xuanbiyijue.com

深入理解V8引擎

悬笔345E绝

48-60 minutes

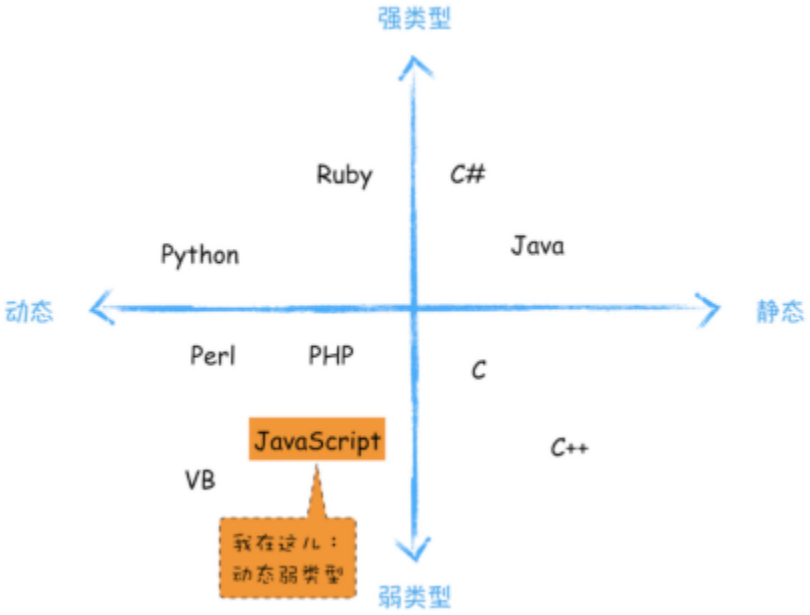
大纲：
js语言和内存空间；
V8基本介绍；
V8解释器执行代码的流程；
V8做的性能优化措施；
V8垃圾回收；
JS语法和数据结构在V8中的实现；

一.预备知识：JS语言和内存空间

1.静态语言就是强类型?

静态、动态语言，强类型、弱类型语言的关系
（1）静态语言，使用前要确定数据类型
（2）动态语言，运行中检查数据类型；
（3）弱类型语言，支持隐式类型转换
（4）强类型，不支持 隐式转换；

C是静态语言，但是它支持隐式转换，是弱类型；



2.JS的内存空间

（1）分为三类，代码空间，栈stack空间，堆heap空间；

为什么不放在一起？因为放在一起，会影响执行上下文切换和执行效率；

（2）堆(heap)和栈(stack)的区别:

1）栈:

空间较小；先进后出；动态分配的空间 一般由程序员分配释放， 若程序员不释放，程序结束时可能由OS回收，分配方式类似于链表。

2）堆:

空间较大；队列优先,先进先出；由操作系统自动分配释放 ，存放函数的参数值，局部变量的值等。

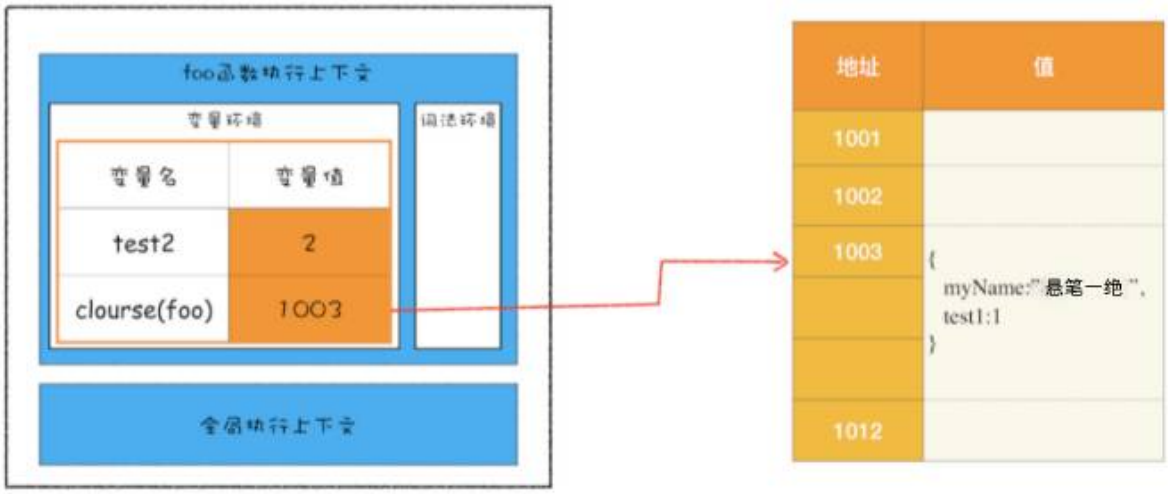
（3）原始类型的数据值都是直接保存在“栈”中的，引用

类型的值是存放在“堆”中?
其实并没有那么简单，在V8源码中
(1)字符串： 存在堆里，栈中为引用地址 ， 如果存在相同字符串，则引用地址相同。
(2)数字： 小整数存在栈中 ， 其他类型存在堆中。
(3)其他类型：引擎初始化时分配唯一地址，栈中的变量存的是唯一的引用。
（4）闭包的内存模型，Closure
比如下面这题，foo中有个闭包对象，有两个属性值myName和test1，存到 堆 中；

```
1  function foo() {
2      var myName = "悬笔一绝"
3      let test1 = 1
4      const test2 = 2
5      var innerBar = {
6          setName:function(newName){ myName =
7  newName },
8          getName:function(){ console.log(test1);
9  return myName }
10     }
11     return innerBar
12 }
13 var bar = foo()
14 bar.setName("悬笔")
```

```
bar.getName()
console.log(bar.getName())
```

执行到foo 函数中“return innerBar” 的调用栈情况~



二.V8基本概念

1.浏览器内核、渲染引擎和JS引擎的区别

（1）浏览器内核又可以分成两部分：渲染引擎(layout engineer或者RenderingEngine)和JS引擎。

（2）渲染引擎：

负责取得网页的内容（HTML、XML、图像等等）、整理讯息（例如加入CSS等），以及计算网页的显示方式，然后会输出至显示器或打印机。浏览器的内核的不同对于网页的语法解释会有不同，所以渲染的效果也不相同。所有网页浏览器、电子邮件客户端以及其它需要编辑、显示网络内容的应用程序都需要内核。

种类：

Tridend：IT浏览器
Geoko：Firefox浏览器
Webkit：safari、andriod、chrome（后来使用Blink渲染引擎）
（3）JS引擎：
解析和执行javascript来实现网页的动态效果。
（4）区别
最开始渲染引擎和JS引擎并没有区分的很明确，后来JS引擎越来越独立，内核就倾向于只指渲染引擎。
渲染引擎使用JS引擎的接口来处理逻辑代码并获取结果。
JS引擎通过桥接接口访问渲染引擎中的DOM及CSSOM

2.主流 JS 引擎

- V8 (Google)
- SpiderMonkey (Mozilla)
- JavaScriptCore (Apple)
  JavaScript Core 引擎是WebKit中默认的JavaScript引擎，也是苹果开源的一个项
  目，应用较为广泛。最初，性能不是很好，从2008年开始了一系列的优化，重新实
  现了编译器和字节码解释器，使得引擎的性能有较大的提升。随后内嵌缓存、基于正
  则表达式的JIT、简单的JIT及字节码解释器等技术引入

进来，JavaScriptCore引擎也
在不断的迭代和发展。
JavaScriptCore与V8有一些不同之处，其中最大的不同就是新增了字节码的中间表
示，并加入了多层JIT编译器（如：简单JIT编译器、DFG JIT编译器、LLVM等）优化
性能，不停的对本地代码进行优化。

