

05_coderwhy前端八股文（五） - V8引擎和内存管理

01-V8引擎的执行原理

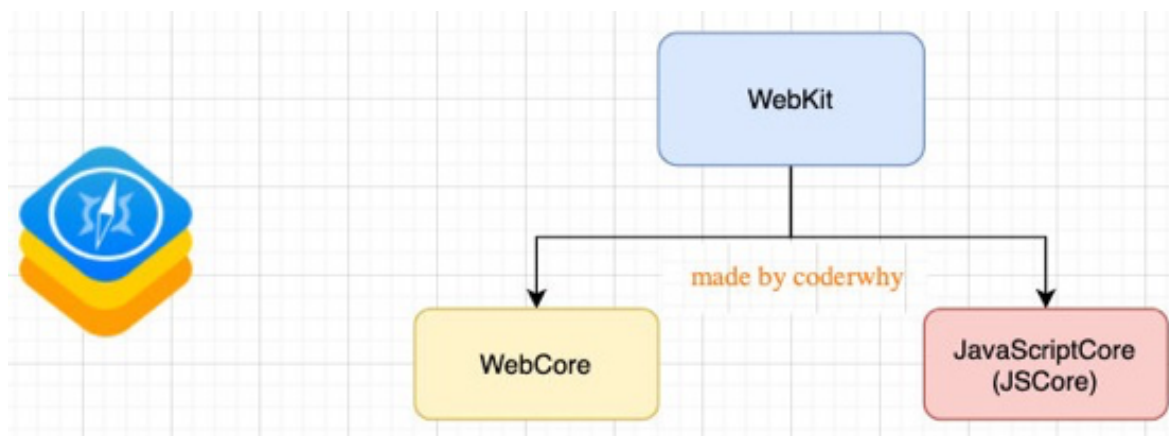
JavaScript引擎

JavaScript代码下载好之后，是如何一步步被执行的呢？（什么是JavaScript引擎？）

我们知道，浏览器内核主要是由两部分组成的，以webkit为例：

- WebCore：负责HTML解析、布局、渲染等相关的工作；
- JavaScriptCore：解析、执行JavaScript代码；

Webkit源码地址：<https://github.com/WebKit/WebKit>



JavaScript引擎非常多，我们来介绍几个比较重要的（面试题：常见的JavaScript引擎有哪些？）：

- SpiderMonkey：第一款JavaScript引擎，由Brendan Eich开发（也就是JavaScript作者），在1996年发布；
- Chakra：Chakra 最初是 Internet Explorer 9 的 JavaScript 引擎，并在后续成为了 Edge 浏览器的引擎，直到 Microsoft 转向 Chromium 架构并采用了 V8。
- JavaScriptCore：JavaScriptCore 是 WebKit 浏览器引擎的一部分，主要用于 Apple 的 Safari 浏览器，它也被用在所有 iOS 设备的应用中。
- V8引擎：V8 是 Chrome 浏览器和 Node.js 的 JavaScript 引擎，也是我们后续讲解的重点。

