垃圾回收

免费的，其实是最贵的

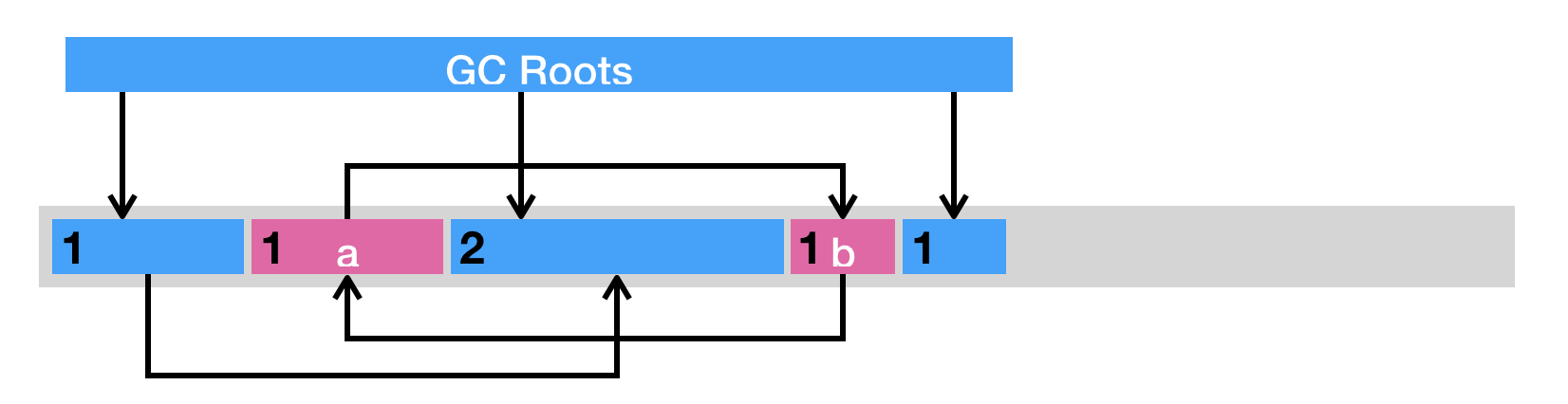
自动回收机制，肯定没法做到像手动回收那般精准高效，而且还会带来不少与垃圾回收实现的相关问题

# Part1

## 引用计数器和可达性分析

垃圾：死亡对象所占据的堆空间。

问题：如何辨别一个对象的死亡？

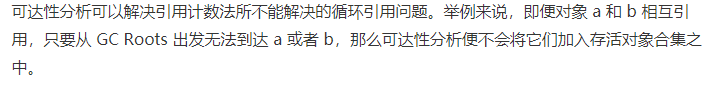
1. 引用计数法
   1. 为每个对象添加一个引用计数器，用来统计指向该对象的引用个数
   2. 一旦引用计数为0，那么对象已死亡，便可以回收了。
   3. 除了需要额外的存储空间来存储计数器，以及繁琐的更新操作，还有个漏洞：无法处理循环引用对象，a和b相互引用
   4. 
2. 可达性分析（目前的主流垃圾回收方法）

实质是将一系列GC Roots作为初始的存活对象合集（live set），然后从该集合出发，探索所有能够被该集合引用到的对象，并将其加入到该集合中。这个过程称之为标记（Mark），最终未被索引的对象就是死亡对象

