

# V8实现javascript核心设计解析

- 前言
- v8如何执行一段js代码
- 懒惰解析
- 闭包
  - 预解析器
- 垃圾回收机制
  - 垃圾是如何产生的?
  - 大概步骤
  - 副垃圾回收器
  - 主垃圾回收器
  - 优化执行效率

## 前言

javascript设计思想



JavaScript的设计思想

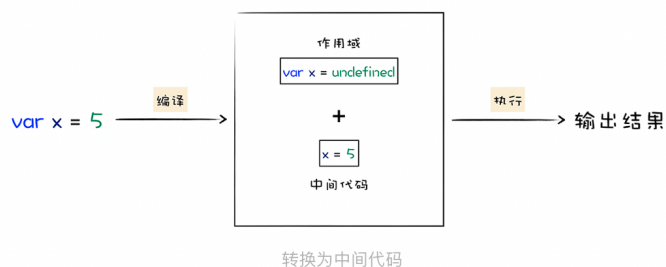
## 什么是V8

V8 是 JavaScript 虚拟机的一种，将js代码翻译成机器可以识别的机器语言



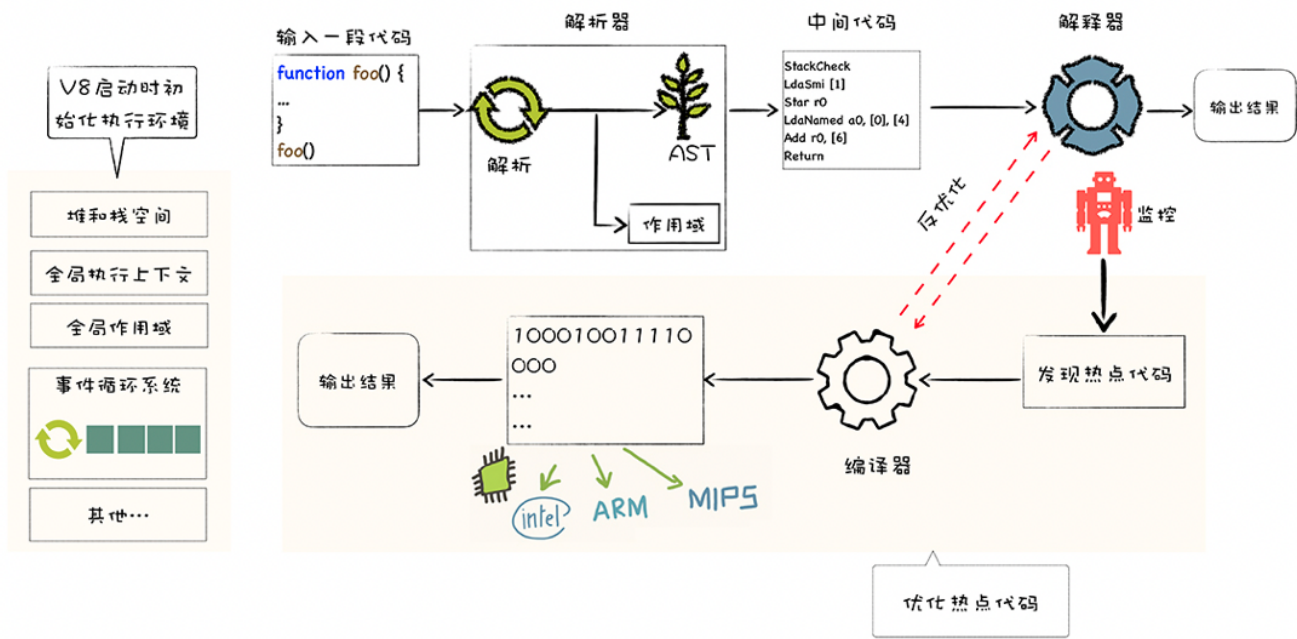
## v8如何执行一段js代码

其主要核心流程分为【编译】和【执行】两步。编译：js=>低级代码，执行：执行低级代码=>输出结果。



V8 混合编译执行和解释执行这两种手段，混合使用编译器和解释器：JIT（Just In Time）

解释执行的启动速度快，但是执行时的速度慢，而编译执行的启动速度慢，但是执行时的速度快。



V8执行一段JavaScript流程图

1. 初始化基础环境；
2. 解析源码生成 AST 和作用域；
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的机器代码；
7. 反优化生成的二进制机器代码。