接下来我们以V8引擎为例，来讲解一下JavaScript代码具体的执行过程。

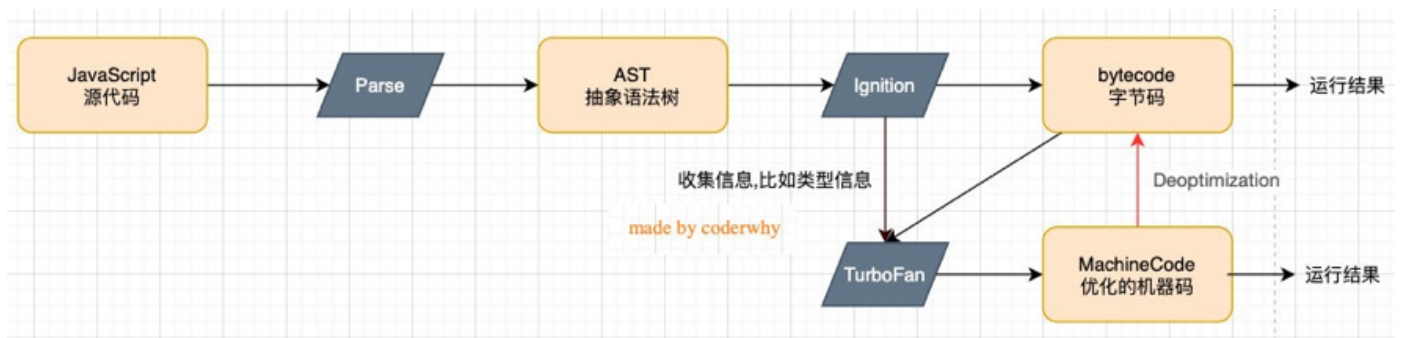
V8引擎的执行

我们来看一下官方对V8引擎的定义：

- V8是用C++编写的Google开源高性能JavaScript和WebAssembly引擎，它用于Chrome和Node.js等。
- 它实现ECMAScript和WebAssembly，并在Windows 7或更高版本，macOS 10.12+和使用x64，IA-32，ARM或MIPS处理器的Linux系统上运行。

- V8可以独立运行，也可以嵌入到任何C++应用程序中。

那么当我们有一段JavaScript代码时，V8引擎是如何执行它们的呢？



过程的解析：（面试题JavaScript代码是如何被执行的？V8引擎如何执行JavaScript代码？）

解析 (Parse)：JavaScript 代码首先被解析器处理，转化为抽象语法树（AST）。这是代码编译的初步阶段，主要转换代码结构为内部可进一步处理的格式。

AST：抽象语法树（AST）是源代码的树形表示，用于表示程序结构。之后，AST 会被进一步编译成字节码。

Ignition：Ignition 是 V8 的解释器，它将 AST 转换为字节码。字节码是一种低级的、比机器码更抽象的代码，它可以快速执行，但比直接的机器码慢。

字节码 (Bytecode)：字节码是介于源代码和机器码之间的中间表示，它为后续的优化和执行提供了一种更标准化的形式。字节码是由 Ignition 生成，可被直接解释执行，同时也是优化编译器 TurboFan 的输入。

TurboFan：TurboFan 是 V8 的优化编译器，它接收从 Ignition 生成的字节码并进行进一步优化。比如如果一个函数被多次调用，那么就会被标记为热点函数，那么就会经过TurboFan转换成优化的机器码，提高代码的执行性能。当然还会包括很多其他的优化手段，如内联替换（Inlining）、死代码消除（Dead Code Elimination）和循环展开（Loop Unrolling）等，以提高代码执行效率。

机器码：经过 TurboFan 处理后，字节码被编译成机器码，即直接运行在计算机硬件上的低级代码。这一步是将 JavaScript 代码转换成 CPU 可直接执行的指令，大大提高了执行速度。

运行时优化：在代码执行过程中，V8 引擎会持续监控代码的执行情况。如果发现之前做的优化不再有效或者有更优的执行路径，它会触发去优化（Deoptimization）。去优化是指将已优化的代码退回到优化较少的状态，然后重新编译以适应新的运行情况。

