**04/05**

1. 弱类型语言：没有明确的数据类型声明，所有变量都用var声明。
2. 脚本语言：计算机可以直接执行源代码。
3. JavaScript是一种基于对象(Object)和事件驱动(Event Driven)并具有安全性能的脚本语言,可广泛用于服务器、PC、笔记本电脑、平板电脑和智能手机等设备。
4. JavaScript——最流行的脚本语言:弱类型语言、动态性语言、解释性语言。
5. JavaScript 组成：ECMAScript（语法）、DOM（文档对象模型）、BOM（浏览器对象模型）。 BOM（Browser Object Model）
6. document.write("Hello JavaScript1<br/>");

//文档输出时换行的第一种写法。

document.write("Hello JavaScript2"+"<br />");

//文档输出时换行的第二种写法。

document.write**ln**("Hello JavaScript2");

//文档输出时文本后加一个空格。

alert("Hello**\n**JavaScript");

//弹出对话框时换行的第一种方法。

alert("Hello"+"\n"+"JavaScript");

//弹出对话框时换行的第二种方法。

console.log("Hello\nJavaScript");

//控制台输出文本换行的第一种写法。

console.log("Hello"+"\n"+"JavaScript");

//控制台输出文本换行的第二种写法。

注意：①window是一个全局对象，在书写时一般省略。例如：window.document；window.alert；window.console②在Hbuild中的<script>标签内，字体颜色为绿色的为对象，字体颜色为黑色的为方法。例如：document.write()中，document是一个文档对象，write是一个方法。

1. 加号（+）有两种含义：①算数运算符；②字符串拼接；

当相加的各数据全部为数字数据类型时，加号的含义为算数运算符。

当相加的各数据有字符串数据类型时，加号的含义在第一个字符串之前为算数运算符，在第一个字符串之后为字符串拼接（括号中的数字为加法）。

例子： var a=2+3+15+"12"+4+"abc"+(2+3)+9+1;

alert(a);输出：20124abc591;

注意：字符串数据类型必须用双引号包裹。

1. <script>标签可以放在文档的任何位置，通常放在<head>中，建议写在</body>之前，使浏览器先加载标签，再运行JavaScript代码，增强用户体验。
2. <script>标签必须成对出现，同一个页面中可以出现多对，且从上往下执行。
3. 如果一个<script>标签用来引入外部JavaScript文件，此标签内不能再写JavaScript代码。
4. <script>标签的type属性可以不写，也可以写text/javascript属性值，这种写法不标准，但是最通用，浏览器支持较好，也可以写application/javascript属性值，这种写法浏览器支持不是特别好。
5. <script>标签的async用来设置脚本是否支持异步，默认属性值为“false”，可定义“true”或“async”。

给外链JavaScript文件的<script>标签添加async属性，使下一个<script>标签的起始执行时间无需等待当前<script>标签的外链JavaScript执行结束。

1. 标识符：指变量、函数、属性的名字，或者函数的参数。
2. JavaScript的变量用来保存值或者表达式，变量的值是可以改变。

变量就是计算机在内存中申请一块空间地址的别名，便于程序员识别。

变量三要素：①变量名 ②变量值 ③变量类型

（1）当一个变量已声明未赋值时，变量的值为undefined。

（2）变量可以先声明后赋值，也可以声明和赋值同时进行。var a;a=10;

（3）可以同时声明多个变量，变量之间用逗号隔开。var a=10,b=37,c=53;

（4）所有变量都用var声明，没有严格区分数据类型，属于弱类型语言。

注意：当声明一个变量时，如果没有给他赋值，无法明确知道变量的类型，这样的语言称作弱类型语言。

JavaScript是一种弱类型的语言，把所有的变量类型都用var声明，而不用明确声明数据类型，只有真正给变量赋值时，变量的类型才决定，也就是说变量的数据类型是由变量的值的类型来决定的。

1. 用typeof 来确定变量的数据类型。

var a=1;alert(typeof a);//等价于alert(typeof(a));

1. 语法中可以省略var。
2. 数据类型

ECMAScript中有5种基本数据类型：undefined、null、boolean、number 、string,还有一种复杂数据类型。也叫引用数据类型object和函数类型function，当然，function也可以看做object类型。ECMAScript 不支持任何创建自定义类型的机制，所有值都成为以上**7 种数据类型**之一。

在JavaScript中，用typeof()来检测变量的数据类型。对于值或变量使用typeof 操作符会返回如下字符串：

undefined： 未定义

boolean ：布尔值

string： 字符串

number ：数值

object： 对象或null

function ：函数

基本数据类型：数字、字符串、布尔、未定义

var a=1; alert(typeof(a)); //返回number 数字

var a="1"; alert(typeof a); //返回string 字符串

var a; alert(typeof (a)) //返回undefined 未定义

var a=false; alert(typeof(a)); //返回boolean 布尔

引用数据类型：函数、数组、对象、日期、**空**

var a=null; alert(typeof(a)); //返回object 空

var a=function (){}; alert(typeof a); //返回function（属于object的一种）

var a=new Array ; alert(typeof a); //返回object 数组

var a=new Object(); alert(typeof a); //返回object 对象

1. 多行注释快捷键：ctrl+shrit+/ 单行注释快捷键：ctrl+/
2. 关键字在JavaScript语法中有特定用途，不能用作标识符，保留字未来可能成为关键字。

**04/06**

1. Bit 位，Byte字节

Bit 位：二进制系统中，每个0或1就是一个位（bit），位（bit）是内存的最小单位。

Byte字节：字节由8个位所组成，是内存储数据的基本单位。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 二进制 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 |
| 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| 13 |  |  |  |  | 1 | 1 |  | 0 |

一个字节可以存储的最大十进制数是多少：一个字节由8个位，每个位中存放一个二进制数，即非0即1，当8bit全部都是1时，就是二进制数11111111，转换为十进制即是：

1\*2^7+1\*2^6+1\*2^5+1\*2^4+1\*2^3+1\*2^2+1\*2^1+1\*2^0=255。

Byte数据类型（字节型）用一个字节（Byte）存储，可区分256个数字，取值范围为0到255。

1B(byte)=8bit；(一个字节由八个位组成)

1KB=1024B(byte)=210B(byte)；

1MB=1024KB=220B(byte)；

1GB=1024MB=230B(byte)；

UTF-8编码：一个英文字符占一个字节，一个中文占三个字节。英文标点占一个字节，中文标点占两个字节。

1. 表达式由一个或多个操作数通过操作符组合而成，就是由数据和运算符组成的一个式子。例如：x+y；x/10；x+23/2+y；x%y等都是表达式。
2. 赋值运算符的含义就是把赋值运算符的右边的数值或表达式的值赋给赋值运算符的左边的变量或表达式。
3. JavaScript中隐式转化

alert(3+true); //输出4，true转化为1

alert(3+false);//输出3，false转化为0

alert(3+null);//输出3，null转化为0

alert(3+new Array());//输出3，new Array()转化为0

alert(3+undefined); //输出NaN

alert(3-"2")//输出1，字符串"2"转化为数字2

alert(100-"23par")//输出NaN

alert(parseInt("23par"))//输出23 parse:解析

注意：parseInt()将字符串转化为数字。

5. var A="A";

var a="a";

alert(A.charCodeAt()); //弹出65

//注意：charCode字符码，字符编码。

alert(a.charCodeAt()); //弹出97

alert(String.fromCharCode(36));

ASICC：美国信息交互标准代码：a 97；A 65；0 48；空格 32

6. JavaScript中的隐式转化

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 数据类型 | | Number | NaN | 纯数字符串 | 其他字符串 | true | false | null | undefined |
| 算数运算符 | 加(+) | Number | NaN | 字符串 | 字符串 | 1 | 0 | 0 | NaN |
| 减(-) | Number | NaN | Number | NaN | 1 | 0 | 0 | NaN |
| 乘(\*) | Number | NaN | Number | NaN | 1 | 0 | 0 | NaN |
| 除(/) | Number | NaN | Number | NaN | 1 | 0 | 0 | NaN |
| 余(%) | Number | NaN | Number | NaN | 1 | 0 | 0 | NaN |
| 一元运算符 | ++ | Number | NaN | Number | NaN | 1 | 0 | 0 | NaN |
| -- | Number | NaN | Number | NaN | 1 | 0 | 0 | NaN |

举例： <script>

var a="";

a=a+1; //由于赋值运算符的右边的a为string。

alert(a); //所以弹出string数据类型1。

</script>

<script>

var a;

a=a+1; //由于赋值运算符的右边的a为undefined

alert(a); //所以弹出NaN

</script>

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 关系运算符  >、<、>=、<=、==、===、!=、!== | 比较 | Number | NaN | 纯数字符串 | 其他字符串 | true | false | null | undefined |
| Number | Num | false | Num | false | Num | Num | Num | false |
| NaN |  | false | false | false | false | false | false | false |
| 纯数字符串 |  |  | **首字符** | **首字符** | Num | Num | Num | false |
| 其他字符串 |  |  |  | **首字符** | false | false | false | false |
| true |  |  |  |  | Num | Num | **Num** | false |
| false |  |  |  |  |  | Num | **false** | false |
| null |  |  |  |  |  |  | Num | Num |
| undefined |  |  |  |  |  |  |  | false |

注意：①null可以与Number数据类型中除0以外的任何数值比较，null隐式转化为数值0。null不可与Number数据类型中的数值0比较，结果总为false。true和false可以与Number数据类型中的任何数值（包括数值0）比较，比较时true隐式转化为1，false隐式转化为0。②这里的true和false不仅可以是比较运算符的结果，还可以是比较运算符的表达式。这里的undefined不仅可以是undefined数据类型，还可以是未被定义的数据变量var a;。③这里的纯数字符串是指字符串中只有数字，包括””。

例子： <script>

alert(true<2);

//在关系运算符中，true转化为num（数值1），1<2返回true

alert((100>1)<2); //100>1为true，true<2正确，返回true。

</script>

<script>

alert(null<2); //弹出true

</script>

<script>

alert(undefined<2); //弹出false

var a; //a为未定义**undefined**数据类型

alert(a<2); //弹出false

</script>

**04/07**

1. prompt(“信息提示”,”默认值”)。 prompt：提示

注意：prompt(“信息提示”,”默认值”)，即使输入数字，提交的也是string数据类型。

1. switch条件语句中的条件表达式(case 值:)默认的判断过程是全等（===）判断，即认为”3”不全等于3。（在if else条件语句中，==不识别数据类型，===识别数据类型。）

例子：<script>

var a="2"

switch(a){

case 2: //不满足该条件表达式

alert("a"+"==="+2); //不执行，不弹出

break;

default:

alert("a"+"!=="+2) //弹出a！==2

}

</script>

3. if else条件语句从前向后执行，当满足第一个条件表达式时，执行对应的条件语句，之后跳出if else条件语句。此时即使后面仍有满足条件的表达式，也不再执行相对应的条件语句。

4. switch条件语句从前向后执行，break语句用来跳出当前switch语句。如果满足一个条件表达(case 值:)式的条件语句后没有break语句，则执行该条件语句和之后第一个break语句之前的所有条件语句。

1. 求1！+2！+3！+……+10！

<script>

var n=1 ;

var num= **1**;

var sum= 0;

while(n<=**10**){

num=**num\*n**;

sum=sum+num;

n++;

}

alert(sum)

</script>

**注意：写循环体时最重要的是明确每一个变量所代表的含义（在Hbuild中标注每一个变量的含义）。**

求和循环核心三变量：循环次数n、各项数值num、各项总和sum。

格式： **for循环**

<script>

var num=初值;

var sum=0;

for (n=1;n<=循环次数;n++) {

num=表达式;

sum=sum+num;

}

alert(sum)

</script>

**过程：①项数赋值循环次数。②求解各项数值表达式。③初值赋值。**

**while循环**

<script>

var n=1 ;

var num=初值;

var sum= 0;

while(n<=循环次数){

num=表达式;

sum=sum+num;

n++;

}

alert(sum)

</script>

**过程：①项数赋值循环次数。②求解各项数值表达式。③初值赋值。**

**do while循环**

<script>

var n=1;

var num= 初值;

var sum=0;

do{

num=表达式;

sum=sum+num;

n=n+1;

}while(n<=循环次数)

alert(sum)

</script>

**过程：①项数赋值循环次数。②求解各项数值表达式。③初值赋值。**

1. 求和 sum=2+22+222+22222+.....+22222222

<script>

var n = 1;

var num = **0**;

var sum = 0;

while (n <= **8**) {

num = **num\*10+2**;

sum = sum + num;

n++;

}

alert(sum)

</script>

过程：①循环次数赋值。②求解各项数值表达式。③初值赋值。

1. 求和 sum=1/1!+1/2!+1/3!+1/4!+1/5!+.....+1/10!

