# 3 基本概念

## 3.4 数据类型

ECMAscriprt五种（简单）基本数据类型

Undefined、Null、String、Number、Boolean

一种复杂数据类型:Object

### 3.4.typeof操作符

### 3.4.2Undefined类型

只有一个值undefined的数据类型

Var message

alert(message); //“undefined”

alert(age); // 错误

### 3.4.3 Null类型

第二个只有一个值null的数据类型

实际上，undefined值是派生自null值得，因此规定

alert(null == undefined);

但他们的用途完全不同

### 3.4.4Boolean类型

Boolean() true false

String 任何非空字符串 “”（空字符串）

Number 任何非零数字值（包括无穷大） 0和NaN

Object 任何对象 null

Undefined undefined

### 3.4.5 Number类型

#### 1、浮点数值

Var floatNum = .1; // 有效，但不推荐

Var floatNuim1 = 1. ; //解析为1

Var floatNum2 = 10.0; //解析为10

Var floatNum = 3.125e7; //等价于31250000

Var floatNum = 3e-7，//等价于0.0000003

{

Var float num1 = 0.1;

Var float num2 = 0.2;

Alert((num1+num2) == 0.3) //由于内存原因 0.1+0.2！=0.3

}

#### 2、数值范围

Infinity（正无穷）-Infinity(负无穷)

Infinity无法参与运算，isFinite()用来判断是否在最大最小之间

#### 3、NaN（非数值）

NaN与任何值的都不相等，包括NaN本身

Alert(NaN == NaN)；//false

任何不能被转换为数值的值都会返回true

Alert (isNaN(NaN)); //true

Alert(isNaN(10)); //false

Alert(isNaN(“10”))； //false

Alert(isNaN(“blue”)); //true

Alert(isNaN(true)); //true

#### 4、数值转换

可以把非数值转换为数值的函数：Number(),parseInt(),parseFloat()

Number（）用于任何数据类型

parseInt(),parseFloat()用于把字符串转换成数值

var num1 = Number（“Hello world!”）; //NaN

var num2 = Number(“ ”); //0

var num3 = Number(“000011” ); //11

var num4 = Number(true); //1

var num1 = parseInt(“1234blue”); //1234

var num2 = parseInt(“ ”); //NaN

var num3 = parseInt(“0xA”) //10

var num4 = parseInt(“22.5”) //22

var num5 = parseInt(“070”) //56

var num6 = parseInt(“70”) //70

var num7 = parseInt(“0xf”) //15

parseInt 指定基数

var num = parseInt(“10”,8) ;//8 按8进制解析

parseFloat

parseFloat与parseInt类似，区别在于

parseFloat第一个小数点有效，第二个无效

parseFloat始终忽略前导的零，只解析十进制

var num1 = parseFloat(“1234blue”); //1234(整数)

var num2 = parseFloat(“0xA”); //0

var num3 = parseFloat(“22.5”); //22.5

var num4 = parseFloat(“22.34.5”); //22.34

var num5 = parseFloat(“0908.5”); //908.5

var num6 = parseFloat(“3.125e7”); //31250000

### 3.4.6 String类型

Var text = “This is the letter sigma:\u03a3.”;

alert (text.length); //输出28

toString()

null和undefined没有toString()方法

String（）

如果有toStirng()方法，调用toString()方法

若值为null,返回‘null’,

若值为undefined,返回‘undefined’。

### 3.4.7 Object类型

Object的每个实例都具有下列属性和方法

constructor:保存着用于创建当前对象的函数。

hasOwnProperty(propertyName):用于检查给定的属性在当前对象实例中是否存在

isPrototypeOf(object):用于检查传入的对象是否是传入对象的原型

propertyIsEnumerable(propertyName):用于检查给定的属性是否能够使用for-in语句枚举。

toLocaleString(): 返回对象的字符串表示，该字符串与执行环境的地区对应。

toString():返回对象的字符串表示

valueOf();返回对象的字符串、数值活布尔值表示，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

1)递增和递减操作符

var num1=2；

var num2 = 20;

var num3=num1--+num2; //等于22

var num4 = --num1+num2; //等于21

var s1 = “2”;

var s2 = “z”;

var b= false;

var f =1.1;

var o={

valueof: function(){

return -1;

}

}

s1++; //值变成数值3

s2++; //值变成NaN;

b++; //值变成数值1；

f--; //值变成0.100000000000000009（由于浮点舍入错误导致）；

o--; //值变成数值-2；

1. 一元加和减操作符

var num=25;

]num = +num; //仍然是25

var s1 = “01”;

var s2 = “1.1”;

var s3 = “z”;

var b = “false”;

var f = 1.1;

var o = {

valueOf:function(){

return -1;

}

};

s1 = +s1;//值变成1

s2= +s2;//值变成2.1

s3= +s3;//值变成NaN

b = +b;//值变成0

f = +f;//值未变，任然是1.1；

o =+o;//值变成数值-1；

一元减操作符主要用于表示负数，如将1转成-1，用于数值时，该值变成负数。应用于非数值时，一元减操作符遵循与一元加操作符相同的规则，最后将得到的数值转换为负数。

### 3.5.2位操作符

位操作符并不直接操作64位的值。而是先将64位的值转换成32位的整数，然后进行操作，最后再将结果返回64位。

对于有符号的整数。32位中的前31位用于表示整数的值。第32位表示数值的符号，0表示正数，1标书负数。这个表示符号的位叫做符号位。正数以纯二进制格式存储。

负数同样以二进制码存储，但使用的格式是二进制补码。计算数值的二进制补码，需要经过三个步骤。

1. 求这个数值绝对值的二进制码（例如，要求-18的二进制补码，先求18的二进制码）；
2. 求二进制反码，即将0替换为1，将1替换为0；
3. 得到的二进制反码加1。
4. 按位非（NOT）~

var num1 = 25; //二进制00000000000000000000000000011001

var num2 = ~ num1; //二进制11111111111111111111111111100110

alert(num2); //-26

对25执行按位非的操作，得到了-26，验证了按位非的本质，即操作数的负值减1，因此，下面的代码也能得到相同的结果。

var num1 = 25;

var num2 = -num1-1;

alert(num2); //-26

由于按位非是在数值表示的最底层操作，速度更快。

1. 按位与（AND）&

按位与操作只在两个数值对应位都是1时才返回1，任何一位是0，结果都是0.

var result = 25 & 3；

alert(result); //1

25 = 0000 0000 0000 0000 0000 0000 0001 1001

3 = 0000 0000 0000 0000 0000 0000 0000 0011

AND 0000 0000 0000 0000 0000 0000 0000 0001

3)按位或（OR）|

有一个位是1的情况下就返回1，只有两个位都是0的情况下才返回0；

var result = 25 | 3;

alert(result); //27

4)按位异或（XOR）^

两个数值对应上只有一个1时才返回1，对应的两位都是1或0，返回0。

var result = 25 ^ 3;

alert(result) ; //26

5）左移<<

将数值的所有位向左移动指定的位数。

var oldValue = 2; //等于二进制的10；

var newValue = oldValue <<5; //等于二进制的1000000，十进制的64

左移不会影响操作数的符号位，换句话说，如果将-2向左移动5位，结果将是-64，而非64。

6）有符号的右移 >>

有符号的右移操作与左移操作恰好相反，即64向右移动5位，结果将变回2

var oldValue=64; //等于二进制的1000000

var newValue = oldValue>>5; //等于二进制的10，即十进制的2

7）无符号右移 >>>  
 对正数来说，无符号右移与有符号右移相同。对负数来说，无符号右移操作符会把负数的二进制码当成正数的二进制码，而且由于负数以其绝对值得二进制补码形式表示，会导致无符号右移后的结果非常大。

var oldValue = -64; //等于二进制的11111111111111111111111111000000；

var newValue = oldValue >>>5; //等于十进制的134217726

### 3.5.3 布尔操作符

1）逻辑非 ！

逻辑非操作符遵循下列规则：

对象：false

空字符串：true

非空字符串：false

数值0：true

非0数值：false

null:true

NaN:true

undefined:true

2)逻辑与 &&

逻辑与可以应用于任何类型的操作数,不仅仅是布尔值,有一个操作数不是布尔值,返回数就不一定是布尔值。

第一个操作数是对象，返回第二个操作数（验证未成功）

第二个操作数是对象，只有第一个操作数求职为true时才返回该对象

若两个操作数均为对象则返回第二个操作数。

有一个操作数是null，返回null

有一个操作数是NaN,返回NaN

有一个操作数是undefined,返回undefined。

逻辑与属于短路操作

let found =false; //为true时，结果会发生错误

let result = (found && someUndefinedVariable); //不会发生错误

alert (result); //会执行（‘false’）

1. 逻辑或 ||

第一个操作数是对象返回第一个操作数

第一个操作数求值结果为false,返回第二个操作数

两个操作数都是null.返回null

两个操作数都是NaN,返回NaN

都是undefined返回undefined。

短路操作符

let found = true;

let result = (found || someUndefinedVariable); //不会发生错误

alert(result); //会执行（‘true’）

### 3.5.4 乘性操作符 ECMAScript定义了三个乘性操作符：乘法、除法和求模。如果参与乘性计算的某操作数不是数值，后台会先使用Number()转型函数将其转换为数值。例如，空字符串被当作0，布尔值true被当作1。

1）乘法 \*

如果操作都是数值，执行常规的乘法计算。若乘积超过了ECMAScript数值的表示范围，则返回Infinity或-Infinity;

