day04 函数

函数概述

函数相当于一个代码空间，他里面可以存储一些代码片段，一般我们会将一些功能性代码抽取放入到函数内，这样的操作就是**封装**。核心就是利用函数来减少冗余代码的出现，形成对应的**复用**。

函数的分类

- 系统函数 （系统本身就写好的 你只需要调用 console.log() 属于全局窗口的 window）
- 内置函数 （对象内部提供的 Math.pow ）
- 自定义函数 (由自己定义的函数)

函数的定义

使用function关键来定义

定义的方式有三种

匿名函数 （没有名字的函数 无法被复用的）

```
//自执行函数 自己执行 准备工作的执行（框架的封装）
//  前面() 表示他是一个整体 后面() 表示执行这个函数
(function () {
    console.log('我是自执行函数')
})()
```

具名函数 （有名字的函数 必须调用才会执行 具备复用性）

```
/*
function 函数名(形参...){
函数体
}
*/
//具名函数
function sayHello(){
    console.log('hello')
}
//调用 函数名()
sayHello()
//结合匿名函数来声明具名函数
var sayHi = function(){
    console.log('hi')
}
sayHi()
```

使用对象构建的形式 （new关键来构建）

```
// new Function('函数体')
var fn = new Function('console.log("你好")')
fn()
```

函数的执行过程

预编译过程

var关键修饰的变量会预编译

```
console.log(a) //undefined
var a = 10
console.log(a) //10
```

function 也会发生预编译

```
fn() //也能执行 function会被预编译
function fn() {
    console.log('测试')
}
console.log(fn1)//undefined
fn1()//var关键词预编译不会读取赋值操作 报错 is not a function
//第二种具名函数的定义
var fn1 = function(){
    console.log('test')
}
```

执行过程

函数声明会发生预编译 调用的时候会发生什么操作

他会去寻找对应的堆空间的函数引用

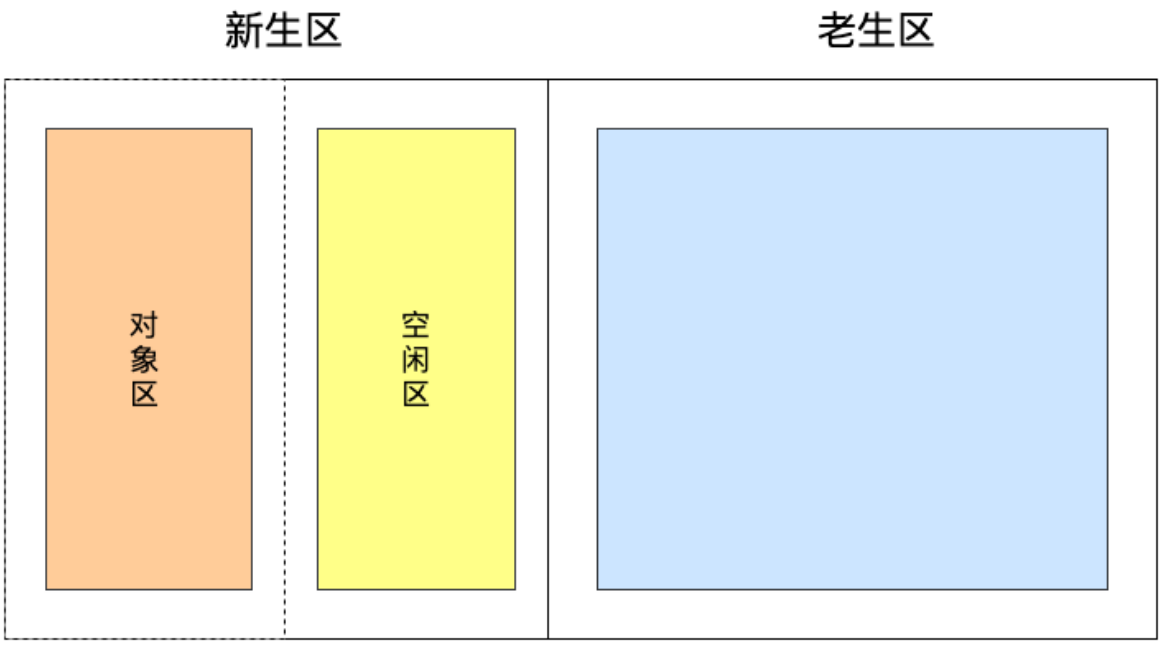
然后再将他推入执行栈中 进行执行

在执行栈中会打开对应的函数内的代码进行执行

当他执行完毕以后 那么gc就会将他回收（可达性）。

回收过程

gc是一个垃圾回收机制 （用于回收当前没有被使用的变量）



回收器

主回收器 Major gc (常用的 回收大的对象 明显)

