Javascript05_函数

如果需要多次使用同一段代码，可以把它们封装成一个函数。

函数的概念

具有特定功能的代码段（完成特定功能的一段代码）

函数的作用

- 可以将完成某一个特定功能的一段代码实现封装，减少代码的冗余
- 可以实现某一个功能被重复使用，提高代码的执行效率
- 函数可分为系统函数和自定义函数

  系统函数：JS内置函数

  parseInt()

  parseFloat()

  isNaN()

  自定义函数：用户定义的函数

函数的定义

```
function 函数名(参数列表){
    函数体;
}
function --函数定义的一个标志，代表后面是一个函数。function是一个函数从无到有的定义过程
函数名--为函数起一个名字，方便以后调用该函数
参数列表--函数被调用时用来存储调用者传输的一些数据，让函数使用这些数据去完成预定的操作。
函数体--完成指定功能的一段代码
```

```
// 定义一个函数输入1-n之间的数字
function outPut(n){
    for(var i = 0; i < n; i++){
        console.log(i+1);
    }
}
```

注意

1、函数名在定义时使用小驼峰结构，即如果函数名由多个单词组成，除第一个单词首字母小写外其余单词首字母必须大写

2、函数名在定义是必须是由字母，数字，下华线组成没并且数字不能开头

3、函数名定义是一般遵循函数名能够反馈函数功能的标准

函数的调用

- 函数调用时是通过: 函数名(参数列表);

```
outPut(100); //输出1-100之间的数字
outPut(200); //输出1-200之间的数字
```

> 注意
>
> 函数定义过程代码不会执行。只有当函数被调用时，函数中的代码才会执行

函数的返回值

- 函数最终的运算结果如果需要返回给调用者，此时可以通过return关键字完成结果的返回，可以把数据用return语句返回给调用函数的地方。

```
// 定义一个生成一个随机数,随机范围[5, 15]
function getRandom(){
    var num = Math.floor(Math.random()*11+5);
    return num;
}
// 调用函数
var result = getRandom();
console.log(result);
```

> 注意
>
> return是函数调用结束的标志，可以用来终端函数执行，即函数调用过程中如果碰到return关键字，即使return关键字下面还有代码，程序都不再执行。因此return必须放在函数所有代码的最下边

函数的分类

- 无参数无返回值

```
// 输出1-100之间的所有数字
function outPut1(){
    for(var i = 0; i < 100; i++){
        console.log(i);
    }
}
outPut1(); //该函数调用时即没有参数也没有返回值，此时函数称为无参无返回值的函数
```

- 有参数无返回值

```
// 定义一个函数输入1-n之间的数字
function outPut(n){
    for(var i = 0; i < n; i++){
        console.log(i+1);
    }
}
outPut(200); //该函数调用时有参数但是没有返回值，此时函数称为有参无返回值的函数
```

- 无参数有返回值

```
// 定义一个生成一个随机数,随机范围[5, 15]
function getRandom(){
    var num = Math.floor(Math.random()*11+5);
    return num;
}
// 调用函数
var result = getRandom(); //该函数被调用时存在返回值但是没有参数，此时函数被称为无参有返回值的函数
console.log(result);
```

- 有参数有返回值

```
// 定义一个函数生成一个任意闭区间的随机值
function getRandom1(min, max){
    var num = Math.floor(Math.random()*(max-min+1)+min);
    return num;
}
// 调用函数
var result = getRandom1(10,15); //函数调用时即存在参数又存在返回值，此时函数被称为有参数有返回值的函数
console.log(result);
```

函数参数

- 实参和形参的区别
  - 实际参数（实参）是指函数在调用过程中传递的参数，形式参数（形参）是指函数在定义过程中所设置的参数。形参只是一个形式，代表函数在调用时需要传递参数，但是形参无法决定参数的数据类型和数据值的大小。实参才能决定对应数据的大小和类型。
  - 注意：实参数据传递给形参时默认执行的值拷贝操作即将实际参数中的数据拷贝一份给形参，因此在函数内部修改形参数据并不会影响实际参数的数据

```
//在函数increment内部对形参number数据更改并不会影响实参num的数据
var num = 20;
// number属于形式参数（形参）
function increment(number){
    number++;
    console.log(number);
}

increment(num); //num属于实际参数（实参）
console.log(num);
```

