详解 Chrome V8 引擎

1、JavaScript 的基本特性和设计思想

JavaScript 的语言设计，借鉴了很多语言的特性，比如: **C 语言的基本语法、Java 的类型系统和内存管理、Scheme 的函数作为一等公民，还有 Self 基于原型（prototype）的继承机制。**JavaScript 是一门非常优秀的语言，特别是**「原型继承机制」和「函数是一等公民」**这两个设计。

JavaScript也有很多的先天不足，例如：使用 new 加构造函数来创建对象，这种方式的背后隐藏了太多的细节，非常容易增加代码出错概率，而且也大大增加了新手的学习成本；初期的 JavaScript 没有块级作用域机制，使得 JavaScript 需要采取变量提升的策略，而变量提升又是非常反人性的设计。

了解 V8 我们需要关注 JavaScript 这些独特的设计思想和特性背后的实现。比如，为了实现函数是一等公民的特性，JavaScript 采取了基于对象的策略；再比如为了实现原型继承，V8 为每个对象引入了 __proto__ 属性。

2、为什么需要 JavaScript 引擎?

我们写的 JavaScript 代码直接交给浏览器或者 Node 执行时，底层的 CPU 是不认识的，也没法执行。**CPU 只认识自己的指令集，指令集对应的是汇编代码。**写汇编代码是一件很痛苦的事情。并且**不同类型的 CPU 的指令集是不一样的，那就意味着需要给每一种 CPU 重写汇编代码。**

JavaScript 引擎可以将 JS 代码编译为不同 CPU(Intel, ARM 以及 MIPS 等)对应的汇编代码，这样我们就不需要去翻阅每个 CPU 的指令集手册来编写汇编代码了。当然，**JavaScript 引擎的工作也不只是编译代码，它还要负责执行代码、分配内存以及垃圾回收。**

```ABAP
# 将一个寄存器中的数据移动到另外一个寄存器中
1000100111011000  #机器指令
mov ax,bx         #汇编指令
```

扩展资料：汇编语言入门教程【阮一峰】 <https://link.zhihu.com/?target=https%3A//link.segmentfault.com/%3Fenc%3DXNRJHXJgBwjZaEkIc4nYSQ%253D%253D.U6iw4DaUv8uHytyOnOOzq7jWt4eyAU8vO6t0lbGjftUK%252FG7Q6iAntMKRfGSlJ%252FFY> 、葡萄的前端杂货铺：想初步了解编译原理？看这篇文章就够了 <https://zhuanlan.zhihu.com/p/362072187>

JavaScript 引擎的主要功能，就是结合 JavaScript 语言的特性和本质来编译执行它。

3、热门 JavaScript 的引擎有哪些?

- V8 (Google)，用 C++编写，开放源代码，由 Google 丹麦开发，是 Google Chrome 的一部分，也用于 Node.js。
- JavaScriptCore (Apple)，开放源代码，用于 webkit 型浏览器，如 Safari ，2008 年实现了编译器和字节码解释器，升级为了 SquirrelFish。苹果内部代号为“Nitro”的 JavaScript 引擎也是基于 JavaScriptCore 引擎的。
- Rhino，由 Mozilla 基金会管理，开放源代码，完全以 Java 编写，用于 HTMLUnit
- SpiderMonkey (Mozilla)，第一款 JavaScript 引擎，早期用于 Netscape Navigator，现时用于 Mozilla Firefox。
- Chakra (JScript 引擎)，用于 Internet Explorer。
- Chakra (JavaScript 引擎)，用于 Microsoft Edge。
- KJS，KDE 的 ECMAScript / JavaScript 引擎，最初由哈里·波顿开发，用于 KDE 项目的 Konqueror 网页浏览器中。
- JerryScript — 三星推出的适用于嵌入式设备的小型 JavaScript 引擎。
- 其他：Nashorn、QuickJS 、 Hermes

今天我们的主角是 **V8，**它是当下使用最广泛的 JavaScript 虚拟机，全球有超过 25 亿台安卓设备，而这些设备中都使用了 Chrome 浏览器，所以我们写的 JavaScript 应用，大都跑在 V8 上。

在正式介绍V8之前我们先来看下V8的调试工具D8。

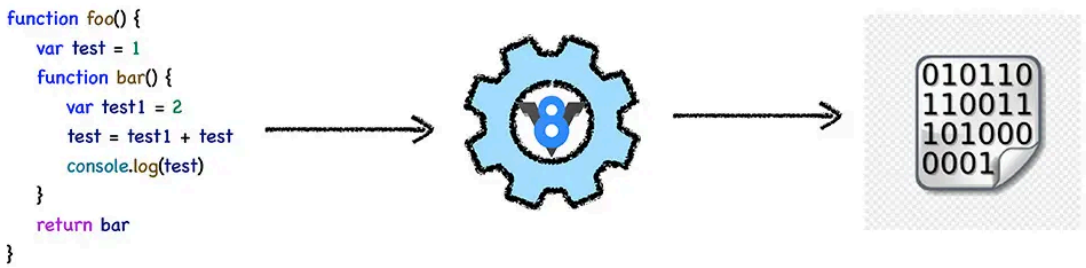
4、什么是 D8?

D8 是一个非常有用的调试工具，你可以把它看成是**debug for V8**的缩写。我们**可以使用 d8 来查看 V8 在执行 JavaScript 过程中的各种中间数据，比如作用域、AST、字节码、优化的二进制代码、垃圾回收的状态，还可以使用 d8 提供的私有 API 查看一些内部信息**。

V8源码编译出来的可执行程序名为d8。d8作为V8引擎在命令行中可以使用的交互shell存在。Google官方已经不记得d8这个名字的由来，但是做为 **delveloper shell** 的缩写，用首字母d和8结合，恰到好处。还有一种说法是d8最初叫 developer shell，因为d后面有8个字符，因此简写为d8，类似于i18n(internationalization)这样的简写。

5、什么是 V8?

V8 是由 Google 开发的开源**JavaScript 引擎，**是 JavaScript **虚拟机**的一种，模拟实际计算机各种功能来**实现代码的编译和执行**。我们可以简单地把 JavaScript 虚拟机理解成是一个翻译程序，将人类能够理解的 **编程语言 JavaScript**，翻译成机器能够理解的**机器语言。**目前主要用在 Chrome 浏览器和 Node.js 中。如下图所示：



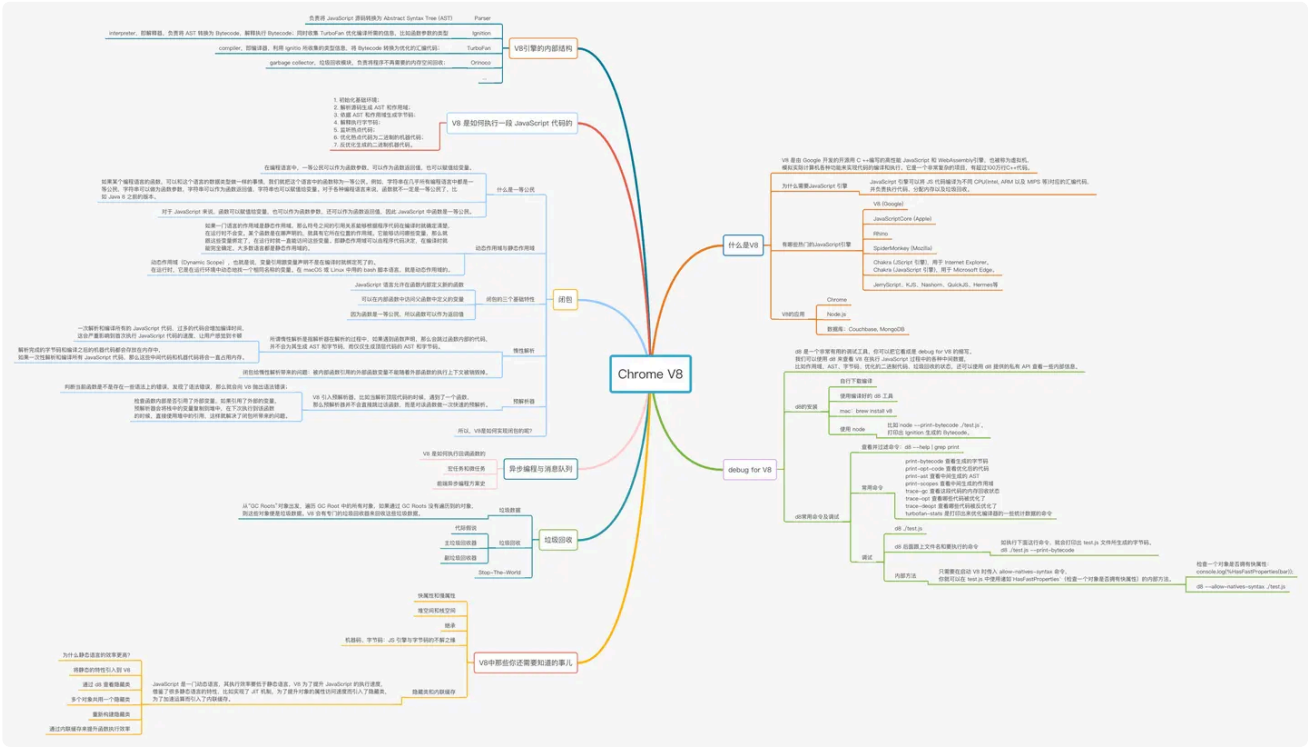
引用自《极客时间–图解 Google V8》

图中中间的**「黑盒」**就是 JavaScript 引擎 V8。目前市面上有很多种 JavaScript 引擎，诸如 SpiderMonkey、V8、JavaScriptCore 等。而由谷歌开发的开源项目 V8 是当下使用最广泛的 JavaScript 虚拟机。

V8 是用 C ++ 编写的开源高性能 JavaScript 和 WebAssembly 引擎，它已被用于 Chrome 和 Node.js 等。可以运行在 Windows 7+，macOS 10.12+和使用 x64，IA–32，ARM 或 MIPS 处理器的 Linux 系统上。**V8 最早被开发用以嵌入到 Google 的开源浏览器 Chrome 中**，第一个版本随着第一版 Chrome 于 2008 年 9 月 2 日发布。**但是 V8 是一个可以独立运行的模块，完全可以嵌入到任何 C ++应用程序中。**著名的 Node.js（一个异步的服务器框架，可以在服务端使用 JavaScript 写出高效的网络服务器）就是基于 V8 引擎的，Couchbase, MongoDB 也使用了 V8 引擎。

和其他JS引擎一样，**V8 会编译 、执行 JavaScript 代码，管理内存，负责垃圾回收，与宿主语言的交互等。通过暴露宿主对象（变量，函数等）到 JavaScript，JavaScript 可以访问宿主环境中的对象，并在脚本中完成对宿主对象的操**

关于V8的方方面面可以看下图，如果图片看不清楚，可以访问这个链接查看：图解V8的方方面面 <https://king-hcj.github.io/images/posts/arts/Chrome-V8.png?raw=true>



