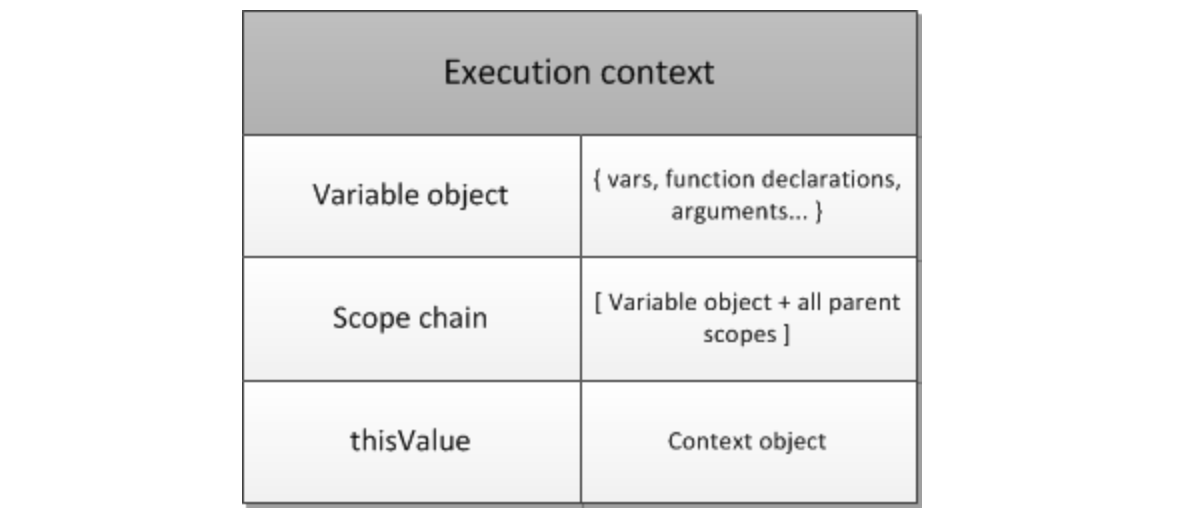
## JS执行上下文

执行上下文可以抽象的表示为一个简单对象。每一个执行上下文都具有跟踪其关联代码的执行进度的一组属性（我们可以称为上下文状态）。

当一段JS代码在运行的时候，它实际上是运行在**执行上下文**中。下面三种类型的JS代码会创建一个新的执行上下文：

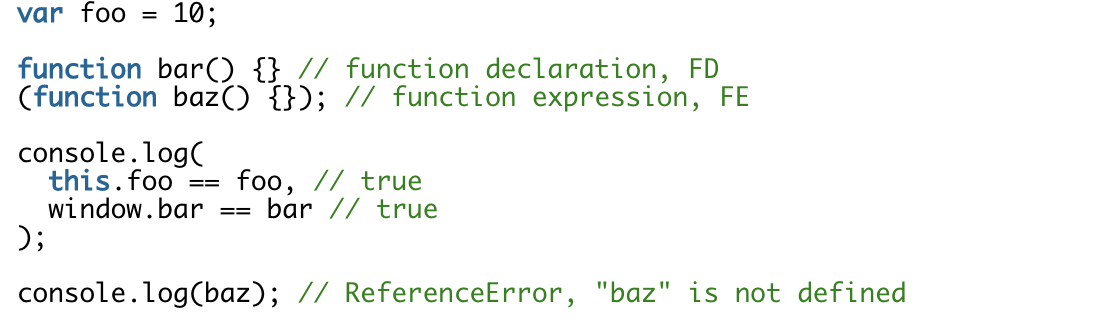
1. 全局上下文是为运行代码主体而创建的执行上下文，也就是说它是为了那些存在于JS函数之外的任何代码创建的。一个程序中只会存在一个全局上下文（它会做两件事：1创建一个全局对象，浏览器中也就是window对象2绑定this为这个全局对象）
2. 每个函数会在**执行的时候**创建自己的执行上下文（通常说的“本地上下文”）
3. 使用eval()也会创建一个新的执行上下文