- 数组作为函数参数
  - 数组作为实参，将数据传递给形参时执行的是址（地址）拷贝，此时在函数内部通过形参修改数组中数据，相当于直接修改实参中数组的数据

```
 //定义一个函数完成对数组生成n个[5,15]随机值
var array = [];
function getRandomArray(arr, n){
    for(var i = 0; i < n ; i++){
        var num = Math.floor(Math.random()*11+5);
        arr.push(num);
    }
        console.log(arr);
}

getRandomArray(array, 5);
console.log(array);
// 当实参array中存储一个数组时，此时array将数据拷贝给形参arr时，是将其中存储的数据的地址拷贝给了形参arr，此时形参通过地址访问数组对数组进行数据更改，等价于通过array访问数据，即影响array中存储的数据
```

函数作为另一个函数的参数--回调函数

- arguments对象

  当函数无法确定参数个数的时候，可以使用arguments来获取
  arguments是函数内置的对象，存储了调用函数时传的所有实参

```
function testFn() {
    console.log(arguments)
 }

testFn(10,20,30)
```

回调函数

应用场景：例如我们有这样一个需求，我们想要封装一个函数这个函数不但要创建一个随机数组，还要把这个数组的最大值输出。但是随后需求变了，不是输出最大值，是输出最小值，让你修改这个函数，你只能把函数中判断最大值的代码改成判断最小值的代码，但是过了一周，需求又变了，想要你再改回输出最大值，你只能再次修改这个函数的部分代码。

改完之后你开始思考：如果之后需求又改了，要求平均数，我还得该这个函数啊！我可以把这个函数中的除了创建数组的那部分代码，把之后求最大值或最小值或平均值的，重新封装一个单独的函数，在原函数中调用这些名字就好了，每次改就只需要改个函数调用的名字就好，简单多了。

虽然方便多了，但是需求一改我还需要去修改原函数的内部代码，虽然只改一个名字，但是还是不方便，如果我把这个函数名字放在这个函数调用的时候再去定义是不是改了需求就不用该函数内部了。所以我们把函数的名字在调用时作为参数传进来就好了呀！即使你传的不是函数名，是匿名函数本身也可以。

某一个函数A作为函数B的一个参数被传入函数B中，在函数B被调用是可以使用函数A中的代码，此过程中函数A被称为函数B的回调函数

```
 //定义一个函数完成数组中最大值的获取
function getMax(arr){
    var max = arr[0];
        for(var i = 1; i < arr.lenght; i++){
                if(max < arr[i]){
                    max = arr[i];
                }
        }
```

```
        return max;
}

//定义一个函数产生n个[10,30]之间的随机值的数组，并且输出该数组中的最大值
function operate(n, callBack){
              var array = [];
              for(var i = 0; i < n; i++){
                  var num = Math.floor(Math.random()*21+10);
                  array.push(num);
              }

              var max = callBack(array);
              console.log('数组最大值为'+max);
}

operate(10, getMax);
```

//函数getMax被作为函数operate的参数传入到operate内部，在operate内部合适的时间完成函数getMax的调用，此时函数getMax被称为函数operate的回调函数

- 回调函数的作用
  - 回调函数可以实现对某一段公共代码的功能的扩充。

```
// 回调函数的作用，比如数组排序
var arrary1 = [1,2,'4', 1, "23", 2, 3];
function sortBy(num1, num2){
        num1 = parseInt(num1);
        num2 = parseInt(num2);
        // 返回差值如果是正数则交换相邻两个位置的数据，如果为负数则不交换相邻两个位置的元素
        return -(num1 - num2);
}
// 注意数组排序过程中sort接收一个回调函数，该回调函数有两个参数，分别存储数组中相邻两个位置的元素数据
arrary1.sort(sortBy);
console.log(arrary1);
```

