# V8 之旅： 垃圾回收器

在之前的几篇文章当中，我们深入了V8引擎的实现，讨论了Full Compiler、Crankshaft以及对象的内部表达。在这篇文章当中，我们来看看V8的 垃圾回收器 。

*本文来自Jay Conrod的A tour of V8: Garbage Collection，其中的术语、代码请以原文为准。*

垃圾回收器是一把十足的双刃剑。其好处是可以大幅简化程序的内存管理代码，因为内存管理无需程序员来操作，由此也减少了（但没有根除）长时间运转的程序的内存泄漏。对于某些程序员来说，它甚至能够提升代码的性能。

另一方面，选择垃圾回收器也就意味着程序当中无法完全掌控内存，而这正是移动终端开发的症结。对于JavaScript，程序中没有任何内存管理的可能——ECMAScript标准中没有暴露任何垃圾回收器的接口。网页应用既没有办法管理内存，也没办法给垃圾回收器进行提示。

严格来讲，使用垃圾回收器的语言在性能上并不一定比不使用垃圾回收器的语言好或者差。在C语言中，分配和释放内存有可能是非常昂贵的操作，为了使分配的内存能够在将来释放，堆的管理会趋于复杂。而在托管内存的语言中，分配内存往往只是增加一个指针。但随后我们就会看到，当内存耗尽时，垃圾回收器介入回收所产生的巨大代价。一个未经琢磨的垃圾回收器，会致使程序在运行中出现长时间、无法预期的停顿，这直接影响到交互系统（特别是带有动画效果的）在使用上的体验。引用计数系统时常被吹捧为垃圾回收机制的替代品，但当大型子图中的最后一个对象的引用解除后，同样也会有无法预期的停顿。而且引用计数系统在频繁执行读取、改写、存储操作时，也会有可观的性能负担。

或好或坏，JavaScript需要一个垃圾回收器。V8的垃圾回收器实现现在已经成熟，其性能优异，停顿短暂，性能负担也非常可控。

## 基本概念

垃圾回收器要解决的最基本问题就是，辨别需要回收的内存。一旦辨别完毕，这些内存区域即可在未来的分配中重用，或者是返还给操作系统。一个对象当它不是处于活跃状态的时候它就死了（废话）。一个对象处于活跃状态，当且仅当它被一个根对象或另一个活跃对象指向。根对象被定义为处于活跃状态，是浏览器或V8所引用的对象。比如说，被局部变量所指向的对象属于根对象，因为它们的栈被视为根对象；全局对象属于根对象，因为它们始终可被访问；浏览器对象，如DOM元素，也属于根对象，尽管在某些场合下它们只是弱引用。

从侧面来说，上面的定义非常宽松。实际上我们可以说，当一个对象可被程序引用时，它就是活跃的。比如：

```
function f() {
  var obj = {x: 12};
  g();   // 可能包含一个死循环
  return obj.x;
}
```

*译注：这里的`obj.x`和`obj`都是活跃的，尽管对其的再度引用是在死循环之后。*

很遗憾，我们无法精确地解决这个问题，因为这个问题实际等价于停机问题，无法确定。因此我们做一个等价约定：如果一个对象可经由某个被定义为活跃对象的对象，通过某个指针链所访问，则它就是活跃的。其他的都被视为垃圾。

# 堆的构成

在我们深入研究垃圾回收器的内部工作原理之前，首先来看看堆是如何组织的。V8将堆分为了几个不同的区域：

- **新生区**：大多数对象被分配在这里。新生区是一个很小的区域，垃圾回收在这个区域非常频繁，与其他区域相独立。
- **老生指针区**：这里包含大多数可能存在指向其他对象的指针的对象。大多数在新生区存活一段时间之后的对象都会被挪到这里。
- **老生数据区**：这里存放只包含原始数据的对象（这些对象没有指向其他对象的指针）。字符串、封箱的数字以及未封箱的双精度数字数组，在新生区存活一段时间后会被移动到这里。
- **大对象区**：这里存放体积超越其他区大小的对象。每个对象有自己`mmap`产生的内存。垃圾回收器从不移动大对象。
- **代码区**：代码对象，也就是包含JIT之后指令的对象，会被分配到这里。这是唯一拥有执行权限的内存区（不过如果代码对象因过大而放在大对象区，则该大对象所对应的内存也是可执行的。*译注：但是大对象内存区本身不是可执行的内存区*）。
- **Cell区、属性Cell区、Map区**：这些区域存放Cell、属性Cell和Map，每个区域因为都是存放相同大小的元素，因此内存结构很简单。

