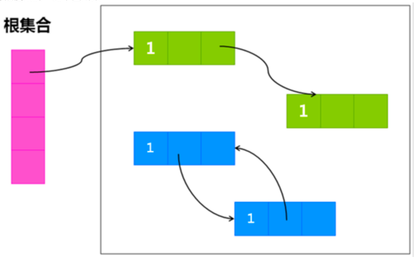
一. 可回收对象的判定

1. **引用计数法**

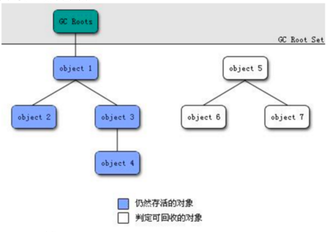
给对象中添加一个引用计数器，每当有一个地方引用它时，计数器值就加1；当引用失效时， 计数器值就减1；任何时刻计数器为0的对象就是不可能再被使用的。如下图所示：



优点：引用计数收集器可以很快地执行，交织在程序的运行之中。这个特性对于程序不能被长时间打断的实时环境很有利。
缺点：很难处理循环引用，比如图中相互引用的两个对象则无法释放。
应用：Python 和 Swift 采用引用计数方案。

2. **可达性分析算法（根搜索算法）**

从GC Roots（每种具体实现对GC Roots有不同的定义）作为起点，向下搜索它们引用的对象，可以生成一棵引用树，树的节点视为可达对象，反之视为不可达。如下图所示：



GC Roots对象
1). 虚拟机栈（帧栈中的本地变量表）中引用的对象。
2). 方法区中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引用的对象。
本地方法栈则为虚拟机所使用的Native方法服务。

二. 可回收对象的判定

1. **保守法**

将所有堆上对齐的字都认为是指针，那么有些数据就会被误认为是指针。于是某些实际是数字的假指针，会背误认为指向活跃对象，导致内存泄露（假指针指向的对象可能是死对象，但依旧有指针指向——这个假指针指向它）同时我们不能移动任何内存区域。

2. **编译器提示法**

如果是静态语言，编译器能够告诉我们每个类当中指针的具体位置，而一旦我们知道对象时哪个类实例化得到的，就能知道对象中所有指针。这是JVM实现垃圾回收的方式，但这种方式并不适合JS这样的动态语言

3. **标记指针法**

标记指针法：这种方法需要在每个字末位预留一位来标记这个字段是指针还是数据。这种方法需要编译器支持，但实现简单，而且性能不错。V8采用的是这种方式。

4. **位图标记(Go语言为例)**

非侵入式标记位定义：既然垃圾回收算法要求给对象加上垃圾回收的标记，显然是需要有标记位的。一般的做法会将对象结构体中加上一个标记域，一些优化的做法会利用对象指针的低位进行标记，这都只是些奇技淫巧罢了。Go没有这么做，它的对象和C的结构体对象完全一致，使用的是非侵入式的标记位。

具体实现：堆区域对应了一个标记位图区域，堆中每个字(不是byte，而是word)都会在标记位区域中有对应的标记位。每个机器字(32位或64位)会对应4位的标记位。因此，64位系统中相当于每个标记位图的字节对应16个堆中的字节。虽然是一个堆字节对应4位标记位，但标记位图区域的内存布局并不是按4位一组，而是16个堆字节为一组，将它们的标记位信息打包存储的。每组64位的标记位图从上到下依次包括

- 16位的 特殊位 标记位
- 16位的 垃圾回收 标记位
- 16位的 无指针/块边界 的标记位
- 16位的 已分配 标记位

这样设计使得对一个类型的相应的位进行遍历很容易。前面提到堆区域和堆地址的标记位图区域是分开存储的，其实它们是以mheap.arena_start地址为边界，向上是实际使用的堆地址空间，向下则是标记位图区域。以64位系统为例，计算堆中某个地址的标记位的公式如下：

- 偏移 = 地址 - mheap.arena_start
- 标记位地址 = mheap.arena_start - 偏移/16 - 1
- 移位 = 偏移 % 16
- 标记位 = *标记位地址 >> 移位

然后就可以通过 (标记位 & 垃圾回收标记位),(标记位 & 分配位),等来测试相应的位。(也就是说，本来64位是一个字，需要4位标记位。但是，为了与字长相对，16个标记位放一起（刚好一个字长）一起表示16个字。并且每类标记位都放在一起AA..AABB...BB)

5. **接下来补充几个概念帮助理解：**

1). 为什么要判断哪些是数据，哪些是指针？

假设堆中有一个long的变量，它的值是8860225560。但是我们不知道它的类型是long，所以在进行垃圾回收时会把个当作指针处理，这个指针引用到了0x2101c5018位置。假设0x2101c5018碰巧有某个对象，那么这个对象就无法被释放了。即使实际上已经没任何地方使用它。由于没有类型信息，我们并不知道这个结构体成员不包含指针，因此我们只能对结构体的每个字节递归地标记下去，这显然会浪费很多时间。

