

# 深入理解javasc ript原型和闭包

看云文档小组



# 目 录

## 说明

- ( 1 ) ——一切都是对象
- ( 2 ) ——函数和对象的关系
- ( 3 ) ——prototype原型
- ( 4 ) ——隐式原型
- ( 5 ) ——instanceof
- ( 6 ) ——继承
- ( 7 ) ——原型的灵活性
- ( 8 ) ——简述【执行上下文】上
- ( 9 ) ——简述【执行上下文】下
- ( 10 ) ——this
- ( 11 ) ——执行上下文栈
- ( 12 ) ——简介【作用域】
- ( 13 ) - 【作用域】和【上下文环境】
- ( 14 ) ——从【自由变量】到【作用域链】
- ( 15 ) ——闭包
- ( 16 ) ——完结
- ( 17 ) ——补this
- ( 18 ) ——补充：上下文环境和作用域的关系

# 说明

---

原文出处：<http://www.cnblogs.com/wangfupeng1988/p/3977924.html>

作者：王福朋

## 说明：

该教程绕开了javascript的一些基本的语法知识，直接讲解javascript中最难理解的两个部分，也是和其他主流面向对象语言区别最大的两个部分——原型和闭包，当然，肯定少不了原型链和作用域链。帮你揭开javascript最神秘的面纱。

为什么要偏偏要讲这两个知识点？

这是我在这么多年学习javascript的经历中，认为最难理解、最常犯错的地方，学习这两个知识点，会让你对javascript有更深层次的理解，至少理解了原型和作用域，就不能再算是javascript菜鸟了。另外，这两方面也是javascript与其他语言不同的地方，学习这样的设计，有助于你开阔眼界，帮助你了解编程语言的设计思路。毕竟，你不能只把眼睛盯在一门语言上。

闲话不多讲，相信奔着这个话题来的朋友，也知道javascript原型和作用域的重要性。

最后说明：被系列教程我不是照搬的其他图书或者网络资料，而是全凭着我对知识的理解而一步一步写的。思路也是我一边写着边想的。有什么不对的地方，欢迎指正。

## ( 1 ) ——一切都是对象

“一切都是对象”这句话的重点在于如何去理解“对象”这个概念。

——当然，也不是所有的都是对象，值类型就不是对象。

首先咱们还是先看看javascript中一个常用的函数——typeof()。typeof应该算是咱们的老朋友，还有谁没用过它？

typeof函数输出的一共有几种类型，在此列出：

```
function show(x) {  
  
    console.log(typeof(x));    // undefined  
    console.log(typeof(10));   // number  
    console.log(typeof('abc')); // string  
    console.log(typeof(true)); // boolean  
  
    console.log(typeof(function () { })); //function  
  
    console.log(typeof([1, 'a', true])); //object  
    console.log(typeof ({ a: 10, b: 20 })); //object  
    console.log(typeof (null)); //object  
    console.log(typeof (new Number(10))); //object  
}  
show();
```

以上代码列出了typeof输出的集中类型标识，其中上面的四种（undefined, number, string, boolean）属于简单的值类型，不是对象。剩下的几种情况——函数、数组、对象、null、new Number(10)都是对象。他们都是引用类型。

判断一个变量是不是对象非常简单。值类型的类型判断用typeof，引用类型的类型判断用instanceof。

```
var fn = function () { };  
console.log(fn instanceof Object); // true
```

好了，上面说了半天对象，各位可能也经常在工作中应对对象，在生活中还得应对活生生的对象。有些个心理不正常或者爱开玩笑的单身人士，还对于系统提示的“找不到对象”耿耿于怀。那么在javascript中的对象，到底该如何定义呢？

对象——若干属性的集合。

java或者C#中的对象都是new一个class出来的，而且里面有字段、属性、方法，规定的非常严格。但是javascript就比较随意了——数组是对象，函数是对象，对象还是对象。对象里面的一切都是属性，只有属性，没有方法。那么这样方法如何表示呢？——方法也是一种属性。因为它的属性表示为键值对的形式

式。

而且，更加好玩的事，javascript中的对象可以任意的扩展属性，没有class的约束。这个大家应该都知道，就不再强调了。

先说个最常见的例子：

```
var obj = {  
  a: 10,  
  b: function (x) {  
    alert(this.a + x);  
  },  
  c: {  
    name: '王福朋',  
    year: 1988  
  }  
};
```

以上代码中，obj是一个自定义的对象，其中a、b、c就是它的属性，而且在c的属性值还是一个对象，它又有name、year两个属性。

这个可能比较好理解，那么函数和数组也可以这样定义属性吗？——当然不行，但是它可以用另一种形式，总之函数/数组之流，只要是对象，它就是属性的集合。

以函数为例子：

```
var fn = function () {  
  alert(100);  
};  
fn.a = 10;  
fn.b = function () {  
  alert(123);  
};  
fn.c = {  
  name: "王福朋",  
  year: 1988  
};
```

上段代码中，函数就作为对象被赋值了a、b、c三个属性——很明显，这就是属性的集合吗。

你问：这个有用吗？

回答：可以看看jQuery源码！

在jQuery源码中，“jQuery”或者“\$”，这个变量其实是一个函数，不信你可以叫咱们的老朋友typeof验证一下。

```
console.log(typeof ($)); // function  
console.log($.trim(" ABC "));
```

验明正身！的确是个函数。那么咱们常用的 \$.trim() 也是个函数，经常用，就不用验了吧！

很明显，这就是在\$或者jQuery函数上加了一个trim属性，属性值是函数，作用是截取前后空格。

javascript与java/C#相比，首先最需要解释的就是弱类型，因为弱类型是最基本的用法，而且最常用，就不打算做一节来讲。

其次要解释的就是本文的内容——一切（引用类型）都是对象，对象是属性的集合。最需要了解的就是对象的概念，和java/C#完全不一样。所以，切记切记！

最后，有个疑问。在typeof的输出类型中，function和object都是对象，为何却要输出两种答案呢？都叫做object不行吗？——当然不行。

具体原因，且听下回分解！

## ( 2 ) ——函数和对象的关系

上文 ( [理解javascript原型和作用域系列 \( 1 \) ——一切都是对象](#) ) 已经提到，函数就是对象的一种，因为通过instanceof函数可以判断。

```
var fn = function () { };  
console.log(fn instanceof Object); // true
```

对！函数是一种对象，但是函数却不像数组一样——你可以说数组是对象的一种，因为数组就像是对象的一个子集一样。但是函数与对象之间，却不仅仅是一种包含和被包含的关系，函数和对象之间的关系比较复杂，甚至有一点鸡生蛋蛋生鸡的逻辑，咱们这一节就缕一缕。

还是先看一个小例子吧。

```
function Fn() {  
    this.name = '王福朋';  
    this.year = 1988;  
}  
var fn1 = new Fn();
```

上面的这个例子很简单，它能说明：对象可以通过函数来创建。对！也只能说明这一点。

但是我要说——对象都是通过函数创建的——有些人可能反驳：不对！因为：

```
var obj = { a: 10, b: 20 };  
var arr = [5, 'x', true];
```

但是不好意思，这个——真的——是一种——“快捷方式”，在编程语言中，一般叫做“语法糖”。

做“语法糖”做的最好的可谓是微软大哥，它把他们家C#那小子弄的不男不女从的，本想图个人见人爱，谁承想还得到处跟人解释——其实它是个男孩！

话归正传——其实以上代码的本质是：

```
//var obj = { a: 10, b: 20 };  
//var arr = [5, 'x', true];  
  
var obj = new Object();  
obj.a = 10;  
obj.b = 20;  
  
var arr = new Array();  
arr[0] = 5;  
arr[1] = 'x';
```

```
arr[2] = true;
```

而其中的 Object 和 Array 都是函数：

```
console.log(typeof (Object)); // function  
console.log(typeof (Array)); // function
```

所以，可以很负责的说——对象都是通过函数来创建的。

现在是不是糊涂了——对象是函数创建的，而函数却又是一种对象——天哪！函数和对象到底是什么关系啊？

别着急！揭开这个谜底，还得先去了解一下另一位老朋友——prototype原型。

本系列文章不打算动辄几千字的长篇大论，咱们小步快跑，不至于看的太乏味。



## ( 3 ) ——prototype原型

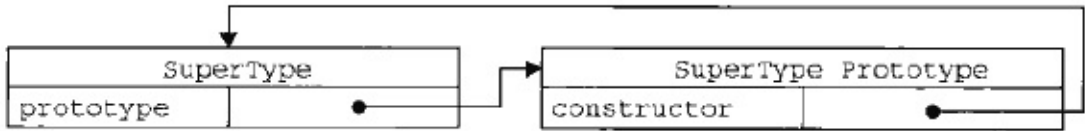
既typeof之后的另一位老朋友！

prototype也是我们的老朋友，即使不了解的人，也应该都听过它的大名。如果它还是您的新朋友，我估计您也是javascript的新朋友。

在咱们的第一节（[深入理解javascript原型和闭包（1）——一切都是对象](#)）中说道，函数也是一种对象。他也是属性的集合，你也可以对函数进行自定义属性。

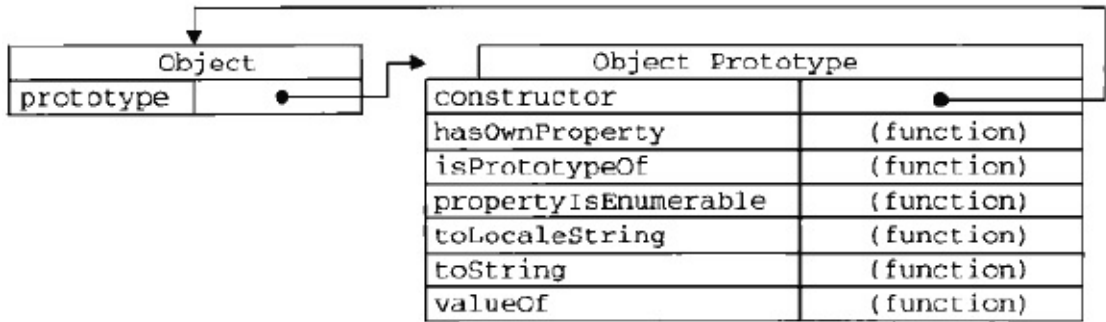
不用等咱们去试验，javascript自己就先做了表率，人家就默认的给函数一个属性——prototype。对，每个函数都有一个属性叫做prototype。

这个prototype的属性值是一个对象（属性的集合，再次强调！），默认的只有一个叫做constructor的属性，指向这个函数本身。



如上图，SuperType是是一个函数，右侧的方框就是它的原型。

原型既然作为对象，属性的集合，不可能就只弄个constructor来玩玩，肯定可以自定义的增加许多属性。例如这位Object大哥，人家的prototype里面，就有好几个其他属性。



咦，有些方法怎么似曾相识？

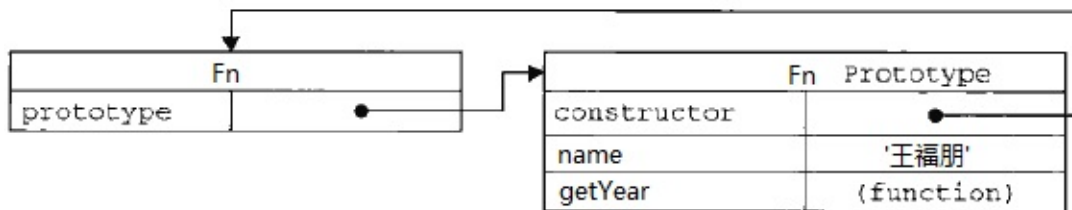
对！别着急，之后会让你知道他们为何似曾相识。

接着往下说，你也可以在自己自定义的方法的prototype中新增自己的属性

```
function Fn() { }
```

```
Fn.prototype.name = '王福朋';
Fn.prototype.getYear = function () {
    return 1988;
};
```

看到没有，这样就变成了



没问题！

但是，这样做有何用呢？ —— 解决这个问题，咱们还是先说说jQuery吧。

```
var $div = $('div');
$div.attr('myName', '王福朋');
```

以上代码中，`$('div')`返回的是一个对象，对象——被函数创建的。假设创建这一对象的函数是`myjQuery`。它其实是这样实现的。

```
myjQuery.prototype.attr = function () {
    //.....
};
$('div') = new myjQuery();
```

不知道大家有没有看明白。

如果用咱们自己的代码来演示，就是这样

```
function Fn() { }
Fn.prototype.name = '王福朋';
Fn.prototype.getYear = function () {
    return 1988;
};

var fn = new Fn();
console.log(fn.name);
console.log(fn.getYear());
```

即，`Fn`是一个函数，`fn`对象是从`Fn`函数`new`出来的，这样`fn`对象就可以调用`Fn.prototype`中的属性。

因为每个对象都有一个隐藏的属性——“`proto`”，这个属性引用了创建这个对象的函数的`prototype`。

即：`fn.proto === Fn.prototype`

这里的"proto"成为“隐式原型”，下回继续分解。

## (4) ——隐式原型

注意：本文不是javascript基础教程，如果你没有接触过原型的基本知识，应该先去了解一下，推荐看《javascript高级程序设计（第三版）》第6章：面向对象的程序设计。

上节已经提到，每个函数function都有一个prototype，即原型。这里再加一句话——每个对象都有一个proto，可成为隐式原型。

这个proto是一个隐藏的属性，javascript不希望开发者用到这个属性值，有的低版本浏览器甚至不支持这个属性值。所以你在Visual Studio 2012这样很高级很智能的编辑器中，都不会有proto的智能提示，但是你不要管它，直接写出来就是了。

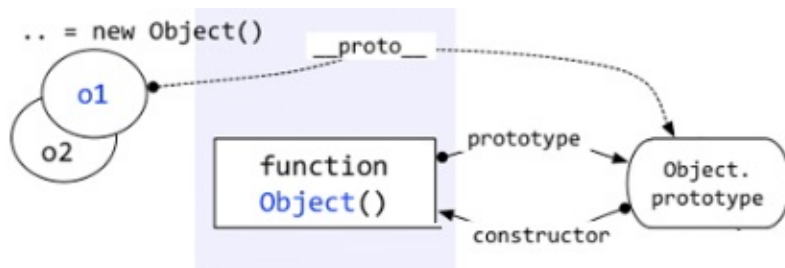
```
var obj = {};  
console.log(obj.__proto__);
```

```
▼ Object ⓘ  
  ▶ __defineGetter__: function __defineGetter__() { [native code] }  
  ▶ __defineSetter__: function __defineSetter__() { [native code] }  
  ▶ __lookupGetter__: function __lookupGetter__() { [native code] }  
  ▶ __lookupSetter__: function __lookupSetter__() { [native code] }  
  ▶ constructor: function Object() { [native code] }  
  ▶ hasOwnProperty: function hasOwnProperty() { [native code] }  
  ▶ isPrototypeOf: function isPrototypeOf() { [native code] }  
  ▶ propertyIsEnumerable: function propertyIsEnumerable() { [native code] }  
  ▶ toLocaleString: function toLocaleString() { [native code] }  
  ▶ toString: function toString() { [native code] }  
  ▶ valueOf: function valueOf() { [native code] }  
  ▶ get __proto__: function __proto__() { [native code] }  
  ▶ set __proto__: function __proto__() { [native code] }
```

