

赠你13张图，助你20分钟打败了「V8垃圾回收机制」！！！！



Sunshine_Lin Lv6

2021年08月13日 00:35 · 阅读 4916

+ 关注

 赠你13张图，助你20分钟打败了「V8垃圾回收机制」！！！！

前言

大家好，我是林三心。前两天，无意中看到了B站上一个讲 **v8垃圾回收机制** 的视频，感兴趣的我看了一下，感觉有点难懂，于是我就在想，大家是不是跟我一样对 **v8垃圾回收机制** 这方面的知识都比较懵，或者说看过这方面的知识，但是看不懂。所以，我思考了三天，想了一下**如何才能用最通俗的话，讲最难的知识点。**

 156

 39

 收藏

普通理解

我相信大部分同学在面试中常常被问到：“**说一说V8垃圾回收机制吧**”

这个时候，大部分同学肯定会这么回答：“垃圾回收机制有两种方式，一种是 **引用法**，一种是 **标记法**”

引用法

就是判断一个对象的引用数，引用数 **为0** 就回收，引用数 **大于0** 就不回收。请看以下代码

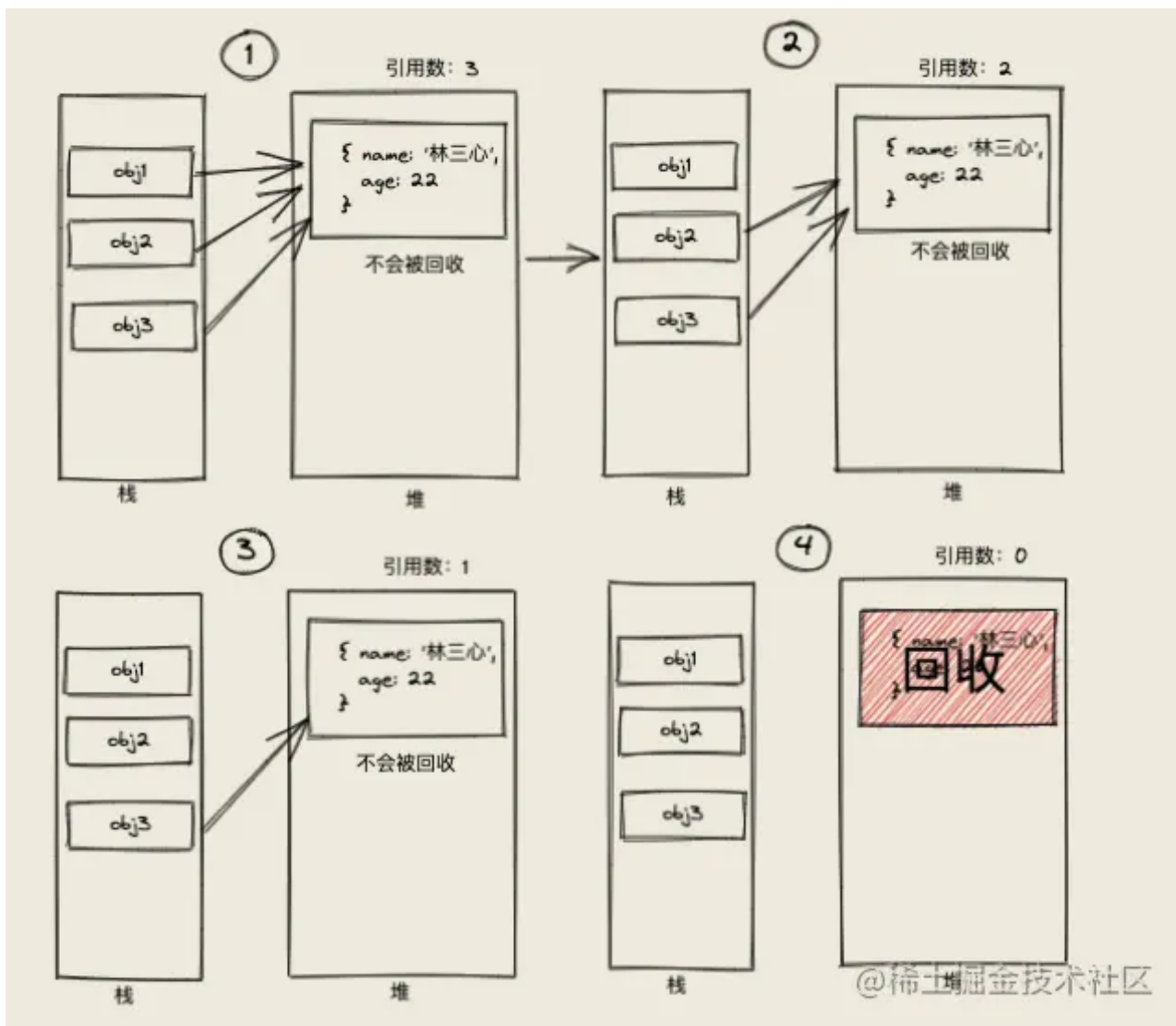
```
let obj1 = { name: '林三心', age: 22 }  
let obj2 = obj1  
let obj3 = obj1  
  
obj1 = null  
obj2 = null  
obj3 = null
```

