# JS深入浅出2

**属性删除**
全局变量 局部变量 全局函数 还是函数里声明局部作用域的函数，都是不可以被delete
但是隐式的创建一个变量
ohNo = 1;
window.ohNo; // 1
delete ohNo; // true

**属性检测**
propertyIsEnumerable这个方法检测对象是否可以被枚举
defineProperty设置对象属性
例如：Object.defineProperty(cat,'price',{enumerable:false,value:1000});
enumerable:true可以枚举，false不可以
var cat = new Object;
cat.legs = 4;
cat.name = "Kitty";
'legs' in cat; // true
'abc' in cat; // false
"toString" in cat; // true,
inherited property!!!

cat.hasOwnProperty('legs'); // true
cat.hasOwnProperty('toString'); // false
cat.propertyIsEnumerable('legs'); // true
cat.propertyIsEnumerable('toString'); // false

自定义变量属性为 枚举为false
**call方法:**
语法：call([thisObj[,arg1[, arg2[, [,.argN]]]]])
定义：调用一个对象的一个方法，以另一个对象替换当前对象。
说明：
call 方法可以用来代替另一个对象调用一个方法。call 方法可将一个函数的对象上下文从初始的上下文改变为由 thisObj 指定的新对象。
如果没有提供 thisObj 参数，那么 Global 对象被用作 thisObj。
bind方法:

**bind**与**call**很相似，例如，可接受的参数都分为两部分，且第一个参数都是作为执行时函数上下文中的this的对象。

不同点有两个：

①bind的返回值是函数

**apply方法：**
语法：apply([thisObj[,argArray]])
定义：应用某一对象的一个方法，用另一个对象替换当前对象。
说明：
如果 argArray 不是一个有效的数组或者不是 arguments 对象，那么将导致一个 TypeError。
如果没有提供 argArray 和 thisObj 任何一个参数，那么 Global 对象将被用作 thisObj， 并且无法被传递任何参数。