V8三大模块的核心分析（GC垃圾回收后续讲解）（面试题：V8引擎包括哪些部分，它们的作用是什么？）

V8引擎本身的源码非常复杂，大概有超过100w行C++代码，通过了解它的架构，我们可以知道它是如何对JavaScript执行的：

Parse模块会将JavaScript代码转换成AST（抽象语法树），这是因为解释器并不直接认识JavaScript代码；

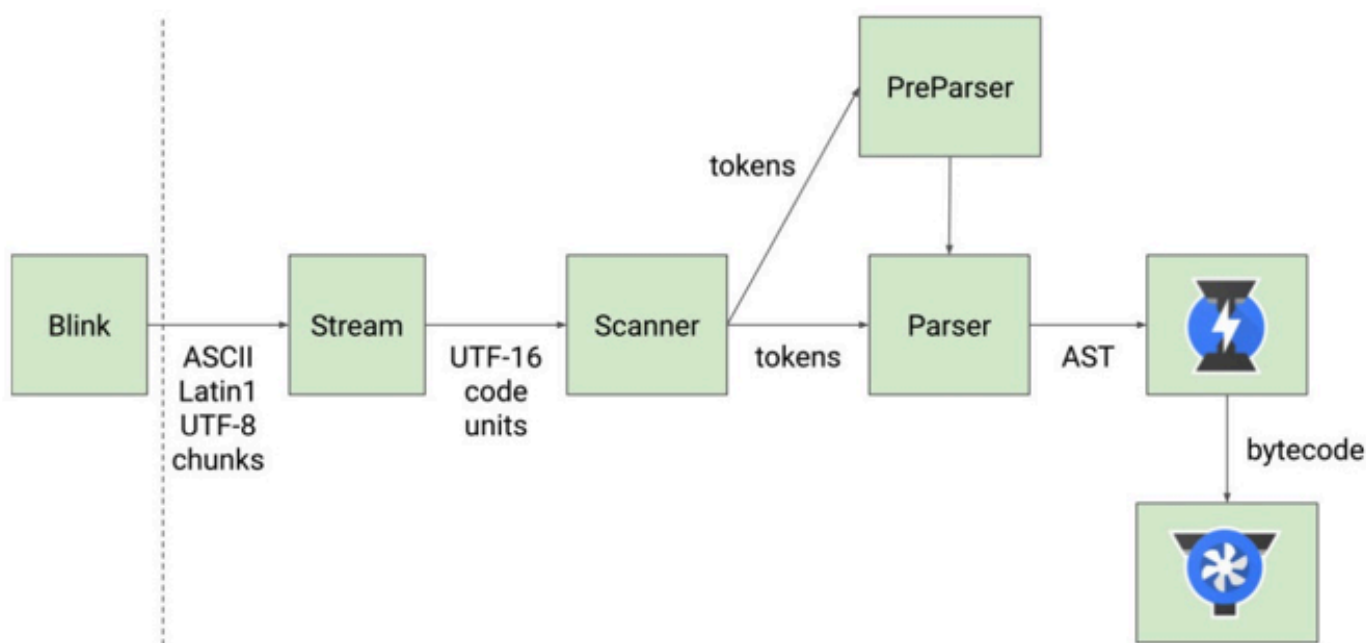
- 将代码转化成AST树是一个非常常见的操作，比如在Babel、Vue源码中都需要进行这样的操作；
- Parse的V8官方文档：<https://v8.dev/blog/scanner>

Ignition是一个解释器，会将AST转换成ByteCode（字节码）

- 同时会收集TurboFan优化所需要的信息（比如函数参数的类型信息，有了类型才能进行真实的运算）；
- 如果函数只调用一次，Ignition会解释执行ByteCode；
- Ignition的V8官方文档：<https://v8.dev/blog/ignition-interpret>

TurboFan是一个编译器，可以将字节码编译为CPU可以直接执行的机器码；

- 如果一个函数被多次调用，那么就会被标记为热点函数，那么就会经过TurboFan转换成优化的机器码，提高代码的执行性能；
- 但是，机器码实际上也会被还原为ByteCode（Deoptimization 去优化），这是因为如果后续执行函数的过程中，类型发生了变化（比如sum函数原来执行的是number类型，后来执行变成了string类型），之前优化的机器码并不能正确的处理运算，就会逆向的转换成字节码；
- TurboFan的V8官方文档：<https://v8.dev/blog/turbofan-jit>



词法分析（英文lexical analysis）

- 将字符序列转换成token序列的过程。
- token是记号化（tokenization）的缩写
- 词法分析器（lexical analyzer，简称lexer），也叫扫描器（scanner）