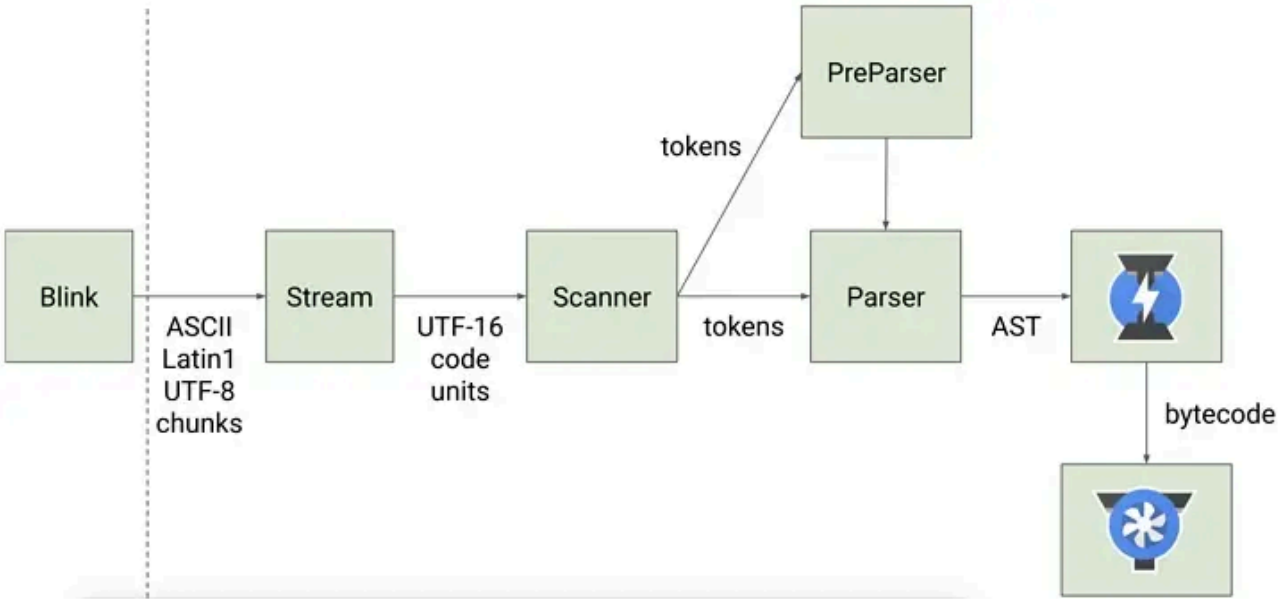
V8

6、V8 引擎的内部结构

V8 是一个非常复杂的项目，有超过 100 万行 C++代码。它由许多子模块构成，其中这 4 个模块是最重要的：

Parse

负责将 JavaScript 源码转换为 Abstract Syntax Tree (AST)。确切的说，在 **Parser** 将 JavaScript 源码转换为 AST前，还有一个叫 **Scanner** 的过程，具体流程如下：



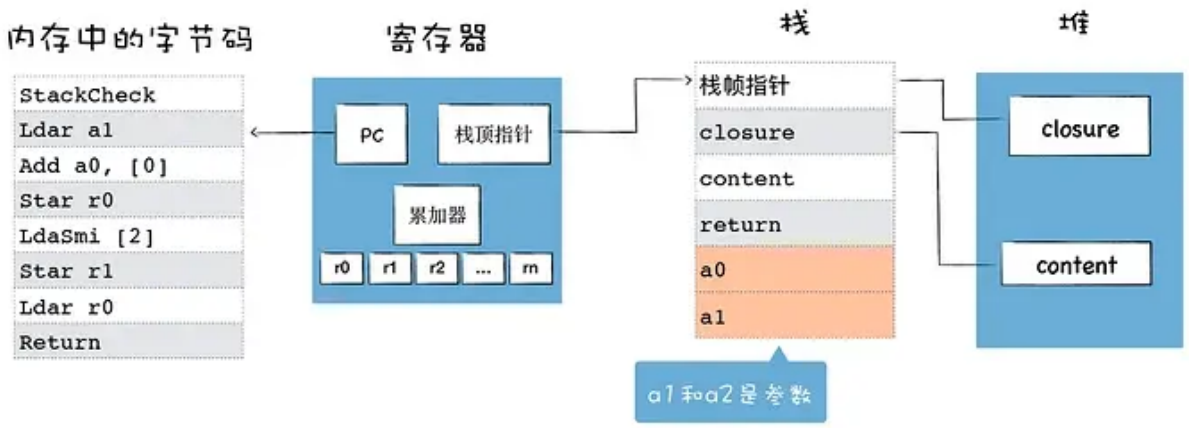
Ignition

即解释器，负责将 AST 转换为 Bytecode，解释执行 Bytecode；同时收集 TurboFan 优化编译所需的信息，比如函数参数的类型；解释器执行时主要有四个模块，内存中的字节码、寄存器、栈、堆。

通常有两种类型的解释器，基于栈 (Stack-based)和 基于寄存器 (Register-based)。

1. 基于栈的解释器：使用栈来保存函数参数、中间运算结果、变量等；
2. 基于寄存器的虚拟机则支持寄存器的指令操作，使用寄存器来保存参数、中间计算结果。

通常，基于栈的虚拟机也定义了少量的寄存器，基于寄存器的虚拟机也有堆栈，其**区别体现在它们提供的指令集体系。大多数解释器都是基于栈的**，比如 Java 虚拟机，.Net 虚拟机，还有早期的 V8 虚拟机。基于堆栈的虚拟机在处理函数调用、解决递归问题和切换上下文时简单明快。而**现在的 V8 虚拟机则采用了基于寄存器的设计**，它将一些中间数据保存到寄存器中。



基于寄存器的解释器架构

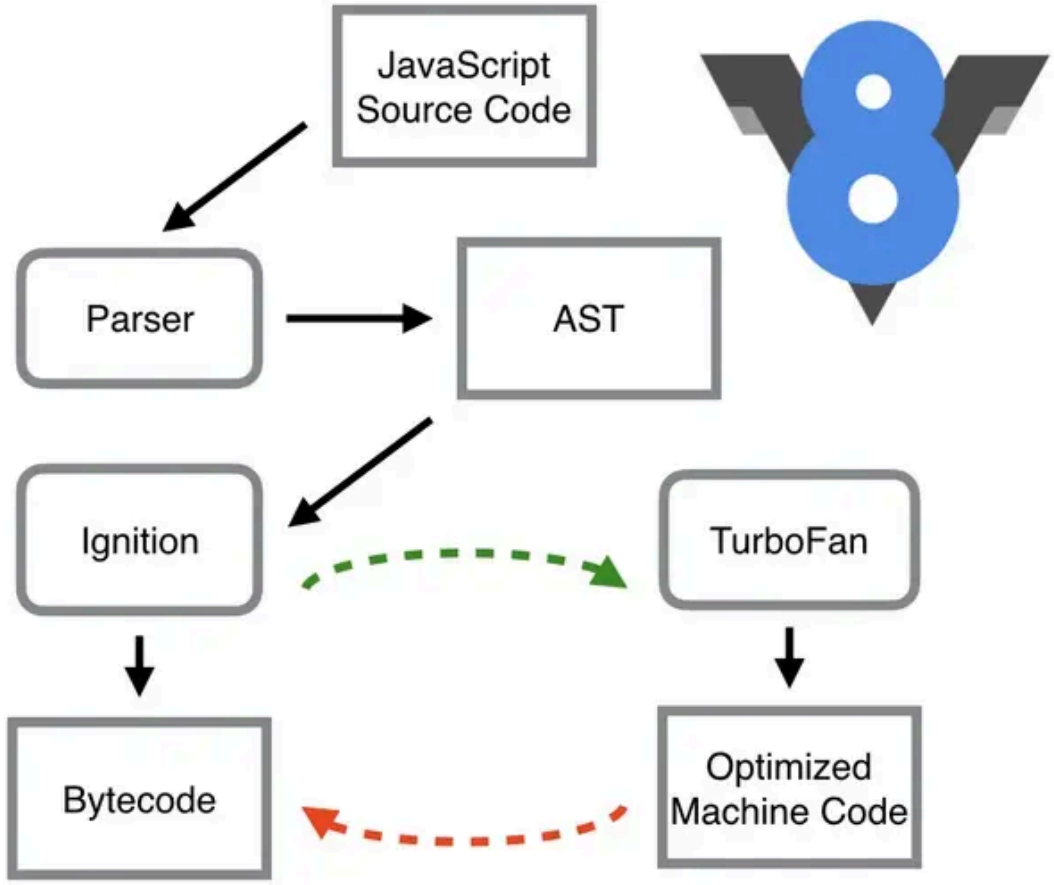
TurboFan

compiler，即编译器，利用 Ignition 所收集的类型信息，将 Bytecode 转换为优化的汇编代码；

Orinoco

garbage collector，垃圾回收模块，负责将程序不再需要的内存空间回收。

其中，Parser，Ignition 以及 TurboFan 可以将 JS 源码编译为汇编代码，其流程图如下：



简单地说，**Parser 将 JS 源码转换为 AST，然后 Ignition 将 AST 转换为 Bytecode，最后 TurboFan 将 Bytecode 转换为经过优化的 Machine Code(实际上是汇编代码)**。

- 如果函数没有被调用，则 V8 不会去编译它。
- 如果函数只被调用 1 次，则 Ignition 将其编译 Bytecode 就直接解释执行了。TurboFan 不会进行优化编译，因为它需要 Ignition 收集函数执行时的类型信息。这就要求函数至少需要执行 1 次，TurboFan 才有可能进行优化编译。
- 如果函数被调用多次，则它有可能会被识别为**热点函数**，且 Ignition 收集的类型信息证明可以进行优化编译的话，这时 TurboFan 则会将 Bytecode 编译为 Optimized Machine Code（已优化的机器码），以提高代码的执行性能。

图片中的红色虚线是逆向的，也就是说Optimized Machine Code 会被还原为 Bytecode，这个过程叫做**Deoptimization**。这是因为 Ignition 收集的信息可能是错误的，比如 add 函数的参数之前是整数，后来又变成了字符串。生成的 Optimized Machine Code 已经假定 add 函数的参数是整数，那当然是错误的，于是需要进行 Deoptimization。

```
function add(x, y) {
  return x + y;
}

add(3, 5);
add('3', '5');
```

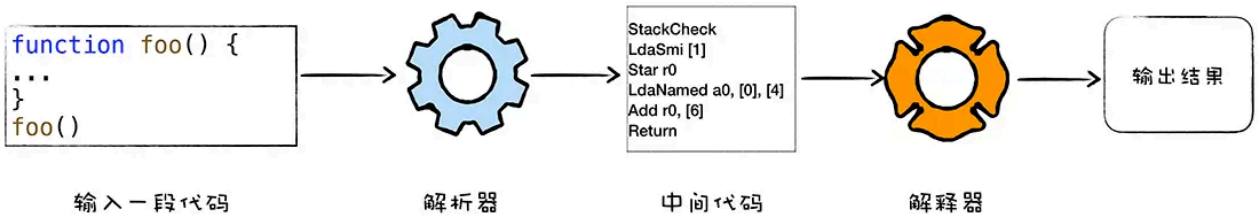
在运行 C、C++以及 Java 等程序之前，需要进行编译，不能直接执行源码；但对于 JavaScript 来说，我们可以直接执行源码(比如：node test.js)，它是在运行的时候先编译再执行，这种方式被称为「**即时编译(Just-in-time compilation)**」，简称为 JIT。因此，V8 也属于 **JIT 编译器**。

7、解释执行和编译执行区别?

