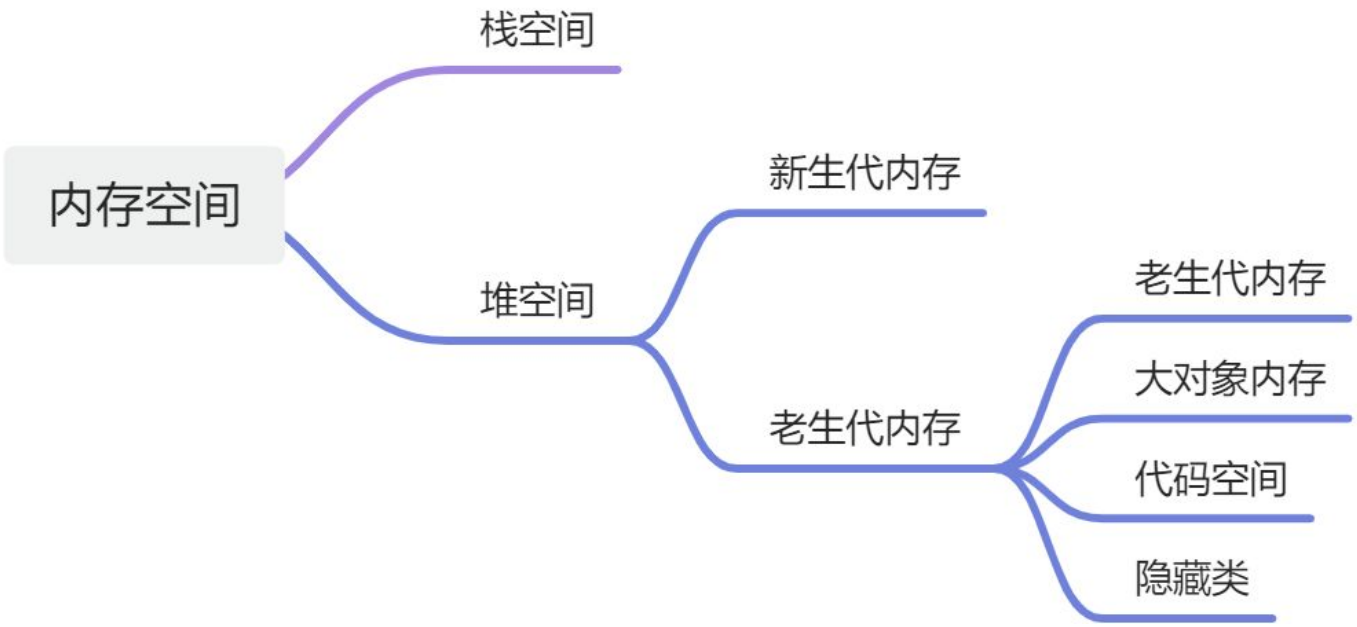
V8引擎：垃圾回收机制

> 💡 JavaScript 的 Orinoco 垃圾回收（Garbage Collection）机制是 V8 引擎的核心功能，主要用于自动管理内存，释放不再使用的对象所占用的内存空间

