21 | 垃圾回收（二）：V8是如何优化垃圾回收器执行效率的?

2020-05-02 李兵

讲述：李兵
时长 13:55　大小 12.75M

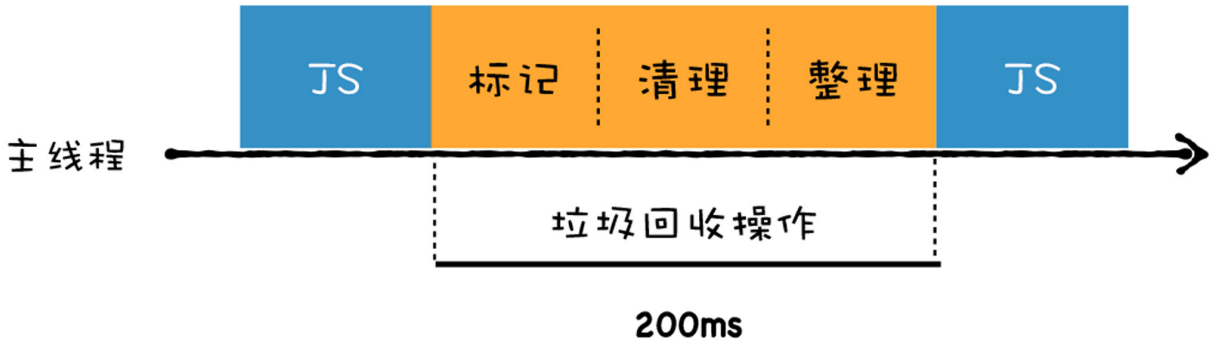
你好，我是李兵。

上节我们介绍了 V8 使用副垃圾回收器和主垃圾回收器来处理垃圾回收，这节课我们看看 V8 是如何优化垃圾回收器的执行效率的。

由于 JavaScript 是运行在主线程之上的，因此，一旦执行垃圾回收算法，都需要将正在执行的 JavaScript 脚本暂停下来，待垃圾回收完毕后再恢复脚本执行。我们把这种行为叫做**全停顿（Stop-The-World）**。

一次完整的垃圾回收分为标记和清理两个阶段，垃圾数据标记之后，V8 会继续执行清理和整理操作，虽然主垃圾回收器和副垃圾回收器的处理方式稍微有些不同，但它们都是主线程

上执行的，执行垃圾回收过程中，会暂停主线程上的其他任务，具体全停顿的执行效果如下图所示：



可以看到，执行垃圾回收时会占用主线程的时间，如果在执行垃圾回收的过程中，垃圾回收器占用主线程时间过久，就像上面图片展示的那样，花费了 200 毫秒，在这 200 毫秒内，主线程是不能做其他事情的。比如，页面正在执行一个 JavaScript 动画，因为垃圾回收器在工作，就会导致这个动画在这 200 毫秒内无法执行，造成页面的**卡顿 (Jank)**，用户体验不佳。

为了解决全停顿而造成的用户体验的问题，V8 团队经过了很多年的努力，向现有的垃圾回收器添加并行、并发和增量等垃圾回收技术，并且也已经取得了一些成效。这些技术主要是从两方面来解决垃圾回收效率问题的：