- Chakra (Microsoft)
- duktape(IOT)
- JerryScript(IOT)
- QuickJS
- Hermes(Facebook-React Native)

3.V8基本介绍

V8是一个由Google开源的采用C++编写的高性能JavaScript和WebAssembly引擎，应用在 Chrome和Node.js等中。它实现了ECMAScript和WebAssembly，运行在Windows 7及以上、macOS 10.12+以及使用x64、IA-32、ARM或MIPS处理器的Linux系统上。V8可以独立运行，也可以嵌入到任何C++应用程序中。

（1）V8由来

V8最初是由Lars Bak团队开发的，以汽车的V8发动机（有八个气缸的V型发动机）进行命名，预示着这将是一款性能极高的JavaScript引擎，在2008年9月2号同chrome一同开源发布。

（2）为什么需要V8

我们写的JavaScript代码最终是要在机器中被执行的，但机器无法直接识别这些高级语言。需要经过一系列的处理，将高级语言转换成机器可以识别的的指令，也就是二进制码，交给机器执行。这中间的转换过程就是V8的具体工作。

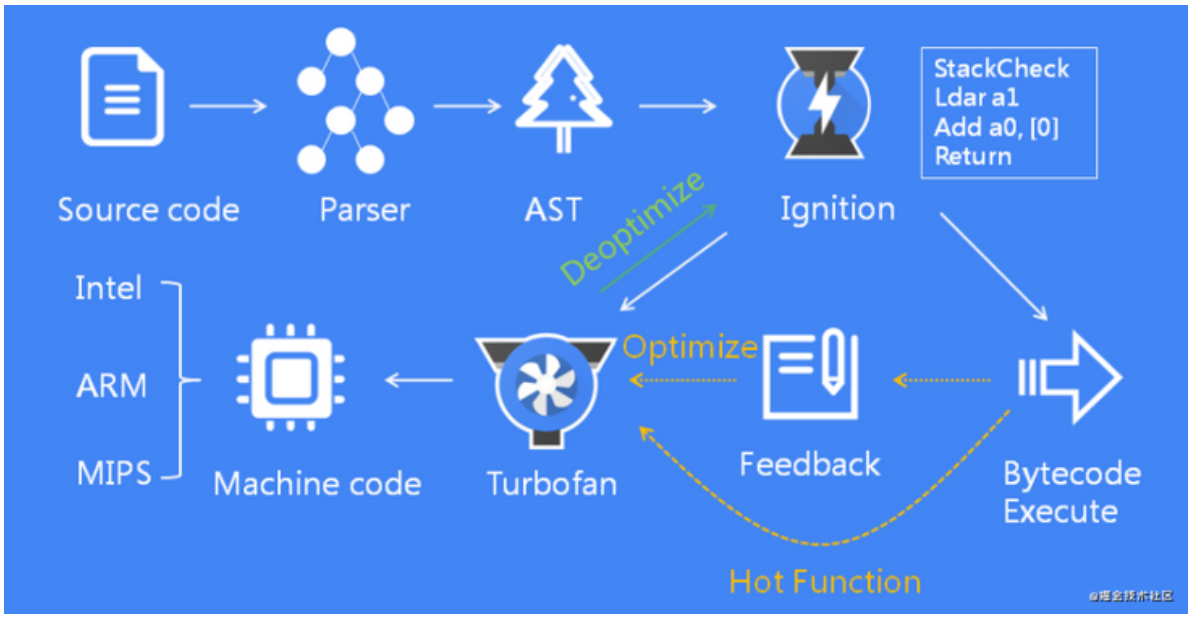
（3）V8组成

首先来看一下V8的内部组成。V8的内部有很多模块，其中最重要的4个如下：

- Parser: 解析器，负责将源代码解析成AST
- Ignition: 解释器，负责将AST转换成字节码并执行，同时会标记热点代码
- TurboFan: 编译器，负责将热点代码编译成机器码并执行
- Orinoco: 垃圾回收器，负责进行内存空间回收

（4）V8工作流程

以下是V8中几个重要模块的具体工作流程图。下面详细分析。



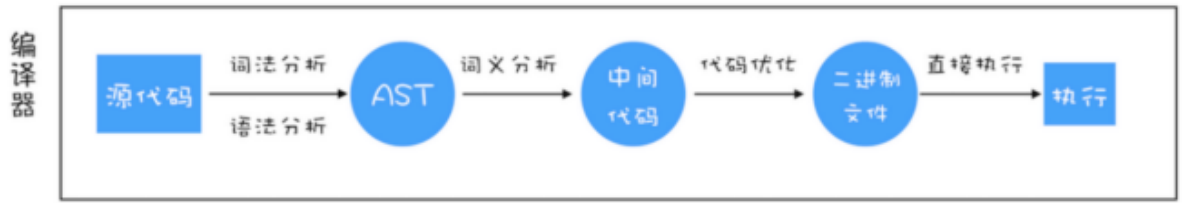
三.V8解释执行JS代码的工作流程

1.编译器Compiler和解释器Interpreter

按语言的执行流程，可以把语言划分为编译型语言和解释型语言。

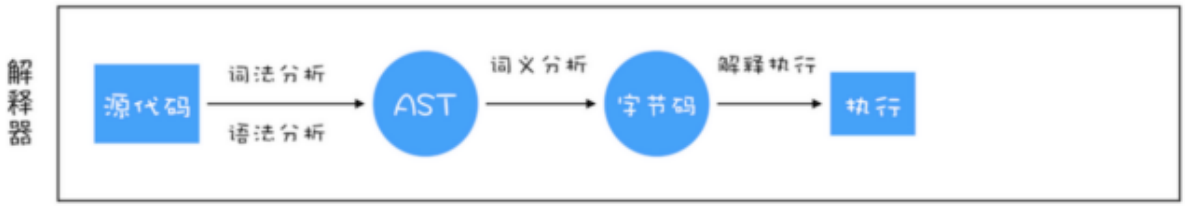
1-1.编译型语言

在程序执行之前，需要经过编译器的编译过程，并且编译之后会直接保留机器能读懂的二进制文件，这样每次运行程序时，都可以直接运行该二进制文件，而不需要再次重新编译了。比如 C/C++、GO 等都是编译型语言。