js 复制代码

 156

 39

 收藏



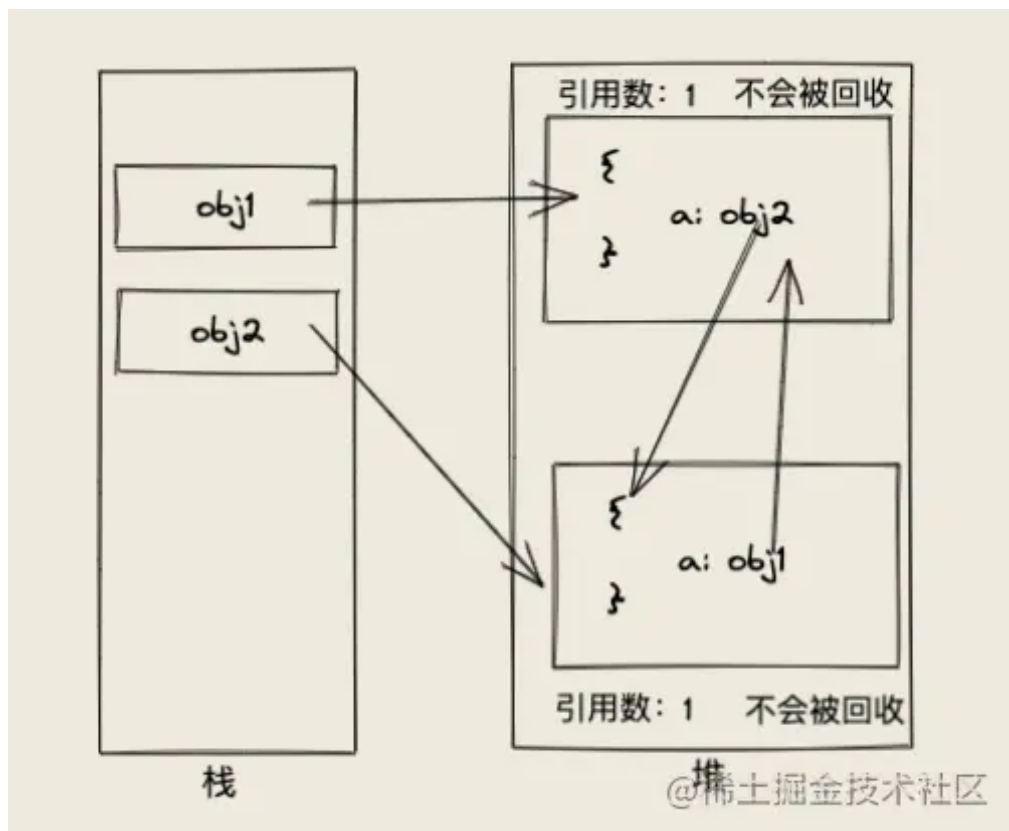
引用法是有缺点的，下面代码执行完后，按理说 `obj1`和`obj2` 都会被回收，但是由于他们互相引用，各自引用数都是1，所以不会被回收，从而造成 **内存泄漏**

👍 156

💬 39

★ 收藏

```
function fn () {  
  const obj1 = {}  
  const obj2 = {}  
  obj1.a = obj2  
  obj2.a = obj1  
}  
fn()
```



标记法

标记法就是，将 **可达** 的对象标记起来，**不可达** 的对象当成垃圾回收。

那问题来了，可不可达，通过什么来判断呢？(这里的可达，可不是可达鸭)



言归正传，想要判断可不可达，就不得不说 **可达性** 了，**可达性** 是什么？就是从初始的 **根对象** (**window或者global**) 的指针开始，向下搜索子节点，子节点被搜索到了，说明该子节点的引用对象可达，并为其进行标记，然后接着递归搜索，直到所有子节点被遍历结束。那么没有被遍历到节点，也就没有被标记，也就会被当成没有被任何地方引用，就可以证明这是一个需要被释放内存的对象，可以被垃圾回收器回收。

```
// 可达
var name = '林三心'
var obj = {
  arr: [1, 2, 3]
}
```

