

# 前端基础进阶（六）：在chrome开发者工具中观察函数调用栈、作用域链与闭包



这波能反杀  尊享会员



213.611 2017-02-19 21:26  打开App



配图与本文无关

在前端开发中，有一个非常重要的技能，叫做断点调试。

在chrome的开发者工具中，通过断点调试，我们能够非常方便的一步一步的观察JavaScript的执行过程，直观感知函数调用栈，作用域链，变量对象，闭包，this等关键信息的变化。因此，断点调试对于快速定位代码错误，快速了解代码的执行过程有着非常重要的作用，这也是我们前端开发者必不可少的一个高级技能。

当然如果你对JavaScript的这些基础概念（执行上下文，变量对象，闭包，this等）了

解还不够的话，想要透彻掌握断点调试可能会有一些困难。但是好在在前面几篇文章，我都对这些概念进行了详细的概述，因此要掌握这个技能，对大家来说，应该会比较轻松的。

这篇文章的主要目的在于借助对于断点调试的学习，来进一步加深对闭包的理解。

## 一、基础概念回顾

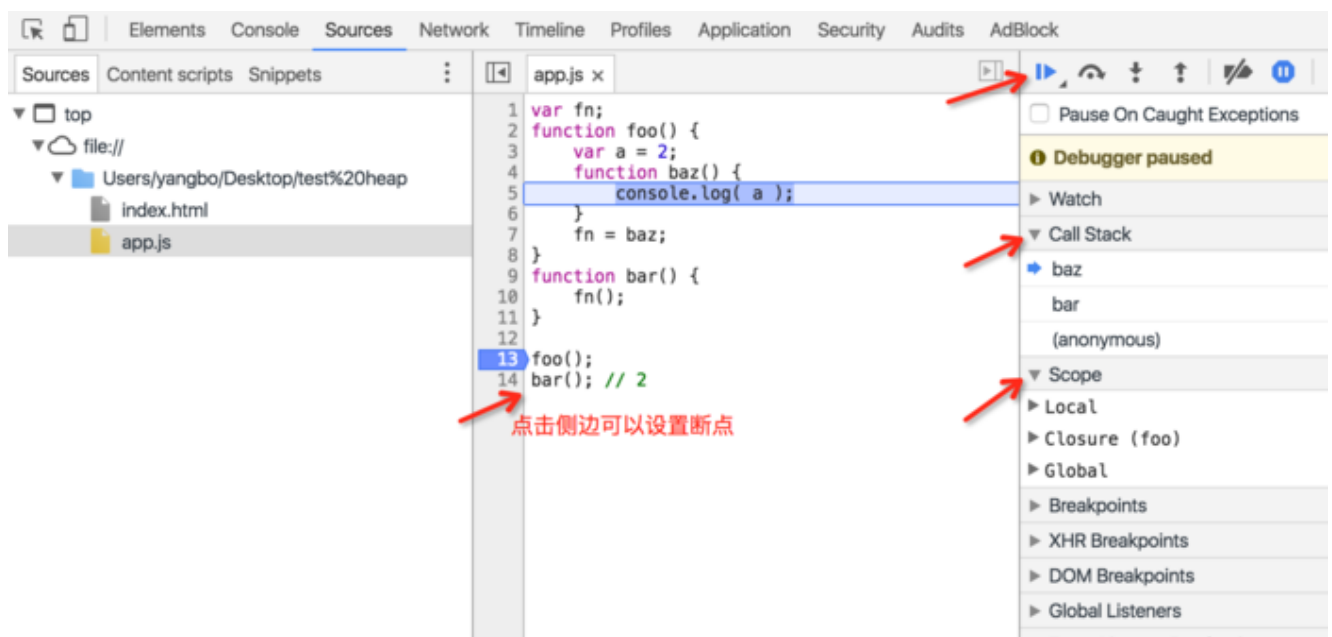
函数在被调用执行时，会创建一个当前函数的执行上下文。在该执行上下文的创建阶段，变量对象、作用域链、闭包、this指向会分别被确定。而一个JavaScript程序中一般来说会有多个函数，JavaScript引擎使用函数调用栈来管理这些函数的调用顺序。函数调用栈的调用顺序与栈数据结构一致。

## 二、认识断点调试工具

在尽量新版本的chrome浏览器中（不确定你用的老版本与我的一致），调出chrome浏览器的开发者工具。

浏览器右上角竖着的三点 -> 更多工具 -> 开发者工具 -> Sources

界面如图。



在我的demo中，我把代码放在app.js中，在index.html中引入。我们暂时只需要关注截图中红色箭头的地方。在最右侧上方，有一排图标。我们可以通过使用他们来控制函数的执行顺序。从左到右他们依次是：

- resume/pause script execution

恢复/暂停脚本执行

- **step over next function call**

跨过，实际表现是不遇到函数时，执行下一步。遇到函数时，不进入函数直接执行下一步。

- **step into next function call**

跨入，实际表现是不遇到函数时，执行下一步。遇到函数时，进入函数执行上下文。

- **step out of current function**

跳出当前函数

- deactivate breakpoints

停用断点

- don't pause on exceptions

不暂停异常捕获

其中跨过，跨入，跳出是我使用最多的三个操作。

上图右侧第二个红色箭头指向的是函数调用栈（call Stack），这里会显示代码执行过程中，调用栈的变化。

右侧第一个红色箭头指向的是作用域链（Scope），这里会显示当前函数的作用域链。

右侧第二行红色箭头指向的是作用域链（Scope），这里会显示当前函数的作用域链。其中Local表示当前的局部变量对象，Closure表示当前作用域链中的闭包。借助此处的作用域链展示，我们可以很直观的判断出一个例子中，到底谁是闭包，对于闭包的深入了解具有非常重要的帮助作用。

