# 基础

### js输出

window.alert() 弹出警告框

document.write() 方法将内容写到 HTML 文档中（如果在文档已完成加载后执行 document.write，整个 HTML 页面将被覆盖。）

innerHTML 写入到 HTML 元素

console.log() 写入到浏览器的控制台

### js大小写敏感

### js中undefined,null,NaN的区别

typeof 用以检测变量的数据类型

|  |
| --- |
| var a1;  var a2 = true;  var a3 = 1;  var a4 = "Hello";  var a5 = new Object();  var a6 = null;  var a7 = NaN;  var a8 = undefined;  alert(typeof a); //显示"undefined" 数据类型：Undefined  alert(typeof a1); //显示"undefined" 数据类型：Undefined  alert(typeof a2); //显示"boolean" 数据类型：Boolean  alert(typeof a3); //显示"number" 数据类型：Number  alert(typeof a4); //显示"string" 数据类型：String  alert(typeof a5); //显示"object" 数据类型：Object  alert(typeof a6); //显示"object" 数据类型：Null  先声明一个变量，但还没有创建对象  alert(typeof a7); //显示"number" 数据类型：Number  alert(typeof a8); //显示"undefined" 数据类型：Undefined |

类型分析：

结论：未定义和定义未赋值的类型为undefined；null是一种特殊的object，NaN是一种特殊的number。

比较运算：

|  |
| --- |
| var a1; //a1的值为undefined  var a2 = null;  var a3 = NaN;  alert(a1 == a2); //显示"true"  alert(a1 != a2); //显示"false"  alert(a1 == a3); //显示"false"  alert(a1 != a3); //显示"true"  alert(a2 == a3); //显示"false"  alert(a2 != a3); //显示"true"  alert(a3 == a3); //显示"false"  alert(a3 != a3); //显示"true" |

结论：定义未赋值的变量值为undefined，undefined和null是相等的，NaN与任何值都不等，与自己也不等

重点：undefined表示：1、未声明的变量，2、声明但没赋值的变量，3、不存在的对象属性

null表示：空值。

所以判断一个变量是否未定义，必须用===undefined，因为==undefined只判断值，不判断类型，null和undefined的值是相等的！

undefined 与null值相等，但数据类型不同，一个为Undefined 类型，一个为Null类型

### js对象

var person={firstname:"John", lastname:"Doe", id:5566};

调用属性： name=person.lastname;

name=person["lastname"];

### 局部变量和全局变量

局部变量：在函数内部使用var声明的变量，只在函数内使用,函数执行完销毁

全局变量：在函数外声明的，网页上的所有脚本和函数都能访问，页面关闭后销毁

这里跟php不同，函数可以不用传参，直接使用全局变量

若向未定义的变量赋值，该变量将自动作为全局变量

在 HTML 中, 全局变量是 window 对象: 所有数据变量都属于 window 对象。

在函数体内，如果不用var 定义变量，那这个变量也是全局的；（不推荐）

|  |
| --- |
| person="John Doe";  function change(){  document.getElementById('demo').innerHTML=person;  document.getElementById('demo').innerHTML=window.person;  } |

变量包含两种不同数据类型的值：基本类型值，引用类型值。

基本类型值：保存在栈内存中的简单数据段，完全保存在内存中的一个位置。基本类型值有这几种: Undefined、 Null、 Boolean、 Number和 String，他们占有固定的内存空间。

引用类型值：保存在堆内存中的对象，但是内存地址保存在栈内存，这样当查询引用类型的变量时，先从栈内存读取内存地址，然后通过地址找到堆中的值。



复制变量:

对于基本类型，复制的是值本身；

引用类型复制的是地址；

|  |
| --- |
| var x = ‘box’; //在栈内存生成一个x ‘box’  var y = x; //在栈内存再生成一个 x ‘box’    x box  y box  栈内存  堆内存  x y 是互相独立的，互不影响了 |
| var x = new Object();  x.name = ‘June’;  var y = x; //把引用地址赋给y  x  y  Object  栈内存  堆内存 |

x y 指向的是同一个对象，若x.name被修改了，y.name也会跟着改变的

**作用域：**

**当调用一个函数内的变量时，首先会在函数体内部找，如果没有，则往当前函数的上一层找，逐层向上查找。**

### HTML事件

|  |  |
| --- | --- |
| **事件** | **描述** |
| onchange | 表单元素的内容改变时触发 |
| onclick | 用户点击 HTML 元素 |
| onmouseover | 用户在一个HTML元素上移动鼠标 |
| onmouseout | 用户从一个HTML元素上移开鼠标 |
| onkeydown | 用户按下键盘按键 |
| onload | 浏览器已完成页面的加载（用于body，页面图片及css等加载完触发） |

### 字符串

|  |
| --- |
| var carname = 'Volvo XC60';  //使用索引来访问字符串中的字符  var character = carname[7];  //使用length计算字符串长度  var sln = carname.length |

### 类型转换

typeof 操作符查看变量类型，他不是函数！

typeof ‘john’ string



请注意：

NaN 的数据类型是 number

数组(Array)的数据类型是 object

日期(Date)的数据类型为 object

null 的数据类型是 object

未定义变量的数据类型为 undefined

使用 constructor 属性来查看对象是否为数组

test.constructor

|  |
| --- |
| function isArray(myArray) {  return myArray.constructor.toString().indexOf("Array") > -1;  } |

全局方法：String（）强制转成字符串

|  |
| --- |
| String（123）; |

等同于Number方法toString()

|  |
| --- |
| var num =213;  var str = num.toString();  document.write(typeof str); |

区别：String（）可以转换null和undefined类型，但是toString（）不能。

类似的布尔、日期都有toString()方法

全局方法：Number() 将任何类型转换为数字

|  |
| --- |
| Number(" ") // 返回 0  Number("") // 返回 0  Number("99 88") // 返回 NaN |

一元运算符 +

|  |
| --- |
| var y = "5"; // y 是一个字符串  var x = + y; // x 是一个数字  //+不能转换字符串,但x还是数字类型的  var y = "John"; // y 是一个字符串  var x = + y; // x 是一个数字 (NaN)  var box = '您的年龄是： ' + 10 + 20; //您的年龄是： 1020，被转换成字符串  先计算前两部分，结果是字符串，后+20，结果还是字符串  var box = 10+ 20 + '是您的年龄'; //30是您的年龄，没有被转成字符  先计算前两部分，两个数字相加，结果是数字，30与字符串相加，结果是字符串。 |

布尔类型转数字

|  |
| --- |
| Number(false) // 返回 0  Number(true) // 返回 1 |

时间类型转数字

|  |
| --- |
| d = new Date();  Number(d) // 返回 1404568027739 |

专门用于把字符串转为数字：

parseInt()

|  |
| --- |
| alert(parsetInt('456Lee')); //456，会返回整数部分  alert(parsetInt('Lee456Lee')); //NaN，如果第一个不是数值，就返回NaN  alert(parseInt('12Lee56Lee')); //12，从第一数值开始取，到最后一个连续数值结束  alert(parseInt('56.12')); //56，小数点不是数值，会被去掉  alert(parseInt('')); //NaN，空返回Na |

parentFloat() 同parseInt类似；

自动转换：

|  |
| --- |
| 5 + null // 返回 5 because null is converted to 0  "5" + null // 返回"5null" because null is converted to "null"  "5" + 1 // 返回 "51" because 1 is converted to "1"  "5" - 1 // 返回 4 because "5" is converted to 5 |

