```
    }
};
```

2. 模块管理

另外一种避免冲突的办法和现代的模块机制很接近，就是从众多模块管理器中挑选一个来使用。使用这些工具，任何库都无需将标识符加入到全局作用域中，而是通过依赖管理器的机制将库的标识符显式地导入到另外一个特定的作用域中。

显而易见，这些工具并没有能够违反词法作用域规则的“神奇”功能。它们只是利用作用域的规则强制所有标识符都不能注入到共享作用域中，而是保持在私有、无冲突的作用域中，这样可以有效规避掉所有的意外冲突。

因此，只要你愿意，即使不使用任何依赖管理工具也可以实现相同的功效。第 5 章会介绍模块模式的详细内容。

3.3 函数作用域

我们已经知道，在任意代码片段外部添加包装函数，可以将内部的变量和函数定义“隐藏”起来，外部作用域无法访问包装函数内部的任何内容。

例如：

```
var a = 2;

function foo() { // <-- 添加这一行

    var a = 3;
    console.log( a ); // 3

} // <-- 以及这一行
foo(); // <-- 以及这一行

console.log( a ); // 2
```

虽然这种技术可以解决一些问题，但是它并不理想，因为会导致一些额外的问题。首先，必须声明一个具名函数 `foo()`，意味着 `foo` 这个名称本身“污染”了所在作用域（在这个例子中是全局作用域）。其次，必须显式地通过函数名（`foo()`）调用这个函数才能运行其中的代码。

如果函数不需要函数名（或者至少函数名可以不污染所在作用域），并且能够自动运行，这将会更加理想。

幸好，JavaScript 提供了能够同时解决这两个问题的方案、

```
var a = 2;
```

```
(function foo(){ // <-- 添加这一行

    var a = 3;
    console.log( a ); // 3

})(); // <-- 以及这一行

console.log( a ); // 2
```

接下来我们分别介绍这里发生的事情。

首先，包装函数的声明以 (function... 而不仅是以 function... 开始。尽管看上去这并不是一个很显眼的细节，但实际上却是非常重要的区别。函数会被当作函数表达式而不是一个标准的函数声明来处理。

区分函数声明和表达式最简单的方法是看 function 关键字出现在声明中的位置（不仅仅是一行代码，而是整个声明中的位置）。如果 function 是声明中的第一个词，那么就是一个函数声明，否则就是一个函数表达式。

函数声明和函数表达式之间最重要的区别是它们的名称标识符将会绑定在何处。

比较一下前面两个代码片段。第一个片段中 foo 被绑定在所在作用域中，可以直接通过 foo() 来调用它。第二个片段中 foo 被绑定在函数表达式自身的函数中而不是所在作用域中。

换句话说，(function foo(){ .. }) 作为函数表达式意味着 foo 只能在 .. 所代表的位置中被访问，外部作用域则不行。foo 变量名被隐藏在自身中意味着不会非必要地污染外部作用域。

3.3.1 匿名和具名

对于函数表达式你最熟悉的场景可能就是回调参数了，比如：

```
setTimeout( function() {
    console.log("I waited 1 second!");
}, 1000 );
```

这叫作*匿名函数表达式*，因为 function().. 没有名称标识符。函数表达式可以是匿名的，而函数声明则不可以省略函数名——在 JavaScript 的语法中这是非法的。

匿名函数表达式书写起来简单快捷，很多库和工具也倾向鼓励使用这种风格的代码。但是它也有几个缺点需要考虑。

1. 匿名函数在栈追踪中不会显示出有意义的函数名，使得调试很困难。

2. 如果没有函数名，当函数需要引用自身时只能使用已经过期的 arguments.callee 引用，比如在递归中。另一个函数需要引用自身的例子，是在事件触发后事件监听器需要解绑自身。
3. 匿名函数省略了对于代码可读性 / 可理解性很重要的函数名。一个描述性的名称可以让代码不言自明。

行内函数表达式非常强大且有用——匿名和具名之间的区别并不会对这点有任何影响。给函数表达式指定一个函数名可以有效解决以上问题。始终给函数表达式命名是一个最佳实践：

```
setTimeout( function timeoutHandler() { // <-- 快看，我有名字了！

    console.log( "I waited 1 second!" );
}, 1000 );
```

3.3.2 立即执行函数表达式

```
var a = 2;

(function foo() {

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

由于函数被包含在一对（ ）括号内部，因此成为了一个表达式，通过在末尾加上另外一个（ ）可以立即执行这个函数，比如 (function foo(){ .. })()。第一个（ ）将函数变成表达式，第二个（ ）执行了这个函数。

这种模式很常见，几年前社区给它规定了一个术语：IIFE，代表立即执行函数表达式（Immediately Invoked Function Expression）；

函数名对 IIFE 当然不是必须的，IIFE 最常见的用法是使用一个匿名函数表达式。虽然使用具名函数的 IIFE 并不常见，但它具有上述匿名函数表达式的所有优势，因此也是一个值得推广的实践。

```
var a = 2;

(function IIFE() {

    var a = 3;
    console.log( a ); // 3

})();

console.log( a ); // 2
```

相较于传统的 IIFE 形式，很多人都更喜欢另一个改进的形式：(function(){ .. }())。仔细观察其中的区别。第一种形式中函数表达式被包含在 ( ) 中，然后在后面用另一个 () 括号来调用。第二种形式中用来调用的 () 括号被移进了用来包装的 ( ) 括号中。

这两种形式在功能上是一致的。选择哪个全凭个人喜好。

IIFE 的另一个非常普遍的进阶用法是把它们当作函数调用并传递参数进去。

例如：

