闭包，不管你平时开发用不用得上，但是面试极大概率会被问到。对于闭包，笔者自身也看了很多资料，对闭包概念很容易产生云里雾里的感觉。所以，本文不会死记硬背式的讲解闭包概念，而是更侧重于讲解闭包的作用，从运用角度切入去了解熟悉闭包。

## 外部作用域读取函数内部变量

看个例子：

function f1() {

var n = 999;

}

console.log(n)// Uncaught ReferenceError: n is not defined

在该例子中，在全局作用域中，试图打印f1函数内部变量n，结果报错了。于是我们调整下，在函数内部再定义多一个函数。

function f1() {

var n = 999;

function f2() {

console.log(n); // 999

}

}

我们在函数f1内部定义多了个f2函数，由于之前讲解过的作用域链的关系，f2可以读取在f1中的变量。那么只要我们把f2作为返回值，就可以实现**外部作用域读取函数内部变量，即在f1外部读取到f1内部的变量了！**

## 闭包是什么

返回函数的代码：

function **f1**() {

var n = 20;

function **f2**() {

return n

}

return bar;

}

var bar = foo();

bar();

**f1函数嵌套f2函数，并且f2函数读取f1函数中的变量n，这样就形成了闭包。**

**关于闭包概念的描述定义有挺多个版本的：**

《你不知道的Javascript上卷》版本：

当函数可以记住并访问所在的词法作用域，即使函数是在当前词法作用域之外执行，这就产生了闭包。

该版本理解，对于上述例子，f1函数和f2以及f2访问n变量行为就形成了闭包。

《javascript高级程序设计》版本：

闭包是指有权访问另外一个函数作用域中的变量的函数。

该版本理解，对于上述例子，f2函数就是闭包。

MDN版本：

闭包是指那些能够访问自由变量的函数。

该版本理解，对于上述例子，f2函数就是闭包。

这么多个版本，是不是看懵了！简而言之，一般情况下访问一个函数作用域中的变量，外部正常是无法访问的，只能通过特殊的技术或者特性来实现，就是在函数作用域中创建内部函数来实现，这样就不会使得函数执行完成后函数定义的变量被回收，这种现象或者特性应该被称为“闭包”，像是《JavaScript权威指南》打的比方，像是把变量包裹了起来，形象的称为“闭包”。

## 闭包的形成

简单的讲，闭包是一种现象或者特性，其形成需要两点：

1. **函数嵌套**
2. **内部函数引用了外部函数的局部变量**

再看个例子巩固下：

var fn = null;

function foo() {

var a = 2;

function innnerFoo() {

console.log(a);

}

fn = innnerFoo; // 将 innnerFoo的引用，赋值给全局变量中的fn

}

function bar() {

fn(); // 此处的保留的innerFoo的引用

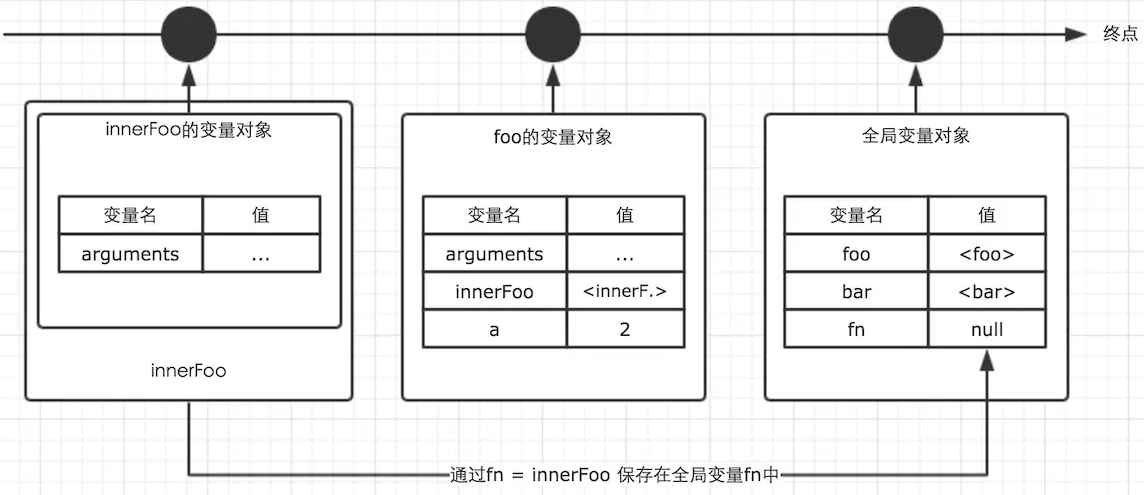
}

foo();

bar(); // 2

上面的例子，foo()执行完之后，该函数中的变量对象占用的内存应该被垃圾收集器释放，但是由于fn=innerFoo，为foo函数内部的innerFoo函数引用被保留下来，赋值给了全局变量fn。由于存在引用关系，foo的变量对象也被保留下来。在执行bar时，fn仍然可以访问foo函数变量对象，所以a的值能被打印。这样也会形成闭包。