第一，**将一个完整的垃圾回收的任务拆分成多个小的任务**，这样就消灭了单个长的垃圾回收任务；

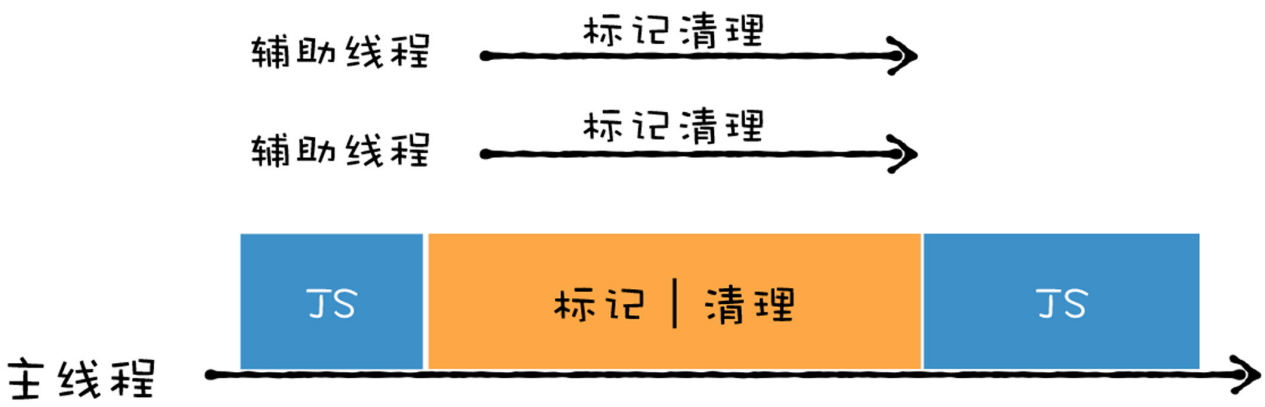
第二，**将标记对象、移动对象等任务转移到后台线程进行**，这会大大减少主线程暂停的时间，改善页面卡顿的问题，让动画、滚动和用户交互更加流畅。

接下来，我们就来深入分析下，V8 是怎么向现有的垃圾回收器添加并行、并发和增量等技术，来提升垃圾回收执行效率的。

并行回收

既然执行一次完整的垃圾回收过程比较耗时，那么解决效率问题，第一个思路就是主线程在执行垃圾回收的任务时，引入多个辅助线程来并行处理，这样就会加速垃圾回收的执行速度，因此 V8 团队引入了并行回收机制。

所谓并行回收，是指垃圾回收器在主线程上执行的过程中，还会开启多个协助线程，同时执行同样的回收工作，其工作模式如下图所示：



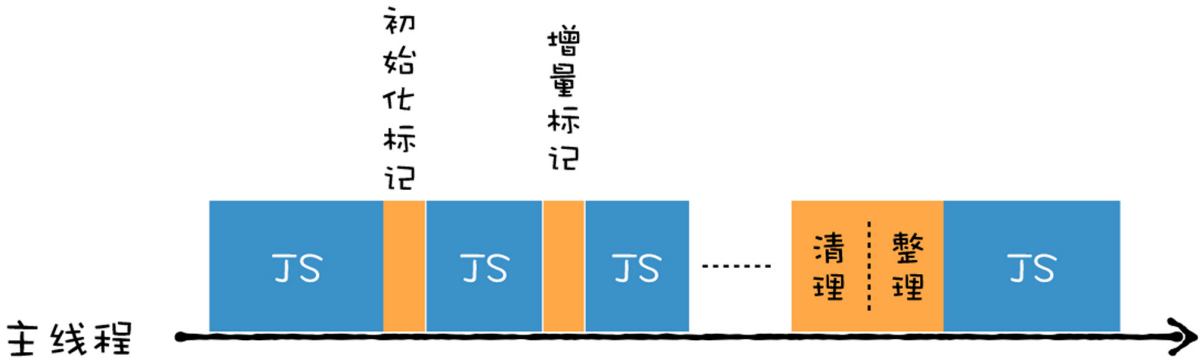
采用并行回收时，垃圾回收所消耗的时间，等于总体辅助线程所消耗的时间（辅助线程数量乘以单个线程所消耗的时间），再加上一些同步开销的时间。这种方式比较简单，因为在执行垃圾标记的过程中，主线程并不会同时执行 JavaScript 代码，因此 JavaScript 代码也不会改变回收的过程。所以我们可以假定内存状态是静态的，因此只要确保同时只有一个协助线程在访问对象就好了。

V8 的副垃圾回收器所采用的就是并行策略，它在执行垃圾回收的过程中，启动了多个线程来负责新生代中的垃圾清理操作，这些线程同时将对象空间中的数据移动到空闲区域。由于数据的地址发生了改变，所以还需要同步更新引用这些对象的指针。

