**数据类型**

虽然ECMAScript是可以动态转化类型的动态弱类型语言，它还是有数据类型的。也就是说，一个对象要属于一个实实在在的类型。  
标准规范里定义了9种数据类型，但只有6种是在ECMAScript程序里可以直接访问的,它们是：Undefined、Null、Boolean、String、Number、Object。

另外3种类型只能在实现级别访问（ECMAScript对象是不能使用这些类型的）并用于规范来解释一些操作行为、保存中间值。这3种类型是：Reference、List和Completion。

因此，Reference是用来解释delete、typeof、this这样的操作符，并且包含一个基对象和一个属性名称；List描述的是参数列表的行为（在new表达式和函数调用的时候）；Completion是用来解释行为break、continue、return和throw语句的。

**原始值类型**

回头来看6中用于ECMAScript程序的数据类型，前5种是原始值类型，包括Undefined、Null、Boolean、String、Number、Object。  
原始值类型例子：

var a = undefined;  
var b = null;  
var c = true;  
var d = 'test';  
var e = 10;

这些值是在底层上直接实现的，他们不是object，所以没有原型，没有构造函数。

大叔注：这些原生值和我们平时用的(Boolean、String、Number、Object)虽然名字上相似，但不是同一个东西。所以typeof(true)和typeof(Boolean)结果是不一样的，因为typeof(Boolean)的结果是function，所以函数Boolean、String、Number是有原型的（下面的读写属性章节也会提到）。

想知道数据是哪种类型用typeof是最好不过了，有个例子需要注意一下，如果用typeof来判断null的类型，结果是object，为什么呢？因为null的类型是定义为Null的。

alert(typeof null); // "object"

显示"object"原因是因为规范就是这么规定的：对于Null值的typeof字符串值返回"object“。

规范没有想象解释这个，但是Brendan Eich (JavaScript发明人)注意到null相对于undefined大多数都是用于对象出现的地方，例如设置一个对象为空引用。但是有些文档里有些气人将之归结为bug，而且将该bug放在Brendan Eich也参与讨论的bug列表里，结果就是任其自然，还是把typeof null的结果设置为object（尽管262-3的标准是定义null的类型是Null，262-5已经将标准修改为null的类型是object了）。

**Object类型**

接着，Object类型（不要和Object构造函数混淆了，现在只讨论抽象类型）是描述 ECMAScript对象的唯一一个数据类型。

Object is an unordered collection of key-value pairs.  
对象是一个包含key-value对的无序集合

对象的key值被称为属性，属性是原始值和其他对象的容器。如果属性的值是函数我们称它为方法 。

例如：

var x = { // 对象"x"有3个属性: a, b, c  
 a: 10, // 原始值  
 b: {z: 100}, // 对象"b"有一个属性z  
 c: function () { // 函数(方法)  
 alert('method x.c');  
 }  
};  
   
alert(x.a); // 10  
alert(x.b); // [object Object]  
alert(x.b.z); // 100  
x.c(); // 'method x.c'

**动态性**

ES中的对象是完全动态的。这意味着，在程序执行的时候我们可以任意地添加，修改或删除对象的属性。

例如：

var foo = {x: 10};  
   
// 添加新属性  
foo.y = 20;  
console.log(foo); // {x: 10, y: 20}  
   
// 将属性值修改为函数  
foo.x = function () {  
 console.log('foo.x');  
};  
   
foo.x(); // 'foo.x'  
   
// 删除属性  
delete foo.x;  
console.log(foo); // {y: 20}

有些属性不能被修改——（只读属性、已删除属性或不可配置的属性）。 我们将稍后在属性特性里讲解。

另外，ES5规范规定，静态对象不能扩展新的属性，并且它的属性页不能删除或者修改。他们是所谓的冻结对象，可以通过应用Object.freeze(o)方法得到。

var foo = {x: 10};  
   
// 冻结对象  
Object.freeze(foo);  
console.log(Object.isFrozen(foo)); // true  
   
// 不能修改  
foo.x = 100;  
   
// 不能扩展  
foo.y = 200;  
   
// 不能删除  
delete foo.x;  
   
console.log(foo); // {x: 10}

在ES5规范里，也使用Object.preventExtensions(o)方法防止扩展，或者使用Object.defineProperty(o)方法来定义属性：

var foo = {x : 10};  
   
Object.defineProperty(foo, "y", {  
 value: 20,  
 writable: false, // 只读  
 configurable: false // 不可配置  
});  
   
// 不能修改  
foo.y = 200;  
   
// 不能删除  
delete foo.y; // false  
   
// 防治扩展  
Object.preventExtensions(foo);  
console.log(Object.isExtensible(foo)); // false  
   
// 不能添加新属性  
foo.z = 30;  
   
console.log(foo); {x: 10, y: 20}

**内置对象、原生对象及宿主对象**

有必要需要注意的是规范还区分了这内置对象、元素对象和宿主对象。

内置对象和元素对象是被ECMAScript规范定义和实现的，两者之间的差异微不足道。所有ECMAScript实现的对象都是原生对象（其中一些是内置对象、一些在程序执行的时候创建，例如用户自定义对象）。内置对象是原生对象的一个子集、是在程序开始之前内置到ECMAScript里的（例如，parseInt, Match等）。所有的宿主对象是由宿主环境提供的，通常是浏览器，并可能包括如window、alert等。

注意，宿主对象可能是ES自身实现的，完全符合规范的语义。从这点来说，他们能称为“原生宿主”对象（尽快很理论），不过规范没有定义“原生宿主”对象的概念。

**Boolean，String和Number对象**

另外，规范也定义了一些原生的特殊包装类，这些对象是：

1. 布尔对象
2. 字符串对象
3. 数字对象

这些对象的创建，是通过相应的内置构造器创建，并且包含原生值作为其内部属性，这些对象可以转换省原始值，反之亦然。

var c = new Boolean(true);  
var d = new String('test');  
var e = new Number(10);  
   
