1. Javascript的组成

Javascript由ECMAScript、DOM、BOM组成

* ECMAScrip

javascript的核心，又称为解释器。主要起翻译作用。因为计算机只认识0和1，ECMAScrip会把对js的所有操作翻译成1和0 给计算机。

ECMAScript可以为不同种类的宿主环境提供核心的脚本编程能力。浏览器是ECMAScript的一个宿主环境，但它并不是唯一的宿主环境。事实上，还有不计其数的其他各种环境（例如 Nombas 的 ScriptEase，以及 Macromedia 同时用在 Flash 和 Director MX 中的 ActionScript）可以运行ECMAScript

ECMAScript 定义的内容有：语法、类型、语句、关键字、保留字、运算符、对象

* DOM

文档对象模型即document，与网页内容进行交互的方法和接口。主要是对html进行操作，比如获取元素、删除元素等

* BOM

浏览器对象模型即window，与浏览器进行交互的方法和接口。主要是对浏览器的操作，比如弹框，打开新窗口、操作cookie等

兼容性

1. ECMAScript几乎不存在兼容性
2. DOM有一些操作不兼容
3. BOM完全不兼容
4. 操作属性的方法

obj.style.width

点代表“的”的意思，凡是用到点的地方都可以用方括号加引号的形式来代替

document.getElementById(‘div’) 相当于document[‘getElementById’]

oBtn.onclick相当于oBtn[‘onclick’]

字符串与变量的区分：

字符串又叫字面量（常量）：它有实际的值，能看到具体的值，需要加引号。如：12，’abc’

变量：它的值不固定会变，不需要加引号。如：var a=?

* 属性与变量

alert(a) //报错 此时a为一个变量，没有定义它，从根上就找不着，浏览器会认为你写错了

alert(window.a) //undefined 此时a照样没有定义，但是a以window.a的形式出现，表示a为window的属性，浏览器会认为a是属性，只不过没有给值

var a;

alert(a); //undefined 上面有var定义a,但是没有给值就为undefined

alert(window.a); //undefined 所有var定义的变量与函数都是window的属性与方法

var a=12;

alert(a); //12 上面有var定义a

alert(window.a); //12 所有var定义的变量与函数都是window的属性与方法

a=12;

alert(a) //12 虽然没有var但是，直接写就表示了a为window的属性

alert(window.a) //12

用没有定义的**变量**会报错，用没有定义的**属性**会undefined

style操作的永远是行间样式，不管读还是写

样式优先级\*<标签<class<id<行间<js js操作属性的优先级最高

让一个div变色，先用style给它变绿，再用className加一个class变红，后者是不会起作用的，因为优先级的原因

所有相对路径都不要判断，除了这个路径的绝对路径里不包含中文，非IE浏览器它会把中文用encodeURI转成带%的形式，IE不会转。包含href、src、颜色值、innerHTML的值都不要做判断

**href & src**

var oImg=document.getElementById("img1");

alert(oImg.href);

//http://127.0.0.1:8020/%E7%BB%83%E4%B9%A0/images/1.jpg 非IE下

//http://127.0.0.1:8020/练习/images/1.jpg IE下

alert(encodeURI('练')); //%E7%BB%83

alert(encodeURI('习')); //%E4%B9%A0

alert(oImg.href=='images/1.jpg') //false oImg.href取的是绝对路径以http://开头的

alert(oImg.href=='http://127.0.0.1:8020/练习/images/1.jpg') //IE下为true，其它全为false

alert(oImg.href=='http://www.a.com/'); //true(如果href里的值为http://www.a.com/)

<a href="images/1.jpg" id="img1"></a>

**颜色值**

var oDiv=document.getElementById("div1");

var a=oDiv.style.color;

//IE下获取的颜色值是你设的值，如果设置的是rgb(),弹出的值是不带空格的，不管设置的时候有没有空格

//非IE下获取的颜色值永远都是rgb(),而且每个值都带空格隔开,不管设置的时候有没有空格

alert(a) //非IE下为rgb(0, 0, 255) IE下为blue 颜色值background:blue;

alert(a) //非IE下为rgb(0, 0, 255) IE下为#0000FF 颜色值background:#0000FF;

alert(a) //非IE下为rgb(0, 0, 255) IE下为rgb(0,0,255) 颜色值background:rgb(0,0,255);

<div id="div1" style="color:rgb(0, 0, 255)"></div>

float写法会有兼容性IE：oDiv.style.styleFloat=’left’;非IE：oDiv.style.cssFloat=’left’;

**获取样式**

IE：currentStyle FF/Chrom：getComputedStyle

复合样式：background-colr、border-left、font-size...

单一样式：width、height、color...

取复合样式currentStyle.backgroundColor

function getStyle(obj,attr){

return obj.currentStyle?obj.currentStyle[attr]:getComputedStyle(obj,false)[attr];

}

var oDiv=document.getElementById("div1");

var a=getStyle(oDiv,'backgroundColor'); //rgb(0, 0, 255)

<div id="div1" style="background-color:rgb(0, 0, 255)"></div>

1. 变量 数据类型 值

判断类型：typof & instanceof

变量本身没有类型，由它的值决定。在 ECMAScript 中值可以被划分为两大类：原始值(原始类型数据)和引用值(对象类型数据)

* 原始值(原始类型数据又称基本类型数据)
* 概念

存储在栈（stack）中，直接存储在变量访问的位置，原因是原始值占据的空间是固定的不会变，所以将他们存储在较小的内存区域（栈）中。这样存储能够迅速查到

不能添加属性与方法，但是有包装对象的可以使用其对应包装对象的属性与方法。它没有标识符，**用字面量**创建的值做比较的话仅比较值，只要值相同就相等

* 类型

数字(还有个NaN)、字符串、布尔值、null、undefined

var a=12; //typeof a number

var b='kaivon'; //typeof b string(唯一没有固定大小的原始类型)

var c=true; //typeof c boolean

var d=null; //typeof d object(js本身的bug，null是原始值，并不是引用值)

var e; //typeof e undefined(未声明的变量typepf后也是undefined,但它与undefined不同，直接用的话，它会报错，undefined不会)

不能添加属性与方法：

var str="kaivon";

str.age=20;

str.show=function(){

alert('abc');

}

alert(str.age) //undefined

str.show(); //报错 str.show is not a function

做比较：

var a='kaivon';

var b='kaivon';

alert(a===b) //true 不仅值相同，类型也相同

前三种原始类型数据都有自己的包装对象，对应为Number、String、Boolean

**包装对象：**让原始值也拥有引用值的属性与方法，但是不能给自己添加属性与方法。只有引用值能给自己添加属性与方法，原始值只是继承了其对应包装对象的原型(prototype)上的属性和方法。但是它不是其包装对象的一个实例，只有用构造函数方法创建的时候它才是

判断原始值的类型，以及区分引用值和原始值时用**typeof**，原始值返回对应的包装对象类型，引用值返回object(除了函数返回function,null返回object)。还可以检查一个变量是否已经声明（下面详细说明）

检测一个值是否是对应构造函数的实例用**instanceof**，返回true或false

var str = 'kaivon';

alert(typeof str); //string类型

alert(str.charAt(0)); //k 继承其包装对象String的charAt方法

alert(str.length); //6 继承其包装对象String的length属性

alert(str.charAt===String.prototype.charAt) //true 原始值的类型是其对应包装对象的方法

alert(str instanceof String); //false str不是String的实例，原始值不是任何对象的实例

alert(new String('kaivon') instanceof String); //true 用new创建的就是String的实例

var a=5.24987290909;

a=a.toFixed(3) //从小数点第4位开始向前四舍五入

alert(a) //5.250

有包装对象的原始值可以和它的包装对象相互转换

转换为包装类型：new String("abc")

转换为原始类型：new String("abc").valueOf()

* 创建方法

1. var方式即字面量创建

var a=12;

var b='kaivon';

var c=true;

1. new方式即构造函数创建（不推荐用，创建后都是object类型，没法做比较）

new方式的参数会遵循数据转换的规则，创建后typeof都是object

**创建数字**

//参数一定要转化后为数字，如果不是，返回NaN

var a=new Number(); //0

var a=new Number(Math.PI); //3.141592653589793

var a=new Number('12'); //12

var a=new Number('kaivon'); //NaN

var a=new Number(true); //1

var a=new Number(false); //0

**创建字符串**

//参数一定要转化后为字符串，如果不是，报错

var b=new String(); //空

var b=new String('kaivon'); //'kaivon'

var b=new String(1); //'0'

var b=new String(true); //'true'

**创建布尔值**

//参数一定要转化后为布尔值，如果不是，报错

var c=new Boolean(); //false

var c=new Boolean(1); //true

var c=new Boolean(0); //false

var c=new Boolean(null); //false

var c=new Boolean(undefined);//false

var c=new Boolean(a); //报错

* 引用值(对象类型数据)
* 概念

存储在堆（heap）中，存储在变量处的是一个指针（point），指向对应的内存处。原因是引用值的大小是不固定的会改变，存储在栈中，会降低查找速度。把值放在内存中，把引用内存的地址（存到栈里）赋给变量，地址的大小是固定的，就能提高查找速度

所有对象都可以添加属性与方法。每个对象都有一个自己唯一的标识符(引用地址)，因此通过字面量或构造函数创建的对象和任何其他对象都不相等。做比较的话同时比较值和标识符(引用地址)，只有值与标识符(引用地址)相同了才相等

* 类型

函数、对象、数组、正则

var a=function (){}; //typeof a function(除了函数外，其它都返回object)

var b={}; //typeof b object

var c=[]; //typeof c object

var d=/\d/; //typeof d object

可以添加属性和方法：

var a=function(){};

a.age='kaivon';

a.show=function(){

alert('abc');

};

alert(a.age); //kaivon 非IE浏览器给函数加不了name属性，name指的是函数名

a.show(); //abc

做比较：

var a=[1,2]; //var创建

var b=[1,2];

alert(a==b) //false 值相同，但标识符(引用地址)不同

var a=new Object(); //构造函数创建

var b=new Object();

alert(a==b) //false 值相同，但标识符(引用地址)不同

var a=[1,2];

var b=a;

alert(a===b) //true 值相等并且标识符(引用地址)也相同

* 创建方法

1. var方式即字面量创建

var a=function (){alert('abc')};

var b={name:'kaivon', age:'20'};

var c=['green','blue','red'];

var d=/\d/;

1. new方式即构造函数创建

**创建函数**

**创建对象**

**创建数组**

**创建正则**

* **Typeof用法**

Typeof会把值转换成字符串，所以做比较的话要与字符串比较

1. 检查一个变量是否存在，是否有值

typeof在两种情况下会返回undefined

1. 变量没有被声明

alert(typeof a); //undefined 变量没有声明 相当于window.a

1. 变量的值是undefined

var a;

alert(typeof a); //undefined 变量声明了但没有赋值

1. 判断一个值等于undefined

如果一个值在未声明的情况下去用，浏览器会报错。如果先用typeof进行判断的话就不会出报错。只有 typeof 可以正常检测未声明的变量的同时还不报错。另一种方法用window.属性

alert(a) //报错 a未声明所以a is not defined

if(typeof a==='undefined'){//a未声明，它的类型与值就是undefined

alert(1); //不报错，弹出1

}

1. 判断一个值不等于undefined也不等于null
2. 区分对象值和原始值

js方法从HTML里拿到属性或值都是字符串类型

var oDiv=document.getElementById("div1");

alert(typeof oDiv.id) //string

alert(typeof oDiv.innerHTML) //string

alert(typeof oDiv.style.width) //string

alert(typeof input.value) //string

<div id="div1" style="width: 100px;">kaivon</div>

1. 类型转换

可以分为显式类型转换(主动去转换类型)与隐式类型转换(浏览器帮助转换类型)

1. 显式类型转换

所谓的显式转换，即运用基本类型函数的构造函数对某个变量进行人为的强制类型转换，包括parseInt(),parseFloat(),Number(),String(),Boolean(),toString()

* 转换成字符串

toString()最没用也最有用的方法。最没用就是加与不加是一样的，它可以被覆盖，防止看源代码。所有数据类型都有这个方法(除了null与undefined)

系统对象下面都是自带的，自己写的对象都是通过原型链找Object下面的

function show(){

alert(1);

}

show.toString=function(){

return '就不让你看';

};

alert(show); //就不让你看

可以改变进制，参数为进制2-32

var str=255;

alert(str.toString(2));

* String类型

字符串转字符串：返回对应字符串，没进制

var a='kaivon';

var b='10';

alert(a.toString()); //kaivon

alert(b.toString(8)); //10

* Boolean类型

布尔值转字符串：返回true或false，没进制

var a=true;

var b=false;

alert(a.toString()); //true

alert(b.toString()); //false

* Number类型

数字转字符串：返回对应数字，有进制，默认为10进制

var a = 10;

var b = 10.0;

alert(a.toString()); //10 默认用10进制转换

alert(b.toString()); //10 默认用10进制转换

可以指定进制

var a = 10;

alert(a.toString(2)); //1010 用2进制转换

alert(a.toString(8)); //12 用8进制转换

alert(a.toString(16)); //a 用16进制转换

* Array类型

数组转字符串：返回数组调用join方法后的值，数组中的元素之间用逗号分隔

var arr=[10,12,['kaivon',8],'red'];

var a=arr.toString(); //'10,12,kaivon,8,red'

var b='10,12,kaivon,8,red';

alert(a===b); //true 值相同，类型也相同

alert(arr.toString()===arr.join()) //true 两个方法相同

* Date类型

日期转字符串：返回对应日期

var d = new Date();

alert(d.toString()); //Fri Jan 08 2016 18:18:06 GMT+0800

* String()

把任何类型强制转换成字符串，返回对应的字符串，相当于调用了toString()方法的结果。 与toString() 方法的唯一不同之处在于，对null和undefined值强制类型转换可以生成字符串而不引发错误

null.toString(); //报错

undefined.toString(); //报错

String(null); //null

String(undefined); //undefined

String(new Object()); //[object Object]

String(function (){alert(1)}); //function (){alert(1)}

String(true); //true

String(12); //12

* 转换成数字
* String类型

字符串转数字：字符串里包含数字的情况下

parseInt(字符串,进制)转换成整数型数字，全局函数不属于任何对象

参数1：值(字符串、数字、对象(如果对象就自动算出对象的结果))，开始结束可以有空格

参数2：进制，有参数：根据参数转换；无参数：根据参数1自动判断，0x开头当作十六进制，0开头当作八进制。1-9开头当作十进制

说明：

1. 从左到右扫描值，如果值的第一位不是数字，返回NaN(第一位必需是数字)
2. 扫描完值没发现有数字则，即转换不了那就返回NaN
3. 扫描到有数字后，即使后面还有数字也不会再走
4. 参数1以0开头，参数2没有IE5678及其它低版本浏览器转换为8进制，其它浏览器转换为10进制（转以0开头的要把进制写上）

parseInt(); //NaN

parseInt(''); //NaN

parseInt('abc12'); //NaN

parseInt('12abc'); //12

parseInt('12abc3'); //12

parseInt("11",2); //3

parseInt("17",8); //15

parseInt("1f",16); //31

parseInt("010"); //没有进制则IE5678及其它低版本为8，其它为10

parseInt("0xA"); //10 0x开头的以16进制转换

parseInt([1,23]); //1

parseFloat()转换成浮点型数字，全局函数不属于任何对象

用法同上，不过只能用十进制。只找到第一个小数点就返回，没找到小数点，有数字就返回数字

parseFloat("102"); //102

parseFloat("10.00"); //10

parseFloat("11.22.33"); //11.22

parseFloat("40 years"); //40

parseFloat("34 45 66"); //34

* Number()

把任何类型强制转换成数字，与parseInt()和parseFloat()的处理方式相似，只是它转换的是整个值，而不是部分值。如果字符串值能被完整地转换，Number()将判断是调用parseInt()还是parseFloat()方法并返回转换后的数字结果，如果不能转换，返回NaN

Number(''); //0

Number(); //0

Number(false) //0

Number([]); //0

Number(null) //0

Number([1]); //1

Number(['1']); //1

Number(true) //1

Number("1.2") //1.2

Number("12") //12

Number(50) //50

Number(undefined) //NaN

Number(['kaivon']); //NaN

Number(['0']); //NaN

Number({}); //NaN

Number('12a'); //NaN

Number(function a(){});//NaN

Number([1,2,3]); //NaN

Number(new object()) //NaN

* 转换成布尔值

Boolean()把任何类型强制转换成布尔值，返回true或false

非空字符串、非零数字或对象、true、Infinity时返回true

Boolean('a'); //true 非空字符串

Boolean(1); //true 非空数字

Boolean(-1); //true 非空数字

Boolean(Infinity); //true 无穷大

Boolean(new Object()); //true 对象

Boolean([]); //true 空数组

Boolean({}); //true 空对象

Boolean(function(){}); //true 空函数(不管函数里是否返回false)

Boolean(function(){return false}); //true 不管函数里返回什么都为true

参数为空、空字符串、数字零、undefined或null、false、NaN返回false

Boolean(); //false 空

Boolean(''); //false 空字符串

Boolean(0); //false 数字0

Boolean(null); //false null

Boolean(undefined); //false undefined

Boolean(false); //false false

Boolean(NaN); //false NaN

1. 隐式类型转换

是指在进行**运算**时JavaScript进行自动的类型转换。

* 加号+
* 对象转字符串

先调用对象的toString()方法，如果成功则返回，如果不成功调用valueOf()方法转成原始值，如果还不成功，则报错，如果成功则再调用toString方法

如果两个数值中有一个为字符串，则都转为字符串，如果没有则都转成数字，如果都是对象，都转为字符串

* 对象转数字

先调用对象的valueOf() 方法，如果成功则返回，如果不成功调用toString()方法转成原始值，如果还不成功，则报错，如果成功再调用Number方法转成数字

[] + [] //'' 首先尝试 valueOf()方法,返回数组本身,再调用 toString()方法，返回一个空字符串.因此，[] + [] 的结果实际上是两个空字符串的连接

[] + {} //[object Object] 空对象转换成字符串为[object Object]，最终的结果其实是把 "" 和 "[object Object]" 两个字符串连接起来

{} + {} //[object Object][object Object]

+[] //0

alert([1,2]+[3,4]) //1,23,4

++[[]][+[]]+[+[]] //10

<http://justjavac.com/javascript/2012/12/20/object-plus-object.html>

* 双等号==

1. 如果存在NaN，一律返回false
2. null只与undefined互等的，与其它任何都不等
3. 有数字的，会把另一个转为数字再对比
4. 有布尔值的，会把布尔值转成数字再对比
5. 有字符串的，对方是对象，把对象toString后比较，有数字，字符串转数字，都是字符串直接比较
6. 有对象的，对方是字符串，就调toString后比较，对方是数字就调valueOf比较

只要能转成数字的都会转成数字去对比

null==undefined; //true

undefined==null; //true

5=='5'; //true 把字符串转成数字再与数字对比 数字与字符串相比，都转成数字

'5'==5; //true 把字符串转成数字再与数字对比

true==5; //false 把布尔值转成数字0(false)或1(true)再对比

true=='5'; //false 布尔值转成数字1,再把字符串转成数字5，再对比

[]=='1'; //false 如果有对象的话，要把对象转成原始值，先valueOf再toString后再对比

除Date外，统统是先valueOf、再toString（Date 在 + 和 == 时优先转化为字串

x+'' //等价于 String(x)

+x //等价于 Number(x)

!!x //等价于 Boolean(x)

1、+ //200 + '3' 变成字符串

2、- \* / % ++ -- //'200' - 3 变成数字

3、> < <= >= //数字的比较 、字符串的比较

4、! 取反 //把右边的数据类型转成布尔值

5、== //先转换成一样的类型，再对比。尽量用三个等号判断，可能会不严谨

6、=== //类型与内容都一样

7、alert() //把所有东西转成字符串

undefined加引号表示字符串，不加引号表示undefined类型

false == 'false' //false

false == undefined //false

false == 'undefined' //false

null == 'undefined' //false

false == null //false null只与自身或undefined相等，其它都不等（包括空字符串）

false == false //true

null == undefined //true undefined只与自身或null相等，其它都不等（包括空字符串）

0 == '' //true

true==1 //true

true==2 //false

undefined==0 //false

null==0 //false

typeof a //undefined alert(a)报错 a is not defined

**原始值转原始值：**数字、字符串、布尔值之间的相互转换

1. 字符串如果不能转为数字，则转化为 NaN （undefined 也会转化为这个）
2. null 可以转化为数字0，undefined不可以转为数字
3. 数字里的 Infinity 和 -Infinity 都可以转化为对应的字符串，转化为布尔值时也是true