上面截图看来，obj.proto和Object.prototype的属性一样！这么巧！

答案就是一样。

obj这个对象本质上是被Object函数创建的，因此obj.proto=== Object.prototype。我们可以用一个图来表示。



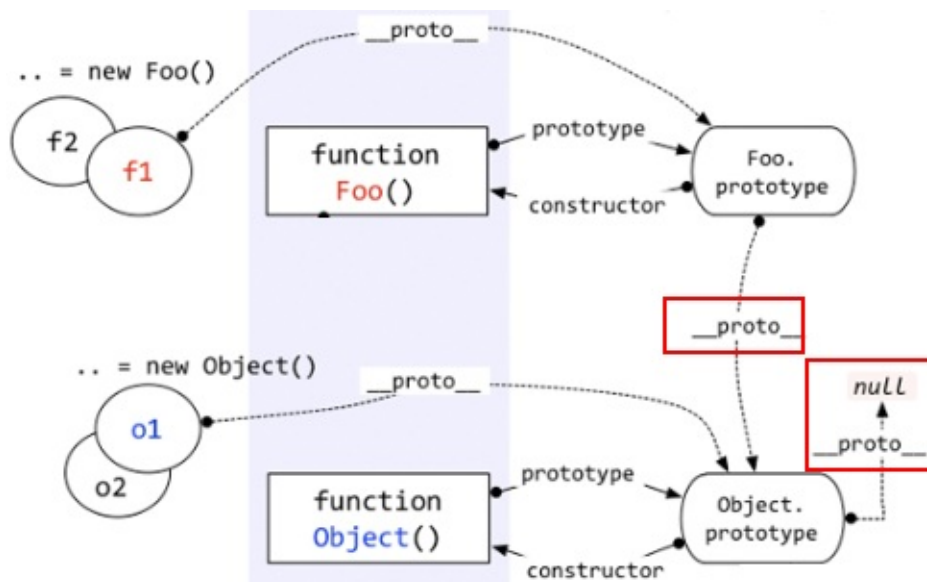
即，每个对象都有一个proto属性，指向创建该对象的函数的prototype。

那么上图中的“Object prototype”也是一个对象，它的proto指向哪里？

好问题！

在说明“Object prototype”之前，先说一下自定义函数的prototype。自定义函数的prototype本质上就是和 `var obj = {}` 是一样的，都是被Object创建，所以它的proto指向的就是Object.prototype。

但是Object.prototype确实一个特例——它的proto指向的是null，切记切记！



还有——函数也是一种对象，函数也有proto吗？

又一个好问题！——当然有。

函数也不是从石头缝里蹦出来的，函数也是被创建出来的。谁创建了函数呢？——Function——注意这个大写的“F”。

且看如下代码。

```
function fn(x, y) {
    return x + y;
};
console.log(fn(10, 20));

var fn1 = new Function("x", "y", "return x + y;");
console.log(fn1(5, 6));
```

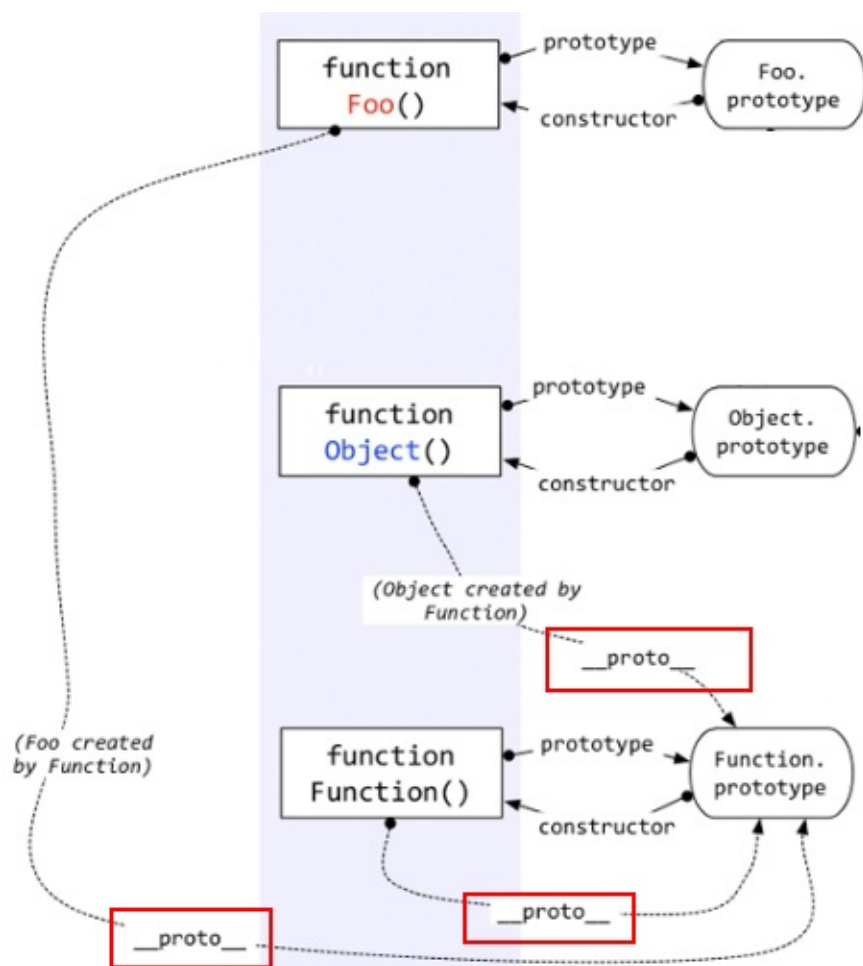
以上代码中，第一种方式是比较传统的函数创建方式，第二种是用new Function创建。

首先根本不推荐用第二种方式。

这里只是向大家演示，函数是被Function创建的。

本文档使用 [看云](#) 构建

好了，根据上面说的一句话——对象的proto指向的是创建它的函数的prototype，就会出现：  
Object.prototype === Function.prototype。用一个图来表示。



上图中，很明显的标出了：自定义函数Foo.prototype指向Function.prototype，Object.prototype指向Function.prototype，唉，怎么还有一个.....Function.prototype指向Function.prototype？这不成了循环引用了？

对！是一个环形结构。

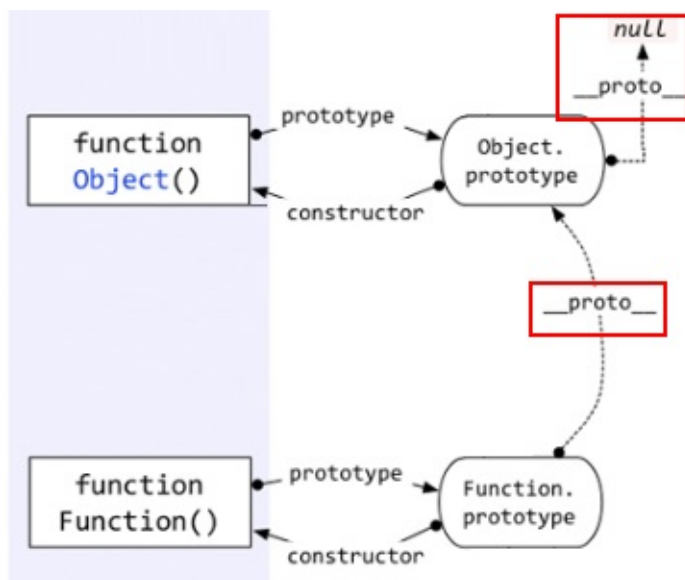
其实稍微想一下就明白了。Function也是一个函数，函数是一种对象，也有proto属性。既然是函数，那么它一定是被Function创建。所以——Function是被自身创建的。所以它的proto指向了自身的Prototype。

篇幅不少了，估计也都看烦了。快结束了。

最后一个问题：Function.prototype指向的对象，它的proto是不是也指向Object.prototype？

答案是肯定的。因为Function.prototype指向的对象也是一个普通的被Object创建的对象，所以也遵循基本的规则。

本文档使用 [看云](#) 构建



OK 本节结束，是不是很乱？

乱很正常。那这一节就让它先乱着，下一节我们将请另一个老朋友来帮忙，把它理清楚。这位老朋友就是——instanceof。

具体内容，请看下节分解。

## ( 5 ) ——instanceof

又介绍一个老朋友——instanceof。

对于值类型，你可以通过typeof判断，string/number/boolean都很清楚，但是typeof在判断到引用类型的时候，返回值只有object/function，你不知道它到底是一个object对象，还是数组，还是new Number等等。

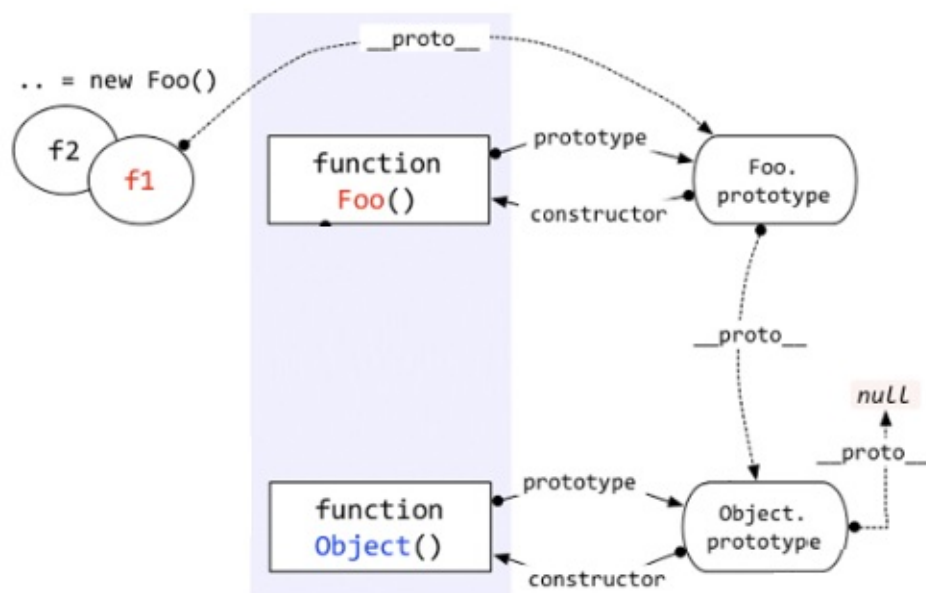
这个时候就需要用到instanceof。例如：

```
function Foo() { }
var f1 = new Foo();

console.log(f1 instanceof Foo); // true
console.log(f1 instanceof Object); // true
```

上图中，f1这个对象是被Foo创建，但是“f1 instanceof Object” 为什么是true呢？

至于为什么过会儿再说，先把instanceof判断的规则告诉大家。根据以上代码看下图：



Instanceof运算符的第一个变量是一个对象，暂时称为A；第二个变量一般是一个函数，暂时称为B。

Instanceof的判断队则是：沿着A的proto这条线来找，同时沿着B的prototype这条线来找，如果两条线能找到同一个引用，即同一个对象，那么就返回true。如果找到终点还未重合，则返回false。

按照以上规则，大家看看“f1 instanceof Object”这句代码是不是true？根据上图很容易就能看出来，



就是true。

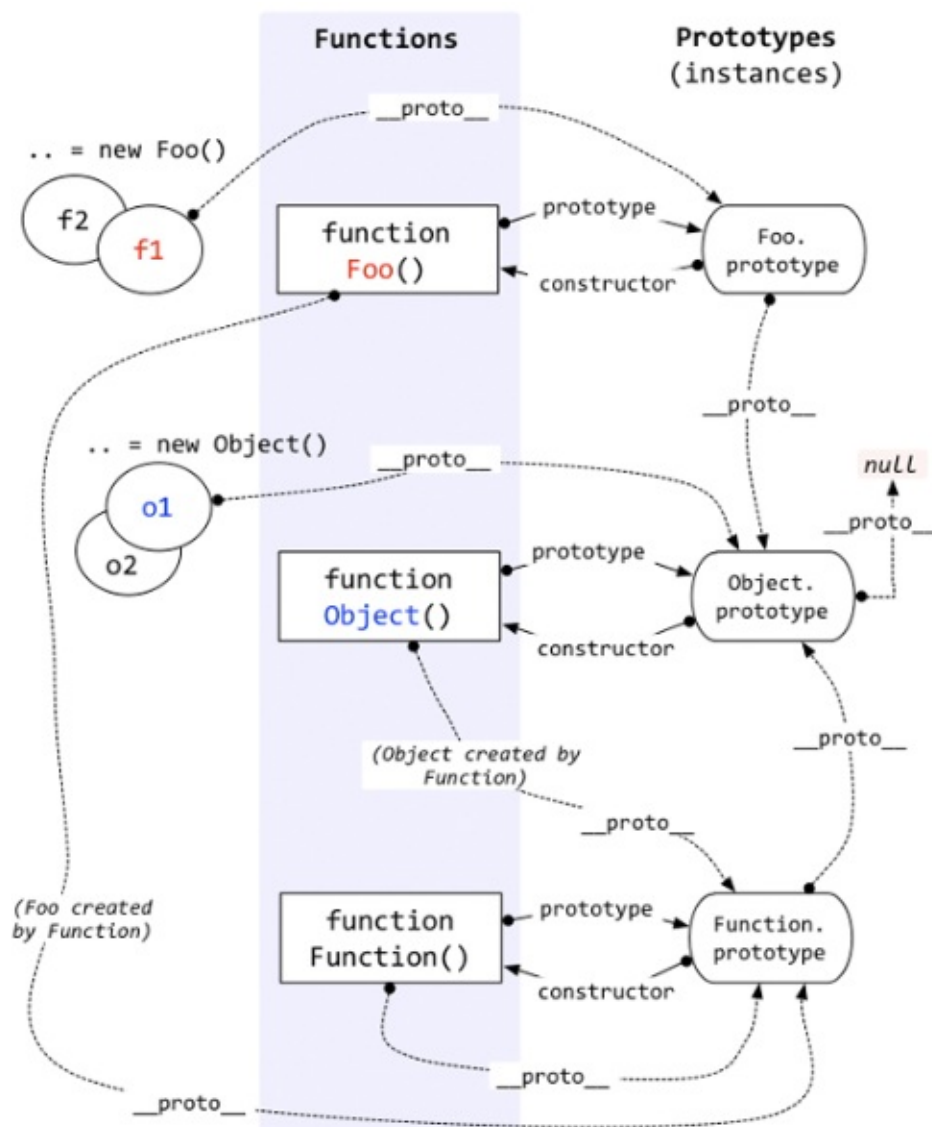
通过上以规则，你可以解释很多比较怪异的现象，例如：

```
console.log(Object instanceof Function); // true
console.log(Function instanceof Object); // true
console.log(Function instanceof Function); // true
```

