浏览器原理2

JavaScript执行机制

只有理解了 JavaScrip 的执行上下文，你才能更好地理解 JavaScript 语言本身

变量提升

```
var myname = undefined
function showName() {
    console.log('函数执行');
}

showName()
console.log(myname)
myname = '京程一灯'

```

所谓的变量提升，是指在 JavaScript 代码执行过程中，JavaScript 引擎把变量的声明部分和函数的声明部分提升到代码开头的“行为”。变量被提升后，会给变量设置默认值，这个默认值就是我们熟悉的 undefined。
实际上变量和函数声明在代码里的**位置是不会改变**的，而且是在编译阶段被 JavaScript 引擎放入内存中。

代码中出现相同的变量或者函数

```
function a(){
    alert(20)
}

```

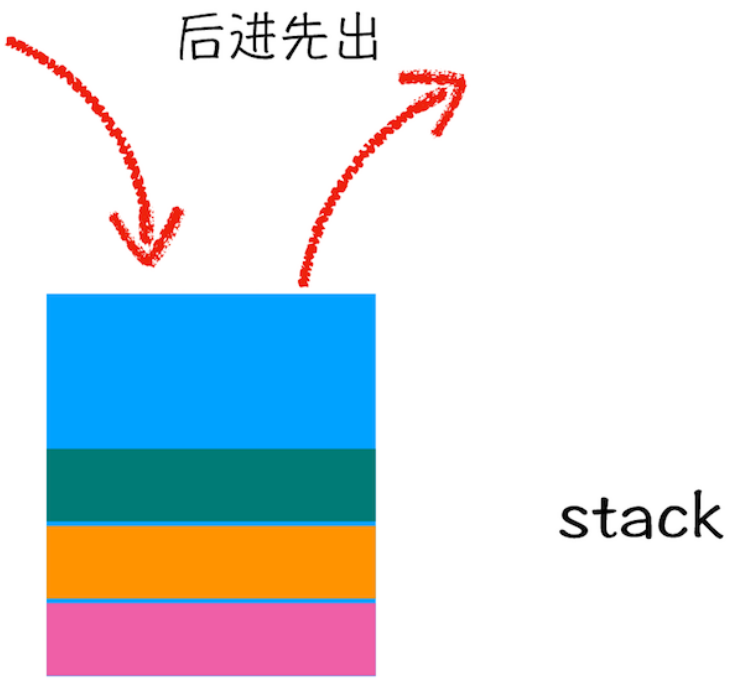
```
alert(a)
a();
a=3;
alert(a)
a=6;
a();

```

调用栈

哪些情况下代码才算是“一段”代码，才会在执行之前就进行编译并创建执行上下文。一般说来，有这么三种情况：

1. 当 JavaScript 执行全局代码的时候，会编译全局代码并创建全局执行上下文，而且在整个页面的生存周期内，全局执行上下文只有一份。
2. 当调用一个函数的时候，函数体内的代码会被编译，并创建函数执行上下文，一般情况下，函数执行结束之后，创建的函数执行上下文会被销毁。
3. 当使用 eval 函数的时候，eval 的代码也会被编译，并创建执行上下文。

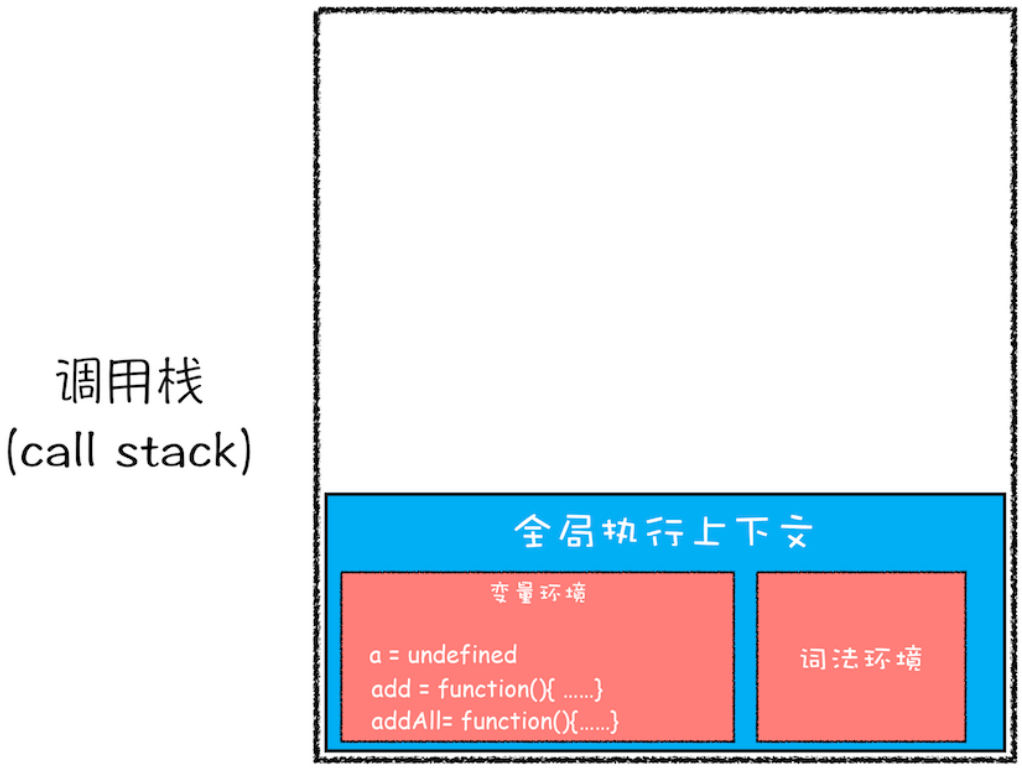
调用栈就是用来管理函数调用关系的一种数据结构.

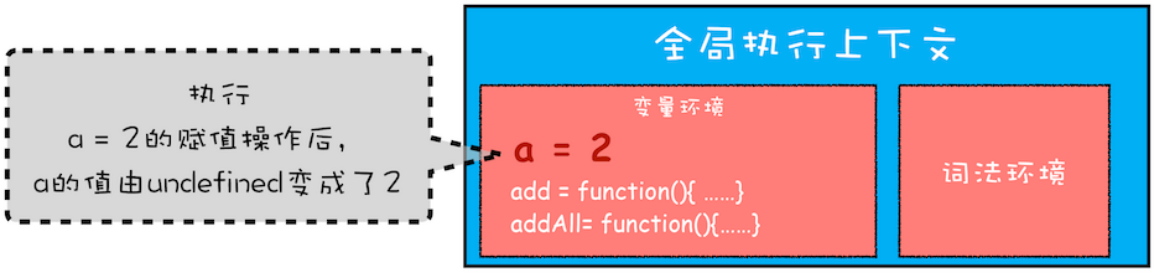



```
var a = 2
function add(b,c){
```

```
3   return b+c
4 }
5 function addAll(b,c){
6     var d = 10
7     var result = add(b,c)
8     return  a+result+d
9 }
10 addAll(3,6)
```

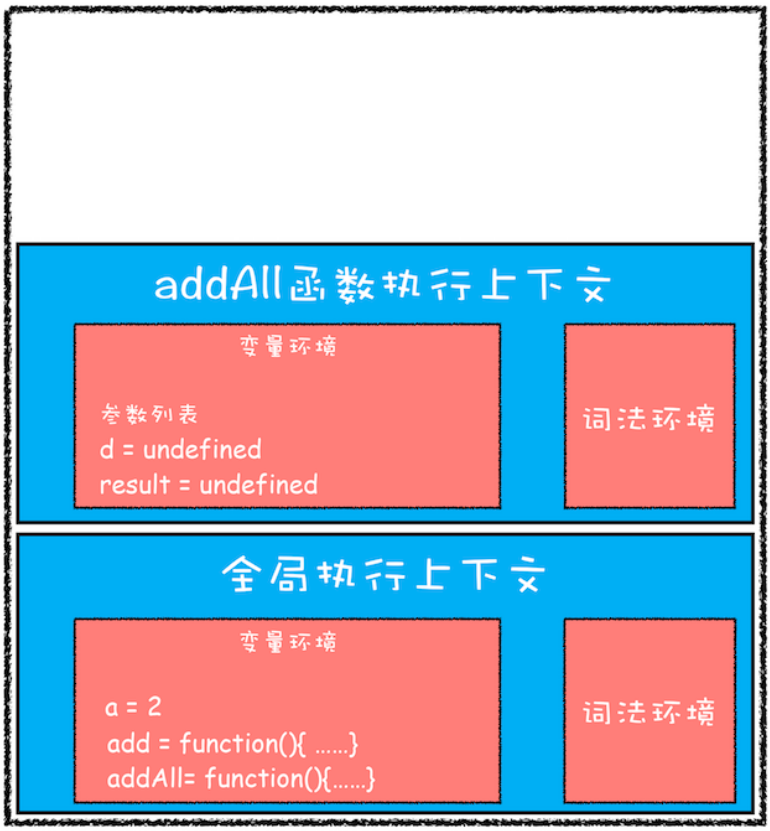
- 第一步，创建全局上下文，并将其压入栈底。





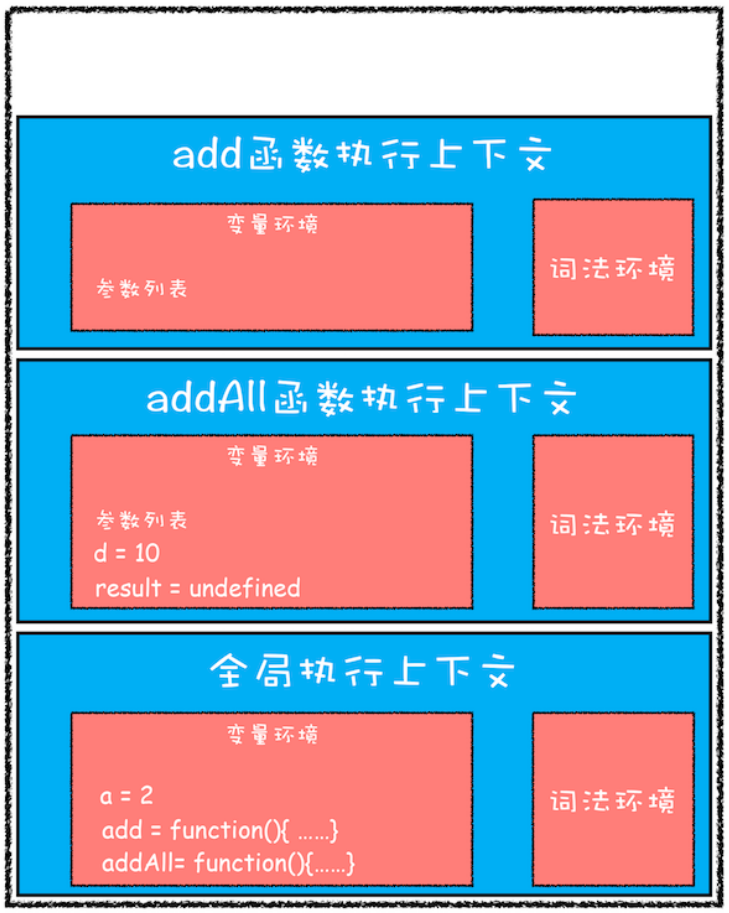
- 调用 addAll 函数

调用栈
(call stack)



