# *补充*

# matches

如果元素被指定的选择器字符串选择，**Element.matches()**  方法返回true; 否则返回false。

有一些浏览器使用前缀, 在非标准名称  matchesSelector () 下实现了这个方法!

## 语法

let result = element.matches(selectorString);

* result 的值为 true 或 false.
* selectorString 是个css选择器字符串.

# 数据类型转换

## **任何简单类型转换成String**

### (1).变量+""; 不希望值发生改变，所以连接"";

### (2).string(变量) 构造函数法

### (3).变量.toString();注意：undefined和null不可以(无toString方法)。

## 2. 任何简单类型转换成Number

此转换容易产生NaN，一旦被转换的变量中含有非数字字符，都容易出现NaN

### (1). 变量-\*/一个数字（有非数字字符会出现NaN）

例：var num1 = “11”- 0; var num2 =“11”\* 1;var num =“11”/1;

JS底层做了一个强制类型转换，把字符串转换成了Number进行运算。

注意: true 转换成数值结果是： 1；

false 转换成数值结果是： 0；

null 转换成数值结果是： 0；

undefined 转换成数值结果是： NaN；

//undefined和任何值运算结果都是NaN

### (2). Number(变量)（有非数字字符会出现NaN）

var num1 = Number(“18”); 把字符变成了数字。

var num2 = Number(“18.99”); 结果为18.99数字型。（有小数也转换）

### (3). parseInt()和parseFloat()（译为取整和取浮点数）

空字符串parseInt()和parseFloat()返回NaN，Number("")返回0

parseInt(变量)：如果变量中首个字符为字母则结果为NaN。

否则取出现首个非数字前的整数。

123 = parseInt(“123.123aaaa”);

parseFloat(变量)：如果变量中首个字符为字母则结果为NaN。

否则取出现首个非数字前的浮点数。（没有小数取整）

123.123 = parseFloat(“123.123aaaa”);

## 3.任何简单类型转换成Boolean

任何数据类型都可以转换成boolean类型，所以和以往两个转换不同；

### (1). Boolean(变量)

var bool = Boolean(“1111”); bool为true；

### (2). ！！变量

第一个逻辑非操作会基于无论什么操作数返回一个与之相反的布尔值

第二个逻辑非操作则对该布尔值求反

于是就得到了这个值真正对应的布尔值

## 4.引用类型的数据转换

### 数组转🡺字符串

join(); 将数组转化为字符串,中间传入参数进行连接

注意:传入多个参数时,只会选取第一个参数,若**不传参,默认以逗号**连接

分析:该方法**不会改变原数组**,返回以传入的参数为连接的字符串形式的数组元素

var arr = ["关羽","张飞","刘备"];

console.log(arr.join());//如果无参,数组中的元素用**逗号**链接成一个字符串

console.log(arr.join("-"));//如果带参,数组中的元素用**参数**链接成一个字符串

console.log(arr.join(" "));//如果是空格,数组中的元素用**空格**链接成一个字符串

console.log(arr.join(""));//无缝连接,用的是"";

### 字符串转🡺数组

split(); 把字符串变成数组

var str = "aaa,bbb,ccc";

console.log(str.split());//把这个字符串做为一个元素; ['aaa,bbb,ccc']

console.log(str.split(""));//空字符串把所有元素都变成单个元素; ["a","a","a",",","b","b","b",",","c","c","c"]

console.log(str.split(","));//指定参数后,参数不会出现在数组中 ["aaa","bbb","ccc"]

push(); 把字符串添加到数组中

var array=[];

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

var tr="<tr>";

tr+="<td>"+datas[i].title+"</td>";

tr+="<td>"+datas[i].url+"</td>";

tr+="<td>"+datas[i].desc+"</td>";

tr+="</tr>";

console.log(typeof tr); //string

//将元素添加到数组里面去.

array.push(tr);

### 对象转🡺字符串

JSON.stringify(obj) 对象转字符串的方式

var data = {

"name":"tom",

"age":26

};

var str = JSON.stringify(data)

//打印的是字符串

console.log(str);

打印结果: {"name":"tom","age":26}

### 字符串转🡺对象

JSON.parse(str) 字符串转换成对象

console.log(JSON.parse(str));

打印结果: Object

age: 26

name: "tom"

\_\_proto\_\_: Object

### Json字符串变数组

Eval()函数

通过eval() 函数可以将JSON字符串转化为对象

### 数组转对象

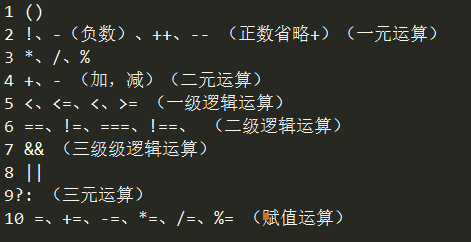
Eval()函数

# 二、操作符与优先级

## 1.操作符种类

1. 算数运算符（+—\*/...）
   1. 一元运算符：正号、负号、++、--、平方等一个变量就能运算
   2. 二元运算符：+-\*/%等两个变量才能运算
   3. 三元运算符： 值1？值2：值3；
2. 逻辑运算符（ |& ! ）（或且非）
3. 比较运算符（<、>、==、>=...）
4. 赋值运算符（=、+=、-=、\*=、/=、%=）

## 2..执行顺序(优先级)



## 3..短路运算

### 1. &&链接两个boolean类型，有一个是false结果就是false。

链接值不是布尔类型时，按照成布尔类型计算，结果本身不变。（非布尔）

如果不是boolean类型值，会按照对应的布尔类型之计算，然后返回的值不变。

例子： 1 = 2&&1； 0 = 0 && 1； 都是true取后面，都是false取前面。

### 2.||链接两个boolean类型，有一个是true结果就是true。

链接值不是布尔类型时，按照成布尔类型计算，结果本身不变。（非布尔）

如果不是boolean类型值，会按照对应的布尔类型之计算，然后返回的值不变。

例子： 2= 2||1； 1 = 0 || 1； 都是true取前面，都是false取后面。

# 三、流程控制

1.选择结构: 共有两种，if语句和switch语句。If常用，switch为特殊情况使用，判断条件出现的情况特别多的时候用switch，其他时候if语句比较方便。

2.循环结构: 共3种，for/while/do...while；可根据需求选择使用；

(1)遍历数组首选for循环，简单循环使用for。

(2)而while循环强调，不记循环次数（不知道循环多少次），首选while。

(3)最后do...while循环强调，无论怎样，至少执行一次是，使用do...while。

1.if语句

1. If语句用法有三种
2. if(条件1){程序1}
3. if(条件1){程序1}else{程序2}
4. if(条件1){程序1}else if(条件2){程序2}...else{程序n}
5. 三目运算（也叫三元运算）（目或者元代表几个表达式）

三目运算可以替代部分if...else...功能，运算简单，使用方便，代码清晰。

表达式1？值1：值2. Switch语句

## 2. switch语句

switch (值1) {

case value1:

程序1；

break; // break 关键字会导致代码执行流跳出 switch 语句

case value2:

程序2；

break;

default:

程序3；

}

注意：

break可以省略，如果省略，代码会继续执行下一个case

switch 语句在比较值时使用的是全等操作符，因此不会发生类型转换

（例如，字符串 "10" 不等于数值 10）。

## 3.for循环

1. 执行流程

for (变量;条件1;条件2){ 执行程序 }

执行过程：变量->条件1->执行程序->条件2->条件1->执行程序.....

