程序与运行结果-5

1、编写一个脚本，0 到 10 完成 10 随机整数的排序
2、js 输出当前访问页面的设备的类型
3、以您认为合理的方式，给下面 HTML 元素绑定 click 事件,使绑定事件之后添加的 li 依然可以触发事修的处理函数
4、去除数组中重复元素
5、填充/TODO：处代码，以达成克隆目标对象的目的
6、编写一段脚本，实现 Function.bind 函数相同的功能
7、获取 URL 地址参数
8、冒泡排序
9、通过继承的方式达到输出结果
10、面向对象中继承实现
11、在金融应用产品中，数值常常使用千分位分隔，请使用 js 实现一个具有此功能的简单常数函数
运行结果

编程

1、编写一个脚本，0 到 10 完成 10 随机整数的排序

```
function sortNumber(a, b) {
      return a - b;//升序
      //return b-a;//降序
}
var iArray = [];
function getRandom(iStart, iEnd) {
      var iChoice = iStart - iEnd;
      return Math.abs(Math.ceil(Math.random() * iChoice)) + iStart;  //ceil() 方法可对一个数进行上舍入。
}
for (var i = 0; i < 10; i++) {
      iArray.push(getRandom(1, 10))
}
iArray.sort(sortNumber);
console.log(iArray)//随机生成  [1, 1, 4, 5, 6, 6, 6, 6, 7, 9]
```

2、js 输出当前访问页面的设备的类型

```
function checkEquipment() {
           var output = {};
           if (navigator.userAgent.match(/(iPhone|iPod|iPad);?/i)) {
               output['ios'] = true;
           } else if (
               navigator.userAgent.match(/android/i)) {
               output['android'] = true;
           } else if (navigator.userAgent.match(/windows/i)) {
               output['windows'] = true;
           } return output;
       };
       console.log(checkEquipment())//对应输出其中一个：{android: true} {ios: true} {windows: true}
```

通过 navigator 属性检查浏览器是什么浏览器 navigator.userAgent;

```
//网页中判断登录状态 toUpperCase()
var userAgentInfo = navigator.userAgent.toLowerCase();//
if (userAgentInfo.match(/MicroMessenger/i) != 'micromessenger') {
    // 网页
} else {
    //微信
}
```

3、以您认为合理的方式，给下面 HTML 元素绑定 click 事件,使绑定事件之后添加的 li 依然可以触发事修的处理函数

1
2
3
4
5

错误方式：55555 改成 let

```
var lis = document.getElementsByTagName('li')
var len = lis.length;
for (var i = 0; i < len; i++) {
    $(lis[i]).click(function () {
        alert(i);
        //555 })
    })
}
```

正确方式一(闭包)01234

```
var lis = document.getElementsByTagName('li');
for (var i = 0; i < lis.length; i++) {
    (function (index) {
        lis [i].onclick = function () {
           alert(index);
        }
    })(i);
};
```

正确方式二（利用 jq 中的 each） 01234

```
var lis = $("ul>li");
$.each(lis, function (index, element) {
    $(element).click(function () {
        alert(index);//索引
    })
})
```

4、去除数组中重复元素

var arr = [1,2,3,'3',4,2];console.log(arr)

一行代码实现数组去重（ES6）

```
let array = Array.from(new Set([1, 1, 1, 2, '3', 3, 2, 4]));
console.log(array);//[1, 2, "3", 3, 4]
```

new Set(array)返回类数组结象 {1, 2, 3, 4, 5}有 size

Array.from 可以把类似数组的对象转换为数组[1, 2, 3, 4, 5]有 length

forEach 去重

```
let arr1 = [3, 0, 2, 3, 2, 1];
let arr2 = [];
arr1.forEach(function (self, index, arr) {
      arr.indexOf(self) === index ? arr2.push(self) : null;
});
console.log(arr2);// [3, 0, 2, 1]
```

方法一：

```
Array.prototype.unique = function () {
    var res = [];
    for (var i = 0, newArr = []; i < this.length; i++) {
          if (!newArr[this[i]]) {
             res.push(arr[i]);
                 newArr[this[i]] = 1;
          }
    }
    return res;
}
var arr = [1, 2, 3, '3', 4, 2];
console.log(arr.unique());//[1, 2, 3, 4]
```

