# 3.基本概念

## 3.1语法

### 3.1.1区分大小写

### 3.1.2 标识符

所谓标识符，就是指变量、函数、属性的名字，或者函数的参数。

Js标识符格式规则：

1. 第一个字符必须是一个字母、下划线或一个美元符号
2. 其它字母可以是字母、下划线、美元符号或数字
3. 按照惯例，ES标识符采用驼峰大小写格式

### 3.1.3 注释

单行注释 //

多行注释

/\*

\*多行注释

\*/

### 3.1.4 严格模式

ES5引入了严格模式（strict mode） 严格模式是为JavaScript定义了一种不同的解析与执行模型。在严格模式下，ECMAScript3中的一些不确定的行为将得到处理，而且对一些不安全的操作也会抛出错误。

整个脚本中启用严格模式，在整个脚本顶部添加以下代码

“use strict”;

在函数内部的上方包含这条编译指示，也可以指定函数在严格模式下执行

function doSomeThing(){

“use strict”;

....

}

### 3.1.5 语句

ECMAScript中的语句以一个分号结尾；如果省略分号，则由解析器确定语句的结尾

虽然分号不是必须的，但是建议不要省略，因为：

1. 避免不完整输入
2. 开发人员可以放心地通过删除多余空格来压缩ECMAScript代码，代码行结尾处没有 分号容易导致压缩错误
3. 加上分号在某些情况下增进代码的性能，因为这样解析器就不必再花费时间推测应 该在哪里插入分号了

始终在控制语句中使用代码块，即使代码中只有一条语句，可以使编码意图更加清晰，而且也能降低修改代码时出错的几率

## 3.2关键字和保留字

## 3.3变量

ECMAScript的变量是松散类型的，所谓的松散类型就是可以用来保存任何类型的数据。换句话说，每个变量仅仅是一个用于保存值得占位符而已。

var message;

像这样未经过初始化的变量，会保存一个特殊的值-undefined

var操作符定义的变量将成为该变量的作用域中的局部变量。

function test(){

var message = “hi”; //局部变量

}

test();

alert(message); //错误

省略var操作符，可以建立一个全局变量

function test(){

message = “hi”; //全局变量

}

test();

alert(message); //”hi”

不推荐，原因有：

1. 局部作用域中定义的全局变量很难维护
2. 由于相应变量不会马上就有定义而导致不必要的混乱
3. 未经生命的变量赋值在严格模式下会抛出ReferenceError错误

可以使用一条语句定义多个变量：

var message = “hi”,

found=false,

age=29;

## 3.4数据类型

ECMAScript中有5种简单数据类型，undefined、null、boolean、number、string，一种复杂数据类型object。

object本质上是一组无序的键值对组成的。

ECMAScript不支持任何创建自定义类型的机制，而所有值最终都将是上述六中数据类型之一。

### 3.4.1typeof操作符

typeof负责提供检测变量的数据类型，可能会返回下列某个字符串

1. undefined
2. boolean
3. string
4. number
5. object
6. function

说明：

1. typeof 操作符的操作数可以使变量，也可以是数值字面量
2. typeof 是一个操作符，而不是函数，因此括号尽管可以使用，但是不是必须的
3. typeof null = “object”，因为null被认为是一个空的对象引用
4. typeof reg 在Safari5及之前的版本、Chrome7及之前版本返回function，而其它浏览器 返回object

### 3.4.2 undefined类型

undefined类型只有一个值，即undefined，在使用var声明变量但未对其初始化时，这个变量的值就是undefined。

不过，包含undefined值得变量与尚未定义的变量还是不一样的。

var message;

//下面这个变量并未声明

//var age;

alert(message); //undefined

alert(age); //Uncaught ReferenceError: age is not defined(…)

对于尚未声明的变量，只能进行一项操作，即typeof。

var message;

console.log(message); //undefined

console.log(age); //undefined

即便未初始化的变量会被自动赋予undefined值，但显示初始化变量依然是明智的选择，如果可以做到这一点，那么当typeof操作符返回undefined值时，我们就知道被监测的变量还没有被声明，而不是尚未初始化。

### 3.4.3 null类型

null类型只有一个值，即null。null值表示一个空指针的对象

var car = null;

console.log(typeof car); //object

如果定义的变量准备在将来用于保存对象，那么最好将该变量初始化为null而不是其它值。

console.log(null == undefined); //true

console.log(null === undefined); //false

== 相等操作符，不过需要注意，这个操作符出于标胶的目的会转换其操作数

只要意在保存对象的变量还没有保存变量时，就应该让该变量保存null值。这样做不仅体现了null作为空指针对象的惯例，而且也进一步区分null和undefined

### 3.4.4 Boolean 类型

Boolean类型有两个字面值 true false。

虽然Boolean类型的字面值只有两个，但是ECMAScript中所有的类型都有与这两个值等价的值，而Boolean()函数可以讲其转换为对应的Boolean值

var message = “hello world”;

console.log(Boolean(message));

|  |  |  |
| --- | --- | --- |
| 数据类型 | 转换为true的值 | 转换为false的值 |
| Boolean | true | false |
| String | 任何非空字符串 | 空字符串 |
| Number | 任何非零的数字值 | 0和NaN |
| Object | 任何对象 | null |
| Undefined |  | null |

这些转换规则对理解流程控制语句（如if语句）自动执行相应的Boolean转换非常重要

var message = “hello message”;

if(message){

console.log(“Hello World”);

}

### 3.4.5 number 类型

number 类型，使用IEEE754格式来表示整数和浮点数值。

数值字面值格式：

1. 十进制 var intNum = 55;
2. 八进制 第一位必须是0，然后是八进制数字序列

var octalNum1 = 070; //八进制的56

var octalNum2 = 079; //无效的八进制数值，解析为79;

八进制字面量在严格模式下是无效的

1. 十六进制 以0x开头，后跟十六进制数字（0-9及A~F），字母大小写无所谓
2. 在进行算术计算时，所有八进制和十六进制的数值最终都会转换成十进制数值

浮点数值

浮点数值，就是该数值必须包含一个小数点，并且小数点后面至少有一个数字。虽然小数点之前可以没有整数，但是不推荐

var floatNum1 = 1.1;

var floatNum2 = 0.1;

var floatNum3 = .1; //有效，但不推荐

由于保存浮点数值需要的内存空间是保存整数的两倍，因此，如果小数点后没有跟任何数字，或者浮点数值本身就是一个整数，如1.0，那么该值会被转换成整数。

var floatNum1 = 1.; //小数点后面没有数字，解析为1

var floatNum2 = 10.0; //整数，解析为10

科学计数法

var floatNum = 3.125e7; //等于31250000

var floatNum1 = 3e-7; //等于0.0000003

在默认情况下，ECMAScript会将小数点后面带有6个0以上的数字转换为科学计数法表示

数值范围

Number.MIN\_VALUE = 5e-324;

Number.MAX\_VALUE = 1.7976931348623157e+308;

正无穷 Infinity 负无穷 -Infinity

超过数值范围的数值无法参与下一次运算

isFinite()函数判断数值是不是位于最大值和最小值之间;

NaN

非数值，Not a Number，

5/0 = NaN;

任何涉及到NaN的计算都会返回NaN，其次，NaN与任何值都不想等，包括NaN本身。

NaN == NaN; //false

isNaN()函数，判断参数是否不是数值；函数在接收到一个参数后，先尝试将这个值转换为数值，某些不是数字的会转换为数值，例如字符串”10”或者”false”。而任何不能转换为数值的都会导致这个函数返回true。

isNaN()函数也适用于对象，首先会调用对象的.valueOf()方法，然后确认方法返回值是否可以转换为数值，如果不能，则给予这个返回值再调用toString()方法，再测试返回值。

数制转换

Number()

可用于任何数据类型

转换规则如下：

1. 如果是Boolean值，true和false将分别转换为1和0
2. 如果是数值，只是简单的传入和返回
3. 如果是null值，返回0
4. 如果是undefined，返回NaN
5. 如果是字符串，遵循以下规则

如果字符串中只包含数字（包括前面带正号或负号的情况），则将其转为十进制数值， 如果有前导0，被忽略

如果字符串中包含有效的浮点格式，如“1.1”，则将其转换为对应的浮点数，同样，也 会忽略前导0

如果包含有效的十六进制格式，例如“0xf”，则将其转换为相同大小的十进制整数值

如果字符串是空的，则将其转换为0

如果字符串包含上述格式之外的字符，则将其转换为NaN

1. 如果是对象，则调用对象的valueOf（）方法，然后依照前面的规则转换返回的值。如果 转换的结果为NaN，则调用toString（）方法，然后再次依照前面的规则转换返回的字 符串值

parseInt()

专门用于把字符串转换成数值

看其是否符合数值格式，它会忽略前面的空格，直至找到第一个非空格字符

1. 如果第一个有效字符不是数字字符或者负号，parseInt（）会返回NaN
2. 对空字符串会返回NaN
3. 如果第一个是数字字符，parseInt（）函数会继续解析第二个字符，知道遇到非数字字符
4. ECMAScript对于八进制字面量，不觉被解析能力，如“070”，会直接解析为70
5. 八进制转换parseInt(“070”,8) ；就会用八进制解析

parseFloat()

专门用于把字符串转换为数值

1. 与parseInt()类似，parseFloat()也是逐个解析，知道字符末尾或者一个无效的浮点数字字符位置。
2. parseFloat()始终都会忽略前导0，只解析十进制格式
3. 如果没有小数点，返回整数

### 3.4.6 String类型

1. length属性可以得到字符串的长度，即16位Unicode字符的数目，如果字符串中包含双字节字符，那么length树形不会精确返回字符数目
2. 字符串是不可变的，一旦创建，值不可改变。
3. toString()方法，返回相应值的字符串表现
4. 数值、布尔值、对象、字符串都有toString()方法，但null和undefined值没有这个方法
5. 参数，输出值的基数，默认10

var num = 10;

console.log(num.toString()); //10

console.log(num.toString(2)); //1010

String()方法 转换规则

1. 如果有toString()方法，，则调用该方法，并返回相应的结果
2. 如果值是null或者undefined，则返回相应的字符串

### 3.4.7 Object 类型

ECMAScript中的对象其实就是一组数据和功能的组合。可通过new操作符后 跟要创建的对象类型的名称来创建。创建Object类型的实例并为其添加属性和方法。就可以创建自定义对象

Objectde 的每个实例都具有以下属性和方法

1. Constructor 保存着用于创建当前对象的函数
2. hasOwnProperty(propertyName) 检查给定的属性在当前对象实例中是否存在
3. isPrototypeOf(object)用于检查传入的对象是否是另一个对象的原型
4. propertyIsEnumerable(propertyName) 检查给定的属性是都能够使用for ..in 语句来枚举
5. toString()返回对象的字符串表示
6. valueOf() 返回对象的字符串、数值或布尔值表示，通常与toString()方法的返回值相同

## 3.5 操作符

### 3.5.1 一元操作符

一元操作符对任何值都是用，不仅适用于整数，还可以用于字符串、布尔值、浮点数值和对象，规则如下：

1. 在应用于一个包含有效数字字符的字符串时，先将其转换为数值，再执行操作，字符串变量变成数值变量，在转换时，用的Number（）函数
2. 在应用于一个不包含有效数值的字符串时，将变量的值设置成NaN
3. 在应用于布尔值时，现将其转为对应的0或1，再执行一元操作符操作
4. 在应用于浮点数值时，直接执行一元操作
5. 在应用于对象时，先调用valueOf()方法，以取得一个可供操作的值，然后对该值应用前述规则，如果是NaN，则调用toString()方法再应用前述规则

# 5 引用类型

## 5.1 Object类型

创建Object实例

1、使用new 操作符后跟Object构造函数

var person = new Object();

person.name = “tiantian”;

person.age = 29;

1. 使用对象字面量表示法

var person = {

name : “tiantian”;

age : 29

}

在使用对象字面量语法时，属性名也可以使用字符串

var person = {

“name” : “tiantian”,

“age” : 29,

5 : true

}

这里的数值属性名会自动转换为字符串

如果使用空花括号，则可以定义只包含默认属性和方法的对象

var person = {}; //与new Object()功能相同

person.name = “tiantian”;

person.age = 29;

关于对象字面量语法，我们推荐只在考虑对象属性名的可读性时使用。

在通过对象字面量定义对象时，实际上不会调用Object构造函数。

对象字面量也是向函数传递大量可选参数的首选方式

访问对象的树形

alert(person.name);

alert(person[“name”]);

alert(person[name]);

方括号的优点：

1. 可以通过变量来访问树形
2. 如果属性中包含会导致语法错误的字符，或使用属性名是关键字或保留字

通常，除非必须使用变量来访问树形，否则我们建议使用点表示法

## 5.2Array类型

ECMAScript的数组特点

1. 数组的每一项都可以保存任何类型的数据
2. 数组的长度是可以动态调整的

创建数组的基本方式

1. 使用Array构造函数

var colors = new Array();

