**理解 JavaScript 中的执行上下文和执行栈**

[技术漫谈](javascript:void(0);) 7月3日

**编者荐语：**

这篇文章让我受益匪浅，推荐给你。

以下文章来源于前端桃园 ，作者桃翁

**[前端桃园](https://mp.weixin.qq.com/s?__biz=MzU1NTgxMDYxMw==&mid=2247483992&idx=1&sn=2c0b669557bec428072dccb289df8ebe&chksm=fbcfe802ccb86114fd2acf1de1949a0764eb02c41dfe065e56ed7636c0421295e235f8acda1c&scene=126&sessionid=1597629603&key=8c6fdd3cc892e824692415869f85949a5e9b3c4ca2a90fce7e84a4bcf3eae0d09d42b4eae72f6b28af6be51fd25bea6afd51b423249f5d984a8a6380d0f42ce7a4335fda923e954847aca2e97a068ae495258410bf6d60ed7d916b9d1347dab72a0f775fd47d09eb804959d27494edefbb6e1d1901c3af4d09789bc7a141f08f&ascene=1&uin=MjY1OTAzOTAxNA%3D%3D&devicetype=Windows+10+x64&version=62090070&lang=zh_CN&exportkey=AQGKV0t3y5um0S43yIm3zvI%3D&pass_ticket=%2Fh0GAhbGoTHSY16HNuKn0FhmrkC4nvwRD5HwxINIAjBQDl5UNysckp03drELaW%2FA&winzoom=1)**

[一个有温度的前端号，不止前端，互联网前沿知识，个人随想，认知提升，学习方法，成为你想成为的那个样子](https://mp.weixin.qq.com/s?__biz=MzU1NTgxMDYxMw==&mid=2247483992&idx=1&sn=2c0b669557bec428072dccb289df8ebe&chksm=fbcfe802ccb86114fd2acf1de1949a0764eb02c41dfe065e56ed7636c0421295e235f8acda1c&scene=126&sessionid=1597629603&key=8c6fdd3cc892e824692415869f85949a5e9b3c4ca2a90fce7e84a4bcf3eae0d09d42b4eae72f6b28af6be51fd25bea6afd51b423249f5d984a8a6380d0f42ce7a4335fda923e954847aca2e97a068ae495258410bf6d60ed7d916b9d1347dab72a0f775fd47d09eb804959d27494edefbb6e1d1901c3af4d09789bc7a141f08f&ascene=1&uin=MjY1OTAzOTAxNA%3D%3D&devicetype=Windows+10+x64&version=62090070&lang=zh_CN&exportkey=AQGKV0t3y5um0S43yIm3zvI%3D&pass_ticket=%2Fh0GAhbGoTHSY16HNuKn0FhmrkC4nvwRD5HwxINIAjBQDl5UNysckp03drELaW%2FA&winzoom=1)

**什么是执行上下文？**

简而言之，执行上下文是评估和执行 JavaScript 代码的环境的抽象概念。每当 Javascript 代码在运行的时候，它都是在执行上下文中运行。

**执行上下文的类型**

JavaScript 中有三种执行上下文类型。

* **全局执行上下文** — 这是默认或者说基础的上下文，任何不在函数内部的代码都在全局上下文中。它会执行两件事：创建一个全局的 window 对象（浏览器的情况下），并且设置this 的值等于这个全局对象。一个程序中只会有一个全局执行上下文。
* **函数执行上下文** — 每当一个函数被调用时, 都会为该函数创建一个新的上下文。每个函数都有它自己的执行上下文，不过是在函数被调用时创建的。函数上下文可以有任意多个。每当一个新的执行上下文被创建，它会按定义的顺序（将在后文讨论）执行一系列步骤。
* **Eval 函数执行上下文** — 执行在 eval 函数内部的代码也会有它属于自己的执行上下文，但由于 JavaScript 开发者并不经常使用 eval，所以在这里我不会讨论它。

**执行栈**

执行栈，也就是在其它编程语言中所说的“调用栈”，是一种拥有 LIFO（后进先出）数据结构的栈，被用来存储代码运行时创建的所有执行上下文。

当 JavaScript 引擎第一次遇到你的脚本时，它会创建一个全局的执行上下文并且压入当前执行栈。每当引擎遇到一个函数调用，它会为该函数创建一个新的执行上下文并压入栈的顶部。

