```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // 包含在 if 中的块作用域常量

    a = 3; // 正常！
    b = 4; // 错误！
}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

3.5 小结

函数是 JavaScript 中最常见的作用域单元。本质上，声明在一个函数内部的变量或函数会在所处的作用域中“隐藏”起来，这是有意为之的良好软件的设计原则。

但函数不是唯一的作用域单元。块作用域指的是变量和函数不仅可以属于所处的作用域，也可以属于某个代码块（通常指 `{ .. }` 内部）。

从 ES3 开始，`try/catch` 结构在 `catch` 分句中具有块作用域。

在 ES6 中引入了 `let` 关键字（`var` 关键字的表亲），用来在任意代码块中声明变量。`if (..) { let a = 2; }` 会声明一个劫持了 `if` 的 `{ .. }` 块的变量，并且将变量添加到这个块中。

有些人认为块作用域不应该完全作为函数作用域的替代方案。两种功能应该同时存在，开发者可以并且也应该根据需要选择使用何种作用域，创造可读、可维护的优良代码。

第 4 章

提升

到现在为止，你应该已经很熟悉作用域的概念，以及根据声明的位置和方式将变量分配给作用域的相关原理了。函数作用域和块作用域的行为是一样的，可以总结为：任何声明在某个作用域内的变量，都将附属于这个作用域。

但是作用域同其中的变量声明出现的位置有某种微妙的联系，而这个细节正是我们将要讨论的内容。

4.1 先有鸡还是先有蛋

直觉上会认为 JavaScript 代码在执行时是由上到下一行一行执行的。但实际上这并不完全正确，有一种特殊情况会导致这个假设是错误的。

考虑以下代码：

```
a = 2;

var a;

console.log( a );
```

你认为 `console.log(..)` 声明会输出什么呢？

很多开发者会认为是 `undefined`，因为 `var a` 声明在 `a = 2` 之后，他们自然而然地认为变量被重新赋值了，因此会被赋予默认值 `undefined`。但是，真正的输出结果是 `2`。

考虑另外一段代码：

```
console.log( a );

var a = 2;
```

鉴于上一个代码片段所表现出来的某种非自上而下的行为特点，你可能会认为这个代码片段也会有同样的行为而输出 2。还有人可能会认为，由于变量 a 在使用前没有先进行声明，因此会抛出 ReferenceError 异常。

不幸的是两种猜测都是不对的。输出来的会是 undefined。

*那么到底发生了什么？*看起来我们面对的是一个先有鸡还是先有蛋的问题。到底是声明（蛋）在前，还是赋值（鸡）在前？

4.2 编译器再度来袭

为了搞明白这个问题，我们需要回顾一下第 1 章中关于编译器的内容。回忆一下，引擎会在解释 JavaScript 代码之前首先对其进行编译。编译阶段中的一部分工作就是找到所有的声明，并用合适的作用域将它们关联起来。第 2 章中展示了这个机制，也正是词法作用域的核心内容。

因此，正确的思考思路是，包括变量和函数在内的所有声明都会在任何代码被执行前首先被处理。

当你看到 var a = 2; 时，可能会认为这是一个声明。但 JavaScript 实际上会将其看成两个声明：var a; 和 a = 2;。第一个定义声明是在编译阶段进行的。第二个赋值声明会被*留在原地*等待执行阶段。

我们的第一个代码片段会以如下形式进行处理：

```
var a;

a = 2;

console.log( a );
```

其中第一部分是编译，而第二部分是执行。

类似地，我们的第二个代码片段实际是按照以下流程处理的：

```
var a;

console.log( a );

a = 2;
```

因此，打个比方，这个过程就好像变量和函数声明从它们在代码中出现的位置被“移动”到了最上面。这个过程就叫作*提升*。

换句话说，*先有蛋（声明）后有鸡（赋值）*。

只有声明本身会被提升，而赋值或其他运行逻辑会留在*原地*。如果提升改变了代码执行的顺序，会造成非常严重的破坏。

