

# 1. this指针/闭包/作用域

## 1. 课程目标



- 初级：
  - 掌握JavaScript基础使用；
- 中级：
  - 掌握JavaScript语法的常见面试题及使用技巧；
- 高级：
  - 掌握JavaScript的核心原理实现；

## 2. 课程大纲

- 欢迎加入训练营；
- 原型&原型链；
- 词法作用域&动态作用域；
- 执行上下文；
- 变量对象；
- 作用域链；
- this；
- 执行上下文；
- 闭包；

## 3. 欢迎加入训练营

上课时间：一般一周2节课，周末晚上8:00~10:00；

一般会课前10~15分钟进入直播间，针对往期内容进行答疑；




EncodeStudio  
印客学院

印客学院2023匠心之作  
内容&服务全面升级

# Web前端 大厂工程师训练营

进阶前端架构  
直击阿里P7

 印客2023大厂前端工程师课程大纲0710.pdf

## 4. 原型&原型链

## 4.1 构造函数创建对象

我们先使用构造函数创建一个对象：

```
1  function Person() {  
2  
3  }  
4  var person = new Person();  
5  person.name = 'chenghuai';  
6  console.log(person.name) // chenghuai
```

在这个例子中，Person 就是一个构造函数，我们使用 new 创建了一个实例对象 person。

## 4.2 prototype

每个函数都有一个 `prototype` 属性，比如：

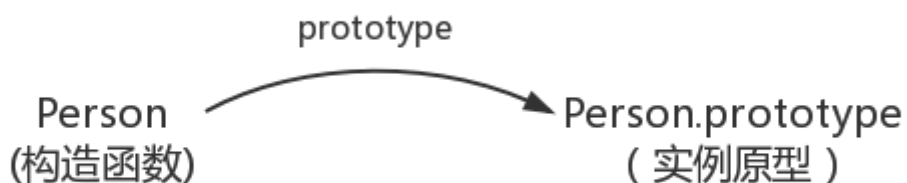
```
1  function Person() {  
2  
3  }  
4  // 虽然写在注释里，但是你要注意：  
5  // prototype是函数才会有的属性  
6  Person.prototype.name = 'chenghuai';  
7  
8  var person1 = new Person();  
9  var person2 = new Person();  
10  
11 console.log(person1.name) // chenghuai  
12 console.log(person2.name) // chenghuai
```

那这个函数的 `prototype` 属性到底指向的是什么呢？是这个函数的原型吗？

其实，函数的 `prototype` 属性指向了一个对象，这个对象正是调用该构造函数而创建的实例的原型，也就是这个例子中的 `person1` 和 `person2` 的原型。

那什么是原型呢？你可以这样理解：每一个JavaScript对象(null除外)在创建的时候就会与之关联另一个对象，这个对象就是我们所说的原型，每一个对象都会从原型"继承"属性。

用一张图表示构造函数和实例原型之间的关系：



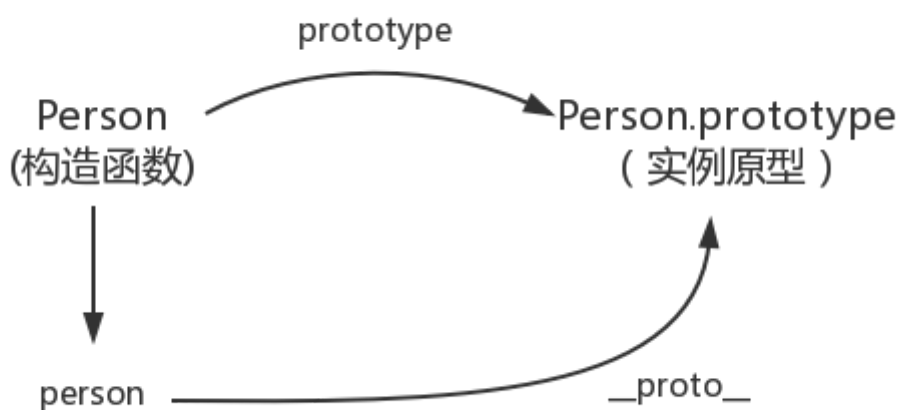
这里用 `Object.prototype` 表示实例原型。

那么该怎么表示实例与实例原型，也就是 `person` 和 `Person.prototype` 之间的关系呢？

### 4.3 proto

这是每一个JavaScript对象(除了 `null`)都具有的一个属性，叫 `__proto__`，这个属性会指向该对象的原型。

```
1 function Person() {  
2  
3 }  
4 var person = new Person();  
5  
6 console.log(person.__proto__ === Person.prototype); // true
```



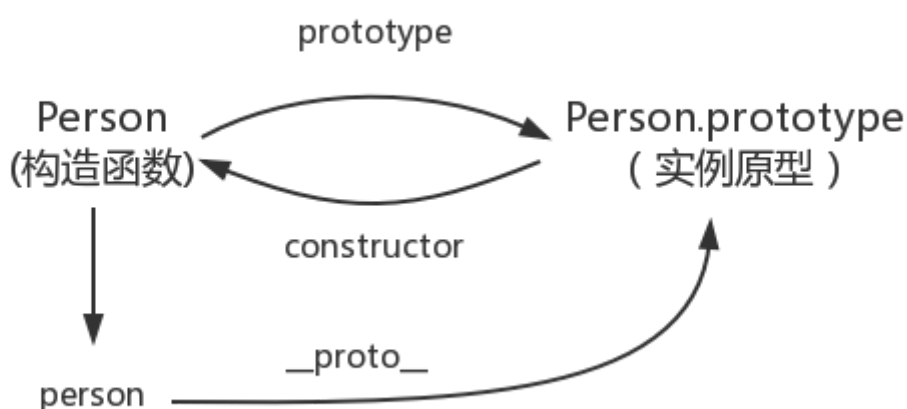
既然实例对象和构造函数都可以指向原型，那么原型是否有属性指向构造函数或者实例呢？

### 4.4 constructor

指向实例倒是没有，因为一个构造函数可以生成多个实例，但是原型指向构造函数是有的：

`constructor`，每个原型都有一个 `constructor` 属性指向关联的构造函数

```
1 function Person() {  
2  
3 }  
4 console.log(Person === Person.prototype.constructor); // true
```