V8 本质上是一个虚拟机，因为计算机只能识别二进制指令，所以要让计算机执行一段高级语言通常有两种手段：

解释执行

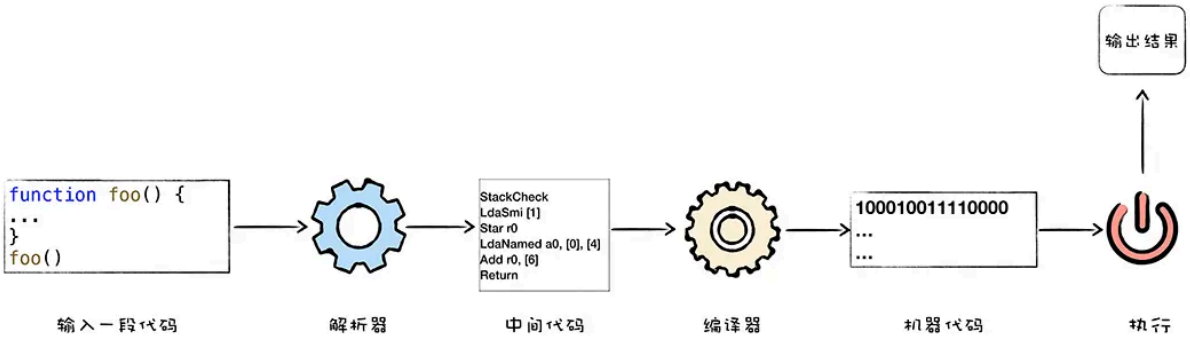
需要先将输入的源代码通过解析器编译成中间代码，之后直接使用解释器解释执行中间代码，然后直接输出结果。具体流程如下图所示：



解释执行流程图

编译执行

采用这种方式时，也需要先将源代码转换为中间代码，然后我们的编译器再将中间代码编译成机器代码。通常编译成的机器代码是以二进制文件形式存储的，需要执行这段程序的时候直接执行二进制文件就可以了。还可以使用虚拟机将编译后的机器代码保存在内存中，然后直接执行内存中的二进制代码。



以上就是计算机执行高级语言的两种基本的方式：**解释执行和编译执行**。但是针对不同的高级语言，这个实现方式还是有很大差异的，比如要执行 C 语言编写的代码，你需要将其编译为二进制代码的文件，然后再直接执行二进制代码。而对于像 Java 语言、JavaScript 语言等，则需要不同虚拟机，模拟计算机的这个编译执行流程。执行 Java 语言，需要经过 Java 虚拟机的转换，执行 JavaScript 需要经过 JavaScript 虚拟机的转换。

解释执行和编译执行都有各自的优缺点

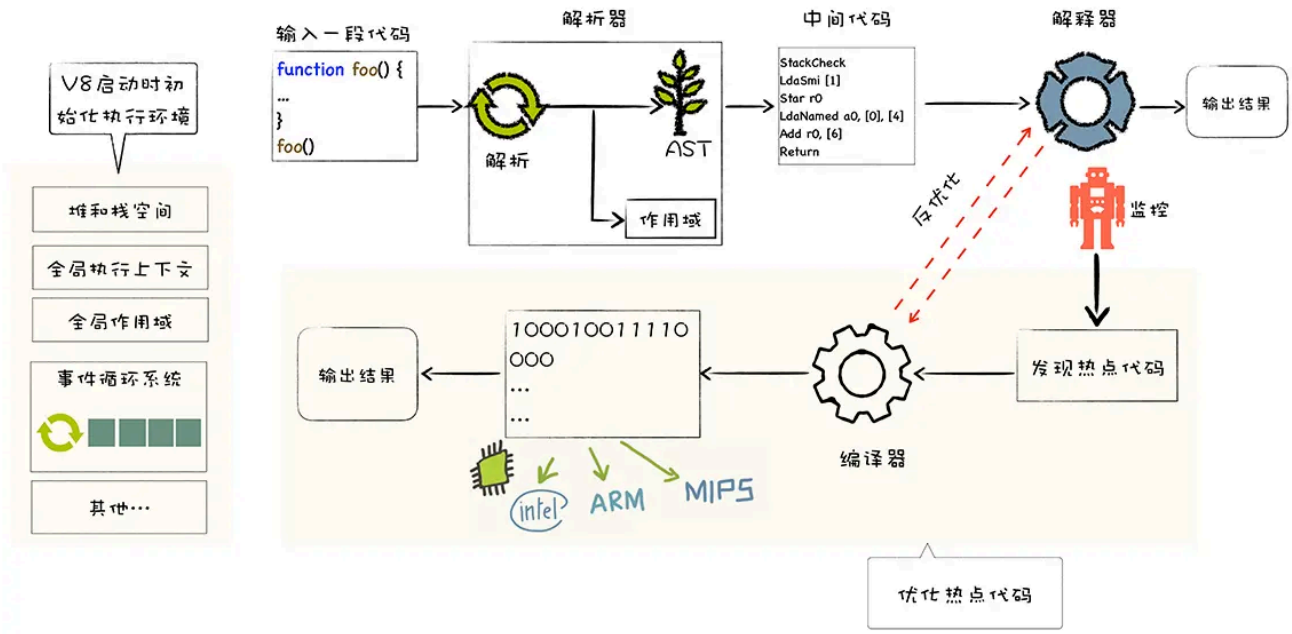
解释执行启动速度快，但是执行时速度慢，而编译执行启动速度慢，但是执行速度快。为了充分地利用解释执行和编译执行的优点，规避其缺点，**V8 采用了一种权衡策略，在启动过程中采用了解释执行的策略，但是如果某段代码的执行频率超过一个值，那么 V8 就会采用优化编译器将其编译成执行效率更加高效的机器代码**。

8、V8 是怎么执行一段 JavaScript 代码的?

在 V8 出现之前，所有的 JavaScript 虚拟机所采用的都是**解释执行**的方式，「这是 JavaScript 执行速度过慢的一个主要原因」。而 V8 率先引入了**即时编译（JIT）**的双轮驱动的设计，这是一种权衡策略，**混合编译执行和解释执行这两种手段**，给 JavaScript 的执行速度带来了极大的提升。

另外，V8 也是早于其他虚拟机引入了**惰性编译、内联缓存、隐藏类**等机制，进一步优化了 JavaScript 代码的编译执行效率。**V8 的出现，将 JavaScript 虚拟机技术推向了一个全新的高度。**

接着通过下图，我们一起看看 V8 执行 JavaScript 代码的完整流程：



V8编译流水线---引用自《极客时间-图解 Google V8》

V8 执行一段 JavaScript 代码所经历的主要流程包括：

- 初始化基础环境；
- 解析源码生成 AST 和作用域；
- 依据 AST 和作用域生成字节码；
- 解释执行字节码；
- 监听热点代码；
- 优化热点代码为二进制的机器代码；
- 反优化生成的二进制机器代码。

V8编译流水线并不复杂，但其中涉及到了很多技术，诸如 **JIT、延迟解析、隐藏类、内联缓存、事件循环系统、垃圾回收机制**等等。这些技术决定着一段 JavaScript 代码能否正常执行，以及代码的执行效率。对提的几个名词做个简单介绍：

「隐藏类（Hide Class）」：是将 JavaScript 中动态类型转换为静态类型的一种技术，可以消除动态类型的语言执行速度过慢的问题。熟悉 了 V8 的工作机制，在编写 JavaScript 时，就能充分利用好隐藏类这种强大的优化特性，写出更加高效的代码。

「惰性解析」：它目的是为了**加速代码的启动速度**，通过对惰性解析机制的学习，可以优化代码更加适应这个机制，从而提高程序性能。

「V8事件循环系统」：事件循环系统和 JavaScript 中的难点**异步编程特性**紧密相关。JavaScript 是单线程的，JavaScript 代码都是在一个线程上执行，如果同一时间发送了多个 JavaScript 执行的请求，就需要排队，也就是进行异步编程。**而V8事件循环系统会调度这些排队任务，保证 JavaScript 代码被 V8 有序地执行。因此也可以说，事件循环系统就是 V8 的心脏，它驱动了 V8 的持续工作。**

「垃圾回收机制」：自动垃圾回收是一种在堆内存中找出哪些对象在被使用，还有哪些对象没被使用，并且将后者删掉的机制。所谓使用中的对象（已引用对象），指的是程序中有指针指向的对象；而未使用中的对象（未引用对象），则没有被任何指针给指向，因此占用的内存也可以被回收掉。而**JavaScript 也是一种自动垃圾回收的语言**。

9、Javascript 与 V8

9.1、一等公民的定义

在编程语言中，**一等公民**可以作为函数参数，可以作为函数返回值，也可以赋值给变量。**如果某个编程语言的函数，可以和这个语言的数据类型做一样的事情，我们就把这个语言中的函数称为一等公民**。

例如，字符串在几乎所有编程语言中都是一等公民，字符串可以做为函数参数，字符串可以作为函数返回值，字符串也可以赋值给变量。对于各种编程语言来说，函数就不一定是一等公民了，比如 Java 8 之前的版本。

对于 JavaScript 来说，函数可以赋值给变量，也可以作为函数参数，还可以作为函数返回值，因此 **JavaScript 中函数是一等公民**。

9.2、动态作用域与静态作用域

如果一门语言的作用域是**静态作用域**，那么**符号之间的引用关系能够根据程序代码在编译时就确定清楚，在运行时不会变**。某个函数是在哪声明的，就具有它所在位置的作用域。它能够访问哪些变量，那么就跟这些变量绑定了，在运行时就一直能访问这些变量。即静态作用域可以由程序代码决定，在编译时就能完全确定。大多数语言都是静态作用域的。

动态作用域（Dynamic Scope）。也就是说，变量引用跟变量声明不是在编译时就绑定死了的。在运行时，它是在运行环境中动态地找一个相同名称的变量。在 macOS 或 Linux 中用的 bash 脚本语言，就是动态作用域的。

9.3、闭包的三个基础特性

- JavaScript 语言允许在函数内部定义新的函数
- 可以在内部函数中访问父函数中定义的变量
- 因为 JavaScript 中的函数是一等公民，所以函数可以作为另外一个函数的返回值

```javascript
// 闭包（静态作用域，一等公民，调用栈的矛盾体）
function foo() {
  var d = 20;
  return function inner(a, b) {
    const c = a + b + d;
    return c;
  };
}
const f = foo();
```

