

## 29 | 性能：通过Orinoco、Jank Busters看垃圾回收

2022-11-24 石川 来自北京



《JavaScript进阶实战课》

[课程介绍 >](#)



讲述：石川

时长 11:32 大小 10.53M



你好，我是石川。

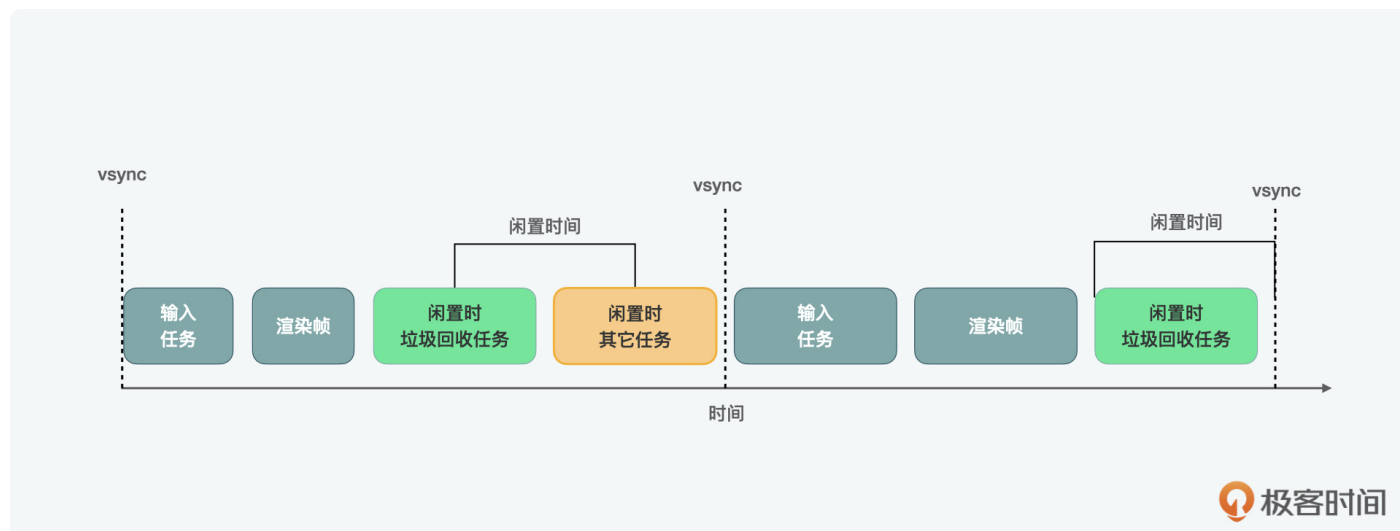
在前两讲中，我们从多线程开发的角度了解了 JavaScript 中的性能优化。

今天，我们再来看一下 JavaScript 中内存管理相关的**垃圾回收**（garbage collection）机制，以及用到的性能优化的相关算法。

实际上，在 JS 语言中，垃圾回收是自动的，也就是说并不是我们在程序开发中手工处理的，但是，了解它对理解内存管理的底层逻辑还是很有帮助的。特别是结合我们前面两节课讲到的，在前端场景中，当我们的程序使用的是图形化的 WebGL+Web Worker 的多线程来处理大量的计算或渲染工作时，了解内存管理机制则是非常必要的。特别提醒一下，这节课会涉及到比较多的理论和底层知识，一定要跟紧我们的课程节奏啊。

### 闲置状态和分代回收

我们在上一讲说到并行和并发的时候，有讲到过，前端的性能指标中，我们通常关注的是流畅度和反应度。在理想的状态下，**为了获得丝滑流畅的体验，我们需要达到 60fps，也就是每帧在 16.6ms 内渲染**。在很多的情况下，浏览器都可以在 16.6ms 内完成渲染，这个时候，如果提前渲染完了，剩下的时间我们的主线程通常是闲置（idle）的状态。而 Chrome 通常会利用这个闲置的时间来做垃圾回收。通过下面的图示，我们可以更加直观地看到主线程上这些任务的执行顺序。