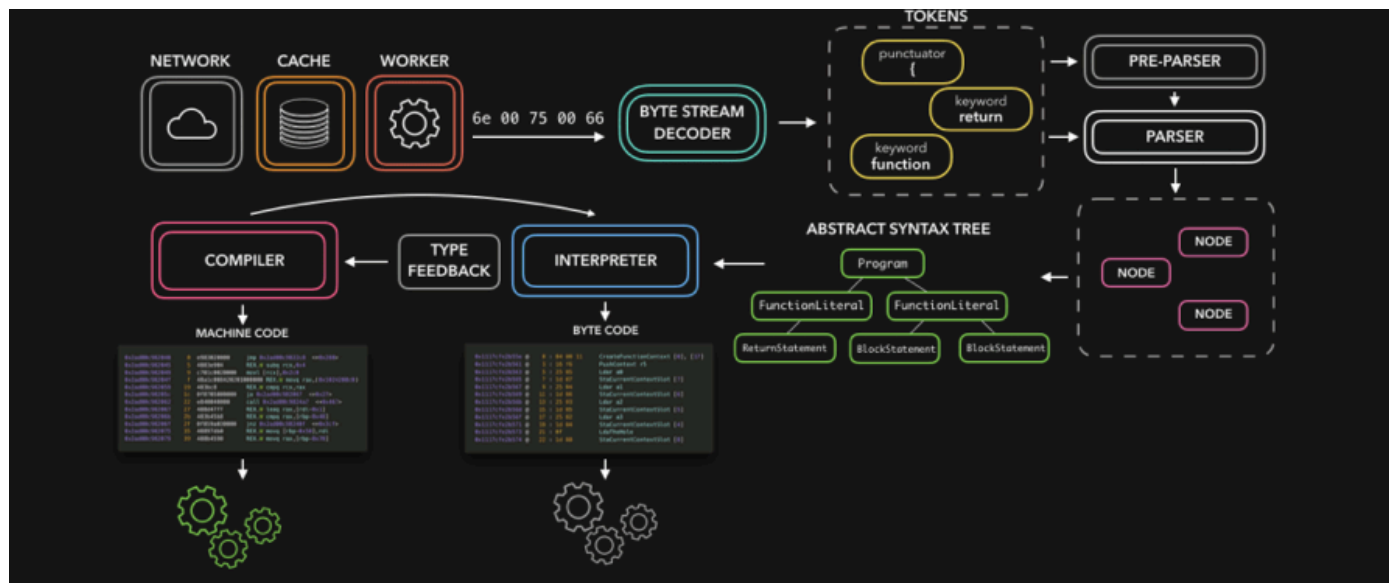
语法分析（英语：syntactic analysis，也叫 parsing）

- 语法分析器也可以称之为parser。

那么我们的JavaScript源码是如何被解析（Parse过程）的呢？

- Blink将源码交给V8引擎，Stream获取到源码并且进行编码转换；
- Scanner会进行词法分析（lexical analysis），词法分析会将代码转换成tokens；
- 接下来tokens会被转换成AST树，经过Parser和PreParser：
 - Parser就是直接将tokens转成AST树架构；
 - PreParser称之为预解析，为什么需要预解析呢？
 - 预解析一方面的作用是快速检查一下是否有语法错误，另一方面也可以进行代码优化。
 - 这是因为并不是所有的JavaScript代码，在一开始时就会被执行。那么对所有的JavaScript代码进行解析，必然会影响网页的运行效率；

- 所以V8引擎就实现了Lazy Parsing（延迟解析）的方案，它的作用是将不必要的函数进行预解析，也就是只解析暂时需要的内容，而对函数的全量解析是在函数被调用时才会进行；
- 比如我们在一个函数outer内部定义了另外一个函数inner，那么inner函数就会进行预解析；
- 生成AST树后，会被Ignition转成字节码（bytecode）并且可以执行字节码，之后的过程就是代码的执行过程（后续会详细分析）。



02-V8内存管理

垃圾回收机制

因为内存的大小是有限的，所以当内存不再需要的时候，我们需要对其进行释放，以便腾出更多的内存空间。

在手动管理内存的语言中，我们需要通过一些方式自己来释放不再需要的内存，比如free函数：

- 但是这种管理的方式其实非常的低效，影响我们编写逻辑的代码的效率；
- 并且这种方式对开发者的要求也很高，并且一不小心就会产生内存泄露（memory leaks），野指针（dangling pointers）；

所以大部分现代的编程语言都有自己的垃圾回收机制：

- 垃圾回收的英文是Garbage Collection，简称GC；
- 对于那些不再使用的对象，我们都称之为是垃圾，它需要被回收，以释放更多的内存空间；
- 而我们的语言运行环境，比如Java的运行环境JVM，JavaScript的运行环境js引擎都会内存 垃圾回收器；
- 垃圾回收器我们也会简称为GC，所以在很多地方你看到GC其实指的是垃圾回收器；

