作用域与作用域链本应该是很简单的概念，但是当用此法作用域、此法分析等抽象概念描述的时候，理解起来往往会很混乱，所以本文将采用图文并茂、通俗易懂的方式讲解作用域与作用域链。

## 作用域(Scope)

### **什么是作用域**

作用域是一套规则，这套规则用来管理JS引擎如何在当前作用域以及嵌套的子作用域中根据变量名或者函数名进行查找，换个角度说，作用域限制了运行时代码中的某些特定部分中变量，函数和对象的可访问性。

示例：

function outFun() {

var inVariable = "内层变量2";

}

outFun();//要先执行这个函数，否则根本不知道里面是啥

console.log(inVariable); // Uncaught ReferenceError: inVariable is not defined

上述例子，存在两个作用域，一个是全局作用域，一个是outFunoutFun函数包裹的函数作用域， 由于inVariable只在函数作用域中定义，在外层的全局作用域中没定义声明，所以在全局作用域下取值报错了。

通过上述例子，我们可以进一步理解：**作用域就是一个独立的地域，该地域中的变量不会外泄和暴露给外部。也就是说作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。**

ES6 之前 JavaScript 没有块级作用域,只有全局作用域和函数作用域。ES6的到来，为我们提供了‘**块级作用域**’,可通过新增命令let和const来体现。

### 作用域的类型

作用域有三种类型：**全局作用域、函数作用域、块级作用域**

**全局作用域**

下列三种情形拥有全局作用域：

1. **最外层函数和在最外层函数定义的变量拥有全局作用域。**

var outVariable = "我是最外层变量"; //最外层变量

function outFun() { //最外层函数

var inVariable = "内层变量";

function innerFun() { //内层函数

console.log(inVariable);

}

innerFun();

}

console.log(outVariable); //我是最外层变量

outFun(); //内层变量

console.log(inVariable); //inVariable is not defined

innerFun(); //innerFun is not defined

这个例子中的outVariable 变量和outFun函数就拥有全局作用域。

1. **所有末定义直接赋值的变量自动声明为拥有全局作用域**

function outFun2() {

variable = "未定义直接赋值的变量";

var inVariable2 = "内层变量2";

}

outFun2();//要先执行这个函数，否则根本不知道里面是啥

console.log(variable); //未定义直接赋值的变量

console.log(inVariable2); //inVariable2 is not defined

在非严格模式下，这个例子中outFun2函数中的variable变量 ，没声明定义就直接赋值了，则会被自动声明未拥有全局作用域。

1. 所有window对象的属性拥有全局作用域

一般情况下，window对象的内置属性都拥有全局作用域，例如window.name、window.location、window.top等等。

经过以上三种情形，我们可以总结出：**在代码中任何地方都能访问到的对象拥有全局作用域。**

全局作用域是有弊端的，如果我们定义的变量都放在全局作用域中，那么就会污染全局的命名空间，稍有不慎就有可能会引起命名冲突。

// 张三写的代码中,写在a.js文件

var data = {a: 100}

// 李四写的代码中,写在b.js文件

var data = {x: true}

console.log(data)// {x: true}

李四定义的变量与张三的data冲突了。

**函数作用域**

怎么解决全局作用域污染命名空间这个问题呢？答案是：使用函数作用域。这就是为何 jQuery、Zepto 等库的源码都会用个函数包裹起来，所有的代码都会放在(function(){....})()中。因为放在里面的所有变量，都不会被外泄和暴露，不会污染到外部命名空间，不会对其他的库或者 JS 脚本造成影响。这是函数作用域的一个体现。

函数作用域,是指声明在函数内部的变量，和全局作用域相反，局部作用域一般只在函数内部可访问到。

function doSomething(){

var blogName="浪里行舟";

function innerSay(){

alert(blogName);

}

innerSay();