在内存管理中，我们有必要先了解几个概念。在垃圾回收中，有个概念是**分代回收**（generational garbage collector），它所做的是将内存堆（memory heap）分代，不同类型的对象被分到**半空间**（semi space），里面包括了**年轻代**（young generation）和**老年代**（old generation）。

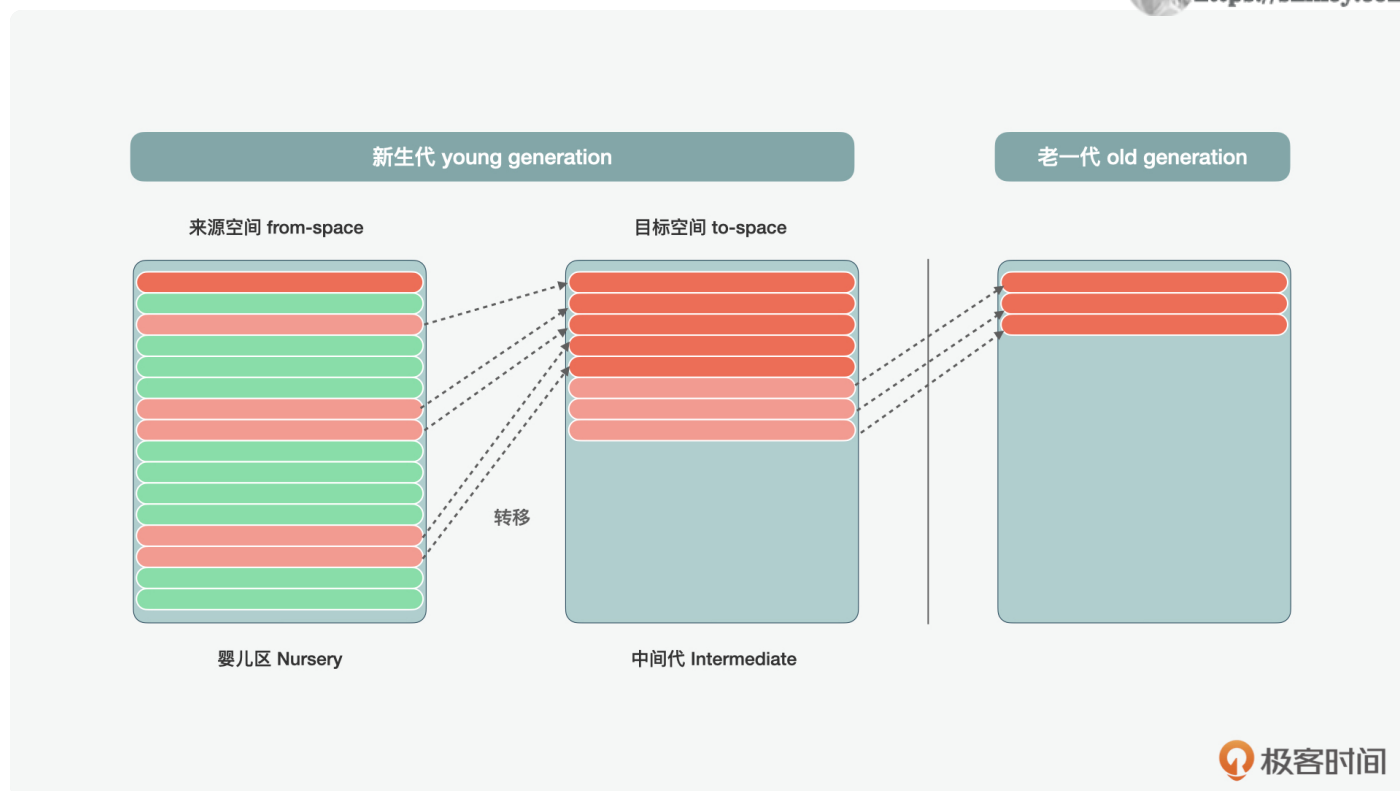
这样的分代专区，是基于垃圾回收界的一个著名的**代际假说**（generational hypothesis）来设置的。在这个假说当中，会认为年轻代中是较新的数据，这些数据中大多对象的生命周期都比较短；而那些在老年代中存活下来的数据，它们的生命周期又会特别长。