每个区域都由一组内存页构成。内存页是一块连续的内存，经`mmap`（或者Windows的什么等价物）由操作系统分配而来。除大对象区的内存页较大之外，每个区的内存页都是1MB大小，且按1MB内存对齐。除了存储对象，内存页还含有一个页头（包含一些元数据和标识信息）以及一个位图区（用以标记哪些对象是活跃的）。另外，每个内存页还有一个单独分配在另外内存区的槽缓冲区，里面放着一组对象，这些对象可能指向其他存储在该页的对象。这就是一套经典配置方案，其他的方案我们稍后讨论。

有了这些背景知识，我们可以来深入垃圾回收器了。

# 识别指针

垃圾回收器面临的第一个问题是，如何才能在堆中区分指针和数据，因为指针指向着活跃的对象。大多数垃圾回收算法会将对象在内存中挪动（以便减少内存碎片，使内存紧凑），因此即使不区分指针和数据，我们也常常需要对指针进行改写。

目前主要有三种方法来识别指针：

- **保守法**：这种方法对于缺少编译器支持的情况非常必要。大体上，我们将所有堆上对齐的字都认为是指针，这就意味着有些数据也会被误认为是指针。于是某些实际是数字的假指针，会被误认为指向活跃的对象，则我们会时常出现一些奇异的内存泄漏。（*译注：因为垃圾回收器会以为死对象仍然还有指针指向，错将死对象误认*

*为活跃对象*）而且我们也不能移动任何内存区域，因为这很可能会导致数据遭到破坏。这样，我们便无法通过紧凑内存来获得任何好处（比如更容易的内存分配、更少的内存访问、更有效的内存局部性缓存）。C/C++的垃圾回收器扩展会采用这种方式，比如Boehm-Demers-Weiser。
*译注：如果内存是紧凑的，则内存分配时可以更容易分配较大片的内存，而无需因内存碎片而不断查找；同时，由于已分配的内存是连续或近似连续的，而Cahce所能缓存的内存有限，如果内存被Cache缓存起来，无需频繁地迫使Cache更换缓存的内存。C/C++由于指针算术的存在，编译器无法确定哪些内存是真正的垃圾，因而无法给垃圾回收器有效的提示，进而导致垃圾回收器不得不采取这样的保守策略。*

- **编译器提示法**：如果我们和静态语言打交道，则编译器能够准确地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是哪个类实例化得到的，我们就能知道对象中所有的指针。JVM选择了这样的方法来进行垃圾回收。可惜，这种方法对于JS这样的动态语言来说不太好使，因为JS中对象的任何属性既可能是指针，也可能是数据。
- **标记指针法**：这种方法需要在每个字的末位预留一位来标记这个字代表的是指针抑或数据。这种方法需要一定的编译器支持，但实现简单，而且性能不俗。V8采用的就是这种方法。某些静态语言也采用了这样的方法，如OCaml。

V8将所有属于$-2^{30}…2^{30}-1$范围内的小整数（V8内部称其为Smis）以32bit字宽来存储，其中的最低一位保持为`0`，而指针的最低两位则为`01`。由于对象以4字节对齐，因此这样表达指针没有任何问题。大多数对象所含有的只是一组标记后的字，因此垃圾回收可以进行的很快。而有些类型的对象，比如字符串，我们确定它只含有数据，因此无需标记。

# 分代回收

脚本中，绝大多数对象的生存期很短，只有某些对象的生存期较长。为利用这一特点，V8将堆进行了分代。对象起初会被分配在新生区（通常很小，只有1-8 MB，具体根据行为来进行启发）。在新生区的内存分配非常容易：我们只需保有一个指向内存区的指针，不断根据新对象的大小对其进行递增即可。当该指针达到了新生区的末尾，就会有一次清理（小周期），清理掉新生区中不活跃的死对象。对于活跃超过2个小周期的对象，则需将其移动至老生区。老生区在*标记 - 清除*或*标记 - 紧缩*（大周期）的过程中进行回收。大周期进行的并不频繁。一次大周期通常是在移动足够多的对象至老生区后才会发生。至于足够多到底是多少，则根据老生区自身的大小和程序的动向来定。

由于清理发生的很频繁，清理必须进行的非常快速。V8中的清理过程称为Scavenge算法，是按照Cheney的算法实现的。这个算法大致是，新生区被划分为两个等大的子区：出区、入区。绝大多数内存的分配都会在出区发生（但某些特定类型的对象，如可执行的代码对象是分配在老生区的），当出区耗尽时，我们交换出区和入区（这样所有的对象都归属在入区当中），然后将入区中活跃的对象复制至出区或老生区当中。在这时我们会对活跃对象进行紧缩，以便提升Cache的内存局部性，保持内存分配的简洁快速。