一个操作数是NaN，结果为NaN。

Infinity与0相乘，结果是NaN;

Infinity与非0数值相乘，结果是Infinity或-Infinity

Infinity与Infinity相乘，结果是Infinity

如果有一个操作数不是数值，则在后台调用Number()将其转换为数值，再应用上面的规则。

1. 除法 /

与乘法操作符类似

如果操作数都是数值，执行常规的除法计算

一个操作数是NaN,结果为NaN

Infinity被Infinity除，NaN；

零被零除，NaN;

非零的有限数被零除，Infinity或-Infinity

Infinity被任何非零数值除，Infinity或-Infinity。

有一个操作数不是数值，后台调用Number()后应用上述规则

1. 求模 %（余数）

操作数都是数值，常规的除法运算，返回余数

被除数无穷大除数有限大，NaN

被除数有限大，除数是0，NaN

Infinity被Infinity除，NaN

被除数有限大除数无穷大，结果是被除数。

被除数是0，结果是0

操作数不是数值，后台调用Number（）转换为数值后应用上述规则

**3.5.5 加性操作符**

1）加法 +

两个操作数都是数值，执行常规加法计算，根据下列规则返回结果

一个操作数是NaN，返回NaN

Infinity加Infinity，Infinity

-Infinity加-Infinity,-Infinity

Infinity 加-Infinity,NaN

+0加+0，+0；

-0+-0,-0;

+0+-0;+0;

如果两个操作数都是字符串，将第二个操作数与第一个操作数拼接；

如果只有一个操作数是字符串，则将另一个操作数转换为字符串后将两个字符串拼接。

如果有一个操作数是对象、数值或布尔值，调用toString()方法取得相应的字符串值，再应用上述规则，undefined和null调用String()函数取得字符串‘undefined’和‘null’；

2)减法 –

两个操作数都是数值，执行常规算数减法操作

Infinity减Infinity，NaN

-infinity减-infinity，NaN

infinity减-inFinity,Infinity

-Infinity减Infinity,-Infinity

+0减+0，+0；

+0减-0，-0；

-0减-0，-0；

若有操作数是字符串、布尔朱、null或undefined，则在后台调用Number()函数转换为数值，再依据前面的规则计算，若转换的结果为NaN,则结果为NaN.

若有一个操作数是对象，调用对象的valueOf（）方法取得表示该对象的 数值，如果得到NaN,则结果为NaN，如果没有valueOf方法，则调用toString()方法。

5-true //4

NaN-1 //NaN

5-3 //2

5-“” //5

5-‘2 //3

5-null //5

**3.5.6 关系操作符**

(<)(>)(<=)(>=)均返回一个布尔值

两个操作数都是数值，指向数值比较

都是字符串，比较两个字符串对应的字符编码值

一个操作数是数值，则将另一个操作数转换为一个数值，执行数值比较

一个操作数是对象，则调用这个对象的valueOf()方法，没有valueOf()调用toString()方法

一个操作数是布尔，先将其转换为数值，再进行比较

‘Brick’< ‘alphabet’; //true

‘23’<‘3’ //true

‘23’<3 //false

‘a’<3 //false,因为‘a’被转换成了NaN 任何操作数与NaN比较，结果都是false

NaN<3 //false

NaN>=3 //false

**3.5.7相等操作符**

1）相等和不相等 == ！=

如果有一个操作数是布尔值，在比较值钱先将其转换为数值false转换为0，true转换为1；

一个操作数是字符串，另一个是数值，先将字符串转换为数值

一个操作数是对象，另一个不是，调用valueOf（）后进行比较

null和undefined是相等的

比较相等性前，不能将null和undefined转换成其他任何值

有一个操作数是NaN，相等操作false,不等操作true,NaN不等于NaN

若两个都是对象，比较他们是不是同一个对象，如果两个操作都指向同一个对象，则返回true,否则返回false.

null==undefined //true

‘NaN’==NaN //false

5== NaN //false

NaN!=NaN //true

false==0 //true

true==1 //true

true==2 //false

undefined==0 // false

null==0 //false

‘5’==5 //true

2)全等和不全等 === ！==

只在两个操作数未经转换就相等的情况下返回true，

undefined==null //true

undefined === null //false

**3.5.8条件操作符**

variable = Boolean\_expression ? true\_value : false\_value;

返回最大值

var max = (num1 > num2 ) ? num1 : num2

**3.5.9 赋值操作符 =**

乘/赋值（\*=）；

除/赋值（/=）

模/赋值（%=）

加/赋值（+=）

减/赋值（-=）

左移/赋值（<<=）

有符号右移/赋值（>>=）

无符号右移/赋值（>>>=）

**3.5.10 逗号操作符**

var num1=1,num2=2,num3=3;

逗号操作符除声明多个变量外，还可用于赋值，赋值时，逗号操作符总会返回表达式中最后一项。

var num= （3,4,4,3,3,3，0）； //num的值为0；

## 3.6语句

**3.6.1 if语句**

**3.6.2 do-while语句**

**3.6.3 while语句**

**3.6.4 for语句**

使用while循环做不到的，使用for循环同样也做不到。

for(;;){ //无限循环

doSomething();

}

**3.6.5 for-in语句**

精准的迭代语句，可用来枚举对象的属性

示例：

for(var propName in window){

doxument.write(propName);

}

属性返回顺序因浏览器而异

**3.6.6 label语句**

在代码中添加标签，以便将来使用

label:statement

示例：

start:for(var i= 0;i<count;i++){

alert(i)

}

可以在将来由break或continue语句引用，一般要与for语句等循环语句配合使用。

**3.6.7 break和continue语句**

break和continue语句都可以与label语句联合使用，从而返回代码中特定位置。

var num = 0;

outermost:

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

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

if(i==5 && j==5){

break outermost;

}

num++;

}

}

alert (num) //55 break outermost退出到外层循环

**3.6.7with 语句**

with语句的作用是将代码的作用域设置到一个特定的对象中，语法如下

with(expression) statement;

定义with简化多次编写同一个对象

var qs = location.search.substring(1);

var hostname = location.hostname;

var url = location.href;

使用with

with(location){

var qs = search.substring(1);

var hostname = hostname;

var url = href;

}

with语句代码块内部，每个变量首先被定义为一个局部变量，若在局部环境中找不到该变量的定义，就会插下一步location对象中是否有同名的属性，如果发现同名属性，则以location对象属性的值作为变量的值。

严格模式下部允许使用with语句。大型应用程序下，不建议使用

**3.6.9 switch语句**

switch(expression){

case value:statement

break;

case value:statement

break;

default:statement

}

var num=25;

switch(true){

case num<0:

alert(“Less than 0”)

break;

case num>=0 && num<=10；

alert(“Between 0 and 20”);

break;

default:

alert(“More than 20”);

}

switch 语句在比较时使用的是全等操作符，不会发生类型转换

## 3.7 函数

**3.7.1理解参数**

function sayHi(){

alert(“Hello”+arguments[0]+“，”+argumerts[1]);

}

function doAdd(num1,num2){

arguments[1]=10;

alert(arguments[0]+num2);

}

重写第二个参数，会自动反映到对应的命名参数，内存空间独立，若只传入一个参数，不会反应，因为arguments对象的长度由传入的参数个数决定。

没有传递的命名参数被自动赋予undefined，严格模式下不能赋值

ECMAScript中所有参数传递的都是值，不可能通过引用传递参数。

**3.7.2 没有重载**

# 4 变量、作用域和内存问题

4.1 基本类型和引用类型的值

5种基本数据类型是按值访问的，因为可以操作保存在变量中的实际的值。引用类型的值是保存在内存中的对象。

### 4.1.1动态的属性

定义基本类型值和引用类型值得方式类似，对不同类型值可执行的操作大相径庭。引用类型的值，可以为其添加、改变、删除属性和方法，基本类型值不可以。

### 4.1.2 复制变量值

从一个变量向另一个变量复制基本类型时，两个变量互不影响。

复制引用类型的值时，两个变量将引用同一个对象。改变一个变量将会影响另一个。

var obj1 = new Object();

var obj2 = obj1;

obj1.name = ‘Nicholas’;

alert(obj2.name); //‘Nicholas

### 4.1.3 传递参数

### 4.1.4 检测类型

typeof

var s =“Nicholas”;

var I = 22;

var b = true;

var u;

var n = null;

var o = new Object();

alert(typeof s); //String

alert(typeof i); //number

alert(typeof b); //Boolean

alert(typeof u); //undefined

alert(typeof n); //object

alert(typeof o); //object

通常，我们并不想知道某个值是对象，而是想知道它是什么类型的对象。使用instanceof

result = variablr instanceof constructor

alert(person instanceof (Object); //变量person是Object吗？

colors instanceof Array; //是Array吗

pattern instanceof RegExp //是RegExp吗

所有引用类型的值都是Object的实例

基本类型不是Object

## 4.2 执行环境及作用域

we中，全局执行环境被认为是window,所有全局变量和函数都是作为window对象的属性和方法创建。

内部环境可以通过作用域链访问所有的外部环境，但外部环境不能访问内部环境中的任何变量和函数

### 4.2.1 延长作用域链

function builfUrl(){

var qs = “?debug = true”;

with(location){

var url = href + qs;

}

return url;

}

### 4.2.2 没有块级作用域

if （true）{

var color=”blue”;

}

alert(color); //‘blue’

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

dosomething(i)