方法二：

```
let arr = [1,2,3,'3',4,2]
let set = new Set(arr);
let newArr = Array.from(set);
console.log(newArr); // [1, 2, 3, "3", 4]
```

方法三：

```
Array.prototype.unique=function(){
    var newArr=[];
    for(var i=0;i<this.length;i++){
        if(newArr.indexOf(this[i])==-1){
            newArr.push(this[i]);
        }
    }
    return newArr;
}
var arr= [1,2,3,'3','3',2,3,4,2];
console.log(arr.unique());//[1, 2, 3, "3", 4]
```

5、填充/TODO：处代码，以达成克隆目标对象的目的

```
var sourceObject ={
    id:0,
    name:'Alan',
    scores:[1,2,3],
    favorite:{
        tag1:'chinese',
        tag2:'math',
    }
}
```

var destObject=clone(sourceObject);

```
function clone(obj) {
    var type = typeof (obj);
    if (type === "string" || type === "boolean" || type === "number") {
           var newone = obj;
     } else if (type === "object") {
        if (obj === null) {
           var newone = null;
         } else if (Object.prototype.toString.call(obj).slice(8, -1) === "Array") {
       var newone = [];
      for (var i = 0; i < obj.length; i++) {
            newone.push(clone(obj[i]));
      }
    } else if (Object.prototype.toString.call(obj).slice(8, -1) === "Object") {
        var newone = {};
        for (var i in obj) {
            newone[i] = clone(obj[i]);
        }
      }
     }
    return newone;
}
```

this.constructor 是什么？

constructor：

1、constructor 始终指向创建当前对象的构造（初始化）函数。

2、每个函数都有一个默认的属性 prototype，而这个 prototype 的 constructor 默认指向这个函数

```
obj 改成数组 var obj = [1, 2]->this.constructor 值就是
f Array() { [native code] }    this.constructor===Array 成立

 obj 改成对象 var obj = { a: 1 }->this.constructor 值就是
f Object() { [native code] } this.constructor===Object 成立



Object.prototype.clone = function () {
    console.log(this.constructor)//f Object() { [native code] }
}
var obj = { a: 1 }
var obj2 = obj.clone()
```

使用 JavaScript 深度克隆一个对象。(百度)

Javascript 中的对像赋值与 Java 中是一样的，都为引用传递。就是说，在把一个对像赋值给一个变量时，那么这个变量所指向的仍就是原来对像的地址。那怎么来做呢？答案是“克隆”。

克隆有两种方法：一种是“浅克隆”，一种是“深克隆”（深度克隆）。

浅克隆：基本类型为值传递，对象指向相一对象指针，而不复制对象本身

深克隆（深度克隆）：所有元素均完全复制，对象完全独立（原对象的修改不影响新对象）。

深度拷贝：数组与对象通用

```
Object.prototype.clone = function () {
  var o = (this.constructor === Array ? [] : {});//{}
  for (var key in this) {  //this ->{name: "johnny"}
    o[key] = typeof this[key]==="object"?this[key].clone():this[key];
  }
  return o;
}
//1 将对象赋值给变量指向同一个数组
var a = []; var b = a; b[0] = 1; console.log(a[0]);  //1
```

其他原型复制方法通用

```
Object.prototype.clone = function () {
    var o = (this.constructor === Array ? [] : {});//{}
    for (var key in this) {  //this ->{name: "johnny"}
      o[key]=typeof this[key]==="object"?this[key].clone() : this[key];
    }
    return o;
}
var destObject = sourceObject.clone();//使用
```

6、编写一段脚本，实现 Function.bind 函数相同的功能

```
Function.prototype.bind = function (context) {
    var self = this; // 保存原函数
    return function () { // 返回一个新函数
     return self.apply(context, arguments); // 执行新函数时，将传入的上下文 context 作为新函数的 this
    }
}
```