增量回收

虽然并行策略能增加垃圾回收的效率，能够很好地优化副垃圾回收器，但是这**仍然是一种全停顿**的垃圾回收方式，在主线程执行回收工作的时候才会开启辅助线程，这依然还会存在效率问题。比如老生代存放的都是一些大的对象，如 window、DOM 这种，完整执行老生代的垃圾回收，时间依然会很久。这些大的对象都是主垃圾回收器的，所以在 2011 年，V8 从又引入了增量标记的方式，我们把这种垃圾回收的方式称为**谓增量式垃圾回收**。

所谓增量式垃圾回收，是指垃圾收集器将标记工作分解为更小的块，并且穿插在主线程不同的任务之间执行。采用增量垃圾回收时，垃圾回收器没有必要一次执行完整的垃圾回收过程，每次执行的只是整个垃圾回收过程中的一小部分工作，具体流程你可以参看下图：

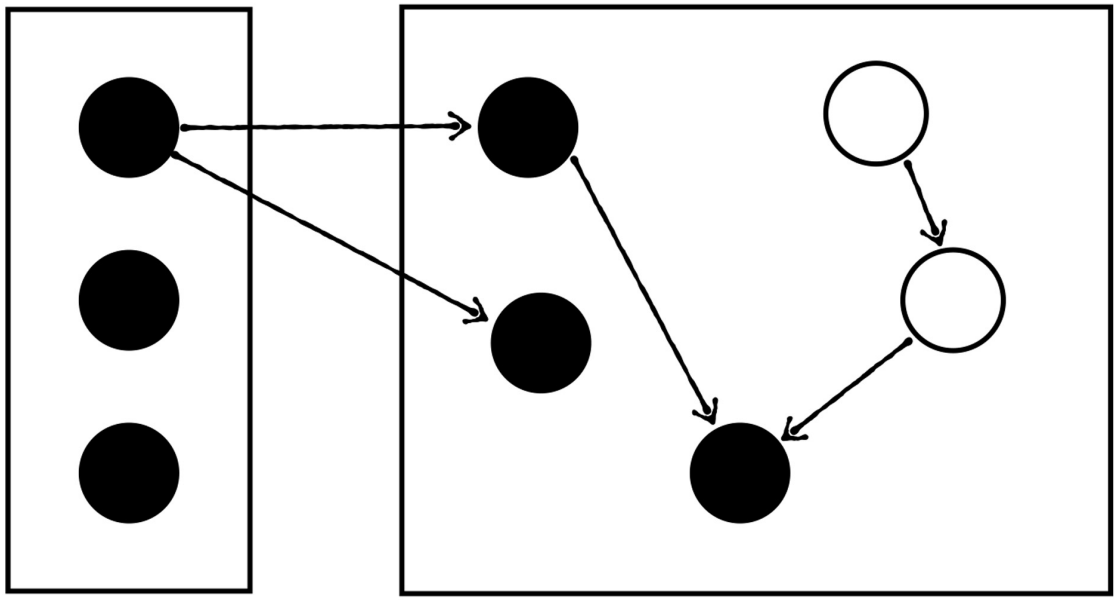


增量标记的算法，比全停顿的算法要稍微复杂，这主要是因为**增量回收是并发的（concurrent）**，要实现增量执行，需要满足两点要求：

1. 垃圾回收可以被随时暂停和重启，暂停时需要保存当时的扫描结果，等下一波垃圾回收来了之后，才能继续启动。
2. 在暂停期间，被标记好的垃圾数据如果被 JavaScript 代码修改了，那么垃圾回收器需要能够正确地处理。

