JS面象对象

在学JS基础时，我们说过JS中的对象，分两阶段：

* 1，对象的初识
* 2，面向对象

复习：

**什么是对象？**

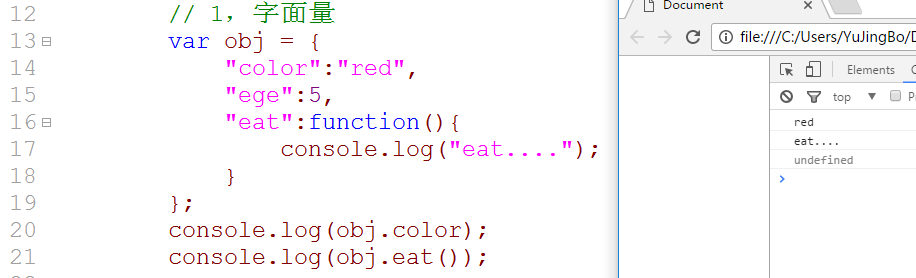
万物皆对象。

**对象有两个特征：**

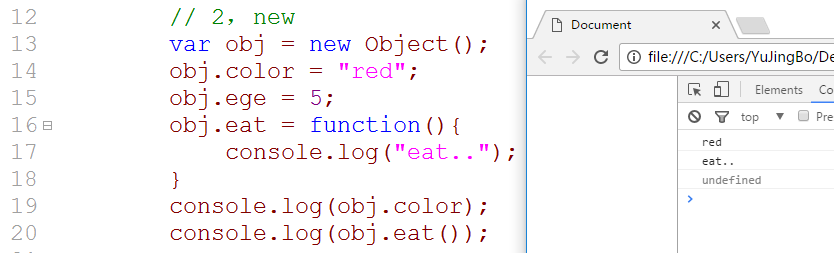
1，静态的特征

2，动态的特征

**创建对象的两种方式：**

**1，字面量**

**2，new**



**如何访问对象的属性和方法：**

对象名.属性名 对象名.方法名() 点语法 conlose.log() window.alert() document.get...

**对象的分类 ：**

1，内置对象 是ECMAScript中已经定义好的对象 单体内置对象（不需要new）

2，宿主对象 BOM, DOM

3，自定义对象

# 一，一些基本概念

## 回顾JS的数据类型

**JS中的数据类型分两类：**

1，基本数据类型

number, string, boolea, undefined, null

2，引用数据类型

array, function, object

## 回顾JS中的对象，JS对象分类

**JS中的对象分三类：**

1，内置对象

2，宿主对象

3，自定义对象

**创建对象的两种方式：**

1，字面量的方式

2，new的方式

**如何访问一个对象的内容：**

1，访问属性：对象名.属性名

2，访问方法：对象名.方法名()

## JS对象加强

**对象是属性的无序集合，每个属性都存放一个原始值，对象或函数。**换句话说，

对象中的属性可以存在基本类型的数据，也可以存储引用数据类型的数据。

下面这三点，一个在把握好：

* 对象是属性的集合
* 属性的摆放是无序的
* 属性值可以是任何的数据类型（基本数据类型，引用数据类型）

数据也是对象： var arr = [1, 2, 3, 4]; var arr = [0:1, 1:2, 2:3, 3:4];

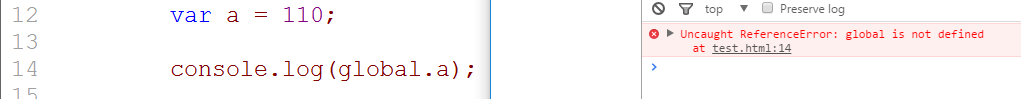
对象是这样定义的：var obj = {k1:v1, k2:v2, k3:v3};

数组是特殊对象，好比，正方形和矩形。

## 特殊的Global对象

这个global，在js中非常特殊， 在客户端js中，这个global对象是不存在。但是记住两点：

1，我们定义的全局变量，全局函数，它们就是global对象属性。但是在客户端global是不存在的。



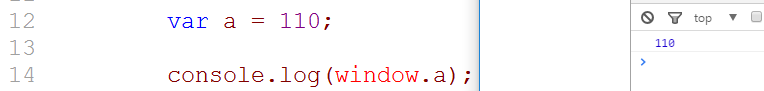
2，其它的本地对象也是它的属性。



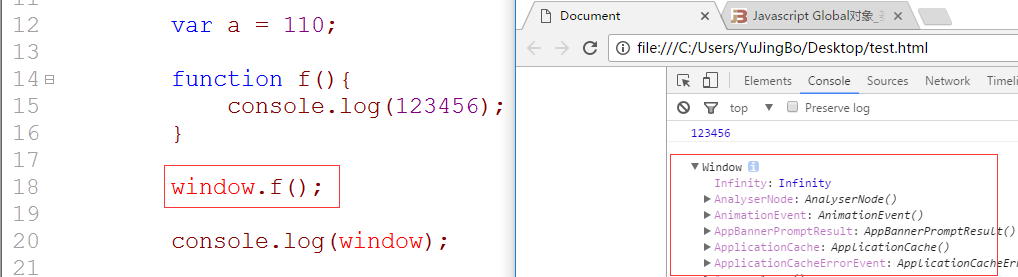
global虽然它不存在，但是我们可以把它看作js程序运行的总管。

## window对象

window对象，我们就可以看作是global对象在**浏览器器端**的个代言人。换句话说，在浏览器端写js程序时，就可以直接使用window来代替global. 如下图:



需要我们注意的是：在服务器端，这个golbal是真实存在的。我们也可以查看一个window对象：



有一点需要注意：在使用的过程中，window是可以省略的。 如：window.alert(). window.setInteral().......

## 什么是面向对象

**面向对象**一种编程思想。 JAVA， PHP是面向对象。

**面向过程**。如C语言就是面向过程的。

**将大象装进冰箱：**

面向过程：1， 打开门 2，将大象放到冰箱中 3，关上门

面向对象：1，冰箱对象.open() 2, 大象.putIn(冰箱) 3，冰箱对象.close()

这里面有两个对象：大象，冰箱

**面向对象一般有四大特征（三大特征）：**

* 封装
* 继承
* 多态
* 抽象

JS不是面向对象，面向对象一般都有class关键字，**JS是基于对象。它是模拟了面向对象的三大特征。在JS中是没有类的概念的。**

**JS是基于对象的，不是面向对象！**

## （7）三条定律，两条链

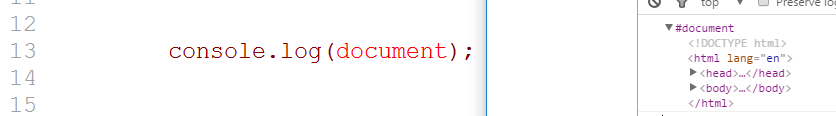
* **1，一切皆对象（数组，函数，对象，变量.....）**
* **2，对象由函数创建**
* **3，对象是属性的集合**
* **作用域链**
* **原型链**

# 二，一切皆对象

## 回顾我们之前接触到的对象

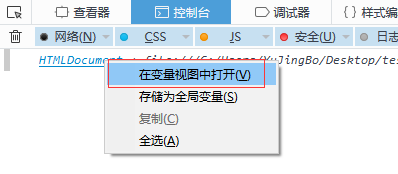
* window
* console
* dom元素
* document

document对象的内容：

谷歌浏览器对于对象的输入不太完整：

我们使用火狐浏览器：

可以借住下面的工具查看：

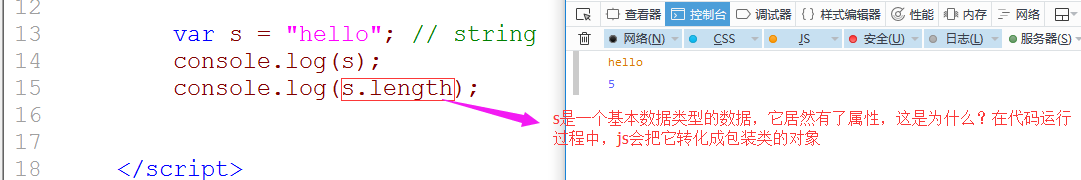


document对象中有一个属性：名为\_\_proto\_\_, 这个属性也是一个对象，我们再往下找一个\_\_proto\_\_就可以找到，我们前面使用过的getelmentById.....

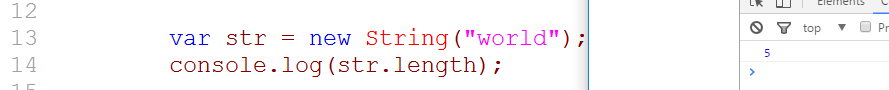


## 基本数据类型也可以当作对象

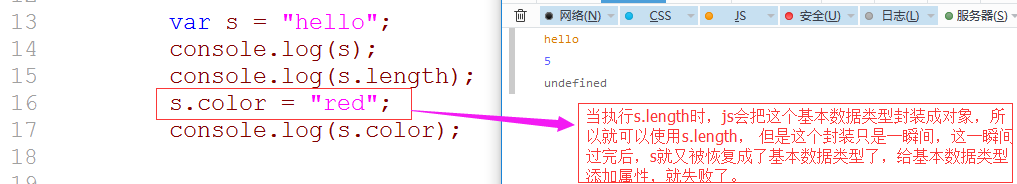
**1, 自动创建的包装类型对象**



上面的包装类型对象和真正的对象是不一样的，对于字符串，我们也可以使用new , 此时，str本身就是一个对象了。



**2，自动创建的包装类型对象只存在瞬间**



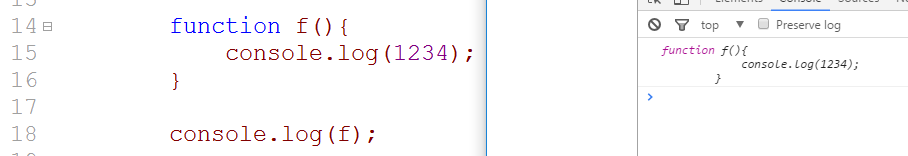
## 引用数据类型也是对象

**引用数据类型分下面三类：**

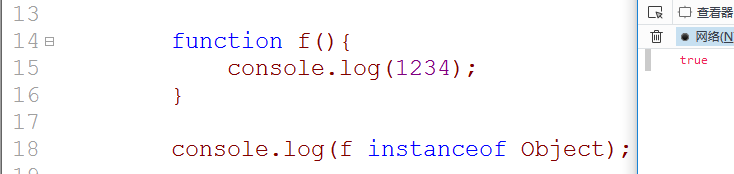
* 函数
* 数组
* 对象

### 函数是对象

我们可以把函数打印出来：

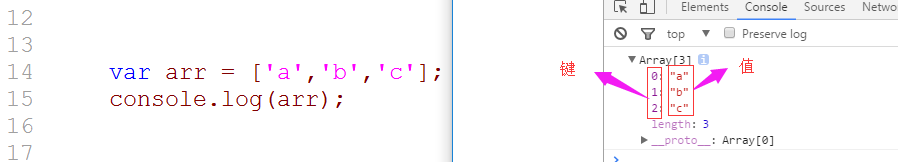


如何判断它是否为对象： instanceof的功能：查看数据是否为指定类型



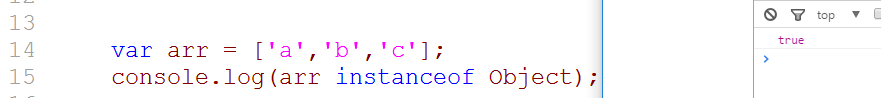
### 数组是否为对象

我们打印一个数组，结果如下： 里面有键盘对：

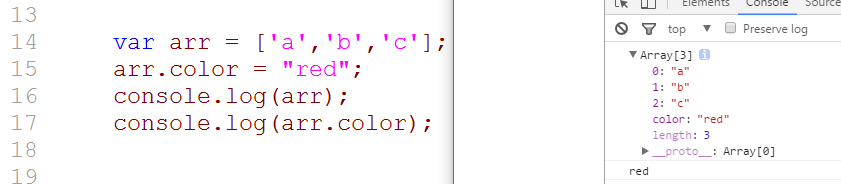


证明这个数组是否为对象：

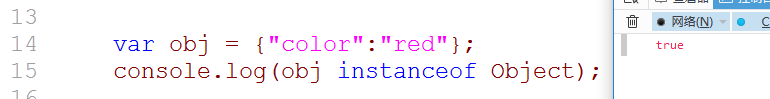
方式一：



方式二：



### 对象是对象



结论：一切都是对象

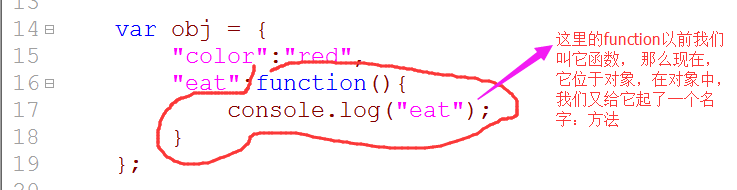
# 三，对象是属性的集合-操作属性

## 函数和方法的区别？

我们前面声明函数有两种方式：函数声明和函数表达式。

函数就是方法，方法就是函数。

当你将一个函数放到一个对象中时，那么这个函数就又有了一个别名：方法。换句话说，方法就是放在对象中的函数。



对象是属性的无序集合。属性分为属性名和属性值，属性名就是键， 属性值就是值。当属性值是一个函数时，此时这个函数，我们就叫它为方法。

## 属性与变量的区别？

对象是属性的无序集合。**在传统的编程语言，**对象是静态特征和动态特征的一个集合，对于静态特征，我们叫它属性，对于动态特征，我们叫它方法。**在JS中**，无论是静态的，还是动态，我们统称它为属性。我们现在也沿用传统语言中习惯，叫动态的特征为方法。

如果一个变量是某个对象的属性名，则称这个变量为对象的属性。

**复习：**

**1，什么是对象？**

对象是属性的无序集合。

* 对象是属性的集合
* 属性的摆放是没有顺序的
* 属性的值可以是任意地数据类型

**2，对象的分类？**

JS中的对象分类：

* 1，内置对象 在ECMASciprt中定义好的对象，我们可以直接使用
* 2，宿主对象 BOM, DOM
* 3，自定义对象 我们自已定义的对象

**3，JS是面向对象的吗？**

JS不是真正地面向对象，因为面向对象一般都有类的概念，但在JS中没有类的概念。一般说JS是基于对象。

**4，三条定律，两条链？**

1. 定律：

* 1，一切皆对象
* 2，对于由函数创造
* 3，对象是属性的集合

两条链：

* 1，作用域链
* 2，原型链

**5，一切皆对象？**

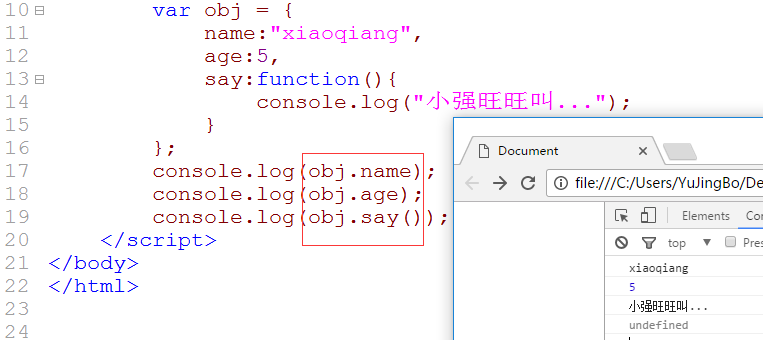
* 1，在JS中有内置对象和宿主对象
* 2，基本数据类型也可以作为对象（包装类型对象）
* 3，函数也是对象
* 4，数组也是对象
* 5，对象也是对象

