# arguments

具备数组的特性，但不是Array数组

|  |
| --- |
| Array.prototype.m = function(){  console.log("Add method m to Array"); } var arr = []; arr.m(); //Add method m to Array  function test(){  //arguments.m(); //arguments.m is not a function  var arr = Array.prototype.slice.call(arguments); //转化为Array对象  arr.m();  } |

检测对象是否为Array 常用方式：

|  |
| --- |
| Array.isArray(obj);  obj instanceof Array; |

# caller

返回一个对函数的引用，该函数调用了当前函数。

functionName.caller

functionName 对象是所执行函数的名称。

说明:对于函数来说，caller 属性只有在函数执行时才有定义。 如果函数是由 Javascript 程序的顶层调用的，那么 caller 包含的就是 null 。

# callee

正被执行的 Function 对象

|  |
| --- |
| (**function**(a,b,c){  ***console***.**log**(arguments.**callee**.**caller**);  **if**(arguments.**length** != arguments.**callee**.**length**) //传递的参数和形参的个数是否相等  arguments.**callee**(1,2,3); })(1,2); |

# apply 与 call

把一个对象的方法放到另一个对象上执行,apply是数组参数，call是可变参数

自己理解：

f.call(obj) ; 就相当于在obj对象里添x加了f的定义，此时f中的this指代obj

f1.call(f2); 就相当于在f2中添加了f1的定义。

call/apply用来改变函数的执行上下文（this）

有两种情况需要注意，传null或undefined时，将是JS执行环境的全局变量。浏览器中是window，其它环境（如node）则是global。

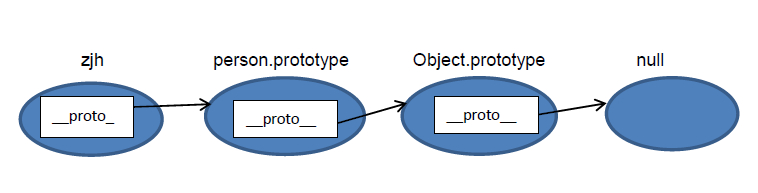
|  |
| --- |
| function base(){  this.name = "base property";  this.args = arguments.length;  this.show = function(){  console.log("I am base function!");  } } function extend(){  base.apply(this,[2,3,4]);  //base.call(this,2,3,4);  console.log("base function params count is:"+this.args);  this.show(); } |

# prototype 与 \_\_proto\_\_

**原型链定义：**

function person(){}

var zjh = new person()



**prototype 和 \_\_proto\_\_理论基础：**

|  |
| --- |
| 1. 原型链的形成是真正是靠\_\_proto\_\_ 而非prototype,当执行对象的一个方法时，先查找自身是否存在该方法，如果不存在，会在原型链上查找，但不会查找自身的prototype。 2. 一个对象的\_\_proto\_\_记录着自己的原型链，决定了自身的数据类型，改变\_\_proto\_\_就等于改变对象的数据类型。 3. 利用构造函数创建对象时，函数的prototype赋值给了新对象的\_\_proto\_\_，决定了新对象的数据类型。 4. 函数的prototype不属于自身的原型链，它是子类创建的核心，决定了子类的数据类型，是连接子类原型链的桥梁。 5. 在原型对象上定义方法和属性的目的是为了被子类继承和使用。 6. 在构造函数里通过this定义的方法和变量，拥有这些方法和变量的不是函数本身，而是构造函数的调用者。 |

**案例1：**

|  |
| --- |
| var animal = function(){};   var dog = function(){};   animal.price = 2000;//   dog.prototype = animal;   var tidy = new dog();   console.log(dog.price) //undefined   console.log(tidy.price) // 2000 |

**案例2：**

|  |
| --- |
| function p(){  this.prop1 = "prop1";  this.func1 = function(){console.log("func1")} } p.prop2 = "prop2"; p.func2 = function(){console.log("func2")}  p.prototype.prop3 = "prop3"; p.prototype.func3 = function(){console.log("func3")}  console.log(p.prop2); /// prop2  function c(){} c.\_\_proto\_\_ = p.prototype; console.log(c.prop3); /// prop3  var o = new p();  {  var obj = {} //创建一个空对象  obj.\_\_proto\_\_ = p.prototype; //把p的prototype赋值给obj的\_\_proto\_\_  //将p的作用域赋给obj，此时this指向obj，并给obj赋值了成员变量prop1  p.call(obj);// 相当于var obj = {this.prop1 = ‘prop1’;..}  return obj; //返回新对象  } console.log(o.prop1); //prop1 console.log(o.prop3); //prop3 |

