学会这个 ECMAScript

学习ES语法机制 —— 抓难点重点要点

开发技巧分享 —— 后续一条线

徐加哲@汤圆创作

xujiazhe.me@gmail.com

- 随着 web 的发展，Javascript 世界上最受流行的语言，也是最被误解的语言
- Javascript 只是表面上语法有点类Java，其他的和Java 和没什么关系；//雷锋和雷峰塔都是名词
- Javascript语言机制吸收了 scheme，self一些特点

Java 和 Java script 出身成长，达成理想

语言的产生和设计目标，进化环境(设计语法机制解决面临问题)，让他们走向了不同的道路，差异挺大！

- Java诞生于面向对象概念正火的时候，write once，run every where，跑在虚拟机，当时热门
- Javascript诞生于navigator浏览器里， 跑在浏览器里面，一个功能就是操作 Dom（最大的图形框架） 说要补充 java(吸引开发者)
- 他们进化的环境，要完成的任务导致他们的语言机制不一样，图形框架回调比较多，如何优雅的继续访问回调前上下文的语法机制——这就有了闭包和函数级的作用域，虽然 java 添加了闭包的特性，js 加入了 new 创建一个对象，但很不一样，在各自语言里很怪异。虽然有一样的概念名词（就像 js 的函数和 C 里的函数，都叫函数）
- java 类化 vs ES 函数化；//函数，差异挺大
- 面向对象机制实现的不同：ES 的原型继承 vs Java 的类继承
- 强弱类型；部署方式 java 字节码，js 的源码
    - python 之父语言设计大师的功力，语法的优雅，语法糖多，表达能力强
    - javascript作者编译器的功力深，语言机制整合(创新)幅度挺大

ES基础类型

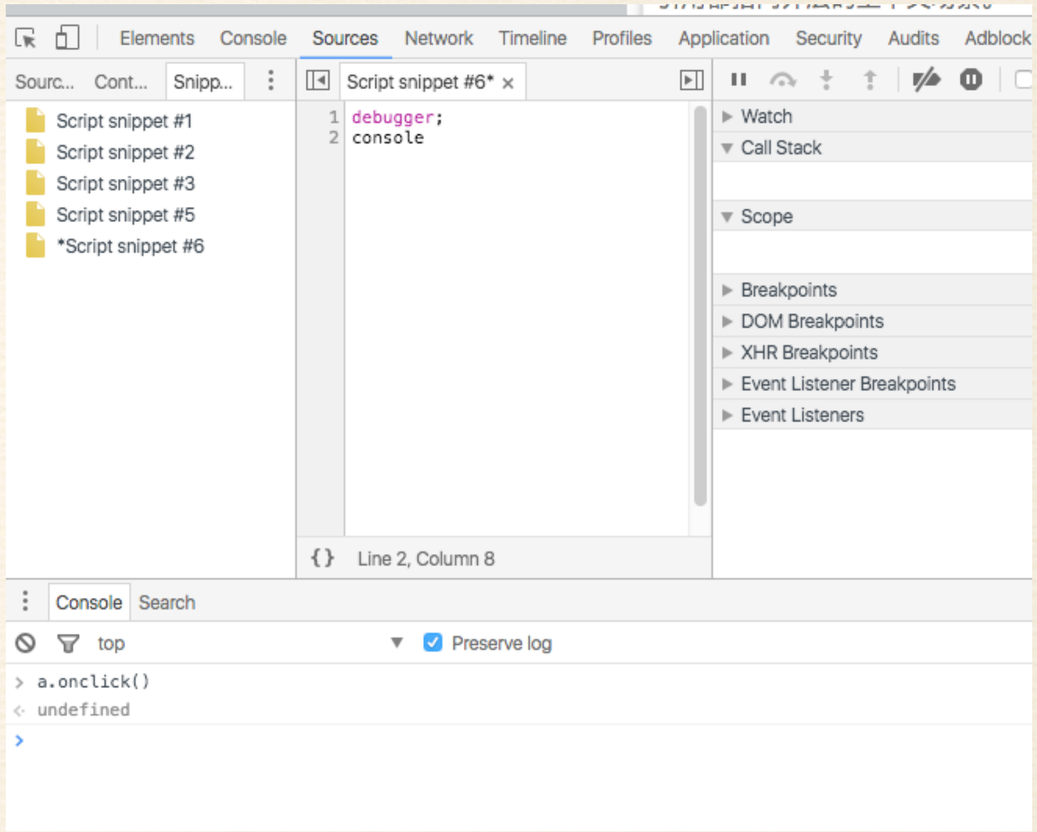
- Boolean　true false
- Null　这个类型只有一个值　null
- Undefined　undefined
- Number　64位的浮点数存储；　NaN, Infinity,
- String　"I'm a string", 'another string', 's', 'string'[0]; -都是字符串，字符串一旦创建，不可改变；ES没有字符