9.4、惰性解析

惰性解析是指解析器在解析的过程中，如果遇到函数声明，那么会跳过函数内部的代码，并不会为其生成 AST 和字节码，而仅仅生成顶层代码的 AST 和字节码。

在编译 JavaScript 代码的过程中，V8 并不会一次性将所有的 JavaScript 解析为中间代码，这主要是基于以下两点：

1. 首先，**如果一次解析和编译所有的 JavaScript 代码**，过多的代码会增加编译时间，这会严重影响到首次执行 JavaScript 代码的速度，让用户感觉到**卡顿**。因为有时候一个页面的 JavaScript 代码很大，如果要将所有的代码一次性解析编译完成，那么会大大增加用户的等待时间；
2. 其次，解析完成的字节码和编译之后的机器代码都会存放在内存中，如果一次性解析和编译所有 JavaScript 代码，那么这些中间代码和机器代码将会一直**占用内存**。

基于以上的原因，所有主流的 JavaScript 虚拟机都实现了惰性解析。**闭包给惰性解析带来的问题**：上文的 d 不能随着 foo 函数的执行上下文被销毁掉。

9.5、预解析器

V8 引入**预解析器**，比如当解析顶层代码的时候，遇到了一个函数，那么预解析器并不会直接跳过该函数，而是对该函数做一次快速的预解析。

1. 判断当前函数是不是存在一些语法上的错误，发现了语法错误，那么就会向 V8 抛出语法错误；
2. 检查函数内部是否引用了外部变量，如果引用了外部的变量，预解析器会将栈中的变量复制到堆中，在下次执行到该函数的时候，直接使用堆中的引用，这样就解决了闭包所带来的问题。

9.6、V8 内部是如何存储对象的：快属性和慢属性

下面的代码会输出什么：

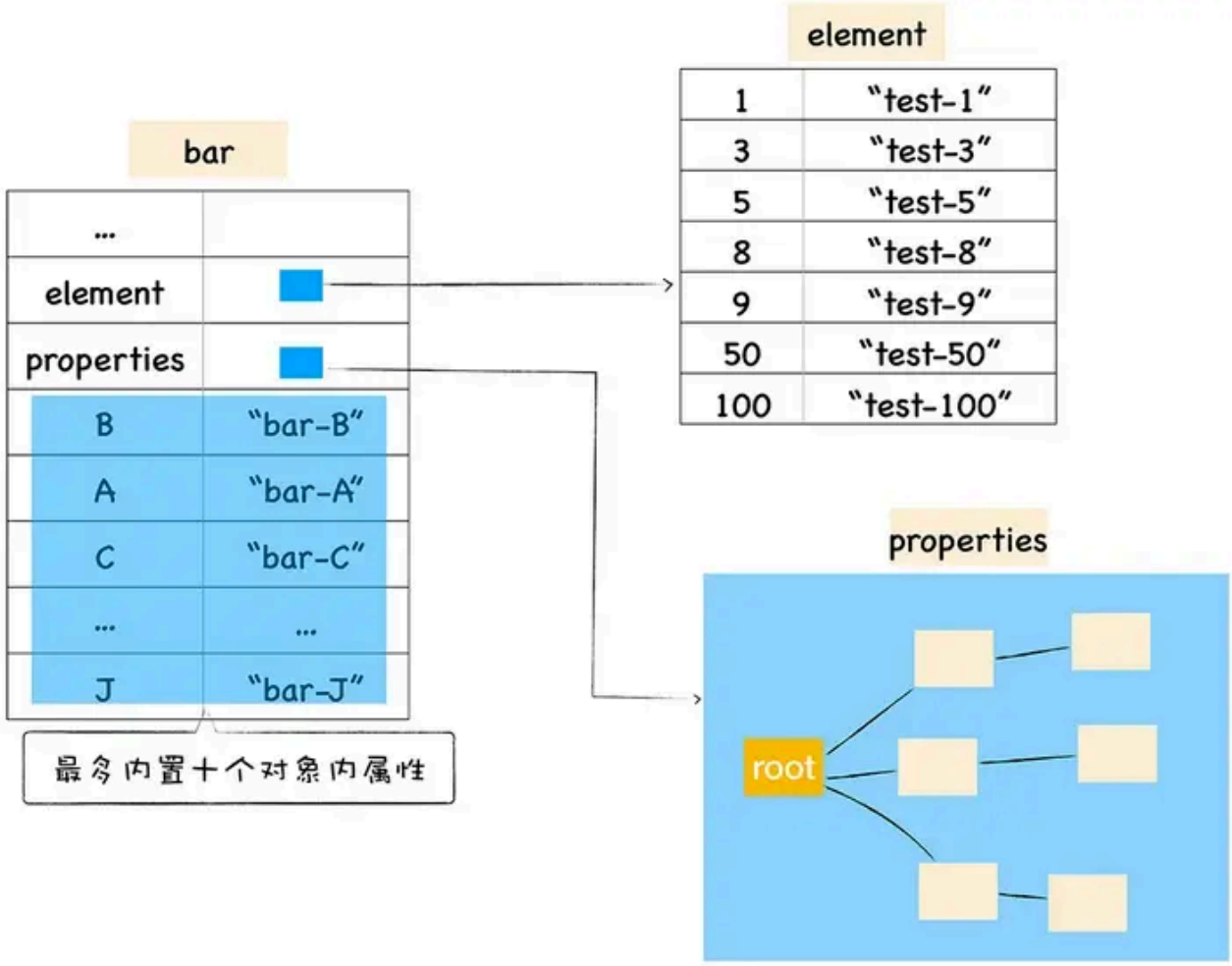
```JavaScript
// test.js
function Foo() {
  this[200] = 'test-200';
  this[1] = 'test-1';
  this[100] = 'test-100';
  this['B'] = 'bar-B';
  this[50] = 'test-50';
  this[9] = 'test-9';
  this[8] = 'test-8';
  this[3] = 'test-3';
  this[5] = 'test-5';
  this['D'] = 'bar-D';
  this['C'] = 'bar-C';
}
var bar = new Foo();

for (key in bar) {
  console.log(`index:${key}  value:${bar[key]}`);
}
//输出:
// index:1  value:test-1
// index:3  value:test-3
// index:5  value:test-5
// index:8  value:test-8
// index:9  value:test-9
// index:50  value:test-50
// index:100  value:test-100
// index:200  value:test-200
// index:B  value:bar-B
// index:D  value:bar-D
// index:C  value:bar-C
```

在ECMAScript 规范中定义了**数字属性应该按照索引值大小升序排列，字符串属性根据创建时的顺序升序排列**。在这里我们把对象中的数字属性称为**排序属性**，在 V8 中被称为 elements，字符串属性就被称为**常规属性**，在 V8 中被称为 properties。在 V8 内部，为了有效地提升存储和访问这两种属性的性能，分别使用了两个线性数据结构来分别保存排序属性和常规属性。同时 v8 将部分常规属性直接存储到对象本身，我们把这称为**对象内属性 (in-object properties)**，不过对象内属性的数量是固定的，默认是 10 个。

```javascript
function Foo(property_num, element_num) {
  //添加可索引属性
  for (let i = 0; i < element_num; i++) {
    this[i] = `element${i}`;
  }
  //添加常规属性
  for (let i = 0; i < property_num; i++) {
    let ppt = `property${i}`;
    this[ppt] = ppt;
  }
}
var bar = new Foo(10, 10);
```

可以通过 Chrome 开发者工具的 Memory 标签，捕获查看当前的**内存快照**。通过增大第一个参数来查看存储变化。（Console面板运行以上代码，打开Memory面板，通过点击 `Take heap snapshot` 记录内存快照，点击快照，筛选出Foo进行查看。可参考使用 chrome-devtools Memory 面板了解Memory面板。）

我们将保存在线性数据结构中的属性称之为“**快属性**”，因为线性数据结构中只需要通过索引即可以访问到属性，虽然访问线性结构的速度快，但是**如果从线性结构中添加或者删除大量的属性时，则执行效率会非常低，这主要因为会产生大量时间和内存开销**。因此，如果一个对象的属性过多时，V8 就会采取另外一种存储策略，那就是“**慢属性**”策略，但慢属性的对象内部会有独立的非线性数据结构（字典）作为属性存储容器。所有的属性元信息不再是线性存储的，而是直接保存在属性字典中



v8 属性存储

因为 JavaScript 中的对象是由一组组属性和值组成的，所以最简单的方式是使用一个字典来保存属性和值，但是由于字典是非线性结构，所以如果使用字典，读取效率会大大降低。为了提升查找效率，**V8 在对象中添加了两个隐藏属性，排序属性和常规属性**，element 属性指向了 elements 对象，在 elements 对象中，会按照顺序存放排序属性。properties 属性则指向了 properties 对象，在 properties 对象中，会按照创建时的顺序保存常规属性。

通过引入这两个属性，加速了 V8 查找属性的速度，为了更加进一步提升查找效率，V8 还实现了内置内属性的策略，当常规属性少于一定数量时，V8 就会将这些常规属性直接写进对象中，这样又节省了一个中间步骤。

但是**如果对象中的属性过多时，或者存在反复添加或者删除属性的操作，那么 V8 就会将线性的存储模式降级为非线性的字典存储模式，这样虽然降低了查找速度，但是却提升了修改对象的属性的速度**。

9.7、堆空间和栈空间

栈空间

现代语言都是基于函数的，每个函数在执行过程中，都有自己的生命周期和作用域，当函数执行结束时，其作用域也会被销毁，因此，我们会使用栈这种数据结构来管理函数的调用过程，我们也把管理函数调用过程的栈结构称之为**调用栈**。

栈空间主要是用来管理 JavaScript 函数调用的，栈是内存中连续的一块空间，同时栈结构是“先进后出”的策略。在函数调用过程中，涉及到上下文相关的内容都会存放在栈上，比如原生类型、引用到的对象的地址、函数的执行状态、this 值等都会存在在栈上。当一个函数执行结束，那么该函数的执行上下文便会被销毁掉。

栈空间的最大的特点是空间连续，所以在栈中每个元素的地址都是固定的，因此栈空间的查找效率非常高，但是通常在内存中，很难分配到一块很大的连续空间，因此，V8 对栈空间的大小做了限制，如果函数调用层过深，那么 V8 就有可能抛出栈溢出的错误。

JavaScript | 运行代码 复制代码

```
// 栈溢出
function factorial(n) {
  if (n === 1) {
    return 1;
  }
  return n * factorial(n - 1);
}
console.log(factorial(50000));
```