js 复制代码

👍 156

💬 39

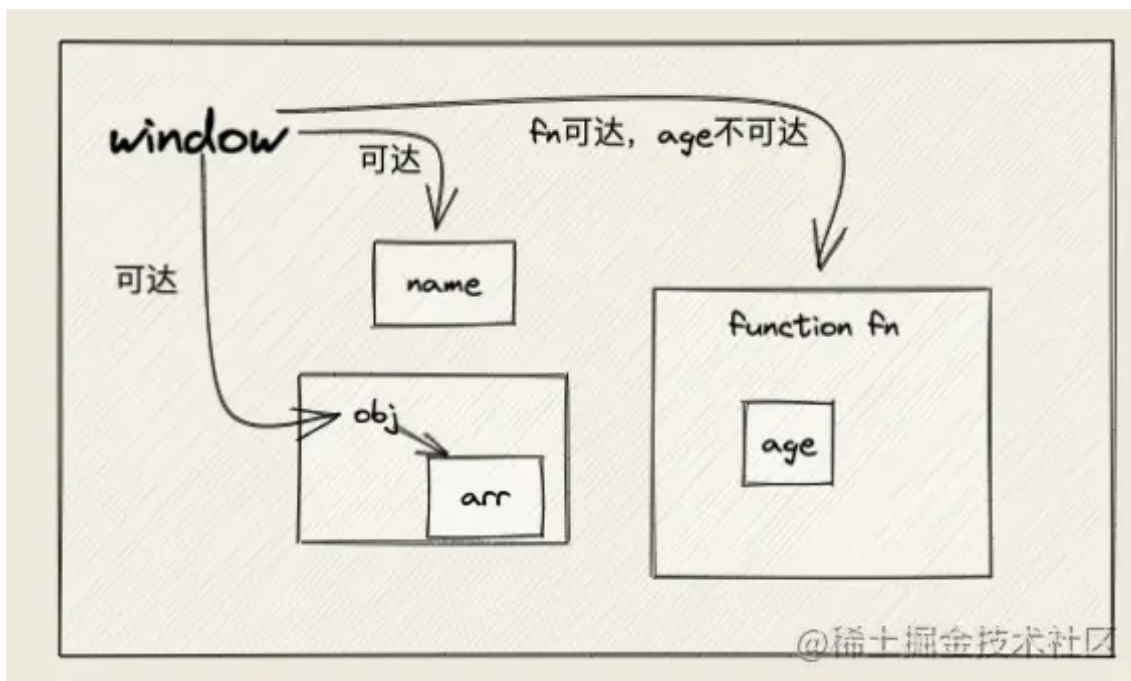
★ 收藏

```
console.log(window.obj) // { arr: [1, 2, 3] }  
console.log(window.obj.arr) // [1, 2, 3]  
console.log(window.obj.arr[1]) // 2
```

```
function fn () {  
  var age = 22  
}
```

// 不可达

```
console.log(window.age) // undefined
```



普通的理解其实是不够的，因为 [垃圾回收机制 \(GC\)](#) 其实不止这两个算法，想要更深入地了解 [v8垃圾回收机制](#)，就继续往下看吧!!!



156



39



收藏

其实JavaScript内存的流程很简单，分为3步：

- 1、分配给 **使用者** 所需的内存
- 2、**使用者** 拿到这些内存，并使用内存
- 3、**使用者** 不需要这些内存了，释放并归还给系统

那么这些 **使用者** 是谁呢？举个例子：

```
var num = ''
var str = '林三心'

var obj = { name: '林三心' }
obj = { name: '林胖子' }
```

js 复制代码

上面这些 `num`, `str`, `obj` 就是就是 **使用者**，我们都知道，JavaScript数据类型分为 **基础数据类型** 和 **引用数据类型**：

- **基础数据类型**：拥有固定的大小，值保存在 **栈内存** 里，可以通过值直接访问
- **引用数据类型**：大小不固定(可以加属性)，**栈内存** 中存着指针，指向 **堆内存** 中的对象空间，通过引用来访问



- 由于栈内存所存的基础数据类型大小是固定的，所以栈内存的内存都是 **操作系统自动分配和释放回收的**
- 由于堆内存所存大小不固定，系统 **无法自动释放回收**，所以需要 **JS引擎来手动释放这些内存**

为啥要垃圾回收

👍 156

💬 39

★ 收藏

在Chrome中，V8被限制了内存的使用（64位约1.4G/1464MB，32位约0.7G/732MB），为什么要限制呢？

- 表层原因：V8最初为浏览器而设计，不太可能遇到用大量内存的场景
- 深层原因：V8的垃圾回收机制的限制（如果清理大量的内存垃圾是很耗时间，这样会引起JavaScript线程暂停执行的时间，那么性能和应用直线下降）

前面说到栈内的内存，操作系统会自动进行内存分配和内存释放，而堆中的内存，由JS引擎（如Chrome的V8）手动进行释放，当我们的代码没有按照正确的写法时，会使得JS引擎的垃圾回收机制无法正确的对内存进行释放（内存泄露），从而使得浏览器占用的内存不断增加，进而导致JavaScript和应用、操作系统性能下降。

V8的垃圾回收算法

1. 分代回收

在JavaScript中，对象存活周期分为两种情况

- 存活周期很短：经过一次垃圾回收后，就被释放回收掉
- 存活周期很长：经过多次垃圾回收后，他还存在，赖着不走

那么问题来了，对于存活周期短的，回收掉就算了，但对于存活周期长的，多次回收都回收不掉，明知回收不掉，却还不断地去做回收无用功，那岂不是消耗性能？



对于这个问题，V8做了[分代回收](#)的优化方法，通俗点说就是：**V8将堆分为两个空间，一个叫新生代，一个叫老生代，新生代是存放存活周期短对象的地方，老生代是存放存活周期长对象的地方**



新生代通常只有 **1-8M** 的容量，而老生代的容量就大很多了。对于这两块区域，V8分别使用了[不同的垃圾回收器和不同的回收算法](#)，以便更高效地实施垃圾回收

- **副垃圾回收器 + Scavenge算法**：主要负责新生代的垃圾回收
- **主垃圾回收器 + Mark-Sweep & Mark-Compact算法**：主要负责老生代的垃圾回收