**原始值转对象：**

用new构造函数

new String("abc")

**对象转原始值**

new String("abc").valueOf()

对象都继承了两个方法来进行到原始值的转换，一个是 toString ，一个是 valueOf，一个是转化为字符串的，一个是转化为数值。只是 toString 的方法强大一些，基本上什么都能转；valueOf 相比之下弱一些，唯一能完整转换的对象就是把 Date() 对象转化为对应的时间戳

对象转数值的方法，先尝试 valueOf （如果能返回原始值就退出）、如果不行再尝试 toString ，否则NaN

对象转字符串：对象转字符串的过程恰好是反过来的，先尝试 toString （能返回原始值就退出），再 valueOf ，否则进行报错

Array.prototype.valueOf = function() {return 10} // 重写valueOf函数

Array.prototype.toString = function() {return '88'} // 重写toString函数

!![] // 结果为true，任意对象转换为布尔值均为真

[] + 1 // 结果为11，在隐式转换里，先valueOf，有原始值且能参与运算则直接返回

[] + '' // 结果为'10'，隐式转换，虽然是要转换为字符串，但同样是先进行了valueOf

Number([]) // 结果为10，显示转换，先进行valueOf

String([]) // 结果为'88'，显式转换，先进行toString

Array.prototype.valueOf = null // 令valueOf失效

[] + 1 // 结果为'881'，隐式转换valueOf不行后转到toString,得到原始值字符串直接返回

Number([]) // 同理，结果为88

* null与undefined的区别
* null == undefined

null是一个表示“无”的对象，转数值(只在跟数字操作的时候，做比较的时候不会转)为0

undefined是一个表示“无”的原始值（原始类型：undefined,boolean,number,string；引用类型：object），转数值为NaN(undefined参与任何数值计算时，其结果一定是NaN)

var a = undefined;

var b = null;

alert(a==b) //true

alert(a===b) //false 因为类型不同

alert(typeof a) //undefined

alert(typeof b) //object

* undefined为window的属性 alert(undefined in window);//true；null是一个object对象
* undefined只与自身或null或者一个元素的值为undefined的元素相等，和其它的都不等；null只与自身或undefinded或获取的元素为null的元素相等

var a; //定义一个未赋值的变量

var b=document.getElementById("app"); //页面中没有app这个ID

undefined === undefined; //true 与自身相等

undefined == null; //true 与true相等

!undefined === true; //true 取反的时候与true相等，并且类型也相等(!会把类型转换成布尔值)

!!undefined === false; //true 取双反的时候与false相等，并且类型也相等

a === undefined; //true a为声明的一个没有值的变量,类型相等

b == undefined; //true 因为null与undefined是相等的，所以a和b与undefined或null都相等

typeof a === 'undefined'; //true typeof会转换成字符串,只会与字符串相等，并且类型相等

typeof a == undefined; //false

undefined == 'undefined'; //false 与字符串不相等

undefined == 0; //false 与数字不相等

undefined == false; //false 与布尔值不相等

undefined == ''; //false 与空值不相等

null === null; //true 与自身相等

null == undefined; //true 与undefined相等

!null === true; //true 取反的时候与true相等，并且类型也相等

!!null === false; //true 取双反的时候与false相等，并且类型也相等

b === null; //true 与空对象相等,类型也相同

a == null; //true 因为null与undefined是相等的，所以a和b与undefined或null都相等

typeof b==='object'; //true 类型相同，都为字符串

typeof b == null; //false b的类型为object，并且被typeof转为字符串了，null不为字符串

null == '0'; //false 与字符串不相等

null == 0; //false 与数字不相等

null == false; //false 与布尔值不相等

null == ''; //false 与空值不相等

//判断一个变量是否存在不能判断它的值，要用typeof判断他的类型是否为字符串undefined’

* undefined和null在if语句中，都会被自动转为false

if (!undefined){

alert(1); //弹出1 undefined is false

}

if (!null){

alert(1); //弹出1 null is false

}

* null可以被转化成为数字0，而undefined不可以

var a=Number(null);

alert(a) //0

var b=5 + null;

alert(b) //5

var a=Number(undefined);

alert(a) //NaN

var b=5 + undefined;

alert(b) //NaN

* 目前的用法：

null表示"没有对象"，即该处不应该有值。典型用法是：

1. 作为函数的参数，表示该函数的参数不是对象
2. 作为对象原型链的终点
3. 空对象或者不存在的对象

alert(Object.getPrototypeOf(Object.prototype)) //null 对象原型链的终点

var a=document.getElementById("app");

alert(a) //null a为空对象或不存在的对象

undefined表示"缺少值"，就是此处应该有一个值，但是还没有定义。典型用法是：

1. 变量被声明了，但没有赋值时，就等于undefined
2. 调用函数时，应该提供的参数没有提供，该参数等于undefined
3. 对象没有赋值的属性，该属性的值为undefined
4. 函数没有返回值时，默认返回undefined

var i;

alert(i) //undefined 变量声明了，没有赋值

function f(x){alert(x)}

f() //undefined 调用函数没有传参

var o = new Object();

alert(o.p) //undefined 对象没有赋值的属性

var x = f();

alert(x) //undefined 函数没有返回值

* null为javascript的关键字，不会被修改，undefined不是保留字也不是关键字，是会被修改的

var undefined=10;

alert(undefined); //在IE678下会被修改，弹出10。其它浏览器为undefined

与undefined做比较的话，其实是找window下属性为undefined的值，window下的属性非常多，所以会对性能有影响。这就是许多前端JS框架为什么常常要自己定义一个局部undefined变量的原因

* 真&假

真：true、非零数字、非空字符串、非空对象

假：false、数字零、空字符串、空对象、undefined

* NaN

not a number是数字类型但不是数字，发生在字符串、布尔值转换失败时

alert( typeof (NaN) ); // number

一旦写程序中出现NaN，肯定进行了非法的运算操作

NaN 是 false alert(NaN)=false;

NaN===NaN fasel，与自己都不相等！！

isNaN()

判断值是不是数字，是数字返回false，不是数字返回true

原理是，给它一个数字，它先丢给Number，让Number去判断，如果Number判断出是一个数字，它就返回false，如果Number判断出不是个数字即NaN，它就返回true

isNaN(2) fase

isNaN(‘字符串’) true

isNaN(function(){‘alert(1)’}) true

isNaN(‘200’) false

isNaN(true) false

isNaN(false) false

isNaN([]) false

1. 数组

New Array(1) //它里面如果放一个数字的话就表示数组的长度

数组的length属性即可以读也可以设置，设置相当于截断数组，但是不用于dom方法获取的数组

字符串里的length是只能读不能写的，DOM方法获取的是元素集合，是个伪数组，只有length属性，不具备数组的这些方法

设置数组length=0的话就是清空数组

push() 向后添加一个 //它有一个返回值，数组的length

pop() 删除最后一个 //返回删除的那个

shift() 删除前一个 //返回删除的那个

unshift() 向前添加一个 //也有一个返回值，数组的length，但ie67不支持

* splice()

splice(起点，长度) //删除任意元素，返回删除的元素

splice(起点，0，多个元素) //插入元素，返回删除的元素

splice(起点，长度，元素) //替换元素，返回删除的元素

截取后会把值转成数组，要的值在数组的第0位上，详见：随机排序数组

* concant() 连接两个数组
* join() 把数组转成字符串

var arr = [ 'aa', 'bb', 'cc' ];

arr.join('') //aabbcc 把数组里的内容都连起来成为字符串

arr.join(‘-’) //aa-bb-cc

arr.join() //aa,bb,cc 括号里不加参数默认加个逗号

* sort() 排序

先都转成字符串，再按照**第一位**字符Unieconde编码从低到高去排序

LocaleCompare 不兼容chrom

字符串的比较，按照编码来比，看页面的编码

var arr=['啊北京','上海','广州','深圳'];

arr.sort(function(arr1,arr2){

return arr1.localeCompare(arr2);

});

alert(arr);

arr.sort(function(a,b){

return a-b //正序 a和b是随机从数组里揪出两个数字相比较。a-b如果是正数就交换a和b的位置，如果是负数就不交换位置，如果相等位置不变

return b-a //倒序

});

倒序reverse();

作用：把数组颠倒过来

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

arr.reverse();

把字符串反转（面试题）

var str = 'abcdef';

alert(str.split('').reverse().join(''));

* 快速排序

排序的方法：

1. 找到一个基准点
2. 建立一个数组分别存储左边和右边的数组
3. 利用递归进行下次比较

function quickSort(arr){

if(arr.length<=1){

return arr;

}

var num=Math.floor(arr.length/2);

//var numValue=arr[num];

var numValue=arr.splice(num,1);

var left=[];

var right=[];

for(var i=0;i<arr.length;i++){

if(arr[i] <numValue){

left.push(arr[i]);

}else{

right.push(arr[i]);

}

}

return quickSort(left).concat([numValue],quickSort(right));

}

alert(quickSort([12,5,37,6,22,40]));

1. DOM

节点的属性都是只读的，不能修改

* 子节点

ChildNodes & children（只读属性）

childNodes 获取**子**元素(只是第**一层**的子元素)，在IE6-8不会取到**换行、文本节点**，在其它的浏览器（标准）里会取到文本和元素节点，并且包含非法嵌套的标签（IE7以下不包含）

children会只获取元素**子节点**，没有兼容性，放心去用，子节点只算第一层节点，但是在IE7以后的浏览器获取的标签不包含非法嵌套的标签，它会把非法嵌套的标签放到合法标签里面

* 节点类型

nodeType==3 文本节点

nodeType==1 元素节点

* 父节点

parrentNode父节点

offsetParent获取元素**有定位**的父节点

绝对定位的元素是根据外层相对定位的元素来定位的，offsetParent会一直往上查，直到查到有相对定位的元素，如果没有定位，那就是Body。IE7以下如果当前元素的某个父级触发了layout，那么offsetParent就是触发了layout的元素

* 首尾节点

都有兼容问题

fistChild,lastChild IE7以下取到是元素节点，其它标签取到的是文本节点

fistElementChild,lastElementChild 针对标准浏览器，非标准没有这个方法，取到元素节点

var oFirst = oUl.firstElementChild || oUl.firstChild;如果当前元素没有了节点的话就会出问题，IE下会返回个null

* 兄弟节点，与首尾节点一样

前面针对IE7以下，后面针对高版本浏览器

previousSibling previousElementSibling

nextSibling nextElementSibling

* 获取节点

getElementsByTagName与getElementById的区别

1. 前面的对象不同，一个是document一个是别的元素。
2. 一个获取一个元素，一个获取一堆元素。
3. getElementsByTagName可以获取动态创建的元素

var aBtn = document.getElementsByTagName('input');// alert(aBtn.length);弹不出来

document.body.innerHTML = '<input type="button" value="按钮" /><input type="button" value="按钮" /><input type="button" value="按钮" />';

// alert(aBtn.length); 可以弹出来

* 插入节点

父级.appendChild(子节点); 在父级末尾添加一个节点

父级.insertBefore(新元素，在谁之前); 在父级前面添加一个节点

insertBefore的问题：在IE下第二个参数的节点如果不存在就会报错，其它浏览器会以appendChild的方式添加进去。最好在插入之前判断一下第二个参数的节点有没有

这些方法既可以操作动态创建的元素，也可以操作已有的元素

* 删除节点

父级.removeChild(子节点);

* 替换节点

父级.replaceChild(新节点，被替换的节点);

* 获取属性

元素.attributes 获取到元素的集合，属性指的是结构里的Id style等

元素.attributes.name 获取到元素属性的名字

元素.attributes.value 获取到元素属性的值

getAttrubute,setAttribute,removeAttribute与.获取的区别

1. .+属性名[]+属性名的方法在标签浏览器下获取不到自定义属性的值，ie67可以。getAttribute、setAttribute方法在任何浏览器下都可以获取到**自定义属性,样式不可以**
2. 获取图片的src，.+属性名[]+属性名的方法获取到的是绝对路径。而getAttribute方法获取到的是与src里一模一样的值，但是Ie67不行
3. 有兼容??

* 文档碎片

document.createDocumentFragment();

例子：一下创建10000个Li，把们他先appendChild到文档碎片里，在循环外面把文档碎片添加到UL里。在高级浏览器上几乎提高不了多少性能，还有可能降低性能，但是对ie7以下的浏览器性能有所提高

* DOM属性

document.title

document.cookie

document.domain 用来设置文档所在的域

document. lastModified 文档最后修改的时间

document. referrer 引用，来源，需要放在服务器上

document. URL 当前文件的路径跟location一样

1. 表格

.tBodies[0] tbody

.rows[0] tr

.cells[0] td

.tHead head

.tFoot foot

删除一行的时候，一定要在表格下的tBodies里removeChild，而不能直接在表格下removeChild

模仿百度搜索，可以输多个关键词中间空格隔开

|  |
| --- |
| window.onload=function() |
| { |
| var oName=document.getElementById('name'); |
| var oBtn=document.getElementById('btn1'); |
| var oTab=document.getElementById('tab1'); |
|  |
| oBtn.onclick=function() |
| { |
| for(var i=0;i<oTab.tBodies[0].rows.length;i++) |
| { |
| var oTxt=oName.value.toLowerCase().split(' '); |
| var oTabValue=oTab.tBodies[0].rows[i].cells[1].innerHTML.toLowerCase(); |
|  |
| for(var j=0;j<oTxt.length;j++) |
| { |
| if(oTabValue.search(oTxt[j])!=-1) |
| { |
| oTab.tBodies[0].rows[i].style.background='yellow'; |
| } |
| } |
| } |
| }; |
| }; |

Html的部分都是表格，在此省略

1. Js代码规格

1.js的分层(功能) : jquery(tools) 组件(ui) 应用(app), mvc(backboneJs)

2.js的规划(管理) : 避免全局变量和方法(命名空间，闭包，面向对象) , 模块化(seaJs,requireJs)

Document.title=123;document.body.innerHTML=’abc’;

Title可以直接用document.而Body需要加一个InnerHTML

1. 作用域、作用域链、预解析、声明、表达式

作用：读写

域：一个空间、范围、区域。一个scrip、函数都是一个域：script、函数、json{}。If、for不是一个域

作用域：在一个特定的空间、范围内对数据进行读写操作

alert(a) //报错a is not defined，先找var定义的a,没找到，所以报错

alert(a) //报错a is not defined，先找var定义的a,没找到，所以报错

a=1;

alert(a) //undefined ，先找var定义的a,找到了，但自动给他赋了个undefined值

var a=1;

* 预解析

浏览器内部有一个专门解析JS的东西叫“JS解析器”。它的工作流程至少分为两部分，还会有语法分析、词法分析等：

1. 找东西（js预解析）：

* 先找var、function、参数（局部变量与var一样）定义的名字（等号前面的且不包含不带var定义的全局变量即声明）先存起来，但不会读取值。

注意：如果代码里有一个函数，里面也有var，而且这个函数被调用了，函数里的var是不会被这一步预解析的，它会在函数调用的时候再被预解析（作用域链的问题，一旦遇到函数调用，就是另一个作用域的开始，会重新按解析流程走）

* 如果是变量或者参数，会给它设定一个值undefined。（原因是有一个偷懒机制，它怕你定义的值不同，可能是字符串，可能是数组等等）
* 如果是函数，会把函数的整体都抓过来

\*在预解析过程中，如果解析出同名的变量，只会留一个。变量和函数重名，只留函数；函数与函数重名，留后来的函数。预解析遇到重名的变量函数其实是筛选，会只留一个(只存在同一个作用域中，不包含嵌套函数的预解析)

1. 逐行解读代码

从头到尾的读代码，遇到有表达式（+ - \* / % ++ -- 参数等，凡是能改变值的动作都叫表达式，**函数声明如果没有调用就不是表达式，改不了值，没有写var的赋值也是表达式，也能改变值**）的代码后，会把后面的值赋给第一步找到的变量，如果是重名的变量，后面的值会取代前面的值（表达式会修改预解析的值，修改的时候不会遵循上面说的重名规则，只存在不同的作用域中）

* 作用域链

变量会自上而下，自里由外的走

如果有两个script标签，js还会按照1、预解析2、逐行读代码的步骤走完一个script后再走另一个script。Js是单线程，所以只能一个一个来。但是在上一个script里定义的变量，下一个script里是可以访问到的<script type="text/javascript">

var a=1;

</script>

<script type="text/javascript">

alert(a) //1 能找到，上面的代码块里已经解析了一个变量a

</script>

<script type="text/javascript">

alert(a) //报错，下边的代码就不会走

</script>

<script type="text/javascript">

var a=1;

</script>

如果代码里有一个函数，里面也有var，而且这个函数被调用了，函数里的var是不会被第一步预解析的，它会在函数调用的时候再被预解析（作用域链的问题，一旦遇到函数调用，就是另一个作用域的开始，会重新按解析流程走）

如果遇到重名，外面的是全局的，函数内的是局部的

函数内部预解析后，走第2 部的时候，它会优先找函数内部有没有赋值，而不会去找外面的

下边的第一条a不管用var定义或者不用var定义效果是一样的。如果没有var直接写成a=1;在第二步的时候看到了赋值，不管前面有没有定义，都会定义个a并赋值

var a = 1; //第一步找到a和fn1，不找函数里。第二步，逐行运行代码，走到这里看到了表达式，就给a赋值为1

function fn1(){

alert(a); //undefined 在局部函数里找到了(下面有var a)a但是没有赋值

var a = 2; //函数内部预解析第一步找到全局a，第二步找到一个局部a把2赋值给局部a

//alert(a); //2

}

fn1(a); //第二步走到这，函数被调用了，新的作用域，再函数内部重新走两步

alert(a); //1 虽然函数内部也有个a，但是是两个作用域，不会替换

var a = 1; //第一步找到a和fn1,不找函数里。第二步，逐行运行代码,并给a赋值

function fn1(){

alert(a); //1,在局部函数里找不到a(里面没有var a，如果有var 就表示定义了)，就会往外找，找到最外面的a。一个作用域里找不到的变量，就会往外找，一层一层往外找（注意跟上面的区别）

a = 2; //赋值，局部里没有var a就往外找，把外面的a赋值为2(局部变量可以改全局变量，局部有能力改外面的值)

}

fn1();

alert(a); //2，被局部变量给改了

var a = 1;

function fn1(a){ //有参数，先找参数，参数就是局部变量与var是一样的

alert(a); //undefined，虽然有参数了，但是里面调用的时候没有传进来值但有参数即var a=undefined

a = 2; //发现参数了，说明有a有定义了，所以改的是局部a的值

}

fn1();

alert(a); //1

var a = 1;

function fn1(a){ //这里读到一个var a=1;函数内部解析第一步，找到a,第二步，a被赋值为1

alert(a); //函数内部解析第二步走到这里，a=1

a = 2; //函数内部解析第二步走到这里，a被赋值为2，改的是自己内部的a，因为上面内部解析的时候找到了var a，所以改的是局部的

}

fn1(a); //函数调用的时候传进去了一个外部最上面定义的var a=1

alert(a); //1 局部有自己的a了，就不改外面的a

x = 1; //第二步看到有表达式了，就赋值，但是没有找到var 定义的x，所以就直接定义了个x并赋值为1

function test(){ //第一步解析出来只有test函数

alert(this.x); //函数内部解析，第一步，什么也没有。第二步找this.index，函数内部没有，往外找，找到了一个x=1，所以this.x就为1。因为调用的this是window，外面定义的x也是window的属性，所以这里没有考虑this

}

test(); // 1

总结：先找东西后逐行解读代码，执行过程中遇到表达式就会改原来的值，如果遇到函数调用就开了一个新的作用域。如果函数调用有参数，参数只是这个函数里的局部变量，有参数的时候，函数里不用var的表达式改变的也是函数的局部变量。然后在新作用域里再重复上面的事情，如果在执行过程中没找到里面有东西（没有参数，或者函数内部没定义）就会返回到父级作用域里找，如果外面也没有就会报错，这就是作用域链的过程

可以用全局变量或者传参的方式找到局部变量

唯独FF在预解析if、for等大括号里的内容时，是解析不到的。提示：尽量不要在if、for里定义变量函数

**函数a嵌套在函数b里，a能够访问b里的变量。a在别的地方定义，但是在b里调用，a是访问不到b里的变量的**

1. 时间对象

var iNew=new Date( 2013, 10, 27, 21,56,0);

var iNew=new Date( 'November 27,2013 22:3:0');

两个date可以直接减，得出来是毫秒

var t = Math.floor((iNew - iNow)/1000); //向下取整

天：Math.floor(t/86400)

时：Math.floor(t%86400/3600)

分：Math.floor(t%86400%3600/60)

秒：t%60

数字形式：new Date( 2013,4,1,9,48,12 );