自动垃圾回收提高了开发效率，使开发者可以更多地关注业务逻辑的实现而非内存管理的细节。

- 这在管理复杂数据结构和大量数据时非常重要。

但是这里又出现了另外一个很关键的问题：GC怎么知道哪些对象是不再使用的呢？

这里就要用到GC的实现以及对应的算法；

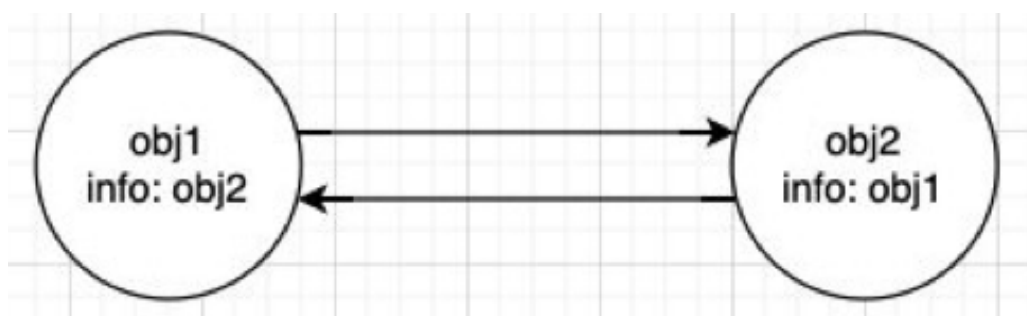
常见GC算法

常见的GC算法 – 引用计数（Reference counting）

引用计数垃圾回收（Reference Counting）：

- 每个对象都有一个关联的计数器，通常称为“引用计数”。
- 当一个对象有一个引用指向它时，那么这个对象的引用就+1；
- 如果另一个变量也开始引用该对象，引用计数加1；如果一个变量停止引用该对象，引用计数减1。
- 当一个对象的引用为0时，这个对象就可以被销毁掉；

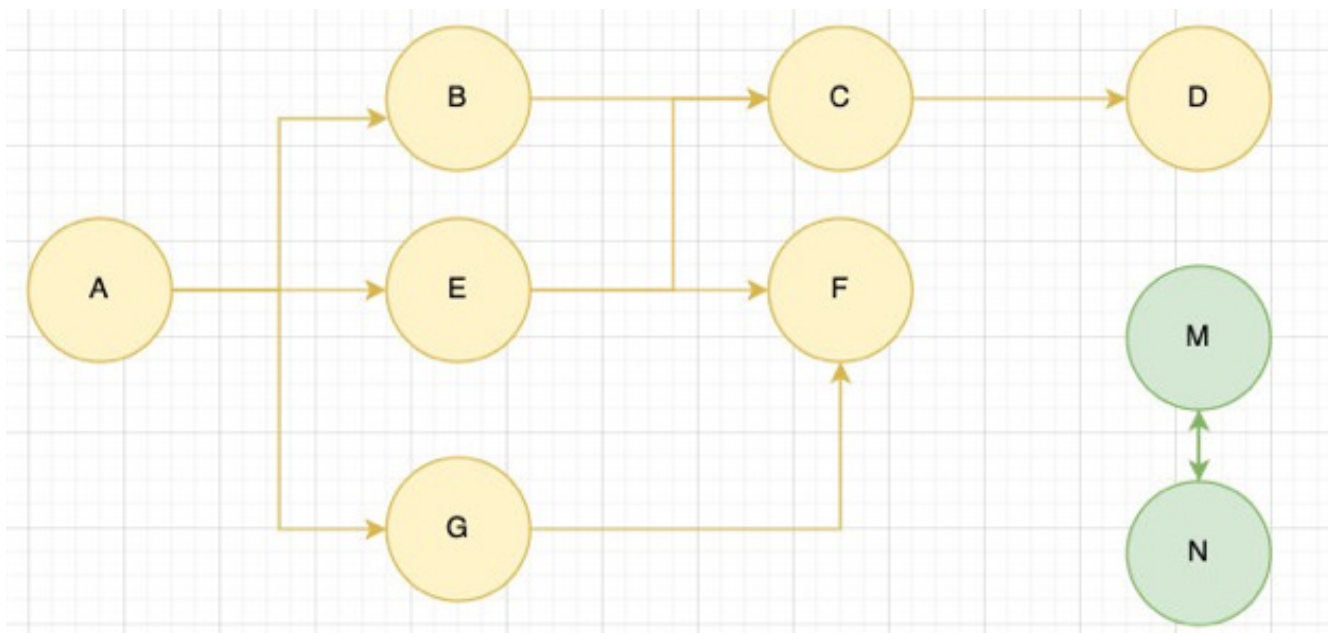
这个算法有一个很大的弊端就是会产生循环引用，当然我们可以通过一些方案，比如弱引用来解决（WeakMap就是弱引用）；



