js 正式课第一周

为什么学习变量提升？

- 是衔接免费课知识点深入学习
- 能够知道书写代码的规范（何时正确，何时错误，为啥要这样写）
- 能够更加了解代码在浏览器的运行机制

什么是变量提升

- 是一个阶段 在代码运行之前做的事情
- 范围：在当前作用域下
- 含义：在当前作用域下，代码执行之前，对带var和funcfion关键字的进行声明和定义
    - 对var是声明 var num
    - 对function是声明+定义 var num = 10；

- 对判断语句里的function只声明不定义

函数定义的步骤

1、首先开辟一个堆内存，假设引用地址fff000
2、函数体的内容以字符串的形式保存在这个内存空间里
3、将引用地址fff000赋值给函数名fn，fn就代表了整个函数

普通函数执行步骤

1、形成一个私有作用域
2、形参赋值
3、变量提升（**重名变量不会重复定义**）
4、函数体的内容转换成代码，从上到下运行

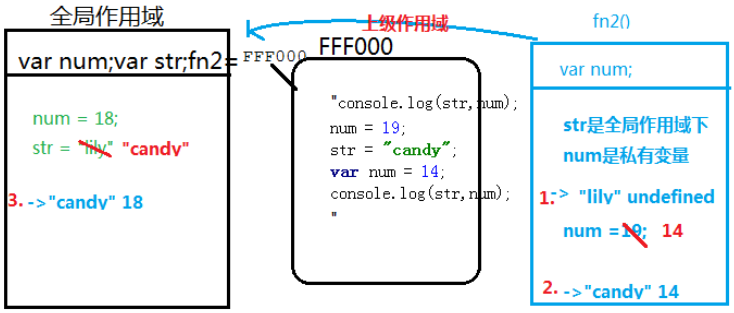
构造函数执行

1、开辟私有作用域
2、形参赋值
3、变量提升（**重名变量不会重复定义**）

4、浏览器默认开辟一个储存空间是object数据类型构造函数里面的this指向这个object地址

作用域

```
console.log(num,str);
var num = 18;
var str = "lily";
function fn2(){
 1. console.log(str,num);
    num = 19;
    str = "candy";
    var num = 14;
 2. console.log(str,num);
}
fn2();
3.console.log(str,num);
```



全局作用域：页面加载时形成 全局变量
私有作用域：函数运行时形成 私有变量：形参，带var关键字的变量
作用域查找变量顺序：对不带var的变量进行查找

- 若在私有作用域有，是私有变量，与外界无关
- 若私有作用域中没有，会往外面查，直到找到window，如果window下也没有就报错
  由当前作用域往上级作用域，然后再往上级作用域，直到到window，这样就形成了作用域链。
  什么是上级作用域？
- 函数在哪定义，他的上级就是谁
- **上级作用域只跟定义有关，跟函数执行无关**