**总结：一切皆对象**

## 属性的操作？

### 1，访问一个属性

**可以通过.语法去访问一个对象的属性：**

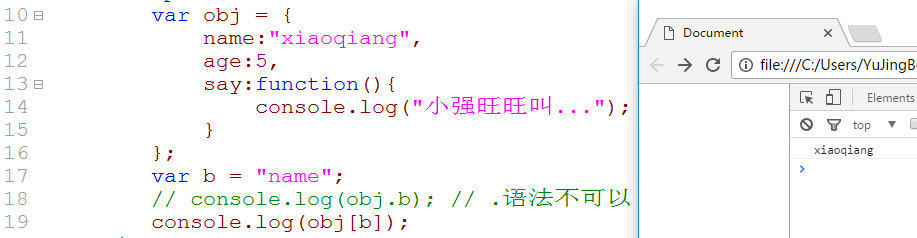


除了这个.语法，还有一个[]语法：



**总结：访问某个属性有两种，一种.语法，一种是[]语法。**

如果说我们一个属性是一个变量，那么只能通过[]语法：

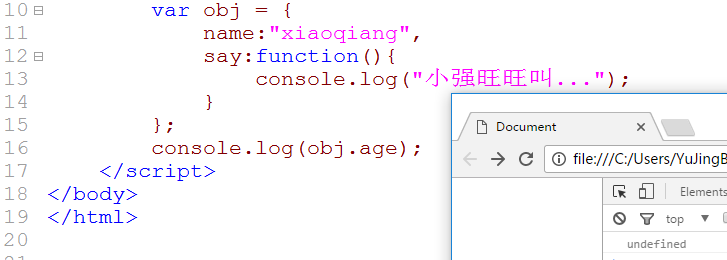


**undefinted目前为止，哪里可以出现：**

1，变量没有赋值

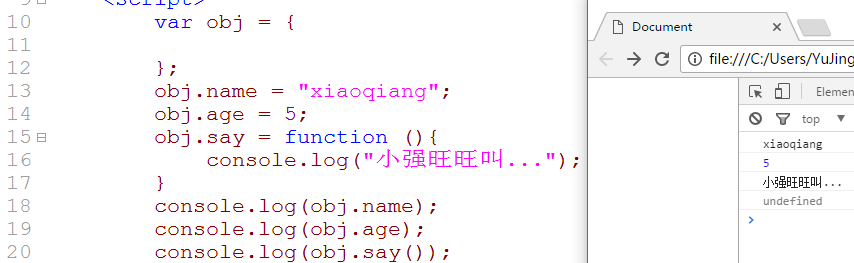
2，函数的默认返回值

3，访问对象的未定义属性

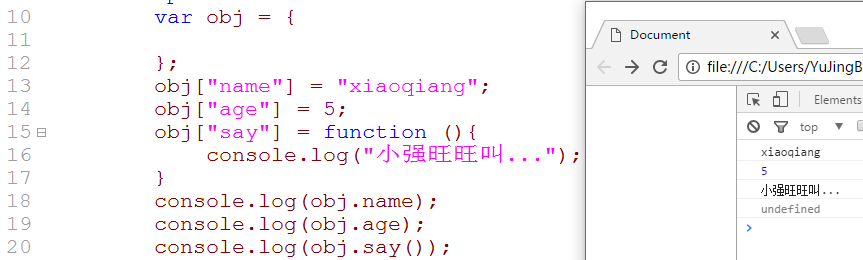


### 2，增加新属性

**我们可以通过.语法给一个对象添加新属性：**

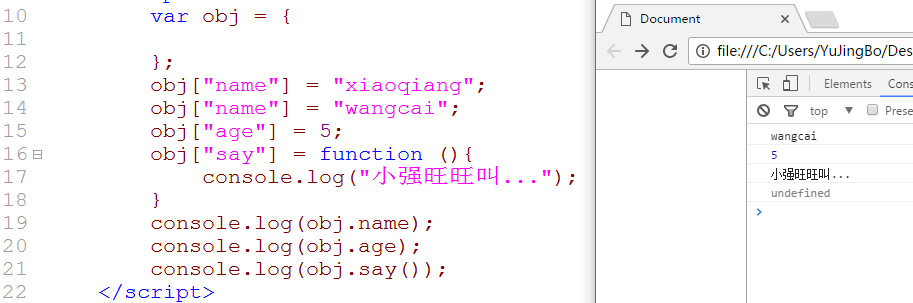


**还有一种方式，就是[]语法：**



**总结：添加新属性有两种，一种.语法，一种是[]语法。**

**注意的问题：如果添加了两个属性名一样的属性：**

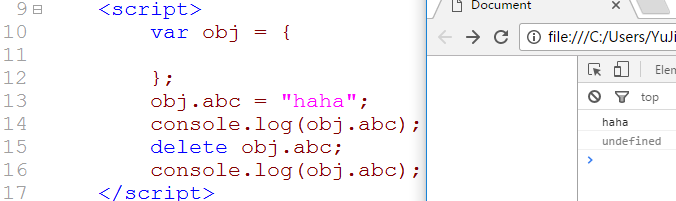


结论：同名属性将会被覆盖。

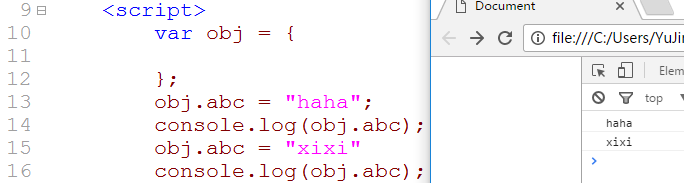
### 3，精细化设置对象的属性

**属性的四个特征:**

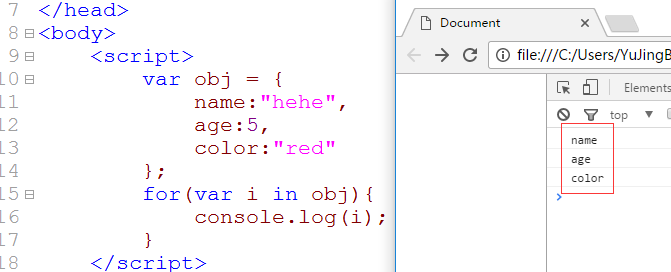
configurable: 是否可以删除。 默认为true 可以删除：



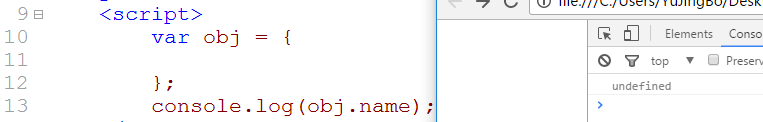
writable: 是否可以修改。 默认为ture, 可以修改：



enumerable:　是否可以枚举。可以使用 for in 默认为ture, 可以枚举：



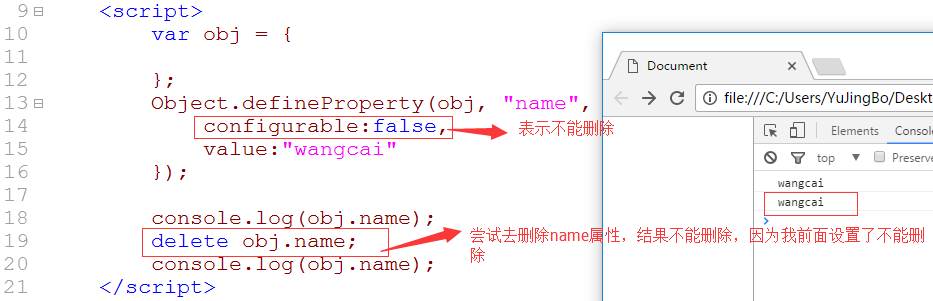
value: 值。 默认值为undefined



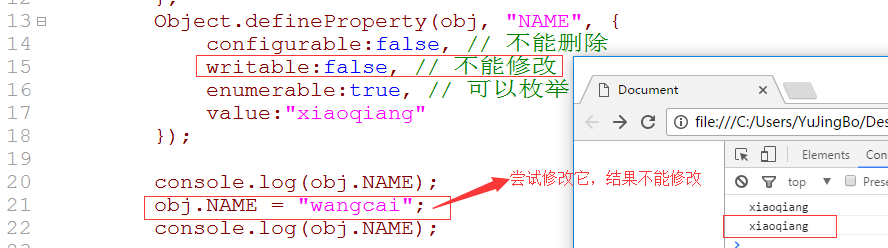
**精细化设置对象的属性**

格式：Object.defineProperty(对象名，“属性名”，{上面的四个特征});

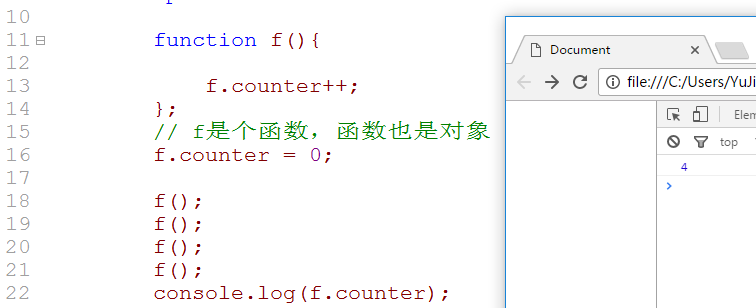
代码如下：



**案例一：设置一个只读属性，模拟常量**

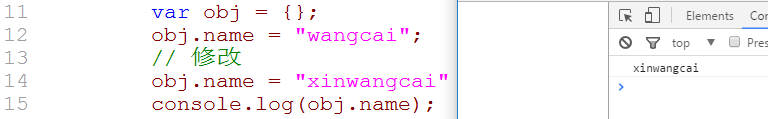


**案例一：模拟函数被调用的次数**

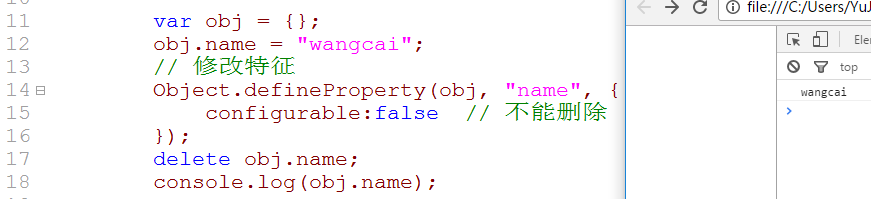


### 4，修改属性

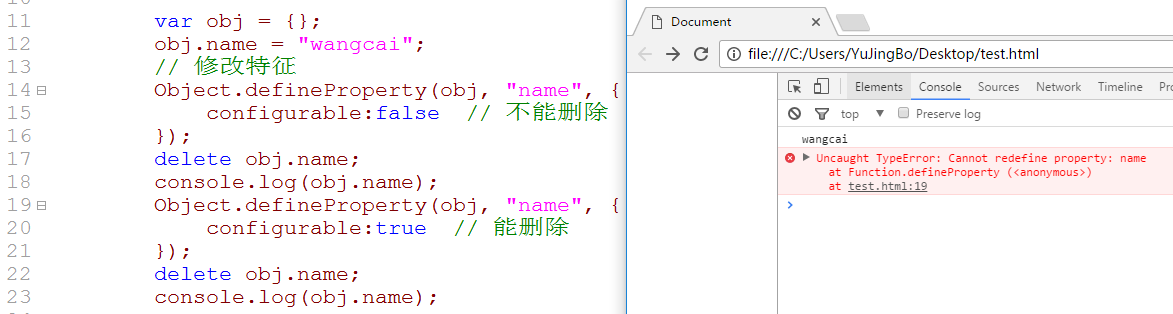
**1，修改属性值**



**2，修改特征**



**注意点：如果一旦设置了不能删除，那么就不能再重新设置回来：**



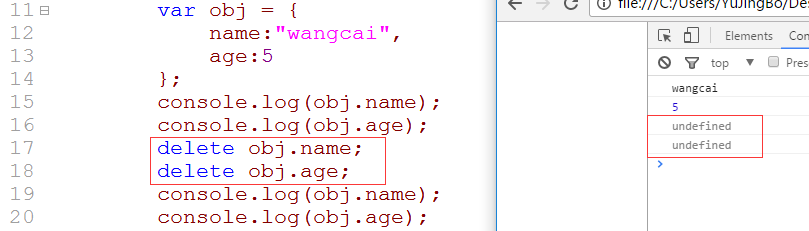
类似于，一个变量已加了var声明，则不可能再转化成不加var声明的。

### 5，删除属性

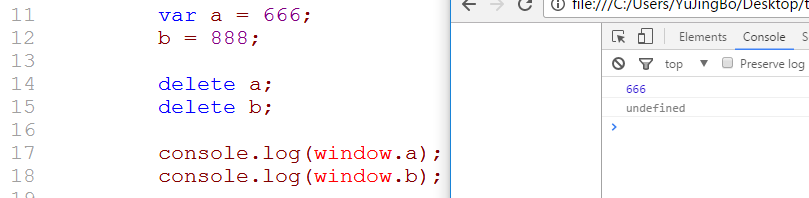
**格式：**

删除属性：delete 对象名.属性名

**代码如下：**



**深入理解：**加var变量为什么不能删除，不加var变量为什么可以删除？



遗漏声明的变量就是window对象属性，是可以被删除的。

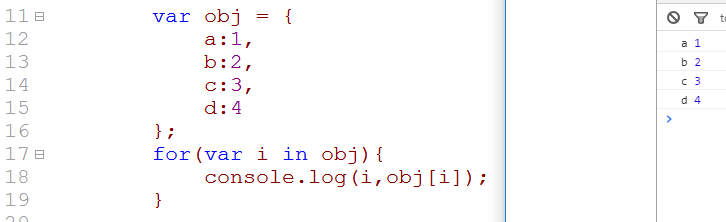
加var声明是不可以被删除的，为什么？因为使用var添加属性有一个特性：configurable:false。

### 6，for in

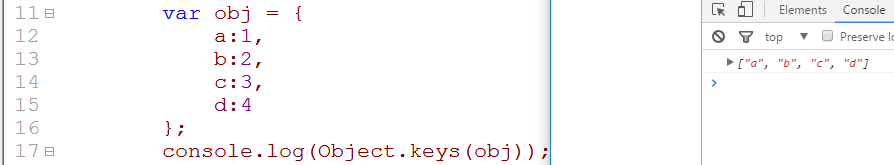
**作用：**用来遍历对象

**格式：**for(属性名 in 对象名){}

代码如下：



### 7，Object.keys()



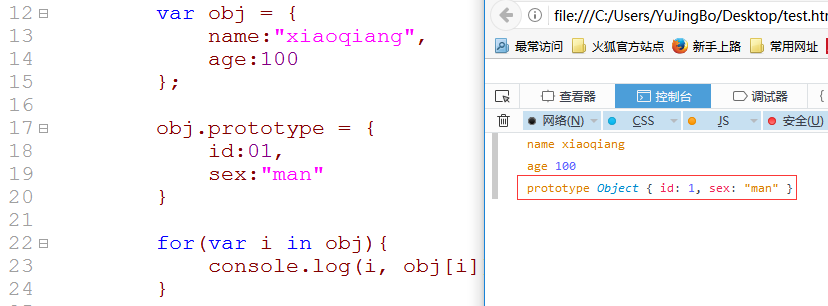
最终得到的是字符串数组。

**for in object.keys()的区别？**

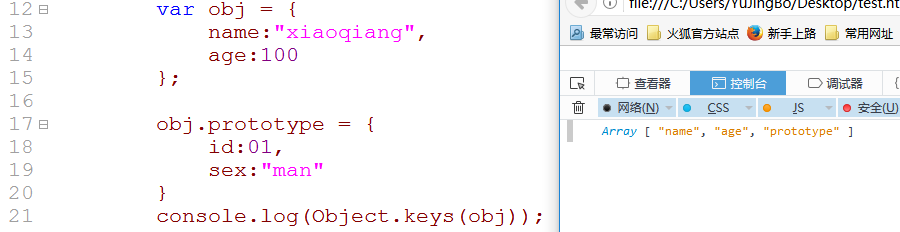
* for in 可以遍历出它自己本身的和它原型对象的属性
* object.keys()只能遍历出它自己本身的

