目录

1.函数

- 函数声明后，才可以调用

```
// 函数的声明 :
function 函数名(形参1,形参2...){
  // 函数体(具体要执行的代码);
}
```

```
// 函数调用 :
函数名(实参);
  // 格式: 函数名(实参1,实参2...)//一般有多少 形参 就有多少 实参一一对应
```

    - 形参：形式上的参数。
    - 实参 : 实际传入的参数。
    - 传参：用实参给形参赋值
        - 不传参 形参就没有值
    - `return` 关键字： `return`后面写什么表达式，函数调用的返回结果就是`return`后面表达式的值。
        - 函数运行的时候，遇到`return`关键字，整个函数会终止。
- 函数的作用：
    1. 使程序变得简洁而清晰
    2. 有利于程序维护
    3. 可以提高程序开发效率
    4. 提高代码的重用性（复用性）
- 封装函数的步骤：
    1. 分析不确定的值
    2. 将不确定值声明形参
    3. 函数名和形参都要见名思意
- 例题 : 求两个数的和

```
function add(num1,num2){  \
  alert(num1 + num2);
}
add(1,2);
add(5,7);
var res = add(2,3);    //给变量赋值不起作用; 因为没有return返回值
```

2.关于函数的练习（答案看4_关于函数的练习）

```
// 1.编写一个函数，计算两个数字的和、差、积、商
//   要求：使用传参的形式
// 2.编写一个函数，计算三个数字的大小，按从小到大顺序输出。
// 3.编写一个函数，输入n为偶数时，调用函数求 1/2 + 1/4 +...+1/n,
    // 当函数为奇数时求 1/1 + 1/3 +...+1/n
```

3.函数的 arguments

- 每一个函数内部都有一个`arguments`，系统内置的。
- `arguments` 是用来存储实际传入的参数。
    - `arguments.length` ： 获取**当前函数传入实参的个数**。
    - `arguments[下标]` 下标是从0开始的。访问对应的数据
- **优先使用形参，除非特殊情况才可以使用arguments**

```
//例子：传入任意个数字的和。 不知道有多少形参，所以使用arguments来获取
function sum(){
  var a = 0;
  for(var i = 0; i < arguments.length;i++){
    a += arguments[i];
  }
  return a;
}
alert(sum(1,20,3));
```

4.函数的作用域。

- 任何程序在执行的时候都要占用内存空间。函数调用的时候也要占用内存空间。
- 函数的垃圾回收机制：调用函数的时候，系统会分配对应的空间给这个函数使用。当函数使用完毕以后，这个内存空间要释放，还给系统。
  - 垃圾回收机制的两个方法：标记清除 ， 引用计数
- 【注】在函数内部声明的变量和形参是属于当前函数的内存空间里的。

```
var a = 2;  //声明在全局的变量叫全局变量，不会随着函数的调用被创建和销毁。
function show(){
  var a = 2;  //声明在函数内部的变量叫局部变量
  a++;
  alert(a);
}
```

  - 内存管理机制：在**函数中声明的变量和形参**，会**随着函数的调用被创建，随着函数的调用结束被销毁**
  - 在函数中声明的变量和形参，有效范围是当前函数（函数的大括号内），作用域为局部作用域。
- 就近原则： 当有同名变量，离哪个作用域近，就使用哪个作用域内的同名变量。
- **在全局作用域中声明的变量、函数都会变成`window`对象的属性和方法。**

5.函数递归。

- 满足以下三个特点就是递归：
  1. 函数自己调用自己
  2. 一般情况下有参数
  3. 一般情况下有`return`
  - 递归可以解决循环能做的所有事情，有一些循环不容易解决的事情，递归也能轻松解决。
  - 递归，都可以写出来，但是不知道为什么是对的。
- 编写递归的方法：
  1. 首先去找临界值，即无需计算，获得的值。
  2. 找这一次和上一次的关系。
  3. 假设当前函数已经可以使用，调用自身计算一次。

```
//利用递归求1~n的和
function sum(n){
    if(n == 1){
        return 1;
    }
    return sum(n - 1) + n;
}
alert(sum(100));
```