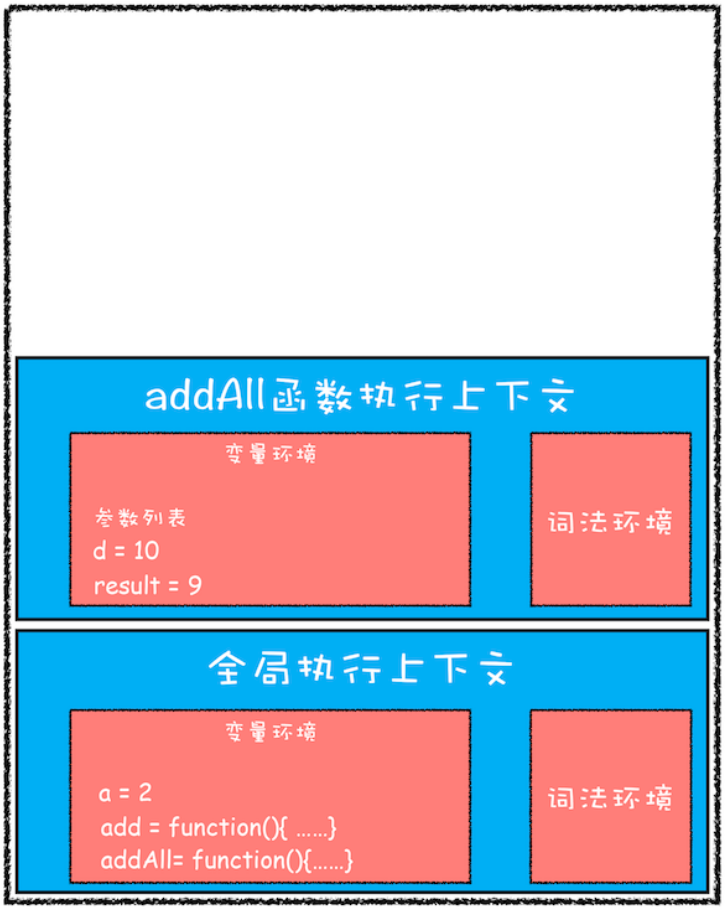
- 执行到 add 函数

调用栈
(call stack)



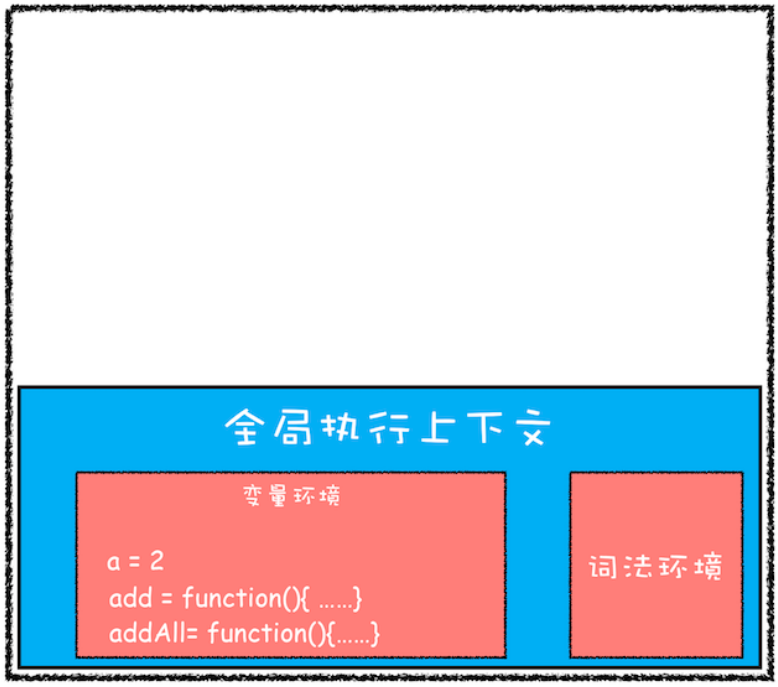
- 执行完 add 函数

调用栈
(call stack)



- 执行完 addAll函数

调用栈
(call stack)



浏览器中查看

块级作用域

正是由于 JavaScript 存在变量提升这种特性，从而导致了很多与直觉不符的代码，这也是 JavaScript 的一个重要设计缺陷。ECMAScript6（以下简称 ES6）已经通过引入块级作用域并配合 let、const 关键字，来避开了这种设计缺陷，但是由于 JavaScript 需要保持向下兼容，所以变量提升在相当长一段时间内还会继续存在。

作用域

作用域是指在程序中定义变量的区域，该位置决定了变量的生命周期。通俗地理解，作用域就是变量与函数的可访问范围，即作用域控制着变量和函数的可见性和生命周期。
在 ES6 之前，ES 的作用域只有两种：全局作用域和函数作用域。
变量提升所带来的问题：

1. 变量容易在不被察觉的情况下被覆盖掉
2. 本应销毁的变量没有被销毁

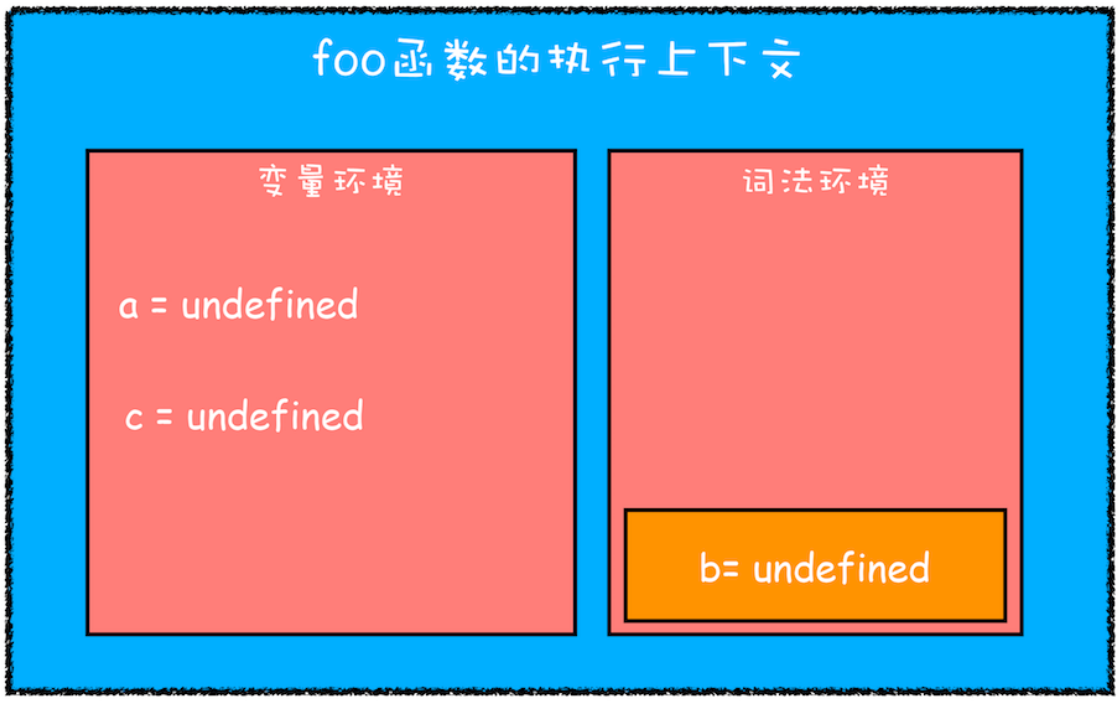
```
function foo(){
  for (var i = 0; i < 7; i++) {
  }
  console.log(i);
}
foo()
```