这些看似很混乱的东西，答案却都是true，这是为何？

正好，这里也接上了咱们上一节说的“乱”。

上一节咱们贴了好多的图片，其实那些图片是可以联合成一个整体的，即：



看这个图片，千万不要嫌烦，必须一条线一条线挨着分析。如果上一节你看的比较仔细，再结合刚才咱们

介绍的instanceof的概念，相信能看懂这个图片的内容。

看看这个图片，你也就知道为何上面三个看似混乱的语句返回的是true了。

问题又出来了。Instanceof这样设计，到底有什么用？到底instanceof想表达什么呢？

重点就这样被这位老朋友给引出来了——继承——原型链。

即，instanceof表示的就是一种继承关系，或者原型链的结构。请看下节分解。

（注：本节的图片来源于

[http://www.ibm.com/developerworks/cn/web/1306\\_jiangjj\\_jsinstanceof/figure1.jpg](http://www.ibm.com/developerworks/cn/web/1306_jiangjj_jsinstanceof/figure1.jpg) )

## (6) ——继承

为何用“继承”为标题，而不用“原型链”？

原型链如果解释清楚了很容易理解，不会与常用的java/C#产生混淆。而“继承”确实常用面向对象语言中最基本的概念，但是java中的继承与javascript中的继承又完全是两回事儿。因此，这里把“继承”着重拿出来，就为了体现这个不同。

javascript中的继承是通过原型链来体现的。先看几句代码

```
function Foo() { }
var f1 = new Foo();

f1.a = 10;

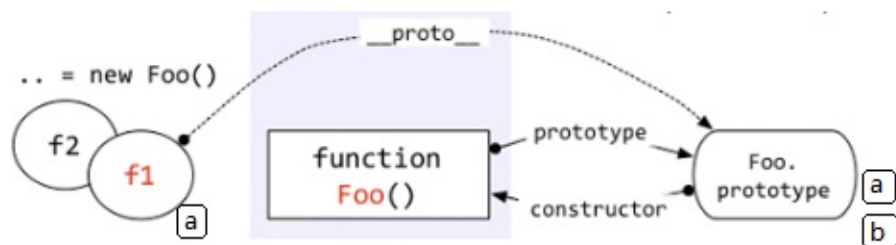
Foo.prototype.a = 100;
Foo.prototype.b = 200;

console.log(f1.a); //10
console.log(f1.b); //200
```

以上代码中，f1是Foo函数new出来的对象，f1.a是f1对象的基本属性，f1.b是怎么来的呢？——从Foo.prototype得来，因为f1.proto指向的是Foo.prototype

访问一个对象的属性时，先在基本属性中查找，如果没有，再沿着proto这条链向上找，这就是原型链。

看图说话：



上图中，访问f1.b时，f1的基本属性中没有b，于是沿着proto找到了Foo.prototype.b。

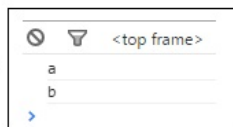
那么我们在实际应用中如何区分一个属性到底是基本的还是从原型中找到的呢？大家可能都知道答案了——hasOwnProperty，特别是在for...in...循环中，一定要注意。

```
function Foo() {}
var f1 = new Foo();

f1.a = 10;

Foo.prototype.a = 100;
Foo.prototype.b = 200;

var item;
for (item in f1) {
  console.log(item);
}
```

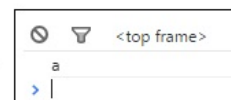


```
function Foo() {}
var f1 = new Foo();

f1.a = 10;

Foo.prototype.a = 100;
Foo.prototype.b = 200;

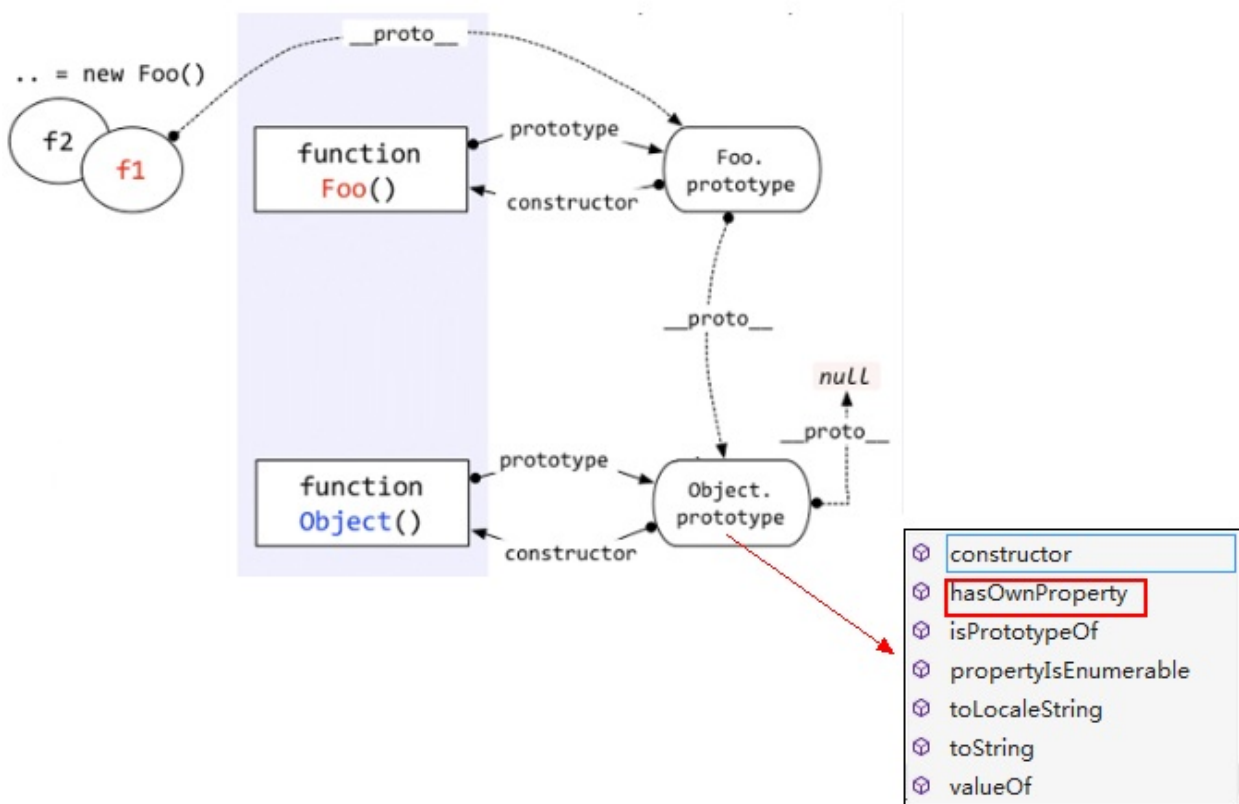
var item;
for (item in f1) {
  if (f1.hasOwnProperty(item)) {
    console.log(item);
  }
}
```



等等，不对！f1的这个hasOwnProperty方法是从哪里来的？f1本身没有，Foo.prototype中也没有，哪儿来的？

好问题。

它是从Object.prototype中来的，请看图：



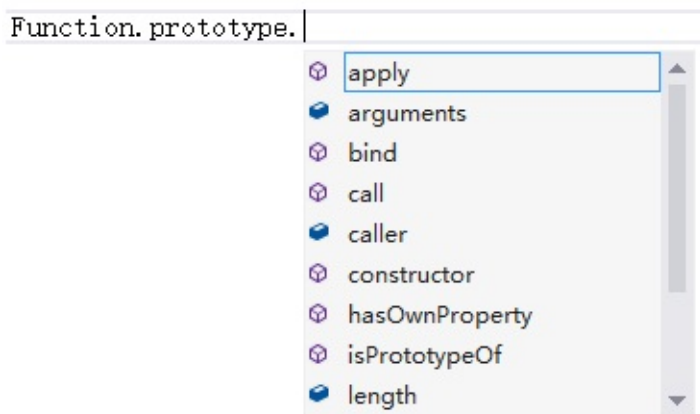
对象的原型链是沿着proto这条线走的，因此在查找f1.hasOwnProperty属性时，就会顺着原型链一直查找到Object.prototype。

由于所有的对象的原型链都会找到Object.prototype，因此所有的对象都会有Object.prototype的方法。这就是所谓的“继承”。

当然这只是一个例子，你可以自定义函数和对象来实现自己的继承。

说一个函数的例子吧。

我们都知道每个函数都有call，apply方法，都有length，arguments，caller等属性。为什么每个函数都有？这肯定是“继承”的。函数由Function函数创建，因此继承的Function.prototype中的方法。不信可以请微软的Visual Studio老师给我们验证一下：



看到了吧，有call、length等这些属性。

那怎么还有hasOwnProperty呢？——那是Function.prototype继承自Object.prototype的方法。有疑问可以看看上一节将instanceof时候那个大图，看看Function.prototype.prototype是否指向Object.prototype。

原型、原型链，大家都明白了吗？

## (7) ——原型的灵活性

在Java和C#中，你可以简单的理解class是一个模子，对象就是被这个模子压出来的一批一批月饼（中秋节刚过完）。压个啥样，就得是个啥样，不能随便动，动一动就坏了。

而在javascript中，就没有模子了，月饼被换成了面团，你可以捏成自己想要的样子。

首先，对象属性可以随时改动。

对象或者函数，刚开始new出来之后，可能啥属性都没有。但是你可以这会儿加一个，过一会儿再加两个，非常灵活。

在jQuery的源码中，对象被创建时什么属性都没有，都是代码一步一步执行时，一个一个加上的。

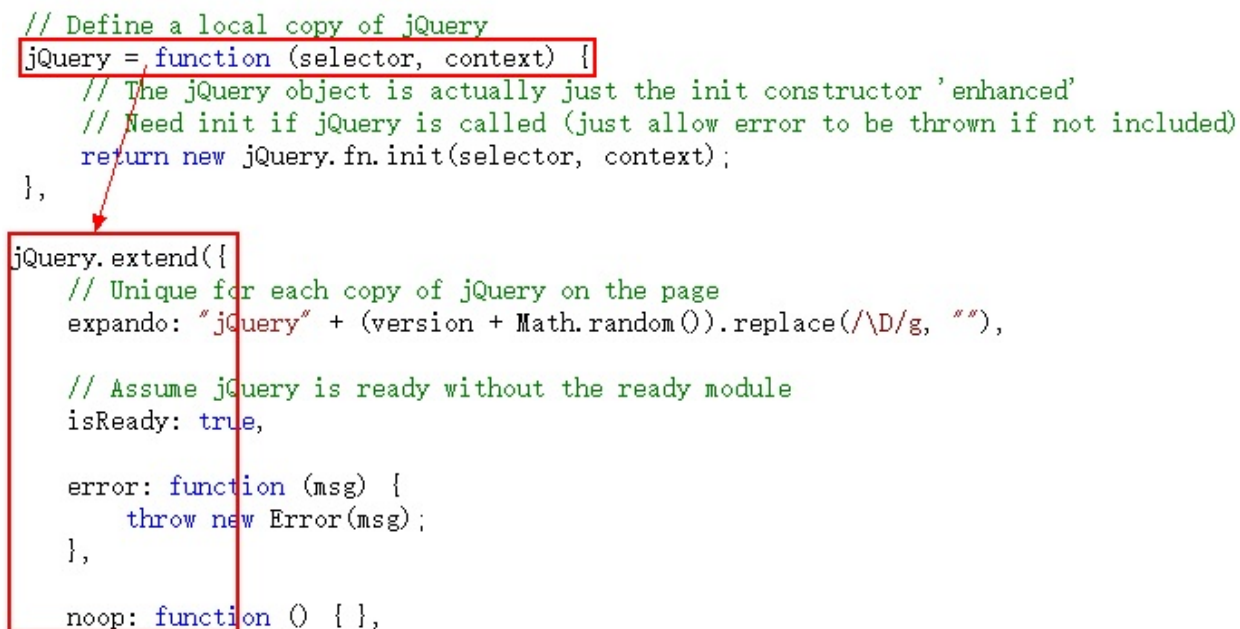
```
// Define a local copy of jQuery
jQuery = function (selector, context) {
    // The jQuery object is actually just the init constructor 'enhanced'
    // Need init if jQuery is called (just allow error to be thrown if not included)
    return new jQuery.fn.init(selector, context);
},

jQuery.extend({
    // Unique for each copy of jQuery on the page
    expando: "jQuery" + (version + Math.random()).replace(/\D/g, ""),

    // Assume jQuery is ready without the ready module
    isReady: true,

    error: function (msg) {
        throw new Error(msg);
    },

    noop: function () {}
});
```



其次，如果继承的方法不合适，可以做出修改。

```
var obj = { a: 10, b: 20 };
console.log(obj.toString()); //[object Object]

var arr = [1, 2, true];
console.log(arr.toString()); //1,2,true
```

如上图，Object和Array的toString()方法不一样。肯定是Array.prototype.toString()方法做了修改。

同理，我也可以自定义一个函数，并自己去修改prototype.toString()方法。

```
function Foo() {}
var f1 = new Foo();

Foo.prototype.toString = function () {
    return '王福朋';
};
console.log(f1.toString()); //王福朋
```

最后，如果感觉当前缺少你要用的方法，可以自己去创建。

例如在json2.js源码中，为Date、String、Number、Boolean方法添加一个toJSON的属性。

```
if (typeof Date.prototype.toJSON !== 'function') {

    Date.prototype.toJSON = function (key) {

        return isFinite(this.valueOf()) ?
            this.getUTCFullYear() + '-' +
            f(this.getUTCMonth() + 1) + '-' +
            f(this.getUTCDate()) + 'T' +
            f(this.getUTCHours()) + ':' +
            f(this.getUTCMinutes()) + ':' +
            f(this.getUTCSeconds()) + 'Z' : null;
    };

    String.prototype.toJSON =
    Number.prototype.toJSON =
    Boolean.prototype.toJSON = function (key) {
        return this.valueOf();
    };
}
```

如果你要添加内置方法的原型属性，最好做一步判断，如果该属性不存在，则添加。如果本来就存在，就没必要再添加了。

## (8) ——简述【执行上下文】上

什么是“执行上下文”（也叫做“执行上下文环境”）？暂且不下定义，先看一段代码：

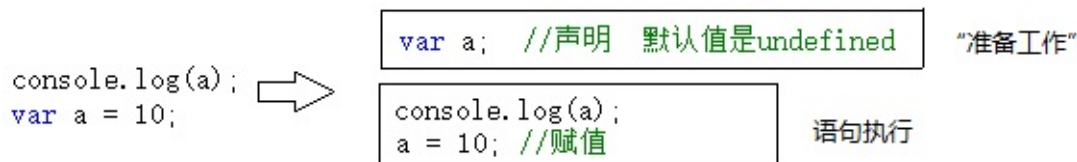
```
console.log(a); ➡ ▶ Uncaught ReferenceError: a is not defined
```

```
console.log(a); ➡ undefined  
var a;
```

```
console.log(a); ➡ undefined  
var a = 10;
```

第一句报错，a未定义，很正常。第二句、第三句输出都是undefined，说明浏览器在执行console.log(a)时，已经知道了a是undefined，但却不知道a是10（第三句中）。

在一段js代码拿过来真正一句一句运行之前，浏览器已经做了一些“准备工作”，其中就包括对变量的声明，而不是赋值。变量赋值是在赋值语句执行的时候进行的。可用下图模拟：