- ES 的对象是键值对集合（无序）
- typeof 返回值有 "number"、"string"、"boolean"、"undefined"、"function"、"object"。
- typeof [1,2]　typeof null　typeof /regexp\d+/　返回值是 object

开发实验

- chrome devtool 浏览器开发工具
  - console 直接输入语句
  - debugger和调试栏
  - snippet code cmd+enter运行

- webstorm 前端神器
  - 安装 node，js代码解释器



ES几个语法

- `===` 和 `==` 区别； false, null, undefined, 0, NaN
- js的 `==` 不同类型比较是否相等的时候，有一套转换规则，很复杂，不用
- /可能产生浮点数，即使两个都是整数；`&&` `||` 运算规则（选择计算哪个表达式作为返回值）
- 引用传递

JS Literals

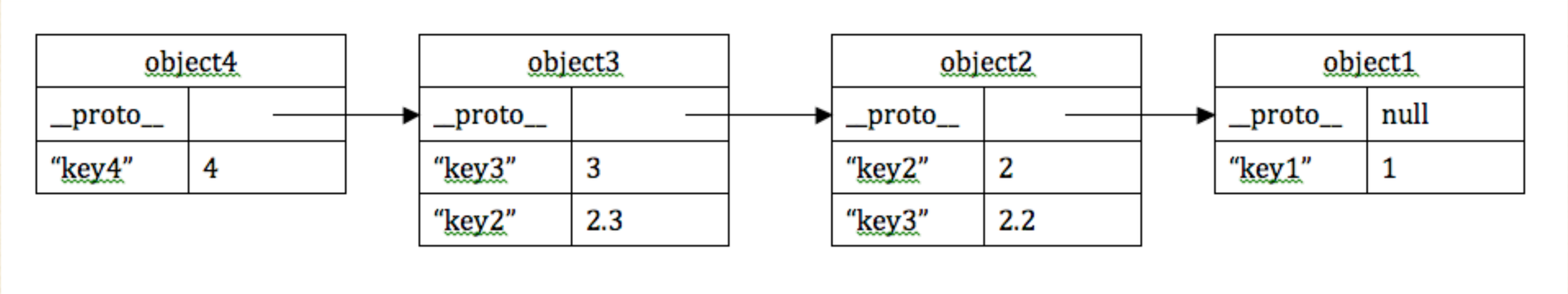
```
var array = [];
var array = new Array();

var regexp = /12[0-9]/ig;
var regexp = new Regexp('12[0-9]', 'ig');

var test_func = function(a, b){
  return a + b;
}
var test_func = new Function('a', 'b', 'return a + b');
```

JS Literals

```
//创建对象 键值对无序集合；JSON 数据交换表示法//没函数
var one_object = {
    "helo": 'word',
    ass: 1,
    another_object: {
        "can't be": "这个key 就得被引号括住"
    },
    fun_name: function(){}
};

var one_object = new Object({"helo": 'word', ass: 1});
one_object.another_object = {"can't be": "这个key 就得被引号括住"};
one_object.fun_name = new Function(......);
//访问使用
one_object.ass = 2;
one_object.another_object["can't be"] = "good";
var value = one_object.zero && one_object.zero.one_key
//得到一个undefined,访问undefined的属性抛异常TypeError
```