以下是这个算法的伪代码描述：

```
        def scavenge():
          swap(fromSpace, toSpace)
          allocationPtr = toSpace.bottom
          scanPtr = toSpace.bottom
```

```
  for i = 0..len(roots):
    root = roots[i]
    if inFromSpace(root):
      rootCopy = copyObject(&allocationPtr, root)
      setForwardingAddress(root, rootCopy)
      roots[i] = rootCopy

  while scanPtr < allocationPtr:
    obj = object at scanPtr
    scanPtr += size(obj)
    n = sizeInWords(obj)
    for i = 0..n:
      if isPointer(obj[i]) and not inOldSpace(obj[i]):
        fromNeighbor = obj[i]
        if hasForwardingAddress(fromNeighbor):
          toNeighbor = getForwardingAddress(fromNeighbor)
        else:
          toNeighbor = copyObject(&allocationPtr, fromNeighbor)
          setForwardingAddress(fromNeighbor, toNeighbor)
        obj[i] = toNeighbor

def copyObject(*allocationPtr, object):
  copy = *allocationPtr
  *allocationPtr += size(object)
  memcpy(copy, object, size(object))
  return copy
```

在这个算法的执行过程中，我们始终维护两个出区中的指针：`allocationPtr`指向我们即将为新对象分配内存的地方，`scanPtr`指向我们即将进行活跃检查的下一个对象。`scanPtr`所指向地址之前的对象是处理过的对象，它们及其邻接都在出区，其指针都是更新过的，位于`scanPtr`和`allocationPtr`之间的对象，会被复制至出区，但这些对象内部所包含的指针如果指向入区中的对象，则这些入区中的对象不会被复制。逻辑上，你可以将`scanPtr`和`allocationPtr`之间的对象想象为一个广度优先搜索用到的对象队列。

*译注：广度优先搜索中，通常会将节点从队列头部取出并展开，将展开得到的子节点存入队列末端，周而复始进行。这一过程与更新两个指针间对象的过程相似。*

我们在算法的初始时，复制新区所有可从根对象达到的对象，之后进入一个大的循环。在循环的每一轮，我们都会从队列中删除一个对象，也就是对`scanPtr`增量，然后跟踪访问对象内部的指针。如果指针并不指向入区，则不管它，因为它必然指向老生区，而这就不是我们的目标了。而如果指针指向入区中某个对象，但我们还没有复制（未设置转发地址），则将这个对象复制至出区，即增加到我们队列的末端，同时也就是对`allocationPtr`增量。这时我们还会将一个转发地址存至出区对象的首字，替换掉Map指针。这个转发地址就是对象复制后所存放的地址。垃圾回收器可以轻易将转发地址与Map指针分清，因为Map指针经过了标记，而这个地址则未标记。如果我们发现一个指针，而其指向的对象已经复制过了（设置过转发地址），我们就把这个指针更新为转发地址，然后打上标记。

算法在所有对象都处理完毕时终止（即`scanPtr`和`allocationPtr`相遇）。这时入区的内容都可视为垃圾，可能会在未来释放或重用。

## 秘密武器：写屏障

上面有一个细节被忽略了：如果新生区中某个对象，只有一个指向它的指针，而这个指针恰好是在老生区的对象当中，我们如何才能知道新生区中那个对象是活跃的呢？显然我们并不希望将老生区再遍历一次，因为老生区中的对象很多，这样做一次消耗太大。

为了解决这个问题，实际上在写缓冲区中有一个列表，列表中记录了所有老生区对象指向新生区的情况。新对象诞生的时候，并不会有指向它的指针，而当有老生区中的对象出现指向新生区对象的指针时，我们便记录下来这样的跨区指向。由于这种记录行为总是发生在写操作时，它被称为写屏障——因为每个写操作都要经历这样一关。

你可能好奇，如果每次进行写操作都要经过写屏障，岂不是会多出大量的代码么？没错，这就是我们这种垃圾回收机制的代价之一。但情况没你想象的那么严重，写操作毕竟比读操作要少。某些垃圾回收算法（不是V8的）会采用读屏障，而这需要硬件来辅助才能保证一个较低的消耗。V8也有一些优化来降低写屏障带来的消耗：