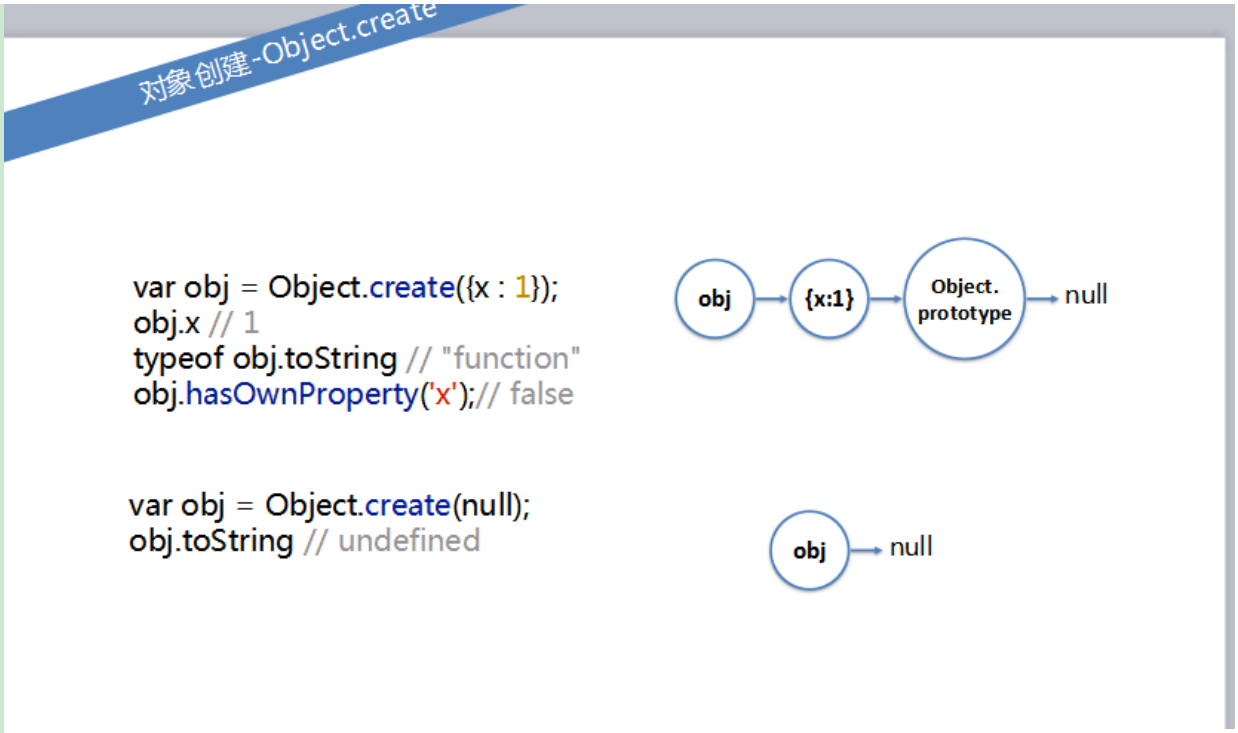
**Function.apply(obj,args)方法能接收两个参数**

obj：这个对象将代替Function类里this对象

args：这个是数组，它将作为参数传给Function（args-->arguments）

call 和 apply 都是为了改变某个函数运行时的 context 即上下文而存在的，换句话说，就是为了改变函数体内部 this 的指向。因为 JavaScript 的函数存在「定义时上下文」和「运行时上下文」以及「上下文是可以改变的」这样的概念。

# 对象创建--Object-create




自己写的变量，默认都是true，都是可以枚举
**getter setter方法**
另一种读写属性的方式
代码如下：

```
var man = {     name : 'Bosn',
 weibo : '@Bosn',
  get age() {
 return new Date().getFullYear() - 1988;
   },
 set age(val) {
    console.log('Age can\'t be set to ' + val);
 }
} console.log(man.age); // 27
 man.age = 100; // Age can't be set to 100 console.log(man.age); //
still 27
```

Object.defineProperty(o,'x',{value: 1, writable: true, configurable: true, enumerable: true});//默认的属性值都为false，value和其他的为undefined

一个属性的所有状态,包括它的数据和元数据,都存储在该属性的特性(attributes)中.属性拥有自己的特性,就像对象拥有自己的属性一样.特性的名称经常写成类似内部属性的形式(双中括号).

下面是命名数据属性拥有的特性:

[[Value]] 存储着属性的值,也就是属性的数据.
[[Writable]] 存储着一个布尔值,表明该属性的值是否可以改变.
下面是命名访问器属性拥有的特性:
属性getter/setter方法
[[Get]] 存储着getter,也就是在读取这个属性时调用的函数,该函数返回的值也就是这个属性的值.
[[Set]] 存储着setter,也就是在为这个属性赋值时调用的函数,该函数在调用时会被传入一个参数,参数的值为所赋的那个新值.
下面是两种类型的属性都有的特性:
[[Enumerable]] 存储着一个布尔值,可以让一个属性不能被枚举,在某些操作下隐藏掉自己(下面会有详细讲解).
[[Configurable]] 存储着一个布尔值,如果为false,则你不能删除这个属性,不能改变这个属性的大部分特性(除了[[Value]]),不能将一个数据属性重定义成访问 器属性,或者反之.换句话说就是:[[Configurable]]控制了一个属性的元数据的可写性.

**I=+I**
js语法的一种特殊写法，相当于把I转换成数值类型。

function foo() {}  空的foo函数
Object.defineProperty(foo.prototype, 'z',  {get : function(){return 1;}});  给foo.prototype.z创建属性z 创建一个get方法，给它返回值1
var obj = new foo();
创建一个对象，这样这个对象的原型就指向foo()
obj.z; // 1
当查看z的时候，根据原型链向上查找 发现get方法，然后返回1
 obj.z = 10;
 给z赋值 会向上找set方法赋值，如果没有就失败
 obj.z; // still 1
 仍然是1
Object.defineProperty(obj, 'z',  {value : 100, configurable: true});
实现对z的修改，添加属性
obj.z; // 100;
值为100
delete obj.z;
再删除值
obj.z; // back to 1
仍然是1

当writable=false的时候不可写

`Object.create(proto, [ propertiesObject ])`

创建一个拥有指定原型和若干个指定属性的对象。

**属性标签**
obj要操作的对象
 设置 属性 z
Object.defineProperty(obj, 'z',  {value : 100, configurable: true});
**value** 属性 为值
获得属性  ，person为对象，salary为属性
Object.getOwnPropertyDescriptor(person, 'salary');
对象创建多个属性
Object.defineProperties(person, {
  title : {value : 'fe', enumerable : true},
  corp : {value : 'BABA', enumerable : true},
  salary : {value : 50000, enumerable : true, writable : true} });
只要configurable为false 不允许修改任何值（除了writable从true修改为false总是允许的）

| 属性 | configurable:true writable:true | configurable:true writable:false | configurable:false writable:true | configurable:false writable:false |
|---|---|---|---|---|
| 修改属性的值 | √ | √* 重设value标签修改 | √ | × |
| 通过属性赋值  修改属性的值 | √ | × | √ | × |
| delete该属性返回true | √ | √ | × | × |
| 修改getter/setter方法 | √ | √ | × | × |
| 修改属性标签* (除了writable从true修改为false总是允许) | √ | √ | × | × |