常见的GC算法 – 标记清除（mark-Sweep）

标记清除：

- 标记清除的核心思路是可达性（Reachability）
- 这个算法是设置一个根对象（root object），垃圾回收器会定期从这个根开始，找所有从根开始有引用到的对象，对于哪些没有引用到的对象，就认为是不可用的对象；
- 在这个阶段，垃圾回收器标记所有可达的对象，之后，垃圾回收器遍历所有的对象，收集那些在标记阶段未被标记为可达的对象。
- 这些对象被视为垃圾，因它们不再被程序中的其他活跃对象或根对象所引用。
- 这个算法可以很好的解决循环引用的问题；



其他GC算法补充

JS引擎比较广泛的采用的就是可达性中的标记清除算法，当然类似于V8引擎为了进行更好的优化，它在算法的实现细节上也会结合一些其他的算法。

标记整理（Mark-Compact）和“标记－清除”相似：

- 不同的是，回收期间同时会将保留的存储对象搬运汇集到连续的内存空间，从而整合空闲空间，避免内存碎片化；

分代收集（Generational collection）—— 对象被分成两组：“新的”和“旧的”。

- 许多对象出现，完成它们的工作并很快死去，它们可以很快被清理；
- 那些长期存活的对象会变得“老旧”，而且被检查的频次也会减少；

增量收集（Incremental collection）

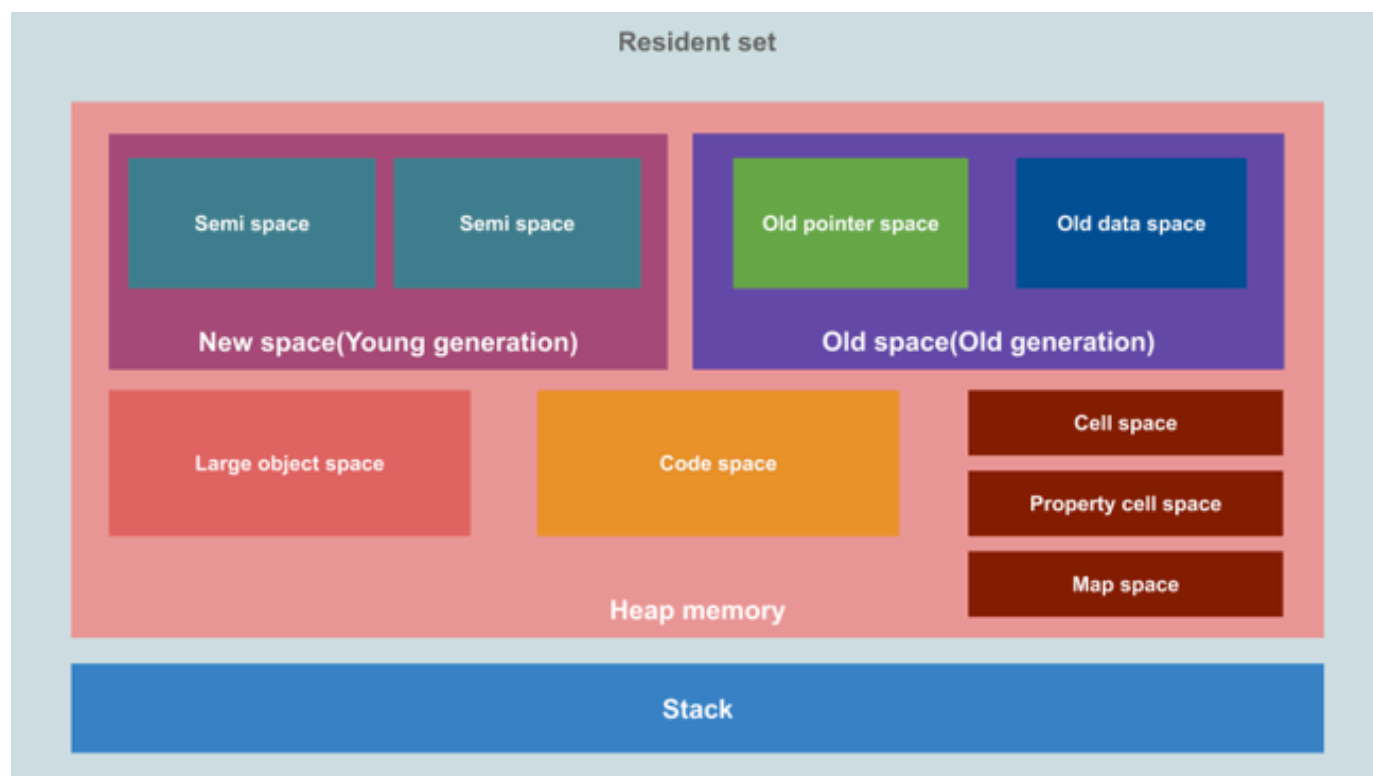
- 如果有许多对象，并且我们试图一次遍历并标记整个对象集，则可能需要一些时间，并在执行过程中带来明显的延迟。
- 所以引擎试图将垃圾收集工作分成几部分来做，然后将这几部分会逐一进行处理，这样会有许多微小的延迟而不是一个大的延迟；