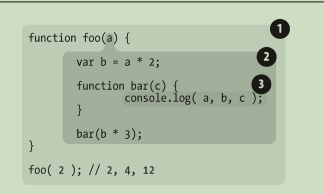
}

alert(blogName); //脚本错误

innerSay(); //脚本错误

blogName与innerSay都属于doSomething函数内部的变量，在全局作用域无法访问。

当函数嵌套时，函数作用域是类似泡泡一样，分层嵌套的。



可看出：

1. 包含着整个全局作用域，其中只有一个标识符：foo。 

2. 包含着 foo 所创建的作用域，其中有三个标识符：a、bar 和 b。 

3. 包含着 bar 所创建的作用域，其中只有一个标识符：c。

**作用域气泡由其对应的作用域块代码写在哪里决定，它们是逐级包含的**。现在只要假设每一个函数都会创建一个新的作用域气泡。 bar 的气泡被完全包含在 foo 所创建的气泡中，唯一的原因是那里就是我们希望定义函数 bar 的位置。

值得注意的是：在ES9之前，**用var声明的变量**，在块语句（大括号“｛｝”中间的语句），如 if 、 switch 条件语句、 for、while 循环语句，它们不会创建一个新的作用域。在块语句中定义的变量将保留在它们已经存在的函数作用域或全局中。

if (true) {

// 'if' 条件语句块不会创建一个新的作用域

var name = 'Hammad'; // name 依然在全局作用域中

}

console.log(name); // 'Hammad'

**块级作用域**

ES6 引入了块级作用域，让变量的生命周期更加可控。块作用域会在块语句（大括号“｛｝”中间的语句），如 if 、 switch 条件语句、 for、while 循环语句中创建，简单的讲，块作用域在一个代码块（由一对花括号包裹）内部创建。

在es6中，应该使用let、const来声明变量，其会将变量作用域限制在当前代码块中，也就是块作用域。块作用域有以下特点：

1. 不存在变量提升。使用let/const声明的变量不会提示到代码块的顶部。

function getValue(condition) {

if (condition) {

let value = "blue";

return value;

} else {

// value 在此处不可用

return null;

}

// value 在此处不可用

}

1. 重复声明会抛错

如果一个变量在代码块内部用let、const已经定义过了，再同名定义就会抛出一个错误。

if (true) {

var count = 30;

let count = 40; // Uncaught SyntaxError: Identifier 'count' has already been declared

}

在同一块作用域中，使用同名声明变量会抛错，但是如果在嵌套的作用域声明同名变量，则不会抛错。

var count = 30;

if (condition) {

let count = 40;// 不会抛出错误

}

1. 循环语句绑定块作用域

没绑定循环语句的情况：

var a = [];

for (var i = 0; i < 10; i++) {

a[i] = function () {

console.log(i);

};

}

a[6](); // 10

上面代码中，变量i是var命令声明的，在全局范围内都有效，所以全局只有一个变量i。每一次循环，变量i的值都会发生改变，而循环内被赋给数组a的函数内部的console.log(i)，里面的i指向的就是全局的i，所有数组a的成员里面的i，指向的都是同一个i，导致运行时输出的是最后一轮的i的值，也就是 10。

绑定块作用域的情况：

如果使用let，声明的变量仅在块级作用域内有效，最后输出的是 6。

var a = [];

for (let i = 0; i < 10; i++) {

a[i] = function () {

console.log(i);

};

}

a[6](); // 6

上面代码中，变量i是let声明的，当前的i只在本轮循环有效，所以每一次循环的i其实都是一个新的变量，所以最后输出的是6。你可能会问，如果每一轮循环的变量i都是重新声明的，那它怎么知道上一轮循环的值，从而计算出本轮循环的值？这是因为 JavaScript 引擎内部会记住上一轮循环的值，初始化本轮的变量i时，就在上一轮循环的基础上进行计算。

另外，for循环还有一个特别之处，就是设置循环变量的那部分是一个父作用域，而循环体内部是一个单独的子作用域。

for (let i = 0; i < 3; i++) {

let i = 'abc';

console.log(i);