}

alert(i); //10

1. 声明变量
2. 查询标识符

## 4.3 垃圾收集

垃圾收集器会按照固定的时间间隔（或代码执行中预定的收集时间），周期性的找出那些不再继续使用的变量，然后释放其占用的内存。

用于标识无用变量的策略通常有以下两种

4.3.1 标记清除

4.3.1 引用计数

### 4.3.3 性能问题

### 4.3.4 管理内存

## 4.4 小节

基本类型：undefined null Boolean Number String

基本类型值和引用类型值具有以下特点：

基本类型值在内存中占据固定大小的空间，被保存在栈内存中；

赋值基本类型值时，会创建值得副本；  
引用类型的值是对象，保存在堆内存中；

包含引用类型值得变量实际上包含的并不是对象本身，而是一个指向该对象的指针；

复制引用类型的值，复制的其实是指针，因此变量最终都指向同一个对象

确定一个值是那种基本类型可以使用typeof操作符，确定一个值是哪种引用类型可以使用instanceof操作符。

执行环境：

执行环境有全局执行环境和函数执行环境之分

每次进入一个新执行环境，都会创建一个用于搜索变量和函数的作用域链；

函数的局部环境不仅有权访问函数作用域中的变量，而且有权访问其父环境及全局环境

全局环境只能访问在全局中定义的变量和函数，不能访问局部环境中的任何数据；

变量的执行环境有助于确定该何时释放内存

JavaScript具有自动垃圾收集机制，开发人员不必关心内存分配和回收问题

垃圾收集：

离开作用域的值将被自动标记为可回收，因此将在垃圾收集期间被删除。

“标记清除”是目前主流的垃圾收集算法，这种算法的思

想是给当前不使用的值加上标记，然后再回收其内存。

另一种垃圾收集算法是“引用计数”，跟踪记录所有值被引用的次数。JavaScript引擎目前都不再使用这种算法。但在IE中访问非原生JavaScript对象时，这种算法仍然可能会导致问题。

当代码中存在循环引用现象时，“引用计数”算法就会导致问题

解除变量的引用不仅有助于消除循环引用现象，对垃圾收集也有好处。为确保有效回收内存，应及时解除不再使用的全局对象、全局对象属性以及循环引用变量的引用。

# 5 引用类型

## 5.1 Object类型

两种创建Object实例的方式:

new

var person = new Object();

person.name = “Nicholas”;

person.age=29;

对象字面量

var person = {

name:“Nicholas”,

age:29,

};

属性

person.name=”Nicholas”

person[“first name”] = “Tom”;

## 5.2 Array类型

var colors = new Array(20);

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

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

var names = new Array(“Greg”); //创建一个包含1项，即字符串“Greg”的数组

可省略new

var colors = Array(3); //创建一个包含3项的数组

var names = Array(“Greg”); //创建一个包含1项，即字符串“Greg”的数组

数组字面量

var colors = [“red”，“blue”,“green”]; //创建一哥包含3个字符串的数组

var names = []; //创建一个空数组

var values = [1,2,]; //尽量避免，这样会创建一个包含2或3项的数组

var options= [,,,,,]; // 尽量避免，这样会创建一个包含5或6项的数组

数组的lenth不是只读的

var colors = [‘red”,“blue”,“green”]； //创建一个包含3个字符串的数组

colors.length = 2;

alert(colors[2]); //undefined

### 5.2.1检测数组

if（Array.isArray(value)）{

//对数组执行某些操作

}

### 5.2.2转换方法

var colors = [“red”,“blue”, “green”]; //创建一个包含3个字符串的数组

alert(colors.toString()); //red,blue,green

alert(colors.valueof()); //red,blue,green

alert(colors); //red,blue,green

alert(colors.join(“||”)); //red//green//blue

### 5.2.3 栈方法

LIFO（Last-In-First-Out，后进先出）的数据结构

push() pop()

var colors = [“red”,“blue”];

colors.push（“brown”）； /./添加另一项

colors[3] = “black”; //添加一项

alert(colors.length); //4

var item = colors.pop(); //取得最后一项

alert（item）; //“black

alert(colors.length); //3

### 5.2.4 队列方法

FIFO(First-In-First\_out)先进先出

push() shift() //后端添加项，前端删除项

var colors = new Array(); //创建一个数组

var count = colors.push(“red”,“green”)； //推入两项

alert(count); //3

count = colors.push(“black”); //推入另一项

alert(count); //3

var item = colors.shift(); //取得第一项

alert(item); //“red”

alert(colors.length); //2

unshift() pop() //前端添加项 后端删除项

var colors = new Array(); // 创建一个数组

var count = colors.unshift(“red”,“green”); //推入两项

alert(count); //2

count = colors.unshift(“black”); //推入另一项

alert(count); //“3”

var item = colors.pop(); //取得最后一项

alert(item); //“green”

alert(colors.length); //2

### 5.2.5 重排序方法

reverse() sort()

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

values.reverse();

alert(value); //4,3,2,1

sort()方法会调用每个数组项的toString()转型方法，然后比较得到的字符串

var values = [0,1,5,10,15];

values.sort();

alert(values); //0,1,10,15,5

sort()方法可以接收一个比较函数作为参数，以便指定谁前谁后

以下是一个简单的比较函数：

function compare（value1,value2）{

if(value1<value2){

return -1;

} else if(value1>value2){

return 1;

} else{

return 0;

}

}

这个比较参数适用于大读数数据类型，只要将其作为参数传递给sort（）方法即可

var values = [0,1,5,10,15];

values.sort(compare);

alert(values); //0,1,5,10,15

数值类型或者valueof方法会返回数值类型的对象类型，可以使用

function compare(value1,value2){

return value2-value1;

}

### 5.2.6 操作方法

concat() 先创建当前数组的一个副本，然后将接收到的参数添加到这个副本的末尾，最后返回新构建的数组。

var colors =[“red”,“green”,“blue”]；

var colors2 = colors.concat(“yellow”,[“black”,“brown”]

)

alert(colors); //red,green,blue

alert(colors2); //red,green,blue,yellow,black,brown

slice()能够基于当前数组中的一个或多个项创建一个新数组。slice方法如果有两个参数，返回包括前不包括后的项。slice方法不影响原始数组。

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

var colors2 = colors.slice(1);

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

alert(colors2); //green,blue,yellow,purple

alert(colors3); //green,blue,yellow

splice()

删除：可以删除任意数量的项，只需指定2个参数，要删除的第一项的位置和要删除的项数

splice(0,2)会删除数组中的前两项

插入：向指定位置插入任意数量的值，需提供3个参数：起始位置、0（要删除的项数）和要插入的项。splice(2,0,“red”,“green”)会从当前数组的位置2开始插入字符串“red”和“green”.

替换：可以向指定位置插入任意数量的项，且同试删除任意数量的项，需指定3个参数：起始位置，要删除的项数和要插入的任意数量的项。splice(2,1,“red”,“green”)会删除当前数组2的项，然后再从2开始插入字符串“red”和“green”

splice方法始终都返回一个数组，改数组中包含从原始数组中删除的项。

var colors = [“red”,“green”,“blue”];

var ewmoved = colors.splice(0,1); //删除第一项

alert(colors); //green,blue

alert(removed); //red,返回的数组中只包含一项

removed = colors.splice(1,1,“red”，“purple”); //插入两项，删除一项

alert(colors); //green,red,purple,orange,blue

alert(removed); //yellow,返回的数组中只包含一项

### 5.2.7 位置方法

indexOf() lastIndexOf() 这两个方法都接收两个参数，要查找的项和（可选的）表示查找起点位置的索引。indexOf()从数组的开头向后查找，lastIndexOf()从数组的末尾向前查找

查找的项必须严格相等===

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

alert(numbers.indexOf(4)); //3

alert(numbers.lastINdexOf(4)); //5

alert(numbers.indexOf(4,4)); //5

alert(numbers.lastIndexOf(4,4)); //3

var person = {name:“Nicholas”};

var people = [{name:“Nicholas”}];

var morePeople = [person];

alert(people.indexOf(person)); //-1

alert(morePeople.indexOf(person)); //0

### 5.2.8 迭代方法

every（）：对数组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

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

forEach():对数组中的每一项运行给定函数，这个方法没有返回值

map():对数组中的每一项运行指定函数，返回每次调用的结果组成的数组

sone(): 对数组中的每一项云行指定函数,如果该函数对任一项返回true，则返回true.