// 转换成原始值  
// 使用不带new关键字的函数  
с = Boolean(c);  
d = String(d);  
e = Number(e);  
   
// 重新转换成对象  
с = Object(c);  
d = Object(d);  
e = Object(e);

此外，也有对象是由特殊的内置构造函数创建： Function（函数对象构造器）、Array（数组构造器） RegExp（正则表达式构造器）、Math（数学模块）、 Date（日期的构造器）等等，这些对象也是Object对象类型的值，他们彼此的区别是由内部属性管理的，我们在下面讨论这些内容。

**字面量Literal**

对于三个对象的值：对象（object）,数组（array）和正则表达式（regular expression），他们分别有简写的标示符称为：对象初始化器、数组初始化器、和正则表达式初始化器：

// 等价于new Array(1, 2, 3);  
// 或者array = new Array();  
// array[0] = 1;  
// array[1] = 2;  
// array[2] = 3;  
var array = [1, 2, 3];  
   
// 等价于  
// var object = new Object();  
// object.a = 1;  
// object.b = 2;  
// object.c = 3;  
var object = {a: 1, b: 2, c: 3};  
   
// 等价于new RegExp("^\\d+$", "g")  
var re = /^\d+$/g;

注意，如果上述三个对象进行重新赋值名称到新的类型上的话，那随后的实现语义就是按照新赋值的类型来使用，例如在当前的Rhino和老版本SpiderMonkey 1.7的实现上，会成功以new关键字的构造器来创建对象，但有些实现（当前Spider/TraceMonkey）字面量的语义在类型改变以后却不一定改变。

var getClass = Object.prototype.toString;  
   
Object = Number;  
   
var foo = new Object;  
alert([foo, getClass.call(foo)]); // 0, "[object Number]"  
   
var bar = {};  
   
// Rhino, SpiderMonkey 1.7中 - 0, "[object Number]"  
// 其它: still "[object Object]", "[object Object]"  
alert([bar, getClass.call(bar)]);  
   
// Array也是一样的效果  
Array = Number;  
   
foo = new Array;  
alert([foo, getClass.call(foo)]); // 0, "[object Number]"  
   
bar = [];  
   
// Rhino, SpiderMonkey 1.7中 - 0, "[object Number]"  
// 其它: still "", "[object Object]"  
alert([bar, getClass.call(bar)]);  
   
// 但对RegExp,字面量的语义是不被改变的。 semantics of the literal  
// isn't being changed in all tested implementations  
   
RegExp = Number;  
   
foo = new RegExp;  
alert([foo, getClass.call(foo)]); // 0, "[object Number]"  
   
bar = /(?!)/g;  
alert([bar, getClass.call(bar)]); // /(?!)/g, "[object RegExp]"

正则表达式字面量和RegExp对象

注意，下面2个例子在第三版的规范里，正则表达式的语义都是等价的，regexp字面量只在一句里存在，并且再解析阶段创建，但RegExp构造器创建的却是新对象，所以这可能会导致出一些问题，如lastIndex的值在测试的时候结果是错误的：

for (var k = 0; k < 4; k++) {  
 var re = /ecma/g;  
 alert(re.lastIndex); // 0, 4, 0, 4  
 alert(re.test("ecmascript")); // true, false, true, false  
}  
   
// 对比  
   
for (var k = 0; k < 4; k++) {  
 var re = new RegExp("ecma", "g");  
 alert(re.lastIndex); // 0, 0, 0, 0  
 alert(re.test("ecmascript")); // true, true, true, true  
}

注：不过这些问题在第5版的ES规范都已经修正了，不管是基于字面量的还是构造器的，正则都是创建新对象。

**关联数组**

各种文字静态讨论，JavaScript对象（经常是用对象初始化器{}来创建）被称为哈希表哈希表或其它简单的称谓：哈希（Ruby或Perl里的概念）， 管理数组（PHP里的概念），词典 （Python里的概念）等。

只有这样的术语，主要是因为他们的结构都是相似的，就是使用“键-值”对来存储对象，完全符合“关联数组 ”或“哈希表 ”理论定义的数据结构。 此外，哈希表抽象数据类型通常是在实现层面使用。

但是，尽管术语上来描述这个概念，但实际上这个是错误，从ECMAScript来看：ECMAScript只有一个对象以及类型以及它的子类型，这和“键-值”对存储没有什么区别，因此在这上面没有特别的概念。 因为任何对象的内部属性都可以存储为键-值”对：

var a = {x: 10};  
a['y'] = 20;  
a.z = 30;  
   
var b = new Number(1);  
b.x = 10;  
b.y = 20;  
b['z'] = 30;  
   
var c = new Function('');  
c.x = 10;  
c.y = 20;  
c['z'] = 30;  
   
// 等等，任意对象的子类型"subtype"

此外，由于在ECMAScript中对象可以是空的，所以"hash"的概念在这里也是不正确的：

Object.prototype.x = 10;  
   
var a = {}; // 创建空"hash"  
   
alert(a["x"]); // 10, 但不为空  
alert(a.toString); // function  
   
a["y"] = 20; // 添加新的键值对到 "hash"  
alert(a["y"]); // 20  
   
Object.prototype.y = 20; // 添加原型属性  
   
delete a["y"]; // 删除  
alert(a["y"]); // 但这里key和value依然有值 – 20

请注意， ES5标准可以让我们创建没原型的对象（使用Object.create(null)方法实现）对，从这个角度来说，这样的对象可以***称之为***哈希表：

var aHashTable = Object.create(null);  
console.log(aHashTable.toString); // 未定义

此外，一些属性有特定的getter / setter方法​​，所以也可能导致混淆这个概念：

var a = new String("foo");  
a['length'] = 10;  
alert(a['length']); // 3

然而，即使认为“哈希”可能有一个“原型”（例如，在Ruby或Python里委托哈希对象的类），在ECMAScript里，这个术语也是不对的，因为2个表示法之间没有语义上的区别（即用点表示法a.b和a["b"]表示法）。