上下文结构（摘自某Facebook程序员解读）

除了上述三个必须的属性（变量对象，作用域链，this值）之外，执行上下文还可以具有其他状态，取决于具体实现。

1. 变量对象：是和执行上下文关联的数据容器，它是一个特殊的对象，用于存储在上下文中定义的变量和函数声明。注意：函数表达式不包含在变量对象中（与函数声明相反）。变量对象是一个抽象的概念，在不同的上下文类型中，实际上是使用不同的对象呈现的，例如全局上下文中，变量对象就是全局对象本身（所以我们可以使用全局对象的属性名称引用全局变量）



活动对象（Activation Object）

在函数上下文中，用活动对象表示变量对象。函数的变量对象也是简单的变量对象，但是除了变量和函数声明外，还储存着形式参数和参数（arguments）对象，我们称之为活动对象。

注意⚠️：ES5中已经将变量对象和活动对象的概念整合到词法环境模型中。

1. 作用域链：作用域链是一个能被代码作用域中查询到标识符的对象列表。这里的规则也很简单并且类似于原型链：如果在自己的作用域（自己的变量对象、活动对象）中找不到某个变量。他会去查找父级的变量对象，以此类推。

关于上下文，标识符是：变量名、函数声明、形式参数等。当函数在其代码中的引用不是局部变量（局部函数或形式参数）时，这类变量称之为自由变量（free variable）。为了精确地查找这些自由变量，我们使用到了作用域链。

通常，作用域链是所有父级变量对象的列表，再加上（在作用域链的前面）函数自身的变量/活动对象。但是作用域链还可能包含其他任何对象，例如在上下文执行期间动态被添加到作用域中的对象（with子对象、catch子句的特殊对象）。

当解析（查找）标识符时，会从活动对象开始搜索作用域链，然后（如果自身活动对象中没有找到标识符）会一直搜索到作用域链的顶层，就像原型链一样。

var x = 10;

(function foo() {

var y = 20;

(function bar() {

var z = 30;

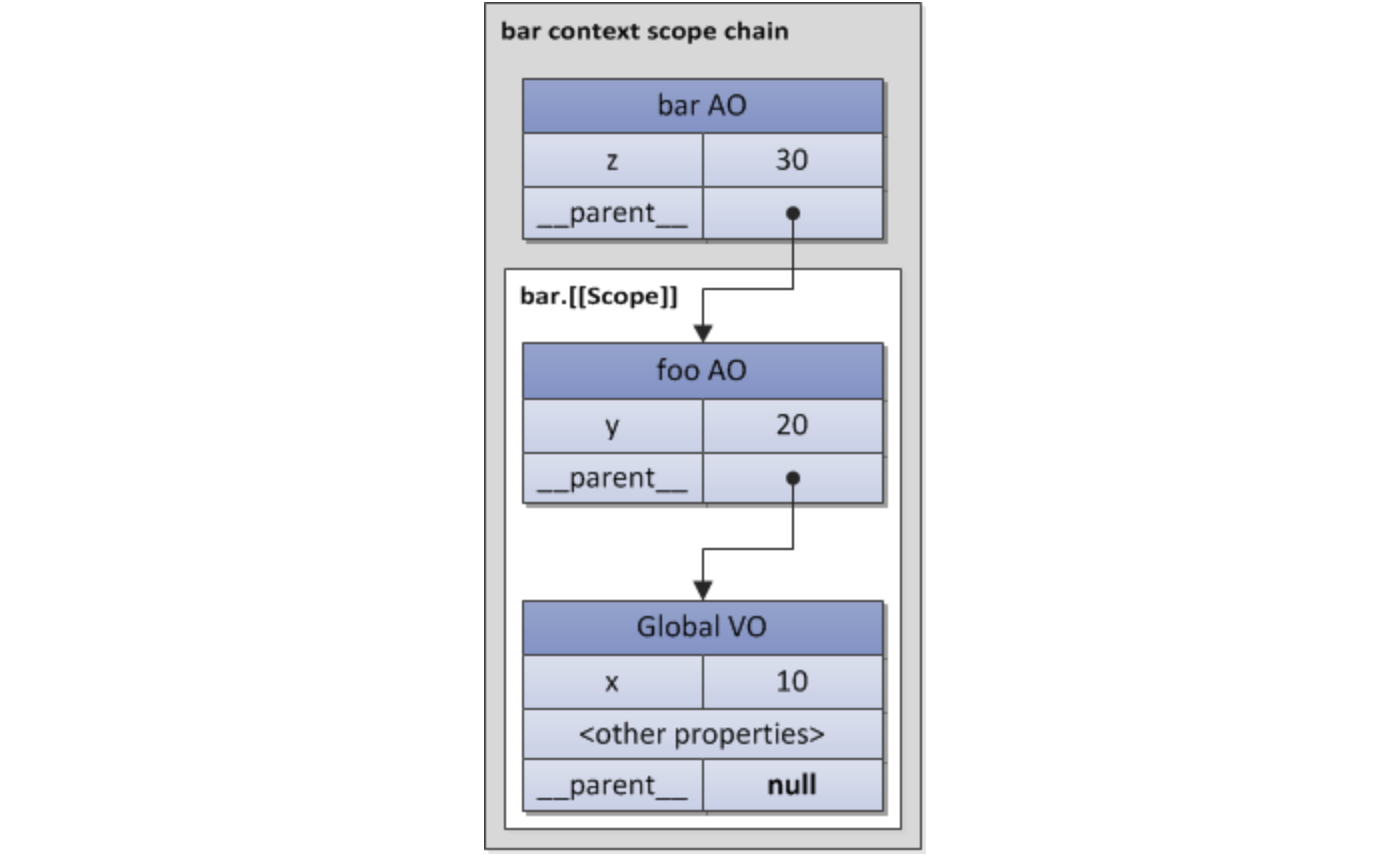
*// x和y是自由变量，可以在bar的作用域链的下一个（bar的活动对象后的）对象中查询到*