直到条件1不成立，跳出循环。

1. 三个表达式均为可选，但是必须写分号！！！

for(;;){程序} 死循环;

# 四. 字符串操作

### indexOf

给字符查索引

Str.indexOf("a") 返回索引值;查不到返回-1

### charAt

功能: 给索引查字符,获取指定位置的字符

Str.charAt(索引值)

### chatCodeAt

功能:获取指定位置字符的ASCll码

### replace

返回一个由替换值替换一些或所有匹配的模式后的新字符串。模式可以是一个字符串或者一个[正则表达式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/RegExp), 替换值可以是一个字符串或者一个每次匹配都要调用的函数。

语法[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/replace$edit#Syntax)

str.replace(regexp|substr, newSubStr|function)

参数

**regexp (pattern)**

一个 [RegExp](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/RegExp) 对象或者其字面量。该正则所匹配的内容会被第二个参数的返回值替换掉。

**substr (pattern)**

一个要被 newSubStr 替换的[字符串](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String)。其被视为一整个字符串，而不是一个正则表达式。仅仅是第一个匹配会被替换。

**newSubStr (replacement)**

 用于替换掉第一个参数在原字符串中的匹配部分的 [字符串](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String)。该字符串中可以内插一些特殊的变量名。参考下面的[使用字符串作为参数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/replace#使用字符串作为参数)。

**function (replacement)**

一个用来创建新子字符串的函数，该函数的返回值将替换掉第一个参数匹配到的结果。参考下面的[指定一个函数作为参数](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/replace#指定一个函数作为参数)。

返回值

一个部分或全部匹配由替代模式所取代的新的字符串。

示例[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/replace$edit#Examples)

在 **replace() 中使用正则表达式**

在下面的例子中，replace() 中使用了正则表达式及忽略大小写标示。

var str = 'Twas the night before Xmas...';

var newstr = str.replace(/xmas/i, 'Christmas');

console.log(newstr); // Twas the night before Christmas...

在**replace()**中使用 **global** 和 **ignore** 选项

下面的例子中,正则表达式包含有全局替换(g)和忽略大小写(i)的选项,这使得replace方法用'oranges'替换掉了所有出现的"apples".

var re = /apples/gi;

var str = "Apples are round, and apples are juicy.";

var newstr = str.replace(re, "oranges");

// oranges are round, and oranges are juicy.

console.log(newstr);

### slice

**slice()** 方法提取一个字符串的一部分，并返回一新的字符串。

语法[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/slice$edit#Syntax)

str.slice(beginSlice[, endSlice])

参数

**beginSlice**

从该索引（以 0 为基数）处开始提取原字符串中的字符。如果值为负数，会被当做 sourceLength + beginSlice 看待，这里的sourceLength 是字符串的长度 (例如， 如果beginSlice 是 -3 则看作是: sourceLength - 3)

**endSlice**

可选。在该索引（以 0 为基数）处结束提取字符串。如果省略该参数，slice会一直提取到字符串末尾。如果该参数为负数，则被看作是 sourceLength + endSlice，这里的 sourceLength 就是字符串的长度(例如，如果 endSlice 是 -3，则是, sourceLength - 3)。

描述[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/slice$edit#Description)

slice() 从一个字符串中提取字符串并返回新字符串。在一个字符串中的改变不会影响另一个字符串。也就是说，slice 不修改原字符串，只会返回一个包含了原字符串中部分字符的新字符串。

**注意**：slice() 提取的新字符串包括beginSlice但**不包括 endSlice**。

例1：str.slice(1, 4) 提取新字符串从第二个字符到第四个 (字符索引值为 1, 2, 和 3)。

例2：str.slice(2, -1) 提取第三个字符到倒数第二个字符。

例子[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/slice$edit#Examples)

使用 **slice()** 创建一个新的字符串

下面例子使用 slice() 来创建新字符串:

var str1 = 'The morning is upon us.';

var str2 = str1.slice(4, -2);

console.log(str2); // OUTPUT: morning is upon u

给 **slice()** **传入负值索引**

下面的例子在 slice() 使用了负值索引:

var str = 'The morning is upon us.';

str.slice(-3); // returns 'us.'

str.slice(-3, -1); // returns 'us'

str.slice(0, -1); // returns 'The morning is upon us'

### split

将一个[String](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/String)对象分割成字符串数组，通过将字符串分成子串。

语法

str.split([separator[, limit]])

// [ ]表示可选项，所以 [, limit] 这种写法才是正确的！

参数

**separator**

指定用来分割字符串的字符（串）。separator 可以是一个字符串或[正则表达式](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/RegExp)。 如果忽略 separator，则返回整个字符串的数组形式。如果 separator 是一个空字符串，则 str 将会把原字符串中每个字符的数组形式返回。

**limit**

一个整数，限定返回的分割片段数量。split 方法仍然分割每一个匹配的 separator，但是返回的数组只会截取最多 limit 个元素。

### substring

();和slice一样;

//1.智能调换.如果前面的比后面的大.前后交换

//2.负数全部截取;

### substr

返回一个字符串中从指定位置开始到指定字符数的字符。

语法

str.substr(start[, length])

参数

**start**

开始提取字符的位置。如果为负值，则被看作 strLength + start，其中 strLength 为字符串的长度（例如，如果 start 为 -3，则被看作 strLength-3）。

**length**

可选。提取的字符数。

示例[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/substr$edit#示例)

例子：使用 **substr**

var str = "abcdefghij";

console.log("(1,2): " + str.substr(1,2)); // (1,2): bc

console.log("(-3,2): " + str.substr(-3,2)); // (-3,2): hi

console.log("(-3): " + str.substr(-3)); // (-3): hij

console.log("(1): " + str.substr(1)); // (1): bcdefghij

console.log("(-20, 2): " + str.substr(-20,2)); // (-20, 2): ab

console.log("(20, 2): " + str.substr(20,2)); // (20, 2):

### join

### toLowerCase&toUpperCase

str.toLowerCase();转小写

str.toUpperCase();转大写

### trim

**trim()**方法会从一个字符串的两端删除空白字符。在这个上下文中的空白字符是所有的空白字符 (space, tab, no-break space 等) 以及所有行终止符字符（如 LF，CR）。

语法[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/Trim$edit#语法)

str.trim()

描述[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/Trim$edit#Description)

trim() 方法并不影响原字符串本身，它返回的是一个新的字符串。

例子[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/Trim$edit#Examples)

使用 **trim()**

下面的例子中将显示小写的字符串 'foo':

var orig = ' foo ';

console.log(orig.trim()); // 'foo'

// 另一个.trim()例子，只从一边删除

var orig = 'foo ';

console.log(orig.trim()); // 'foo'

兼容旧环境[**EDIT**](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/String/Trim$edit#兼容旧环境)

如果 trim() 不存在，可以在所有代码前执行下面代码

if (!String.prototype.trim) {

String.prototype.trim = function () {

return this.replace(/^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g, '');

};

