# JSCORE04

## 复习

### 闭包

* 什么是闭包: JS的函数具有特殊的设计 -- 在声明时会自动保存其所在的词法环境
  + Scopes: 此属性,就是函数中 用来保存词法环境的属性
    - 此变量中会保存多个属性, 各种作用域, 按照就近原则排序 -- 作用域链
* 有什么用?
* 因为函数具有自己的作用域, 独立在 全局作用域之外.
* 如果期望声明变量, 又不想把变量保存在全局区域--会造成全局污染!
* 此时就需要用函数来创建一个独立的作用域空间 -- 完成这种作用的函数 就叫闭包!
* 闭包: 封闭的包围
* 所有的函数都具有闭包的条件, 就看如何使用.
  1. 如果单纯为了复用代码, 就是普通函数
  2. 如果为了创建作用域 保存变量 , 就是闭包!
* 写法
* // 推荐使用匿名函数自调书写: 不是必备的 , 因为比命名函数节省代码和内存
    
  var b = (function (){
    
   var name ='123123';
    
     
   // 只有被window保存的属性, 才能活下来. 否则会自动释放!
    
   // 想让 name变量不被释放, 则需要让使用name的东西, 存储在window里
    
   window.a = function (){
    
   console.log(name)
    
   }
    
     
   return function (){
    
   console.log(name)
    
   }
    
  })()

### 原型

* JS独有的一个设计: 同JAVA的继承机制
  + 每个对象都有一个 \_\_proto\_\_ 属性, 其中保存了此对象的一些基础通用方法
  + JS引擎的执行机制: 当对对象读取一个属性时, 如果对象没有, 则到其原型中查找, 原型没有, 就到原型的原型中查找... 直到最根位置:
    - 最根的原型就是系统的 Object构造函数的 prototype
* 两个称呼问题:
  + prototype: 构造函数中的原型, 叫 prototype
  + \_\_proto\_\_: 对象里的原型叫 \_\_proto\_\_
  + 本质是同一个对象, 因为 对象的 \_\_proto\_\_ 是构造函数 在构造对象的时候, 把 prototype 赋值给了 那个对象 对象.\_\_proto\_\_ = 构造函数.prototype
* 原型链: prototype chain
  + JS引擎的执行机制: 当对对象读取一个属性时, 如果对象没有, 则到其原型中查找, 原型没有, 就到原型的原型中查找... 直到最根位置:

### 严格模式

在脚本中书写 "use strict"; 下方的所有代码都会开启严格模式!

* 为对象替换原型: Object.setPrototypeOf(对象, 原型对象)
  + 只会为当前对象替换原型, 不会影响其他 同构造函数 构造的对象
* 为构造函数替换原型:构造函数.prototype = 新原型对象
* for..in.. : 会遍历 原型链中 所有的属性 -- 不含不可遍历的属性
  + Object.keys: 只读取对象自身的属性, 不读原型
    - 利用此方法来判断空对象: Object.keys(对象).length==0
* 严格模式: 开启之后可以让系统帮你规避很多错误, 以后所有的框架 默认都会开严格模式!
  + 防止意外的全局变量声明: 变量名 = 值; 被禁止使用, 必须用 var let const声明
  + 同 为了方式全局变量声明: 全局中的函数, 其中的this指向undefined 而不是 window
    - 构造函数应该是new 调用的, 一旦忘记写new了, 则不会向window注入属性
  + 静默失败 改为 不静默!
    - 很多失败, 例如修改只读属性的值, 之前不报错. 严格模式下会报错!
  + callee: 是 arguments的一个属性, 保存了当前执行的函数
    - 用途: 匿名函数做递归时, 函数内部代表当前函数 -- 性能低下
    - 替代方式; 用命名函数来实现递归函数即可!
* 精确配置属性:
* 每个属性都可以有 6 个配置项
  + value: 默认值
  + configurable: 是否可以重新配置
  + enumerable: 是否可以遍历
  + writable: 是否可写
  + get: getter, 读取属性时触发 对象.属性名
    - 作用: 计算属性
  + set: setter, 为属性赋值时触发 对象.属性名 = 值
    - 作用: 赋值检测 -- 如果属性带有一些要求: 年龄(1-100) 手机号(格式..)
* 直接声明属性: {属性名: 值} 其配置项默认都是true, 可写,可改,可遍历
* 用Object.defineProperty(对象, 属性名, 配置项): 新增属性所有配置默认是false, 不可写, 不可改, 不可遍历
* get 和 set 属性 与 write 和 enumerable : 互斥, 不能同时存在
* 同时配置多个属性:Object.defineProperties(对象, {多个属性...})