JavaScript 是如何支持块级作用域的

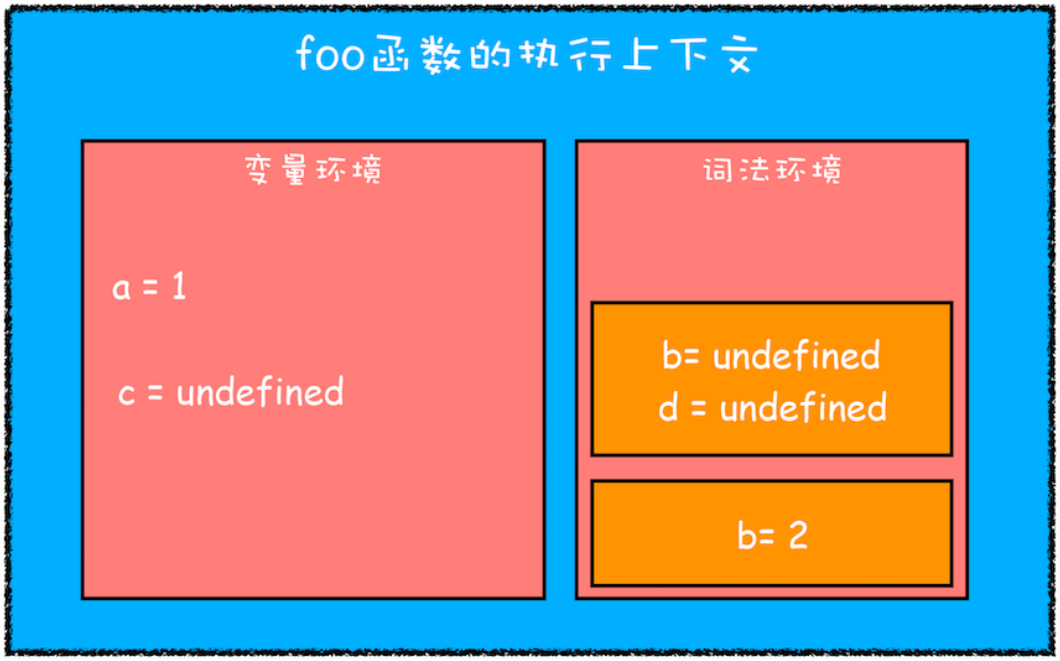
```
function foo(){
    var a = 1
    let b = 2
    {

      let b = 3
      var c = 4
      let d = 5
      console.log(a)
      console.log(b)
    }
    console.log(b)
```

```
13      console.log(c)
14      console.log(d)
15  }
16  foo()
```

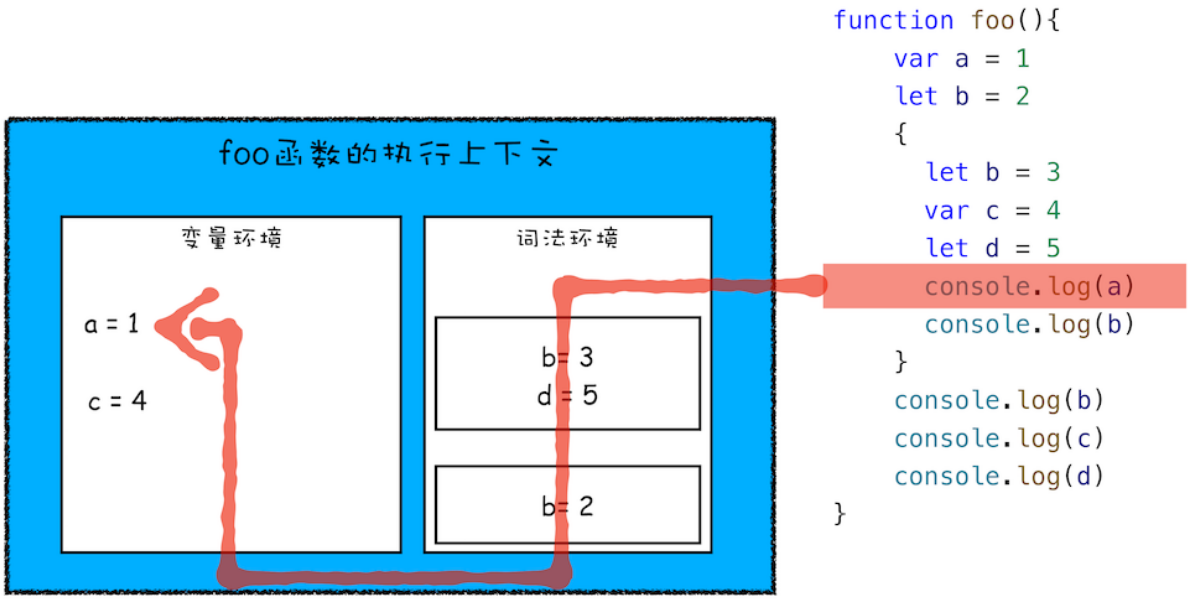
- 编译并创建执行上下文



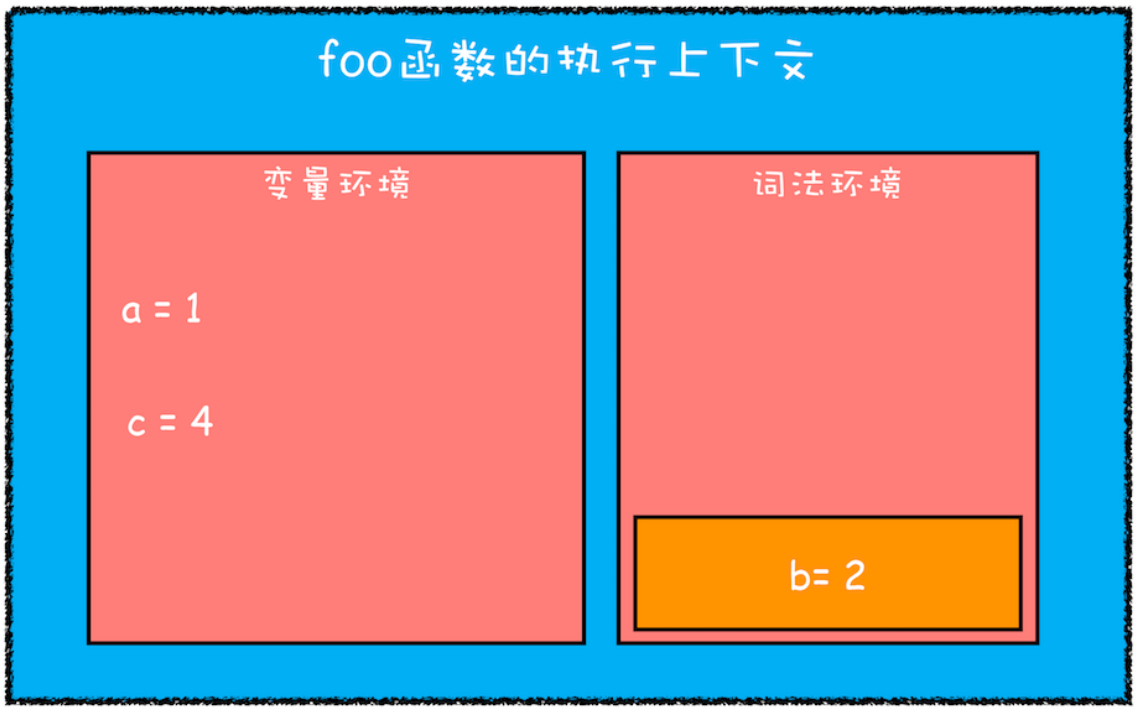
- 继续执行代码



- 变量查找过程



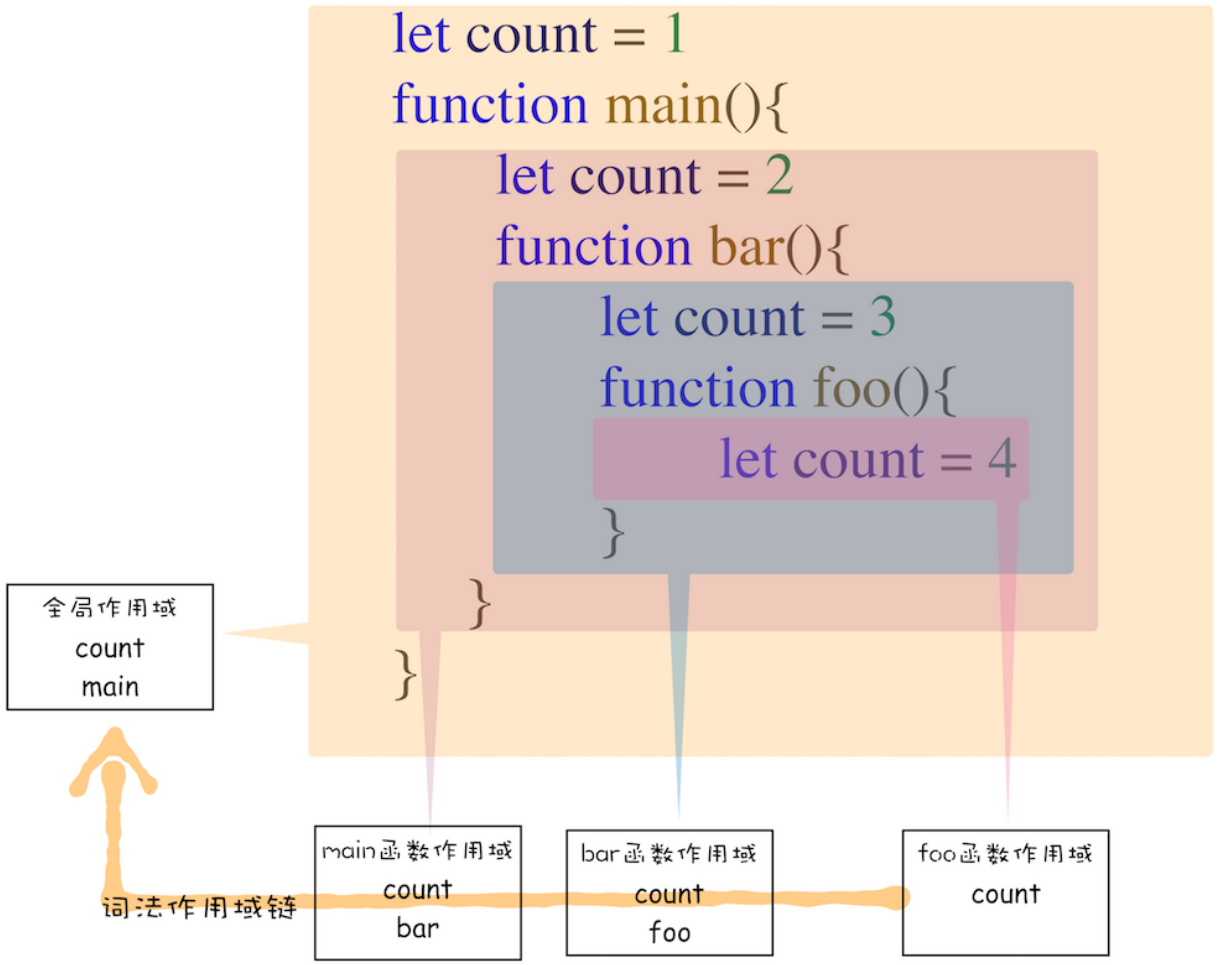
- 最终执行上下文




```
let a =2
console.log(a)
{
  console.log(a)
  function a(){console.log(1)}
}
console.log(a)
```