### 三、断点设置

在显示代码行数的地方点击，即可设置一个断点。断点设置有以下几个特点：

- 在单独的变量声明(如果没有赋值)，函数声明的那一行，无法设置断点。
- 设置断点后刷新页面，JavaScript代码会执行到断点位置处暂停执行，然后我们就可以使用上边介绍过的几个操作开始调试了。
- 当你设置多个断点时，chrome工具会自动判断从最早执行的那个断点开始执行，因此我一般都是设置一个断点就行了。

### 四、实例

接下来，我们借助一些实例，来使用断点调试工具，看一看，我们的demo函数，在执行过程中的具体表现。

```
// demo01

var fn;
function foo() {
  var a = 2;
  function baz() {
    console.log( a );
  }
  fn = baz;
}

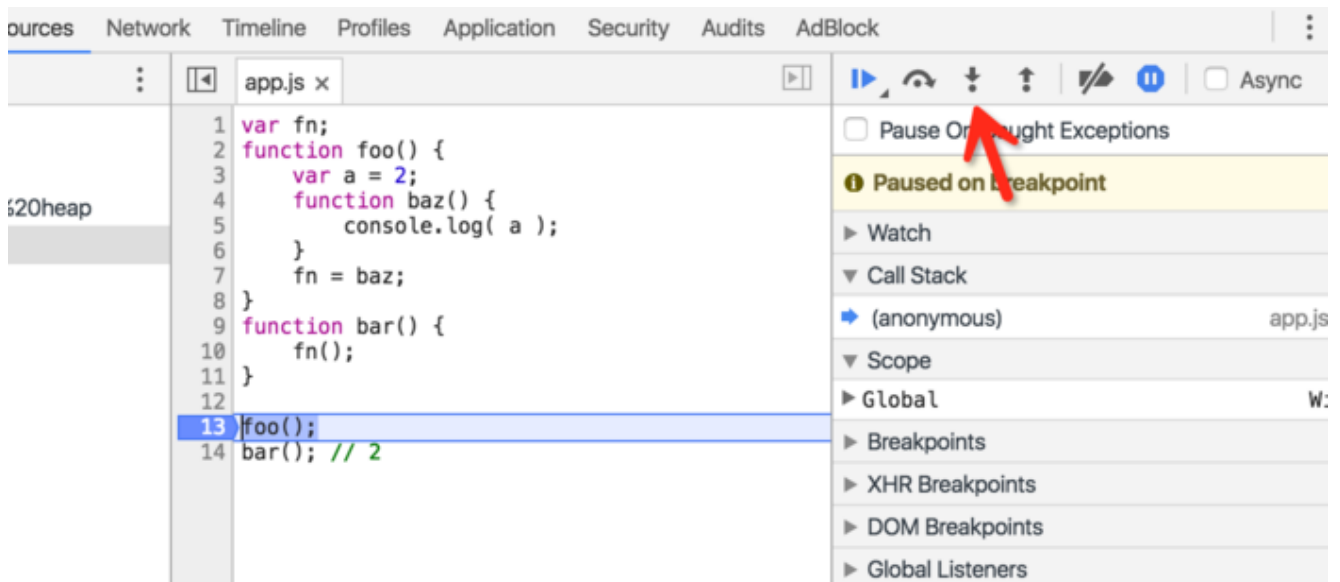
function bar() {
```

```
fn();  
  
}  
  
foo();  
  
bar(); // 2
```

在向下阅读之前，我们可以停下来思考一下，这个例子中，谁是闭包？

这是来自《你不知道的js》中的一个例子。由于在使用断点调试过程中，发现chrome浏览器理解的闭包与该例子中所理解的闭包不太一致，因此专门挑出来，供大家参考。我个人更加倾向于chrome中的理解。

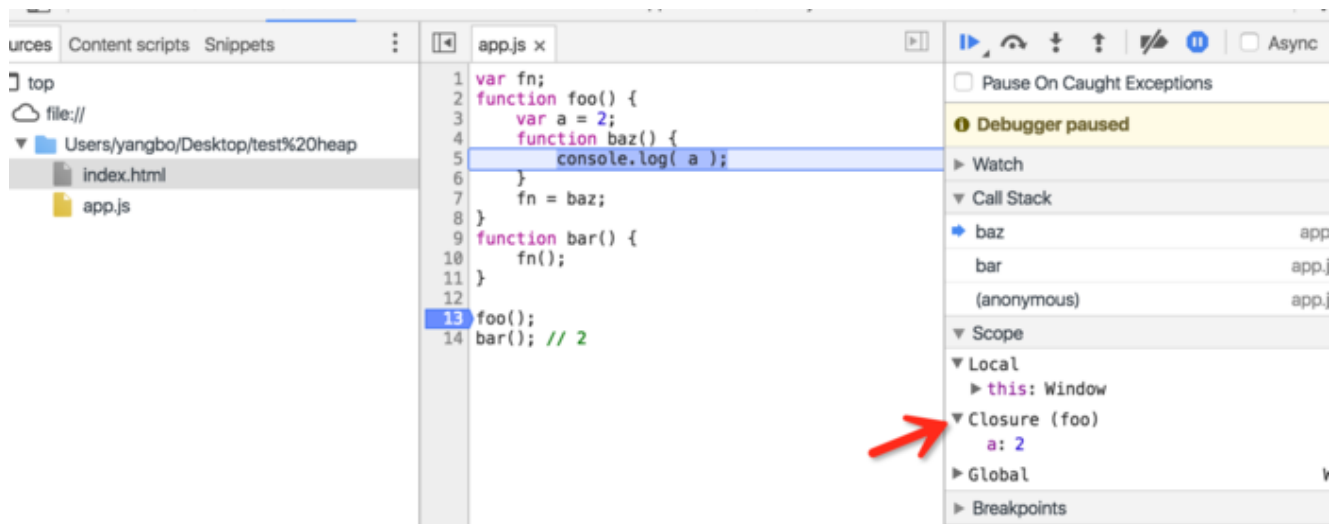
- 第一步：设置断点，然后刷新页面。



设置断点

- 第二步：点击上图红色箭头指向的按钮（step into），该按钮的作用会根据代码执行顺序，一步一步向下执行。在点击的过程中，我们要注意观察下方call stack 与 scope的变化，以及函数执行位置的变化。