  - 传入100;`if`语句不执行，执行`return sum(100 - 1) + 100`,`sum(99)`的值不知道，又要开辟一块内存求`sum(99)`,又执行`return sum(99 - 1) + 99`,`sum(98)`的值也不知道，又要开辟一块内存求`sum(98)`..... 一直到`sum(1)`,可以进入`if`，返回值`1`，关闭`sum(1)`分配的内存，然后`sun(2)`;也能求出来，返回值给`sum(3)`,关闭`sum(3)`分配的内存...直到返回到`sum(100)`,才会关闭所有开辟的内存，**占用率大**，如果求的过程更复杂，容易崩溃。

- 递归练习(答案看8_递归练习.html)
  1. 斐波那契数列
     兔子繁殖问题，设有一对新生兔子，从第四个月开始它们每个月月初都生一个兔子，新生兔子的第四个月月初开始，又每个月生一对兔子。
     按此规律，假设兔子没有死亡，n(n <= 20)个月月末共有多少对兔子。

6.数组

- 数组：用一个变量去存储一堆数据的数据结构。
- 数组声明 : 3种

```
// 1.通过new创建数组
var arr = new Array();

// 2.省略new运算符创建数组
var arr = Array();
// 【注】上述两种方法,传入参数只有一个数字,直接声明这么长一个空数组。
  var arr = Array(10);  //声明了一个长度为10的空数组，数组中没有数据
  alert(arr);           //输出结果为  ,,,,,,,,,  九个逗号

  //可以给指定的第几位元素进行赋值
  arr[2] = 3;

// 3.数组常量进行赋值。(JS一般使用中括号[];)  -->  多使用这个方式
var arr = [1,3,"hello"];
```

  - 声明 0-99的数组

```
var arr = Array.from(new Array(100).keys())
```

  - `数组.length` 返回数组中【元素】的个数。
    - 元素：将数组中存储的每一个数据，叫做数组的元素。
    - 访问数组的元素：

```
arr[0]; arr[1]; 数组[下标]
// 【注】下标是从0开始的。
```

- 组和循环是天生一对。

```
var arr = [100,true,"hello"];        //定义数组
alert(arr.length);                   //输出当前数组元素个数
for(var i = 0; i < arr.length; i++){
  document.write(arr[i] + "<br/>");    //打印数组每个元素。
}
```

- 数组的练习(答案看10_数组练习.html)

```
// 关键字 : Math.random()   随机[0,1]   永远取不到1
//随机0~9的整数 : parseInt(Math.random() * 10)

// 以下通过循环给数组每个元素赋值，随机数。
    var arr = Array(10);      //创建长度为10的空数组
    for(var i = 0; i < arr.length; i++){
        arr[i] = parseInt(Math.random() * 10);
    }
    document.write(arr);
```

7.数组的遍历

- 遍历：比如点名时，全班同学都报出自己的名字
- 在页面上分别将每一个数输出
  - `for` 循环 遍历

    ```
    for(var i = 0; i < arr.length; i++){
      document.write(arr[i] + " ");
    }
    ```

  - `for...in` 遍历

    ```
    for(var i in arr){
      document.write(arr[i] + " ");
    }
    ```

  - for循环遍历 和 for...in快速遍历的区别
    - `for` 循环需要判断
    - `for...in` 不需要判断
      - `for...in` 循环会把某个类型的原型(`prototype`)中方法与属性给遍历出来

8.数组的方法

- 栈结构：从同一头进，从同一头出。
  - 栈：木盆，比如衣服最开始放入的，最后才能拿出来，最后放入的，拿出时，开始就能拿到
  - 特点：**先进后出**

- 数组两个方法形成栈结构
  - `push()`：给数组的末尾添加元素。

```
arr.push(参数1，数组...);  //插入数组，不会被拆开，会形成二维数组
```

    - 返回值：插完元素以后数组的长度。

```
var arr = Array("北京","上海","广东");
arr.push("深圳","南京","厦门");
alert(arr);
alert(arr.push);         //返回值为6

var res = arr.push("深圳","南京","厦门");
alert(res);              //返回值也是6
```

    - `push`方法不会像`concat`一样，把数组拆分再插入。
  - `pop()`：从数组末尾取下一个元素

```
arr.pop();  // 无参数
```

    - 返回值：取下一个元素

```
var arr = Array("北京","上海","广东");
var res = arr.pop();  //拿取pop的返回值
alert(res);           //输出pop返回值    结果-广州
alert(arr);           //输出arr数组      结果-北京，上海
```

- 队列结构：从末尾进，从头部出。
  - 特点：先进先出
  - `unshift()`：从数组的头部插入元素。

```
arr.unshift(参数1,参数2...);
```