# hoisting

1.变量提升

就是把变量提升提到函数的top的地方。只是提升变量的声明，并不会把赋值也提升上来

|  |
| --- |
| var v='Hello World';  (function(){  var v;  alert(v);  v='I love you';  })() |

2.函数提升

函数提升是把整个函数都提到前面去。

在我们写js code 的时候，我们有两种写法，一种是函数表达式，另外一种是函数声明方式。我们需要重点注意的是，**只有函数声明形式才能被提升。**

|  |
| --- |
| function myTest(){  // foo();  function foo(){  //函数声明方式，提升 alert("我来自 foo");  } }  myTest();  function myTest(){  foo();  var foo =function foo(){  //函数表达式,不会提升 alert("我来自 foo");  } }  myTest(); |

# scope

**javascript中一个名字(name)以四种方式进入作用域(scope)，其优先级顺序如下：**  
1、语言内置：所有的作用域中都有 this 和 arguments 关键字  
2、形式参数：函数的参数在函数作用域中都是有效的  
3、函数声明：形如function foo() {}  
4、变量声明：形如var bar;

(1)函数作用域

javascript没有块级作用域，而是函数作用域.

所谓函数作用域就是说：变量在声明它们的函数体以及这个函数体嵌套的任意函数体内都是有定义的。

|  |
| --- |
| function t(flag){  if(flag){  var s="ifscope";  for(var i=0;i<2;i++)  ;  }  console.log(i);  console.log(s);  }  t(true);// 2 ifscope |

(2)变量作用域

Js中没有用var声明的变量都是全局变量，而且是顶层对象的属性。

|  |
| --- |
| function t(flag){  if(flag){  s="ifscope";  for(var i=0;i<2;i++)  ;  }  console.log(i);  console.log(s);  }  t(true); // ifscope |

当使用var声明一个变量时，创建的这个属性是不可配置的，也就是说无法通过delete运算符删除

|  |
| --- |
| var name=1    ->不可删除  sex=”girl“         ->可删除  this.age=22    ->可删除  (function(x){  delete x;  return x;  })(1); //不可删除 |

(3)作用域链(scope chain)

|  |
| --- |
| name="lwy"; function t(){  var name="tlwy";  function s(){  var name="slwy";  console.log(name);  }  function ss(){  console.log(name);  }  s();//slwy  ss();//tlwy } |

当执行s时，将创建函数s的执行环境(调用对象),并将该对象置于链表开头，然后将函数t的调用对象链接在之后，最后是全局对象。即：s()->t()->window

然后从链表开头寻找变量name,很明显name是"slwy"。

但执行ss()时，作用域链是： ss()->t()->window,所以name是”tlwy"

1. with语句

with语句主要用来临时扩展作用域链，将语句中的对象添加到作用域的头部。

|  |
| --- |
| person={name:"yhb",age:22,height:175,wife:{name:"lwy",age:21}}; with(person.wife){  console.log(name); } |

with语句将person.wife添加到当前作用域链的头部，所以输出的就是：“lwy".

with语句结束后，作用域链恢复正常。

|  |
| --- |
| var global\_scope = "global"; var fun = function(){  var fun\_scope = "fun scope";  return function(){  var inner\_func\_scope = "inner fun scope";  return global\_scope + "\n" + fun\_scope + "\n" + inner\_func\_scope;  }; }; var fp = fun();  fp() //scope chain: fp->fun->window console.log(fp()); |

# this

this关键字总是指向当前函数的所有者对象(执行空间)

this 总是在运行时才能确定其具体的指向, 也才能知道它的调用对象

作用域: 它指明的是一个数值或者表达式所关联的上下文(能够被引用的执行空间).

如果在global的scope来调用,则指向的是bowser的顶级对象window 例如: get\_name()

|  |
| --- |
| var name = "The Window"; var object = {  name : "My Object",  getNameFunc : function(){  return function(){  return this.name;  };  } };  var f = object.getNameFunc();  f()//scope chain: f->getNameFunc->window  // this.name 指向当前函数的所有者对象(运行时) window  因此 this.name == window.name  alert(f()); //The Window  var name = "The Window"; var object = {  name : "My Object",  getNameFunc : function(){  var that = this;  return function(){  return that.name;  };  } };  var f1 = object.getNameFunc();  f1();//My Object  // scope chain: f1-> getNameFunc -> window  // this: window  // that 在scope chain中查找that,that在getNameFunc中，getNameFunc中的this指代的是object,that= this,那that就指代object对象，故that.name 就是object.name |
| scope和this总结：  （1）this:当前函数的所有者（对象），即调用该函数的对象，this只跟原型有关系。  （2）scope:当前执行表达式的上下文  （3）查找变量：如果是非this变量， 先确定当前函数的scope chain,然后按照顺序查找变量;如果是this变量，则要确定当前scope的调用者。  (4)一个函数里定义了this变量（this.xxx=xxx），该函数不拥有这个属性，它的作用是当该函数执行时，this变量会被赋值给其调用者本身，此时调用者就拥有了该属性。  function f(){  this.props = “hello”;  }  f.props // undefined;  window.f();  window.props; //hello  或者：  var obj = {};  f.call(obj);  obj.props; //hello |

