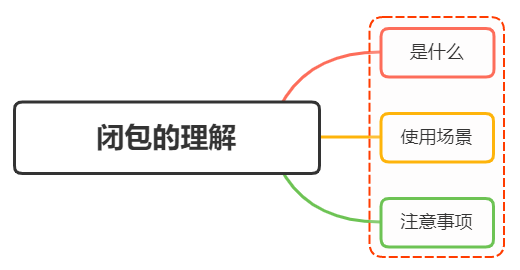
# 面试官：说说你对闭包的理解？闭包使用场景



## 一、是什么

一个函数和对其周围状态（lexical environment，词法环境）的引用捆绑在一起（或者说函数被引用包围），这样的组合就是闭包（closure）

也就是说，闭包让你可以在一个内层函数中访问到其外层函数的作用域

在 JavaScript中，每当创建一个函数，闭包就会在函数创建的同时被创建出来，作为函数内部与外部连接起来的一座桥梁

下面给出一个简单的例子

function init() {  
 var name = "Mozilla"; // name 是一个被 init 创建的局部变量  
 function displayName() { // displayName() 是内部函数，一个闭包  
 alert(name); // 使用了父函数中声明的变量  
 }  
 displayName();  
}  
init();

displayName() 没有自己的局部变量。然而，由于闭包的特性，它可以访问到外部函数的变量

## 二、使用场景

任何闭包的使用场景都离不开这两点：

* 创建私有变量
* 延长变量的生命周期

一般函数的词法环境在函数返回后就被销毁，但是闭包会保存对创建时所在词法环境的引用，即便创建时所在的执行上下文被销毁，但创建时所在词法环境依然存在，以达到延长变量的生命周期的目的

下面举个例子：

在页面上添加一些可以调整字号的按钮

function makeSizer(size) {  
 return function() {  
 document.body.style.fontSize = size + 'px';  
 };  
}  
  
var size12 = makeSizer(12);  
var size14 = makeSizer(14);  
var size16 = makeSizer(16);  
  
document.getElementById('size-12').onclick = size12;  
document.getElementById('size-14').onclick = size14;  
document.getElementById('size-16').onclick = size16;

### 柯里化函数

柯里化的目的在于避免频繁调用具有相同参数函数的同时，又能够轻松的重用

// 假设我们有一个求长方形面积的函数  
function getArea(width, height) {  
 return width \* height  
}  
// 如果我们碰到的长方形的宽老是10  
const area1 = getArea(10, 20)  
const area2 = getArea(10, 30)  
const area3 = getArea(10, 40)  
  
// 我们可以使用闭包柯里化这个计算面积的函数  
function getArea(width) {  
 return height => {  
 return width \* height  
 }  
}  
  
const getTenWidthArea = getArea(10)  
// 之后碰到宽度为10的长方形就可以这样计算面积  
const area1 = getTenWidthArea(20)  
  
// 而且如果遇到宽度偶尔变化也可以轻松复用  
const getTwentyWidthArea = getArea(20)

### 使用闭包模拟私有方法

在JavaScript中，没有支持声明私有变量，但我们可以使用闭包来模拟私有方法

下面举个例子：

var Counter = (function() {  
 var privateCounter = 0;  
 function changeBy(val) {  
 privateCounter += val;  
 }  
 return {  
 increment: function() {  
 changeBy(1);  
 },  
 decrement: function() {  
 changeBy(-1);  
 },  
 value: function() {  
 return privateCounter;  
 }  
 }  
})();  
  
var Counter1 = makeCounter();  
var Counter2 = makeCounter();  
console.log(Counter1.value()); /\* logs 0 \*/  
Counter1.increment();  
Counter1.increment();  
console.log(Counter1.value()); /\* logs 2 \*/  
Counter1.decrement();  
console.log(Counter1.value()); /\* logs 1 \*/  
console.log(Counter2.value()); /\* logs 0 \*/

上述通过使用闭包来定义公共函数，并令其可以访问私有函数和变量，这种方式也叫模块方式

两个计数器 Counter1 和 Counter2 是维护它们各自的独立性的，每次调用其中一个计数器时，通过改变这个变量的值，会改变这个闭包的词法环境，不会影响另一个闭包中的变量

### 其他

例如计数器、延迟调用、回调等闭包的应用，其核心思想还是创建私有变量和延长变量的生命周期

## 三、注意事项

如果不是某些特定任务需要使用闭包，在其它函数中创建函数是不明智的，因为闭包在处理速度和内存消耗方面对脚本性能具有负面影响

例如，在创建新的对象或者类时，方法通常应该关联于对象的原型，而不是定义到对象的构造器中。

原因在于每个对象的创建，方法都会被重新赋值

function MyObject(name, message) {  
 this.name = name.toString();  
 this.message = message.toString();  
 this.getName = function() {  
 return this.name;  
 };  
  
 this.getMessage = function() {  
 return this.message;  
 };  
}

上面的代码中，我们并没有利用到闭包的好处，因此可以避免使用闭包。修改成如下：

function MyObject(name, message) {  
 this.name = name.toString();  
 this.message = message.toString();  
}  
MyObject.prototype.getName = function() {  
 return this.name;  
};  
MyObject.prototype.getMessage = function() {  
 return this.message;  
};