以上方法都不会修改数组中的包含的值。

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

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

return (item>2);

})

alert(everyResult); //false

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

return (item>2);

});

alert(someResult); //true

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

return (item>2);

})

alert(filterResult); //[3,4,5,4,3]

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

return item\*2;

})

alert(mapResult); //[2,4,6,8,10,8,6,4,2]

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

//执行某些操作

})

### 5.2.9 归并方法

两个归并数组的方法：reduce()和reduceRight(),迭代数组的所有项，然后构建一个最终返回的值。resuce()方法从数组的第一项开始，逐个到最后，reduceRight()从数组的最后一项开始，向前遍历到第一项。

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

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

return prev+cur;

})

alert(sum); //15

第一次执行回调函数，prev是1，cur是2，第二次，prev是3（1+2）。reduceRight（）的作用类似，只不过方向相反。

## 5.3 Date类型

### 5.3.1继承的方法

toLocalString(),toString(),valueOf()

### 5.3.2 日期格式化方法

### 5.3.3 日期、时间组件方法

## 5.4 RegExp类型

var expression = /pattern/flags;

支持以下3个标志：

g:全局模式，被应用于所有字符串，非发现第一个匹配项时立即停止

i:不区分大小写模式

m:多行模式，在达到一行文本末尾时还会继续查找下一行中是否存在你与模式匹配的项。

正则表达式字面量：

re = /cat/g;

构造函数创建：

new RegExp(“cat”,“g”);

使用正则表达式字面量和使用RegExp构造函数创建的正则表达式不一样，正则表达式字面量始终会共享同一个RegExp实例，而使用构造函数创建的每一个RegExp实例都是新实例。

var re = null,

i;

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

re = /cat/g;

re.test(“catastrophe”);

};

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

re = new RegExp(“cat”,“g”);

re.test(“catastrophe”);

}

第一个循环中，即使是循环体重指定的，但实际上只为/cat/创建了一个RegExp实例，由于实例属性不会充值，所以在循环中再次调用test()方法会失败，这是因为第一个调用test()找到了“cat”,但第二次调用是从索引为3的字符（上一次匹配的末尾）开始的，所以就找不到它了。由于会测试到字符串末尾，下次调用test()就又从头开始了。

第二个循环使用构造函数，因为每次调用都会创建一个新的RegExp实例，所以每次调用test()都会返回true。

### 5.4.1 RegExp实例属性

global:布尔值，表示是否设置了g标志

ignoreCase:布尔值，表示是否设置可i标志

lastIndex:整数，表示开始搜索下一个匹配项的字符位置，从0算起。

multiline:布尔值，表示是否设置了m标志。

source:正则表达式的字符串表示，按照字面量形式而非传入构造函数中的字符串模式返回。

### 5.4.2 RegExp实例方法

var text = “cat, bat ,sat, fat”;

var pattern = /.at/;

var matches = pattern1.exec(text); //0

alert(matches[0]); //cat

alert(pattern1.lastIndex); //0

matches = pattern1.exec(text);

alert(matches.index); //0

alert(matches[0]); //cat

alert(pattern1.lastIndex); //0

var pattern2 = /.at/g;

var matches = pattern2.exec(text);

alert(matches.index); //0

alert(matches[0]); //cat

alert(pattern2.lastIndex); //3

matches = pattern2.exec(text);

alert(matches.index); //5

alert(matches[0]); //bat

alert(pattern2.lastIndex); //8

test() 接受一个字符串参数，匹配时返回true,否则返回false,

var test = “000-00-0000”;

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

if(pattern,test(text)){

alert(“The pattern was matched.”);

}

### 5.4.3 RegExp构造函数属性

。。。。。。。。。。

### 5.4.4 模式的局限性

## 5.5 Function类型

function sum(num1,num2){

return num1+num2;

}

var sum = function(num1,num2){

return num1+num2;

};

### 5.5.1 没有重载（深入理解）

function addSomeNumber(num){

retrun num+100;

}

function addSomeNumber(num){

return num+200;

}

var result = addSomeNumber(100); //300

### 5.5.2 函数声明与函数表达式

解析器率先读取函数声明，使其在执行任何代码之前可以访问

必须等到解析器执行到函数表达式的代码行，函数表达式才会被真正执行。

alert (sum(10,10));

function(sum1,sum2){

return num1+num2;

}

alert(sum(10,10));

var sum = function(sum1,sum2){

return num1+num2;

}; //unexpected identifier错误

### 5.5.3 作为值的函数

function callSomeFunction(someFunction,someArgument){

return someFunction(someArgument);

}

function add10(num){

return num+10;

}

var result1 = callSomeFunction(add10,10);

alert(result1); //20

function getGreeting(name){

return “Hello,”+name;

}

var result2 = callSomeFunction(getGreeting,“Nicholas”);

alert(result2) ; //”Hello,Nicholas”;

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:“Zachary”,age:20},{name:“Nicholas”,age:29}]

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

alert(data[0].name); //Nicholas

data.sort(createComparisonFunction(“age”))

alert(data[0].name); “Zachary”

### 5.5.4 函数内部属性

在函数内部，有两个特殊对象：arguments和this，arguments有一个callee的属性，该属性是一个指针，指向拥有这个arguments对象的函数。

function factorial(num){

if(num <=1){

return 1;

}else{

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

}

}

var trueFactorial = factorial;

factorial = function(){

return 0;

}

alert(trueFactoral(5)); //120

alert(factorial(5)); //0

函数内部的另一个特殊对象时this,this引用的是函数数据以执行的环境对象。

window.color=“red”;

var o = {color:“blue”};

function sayColor(){

alert(this.color);

}

sayColor(); //“red”

o.sayColor = sayColor;

o.sayColor(); //“blue”

调用当前函数的函数的引用：caller,如果是在全局作用域中调用当前函数，值为null。

function outer(){

inner();

}

function inner(){

alert(inner.caller);

}

outer(); //显示outer()函数的源代码

因为outer()调用了inter(),所以inner.caller指向outer（）；

function outer(){

inner();

}

function inner(){

alert(arguments.callee.caller);

}

outer();

### 5.5.5 函数属性和方法

函数是对象，因此函数也有属性和方法，每个函数包含两个属性：length和prototype

function sayName(name){

alert(name);

}

function sum(num1,num2){

return num1+num2;

}

function sayHi(){

alert(“hi”);

}

alert(sayName.length); //1

alert(sum.length); //2

alert(sayHi.length); //0

每个函数都包含两个非继承而来的方法：apply()和call();

在特定的作用域中调用函数，设置函数体内this对象的值。

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是Array的实例，也可以是arguments对象。

function sum(num1,num2){

return num1+num2;

}

function callSum1(num1,num2){

return sum.apply(this,arguments);

}

function callSum2(num1,num2){

return sum.apply(this,[num1,num2]);

}

alert(callSum1(10,10)); //20

alert(callSum2(10,10)); //20

call()方法与applay()方法的作用相同，区别在于接收参数的方式不同，对于call()方法而言，第一个参数this没有变化，其余参数直接传递给函数，也就是说使用call()方法时，传递的参数必须逐个列举出来。

function sum(num1,num2){

return num1+num2;

}

function callSum(num1.num2){

return sum.call(this,num1,num2);

}

alert(callSum(10,10)); //20

apply（）和call()除传递参数外，真正强大的地方是能够扩充函数的作用域。

window.color = “red”;

var o={color.red};

function sayColor(){

alert(this.color);

}

sayColor(); //red

sayColor.call(this); //red

sayColor.call(window); //red

sayColor.call(o); //blue

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

window.color = “red”;

var o = {color:“blue”};

function sayColor(){

alert(this.color);

}

var objectSayColor = sayColor.bind(o);

objectSayColor(); //blue;

## 5.6 基本包装类型

3个特殊的引用类型：Boolean,Number,String.

实际上，每当读取一个基本类型的时候，后台就会创建一个对应的基本包装类型的对象，从而让我们能够调用一些方法来操作这些数据。

var s1 = “some text”；

var s2 = s1.substring(2);

基本类型不是对象，理论上不应该有方法，后台自动完成了一系列的处理。

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  
   var s1 = new String(“some text”);

var s2 = s1.substring(2);

s1 = null;

这样，基本的字符串就跟对象一样了，上述步骤也适用于Boolean,Number对应的布尔值和数字值

引用类型与基本包装类型的区别在于对象的生存期。new操作符创建的引用类型的实例，在执行流离开当前作用域之前都一直保存早内存中。自动创建的基本包装类型只存在于一行代码执行的瞬间，然后立即被销毁。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”;

alert(s1.color); //undefined;

Object函数也像工厂方法一样，根据传入值得类型返回相应基本包装类型的实例。

var obj = new Object（“some text”）;

alert(obj instanceof String); //true

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 number = Number(value); //转型函数

alert(typeof number); //“number”

var obj = new Number(value); //构造函数

alert（typeof obj）; //object

### 5.6.1 Boolean 类型

Boolean类型的实例重写了valueOf()，返回基本类型值true,false,重写了toString()方法，返回字符串“true”,”false”

Boolean用处不大，因为经常会造成人们的误解

var falseObject = new Boolean(false);

var result = falseObject&&true；

alert（result） //true

var fasleValue = fasle;

result = faseValue&& true;

alert(result); //false

alert(typeof falseObject); //object

alert(typeof falseValue); //Boolean

alert(falseObject instanceof Boolean); //true

alert(falseValue instanceof Boolean); //false

### 5.6.2 Number类型

为toString()方法传递一个表示基数的参数，定义返回几进制数值的字符串形式。

var num=10；

alert(num.toString()); //‘10’

alert(num.toString(2)); //‘1010’

alert(num.toString(8)); //‘12’

toFixed()方法按照指定的小数位返回数值的字符串表示

var num= 10；

alert(num.toFixed(2)); //’10.00’

var num =10.005;

alert(num.toFixed(2)); //’10.01’;

格式化数值方法toExponential();该方法返回以指数表示法（也称e表示法），表示的数值的字符串形式。

var num =10;

alert(num.toExponential(1)); //‘1.0e+1’

得到表示某个数值的最合适的格式，使用toPrecision（）方法，toPrecision可能会返回固定大小(fixed）格式,也可能返回指数（exponential）格式。这个方法接收一个参数，即表示数值的所有数字的位数（不包括指数部分）。

var num= 99；

alert（num.toPrecision(1)）; //“1e+2”

alert(num.toPrecision(2)); //“99”

alert（num.toPrecison(3)）; //“99.0”

var numberObject = new Number(10);

var numberValue=10;

alert(typeOf numberObject); //“object”

alert(typeOf numberValuye); //number

alert(numberObject instanceof Number); //true

alert(numberValue insatnceof Number); //false

### 5.6.3 String类型

var stringObject = new String(“hello word”);