1.1 新生代

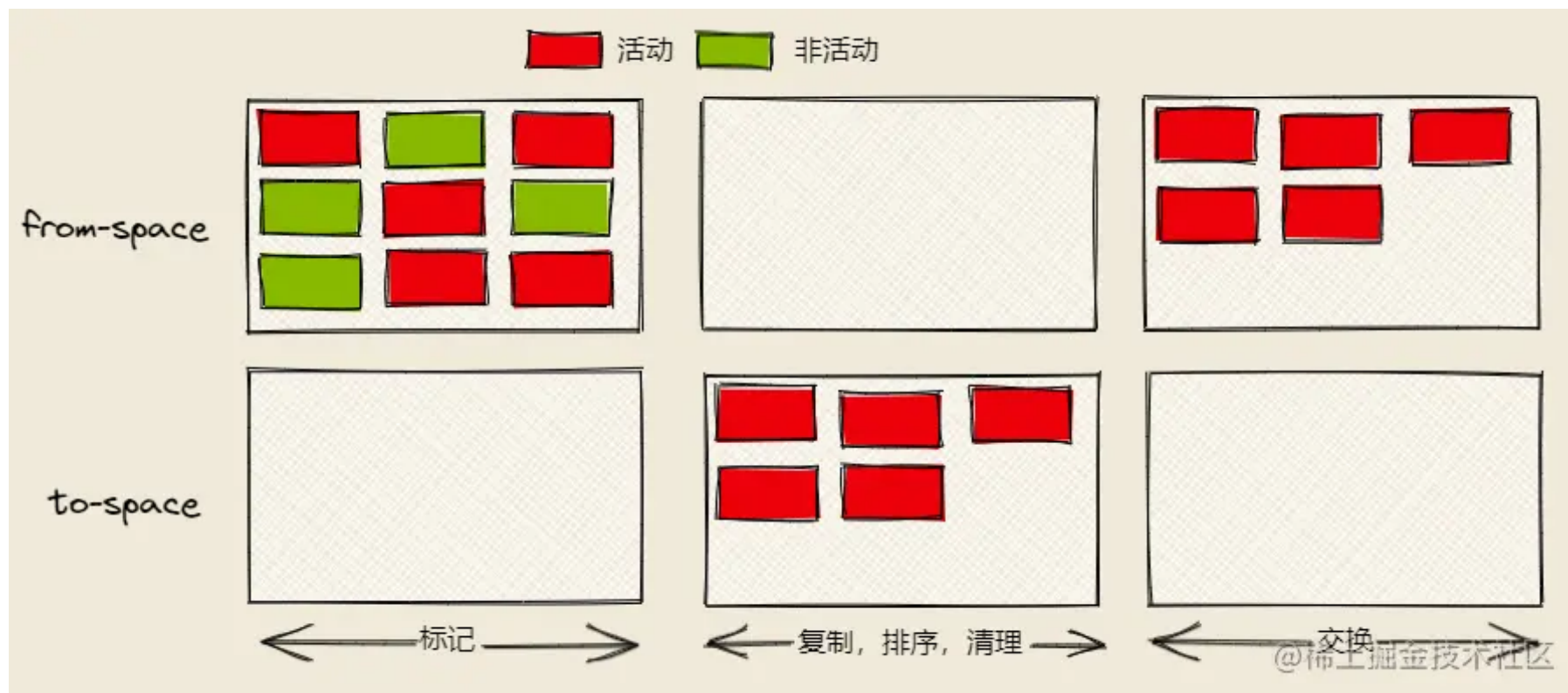
在JavaScript中，任何对象的声明分配到的内存，将会先被放置在新生代中，而因为大部分对象在内存中存活的周期很短，所以需要有一个效率非常高的算法。在新生代中，主要使用 **Scavenge** 算法进行垃圾回收，**Scavenge** 算法是一个典型的牺牲空间换取时间的复制算法，在占用空间不大的场景上非

👍 156

💬 39

★ 收藏

Scavange算法 将新生代堆分为两部分，分别叫 `from-space` 和 `to-space`，工作方式也很简单，就是将 `from-space` 中存活的活动对象复制到 `to-space` 中，并将这些对象的内存有序的排列起来，然后将 `from-space` 中的非活动对象的内存进行释放，完成之后，将 `from space` 和 `to space` 进行互换，这样可以使得新生代中的这两块区域可以重复利用。



具体步骤为以下4步：

- 1、标记活动对象和非活动对象
- 2、复制 `from-space` 的活动对象到 `to-space` 中并进行排序



156



39



收藏

- 4、将 `from-space` 和 `to-space` 进行角色互换，以便下一次的 `Scavenge` 算法 垃圾回收

那么，垃圾回收器是怎么知道哪些对象是活动对象，哪些是非活动对象呢？

这就要不得不提一个东西了——`可达性`。什么是可达性呢？就是从初始的 `根对象`（`window`或者`global`）的指针开始，向下搜索子节点，子节点被搜索到了，说明该子节点的引用对象可达，并为其进行标记，然后接着递归搜索，直到所有子节点被遍历结束。那么没有被遍历到节点，也就没有被标记，也就会被当成没有被任何地方引用，就可以证明这是一个需要被释放内存的对象，可以被垃圾回收器回收。

新生代中的对象什么时候变成老生代的对象？

在新生代中，还进一步进行了细分。分为 `nursery` 子代 和 `intermediate` 子代 两个区域，一个对象第一次分配内存时会被分配到新生代中的 `nursery` 子代，如果经过下一次垃圾回收这个对象还存在新生代中，这时候我们将此对象移动到 `intermediate` 子代，在经过下一次垃圾回收，如果这个对象还在新生代中，`副垃圾回收器` 会将该对象移动到老生代中，这个移动的过程被称为 `晋升`