```
foo();

function foo() {
    console.log( a ); // undefined
    var a = 2;
}
```

foo 函数的声明（这个例子还*包括*实际函数的隐含值）被提升了，因此第一行中的调用可以正常执行。

另外值得注意的是，*每个作用域*都会进行提升操作。尽管前面大部分的代码片段已经简化了（因为它们只包含全局作用域），而我们正在讨论的 foo(..) 函数自身也会在内部对 var a 进行提升（显然并不是提升到了整个程序的最上方）。因此这段代码实际上会被理解为下面的形式：

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
}

foo();
```

可以看到，函数声明会被提升，但是函数表达式却不会被提升。

```
foo(); // 不是 ReferenceError，而是 TypeError!

var foo = function bar() {
    // ...
};
```

这段程序中的变量标识符 foo() 被提升并分配给所在作用域（在这里是全局作用域），因此 foo() 不会导致 ReferenceError。但是 foo 此时并没有赋值（如果它是一个函数声明而不是函数表达式，那么就会赋值）。foo() 由于对 undefined 值进行函数调用而导致非法操作，因此抛出 TypeError 异常。

同时也要记住，即使是具名的函数表达式，名称标识符在赋值之前也无法在所在作用域中

使用：

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

这个代码片段经过提升后，实际上会被理解为以下形式：

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

4.3 函数优先

函数声明和变量声明都会被提升。但是一个值得注意的细节（这个细节可以出现在有多个“重复”声明的代码中）是函数会首先被提升，然后才是变量。

考虑以下代码：

```
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

foo = function() {
    console.log( 2 );
};
```

会输出 1 而不是 2！这个代码片段会被引擎理解为如下形式：

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
    console.log( 2 );
};
```

注意，var foo 尽管出现在 function foo()... 的声明之前，但它是重复的声明（因此被忽略了），因为函数声明会被提升到普通变量之前。

尽管重复的 var 声明会被忽略掉，但出现在后面的函数声明还是可以覆盖前面的。

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

虽然这些听起来都是些无用的学院理论，但是它说明了在同一个作用域中进行重复定义是非常糟糕的，而且经常会导致各种奇怪的问题。

一个普通块内部的函数声明通常会被提升到所在作用域的顶部，这个过程不会像下面的代码暗示的那样可以被条件判断所控制：

```
foo(); // "b"

var a = true;
if (a) {
    function foo() { console.log("a"); }
}
else {
    function foo() { console.log("b"); }
}
```

但是需要注意这个行为并不可靠，在 JavaScript 未来的版本中有可能发生改变，因此应该尽可能避免在块内部声明函数。

4.4 小结

我们习惯将 var a = 2; 看作一个声明，而实际上 JavaScript 引擎并不这么认为。它将 var a 和 a = 2 当作两个单独的声明，第一个是编译阶段的任务，而第二个则是执行阶段的任务。

这意味着无论作用域中的声明出现在什么地方，都将在代码本身被执行前首先进行处理。可以将这个过程形象地想象成所有的声明（变量和函数）都会被“移动”到各自作用域的最顶端，这个过程被称为提升。

声明本身会被提升，而包括函数表达式的赋值在内的赋值操作并不会提升。

要注意避免重复声明，特别是当普通的 var 声明和函数声明混合在一起的时候，否则会引起很多危险的问题！

第5章

作用域闭包

接下来的内容需要对作用域工作原理相关的基础知识有非常深入的理解。

我们将注意力转移到这门语言中一个非常重要但又难以掌握，近乎神话的概念上：闭包。如果你了解了之前关于词法作用域的讨论，那么闭包的概念几乎是不言自明的。魔术师的幕布后藏着一个人，我们将要揭开他的伪装。我可没说这个人是 Crockford[1]！

在继续学习之前，如果你还是对词法作用域相关内容有疑问，可以重新回顾一下第 2 章中的相关内容，现在是个好机会。

5.1 启示

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，但是需要付出非常多的努力和牺牲才能理解这个概念。

回忆我前几年的时光，大量使用 JavaScript 但却完全不理解闭包是什么。总是感觉这门语言有其隐蔽的一面，如果能够掌握将会功力大涨，但讽刺的是我始终无法掌握其中的门道。还记得我曾经大量阅读早期框架的源码，试图能够理解闭包的工作原理。现在还能回忆起我的脑海中第一次浮现出关于“模块模式”相关概念时的激动心情。

那时我无法理解并且倾尽数年心血来探索的，也就是我马上要传授给你的秘诀：JavaScript

注 1：Douglas Crockford 是 Web 开发领域最知名的技术权威之一，ECMA JavaScript 2.0 标准化委员会委员，被 JavaScript 之父 Brendan Eich 称为 JavaScript 界的宗师。——译者注

中闭包无处不在，你只需要能够识别并拥抱它。 闭包并不是一个需要学习新的语法或模式才能使用的工具，它也不是一件必须接受像 Luke[2] 一样的原力训练才能使用和掌握的武器。

闭包是基于词法作用域书写代码时所产生的自然结果，你甚至不需要为了利用它们而有意识地创建闭包。闭包的创建和使用在你的代码中随处可见。你缺少的是根据你自己的意愿来识别、拥抱和影响闭包的思维环境。

最后你恍然大悟：原来在我的代码中已经到处都是闭包了，现在我终于能理解它们了。理解闭包就好像 Neo[3] 第一次见到矩阵 [4] 一样。

5.2 实质问题

好了，夸张和浮夸的电影比喻已经够多了。

下面是直接了当的定义，你需要掌握它才能理解和识别闭包：

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。

下面用一些代码来解释这个定义。

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

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

    bar();
}

foo();
```