String 类型的每个实例都有一个length方法。

var stringValue = “hello word”;

alert(stringValue.length); //“11”

1. 字符方法: charAt 和charCodeAt();

charAt()以单字符串的形式返回给定位置的那个字符

var stringValue = “hello world”;

alert(stringValue.charAt(1)); //“e”

alert(stringValue.charCodeAt(1)); //输出“101” 得到字符编码

alert（stringValue[1]）; //“e”

2) 字符串操作方法

var stringValue = “hello”；

var result = stringValue.concat(“world”)；

alert（result）; //“hello world”;

alert(stringValue); //“hello”

var stringValue = “hello”;

var result = stringValue.concat(“world”，“！”)；

alert（result）; //“hello world!”

alert(stringValue); //“hello”

不过实践中更多的是使用“+”；

基于字符串创建新字符串的方法:slice(),substr()和substring();三个方法都返回被操作字符串的一个子字符串，而且都接受一或两个参数，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到哪里结束。slice()和substring()的第二个参数指定的是子字符串最后一个字符后面的位置。substr()的第二个参数指定的是返回的字符个数。

var stringValue = “hello world”;

alert(stringValue.slice(3)); //“lo world”

alert(stringValue.substring(3)); //“lo world”

alert(stringValue.subdtr(3)); //“lo world”

alert(stringValue.slice(3,7)) ; //“lo w”;

alert(stringVallue.subString(3,7)); //“lo w”;

alert(stringValue.substr(3,7)); //“lo wowl”

1. 字符串位置方法

indexOf()和lastIndexOf()

var stringValue =“hello world”;

alert(stringValue.indexOf(“o”)); //4

alert(stringValue.lastIndexOf(“o”)); //7

var stringValue = “hello world”;

alert(stringValue.indexOf(“o”,6)); //7

alert(stringValue.lastIndexOf(“o”,6)); //4

var stringValue = “Lorem ipsum dolor sit amet,consectur adipsicing elit”;

var position = new Array();

var pos = stringValue.indexOf(“e”);

while(pos >-1){

positon.push(pos);

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

}

alert(position); //“3,24,32,35,52”

1. trim()方法

创建字符串的副本，删除前置及后缀的所有空格

var stringValue= “ hello world ”;

var trimmedStringVale = stringValue.trim();

alert(stringValue); //” hello world ”

alert(trimmedStringValue); //“hello world”

1. 字符串大小写装换方法

toLowerCase() toLocalLowerCase()、toUpperCase()、LocalUpperCase()

1. 字符串的模式匹配方法

match 本质上与调用RegExp的exec()方法相同。

var text = “cat,bat,sat,fat”;

var pattern = /.at/;

//与pattern.exec(text)相同

var matches = text.match(pattern);

alert(matches.index); //0

alert（matches[0]）; //“cat”

alert(pattern.lastIndex)； //0

search()方法返回字符串中第一个匹配项的索引，如果没有找到匹配项，则返回-1.

var text = “cat,bat,sat,fat”;

var pos = text.search(/at/);

alert(pos); //1

replace方法接收两个参数：第一个参数可以是一个RegExp对象或者一个字符串（这个字符串不会被转换成正则表达式），第二个参数可以试一个字符串或一个函数。如果第一个是字符串，则只会替换第一个字符串，要想替换所有自妇产，只能提供一个正则表达式，而且指定全局标志（g）

var text = “cat,bat,sat,fat”;

var result = text.replace(“at”,‘ond’);

alert(result); //“cond,bat,sat,fat”;

result = text.replace(/at/g,“ond”);

alert(result); //“cond,bond,sond,fond”;

function htmlEscape(text){

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

switch(match){

case “<”:

return “&lt;”;

case “>”:

return “&gt;”;

case “&”:

return “&amp;”;

case “\””;

return “&quot;”;

}

})

}

alert(htmlEscape(“<p class=\”greeting\”>Hello world!</p>”));

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

split()

基于指定的分隔符将一哥字符串分割成多个子字符串，并将结果放在一个数组中，分隔符可以是字符串也可以是一个RegExp对象（这个方法不会把字符串看成正则表达式）。split方法可以接受可选的第二个参数，用于指定数组的大小。

var colorText = “red,blue,green,yellow”

var colors1 = colorText.split(“,”); //[“red”,”blue”,”green”,”yellow”]

var colors2 = colorText.split(“,”,2); //[“red”,”blue”]

var colors3 = colorText.split(/[^\,]+/); //[“”,”,”,”,”,”,”,””];

1. localeCompare()方法

比较两个字符串

字符串在字母表中排在字符串参数之前，返回一个负数。（大多数情况下是-1）

字符串等于字符串参数，返回0

字符串在参数之后，返回正数（大多数情况下是1）

var stringValue= “yellow”;

alert(stringValue.localeCompare(“brick”)); //1

alert(stringValue.localeCompare(“yellow”)); //0

alert(stringValue.localeCompare(“zoo”))； //-1;

实际实现时，最好与0比较进行判断

8）fromCharCode()

接收一或多个字符编码，然后将他们转换成一个字符串，本质上来看，与charCodeAt()执行相反的操作。

alert(String.fromCharCode(10 4,101,108,108,111)); //”hello”

9） HTML方法

## 5.7 单体内置对象

### 5.7.1 Global对象

1、URI编码方法

URI方法encodeURI()、encodeURIComponent()、decodeURI()和decodeURUComponent()用于替代已经被废弃的escape()和unescape()方法。

encodeURI（）不会对本身属于URI的特殊字符进行编码。例如冒号、正斜杠、问号和井字号；而encodeURIComponent()则会对它发现的任何非标准字符进行编码。

2、eval()方法

通过eval（）执行的代码被认为是包含该次调用的执行环境的一部分，被执行的代码具有与该执行环境相同的作用域链。

3 、Global对象的属性

4、window对象

### 5.7.2 Math对象

1、Math对象的属性

2、min（）和max（）方法

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

alert(max); //54

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

alert(min); //3

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

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

3、舍入方法，将小数值舍入为整数的几个方法：Math.ceil()、Math,floor()、Math.round()

Math.ceil()执行向上舍入，将数值向上舍入最接近的整数

Math,floor()向下舍入，将数值向下舍入为最接近的整数

Mah.round()标准舍入，四舍五入为最接近的整数

alert（Math.ceil(25.1)）; //26

alert(Math.round(25.5)); //26

alert(Math.floor((25.9)); //25

4、random()方法

Math.random()方法返回大于等于0小于1的一个随机数。

从某个整数范围内随机选择一个值：

值 = Math.floor(Math.random() \* 可能值的总数 + 第一个可能的值)；

function selectFrom(lowerValue,upperValue){

var choices = upperValue -lowerValue +1；

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

}

var colors = [“red”,“green”,“blue”,“yellow”,“black”,“purple”,“brown”];

var color = colors[selectFrom(0,solors.length-1)];

alert(color); //可能是数组中包含的任何一个字符串。

5、其他方法

## 5.8 小节

# 6 面向对象的程序设计

## 6.1 理解对象

var person = new Object();

person,name = “Nicholas”;

person.age = 29;

person.job = “Software Engineer”;

person.sayName = function(){‘

alert(this.name);

};

var person={

name:“Nicholas”,

age:29,

job:“Software Engineer”,

sayName:function(){

alert(this.name);

}

}

6.1.1 属性类型

内部才用的特性[[Enumerrable]]

两种属性：数据属性和访问器属性

1. 数据属性

[[Configurable]]:能否通过delete删除属性从而定义属性，修改属性的特性，把属性修改为访问器属性。默认为true。

[[Enumerable]]:能否通过for-in循环返回属性。默认true

[[Writable]]:能否修改属性的值，默认true

[[Value]]:包含这个属性的数据值。默认值是undefined。

var person = {

name:“Nicholas”//[[Value]]特性被设置为“Nicholas”

};

要修改属性默认的特性，必须使用Object.defineProperty()方法，接收三个参：属性所在的对象、属性的名字和一个描述符对象（configurable、enumerable、writeble、Value）。

var person = {};

Object.defineProperty(person,“name”,{

writeble:false,

value:“Nicholas”

})；

alert（person.name）; //“Nicholas”

person.name = “Greg”;

alert(person.name); //“Nicholas”

var person={};

Object.defineProperty(person,“name”{

configurable:false,

value:“Nicholas”

})；

alert（person.name）; //“Nicholas”

delete person.name；

alert(person.name); //“Nicholas”

一旦把属性定义为不可配置的，就不能再把它变回可配置了。在调用Object.defineProperty()方法时，如果不指定，configurable、enumerable和writeble特性的默认值都是false。

1. 访问器属性

访问器属性有如下4个特性：

[[Configureable]]:能否通过delete删除属性，修改属性的特性，能够把属性修改为数据属性。对于直接在对象上定义的属性，这个特性的默认值为true

[[Enumerable]]:表示能否通过for-in 循环返回属性，直接在对象上定义的属性，默认为true

[[Get]]:读取属性是调用的函数，默认undefined

[[Set]]:写入属性时调用的函数，默认为undefined

访问器属性不能直接定义，必须使用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;

this.edition += newValue – 2004;