所以，这里可以得到：

```
1 function Person() {  
2  
3 }  
4  
5 var person = new Person();  
6  
7 console.log(person.__proto__ == Person.prototype) // true  
8  
9 console.log(Person.prototype.constructor == Person) // true  
10  
11 console.log(Object.getPrototypeOf(person) === Person.prototype) // true
```

## 4.5 实例与原型

当读取实例的属性时，如果找不到，就会查找与对象关联的原型中的属性，如果还查不到，就去找原型的原型，一直找到最顶层为止。

举个例子：

```
1  function Person() {
2
3  }
4
5  Person.prototype.name = 'chenghuai';
6
7  var person = new Person();
8
9  person.name = 'huaicheng';
10 console.log(person.name) // huaicheng
11
12 delete person.name;
13 console.log(person.name) // chenghuai
```

在这个例子中，我们给实例对象 person 添加了 name 属性，当我们打印 `person.name` 的时候，结果自然为 `huaicheng`。

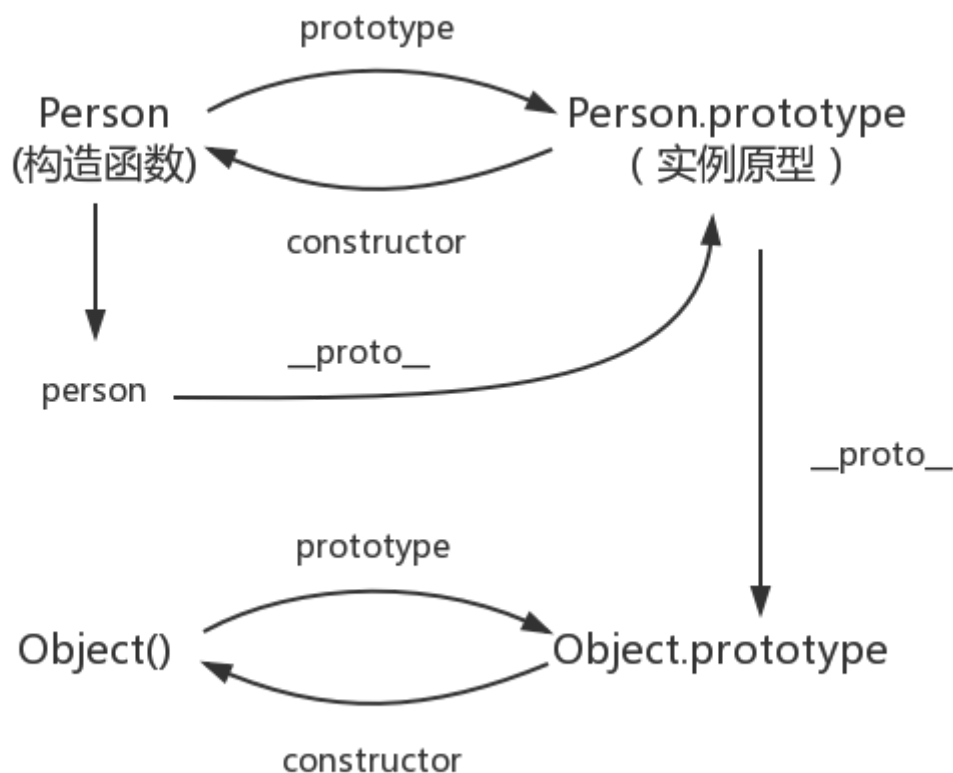
但是当我们删除了 person 的 name 属性时，读取 `person.name`，从 person 对象中找不到 name 属性就会从 person 的原型也就是 `person.__proto__`，也就是 `Person.prototype` 中查找，结果为 `chenghuai`。

## 4.6 原型的原型

如果在原型上还没有找到呢？原型的原型又是什么呢？

```
1  var obj = new Object();
2  obj.name = 'chenghuai'
3  console.log(obj.name) // chenghuai
```