引擎会执行那些执行上下文位于栈顶的函数。当该函数执行结束时，执行上下文从栈中弹出，控制流程到达当前栈中的下一个上下文。

让我们通过下面的代码示例来理解：

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');

上述代码的执行上下文栈。

当上述代码在浏览器加载时，JavaScript 引擎创建了一个全局执行上下文并把它压入当前执行栈。当遇到 first() 函数调用时，JavaScript 引擎为该函数创建一个新的执行上下文并把它压入当前执行栈的顶部。

当从 first() 函数内部调用 second() 函数时，JavaScript 引擎为 second() 函数创建了一个新的执行上下文并把它压入当前执行栈的顶部。当 second() 函数执行完毕，它的执行上下文会从当前栈弹出，并且控制流程到达下一个执行上下文，即 first() 函数的执行上下文。

当 first() 执行完毕，它的执行上下文从栈弹出，控制流程到达全局执行上下文。一旦所有代码执行完毕，JavaScript 引擎从当前栈中移除全局执行上下文。

**怎么创建执行上下文？**

到现在，我们已经看过 JavaScript 怎样管理执行上下文了，现在让我们了解 JavaScript 引擎是怎样创建执行上下文的。

创建执行上下文有两个阶段：**1) 创建阶段** 和 **2) 执行阶段**。

**The Creation Phase**

在 JavaScript 代码执行前，执行上下文将经历创建阶段。在创建阶段会发生三件事：

1. **this** 值的决定，即我们所熟知的 **This 绑定**。
2. 创建**词法环境**组件。
3. 创建**变量环境**组件。

所以执行上下文在概念上表示如下：

ExecutionContext = {  
  ThisBinding = <this value>,  
  LexicalEnvironment = { ... },  
  VariableEnvironment = { ... },  
}

**This 绑定：**

在全局执行上下文中，this 的值指向全局对象。(在浏览器中，this引用 Window 对象)。

在函数执行上下文中，this 的值取决于该函数是如何被调用的。如果它被一个引用对象调用，那么 this 会被设置成那个对象，否则 this 的值被设置为全局对象或者undefined（在严格模式下）。例如：

let foo = {  
  baz: function() {  
  console.log(this);  
  }  
}  
foo.baz();   // 'this' 引用 'foo', 因为 'baz' 被  
             // 对象 'foo' 调用  
let bar = foo.baz;  
bar();       // 'this' 指向全局 window 对象，因为  
             // 没有指定引用对象

**词法环境**

官方的 ES6 文档把词法环境定义为

**词法环境**是一种规范类型，基于 ECMAScript 代码的词法嵌套结构来定义**标识符**和具体变量和函数的关联。一个词法环境由环境记录器和一个可能的引用**外部**词法环境的空值组成。

简单来说**词法环境**是一种持有**标识符—变量映射**的结构。（这里的**标识符**指的是变量/函数的名字，而**变量**是对实际对象[包含函数类型对象]或原始数据的引用）。

现在，在词法环境的**内部**有两个组件：(1) **环境记录器**和 (2) 一个**外部环境的引用**。

1. **环境记录器**是存储变量和函数声明的实际位置。
2. **外部环境的引用**意味着它可以访问其父级词法环境（作用域）。

**词法环境**有两种类型：

* **全局环境**（在全局执行上下文中）是没有外部环境引用的词法环境。全局环境的外部环境引用是 **null**。它拥有内建的 Object/Array/等、在环境记录器内的原型函数（关联全局对象，比如 window 对象）还有任何用户定义的全局变量，并且 this的值指向全局对象。
* 在**函数环境**中，函数内部用户定义的变量存储在**环境记录器**中。并且引用的外部环境可能是全局环境，或者任何包含此内部函数的外部函数。