}

// abc

// abc

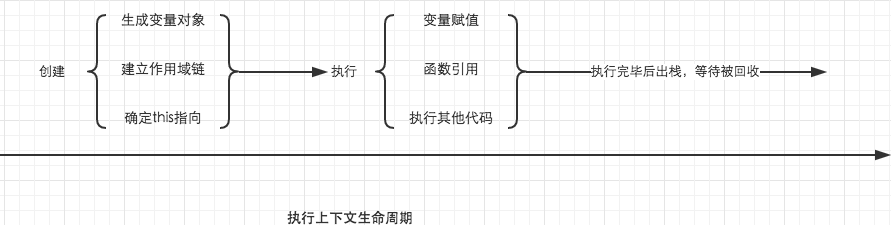
// abc

上面代码正确运行，输出了 3 次abc。这表明函数内部的变量i与循环变量i不在同一个作用域，有各自单独的作用域。

## 作用域链

作用域和作用域链不是同个东西。作用域是一套规则，而作用域链则是代码执行过程中，会变化的一条索引路径。

回顾下我们在《深入理解Javascript 执行上下文和执行栈》中的执行上下文的生命周期：



在函数调用激活时，会开始创建对应的执行上下文，在执行上下文生成的过程中，变量对象、作用域链、this指向确定。变量对象已经在上述文章详细分析了，在这里我们将细说作用域链。

**作用域链，是由当前环境与上层环境的一系列变量对象组成，它保证了当前执行环境对符合访问权限的变量和函数的有序访问。**

示例：

var a = 20;

function test() {

var b = a + 10;

function innerTest() {

var c = 10;

return b + c;

}

return innerTest();

}

test();

在上面的例子中，全局，函数test，函数innerTest的执行上下文先后创建。我们设定他们的变量对象分别为VO(global)，VO(test), VO(innerTest)。而innerTest的作用域链，则同时包含了这三个变量对象，所以innerTest的执行上下文可如下表示。

innerTestEC = {

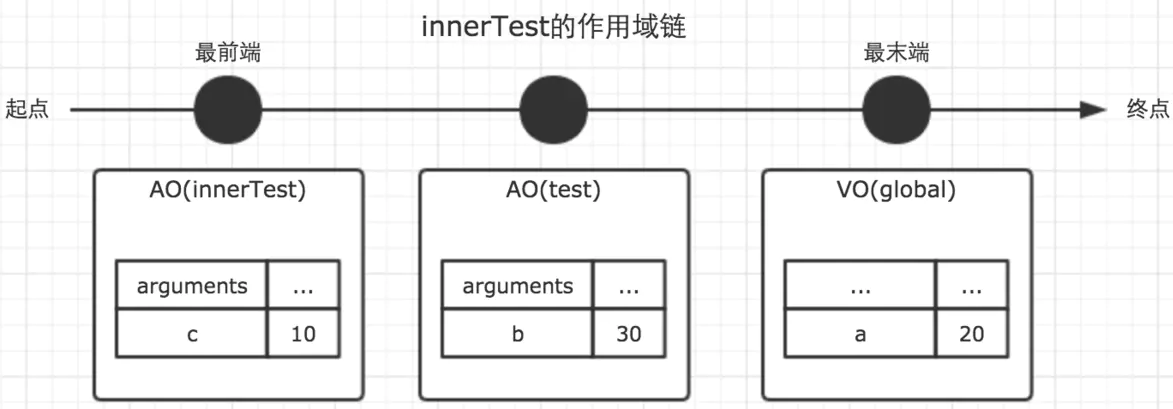
VO: {...}, // 变量对象

scopeChain: [VO(innerTest), VO(test), VO(global)], // 作用域链

}

我们这里使用一个数组来模拟作用域链，数组的第一项scopeChain[0]为作用域链的最前端，而数组的最后一项，为作用域链的最末端，所有的最末端都为全局变量对象。