<script>

var n = 1;

var num = **1**;

var sum = 0;

while (n <=**10** ) {

num =**1/((1/num)\*n)**;

sum = sum + num;

n++;

}

alert(sum)

</script>

过程：①循环次数赋值。②求解各项数值表达式。③初值赋值。

1. 有一只猴子吃一堆果子，已知每天吃昨天剩下的果子的一半再多一个，直到第10天才吃完。问原来这堆果子有多少个？

<script>

var n = 1;

var num = **0**;

var sum = 0;

while (n <= **10**) {

num =**2\*num+2**;

sum = sum + num;

n++;

}

alert(**num**)

</script>

过程：①循环次数赋值。②求解各项数值表达式。③初值赋值。

1. 国际象棋棋盘有64格，若在第1格放1粒谷；第2格放2粒谷；第3格放4粒谷；第4格放8粒谷……如此一直放到第64格。假设2000000粒谷有一顿重，问需要多少吨谷才够放？

<script>

var n = 1;

var num = **0.5**;

var sum = 0;

while (n <= **64**) {

num =**num\*2;**

sum = sum + num;

n++;

}

alert(sum)

</script>

过程：①循环次数赋值。②求解各项数值表达式。③初值赋值。

10. 求出1-100之间所有除以3余1的数,并求和

<script>

var n=1;

var num=**1**;

var sum=0;

while (n<=**100**){

num=**n**;

if(num%3==1){

sum=sum+num;

}

n++;

}

alert(sum);

</script>

过程：①循环次数赋值。②求解各项数值表达式。③初值赋值。

**04/08**

1. 输入任何一个正整数，反向输出各位上的数字，比如输入654321，输出123456。

<script>

var num=prompt("请输入","");

do{

document.write(num%10);

num=Math.floor(num/10);

}

while(num>=1);

</script>

1. 求出1到100之间所有的质数。

<script>

for (var i=2;i<=1000;i++) {

var mask=true;

for (var j=2;j<i;j++) {

if(i%j==0){

mask=false;

}

}

if(mask==true){

document.write(i+"<br />")

}

}

</script>

3. 数组

<script>

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

alert(arr[1]);

//数组下标值的排序从0开始，第二个元素的底为1，弹出元素2。

alert(arr.length);

//数组的长度为数组元素的个数，比数组元素下标的最大值大1，弹出5。

arr.splice(1,2);

//表示删去第一个元素之后的两个元素，删去2,3。

alert(arr);

//弹出数组[1,4,5]

</script>

4. 字符串(String)常用属性和方法

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 属性/方法 | | 含义 | 返回值 | 原字符串 |
| 属性 | 字符串.length | 获取字符串长度 | 字符串长度 | 不变 |
| 获取类方法 | 字符串.charAt  (下标值) | 获取指定  位置字符 | 指定位置字符 | 不变 |
| 字符串.charCodeAt  (下标值) | 获取指定位置字符的ASCII码 | 指定位置  字符的ASCII码 | 不变 |
| **String**.formCharCodeAt(ASCII码) | 获取ASCII码  转化而成的字符  (**静态方法**) | 字符 | 不变 |
| 截取类 | 字符串.slice(起始底,起始底+截个数) | 截取字符串 | 截取的字符串 | 不变 |
| 查找类方法 | 字符串.indexOf  (“待索引字符串”) | 字符串中  索引字符串 | 若存在,返回**第一个位置**;否则,返回-1 | 不变 |
| 字符串.lastIndexOf  (“待索引字符串”) | 字符串中  索引字符串 | 若存在,返回**最后一个位置**;否则,返回-1 | 不变 |
| 字符串.search(“待索引字符串”) | 字符串中索引字符串(**适用于正则表达式**) | 若存在,返回**第一个位置**;否则,返回-1 | 不变 |
| 字符串.match(“待匹配字符串”) | 字符串中  匹配字符串 | 若存在,返回**匹配字符串**;否则,返回null | 不变 |
| 字符串.startsWith  (“待匹配字符串”) | 字符串中  匹配字符串 | 若字符串以匹配字符串开头，则返回true | 不变 |
| 字符串.endsWith  (“待匹配字符串”) | 字符串中  匹配字符串 | 若字符串以匹配字符串结尾，则返回true | 不变 |
| 转化类方法 | 字符串.replace(“**原**字符串”,“**新**字符串”) | 字符串中  替换字符串 | 替换**第一个**  **“原字符串”** | 不变 |
| 字符串.toUpperCase()  /toLowerCase() | 字符串  大小写转换 | 大小写转换  之后的字符串 | 不变 |
| 字符串.split  (“分隔符”) | 将字符串以分隔符分隔成为数组 | 数组 | 不变 |
| 比较类方法 | 字符串1  .localeCompare  (字符串2) | 按照本地习惯(拼音)比较 | 若字符串1位置靠后,返回1;若相同,返回0;若靠前,返回-1 | 不变 |

注意：①字符串的所有属性和方法，都无法改变原字符串。②任何一个查找类方法最多返回一个索引/匹配项位置或索引/匹配项;replace方法只会替换第一个“原字符串”。

**04/09**

1. 随机产生一个大于0小于1的数值(0<x<1)方法为：Math.random()

随机产生一个大于0小于n的数值(0<x<n)方法为：Math.random()\*n

随机产生一个大于0小于n+1的数值(0<x<n+1)方法为：

Math.random()\*(n+1)

随机产生一个从0到n-1的整数(0<=x<=n-1)方法为：

Math.floor(Math.random()\*n)

Math.round(Math.random()\*(n-1))

随机产生一个从0到n的整数(0<=x<=n)方法为：

Math.floor(Math.random()\*(n+1))

**Math.round(Math.random()\*n)**

1. 数组的属性：length属性表示数组的长度，即其中元素的个数。
2. 数组的常用方法：

（1）concat()连结两个或更多的数组 concatenate连结

语法：数组1.concat(数组2,数组3,……数组n)

返回值：连结后的新数组

<script>

var arr1=[1,"2",4];

var arr2=[5,new Object,6]

var arr3=arr1.concat(arr2)

document.write(arr3+"<br />")

//页面输出返回值1,2,4,5,[object Object],6

</script>

（2）join()将数组的所有元素放入一个字符串，元素通过指定的字符串进行分隔。（即是将数组转化为一个字符串，数组元素之间可设置分隔符）。

语法：数组.join(“分隔符”)，若不设置分隔符join()，默认分隔符为逗号。

返回值：生成的字符串

<script>

var arr=[1,"2",4,5,new Object,6]

var arr1=arr.join("")

document.write(arr1)//页面输出新字符串及返回值 //1245[object Object]6，无空格。

document.write(typeof arr) //页面输出Object

</script>

（3）pop()删除并返回数组最后一个元素

语法：数组.pop() 删除数组最后一个元素，并返回数组最后一个元素

返回值：最后一个元素

<script>

var arr = [1, "2", 4, 5, new Object, 6]

var arr1 = arr.pop()

document.write(arr1+"<br/>") //页面输出返回值6。

document.write(arr)

//页面输出新数组1,2,4,5,[object Object]。

</script>

（4）push()向数组的末尾添加一个或更多的元素，并返回新的长度

语法：数组.push(元素1，元素2，……)

返回值：新的长度

例子： <script>

var arr=[1,"2",4,5,new Object,6]

var arr1=arr.push("12",new Object,3)

document.write(arr1+"<br/>") //页面输出返回值9。

document.write(arr)

//页面输出新数组1,2,4,5,[object Object],6,12,[object Object],3

</script>

含义：arr添加了三个元素，而arr1为数值9。

（5）reverse()颠倒数组中元素的顺序

语法：数组.reverse()

返回值：颠倒后的数组

例子：<script>

var arr=[1,"2",4,5,new Object,6]

arr.reverse();

document.write(arr)

//页面输出6,[object Object],5,4,2,1

</script>

注意：数组.reverse()语法正确，而字符串.reverse()语法错误。

注解：字符串转化为数组的方法为：字符串.split(“分隔符”)

例子：<script>

var str=window.prompt("请输入数值")

//prompt提示框输入字符串str。

var arr=str.split(""); //字符串str转化为数值arr。

arr.reverse(); //颠倒数组arr元素中的顺序。

var str2=arr.join("") //将数组arr转化为字符串str2，并且没有分隔符。

document.write(str+"<br />"); //页面输出字符串str：546285753

document.write(arr+"<br />"); //页面输出颠倒顺序后的数组arr：3,5,7,5,8,2,6,4,5

document.write(str2) //页面输出字符串str2：357582645

</script>

（6）shift()删除并返回数组的第一个元素

语法：数组.shift() 删除数组第一个元素。

返回值：删除的第一个元素。

<script>

var arr = ["a", "2", 4, 5, new Object, 6]

var arr1 = arr.shift()

document.write(arr1+"<br/>") //页面输出返回值a。

document.write(arr) //页面输出新数组2,4,5,[object Object],6。

</script>

（7）slice()从某个指定的数组返回选定的元素，不会修改数组。

语法：数组.slice(选择元素的起始底，选择元素的结束次序)

返回值：选择的元素。

<script>

var arr = ["a", "2", 4, 5, new Object, 6]

var arr1 = arr.slice(2,5)

document.write(arr1+"<br/>") //页面输出返回值4, 5, new Object

document.write(arr) //**页面输出a,4,5,[object Object],6。**

</script>

注意：该方法并不会修改数组。

（8）sort()对数组的元素进行排序

语法：数组.sort()

返回值：排序之后的数组。

<script>

var arr = [21,11,-4,9,45,6]

arr.sort()

document.write(arr+"<br />")

//页面输出-4,11,21,45,6,9

var t=function (a,b){

return a-b

};

arr.sort(t);

document.write(arr) //页面输出-4,6,9,11,21,45

</script>

注意: **return a-b不可以替换为return a>b。因为a-b有三类结果:正数、负数和0，而a>b只有两个结果1和0**。

（9）splice()从数组中添加/删除元素，然后返回被删除的元素。

语法：数组.splice(删除元素的起始底，删除元素的个数，增加的元素1，增加的元素2……)

返回值：被删除的元素。

<script>

var arr = ["a", "2", 4, 5, new Object, 6]

var arr1 = arr.splice(2,1,"bc",3)

document.write(arr1+"<br />")//页面输出4

document.write(arr) //页面输出a,2,bc,3,5,[object Object],6。

</script>

（10）tostring()将数组转化为字符串

语法：数组.toString()

返回值：字符串

<script>

var arr = ["a","2",new Object,6]

arr.toString()

document.write(arr) //页面输出a,2,[object Object],6

</script>

（11）unshift()向数组开头添加一个或更多元素，并返回新的长度

语法：数组.unshift(“元素1”,”元素2”,”元素3”)