栈的优势和缺点：

1. 栈的结构非常适合函数调用过程。
2. **在栈上分配资源和销毁资源的速度非常快**，这主要归结于栈空间是连续的，分配空间和销毁空间只需要移动下指针就可以了。
3. 虽然操作速度非常快，但是栈也是有缺点的，其中最大的缺点也是它的优点所造成的，那就是**栈是连续的**，所以要想在内存中分配一块连续的大空间是非常难的，因此**栈空间是有限的**。

堆**空间**

堆空间是一种**树**形的存储结构，**用来存储对象类型的离散的数据**，JavaScript 中除了原生类型的数据，其他的都是对象类型，诸如函数、数组，在浏览器中还有 window 对象、document 对象等，这些都是存在堆空间的。

宿主在启动 V8 的过程中，会同时创建堆空间和栈空间，再继续往下执行，产生的新数据都会存放在这两个空间中。

9.8、继承

继承就是一个对象可以访问另外一个对象中的属性和方法，在 JavaScript 中，我们通过原型和原型链的方式来实现了继承特性

javaScript 的每个对象都包含了一个隐藏属性 `__proto__` ，我们就把该隐藏属性 `__proto__` 称之为该对象的原型 (prototype)， `__proto__` 指向了内存中的另外一个对象，我们就把 `__proto__` 指向的对象称为该对象的原型对象，那么该对象就可以直接访问其原型对象的方法或者属性。

JavaScript 中的继承非常简洁，就是每个对象都有一个原型属性，该属性指向了原型对象，查找属性的时候，JavaScript 虚拟机会沿着原型一层一层向上查找，直至找到正确的属性。

JavaScript | 运行代码 复制代码

```
var animal = {
  type: 'Default',
  color: 'Default',
  getInfo: function () {
    return `Type is: ${this.type}, color is ${this.color}.`;
  },
};
var dog = {
  type: 'Dog',
  color: 'Black',
};
```

利用 `__proto__` 实现继承：

JavaScript | 运行代码 复制代码

```
dog.__proto__ = animal;
dog.getInfo();
```

通常隐藏属性是不能使用 JavaScript 来直接与之交互的。虽然现代浏览器都开了一个口子，让 JavaScript 可以访问隐藏属性 `__proto__` ，但是在实际项目中，我们不应该直接通过 `__proto__` 来访问或者修改该属性，其主要原因有两个：

- 首先，**这是隐藏属性，并不是标准定义的**;
- 其次，**使用该属性会造成严重的性能问题**。因为 JavaScript 通过隐藏类优化了很多原有的对象结构，所以通过直接修改 `__proto__` 会直接破坏现有已经优化的结构，触发 V8 重构该对象的隐藏类！

9.9、构造函数是怎么创建对象的?

在 JavaScript 中，**使用 new 加上构造函数的这种组合来创建对象和实现对象的继承**。不过使用这种方式**隐含的语义过于隐晦**。其实是 JavaScript 为了吸引 Java 程序员、在语法层面去蹭 Java 热点，所以就被硬生生地强制加入了非常不协调的关键字 new。

```javascript
function DogFactory(type, color) {
  this.type = type;
  this.color = color;
}
var dog = new DogFactory('Dog', 'Black');
```

其实当 V8 执行上面这段代码时，V8 在背后悄悄地做了以下几件事情：

```javascript
var dog = {};
dog.__proto__ = DogFactory.prototype;
DogFactory.call(dog, 'Dog', 'Black');
```

9.10、隐藏类和内联缓存

JavaScript 是一门动态语言，其执行效率要低于静态语言，V8 为了提升 JavaScript 的执行速度，借鉴了很多静态语言的特性，比如实现了 JIT 机制，**为了提升对象的属性访问速度而引入了隐藏类，为了加速运算而引入了内联缓存。**

为什么静态语言的效率更高?

静态语言中，如 C++ 在声明一个对象之前需要定义该对象的结构，代码在执行之前需要先被编译，编译的时候，每个对象的形状都是固定的，也就是说，在代码的执行过程中是无法被改变的。可以直接通过**偏移量**查询来查询对象的属性值，这也就是静态语言的执行效率高的一个原因。

JavaScript 在运行时，对象的属性是可以被修改的，所以当 V8 使用了一个对象时，比如使用了 obj.x 的时候，它并不知道该对象中是否有 x，也不知道 x 相对于对象的偏移量是多少，也就是说 V8 并不知道该对象的具体的形状。那么，当在 JavaScript 中要查询对象 obj 中的 x 属性时，V8 会按照具体的规则一步一步来查询，这个过程非常的慢且耗时。

将静态的特性引入到 V8

V8 采用的一个思路就是将 JavaScript 中的对象静态化，也就是 V8 在运行 JavaScript 的过程中，会假设 JavaScript 中的对象是静态的。

具体地讲，V8 对每个对象做如下两点假设：

1. 对象创建好了之后就不会添加新的属性；
2. 对象创建好了之后也不会删除属性。

符合这两个假设之后，V8 就可以对 JavaScript 中的对象做深度优化了。V8 会为每个对象创建一个**隐藏类**，对象的隐藏类中记录了该对象一些基础的布局信息，包括以下两点：

1. 对象中所包含的所有的属性；

2．每个属性相对于对象的偏移量。

有了隐藏类之后，那么当 V8 访问某个对象中的某个属性时，就会**先去隐藏类中查找该属性相对于它的对象的偏移量**，有了偏移量和属性类型，V8 就可以直接去内存中取出对应的属性值，而不需要经历一系列的查找过程，那么这就大大提升了 V8 查找对象的效率。

在 V8 中，把隐藏类又称为 map，每个对象都有一个 map 属性，其值指向内存中的隐藏类；

map 描述了对象的内存布局，比如对象都包括了哪些属性，这些数据对应于对象的偏移量是多少。

通过 d8 查看隐藏类

JavaScript | 运行代码 复制代码

```
// test.js
let point1 = { x: 100, y: 200 };
let point2 = { x: 200, y: 300 };
let point3 = { x: 100 };
%DebugPrint(point1);
%DebugPrint(point2);
%DebugPrint(point3);

./d8 --allow-natives-syntax ./test.js

# ===============
  DebugPrint: 0x1ea3080c5bc5: [JS_OBJECT_TYPE]
# V8 为 point1 对象创建的隐藏类
  - map: 0x1ea308284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
  - prototype: 0x1ea308241395 <Object map = 0x1ea3082801c1>
  - elements: 0x1ea3080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
  - properties: 0x1ea3080406e9 <FixedArray[0]> {
  #x: 100 (const data field 0)
    #y: 200 (const data field 1)
}
0x1ea308284ce9: [Map]
  - type: JS_OBJECT_TYPE
  - instance size: 20
  - inobject properties: 2
  - elements kind: HOLEY_ELEMENTS
  - unused property fields: 0
  - enum length: invalid
  - stable_map
  - back pointer: 0x1ea308284cc1 <Map(HOLEY_ELEMENTS)>
  - prototype_validity cell: 0x1ea3081c0451 <Cell value= 1>
  - instance descriptors (own) #2: 0x1ea3080c5bf5 <DescriptorArray[2]>
  - prototype: 0x1ea308241395 <Object map = 0x1ea3082801c1>
  - constructor: 0x1ea3082413b1 <JSFunction Object (sfi = 0x1ea3081c557d)>
  - dependent code: 0x1ea3080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
  - construction counter: 0

# ===============
  DebugPrint: 0x1ea3080c5c1d: [JS_OBJECT_TYPE]
# V8 为 point2 对象创建的隐藏类
  - map: 0x1ea308284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
  - prototype: 0x1ea308241395 <Object map = 0x1ea3082801c1>
  - elements: 0x1ea3080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
  - properties: 0x1ea3080406e9 <FixedArray[0]> {
  #x: 200 (const data field 0)
    #y: 300 (const data field 1)
}
0x1ea308284ce9: [Map]
  - type: JS_OBJECT_TYPE
  - instance size: 20
  - inobject properties: 2
```

```
    - elements kind: HOLEY_ELEMENTS
    - unused property fields: 0
    - enum length: invalid
    - stable_map
    - back pointer: 0x1ea308284cc1 <Map(HOLEY_ELEMENTS)>
    - prototype_validity cell: 0x1ea3081c0451 <Cell value= 1>
    - instance descriptors (own) #2: 0x1ea3080c5bf5 <DescriptorArray[2]>
    - prototype: 0x1ea308241395 <Object map = 0x1ea3082801c1>
    - constructor: 0x1ea3082413b1 <JSFunction Object (sfi = 0x1ea3081c557d)>
    - dependent code: 0x1ea3080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
    - construction counter: 0

# ===============
  DebugPrint: 0x1ea3080c5c31: [JS_OBJECT_TYPE]
# V8 为 point3 对象创建的隐藏类
    - map: 0x1ea308284d39 <Map(HOLEY_ELEMENTS)> [FastProperties]
    - prototype: 0x1ea308241395 <Object map = 0x1ea3082801c1>
    - elements: 0x1ea3080406e9 <FixedArray[0]> [HOLEY_ELEMENTS]
    - properties: 0x1ea3080406e9 <FixedArray[0]> {
    #x: 100 (const data field 0)
      }
0x1ea308284d39: [Map]
    - type: JS_OBJECT_TYPE
    - instance size: 16
    - inobject properties: 1
    - elements kind: HOLEY_ELEMENTS
    - unused property fields: 0
    - enum length: invalid
    - stable_map
    - back pointer: 0x1ea308284d11 <Map(HOLEY_ELEMENTS)>
    - prototype_validity cell: 0x1ea3081c0451 <Cell value= 1>
    - instance descriptors (own) #1: 0x1ea3080c5c41 <DescriptorArray[1]>
    - prototype: 0x1ea308241395 <Object map = 0x1ea3082801c1>
    - constructor: 0x1ea3082413b1 <JSFunction Object (sfi = 0x1ea3081c557d)>
   - dependent code: 0x1ea3080401ed <Other heap object (WEAK_FIXED_ARRAY_TYPE)>
   - construction counter: 0
```

多个对象共用一个隐藏

在 V8 中，**每个对象都有一个 map 属性**，该属性值指向该对象的隐藏类。不过**如果两个对象的形状是相同的，V8 就会为其复用同一个隐藏类**，这样有两个好处：

- 减少隐藏类的创建次数，也间接加速了代码的执行速度；
- 减少了隐藏类的存储空间。

那么，什么情况下两个对象的形状是相同的，要满足以下两点：

- 相同的属性名称；
- 相等的属性个数。

重新构建隐藏类

给一个对象添加新的属性，删除新的属性，或者改变某个属性的数据类型都会改变这个对象的形状，那么势必也就会触发 V8 为改变形状后的对象重建新的隐藏类。

```
// test.js
let point = {};
%DebugPrint(point);
point.x = 100;
%DebugPrint(point);
point.y = 200;
%DebugPrint(point);

# ./d8 --allow-natives-syntax ./test.js
DebugPrint: 0x32c7080c5b2d: [JS_OBJECT_TYPE]
 - map: 0x32c7082802d9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 ...

DebugPrint: 0x32c7080c5b2d: [JS_OBJECT_TYPE]
 - map: 0x32c708284cc1 <Map(HOLEY_ELEMENTS)> [FastProperties]
 ...

DebugPrint: 0x32c7080c5b2d: [JS_OBJECT_TYPE]
 - map: 0x32c708284ce9 <Map(HOLEY_ELEMENTS)> [FastProperties]
 ...
```