这是第一种情况。

下面还有。先来个简单的。

```
console.log(this); ➡ ▶ Window {top: Window, window: Wind
```

有js开发经验的朋友应该都知道，你无论在哪个位置获取this，都是有值的。至于this的取值情况，比较复杂，会专门拿出一篇文章来讲解。

与第一种情况不同的是：第一种情况只是对变量进行声明（并没有赋值），而此种情况直接给this赋值。这也是“准备工作”情况要做的事情之一。

下面还有。。。第三种情况。

在第三种情况中，需要注意代码注释中的两个名词——“函数表达式”和“函数声明”。虽然两者都很常用，但是这两者在“准备工作”时，却是两种待遇。



```
console.log(f1);    // function f1() {}  
function f1() {}    // 函数声明  
  
console.log(f2);    // undefined  
var f2 = function () {}; // 函数表达式
```

看以上代码。“函数声明”时我们看到了第二种情况的影子，而“函数表达式”时我们看到了第一种情况的影子。

没错。在“准备工作”中，对待函数表达式就像对待“var a = 10”这样的变量一样，只是声明。而对待函数声明时，却把函数整个赋值了。

好了，“准备工作”介绍完毕。

我们总结一下，在“准备工作”中完成了哪些工作：

- 变量、函数表达式——变量声明，默认赋值为undefined；
- this——赋值；
- 函数声明——赋值；

这三种数据的准备情况我们称之为“执行上下文”或者“执行上下文环境”。

这里插一句题外话：通过以上三种情况，你可能会联想到网上的有些考js语法的题目/面试题。的确，几乎每个js语法题中都有这种题目出现。之前你遇到这种题目是不是靠背诵来解决？背过了，隔几天又忘记了。——任何问题，都要去追根溯源，要知道这个问题是真正出自哪一块知识点，要真正去理解。光靠背诵是没用的。

细心的朋友可能会发现，我们上面所有的例子都是在全局环境下执行的。

其实，javascript在执行一个代码段之前，都会进行这些“准备工作”来生成执行上下文。这个“代码段”其实分三种情况——全局代码，函数体，eval代码。

这里解释一下为什么代码段分为这三种。

所谓“代码段”就是一段文本形式的代码。

首先，全局代码是一种，这个应该没有非议，本来就是手写文本到标签里面的。


```
<script type="text/javascript">  
    //代码段……  
</script>
```

其次，eval代码接收的也是一段文本形式的代码。

```
eval("alert(123)");
```

最后，函数体是代码段是因为函数在创建时，本质上是 `new Function(...)` 得来的，其中需要传入一个文本形式的参数作为函数体。

```
function fn(x) {  
    console.log(x + 5);  
}
```



```
var fn = new Function("x", "console.log(x + 5)");
```

这样解释应该能理解了。

最后，`eval`不常用，也不推荐大家用。

下一节我们介绍函数的情况，并一起总结一下执行上下文到底包含哪些内容。敬请期待。

## ( 9 ) ——简述【执行上下文】下

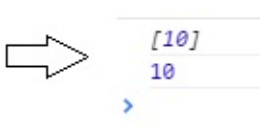
继续上一篇文章 ( <http://www.cnblogs.com/wangfupeng1988/p/3986420.html> ) 的内容。

上一篇我们讲到在全局环境下的代码段中，执行上下文环境中有何数据：

- 变量、函数表达式——变量声明，默认赋值为undefined；
- this——赋值；
- 函数声明——赋值；

如果在函数中，除了以上数据之外，还会有其他数据。先看以下代码：

```
function fn(x) {  
    console.log(arguments);  
    console.log(x);  
}  
fn(10);
```



The diagram illustrates the execution context for the function `fn` when called with the argument `10`. It shows a right-pointing arrow from the function call `fn(10);` to a representation of the execution context. This context contains two entries: `arguments` with the value `[10]` and `x` with the value `10`.

以上代码展示了在函数体的语句执行之前，`arguments`变量和函数的参数都已经被赋值。从这里可以看出，函数每被调用一次，都会产生一个新的执行上下文环境。因为不同的调用可能就会有不同的参数。

另外一点不同在于，函数在定义的时候（不是调用的时候），就已经确定了函数体内部自由变量的作用域。至于“自由变量”和“作用域”是后面要专门拿出来讲述的重点，这里就先点到为止。用一个例子说明一下：

```
var a = 10;  
function fn() {  
    console.log(a); //a是自由变量  
} //函数创建时，就确定了a要取值的作用域  
  
function bar(f) {  
    var a = 20;  
    f(); // 打印“10”，而不是“20”  
}  
bar(fn);
```

好了，总结完了函数的附加内容，我们就此要全面总结一下上下文环境的数据内容。

全局代码的上下文环境数据内容为：

普通变量（包括函数表达式），如： <code>var a = 10;</code>	声明（默认赋值为undefined）
函数声明，如： <code>function fn() {}</code>	赋值
this	赋值

如果代码段是函数体，那么在此基础上需要附加：

|||

|---|---|

| 参数 | 赋值 |

| arguments | 赋值 |

| 自由变量的取值作用域 | 赋值 |

给执行上下文环境下一个通俗的定义——在执行代码之前，把将要用到的所有的变量都事先拿出来，有的直接赋值了，有的先用undefined占个空。

了解了执行上下文环境中的数据信息，你就不用再去死记硬背那些可恶的面试题了。理解了就不用背诵！

讲完了上下文环境，又来了新的问题——在执行js代码时，会有数不清的函数调用次数，会产生许多个上下文环境。这么多上下文环境该如何管理，以及如何销毁而释放内存呢？下一节将通过“执行上下文栈”来解释这个问题。

不过别着急，在解释“执行上下文栈”之前，还需要把this说一下，this还是挺重要的。

说完this，接着说执行上下文栈。

## ( 10 ) ——this

接着上一节讲的话，应该轮到“执行上下文栈”了，但是这里不得不插入一节，把this说一下。因为this很重要，js的面试题如果不出几个与this有关的，那出题者都不合格。

其实，this的取值，分四种情况。我们来挨个看一下。

在此再强调一遍一个非常重要的知识点：在函数中this到底取何值，是在函数真正被调用执行的时候确定的，函数定义的时候确定不了。因为this的取值是执行上下文环境的一部分，每次调用函数，都会产生一个新的执行上下文环境。

情况1\*\*：构造函数\*\*

所谓构造函数就是用来new对象的函数。其实严格来说，所有的函数都可以new一个对象，但是有些函数的定义是为了new一个对象，而有些函数则不是。另外注意，构造函数的函数名第一个字母大写（规则约定）。例如：Object、Array、Function等。

```
function Foo() {
    this.name = '王福朋';
    this.year = 1988;

    console.log(this); // Foo {name: "王福朋", year: 1988}
}

var f1 = new Foo();

console.log(f1.name); // 王福朋
console.log(f1.year); // 1988
```

以上代码中，如果函数作为构造函数用，那么其中的this就代表它即将new出来的对象。

注意，以上仅限new Foo()的情况，即Foo函数作为构造函数的情况。如果直接调用Foo函数，而不是new Foo()，情况就大不一样了。

```
function Foo() {
    this.name = '王福朋';
    this.year = 1988;

    console.log(this); // Window {top: Window, window: Window, location: Location, exte
}

Foo();
```

这种情况下this是window，我们后文中会说到。

## 情况2\*\*：函数作为对象的一个属性\*\*

如果函数作为对象的一个属性时，并且作为对象的一个属性被调用时，函数中的this指向该对象。

```
var obj = {
  x: 10,
  fn: function () {
    console.log(this);    // Object {x: 10, fn: function}
    console.log(this.x);  // 10
  }
};

obj.fn();
```

以上代码中，fn不仅作为一个对象的一个属性，而且的确是作为对象的一个属性被调用。结果this就是obj对象。

注意，如果fn函数不作为obj的一个属性被调用，会是什么结果呢？

```
var obj = {
  x: 10,
  fn: function () {
    console.log(this);    // Window {top: Window, window: Wi
    console.log(this.x);  // undefined
  }
};

var fn1 = obj.fn;
fn1();
```

如上代码，如果fn函数被赋值到了另一个变量中，并没有作为obj的一个属性被调用，那么this的值就是window，this.x为undefined。

## 情况3\*\*：函数用call或者apply调用\*\*

当一个函数被call和apply调用时，this的值就取传入的对象的值。至于call和apply如何使用，不会的朋友可以去查查其他资料，本系列教程不做讲解。

```
var obj = {
  x: 10
};

var fn = function () {
  console.log(this);    //Object {x: 10}
  console.log(this.x);  //10
}
fn.call(obj);
```

## 情况4\*\*：全局 & 调用普通函数\*\*

本文档使用 [看云](#) 构建

在全局环境下，this永远是window，这个应该没有非议。

```
console.log(this === window); // true
```

普通函数在调用时，其中的this也都是window。

```
var x = 10;

var fn = function () {
    console.log(this);    //Window {top: Window, window: Wi
    console.log(this.x);  //10
}
fn();
```

以上代码很好理解。

不过下面的情况你需要注意一下：

```
var obj = {
    x: 10,
    fn: function () {

        function f() {
            console.log(this);    //Window {top: Window, window: Wind
            console.log(this.x);  //undefined
        }
        f();

    }
};
obj.fn();
```

函数f虽然是在obj.fn内部定义的，但是它仍然是一个普通的函数，this仍然指向window。

完了。

看到了吧，this有关的知识点还是挺多的，不仅多而且非常重要。

最后，既然提到了this，有必要把一个非常经典的案例介绍给大家，又是jQuery源码的。

```
jQuery.extend = jQuery.fn.extend = function () {  
    // ...此处省略若干行...  
  
    // extend jQuery itself if only one argument is passed  
    if (i === length) {  
        target = this;  
        i--;  
    }  
  
    // ...此处省略若干行...  
};
```

以上代码是从jQuery中摘除来的部分代码。jQuery.extend和jQuery.fn.extend都指向了同一个函数，但是当执行时，函数中的this是不一样的。

执行jQuery.extend(...)时，this指向jQuery；执行jQuery.fn.extend(...)时，this指向jQuery.fn。

这样就巧妙的将一段代码同时共享给两个功能使用，更加符合设计原则。

好了，聊完了this。接着上一节继续说“执行上下文栈”。

注意：还有一部分this的内容本文中没有讲到，已经补充到这里：<http://www.cnblogs.com/wangfupeng1988/p/3996037.html>

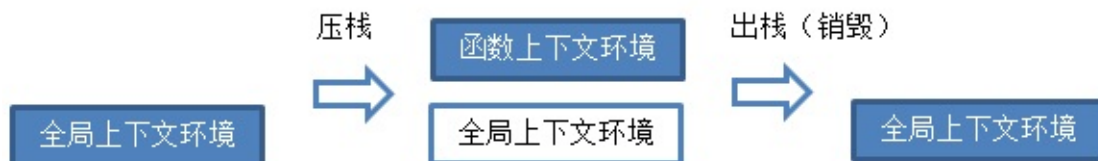


## ( 11 ) —— 执行上下文栈

继续上文的内容。

执行全局代码时，会产生一个执行上下文环境，每次调用函数都又会产生执行上下文环境。当函数调用完成时，这个上下文环境以及其中的数据都会被消除，再重新回到全局上下文环境。处于活动状态的执行上下文环境只有一个。

其实这是一个压栈出栈的过程——执行上下文栈。如下图：



可根据以下代码来详细介绍上下文栈的压栈、出栈过程。

```

1  var a = 10,                                // 1. 进入全局上下文环境
2      fn,
3  bar = function (x) {
4      var b = 5;
5      fn(x + b);                               // 3. 进入fn函数上下文环境
6  };
7
8  fn = function (y) {
9      var c = 5;
10     console.log(y + c);
11 }
12
13 bar(10);                                   // 2. 进入bar函数上下文环境
  
```

如上代码。

在执行代码之前，首先将创建全局上下文环境。

全局上下文环境	
a	undefined
fn	undefined
bar	undefined
this	window

然后是代码执行。代码执行到第12行之前，上下文环境中的变量都在执行过程中被赋值。

全局上下文环境	
a	10
fn	function
bar	function
this	window

执行到第13行，调用bar函数。

跳转到bar函数内部，执行函数体语句之前，会创建一个新的执行上下文环境。

bar函数-执行上下文环境	
b	undefined
x	10
arguments	[10]
this	window

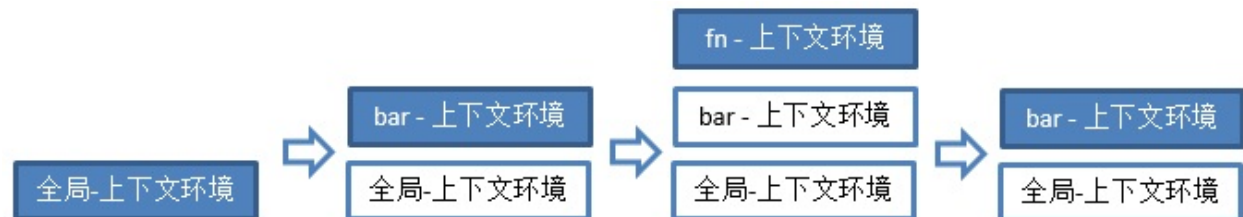
并将这个执行上下文环境压栈，设置为活动状态。



执行到第5行，又调用了fn函数。进入fn函数，在执行函数体语句之前，会创建fn函数的执行上下文环境，并压栈，设置为活动状态。



待第5行执行完毕，即fn函数执行完毕后，此次调用fn所生成的上下文环境出栈，并且被销毁（已经用完了，就要及时销毁，释放内存）。



同理，待第13行执行完毕，即bar函数执行完毕后，调用bar函数所生成的上下文环境出栈，并且被销毁（已经用完了，就要及时销毁，释放内存）。



好了，我很耐心的给大家介绍了一段简短代码的执行上下文环境的变化过程，一个完整的闭环。其中上下文环境的变量赋值过程我省略了许多，因为那些并不难，一看就知道。

讲到这里，我不得不很遗憾的跟大家说：其实以上我们所演示的是一种比较理想的情况。有一种情况，而且是很常用的一种情况，无法做到这样干净利落的说销毁就销毁。这种情况就是伟大的——闭包。

要说闭包，咱们还得先从自由变量和作用域说起。

## ( 12 ) ——简介【作用域】

---

提到作用域，有一句话大家（有js开发经验者）可能比较熟悉：“javascript没有块级作用域”。所谓“块”，就是大括号“{ }”中间的语句。例如if语句：

```
1  var i = 10;
2  if (i > 1) {
3      var name = '王福朋';
4  }
5  console.log(name); //王福朋
```

再比如for语句：

```
1  for (var i = 0; i < 10; i++) {
2      //....
3  }
4  console.log(i); //10
```

所以，我们在编写代码的时候，不要在“块”里面声明变量，要在代码的一开始就声明好了。以避免发生歧义。如：

```
1  var i;
2  for (i = 0; i < 10; i++) {
3      //....
4  }
5  console.log(i); //10
```