返回值：新的长度

<script>

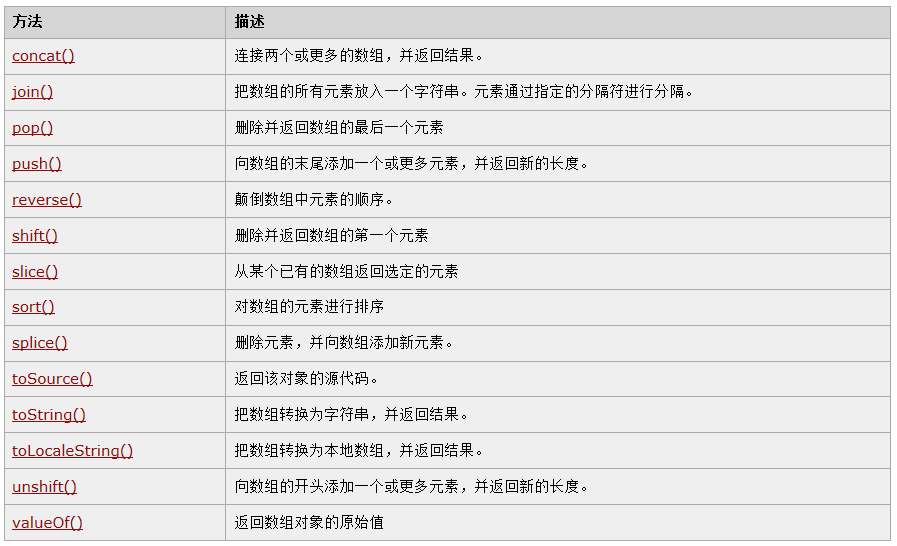
var arr = ["a","2",new Object,6]

var arr1=arr.unshift(1,"bc")

document.write(arr1+"<br />")//页面输出6

document.write(arr) //页面输出1,bc,a,2,[object Object],6

</script>



|  |  |  |  |
| --- | --- | --- | --- |
| 方法 | 含义 | **返回值** | **原数组** |
| 数组.concat(数组1,数组2，...) | 合并数组 | 新数组 | 不变 |
| 数组.reverse() | 数组反转 | 反转之后的数组 | 改变 |
| 数组.join(“分隔符”) | 放入字符串 | 字符串 | 不变 |
| 数组.sort() | 数组排序 | 新数组 | 改变 |
| 数组.push(元素1，元素2，...) | 末尾添加元素（可多） | 新的长度 | 改变 |
| 数组.pop() | 删去尾元素（一个） | 删去的尾元素 | 改变 |
| 数组.unshift(元素1，元素2，....) | 起始添加元素（可多） | 新的长度 | 改变 |
| 数组.shift() | 删去首元素（一个） | 删去的首元素 | 改变 |
| 数组.slice(起始底，起始底+截个数) | 选择数组元素 | 被选元素 | 不变 |
| **数组.splice(起始底，删个数，元素1，元素2，....)** | **删除/添加数组元素** | **被删元素** | **改变** |
| 数组.toString() | 转化为字符串 | 字符串  注：无法改变分隔符 | 改变 |
| 数组.indexOf  (“待索引项”) | 数组中  索引单项 | 若存在,返回**第一个位置**;否则,返回-1 | 不变 |
| 数组.lastIndexOf  (“待索引项”) | 数组中  索引单项 | 若存在,返回**最后一个位置**;否则,返回-1 | 不变 |

注意：①添加元素返回值为新的长度，删减元素返回值为被删减元素。②首尾添加元素可添加多个，首尾删减元素只能删减一个。③字符串转化为数组的方法为：字符串.split(“分隔符”)，字符串分割为的数组各元素仍为字符串。 （join合并；split分割）④indexOf不仅仅是字符串的方法，也是数组的方法。

实例:数组去重

<script>

var arr=[1,2,3,3,5,6,5];

var arr1=[];

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

if (arr1.indexOf(arr[i])==-1) {

arr1.push(arr[i])

};

};

alert(arr1)

</script>

<script>

var arr=[1,2,2,2,5,6,8,5]

var arr1=arr.filter(function(item,index){

return arr.indexOf(item)==index;

})

</script>

1. 函数是为了完成一个具体的业务逻辑，如果需要函数在完成业务逻辑的同时返回一个函数值，可以使用return将函数中某一变量或表达式设定为函数值。
2. 无论是局部变量还是全局变量，都必须先声明后使用。
3. 冒泡排序：

<script>

var str=window.prompt();

document.write("字符串"+str+"<br />")

var arr=str.split(",")

document.write("数组"+arr+"<br />")

for (i=1;i<arr.length;i++) {

document.write("第"+i+"趟排序"+"<br />")

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

document.write("第"+i+"趟"+"第"+(j+1)+"次序"+"<br />")

if(parseInt(arr[j])>parseInt(arr[j+1])){

var temp;

temp=arr[j];

arr[j]=arr[j+1];

arr[j+1]=temp;

}

document.write(arr+"<br />")

}

if(isSort(arr)){

break;

}

}

function isSort(a){

var mask=true;

for (var t=0;t<a.length;t++) {

if(parseInt(a[t])>parseInt(a[t+1])){

mask=false;

}

}

return mask;

}

</script>

1. <script>

var a=prompt("提示文字","默认值") //str为字符串数据类型。

var b=a.split(",")//arr为数组数据类型

var c=b[1] //数组数据类型b的每一个元素仍是字符串数据类型

var d=parseInt(c)//将字符串数据类型转化为Number数据类型

alert(typeof a);//弹出string

alert(typeof b);//弹出object

alert(typeof c);//弹出string

alert(typeof d);//弹出Number

</script>

**04/11**

1. window对象的confirm方法： 显示一个带有提示信息、确认和取消按钮的对话框。

语法：confirm(“提示信息”)

注意：如果将confirm方法赋值给一个变量，当处理该对话框时，若点击确认，该变量为布尔数据类型true，若点击取消，该变量为布尔数据类型false。

原理： <script>

var a=confirm(); //弹出对话框

alert(a); //若对话框点击确认，则弹出ture；若对 话框点击取消，则弹出false。

</script>

例子：

<p id="pid">文字</p>

<button id="btn">删除</button>

<script>

var p=document.getElementById("pid")

var btn=document.getElementById("btn")

btn.addEventListener("click",demo)

function demo(){

var a=confirm("确认删除吗？")

if(a){

p.innerHTML="";

alert("已经删除")

}

}

</script>

注意：DOM2级事件处理程序中addEventListener("事件",demo)，**demo不可加括号。**

1. <script>中的代码无法操作(document)文档流上方的HTML元素对象。

3. 计时器

<p id="pid"></p>

<button onclick="demo2()">按钮</button>

<script>

function demo(){

p=document.getElementById("pid")

var d=new Date();

var year=d.getFullYear();

var month=d.getMonth();

var dates=d.getDate(); //获得的是日期号

var h=d.getHours();

var m=d.getMinutes();

var s=d.getSeconds();

var fulltime=year+"年"+month+"月"+dates+"日"+h+":"+m+":"+s

p.innerHTML=fulltime;

}

var begin=setInterval(**demo**,1000);

//计时器中调用函数的第一种写法，无括号无引号。

//在句柄中添加函数只能用这种方法，无括号无引号。

// setInterval(**"demo()"**,1000);

//计时器中调用函数的第二种写法，有括号有引号。

// setInterval(**function(){demo()}**,1000);

//计时器中调用函数的第三种写法。

function demo2(){

clearInterval(begin)

}

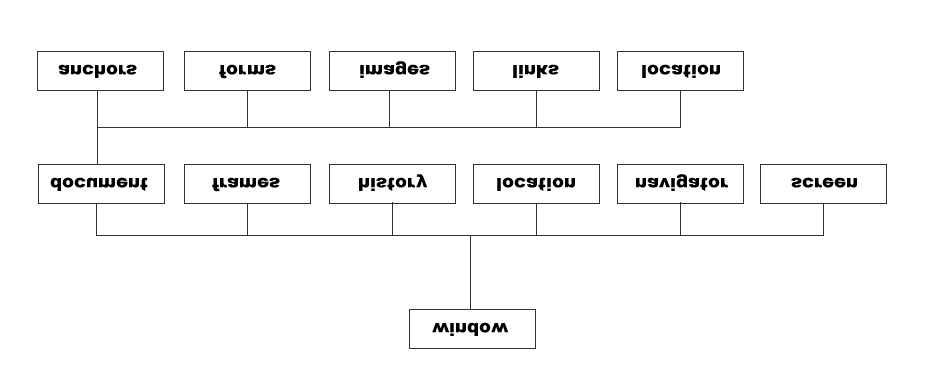
</script>

注意：var begin=setInterval(demo,1000); begin是统计setInterval函数的个数用的，指明是第几个setInterval函数。在本例题中，begin===1。如果不声明begin变量，clearInterval(begin)中的begin改为1，代码运行效果不变。

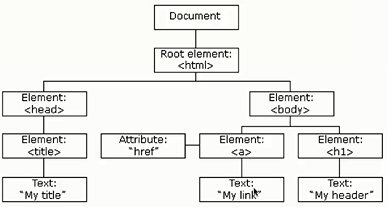
4. BOM浏览器对象模型(Browser Object Model)：使用对象描述了浏览器的各部分内容。

Window对象、计时器、History对象、Location对象、Screen对象、Navigator对象、弹出窗口、Cookies。

页面刷新: location.reload();



DOM文档对象模型(Document Object Model)：当一个html页面加载到浏览器的时候，那么浏览器会为每个标签都创建一个对应的对象描述该标签的所有信息，那么我们看到的网页信息实际上就是看到了这些标签对象的信息，如果我们需要操作页面数据，那么我们就可以通过这些标签对象进行操作。



5. Location对象用来更改当前页面的URL

<button id="btn1">按钮</button>

<script>

document.getElementById("btn1").addEventListener("click",demo)

//在句柄中添加函数只能用这种方法，无括号无引号。

function demo(){

location.href="new\_file.html"

}

</script>

1. 在服务器通过location.search获取数据（键值对）。

页面1代码：

<form method="get" action="new\_file.html">

<input type="text" name="usename" />

<input type="password" name="passord" />

<input type="submit" />

</form>

页面2(new\_file.html)代码：

<button id="btn">按钮</button>

<script>

var btn= document.getElementById("btn");

btn.addEventListener("click", demo)

function demo() {

var a = **location.search** //取得当前 URL的查询字串(?符号及后面的部分)（字符串格式）

alert(a)

var b = **a.substring(1)**

//截取从第二位开始的查询字串(substring:子串、子链)

var c = b.split("&")

//将查询字串以"&"为分隔符分隔为数组

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

alert(c[i]) //逐个弹出每一个数组元素

alert(c[i].split("=")[1])

//弹出每一个键值对的结果

};

};

</script>

7. JavaScript中的For...in 声明

for...in 声明用于遍历数组或者对象的属性（对数组或者对象的属性进行循环操作）。for ... in 循环中的代码每执行一次，就会对数组的元素或者对象的属性进行一次操作。

语法：for (变量 in 对象){代码块}

注意：for ... in 循环在遍历数组时，变量指每一个数组的下标值，对象[变量]指每一个数组元素。for ... in 循环在遍历对象时，变量指每一个对象的属性名，对象[变量]指每一个对象的属性值。

例子：获取浏览器(window)对象导航(navigator)子对象的所有属性名和属性值。

<script>

for (i in window.navigator) {

document.write(i+"="+window.navigator[i]+"<br />")

};

</script>

8. B/S结构(Browser/Serve，浏览器/服务器模式)，是WEB兴起后的一种网络结构模式，WEB浏览器是客户端最主要的应用软件。这种模式统一了客户端，将系统功能实现的核心部分集中到服务器上，简化了系统的开发、维护和使用。

C/S结构(Client/Serve，客户端和服务器模式)，是软件系统体系结构，通过它可以充分利用两端环境的优势，将任务合理分配到Client客户端和Serve服务端来实现。

1. document.images：获取文档上所有的img标签的集合。

document.images[0]：获取文档上第1个img标签。

document.forms：获取文档上所有的form标签的集合。

document.forms[n]：获取文档上第n+1个form标签。

举例： <img src="img/shili.jpg" />

<script>

document.images[0].style.border="**3px solid red**"

</script>

注意：这种写法只适用于forms、images两种情况。

**04/12**

1. 获取当前文档URL的两个方法：

<script>

alert(document.URL);

alert(location.href)

</script>

1. 使用childNodes返回的是所有子节点的集合（不包括子节点的子节点）。这些子节点中可能有元素节点、文本节点、注释节点，而不可能有属性节点（属性节点不是任何节点的子节点）。节点有三个属性：nodeName、nodeType、nodeValue。 我们可以通过节点的nodeName属性获取节点名，nodeType属性获取节点类型，nodeValue属性获取节点值。

只有元素节点有子节点（子节点可能是元素节点、文本节点以及注释节点），而元素节点、文本节点、注释节点都有父节点，属性节点既没有子节点又没有父节点。

<script>标签属于元素节点，<script>标签内所有代码（包括注释内容）为<script>标签的**一个**子节点，并且此子节点类型为**文本节点**。即：<script>标签内只有一个子节点，为文本节点，注释内容不被认为是一个注释节点，而被认为是此文本节点的一部分。

注意节点的属性与元素的属性的区别：①节点有三个属性：nodeName、nodeType、nodeValue。②在HTML中，元素和元素的属性都属于节点。

例题：遍历一个元素节点的所有属性节点名、值、类型。

<input id="ipt" type="text" value="123" name="qer" />

<script>

var ipt=document.getElementById("ipt").**attributes**;

for (n=0;n<ipt.length;n++) { document.write(ipt[n].nodeName+";"+ipt[n].nodeValue+';'+ipt[n].nodeType+"<br />")

