时的词法作用域，因此它也可以如预期般访问变量 a。

这个函数在定义时的词法作用域以外的地方被调用。闭包使得函数可以继续访问定义时的词法作用域。

当然，无论使用何种方式对函数类型的值*进行传递*，当函数在别处被调用时都可以观察到闭包。

```
function foo() {
    var a = 2;

    function baz() {
        console.log( a ); // 2
    }

    bar( baz );
}

function bar(fn) {
    fn(); // 妈妈快看呀，这就是闭包!
}
```

把内部函数 baz 传递给 bar，当调用这个内部函数时（现在叫作 fn），它涵盖的 foo() 内部作用域的闭包就可以观察到了，因为它能够访问 a。

传递函数当然也可以是间接的。

```
var fn;

function foo() {
    var a = 2;

    function baz() {
        console.log( a );
    }

    fn = baz; // 将 baz 分配给全局变量
}

function bar() {
    fn(); // 妈妈快看呀，这就是闭包!
}

foo();

bar(); // 2
```

无论通过何种手段将内部函数*传递*到所在的词法作用域以外，它都会持有对原始定义作用域的引用，无论在何处执行这个函数都会使用闭包。

5.3 现在我懂了

前面的代码片段有点死板，并且为了解释*如何使用闭包*而人为地在结构上进行了修饰。但我保证闭包绝不仅仅是一个好玩的玩具。你已经写过的代码中一定到处都是闭包的身影。现在让我们来*搞懂*这个事实。

```
function wait(message) {

    setTimeout( function timer() {
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

将一个内部函数（名为 `timer`）传递给 `setTimeout(..)`。`timer` 具有涵盖 `wait(..)` 作用域的闭包，因此还保有对变量 `message` 的引用。

`wait(..)` 执行 1000 毫秒后，它的内部作用域并不会消失，`timer` 函数依然保有 `wait(..)` 作用域的闭包。

深入到引擎的内部原理中，内置的工具函数 `setTimeout(..)` 持有对一个参数的引用，这个参数也许叫作 `fn` 或者 `func`，或者其他类似的名字。引擎会调用这个函数，在例子中就是内部的 `timer` 函数，而词法作用域在这个过程中保持完整。

这就是闭包。

或者，如果你很熟悉 jQuery（或者其他能说明这个问题的 JavaScript 框架），可以思考下面的代码：

```
function setupBot(name, selector) {
    $( selector ).click( function activator() {
        console.log( "Activating: " + name );
    } );
}

setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

我不知道你会写什么样的代码，但是我写的代码负责控制由闭包机器人组成的整个全球无人机大军，这是完全可以实现的！

玩笑开完了，本质上*无论何时何地*，如果将函数（访问它们各自的词法作用域）当作第一级的值类型并到处传递，你就会看到闭包在这些函数中的应用。在定时器、事件监听器、Ajax 请求、跨窗口通信、Web Workers 或者任何其他的异步（或者同步）任务中，只要使用了*回调函数*，实际上就是在使用闭包！

第 3 章介绍了 IIFE 模式。通常认为 IIFE 是典型的闭包例子，但根据先前对闭包的定义，我并不是很同意这个观点。

```
var a = 2;

(function IIFE() {
    console.log( a );
})();
```

虽然这段代码可以正常工作，但严格来讲它并不是闭包。为什么？因为函数（示例代码中的 IIFE）并不是在它本身的词法作用域以外执行的。它在定义时所在的作用域中执行（而外部作用域，也就是全局作用域也持有 a）。a 是通过普通的词法作用域查找而非闭包被发现的。

尽管技术上来讲，闭包是发生在定义时的，但并不非常明显，就好像六祖慧能所说：“既非风动，亦非幡动，仁者心动耳。”[5]。

尽管 IIFE 本身并不是观察闭包的恰当例子，但它的确创建了闭包，并且也是最常用来创建可以被封闭起来的闭包的工具。因此 IIFE 的确同闭包息息相关，即使本身并不会真的使用闭包。

亲爱的读者，现在把书放下，我有一个任务要给你。打开你最近写的 JavaScript 代码，找到其中的函数类型的值并指出哪里已经使用了闭包，即使你以前可能并不知道这就是闭包。

等你呦！

现在你懂了吧！

5.4 循环和闭包

要说明闭包，for 循环是最常见的例子。

```
for (var i=1; i<=5; i++) {
    setTimeout( function timer() {
        console.log( i );
    }, i*1000 );
}
```

注 5：原文为 it’s a tree falling in the forest with no one around to hear it，同六祖慧能的风幡之动禅喻近义，比喻客观存在和观察认知之间的关系。——译者注

由于很多开发者对闭包的概念认识得并不是很清楚，因此当循环内部包含函数定义时，代码格式检查器经常发出警告。我们在这里介绍如何才能正确地使用闭包并发挥它的威力，但是代码格式检查器并没有那么灵敏，它会假设你并不真正了解自己在做什么，所以无论如何都会发出警告。