自动转成字符串

当在比较运算时，null和undefined没有自动转换

|  |
| --- |
| undefined==0 false  null ==0 false |

当尝试输出一个对象或一个变量时，会自动调用变量的toString()方法；

|  |
| --- |
| document.getElementById("demo").innerHTML = myVar;  // if myVar = {name:"Fjohn"} // toString 转换为 "[object Object]"  // if myVar = [1,2,3,4] // toString 转换为 "1,2,3,4"  // if myVar = new Date() // toString 转换为 "Fri Jul 18 2014 09:08:55 GMT+0200" |

### 正则

stringObject.search(regex|string)

返回第一个与正则匹配到的子串的起始位置，无匹配返回-1；

|  |
| --- |
| var str = "Visit w3cschool";  var n = str.search(/w3cschool/i); js的正则表达式不加引号 |

参数也可以直接是字符串，字符串会转换成正则表达式。

此时，类似于字符串的indexOf()方法,返回首次出现的位置

|  |
| --- |
| var str = "Visit w3cschool!";  var n = str.search("w3cschool");  var n = str.indexOf("w3cschool"); |

stringObject.match(regex|string)

返回匹配到的结果的数组。

stringObjet.replace(regex,replacement)

返回替换后的新字符串

### 错误处理

* + 1. try catch

|  |
| --- |
| try  {  adddlert("Welcome guest!");  }  catch(err)  {  txt="本页有一个错误。\n\n";  txt+="错误描述：" + err.message + "\n\n";  txt+="点击确定继续。\n\n";  alert(txt);  } |

* + 1. throw

throw会抛出错误，在catch里会捕捉到错误err

|  |
| --- |
| try {  var x=*document*.getElementById("demo").value;  if(x=="") throw "empty";  if(isNaN(x)) throw "not a number";  if(x>10) throw "too high";  if(x<5) throw "too low"; } catch(*err*) {  var *y*=*document*.getElementById("mess");  *y*.innerHTML="Error: x is " + *err* + "."; } |

### 调试

* + 1. 断点

调试模式的sources里，在页面的左边栏行号处点击，即为设置断点。

* + 1. debugger

在js里添加debugger，效果跟在调试模式的断点一样。

|  |
| --- |
| var x = 15 \* 5;  debugger;  document.getElementbyId("demo").innerHTML = x; |

### 表单验证



### Json

json语法规则：

数据为 键/值 对。

数据由逗号分隔。

大括号保存对象

方括号保存数组

example:

|  |
| --- |
| {"employees":[  {"firstName":"John", "lastName":"Doe"},  {"firstName":"Anna", "lastName":"Smith"},  {"firstName":"Peter", "lastName":"Jones"}  ]}  在以上实例中，对象 "employees" 是一个数组。包含了三个对象。  每个为个对象为员工的记录（姓和名）。 |

json字符串转换为javascript对象

|  |
| --- |
| var obj = JSON.parse(text); |



js中用 . 来访问对象属性，用 [] 访问数组元素。

### javascript:void(0)

void：计算一个表达式但是不返回值

|  |
| --- |
| function getValue(){   var a,b,c;   a = void ( b = 5, c = 7 );   document.write('a = ' + a + ' b = ' + b +' c = ' + c ); }  void不返回值，所以a没有赋值  a = undefined b = 5 c = 7 |

定义一个死链接：

<a href="javascript:void(0);">

但是javascript:void(0)在id中会引起一些问题，所以最安全的方法还是href=“####”

# 函数

### 提升

提升（Hoisting）是 JavaScript 默认将当前作用域提升到前面去的的行为。

所以，函数可以在声明之前就调用，没有顺序的问题。

|  |
| --- |
| myFunction(5);  function myFunction(y) {  return y \* y;  } |

### 自调用函数

|  |
| --- |
| (function () {  var x = "Hello!!"; // 我将调用自己  })();  前面括号内是个匿名函数，它就等于一个函数名，后面加个括号等于调用，也可以传递参数。 |

### arguments对象

js中的函数可以不传形参，函数体里使用arguments[i]来代表传入的参数。

函数显式参数在函数定义时列出。

函数隐藏参数(arguments)在函数调用时传递给函数真正的值。

即函数定义时function test(){…}没有传显式参数，但在函数调用时传递真实值。

arguments可以代表参数的集合

JavaScript 函数有个内置的对象 arguments 对象.

arguments 对象包含了函数调用的参数数组。

arguments.length 参数个数

arguments[i] 第几个参数

arguments.callee 指向函数名，用在递归函数里。

当把函数名改了，函数内部的函数名用arguments.callee表示，则不用一个一个的替换

### 函数调用

每种方式的不同在于 this 的初始化。

在js中，this指向函数执行时的当前对象。

* + 1. 作为一个函数调用

|  |
| --- |
| function myFunction() {     alert(this); } myFunction();           // [object Window]  以上函数不属于任何对象，但在js中它始终属于默认的全局对象；  在 HTML 中默认的全局对象是 HTML 页面本身，所以函数是属于 HTML 页面。  在浏览器中的页面对象是浏览器窗口(window 对象)。以上函数会自动变为 window 对象的函数。 所以这里this指的就是window  myFunction() 和 window.myFunction() 是一样的： |

* + 1. 作为对象方法调用

|  |
| --- |
| var myObject = {  firstName:"John",  lastName: "Doe",  fullName: function () {  return this.firstName;  }  }  myObject.fullName(); // 返回 "John Doe" |

fullName 属于myObject对象，所以this即为myObject对象

### 匿名函数和js闭包

闭包是指有权访问另一个函数作用域中的变量的函数。

常见的创建方式：在一个函数内部再创建一个函数；

|  |
| --- |
| function box() {  var name = 'box';  return function () {  return name;  }  }  alert(box()()); |

使用闭包可以把局部变量贮存在内存中，可以避免使用全局变量。但也有可能占用过多内存导致性能下降。

此时可以box = null来解除引用，释放内存；

闭包中的this，指的是window

|  |
| --- |
| var user = 'TheWindow';  var obj = {  user: 'The Object',  getUserFunction: function () {  return function () { //闭包不属于obj，里面的this指向window  return this.user;  }  }  };  alert(obj.getUserFunction()()); |

解决：

|  |
| --- |
| var obj = {  user: 'The Object',  getUserFunction: function () {  var that = this; //这里作用域的this是obj  return function () { //这里作用域的this是window  return that.user;  }  }  }; |

或者用对象冒充：

|  |
| --- |
| var user = 'TheWindow';  var obj = {  user: 'The Object',  getUserFunction: function () {  return function () { //闭包不属于obj，里面的this指向window  return this.user;  }  }  };  alert(obj.getUserFunction().call(obj)); |

**私有作用域：**

|  |
| --- |
| function box() {  for (var i=0;i<5;i++) {  }  alert(i);  }  box(); |

像if(){}、 for(){}这样的，它没有作用域，在外面还是可以访问到里面的变量。但是我们希望变量出了他们就该被销毁，这时使用自我执行的匿名函数，就可以实现私有作用域

|  |
| --- |
| function box() {  (function () {  for (var i=0;i<5;i++) {  }  })();  alert(i);  }  box(); |

通用：

|  |
| --- |
| (function (){  })() |

自动调用完里面的变量就自动销毁了，不用手工null了。

**私有变量和私有方法：**

|  |
| --- |
| function Box() {  var name = 'John'; //私有属性  function run() { //私有方法  return 'running';  }  this.name2 = name; //对外公共的特权属性  this.running = function () { // 对外公共的特权方法  return run();  }  }  var box = new Box();  alert(box.name2); |