【AST】。。。

【字节码】

早期V8直接把 JS 编译成机器码，机器码的执行性能非常之高。后来chrome遇到了一个bug，V8进行重构，引入了字节码。

原方案问题：机器码占空间很大，v8 没有办法把所有 js 代码编译成机器码缓存下来，因为这样不仅缓存占用的内存、磁盘空间很大，而且退出 Chrome 再打开时序列化、反序列化缓存所花费的时间也很长，时间、空间成本都接受不了。

字节码优点：占用空间小 => 可缓存编译后的字节码 => 第二次执行更快

V8 并不会一次性将所有的 JavaScript 解析为中间代码

1. 编译不必要的代码会占用 CPU 资源。
2. 在 GC 前会占用不必要的内存空间。
3. 编译后的代码会缓存在磁盘，占用磁盘空间。

## 懒惰解析

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。

只有IIFE才会直接被解析。

```
function foo(a,b) {  
    var d = 100;  
    var f = 10;  
    return d + f + a + b;  
}  
var a = 1;  
var c = 4;  
foo(1, 5);
```

懒惰解析看着很简单，但是遇到闭包时，变量的处理就比较复杂。

```
function foo1() {
  var a = 1;
  var b = 2;
  return function foo2() {
    console.log('111');
  };
}
var foo3 = foo1();
foo3();
```

foo1  
(anonymous)

▼ Scope

▼ Local

- ▶ Return value: *f* foo2()
- ▶ a: 1
- ▶ b: 2
- ▶ this: Window
- ▶ Global

foo2  
(anonymous)

▼ Scope

▼ Local

- ▶ this: Window
- ▶ Global

## 闭包

```
function foo1(){
  var a = 1;
  var b = 2;
  return function foo2() {
    console.log(a);
  }
}
var foo3 = foo1();
foo3();
```

## 预解析器

当v8解析顶层代码时，遇到函数，预解析器不会跳过而是对该函数做一次快速的预解析。

1. 是否有语法错误 => 抛异常
2. 函数内部是否引入外部变量，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

The screenshot shows the Chrome DevTools interface. The 'Call Stack' panel on the left shows the function 'foo2' (anonymous) being called. The 'Scope' panel on the right shows the local scope with 'this' pointing to 'Window', and the closure scope for 'foo1' with a variable 'a' set to '1'. The global scope is also visible.

```
function foo1() {  
  let obj1 = {};  
  let obj2 = { b: obj1 };  
  obj1.a = obj2;  
  return function foo2() {  
    console.log(obj1);  
  };  
}  
var foo3 = foo1();  
foo3();  
console.dir(foo3);
```

循环引用

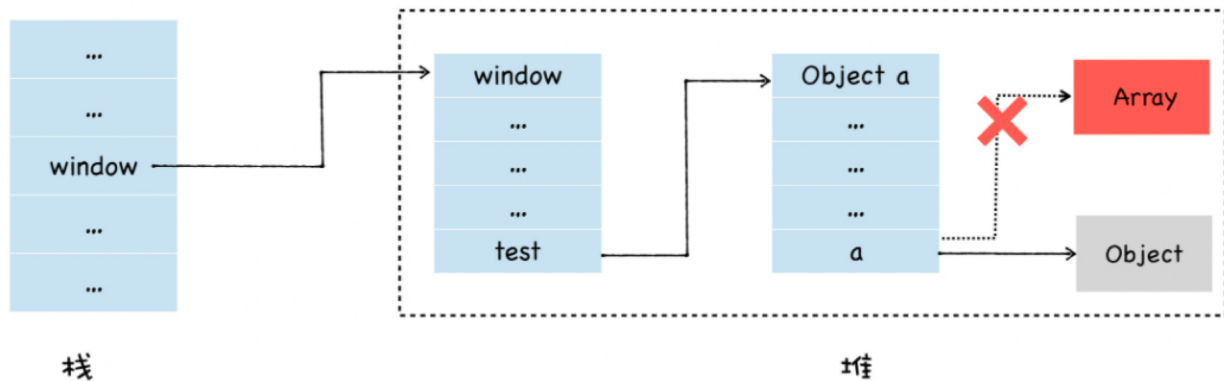
The screenshot shows the Chrome DevTools 'Scope' panel for the function 'foo2'. It displays a complex state where the 'obj1' variable in the closure of 'foo1' now points to the 'foo2' function itself, creating a circular reference. The 'obj2' variable also points to 'obj1', which points to 'foo2', which points to 'obj1', and so on. This illustrates how the pre-parser handles recursive function calls by copying references to the heap.