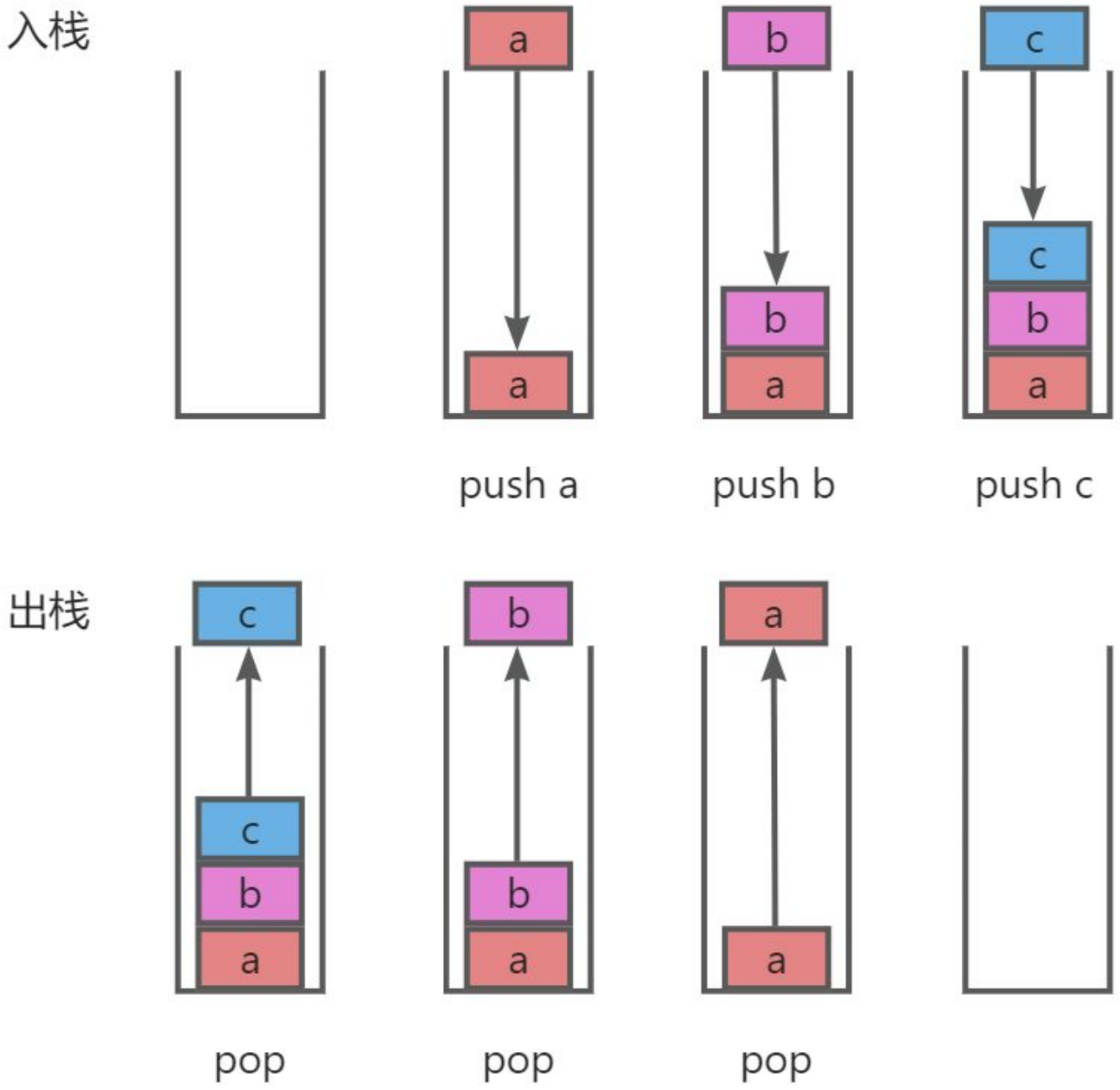
了解垃圾回收机制之前先预览一下这些概念：

内存空间：



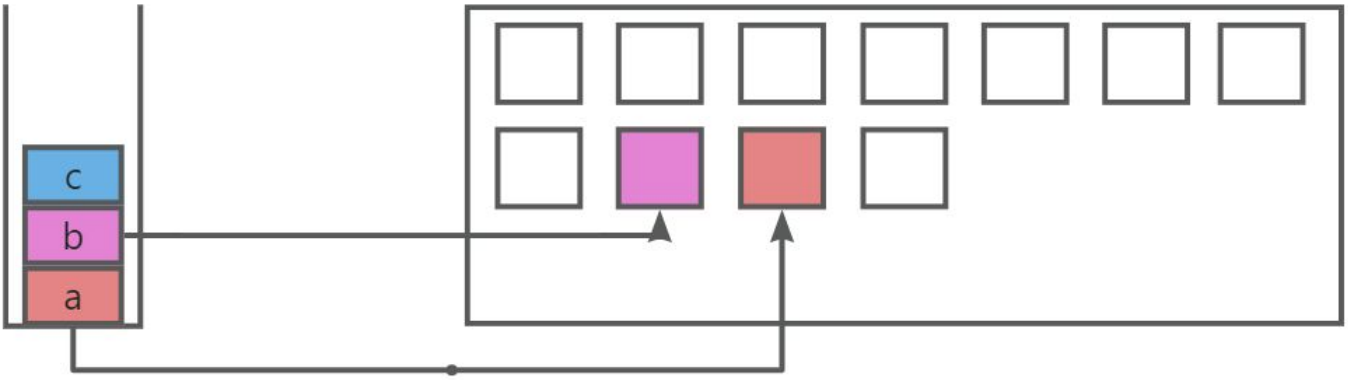
- 堆与栈属于内存管理单元
- 在 JavaScript 中，基础数据类型和引用数据类型的地址存储在栈中，而引用数据类型的值存储在堆中（包括在全局作用域下声明的全局变量，因为全局变量会被作为属性赋值给全局对象）
- 就容量而言，堆空间上限远大于栈空间