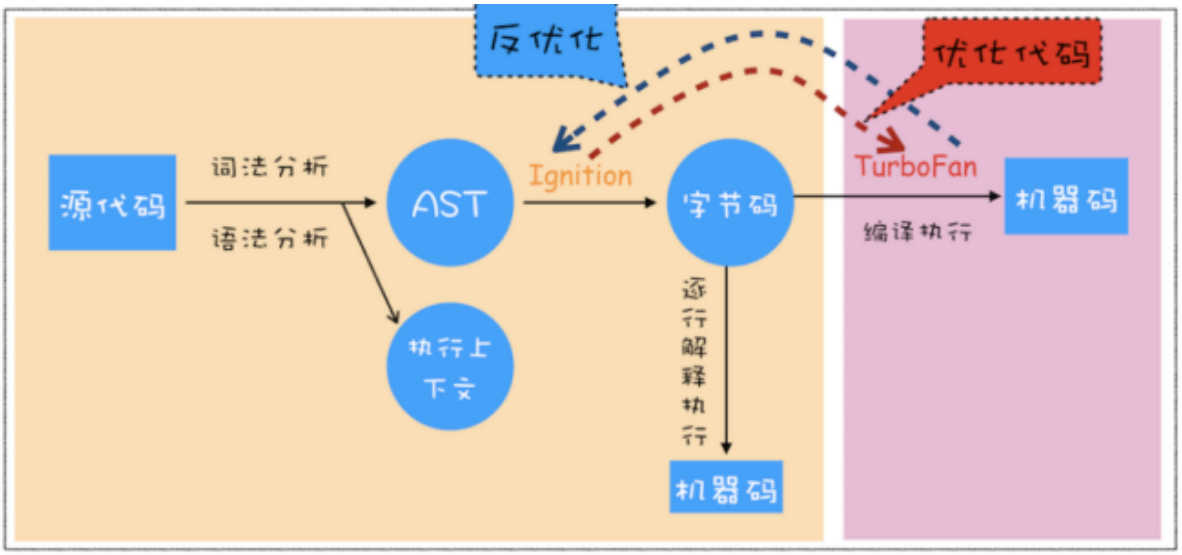


1-2.解释型语言

在每次运行时都需要通过解释器对程序进行动态解释和执行。比如 Python、JavaScript 等都属于解释型语言。



2.V8执行一段代码流程图



2-1.生成抽象语法树（AST）和执行上下文

2-1-1.AST

（1）编译器或解释器，他们不理解高级语言，只可以理解AST；

（2）可以把 AST 看成代码的结构化表示；

在线AST网站 https://resources.jointjs.com/demos/javascript-ast

（3）AST应用非常广泛

1）Babel：

Babel 的工作原理就是先将 ES6 源码转换为 AST，然后再将 ES6 语法的 AST 转换为 ES5 语法的 AST，最后利用 ES5 的 AST 生成 JavaScript 源代码；

2）ESLint：利用 AST 来检查代码规范化的问题。等等；

（4）AST的基本概念和内容

AST节点主要有 标识符 Identifer、各种字面量 xxLiteral、各种语句 xxStatement，各种声明语句 xxDeclaration，各种表达式 xxExpression，以及 Class、Modules、File、Program、Directive、Comment等等；

不过多展开，详细可以参考这一篇：https://mp.weixin.qq.com/s/v7T-7h3TPPnnooJDOFsnSQ

2-1-2.AST生成过程

Parser解析器负责将源代码转换成抽象语法树AST。在

转换过程中有两个重要的阶段：词法分析（Lexical Analysis）和语法分析（Syntax Analysis）。

（1）词法分析

也称为分词tokenize，是将字符串形式的代码转换为标记（token）序列的过程。
这里的token是一个字符串，是构成源代码的最小单位，类似于英语中单词，词法分析也可以理解成 将英文字母组合成单词的过程 。
词法分析过程中不会关心单词之间的关系。比如：词法分析过程中能够将括号标记成token，但并不会校验括号是否匹配。
JavaScript中的token主要包含以下几种：

1	关键字：var、let、const等
2	标识符：没有被引号括起来的连续字符，可能是一
3	个变量，也可能是 if、else 这些关键字，又或者是
4	true、false 这些内置常量
5	运算符：+、-、 *、/ 等
6	数字：像十六进制，十进制，八进制以及科学表达
7	式等
8	字符串：变量的值等
	空格：连续的空格，换行，缩进等
	注释：行注释或块注释都是一个不可拆分的最小语

	法单元 标点：大括号、小括号、分号、冒号等

以下是const a = ‘hello world’经过esprima词法分析后生成的tokens。

```
[
  {
    type: 'Keyword',
    value: 'const',
  },
  {
    type: 'Identifier',
    value: 'a',
  },
  {
    type: 'Punctuator',
    value: '=',
  },
  {
    type: 'String',
    value: '"hello world"',
  },
];
```

（2）语法分析,也叫解释parse

语法分析是将词法分析产生的token按照某种给定的形式文法转换成AST的过程。也就是 把单词组合成句子的过程 。

如果源码符合语法规则，这一步就会顺利完成，但如果源码存在语法错误，这一步就会终止，并抛出一个“语法错误”。

上述const a = ‘hello world’经过语法分析后生成的AST如下：

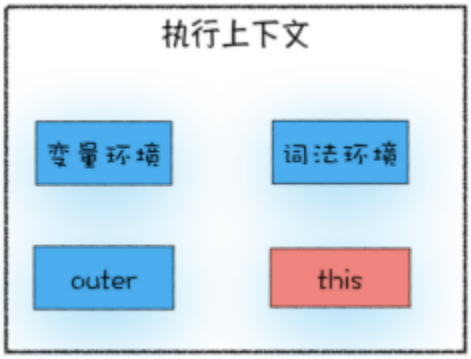
```
{
    "type": "Program",
    "body": [
        {
            "type": "VariableDeclaration",
            "declarations": [
                {
                    "type": "VariableDeclarator",
                    "id": {
                        "type": "Identifier",
                        "name": "a"
                    },
                    "init": {
                        "type": "Literal",
```

```
15                      "value": "hello world",
16                      "raw": "'hello world'"
17                  }
18              }
19          ],
20          "kind": "const"
21      }
22  ],
23  "sourceType": "script"
24 }
```