1.2 老生代

新生代空间的对象，身经百战之后，留下来的老对象，成功晋升到了老生代空间里，由于这些对象都是经过多次回收过程但是没有被回收走的，都是一群生命力顽强，存活率高的对象，所以老生代里，回收算法不宜使用 `Scavenge` 算法，为啥呢，有以下原因：

- `Scavenge` 算法 是复制算法，反复复制这些存活率高的对象，没什么意义，效率极低
- `Scavenge` 算法 是以空间换时间的算法，老生代是内存很大的空间，如果使用 `Scavenge` 算法，空间资源非常浪费，得不偿失啊。。

所以老生代里使用了 `Mark-Sweep` 算法 (标记清理) 和 `Mark-Compact` 算法 (标记整理)



156



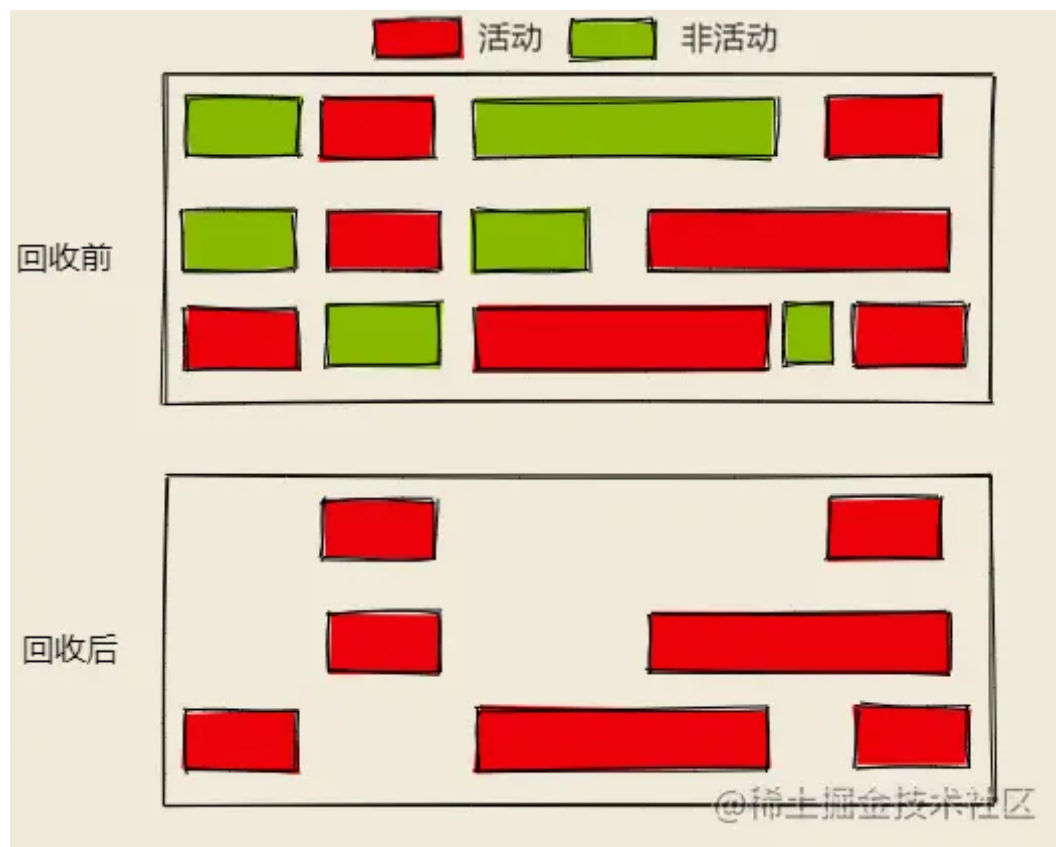
39



收藏

Mark-Sweep 分为两个阶段，标记和清理阶段，之前的 Scavenge算法 也有标记和清理，但是 Mark-Sweep算法 跟 Scavenge算法 的区别是，后者需要复制后再清理，前者不需要，Mark-Sweep 直接标记活动对象和非活动对象之后，就直接执行清理了。

- 标记阶段：对老生代对象进行第一次扫描，对活动对象进行标记
- 清理阶段：对老生代对象进行第二次扫描，清除未标记的对象，即非活动对象



由上图，我想大家也发现了，有一个问题：清除非活动对象之后，留下了很多 零零散散的空位 。



156



39



收藏

Mark-Sweep算法 执行垃圾回收之后，留下了很多 零零散散的空位 ，这有什么坏处呢？如果此时进来了一个大对象，需要对此对象分配一个大内存，先从 零零散散的空位 中找位置，找了一圈，发现没有适合自己大小的空位，只好拼在了最后，这个寻找空位的过程是耗性能的，这也是 Mark-Sweep算法 的一个 缺点

这个时候 Mark-Compact算法 出现了，他是 Mark-Sweep算法 的加强版，在 Mark-Sweep算法 的基础上，加上了 整理阶段 ，每次清理完非活动对象，就会把剩下的活动对象，整理到内存的一侧，整理完成后，直接回收掉边界上的内存



2. 全停顿(Stop-The-World)

说完V8的分代回收，咱们来聊聊一个问题。JS代码的运行要用到JS引擎，垃圾回收也要用到JS引擎，那如果这两者同时进行了，发生冲突了咋办呢？答案是， 垃圾回收优先于代码执行 ，会先停止代码的执行，等到垃圾回收完毕，再执行JS代码。这个过程，称为 全停顿