}

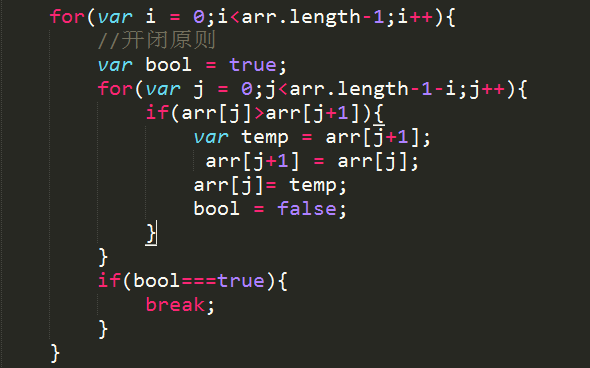
# 五. 数组

1. **创建数组**:

var arr = new Array();

for（var i= 0；i<arr.length;i++）{};

1. **冒泡排序**



## 3. **数组操作**

### (1) 添加和删除

**push()** //在数组最后面插入项，返回修改后的数组

数组1改后的长度 = 数组1.push(元素1);

**pop()** //取出数组中的最后一项，返回最后一项

被删除的元素 = 数组1.pop();

分析：不接受传参（传参会被忽略）

**unshift()** //在数组最前面插入项，返回修改后的数组

数组1改后的长度 = 数组1.unshift(元素1);

**shift()**  //取出数组中的第一个元素，返回第一项

被删除的元素 = 数组1.shift();

### (2) 翻转和排序

1.reverse();数组元素反转：原数组同时被修改,不接受传参.

2.sort(); 数组元素排序：原数组同时被修改,默认从小到大.

Arr.sort(function( a,b ){ return a- b }) 升序排列

Arr.sort(function( a,b ){ return b- a }) 降序排列

1. 数字优先字母
2. 单词排序按首字母排序,若首字母相同则比较第二个字母,以此类推
3. 多位数排序,同样从第一位开始比较,若相同则比较第二位,以此类推 如: [ ‘1’ ,’ 12’, ‘3’ ,’33’]

### (3) 链接截取替换查找

1.concat从调用者数组的最后面链接另一个数组，最后返回一个新数组（原数组不变）

2.slice (start,end);数组元素的截取；(根据索引值截取，start,end都是索引值)

**console**.log(arr3.slice());//截取整个数组；  
**console**.log(arr3.slice(2));//从索引值为2，截取到数组最末尾；  
**console**.log(arr3.slice(2,4));//按索引值截取，包左不包右；[2,4);  
**console**.log(arr3.slice(-2));//负值为最后几位;  
**console**.log(arr3.slice(4,2));//前大后小为[];

1. splice(); 替换和删除数组中元素的功能;(原数组被修改);

var aaa = arr3.splice();//未指定要删除的元素.  
**console**.log(aaa);  
**console**.log(arr3);  
  
var aaa = arr3.splice(0);//一个参数为:从该索引值位置截取到最后;  
**console**.log(aaa);//原数组整体截取//删除;  
**console**.log(arr3);//原数组没有了//删除;  
  
  
var aaa = arr3.splice(0,3);//一个参数为:从该索引值位置截取到最后;第二个参数为要截取的个数;  
**console**.log(aaa);//原数组截取3个组成数组 //删除;  
**console**.log(arr3);//原数组剩下的  
  
//删除的特殊用法:可以删除指定位置的元素;  
arr3.splice(2,1);//指定索引值删除;  
**console**.log(arr3);  
  
  
  
  
替换;第三个参数  
var aaa = arr3.splice(2,1,"abc");//替换  
**console**.log(arr3);  
  
var aaa = arr3.splice(2,0,"abc");//指定位置添加  
**console**.log(arr3);  
  
批量替换;不能使用数组,只能一个一个的使用参数;  
arr3.splice(3,3,"你","我","他");  
**console**.log(arr3);  
  
var arr4 = ["你","我","他"];  
arr3.splice(3,3,arr4);//数组整体塞进去；  
**console**.log(arr3);

4. indexOf()、lastIndexOf() ：给元素查索引值，一个从前往后查，一个人从后往前查；

// 如果没找到返回-1； （索引值一旦加载永远不会改变）；  
  
var arr = ["a","b","c","b","a","d","e"];  
**console**.log(arr.indexOf("a"));//0  
**console**.log(arr.lastIndexOf("a"));//4  
  
**console**.log(arr.indexOf("c"));//2  
**console**.log(arr.lastIndexOf("c"));//2  
  
//查不到是-1；  
**console**.log(arr.indexOf("x"));  
**console**.log(arr.lastIndexOf("x"));

5.join把数组变成字符串作用是将数组各个元素是通过指定的分隔符进行连接成为一个字符串。(**把数组转换成字符串,** **不会改变原数组**)

var arr = ["关羽","张飞","刘备"];  
  
**console**.log(arr.join());//如果无参,数组中的元素用逗号链接成一个字符串  
**console**.log(arr.join("-"));//如果带参,数组中的元素用参数链接成一个字符串  
**console**.log(arr.join(" "));//如果是空格,数组中的元素用空格链接成一个字符串  
**console**.log(arr.join(""));//无缝链接,用的是"";  
**console**.log(arr);//原数组不会被修改

### 转换方法

1.arr.toString()

var arr = [‘a’,’b’,’c’];

var result = arr.toString();

console.log(result); // ‘a,b,c’

console.log(arr); // [‘a’,’b’,’c’]

功能：将数组转化为字符串，中间以逗号连接.

分析:1.该方法不会改变原数组2.该方法返回以逗号链接的字符串形式的数组.

2.join();

功能:将数组转化为字符串,中间传入参数连接;

分析:1.该方法不会改变原数组2.该方法返回以传入的参数连接的字符串形式的数组.

### (5) 遍历方法(ES5扩展)

##### 1. ForEach

功能: 遍历数组中的每一个元素, 并且将每一个元素的处理过程交给回调函数

语法: 数组.forEach( callback )

回调函数中会传入两个参数, 一个是遍历的数组元素, 一个是当前的索引.

例如: 遍历打印数组中的每一个元素与索引号

传统:

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

console.log( i + ', ' + arr[ i ] ); // 打印, 索引号 与 当前元素

}

forEach:

arr.forEach( function ( value, index ) {

console.log( index + ', ' + value );

} );

比较jq 中的 each 方法

##### JQ中each方法与forEach方法的异同

-> 参数

jq: ( arr, 回调函数 ), 回调函数的参数是 i, v

数组: ( 回调函数 ), 回调函数的参数是 v, i

-> 返回值

jq: 有返回值, 即当前遍历的数组或对象.

数组: **没有返回值**

补充: jq 方法可以遍历**数组, 伪数组, 对象**; 而数组中的方法只能遍历数组.

-> 执行的功能

jq 中 this 指的是当前遍历的对象, 而数组中的 this 默认是 window.

在 jq 中遍历的数据如果是一个基本类型的数据( number, boolean, string ),

this 指其包装对象. 如果需要拿到遍历的基本类型数组的值, 要么使用 valueOf, 要么使用参数 v

数组的 forEach 方法不能停止, 只能全部遍历完毕才会结束.

jq 中直接给回调函数返回 false 即可跳出循环.

##### map

功能: 将一个集合按照特定的规律变成另一个集合..