其实原型对象就是通过 Object 构造函数生成的，结合之前所讲，实例的 `proto` 指向构造函数的 `prototype`，所以我们再更新下关系图：



## 4.7 原型链

那 `Object.prototype` 的原型呢？

null，我们可以打印：

```
1 console.log(Object.prototype.__proto__ === null) // true
```

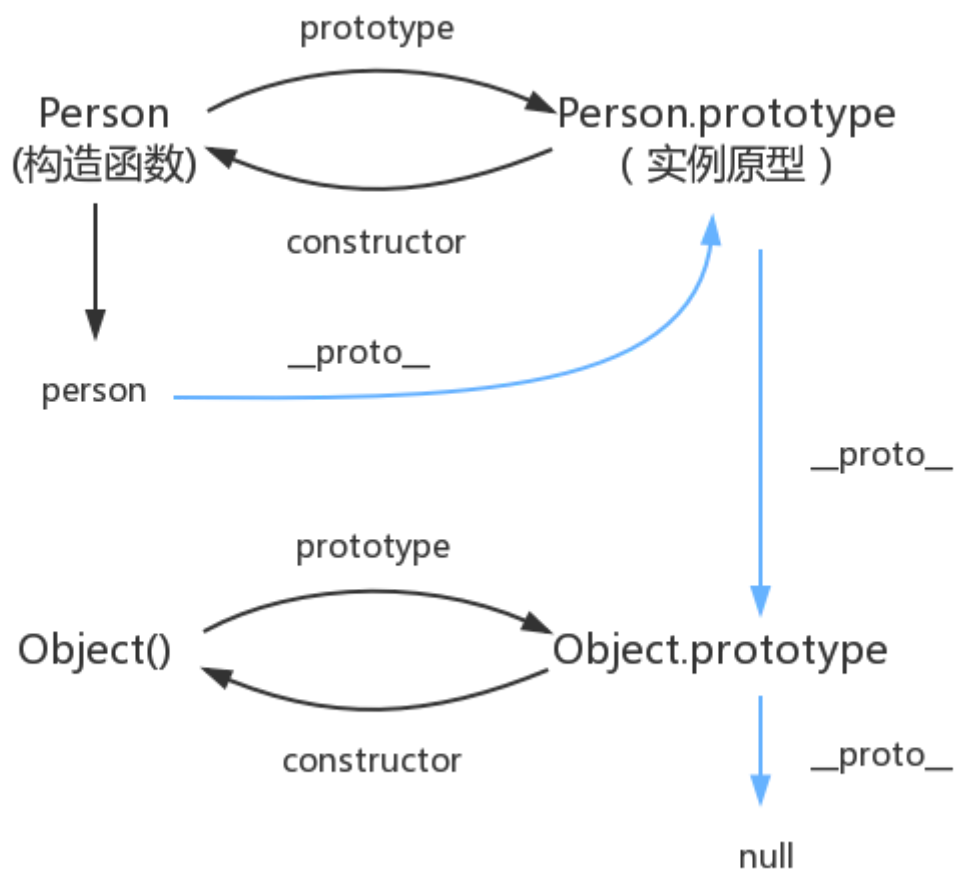
然而 null 究竟代表了什么呢？

null 表示“没有对象”，即该处不应该有值。

所以 `Object.prototype.__proto__` 的值为 null 跟 `Object.prototype` 没有原型，其实表达了一个意思。

所以查找属性的时候查到 `Object.prototype` 就可以停止查找了。

最后一张关系图也可以更新为：



其中，蓝色为原型链

## 4.8 其他

### 4.8.1 constructor

首先是 `constructor` 属性：

```
1  function Person() {
2
3  }
4  var person = new Person();
5
6  console.log(person.constructor === Person); // true
```

当获取 `person.constructor` 时，其实 `person` 中并没有 `constructor` 属性,当不能读取到 `constructor` 属性时，会从 `person` 的原型也就是 `Person.prototype` 中读取，正好原型中有该属性，所以：



```
1 person.constructor === Person.prototype.constructor
```

## 4.8.2 proto

绝大部分浏览器都支持这个非标准的方法访问原型，然而它并不存在于 `Person.prototype` 中，实际上，它是来自于 `Object.prototype`，与其说是一个属性，不如说是一个 `getter/setter`，当使用 `obj.__proto__` 时，可以理解成返回了 `Object.getPrototypeOf(obj)`。

## 4.8.3 继承

关于继承，前面提到“每一个对象都会从原型‘继承’属性”，实际上，继承是一个十分具有迷惑性的说法，引用《你不知道的JavaScript》中的话，就是：

继承意味着复制操作，然而 JavaScript 默认并不会复制对象的属性，相反，JavaScript 只是在两个对象之间创建一个关联，这样，一个对象就可以通过委托访问另一个对象的属性和函数，所以与其叫继承，委托的说法反而更准确些。

# 5. 词法作用域和动态作用域

## 5.1 作用域

作用域是指程序源代码中定义变量的区域。

作用域规定了如何查找变量，也就是确定当前执行代码对变量的访问权限。

JavaScript 采用词法作用域 (`lexical scoping`)，也就是静态作用域。

## 5.2 静态作用域和动态作用域

因为 JavaScript 采用的是词法作用域，函数的作用域在函数定义的时候就决定了。

而与词法作用域相对的是动态作用域，函数的作用域是在函数调用的时候才决定的。

```
1  var value = 1;
2
3  function foo() {
4      console.log(value);
5  }
6
7  function bar() {
8      var value = 2;
9      foo();
10 }
11
12 bar();
```

```
13
14 // 结果是 ???
```

假设JavaScript采用静态作用域，让我们分析下执行过程：

执行 foo 函数，先从 foo 函数内部查找是否有局部变量 value，如果没有，就根据书写的位置，查找上面一层的代码，也就是 value 等于 1，所以结果会打印 1。

假设JavaScript采用动态作用域，让我们分析下执行过程：

执行 foo 函数，依然是从 foo 函数内部查找是否有局部变量 value。如果没有，就从调用函数的作用域，也就是 bar 函数内部查找 value 变量，所以结果会打印 2。

前面我们已经说了，JavaScript采用的是静态作用域，所以这个例子的结果是 1。

## 5.3 动态作用域

什么语言是动态作用域？

bash 就是动态作用域，不信的话，把下面的脚本存成例如 scope.bash，然后进入相应的目录，用命令行执行 bash ./scope.bash，看看打印的值是多少。

```
1 value=1
2 function foo () {
3     echo $value;
4 }
5 function bar () {
6     local value=2;
7     foo;
8 }
9 bar
```

## 5.4 思考

看一个面试题：

```
1 // case 1
2 var scope = "global scope";
3 function checkscope(){
4     var scope = "local scope";
5     function f(){
6         return scope;
7     }
8     return f();
9 }
```

```
10  checkscope();
11
12  // case 2
13  var scope = "global scope";
14  function checkscope(){
15      var scope = "local scope";
16      function f(){
17          return scope;
18      }
19      return f;
20  }
21  checkscope()();
```

两段代码各自的执行结果是多少？

local scope

因为JavaScript采用的是词法作用域，函数的作用域基于函数创建的位置。

而引用《JavaScript权威指南》的回答就是：

JavaScript 函数的执行用到了作用域链，这个作用域链是在函数定义的时候创建的。嵌套的函数 f() 定义在这个作用域链里，其中的变量 scope 一定是局部变量，不管何时何地执行函数 f()，这种绑定在执行 f() 时依然有效。

但是在这里真正想让大家思考的是：

虽然两段代码执行的结果一样，但是两段代码究竟有哪些不同呢？

## 6. 执行上下文

### 6.1 顺序执行

写过 JavaScript 的开发者都会有个直观的印象，那就是顺序执行：

```
1  var foo = function () {
2
3      console.log('foo1');
4
5  }
6
7  foo(); // foo1
8
9  var foo = function () {
10
11      console.log('foo2');
12
```

```
13 }
14
15 foo(); // foo2
```

那这段呢？

```
1 function foo() {
2
3     console.log('foo1');
4
5 }
6
7 foo(); // foo2
8
9 function foo() {
10
11     console.log('foo2');
12
13 }
14
15 foo(); // foo2
```

打印的结果却是两个 foo2。

这是因为 JavaScript 引擎并非一行一行地分析和执行程序，而是一段一段地分析执行。当执行一段代码的时候，会进行一个“准备工作”，那这个“一段一段”中的“段”究竟是怎么划分的呢？