a&&window.a

- 如果全局变量没有a，console.log(a)报错，console.log(window.a)是undefined

变量提升的细节知识点

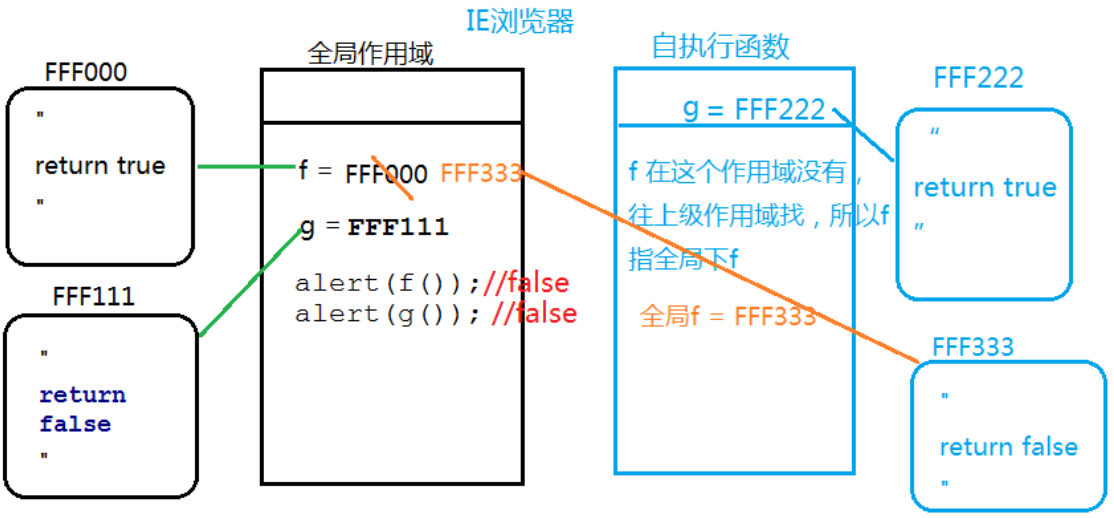
- 1.只对等号左边的变量提升，
- 2.函数作为值存在时不进行变量提升
- return后面的不会变量提升
- return下面的会进行变量提升
- 3.重名变量不会重复声明，但会重复定义
- 4.自执行函数本身不进行变量提升，内部会形成变量提升
- 5.不管条件成立不成立，对带var的关键字进行声明 对function关键字标准浏览器只声明不定义，ie既声明也定义

逻辑与（&&）逻辑或||

> 条件判断中&&和||都会对两边的内容转换成true或false
> &&两边同时为真才为真，有一边为假则为假
> ||只要有一边为真则为真，同时为假则为假
> 不是在条件判断中（赋值操作中），规律如下：
> &&左边为真返回右边，左边为假返回左边
> ||左边为真返回左边，左边为假返回右边

运算符优先级

算数运算符>比较运算符>逻辑运算符>赋值运算符




```
f = function () {return true};
g = function () {return false};
(function () {
if (g () && [] == ![]) {
f = function f() {
return false
};
function g() {
return true
}
}
})();
alert(f());
alert(g());
```

闭包

- 函数运行时形成一个不受外界干扰的私有作用域，这就是一个闭包，闭包是一种机制
- 封装性（保存）
- 保护的作用，防止里面的内容跟外界内容冲突和覆盖

闭包的几种形式

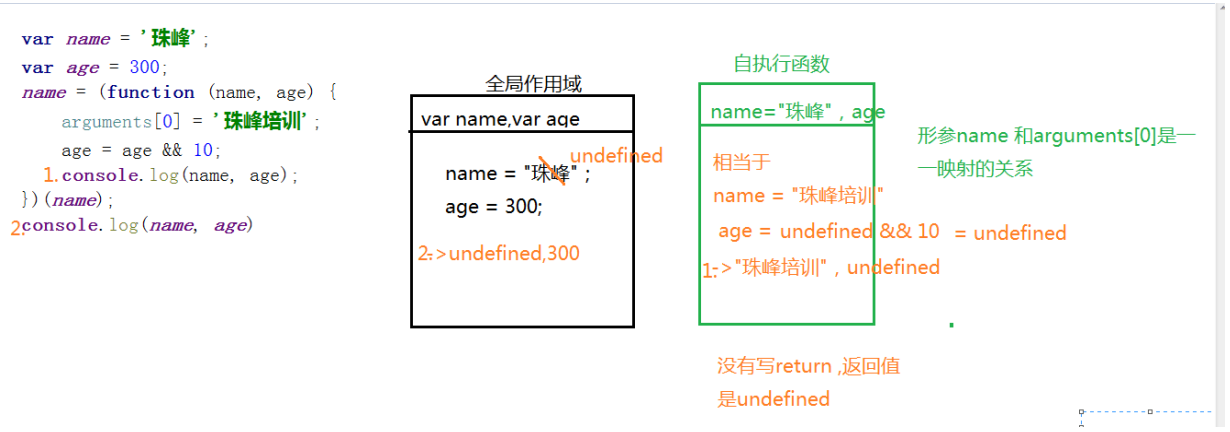
- 自执行函数

```
//utils是一个模块
    var utils = (function(){
        var name = "lily"; //私有的变量
        function login(){console.log("login")}
        function register(){console.log("register")} //私有的方法
        return {
            login:login
        }
    })();
    utils.login(); //调用里面方法  模块名.方法名()
```