使用回调函数的场景

- 当我们发现某一段代码在开发中大量出现，但是该段代码中有一小部分每一次不太一样，此时这一小部分代码可以作为回调函数放到这一大段代码对应的函数内部。

变量作用域

- 全局变量：定义在函数外部的变量被称为全局变量，全局变量的作用域：从变量定义开始直到程序运行结束，任意位置都可以访问到全局变量

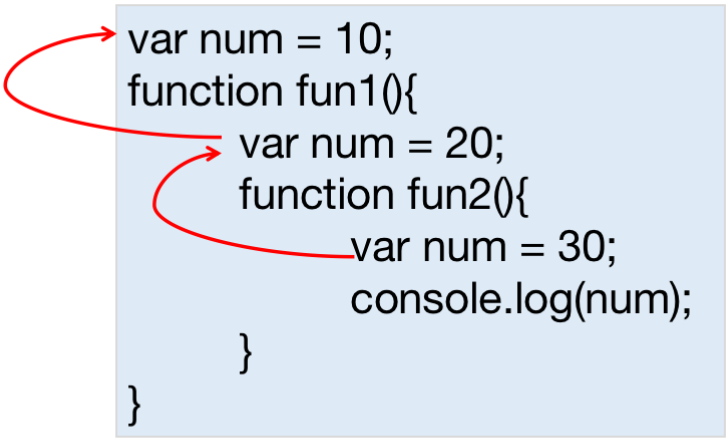
```
var a = 10;
function outPut(){
    console.log(a);
}
outPut();
//变量a定义在函数外部，此时变量a被称为全局变量。全局变量可以在任意位置被访问到
```

- 局部变量

```
function outPut(a){
    console.log(a);
    var b = 10;
    console.log(b);
}
outPut(10);
//变量a和变量B均被定义在函数内部，此时变量a和变量b被称为局部变量。注意局部变量只能被当前函数使用，函数调用结束以后局部变量会被系统自动回收
```

在开发过程中一般遵循首先使用局部变量，如果需要多个函数之间共享数据，此时才会使用全局变量。

- 作用域链



只要是代码，就至少有一个作用域
函数内部为局部作用域
内部函数可以访问外部函数的变量
但是外部函数无法访问内部函数的变量
内部函数访问外部函数中的变量是，采取的是链式查找的方式来决定取哪个值，这种方式称之为作用域链（就近原则）

JS特殊的函数

匿名函数

匿名函数，即函数没有名字

```
 // 匿名函数
var fun = function (){
    console.log('你好世界');
}
fun();
//匿名函数即函数没有名字，此时为了使用该函数，可以通过定义变量存储该函数，用变量名代替函数名执行
```

```
(function(){
    console.log('你好世界');
})();
//函数在定义完成以后直接调用该定义函数
```

```
 var timer = setInterval(function(){
        console.log('你好世界');
}, 200);
```

> 注意：匿名函数在使用过程中相对比较简单灵活

IIFE(立即执行函数)

IIFE: Immediately Invoked Function Expression，意为立即调用的函数表达式，也就是说，声明函数的同时立即调用这个函数。

对比一下，这是不采用IIFE时的函数声明和函数调用：

```
function foo(){
  var a = 10;
  console.log(a);
}
foo();
```

下面是IIFE形式的函数调用：

```
(function foo(){
  var a = 10;
  console.log(a);
})();
```

函数的声明和IIFE的区别在于，在函数的声明中，我们首先看到的是function关键字，而IIFE我们首先看到的是左边的（也就是说，使用一对（）将函数的声明括起来，使得JS编译器不再认为这是一个函数声明，而是一个IIFE，即需要立刻执行声明的函数。
两者达到的目的是相同的，都是声明了一个函数foo并且随后调用函数foo。

递归函数

- 递归函数：函数内部调用一个和当前函数名字相同的函数，该过程被称为递归函数。
- 注意：递归函数很容易出现死循环，并且递归函数逻辑结构复杂，能够使用递归函数解决的问题基本都可以使用循环解决，综上递归函数不建议大量使用。

```
// 求5的阶乘 5*4*3*2*1
function jiecheng(n){
    if(n == 1){
        return 1;
    }else {
        return n*jiecheng(n-1);
    }
}

console.log(jiecheng(10));

//递归函数：函数内部调用了一个和该函数名字相同的函数去执行下一刻任务。
//注意:递归函数任务的执行分为两部分，分别是任务的分发和任务的执行，首先递归函数第一次被调用，函数会将需要完成的任务分成两部分，一部分分发给同名函数，另一部分处理。最后函数任务的执行从最后一个接收任务的函数开始。所以递归函数必须有返回值
```

函数注释

```
/**
 * 求两数之和
 * @param  {number}  num1    被加数
 * @param  {number}  num2    加数
 * @return {number}  和
 * @author Andy
 * @version 1.0
*/
  function sum(num1,num2){
  return num1 + num2;
  }
```

符号	用法
@param	@param {类型} 参数名 描述
@return	@return {类型} 描述
@author	@author 作者
@version	@version 版本号