访问对象继承属性

- 一个对象可以看成一块内存、或者一个哈希表
- js 是对象继承的，也就是所谓的原型继承. 原型链是单链
- 访问一个对象(一块内存)的属性 例 object4.key1，找不到顺着原型链往上找



object4.__proto__ 是一个对象 object4.__proto__.name 找不到一直到 object4.__proto__.__proto__ 为 null 为止，找不到返回 undefined !

```
object4.key1 === 1; object4.key2 === 2.3;
delete object3.key2;  object4.key2 === 2;
```

JS 函数调用

```
function add(a, b, c){ //add 是个函数对象,函数对象有一些方法
    this.sum += a+b;
    typeof c === undefined;
    return this.sum;
}
var test = { sum: 10,    add: add },  test2 = {sum:2};
var array = [3, 4];
```

函数被调用的时候 会有两个附加参数 **this, arguments;** 四种调用方式

- 1 直接调用 add(1,2); //window 返回 NaN
- 2 test.add(2, 3, 4); //test.sum === 15 超级延迟绑定
- 3 apply 调用, 指定this对象; add.apply(test2, array);
- 4 构造器调用 也就是new调用

JS new 构造函数

- 函数对象有调用属性
- 我们在 C++ 里面 new 一个对象的时候，分一块内存a(包括父类内存)，这个对象绑定到那个类的代码， 调用那个构造函数(有继承先Base构造)把那块内存初始化成一个对象
- 我们在 js 里面调用 new 对象的时候，分一块内存b，把那对象的原型链__proto__变量设置到那个构造函数的原型的值**prototype(**指向了一块要被继承的内存/对象**)**；调用那个构造函数代码，初始化那块内存b； 如果那个构造函数返回值是对象，就返回那个对象，否则返回那块内存b
- 略类似Java纯静态继承

```
var Catamount = function(name){
    this.name = name;
    this.type = "猫科动物";
};
Catamount.prototype.get_name = function(){
    return this.name;
};
Catamount.prototype.hunting = function(){
    console.log('hunting food');
    var get_it = true;
    return get_it;
};

var babby_cat = new Catamount('KPI');
babby_cat.hunting();
```

var *babby_cat* = new ***Catamount***('KPI');

babby_cat.hunting();

var ***Catamount*** = function(*name*){

this.name = *name*;

this.type = "猫科动物";

}; //还有原型上两个方法的 代码

baby_cat	
__proto__	
"name"	"KPI"
"type"	"猫科动物"
constructor	**BosiCat**

Catamount	
__proto__	Function.prototype
prototype	
this.name = *name*;	
this.type = "猫科动物";	

Function	
prototype	
__proto__	**Function**.prototype
this.prototype={	
constructor: this};	

Object	
prototype	
__proto__	**Function**.prototype

Catamount.prototype	
__proto__	
constructor	*Catamount*
get_name	.....
hunting	.....

Function.prototype	
__proto__	Object.prototype
constructor	**Function**
bind	.....
apply	.....

Object.prototype	
__proto__	null
constructor	Object
toString	

bosi	
__proto__	
"helo"	"word"

BosiCat.prototype	
constructor	BosiCat
__proto__	

Array	
prototype	
__proto__	**Function**.prototype

Array.prototype	
__proto__	
数组方法 s	

function **BosiCat** (){

this['helo']='word';}

BosiCat.prototype=baby_cat;

BosiCat.prototype.constructor= BosiCat;

var bosi=new **BosiCat** ();

BosiCat	
__proto__	Function.prototype
prototype	
this['helo']='word';	