语法: 数组.map( 回调函数 )

简单的来理解, map也是在遍历数组. 这种理解不准确…

**分析**: 如果回调函数没有返回值, 注意默认返回了 undefined

例如: 有数字构成的集合, 将其中 的数字扩大一倍.

map 的实现逻辑

1> map 函数要返回一个数组

2> map 返回的这个数组的元素由 **回调函数的返回值**决定

map 函数也是如此, map 函数中的回调函数来实现我们的规则

var newArr = arr.map( function ( v, i ) {

return v \* 2;

});

我们的要求是将数组中的元素转换成另一个东西, 那么这个**转换的过程由回调函数**来实现

典型的案例:

'a,b,c'.split( ',' ).map(function ( v ) {

return '<td>' + v + '</td>';

});

// [ 'a', 'b', 'c' ] => [ '<td>a</td>', '<td>b</td>', '<td>c</td>' ]

将箭头函数引入 map. 上面的案例就可以改良成

'a,b,c'.split( ',' ).map( v => '<td>' + v + '</td>' );

##### jq 中map 方法与map的异同

-> 参数

 与其回调函数的参数一样

-> 返回值

数组的返回值是一个数组, jq 的返回值也是数组

**数组的返回值必须保证每一个回调函数都有返回数据. 如果没有相当于 返回 undefined**

jq 中 map 方法的返回值中的元素由 回调函数的返回值决定, 如果没有回调函数返回,则为空数组

其实在 **jq 中 map** 相当于**结合了数组中提供的 map 与 filter** 的功能

-> 执行的功能

如果 jq 作为 $.map 来调用, 回调函数中不能使用 this, 其 this 是 window, 参数是 v,i

如果 jq 作为 $( ... ).map() 来调用, 回调函数中的 this 就是当前遍历的元素. 其参数是 i, v

补充:

$.map( arr, function ( v, i ) { ... } )

$( '...' ).map( function () { ... } )

##### 3.4. every 和 some

some 方法, 表示判断数组中的元素只要含有某一个条件即可

every 方法, 表示判断数组中的每一个元素必须含有某个条件

语法: 数组.方法名( 回调函数 ) -> boolean

说明:

1> 回调函数依旧是遍历数组的每一个元素

2> 回调函数的参数依旧是 v, i

3> 回调函数的作用是用来**判断数组中的元素**, 所以回调函数有返回值, 返回一个 boolean

4> some 方法凡是发现满足了条件的元素就会停止遍历, 并返回 true, 否则返回 false.

5> every 方法凡是发现了一个元素不满足条件就会停止遍历, 并返回 false, 否则返回 true.

该方法与逻辑中断类似 是**逻辑中断的升级版**.

|| 表达式1 || 表达式2

如果表达式1为假, 才会判断表达式2, 如果表达式1已经真 那么不执行表达式2, 就得到结果

&& 表达式1 && 表达式2

如果表达式1为假, 那么不在判断表达式2, 直接得到结果为假, 如果表达式1为真, 再判断表达式2

##### 5. filter

功能: 将一个数组中符合某要求的所有元素筛选出来, 组成一个新的数组返回.

语法: 数组.filter( 回调函数 ) -> 新数组

回调函数的参数依旧是 v, i

回调函数判断的时候, 判断元素 v 是否符合要求, 如果符合返回 true, 否则返回 false

filter 就会将所有的**符合元素组成新的数组**

案例: 将所有的数字中奇数取出来.

var arr = [ 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 ];

arr.filter( v => v % 2 == 1 );

等价于

arr.filter( function ( v ) {

return v % 2 == 1;

} );

filter: 在数组中**筛选你需要的数据**, 组成一个新数组( MDN 上找案例 ).

类比的案例: 有一个字符串数组, 将其中元素中含有 字符 e 的字符串取出来, 构成新数组

var arr = [ 'e', 'ab', 'abcde', 'abcdf' ];

// 语法:

var newArr = arr.filter(function ( v ) {

return v.indexOf( 'e' ) > -1;

});

**尝试使用自己的算法**将其功能实现出来

function filter ( arr, callback ) {

var res = [];

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

if ( callback( arr[ i ], i ) ) {

// 如果判断为真才会进入

res.push( arr[ i ] );

}

}

// 返回值? 是一个新数组

return res;

}

# 六. 函数基础

1. 定义

函数就是可以重复执行的代码块。

function 函数名（形参）{执行程序；}；（可以有名字，也可以没有名字）（匿名函数）

1. 返回值

当需要用到函数内部的值的时候 需要返回外部才能接 收到可以使用；

返回方法：return 返回值；

3.函数的注意事项

(1).函数名等于整个函数体；

(2).函数名（） ；是执行函数，最后留下的是返回值；

(3).函数不掉用不执行；

(4).加载函数的时候，只加载函数名，不加载函数体；（不调用的情况下，只加载函数名，不加载函数体）

(5).形参相当于局部变量；

(6).就近原则使用变量；

(7).两个平行的函数中变量不会相互影响；

4.作用域

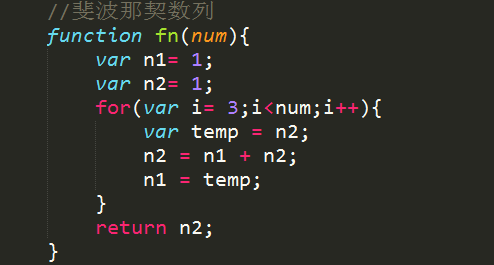
(1).全局变量

A).在外层函数外面带有var的变量

B).任何位置不带var的变量

(2).局部变量:函数内部带有var的变量。

5.斐波那契数列



6.arguments介绍

(1).作用：保存函数的实际参数

(2).arguments仅在函数内部有效。

(3).arguments是一个伪数组（即，只是与数组类似，并非一个真正的数组）

(4).因为可以使用方括号语法访问它的每一个元素（即第一个元素是 arguments[0] ，第二个元素是 argumetns[1] ，以此类推），使用 length 属性来确定参数长度。 没有传递值的命名参数将自动被赋予 undefined 值，跟定义了变量但又没有初始化一样。

# 六. 函数高级

函数是一种复杂数据类型，包括：回调函数，递归，闭包

## 1.回调函数

* + - 1. 函数作为参数传递和调用 它定义了一种规则；

## 2.递归

数学:所谓的递归就是将一个未知的问题划分为一个已经解决过的问题

在代码中的递归:函数直接或间接的调用自己就是递归

基本常识

1. 函数在执行的时候一定会分配内存空间
2. 在代码中一般函数运行开始分配内存,;函数执行结束后再释放内存
3. 如果函数在递归的时候,没有想办法跳出(不再自己调用自己)就会构成死递归,直到最后内存溢出.

首先假设算法已经写好,然后通过寻找一种递推的关系,将我么的算法转换成” 已经解决”的问题

1.首先假设你已经知道怎么调用了func()

2.通过代码的演变将其转换成当前的问题

3. 此时就已经得到递归体( 递归函数的写法 )

4.设置递归临界条件, 确定什么时候结束递归调用

不要分析递归的每一步执行过程

斐波那契数列(面试)

## 3.闭包

函数作为返回值；就是能够读取其他函数内部变量的函数。

## 4.箭头函数(ES6新增)

它的作用是为了简化函数的编写, 与函数中可能出现的 bug

箭头函数在逻辑上等价于函数表达式

**语法:**

( 参数列表 ) => { 函数体 }

使用案例:

var func = function ( v ) {

console.log( v );

};

var func = ( v ) => {

console.log( v );

};

其实用价值就是使用在回调函数中.

arr.forEach( ( v ) => {

// 大多数情况下使用在回调函数中

} )

**补充:**

1) 如果箭头函数的参数只有一个 圆括号可以省略.

2) 如果箭头函数没有参数或多个参数必须写圆括号.