GC Roots：

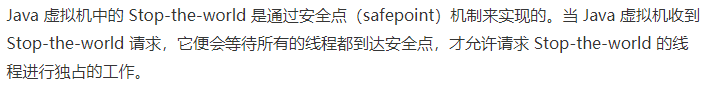
理解为堆外对堆内的引用，包括：

1. Java方法栈帧中的局部变量
2. 已加载类的静态变量
3. JNI handles；
4. 已启动且未停止的java线程



## Stop-the-world以及安全点

为解决多线程造成的漏报，stop-the-world是停止其他非垃圾回收线程的工作，直到垃圾回收完成，这就造成了垃圾回收的暂停时间（GC pause）



对于解释器来说，字节码与字节码之间均可作为安全点。Java虚机采取的做法是，当有安全点请求时，执行一条字节码便进行一次安全检测。

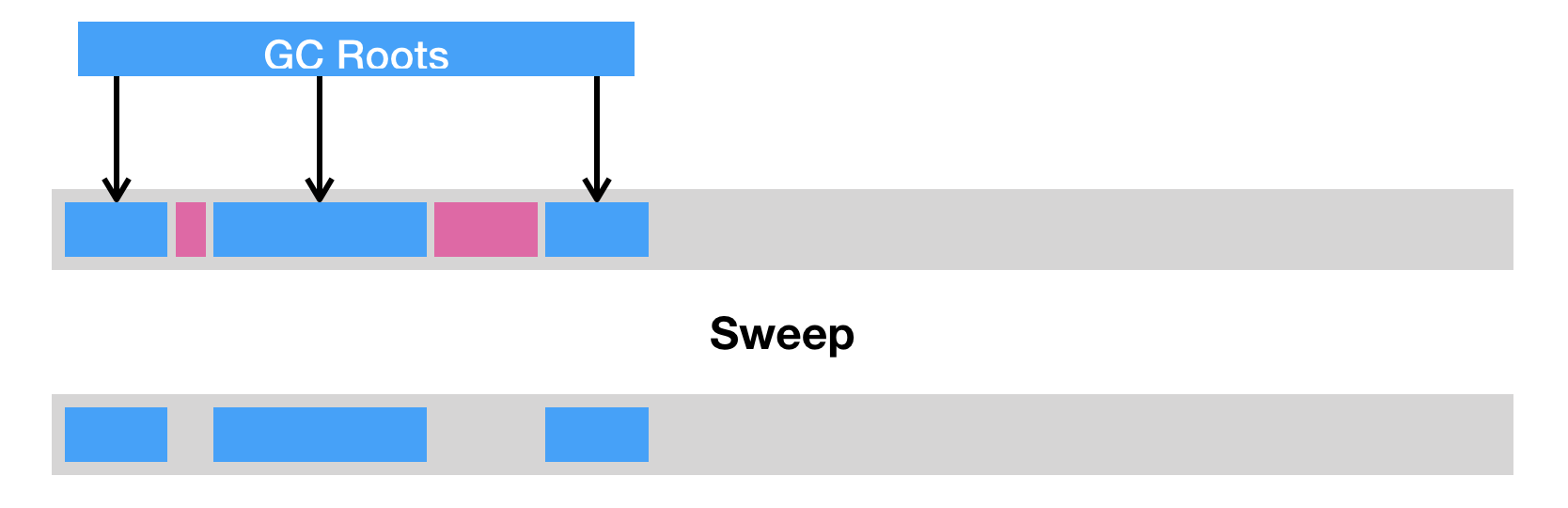
执行即时编译器生成的机器码则相应比较复杂。由于这些代码直接运行在底层硬件之上，不受java虚机掌控，因此在生成机器码时候，即时编译器需要插入安全点检测，以避免机器码长时间没有安全点检测的情况。HotSpot虚机做法就是在生成的代码出口以及非计数循环回边（back-edge）处插入安全检测点。