如果预先知道数组要保存的项目的数量，也可以给构造函数传递该数量，而该数量会自动变成length属性

var colors = new Array(20);

也可以向构造函数传递数组中应该包含的项

var colors = new Array(“red”,”blue”,’green’);

当然，给构造函数传递一个值也可以创建数组，但是如果是数值，则会按照创建包含给定项数的数组；而如果是其它类型，则会创建包括那个值的数组。

var colors = new Array(3); //创建一个包含3项的数组

var colors = new Array(“3”); //创建一个包含一项，即字符串3的数组

另外，new 可以省略掉

1. 数组字面量表示法

var colors = [“red”,”blue”,”green”];

var names = [];

### 5.2.1 检测数组

instanceof 对于一个网页或者一个全局作用域而言，使用innstanceof操作符就可以得到满意的结果。

instanceof操作符的问题在于，它假定单一的全局执行环境。如果网页中包含多个框架，那实际上就存在两个不同的全局执行环境，从而存在两个以上不同版本的Array构造函数。

Array.isArray()方法，最终确定某个值到底是不是数组

### 5.2.2 转换方法

toString()，先调用每一项的toString()方法，然后每个值的字符串拼接并以逗号分隔的字符串

valueOf(),返回的还是数组

toLocaleString(),经常返回与toString()和valueOf()方法相同的值，但是也不总如此。当调用toLocaleString()方法时，它会调用每一项的toLocaleString()方法，而不是toString()方法。

var person1 = {

toLocaleString:function(){

return "tiantian";

},

toString : function(){

return "tiantian";

}

}

var person2 = {

toLocaleString:function(){

return "天天";

},

toString:function(){

return "tiantian"

}

}

var people = [person1,person2];

alert(people);

alert(people.toString());

alert(people.toLocaleString());

join()方法，可以使用不同的分隔符来构建这个字符串。join()方法只接受一个参数，即分隔符的字符串。

如果不给join()方法传入任何值，或者给它传入undefined，则使用逗号作为分隔符。如果某一项是null或undefined，那么以空字符串来表示

### 5.2.3 栈方法

ECMAScript 数组 栈，后进先出

push()方法可以接收任意数量的参数，把它们猪哥添加到数组末尾，并返回数组的长度

pop()方法则从数组末尾移除最后一项，减少数组的length，然后返回移除的项

### 5.2.4 队列方法

先进先出

从数组前端取得项的，减少数组的length，然后返回移除的项的方法是shift()

结合push()和shift()，可以像使用队列一样使用数组

ECMAScript还为数组提供了一个unshift()方法，unshift()与shift()方法的用途相反：

它能在数组前端添加任意个项并返回数组的长度

因此同时使用unshift()与pop()方法，可以相反方向模拟队列

### 5.2.5 重排序方法

reverse() 反转数组项的顺序

sort() 按照顺序排列数组项，先调用每个数组项的toString()的转型方法，然后比较得到的字符串

两种方法都不是理想方案

向sort()传递一个比较函数适用于大多数数据类型，

1. 、如果第一个参数应该位于第二个参数之前，则返回一个负数；
2. 如果两个参数相等，则返回0；
3. 如果第一个参数应该位于第二个参数之后，则返回一个正数。

对于数值或者valueOf()方法会返回数值类型的对象类型，可以使用以下比较函数

function compare(value1,value2){

return value1-value2;

}

### 5.2.6 操作方法

ECMAScript为操作已经包含在数组中的项提供了很多方法

concat()可以基于当前数组中的所有项创建一个新数组。

这个方法会先创建当前数组的一个副本，然后将接受到的参数添加到这个副本的末尾，最后返回新构建的数组

1. 在没有传递参数时，它只是复制当前数组并返回副本
2. 如果传递给一个或多个数组，则该方法会将这些数组中的每一项都添加到结果数组中
3. 如果传递的不是数组，这些值会简单的添加到结果数组的末尾

var colors = ["red","green","blue"];

var colors2 = colors.concat("yellow",["black","brown"]);

console.log(colors);

console.log(colors2);

slice() 它能够基于当前数组中的一或多个项建立一个新数组。

slice()方法会接受一个或两个参数，即要返回项的起始和结束位置，

1. 只有一个参数的情况下，slice()方法返回从该参数指定位置开始到当前数组末尾的所有项
2. 如果两个参数，该方法返回起始和结束位置之间的项，但不包括结束位置的项。

3、如果slice()方法的参数中有一个是负数，则用数组长度加上该数来确定相应的位置

如一个包含5项的数组上调用slice(-2,-1)与调用slice(3,4)得到的结果相同，如果结束为 止小于起始位置，则返回空数组

var colors = ["red","green","blue","yellow","purple"];

var colors2 = colors.slice(1);

var colors3 = colors.slice(1,4);

console.log(colors2);

console.log(colors3);

splice()方法 主要用途是向数组中部插入项

1. 删除 可以删除任意数量的项，只需要指定2个参数：要删除的第一项的位置和要删除的项数
2. 插入 可以向指定位置插入任意数量的项，只需提供三个参数：起始位置，0（要删除的项数）和要插入的项，如果要插入多个项，可再传入第四、第五，以至任意多个项
3. 替换 可以向指定位置插入任意数量的项，并同时删除任意数量的项，只需要指定3个参数：起始项位置、要删除的项数、和要插入的任意数量的项，插入的项不必与删除的项数相等

splice()方法始终都会返回一个数组，该数组中包含从原始数组中删除的项，如果没有删除任何项，则返回一个空数组。

### 5.2.7 位置方法

IndexOf() lastIndexOf()

这两个方法都接受两个参数：要查找的项和表示查找起点位置的索引（可选）

1. indexOf()方法从数组的开头位置向后查找，lastIndexOf()方法则从数组的末尾开始查找
2. 两个方法都返回要查找项在数组中的位置，或者在没找到的情况下返回-1。
3. 在比较第一个参数与数组中每一项时，会使用全等操作符，要求查找的想必须严格相等

var numbers = [1,2,3,4,5,4,3,2,1];

console.log(numbers.indexOf(4));

console.log(numbers.lastIndexOf(4));

console.log(numbers.indexOf(4,4));

console.log(numbers.lastIndexOf(4,4));

var person = {name:"tiantian"};

var people = [{name:"tiantian"}];

var morePeople = [person];

console.log(people.indexOf(person));

console.log(morePeople.indexOf(person));

### 5.2.8 迭代方法

1. every()

对数组中的每一项运行给定函数，如果该函数对每一项都返回true,则返回true

1. filter()

对数组中的每一项运行给定函数，返回该函数会返回true的项组成的数组

1. forEach()

对数组中的每一项运行给定函数，这个方法没有返回值

1. map()

对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组

1. some()

对数组中的每一项运行给定函数，如果该函数对任一项返回true，则返回true

var numbers = [1,2,3,4,5,4,3,2,1];

var everyResult = numbers.every(function(item,index,array){

return item > 2;

});

console.log(everyResult);

var someResult = numbers.some(function(item,index,array){

return item >2;

});

console.log(someResult);

var numbers = [1,2,3,4,5,4,3,2,1];

var filterResult = numbers.filter(function(item,index,array){

return item > 2;

});

console.log(filterResult);

var numbers = [1,2,3,4,5,4,3,2,1];

var mapResult = numbers.map(function(item,index,array){

return item\*2;

});

console.log(mapResult);

var numbers = [1,2,3,4,5,4,3,2,1];

var numers2 = [];

numbers.forEach(function(item,index,array){

numers2 [index] = item ++ ;

});

console.log(numers2 );

### 5.2.9 缩小方法

reduce() reduceRight()

1. 两个方法都会迭代数组的所有项，然后构建一个最终返回的值
2. reduce()方法从数组的第一项开始，reduceRight()则从最后一项开始
3. 两个方法都接收两个参数，一个在每项上都调用的函数，和作为虽小基础的初始值
4. 传入的参数接收4个参数，前一个值，当前值，项的索引，数组对象
5. 这个函数返货的任何值都会作为第一个参数自动传给下一项

var values = [1,2,3,4,5];

var sum = values.reduce(function(prev,cur,index,array){

return prev + cur;

});

console.log(sum);

## 5.3 Date类型

创建一个Date对象

var date = new Date();

1. 不传递参数的情况下，新创建的对象自动获得当前日期和时间
2. 如果想根据特定的时间创建日期对象，则必须传入该日期的毫秒数

Date.parse()接收一个字符串参数，然后尝试根据这个字符串返回相应日期的毫秒数

接收格式：

1. “月/日/年”，如“6/13/2004”
2. “英文/月名 日，年”，如“January 12,2014”；
3. “英文星期几 英文月 日 年 时：分：秒 时区”，如Tue May 25 2004 00:00:00 GMT-0700
4. ISO8601 扩展格式 YYYY-MM-DDTHH:mm:ss.sssZ，例如 2004-05-26T00:00:00 只有兼容ECMAScript5 的实现支持这种格式

如果传入Date.parse()方法的字符串不能表示日期，那么它会返回NaN。

如果直接将表示日期的字符串传递给Date()构造函数，也会在后台调用Date.parse()

var someDate = new Date(“May 24, 2014”);

Date.UTC()的参数分别是 年、基于0的月份（0-11）、日（1-31）、小时（0-23）、分钟、秒、毫秒。只有前两个参数（年、月）是必须的，如果没有提供其余的参数，月假设为1，其它参数假设为0

Date()构造函数也会模仿Date.UTC()但有一点不同：日期和时间基于本地时区而非GMT来创建。

Date.now()方法返回调用这个方法的日期和时间的毫秒数。

var start = Date.now();  
doSomeThing();

var stop = Date.now();

var result = stop - start;

在不支持它的浏览器中，使用+操作符把Date对象转换成字符串，也可以达到相同目的。

var start = +new Date();

doSomeThing();

var stop = +new Date();

var result = stop - start;

### 5.3.1 继承的方法

toLocaleString() 会按照与浏览器设置的地区相适应的格式返回日期和时间

这就意味着时间格式中可能会包含AM或PM，但不会包含时区信息

toString()方法则通常会返回带有时区信息的日期和时间

在不同的浏览器中返回的日期和时间格式可谓大相径庭，事实上，这一差别在显示时1没有什么价值

Date类型的valueOf()方法则根本不会返回字符串，而是返回日期的毫秒表示，因此可以方便比较日期

### 5.3.2日期格式化方法

1. toDateString() 以特定于实现的格式显示星期几、月、日和年
2. toTimeString() 以特定于实现的格式显示时、分、秒和时区
3. toLocaleDateString()以特定于地区的格式显示星期几、月、日和年
4. toLocaleTimeString()以特定于地区的格式显示时、分、秒

这些字符串格式化方法的输出也是因浏览器而异

toGMTString()与toUTCString()等价，不过现在ECMAScript推荐现在一律使用toUTCString()

### 5.3.3 日期/时间组件方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| getTime() | 返回表示日期的毫秒数；与valueOf()方法返回 |
| setTime(毫秒) | 以毫秒数设置日期，会改变整个日期 |
| getFullYear() | 取得4位数的年份(如2007而非07) |
| getUTCFullYear() | 返回UTC日期的4位数年份 |
| setFullYear(年) | 设置日期年份，传入的年分值必须是4位数字 |
| setUTCFullYear(年) | 设置UTC日期年份，传入的年分支必须是4位数字 |
| getMonth() | 返回日期中的月份，其中0表示1月，11代表12月 |
| getUTCMonth() | 返回UTC日期中的月份，其中0表示1月，11代表12月 |
| setMonth(月) | 设置日期的月份，传入的月份>=0，超过11增加年份 |
| setUTCMonth(月) | 设置UTC日期的月份，传入的月份>=0，超过11增加年份 |
| getDate() | 返回日期月份中的天数(1到31) |
| getUTCDate() | 返回UTC日期月份中的天数(1到31) |
| setDate() |  |
| setUTCDate() |  |
| getHours() |  |

## 5.4 RegExp类型

创建正则表达式

var expression = /pattern/flags;

pattern 可以是任何简单或复杂的正则表达式，可以包含字符、限定符、分组、向前查找以及反向引用。

flags标志

1. g 表示全局模式。即模式将被应用于所有字符串，而非发现第一个匹配项时立即停止。
2. i 表示不区分大小写(case insensitive)模式，即在确定匹配项时忽略模式与字符串的大小写
3. m 表示多行模式(multiline)，即在达到一行末尾时还会继续查找下一行中是否存在与模式匹配的项

/\*

\*匹配字符串中所有 at的实例

\*/

var patterna = /at/g;

/\*

\*匹配字符串中第一个bat或者cat的实例，不区分大小写

\*/

var pattern2 = /[bc]at/i;

/\*

\*匹配字所有以 at结尾3个字符的组合，不区分大小写

\*/

var pattern = /.at/gi;