所以在 V8 中有一副一主两个垃圾回收器，分别负责年轻代和老年代的垃圾回收。

**副垃圾回收器**（minor GC, scavenger）的作用就是回收新生代中生命周期较短的对象，并且将生命周期较长的对象移动到老年代的半空间。年轻代空间里又包含**对象区域**（from-space）和**空闲区域**（to-space）。

这里你可能会想，为啥年轻代里还要再分两个区呢？因为这样可以方便对数据所进行的处理。在对象区域，数据会被标记成存活和垃圾数据。之后垃圾数据会被清除，存活数据会被晋升整

理到空闲区域。这时，空闲区域就变成了对象区域，对象区域就变成了空闲区域。也就是说在不创建新的区域的情况下，可以沿用这两个区域交换执行标记和清除的工作。



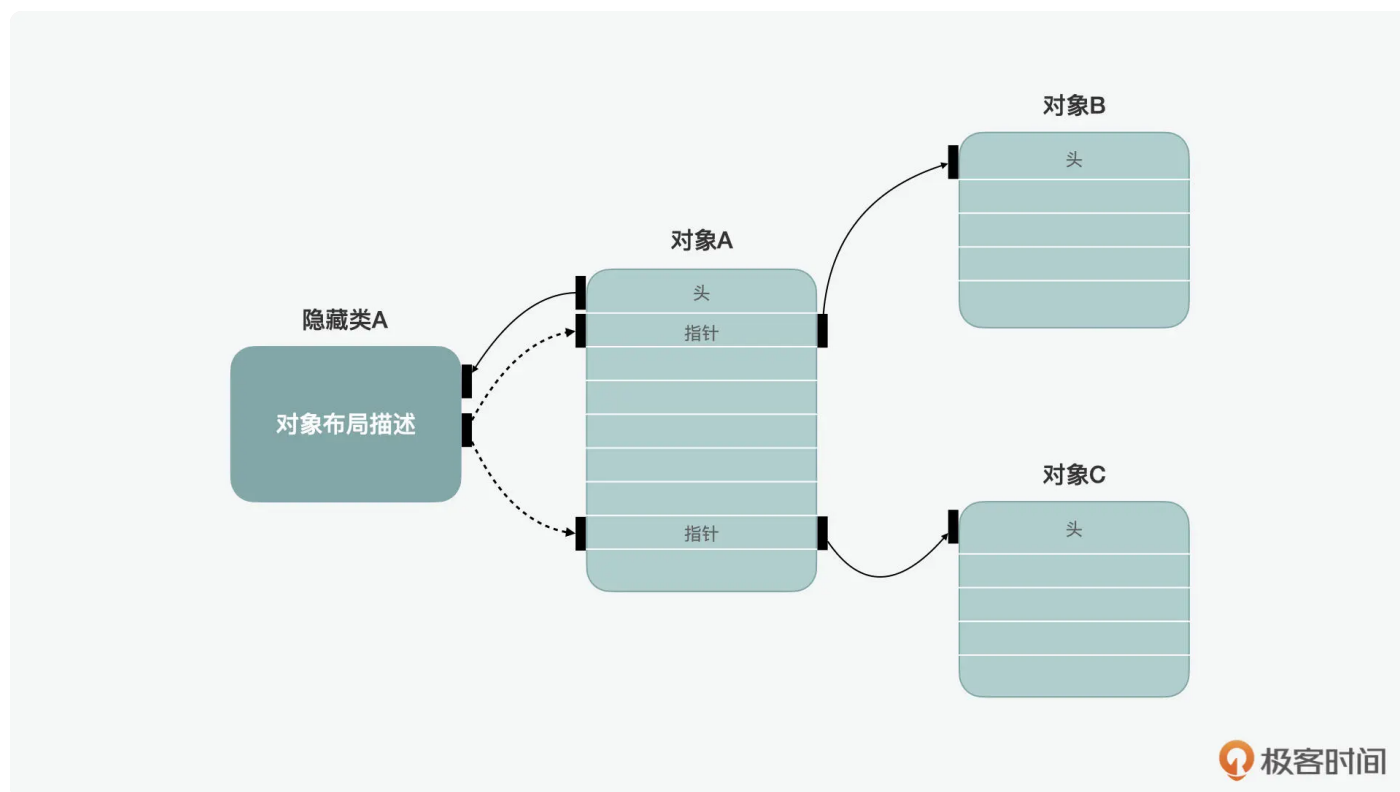
而**主垃圾回收器**（major GC）会在老年代半空间中的对象增加到一定限度的时候，对可以清除的对象做渐进的标记。通常当页面在很长一段时间空闲时，会进行全量清理，清除的动作是由专属的清除线程来完成的，最后对于碎片化的内存还要进行整理的动作。所以整体下来，主回收器的操作流程是**标记 - 清除 - 整理**（mark-sweep-compact）。