在ECMAScript中的“property属性”的概念语义上和"key"、数组索引、方法没有分开的，这里所有对象的属性读写都要遵循统一的规则：检查原型链。

在下面Ruby的例子中，我们可以看到语义上的区别：

a = {}  
a.class # Hash  
   
a.length # 0  
   
# new "key-value" pair  
a['length'] = 10;  
   
# 语义上，用点访问的是属性或方法，而不是key  
   
a.length # 1  
   
# 而索引器访问访问的是hash里的key  
   
a['length'] # 10  
   
# 就类似于在现有对象上动态声明Hash类  
# 然后声明新属性或方法  
   
class Hash  
 def z  
 100  
 end  
end  
   
# 新属性可以访问  
   
a.z # 100  
   
# 但不是"key"  
   
a['z'] # nil

ECMA-262-3标准并没有定义“哈希”（以及类似）的概念。但是，有这样的结构理论的话，那可能以此命名的对象。

**对象转换**

将对象转化成原始值可以用valueOf方法，正如我们所说的，当函数的构造函数调用做为function（对于某些类型的），但如果不用new关键字就是将对象转化成原始值，就相当于隐式的valueOf方法调用：

var a = new Number(1);  
var primitiveA = Number(a); // 隐式"valueOf"调用  
var alsoPrimitiveA = a.valueOf(); // 显式调用  
   
alert([  
 typeof a, // "object"  
 typeof primitiveA, // "number"  
 typeof alsoPrimitiveA // "number"  
]);

这种方式允许对象参与各种操作，例如：

var a = new Number(1);  
var b = new Number(2);  
   
alert(a + b); // 3  
   
// 甚至  
   
var c = {  
 x: 10,  
 y: 20,  
 valueOf: function () {  
 return this.x + this.y;  
 }  
};  
   
var d = {  
 x: 30,  
 y: 40,  
 // 和c的valueOf功能一样  
 valueOf: c.valueOf  
};  
   
alert(c + d); // 100

valueOf的默认值会根据根据对象的类型改变（如果不被覆盖的话），对某些对象，他返回的是this——例如：Object.prototype.valueOf()，还有计算型的值：Date.prototype.valueOf()返回的是日期时间：

var a = {};  
alert(a.valueOf() === a); // true, "valueOf"返回this  
   
var d = new Date();  
alert(d.valueOf()); // time  
alert(d.valueOf() === d.getTime()); // true

此外,对象还有一个更原始的代表性——字符串展示。 这个toString方法是可靠的，它在某些操作上是自动使用的：

var a = {  
 valueOf: function () {  
 return 100;  
 },  
 toString: function () {  
 return '\_\_test';  
 }  
};  
   
// 这个操作里，toString方法自动调用  
alert(a); // "\_\_test"  
   
// 但是这里，调用的却是valueOf()方法  
alert(a + 10); // 110  
   
// 但，一旦valueOf删除以后  
// toString又可以自动调用了  
delete a.valueOf;  
alert(a + 10); // "\_test10"

Object.prototype上定义的toString方法具有特殊意义，它返回的我们下面将要讨论的内部[[Class]]属性值。

和转化成原始值（ToPrimitive）相比，将值转化成对象类型也有一个转化规范（ToObject）。

一个显式方法是使用内置的Object构造函数作为function来调用ToObject（有些类似通过new关键字也可以）：

var n = Object(1); // [object Number]  
var s = Object('test'); // [object String]  
   
// 一些类似，使用new操作符也可以  
var b = new Object(true); // [object Boolean]  
   
// 应用参数new Object的话创建的是简单对象  
var o = new Object(); // [object Object]  
   
// 如果参数是一个现有的对象  
// 那创建的结果就是简单返回该对象  
var a = [];  
alert(a === new Object(a)); // true  
alert(a === Object(a)); // true

关于调用内置构造函数，使用还是不适用new操作符没有通用规则，取决于构造函数。 例如Array或Function当使用new操作符的构造函数或者不使用new操作符的简单函数使用产生相同的结果的：

var a = Array(1, 2, 3); // [object Array]  
var b = new Array(1, 2, 3); // [object Array]  
var c = [1, 2, 3]; // [object Array]  
   
var d = Function(''); // [object Function]  
var e = new Function(''); // [object Function]

有些操作符使用的时候，也有一些显示和隐式转化：

var a = 1;  
var b = 2;  
   
// 隐式  
var c = a + b; // 3, number  
var d = a + b + '5' // "35", string  
   
// 显式  
var e = '10'; // "10", string  
var f = +e; // 10, number  
var g = parseInt(e, 10); // 10, number  
   
// 等等

**属性的特性**

所有的属性（property） 都可以有很多特性（attributes）。

1. {ReadOnly}——忽略向属性赋值的写操作尝，但只读属性可以由宿主环境行为改变——也就是说不是“恒定值” ;
2. {DontEnum}——属性不能被for..in循环枚举
3. {DontDelete}——糊了delete操作符的行为被忽略（即删不掉）;
4. {Internal}——内部属性，没有名字（仅在实现层面使用），ECMAScript里无法访问这样的属性。

*注意，在ES5里{ReadOnly}，{DontEnum}和{DontDelete}被重新命名为[[Writable]]，[[Enumerable]]和[[Configurable]]，可以手工通过Object.defineProperty或类似的方法来管理这些属性。*

var foo = {};  
   
Object.defineProperty(foo, "x", {  
 value: 10,  
 writable: true, // 即{ReadOnly} = false  
 enumerable: false, // 即{DontEnum} = true  
 configurable: true // 即{DontDelete} = false  
});  
   
console.log(foo.x); // 10  
   
// 通过descriptor获取特性集attributes  
var desc = Object.getOwnPropertyDescriptor(foo, "x");  
   
console.log(desc.enumerable); // false  
console.log(desc.writable); // true  
// 等等