字符串形式：new Date('June 10,2013 12:12:12');

getDay()，0表示星期天，1表示星期1，以此类推，外国人以星期日为一个星期的第一天

1. 字符串操作

var str = '妙味课堂';

* charAt()

str.length; //4

str.charAt(); //妙 返回第0个字符串

取字符串里的某一位用charAt，不要用方括号的形式，不兼容

str[0] IE7以下不支持，其它浏览器都支持

charAt(0) 支持所有浏览器（推荐方法）

str.length 所有浏览器都支持

* charCodeAt & fromCharCode() 参数为字符的索引

str.charCodeAt() ; // 22937 返回第0个字符编码

str.charCodeAt(1) ; // 21619

// 0~9的编码48~57 a~z的编码97~122 A~Z的编码65~90

string.fromCharCode(22937, 21619) ); //妙味 返回编码对应的字符与charCodeAt相反

* indexOf() & lastIndexOf()

str.indexOf('味') //1 返回字符串所在的位置，从左到右查找，找不到返回-1

str.indexOf(‘m’, 5) //从字符串的第5位开始找m，不能为负数（默认当成0处理）

str.lastIndexOf() //从后往前找

* 字符串比较

alert( '莫涛' > '杜鹏' ); //true 比较的是第一个字的unieCode编码值，第一位相同比第二位

alert( '10000' > 2 ); //true 都转成数字比较

alert( '10000' > '2' ); //false 都转成字符串比较第一位的编码

* substring() 参数为一个返回截掉后的字符，参数为两个返回截掉的字符

str.substring(); //截取全部，0-str,length，返回截取后的字符串

str.substring(1) //’味课堂’ 截掉第一个，返回截掉后的字符

str.substring(2,0) ; //’妙味’ 返回截掉的字符，从第0位开始截掉2个字符，参数的位置无所谓，它会自动识别，大的往后扔，小的往前扔

str.substring(-3, 2) ; // -3 当成0处理

str.substring(2, -3) ; //交换参数位置==str.substring(-3, 2)

* slice() 参数为一个返回截掉后的字符，参数为两个返回截掉的字符

str.slice(2); //’课堂’ 从第0位开始截2位，返回截掉后的字符

str.slice(2, 0) ; //返回空 参数不交换位置

str.slice(-4, -2) ; //‘妙味’ 负数从后面倒着往前数~

str.slice(2,3) //’课’ 截掉从第2位到3的前一位，返回截掉的字符

str.slice(-2) ; //’课堂’ 不会当成0为处理，把最后一位当-1，截取最后两个字符

substring与slice作用都一样，都是截取字符串

* split() 把字符串分隔成数组

str.split() //[‘妙味课堂’] 没参数把整体做为数组

str.split(‘’) //["妙","味","课","堂"] 空参数，把每个字符做为数组里一个内容

str.split(‘@’) //有参数，以参数进行分隔成数组

* toUpperCase() & toLowerCase()

str.toUpperCase(); // 转成大写（针对字母）

str.toLowerCase(); // 转成小写

1. 函数声明与函数表达式

* 函数声明

function aaa(){}

* 函数表达式（函数名可写可不写），有名字的叫命名，没名字的匿名

var a= function aaa(){}

var b= function (){}

~function fn(){}

-function fn(){}

+function fn(){}

!function fn(){}

区别：

1. 函数表达式可以后面直接加括号执行，函数声明不可以

function aaa(){alert(1)}(); //函数声明(不可以执行 报错)

var a=function aaa(){alert(1)}(); //函数表达式(可以执行)

~function aaa(){alert(1)}(); //函数表达式(可以执行)

1. 函数声明可以被提前解析出来称为函数声明预解析

if(true){

function aaa(){

alert(1);

}

}else{

function aaa(){

alert(2);

}

}

aaa(); //FF下弹1 其它弹2

aaa函数为函数声明，它会预解析，各个浏览器下边执行的结果会不同。FF下弹1，其它弹2。如果换成函数表达式就会没问题了，var a=function aaa(){alert(1)} //1

提示：如果想在不同的语句里执行不同的函数，一定不要用函数声明，要用函数表达式

1. 如果调用命名函数fn()各浏览器正确执行，而调用aaa()，则只有ie能执行，其它报错。内部的名字在外面是不可以被调用的。IE8以下的浏览器的解析是有问题的，它会把下面的函数解析成两个函数，一个fn，一个aaa

var fn = function aaa() {

alert(1);

alert(typeof aaa) //在函数内部是可以调用的，能找到

}

fn()

aaa(); //除IE外会报错，找不到

1. 面试题

(function fn(){alert(1)})

fn();

ie8以下能弹出来，其它报错

原因：此函数加了个括号，就成了函数表达式，而函数表达式的名字在外面是调用不到的

* 函数返回值

函数的返回值与参数相反，参数是往函数里加东西，而返回值在往函数外面返东西。返回的东西可以是一个值，也可以是一个表达式，如果是表达式，函数会先计算然后再返回。参数可以有多个，而返回值只能有一个。不过可以通过相关的办法做到返回出去多个值，比如json。如果return个空，会是Undefined

1. 函数返回值

alert(fn2()); //弹出来整个函数，函数名加括号=return后面的内容

function fn2(){

return fnction(){

alert(1)

};

};

fn2()(); //1 fn2()执行后是一个函数名，函数名加()就是执行的意思

return后面跟的所有代码都不会执行

每个函数都默认有一个return值undefined，如果自己没有加return就会返回undefined

Arguments为函数所有参数的组合，别名不定参或可变参或者形参，可以不写，用arguments[i]可以获取到。调用时候括号里的参数叫实参，必需要写。

Arguments是实参的集合与byTagName性质一样

getCOmputedStyle

getCOmputedStyle得到的样式是浏览器计算过后的样式，会有一些问题

background: url() red …… 复合样式（不要获取）

backgroundColor 单一样式（不要用来做判断，浏览器下会得到不同的值）

不要有空格

要获取未设置后的样式：不兼容

getComputedStyle( $('div1'), 250 ) 后面的参数是遗留问题，FF4.0以前必需要加这个参数

1. 随机数

x ~ y

var x = 3;

var y = 49;

alert( Math.round( Math.random()\*(y-x) + x ) );

0~x

alert( Math.round( Math.random()\*x) );

1~x

alert( Math.ceil( Math.random()\*x) );

随机排序数组

function randomArray(arr){

var newArr1=[];

var newArr2=[];

for(var i=0;i<arr.length;i++){

newArr1[i]=arr[i];

}

for(var i=0;i<arr.length;i++){

var rad=Math.random()\*newArr1.length;

var a=newArr1.splice(rad,1)[0]; //[Object] [{x:12,y:5}] splice会把一个值转成数组，把值放在数组的第一位，如果直接用的话，它是一个对象，不是实际的值，会有问题

newArr2.push(a);

}

return newArr2;

}

var obj1=[{x:0,y:20},{x:35,y:46},{x:12,y:5}];

var obj2=randomArray(obj1);

//console.log(obj1); //[Object, Object, Object] 每个Object里是 x:0 y:20

//console.log(obj2); //[Object, Object, Object] 每个Object里是 x:0 y:20

1. 事件

* 焦点事件

obj.focus() //让obj得到焦点

obj.blur() //让obj失去焦点

obj.select() //选择obj里的文本内容，只支持能够用户输入的内容，div里的内容不行

* Event对象

只在有事件的函数里发生，没事件的函数不会发生

IE/chrome：Event是一个内置全局对象

标准浏览器：通过事件函数第一个参数传进来

* IE与DOM标准浏览器的事件流

|  |  |  |
| --- | --- | --- |
|  | IE | DOM标准 |
| 事件类型 | 只支持冒泡型事件 | 捕获型与冒泡型 |
| 绑定事件 | attachEvent | addEventListener |
| 移除绑定事件 | detachEvent | removeEventListener |
| 阻止冒泡 | event.cancelBubble=true | event.stopPropagation() |
| 取消浏览器的默认行为 | event.returnValue=false  return false | event.preventDefault() |
| 执行顺序 | 从里到外 | 捕获：从外到里  冒泡：从里到外 |

决定事件捕获与否的是绑定事件函数的参数，attachEvent没有第三个参数，addEventListener有第三个参数，正是第三个参数决定捕获还是冒泡，false为冒泡，true为捕获。所以IE下没有捕获，DOM标准下有捕获

* 事件绑定函数的区别

IE：obj.attachEvent(事件名称，事件函数);

1.没有捕获

2.事件名称有on

3.事件函数执行的顺序：标准浏览器为正序 非标准浏览器(ie)为倒序

4.this指向window

标准：obj.addEventListener(事件名称，事件函数，是否捕获);

1.有捕获

2.事件名称没有on

3.事件执行的顺序是正序

4.this指向触发该事件的对象

* call & apply

函数的调用可以直接写函数名()，也可以用call或者apply来调用

* call有两个作用

1、调用函数

fn1() == fn1.call() 效果是一样的，平时的函数调用也可以用call

1. 改变this指向

function fn1(a, b) {

alert(this);

alert(a + b);

}

fn1(); //this指向window

fn1.call('kaivon',12,5); //this指向kaivon

fn1.call(null, 10, 20); //第一个参数改变this如果为空或null，this还指向window，从第二个参数开始就是原来函数的参数列表

* apply也是两个作用

1. 调用函数

fn1() == fn1.apply() 效果是一样的，平时的函数调用也可以用apply

1. 改变this

fn1(); //this指向window

fn1.apply('kaivon',[12, 5]); //this指向kaivon

fn1.apply(null, [10, 20]); //第一个参数改变this如果为空或null，this还指向window，从第二个参数开始就是原来函数的参数列表，与call的不同是，函数的实参需要放到数组里，不然报错

* 封装事件绑定函数

function bind(obj, evname, fn) {

if (obj.addEventListener) {

obj.addEventListener(evname, fn, false);

} else {

obj.attachEvent('on' + evname, function() {

fn.call(obj);

});

}

}

* 鼠标滚轮事件

ie/chrome... : onmousewheel

event.wheelDelta //判断滚动方向

上：120

下：-120

firefox : DOMMouseScroll 必须用addEventListener

event.detail //判断滚动方向

上：-3

下：3

* 阻止默认事件

return false阻止的是 obj.on事件名称=fn 所触发的默认行为

同样一个事件，不同的绑定事件方法，阻止默认事件也是不同的方法。用on的事件直接可以return false阻止默认事件，用事件绑定函数绑定的事件必需用特有的方法去阻止

addEventListener绑定的事件需要通过event下面的preventDefault();

1. 获取页面元素的位置

* clientHeight和clientWidth

都是只读属性，不能对它们赋值

网页上的每个元素，都有clientHeight和clientWidth属性。这两个属性指元素的内容部分再加上padding的值，不包括border和滚动条占用的空间

document元素的clientHeight和clientWidth属性，就代表了网页的大小

function getViewport() {

//判断文档模型BackCompat(非标准文档)：标准兼容模式关闭；CSS1Compat(标准文档)：标准兼容模式开启。

//大多数情况下，都是document.documentElement.clientWidth返回正确值。但是，在IE6的quirks模式中，document.body.clientWidth返回正确的值，因此函数中加入了对文档模式的判断(很少见)。

if (document.compatMode == "BackCompat") {

return {//非标准文档下IE获取文档宽高

width: document.body.clientWidth,

height: document.body.clientHeight

}

} else {

return {//标准文档下IE获取文档宽高

width: document.documentElement.clientWidth,

height: document.documentElement.clientHeight

}

}

}

* offsetWidt与offsetHeight

元素的实际宽度与高度，包括border

* 获取元素有相对定位父级的位置

对于表格和iframe中的元素不适用

function getElementPos(element) {

var aLeft = element.offsetLeft;

var aTop = element.offsetTop;

var current = element.offsetParent;

while (current !== null) {

aLeft += current.offsetLeft;

aTop += current.offsetTop;

current = current.offsetParent;

}

return {x: actualLeft, y: actualTop};

}

1. cookie

存储数据，当用户访问了某个网站（网页）的时候，可以通过cookie来向访问者电脑上存储数据。有时候需要在服务器环境下操作

cookie的特性

1. 不同的浏览器存放的cookie位置不一样，是不能通用的
2. cookie的存储是以域名形式进行区分的
3. cookie的数据可以设置名字
4. 一个域名下存放cookie的个数以及每个cookie内容的大小(不超过4kb，否则会被截掉)都是有限制的，不同的浏览器都不同

IE6及以下 20个

IE7+ 50个

Firefox 50个

Chrome、Safari没有做硬性限制

1. 设置cookie的时候内容最好encodeURI
2. cookie默认是临时存储的，当浏览器关闭进程的时候自动销毁
3. IE和Opera会清理近期最少使用的cookie，Firefox会随机清理cookie
4. IE 提供了一种存储可以持久化用户数据，叫做uerData，从IE5.0就开始支持。每个数据最多128K，每个域名下最多1M。放在缓存中，如果缓存没有清理，那么会一直存在
5. 不安全。如果cookie被人拦截了，那人就可以取得所有的session信息。即使加密也与事无补，因为拦截者并不需要知道cookie的意义，他只要原样转发cookie就可以达到目的了
6. 每次你请求一个新的页面的时候Cookie都会被发送过去，这样无形中浪费了带宽
7. cookie还需要指定作用域，不可以跨域调用

我们通过document.cookie来获取当前网站下的cookie的时候，得到的字符串形式的值，他包含了当前网站下所有的cookie。他会把所有的cookie通过一个分号+空格的形式串联起来username=leo; age=32; color=blue;

function setCookie(key, value, t) { //设置cookie

var oDate = new Date();

oDate.setDate( oDate.getDate() + t );

document.cookie = key + '=' + encodeURI(value) + ';expires=' + oDate.toGMTString();

//设置过期时间

}

function getCookie(key) { //获取cookie

var arr1 = document.cookie.split('; ');

for (var i=0; i<arr1.length; i++) {

var arr2 = arr1[i].split('=');

if ( arr2[0] == key ) {

return decodeURI(arr2[1]);

}

}

}

function removeCookie(key) { //删除cookie

setCookie(key, '', -1);

}

本地存储

useData：从IE5.0就开始支持。每个数据最多128K，每个域名下最多1M。这个持久化数据放在缓存中，如果缓存没有清理，那么会一直存在

sessionStorage：储一个会话（session）中的数据，会话结束后数据也随之销毁

localStorage：持久化的本地存储，除非主动删除数据，否则数据是永远不会过期

1. Ajax

* get与post的区别

get 通过网址，容量小（url有限制，浏览器不同限制不同，一般在2000个字符），安全性差，有缓存，用于获取数据，只能传字符串类型数据，传中文需要转码，通过url传值

post 通过http content(不通过网址)，容量大（理论上无限制），安全性好一点，没缓存，用于提交数据，传中文、二进制也没问题，传中文不用转码（请求头设置了转码形式），通过表单传值

* 同步&异步

**同步：**提交请求——等待服务器处理——处理完毕后返回，这个期间客户端浏览器不能干任何事。

阻塞模式：前面代码会影响后面代码执行，前面没走完，后面就别想走。

同步模式用的很少，如果用的话可以用回调函数来代替

**异步：**请求通过事件触发——服务器处理（这时浏览器仍然可以做其它事情）——处理完毕。

非阻塞模式：前面代码不会影响后面代码执行，前面没走完，后面也可以走

Js里都是同步的，定时器是异步的

同步就是你叫我去吃饭，我听到了就和你一块去吃饭。如果没听到，你就不停的叫，直到我告诉你听到了，才一起去吃饭

异步就是你叫我，然后自己去吃饭，我得到消息后可能立即走，也可能下班才去吃饭

* 跨域

在不同的域名下进行数据交互就会出跨域，原因是ajax是通过XMLHttpRequest来进行数据交互的，而XMLHttpRequest是不允许跨域的

1. 主域与子域之间：www.a.com与b.a.com
2. 不同的域名下边：www.a.com与[www.b.com](http://www.b.com)，即使指向同一个IP地址

Iframe里有一个别的地址的iframe，也是不能用dom操作里面的iframe(被包含的子页面)这是安全限制问题

解决方法：

1. 主域与子域之间跨域设置document.domain=’a.com’(主域名的地址);
2. 不同域名下能过服务器的代理，服务器那边对XMLHttpRequest做一些运算。优点是可以在任何域名下跨域，缺点是会增加服务器的带宽。用php去请求数据，把php文件放到与js同域名下，起到一个中转作用
3. 用script标签即jsonp的方法，script可以引进任何文件，只要返回的数据符合js的语法即可用。Jsonp为json+padding(内填充)，意思为把json当作一个内填充的东西填到一个盒子(回调函数)里，再利用回调函数的方法取到它

Jsonp的顺序：先定义callBack函数，再请求数据。因为script请求的数据是以callBack的形式回调的，script标签一放到页面中，就会去调用callBack函数，如果没有定义的就会出错，先定义后调用

Jsonp的缺点：不受同源策略的限制，多少会有安全隐患；它只支持GET请求而不支持POST请求

1. iframe

利用iframe的src，把数据放到地址问号后面，再用self.iframe.hash来取值

1. Window.name与iframe的hash道理一样
2. Flash
3. Html5的postMessage（不受同源策略影响）以及XMLHttpRequest2.0版本

* 同源策略

Ajax之所以不能跨域其实是因为XMLHttpRequest受到同源策略的限制，只能让它访问同源下的数据，不能访问不同源下的数据

一段脚本只能读取来自于同一来源的窗口和文档的属性，这里的同一来源指的是主机名(域名)、协议(http/https)和端口号的组合。在没明确授权的情况下，不能读写对方的资源，它是浏览器最核心也最基本的安全功能。

IE未将端口号加入到同源策略的组成部分之中

<http://www.360.cn/c/d.html> 成功

<http://www.360.cn/d/e/f.html> 成功

<https://www.360.cn/a/c.html> 失败 协议不同

<http://www.360.cn:81/a/c.html> 失败 端口不同

<http://www.sina.com.cn/a/b.html> 失败 主机名不同

带src属性的<script><img><iframe><link>等标签是不需要遵守同源策略的，但是通过src加载的资源，浏览器限制了javascript的权限，不能进行各种的读写。从而，即使请求发了，敏感数据回来了，也是取不 到的。

**变更源**

1、document.domain

如果在 http://a.kaivon.com/ 中的一个脚本执行了下列语句：

document.domain = "kaivon.com";

执行后页面将会把a.kaivon.com与kaivon.com设置为同一个源。但是在kaivon.com中就不能再设置document.domain为其它域名，只能设置为document.domain = "kaivon.com";

1. Adobe Flash使用crossdomain.xml文件控制Flash接收数据
2. Silverlight是微软推出的一款插件，其与Flash使用相同的同源策略。然而，其使用的clientaccess-policy.xml
3. Adobe Reader PDF

绕过同源策略方法：

postMessage、JSONP、WebSocket

<http://www.freebuf.com/articles/web/65468.html>

* Ajax库

function ajax(method, url, data, success) {

var xhr = null; //创建ajax对象

try {

xhr = new XMLHttpRequest(); //除了IE6及以下的所有浏览器

} catch (e) {

xhr = new ActiveXObject('Microsoft.XMLHTTP'); //IE6及以下

}

/\*

如果用if else 判断XMLHttpRequest的话，IE6会出错，原因是这个东西IE6下不存在，判断一个不存在的东西，IE6会报错，要用window.XMLHttpRequest去判断。

try catch可以避免，先走try里的代码，如果有错的话就走catch里，并把错误信息放到e里面。如果没有错catch就不会走。try里可以出错，有错误浏览器也不会报错

如果try里可以throw new Error('有错了')是主动报错。没有错也可以主动报错，报完错后就走catch,并把错误信息放到e里

\*/

if (method == 'get' && data) { //get方式，通过url提交，需要拼接url

url += '?' + data;

}

xhr.open(method,url,true); //连接服务器（方法，路径，是否异步）

if (method == 'get') {

xhr.send();

} else {

//xhr.setRequestHeader（文档，文档格式）

xhr.setRequestHeader('content-type', 'application/x-www-form-urlencoded');

xhr.send(data);

}

xhr.onreadystatechange = function() { //状态改变所触发的事件

if ( xhr.readyState == 4 ) {

if ( xhr.status == 200 ) {

success && success(xhr.responseText);

} else {

alert('出错了,Err：' + xhr.status);

}

}

}

}

提交数据方式的区别：

get通过url提交，需要拼接url

post通过http content（HTTP协议的响应头）提交，需要设置请求头，而且数据要放到send方法的参数里

responseText：返回字符串格式的数据

responseXML：返回XML格式的数据

readyState：ajax工作状态

0    （初始化）还没有调用open()方法

1    （载入）已调用send()方法，正在发送请求

2    （载入完成）send()方法完成，已收到全部响应内容

3    （解析）正在解析响应内容

4    （完成）响应内容解析完成，可以在客户端调用了

status：http(服务器)状态码:

1开头：消息

2开头：成功

3开头：重定向

4开头：请求错误

5开头：服务器错误

**注意：**

1. 请求的地址后面要加一个时间戳，防止IE下的缓存
2. 提交中文数据的时候一定要encodeURI进行编码，取到的数据有中文也要encodeURI
3. 前后端的请求方式要统一，都用get或者都用post，如果两种不同的方式会取不到数据

* 解析请求过来的数据

1. json.parse()

推荐方法，不过IE67需要引入json.js文件到页面中去解决

1. new Function

return (new Function('return '+data))();

1. eval

它的功能是把对应的字符串解析成JS代码并运行，不安全，耗性能，会走两次，一次解析成js语句，一次执行

eval("("+data+")") //不推荐存在安全问题

由于json是以”{}”的方式来开始以及结束的，在JS中，它会被当成一个语句块来处理，所以必须强制性的将它转换成一种表达式

加上圆括号的目的是迫使eval函数在处理JavaScript代码的时候强制将括号内的表达式（expression）转化为对象，而不是作为语句（statement）来执行。举一个例子，例如对象字面量{}，如若不加外层的括号，那么eval会将大括号识别为JavaScript代码块的开始和结束标记，那么{}将会被认为是执行了一句空语句。

* XMLHttpRequest Level2

var xhr = new XMLHttpRequest();

if(typeof xhr.withCredentials === 'undefined'){

//不支持

}esle{

//支持

}

**与Level1的区别**

1. 老版本只支持文本数据的传送，无法用来读取和上传二进制文件
2. 传送和接收数据时，没有进度信息，只能提示有没有完成
3. 不能跨域

在标准浏览器下，XMLHttpRequest对象已经是升级版本，支持了更多的特性，可以跨域。但是，如果想实现跨域请求，还需要后端的相关配合才可以

后端在输出的时候需要加一个允许访问该资源的域的头部信息

header('Access-Control-Allow-Origin:http://www.a.com'); //允许a.com访问本网站的内容

XMLHttpRequest增加很多功能，他也不推荐使用onreadystatechange这个事件来监听，推荐使用onload

IE6以上支持办法：oXDomainRequest

var oXDomainRequest = new XDomainRequest();

oXDomainRequest.onload = function() { //IE下的新事件

alert(this.responseText);

}

oXDomainRequest.open('get', 'http://www.b.com/ajax.php', true);

oXDomainRequest.send();

<https://msdn.microsoft.com/en-us/library/cc288060(VS.85).aspx>

**timeout**

新增加的超时功能

xhr.timeout = 3000;

xhr.ontimeout = function(){

alert('请求超时！');

}

**FormData对象**

用来模拟表单，表单在发送数据的时候需要一个名字与一个值，FormData对象提供了两个参数

var formData = new FormData();

formData.append('username', '张三');

formData.append('id', 123456);

xhr.send(formData);

**上传文件：**

利用input type="file"控件把文件上传到服务器

文件存放于input的files中，是个数组存放多个文件的集合

var formData = new FormData();

for (var i = 0; i < files.length;i++) {

formData.append('name', files[i]);

}

xhr.send(formData);

**接收二进制数据**

**进度信息**

传送数据的时候，有一个progress事件，用来返回进度信息。

它分成上传和下载两种情况。下载的progress事件属于XMLHttpRequest对象，上传的progress事件属于XMLHttpRequest.upload对象。

xhr.onprogress = updateProgress; //下载进度事件

xhr.upload.onprogress = updateProgress; //上传进度事件

function updateProgress(event) {

//lengthComputable 为真的时候event.total才有数字，否则为0

if (event.lengthComputable) {

//event.total 传输的总字节

//event.loaded 已经传输的字节

var percentComplete = event.loaded / event.total;

}

}

与progress事件相关的，还有其他五个事件，可以分别指定回调函数：

load //传输成功完成

abort //传输被用户取消

error //传输中出现错误

loadstart //传输开始

loadEnd //传输结束，但是不知道成功还是失败

//oMyFile.value 获取到的是file控件的value值，是文件的名字，不是选择的文件

//oMyFile.files 存放的是选中的文件列表

//每个文件中包含很多信息：大小、name、路径、最后修改时间等console.log(oMyFile.files[0])

var xhr = new XMLHttpRequest();

xhr.onload = function() { //ajax完成事件

alert('OK,上传完成');

}

var oUpload = xhr.upload; //上传文件对象

oUpload.onprogress = function(ev) {

var iScale = ev.loaded / ev.total;

oDiv2.style.width = 300 \* iScale + 'px';

oDiv3.innerHTML = iScale \* 100 + '%';

}

xhr.open('post', 'post\_file.php', true);

xhr.setRequestHeader('X-Request-With', 'XMLHttpRequest'); //post方式必需设置请求头，这里的是固定格式

var oFormData = new FormData(); //通过FormData来构建提交数据

oFormData.append('file', oMyFile.files[0]); //第一个参数是后端定的，前端必需与后端保持一致

xhr.send(oFormData);

1. 面向对象

* 对象：一组包含数据的属性和对属性中包含数据进行操作的方法，称为对象
* 面向对象的特性

1. 抽象：把主要的属性抽出来，把最主要的跟问题相关的特征放在一起
2. 封装：把内部的结构、内部的细节封起来。只能通过对象去访问
3. 继承：遗传，从父类上继承出一些方法、属性，子类又有一些自己的属性。分为多重继承与多态

多态：多对象的不同形态（js里用的不多）。就像一个usb口，可以让很多设备都连接电脑，同是连接电脑以后做不同的事情。只用写一个接口就好，后端用。

多重继承：同时继承两个类的属性与方法

* this & new

function Show(){

alert(this);

}

Show(); //[object Window]

new Show(); //[object Object] new的对象,不赋值直接调的话this就指向Object

**new 的作用：**

1. 创建一个新对象
2. this指向该对象
3. 继承了应该对象的原型

所谓构造函数，就是通过这个函数生成一个新对象

如果在调用函数的时候在前面加一个new,this会指向新创建的对象。而构造函数里用this后，系统会自动创建一个对象，在函数的最后会自动返回new出来的对象

new后边的函数就叫构造函数

当一个构造函数创建完后，JS会自己动为这个构造函数添加一个constructor属性并指向这个构造函数Show.prototype.constructor=Show; 然后构造函数new出来的每一个实例的constrator都指向Show

**constructor的隐患：**

1. 可以手动修改

Show.prototype.constructor=Array;

var a1=new Show();

alert(a1.constructor); //function Array(){[native code]} 指向了Array

1. 写法不同，指向就不同

function Show(){

}

Show.prototype.name='kaivon'; //这样写是往原型上添加属性，原型不会受影响

Show.prototype.age=20;

var a1=new Show();

alert(a1.constructor); //function Show(){}

Show.prototype={ //这样写就不是添加了，而是用一个json对其覆盖

//constructor:Show, 修正constructor的指向问题

name:'kaivon',

age:20

};

var a1=new Show();

alert(a1.constructor); //function Object(){[native code]}

两种原型写法不同，一种是添加，一种是覆盖。所以他们的constructor指向就会不同。所以创建一个构造函数的时候最好要把它的constructor指向修正一下Show.constructor=Show;

**当前的方法属于谁，this就是谁**

1. 找this的话就看函数前面有没有点，有点的话this就是点前面的对象，没有点this就是window
2. 当new去调用函数，这时的this不会看前面有没有点，this指的是new出来的对象。而且函数的返回值直接就是this
3. 隐式返回：正常的函数里面如果没有return的话，返回undefined，有return的话返回return后面的东西。如果用了new了一个函数，函数里面不用return，会默认返回new创建的对象(this)。var p1=new Person(); this指的就是p1

* 工厂方式（封装函数）

1. 原料 2、加工 3、出厂

function createPerson(name, age){

var obj = new Object(); //原料

//加工

obj.name = name;

obj.age = age;

obj.showName = function() {

alert(this.name);

};

obj.showAge = function() {

alert(this.age);

};

return obj; //出厂

}

var obj = createPerson('kaivion', '24')

obj.showName();

var obj2 = createPerson('blue', '27');

alert(obj.showName == obj2.showName); //false

缺点：

1. 没有new ，系统对象创建的时候都带new
2. 函数重复导致资源浪费

解决办法：称为混合方法，用构造函数与原型结合的方式

function createPerson(name, age) {

//系统会自动生成一个对象 var this=new Object();

this.name = name;

this.age = age;

this.showName

//系统会自动返回this return this;

}

createPerson.prototype.showName = function() {

alert(this.name);

};

createPerson.prototype.showAge = function() {

alert(this.age);

};

//调用的时候new一个

var obj = new createPerson('kaivion', '24')

obj.showName();

var obj2 = new createPerson('blue', '27');

alert(obj.showName == obj2.showName); //true,它们的方法都源自原型，所以相同

* 传值与传引用

基本数据(原始值)：数字、字符串、布尔值、undefined、空对象等。

对象数据(对象)：对象、函数、数组。

基本数据类型只有一个值，操作它们也仅仅操作值。对象数据类型不仅有值，还有一个在内存里的引用地址，操作它们不光操作值，还操作引用地址

**复制一个对象，如果让整个对象名相等，那复制的是引用。如果让对象里每一个属性的值相等，那复制的是值**

**比较：**基本类型只比较值，对象类型同时比较值与引用地址

var a=1;

var b=1;

alert(a==b) //true 值相同

var a=[1,2,3];

var b=[1,2,3];

alert(a==b) //false 值相同，引用地址不同

var a = [1,2,3];

var b = a; //赋值会把值与引用地址都赋过去

alert( a==b ); //true 值相同，引用地址相同

**赋值：**基本类型赋值，对象类型同时赋值与引用地址

var a=1;

var b=a; //把a赋值给b，没有引用地址，a与b相同

b=2 //b又重新赋值，也没有引用地址，a与b不同

alert(a==b) //false

alert(a) //1

alert(b) //2

var a=[1,2,3];

var b=a; //把a赋值给b，引用地址也赋给b，a与b相同，a的引用不变

b=[1,2,3,4]; //b又重新赋值，引用地址也重新赋值，a与b不同

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

alert(a); //1,2,3

**修改：**基本类型修改值，对象类型同时修改值与引用地址

var a=5;

var b=a; //把a赋值给b，没有引用地址，a与b相同

b+=3; //b进行修改，没有引用地址，a与b不同

alert(b) //8

alert(a) //5

var a=[1,2,3];

var b=a; //把a赋值给b，引用地址也赋给b，a与b相同

b.push(4); //b进行修改，共用引用地址也修改，a与b相同，都变成修改后的值

alert(b); //1,2,3,4 不仅复制了值，而且复制了引用，在内存中都指向一个地址，改一个都会变

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

修改与赋值虽然都会改对象类型的引用地址（新生成个引用地址），但还是有所不同，赋值的话原来(a)的引用地址不变。修改的话原来(a)的引用地址会变成新生成的引用地址

**例子：**

传引用

var arr1 = [1, 2, 3];

var arr2 = [];

arr2 = arr1;

arr2.push(4);

alert(arr1); //1,2,3,4

alert(arr2); //1,2,3,4

传值

var arr1 = [1, 2, 3];

var arr2 = [];

for (var i = 0; i < arr1.length; i++) {

arr2.push(arr1[i]);

}

arr2.push(4);

alert(arr1); //1,2,3

alert(arr2); //1,2,3,4

递归：递代表传递，归代表返回。就是一层一层的传递，直到最后一层后再返回（从后向前）再传递

1. 函数调用函数本身
2. 最后一次判断一个终止条件

function test(n) {

if(n==1){

return 1; //会执行4次，一层一层往回返

}

return n\*test(n-1); //一层一层传

}

alert(test(4)); //1\*2\*3\*4=24

深度克隆对象，利用递归的形式循环找到最深嵌套的属性，并赋值。复制的是每个属性的值，而不是整个对象引用的复制。克隆后的对象修改属性不会影响被克隆的对象

var obj = {a: {b: 10}};

function copy(obj) { //浅拷贝，会有问题。如果对象的属性是多层嵌套的话，只能克隆第一层属性，而里面的属性就不是复制值了，复制的是引用，修改新对象的属性依然影响老对象的属性

var newObj = {};

for (var attr in obj) {

newObj[attr] = obj[attr];

}

return newObj;

}

//这套程序会有问题，如果对象里有嵌套的数组，会把数组也转成对象类型，以至于不能使用length属性（已解决）

function deepCopy(obj) { //深拷贝，参数上来就规定了类型，只要不是对象类型就不会停止，如果是别的类型就会一直递归

if (typeof obj != 'object') { //如果不是对象的话就是单个属性没有嵌套了

return obj;

}

if(obj instanceof Array){ //如果是数组，要复制成数组的形式，要不会把数组复制成对象，没有length属性

var newObj=[];

}else{

var newObj={};

}

for (var attr in obj) {

//方括号代表的是值，函数的参数传的也是值

newObj[attr] = deepCopy(obj[attr]);

}

return newObj;

}

var obj2 = deepCopy(obj);

obj2.a.b = 20;

alert(obj.a.b); //10

* 原型&原型链

原型：

类、构造函数、原型其实指的都是同一个东西

指的就是对象的构造函数。作用是改写对象下面的公用方法或属性，让它们在内存中只存在一份，提高性能

（原因是：创建一个对象就有一个方法，这个方法与构造函数里的方法是相同的，但是它们的地址不同。如果有很多对象，那内存中就会有很多个相同的方法，消耗性能。所以就可以把方法写在构造函数的prototype里，内存中就只有一份，再用for in复制它的所有prototype到子类的prototype中，不仅每个子类复制了方法的值，而且复制了方法的地址。提高性能）

Js是基于原型的程序，面向对象都是用构造函数创建对象，给对象添加属性，给prototype添加方法。只有对象才有属性。

prototype就相当于css里的class，可以给一组元素添加样式，往类上加东西。普通的属性相当于style可以覆盖原型里的属性跟方法。

var arr=[];

arr.number=10;

Array.prototype.number=20;

alert(arr.number) //10

原型链：

实例对象与原型之间的链接叫原型链(\_proto\_)在firebug dom里可以看到。原型链有点像变量的作用域，new一个对象后取他的属性或方法，他会先在自己身上找，如果有的话就不找了，如果没有的话就会往原型身上找（由里到外）。这就说明为什么一个属性或方法同时存在于原型与new对象上，它的值最终会取new对象身上的

原型链最外层是Object，所以的方法或属性找不到都会取Object.prototye.方法/Object.属性

* hasOwnProperty

看是不是对象自身（唯一）下面的属性，是自己的属性还是继承prototpyec对象的属性。它是原型链最外层Object身上的方法

var arr = [];

arr.num = 10;

Array.prototype.num2 = 20;

alert(arr.hasOwnProperty('num')); //true

alert(arr.hasOwnProperty('num2')); //false num2是在原型下赋值的，说明所有数组对象都有num2的属性，就不是唯一了

* constructor

查看对象的构造函数，用来判断类型

function Aaa(){

}

var a1 = new Aaa();

alert( a1.constructor ); //function Aaa(){}

var arr = [];

alert( arr.constructor == Array ); //true

只要定义一个构造函数，下面这句话系统就会自动给你加上。不过可以手动去改（没必要），但是会在不经意间被改掉

Aaa.prototype.constructor = Aaa; //每一个函数都会有的，都是自动生成的

constructor被修改例子：

function Aaa(){

}

/\*Aaa.prototype.name = '小明';

Aaa.prototype.age = 20;\*/

Aaa.prototype = {

constructor : Aaa, //如果不加的话constructor就是Object

name : '小明',

age : 20

};

var a1 = new Aaa();

alert( a1.constructor ); //Aaa 如果json里不加constructor的话，会弹出Object

For in系统自带的属性与方法是找不到的

function Aaa(){

}

Aaa.prototype.name = 10;

Aaa.prototype.constructor = Aaa;

for( var attr in Aaa.prototype ){

alert(attr); //找不到constructor

}

* instanceof

查看对象与构造函数在原型链上是否有关系，语法是前后用空格。也可以做类型判断

function Aaa(){

}

var a1 = new Aaa();

alert( a1 instanceof Aaa); //true

alert( a1 instanceof Object ); //true 每个对象的最外层都是Object

var arr = [];

alert( arr instanceof Array ); //true

* tostring()

系统对象下面都是自带的，自己写的对象都是通过原型链找Object下面的

1. 把对象转成字符串，任何一个对象都可以调用这个方法
2. 转进制
3. 做类型判断，推荐方法

var arr = [];

alert( arr.toString == Object.prototype.toString ); //false arr为系统对象，它的toString是自身的

function Aaa(){

}

var a1 = new Aaa();

alert( a1.toString == Object.prototype.toString ); //true a1为自己创建的对象，它的toString是Object

var arr = [];

alert( Object.prototype.toString.call(arr) == '[object Array]' ); //true '[object Array]'

window.onload = function(){

var oF = document.createElement('iframe');

document.body.appendChild( oF );

var ifArray = window.frames[0].Array;

var arr = new ifArray();

//操作iframe里的对象做类型判断，constructor与instanceof都会失效

//alert( arr.constructor == Array ); //false

//alert( arr instanceof Array ); //false

alert( Object.prototype.toString.call(arr) == '[object Array]' ); //true

};

* 继承

在原有对象的基础上稍做修改，得到一个新对象，不影响原有对象的功能

子类不影响父类，子类可以继承父类的一些功能 ( 代码复用 )

call的用法：第一个参数为this的值，其它参数跟着后面写

function show(a, b) {

alert('this是：' + this + '\na是：' + a + '\nb是：' + b);

//this是[object Window] a是12 b是5

};

show(12, 5); //简写

show.call(window, 12, 5); //函数调用的本质写法

1. **拷贝继承**（通用型的 有new或无new的时候都可以）

属性继承：用call调用父类的构造函数

方法继承：for in循环prototype赋值(jquery也是采用拷贝继承extend)。

function CreatePerson(name,sex){ //父类

this.name = name;

this.sex = sex;

}

CreatePerson.prototype.showName = function(){

alert( this.name );

};

var p1 = new CreatePerson('小明','男');

//p1.showName();

function CreateStar(name,sex,job){ //子类

CreatePerson.call(this,name,sex); //用call的方法复制父类的属性，且改变this指向子类

this.job = job;

}

//方法复制用for in,如果直接复制，是复制了引用地址，一个改变全改变

extend( CreateStar.prototype , CreatePerson.prototype );

CreateStar.prototype.showJob = function(){

//子类创建自己的方法

};

var p2 = new CreateStar('黄晓明','男','演员');

p2.showName(); //黄晓明 弹出子类自己的name

//for in封装函数

function extend(obj1,obj2){

for(var attr in obj2){

obj1[attr] = obj2[attr];

}

}

1. **类继承**（new构造函数）

Js中没有类的概念，后端有。把构造函数想像成类。类继承是通过构造函数来继承的方式，从后端衍变过来的

**属性和方法继承的时候，要分开继承**。父类子类都用构造函数创建，子类用的时候去new一个新的对象用

属性继承：用call调用父类的构造函数

方法继承：

1. 声明一个空function对象
2. 空对象的prototype=父类的prototype（直接赋值，传的是引用）
3. 子类prototype=new一个空对象
4. 子类prototype.constructor=子类构造函数（修正constructor指向）

**问题版：**

function Aaa(){ //父类

this.name = 'kaivon';

}

Aaa.prototype.showName = function(){

alert( this.name );

};

function Bbb(){ //子类

}

//Bbb.prototype=Aaa.prototype 或者

Bbb.prototype = new Aaa();//类式继承，面试会问到的一句话实现继承（有两个问题）

var b1 = new Bbb();

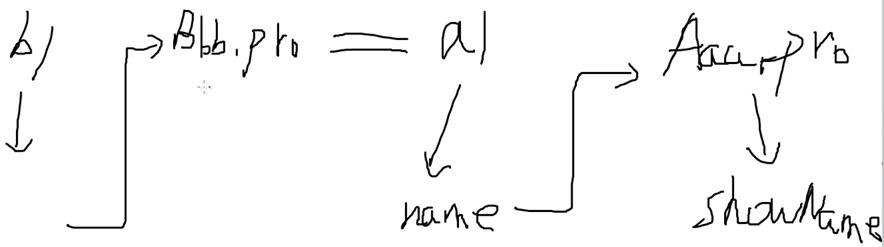
b1.showName(); //kaivon b1会继承name属性与showName方法

b1.name //kaivon 都可以找到

解释：

我们把Aaa的子类new Aaa()称为a1

1. Aaa为构造函数，a1是Aaa的子类，它们俩个中间有原型链
2. 把a1赋值给Bbb的原型，属于引用关系
3. Bbb为构造函数，b1是Bbb的子类，它们俩中间也有个原型链
4. b1调showName，先找b1有没有showName，没找到，再找Bbb，Bbb也没有，就会找与它相等的a1，发现它也没有，然后再找Aaa，发现Aaa有showName，所以就能调用



