你真的了解 JavaScript 中的作用域么？

这两天刚好和朋友讨论到闭包，这个 JavaScript 中的“神兽”，很多同学会觉得闭包这玩意太闹心了，怎么着都理解不了...其实刚接触 JavaScript 的时候我也是这样的。

但是呢，闭包却非常重要！非常重要！非常重要！ 在《你不知道的 JavaScript》中甚至这样写道“**对于那些有一点 JavaScript 使用经验但从未真正理解闭包概念的人来说，理解闭包可以看作是某种意义上的重生**”。

所以看到这里，各位亲是不是迫切的想要深入的去了解一下闭包了呢？ 不急，不急，对于真正的理解闭包有一个非常重要的前置知识，那就是**作用域与词法作用域**，如果你没能好好理解词法作用域，那么闭包是肯定理解不了的！那么接下来就好好的理解一下词法作用域吧。

作用域

我们先抛出一个概念：“**词法作用域是作用域的一种工作模型**”，先不管这句话的深层次的意思，就但看表面，我们就应该可以得出一个结论，那就是没有作用域的概念就没有词法作用域的概念。所以...接下来，你懂的...

什么是作用域

一言以蔽之，“**作用域就是一套规则，用于确定在何处以及如何查找变量（标识符）的规则**”。在这句话中读到一个关键点 *查找变量（标识符）*，那么就从查找变量说起吧。

先看一段及其简单的代码

```
function foo() {
        var a = '';
        console.log(a); // 输出""
}
foo();
```

在 foo 函数执行的时候，输出一个 a 变量，那么这个 a 变量是哪里来的嘞，有看到函数第一行有定义 a 变量的代码 var a = ''。

再看一段同样简单的代码

```
var b = 'programmer';
```

```
function foo() {
        console.log(b); // 输出"programmer"
}
foo();
```

同样的道理，在输出 b 的时候，自己函数内部没有找到变量 b，那么就在外层的全局中查找，找到了就停止查找并输出了。

注意以上两段代码都有查找变量，第一段代码是在**函数**中找到 a 变量，第二段代码是在**全局**中找到 b 变量。现在闭上眼睛，我要给加粗的这两个词的后面加上几个字了！

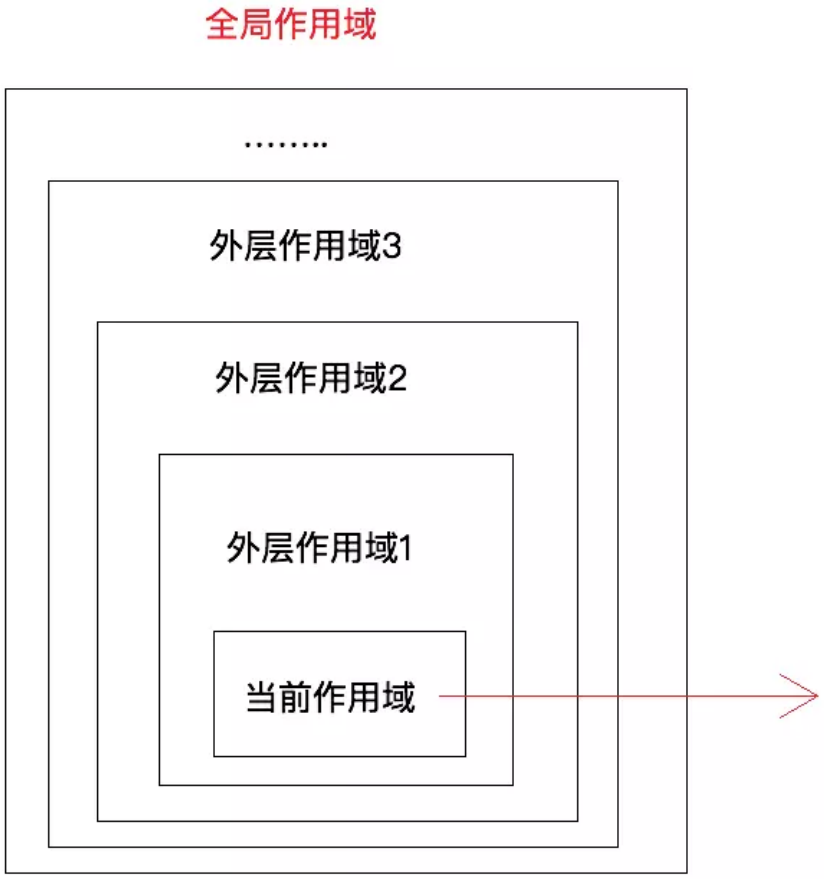
好了，打开眼睛，Duang，Duang --->**函数作用域**、**全局作用域**，把这两个词换入到原来那句话中，第一段代码是在**函数作用域**中找到 a 变量，第二段代码是在**全局作用域**中找到 b 变量。

