JavaScript权威指南

1.类型，值和变量

js中的数据类型分为两类，**原始类型**和**对象类型。**原始类型包括数字，字符串和布尔值还有两个特殊的原始值：`null`（空）和`ubdefined`(未定义)

对象是属性(property)的集合，每个属性由属性名=属性值构成，还有全局对象(global object).

普通的js对象(Object对象)是命名值的无序集合。JS同样定义了一种特殊的对象:数组array,表示带有编号的有序集合，数组拥有一些和普 通对象不同的特有行为特性

js还定义了另一组特殊的对象：函数，函数和其他对象都不一样，js可以将函数当做普通对象来对待

如果函数用来初始化（使用new来构造）则将其称作**构造函数**（constructor）,每个构造函数都定义了一**类**(class)对象，**类**可以看成对象类型的的子类型。除了数组类（Array）和函数类（Function）外，js还定义了其他三种有用的类：

- 日期Data类：定义了代表日期的对象
- 正则RegExp类：定义了正则表达式
- 错误Error类：定义了表示js程序中运行时的错误和语法错误的对象
- 也可以通过定义自己的构造函数来定义需要的类

JS的类型可以分为原始类型和对象类型，也可以分为可以拥有方法的类型和不能拥有方法的类型，也可以分为可变类型和不可变类型，可变类型的值是可以修改的，如对象和数组；不可变类型为数字，布尔值，null和undefined。**在js中字符串是不可变的**，可以访问字符串的任意位置的文本，但不能修改字符串的内容。

JS可以自动自由的进行数据类型的转换，如在期望使用字符串的地方使用了数字，js会自动将数字转换为字符串（如alert(12345)）

JS的变量是无类型的，变量可以被赋予任何类型的值，不在函数内声明的变量叫做全局变量

在JS中不区分整形与浮点型，所有的数都由浮点型表示

JS中的算术运算

js中的复杂运算通过作为***Math***对象的属性顶底函数和常量实现

- `Math.pow(2,53)`,`Math.sqrt(3)`:幂运算，表示2的53次方和开根运算（也可以直接分数次幂来计算开根如：`Math.pow(10,1/3)`就是10开3次根）
- `Math.round()` `Math.ceil()` `Math.floor` :四舍五入，向上取整以及向下取整
- `Math.abs()`:求绝对值

- Math.max(a,b,c),Math.min():返回最大/最小值
- Math.random():生成一个0~1的伪随机数
- Math.E,Math.PI:e和π
- Math.sin()，Math.cos():三角函数
- Math.log(10)/Math.LN2: $\log_2 10$
- Math.exp(3):e^3

在js中被零除并不会报错，而是返回一个无穷大(Infinity)或负无穷大，或是用零除以零返回NaN（not-a-number)

日期和事件

使用Data创建事件对象，具体创建方法如下所示：

```
var then=new Data(2011,0,1)//2011年1月1日（顺序是年，月，日，其中月是从0开始到11）
var later=new Date(2021,0,2,17,10,30)//加了小时分钟和秒
var now=new Date()//什么都不加就是获取当前的时间
var elapsed=now-then//日期减法，计算时间间隔的毫秒数
```

结果如下：

```
Fri Jan 01 2021 00:00:00 GMT+0800 (中国标准时间)
Sat Jan 02 2021 17:10:30 GMT+0800 (中国标准时间)
Tue Nov 02 2021 19:10:18 GMT+0800 (中国标准时间)
26421018744
```

获取Data对象的指定内容：

```
now.getFullYear()       =2021
now.getMonth()          =10//获取当前的月份(当前是11月但是月份显示要减一，所以11-1=10)
now.getDay()            =2
now.getHours()          =19
```

对data对象使用valueOf()会返回从1970年1月1日以来的毫秒数 now.valueOf(),对其他对象使用的话 只是简单返回对象本身

字符串

字符串可以调用的方法：var s="hello,world"

- s.length:获得字符串的长度，为11（标点也占位置）
- s.charAt(n):字符串中第n+1个元素，等价于s[n]
- s.substring(a,b)或s.slice(a,b):字符串中第a+1到b之间的所有元素
- s.slice(-3):最后3个字符，rld
- s.indexOf("l"):字符l首次出现的位置，为2
- s.indexOf("l",3):在位置3及以后最先出现l的位置，为3
- s.lastIndexOf("l"):字符l最后出现的位置，为10
- s.split(','):以逗号","将字符串分开
- s.replace("h","H"):全字符替换
- s.toUpperCase():全部变成大写

全局对象

全局对象的属性时全局定义的符号，js程序可以直接使用

- 全局属性：如undefined,Infinity和NaN
- 全局函数：比如eval()；isNaN()；parseInt()等
- 构造函数：如Date(),RegExp(),String(),Object().和Array()
- 全局对象：如Math和JSON

在代码的最顶层--不再任何函数内的js代码，可以使用使用js关键字this来引用全局对象：

```
var global=this
```

包装对象

如果直接给数字，字符串或者布尔值属性的话，就像下面所示：

```
var s='aaa'
s.len=3//直接给s这个字符串一个len属性
var t=s.len//这样写的话在运行时t的值为undefined
```

js会忽略你这么直接给他们属性，修改只是发生在临时对象上，而这个临时对象并没有保存下来，存取字符串，数字或是布尔型时创建的对象称作**包装对象**

字符串，数字，布尔值的属性是只读的，不能给他们新的属性

但是若是创建了对象的话就可以给与他们属性值了，如下所示

```
var ss=new String(sssss)
ss.len=5
var t=ss.len//此时就会返回5了
```

原始值和对象

js中的**原始值**（undefined，null，布尔值，数字和字符串）与**对象**（如数组和函数）有着根本的区别。原始值是不可更改的，任何方法都无法更改一个原始值，原始值的比较是值的比较，他们的值相等时他们才相等

对象和原始值不同，首先，他们的值是可以修改的，如下：

```
var o={x:1}//定义一个对象,并有一个x的属性，值为1
o.x=2   //修改这个对象
o.y=3   //再次修改这个对象，为其增加一个新的属性y

var a=[1,2,3] //数组也是可以修改的
a[0]=0    //可以修改数组中的元素的值
a[3]=4    //也可以给数组添加一个新的元素
```

用大括号{ }包起来的是对象，用中括号[ ]包起来的是数组

对象的比较并非值的比较，即使两个对象包含相同的属性和值他们也不是相等的

我们通常将对象称为引用类型以此来与js中的基本类型相区分,对象的比较是引用的比较，当且仅当他们引用同一个基对象时，他们才相等

```
var a=[]
var b=a //变量b引用同一个数组
b[0]=1  //通过变量b来修改引用的数组
//a[0]此时也变成1了
//a==b 为TRUE因为他们都是引用的同一个数组
```

如果我们想比较两个单独的对象或是数组，就得循环比较他们所有的元素或是属性，全部相同时才是一样的（元素或属性值比较时就同时判断了是不是同一个引用对象了）

格式转换

js中会自动进行格式转换，但是在一些特殊情况或是为了使代码更方便阅读而做**显示类型转换**

```
Number("3")

String(sss)  //或是 sss.toString()  除了null和undefined之外任何值都拥有这种方法,就算是函数或是数组也可以这么用
[1,2,3].toString()//=>"1,2,3"
(function(x){f(x)}).toString//=>"function(x){\n f(x) \n}"

Boolean([]) //只要不是空就是true
object(3)   //=>new Number(3)
```