3) 如果函数体只有一句话, 花括号可省略, return 也可以省略

# 七. 面向对象

## **0. 面向对象&面向过程**

-> 简单的历史( 伪历史 )

早期是面向过程

因为有了大型项目出现了面向对象

-> **面向过程**开发注重过程的含义, 也就是说需要明确每一个细节步骤, 包括顺序, 结构, 关系等.

常规的开发需要做什么**:一步一步完成怎么去做, 就是在写算法.**

-> **面向对象**就是使用对象来进行开发. 关注的是用什么对象可以实现功能, 不去关注对象中的细节.

常规的开发需要做什么**:**就是**使用对象进行开发**, 常规做法就是在**找可以完成我这件事情的对象.**

-> 面向对象是对面向过程的**封装**

## **1. 三大特性( 四大特性 )**

**这些面向对象的理论都是建立在 C++ 的基础之上**

1) **抽象性**. 在研究的具体的个体身上抽取我们需要研究的数据, 将数据组合到一起构成表征这个具体事务的数据集. 那么我们就称该数据集为 这个具体事务的抽象.

对象的抽象性就是说**不在一个具体的环境中, 就不知道这个数据集表示的是什么对象**.

2) **封装性**. 原本是用于描述将一些复杂的步骤, 操作方式, 执行过程等繁琐的内容包裹起来, 屏蔽复杂, 将简单的接口暴露出来. 那么就可以通过这个简单的接口在实现繁琐的功能.而不需要了解其内部复杂的逻辑.

常见的封装

1> 函数是对过程的封装.

2> 对象是对数据与功能封装.

…

3) **继承性**. 原本我没有, 但是别人有, 我从别人那里也得到了这个东西, 那么就说我继承它.

在代码中, 所谓的继承就是对数据进行复用.

1. **多态性**. 多态就是指多种状态, 原意是说: 一个变量调用同一个方法, 由于变量中装配的数据不一样, 那么在调用方法的时候, 得到的结果也不相同. 这个调用称为多态.

## **2.什么是对象**

-> 对象的普遍概念( 广义的概念 )

-> 程序中什么对象呢?

在传统的 oop 中( c++, java, c#, ... ), 将具有三大特性, 的能帮我实现功能的代码结构称为对象.

在 js 中, 就是键值对, 即在 js 中将**键值对**作为对象.

-> js 中键值对的抽象性: { name: 'jim', age: 3 }

-> js 中键值对的封装性:

在 js 中函数是一个特殊的数据类型, 即函数是一个 "值".

var func = function () { };

在 js 中键值对的值可以是函数, 那么即可提现封装性.

说对象是对数据与功能的封装

如果键值对的值是一个函数, 那么是对功能封装, 称这个函数为该对象的方法( method )

如果键值对的值是一个非函数的数据, 那么是对数据的封装, 称这个数据为对象的属性( property )

-> js 中键值对的继承性( 实现方法是一个难点 )

js 的对象如果为其赋值一个属性, 那么就会给他添加一个属性

如果使用 delete 对象.属性 就会将对象的属性删除

-> js 中键值对的多态性( 天生具备 )

## **3. 对象使用方式**

#### 3.1 作为数据的对象

例如: 我们要做一个学生管理系统, 每一个学生需要维护超过 10 信息: 姓名, 编号, 出生, 入学时间, 电话, 邮箱, 家庭住址, 监护人, 身份证, 性别, ...此时如果维护的数据量过大, 那么维护和使用都会非常困难 因此常常使用一个键值对来表示一个学生数据, 使用数组维护多个学生数据.

此时对象的特点是: **几乎全由数据组成没有方法**.

在有些地方, 为了**使函数的功能变得强大**, 需要考虑根据不同的参数实现不同的函数调用. 那么可以考虑**使用对象作为参数**. 例如:

jq 中的 css 方法:

1) 传入一个键值对: 两个参数, 作为设置样式的作用. $( '...' ).css( 'color', 'red' )

2) 只带有一个参数, 是一个字符串, 表示获得该样式的值: $( '...' ).css( 'color' )

3) 带有一个参数, 但是参数由键值对组成, 表示批量设置样式. 假设不使用对象作为参数.那么其功能就会收到限制. 例如我想设置元素的 color, width, height 样式.

如果不使用对象, 那么必须保证函数至少需要三个参数:

function css( color, width, height ) {

// ...

}

如果只需要设置颜色, 很简单 css( 'red' )

但是如果只设置高度, css( null, null, '300px' )

有时, 函数不需要确定参数, 但是会根据参数的不同实现不同的功能, 那么我们一般将参数写成 对象.

#### 3.2 作为方法集的对象

在实际开发过程中, 有些函数方法名字会非常的长, 在使用的时候会非常麻烦, 因此可以考虑将其 封装成一个短名字的函数. 在实际开发中封装过多的函数, 也会影响到全局作用域.

引入 jq 时候, 我们既可以使用 $ 也可以使用 jQuery. jq 早期是学习 prototype 的 js 库.prototype 库中使用的就是 $. 因此给出两个名字, 即使其中一个名字已经被覆盖, 也不会影响使用.

全局范围内名字过多, 会影响到其他的库, 名字越多, 影响的几率越大. 因此, 一般使用一个对象作容器, 将所有的工具方法定义到对象中, 那么在整个全局范围内只会添加一个名字. 大大减小冲突的可能.

var \_o = {  
 id: function ( idName ) {  
 return **document**.getElementById( idName );  
 },  
 tag: function () { },  
 **class**: ...  
 text: ...  
 attr: ...  
...  
};

#### 3.3 作为模块的对象

所谓的模块, 模块本意是说部分. 它是一类按照 "特定逻辑" 组合到一起的代码结构.实际开发的时候, 利用模块之间的组合来实现具体的逻辑

#### 3.4 作为组件的对象

所谓的 组件 就是组成的原件, 比起模块的概念 组件更加精确. 组件单只用于显示的模块.

我的理解: 组件是由组件或模块组成的用于独立完成一类特殊功能的, 在逻辑与功能上与其他代码结构独立的模块

## 4.创建对象

## 5.继承

#### 4.1对象字面量( 直接量, literal )来创建

#### 4.2对象的字面量和对象的动态特性结合创建对象

例如要创建一个对象 Stu( name, age, gender, sayHello )

var obj = {};  
obj.**name** = 'jim';  
obj.**age** = 19;  
obj.**gender** = '男';  
obj.sayHello = function () {  
 **console**.log( '你好' );  
};

**缺点: 没有办法复用**

#### 4.3.工厂函数

function factory( name, age, gender ) {  
 var o = {};  
 o.**name** = name;  
 o.**age** = age;  
 o.**gender** = gender;  
 o.sayHello = function () {  
 **console**.log( '你好' );  
 }  
 return o;  
}   
  
var obj = factory( 'jim', 19, '男' );

**注意, 一般工厂函数常常命名为 createXXX, init, ...**

**缺点:无法解决对象识别的问题(即怎样知道一个对象的类型)**

#### 4.4 构造函数

**定义方法:**

**构造函数与普通函数没有直接的区别.任何一个函数都可以作为构造函数来使用.如果将构造函数作为创建对象的方式来使用,需要的是利用构造函数来对对象进行初始化.**

**对象初始化就是给对象的各个属性第一次赋值,使用语法:this.xxx = vvv;**

**在构造函数中this特指刚刚创建出来的对象.**

**例如:**

function Person( name, age, gender ) {  
 this.**name** = name;  
 this.**age** = age;  
 this.**gender** = gender;  
 this.sayName = function(){  
 alert(this.**name**);  
 }  
}  
var person1 = new Person('Nicholas',29,'boy');  
var person2 = new Person('Rose',21,'girl');