- 函数的返回值是一个小函数
    - 利用预处理的思想：先提前处理公有的内容，等到return后面小函数运行时会用到提前处理的内容 例如bind方法的封装就利用了这种思想

- 函数内部返回对象（高级单例模式）
    - 解决了公有和私有的问题

- 例子：

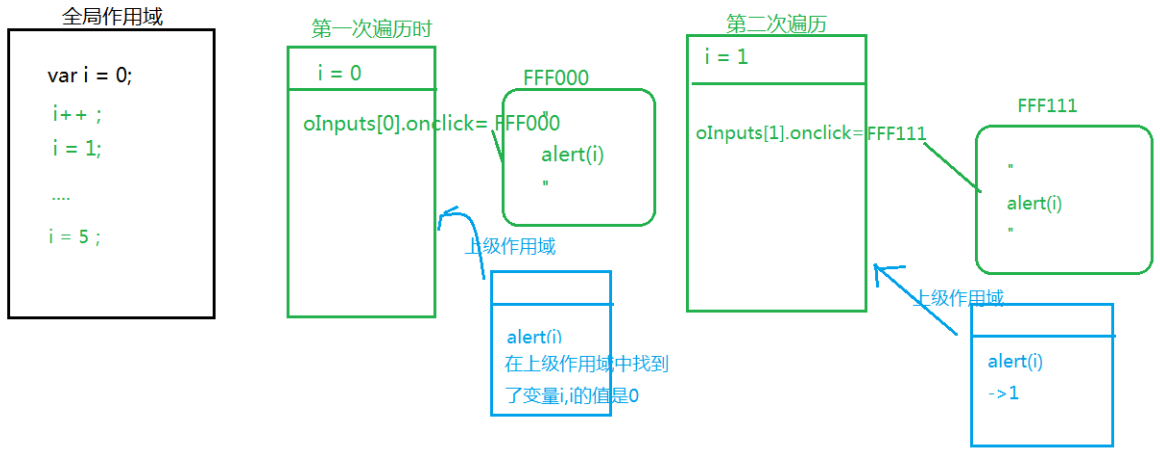



```
var name = '珠峰';
    var age = 300;
    name = (function (name,age) {
```

```
        arguments[0] = '珠峰培训';
        age = age && 10;
        console.log(name, age);
    })(name);
    console.log(name, age);
```

例子：

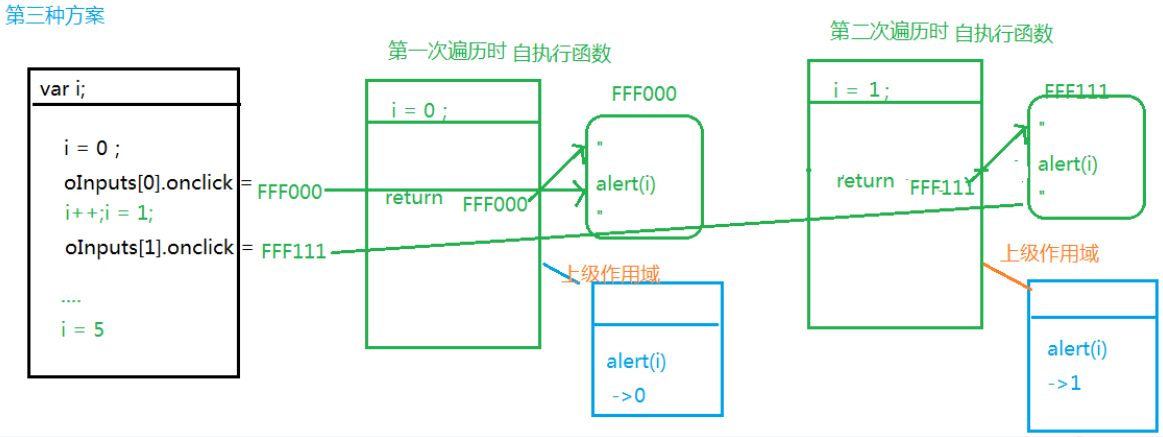
第一种闭包方式




```
//es5中{}不表示一个块级作用域，for循环中的{}仅仅表示循环体中的括号，i变量都表示
//全局下的i，因此时间绑定的方法执行时收获的i是全局下的i

for (var i = 0;i< oInput.length;i++){
        (function (i) {
            oInput[i].onclick = function () {
                alert(i);
            }
        })(i)
    }
```