很多人会误解为当前作用域与上层作用域为包含关系，但其实并不是。以最前端为起点，最末端为终点的单方向通道我认为是更加贴切的形容。如图。



作用域链是由一系列的变量对象组成的，在该单向链路上，查询变量对象的标示符（变量名），这样就能访问到上一层作用域中的变量。通俗的讲，就是当访问一个变量时，解释器会首先在当前作用域查找标示符，如果没有找到，就去父作用域找，直到找到该变量的标示符或者不在父作用域中，这就是作用域链。

作用域链和原型继承查找时的区别：如果去查找一个普通对象的属性，但是在当前对象和其原型中都找不到时，会返回undefined；但查找的属性在作用域链中不存在的话就会抛出ReferenceError。

接下来，我们从函数嵌套和无函数嵌套的角度来分析作用域链。

**无嵌套的函数**

// my\_script.js

"use strict";

var foo = 1;

var bar = 2;

function myFunc() {

var a = 1;

var b = 2;

var foo = 3;

console.log("inside myFunc");

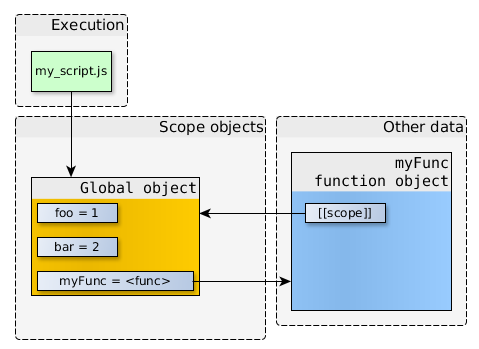
}

console.log("outside");

myFunc();

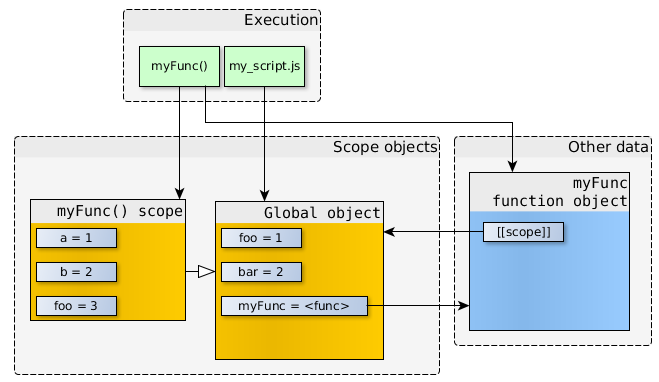
**定义时：**当myFunc被定义的时候，myFunc的标识符（identifier）就被加到了全局对象中，这个标识符所引用的是一个函数对象（myFunc function object）。

内部属性[[scope]]指向当前的作用域对象，也就是函数的标识符被创建的时候，我们所能够直接访问的那个作用域对象（即全局对象）。



myFunc所引用的函数对象，其本身不仅仅含有函数的代码，并且还含有指向其被创建的时候的作用域对象。

调用时：当myFunc函数被调用的时候，一个新的作用域对象被创建了。新的作用域对象中包含myFunc函数所定义的本地变量，以及其参数（arguments）。这个新的作用域对象的父作用域对象就是在运行myFunc时能直接访问的那个作用域对象（即全局对象）。



**有嵌套的函数**

当函数返回没有被引用的时候，就会被垃圾回收器回收。但是对于闭包，即使外部函数返回了，函数对象仍会引用它被创建时的作用域对象。

"use strict";

function createCounter(initial) {

var counter = initial;

function increment(value) {

counter += value;

}

function get() {

return counter;

}

return {

increment: increment,

get: get

};