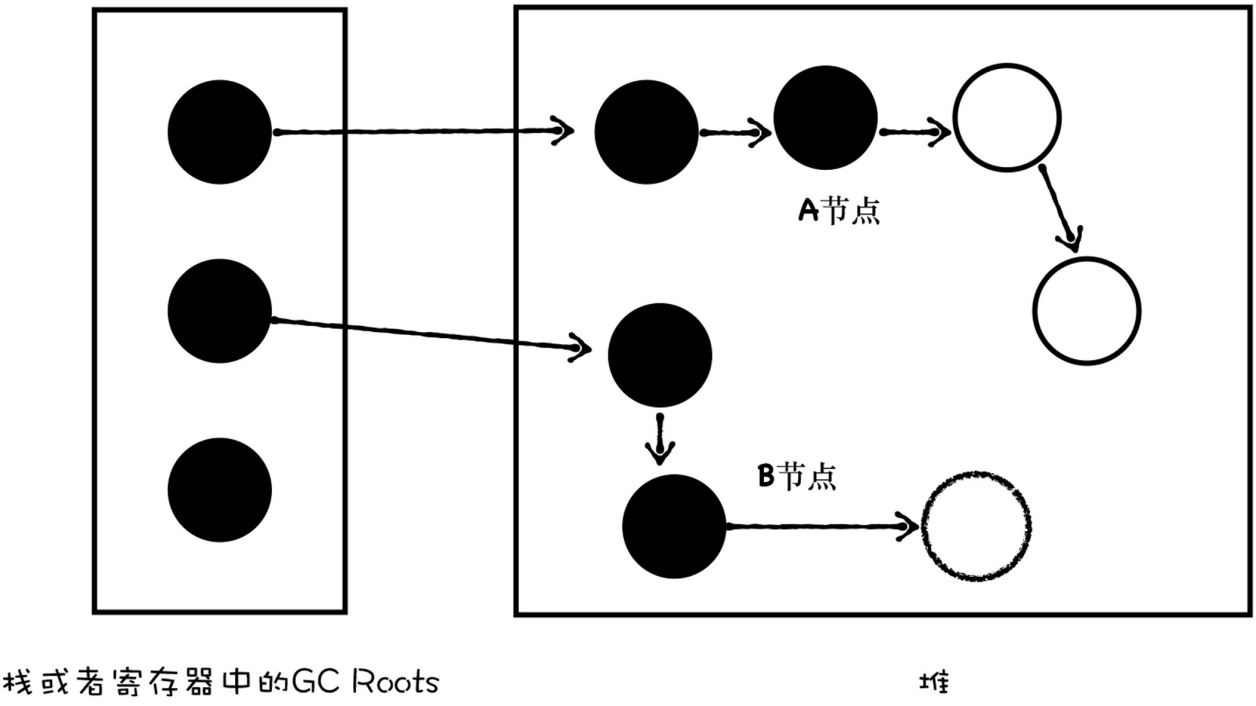
我们先来看看第一点，V8 是如何实现垃圾回收器的暂停和恢复执行的。

这里我们需要知道，在没有采用增量算法之前，V8 使用黑色和白色来标记数据。在执行一次完整的垃圾回收之前，垃圾回收器会将所有的数据设置为白色，用来表示这些数据还没有被标记，然后垃圾回收器在会从 GC Roots 出发，将所有能访问到的数据标记为黑色。遍历结束之后，被标记为黑色的数据就是活动数据，那些白色数据就是垃圾数据。如下图所示：



如果内存中的数据只有两种状态，非黑即白，那么当你暂停了当前的垃圾回收器之后，再次恢复垃圾回收器，那么垃圾回收器就不知道从哪个位置继续开始执行了。

比如垃圾回收器执行了一小段增量回收后，被 V8 暂停了，然后主线程执行了一段 JavaScript 代码，然后垃圾回收器又被恢复了，那么恢复时内存状态就如下图所示：



那么，当垃圾回收器再次被启动的时候，它到底是从 A 节点开始标记，还是从 B 节点开始执行标注过程呢？因为没有其他额外的信息，所以垃圾回收器也不知道该如何处理了。