**对象的标签**
prototype 标签
class标签
extensible：可扩展(对象标签)
Object.isExtensible(obj)：判断obj是否可扩展
Object.preventExtensions(obj)：取消可扩展性 (不会改变已有的属性的属性标签值)

Object.seal(obj)：设置configurable(属性标签)为false（会将所有已有的属性的configurable设置为false）
Object.isSealed();
Object.freeze(obj)：冻结方法 不管configurable 还是 writable(属性标签)为false
Object.jsFrozen();判断对象当前是否被冻结
Object.getOwnPropertyDescriptor(obj, 'x'); 查看当前标签属性
以上方法不影响该对象的原型链，需要遍历原型链逐一修改。

**序列化、其它对象方法**
**如：**

```
var obj = {x : 1, y : true, z : [1, 2, 3], nullVal : null};
 JSON.stringify(obj); // "{"x":1,"y":true,"z":[1,2,3],"nullVal":null}"
 obj = {val : undefined, a : NaN, b : Infinity, c : new Date()};
 JSON.stringify(obj); // "{"a":null,"b":null,"c":"2015-01-20T14:15:43.910Z"}"
 obj = JSON.parse('{"x" : 1}');
 obj.x; // 1
```

JSON.stringiffy(obj),js对象序列化成json格式，json格式中key是严格用双引号引起来的。
JSON.parse(jsonObject);返回js对象 将JSON变成js对象
如果属性的值是undefined的时候就不会出现，如果是NAN，infinity的话，就会转换为null，如果是时间的话UTC的时间格式
obj=JSON.parse('{"x":1}');可以让对象的x属性为1
自定义序列化

```
var obj = {     x : 1,
   y : 2,
  o : {
  o1 : 1,
 o2 : 2,
 toJSON : function () {          //
  return this.o1 + this.o2;
  }     } };
JSON.stringify(obj); // "{"x":1,"y":2,"o":3}"
```

对象方法

```
var obj = {x : 1, y : 2};
 obj.toString(); // "[object Object]"
obj.toString = function() {return this.x + this.y};
 "Result " + obj; // "Result 3", by toString
+obj; // 3, from toString
obj.valueOf = function() {return this.x + this.y + 100;};
 +obj; // 103, from valueOf "Result "
+ obj; // still "Result 3"
```

“+obj”转换为基本类型时，会找对象本身中valueOf方法，如果valueOf返回的是基本类型，就将其作为obj的值返回，不再向下查找，如果不是，就调用toString
“+obj”中的加号，JS会自动将对象转换成基本类型。ValueOf和toString都存在的情况下，会先找ValueOf中定义的结果，两个属性若都返回object，则会报错

**创建数组，数组操作，**
数组是值的有序集合，每个值叫元素，每个元素在数组中都有数字位置编号，也就是索引，**JS中的数组是弱类型的，**数组中可以**含有不同类型的元素**
**数组元素甚至可以是对象或其他数组**
数组是动态的，无须指定大小
创建数组
1）var students=[]
2）var students=new Array() //new可以省略
数组元素的增删
arr.push(); 在数组的末尾添加一个元素 /arr[arr.length]=x;
arr.unshift();在数组的第一个元素之前添加一个元素
delete arr[2]; 删除数组指定的元素，实际上是将指定的元素用 undefined 代替
arr.length-=1;删除最后一个尾部的元素
arr.pop(); 删除数组最后一个元素 / arr.length-=1;
arr.shift();在数组的头部删除一个元素
4、数组迭代
使用for in遍历，原型链上定义的值也会被输出

```
var i;
for (i in arr) {
console(arr[i]);
} //for....in 语句输出时不一定按照顺序输出
```