所以，懂了没有呢？通俗的讲，**作用域就是查找变量的地方**。在某函数中找到该变量，就可以说在该函数作用域中找到了该变量；在全局中找到该变量，就可以说在全局作用域中找到了该变量！

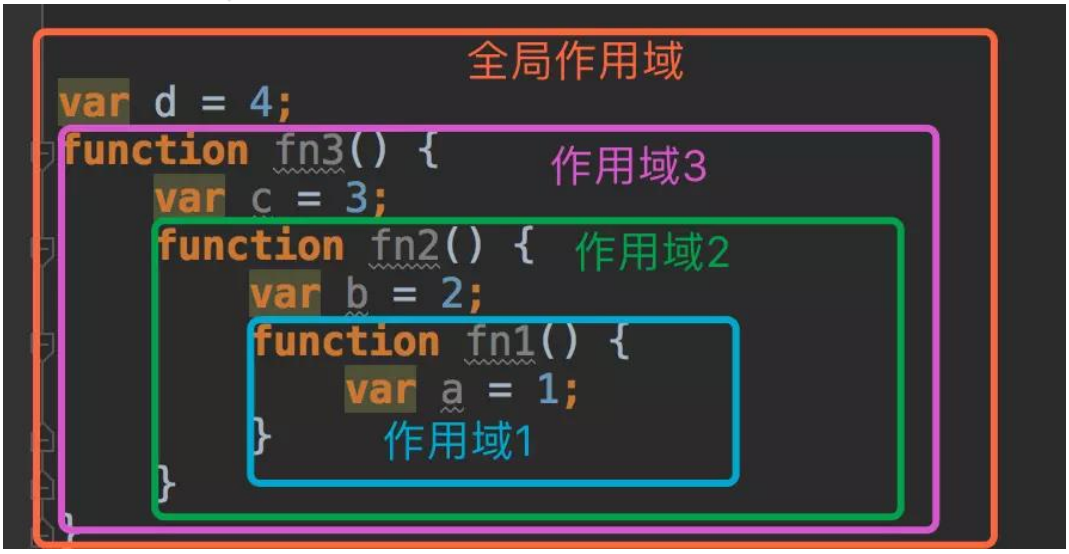
不知道各位同学有没注意到一个细节，我们在查找 b 变量的时候，先在函数作用域中查找，没有找到，再去全局作用域中查找，有一个往外层查找的过程。我们好像是顺着一条链条从下往上查找变量，这条链条，我们就称之为**作用域链**。

作用域嵌套

在还没有接触到 ES6 的 let、const 之前，只有函数作用域和全局作用域，函数作用域肯定是在全局作用域里面的，而函数作用域中又可以继续嵌套函数作用域，如图：



用代码表示：



以上两张图可以很直观的看出作用域的嵌套关系了吧。查找变量也是顺着红色的箭头走的，从里到外，这从里到外的各层作用域就组成了作用域链。

作用域中变量（标识符）的查找规则

首先声明一点，JavaScript 是有编译过程的，不要惊讶，真的有！也就是说 var name = 'iceman'这段代码，其实这是有两个动作的：

- 编译器在当前作用域中声明一个变量 name
- 运行时引擎在作用域中查找该变量，找到了 name 变量并为其赋值

证明以上的说法：

```
console.log(name); // 输出 undefined
var name = 'iceman';
```

在 var name = 'iceman'的上一行输出 name 变量，并没有报错，输出 undefined，说明输出的时候该变量已经存在了，只是没有赋值而已。

其实编译器是这样工作的，在代码执行之前从上到下的进行编译，当遇到某个用 var 声明的变量的时候，先检查在当前作用域下是否存在了该变量。如果存在，则忽略这个声明；如果不存在，则在当前作用域中声明该变量。