由于新生代空间小，并且存活对象少，再配合 Scavenge算法 ，停顿时间较短。但是老生代就不一样了，某些情况活动对象比较多时，停顿时间就会较长，使得页面出现了 卡顿现象 。

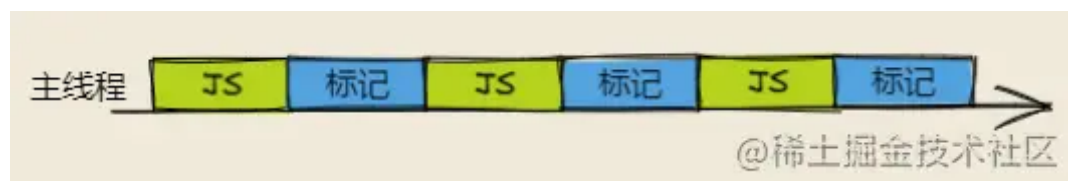
3. Orinoco优化

Orinoco为V8的垃圾回收器的项目代号，为了提升用户体验，解决 全停顿问题 ，它提出了 增量标记、惰性清理、并发、并行 的优化方法。



咱们前面不断强调了 **先标记，后清除**，而增量标记就是在 **标记** 这个阶段进行了优化。我举个生动的例子：路上有很多 **垃圾**，害得 **路人** 都走不了路，需要 **清洁工** 打扫干净才能走。前几天路上的垃圾都比较少，所以路人们都等到清洁工全部清理干净才通过，但是后几天垃圾越来越多，清洁工清理的太久了，路人就等不及了，跟清洁工说：“你打扫一段，我就走一段，这样效率高”。

大家把上面例子里，**清洁工清理垃圾的过程—标记过程**，**路人—JS代码**，一一对应就懂了。当垃圾少量时不会做增量标记优化，但是当垃圾达到一定数量时，增量标记就会开启：**标记一点，JS代码运行一段**，从而提高效率



3.2 惰性清理(Lazy sweeping)

上面说了，增量标记只是针对 **标记** 阶段，而惰性清理就是针对 **清除** 阶段了。在增量标记之后，要进行清理非活动对象的时候，垃圾回收器发现了其实就算是不清理，剩余的空间也足以让JS代码跑起来，所以就 **延迟了清理**，让JS代码先执行，或者 **只清理部分垃圾**，而不清理全部。这个优化就叫做 **惰性清理**

整理标记和惰性清理的出现，大大改善了 **全停顿** 现象。但是问题也来了：增量标记是 **标记一点，JS运行一段**，那如果你前脚刚标记一个对象为活动对象，后脚JS代码就把此对象设置为非活动对象，或者反过来，前脚没有标记一个对象为活动对象，后脚JS代码就把此对象设置为活动对象。总结起来就是：标记和代码执行的穿插，有可能造成 **对象引用改变，标记错误** 现象。这就需要使用 **写屏障** 技术来记录这些引用关系的变化



156



39



收藏

3.3 并发(Concurrent)

并发式GC允许在垃圾回收的同时不需要将主线程挂起，两者可以同时进行，只有在个别时候需要短暂停下来让垃圾回收器做一些特殊的操作。但是这种方式也要面对增量回收的问题，就是在垃圾回收过程中，由于JavaScript代码在执行，堆中的对象的引用关系随时可能会变化，所以也要进行 **写屏障** 操作。



3.4 并行

并行式GC允许主线程和辅助线程同时执行同样的GC工作，这样可以让辅助线程来分担主线程的GC工作，使得垃圾回收所耗费的时间等于总时间除以参与的线程数量（加上一些同步开销）。



@稀土掘金技术社区



V8当前的垃圾回收机制

2011年，V8应用了 **增量标记机制**。直至2018年，Chrome64和Node.js V10启动 **并发标记（Concurrent）**，同时在并发的基础上添加 **并行（Parallel）** **技术**，使得垃圾回收时间大幅度缩短。

副垃圾回收器

V8在新生代垃圾回收中，使用并行（parallel）机制，在整理排序阶段，也就是将活动对象从 **from-to** 复制到 **space-to** 的时候，启用多个辅助线程，并行的进行整理。由于多个线程竞争一个新生代的堆的内存资源，可能出现有某个活动对象被多个线程进行复制操作的问题，为了解决这个问题，V8在第一个线程对活动对象进行复制并且复制完成后，都必须去维护复制这个活动对象后的指针转发地址，以便于其他协助线程可以找到该活动对象后可以判断该活动对象是否已被复制。



@稀土掘金技术社区



156



39



收藏

V8在老生代垃圾回收中，如果堆中的内存大小超过某个阈值之后，会启用并发（Concurrent）标记任务。每个辅助线程都会去追踪每个标记到的对象的指针以及对这个对象的引用，而在JavaScript代码执行时候，并发标记也在后台的辅助进程中进行，当堆中的某个对象指针被JavaScript代码修改的时候，写入屏障（[write barriers](#)）技术会在辅助线程在进行并发标记的时候进行追踪。