隐式的类型转换：js中的某些运算符会做隐式的类型转换，

如+运算符的一个操作数是字符串，他将会把另一个操作数也化为字符串。

一元+操作符将字符串转换为数字，

一元!运算符将其操作数转换为布尔值并取反，例子如下：

```
x+'' //等价于String(x)
+x   //等价于Number（x）,也可以写成x-0
!!x  //等价于Boolean(x),注意是双感叹号，否则会取反

typeof(now +1)//=>string +将日期转换为字符串
typeof(now -1)//=>number -将日期对象转换为数字
```

控制输出的小数的格式：

```
var n=12345.678
//toFixed(n)是直接控制小数点后的位数为n
n.toFixed(0) //不要小数点后面的，而且会四舍五入，返回值为12346
n.toFixed(2)//12345.68
n.toFixed(5)//12345.67800 位数不够会补零
//toPrecision(n)是控制整个数的长度，长度不够的话会使用科学计数法，不足时也会补零,也会四舍五入
n.toPrecision(6)//12345.7
```

还可以使用paresInt()和parseFloat()

变量

如果在局部变量的作用域内定义了与全局变量同名的局部变量，则局部变量会遮盖全局变量

定义全局变量时可以不写var,但声明局部变量时必须用var

注意:

看下面的例子，判断js程序输出到底是什么

```
var scope='global'
function(){
 console.log(scope)//猜一猜这一行会输出什么
 var scopr='local'
 console.log(scope)
}
```

第一个输出不是意象中的global而是undefined,因为局部变量在整个函数体始终是有定义的，只要在作用域内定义了同名的局部变量，就会直接覆盖掉全局变量，但在var时才对同名的局部变量进行赋值，以上代码相当于

```
function(){
 var scope
 console.log(scope)
 var scopr='local'
 console.log(scope)
}
```

所以一般将函数中声明的局部变量都放在最顶端

2.表达式和运算符

初始化数组: `var a=[[1,2],[2,3],[3,4]]`

初始化对象: `var p={x:1.2,y:2.3}` 对象也可以嵌套`var rect={up:{x:2,y:3},down:{x:4,y:6}}`

对象的属性的值可以是表达式，属性的名字也可以是字符串('rec':4+o.len)

属性访问表达式:

通过属性访问表达式所得到的是一个对象属性或是一个元租的值，js有两种方法可以获得属性:

```
expression.identifier  //元素.属性
e[a]//元素[属性]

//如下所示:
var o={x:1,y:{z:3}}
var a=[o,4,[5,6]]
o.x     //=>1
o.y.z   //=>3嵌套的可以这么点下去来求嵌套内的属性
o['x']  //=>1

a[2]["1"]//=>6 a[2]中索引为1的值，与a[2][1]，a['2'][1]结果相同
a[0].x   //=>1 a[0]即o的x属性的值
```

运算符

1.**++**:

i++和++i是不同的,差别如下

```
var i=1 ,j=++i  //i和j的值都是2
var i=1, j=i++  //i是2，j是1
//--也同理
```

2.**in运算符**:

如果右侧的对象拥有一个名为左操作数的属性名则表达式返回true

```
var point={x:1,y:1}
"x" in point //true

var data=[7,8,9]
'0' in data //true，因为数组有0这个属性值,相当于是data[0]
2 in data   //true 同上，数组的属性可以是数字或是字符串
3 in data   //false 因为这个数组长度是3，只有0,1,2三个属性，没有3这个属性
```

这个in不能用来判断一个字符串中是否有某个元素，只能判断有没有指定的属性值

3.**instanceof运算符**:

判断对象是什么类型的对象

```
var d=new Data()
d instanceof Data //true
d instanceof Object//True 所有的对象都是Object的实例
d instanceof Number//false
```

4.**typeof运算符**:

是一元运算符，放在单个操作数的前面，返回值为操作数类型的一个字符串，typeof最常用的用法是写在表达式中:

```
(typeof value=='string')?"'"+value+"'":value//如果value的类型是字符串则给他加上单引号，不是则不加
```

也可以用在Switch语句中。

对于所有的对象和数组的typeof运算结果是object,如果想区分对象的类，则需要使用其他手段如instanceof运算符，class特性以及constructor属性

5.**delete运算符**:

用来删除对象属性或是数组元素，没有返回值

```
var o={x:1,y:2}
delete o.x
x in o //=>false

var a =[1,2,3]
delete a[2]
2 in a //=>false ,将a[2]=3这个属性删了
a.length//=>3 注意数组的长度没有改变，虽然上边删了一个数，但是留下了一个空洞' ',数组的长度没有变，可以通过in看这个这个属性中有值么
```

删的是属性而不是元素，更不能直接删var定义的变量

6.**void运算符**:

就是空，可以当做一个空函数在超链接中调用来以此来使得超链接点击后会变颜色

```
<a href="javascript:void" > </a>
```

7.**逗号运算符**:

可以使for循环多次循环

```
for(var i=1,j=0;i<j;i++,j--)
```

eval()

一般情况下，eval（）会执行括号内的表达式或是函数，但是eval()还具有更改局部变量和全局变量的能力

```
var q='global',p='global'
var geval=eval
function f(){
  var q='local'
  eval('q+="change"')
  return q
}
function g(){
  var p='local'
  geval('p+="change"')
  return p
}
function test(){
  console.log(f(),q)  //输出为 localchange和global改变了局部变量，从local=>localchange
此时有没有eval都无所谓
  console.log(g(),p)  //输出为 local和globalchange改变了全局变量，从
global=>globalchange
}
```

想要在函数中改变全局变量的话就得先定义一个非限定的eval名字如`var geval=eval`,然后使用这个非限定的eval来处理全局变量`geval('p+="change"')`

3.语句

循环

循环可以没有循环体，称为空循环语句，需要加一个分号

for语句不仅可以对数字进行循环，还可以用for来循环遍历链表数据结构，并返回链表中的最后一个对象

```
function tail(o){
    for(;o.next;o=o.next);/*empty*/   //空正文，如果o.next为真就执行o=o.next    就是说如果链表中下一个元素仍存在就使当前元素变为下一个元素
    return o
}
```

和python一样，js也可以使用for('' in '') for/in循环,可以方便的遍历对象属性成员

```
for (var p in o){//将o中的所有属性名称依次给p
    alert(p+':'o[p]) //输出属性的名字和属性的值
}
```

要注意的是每次循环都会计算for括号里的表达式，如可以用以下的方式将所有对象属性复制至一个数组中：

```
var o={x:1,y:2,z:3}
var a=[],i=0
for(a[i++] in o)/*empty*/ //空正文，但是已经将o中的属性全部给a了

//也可以
for(i in a) console.log(i)
```

for/in循环并不会遍历对象的所有属性，只有**可枚举**的属性才会被遍历到

跳转以及异常处理

标签语句

语句是可以加标签的，可以通过标签跳转到指定的语句，标签是由标识符和冒号组成（如汇编的跳转）

通过给语句定义标签，就可以在程序的任何地方通过签名引用这条语句，也可以同时对多个语句定义标签，一个语句也可以有多个标签(带有标签的语句还可以带有标签)

```
mainloop: while(){
                ...
                continue mainloop;//跳到下一次循环
                ...
                }
```

break也可以后面跟一个标签跳出当前循环跳到标签语句，但是不管brek带不带标签，它的控制权都无法越过程序的边界，不能从函数内部通过标签来跳转到函数外部。

throw语句

异常为当繁盛了某种异常情况或产生错误时产生的一个信号。**抛出异常**就是用信号通知发生了错误或是异常情况，**捕获异常**是指处理这个信号，即采取必要的手段从异常中恢复