与其它语言中的正则表达式类似，pattern中使用的所有元字符都必须转义。正则表达式中的元字符包括

( [ { \ ^ $ | ) ? \* + . }

这些元字符在正则中都有一或多种特殊用途，因此必须转义

构造函数定义法

/\*

\* 匹配第一个 bat 或cat，不区分大小写

\*/

var pattern = /[bc]at/i;

/\*

\* 与pattern1相同，只不过是使用构造函数创建的

\*/

var pattern = new RegExp(“[bc]at”,”i”);

由于RegExp构造函数的两个参数都是字符串，，所以在某些情况下，要对字符进行双重转义。

ECMAScript5明确规定，使用正则表达式字面量必须像直接调用RegExp构造函数一样，每次都创建新的RegExp实例。

### 5.4.1 RegExp实例属性

1. global布尔值，表示是否设置了g标志
2. ignoreCase 布尔值，表示是否设置了i标志
3. lastIndex 整数，表示开始搜索下一个匹配项的字符位置，从0算起
4. multiline 布尔值 表示是否设置了m标志
5. source 正则表达式的字符串表示，按照字面量形式返回

### 5.4.2 RegExp 实例方法

exec() 专门为捕获组而设计的

接收一个参数，即要应用模式的字符串

返回包含第一个匹配项信息的数组，在没有匹配项的情况下返回null。

返回的Array实例，包含两个额外属性：index和input

index表示匹配项在字符串中的位置，

input表示应用正则表达式的字符串

在数组中，第一项是与整个模式匹配的字符串，其他项是与模式中捕获组匹配的字符串（如果模式中没有捕获组，则该数组只包含一项）

对于exec()方法而言，即使在模式中设置了全局标志g，它每次也只返回一个匹配项

在不设置全局标志的情况下，在同一个字符串上多次调用exec()将始终返回第一个匹配项的的信息

在设置全局标志的情况下，每次调用exec()则都会在字符串中继续查找新匹配项。

var text = "cat, bat, sat, fat";

var pattern1 = /.at/g;

var matches = pattern1.exec(text);

console.log(matches.index);

console.log(matches[0]);

console.log(pattern1.lastIndex);

matches = pattern1.exec(text);

console.log(matches.index);

console.log(matches[0]);

console.log(pattern1.lastIndex);

test()

它接收一个字符串参数参数，在模式与该参数匹配的时候返回true，否则返回false

var text = "000-00-0000";

var pattern = /\d{3}-\d{2}-\d{4}/;

if(pattern.test(text)){

console.log("the pattern was mathced.");

}

RegExp实例集成的toLocaleString()和toString()方法都会返回正则表达式的字面量，与创建正则表达式的方式无关。

var pattern = new RegExp("\\[bc\\]at","gi");

console.log(pattern.toString());

console.log(pattern.toLocaleString());

正则表达式的valueOf()方法返回正则表达式本身。

### 5.4.3 RegExp构造函数属性

|  |  |  |
| --- | --- | --- |
| 长属性名 | 短属性名 | 说明 |
| input | $\_ | 最近一次要匹配的字符串。Opera未实现此属性 |
| lastMatch | $& | 最近一次的匹配项。Opera未实现此属性 |
| lastParen | $+ | 最近一次匹配的捕获组。Opera未实现此属性 |
| leftContext | $` | input字符串中lastMatch之前的文本 |
| multiline | $\* | 布尔值，表示是否所有表达式都是用多行模式 |
| rightContext | $’ | input字符串中lastMatch之后的文本 |

使用这些属性可以从exec()或test()执行的操作中提取更具体的信息

var text = “this has been a short summer”;

var pattern = /(.)hort/g;

/\*

\* 注意：Opera不支持input、lastMatch、lastParen和mutiline属性

\* IE不支持multiline属性

\*/

if(pattern.test(text)){

console.log(RegExp.input); //this has been a short summer

console.log(RegExp.leftContext); //this has been a

console.log(RegExp.rightContext); // summer

console.log(RegExp.lastMatch); //short

console.log(RegExp.lastParen); //s

console.log(RegExp.multiline); //false

}

## 5.5 Function类型

函数实际上是对象，每个函数都Function类型的实例，而且都与其它引用类型一样具有属性和方法

函数声明

function sum(num1,num2){

return num1 + num2;

}

这与使用函数表达式定义函数的方式相差无几

var sum = function(){

return num1 + num2 ;

};

Function构造函数,Function构造函数可以接收任意数量的参数，但最后一个参数始终都被看成函数体。

var sum = new Function(“num1”,”num2”,”return num1 + num3”);

由于函数名仅仅是指向函数的指针，因此函数名与包含对象指针的其它变量没有什么不同，如下例所示

function sum(num1,num2){

return num1 + num2;

}

console.log(sum(10,10));

var anotherSum = sum;

console.log(anotherSum(10,10));

sum = null;

console.log(anotherSum(10,10));

使用不带圆括号的函数名是访问函数指针，而非调用函数

### 5.5.1 没有重载（深入理解）

其实函数名就是一个指针

function addSomeNumber(num){

return num + 100;

}

function addSomeNumber(num){

return num + 200;

}

console.log(addSomeNumber(100)); //300

以上代码其余与下面的代码没什么区别

var addSomeNumber = function(num){

return num + 100;

}

addSomeNumber = function(num){

return num + 200;

}

console.log(addSomeNumber(100)); //300

在创建第二个函数时，实际上覆盖了引用第一个函数的变量addSomeNumber

### 5.5.2 函数声明与函数表达式

解析器在向执行环境中加载数据时，对函数声明和函数表达式并非一视同仁

1. 解析器会率先读取函数声明，并使其在执行任何代码之前可用
2. 函数表达式，则必须等到解析器执行到它所在的代码行，才会真正被执行

console.log(sum(10,10)); //20

function sum(num1,num2){

return num1 + num2;

}

console.log(sum(10,10));

var sum = function(num1,num2){

return num1 + num2;

}

### 5.5.3 作为值的函数

函数名本身就是变量，所以函数也可以作为值来使用。

function add10(num){

return num + 10;

}

function callSomeFunction(someFunction,someArgument){

return someFunction(someArgument);

}

var result1 = callSomeFunction(add10,10);

console.log(result1); //20

要访问函数而不执行函数的话，必须去掉函数名后面的那对圆括号

当然，可以从一个函数中返回另一个函数

function createComparisonFunction(propertyName){

return function(object1,object2){

var value1 = object1[propertyName];

var value2 = object2[propertyName];

if(value1 < value2){

return -1;

}else if(value1 > value2){

return 1;

}else{

return 0;

}

}

}

var data = [{name:”tiantian”,age:28},{name : “jingjing”,age:29}];

data.sort(createComparisonFunction(“name”));

console.log(data);

data.sort(createComparisonFunction(“age”));

console.log(data);

### 5.5.4 函数内部属性

arguments

arguments的主要用途是保存函数的参数，但这个对象还有一个属性callee，该属性是个指针，指向拥有这个arguments对象的函数

function factorial(num){

if(num <= 1){

return 1;

}else{

return num \* factorial(num-1);

}

}

这样写的时候，函数的执行与函数名耦合，而且当函数没有名称时，这种写法也不奏效

function factorial(num){

if(num <= 1){

return 1;

}else{

return num \* arguments.callee(num - 1);

}

}

this

this引用的是函数据以执行的环境对象（当王爷的全局作用域中调用函数时，this对象引用的就是window）

window.color = 'red';

var o = {color:'blue'};

function sayColor(){

console.log(this.color);

}

sayColor();

o.sayColor = sayColor;

o.sayColor();

caller 这个属性中保存着调用当前函数的函数引用如果是在全局作用域中，调用当前函数，它的值为null。

function outer() {

inner();

}

function inner() {

console.log(inner.caller);

}

outer();

以上代码会显示outer()函数的源代码。因为outer()调用了inner()，所以inner.caller就指向了outer()。为了实现松耦合，也可以通过arguments.callee.caller()来访问相同的信息

在严格模式下，arguments.callee会导致错误，所以，ECMAScript5还定义了arguments.caller属性，但在严格模式下也会导致错误，而在非严格模式下这个属性始终是undefined

严格模式下还有一个限制，不能为函数的caller属性赋值，否则会导致错误

### 5.5.5 函数的属性和方法

length 表示函数希望接收的命名参数的个数

function sayName(name) {

console.log(name);

}

function sum(num1, num2) {

return num1 + num2;

}

function sayHi() {

console.log("hi");

}

console.log(sayName.length); //1

console.log(sum.length); //2

console.log(sayHi.length); //0

prototype 是保存他们所有实例方法的真正所在，诸如toString()和valueOf()方法实际上都保存在prototype名下，只不过通过各自对象的实例访问罢了。

在创建自定义引用类型以及实现继承时，prototype属性是极为重要的。

在ECMAScript5中，prototype属性是不可枚举的，因此使用for-in无法发现。

每个函数都包含两个非继承而来的方法apply() ，call()。

这两个方法的用途都是在特定的作用域中调用函数，实际上等于设置this的值

apply()方法接收两个参数：一个是在其中运行函数的作用域，另一个是参数数组。其中，第二个参数可以使Array实例，也可以是arguments对象

function sum(sum1,sum2){

return sum1 + sum2;

}

function callSum1(sum1,sum2){

return sum.apply(this,arguments);

}

function callSum2(sum1,sum2){

return sum.apply(this,[sum1,sum2]);

}

console.log(callSum1(10,10));

console.log(callSum2(10,10));

在严格模式下，未指定环境对象而调用的函数，则this值不会转型为window。除非明确把函数添加到某个对象或者调用apply()或call()否则，this的值是undefined

call()方法apply()方法作用相同，区别在于其余参数都必须直接传递给函数，换句话说，在使用call()方法时，传递给函数的参数必须逐个列举出来

事实上，传递参数并不是apply()和call()真正的用武之地；他们真正强大的地方是能够扩充函数赖以运行的执行环境。

window.color = "red";

var o = {color: "blue"};

function sayColor() {

console.log(this.color);

}

sayColor(); //red

sayColor.call(this); //red

sayColor.call(window); //red

sayColor.call(o); //blue

使用call()或apply()来扩充作用域的最大好处，就是对象不需要与方法有任何耦合关系。

bind() 这个方法会创建一个函数的实例，其this对象会被绑定到传给bind()函数的值

window.color = "red";

var o = {color:"blue"};

function sayColor() {

console.log(this.color);

}

var objectSayColor = sayColor.bind(o);

objectSayColor(); //blue

每个函数继承的toLocaleString()和toString()、valueOf()方法都会返回函数的代码

## 5.6 基本包装类型

实际上，每当读取一个基本类型之的时候，后台就会创建一个对应的基本包装类型的对象，从而能够吊用一些方法来操作数据

var s1 = "some text";

var s2 = s1.toString();

s1是字符串，当然是基本类型，但是它竟然调用了方法，其实后台已经自动完成了一列写处理，当第二行代码访问s1时，访问过程处于一种读取模式，也就是从内存中读取这个字符串的值。而在赌气模式中访问字符串，后台都会自动完成下列处理

1. 创建String 类型的一个实例
2. 在实例上调用指定的方法；
3. 销毁这个实例

引用类型与基本包装类型的主要区别就是对象的生存周期

使用new操作符创建的引用类型的实例，在执行流离开当前执行环境之前，一直保存在内存中，而自动创建的基本包装类型的对象，则只存在于一行代码的瞬间，然后立即销毁。这意味着我们不能在运行时为基本类型值添加属性和方法

var s1 = "some text";

s1.color = "red";

console.log(s1.color); //undefined

Object构造函数也会像工厂方法一样，根据传入值的类型返回相应基本包装类型的实例

var obj = new Object("some text");

console.log(obj instanceof String); //true

要注意的是，使用new 调用基本包装类型的构造函数，与直接调用同名的转型函数是不一样的

var value = "25";

var numer = Number(value);

console.log(typeof number); //undefined

var obj = new Number(value);

console.log(typeof obj); //object

### 5.6.1 Boolean类型

Boolean类型重写了valueOf()方法，返回基本类型值true或者false

重写了toString()方法，返回“true”“false”

然而，Boolean对象在ECMAScript中的用处不大，因为它经常会造成误解。最常见的问题就是它在布尔表达式中使用Boolean对象

var falseObject = new Boolean(false);

var result = falseObject && true;

console.log(result); //true

var falseValue = false;

result = falseValue && true;

console.log(result); //false

基本类型和引用类型的布尔值还有两个区别

typeof 操作符对基本类型返回 boolean，而对引用类型返回object

Boolean对象时Boolean类型的实例，所以使用instanceof操作符测试Boolean对象会返回true，而测试基本类型的布尔值则会返回false

console.log(typeof falseObject); //object  
console.log(typeof falseValue); //boolean  
console.log(falseObject instanceof Boolean); //true  
console.log(falseValue instanceof Boolean); //false

理解基本类型的布尔值与Boolean对象之间的区别非常重要

当然，我们的建议是永远不要使用Boolean对象

### 5.6.2 Number对象

创建Number对象

var number = new Number(10);

