通过引用计数算法，也就永远无法通知GC收集器回收它们。

可达性分析算法

可达性分析算法（Reachability Analysis）的基本思路是，通过一些被称为引用链（GC Roots）的对象作为起点，从这些节点开始向下搜索，搜索走过的路径被称为（Reference Chain），当一个对象到GC Roots没有任何引用链相连时（即从GC Roots节点到该节点不可达），则证明该对象是不可用的。

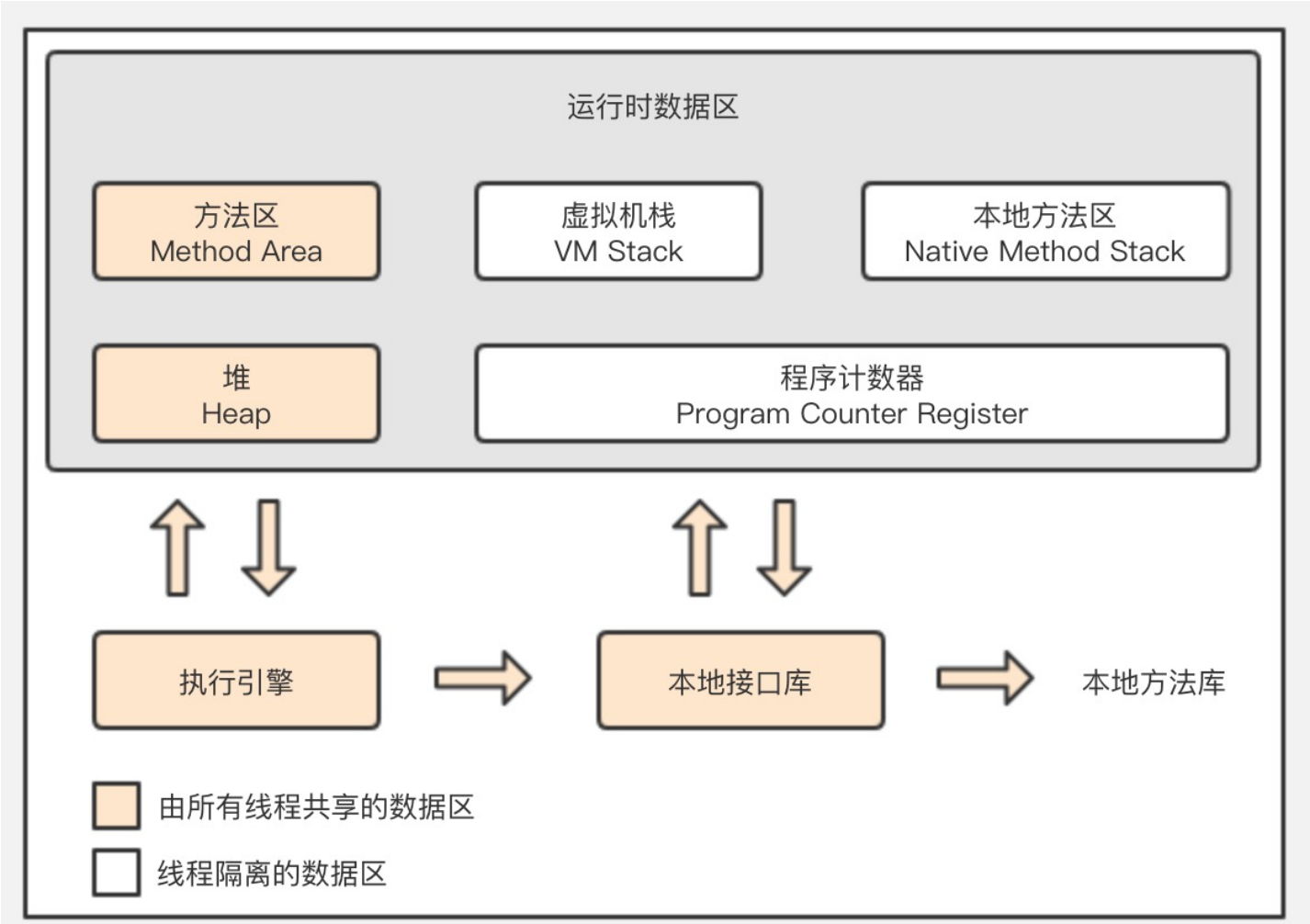


通过可达性算法，成功解决了引用计数所无法解决的问题-“循环依赖”，只要你无法与GC Root建立直接或间接的连接，系统就会判定你为可回收对象。那这样就引申出了另一个问题，哪些属于GC Root?