作用域链和闭包

词法作用域是代码阶段就决定好的，和函数是怎么调用的没有关系。




```
1
2  function bar() {
3      console.log(myName)
4  }
5  function foo() {
6      var myName = "一灯"
7      bar()
8  }
9  var myName = "京程一灯"
10 foo()
```

其实在每个执行上下文的变量环境中，都包含了一个外部引用，用来指向外部的执行上下文，我们把这个外部引用称为 outer。当一段代码使用了一个变量时，JavaScript 引擎首先会在“当前的执行上下文”中查找该变量，比如上面那段代码在查找 myName 变量时，如果在当前的变量环境中没有查找到，那么 JavaScript 引擎会继续在 outer 所指向的执行上下文中查找。

闭包

```
function foo() {
    var myName = "一灯"
    let test1 = 1
    const test2 = 2
    var innerBar = {
        getName:function(){
            console.log(test1)
            return myName
        },
        setName:function(newName){
            myName = newName
        }
    }
    return innerBar
}
var bar = foo()
bar.setName("京程一灯")
bar.getName()
console.log(bar.getName())
```

根据词法作用域的规则，内部函数 getName 和 setName 总是可以访问它们的外部函数 foo 中的变量，所以当 innerBar 对象返回给全局变量 bar 时，虽然 foo 函数已经执行结束，但是 getName 和 setName 函数依然可以使用 foo 函数中的变量 myName 和 test1。

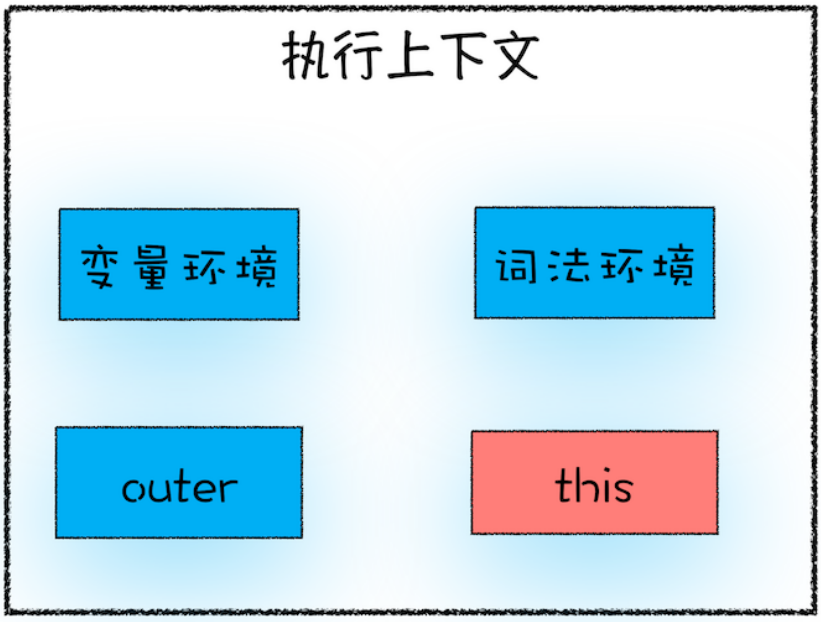
foo 函数执行完成之后，其执行上下文从栈顶弹出了，但是由于返回的 setName 和 getName 方法中使用了 foo 函数内部的变量 myName 和 test1，所以这两个变量依然保存在内存中。这像极了 setName 和 getName 方法背的一个专属背包，无论在哪里调用了 setName 和 getName 方法，它们都会背着这个 foo 函数的专属背包。这个背包称为 foo 函数的闭包。

当调用 bar.getName 的时候，右边 Scope 项就体现出了作用域链的情况：Local 就是当前的 getName 函数的作用域，Closure(foo) 是指 foo 函数的闭包，最下面的 Global 就是指全局作用域，从“Local–>Closure(foo)–>Global”就是一个完整的作用域链。

闭包：在 JavaScript 中，根据词法作用域的规则，内部函数总是可以访问其外部函数中声明的变量，当通过调用一个外部函数返回一个内部函数后，即使该外部函数已经执行结束了，但是内部函数引用外部函数的变量依然保存在内存中，我们就把这些变量的集合称为闭包。比如外部函数是 foo，那么这些变量的集合就称为 foo 函数的闭包。

通常，如果引用闭包的函数是一个全局变量，那么闭包会一直存在直到页面关闭；但如果这个闭包以后不再使用的话，就会造成内存泄漏。尽量让它成为一个局部变量。

this



全局执行上下文中的 this指向 window 对象
函数执行上下文中的 this：

- 通过函数的 call ，apply，bind方法设置
- 通过对象调用方法设置
- 通过构造函数中设置 new

```
1
2 function CreateObj(){
3   this.name = "yideng"
4 }
5 var myObj = new CreateObj()
```

首先创建了一个空对象 tempObj；接着调用 CreateObj.call 方法，并将 tempObj 作为 call 方法的参数，这样当 CreateObj 的执行上下文创建时，它的 this 就指向了 tempObj 对象；然后执行 CreateObj 函数，此时的 CreateObj 函数执行上下文中的 this 指向了 tempObj 对象；最后返回 tempObj 对象。

this 的设计缺陷以及应对方案

- 嵌套函数中的 this 不会从外层函数中继承

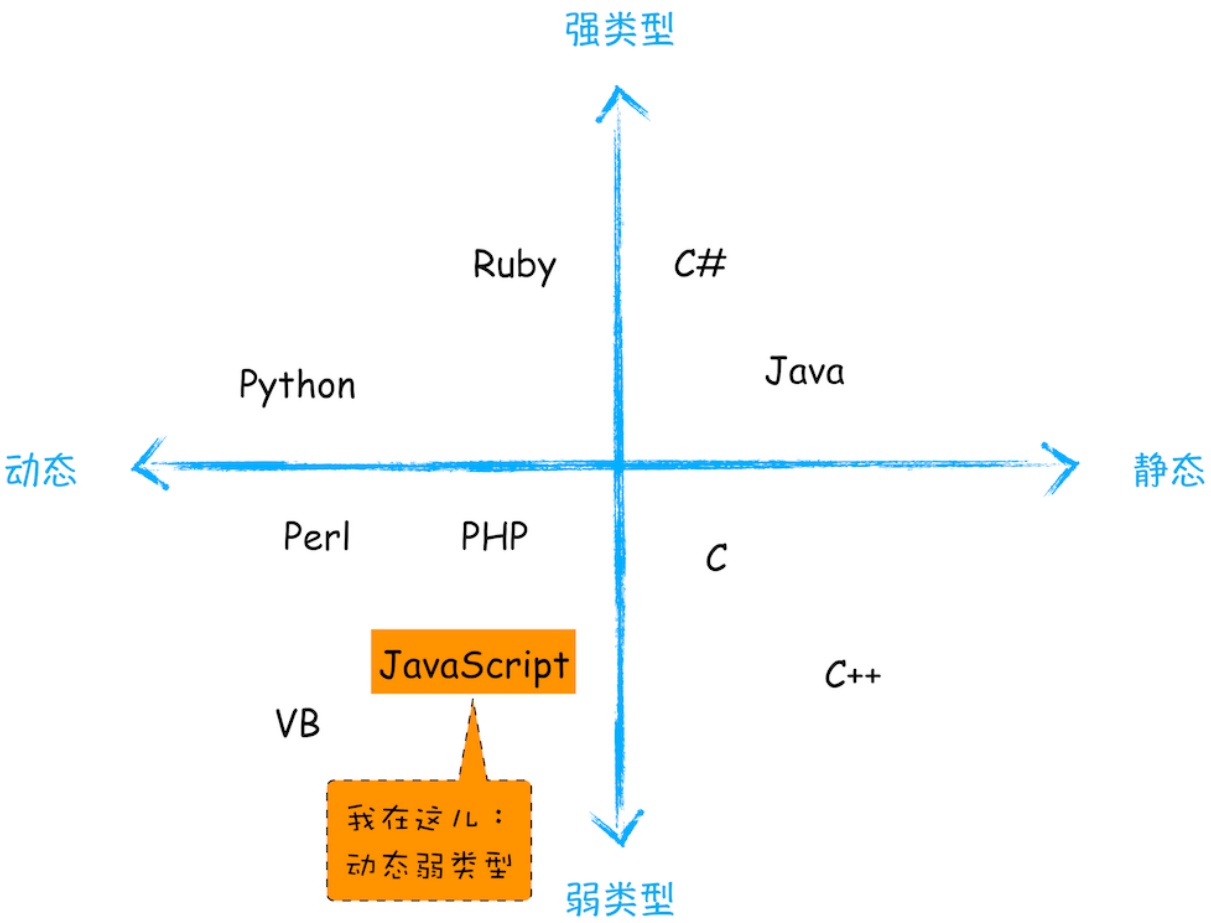
```
1
2 var myObj = {
3   name : "yideng",
4   showThis: function(){
5     console.log(this)
6     function bar(){console.log(this)}
7     bar()
8   }
9 }
10 myObj.showThis()
```

