## 前言

作为一名 JavaScript 开发者，知道 JavaScript 程序内部的执行机制，理解执行上下文和执行栈是有助于理解其他的 JavaScript 概念如提升机制、作用域和闭包等。本文尽可能图文并茂，通俗易懂的方式来讲解执行上下文相关知识点。

### 什么是执行上下文

简而言之，执行上下文就是当前 JavaScript 代码被解析和执行时所在环境的抽象概念。这里我们简单拆分下【执行】【上下文】，上下文，我们在高中语文或者英语阅读理解的时候经常听到老师说：根据上下文理解，我们知道xxxxxxx。老师说的上下文是文章上下段落，类比的话，我们可以理解程序中的上下文是代码段或者代码段所处的环境。另外，JavaScript 中运行任何的代码都是在执行上下文中运行。

### 执行上下文的类型

执行上下文有三种类型：

1. **全局执行上下文：**不在任何函数中的代码都位于全局执行上下文中。全局执行上下文完成了两件事：1.创建一个全局对象，在浏览器中这个全局对象就是window对象。2.将this指向window对象。在网页中，只能存在一个全局执行上下文。
2. **函数执行上下文：**每次调用函数的时候，都会为该函数创建一个新的执行上下文。值的注意的是，每个函数都拥有自己的执行上下文，但是只有在函数被调用的时候才会被创建。理论上，一个程序中可以存在任意数量的函数执行上下文，前提是不存在函数没终止条件的无限递归调用自己，造成堆栈溢出最大值。函数执行上下的执行是有顺序的，具体在下文讨论。
3. **eval函数执行上下文：** 由于我们不常用eval函数，所以该类型不讨论

## 执行栈

在之前的《内存空间中的数据结构》文章中，我们已经讲述了栈的特点是：后进先出。同样的执行栈就是该特点，先调用的函数就会先入栈，之后调用的函数再跟着入栈，函数调用完之后再出栈，最先入栈的函数（最先调用的函数）反而最后出栈。从执行栈的角度来讲就是：当 JavaScript 引擎首次读取你的脚本时，它会创建一个全局执行上下文并将其推入当前的执行栈。每当发生一个函数调用，引擎都会为该函数创建一个新的执行上下文并将其推到当前执行栈的顶端。 引擎会运行执行上下文在执行栈顶端的函数，当此函数运行完成后，其对应的执行上下文将会从执行栈中弹出，上下文控制权将移到当前执行栈的下一个执行上下文。

说了这么多还是直接上代码形象点。

let a = 'Hello World!';

function first() {

console.log('Inside first function');

second();

console.log('Again inside first function');

}

function second() {

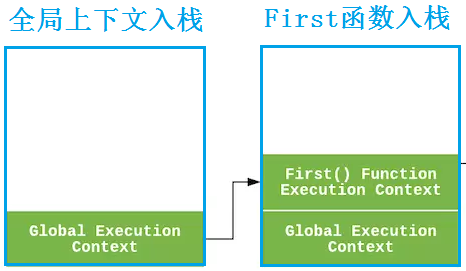
console.log('Inside second function');

}

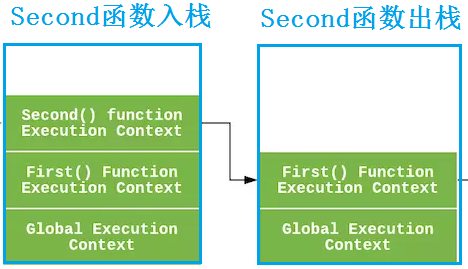
first();

console.log('Inside Global Execution Context');

上述代码，从执行栈的角度来看，流程如下：



当上述代码在浏览器中加载时，JavaScript 引擎会创建一个全局执行上下文并且将它推入当前的执行栈。当调用 first() 函数时，JavaScript 引擎为该函数创建了一个新的执行上下文并将其推到当前执行栈的顶端。