7、获取 URL 地址参数

```
//方法 1
url =location.href;// var url = 'http://baidu.com?a=1&b=55';
var theRequest = {};
if (url.indexOf("?") != -1) {
   var str = url.substr(url.indexOf("?")+1);//a=1&b=55
   strs = str.split("&");
   for(var i = 0; i < strs.length; i ++) {
      theRequest[strs[i].split("=")[0]] = strs[i].split("=")[1];
   }
   console.log(theRequest)     //{a: "1", b: "55"}
}
```

```
//方法 2
function GetRequest() {
   var url = location.search; // search  取"?"符后的字串 ?a=2&b=33
   var theRequest = {};
   if (url.indexOf("?") != -1) {
      var str = url.substr(1);//a=2&b=33
      strs = str.split("&");//["a=2", "b=33"]
      for(var i = 0; i < strs.length; i ++) {
         theRequest[strs[i].split("=")[0]] = strs[i].split("=")[1];
      }
   }
   return theRequest;
}
Request = GetRequest();
console.log(Request)//{a: "2", b: "33"}
//方法 3 实用
// url?a=1&b=2
function getQueryString(name) {
           var reg = new RegExp('(^|&)' + name + '=([^&]*)(&|$)', 'i');
           var r = location.search.substr(1).match(reg);
   if (r != null) {
       return unescape(r[2]);
   }
   return null;
}// console.log(getQueryString('a'))
```

8、冒泡排序：

```
function sort(arr){
   for(var i=0;i<arr.length;i++){
      for(var j=0;j<arr.length-i-1;j++){
         if(arr[j]>arr[j+1]){
            var temp = arr[j];
               arr[j]=arr[j+1];
               arr[j+1]=temp;
         }

      }
   }
}
var arr=[1,2,5,32,54,33];
sort(arr);
console.log(arr); //[1, 2, 5, 32, 33, 54]
```

9、通过继承的方式达到输出结果 123

```
var AClass = function(){
     this.id=123;
}
var BClass=function(id){
     AClass.call(this,id)
}
var bInstance = new BClass;
```

```
BClass.prototype.sayId=function(){
     console.log(this.id)
}
```

```
bInstance.sayId();
```

10、面向对象中继承实现

javascript 面向对象中的继承实现一般都使用到了构造函数和 Prototype 原型链，简单的代码如下：

```
function Animal(name) {
     this.name = name;
}
Animal.prototype.getName = function () {
     alert(this.name)
}
function Dog() { };
Dog.prototype = new Animal("Buddy");
Dog.prototype.constructor = Dog;
var dog = new Dog();
dog.getName();//Buddy
```

11、在金融应用产品中，数值常常使用千分位分隔，请使用 js 实现一个具有此功能的简单常数函数

```
function toNum(value) {
     if (!value) return ' '
        var intPart = Number(value).toFixed(0) // 获取整数部分
        var intPartFormat = intPart.toString().replace(/(\d)(?=(?:\d{3})+$)/g, '$1,')
// 将整数部分逢三一断
       return intPartFormat;
     }
var n = 12001
console.log( toNum(n))//12,001   12001.6->12,002
```

运行结果

https://www.bbsmax.com/A/kjdwqm1wJN/

0、写一段脚本，实现观察者模式

0、写一段脚本，以你认为最优的方式向<body>中添加 10 个 Hello World 然后将所有的 p 标签删除，

1）var 和 function 同名并不冲突，在部分浏览器中，var 会遮蔽同名 function，导致 function 失效”，只要 test 被赋值就会 not a function ，只 var test;显示 2

```
var test = 1;
function test(
console.log(index);
    index = 3;
}
test(2)//结果：test is not a function
```

2、请问下述代码最终结果是什么？ 打印结果：55555

```
for(var i=0;i<5;i++){
    setTimeout(function(){
        console.log(i)//55555
    },1000)
    //console.log('i',i)//01234  这里能正常打印
}
```

原因分析：setTimeout()是一个异步处理函数，它会等待所有的主线程任务处理完，才开始执行自己的内部的任务，每隔 1s 往任务队列中添加一个任务【闭包函数，setTimeout()中的函数，现在还没执行】，当主线程执行完时，这时 i=5，

才开始执行任务队列中的任务【闭包函数，setTimeout()中的函数开始执行，执行 5 次】。

for 循环括号内的就是主线程，执行完时 i 是 5，所以会打印出 5 次 5；

如果想打印出 0,1,2，3，4

解决方案：

1. 把 var 改为 let，let 是块极作用域，每次 for 循环都会把对应的 i 绑定到添加的任务【闭包函数，setTimeout()中的函数】中,所以当主线程执行完时，也不会影响到每个任务中 i。所以可以打印出 0　1　2 3 4
2. 把定时器放在一个自执行函数中用 i 当做参数

```
for(var i=0;i<5;i++){
    (function(i){
     setTimeout(function(){
            console.log(i);//01234
       },1000)
    })(i)
}
```