问题1：b1.constructor //Aaa

原因：new Aaa()直接复制给Bbb了，new Aaa()会把Bbb的constructor给覆盖，把指向修改成Aaa了

问题2：如果Aaa的this.name=[1,2,3,4];b1.name.push(4);再创建一个实例var b2=new Bbb();b2的name就是修改后的值了

原因：Aaa与Bbb是互相引用的关系，它们找的都是同一个Aaa的name，一个实例修改了Aaa的name，另一个继续引用它的name肯定会变了

**无问题版：**

function Aaa(){ //父类

this.name = [1,2,3];

}

Aaa.prototype.showName = function(){

alert( this.name );

};

function Bbb(){ //子类

Aaa.call(this);

}

var F = function(){}; //解决问题2

F.prototype = Aaa.prototype;

Bbb.prototype = new F();

Bbb.prototype.constructor = Bbb; //修正指向问题，不修正会指向Aaa(解决问题1)

var b1 = new Bbb();

b1.name.push(4);

var b2 = new Bbb();

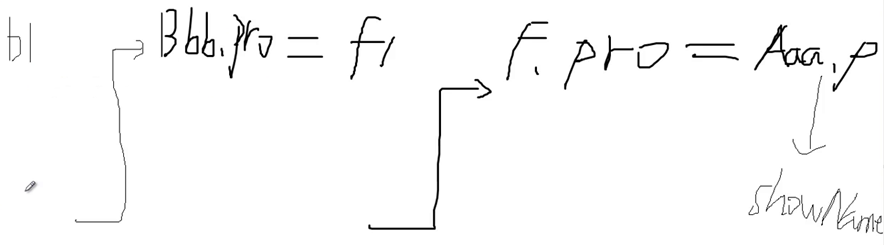
alert( b2.name ); //[1,2,3]b1添加name，b2不会影响，都有this.name

解释：

我们把Aaa的子类new Aaa()称为a1

1. Aaa为构造函数
2. F为一个空函数，把Aaa的原型赋给F的原型，属于引用关系
3. new F属于F的子类，它们俩中间也有个原型链
4. 把new F赋值给Bbb的原型，也属于引用关系
5. b1是Bbb的子类，它们俩中间也有个原型链
6. b1调showName，先找b1有没有showName，没找到，再找Bbb，Bbb也没有，就会找与它相等的new F，发现它也没有，然后再找F，F也没有，再找Aaa，发现Aaa有showName，所以就能调用

如果找name属性的话会发再找不到，因为上面的F空函数没有属性，只有方法，所以下边的赋值只能赋值方法，没有给属性赋值所以找不到属性。也就避免了，实例修改属性全都变的问题



1. **原型继承**（无new的对象）

借助原型prototype来实现对象继承

var a = {

name : '小明'

};

var b = cloneObj(a);

b.name = '小强';

alert( a.name ); //小明

alert( a.name ); //小强

function cloneObj(obj){

var F = function(){};

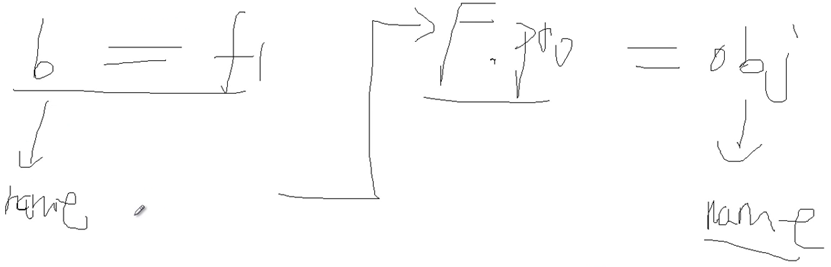
F.prototype = obj;

return new F();

}

解释：

1. 函数F为空函数，把F.prototype赋值给参数obj，形成引用关系
2. 把参数new F直接返回出去赋值给b，b现new F形成引用关系



* 改写面向对象的方法

原则：把不一样的东西放在构造函数里，把一样的东西放在原型里，用构造函数来加属性，用原型加方法。

改写面向对象的程序：

1. 不能有函数嵌套
2. 把变量变成对象的属性
3. 把函数变成对象的方法
4. 方法与属性都得加this
5. 调用函数的时候需要先存一下this，然后用function来调用
6. onload中创建对象

function Drag(id) {

var \_this = this;

this.disX = 0;

this.disY = 0;

this.oDiv = document.getElementById(id);

this.oDiv.onmousedown = function(ev) {

\_this.fnDown(ev);

return false;

};

};

Drag.prototype.fnDown = function(ev) {

var \_this = this;

var oEvent = ev || event;

this.disX = oEvent.clientX - this.oDiv.offsetLeft;

this.disY = oEvent.clientY - this.oDiv.offsetTop;

document.onmousemove = function(ev) {

\_this.fnMove(ev);

}

document.onmouseup = function() {

\_this.fnUp();

};

}

Drag.prototype.fnMove = function(ev) {

var oEvent = ev || event;

this.oDiv.style.left = oEvent.clientX - this.disX + 'px';

this.oDiv.style.top = oEvent.clientY - this.disY + 'px';

}

Drag.prototype.fnUp = function() {

document.onmousemove = null;

document.onmouseup = null;

}

function LimitDrag(id) {

Drag.call(this, id);

}

for (var i in Drag.prototype) {

LimitDrag.prototype[i] = Drag.prototype[i];

}

LimitDrag.prototype.fnMove = function(ev) {

var oEvent = ev || event;

var l = oEvent.clientX - this.disX;

var t = oEvent.clientY - this.disY;

if (l < 0) {

l = 0;

} else if (l > document.documentElement.clientWidth - this.oDiv.offsetWidth) {

l = document.documentElement.clientWidth - this.oDiv.offsetWidth;

}

if (t < 0) {

t = 0;

} else if (t > document.documentElement.clientHeight - this.oDiv.offsetHeight) {

t = document.documentElement.clientHeight - this.oDiv.offsetHeight;

}

this.oDiv.style.left = l + 'px';

this.oDiv.style.top = t + 'px';

};

window.onload = function() {

var obj1 = new Drag('div1');

var obj2 = new LimitDrag('div2');

};

* 创建类的三个方法

1. 构造函数法

这是经典方法，也是教科书必教的方法。它用构造函数模拟"类"，在其内部用this关键字指代实例对象。缺点是，比较复杂，用到了this和prototype，编写和阅读都很费力

function Cat() {//创建构造函数

this.name = "大毛";

}

//共用的属性与方法都添加到prototype上

Cat.prototype.makeSound = function() {

alert("喵喵喵");

}

var cat1 = new Cat();　 //new 一个实例

alert(cat1.name); //大毛

cat1.makeSound(); //喵喵喵

1. Object.create()法

为了解决"构造函数法"的缺点，更方便地生成对象，Javascript的国际标准ECMAScript第五版（目前通行的是第三版），提出了一个新的方法Object.create()，IE8以上支持。这种方法比"构造函数法"简单，但是不能实现私有属性和私有方法，实例对象之间也不能共享数据，对"类"的模拟不够全面

var Cat = {

name: "大毛",

makeSound: function() {

alert("喵喵喵");

}

};

//兼容IE678

if (!Object.create) {

Object.create = function(o) {

function F() {}

F.prototype = o;

return new F(); //IE678还是用构造函数的方法创建对象

};

}

var cat1 = Object.create(Cat); //不需要用new

alert(cat1.name); // 大毛

cat1.makeSound(); // 喵喵喵

1. Json法

荷兰程序员Gabor de Mooij提出了一种比Object.create()更好的新方法，他称这种方法为"极简主义法"，这种方法不使用this和prototype，代码部署起来非常简单

1. 继承类的属性与方法
2. 实例有自己私有属性和私有方法

写法见**组件开发推荐写法**

* 系统对象

1. 本地对象-----非静态对象 与js的运行环境无关

Object、Function、Array、String、Boolean、Number、Date、RegExp、Error

需要用new方法实例化出来才能用

1. 内置对象-----静态对象 与js的运行环境无关

Global、Math等，不能new出来用，只能用它(类)自身的方法。不需要经过new出来，不需要实例化，直接就能用的对象。

Global是个仅存在概念中的一个对象。有一些全局函数，全局对象，它们的归属就是Global。没法直接使用，仅仅是个概念、虚假、不存在的

1. 宿主对象-----由浏览器提供的对象

DOM、BOM

指js的运行环境—浏览器

Node.js运动在后台，所以它的环境就是后台服务器环境，这时候它就没有DOM、BOM而有一套新的宿主对象

说明：前两种对象的运行与环境无关，是js本身具备的

1. 组件开发、自定义事件

* 组件开发

组件由参数、方法、事件（自定义事件）三部分组成

1. 用面向对象去写。用的时候new一个对象再调用初始化函数，传实际参数，没实际参数走默认参数
2. 构造函数里放默认参数（json格式）与公用属性
3. 初始化函数里让实际参数覆盖默认参数（extend）
4. 实际功能函数里，先用默认参数。再根据实际参数值的不同，做不同的功能

var d2=new Dialog();//new一个对象

d2.init({//设置实际参数，写一个改一个，没有走默认

width:200,

height:400,

title:'新闻',

});

function Dialog(){

//所有的默认参数与公用属性都放这里

this.initial={

width:300,

height:200,

pos:'center',

drag:false,

title:'',

mark:false,

onOff:true,

drag:false

};

this.viewW=document.documentElement.clientWidth;

}

//初始化函数

Dialog.prototype.init=function(opt){

extend(this.initial,opt); //让实际参数覆盖默认参数

this.create();

};

//实际功能函数，要写默认参数，初始化的时候会被实际参数所覆盖

Dialog.prototype.create=function(){

this.box.innerHTML='<div class="title"><span>'+this.initial.title+'</span><a href="javascript:;" class="close">X</span></div><div class="content"></div>';

//创建div等一些操作，创建完以后，下面的函数给他设置属性

this.setData();

};

Dialog.prototype.setData=function(){

this.box.style.width=this.initial.width+'px';

//根据实际参数不同的值，让对象去具不同的功能

if(this.initial.drag){

this.drag();

}

};

function extend(obj1,obj2){

for(var attr in obj2){

obj1[attr]=obj2[attr];

}

}

* 自定义事件

主要跟函数有关系，让函数能够具备事件的某些特性（绑定事件可以让同一个事件有多个不同功能的处理函数）。多人协作的时候可能需要往同一个函数里写不同功能，如果写多的时候就可能会乱。把不同的功能放在同一个函数名里，调用的时候不会被覆盖。类似于绑定事件的功能，同一个事件可以处理不同的函数，而且不会被覆盖。

自定义事件只能通过主动调用去触发

定义分两步：

1. 绑定函数：在事件绑定函数里用映射的方法把对象，事件，事件处理函数的关系联起来，存到数组里
2. 主动调用函数：循环数组（存的是事件函数），调用每一个函数

用的时候，先调用绑定函数，再调用执行函数

function bindEvent(obj,events,fn){

//obj：楼层 events：书架 fn：一本书

//一楼里前端书架下的DOM编程艺术那本书

//一楼 obj （对象-单个） oDiv

//前端书架 obj.listeners （事件-对象） hide

//DOM编程艺术 fn （事件函数-数组） alert（1）、alert(2)、alert(3)

obj.listeners = obj.listeners || {}; //一上来没有走后面，下次有的话就走前面

obj.listeners[events] = obj.listeners[events] || []; //事件做为数组，可能有很多事件函数

obj.listeners[events].push( fn );

//oDiv.hide[alerr(1),alert(2),alert(3)]

//把自定义事件绑定起来

if(obj.nodeType){ //不加会报错，判断是不是dom节点

if(obj.addEventListener){

obj.addEventListener(events,fn,false);

//oDiv.addEventListener(hide,alert(1),false);

}

else{

obj.attachEvent('on'+events,fn);

//oDiv.attachEvent(onhide,alert(1));

}

}

}

function fireEvent(obj,events){ //主动触发自定义事件

if(obj.listeners && obj.listeners[events]){ //第一次可能为空，会报错

for(var i=0;i<obj.listeners[events].length;i++){

obj.listeners[events][i]();

}

}

}

//用的时候先绑定，再调用函数

bindEvent(oSpan , 'show',function(){

alert(3);

});

bindEvent(oSpan , 'show',function(){

alert(4);

});

fireEvent(oSpan , 'show'); //3 , 4

使用：在默认参数里可以不用定义事件，new对象后用bindEvent给对象添加事件，在功能函数里有需要用到的地方用fireEvent触发。一定要先定义后触发

组件开发推荐写法：

(function(window,undefined){

function Person(cfg){

this.config=cfg||{};

this.initial={

name:'kaivon',

age:20,

sex:'男',

height:180,

color:'黄色'

};

this.init();

}

Person.prototype={

init: function(){

this.extend(this.initial,this.config);

},

showMessage: function(){

alert('名字：'+this.initial.name+'; 年龄：'+this.initial.age+'; 性别：'+this.initial.sex+'; 身高：'+this.initial.height+'; 肤色：'+this.initial.color);

},

extend: function(obj1,obj2){

for(var attr in obj2){

obj1[attr]=obj2[attr];

}

}

};

window.Person=window.Person || Person;

})(window);

var newPer1=new Person({name:'zhang', age:15}).showMessage();

var newPer2=new Person({name:'yang', age:30, sex:'女', height:190, color:'黑色'});

newPer2.showMessage();

1. 正则表达式

正则 : 也叫做规则，让计算机能够读懂人类的规则

正则的性能要比传统字符串的性能要高，正则都是操作字符串的，

写法：

var re = /a/; //简写，不能为空。性能好，推荐

var re = new RegExp('a'); //全称，Reg为正则的简写，Exp为表达式的简写。当正则需要传参的时候用这种写法

* test方法

匹配字符串，如果匹配成功就返回真，如果匹配失败就返回假

var str = 'abcd23524de454f';

var re = /b/; //true

var re = /w/; //false

var re = /bc/; //true

var re = /bd/; //false bd为一个整体，str里找不到

var re = /\D/; //true 是不是非数字

alert( re.test(str) ); //用法

* 转义字符

.(点)：任意字符

\.：真正的点

\s：空格

\S：非空格

\d：数字

\D：非数字

\w：字符(字母、数字、下划线\_ )

\W：非字符

\b：独立的部分(起始、结束、空格)

\B：非独立的部分

\1：重复的第n个子项（\后面是几，就是第几个子项，作用在前面的字符不确定，后面没法写死。应用：找字符串中重复最多的字符和次数）

var str = 'a你c';

var re = /a.c/; //str第二位是任意字符都可以匹配成功

alert(re.test(str)); //true

var str = 'a.c';

var re = /a\.c/; //str第二位是真正的点才能匹配成功

alert(re.test(str)); //true

var str = 'onetwo';

var re = /\bone/; //true o前面什么也没有，是开始的位置，匹配成功

var re = /one\b/; //false e后面还有t，不是开始不是结束也不是空格，匹配不成功

alert( re.test(str) );

var str = 'abca';

var re = /(a)(b)(c)\1/; //\1与(a)相等，str的最后一位也是a，所以匹配成功

alert( re.test(str) ); //true

var str = 'abca';

var re = /(a)(b)(c)\2/; //\2与(b)相等，str的最后一位也是a，所以匹配不成功

alert( re.test(str) ); //false

var re = /\w\w/;

var re = /(\w)\1/; //此时第一项的字符不确定，用\1就会完全匹配

* search方法

匹配字符串 , 如果匹配成功，就返回匹配成功的位置，如果匹配失败就返回-1与indexOf一样

默认区分大小写，如果不想区分大小写在最后加个标识i

var str = 'abcdef';

var re = /b/; //1

var re = /de/; //3

var re = /y/; //-1

var re = /B/; //-1

var re = /B/i; //1 最后的i表示不区分大小写

var re = new RegExp('B','i'); //全称写法i的位置

alert( str.search(re) ); //用法

* match方法

匹配字符串，如果匹配成功，就返回匹配成功的数组，如果匹配不成功，就返回null

默认匹配成功一次就会结束，不会继续匹配，如果想全部查找，就要加标识 g(全局匹配)

量词：匹配不确定的位置 +号表示至少出现一次>=1

var str = 'haj123sdk54hask33dkhalsd879';

var re = /\d/; //1 找到1后就不找了

var re = /\d/g; //1,2,3,5,4,3,3,8,7,9 找到所有数字

var re = /\d\d/g; //12,54,33,87 写一个d只找一个数字，写两个找一对数字

var re = /\d+/g; //123,54,33,879 +号表示只要有连着的数字都会找到

alert( str.match(re) ); //用法

* replace方法

匹配字符串，匹配成功的字符去替换成新的字符串

写法：字符串.replace(正则,新的字符串)

var str = 'aaa';

var re = /a/; //baa

var re = /a/g; //bbb

var re = /a+/; //b

alert(str.replace(re,'b')); //用法

|一个竖杠表示或的意思。第二个参数可以是字符串，也可以是一个回调函数

var re = /菲称|中国船|监视之下/g;

aT[1].value = aT[0].value.replace(re,function(str){

//函数的第一个参数：就是匹配成功的字符

var result = '';

for(var i=0;i<str.length;i++){

result += '\*';

}

return result;

});

* 匹配子项

**分组：**没有括号，量词只针对前面一个，有括号就是分组了，量词针对括号里的整体

var str = '2013-6-7';

var re = /\d+-/g; //2013-和6-

var re = /\d-+/g; //3-和6-

var re = /(\d-)+/g; //3-6- 加括号表示把数字与横杠做为整体

str.replace(re,function($0){

alert( $0 )

});

**子项：**正则里的每小括号叫做正则的子项

把正则的整体叫做（母亲）

第一个小括号里面的正则，叫第一个子项(母亲的第一个孩子)

第二个小括号里面的正则，叫第二个子项(母亲的第二个孩子)以此类推

var str = '2013-6-7';

var re = /(\d+)(-)/g;

str = str.replace(re,function($0,$1,$2){

//第一个参数：$0（整体-母亲） 2013-和6-

//第二个参数：$1(第一个子项-第一个孩子) 2013和6

//第二个参数：$1(第二个子项-第二个孩子) -和-

//return $1 + '.'; //2013.6.7

return $0.substring(0,$0.length-1) + '.'; //2013.6.7

});

alert( str ); //2013.6.7

**Math匹配子项：**只有不加g的时候才可以获取到子项的集合

var str = 'abc';

var re = /(a)(b)(c)/;

alert( str.match(re) ); //abc,a,b,c

* 字符类

[] 里面的整体代表一个字符

var str = 'abc';

var re = /a[bde]c/; //re只代表3位，中括号里代表一个字符，表示str的第二个字符是b、d、e都可以匹配成功

alert( re.test(str) ); //true

var str = 'abdc';

var re = /a[bde]c/; //re只有三位，str有四位，re里c对应str里的d，所以匹配不成功

alert( re.test(str) ); //false

* 排除

^ 如果^写在[]里面的话，就代表排除的意思，如果不在中括号里就是开始的意思

var str = 'abc';

var str = 'awc'; //true re里的bde不包含str里的w，匹配成功

var re = /a[^bde]c/; //str的第二位里要不包含bde任意一个字符才能匹配成功

alert( re.test(str) ); //false

* 范围

放在中括号里[]，里面放的是一个范围的字符

var str = 'abc';

var re = /a[a-z0-9A-Z]c/; //str的第二位是在a到z的范围内，匹配成功

alert( re.test(str) ); //true

过滤HTML标签

var re = /<[^>]+>/g; //开始一个< 中括号里的意思：如果不是>都可以，从<到>里的内容都匹配

aT[1].value = aT[0].value.replace(re,''); //替换为空

* 量词

放到大括号里{}

{4,7} //出现4到7次

{4,} //至少出现4次

{4} //正好出现4次

+ : {1,} //至少出现一次\d{1,}

? : {0,1} //出现0次或者1次

\* : {0,} //至少出现0次

var str = 'ab';

var re = /ab+/; //true b在str里至少出现1次

var str = 'ac';

var re = /ab+/; //false b没在str里出现

var re = /ab\*/; //true b可以在str里出现0次

alert(re.test(str));