当并发标记完成或者动态分配的内存到达极限的时候，主线程会执行最终的快速标记步骤，这个时候主线程会挂起，主线程会再一次的扫描根集以确保所有的对象都完成了标记，由于辅助线程已经标记过活动对象，主线程的本次扫描只是进行check操作，确认完成之后，某些辅助线程会进行清理内存操作，某些辅助进程会进行内存整理操作，由于都是并发的，并不会影响主线程JavaScript代码的执行。



结语

读懂了这篇文章，下次面试官问你的时候，你就可以不用傻乎乎地说：“引用法和标记法”。而是可以更全面地，更细致地征服面试官了。

后续会出一篇讲[项目中内存泄漏](#)的文章，敬请期待！！

👍 156

💬 39

★ 收藏

如果你觉得此文对你有一丁点帮助，点个赞，鼓励一下林三心哈哈。或者加入我的群哈哈，咱们一起摸鱼一起学习：meron857287645

分类：

前端

标签：

前端

JavaScript

文章被收录于专栏：



面试——百毒不侵

面试

关注专栏



项目优化

记录一些，我在开发中对项目进行的优化处理

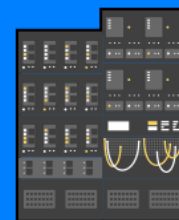
关注专栏

 156

 39

 收藏

找对——
属于你的
技术圈子



回复进群加入
掘金
微信交流群



评论

输入评论 (Enter换行, Ctrl + Enter发送)

热门评论


 156

 39

 收藏

有权威的一手资料出处吗？这类底层的东西怎么证明你说的是对的而不是个人的猜想呢？

👍 点赞 💬 8

 **Sunshine_Lin** Lv6 (作者)

6月前

参考资料

👍 点赞 💬 回复

 **FBI_Warning** Lv1 回复 **Sunshine_Lin**

6月前

但它们也是二手资料呀 🤔

“参考资料”

👍 点赞 💬 回复

查看更多回复 ▾

 **JR** Lv1 切图仔 @ 自由人

7月前

由于栈内存所存的基础数据类型大小是固定的，所以栈内存的内存都是操作系统自动分配和释放回收的
由于堆内存所存大小不固定，系统无法自动释放回收，所以需要JS引擎来手动释放这些内存

\n

这个怎么理解 为什么大小固定就系统自动回收 大小不一就不自动

👍 156

💬 39

★ 收藏



Sunshine_Lin Lv6 (作者)

7月前

栈内存存的都是基础数据，拓展性不大，但是堆内存存的都是引用数据，可以随时拓展。比如一个对象可以新增属性，一个数组可以不断push

👍 点赞 💬 回复



JR Lv1 回复 Sunshine_Lin

7月前

我知道 但是为什么固定大小的就能系统回收 大小不一的就需要V8了？

“栈内存存的都是基础数据，拓展性不大，但是堆内存存的都是引用数据，可以随时拓展。比如一个对象可以新增属性，一个数组...”

👍 点赞 💬 回复

查看更多回复 ▾

全部评论 39

🕒 最新

🔥 最热



火车头 Lv2 大前端 @ 家里蹲

8天前

写的很好，深入浅出，不过堆内存中应该还有large-object-space、map space、code space吧？

👍 点赞 💬 回复



寄何方 前后端渣渣

24天前

最后一部分的副垃圾回收器from-to和space-to是不是写错了，因为前面写的是from-space和to-space

👍 156

💬 39

★ 收藏

Running_slave Lv1

1月前

写的非常棒！学习学习了💖💖

👍 点赞 💬 回复

人der_了 Lv1

5月前

新生代Scavange算法这里我个人感觉有点问题。既然复制操作已经将活动数据复制到了to-space，为什么不直接清空from-space，而是单独去除了非活动数据。对应的图给我的感觉就是在复制操作后直接清空了from-space，文章中说的感觉没和图对上。

👍 点赞 💬 回复

FBI_Warning Lv1 前端卡卡西 @ FBI

7月前

有权威的一手资料出处吗？这类底层的東西怎么证明你说的是对的而不是个人的猜想呢？

👍 点赞 💬 8

Sunshine_Lin Lv6 (作者)

6月前

参考资料

👍 点赞 💬 回复

FBI_Warning Lv1 回复 Sunshine_Lin

6月前

但它们也是二手资料呀🤔

👍 156

💬 39

☆ 收藏

👍 点赞 💬 回复

查看更多回复 ▾



Alex59014 法师

7月前

牛逼

👍 点赞 💬 回复



JR Lv1 切图仔 @ 自由人

7月前

由于栈内存所存的基础数据类型大小是固定的，所以栈内存的内存都是操作系统自动分配和释放回收的
由于堆内存所存大小不固定，系统无法自动释放回收，所以需要JS引擎来手动释放这些内存

\n

这个怎么理解 为什么大小固定就系统自动回收 大小不一就不自动

👍 点赞 💬 6



Sunshine_Lin Lv6 (作者)

7月前

栈内存存的都是基础数据，拓展性不大，但是堆内存存的都是引用数据，可以随时拓展。比如一个对象可以新增属性，一个数组可以不断push

👍 点赞 💬 回复



JR Lv1 回复 **Sunshine_Lin**

7月前

我知道 但是为什么固定大小的就能系统回收 大小不一的就需要V8了？

👍 156

💬 39

★ 收藏

👍 点赞 💬 回复

查看更多回复 ▾

天之痕999

7月前

今天：

我：‘学会啦学会啦’

明天：

面试官：‘讲讲js垃圾回收机制’