经过Parser解析器生成的AST将交由Ignition解释器进行处理。

2-1-3.生成执行上下文

之前的文章介绍过，执行上下文主要是代码在执行过程中的环境信息



2-2.生成字节码

（1）字节码是介于AST和机器码之间的一种代码，与特定类型的机器代码无关，需要通过解释器转换成机器码才可以执行。

（2）在V8的5.9版本之前是没有字节码的，直接把AST转成机器码

效率性能很高，但是有内存占用大、启动时间长、代码复杂度高这几个问题

1）机器码占用内存很大，早期的小内存手机上内存占用问题明显

2）直接编译成机器码导致编译时间长，启动速度慢

3）需要针对不同的CPU架构编写不同的指令集，复杂度很高

V8团队画了快4年时间，引入字节码，才有现在的架构；

（3）js代码、字节码、机器码占用空间对比

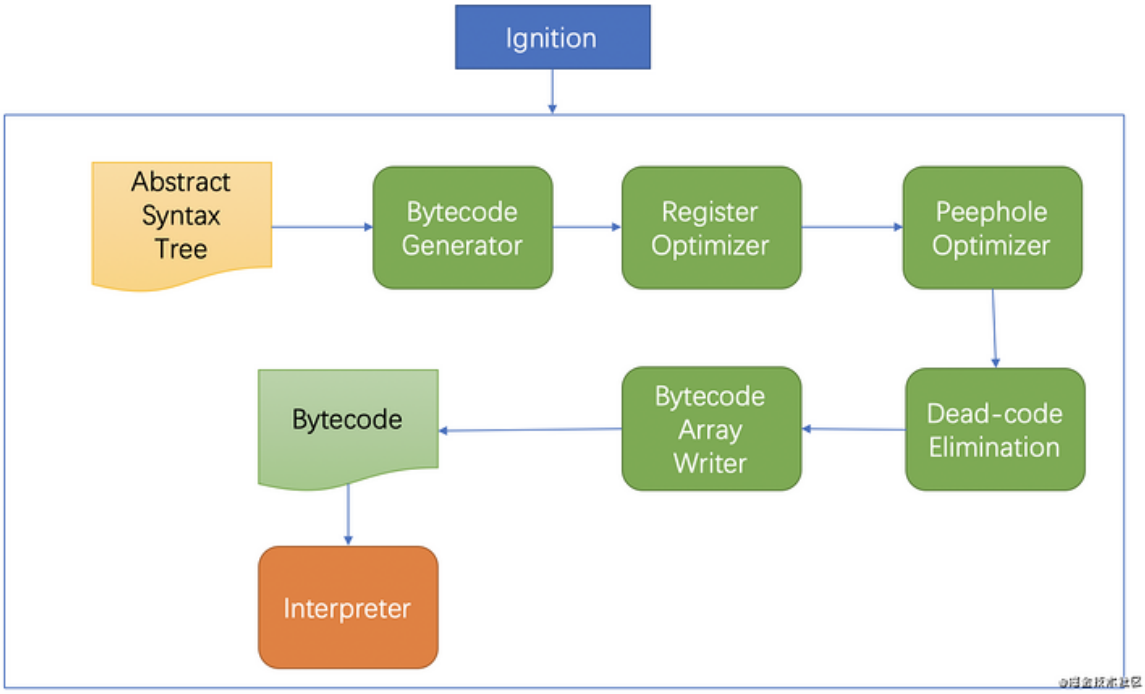
2-3.执行代码

2-3-1.Ignition解释器

（1）Ignition解释器的工作流程图

AST需要先通过字节码生成器(ByteCodeGenerator)，再

经过一系列的优化之后才能生成字节码。



其中的优化包括：

- Register Optimizer：主要是避免寄存器不必要的加载和存储
- Peephole Optimizer：寻找字节码中可以复用的部分，并进行合并
- Dead-code Elimination：删除无用的代码，减少字节码的大小

（2）如果有一段第一次执行的字节码，解释器 Ignition 会逐条解释执行。

在执行的过程中，会监视代码的执行情况并记录执行信息，如函数的执行次数、每次执行函数时所传的参数等。当同一段代码被执行多次，就会被标记成 热点代码 。热点代码会交给TurboFan编译器进行处理。

2-3-2.编译器 TurboFan

（1）TurboFan拿到Ignition标记的热点代码后，会先进行优化处理，然后将优化后字节码编译成更高效的 机器码 存储起来。下次再次执行相同代码时，会直接执行相应的机器码，这样就在很大程度上提升了代码的执行效率。

（2）当一段代码不再是热点代码后，TurboFan会进行去优化 的过程，将优化编译后的机器码还原成字节码，将代码的执行权利交还给Ignition。

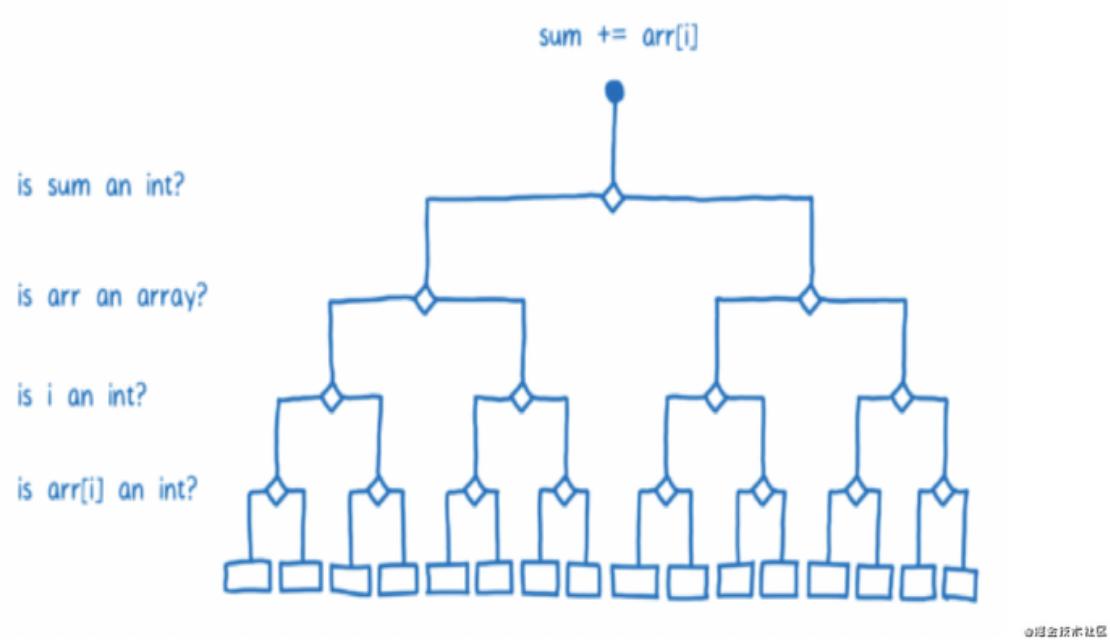
（3）看个例子

以sum += arr[i]为例，由于JS是动态类型的语言，每次的sum和arr[i]都有可能是不同的类型，在执行这段代码时，Ignition每次都会检查sum和arr[i]的数据类型。当发现同样的代码被执行了多次时，就将其标记为热点代码，交给TurboFan。

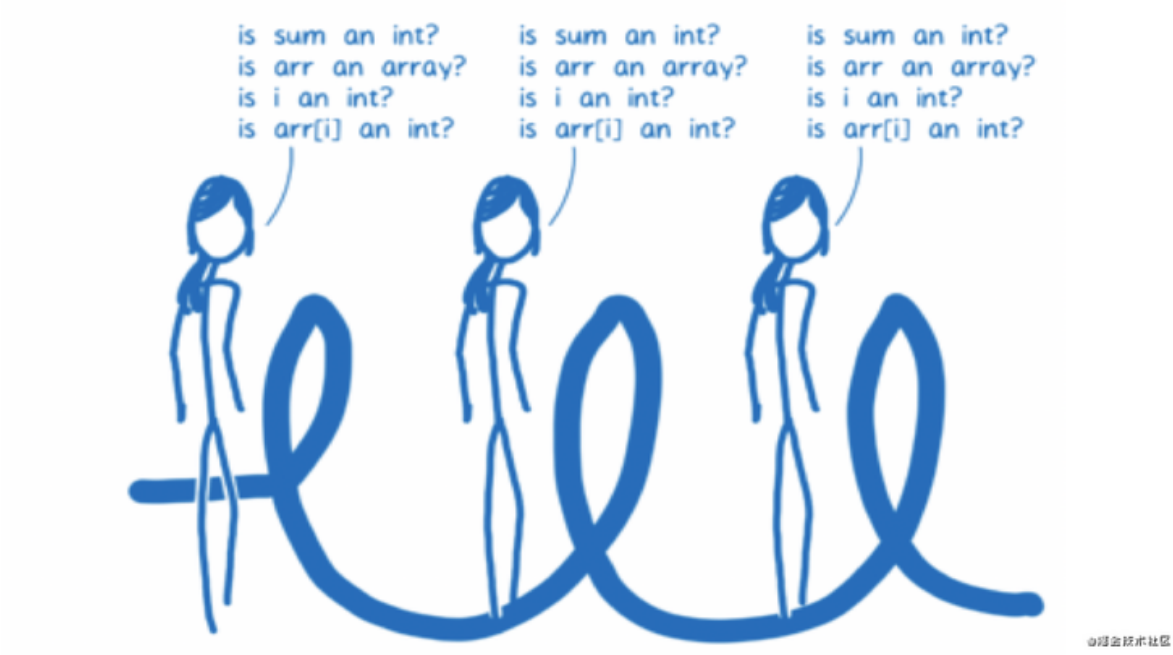
TurboFan在执行时，如果每次都判断sum和arr[i]的数据类型是很浪费时间的。因此在优化时，会根据之前的几次执行确定sum和arr[i]的数据类型，将其编译成机器码。下次再执行时，省去了判断数据类型的过程。

但如果在后续的执行过程中，arr[i]的数据类型发生了改变，之前生成的机器码就不满足要求了，TurboFan会把之前生成的机器码丢弃，将执行权利再交给Ignition，完成去优化的过程。

