## 熟练掌握闭包

理解闭包只需要学会三个基本的事实。

第一个事实：js允许你引用在当前函数以外定义的变量。

对于这一句话，我最开始被误导，直到我亲自做实验猜得出了结论，结论就是我的理解是错误的。 我的错误想法是js函数使用的外部变量的方式是引用，把外部变量作为了一个引用类型。但是事实上，这句话的意思只是简单的说明了你可以沿着作用域链搜索外部变量，并使用外部变量而已。

以下是我的试验：

既然要判断是不是真的作为引用处理，这很简单

var i = 0;

var arr = [];

(function a(){

arr.push(i);

*console*.log(arr);

})();

i++;

(function a(){

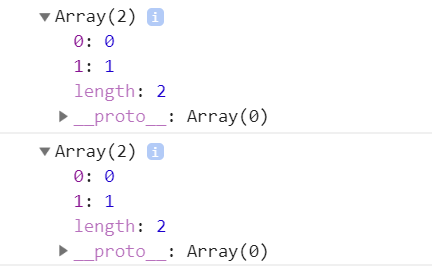
arr.push(i);

*console*.log(arr);

})();

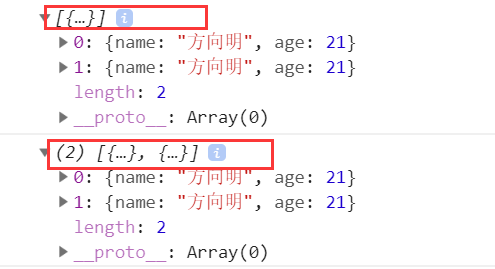
// 如果输出第二次输出01则非引用，是使用变量的值

// 如果是11，则是将外部变量作为引用类型，处理方式和对象一样



结论是[0,1]，所以结论得出，外部变量将不会被按照引用类型处理。只是简单的利用了外部变量的值而已（如果外部变量本身就是引用类型，就还是按照引用类型处理）。相当于是外部变量的一个拷贝。

为了验证这一点，只需要把外部变量换成一个真真正正的引用类型即可。这里换成一个数组：



可以看出，后者数组中，第0和第1的对象都是最新的状态，有两个属性，而不是第0的对象只有Name属性。

值得注意的是，我框红框框的地方，尽管第一个console的结果只显示有一个{...}，但是事实上，console的也只是i那时候的“闪照”,至于下面的信息（没有被红框框住的部分），则是直接引用的当前数组的最新状态，并不是‘闪照’。

话回正题，接着谈谈闭包。

下面这个例子其实我认为并不是闭包，因为内部的函数在外部函数返回的时候就已经被调用了，返回的只是一个结果而并不是一个函数指针，之后就会被清除内存，也就是不会在跟踪外部函数中的相关变量了。

function makeSandwich(){

var magicIngredient = "peanut butter";

function make(*filling*){

return magicIngredient + "and" + filling;

}

return make("jelly");

}

var sandwich = makeSandwich();

*console*.log(sandwich); // peanut butter and jelly

第二个事实：即使外部函数已经返回，当前函数仍然可以引用在外部函数所定义的变量。

function makeSandwich(){

var magicIngredient = "peanut butter";

function make(*filling*){

return magicIngredient + "and" + filling;

}

return make;

}

var sandwich = makeSandwich();

*console*.log(sandwich("jelly")); // peanut butter and jelly

*console*.log(sandwich("egg")); // peanut butter and jelly

这个例子与上一个例子几乎完全相同。但是不同的是，不是在外部的makeSandwich函数中立即调用了make函数，而是返回了make函数的指针。

但是这是如何工作的呢？ 事实上，js函数值包含了比调用他们时执行所需要的代码还要多的信息（这部分细节会在高程的笔记中提到）。而且，js函数值还在内部储存他们可能会引用的定义在其封闭作用域的变量。**那些在其所涵盖的作用域内跟踪变量的函数被称为闭包（我认为说白了就是函数内作用域包含外部函数的变量的，而且还没有在外部函数内被立即调用，是作为一个函数指针被外部返回的）。**所以，在这里make函数就是一个闭包。他在被外部函数返回的时候并没有立即得到返回，也就是里面引用的变量还没有发挥作用（**闭包对外部函数的变量是引用**），因此不能在外部函数返回的时候立即销毁外部函数。

那么问题来了，外部函数返回的时候并没有执行内部函数，只是返回一个函数的指针？他是如何知道内部函数到底有没有引用外部变量的呢？

我的猜想是，当js函数在声明的时候就会为每个涉及到的变量开辟内存空间，知道函数声明完毕，得知函数的所占内存空间后结束声明。然后在执行js函数的时候进行在判断是否有逻辑错误或者语法错误等。所以在声明的时候应该就知道是否引用了外部变量了。

详情有待探索。。。

闭包是js中最优雅、最有表现力的特性之一，也是许多惯有法的核心。Js甚至还提供了一种更为方便的方式创建闭包——函数表达式。

function makeSandwich(*filling*){

var magicIngredient = "peanut butter";

return function(){

return magicIngredient + "and" + filling;

};

}

*console*.log(makeSandwich("jelly")()); // peanut butter and jelly

*console*.log(makeSandwich("egg")()); // peanut butter and egg

因为我们不打算让内部函数在外部函数内调用，因此我们不必给内部函数起名，直接返回内部函数的表达式即可。

第三个事实是：闭包可以更新外部变量的值。实际上，闭包储存的是外部变量的引用，而不是他们的副本。因此，我们可以借助闭包来间接访问外部函数的私有变量。

function myFun(){

var val = "方向明";

return {

set:function(*newVal*){

val = newVal;

return "修改成功";

},

get:function(){

return val;

},

type:function(){

return typeof val;

}

}

}

var change = myFun();

*console*.log(change.set("白雪欣")); // 修改成功

*console*.log(change.get()) // 白雪新

*console*.log(change.type()) // string

小结：

函数可以引用定义在其外部作用域的变量

闭包比创建他们的函数有更长的声明周期。外部函数返回，但是闭包的函数还是能引用外部函数的值。

闭包在内部储存其外部变量的引用，并能读写这些变量。因此可以借助闭包来访问和修改函数的私有变量。