**内部属性和方法**

对象也可以有内部属性（实现层面的一部分），并且ECMAScript程序无法直接访问（但是下面我们将看到，一些实现允许访问一些这样的属性）。 这些属性通过嵌套的中括号[[ ]]进行访问。我们来看其中的一些，这些属性的描述可以到规范里查阅到。

每个对象都应该实现如下内部属性和方法：

1. [[Prototype]]——对象的原型（将在下面详细介绍）
2. [[Class]]——字符串对象的一种表示（例如，Object Array ，Function Object，Function等）;用来区分对象
3. [[Get]]——获得属性值的方法
4. [[Put]]——设置属性值的方法
5. [[CanPut]]——检查属性是否可写
6. [[HasProperty]]——检查对象是否已经拥有该属性
7. [[Delete]]——从对象删除该属性
8. [[DefaultValue]]返回对象对于的原始值（调用valueOf方法，某些对象可能会抛出TypeError异常）。

通过Object.prototype.toString()方法可以间接得到内部属性[[Class]]的值，该方法应该返回下列字符串： "[object " + [[Class]] + "]" 。例如：

var getClass = Object.prototype.toString;  
   
getClass.call({}); // [object Object]  
getClass.call([]); // [object Array]  
getClass.call(new Number(1)); // [object Number]  
// 等等

这个功能通常是用来检查对象用的，但规范上说宿主对象的[[Class]]可以为任意值，包括内置对象的[[Class]]属性的值，所以理论上来看是不能100%来保证准确的。例如，document.childNodes.item(...)方法的[[Class]]属性，在IE里返回"String"，但其它实现里返回的确实"Function"。

// in IE - "String", in other - "Function"  
alert(getClass.call(document.childNodes.item));

**构造函数**

因此，正如我们上面提到的，在ECMAScript中的对象是通过所谓的构造函数来创建的。

Constructor is a function that creates and initializes the newly created object.  
构造函数是一个函数，用来创建并初始化新创建的对象。

对象创建（内存分配）是由构造函数的内部方法[[Construct]]负责的。该内部方法的行为是定义好的，所有的构造函数都是使用该方法来为新对象分配内存的。

而初始化是通过新建对象上下上调用该函数来管理的，这是由构造函数的内部方法[[Call]]来负责任的。

注意，用户代码只能在初始化阶段访问，虽然在初始化阶段我们可以返回不同的对象（忽略第一阶段创建的tihs对象）：

function A() {  
 // 更新新创建的对象  
 this.x = 10;  
 // 但返回的是不同的对象  
 return [1, 2, 3];  
}  
   
var a = new A();  
console.log(a.x, a); undefined, [1, 2, 3]

引用15章函数——创建函数的算法小节，我们可以看到该函数是一个原生对象，包含[[Construct]] ]和[[Call]] ]属性以及显示的prototype原型属性——未来对象的原型（注：NativeObject是对于native object原生对象的约定，在下面的伪代码中使用）。

F = new NativeObject();  
   
F.[[Class]] = "Function"  
   
.... // 其它属性  
   
F.[[Call]] = <reference to function> // function自身  
   
F.[[Construct]] = internalConstructor // 普通的内部构造函数  
   
.... // 其它属性  
   
// F构造函数创建的对象原型  
\_\_objectPrototype = {};  
\_\_objectPrototype.constructor = F // {DontEnum}  
F.prototype = \_\_objectPrototype

[[Call]] ]是除[[Class]]属性（这里等同于"Function" ）之外区分对象的主要方式，因此，对象的内部[[Call]]属性作为函数调用。 这样的对象用typeof运算操作符的话返回的是"function"。然而它主要是和原生对象有关，有些情况的实现在用typeof获取值的是不一样的，例如：window.alert (...)在IE中的效果：

// IE浏览器中 - "Object", "object", 其它浏览器 - "Function", "function"  
alert(Object.prototype.toString.call(window.alert));  
alert(typeof window.alert); // "Object"

内部方法[[Construct]]是通过使用带new运算符的构造函数来激活的，正如我们所说的这个方法是负责内存分配和对象创建的。如果没有参数，调用构造函数的括号也可以省略：

function A(x) { // constructor А  
 this.x = x || 10;  
}  
   
// 不传参数的话，括号也可以省略  
var a = new A; // or new A();  
alert(a.x); // 10  
   
// 显式传入参数x  
var b = new A(20);  
alert(b.x); // 20

我们也知道，构造函数（初始化阶段）里的shis被设置为新创建的对象 。

让我们研究一下对象创建的算法。

**对象创建的算法**

内部方法[[Construct]] 的行为可以描述成如下：

F.[[Construct]](initialParameters):  
   
O = new NativeObject();  
   
// 属性[[Class]]被设置为"Object"  
O.[[Class]] = "Object"  
   
// 引用F.prototype的时候获取该对象g  
var \_\_objectPrototype = F.prototype;  
   
// 如果\_\_objectPrototype是对象，就:  
O.[[Prototype]] = \_\_objectPrototype  
// 否则:  
O.[[Prototype]] = Object.prototype;  
// 这里O.[[Prototype]]是Object对象的原型  
   
// 新创建对象初始化的时候应用了F.[[Call]]  
// 将this设置为新创建的对象O  
// 参数和F里的initialParameters是一样的  
R = F.[[Call]](initialParameters); this === O;  
// 这里R是[[Call]]的返回值  
// 在JS里看，像这样:  
// R = F.apply(O, initialParameters);  
   
// 如果R是对象  
return R  
// 否则  
return O

请注意两个主要特点：

1. 首先，新创建对象的原型是从当前时刻函数的prototype属性获取的（这意味着同一个构造函数创建的两个创建对象的原型可以不同是因为函数的prototype属性也可以不同）。
2. 其次，正如我们上面提到的，如果在对象初始化的时候，[[Call]]返回的是对象，这恰恰是用于整个new操作符的结果：

function A() {}  
A.prototype.x = 10;  
   
var a = new A();  
alert(a.x); // 10 – 从原型上得到  
   
// 设置.prototype属性为新对象  
// 为什么显式声明.constructor属性将在下面说明  
A.prototype = {  
 constructor: A,  
 y: 100  
};  
   
var b = new A();  
// 对象"b"有了新属性  
alert(b.x); // undefined  
alert(b.y); // 100 – 从原型上得到  
   