每次给对象添加了一个新属性之后，该对象的隐藏类的地址都会改变，这也就意味着隐藏类也随着改变了；如果删除对象的某个属性，那么对象的形状也就随着发生了改变，这时 V8 也会重建该对象的隐藏类；

最佳实践

- 使用字面量初始化对象时，要保证属性的顺序是一致的；
- 尽量使用字面量一次性初始化完整对象属性；
- 尽量避免使用 delete 方法。

通过内联缓存来提升函数执行效率

虽然隐藏类能够加速查找对象的速度，但是在 V8 查找对象属性值的过程中，依然有查找对象的隐藏类和根据隐藏类来查找对象属性值的过程。如果一个函数中利用了对象的属性，并且这个函数会被多次执行

```
Plain Text | 复制代码
```

```
function loadX(obj) {
  return obj.x;
}
var obj = { x: 1, y: 3 };
var obj1 = { x: 3, y: 6 };
var obj2 = { x: 3, y: 6, z: 8 };
for (var i = 0; i < 100; i++) {
  // 对比时间差异
  console.time(`---${i}----`)
  loadX(obj);
  console.timeEnd(`---${i}----`)
  loadX(obj1);
  // 产生多态
  loadX(obj2);
}
```

通常 V8 获取 obj.x 的流程：

- 找对象 obj 的隐藏类；
- 再通过隐藏类查找 x 属性偏移量；
- 然后根据偏移量获取属性值，在这段代码中 loadX 函数会被反复执行，那么获取 obj.x 的流程也需要反复被执行；

内联缓存及其原理：

- 函数 loadX 在一个 for 循环里面被重复执行了很多次，因此 V8 会想尽一切办法来压缩这个查找过程，以提升对象的查找效率。这个加速函数执行的策略就是**内联缓存 (Inline Cache)**，简称为 IC；
- **IC 的原理**：在 V8 执行函数的过程中，会观察函数中一些**调用点 (CallSite)** 上的关键中间数据，然后将这些数据缓存起来，当下次再次执行该函数的时候，V8 就可以直接利用这些中间数据，节省了再次获取这些数据的过程，因此 V8 利用 IC，可以有效提升一些重复代码的执行效率。
- IC 会为每个函数维护一个**反馈向量 (FeedBack Vector)**，反馈向量记录了函数在执行过程中的一些关键的中间数据。
- 反馈向量其实就是一个表结构，它由很多项组成的，每一项称为一个插槽 (Slot)，V8 会依次将执行 loadX 函数的中间数据写入到反馈向量的插槽中。
- 当 V8 再次调用 loadX 函数时，比如执行到 loadX 函数中的 return obj.x 语句时，它就会在对应的插槽中查找 x 属性的偏移量，之后 V8 就能直接去内存中获取 obj.x 的属性值了。这样就大大提升了 V8 的执行效率。

单态、多态和超态：

- 如果一个插槽中只包含 1 个隐藏类，那么我们称这种状态为单态 (monomorphic)；

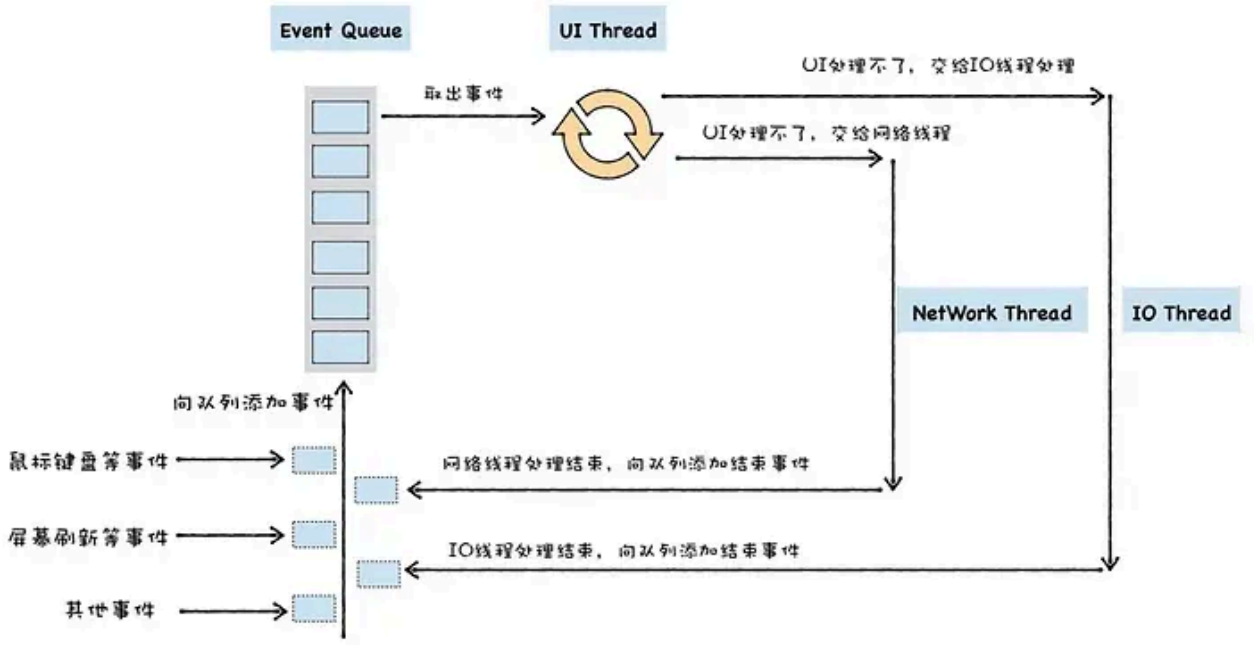
- 如果一个插槽中包含了 2 ～ 4 个隐藏类，那我们称这种状态为多态 (polymorphic)；
- 如果一个插槽中超过 4 个隐藏类，那我们称这种状态为超态 (magamorphic)。
- 单态的性能优于多态和超态，所以我们需要稍微避免多态和超态的情况。要避免多态和超态，那么就尽量默认所有的对象属性是不变的，比如你写了一个 loadX(obj) 的函数，那么当传递参数时，尽量不要使用多个不同形状的 obj 对象。

V8 引入了内联缓存（IC），**IC 会监听每个函数的执行过程，并在一些关键的地方埋下监听点**，这些包括了加载对象属性 (Load)、给对象属性赋值 (Store)、还有函数调用 (Call)，V8 会将监听到的数据写入一个称为反馈向量 (FeedBack Vector) 的结构中，同时 V8 会为每个执行的函数维护一个反馈向量。有了反馈向量缓存的临时数据，V8 就可以缩短对象属性的查找路径，从而提升执行效率。但是针对函数中的同一段代码，如果对象的隐藏类是不同的，那么反馈向量也会记录这些不同的隐藏类，这就出现了多态和超态的情况。**我们在实际项目中，要尽量避免出现多态或者超态的情况**。

9.11、异步编程与消息队列

1、V8 是如何执行回调函数的?

回调函数有两种类型：同步回调和异步回调，同步回调函数是在执行函数内部被执行的，而异步回调函数是在执行函数外部被执行的。通用 UI 线程宏观架构：



UI 线程提供一个**消息队列**，并将待执行的事件添加到消息队列中，然后 UI 线程会不断循环地从消息队列中取出事件、执行事件。关于异步回调，这里也有两种不同的类型，其典型代表是 setTimeout 和 XMLHttpRequest：

- setTimeout 的执行流程其实是比较简单的，在 setTimeout 函数内部封装回调消息，并将回调消息添加进消息队列，然后主线程从消息队列中取出回调事件，并执行回调函数。

- XMLHttpRequest 稍微复杂一点，因为下载过程需要放到单独的一个线程中去执行，所以执行 XMLHttpRequest.send 的时候，宿主会将实际请求转发给网络线程，然后 send 函数退出，主线程继续执行下面的任务。网络线程在执行下载的过程中，会将一些中间信息和回调函数封装成新的消息，并将其添加进消息队列中，然后主线程从消息队列中取出回调事件，并执行回调函数。

2、宏任务和微任务

「调用栈」：调用栈是一种数据结构，用来管理在主线程上执行的函数的调用关系。主线程在执行任务的过程中，如果函数的调用层次过深，可能造成栈溢出的错误，我们**可以使用 setTimeout 来解决栈溢出的问题**。setTimeout 的本质是将同步函数调用改成异步函数调用，这里的异步调用是将回调函数封装成宏任务，并将其添加进消息队列中，然后主线程再按照一定规则循环地从消息队列中读取下一个宏任务。

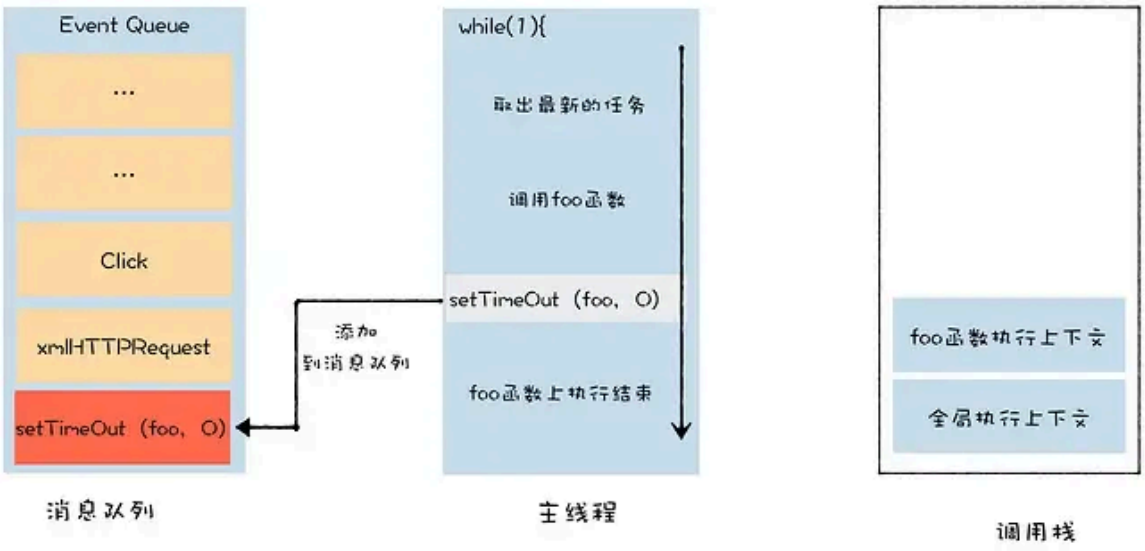
「宏任务」：就是指消息队列中的等待被主线程执行的事件。每个宏任务在执行时，V8 都会重新创建栈，然后随着宏任务中函数调用，栈也随之变化，最终，当该宏任务执行结束时，整个栈又会被清空，接着主线程继续执行下一个宏任务。