一步一步执行，当函数执行到上例子中



baz函数被调用执行，foo形成了闭包

我们可以看到，在chrome工具的理解中，由于在foo内部声明的baz函数在调用时访问了它的变量a，因此foo成为了闭包。这好像和我们学习到的知识不太一样。我们来看看在《你不知道的js》这本书中的例子中的理解。

传递函数当然也可以是间接的。

```
var fn;

function foo() {
  var a = 2;

  function baz() {
    console.log( a );
  }

  fn = baz; // 将 baz 分配给全局变量
}

function bar() {
  fn(); // 妈妈快看呀，这就是闭包!
}

foo();

bar(); // 2
```

书中的注释可以明显的看出，作者认为fn为闭包。即baz，这和chrome工具中明显是不一样的。

而在备受大家推崇的《JavaScript高级编程》一书中，是这样定义闭包。

## 7.2 闭包

有不少开发人员总是搞不清匿名函数和闭包这两个概念，因此经常混用。闭包是指有权访问另一个函数作用域中的变量的函数。创建闭包的常见方式，就是在**一个函数内部创建另一个函数**，仍以前面的createComparisonFunction()函数为例，注意加粗的代码。

```
function createComparisonFunction(propertyName) {
    return function(object1, object2){
        var value1 = object1[propertyName];
        var value2 = object2[propertyName];

        if (value1 < value2) {
```

JavaScript高级编程中闭包的定义

### 7.2.1 闭包与变量

作用域链的这种配置机制引出了一个值得注意的副作用，即闭包只能取得包含函数中任何变量的最后一个值。别忘了闭包所保存的是整个变量对象，而不是某个特殊的变量。下面这个例子可以清晰地说明这个问题。



```
function createFunctions(){
    var result = new Array();

    for (var i=0; i < 10; i++){
        result[i] = function(){
```

书中作者将自己理解的闭包与包含函数所区分

这里chrome中理解的闭包，与我所阅读的这几本书中的理解的闭包不一样。其实在之前对于闭包分析的文章中，我已经有对这种情况做了一个解读。[闭包详解](#)

闭包是一个特殊对象，它由执行上下文(代号A)与在该执行上下文中创建的函数(代号B)共同组成。

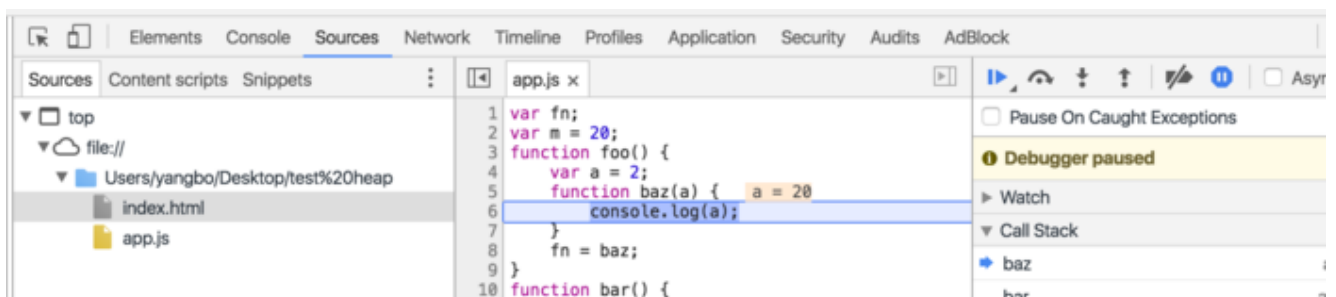
当B执行时，如果访问了A中变量对象中的值，那么闭包就会产生。

那么在大多数理解中，包括许多著名的书籍，文章里都以函数B的名字代指这里生成的闭包。而在chrome中，则以执行上下文A的函数名代指闭包。

我们修改一下demo01中的例子，来看看一个非常有意思的变化。

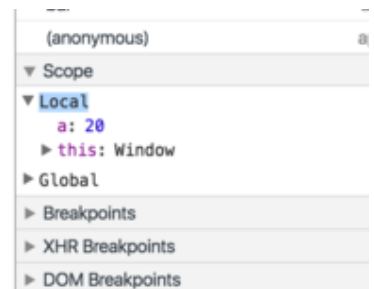
```
// demo02  
  
var fn;  
var m = 20;  
  
function foo() {  
    var a = 2;  
  
    function baz(a) {  
        console.log(a);  
    }  
  
    fn = baz;  
}  
  
function bar() {  
    fn(m);  
}  
  
foo();  
bar(); // 20
```

这个例子在demo01的基础上，我在baz函数中传入一个参数，并打印出来。在调用时，我将全局的变量m传入。输出结果变为20。在使用断点调试看看作用域链。





```
11     fn(m);
12 }
13
14 foo();
15 bar(); // 2
```