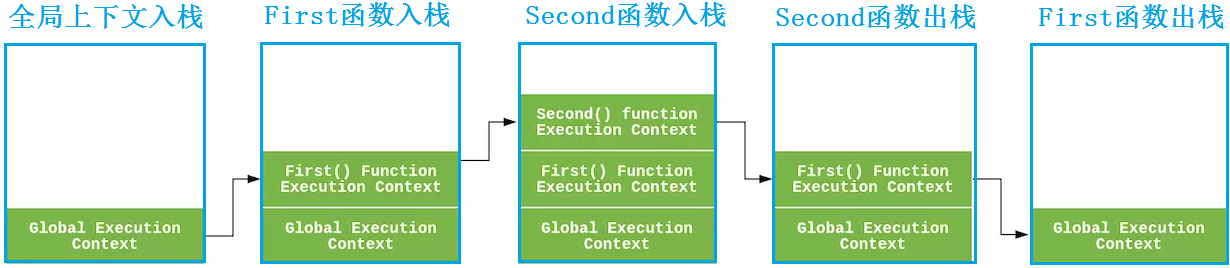


当在 first() 函数中调用 second() 函数时，Javascript 引擎为该函数创建了一个新的执行上下文并将其推到当前执行栈的顶端。当 second() 函数执行完成后，它的执行上下文从当前执行栈中弹出，上下文控制权将移到当前执行栈的下一个执行上下文，即 first() 函数的执行上下文。



当 first() 函数执行完成后，它的执行上下文从当前执行栈中弹出，上下文控制权将移到全局执行上下文。一旦所有代码执行完毕，Javascript 引擎把全局执行上下文从执行栈中移除。

整个流程：



全局上下文在浏览器窗口关闭后出栈。注意：函数中，遇到return能直接终止可执行代码的执行，会直接将当前上下文弹出栈。

根据上述的代码和流程图，我们可以对执行上下文做一些总结：

1. 执行上下文的流程是单线程的。（javascript是单线程的）
2. 执行方式是同步的。只有栈顶的上下文处于执行中，其它上下文需要等待。
3. 全局上下文在程序中只有唯一一个，并且在关闭浏览器时出栈
4. 函数的执行上下文并没有个数限制（前提是函数不要没终止条件的无限递归调用自己）
5. 每个函数被调的时候，该函数的新的执行上下文才会被创建，即使是调用函数自身也会创建新的执行上下文。

## 执行上下文的生命周期

经过上述讲解，我们知道当调用一个函数的时候，一个新的上下文就会被创建，接下来，本文将讲解执行上下文的生命周期。该生命周期主要分为三部分：创建、代码执行、执行完毕出栈

### 创建阶段

在创建阶段，执行上下文会：

1. 创建变量对象（变量对象概念下文会详解）
2. 建立作用域链
3. 确定this指向

### 代码执行阶段

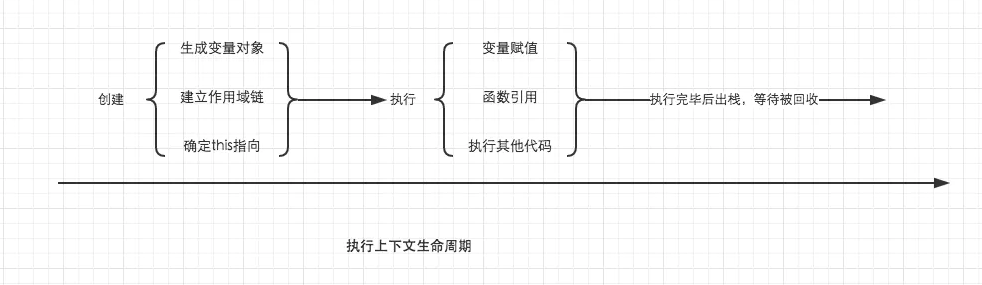
创建阶段之后，就会进入代码执行阶段：

1. 完成变量赋值
2. 函数引用
3. 执行其他代码

### 执行完毕出栈

这个阶段主要是，函数执行完之后，出栈等待被回收。

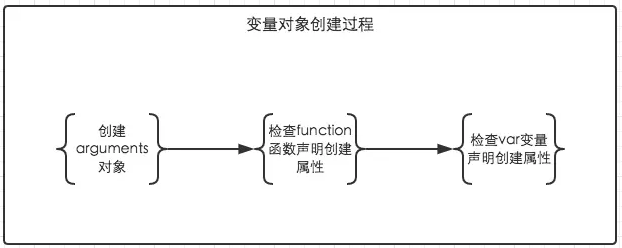
**整体流程为：**



从整个执行上下文流程上看，其中涉及了很多极为重要的概念：变量对象、作用域链、this指向等。本文主要讲解变量对象。作用域链、this指向会在后面系列文章讲解。

## 详解变量对象

变量对象其实就是一个对象，存储了函数参数、function关键字声明的函数、声明的变量。我们已经知道了，执行上下文的创建会生成变量对象，但是变量对象生成时会经历以下过程。