其实，你光知道“javascript没有块级作用域”是完全不够的，你需要知道的是——javascript\*\*除了全局作用域之外，只有函数可以创建的作用域\*\*。

所以，我们在声明变量时，全局代码要在代码前端声明，函数中要在函数体一开始就声明好。除了这两个地方，其他地方都不要出现变量声明。而且建议用“单var”形式。

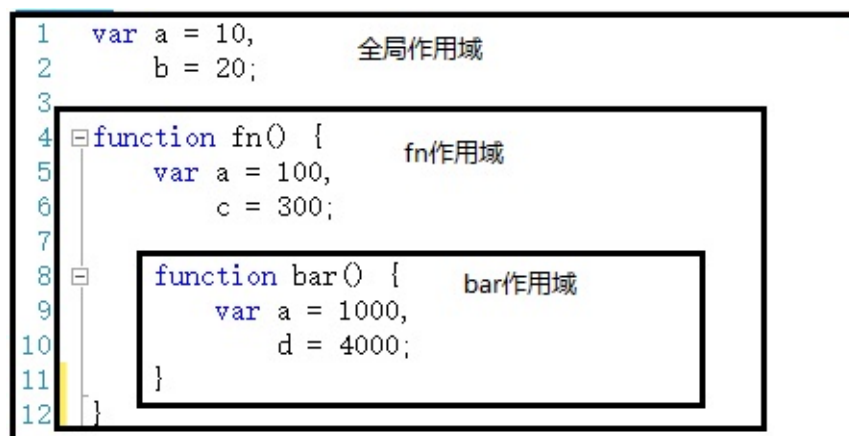
jQuery就是一个很好的示例：

```

66     var
67         // Use the correct document accordingly with window argument (sandbox)
68         document = window.document,
69
70         version = "2.1.1",
71
72         // Define a local copy of jQuery
73     jQuery = function (selector, context) {
74         // The jQuery object is actually just the init constructor 'enhanced'
75         // Need init if jQuery is called (just allow error to be thrown if not included)
76         return new jQuery.fn.init(selector, context);
77     },
78
79     // Support: Android<4.1
80     // Make sure we trim BOM and NBSP
81     rtrim = /^[\s\uFEFF\xA0]+|[\s\uFEFF\xA0]+$/g,
82
83     // Matches dashed string for camelizing
84     rmsPrefix = /^-ms-/,
85     rdashAlpha = /-([a-z])/gi,
86
87     // Used by jQuery.camelCase as callback to replace()
88     fcamelCase = function (all, letter) {
89         return letter.toUpperCase();
90     };

```

下面继续说作用域。作用域是一个很抽象的概念，类似于一个“地盘”



如上图，全局代码和fn、bar两个函数都会形成一个作用域。而且，作用域有上下级的关系，上下级关系的确定就看函数是在哪个作用域下创建的。例如，fn作用域下创建了bar函数，那么“fn作用域”就是“bar作用域”的上级。

作用域最大的用处就是隔离变量，不同作用域下同名变量不会有冲突。例如以上代码中，三个作用域下都声明了“a”这个变量，但是他们不会有冲突。各自的作用域下，用各自的“a”。

说到这里，咱们又可以拿出jquery源码来讲讲了。

jQuery源码的最外层是一个自动执行的匿名函数：

```
1  (function(window, undefined) {  
2  
3      var a;  
4      var b;  
5      var c;  
6  
7      //...  
8  
9  }(window));
```

为什么要这样做呢？

原因就是jQuery源码中，声明了大量的变量，这些变量将通过一个函数被限制在一个独立的作用域中，而不会与全局作用域或者其他函数作用域的同名变量产生冲突。

全世界的开发者都在用jQuery，如果不这样做，很可能导致jQuery源码中的变量与外部javascript代码中的变量重名，从而产生冲突。

作用域这块只是很不好解释，咱们就小步快跑，一步一步慢慢展示给大家。

下一节将把作用域和执行上下文环境结合来说一说。

可见，要理解闭包，不是一两句话能说清楚的。。。

## ( 13 ) - 【作用域】和【上下文环境】

上文简单介绍了作用域，本文把作用域和上下文环境结合来说一下，会理解的更深一些。

```
1  var a = 10, 全局作用域
2      b = 20;
3
4  function fn(x) {
5      var a = 100, fn作用域
6          c = 300;
7
8      function bar(x) {
9          var a = 1000,
10              d = 4000;
11      }
12      bar(100);
13      bar(200);
14  }
15
16
17  fn(10);
```

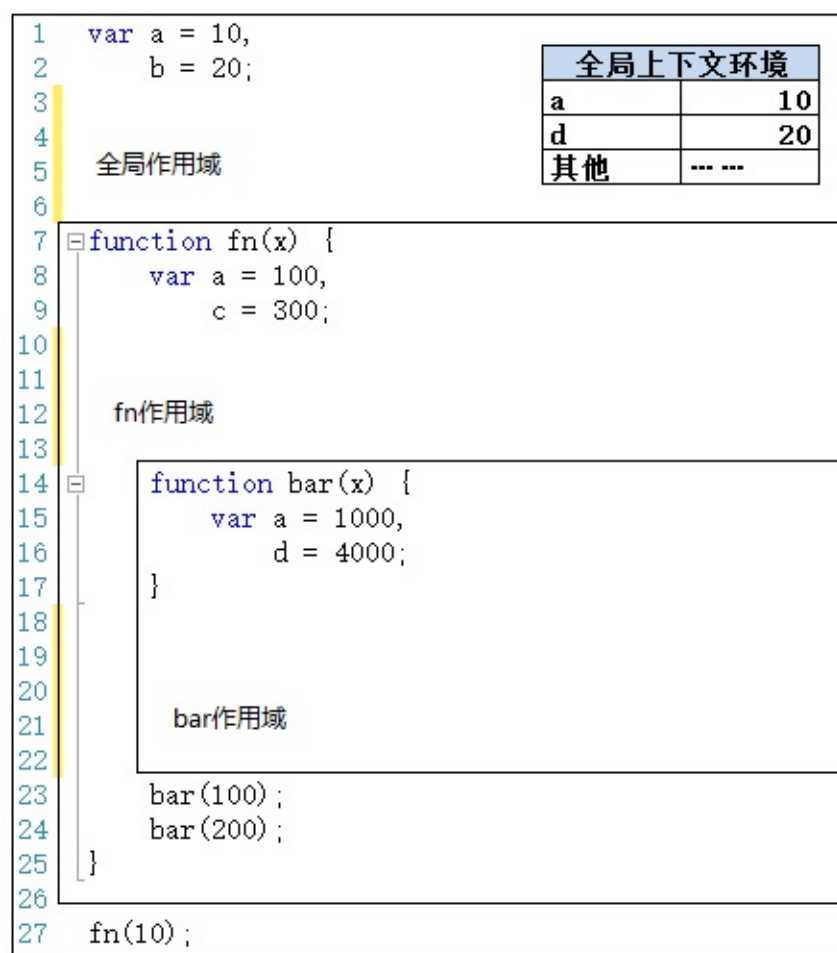
如上图，我们在上文中已经介绍了，除了全局作用域之外，每个函数都会创建自己的作用域，作用域在函数定义时就已经确定了。而不是在函数调用时确定。

下面我们将按照程序执行的顺序，一步一步把各个上下文环境加上。另外，对上下文环境不了解的朋友，可以去看看之前的两篇文章：

<http://www.cnblogs.com/wangfupeng1988/p/3986420.html>

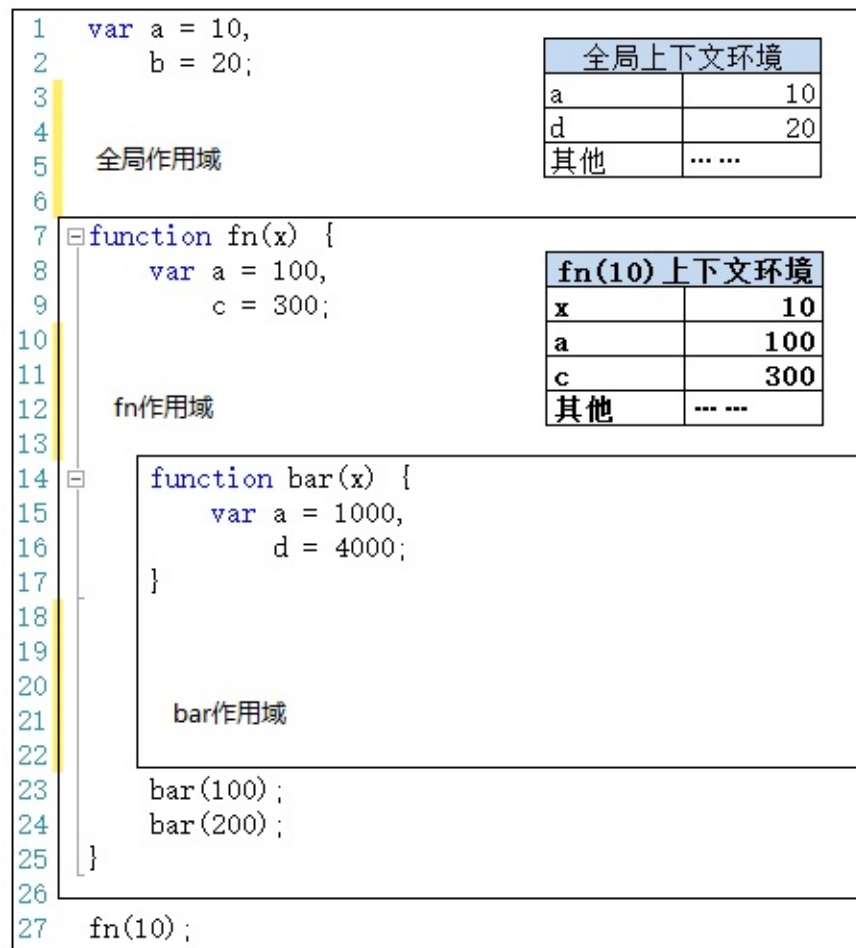
<http://www.cnblogs.com/wangfupeng1988/p/3987563.html>