# == 、===、!= 、!==

**1、对于string,number等基础类型，==和===、!=和!==是有区别的**

1）不同类型间比较，==之比较“转化成同一类型后的值”看“值”是否相等，===如果类型不同，其结果就是不等

2）仅当值不相等时，!=成立；值或类型不相等时，则!==成立。

3）同类型比较，直接进行“值”比较，两者结果一样

**2、对于Array,Object等高级类型，==和===、!=和!==是没有区别的**

进行“指针地址”比较

**3、基础类型与高级类型，==和===、!=和!==是有区别的**

1）对于==和!=，将高级转化为基础类型，进行“值”比较

2）因为类型不同，===结果为false，!== 结果为true

# NaN、null 及 undefined

Jscript有三种主要数据类型、两种复合数据类型和两种特殊数据类型。

主要（基本）数据类型是：字符串 数值 布尔

复合（引用）数据类型是：对象 数组

特殊数据类型是：null undefined

(1)NaN

如果把 NaN 与任何值（包括其自身）相比得到的结果均是 false，所以要判断某个值是否是 NaN，不能使用 == 或 === 运算符,要使用isNaN() 。isNaN() 函数通常用于检测 parseFloat() 和 parseInt() 的结果，以判断它们表示的是否是合法的数字。当然也可以用 isNaN() 函数来检测算数错误，比如用 0 作除数的情况。

(2)null

指出一个变量中没有包含有效的数据

产生 null 的原因是：对一个变量显式地赋值为 null。 包含 null 的表达式之间的任何操作。

在 Jscript中，数据类型 null 只有一个值：null。关键字 null 不能用作函数或变量的名称。包含 null 的变量包含“无值”或“无对象”。换句话说，该变量没有保存有效的数、字符串、boolean、数组或对象。可以通过给一个变量赋 null 值来清除变量的内容。

请注意，在 Jscript 中，null 与 0 不相等（与在 C 和 C++ 中不同）。同时应该指出的是，Jscript中 typeof 运算符将报告 null 值为 Object 类型，而非类型 null。这点潜在的混淆是为了向下兼容。

(3) undefined (undefined)

对象属性或方法不存在，或声明了变量但从未赋值。

即当你使用了对象未定的属性或者未定义的方法时或当你声明一个变量，但你确从未对其进行赋值，便对其进行操作(当然赋值除外)，会有"undefined"提示

|  |
| --- |
| var a1; var a2 = null; var a3 = NaN;  typeof a1; // undefined typeof a2; // object typeof a3; // number  a1 == a2; //true a1 == a3; //false a2 == a3; //false  isNaN(a3);//true  a1 == 0; // false  a2 == 0; // false  a3 == 0;// false  a1 == false //false  a2 == false //false  a3 == false // false  a1+3; //NaN a2+3; //3 a3+3; //NaN  a1+a2;// NaN; a1+a3;// NaN; a2+a3;// NaN; |

# typeof 与 instanceof

typeof返回一个表达式的数据类型的字符串，返回结果为js基本的数据类型，包括number,boolean,string,object,undefined,function.语法为typeof(**data**) 或 typeof **data**

|  |
| --- |
| 1. typeof(1);//            number 2. typeof("1");//            string 3. typeof(true);//            boolean 4. typeof(a);//            undefined(if a not be defined) 5. typeof([]);//            object 6. typeof(function(){});// function 7. typeof(new Date());//    object 8. typeof({});//    object |

可以看出typeof这个函数，可以很容易的分清楚基础类型(number，string，boolean，undefined，function)，但他无法分辨object类型的具体类型，如Array，Date，以及一些自定义类。

instanceof则为判断一个对象是否为某一数据类型，或一个变量是否为一个对象的实例;返回boolean类型  
语法为 **o** instanceof **A**

|  |
| --- |
| 1. [] instanceof Array; //true 2. [] instanceof Object; //true 3. new Person() instanceof Person; //true 4. new Person() instanceof Object; //true 5. new Person() instanceof Date; //false 6. document.getElementById('log') instanceof HTMLDivElement;//true |

instanceof检测的是原型，它可以很清楚明确的判断出一个变量的类型。

判断一个变量是否被定义：typeof(a) == 'undefined';