**它们的区别代码说明：**

**for in**

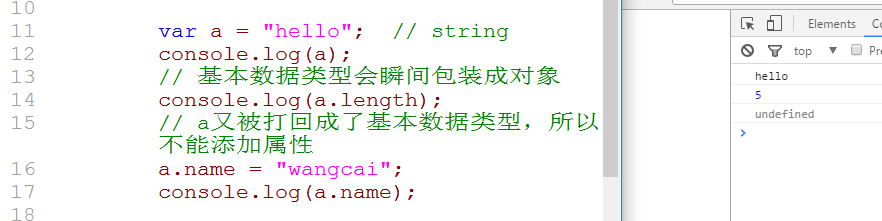


**Object.keys()**



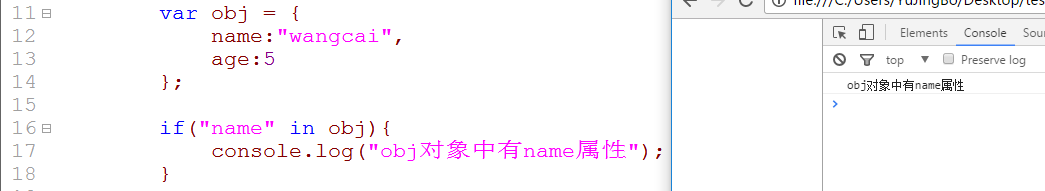
## 给基本数据类型的变量设置属性是无效的

基本数据类型会瞬间包装成对象。这个包装是瞬间的，包装完后立马被打回成基本数据类型。给基本数据类型是不能添加属性的。

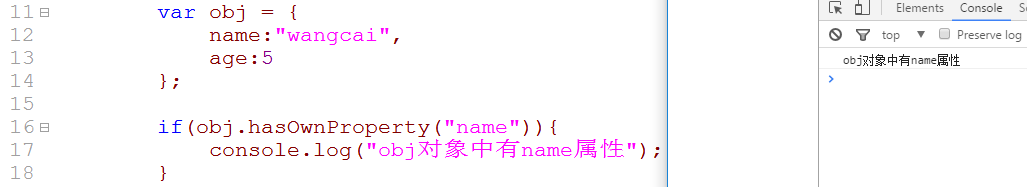


## 如何判断对象是否具有某个属性

**in : 会判断一个属性是否在一个对象中**



**hasOwnProperty:** **会判断一个属性是否在一个对象中**

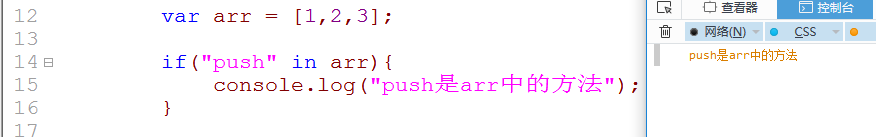


**in和hasOwnProperty的区别？**

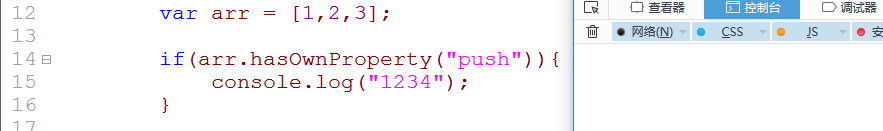
* in 除了找自己本身的属性，还会去找它原型对象中的属性
* hasOwnProperty只会在它本身属性中找

区别例子：

in:



hasOwnProperty:



说明了,hasOwnProperty只在自己属性中找。

## 对象与数组的相同点，与不同点

**JS数据类型分两类：**

* 基本数据类型
* 引用数据类型

函数

数组

对象

**数组也是一个对象，只是这个对象比较特殊，类比矩形与正方形**

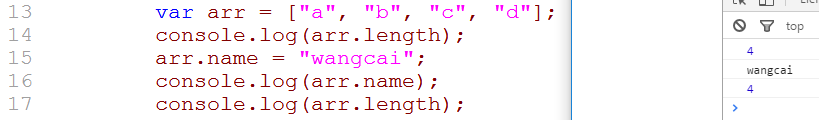
**相同点：**

都可以使用数字下标。

都可以存储很多数据。

**不同点：**

给数组设置字符串下标时，它的length不会增加：



使用forin输出数组时，它会把它原型对象中的属性也输出：

**结论：**

当遍历一个数组时，使用for

当遍历一个对象时，使用for in

## 利用属性的特性完成两个例子

**案例一：数组去重**  
 如var arr = [1,2,1,4,4,3,5,3], 把重复的去掉， 得到一个新数组：newarr = [1,2,4,3,5]

**分析：**

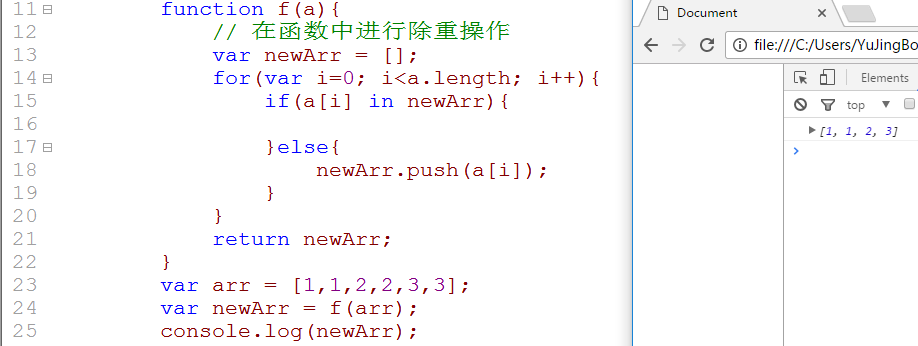
1，定义一个函数f

2，newArr = f(oddArr)

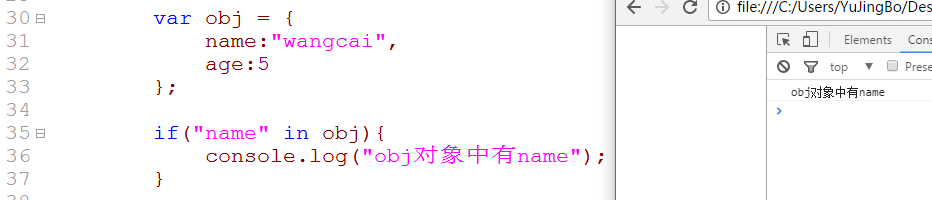
3，重点是f的功能

**代码如下：**

下面的代码的思路是不对的：



in的用法：

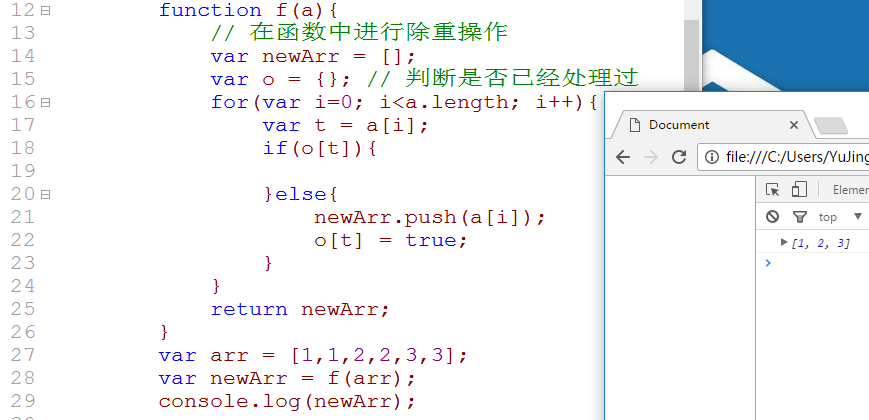


in是判断属性名是否在对象中，我们说数组也是对象，var arr = [“a”,”b”,”c”]; 对象这个数组来说，下标索引是属性名，里面的”a”, “b”,”c”它们是属性值。我们要去重的是属性值，不是属性名，所以上面的代码不能实现我们想要的功能。

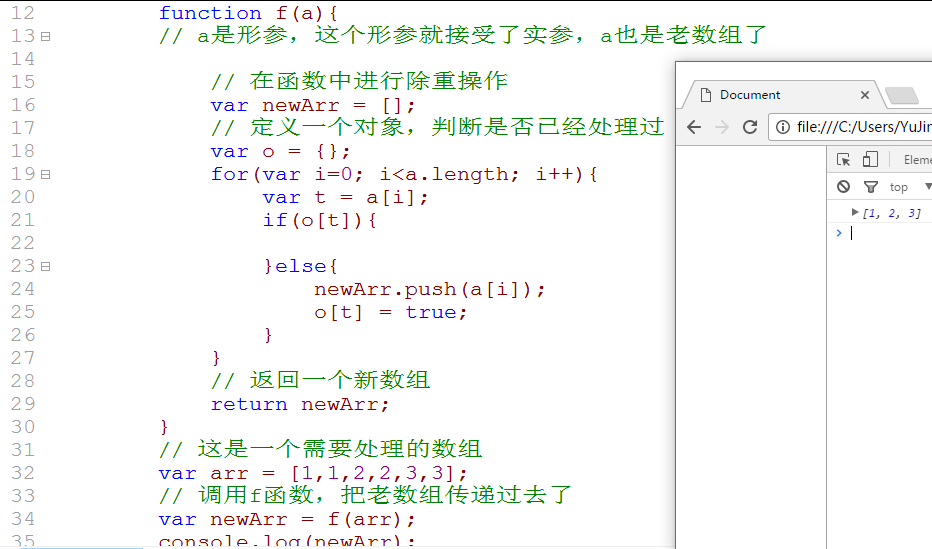
**正确的做法：**

应该判断属性值是否存在，不能判断属性名。

正确代码如下：



分析如下：把代码分析3遍。



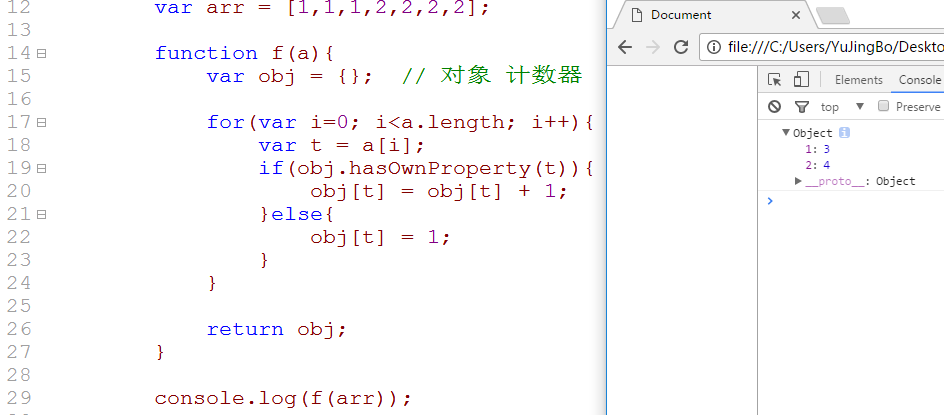
**案例二：统计数组中各个元素出现的次数**

如 var arr = [1,2,3,4,1,2,3,4,1,1]；对于这个数组统计1出现多少次，2出现了多少次，....

分析：

1，搞一个计数器，当我第一次遇见一个元素时，把计数器设为1，当我再遇见同一个元素时，把计数器加1，.....

代码：

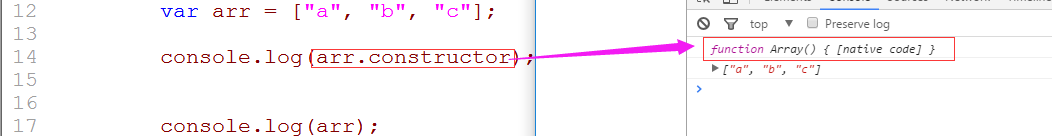


# 四，对象是由函数创造

**一切皆对象**：基本数据类型数据是对象（包装对象），array也是对象，funciton也是对象，object也是对象。

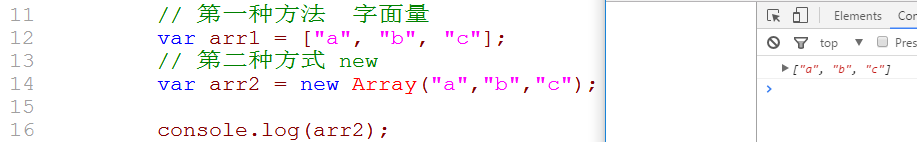
## 数组是由函数创造

对象中有一个特殊属性，叫：constructor 这个属性指向了创建这个对象的那个函数。如下：



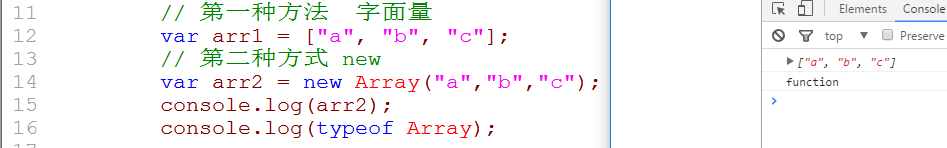
上面的代码说明了，数组也是函数创建的，这个函数的名字叫：Array. 这个函数就是用来创建数组的。

复习：创建数组有几种方式？



说明：1，第一种方法是第二种方法的一个快捷方式，也叫“语法糖”

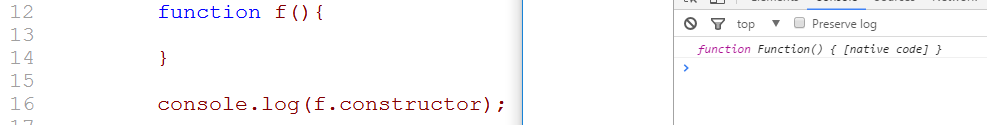
Array是一个函数，通常我们把这类函数起了一个名字：构造器：



## 函数是由函数创造

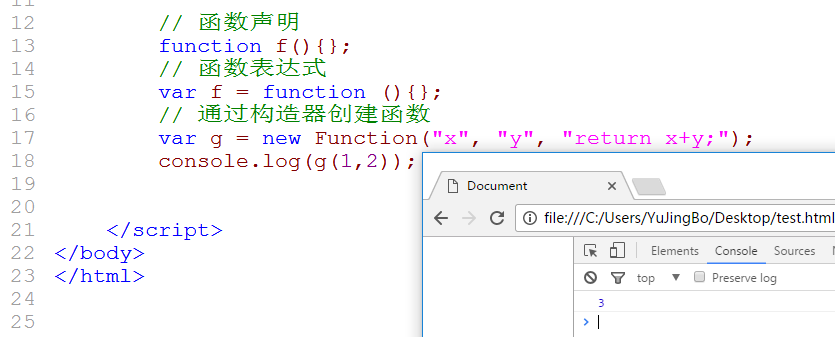
前面说过，函数也是一个对象，对象有一个特殊的属性：constructor ， 这个属性指向创建这个对象的函数。