到底JavaScript引擎遇到一段怎样的代码时才会做“准备工作”呢？

```
1 console.log(add2(1,1)); //输出2
2 function add2(a,b){
3     return a+b;
4 }
5 console.log(add1(1,1)); //报错: add1 is not a function
6 var add1 = function(a,b){
7     return a+b;
8 }
9
10 // 用函数语句创建的函数add2，函数名称和函数体均被提前，在声明它之前就使用它。
11 // 但是使用var表达式定义函数add1，只有变量声明提前了，变量初始化代码仍然在原来的位置，没法提前执行。
```

## 6.2 可执行代码

这就要说到 JavaScript 的可执行代码( `executable code` )的类型有哪些了？

其实很简单，就三种，全局代码、函数代码、eval代码。

举个例子，当执行到一个函数的时候，就会进行准备工作，这里的“准备工作”，让我们用个更专业一点的说法，就叫做"执行上下文( `execution context` )"。

## 6.3 执行上下文栈

JavaScript 引擎创建了执行上下文栈（Execution context stack，ECS）来管理执行上下文

为了模拟执行上下文栈的行为，让我们定义执行上下文栈是一个数组：

```
1  ECStack = [];
```

试想当 JavaScript 开始要解释执行代码的时候，最先遇到的就是全局代码，所以初始化的时候首先就会向执行上下文栈压入一个全局执行上下文，我们用 `globalContext` 表示它，并且只有当整个应用程序结束的时候，ECStack 才会被清空，所以程序结束之前，ECStack 最底部永远有个 `globalContext`：

```
1  ECStack = [  
2      globalContext  
3  ];
```

当JavaScript 遇到下面的这段代码了：

```
1  function fun3() {  
2      console.log('fun3')  
3  }  
4  
5  function fun2() {  
6      fun3();  
7  }  
8  
9  function fun1() {  
10     fun2();  
11 }  
12  
13 fun1();
```

当执行一个函数的时候，就会创建一个执行上下文，并且压入执行上下文栈，当函数执行完毕的时候，就会将函数的执行上下文从栈中弹出。知道了这样的工作原理，让我们来看看如何处理上面这段代码：

```
1  // 伪代码
2
3  // fun1()
4  ECStack.push(<fun1> functionContext);
5
6  // fun1中竟然调用了fun2，还要创建fun2的执行上下文
7  ECStack.push(<fun2> functionContext);
8
9  // 擦，fun2还调用了fun3!
10 ECStack.push(<fun3> functionContext);
11
12 // fun3执行完毕
13 ECStack.pop();
14
15 // fun2执行完毕
16 ECStack.pop();
17
18 // fun1执行完毕
19 ECStack.pop();
20
21 // javascript接着执行下面的代码，但是ECStack底层永远有个globalContext
```

## 6.4 回顾上文

```
1  // case 1
2  var scope = "global scope";
3  function checkscope(){
4      var scope = "local scope";
5      function f(){
6          return scope;
7      }
8      return f();
9  }
10 checkscope();
11
12 // case 2
13 var scope = "global scope";
14 function checkscope(){
```

```
15     var scope = "local scope";
16     function f(){
17         return scope;
18     }
19     return f;
20 }
21 checkscope()();
```

两段代码执行的结果一样，但是两段代码究竟有哪些不同呢？

答案就是执行上下文栈的变化不一样。

模拟第一段代码：

```
1  ECStack.push(<checkscope> functionContext);
2  ECStack.push(<f> functionContext);
3  ECStack.pop();
4  ECStack.pop();
```

模拟第二段：

```
1  ECStack.push(<checkscope> functionContext);
2  ECStack.pop();
3  ECStack.push(<f> functionContext);
4  ECStack.pop();
```

这就是上文说到的区别。

## 7. 变量对象

### 7.1 基础

当 JavaScript 代码执行一段可执行代码(executable code)时，会创建对应的执行上下文(execution context)。

对于每个执行上下文，都有三个重要属性：

- 变量对象(Variable object，VO)；
- 作用域链(Scope chain)；
- this；

这里着重讲变量对象的内容

## 7.2 变量对象

变量对象是与执行上下文相关的数据作用域，存储了在上下文中定义的变量和函数声明。

因为不同执行上下文下的变量对象稍有不同，所以我们来聊聊全局上下文下的变量对象和函数上下文下的变量对象。

## 7.3 全局上下文

1. 全局对象是预定义的对象，作为 JavaScript 的全局函数和全局属性的占位符。通过使用全局对象，可以访问所有其他所有预定义的对象、函数和属性。
2. 在顶层 JavaScript 代码中，可以用关键字 `this` 引用全局对象。因为全局对象是作用域链的头，这意味着所有非限定性的变量和函数名都会作为该对象的属性来查询。
3. 例如，当 JavaScript 代码引用 `parseInt()` 函数时，它引用的是全局对象的 `parseInt` 属性。全局对象是作用域链的头，还意味着在顶层 JavaScript 代码中声明的所有变量都将成为全局对象的属性。

简单点说：

1. 可以通过 `this` 引用，在客户端 JavaScript 中，全局对象就是 `Window` 对象。

```
1 console.log(this);
```

2. 全局对象是由 `Object` 构造函数实例化的一个对象。

```
1 console.log(this instanceof Object);
```

3. 预定义的属性是否可用

```
1 console.log(Math.random());
2 console.log(this.Math.random());
```

4. 作为全局变量的宿主

```
1 var a = 1;
2 console.log(this.a);
```

5. 客户端 JavaScript 中，全局对象有 `window` 属性指向自身



```
1  var a = 1;
2  console.log(window.a);
3
4  this.window.b = 2;
5  console.log(this.b);
```

综上，对JS而言，全局上下文中的变量对象就是全局对象。

## 7.4 函数上下文

在函数上下文中，我们用活动对象(activation object, AO)来表示变量对象。

活动对象和变量对象其实是一个东西，只是变量对象是规范上的或者说是引擎实现上的，不可在JavaScript环境中访问，只有到当进入一个执行上下文中，这个执行上下文的变量对象才会被激活，所以才叫 activation object，而只有被激活的变量对象，也就是活动对象上的各种属性才能被访问。