————————————————————————————————

2、结果ƒ a(){}

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

```
    function a() { }
})(100)
```

解释： a 可以声明提前，赋值不提前，函数可以随时调取，这里 a 不是参数是变量了

函数在预解析的时候会被提到最前面，这个函数名和变量名重名，会显示函数

把 function a(){}改成 function b(){}就能显示 100 了

100 没起作用是**局部变量优先规则**

```
(function (a) {
    var a;
    console.log(a)
    a = 10;
    function a() { }
})(100)
```

局部变量优先原则，原理同下：

```
var a = 5;
function fn() {
    var a = 10;
    console.log(a)    // 10，局部变量优先，在局部找到 a 后，不会再向外查找
}
```

结果：undefined

```
var a = 10;
 (function () {
     console.log(a);//undefined
     var a = 100;
})();
```

解释：

声明会提升到 function 最开头，但赋值发生在最后上面的代码等价于：打印 a 的时候，a 并没有在 function 内赋值，所以是 undefined

```
va
 (function () {
      var a;
      console.log(a);
      a = 100;
})();
```

2、写出以下执行结果，谈谈上下文和作用域的区别？

结果：

~undefined

this:1

return1:2

return2:3

setTimeout:3

```
var a = 1;
function test() {
     console.log('~' + a); //声明提前
     var a = 2;
     console.log('this:' + this.a)  //全局
     setTimeout(function () {
         console.log('setTimeout:' + 3)
     }, 0)
     return function () {
          console.log('return1:' + a)
           a = 3
           console.log('return2:' + a)
     }
 }
 test()();
```

3、写下结果：说说 Event Loop Task Queue? 23541

Promise 定义之后便会立即执行,其次主任务，其后的.then() 然后 setTimeout

Promise 比 setTimeout()先执行

```
执行顺序 Promise-> 全局 ->then->setTimeout
```

```
setTimeout(function () {
    console.log(1)
}, 0)
new Promise(function executor(resolve) {
   console.log(2)
   for (var i = 0; i < 10000; i++) {
        i == 9999 && resolve();
   }
   console.log(3)
}).then(function () {
   console.log(4)
})
console.log(5)
```

结果：x 值是 3,到 case1 后 x 是 2 在执行 case2

```
var x=0;
switch(++x){
     case 0:
     ++x;
     case 1:
     ++x;
     case 2:
     ++x;
}
```

4、结果： test is not a function

```
var test=1
function test(index){
  console.log(index)
  index=3
}
test(2) //test is not a function
```

解释：var 和 function 同名并不冲突，但在部分浏览器中，var 会遮蔽同时的 function 导致 function 失效，只是 test 被赋值就会 not a function，如 var test 只声明就会显示 2

结果：1201

此题的关键在于明白 var result = test() 与 var result2 = test()此处分别产生了两个闭包，它们之间是独立，互不干扰的

闭包的产生：**在 A 函数中定义了 B 函数，B 函数中引用了 A 函数中的变量，此时会产生闭包**

```
function addCount() {
     var n = 0
     function add() {
          n++
         console.log(n)
     }
      return { n: n, add: add }
}
var newObj = addCount();
var newObj2 = addCount();
newObj.add()
newObj.add()
console.log(newOj.n)
newObj2.add()
```

5、运行 test()和 new test()的结果是什么

```
var a = 5;
a = 0;
alert(a);
alert(this.a);
var a;
alert(a);
function test() {}
```

参考答案：

test(): 0 5 0

new test(): 0 undefined 0

6、参考答案：

```
alert(typeof(null)) -> object
alert(typeof(undefined)) -> undefined
alert(typeof(NaN)) -> number
alert(NaN==undefined) -> false
alert(NaN==NaN) -> false
var str="123abc";alert(typeof(str++)) -> number
alert(str) -> NaN
```

7、结果：10

```
if('a' in window) {
    var a = 10;
}
console.log(a);    // 10
```

解释：**变量提升、window 的变量**

首先，if(){}的花括号并不像 function(){}的花括号一样，具有自己的块级作用域，if 的花括号还是全局的环境。根据 JavaScript 的变量提升机制，var a 会被 js 引擎解释到第一行，如下：

```
var a;
if ('a' in window) {
  a = 10;
}}
```