第二种闭包



```
for (var i = 0;i <oInput.length;i++){
        oInput[i].onclick = function (i) {
           return function () {
               alert(i);
           }
        }(i);
    }
```

第三种闭包

```
for (var i = 0;i < oInput.length;i++){
        changeTab(i);
    }
    function changeTab(i) {
        alert(i);
    }
```

第四种闭包es6

```
//声明变量es5  var   es6   let   for循环中的{}表示块级作用域
for (let i = 0;i < oInput.length;i++){
    oInput[i].onclick = function () {
            alert(i);
        }
    }
```

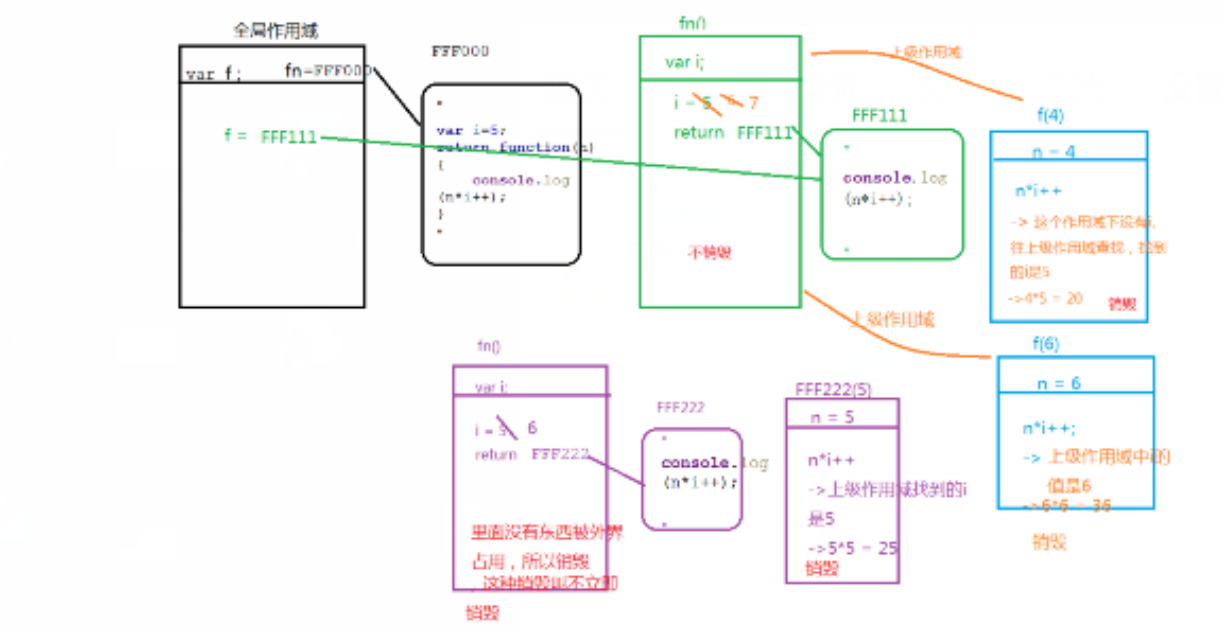
1、let没有变量提升 变量必须是声明后再用
2、在当前作用域 es6中不允许重名变量，有重名变量会报错
const和var区别 const是常量
1、let没有变量提升 变量必须是声明后再用
2、在当前作用域 es6中不允许重名变量，有重名变量会报错
3、const是常量不可以重新赋值
第五种

```
for (var i = 0;i < oInput.length;i++){
        oInput[i].cur = i;
        oInput[i].onclick = function () {
            alert(this.cur);
        }
    }
```