栈空间 （stack）：后进先出



无论是全局或是函数环境，代码运行时会创建执行上下文，并压入栈中，当代码执行完成，当前环境下的执行上下文又自动出栈，释放空间，所以垃圾回收机制并不需要关注它

堆空间（heap）：垃圾回收机制关注的对象



栈空间的值存储引用数据类型在堆中的地址

世代假说：

引擎开发团队通过对内存使用情况的观察得出以下结论：

1. 大多数新创建的对象会在分配内存后很快消亡
2. 老的对象活得更久

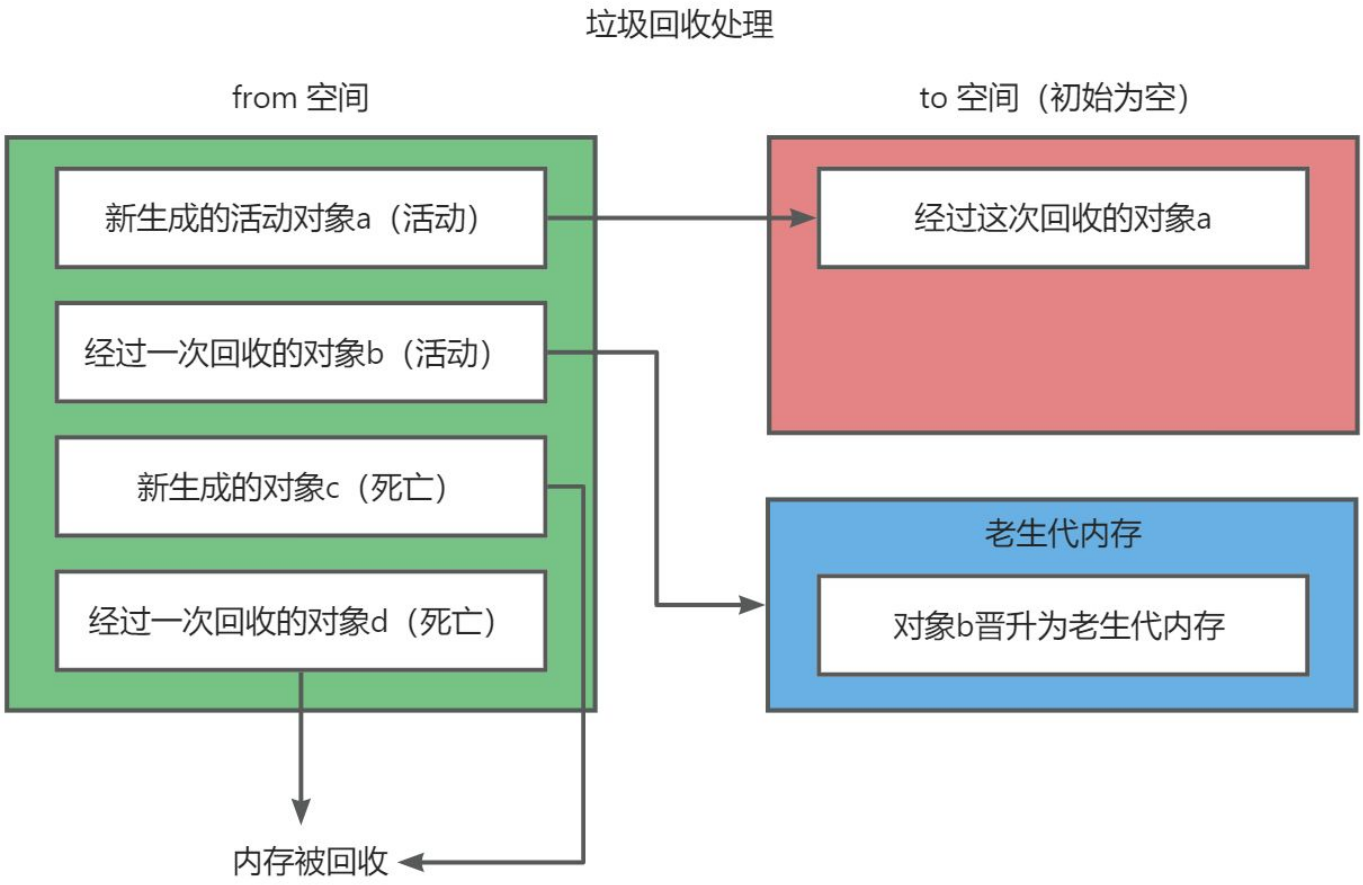
因此根据存活时间将内存分为新生代和老生代两种类型

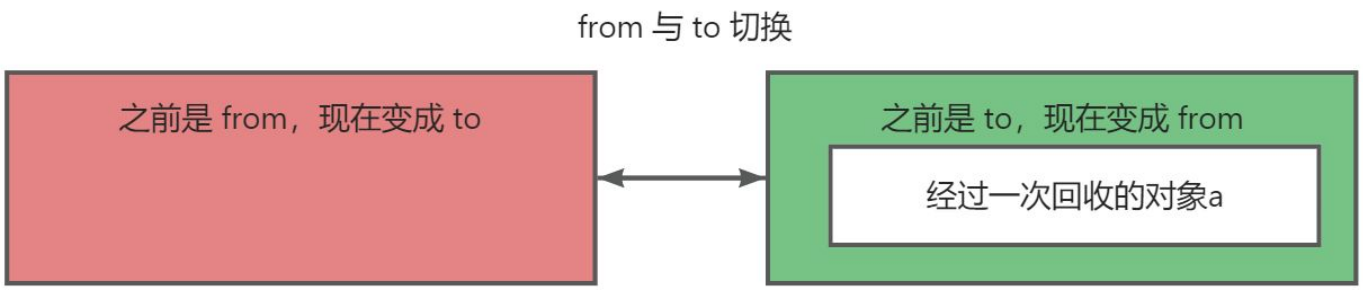
1. 新生代内存：新出生或是只经过一次垃圾回收的对象占据的内存
2. 老生代内存：经过两次或更多垃圾回收的对象占据的内存

新生代垃圾回收（Minor GC）：

清道夫（Scavenger）算法：

1. 算法将新生代内存平分为 from 和 to 两个相同大小的空间，其中 from 为工作空间，to 为闲置空间，64 位的 Chrome 中每个的空间为 16M，共计 32M
2. 初始时，所有对象都会存储在 from 空间内
3. 垃圾回收机制检查 from 空间，将不再活跃的对象清除，将存活的对象将送往 to 空间，若是满足以下条件则将其晋升为老生代内存
    a. 如果该对象之前已经经过一次垃圾回收
    b. to 空间达到预设的上限 25%（并非内存容量上限，因为下次垃圾回收前还需要留出空间继续收容新生成的对象）
4. to 空间与 from 空间指针切换，可以理解为 to 空间未晋级的对象又回到了 from 空间