闲时收集（Idle-time collection）

- 垃圾收集器只会在 CPU 空闲时尝试运行，以减少可能对代码执行的影响。

V8内存图详解

事实上，V8引擎为了提供内存的管理效率，对内存进行非常详细的划分：



新生代空间 (New Space / Young Generation)

- 作用：主要用于存放生命周期短的小对象。这部分空间较小，但对象的创建和销毁都非常频繁。
- 组成：新生代内存被分为两个半空间：From Space 和 To Space。
 - 初始时，对象被分配到 From Space 中。
 - 使用复制算法（Copying Garbage Collection）进行垃圾回收。
 - 当进行垃圾回收时，活动的对象（即仍然被引用的对象）被复制到 To Space 中，而非活动的对象（不再被引用的对象）被丢弃。
 - 完成复制后，From Space 和 To Space 的角色互换，新的对象将分配到新的 From Space 中，原 To Space 成为新的 From Space。

老生代空间 (Old Space / Old Generation)

- 作用：存放生命周期长或从新生代晋升过来的对象。
- 当对象在新生代中经历了一定数量的垃圾回收周期后（通常是一到两次），且仍然存活，它们被认为是生命周期较长的对象。
- 分为三个主要区域：
 - 老指针空间 (Old Pointer Space)：主要存放包含指向其他对象的指针的对象。
 - 老数据空间 (Old Data Space)：用于存放只包含原始数据（如数值、字符串）的对象，不含指向其他对象的指针。

大对象空间 (Large Object Space)：用于存放大对象，如超过新生代大小限制的数组或对象。

- 这些对象直接在大对象空间中分配，避免在新生代和老生代之间的复制操作。

代码空间 (Code Space)：存放编译后的函数代码。

单元空间 (Cell Space): 用于存放小的数据结构如闭包的变量环境。

属性单元空间 (Property Cell Space): 存放对象的属性值

- 主要针对全局变量或者属性值，对于访问频繁的全局变量或者属性值来说，V8在这里存储是为了提高它的访问效率。

映射空间 (Map Space): 存放对象的映射（即对象的类型信息，描述对象的结构）。

- 当你定义一个 Person 构造函数时，可以通过它创建出来person1和person2。
- 这些实例（person1 和 person2）本身存储在堆内存的相应空间中，具体是新生代还是老生代取决于它们的生命周期和大小。
- 每个实例都会持有一个指向其映射的指针，这个映射指明了如何访问 name 和 age 属性（目的是访问属性效果变高）。

堆内存 (Heap Memory) 与 栈 (Stack)

- 堆内存: JavaScript 对象、字符串等数据存放的区域，按照上述分类进行管理。
- 栈: 用于存放执行上下文中的变量、函数调用的返回地址（继续执行哪里的代码）等，栈有助于跟踪函数调用的顺序和局部变量。

03-什么是垃圾回收机制？并且它是如何在现代编程语言中管理内存的？

虽然随着硬件的发展，目前计算机的内存已经足够大，但是随着任务的增多依然可能会面临内存紧缺的问题，因此管理内存依然非常重要。（前提）

垃圾回收（Garbage Collection, GC）是自动内存管理的一种机制，它帮助程序自动释放不再使用的内存。在不需要手动释放内存的现代编程语言中，垃圾回收机制扮演着非常重要的角色，通过自动识别和清除“垃圾”数据来防止内存泄漏，从而管理内存资源。（作用）