console.log(x, y, z)

})()

})()

我们可以假设通过隐式的\_\_parent\_\_属性来链接作用域对象，该属性指向链中的下一个对象。该技术在ES5中被应用（被命名为outer）。作用域链的另一种表现形式也可以是一个简单数组。这里我们演示一下使用\_\_parent\_\_概念的上述代码：



在代码执行期间，可以用with或者catch子句对象来扩展作用域链。由于这些对象也都是简单对象，所以他们也可能具有原型（和原型链）。这也导致了作用域查找其实是二维的1首先考虑作用域链；2其次在每个作用域链接上，深入到链接的原型链（如果链接有原型）

Object.prototype.x = 10;

var w = 20;

var y = 30;

*//在SpiderMonkey（Mozilla项目的一部分，用C语言实现的JS脚本引擎）中*

*//全局上下文的变量对象继承自“Object.prototype”*

*// 所以这里我们可以打印出没有定义的全局变量x*

*// x是从原型链中查找到的。*

console.log(x);

(function foo() {

*// foo的局部变量*

var w = 40;

var x = 100;

*// x在Object.prototype中被查找到*

*// 因为{z: 50}继承了它*

with ({z: 50}) {

console.log(w, x, y, z); *// 40 10 30 50*

}

*// with对象从作用域链中移除后*