- 大多数的脚本执行时间都是发生在Crankshaft当中的，而Crankshaft常常能静态地判断出某个对象是否处于新生区。对于指向这些对象的写操作，可以无需写屏障。
- Crankshaft中新出现了一种优化，即在对象不存在指向它的非局部引用时，该对象会被分配在栈上。而一个栈上对象的相关写操作显然无需写屏障。（*译注：新生区和老生区在堆上。*）
- “老→新”这样的情况相对较为少见，因此通过将“新→新”和“老→老”两种常见情况的代码做优化，可以相对提升多数情形下的性能。每个页都以1MB对齐，因此给定一个对象的内存地址，通过将低20bit滤除来快速定位其所在的页；而页头有相关的标识来表明其属于新生区还是老生区，因此通过判断两个对象所属的区域，也可以快速确定是否是“老→新”。
- 一旦我们找到“老→新”的指针，我们就可以将其记录在写缓冲区的末端。经过一定的时间（写缓冲区满的时候），我们将其排序，合并相同的项目，然后再除去已经不符合“老→新”这一情形的指针。（*译注：这样指针的数目就会减少，写屏障的时间相应也会缩短*）

## “标记 - 清除”算法与“标记 - 紧缩”算法

Scavenge算法对于快速回收、紧缩小片内存效果很好，但对于大片内存则消耗过大。因为Scavenge算法需要出区和入区两个区域，这对于小片内存尚可，而对于超过数MB的内存就开始变得不切实际了。老生区包含有上百MB的数据，对于这么大的区域，我们采取另外两种相互较为接近的算法：“标记 - 清除”算法与“标记 - 紧缩”算法。

这两种算法都包括两个阶段：标记阶段，清除或紧缩阶段。

在标记阶段，所有堆上的活跃对象都会被标记。每个页都会包含一个用来标记的位图，位图中的每一位对应页中的一字（*译注：一个指针就是一字大小*）。这个标记非常有必要，因为指针可能会在任何字对齐的地方出现。显然，这样的位图要占据一定的空间（32位系统上占据3.1%，64位系统上占据1.6%），但所有的内存管理机制都需要这样占用，因此这种做法并不过分。除此之外，另有2位来表示标记对象的状态。由于对象至少有2字长，因此这些位不会重叠。状态一共有三种：如果一个对象的状态为白，那么它尚未被垃圾回收器发现；如果一个对象的状态为灰，那么它已被垃圾回收器发现，但它的邻接对象仍未全部处理完毕；如果一个对象的状态为黑，则它不仅被垃圾回收器发现，而且其所有邻接对象也都处理完毕。

如果将堆中的对象看作由指针相互联系的有向图，标记算法的核心实际是深度优先搜索。在标记的初期，位图是空的，所有对象也都是白的。从根可达的对象会被染色为灰色，并被放入标记用的一个单独分配的双端队列。标记阶段的每次循环，GC会将一个对象从双端队列中取出，染色为黑，然后将它的邻接对象染色为灰，并把邻接对象放入双端队列。这一过程在双端队列为空且所有对象都变黑时结束。特别大的对象，如长数组，可能会在处理时分片，以防溢出双端队列。如果双端队列溢出了，则对象仍然会被染为灰色，但不会再被放入队列（这样他们的邻接对象就没有机会再染色了）。因此当双端队列为空时，GC仍然需要扫描一次，确保所有的灰对象都成为了黑对象。对于未被染黑的灰对象，GC会将其再次放入队列，再度处理。

以下是标记算法的伪码：

```
markingDeque = []
overflow = false

def markHeap():
  for root in roots:
    mark(root)

  do:
    if overflow:
      overflow = false
      refillMarkingDeque()

    while !markingDeque.isEmpty():
      obj = markingDeque.pop()
      setMarkBits(obj, BLACK)
      for neighbor in neighbors(obj):
        mark(neighbor)
  while overflow


def mark(obj):
  if markBits(obj) == WHITE:
    setMarkBits(obj, GREY)
    if markingDeque.isFull():
      overflow = true
    else:
      markingDeque.push(obj)

def refillMarkingDeque():
  for each obj on heap:
    if markBits(obj) == GREY:
      markingDeque.push(obj)
      if markingDeque.isFull():
        overflow = true
        return
```