判断一个变量类型： [] instanceof Array;

# Boolean() 和 !! 和 ||

一个值为 true 或者 false 的表达式。

如果需要，非Boolean表达式也可以被转换为 Boolean 值，但是要遵循下列规则：

所有的对象都被当作 true，当且仅当字符串为空时，该字符串被当作 false。

null 和 undefined 被当作 false。

当且仅当数字为0时，该数字被当作 false。

|  |
| --- |
| var obj = {}; or //p1 = obj.prop  var obj = {prop:null}; //p2 = obj.prop  var obj = {prop:0}; //p3 = obj.prop  var obj = {prop:''} //p4 = obj.prop  下面三个表达式是等价的：  var r = Boolean(obj.prop);  var r = obj.prop|| false;  var r = !!obj.prop;  注意：对于null,undefined,0,’’这四个值，布尔表达式仅将其当作false来处理,但这四个值不一定相等，除过p1==p2 和 p3==p4 其余的都互不相等。 |

# closure

**(1) 基础概念**

所谓“闭包”，指的是一个拥有许多变量和绑定了这些变量的环境的表达式（通常是一个函数），因而这些变量也是该表达式的一部分。

***自己理解:***

当一个内部函数被其外部函数之外的变量引用时，就形成了一个闭包。此时，其外部函数中的变量会始终保存在内存中，不会被GC回收。

作用： （1）在函数外部访问函数内部的局部变量 (2) 这些变量的值始终保存在内存中

|  |
| --- |
| function a(){  var i=0;  function b(){  alert(++i);  }  return b; } var c=a(); //调用a()返回b, b被变量c引用,此时a中的变量i会被一直保存在内存中 c(); //1 |

这段代码有两个特点：

1、函数b嵌套在函数a内部

2、函数a返回函数b

这样在执行完var c=a( )后，变量c实际上是指向了函数b，再执行c( )后就会弹出一个窗口显示i的值（第一次为1）。这段代码其实就创建了一个闭包，为什么？因为函数a外的变量c引用了函数a内的函数b，就是说：

当函数a的内部函数b被函数a外的一个变量引用的时候，就创建了一个闭包。

简而言之，闭包的作用就是在a执行完并返回后，闭包使得Javascript的垃圾回收机制[GC](http://baike.baidu.com/subview/315216/9956046.htm)不会收回a所占用的资源，因为a的内部函数b的执行需要依赖a中的变量。

如果a返回的不是函数b，情况就完全不同了。因为a执行完后，b没有被返回给a的外界，只是被a所引用，而此时a也只会被b引 用，因此函数a和b互相引用但又不被外界打扰（被外界引用），函数a和b就会被[GC](http://baike.baidu.com/subview/315216/9956046.htm)回收。

**(2) GC**

在Javascript中，如果一个对象不再被引用，那么这个对象就会被GC回收。如果两个对象互相引用，而不再被第3者所引用，那么这两个互相引用的对象也会被回收。因为函数a被b引用，b又被a外的c引用，这就是为什么函数a执行后不会被回收的原因。

1. **execution context、call object**

如果要更加深入的了解闭包以及函数a和嵌套函数b的关系，我们需要引入另外几个概念：函数的执行环境（execution context）、活动对象（call object）、作用域（scope）、作用域链（scope chain）。以函数a从定义到执行的过程为例阐述这几个概念。

1、当定义函数a的时候，js解释器会将函数a的作用域链（scope chain）设置为定义a时a所在的“环境”，如果a是一个全局函数，则scope chain中只有window对象。

2、当函数a执行的时候，a会进入相应的执行环境（execution context）。

3、在创建执行环境的过程中，首先会为a添加一个scope属性，即a的作用域，其值就为第1步中的scope chain。即a.scope=a的作用域链。

4、然后执行环境会创建一个活动对象（call object）。活动对象也是一个拥有属性的对象，但它不具有原型而且不能通过JavaScript代码直接访问。创建完活动对象后，把活动对象添加到a的作用域链的最顶端。此时a的作用域链包含了两个对象：a的活动对象和window对象。

5、下一步是在活动对象上添加一个arguments属性，它保存着调用函数a时所传递的参数。

6、最后把所有函数a的形参和内部的函数b的引用也添加到a的活动对象上。在这一步中，完成了函数b的的定义，因此如同第3步，函数b的作用域链被设置为b所被定义的环境，即a的作用域。

到此，整个函数a从定义到执行的步骤就完成了。此时a返回函数b的引用给c，又函数b的作用域链包含了对函数a的活动对象的引用，也就是说b可以访问到a中定义的所有变量和函数。函数b被c引用，函数b又依赖函数a，因此函数a在返回后不会被GC回收。