**二维数组**

**多维数组**

稀疏数组并不包含有从0开始的连续索引。一般length属性值比实际元素个数大。
设置为undefined也算是有的
稀疏数组的遍历得用for in
稀疏数组， 长度存在，但也许其中赋值个数不连续，稀稀拉拉的。
var arr=[,,]
0 in arr;//false
可以通过这样逗号留空的方式创建数组[,,] 也是稀疏数组。
{} => Object.prototype;
[] =>Array.prototype;
Array.prototype.join//将数组转换为字符串，可以添加间隔字符，不修改原数组
Array.prototype.reverse//将数组逆序，修改原数组
Array.prototype.sort//排序。默认按照字母顺序排序，可以添加函数进行判断，函数返回负值不换位，修改原数组
Array.prototype.concat//合并数组，不修改原数组
Array.prototype.slice//返回部分数组.负数索引表示倒数从后往前的，不修改原数组
Array.prototype.splice//数组拼接。删除从某个位置的片段同时也可以去添加元素，修改原数组
sort（ ）数字排序
遍历数组用foreach

```
arr.sort(function(a, b) {
    return a - b; }); // [3, 13, 24, 51]

arr = [{age : 25}, {age : 39}, {age : 99}];
arr.sort(function(a, b) {
    return a.age - b.age;
});
arr.forEach(function(item) {//遍历数组用foreach
    console.log('age', item.age);
});
```

7.foreach 遍历数组。接收函数做参数。三个参数。H5以上
8.map 数组映射 接收函数参数。对每个元素处理。原数组未修改。返回新数组
9、filter 数组过滤。选每个值 返回新数组。原数组不修改
10、every、some 数组判断 传入条件函数
every 是不是每个数组元素都符合某个条件 都符合才返回true
some 只要有一个数组元素符合条件就返回true
11、reduce、reduceRight
reduce 数组元素两两操作，得到唯一的值
reduceRight。从右到左顺序。功能同上
原数组不变。
12 indexOf/lastIndexOf 数组检索
indexOf
一个参数
两个参数
第二个参数为负数
lastIndexOf:从右向左找
13 isArray 判断是否为数组 H5
Array.isArray([]) //构造器上的属性
instanceof Array
({}).toString.apply([]) === '[object Array]'
[].constructor === Array;

arr.forEach(x,index,a); // 数组的遍历，简化的for in,第一个参数是数组的元素，第二个参数是数组元素的位置从0开始计算，第三个参数是数组本身
arr.map(function(x){return x + 10;}) //[11,12,13] 数组映射，对数组的转换，参数x代表数组元素。原数组不会改变
arr.filter(function(x,index){return index%3===0 || x>=8;}); // return [1,4,7,8,9,10] 数组过滤，第一个参数是数组的内容，第二个参数是数组的序列号。不会改变原数组
arr.every(function(x){return x<10;});
arr.some(function(x){return x<10;}); // 数组判断， every是每一个元素都满足条件返回true否则返回false，some是只要有一个满足条件就返回true，全部不满足返回false
arr.reduce(function(x,y){return x+y},0); //6 把每一个参数传如函数进行处理，这里是以0为初始值依次和数组元素相加每次返回值都相当于x传入下一次函数内，不会改变原函数
reduceRight功能与reduce相同，区别是从右往左遍历
arr.indexOf(a,b); //数组检索 第一个是要检索的内容元素，第二个参数是开始检索的位置，负数是从结尾开始计算的位置开始向后检索，检索结果返回检索元素第一次出现的位置，没有检索到返回-1；

**判断是否数组**

```
Array.isArray([]); //true
[] instanceof Array; //true
([]).toString.apply([]) ==='[object Array]';// true
[].constructor ===Array; //true, 判读是否是数组对象
```

**数组VS一般对象**

| 相同 | 不同点 |
| --- | --- |
| 都可以继承 数组是对象，对象不一定是数组<br>都可以当做对象添加删除属性 | 数组自动更新length 按索引访问数组常常比访问一般对象属性明显迅速。 数组对象继承Array.prototype上的大量数组操作方法 |

# 函数

函数是一块JavaScript 代码，被定义一次，单可以执行跟调用多次。js中的函数也是对象。所以js函数可以像其他对象那样操作和传递。
所以我们也常叫js中的函数为函数对象；
函数的返回值是依赖return语句的，如果没有return语句，会在所有函数语句执行完后返回一个undefined；
函数名 参数列表 函数体

**调用函数的方法**

| 直接调用 | 对象方法 | 构造器 | call/apply/bind |
| --- | --- | --- | --- |
| foo(); | o.method(); | new Foo(); | func.call(o); |