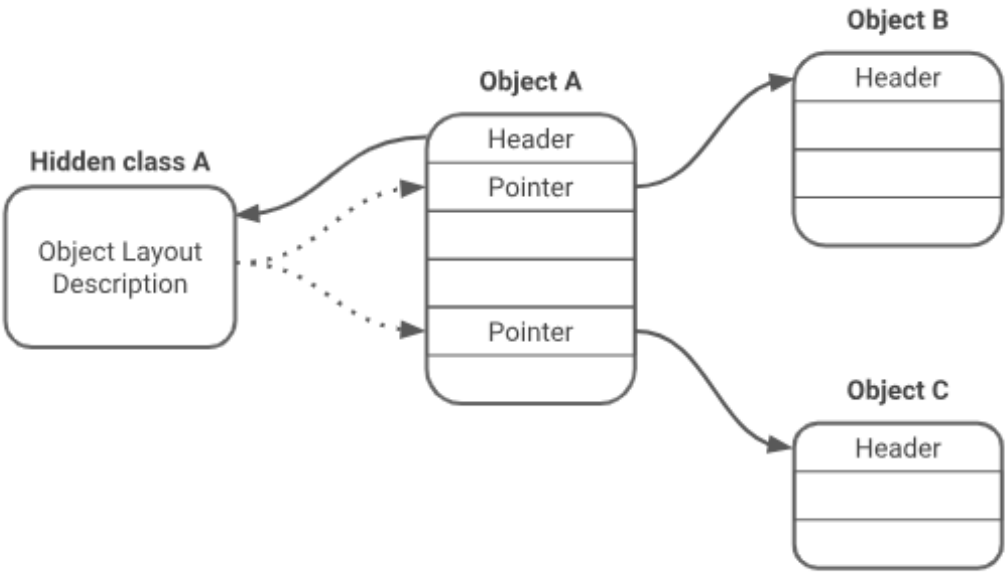


老生代垃圾回收（Minor GC）：

- 容量：老生代内存空间比新生代大得多，64 位 chrome 中容量上限为 1.4 G
- 分类：老生代内存被分为以下几类
  a. old object space：由寻常新生代内存晋级而来的老生代内存
  b. large object space：用于存储体积过大的对象，大对象通常不会通过拾荒算法进行筛选，而是直接进入该空间
  c. Map space：用于存储隐藏类（隐藏类是 V8 的优化方案，参考了 C++ 中隐藏类的设定，将对象视作属性不可变，使用偏移量来计算属性值）

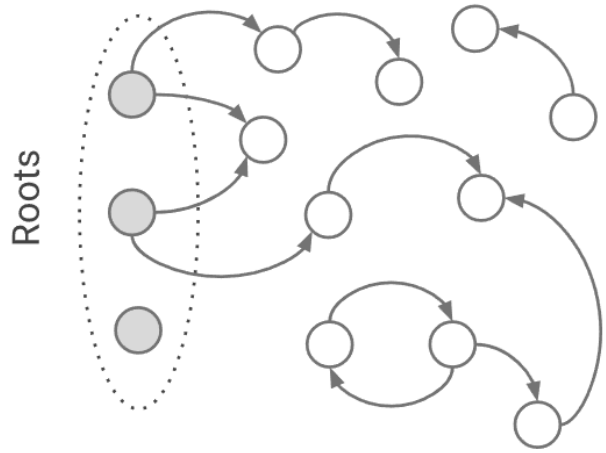
d. code space：代码空间，存储可执行的代码（包括引擎解释器生成的字节码，编译器生成的机器码）