活动对象是在进入函数上下文时刻被创建的，它通过函数的 arguments 属性初始化。arguments 属性值是 Arguments 对象。

## 7.5 执行过程

执行上下文的代码会分成两个阶段进行处理：分析和执行，我们也可以叫做：

1. 进入执行上下文；
2. 代码执行；

### 7.5.1 进入执行上下文

当进入执行上下文时，这时候还没有执行代码，

变量对象会包括：

1. 函数的所有形参 (如果是函数上下文)
  - 由名称和对应值组成的一个变量对象的属性被创建；
  - 没有实参，属性值设为 undefined；
2. 函数声明
  - 由名称和对应值（函数对象(function-object)）组成一个变量对象的属性被创建；
  - 如果变量对象已经存在相同名称的属性，则完全替换这个属性；
3. 变量声明
  - 由名称和对应值（undefined）组成一个变量对象的属性被创建；
  - 如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性；

举个例子：

```

1  function foo(a) {
2      var b = 2;
3      function c() {}
4      var d = function() {};
5
6      b = 3;
7
8  }
9
10 foo(1);

```

在进入执行上下文后，这时候的 AO 是：

```

1  AO = {
2      arguments: {
3          0: 1,
4          length: 1
5      },
6      a: 1,
7      b: undefined,
8      c: reference to function c() {},
9      d: undefined
10 }

```

## 7.5.2 代码执行

在代码执行阶段，会顺序执行代码，根据代码，修改变量对象的值

还是上面的例子，当代码执行完后，这时候的 AO 是：

```

1  AO = {
2      arguments: {
3          0: 1,
4          length: 1
5      },
6      a: 1,
7      b: 3,
8      c: reference to function c() {},
9      d: reference to FunctionExpression "d"
10 }

```

到这里变量对象的创建过程就介绍完了，让我们简洁的总结我们上述所说：

1. 全局上下文的变量对象初始化是全局对象；
2. 函数上下文的变量对象初始化只包括 Arguments 对象；
3. 在进入执行上下文时会给变量对象添加形参、函数声明、变量声明等初始的属性值；
4. 在代码执行阶段，会再次修改变量对象的属性值；

### 7.5.3 思考题

example 1

```
1  function foo() {  
2      console.log(a);  
3      a = 1;  
4  }  
5  
6  foo(); // ???  
7  
8  function bar() {  
9      a = 1;  
10     console.log(a);  
11 }  
12 bar(); // ???
```

第一段会报错：Uncaught ReferenceError: a is not defined。

第二段会打印：1。

这是因为函数中的 "a" 并没有通过 var 关键字声明，所有不会被存放在 AO 中。

第一段执行 console 的时候，AO 的值是：

```
1  AO = {  
2      arguments: {  
3          length: 0  
4      }  
5  }
```

没有 a 的值，然后就会到全局去找，全局也没有，所以会报错。

当第二段执行 console 的时候，全局对象已经被赋予了 a 属性，这时候就可以从全局找到 a 的值，所以会打印 1。

example 2

```
1 console.log(foo);
2
3 function foo(){
4     console.log("foo");
5 }
6
7 var foo = 1;
```

会打印函数，而不是 undefined。

这是因为在进入执行上下文时，首先会处理函数声明，其次会处理变量声明，如果变量名称跟已经声明的形式参数或函数相同，则变量声明不会干扰已经存在的这类属性。

## 8. 作用域链

上文讲到，当JavaScript代码执行一段可执行代码( `executable code` )时，会创建对应的执行上下文( `execution context` )。

对于每个执行上下文，都有三个重要属性：

- 变量对象(Variable object, VO)
- 作用域链(Scope chain)
- this

本节讲作用域链。

### 8.1 作用域链

上节讲到，当查找变量的时候，会先从当前上下文的变量对象中查找，如果没有找到，就会从父级(词法层面上的父级)执行上下文的变量对象中查找，一直找到全局上下文的变量对象，也就是全局对象。这样由多个执行上下文的变量对象构成的链表就叫做作用域链。

### 8.2 函数创建

上文的词法作用域与动态作用域中讲到，函数的作用域在函数定义的时候就决定了。

这是因为函数有一个内部属性 `[[scope]]`，当函数创建的时候，就会保存所有父变量对象到其中，你可以理解 `[[scope]]` 就是所有父变量对象的层级链，但是注意：`[[scope]]` 并不代表完整的作用域链！

举个例子：

```
1
2 function foo() {
3     function bar() {
4         ...
5     }
```

```
6 }
```

函数创建时，各自的[[scope]]为：

```
1
2  foo. [[scope]] = [
3    globalContext.V0
4  ];
5
6  bar. [[scope]] = [
7    fooContext.A0,
8    globalContext.V0
9  ];
10
```

## 8.3 函数激活

当函数激活时，进入函数上下文，创建 VO/AO 后，就会将活动对象添加到作用链的前端。

这时候执行上下文的作用域链，我们命名为 Scope：

```
1  Scope = [AO].concat([[Scope]]);
```

至此，作用域链创建完毕。

## 8.4 总结

结合着之前讲的变量对象和执行上下文栈，我们来总结一下函数执行上下文中作用域链和变量对象的创建过程：

```
1  var scope = "global scope";
2  function checkscope(){
3    var scope2 = 'local scope';
4    return scope2;
5  }
6  checkscope();
```

执行过程如下：

1.checkscope 函数被创建，保存作用域链到 内部属性[[scope]]

```
1  checkscope.[[scope]] = [  
2      globalContext.V0  
3  ];
```

2.执行 checkscope 函数，创建 checkscope 函数执行上下文，checkscope 函数执行上下文被压入执行上下文栈

```
1  ECStack = [  
2      checkscopeContext,  
3      globalContext  
4  ];
```

3.checkscope 函数并不立刻执行，开始做准备工作，第一步：复制函数[[scope]]属性创建作用域链

```
1  checkscopeContext = {  
2      Scope: checkscope.[[scope]],  
3  }
```

4.第二步：用 arguments 创建活动对象，随后初始化活动对象，加入形参、函数声明、变量声明

```
1  checkscopeContext = {  
2      AO: {  
3          arguments: {  
4              length: 0  
5          },  
6          scope2: undefined  
7      },  
8      Scope: checkscope.[[scope]],  
9  }
```

