# 我所理解的闭包

## 什么是闭包?

简单通俗（错误）来说，闭包就是一个函数，只不过这个函数是在另一个函数内声明的。这句话是不太严谨，但是此刻我们先这么理解就对了。

举个例子，通常我们声明函数是这样的：

```
function func1() {
    //...
}
```

如果一个函数是在另一个函数中声明的：

```
function func2() {
    function func3() {
        //...
    }
}
```

那么，这里的`func3`函数就是传说中的闭包。这个闭包`func3`跟正常声明的`func1`函数没有什么大区别，除了这一点： **func3函数可以访问func2函数的作用域** 。由于，嵌套的函数可以访问到其外层作用域中声明的变量，`func2`内部定义的函数`func3`当然可以访问其父函数`func2`的变量了。但是，**如果这个func3函数在func2以外也可以调用的话就厉害了**。先实现一下：

```
function func2() {
    function func3() {
        //...
    }
    // 函数是一等公民（普通对象），当然可以作为返回值了。
    return func3;
    // 还有别的办法可以把这个函数传递出去，比如在func2之前先声明一个变量，在这里将func3赋给它也是可以的。
}
```

然后再运行一下这个`func2()`函数，并把结果赋给一个变量`func4`

```
let func4 = func2()
```

这个`func4`就是`func3`，同一个东西的不同名字了。这样的话，我们就可以在`func2`以外通过`func4`来就能调用`func3`了。所以闭包是指那些能够可以**记忆**它被创建时候的环境(也就是不管走到哪里，都带着跟着它发家的那些小弟)的函数，也就是说：**闭包就是说在函数内部定义了一个函数，然后这个函数调用到了父函数内的相关临时变量，这些相关的临时变量就会存入闭包作用域里面，这个函数是有权访问父函数作用域中的变量的函数。**

## 闭包有什么用

先来段代码：

```
function func2 () {
  let a = 1

  function func3 () {
    let b = 2
    a = a + b
    console.log(a, b)
  }
  return func3
}
let func4 = func2()
func4() // 3, 2
func4() // 5, 2
```

`func2`函数已经执行完毕了，我们还是能通过`func4`函数访问并且修改`func2`函数中定义的变量`a`。

再来一个

```
function func2 (a) {
  function func3 (b) {
    console.log(a + b)
  }

  return func3
}
let func4 = func2(1)
let func5 = func2(2)
func4(100) // 101
func5(100) // 102
```

两次调用func2时传入不同的参数，调用返回的函数也就不同，返回的函数有什么不同呢？他们的访问到的变量a是不同的（对func4，a是1， 对func5，a是2），所以当我执行func4，func5时传入了同样的参数100，返回值却分别是101， 102，原因就在于这两个函数保存着的a是不同的。所以闭包的一个特性是： **保存临时变量**

## 为什么需要闭包呢

局部变量无法共享和长久的保存，而全局变量可能造成变量污染，所以我们希望有一种机制既可以长久的保存变量又不会造成全局污染。

## 利用闭包创建模块

其实这个闭包很有用，我们看以下代码：

```
function CoolModule () {
  var something = 'cool'
  var another = [1, 2, 3]

  function doSomething () {
    console.log(something)
  }

  function doAnother () {
    console.log(another.join('!'))
  }

  return {
    doSomething: doSomething,
    doAnother: doAnother
  }
}
var foo = CoolModule()
foo.doSomething() // cool
foo.doAnother() // 1 ! 2 ! 3
```

这个模式就被称为模块，最常见的实现模块模式的方法通常被称为*模块暴露*，这里展示其变体。

> 模块模式需要具备两个必要条件：
>
> 1. 必须有外部的封闭函数（`CoolModule()`），该函数必须至少被调用一次（每次调用都会创建一个新的模块实例）。
> 2. 封闭函数必须返回至少一个内部函数，这样内部函数才能在私有作用域中形成闭包，并且可以访问或者修改私有的状态。

以前也偷偷地就用了闭包了，只是当时还没这么叫：

```
function say (message) {
  setTimeout(function timeoutHandler () {
    console.log(message)
  }, 1000)
}

say('我以前用过闭包?嗯，这个就是！')
```

这里的`timeoutHandler`函数就是我们神奇的闭包——它是在`say`函数中定义的，它在`say`函数执行了一秒以后才执行的，但是依然可以访问到`say`函数的变量`message`。只是通常见到的这个函数都是匿名函数，不过这个跟它匿名不匿名没有什么关系。

在`for`循环中也可以使用闭包，比如从1到5，每隔一秒打印一个数字：

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function timeoutHandler () {
    console.log(i)
  }, i * 1000)
}
```

这个是可以顺利实现的，因为ES2015之后，在花括号(块级作用域)中定义一个函数，也可以形成闭包，`timeoutHandler`虽然不是在函数中定义的，但是它在实例化(循环5次就相当于实例化了5个timeoutHandler函数，也就是把它的名字复制给了5个完全不同的变量)时也是形成了闭包。每一个timeoutHandler访问到的i是实例化时的i，因为这个for循环基本上等同于：

```
{
  let i = 1

  function timeoutHandler () {
    console.log(i)
  }

  setTimeout(timeoutHandler, i * 1000)
}
{
  let i = 2

  function timeoutHandler () {
    console.log(i)
  }

  setTimeout(timeoutHandler, i * 1000)
}
{
  let i = 3

  function timeoutHandler () {
    console.log(i)
  }

  setTimeout(timeoutHandler, i * 1000)
}
{
  let i = 4

  function timeoutHandler () {
    console.log(i)
  }
  setTimeout(timeoutHandler, i * 1000)
}
{
  let i = 5
```

```
  function timeoutHandler () {
    console.log(i)
  }
  setTimeout(timeoutHandler, i * 1000)
}
```

这么看来就明白了吧。所以最后的结论：**闭包就是一种特殊的函数，它可以访问到定义它的作用域内的所有变量，即使是在别的地方调用。**