}

</script>

注意：①ipt为input元素节点的属性节点集合。②for(){}循环遍历input每一个属性节点的名、值、类型。

1. 一切事物皆对象，对象都拥有属性和方法。属性即对象带有的一些特定性质，方法即对象的使用方式。获取属性（特定性质）不加括号，调用方法（使用方式）要加括号。
2. 把某个元素节点的非空白子节点存入数组

<script>

var html=document.getElementsByTagName("html")[0];

var son=html.childNodes

var arr=new Array()

for (n=0;n<html.childNodes.length;n++) {

if (son[n].nodeType==3&&son[n].nodeValue.trim()=='') {

continue;

} else{

arr.push(son[n]);

};

};

alert(arr.length) //弹出2

</script>

注意：字符串.trim()方法，将字符串两边的空格删去。字符串.trimleft()方法，将字符串左边的空格删去。字符串.trimright()方法，将字符串右边的空格删去。

例子： <script>

var str=" ad2cd "

alert(str.length);//弹出13

alert(str.trimLeft().length);//弹出11

alert(str.trimRight().length);//弹出7

alert(str.trim().length);//弹出5

</script>

**04/13**

1. 递归：函数调用自身的编程技巧称为递归。

注意：①如果要用递归得到数学运算的最终结果，每一个递归函数的返回值即是当前递归次数的最终数学运算结果。②用历史递归函数所有的返回值（函数值）和参数（即递归次数）表示当前递归函数的返回值（函数值）。

例子1：求1+2+3+……100

<script>

function getSum(n){

if (n==1) {

return 1;

} else{

return getSum(n-1)+n

}

}

alert(getSum(100))

</script>

例子2：求1,1,2,3,5,8……的第25项

<script>

function getNum(n){

if (n==1||n==2) {

return 1;

} else{

return getNum(n-1)+getNum(n-2);

}

}

alert(getNum(25))

</script>

例子3：求1！+2！+3！+……+10！

<script>

function getSum(n) {

if (n == 1) {

return 2;

} else if (n == 2) {

return 3;

} else {

return getSum(n - 1) + (getSum(n - 1) - getSum(n - 2)) \* n;

//用上一次和上上一次递归函数的返回值（函数值）以及n表示当前递归函数的返回值（函数值）。

}

}

alert(getSum(10))

</script>

1. 创建一个函数，将html元素的id名作为函数的参数，将该html元素作为函数相对应的返回值。

<p id="pid">123</p>

<div id="div">abc</div>

<script>

function id(a){

return document.getElementById(a)

}

alert(id("pid").innerHTML) //弹出123

alert(id("div").innerHTML) //弹出abc

</script>

注意：在调用此函数时，实际参数的赋值必须为**带引号的ID名**。

3. 创建一个insertAfter(新子节点，旧子节点)函数，该函数可以使新子节点插入到旧子节点之后。

<div id="div1">第一个DIV</div>

<div id="div2">第一个DIV</div>

<div id="div3">第一个DIV</div>

<script>

var p=document.createElement("p");

p.innerHTML="这是新增的p元素";

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

var div2=document.getElementById("div2");

var div3=document.getElementById("div3");

function insertAfter(new1,old1){

var parent=old1.parentNode

if (old1==parent.lastChild) {

parent.appendChild(new1)

} else{

parent.insertBefore(new1,old1.nextSibling)

}

}

insertAfter(p,div2);

</script>

注意：createElement()创建元素节点；parentNode父节点；childNode**s**子节点**集**；firstChild第一个子节点；lastChild最后个子节点；previousSibling前一个兄弟节点；nextSibling后一个兄弟节点。

**JavaScript**

ECMAScript：解释器、翻译，将程序代码与0、1之间相互翻译转化。

DOM使JavaScript拥有操作页面（document文档根节点）的能力。

BOM使JavaScript拥有操作浏览器（window）的能力。

**DOM**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **节点** | | | | | |
| **Node节点-节点属性（分类）** | | | | | |
| 节点  属性 | 节点 | nodeName | nodeType | nodeValue | 备注 |
| 元素 | 元素名称 | 1 | null | document不属于元素节点，html属于元素节点 |
| 属性 | 属性名称 | 2 | 属性值 |
| 文本 | #text | 3 | 文本内容 |
| 注释 | #comment | 8 | 注释内容 |
| document | #document | 9 | null |

①Document是文档根节点，属于window对象（网页归属于浏览器）。②script是元素节点，document是节点，但不是元素节点；window不是节点。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Node节点-节点定位** | | | | |
| 定位节点 | 定位**元素**节点 | **children** | **元素子节点集**  **（最常用）** | 通过父节点定位 |
| firstElementChild | 第一个元素子节点**（常用）** |
| lastElementChild | 最后一个元素子节点**（不能用）** |
| previousElementSibling | 前一个元素兄弟节点**（常用）** | 通过兄弟元素节点定位 |
| nextElementSibling | 后一个元素兄弟节点节点**（常用）** |
| 定位**元素**节点、**文本**节点、**注释**节点 | parentNode | 父节点 | 通过节点层次结构可以定位到3种类型的节点 |
| childNodes | 子节点**集**  **（不常用）** |
| firstChild | 第一个子节点 |
| lastChild | 最后一个子节点 |
| previousSibling | 前一个兄弟节点 |
| nextSibling | 后一个兄弟节点 |
| 定位**属性**节点 | **元素**.attributes | 定位属性节点**集** | 属性节点通过对应元素定位 |
| 元素.属性名 | 获得某个**属性值**（常用方法） |

注意：节点集即是一个存储多个节点的集合——**类数组(类数组不是数组)**。

JavaScript中的类数组:节点集合、函数参数集合arguments。

|  |  |  |  |
| --- | --- | --- | --- |
| **Node节点-节点操作** | | | |
| 节点类型 | 语法 | 含义 | 备注 | |
| Docu  ment | write() | 将字符串或元素插入文档 | 文档根节点 | |
| 属性  节点 | getAttribute(“属性名”) | 获取某个属性值（不常用） | 操作属性节点是**对应**元素节点的方法 | |
| 元素.属性名=“属性值” | 获取或设置某个  **属性值**（常用） |
| setAttribute(“属性名称”，“新的属性值”) | 设置某个属性值（不常用） |
| removeAttribute  (“属性名”) | 移除某个属性  （不常用） |
| 元素节点 | document.createElement(“元素节点名称”) | 创建元素节点 | 创建元素节点为文档**根节点**的方法；添加、插入、删除、替换元素子节点为元素**父节点**的方法；克隆元素节点为元素**模节点**的方法 | |
| 父节点.appendChild  (元素子节点) | 添加子节点 |
| 父节点.insertBefore(新子节点,老子节点); | 插入子节点 |
| 父节点.remove**Child**  (被删子节点) | 删除子节点 |
| 父节点.repalceChild  (“新节点”，旧节点) | 替换子节点 |
| 模节点.cloneNode  (true/false) | 克隆节点 |
| 文本节点 | document.creatTextNode(“文本内容”) | 创建文本节点 | 创建文本节点为文档**根节点**的方法；添加文本子节点为**父元素节点**的方法 | |
| 父节点.appendChild  (文本子节点) | 添加文本子节点 |

注意：①获取或设置CSS行间样式的属性值方法为——元素.style.属性名=“属性

值”。②父节点.appendChild(文本子节点)的含义为将其他位置的节点**移动（而不是复制）**到父节点内部尾部。

|  |  |  |  |
| --- | --- | --- | --- |
| **文档根节点-document节点** | | | |
| document对象属性 | 语法 | 返回值 | 备注 |
| document.title | 文档标题 | 可输出可修改 |
| document.URL | 当前文档URL | 可输出不可修改  location.href  可修改 |
| document.refereer | 上一个文档的URL | 可输出无法改变 |
| document.body | 当前文档的body元素 | 唯一 |
| document.forms | 当前文档所有表单集合 | 集合 |
| document.images | 当前文档所有图片集合 | 集合 |
| document对象方法 | getElementById() | **定位**指定ID的**元素** | 唯一 |
| getElement**s**ByClassName() | **定位**指定类名  的**元素**集合 | 集合 |
| getElement**s**ByName() | **定位**指定名称  的元素**集合** | 集合 |
| getElement**s**ByTagName() | **定位**指定标签名  的元素**集合** | 集合 |

注意：getElementById()等定位元素的方法不仅仅是document对象的方法，而是**外层元素定位内层元素**的通用方法。

|  |  |  |  |
| --- | --- | --- | --- |
| **DOM操作table表格** | | | |
|  | 语法 | 含义 | 备注 |
| 元素节点定位 | table.caption | 定位标题元素节点 | DOM  提供了定位table表格中  子元素节点  的新方法  **（点的方法）**  定位元素集  中某个元素  用中括号 |
| table.tHead | 定位表头元素节点 |
| table.tBodies | 定位表体元素**集**节点 |
| table.tFoot | 定位表尾元素节点 |
| **table.**tBodies[0].rows | 定位行元素**集**节点 |
| **table.**tBodies[0].rows[n] | 定位某行元素节点 |
| **table.**tBodies[0].rows[n].cells | 定位单元格  元素**集**节点 |
| **table.**tBodies[0].  rows[n].cell[n] | 定位某单元格  元素节点 |
| 元素节点操作 | table.createTHead() | 创建并插入  表头元素节点 | DOM  提供了操作（创建并  插入、删除）  table表格中  元素节点的  新方法。  创建并插入、删除某个元素节点用小括号，括号中若有数字表示下标值 |
| table.deleteTHead() | 删除表头元素节点 |
| table.createCaption() | 创建并插入  标题元素节点 |
| table.deleteCaption() | 删除标题元素节点 |
| table.createTFoot() | 创建并插入  表尾元素节点 |
| table.deleteTFoot() | 删除表尾元素节点 |
| **table.**tBodies[0].insertRow(n) | 创建并插入  某行元素节点 |
| **table.**tBodies[0].deleteRow(n) | 删除某行元素节点 |
| rows[n].insertCell(n) | 创建并插入  某单元格元素节点 |
| rows[n].deleteCell(n) | 删除某单元格  元素节点 |

注意：只要是元素集合(**类数组**)，就可以获取元素个数。

例如：tBodies.length;rows.length;cells.length等等。

|  |  |  |  |
| --- | --- | --- | --- |
| DOM操作select下拉列表 | | | |
|  | 语法 | 含义 | 备注 |
| 元素节点定位 | **select**.options | 定位选项集合 | DOM  提供了定位select下拉列表中option子元素节点**集**的新方法 |
| **select**.selectedOptions | **定位被选中**  **选项集合** |
| 元素节点操作 | **select**.**add**(new Option(“innerHTML内容”,“value值”)) | 给下拉列表添加新选项  **（常用）** | DOM  提供了添加select下拉列表中option子元素节点的新方法 |
| **select**.selectedIndex | 获取第一个被选项的下标值  （不常用） | DOM  提供了获取select下拉列表中被选option子元素节点下标值的方法 |
| **select**.selectedOptions[n].index | 获取第n+1个被选项的下标值  （常用） |

注意：①alert(select.options[select.selectedIndex].innerHTML)不管被选中选项有几个，只弹出第一个被选项的innerHTML。

②alert(select.selectedOptions[n].innerHTML)弹出第n+1个被选项的innerHTML。

③alert(select.options[n].selected)，若第n+1个选项已经被选，弹出true；若第n+1个选项未被选择，弹出false。

**04/14**

1. 文档流:浏览器自上而下翻译标签。<script>标签内的代码只能识别和操作在**识别和操作时已经加载的DOM节点**。

注意：文档流的执行顺序由上而下**不同步**：①<script>标签内触发函数中的代码可以识别和修改所有函数触发时已加载的DOM节点。②<script>标签内的一般函数（随<script>标签的加载而调用）不可以识别和修改<script>标签下方的DOM节点。

例子1：

</body>

<div>

<button id="btn1" onclick="demo()">按钮1</button>

</div>

<script>

for (n=0;n<document.body.childNodes.length;n++) {

alert(document.body.childNodes[n].nodeName)

//最后一个弹出SCRIPT

}

function demo(){

for (n=0;n<document.body.childNodes.length;n++) {

alert(document.body.childNodes[n].nodeName)

//增加三个子节点，最后一个弹出#text

}

}

</script>

<p>文字</p>

</body>

例子2：

<script>

document.getElementById("pid").innerHTML="123"

//无法操作

</script>

<p id="pid">文字</p>

2. 若元素节点前后都有一个文本节点（是否非空都适用），删除此元素节点，两个文本节点不会合并。即：父节点中删除一个子节点，父节点的子节点数量只会减少一个（被删减节点），而不会发生由于空子文本节点合并产生的子节点数目减少。