为了解决这个问题，V8 采用了**三色标记法**，除了黑色和白色，还额外引入了灰色：

1. 黑色表示这个节点被 GC Root 引用到了，而且该节点的子节点都已经标记完成了；
2. 灰色表示这个节点被 GC Root 引用到，但子节点还没被垃圾回收器标记处理，也表明目前正在处理这个节点；
3. 白色表示这个节点没有被访问到，如果在本轮遍历结束时还是白色，那么这块数据就会被收回。

引入灰色标记之后，垃圾回收器就可以依据当前内存中有没有灰色节点，来判断整个标记是否完成，如果没有灰色节点了，就可以进行清理工作了。如果还有灰色标记，当下次恢复垃圾回收器时，便从灰色的节点开始继续执行。

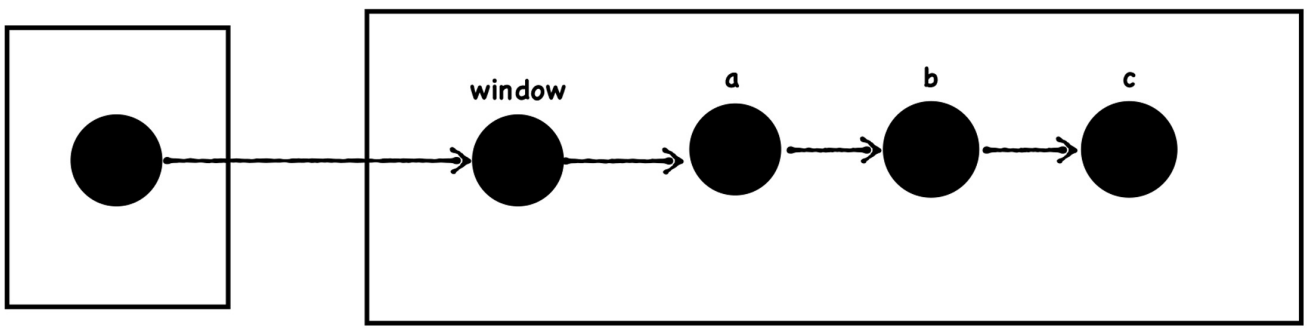
因此采用三色标记，可以很好地支持增量式垃圾回收。

接下来，我们再来分析下，标记好的垃圾数据被 JavaScript 修改了，V8 是如何处理的。我们看下面这样的一个例子：

复制代码

```
window.a = Object()
window.a.b = Object()
window.a.b.c=Object()
```

执行到这段代码时，垃圾回收器标记的结果如下图所示：

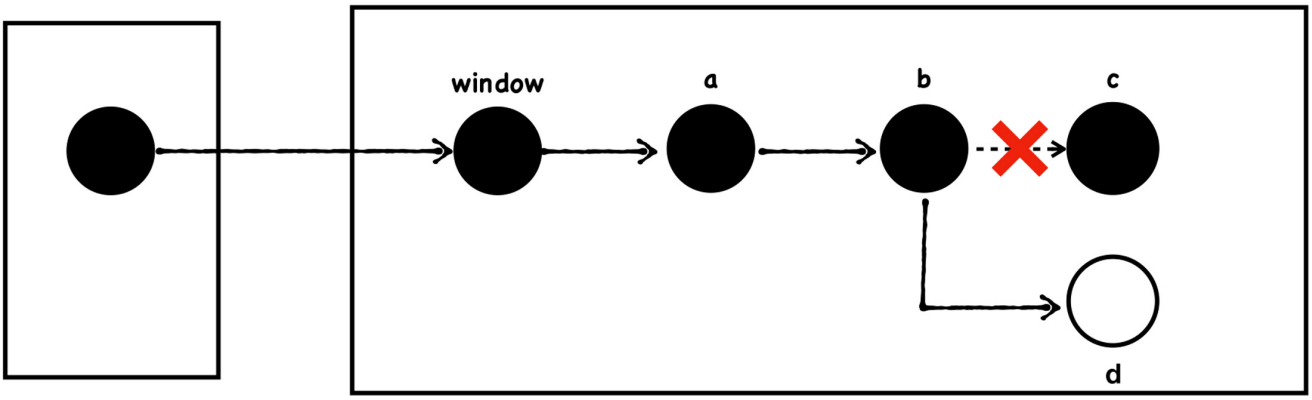


然后又执行了另外一个代码，这段代码如下所示：

```
window.a.b = Object() //d
```

复制代码

执行完之后，垃圾回收器又恢复执行了增量标记过程，由于 b 重新指向了 d 对象，所以 b 和 c 对象的连接就断开了。这时候代码的应用如下图所示：



这就说明一个问题，当垃圾回收器将某个节点标记成了黑色，然后这个黑色的节点被续上了一个白色节点，那么垃圾回收器不会再次将这个白色节点标记为黑色节点了，因为它已经走过这个路径了。

但是这个新的白色节点的确被引用了，所以我们还是需要想办法将其标记为黑色。

为了解决这个问题，增量垃圾回收器添加了一个约束条件：**不能让黑色节点指向白色节点**。

通常我们使用**写屏障 (Write-barrier) 机制**实现这个约束条件，也就是说，当发生了黑色的节点引用了白色的节点，写屏障机制会强制将被引用的白色节点变成灰色的，这样就保证了黑色节点不能指向白色节点的约束条件。这个方法也被称为**强三色不变性**，它保证了垃圾回收器能够正确地回收数据，因为在标记结束时的所有白色对象，对于垃圾回收器来说，都是不可到达的，可以安全释放。