在内存管理中，特别是垃圾回收中，它的底层逻辑其实很简单，总结起来其实就 3 点：

- 如何标记存活的对象；
- 回收扫清非存活的对象；
- 回收后对碎片进行整理。主回收器也不例外。

首先，我们先来看一下**标记**（mark）。标记是找到可触达对象的过程。垃圾回收器通过可触达度来判断对象的“活跃度”。这也就代表着要保留运行时（runtime）内部当前可访问的任何对象，同时回收任何无法访问的对象。标记从一组已知的对象指针开始，如全局对象和执行堆栈中当前活动的函数，称为根集。

GC 将根（root）标记为存活的，并根据指针递归发现更多存活对象，标记为可访问。之后，堆上所有未标记的对象都被视为无法从应用程序访问的可回收对象。从数据结构和算法的角度，我们可以把标记看作是图的遍历，堆上的对象是图的节点（node）。从一个对象到另一个对象的指针是图的边（edge）。基于图中的一个节点，我们可以使用对象的隐藏类找到该节点的所有向外边缘。




标记完成后，在**清除**（sweep）的过程中，GC 会发现无法访问的对象留下的连续间隙，并将它们添加到一个被称为空闲列表（free list）的数据结构中。空闲列表由内存块的大小分隔，以便快速查找。将来，当我们想分配内存时，我们只需查看空闲列表并找到适当大小的内存块即可。

清除后的下一步就是**整理**（compact），你可以把它想象成我们平时电脑上的硬盘碎片整理，将存活的对象复制到当前未被压缩的其它内存页中（使用内存页的空闲列表）。通过这种方式，可以利用非存活的对象留下的内存中的小而分散的间隙，这样可以优化内存中的可用空间。

V8 的 Orinoco 项目是为了能不断提高垃圾回收器的性能而成立的，它的目的是通过**减少卡顿**（jank buster），提高浏览器的流畅度和响应度。在这个优化的过程中，V8 在副回收器中用到了并发和并行。下面，我们就分别来看看它们的原理及实现。

## 副回收器中使用的并行

首先，我们先来看看副回收器（minor GC，Scavenger）用到的并行回收（Scavenger Parallel）。平行回收，顾名思义，就是垃圾回收的工作是在多线程间平行完成的。相比并发，它更容易处理，因为在回收的时候，主线程上的工作是全停顿的（stop the world）。  


