

浅谈匿名函数和闭包



就那ck (/u/f12aed2af88f) [+ 关注](#)

2017.07.25 18:48* 字数 2560 阅读 182 评论 2 喜欢 7

(/u/f12aed2af88f)

前言

相信很多前端小伙伴在工作和学习中，都会或多或少的接触和了解到**匿名函数**和**闭包**。也去网上搜索了不少的资料，查到资料 and 解释都各有说辞，甚至有些解释本身就是不正确的，这更加让人头疼。今天就来聊一聊**匿名函数**和**闭包**，浅谈一下他们之间的关系（实际上他们之间并没有什么直接关系！important）。

什么是匿名函数

与**匿名函数**相对应的是**具名函数**，具名函数非常简单：`function myFn(){}` ，这就是个具名函数这个函数的name是myFn。可以测试一下：

```
function myFn(){  
}  
console.log(myFn.name);//myFn
```

特别说明一下，函数表达式也是一种具名函数的定义方式。比如`var myFn1 = function() {}`，打印`myFn1.name`，也会得到`myFn1`。

再说匿名函数，一般用到匿名函数的时候都是立即执行的。通常叫做自执行匿名函数或者自调用匿名函数。常用来构建沙箱模式，作用是**开辟封闭的变量作用域环境**，在多人联合工作中，合并js代码后，不会出现相同变量互相冲突的问题。立即执行的匿名函数有很多种写法，常见的有以下两种：

```
(function(){  
  console.log("我是匿名方式1");  
})();//我是匿名方式1  
  
(function(){  
  console.log("我是匿名方式2");  
})();//我是匿名方式2  
  
console.log((function(){}).name);//'' name为空
```



两者的区别就是：一个是发起执行的括号在匿名函数括号的外面，另外一个发起执行的括号在匿名函数的里面。实际中的书写方式个人推荐第一种，这种写法更符合调用机制，调用时的参数也比较明显，如下：

```
(function(i,j,k){
  console.log(i+j+k);
})(1,3,5);
//9
```

还有其他一些自执行匿名函数的写法，如下：

```
-function(){
  console.log("我是匿名方式x");
}();
console.log(-function(){}.name); //-0
+function(){
  console.log("我是匿名方式x");
}();
console.log(+function(){}.name); //0
~function(){
  console.log("我是匿名方式x");
}();
console.log(~function(){}.name); //-1
!function(){
  console.log("我是匿名方式x");
}();
console.log(!function(){}.name); //true
void function(){
  console.log("我是匿名方式x");
}();
console.log(void function(){}.name); //undefined
```

这几种操作符，有时会影响结果的类型，不推荐使用，大家可以查下资料看看各种方式之间的差别。具名函数其实也可以立即执行，在此不做太多的伸展（本文主要目的是为了说明匿名函数和闭包之间的关系）。

实际上，**立即执行的匿名函数并不是函数**，因为已经执行过了，所以它是一个结果，这个结果是对当前这个匿名函数执行结果的一个引用（函数执行默认return undefined）。这个结果可以是一个字符串、数字或者null/false/true，也可以是对象、数组或者一个函数（对象和数组都可以包含函数），当返回的结果包含函数时，这个立即执行的匿名函数所返回的结果就是典型的闭包了。

闭包是怎么定义的，该如何理解

闭包本身定义比较抽象，MDN官方上解释是：***A closure is the combination of a function and the lexical environment within which that function was declared.***

中文解释是：**闭包是一个函数和该函数被定义时的词法环境的组合。**



很多地方可以看到一个说法：js中每个函数都是一个闭包，这样理解也是没有问题的，不过会增加对闭包的理解难度，这里先不这么理解，可以按照闭包起的作用来理解它：就是能在一个函数外部执行这个函数内部的定义方法，并访问内部的变量

在此，先看个经典的使用闭包的案例，实现在函数外部访问函数内部的局部变量：

```
function box(){
  var a = 10;
  function inner(){
    return a;
  }
  return inner;
}
var outer = box();
console.log(outer());//10
```