标记清除（Mark-sweep）：三色标记法

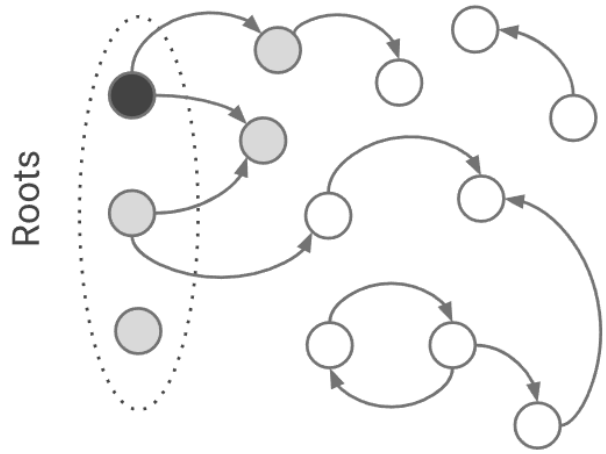


图示 1. 对象关系图

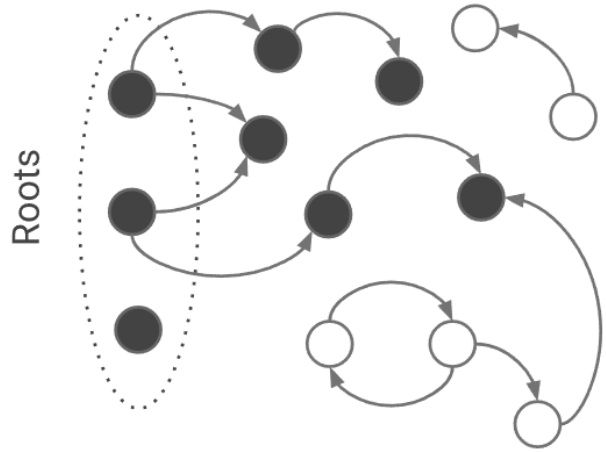
1. 图遍历：标记过程可以认为是图遍历，堆上的对象是图的节点，而一个对象指向另一个对象是图的边，可以使用对象的隐藏类（Hidden class）找出节点所有的外出边
2. 遍历过程：从根（已知的活动对象，例如全局对象和当前活动函数）开始，通过切换指针去访问老生代内存中的对象，并给予标记，直至所有可以访达（可以理解为被引用了）的对象标记完毕，此时应用程序将无法访问堆中未被标记的对象，从而实行安全回收
3. 标记工作表（marking-worklist）：栈结构，用于存储遍历过程中可访达的对象，当发现一个可访达的对象时，会将它压入标记工作表内



图示 2. 从根节点开始标记



图示 3. 收集器通过处理其指针将灰色对象变为黑色

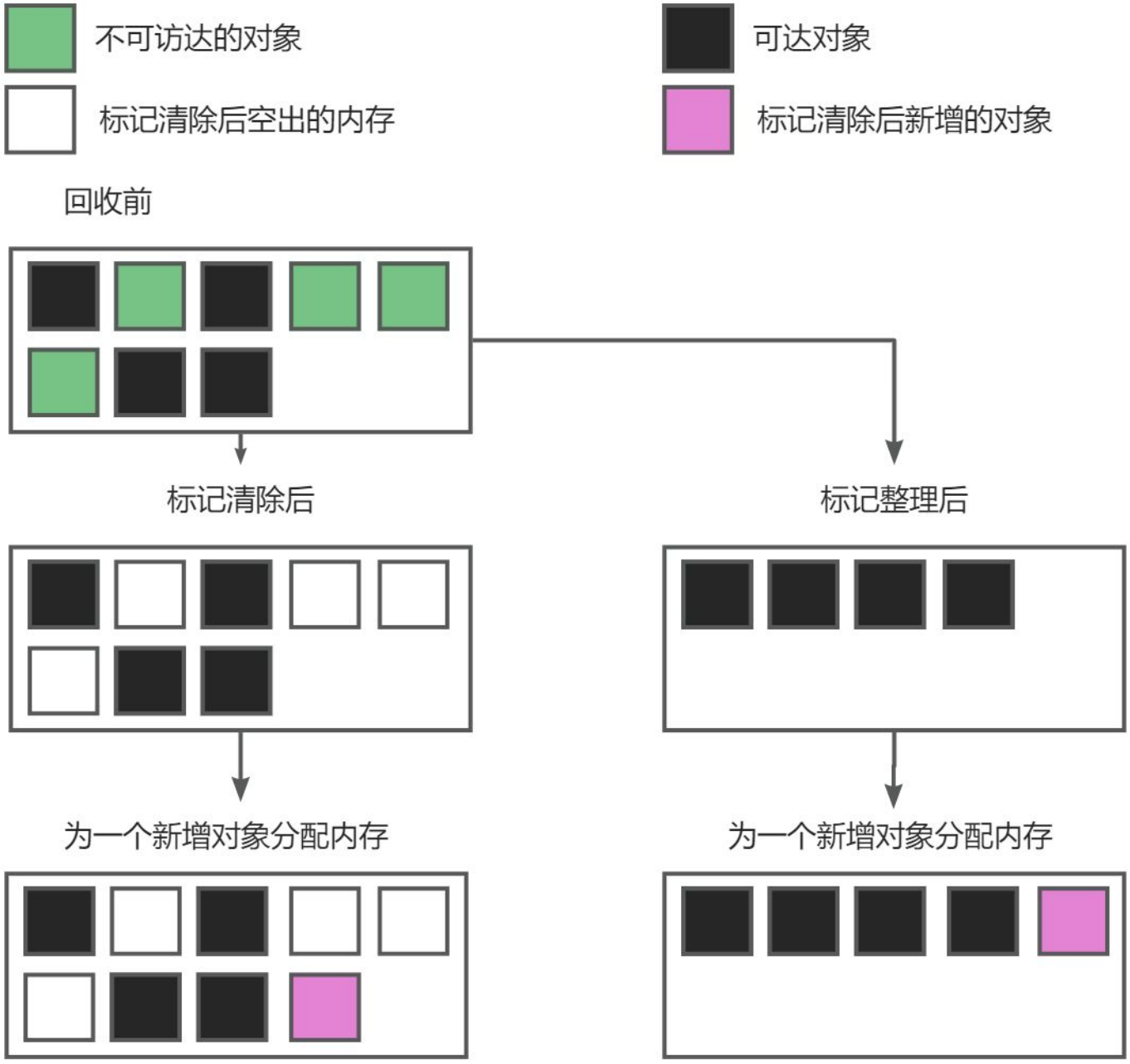


图示 Figure 4. 标记完成后的最终状态

1. 白：初始时，根对象为黑色，而其它所有的对象均为白色，意味着收集器尚未发现它们
2. 灰：当收集器发现一个可访达的对象时，将其标记为灰色并推入标记工作表中
3. 黑：当标记工作表中弹出对象并访问它的所有属性时，此时灰色将变为黑色
4. 白：当所有灰色都转为黑色后，剩余的白色意味着无法访达，可以被回收