函数 bar 中的 this 指向的是全局 window 对象，而函数 showThis 中的 this 指向的是 myObj 对象。
箭头函数解决

- 普通函数中的 this 默认指向全局对象 window

在严格模式下，默认执行一个函数，其函数的执行上下文中的 this 值是 undefined

栈空间和堆空间



JavaScript 是一种弱类型的、动态的语言。
JavaScript 中的数据类型一种有 8 种，它们分别是：

类型	描述
Boolean	只有true和false两个值。
Null	只有一个值null。
Undefined	一个没有被赋值的变量会有个默认值 undefined，变量提升时的默认值也是undefined。
Number	根据 ECMAScript 标准，JavaScript 中只有一种数字类型：基于 IEEE 754 标准的双精度 64 位二进制格式的值，-(263 -1) 到 263 -1。
BigInt	JavaScript 中一个新的数字类型，可以用任意精度表示整数。使用 BigInt，即使超出 Number 的安全整数范围限制，也可以安全地存储和操作。
String	用于表示文本数据。不同于类 C 语言，JavaScript 的字符串是不可更改的。
Symbol	符号类型是唯一的并且是不可修改的，通常用来作为Object的key。
Object	在 JavaScript 里，对象可以被看作是一组属性的集合。

原始数据类型是存储在栈空间中的，引用类型的数据是存储在堆空间中的

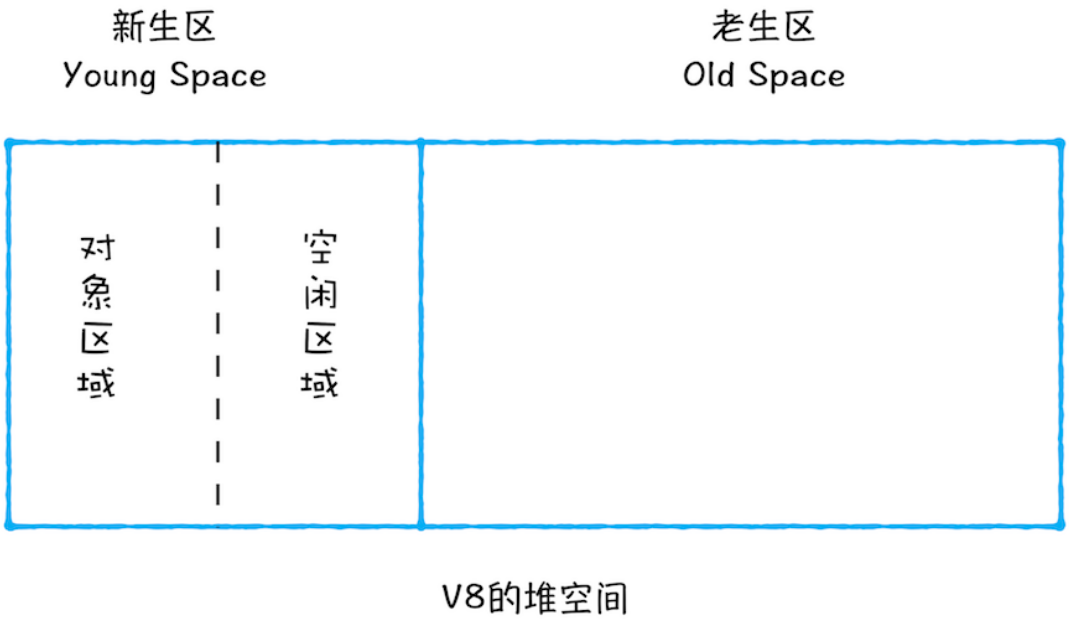
垃圾回收

JavaScript 引擎会通过向下移动 ESP 来销毁该函数保存在栈中的执行上下文。
要回收堆中的垃圾数据，就需要用到 JavaScript 中的垃圾回收器了。
代际假说和分代收集
代际假说有以下两个特点：第一个是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；第二个是不死的对象，会活得更久。
V8 中会把堆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，老生代中存放的生存时间久的对象。
副垃圾回收器，主要负责新生代的垃圾回收。主垃圾回收器，主要负责老生代的垃圾回收。
垃圾回收器的工作流程

- 第一步是标记空间中活动对象和非活动对象。所谓活动对象就是还在使用的对象，非活动对象就是可以进行垃圾回收的对象。
- 第二步是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。
- 第三步是做内存整理。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，如果需要分配较大连续内存的时候，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，比如接下来我们要介绍的副垃圾回收器

副垃圾回收器

Scavenge 算法：

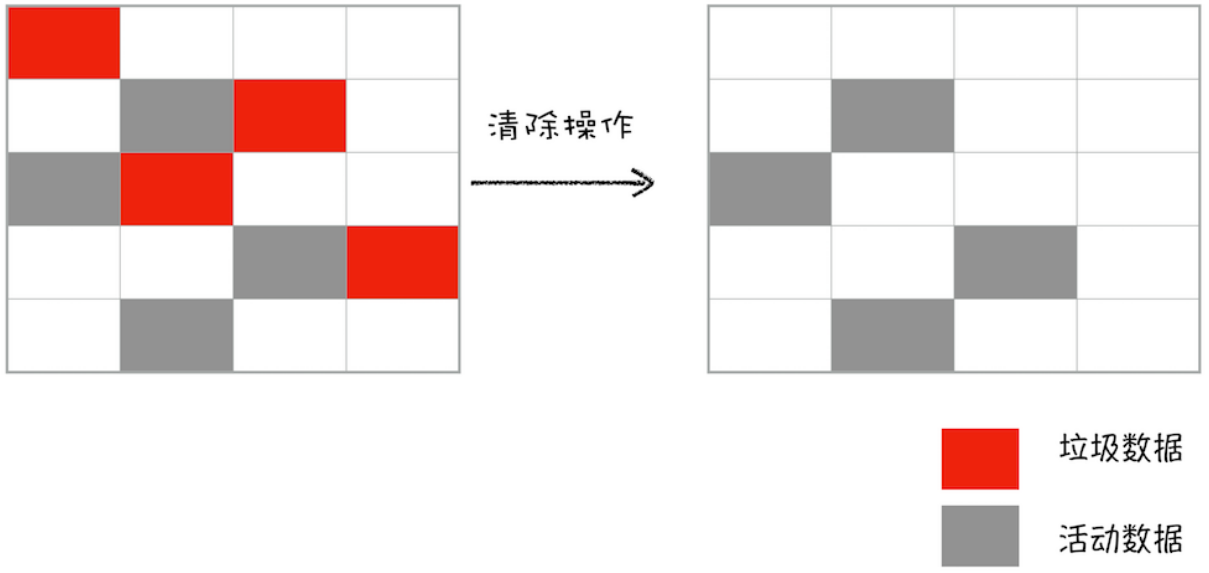


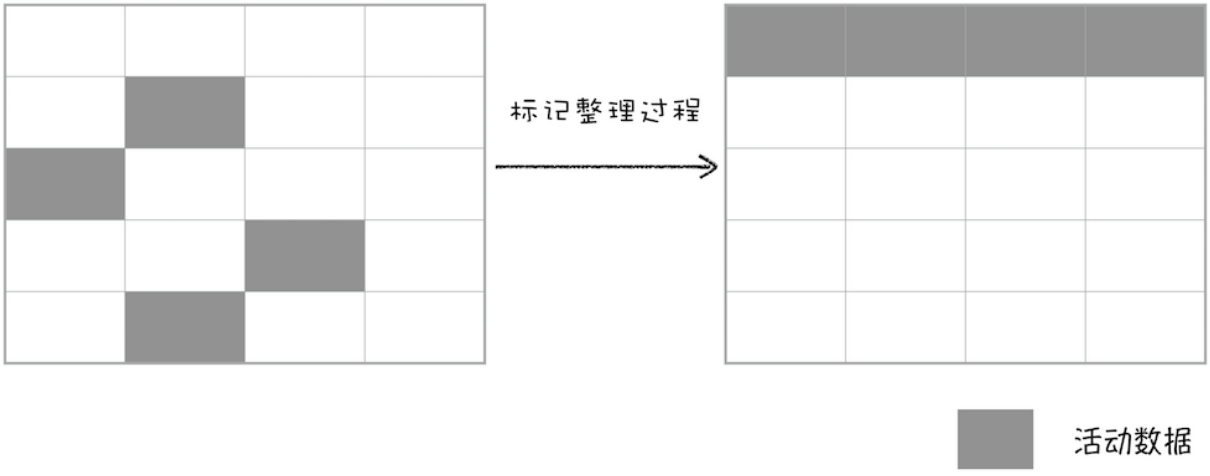
V8的堆空间

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。经过两次垃圾回收依然还存活的对象，会被移动到老生区中。

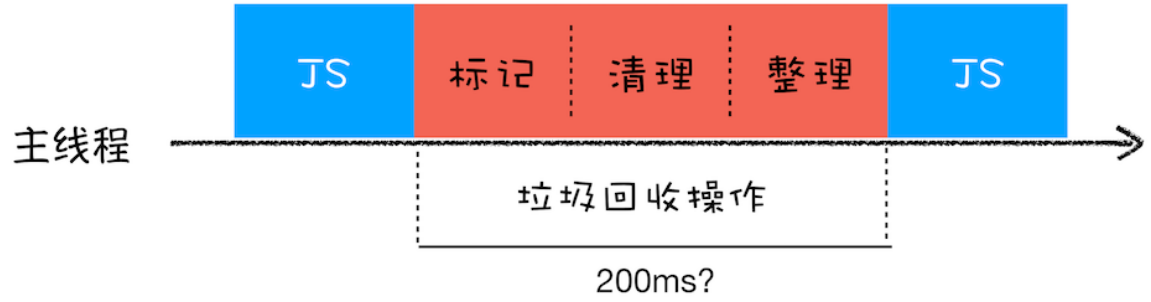
主垃圾回收器

采用标记 - 清除（Mark-Sweep）的算法进

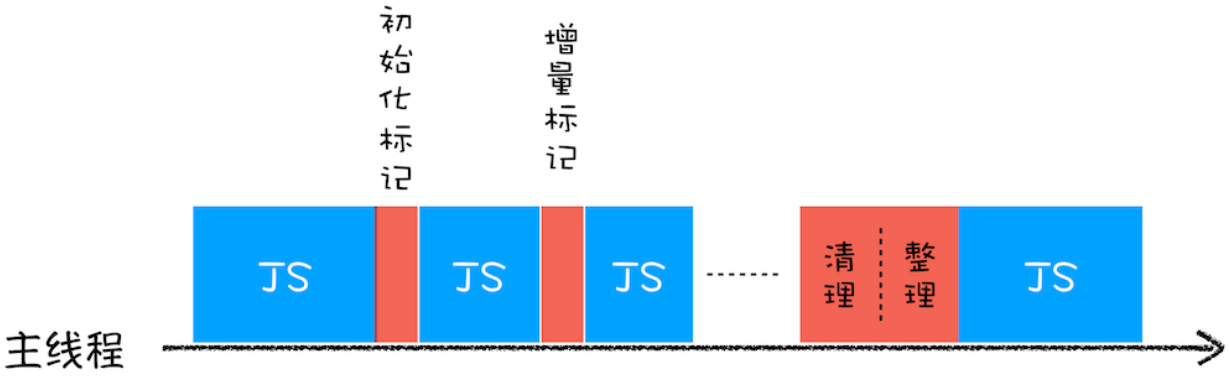




一旦执行垃圾回收算法，都需要将正在执行的 JavaScript 脚本暂停下来，待垃圾回收完毕后再恢复脚本执行。我们把这种行为叫做全停顿（Stop-The-World）。