正常情况，box执行过后，会被回收机制回收所占用的内存，包括其内部定义的局部变量。但是此时box执行过后返回一个内部的函数inner，这个inner引用了内部的变量a，inner又被外部outer给接收，回收机制检查到内部的变量被引用，就不会执行回收。

但是看到这里，还是一脸蒙比，哪里使用了闭包？貌似有三个函数呀，一个box，一个inner还有一个outer = box()。

- 这个案例中用到的闭包其实是**inner和inner被定义时的词法环境**，这个闭包被return出来后被外部的outer引用，因此可以在box外部执行这个inner，inner能够读取到box内部的变量a。
- 使用这个闭包的目的是为了在box外部访问a，就是通过执行outer()。

用匿名函数实现闭包

上面的例子是在具名函数box内部用一个具名函数inner实现了闭包，那怎么使用匿名函数实现闭包呢，也很简单：



```
//第一步直把内部inner这个具名函数改为匿名函数并直接return， 结果同样是10
function box(){
  var a = 10;
  return function(){
    console.log(a) ;
  }
}
var outer = box();
outer();//10
//第二步把外部var outer = box()改成立即执行的匿名函数
var outer = (function(){
  var a=10;
  return function(){
    console.log(a);
  }
})();
//outer 作为立即执行匿名函数执行结果的一个接收，这个执行结果是闭包，outer等于这个闭包。
//执行outer就相当于执行了匿名函数内部return的闭包函数
//这个闭包函数可以访问到匿名函数内部的私有变量a，所以打印出10
outer();//10
```

这样我们就改写成了由匿名函数实现的闭包，真正使用到的闭包是内部的被return的函数和这个函数所定义时的环境。由此可以说明：**闭包跟函数是否匿名没有直接关系**，匿名函数和具名函数都可以创建闭包。

for循环的问题及解决方案

还有一个令人感到困惑，工作和学习中也经常遇见的问题是在for循环中：

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

我们希望打印出来0,1,2,3,4，然而打印出来的是5个5，很尴尬。什么原因引起的这问题呢？这是因为setTimeout的回调函数并不是立即执行的而是要等到循环结束才开始计时和执行（在此对运行机制不伸展），要说明的一点是**js中函数在执行前都只对变量保持引用**，并不会真正获取和保存变量的值。所以等循环结束后i的值是已经是5了，因此执行定时器的回调函数会打印出5个5。

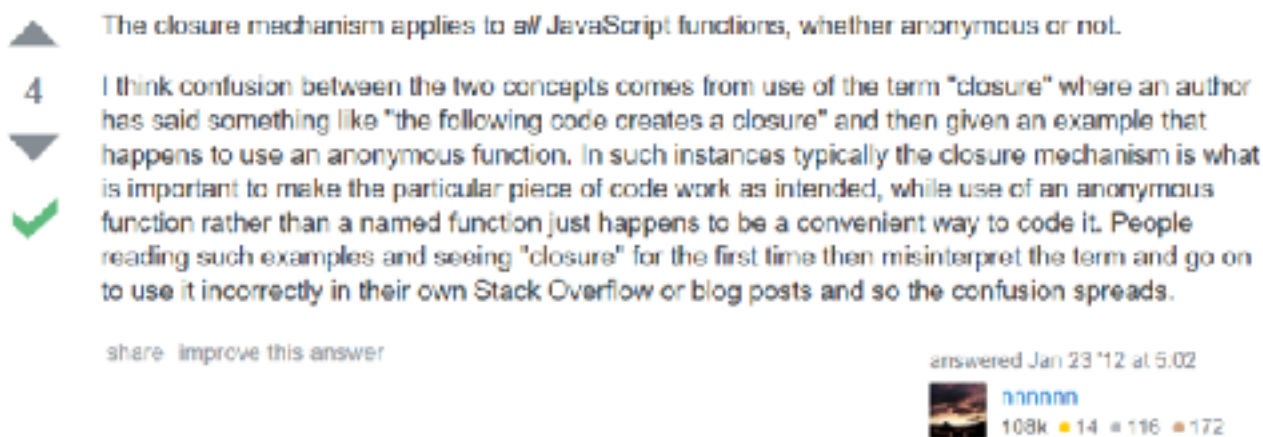
1) 怎么解决这个问题？

最常见的解决方法就是给定时器外面加一个立即执行的匿名函数，并把当前循环的i作为实参传入这个立即执行的匿名函数。如下：



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

可以得到预想的结果：0,1,2,3,4，此时很多人认为这个立即执行的匿名函数就是闭包，其实这么理解是错误的，然后在错误的理解之上又扩展了好多案例，导致其他人看后不知所措，一头雾水。附上一张Stack Overflow上一位同学的回答截图，我觉得他说的特别有道理：