**函数声明**

```
function add (a, b) {
a = +a;
b = +b;
if (isNaN(a) || isNaN(b)) {
return;
}
return a + b;
]
```

**函数表达式**

| 函数表达式 | 立即执行函数表达式 | 返回值函数表达式 | 命名函数表达式 |
| --- | --- | --- | --- |
| // function variable<br>var add = function (a, b) {<br>// do sth<br>}; | // IEF(Immediately Executed Function)<br>(function() {<br>// do sth<br>})(); | // first-class function<br>return function() {<br>// do sth<br>}; | // NFE (Named Function Expression)<br>var add = function foo (a, b) {<br>// do sth<br>}; |

还有一种不常用的创建函数的方式
就是使用函数构造器

```
var func=new Function('a','b','console(a+b);');
var func = Function('a', 'b', 'console.log(a + b);');
func(1, 2); // 3
```

**三种对象创建方式**

| | 函数声明 | 函数表达式 | 函数构造器 |
| --- | --- | --- | --- |
| 前置 | √ | | |
| 允许匿名 | | √ | √ |
| 立即调用 | | √ | √ |
| 在定义该函数的作用域通过函数名访问 | √ | | |
| 没有函数名 | | | √ |

**THIS**

**全局的this(浏览器)**

this=window

一般函数的this(浏览器)

```
funciton f1(){
return this;
}
f1()===window;
```

## 作为对象方法的函数的this

```
var o = {
prop: 37,
f: function() {
return this.prop;
}
};
console.log(o.f()); // logs 37
var o = {prop: 37};
function independent() {
return this.prop;
}
o.f = independent;
console.log(o.f()); // logs 37
```

## 对象原型链上的this

```
var o = {f:function(){ return this.a + this.b; }};
var p = Object.create(o);
p.a = 1;
p.b = 4;
console.log(p.f()); // 5
```

## get/set方法与this

```
function modulus(){
return Math.sqrt(this.re * this.re + this.im * this.im);
}
var o = {
re: 1,
im: -1,
get phase(){
return Math.atan2(this.im, this.re);
}
};
```

```
Object.defineProperty(o, 'modulus', {
get: modulus, enumerable:true, configurable:true});
console.log(o.phase, o.modulus); // logs -0.78 1.4142
```

## 构造器中的this

```
function MyClass(){
this.a = 37;
}
var o = new MyClass();
console.log(o.a); // 37
function C2(){
this.a = 37;
return {a : 38};
}
o = new C2();
console.log(o.a); // 38
```

当使用new时 this会指向原型为Myclass().prototype的一个空对象
函数的返回值，如果没写return语句，或者return语句为基本类型时，就会将this作为返回值
但是如果return语句是一个对象的话，就会将对象作为返回值

## call/apply方法与this

```
 function add(c, d){
return this.a + this.b + c + d;
}
var o = {a:1, b:3};
add.call(o, 5, 7); // 1 + 3 + 5 + 7 = 16
add.apply(o, [10, 20]); // 1 + 3 + 10 + 20 = 34
function bar() {
console.log(Object.prototype.toString.call(this));
}
bar.call(7); // "[object Number]"
```

## bind方法与this

```
function f(){
return this.a;
}
var g = f.bind({a : "test"});
console.log(g()); // test
var o = {a : 37, f : f, g : g};
console.log(o.f(), o.g()); // 37, test
```

fun.call(obj,arg0,arg1);<br>
fun.apply(obj,[arg0,arg1]);<br>
两个函数都是函数fun请求对象obj调用，其他参数为函数实参
不同点在于call直接写实参，apply用数组。
作为对象方法的函数的this返回对象本身。
一般函数中的this返回window/
函数在严格模式下 “use strict"/return this===undefined

**这网上有博客总结了代码过程中this的调用 ，或许有点意义**

this是Javascript语言的一个关键字。

它代表函数运行时，自动生成的一个内部对象，只能在函数内部使用。

**情况一：纯粹的函数调用**
这是函数的最通常用法，属于全局性调用，因此this就代表全局对象Global。

**情况二：作为对象方法的调用**

函数还可以作为某个对象的方法调用，这时this就指这个上级对象。

**情况三 作为构造函数调用**

所谓构造函数，就是通过这个函数生成一个新对象（object）。这时，this就指这个新对象。

**情况四 apply调用**

apply()是函数对象的一个方法，它的作用是改变函数的调用对象，它的第一个参数就表示改变后的调用这个函数的对象。因此，this指的就是这第一个参数。

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Operators/thisthis详解