接着有个知识点，**全局变量是 window 对象的属性**，所以'a' in window 会返回 true，答案就很直白了。

这道题我在做的时候踩了个坑，我在代码编辑器里写了如下代码：

```
window.onload = function(){
if('a' in window){
        var a = 10;
}
console.log(a)    // undefined
```

这时候，a 这个变量是定义在匿名函数 function(){}里的，属于该函数的局部变量，所以 a 不再是 window 的对象。大家一定要注意细节。

8、结果：hello 666666 888888 world!

```
async function sayHello() {
  console.log('Hello')
  await sleep(1000)
  console.log('world!')
}
function sleep(ms) {
    return new Promise(resolve =>  {
    console.log("666666");
    setTimeout(resolve, ms);
    console.log("888888")})
}
sayHello()    // hello 666666 888888 world!
```

解释：

async =true ； 就表示异步的

async 表示这是一个 async 函数，await 只能用在这个函数里面。

await 表示在这里等待 promise 返回结果了，再继续执行。

首先打出 hello，到了 await，会等待 promise 的返回，所以“world”不会立刻打出，接着进入 sleep 函数，打出 666，接着开了一个 1 秒的定时器，虽然 js 是单线程的，但 setTimeout 是异步的，在浏览器中，异步操作都是被加入到一个称为“events loop”队列的地方，浏览器只会在所有同步代码执行完成之后采取循环读取的方式执行这里面的代码，所以 resolve 被加入任务队列，先打印了 888，一秒后执行了 resolve，表示 promise 成功返回，打出了 world。

9、上方的函数作用域中 a 被重新赋值，未被重新声明，且位于 console 之下，所以输出全局作用域中的 a。

```
var a = 1;
function test3() {
  console.log(a); // 1
  a = 2;
}
test3();
```

10、上方函数作用域中使用了 ES6 的 let 重新声明了变量 b，而 let 不同于 var 其不存在变量提升的功能，所以输出报错“b is not defined”。

```
let b = 1;
function test4() {
console.log(b); // b is not defined
let b = 2;//不加这就是 1
}
test4();
```

11、上方的函数作用域中用 let 声明了 a 为 1，并在块级作用域中声明了 a 为 2，因为 console 并不在函数内的块级作用域中，所以输出 1。

```
function test5() {
  let a = 1;
  {
    let a = 2;
  }
  console.log(a); // 1
}
test5();
```

12、上方利用 typeof 比较数组和对象，因为 typeof 获取 NULL、数组、对象的类型都为 object，所以 console 为 true。

```
var arr = [],
arr2 = {};
console.log(typeof (arr) === typeof (arr2)); // true
```

13、上方利用 instanceof 判断一个变量是否属于某个对象的实例，因为在 JavaScript 中数组也是对象的一种，所以两个 console 都为 true。

```
var arr = [];
console.log(arr instanceof Object); // true
console.log(arr instanceof Array); // true
```

二、this 指向

1、This 作用对象方法被调用指向对象

```
var obj = {
    name: 'xiaoming',
    getName: function () {
     return this.name
    }
};
console.log(obj.getName()); //xiaoming
```

nameFn()作为函数调用，obj.getName()这是作用方法调用

```
var obj = {
  myName: 'xiaoming',
  getName: function () {
    console.log(this)
    return this.myName
  }
};
var nameFn = obj.getName;
console.log(nameFn()); // undefined
```

nameFn()作为函数调用，加 apply 后 this 指向 obj2 对象

```
var obj = {
  myName: 'xiaoming',
  getName: function () {
    return this.myName
  }
};
var obj2 = {
  myName: 'xiaohua'
};
var nameFn = obj.getName;
console.log(nameFn.apply(obj2)); // 'xiaohua'
```

函数参数

Arr.slice(start,end)下标开始，结束（不包含该元素）

```
var arr = new Array(6)
arr[0] = "George"
arr[1] = "John"
arr[2] = "Thomas"
arr[3] = "James"
arr[4] = "Adrew"
arr[5] = "Martin"
document.write(arr.slice(2, 4)) //Thomas,James
```