「微任务」：你可以把微任务看成是一个需要异步执行的函数，执行时机是在主函数执行结束之后、当前宏任务结束之前。

JavaScript 中之所以要引入微任务，主要是由于主线程执行消息队列中宏任务的时间颗粒度太粗了，无法胜任一些对精度和实时性要求较高的场景，微任务可以在实时性和效率之间做一个有效的权衡。另外使用微任务，可以改变我们现在的异步编程模型，使得我们可以使用同步形式的代码来编写异步调用。

微任务是基于消息队列、事件循环、UI 主线程还有堆栈而来的，然后基于微任务，又可以延伸出协程、Promise、Generator、await/async 等现代前端经常使用的一些技术。

```
// 不会使浏览器卡死
function foo() {
  setTimeout(foo, 0);
}
foo();
```



微任务

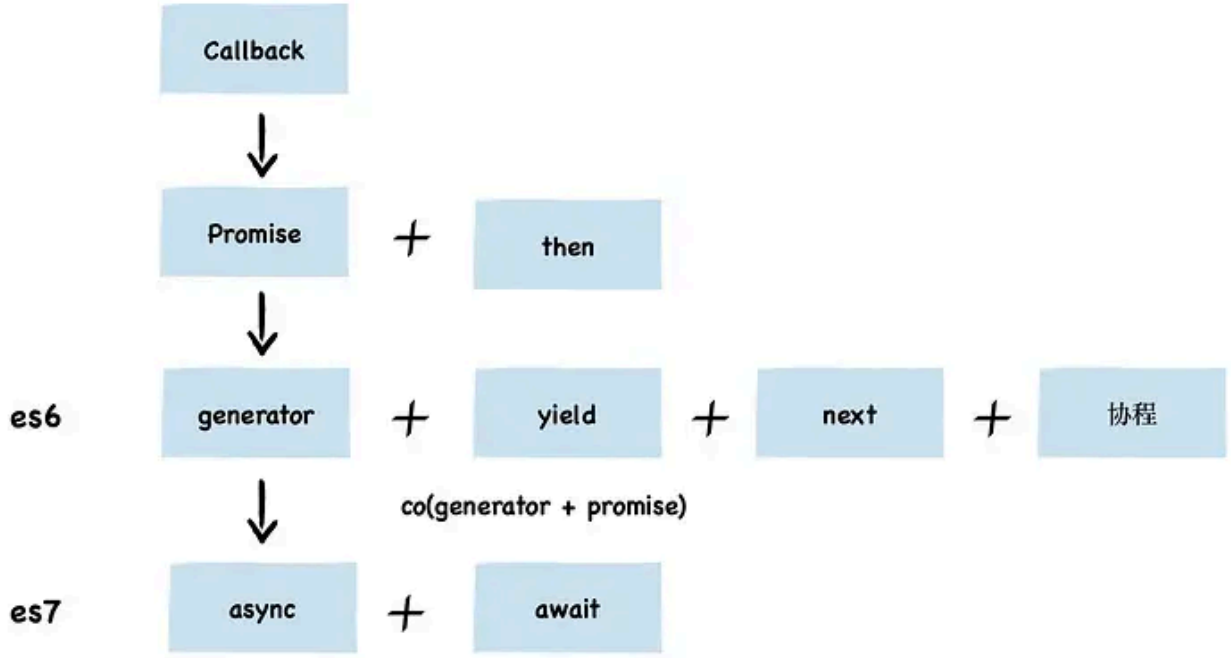
```
// 浏览器console控制台可使浏览器卡死（无法响应鼠标事件等）
function foo() {
  return Promise.resolve().then(foo);
}
foo();
```

如果当前的任务中产生了一个微任务，通过 Promise.resolve() 或者 Promise.reject() 都会触发微任务，触发的微任务不会在当前的函数中被执行，所以**执行微任务时，不会导致栈的无限扩张**

和异步调用不同，微任务依然会在当前任务执行结束之前被执行，这也就意味着**在当前微任务执行结束之前，消息队列中的其他任务是不可能被执行的**。因此在函数内部触发的微任务，一定比在函数内部触发的宏任务要优先执行。

微任务依然是在当前的任务中执行的，所以如果在微任务中循环触发新的微任务，那么将导致消息队列中的其他任务没有机会被执行。

3、前端异步编程方案史



Callback 模式的异步编程模型需要实现大量的回调函数，大量的回调函数会打乱代码的正常逻辑，使得代码变得不线性、不易阅读，这就是我们所说的**回调地狱问题**。

Promise 能很好地解决回调地狱的问题，我们可以按照线性的思路来编写代码，这个过程是线性的，非常符合人的直觉。

但是这种方式**充满了 Promise 的 then() 方法**，如果处理流程比较复杂的话，那么整段代码将**充斥着大量的 then，语义化不明显，代码不能很好地表示执行流程**。我们想要通过**线性的方式**来编写异步代码，要实现这个理想，**最关键的是要能实现函数暂停和恢复执行的功能**。而**生成器**就可以实现函数暂停和恢复，我们可以在生成器中使用同步代码的逻辑来异步代码 (实现该逻辑的核心是协程)。

但是在生成器之外，我们还需要一个**触发器**来驱动生成器的执行。前端的最终方案就是 async/await，async 是一个可以暂停和恢复执行的函数，在 async 函数内部使用 await 来暂停 async 函数的执行，await 等待的是一个 Promise 对象，如果 Promise 的状态变成 resolve 或者 reject，那么 async 函数会恢复执行。因此，使用 async/await 可以实现以同步的方式编写异步代码这一目标。和生成器函数一样，使用了 async 声明的函数在执行时，也是一个单独的协程，我们可以使用 await 来暂停该协程，由于 await 等待的是一个 Promise 对象，我们可以 resolve 来恢复该协程

协程 是一种比线程更加轻量级的存在。你可以把协程看成是跑在线程上的任务，一个线程上可以存在多个协程，但是在线程上同时只能执行一个协程。比如，当前执行的是 A 协程，要启动 B 协程，那么 A 协程就需要将主线程的控制权交给 B 协程，这就体现在 A 协程暂停执行，B 协程恢复执行；同样，也可以从 B 协程中启动 A 协程。通常，如果从 A 协程启动 B 协程，我们就把 A 协程称为 B 协程的父协程。

正如一个进程可以拥有多个线程一样，一个线程也可以拥有多个协程。每一时刻，该线程只能执行其中某一个协程。最重要的是，**协程不是被操作系统内核所管理，而完全是由程序所控制（也**

就是在用户态执行）。这样带来的好处就是性能得到了很大的提升，不会像线程切换那样消耗资源。参考：co 函数库的含义和用法 <https://link.zhihu.com/?target=https%3A//link.segmentfault.com/%3Fenc%3DIzeS4%252FN551QfBQvX883Anw%253D%253D.4u404WKlSK1qbNyypopPIBUD%252BLB2YnBWwBCmWMcqLLGSLQGideEI7uJeHJVnvlty>

9.12、垃圾回收

1、垃圾数据

从 **GC Roots** 对象出发，遍历 GC Root 中的所有对象，如果通过 GC Roots 没有遍历到的对象，则这些对象便是垃圾数据。V8 会有专门的垃圾回收器来回收这些垃圾数据。

2、垃圾回收算法

垃圾回收大致可以分为以下几个步骤：

- 第一步，**通过 GC Root 标记空间中活动对象和非活动对象**。目前 V8 采用的**可访问性（reachability）算法**来判断堆中的对象是否是活动对象。具体地讲，这个算法是将一些 GC Root 作为初始存活的对象的集合，从 GC Roots 对象出发，遍历 GC Root 中的所有对象：
  - 通过 GC Root 遍历到的对象，我们就认为该对象是**可访问的（reachable）**，那么必须保证这些对象应该在内存中保留，我们也称可访问的对象为**活动对象**；
  - 通过 GC Roots 没有遍历到的对象，则是**不可访问的（unreachable）**，那么这些不可访问的对象就可能被回收，我们称不可访问的对象为**非活动对象**。
  - 在**浏览器环境中，GC Root 有很多**，通常包括了以下几种 (但是不止于这几种)：
    - 全局的 window 对象（位于每个 iframe 中）；
    - 文档 DOM 树，由可以通过遍历文档到达的所有原生 DOM 节点组成；
    - 存放栈上变量。
- 第二步，**回收非活动对象所占据的内存**。其实就是在所有的标记完成之后，统一清理内存中所有被标记为可回收的对象。
- 第三步，**做内存整理**。一般来说，频繁回收对象后，内存中就会存在大量不连续空间，我们把这些不连续的内存空间称为**内存碎片**。当内存中出现了大量的内存碎片之后，如果需要分配较大的连续内存时，就有可能出现内存不足的情况，所以最后一步需要整理这些内存碎片。但这步其实是可选的，因为**有的垃圾回收器不会产生内存碎片(比如副垃圾回收器)**。

3、垃圾回收

V8 依据**代际假说**，将堆内存划分为**新生代和老生代**两个区域，新生代中存放的是生存时间短的对象，老生代中存放生存时间久的对象。代际假说有两个特点：

- 第一个是大部分对象都是“**朝生夕死**”的，也就是说**大部分对象在内存中存活的时间很短**，比如函数内部声明的变量，或者块级作用域中的变量，当函数或者代码块执行结束

时，作用域中定义的变量就会被销毁。因此这一类对象一经分配内存，很快就变得不可访问；

- 第二个是**不死的对象，会活得更久**，比如全局的 window、DOM、Web API 等对象。

为了提升垃圾回收的效率，V8 设置了两个垃圾回收器，主垃圾回收器和副垃圾回收器。

- **主垃圾回收器**负责收集老生代中的垃圾数据，**副垃圾回收器**负责收集新生代中的垃圾数据。
- **副垃圾回收器采用了 Scavenge 算法**，是把新生代空间对半划分为两个区域（有些地方也称作From和To空间），一半是对象区域，一半是空闲区域。新的数据都分配在对象区域，等待对象区域快分配满的时候，垃圾回收器便执行垃圾回收操作，之后将存活的对象从对象区域拷贝到空闲区域，并将两个区域互换。
  - 这种角色翻转的操作还能让新生代中的这两块区域无限重复使用下去。
  - 副垃圾回收器每次执行清理操作时，都需要将存活的对象从对象区域复制到空闲区域，复制操作需要时间成本，如果新生区空间设置得太大了，那么每次清理的时间就会过久，所以为了执行效率，一般**新生区的空间会被设置得比较小**。
  - 副垃圾回收器还会采用**对象晋升策略**，也就是移动那些经过两次垃圾回收依然还存活的对象到老生代中。
- 主垃圾回收器回收器主要负责**老生代中的垃圾数据的回收操作，会经历标记、清除和整理过程**。
  - 主垃圾回收器主要负责老生代中的垃圾回收。除了新生代中晋升的对象，一些大的对象会直接被分配到老生代里。
  - 老生代中的对象有两个特点：一个是对象占用空间大；另一个是对象存活时间长。