**调用方法**

//当做构造函数使用  
var person = new Person('Nicholas',29,'boy');  
person.sayName(); //'Nicholas'  
  
//当做普通函数调用  
Person('Greg',27,'boy'); //添加到window  
**window**.sayName(); //'Greg'  
  
//在另外一个对象的作用域中调用  
var o = new Object();  
Person.call(o,'kristen',25,'girl');  
o.sayName(); //'kristen'

**构造函数的本质是利用了对象的动态特性给对象添加成员**

在创建对象的时候 使用了new构造函数,其本质是new关键字创建了对象,而对象是一个引用类型,创建对象以后将对象的引用传递到构造函数中,构造函数的默认参数this接收该引用随后利用对象动态特征给this添加成员.当构造函数执行结束,默认返回this,如果手写return后面跟的基本类型,那么return无效.

new 构造函数()

创建好处:

1> 创建出来的对象具有类型描述

2> 使用构造函数有一个默认的继承行为( 原型式继承 )

使用构造函数的语法:

1. 定义

程序中的对象，就像一个箱子，里面装满了内容

属性和方法对象里面的变量和函数就叫做属性和方法数据类型 为object；属性和方法的唯一区别在于：值是函数还是非函数。

# 八. DOM操作

document object model 文档对象模型

->dom是用来操作html( 标签->节点->字符串) html本质就是一个**字符串**

为什么用DOM???

在没有dom的时候,操作页面就是在操作字符串,非常复杂,因此有人抽象出来DOM对象,将一个HTML文档看成一个根对象,组成DOM树,从此而来一整套的DOM操作API.

页面中所有的内容都是对象:文本,标签,属性

在dom的观点中所有的对象只有两种关系,兄弟关系和父子关系,那么将其所有对象构成树的对象,因此将所有的对象称为节点.从此而来一整套的DOM操作API

->DOM-Core(核心DOM)是一个标准,通用型,所有的都可以进行操作(直接操作节点)

和 HTML DOM 在核心DOM的基础上,单独对于HTML做的简化操作的(对原始方法的封装)

1.**属性操作**

### 1.1 普通和特殊属性

id/title/src

img.**src**="image/2.jpg";

className/href/value/innerHTML/innerText

box.**className**="hide";  
a.**innerHTML**="<h1>显示</h1>";

inp.**value**="我是谁";

### 1.2 style的属性

trArr[i].**style**.**background** = "#c0c0c0";

div.**style**.**width** = "500px";  
div.**style**.**height** = "500px";  
div.**style**.**opacity** = "0.1";

this.**style**.**border** = "1px solid red";

**document**.**body**.**style**.**backgroundImage** = "url(image/"+(this.**index**+1)+".jpg)";

div.**style**.**position** = "absolute";  
div.**style**.**left** = "-99999px";

this.**style**.**zIndex** = 1000;//设置层级

特点:

//1.样式少的时候使用  
//2.style是对象  
//3.值是字符串,没有设置是""  
//4.命名规则,驼峰命名,和css不一样  
//5.设置了类样式不能够获取(只和行内式交互,和内嵌和外链无关)  
//6.box.style.cssText = "字符串形式的样式";

### 1.3 方法形式的属性

1.**dom对象.属性===dom对象[属性]**

2.**get/set/removeAttribute();**

**相同点:** **都可以操作dom元素的已有元素,获取值,赋值,删除值都可以;**

<div id="box" title="aaa">你好</div>  
 //相同点:   
 var box = **document**.getElementById("box");  
  
 **console**.log(box["title"]);//获取值  
 **console**.log(box.**title**);//获取值  
 **console**.log(box.getAttribute("title"));//获取值  
  
 box.**title** = "bbb";//设置值  
 box.setAttribute("title","ccc");//设置值  
 **console**.log(box.getAttribute("title"));//获取值 ccc  
  
 box.**title** = ""; //清空索引值  
 box.removeAttribute("title");//删除属性

**不同点:**

//1.特殊属性的操作: className/class; innerHtml  
//2.自定义属性:方法的形式设置的自定义属性会出现在标签中,而另一个不会;  
//3.自定义属性:怎么绑定的值就怎么获取,不能交叉设置和获取  
  
 box.setAttribute("aaa","nihao");//能出现在标签中  
 box.**bbb** = "我是bbb的值";//不出现在标签中  
 **console**.log(box.getAttribute("aaa"));  
 **console**.log(box.**bbb**);  
  
//自定义属性:在对象上面挂一个变量;  
 var obj = new Object();  
 obj.**name** = "张三";

## **2.访问关系**

1. .父节点。节点.parentNode(); 兼容性好
2. .兄弟节点。 兼容性不好

nextElementSibling();在火狐谷歌IE9都指的是下一个元素节点。 IE678不支持；

nextSibling(); IE678中指下一个元素节点（标签），在火狐谷歌IE9+以后都指的是下一个节点（包括空文档和换行节点）。

previousElementSibling();------------------------兼容同上；

previousSibling();

1. .单个子节点 通过自己找单个子节点 。兼容性不好

firstElementChild()；在火狐谷歌IE9+以后都指的是第一个节点（包括空文档和换行节点）。

firstChild(); IE678中指第一个子元素节点（标签）。

lastElementChild()；——————兼容同上；

lastChild()；

1. .所有子节点，通过自己着所有子节点 兼容性不好

childNodes；w3c标准，它返回指定元素的子元素集合，包括HTML节点，所有属性，文本节点；火狐 谷歌等高本版会把换行也看做是子节点

children；非标准属性，它返回指定元素的子元素集合。

但它只返回HTML节点，甚至不返回文本节点，虽然不是标准的DOM属性，但它和innerHTML方法一样，得到了几乎所有浏览器的支持。

注意：children在IE6/7/8中包含注释节点 ，在IE678中，注释节点不要写在里面。

节点自己.parentNode.children[index];随意得到兄弟节点。

function siblings(elm) {

var a = [];

var p = elm.parentNode.children;