所以在 V8 中，每次执行如 `window.a.b = value`的写操作之后，V8 会插入写屏障代码，强制将 value 这块内存标记为灰色。

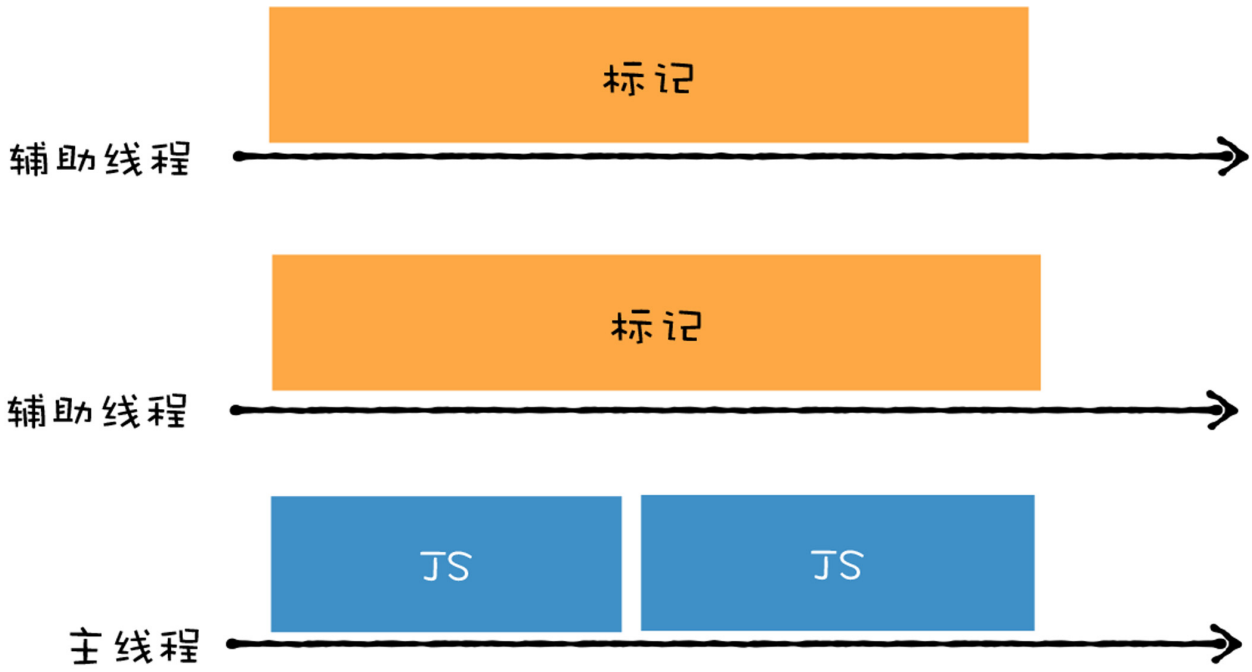
并发 (concurrent) 回收

虽然通过**三色标记法和写屏障**机制可以很好地实现增量垃圾回收，但是由于这些操作都是在主线程上执行的，如果主线程繁忙的时候，增量垃圾回收操作依然会增加降低主线程处理任务的**吞吐量 (throughput)**。

结合并行回收可以将一些任务分配给辅助线程，但是并行回收依然会阻塞主线程，那么，有没有办法在不阻塞主线程的情况下，执行垃圾回收操作呢?