// 但a对象的原型依然可以得到原来的结果  
alert(a.x); // 10 - 从原型上得到  
   
function B() {  
 this.x = 10;  
 return new Array();  
}  
   
// 如果"B"构造函数没有返回（或返回this）  
// 那么this对象就可以使用，但是下面的情况返回的是array  
var b = new B();  
alert(b.x); // undefined  
alert(Object.prototype.toString.call(b)); // [object Array]

让我们来详细了解一下原型

**原型**

每个对象都有一个原型（一些系统对象除外）。原型通信是通过内部的、隐式的、不可直接访问[[Prototype]]原型属性来进行的，原型可以是一个对象，也可以是null值。

**属性构造函数(Property constructor)**

上面的例子有有2个重要的知识点，第一个是关于函数的constructor属性的prototype属性，在函数创建的算法里，我们知道constructor属性在函数创建阶段被设置为函数的prototype属性，constructor属性的值是函数自身的重要引用：

function A() {}  
var a = new A();  
alert(a.constructor); // function A() {}, by delegation  
alert(a.constructor === A); // true

通常在这种情况下，存在着一个误区：constructor构造属性作为新创建对象自身的属性是错误的，但是，正如我们所看到的的，这个属性属于原型并且通过继承来访问对象。

通过继承constructor属性的实例，可以间接得到的原型对象的引用：

function A() {}  
A.prototype.x = new Number(10);  
   
var a = new A();  
alert(a.constructor.prototype); // [object Object]  
   
alert(a.x); // 10, 通过原型  
// 和a.[[Prototype]].x效果一样  
alert(a.constructor.prototype.x); // 10  
   
alert(a.constructor.prototype.x === a.x); // true

但请注意，函数的constructor和prototype属性在对象创建以后都可以重新定义的。在这种情况下，对象失去上面所说的机制。如果通过函数的prototype属性去编辑元素的prototype原型的话（添加新对象或修改现有对象），实例上将看到新添加的属性。

然而，如果我们彻底改变函数的prototype属性（通过分配一个新的对象），那原始构造函数的引用就是丢失，这是因为我们创建的对象不包括constructor属性：

function A() {}  
A.prototype = {  
 x: 10  
};  
   
var a = new A();  
alert(a.x); // 10  
alert(a.constructor === A); // false!

因此，对函数的原型引用需要手工恢复：

function A() {}  
A.prototype = {  
 constructor: A,  
 x: 10  
};  
   
var a = new A();  
alert(a.x); // 10  
alert(a.constructor === A); // true

注意虽然手动恢复了constructor属性，和原来丢失的原型相比，{DontEnum}特性没有了，也就是说A.prototype里的for..in循环语句不支持了，不过第5版规范里，通过[[Enumerable]] 特性提供了控制可枚举状态enumerable的能力。

var foo = {x: 10};  
   
Object.defineProperty(foo, "y", {  
 value: 20,  
 enumerable: false // aka {DontEnum} = true  
});  
   
console.log(foo.x, foo.y); // 10, 20  
   
for (var k in foo) {  
 console.log(k); // only "x"  
}  
   
var xDesc = Object.getOwnPropertyDescriptor(foo, "x");  
var yDesc = Object.getOwnPropertyDescriptor(foo, "y");  
   
console.log(  
 xDesc.enumerable, // true  
 yDesc.enumerable // false  
);

显式prototype和隐式[[Prototype]]属性

通常，一个对象的原型通过函数的prototype属性显式引用是不正确的，他引用的是同一个对象，对象的[[Prototype]]属性：

a.[[Prototype]] ----> Prototype <---- A.prototype

此外， 实例的[[Prototype]]值确实是在构造函数的prototype属性上获取的。

然而，提交prototype属性不会影响已经创建对象的原型（只有在构造函数的prototype属性改变的时候才会影响到)，就是说新创建的对象才有有新的原型，而已创建对象还是引用到原来的旧原型（这个原型已经不能被再被修改了）。

// 在修改A.prototype原型之前的情况  
a.[[Prototype]] ----> Prototype <---- A.prototype  
   
// 修改之后  
A.prototype ----> New prototype // 新对象会拥有这个原型  
a.[[Prototype]] ----> Prototype // 引导的原来的原型上

例如：

function A() {}  
A.prototype.x = 10;  
   
var a = new A();  
alert(a.x); // 10  
   
A.prototype = {  
 constructor: A,  
 x: 20  
 y: 30  
};  
   
// 对象a是通过隐式的[[Prototype]]引用从原油的prototype上获取的值  
alert(a.x); // 10  
alert(a.y) // undefined  
   
var b = new A();  
   
// 但新对象是从新原型上获取的值  
alert(b.x); // 20  
alert(b.y) // 30

因此，有的文章说“动态修改原型将影响所有的对象都会拥有新的原型”是错误的，新原型仅仅在原型修改以后的新创建对象上生效。

这里的主要规则是：对象的原型是对象的创建的时候创建的，并且在此之后不能修改为新的对象，如果依然引用到同一个对象，可以通过构造函数的显式prototype引用，对象创建以后，只能对原型的属性进行添加或修改。

**非标准的\_\_proto\_\_属性**

然而，有些实现（例如SpiderMonkey），提供了不标准的\_\_proto\_\_显式属性来引用对象的原型：

function A() {}  
A.prototype.x = 10;  
   
var a = new A();  
alert(a.x); // 10  
   
var \_\_newPrototype = {  
 constructor: A,  
 x: 20,  
 y: 30  
};  
   