作用域销毁

全局作用域：页面关闭时销毁
私有作用域：立即销毁 、 不立即销毁 、不销毁
正常情况下，方法运行完后，产生作用域就会销毁，但也有些特殊情况：
1、当一个函数A返回另一个函数时，不会立即销毁，等返回这个小函数运行完后，这个函数A才销毁 —-这叫不立即销毁
2、当函数内部的内容被外界占用了，就不销毁

- 作用域不销毁的缺点和有点
  - 缺点：会占用内存
  - 优点：上次的结果会保留下来
    例题：




```
function fn() {
      var i = 5;
      return function (n) {
          console.log(n * i++);
      }
  }
  var f = fn();
  f(4);
  fn()(5);
  f(6);
```

堆内存销毁：
如果堆内存的引用地址被变量占用了，这时堆内存不会销毁，只有没有变量指向这个堆内存时，浏览器会在空闲时回收这块内存，这称为垃圾回收机制

this

this是函数执行的主体
绑定事件中，给谁绑定事件this就是谁
`myLis[i].onclick = clickFn`
clickFn是点击时的方法，循环绑定中给所有myLis绑定的是同一个方法，this代表点击时的myLis
1.看方法名前面有没有点，若没有则this是window,若有点，点前面是谁this就是谁
2.->自执行函数的this是window

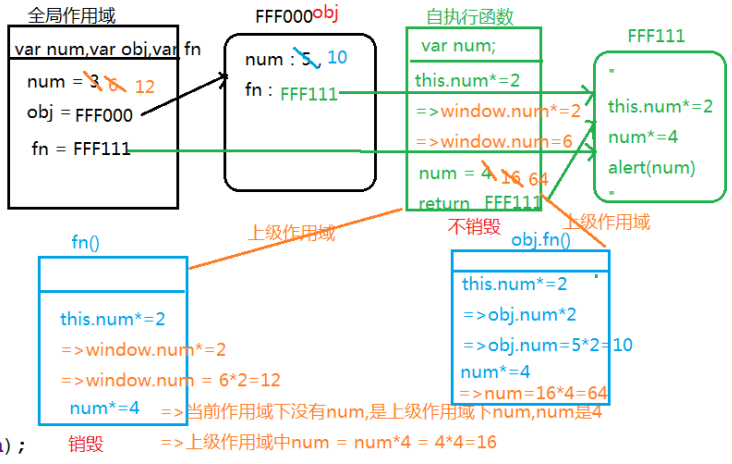
```
var num = 10;
    var obj = {
        num:20,
        fn:(function(){
            console.log(this);
        })()
    }
    ;(function(){
        console.log(this);
    })();
```

3.->事件绑定函数中是this是绑定的元素

```
var oDiv = document.getElementById("div1");
    oDiv.onclick = function(){
        console.log(this);
    }
```

例子：

```
var num = 3;
var obj = {
    num:5,
    fn : (function(){
        this.num*=2;
        var num = 4;
        return function(){
            this.num *=2;
            num *= 4;
            alert(num);
        }
    })()
}
var fn = obj.fn;
alert(num);//6
fn(); //16
obj.fn(); //64
console.log(window.num,obj.num);
```




```
var num = 3;
    var obj = {
```

```
        num:5,
        fn:(function () {
            this.num*= 2;
            var num = 4;
            return function () {
                this.num *= 2;
                num *= 4;
                alert(num);
            }
        })()
    };
    var fn = obj.fn;
    alert(num);
    fn();
    obj.fn();
    console.log(window.num, obj.num);
```