标记算法结束时，所有的活跃对象都被染为了黑色，而所有的死对象则仍是白的。这一结果正是清理和紧缩两个阶段所期望的。

标记算法执行完毕后，我们可以选择清理或是紧缩，这两个算法都可以收回内存，而且两者都作用于页级（注意，V8的内存页是1MB的连续内存块，与虚拟内存页不同）。

清理算法扫描连续存放的死对象，将其变为空闲空间，并将其添加到空闲内存链表中。每一页都包含数个空闲内存链表，其分别代表小内存区（<256字）、中内存区（<2048字）、大内存区（<16384字）和超大内存区（其它更大的内存）。清理算法非常简单，只需遍历页的位图，搜索连续的白对象。空闲内存链表大量被scavenge算法用于分配存活下来的活跃对象，但也被

紧缩算法用于移动对象。有些类型的对象只能被分配在老生区，因此空闲内存链表也被它们使用。

紧缩算法会尝试将对象从碎片页（包含大量小空闲内存的页）中迁移整合在一起，来释放内存。这些对象会被迁移到另外的页上，因此也可能会新分配一些页。而迁出后的碎片页就可以返还给操作系统了。迁移整合的过程非常复杂，因此我只提及一些细节而不全面讲解。大概过程是这样的。对目标碎片页中的每个活跃对象，在空闲内存链表中分配一块其它页的区域，将该对象复制至新页，并在碎片页中的该对象上写上转发地址。迁出过程中，对象中的旧地址会被记录下来，这样在迁出结束后V8会遍历它所记录的地址，将其更新为新的地址。由于标记过程中也记录了不同页之间的指针，此时也会更新这些指针的指向。注意，如果一个页非常“活跃”，比如其中有过多需要记录的指针，则地址记录会跳过它，等到下一轮垃圾回收再进行处理。

# 增量标记与惰性清理

你应该想到了，当一个堆很大而且有很多活跃对象时，标记-清除和标记-紧缩算法会执行的很慢。起初我研究V8时，垃圾回收所引发的500-1000毫秒的停顿并不少见。这种情况显然很难接受，即使是对于移动设备。

2012年年中，Google引入了两项改进来减少垃圾回收所引起的停顿，并且效果显著：增量标记和惰性清理。

增量标记允许堆的标记发生在几次5-10毫秒（移动设备）的小停顿中。增量标记在堆的大小达到一定的阈值时启用，启用之后每当一定量的内存分配后，脚本的执行就会停顿并进行一次增量标记。就像普通的标记一样，增量标记也是一个深度优先搜索，并同样采用白灰黑机制来分类对象。

但增量标记和普通标记不同的是，对象的图谱关系可能发生变化！我们需要特别注意的是，那些从黑对象指向白对象的新指针。回忆一下，黑对象表示其已完全被垃圾回收器扫描，并不会再进行二次扫描。因此如果有“黑→白”这样的指针出现，我们就有可能将那个白对象漏掉，错当死对象处理掉。（*译注：标记过程结束后剩余的白对象都被认为是死对象。*）于是我们不得不再度启用写屏障。现在写屏障不仅记录“老→新”指针，同时还要记录“黑→白”指针。一旦发现这样的指针，黑对象会被重新染色为灰对象，重新放回到双端队列中。当算法将该对象取出时，其包含的指针会被重新扫描，这样活跃的白对象就不会漏掉。

增量标记完成后，惰性清理就开始了。所有的对象已被处理，因此非死即活，堆上多少空间可以变为空闲已经成为定局。此时我们可以不急着释放那些空间，而将清理的过程延迟一下也并无大碍。因此无需一次清理所有的页，垃圾回收器会视需要逐一进行清理，直到所有的页都清理完毕。这时增量标记又蓄势待发了。

Google近期还新增了并行清理支持。由于脚本的执行线程不会再触及死对象，页的清理任务可以放在另一个单独的线程中进行并只需极少的同步工作。同样的支持工作也正在并行标记上开展着，但目前还处于早期试验阶段。

# 总结

垃圾回收真的很复杂。我在文章中已经略过了大量的细节，而文章仍然变得很长。我一个同事说他觉得研究垃圾回收器比寄存器分配还要可怕，我表示确实如此。也就是说，我宁可将这些繁琐的细节交给运行时来处理，也不想将其交给所有的应用开发者来做。尽管垃圾回收存在一些性能问题而且偶尔会出现灵异现象，它还是将我们从大量的细节中解放了出来，以便让我们集中精力于更重要的事情上。

如果你还想了解更多垃圾回收上的东西，我建议你读读Richard Jones和Rafael Lins写的《Garbage Collection》，这是一个绝好的参考，涵盖了大量你需要了解的内容。你可能还对《Garbage First Garbage-Collection》感兴趣，这是一篇描述JVM所使用的垃圾回收算法的论文。