在js中，当产生错误或是程序使用throw语句时就会显式的抛出异常，使用`try/catch/finally`语句可以捕获异常

通过throw抛出的异常可以是任意类型的，可以是一个代表错误码的数字，或者包含刻度的错误消息的字符串。一般js中抛出异常时通常采用`Error`类型和其子类型，一个**Error对象**有一个name属性表示错误类型，一个message属性用来存放传递给构造函数的字符串，例子如下所示：

```
function factorial(x){
    //如果输入参数是非法的则抛出一个异常
    if(x<0) throw new Error('x不能是负数')
    //否则计算出一个值并且正常的返回它
    for(var f=1;x>1;f*=x,x--);/*empty*/
    return f
}
```

当抛出异常时，js会立刻停止当前正在执行的程序，并跳转到就近的异常处理程序，如果抛出异常的代码块没有一条相关联的catch从句，解释器会一直向高层检查，如果最后就是找不到，js就将其当做程序错误来处理，并报告给用户。

抛异常可以抛中文如`throw "输入不能比设定值小"`

try/catch/finally语句

这些语句是js的异常处理机制，try从句定义了需要处理的异常所在的代码块，catch从句跟随在try之后，当try块内部某处发生了异常时，调用catch内的代码逻辑，catch后跟随finally从句，不管try中是否发生异常，finally中的逻辑总是会执行，这三个语块都需要用大括号括起来，实例如下：

```
try{
    //通常这里的代码不会出现任何问题，但有时会抛出一个异常（程序抛或是使用throw抛或是调用方法间接抛）
}
catch(e){
    //try中内容抛出异常时才会来到这里面执行代码
    //这里可以通过局部变量e来获得对Error对象或者抛出的其他值的引用
    //这里可以处理这个异常，也可以忽略这个异常，还可以通过throw重新抛出异常
}
finally{
    //总是会执行，终止try语句块的方法如下：
    //1.正常终止，执行完语句块的最后一条语句
    //2.通过break，continue或return语句终止
    //3.抛出一个异常，异常被catch捕获
    //4.抛出一个异常，异常未被捕获，继续向上传播
}
```

关键字catch后跟了一个小括号中有一个**标识符**e,这个标识符和函数参数很像，当捕获一个异常时，把和这个异常相关的值（比如Error对象）赋值给这个参数，它只在catch语句块内有定义

如对上述的factorial函数所抛出的不能是负数异常进行处理：

```
try{
    var n=Number(prompt("请输入一个正整数",""))
    var f=factorial(n)//假如输入是合法的则输出这个数的阶乘
    alert(n+'!='+f)
  }
catch (ex){
    //如果不合法，则执行这里的逻辑
    alert(ex)//告诉用户产生了什么错误，因为之前设置的错误为('x不能是负数')，所以报错也报这个
  }
```

不一定非要用finally，它通常用于清理工作

其他语句类型

with语句

在获取网页上的表单中的元素时，可以用如下的方法来访问一个表单的元素：

```
document.forms[0].address.value //这是访问页面中的第一个表单的name为address的用户的输入值
```

如果这种表达式在代码中多次出现，则可以使用with语句将form对象添加至作用域链的顶层：

```
with(document.forms[0]){
    //可以直接访问表单元素
    name.value=''
    address.value=''
    email.value=''
}
```

debugger语句

用来在调试程序时产生一个断点方便进行调试，如下：

```
function f(0){
    if (o==undefined) debugger  //这一行代码用于临时调试
}
```

'use strict'

使用严格模式，只能出现在脚本代码的开始或是函数体的开始。严格模型后with不能用，所有变量使用前都得先声明等等等等，代码的规则就变严了

4.对象

对象是js的一种基本数据类型，是一种复合值，它将很多值聚合在一起，可以通过名字访问这些值。

对象可以看成属性的无序集合，每个属性都是一个名/值对。属性名是字符串，我们可以把对象看成从字符串到值的映射。

这种数据结构还有很多种叫法，如散列（hash），散列表，字典，关联数组。

对象不仅仅是字符串到值的映射，除了可以保持自有的属性，js中的对象还可以从一个称为原型的对象**继承属性**，对象的方法通常是继承的属性，这种原型式继承是js的核心特征

js的对象是动态的，可以新增属性也可以删除属性，但他们常用来模拟静态对象以及静态类型语言中的“结构体”

因为对象是可变的，我们通常通过引用而非值来操作对象，如下：

```
var y=x //y和x是对于同一个对象的引用而不是这个对象的副本，所以通过y来修改这个对象也会对x造成影响
```

对象最常见的用法是：创建，设置，查找，删除，检测和枚举。

对于每个属性除了名字和值之外，它们还有一些**属性特性**:可写，可枚举，可配置（是否可以删除或修改属性）

除了包含属性之外，每个对象还拥有三个相关的**对象特性**：

- 对象的原型(prototype)：指向另外一个对象，本对象的属性继承自它的原型对象
- 对象的类（class)：是一个标识对象类型的字符串
- 对象的扩展标记（extensible flag）:指明了是否可以向该对象添加新的属性(ES5)

对象的分类：

- 内置对象：如数组函数，日期和正则表达式都属于内置对象
- 宿主对象(host object):
- 自定义对象：由js代码所创建的对象
- 自有属性：是直接在对象中定义的属性
- 继承属性：是在对象的原型对象中定义的对象

创建对象

可以通过对象直接量，关键字new或Object.create()函数来创建

对象直接量

创建对象的最简单的方式，直接用大括号创建对象，并给出属性名称和属性值

```
var book={
    "main title":"javascript", //因为属性名字中有空格，所以必须加引号用字符串表示
    "for":"all audiences",     //for是保留字（js中有for这个语法），所以必须加引号
    author:{                   //这里的属性值是一个对象
    firstname:"David",    //这里的属性名就可以不加引号了
    lastname:"Flanagan"
    }
}

//对象的属性值可以是任何数据类型，也可以是函数如
var a={
    next:function(){ return 0 },
    t:fllow()
}
//调用这些属性时就相当于调用它们的函数方法 (a.next的值就是0)
```

对象直接量每次运算都会初始化一个新的对象，如果在循环中用了对象直接量会创建很多的对象

通过new来创建对象

直接就new加一个构造函数就可以了，如var a =new Data()

原型

每一个js对象（null除外）都和另一个对象想关联，那'另一个对象'就是原型，每一个对象都从原型继承属性。

所用通过对象直接量创建的对象都具有同一个原型对象，并可以通过Object.prototypeKaua获得对原型的引用。通过关键字new创建的对象的原型就是所使用的构造函数的prototype属性的值，如new Array()创建的对象的原型就是Array.prototype

Object.crate()

是一个静态函数，使用时只需要传入所需的原型对象就可`var p1=Object.create({x:1,y:2})`
(这里的原型对象就是{x:1,y:2})

下例是通过原型继承来创建一个新的对象

```
function inherit(p){
    if (p==null) throw TypeError()  //首先原型对象不能为空
    if(Object.create)
        return Object.create(p) //如果Object.create方法存在，则直接使用其来创建新的对象，函数到此结束
    var t=typeof p
    if(t!=='object'&&t!=='function') throw TypeError() //必须是函数或是对象才能被继承
    function f(){}  //定义一个空的构造函数
    f.prototype=p   //将原型设置为p
    return new f()  //返回一个原型为p的新的对象
}

inherit(o)//返回了一个继承自原型对象o的属性的新对象
```