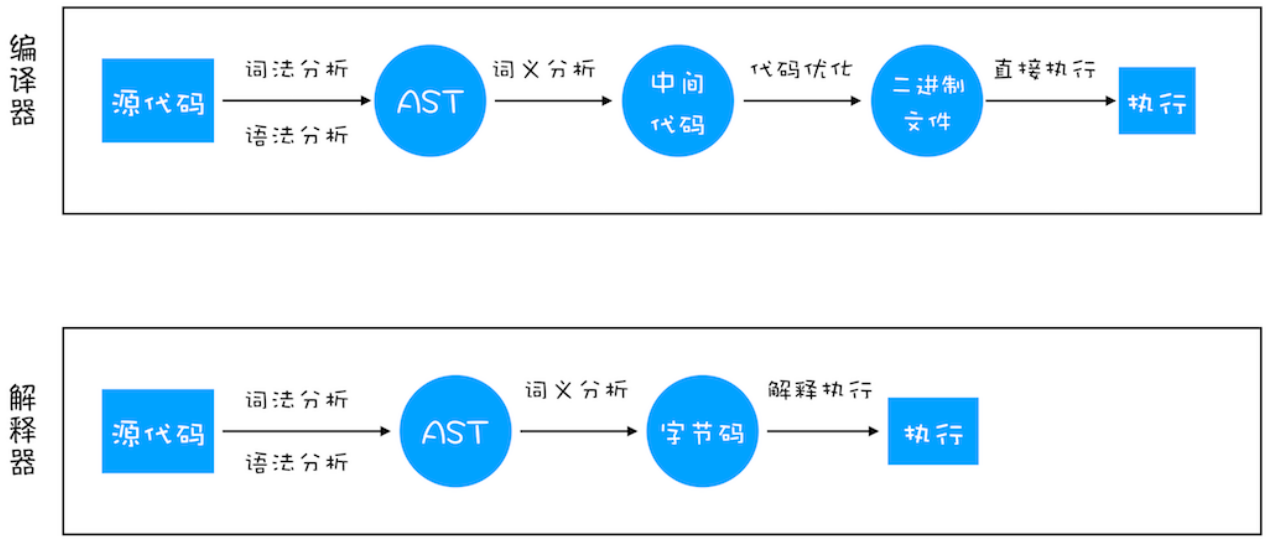


增量标记（Incremental Marking）算法



编译器和解释器

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。而由解释型语言编写的程序，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。



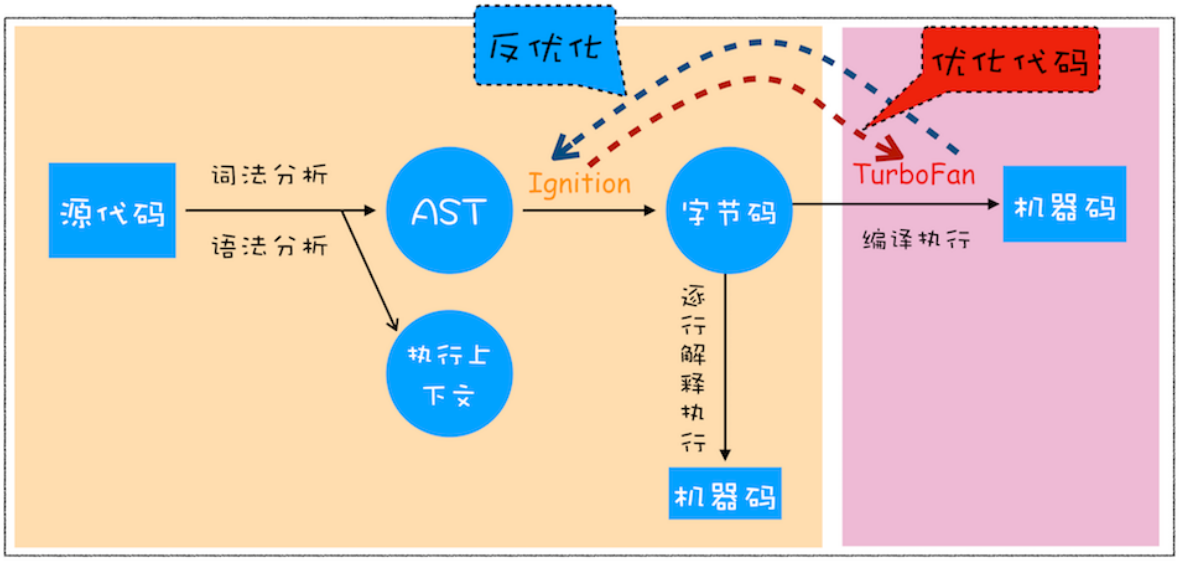
- 在编译型语言的编译过程中，编译器首先会依次对源代码进行词法分析、语法分析，生成抽象语法树（AST），然后是优化代码，最后再生成处理器能够理解的机器码。如果编译成功，将会生成一个可执行的文件。但如果编译过程发生了语法或者其他的错误，那么编译器就会抛出异常，最后的二进制文件也不会生成成功。
- 在解释型语言的解释过程中，同样解释器也会对源代码进行词法分析、语法分析，并生成抽象语法树（AST），不过它会再基于抽象语法树生成字节码，最后再根据字节码来执行程序、输出结果。

V8 是如何执行一段 JavaScript 代码的

- 将源代码转换为抽象语法树，并生成执行上下文

AST：AST 是非常重要的一种数据结构，在很多项目中有着广泛的应用。其中最著名的一个项目是 Babel。Babel 是一个被广泛使用的代码转码器，可以将 ES6 代码转为 ES5 代码，这意味着你可以现在就用 ES6 编写程序，而不用担心现有环境是否支持 ES6。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。

1. 第一阶段是分词（tokenize），又称为词法分析
2. 第二阶段是解析（parse），又称为语法分析



· 生成字节码

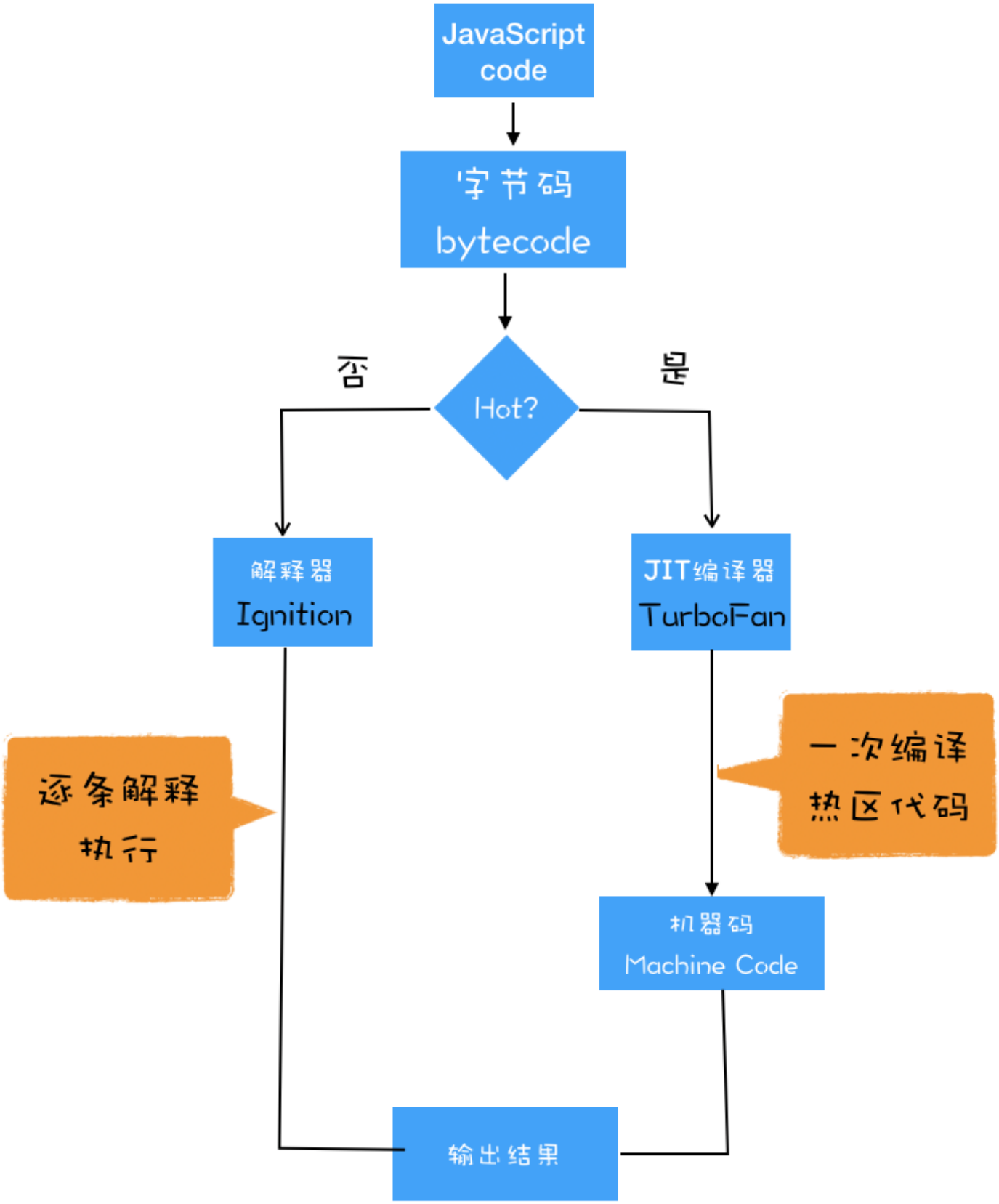
解释器 Ignition 就登场了，它会根据 AST 生成字节码，并解释执行字节码。