* 起始与结束

^：正则最开始的位置，如果不在中括号里就代表开始的意思

$：正则最后的位置，代表结束的意思

匹配QQ号：

var re = /^[1-9]\d{4,11}$/; //开始为1-9的数字，接着以是4-11位的数字结束

去掉前后空格：

var re = /^\s+|\s+$/g; //开始至少有一个空格 或者 最后至少有一个空格

str.replace(re,'');

在工作中，最好把正则做为一个json对象，把能用到的验证都放到对象里

var re = {

qq : /[1-9][0-9]{4,9}/,

email : /^\w+@[a-z0-9]+(\.[a-z]+){1,3}$/,

number : /\d+/

};

re.email

1. 闭包

* 特点

1. 函数嵌套函数
2. 内部的函数可以引用外部函数的参数和变量
3. 参数或变量不会被垃圾回收机制所收回

在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁

* 垃圾回收机制

变量在函数运行完后，没有在别的地方再引用，就会被回收清除。Js会周期性的遍历所有变量，一旦变量没有被引用就会被清除

function test1(a){

var b=5;

function test2(){

alert(a);

alert(b);

//a和b都能访问到。函数嵌套函数，内部函数可以访问到外部函数的参数与变量

}

test2();

}

test1();

//test1函数执行后因为变量b在test2中有引用，所以函数执行完后是可以访问到的，即不会被js的垃圾回收机制给回收

function test3(){

var b=1;

}

test3();

alert(b); //报错

//test3函数执行后，变量b没有在别的地方被调用，所以就会被js的垃圾回收机制给回收，再别的地方就访问不到

function test4(){

var a=1;

a++;

alert(a);

}

test4(); //2

test4(); //2

//两次都弹出的是2，不会累加，就是因为垃圾回收机制，函数调用完以后a就会被回收，再调用的时候又会重新赋值

* 闭包的用途

1. 需要一个变量长期保持在内存中
2. 避免全局变量的污染
3. 私有成员的存在，可以读取函数内部的变量

//return的外面函数赋值给谁，谁的值就是return的结果

function test5() {

var a = 1;

return function() {

a++;

alert(a);

}

}

//test5执行后会return个函数出去，而b等于它的结果，即b就是函数return后的值

var b = test5();

b(); //2 此时b为一个函数,加括号表示执行后的结果

b(); //3 变量a是函数test5内的局部变量，内部函数可以访问到，并且可以修改他的值。他始终会存在内存当中，不会被垃圾回收机制回收。如果没存在，第二次调用就会访问不到

//把函数体放到括号里就形成了函数表达式，再加一个括号就是调用函数的意思，自执行。函数执行后，会把里面的函数返回给了test变量，也就是说，test就是嵌套函数的结果

var test6 = (function() {

var a = 1;

return function() {

a++;

alert(a);

}

})();

test6(); //2

test6(); //3

* 闭包的用法

1. 模块化代码
2. 在循环中找到元素对应的索引

//私有方法 模块化代码

var test7 = (function() {

var a = 1;

function test8() {

a++;

alert(a);

}

function test9() {

a++;

alert(a);

}

return {

fn1: test8,

fn2: test9

}

})();

test7.fn1(); //2

test7.fn2(); //3

1. 在循环中找到元素对应的索引

循环时函数自执行，i的值会存到内存当中，就是私有成员。点击的时候，是内部函数访问外部函数的变量

window.onload = function() {

var aLi = document.getElementsByTagName('li');

//第一种

for (var i = 0; i < aLi.length; i++) {

(function(i) {

aLi[i].onclick = function() {

alert(i);

}

})(i);

}

//第二种

for (var i = 0; i < aLi.length; i++) {

aLi[i].onclick = (function(i) {

return function() {

alert(i);

}

})(i)

}

};

* 闭包需要注意的地方

在IE获取到的DOM对象操作自身的属性，会形成互相引用就会造成内存泄露，变量不会被垃圾回收机制所回收，得不到释放。只有关闭浏览器的时候才会被释放

先获取一个元素。在这个元素操作函数内，访问这个元素（在函数外面获取的）的属性，发生了内外互相引用，会造成内存泄漏

解决办法：在页面unload的时候把它的事件给取消掉或者把属性在函数外面定义好，在函数内用，用完后清除或者在退出函数之前，将不使用的局部变量全部删除

window.onload = function() {

//第一种方法

var oDiv = document.getElementById('div1');

oDiv.onclick = function() {

alert(oDiv.id);

};

window.unload = function() {

oDiv.onclick = null;

}

//第二种方法，把属性在外面定义好，函数里面用，用完了再删除掉

var oDiv = document.getElementById('div1');

var id = oDiv.id;

oDiv.onclick = function() {

alert(id);

};

oDiv = null;

};

闭包后在父函数外部可以改变内部变量的值。所以，如果你把父函数当作对象（object）使用，把闭包当作它的公用方法（Public Method），把内部变量当作它的私有属性（private value），这时一定要小心，不要随便改变父函数内部变量的值。

注意：return的时候第一个括号一定要跟着它在一行，不然会报错

return

{

} //这样写会报错

* 垃圾回收机制

当创建一个对象时，JavaScript 会自动为该对象分配适当的内存。从这一刻起，垃圾回收器就会不断对该对象进行评估，以查看它是否仍是有效的对象。

垃圾回收器定期扫描对象，并计算引其它对象引用了这个对象的的数量。如果引用数量为 0（没有其他对象引用过该对象），或对该对象的惟一引用是循环的，那么该对象的内存就会被回收

* 内存泄漏

任何对象在您不再拥有或不需要它之后仍然存在内存当中，Firefox 和旧版的 Internet Explorer 都存在过内存泄漏，而且内存泄露一直持续到浏览器关闭。

造成内在泄漏的方式：

1. 闭包
2. console.log
3. 循环引用：某个DOM绑定了一个事件，然后这个事件里也引用了这个DOM元素，这样的话就会造成一个循环引用
4. setTimeout 的第一个参数使用字符串而非函数的话

解决办法：

用事件委托

* (function(window,undefined){})(window);

如果把window对象传进来就表示：

1、把这个类直接添加到window身上；window属于js最顶端的对象，根据作用域链的作用，如果一层一层的查的话，查到window的时候会比较慢。如果把window当作参数传进来，就表示，变量直接找的是参数的window，不会找外面的window，查找速度快

1. 有利于压缩，window。如果不传参，函数里的window是没法压缩的，外面没有参数告诉它window是谁，压缩了也不知道是谁。如果把形参换成一个字母，在函数内部会省下很多空间

传undefined的意思：

1. undefined是window的一个属性，不是保留字也不是关键字。在外面是可以被修改的（IE9以下）。所以为了避免被修改就以参数的形式传进去，不受外面修改的影响。
2. 有时候判断的时候会与undefined做对比，undefined是window对象的属性，做比较的时候会去找window的属性，而window的属性非常的多，所以会有性能的问题。而把undefined做为参数传进来，如果做比较的话就不会去找window下的属性了

var undefined=10;

alert(undefined); //在IE678下会被修改，弹出10。其它浏览器为undefined

1. JS性能优化、DOM性能优化

* 概念

JavaScript是由ECMAScript，DOM和BOM三者组成的，

ECMAScript主要是js的语法，控制语句。

DOM就是我们平时操作HTML标签的一些方法包括getElementById,childNodes,appendChild,innerHTML等方法。

BOM是跟浏览器相关的一些操作

DOM与ECMAScript是被浏览器分成了两个不同的文件。它们的关系就像两个独立的小岛，js操作DOM需要从一个岛到另一个岛，DOM的性能就是岛与岛之间的桥梁，每操作一次就需要过一次桥，过桥的次数多了就会造成性能下降。想要优化DOM的操作就只能减少过桥的次数，即减少ECMAScript与DOM之间的操作

* 网页的生成过程

1. HTML代码转化成DOM
2. CSS代码转化成CSSOM（CSS Object Model）
3. 结合DOM和CSSOM，生成一棵渲染树（包含每个节点的视觉信息）
4. 生成布局（layout），即将所有渲染树的所有节点进行平面合成
5. 将布局绘制（paint）在屏幕上

这五步里面，第一步到第三步都非常快，耗时的是第四步和第五步。生成布局"（flow）和"绘制"（paint）这两步，合称为"渲染"（render）

* 重排和重绘

1. 重排：改变元素的形状、宽高位置等属性。重排结束后要把它显示出来就是重绘
2. 重绘：改变元素的颜色等属性。浏览器显示内容就是重绘的过程

"重绘"不一定需要"重排"，比如改变某个网页元素的颜色，就只会触发"重绘"，不会触发"重排"，因为布局没有改变。但是，"重排"必然导致"重绘"，比如改变一个网页元素的位置，就会同时触发"重排"和"重绘"，因为布局改变了，浏览器需要重新显示

重排和重绘会非常耗费资源，是导致网页性能低下的根本。提高网页性能，就是要降低"重排"和"重绘"的频率和，尽量少触发重新渲染

多数显示器默认频率是60Hz，即1秒刷新60次，所以理论上最小间隔为1/60\*1000ms ＝ 16.7ms

**渲染**

网页生成的时候，至少会渲染一次。用户访问的过程中，还会不断重新渲染。以下三种情况，会导致网页重新渲染

1. 修改DOM
2. 修改样式表
3. 用户事件（比如鼠标悬停、页面滚动、输入框键入文字、改变窗口大小等等）

现在的浏览器已经很智能了，会尽量把所有的变动集中在一起，排成一个队列，然后一次性执行，尽量避免多次重新渲染

ie中认为没有hasLayout属性的dom节点的js操作不会触发render， 再比如有时dom没有display=block的属性则改变其属性不会触发rander。

样式的写操作之后，如果有下面这些属性的读操作，都会引发浏览器立即重新渲染

offsetTop/offsetLeft/offsetWidth/offsetHeight

scrollTop/scrollLeft/scrollWidth/scrollHeight

clientTop/clientLeft/clientWidth/clientHeight

getComputedStyle()

* **优化技巧**
* 分离读写操作

写的操作会引发浏览器重新渲染，应把读的操作与写的操作分开来写。不要两个读操作之间，加入一个写操作

div.style.color = 'blue'; //写操作，重绘第一次

var margin = parseInt(div.style.marginTop); //读操作，把两次写的操作分开，就会分别进行重绘

div.style.marginTop = (margin + 10) + 'px'; //写操作，重绘第二次

推荐写法：

var left = div.offsetLeft; //读操作

var top = div.offsetTop; //读操作

div.style.left = left + 10 + "px"; //写操作

div.style.top = top + 10 + "px"; //写操作

规则：

样式表越简单，重排和重绘就越快

重排和重绘的DOM元素层级越高，成本就越高

table元素的重排和重绘成本，要高于div元素

* 分离ECMAScript与DOM之间的操作

window.onload = function() {

var oDiv = document.getElementById('div');

//循环语句里操作DOM，循环一次过一次桥。chrom下 470ms左右

console.time('kaivon');

for (var i = 0; i < 5000; i++) {

oDiv.innerHTML += 'a';

}

console.timeEnd('kaivon');

//循环语句里操作变量，没有过桥。把变量赋给DOM，只过一次桥。chrom下 1 ms左右

var str = '';

console.time('kaivon');

for (var i = 0; i < 5000; i++) {

str += 'a';

}

oDiv.innerHTML = str;

console.timeEnd('kaivon');

};

* cssText

为元素添加多个样式的时候，最高性能的就是加个class，其次是cssText，不管有多少样式操作，浏览器只会渲染一次

var left = 10;

var top = 10;

//性能差

obj.style.left = left + "px";

obj.style.top = top + "px";

//性能好

obj.style.cssText += "; left: " + left + "px; top: " + top + "px;";

cssText所有浏览器都支持，但是会有一些问题

1. 覆盖原来的样式，原因是替换当前对象为新对象了

<div style="color:red;"></div>

div.style.cssText = "width:200px;"; //这样写的话你会发现div的颜色值没有了，只剩下宽度样式

1. IE678需要单独加个分号

解决办法：

function setStyle(el, strCss) {

function endsWith(str, suffix) {

var l = str.length - suffix.length;

return l >= 0 && str.indexOf(suffix, l) == l;

}

var sty = el.style,

var cssText = sty.cssText;

if (!endsWith(cssText, ';')) {

cssText += ';';

}

sty.cssText = cssText + strCss;

}

* 缓存布局信息和length属性

如果某个样式是通过重排得到的，那么最好缓存结果。避免下一次用到的时候，浏览器又要重排。把length存到变量里，循环的时候用变量

window.onload = function() {

var oDiv = document.getElementById('div');

//低性能，每运动一次要取一下left与top的值

setInterval(function() {

oDiv.style.left = oDiv.offsetLeft + 1 + 'px';

oDiv.style.top = oDiv.offsetTop + 1 + 'px';

}, 30);

//高性能，只用取一次值(把它们先缓存下来)，定时器里不用反复去取

var L = oDiv.offsetLeft;

var T = oDiv.offsetTop;

setInterval(function() {

L++;

T++;

oDiv.style.left = L + 'px';

oDiv.style.top = T + 'px';

}, 30);

//把length的值存在一个变量里，不用每次循环都要去取它的值

var len = aLi.length;

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

aLi.innerHTML = 'li';

}

};

* 所有操作放在appendChild之前

window.onload = function() {

var oUl = document.getElementById('ul');

console.time('kaivon');

for (var i = 0; i < 5000; i++) {

var aLi = document.createElement('li');

//appendChild后再操作aLi性能会降，chrom下无多大影响，ff下591ms

//appendChild前操作aLi性能会提升，ff下390ms

aLi.innerHTML = 'li';

oUl.appendChild(aLi);

}

console.timeEnd('kaivon');

}

* 用innerHTML而不用createElement

Chrom、Safari：DOM方法创建要比innerHTML方法创建效率高

FF、IE：innerHTML方法创建要比DOM方法创建效率高

因为webkit内核的浏览器本身性能就很高，所以可以忽略创建方法，而FF与IE相对来说性能会低点，所以首先要考虑它们，最好用innerHTML的方法来创建元素

window.onload = function() {

var oUl = document.getElementById('ul');

var str = '';

//DOM方法创建的chrom下11ms,FF下289ms

console.time('kaivon');

for (var i = 0; i < 5000; i++) {

var oLi = document.createElement('li');

oUl.appendChild(oLi);

}

console.timeEnd('kaivon');

//innerHTML方法创建的chrom下30ms,FF下224ms

console.time('kaivon');

for (var i = 0; i < 5000; i++) {

str += '<li></li>';

}

oUl.innerHTML += oLi;

console.timeEnd('kaivon');

};

* 节点克隆

复制新节点会比创建新节点要效率高

window.onload = function() {

var oUl = document.getElementById('ul');

//没用克隆方法创建 chrom下36ms,FF下324ms

console.time('kaivon');

for (var i = 0; i < 5000; i++) {

var oLi = document.createElement('li');

oLi.innerHTML = 'li';

oUl.appendChild(oLi);

}

console.timeEnd('kaivon');

//克隆方法创建 chrom下8ms,FF下232ms。复制新节点会比创建新节点要效率高

console.time('kaivon');

var oLi = document.createElement('li');

for (var i = 0; i < 5000; i++) {

var newLi = oLi.cloneNode(true);

oUl.appendChild(oLi);

}

console.timeEnd('kaivon');

};

* webWorker

如果网页动画能够做到每秒60帧，就会跟显示器同步刷新，达到最佳的视觉效果。这意味着，一秒之内进行60次重新渲染，每次重新渲染的时间不能超过16.66毫秒。

如果想达到60帧的刷新率，就意味着JavaScript线程每个任务的耗时，必须少于16毫秒。一个解决办法是使用Web Worker，主线程只用于UI渲染，然后跟UI渲染不相干的任务，都放在Worker线程。

* 尽量只获取元素的节点方法

用childNodes获取到了节点，但它包括了元素节点，文本节点，注释节点等。有些是不需要的，就没必要获取

children只会获取元素的结点

FistChild用fistElementChild替代

* 选择器API

利用querySelector、querySelectorAll

IE8以下不支持

var oUl=document.getElementById('ul');

var aLi=oUl.getElementsByTagName('li');

var aLi=document.querySelectorAll('#ul li');

* window.requestAnimationFrame()

window.requestIdleCallback() timeRemaining() 方法 didTimeout属性

新增的方法，跟运动有关，不过各浏览器支持不乐观，Chrom31+ FF26+ IE10+

<http://www.cnblogs.com/Wayou/p/requestAnimationFrame.html>

1. 事件委托、事件代理

利用冒泡的原理把事件加到父级上，触发执行效率。实例：新浪微博新发的微博也会有事件

* 优点

1. 可以提高性能，减少循环

window.onload = function() {

var oUl = document.getElementById('ul');

var aLi = oUl.getElementsByTagName('li');

oUl.onmouseover = function(ev) {

var oEvent = ev || event;

var target = oEvent.target || oEvent.srcElement; //target为每个li

if (target.tagName.toLowerCase() == 'li') {

target.style.background = 'red';

}

};

};

1. 新添加的元素也有原有的事件

如果用常规的循环的办法给ul动态的添加li，添加后的li是没有之前的事件的，是因为新添加的时候，循环已经结束，前面循环的Length取的是添加之前的length。

window.onload = function() {

var oUl = document.getElementById('ul');

var aLi = oUl.getElementsByTagName('li');

var oInput = document.getElementById('input');

var noIndex = 4;

//动态添加的也会有之前的事件，用常规的循环li的方法加的事件，动态添加上去后是没有事件的。因为它的事件是加到父级上的。这里我没有写

oInput.onclick = function() { //添加新元素

noIndex++;

var oLi = document.createElement('li');

oLi.innerHTML = 1111 \* noIndex;

oUl.appendChild(oLi);

}

oUl.onmouseover = function(ev) {

var oEvent = ev || event;

var target = oEvent.target || oEvent.srcElement;

if (target.tagName.toLowerCase() == 'ul') return;

target.style.background = 'red';

};

oUl.onmouseout = function(ev) {

var oEvent = ev || event;

var target = oEvent.target || oEvent.srcElement;

if (target.tagName.toLowerCase() == 'ul') return;

target.style.background = '';

};

};

* 事件源

不管在哪个事件中，操作的元素就是事件源对象，如果有多层嵌套，它会找到最里层

IE：window.event.srcElement

DOM标准：event.target

事件源对象的标签名：tagName或者nodeName

1. 枚举算法
2. 鼠标选中文字

|  |
| --- |
| window.onload=function() |
| { |
| *//选取文字的方法* |
| function selectText() |
| { |
| if(document.selection) |
| { |
| *//ie* |
| return document.selection.createRange().text; |
| } |
| else |
| { |
| *//dom标准* |
| return window.getSelection().toString(); |
| } |
| }; |
|  |
| var oP=document.getElementById('p1'); |
| var oDiv=document.getElementById('ico'); |
|  |
| oP.onmouseup=function(ev) |
| { |
| var oEvent=ev||event; |
| var l=oEvent.clientX; |
| var t=oEvent.clientY; |
| if(selectText().length>10) |
| { |
| setTimeout(function(){ |
| oDiv.style.display='block'; |
| oDiv.style.left=l+'px'; |
| oDiv.style.top=t+'px'; |
| },100); |
| } |
| else |
| { |
| oDiv.style.display='none'; |
| } |
| } |
| oP.onclick=function(ev) |
| { |
| var oEvent=ev||event; |
| oEvent.cancelBubble=true; |
| }; |
| document.onclick=function() |
| { |
| oDiv.style.display='none'; |
| }; |
| }; |

1. js操作iframe

* 获取iframe的window对象：contentWindow

oFrame.contentWindow //iframe的window对象 所有浏览都支持

oFrame.contentDocument //document对象 ie67不支持

oFrame.contentWindow.document.getElementById('div1').style.color='#f00';

* iframe获取父级的window对象：window.parent

window.parent //父级的window对象，所有浏览器都支持。操作父级必需放到服务器下

window.parent.document.getElementById('div1').style.color='#0f0';

如果有嵌套多层iframe，获取最顶层的window对象

window.top //最顶层window对象

注意：操作父级需要放到服务器环境下进行，要不会报错

* Iframe事件

oIframe.onload=function () {

alert(123);

};

//IE下的onload事件必需要绑定才可以,我的浏览器为什么不用绑定也可以，难道是IE6

oIframe.attachEvent('onload',function(){

alert(123);

});

* iframe防钓鱼处理

有的网站做了iframe处理（防钓鱼），你用iframe引用的时候，他会把你的页面跳转到它的页面。做法就是，判断我是不是最顶层(如果最顶层就没有嵌套)，如果不是的话，就让最顶层的location变成我的地址

if(window!=window.top){

window.top.location.href=window.location.href;