正常情况下，我们对这段代码行为的预期是分别输出数字 1~5，每秒一次，每次一个。

但实际上，这段代码在运行时会以每秒一次的频率输出五次 6。

这是为什么？

首先解释 6 是从哪里来的。这个循环的终止条件是 `i` 不再 `<=5`。条件首次成立时 `i` 的值是 6。因此，输出显示的是循环结束时 `i` 的最终值。

仔细想一下，这好像又是显而易见的，延迟函数的回调会在循环结束时才执行。事实上，当定时器运行时即使每个迭代中执行的是 `setTimeout(.., 0)`，所有的回调函数依然是在循环结束后才会被执行，因此会每次输出一个 `6` 出来。

这里引伸出一个更深入的问题，代码中到底有什么缺陷导致它的行为同语义所暗示的不一致呢？

缺陷是我们试图假设循环中的每个迭代在运行时都会给自己“捕获”一个 `i` 的副本。但是根据作用域的工作原理，实际情况是尽管循环中的五个函数是在各个迭代中分别定义的，但是它们都被封闭在一个共享的全局作用域中，因此实际上只有一个 `i`。

这样说的话，当然所有函数共享一个 `i` 的引用。循环结构让我们误以为背后还有更复杂的机制在起作用，但实际上没有。如果将延迟函数的回调重复定义五次，完全不使用循环，那它同这段代码是完全等价的。

下面回到正题。缺陷是什么？我们需要更多的闭包作用域，特别是在循环的过程中每个迭代都需要一个闭包作用域。

第 3 章介绍过，IIFE 会通过声明并立即执行一个函数来创建作用域。

我们来试一下：

```
for (var i=1; i<=5; i++) {
    (function() {
        setTimeout( function timer() {
            console.log( i );
        }, i*1000 );
    })();
}
```

这样能行吗？试试吧，我等着你。

我不卖关子了。这样不行。但是为什么呢？我们现在显然拥有更多的词法作用域了。的确每个延迟函数都会将 IIFE 在每次迭代中创建的作用域封闭起来。

如果作用域是空的，那么仅仅将它们进行封闭是不够的。仔细看一下，我们的 IIFE 只是一个什么都没有的空作用域。它需要包含一点实质内容才能为我们所用。

它需要有自己的变量，用来在每个迭代中储存 i 的值：

```
for (var i=1; i<=5; i++) {
    (function() {
        var j = i;
        setTimeout( function timer() {
            console.log( j );
        }, j*1000 );
    })();
}
```

行了！它能正常工作了！。

可以对这段代码进行一些改进：

```
for (var i=1; i<=5; i++) {
    (function(j) {
        setTimeout( function timer() {
            console.log( j );
        }, j*1000 );
    })( i );
}
```

当然，这些 IIFE 也不过就是函数，因此我们可以将 i 传递进去，如果愿意的话可以将变量名定为 j，当然也可以还叫作 i。无论如何这段代码现在可以工作了。

在迭代内使用 IIFE 会为每个迭代都生成一个新的作用域，使得延迟函数的回调可以将新的作用域封闭在每个迭代内部，每个迭代中都会含有一个具有正确值的变量供我们访问。

问题解决啦！

重返块作用域

仔细思考我们对前面的解决方案的分析。我们使用 IIFE 在每次迭代时都创建一个新的作用域。换句话说，每次迭代我们都需要一个块作用域。第 3 章介绍了 let 声明，可以用来劫持块作用域，并且在这个块作用域中声明一个变量。

本质上这是将一个块转换成一个可以被关闭的作用域。因此，下面这些看起来很酷的代码就可以正常运行了：

```
for (var i=1; i<=5; i++) {
    let j = i; // 是的，闭包的块作用域！
    setTimeout( function timer() {
        console.log( j );
    }, j*1000 );
}
```

但是，这还不是全部！（我用 Bob Barker[6] 的声音说道）for 循环头部的 let 声明还会有一个特殊的行为。这个行为指出变量在循环过程中不止被声明一次，每次迭代都会声明。随后的每个迭代都会使用上一个迭代结束时的值来初始化这个变量。

```
for (let i=1; i<=5; i++) {
    setTimeout( function timer() {
        console.log( i );
    }, i*1000 );
}
```

很酷是吧？块作用域和闭包联手便可天下无敌。不知道你是什么情况，反正这个功能让我成为了一名快乐的 JavaScript 程序员。

5.5 模块

还有其他的代码模式利用闭包的强大威力，但从表面上看，它们似乎与回调无关。下面一起来研究其中最强大的一个：模块。

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

正如在这段代码中所看到的，这里并没有明显的闭包，只有两个私有数据变量 something 和 another，以及 doSomething() 和 doAnother() 两个内部函数，它们的词法作用域（而这就是闭包）也就是 foo() 的内部作用域。