当函数b执行的时候亦会像以上步骤一样。因此，执行时b的作用域链包含了3个对象：b的活动对象、a的活动对象和window对象，如下图所示：

如图所示，当在函数b中访问一个变量的时候，搜索顺序是先搜索自身的活动对象，如果存在则返回，如果不存在将继续搜索函数a的活动对象，依次查找，直到找到为止。如果整个作用域链上都无法找到，则返回undefined。如果函数b存在prototype原型对象，则在查找完自身的活动对象 后先查找自身的原型对象，再继续查找。这就是Javascript中的变量查找机制。

闭包可以用在许多地方。它的最大用处有两个，一个是可以读取函数内部的变量，另一个就是让这些变量的值始终保持在内存中。

# function

**（1）函数定义**

函数是由这样的方式进行声明的：关键字 function、函数名、一组参数，以及置于括号中的待执行代码。

函数的构造语法有这三种：

|  |
| --- |
| **1.function语句**  function functionName(arg0, arg1, ... argN) { statements }  function f(x){return x\*x};  **2.Function()构造函数**  var function\_name = new Function(arg1, arg2, ..., argN, function\_body);  var f = new Function("x","return x\*x;");  **3. 函数直接量**  var func = function(arg0, arg1, ... argN) { statements };  var f = function(x){return x\*x;}; |

虽然函数直接量创建的是未命名函数，但是它的语法也规定它可以指定函数名，这在编写调用自身的递归函数时非常有用，例如：

|  |
| --- |
| var f = function fact(x) { //fact 仅在其作用域可见  　　if (x <= 1) {  　　 return 1;  　　} else {  　　 return x \* fact(x - 1);  　　}  }; |

注：它并没有真正创建一个名为fact()函数，只是允许函数体用这个名字来引用自身。 JavaScript1.5之前的版本中没有正确实现这种命名的函数直接量。直接量创建的函数实现递归还可以通过callee来实现：

|  |
| --- |
| var f = function fact(x) {  　　if (x <= 1) {  　　 return 1;  　　} else {  　　 return x \* arguments.callee(x - 1);  　　}  }; |

arguments.callee返回的就是正在执行函数的本身。

1. **函数声明、函数表达式、匿名函数**

函数声明：function fnName () {…};使用function关键字声明一个函数，再指定一个函数名，叫函数声明。

函数表达式 var fnName = function () {…};使用function关键字声明一个函数，但未给函数命名，最后将匿名函数赋予一个变量，叫函数表达式，这是最常见的函数表达式语法形式。

匿名函数：function () {}; 使用function关键字声明一个函数，但未给函数命名，所以叫匿名函数，**匿名函数属于函数表达式**，匿名函数有很多作用，赋予一个变量则创建函数，赋予一个事件则成为事件处理程序或创建闭包等等

函数声明和函数表达式不同之处在于，一、Javascript引擎在解析javascript代码时会‘函数声明提升’（Function declaration Hoisting）当前执行环境（作用域）上的函数声明，而函数表达式必须等到Javascirtp引擎执行到它所在行时，才会从上而下一行一行地解析函数表达式，二、函数表达式后面可以加括号立即调用该函数，函数声明不可以，只能以fnName()形式调用

|  |
| --- |
| fnName(); function fnName(){  ... } //正常，因为‘提升’了函数声明，函数调用可在函数声明之前 fnName(); var fnName=function(){  ... } //报错，变量fnName还未保存对函数的引用，函数调用必须在函数表达式之后 var fnName=function(){  alert('Hello World'); }(); //函数表达式后面加括号，当javascript引擎解析到此处时能立即调用函数 function fnName(){  alert('Hello World'); }(); //不会报错，但是javascript引擎只解析函数声明，忽略后面的括号，函数声明不会被调用 function(){  console.log('Hello World'); }(); //语法错误，虽然匿名函数属于函数表达式，但是未进行赋值操作， //所以javascript引擎将开头的function关键字当做函数声明，报错：要求需要一个函数名 |

在理解了一些函数基本概念后，回头看看( function(){…} )()和( function (){…} () )这两种立即执行函数的写法，最初我以为是一个括号包裹匿名函数，并后面加个括号立即调用函数，当时不知道为什么要加括号，后来明白，要在函数体后面加括号就能立即调用，则这个函数**必须是函数表达式，不能是函数声明**。

|  |
| --- |
| (function(a){  console.log(a); //firebug输出123,使用（）运算符 })(123);  (function(a){  console.log(a); //firebug输出1234，使用（）运算符 }(1234));  !function(a){  console.log(a); //firebug输出12345,使用！运算符 }(12345);  +function(a){  console.log(a); //firebug输出123456,使用+运算符 }(123456);  -function(a){  console.log(a); //firebug输出1234567,使用-运算符 }(1234567);  var fn=function(a){  console.log(a); //firebug输出12345678，使用=运算符 }(12345678) |

