# 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;

}

};

}

### 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