灰色分割线 解释器初始化

var *Catamount* = function(*name*){...}

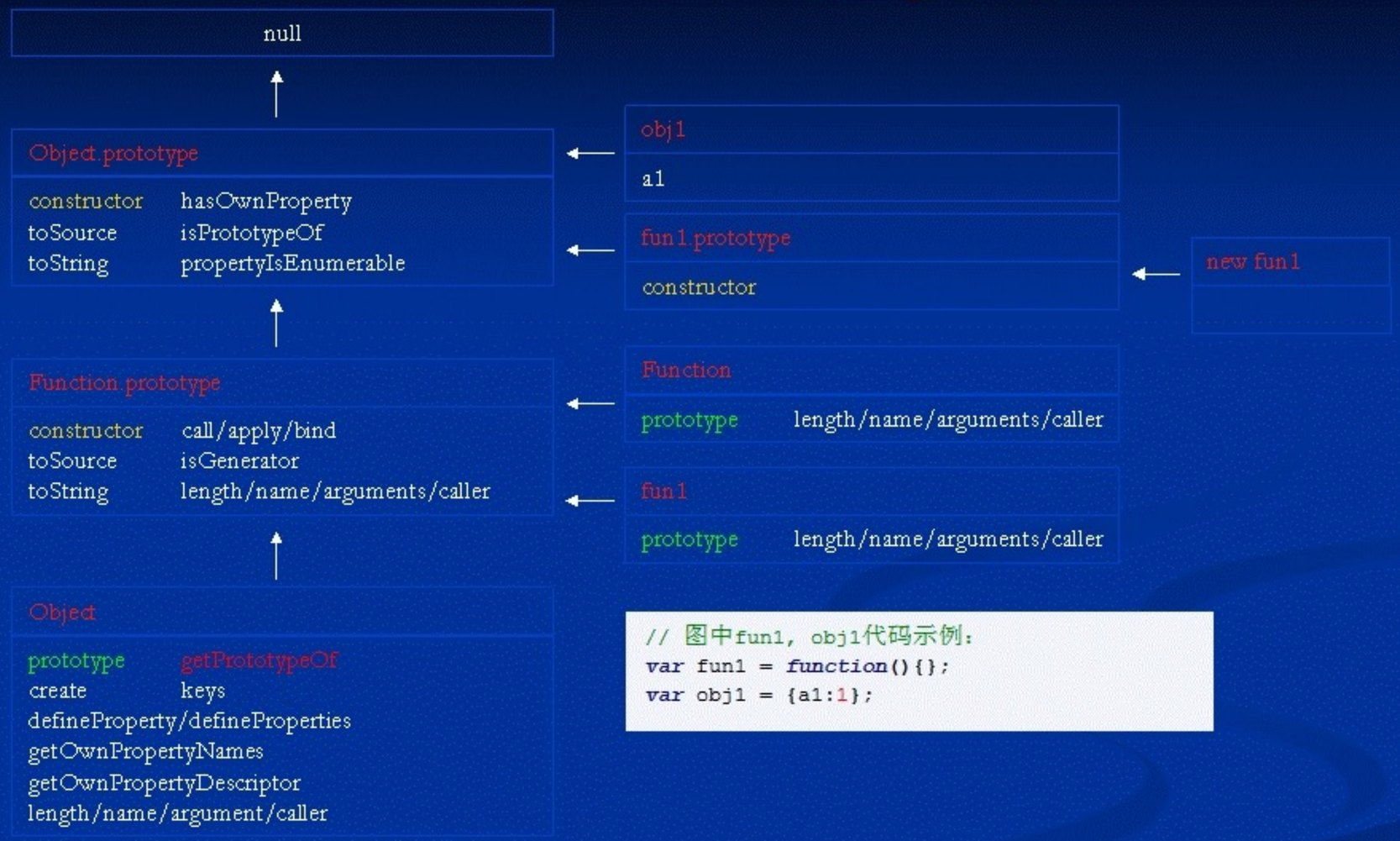
1 分一块内存

2 原型链指向 Function 原型

3 调用 this.constructor 也就是调用了 Function 的代码，操作 this(注意 this 绑定)

anony 是一个特殊的构造，构造了 Function，Object

Javascript 原型链: __proto__




```
// 图中fun1, obj1代码示例:
var fun1 = function(){};
var obj1 = {a1:1};
```

图中显示三者关系:

prototype only function has prototype, 指向一块内存，被new的实例继承

constructor only prototype has constructor, 指向创建它的function

__proto__ 箭头所示形成的chain

除Function.prototype类型为"function", 剩余所有prototype的类型均为"object".