5.第三步：将活动对象压入 checkscope 作用域链顶端

```
1  checkscopeContext = {  
2      AO: {  
3          arguments: {  
4              length: 0  
5          },  
6          scope2: undefined  
7      },
```

```
8     Scope: [AO, [[Scope]]]
9 }
```

6.准备工作做完，开始执行函数，随着函数的执行，修改 AO 的属性值

```
1  checkscopeContext = {
2      AO: {
3          arguments: {
4              length: 0
5          },
6          scope2: 'local scope'
7      },
8      Scope: [AO, [[Scope]]]
9  }
```

7.查找到 scope2 的值，返回后函数执行完毕，函数上下文从执行上下文栈中弹出

```
1  ECStack = [
2      globalContext
3  ];
```

## 9. this

对于每个执行上下文，都有三个重要属性

- 变量对象(Variable object, VO)
- 作用域链(Scope chain)
- this

本节主要讲this

### 9.1 Types

Types are further subclassified into ECMAScript language types and specification types.

An ECMAScript language type corresponds to values that are directly manipulated by an ECMAScript programmer using the ECMAScript language. The ECMAScript language types are Undefined, Null, Boolean, String, Number, and Object.

A specification type corresponds to meta-values that are used within algorithms to describe the semantics of ECMAScript language constructs and ECMAScript language types. The

specification types are Reference, List, Completion, Property Descriptor, Property Identifier, Lexical Environment, and Environment Record.

我们简单的翻译一下：

ECMAScript 的类型分为语言类型和规范类型。

ECMAScript 语言类型是开发者直接使用 ECMAScript 可以操作的。其实就是我们常说的Undefined, Null, Boolean, String, Number, 和 Object。

而规范类型相当于 meta-values，是用来用算法描述 ECMAScript 语言结构和 ECMAScript 语言类型的。规范类型包括：Reference, List, Completion, Property Descriptor, Property Identifier, Lexical Environment, 和 Environment Record。

我们只要知道在 ECMAScript 规范中还有一种只存在于规范中的类型，它们的作用是用来描述语言底层行为逻辑。

这里要讲的重点是便是其中的 Reference 类型。它与 this 的指向有着密切的关联。

## 9.2 Reference

那什么又是 Reference ？

The Reference type is used to explain the behaviour of such operators as delete, typeof, and the assignment operators.

所以 Reference 类型就是用来解释诸如 delete、typeof 以及赋值等操作行为的。

这里的 Reference 是一个 Specification Type，也就是“只存在于规范里的抽象类型”。它们是为了更好地描述语言的底层行为逻辑才存在的，但并不存在于实际的 js 代码中。

再看接下来的这段具体介绍 Reference 的内容：

A Reference is a resolved name binding.

A Reference consists of three components, the base value, the referenced name and the Boolean valued strict reference flag.

The base value is either undefined, an Object, a Boolean, a String, a Number, or an environment record (10.2.1).

A base value of undefined indicates that the reference could not be resolved to a binding. The referenced name is a String.

这段讲述了 Reference 的构成，由三个组成部分，分别是：

- base value;
- referenced name;
- strict reference;

可是这些到底是什么呢？



我们简单的理解的话：

base value 就是属性所在的对象或者就是 EnvironmentRecord，它的值只可能是 undefined, an Object, a Boolean, a String, a Number, or an environment record 其中的一种。

referenced name 就是属性的名称。

举个例子：

```
1  var foo = 1;
2
3  // 对应的Reference是:
4  var fooReference = {
5      base: EnvironmentRecord,
6      name: 'foo',
7      strict: false
8  };
```

```
1  var foo = {
2      bar: function () {
3          return this;
4      }
5  };
6
7  foo.bar(); // foo
8
9  // bar对应的Reference是:
10 var BarReference = {
11     base: foo,
12     propertyName: 'bar',
13     strict: false
14 };
```

而且规范中还提供了获取 Reference 组成部分的方法，比如 GetBase 和 IsPropertyReference。

### 9.2.1 GetBase

GetBase(V). Returns the base value component of the reference V.

返回 reference 的 base value。

```
1  var foo = 1;
2
3  var fooReference = {
```

```
4     base: EnvironmentRecord,  
5     name: 'foo',  
6     strict: false  
7   };  
8  
9   GetValue(fooReference) // 1;
```

GetValue 返回对象属性真正的值，但是，调用 GetValue，返回的将是具体的值，而不再是一个 Reference

## 9.2.2 IsPropertyReference

IsPropertyReference(V). Returns true if either the base value is an object or HasPrimitiveBase(V) is true; otherwise returns false.

如果 base value 是一个对象，就返回true。

## 9.3 如何确定this的值

关于 Reference 讲了那么多，为什么要讲 Reference 呢？到底 Reference 跟本文的主题 this 有哪些关联呢？

- Let ref be the result of evaluating MemberExpression;
- if Type(ref) is Reference, then
  - If IsPropertyReference(ref) is true, then
  - Let thisValue be GetBase(ref).
- Else, the base of ref is an Environment Record
  - Let thisValue be the result of calling the ImplicitThisValue concrete method of GetBase(ref).
- Else, Type(ref) is not Reference.
- Let thisValue be undefined.

让我们描述一下：

1. 计算 MemberExpression 的结果赋值给 ref;
2. 判断 ref 是不是一个 Reference 类型;
  - a. 如果 ref 是 Reference，并且 IsPropertyReference(ref) 是 true, 那么 this 的值为 GetBase(ref)
  - b. 如果 ref 是 Reference，并且 base value 值是 Environment Record, 那么this的值为 ImplicitThisValue(ref)
  - c. 如果 ref 不是 Reference，那么 this 的值为 undefined;

## 9.4 具体分析

### 9.4.1 计算 MemberExpression 的结果赋值给 ref

什么是 MemberExpression? 看规范 11.2 Left-Hand-Side Expressions:

MemberExpression:

- PrimaryExpression // 原始表达式
- FunctionExpression // 函数定义表达式
- MemberExpression [ Expression ] // 属性访问表达式
- MemberExpression . IdentifierName // 属性访问表达式
- new MemberExpression Arguments // 对象创建表达式