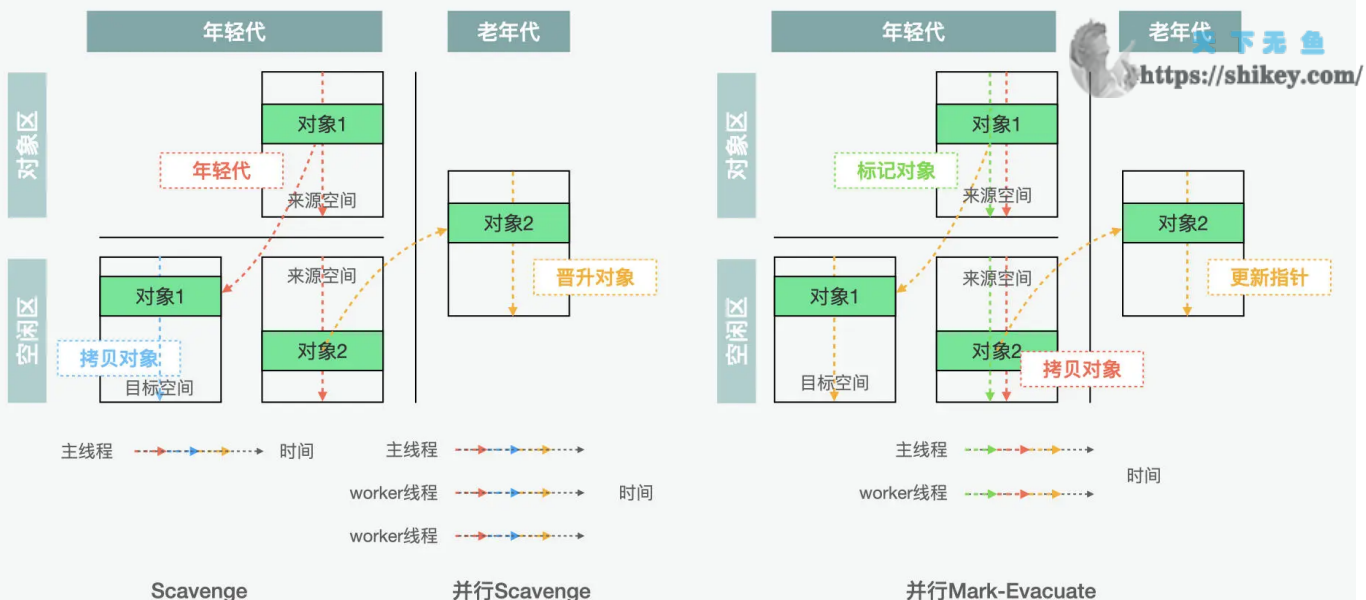
V8 用并行回收在工作线程间分配工作。每个线程会被分到一定数量的指针，线程会根据指针把存活的对象疏散到对象空间。因为不同的任务都有可能通过不同的路径找到同一个对象并且做疏散，所以这些任务是通过原子性的读写、对比和交换来操作避免竞争条件的。成功移动了对象的线程会再更新指针供其它线程参考更新。



早期，V8 所用到的是单线程的**切尼半空间复制算法**（Cheney's semispace copying algorithm）。后来，把它改成多线程。与单线程一样，这里的收集工作主要分 3 步：扫描根、在年轻代中复制、向老年代晋升以及更新指针。

这三步是交织进行的。这是一个类似于**霍尔斯特德半空间复制回收器**（Halstead's semispace copying collector）的 GC，不同之处在于，V8 使用动态工作窃取（work stealing）和相对较为简单的负载均衡机制来扫描根。





在此期间，V8 也曾尝试过一种叫做**标记转移**（Mark Evacuate algorithm）的算法。这种算法的主要优点是，可以利用 V8 中已经存在的较为完整的 Mark Sweep Compact 收集器作为基础，进行并行处理。

这里的并行处理分为三步：首先是将年轻代做标记；标记后，将存活的对象复制到对象空间；最后更新对象间的指针。

这里虽然是多线程，但它们是**锁步**（lock step）完成的，也就是说虽然这三步本身可以在不同的线程上平行执行，但线程之间必须在同步后再到下一阶段。所以，它的性能要低于前面说的**交织**完成的 Scavenger 并行算法。