面向对象

单例模式

> 最开始JS语言并没有引用数据类型，只有基本数据。会产生全局变量污染(冲突)问题

```
var name = 'zhufeng'
var age = 18

// 如果没有对象数据类型，我们有需要另一个name或者age就会和之前的变量产生冲突
// 可以给name加前后缀，但是操作起来很麻烦，代码也不易维护，没有扩展性
var name = 'jay'
var age = 40
```

> 代码的可扩张性
> 在我们需要操作一个人的name和age的时候，我们并不知道之后会不会还有其他的人需要加入进来，如果我们直接使用name和age变量，之后会很难维护，所以我们可以用一个对象将这两个属性包含起来

```
function add1() {
    var total = 0
```

```
    // 最后一项是需要减的
    for (var i = 0; i < arguments.length; i++) {
        if (i == arguments - 2) {
            // 倒数第二项是需要和前面累加的相乘
            total = total * arguments[i]
        } else  {
            total += arguments[i]
        }
    }
    return total - arguments[arguments.length-1]
}
add1(1,2,3,4,5,7)
function add(data) {
    var total = 0
    var addNum = data.addNum
    for (var i = 0; i < addNum.length; i++) {
        total = total + addNum[i]
    }
    total = total * data.multiply
    total = total - data.subtract
    return total
}
add({
    addNum: [1,2,3,4,5],
    multiply: 5, // 累加后需要乘的数
    subtract: 7  // 最后需要减的数
})
```

> 例如累加方法，如果我们是一个个的传入数字，然后遍历arguments，前期实现的需求。但是如果后面想要对累加的数进行其他操作就很困哪，要么把传参方式改了但是需要重写方法，要么就在原函数里面无限的修改，最后别人也不知道你代码写的是啥。
> 所以在最开始我们传参的时候就可以使用 `单例模式` 也就是传一个对象

模块化开发

百度首页

> 我们分成以下几个模块 一般我们一个文件就是一个模块
> 主页面模块 入口文件(index)
> 搜索设置模块 searchModel
> 天气模块 wearthModel

```
var searchModel = {

}
// 这两个可能是在不同的文件
var wearthModel = {

}
```

> 对象是将描述同一事物的属性和方法合并，这也是单例模式的使用
> 将函数的参数以一个对象的形式传递，也是使用了单例模式的思想

高级单例模式

> 高级单例模式就是融合了其他设计模式(发布订阅模式，promise)去写的代码
> 如下 就是单例模式和闭包思想的融合，就是高级单例模式

```
function fn1() {

    return {}
}
var dataFn1 = fn1()
```

工厂模式

> 就是将实现某一功能的代码封装成一个函数，下次再需要实现这个功能的时候调用这个函数即可
> `低耦合高内聚` 减少页面不必要代码，提高代码利用率

```
// 累加一组数[1,2,3] 然后乘一个数5
// 在累加一组数[4,5,6] 再除一个数2 返回结果
function operator(data) {
    var total = 0
    var arrOne = data.arrOne
    var multiply = data.multiply
    var arrTwo = data.arrTwo
    var divide = data.divide
    for (var i = 0; i < arrOne.length; i++) {
```

```
            total = total +  arrOne[i]
        }
        total = total * multiply
        for (var i = 0; i < arrTwo.length; i++) {
            total = total +  arrTwo[i]
        }
        total = total / divide
        return total
    }
    operator({
        arrOne: [1,2,3],
        multiply: 5, // 乘
        arrTwo: [4,5,6],
        divide: 2 // 除
    })
```

> 累加一组数[1,2,3] 然后乘一个数5 在累加一组数[4,5,6] 再除一个数2 返回结果
> 上面就是需要实现的功能，我们把这个功能封装成函数operator 这就是工厂模式
> operator执行的时候传过去一个对象就是单例模式的使用