51CTO.com 技术博客 Blog

来自网络的原型图

函数域属性

- 函数对象当参数，返回值,函数对象有一个调用属性,一个函数函数对象也有函数成员
- 一个函数运行一次就生成一个域对象，这个域 有函数有该次运行生成的所有变量
- 一个函数对象b 在一个函数a内代码运行生成的时候，在这个函数对象b上有一个[[scope]]属性，指向生成这个函数a 执行生成的那个域对象，当执行函数b内代码 本函数域 找不到的名字 UnfoundName，就本函数对象b [[scope]]属性指向的那个域对象找 名字UnfoundName，如果再找不到，就往上一个生成函数对象a的那个域对象里找
- scope chains 域链讲解；本函数内提前声明就是产生模板域 声明提前讲解

闭包、作用域

```
function makeAdder(x) {
    return function A(y) {
        debugger;
        return x + y;
    };
}
var add5 = makeAdder(5);
var add10 = makeAdder(10);
console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

```
▶ Watch
▼ Call Stack
A
(anonymous function)
▼ Scope
▼ Local
  ▶ this: Window
    y: 2
▼ Closure (makeAdder)
    x: 5
▼ Global
    Infinity: Infinity
  ▶ $: function (e,t)
```

```
Console  Search
top
> console.dir(add5)
  ▼ function A(y)
      arguments: null
      caller: null
      length: 1
      name: "A"
    ▼ prototype: Object
      ▶ constructor: function A(y)
      ▶ __proto__: Object
    ▶ __proto__: function ()
    ▼ <function scope>
      ▼ Closure
          x: 5
      ▶ Global: Window
<· undefined
```

在 chrome devtool 里面运行，看左边的 Closure 闭包

console.dir 列出一个对象的可访问属性??

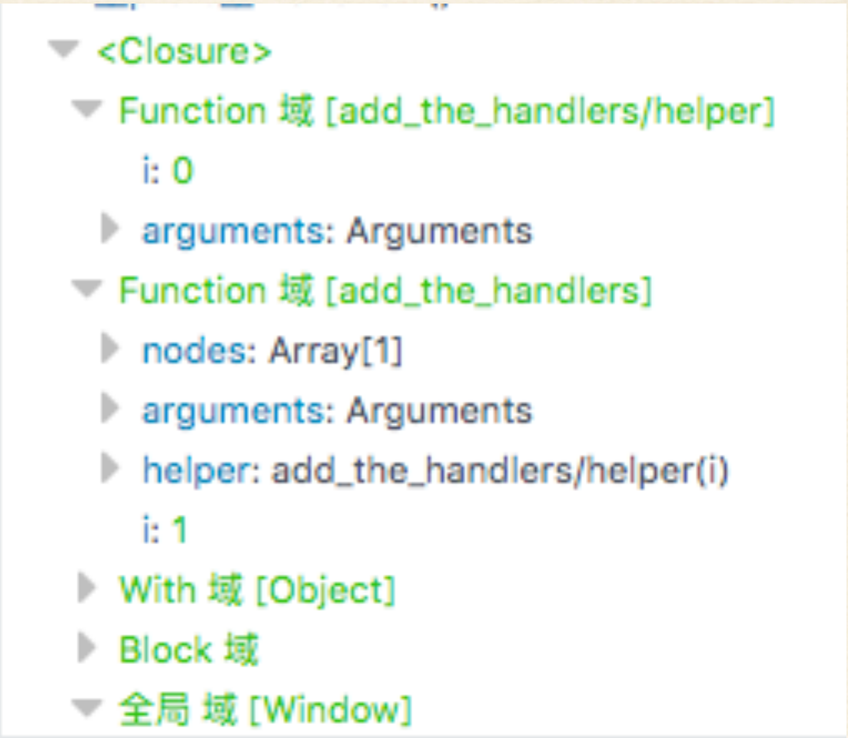
一些节点注册回调函数