这个函数返回的新对象继承了参数对象的属性，这个函数的一个作用是防止库函数无意间修改那些不受你控制的对象，从而影响最顶层的对象的属性值。

像这样继承原型的话更改属性值不会影响原型的属性值，只会影响继承对象自身，而不是原始对象。使用方法如下：

```
var o={x:"dont change this value!"}
library_function(inherit(o)) //这样就可以防止对原型o的修改了，现在是创建了一个新的对象继承o的所有属性值和o没有关系了。
```

通过Object.create()和inherit()可以通过对象来创建的对象，这样创建的对象不会影响其原对象

属性的查询和设置

之前已经提到过，可以使用.或是[]获得属性的值，获取对象属性的值和给对象的属性赋值时一样的。如果属性名字是非法的（如有空格或是js中的关键字）那就智能通过中括号[]来访问属性的值了，如`a['for']`或是`a['first name']`

继承

js对象具有自有属性，也有一些属性是从原型对象继承而来的。假设要查询对象o的属性x,如果o中不存在x，那么会继续在o的原型对象中查询属性x,若原型对象中也没有x这个属性且自身仍有原型对象，那么就继续向上查询，直到查到x属性或是一个原型是null的对象为止，通过这个链可以实现属性的继承

```
var o={}
o.x=1   //给对象o一个x的属性，值为1
var p=inherit(o)//用o作为原型对象创建一个新的p对象
p.y=2
var q=inherit(p)
q.z=3
var s=q.toString() //toString继承自Object.prototype
q.x+q.y  //=>3  虽然在q这个对象中没有定义x和y属性，但它的原型对象中有，在p中找到了y属性，在o中找到了x属性
         //如果q中有这个属性就不会再向上找了，就是说可以有与原型相同的属性但是不同的值（相同属性名会覆盖，但是原型的属性不会被更改）
```

属性赋值操作先检查原型链，以此判定是否允许赋值操作。例如，如果o继承自一个只读属性x，那么赋值操作时不允许的。

如果允许属性赋值操作，它也总是在院士对象上创建属性或对已有的属性赋值，而不会去修改原型链。

在js中，只有查询属性时才会体会到继承的存在，而设置属性和继承无关，这是js的一个重要的特性，该特性让程序员可以有选择的覆盖继承的属性

```
var a={x:1}
var b=inherit(a)
b.y=2
b.x=5;//覆盖了原来的属性
a.x //=>1,原型的属性值没有被修改
```

如果想获取一个对象的属性的长度但又不知道这个属性有没有被定义可以

```
var len=book&&book.want&&book.want.length  //前两个是看对象和属性值是否存在，如果都存在的话是布尔型的值为1，与长度相与还是长度
```

给内置的构造函数赋值会失败但是不报错

```
Object.prototype=0 //不会报错但也不会进行更改
```

删除属性

delete运算符可以删除对象的属性，它只是断开了属性和宿主对象之间的联系，而不会去操作属性中的属性值，delete只能删除自有属性，而不能删除继承属性，要想删除继承属性得从这个属性的原型上去删除，这样会影响到所有继承自这个对象的对象

```
a={p:{x:1}};
b=a.p
delete a.p //这样之后b.x的值仍为1
```

这样已删除的属性的引用依然存在，js在实现时可能会因为这种不严谨的代码而造成内存泄漏，所以在销毁对象的属性时，要遍历属性中的属性，依次删除

（因为delete只是断开了链接，这个属性的名字和属性值还在，就可以被b继续调用）

当delete删除成功后或是没有产生任负作用时，会返回一个true

```
delete o.x  //删除成功，返回ture
delete o.x //x属性已经被删除了，无意义，但是也返回true
delete 1 //无意义，但是也返回ture
```

不能删除不可配置的属性，不能删除全局变量或是全局函数

检测属性

js对象可以看成属性的集合。我们经常会检测集合中成员的所属关系来判断某个属性是否存在于某个对象中。

可以通过in运算符，hasOwnPreperty()和propertyIsEnumerable()方法来完成这个工作，甚至仅通过属性查询也能做到这一点。

以下都不能看继承属性，包括in,但是不知道为啥in能看toString...

```
var pp={x:1,y:2,z:3}
var oo=inherit(pp)
oo.z=5
```

此时看oo的属性就只有z没有x和y，但是如果写oo.x也能返回1，就是使用下面的几种检测方法都查不到x和y除了最后一种!==undefined.最后一种能向上查继承属性

- In运算符：

  ```
  var o={x:1}
  x in o //true
  y in o //false
  "toString" in o //true o继承toString的属性
  ```

- hasOwnProperty()：这个方法用来检测给定的名字是否是对象的自有属性，对于继承属性它将返回false。如o.hasOwnProperty("x")为true，o.hasOwnProperty("toString")为false

  这上下两个方法的属性值必须是字符串，得加引号

  （如果是从对象中提取出的属性本身就是字符串了，就不用带了，如for (i in o) i为属性就已经是字符串了）

  这个与和下面那个与in的区别为**in操作符只要通过对象能访问到属性就返回true。hasOwnProperty()只在属性存在于实例中时才返回true。**

  实例化和继承是两个东西，实例就是new了一个新对象。

- propertyIsEnumerable()：是上面那个的升级版，当待检测属性是可枚举的时候才会返回True

  ```
  var o=inherit({y:2})
  o.x=1;
  o.propertyIsEnumerable("x") //true
  o.propertyIsEnumerable("y") //false y是继承来的
  ```

- 还有一种更简单的方法是使用!==来判断一个属性是否为undefined （这种方法可以查继承属性）

  ```
  var o={x:1}
  o.x!==undefined //true  o中有属性x
  o.y!==undefined //false o中没有属性y
  ```

  注意是!==而不是!=,两个等号可以区分出undefined和null，出现null时表示为定义了属性但是没赋值，undefined就是没有定义这个属性,有时可以不做区分

  ```
  if(o.x !=null) o.x*=2 //如果o有x这个属性且这个属性的值不为空则乘2
  if(o.x) o.x*=2 //若x不是undefined,null,false," ",0或NaN则乘2
  ```

枚举属性

我们经常会遍历对象的所有属性，通常使用for in来循环遍历，ES5中提供了两个更好用的方法

```
var o={x:1,y:2,z:3}
for(p in o) console.log(p) //通过for in循环来遍历

for (p in o){
    if(!o.hasOwnProperty(p)) continue //跳过继承自原型的属性
}
for (p in o){
    if(typeof o[p]==='function') continue //跳过方法
}
```

以下为几个有用的工具函数

```
//复制对象p的所有属性到对象o
function  extend(o,p){  //将对象p的所有属性给o，返回值为对象o（单单执行这个函数就已经改变了对象o的属性了，同时也将对象o返回）
  for(prop in p){
    o[prop]=p[prop]
  }
  return o
}


//复制对象p的所有属性到对象o，如果有重名属性则保留o的属性
function merge(o,p){  //过滤掉o中已存在的属性，只是将o中不存的的p存在的属性给o（不要同名的属性来覆盖本身的属性）
  for(prop in p){
    if(o.hasOwnProperty(prop)) continue; //有同名属性则跳过这个循环，继续下个循环，看下个属性
    o[prop]=p[prop]
  }
  return o
}


//只留下o中与p重名的属性
function restrict(o,p){ //如果o中的属性在p中没有同名属性则从o中删除这个属性
  for (prop in o){
    if(!(prop in p)) delete o[prop]
  }
  return o
}


//删除所有与p重名的o的属性
function subtract(o,p){   //如果o和p存在同名属性，删除o的同名属性
  for (prop in p) delete o[prop]  //因为删除一个空属性不会报错，所以可以这么写
  return o
}

//求o和p的并集，重名对象使用p的属性值
function union(o,p){return extend(extend({},o),p)}
    //返回一个新对象，这个对象拥有在o和p出现过的所有属性（就像是求o和p的并集，如果有重名属性则使用p的属性）
```

```
//求o和p的交集，重名对象使用o的属性
function  intersection(o,p){return restrict(extend({},o),p)}
    //返回一个新对象，这个对象的属性为o和p都有的属性(就像是求o和p的交集，如果有重名属性则使用o的属性)
//之所以要用extend是创建一个新对象来返回，这样的话就不会改变o对象的值了。


//返回一个对象的所有属性名，以数组的形式返回
function keys(o){//返回一个数组，数组中为o中可枚举的所有自有属性的名字
  if(typeof o!=='object') throw  TypeError() //参数必须是对象
  var result=[]  //这是要返回的数组
  for(var prop in o){
    if(o.hasOwnProperty(prop))//判断是否是自有属性
      result.push(prop) //将属性名添加到数组中
  }
  return result
}
```