副回收器 Minor gc （回收碎片化内容 使用频繁）

回收机制

标记清除 （设置true false标记）

引用计算 （引用操作进行++）

函数的参数

- 没有实际的参数叫形参 用于定义 (随便写)
- 有实际值得参数叫实参 用于传递

示例

求和 传入俩个数值求他们之间的和

```
//a b为形参
function sum(a,b){
    console.log(a+b)
}
sum(1,2) //1 2为实参
```

注意事项: js允许少传参 必须这个参数没有被使用到（没有函数的调用） 否则会出现错误

练习

传递一个数 判断是否回文

```
function isPalindrome(n) {
    //判断当前是否为数值
    if(isNaN(Number(n))){
        //错误提示
        // console.error('当前输入的内容出错')
        //抛出一个新的错误 底下的内容不会被执行
        throw new Error('当前输入的内容出错')
    }
    //个数为0不可能是回文数
    if(n%10==0 && n!=0){
        console.log('这个数不是回文数')
    }else if (n < 10 && n >= 0) { //小于10 大于0的数为回文
        console.log('当前数为回文数')
    } else {
        //需要将当前这个数进行反转
        var x = n
        var y = 0
        //反转操作
        while (x > y) {
            y = y * 10 + x % 10
            x = parseInt(x / 10)
        }
        //偶数情况下
        if (x == y) {
            console.log('当前数为回文数')
        } else if (x == parseInt(y / 10)) {
            console.log('当前数为回文数')
```

```
        } else {
            console.log('这个数不是回文数')
        }
    }
}
isPalindrome(prompt('输入数值进行判断'))
```

传入一个数判断是否是水仙花数

```
function isNarcissusNumber(number){
     //判断当前是否为数值
    if(isNaN(Number(n))){
        //错误提示
        // console.error('当前输入的内容出错')
        //抛出一个新的错误 底下的内容不会被执行
        throw new Error('当前输入的内容出错')
    }
    //验证

 if(Math.pow(number%10,3)+Math.pow(parseInt(number/10%10),3)+Math.pow(parseInt(n
umber/100),3)==number){
        console.log('当前数为水仙花数')
    }else{
        console.log('当前数不是水仙花数')
    }
}
isNarcissusNumber(prompt('输入数值进行判断'))
```

函数考虑其复用性的同时，必须要考虑多种情况的产生

return关键词

return 是用于在函数中返回对应的结果的，默认情况下函数return undefined。**当return完那么对应的函数执行就结束了。**

```
function fn(){

}
console.log(fn())//undefined
```

示例

输入俩个数返回他们的和

```
function sum(number1, number2) {
    return number1 + number2
    console.log('我是后面的代码')//不会被执行
}
var i = sum(1, 2)
console.log(i) //3
```

练习

输入俩个数返回他们的最小公倍数和最大公约数的和

```

```

```
function fn(x, y) {
    //得到他的最小公倍数 和 最大公约数
    //得到最大最小值
    var max = Math.max(x,y)
    var min = Math.min(x,y)
    while (true) {
        if (max % x == 0 && max % y == 0) {
            //得到最小公倍数
            break
        }
        max++
    }
    while (true) {
        if (x % min == 0 && y % min == 0) {
            //得到最小公倍数
            break
        }
        min--
    }
    return max + min
}
```

总结

- break 跳出循环和switch 不会结束function
- continue 只能用于循环中 跳过本次循环 进入下次
- return 结束整个函数 返回对应的数据 （放在最后）
- throw new Error 结束整个程序

arguments

arguments是一个参数列表,参数列表其实是一个**伪数组**（伪装的数组 有数组的一些特性 但是不是数组（不具备数组的方法））。他可以用于获取所有的参数（传递的参数）

```
function fn(a, b, c) {
    console.log(arguments) //[1,2]
    //arguments 长度属性 length （实际传入的参数个数） callee 指向的是当前的方法
    console.log(arguments.length)
    //访问 arguments里面的参数 使用对应的下标来访问 下标从0开始 到length-1结束
    console.log(arguments[0])//获取第一个参数 1
    console.log(arguments[1])//获取第二个参数 2
}
fn(1, 2)
```

arguments的属性及方法

- length属性 用于获取对应的传入参数个数
- callee 方法 指向当前的函数

arguments访问对应的参数使用下标访问

- 下标从0开始 到 arguments.length-1结束
- 0表示是第一个参数 那么5表示第六个参数 x表示x+1个参数
- 省略对应的形参 直接传入实参 在函数中使用arguments来接收的对应的实参

