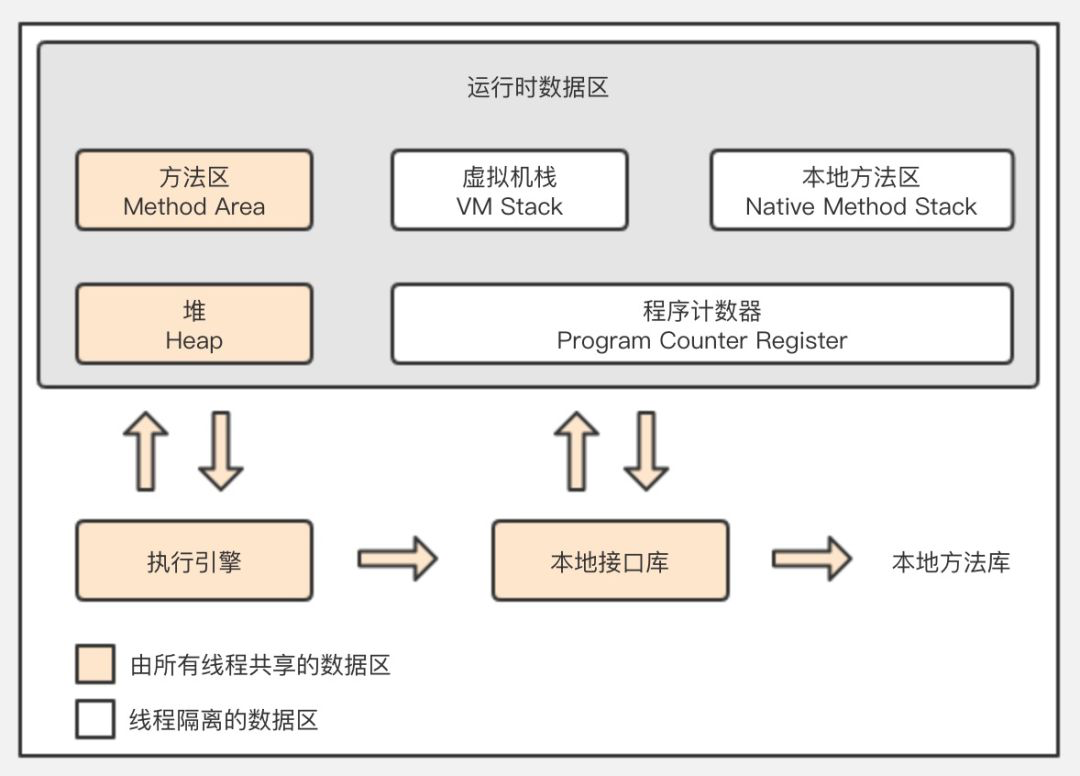
1. **jvm运行时数据区**



**堆：**几乎所有对象，数组都存放在该区域（不一定都会分配在堆上，如果启用了栈上分配，标量替换等对象逃逸分析优化技术，则直接在栈上分配，对象随着方法的销毁而销毁），该区域是所有线程共享的。

**方法区（非堆）：**存放类的信息（类的版本，字段，方法，接口等描述信息），编译期的以及运行时（例如String.intern方法）的常量

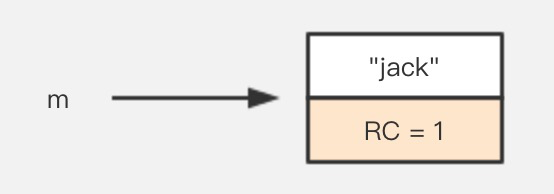
1. **什么是垃圾回收？**

释放内存空间，防止内存溢出。JVM中的垃圾回收主要是对堆和非堆（方法区）中的对象进行清除和回收。

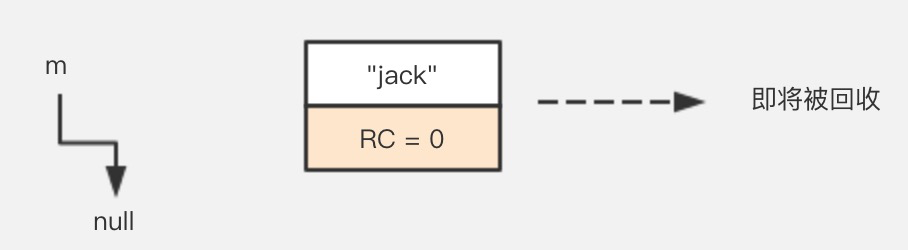
1. **垃圾的定义，哪些是垃圾（什么样的内存需要被回收）**

**引用计数算法：**如果一个对象被另外一个对象/变量引用，则该对象的引用计数加1，如果该对象的引用被删除了，则引用计数减1，当引用计数=0表示该对象可以被回收了。

String m = **new** String(**"Jack"**);



m = **null**;