3. style对象只能访问和设置行内样式，运用style对象的一些属性和方法。

style.cssText style 中的CSS 行内样式代码

style.length 属性数量

style.getPropertyValue(“属性名”) 返回该属性的属性值

style.item(index) 返回指定位置CSS 属性名称

style.removeProperty(“属性名”) 从样式中删除指定属性

style.setProperty(“属性名”,”属性值”) 给属性设置为相应的值（不常用）

style.属性名=“属性值” 给属性设置为相应的值（常用）

<body>

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

height: 100px;color:red;background-color: aqua;">文字</div>

<script>

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

alert(div.style.cssText);

alert(div.style.length);

alert(div.style.getPropertyValue("color"));

alert(div.style.item(1))

div.style.removeProperty("background-color")

div.style.setProperty("background-color","#666")

</script>

</body>

4. 在做表格table时，即使不用tbody包裹tr，浏览器在解析时，仍会给所有的tr外套一个tbody。为避免混乱，在做表格时需要加上tbody。**在添加、插入、删除、替换元素子节点时，父节点用子节点.parentNode表示更准确，因为子节点可能有隐藏的父节点。**

<table>

<tr></tr>

<tr></tr>

</table>

<script>

tr=document.getElementsByTagName("tr");

for (n=0;n<tr.length;n++) {

alert(tr[n].parentNode.nodeName) //弹出TBODY(2次)

}

</script>

**04/15**

1. select无法设置value值，select的value值即是被选中(selected)选项(option)的value值，并且随被选中选项(option)的变化而变化。

例题：<select id="sel" >

<option value="A">1</option>

<option value="B">2</option>

<option value="C">3</option>

<option value="D">4</option>

</select>

<script>

var sel=document.getElementById("sel");

alert(sel.options.length) //弹出4

sel.onchange=function(){

alert(sel.value)

//弹出select选项的value值，为被选中选项的value值。

}

</script>

注意：①option标签innerHTML值为1,2,3,4,value值为A，B，C，D。②与table表格相似，DOM提供了定位下拉列表(select)元素节点中选项(option)子元素节点集的新方法(select.options)**（点的方法）**。

**04/18**

1. Step over next function ——执行下一个同级函数。（逐方法）

Step into next function——执行下一个语句。（逐语句）

Step out of current function——跳到外层函数。

1. 面向对象：OOP——Object Oriented programming
2. 对象是具体的，类是抽象的，类由对象抽象而成，对象由类创建而成。
3. 在JavaScript中，构造函数**等价于**类。
4. JavaScript为动态语言，可以给对象（内置对象和自定义对象）动态添加、删减、修改属性和方法。而在静态语言中，不可以执行此类操作。

**04/19**

1. 原型与构造函数（类）一一对应。原型与构造函数的名字相同，都为构造函数名，但是构造函数的数据类型为function，而原型数据类型为Object。

注意：原型与构造函数**不一定**一一对应。

赋值—— 若：Worker.prototype**=**Person.prototype

则：**Person类、Worker类—→Person原型**

过程：此时Person、Worker两个构造函数的prototype都指向Person原型，而Person原型的constructor指向Person构造函数。**（P31蓝色箭线起始位置分叉）**

继承——若：for (i in Person.prototype) {

Worker.prototype[i] = Person.prototype[i]}

则：**Person原型—**→**Person类；Worker原型—**→**Worker类**

过程：此时Person构造函数的prototype都指向Person原型，Worker构造函数的prototype都指向Worker原型。其实，Person与Worker在原型链中是**平行**的关系，而不是后代的关系。**（P31红色箭线起始位置分叉）**

2. 对象和原型都可以定义属性和方法，构造函数（类）没有对象或原型的属性和方法，构造函数中的this指代的不是构造函数自身，而是构造函数所实例出的对象。

3. 给原型添加新属性，与该原型相对应（原型—构造函数—对象）的所有对象都拥有该属性；给对象添加新属性，不会在对应原型中添加对应的属性；修改对象从原型中继承属性的属性值，该对象会有两个同名属性，一个是该对象自有的属性，一个是从原型中继承的属性，两个同名属性的属性值不同，自有属性的属性值**优先级**高于原型属性的属性值，自有属性的属性值不影响原型同名属性的属性值，原型同名属性的属性值不变；修改原型中已有属性的属性值，如果对象已定义同名属性的新的属性值，则对象的属性值不变，如果对象没有定义同名属性的新的属性值，对象的属性值随原型属性值的改变而改变。

即：修改原型属性，不一定影响对象；修改对象属性，一般不影响原型。

注意：所有的**自定义**对象**都可以**通过对象直接给原型添加属性，**也都可以**添加自有属性，所有的**内置**对象（包括：1，“abc”）**都可以**通过对象直接给原型添加属性，**但并不是**所有的内置对象都可以添加自有属性——①Object对象数据类型**既可以**添加自有属性，**又可以**给原型添加属性：数组new Array、函数function、元素（例外情况：null为Object数据类型但既不可以添加自有属性属性，又不可以给原型添加属性。）。②直接写的基本数据类型**不可以**添加自有属性，**但可以**通过直接写的基本数据类型给原型添加属性：boolean、number、string。（例外情况：undefined为基本数据类型但既不可以添加自有属性属性，又不可以给原型添加属性。）③从类中new出来的基本数据类型为Object，**既可以**添加自有属性，**又可以**给原型添加属性。

var t1=10;——t1是一个Number数据类型，不是一个对象。

var t2=new Number(10);——此时t2是Object数据类型，是一个对象。

var t3=“abc”;——t3是一个String数据类型，不是一个对象。

var t4=new String(“abc”);——此时t4是Object数据类型，是一个对象。

即：直接写的基本数据类型不是对象，new出来的一切对象皆对象，**无论是不是对象，都可以直接给原型添加属性，但只有对象可以添加自有属性。**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 对象 | 分类 | 数据  类型 | 给对象  添加属性 | 通过对象给类添加属性 | 备注 |
| 内置  对象 | 类中实例出的基本数据类型 | Object | √ | √ |  |
| 直接写的基本数据类型 | 不是Object | × | √ | Undefined例外 |
| 引用数据类型 | Object | √ | √ | Null例外 |
| 自定义对象 | 类中实例出  的自定义对象 | Object | √ | √ |  |

4. 既可以通过构造函数（类）给原型添加或修改属性和方法，又可以通过对象直接给原型添加或修改属性和方法。原型由两部分组成，一部分是原型属性与属性值组成的键值对集合，一部分是原型的constructor属性，该属性指向对应的构造函数（类）（每一个函数**都有**一个prototype原型属性，这个属性是一个指针，指向该函数的原型对象）。构造函数有三个作用，一个作用是给原型添加或修改属性，第二个作用是实例出对象，第三个作用是给构造函数实例出的对象添加各自不同的属性值。构造函数在原型构造对象过程中存在的最大意义即是实例出对象，在对象与原型之间搭建桥梁；第二意义是对实例出的对象添加各自不同的属性值，第三意义是给原型添加或修改属性，由于构造函数实例出的对象也可以直接给原型添加或修改属性，所以这一过程虽然很常用，但意义并不重大。

5. 构造函数（类）.prototype.属性名=属性值。——含义为给构造函数、实例对象的原型添加（修改）一个属性。

构造函数（类）.prototype={}——含义为用一个新的对象覆盖构造函数的原型对象。此时新原型对象的constructor属性值无法指向当前构造函数，需要定义新原型对象的constructor属性为当前构造函数。

1. 对象.\_\_proto\_\_.属性名=属性值。——含义为给实例对象、构造函数的原型添加（修改）一个属性。
2. 在JavaScript中，一切对象都有\_proto\_属性，而只有函数有Prototype属性。
3. 原型链：①由字面量或工厂方式定义的对象到Object；②由构造函数（类）到Object。③由构造函数实例出的对象到Object。

①：字面量构造对象/工厂方式构造对象P3—\_ptoto\_→Object

②：Person构造函数（类）—\_ptoto\_→function()—\_ptoto\_→Object

③：P1、P2—\_ptoto\_→Person(Person构造函数(类)的prototype属性属性值)—\_ptoto\_→Object

原型链：①第一行矩形中内容表示原型对象，底行矩形中内容表示构造函数。

②箭头起始矩形中内容为对象，箭头上的椭圆表示该对象的属性名，箭头指向矩形中内容为该对象的属性值。

③没有椭圆的箭头表示该对象的\_proto\_属性值，没有椭圆的箭头组成原型链。

过程： ①所有对象都有\_proto\_属性，所有的\_proto\_属性都指向原型。

②所有原型的\_proto\_属性都指向object原型，所有构造函数的\_proto\_属性都指向上一级原型。

③只有函数有prototype属性，函数的prototype属性指向它的原型。

结论： ①函数的prototype指向它的原型、函数的\_proto\_指向上一级原型，原型的\_proto\_指向Object原型。

②函数—prototype→同级原型；函数—\_proto\_→上一级原型；原型—construct→同级函数；原型—\_proto\_→object原型。

③类即是一个函数，原型的原型即是一个对象（Object），可以设置对象的原型指向一个明确的Object（原型或实例对象）。

字面量与工厂方式创建的对象

的实例对象

prototype

constructor

prototype

constructor

prototype

constructor

构造函数的实例对象

的实例对象

function原型

object原型

person原型

Object函数

的实例对象

Function函数

的实例对象

Person类（构造函数）

的实例对象

例子1：

<script>

var p1={name:'Jake'} //字面量

function Person(a){ //工厂方式

var o=new Object();

o.name=a;

return o;

}

p2=Person('Jake')

function demo(a){ //构造函数

this.name=a;

}

p3=new demo('Jake');

</script>

<script>

function Person(){}

var person1=new Person();

person1.\_\_proto\_\_.name='Jake';//通过对象给原型添加属性。

alert(Person.prototype.name);

//通过构造函数访问原型的属性，弹出Jake。

Person.prototype.age='26';//通过构造函数给原型添加属性。

alert(person1.\_\_proto\_\_.age);

//通过对象访问原型的属性，弹出26。

alert(person1.name);//访问对象的原型属性，弹出Jake。

alert(Person.age);

//弹出undefined，构造函数没有继承原型的属性。注意：每一个函数都有一个name属性，该属性的属性值是指向该函数的函数名。每一个函数都有一个prototype属性，该属性的属性值指向该函数的原型对象。

person1.name='Jason';

//给对象新添加一个在原型中同名的自有属性。

alert(person1.name);//弹出Jason。

alert(person1.\_\_proto\_\_.name)

//弹出jake，原型对象的同名name属性的属性值并未改变。

</script>

注意：对象可以访问原型的属性，可以继承原型的属性，可以直接给原型添加（修改）属性，可以新添加原型中同名的自有属性，也可以创建原型中所没有的自有属性；构造函数可以访问原型的属性，可以给原型添加（修改）属性，可以给对象添加（修改）自有属性，也可以实例出对象。

例子2：

<script>

function Person(){}

Person.prototype={ //创建新的原型，覆盖之前的原型。

constructor:'Person',//定义新原型的构造方法指向Person。

name:[1,2], //给原型对象添加name属性。

}

var p1=new Person();//在构造函数中实例（new）出对象p1。

var p2=new Person();//在构造函数中实例（new）出对象p2。

p1.name.push(4);

//这种方式并不是给对象p1新添加一个在原型中同名的name自有属性，而是修改p1的原型的name属性值。

alert(p1.name); //获取对象p1的name属性值，弹出1,2,4。

alert(p2.name);

//获取对象p2的name属性值，弹出1,2,4。说明原型的name属性值被改变。

</script>

例子3：

<script>

function Person(){}

Person.prototype={ //新的原型覆盖之前的原型。

constructor:'Person', //定义新原型的构造方法指向Person。

name:[1,2], //给原型添加name属性。

say:function(){ //给原型添加say属性。

return this.name;//返回原型对象的name属性值。

}

}

var p1=new Person();//在构造函数中实例（new）出对象p1。

var p2=new Person();//在构造函数中实例（new）出对象p2。

alert(p1.say());

//获取对象p1在原型中继承的say属性，弹出1,2。

p1.name=[1,2,3];

//给对象P1新添加一个在原型中同名的name自有属性，属性值为[1,2,3]。

alert(p1.name);

//获取p1的name属性值，由于name自有属性的属性值优先级高于原型中继承的同名name属性的属性值，所以弹出1,2,3。

alert(p2.name);

//弹出1,2。说明原型的name属性值未被改变。

p1.name.push(4);