字节码就是介于 AST 和机器码之间的一种代码。但是与特定类型的机器码无关，字节码需要通过解释器将其转换为机器码后才能执行。

· 执行代码

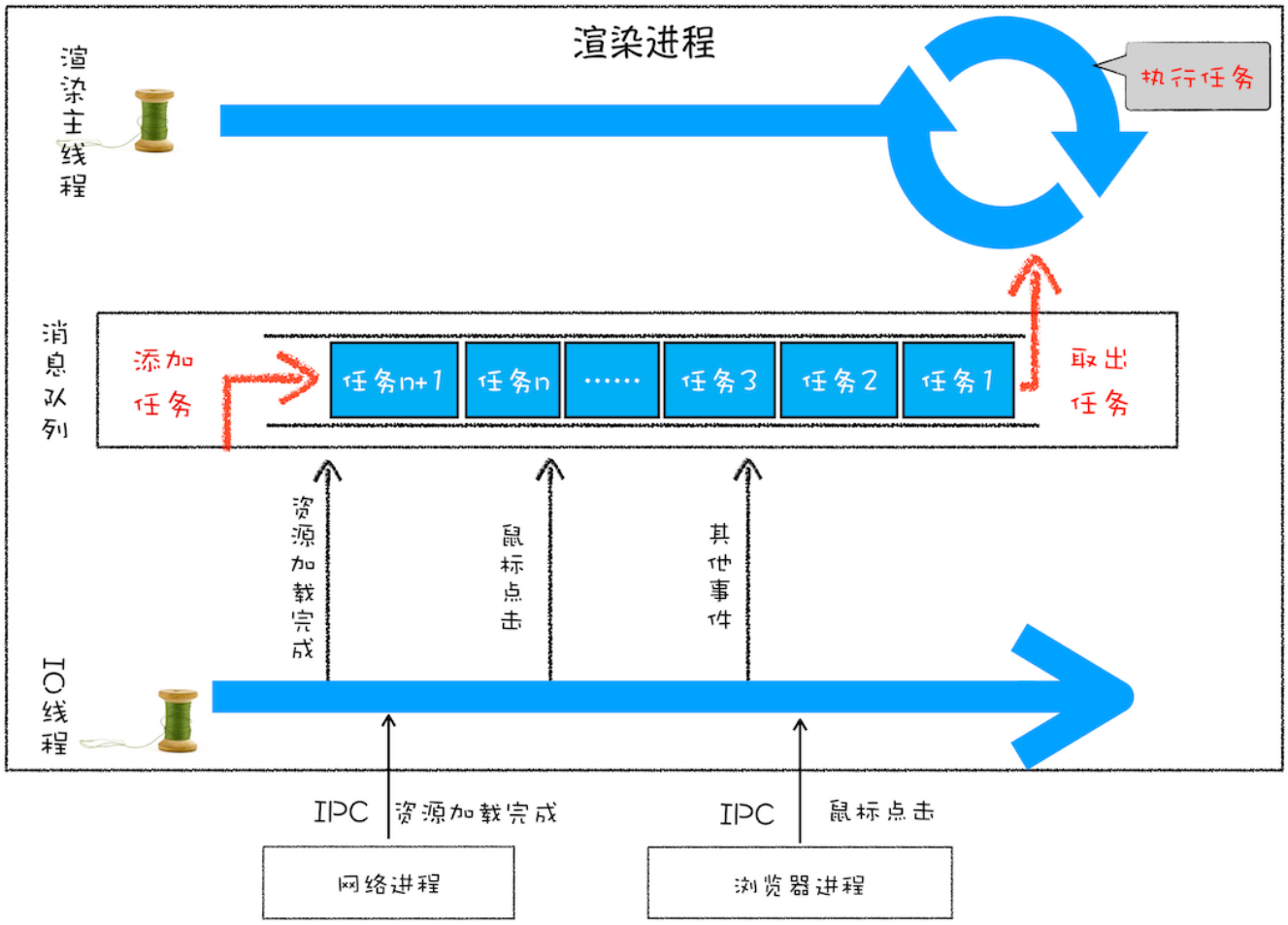
生成字节码之后，接下来就要进入执行阶段了。通常，如果有一段第一次执行的字节码，解释器 Ignition 会逐条解释执行。到了这里，相信你已经发现了，解释器 Ignition 除了负责生成字节码之外，它还有另外一个作用，就是解释执行字节码。在 Ignition 执行字节码的过程中，如果发现有热点代码（HotSpot），比如一段代码被重复执行多次，这种就称为热点代码，那么后台的编译器 TurboFan 就会把该段热点的字节码编译为高效的机器码，然后当再次执行这段被优化的代码时，只需要执行编译后的机器码就可以了，这样就大大提升了代码的执行效率。

字节码配合解释器和编译器是最近一段时间很火的技术**即时编译（JIT）**



消息队列和事件循环

要想在线程运行过程中，能接收并执行新的任务，就需要采用事件循环机制。



消息队列是一种数据结构，可以存放要执行的任务。它符合队列“先进先出”的特点，也就是说要添加任务的话，添加到队列的尾部；要取出任务的话，从队列头部去取。

消息队列：输入事件（鼠标滚动、点击、移动）、微任务、文件读写、WebSocket、JavaScript 定时器等等。除此之外，消息队列中还包含了很多与页面相关的事件，如 JavaScript 执行、解析 DOM、样式计算、布局计算、CSS 动画等。

页面使用单线程的缺点

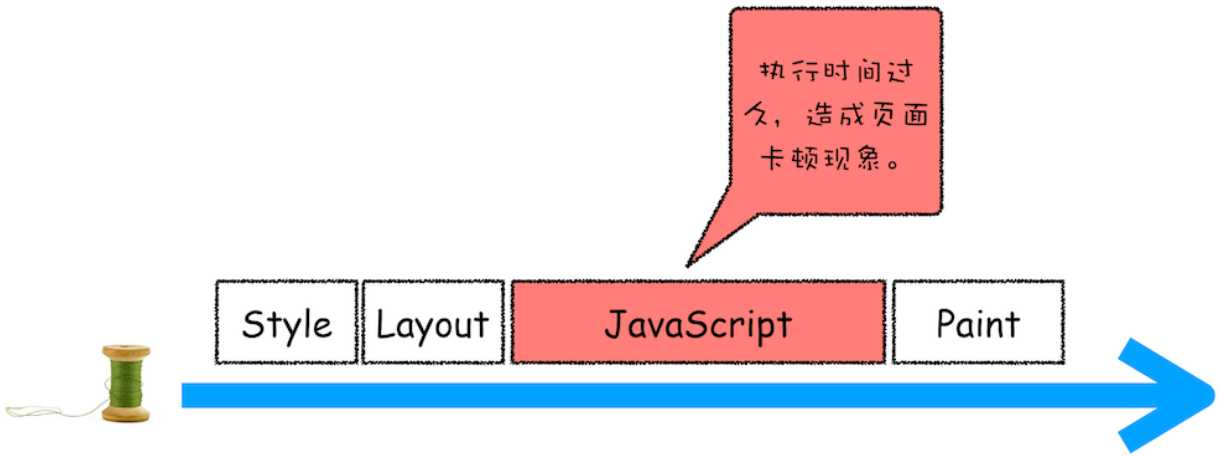
页面线程所有执行的任务都来自于消息队列。消息队列是“先进先出”的属性，也就是说放入队列中的任务，需要等待前面的任务被执行完，才会被执行。

如何处理高优先级的任务

如果 DOM 发生变化，采用同步通知的方式，会影响当前任务的执行效率；如果采用异步方式，又会影响到监控的实时性。

通常我们把消息队列中的任务称为宏任务，每个宏任务中都包含了一个微任务队列，在执行宏任务的过程中，如果 DOM 有变化，那么就会将该变化添加到微任务列表中，这样就不会影响到宏任务的继续执行，因此也就解决了执行效率的问题。

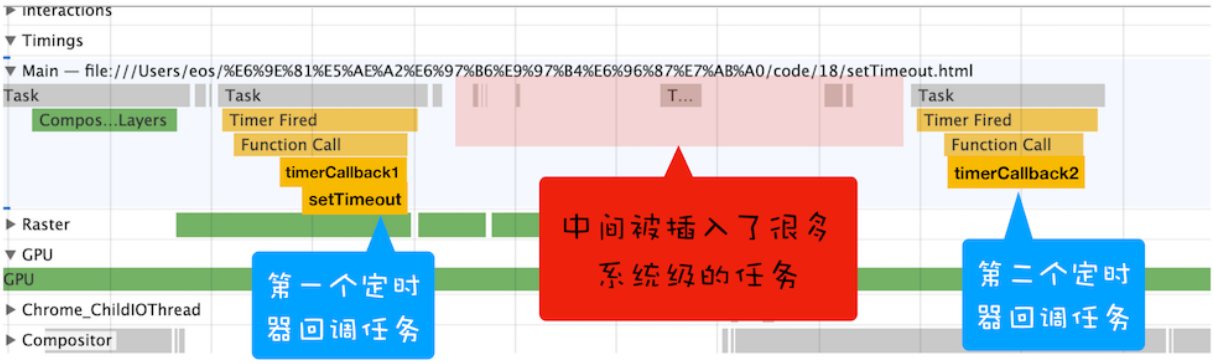
单个任务执行时长过久的问题



如果在执行动画过程中，其中有个 JavaScript 任务因执行时间过久，占用了动画单帧的时间，这样会给用户制造了卡顿的感觉，这当然是极不好的用户体验。针对这种情况，JavaScript 可以通过回调功能来规避这种问题，也就是让要执行的 JavaScript 任务滞后执行。

