闭包，是 javascript 中独有的一个概念，对于初学者来讲，闭包是一个特别抽象的概念，特别是 ECMA 规范给的定义，如果没有实战经验，你很难从定义去理解它。因此，本文不会对闭包的概念进行大篇幅描述，直接上干货，让你分分钟学会闭包！

1 闭包--爱的初体验

在接触一个新技术的时候，我首先会做的一件事就是：找它的 demo code。对于码农们来说，代码有时候比自然语言更能理解一个事物。 其实，闭包无处不在，比如：jQuery、zepto 的主要代码都包含在一个大的闭包中，所以下面我先写一个最简单最原始的闭包 demo，好让你在大脑里 产生闭包的画面：

```
function A(){
    function B(){
       console.log("Hello Closure!");
    }
    return B;
}
var b = A();
b();//Hello Closure!
```

这是史上最简单的闭包，不能再简单了，再简单就不是闭包了！

有了初步的认识后，我们简单分析一下它和普通函数有什么不同，这样我们才能从“茫茫人海”中一眼认出“她”。

上面代码翻译成自然语言如下：

(1)定义了一个普通函数 A

(2)在 A 中定义了普通函数 B

(3)在 A 中返回 B

(4)执行 A(),把 A 的返回结果赋值给变量 b

(5)执行 b()

把这 5 步操作总结成一句扯淡的话就是：

函数 A 的内部函数 B 被函数 A 外的一个变量 b 引用

把这句扯淡的话再加工一下就变成了闭包的定义：

当一个内部函数被其外部函数之外的变量引用时，就形成了一个闭包。

不要刻意去记住这个定义，我告诉你这个定义的目的是想让你理解上面的 5 步操作就是在阐述闭包的定义。

因此，当你执行了上述 5 步操作的时候，你就已经定义了一个闭包！

这就是闭包。

2 闭包的作用

在了解闭包的作用之前，我们先了解一下 javascript 中的 GC 机制:在 javascript 中，如果一个对象不再被引用，那么这个对象就会被 GC 回收，否则这个对象一直会保存在内存中。

在上述例子中，B 定义在 A 中，因此 B 依赖于 A,而外部变量 b 又引用了 B, 所以 A 间接的被 b 引用，也就是说，A 不会被 GC 回收，会一直保存在内存中。为了证明我们的推理，上面的例子稍作改进：

```
function A(){
    var count = 0;
    function B(){
       count ++;
       console.log(count);
    }
    return B;
}
var b = A();
b();// 1
b();// 2
b();// 3
```

count 是 A 中的一个变量，它的值在 B 中被改变，B 每执行一次，count 的值就在原来的基础上累加 1。因此，A 中的 count 一直保存在内存中。

这就是闭包的作用，有时候我们需要一个模块中定义这样一个变量：希望这个变量一直保存在内存中但又不会“污染”全局的变量，这个时候，我们就可以用闭包来定义这个模块。

3 高端写法

上面的写法其实是最简单最原始的写法，而在实际应用中，没人这么玩，特别是在一些大型 JS 框架中更不会这么写。我之所以还要告诉你这种写法，是因为干扰因素越少越容易专注于一件事。下面我用常用的写法来写一个简单的 demo 组件：

```
(function(document){
    var viewport;
    var obj = {
        init:function(id){
           viewport = document.querySelector("#"+id);
        },
        addChild:function(child){
            viewport.appendChild(child);
        },
        removeChild:function(child){
            viewport.removeChild(child);
        }
    }
    window.jView = obj;
})(document);
```

这个组件的作用是：初始化一个容器，然后可以给这个容器添加子容器，也可以移除一个容器。功能很简单，但这里涉及到了另外一个概念：立即执行函数。 简单了解一下就行。主要是要理解这种写法是怎么实现闭包功能的。

可以将上面的代码结构分成两部分：(function(){})()　　红色部分是一个表达式，而这个表达式本身是一个匿名函数，所以在这个表达式后面加()就表示执行这个匿名函数。

因此这段代码执行执行过程可以分解如下：

```
var f = function(document){
    var viewport;
    var obj = {
        init:function(id){
            viewport = document.querySelector("#"+id);
        },
        addChild:function(child){
            viewport.appendChild(child);
        },
        removeChild:function(child){
            viewport.removeChild(child);
        }
    }
    window.jView = obj;
};
f(document);
```

在这段代码中似乎看到了闭包的影子，但 f 中没有任何返回值，似乎不具备闭包的条件，注意这句代码：

```
window.jView = obj;
```

jView 是在 f 中定义的一个对象，这个对象中定义了一系列方法， 执行 window.jView = obj 就是在 window 全局对象定义了一个变量 jView，并将这个变量指向 obj 对象，即全局变量 jView 引用了 obj. 而 obj 对象中的函数又引用了 f 中的变量 viewport,因此 f 中的 viewport 不会被 GC 回收，会一直保存到内存中，所以这种写法满足闭包的条件。

4 简单的总结语

这是对闭包最简单的理解,当然闭包还有其更深层次的理解,这个就涉及的多了,你需要了解 JS 的执行环境(execution context)、活动对象(call object)以及作用域(scope)和作用域链(scope chain)的运行机制。但作为一个初学者，暂时不必了解这些，有了简单的理解之后，一定要在实际项目中用起来，等你用的多了，对于闭包，你自然会有更深 层次的理解!