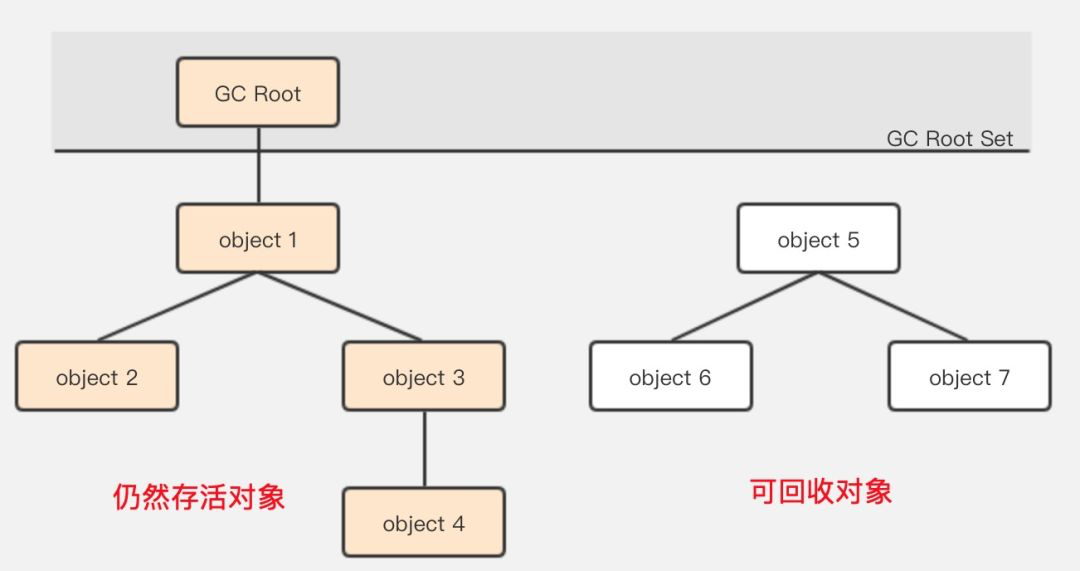
**优点**：实现简单，判定高效，应用：微软的com技术，python。

那为什么Java没有使用这种算法呢？主要是因为无法解决引用循环依赖问题。看以下例子：

**public class** ReferenceCountingGC {  
 **public static void** main(String[] args) {  
 *// 定义两个对象* Student student = **new** Student(); //RC=1  
 Teacher teacher = **new** Teacher(); //RC=1  
  
 *// 对象相互引用* student.**instance** = teacher; //RC=1+1=2  
 teacher.**instance** = student; //RC=1+1=2  
  
  
 *// 两个对象置为NULL* student = **null**; //RC=2-1=1  
 teacher = **null**; //RC=2-1=1  
 }  
  