for(var i =0,pl= p.length;i<pl;i++) {

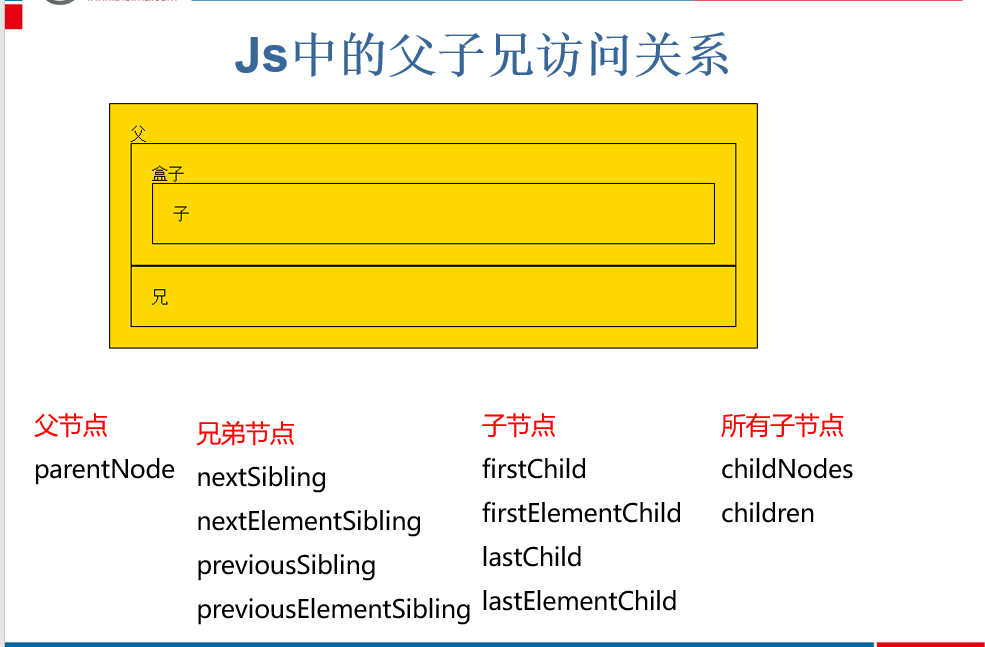
if(p[i] !== elm) a.push(p[i]);

}

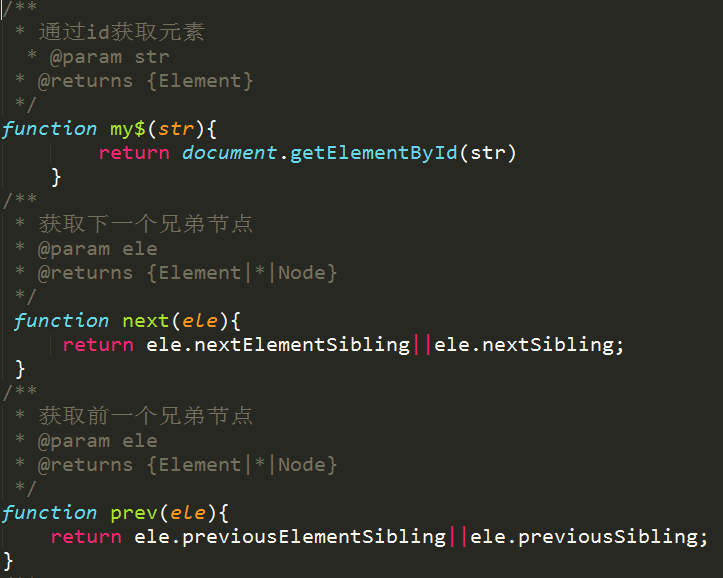
return a;

}

定义一个函数,必须**传递自己**,再定义一个数组，获得自己的**父亲**，再获得自己父亲的所有儿子（包括自己）。遍历所有的儿子，只要**不是自己**就放进数组中。



**访问关系的封装**







1. **节点操作**

### 3.1 创建

a).document.write();识别标签。

缺点：容易覆盖页面内容，所以几乎没人用

b).innerHTML 识别标签

优点：未来经常使用，1.属性添加特别方便 2.可以穿件多个嵌套标签

缺点：老元素事件容易被清楚

c).document.createElement("");专门用来穿件标签用的

优点:不影响老元素事件

缺点 属性绑定有点啰嗦，标签嵌套也啰嗦；

**追加节点**

1. .父节点.appendChild(子节点)；在父节点最后添加，
2. .父节点.insetBefore(新节点，参考节点)；在参考节点之前添加；

### 3.2 删除

父节点.removeChild(子节点);

不知道父级的情况下，可以这么写：node.parentNode.removeChild(node)

### 3.3 复制

节点.cloneNode(bollean);

如果是true；深层复制（自己和子代）

如果是false 浅层复制 （自己）

注意 **复制节点 不能复制事件**

### 3.4 清空

father.innerHTML=””;在清空内容的时候和事件相关的函数不会被销毁，任然遗留在内存中

因此如果方便的话，建议使用father.removeChild(son)来清除

1. **设置内容**

### innerHTML

用途: 设置和获取的内容格式为是**标签+文本**

box1.innerHTML = "<h1>我是innerHTML赋值的内容</h1>";

字符串拼接:

插入tr先创建tr(document.creatElement),再用innerHTML写其里面的内容:

//拼接tr标签  
var tr = **document**.createElement('tr');  
tr.**innerHTML** =  
 '<td>' + lessonValue + '</td>' +  
 '<td>' + schoolValue + '</td>' +  
 '<td><a href="javascript:;" class="get">GET</a></td >';  
**console**.log(tr);  
//插入  
tbody.appendChild(tr);

### innerText

innerText只能设置和获取元素中的文本内容

# 九. BOM（浏览器对象模型）

1. Window介绍

Window是整个浏览器；document是这个浏览器页面中的一个最大的节点，window是Javascript中顶级对象，里面包含着属性和方法，比如变量是window的属性，最外层函数是window的方法，但平时window可以省略；

1. window中的两个方法

Window.open();

1. .变量= window.open("http://www.jd.com","\_blank","width = 200,height = 200,top = 200")

在新页面中打开一个新窗口

设置新窗口的大小和位置

变量 . moveto(x,y);新窗口移动到距离屏幕距离；

变量.moveBy(x,y)；在原有基础上添加或者时减少多少

变量.resizeTo(x,y);新窗口大小改变为多少

变量.resizeBy(x,y);在原有基础上添加或者减少

window.close关闭新窗口；

window.close(变量)；

变量.close();

1. window的四大属性
2. .location 地址
3. .location.href 地址，页面；

Location.assign() 改变浏览器地址栏的地址，并记录到历史中,可以点击后退返回原页面；

Location.href() 实际开发使用最多的页面跳转,改变浏览器地址栏地址，并记录到历史中

Loaction.relace() 替换浏览器的之蓝的地址不会记录到历史中,并且无法返回；

B).location.hash; 返回url中#后面的内容；

C).location.host; 主机名 包括端口；

D).location.hostname; 主机名；

E).location.pathname; url中的历经部分；

F).location.protocol; 协议，一般是http https；

G).location.search; 查询字符串；

H).reload 刷新页面；

1. Navigation 和浏览器有关

判断浏览器(记忆)



1. Histroy 历史记录
2. Screen 暂时不学习。
3. 定时器

两种定时器：

1. .间歇定时器：setInterval(函数，间隔事件)；
2. 延时定时器：setTimeout(函数，倒计事件)；
3. 注意：要用定时器先清定时器；
4. 清楚定时器 clearInterval(定时器变量名);
5. 时间对象Date；

Date是日期。Data是数据。

获取时间：

var date = new Date();

### 推荐方法

1. var date1 = new Date(); 本地时间。
2. 设定制定时间：（兼容最强）

var date2 = new Date("2016/01/27 12:00:00")

### 不推荐的方法

var date3 = new Date('Wed Jan 27 2016 12:00:00 GMT+0800 (中国标准时间)');

var date4 = new Date(2016, 1, 27);

(2).获取时间和日期方法；

getDate() 获取日 1-31

getDay () 获取星期 0-6（0代表周日）

getMonth () 获取月 0-11（1月从0开始）

getFullYear () 获取完整年份（浏览器都支持）

getHours () 获取小时 0-23

getMinutes () 获取分钟 0-59

getSeconds () 获取秒 0-59

getMilliseconds () 获取毫秒 （1s = 1000ms）

getTime () 返回累计毫秒数(从1970/1/1午夜)