上面的这段简单的代码包含两种查找类型：输出变量的值的时候的查找类型是 RHS，找到变量为其赋值的查找类型是 LHS。

我猜各位同学一定可以猜到“L”和“R”的含义，这里的左侧和右侧指的是在赋值操作的左侧和右侧。也就是说，变量出现在赋值操作的左侧时进行 LHS 查询，出现在右侧时进行 RHS 查询。

用一句通俗的话来讲，RHS 就是取到它的源值。

注意：“赋值操作的左侧和右侧”，并不意味着只是“=”，实际上赋值操作还有好几种形式。

在作用域中查找变量都是 RHS，并且查找的规则是从当前作用域开始找，如果没找到再到父级作用域中找，一层层往外找，如果在全局作用域如果还没找到的话，就会报错了：**ReferenceError: 某变量 is not defined**

所有的赋值操作中查找变量都是 LHS。其中 a=4 这类赋值操作，也是会从当前作用域中查找，如果没有找到再到外层作用域中找，如果到全局变量啊这个变量，在非严格模式下会创建一个全局变量 a。不过，非常不建议这么做，因为轻则污染全局变量，重则造成内存泄漏（比如：a = 一个非常大的数组，a 在全局变量中，一直用有引用，程序不会自动将其销毁）。

词法作用域

在上面的作用域介绍中，我们将作用域定义为一套规则，这套规则来管理浏览器引擎如何在当前作用域以及嵌套的作用域中根据变量（标识符）进行变量查找。

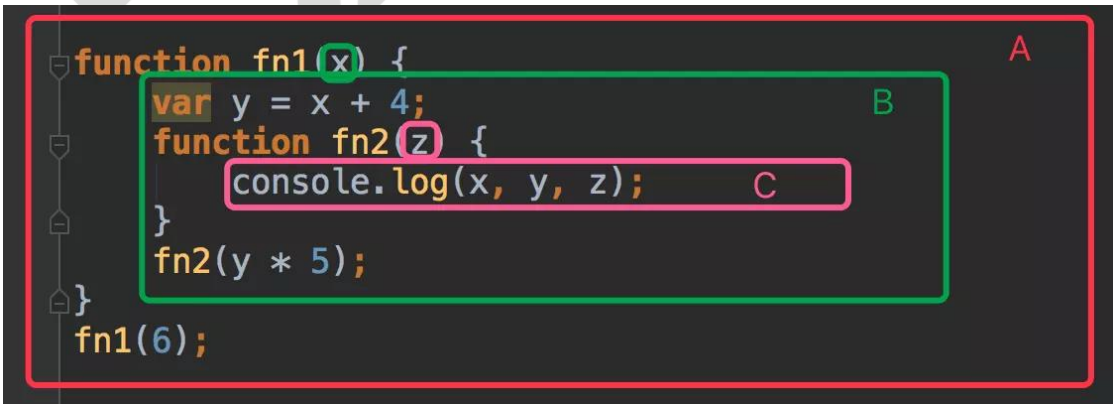
我们在前面有抛出一个概念：“词法作用域是作用域的一种工作模型”，作用域有两种工作模型，在 JavaScript 中的词法作用域是比较主流的一种，另一种动态作用域（比较少的语言在用）。

所谓的词法作用域就是**在你写代码时将变量和块作用域写在哪里来决定，也就是词法作用域是静态的作用域，在你书写代码时就确定了**。

请看以下代码：

```
function fn1(x) {
        var y = x + 4;
        function fn2(z) {
                console.log(x, y, z);
        }
        fn2(y * 5);
}
fn1(6); // 6 10 50
```

这个例子中有个三个嵌套的作用域，如图：



- A 为全局作用域，有一个标识符：fn1
- B 为 fn1 所创建的作用域，有三个标识符：x、y、fn2
- C 为 fn2 所创建的作用域，有一个标识符：z

作用域是由期代码写在哪里决定的，并且是逐级包含的。

在此强调，词法作用域就是作用域是由书写代码时函数声明的位置来决定的。编译阶段就能够知道全部标识符在哪里以及是如何声明的，所以词法作用域是

静态的作用域，也就是词法作用域能够预测在执行代码的过程中如何查找标识符。

注 1：eval()和 with 可以通过其特殊性用来“欺骗”词法作用域，不过正常情况下都不建议使用，会产生性能问题。

注 2：ES6 中有了 let、const 就有了块级作用域。