还真有，这就是我们要来重点研究的**并发回收机制**了。

所谓并发回收，是指主线程在执行 JavaScript 的过程中，辅助线程能够在后台完成执行垃圾回收的操作。并发标记的流程大致如下图所示:



并发回收的优势非常明显，主线程不会被挂起，JavaScript 可以自由地执行 ，在执行的同时，辅助线程可以执行垃圾回收操作。

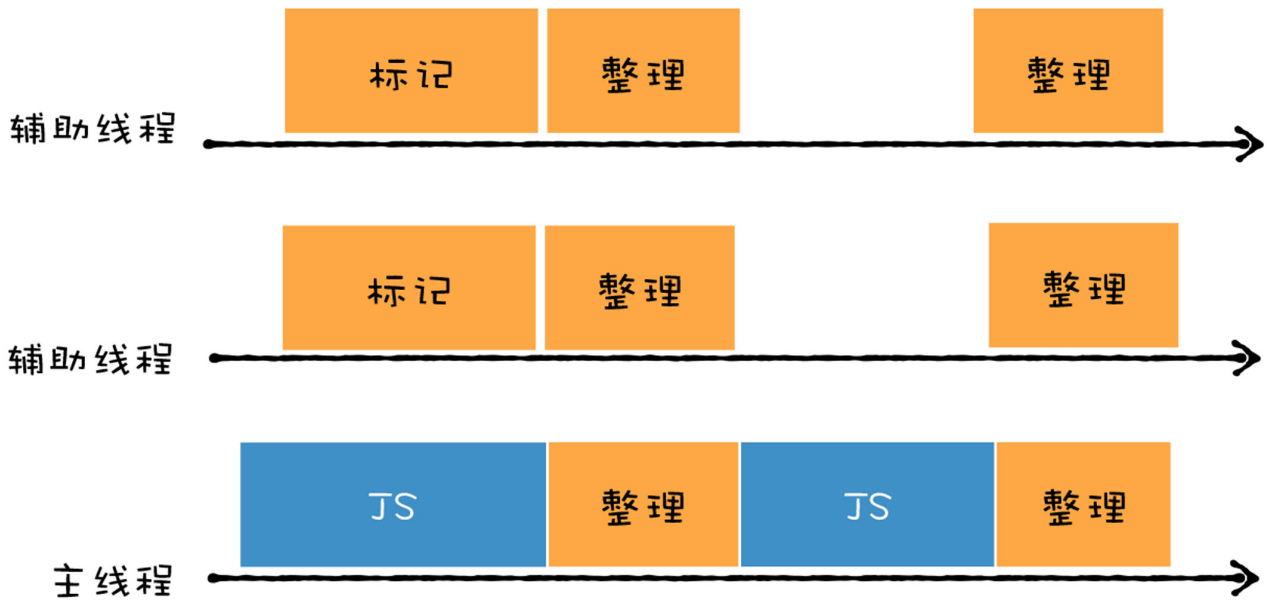
但是并发回收却是这三种技术中最难的一种，这主要由以下两个原因导致的:

第一，当主线程执行 JavaScript 时，堆中的内容随时都有可能发生变化，从而使得辅助线程之前做的工作完全无效;