# 十. 四大事件对象（offset、scroll、client、event）

1. offset家族
2. .offsetWidth/offsetHeight;检测盒子的宽高；

offsetWidth = width+padding+border;

offsetHight = hight+padding+border;

返回值为不带单位的num类型值；

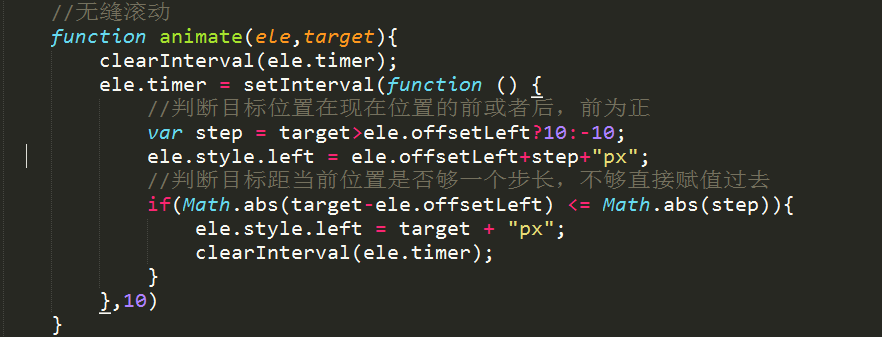
1. .offsetLeft/offsetTop;

检测盒子距带有定位的父系盒子的距离，没有以浏览器页面为准；

返回值为不带单位的num类型值；

1. .offsetParent; 检测盒子带有定位的父系盒子，没有以body为基准；
2. .无缝滚动轮播图；

函数封装（最终版）；



目前无缝滚动封装还不够完美，因为offset 取值为四舍五入取值 完善需要用到自己封装的 getStyle(),方法 去后面缓动框架去找，替换掉 ele.offsetLeft 属性;

切记：getStyle() 方法获取的为带有单位的字符串，需要先取整；

1. scroll家族
2. .scrollTop/scrollLeft

定义:检测盒子被卷曲的部分,调用者是body/html或者盒子;

如果是body/html 但用 那么代表 页面被卷去的头部或者左侧部分;

如果是盒子调用,那么代表,该盒子遮挡住子盒子左侧和顶部的部分.

1. .scrollWidth/scrollHeight

定义:scollWidth 和scrollHeight 都是检测盒子宽高的意思;(用的不多,兼容性不好,且收到内容影响);

scrollWidth/scrollHeight = width/height + padding

缺点: 火狐谷歌 ie9+

如果盒子内容超出盒子,以内容为基准;不超出以盒子为基准;

缺点:ie 678

永远一内容为基准(而且 受到padding 的影响).

1. .scrollTop/scrollLeft(兼容问题);

Scroll家族用的最多的是网页被遮挡的顶部或左侧部分,如果是body/html调用,scrollTop/scrollLeft会出现兼容问题

1. .不同的浏览器兼容不同
2. .怪异浏览器:google,支持document.body.scrollTop/Left;
3. .正常浏览器支持:

ie678 只支持这种方式

Document.documentElement.scrollTop/LEft;

B).正常浏览器

1. .有DTD约束

Document.documentElement.scrollTop/Left;

b).没有DTD约束

Document.body.srollTop/Left;

C).w3c在这种情况下做出的标准

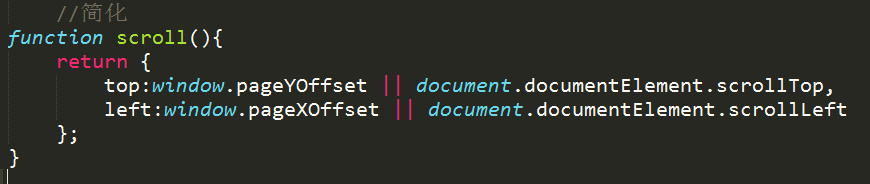
不管有没有DTD约束都要支持(ie678 不支持)

Window.pageXoffset;

Window.pageYoffset;

提示: 关于为什么好多w3c给定的标准 ie 678 不支持,是因为在w3c标准推行时候,微软公司已经放弃了 ie 678,造成很多的属性或方法不被支持,但目前国内许多老牌公司仍然使用着window 98/xp/2000等系统,更换成本大,就需要程序员写出的代码需要兼容 ie 678.

Scroll 的兼容写法;



1. .getStyle()方法

到目前为止,因为没有一个合适的方法或属性能获得一个准确的元素的位置值,所以引入新的方法

Window.getComputedStyle(变量,null).width;

Window.getComputedStyle(变量,null).height;

Window.getComputedStyle(变量,null).left;

Window.getComputedStyle(变量,null).top;

......

获取元素最高属性的值,带有单位,支持火狐,谷歌,ie9+;

元素.currentStyle.width;

元素.currentStyle.height;

元素.currentStyle.left;

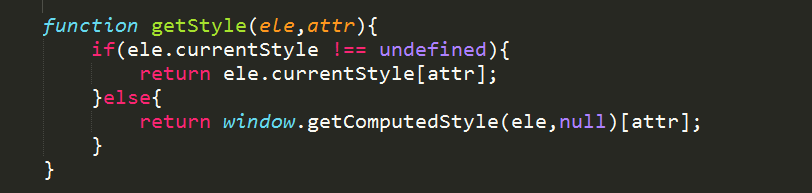
元素.currentStyle.top;

......

获取元素最高属性值,带有单位,支持 ie 678;

封装一个getStyle 方法,兼容ie 678;

(见下图....)



1. .缓动动画 框架封装



1. client家族
2. .clientWidth/clientHeight
3. .调用者是一个dom元素,检测该元素的的宽高;
4. .调用者body/html,浏览器可视区域的宽高;
5. .clientHeight/clientWidth = height+padding;
6. .clientTop/clientLeft

获取元素上边框和左边框;

1. .clientWidth/Height 兼容写法

获取可视区域的宽和高 受到DTD约束的影响

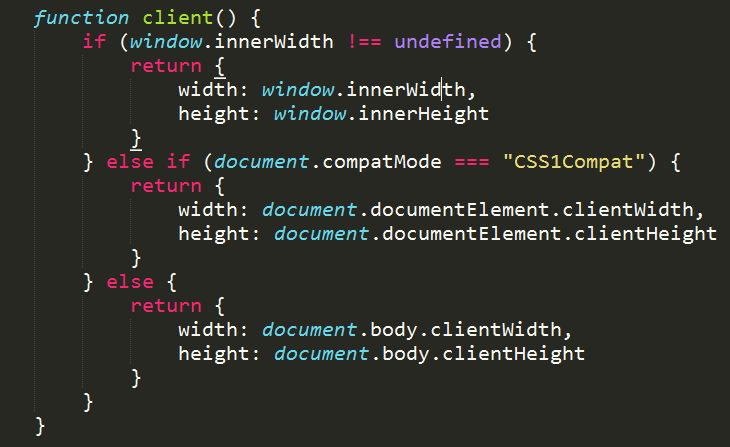
有约束: document.documentElement.clientWidth;

无约束: document.body.clientWidth;

w3c给的新标准;但ie 678 不支持

window.innerWidth/innerWidth;

引入:新方法 检测页面有没有DTD约束 兼容ie 678;



1. .检测屏幕大小

window.screen.width;

window.screen.height;