但对象的方法，多次实例化会多次创建，可以用下面方法解决：

**在私有作用域中创建对外公共的方法：**

|  |
| --- |
| (function () {  var user = '';  Person = function (value) {  user = value;  };  Person.prototype.getUser = function () {  return user;  };  Person.prototype.setUser = function (value) {  user = value;  }  })();  var x = new Person('June');  alert(x.getUser()); |

以上对象的声明使用使用的是Person = function(){} ，而不是function Person(){},若使用后者就成私有函数了，无法再全局调用。

### 属性和方法

每个函数都有两个属性：length和prototype。

length：函数希望接收的参数的个数

protetype下有两个方法：apply()和call(),这两个函数都在特定的作用域中调用函数，实际上等于设置函数体内this对象的值。

|  |
| --- |
| var color = '红色的';  var box = {'color':'绿色的'};  function test() {  alert(this.color);  }  function test1() {  test.apply(box); //改变test函数的作用域，即改变test函数里this的值，原来//是全局作用域，现在改成box了，所以color也跟着变了  }  test1(); |

apply和call基本相同，只是apply（作用域，参数数组或arguments）

而call（作用域，参数1，参数2，…）

# DOM

### 概述

DOM树：包含文档（document）、元素（element）、属性（attribute）



查找HTML元素：

1. var x=document.getElementById("intro");
2. var x=document.getElementsByName(‘name’) 返回对象集合
3. var y=x.getElementsByTagName("p");返回对象集合
4. var x=document.getElementsByClassName("intro"); 返回对象集合

### 改变HTML内容

DOM操作只有等到html文档加载完毕才能获取，所以一般js都放到后面。

假如非要放前面可以使用：

window.onload = function(){

}

document.write(Date());

document.getElementById(id).innerHTML=new HTML

改变html元素属性

document.getElementById(id).attribute=new value attribute此处指具体的元素属性

### 改变CSS

document.getElementById(id).style.property=new style

style使用3.2中的attribute获取属性，后面的property继续获取具体的属性值

注意：

1. 这里如果property是类似于“font-size”这样的有短横线的，需要转成小驼峰式fontSize。

2. style只能获取行内样式，不能获取内联和外链式的样式。

### DOM事件

1）事件属性：

|  |
| --- |
| <button onclick="displayDate()">Try it</button> |

这里，onclick类似于style、id等，也属于元素的属性，只不过是事件类型的属性；

2）通过DOM分配事件

等同于通过DOM给元素添加了事件属性；

|  |
| --- |
| <script>  document.getElementById("myBtn").onclick=function(){displayDate()};  </script> |

3）常用事件

onclick

onload页面加载完触发

onchange改变输入字段的内容时触发

onmouseover鼠标移至元素上触发

onmouseout鼠标移除元素时触发

onmousedown鼠标按下时触发

onmouseup鼠标松开时触发

onclick 其实在鼠标松开时先触发onclick，后触发onmouseup。

onfocus 当元素获取焦点时触发

### 事件监听

element.addEventListener(event, function, useCapture);

第一个参数是事件的类型 (如 "click" 或 "mousedown").

第二个参数是事件触发后调用的函数。

第三个参数是个布尔值用于描述事件是冒泡（默认）（false）还是捕获（true）。该参数是可选的。

注意:不要使用 "on" 前缀。 例如，使用 "click" ,而不是使用 "onclick"。

冒泡：内部元素的事件先被触发，后触发外部元素的事件

捕获：外部元素的事件先被触发，后触发内部元素的事件

该属性只是添加监听，不会覆盖已存在的监听；

可以添加多个同类的事件监听，比如两个click事件

可以向任何DOM元素添加事件监听，不仅仅是HTML元素，还可以是window对象等

### DOM元素

1）创建新的 HTML 元素：

先创建该元素，然后向一个已存在的元素追加该元素。

|  |
| --- |
| //创建p元素  var para=document.createElement("p");  //创建文本节点  var node=document.createTextNode("This is a new paragraph.");  //将文本节点追加进p元素内  para.appendChild(node);  //获取div1元素  var element=document.getElementById("div1");  //将填充了文本的p元素追加进已经存在的div1元素中  element.appendChild(para); |

2）删除已有的HTML元素

|  |
| --- |
| <div id="div1">  <p id="p1">This is a paragraph.</p>  <p id="p2">This is another paragraph.</p>  </div>  <script>  var parent=document.getElementById("div1");  var child=document.getElementById("p1");  //先将父子两个元素获取到  parent.removeChild(child);  </script> |

注意：js中删除元素时必须先找到该元素的父元素，使用removeChild删除子元素。不能 不引用父元素的情况下直接删除子元素。

但是 ，可以先将子元素获取到，使用parentNode获取他的父元素：

|  |
| --- |
| var child=document.getElementById("p1");  child.parentNode.removeChild(child); |

### Document对象

Document对象是Window 对象的一部分，可通过 window.document 属性对其进行访问。

HTML对象都有标准属性：

|  |  |
| --- | --- |
| **属性** | **描述** |
| [className](http://www.w3school.com.cn/jsref/prop_classname.asp) | 设置或返回元素的 class 属性。 |
| [dir](http://www.w3school.com.cn/jsref/prop_dir.asp) | 设置或返回文本的方向。 |
| [lang](http://www.w3school.com.cn/jsref/prop_lang.asp) | 设置或返回元素的语言代码。 |
| [title](http://www.w3school.com.cn/jsref/prop_title.asp) | 设置或返回元素的 title 属性。 |

#### Document 对象集合

document.anchors 返回文档所有锚点的集合

//锚点即<a name=’a’></a> 有name属性的a链接

document.forms 返回对文档中所有 Form 对象的引用

|  |
| --- |
| document.forms[0].name |

document.images 返回文档找那个所有图片集合

#### Document对象属性

document.cookie 返回所有cookie的名/值对

document. domain 返回当前文档的域名

document. lastModified 返回文档被最后修改的日期和时间。

document. referrer 返回当前文档的上一级目录

document. title 返回当前文档的title，title标签里的内容

document.URL 返回当前文档的 完整URL。

document.body获取body标签

#### Document对象方法

document.open() 打开一个新文档，并擦除当前文档的内容。

|  |
| --- |
| var newDoc=document.open("text/html","replace");  var txt="<html><body>Learning about the DOM is FUN!</body></html>";  newDoc.write(txt); //向新文档写入内容  newDoc.close();//必须记住关闭文档， |

#### Anchor 对象（链接）

document.getElementById('w3s').href 返回a链接的href

document.getElementById('w3s').name返回a链接的name

document.getElementById('w3s').target= ‘\_blank’ 修改a链接的target

#### Button对象

|  |  |
| --- | --- |
| **属性** | **描述** |
| [disabled](http://www.w3school.com.cn/jsref/prop_pushbutton_disabled.asp) | 设置或返回是否禁用按钮。 |
| [form](http://www.w3school.com.cn/jsref/prop_pushbutton_form.asp) | 返回对包含按钮的表单的引用。向上找一层 |
| [id](http://www.w3school.com.cn/jsref/prop_pushbutton_id.asp) | 设置或返回按钮的 id。 |
| [name](http://www.w3school.com.cn/jsref/prop_pushbutton_name.asp) | 设置或返回按钮的名称。 |
| [type](http://www.w3school.com.cn/jsref/prop_pushbutton_type.asp) | 返回按钮的表单类型。 |
| [value](http://www.w3school.com.cn/jsref/prop_pushbutton_value.asp) | 设置或返回显示在按钮上的文本。 |