```
var add_the_handlers = function(nodes){
    for(var i = -1;nodes[++i];){
        nodes[i].onclick = function(e){
            alert(i);
        };
    }
};
```

当这些节点点击回调的时候，alert 出来的都是一样的值　nodes 为3 那么 for 循环生成了3个函数对象，指向了同一个域对象 这个与对象里面有那次运行生成的值(i, nodes, this, arguments)

因为这些 for 循环生成的这些函数查找 i 都是在 add_the_handlers那次运行生成的域对象 里面查找的，那个域的 i for跑完就=== nodes.length了

```
var add_the_handlers = function(nodes) {
    var helper = function(i) {
        return function fa(e) {
            debugger;
            alert(i);alert(a);
        };
    };
    for (var i = -1; nodes[++i]; ) {
        nodes[i].onclick = helper(i);
    }
};
var a = [{}];
add_the_handlers(a);
a[0].onclick();
```



a[0].onclick 指向函数对象的 scope 指向 helper(0)运行那次的域，那个域有形参 i，helper函数对象的 scope 指向 add_the_handers 那个域有 i和 nodes；

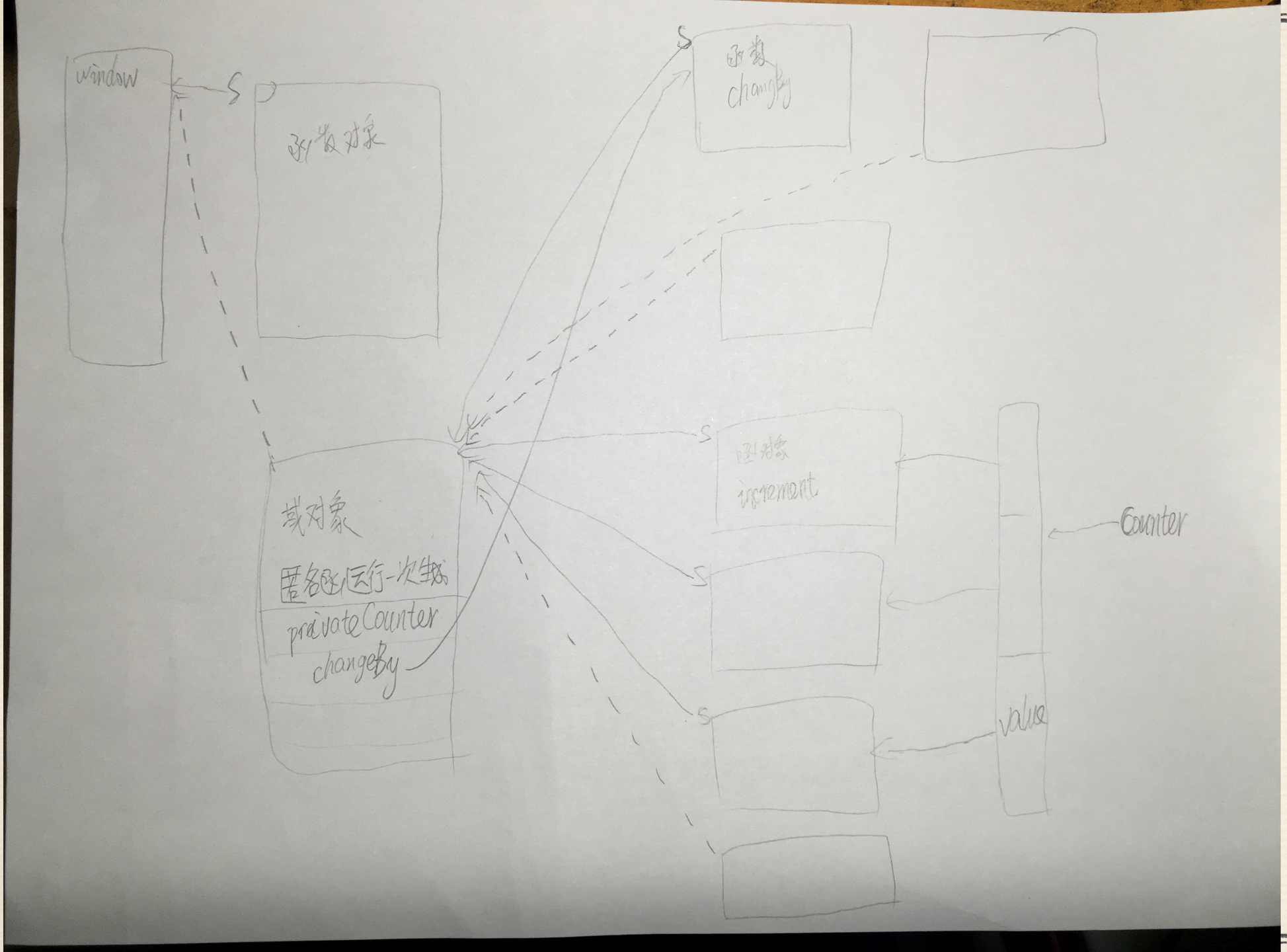
a[0].onclick 的 scope chains 就是。。看 firefox debugger 的截图

闭包、作用域

