**讲透bind**

**特点：**

* bind() 方法会创建一个新函数。当这个新函数被调用时，bind() 的第一个参数将作为它运行时的 this，之后的一序列参数将会在传递的实参前传入作为它的参数。
* 一个绑定函数也能使用new操作符创建对象：这种行为就像把原函数当成构造器。提供的 this 值被略，同时调用时的参数被提供给模拟函数。

**mdn的Polyfill**

1. if (!Function.prototype.bind) {
2. //防止Function.prototype.bind.call(obj,param)这种调用改变this
3. Function.prototype.bind = function(oThis) {
4. if (typeof this !== 'function') {
5. throw new TypeError('Function.prototype.bind - what is trying to be bound is not callable');
6. }
7. var aArgs = Array.prototype.slice.call(arguments, 1),
8. fToBind = this,
9. fNOP = function() {},
10. fBound = function() {
11. // this instanceof fBound === true时,说明返回的fBound被当做new的构造函数调用
12. return fToBind.apply(this instanceof fBound
13. ? this
14. : oThis,
15. // 获取调用时(fBound)的传参.bind 返回的函数入参往往是这么传递的
16. aArgs.concat(Array.prototype.slice.call(arguments)));
17. };
18. // 维护原型关系
19. if (this.prototype) {
20. // 当执行Function.prototype.bind()时, this为Function.prototype
21. // this.prototype(即Function.prototype.prototype)为undefined
22. fNOP.prototype = this.prototype;
23. }
24. // 下行的代码使fBound.prototype是fNOP的实例,因此
25. // 返回的fBound若作为new的构造函数,new生成的新对象作为this传入fBound,新对象的\_\_proto\_\_就是fNOP的实例
26. fBound.prototype = new fNOP();
27. return fBound;
28. };}

**词法作用域：**

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

* 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。

**执行顺序：**

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

然而去看这段代码：

1. function foo() {
2. console.log('foo1');
3. }
4. foo(); // foo2
5. function foo() {
6. console.log('foo2');
7. }
8. foo(); // foo2

可执行代码(executable code)的类型: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)"。

**执行上下文栈:**

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

**变量对象:**

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

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。

**全局上下文:**

全局上下文中的变量对象就是全局对象！

**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 呐，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。

**执行过程:**

1. 进入执行上下文
2. 代码执行

**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

1. 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

1. 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

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

**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. }

**作用域链**

* 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。
* 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！

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

函数创建时，各自的[[scope]]为：

1. foo.[[scope]] = [
2. globalContext.VO
3. ];
5. bar.[[scope]] = [
6. fooContext.AO,
7. globalContext.VO
8. ];

**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 ScopeChain

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

至此，作用域链创建完毕。

**this**

其实很复杂，简单来说是谁调用指向谁

**具体执行分析**

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. checkscope();

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：

1. 复制函数 [[scope]] 属性创建作用域链，
2. 用 arguments 创建活动对象，
3. 初始化活动对象，即加入形参、函数声明、变量声明，
4. 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.VO],
10. this: undefined
11. }

6.执行 f 函数，创建 f 函数执行上下文，f 函数执行上下文被压入执行上下文栈

1. ECStack = [
2. fContext,
3. checkscopeContext,
4. globalContext
5. ];

7.f 函数执行上下文初始化, 以下跟第 4 步相同：

1. 复制函数 [[scope]] 属性创建作用域链
2. 用 arguments 创建活动对象
3. 初始化活动对象，即加入形参、函数声明、变量声明
4. 将活动对象压入 f 作用域链顶端
5. fContext = {
6. AO: {
7. arguments: {
8. length: 0
9. }
10. },
11. Scope: [AO, checkscopeContext.AO, globalContext.VO],
12. this: undefined
13. }