// 引用到新对象  
A.prototype = \_\_newPrototype;  
   
var b = new A();  
alert(b.x); // 20  
alert(b.y); // 30  
   
// "a"对象使用的依然是旧的原型  
alert(a.x); // 10  
alert(a.y); // undefined  
   
// 显式修改原型  
a.\_\_proto\_\_ = \_\_newPrototype;  
   
// 现在"а"对象引用的是新对象  
alert(a.x); // 20  
alert(a.y); // 30

注意，ES5提供了Object.getPrototypeOf(O)方法，该方法直接返回对象的[[Prototype]]属性——实例的初始原型。 然而，和\_\_proto\_\_相比，它只是getter，它不允许set值。

var foo = {};  
Object.getPrototypeOf(foo) == Object.prototype; // true

**对象独立于构造函数**

因为实例的原型独立于构造函数和构造函数的prototype属性，构造函数完成了自己的主要工作（创建对象）以后可以删除。原型对象通过引用[[Prototype]]属性继续存在：

function A() {}  
A.prototype.x = 10;  
   
var a = new A();  
alert(a.x); // 10  
   
// 设置A为null - 显示引用构造函数  
A = null;  
   
// 但如果.constructor属性没有改变的话，  
// 依然可以通过它创建对象  
var b = new a.constructor();  
alert(b.x); // 10  
   
// 隐式的引用也删除掉  
delete a.constructor.prototype.constructor;  
delete b.constructor.prototype.constructor;  
   
// 通过A的构造函数再也不能创建对象了  
// 但这2个对象依然有自己的原型  
alert(a.x); // 10  
alert(b.x); // 10

**instanceof操作符的特性**

我们是通过构造函数的prototype属性来显示引用原型的，这和instanceof操作符有关。该操作符是和原型链一起工作的，而不是构造函数，考虑到这一点，当检测对象的时候往往会有误解：

if (foo instanceof Foo) {  
 ...  
}

这不是用来检测对象foo是否是用Foo构造函数创建的，所有instanceof运算符只需要一个对象属性——foo.[[Prototype]]，在原型链中从Foo.prototype开始检查其是否存在。instanceof运算符是通过构造函数里的内部方法[[HasInstance]]来激活的。

让我们来看看这个例子：

function A() {}  
A.prototype.x = 10;  
   
var a = new A();  
alert(a.x); // 10  
   
alert(a instanceof A); // true  
   
// 如果设置原型为null  
A.prototype = null;  
   
// ..."a"依然可以通过a.[[Prototype]]访问原型  
alert(a.x); // 10  
   
// 不过，instanceof操作符不能再正常使用了  
// 因为它是从构造函数的prototype属性来实现的  
alert(a instanceof A); // 错误，A.prototype不是对象

另一方面，可以由构造函数来创建对象，但如果对象的[[Prototype]]属性和构造函数的prototype属性的值设置的是一样的话，instanceof检查的时候会返回true：

function B() {}  
var b = new B();  
   
alert(b instanceof B); // true  
   
function C() {}  
   
var \_\_proto = {  
 constructor: C  
};  
   
C.prototype = \_\_proto;  
b.\_\_proto\_\_ = \_\_proto;  
   
alert(b instanceof C); // true  
alert(b instanceof B); // false

**原型可以存放方法并共享属性**

大部分程序里使用原型是用来存储对象的方法、默认状态和共享对象的属性。

事实上，对象可以拥有自己的状态 ，但方法通常是一样的。 因此，为了内存优化，方法通常是在原型里定义的。 这意味着，这个构造函数创建的所有实例都可以共享找个方法。

function A(x) {  
 this.x = x || 100;  
}  
   
A.prototype = (function () {  
   
 // 初始化上下文  
 // 使用额外的对象  
   
 var \_someSharedVar = 500;  
   
 function \_someHelper() {  
 alert('internal helper: ' + \_someSharedVar);  
 }  
   
 function method1() {  
 alert('method1: ' + this.x);  
 }  
   
 function method2() {  
 alert('method2: ' + this.x);  
 \_someHelper();  
 }  
   
 // 原型自身  
 return {  
 constructor: A,  
 method1: method1,  
 method2: method2  
 };  
   
})();  
   
var a = new A(10);  
var b = new A(20);  
   
a.method1(); // method1: 10  
a.method2(); // method2: 10, internal helper: 500  
   
b.method1(); // method1: 20  
b.method2(); // method2: 20, internal helper: 500  
   
// 2个对象使用的是原型里相同的方法  
alert(a.method1 === b.method1); // true  
alert(a.method2 === b.method2); // true

**读写属性**

正如我们提到，读取和写入属性值是通过内部的[[Get]]和[[Put]]方法。这些内部方法是通过属性访问器激活的：点标记法或者索引标记法：

// 写入  
foo.bar = 10; // 调用了[[Put]]  
   
console.log(foo.bar); // 10, 调用了[[Get]]  
console.log(foo['bar']); // 效果一样

让我们用伪代码来看一下这些方法是如何工作的：

**[[Get]]方法**

[[Get]]也会从原型链中查询属性，所以通过对象也可以访问原型中的属性。

O.[[Get]](P):  
   
// 如果是自己的属性，就返回  
if (O.hasOwnProperty(P)) {  
 return O.P;  
}  
   
// 否则，继续分析原型  
var \_\_proto = O.[[Prototype]];  
   
// 如果原型是null，返回undefined  
// 这是可能的：最顶层Object.prototype.[[Prototype]]是null  
if (\_\_proto === null) {  
 return undefined;  
}  
   
// 否则，对原型链递归调用[[Get]]，在各层的原型中查找属性  
// 直到原型为null  
return \_\_proto.[[Get]](P)