}

}

});

book.year = 2005;

alert(book.edition); //2

不一定非要同时指定getter和setter，只指定getter意味着属性是不能写，只指定setter意味着不能读。

### 6.1.2 定义多个属性

Object.defineProperties(),接收两个对象参数，第一个是要添加和修改其属性的对象，第二个对象时属性。

var book = {；

Object.d额fineProperties(book,{

\_year:{

value:2004

}

edition:{

value:1

}

year:{

get:function(){

return this.\_year;

},

set(newValue){

if(newValu>2004){

this.\_year = newValue;

this.edition + = newValue-2004;

}

}

}

}

以上代码定义了两个数据属性和一个访问器属性

### 6.1.3 读取属性的特性

Object.getOwnPropertyDescription()

## 6.2 创建对象

使用Object构造函数或对象字面量创建单个对象的缺点是：使用同一个接口创建很多对象时会产生大量的重复代码。

### 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(){

alert(this.name);

};

return o;

}

var person1 = createPerson(“Nicholas”,29,”Software Engineer”);

var person2 = createPerson(“Greg”,27,”Doctor”);

工厂模式的缺点是：无法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(){

alert(this.name);

};

}

var person1 = new Person(“Nicholas”,29,”Software Engineer”);

var person2 = new Person(“Greg”,27,”Doctor”);

Person与createPerson的区别：

没有显示的创建对象

直接将属性和方法赋给了this对象

没有return 语句

要创建Person的新实例，必须使用new操作符，以这种方式调用构造函数会经历以下四个步骤：

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

alert（person1.constructor ==Person）; //true

alert(person.constructor == Person);//true

alert(person1 instanceof Objet); //true

alert(person1.instanceOf Person); //true

alert(person2 instanceOf Object); //true

alert(person2.instanceOf Person); //true

1. 将构造函数当作函数

任何函数通过new操作符调用，就可以作为构造函数，不通过new操作符调用就是普通函数。

//当作构造函数使用

var person = new Person(“Nicholas”,29,”Software Engineer”);

person.sayName(); //“Nicholas”

//作为普通函数调用

Person（“Greg”，27，“Doctor”）; //添加到window

window.sayName(); //”Greg”

//在另一个对象的作用域中调用

var o = new Object();

Person.call(o,”Kristen”,25,“Nurse”);

o.sayName(); //“Kristen”

2、构造函数的饿问题

问题在于每个方法都要遭每个实例上重新创建。

function Person(name,age,job){

this.name=name;

this.age = age;

this.job= job;

this.sayName = new Function(“alert(this.name)”); //与声明函数在逻辑上是等价的

}

alert(person1.sayName==person2.sayName); //false

可通过把函数定义转移到构造函数外部来解决

function Person(name，age,job){

this.name = name;

this.age = age;

this.job= job;

this.sayName = sayName;

}

function sayName(){

alert(this.name);

}

var person1 = new Person(“Nicholas”,29,”Software Engineer”);

var person2 = new Person(“Greg”,27,”Doctor”);

存在问题：sayName属性被设置成全局函数，但是这个函数只能被某个对象调用，若有多个方法，则需要定义多个全局函数，缺乏封装性，这个问题可以使用原型链模式解决。

### 6.2.3 原型模式

function Person(){

}

Person.prototype.name= “Nicholas”;

Person.prototype.age = 29;

Person.prototype.job = “Software Engineer”;

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

person1.sayName(); //”Nicholas”

var person2 = new Person();

person2.sayName(); //“Nicholas

alert(person1.sayName = person2.sayName); //true

1. 理解原型对象

alert(Person.prototype.isPrototypeOf(person1)); //true

alert(Person.prototype.isPrototypeOf(person2)); //true

Object.getPrototypeOf(); //返回[[Prototype]]的值

alert（Object.getPrototypeOf(person1)==Person.prototype）; //true

alert(Object.getPrototypeOf[person1.name]); //”Nicholas”

可以通过对象实例访问保存在原型中的值，但是不能通过对象实例重写原型中的值，在实例中添加与实例原型中属性同名的属性，将屏蔽原型中属性。

function Person(){

}

Person.prototype.name = “Nicholas”;

Person.prototype.age=29;

Person.prototype.job=“Software Engineer”;

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

var person2 = new Person();

person1.name = “Greg”;

alert(person1.name); //“Greg”----来自实例

alert(person2.name); //“Nicholas”---来自原型

delete person1.name;

alert(person1.name); //“Nicholas----来自原型

hasOwnPrototype()方法只在给定属性存在于对象实例中时，才会返回true

function Person(){

}

Person.prototype.name = “Nicholas”;

Person.prototype.age=29;

Person.prototype.job=“Software Engineer”;

Person.protptype.sayName = function(){

aslert(this.name);

};

var person1 = new Person();

var person2 = new Person();

alert(person1.hasOwnProperty(“name”)); //false

person1.name=“Greg”;

alert(person1.name); //“Greg”-----来自实例

alert(person1.hasOwnProperty(“name”)); //false

delete person2.name;

alert(person1.name); //“Nicholas”---来自原型

alert(person1.hasOwnProperty(“name”)); //false

1. 原型与in操作符

单独使用和在for-in循环中使用。无论属性存在于实例还是原型中，in操作符会返回true

function Person(){

}

Person.prototype.name = “Nicholas”;

Person.prototype.age=29;

Person.prototype.job=“Software Engineer”;

Person.prototype.sayName = function(){

alert(this.name);

};

var person1 = new Person();

var person2 = new Person();

alert(person1.hasOwnProperty(“name”)); //false

alert(“name” in person1); //true

person1.name = “Greg”;

alert(person1.name); //”Greg”---来自实例

alert(person1.hasOwnProperty(“name”)); //true

alert(“name” in person1); //true

alert(person2.name); //“Nicholas”----来自原型

alert(person2.hasOwnProperty(“name”)); //false

alert(“name” in person2); //true

delete person1.name;

alert(person1.name); //“Nicholas”---来自原型

alert(person1.hasOwnProperty(“name”)); //false

alert(“name” in person1); //true

判断属性是否是原型中的属性

function hasPrototypeProperty(object,name){

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

}

function Person(){}

Person.prototype.name = “Nicholas”;

Person.prototype.age=29;

Person.prototype.job=“Software Engineer”;

Person.prototype.sayName = function(){

alert(this.name);

};

var person = new Person();

alert(hasPrototypeProperty(person,“name”)); //true

person.name=”Greg”;

alert(hasPrototypeProperty(person,“name”)); //fasle

for in 返回存在于实例、原型以及屏蔽了原型中[[Enumerable]]为false的属性。

IE早期版本存在bug

var o = {

toString:function(){

return “My Object”;

}

};

for(var prop in o){

if(prop == “toString”){

alert(“Found toString”)； //在IE中不会显示

}

}

要取得对象上所有可枚举的实例属性，可使用Object.keys()方法；

function Person(){

}

Person.prototype.name = “Nicholas”;

Person.prototype.age=29;

Person.prototype.job=“Software Engineer”;

Person.prototype.sayName = function(){

alert(this.name);

};

var keys = Object.keys(Person.prototype);

alert(keys); //“name,age,job,sayName”

var keys = Object.keys(Person.prototype);

alert(keys); //“name,age,job,sayName”

var p1 = new Person();

p1.name = “Rob”;

p1.age=31;

var plkeays = Object.keys(p1);

alert(plkeys); //“name,age”

获得所有实例属性，无论是否可枚举

var keys = Object.getOwnPropertyNames(Person.prototype);

alert(keys); //“constructor，name,age,job,name”

1. 更简单的原型语法

function Person(){

}

Person.prototype = {

name:“Nicholas”,

age:29,

job:“Software Engineer”,

sayName:function(){

alert(this.name);

}

};

var friend = new Person();

alert(friend instanceof Object); //true

alert(friend instanceof Person); // true

alert(friend.constructor == Person); //false

alert(friend.constructor == Object); //true

最终结果相同，但是constructor属性不再指向Person了，本质上重写了默认的prototype对象。

可以将constructor设置回适当的值

function Person(){

}

Person.prototype = {

constructor:Person,,

age:29,

job:“Software Engineer”,

sayName:function(){

alert(this.name);

}

}

但是上述方法重置constructor属性会导致它的[[Enimerable]]特性被设置为true，默认是不可枚举的。

function Person(){

}

Person.prototype = {

age:29,

job:“Software Engineer”,

sayName:function(){

alert(this.name);

}

}

//重设构造函数，只适用于ES5兼容的浏览器

Object.defineProperty(Person.prototype,“constructor”,{

enumerable:false,

value:Person

})

1. 原型的动态性

var friend = new Person（）;

Person.prototype.sayHi = functiob(){

alert(“hi”);

}

friend.sayHi(); //“hi”(没有问题!)