接下来考虑以下代码：

```
function CoolModule() {
    var something = "cool";
```

注 6：Bob Barker 是美国著名的电视节目主持人。——译者注

```
        var another = [1, 2, 3];

        function doSomething() {
            console.log( something );
        }

        function doAnother() {
            console.log( another.join( " ! " ) );
        }

        return {
            doSomething: doSomething,
            doAnother: doAnother
        };
    }

    var foo = CoolModule();

    foo.doSomething(); // cool
    foo.doAnother(); // 1 ! 2 ! 3
```

这个模式在 JavaScript 中被称为*模块*。最常见的实现模块模式的方法通常被称为*模块暴露*，这里展示的是其变体。

我们仔细研究一下这些代码。

首先，`CoolModule()` 只是一个函数，必须要通过*调用它*来创建一个模块实例。如果不执行外部函数，内部作用域和闭包都无法被创建。

其次，`CoolModule()` 返回一个用对象字面量语法 `{ key: value, ... }` 来表示的对象。这个返回的对象中含有对内部函数而*不是*内部数据变量的引用。我们保持内部数据变量是隐藏且私有的状态。可以将这个对象类型的返回值看作本质上是*模块的公共* API。

这个对象类型的返回值最终被赋值给外部的变量 `foo`，然后就可以通过它来访问 API 中的属性方法，比如 `foo.doSomething()`。

从模块中返回一个实际的对象并不是必须的，也可以直接返回一个内部函数。jQuery 就是一个很好的例子。`jQuery` 和 `$` 标识符就是 jQuery 模块的公共 API，但它们本身都是函数（由于函数也是对象，它们本身也可以拥有属性）。

`doSomething()` 和 `doAnother()` 函数具有涵盖模块实例内部作用域的闭包（通过调用 `CoolModule()` 实现）。当通过返回一个含有属性引用的对象的方式来将函数传递到词法作用域外部时，我们已经创造了可以观察和实践闭包的条件。

如果要更简单的描述，模块模式需要具备两个必要条件。

1. 必须有外部的封闭函数，该函数必须至少被调用一次（每次调用都会创建一个新的模块实例）。
2. 封闭函数必须返回至少一个内部函数，这样内部函数才能在私有作用域中形成闭包，并且可以访问或者修改私有的状态。

一个具有函数属性的对象本身并不是真正的模块。从方便观察的角度看，一个从函数调用所返回的，只有数据属性而没有闭包函数的对象并不是真正的模块。

上一个示例代码中有一个叫作 `CoolModule()` 的独立的模块创建器，可以被调用任意多次，每次调用都会创建一个新的模块实例。当只需要一个实例时，可以对这个模式进行简单的改进来实现单例模式：

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

我们将模块函数转换成了 IIFE（参见第 3 章），立即调用这个函数并将返回值直接赋值给单例的模块实例标识符 `foo`。

模块也是普通的函数，因此可以接受参数：

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );
```

```
foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"
```

模块模式另一个简单但强大的变化用法是，命名将要作为公共 API 返回的对象：

```
var foo = (function CoolModule(id) {
    function change() {
        // 修改公共 API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }

    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE
```

通过在模块实例的内部保留对公共 API 对象的内部引用，可以从内部对模块实例进行修改，包括添加或删除方法和属性，以及修改它们的值。

5.5.1 现代的模块机制

大多数模块依赖加载器 / 管理器本质上都是将这种模块定义封装进一个友好的 API。这里并不会研究某个具体的库，为了宏观了解我会简单地介绍一些核心概念：

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply( impl, deps );
    }

    function get(name) {
        return modules[name];
    }
```

```
        return {
            define: define,
            get: get
        };
    })();
```

这段代码的核心是 `modules[name] = impl.apply(impl, deps)`。为了模块的定义引入了包装函数（可以传入任何依赖），并且将返回值，也就是模块的 API，储存在一个根据名字来管理的模块列表中。

下面展示了如何使用它来定义模块：

```
MyModules.define( "bar", [], function() {
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar) {
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
var foo = MyModules.get( "foo" );

console.log(
    bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

"foo" 和 "bar" 模块都是通过一个返回公共 API 的函数来定义的。"foo" 甚至接受 "bar" 的示例作为依赖参数，并能相应地使用它。

为我们自己着想，应该多花一点时间来研究这些示例代码并完全理解闭包的作用吧。最重要的是要理解模块管理器没有任何特殊的“魔力”。它们符合前面列出的模块模式的两个特点：为函数定义引入包装函数，并保证它的返回值和模块的 API 保持一致。

换句话说，模块就是模块，即使在它们外层加上一个友好的包装工具也不会发生任何变化。