## 垃圾回收三种方式

### 清除（sweep）

将死亡对象所占据的内存标记为空闲内存，并记录在一个空闲列表之中（free list）

当需要新建对象时候，内存管理模块便会从该空闲列表中寻址，并划分给新建的对象。

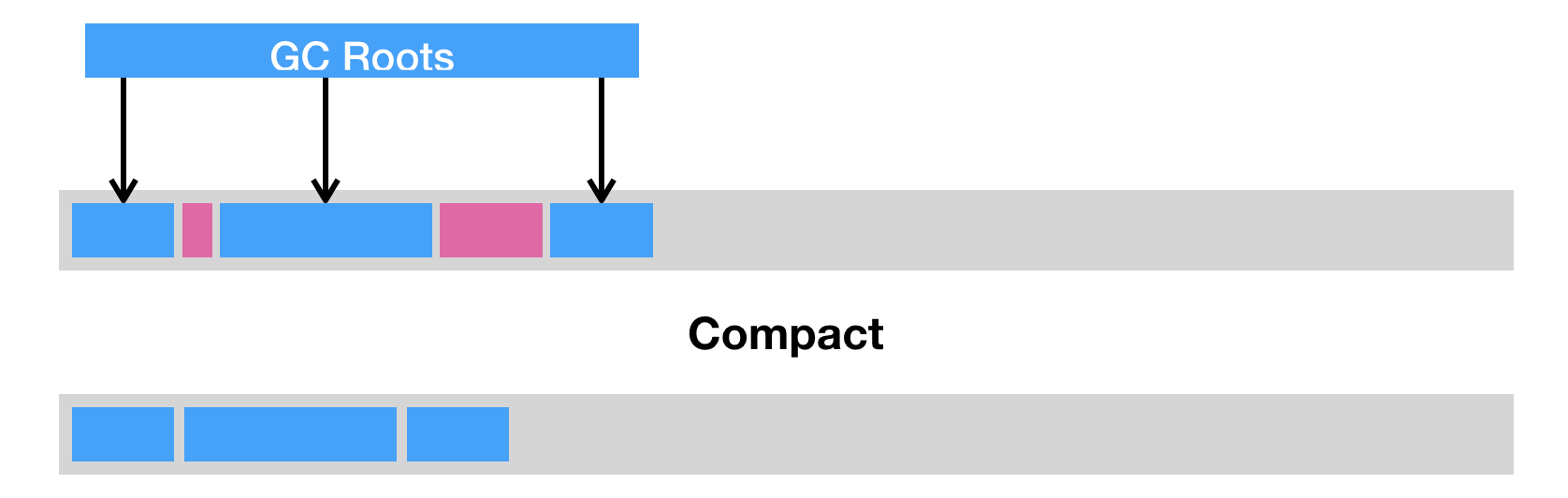


造成：内存碎片化，可能出现总内存足够，但是无法分配内存的情况。

缺点：分配效率低，如果是一块连续内存空间，我们可以通过指针加法来做分配。而对于空闲列表，虚机需要逐个访问空闲列表中的项，来查找能够放入新建对象的空闲内存。

### 压缩（compact）

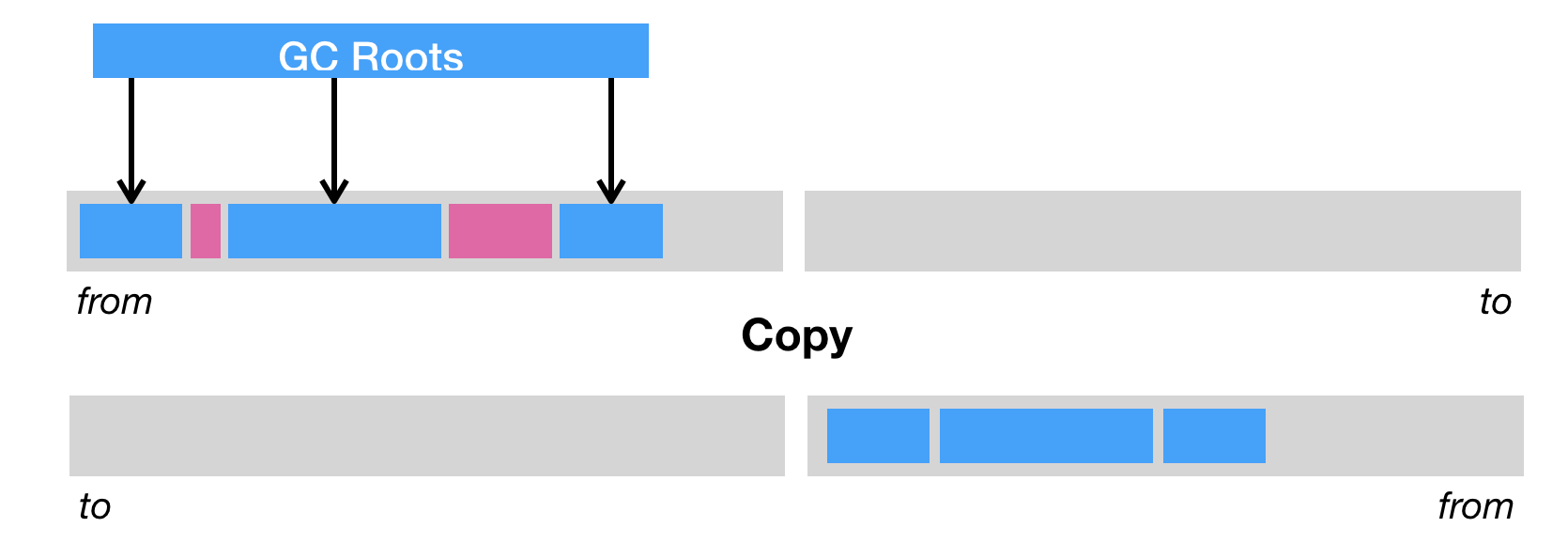
将存活的对象聚集到内存区域的起始位置，从而留下一段连续的内存空间，



特点：解决了内存碎片化的问题，但是代价是压缩算法的性能开销。

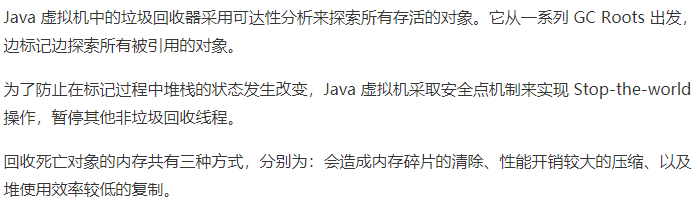
### 复制（copy）

将内存分为相等的两等分，分别用指针from和to来维护。并且只是用from指针执行内存区域来分配内存。当发生垃圾回收的时候，将存活对象复制到to指针指向的区域，然后交换from和to指针的内容，