第一步，在加载程序时，已经确定了全局上下文环境，并随着程序的执行而对变量就行赋值。

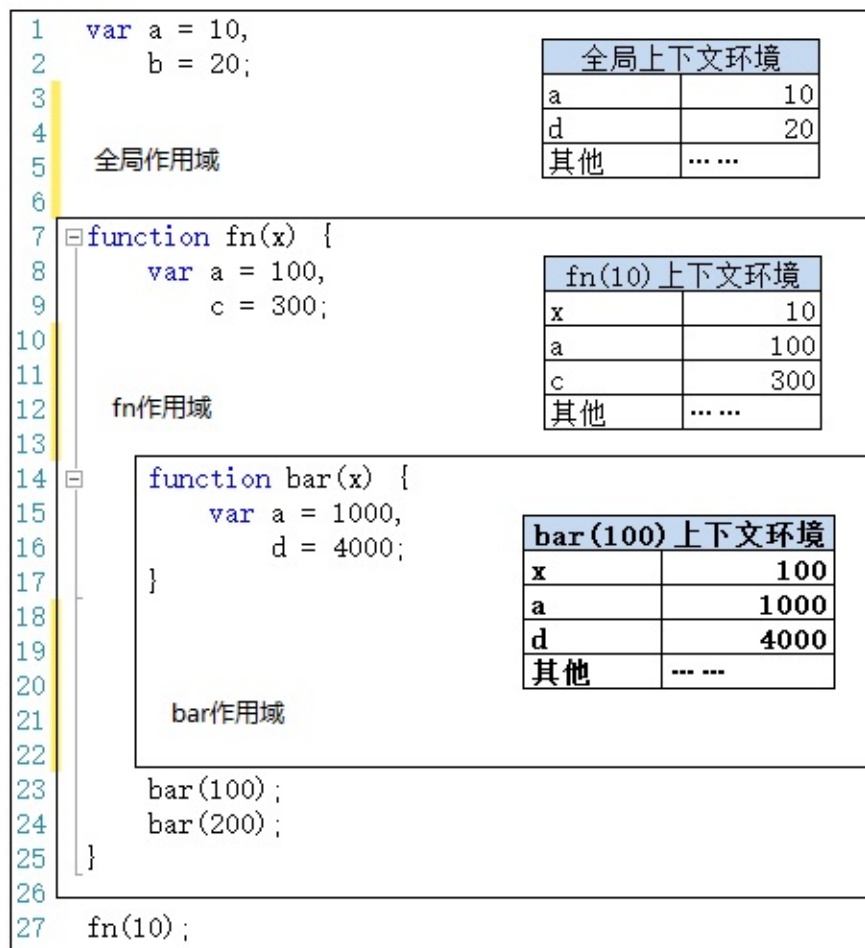


第二步，程序执行到第27行，调用fn(10)，此时生成此次调用fn函数时的上下文环境，压栈，并将此上下文环境设置为活动状态。

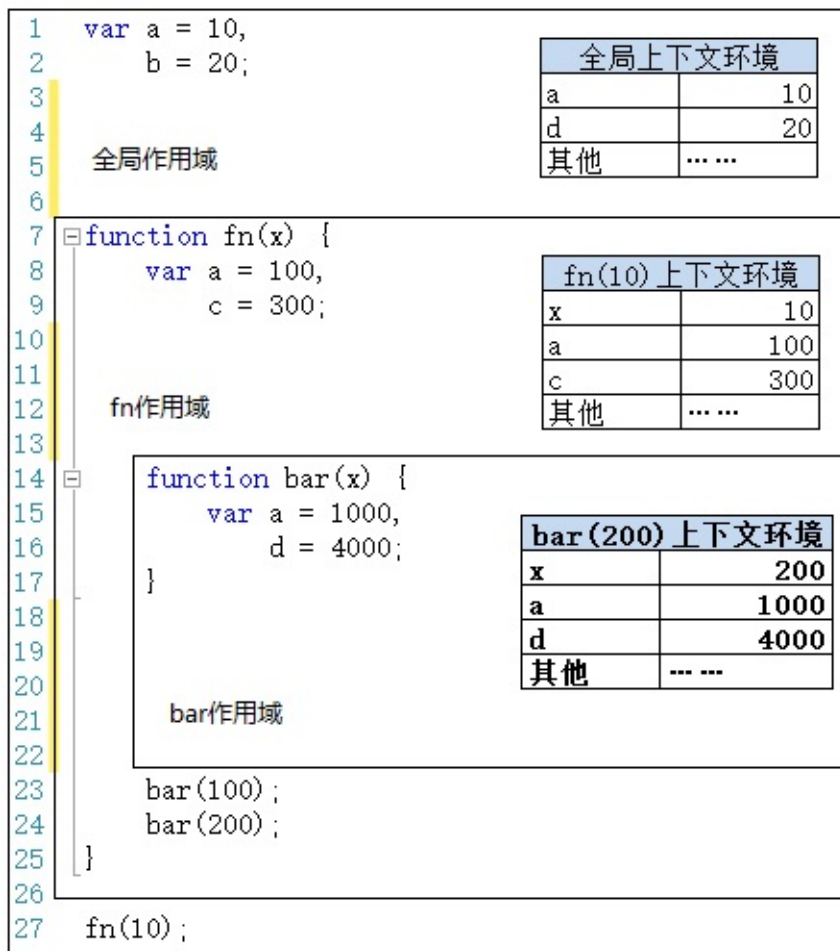




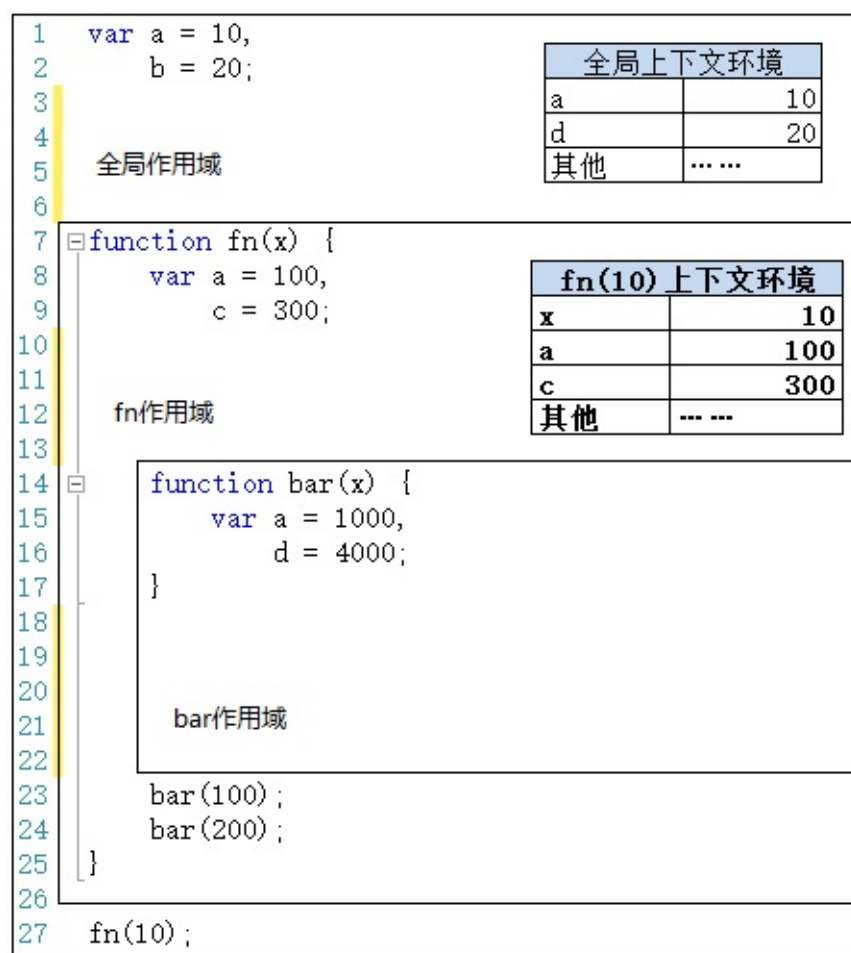
第三步，执行到第23行时，调用bar(100)，生成此次调用的上下文环境，压栈，并设置为活动状态。



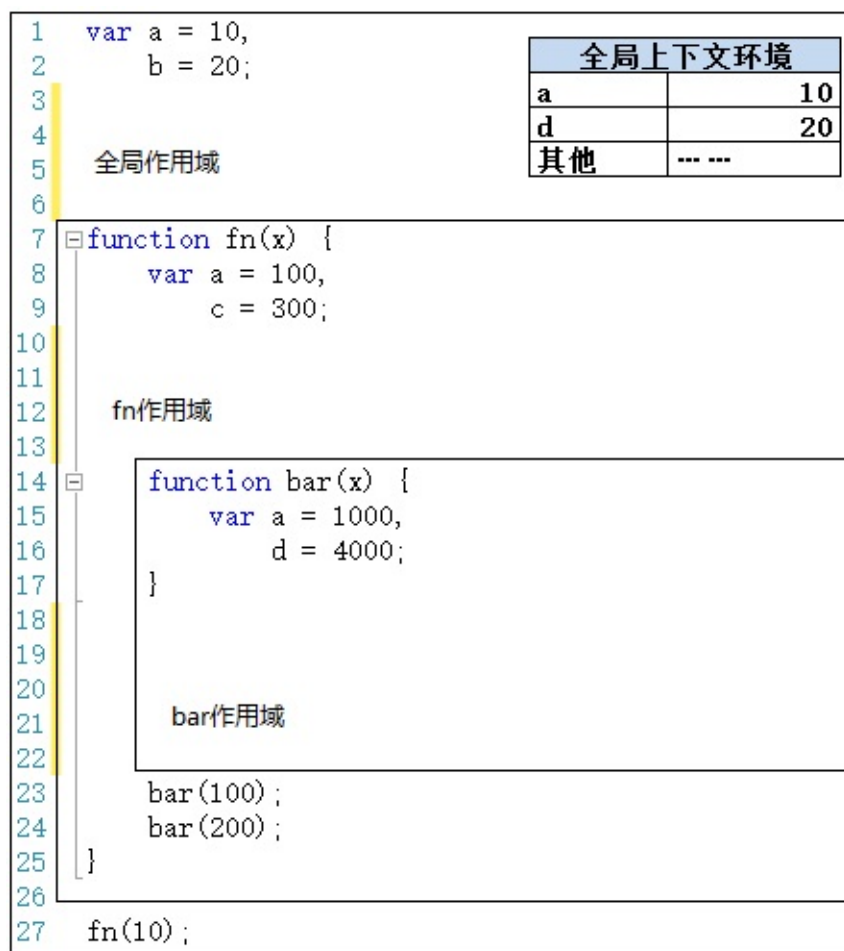
第四步，执行完第23行，bar(100)调用完成。则bar(100)上下文环境被销毁。接着执行第24行，调用bar(200)，则又生成bar(200)的上下文环境，压栈，设置为活动状态。



第五步，执行完第24行，则bar(200)调用结束，其上下文环境被销毁。此时会回到fn(10)上下文环境，变为活动状态。

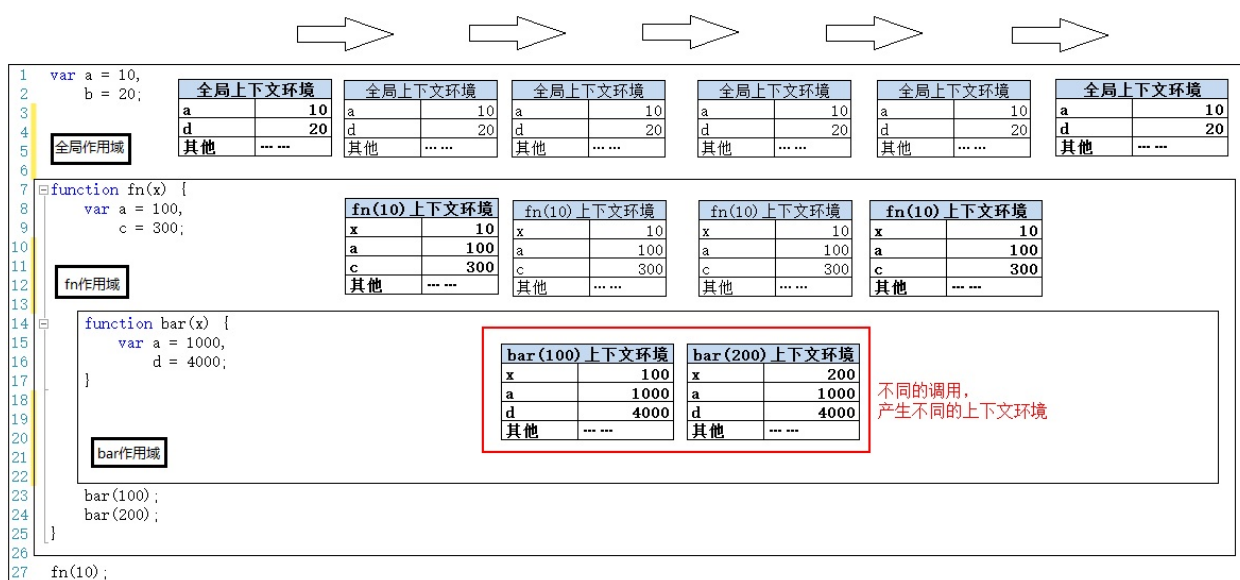


第六步，执行完第27行代码，fn(10)执行完成之后，fn(10)上下文环境被销毁，全局上下文环境又回到活动状态。



结束了。像老太太的裹脚布——又臭又长！

最后我们可以把以上这几个图片连接起来看看。



连接起来看，还是挺有意思的。作用域只是一个“地盘”，一个抽象的概念，其中没有变量。要通过作用域对应的执行上下文环境来获取变量的值。同一个作用域下，不同的调用会产生不同的执行上下文环境，继而产生不同的变量的值。所以，作用域中变量的值是在执行过程中产生的确定的，而作用域却是在函数创建时就确定了。

所以，如果要查找一个作用域下某个变量的值，就需要找到这个作用域对应的执行上下文环境，再在其中寻找变量的值。

虽然本文很长，但是文字较少，图片居多，图片都有形象的展示，大家花十几分钟也能慢慢看完。但是，这节内容真的很重要。

以上代码中，咱们还没有设计到跨作用域取值的情况，即——自由变量。详细内容且听下回分解。

## ( 14 ) ——从【自由变量】到【作用域链】

先解释一下什么是“自由变量”。

在A作用域中使用的变量x，却没有在A作用域中声明（即在其他作用域中声明的），对于A作用域来说，x就是一个自由变量。如下图

```
1  var x = 10;
2
3  function fn() {
4      var b = 20;
5
6      console.log(x + b); //这里的x在这里就是一个自由变量
7  }
```

如上程序中，在调用fn()函数时，函数体中第6行。取b的值就直接可以在fn作用域中取，因为b就是在这里定义的。而取x的值时，就需要到另一个作用域中取。到哪个作用域中取呢？

有人说过要到父作用域中取，其实有时候这种解释会产生歧义。例如：

```
1  var x = 10;
2  function fn() {
3      console.log(x);
4  }
5
6  function show(f) {
7      var x = 20;
8
9      (function () {
10         f(); //10，而不是20
11     })();
12 }
13 show(fn);
```

所以，不要在用以上说法了。相比而言，用这句话描述会更加贴切——要到创建这个函数的那个作用域中取值——是“创建”，而不是“调用”，切记切记——其实这就是所谓的“静态作用域”。

对于本文第一段代码，在fn函数中，取自由变量x的值时，要到哪个作用域中取？——要到创建fn函数的那个作用域中取——无论fn函数将在哪里调用。

上面描述的只是跨一步作用域去寻找。

如果跨了一步，还没找到呢？——接着跨！——一直跨到全局作用域为止。要是在全局作用域中都没有找到，那就是真的没有了。

这个一步一步“跨”的路线，我们称之为——作用域链。

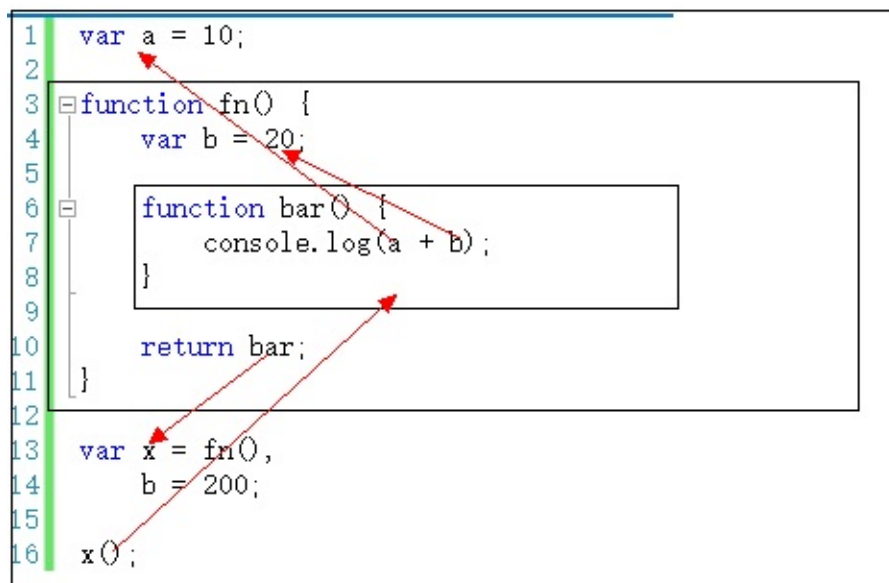
我们拿文字总结一下取自由变量时的这个“作用域链”过程：（假设a是自由量）

第一步，现在当前作用域查找a，如果有则获取并结束。如果没有则继续；

第二步，如果当前作用域是全局作用域，则证明a未定义，结束；否则继续；

第三步，（不是全局作用域，那就是函数作用域）将创建该函数的作用域作为当前作用域；

第四步，跳转到第一步。



以上代码中：第13行，`fn()`返回的是`bar`函数，赋值给`x`。执行`x()`，即执行`bar`函数代码。取`b`的值时，直接在`fn`作用域取出。取`a`的值时，试图在`fn`作用域取，但是取不到，只能转向创建`fn`的那个作用域中去查找，结果找到了。

这一节看似很轻松的把作用域链引出来，并讲完了。之所有轻松是有前几节的基础，否则将很难解释。

接下来咱们开始正式说说一直期待依旧的朋友——闭包。敬请期待下一节。



## ( 15 ) ——闭包

---

前面提到的上下文环境和作用域的知识，除了了解这些知识之外，还是理解闭包的基础。

至于“闭包”这个词的概念的文字描述，确实不好解释，我看过很多遍，但是现在还是记不住。

但是你只需要知道应用的两种情况即可——函数作为返回值，函数作为参数传递。

第一，函数作为返回值

```
1 function fn() {  
2     var max = 10;  
3  
4     return function bar(x) {  
5         if (x > max) {  
6             console.log(x);  
7         }  
8     };  
9 }  
10  
11 var f1 = fn();  
12 f1(15);
```

如上代码，bar函数作为返回值，赋值给f1变量。执行f1(15)时，用到了fn作用域下的max变量的值。至于如何跨作用域取值，可以参考上一节。

第二，函数作为参数被传递

```
1 var max = 10,  
2 fn = function (x) {  
3     if (x > max) {  
4         console.log(x);  
5     }  
6 };  
7  
8 (function (f) {  
9  
10     var max = 100;  
11     f(15);  
12  
13 })(fn);
```