*// x在foo函数上下文的活动对象中查找到*

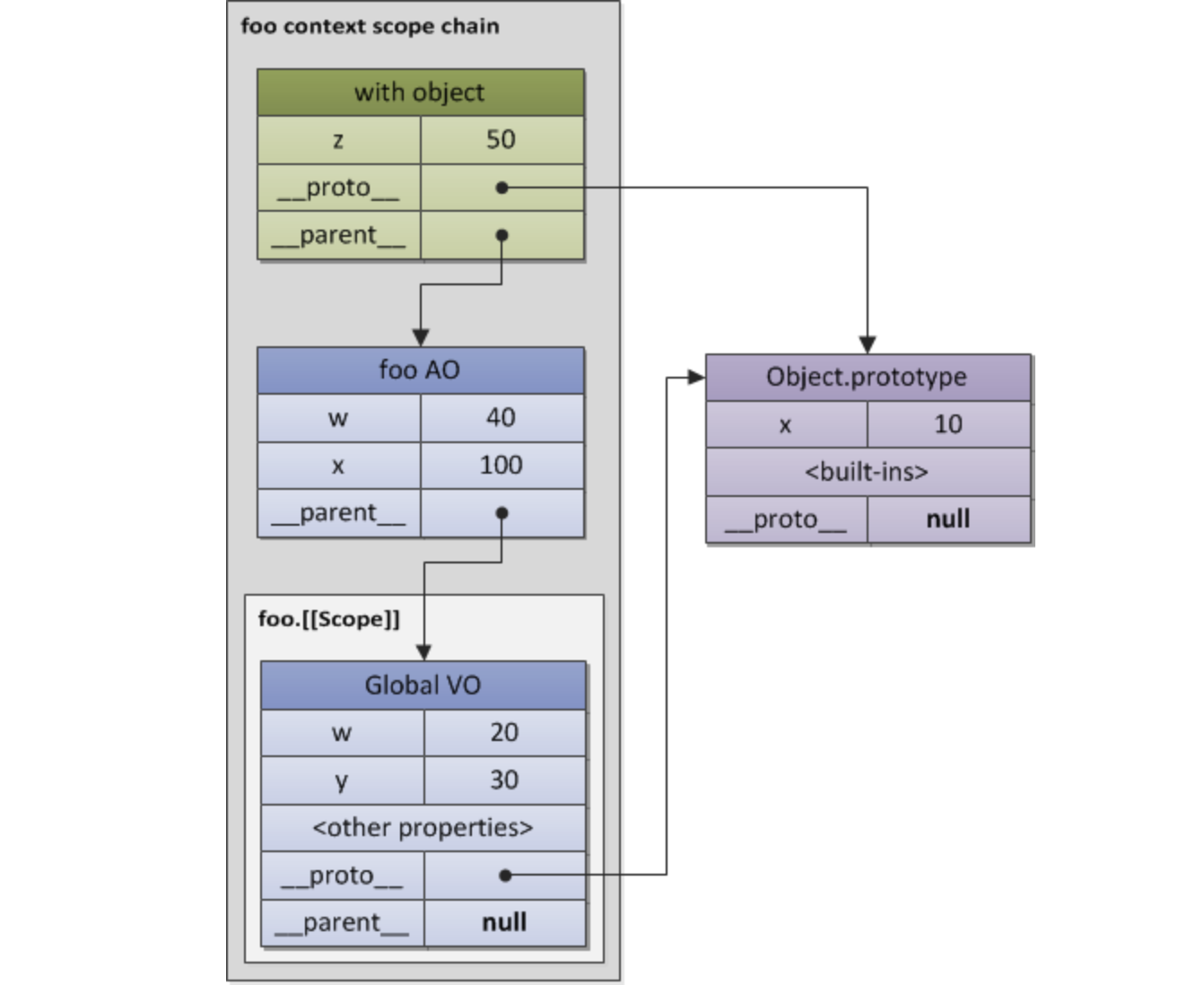
*// w同样也是局部的*

console.log(x, w); *// 100 40*

console.log(window.w); *// 20*

})()

结构如下：在查找\_\_parent\_\_链接前，会先考虑\_\_proto\_\_链



注意⚠️：并不是所有实现中，全局对象都从Object.prototype继承。

在所有父变量对象都存在的时候，从内部函数获取父级数据并没有什么特别的，我们只是遍历作用域链来解析（查找）我们所需要的变量。但是我们知道当上下文结束后，其所有状态和其本身都将被销毁。同时，内部函数从父函数返回。此外，这个被返回函数

可能稍后在另一个上下文中被激活。如果一个自由变量的上下文已经消失，那个这种激活将是什么？在一般的理论中，有助于该理论的概念称之为闭包（closure）。在ES中，闭包和作用域链概念直接相关。下一节（JS闭包）中我们会详细分析闭包。

1. this值：是一个与执行上下文相关的特殊对象。因此我们可以将其称为上下文对象（即在一个被激活的执行上下文中的对象）。任何对象都可以用作上下文的this值，一个重要的提醒是：this值是执行上下文的属性，而不是变量对象的属性。

这个特性非常重要，因为和变量相比，该值永远不会参与标识符的解析（查找过程）。即在代码中访问它时，是直接在执行上下文中获取而无需任何作用域链查找。该值在进入上下文时就被确定了。