方法：

blur()

focus()

click()

#### Form对象

formObject.elements 返回表单里所有元素的数组

三种方法获取form里的某个元素：

|  |
| --- |
| formObject.elements[0] |
| formObject.elements[name] |
| formObject.name //name是input的name |

form对象属性：

formObject.length 返回表单里元素的数量（length是表单的属性，不需要获取元素的集合即可使用）

formObject.id

formObject. action

formObject. method

formObject.name

form对象方法

formObject.reset() 表单重置

formObject .submit() 表单提交

事件句柄

onsubmit和onreset

它们是表单的属性，指向一个事件句柄函数，当点击reset按钮时，会调用test()

|  |
| --- |
|  |

#### Image对象

属性：

align

alt

border

height 、width

id

name

src

事件句柄

onload onerror onabort

#### select 对象

formObject.options 下拉列表所有选项的集合

formObject.options[i] 选中第几个下拉选项

formObject.options[i].text 某一个下拉选项的文本

formObject.options.length 下拉列表共有几个选项

formObject.options.length=0；所有下拉选项都会被清除

select对象属性

|  |  |
| --- | --- |
| **属性** | **描述** |
| [disabled](http://www.w3school.com.cn/jsref/prop_select_disabled.asp) | 设置或返回是否应禁用下拉列表。 |
| [form](http://www.w3school.com.cn/jsref/prop_select_form.asp) | 返回对包含下拉列表的表单的引用。 |
| [id](http://www.w3school.com.cn/jsref/prop_select_id.asp) | 设置或返回下拉列表的 id。 |
| [length](http://www.w3school.com.cn/jsref/prop_select_length.asp) | 返回下拉列表中的选项数目。 |
| [multiple](http://www.w3school.com.cn/jsref/prop_select_multiple.asp) | 设置或返回是否选择多个项目。 |
| [name](http://www.w3school.com.cn/jsref/prop_select_name.asp) | 设置或返回下拉列表的名称。 |
| [selectedIndex](http://www.w3school.com.cn/jsref/prop_select_selectedindex.asp) | 设置或返回下拉列表中被选中项目的索引号  若多选，只返回第一个的下标 |
| [size](http://www.w3school.com.cn/jsref/prop_select_size.asp) | 设置或返回下拉列表中的可见行数。 |
| [tabIndex](http://www.w3school.com.cn/jsref/prop_select_tabindex.asp) | 设置或返回下拉列表的 tab 键控制次序。 |
| [type](http://www.w3school.com.cn/jsref/prop_select_type.asp) | 返回下拉列表的表单类型。 |

select对象方法：

selectObject.add(option) 向下拉列表添加一个选项

selectObject.blur() 从下拉列表移开焦点

selectObject.focus() 在下拉列表设置焦点

selectObject.remove( option ) 从下拉列表删除一个选项

事件句柄

onchange 当改变选择时调用

#### table对象

table对象集合：

tableObject.rows 返回表格中所有行的数组

tableObject.rows[i].rowIndex 返回某一行的索引下标，便于删除行

tableObject.row[0].celles 返回表格中某一行的所有单元格的数组

table对象属性：

tableObject.border 设置表格边框

tableObject.width 设置表格宽度 50% or px

tableObject.cellpadding 表格单元格内边距

tableObject.cellspacing 表格单元格外边距

tableObject.id 返回表格id

table对象方法：

tableObject.deleteRow( rowIndex )

注：rowIndex可以通过传入this或event通过获得当前对象，利用当前行的rowIndex属性获得

tableObject.insertRow(rowIndex)

注：在rowIndex这行前面插入一个新行，返回新行的rowObject；可以在新行里插入单元格，rowObject.insertCell(0)

|  |
| --- |
| var newRow = document.getElementById('tab').insertRow(0);  var cell1 = newRow.insertCell(0);  cell1.innerHTML='new cell1'; |

### Element对象(节点)

在 HTML DOM （文档对象模型）中，每个部分都是节点：

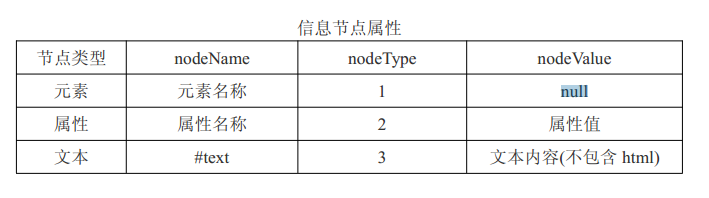
文档本身是文档节点

所有 HTML 元素是元素节点

所有 HTML 属性是属性节点

HTML 元素内的文本是文本节点

注释是注释节点



上图：元素节点只是这个元素标签，如<div></div> ，并没有nodeValue；

文本节点就是代表文本本身，即一段文字，它没有标签，所以nodeName标记为#text;

属性和方法：

document.createElement("p"); 创建元素节点

document.createTextNode("This is a new paragraph."); 创建文本节点

* + - * element.appendChild(node) 向元素添加新的子节点，作为最后一个子节点
      * element.removeChild(子节点) 父节点中移除子节点（先要获取子节点）
      * element.attributes 返回元素的所有属性 element.attributes.length
      * element.childNodes 元素子节点的集合

注意：集合中会包含有换行和空格，他们都将作为一个子节点存在，慎用！

element.childNodes[i] 返回第几个子节点

element.childNodes.length 返回子节点的个数

* + - * element.children 元素的所有子元素（忽略空白节点）
      * element.className 元素的class属性
      * element.cloneNode(true) 克隆元素 true表示连带复制里面的内容；

返回克隆的元素，true参数表示也克隆子节点

|  |
| --- |
| var clone = element.cloneNode(true);  document.body.appendChild(clone); |

* element.firstChild 返回首个子节点
* element.lastChild 返回元素最后一个子元素
* element.parentNode 返回父节点
* element.previousSlibing 返回上一个兄弟元素
* element.nextSlibing 返回下一个兄弟元素
* element.getAttribute(属性名) 获取属性值
* element.hasAttribute(属性名) 判断元素是否含有该属性
* element.removeAttribute(属性名) 移除属性
* element.setAttribute(属性名，属性值)
* element.id 返回元素id
* element.innerHTML 返回元素内容

innerHTML和nodeValue的区别：

1获取到文本节点时，无法使用innerHTML这个属性输出文本内容的；必须在获取到元素节点才能获取到innerHTML。

2 赋值时，nodeValue会转义文本里的特殊字符，而innerHTML则不会

|  |
| --- |
| box.childNodes[0].nodeValue= '<strong>abc</strong>';//结果: <strong>abc</strong>  box.innerHTML= '<strong>abc</strong>'; //结果为： abc |

* element.style 返回元素的样式
* existingItem.parentNode.insertBefore(newItem,existingItem) 在已存在的元素前插入新元素

parent 父元素

newItem 新建的元素

existingItem 父元素已经存在的子元素

注意：insertBefore其实跟appendChild一样，都是内部插入，只是insertBefore需要传入两个参数:新节点、在哪个节点前插入，所以一般情况下为了方便使用，都是先把需要在它前面插入节点的那个节点获取到，然后用element.parentNode.insertBefore(new,element);