示例

传入不定的参数 计算他们之间的和

```
function sum(){
    var result = 0
    //arguments会接收所有的实参 那么我们就可以通过循环遍历对应的参数进行相关操作
    for(var i=0;i<arguments.length;i++){
        result+= arguments[i]
    }
    return result
}
console.log(sum(1,2,3,4,5,6,7))
console.log(sum(1,2))
```

作用域及作用域链

作用域概述

一个变量的作用范围称为作用域，作用域主要划分为**全局作用域**（全局可用），**局部作用域**（局部可用又为**函数作用域**）

示例

```
var a = 10 //全局作用域
function fn(){
    var a = 20 //局部作用域
    var b = 30
}
fn()
console.log(a)//10
console.log(b) //b is not defined
```

在全局中不能访问局部作用域的变量

题目

```
console.log(a)//undefinde
var a = 10
function fn(){
    // a = undefined
    a = 20 //局部变量赋值
    console.log(a) //局部变量打印
    var a = 30 //局部作用域 var修饰关键进行预编译
}
function fn1(){
    a = 40 //全局作用域
    var b = 30
    console.log(a+b)
}
function fn2(){
    //预编译
    console.log(a) //undefined
    var a = 50
}
fn1()//70
console.log(a)//40
fn()//20
```

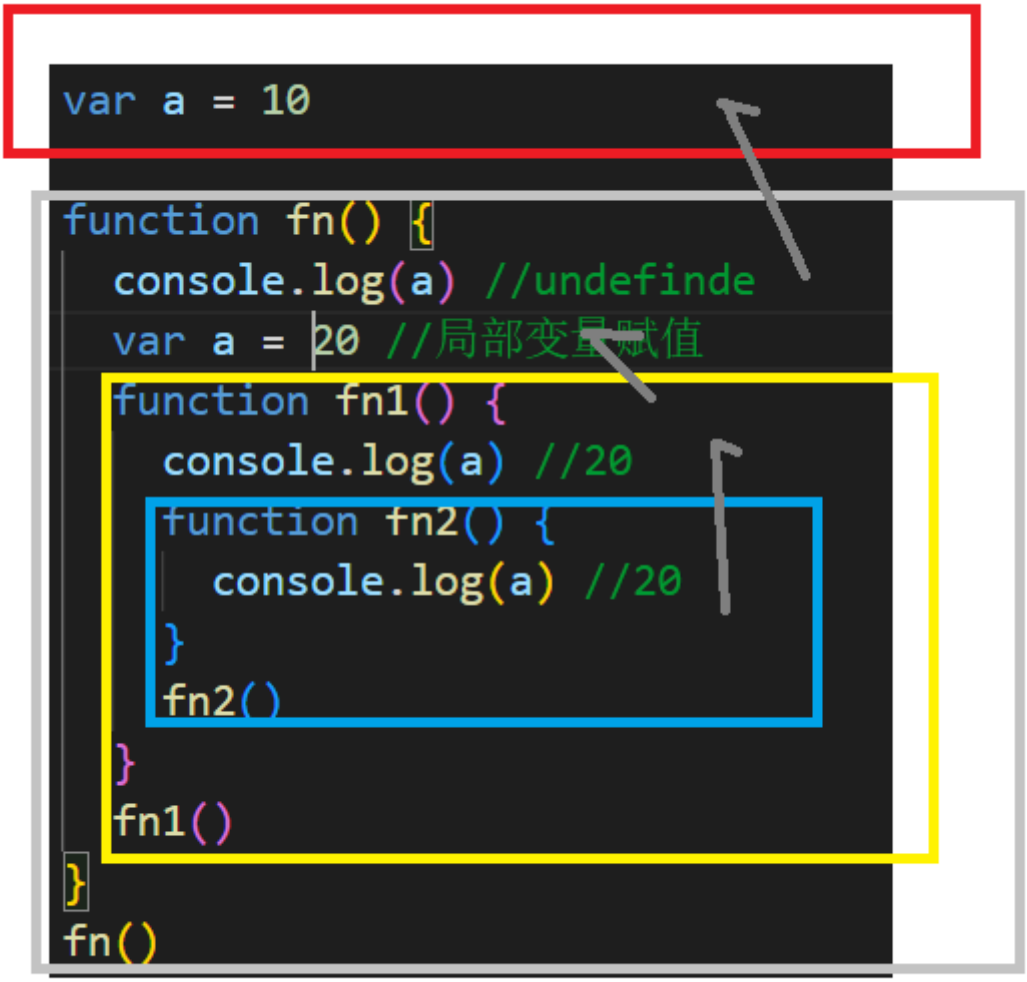
```
console.log(a)//40
fn2()//undefinde
console.log(a)//40
```