可以看到输出结果，在function前面加！、+、 -甚至是逗号等到都可以起到函数定义后立即执行的效果，而（）、！、+、-、=等运算符，**都将函数声明转换成函数表达式**，消除了javascript引擎识别函数表达式和函数声明的歧义，告诉javascript引擎这是一个函数表达式，不是函数声明，可以在后面加括号，并立即执行函数的代码。

加括号是最安全的做法，因为！、+、-等运算符还会和函数的返回值进行运算，有时造成不必要的麻烦。

不过这样的写法有什么用呢？

javascript中没用私有作用域的概念，如果在多人开发的项目上，你在全局或局部作用域中声明了一些变量，可能会被其他人不小心用同名的变量给覆盖掉，根据javascript函数作用域链的特性，可以使用这种技术可以模仿一个私有作用域，用匿名函数作为一个“容器”，“容器”内部可以访问外部的变量，而外部环境不能访问“容器”内部的变量，所以( function(){…} )()内部定义的变量不会和外部的变量发生冲突，俗称“匿名包裹器”或“命名空间”。

1. **函数的调用场景**

1 纯粹的函数调用

2 作为对象方法的调用

3 作为构造函数调用

4 apply调用

# constructor

在JavaScript中，创建对象的方式包括两种：对象字面量和使用new表达式。

**（1）对象字面量**

|  |
| --- |
| var obj = {      p:”prop”,      print:function(){          alert(this.p);      }  } |

**（2）new表达式**

考察 [[Construct]] 内部方法，先给出语言规范的描述：

1.引擎将先创建一个语言原生对象，即“{}”或“new Object”，在此我们称之为O，然后设置其内部属性标识 [[Class]] 为“Object”。接下来，得到构造器 F 的 prototype（根据后文的意思，它可能不是一个对象）。如果 F.prototype 是对象，那么将O的内部 [[Prototype]] 属性指向 F.prototype。

请注意，诸如 [[Prototype]] 等为引擎内部标识符，对我们并不可见。[[Prototype]] 正是用于给内部维护原型链，虽然在我们看来，一个对象实例无法直接回溯到其原型（然而引擎内部可以），必须通过构造器中转，即 obj.constructor.prototype。

2.接着，如果 F.prototype 不是 object，那么将 O 的内部 [[Prototype]] 属性指向“the Object prototype object”（你可以参考[这里](http://www.pushiming.com/blog/2009/09/whos-object-prototype/)）。等到 O 的 [[Prototype]] 有了自己的归属以后，引擎调用构造器 F 的 [[Call]] 内部方法，以 O 作为 this 对象，并将传入 [[Construct]] 的参数作为入口参数——如果有的话（即诸如“new Object()”最后括号内的参数）传递过去。最后，如果 [[Call]] 的返回值是对象，那么创建成功并返回此对象，否则回头重来。

根据以上内容，我们可以构造一个伪 [[Construct]] 方法来模拟此流程：

|  |
| --- |
| function MyObject(age) {  this.age = age;  return this; //可以返回this、非对象类型数据和其他对象  }  MyObject.construct = function() {  var o = {}; //创建一个空的对象  var Constructor = MyObject; //引用构造函数本身  o.\_\_proto\_\_ = Constructor.prototype; //构造函数的prototype决定了新对象的数据类型  Constructor.apply(o, arguments);  //如果构造函数MyObject无返回值，或者返回非对象类型数据，则默认返回o;  //如果构造函数MyObject返回的是对象，则会将该对象作为new的对象并返回。  return o;  };  var obj1 = new MyObject(10);  var obj2 = MyObject.construct(10);  alert(obj2 instanceof MyObject); // true |

不同于其它的主流[编程](http://www.2cto.com/kf)语言，JavaScript的构造函数并不是作为类的一个特定方法存在的；当任意一个普通函数用于创建一类对象时，它就被称作构造函数，或构造器。一个函数要作为一个真正意义上的构造函数，需要满足下列条件：

1. 在函数内部对新对象（this）的属性进行设置，通常是添加属性和方法。
2. 构造函数可以包含返回语句（不推荐），但返回值必须是this，或者其它非对象类型的值；如果是其他对象的话，构造函数会将该对象作为最终的新对象返回，导致最终new出来的对象不是预期的结果。
3. 是实现继承的必经之路。(\_\_proto\_\_)