代码如下：



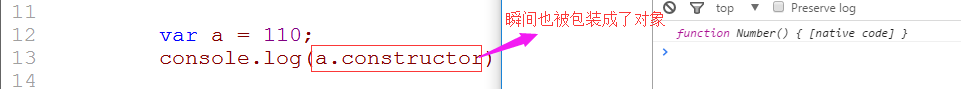
说明：这个函数比较特殊，它的名字是Function， F还是大写的，它也是构成器。也说明了，我们平常写的函数也是由Function创建的。

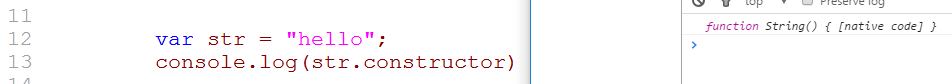
复习：创建函数有几种方式：



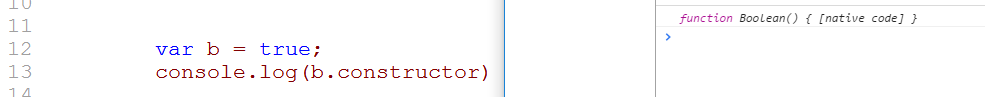
## 基本数据类型的变量可以由函数创造

**数值类型：**

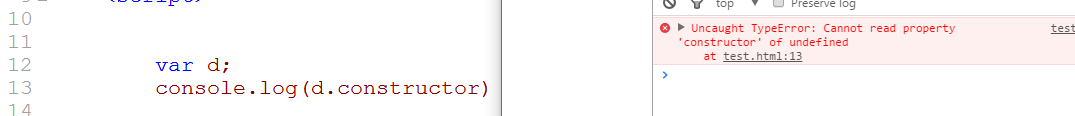


**字符串类型：**

**布尔类型：**

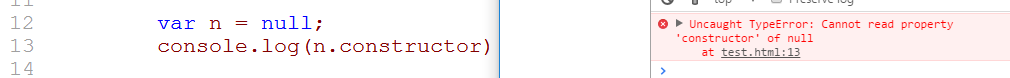


**undefined:**



说明了：没有Undefined。

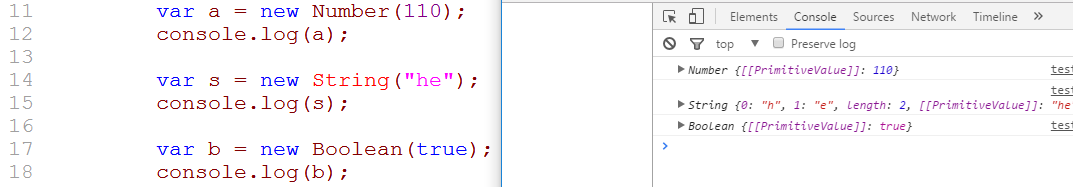
**null:**



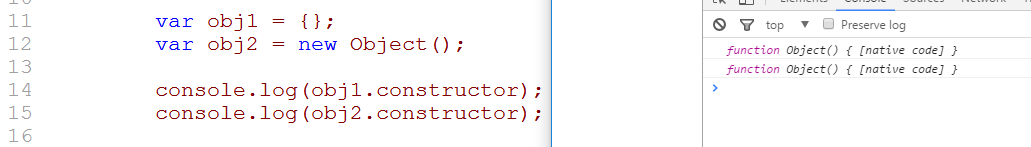
总结：

对于基本数据类型：也有几个构造器：Number(), String(), Boolean()

创建基本数据类型的数据两种： 字面量， new



## 对象由函数创造



函数名字是Object, Object也是一个构造器。

**结论：一切对象由函数创建**

# 五，原型与原型链

## ，什么是原型

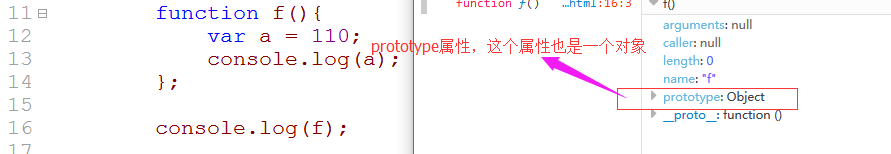
**prototype就是原型**

### 1，每一个函数都有一个prototype属性

**函数：**

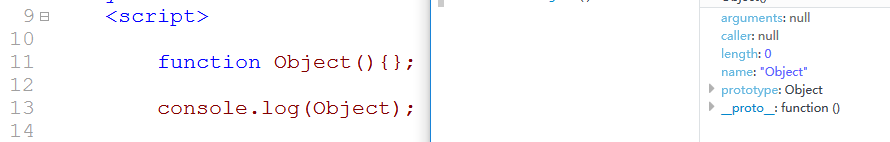
* 自定义函数
* 构造器

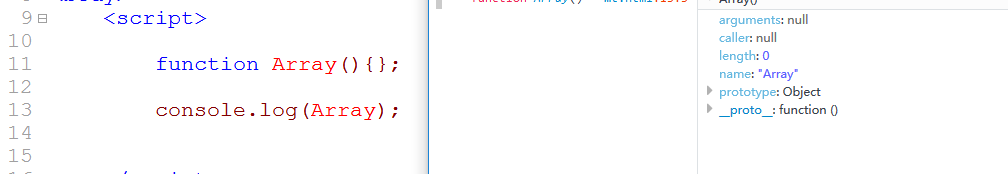
**自定义函数**



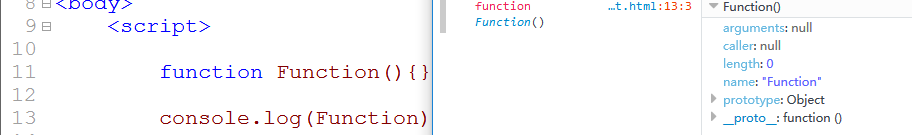
**构造器 Object, Array, Function, Boolean, String, Number**

**Object:**



**Array:**

**Function:**



....

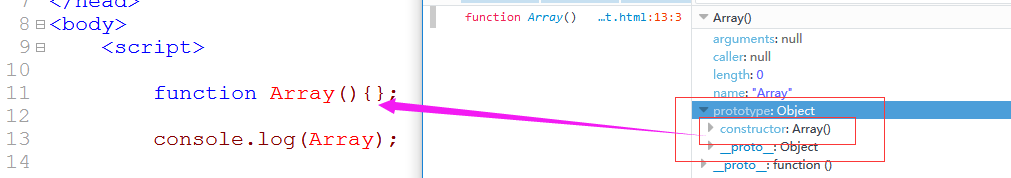
总结 ： 每个函数，都有很多属性，其中一定会有一个属性叫prototype。这个属性的值是一个对象。

这个prorotype属性，它的属性值是一个对象，这个对象中有很多方法，但是一定有一个方法叫:constructor constructor方法指向这个函数本身：代码如下：

自定义函数：



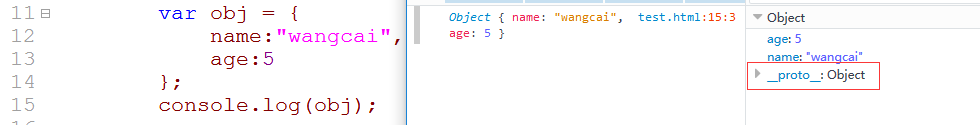
构造器，Array:



......

总结：第一个函数都有一个prototype属性，这个属性的属性值是一个对象，这个对象中有很多方法，其中一定有一个constructior方法， 这个方法指向函数本身。

### 2，每一个对象都有一个\_ \_proto\_ \_属性

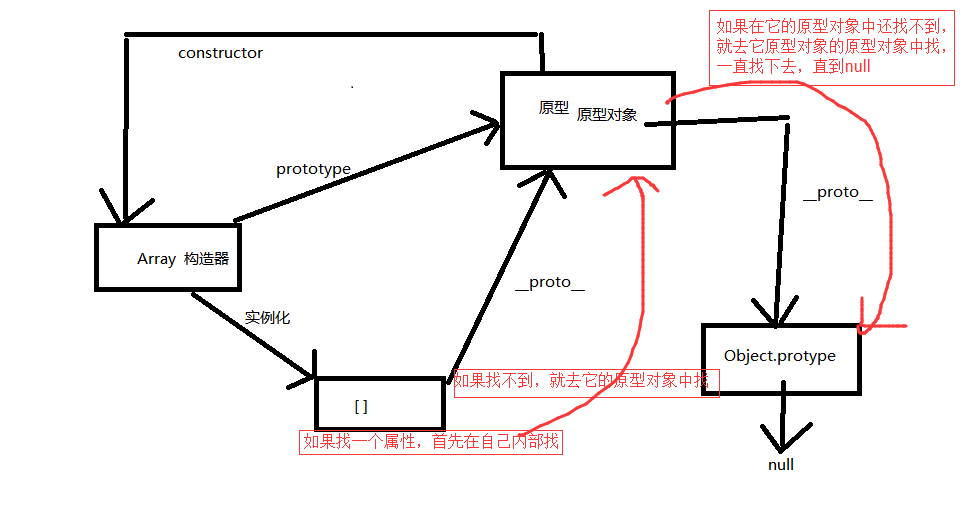


函数也是一个对象，所以函数有一个prototype属性，也有一个\_\_proto\_\_属性：



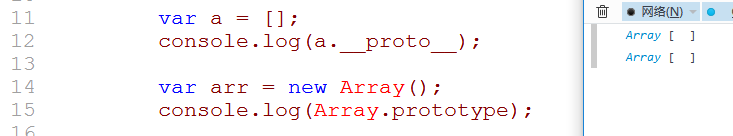
**每一个对象都有一个\_ \_ proto \_ \_ 属性，这个属性 指向 创建这个对象的函数的prototype.**

**根据上面这句，举个例子，以数组为例子：**

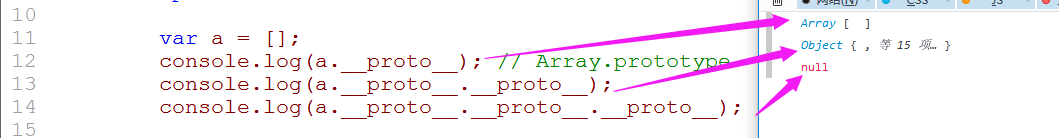


## ，构造器的原型

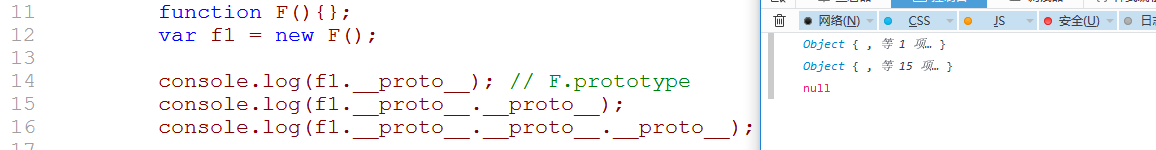
对象的\_\_proto\_\_和创建这个对象的函数的prototype指向是同一个对象：



对于数组 ，对于\_\_proto\_\_，一直向上找，就形成了原型链，直到null

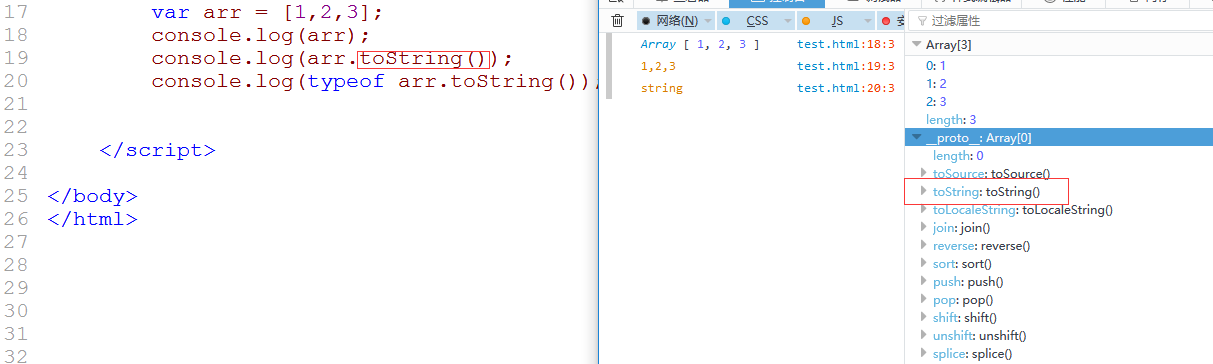


看一下函数：



## （3），原型链

如果访问一个对象的属性时，先在这个对象自己的属性中找，如果找不到，则沿着\_\_proto\_\_这个属性向上找，如果\_\_proto\_\_这个对象还找到不到，就会在\_\_proto\_\_对象的\_\_proto\_\_属性中找，依次找到去，这就是原型链。

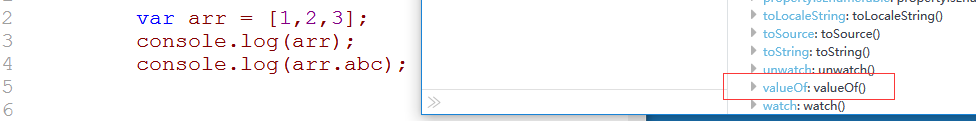


再举个例子：

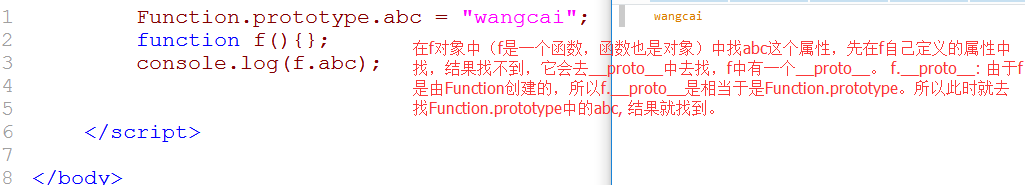


上面的例子说明了，arr这个数组里面有一个valueof的属性。我们看一个，arr里面这个valueOf这个属性在什么地方？

我们在arr这个对象的\_\_proto\_\_的对象的\_\_proto\_\_这个对象中找到了valueOf这个方法。



**理解原型链：**

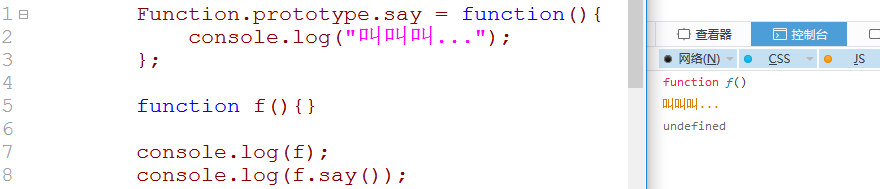


原型链强化：

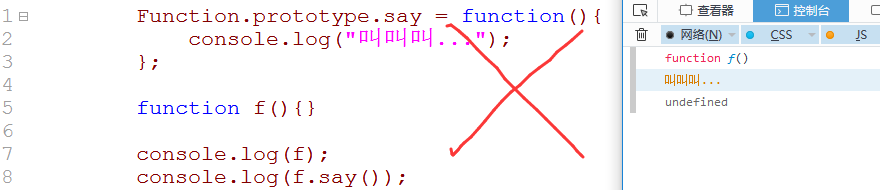
看如下代码：



对于上面的代码，它会报一个错，因为在f里面没有定义say，那么它会去它的原型对象中找，在原型对象中也找不到，又去它原型对象中的原型对象中找，..... 最终也找到，就报了一个错。现在，我们给它的原型对象添加一个say的方法：