Java内存区域

在Java语言中，可作为GC Root的对象包括以下4种

- 虚拟机栈（栈帧中的本地变量表）中引用的对象
- 方法区中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（即一般说的Native方法）引用的对象



虚拟机栈（栈帧中的本地变量表）中引用的对象 此时的s，即为GC Root，当s置空时，localParameter对象也断掉了与GC Root的引用链，将被回收。

```
public class StackLocalParameter {
    public StackLocalParameter(String name){}
}

public static void testGC(){
    StackLocalParameter s = new StackLocalParameter("localParameter");
    s = null;
}
```

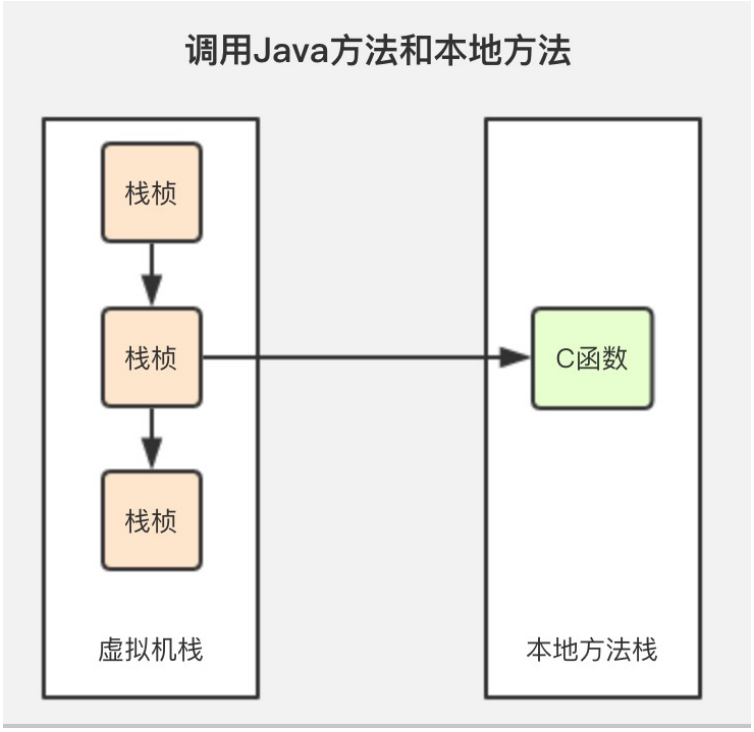
方法区中类静态属性引用的对象 s为GC Root，s置为null，经过GC后，s所指向的properties对象由于无法与GC Root建立关系被回收。 而m作为类的静态属性，也属于GC Root，parameter对象依然与GC root建立着连接，所以此时parameter对象并不会被回收。

```
public class MethodAreaStaicProperties {
    public static MethodAreaStaicProperties m;
    public MethodAreaStaicProperties(String name){}
}

public static void testGC(){
    MethodAreaStaicProperties s = new MethodAreaStaicProperties("properties");
    s.m = new MethodAreaStaicProperties("parameter");
    s = null;
}
```

方法区中常量引用的对象 m即为方法区中的常量引用，也为GC Root，s置为null后，final对象也不会因没有与GC Root建立联系而被回收。

```
public class MethodAreaStaicProperties {
    public static final MethodAreaStaicProperties m = MethodAreaStaicProperties("f
inal");
    public MethodAreaStaicProperties(String name){}
}

public static void testGC(){
    MethodAreaStaicProperties s = new MethodAreaStaicProperties("staticProperties"
);
    s = null;
}
```