2). 垃圾收集器（CMS收集器为例）几个阶段

- 初始标记：初始标记仅仅是标记一下GC Roots能直接关联到的对象，速度很快
- 并发标记：并发标记就是进行GC Roots Trancing的过程
- 最终标记：为了修正并发标记期间因用户程序继续运行而导致标记产生变动那一部分对象的标记记录，这个阶段的停顿时间比初始标记稍长一些，但远比并发标记时间短
- 筛选回收

3). stop the world

因为垃圾回收的时候，需要整个的引用状态保持不变，否则判定是垃圾，等我稍后回收的时候它又被引用了，这就全乱套了。所以，GC的时候，其他所有的程序执行处于暂停状态，卡住了。这个概念提前引入，在这里进行对比，效果会更好些。与标记阶段对比，stop the world发生在回收阶段。

三. 垃圾回收算法

1. **标记清除算法 (Mark-Sweep)**

标记-清除算法分为两个阶段：标记阶段和清除阶段。
标记阶段的任务是标记出所有需要被回收的对象，清除阶段就是回收被标记的对象所占用的空间。
优点是简单，容易实现。缺点是容易产生内存碎片，碎片太多可能会导致后续过程中需要为大对象分配空间时无法找到足够的空间而提前触发新的一次垃圾收集动作。（因为没有对不同生命周期的对象采用不同算法，所以碎片多，内存容易满，gc频率高，耗时，看了后面的方法就明白了）

2. **分代回收算法**

根据对象存活的生命周期将内存划分为若干个不同的区域。不同区域采用不同算法（复制算法，标记整理算法），这就是分代回收算法。
一般情况下将堆区划分为老年代（Old Generation）和新生代（Young Generation），老年代的特点是每次垃圾收集时只有少量对象需要被回收，而新生代的特点是每次垃圾回收时都有大量的对象需要被回收，那么就可以根据不同代的特点采取最适合的收集算法。

1). 新生代回收
新生代使用Scavenge算法进行回收。在Scavenge算法的实现中，主要采用了Cheney算法。

- Cheney算法是一种采用复制的方式实现的垃圾回收算法。它将内存一分为二，每一部分空间称为semispace。在这两个semispace中，一个处于使用状态，另一个处于闲置状态。
- 简而言之，就是通过将存活对象在两个semispace空间之间进行复制。复制过程采用的是**BFS（广度优先遍历）的思想，从根对象出发，广度优先遍历所有能到达的对象。**
- 优点：时间效率上表现优异（牺牲空间换取时间）
- 缺点：只能使用堆内存的一半

2). 老生代回收
Mark-Sweep（标记清除）：标记清除分为标记和清除两个阶段。主要是标记清除只清除死亡对象，而死亡对象在老生代中占用的比例很小，所以效率较高。
Mark-Compact（标记整理）：标记整理正是为了解决标记清除所带来的内存碎片的问题。大体过程就是 双端队列标记黑（邻接对象已经全部处理），白（待释放垃圾），灰（邻接对象尚未全部处理）三种对象。标记算法的核心就是深度优先搜索.

3. **扩展问题**

1).触发GC时机
一般都是内存满了就回收，下面列举几个常见原因：

- GC_FOR_MALLOC: 表示是在堆上分配对象时内存不足触发的GC。
- GC_CONCURRENT: 当我们应用程序的堆内存达到一定量，或者可以理解为快要满的时候，系统会自动触发GC操作来释放内存。
- GC_EXPLICIT: 表示是应用程序调用System.gc、VMRuntime.gc接口或者收到SIGUSR1信号时触发的GC。
- GC_BEFORE_OOM: 表示是在准备抛OOM异常之前进行的最后努力而触发的GC。

2). 写屏障（一个老年代的对象需要引用年轻代的对象，该怎么办？）
如果新生代中的一个对象只有一个指向它的指针，而这个指针在老生代中，我们如何判断这个新生代的对象是否存活？为了解决这个问题，需要建立一个列表用来记录所有老生代对象指向新生代对象的情况。每当有老生代对象指向新生代对象的时候，我们就记录下来。
当垃圾回收发生在年轻代时，只需对这张表进行搜索以确定是否需要进行垃圾回收，而不是检查老年代中的所有对象引用。

3). 深度、广度优先搜索（为什么新生代用广度搜索，老生代用深度搜索）
深度优先DFS—般采用递归方式实现，处理tracing的时候，可能会导致栈空间溢出，所以一般采用广度优先来实现tracing（递归情况下容易爆栈）。广度优先的拷贝顺序使得GC后对象的空间局部性（**memory** locality）变差（相关变量散开了）。广度优先搜索法一般无回溯操作，即入栈和出栈的操作，所以运行速度比深度优先搜索算法法要快些。深度优先搜索法占内存少但速度较慢，广度优先搜索算法占内存多但速度较快。