8.f 函数执行，沿着作用域链查找 scope 值，返回 scope 值

9.f 函数执行完毕，f 函数上下文从执行上下文栈中弹出

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

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

**闭包**

**定义：**

1. 闭包是指那些能够访问自由变量的函数（MDN）、闭包是指有权访问另外一个函数作用域中的变量的函数（红宝书p178）、《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。
2. 从实践角度：以下函数才算是闭包：
3. 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）
4. 在代码中引用了自由变量
5. **例子：**
6. var scope = "global scope";
7. function checkscope(){
8. var scope = "local scope";
9. function f(){
10. return scope;
11. }
12. return f;
13. }
14. var foo = checkscope();
15. foo();
16. 这里直接给出简要的执行过程：
17. 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
18. 全局执行上下文初始化
19. 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
20. checkscope 执行上下文初始化，创建变量对象、作用域链、this等
21. checkscope 函数执行完毕，checkscope 执行上下文从执行上下文栈中弹出
22. 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
23. f 执行上下文初始化，创建变量对象、作用域链、this等
24. 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，那就是：

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？

我们知道 f 执行上下文维护了一个作用域链：

1. fContext = {
2. Scope: [AO, checkscopeContext.AO, globalContext.VO],
3. }

**call和apply的模拟实现**

call() 方法在使用一个指定的 this 值和若干个指定的参数值的前提下调用某个函数或方法。

1. Function.prototype.call2 = function (context) {
2. var context = context || window;
3. context.fn = this;
4. var args = [];
5. for(var i = 1, len = arguments.length; i < len; i++) {
6. args.push('arguments[' + i + ']');
7. }
8. var result = eval('context.fn(' + args +')');
9. delete context.fn
10. return result;
11. }

14. Function.prototype.apply = function (context, arr) {
15. var context = Object(context) || window;
16. context.fn = this;
17. var result;
18. if (!arr) {
19. result = context.fn();
20. }
21. else {
22. var args = [];
23. for (var i = 0, len = arr.length; i < len; i++) {
24. args.push('arr[' + i + ']');
25. }
26. result = eval('context.fn(' + args + ')')
27. }
28. delete context.fn
29. return result;
30. }

**New**

new 运算符创建一个用户定义的对象类型的实例或具有构造函数的内置对象类型之一

1. function objectFactory() {
2. var obj = new Object(),
3. Constructor = [].shift.call(arguments);
4. obj.\_\_proto\_\_ = Constructor.prototype;
5. var ret = Constructor.apply(obj, arguments);
6. return typeof ret === 'object' ? ret : obj;
7. };
8. function Otaku (name, age) {
9. this.name = name;
10. this.age = age;
12. this.habit = 'Games';
13. }
14. Otaku.prototype.strength = 60;
15. Otaku.prototype.sayYourName = function () {
16. console.log('I am ' + this.name);
17. }
18. var person = objectFactory(Otaku, 'Kevin', '18')
19. console.log(person.name) // Kevin
20. console.log(person.habit) // Games
21. console.log(person.strength) // 60
22. console.log(person.age) // 18
23. console.log(person.sayYourName())//I am Kevin

**柯里化**

在数学和计算机科学中，柯里化是一种将使用多个参数的一个函数转换成一系列使用一个参数的函数的技术。

乞丐版：完整版很深

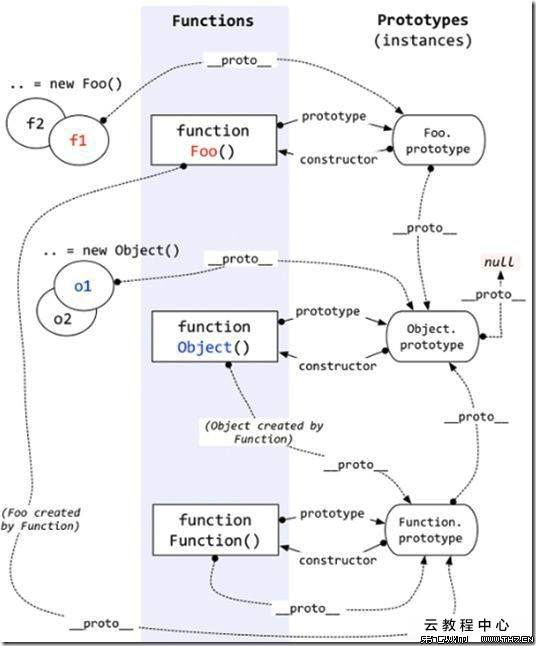
1. var curry = function (fn) {
2. var args = [].slice.call(arguments, 1);
3. return function() {
4. var newArgs = args.concat([].slice.call(arguments));
5. return fn.apply(this, newArgs);
6. };
7. };
9. function add(a, b) {
10. return a + b;
11. }
12. var addCurry = curry(add, 1, 2);
13. addCurry() // 3//或者var addCurry = curry(add, 1);addCurry(2) // 3//或者var addCurry = curry(add);addCurry(1, 2) // 3