Number类型也重写了valueOf() 、toLocaleString() 、toString()方法。

valueOf()方法返回对象表示的基本类型的数值

另外两个则返回字符串形式的数值

toString()方法传递一个表示基础的参数，告诉他返回几进制数值的字符串形式

var num = 10;

console.log(num.toString()); //10

console.log(num.toString(2)); //1010

console.log(num.toString(8)); //12

console.log(num.toString(10)); //10

console.log(num.toString(16)); //a

toFixed()方法会按照指定的小数位返回数值的字符串表示

var num = 10;

console.log(num.toFixed(2)); //10.00

如果数值本身包含的小数比指定的还多，那么接近指定的最大小数位的值就会舍入

var num = 10.005;

console.log(num.toFixed(2)); //10.01

但需要注意，不同的浏览器给这个方法设定的摄入规则可能会不同

toFixed()方法可以表示带有0到20个小数位的数值

toExpinential()

该方法可以返回以指数表示法,接受一个参数，同样指定输出结果中的小数位数

var num = 10;

console.log(num.toExponential(1)); //1.0e+1

如果你想得到表示某个数值最合适的格式，就应该使用toPrecision()方法

它可能范湖fixed格式，也可能范湖指数格式；具体规则是看哪种格式最合适。

这个参数接收一个参数，即表示数字的额所有数字的位数，不包括指数部分

var num = 99;  
console.log(num.toPrecision(1)); //1e+2  
console.log(num.toPrecision(2)); //99  
console.log(num.toPrecision(3)); //99.0

toPrecision()会根据要处理的数值决定到底是调用toFixed()还是调用toExponential()。而这三个方法都可以向上或向下舍入

toPrecision()方法可以表现1到21位小数

### 5.6.3 String类型

声明String类型

var stringObject = new String(“some text”);

String类型的valueOf() toString() toLocaleString()都返回对象所表示的字符串的值

length属性，表示字符串包含多少字符

var stringValue = “hello world”;

console.log(stringValue.length); //11

即使字符串中包含双字节字符，每个字符也仍然算一个字符

1. 字符方法

charAt(1) charCodeAt() 这两个方法都接收一个参数，即基于0的字符位置

charAt()方法以单字节字符串的形式返回给定位置的那个字符串

var stringValue = "hello world";

console.log(stringValue.charAt(0)) //h

如果想得到字符编码

var stringValue = "hello world";

console.log(stringValue.charCodeAt(0)) //104

charAt()也可以用方括号加数字索引来访问字符串中的特定字符

var stringValue = "hello world";

console.log(stringValue[1]); //e

1. 字符串的操作方法

concat(),用于将一或多个字符串拼接起来，返回拼接得到的新字符串

var stringValue = "hello world";

var result = stringValue.concat("!");

console.log(result);

console.log(stringValue);

实际上，concat()可以接收任意多个参数，但是实践中，更多还是用 +

slice() 第一个参数指定子字符串的开始位置，第二个参数表示子字符串的结束位置

substr() 第一个参数指定子字符串的开始位置，第二个参数指定的是返回的字符个数

substring() 第一个参数指定子字符串的开始位置，第二个参数表示子字符串的结束位置

var stringValue = "hello world";

console.log(stringValue.slice(3)); //lo world

console.log(stringValue.substring(3)); //lo world

console.log(stringValue.substr(3)); //lo world

console.log(stringValue.slice(3,7)); //lo w

console.log(stringValue.substring(3,7)); //lo w

console.log(stringValue.substr(3,7)); //lo worl

都对原始字符串没有影响

当传递的参数为负值 的情况下

slice()会将负数从末尾处开始，最后一个字符的下标定为-1

substr()会将第一个参数从末尾处开始，最后一个字符定为-1，第二个参数转换为0

substring()会将所有负的参数转换为0

var stringValue = "hello world";

console.log(stringValue.slice(-3)); //rld

console.log(stringValue.substring(-3)); / /hello world

console.log(stringValue.substr(-3)); //rld

console.log(stringValue.slice(3,-4)); //lo w

console.log(stringValue.substring(3,-4)); //hel

console.log(stringValue.substr(3,-4)); //””

1. 字符串的位置方法

indexOf() 从字符串的开头向后搜索字符串，并返回字符串的位置

lastIndexOf() 从字符串的末尾向前搜索字符串

var stringValue = "hello world";

console.log(stringValue.indexOf("o")); //4

console.log(stringValue.lastIndexOf("o")) //7

两个方法都可以接收第二个参数，表示从字符串中的哪个位置开始搜索。indexOf()会从该参数指定的位置向后搜索；而lastIndexOf()则会从指定位置向前搜索，忽略该位置之后的所有字符

var stringValue = "hello world";

console.log(stringValue.indexOf("o",6));

console.log(stringValue.lastIndexOf("o",6))

在使用第二个参数的情况下，可以使用循环调用indexOf()和lastIndexOf()来找到所有匹配的字符串

var stringValue = "Lorem ipsum dolor sit amet,consectetur adipisicing elit";

var positions = new Array();

var pos = stringValue.indexOf("e");

while (pos > -1){

positions.push(pos);

pos = stringValue.indexOf("e", pos + 1);

}

console.log(positions); //[3, 24, 31, 34, 51]

1. trim()这个方法会创建一个字符串的副本，删除前置集后缀的所有空格，然后返回结果

trimLeft() trimRight() 只有FireFox和Chrome、Safari Opera Edge支持，IE并不支持

var stringValue = " hello world ";  
var trimedStringValue = stringValue.trim();  
console.log(trimedStringValue); //hello world  
console.log(stringValue); // hello world

5、大小写转换

toLowerCase()

toLocaleLowerCase() 针对特定地区的实现

toUpperCase()

toLocaleUpperCase() 针对特定地区的实现

在不知道自己的代码将在哪种环境下运行的情况下，还是针对地区的方法比较稳妥

1. 字符串的模式匹配方法

match()在字符串上调用这个方法，本质上与调用RegExp的exec()方法相同

只接受一个参数，要么是正则表达式，要么是RegExp对象

var text = "bat cat sat fat";

var pattern = /.at/;

var matches = text.match(pattern);

console.log(matches.index); //0

console.log(matches[0]); //bat

console.log(pattern.lastIndex); //0

search() 参数与match()方法相同，返回字符串中第一个匹配项的索引，如果没有，则返回-1而且search始终是从字符串开头向后查找

replace() 接收两个参数

RegExp对象或者一个字符串(不会被转换成正则表达式)

可以使一个字符串或者一个函数

如果第一个参数是字符串，那么只会替换第一个子字符，要想替换所有字符串，唯一办法就是提供一个正则，而且要指定全局g标志

var text = "cat,bat,sat,fat";  
 var result = text.replace("at","ond");  
 console.log(result); //cond,bat,sat,fat

var text = "cat,bat,sat,fat";

var result = text.replace(/at/g,"ond");

console.log(result); //cond,bond,sond,fond

replace()方法的第二个参数也可以是一个函数。

1. 在只有一个匹配项的情况下，会向这个函数传递三个参数，模式的匹配项，位置，原始字符串
2. 在正则表达式中定义了多个捕获组的情况下，传递给函数的参数依次是模式的匹配项、第一个捕获组的匹配项，第二个捕获组的匹配项。。。但是最后两个参数仍然是模式的匹配项在字符串中的位置和原始字符串

这个函数应该返回一个字符串，表示应该被替换的额匹配项使用函数作为replace()方法的第二个参数可以实现更加精细的替换操作

function htmlEscape(text) {

return text.replace(/[<>"]/g,function (match,pos,originalText) {

switch (match){

case "<":

return "&lt;";

break;

case ">":

return "&gt;";

break;

case "&":

return "&amp;";

break;

case "\"":

return "&quot;";

}

});

}

console.log(htmlEscape("<p class=\"greetting\">Hello world!</p>"));

// &lt;p class=&quot;greetting&quot;&gt;Hello world!&lt;/p&gt;

split()可以基于指定的分隔符将一个字符串分割成多个子字符串，并将结果放进一个数组中

分隔符可以是一个字符串，也可以是一个RegExp对象。

第二个参数，用于指定数组大小，以确保返回的数组不会超过指定大小

1. localeCompare()
2. 如果字符串在字母表中应该排在字符串参数之前，则返回一个负数(大多数情况下是 -1)，具体情况视情况而定
3. 如果字符串等于字符串参数，则返回0
4. 如果字符串在字母表中应该排在字符串之后，则返回一个整数(大多数情况下是-1)

**var** stringValue = "yellow";  
console.log(stringValue.localeCompare("brick"));  
console.log(stringValue.localeCompare("yellow"));  
console.log(stringValue.localeCompare("zoo"));

lcoaleCompare()方法比较与众不同的地方，就是实现所支持的地区决定了这个方法的行为。比如美国以英语作为ECMAScript实现的标准语言，因此lcoaleCompare()就是区分大小写的，于是大写字母在字母表中排在小写字母前头。不过在别的地区就不一定了

1. fromCharCode()

静态方法，接收一个或多个字符编码，然后将热门转换成字符串。从本质上讲，这个方法与实例方法charCodeAt()执行的是相反操作

1. HTML方法

## 5.7 单体内置对象

有ECMAScript实现提供的、不依赖与宿主环境的对象，这些对象在ECMAScript程序执行之前就已经存在了

开发人员不必显示地实例化内置对象，因为它们已经实例化了

Global Math

### 5.7.1 Global对象

不属于任何对象的属性和方法，都是Global对象的属性

如isNaN() isFinite() parseInt() parseFloat()

除此之外，Global对象还包括其它一些方法

1. URI编码方法

可以对URI进行编码，它们用特殊的UTF-8编码替换所有无效的字符，从而让浏览器能够接受和理解

区别：

1. encodeURI() 主要用于整个URI，encodeURIComponent() 主要用于对URI中的一段进行编码
2. encodeURI()不会对本身属于URI的特殊字符进行编码，例如 : / ? #

而encodeURIComponent()则会对它发现的任何非标准字符进行编码

var uri = "http://www.wrox.com/illegal value.htm#start";

console.log(encodeURI(uri)); //http://www.wrox.com/illegal%20value.htm#start

console.log(encodeURIComponent(uri)); //http%3A%2F%2Fwww.wrox.com%2Fillegal%20value.htm%23start

与之相反的，decodeURI() decodeURIComponent()进行解码

1. eval() 就像是一个完整的ECMAScript解析器

它只接受一个参数，即要执行的ECMScript字符串

1. Global对象的属性

所有原生引用类型的构造函数，像Object Function；还有特殊的值 undefined、NaN、Infinity都是Global对象的属性。

|  |  |  |  |
| --- | --- | --- | --- |
| 属性 | 说明 | 属性 | 说明 |
| undefined | 特殊值 | Date | 构造函数 |
| NaN | 特殊值 | RegExp | 构造函数 |
| Infinity | 特殊值 | Error | 构造函数 |
| Object | 构造函数 | EvalError | 构造函数 |
| Array | 构造函数 | RangeError | 构造函数 |
| Function | 构造函数 | ReferenceError | 构造函数 |
| Boolean | 构造函数 | SyntaxError | 构造函数 |
| String | 构造函数 | TypeError | 构造函数 |
| Number | 构造函数 | URIError | 构造函数 |

ECMAScript5明确禁止给undefined NaN infinity赋值

1. window对象

ECMAScript虽然没有指出如何直接访问Gloabal对象，但Web浏览器都将这个对象作为window对象的一部分加以实现，因此在全局执行环境中声明的所有变量和函数，就都成了window对象的属性

var color = "red";

function sayColor() {

console.log(window.color);

}

window.sayColor(); //red

另一种获取Global对象的方法

var global = function(){

return this;

}();

在没有给函数明确指定this的值的情况下，this值等于Global对象

### 5.7.2 Math对象

Math对象提供的计算功能执行起来，要比在JS直接编写的计算功能快得多

1. Math对象的属性

|  |  |
| --- | --- |
| 属性 | 说明 |
| Math.E | 自然对数的底数，即常量e的值 |
| Math.LN10 | 10的自然对数 |
| Math.LN2 | 2的自然对数 |
| Math.LOG2E | 以2位底的e的对数 |
| Math.LOG10E | 以10为底的e的对数 |
| Math.PI | π的值 |
| Math.SQRT1\_2 | 1/2的平方根(即2的平方根的倒数) |
| Math.SQRT2 | 2的平方根 |

1. min() max()

用于确定一组数值中的最小值和最大值，这两个方法都可以接收任意多个数值参数

var max = Math.max(3,54,32,16);

console.log(max);

var min = Math.min(3,54,32,16);

console.log(min);

要找到数组中的最大值和最小值

var values = [1,2,3,4,5,6,7,8];

var max = Math.max.apply(this,values);

console.log(max);

1. 舍入方法

Math.ceil() 执行向上摄入，即它总是将数值向上摄入为最接近的整数

Math.floor() 执行向下舍入，即它总是将数值向下舍入为最接近的整数

Math.round() 指执行标准摄入，即它总是将数值四舍五入为最接近的整数

1. random()