//通过这种方式修改对象p1的name**自有属性**的属性值。

alert(p1.name); //弹出1,2,3,4。

alert(p2.name);

//弹出1,2。说明原型的name属性值仍未被改变。

p1.name1=['a','b','c'];//在对象p1中增加name1属性。

alert(p1.name1);//弹出a,b,c

alert(p2.name1);

//弹出undefined。说明原型中并未添加name1属性。

</script>

例子4（对象自定义原型1）：

<script>

var a = { x: 10,calculate: function(z) { return this.x + this.y + z } };

//字面量构造对象a，未自定义对象a的原型，对象a的原型指向Object。

var b = { y: 20,\_\_proto\_\_: a};

//字面量构造对象b，自定义**对象b**的原型为**对象a**。

var c = { y: 30};

//字面量构造对象c，未自定义对象c的原型，对象c的原型指向**Object**。

alert(b.calculate(30));//弹出60

</script>

例子5（对象自定义原型2）：

<script>

var a={age:10,say:function(){

alert('hello')

}};

var b={age:20,\_\_proto\_\_:a};

b.say();//弹出hello。

var c={age:30,\_\_proto\_\_:b};

c.say();//弹出hello。

</script>

**04/20**

1. 静态属性：类的属性；实例属性：对象的属性。
2. 替换字符串中字符的方法：

语法：replace(“原内容”，“新内容”)

例子1：<script>

var a="hello world"

document.write(a.replace("world","世界"))

</script>

例子2：<script>

var str='123';

str[0]='a';

alert(str);

//弹出123，说明str[n]的方式只能够获取而不能修改某一位置字符。

alert(str.charAt(1))//弹出2。

str.charAt(1)='b';

//语法错误，charAt(n)的方式只能够获取而不能修改某一位置字符。

</script>

注意：修改字符串中的字符只有一种方法：replace(“原内容”，“新内容”)。使用此方法时需要注意：①若有重复的原内容，只修改第一处。②返回值为新字符串，原字符串不变。

例子3：<script>

var str=prompt();

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

if(str[n]=='a'){

var str=str.replace(str[n],'A');

//①因为replace只替换第一个a字母，所以必须放在循环里。

//②因为返回值为新字符串，原字符串不变，所以必须将替换后的字符串赋值给原字符串。

}

}

alert(str);

</script>

**04/21**

1. <script>

var reg=/ac/i;

var str="acAcACaCacAcACaC"

alert(reg.test(str));//弹出true。

alert(reg.exec(str));//弹出ac。

alert(reg.exec(str));//弹出ac。

alert(reg.exec(str).index);//弹出0。

alert(reg.exec(str).index);//弹出0。

//正则表达式的test方法返回ture和false，正则表达式的exec方法返回匹配项，正则表达式匹配项的index方法返回匹配项位置。

//若不设置全局搜索g，则正则表达式reg每次执行test方法或exec方法，都只定位到第一个匹配项。

</script>

<script>

var reg=/ac/ig;

var str="acAcACaCacAcACaC"

alert(reg.test(str));//弹出true。

alert(reg.exec(str));//弹出Ac。

alert(reg.exec(str));//弹出AC。

alert(reg.exec(str).index);//弹出6。

alert(reg.exec(str).index);//弹出8。

//若设置全局搜索g，则正则表达式reg每次执行test方法或exec方法之后，下一次执行test方法和exec方法时都会定位到下一个匹配项。

</script>-->

<script>

var reg=/ac/ig;

var str="acAcACaCacAcACaC"

var result;

while((result=reg.exec(str))!=null){

document.write(result.index+"——"+result+"<br />")

}

</script>

**问题：为什么必须将reg.exec(str)赋值给一个result？**

**04/22**

1. parseInt('n进制数字',n) ——将n进制数字转化为10进制数字，当n等于10时，可以简写为parseInt('10进制数字')。

<script>

alert(parseInt('23',10)) //弹出23。

alert(parseInt('23',16)) //弹出35。

alert(parseInt('11',2)) //弹出3。

alert(parseInt('23',8)) //弹出19。

</script>

1. 当未将函数赋值给一个变量时，函数名即代表**函数整体**，通过函数名加括号()调用**函数执行**；当将函数赋值给一个变量时，该变量代表**函数整体**，通过变量名加括号()调用**函数执行**。

在书写代码时，如果想**调用函数执行**，通过函数名加括号或者变量名加括号的方式，如果想要**引入函数整体**，则通过函数名不加括号、变量名不加括号或者直接写整个函数整体的方式。

应用：①DOM2级事件处理程序中addEventListener("事件",demo)，demo表示**引入函数整体**，而不是**调用函数执行**，因此不可加括号。

②事件函数有两种写法，一种是事件等于**函数整体**，一种是事件等于**函数名无括号**（等价于函数整体）。

例如：<script>

function demo(){

alert('1')

}

demo()//弹出1

alert(demo) //弹出function demo(){alert('1')}。

</script>

<script>

var a=function demo(){

alert('1')

}

a()//弹出1

alert(a)//弹出function demo(){alert('1')}

alert(demo) //语法错误：demo is not defined

</script>

1. 构造函数也是一个对象，既可以给构造函数添加属性，又可以给构造函数实例出的对象添加属性。

<script>

function Person(age){

Person.age1='10';

//构造函数也是一个对象，给构造函数添加属性，与实例出的对象无关。

this.age=age;

//给构造函数实例出的对象添加属性，与构造函数无关。

}

alert(typeof Person);//弹出function

var p1=new Person(20);

alert(p1.age1); //弹出undefined。

alert(p1.age); //弹出20。

alert(Person.age1); //弹出10。

alert(Person.age);//弹出undefined。

</script>

给构造函数自身添加属性：**只有函数被调用之后，函数内的代码才会被执行，构造函数才会被添加属性。**

<script>

function Person(){

Person.age=1;

};

alert(Person.age);

//此时构造函数内代码未执行，构造函数的age属性未添加，弹出undefined。

Person();

alert(Person.age);

//此时构造函数内代码已执行，构造函数的age属性已添加，弹出1。

</script>

1. 检测及遍历对象的属性

语法：①”属性名” in 对象——判断该属性是否是该对象的属性（自有属性或者原型属性），返回值为布尔类型。

②对象.hasOwnProperty(“属性名”);——判断该属性是否是该对象的自有属性，返回值为布尔类型。

③for(变量 in 对象){}遍历对象的属性（自有属性或者原型属性），在花括号中，变量指代该对象的属性名，对象[变量]指代该对象的属性值。

④delete 对象.属性名;——删除对象的该属性。

注意：在方法①和方法②中，引号不可省略。

例题：<script>

function Person(name,age){

Person.type="函数"; //给构造函数添加自有属性

this.name=name; //给实例对象添加自有属性

this.age=age; //给实例对象添加自有属性

this.say=function(){ //给实例对象添加自有属性

alert('你好')

}

}

Person.prototype.height="1.8米";

//给构造函数的prototype（实例对象的\_proto\_）添加属性

//此处应注意：构造函数的prototype不等价于构造函数的\_proto\_

var p1=new Person("Jake","26");

//从构造函数中实例出对象p1

alert('type' in Person);//弹出true

alert('type' in p1);//弹出false

alert('age' in Person);//弹出false

alert('name' in p1);//弹出true

alert('height' in Person);

//弹出false。因为Person.prototype指向Person原型对象，指向p1的\_proto\_原型对象，Person的\_proto\_指向上一级构造函数的原型对象

alert(p1.hasOwnProperty('age'))//弹出true

alert(p1.hasOwnProperty('height'))//弹出false

alert(Person.hasOwnProperty('type'))//弹出true

//属性名要加引号

for(pro1 in p1){

document.write("属性名："+pro1+";属性值："+p1[pro1]+"<br />")

//弹出name、age、say、height属性名及对应属性值

}

for(pro2 in Person){

document.write("属性名："+pro2+";属性值："+Person[pro2]+"<br />")

//弹出type属性名及对应属性值**（无height属性）**

}

delete p1.name;//删除实例对象p1的name属性

for(pro1 in p1){

document.write("属性名："+pro1+";属性值："+p1[pro1]+"<br />")

//弹出age、say、height属性名及对应属性值

}

</script>

1. 每一个**栈空间**中存放的可能是**基本数据**类型（Number、string、……）的**数据本身**，也可能是**引用数据**类型在堆中存放的数据**地址**。(栈:stack;堆:heap)
2. 对于基本数据类型而言，每一个**变量名**与每一个**栈空间**与每一个**数据本身**都是一一对应的关系，并不存在多个变量名对应一个数据的情况。对于引用数据类型而言，每一个**变量名**对应每一个**栈空间**，**多个栈空间**可能存放**同一个堆地址**，指向**同一个数据本身**。即：对于基本数据类型，**变量名**与**数据本身**一一对应，相互并不影响。对于引用数据类型，多个**变量名**可能对应同一个**数据本身**，独自都可以修改堆空间中存放的**数据本身**。

对于**基本数据类型**，a=b的含义为将变量b对应栈空间中存放的**数据本身**复制一份存放在变量a对应的栈空间中，**数据本身发生复制**。 对于**引用数据类型**以及自定义对象，a=b的含义为将变量b对应栈空间中存放的**堆地址**复制一份存放在变量a对应的栈空间中，堆地址对应的堆空间中存放的**数据本身并未发生复制**。

对于引用数据类型以及自定义对象，即使a=b，变量a与变量b也并不对应同一块**栈空间**，只是它们所对应的不同栈空间中存放的是同一个**堆地址**。

例题1-1： <script>

var a=[1];

var b=a;

//变量a对应栈空间中存放的堆地址复制一份存放在变量b对应栈空间中。

a=a.push(2);

//修改变量a对应栈空间中存放的堆地址对应的堆空间中的数据本身。

alert(b); //弹出1,2

</script>

例题1-2： <script>

var a=new Object();

var b=a;

//变量a对应栈空间中存放的堆地址复制一份存放在变量b对应栈空间中。

a.age=12;

//修改变量a对应栈空间中存放的堆地址对应的堆空间中的数据本身。

alert(b.age);//弹出12

</script>

例题1-3： <script>

var a=[1];

var b=a;

//变量a对应栈空间中存放的堆地址复制一份存放在变量b对应栈空间中。

b.age=2;

//修改变量b对应栈空间中存放的堆地址对应的堆空间中的数据本身。

a.age=1;

//修改变量a对应栈空间中存放的堆地址对应的堆空间中的数据本身。

alert(b.age);//弹出1

</script>

注意：只有**点**能够修改堆空间中存放的数据本身，只有**等号**则重新开辟一块堆空间存放新数据。

例题2-1： <script>

var a=[1];

var b=[1];

//与b=a的含义不同。此时虽然数组b与数组a的数值相等，但是含义为**新申请一块堆空间**，存放数据[1],在变量b对应的栈空间中存放该堆地址。此时有两个堆空间都存放数据[1]。变量a与变量b对应不同的数据，互不影响。

a.age=23;

alert(b.age);//弹出undefined。

</script>

例题2-2： <script>

var a=[1];

var b=a;

//变量a对应栈空间中存放的堆地址复制一份存放在变量b对应栈空间中。

a=[1,2];

//注意:这一步为**重新申请一块堆空间**，存放数据[1,2]，并在变量名a对应栈空间中清除老的堆地址，存放该新申请的堆地址。——①并未改变变量名a对应的栈空间以前存放的堆地址对应的数据本身。②并未改变变量名b对应的栈空间中所存放的堆地址。

alert(b);//弹出1。

</script>

1. 全局变量的作用范围：从当前全局变量的**声明位置**到最后一个script标签的**结束位置**。

例题1-1： <script>

var a=1;

alert(a);//弹出1。

</script>

例题1-2： <script>

alert(a);//弹出undefined。

var a=1;

</script>

例题1-3： <script>

var a=1;

</script>

<script>

alert(a);//弹出1。

</script>

例题1-4： <script>

alert(a);//运行错误：a is not defined

</script>

<script>

var a=1;

</script>

1. JavaScript预解析是把**变量**、**函数**以及函数的**参数预解析**到他们能调用的环境中，并将其赋一个**初始值**。

预解析对象：①**var 变量名** ②**function 变量名(){}**

③**函数形式参数（等价于var 变量名）**

预解析结果：var 变量 ——解析为—→**变量名=undefined**

function 变量名(){函数体}——解析为—→**变量名=function 变量名(){函数体}**

同一变量名:②预解析结果优先级高于①预解析结果undefined的优先级。

所有变量声明在预解析阶段完成，当开始逐行解读代码后只能通过表达式（赋值运算符）来改变变量的值和类型。（动态语言）

注意：JavaScript不会预解析的两种情况

①无var的变量。②变量名=function (){}

例题1-1： <script>

alert(a);//弹出undefined。

var a=1;