属性getter和setter

对象是由名字，值和一组特性(attribute)构成的。属性的值可以用getter或是setter代替，由这两种方法定义的属性称作**存储器属性**，它不同于数据属性，数据属性只有一个简单的值。

而储存器属性不具有可写性，如果属性同时具有getter和setter方法那它就是一个读/写属性，如果它只有getter方法name它只有一个可读属性，如果只有一个setter方法，那它是一个只写属性，读取只写属性时总是返回undefined

定义存储器属性最简单的方法时使用对象直接量语法的一种拓展写法：

```
var o={
    data_prop:value,//普通的数据属性

    //存储器属性都是成对定义的函数
    get accessor_prop(){/* 这里是函数体*/},
    set accessor_prop(value){/*这里是函数体*/}
}
```

注意，这里没有使用冒号将属性名和函数体分隔开,但在函数体的结束和下一个方法或数据属性之间有逗号分隔。

函数体结束要加逗号

存储器属性是可以继承的

存储器属性的应用：智能检测属性的写入值以及在每次属性读取时返回不同值：

```
//这个对象产生严格自增的序列号
var serialnum={
    //这个属性包含下一个序列号
    $n:0, //$符号暗示这是一个私有属性

    get next(){return this.$n++;},//返回当前值然后自加
    set next(n){
        if(n>=this.$n) this.$n=n;
        else throw "序列号的值不能比当前值小"
    }
}
```

每次调用serialnum对象的next方法时，相当于读取next属性，就看get，get使返回当前的n并且n加1。

想给序列号n通过next()方法赋值时(如next(5)就是想把n赋成5)就看set，set会判断你想设定的值和当前值的大小，如果你想设定的值大的话就将n的值改为你设定的值，否则抛出异常

简单来说就是每次出现`serialnum.next`;都会使n的值加一(可以通过`serialnum.$n`来查看)。如果想给n赋值且给定一些条件（如不能小于现在的n）就可以通过`serialnum.next=n`来赋值

（也可以直接根据serialnum.$n来修改，但是这样就没法加条件了）

属性的特性：

属性除了包含名字和值以外，属性还包含一些表示它们本身是否可写，可枚举和可配置的属性。

所有创建的属性都是可写的，可枚举的和可配置的，且无法对这些特性做修改。本节讲述ES5中查询和蛇者这些特性属性的API，这些API非常重要，因为：

- 可以通过这些API给对象原型添加方法，并将他们设置成不可枚举的，这让他们看起来更像内置方法
- 可以通过这些API给对象定义不能修改或删除的属性，借此锁定这个对象

在本节中，我们将存储器属性的getter和setter方法看成是属性的特性，同样我们可以把数据属性的值同样看成属性的特性。因此，可以认为一个属性包含一个名字和4个特性。数据属性的四个特性是：

- 它的值
- 可写性
- 可枚举性 :指的是使用枚举属性那一章的方法是否可以看到这个属性（如使用for p in o），p只会显示出可枚举的属性，而获取不了不可枚举的属性。
- 可配置性

存储器属性不具有值特性和可写性，它们的可写性是由setter方法存在与否决定的，所以存储器属性的四个特性是

- 读(get)
- 写入(set)
- 可枚举
- 可配置

为了实现属性特性的查询和设置操作，ES5中定义了一个名为**“属性描述符”**的对象，这个对象代表那4个特性。描述符对象的属性和它们所描述的属性特性是重名的。因此，数据属性的描述符对象的属性有**value**,**writable**,**enumerable**,**configurable**。

存储器属性的描述符对象则用get属性和set属性代替`value`和`writable`。其中`writable`,`enumerable`,`configurable`都是布尔值。get属性和set属性是函数值。

通过调用`Object.getOwnPropertyDescriptor(a,b)`可以获得某个对象a的某个特定属性b的属性描述符：

```
Object.getOwnPropertyDescriptor({x:1},'x')//返回值为{value: 1, writable: true,
enumerable: true, configurable: true}

//如果是存储器属性，则前两个属性描述符为get和set
//对于继承和不存在的属性返回undefined
```

可以看出Object.getOwnPropertyDescriptor（）只能获得自有属性的描述符，如果要想获得继承属性的特性，需要遍历原型链（Object.getPrototypeOf()）

如果想要设置属性的特性，或是想让新建的属性具有某种特性，需要调用Object.defineProperty(对象，'属性',{属性描述符})。传入要修改的对象，要创建或修改的属性的名称以及属性描述符对象：

```
var o={} //创建一个空对象
Object.defineProperty(o,"x",{  //给对象o添加一个不可枚举的属性x
    value:1,
    writable:true,
    enumerable:false,
    configurable:true
});

o.x //=>1
Object.keys(0) //=>[]   属性是存在的，但不可枚举（for/in等方法获取不到x这个属性）

Object.defineProperty(o.'x',{writable:false}) //如果定义对象o的属性x为只读，则对x进行更改时会失败但是不报错
o.x=2 //失败但是不报错

//此时属性依然是可配置的，因此可以通过描述符这种方式对value的值进行更改
Object.defineProperty(o,"x",{value:2})

//现在将x从数据属性修改为存储器属性：
Object.defineProperty(o,'x',{get function(){return 0}})
o.x //=>此时x属性的值就为0了
```

注意，这个方法要么修改已有属性要么修改新建属性，不能修改继承属性。

如果要同时修改或创建多个属性，则需要用

```
Object.defineProperties({}, {
   x:{value:2,writable:true},
   y:{value:3,writable:false},
   r:{get function(){return 0},
        enumerable:true}
})
```

这段代码从一个空对象开始，给他添加两个数据属性和一个只读的存储器属性，Object.defineProperties返回修改后的对象。

如果对象是不可拓展的，则可以编辑已有的自有属性，但不能给他添加新的属性，

如果属性时不可配置的，则不能修改它的可配置性和可枚举性（存储器属性则不能修改get和set）。不能将它的可写性从false改为true

如果数据属性是不可配置且不可写的，则不能修改它的值，但可配置但不可写的可以先通过配置将其改为能写的再改值。

之前实现了一个extend函数可以把对象复制到一个新的对象中，但是这个复制对于属性的描述符并不能进行复制，所以可以对extend方法进行升级

```
Object.defineProperty(Object.prototype,"extend",{ //给Object.prototype添加一个不可枚举的extend方法
  writable:true,
  enumerable:false,//定义为不可枚举的
  configurable:true,
  value:function(o){
    var names=Object.getOwnPropertyNames(o)
    for (var i=0;i<names.length;i++){ //遍历所有的自有可枚举属性
      if(names[i] in this) continue ;//如果属性存在，则跳过
      var desc=Object.getOwnPropertyDescriptor(o,names[i])//获得对象o的所有的属性的描述符
      Object.defineProperty(this,names[i],desc)
    }
  }
})
```