注意⚠️：在ES6中，this值变成了词法环境中的一个属性，即ES3中的变量对象的一个属性。此举目的在于支持尖头函数功能（() => void），该功能具有词法this，它们从父级上下文中继承。

在全局上下文中，this值就是全局对象其自身（意味着this等于变量对象。）

var x = 10;

console.log(x, this.x, window.x); *// 10 10 10*

在函数上下文中，this值在每一次单独的调用中可能会不同。在这里，this值由调用者通过调用表达式的形式提供（即函数被激活的方式）。例如下例中，foo是一个被调用方，它在全局上下文（一个调用方）中被调用。

function foo() {

console.log(this);

}

foo(); *// 全局对象*

foo.prototype.constructor(); *// foo.prototype*

var bar = {

baz: foo,

};

bar.baz(); *// bar*

(bar.baz)(); *// 还是bar*

(bar.baz = bar.baz)(); *// 但这是全局对象*

(bar.baz, bar.baz)(); *// 全局对象*

(false || bar.baz)(); *// 全局对象*

var otherFoo = bar.baz;

otherFoo(); *// 还是全局对象*

现在我们比较在全局上下文和函数上下文中this值的区别：

* 全局上下文中的this值：

此时，情况变得很简单。全局代码中，this值总是全局对象本身。因此可以间接的引用它。

*// 显示地定义全局对象属性*

this.a = 10; *// global.a = 10*

console.log(a); *// 10*

*// 隐式地定义，通过分配给非完全限定(unqualified)的标识符*

b = 20;

console.log(this.b); *// 20*

*// 同样隐式地通过变量声明，因为变量对象在全局上下文中就是全局对象本身*

var c = 30;

console.log(this.c); *// 30*

* 函数上下文中的this值：

在这类代码中this值第一个（并且可能是最重要的）特性，就是没有静态的绑定到函数。

上面我们提到过，this值在进入上下文时才被确定。并且每一次使用函数时this值都可能不同。然而，在代码运行时this值是不可变的，即不可能为其分配一个新值，因为它不是变量。

var foo = {x: 10};

var bar = {

x: 20,

test: function() {

console.log(this === bar); *// true*

console.log(this.x); *// 20*

this = foo; *// 错误，不可修改this*

console.log(this.x); *// 20 如果上一行不会报错的话: 应该是10*

}

}

*// 进入上下文时this值被确定为bar，*

bar.test(); *// true 20 20*

foo.test = bar.test;

*// 然而这里this值指向了foo，尽管我们调用同一个函数*

foo.test(); *// false 10 10*

那么究竟是什么影响了函数上下文中的this？有以下几个因素：

首先，在一般的函数调用中，this由调用者（激活代码上下文）提供。即调用该函数的父上下文。并且this的值由调用表达式的形式确定（换句话说，通过语法上函数调用的形式确定）。调用表达式的确切形式，即调用函数的方式会影响被调用上下文的this值而不会影响任何其他值。

我们看，即使普通的全局函数也可以通过不同形式的调用表达式激活，这会影响this的值。

function foo() {

console.log(this);

}

foo(); *// 全局对象*

console.log(foo === foo.prototype.constructor); *// true*

*// 但是另一种形式的调用表达式（同一个函数），this值却不同*

foo.prototype.constructor(); *// foo.prototype*

类似地我们可以调用一个被定义为某个对象方法的函数，但是this值并不会被设置为那个对象。

var foo = {

bar: function() {

console.log(this);

console.log(this === foo);

}

}

foo.bar(); *// foo true*

var funcTest = foo.bar;