面向对象

> 面向过程 C
> 面向对象 java php c++ javaScript

对象 类 实例

> 万物皆对象，我们研究学习的所有东西都是对象
> 类 对象的具体划分 比如说 马类 人类 植物类
> 实例 某一类中的具体实物 人中的 你 我 他

JS中的内置类

数据类型

- Number
- String
- Boolean

- Null
- Undefined
- Object
    - Array
    - RegExp
    - Date

- Function
  元素集合类
- NodeList
- HTMLCollection

> Null 和 Undefined浏览器禁止我们操作
> document > HTMLDocument > Document(getElementById) > Node > EventTarget >
> Object
> li > HTMLLIElement > HTMLElement > Element > Node > EventTarget > Object

基于面向对象(构造函数方式)创建数据

> 使用面向对象的方式创建基本数据类型
> 赋值给一个变量后，可以像字面量方式创建的数据一样使用
> 但是面向对象方式创建的数据都是object数据类型

```
var a = 100; // 字面量方式创建一个Number类型数据赋值给a变量
var b = new Number(100) // 通过面向对象的方式创建一个数据赋值给b
console.log(typeof a, typeof b)
// number object
b + 100 // 200
```

> 使用面向对象的方式创建引用数据类型

```
var arr = [1, 2, 3]
var arrNew = new Array(1, 2, 3)
console.log(arr, arrNew)
// [1, 2, 3] [1, 2, 3]
var arrOne = [9]
var arrOneNew = new Array(9)
console.log(arrOne, arrOneNew)
```

```
// [9] [empty × 9]
```

使用面向对象的方式创建数组
当参数多于1的时候，和字面量方式没有区别
但是只有一个参数的时候面向对象方式创建的数组结果就是 长度为参数，然后每个值都为空的数组

字面量方式创建对象，一般都是创建空对象

```
var obj = {name: 'zhufeng'} // 字面量方式创建对象
var objNew = new Object() // 基本用来创建空对象
```

普通函数执行 VS 构造函数执行

```
function Fn() {
    this.name = 'zhufeng'
}
Fn()
var f = new Fn()
console.log(f, '>>>>>')
```

当函数执行前面有new 那么这就不再是函数执行了，而是构造函数的执行,
返回值就是当前类的实例
如果我们创建的是类(构造函数)，都会将首字母大写，以此和普通函数进行区分
普通函数执行：
开辟私有作用域 =》 形参赋值 =》 变量提升 =》 代码执行
构造函数执行
开辟私有作用域 =》 形参赋值 =》 变量提升
浏览器默认开辟一个存储空间是object数据类型，构造函数里面的this指向这个object地址
this也就是当前的实例
如果没有return 浏览器会默认把她创建的object返回出去,也就是this(当前类的实例)被返回出去
如果有return但是返回的是基本数据类型，浏览器会忽视这个return继续返回this(当前类的实例)
如果return的是一个对象 那么浏览器就会把这个对象返回，忽视this(当前类的实例)

原型

> 每一个函数(类)，都有一个prototype属性，存储的是给当前类使用的公有的属性和方法。prototype是一个对象，浏览器默认给他开辟堆内存
> 每一个prototype都有一个默认属性constructor指向类本身
> 每一个实例(对象)，都有一个**proto**属性，指向所属类的原型

prototype和proto的关系是什么？

- prototype是显式原型，它是指向函数的原型对象。（函数创建之后就会产生prototype属性）
    - 显式原型的作用：用来实现基于原型的继承与属性的共享
- proto是隐式原型，它所指向的是创建这个对象的函数（constructor）的prototype，可以通过 object.setPrototypeOf();来获得一个对象的proto属性；
    - 隐式原型的作用：构成原型链，同样用于实现基于原型的继承。举个例子，当我们访问obj这个对象中的x属性时，如果在obj中找不到，那么就会沿着proto依次查找。