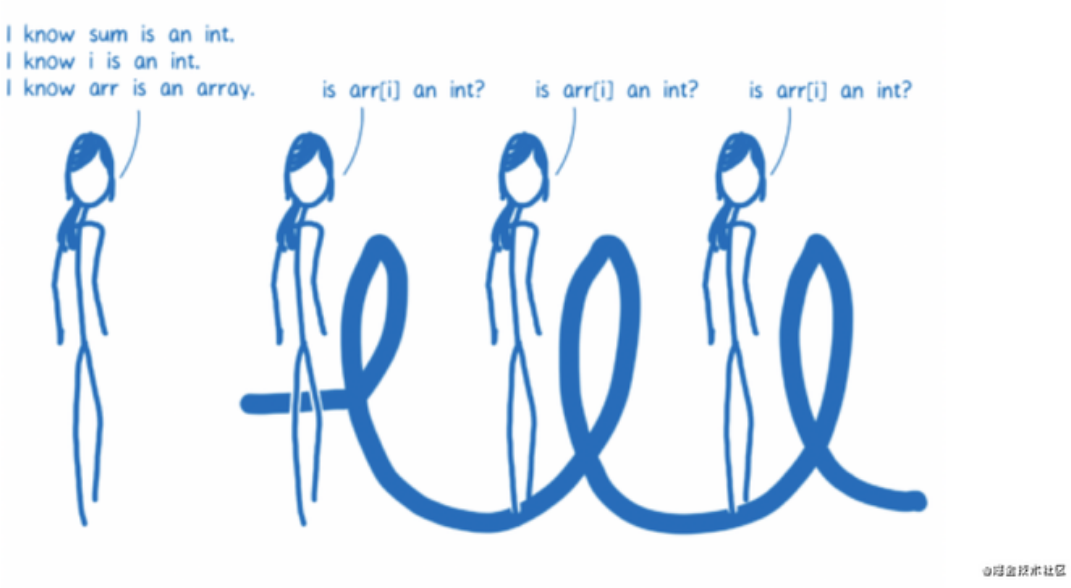
热点代码：


sum += arr[i]
is sum an int?
is arr an array?
is i an int?
is arr[i] an int?

优化前：


is sum an int?
is arr an array?
is i an int?
is arr[i] an int?
is sum an int?
is arr an array?
is i an int?
is arr[i] an int?
is sum an int?
is arr an array?
is i an int?
is arr[i] an int?

优化后：



2-3-3.V8 的解释器和编译器的命名

解释器Ignition~点火器；
编译器TurboFan~涡轮增压发动机；
寓意着代码启动时通过点火器慢慢发动，一旦启动，涡轮增压介入，其执行效率随着执行时间越来越高效率;

2-3-4.sparkplug 火花塞：JS短期会话编译器管道

（1）以前只有Ignition和Turbofan，但光是有高度优化的编译器（如 TurboFan）是不够的。
前端开发的不断变化，特别是对于 短生命周期的会话，例如加载网站或VUE-CLI命令行工具等这种短期JS执行，Turbofan来不及优化，在高优化编译器开始优化之前就已经有很多工作要做，更没有时间去生成什么优化代码了。所以加入了sparkplug；

（2）在V8的v9.1版本加入，chrome91已经支持；
官网链接：https://v8.dev/blog/sparkplug
中文翻译：https://mp.weixin.qq.com/s/w-eV2u_ND9ilo1zLbzkOaQ

2-3-5.即时编译JIT

1.编译可以选择放在两个时机执行：

- 代码构建时，被称为AOT（Ahead Of Time，提前编译或预编译），宿主环境获得的是编译后的代码
- 代码在宿主环境执行时，被称为JIT（Just In Time，即时编译），代码在宿主环境编译并执行

2.JIT与AOT的区别还包括：

- 使用JIT的应用在首次加载时慢于AOT，因为其需要先编译代码，而使用AOT的应用已经在构建时完成编译，可以直接执行代码
- 使用JIT的应用代码体积普遍大于使用AOT的应用，因为在运行时会多出编译器代码

3.V8采用的JIT即时编译方案

是字节码配合解释器和编译器的一种技术；

4.前端框架的JIT和AOT

4-1.可以用两个步骤描述前端框架的工作原理：

（1）根据组件状态变化找到变化的UI
（2）将UI变化渲染为宿主环境的真实UI
借助AOT对模版语法编译时的优化，就能减少步骤1的开销。

4-2.模版语法描述UI的前端框架

（1）大部分采用 模版语法描述UI 的前端框架都会进行的优化，比如Vue3、Angular、Svelte。
其本质原因在于模版语法的写法是固定的，固定意味着「可分析」。
（2）「可分析」意味着在编译时可以标记模版语法中的静态部分（不变的部分）与动态部分（包含自变量，可变的部分），使步骤1在寻找变化的UI时可以跳过静态部分。
甚至Svelte、Solid.js直接利用AOT在编译时建立了「组件状态与UI中动态部分的关系」，在运行时，组件状态

变化后，可以直接执行步骤2。

4-3.JSX描述UI的前端框架

采用 JSX描述UI 的前端框架则很难从AOT中受益。
原因在于JSX是ES的语法糖，作为JS语句只有执行后才能知道结果，所以很难被静态分析。
为了让使用JSX描述UI的前端框架在AOT中受益，有两个思路：

- 使用新的AOT思路
  prepack是meta（原Facebook）推出的一款React编译器，用来实现AOT优化；但是由于复杂度以及人力成本考虑，prepack项目已于三年前暂停了。
- 约束JSX的灵活性
  Solid.js同样使用JSX描述视图，他实现了几个内置组件用于描述UI的逻辑，从而减少JSX的灵活性，使AOT成为可能；

5.在其他领域的使用

Java 和 Python 的虚拟机，苹果的 SquirrelFish Extreme 和 Mozilla 的 SpiderMonkey 也是基于JIT这个技术；

3.总结：

V8执行一段JS代码的大致流程

（1）初始化基础环境；

（2）解析源码生成AST和作用域（执行上下文）

（3）依据AST和作用域生成字节码；

（4）解释执行字节码；

（5）监听热点代码；

（6）优化热点代码为二进制机器代码；

（7）去优化生成的二进制机器代码；

（8）短生命周期的会话使用sparkplug优化；

四.V8做的性能优化措施

1.脚本流

一边下载，一边解析，节省时间；

2.字节码缓存

访问同一个页面时直接复用之前的字节码，不再重新编译生成；

3.内联缓存

Inline Cache，简称为 IC

将主函数中调用的函数， 直接换成要执行的语句 ； 加快

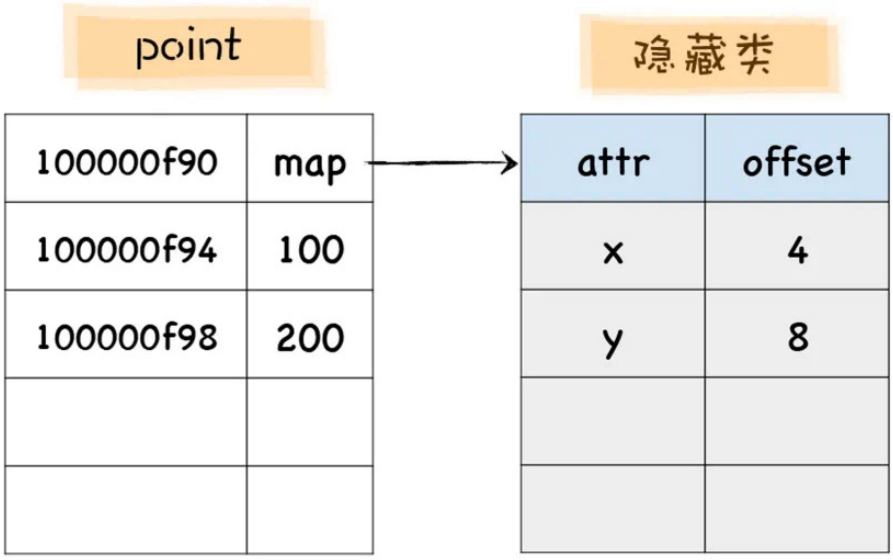
执行速度；

4.延迟解析

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。

5.隐藏类

Object Shapes 或者叫做Hidden Class(隐藏类)
（1）通过隐藏类快速定位到动态加入的属性；
V8会先为 point 对象创建一个隐藏类，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。隐藏类描述了对象的属性布局，它主要包括了属性名称和每个属性所对应的 偏移量 ，比如下面的例子， point 对象的隐藏类就包括了 x 和 y 属性，x 的偏移量是 4，y 的偏移量是 8