console.log(funcTest === foo.bar); *// true*

*// 再一次，我们使用不同形式的调用表达式（同一个函数），this值不相同*

funcTest(); *// 全局对象 false*

那么调用函数表达式的形式又是如何影响this值的呢？这个和引用类型相关（Reference）引用类型我们可以抽象为：

*// 伪代码将应用类型表示为一个具有两个属性的对象*

*// base: 属性的所属对象*

*// propertyName: 属性名*

*// ES5中，引用类型还会有一个strict属性，标识一个引用是否在严格模式下处理*

var valueOfReferenceType = {

base: <base object>,

propertyName: <property name>

}

引用类型的值只会存在于两种情况下：

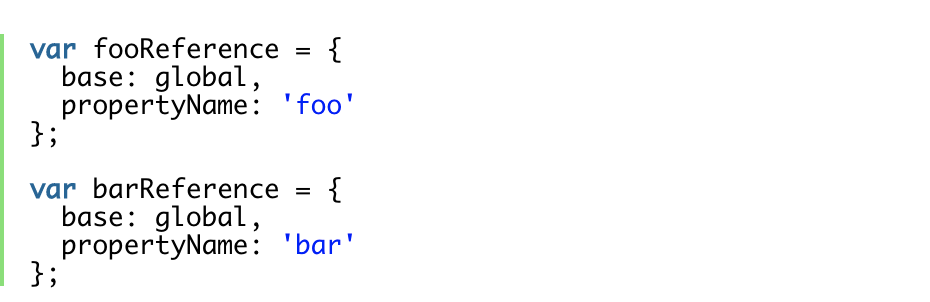
1. 当我们处理一个标识符
2. 或使用属性访问器

标识符有：变量名称、函数名称、函数参数名、全局对象的非完全限定属性名，标识符被算法处理完总是返回一个引用类型的值（这对this值很重要）

var foo = 10;

function bar() {}

相应的引用类型值大概这样

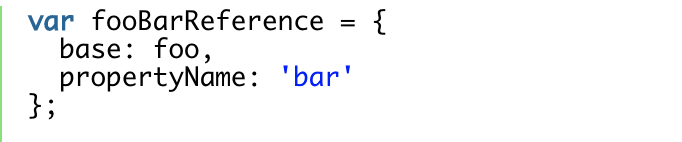


属性访问器有两种变体：

foo.bar();

foo['bar']();

相应的引用类型值大概这样：



我们来看在函数执行上下文中，引用类型的值是如何与this值关联起来的？在一个函数上下文中决定this值的一般规则如下：

如果在调用括号(…)的左侧有一个引用类型的值，那么this值将设置为这个引用类型值的base对象。

其他情况下（具有不同于引用类型的任何其他值类型），那么this值总是会被设置为null。但是由于this值置为null并没有任何意义，所以会被隐式地转换为全局对象。注意⚠️：ES5的严格模式下，不会强制把null转为全局对象，而是置为undefined

下面我门在看一些例子：

function foo() {

return this;

}

foo(); *//全局对象*

这时(…)的左侧是一个引用类型的值（因为foo是一个标识符）。然后this值被设置为base对象，即全局对象。

属性访问器也是类似地：

var foo = {

bar: function () {

return this;

}

}

foo.bar(); *// foo*

此时我们也在(…)左侧有一个引用类型的值（base对象为foo），这个base对象在函数激活时被用作了this值。

然而激活同一个函数用不同形式的调用表达式，我们有其他的this值：

var foo = {

bar: function () {

return this;