请注意，因为[[Get]]在如下情况也会返回undefined：

if (window.someObject) {  
 ...  
}

这里，在window里没有找到someObject属性，然后会在原型里找，原型的原型里找，以此类推，如果都找不到，按照定义就返回undefined。

注意：in操作符也可以负责查找属性（也会查找原型链）：

if ('someObject' in window) {  
 ...  
}

这有助于避免一些特殊问题：比如即便someObject存在，在someObject等于false的时候，第一轮检测就通不过。

**[[Put]]方法**

[[Put]]方法可以创建、更新对象自身的属性，并且掩盖原型里的同名属性。

O.[[Put]](P, V):  
   
// 如果不能给属性写值，就退出  
if (!O.[[CanPut]](P)) {  
 return;  
}  
   
// 如果对象没有自身的属性，就创建它  
// 所有的attributes特性都是false  
if (!O.hasOwnProperty(P)) {  
 createNewProperty(O, P, attributes: {  
 ReadOnly: false,  
 DontEnum: false,  
 DontDelete: false,  
 Internal: false  
 });  
}  
   
// 如果属性存在就设置值，但不改变attributes特性  
O.P = V  
   
return;

例如：

Object.prototype.x = 100;  
   
var foo = {};  
console.log(foo.x); // 100, 继承属性  
   
foo.x = 10; // [[Put]]  
console.log(foo.x); // 10, 自身属性  
   
delete foo.x;  
console.log(foo.x); // 重新是100,继承属性

请注意，不能掩盖原型里的只读属性，赋值结果将忽略，这是由内部方法[[CanPut]]控制的。

// 例如，属性length是只读的，我们来掩盖一下length试试  
   
function SuperString() {  
 /\* nothing \*/  
}  
   
SuperString.prototype = new String("abc");  
   
var foo = new SuperString();  
   
console.log(foo.length); // 3, "abc"的长度  
   
// 尝试掩盖  
foo.length = 5;  
console.log(foo.length); // 依然是3

但在ES5的严格模式下，如果掩盖只读属性的话，会保存TypeError错误。

**属性访问器**

内部方法[[Get]]和[[Put]]在ECMAScript里是通过点符号或者索引法来激活的，如果属性标示符是合法的名字的话，可以通过“.”来访问，而索引方运行动态定义名称。

var a = {testProperty: 10};  
   
alert(a.testProperty); // 10, 点  
alert(a['testProperty']); // 10, 索引  
   
var propertyName = 'Property';  
alert(a['test' + propertyName]); // 10, 动态属性通过索引的方式

这里有一个非常重要的特性——属性访问器总是使用ToObject规范来对待“.”左边的值。这种隐式转化和这句“在JavaScript中一切都是对象”有关系，（然而，当我们已经知道了，JavaScript里不是所有的值都是对象）。

如果对原始值进行属性访问器取值，访问之前会先对原始值进行对象包装（包括原始值），然后通过包装的对象进行访问属性，属性访问以后，包装对象就会被删除。

例如：

var a = 10; // 原始值  
   
// 但是可以访问方法（就像对象一样）  
alert(a.toString()); // "10"  
   
// 此外，我们可以在a上创建一个心属性  
a.test = 100; // 好像是没问题的  
   
// 但，[[Get]]方法没有返回该属性的值，返回的却是undefined  
alert(a.test); // undefined

那么，为什么整个例子里的原始值可以访问toString方法，而不能访问新创建的test属性呢？

答案很简单：

首先，正如我们所说，使用属性访问器以后，它已经不是原始值了，而是一个包装过的中间对象（整个例子是使用new Number(a)），而toString方法这时候是通过原型链查找到的：

// 执行a.toString()的原理:  
   
1. wrapper = new Number(a);  
2. wrapper.toString(); // "10"  
3. delete wrapper;

接下来，[[Put]]方法创建新属性时候，也是通过包装装的对象进行的：

// 执行a.test = 100的原理：  
   
1. wrapper = new Number(a);  
2. wrapper.test = 100;  
3. delete wrapper;

我们看到，在第3步的时候，包装的对象以及删除了，随着新创建的属性页被删除了——删除包装对象本身。

然后使用[[Get]]获取test值的时候，再一次创建了包装对象，但这时候包装的对象已经没有test属性了，所以返回的是undefined：

// 执行a.test的原理:  
   
1. wrapper = new Number(a);  
2. wrapper.test; // undefined

这种方式解释了原始值的读取方式，另外，任何原始值如果经常用在访问属性的话，时间效率考虑，都是直接用一个对象替代它；与此相反，如果不经常访问，或者只是用于计算的话，到可以保留这种形式。

继承

我们知道，ECMAScript是使用基于原型的委托式继承。链和原型在原型链里已经提到过了。其实，所有委托的实现和原型链的查找分析都浓缩到[[Get]]方法了。

如果你完全理解[[Get]]方法，那JavaScript中的继承这个问题将不解自答了。

经常在论坛上谈论JavaScript中的继承时，我都是用一行代码来展示，事实上，我们不需要创建任何对象或函数，因为该语言已经是基于继承的了，代码如下：

alert(1..toString()); // "1"

我们已经知道了[[Get]]方法和属性访问器的原理了，我们来看看都发生了什么：

1. 首先，从原始值1，通过new Number(1)创建包装对象
2. 然后toString方法是从这个包装对象上继承得到的

为什么是继承的？ 因为在ECMAScript中的对象可以有自己的属性，包装对象在这种情况下没有toString方法。 因此它是从原理里继承的，即Number.prototype。

注意有个微妙的地方，在上面的例子中的两个点不是一个错误。第一点是代表小数部分，第二个才是一个属性访问器：

1.toString(); // 语法错误！  
   
(1).toString(); // OK  
   
1..toString(); // OK  
   
1['toString'](); // OK

**原型链**

让我们展示如何为用户定义对象创建原型链，非常简单：

function A() {  
 alert('A.[[Call]] activated');  
 this.x = 10;  
}  
A.prototype.y = 20;  
   
var a = new A();  
alert([a.x, a.y]); // 10 (自身), 20 (继承)  
   
function B() {}  
   
// 最近的原型链方式就是设置对象的原型为另外一个新对象  
B.prototype = new A();  
   