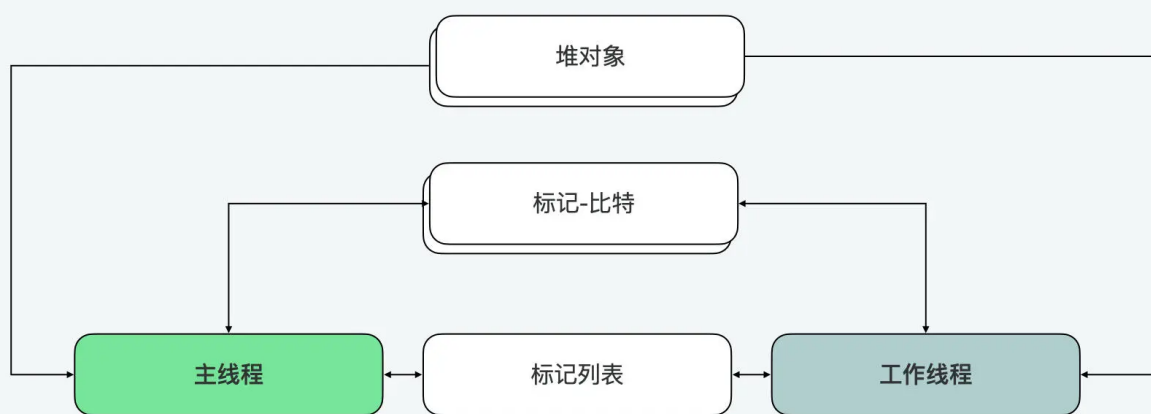
## 主回收器中使用的并发

### 并发标记

说完了副回收器中的并行 GC，我们再来看看主回收器中用到的**并发标记**（concurrent marking）。在主线程特别繁忙的情况下，标记的工作可以独立在多个工作线程上完成标记操作。但由于在此期间，主线程还在执行着程序，所以它的处理相比并行会复杂一些。

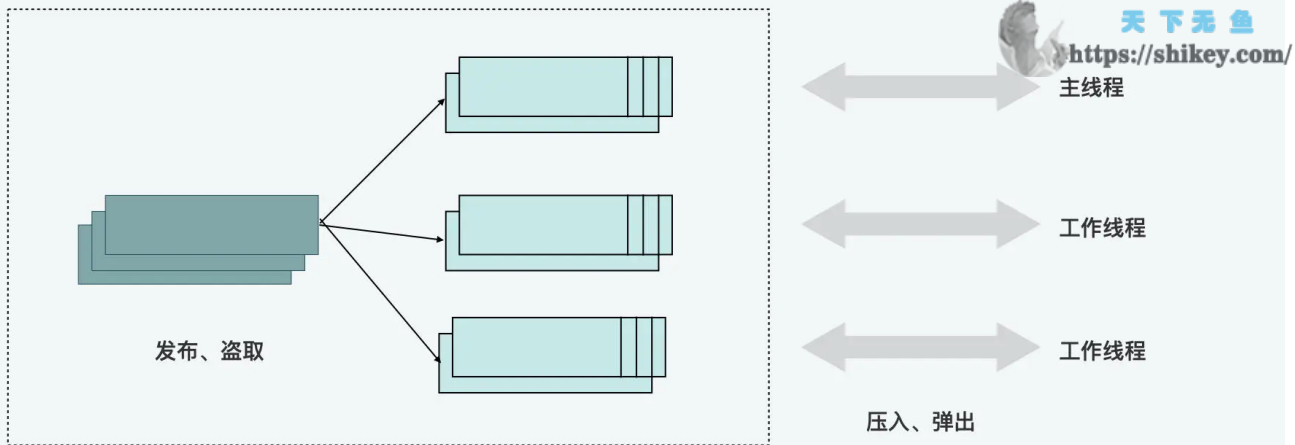


在说到并发标记前，我们先来看看标记工作怎么能在不同的线程间同时执行。在这里，对象对于不同的主线程和工作线程是只读的，而对象的标记位和标记工作列表是既支持读，也支持写访问的。