标记整理（Mark-compact）：标记清除的优化

1. 原因：标记清除是老生代内存的主要处理算法，但它存在一个缺点：磁盘会变得碎片化，被回收的内存散乱零落地分布在堆中角落中，因此引擎无法很好地利用这些空间（为新生成的对象分配内存），因此设计了标记整理算法来整理内存空间
2. 实现：在标记的基础上，将剩余的可达对象整理并置放在一起，再回收剩余的内存空间
3. 缺陷：但整理算法（移动对象）开销非常昂贵，因此仅作为标记清除的补充，只会在剩余内存空间不足的情况下使用

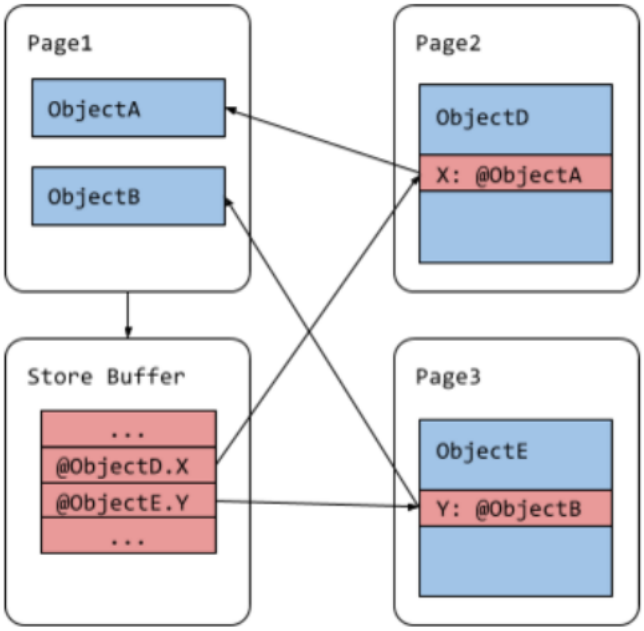


除了标记整理以外，Orinoco 还采取了以下优化方案

并行记忆集：

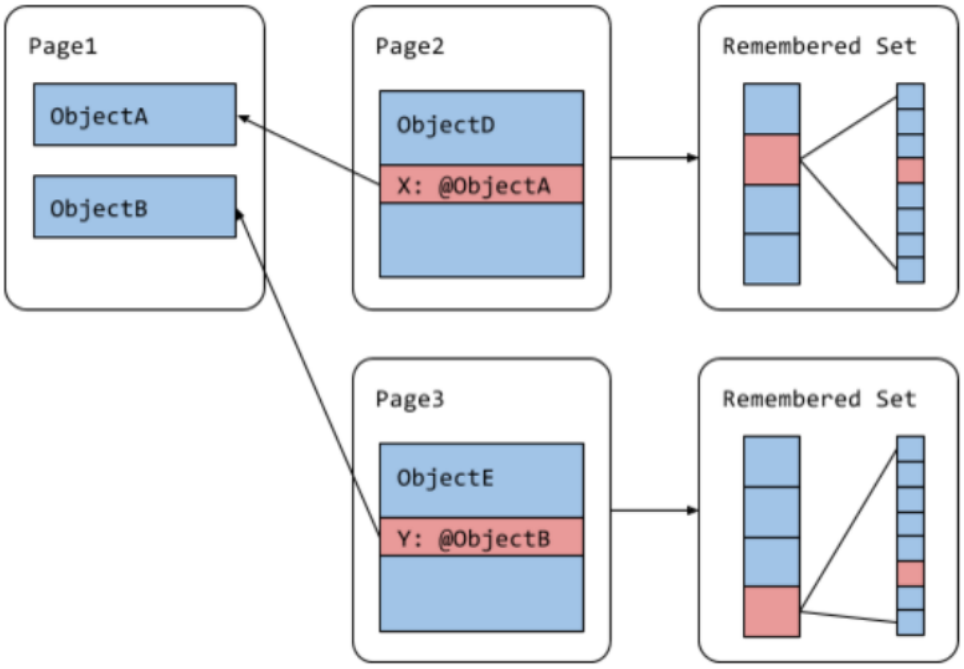
1. 当一个对象在堆上移动位置时，垃圾回收器必须找到包含被移动对象的旧位置的所有指针，并用新位置更新它们（因为地址改变，所有引用到该对象的对象或栈中的数据需要更新指针，即内存地址）
2. 但遍历堆查找指针过程非常缓慢，因此 V8 使用称为记忆集（remembered set） 的数据结构来跟踪堆上所有有意义的指针。
3. 如果指针指向在垃圾回收期间可能移动的对象，则该指针是有意义的。例如，新生代内存需要在 from 与 to 空间中移动；指向高度分散的页面中的对象的指针也很有意义，因为这些对象在标记整理的过程中将移动到其它地方