}

1. 图片加载

图片最好标明高度和宽度，如果浏览器没有找到这两个参数，它需要一边下载图片一边计算大小，如果图片很多，浏览器需要不断地调整页面。这不但影响速度，也影响浏览体验。

* 预加载

function loadImg(url,fn){

var oImg=new Image();

oImg.onload=function(){

oImg.onload=null;

fn.call(oImg);

}

oImg.src=url;

}

问题（瀑布流）：如果图片多的话，图片加载完取不到对应的索引值，想用索引值的话得在循环里用闭包的方法取i的值

img.complete //有兼容问题

ie、opera下，当图片加载过一次以后，图片就会缓存，下次就不会触发complete，而是直接从缓存中加载过来。其它浏览器没有问题，会触发complete事件

img.onload

任何浏览器都会触发。但是src的赋值与 onload事件的绑定，浏览器不同，顺序不同

在ie和opera下，先赋值src，再onload。不过图片会缓存，会错过了onload事件的触发，所以应该先绑定onload事件，然后再给src赋值

* 按需加载

1. 设置图片的地址为一张loading图片，或者一个空白图片，避免IE下出现个红叉
2. 把图片的地址放到自定义属性里，等图片进入可视区的时候，把自定义属性的值赋给src

showImg();

window.onscroll=showImg;

function showImg () {

var scrollTop=document.documentElement.scrollTop||document.body.scrollTop;

for (var i = 0; i < aImg.length; i++) {

if(aImg[i].getAttribute('src')!='')continue; //如果地址已经有了就跳出循环

var bDisplay=aImg[i].offsetTop+oList.offsetTop<document.documentElement.clientHeight+scrollTop;

if(bDisplay){

aImg[i].src=aImg[i].getAttribute('\_src');

}

}

}

1. Json对象（单体）

优点：简单

缺点：不适合多对象

适用于：整个程序里只有一个，写起来比较简单

Json里的数据最好全用引号引起来，会比较安全，不会出问题

Json转换对象与字符串的方法（IE7及以下不支持）可以通过引入json.js文件到页面中去解决

var s = JSON.stringify( obj ); //把对象转换成字符串

var o = JSON.parse( s ); //把字符串转换成对象

* Json与数组的区别

var json={a:12, b:5, c:7}

var arr=[12,5,7]

区别：

1. json的下标是字符串，数组的下标是数字。Json[‘a’] arr[0]
2. json没有length属性，数组有length，不能用for循环
3. 循环：json只能用for in，而数组即可以用for循环也可以用for in
4. 命名空间

可以让多个相同名字的函数同时存在，就是以json对象的形式去写代码

JavaScript 的命名空间并不是真正的命名空间, 只是在脚本内部创建一个封闭的小空间, 必须通过特定的空间名称才能对空间内部的代码进行访问。这样可以防止同名函数和变量发生冲突, 也可以更方便地管理代码，使用之后就能防止和某些插件的代码发生冲突。

var zns = {};

zns.common = {};

zns.fx = {};

zns.site = {};

zns.common.getUser = function() {

alert('a');

};

zns.fx.getUser = function() {

alert('b');

};

zns.site.getUser = function() {

alert('c');

};

zns.common.getUser();

zns.fx.getUser();

zns.site.getUser();

1. Script标签事件

用Script请求数据，有时候会出现问题，发现数据为空。是因为动态的创建script标签后，没有等它onload后就执行函数，会找不到数据。每个标签都有onload属性，操作它们需要等他们加载完成后再操作

function sendApi(url, callBack) {

var oScript = document.createElement('script');

oScript.type = 'text/javascript';

oScript.src = url;

var scriptLoaded = 0;

// IE和opera支持onreadystatechange

// safari、chrome、opera、ff支持onload

oScript.onload = oScript.onreadystatechange = function() {

// 避免opera下的多次调用

if (scriptLoaded) return;

if (!oScript.readyState || oScript.readyState == "loaded" || oScript.readyState == "complete") {

scriptLoaded = 1;

callBack();

}

}

document.getElementsByTagName('head')[0].appendChild(oScript);

}

sendApi('http://vact.zhidao.baidu.com/api.php?act=system\_now', function() {

alert(1);

});

1. 无缝连续滚动（只适用于办显示一张图片）

原理：利用定位，当运动到头后，让第一个li定位到最后面。然后运动后再让第一个Li回去

1. 自动轮播需要两个索引，一个用来给点，一个给ul，ul的索引要比点的大1
2. 每个li要用相对定位
3. Li运动到最后一个，运动停止后，让第一个Li的left值到最后
4. Ul多运动一格
5. Ul多运动一格停止后，让第一个Li的left值回到0，再让Ul的left回到0

|  |
| --- |
| var oDiv=document.getElementById('div1'); |
| var oUl=oDiv.children[0]; |
| var aLi=oUl.children; |
| var aImg=oUl.getElementsByTagName('img'); |
|  |
| var oBtn=oDiv.children[1]; |
| var aBtn=oBtn.children; |
|  |
| var iNow=0; |
| var iNow2=0; |
| var t=null; |
|  |
| var newImg=new Image(); |
| newImg.onload=function() |
| { |
| oUl.style.width=aImg[0].offsetWidth\*aImg.length+'px'; |
|  |
| *//图片居中* |
| function toResize() |
| { |
| var viewWidth=document.documentElement.clientWidth; |
| *//大于1000的时候一直居中显示，小于1000的时候出横向滚动条* |
| if(viewWidth>1000) |
| { |
| for(var i=0;i<aImg.length;i++) |
| { |
| aImg[i].style.left=-(aImg[0].offsetWidth-viewWidth)/2+'px'; |
| } |
| } |
| } |
| toResize(); |
| window.onresize=toResize; |
|  |
| *//轮播* |
| function toRun() |
| { |
| *//滚动到最后一个了* |
| if(iNow==aLi.length-1) |
| { |
| iNow=0; |
| *//让第一个li的位置移到最后* |
| aLi[0].style.left=aImg[0].offsetWidth\*aImg.length+'px'; |
| } |
| else |
| { |
| iNow++; |
| } |
|  |
| *//Ul的滚动系数，要比iNow多运动一个，为了能让加到最后的Li运动过来，形成连贯的动作* |
| iNow2==aLi.length?iNow2=0:iNow2++; |
|  |
| for(var i=0;i<aBtn.length;i++) |
| { |
| aBtn[i].className=''; |
| } |
| aBtn[iNow].className='active'; |
|  |
| *//ul的运动系数用iNow2，判断条件要在运动结束后* |
| startMove(-iNow2\*aImg[0].offsetWidth,function(){ |
| *//第一个Li已经接上尾了，运动过来了* |
| if(iNow==0) |
| { |
| *//让第一个Li回来原来的位置，同时让ul回到0的位置，系数也变为0* |
| aLi[0].style.left=0; |
| oUl.style.left=0; |
| iNow2=0; |
| } |
| }); |
| }; |
|  |
| t=setInterval(toRun,3000); |
| oUl.onmouseover=function() |
| { |
| clearInterval(t); |
| t=null; |
| }; |
| oUl.onmouseout=function() |
| { |
| t=setInterval(toRun,3000); |
| }; |
| }; |
| newImg.src=aImg[0].src; |

1. 任意张图片无缝滚动

原理：

1. 先复制0到n个节点，添加到最后面
2. 滚动，滚动后删除刚才复制的节点
3. 删除之前的节点后，让ul回到0点
4. 往前面添加的时候需要复制最后的节点
5. 添加完节点后一定要重置Ul的宽度，不然后会折行（出过问题）
6. 往前面添加节点，一定要改变ul的位置

|  |
| --- |
| var oLeftBtn=document.getElementById('left\_btn'); |
| var oRightBtn=document.getElementById('right\_btn'); |
| var oDiv=document.getElementById('div1'); |
| var oUl=oDiv.children[0]; |
| var aLi=oUl.children; |
| var moveNum=2; |
| var bBtn=true; |
|  |
| function setWidth() |
| { |
| oUl.style.width=(aLi[0].offsetWidth+10)\*aLi.length+'px'; |
| } |
| setWidth(); |
| *//往左走* |
| oLeftBtn.onclick=function() |
| { |
| *//上来设为true，可以点一次，点后设为false，等运动完后再设为true* |
| if(bBtn) |
| { |
| bBtn=false; |
| *//复制0到moveNum个的节点，添加到最后面* |
| for(var i=0;i<moveNum;i++) |
| { |
| var newNode=aLi[i].cloneNode(true); |
| oUl.appendChild(newNode); |
| } |
| setWidth(); |
| *//添加完运动* |
| startMove(-moveNum\*(aLi[0].offsetWidth+10),function(){ |
| *//****运动完后移除添加的moveNum个节点，注意移除一个后，第二个会顶上，所以只用永远第0个节点即可，不用移除第i个节点*** |
| for(var i=0;i〈moveNum;i++) |
| { |
| oUl.removeChild(aLi[0]); |
| } |
| *//运动停止后，ul的left值已经为运动后的值了，而移除完节点后，里面的Li都会从0开始排序。所以要在运动完后让ul瞬间到0的位置* |
| oUl.style.left=0; |
|  |
| bBtn=true; |
| }); |
| } |
| }; |
| *//往右走* |
| oRightBtn.onclick=function() |
| { |
| if(bBtn) |
| { |
| bBtn=false; |
| *//复制aLi.length到aLi.length-moveNum个的节点，添加到最前面* |
| for(var i=0;i<moveNum;i++) |
| { |
| *//aLi.length-1-i，是从后向前复制* |
| var newNode=aLi[aLi.length-1-i].cloneNode(true); |
| *//添加到是前面* |
| oUl.insertBefore(newNode,aLi[0]); |
| } |
| *//添加完li后一定要重置ul的宽度，不然，新加的Li会因为宽度不够而折行* |
| setWidth(); |
|  |
| *//改变当前的ul的left值，才能运动到0的位置* |
| oUl.style.left=-moveNum\*(aLi[0].offsetWidth+10)+'px'; |
|  |
| startMove(0,function(){ |
| *//运动完后移除添加到最后面的moveNum个节点，* |
| for(var i=0;i<moveNum;i++) |
| { |
| oUl.removeChild(aLi[aLi.length-1]); |
| } |
| *//运动停止后，ul的left值已经为运动后的值了，而移除完节点后，里面的Li都会从0开始排序。所以要在运动完后让ul瞬间到0的位置* |
| oUl.style.left=0; |
|  |
| bBtn=true; |
| }); |
| } |
| }; |

1. 微博登录邮箱提示

要点：

1. input的事件，IE：onpropertychange，FF/Chrom：oninput。此事件是input一发生改变就发生，不管输入还是复制进去的
2. 拆分字符串，以@符号为界，分成两部分再去检测
3. 事件冒泡，点击输入框以外UL需要消失，要阻止事件冒泡

|  |
| --- |
| var mail=['sina.com','163.com','qq.com','126.com','vip.sina.com','sina.cn','hotmail.com','gmail.com','sohu.com','yahoo.cn','139.com','wo.com.cn','189.cn']; |
| var oLogin=document.getElementById('login'); |
| var oInput=oLogin.children[0]; |
| var oUl=oLogin.children[1]; |
|  |
| for(var i=0;i<mail.length;i++) |
| { |
| var oLi=document.createElement('li'); |
| oLi.innerHTML='@'+mail[i]; |
| oUl.appendChild(oLi); |
| } |
| var aLi=oUl.children; |
| oUl.innerHTML='<li>请选择邮箱类型</li><li></li>'+oUl.innerHTML; |
| oUl.style.display='none'; |
|  |
| oInput.onkeydown=function() |
| { |
| oUl.style.display='block'; |
| }; |
| oInput.onclick=function(ev) |
| { |
| var oEvent=ev||event; |
| oEvent.cancelBubble=true; |
| } |
| document.onclick=function(ev) |
| { |
| var oEvent=ev||event; |
| oUl.style.display='none'; |
| }; |
|  |
| function input(obj,fn) |
| { |
| *//检测IE浏览器* |
| var ie=!-[1,]; |
| *//表单事件，ff/chrom为oninput，IE下为onpropertychange* |
| ie?obj.onpropertychange=fn:obj.oninput=fn; |
| } |
| input(oInput,change); |
|  |
| function change() |
| { |
| aLi[1].innerHTML=oInput.value; |
|  |
| var intvalue=oInput.value; |
| *//把邮箱以@符号为界，分隔为两部分* |
| var userName=intvalue.split('@')[0]; |
| var realmName=intvalue.split('@')[1]; |
|  |
| *//用户还没输入@后面的时候，realmName为undefined* |
| if(typeof realmName=='undefined')realmName=''; |
|  |
| for(var i=2;i<aLi.length;i++) |
| { |
| *//ul里前面加了两个Li,所以要从2开始* |
| aLi[i].innerHTML=userName+'@'+mail[i-2]; |
|  |
| *//前面的条件是用户正在输入的域名在li里能找到，后面条件是用户已经输入完域名了* |
| if(aLi[i].innerHTML.indexOf(realmName)==-1 || realmName==mail[i-2]) |
| { |
| *//没找到全不显示* |
| aLi[i].style.display='none'; |
| } |
| else |
| { |
| *//找到了，只显示找到的* |
| aLi[i].style.display='block'; |
| } |
|  |
| aLi[i].onmouseover=function() |
| { |
| this.className='active'; |
| }; |
| aLi[i].onclick=function() |
| { |
| oInput.value=this.innerHTML; |
| oUl.style.display='none'; |
| }; |
| aLi[i].onmouseout=function() |
| { |
| this.className=''; |
| }; |
| } |
| } |
|  |

1. 瀑布流

要点&问题：

1. 一定要在图片Onload后再去算高度，图片是先执行完所有src，然后再执行onload里的内容
2. 元素在appendChild之前可以做任何操作，就像已经存在一样，任何操作都要在appendChild之前做，可以提高性能
3. 加载Li之前取的所有li，在加载之后操作它的length会发现，它的Length是加载后的Length。
4. 定位，利用上一条Li的length永远是所有的Length这一点，去根据最后一排的Li来定位会很方便
5. 循环的时候，一旦有条件满足了，就break一下，可以提高性能

|  |
| --- |
| window.onload=function() |
| { |
| var oWrap=document.getElementById('wrap'); |
| var oUl=oWrap.children[0]; |
| var aLi=oUl.children; |
| var padding=10; *//间距* |
| var addNum=4; *//加载数据的条数* |
|  |
| *//列数* |
| var c=Math.floor(oWrap.offsetWidth/aLi[0].offsetWidth); |
|  |
| for(var i=0;i<aLi.length;i++) |
| { |
| aLi[i].style.left=i%c\*(aLi[0].offsetWidth+padding)+'px'; |
| if(i>=c) |
| { |
| *//从第二排开始定位高度值* |
| aLi[i].style.top=aLi[i-c].offsetTop+aLi[i-c].offsetHeight+padding+'px'; |
| } |
| } |
|  |
| window.onscroll=function() |
| { |
| var scrollTop=document.documentElement.scrollTop||document.body.scrollTop; |
| var viewHeight=document.documentElement.clientHeight; |
|  |
| *//循环最后四个* |
| for(var i=aLi.length-c;i<aLi.length;i++) |
| { |
| *//每个Li离最上面的高度* |
| var liTop=aLi[i].offsetHeight+aLi[i].offsetTop; |
|  |
| *//首先进行可视区的Li(滚动条已经滚过了的Li)* |
| if(liTop<scrollTop+viewHeight) |
| { |
| *//防止超过总数据，这里给了个定值40条* |
| if(addNum+aLi.length>40) |
| { |
| addNum=40-aLi.length; |
| } |
|  |
| *//加载数据。这里写的时候出现个问题，循环的图片地址一定要跟onload里的地址一样，不然就会出错* |
| var newImg=new Image(); |
| newImg.onload=function() |
| { |
| for(var j=0;j<addNum;j++) |
| { |
| var oLi=document.createElement('li'); |
| oLi.innerHTML='<a href="#"><img src="img/'+(aLi.length+1)+'.jpg" />images\_'+(aLi.length+1)+''; |
|  |
| *//新加载的以它上面那个的位置来定位* |
| oLi.style.left=aLi[aLi.length-c].offsetLeft+'px'; |
| oLi.style.top=aLi[aLi.length-c].offsetTop+aLi[aLi.length-c].offsetHeight+padding+'px'; |
| oLi.style.opacity=0; |
| oLi.style.filter='alpha(opacity:0)'; |
|  |
| startMove(oLi,{opacity: 100}); |
| oUl.appendChild(oLi); |
| } |
| } |
|  |
| *//从1开始，为了避免重复加载 加载之前的最后一张* |
| for(var k=1;k<addNum;k++) |
| { |
| newImg.src='img/'+(aLi.length+k)+'.jpg' |
| } |
|  |
| *//为了性能，找到以后就不要循环了* |
| break; |
| } |
| } |
| }; |
| } |
|  |

1. 微博发布框

要点：

1. 汉字与字节。一个汉字gbk下为两个字节，utf-8下为三个字节，一个字母一个字节
2. 把汉字分为两个字节，两个字节算一个字的话，取的的字节数要除以2
3. 输入框的兼容事件，ie：onpropertychange，标准：oninput
4. 闪的背景，用定时器不断改变背景色

|  |
| --- |
| var oPublic=document.getElementById('public'); |
| var oP=oPublic.children[0]; |
| var oTextarea=oPublic.children[1]; |
| var oA=oPublic.children[2]; |
| var ie=!-[1,]; *//判断ie的条件* |
| var t=null; |
| var iNum=0; |
|  |
| oTextarea.onfocus=function() |
| { |
| *//获取焦点* |
|  |
| *//框里有字的话，重新获取焦点要显示有字的状态* |
| if(oTextarea.value) |
| { |
| change(); |
| } |
| else |
| { |
| oP.innerHTML='打击虚假消息，建设文明微博，还可以输入140字'; |
| oP.className='textColor'; |
| } |
| } |
| oTextarea.onblur=function() |
| { |
| *//失去焦点* |
| *//框里有字的情况下失去焦点后要显示有字的状态* |
| if(!oTextarea.value) |
| { |
| oP.innerHTML='张国荣十年祭，陈淑芬曝哥哥最后留言　热门微博'; |
| oP.className=''; |
| } |
| } |
|  |
| if(ie) |
| { |
| oTextarea.onpropertychange=change; |
| } |
| else |
| { |
| oTextarea.oninput=change; |
| } |
|  |
| function change() |
| { |
| *//按钮的状态* |
| oTextarea.value?oA.className='send':oA.className=''; |
|  |
| *//正则是把汉字找到替换成两个字母，再取字母的长度* |
| var num=oTextarea.value.replace(/[^\x00-\xff]/g,'aa').length; |
| *//两个字母为一个字，要除以2，向上取整为了，输入一个字母的话要显示为一个字* |
| num=Math.ceil(num/2); |
|  |
| if(num<=140) |
| { |
| oP.innerHTML='发言请遵守社区公约，还可以输入<span>'+(140-num)+'</span>字'; |
| } |
| else |
| { |
| oA.className=''; |
| oP.innerHTML='发言请遵守社区公约，已经超过<span style="color:#f00;">'+(num-140)+'</span>字'; |
| } |
| } |
|  |
| oA.onclick=function() |
| { |
| if(this.className) |
| { |
| alert('成功'); |
| } |
| else |
| { |
| clearInterval(t); |
| t=setInterval(function(){ |
| iNum++; |
| if(iNum>=4) |
| { |
| iNum=0; |
| clearInterval(t); |
| } |
|  |
| if(iNum%2) |
| { |
| oTextarea.style.background='#f00'; |
| } |
| else |
| { |
| oTextarea.style.background=''; |
| } |
| },100); |
| } |
| }; |
|  |