返回介于0~1之间一个随机数，不包括1和0

从某个整数范围内随机选择一个值

Math.floor(Math.random()\*可能值的总和 + 第一个可能的值)

function selectFrom(lowerValue,upperValue){

var choices = upperValue - lowerValue;

return Math.floor(Math.random() \* choices + lowerValue);

}

1. 其它方法

|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 说明 | 方法 | 说明 |
| Math.abs() | 返回num的绝对值 | Math.asin(x) | 返回x的反正弦值 |
| Math.exp(num) | 返回Math.E的num次幂 | Math.atan(x) | 返回x的反正切值 |
| Math.log(num) | 返回num的自然对数 | Math.atan2(y,x) | 返回y/x的反正切值 |
| Math.pow(num,power) | 返回num的power次幂 | Math.cos(x) | 返回x的余弦值 |
| Math.sqrt(num) | 返回num的平方根 | Math.sin(x) | 返回x的余弦值 |
| Math.acos(x) | 返回x的反余弦值 | Math.tan(x) | 返回x的正切值 |

# 面向对象的程序设计

无序属性的集合，其属性可以包含基本值、对象或者函数

## 6.1 理解对象

var person = new Object();

person.name = "tiantian";

person.age = 29;

person.job = "Software Engineer";

person.sayName = function () {

console.log(this.name);

}

### 6.1.1 属性类型

1. 数据属性

数据属性包含一个数据值的位置，在这个位置可以读取和写入值

[[Configurable]] 表示能否通过delete删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为访问器属性。像前面例子中哪样直接在对象上定义的属性，他们的这个属性默认为true

[[Enumerable]] 表示能否通过for-in循环返回属性。像前面例子中哪样直接在属性上定义的属性，他们的这个特性默认值为true

[[Writable]] 表示能否修改属性的值。像前面例子中哪样直接在对象上定义的属性，他们的这个属性，默认为true

[[Value]]：包含这个属性的数据值。读取属性值的时候，从这个位置读；写入属性值的时候，把新值保存在这个位置。这个特性的默认值为undefined

要修改属性默认的特性，必须使用ECMAScript5的Object.defineProperty()方法。这个方法接收三个参数：属性所在的对象，属性的名字，和一个描述符对象。其中描述符对象的属性必须是：configurable/enumerable/writable和value。设置其中的一个或多个值，可以修改对应的特性值

var person = {};  
Object.defineProperty(person,"name",{  
 writable:false,  
 value:"tiantian"  
})  
console.log(person.name); //tiantian  
person.name = "yuanjing";  
console.log(person.name); //tiantian

把configurable设置为false，表示不能从对象中删除属性，如果对这个属性调用delete，则在严格模式下会导致错误，而且一旦把属性定位不可配置的，就不能再把它变回可配置了，此时再调用Object.defineProperty()方法修改除writable之外的特性，都会导致错误

var person = {};

Object.defineProperty(person, "name", {

configurable: false,

value: "tiantian"

});

console.log(person.name);

person.name = "yuanjing";

console.log(person.name);

可以多次调用Object.defineProperty()方法修改同一个属性，但在把configurable特殊设为false以后，就有会有限制

var person = {};  
Object.defineProperty(person,"name",{  
 configurable:false,  
 value:"tiantian"  
});

//抛出错误  
Object.defineProperty(person,"name",{  
 configurable:true,  
 value:"yuanjing"  
})

Uncaught TypeError: Cannot redefine property: name

at Function.defineProperty (<anonymous>)

at <anonymous>:6:8

1. 访问器属性

访问器属性不包含数据值；它们包含一对儿getter和setter函数

当读取访问器属性时，会调用getter函数，这个函数负责返回有效的值，在写入访问器属性时，会调用setter函数并传入值

[[Configurable]] 表示能否通过delete删除属性从而重新定义属性，能否修改属性的特性，或者能否把属性修改为数据属性。对于直接在对象上定义的属性，这个特性的默认值是true

[[Enumerable]]：表示能否通过for-in循环返回属性。对于直接在对象上定义的属性，这个特性的默认值是true

[[Get]] 在读取属性时调用的函数。默认值是undefined

[[Set]] 在写入属性时调用的函数。默认值为undefined

访问器属性不能直接定义，必须通过Object.defineProperty()来定义

var book = {  
 \_year : 2004,  
 edition:1  
};  
Object.defineProperty(book, "year", {  
 get: function () {  
 return this.\_year;  
 },  
 set: function (newValue) {  
 if (newValue > 2004) {  
 this.\_year = newValue;  
 debugger;  
 this.edition += newValue - 2004;  
 }  
 }  
});  
book.year = 2005;  
console.log(book.edition);

\_year前面的下划线是一种常用记号，用于表示只能通过对象方法访问的属性。

不一定非要同时制定getter和setter，只制定getter意味着属性是不能写，尝试写入会被忽略。类似地，没有制定setter函数的属性也不能读，否则在非严格模式下会返回undefined，

在严格模式下会抛出错误

### 6.1.2 定义多个属性

var book = {};  
Object.defineProperties(book,{  
 \_year:{  
 value:2004  
 },  
 edition:{  
 value:1  
 },  
 year:{  
 get:function () {  
 return this.\_year;  
 },  
 set:function (newValue) {  
 if (newValue > 2004) {  
 this.\_year = newValue;  
 this.edition += newValue - 2004;  
 }  
 }  
 }  
});

### 6.1.3 读取属性的特征

var book = {};

Object.defineProperties(book,{

\_year:{

value:2004

},

edition:{

value:1

},

year:{

get:function () {

return this.\_year;

},

set:function (newValue) {

if (newValue > 2004) {

this.\_year = newValue;

this.edition += newValue - 2004;

}

}

}

});

var descriptor = Object.getOwnPropertyDescriptor(book, "\_year");

console.log(descriptor.value); //2004

console.log(descriptor.configurable); //false

console.log(typeof descriptor.get); //undefined

descriptor = Object.getOwnPropertyDescriptor(book, "year");

console.log(descriptor.value); //undefined

console.log(descriptor.enumerable); //false

console.log(typeof descriptor.get) //function

## 6.2 创建对象

### 6.2.1 工厂模式

function createPerson(name,age,job) {  
 var o = new Object();  
 o.name = name;  
 o.age = age;  
 o.job = job;  
  
 o.sayName = function () {  
 console.log(this.name);  
 };  
 return o;  
}  
var person = createPerson("tiantian", 28, "程序猿");  
person.sayName(); //tiantian

console.log(typeof person); //object

工厂模式虽然解决了创建多个相似对象的问题，但是却没解决对象识别的问题，即怎样知道一个对象的类型

### 6.2.2 构造函数模式

function Person(name,age,job) {  
 this.name = name;  
 this.age = age;  
 this.job = job;  
 this.sayName = function () {  
 console.log(this.name);  
 }  
}  
var person = new Person("tiantian", 28, "程序狗");  
console.log(person instanceof Person); //true  
person.sayName();

步骤：

1. 创建一个新对象；
2. 将构造函数的作用域赋给新对象（因此this就指向了这个新对象）
3. 执行构造函数中的代码
4. 返回新对象

//当做构造函数使用  
var person = new Person("tiantian",28,"程序猿");  
person.sayName();  
//作为普通函数使用  
Person("yuanjing", 28, "老师");  
window.sayName();  
//在另一个对象的作用域中使用  
var o = new Object();  
Person.call(o,"tiantian",28,"程序猿");  
o.sayName();

1. 构造函数的问题

function Person(name,age,job) {  
 this.name = name;  
 this.age = age;  
 this.job = job;  
 var sayName = function () {  
 console.log(this.name);  
 }  
}

不同实例上的同名函数是不相等的，会导致不同的作用域和标识符解析，但是创建Function新实例的机制仍然是相同的

### 6.2.3 原型模式

每个对象都有一个prototype属性，这个属性是一个指针，指向一个对象，而这个对象的用途就是包含可以由特定类型的所有实例共享的属性和方法

function Person(){

}

Person.prototype.name = "tiantian";

Person.prototype.age = 28;

Person.prototype.job = "程序狗";

Person.prototype.sayName = function () {

console.log(this.name)

}

var person1 = new Person();

person1.sayName(); //天天

var person2 = new Person();

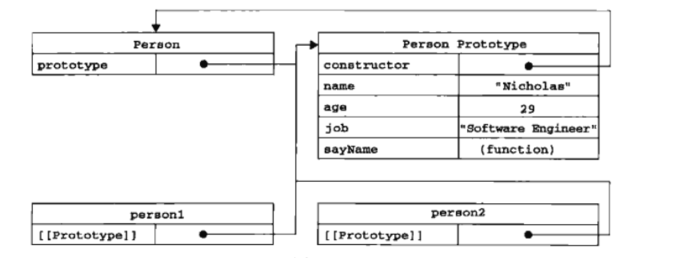
person2.sayName(); //天天

console.log(person1.sayName === person2.sayName); //true

1. 理解原型对象

无论什么时候，只要创建了一个新函数，就会根据一组特定的规则为该函数创建一个prototype属性，这个属性指向函数的原型对象。在默认情况下，所有原型对象都会自动获得一个constructor（构造函数）属性，这个属性包含一个指向prototype属性所在函数的指针。而通过这个构造函数，我们还可以继续为原型对象添加其他属性和方法

创建了自动以的构造函数以后，其原型对象默认只会取得constructor属性；至于其它方法，则都是从Object继承而来的。当调用构造函数创建一个实例后，该实例的内部包含一个指针(内部属性)



虽然在所有实现中都无法访问到[[Prototype]]，但可以通过isPrototypeOf()方法来确定对象之间是否存在这种关系。

console.log(Person.prototype.isPrototypeOf(person1)); //true  
console.log(Person.prototype.isPrototypeOf(person2)); //true

Object.getPrototypeOf() 这个方法返回了[[Prototype]]的值

console.log(Object.getPrototypeOf(person1) == Person.prototype); //true  
console.log(Object.getPrototypeOf(person1).name); //tiantian

虽然可以通过对象实例访问保存在原型中的值，但却不能通过对象实例重写原型中的值。如果我们在实例中添加了一个属性，而该属性与实例原型中的一个属性同名，那我们就在市里中创建该属性，该属性将会屏蔽原型中的那个属性

function Person(){  
}  
Person.prototype.name = "tiantian";  
Person.prototype.age = 28;  
Person.prototype.job = "程序狗";  
Person.prototype.sayName = function () {  
 console.log(this.name)  
}  
var person1 = new Person();  
var person2 = new Person();  
person1.name = "yuanjing";  
console.log(person1.name)  
console.log(person2.name);

添加同名属性，只会阻止我们访问原型中的那个属性，但不会修改那个属性。即使这个属性设置为null，也只会在实例中设置这个属性，而不会恢复其指向原型的连接。不过使用delete操作符则可以完全删除实例属性，从而能够重新访问原型中的属性

function Person(){  
}  
Person.prototype.name = "tiantian";  
Person.prototype.age = 28;  
Person.prototype.job = "程序狗";  
Person.prototype.sayName = function () {  
 console.log(this.name)  
}  
var person1 = new Person();  
var person2 = new Person();  
person1.name = "yuanjing";  
console.log(person1.name) //yuanjing  
console.log(person2.name); //tiantian  
delete person1.name;  
console.log(person1.name); //tiantian

使用hasOwnProperty()方法可以检测一个属性是存在于实例中，还是存在于原型中。这个方法继承于Object

function Person(){  
}  
Person.prototype.name = "tiantian";  
Person.prototype.age = 28;  
Person.prototype.job = "程序狗";  
Person.prototype.sayName = function () {  
 console.log(this.name)  
}  
var person1 = new Person();  
var person2 = new Person();  
console.log(person1.hasOwnProperty("name")); //false  
person1.name = "yuanjing";  
console.log(person1.name); //yuanjing  
console.log(person1.hasOwnProperty("name")); //true  
console.log(person2.name); //tiantian  
console.log(person2.hasOwnProperty("name")) //false  
delete person1.name;  
console.log(person1.name); //tiantian  
console.log(person1.hasOwnProperty("name")); //true

1. 原型与in操作符

function Person(){

}

Person.prototype.name = "tiantian";

Person.prototype.age = 28;

Person.prototype.job = "程序狗";

Person.prototype.sayName = function () {

console.log(this.name)

}

var person1 = new Person();

var person2 = new Person();

console.log(person1.hasOwnProperty("name")); //false

console.log("name" in person1); //true

person1.name = "yuanjing";

console.log(person1.hasOwnProperty("name")); //true

console.log("name" in person1);

name属性要么是直接在对象上访问到的，要么是通过原型访问到的，同时使用hasOwnProperty()和in操作符，就可以确定该属性到底是存在于对象中，还是存在于原型中。

function hasPrototypeProperty(object,name) {

return name in object && !object.hasOwnProperty(name);