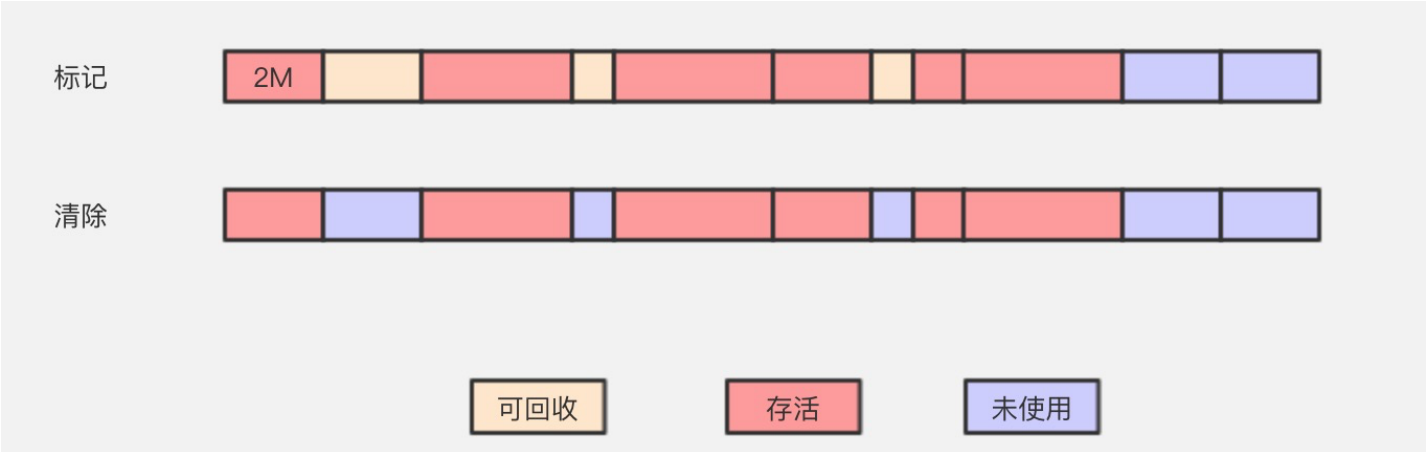
本地方法栈中引用的对象 任何native接口都会使用某种本地方法栈，实现的本地方法接口是使用C连接模型的话，那么它的本地方法栈就是C栈。当线程调用Java方法时，虚拟机会创建一个新的栈帧并压入Java栈。然而当它调用的是本地方法时，虚拟机会保持Java栈不变，不再在线程的Java栈中压入新的帧，虚拟机只是简单地动态连接并直接调用指定的本地方法。



怎么回收垃圾

在确定了哪些垃圾可以被回收后，垃圾收集器要做的事情就是开始进行垃圾回收，但是这里面涉及到一个问题是：如何高效地进行垃圾回收。由于Java虚拟机规范并没有对如何实现垃圾收集器做出明确的规定，因此各个厂商的虚拟机可以采用不同的方式来实现垃圾收集器，这里我们讨论几种常见的垃圾收集算法的核心思想。