4、Stop-The-World

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）**。

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。

10、机器码、字节码

1、字节码

早期的 V8 为了提升代码的**执行速度**，直接将 JavaScript 源代码编译成了**没有优化的二进制机器代码**，如果某一段二进制代码执行频率过高，那么 V8 会将其标记为**热点代码**，热点代码会被优化编译器优化，优化后的机器代码执行效率更高。

随着移动设备的普及，V8 团队逐渐发现将 JavaScript 源码直接编译成二进制代码存在两个致命的问题：

1. **时间问题**：编译时间过久，影响代码启动速度；
2. **空间问题**：缓存编译后的二进制代码占用更多的内存。

这两个问题无疑会阻碍 V8 在移动设备上的普及，于是 V8 团队大规模重构代码，引入了中间的字节码。字节码的优势有如下三点：

- **解决启动问题**：生成字节码的时间很短；
- **解决空间问题**：字节码虽然占用的空间比原始的 JavaScript 多，但是相较于机器代码，字节码还是小了太多，缓存字节码会大大降低内存的使用。
- **代码架构清晰**：采用字节码，可以简化程序的复杂度，使得 V8 移植到不同的 CPU 架构平台更加容易。

如何查看字节码

Plain Text | 复制代码

```
// test.js
function add(x, y) {
  var z = x + y;
  return z;
}
console.log(add(1, 2));
```

运行 `./d8 ./test.js --print-bytecode`：

```
[generated bytecode for function: add (0x01000824fe59 <SharedFunctionInfo add>)]
Parameter count 3 #三个参数，包括了显式地传入的 x 和 y，还有一个隐式地传入的 this
Register count 1
Frame size 8
         0x10008250026 @    0 : 25 02             Ldar a1 #将a1寄存器中的值加载到
         0x10008250028 @    2 : 34 03 00          Add a0, [0]
         0x1000825002b @    5 : 26 fb             Star r0 #Store Accumulator to
         0x1000825002d @    7 : aa                Return  #结束当前函数的执行，并
Constant pool (size = 0)
Handler Table (size = 0)
Source Position Table (size = 0)
3
```

常用字节码指令：

- Ldar：表示将寄存器中的值加载到累加器中，你可以把它理解为 LoaD Accumulator from Register，就是把某个寄存器中的值，加载到累加器中。
- Star：表示 Store Accumulator Register， 你可以把它理解为 Store Accumulator to Register，就是把累加器中的值保存到某个寄存器中
- Add： `Add a0, [0]` 是从 a0 寄存器加载值并将其与累加器中的值相加，然后将结果再次放入累加器。
  add a0 后面的[0]称之为 feedback vector slot，又叫**反馈向量槽**，它是一个数组，解释器将解释执行过程中的一些数据类型的分析信息都保存在这个反馈向量槽中了，目的是为了给 TurboFan 优化编译器提供优化信息，很多字节码都会为反馈向量槽提供运行时信息。
- LdaSmi：将小整数（Smi）加载到累加器寄存器中
- Return：结束当前函数的执行，并将控制权传回给调用方。返回的值是累加器中的值。

2、机器码（Bytecode）

某种程度上就是汇编语言，只是它没有对应特定的 CPU，或者说它对应的是虚拟的 CPU。这样的话，生成 Bytecode 时简单很多，无需为不同的 CPU 生产不同的代码。要知道，V8 支持 9 种不同的 CPU，引入一个中间层 Bytecode，可以简化 V8 的编译流程，提高可扩展性。

如果我们在不同硬件上去生成 Bytecode，会发现生成代码的指令是一样的。

11、13 个 JavaScript 性能提升技巧

Daniel Clifford 在Google I/O2012 上做了一个精彩的演讲“**Breaking the JavaScript Speed Limit with V8**”。在演讲中，他深入解释了 13 个简单的代码优化方法，可以让你的JavaScript 代码在 Chrome V8 引擎编译/运行时更加快速。在演讲中，他介绍了怎么优化，并解释了原因。下面简明的列出了**13 个 JavaScript 性能提升技巧**:

1. 在构造函数里初始化所有对象的成员(所以这些实例之后不会改变其隐藏类)；
2. 总是以相同的次序初始化对象成员；
3. 尽量使用可以用 31 位有符号整数表示的数；
4. 为数组使用从 0 开始的连续的主键；
5. 别预分配大数组(比如大于 64K 个元素)到其最大尺寸，令其尺寸顺其自然发展就好；
6. 别删除数组里的元素，尤其是数字数组；
7. 别加载未初始化或已删除的元素；
8. 对于固定大小的数组，使用”array literals“初始化（初始化小额定长数组时，用字面量进行初始化）；
9. 小数组(小于 64k)在使用之前先预分配正确的尺寸；
10. 请勿在数字数组中存放非数字的值(对象)；
11. 尽量使用单一类型（monomorphic）而不是多类型（polymorphic）（如果通过非字面量进行初始化小数组时，切勿触发类型的重新转换）；
12. 不要使用 `try{} catch{}` （如果存在 `try/catch` 代码快，则将性能敏感的代码放到一个嵌套的函数中）；
13. 在优化后避免在方法中修改隐藏类。

演

12、在 V8 引擎里 5 个优化代码的技巧

1. **对象属性的顺序**: 在实例化你的对象属性的时候一定要使用相同的顺序，这样隐藏类和随后的优化代码才能共享；

2．**动态属性**: 在对象实例化之后再添加属性会强制使得隐藏类变化，并且会减慢为旧隐藏类所优化的代码的执行。所以，要在对象的构造函数中完成所有属性的分配；

3．**方法**: 重复执行相同的方法会运行的比不同的方法只执行一次要快 (因为内联缓存)；

4．**数组**: 避免使用 keys 不是递增的数字的稀疏数组，这种 key 值不是递增数字的稀疏数组其实是一个 hash 表。在这种数组中每一个元素的获取都是昂贵的代价。同时，要避免提前申请大数组。最好的做法是随着你的需要慢慢的增大数组。最后，不要删除数组中的元素，因为这会使得 keys 变得稀疏；

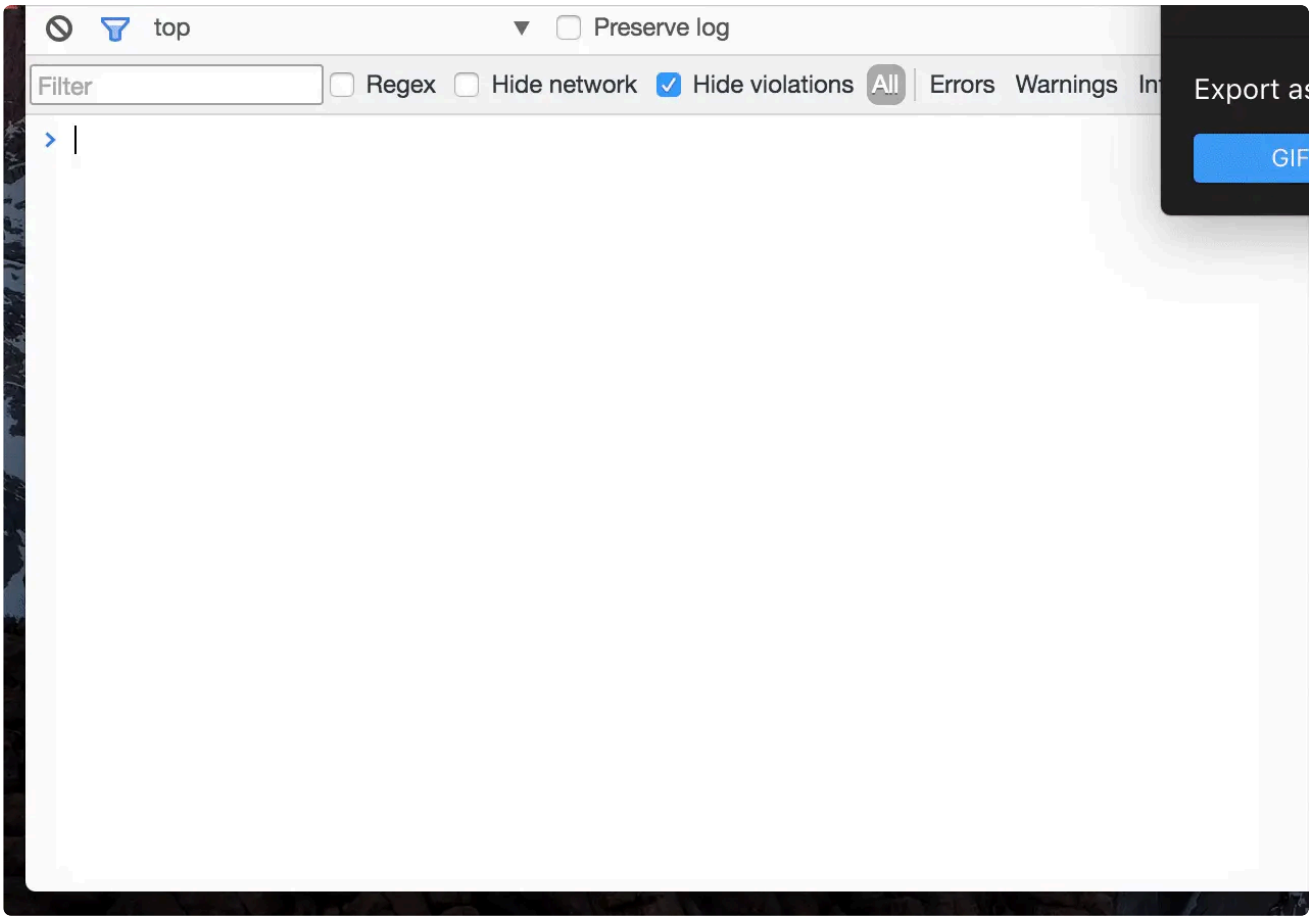
5．**标记值 (Tagged values)**: V8 用 32 位来表示对象和数字。它使用一位来区分它是对象 (flag = 1) 还是一个整型 (flag = 0)，也被叫做小整型(SMI)，因为它只有 31 位。然后，如果一个数值大于 31 位，V8 将会对其进行 **box 操作**，然后将其转换成 double 型，并且创建一个新的对象来装这个数。所以，为了避免代价很高的 box 操作，尽量使用 31 位的有符号数。

资

13、JavaScript 启动性能瓶颈分析与解决方案

Chrome插件Console Importer推荐：Easily import JS and CSS resources from Chrome console. （可以在浏览器控制台安装 loadsh、moment、jQuery 等库，在控制台直接验证、使用这些库。）

效果图：



69e6d7ae093f.webp&title=%E8%AF%A6%E8%A7%A3%20Chrome%20V8%20%E5%BC%95%E6%93%