 **static class** Student {  
 **public** Teacher **instance**;  
 }  
  
 **static class** Teacher {  
 **public** Student **instance**;  
 }  
  
}

**缺点**：无法解决循环依赖引用问题

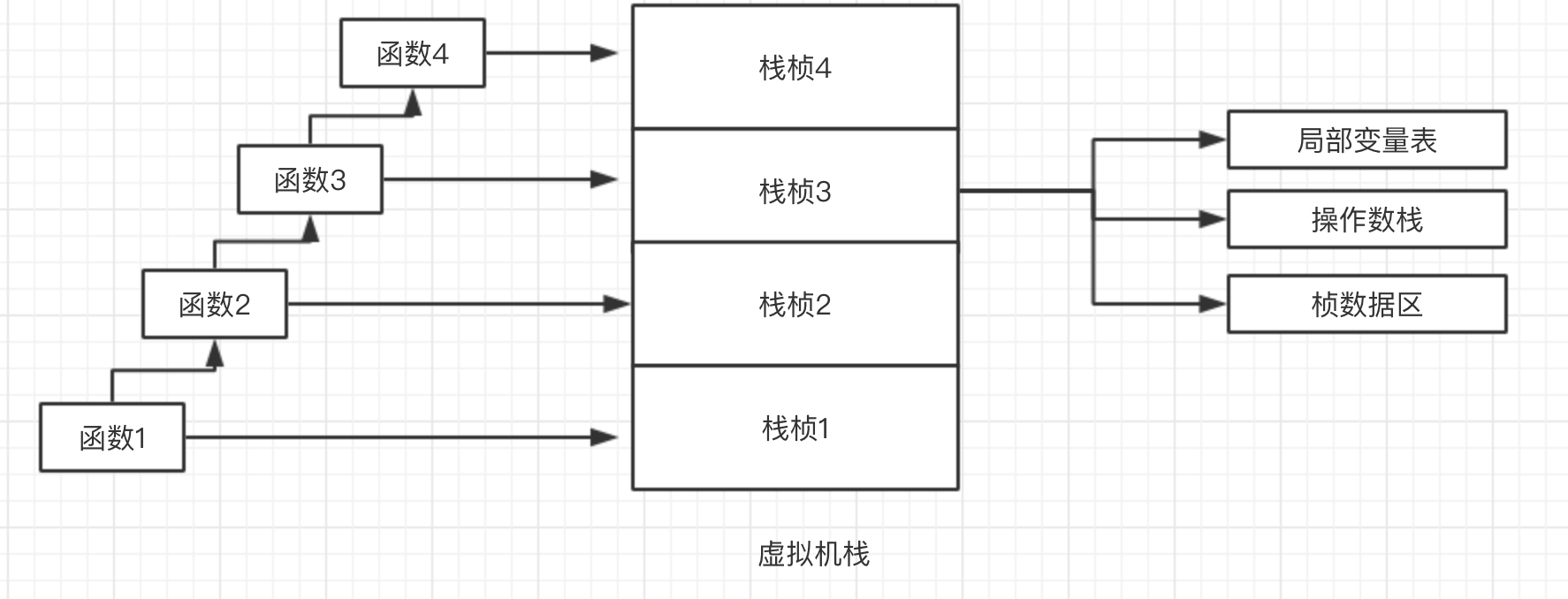
**根可达性分析算法：**基本原理是找到一个可以作为GC roots（根结点）的对象作为起点，从这些结点开始向下搜索，搜索所走过的路径称为引用链，当一个对象到GC roots不可达的时候，则证明该对象不可用，可以被回收。



那什么对象可以作为GC roots？有以下这些：

1. 虚拟机栈（栈帧中的本地变量表）中引用的对象
2. 方法区中类静态属性引用的对象
3. 方法区中常量引用的对象
4. 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，jclasslib, 结合例子）中引用的对象：



LocalVariableTableGc

方法区中类静态属性引用的对象：

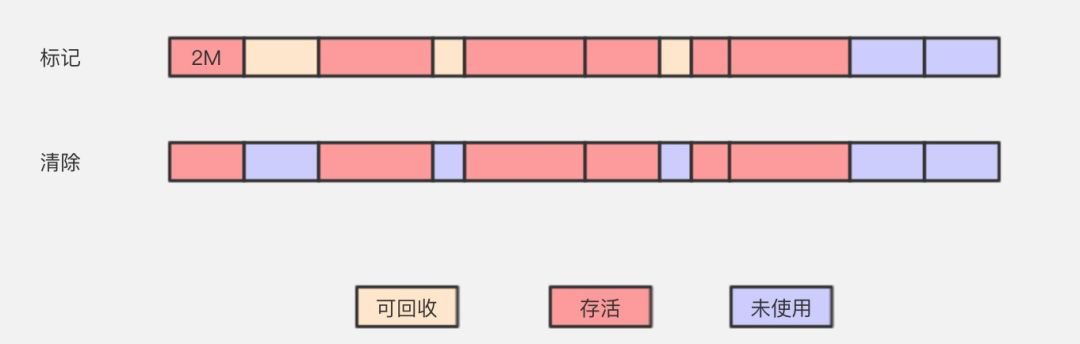
MethodAreaStaticProperties

方法区中常量引用的对象:

MethodAreaStaticFinalProperties

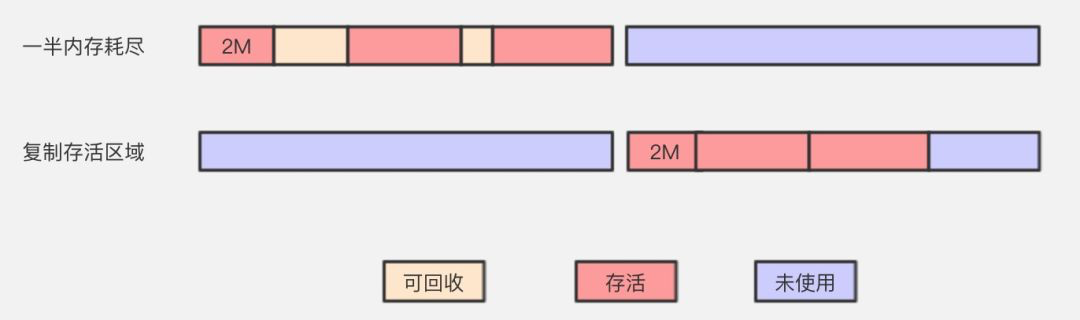
1. **怎样回收垃圾**

标记-清除算法：



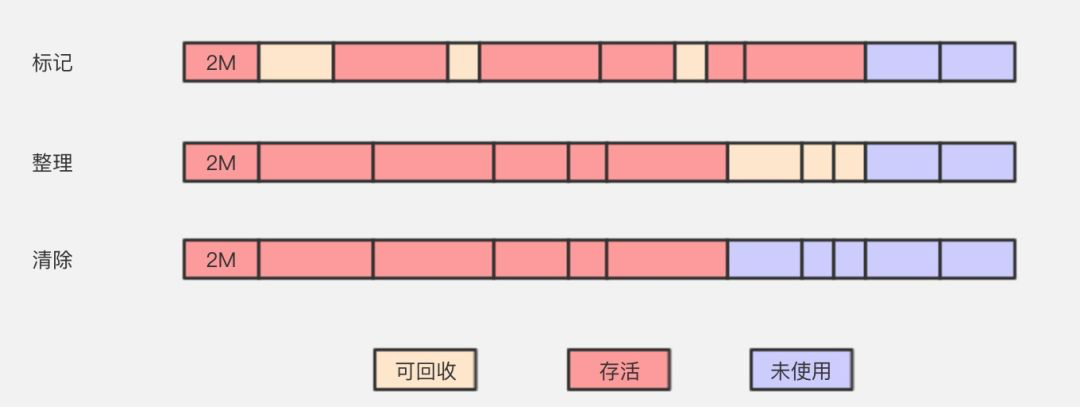
缺点：导致内存碎片问题

标记-复制算法：



缺点：空间使用率低（空间换时间，有连续空间，分配大对象成功的几率大，不像标记-清除算法：几率相对低）

标记-整理算法：

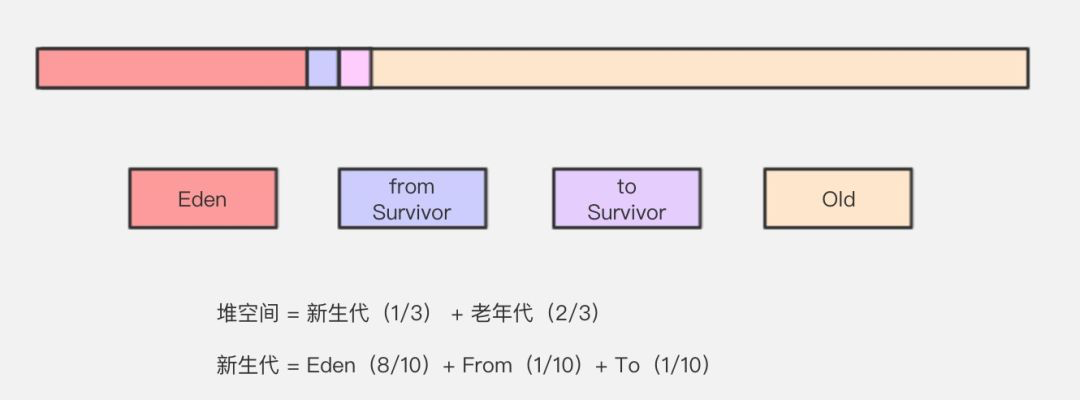


缺点：解决了标记-清除算法的内存碎片问题以及标记-复制算法空间利用率低的问题，但是如果需要不断地变换对象的内存地址，在效率上比复制算法要差

分代收集算法：

融合前面三种算法，针对不同的堆内存情况采取不同的算法。

1. **堆内存模型和回收策略**（演示gc参数配置的回收策略例子，如何查看默认值等）



Eden区：

新生成的对象会分配到此空间（不一定，是否开启TLAB），如果Eden区空间不够就会发生Minor GC（发生在新生代），Minor GC比较频繁，速度快。经过Minor GC整个Eden区会被清空，大部分对象都是“朝生夕死”，剩下的存活对象会进入到From区（复制算法），如果From区空间不够则进入Old区

Eden

Survivor区：

Survivor区相当于Eden区和Old区之间的一个缓冲区。它分为两个：From，To，每次Minor

GC的时候会将Eden区存活的对象以及From中的对象复制到To survivor区中（下一次Minor GC的时候To变成From角色，From变成To角色，它们角色会互相变换）

为什么需要Survivor区呢？对象之间从Eden区进入到Old区不就好了吗？

如果直接进入到Old区很容易导致这个区域满了，这样就埋下了Major GC（发生在老年代）的隐患，Major GC是很耗时间的（发生GC的时候会stop-the-world，应用程序停止工作），意味着整个程序的停顿时间就变长了。虽然有很多对象在第一次Minor GC不会被回收，但是可能第二次，三次就需要被清除掉了（有年龄限制，每Minor GC一次如果还存活年龄+1，默认是最大年龄是15，如果达到15还存活则进入老年代，可以通过参数配置）。很明显直接进入Old区不是很明智的选择。