旧的记忆集：



1. 旧记忆集的实现方式为存储缓冲区（store buffers）数组，为新生代和每个零散的老生代页面设置一个存储缓存区，里面存储了所有传入指针的地址
2. 条目被附加到写屏障（write barrier）中的存储缓冲区中，以保护 JavaScript 代码中的写操作，但着可能导致重复的条目
3. 两个不同的存储缓冲区可能包含相同的指针，导致指针更新阶段的并行化变得很困难，因为两个线程试图更新同一指针会导致数据争用

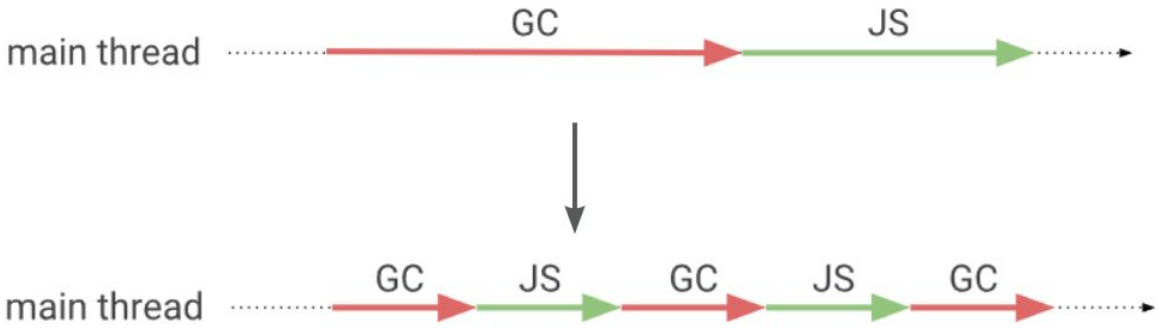
新的记忆集：



1. 现在每个页面不再将有意义的指针存储在数组中，而是将源自该页面的有意义的指针的偏移量存储在位图的存储区（buckets）中
2. 现在每个存储区指向固定长度的位图或者为空，位图中的一位对应着指针偏移量
3. 这种数据结构使得可以基于页面并行化指针更新，能够删除复杂的代码来处理记忆集溢出（remembered set overflow）

增量标记：从全停顿（stop-the-world）转换为增量标记

1. 原因：GC 会占用主线程，过长时间的停顿会导致应用无响应造成卡顿
2. 实现：当老生代中活动对象的数量接近堆限制时，开始增量标记。期间，垃圾收集器将标记工作分解为更小的块，标记工作与 js 代码交替执行，因此可以在主线程空闲时间调度执行标记工作



3. 代价：
   a. 应用程序需要通知 GC 关于改变对象图的所有操作，V8 通过 Dijkstra 风格的写屏障（write-

barrier）机制来实现通知

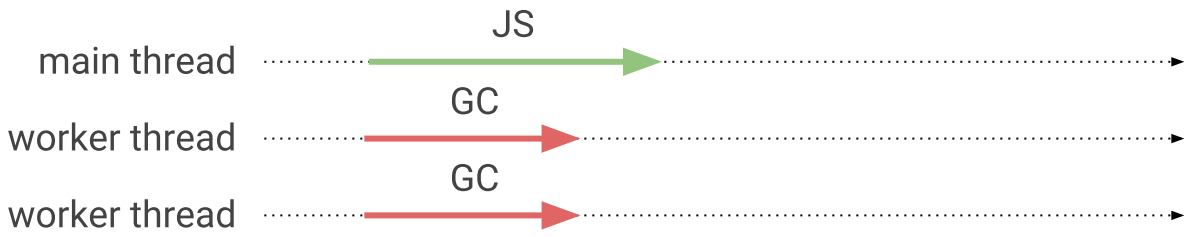
b. 强三色不变性：write-barrier 强制不变黑的对象指向白色对象，保证应用程序不能在垃圾收集器中隐藏活动对象，因此标记结束时的所有白色对象对于应用程序来说都是不可达的，可以安全释放

```C++
// 调用 `object.field = value` 之后
write_barrier(object, field_offset, value) {
    if (color(object) == black && color(value) == white) {
        set_color(value, grey);
        marking_worklist.push(value);
    }
}
```