* element.nodeName 返回节点的名称

若节点是元素，返回元素标签名

若节点是属性，返回属性名

* parent.contains(child);判断父节点里是否有该子节点。

### Element对象的移动

|  |
| --- |
| var x = document.getElementById('div');  x.style.width 只能获取行内样式，可读可写，若是通过class属性引用了一个外部样式表，x.style.width 获取不到该属性；  getComputedStyle(x,false).width 是个全局方法，获取对象自身的宽度（不含padding和border），不存在x.style.width的问题。仅可读  x.style.width 返回值为32px  若要使用，需要用parseInt()处理下，只保留数字  getComputedStyle(x,false).width 返回值 是32 ，不带单位px  同：他们都获取的是自身的宽度，不含padding和width  x. offsetWidth 包含width+padding+border 在有内边距和边框时慎用 仅可读 |

//元素的总宽高，包括溢出而无法展现在网页中的不可见部分

document.body.scrollWidth 网页正文全文宽

document.body.scrollHeight网页正文全文高

document.body.scrollTop 网页被卷去的高

document.body.scrollLeft 网页被卷去的左

例子：

|  |
| --- |
| <div id="test" style="width: 100px;height: 100px;overflow:scroll;line-height:200px;">内容</div> |

div好比一个容器，长宽100，但里面的内容行高200，超过了容器，溢出，此时出现滚动条，当滚动条下拉时

|  |
| --- |
| var x = document.getElementById(‘test’);//获取div元素  x. scrollHeight //div里内容的总高度，应该是200，包含不可见部分  var clientHeight = window.innerHeight  || document.documentElement.clientHeight  || document.body.clientHeight; // 可见区域的宽 （不含滚动条）  x.scrollTop; // 滚动条下拉的高度 |

### Event对象

* + 1. 事件句柄

事件通常与函数结合使用，函数不会在事件发生前被触发。

onabort 图像被中断时触发，即当 用户点了浏览器的×，停止加载时触发

onblur 失去焦点触发

onfocus 获取焦点时触发

onchange 域的内容被改变时触发（可能是文本框，单选框等等）

onclick ondbclick 单击/双击时触发

onerror 加载文档或图片时错误 触发

onkeydown 键被按下时触发

onkeyup 键被松开时触发

onkeypress 键被按下并松开触发

实例：

不允许输入数字：

|  |
| --- |
| //使用时都需要把event传入函数里  <input type="text" onkeydown="return myFunction (event)" />  function myFunction(e) {  var keycode;  if (window.event) { // IE  keycode = e.keyCode;  }else if (e.which) { // Netscape/Firefox/Opera  keycode = e.which;  }  var keyChar = String.fromCharCode(keycode);// Unicode转成字符串  if (keyChar.search(/\d/)>-1) {  return false;  }else {  return true;  }  } |

* + 1. 鼠标键盘属性

event.clientX event.clientY 鼠标在当前窗口的水平和垂直坐标

event.screenX event.screenY 鼠标在屏幕的水平和垂直坐标

event.keyCode 被敲击的键的 Unicode 字符码(IE)

event.which 被敲击的键的 Unicode 字符码(非IE)

* + 1. 标准Event属性

event.target 返回触发此事件的元素（事件的目标节点）

# 对象

JavaScript 中的所有事物都是对象：字符串、数值、数组、函数... 对象只是带有属性和方法的特殊数据类型。

访问对象属性：objectName.propertyName

访问对象方法：objectName.methodName()

使用for （var x in box）来循环遍历对象的属性名。

### 创建直接的对象实例

|  |
| --- |
| person=new Object();  person.firstname="John";  person.lastname="Doe";  person.age=50;  person.eyecolor="blue"; |

### 使用对象构造器

|  |
| --- |
| function Person(firstname,lastname,age,eyecolor) //一般使用大写，与普通函数做区别  {  this.firstname=firstname;  this.lastname=lastname;  this.age=age;  //对象的方法  this.changeName=function (name)  {  this.lastname=name;  };  } |

这里的this指的是正在执行的函数本身；

运行时，this一般用在对象的方法中，指向调用该方法的对象；

|  |
| --- |
| var oCar = new object();  cCar.showColor = function(){alert(this.color);};//this就指的oCar对象 |

有了对象构造器，可以这样创建对象：

var myFather=new Person("John","Doe",50,"blue");

### Number对象

1. Infinity 无穷大
2. - Infinity 无穷小
3. NaN 非数字值
4. isNaN() 一个值不是数字时返回true

### 字符串对象

1）属性：length 字符串长度

text.length

以下都是方法：

2）indexOf（char，start）字符串中某个指定的字符char首次出现的位置，start为开始检索的位置。

|  |
| --- |
| var str = "Hello world, welcome to the universe."; |
| str.indexOf(’world’) |

返回：索引下标，未找到返回-1；

3）lastIndexOf(char) 在字符串的末尾开始查找某字符出现的位置

4）match（char）

str.match("world") 同‘正则’章节相同，返回匹配到的字符；

1. replace（char,replacement）替换字符
2. toUpperCase() / toLowerCase() 大小写转换

str.toUpperCase();

1. split(separator) 将字符串以separator分割为数组；

|  |
| --- |
| txt="a,b,c,d,e" // String  txt.split(","); // Split on commas |

1. str.charAt(index) 返回指定位置的字符 index为索引下标
2. str.slice(m,n) 返回m到n之间的字符串 m n 都是索引

str.substring(m,n)同上

1. str.substr(m,length) 返回从m开始的length个字符
2. str.toUpperCase();大写

str.toLowerCase();小写

### 日期对象

1. var d = new Date() 获得当日的日期
2. Date() 也可以单独使用获取时间；
3. d. getFullYear() 获取年份
4. d. getTime() 返回 1970 年 1 月 1 日至今的毫秒数。
5. d.setTime() 以毫秒设置 Date 对象
6. d.setFullYear(2020,10,3); 设置具体的日期 //js里月份是0~11,0即是1月
7. d.getDay() 返回星期几 0-星期日 1-星期一…
8. d. getDate() 返回月份天数
9. d.getMonth() 返回实际月份-1

d.getHours() 返回小时

d.getMinutes()返回分钟

d.getSeconds()返回秒

1. 日期比较

|  |
| --- |
| var x=new Date();  x.setFullYear(2100,0,14);  var today = new Date();  if (x>today)  {  alert("Today is before 14th January 2100");  }  else  {  alert("Today is after 14th January 2100");  } |

1. 显示时钟

|  |
| --- |
| function time(){  var d = new Date();  var h = d.getHours();  var m = d.getMinutes();  var s = d.getSeconds();  m = checkTime(m);  s = checkTime(s);  document.getElementById('demo').innerHTML = h +':' +m+':'+s;  setTimeout("time()",500); //这里调用函数一定要加引号！！！  }  function checkTime(k) {  if(k<10){  return '0'+k;  }  return k;  } |

### 数组对象

1. 创建数组

|  |
| --- |
| var myCars=["Saab","Volvo","BMW"];  或var myCars=new Array("Saab","Volvo","BMW"); |

1. 访问数组

var name=myCars[0];

1. 方法与属性

arr.length 数组元素的个数

arr.indexOf(‘a’) a在数组arr中的下标

arr1.concat(arr2[,arr3]); 合并一个或多个数组

arr.join() 把数组拼成字符串

末尾：

arr.pop() 删除数组最后一个元素

arr.push(new) 在arr数组尾部添加元素new

开头：

arr.shift() 删除数组第一个元素

arr.unshift(new) 在数组开头插入一个元素new

arr.reverse() 数组反转

arr.slice(start,end) 截取start到end下标之间的数组元素，前包含后不包含,他返回一个新数组，原数组不变

arr.splice(start,length,replace1,replace2…) 截取start开始的length长度的数组，将他替换成replace

注意：这里的start是绝对索引，length指向后执行删除动作多少次，并且是连续执行。并且删除之后后面的元素索引会跟着调整，前面的不变化。举例：

|  |
| --- |
| var a=new Array();  a[2]=2; a[3]=3; a[7]=4; a[8]=5;  console.log(a.splice(3,4)); //[3]  console.log(a.length); //5  console.log(a); //[2: 2, 3: 4, 4: 5] |