}

在使用for-in循环时，返回的是所有能够通过对象访问的、可枚举的属性和方法，其中包括存在于实例中的属性，也包括存在于原型中的属性。屏蔽了原型中不可枚举属性的实例属性也会在for-in循环中返回。因为根据规定，所有开发人员定义的属性都是可枚举的

var o = {  
 toString:function () {  
 return "My Object";  
 }  
}  
  
for (var pro in o) {  
 if (pro == "toString") {  
 console.log("found toString");  
 }  
}

Object.keys() 获取对象上所有可枚举的实例属性

function Person(){  
}  
Person.prototype.name = "tiantian";  
Person.prototype.age = 28;  
Person.prototype.job = "程序狗";  
Person.prototype.sayName = function () {  
 console.log(this.name)  
}  
var keys = Object.keys(Person.prototype);  
console.log(keys); //name age job sayName  
var p1 = new Person();  
p1.name = "Yuanjing";  
p1.age = 28;  
var p1keys = Object.keys(p1);  
console.log(p1keys); //name age

Object.getOwnPropertyNames() 获取所有的实例属性

keys = Object.getOwnPropertyNames(Person.prototype);  
console.log(keys); //constructor,name,age,job,sayName

1. 更简单的原型语法

function Person() {  
}  
Person.prototype = {  
 name: "tiantian",  
 age:29,  
 job:"程序员",  
 sayName:function () {  
 console.log(this.name);  
 }  
};

例外：constructor属性不再指向Person了

var friend = new Person();

console.log(friend instanceof Object); //true

console.log(friend instanceof Person); //true

console.log(friend.constructor == Person) ;//false

console.log(friend.constructor == Object);//true

如果constructor的值真的很重要，可以这么写

function Person() {  
}  
Person.prototype = {  
 constructor:Person,  
 name: "tiantian",  
 age:29,  
 job:"程序员",  
 sayName:function () {  
 console.log(this.name);  
 }  
};

但是这样会导致它的[[Enumerable]]属性被设置为true，默认情况下，原生的constructor属性是不可枚举的。但是我们可以使用Object.defineProperty()

function Person() {  
}  
Person.prototype = {  
 name: "tiantian",  
 age:29,  
 job:"程序员",  
 sayName:function () {  
 console.log(this.name);  
 }  
};  
Object.defineProperty(Person.prototype, "constructor", {  
 enumerable: false,  
 value: Person  
});

1. 原型的动态性

由于在原型中查找值的过程是一次搜索，因此我们对原型对象所做的任何修改都能够立即从实例上反应出来，即使是先创建了实例后修改原型也照样如此

var friend = new Person();  
Person.prototype.sayHi = function () {  
 console.log("hi");  
};  
friend.sayHi(); //hi

但如果是重写整个原型对象，那么情况就不一样了，调用构造函数时，会为实例添加一个指向最初原型的[[Prototype]]指针，而把原型修改为另一个对象就等于切断了构造函数与最初原型之间的联系。请记住，实例中的指针仅指向原型，而不指构造函数

function Person() {  
}  
var friend = new Person();  
Person.prototype = {  
 constructor:Person,  
 name: "tiantian",  
 age:29,  
 job:"程序员",  
 sayName:function () {  
 console.log(this.name);  
 }  
};  
friend.sayName(); //Uncaught TypeError: friend.sayName is not a function

1. 原生对象的原型

通过原生对象的原型，不仅可以取得所有默认方法的引用，而且也可以定义新方法。可以修改定义对象的原型一样修改原生对象的原型，因此可以随时添加方法。

String.prototype.startsWith = function (text) {  
 return this.indexOf(text) == 0;  
};  
var mesg = "Hello World";  
console.log(mesg.startsWith("Hello"));

我们不推荐在产品化的程序中修改远程对象的原型。可能会导致明明冲突，而且这样做也可能会意外地重写原生方法

1. 原生模式的问题
2. 它省略了为构造函数传递初始化参数这一环节，结果所有实例在默认情况下都将获 得相同的属性值
3. 原型中所有属性都是被很多实例共享的，这种共享对于函数非常合适。对于那些包含基本值的属性倒也说得过去，毕竟，通过在实例上添加一个同名属性，可以隐藏原型中对应属性。然而，对于包含引用类型值的属性来说，问题就大了

function Person() {  
  
}  
Person.prototype = {  
 constructor:Person,  
 name: "tiantian",  
 age:29,  
 job:"程序员",  
 friends :["Shelby","Court"],  
 sayName:function () {  
 console.log(this.name);  
 }  
};  
var person1 = new Person();  
var person2 = new Person();  
person1.friends.push("Van");  
console.log(person1.friends); //["Shelby", "Court", "Van"]  
console.log(person2.friends); //["Shelby", "Court", "Van"]  
console.log(person1.friends === person2.friends); //true

实例一般都是要有属于自己的全部属性的，而这个问题正是我们很少看到有人单独使用原型模式的原因。

### 6.2.4 组合使用构造函数模式和原型模式

构造函数模式用于定义实例属性，而原型模式用于定义方法和共享的属性。结果，每个实例都会有自己的一份实例属性的副本，但同时又共享着对方法的引用，最大限度地节省了内存。另外，这种混合模式还支持向构造函数传递参数。

function Person(name, age, job) {  
 this.name = name;  
 this.age = age;  
 this.job = job;  
 this.friends = ["meixue", "yuanjing"];  
}  
Person.prototype = {  
 construtor: Person,  
 sayName:function () {  
 console.log(this.name);  
 }  
};  
var person1 = new Person("tiantian", 28, "程序狗");  
var person2 = new Person("xiaopan", 29, "SEO");  
person1.friends.push("Van");  
console.log(person1.friends); //["meixue", "yuanjing", "Van"]  
console.log(person2.friends); //["meixue", "yuanjing"]  
console.log(person1.friends === person2.friends); //false  
console.log(person2.sayName === person1.sayName); //true

### 6.2.5 动态原型模式

function Person(name, age, job) {  
 this.name = name;  
 this.age = age;  
 this.job = job;  
 if (typeof this.sayName != 'function') {  
 Person.prototype.sayName = function () {  
 console.log(this.name);  
 }  
 }  
}

使用动态原型模式时，不能使用对象字面量重写原型。如果在已经创建了实例的情况下重写原型，那么就会切断先用实例与新原型之间的联系

### 6.2.6 寄生构造函数模式

function Person(name,age,job) {  
 var o = new Object();  
 o.name = name;  
 o.age = age;  
 o.job = job;  
 o.sayName = function () {  
 console.log(this.name);  
 };  
 return o;  
}  
var friend = new Person("元天天",29,"软件工程师");  
friend.sayName();

## 6.3 继承

许多面向对象的语言有两种继承方式：接口继承和实现继承。

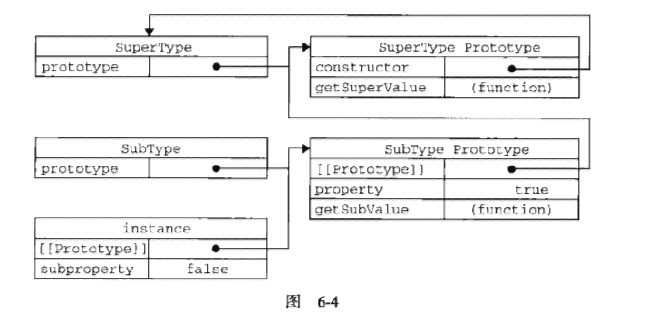
接口继承只继承方法签名，而实现继承则继承实际的方法。

由于函数没有签名，在ECMAScript中无法实现接口继承。ECMAScript只支持实现继承，而且其实现继承主要是依靠原型链

### 6.3.1 原型链

function SuperType() {  
 this.property = true;  
}  
SuperType.prototype.getSuperValue = function () {  
 return this.property;  
};  
function SubType() {  
 this.subProperty = false;  
}  
SubType.prototype = new SuperType();  
SubType.prototype.getSubValue = function () {  
 return this.subProperty;  
};  
var instance = new SubType();  
console.log(instance.getSuperValue());

其中的继承关系如图所示



我们没有使用SubType默认提供的原型，而是给它换了一个新原型；这个新原型就是SuperType的实例。于是，新原型不仅具有作为一个SuperType的实例所拥有的全部属性和方法，而且其内部还有一个指针，指向了SuperType的原型。最终结果就是这样的：instance只想SubType的原型，SubType的原型，SubType的原型又指向了SuperType的原型。

getSuperType()方法仍然还在SuperType.prototype中，但property则位于SubType.prototype中。这是因为property是一个实例属性，而getSuperValue()则是一个原型方法。既然SubType.prototype现在是SuperType的实例，那么property当然就位于该实例中了。此外，要注意instance.constructor现在指向的是SuperType，这是因为原来SubType.prototype中的constructor被重写的缘故

1. 别忘记默认的原型

所有引用类型都默认继承了Object，而这个继承也是通过原型链实现的。所有函数的默认原型都是Object的实例，因此默认原型都会包含一个内部指针，只想Object.prototype。这也正是所有自定义类型都会继承toString()、valueOf()等默认方法的根本原因。

1. 确定原型和实例的关系

Instanceof操作符

console.log(instance instanceof Object); //true  
 console.log(instance instanceof SuperType); //true  
 console.log(instance instanceof SubType); //true

只要是原型链中出现过的原型，都可以说是该原型链派生的实例的原型

isPrototypeOf()

console.log(Object.prototype.isPrototypeOf(instance)); //true

console.log(SuperType.prototype.isPrototypeOf(instance)); //true console.log(SubType.prototype.isPrototypeOf(instance)); //true

3、谨慎定义方法

子类型有时候需要重写超类型中的某个方法，或者要添加超类型中不存在的方法。但不管怎样，给原型家方法的代码一定要放在替换原型的语句之后

function SuperType() {

this.property = true;

}

SuperType.prototype.getSuperValue = function () {

return this.property;

};

function SubType() {

this.subProperty = false;

}

//继承了SuperType

SubType.prototype = new SuperType();

//添加新方法

SubType.prototype.getSubValue = function () {

return this.subProperty;

};

//重写超类方法

SubType.prototype.getSuperValue = function () {

return this.subProperty;

};

var instance = new SubType();

console.log(instance.getSuperValue());

在通过原型链实现继承时，不能使用对象字面量创建原形方法。因为这样会重写原型链

SubType.prototype = {

getSubValue:function () {

return this.subProperty;

},

getSuperValue:function () {

return this.property;

}

};

1. 原型链的问题

1、包含引用类型值的原型。包含引用类型值的原型的原型属性会被所有实例所共享；而这也是为什么要在构造函数中，而不是在原型对象中定义属性的原因。在通过原型来实现继承时，原型实际上会变成另一个类型的实例。于是，原先的实例属性也就成为了现在的子类的原型属性了

function SuperType() {

this.colors = ["red","blue","green"];

}

function SubType() {}

SubType.prototype = new SuperType();

var instance = new SubType();

instance.colors.push("black");

console.log(instance.colors);

var instance2 = new SubType();

console.log(instance2.colors);

1. 原型链的第二个问题，在创建子类型的实例时，不能向超类型的构造函数传递参数。实际上，应该时说没有办法在不影响所有对象实例的情况下，给超类型构造函数传递参数

### 6.3.2 借用构造函数

function SuperType() {

this.colors = ["red","blue","green"];

}

function SubType() {

SuperType.call(this);

}

var instance1 = new SubType();

instance1.colors.push("black");

console.log(instance1.colors);

var instance2 = new SubType();

console.log(instance2.colors);

1. 传递参数

可以再子类型构造函数中向超类构造函数传递参数

function SuperType(name){

This.name = name;

}

function SubType(name,age){

SuperType.call(this,name);

This.age = age;

}

var instance = new SubType();

console.log(instance.name);

console.log(instance.age);

1. 借用构造函数的问题
2. 方法都在构造函数中定义，因此函数复用就无从谈起
3. 在超类型的原型中定义的方法，对子类型不可见，结果所有类型都只能使用构造函数模式

function SuperType(name) {  
 this.name = name;  
}  
SuperType.prototype.getName = function () {  
 return this.name;  
}  
function SubType(name,age) {  
 SuperType.call(this,name);  
 this.age = age;  
}  
var constance = new SubType("tiantian",28);  
console.log(constance.getName());

Uncaught TypeError: constance.getName is not a function

at <anonymous>:12:23

### 6.3.3 组合继承

function SuperType(name) {  
 this.name = name;  
 this.color = ["red","blue","green"];  
}  
SuperType.prototype.sayName = function () {  
 console.log(this.name);  
}  
function SubType(name, age) {

//继承属性  
 SuperType.call(this,name);  
 this.age = age;  
}

//继承方法  
SubType.prototype = new SuperType();  
SubType.prototype.sayAge = function () {  
 console.log(this.age);  
}  
var instance = new SubType("tiantian", 28);  
instance.color.push("black");  
console.log(instance.color);  
instance.sayName();  
instance.sayAge();  
var instance2 = new SubType("yuanjing", 29);  
console.log(instance2.color);  
instance2.sayName();  
instance2.sayAge();

