你真的了解 JavaScript 中的闭包么？

现在去面试前端开发的岗位，如果你对面试官也是个前端，并且不是太水的话，你有很大的概率会被问到 JavaScript 中的闭包。因为这个闭包这个知识点真的很重要，还非常难掌握。

什么是闭包

什么是闭包，你可能会搜出很多答案....

《JavaScript 高级程序设计》这样描述：

闭包是指有权访问另一个函数作用域中的变量的函数；

《JavaScript 权威指南》这样描述：

从技术的角度讲，所有的 JavaScript 函数都是闭包：它们都是对象，它们都关联到作用域链。

《你不知道的 JavaScript》这样描述：

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

我最认同的是《你不知道的 JavaScript》中的描述，虽然前面的两种说法都没有错，但闭包应该是基于词法作用域书写代码时产生的自然结果，是一种现象！你也不用为了利用闭包而特意的创建，因为闭包的在你的代码中随处可见，只是你还不知道当时你写的那一段代码其实就产生了闭包。

讲解闭包

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

看一段代码

```
function fn1() {
        var name = 'duyi';
        function fn2() {
                console.log(name);
        }
        fn2();
```

```
}
fn1();
```

如果是根据《JavaScript 高级程序设计》和《JavaScript 权威指南》来说，上面的代码已经产生闭包了。fn2 访问到了 fn1 的变量，满足了条件“有权访问另一个函数作用域中的变量的函数”，fn2 本身是个函数，所以满足了条件“所有的 JavaScript 函数都是闭包”。

这的确是闭包，但是这种方式定义的闭包不太好观察。

再看一段代码：

```
function fn1() {
        var name = 'duyi';
        function fn2() {
                console.log(name);
        }
        return fn2;
}
var fn3 = fn1();
fn3();
```

这样就清晰地展示了闭包：

- fn2 的词法作用域能访问 fn1 的作用域
- 将 fn2 当做一个值返回
- fn1 执行后，将 fn2 的引用赋值给 fn3
- 执行 fn3，输出了变量 name

我们知道通过引用的关系，fn3 就是 fn2 函数本身。执行 fn3 能正常输出 name，这不就是 fn2 能记住并访问它所在的词法作用域，而且 fn2 函数的运行还是在当前词法作用域之外了。

正常来说，当 fn1 函数执行完毕之后，其作用域是会被销毁的，然后垃圾回收器会释放那段内存空间。而闭包却很神奇的将 fn1 的作用域存活了下来，**fn2 依然持有该作用域的引用，这个引用就是闭包**。

总结：**某个函数在定义时的词法作用域之外的地方被调用，闭包可以使该函数极限访问定义时的词法作用域**。

注意：对函数值的传递可以通过其他的方式，并不一定值有返回该函数这一条路，比如可以用回调函数：

```
function fn1() {
        var name = 'duyi';
        function fn2() {
                console.log(name);
        }
        fn3(fn2);
}
function fn3(fn) {
        fn();
}
fn1();
```

本例中，将内部函数 fn2 传递给 fn3，当它在 fn3 中被运行时，它是可以访问到 name 变量的。

所以无论通过哪种方式将内部的函数传递到所在的词法作用域以外，它都回持有对原始作用域的引用，无论在何处执行这个函数都会使用闭包。

再次解释闭包

以上的例子会让人觉得有点学院派了，但是闭包绝不仅仅是一个无用的概念，你写过的代码当中肯定有闭包的身影，比如类似如下的代码：

```
function waitSomeTime(msg, time) {
        setTimeout(function () {
                console.log(msg)
        }, time);
}
waitSomeTime('hello', 1000);
```

定时器中有一个匿名函数，该匿名函数就有涵盖 waitSomeTime 函数作用域的闭包，因此当 1 秒之后，该匿名函数能输出 msg。

另一个很经典的例子就是 for 循环中使用定时器延迟打印的问题：

```
for (var i = 1; i <= 10; i++) {
        setTimeout(function () {
                console.log(i);
        }, 1000);
}
```

在这段代码中，我们对其的预期是输出 1~10，但却输出 10 次 11。这是因为 setTimeout 中的匿名函数执行的时候，for 循环都已经结束了，for 循环结束的条件是 i 大于 10，所以当然是输出 10 次 11 咯。

究其原因：i 是声明在全局作用中的，定时器中的匿名函数也是执行在全局作用域中，那当然是每次都输出 11 了。

原因知道了，解决起来就简单了，我们可以让 i 在每次迭代的时候，都产生一个私有的作用域，在这个私有的作用域中保存当前 i 的值。

```
for (var i = 1; i <= 10; i++) {
        (function () {
                var j = i;
                setTimeout(function () {
                        console.log(j);
                }, 1000);
        })();
}
```

这样就达到我们的预期了呀，让我们用一种比较优雅的写法改造一些，将每次迭代的 i 作为实参传递给自执行函数，自执行函数中用变量去接收：

```
for (var i = 1; i <= 10; i++) {
        (function (j) {
                setTimeout(function () {
                        console.log(j);
                }, 1000);
        })(i);
}
```

闭包的应用

闭包的应用比较典型是定义模块，我们将操作函数暴露给外部，而细节隐藏在模块内部：

```
function module() {
        var arr = [];
        function add(val) {
                if (typeof val == 'number') {
                        arr.push(val);
                }
        }
```

```
        function get(index) {
                if (index < arr.length) {
                        return arr[index]
                } else {
                        return null;
                }
        }
        return {
                add: add,
                get: get
        }
}
var mod1 = module();
mod1.add(1);
mod1.add(2);
mod1.add('xxx');
console.log(mod1.get(2));
```