特点：解决内存碎片化，空间使用效率极其低下。

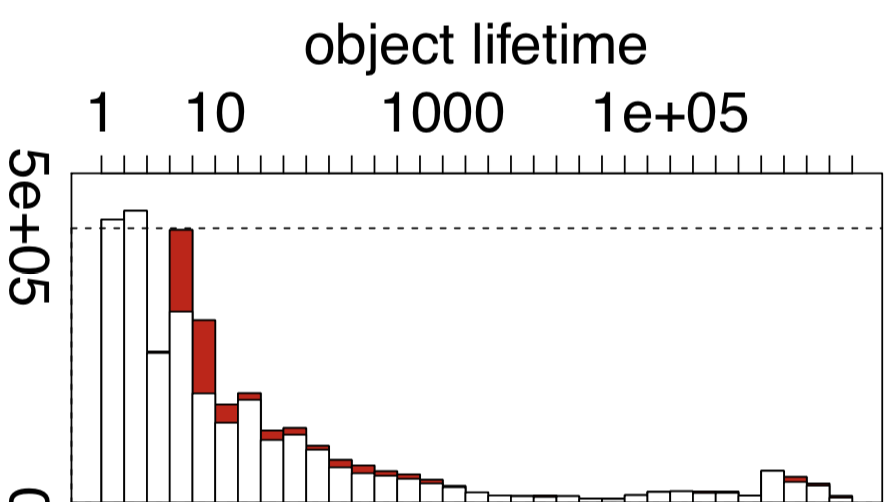
## 综上

目前所用的垃圾回收器会综合以上的优点，规避缺点。



# Part2

分代回收思想



（java生命周期直方图，红色表示被逃逸分析优化的部分）

堆空间：

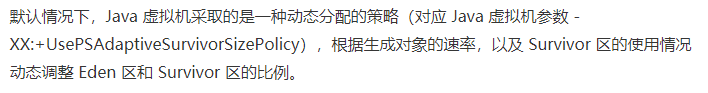
新生代：用来存储新建的对象

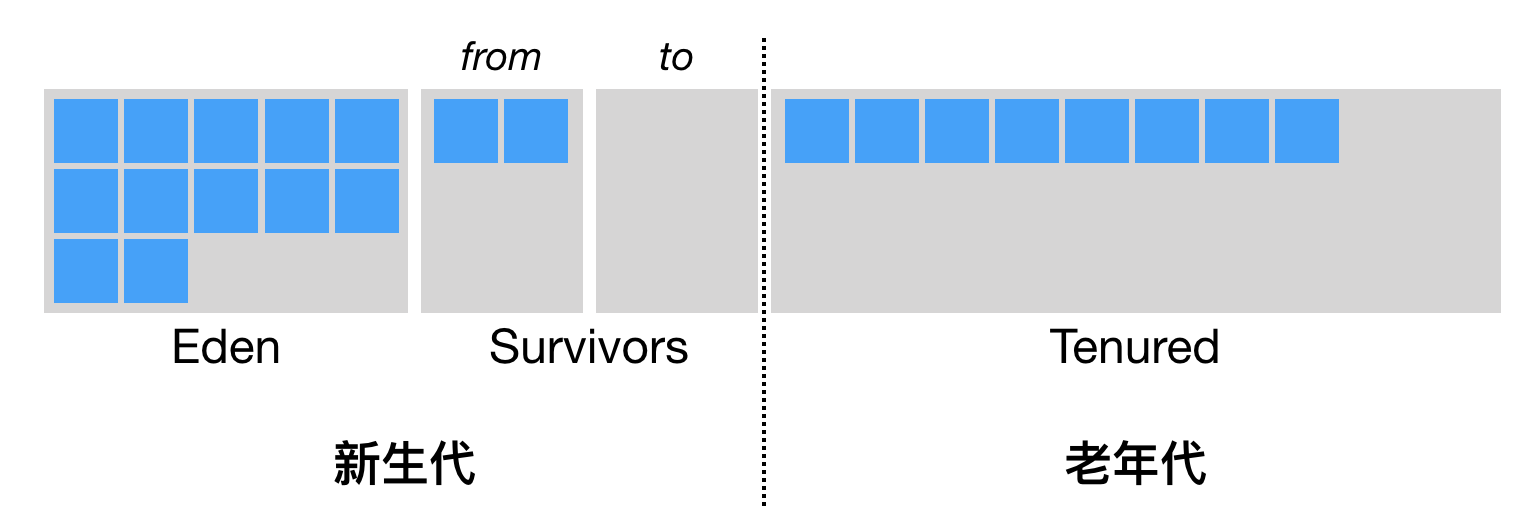
老生代：当对象存活时间足够长的时候，将移至老生代

虚机可以给不同代使用不同的回收算法，大部分java对象只存活一小段时间，那么便可以频繁采用耗时较短的回收算法。让大部分在新生代就回收掉。

当真正触发老年代回收的时候，则代表假设出错了，或者堆空间已经耗尽，那么就需要不计成本做一次全堆扫描。（现代的垃圾回收器都在并发收集的道路上发展，来避免这种全堆扫描的情况）

## Java虚机的堆划分





新生代包括Eden和两个大小相等的Survivors



## Minor GC（标记-复制算法）

当Eden区的空间耗尽了，这时java会触发一次Minor GC，来收集新生代的垃圾。存活下来的对象会被送到Survivors区中。To指向的survivor区是空的，当发生Minor GC的时候，Eden区和from指向的Survivor区会被复制到to指向的Survivor区中，然后交换from和to指针，以保证下次Minor GC时，to指向的Survivor区是空的。