组合继承避免了原型链和借用构造函数的缺陷，融合了他们的优点，成为最常用的继承模式

### 6.3.4 原型式继承

function object(o) {

function F(){}

F.prototype = o;

return new F();

}

var person = {

name: "tiantian",

friends : ["yuanjing","qipeng","liuyuan"]

};

var anotherPerson = object(person);

anotherPerson.name = "pengxian";

anotherPerson.friends.push("Rob");

var yetAnotherPerson = object(person);

yetAnotherPerson.name = "haozhan";

yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);

新的对象以person为原型，所以它的原型中就包含一个基本类型值和一个引用类型值。这意味着person.friends不仅属于person所有，而且也会被anotherPerson以及yetAnotherPerson共享。实际上，这相当于又创建了person对象的两个副本

ECMAScript5 通过新增Object.create()方法规范了原型式继承

var person = {

name: "tiantian",

friends : ["yuanjing","qipeng","liuyuan"]

};

var anotherPerson = Object.create(person);

anotherPerson.name = "pengxian";

anotherPerson.friends.push("Rob");

var yetAnotherPerson = Object.create(person);

yetAnotherPerson.name = "haozhan";

yetAnotherPerson.friends.push("Barbie");

console.log(person.friends);

Object.create()方法的第二个参数与Object.defineProperties()方法的第二个参数格式相同，每个属性都是通过自己的描述符定义的。

var anotherPerson = Object.create(person, {

name: {

value: "feilong"

}

});

console.log(anotherPerson.name);

### 6.3.5 寄生式继承

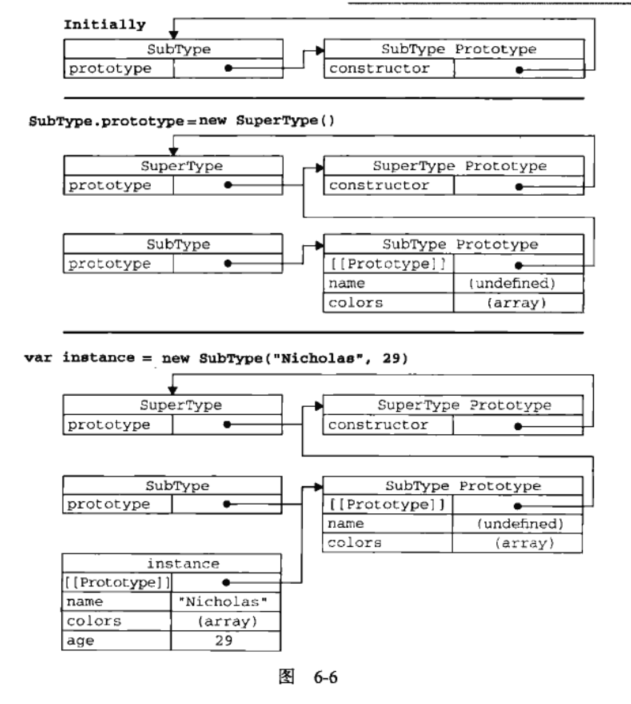
function createAnotherPerson(original) {  
 var clone = object(original);  
 clone.sayHi = function () {  
 console.log("hi");  
 };  
 return clone;  
}

在主要考虑对象而不是自定义类型和构造函数的情况下，寄生式继承也是一种有用的模式

使用寄生式继承来为对象添加函数，会由于不能做到函数复用而降低效率。

### 6.3.6 寄生组合式继承

function SuperType(name){  
 this.name = name;  
 this.color = ["red","blue","green"];  
}  
SuperType.prototype.sayName = function () {  
 console.log(this.name);  
}  
function SubType(name, age) {  
 SuperType.call(this, name); //第二次调用SuperType()  
 this.age = age;  
}  
SubType.prototype = new SuperType(); //第一次调用SuperType()  
SubType.prototype.constructor = SubType;  
SubType.prototype.sayAge = function () {  
 console.log(this.age);  
};  
var instace = new SubType("tiantian", 29);



function SuperType(name){  
 this.name = name;  
 this.color = ["red","blue","green"];  
}  
SuperType.prototype.sayName = function () {  
 console.log(this.name);  
}  
function SubType(name, age) {  
 SuperType.call(this, name);  
 this.age = age;  
}  
function inheritPrototype(subType, superType) {  
 var prototype = Object.create(subType.prototype);  
 prototype.constructor = subType;  
 subType.prototype = prototype;  
}  
inheritPrototype(SubType, SubType);  
SubType.prototype.sayAge = function () {  
 console.log(this.age);  
}

这个例子的高效体现在它只调用了一次SuperType构造函数，并且因此避免了在SubType.prototype上面创建不必要的多余的属性。与此同时，原型链还能保持不标；因此还能正常使用instanceof()和 isPrototypeOf()。寄生组合式继承是引用类型最理想的集成范式

# 函数表达式

定义函数的方式有两种：一种是函数声明，另一种就是函数表达式

1、函数声明：

function functionName(arg0,arg1,arg2){

//function body

}

Firfox、Safari、Chrome、Opera都给函数定义了一个非标准的name属性

console.log(functionName.name); //functionName

//注意IE不可用，会返回undefined

关于函数声明，它的一个重要特征就是函数声明提升(function declaration hoisting)，代码在执行之前会读取函数声明。这意味着可以把函数声明放在调用它的语句后面

sayHi();

function sayHi(){

console.log(“hi”);

}

1. 函数表达式

var functionName = function(arg0,arg1,arg2){

//函数体

}

这种情况下创建的函数叫匿名函数，匿名函数的name属性也是空字符串

在使用前必须赋值

sayHi();

var sayHi = function () {

console.log("Hi");

}

这样会报错，sayHi is not a function

if(condition){

function sayHi(){

console.log(“Hi”);

}

}else{

function sayHi(){

console.log(“Yo”);

}

}

永远不要这么做，但是如果使用函数表达式，就没什么问题了

var sayHi;

if(condition){

sayHi = function(){

console.log(“Hi”);

}

}else{

sayHi = function(){

console.log(“Yo”);

}

}

## 7.1 递归

递归函数是在一个函数通过名字调用自身的情况下构成的

function factorial(num){

if(num <= 1){

return 1;

}else{

return num \* factorial(num -1 );

}

}

这是一个经典的递归阶乘函数，虽然表面看起来没什么问题，但是下面的代码却可能导致它出错

var anotherFactorial = factorial;

factorial = null;

console.log(anotherFactorial(4)); //出错

使用arguments.callee可以解决这个问题

arguments.callee是一个指向正在执行的函数的指针，因此可以用它来实现函数的递归调用

但是在严格模式下，不能通过arguments.calllee，访问这个属性会导致错误。不过，可以使用命名函数表达式来达成相同的效果

var factorial = (function f(num) {

if (num <= 1) {

return 1;

}else {

return num \* f(num - 1);

}

});

## 7.2 闭包

闭包是指有权访问另一个函数作用域中变量的函数。

function createComparisonFunction(propertyName) {

return function (object1, object2) {

**var value1 = object1[propertyName];**

**var value2 = object2[propertyName];**

if (value1 < value2) {

return -1;

}else if (value1 > value2) {

return 1;

}else {

return 0;

}

};

}

突出显示的那两行代码是内部函数中的代码，这两行代码访问了外部函数中的变量propertyName。即使这个内部函数被返回了，而且是其它地方被调用了，但它仍然可以访问变量propertyName。之所以还能够访问这个变量，是因为内部函数的作用域链中包含createComparisonFunction()的作用域。

当某个函数第一次被调用时，会创建一个执行环境，及相应的作用域链，并把作用域链赋值给一个特殊的内部属性，即[[Scope]]。然后，使用this arguments和其它明明参数的值来初始化函数的活动对象。但在作用域链中，外部函数的活动对象始终处于第二位，外部函数的外部函数的活动对象处于第三位，直至作为作用域链重点的全局执行环境。

在函数执行过程中，为读取和写入变量的值，就需要在作用域链中查找变量。

function compare(value1,value2){

if(value1 < value2){

return -1;

}else if(value1 > value2){

return 1;

}else{

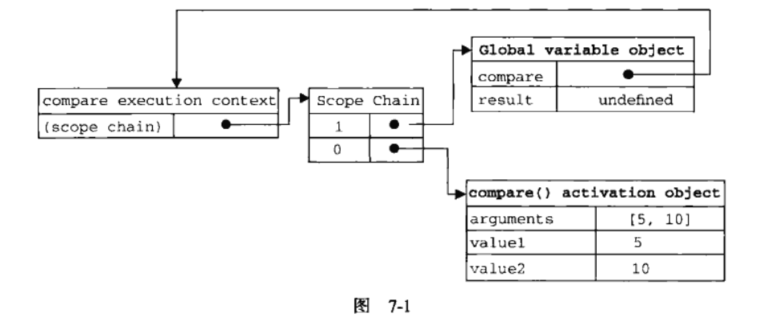
return 0;

}

}

var result = compare(5, 10);

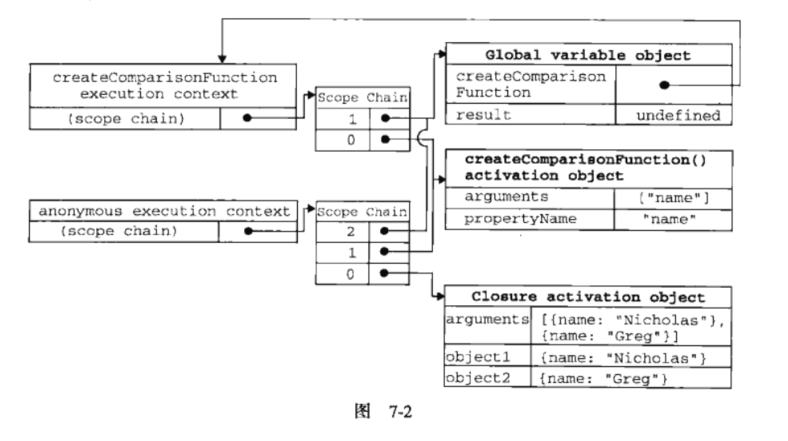
以上代码先定义了compare()函数，然后又在全局作用域中执行了它。当第一次调用compare()时，会创建一个包含this arguments value1 value2的活动对象。全局执行环境的变量对象（包含this result compare）在compare()执行环境的作用域链中则处于第二位。



后台的每个执行环境都有一个表示变量的对象--变量对象。全局环境的变量对象始终存在，而像compare()函数这样的局部变量环境的变量对象，则只在函数执行的过程中存在。在创建compare()函数时，会创建一个预先包含全局变量对象的作用域链，这个作用域链被保存在内部的[[Scope]]属性中。当调用compare()，会为函数创建一个执行环境，然后通过赋值函数的[[Scope]]属性中的对象属性中的作用域链。此后又有一个活动对象被创建并推入执行环境作用域链前端。

无论什么时候在函数中访问一个变量，都会从作用域链中搜索具有相应名字的变量。一般来讲，当函数执行完毕后，局部活动对象就会被销毁，内存中仅保存全局作用域（全局执行环境的变量对象）。但是闭包的情况又有所不同。

在外部函数中定义的内部函数将会将包含外部函数的活动对象添加到它的作用域中。因此，在createComparisonFunction()函数内部定义的函数的作用域链中，实际上将包含外部函数的活动对象



var compare = createComparsionFunction(“name”);

var result = compare({name:”tiantian”},{name:”yuanjing”});

在匿名函数从createComparisonFunction()中返回后，它的作用域链被初始化为包含createComparisonFunction()函数的活动对象和全局变量。这样匿名函数就可以访问在createComparisonFunction()中定义的所有变量。更为重要的是，createComparisonFunction()函数执行文笔后，其活动对象也不销毁，因为匿名函数的作用域链仍然引用这个活动对象。换句话说，当createComparisonFunction()函数返回后，其执行环境的作用域链会被销毁，但它的活动对象仍然会留在内存种；直到匿名函数被销毁后，createComparisonFunction()的活动对象才会被销毁，例如：

var compareNames = createComparisonFunction(“name”);

var result = compareNames({name:”tiantian”},{name:”yuanjing”});

compareNames = null;

由于闭包携带包含它的函数的作用域，因此会比其它函数占用更多的内存。过度使用闭包会导致内存占用过多。

### 7.2.1 闭包与变量

作用域链的这种配置机制引出了一个值得注意的副作用，即闭包只能取得包含函数中任何变量的最后一个值。因为闭包所保存的是整个变量对象，而不是某个特殊的变量。

function createFunctions(){

var result = new Array();

for(var i = 0;i < 10 ;i++){

result[i] = function(){

return i;

}

}

return result;

}

实际上，每个函数都会返回10；

function createFunctions() {

var result = new Array();

for (var i = 0;i < 10;i ++){

result[i] = function (num) {

return function () {

return num;

};

}(i);

}

return result;