对象的三个属性

每个对象都有与之相关的原型（prototype）,类(class),和可拓展性(extensible attribute)

property是属性，prototype是原型

原型属性:

对象的原型属性是用来继承属性的，原型属性实在实例对象创建之初就设置好的。

- 通过对象直接量创建的对象使用Object.prototype作为它们的原型
- 通过new创建的对象使用构造函数的prototype属性作为它们的原型（如Data.prototype等）
- 通过Object.create()创建的对象使用其第一个参数作为它们的原型（可以是null）

在ES5中，可以通过Object.getPrototypeOf()可以查询它的原型

要想检测一个对象是否是另一个对象的原型（或处于原型链中），请使用a.isPrototypeOf(b)方法来判断b是不是继承自a（a是不是b的原型）

注意：isPrototypeOf() 和instanceof运算符非常相似

类属性:

对象的类属性(class attribute)是一个字符串，用以展示对象的类型信息，只有一种简介的方法可以查询它。默认的toString()方法，返回如下格式的字符串

```
[object class]
```

因此，要想获得对象的类，可以调用对象的toString（）方法,然后提取已返回字符串的第8个到倒数第二个位置之间的字符。所以定义一个classof()函数来进行类的判断

可拓展性：

对象的课拓展性用以表示是否可以给对象添加新属性。所有内置对象和自定义对象都是显式可拓展的，除非将它们转换为不可拓展的。

ES5中定义了用来查询和设置对象可拓展性的函数。通过将对象传入Object.exExtensible()来判断对象是否是可拓展的。如果想将对象转换为不可拓展的，需要调用Object.preventExtensions()将待转换的参数传进去。

注意：

1.一旦将对象转换为不可拓展的，就再也无法将其转回可拓展的了

2.Object.preventExtensions()只影响对象本身的可拓展性，**如果给一个不可拓展的对象的原型添加属性，这个不可拓展的对象同样会继承这些新属性**

可拓展属性的目的是将对象锁定，避免外界的干扰。对象的课拓展性通常和属性的可配置性与可写性配合使用。

Object.seal()和Object.preventExtensions()类似，Object.seal除了能将对象设置为不可拓展的，还可以将对象所有的自有属性都设置为不可配置的。

也就是说不能给这个对象添加新属性，而且它已有的属性也不能删除或配置，不过它已有的可写属性依然可以设置。对于那些已经封闭(sealed)的对象是不能解封的。可以通过Object.isSealed()来检测对象是否封闭。

Object.fressze()将更严格的锁定对象--冻结。除了将对象设置为不可拓展的和将其属性设置为不可配置的意外，还可以将它的自有的所有数据属性设置为只读，通过Object.isFrozen()来检测对象是否冻结。

Object.preventExtensions(),Object.seal()和Object.fressze()都返回传入的对象，可以通过函数嵌套的方式调用它们

```
var o=Object.seal(Object.create(Object.freeze({x:1}),{y:{value:2,writable:true}}))
      //创建一个封闭对象包括一个冻结的原型和一个不可枚举的属性
```

序列化对象：

对象序列化是指将对象的状态转换为字符串，也可以将字符串还原为对象。ES5中提供了内置函数JSON.stringify()和JSON.parse()用来序列化和还原js对象。这些方法都使用JSON作为数据交换的格式。JSON的全称是JS对象表示法,它的语法和js对象与数组直接量的语法非常相近

要注意的是

- JSON.parse()依然只保留字符串形式而不会将它们还原成原始日期对象
- 函数，正则表达式，Error对象和undefined不能序列化和还原

- JSON.stringify()只能序列化对象可枚举的自有属性，对于一个不能序列化的属性来说，在序列化后的输出字符串中会将这个属性省略掉

对象的方法：

所有js的对象都从Object.prototype继承属性（除了那些不是通过原型显式创建的对象）,上文已经提到过一些对象的方法了，如下：

- hasOwnProperty() :看一个对象中是否有某个自有属性
- propertyIsEnumerable():看这个对象中的这个属性是不是可枚举的
- isPrototypeOf()：看这个对象是不是那个对象的原型
- Object.create():创建对象
- Object.getPrototypeOf():查询一个对象的原型

本节将对定义在Object.prototype里的对象方法展开讲解（一些特定的类会重写这些方法）

toString()方法：

没有参数，它将返回一个调用这个方法对象值的字符串。在需要将对象转换为字符串的时候，js都会调用这个方法。

默认的toString()方法返回值所带的信息很少(但在检测对象的类型(class)时非常有用)。因此很多类都带有自定义的toString。

- 如当数组转换为字符串时，结果是一个数组元素列表，只是每个元素都转换成了字符串。
- 再比如当函数转换为字符串的时候，得到的是函数的源代码。

有`Array.toString()`,`Data.toString()`以及`Function.toString()`

toJSON()方法：

Object.prototype实际上并没有定义这个方法，但对于需要执行序列化的对象来说，JSON.stringify()方法会调用toJSON()方法.如果在待序列化的对象中存在这个方法则调用它，返回值是序列化的结果，而不是原始的对象。

valueOf()方法：

与toString()方法非常相似，但往往当js需要将对象转换成为某种原始值而非字符串的时候才会调用它，尤其是转换为数字的时候。

如果在需要使用原始值的上下文中使用了对象，js会自动调用这个方法。和toString（）一样，一些类中内置定义了valueOf（）方法如Data.valueOf()

数组

数组是值的有序集合，每个值叫做一个**元素**，而每个元素在数字中有一个位置，以数字表示，称为**索引**。JS中数组是无类型的，数组元素可以是任意类型，数组元素甚至也可能是对象或其他数组。JS的数组是动态的，根据需要它们会增长或缩减，并且在创建数组时无需声明一个固定的大小。

JS的数组可能是稀疏的，数组元素的索引不一定要连续。

每个数组都有一个length属性，针对非稀疏数组，该属性就是数组元素的个数。

通过索引访问数组元素一般比通过访问常规的对象属性要快很多。

数组继承自Array.prototype中的属性。

创建数组是通过中括号[ ]来创建数组，数组直接量可以是任意的表达式

```
var i=0
var a=[i+1,true,,"a",[4,{x:1}]] //数组中可以有任何东西还可以嵌套，也可以为空如
a[2]=undefined

var a=new Array(2,4,1,'aa')//也可以通过这种方式来定义数组，如果参数有多个则就是定义数组的
元素，如果参数只有一个就是定义一个参数长度的空数组
var b=new Array(5) //定义一个长度为5的空数组
```

注意：数组可以使用负数或非整数来索引数组，此时数值将转换为字符串，以这个字符串作为属性名来用，此时这就是常规对象的属性而不是数组的索引了

但是如果使用非负整数的字符串那就可以直接当做索引来用

```
a[-1.23]=true //对a这个数组对象创建一个属性名为'-1.23'的属性且其值为true
a['1000']=0  //就相当于a[1000]=0
a[1.0000]   //就相当于a[1]
```

越界：当试图查询任何对象中不存在的属性时(如就定义到了a[10]去查询a[11])此时不会报错，但是得到的值为undefined，对象中也存在这种现象。

数组是对象那么就可以从原型中继承元素，在ES5中数组可以定义元素的getter和setter方法。

稀疏数组：

通常数组的length值代表数组中元素的个数，但如果数组是稀疏数组则length的值大于数组个数

注意，当数组直接量中省略值并不会创建稀疏数组，省略的元素在数组中是存在的，其值为undefined,可以用in 操作符检验两者之间的区别

```
var a1=[,,,] //数组是[undefined,undefined,undefined]
var a2=new Array(3) //该数组长度为3但是没有元素
0 in a1 //=>true a1是有0这个属性的，在索引0处有值undefined
0 in a2//=>false a2在索引0处没有元素

var a=[2,] //则a的长度为1，a中没有1这个属性
```