举个例子:

```
1  function foo() {
2      console.log(this)
3  }
4
5  foo(); // MemberExpression 是 foo
6
7  function foo() {
8      return function() {
9          console.log(this)
10     }
11 }
12
13 foo()(); // MemberExpression 是 foo()
14
15 var foo = {
16     bar: function () {
17         return this;
18     }
19 }
20
21 foo.bar(); // MemberExpression 是 foo.bar
```

所以简单理解 MemberExpression 其实就是()左边的部分。

### 9.4.2 判断 ref 是不是一个 Reference 类型。

关键就在于看规范是如何处理各种 MemberExpression，返回的结果是不是一个 Reference 类型。

```

1  var value = 1;
2
3  var foo = {
4    value: 2,
5    bar: function () {
6      return this.value;
7    }
8  }
9
10 //示例1
11 console.log(foo.bar());
12 //示例2
13 console.log((foo.bar)());
14 //示例3
15 console.log((foo.bar = foo.bar)());
16 //示例4
17 console.log((false || foo.bar)());
18 //示例5
19 console.log((foo.bar, foo.bar)());

```

### 9.4.3 foo.bar()

上面的demo种，`MemberExpression` 计算的结果是 `foo.bar`，那么 `foo.bar` 是不是一个 Reference 呢？

根据规范，这里展示了一个计算的过程，什么都不管了，就看最后一步：

Return a value of type Reference whose base value is baseValue and whose referenced name is propertyNameString, and whose strict mode flag is strict.

我们得知该表达式返回了一个 Reference 类型

根据之前的内容，我们知道该值为：

```

1  var Reference = {    base: foo,    name: 'bar',    strict: false };

```

接下来按照流程：

1. 如果 ref 是 Reference，并且 `IsPropertyReference(ref)` 是 true，那么 this 的值为 `GetBase(ref)`

该值是 Reference 类型，那么 `IsPropertyReference(ref)` 的结果是多少呢？

前面我们说了 `IsPropertyReference` 方法，如果 base value 是一个对象，结果返回 true。

base value 为 foo，是一个对象，所以 `IsPropertyReference(ref)` 结果为 true。

这个时候我们就可以确定 this 的值：

```
1  this = GetBase(ref),
```

GetBase 也已经铺垫了，获得 base value 值，这个例子中就是foo，所以 this 的值就是 foo，示例1的结果就是 2。

#### 9.4.4 (foo.bar)()

```
1  console.log((foo.bar)());
```

foo.bar 被 () 包住

Return the result of evaluating Expression. This may be of type Reference.

NOTE This algorithm does not apply GetValue to the result of evaluating Expression.

实际上 () 并没有对 `MemberExpression` 进行计算，所以其实跟示例 1 的结果是一样的。

#### 9.4.5 (foo.bar = foo.bar)()

看示例3，有赋值操作符，

因为使用了 GetValue，所以返回的值不是 Reference 类型，

按照之前讲的判断逻辑，如果 ref 不是Reference，那么 this 的值为 undefined

this 为 undefined，非严格模式下，this 的值为 undefined 的时候，其值会被隐式转换为全局对象。

#### 9.4.6 (false || foo.bar)()

示例4，因为使用了 GetValue，所以返回的不是 Reference 类型，this 为 undefined。

#### 9.4.7 (foo.bar, foo.bar)()

看示例5，因为使用了 GetValue，所以返回的不是 Reference 类型，this 为 undefined。

#### 9.4.8 总结

```
1
2  var value = 1;
3
4  var foo = {
```

```

5     value: 2,
6     bar: function () {
7         return this.value;
8     }
9 }
10
11 //示例1
12 console.log(foo.bar()); // 2
13 //示例2
14 console.log((foo.bar)()); // 2
15 //示例3
16 console.log((foo.bar = foo.bar)()); // 1
17 //示例4
18 console.log((false || foo.bar)()); // 1
19 //示例5
20 console.log((foo.bar, foo.bar)()); // 1
21

```

注意：以上是在非严格模式下的结果，严格模式下因为 this 返回 undefined，所以示例 3 会报错。

## 10. 执行上下文

### 10.1 思考题

```

1  var scope = "global scope";
2  function checkscope(){
3      var scope = "local scope";
4      function f(){
5          return scope;
6      }
7      return f();
8  }
9  checkscope();

```

```

1  var scope = "global scope";
2  function checkscope(){
3      var scope = "local scope";
4      function f(){
5          return scope;
6      }
7      return f;
8  }

```

```
9  checkscope()();
```

两段代码都会打印'local scope'，在上文讲到了两者的区别在于执行上下文栈的变化不一样，本节会在此基础上，详细的解析执行上下文栈和执行上下文的具体变化过程。

## 10.2 具体执行分析

我们分析第一段代码：

```
1  var scope = "global scope";
2  function checkscope(){
3      var scope = "local scope";
4      function f(){
5          return scope;
6      }
7      return f();
8  }
9  checkscope();
```

执行过程如下：

1. 执行全局代码，创建全局执行上下文，全局上下文被压入执行上下文栈

```
1  ECStack = [
2      globalContext
3  ];
```

2. 全局上下文初始化

```
1  globalContext = {
2      VO: [global],
3      Scope: [globalContext.VO],
4      this: globalContext.VO
5  }
```

3. 初始化的同时，checkscope 函数被创建，保存作用域链到函数的内部属性[[scope]]

```
1  checkscope.[[scope]] = [
2      globalContext.VO
3  ];
```

4. 执行 checkscope 函数，创建 checkscope 函数执行上下文，checkscope 函数执行上下文被压入执行上下文栈

```
1 ECStack = [  
2     checkscopeContext,  
3     globalContext  
4 ];
```

5. checkscope 函数执行上下文初始化：

- a. 复制函数 [[scope]] 属性创建作用域链；
- b. 用 arguments 创建活动对象；
- c. 初始化活动对象，即加入形参、函数声明、变量声明；
- d. 将活动对象压入 checkscope 作用域链顶端；

同时 f 函数被创建，保存作用域链到 f 函数的内部属性 [[scope]]

```
1 checkscopeContext = {  
2     AO: {  
3         arguments: {  
4             length: 0  
5         },  
6         scope: undefined,  
7         f: reference to function f(){}  
8     },  
9     Scope: [AO, globalContext.V0],  
10    this: undefined  
11 }
```