1. 初始化生成arguments对象：检查当前上下文中的参数，建立该对象下的属性与属性值。
2. 检查当前上下文中是否有使用function关键字声明的函数。在变量对象中以函数名建立一个属性，属性值为指向该函数所在内存地址的引用
3. 检查当前上下文中的var的变量声明，每找到一个变量声明，就在变量对象中以变量名建立一个属性，属性值为undefined。（变量提升）
4. 上述的属性理解为key，属性值理解为value。{key：value}，并且如果变量与函数同名了，则取函数的属性值。（函数优先级高于变量）

示例：

console.log(foo); // function foo

function foo() { console.log('function foo') }

var foo = 20;

为了更好的理解变量对象，我们换个角度，从执行上下文的角度看变量对象从请看下列示例1：

function test() {

console.log(a);

console.log(foo());

var a = 1;

function foo() {

return 2;

}

}

test();

在上述代码中，当全局作用域运行到test()时，test()的执行上下文开始创建。

**执行上下文创建过程**

testEC = {

VO： {

arguments: {...},

foo: <foo reference> // 表示foo的地址引用

a: undefined

}，

this: Window

}

上述伪代码中：

testEC ：表示test函数的执行栈

VO：是Variable Object的缩写，即变量对象

值的注意的是，未进入执行阶段之前，变量对象中的属性都是不能访问的，只有进入执行阶段之后，变量对象才会转变为活动对象，里面的属性都能被访问，然后开始执行阶段的：完成变量赋值、函数引用、执行其他代码。

**执行上下文的执行阶段**

VO -> AO // Active Object

AO = {

arguments: {...},

foo: <foo reference>,

a: 1,

this: Window

}

所以，上述示例1代码执行顺序变成：

function test() {

function foo() {

return 2;

}

var a;

console.log(a);

console.log(foo());

a = 1;

}

test();

## 全局上下文的变量对象

在浏览器中，全局执行上下文的变量对象比较特殊，就是window对象。而且，this的指向也是window。

伪代码：

windowEC = {

VO: Window,

scopeChain: {},

this: Window

}

只要浏览器窗口不关闭，全局上下文就会一直存在。

## let/const

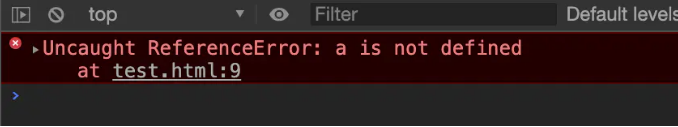
在上述中，var声明的变量会变量提升，然后把undefined赋值给变量，那么let、const声明的变量也会提升吗？答案是不会。

看示例：

我们先使用一个未定义的变量

console.log(a)

直接运行，控制台打印：



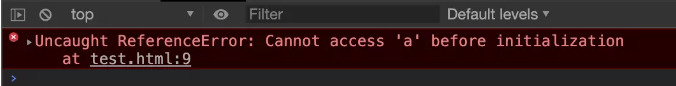
表示变量未定义。

如果我们调用后，才用let定义变量

console.log(a)

let a = 10

则



let/const声明的变量，仍然会提前被收集到变量对象中，但和var不同的是，let/const定义的变量，不会在这个时候给他赋值undefined。因为完全没有赋值，即使变量提升了，我们也不能在赋值之前调用他。这就是我们常说的暂时性死区。

最后，通过执行上下文的分析，相信对我们理解变量提升、函数执行等重要概念，有了新的看知识角度。如果要减少变量提升对代码造成的负面影响，就要保持好的开发习惯，尽量把变量声明放最前面写。