我：‘标记法和引用法’

👍 4 💬 回复

LIMYOONA90 前端工程师

7月前

太难了！

👍 点赞 💬 回复

快秃了才变强 Lv2 前端 @ 保密

7月前

我了解的js垃圾回收机制，貌似有 标记清楚，从根节点出发，清理堆中间没打标记的，还有就是计数清除，打+1或者-1的标记，标记为0会被清除。



👍 点赞 💬 回复

ethan_Yin Lv1 切图仔 @ XXX

7月前

👍 156

💬 39

★ 收藏

👍 点赞 1



ethan_Yin Lv1

7月前

好的，看评论知道了

👍 点赞 回复



唯唯诺诺__

7月前

大哥 也没有交流群。带带

👍 点赞 1



Sunshine_Lin Lv6 (作者)

7月前

结语那里有

👍 点赞 回复



Chienlili 切图仔仔

7月前

没啥感觉. 看看就忘

👍 点赞 6



Sunshine_Lin Lv6 (作者)

7月前

这块知识确实有点枯草乏味哈哈。。。别说你，就算是我，写了这篇，过几个月就忘了

👍 156

💬 39

★ 收藏



Chienlili 回复 Sunshine_Lin

7月前

博学

“这块知识确实有点枯草乏味哈哈。。。别说你，就算是我，写了这篇，过几个月就忘了”

👍 点赞 💬 回复

查看更多回复 ▾



快跑啊小卢_ Lv4 公众号「前端快快跑」...

7月前

胖总画图用什么软件画的呀 🤔

👍 1 💬 1



Sunshine_Lin Lv6 (作者)

7月前

excalidraw

👍 点赞 💬 回复



10078

7月前

林哥yyds

👍 点赞 💬 回复



XiaoLin_Java Lv3 Java开发工程师 @ 不...

7月前

👍 156

💬 39

★ 收藏

 点赞  回复

相关推荐

摸鱼的春哥 | 2月前 | 前端 · JavaScript

2022，前端的天☁️要怎么变？

 6.8w  655  248

HighClassLickDog | 4月前 | 前端 · JavaScript

因为懒，我把公司的后管定制成了低代码中台

 6.4w  457  171

神三元 | 2年前 |

关于v8垃圾回收记住，看这篇就够啦☺

 3068  43  3

前端胖头鱼 | 5月前 | JavaScript · Vue.js · 前端

就因为JSON.stringify，我的年终奖差点打水漂了

 7.6w  1064  233

CUGGZ | 5月前 | 前端 · JavaScript · 程序员

 156

 39

 收藏

👁 7.2w 📁 1727 💬 59

Sunshine_Lin | 7月前 | 前端 · JavaScript · Vue.js

15张图，20分钟吃透Diff算法核心原理，我说的！！

👁 5.7w 📁 2231 💬 246

大帅老猿 | 10月前 | 前端 · JavaScript

产品经理：你能不能用div给我画条龙？

👁 10.8w 📁 2903 💬 576

蔓蔓雒轩 | 3年前 | JavaScript · 前端 · Promise

Promise不会？？看这里！！！史上最通俗易懂的Promise！！！

👁 7.8w 📁 1495 💬 42

浪里行舟 | 3年前 | JavaScript · 前端

九种跨域方式实现原理（完整版）

👁 13.6w 📁 2607 💬 107

大帅老猿 | 9月前 | 前端 · JavaScript · HTML

2天赚了4个W，手把手教你用Threejs搭建一个Web3D汽车展厅！

👁 5.9w 📁 1419 💬 240

前端阿飞 | 4月前 | 前端 · JavaScript

📁 156

💬 39

★ 收藏

👁 8.8w 👍 2215 💬 178

chengdwu | 7月前 | JavaScript · 前端

V8垃圾回收的工作机制

👁 265 👍 3 💬 评论

lzg9527 | 2年前 | 前端 · JavaScript

总结移动端H5开发常用技巧（干货满满哦！）

👁 5.4w 👍 1923 💬 58

Sunshine_Lin | 5月前 | 前端 · JavaScript · ECMAScript 6

「万字总结」熬夜总结50个JS的高级知识点，全都会你就是神！！

👁 7.5w 👍 2175 💬 107

非优秀程序员 | 4月前 | 前端 · JavaScript

如何用 CSS 中写出超级美丽的阴影效果

👁 30.7w 👍 233 💬 12

神三元 | 2年前 | JavaScript · 前端

(建议收藏)原生JS灵魂之问, 请问你能接得住几个? (上)

👁 14.7w 👍 3258 💬 232

寻找海蓝96 | 4年前 | JavaScript · 面试 · 前端

👍 156

💬 39

★ 收藏

👁 6.7w 👍 1675 💬 102

老腰 | 2年前 | 前端 · JavaScript · 全栈

8年前端开发的知识点沉淀(不知道会多少字，一直写下去吧...)

👁 9.3w 👍 2645 💬 183

王仕军 | 5年前 | JavaScript · 面试 · 前端

80% 应聘者都不及格的 JS 面试题

👁 7.6w 👍 1994 💬 129

大帅老猿 | 9月前 | 前端 · JavaScript

三种前端实现VR全景看房的方案！说不定哪天就用得上！

👁 6.5w 👍 2922 💬 260

👍 156

💬 39

★ 收藏