在存在连续逗号的时候，就相当于插入undefined。如[1,,3]和[1,undefined,3]是一模一样的

如果对数组进行跳位赋值隔得位为空(empty)而不是undefined,此时就是稀疏数组了如

```
var a=[1,2,3]
a[4]=3 //此时数组a为[1,2,3,empty,3]
3 in a //=>false 没有3这个属性
4 in a //=>true 有4这个属性
```

数组的长度

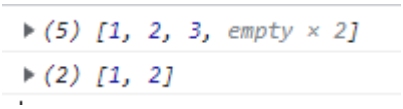
每个数组都有一个length属性，它是随着数组的变化动态变化的。

我们还可以对length的值进行更改，可以增加也可以删减，如果设置的length值大于当前的长度会在数组的尾部创建一个空的区域，当设置的length的值小于当前的length时就会将大于length-1的值全删了，如下:

```
var a=[1,2,3]

a.length=5
console.log(a)
a.length=2
console.log(a)
```

结果如下:



我们也可以对length这个属性设置属性符，如将它改为只读

```
a=[1,2,3]
Object.defineProperty(a,'length',{writable:false}) //此时对length做修改就不起作用了，就相当于与将数组给锁了，无法进行元素的增添了。
```

数组元素的增加删除

添加数组元素萨最简单的方法：为新的索引值赋值如a[4]='aa'

也可以使用push（）方法在数组末尾增加一个或多个元素

```
a=[]
a.push("zero","two")//在数组a的末尾添加两个元素,在末尾添加新元素等价于给a[a.length]赋值
```

可以像删除对象属性一样使用delete运算符来删除数组元素：

```
a=[1,2,3]
delete a[1]
1 in a//false 数组索引1并不在数组中定义
a.length//=>3:delete操作并不影响数组长度
```

删除数组元素与为其赋值为undefined是类似的，注意使用delete操作符并不会修改数组的length属性，也不会将元素从高索引处移下来来填充。

- 我们还可以通过设置数组的length值来删除数组末尾的元素。
- 数组还有pop()方法，它和push()一起使用，pop以此使数组长度减一，删除最后一个元素并返回被删除元素的值
- 还有一个shift()方法（和unshift()一起使用），从数组头部删除一个元素，但是与delete不同的是shift()将所有的元素向下移1
- sploce()是一个通用的方法来插入，删除或是替换数组元素，它会根据需要修改length属性并移动元素到更高或较低的索引处

数组遍历

使用for循环是遍历数组元素最常见的方法

```
for (var i =0;i<a.length;i++){} //这样的话每次都要查询数组长度，可以优化一下
for (var i=0,len=a.length;i<len;i++){ //这样的话只需要查询一次数组的长度就可以了
 if(!a[i]) continue //跳过null,undefined和不存在的元素
 if(a[i]===undefined) continue //跳过undefined和不存在的元素
 if(!(i in a)) continue //跳过不存在的元素
}
```

还可以使用for/in循环处理稀疏数组：

```
for(var index in sparseArray){
    var value=sparseArray[index] //此时就获取了稀疏数组的索引和值了
}
//因为for in 可以枚举继承的属性名如
var a=[1,2,3]
var b=Object.create(a)  //以a为原型创建b
console.log(b) //此时b没有自有属性，为空{}
for(i in b) console.log(b[i])//但是b[0],b[1]都是存在的，可以直接查询使用b[1]或者用for
in来遍历继承的属性名

for(var i in a){
    if(!a.hasOwnProperty(i)) continue //这样就可以跳过继承的属性
}
```

通常数组的遍历实现是升序的，但不能保证一定是这样的。特别的如果数组同时拥有对象属性和数组元素，name返回的属性名很可能是按照创建的顺序而不是索引数值的大小顺序。如果算法依赖于遍历的顺序，最好不要使用for in循环而使用常规的for循环。

多维数组

js不支持真正的多维数组，不过可以使用数组的数组来近似。访问数组中的数组只要简单的使用两次中括号[]操作符即可，如a[x][y]

数组方法：

1.join()

Array.join()方法将数组中的所有元素都转换为字符串并链接到一起返回生成的字符串，可以指定一个可选的字符串来在生成的字符串中来分隔各个数组元素，如果不指定则默认使用逗号

```
var a=[1,2,3]
a.join() //=>返回'1,2,3'，默认使用逗号分隔
a.join("")//=>返回'123'

//还可以使用join来进行填充
var b=new Array(5) //创建一个长度为5的空数组
b.join('-')//=>返回'----'
```

Array.join()方法是String.split()方法的逆操作，splite是将字符串按照一定的分割方式将字符串分割成数组

2.reverse()

这个方法将数组中的元素颠倒顺序，返回逆序的数组。它使用的是替换，并没有创建新的数组，而是在原先的数组中重新排列它们如321=>123

3.sort()

这个方法是将数组按照一定的顺序进行排序并返回排序后的数组。当不带参数调用sort（）时，数组元素以字母顺序排序

```
var a=new Arrray('banan','apple','cherry')
a.sort() //返回['apple','banana','cherry']
```

如果数组包含undefined元素，它们会排到数组的尾部

如果数组同时包含大小写的英文字符串，则默认先排大写再排小写

如果想要按照其他方式而非字母表顺序进行排序时必须给sort()方法传递一个比较函数。该函数决定了它的两个参数在排好序的数组中的先后顺序

```
var a=[33,4,1111,222]
a.sort() //返回值按照字母表顺序：1111,222,33,4
a.sort(function(a,b){return a-b}) //按照函数的返回值大于0（a>b，即前面的首数字大于后面的首数字）来进行排序，结果为4,33,222,1111
```

如果要对sort（）设置参数则参数必须为函数

4.concat()

该方法创建并返回一个新数组，它的元素包括调用concat()的原始数组的元素和concat()的每个参数。如果这些参数中任何一个为数组，则链接的是数组的元素而不是数组本身。

注意：concat()不会递归扁平化的数组，也不会修改调用的数组

```
var a=[1,2,3]
a.concat(4,5) //=>返回[1,2,3,4,5],相当于a.concat([4,5])
a.concat([4,5],[6,7])//返回[1,2,3,4,5,6,7]
a.concat(4,[5,[6,7]])//返回[1,2,3,4,5,[6,7]]
//使用concat不会修改a本身，只是返回融合后的数组
```

5.slice()

该方法返回指定数组的一个片段或是子数组，它的两个参数分别指定了片段的开始和结束的位置。返回的第一个参数指定的位置和所有到但不含第二个参数指定的位置之间的所有数组元素。

如果只指定一个参数，返回的数组将包含从参数的位置到数组的末尾

如果参数出现负数，它表示相对于数组末尾的位置

```
var a=[33,4,1111,222]

a.slice(0,2) //返回33,4
a.slice(-2) //返回1111,222
//注意slice也不会改变原数组
```

6.splice()