闭包没了，作用域链中没有包含foo了。

是不是结果有点意外，闭包没了，作用域链中没有包含foo了。我靠，跟我们理解的好像又有点不一样。所以通过这个对比，我们可以确定闭包的形成需要两个条件。

- 在函数内部创建新的函数；
- 新的函数在执行时，访问了函数的变量对象；

还有更有意思的。

我们继续来看看一个例子。

```
// demo03
```

```
function foo() {
    var a = 2;

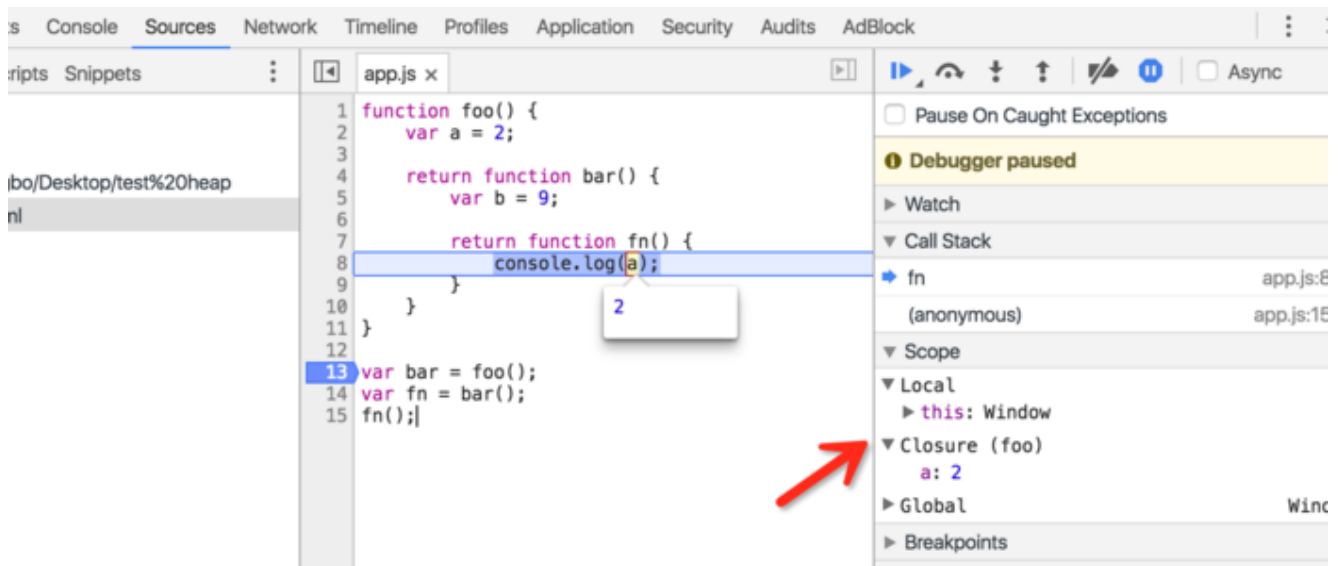
    return function bar() {
        var b = 9;

        return function fn() {
            console.log(a);
        }
    }
}
```

```
var bar = foo();
```

```
var fn = bar();  
  
fn();
```

在这个例子中，fn只访问了foo中的a变量，因此它的闭包只有foo。

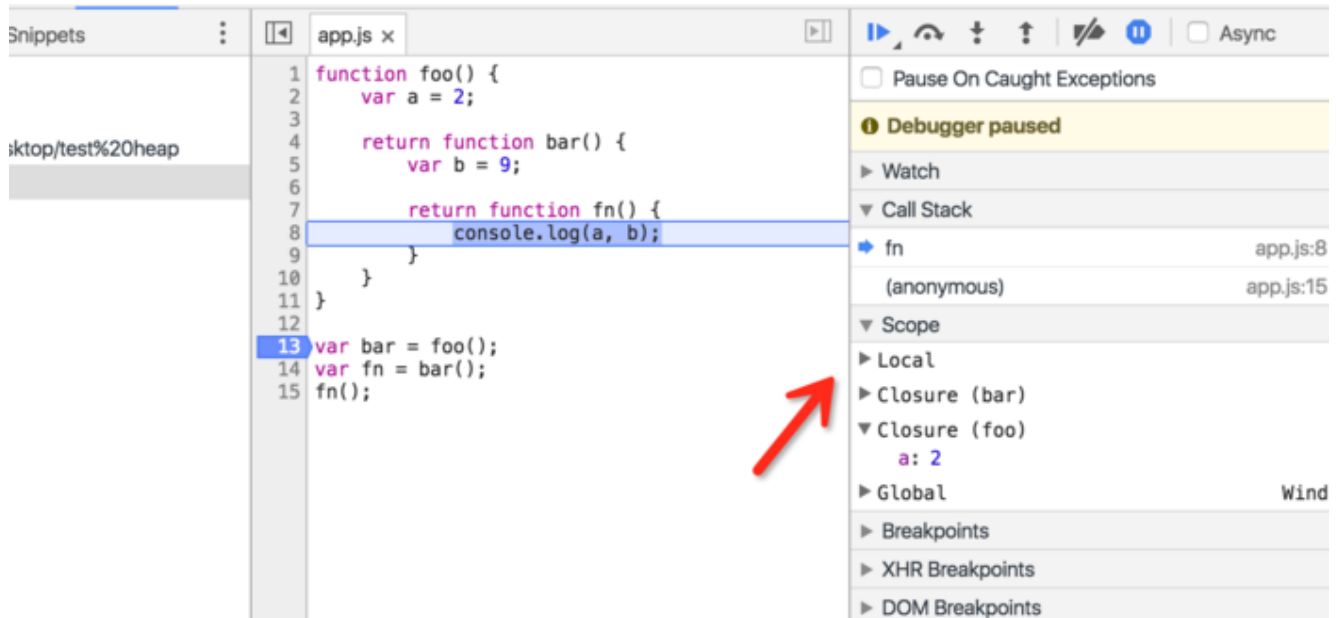


闭包只有foo

修改一下demo03，我们在fn中也访问bar中b变量试试看。

```
// demo04  
  
function foo() {  
  var a = 2;  
  
  return function bar() {  
    var b = 9;  
  
    return function fn() {  
      console.log(a, b);  
    }  
  }  
}
```

```
var bar = foo();  
var fn = bar();  
fn();
```