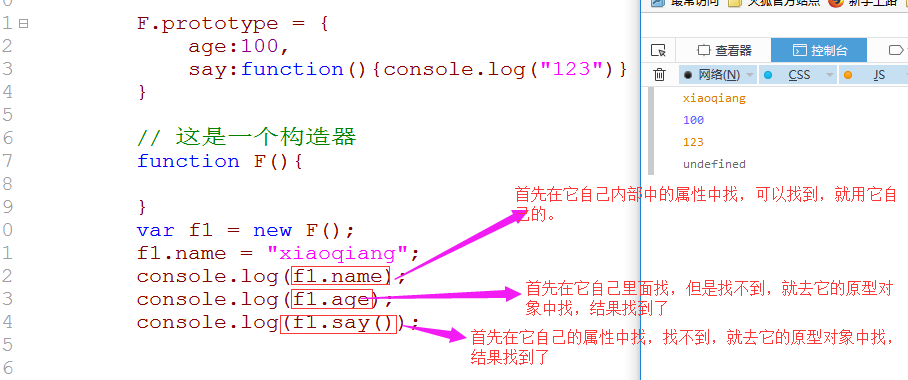


注意：我们不要去修改构造器的prototype属性。因为可能会覆盖。



**什么时候可以去覆盖？**

**当我们自己写了一个构造器，那么就可以覆盖了！**



# 六，JSON

## （1）JSON介绍

**1，什么是JSON**

**2，如何写一个JSON**

**3，怎么去检验一个JSON是否正确**

**什么是JSON？**

JSON（JavaScript Object Notation JS对象表示法），是一种轻量级的数据交换格式。JSON容易让人们进行阅读和编写。

**理解：**

1，JSON指的是JavaScript对象表示法

2，JSON是轻量级的文本数据交换格式

3，JSON独立与编程语言

**JSON只有两种结构：**

* 数组 []
* 对象 {}

**JOSN语法规则：**

1，数据在键值对中

2，数据由逗号分隔

3，{}保存对象

4，[]保存数组

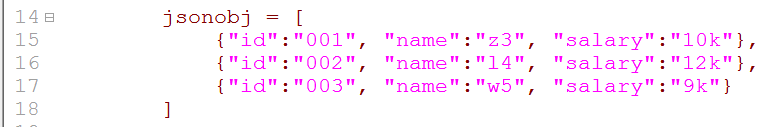
5，键名必须要使用“”,不能省略，也不能写’’

6，值可以是如下的数据类型：

数字，字符串，布尔值，数组，对象，null, 函数不可以

**写JSON:**

**数组结构：**

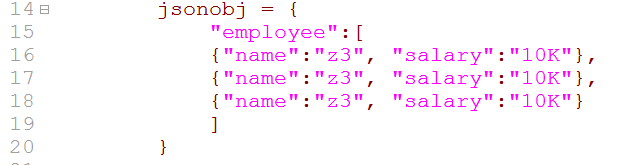


如何检验我们写的json数据是否正确呢？ 在线检验：<http://www.bejson.com/>

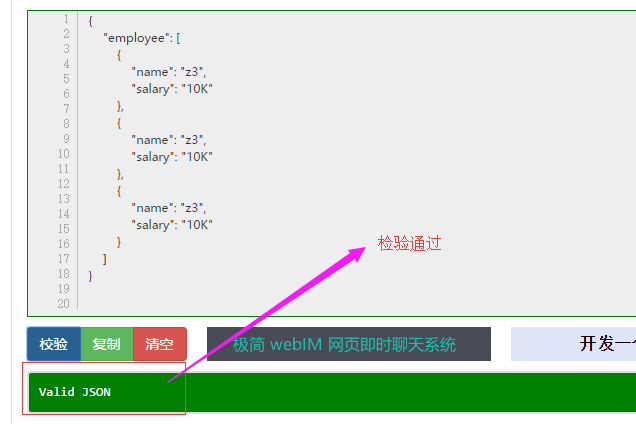
现在我们检验一下我们写的是否正确：



**对象结构：**



也去检验一下：



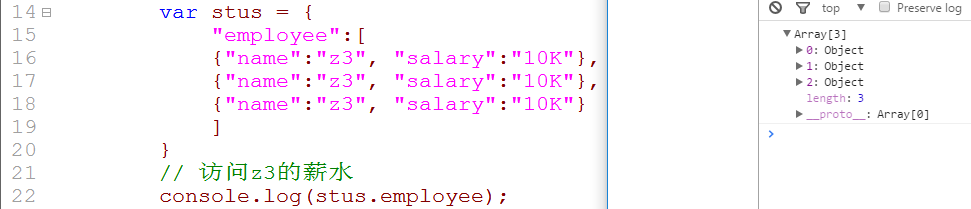
如果你写的格式不对，那么就通不过检验：



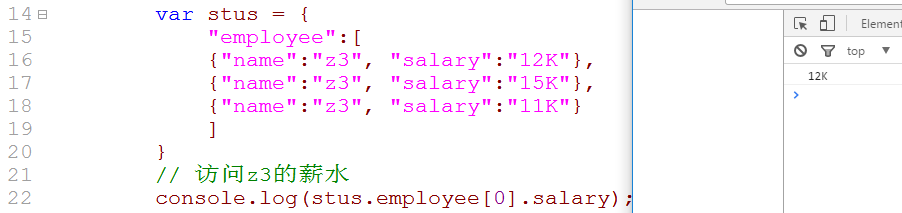
## JSON操作

### 1，访问

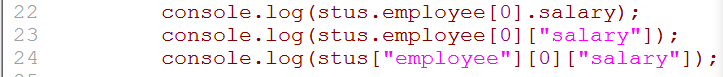
下面的访问，只能拿到一个数组，数组中有三个对象：



下面的写法会拿到z3的薪水：



除了上面的写法，还有其它几个写法：



**总结：**

1，对于对象，可以使用.获取属性，也可以通过[]来获取属性

2，对于数组，只能通过[]来获取属性

### 2，修改

### 3，删除

### 4，遍历

说明：对于数组结构的，用for遍历，对于对象结构的，用for in

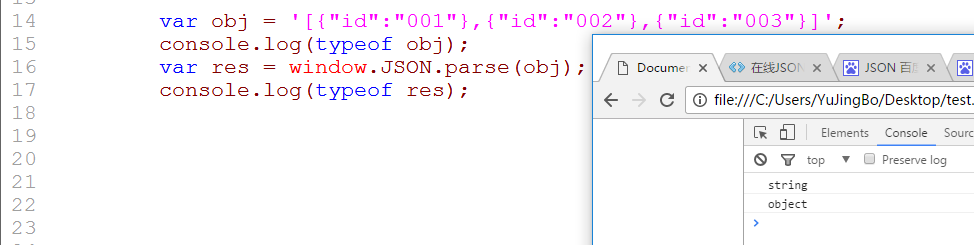
## JSON对象

为了方便我们去操作JSON数据，每一种编程语言都提供了相应的办法，来操作我们的JSON数据，在JS中就有一个JSON对象，JSON对应就可以方便的来操作我们JSON。

JSON字符串：就是把JSON对象前后加一个单引号，就变成了JSON字符串。

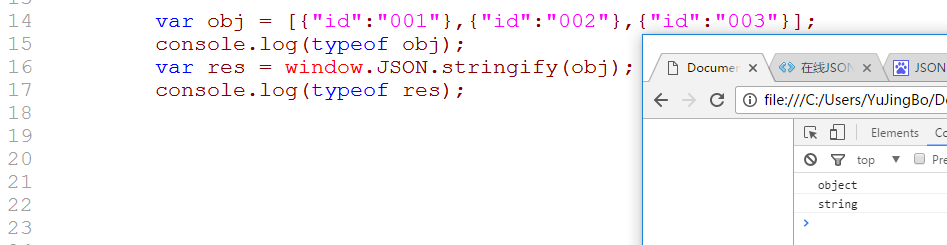
### 1，把一个字符串转成JSON对象

JSON.parse() 把一个字符串转化成JSON对象



### 2，把一个JSON对象转成字符串

JSON.stringify() 把一个JSON对象转成字符串



# 七，this

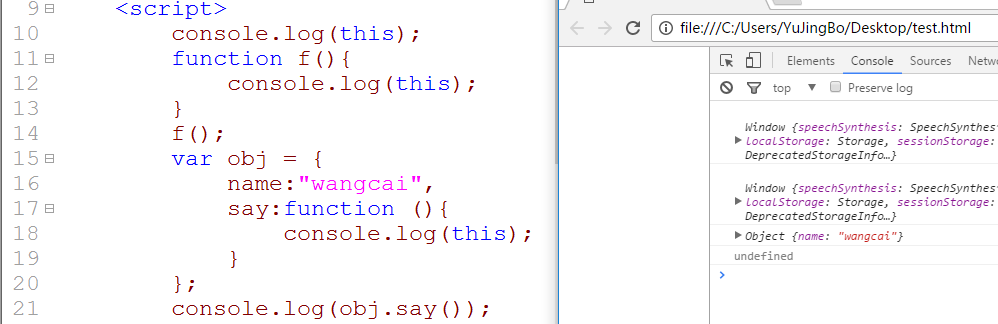
前面在事件中学过this， 那个时候，this是指事件源。

this “这个” that “那个”

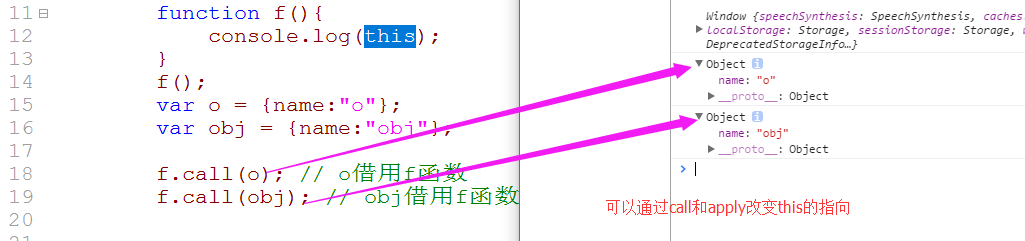
## （1）三条定律

### a, 到处都有this

只要能写js代码的地方，都可以访问this。



### b，this的值随时可能改变，只有在调用时才能确定



### c，不能手动修改this的值



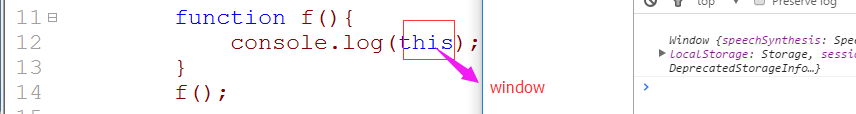
### 总结：this到底是什么

**this也是一个对象。这个对象指向当前代码中所需要的各种资源。**

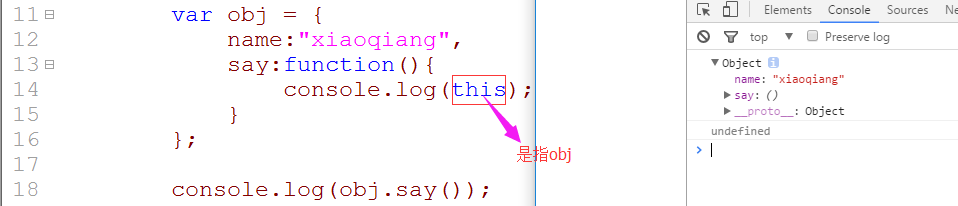
## （2）四种情况

一般情况下，this是出现在函数的内部。它出现在函数的内部，那么我们分了四种情况讨论。

### 1，函数不是某个对象的属性，而是一个普通的函数。也就是说它不是方法。

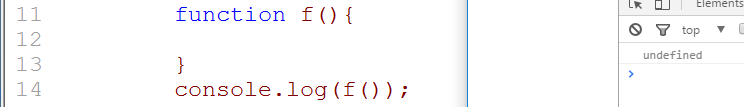


### 2，函数是某个对象的方法

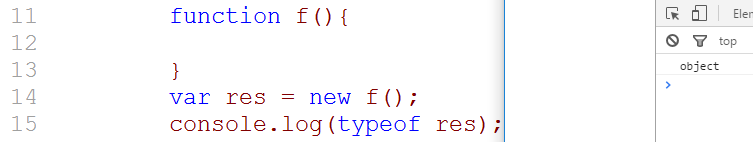


### 3，如果在调用函数之前加上一个new, 它会创建并返回一个对象，而这个函数内部的this就指向个对象。

如果一个函数没有返回值，那么它默认返回undefined:

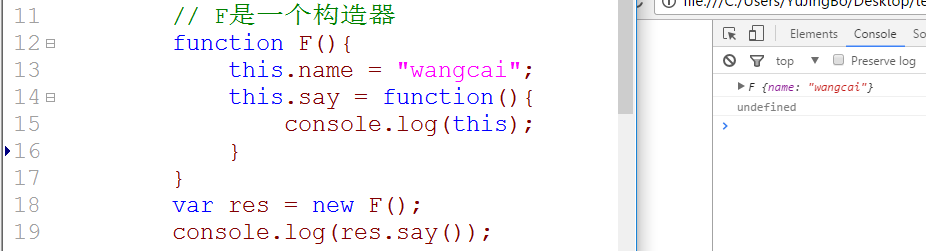


上面的写法，没有加new 的， 加上一个new ，那么它会返回一个对象：



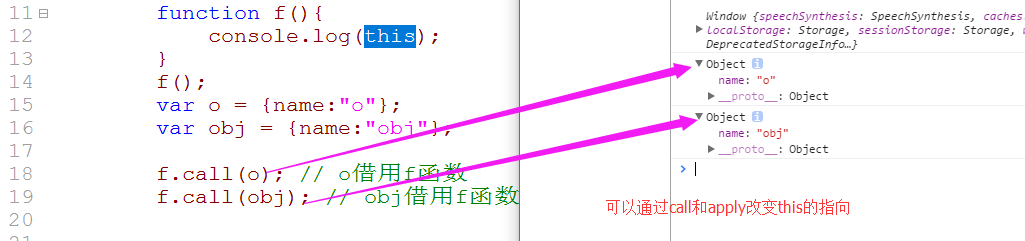
说明，我们前面都是去new一个构造器， new String, new Function, new Boolean....