}

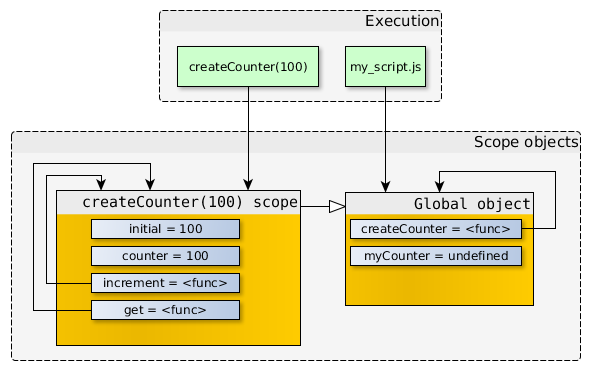
var myCounter = createCounter(100);

console.log(myCounter.get()); // 返回 100

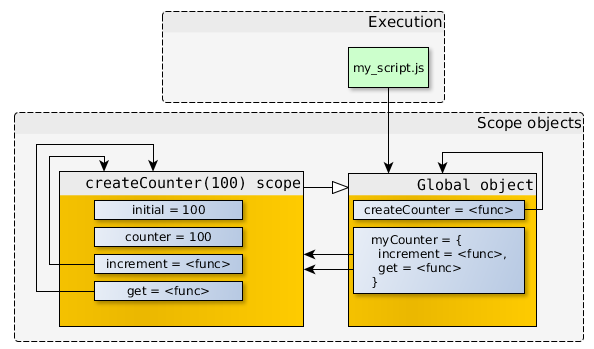
myCounter.increment(5);

console.log(myCounter.get()); // 返回 105

当调用 createCounter(100) 时，内嵌函数increment和get都有指向createCounter(100) scope的引用。假设createCounter(100)没有任何返回值，那么createCounter(100) scope不再被引用，于是就可以被垃圾回收。



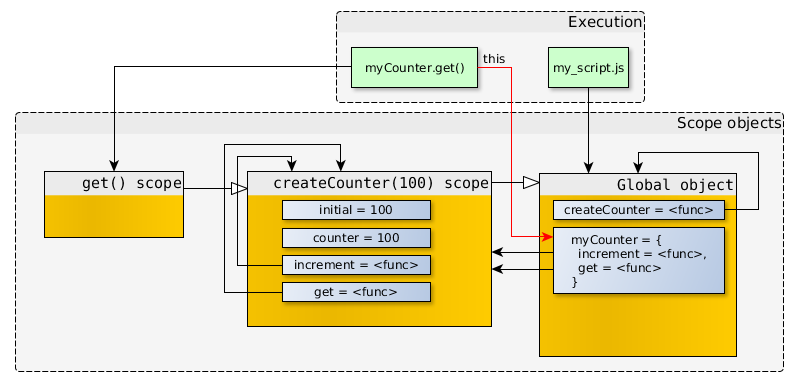
但是createCounter(100)实际上是有返回值的，并且返回值被存储在了myCounter中，所以对象之间的引用关系如下图：



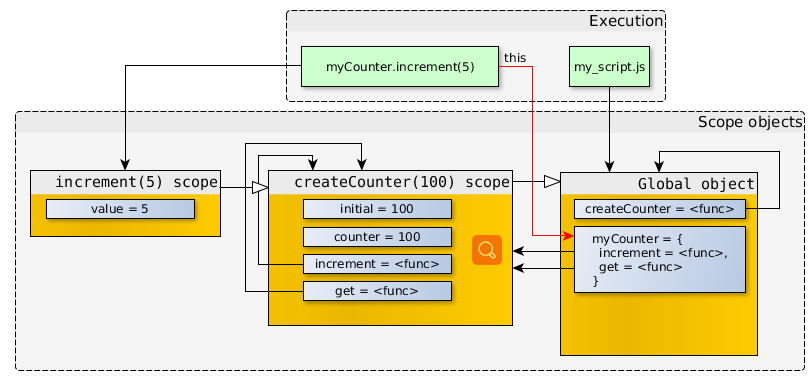
即使createCounter(100)已经返回，但是其作用域仍在，并且只能被内联函数访问。可以通过调用myCounter.increment() 或 myCounter.get()来直接访问createCounter(100)的作用域。