这段代码看起来和嵌套作用域中的示例代码很相似。基于词法作用域的查找规则，函数 `bar()` 可以访问外部作用域中的变量 `a`（这个例子中的是一个 RHS 引用查询）。

这是闭包吗？

技术上来讲，也许是。但根据前面的定义，确切地说并不是。我认为最准确地用来解释 `bar()` 对 `a` 的引用的方法是词法作用域的查找规则，而这些规则只是闭包的一部分。（但却是非常重要的一部分！）

注 2：《星球大战》系列电影中的人物。——译者注
注 3：电影《骇客帝国》的主角。——译者注
注 4：电影《骇客帝国》中拥有自我意识主宰一切的超级计算机。——译者注

从纯学术的角度说，在上面的代码片段中，函数 bar() 具有一个涵盖 foo() 作用域的闭包（事实上，涵盖了它能访问的所有作用域，比如全局作用域）。也可以认为 bar() 被封闭在了 foo() 的作用域中。为什么呢？原因简单明了，因为 bar() 嵌套在 foo() 内部。

但是通过这种方式定义的闭包并不能直接进行观察，也无法明白在这个代码片段中闭包是如何工作的。我们可以很容易地理解词法作用域，而闭包则隐藏在代码之后的神秘阴影里，并不那么容易理解。

下面我们来看一段代码，清晰地展示了闭包：

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

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

    return bar;
}

var baz = foo();

baz(); // 2 —— 朋友，这就是闭包的效果。
```

函数 bar() 的词法作用域能够访问 foo() 的内部作用域。然后我们将 bar() 函数本身当作一个值类型进行传递。在这个例子中，我们将 bar 所引用的函数对象本身当作返回值。

在 foo() 执行后，其返回值（也就是内部的 bar() 函数）赋值给变量 baz 并调用 baz()，实际上只是通过不同的标识符引用调用了内部的函数 bar()。

bar() 显然可以被正常执行。但是在这个例子中，它在自己定义的词法作用域以外的地方执行。

在 foo() 执行后，通常会期待 foo() 的整个内部作用域都被销毁，因为我们知道引擎有垃圾回收器用来释放不再使用的内存空间。由于看上去 foo() 的内容不会再被使用，所以很自然地会考虑对其进行回收。

而闭包的“神奇”之处正是可以阻止这件事情的发生。事实上内部作用域依然存在，因此没有被回收。谁在使用这个内部作用域？原来是 bar() 本身在使用。

拜 bar() 所声明的位置所赐，它拥有涵盖 foo() 内部作用域的闭包，使得该作用域能够一直存活，以供 bar() 在之后任何时间进行引用。

bar() 依然持有对该作用域的引用，而这个引用就叫作闭包。

因此，在几微秒之后变量 baz 被实际调用（调用内部函数 bar），不出意料它可以访问定义