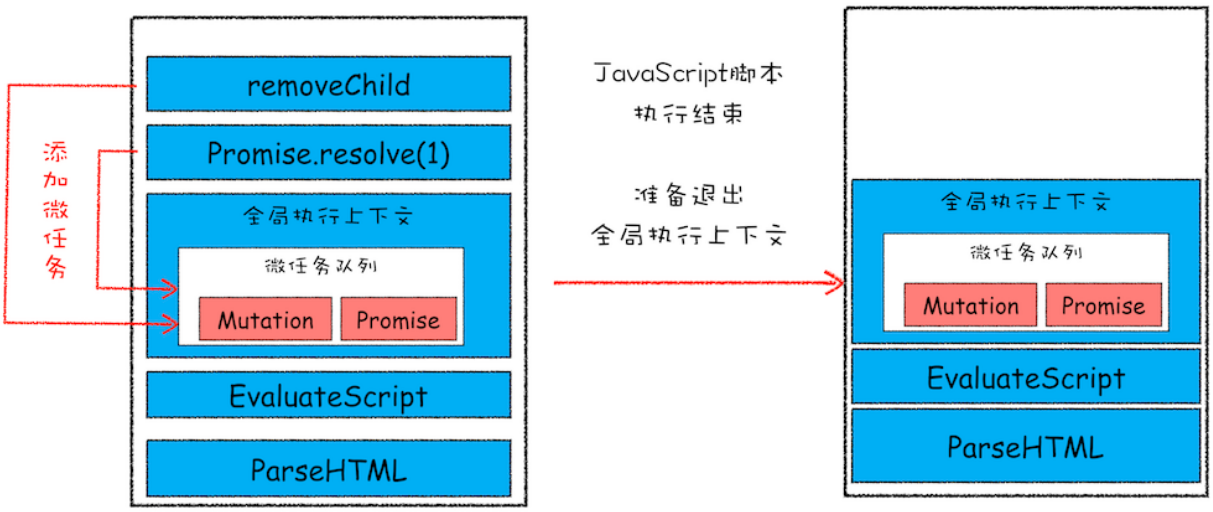
宏任务

- 渲染事件（如解析 DOM、计算布局、绘制）；
- 用户交互事件（如鼠标点击、滚动页面、放大缩小等）；
- JavaScript 脚本执行事件；
- 网络请求完成、文件读写完成事件。



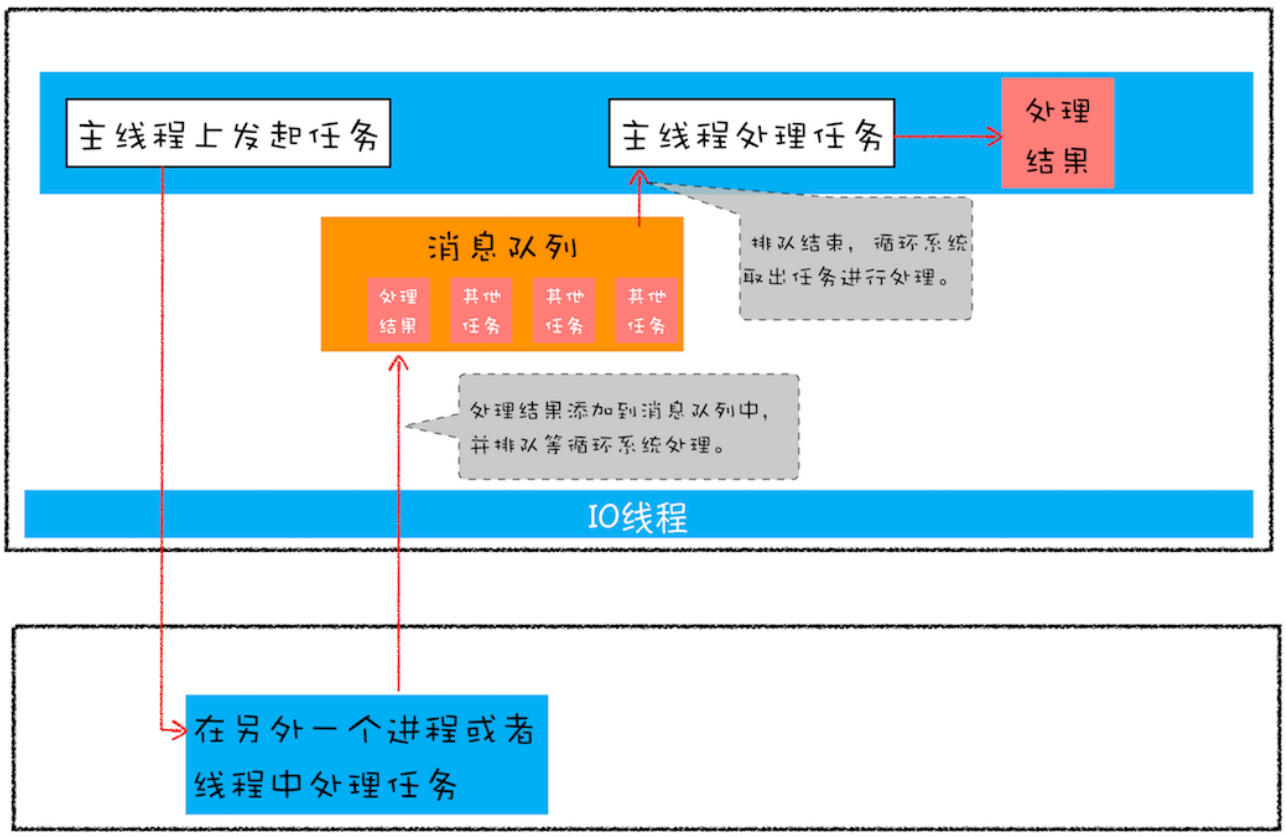
微任务

微任务就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。
当 JavaScript 执行一段脚本的时候，V8 会为其创建一个全局执行上下文，在创建全局执行上下文的同时，V8 引擎也会在内部创建一个微任务队列。
MutationObserver、Promise

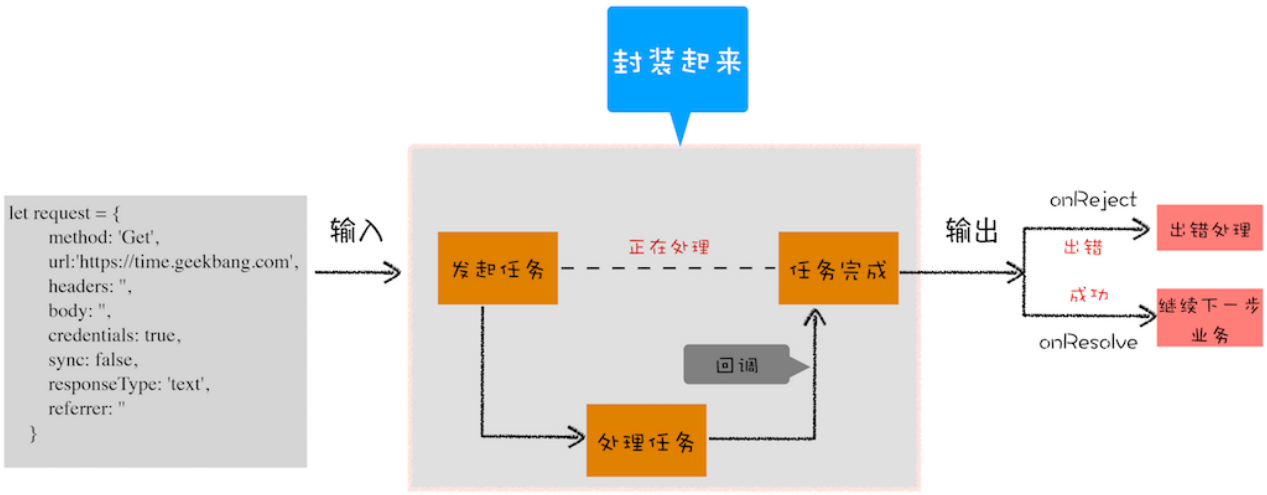


Promise

异步编程模型



封装异步代码，让处理流程变得线性



Promise：消灭嵌套调用和多次错误处理
1、为什么要引入微任务？

由于promise采用.then延时绑定回调机制，而new Promise时又需要直接执行promise中的方法，即发生了先执行方法后添加回调的过程，此时需等待then方法绑定两个回调后才能继续执行方法回调，便可将回调添加到当前js调用栈中执行结束后的任务队列中，由于宏任务较多容易堵塞，则采用了微任务

2、Promise 是如何实现回调函数返回值穿透的？
首先Promise的执行结果保存在promise的data变量中，然后是.then方法返回值为使用resolved或rejected回调方法新建的一个promise对象，即例如成功则返回new Promise（resolved），将前一个promise的data值赋给新建的promise

3、Promise 出错后，是怎么通过“冒泡”传递给最后那个捕获
promise内部有resolved_和rejected_变量保存成功和失败的回调，进入.then（resolved，rejected）时会判断rejected参数是否为函数，若是函数，错误时使用rejected处理错误；若不是，则错误时直接throw错误，一直传递到最后的捕获，若最后没有被捕获，则会报错。可通过监听unhandledrejection事件捕获未处理的promise错误

async/await：使用同步的方式去写异步代码

生成器 VS 协程