- var 关键词会进行变量提升
- 只要是在function中使用var关键词声明那么这个变量就是局部变量 那么在这个里面使用到所有这个变量都是指向这个局部变量
- 如果在function中没有使用var关键词声明那么这个变量就是全局变量

作用域链

作用域链就是逐层向上查找对应的作用域（变量声明）形成的链子，如果没有找到那么就会报错。

```
var a = 10
function fn(){
    console.log(a)//undefinde
    var a = 20 //局部变量赋值
    function fn1(){
        console.log(a) //20
        function fn2(){
            console.log(a)//20
        }
        fn2()
    }
    fn1()
}
fn()
```




```
var x = 1;
function f(x, y = function () { x = 3; console.log(x); }) {
  console.log(x)//undefinde
  var x = 2
  y()//3
  console.log(x)//2
}
f()
console.log(x)//1
//undefinde 3 2 1
```

事件驱动

事件驱动就是说通过触发一个行为执行对应的函数，这个被称为事件驱动

获取输入框的值

html准备

```
  <input type="text" id="input"><button id="btn">点击按钮</button>
```

当点击按钮的时候打印输入框的值

1.获取按钮

```
var btn = document.getElementById('btn')
```

2.点击按钮 触发对应的事件 来执行对应的函数 (不需要手动调用的)

```
btn.onclick = fn
```

4.在函数内打印input框中的内容

- 获取输入框
- 拿到输入框的值

```
function fn(){
    //获取输入框
    var input = document.getElementById('input')
    //获取输入框的值
    var value = input.value
    console.log(value)
}
```

简单的dom操作

- 通过id获取对应的dom元素（标签）
- document.getElementById('id名字')

递归 （Ologn）

递归是一个算法，算术其实就是固定的套路，递归算法是为了降低时间复杂度提高效率所设计的算法，他可以完成所有循环可以做的事情。

递归的用途 (可以在不知道层级的情况下走到底)

- 文件目录遍历
- DFS查找
- 多级对象分析合并
- 深拷贝
- ...

递归的流程

- 初始值 （不变的值）
- 规律
- 自己调自己

示例

```
function 函数名(参数){
    if(条件){
        初始值 进行返回
    }else{
        规则值 返回值 自己调用自己
    }
}
```

求1+100的和

```
//n表示次数 1次 值为1 2次 值为3
function fn(n){
    if(n==1){
        return 1
    }else{
        return n+fn(n-1)
    }
}
console.log(fn(100))
```

求1-100之间的偶数和

```
//n表示次数 50
function fn1(n){
    if(n==1){
        return 2
    }else{
        return fn1(n-1)+2*n
    }
}
fn1(50)
```

1 1 2 3 5 8 13 21 第20位是什么

```
function fn(n){
    if(n==1 || n==2){
        return 1
    }else{
        return fn(n-1)+fn(n-2)
    }
}
console.log(fn(20))
```

1 3 6 10 15 第10位是什么

```
function fn(n){
    if(n==1){
        return 1
    }else{
        return fn(n-1)+n
    }
}
console.log(fn(10))
```

1 1 2 3 6 11 20 第20位是什么

```
function fn(n){
    if(n==1 || n==2){
        return 1
    }else if(n==3){
        return 2
    }else if(n==4){
        return 3
    }else{
        return fn(n-1)+fn(n-2)+fn(n-3)
    }
}
console.log(fn(8))
```

1 5 11 19 29 41 问第15位是什么

```
function fn(n){
    if(n==1){
        return 1
    }else{
        return fn(n-1)+2*n
    }
}
console.log(fn(15))
```