**继承**

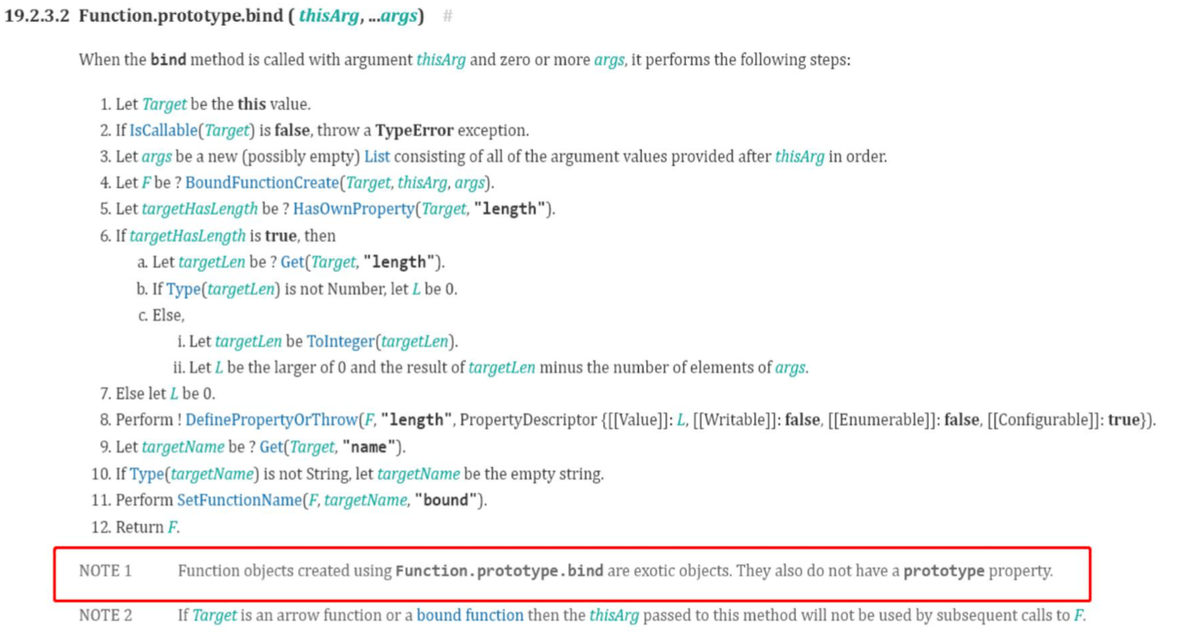
其他都是浮云，一招搞定

1. function Rectangle(length,width){
2. this.l = length
3. this.w = width
4. }
5. Rectangle.prototype.getArea = function(){
6. return this.l\*this.w
7. }
8. function Square(length){
9. Rectangle.call(this,length,length)
10. }
11. Square.prototype = Object.create(Rectangle.prototype,{
12. constructor:{
13. value:Square
14. }
15. })
16. var square = new Square(3)
17. console.log(square.getArea())
18. console.log(square instanceof Square)
19. console.log(square instanceof Rectangle)

**原型链**



**后续**

****