</script>

例题1-2： <script>

alert(demo);//弹出undefined。

var demo=function(){

alert(1)

}

</script>

例题1-3： <script>

alert(a);//运行错误，a is not defined。

a=1;

</script>

例题1-4： <script>

alert(demo); //弹出function demo(){alert(1)}。

function demo(){

alert(1)

}

</script>

例题1-5： <script>

alert(demo);//运行错误：demo is not defined。

demo=function(){

alert(1)

}

</script>

1. 变量预解析与变量值修改

预解析的意义———生成当前环境中的变量对象。

变量预解析：①**var 变量名** ②**function 变量名(){}**  ③**函数形式参数**

变量值修改：①**var 变量名=** ②**变量名=**（**赋值运算符**是**修改变量值**的标志）

|  |  |  |  |
| --- | --- | --- | --- |
| 对比 | 预解析 | 修改变量  名对应的值 | 修改变量名  对应的函数体 |
| var 变量名= | √ | √ |  |
| 变量名= | × | √ |  |
| function 变量名(){} | √ |  | × |
| 变量名=function(){} | × |  | √ |

注意：①var 变量名=function(){} 可预解析可修改，预解析结果为变量名=undefined，修改后结果为变量名=function(){}②function 变量名(){}这种写法虽然也表示变量名等于整个函数体，但是由于没有赋值运算符，所以**只会在预解析过程中修改变量值，不会在代码执行过程中修改变量的值。**

例子1：<script>

alert(a);//弹出function a(){ alert ("better");};

var a=1;//可预解析，可修改。

alert (a);//弹出1。

var a=function (){

alert("good"); }; //可预解析，可修改。

alert (a);//弹出function (){ alert("good");};

a=3;//不可预解析，可修改。

alert(a);//弹出3。

function a(){ alert ("better");};//可预解析，不可修改。

alert (a) //**弹出3。**

</script>

例子2：<script>

alert(a);//预解析之后，a=function a() {alert("pleased")}; //故弹出 function a() {alert("pleased")};

function a() {

alert("bad")

}; //可预解析，不可修改，a=function a() {alert("pleased")}

a();//弹出pleased。

var a = 3;//可预解析，可修改。a=3;

alert(a);//弹出3。

var a = function() {

alert("happy")

};//可预解析，可修改。 a = function() {alert("happy")};

a();//弹出happy。

function a() {

alert("pleased")

};//可预解析，不可修改。a = function() {alert("happy")};

a();//弹出happy。

</script>

注意：①关注代码执行过程中变量名a对应的值是否发生改变，只有出现赋值运算符时，变量a的值才会被修改。②只有当变量名a对应的值是一个函数体的时候，a()才会执行，并且执行变量名a当前对应函数体中的代码。

10.执行环境——每个函数都有自己的执行环境。当执行流进入一个函数时，函数的环境就会被推入一个环境栈中。而在函数执行后，栈将其环境弹出，把控制权返回给之前的执行环境。<script></script>也是一个环境。

变量对象——存放当前所在环境中的所有变量和函数；

作用域——变量可产生作用的区域，基本相当于当前执行环境；

作用域链(由内向外搜索变量)——当代码进入一个新环境时，会产生新的变量对象，变量对象有新的作用域，新的作用域与上一级作用域链接形成作用域链。

函数形参预解析过程——函数的形式参数在函数调用过程中第一次被赋值，在函数执行过程中可能会被多次赋值。

举例1-1: <button id="btn">111</button>

<script>

btn.onclick = function() {

name = 10;//这里的name未声明，为全局变量

b(name); //等价于b(10),10为实参,调用函数b时赋值

alert(name); //弹出全局变量name等于10

};

function b(name) {

//隐含条件:var name=“实参(10)”;此时name为局部变量

name = 20; //变量name已声明,为局部变量name=20

};

</script>

举例1-2:<button id="btn">222</button>

<script>

btn.onclick = function() {

name = 10;//这里的name未声明，为全局变量

b(name); //等价于b(10),10为实参,调用函数b时赋值

alert(name); //弹出全局变量name等于20

};

function b(n) {

//隐含条件:var n=“实参”;此时n为局部变量。

name = 20; //变量name未声明，为全局变量name=20

};

</script>

例题2-1：<script>

var a = 1; //外层执行环境(全局)变量a=1

function fn1() {

alert(a); //内层执行环境预解析结束，无a，调用外层执行环境(全局)变量a=1

a = 2; //改变外层执行环境(全局)变量a=2

};

fn1();//弹出1

alert(a)//弹出2

</script>

例题2-2：<script>

var a = 1; //外层执行环境(全局)变量a=1

function fn2() {

alert(a); //内层执行环境预解析结束，内层环境变量a=undefined

var a = 2; //改变内层环境变量a=2

};

fn2();//弹出undefined

alert(a) //弹出外层环境(全局)变量a=1

</script>

例题2-3： <script>

var a = 1;//全局变量a=1

function fn1(a) {//隐含条件:var a=“实参”;此时a为局部变量,未传入实参,局部变量a=undefined

alert(a);

a = 2;//改变局部变量a=2

}

fn1();//执行函数，弹出局部变量a等于undefined

alert(a);//弹出全局变量a=1

</script>

例题2-4： <script>

var a = 1;//全局变量a=1。

function fn1(a) {//隐含条件:var a=“实参”;此时a为局部变量,传入实参1,局部变量a=1

alert(a);

a = 2;//改变局部变量a=2

}

fn1(1);//执行函数，弹出局部变量a等于1

alert(a);//弹出全局变量a=1

</script>

11.元素.scrollTop含义为元素滚动条的向下滚动的距离、元素可视区向下移动的距离、元素向上卷动的距离。

**04/25**

1. 任何在函数中声明的变量，都可以认为是私有变量，一般不能在函数外部访问这些变量。私有变量包括函数的参数、局部变量和在函数内部定义的其他函数。

**04/26**

1. 闭包：将父函数的局部变量**放在子函数中**，将子函数作为父函数的返回值，扩展局部变量的访问范围，使函数外可以访问和修改局部变量。
2. 任何一个函数都是一个对象的方法，函数中的this指代该函数作为一个方法所属于的对象。一个自由函数是全局变量window的方法。
3. 对象属性的访问权限：

①私有变量(var a=1;)只能被私有函数或特权方法访问。

②私有函数(var fn=function(){};)只能被特权方法访问。

③特权方法(this.say=function(){})可以在对象外部访问。

**04/27**

0. 默认情况： 子类—prototype→子类的原型

子类的原型—constructor→子类

Student.prototype=**Student.prototype**  Student.prototype.constructor=Student;

1. 继承的第一种方法： **方法.call(对象，实参1，实参2……)**

2.继承的第二种方法： **子类**—prototype→**父类实例**

子类原型指向父类实例 **父类实例（子类原型）**—constructor→**子类**

Student.prototype=**new Person();** Student.prototype.constructor=Student;

3.继承的第三种方法： **子类**—prototype→**父类原型**

子类原型指向父类原型 **父类原型（子类原型）**—constructor→**子类**

Student.prototype=**Person.prototype;** Student.prototype.constructor=Student;

4.继承的第四种方法： 定义一个**空类**

定义空类作为桥梁 **空类**—prototype→**父类**

**子类**—prototype→**空类实例**

**空类实例（子类原型）**—constructor→**子类**

var F=function(){}

F.prototype=**Parent.prototype;**

Child.prototype=**new F;**

Child.prototype.constructor=Child;

5.继承的第五种方法： ①拷贝**类**的**原型属性**for in

for (i in Parent.prototype) {

Child.prototype[i]=Parent.prototype[i];

};

②拷贝**对象**的**所有属性（浅拷贝）** for in

for (i in p1) {

c1[i]=p1[i];

}; //注意：P40——点与等号的对比。

③拷贝**对象**的**所有属性（深拷贝）** for in

实现数组与对象**数据本身的拷贝**，而不仅复制栈地址。

function extend(child, parent) { //深拷贝封装函数

for (i in parent) {

if (**(typeof parent[i]) === 'object'**) {

child[i] = (**parent[i].constructor === Array**) ? [] : {};

extend(child[i], parent[i])

} else {

child[i] = parent[i];

};

};

};

注释: 可以通过一个对象的构造器来判断它是一个对象还是一个数组。

6. 继承的第六种方法：**对象**继承**对象**的属性，与类无关。

语法：var 子对象=Object.create(父**对象**，子对象的**自有属性**) 。

举例：var 子对象名 = Object.create(父对象名, {

属性名1: {

**writable**: true,//该属性是否可**修改**

**enumerable**: true,//该属性是否可**枚举**

**configurable**: true,//该属性是否可**删除**

value: ‘属性值’, //该属性的属性值。

},

属性名2: {

writable: true,

enumerable: true,

configurable: true,

value: ‘属性值’

});

**04/28**

1. Div随鼠标移动

思路1：鼠标在页面移动时，div的位置等于鼠标在**页面可视区的位置**加上**页面的卷动**距离。

<script>

var div=document.getElementById("div");

document.onmousemove=function(ev){

//给document添加鼠标移动事件，使鼠标在页面任何位置滚动都可以触发此事件。

var oEvent=ev||event;

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

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

div.style.left=oEvent.clientX+scrollLeft+'px';

//鼠标移动事件发生时，使div的位置与鼠标的位置相同。

div.style.top=oEvent.clientY+scrollTop+'px';

}

</script>

思路2：鼠标在页面移动时，div的**位置**等于鼠标在**页面上的位置**。

<script>

div=document.getElementById('div');

document.onmousemove=function(){

div.style.left=event.pageX+'px'

div.style.top=event.pageY+'px'

}

</script>

注意： **pageX/Y—页面**；clinetX/Y—可视区；screenX/Y—屏幕

screenX/Y > pageX/Y >= clinetX/Y

scrollTop/Left——卷动

鼠标在**页面上的位置**等于鼠标在**可视区的位置**加上页面的**卷动距离**。

oEvent.pageY==oEvent.clientY+document.body.scrollTop;

oEvent.pageX==oEvent.clientX+document.body.scrollLeft;

2. 阻止事件冒泡的四种方法——停止(stop)、取消(cancal)

event.stopPropagation();//停止事件冒泡

ev.stopPropagation();//停止事件冒泡

event.cancelBubble=true;//取消事件冒泡

ev.cancelBubble=true;//取消事件冒泡

1. 阻止事件默许行为——阻止(provent)

event.preventDefault();//阻止事件默许行为

event.preventDefault();//阻止事件默许行为

4. json就是javascript中的对象和数组，在Json中，键和值都应该用**双引号**包裹,**不可用单引号包裹**(在Json文件中会出错)。

**04/29**

1. ①可以使用eval函数将字符串转换为JSON对象。

eval('('+string+')');

②高级版本的浏览器中将字符串转为JSON格式：**JSON.parse(string)**

高级版本的浏览器中将JSON转为字符串：**JSON.stringify(obj)**

2. 用for in解析JSON数据。

var people = {

"programmers": [{

"firstName": "Brett",

"lastName": "McLaughlin",

"email": "aaaa"

}, {

"firstName": "Jason",

"lastName": "Hunter",

"email": "bbbb"

}, {

"firstName": "Elliotte",

"lastName": "Harold",

"email": "cccc"

}],

"authors": [{

"firstName": "Isaac",

"lastName": "Asimov",

"genre": "science fiction"

}, {

"firstName": "Tad",

"lastName": "Williams",

"genre": "fantasy"

}, {

"firstName": "Frank",

"lastName": "Peretti",

"genre": "christian fiction"

}],

"musicians": [{

"firstName": "Eric",

"lastName": "Clapton",

"instrument": "guitar"

}, {

"firstName": "Sergei",

"lastName": "Rachmaninoff",

"instrument": "piano"

}]

};

for (i in people) {

for (n = 0; n < people[i].length; n++) {

var tr = document.createElement('tr');

for (m in people[i][n]) {

var td = document.createElement('td');

td.innerHTML = people[i][n][m];

tr.appendChild(td);

};

document.getElementById(i).appendChild(tr);

};

};

注意：var people = **{**"programmers": **[{**"firstName": "Brett",……

perple为第一层(对象{}),people[i]为第二层(数组[]),people[i][n]为第三层(对象{}),people[i][n][m]为第三层对象的每一个值。