如上代码中，fn函数作为一个参数被传递进入另一个函数，赋值给f参数。执行f(15)时，max变量的取值是10，而不是100。

上一节讲到自由变量跨作用域取值时，曾经强调过：要去创建这个函数的作用域取值，而不是“父作用

域”。理解了这一点，以上两端代码中，自由变量如何取值应该比较简单。（不明白的朋友一定要去上一节看看，这个很重要！）

另外，讲到闭包，除了结合着作用域之外，还需要结合着执行上下文栈来说一下。

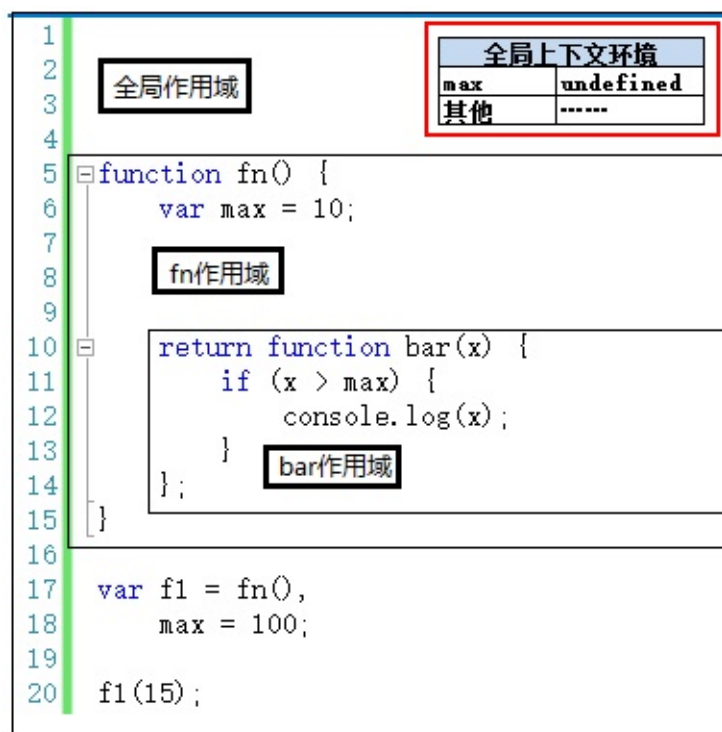
在前面讲执行上下文栈时（<http://www.cnblogs.com/wangfupeng1988/p/3989357.html>），我们提到当一个函数被调用完成之后，其执行上下文环境将被销毁，其中的变量也会被同时销毁。

但是在当时那篇文章中留了一个问号——有些情况下，函数调用完成之后，其执行上下文环境不会接着被销毁。这就是需要理解闭包的核心内容。

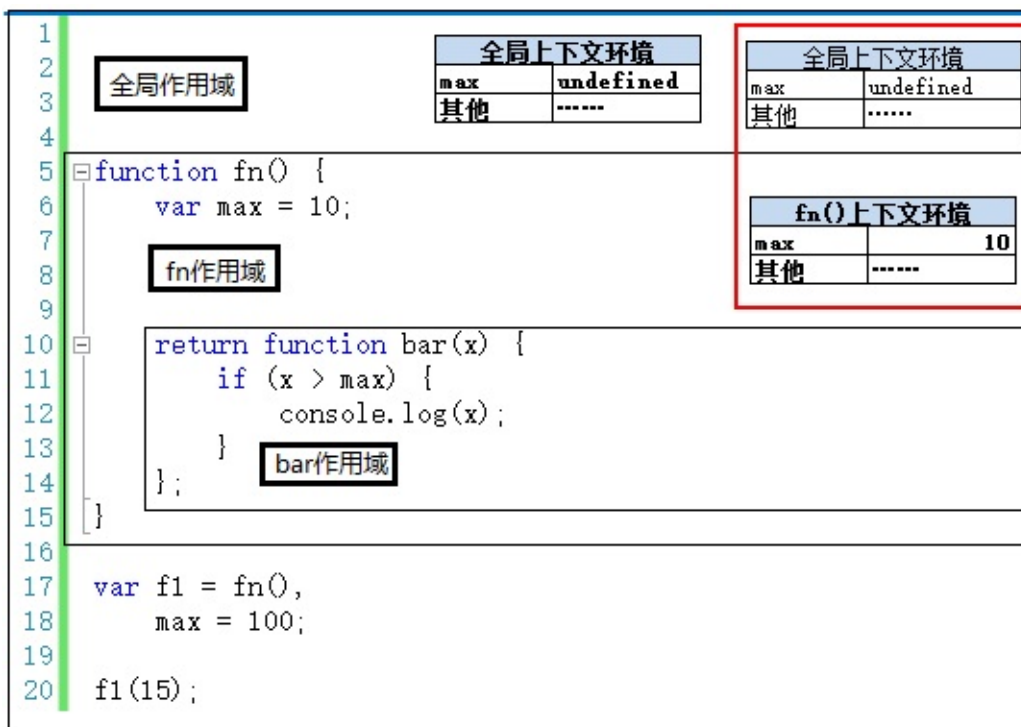
咱们可以拿本文的第一段代码（稍作修改）来分析一下。

```
1
2
3
4
5 function fn() {
6   var max = 10;
7
8   fn作用域
9
10  return function bar(x) {
11    if (x > max) {
12      console.log(x);
13    }
14    bar作用域
15  };
16
17  }
18
19  var f1 = fn(),
20    max = 100;
21
22  f1(15);
```

第一步，代码执行前生成全局上下文环境，并在执行时对其中的变量进行赋值。此时全局上下文环境是活动状态。



第二步，执行第17行代码时，调用fn()，产生fn()执行上下文环境，压栈，并设置为活动状态。



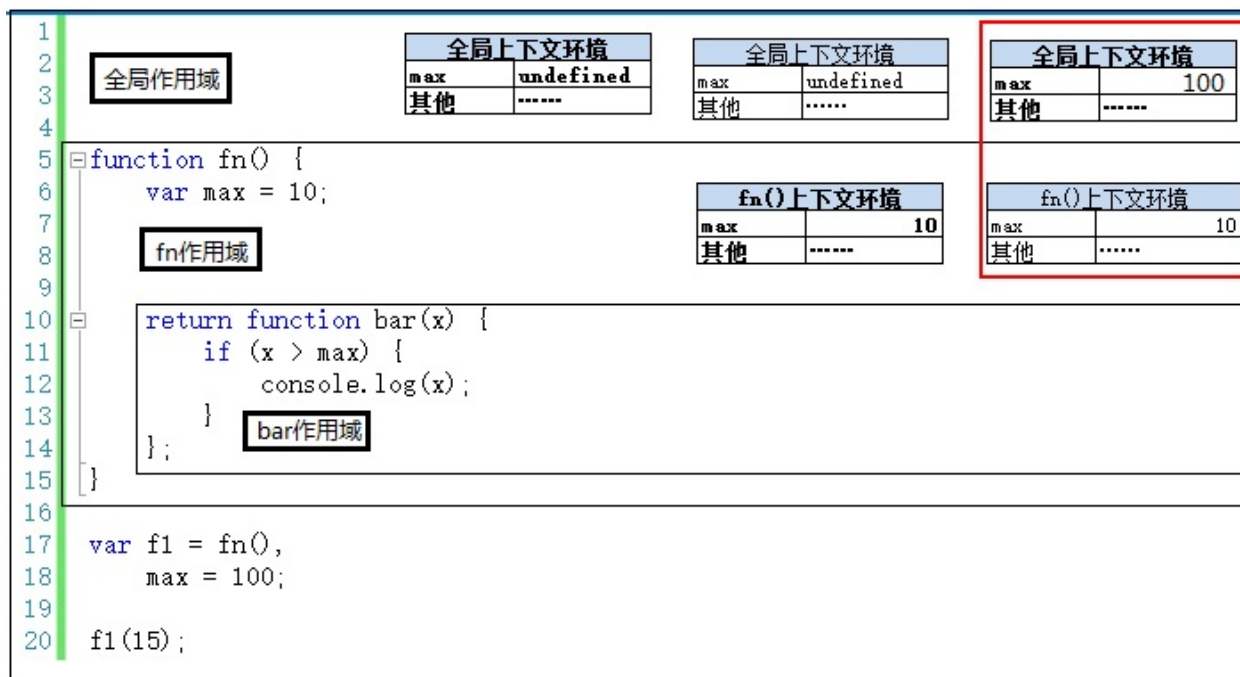
第三步，执行完第17行，fn()调用完成。按理说应该销毁掉fn()的执行上下文环境，但是这里不能这么做。注意，重点来了：因为执行fn()时，返回的是一个函数。函数的特别之处在于可以创建一个独立的作用域。而正巧合的是，返回的这个函数体中，还有一个自由变量max要引用fn作用域下的fn()上下文环境

本文档使用 [看云](#) 构建

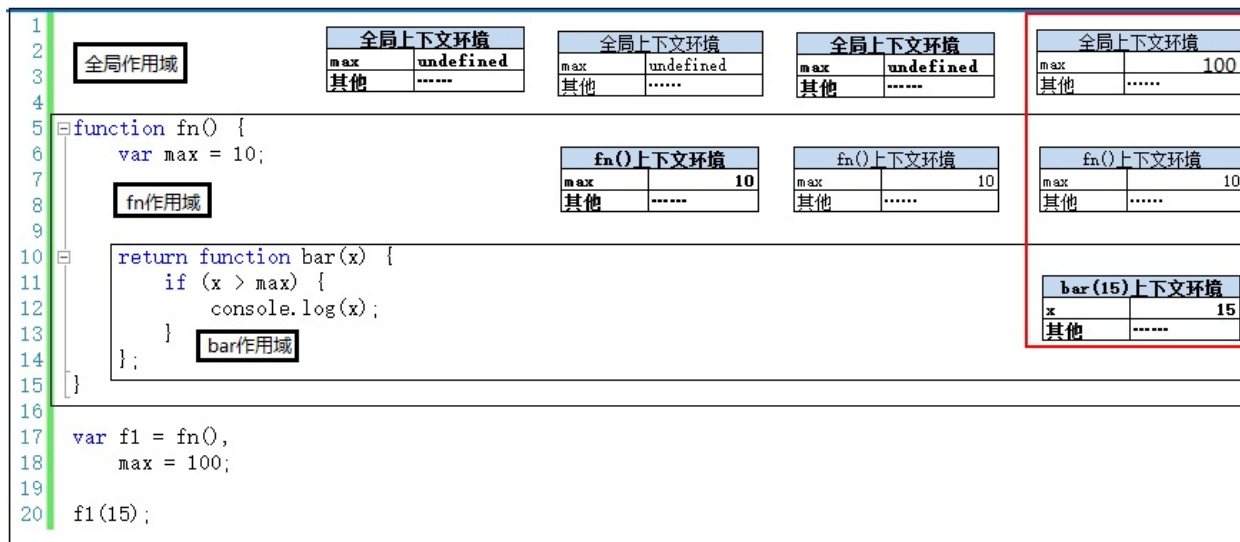
中的max。因此，这个max不能被销毁，销毁了之后bar函数中的max就找不到值了。

因此，这里的fn()上下文环境不能被销毁，还依然存在与执行上下文栈中。

——即，执行到第18行时，全局上下文环境将变为活动状态，但是fn()上下文环境依然会在执行上下文栈中。另外，执行完第18行，全局上下文环境中的max被赋值为100。如下图：



第四步，执行到第20行，执行f1(15)，即执行bar(15)，创建bar(15)上下文环境，并将其设置为活动状态。



执行bar(15)时，max是自由变量，需要向创建bar函数的作用域中查找，找到了max的值为10。这个过程

在作用域链一节已经讲过。

这里的重点就在于，创建bar函数是在执行fn()时创建的。fn()早就执行结束了，但是fn()执行上下文环境还存在与栈中，因此bar(15)时，max可以查找到。如果fn()上下文环境销毁了，那么max就找不到了。

使用闭包会增加内容开销，现在很明白了吧！

第五步，执行完20行就是上下文环境的销毁过程，这里就不再赘述了。

闭包和作用域、上下文环境有着密不可分的关系，真的是“想说爱你不容易”！

另外，闭包在jQuery中的应用非常多，在这里就不一一举例子了。所以，无论你是想了解一个经典的框架/类库，还是想自己开发一个插件或者类库，像闭包、原型这些基本的理论，是一定要知道的。否则，到时候出了BUG你都不知道为什么，因为这些BUG可能完全在你的知识范围之外。

到现在闭包就简单介绍完了，下一节我们再总结一下。

## ( 16 ) —— 完结

---

之前一共用15篇文章，把javascript的原型和闭包。

首先，javascript本来就“不容易学”。不是说它有多难，而是学习它的人，往往都是在学会了其他语言之后，又学javascript。有其他语言的学习经历和实践经历，再加上自学javascript，边学边用，肯定会产生许多误解，走许多弯路。我就没少经历，也算是一种教训。

其次，原型和闭包又是一对难兄难弟，一来是他俩比较难懂，而来是他俩都或多或少的给初级开发人员带来许多BUG。不懂原型和闭包，你也可以开发javascript程序，但是你写不出高质量、符合设计原则的javascript程序。

因此，还是强调基础，强调理论！理论和实践相结合不是一句空话。

由此想到了我平时练习投篮。一般喜欢篮球的人都是周末和别人去一起打篮球玩，我之前也是如此。但是我从今年春天开始，每周另外抽出一小时时间，自己一个人去练习投篮。练习正确的投篮姿势，强迫自己在练习的时候使用正确姿势，时间长了就形成了肌肉记忆。

所以，工作之余，切不可忘记充电。



写这些东西，我也参考了好多资料，包括博客园、csdn好多技术专家的博客，各种javascript书籍，json.js源码，jQuery源码。但是我都没有生搬硬套里面的术语和段落。

自己写东西，不是抄袭，不是写毕业论文。一定要有自己的思考和总结。学习、工作中都是如此。

此前15篇文章中所有的描述、解释、例子包括图片，都体现了我自己的理解，大部分图片都是我自己画出来的。包括在写每一篇文章之前，我都要去思考，应该用何种方式去表达，才能让读者更容易理解和接收。

最后，看到好多文章都是长篇大论，恨不得用一篇文章解释完所有的内容。而我的文章都是小步快

跑，看完一篇估计也就十分钟。这是我在看《明朝那些事儿》时发现的一个思路。这本书一节只有很少的内容，而且语言简单，演绎性很强，一节紧扣着一节，能黏住读者。

因此，我在写文章时，会先用一个大家都概念引入正题，然后进一步解释。在一篇文章的最后，要抛出一个疑问引出下一节。

原型和闭包这个系列，正好填补了javascript教程一个空缺，即使《javascript高级程序设计》中，都没有将原型和闭包讲解的如此深入。希望大家多多支持吧。

就此，结束。

## ( 17 ) ——补this

本文对《深入理解javascript原型和闭包 ( 10 ) ——this》一篇进行补充，原文链接：<http://www.cnblogs.com/wangfupeng1988/p/3988422.html>

原文中，讲解了在javascript中this的各个情况，写完之后发现还落下一情况，就此补充。

原文中this的其中一种情况是构造函数的，具体的内容可以参考原文，此处不再赘述。

要补充的内容是，在构造函数的prototype中，this代表着什么。

```
1
2 function Fn() {
3     this.name = '王福朋';
4     this.year = 1988;
5 }
6
7 Fn.prototype.getName = function () {
8     console.log(this.name);
9 }
10
11 var f1 = new Fn();
12 f1.getName();      // 王福朋
13
```