当数组并不是从0索引开始的，并且是不连续的时，就证明，splice中start指的是绝对索引，length是start后面的连续的索引，不管这个索引存不存在。

arr.toString() 数组转化成字符串

arr.sort() 按字母表升序排序（即按最左边的第一个字母的顺序）

当sort（）有参数时：

|  |
| --- |
| var a=new Array(7,8,9,10,11); a.sort(function(v1,v2){ return v1-v2; });  sort内部使用快速排序，每次比较两个元素大小的时候如果没有参数，则直接判断字母表，如果有参数，则把正在比较的两个参数传入自定义方法并调用（正在比较的两个数会传给自定义方法的v1、v2），如果返回值大于0表示v1>v2，如果等于0，表示v1=v2，如果小于0，表示v1<v2，其实我们传入的方法就是告诉sort怎么比较两个元素谁大谁小，至于排序移动元素过程人家写好了。 |

str.split(‘,’) 在字符串的每个，处分割，成为数组

1. 注意：js中数组的length是根据数组中最大的index+1决定的，并不是真实的数组个数
2. js数组也是对象，数组会把索引值转换为对应的字符串作为对象的属性名。
3. 删除数组元素

delete arr[1] 这种删除并不能改变数组原先的索引，并且length也不会改变。因此要想元素的索引跟随被删元素一起变动，可以使用pop shift 等操作

7）新增数组元素

var box =['a','b','c','d'];

box[box.length] = 'e';

8）转换方法

数组有toLocaleString() toString()和valueOf()方法，且都是内置的。

toString()和valueOf()两个无论重写了谁，返回的值都是一样的！

他们三个会将每个值进行字符串的拼接，以逗号隔开。

### 布尔对象

|  |
| --- |
| var myBoolean=new Boolean(); |

如果布尔对象无初始值或者其值为:

0

-0

null

""

false

undefined

NaN

那么对象的值为 false。否则，其值为 true（即使当自变量为字符串 "false" 时）！

var myBoolean = new Boolean(‘false’);

### Math算数对象

1. Math.round（2.5） 四舍五入
2. Math.random(); 返回 0 到 1 之间的随机数。
3. Math.max(1,3,5,7) 返回几个数中最大数
4. Math.min(2,4,6,8) 返回几个数中的最小数

### 全局方法

encodeURI(x);只编码中文

encodeURIComponent(x) 编码比较彻底

decodeURI(x)

decodeURIComponent(x) 解码

eval(js代码)；运行里面的js代码字符串

# Window

window对象，它表示浏览器窗口；

所有 JavaScript 全局对象、函数以及变量均自动成为 window 对象的成员。

全局变量是 window 对象的属性。

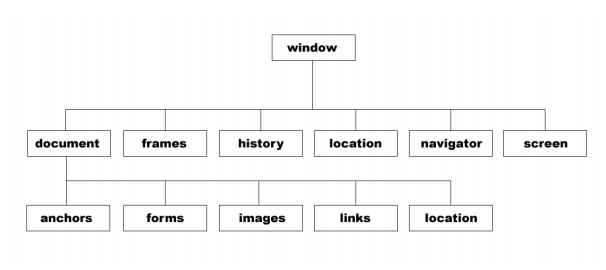
全局函数是 window 对象的方法。

甚至 HTML DOM 的 document 也是 window 对象的属性之一：

|  |
| --- |
| window.document.getElementById("header"); |

获取浏览器窗口大小：（不同浏览器不同的方法）

|  |
| --- |
| var w=window.innerWidth // Internet Explorer、Chrome、Firefox、Opera 以及 Safari：  || document.documentElement.clientWidth // Internet Explorer 8、7、6、5：  || document.body.clientWidth; // Internet Explorer 8、7、6、5：  var h=window.innerHeight  || document.documentElement.clientHeight  || document.body.clientHeight; |



window对象是最顶层的对象，它有六大属性，六大属性本身也是对象；

六大属性之一的document属性，也是对象，且它下面有五大属性（也是对象）；

### 常用方法

window.open(url，name，features) - 打开新窗口

url :新窗口url

name:新窗口名称，凡是以这个名称打开的窗口，都在这个窗口里加载页面，不会弹出新页面；\_blank:跟不起名字一样，新窗口打开，\_parent:在本页打开；

features：新串口的特征

|  |
| --- |
| var myWindow = window.open(url，‘newName’,'width=200,height=100');//产生一个新的window对象 |
| myWindow.focus();  myWindow.close();  myWindow.closed 返回true，新窗口已关闭  myWindow.name 返回新窗口的名字,此处返回newName  myWindow.opener 返回新窗口的父窗口 |

window.close() - 关闭当前窗口

window.moveTo() - 移动当前窗口

window.resizeTo() - 调整当前窗口的尺寸

window.focus() 把键盘焦点给予一个窗口

window.print() 打印当前窗口

window.scrollBy(xnum,ynum) 将文档向右滚动xnum，向下滚动ynum像素

window. scrollTo(xpos,ypos) 将文档滚动到指定坐标

### Window Screen

window.screen 对象包含有关用户屏幕的信息。编写时可以不使用 window 这个前缀。

screen.availWidth - 可用的屏幕宽度

screen.availHeight - 可用的屏幕高度

### window.location

window.location 对象用于获得当前页面的地址 (URL)，并把浏览器重定向到新的页面。

使用时也可以不加window前缀。

location.hostname 返回 web 主机的域名

location.pathname 返回当前页面的路径和文件名

location.port 返回 web 主机的端口 （80 或 443）

location.protocol 返回所使用的 web 协议（http:// 或 https://）

location.href 返回当前页面的URL

以下三个等效：

1)给location.href赋值，即location.href = 'http://www.baidu.com' 则是重定向到新url；

2)location.assign('http://www.baidu.com') //加载新文档

3)location.replace(newUrl)

location.reload() 重新加载当前文档

### Window History

window.history 对象包含浏览器的历史。编写时可以不使用 window 这个前缀。

history.back() - 与在浏览器点击后退按钮相同

history.forward() - 与在浏览器中点击按钮向前相同

### 弹窗

1. alert（）
2. confirm（提示信息） 当点击确认，返回true，点击取消，返回false
3. prompt（"sometext","defaultvalue"）

一般用于询问，提示框出现后，你需要输入一个值，然后点击确定或取消，当点击确定后，prompt返回你的输入值，点取消时返回null

### 计时事件

以下是Window对象的两个方法

setInterval（"javascript function",milliseconds）

间隔指定的毫秒数，不停地执行function的代码；

注意：这里的函数，必须加引号！

直接写函数名，不用加（）；

|  |
| --- |
| setInterval(‘fun’,2000); |

**推荐：**

**扩展性、封住性好**

|  |
| --- |
| setInterval(function () {  document.write("<p>new line</p>");  },2000); |

clearInterval（interval）停止setInterval()执行

|  |
| --- |
| myVar=setInterval("javascript function",milliseconds);  clearInterval(myVar); |

setTimeout（"javascript 函数",毫秒数）

一定时间后执行函数；

clearTimeout（）若上面的延时还未运行，使用此方法停止

|  |
| --- |
| myVar=setTimeout("javascript function",milliseconds);  clearTimeout(myVar); |