## 垃圾回收机制

计数垃圾回收策略，无法解决上面的问题。==> 标记清除回收策略

## 垃圾是如何产生的？

```
window.test = new Object()  
window.test.a = new Uint16Array(100)  
window.test.a = new Object()
```



## 大概步骤

标记

活动/非活动对象

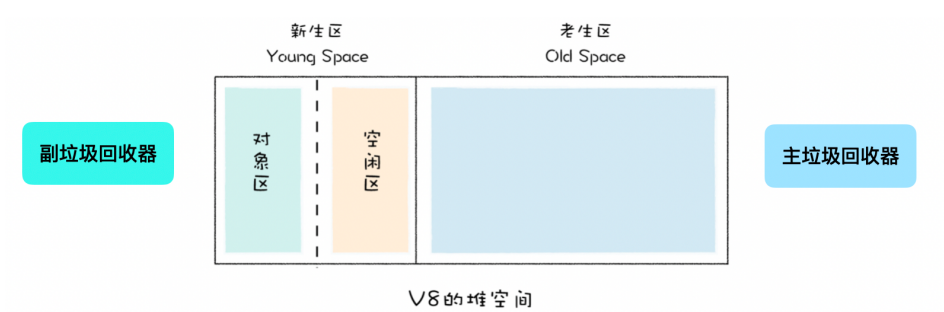
回收

整理

内存碎片

代际假说：

- 大部分对象在内存中存在的时间很短，简单来说，就是很多对象一经分配内存，很快就变得不可访问
- 不死对象，会活得很久，例如：window、document、window下的全局对象