如果在调用函数之前加上一个new, 它会创建并返回一个对象，而这个函数内部的this就指向个对象。



### 4，可以通过Function对象的apply和call就去来指定函数内部中的this指向

代码如下：



## call()和apply()

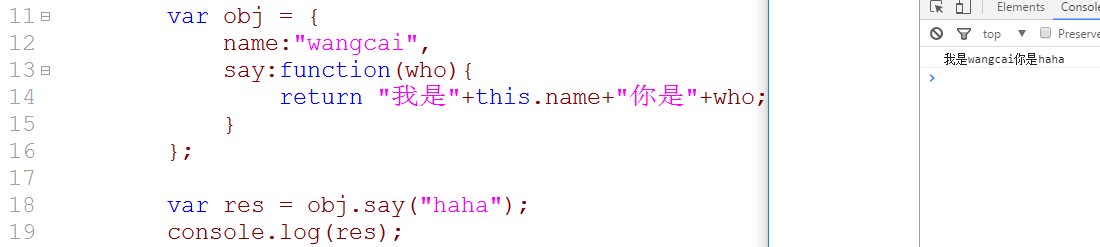
在函数中有两个方法，一个call(), 还有一个apply()

作用：它可以让一个对象去“借用”另一个对象的方法。

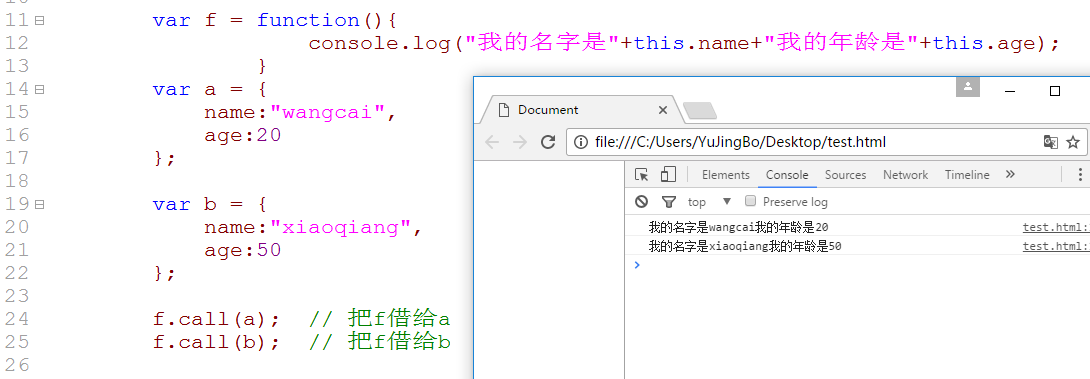
把方法中的this用传入的对象来代替。

代码如下：

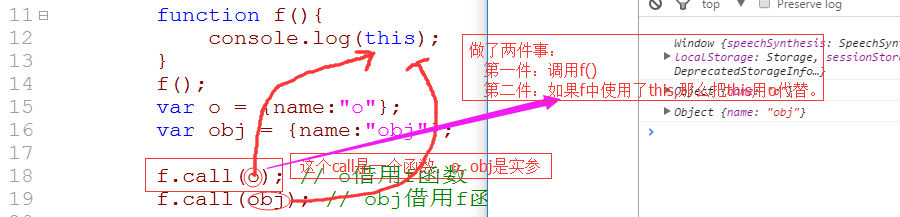
我们写一个对象，在这个对象中定义一个方法：



再写一个对象，去借用上面对象中的say方法：

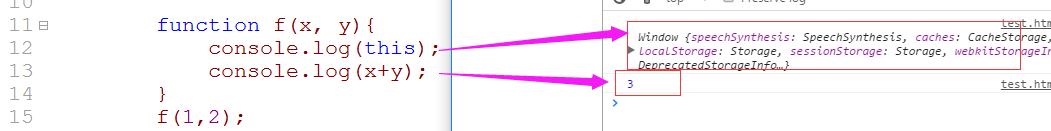


进一步理解call:

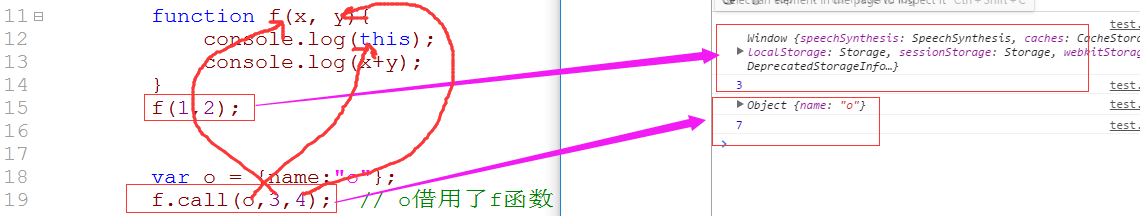


上面的说函数是没有形参的，我们讨论一下有参数的情况：

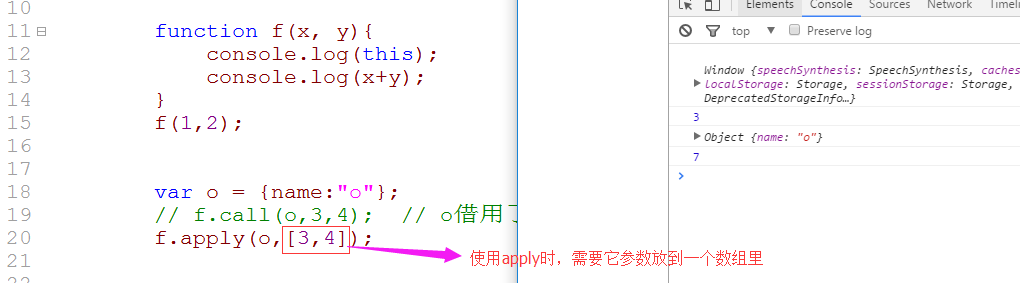
没有使用call,如下：



使用call如下：



call和apply是一样的（没有参数），只是它的参数不一样。



**总结this:**

如果在函数的内部有this，如何如何确定this的值？

1，如果一个方法前面有对象，this就指向这个对象.

例：obj.say()， say中的this就指向obj

2，如果一个函数没有说明是哪个对象的，那么这个this肯定指向window

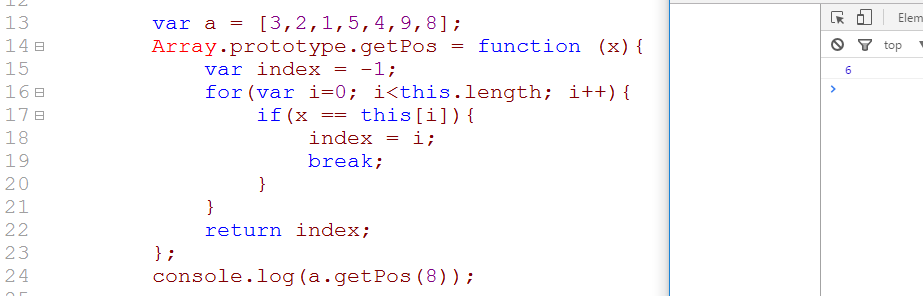
3，如果使用了call(),apply()那么this就指向这两个方法中的第一个参数

例：f.call(obj) 此时f中的this就指向了obj

## 综合应用

### 1，在数组中找指定的元素

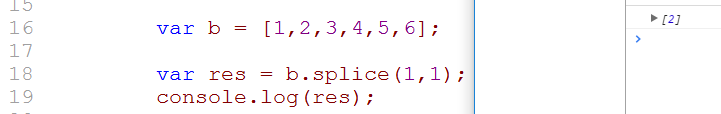
在数组中找指定的元素，如果找到，返回这个元素的下标。



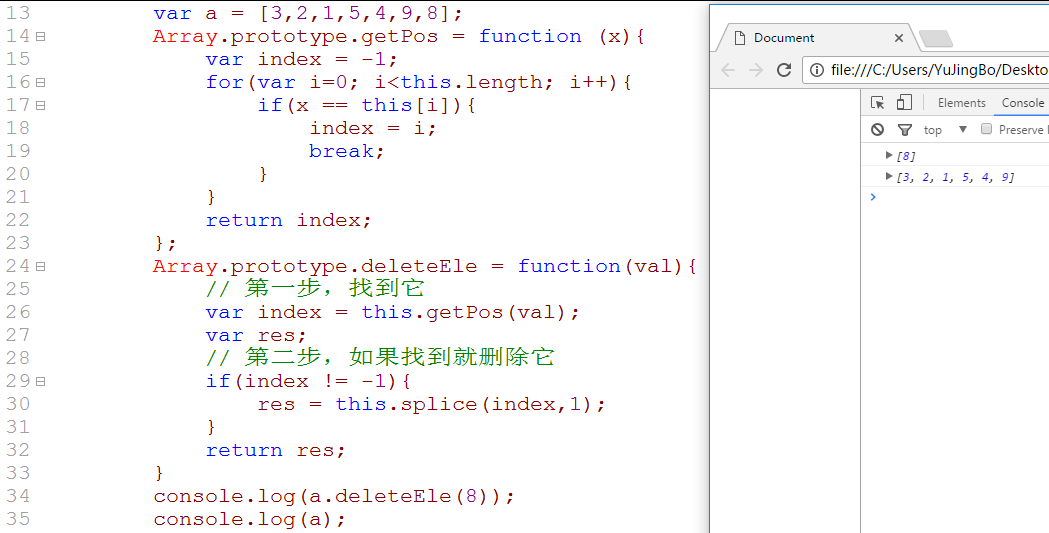
### 2，删除数组中指定的元素

splice() 从数组中删除某个元素，返回删除的元素。

splice()的用法：



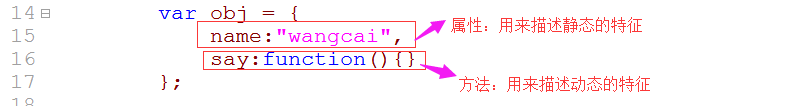
完整代码如下：



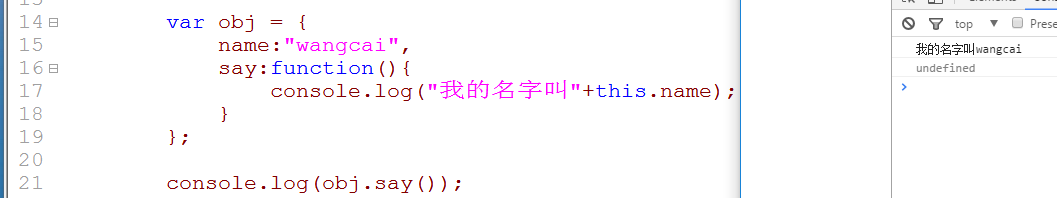
# 八，对象创建的N种方法

## 字面量方法

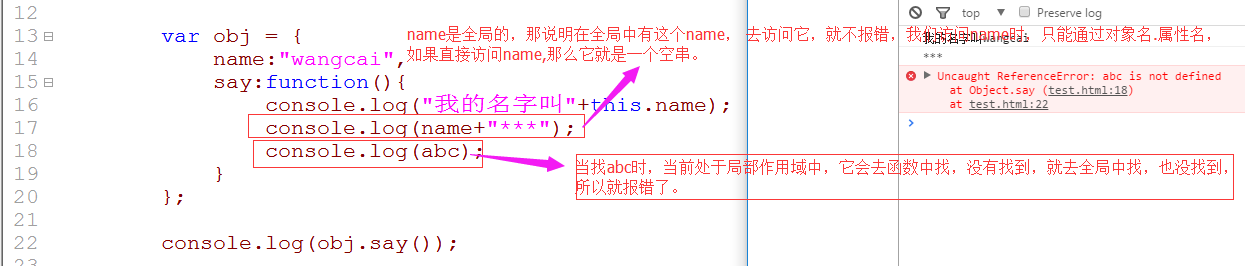
可以通过字面量去创建一个对象：



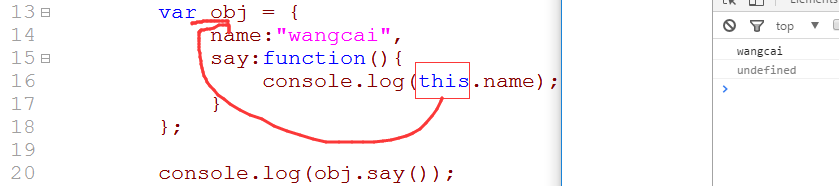
深入理解：当我们在方法内部去访问属性时，一定要加上this, 此时this就指向obj



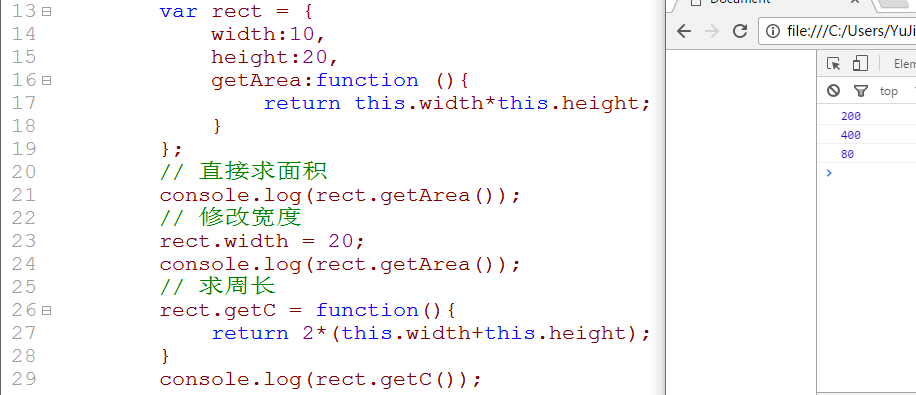
再分析一下：



一定要注意，当访问对象内部的属性时，一定要加上this:



**练习：使用字面量方式去创建一个矩形对象：**



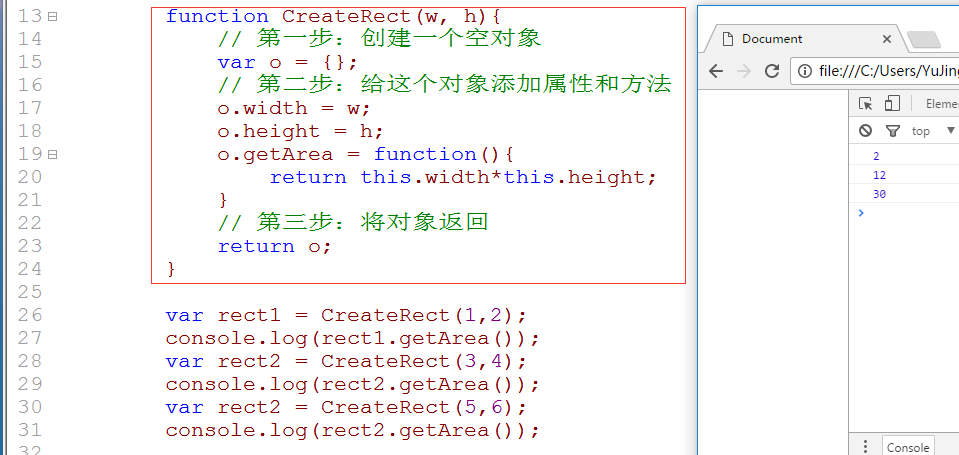
**字面量创建对象的有缺点：**

优点：直接了当，一目了然。

缺点：不能批量生产对象，对象只能一个一个的写。

## 工厂模式方法

为了解决字面量不能批量生产对象的问题。



**为什么叫工厂模式？**

1，需要原材料（提供属性值）

2，加工

3，返回成品

**工厂模式创建对象的有缺点：**

优点：可以批量产生对象

缺点：它生产的对象没有“商标”

## 构造器方法

### 1，深入理解构造器

A, 构造器的作用：产生对象。 对象是由函数（构造器）创建的

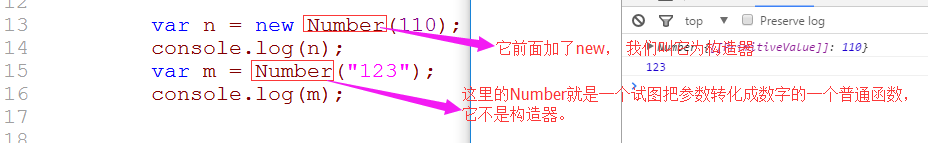
B，如果说我们在调用函数时，给它前面加上了一个new ,那么此时这个函数就 构造器。通常对于一个构造器，我们会把它的首字母大写。