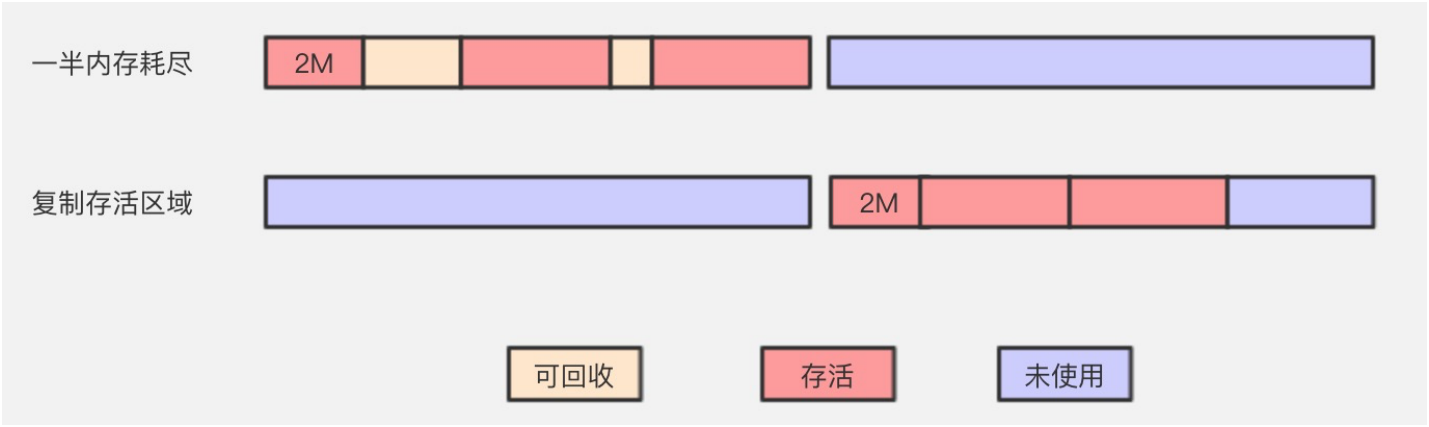
标记-清除算法



标记清除算法（Mark-Sweep）是最基础的一种垃圾回收算法，它分为2部分，先把内存区域中的这些对象进行标记，哪些属于可回收标记出来，然后把这些垃圾拎出来清理掉。就像上图一样，清理掉的垃圾就变成未使用的内存区域，等待被再次使用。 这逻辑再清晰不过了，并且也很好操作，但它存在一个很大的问题，那就是内存碎片。

上图中等方块的假设是2M，小一些的是1M，大一些的是4M。等我们回收完，内存就会切成了很多段。我们知道开辟内存空间时，需要的是连续的内存区域，这时候我们需要一个2M的内存区域，其中有2个1M是没法用的。这样就导致，其实我们本身还有这么多的内存的，但却用不了。

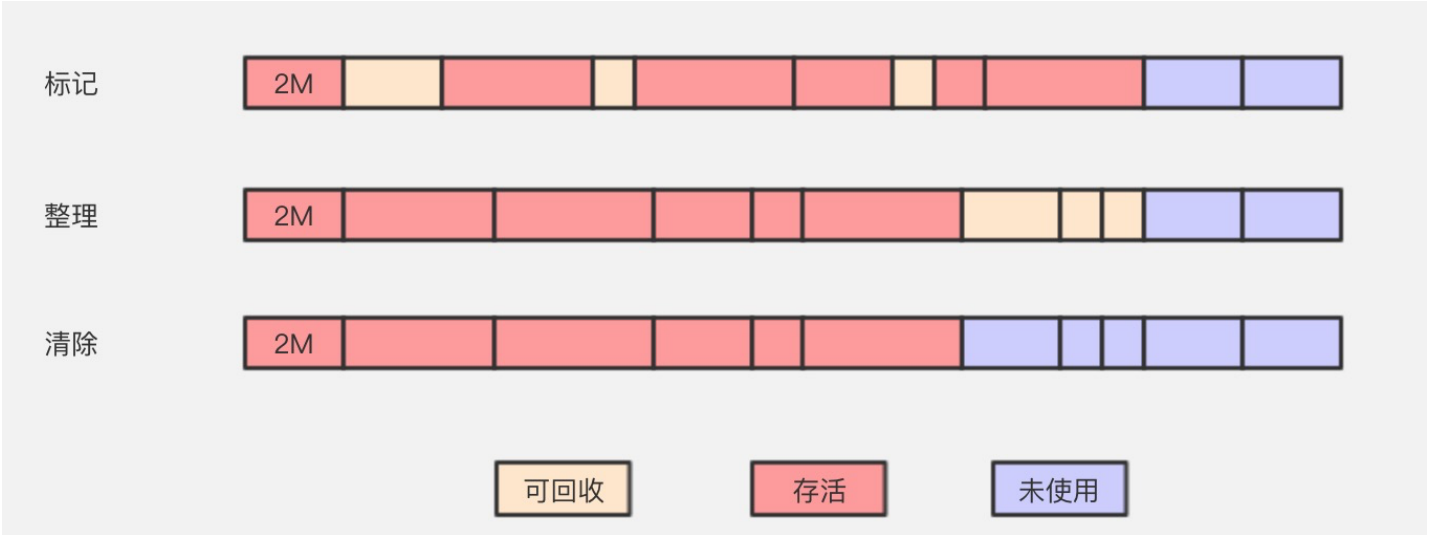
复制算法



复制算法（Copying）是在标记清除算法上演化而来，解决标记清除算法的内存碎片问题。它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。保证了内存的连续可用，内存分配时也就不用考虑内存碎片等复杂情况，逻辑清晰，运行高效。