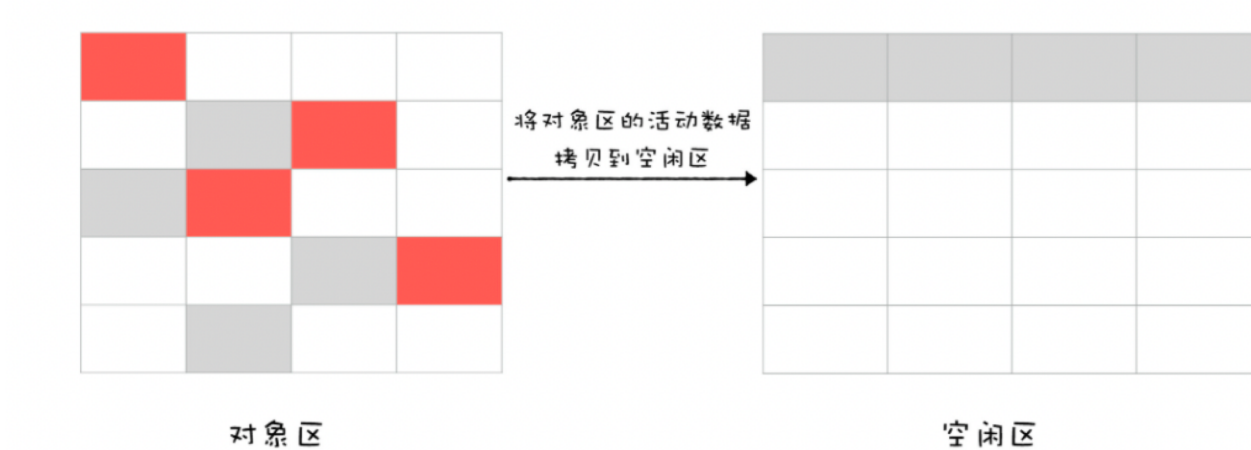


## 副垃圾回收器

Scavenge 算法

新生代空间对半划分为两个区域，一半是对象区域，一半是空闲区域。

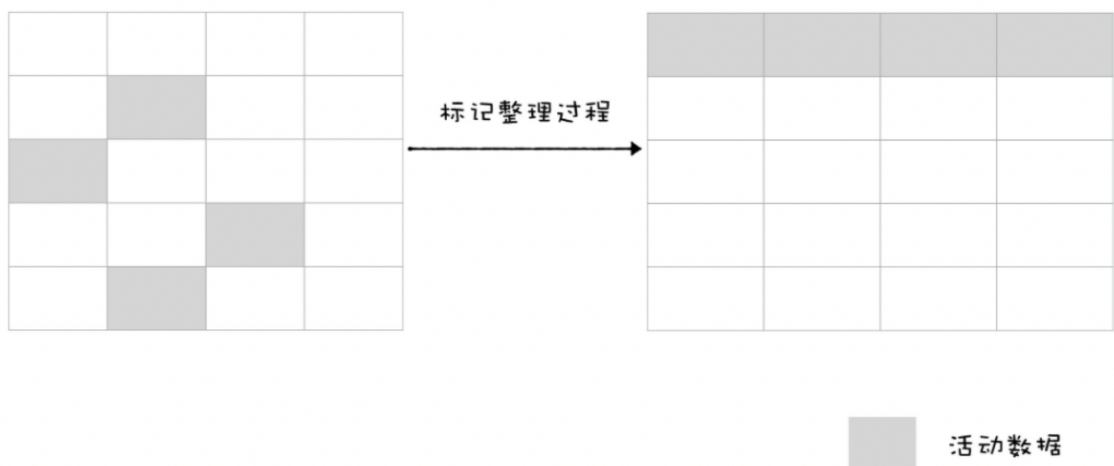
新的数据都分配在对象区域，等待对象区域快分配满的时候，垃圾回收器便执行垃圾回收操作，之后将存活的对象从对象区域拷贝到空闲区域，并将两个区域互换。



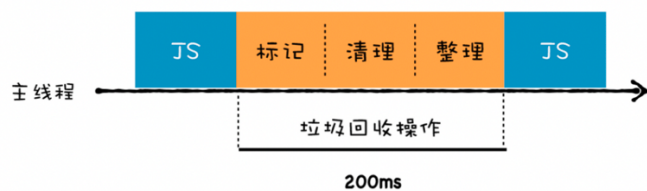
## 主垃圾回收器

标记 - 整理 算法

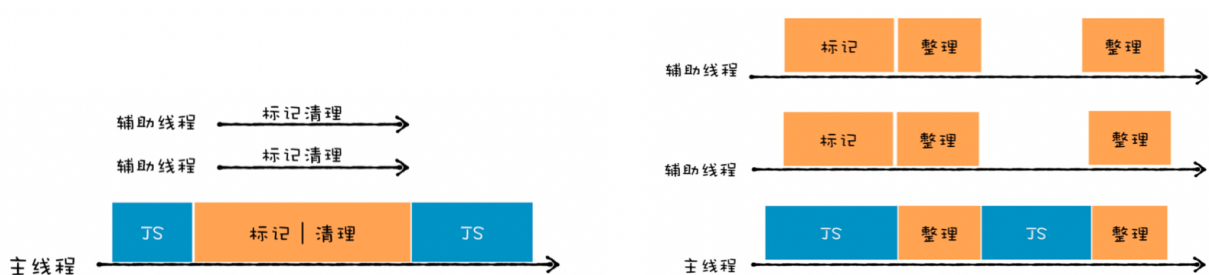




## 优化执行效率



## 全停顿 (Stop-The-World)



小思考

```
function strToArray(str) {  
  let i = 0  
  const len = str.length  
  let arr = new Uint16Array(str.length)  
  for (; i < len; ++i) {  
    arr[i] = str.charCodeAt(i)  
  }  
  return arr;  
}  
  
function foo() {  
  let i = 0  
  let str = 'test V8 GC'  
  while (i++ < 1e5) {  
    strToArray(str);  
  }  
}  
  
foo()
```

参考文档：

[JS 引擎与字节码的不解之缘](#)

[JS中的循环引用及问题](#)

[V8角度看闭包](#)

[图解V8](#)

[js垃圾回收机制](#)

[精度V8 引擎 Lazy Parsing](#)