image

原文地址：<https://stackoverflow.com/questions/8967214/what-is-the-difference-between-a-closure-and-an-anonymous-function-in-js> (<https://link.jianshu.com?t=https%3A%2F%2Fstackoverflow.com%2Fquestions%2F8967214%2Fwhat-is-the-difference-between-a-closure-and-an-anonymous-function-in-js>)。

2) 那到底这个for循环中的闭包是什么呢，其中的自执行匿名函数又起到什么作用呢？

我们可以试着把这个自执行的匿名函数改写为具名的函数，来测试下结果：

```
for(var i = 0;i<5;i++){
  function hasNameFn(i){
    setTimeout(function(){
      console.log(i);
    },100*i);
  };
  hasNameFn(i);
}
```



可以发现结果和使用匿名函数的结果是一样的，所以这里也可以说明闭包跟匿名函数没什么直接关系。

这个for循环中的闭包怎么理解以及自执行匿名函数的作用：

- 这个for循环产生的闭包其实是定时器的回调函数，**这些回调函数的执行环境是window**，类似刚才例子中的引用inner的全局outer的执行环境，匿名函数则相当于刚才例子中的box函数。
- 而自执行的匿名函数的作用也很简单：就是每一次循环创建一个私有词法环境，执行时把当前的循环的i传入，保存在这个词法环境中，这个i就类似上面box函数中var声明的a。
- 刚才有说到函数在被执行前都只是保存对变量的引用，**自执行的匿名函数正是因为执行了**，所以能够获取当前的变量i的值。因此定时器的回调函数在执行时引用的i就已经确定了具体的值。
- 或许我们改写一下，这么看就能更清晰明了一些：

```
for(var i = 0;i<5;i++){
  (function(j){
    setTimeout(function(){
      console.log(j);
    },100*j);
  })(i);
}
```

改写后的匿名函数形参用j来表示，匿名函数执行时传入实参i，此时定时器里面打印的其实是j，匿名函数立即执行，j的值也会确定。所以最后每次定时器的回调函数打印的结果也都是这个已经被匿名函数所确定的值。

3) 其他的解决方案

解决刚才for循环的问题，其实根本要解决的问题是如何让每次循环的定时器的回调函数引用当前的i，而不是循环结束后的i。

最简单的方法是使用es6 let，能够为变量创建块级作用域：




```
for(let i = 0;i<5;i++){
  setTimeout(function(){
    console.log(i);
  },100*i);
}
//改写成下面这么写更好理解一些
for(var i = 0;i<5;i++){
  let j = i;
  setTimeout(function(){
    console.log(j);
  },100*j);
}
```

还可以用bind绑定当前的i给定时器的回调函数（实际上bind方法内部还是实现了一个对调用者的柯里化闭包，并保存了执行时传入的参数给调用者）：

```
for(var i = 0;i<5;i++){
  setTimeout(function(i){
    console.log(i);
  }.bind(this,i),100*i);
}
```

可以得到跟使用立即执行函数同样的效果，所以说**匿名函数**和**闭包**之间并没有什么关系，只不过很多时候在用到匿名函数解决问题的时候恰好形成了一个闭包，就导致很多人分不清楚匿名函数和闭包的关系。

至此，关于匿名函数和闭包的关系，也聊的差不多了，希望能给那些对匿名函数和闭包比较迷惑的小伙伴一些帮助，同时文章中有不足的地方，也请大伙给予指出，一起学习进步！

 前端学习 (/nb/7223013)

[举报文章](#) © 著作权归作者所有