第二，主线程和辅助线程极有可能在同一时间去更改同一个对象，这就需要额外实现读写锁的一些功能了。

尽管并行回收要额外解决以上两个问题，但是权衡利弊，并行回收这种方式的效率还是远高于其他方式的。

不过，这三种技术在实际使用中，并不是单独的存在，通常会将其融合在一起使用，V8 的主垃圾回收器就融合了这三种机制，来实现垃圾回收，那它具体是怎么工作的呢？你可以先看下图：



可以看出来，主垃圾回收器同时采用了这三种策略：

首先主垃圾回收器主要使用并发标记，我们可以看到，在主线程执行 JavaScript，辅助线程就开始执行标记操作了，所以说标记是在辅助线程中完成的。

标记完成之后，再执行并行清理操作。主线程在执行清理操作时，多个辅助线程也在执行清理操作。

另外，主垃圾回收器还采用了增量标记的方式，清理的任务会穿插在各种 JavaScript 任务之间执行。

总结

V8 最开始的垃圾回收器有两个特点，第一个是垃圾回收在主线程上执行，第二个特点是一次执行一个完整的垃圾回收流程。

由于这两个原因，很容易造成主线程卡顿，所以 V8 采用了很多优化执行效率的方案。

第一个方案是并行回收，在执行一个完整的垃圾回收过程中，垃圾回收器会使用多个辅助线程来并行执行垃圾回收。

第二个方案是增量式垃圾回收，垃圾回收器将标记工作分解为更小的块，并且穿插在主线程不同的任务之间执行。采用增量垃圾回收时，垃圾回收器没有必要一次执行完整的垃圾回收过程，每次执行的只是整个垃圾回收过程中的一小部分工作。

第三个方案是并发回收，回收线程在执行 JavaScript 的过程，辅助线程能够在后台完成的执行垃圾回收的操作。

主垃圾回收器就综合采用了所有的方案，副垃圾回收器也采用了部分方案。

思考题

虽然 V8 为执行垃圾回收的效率做了大量的优化，但是在实际项目中我们依然要关心内存问题，那么今天我留给你的思考题是：在使用 JavaScript 时，如何避免内存泄漏？欢迎你在留言区与我分享讨论。

感谢你的阅读，如果你觉得这一讲的内容对你有所启发，也欢迎把它分享给你的朋友。

上一篇　20 | 垃圾回收（一）：V8的两个垃圾回收器是如何工作的?

精选留言 (4)

写留言

sugar
2020-05-02

先说说思考题：内存泄漏问题的定位，一般是通过chrome的devtool中memory report来观察的，nodejs环境中的mem leak case我们研究的比较多，一般通过结合memwatch等c++扩展包把report文件dump在线上机磁盘上，然后download下来在本地的chrome浏览器devtool中进行复盘。比较常见的case是一些js工程师对scope的理解不够深，复杂的闭包里出现了隐式的引用持有却没释放。此类问题一般隐蔽性比较强，而且如果不是大...

展开

4

sugar
2020-05-02

另外还想请教一点，不知我的理解对不对。在nodejs这一端，结合今天的课程我们知道v8的gc是有线程优化的，那么是不是说 在线上服务器，我们如果给node提供2核心甚至更多

核心的运行环境，能够使gc的stw时间更短？gc效率更高？ 不知这一点我的理解对不对，如果对的话，引申出一个值得探讨的问题：我们作为线上node集群的资源利用管理角度，应该如何判定当前的node进程gc的stw时间是多长，我又该提供给它多少核心的宿主环...

展开

1

林克的小披风

2020-05-03

请教一下，写屏障机制有一点没理解。文中说，当黑节点指向白节点，会把白节点变为灰节点。但是，黑节点表示自己和子节点都已经标记为引用，灰节点表示子节点还没开始标记，那黑节点指向灰节点是矛盾的吗？

HoSalt

2020-05-03

别把对象关联到全局变量上，避免循环引用

展开