c. 但由于 write-barrier 机制的成本，增量标记会降低应用程序吞吐量

并发标记：增加应用吞吐量

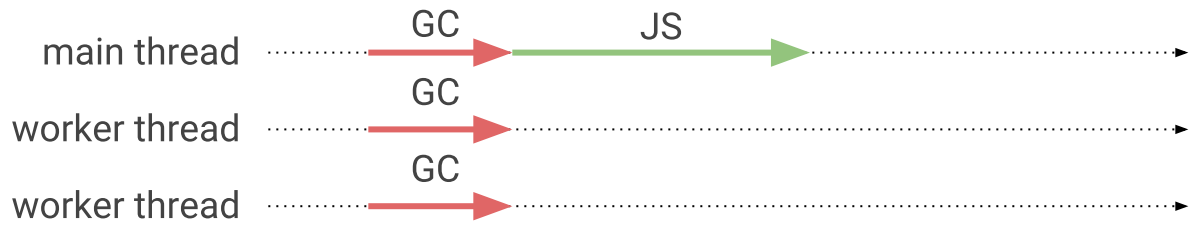
并发标记发生在工作线程上，当并发标记进行时，应用程序可以继续运行



并发标记过程

并行标记：增加应用吞吐量

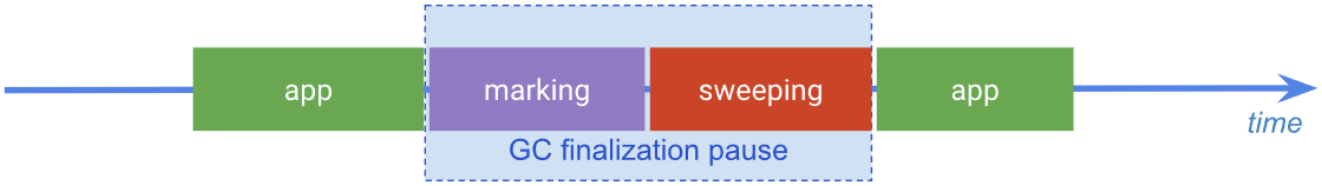
并行标记发生在主线程和工作线程上，应用程序在整个并行标记阶段暂停



并行标记过程

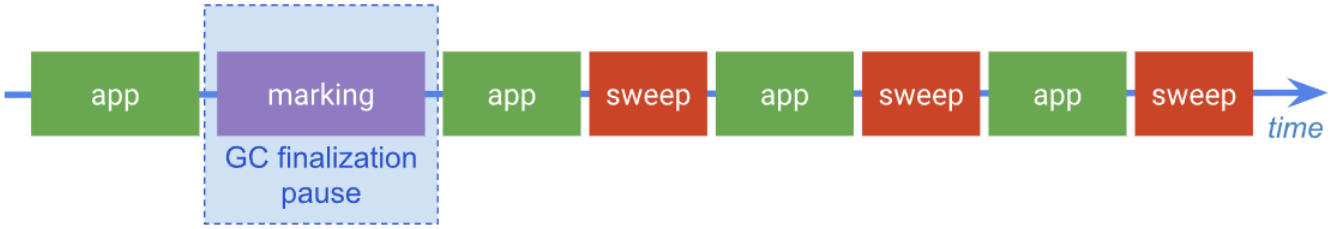
增量清除：

全量清除阻塞 js 线程的执行导致卡顿



全量清除（Stop-the-world sweeping）

降低清除导致的暂停时间对 js 线程的影响



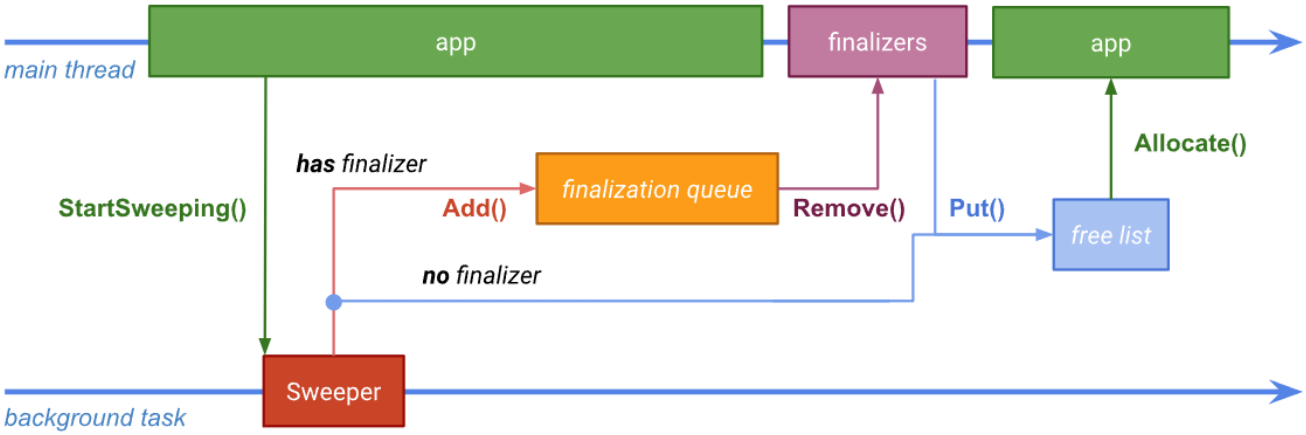
增量清除（Incremental sweeping）

并发清除：多线程并发

随着应用程序及生成对象的增大，清除开始影响应用程序的性能，为改善增量清除，同时采用后台任务来进行同时回收内存

但为避免清除程序的后台任务与分配新对象的应用程序之间的数据争用，需遵守以下原则：

1. 清除程序仅处理死亡内存，根据定义死亡内存是应用程序无法访问的
2. 该应用程序仅在已经清除的页面上分配内存，根据定义清除程序将不再处理这些页面



使用后台任务的并发清除（Concurrent sweeping）

终结器（finalizer）：可能需要访问对象的所有有效负载，因此将相应的内存添加到空闲列表将延迟到执行终结器之后。如果没有可被执行的终结器，则在后台线程上运行的清除程序会立即将回收的内存添加到空闲列表中

以上是关于 V8 垃圾回收机制的简单介绍，如果需要了解更详尽的内容，可以浏览 V8 官方文档

参考资料

1. https://v8.js.cn
2. 《深入浅出 Node.js》