（2）注意：动态加入的属性顺序不一样，会造成生成不同的隐藏类，我们动态赋值同一个构造函数对象的时候，尽量保证 顺序也是一致的 。

6.JIT即时编译

将热点代码编译成机器码；执行越久，越多的代码编译成机器码，速度越快；
常用的函数传入的类型保持固定，并且对象的属性越稳定，越有利于性能。

7.快属性，慢属性

（1） 数字 存储在 排序 属性，在 V8 中被称为 elements 。

字符串 存储在 常规 属性，在 V8 中被称为 properties。
（2）读取顺序：
排序属性~数字，按 索引值大小 升序排序
常规属性~字符串，按 创建先后 升序排列。
（3）10 个及 10 个以内会在内部生成属性，
大于十个在 properties 里 线性存储，快属性 。
数量大的情况改为 散列表存储，慢属性 。

五.V8垃圾回收机制

1.V8的内存使用、限制和设置

1-1.内存使用情况

使用 process.memoryUsage(),返回如下

```
{
  heapTotal: 1826816,
  heapUsed: 650472,
  external: 49879,
  rss: 4935680,
}
```

heapTotal 和 heapUsed 代表 V8 的内存使用情况。
external 代表 V8 管理的，绑定到 Javascript 的 C++对象的内存使用情况。
rss, 驻留集大小, 是给这个进程分配了多少物理内存(占总分配内存的一部分) 这些物理内存中包含堆，栈，和代码段。

1-2. V8 的内存限制、为什么这样设计?

（1）默认情况下
32位系统 新生代内存大小为16MB，老生代内存大小为700MB
64位系统下， 新生代内存大小为32MB，老生代内存大小为1.4GB 。
（2）因为 1.5GB 的垃圾回收堆内存，V8 需要花费 50

毫秒以上，做一次非增量式的垃圾回收甚至要 1 秒以上。这是垃圾回收中引起 Javascript 线程暂停执行的事件，在这样的花销下，应用的性能和影响力都会直线下降。

（3）可以修改上限值

比如webgl项目中，当模型特别大的时候，可以调整老生代空间的大小，具体代码如下

node 在启动时传递 –max-old-space-size 或 –max-new-space-size 来调整内存限制的大小，示例如下：node –max-old-space-size=1700 test.js

```
# node在启动时可以传递参数来调整限制内存大小。以下方式就调整了node的使用内存
node --max-old-space-size=1700 // 单位是M
node --max_semi_space_size=1024 // 单位是KB
```

（4）前端常见的造成V8无法回收内存导致内存泄露的原因：闭包和全局变量、事件监听、定时器等等；

2.常用的垃圾回收机制如下：

类型	方法	是否停止程序
引用计数(Reference Counting)	每个对象配置一个计数器即可,每当引用它的对象被删除时，就将其引用数减1，当其引用计数为0时，即可清除	否
标记-清除(Mark-Sweep)	标记阶段标记活对象，清除阶段清除未被标记的对象	是

停止-复制 (Stop-Copy)	内存分为两块，并将正在使用的内存中的存活对象复制到未被使用的内存块中，之后，清除正在使用的内存块中的所有对象，交换两个内存的角色	是
标记-压缩 (Mark-Compact)	对所有可达对象做一次标记，将所有的存活对象压缩到内存的一端，以减少内存碎片	是
增量算法 (Incremental Collecting)	每次，垃圾收集线程只收集一小片区域的内存空间，接着切换到应用程序线程。依次反复，直到垃圾收集完成。	否

1.引用计数法；

（1）跟踪记录每个值被引用的次数
（2）当声明变量并将一个引用类型的值赋值给该变量时，则这个值的引用次数加1，
（3）同一值被赋予另一个变量，该值的引用计数加1 。
（4）当引用该值的变量被另一个值所取代，则引用计数减1，
（5）当计数为 0 的时候，说明无法在访问这个值了，系统将会收回该值所占用的内存空间。
（6）缺点： 循环引用 的时候，引用次数不为0，不会被释放；

2.标记清除法；

（1）从全局的对象开始，把所有引用的对象标记一遍，没被标记的就清掉。这样不管是没被引用的，还是循环引用但是都没被别的对象引用的，都可以检查出来，判断哪些变量没有在执行环境中引用，进行删除；

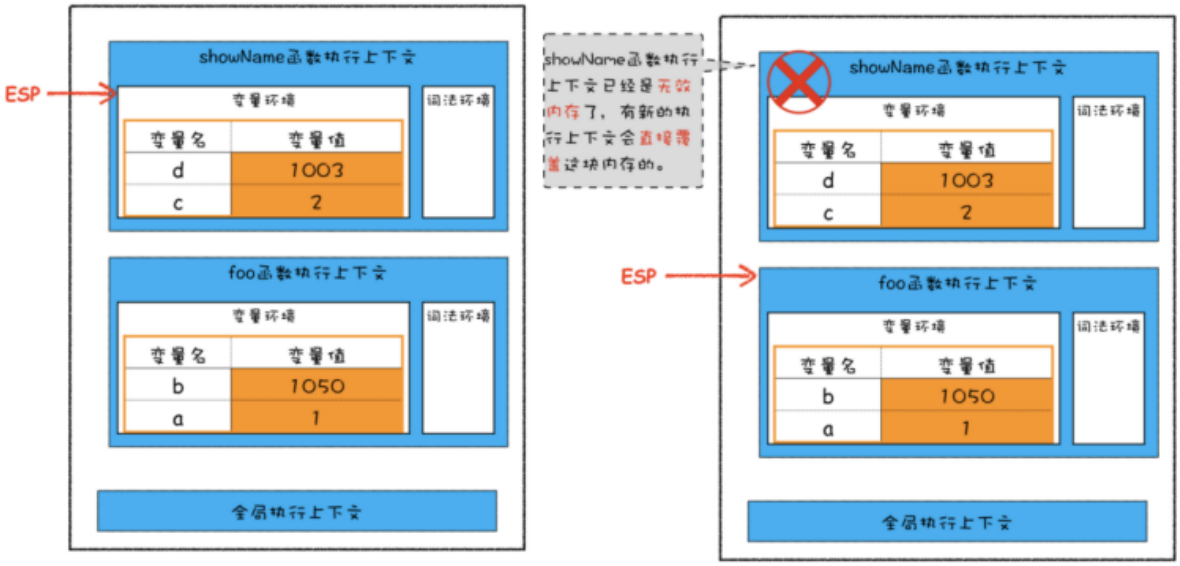
（2）缺点：万一有的不用的对象被放到 全局 了，那就永远不会回收了。

3.其他常见语言的垃圾回收机制

（1）C、C++ 的内存都是程序员 手动管理 的
比如 C++ 的 class 有构造函数和析构函数，构造函数里申请内存，析构函数里面就把这些内存释放掉；手动管理内存的方式比较麻烦。
（2）Java的垃圾回收跟JS类似；有专门的垃圾回收器，最开始通过引用计数，后来改成了标记清除；
（3）Rust
(1)不需要程序员手动管理内存，但也没有垃圾回收，却把内存管理的更好，而且能避免 99% 的内存泄漏问题。
(2) 所有权机制
Rust认为堆中的对象之所以难管理就是因为被太多地方引用了。所以它限制对象只能属于某个函数，只能有 一个引用 ，别的引用自己复制一份去，这样函数调用结束就可以把用到的堆中的对象全部回收了，根本不会留下垃圾。
所有权机制通过限制对象的引用的方式来做到了不需要垃圾回收器也能很好的管理内存。而且也没有 js 那种不小心把对象放到全局就会内存泄漏的问题（因为只允许一个引用）