标记工作列表的实现对于性能至关重要，因为它可以平衡“完全使用线程的局部变量”或“完全使用并发”的两种极端情况。

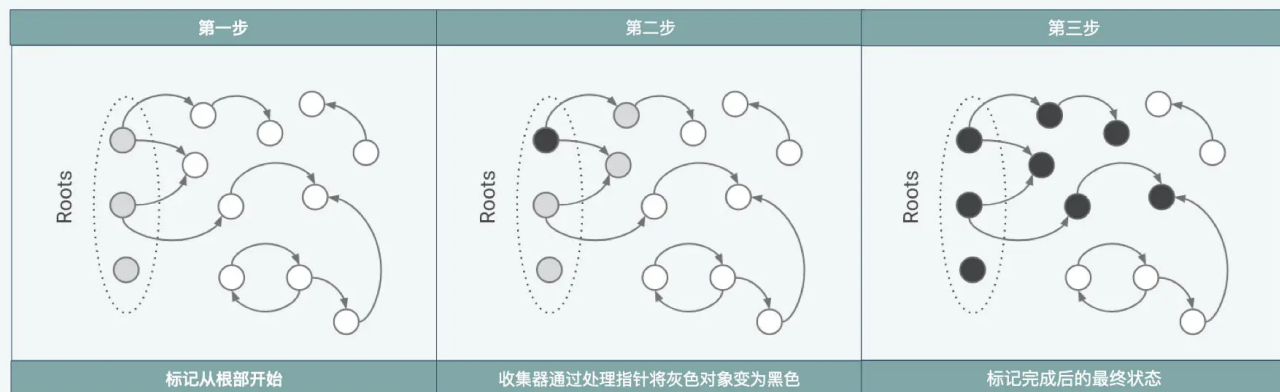
下图显示了 V8 使用基于**分段标记**的工作列表的方法，来支持线程局部变量的插入和删除，从而起到平衡这两种极端场景的作用。一旦一个分段已满，就会被发布到一个共享的全局池中，在那里它可以被线程窃取。通过这种方式，V8 允许标记工作线程尽可能长时间地在本地运行而不进行任何同步。



## 增量标记

除了并发标记法之外，主回收器还用到了**增量标记法**，也就是利用主线程空闲时间处理增量标记的任务。为了实现增量标记，要保证之前进行一半的工作有“记忆”；同时要处理期间 JavaScript 对原有对象可能造成的变化。在这里，V8 运用了三色标记法和写屏障。

三色标记法的原理是将从根部开始引用的节点标记成黑、灰和白色。黑色是引用到也标记处理的，灰色是引用到但未标记处理的，白色是未被引用到的。所以当没有灰色节点的时候，便可以清理，如果有灰色的，就要恢复标记，之后再清理。





因为增量标记是断断续续进行的，所以被标记好的数据存在可能被 JavaScript 修改的情况，比如一个被引用的数据被重新指向了新的对象，它和之前的对象就断开了，但因为垃圾回收器已经访问过旧的节点，而不会访问新的，新的节点就会因此而被记录成未被引用的白色节点。所以在这里必须做一个限制，就是不能让黑色节点指向白色的节点。而这里的限制就是通过一个写屏障来实现的。

## 总结

这节课我们通过 V8，了解了 JS 引擎是如何利用闲置状态来做垃圾回收的，以及考虑到程序性能时，这种回收机制可能带来的卡顿和性能瓶颈。我们看到 Chrome 和 V8 为了解决性能问题，通过分代和主副两个回收器做了很多的优化。这里面也用到了很多并发和并行的工作线程来提高应用执行的流畅度。


虽然对于一般的 Web 应用，这些问题并不明显。但是随着 Web3.0、元宇宙的概念的兴起，以及 WebGL+Web Worker 的并行、并发实现越来越普及，由此使得前端对象创建和渲染工作的复杂度在不断提高，所以内存管理和垃圾回收将是一个持续值得关注的话题。

## 思考题

虽然我们说 JavaScript 的垃圾回收是自动的，但是我们在写代码的时候，有没有什么“手工”优化内存的方法呢？

欢迎在留言区分享你的答案、交流学习心得或者提出问题，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。我们下节课再见！

分享给需要的人，Ta 购买本课程，你将得 18 元

 生成海报并分享

 赞 0  提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

# Vue3 企业级项目实战课

进阶高手的 Vue3+Node.js 全栈开发训练

杨文坚

前阿里前端 leader

前腾讯 IMWeb 团队高级前端工程师



新版升级：点击「👤 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

## 精选留言

写留言

由作者筛选后的优质留言将会公开显示，欢迎踊跃留言。