1. 烟花效果

要点：

1. 颜色值，颜色值有一种表示方式是16进制。它的范围是从黑到白，即#000000 ~ #ffffff换成10进制为0~16777215。颜色值要取一个随机的
2. 字符串转换为16进度用toString(16)，toString的参数可以放进制单位
3. 取随机数的范围：Math.random()\*n-m，范围为-n ~n-m的范围。Math.random()\*50-30，范围为-30~50-30-1。每个烟花都要有一个随机的速度
4. 清除定时器，一个是炮弹的定时器，一个是烟花的定时器。后者需要点技巧
5. 定时器里有循环，循环的速度非常快，比定时器的时间要快。所以定时器每走一次，循环要循环完一轮。Js是单线程的，所以定时器一定会在循环完后再走。
6. 清除循环外的定时器技巧，在定时器上来就给一个count为0，在循环里累加count。但是循环里要给一个跳出循环的条件，这个是重点，一定要给。在定时器的结尾判断count为0的时候清除定时器。（循环给跳出的条件就是表明，循环已经结束了，不需要循环了，该清除定时器了）
7. 抛物线的模拟方法：烟花都给一个随机数后，让它们的Y轴的速度累加，就会往下运动，形成抛物线

|  |
| --- |
| function getColor() |
| { |
| *//颜色值的16进制范围为#000000~#ffffff,转为10进制后为0~16777215* |
| *//toString的参数为进制单位* |
| var str=Math.ceil(Math.random()\*16777215).toString(16); |
|  |
| *//不够6位的要在前面补0* |
| while(str.length〈6) |
| { |
| str='0'+str; |
| } |
| return '#'+str; |
| } |
| document.onclick=function(ev) |
| { |
| var oEvent=ev||event; |
| var oRedDiv=document.createElement('div'); |
| var l=oEvent.clientX; |
| var t=oEvent.clientY; |
| var aDiv=[]; |
|  |
| oRedDiv.className='pao'; |
| oRedDiv.style.left=l+'px'; |
| oRedDiv.style.top=document.documentElement.clientHeight+'px'; |
|  |
| document.body.appendChild(oRedDiv); |
|  |
| var timer=setInterval(function(){ |
| oRedDiv.style.top=oRedDiv.offsetTop-100+'px'; |
|  |
| *//炮弹高度已经超过鼠标点的高度，要消失，出花* |
| if(oRedDiv.offsetTop<=t) |
| { |
| document.body.removeChild(oRedDiv); |
| clearInterval(timer); |
|  |
| for(var i=0;i<40;i++) |
| { |
| var oDiv=document.createElement('div'); |
| oDiv.className='hua'; |
| oDiv.style.background=getColor(); |
| oDiv.style.left=l+'px'; |
| oDiv.style.top=t+'px'; |
| *//随机的范围为-30~30* |
| oDiv.speedX=Math.random()\*60-30; |
| oDiv.speedY=Math.random()\*60-30; |
|  |
| aDiv.push(oDiv); |
| document.body.appendChild(oDiv); |
| } |
|  |
| var newTimer=setInterval(function(){ |
| var count=0; |
| for(var i=0;i<aDiv.length;i++) |
| { |
| *//定时器走一下，循环就要循环一轮。烟花有的快，有的慢。当有一个移出屏幕后，此时定时器还没有结束，循环会进行下一轮循环，循环到空的就会报错。所以要上来检测一下* |
| if(!aDiv[i])continue; |
| aDiv[i].style.left=aDiv[i].offsetLeft+aDiv[i].speedX+'px'; |
| aDiv[i].style.top=aDiv[i].offsetTop+aDiv[i].speedY+'px'; |
|  |
| *//让Y轴的速度加加后，就会加速往下掉，模拟抛物线原理* |
| aDiv[i].speedY+=2; |
|  |
| *//烟花飞出浏览器的条件，从左边出去||从右边出去||从下边出去。因为Y轴要加速，所以上边的出去后也会掉下来* |
| if(aDiv[i].offsetLeft<0 || aDiv[i].offsetLeft>document.documentElement.clientWidth || aDiv[i].offsetTop>document.documentElement.clientHeight) |
| { |
| document.body.removeChild(aDiv[i]); |
| aDiv[i]=null; |
| } |
|  |
| count++; |
| } |
| if(count==0) |
| { |
| *//定时器上来定义这个变量，然后在循环里累加，但是循环里有个条件是跳出循环，当循环里的内容都跳出循环的时候，即所有烟花都已经移出浏览器了。这时候循环就不再执行了。它的值就不会累加，这时需要清除定时器* |
| clearInterval(newTimer); |
| } |
| },30); |
| } |
| },30); |
| }; |

1. 多种效果的幻灯片
2. 遇到的问题
3. 图片加载onload的时候，图片的src与onload函数里所用的图片的src一定要保持一致。图片会先循环完所有的src后再去执行onload里的东西。（瀑布流）
4. 如果循环让图片onload后加载，它不会按循环的顺序加载，而是会按图片大小，哪个图片小就先加载，大的后加载。图片命名尽量避免特殊符号，IE遇到特殊符号会有加载问题（360度图片旋转）
5. 有多个运动的情况下，点击当前的一定要清除其它的定时器，不然在快速点击不同的对象时候，都有定时器，都在运动就会出现问题（腾讯视频幻灯片）
6. display:inline-block; IE6/IE7下只有内联元素起作用，像a\span等;其它浏览器全兼容

解决办法：div {display:inline-block; } div {display:inline;zoom:1;}

但是如果div里的元素有绝对定位的话，就会失效。另一种解决办法用hack解决： div {display:inline-block; \*display:inline; \*zoom:1; (针对IE6) \_display:inline; \_zoom:1;}

1. setAttribute设置样式的时候是不兼容的，设置属性全兼容

<div id="div1" style="background:red;">asdfasdf</div>

oDiv.style.setAttribute('background', 'green');

这样是不兼容的

1. b函数在函数a里调用，b函数调用的时候是接不到a的参数的
2. json对象没有定义完前，this是指向window的，只有看到另一半大括号才算定义完成。在定义完成之前属性是访问不到的，也不能访问到上面已定义的属性。只能定义完成后访问，访问属性也只能用this.属性名，不能单独用属性名

对象属性的互相引用，先后定义两个对象，在第一个对象定义完成后就立马去调用下边对象的属性是找不到的，因为有定义的先后顺序。但是在有事件的情况下去调用下边对象的属性就能调到。因为执行事件时候两个对象已经定义完毕。

跟正常定义函数，事件去调用函数一样

|  |
| --- |
| var obj={ |
| box:document.getElementById('wrap'), |
| oUl:this.box.children, *//报错，访问不到this.box,属性不能访问已有的属性* |
| }; |

1. innerHTML相关问题：
2. 在innerHTML之前获取的元素，在innerHTML之后是不能使用它的**属性**的，可以使用元素，但宽高length等属性都会找不到。可以用appendChild代替，或者在之后重新获取一下
3. innerHTML必需添加合理嵌套的标签，不然也会报错
4. 表格不能用InnerHTML(会报错)，只能把整个table用字符串的方式加到div里（经验新锐写手专题）
5. IE不支持用innerHTML方法给select添加option(经验表单页面，省份联动select)
6. 获取li里的innerHTML时候，IE6下会在后面加一个空格
7. document.write重绘的是整个页面，innerHTML重绘的是页面的一部分
8. IE6、IE7相关问题：
9. 运动的时候，所运动的属性在IE下没有给初始值，侧IE下就不会运动
10. IE6下定位的时候，一定要给两个值，要不会出错，其它浏览器可以只给一个值
11. IE6下给margin负值是不起作用的，只能用相对定位
12. IE67下元素有margin值，如果再给它定位，margin值就会不起作用，原因是父级没有定位，解决办法就是给父级定位
13. IE6下如果a标签的href里是javascript:;那么它的onclick会失效，需要在事件函数的末尾加上return false;(知道表单)
14. IE67元素透明的话，无法触发此无素的事件。解决办法,设置背景，或者给一个背景图片，url里随便写
15. IE67下ul overflow:hidden; li position:relative;会不受overflow:hidden的限制
16. 标准浏览器了，可以用.id名获取到这个id，其它浏览器不行
17. 获取select下选中的option的方法以及select改变的事件，事件是onchange 获取方法是：select.options[select.selectedIndex].text，select有个options属性，是所有的option，有个selectedIndex属性为当前选中的option的系数。
18. Offset的问题：

offsetLeft/offsetTop是相对于有定位的第一个父级的位置

父级没定位：不管自己有没有定位是到html的距离

父级有定位：

IE67自己没定位：到body的距离

IE67自己有定位：到定位父级的距离

解决办法是给元素自身还有父级都加定位

会受border与margin的影响

1. 变量作用域。函数a在别处定义，在函数b里调用，b里的变量a调用的时候访问不到，只能重新在b里定义一个变量
2. 如果一个a标签点击后去请求ajax数据，要么给a的href设值javascript:;，要么点击事件的最后加一个return false。不然ajax会出错。最好同时都给加上

知道团队活动遇到的问题：

1.数据结口一定要加上时间戳，防止IE下的缓存

2.如果请求过来的数据里有需要按钮要交互之类的，最好别用innerHTML，或者给事件里setTimeou(function(){},0)，IE下数据可能会有延迟

3.细心

成都百科活动问题

1.如果运动某个属性，IE下一定要在样式里写上这个属性，不然IE下的运动会有问题

2.事件冒泡，一个层里有两处事件一定要阻止事件冒泡!!!!，不然第二个事件会不起作用

5.如果切换样式的话，最好用class而不用id

1. Jquery源码

* typeof

判断类型是否为undefined最好用typeof，全兼容

window.a==undefined; //IE9以及以下去判断xmlNode.method或属性的时候，用这个方法是判断不出来的

typeof window.a=='undefined';

Jquery里会把很多全局属性都存起来，目的就是为了压缩，压缩的时候能把这个变量压缩成一个单词

* $('li').css('background','red');

可以看作两个单独部分(两个处理函数)：

1. $('li')

这里面要获取的是对象

this=[0:'li',1:'li',2:'li',3:'li',lenth:3]; //还会有其它属性，length为元素的长度

1. css('background','red');

这里面要把每个对象做对应的处理

由于函数作用域的关系，在1里面获取的属性在2里是访问不到的。然而面向对象里有一个this，它可以被当作每个对象，而且在任何地方都可以访问到。所以jquery就把1里的元素都放到构造函数里，在2里就可以访问了

1. Javascript异步编程

Javascript语言的执行环境是"单线程"。是指一次只能完成一件任务。如果有多个任务，就必须排队，前面一个任务完成，再执行后面一个任务，以此类推。坏处是只要有一个任务耗时很长，后面的任务都必须排队等着，会拖延整个程序的执行。常见的浏览器无响应（假死），往往就是因为某一段Javascript代码长时间运行（比如死循环），导致整个页面卡在这个地方，其他任务无法执行

实现异步编程的方法

* 回调函数(利用setTimeout)

function a(callBack) {

setTimeout(function(){

//如果a需要处理很庞大的数据，a函数的代码放到setTimeout里

var n = 0;

for (var i = 1; i < 100000000; i++) {

n = n+i;

}

document.getElementById('a1').value = n;

},0);

callBack(); //b函数放到setTimeout外面，就会先执行b后执行a

}

function b() {

document.getElementById("a2").value = 22222;

}

a(b);

* 事件监听(自定义事件)

采用事件驱动模式。任务的执行不取决于代码的顺序，而取决于某个事件是否发生。可以绑定多个事件，每个事件可以指定多个回调函数，有利于实现模块化。缺点是整个程序都要变成事件驱动型，运行流程会变得很不清晰

f1.on('done', f2); //f1发生done事件后执行f2

function f1(){

setTimeout(function () {

// f1的任务代码

f1.trigger('done'); //执行完成后立马触发done事件，再执行f2

}, 1000);

}

* 发布/订阅(观察者模式)

假定存在一个"信号中心"，某个任务执行完成，就向信号中心"发布"一个信号，其他任务可以向信号中心"订阅"这个信号，从而知道什么时候自己可以开始执行。与"事件监听"类似

jQuery.subscribe("done", f2); //f2订阅done事件，如果接收到done信号就执行

function f1(){

setTimeout(function () {

// f1的任务代码

jQuery.publish("done"); //f1执行完后发送done信号

}, 1000);

}

jQuery.unsubscribe("done", f2); //f2执行完后取消订阅

* Promises对象

Promises对象是CommonJS工作组提出的一种规范，目的是为异步编程提供统一接口。

简单说，它的思想是，每一个异步任务返回一个Promise对象，该对象有一个then方法，允许指定回调函数

1. ECMAScript5的js API
2. ECMAScript5的JS严格模式

* 描述

除了正常运行模式，ECMAscript5添加了第二种运行模式："严格模式"strict mode，这种模式使得Javascript在更严格的条件下运行

* 目的

设立"严格模式"的目的，主要有以下几个：

1. 消除Javascript语法的一些不合理、不严谨之处，减少一些怪异行为;
2. 消除代码运行的一些不安全之处，保证代码运行的安全
3. 提高编译器效率，增加运行速度
4. 为未来新版本的Javascript做好铺垫

* 调用

**针对整个脚本文件**

将"use strict"放在脚本文件的第一行，则整个脚本都将以"严格模式"运行。如果这行语句不在第一行，则无效，整个脚本以"正常模式"运行。IE10以下不支持，老版本的浏览器会把它当作一行普通字符串，加以忽略

<script type="text/javascript">

"use strict";

console.log("这是严格模式");

</script>

<script type="text/javascript">

console.log("这是正常模式");

</script>

**针对单个函数**

将"use strict"放在函数体的第一行，则整个函数以"严格模式"运行。更好的做法是，将整个脚本文件放在一个立即执行的匿名函数之中

function strict() {

"use strict";

return "这是严格模式";

}

function notStrict() {

return "这是正常模式";

}

* 全局变量必需用var声明

"use strict";

v = 1; //报错，v未声明

for(i = 0; i < 2; i++) {} //报错，i未声明

* 禁止使用with语句

"use strict";

var v = 1;

with (o){ // 语法错误

v = 2;

}

* eval里的变量只能用在自己的作用域

"use strict";

var x = 2;

console.info(eval("var x = 5; x")); // 5

console.info(x); // 2

* 禁止this关键字指向全局对象

function f(){

return !this; //false，因为this指向window，!this就是false

}

f();

function f(){

"use strict";

return !this; //true，因为严格模式下，this的值为undefined，所以!this为true

}

f();

因此，使用构造函数时，如果忘了加new，this不再指向全局对象，而是报错

function f(){

"use strict";

this.a = 1;

};

f();//报错，this未定义

* 禁止在函数内部遍历调用栈

function f1(){

"use strict";

f1.caller; //报错

f1.arguments; //报错

}

f1();

* 禁止删除变量

严格模式下无法删除变量。只有configurable设置为true的对象属性，才能被删除

"use strict";

var x;

delete x; //语法错误

var o = Object.create(null, { //新增的创建对象的方法Object.create

'x': {

value: 1,

configurable: true

}

});

delete o.x; //删除成功

* 显式报错

正常模式下，对一个对象的只读属性进行赋值，不会报错，只会默默地失败。严格模式下，将报错

"use strict";

var o = {};

Object.defineProperty(o, "v", { value: 1, writable: false });

o.v = 2; // 报错

严格模式下，对一个使用getter方法读取的属性进行赋值，会报错

"use strict";

var o = {

get v() { return 1; }

};

o.v = 2; //报错

严格模式下，对禁止扩展的对象添加新属性，会报错

"use strict";

var o = {};

Object.preventExtensions(o);

o.v = 1; //报错

严格模式下，删除一个不可删除的属性，会报错

"use strict";

delete Object.prototype; //报错

* 对象不能有重名的属性

"use strict";

var o = {

p: 1,

p: 2

}; //语法错误

* 函数不能有重名的参数

正常模式下，如果函数有多个重名的参数，可以用arguments[i]读取。严格模式下，这属于语法错误

"use strict";

function f(a, a, b) { //语法错误

return ;

}

* 禁止八进制表示法

正常模式下，整数的第一位如果是0，表示这是八进制数，比如0100等于十进制的64。严格模式禁止这种表示法，整数第一位为0，将报错

"use strict";

var n = 0100; //语法错误

* 不允许对arguments赋值

"use strict";

arguments++; //语法错误

var obj = { set p(arguments) { } }; //语法错误

try { } catch (arguments) { } //语法错误

function arguments() { } //语法错误

var f = new Function("arguments", "'use strict'; return 17;"); //语法错误

* arguments不再追踪参数的变化

function f(a) {

a = 2;

return [a, arguments[0]];

}

f(1); //正常模式为[2,2]

function f(a) {

"use strict";

a = 2;

return [a, arguments[0]];

}

f(1); //严格模式为[2,1]

* 禁止使用arguments.callee

无法在匿名函数内部调用自身了

"use strict";

var f = function() { return arguments.callee; };

f(); //报错

* 函数必须声明在顶层

将来Javascript的新版本会引入"块级作用域"。为了与新版本接轨，严格模式只允许在全局作用域或函数作用域的顶层声明函数。也就是说，不允许在非函数的代码块内声明函数

"use strict";

if (true) {

function f() { } //语法错误

}

for (var i = 0; i < 5; i++) {

function f2() { } //语法错误

}

* 保留字

为了向将来Javascript的新版本过渡，严格模式新增了一些保留字：implements, interface, let, package, private, protected, public, static, yiel, class, enum, export, extends, import, super使用这些词作为变量名将会报错

function package(protected) { //语法错误

"use strict";

var implements; //语法错误

}

<http://www.ruanyifeng.com/blog/2013/01/javascript_strict_mode.html>

1. 知识点类

* 判断类型方法，推荐最后一种：

obj.constructor == Array //true false

obj instanceof Array //true false

Object.prototype.toString.call(obj) == '[object Array]' //true false

* 子级影响父级

IE下嵌套层中，给父级添加over、out事件后，子级会影响父级的事件（放大镜实例），不是冒泡的原因

解决办法：

1. 添加一个遮罩层，放在最下边，并且把嵌套的分开。让遮罩层全透明
2. 用onmouseenter、onmouseleave代替onmouseover、onmouseout事件，这两个事件子级不会影响父级。可能会有兼容性问题，这两个事件最早是IE的，其它的老版本浏览器（chrom、firefox、safiry）不支持，新版的都支持
3. 用js写个判断两层是不是嵌套关系的函数去解决

function elContains(a, b){ //判断两个元素是否是嵌套关系，是嵌套返回真，不是返回假

return a.contains ? a != b && a.contains(b) : !!(a.compareDocumentPosition(b) & 16);

}

a = this, b = ev.relatedTarget;

* 布局转换问题

在同一个代码块中，设置css样式会有优先级问题（无定位转定位），position的样式优先级要高于left、top。

aLi[i].style.left = aLi[i].offsetLeft + 'px';

aLi[i].style.top = aLi[i].offsetTop + 'px';

aLi[i].style.position = 'absolute';

这样所有li都会叠在一块，因为position的优先级高，即使写在后面，也会优先执行。顺序是先执行定位再设置offset值，先执行定位后，会全部叠在一块，left、top的值就都为0

解决办法：

1. 分代码块去写，先把offset放在上面代码块中，再把定位放到下面代码块中。分开代码块要注意margin的值。要在定位代码块中把margin的值去掉。与offset值不包含margin值，所以要清掉margin值
2. 把left、top存到数组里，转换定位时候取数组里对应的值（推荐方法）

* 小数的问题（运动中透明度会用100，而不用0.几）

alert(0.1+0.2) //0.30000000000000004

alert(0.7+0.2) //0.8999999999999999

算出的值是近似值，小数点后两位四舍五入后得到正确值。计算机存储的小数为近似值。

1/3=0.33333 3无限循环，如果3无限循环计算机有再大的内在也不够存

原因：

JavaScript 浮点数计算的 Bug。和计算机最终转换成二进制计算有关系

* 速度版运动框架的Bug

1. 在运动的过程当中，把浏览器最小化或者切换到别的页面，过一段时间再打开后发现运动还没有完成。是因为最小化或切换页面后浏览器让定时器变慢了。浏览器为了提高性能，在你不看的时候，它会把定时器变得缓慢来提高性能，当切换回来后，浏览器再把定时器切换正常。(如果页面不是激活状态下的话，动画会自动暂停，有效节省了CPU开销。与requestAnimationFrame相同)
2. 速度版的运动如果运动几个属性值比例不同，就不能同时到达，因为速度相同，距离不同的话肯定不能同时到达。用时间版的解决

解决办法：

1. 当浏览器缩小或切换的时候，让定时器定住。切回来的时候再开开，接着上一次走

window.onfocus = function(){ //加给window表示当页面显示时候

console.log(1);

timer = setInterval(toRun,2000);

};

window.onblur = function(){ //当页面不显示的时候

console.log(2);

clearInterval(timer);

};

1. 用时间版运动框架解决

* 找字符串中重复最多的字符和次数

var str = 'assssjdssskssalsssdkjsssdss';

var arr = str.split('');

str = arr.sort().join(''); //str=aadddjjkklsssssssssssssssss

var value = '';

var index = 0;

var re = /(\w)\1+/g; //\w：字符，括起来成子项，\1：找重复的子项，+：重复多个

str.replace(re,function($0,$1){

//$0：aa ddd jj kk sssssssssssssssss 没有l，因为l只有一个，不重复

if(index<$0.length){ //如果长度小就会一直走if，直到找到最大length就不走了

index = $0.length;

value = $1;

}

});

alert('最多的字符:'+value+',重复的次数:'+index);