**环境记录器**也有两种类型（如上！）：

1. **声明式环境记录器**存储变量、函数和参数。
2. **对象环境记录器**用来定义出现在**全局上下文**中的变量和函数的关系。

简而言之，

* 在**全局环境**中，环境记录器是对象环境记录器。
* 在**函数环境**中，环境记录器是声明式环境记录器。

**注意 —** 对于**函数环境**，**声明式环境记录器**还包含了一个传递给函数的 arguments 对象（此对象存储索引和参数的映射）和传递给函数的参数的 **length**。

抽象地讲，词法环境在伪代码中看起来像这样：

GlobalExectionContext = {  
  LexicalEnvironment: {  
    EnvironmentRecord: {  
      Type: "Object",  
      // 在这里绑定标识符  
    }  
    outer: <null>  
  }  
}  
FunctionExectionContext = {  
  LexicalEnvironment: {  
    EnvironmentRecord: {  
      Type: "Declarative",  
      // 在这里绑定标识符  
    }  
    outer: <Global or outer function environment reference>  
  }  
}

**变量环境：**

它同样是一个词法环境，其环境记录器持有**变量声明语句**在执行上下文中创建的绑定关系。

如上所述，变量环境也是一个词法环境，所以它有着上面定义的词法环境的所有属性。

在 ES6 中，**词法环境**组件和**变量环境**的一个不同就是前者被用来存储函数声明和变量（let和 const）绑定，而后者只用来存储 var 变量绑定。

我们看点样例代码来理解上面的概念：

let a = 20;  
const b = 30;  
var c;  
function multiply(e, f) {  
 var g = 20;  
 return e \* f \* g;  
}  
c = multiply(20, 30);

执行上下文看起来像这样：

GlobalExectionContext = {  
  ThisBinding: <Global Object>,  
  LexicalEnvironment: {  
    EnvironmentRecord: {  
      Type: "Object",  
      // 在这里绑定标识符  
      a: < uninitialized >,  
      b: < uninitialized >,  
      multiply: < func >  
    }  
    outer: <null>  
  },  
  VariableEnvironment: {  
    EnvironmentRecord: {  
      Type: "Object",  
      // 在这里绑定标识符  
      c: undefined,  
    }  
    outer: <null>  
  }  
}  
FunctionExectionContext = {  
  ThisBinding: <Global Object>,  
  LexicalEnvironment: {  
    EnvironmentRecord: {  
      Type: "Declarative",  
      // 在这里绑定标识符  
      Arguments: {0: 20, 1: 30, length: 2},  
    },  
    outer: <GlobalLexicalEnvironment>  
  },  
  VariableEnvironment: {  
    EnvironmentRecord: {  
      Type: "Declarative",  
      // 在这里绑定标识符  
      g: undefined  
    },  
    outer: <GlobalLexicalEnvironment>  
  }  
}

**注意** — 只有遇到调用函数 multiply 时，函数执行上下文才会被创建。

可能你已经注意到 let 和 const 定义的变量并没有关联任何值，但 var 定义的变量被设成了 undefined。

这是因为在创建阶段时，引擎检查代码找出变量和函数声明，虽然函数声明完全存储在环境中，但是变量最初设置为 undefined（var 情况下），或者未初始化（let 和 const 情况下）。

这就是为什么你可以在声明之前访问 var 定义的变量（虽然是 undefined），但是在声明之前访问 let 和 const 的变量会得到一个引用错误。

这就是我们说的变量声明提升。

**执行阶段**

这是整篇文章中最简单的部分。在此阶段，完成对所有这些变量的分配，最后执行代码。

**注意** — 在执行阶段，如果 JavaScript 引擎不能在源码中声明的实际位置找到 let 变量的值，它会被赋值为 undefined。

**结论**

我们已经讨论过 JavaScript 程序内部是如何执行的。虽然要成为一名卓越的 JavaScript 开发者并不需要学会全部这些概念，但是如果对上面概念能有不错的理解将有助于你更轻松，更深入地理解其他概念，如变量声明提升，作用域和闭包。

* 原文地址：Understanding Execution Context and Execution Stack in Javascript
* 原文作者：Sukhjinder Arora
* 译文出自：掘金翻译计划
* 本文永久链接：https://github.com/xitu/gold-miner/blob/master/TODO1/understanding-execution-context-and-execution-stack-in-javascript.md
* 译者：CoolRice
* 校对者：linxuesia, CoderMing