三、 闭包问题

```
var elem = document.getElementsByTagName('li'); // 如果页面上有 5 个 li
```

```
for (var i = 0; i < elem.length; i++) {
    elem[i].onclick = function () {
      alert(i); // 总是 5
    };
}
// 上方是一个很常见闭包问题， 点击任何 div 弹出的值总是 5， 因为当你触发点击事件的时候 i 的值早已是 5，
可以用下面方式解决：在绑定点击事件外部封装一个立即执行函数， 并将 i 传入该函数即可。
var elem = document.getElementsByTagName('li'); // 如果页面上有 5 个 li
for (var i = 0; i < elem.length; i++) {
  (function (w) {
    elem[w].onclick = function () {
      alert(w); // 依次为 0,1,2,3,4
    }
  })(i);
}
```

Object.assign()方法进行对象的深拷贝可不改变源对象，返回目标对象

`Object.assign` 方法用于对象的合并，将源对象（source）的所有可枚举属性，复制到目标对象（target）。

`Object.assign` 方法的第一个参数是目标对象，后面的参数都是源对象。

```
var obj2 = {
  name: 'xiaoming',
  age: 23
};
var newObj2 = Object.assign({}, obj2, {
  color: 'blue'
});
newObj2.name = 'xiaohua';
console.log(obj2.name); // 'xiaoming'
console.log(newObj2.name); // 'xiaohua'
console.log(newObj2.color); // 'blue'
console.log(newObj2) // {name: "xiaohua", age: 23, color: "blue"}
console.log(obj2) //{name: "xiaoming", age: 23}
```

我们也可以使用 Object.create()方法进行对象的拷贝，Object.create()方法可以创建一个具有指定原型对象和属性的新对象。

```
var obj3 = {
name: 'xiaoming',
age: 23
};
var newObj3 = Object.create(obj3);
newObj3.name = 'xiaohua';
console.log(obj3.name); // 'xiaoming'
console.log(newObj3.name); // 'xiaohua'
```

运行结果：2

```
alert(1 && 2)
```

因为 js 的逻辑运算符并不做转换，它是把为真的那个值本身给你，而不是转成 boolean，这个跟绝大多数语言都不一样（当然，其他语言本身&&也不允许 boolean 以外的值参与运算）

1 && 2 => 2

false || 'abc' => 'abc'

---

this 的指向——运行结果：11

```
window.val = 1;
var json = {
    val: 10,
    dbl: function () {
        val * 2;  //局部优先 val 是 1,this.val 是 10
        console.log(this.val)
    }
}
json.dbl();
console.log(json.val + val)//11
```

面向对象-new 时的初始值——最重要是理解 js 中变量在原型链中查找的顺序，

new C1().name：由于这里没有参数，默认被赋值成了 undefined，所以到了 if 这里就进不去了，因此在 C1 本地属性中找不到 name 这个属性，只能傻逼兮兮的往外找了，又因为 C1.prototype.name = "Tom"的存在，在 prototype 中找到了 name 属性，所以最后打印出来的答案是"Tom"

new C2().name： 由于这次还是没有参数，同样默认被赋值成了 undefined，于是本地属性 name 被赋值成了 undefined。于是在查找的时候一下子就查到了 name 的值为 undefined，因此 C2.prototype.name = "Tom"并没有什么卵用，最终答案为 undefined

new C3().name：同样是没有参数，undefined 作为参数进来以后情况变成了这样：this.name = undefined || "John"，然后结果很明显了本地属性 name 被赋值成"John"。接着是从内往外查找，一下子就锁定了本地属性 name，此时的值为"John"。因此 C3.prototype.name = "John"同样没有什么用

```
function C1(name) {
    if (name) {
        this.name = name;
    }
}
function C2(name) {
    this.name = name;
    console.log(this.name)
}
function C3(name) {
    this.name = name || 'john'//undefined||'john'->john
}
C1.prototype.name = 'Tom';
C2.prototype.name = 'Tom';
C3.prototype.name = 'Tom';
// 结果：Tomundefinedjohn
console.log((new C1().name) + (new C2().name) + (new C3().name));
```

总结：

如果原型链上有相同的方法。那么会优先找本地方法，找到并执行，原型链上的方法就不执行了。

同样的，属性的查找也是这么个顺序。

```
function Foo() {
    this.say = function () {
        console.log('本地方法');
    }
}
Foo.prototype.say = function () {
    console.log('prototype 方法');
}
new Foo().say();//结果：本地方法
```

4：考点 自执行函数,形参与实参,默认参数是 5,所以 test 传值 2 根本就没接收,所以答案是 10

```
var test = function (i) {
    return function () {
        console.log(i * 2)//10
    }(2)
}
test(5)
```