上面的图很清楚，也很明显的暴露了另一个问题，合着我这140平的大三房，只能当70平米的小两房来使?代价实在太高。

标记整理算法



标记整理算法（Mark-Compact）标记过程仍然与标记-清除算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，再清理掉端边界以外的内存区域。

标记整理算法一方面在标记-清除算法上做了升级，解决了内存碎片的问题，也规避了复制算法只能利用一半

内存区域的弊端。看起来很美好，但从上图可以看到，它对内存变动更频繁，需要整理所有存活对象的引用地址，在效率上比复制算法要差很多。

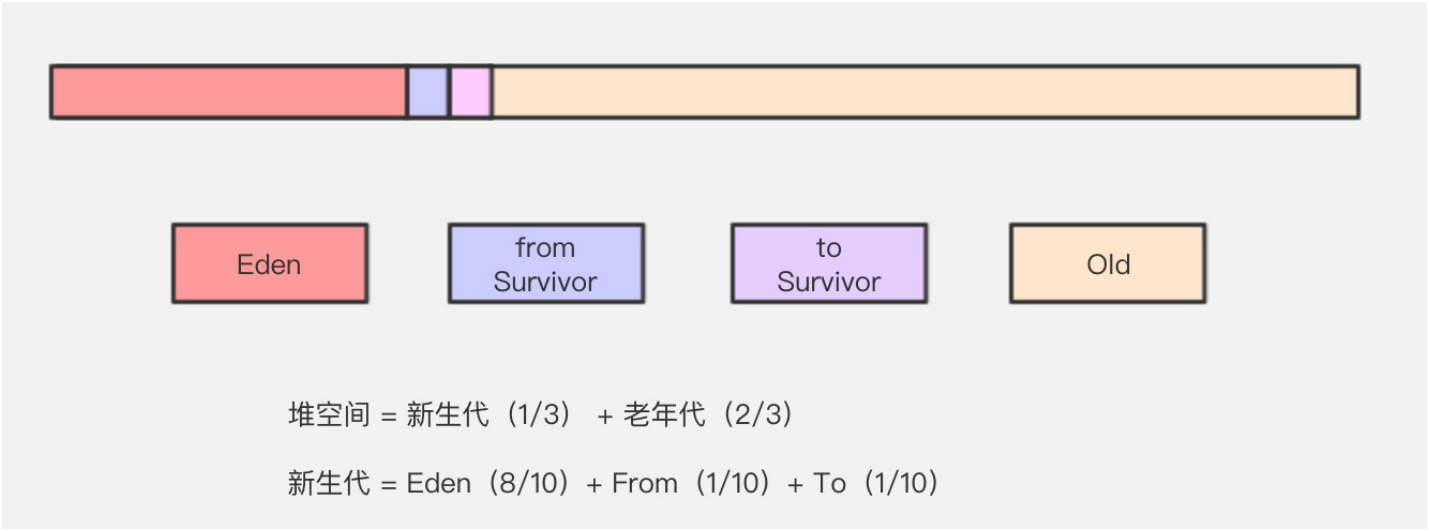
分代收集算法

分代收集算法（Generational Collection）严格来说并不是一种思想或理论，而是融合上述3种基础的算法思想，而产生的针对不同情况所采用不同算法的一套组合拳。

对象存活周期的不同将内存划分为几块。一般是把Java堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用标记-清理或者标记-整理算法来进行回收。

so，另一个问题来了，那内存区域到底被分为哪几块，每一块又有什么特别适合什么算法呢?

内存模型与回收策略



Java堆（Java Heap）是JVM所管理的内存中最大的一块，堆又是垃圾收集器管理的主要区域，这里我们主要分析一下Java堆的结构。

Java堆主要分为2个区域-年轻代与老年代，其中年轻代又分Eden区和Survivor区，其中Survivor区又分From和To2个区。可能这时候大家会有疑问，为什么需要survivor区，为什么survivor还要分2个区。不着急，我们从头到尾，看看对象到底是怎么来的，而它又是怎么没的。

Eden区

IBM公司的专业研究表明，有将近98%的对象是朝生夕死，所以针对这一现状，大多数情况下，对象会在新生代Eden区中进行分配，当Eden区没有足够空间进行分配时，虚拟机会发起一次Minor GC，Minor GC相比Major GC更频繁，回收速度也更快。 通过Minor GC之后，Eden会被清空，Eden区中绝大部分对象会被回收，而那些无需回收的存活对象，将会进到Survivor的From区（若From区不够，则直接进入Old区）。