这个时候闭包变成了两个

这个时候，闭包变成了两个。分别是`bar`，`foo`。

我们知道，闭包在模块中的应用非常重要。因此，我们来一个模块的例子，也用断点工具来观察一下。

```
// demo05  
(function() {  
  
  var a = 10;  
  var b = 20;  
  
  var test = {  
    m: 20,  
    add: function(x) {
```

```

        return a + x;
    },
    sum: function() {
        return a + b + this.m;
    },
    mark: function(k, j) {
        return k + j;
    }
}

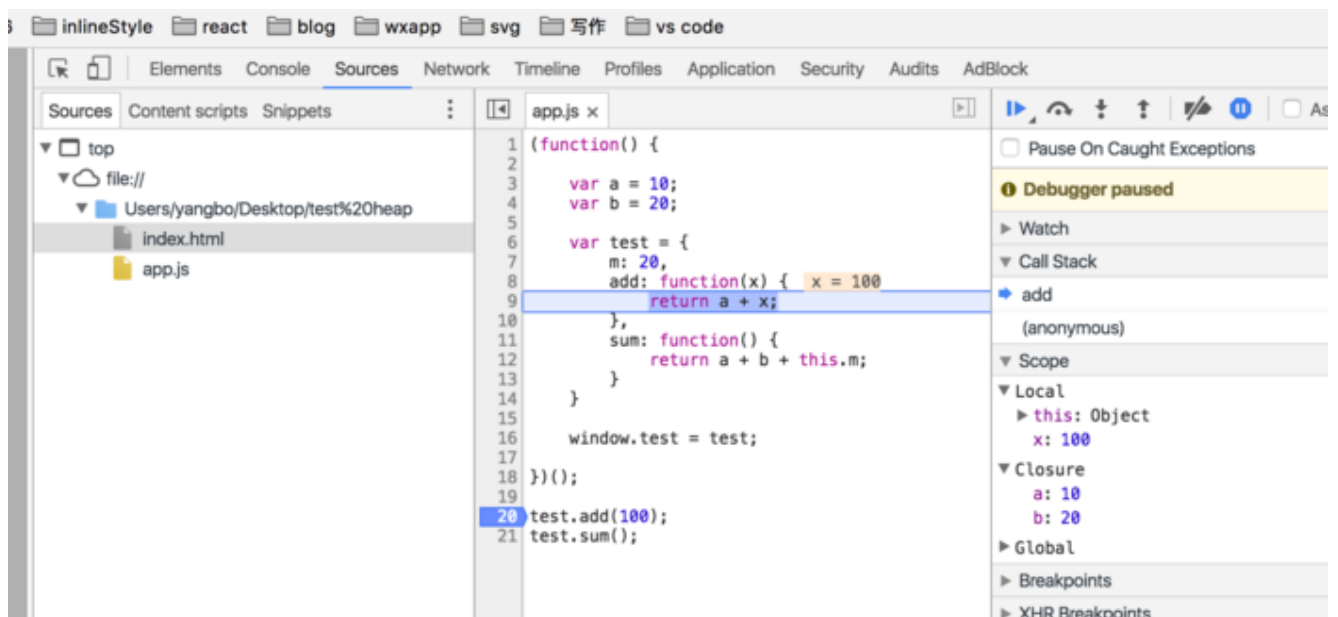
window.test = test;

})();

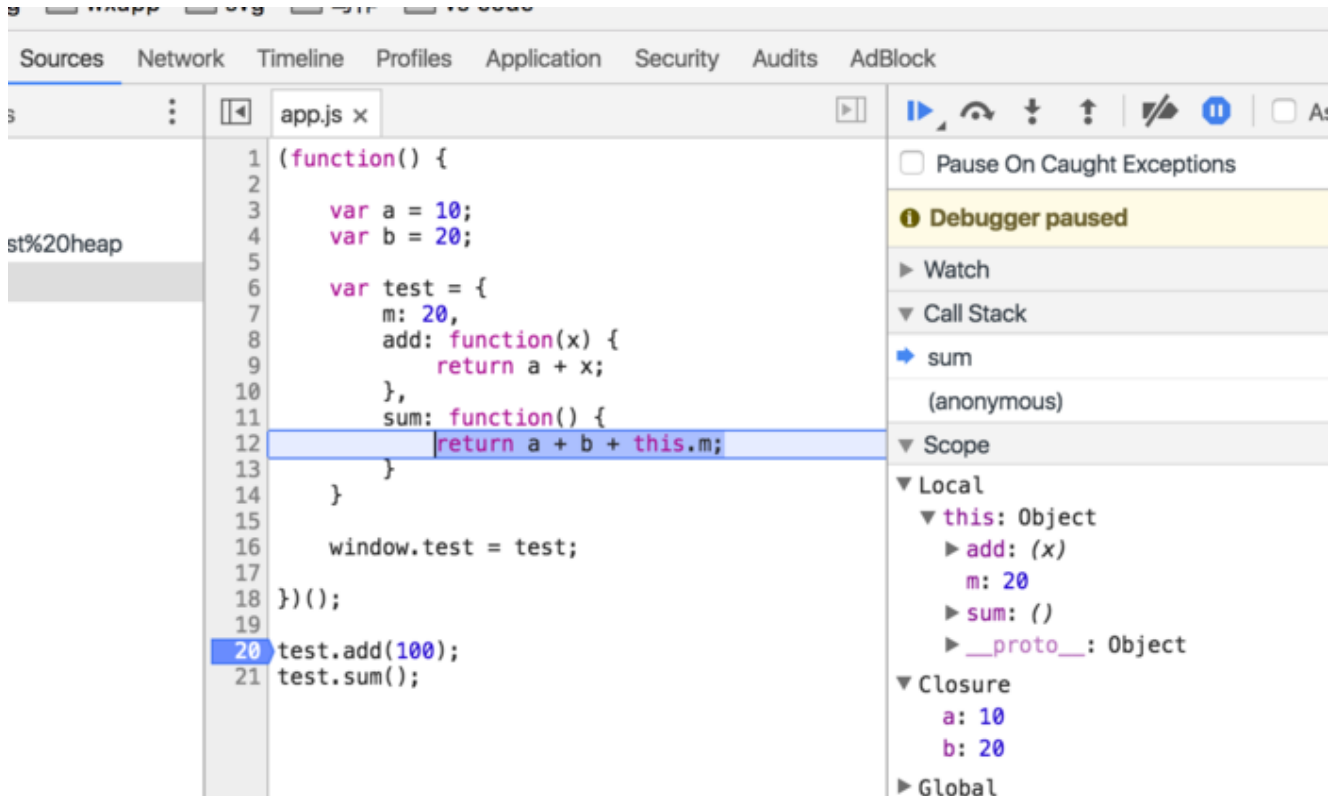
test.add(100);
test.sum();
test.mark();

var _mark = test.mark;
_mark();

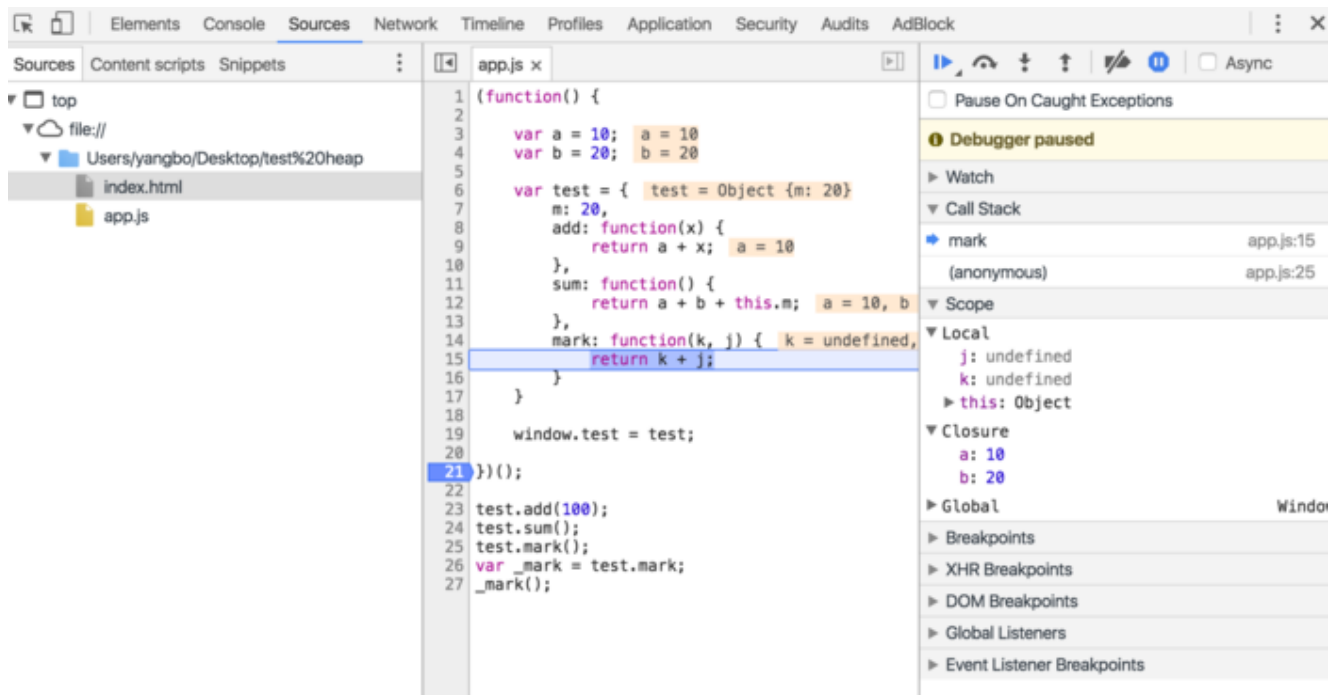
```