6. f 函数执行，沿着作用域链查找 scope 值，返回 scope 值；
7. f 函数执行完毕，f 函数上下文从执行上下文栈中弹出；

```
1 ECStack = [  
2     checkscopeContext,  
3     globalContext  
4 ];
```

8. checkscope 函数执行完毕，checkscope 函数执行上下文从执行上下文栈中弹出



```
1 ECStack = [  
2     globalContext  
3 ];
```

## 11. 闭包

MDN 对闭包的定义为：

闭包是指那些能够访问自由变量的函数。

那什么是自由变量呢？

自由变量是指在函数中使用的，但既不是函数参数也不是函数的局部变量的变量。

由此，我们可以看出闭包共有两部分组成：

闭包 = 函数 + 函数能够访问的自由变量

```
1 var a = 1;  
2  
3 function foo() {  
4     console.log(a);  
5 }  
6  
7 foo();
```

foo 函数可以访问变量 a，但是 a 既不是 foo 函数的局部变量，也不是 foo 函数的参数，所以 a 就是自由变量。

所以在《JavaScript权威指南》中就讲到：从技术的角度讲，所有的JavaScript函数都是闭包。

但是，这是理论上的闭包，其实还有一个实践角度上的闭包。

ECMAScript中，闭包指的是：

1. 从理论角度：所有的函数。因为它们都在创建的时候就将上层上下文的数据保存起来了。哪怕是简单的全局变量也是如此，因为函数中访问全局变量就相当于是在访问自由变量，这个时候使用最外层的作用域；
2. 从实践角度：以下函数才算是闭包：
  - a. 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）；
  - b. 在代码中引用了自由变量；

接下来就来讲讲实践上的闭包。

## 11.1 分析

```
1  var scope = "global scope";
2  function checkscope(){
3      var scope = "local scope";
4      function f(){
5          return scope;
6      }
7      return f;
8  }
9
10 var foo = checkscope();
11 foo();
```

先我们要分析一下这段代码中执行上下文栈和执行上下文的变化情况。

这里直接给出简要的执行过程：

1. 进入全局代码，创建全局执行上下文，全局执行上下文压入执行上下文栈；
2. 全局执行上下文初始化；
3. 执行 `checkscope` 函数，创建 `checkscope` 函数执行上下文，`checkscope` 执行上下文被压入执行上下文栈；
4. `checkscope` 执行上下文初始化，创建变量对象、作用域链、`this`等；
5. `checkscope` 函数执行完毕，`checkscope` 执行上下文从执行上下文栈中弹出；
6. 执行 `f` 函数，创建 `f` 函数执行上下文，`f` 执行上下文被压入执行上下文栈；
7. `f` 执行上下文初始化，创建变量对象、作用域链、`this`等；
8. `f` 函数执行完毕，`f` 函数上下文从执行上下文栈中弹出；

了解到这个过程，我们应该思考一个问题：

1. 当 `f` 函数执行的时候，`checkscope` 函数上下文已经被销毁了啊(即从执行上下文栈中被弹出)，怎么还会读取到 `checkscope` 作用域下的 `scope` 值呢？

当我们了解了具体的执行过程后，我们知道 `f` 执行上下文维护了一个作用域链：

```
1  fContext = {
2      Scope: [A0, checkscopeContext.A0, globalContext.V0],
3  }
```

因为这个作用域链，f 函数依然可以读取到 `checkscopeContext.A0` 的值，说明当 f 函数引用了 `checkscopeContext.A0` 中的值的时候，即使 `checkscopeContext` 被销毁了，但是 JavaScript 依然会让 `checkscopeContext.A0` 活在内存中，f 函数依然可以通过 f 函数的作用域链找到它，正是因为 JavaScript 做到了这一点，从而实现了闭包这个概念。

所以，让我们再看一遍实践角度上闭包的定义：

1. 即使创建它的上下文已经销毁，它仍然存在（比如，内部函数从父函数中返回）；
2. 在代码中引用了自由变量；

## 11.2 思考题

```
1  var data = [];  
2  
3  for (var i = 0; i < 3; i++) {  
4      data[i] = function () {  
5          console.log(i);  
6      };  
7  }  
8  
9  data[0]();  
10 data[1]();  
11 data[2]();
```

答案是都是 3，让我们分析一下原因：

当执行到 `data[0]` 函数之前，此时全局上下文的 VO 为：

```
1  globalContext = {  
2      VO: {  
3          data: [...],  
4          i: 3  
5      }  
6  }
```

当执行 `data[0]` 函数的时候，`data[0]` 函数的作用域链为：

```
1  data[0]Context = {  
2      Scope: [A0, globalContext.VO]  
3  }
```

data[0]Context 的 AO 并没有 i 值，所以会从 globalContext.VO 中查找，i 为 3，所以打印的结果就是 3。

data[1] 和 data[2] 是一样的道理。

所以改成闭包：

```
1  var data = [];  
2  
3  for (var i = 0; i < 3; i++) {  
4      data[i] = (function (i) {  
5          return function(){  
6              console.log(i);  
7          }  
8      })(i);  
9  }  
10  
11 data[0]();  
12 data[1]();  
13 data[2]();
```

当执行到 data[0] 函数之前，此时全局上下文的 VO 为：

```
1  globalContext = {  
2      VO: {  
3          data: [...],  
4          i: 3  
5      }  
6  }
```

跟没改之前一模一样。

当执行 data[0] 函数的时候，data[0] 函数的作用域链发生了改变：

```
1  data[0]Context = {  
2      Scope: [AO, 匿名函数Context.AO globalContext.VO]  
3  }
```

匿名函数执行上下文的AO为：

```
1  匿名函数Context = {
```

```
2      AO: {  
3          arguments: {  
4              0: 0,  
5              length: 1  
6          },  
7          i: 0  
8      }  
9  }
```

data[0]Context 的 AO 并没有 i 值，所以会沿着作用域链从匿名函数 Context.AO 中查找，这时候就会找 i 为 0，找到了就不会往 globalContext.VO 中查找了，即使 globalContext.VO 也有 i 的值(值为3)，所以打印的结果就是0。

data[1] 和 data[2] 是一样的道理。