}

### 7.2.2 关于this对象

匿名函数的执行环境具有全局性，因此其this对象通常指向window。但有时候由于编写闭包的方式不同，这一点可能不会那么明显

var name = "the window";

var object = {

name : "my object",

getNameFunction:function () {

return function(){

return this.name;

};

}

};

console.log(object.getNameFunction()()); // the window

不过，把外部作用域中的this对象保存在一个闭包能够访问的变量里，就可以让闭包访问该对象了。

var name = "the window";

var object = {

name : "my object",

getNameFunction:function () {

var that = this;

return function(){

return that.name;

};

}

};

console.log(object.getNameFunction()());

object.getName(); //my object

(object.getName)(); //my object

(object.getName = object.getName)(); //the window

### 7.2.3 内存泄漏

由于IE9之前的版本对JScript对象和Dom对象使用不同的垃圾回收机制，因此闭包在IE的这些版本中会导致一些特殊的问题。具体来说，如果闭包的作用域链中保存着一个HTML元素，那么就意味着该元素将无法回收。

function assignHandler() {

var element = document.getElementById("someElement");

element.onclick = function () {

console.log(element.id);

};

}

这个闭包创建了一个循环引用。由于匿名函数保存了一个对assignHanlder()活动对象的引用，因此就会导致无法见啥element的引用数。只要匿名函数存在，element的引用数至少也是1，因此它占用的内存就永远不会回收。不过这个问题可以通过赋值来解决。

function assignHandler() {

var element = document.getElementById("someElement");

**var id = element.id;**

element.onclick = function () {

**console.log(id);**

};

element = null;

}

## 7.3 模仿块级作用域

function outputNumbers(count) {  
 for (var i = 0; i< count;i++){  
 console.log(i);  
 }  
 console.log(i);  
}  
outputNumbers(5);

在java、C++等语言中，变量i只会在for循环遵循的语句中有定义，循环一旦结束，变量i就会销毁。可是在JavaScript中，变量i是定义在outputNumbers的活动对象中的，因此从它有定义开始，就可以在函数内部随处访问它。即使像下面这样错误地重新声明一个变量，也不会改变它的值。

function outputNumbers(count) {  
 for (var i = 0; i< count;i++){  
 console.log(i);  
 }  
 var i;  
 console.log(i);  
}  
outputNumbers(5);

JavaScript从来不会告诉你是否多次声明了同一个变量，遇到这种情况，它只会对后续的声明视而不见（不过，它会执行后续声明中的变量初始化）。匿名函数可以用来模仿块级作用域来避免这个问题。

(function () {

//块级作用域

})();

以上代码定义并立即执行了一个匿名函数。将函数声明包含在一对圆括号中，表示它实际是一个函数表达式。而紧随其后的另一对圆括号会立即调用这个函数。

function () {  
 //这里是块级作用域  
}(); //出错

是因为JavaScript将function关键字当做一个函数声明的开始，而函数声明后面不能跟圆括号。然而，函数表达式的后面可以跟圆括号。只需要加上一对圆括号即可。

## 7.4 私有变量

严格来讲，JavaScript中没有私有成员的概念；所有对象的属性都是共有的。不过，倒是有一个私有变量的概念。任何在函数中定义的变量，都可以认为是私有变量，因为不能在函数的外部访问这些变量。私有变量包括函数的参数，局部变量和在函数内部定义的其它函数。

function add(num1,num2){

var sum = num1 + num2;

return sum;

}

在这个函数内部，有3个私有变量：num1、num2和sum。在函数内部可以访问这几个变量，但是在函数外部则不能访问他们。如果在这个函数内部创建一个闭包，那么闭包通过自己的作用域链也可以访问这些变量。而利用这一点，就可以创建用于访问私有变量的共有方法。

我们把有权访问私有变量和私有函数的公有方法成为特权方法。有两种在对象上创建特权方法的方式。

function myObject(){  
 var privateVariable = 10;  
 function privateFunction(){  
 return false;  
 }  
 this.publicMethod = function(){  
 privateVariable++;  
 return privateFunction();  
 }  
}

第一种方法：在构造函数内部定义了所有私有变量和函数。然后又继续创建了能够访问这些私有成员的特权方法。

### 7.4.1 静态私有变量

通过在私有作用域中定义私有变量或函数，同样也可以创建特权方法

(function(){  
 var privateVariable = 10;  
 function priavateFunction() {  
 return false;  
 }  
 MyObject = function(){};  
  
 MyObject.prototype.publicMethod = function(){  
 privateVariable++;  
 return priavateFunction();  
 }  
})();

需要注意的是，这个模式在定义构造函数时并没有使用函数声明，而是使用了函数表达式。函数声明只能创建局部函数，但那并不是我们想要的。处于同样的原因，我们也没有在声明MyObject时使用var关键字。初始化未经声明的变量，总是会创建一个全局变量。因此，MyObject就成为一个全局变量，能够在私有所用于之外被访问到。但是在严格模式下回导致错误。

(function(){  
 var name = '';  
 Person = function(value){  
 name = value;  
 };  
 Person.prototype.getName = function(){  
 return name;  
 };  
 Person.prototype.setName = function(value){  
 name = value;  
 };  
})();  
var person1 = new Person("tiantian");  
console.log(person1.getName()); //tiantian  
person1.setName("yuanjing");   
console.log(person1.getName()); //yuanjing  
var person2 = new Person("yuanna");  
console.log(person1.getName()); //yuanna  
console.log(person2.getName()); //yuanna

以这种方式创建静态私有变量会因为使用原型而增进代码的复用，但每个实例都没有自己的私有变量。到底是使用实例变量，还是师太私有变量，最终还是要视你的需求而定

### 7.4.2 模块模式

模块模式是为单例创建私有变量和特权方法。所谓单例singleton，指的就是只有一个实例的对象，JavaScript是以对象字面量的方式来创建单例对象的。

var singleton = {

name : value,

method :function(){

//这里是方法的代码

}

}

模块模式通过为单例添加私有变量和特权方法能够使其增强。

var singleton = function(){

var privateVariable = 10;

function privateFunction(){

return false;

}

return {

publicProperty:true,

publicMethod:function(){

privateVariable++;

return privateFunction();

}

}

}();

var application = function(){  
 var components = new Array();  
 components.push(new BaseComponent());  
 return{  
 getComponentsCount:function(){  
 return components.length;  
 },  
 registerComponent:function(component){  
 if (typeof component == 'object') {  
 components.push(component);  
 }  
 }  
 };  
}();

### 7.4.3 增强的模块模式

**var** singleton = **function**(){  
 **var** privateVarible = 10;  
 **function** privateFuntion(){  
 **return false**;  
 }  
 **var** object = **new** CustomType();  
  
 object.publicProperty = **true**;  
  
 object.publicMethod = **function**(){  
 privateVarible++;  
 **return** privateFuntion();  
 };  
 **return** object;  
}();

# 8 、BOM

## 8.1 window对象

BOM的核心对象是window，它表示浏览器的一个实例。在浏览器中，window对象有双重角色，它既是JavaScript访问浏览器的一个接口，又是ECMAScript规定的Global对象。这意味着在网页中定义的任何一个对象、变量和函数，都以window作为其Global对象，因此有权限访问parseInt等方法

### 8.1.1 全局作用域

var age = 29;  
function sayAge(){  
 alert(this.age);  
}  
alert(window.age);  
sayAge();  
window.sayAge();

抛开全局变量会成为window对象的属性不谈，定义全局变量与window对象上直接定义属性还是有一点差别的：全局变量不能通过delete操作符删除，而在window对象上定义的属性可以。

**var** age = 29;  
window.color = **'red'**;  
*//在IE < 9 时抛出错误，在其它浏览器中都返回false***delete** window.age;  
*//在IE < 9 时抛出错误，在其它浏览器中都返回true***delete** window.color;  
console.log(window.age); *//29*console.log(window.color); *//undefined*

*经过实际研究发现，IE8会提示对象不支持此操作，IE9以上会成功删除两个属性，两个都会返回undefined，Chrome和FireFox都会如上图所示显示。*

*可见，对于IE，全局变量和window对象的属性是基本相同的，但对于Chrome和FireFox，是有delete的差别的*

### 8.1.2 窗口关系及框架

### 8.1.3 窗口位置

IE、Opera、Safari和Chrome都提供了screenLeft和screenTop属性，分别用于表示窗口相对于屏幕左边和上边的位置。

FireFox则在screenX和screenY属性中提供相同的窗口位置信息，Safari和Chrome也支持这两个属性。Opera虽然也支持screenX和screenY属性，但与screenLeft和screenTop属性并不对应。

var leftPos = (typeof window.screenLeft == 'number') ? window.screenLeft :window.screenX;  
var topPos = (typeof window.screenTop == 'number') ? window.screenTop :window.screenY;

还有一个小问题，在IE、Opera和Chrome中，screenLeft和screenTop中保存的是从屏幕左边和上边到由window对象表示的页面可见区域的距离。换句话说，如果window对象是最外层对象，而且浏览器窗口紧贴屏幕最上端，即y坐标为0，那么screenTop的值就是位于页面可见区域上方的浏览器工具栏的像素高度。但是在FireFox中，screenXscreenY中保存的是整个浏览器窗口相对于屏幕的坐标。即窗口的y坐标为0时返回0。

*完全不懂这块在说什么，根据测试Chrome和FireFox都是浏览器窗口相对于屏幕的坐标,*

*IE才是工具栏的高度，不招人待见的IE*

更让人捉摸不透的是，FireFox、Safari和Chrome始终返回每个框架的top.screenX和top.screenY值。即使在页面由于被设置了外边距便宜的情况下，相对于window对象使用screenX和screenY每次也都会返回相同的值。而IE和Opera则会给出框架相对于屏幕边界精确的坐标值.

最终的结果，就是无法在跨浏览器的条件下取得窗口左边和上边的精确坐标值。然而，使用moveTo()和moveBy()方法倒是有可能将窗口的位置精确地移动到一个新位置。

moveTo()接收的是新位置的x和y坐标值，而moveBy()接收的是在水平和垂直方向上移动的像素数。

不过貌似，chrome和FireFox以及IE都禁用了，所以，这几个函数没个卵用

### 8.1.4 窗口大小

IE9+、FireFox、Safari、Opera和Chrome均提供了4个属性

innerWidth、innerHeight、outerWidth和outerHeight。

IE9+、Safari、FireFox中，outerWidth和outerHeight返回浏览器窗口本身的尺寸

### 8.1.5 导航和打开窗口

window.open()方法既可以导航到一个特定的URL，也可以打开一个新的浏览器窗口。这个方法可以接收4个参数：URL，窗口目标，一个特性字符串以及一个标识新页面是否取代浏览器历史记录中当前加载页面的布尔值

如果window.open()传递了第二个参数，而且该擦书是已有窗口或框架名称，那么就会在具有该名称的窗口或框架中加载第一个参数指定的URL，此外，第二个参数也可以是下列任何一个特殊的窗口名称：\_self \_parent \_top \_blank

1. 弹出窗口

第三个参数是一个逗号分隔的设置字符串，表示新窗口具有哪些特性。

|  |  |  |
| --- | --- | --- |
| 设置 | 值 | 说明 |
| fullscreen | yes或no | 表示浏览器窗口是否最大化，IE好使，FireFox一直是全屏(规定高度和宽度后才会变成弹窗)，Chrome一直是小窗口 |
| height | 数值 | 表示新窗口的高度。不能小于100，IE FireFox好使 |
| left | 数值 | 表示新窗口的左坐标。不能是负值，Chrome一直是固定的窗口 |
| location | yes或no | 表示是否在浏览器窗口中显示地址栏。不同的浏览器默认值不同。如果设置为no，地址栏可能会隐藏，也可能会被禁用(IE默认禁用),Chrome默认禁用，如果使用yes，则变为普通标签,FireFox无论如何都不会被编辑 |
| menubar | yes或no | 表示是否在浏览器窗口中显示菜单栏，默认是no |
| resizable | yes或no | 表示是否可以通过拖动浏览器窗口的边框改变其大小，默认为no |
| scrollbars | yes或no | 表示是如果内容在窗口显示不下，是否允许滚动。默认值是no |
| toolbar | yes或no | 表示是否在在浏览器中显示工具栏，默认是no |
| top | 数值 | 表示新窗口的上坐标 |
| width | 数值 | 表示新窗口的宽度，不能小于100 |

window.open()方法会返回一个指向新窗口的引用。引用的对象与其它window对象大致相似，但我们可以对其进行更多控制。例如，大部分浏览器是不允许我们针对浏览器窗口调整大小或移动位置，但却允许我们针对通过window.open()创建的窗口调整大小或移动位置。

**var** lalala = window.open(**'https://www.zhihu.com'**,

**'\_blank'**,

**'height=400,width=500,top=10,left=10,resizable=yes'**);  
lalala.resizeTo(500, 500);  
lalala.moveTo(100, 100);  
lalala.close();

弹出窗口变比后，窗口的引用依然在，但除了像下面这样检测其closed属性之外，已经没有其它用处。

lalala.close();

console.log(lalala.closed); //true