当myCounter.increment() 或 myCounter.get()被调用时，新的作用域对象会被创建，并且该作用域对象的父作用域对象会是当前可以直接访问的作用域对象。

调用get()时，当执行到return counter时，在get()所在的作用域并没有找到对应的标示符，就会沿着作用域链往上找，直到找到变量counter，然后返回该变量。



单独调用increment(5)时，参数value保存在当前的作用域对象。当函数要访问counter时，没有找到，于是沿着作用域链向上查找，在createCounter(100)的作用域找到了对应的标示符，increment()就会修改counter的值。除此之外，没有其他方式来修改这个变量。闭包的强大也在于此，能够存贮私有数据。



**再说点扩展的事。看代码：**

对于上面的counter示例，再说点扩展的事。看代码：

//myScript.js

"use strict";

function createCounter(initial) {

/\* ... see the code from previous example ... \*/

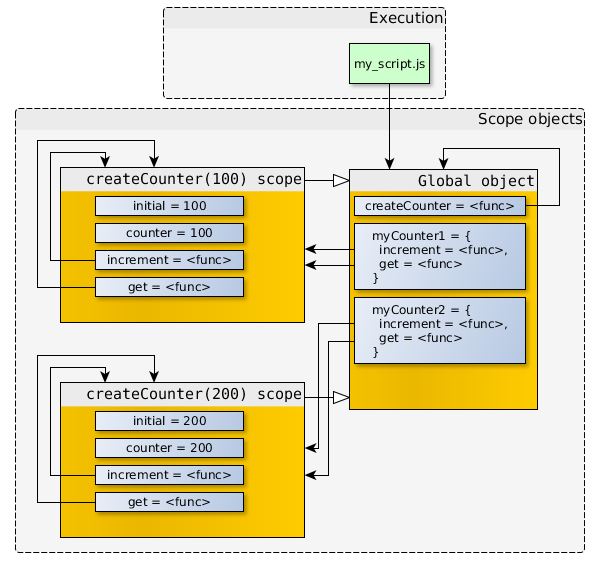
}

//-- create counter objects

var myCounter1 = createCounter(100);

var myCounter2 = createCounter(200);

myCounter1 和 myCounter2创建之后，关系图是酱紫的：



在上面的例子中，myCounter1.increment和myCounter2.increment的函数对象拥有着一样的代码以及一样的属性值（name，length等等），但是它们的[[scope]]指向的是不一样的作用域对象。

这才有了下面的结果：

var a, b;

a = myCounter1.get(); // a 等于 100

b = myCounter2.get(); // b 等于 200

myCounter1.increment(1);

myCounter1.increment(2);

myCounter2.increment(5);

a = myCounter1.get(); // a 等于 103

b = myCounter2.get(); // b 等于 205

## 作用域与执行上下文

许多开发人员经常混淆作用域和执行上下文的概念，误认为它们是相同的概念，但事实并非如此。

我们知道JavaScript属于解释型语言，JavaScript的执行分为：解释和执行两个阶段,这两个阶段所做的事并不一样：

**解释阶段：**

词法分析

语法分析

作用域规则确定

**执行阶段**：

创建执行上下文

执行函数代码

垃圾回收

JavaScript解释阶段便会确定作用域规则，因此作用域在函数定义时就已经确定了，而不是在函数调用时确定，但是执行上下文是函数执行之前创建的。执行上下文最明显的就是this的指向是执行时确定的。而作用域访问的变量是编写代码的结构确定的。

作用域和执行上下文之间最大的区别是：

执行上下文在运行时确定，随时可能改变；作用域在定义时就确定，并且不会改变。

一个作用域下可能包含若干个上下文环境。有可能从来没有过上下文环境（函数从来就没有被调用过）；有可能有过，现在函数被调用完毕后，上下文环境被销毁了；有可能同时存在一个或多个（闭包）。同一个作用域下，不同的调用会产生不同的执行上下文环境，继而产生不同的变量的值。