①每一个{}或者[]表示一层，在第一个闭合括号(]或者})之前有几个起始括号则表示有有几层。本题中第一个闭合括号为}，在第一个闭合括号之前有三个起始括号，依次为{[{，则表示一共有三层，由外到内依次为json、arr、json。

②数组用for循环，json用for in 遍历。

③无论json还是arr，对象.[i]都表示内层的一个对象(json或者arr)。

④解析时，最内层代表table的一行，最内层的每一个值代表当前行的每一个单元格。

解析过程：打通每一层通道—→最内层**内**创建插入单元格，最内层**外**创建插入行。

第一步：利用for循环(数组)和for in(json)打通每一层通道。

for (var i in people) {

//进入第一层内部

for (var j=0;j<people[i].length;j++) {

//进入第二层内部

for (var n in people[i][j]) {

//进入第三层内部

};

};

};

第二步：最内层内创建插入单元格，最内层外创建插入行。

for (var i in people) {

for (var j=0;j<people[i].length;j++) {

var tr=document.createElement('tr');

for (var n in people[i][j]) {

var td=document.createElement('td');

td.innerHTML=people[i][j][n];

tr.appendChild(td);

};

document.getElementById(i).appendChild(tr);

};

};

1. 字面量实现单例模式

<script>

var p = {

age:20,

say: function() {

alert('hello')

};

};

var p1 = p;

var p2 = p;

alert(p1 === p2); //弹出true

</script>

4. 构造函数内部判断实现单例模式

function Person() {

if (Person.unique !== undefined) {

return Person.unique;

//若构造函数Person的unique属性值已定义，则返回构造函数Person的unique属性值。

//否则执行以下代码。

};

this.name = 'Jake';

//给实例对象添加属性。

this.say = function() {

alert('hello')

};

//给实例对象添加方法。

Person.unique = this;

//定义构造函数的uneque属性，将构造函数的实例对象赋值给构造函数的unique属性。

};

var p1 = new Person();

var p2 = new Person();

alert(p1 === p2); //弹出true

alert(Person.unique === p1); //弹出true。

Person.unique.say() //弹出hello。

5. 闭包方式实现单例模式

<script>

var single=(function(){

//声明一个全局变量singe，single的值为匿名函数的返回值unique。

var unique;

//在匿名函数内声明局部变量unique。

function Person(){

this.age='20';

this.say=function(){

alert('hello')

};

};

//在匿名函数内创建构造函数Person。

if (unique===undefined) {

unique=new Person();

};

//如果局部变量unique未定义，则将构造函数的实例对象赋值给该局部变量。

return unique;

//返回该局部变量（实例对象）。

})()

var p1=single;

var p2=single;

alert(p1===p2);//弹出true。

</script>

6. 重写构造方式实现单例模式

原理： <script>

function demo(){

alert(1);

demo=function(){

alert(2);

};

};

demo();//弹出1

demo();//弹出2

demo();//弹出2

demo();//弹出2

</script>

过程：第一次执行函数有两步：①弹出1。②改变变量demo所指向的函数体。

<script>

function Person(){

this.age='20';

//给实例对象添加属性。

this.say=function(){

alert('hello')

};

//给实例对象添加方法。

var instance = this;

//声明一个局部变量instance，将Person构造函数的实例对象赋值给该变量。

Person=function(){

return instance;

};

//改变构造函数内部的函数体，**使第一次之后实例对象时直接返回变量instance**。

};

var p1=new Person();

var p2=new Person();

alert(p1===p2);//弹出true。

</script>

**05/03**

1. XMLHttpRequest对象：发送请求到服务器并获得返回结果。
2. aJax使用步骤：①创建一个XMLHttpRequest对象；②使用open方法设置ajax参数；③与服务器建立连接，发送数据到服务端；④在onreadystatechange事件函数中接收服务器数据。
3. **onreadyStatechange事件，当XMLHttpRequest对象的readyState属性值发生变化时，触发该事件。**
4. readyState属性:XMLHttpRequest对象的状态信息

0——XMLHttpRequest对象没有完成初始化。

1——XMLHttpRequest对象开始发送请求。

2——XMLHttpRequest对象请求发送完成，已收到全部响应内容。

3——XMLHttpRequest对象开始读取响应，还没有结束

4——XMLHttpRequest对象读取响应结束。

(0)未初始化—→(1)发送请求—→(2)发送完成—→(3)读取响应—→(4)读取结束

new XMLHttpRequest()—**(0)**→open()—**(1)**→send()—**(1)—(4)**→responseText

1. status属性:XMLHttpRequest对象的状态码

200——服务器响应正常

400——无法找到请求的资源

403——没有访问权限

404——访问的资源不存在

500——服务器内部错误

new XMLHttpRequest()—**(0)**→open()—**(0)**→send()—**(0)—(200)**→responseText

1. responseText获得响应的文本内容；responseXML获得响应的XML文档对象
2. 创建一个服务器，并在页面上输出:"hello world!"

//加载和返回内置的HTTP模块

var http = require("http");

//创建一个http的服务器

http.createServer(function(request, response) {

//向请求的客户端发送响应,其中第一个参数为状态码，第二个参数为设置信息的键值对

response.writeHead(200, {

"context-type": "text/html"

});

//页面上打印的内容

response.write("hello world!");

//结束请求响应

response.end();

//服务器运行在8080端口

}).listen(8080);

//控制台上输出Server running at localhost:8080

console.log("Server running at localhost:8080");

**05/04**

1.<button>按钮</button> //自带跳转功能

<input type="button" value="按钮"/> //无跳转功能。

<input type="submit" value="按钮"/> //自带跳转功能。

1. Ajax——省市联级

<!--[ { "北京": [ "朝阳", "海淀", "昌平" ] }, { "河北": [ "石家庄", "廊坊", "邯郸", "唐山" ] }, { "山东": [ "烟台", "青岛", "泰安", "威海" ] } ]-->

本json一共有三层：数组[—对象{—数组[。

<body>

<select id="sel1">

<option>请选择</option>

</select>

<select id="sel2">

<option>请选择</option>

</select>

<script>

var xhr = new XMLHttpRequest();

xhr.open('GET', 'http://localhost:3000/api/5', true);

xhr.send(null);

xhr.onreadystatechange = function() {

if (xhr.readyState == 4 && xhr.status == 200) {

ojson = JSON.parse(xhr.responseText);

};

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

//进入第一层内部。

for (var j in ojson[i]) {

//进入第二层内部。

**sel1.add(new Option(j, j));**

//将第二层内部的键插入到sel1。

};

};

};

sel1.onchange = function() {

sel2.options.length = 1;

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

//进入第一层内部。

for (var j in ojson[i]) {

//进入第二层内部。

if (this.value == j) {

//判断第二层内部的键是否等于sel1被选项的value值。

for (var n = 0; n < ojson[i][j].length; n++) {

//进入第三层内部。

sel2.add(new Option(ojson[i][j][n], ojson[i][j][n]));

//将第三层内部的值插入到sel2。

};

};

};

};

};

</script>

</body>

1. Ajax库

function ajax(obj) {

var xhr = new XMLHttpRequest();

obj.url = obj.url + '?rand=' + Math.random();

xhr.open(obj.method, obj.url, obj.async);

xhr.send(null);

if (obj.async === false) {

callback();

}

if (obj.async === true) {

xhr.onreadystatechange = function() {

if (xhr.readyState == 4) {

callback();

}

};

};

function callback() {

if (xhr.status == 200) {

obj.success(xhr.responseText); //回调

} else {

alert('数据返回失败！状态代码：' + xhr.status + '，状态信息：' + xhr.statusText);

};

};

};

ajax({

'method': 'GET',

'url': 'a.txt',

'async': **true**,

'success': function(a) {

alert(a)

}

})

注意：执行对象作为参数的函数时调用了该对象的一个方法。

ajax()为一个函数，函数的实参为一个对象，对象有一个success方法。

对象作为实参调用ajax()函数→执行ajax()函数代码→xhr.responseText作为实参调用对象的success方法。

即：对象是ajax()函数的实参，xhr.responseText是对象success方法的实参。

1. Json属性值的访问方式

<script>

var person={a:1,'b':2,'c':function(){alert(3)}}

alert(person.a);//弹出1

alert(person['a']);//弹出1

alert(person[a]);//语法错误

alert(person.b);//弹出2

alert(person['b']);//弹出2

alert(person.'b');//语法错误

person.c();//弹出3

</script>

注意：①无论Json中属性名有无引号，通过对象点属性名访问时属性名不可有引号，通过对象中括号属性名访问时属性名必须加引号。②对象的属性名、属性值以及方法名应该加引号，对象的方法值不可加引号。③若调用对象的方法，则只能用对象点方法名小括号的调用方式。

**05/05**

1. ECMAScript是一种标准和规范，JavaScript和JScript是这种规范的实现和扩展。
2. **ECMAScript5中关于对象(Object)的新方法：**

(1) 获取对象某个属性的描述符：Object.getOwnPropertyDescriptor(对象, “属性名”) 。

(2) 定义或修改对象某个属性的属性特征(**不常用**)：

描述符.value/writable/enumerable/configurable="值"/true/false

Object.defineProperty(对象,"属性名",描述符);

注意：这种方法不常用。若使用，则需要两步：修改描述符—→定义/修改属性

(3) 定义或修改对象某个**自有属性**的属性特征(**常用**)：

Object.defineProperty(对象,"属性名",{

value:"属性值",

writable:true/false,

enumerable:true/false,

configurable:true/false,

});

注意：这种方法只能定义对象**自有属性**的属性特征，而不能定义对象**原型属性**的属性特征。

(4) Object.preventExtensions(对象)—属性**不可**添加、可删除、可修改

Object.seal(对象) —属性**不可**添加、**不可**删除、可修改

Object.freeze(对象) —属性**不可**添加、**不可**删除、**不可**修改

注意：(3)为定义或修改对象**某个属性**的属性特征，(4)为定义或修改对象**所有属性**的特征。定义或修改对象某个属性的属性特征时，有四个可定义/修改项：**属性值**、**可修改**、**可枚举**、**可删除**。定义或修改对象所有属性的特征时，有三个可定义/修改项：**可添加**、**可删除**、**可修改**。

(5) Object.get**Prototype**Of(对象) 得到对象的**\_proto\_**属性值。

**3. ECMAScript5中关于数组(Array)的新方法：**

ECMAScript5中，大部分数组新方法的参数为一个匿名函数，并且数组的每一个元素调用一次该匿名函数，该匿名函数一般有两个参数:**数组元素和索引值**。

(1)数组.forEach():从头到位遍历数组，每个元素都调用指定的函数（该方法的参数），该方法在元素遍历结束之前无法终止。

<script>

var arr=[1,3,5,7,9];

var sum=0

arr.forEach(function(item,index){

sum=sum+item;

});

alert(sum);//弹出25。

arr.forEach(function(item,index){

// item=item+1;语法错误，只能通过arr[index]改变数组元素的值。

arr[index]=item+1;

});

alert(arr[1]);//弹出4。

</script>

注意：数组.forEach()可以取代for循环在数组中的应用，**无返回值(return)**。

(2)数组.map():调用数组的每一个元素传递给指定函数，并**返回**一个数组，它包含该函数的返回值。

<script>

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

var arr1 = arr.map(function(item) {

**return** item \* item;

});

alert(arr1) //弹出1,4,9,16,25

</script>

(3)数组.filter():创建一个新的匹配过滤条件的数组。

<script>

var arr1 = ['Charles', 'Mark', 'Bill', 'Vincent', 'William', 'Joseph']

var arr2 = arr1.filter(

function(item, index) {

**return** item.indexOf("ll") > -1

});

alert(arr2); //弹出Bill、William。

</script>

(4)数组.some()查看数组**是否有**符合条件的元素

<script>

var arr1 =

"Charles,Mark,Bill,Vincent,William,Joseph".split(",");

alert(arr1.some(

function(item, index) {

**return** item.length < 5;

}

)); //true

alert(arr1.some(

function(item, index) {

**return** item.length < 10;

}

)) //true

</script>

(5)数组.every()查看数组元素**是否全**部符合条件。

<script>

var arr1 =

"Charles,Mark,Bill,Vincent,William,Joseph".split(",");

alert(arr1.every(

function(item, index) {

**return** item.length < 5;

}

)); //false

alert(arr1.every(

function(item, index) {

**return** item.length < 10;

}

)) //true

</script>