}

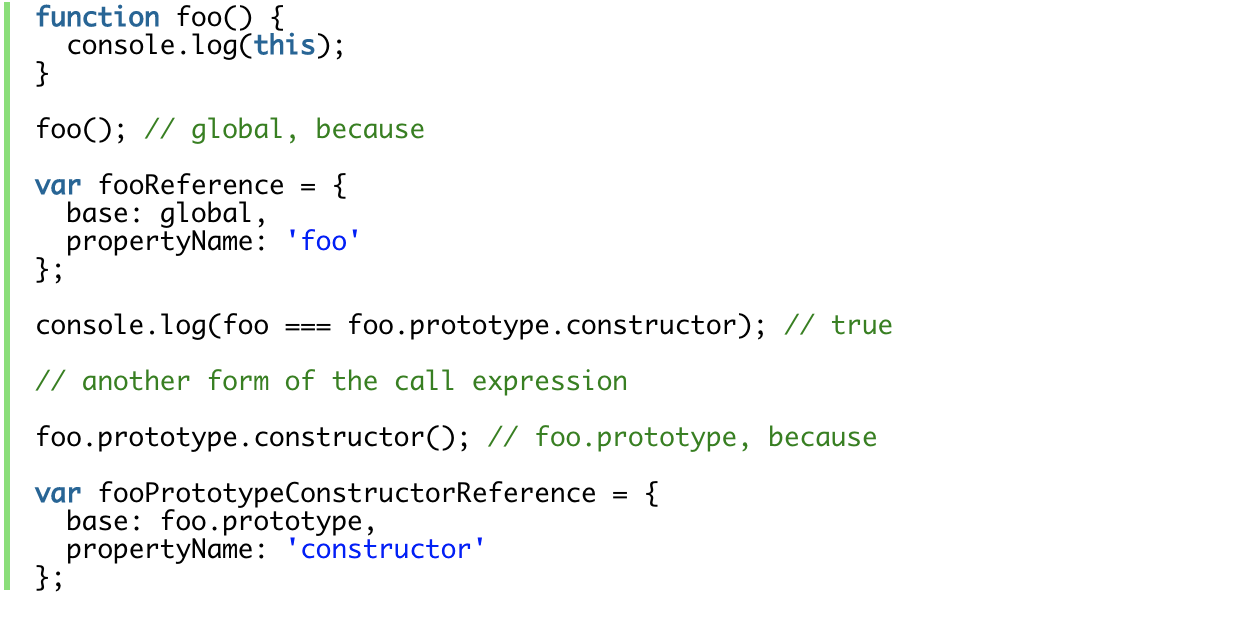
}

test = foo.bar;

test(); *// 全局对象*

此时test是一个标识符，会生成一个引用类型值（base对象为全局对象）

现在我们可以准确地给说出，为什么同一个方法在不同的函数调用表达式形式下会有不一样的this值。答案就是引用（Reference）类型的中间值不一样



另一个（典型的）通过调用表达式形式动态确定this值的例子：

function foo(){

console.log(this.bar);

}

var x = {bar: 10};

var y = {bar: 20};

x.test = foo;

y.test = foo;

x.test(); *// 10*

y.test(); *// 20*

(…)左侧为非引用类型值的情况，this值被设置为null，然后隐式地被置为全局对象。

我们看一些例子：

(function () {

console.log(this); *// null => 全局对象*

})()

此时没有引用类型值（既不是标识符，也不是属性访问器）this值最终被置为全局对象。

(function () {

'use strict';

console.log(this); *// undefined*

})()

严格模式下，不会设置为全局对象，而是undefined

一些更复杂的例子：

var foo = {

bar: function() {

console.log(this);

}

}

foo.bar(); *// Reference 输出foo*

(foo.bar)(); *// Reference 输出foo*

(foo.bar = foo.bar)(); *// null => 全局对象*

(false || foo.bar)(); *// null => 全局对象*

(foo.bar, foo.bar)(); *// null => 全局对象*

引用类型但this值为null：

function foo() {

function bar() {

console.log(this);

}

bar(); *// 相当于AO.bar() ES3中如果base对象是一个活动对象，那么返回null*

}

foo(); *// null => 全局对象*

此时局部变量、内部函数、形式参数存储在活动对象（AO）中。AO.bar()又等价于null.bar()。所以this值又被隐式地只为全局对象。注意⚠️：ES3规范中，当一个base对象是一个AO时，返回null。

var x = 10;

with({

foo: function() {

console.log(this.x);

},

x: 20,

}) {

foo(); *// 20 因为with语句将fooReference的base对象置为了\_\_withObject*

}

try {

throw function() {

console.log(this);

}

} catch(e) {

e(); *// 在ES3中：\_\_catchObject；在ES5中：全局对象*

}

当一个函数被当作构造函数调用时的this值：

function A() {

console.log(this); *// 新创建的对象，下边的对象 'a'*

this.x = 10;

}

var a = new A();

console.log(a.x);

手动设置一个函数调用的this值：

Function.prototype中定义了两个方法（因此，所有方法都可以访问它们）：apply和call方法。它们之间区别甚微，第一个参数都是作为调用上下文中的this值（并且只有该参数是必须的）。区别在于apply方法第二个参数接收一个数组（或者类数组的对象，比如arguments）；call方法第二个参数接收任何参数。

var b = 10;

function a(c) {

console.log(this.b);

console.log(c);