该方法是在数组中插入或删除元素的常用方法，不同于以上的两种方法concat和slice，splice会改动调用的数组

它的第一个参数指定了插入或删除的起始位置，第二个参数定义了删除元素的个数。如果省略了第二个参数，则将从开始的那个参数之后全删了。

splice()返回一个由删除的元素构成的数组，被更改后的数组仍为原数组

```
var a=[1,2,3,4,5, 6,7,8]
a.splice(4) //返回[5,6,7,8] a变为[1,2,3,4]
a.splice(1,2)//返回[2,3],a为[1,4]
```

splice前两个参数指定了需要删除的数组元素，紧接着的后面的任意个数的参数指定为需要插入到数组中的元素

```
var a=[1,2,3,4,5]
a.splice(2,0,'a','b') //第二个位置开始不删元素，添加'a''b'两个元素，返回[].a为
[1,2,'a','b',3,4,5]
a.splice(2,2,[1,2],3) //返回['a','b'],a为[1,2,[1,2],3,3,4,5]
//splice可以同时进行删除和增加元素
```

7.push()和pop()

该方法将数组当做栈来使用，push是在数组的末尾添加一个或多个元素(push(1,2,3))，并返回新的数组长度。pop是删除数组的最后一个元素，减小数组长度并返回它删除额值

8.unshift()和shift()

类似于上面的push和pop,不一样的是unshift是在数组的头部而不是尾部进行元素的插入和删除操作。

unshift()在数组的头部添加一个或多个元素，并将已存在的元素移动到更高索引的位置来获取足够的空间，最后返回数组的新长度。

shift()删除数组的第一个元素并将其返回，然后把随后的元素下移一个位置来填充数组头部的空缺

9.toString()和toLocalString()

该方法将其每个元素转化为字符串，并且输出用逗号分隔的字符串列表（和join不一样，这个分隔不能改，只能用逗号）

ES5中的数组方法：

ES5中定义了9个新的数组方法来遍历，映射，过滤，检测，简化和搜索数据。

大部分的ES5方法的第一个参数接收一个函数，并对数组的每个元素（或一些元素）调用一次该函数。大多数情况下调用的函数使用三个参数：**数组元素的值**，**元素的索引**和**数组本身**。通常只需要第一个参数值，可以忽略后两个参数。

如果有第二个参数，则调用的函数被看做是第二个参数的方法。也就是说在调用函数时传递进去的第二个参数作为它的this关键字的值来使用。

被调用的函数的返回值非常重要，但是不同方法处理返回值的方式也不一样.

ES5中的数组方法都不会修改它们调用的原始数组。

1.forEach()

从头至尾遍历数组，为每个元素指定的函数。传递的函数作为forEach()的第一个参数，然后forEach()使用三个参数调用该函数。如果只关心数组元素的值，可以编写只有一个参数的函数

```
var data=[1,2,3,4,5]
var sum=0
data.forEach(function(value){sum+=value}) //将每个值累加到sum上
sum              //为15
data.forEach(function(v,i,a){a[i]=v+1})//数组的每个元素自加一  三个参数分别是v:每个元素的值 i：索引 a：数组本身
data  //[2,3,4,5,6]
```

注意forEach无法在所有的元素都传给调用的函数之前停止遍历，像是break之类的都不好使，如果想提前终止就必须将foreach放到try块中

2.map()

将调用数组的每个元素出啊递给指定的函数，并返回一个数组，它包含该函数的返回值，例如：

```
a=[1,2,3]
b=a.map(function(x){return x*x}) //b为[1,4,9]  就是将a中的每个元素的值当做参数输入进函数中返回值为b对应位置的元素的值
```

和forEach不同的是map需要有返回值，map返回的是新数组不会修改原来的数组。如果输入为稀疏数组输出也是稀疏数组

3.fliter()

该方法返回的数组元素时调用的数组的一个子集，传递的函数是用来逻辑判定的，返回为true 或是false.如果判断为true或是能转换为true的值，那么当前元素就是这个子集的成员，他将被添加到一个作为返回值的数组中

```
a=[5,4,3,2,1]
smallvalues=a.fliter(function(x){return x<3}) //[2,1]
everyother=a.fliter(function(x,i){return i%2==0}) //[5,3,1] 就是选出能被2整除的索引（a[0],a[2],a[4]）
```

4.every()和some()

是数组的逻辑判定，它们对数组元素应用指定的函数进行判定，返回true或false

```
a=[1,2,3,4,5]
a.every(function(x){return x<10}) //=>true所有的元素的值都小于10
a.every(function(x){return x%2==0})//=>false 不是所有的元素的值都是偶数
```

some()方法就像数学中的'存在'；当数组中至少有一个元素符合判定条件则返回true，所有元素都不符合则返回false

```
a.some(function(x){return x%2==0})//=>true a中包含偶数
a.some(isNaN) //false a中不包含非数值元素
```

every()在遇见第一个false时停止遍历，some()在遇见第一个true时停止遍历

在空数组上进行遍历时，every返回true，some返回false

5.reduce()和reducceRight()

使用指定的数组元素进行组合，生成单个值，也可称为 '注入' 和 '折叠'

```
var a=[1,2,3,4,5]
var sum=a.reduce(function(x,y){return x+y},0) //数组求和
var product=a.reduce(function(x,y){return x*y},1)//数组求积
var max=a.reduce(function (x,y){return (x>y)?x:y})//数组求最大值
```

reduce需要两个参数，第一个时执行化简操作的函数，化简操作就是用某种方法把两个值组合或者化简为一个值并返回化简后的值。

第二个参数（可选）一个传递给化简函数的初始值，(求和就从0开始加，求积就从1开始乘)，当不设置初始值时它将使用数组的第一个元素作为其初始值

在空数组上不设置初始值调用reduce会报错，如果数组只有一个元素且没有设置初始值则仅仅只会返回那个值而不会进行任何操作

reduceRight和reduce差不多，不过reduceRight是从右向左进行处理

6.indexOf()和lastIndexOf()

搜索整个数组中具有给定值的元素，返回找到的第一个元素的索引，如果没有找到就返回-1.

indexOf()是从头到尾搜索，而lastIndexOf()是从尾到头搜索。

```
a=[0,1,2,1,0]
a.indexOf(1) //1
a.lastIndexOf(1)//3
a.indexOf(3) //-1 没有为3的元素值
```

不同于上面的方法，index不接受一个函数作为其参数。其第一个参数为所需要搜索的值，第二个参数是可选的。它指定数组中的一个索引，从那里开始搜索。如果省略第二个参数则从头开始搜索。对于lastIndexOf()第二个参数也可以是负数,代表与末尾的偏移量。

以下定义一个函数查找在数组中出现过的所有的x，并返回一个包含匹配索引的数组

```
function  findall(a,x){
  var result=[],len=a.length,pos=0;//分别为将会返回的索引值的数组，待搜索数组的长度以及起始位置
  while (pos<len){          //判断是否遍历完整个数组
    pos=a.indexOf(x,pos)    //寻找指定的值
    if(pos===-1)break       //若不存在则直接跳出循环返回空数组
    result.push(pos)        //若在则将索引压进结果中
    pos=pos+1               //位置加一再来一遍看看还有没有
  }
  return result
}
```

注意，字符串也有indexOf()方法和lastIndexOf()方法，功能与数组的类似

数组类型

在ES5中可以通过Array.isArray(a) 来判断a是不是数组

类数组对象

js中的数组有一些特性是其他对象所没有的：

- 当有新元素添加到列表中时，自动更新length属性
- 设置length为一个较小值将截断数组
- 从Array.prototype中继承了一些有用的方法
- 其类属性为"Array"

我们可以将拥有一个数组length属性和对应的非负整数的对象看成一种类型的数组，这种“类数组”虽然不能使用上述的方法也不能通过length完成一些操作，但是仍可以像真正的数组那样去遍历它们。