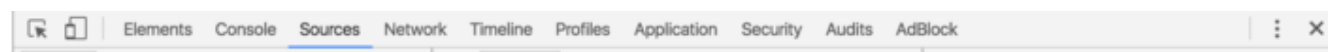
add执行时，闭包为外层的自执行函数，this指向test

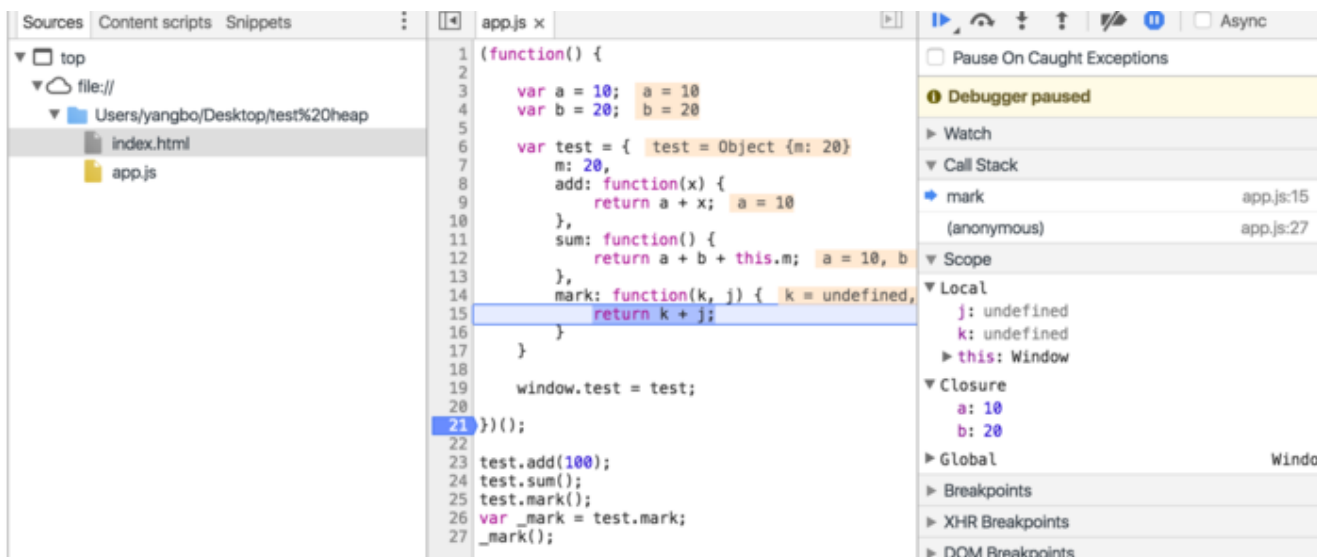


sum执行时，同上



mark执行时，闭包为外层的自执行函数，this指向test





\_mark执行时，闭包为外层的自执行函数，this指向window

注意：这里的this指向显示为Object或者Window，大写开头，他们表示的是实例的构造函数，实际上this是指向的具体实例

test.mark能形成闭包，跟下面的补充例子（demo07）情况是一样的。

我们还可以结合点断调试的方式，来理解那些困扰我们很久的this指向。随时观察this的指向，在实际开发调试中非常有用。

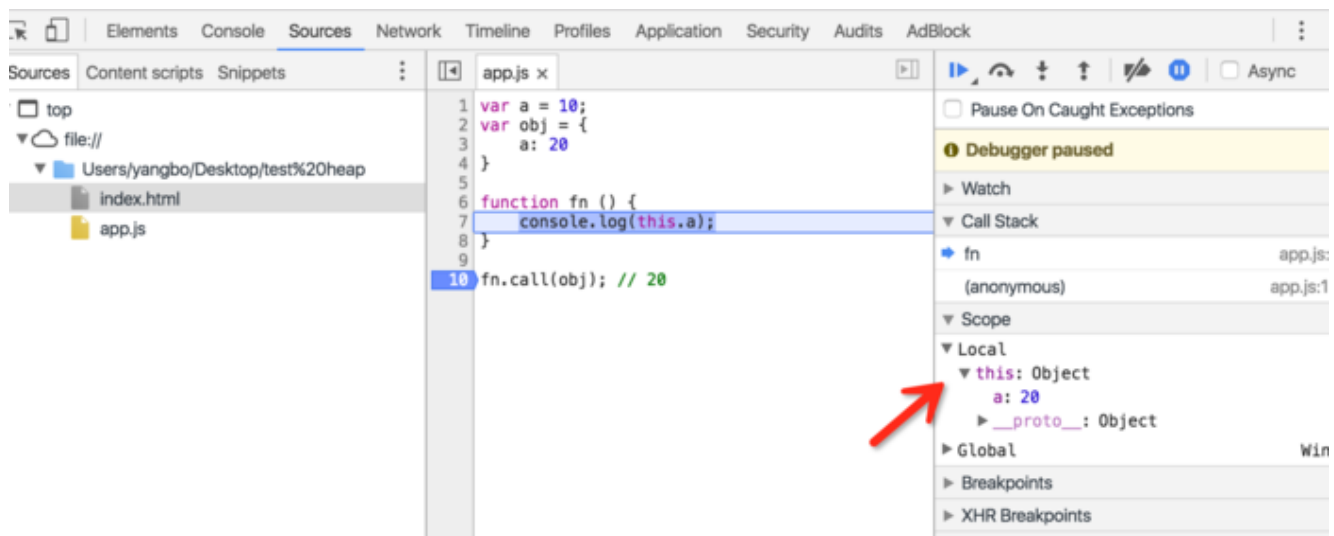
```
// demo06

var a = 10;

var obj = {
  a: 20
}

function fn () {
  console.log(this.a);
}
```

```
fn.call(obj); // 20
```



this指向obj

最后继续补充一个例子。

```
// demo07

function foo() {
  var a = 10;

  function fn1() {
    return a;
  }

  function fn2() {
    return 10;
  }

  fn2();
}
```

```
foo();
```

这个例子，和其他例子不太一样。虽然fn2并没有访问到foo的变量，但是foo执行时仍然变成了闭包。而当我将fn1的声明去掉时，闭包便不会出现了。

那么结合这个特殊的例子，我们可以这样定义闭包。

闭包是指这样的作用域(**foo**)，它包含有一个函数(**fn1**)，这个函数(**fn1**)可以调用被这个作用域所封闭的变量(**a**)、函数、或者闭包等内容。通常我们通过闭包所对应的函数来获得对闭包的访问。

更多的例子，大家可以自行尝试，总之，学会了使用断点调试之后，我们就能够很轻松的了解一段代码的执行过程了。这对快速定位错误，快速了解他人的代码都有非常巨大的帮助。大家一定要动手实践，把它给学会。

最后，根据以上的摸索情况，再次总结一下闭包：

- 闭包是在函数被调用执行的时候才被确认创建的。
- 闭包的形成，与作用域链的访问顺序有直接关系。
- 只有内部函数访问了上层作用域链中的变量对象时，才会形成闭包，因此，我们可以利用闭包来访问函数内部的变量。

大家也可以根据我提供的这个方法，对其他的例子进行更多的测试，如果发现我的结论有不对的地方，欢迎指出，大家相互学习进步，谢谢大家。

点喜欢赚钻 最高日赚数百元







等5人已打赏

赞赏支持

© 著作权归作者所有



## 乐城国际贸易城

乐城国际贸易城

### 精彩评论

写评论



不净莲华

3

个人理解，闭包就是在函数外部调用函数内部的函数，本来函数外部是无权访问函数内部的函数或者变量，但是通过外部变量引用或者返回函数方式使得函数能够在外部调用，在调用时执行上下文创建和执行过程中保留了变量对象和作用域链使其不会被垃圾回收的过程就是闭包