如上代码，在Fn.prototype.getName函数中，this指向的是f1对象。因此可以通过this.name获取f1.name的值。

其实，不仅仅是构造函数的prototype，即便是在整个原型链中，this代表的也都是当前对象的值。



## ( 18 ) ——补充：上下文环境和作用域的关系

---

本系列用了大量的篇幅讲解了上下文环境和作用域，有些人反映这两个是一回事儿。本文就用一个小例子来说明一下，作用域和上下文环境绝对不是一回事儿。

再说明之前，咱们先用简单的语言来概括一下这两个的区别。

### 00 上下文环境：

可以理解为一个看不见摸不着的对象（有若干个属性），虽然看不见摸不着，但确实实实在在存在的，因为所有的变量都在里面存储着，要不然咱们定义的变量在哪里存？

另外，对于函数来说，上下文环境是在调用时创建的，这个很好理解。拿参数做例子，你不调用函数，我哪儿知道你要给我传什么参数？

### 01 作用域：

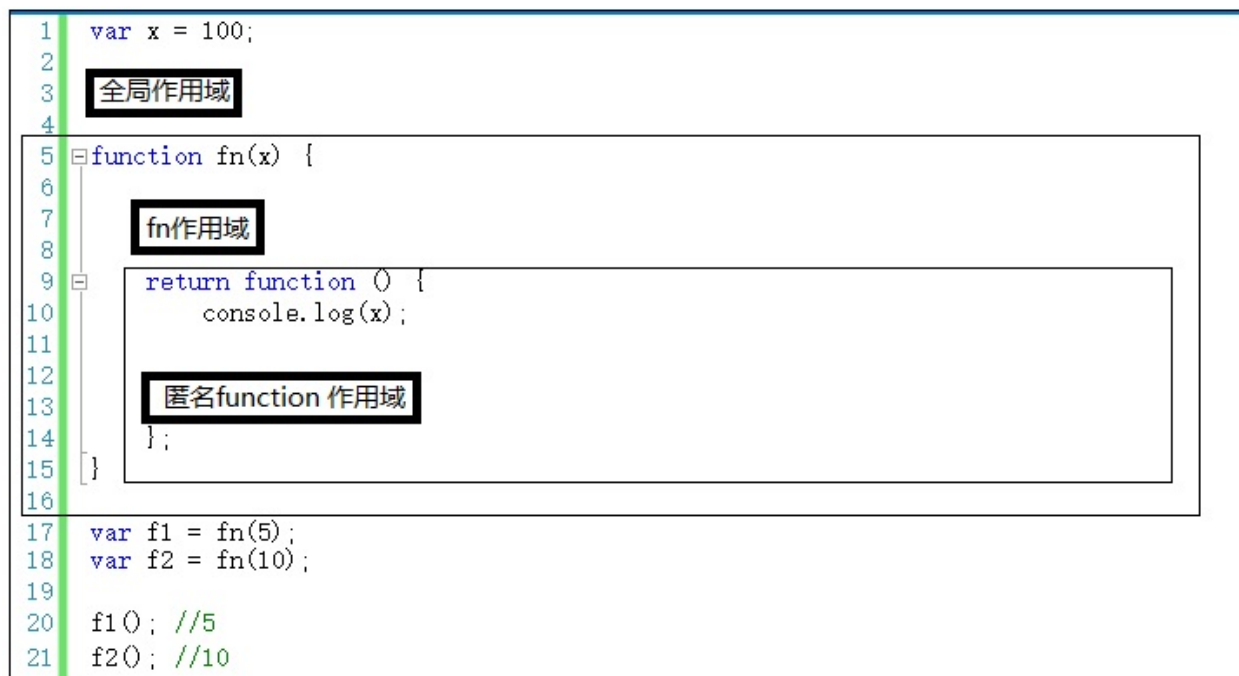
首先，它很抽象。第二，记住一句话：除了全局作用域，只有函数才能创建作用域。创建一个函数就创建了一个作用域，无论你调用不调用，函数只要创建了，它就有独立的作用域，就有自己的一个“地盘”。

### 02 两者：

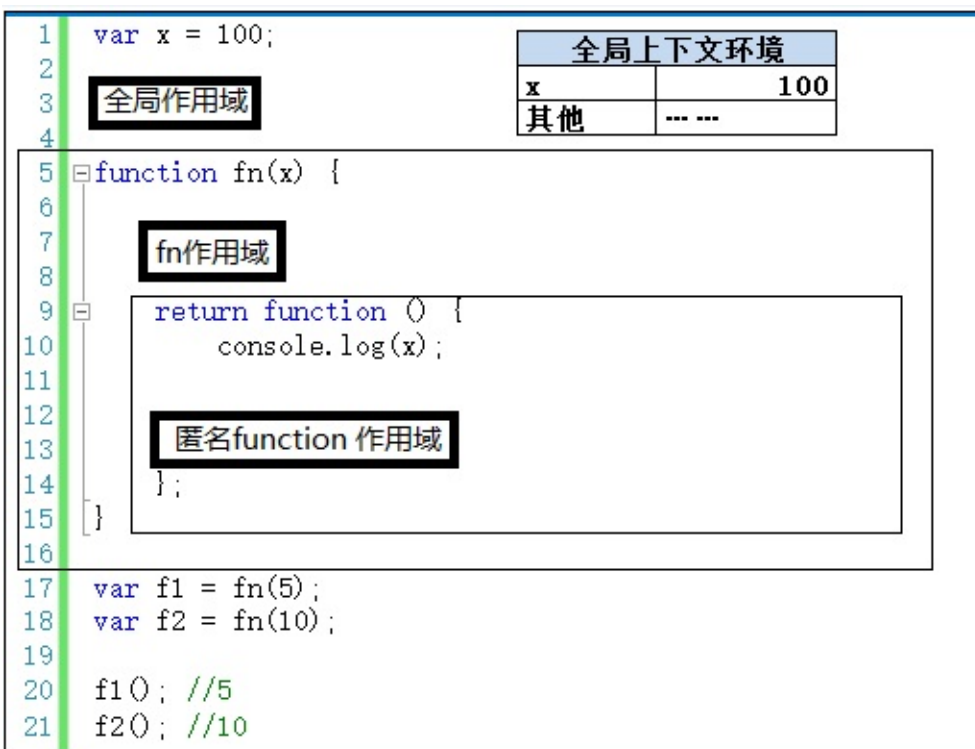
一个作用域下可能包含若干个上下文环境。有可能从来没有过上下文环境（函数从来就没有被调用过）；有可能有过，现在函数被调用完毕后，上下文环境被销毁了；有可能同时存在一个或多个（闭包）。

上面的文字不理解没关系，且看下面的例子。

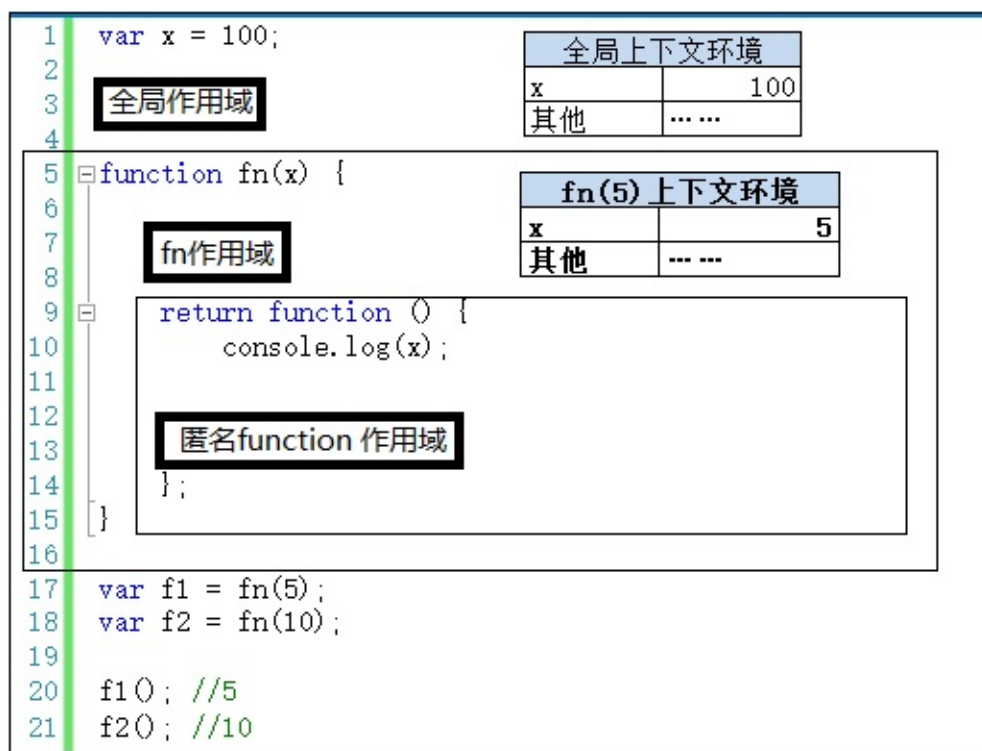
第一，除了全局作用域外，每个函数都要创建一个作用域。作用域之间的变量是相互独立的。因此，全局作用域中的x和fn作用域中的x，两者毫无关系，互不影响，和平相处。



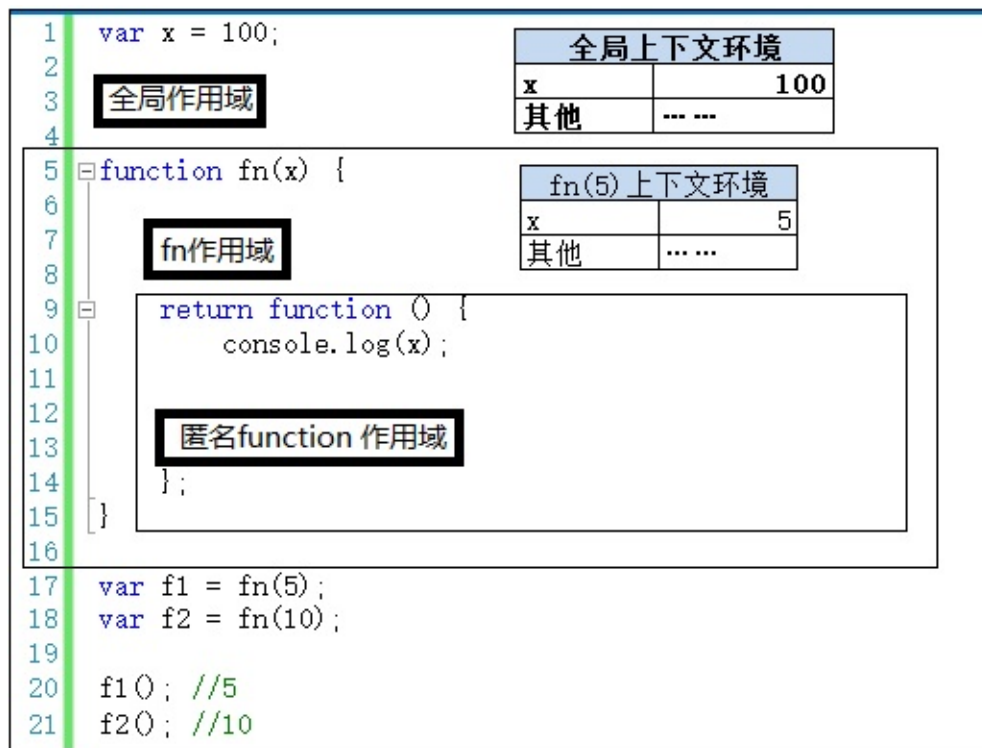
第二，程序执行之前，会生成全局上下文环境，并在程序执行时，对其中的变量赋值。



第三，程序执行到第17行，调用fn(5)，会产生fn(5)的上下文环境，并压栈，并设置为活动状态。

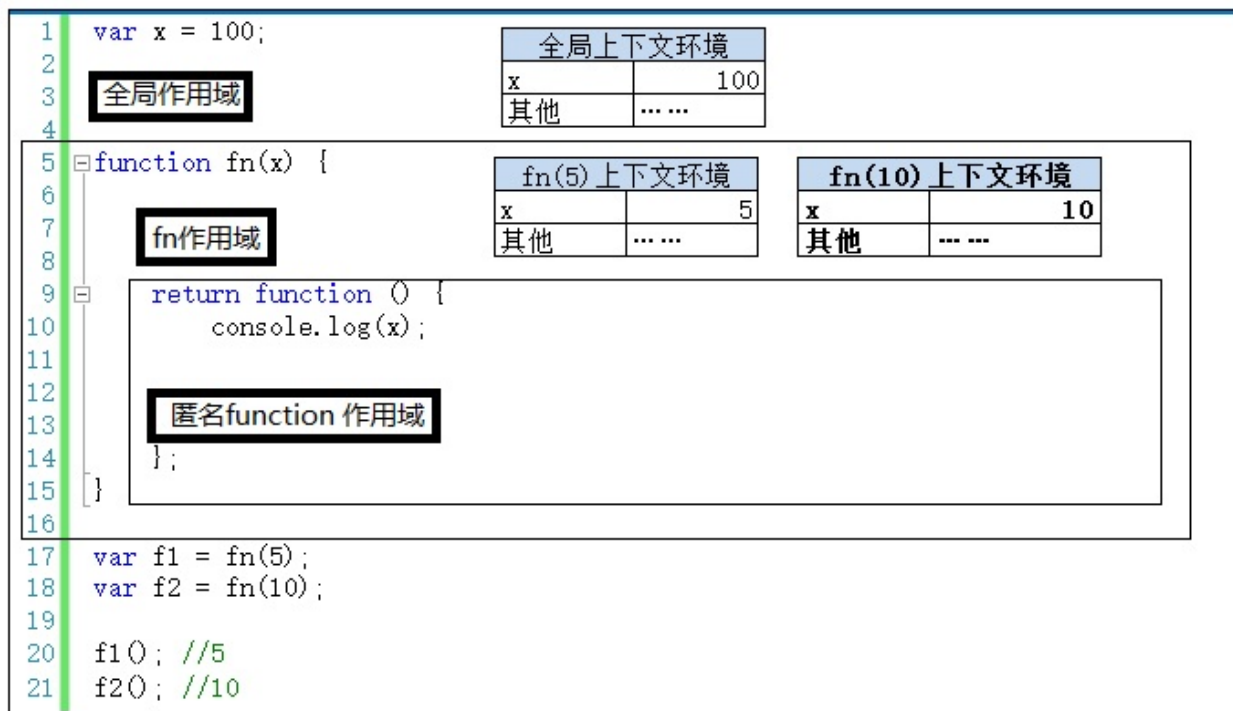


第四，执行完第17行，fn(5)的返回值赋值给了f1。此时执行上下文环境又重新回到全局，但是fn(5)的上下文环境不能就此销毁，因为其中有闭包的引用（可翻看前面文章，此处不再赘述）。



第五，继续执行第18行，再次调用fn函数——fn(10)。产生fn(5)的上下文环境，并压栈，并设置为活动状态

态。但是此时fn(5)的上下文环境还在内存中——一个作用域下同时存在两个上下文环境。



讲到这里，重点已经讲出来了，之后的场景这里就不再赘述了。

目的还是希望大家能通过这个例子，来理清上下文环境和作用域的关系。当然，也不是非得像个学院派似的一字一文的把概念说出来，简单理解一下，对用闭包是有帮助的。