    - 返回值：插完元素以后数组的长度。

```
var arr = ["唐朝","元朝","清朝"];
var res = arr.unshift("汉朝"); // ["汉朝","唐朝","元朝","清朝"]
```

  - `shift()`：从数组的头部取下一个元素

```
arr.shift(); // 没有参数
```

    - 返回值：取下的元素

```
var arr = ["唐朝","元朝","清朝"];
var res = arr.shift();
alert(res);   // 返回值为 "唐朝"
alert(arr);   // ["元朝","清朝"];
```

9.数组的方法2

- `arr.concat()` **不会改变原数组**
  1. 拷贝原数组，生成新数组。
  2. 合并数组。

```
arr.concat(数组,数据,....)
```

  - 返回值：合并成的新数组。原数组不会被改变。
  - **传入数组，数组中的元素要单独拆出来再进行合并。**

```
var arr1 = [10,20,30];
var arr2 = [40,50,60];
var newArr = arr1.concat(arr2,"hello",true);
alert(newArr);      //结果 10,20,30,40,50,60,hello,true
alert(newArr.length);//8
alert(arr1);        //结果10,,20,30
```

- `arr.slice()` **不会改变原数组**
  - 获取当前数组指定区域的元素 [start,end) , 提取出元素生成新数组

```
arr.slice(start,end);    [start,end)  不包含end
```

  - 省略 `end` ,会从 `start` 处复制到结束位置
  - 返回值：生成新数组，原数组不会发生任何改变。
- `arr.splice()` **会改变原数组**

```
arr.splice(start,length,数据1,数据2....)
```

  - `start` 开始截取的位置
  - `length` 截取元素的长度
  - 第三个参数开始：在start位置插入的元素。
  - 当**参数只有一个数值时，会从数值下标位置向后截取全部**
  - 返回值：截取下来的元素组成的数组

  - 增加 不截取元素，添加元素

```
var arr = [10,20,30,40,50,60];
var res = arr.splice(1,0,"hello","world");
alert(arr);     //结果 10 hello world 20 30...60
alert(res);     //结果为空，因为没有截取的值
```

  - 删除 不添加元素，截取元素

```
var res = arr.splice(1,2);
alert(arr);     //结果 20,30被删除
alert(res);     //结果为20,30
```

- 修改（先删除再增加）

```
var res = arr.splice(2,1,"hello");
alert(arr);     //结果  30被换成hello
alert(res);     //结果为30
```

- `arr.join()` **不会改变原数组**
  - 将数组中的元素，用传入的拼接符，拼接成一个字符串。

```
arr.join("元素拼接符号")
```

  - 返回值：拼接好的字符串。不会改变原数组。

```
var arr = [10,20,30];
var str = arr.join("--");
alert(str);     //结果 10--20--30
alert(arr);     //结果 10,20,30
```

- `arr.reverse()` 逆序 **会改变原数组的排序方式**

```
var arr = [10,20,30,40,50,60];
arr.reverse();
alert(arr);     //结果 60,50,40,30,20,10
```

- `arr.sort()` **会改变原数组**

```
arr.sort(function(){})
```

  - 默认将数组从小到大排序。是按照字符串进行比较大小的，逐一比较，比较出大小就判定
  - 从小到大排序

```
var arr = [10,5,30,1,3,20];
arr.sort(function(value1,value2){
  return value1 - value2;
})
alert(arr);
```

  - 从大到小
    - 只需要把 `return value1 - value2;` 改为 `return value2 - value1;`
  - 箭头函数 `sort` 优化 (函数体只有一个return的情况下，不用加{})

```
arr.sort( (a,b) => a - b)
```

- 数组的练习
  - 定义一个含有30个整型的元素的数组，按顺序分别赋予从2开始的偶数；然后按顺序，每五个数求出一个平均值，放在另一个数组中并输出，试编程。

10.引用数据类型。

- 运行程序：
  1. 准备运行程序要用的空间(一旦分配好以后，内存大小无法进行改变)
  2. 开始运行程序
- 数组存储的方式：
  - 数组被系统分成2部分
    1. **堆**：数组的**数据存放在堆**里
    2. 程序运行段(**栈**)：数组**地址放在栈**，地址**指向对应的堆内存**

```
//创建一个堆(堆的编码为666)，在栈创建一个地址指向 堆666 的arr1的数组.
var arr1 = [10,20,30,40];
var arr2 = arr1;    //在程序运行段创建一个编号为666的arr2的数组
arr2.push(50,60);   //在编号为666的堆中插入新元素。
alert(arr1);    //10,20,30,40,50,60
alert(arr2);    //10,20,30,40,50,60
```