C，任意一个函数都可以当作一个构造器

D，构造器是一个函数，也是一个对象，因为函数就是对象

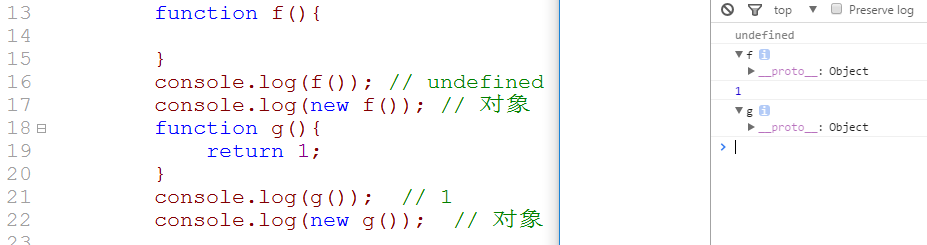
E，构造器首字母大写，但是不是强制，但是是行业默认的规则

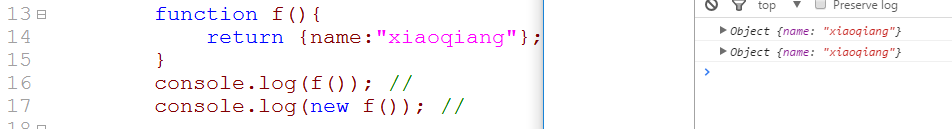
看一下，下面代码，有什么区别：



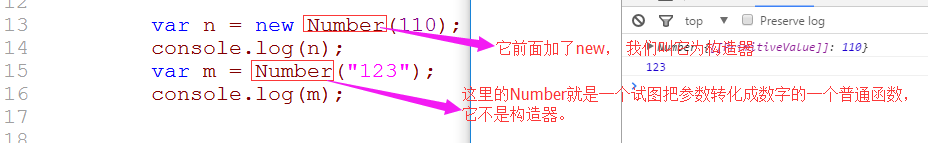
### 2，不加new会怎样？

如果你在一个函数前面加上了new， 此时这个函数就是一个构造器，无论你在这个构造器中返回了什么，都返回一个对象。



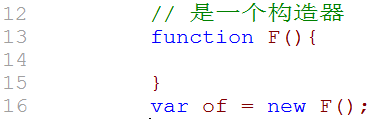
如果构造器返回值是一个对象，那么使用new和不使用new得到的返回值是一样的。

对于本地的一些构造器，如Number, String, Array等，它们如果不加new，它就是一个普通的函数，不是构造器。



### 3，new究竟干了什么事？

对于下面的代码，new做了什么事：



new一共做了4件事：

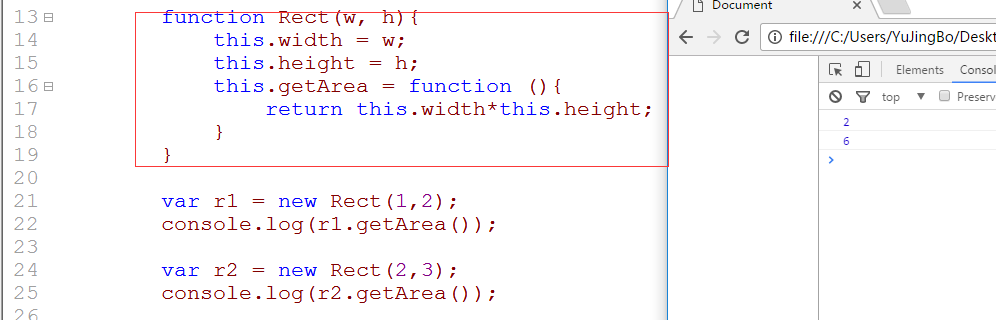
1，创建一个对象 var of = {};

2，F.call(of) 将this指向of

3，of.\_\_proto\_\_ = F.prototype // 上o具备原型对象上的方法

4，return of

### 4，构造器模式

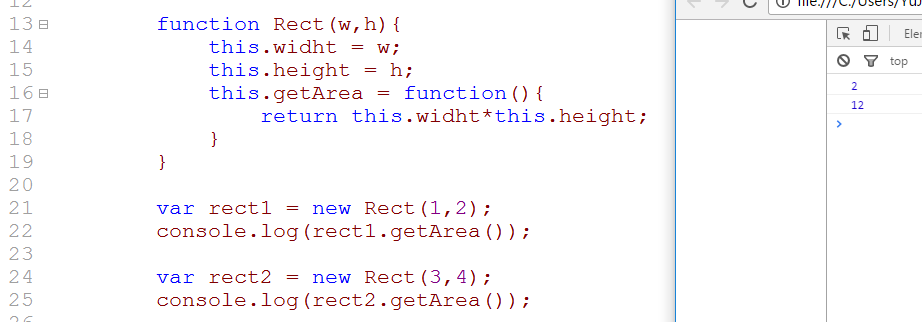


**构造器模式创建对象的有缺点：**

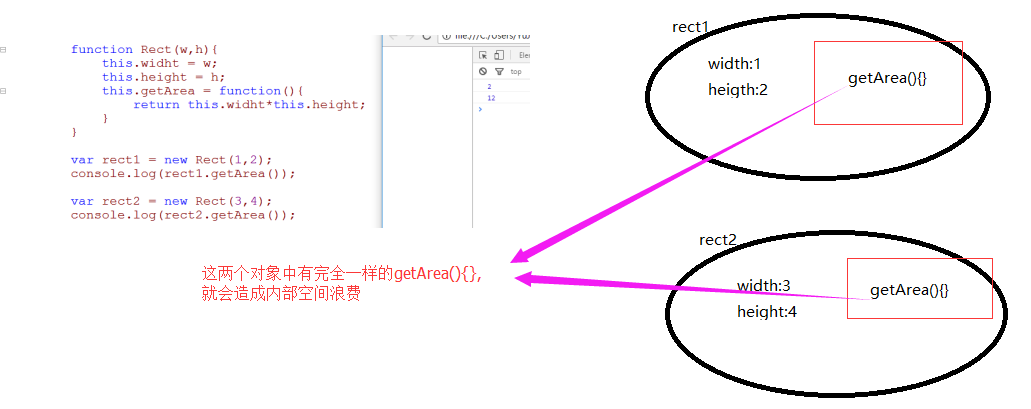
优点：解决了对象的来源不明问题， new的方式是其它大多数编程语言中的方式，被很多程序员接受，构造器模式也能批量产生。

缺点：创建出来的对象都有相同的属性，有可能浪费空间

分析构造器模式的不足的地方：

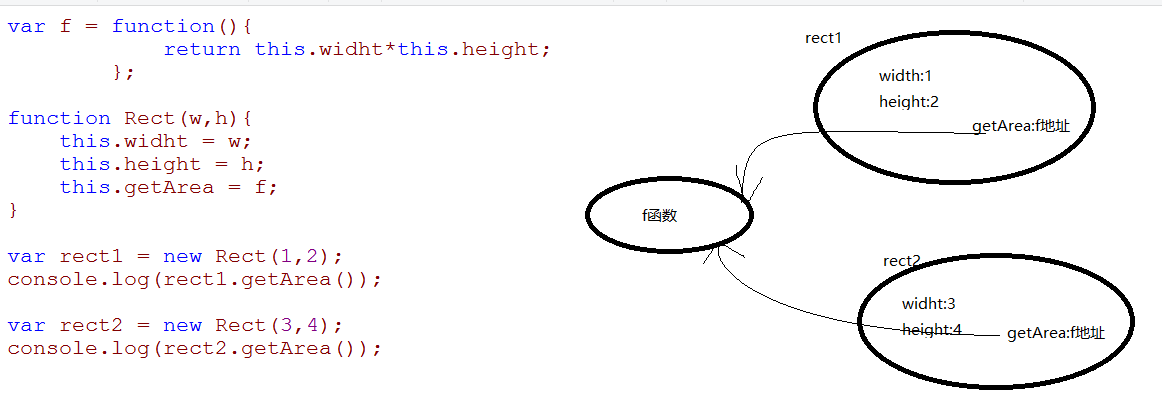


不足之处：会造成内存空间浪费



有没有办法去解决：

办法一：使用外部公有的函数代替对象中的方法



**使用办法一有什么不好地方：**

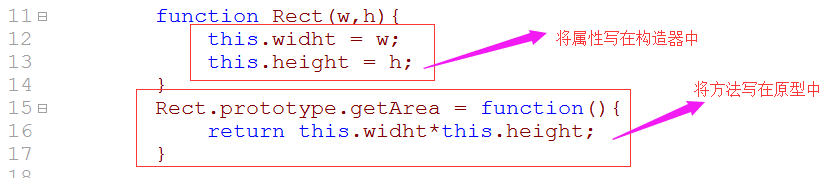
1，把一个实现具体功能的函数写在构造器外面，表面上看起来和Rect没有任何关系。