58楼 · 2018-10-15 10:59

打开App，查看全部评论

### 推荐阅读

更多精彩内容 >



下载简书App  
你也可以写文章赚赞赏

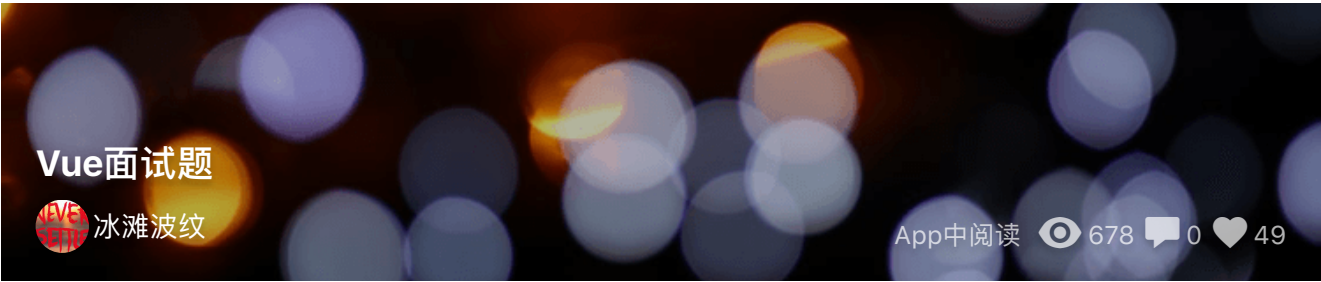
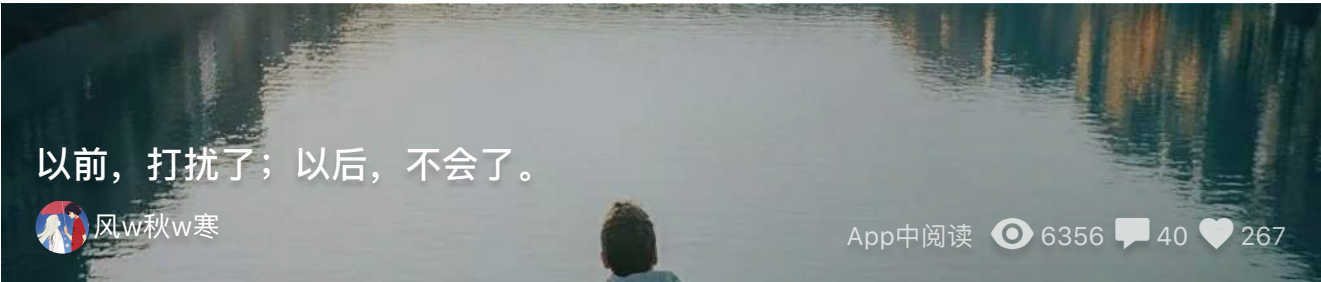


希望，就像钻石一样珍贵



11山山

App中阅读 4305 101 205







JS语法

元素JavaScript知识点梳理与经典百例

醉枫々染墨

控制语句

条件  
循环

函数function

App中阅读 518 0 43



我自学前端的，东学西学感觉很虚，请求指导

我是老尚

App中阅读 153 0 3



JavaScript防抖节流原理

持续触发

poetries

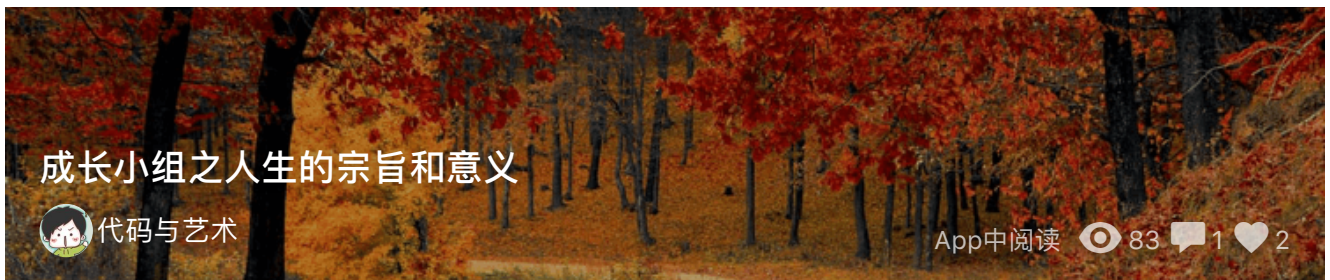
App中阅读 568 0 36



2019年成都web前端开发工资是多少

IT知了酱

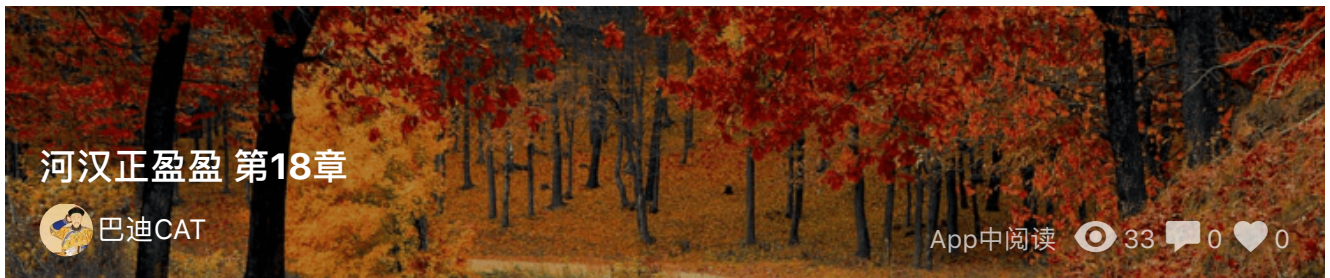
App中阅读 1564 6 1



成长小组之人生的宗旨和意义

代码与艺术

App中阅读 83 1 2

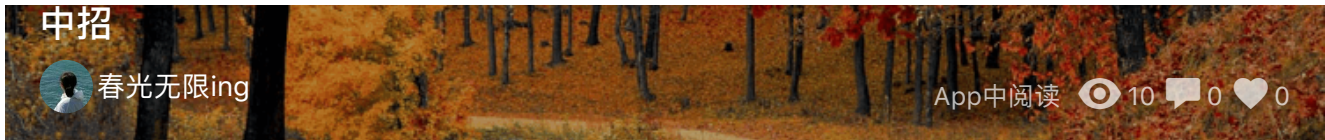


河汉正盈盈 第18章

巴迪CAT

App中阅读 33 0 0





创作你的创作，  
接受世界的赞赏



退出 | 打开App | 热门文章

 下载简书，创作你的创作