实例中的指针仅指向原型，而不指向构造函数

function Person(){

}

var friend = new Person();

Person.prototype = {

constructoe:Person,

name：“Nicholas”,

age:29,

sayName:function(){

alert(this.name);

}

};

friend.sayName(); //error

1. 原生对象的原型

所有原生引用类型（Object，Array,String,等等）都是在其构造函数的原型上定义了方法。

alert(typeof Array.prototype.sort); //“function”

alert(typeof String.prototype.substring); //“function”

String.prototype.startsWith = function(text){

return this.indexOf(text)==0;

}

var msg = “Hello world!”;

alert(msg.startsWith(“Hello”)); //true

不推荐在产品化的程序中修改原生对象的原型。

1. 原型对象的问题

function Person(){

}

Person.prototype = {

constructor:Person,

name:“Nicholas”,

age:29,

job:“Software Engineer”,

friends:[“Shelby”,“Court”],

sayName: function(){

alert(this.name);

}

};

var person1 = new Person();

var person2 = new Person();

person1.friends.push(“Van”);

alert(person1.friends); //“Shelby,Court,Van”

alert(person1.friends); //“Shelby,Court,Van”

alert(person1.friends === person2.friends); //true

### 6.2.4 组合使用构造函数模式和原型模式

function Person（name,age,job）{

this.name = name;

this.age = age;

this.job = job;

this,friecs = [“Shelby”,“Court”];

}

Person.prototype = {

constructor:Person

sayName:function(){

alert(this.name);

}

}

var person1 = new Person(“Nicholas”,29,“Software Engineer”);

var person2 = new Person(“Greg”，27，“Doctor”)；

person1.friends.push(“Van”)；

alert(person1.friends); //“Shelby”,Count,Van”

alert(person2.friends); //“Shellby,Count”

alert(person1.friends === person2.friends); //false

alert(person1.sayName===person2.sayName);

### 6.2.5 动态原型模式

可以通过检查某个应该存在的方法是否有效，来决定是否需要初始化原型。

function Person(name,age,job){

//属性

this.name = name;

this.age = age;

this.job = job;

//方法

if（typeof this.sayName!=”function”）{

Person.prototype.sayName = function(){

alert(this.name);

}

};

}

var friend = new Person(“Nicholas”,29,“Software Engineer”);

friend.sayName();

### 6.2.6 寄生构造函数模式

function Person（name,age,job）{

var o = new Object();

o.name = name;

o.ag = age;

o.job = job;

o.sayName = function(){

alert(this.name);

};

return o;

}

var friend = new Person(“Nicholas”,29,“Software Engineer”);

friend.sayName(); //“Nicholas”

functionSpecialArray(){

var values = new Array();

values.push.apply(values,arguments);

values.toPipedString = function(){

return this.join(“|”)

};

//返回数组

return values;

}

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

alert（colors.toPipedString()）;//“red|blue|green”

### 6.2.7 稳妥构造函数模式

稳妥对象指的是没有公共属性，且其方法也不引用this的对象。

function Person（name,age,job）{

//创建要返回的对象

var o = new Object();

//可以在这里定义私有变量和函数

//添加方法

o.sayName = function(){

alert(name);

};

//返回对象

return o;

}

var friend = Person(“Nicholas”,29,“Software Engineer”);

friend.sayName(); //”Nicholas”

## 6.3 继承

### 6.3.1 原型链

1、别忘记默认的原型

Object

2、确定原型和实例的关系

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();

SubType.prototype.getSubValue = function(){

return this.subproperty;

};

var instance = new SubType();

alert(instance.getSuperValue()); //true

alert(instance instanceof Object); // true

alert(instance instanceof SuperType); //true

alert(instance instanceod SubType); //true

alert(Object.prototype.isPrototypeOf(instance)); //true

alert(SuperType.prototype.isPrototypeOf(insatnce)); //true

alert(SubType.prptotype.isPrototypeOf(insatnce)); //true

3、谨慎的定义方法

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;

};

//重写超类型中的方法

SubType.prototype.getSuperValue = function(){

return false;

};

var instance = new SubType();

alert(istance.getSuperValue()); //false

通过原型链实现继承时，不能使用对象字面量创建原型方法，否则会重写原型链。

function SuperType(){

this.property = true;

}

SuperType.prototype.getSuperValue = function(){

return this.property;

};

//使用字面量添加新方法，会导致上一行代码无效

SubType.prototype = {

getSubValue:function(){

return this.subproperty;

}，

someOtherMethod:function(){

return false;

}

};

var instance = new SubType();

alert(instance.getSuperValue()); //error!

1. 原型链的问题

function SuperType(){

this.colors = [“red”,“blue”,“green”]；

}

function SubType(){}

//继承了SuperType

SubType.prototype = new SuperType();

var instance1 = new SubType();

instance1.colors.push(“black”);

alert(instance1.colors); //”red,blue,green,black”

var instance2 = new SubType();

alert(instance2.colors); //”red,blue,green,black”

### 6.3.2 借用构造函数（伪造对象/经典继承）

function SuperType(){

this.colors = [“red,””blue”,”green”];

}

function SubType(){

//继承了SuperType

SuperType.call(this);

}

var instance1 = new SubType();

instance1.colors.push(“black”); //“red,blue,green,black”

alert(instance1.colors); //”red,blue,green,black”

var insatnce2 = new SubType();

alert(instance2.colors); //”red,blue,green”

1. 传递参数

function SuperType(name){

this.name = name;

}

function SubType(){

//继承了SuperType,同时还传递了参数

SuperTYpe.call(this,“Nicholas”);

//实例属性

this.age = 29;

}

var instance = new SubType();

alert(instance.name); //“Nicholas”;

alert(instance.age); //29

### 6.3.3 组合继承

伪经典继承，指的是将原型链和借用构造函数的技术组合到一块

function SuperType(name){

this.name = name;

this.colors = [“red”,“blue”,“green”];

}

SuperType.prototype.sayName = function(){

alert(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.constructor = SubType;

SubType.prototype.sayAge = function(){

alert(this.age);

}

var instance1 = new SubType(“Nicholas”,29);

instance1.colors.push(“black”);

alert（instance1.colors）; //“red,blue,green,black”

instance1.sayName(); //“Nicholas”；

instance1.sayAge(); //29

### 6.3.4 原型式继承

function object(o){

function F(){}

F.prototype = o;

return new F();

}

var person = {

name:”Nicholas”,

friends:[“Shelby”,”Court”,”Van”]

};

var anotherPerson = object(person);

var person = {

name:“Nicholas”，

friends:[“Shelly”,“Court”,”Van”]

};

var anotherPerson = object(person);

anotherPerson.name = “Greg”;

anotherPerson.friends.push(“Rob”)；

var yetAnotherPerson = object(person);

yetAnotherPerson.name = “Linda”;

yetAnotherPerson.friends.push(“Barbie”);

alert(person.friends); //”Shelly,Court,Van,Rob，Barbie”

var person = {

name:“Nicholas”,

friends:[“Shelby”,”Court”,”Van”]

};

var anotherPerson = Object.create(person);

anotherPerson.name = “Linda”

var person = {

name:“Nicholas”,

friends:[“Shelly”,”Court”,”Van”]

};

var anotherPerson = Object.create(person);

anotherPerson.name=”Greg”;

anotherPerson.friends.push(“Rob”);

var yetAnotherPerson = Object.create(person);

yetAnotherPerson.name = “Linda”;

yetAnotherPerson.friends.push(“Barbie”);

alert(person.friends); //”Shelby,Court,Van,Rob,Barbie”

var person = {

name:“Nicholas”,

friends:[“Shelby”,”court”,”Van”]

}

var anotherPerson = Object.create(person,{

name:{

value:”Greg”

}

});

alert(anotherPerson.name); //”Greg”

### 6.3.5 寄生式继承

function createAnother(original){

var clone =object(original); //通过调用函数创建一个新对象

clone.sayHi = function() { //以某种方式来增强这个对象

alert(“hi”);

};

return clone; //返回这个对象

}

var person = {

name:“Nicholas”,

friends:[“Shelly”,”Court”,”Van”]

};

var anotherPerson = createAnother(person);

anotherPerson.sayHi(); //”hi”

### 6.3.6 寄生组合式继承

function SuperType(name){

this.name = name;

this.colors = [“red”,”blue”,”green”];

}

SuperType.prototype.sayName = function(){

alert(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(){

alert(this.age);

};

function inheritPrototype(subtype,superType){

var prototype = object(supertype.prototype); //创建对象

prototype.constructor = subtype; //增强对象

subtype.prototype = prtototype; //指定对象

}

function SuperType(name){

this.name = name;

this.colors = [“red”,”blue”,”green”];

}

SuperType.prototype.sayName = function(){

alert(this.name);

};

function SubType(name,age){

SuperType.call(this,name);

this.age = age;

}

inheritPrototype(SubType,SuperType);

SubType.prototype.sayAge = function(){

alert(this.age);

};

## 6.4小节

# 第7章 函数表达式

函数声明，声明提升

sayHi();

function sayHi(){

alert(“Hi”)；

}

函数表达书（匿名函数）

asyHi(); //错误；函数不存在

var sayHi = function(){

alert(“Hi！”)；

}；

//不要这样做

if（condition）{

function sayHi(){

alert(“Hi!”);

}

}else{

function sayHi(){

alert(“Yo!”);

}

}

//可以这样做

var sayHi;

if(condition){

sayHi = function(){

alert(“Hi”)；

}；

}

else{

sayHi = function(){

alert(“Yo!”);

};

}

## 7.1 递归

function factorial(num){

if(num <=1){

return 1;

} else{

return num\* factorial(num-1);

}

}

var anotherFactorial = factorial;

factorial = null;

alert(anotherFactorial(4)); //出错！

function factorial(num){

if(num<= 1){

return 1;

}else{

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

}

}

严格模式下，不能通过脚本访问arguments.callee，可以使用命名函数来表达相同的结果

var factorial = (function f(num){

if(num <=1){

return 1;

}else{

return num \* f(num-1);

}

});

## 7.2 闭包

有权访问另一个函数作用域中的变量的函数

### 7.2.1 闭包与变量

闭包只能取得包含函数中任何变量的最后一个值。

function createFunctions(){

var result = new Array();

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

result[i] = function(){

return i;