### js cookie

js cookie 以名/值对形式存储 ，如下所示：

|  |
| --- |
| username=John Doe |

js创建cookie：

|  |
| --- |
| document.cookie="username=John Doe; expires=Thu, 18 Dec 2013 12:00:00 GMT"; |

这里的过期时间必须是（根据UTC）字符串

d.toUTCString() 需要这样转换一下

js读取cookie：

|  |
| --- |
| var x = document.cookie; |

document.cookie 将以字符串的方式返回所有的 cookies，类型格式：

cookie1=value; cookie2=value; cookie3=value;

js 删除cookie：

|  |
| --- |
| document.cookie = "username=; expires=Thu, 01 Jan 1970 00:00:00 GMT"; |

过期时间设置为之前的时间；删除时不用指定cookie的值

创建、读取、检查、删除cookie实例：

|  |
| --- |
| function setCookie(cname,cvalue,days) {  var d = new Date();  d.setTime(d.getTime()+days\*24\*3600\*1000);  document.cookie = cname +'=' +cvalue +';expires = '+ d.toUTCString();  }  function getCookie(cname) {  var cookies = document.cookie.split(';');  for (var i=0;i<cookies.length;i++) {  if (cookies[i].indexOf(cname+'=')>-1) {  return cookies[i].substring(cname.length+1,cookies[i].length);  }  }  return '';  }  function checkCookie(cname) {  var res = getCookie(cname);  if (res != '') {  return true;  }else {  return false;  }  }  function deleteCookie(cname) {  var res = checkCookie(cname);  if (res) {  setCookie(cname,'',-1);  }else {  alert('此cookie不存在');  }  } |

# HTML 对象

### <input> text

属性：

defaultValue：设置或返回文本域的默认值

# js正则

### 创建

var x = new RegExp(‘x’,’ig’);

等同于 var x = /x/ig; 不要加引号

修饰符：i忽略大小写 g全局匹配 m多行匹配

### 测试正则

pattern.test(string) 匹配成功返回true 失败返回false

pattern.exec(string) 匹配成功返回匹配结果的数组，失败返回null

### 匹配

String对象的四个方法：

|  |  |
| --- | --- |
| 方 法 | 含 义 |
| match(pattern) | 返回 pattern中的子串或 null |
| replace(pattern,replacement) | 用 replacement替换pattern |
| search(pattern) | 返回字符串中 pattern开始位置，无匹配返-1 |
| split(pattern) | 返回字符串按指定 pattern拆分的数组 |

|  |
| --- |
| \*使用前瞻捕获\*/  varpattern=/(goo(?=gle))/; //goo后面必须跟着gle才能捕获  varstr='google';  alert(pattern.exec(str)); 返回的是goo，而不是google |

# 面向对象与原型

### 创建对象

工厂模式：

|  |
| --- |
| function createObject(name,age) {  var obj = new Object();  obj.name=name;  obj.age = age;  obj.run = function(){  return this.name+' running';  };  return obj;  }  var x = createObject('Jack',23);  var y = createObject('Rose',22);  alert(x.run());  alert(y.run()); |

构造函数创建：

|  |
| --- |
| function Box(name,age) {  this.name = name;  this.age = age;  this.run = function () {  return this.name +this.age+'running';  };  }  var x =new Box('Jack',23);  var y =new Box('Rose',22); |

注意：1.构造函数没有new object，但是后台会自动var obj =new Object;

2.this就相当于obj

3.构造函数不许要返回对象，后台自动返回。

4.构造函数一般函数名大写,跟普通函数以示区分

5.创建对象时，必须使用new ：new Box();

对象冒充：

|  |
| --- |
| function Box(name,age) {  this.name = name;  this.age = age;  this.run = function () {  return this.name +this.age+'running';  };  }  var z = new Object();  Box.call(z,'Jack',23); //改变box的作用域  alert(z.run()); |

### 原型

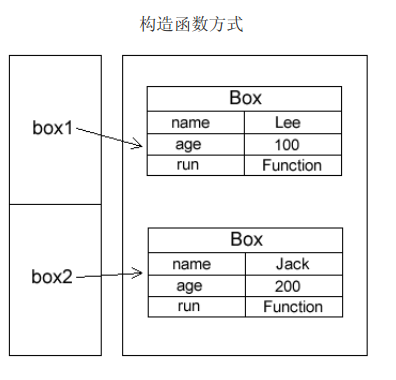
相当于一个父类，任何实例化的对象都会包含原型里的属性和方法。

每个函数都有一个prototype属性，这个属性是一个对象，包含可以由特定类型的所有实例共享的属性和方法。

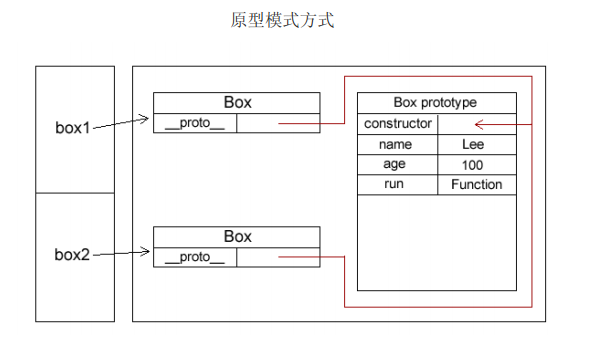
|  |
| --- |
| function Box() {} //构造函数里没有任何东西，如果有，叫做实例属性/实例方法  Box.prototype.name = 'Jack'; //原型属性  Box.prototype.age =22;  Box.prototype.run = function () { //原型方法  return this.name+this.age+'running';  };  var x = new Box(); //一样用new实例化  alert(x.run()); |

注意：若是实例方法，不同实例化，他们的方法地址是不同的，是唯一的。

若是原型方法，不同的实例化，他们的原型方法地址是共享的，都是一样的。



box1 和box2 是两个实例化对象，占用不同的内存，内存地址也不一样。



\_\_proto\_\_属性是实例指向原型对象的一个指针，指向构造函数的原型属性constructor，通过这两个属性就可以访问到原型里的属性和方法了。

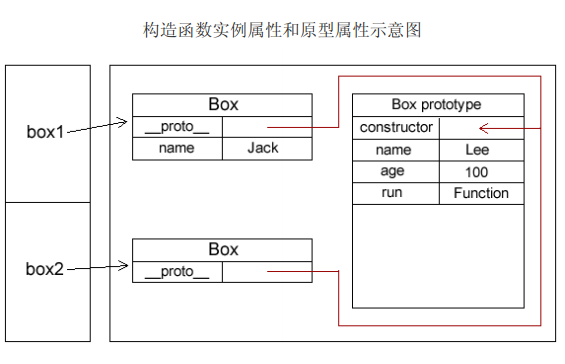
box1.\_\_proto\_\_.constructor即指向function Box(){}

原型模式的执行流程：

1.先查找构造函数实例里的属性和方法，如果有就返回。

2.如果构造函数实例里没有，则去他的原型对象里找，如果有就返回。

|  |
| --- |
| function Box() {  this.name= 'Rose'; //实例里的属性和方法  } //构造函数里没有任何东西，如果有，叫做实例属性/实例方法  Box.prototype.name = 'Jack'; //原型属性  Box.prototype.age =22;  Box.prototype.run = function () { //原型方法  return this.name+this.age+'running';  };  var x = new Box();  alert(x.name); //输出Rose |



box.hasOwnProperty(‘name’); //判断实例中是否有name属性。

‘name’ in box //判断name是否属于box对象的属性，无论他是实例属性还是原型属性；存在返回true；否则返回false

**字面量方式创建原型对象：**

|  |
| --- |
| function Box() {}  Box.prototype = {  'name':'Jack',  'age':23,  'run':function () {  return 'running';  }  };  var x = new Box();  alert(x.run()); |