```
var Counter = (function () {
    var privateCounter = 0;
    function changeBy(val) {
        debugger;
        privateCounter += val;
    }
    return {
        increment: function () {
            changeBy(1);
        }, decrement: function () {
            changeBy(-1);
        }, value: function () {
            return privateCounter;
        }, that: this
    }
})();
Counter.increment();
console.log(Counter.value());
```

(function(a){alert(a);})(1);
匿名函数直接调用

increment、decrement、value、changeBy 指向的函数对象域链是一样的



上次运行的对象(函数对象，域对象链，对象）图

函数对象[[scope]]指向实线
域对象继承指向 虚线

闭包、作用域

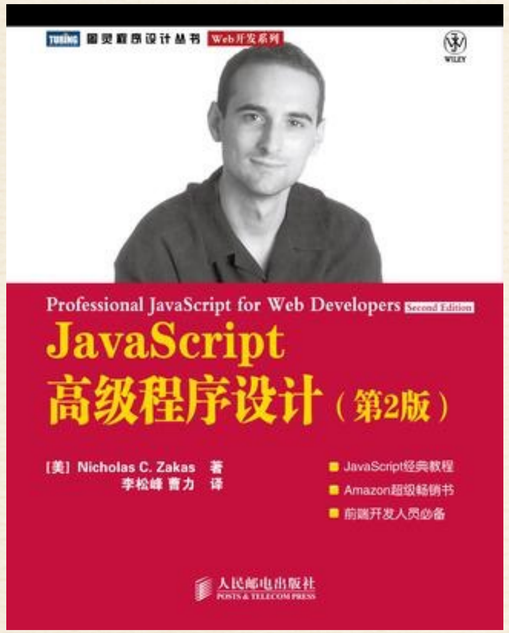
```
String.prototype.deentityify = function() {
//字符实体表，它映射字符实体的名字到对应的字符
    var entity = {
        quot: '"',
        lt: '<',
        gt: '>'
    };
    //返回 deentityify 方法.
    return function () {
//这才是 deentityify 方法，它调用字符串的 replace 方法
//查找'&'开头和';'结束的子字符串。如果这些字符可以在字符串 实体表中找到。
//那么就将该字符实体 替换为映射表中的值。它用到了一个正则表达式
        return this.replace(/&([^&;]+);/g,
            function (sub_str, match_group){
                var r = entity[match_group];
                return typeof r === 'string' ? r : sub_str;
            }
        )
    }
}();