};

}

return result;

}

结果是每个函数都返回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;

}

没有直接把闭包赋值给数组，而是定义了一个匿名函数，并将立即执行该匿名函数的结果赋给数组。这里的匿名函数有一个参数num,也就是最终的函数要返回的值。调用匿名函数时，我们传入了变量i，由于函数参数是按值传递的，所以就将变量i的当前值复制给参数num.而在这个匿名函数的内部，又创建返回了一个访问num的闭包，这样一来，result数组中的每个函数都有自己num变量的一个副本。

### 7.2.2 关于this对象

this对象是在运行时基于函数的执行环境绑定的：在全局函数中，this等于window，而当函数作为某个对象的方法调用时，this等于把那个对象，不过，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”,

getNameFunc:function(){

return function(){

return this.name;

};

}

};

alert(object.getNameFunc()()); //”The window”

var name=”The Window”;

var object = {

name:”My Object”,

getName:function(){

return this.name;

}

};

object.getName(); //”My Object”

(object.getName)(); //”My Object”

(object.getName=object.getName)(); //”The Window”,在非严格模式下

### 7.2.3 内存泄露

function assignHandler(){

var element = document.getElementById(“someElement”);

element.onclick = function(){

alert(element.id);

};

}

function assignHandler(){

var element = document.getElementById(“someElement”);

var id = element.id;

element.onclick = function(){

alert(id);

};

element = null;

}

## 7.3 模仿块级作用域

function outputNumbers(count){

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

alert(i);

}

alert(i); //计数

}

function outputNumbers(count){

(function(){

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

alert(i);

}

})();

alert(i); //导致一个错误！

}

## 7.4 私有变量

function MyObject(){

//私有变量和私有函数

var privateVariable = 10;

function privateFunction(){

return false;

}

//特权方法

this.publicMethod = function(){

privateVariable++;

return privateFunction();

};

}

function Person(name){

this.getName = function(){

return name;

};

this.setName = function(value){

name = value;

};

}

var person = new Person(“Nicholas”);

alert(person.getName()); // “Nicholas”

person.setName(“Greg”);

alert(person.getName()); //”Greg”

### 7.4.1 静态私有变量

（function(){

//私有变量和私有函数

var privateVariable = 10;

function privateFunction(){

return false;

}

// 构造函数

MyObject = function(){};

// 公有/特权方法

MyObject.prototype.publicMethod = function(){

privateVariable++;

return privateFunction();

};

}

）（）；

(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(“Nicholas”);

alert(person1.getName()); //”Nicholas”

person1.settName(“Greg”);

alert(person1.getName()); //”Greg”

var person2 = new Person(“Michael”);

alert(person1.getName()); //”Michael”

alert(person2.getName()); //”Michael”

### 7.4.2 模块模式

js以对象字面量的方式来创建单例对象

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得到增强。

var singleton = function(){

//私有变量和私有函数

var privateVariable = 10;

function privateFunction(){

return false;

}

//特权/公有方法和属性

return{

publicProperty:true,

publicMethod:function(){

privateVariable++;

return privateFunction();

}

};

}();

这个模块模式使用了一个返回对象的匿名函数，在这个匿名函数内部，首先定义了私有变量和函数。然后，将一个对象字面量作为函数的值返回。返回的对象字面量中只包含可以公开的属性和方法。

var application = function(){

//私有变量和函数

var components = new Array();

//初始化

components.push(new BaseComponent());

//公共

return{

getComponentCount:function(){

return components.length;

},

registerComponent:function(component){

if(typeof component == “object”){

components.push(component);

}

}

};

}();

### 7.4.3 增强的模块模式

适合那些单例必须是某种类型的实例，同时还必须添加某些属性和（或）方法对其增强的情况。

var singleton = function(){

//私有变量和私有函数

var privateVariable=10;

function privateFunction(){

retrun false;

}

//创建对象

var object = new CustomType();

//添加特权/公有属性和方法

object.publicProperty = true;

object.publicMethos = function(){

privateVariable++;

return privateFucntion();

};

//返回这个对象

return object;

}

var application = function(){

//私有变量和函数

var components = new Array();

//初始化

components.push(new BaseComponent());

//创建applicationf的一个局部副本

var app = new BaseComponent（）；

//公共接口

app.getComponentCount=function(){

return components.length;

};

app.registerComponent=function(component){

if(typeof component == “object”){

components.push(component);

}

};

//返回这个副本

return app;

}();

## 7.5 小节

函数表达式无需对函数命名。匿名函数也叫拉姆达函数。

函数表达式的特点：

函数表达式不同于函数声明。函数声明要求有名字，函数表达式不需要。没有名字的函数表达式也叫匿名函数。

无法确定如何引用函数的情况下，递归函数就会变得比较复杂。

递归函数应该始终使用arguments.callee来递归调用自身，不要使用函数名-函数名可能会发生变化。

当在函数内部定义了其他函数时，就创建了闭包。闭包有权访问包含函数内部的所有变量，原理如下：

1、在后台执行环境中，闭包的作用域链包含着它自己的作用域、包含函数的作用域和

全局作用域。

1. 通常，函数的作用域及其所有变量都会在函数执行结束后被销毁
2. 但是，当函数返回了一个闭包时，这个函数的作用域将会一直在内存中保存到闭包不存在为止。

使用闭包可以在js中模仿块级作用域（js本身没有块级作用域的概念），要点如：

1. 创建并立即调用一个函数，这样既可以执行其中的代码，又不会在内存中留下对该函数的引用。
2. 结果就是函数内部的所有变量都会被立即销毁—除非将某些变量赋值给了包含作用域（即外部作用域）中的变量。

闭包还可以用于在对象中创建私有变量，相关概念和要点如下。

1. 即使js中没有正式的私有对象属性的概念，但可以使用闭包来实现公有方法，而通过公有方法可以访问在包含作用域中定义的变量。
2. 有权访问呢私有变量的公有方法叫做特权方法。
3. 可以使用构造函数模式、原型模式来实现自定义类型的特权方法，也可以使用模块模式、增强的模块模式来实现单例的特权方法。

# 第8章 BOM

## 8.1 window对象

BOM(浏览器对象)的核心对象是window，它表示浏览器的一个实例。window对象既是通过js访问浏览器窗口的一个接口，又是ES规定的Global对象。

### 8.1.1 全局作用域

var age = 29;

functionsayAge(){

alert(this.age);

}

alert(window.age); //29

sayAge(); //29

window.sayAge(); //29

全局变量不能通过delete操作符删除，直接在window对象上的定义的属性可以。

var age = 29;

window.color = “red”;

//在IE<9时抛出错误，在其他所有浏览器中都返回false

delete window.age;

//在IE<9时抛出错误，在其他所有浏览器中都返回true

delete window,color; // return true

alert(window.age); //29

alert(window.color); //undefined

使用var语句添加的window属性有一个名为【【Configurable】】的特性，这个特性的值被设置为false,这样定义的属性不可以通过delete操作符删除IE8及更早版本在遇到delete删除window属性的语句时，不管该属性最初是如何创建的，都会抛出错误。IE9及更高版本不会。

尝试访问未声明的变量会抛出错误，但是通过查询window对象，可以知道某个可能未声明的变量是否存在。

//这里会抛出错误，因为oldValue未定义

var newValue = oldValue;

//这里不会抛出错误，因为这是一次属性查询

//newValue的值是undefined

var new Value = window.oldValue;

### 8.1.2 窗口关系及框架

window.psrent window.topd

### 8.1.3 窗口位置

下列代码可以跨浏览器取得窗口左边和上边的位置。

var leftPos = (typeof window.screenLeft ==”number”)?window,screenLeft:window.screenX;

var topPos = (typeof window.scrrenTop == “number”)?window.screenTop:window.scrrenY;

这个例子运用二元操作符首先确定screenLeft和screenTop属性是否存在，如果是（在IE、Safari、Opera和Chome中），则取得这两个属性的值。如果不存在（在Firefox中），则取得screenX 和screenY的值。

### 8.1.4 窗口大小

resizeTo(); //接收浏览器窗口的新宽度和新高度

reseizeBy()； //接收新窗口与原窗口的宽度和高度之差

这两个方法有可能被浏览器禁用，另外，这两个方法不适用于框架，只能对最外层的window对象使用。

### 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。

1. 弹出窗口

var wroxWin = window.open(<http://www.wrox.com/,“wroxWindow”,”height> = 400,width=400,top = 10,left=10,reseizable=yes”)

//调整大小

wroxWin.resizeTo(500,500);

//移动位置

wroxWin.moveTo(100,100);

//调用close()方法关闭新打开的窗口

wroxWin.close(); //仅适用于通过window.open（）打开的弹出窗口

alert(wroxWin.closed); //true

新创建的window对象有一个opener属性，其中保存着打开它的原始窗口对象。这个属性只在弹出窗口的最外层window对象（top）中有定义，而且指向window.open()的窗口或框架。

var wroxWin = window.open(<http://www.wrox.com/,“wroxWindow”,”height> = 400,width=400,top = 10,left=10,reseizable=yes”);

alert(wroxWin.opener == window); //true

var wroxWin = window.open(<http://www.wrox.com/,“wroxWindow”,”height> = 400,width=400,top = 10,left=10,reseizable=yes”);

wroxWin.opener = null;

将opener属性设置为null，就是告诉浏览器新创建的标签页不需要与打开它的标签页通信，因此可以在独立的进程中运行。标签页之间的联系一旦切断，将没有办法恢复。