3.调用栈中的数据的GC

调用栈有一个记录当前执行状态的指针（称为 ESP）
JS引擎通过下移ESP指针来销毁 栈顶 某个执行上下文的过程。
如下图，上面那个已经是无效的，有新的会直接覆盖；



4.堆空间中数据的GC

有两种垃圾回收器
（1）主垃圾回收器，全量标记和整理
（2）副垃圾回收器～Scavenger清道夫，只从新生代回收垃圾。

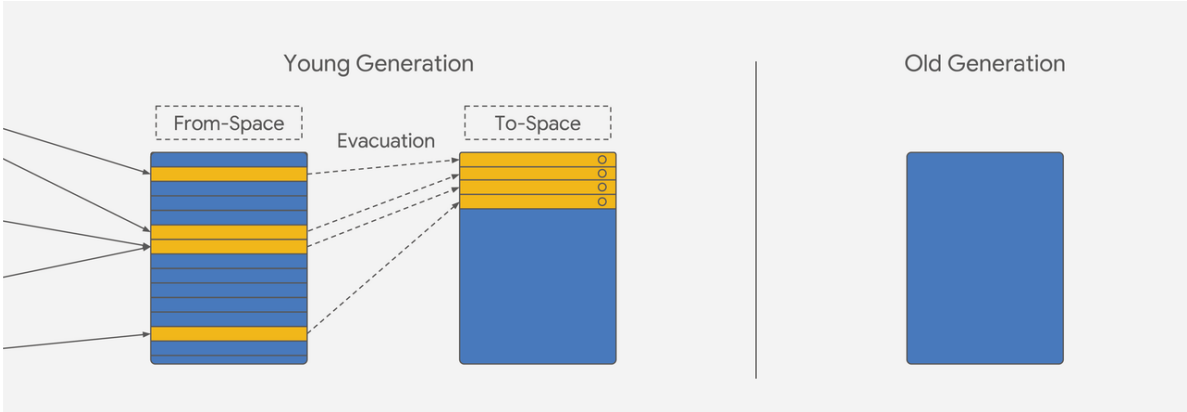
1.代际假说和分代收集~新生代，老生代；

（0）提高垃圾回收的效率，V8将堆分为新生代和老生代两个部分，

（1）其中新生代为 存活时间较短 的对象(需要经常进行垃圾回收)，内存占用小，GC频繁；
只支持1-8M容量；使用副垃圾回收器；
（2）而老生代为 存活时间较长 的对象(垃圾回收的频率较低)，内存占用多，GC不频繁；
使用主垃圾回收器；

2.新生代的GC算法

新生代的对象通过 Scavenge 算法进行GC。在 Scavenge 的具体实现中，主要采用了 Cheney 算法。
(1)Cheney 算法是一种采用停止复制（stop-copy）的方式实现的垃圾回收算法。
(2)它将堆内存 一分为二 ，每一部分空间成为 semispace-半空间。
(3)在这两个 semispace 空间中，只有一个处于使用中，另一个处于闲置中。
(4)处于使用中的 semispace 空间成为 From 对象空间，处于闲置状态的空间成为 To 空闲空间。
(5)当我们分配对象时，先是在 From 空间中进行分配。当开始进行垃圾回收时，会检查 From 空间中的存活对象，这些存活对象将被复制到 To 空间中，同时还会将这些对象有序的排列起来~~相当于内存整理，所以没有内存碎片，而非存活对象占用的空间将被释放。
完成复制后，From空间和To空间的角色发生 对换 。



(6)Scavenge 是典型的空间换取时间的算法，而且复制需要时间成本，无法大规模地应用到所有的垃圾回收中，但非常适合应用在新生代中进行快速频繁清理。

3.对象晋升策略

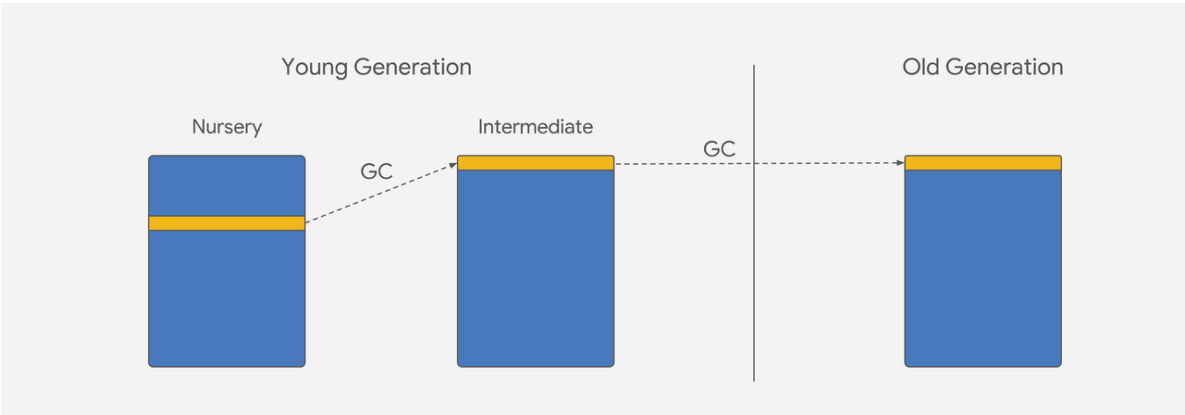
（1）对象从新生代中移动到老生代中的过程称为晋升。

（2）晋升条件主要有两个：

(1)对象是否经历过 两次 Scavenge 回收都未清除，则移动到老生代

新生代分为两个区域，nursery子代，intermediate子代两个区域。

初始化分配在nursery，第一轮放到intermediate，第二轮放到老生代。



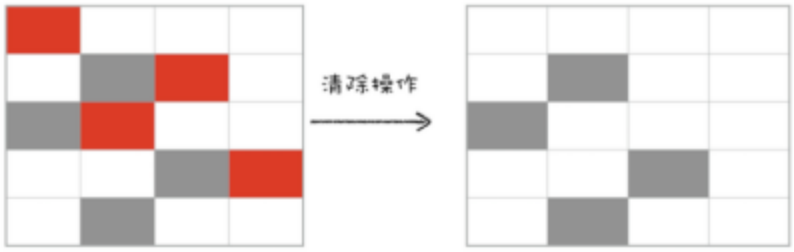
(2)To 空间已经使用超过 25% ， To 空间对象移动到老生代
因为这次 Scavenge 回收完成后，这个 To 空间将变成 From 空间，接下来的内存分配将在这个空间中进行，如果占比过高，会影响后续的内存分配