// 修复原型的constructor属性，否则的话是A了   
B.prototype.constructor = B;  
   
var b = new B();  
alert([b.x, b.y]); // 10, 20, 2个都是继承的  
   
// [[Get]] b.x:  
// b.x (no) -->  
// b.[[Prototype]].x (yes) - 10  
   
// [[Get]] b.y  
// b.y (no) -->  
// b.[[Prototype]].y (no) -->  
// b.[[Prototype]].[[Prototype]].y (yes) - 20  
   
// where b.[[Prototype]] === B.prototype,  
// and b.[[Prototype]].[[Prototype]] === A.prototype

这种方法有两个特性：

首先，B.prototype将包含x属性。乍一看这可能不对，你可能会想x属性是在A里定义的并且B构造函数也是这样期望的。尽管原型继承正常情况是没问题的，但B构造函数有时候可能不需要x属性，与基于class的继承相比，所有的属性都复制到后代子类里了。

尽管如此，如果有需要（模拟基于类的继承）将x属性赋给B构造函数创建的对象上，有一些方法，我们后来来展示其中一种方式。

其次，这不是一个特征而是缺点——子类原型创建的时候，构造函数的代码也执行了，我们可以看到消息"A.[[Call]] activated"显示了两次——当用A构造函数创建对象赋给B.prototype属性的时候，另外一场是a对象创建自身的时候！

下面的例子比较关键，在父类的构造函数抛出的异常：可能实际对象创建的时候需要检查吧，但很明显，同样的case，也就是就是使用这些父对象作为原型的时候就会出错。

function A(param) {  
 if (!param) {  
 throw 'Param required';  
 }  
 this.param = param;  
}  
A.prototype.x = 10;  
   
var a = new A(20);  
alert([a.x, a.param]); // 10, 20  
   
function B() {}  
B.prototype = new A(); // Error

此外，在父类的构造函数有太多代码的话也是一种缺点。

解决这些“功能”和问题，程序员使用原型链的标准模式（下面展示），主要目的就是在中间包装构造函数的创建，这些包装构造函数的链里包含需要的原型。

function A() {  
 alert('A.[[Call]] activated');  
 this.x = 10;  
}  
A.prototype.y = 20;  
   
var a = new A();  
alert([a.x, a.y]); // 10 (自身), 20 (集成)  
   
function B() {  
 // 或者使用A.apply(this, arguments)  
 B.superproto.constructor.apply(this, arguments);  
}  
   
// 继承：通过空的中间构造函数将原型连在一起  
var F = function () {};  
F.prototype = A.prototype; // 引用  
B.prototype = new F();  
B.superproto = A.prototype; // 显示引用到另外一个原型上, "sugar"  
   
// 修复原型的constructor属性，否则的就是A了  
B.prototype.constructor = B;  
   
var b = new B();  
alert([b.x, b.y]); // 10 (自身), 20 (集成)

注意，我们在b实例上创建了自己的x属性，通过B.superproto.constructor调用父构造函数来引用新创建对象的上下文。

我们也修复了父构造函数在创建子原型的时候不需要的调用，此时，消息"A.[[Call]] activated"在需要的时候才会显示。

为了在原型链里重复相同的行为（中间构造函数创建，设置superproto，恢复原始构造函数），下面的模板可以封装成一个非常方面的工具函数，其目的是连接原型的时候不是根据构造函数的实际名称。

function inherit(child, parent) {  
 var F = function () {};  
 F.prototype = parent.prototype  
 child.prototype = new F();  
 child.prototype.constructor = child;  
 child.superproto = parent.prototype;  
 return child;  
}

因此，继承：

function A() {}  
A.prototype.x = 10;  
   
function B() {}  
inherit(B, A); // 连接原型  
   
var b = new B();  
alert(b.x); // 10, 在A.prototype查找到

也有很多语法形式（包装而成），但所有的语法行都是为了减少上述代码里的行为。

例如，如果我们把中间的构造函数放到外面，就可以优化前面的代码（因此，只有一个函数被创建），然后重用它：

var inherit = (function(){  
 function F() {}  
 return function (child, parent) {  
 F.prototype = parent.prototype;  
 child.prototype = new F;  
 child.prototype.constructor = child;  
 child.superproto = parent.prototype;  
 return child;  
 };  
})();

由于对象的真实原型是[[Prototype]]属性，这意味着F.prototype可以很容易修改和重用，因为通过new F创建的child.prototype可以从child.prototype的当前值里获取[[Prototype]]：

function A() {}  
A.prototype.x = 10;  
   
function B() {}  
inherit(B, A);  
   
B.prototype.y = 20;  
   
B.prototype.foo = function () {  
 alert("B#foo");  
};  
   
var b = new B();  
alert(b.x); // 10, 在A.prototype里查到  
   
function C() {}  
inherit(C, B);  
   
// 使用"superproto"语法糖  
// 调用父原型的同名方法  
   
C.ptototype.foo = function () {  
 C.superproto.foo.call(this);  
 alert("C#foo");  
};  
   
var c = new C();  
alert([c.x, c.y]); // 10, 20  
   
c.foo(); // B#foo, C#foo

注意，ES5为原型链标准化了这个工具函数，那就是Object.create方法。ES3可以使用以下方式实现：

Object.create ||  
Object.create = function (parent, properties) {  
 function F() {}  
 F.prototype = parent;  
 var child = new F;  
 for (var k in properties) {  
 child[k] = properties[k].value;  
 }  
 return child;  
}  
  
// 用法  
var foo = {x: 10};  
var bar = Object.create(foo, {y: {value: 20}});  
console.log(bar.x, bar.y); // 10, 20

此外，所有模仿现在基于类的经典继承方式都是根据这个原则实现的，现在可以看到，它实际上不是基于类的继承，而是连接原型的一个很方便的代码重用。

**结论**