V8 垃圾回收机制

- javascript 的内存是自动分配的。
- 函数的作用域(handleScope)中包含了该函数中声明的所有变量，当该函数执行完毕后，对应的执行上下文从栈顶弹出，函数的作用域会随之销毁，其包含的所有变量也会统一释放并被自动回收。

变量不被回收，内存被占用。那么就会导致内存暴增

- 为什么要垃圾回收
- 我们知道，在 V8 引擎逐行执行 JavaScript 代码的过程中，当遇到函数的情况时，会为其创建一个函数执行上下文(Context)环境并添加到调用堆栈的栈顶，函数的作用域(handleScope)中包含了该函数中声明的所有变量，当该函数执行完毕后，对应的执行上下文从栈顶弹出，函数的作用域会随之销毁，其包含的所有变量也会统一释放并被自动回收。试想如果在这个作用域被销毁的过程中，其中的变量不被回收，即持久占用内存，那么必然会导致内存暴增，从而引发内存泄漏导致程序的性能直线下降甚至崩溃，因此内存在使用完毕之后理当归还给操作系统以保证内存的重复利用。
- JS 单线程机制：作为浏览器的脚本语言，JS 的主要用途是与用户交互以及操作 DOM，那么这也决定了其作为单线程的本质，单线程意味着执行的代码必须按顺序执行，在同一时间只能处理一个任务。试想如果 JS 是多线程的，一个线程在删除 DOM 元素的同时，另一个线程对该元素进行修改操作，那么必然会导致复杂的同步问题。既然 JS 是单线程的，那么也就意味着在 V8 执行垃圾回收时，程序中的其他各种逻辑都要进入暂停等待阶段，直到垃圾回收结束后才会再次重新执行 JS 逻辑。因此，由于 JS 的单线程机制，垃圾回收的过程阻碍了主线程逻辑的执行。
- V8 引擎为了减少对应用的性能造成的影响，采用了一种比较粗暴的手段，那就是直接限制堆内存的大小，毕竟在浏览器端一般也不会遇到需要操作几个 G 内存这样的场景。但是在 node 端，涉及到的 I/O 操作可能会比浏览器端更加复杂多样，因此更有可能出现内存溢出的情况。不过也没关系，V8 为我们提供了可配置项来让我们手动地调整内存大小，但是需要在 node 初始化的时候进行配置，我们可以通过如下方式来手动设置。

引用计数的缺陷。 但是因为采用了引用计数的算法，两个变量均存在指向自身的引用，因此依旧无法被回收，导致内存泄漏。

新生代（new_space） 老生代(old_space) 大对象区(large_object_space) 代码区(code_space) map 区(map_space)

新生代：Scavenge

老生代：Mark-Sweep & Mark-Compact

Mark-Sweep(标记清除) 分为了两个阶段， 标记 和 清楚 两个阶段。
https://juejin.cn/post/6844904016325902344

Mark-Compact(标记整理) 对不连续的碎片进行整理。移动对象在堆中的位置。

得益于增量标记的好处，V8 引擎后续继续引入了延迟清理(lazy sweeping)和增量式整理(incremental compaction)，让清理和整理的过程也变成增量式的。同时为了充分利用多核 CPU 的性能，也将引入并行标记和并行清理，进一步地减少垃圾回收对主线程的影响，为应用提升更多的性能。

标记阶段：通过根值（根节点），标记出从根值开始的所有可达值； 清除阶段：未被标记的则为垃圾对象，在清除阶段会被清除。

如何避免内存泄漏

1. 尽可能少创建全局变量
2. setTimeout 或者 setInterval（定时器造成的内存泄漏）
3. 正确地使用闭包
4. 清理 DOM 元素

什么是 javascript 的弱引用。 弱引用只是在 GC 中的概念。说的

根节点到不了的就视为非活动，就视为垃圾。。

因为所清除的对象地址可能不连续。所以会引入标记整理的概念

在堆中移动对象的位置。移动完之后再回收对应位置的内容

JavaScript 的垃圾回收机制熟悉吗？讲讲 可以。

浏览器的垃圾回收主要分为两种， 标记清除法 & 引用计数法。 引用计数法 会根据对象的引用进行计数。引用计数的含义是跟踪记录每个值被引用的次数。当主线程任务执行完后，会查看当前引用次数为 0 的对象。对齐内存空间进行回收。但是这种方式在两个对象循环因引用时会导致内存释放不了。 所以我们的 V8 引擎采取的是标记清除法。

V8 引擎采取的是分代式垃圾回收机制。 V8 的内存被开辟成五个区域 其中包括

1. 新生代区(new_space)
2. 老生代区(old_space)
3. 大对象区(large_object_space)
4. 代码区(code_space)
5. map 区(map_space)

新生代区。scavenge 算法

step1

- 那整体流程时这样的。
- 新生代区会等分化成 2 个区块。
- 假设命名两个区块为 Form 和 to
- 刚创建的对象都会放到 Form 块
- 当主线程代码执行完后，会马上执行一次 GC
- 会全量的把还有在引用的对象放到我们的 to 当中
- 剩下在 Form 的对象未被引用，则清除，被回收。
- 然后次数 to 的角色与 Form 的角色 将会在下一次垃圾回收时互换。

新生代晋升

step2

- 新生代的对象会在一定时间后会晋升到我们的老生代区中。
- 新生代晋升必须要

- 满足以下任何一个条件之一
    1. 最少经历一次 scavenge
    2. scavenge 后 To 区域内存大于 25%时。

进行老生代区后

step3

- 进行老生代区后，使用的是标记清除法进行 GC

    1. 垃圾回收器会以 window 为根节点，从全局触发去寻找可被访问到的变量，如被访问到则视为活动的，
    2. 未被访问到的则会视为垃圾，进行垃圾回收，内存释放。

- 标记清楚会导致内存空间存在不连续的状态，因为我们清除的对象占用的内存地址可能是不连续的。所以为了解决不连续的问题，就有了标记整理，讲活动的对象往堆的一段进行移动，完成后再释放掉左/右 边界的内存。 所以标记清除法的整体流程就是 标记 - 整理 - 清除