Survivor区

Survivor区相当于是Eden区和Old区的一个缓冲，类似于我们交通灯中的黄灯。Survivor又分为2个区，一个是From区，一个是To区。每次执行Minor GC，会将Eden区和From存活的对象放到Survivor的To区（如果To区不够，则直接进入Old区）。

为啥需要

不就是新生代到老年代么，直接Eden到Old不好了吗，为啥要这么复杂。想想如果没有Survivor区，Eden区每进行一次Minor GC，存活的对象就会被送到老年代，老年代很快就会被填满。而有很多对象虽然一次Minor GC没有消灭，但其实也并不会蹦跶多久，或许第二次，第三次就需要被清除。这时候移入老年区，很明显不是一个明智的决定。

所以，Survivor的存在意义就是减少被送到老年代的对象，进而减少Major GC的发生。Survivor的预筛选保证，只有经历16次Minor GC还能在新生代中存活的对象，才会被送到老年代。

为啥需要俩?

设置两个Survivor区最大的好处就是解决内存碎片化。

我们先假设一下，Survivor如果只有一个区域会怎样。Minor GC执行后，Eden区被清空了，存活的对象放到了Survivor区，而之前Survivor区中的对象，可能也有一些是需要被清除的。问题来了，这时候我们怎么清除它们？在这种场景下，我们只能标记清除，而我们知道标记清除最大的问题就是内存碎片，在新生代这种经常会消亡的区域，采用标记清除必然会让内存产生严重的碎片化。因为Survivor有2个区域，所以每次Minor GC，会将之前Eden区和From区中的存活对象复制到To区域。第二次Minor GC时，From与To职责兑换，这时候会将Eden区和To区中的存活对象再复制到From区域，以此反复。

这种机制最大的好处就是，整个过程中，永远有一个survivor space是空的，另一个非空的survivor space是无碎片的。那么，Survivor为什么不分更多块呢？比方说分成三个、四个、五个?显然，如果Survivor区再细分下去，每一块的空间就会比较小，容易导致Survivor区满，两块Survivor区可能是经过权衡之后的最佳方案。

Old区

老年代占据着2/3的堆内存空间，只有在Major GC的时候才会进行清理，每次GC都会触发“Stop-The-World”。内存越大，STW的时间也越长，所以内存也不仅仅是越大就越好。由于复制算法在对象存活率较高的老年代会进行很多次的复制操作，效率很低，所以老年代这里采用的是标记-整理算法。

除了上述所说，在内存担保机制下，无法安置的对象会直接进到老年代，以下几种情况也会进入老年代。

大对象

大对象指需要大量连续内存空间的对象，这部分对象不管是不是“朝生夕死”，都会直接进到老年代。这样做主要是为了避免在Eden区及2个Survivor区之间发生大量的内存复制。当你的系统有非常多“朝生夕死”的大对

象时，得注意了。

长期存活对象

虚拟机给每个对象定义了一个对象年龄（Age）计数器。正常情况下对象会不断的在Survivor的From区与To区之间移动，对象在Survivor区中没经历一次Minor GC，年龄就增加1岁。当年龄增加到15岁时，这时候就会被转移到老年代。当然，这里的15，JVM也支持进行特殊设置。

动态对象年龄

虚拟机并不重视要求对象年龄必须到15岁，才会放入老年区，如果Survivor空间中相同年龄所有对象大小的综合大于Survivor空间的一半，年龄大于等于该年龄的对象就可以直接进去老年区，无需等你“成年”。

这其实有点类似于负载均衡，轮询是负载均衡的一种，保证每台机器都分得同样的请求。看似很均衡，但每台机的硬件不通，健康状况不同，我们还可以基于每台机接受的请求数，或每台机的响应时间等，来调整我们的负载均衡算法。

参阅书籍：《深入理解Java虚拟机》 https://detail.tmall.com/item.htm?id=529766776721

加入我们

【稳定大于一切】打造国内稳定性领域知识库，**让无法解决的问题少一点点，让世界的确定性多一点点。**

- GitHub 地址
- 钉钉群号：23179349