问题是，这样创建的对象的constructor属性不会指向Box，而指向了Object，原因：

function Box(){} 时，每创建一个函数就会同时创建他的prototype，这个对象也会自动获取constructor属性。

字面量创建时，Box.prototype={} 相当于new Object，创建了一个新对象，新对象的constructor属性重写了Box原来的constructor，因此会指向新对象，这个新prototype没有指定构造函数，因此默认指向Object。

解决：

|  |
| --- |
| function Box() {}  Box.prototype = {  constructor: 'Box', //手动指明  'name':'Jack',  'age':23,  'run':function () {  return 'running';  }  };  var x = new Box(); |

原型的声明是有先后顺序的，所以，重写的原型会切断之前的原型。

|  |
| --- |
| functionBox(){}; //首先，没创建一个函数会产生一个prototype原型对象。  Box.prototype={ //原型被重写了  constructor:Box,  name:'Lee',  age:100,  run:function(){  returnthis.name+this.age+'运行中...';  }  };  Box.prototype={ //不会保留之前的原型对象，完全覆盖重写；  age=200  };  varbox=newBox(); //在这里声明  alert(box.run()); //box只是最初声明的原型 |

原型的缺点：

共享和无法传参；

|  |
| --- |
| function Box(name,age) {  this.name = name;  this.age = age;  }  Box.prototype = {  constructor:Box,  family :['father','mother'],  run : function () {  return this.name+this.age+'running';  }  };  var x = new Box('Jack',23);  var y = new Box('Rose',22);  x.family.push('brother');  alert(y.family); |

当实例化x之后，对原型参数family进行修改，y实例化后，读取到的原型属性时被x修改过的。

所以采用组合构造函数+原型模式：

|  |
| --- |
| function Box(name,age) {  this.name = name; //将不共享的放到构造函数里  this.age = age;  this.family = ['father','mother']  }  Box.prototype = {  constructor:Box,  run : function () { //将共享的使用原型模式  return this.name+this.age+'running';  }  };  var x = new Box('Jack',23);  var y = new Box('Rose',22);  x.family.push('brother');  alert(y.family); |

这种混合模式，很好的解决了共享和传参的问题；

**动态原型模式**：把构造函数和原型封装起来：

注意：此时不能再使用字面量的方式重写原型，这样会切断实例和新原型之间的联系；

即不能Box.prototype ={…}这样创建原型了。

|  |
| --- |
| function Box(name,age) {  this.name = name;  this.age = age;  this.family = ['father','mother'];  alert('初始化前');  Box.prototype.run =function () {  return this.name+this.age+'running';  };  alert('初始化后');  }  var x = new Box('Jack',23);  var y = new Box('Rose',22); |

这样，当你每实例化一个对象时，都会初始化一次run(),都会分配一个内存地址，浪费资源；

可以这样解决：

|  |
| --- |
| function Box(name,age) {  this.name = name;  this.age = age;  if(typeof this.run!='function'){ //初始化过一次后，run方法已经存在，就不在向下执行  alert('初始化前');  Box.prototype.run =function () {  return this.name+this.age+'running';  };  alert('初始化后');  }  } |

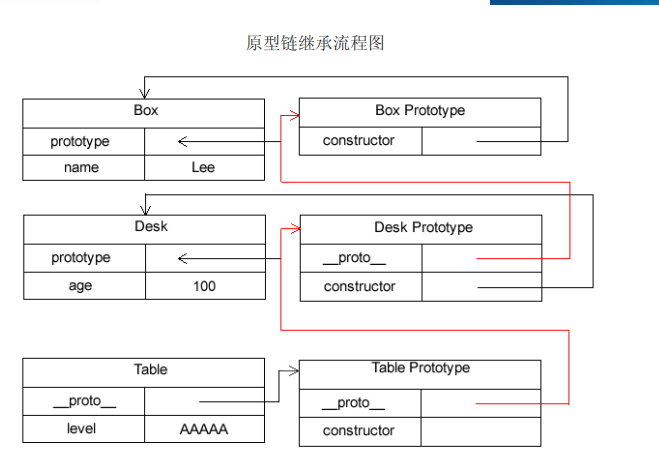
### 继承

js的继承依靠原型链完成；

|  |
| --- |
| function Box() { //超类型（父类、基类）  this.name = 'father';  }  Box.prototype.gender = 23;  function Desk() { //子类型  this.age='23';  }  Desk.prototype = new Box();  var x = new Desk();  alert(x.gender); |

通过原型链继承，将超类型实例化后的对象实例，赋给子类型的原型属性。

new Box() 会将Box构造里的信息和原型里的信息都交给Desk的原型；



**就近原则**：

如果构造函数和原型里都有同一个属性名，则遵循就进原则，先在构造函数里找，没有的话再去原型里找。

**组合继承**： 对象冒充+原型链继承

但由于共享和无法传参的问题，可以用组合继承解决：

|  |
| --- |
| function Box(name) {  this.name = name;  }  Box.prototype.age = 23;  function Desk(name) {  Box.call(this,name);  }  // Desk.prototype = new Box(name); 原型链继承  var x = new Desk('John');  alert(x.age); |

使用对象冒充，能传递参数了，但不能继承超类型的原型，所以还需要跟原型链继承一块使用。

原型式继承：

|  |
| --- |
| function obj(o){ //传递一个原型对象进去  function F(){} //创建一个构造函数  F.prototype = o; //将原型对象赋值给F  return new F(); //将F实例化的对象返回  }  var box = {  name:'Yang',  age:23  };  var box1 = obj(box); //获得一个以box为原型的实例化对象，即实现了继承  alert(box1.age); |

原型继承也存在共享的问题（一个对象修改了原型属性，另一个对象会共享）

寄生式继承：原型模式+工厂模式

|  |
| --- |
| function create(o) {  var f =obj(o);  f.run= function () {  return this.name+' running'; //同样，会共享引用  };  return f;  }  function obj(o){  function F(){}  F.prototype = o;  return new F();  }  var box = {  name:'Yang',  age:23,  family:['a','ab','c']  };  var box1 = create(box);  alert(box1.run()); |

将给实例化的对象增加私有属性或方法的过程进行了封装；

寄生组合继承：

之前的组合继承，因为对象冒充，和原型链继承，会调用两次超类型；

|  |
| --- |
| function Box(name) {  this.name = name;  }  Box.prototype.age = 23;  function Desk(name) {  Box.call(this,name);  }  //本来因为call对象冒充继承不了原型对象的属性和方法，所以要使用原型链继承，但这样会重复调用构造函数的实例属性和方法  // Desk.prototype = new Box(name);  //两个参数是两个构造函数名;前面的是超类型，后面的是子类型  function create(box,desk) {  //仅仅将box的原型对象赋值给了desk，解决了两次调用的问题。  //但要把construcor指针给纠正过来  var f =obj(box.prototype);  f.constructor = desk;  desk.prototype = f;  }  function obj(o){  function F(){}  F.prototype = o;  return new F();  }  create(Box,Desk);//Box是超类型，Desk是子类型  var x = new Desk('June');  alert(x.constructor); |