# hasOwnProperty和 isPrototypeOf

hasOwnProperty：是用来判断一个对象是否有你给出名称的属性或对象。不过需要注意的是，此方法无法检查该对象的原型链中是否具有该属性，该属性必须是对象本身的一个成员。

isPrototypeOf是用来判断要检查其原型链的对象是否存在于指定对象实例中，是则返回true，否则返回false。

|  |
| --- |
| function siteAdmin(nickName,siteName){  this.nickName=nickName;  this.siteName=siteName; } siteAdmin.prototype.showAdmin = function() {}; siteAdmin.prototype.showSite = function(siteUrl) {  this.siteUrl=siteUrl;  return this.siteName+"的地址是"+this.siteUrl; }; var matou=new siteAdmin("愚人码头","WEB前端开发"); var matou2=new siteAdmin("愚人码头","WEB前端开发"); matou.age="30";  alert(matou.hasOwnProperty("nickName"));//true alert(matou.hasOwnProperty("age"));//true alert(matou.hasOwnProperty("showAdmin"));//false alert(matou.hasOwnProperty("siteUrl"));//false // showAdmin是siteAdmin.prototype对象的静态方法  alert(siteAdmin.prototype.hasOwnProperty("showAdmin"));//true  //siteUrl是siteAdmin.prototype.showSite的this变量，this变量在该函数执行时将其赋值给调用者本身，此时该调用者就拥有了这个变量。  // var obj = {};  // siteAdmin.prototype.showSite.call(obj,”111”);  // obj. siteUrl;//111 alert(siteAdmin.prototype.hasOwnProperty("siteUrl"));//false  // siteAdmin.prototype.showSite(111);  // siteAdmin.prototype.hasOwnProperty("siteUrl") //true alert(siteAdmin.prototype.isPrototypeOf(matou))//true alert(siteAdmin.prototype.isPrototypeOf(matou2))//true |

# delete

# ; 与 ,

# Number()、parseInt()、parseFloat()

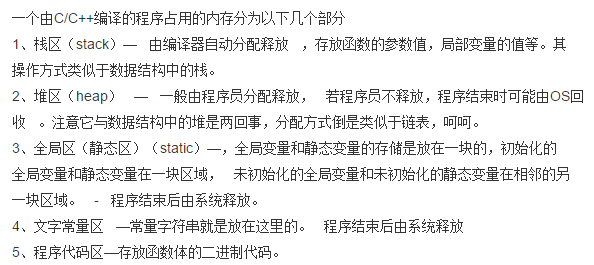
Number()的强制类型转换与parseInt()和parseFloat()方法的处理方式相似，只是它转换的是整个值，而不是部分值。parseInt()和parseFloat()方法只转换第一个无效字符之前的字符串。如“3.4.5”被转换成“3.4”，用Number()进行强制类型转换将返回NAN,如果字符串值能被完整地转换，Number()将判断是调用parseInt()还是parseFloa

# 事件委托

1. var obj = {key:value} key的值必须是一个string,如果传递的是一个object,会自动执行 object.toString()



# 作用域



# eval

<http://www.cnblogs.com/fishtreeyu/archive/2011/11/05/2237190.html>

new Function(str) 和eval具有同样的功能，都是执行一段字符串代码

在JS中将JSON的字符串解析成JSON数据格式，一般有两种方式：

1.一种为使用eval()函数。

2. 使用Function对象来进行返回解析。

# 位运算

# 数组操作 ES5 Array增强的9个API

**1、filter** 返回一个匹配过滤条件的数组。

|  |
| --- |
| var arr = ["a","b","a","c"]; var newArr = arr.filter(function(item){  return item === "a"; });  newArr -> ["a", "a"] |

**2、****forEach** 遍历数组

|  |
| --- |
| var arr = ['a','b','c','d']; arr.forEach(function(v,i,r){  console.log(v,i,r); })  ->  a 0 ["a", "b", "c", "d"]  b 1 ["a", "b", "c", "d"]  c 2 ["a", "b", "c", "d"]  d 3 ["a", "b", "c", "d"] |

**3、map** 对数组的每个元素进行操作后，返回一个新的数组

|  |
| --- |
| var arr = [  {w:10,h:10},  {w:15,h:20},  {w:12,h:12} ]; var newArr = arr.map(function(item){  item.area = item.w \* item.h;  return item; }); |

**4、reduce 实现数组元素的累加**

|  |
| --- |
| var arr = [1,2,3,4];  var result = arr.reduce(function(prev,next){  return prev + next;  });  result -> 10 |