    - 数组的变量存储的是数组的地址(堆的编号)。
    - `=` 赋值运算符是属于浅拷贝，简单数据类型拷贝原数据，复杂数据类型拷贝栈内地址
- 不想让两个数组公用一个堆，使用`concat()`可以浅拷贝(只能拷贝一层)

```
var arr1 = [10,20];
var arr2 = arr1.concat();   //这样就是两个堆，每个数组对应一个。
arr2.push(30,40);
// 拷贝二维数组行不通
```

11.声明提升

- 内存分配，一次分配，不可更改
- 预编译：在所有代码运行之前，计算机将代码从头到尾看一遍。将这个程序需要的空间一次性分配好。
- 声明提升：在当前作用域，声明变量和函数，会提升在整个 代码 的最前面运行。
  - 局部作用域，声明变量和函数，会提升在整个 局部作用域 的最前面运行。

```
alert(num);     //undefined
var num = 10;
alert(num);     //10

// 上述代码在执行时的真实情况
var num;        //声明提升
alert(num);
num = 10;
alert(num);
```

    - 变量的声明提升：只提升声明的变量，不提升变量的赋值。
    - 函数的声明提升：整个函数都会提升。
    - 函数提升优先级比变量提升要高。

```
// 同名函数和变量，总是 函数提升要大
console.log(a) -> a为函数
var a = 3;
function a(){
    alert(1)
}
```

- 【注】一个script标签也是一个作用域

12.省略var声明变量 （属于语法错误，不建议使用）

- 省略var，直接去强制给一个变量赋值，这个变量会被JS强制声明为全局变量。
  - 【注】不建议，属于语法错误。

13.二维数组

- 数组储存数据，数组中的每一个元素，元素可以是任意的数据类型。
- 二维数组：人为起的,不是官方概念。

```
var arr1 = [10,20,30];
var arr = [true,100,"hello",arr1];
alert(arr.length);  // 4
alert(arr[3]);      // [10,20,30]

alert(arr1[1]);     // 20
alert(arr[3][1]);   // 20 二维数组访问下标的方式
```

- 练习：通过循环按顺序执行一个5x5的二维数组a，赋值1到25的自然数，然后输出该数组的左下半三角，试编程。

14.冒泡排序(具体看19_冒泡排序.html)

- 规则：前后两个数两两进行比较，如果符合交换条件，就交换位置。
- 规律：冒泡排序的每一轮排序，都可以找出一个较大的数放在正确的位置。
- 分析:
  - 比较的轮数 = 数组长度 - 1;
  - 每一轮比较的次数 = 数组的长度 - 当前的轮数;

```
var arr = [9,8,7,6,5,4];

for(var i = 0; i < arr.length - 1; i++){
  //每一轮比较次数
  for(var j = 0;j < arr.length - (i+1);j++){
    if(arr[j] > arr[j + 1]){
      var tmp = arr[j];
      arr[j] = arr[j + 1];
      arr[j + 1] = tmp;
    }
  }
  document.write("<br/>");
}
alert(arr);
```

15.选择排序(打擂台法)

- 规则：选出一个位置，这个位置上的数，和后面所有的数进行比较，如果比较出大小，就交换两个数的位置。
- 规律：每次都能选出最小的数放在正确的位置

```
for(var i = 0; i < arr.length - 1; i++){
  for(var j = i + 1;j < arr.length; j++){
    if(arr[i] < arr[j]){    //从大到小排序
      var tmp = arr[i];
      arr[i] = arr[j];
      arr[j] = tmp;
    }
  }
}
```

16.数组练习 (21_数组练习.html)

1. 随机给出一个五位以内的数，然后输出改数有多少位，分别是什么。
2. 有一个从小到大排好序的数组，现输入一个数，要求按原来的规律将它插入数组中。
   [2, 3, 4, 56, 67, 98] //63
3. 编写函数map(arr)把数组中的每一个数字都增加30%。
4. 编写函数has(arr , 60) 判断数组中是否存在60这个元素，存在返回true，不存在返回false
5. 生成13位条形码(对之前的知识综合练习)具体看 21_数组练习.html