Java虚机会记录Survivor区来回复制了几次，如果一个对象被复制的对象次数为15（-XX:MaxTenuringThreshold），那么将被晋升（promote）至老年代。

理想情况下，Eden区中的对象基本死亡，那么需要复制的数据非常少，因此采用这种标记-复制算法效果极好。

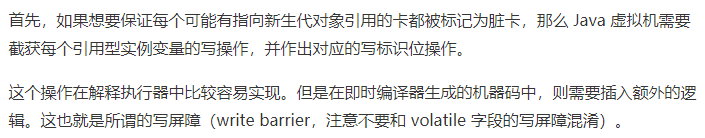
如果老年代的对象引用新生代的对象，那么在标记存活的时候需要扫描老年代中的对象，然后这个引用也作为GC Roots。

引发全堆扫描？？？

## 卡表

HotSpot给出解决方案是（Card Table）的技术。将堆划分为大小为512字节的卡，并且维护一个卡表，用来存储每张卡的一个标识位。这个标识位代表对应的卡是否可能存有指向新生代对象的引用。如果可能存在，那么我们认为这行卡是脏的。

在Minor GC时，不需扫描整个老年代，只需找脏卡，将其中的对象加入到Minor GC中的GC Roots里。当完成所有java扫描后，虚机便会将所有标识位清零。



HotSpot会在对象的头中标记字段里面记录年龄（GC的次数），分配到的空间只有4位，所以最多只能记录15。当Minor GC15次之后还存活的对象会放到老生代。