**5、reduceRight** 从右向左实现数组元素的累加

|  |
| --- |
| var arr = ["a","b","c", "d"]; var result = arr.reduceRight(function(prev,next){  prev += "-" + next;  return prev; });  result -> d-c-b-a |

**6、some遍历一个数组，当返回true的时候停止遍历**

|  |
| --- |
| var arr = [ 1, 2, 3, 4]; arr.some( function( item, index, array ){  console.log( item, index, array);  return item > 2; });  ->  **1 0 [1, 2, 3, 4]**  **2 1 [1, 2, 3, 4]**  **3 2 [1, 2, 3, 4]** |

**7、every遍历一个数组，当返回false的时候停止遍历**

|  |
| --- |
| var arr = [ 1, 2, 3, 4]; arr.every( function( item, index, array ){  console.log( item, index, array );  return item > 2; });  -> **1 0 [1, 2, 3, 4]** |

**8、indexOf** 从数组第一个位置检索，返回第一个找到的位置，如果不存在则返回-1

|  |
| --- |
| var arr = ["a","b","a","c"]  arr.indexOf("a") //0 默认从位置0开始检索  arr.indexOf("a",1) //2 指定从位置1开始检索  arr.indexOf("f") //-1 |

**9、lastIndexOf** 从数组第一个位置检索，返回最后一个找到的位置，如果不存在则返回-1

|  |
| --- |
| var arr = ["a","b","a","c"]  arr.indexOf("a") // 2  arr.indexOf("f") //-1 |

**1、shift** 删除数组的第一个元素,返回删除的值。

|  |
| --- |
| var arr = [1,2,3,4]  arr.shift(); //1  arr -> [2, 3, 4] |

**2、unshift** 把参数加到数组的前面，返回数组的长度

|  |
| --- |
| var arr = [1,2,3,4]  arr.unshift(100,200); //6  arr >> [100, 200, 1,2, 3, 4] |

**3、pop 删除数组最后一个元素，返回数组的长度**

|  |
| --- |
| var arr = [1,2,3,4]  arr.pop();//3  arr >> [1,2,3] |

**4、push**将参数加载到数组的最后，返回数组的长度

|  |
| --- |
| var arr = [1,2,3,4]  arr.push(5);//5  arr >> [1,2,3,4,5] |

**5、concat** 将参数合并到数组中

|  |
| --- |
| var arr = [1,2,3,4]  arr.concat([100,200],1000);  arr >> [1,2,3,4,100,200,1000] |

**6、splice**从start位置开始删除count项，并从该位置起插入v1,v2,.. 并返回删除的项。

|  |
| --- |
| var arr = [1,2,3,4]  arr.splice(1,2,100,200) //从位置1处开始删除2个元素，并将100，200添加到位置1的地方  arr >> [1, 100, 200, 4] |

**7、reverse** 将数组反序，并返回新数组

|  |
| --- |
| var arr = [1,2,3,4]  arr.reverse() // [4, 3, 2, 1] |

**8、sort** 按指定的参数对数组排序

|  |
| --- |
| var arr = [4,2,1,5,3]  arr.sort() //[1, 2, 3, 4, 5] |

**9、slice**从原数组中指定开始下标到结束下标之间的项组成的新数组并返回。

|  |
| --- |
| var arr = [1,2,3,4]  arr.slice(1,3); //[2, 3],不包含结束边界位置的值 |

**10、join**将数组的元素组起一个字符串，以separator为分隔符，省略的话，则默认用逗号为分隔符

|  |
| --- |
| var arr = [1,2,3,4]  arr.join() //"1,2,3,4"  arr.join("|") //"1|2|3|4" |

**11、fill 用参数填充数组元素的每一个位置，旧的元素被覆盖。**

|  |
| --- |
| var arr = [1,2,3,4]  arr.fill("8") //["8", "8", "8", "8"] |

**1、Array.from 将一个类数组的**

|  |
| --- |
| (function(a,b,c,d){  var arr = Array.from(arguments);  arr.reverse();  return arr; })(1,2,3,4) // [4, 3, 2, 1] |

**23、Array.isArray**

|  |
| --- |
| (function(a,b,c,d){  console.log(Array.isArray(arguments));  var arr = Array.from(arguments);  arr.reverse();  console.log(Array.isArray(arr)); })(1,2,3,4) |

**24、Array.of**

|  |
| --- |
| Array.of(1,2,3,4)  [1, 2, 3, 4] |

<http://www.zhangxinxu.com/wordpress/2013/04/es5%E6%96%B0%E5%A2%9E%E6%95%B0%E7%BB%84%E6%96%B9%E6%B3%95/#some>

**https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Array/sort**