console.log('&lt;&quot;&gt;'.deentityify());
```

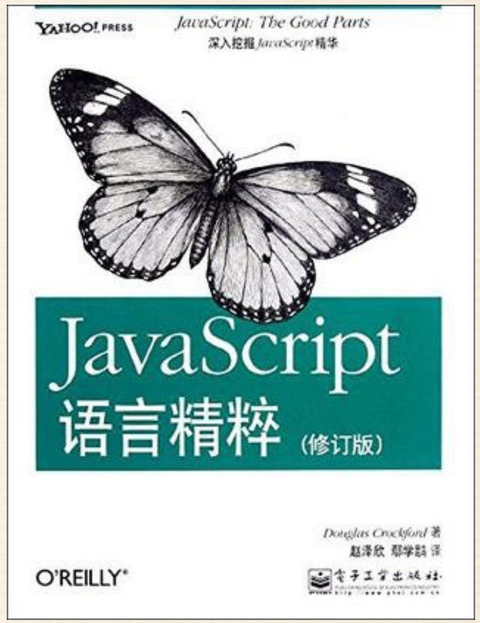
字符串，数值这些常量后面加上.运算符时候会被js包装包装一下，数值会被 Integer 包装，字符串会被 String 包装，然后可以调用 String 的一些方法

JavaScript 快速测试

```javascript
//-------题目分割线----------
var F = function () { };
Object.prototype.a = function () {};
Function.prototype.b = function () {};
var f = new F(); //f.a ?? f.b ??
//-------题目分割线----------
var foo = 1;
function v() {
    console.log(foo);
    var foo = 2;
    console.log(foo);
};
v();
//-------题目分割线----------
function printing() {
    console.log(1);
    setTimeout(function () {
        console.log(2);
    }, 1000);
    setTimeout(function () {
        console.log(3);
    }, 0);
    console.log(4);
}
printing();  console.log(5);
//-------题目分割线----------
function A() {
    this.x = 1;
    return {x: 100};
}
var a = new A;
console.log(a);
//-------题目分割线----------
var o = {
    age: 22,
    getAge: function () {
        return this.age;
    }
};
var p = o.getAge;
p();
//-------题目分割线----------
function setName(obj) {
    obj.name = "Tom";
    obj = new Object();
    obj.name = "Mike";
};
var person = new Object();
setName(person);
console.log(person.name);
//-------题目分割线----------
var name = "World";
(function () {
    if( typeof name === 'undefined'){
        var name = 'Jack';
        console.log('Goodbye ' + name);
    }else {
        console.log('Hello ' + name);
    }
})();
//-------题目分割线----------
function showCase(value) {
    switch(value){
        case 'A':
            console.log('Case A');
            break;
        case 'B':
            console.log('Case B');
            break;
        case undefined:
            console.log('undefined');
            break;
        default:
            console.log('Do not know!');
    }
}
showCase(new String('A'));
showCase(String('A'));
//-------题目分割线----------
```

以后长期用 js，要踩的坑多着呢





JavaScript高级编程和犀牛书<JavaScipt 权威指南>
包括不少前端其他的东东

js 语言的子集，只讲解 js 语言本身作者认为好的部分

程序员代码面试指南
JavaScript 设计模式与开发实践
编程珠玑
Node.js实战
你不知道的JavaScript（上卷）
JavaScript 权威指南
jQuery技术内幕
数据结构（C语言版）
锋利的jQuery（第2版）
Android 4编程入门经典
深入浅出node.js
算法设计与分析
JavaScript编程实战
JavaScript DOM编程艺术（第2版）
编写高质量代码 Web前端开发修炼之道
jQuery UI开发指南
JavaScript框架设计
编写可测试的JavaScript代码
JavaScript 忍者秘籍
JavaScript语言精粹（修订版）
HTML5 Canvas核心技术
学用JavaScript数据结构与算法
JavaScript 高级程序设计（第3版）
高性能JavaScript

前端书籍

图片来自网络