4.写屏障

写缓冲区中有一个列表(CrossRefList)，列表中记录了所有 老生区对象指向新生区 的情况
这样可以快速找到指向新生代该对象的老生代对象，根据他是否活跃，来清理这个新生代对象；

5.老生代的GC

（1）老生代的内存空间较大且存活对象较多，使用新生代的Scavenge 复制算法，会耗费很多时间，效率不高；而且还会浪费一半的空间；
（2）为此V8使用了 标记-清除算法 (Mark-Sweep)进行垃圾回收，并使用 标记-压缩算法 (Mark-Compact)整理内存碎片，提高内存的利用率。步骤如下：
1） 对老生代进行第一遍扫描，标记存活的对象
从一组根元素开始，递归遍历这组根元素，在这个遍历过程中，能到达的元素称为活动对象，没有到达的元素就可以判断为垃圾数据；
2） 对老生代进行第二次扫描，清除未被标记的对象



3） 标记-整理算法，标记阶段一样是递归遍历元素，整理阶段是将存活对象往内存的一端移动

4） 清除掉存活对象边界外的内存

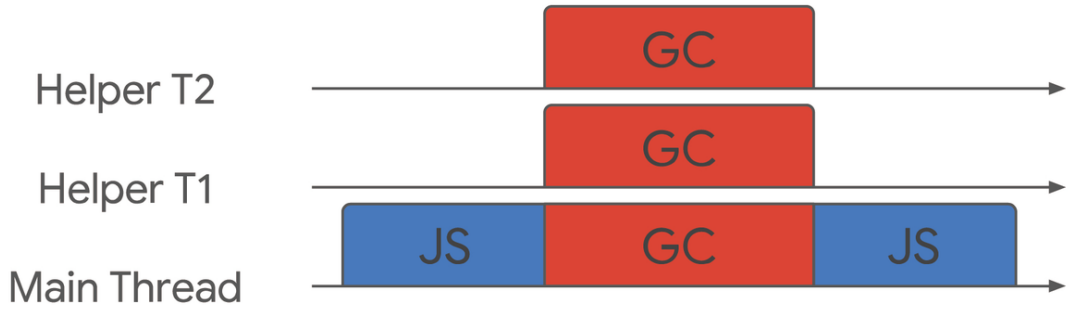
注意，不管那种，整理内存后只要地址有变化，需要及时更新到调用栈的；

6.全停顿Stop-The-World

（1）JavaScript 是运行在主线程之上的，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）；

（2）解决办法~

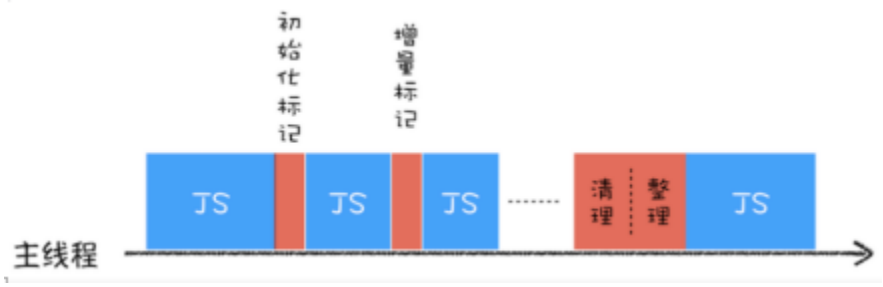
1）并行垃圾回收

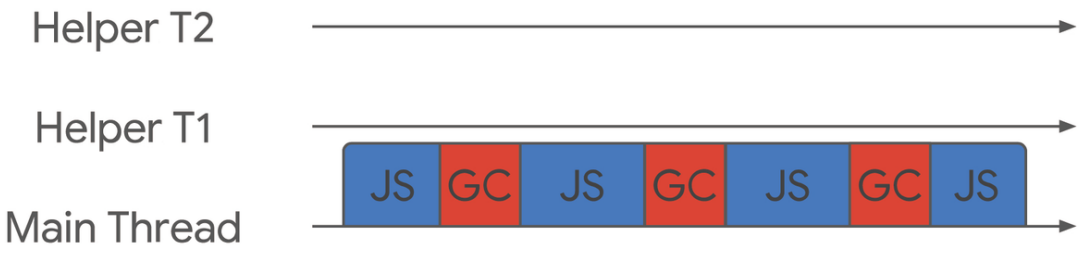


2）增量垃圾回收

增量标记算法Incremental Marking

V8 将标记过程分为一个个的子标记过程，同时让垃圾回收标记和 JavaScript 应用逻辑交替进行，直到标记阶段完成；





3）并发垃圾回收

（3）新生代因为内存小，活动对象少，全停顿影响不大，但是老生代可能会造成卡顿明显；
所以新生代的副垃圾回收器使用 并行 方案，主垃圾回收器使用 并发 方案。

六.JS一些常见语法、数据结构在V8中的实现；

1.sort() 在V8中的实现；

源码地址：
https://github.com/v8/v8/blob/98d735069d0937f367852ed968a33210ceb527c2/src/js/array.js#L709
Array.sort 在数组长度上采用了不同排序算法的优化，排序的元素个数是 n 的时候，那么就会有以下几种情况：
（1）当 n<=10 时，采用 插入 排序；
（2）当 n>10 时，采用 三路快速排序；
（3）10< n <=1000，采用 中位数 作为哨兵元素；

（4）n>1000，每隔 200~215 个元素挑出一个元素，放到一个新数组中，然后对它排序，找到 中间位置 的数，以此作为中位数。

2.数组其他方法的V8实现源码

pop：https://github.com/v8/v8/blob/98d735069d0937f367852ed968a33210ceb527c2/src/js/array.js#L394

Push:
https://github.com/v8/v8/blob/98d735069d0937f367852ed968a33210ceb527c2/src/js/array.js#L414

map：
https://github.com/v8/v8/blob/98d735069d0937f367852ed968a33210ceb527c2/src/js/array.js#L1036

Slice :https://github.com/v8/v8/blob/98d735069d0937f367852ed968a33210ceb527c2/src/js/array.js#L586

Fliter:
https://github.com/v8/v8/blob/98d735069d0937f367852ed968a33210ceb527c2/src/js/array.js#L1024

3.哈希表在V8中的使用

（1） js 超大数组
v8 将其转成了 NumberDictionary；
NumberDictionary 底层是用 开放寻址法 实现的 HashTable，当数组长度超过 33554432(32 x 1024 x 1024) 时，JSArray 的内部实现，会由 FastElement 模式（FixedArray 实现），变成 SlowElement 模式（HashTable 实现）。
（2）ES6 中的 Set 和 Map 是用 链表法 （链地址法）实现的 HashTable，内部类是 OrderedHashTable。
Map是有序的，C++中map是字典序，JS中是按照插入顺序；
（3）redis使用链表法解决哈希冲突；

4.async、await在V8的实现和优化

详细见这一篇：https://v8.js.cn/blog/fast-async/
V8实现更快的异步函数和 Promise小结
（1）两个重要优化，使异步函数更快：

- 删除两个额外的 microtick
- 去除了 throwaway promise。

（2）最重要的是，我们通过零成本异步堆栈跟踪改进了开发体验，这些可以使用在异步函数的 await 表达式和异步函数中使用 Promise.all()。

为 JavaScript 开发者提供了一些很好的性能建议：

- 使用 async 函数和 await 替代手写的 Promise 代码
- 坚持 JavaScript 引擎提供的原生 Promise 实现，以避免在 await 中使用额外的两个 microtick。

~持续更新ing~

同时欢迎关注我的个人微信公众号：