在运行时，垃圾回收机制主要通过追踪每个对象的生命周期来工作。（原理）

- 对象通常在它们不再被程序的任何部分引用时被视为垃圾。
- 一旦这些对象被识别，垃圾回收器将自动回收它们占用的内存空间，使这部分内存可以重新被分配和使用。

垃圾回收机制有几种不同的实现方法，最常见的包括（实现，可以先回答几种，表示你对GC的理解，等下再回答V8的内容）：

- **引用计数**: 每个对象都有一个与之关联的计数器，记录引用该对象的次数。当引用计数变为零时，意味着没有任何引用指向该对象，因此可以安全地回收其内存。
- **标记-清除**: 这种方法通过从根对象集合开始，标记所有可达的对象。所有未被标记的对象都被视为垃圾，并将被清除。
- **标记-整理**: 与标记-清除相似，但在清除阶段，它还会移动存活的对象，以减少内存碎片。

04-V8引擎的垃圾回收机制具体是如何工作的？

V8引擎使用了一种高度优化的垃圾回收机制来管理内存，采用了标记-清除、标记整理，同时又结合了多种策略来实现高效的内存管理，包括结合了分代回收（Generational Collection）和增量回收（Incremental Collection）等多种策略。

- **分代回收**：V8将对象分为“新生代”和“老生代”。新生代存放生命周期短的小对象，使用高效的复制式垃圾回收算法；而老生代存放生命周期长或从新生代晋升而来的对象，使用标记-清除或标记-整理算法。这种分代策略减少了垃圾回收的总体开销，尤其是针对短命对象的快速回收。
- **增量回收**：为了减少垃圾回收过程中的停顿时间，V8实现了增量回收。这意味着垃圾回收过程被分解为许多小步骤，这些步骤穿插在应用程序的执行过程中进行。这有助于避免长时间的停顿，改善了应用程序的响应性和性能。
- **延迟清理和空闲时间收集**：V8还尝试在CPU空闲时进行垃圾回收，以进一步减少对程序执行的影响。

这些技术的结合使得V8能够在执行JavaScript代码时有效地管理内存，同时最小化垃圾回收对性能的影响。

（另外可以结合V8内存图来表现出你对V8内存的深入理解）

05-JavaScript有哪些操作可能引起内存泄漏？如何在开发中避免？（性能优化）

在JavaScript中，内存泄漏通常是指程序中已经不再需要使用的内存，由于某些原因未被垃圾回收器回收，从而导致可用内存逐渐减少。

这些内存泄漏通常是由不当的编程实践引起的，常见的引起内存泄漏的操作有如下情况：

- **全局变量滥用**：创建的全局变量（例如，忘记使用 `var`，`let`，或 `const` 声明变量）可能会导致这些变量不被回收。
- **未清理的定时器和回调函数**：比如使用 `setInterval` 在不适用时没有及时清除，阻止它们被回收。
- **闭包**：闭包可以维持对外部函数作用域的引用，如果这些闭包一直存活，它们引用的外部作用域（及其变量）也无法被回收。
- **DOM 引用**：JavaScript 对象持有已从 DOM 中删除的元素的引用，这会阻止这些 DOM 元素的内存被释放。
- **监听器的回调**：在使用完毕后没有从 DOM 元素上移除事件监听器，这可能导致内存泄漏。

开发中如何避免呢？重要的是平时在开发中就要尽量按照规范来编写代码。

- **使用局部变量**：尽量使用局部变量，避免无限制地使用全局变量。
- **及时清理**：使用 `clearInterval` 或 `clearTimeout` 取消不再需要的定时器。
- **优化闭包的使用**：理解闭包和它们的工作方式。只保留必要的在闭包中，避免循环引用。
- **谨慎操作 DOM 引用**：当从 DOM 中移除元素时，确保删除所有相关的 JavaScript 引用。包括DOM元素监听器的移除。
- **工具和检测**：利用浏览器的开发者工具进行性能分析和内存分析。
- **代码审查**：定期进行代码审查，关注那些可能导致内存泄漏的编程实践，比如对全局变量的使用、事件监听器的添加与移除等。