2，没体现出面向对象的三大特征。（封装，继承，多态）

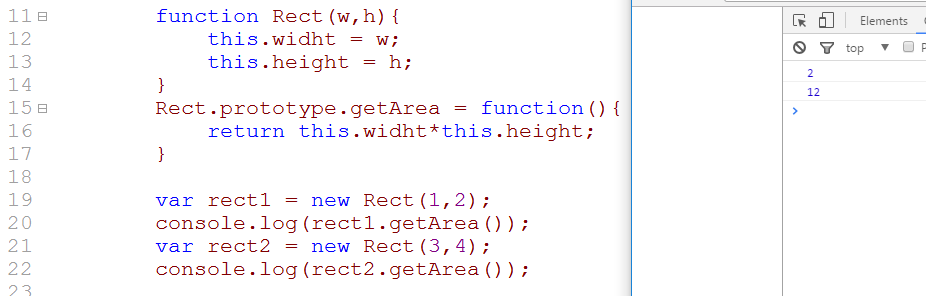
对于上面的办法一的不足之处，我们要学习下一下模式，原型模式。

## 原型模式

原型模式代码如下：



原型模式创建对象：



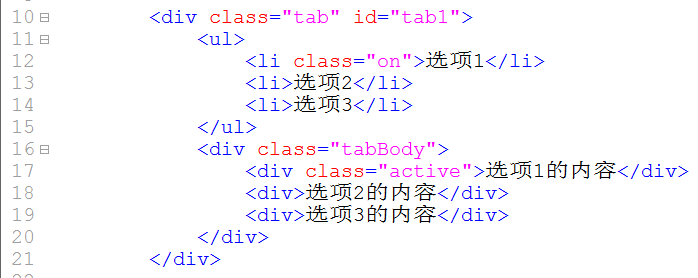
使用原型模式，就可以解决上面的办法一中的不足之处。

# 九，应用

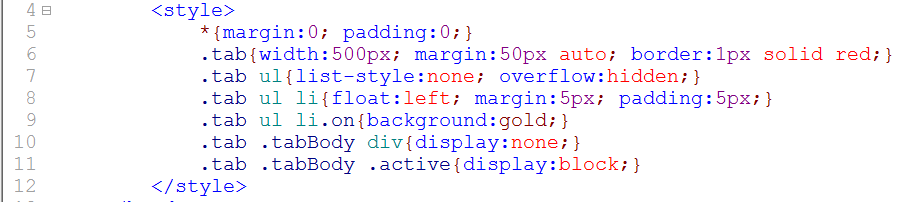
## 选项卡

### 1，使用传统方式实现选项卡

**HTML部分：**



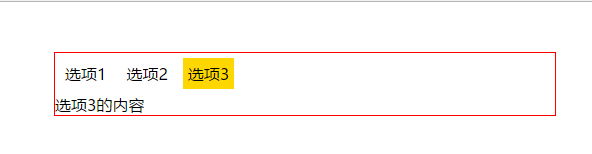
**CSS部分：**



**JS部分：**



**最终效果：**



### 2，如果我们页面中有多个tab时，我们需要封装代码（函数）

代码：



### 3，使用面向对象方式实现选项卡

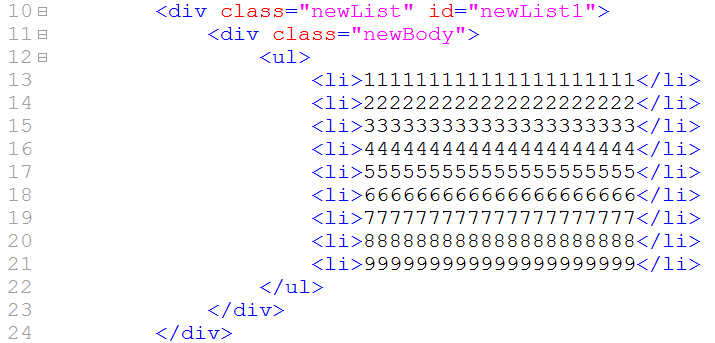
代码如下：



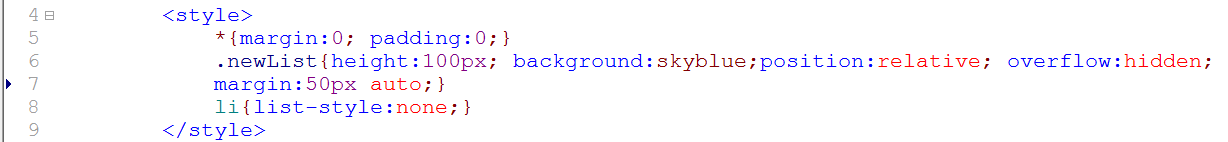
## 滚动的新闻列表

### 1，传统方式实现

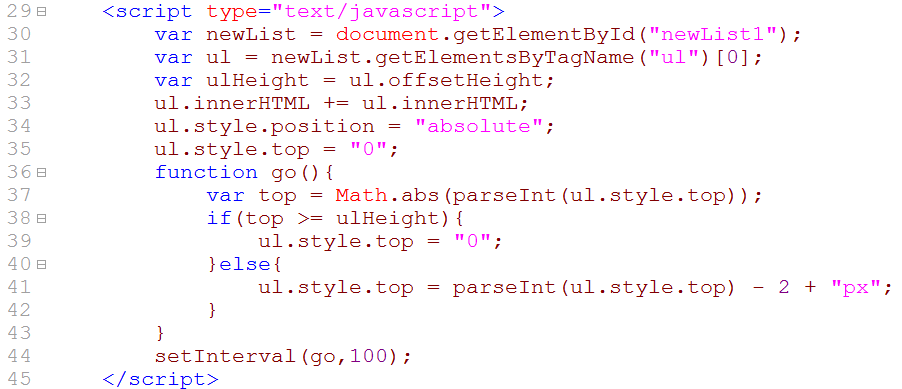
**HTML部分：**

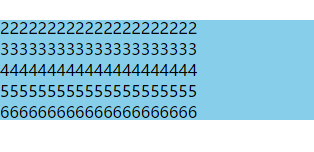


**CSS部分：**

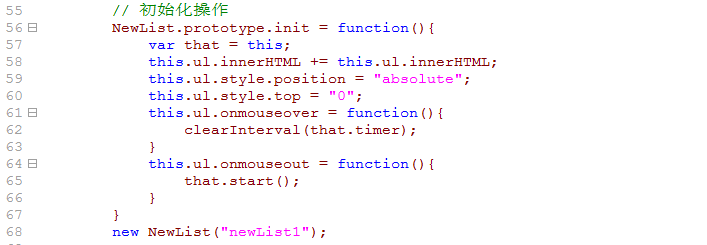


**JS部分**

 **效果：**



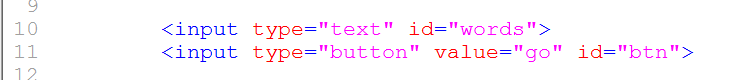
### 2，面向对象方式实现



## 弹幕

### 1，传统的方式实现

**HTML部分：**



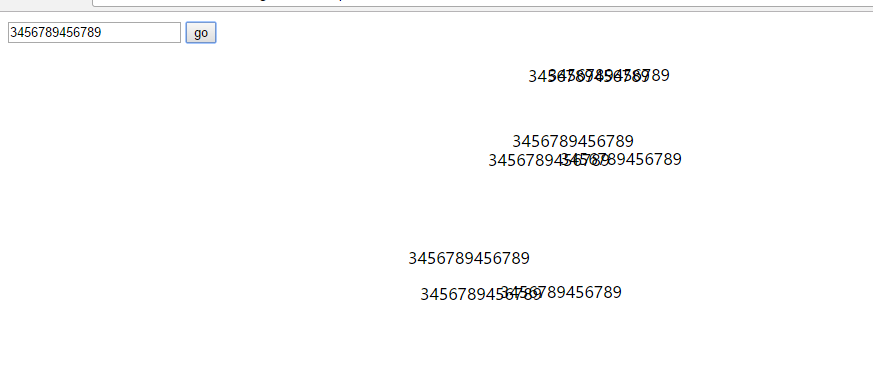
**CSS部分：**

没有样式。

**JS部分：**



**效果：**

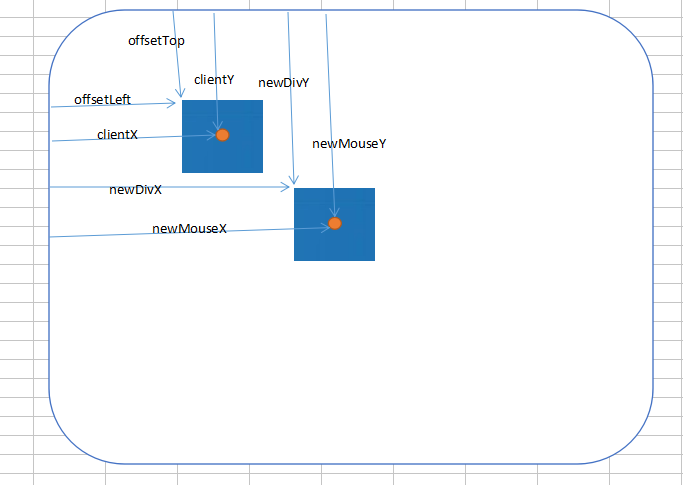


### 2，面向对象的方式实现

**看当日代码**

## 拖拽

原理：



### 1，传统方式实现

**看当日代码**

### 2，面向对象方式实现

**看当日代码**

# 十，继承

**例子：**

1，儿子继承父亲的财产

2，儿女继承父母的基因

继承：不劳而获

## （1），JS中继承的实现思想背景

**以C++以例：**

类（class），实例(对象)， 子类， 父类

面向对象的：封装，继承，多态，抽象

**在JS中：**

是没有类的概念，是通过构造器进行模拟的。 有一个非常奇特的“原型链”。

**从古说起：**

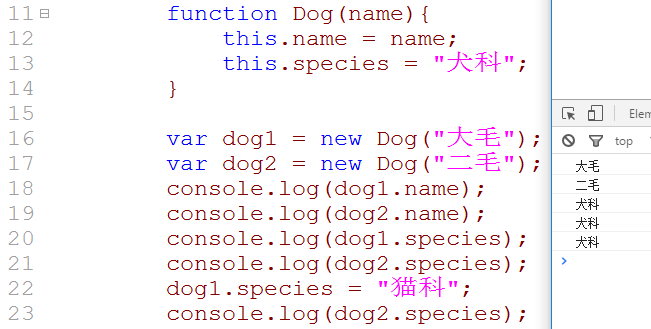
94，网景，发布一款浏览器，上面有一个需要用户填写用户的表单，这个表单只能提交给服务器，让服务进行判断，用户是否填写了这个表单，如果填写了，就返回一个成功填写的信息，如果没有填写，就让用户重写填写。这种模式造成了服务器资源的浪费和时间的浪费。

基于这个需求，网景公司，它急需要一门脚本语言，与浏览器进行互动。Brendan进入到了风景公司，就开始发明这门编程语言，那个时候以**小巧**为目的进行开发。在开发过程中，JAVA诞生了，也是面向对象语言的一个爆发期。Brendan就把JS中所有的数据类型都定义成了对象，也就有了“在JS中，一切皆对象”。他想，我这个JS中用不用继承。最后，结果，JS中不需要继承，JS中都是对象，他想，这么多的对象，我需要将这些对象联系起来，Brendan妥协了，它也引入“继承”的思想。最后它就引入prototype。

Brendan， new运算符，构造函数（构造器）。

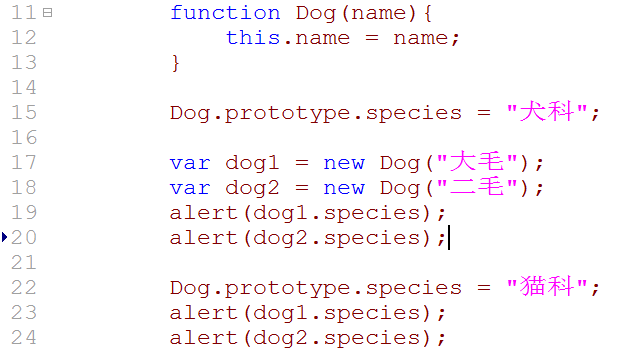
**new运算符的缺点：**

每一个实例对象，都有自己独立的属性和方法，不能做到数据共享，也造成了资源的浪费：



**prototype属性的引入：**

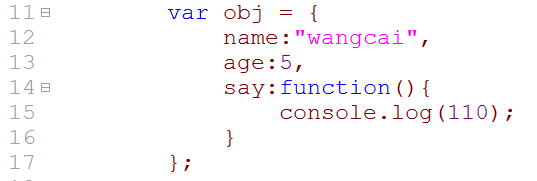
Brendan引入prototype属性。把所有需要共享的属性和方法放到prototype里面，把不需要共享的放到构造器中。



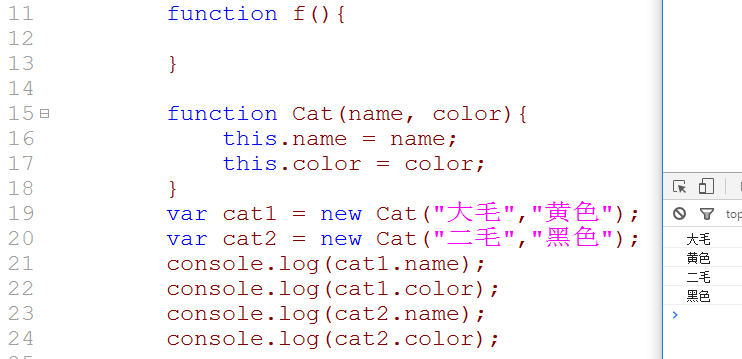
## （2），封装

将属性与方法封装成一个对象，形成一个整体。

### 1，字面量方式的封装



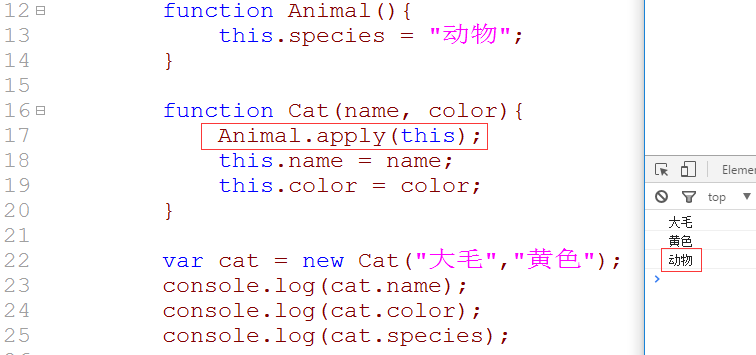
### 2，构造器方式的封装



### 3，原型方式的封装

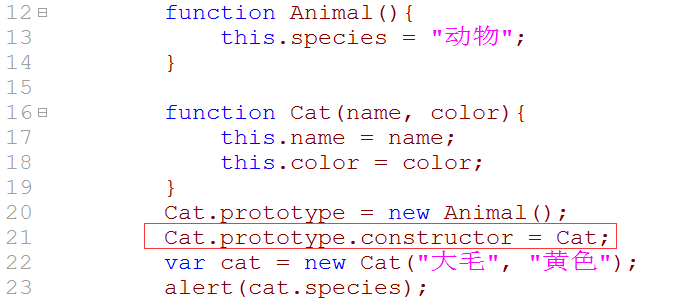
## （3），构造函数（构造器）继承的几种方式？

### 1，构造函数绑定

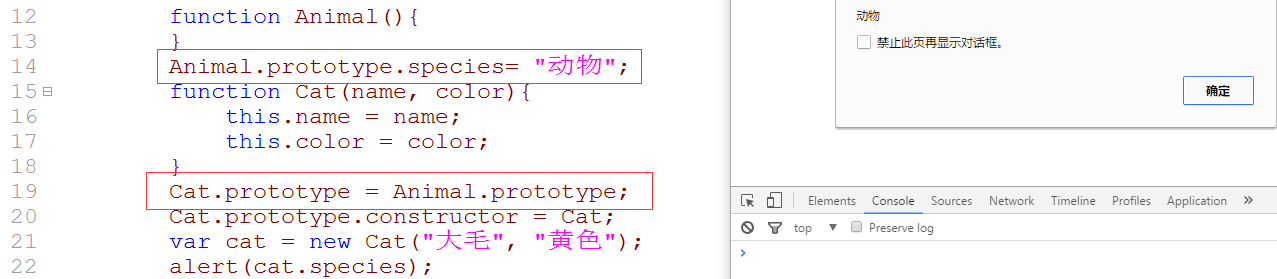


### 2，prototype模式

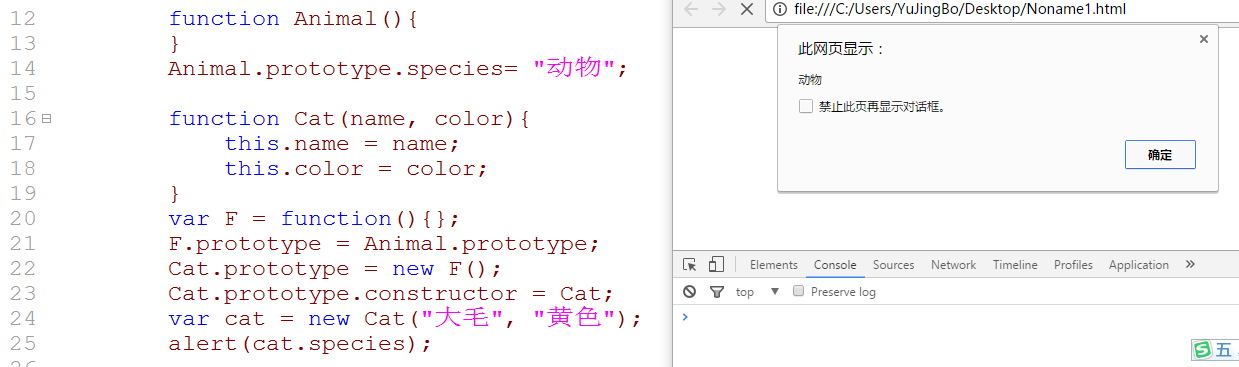
由于上面我们改变了Cat的原型，那么对应的constructor也改变了，我们需要手动改过来：



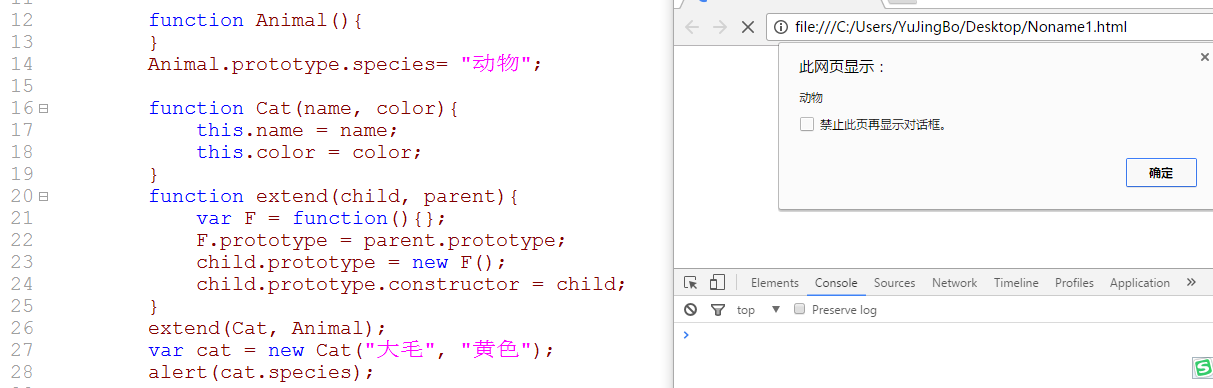
### 3，直接继承prototype



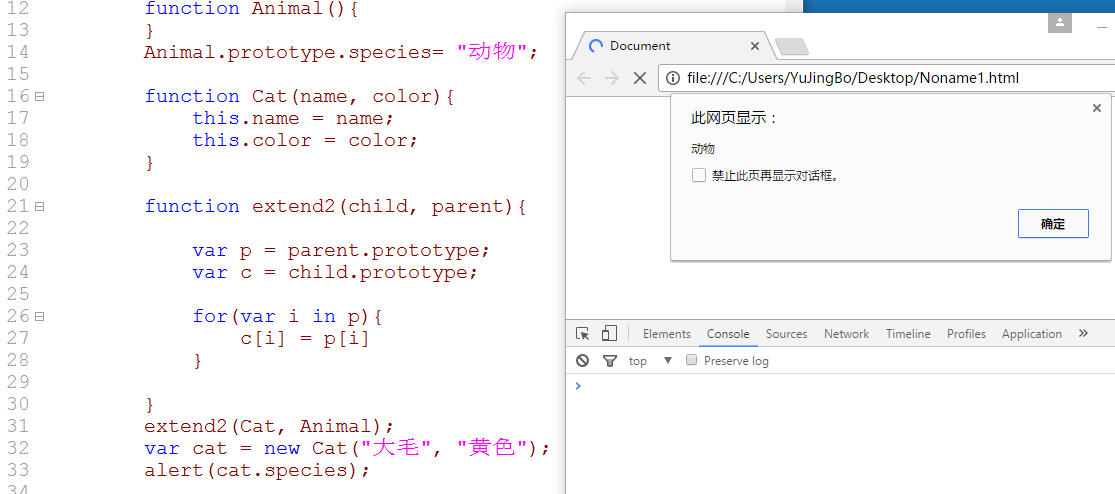
### 4，利用空对象作为中介



能不能封装一个方法：

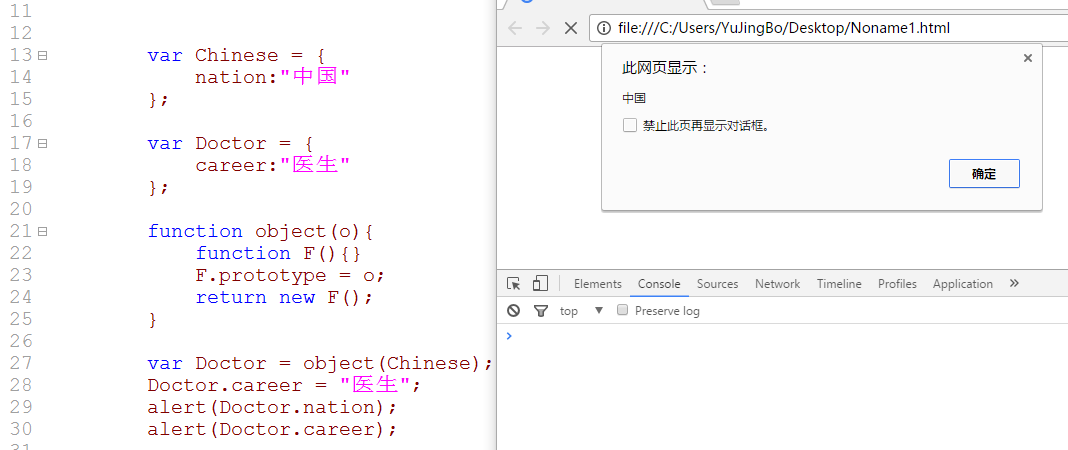


### 5，拷贝继承



## （4），非构造函数继承（字面量创建的对象）

### 1，object()方法



### 2，浅拷贝



### 3，深拷贝