## let 关键词

// let 和 var 的差别
  
// let 是 ES6中提供的 代替var 的 变量声明方式
  
// var的缺点: 变量提升+ var声明的变量都会存储在window中, 造成全局污染
  
var aaa = 123123;
  
   
// let: 没有全局污染, 声明的变量存储在一个与window同级别的 脚本区域, 专门存储自定义变量
  
let abb = 123123;
  
let abb = 2342;

## 保护属性的方法

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 "use strict";
  
 // 保护对象的方法:
  
 // 有时候我们需要把整个对象都保护起来
  
 var yanan = {
  
 name: "亚楠",
  
 age: 19,
  
 phone: "13434435548",
  
 };
  
   
 // 保护级别1 - 阻止扩展: 不能新增属性
  
 Object.preventExtensions(yanan);
  
 // Cannot add property boyFriend, object is not extensible
  
 // yanan.boyFriend = "彭于晏"; //无法新增
  
 yanan.age = 23;
  
 delete yanan.age; //删除属性
  
 console.log(yanan);
  
   
 // 保护级别2 - 不能增删
  
 // seal: 密封
  
 Object.seal(yanan);
  
 // yanan.boyFriend = "吴彦祖";
  
 // Cannot delete property 'name'
  
 // delete yanan.name;
  
   
 // 级别3 - 冻结: freeze 不能 增删改
  
 Object.freeze(yanan);
  
 // yanan.bf = "阿坤";
  
 // delete yanan.name;
  
   
 // Cannot assign to read only property 'name'
  
 yanan.name = 33; //无法修改只读属性
  
 </script>
  
 </body>
  
</html>

## 创建对象&指定原型

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 创建对象 并 指定原型
  
   
 // setPrototypeOf(): 可以为已存在的一个对象, 设置原型
  
 // 此方法执行要分两步
  
 // 1. 先有一个对象
  
 // 2. 再为这个对象指定原型
  
   
 // 制作1个原型
  
 var mayun = {
  
 mayun: 9999999999,
  
 houses: 11111,
  
 };
  
 // 参数1: 新对象的原型
  
 // 参数2: 新对象的各种属性设置
  
 var xiaoLiang = Object.create(mayun, {
  
 name: {
  
 value: "小亮",
  
 // 通过方法新增对象, 则所有配置项默认值是false
  
 writable: true, //可写
  
 enumerable: true, //可遍历
  
 },
  
 age: {
  
 value: 2,
  
 writable: true,
  
 enumerable: true,
  
 },
  
 });
  
   
 console.log(xiaoLiang);
  
   
 // create: 是简化之前的操作
  
 // 1. 创建新对象
  
 // 2. 指定原型
  
 // 3. 为新对象的每个属性 增加自定义的配置
  
 </script>
  
 </body>
  
</html>

## 函数的call方法, 可以指定this

函数的call方法, 可以在触发函数的同时替换其this指向

<!DOCTYPE html>
  
<html lang="en">
  
 <head>
  
 <meta charset="UTF-8" />
  
 <meta http-equiv="X-UA-Compatible" content="IE=edge" />
  
 <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  
 <title>Document</title>
  
 </head>
  
 <body>
  
 <script>
  
 // 函数call方法
  
 // 函数默认情况下: 函数中this指向其所在的对象!
  
 // 利用call可以强制修改 函数中this的指向!
  
   
 var name = "全局区域";
  
   
 var yanan = {
  
 name: "亚楠",
  
 intro: function () {
  
 console.log(`${this.name} 是最棒的!`);
  
 console.log("this:", this);
  
 },
  
 };
  
   
 yanan.intro(); // 亚楠
  
 // 函数是对象类型 -- 函数诞生 new Function()
  
 // 对象类型是引用类型
  
 var intro = yanan.intro;
  
 intro(); // 全局区域
  
   
 // 利用call强制替换函数 执行时 的 this
  
 console.log("利用call 替换函数的this 然后执行");
  
 intro.call(yanan);
  
   
 intro.call({ name: "亮亮" });
  
 // 伪代码: call实现了什么
  
 // 相当于: intro.this = {name:"亮亮"}
  
 // intro()
  
   
 function demo() {
  
 console.log("demo.this:", this);
  
 }
  
 // 执行时, 把demo中的this 替换成 参数1: yanan
  
 demo.call(yanan);
  
 </script>
  
 </body>
  
</html>