为什么会有两个Survivor区？

采用标记-复制算法，所以需要有一个区域用来作复制数据用的。如果只有一个Survivor区的话就做不到来回复制了。这样也有一个好处，只有一小块Survivor区存在空间的浪费。

Old区：

只有在Major GC（也叫Full GC）的时候才会清理该区域，大概率都会stop-the-world。并不是内存越大越好，越大意味着清理就越耗时，程序停顿时间越长。该区域存活率比较高，不适用标记-复制算法，这个区域一般采用标记-整理算法

除了上面提及的情况外，在内存担保机制下，无法分配的对象会直接进入老年代。以下这几种情况也会让对象直接进入Old区：

Old

大对象：

对于比较大的对象，为了避免在新生代中复制来复制去，消耗内存，可以直接让它进入到老年代。可以通过参数-XX:PretenureSizeThreshold（注意Parallel Scavenge收集器该参数无效）设置。

长期存活的对象（年龄达到阈值）：

如果对象在From和To survivor区移动来移动去，只要在这两个区域中每经历一次Minor GC都会给它的年龄计数+1，当达到阈值就进入老年代，可以通过参数（-XX:MaxTenuringThreshold）配置该年龄。

动态对象年龄：

jvm并不是说一定会等到对象的年龄达到阈值才进入老年代，如果survivor中同年龄的对象总大小大于其中一个Survivor的空间的一半，年龄大于或者等于该年龄的对象就直接进入老年代。

1. **什么时候触发垃圾回收**（分配过程，第五章，TLAB）

Minor GC触发机制：

当年轻代满时就会触发Minor GC，这里的年轻代满指的是Eden代满，Survivor满不会引发GC。

Full GC触发机制：

（1）调用System.gc时，系统建议执行Full GC，但是不必然执行

（2）老年代空间不足

（3）方法区空间不足

（4）通过Minor GC后进入老年代的平均大小大于老年代的可用内存

（5）由Eden区、survivor space1（From Space）区向survivor space2（To Space）区复制时，对象大小大于To Space可用内存，则把该对象转存到老年代，且老年代的可用内存小于该对象大小

参考：深入理解Java虚拟机-周志明