该闭包的作用域链如下图：



值的注意的是，闭包的产生并不会使得作用域链发送任何改变，也就是作用域的规则不会改变。

调整上述例子，如果我们在函数bar中声明一个变量c，并在闭包fn中试图访问该变量，运行结果会抛出错误：

var fn = null;

function foo() {

var a = 2;

function innnerFoo() {

console.log(c); // 在这里，试图访问函数bar中的c变量，会抛出错误

console.log(a);

}

fn = innnerFoo; // 将 innnerFoo的引用，赋值给全局变量中的fn

}

function bar() {

var c = 100;

fn(); // 此处的保留的innerFoo的引用

}

foo();

bar();

为什么会抛错？这就是之前文章提到的作用域知识，函数的作用域是在声明时定义的，不是在调用执行时决定的。fn()其实就是innerFoo函数的引用调用，本质还是调用innerFoo，innerFoo是在foo函数中定义的，所以他只能访问自身和foo函数的变量对象。

## 闭包的运用

经过上述的讲述，我们可以总结出闭包有以下作用：

1. **能够访问到函数的内部变量**
2. **可以使得被访问变量保存在内存中，即使变量所处的函数已经执行完（双刃剑，提供变量访问便利的同时，也容易造成内存泄漏，造成性能问题）**

利用这两个闭包作用，在我们实际开发中，我们可以运用闭包来**实现js模块化。**

具体请看下面的例子:

//index.html文件

<script type="text/javascript" src="myModule.js"></script>

<script type="text/javascript">

myModule2.doSomething()

myModule2.doOtherthing()

</script>

//myModule.js文件

(function () {

var msg = 'Beijing'//私有数据

//操作数据的函数

function doSomething() {

console.log('doSomething() '+msg.toUpperCase())

}

function doOtherthing () {

console.log('doOtherthing() '+msg.toLowerCase())

}

//向外暴露对象(给外部使用的两个方法)

window.myModule2 = {

doSomething: doSomething,

doOtherthing: doOtherthing

}

})()

在该例子中，将函数doSomething、doOtherthing 和变量msg 都封装在匿名自执行函数中，只向外暴露了myModule2 对象，其拥有了doSomething、doOtherthing两个方法。这是只需要调用该对象对应函数就能实现对应功能，并且变量msg的值也一直保存在内存中。这样看来匿名自执行函数包裹的就相遇一个模块。

另外的运用是使用闭包去解决循环定义事件带来的问题。

示例：

<button>测试1</button>

<button>测试2</button>

<button>测试3</button>

<script type="text/javascript">

var btns = document.getElementsByTagName('button')

for (var i = 0; i < btns.length; i++) {

btns[i].onclick = function () {

console.log('第' + (i + 1) + '个')

}

}

</script>

万万没想到，点击任意一个按钮，后台都是弹出“第四个”,这是因为i是全局变量,执行到点击事件时，此时i的值为3。那该如何修改，最简单的是用let声明i

for (let i = 0; i < btns.length; i++) {

btns[i].onclick = function () {

console.log('第' + (i + 1) + '个')

}

}

另外我们可以通过闭包的方式来修改:

for (var i = 0; i < btns.length; i++) {

(function (j) {

btns[j].onclick = function () {

console.log('第' + (j + 1) + '个')

}

})(i)

}

## 闭包的内存泄漏

所谓内存泄漏指任何对象在您不再拥有或需要它之后仍然存在。

在之前的文章，我们讲解过javascript的自动垃圾回收机制，那么对于全局和函数作用域什么时候会释放内存呢？

* 全局作用域----只有当页面关闭的时候全局作用域才会销毁
* 私有的作用域----只有函数执行才会产生

一般情况下，新的私有的作用域在函数执行时形成，当私有作用域中的代码执行完成后，该作用域都会主动的进行释放和销毁内存。但当遇到函数执行返回了一个引用数据类型的值，并且在函数的外面被一个其他的东西给接收了，这种情况下一般形成的私有作用域都不会销毁。

如下面这种情况：

function fn(){

var num=100;

return function(){

Console.log(num)

}

}

var f=fn();//fn执行形成的这个私有的作用域就不能再销毁了

也就是像上面这段代码，fn函数内部的私有作用域会被一直占用的，发生了内存泄漏。闭包不能滥用，否则会导致内存泄露，影响网页的性能。闭包使用完了后，要立即释放资源，将引用变量指向null。

接下来我们看下有关于内存泄漏的一道经典面试题：

function outer(){

var num=0;//内部变量

return function add(){//通过return返回add函数，就可以在outer函数外访问了

num++;//内部函数有引用，作为add函数的一部分了

console.log(num);

};

}

var func1=outer();

func1();//实际上是调用add函数， 输出1

func1();//输出2 因为outer函数内部的私有作用域会一直被占用

var func2=outer();

func2();// 输出1 每次重新引用函数的时候，闭包是全新的。

func2();// 输出2