}

a(20); *// this === 全局对象 this.b === 10 c === 20;*

a.call({b: 20}, 30); *// this === {b: 20} this.b === 20 c === 30*

a.apply({b: 30}, [40]); *// this === {b: 30} this.b === 30 c === 40*

Function.prototype还定义了一个bind方法： 该方法返回一个新函数。

新函数的this值被指定为第一个参数，其余参数作为新函数的参数。如果bind方法参数为空或者thisArg为null、undefined，那么执行作用域的this将被视为新函数的thisArg。注意⚠️：bind方法返回的新函数的this值不再能被改变。

每一个上下文在本质上都是一种作用域层级。每段代码开始执行的时候都会创建一个新的上下文来运行它，并在代码推出的时候销毁掉。

我们来看一段代码，做一个简要的分析：

let outputElem = document.getElementById('output');

let userLanguages = {

Mike: 'en',

Teresa: 'es',

}

function greetUser(user) {

function localGreeting(user) {

let greeting;

let language = userLanguages[user];

switch(language) {

case 'es':

greeting = `¡Hola, ${user}`;

break;

case 'en':

default:

greeting = `Hello, ${user}`;

break;

}

return greeting;

}

outputElem.innerHTML += localGreeting(user) + '<br/>\r';

}

greetUser('Mike');

greetUser('Teresa');

greetUser('Veronica');

这段代码包含了三个执行上下文，其中有些会在程序运行过程中多次创建和销毁。每个上下文创建的时候会被推入执行上下文栈。当推出的时候，它会从执行上下文栈中移除。

* 程序开始运行时，全局上下文就会被创建好。---[GloCtxt]
* 执行到greetuser(‘Mike’)时，会为greetUser()函数创建一个它的执行上下文，这个上下文会被推入执行上下文栈中。---[ GloCtxt, GreUserCtxt]
* 当greetUser()调用localGreeting()的时候会为localGreeting()创建一个它的执行上下文并将其推入执行上下文栈中。---[ GloCtxt, GreUserCtxt, LocGreCtxt]。在localGreeting()退出的时候它的上下文会被执行上下文栈弹出并销毁。---[ GloCtxt, GreUserCtxt]。程序从执行上下文栈中获取下一个上下文并恢复执行，即greetingUser()的剩余部分开始执行。
* greetingUser()执行完毕其上下文也弹出销毁。---[GloCtxt]
* 当greetuser(‘Teresa’)执行时，程序有为其创建一个上下文并推入栈顶---[ GloCtxt, GreUserCtxt]
* …
* …
* …
* …
* …
* 主程序退出全局执行上下文从执行栈中弹出。此时栈中所有上下文都已弹出，程序执行完毕。

以这种方式来使用执行上下文，使得每一个程序和函数都能够拥有自己的变量和其他对象。每个上下文还能额外的跟踪程序中下一行需要执行的代码以及一些对上下文很重要的信息。以这种方式使用上下文和上下文栈使得我们可以对程序运行的一些基础部分进行管理，包括局部变量和全局变量，函数调用与返回等。

关于递归函数（多次调用自身的函数），每次递归调用自身都会创建一个新的执行上下文，这使得JS运行时能够追踪递归的层级和递归中得到的返回值，这也意味着每次递归都会消耗内存来创建新的上下文。

执行上下文分两个阶段：

1. 创建阶段

三件事情：

1确定this，官方称为This Biding；

2确定词法环境（ES5新概念），ES3中就是作用域链。包含**环境记录（ES3中的活动对象）**和**对外部环境的引入记录**两部分组成；

3确定变量环境，也可以称之为词法环境。ES6中它和词法环境的差别在于词法环境用于存储函数声明、let、const声明的变量；而变量环境只用于存储var声明的变量。

创建阶段var声明的变量已经被赋值为undefined而let const声明的变量为<uninitialized>。函数声明被赋值为自身函数。这也是就是函数声明和变量声明提升的本质所在。注意⚠️：变量声明提升只是声明提升，初始化并不会提升。



1. 执行阶段

执行阶段根据之前环境记录对应赋值，var声明的变量有值就赋值（创建阶段为undefined），let const变量有值赋值（创建阶段为<uninitialized>），没有值则赋值为undefined。