```
var a = 2;

(function IIFE( global ) {

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

console.log( a ); // 2
```

我们将 window 对象的引用传递进去，但将参数命名为 global，因此在代码风格上对全局对象的引用变得比引用一个没有“全局”字样的变量更加清晰。当然可以从外部作用域传递任何你需要的东西，并将变量命名为任何你觉得合适的名字。这对于改进代码风格是非常有帮助的。

这个模式的另外一个应用场景是解决 undefined 标识符的默认值被错误覆盖导致的异常（虽然不常见）。将一个参数命名为 undefined，但是在对应的位置不传入任何值，这样就可以保证在代码块中 undefined 标识符的值真的是 undefined：

```
undefined = true; // 给其他代码挖了一个大坑！绝对不要这样做！

(function IIFE( undefined ) {

    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }

})();
```

IIFE 还有一种变化的用途是倒置代码的运行顺序，将需要运行的函数放在第二位，在 IIFE 执行之后当作参数传递进去。这种模式在 UMD（Universal Module Definition）项目中被广泛使用。尽管这种模式略显冗长，但有些人认为它更易理解。

```
var a = 2;
```

```
(function IIFE( def ) {
    def( window );
})(function def( global ) {

    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

});
```

函数表达式 def 定义在片段的第二部分，然后当作参数（这个参数也叫作 def）被传递进 IIFE 函数定义的第一部分中。最后，参数 def（也就是传递进去的函数）被调用，并将 window 传入当作 global 参数的值。

3.4 块作用域

尽管函数作用域是最常见的作用域单元，当然也是现行大多数 JavaScript 中最普遍的设计方法，但其他类型的作用域单元也是存在的，并且通过使用其他类型的作用域单元甚至可以实现维护起来更加优秀、简洁的代码。

除 JavaScript 外的很多编程语言都支持块作用域，因此其他语言的开发者对于相关的思维方式会很熟悉，但是对于主要使用 JavaScript 的开发者来说，这个概念会很陌生。

尽管你可能连一行带有块作用域风格的代码都没有写过，但对下面这种很常见的 JavaScript 代码一定很熟悉：

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

我们在 for 循环的头部直接定义了变量 i，通常是因为只想在 for 循环内部的上下文中使用 i，而忽略了 i 会被绑定在外部作用域（函数或全局）中的事实。

这就是块作用域的用处。变量的声明应该距离使用的地方越近越好，并最大限度地本地化。另外一个例子：

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

bar 变量仅在 if 声明的上下文中使用，因此如果能将它声明在 if 块内部中会是一个很有意义的事情。但是，当使用 var 声明变量时，它写在哪里都是一样的，因为它们最终都会

属于外部作用域。这段代码是为了风格更易读而伪装出的形式上的块作用域，如果使用这种形式，要确保没在作用域其他地方意外地使用 bar 只能依靠自觉性。

块作用域是一个用来对之前的最小*授权*原则进行扩展的工具，将代码从在函数中隐藏信息扩展为在块中隐藏信息。

再次考虑 for 循环的例子：

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

为什么要把一个只在 for 循环内部使用（至少是*应该*只在内部使用）的变量 i 污染到整个函数作用域中呢？

更重要的是，开发者需要*检查*自己的代码，以避免在作用范围外意外地使用（或复用）某些变量，如果在错误的地方使用变量将导致未知变量的异常。变量 i 的块作用域（如果存在的话）将使得其只能在 for 循环内部使用，如果在函数中其他地方使用会导致错误。这对保证变量不会被混乱地复用及提升代码的可维护性都有很大帮助。

但可惜，表面上看 JavaScript 并没有块作用域的相关功能。

除非你更加深入地研究。

3.4.1 with

我们在第 2 章讨论过 with 关键字。它不仅是一个难于理解的结构，同时*也是*块作用域的一个例子（块作用域的一种形式），用 with 从对象中创建出的作用域仅在 with 声明中而非外部作用域中有效。

3.4.2 try/catch

*非常*少有人会注意到 JavaScript 的 ES3 规范中规定 try/catch 的 catch 分句会创建一个块作用域，其中声明的变量仅在 catch 内部有效。

例如：

```
try {
    undefined(); // 执行一个非法操作来强制制造一个异常
}
catch (err) {
    console.log( err ); // 能够正常执行!
}

console.log( err ); // ReferenceError: err not found
```

正如你所看到的，err 仅存在 catch 分句内部，当试图从别处引用它时会抛出错误。

尽管这个行为已经被标准化，并且被大部分的标准 JavaScript 环境（除了老版本的 IE 浏览器）所支持，但是当同一个作用域中的两个或多个 catch 分句用同样的标识符名称声明错误变量时，很多静态检查工具还是会发出警告。实际上这并不是重复定义，因为所有变量都被安全地限制在块作用域内部，但是静态检查工具还是会很烦人地发出警告。

为了避免这个不必要的警告，很多开发者会将 catch 的参数命名为 err1、err2 等。也有开发者干脆关闭了静态检查工具对重复变量名的检查。

也许 catch 分句会创建块作用域这件事看起来像教条的学院理论一样没什么用处，但是查看附录 B 就会发现一些很有用的信息。

3.4.3 let

到目前为止，我们知道 JavaScript 在暴露块作用域的功能中有一些奇怪的行为。如果仅仅是这样，那么 JavaScript 开发者多年来也就不会将块作用域当作非常有用的机制来使用了。

幸好，ES6 改变了现状，引入了新的 let 关键字，提供了除 var 以外的另一种变量声明方式。

let 关键字可以将变量绑定到所在的任意作用域中（通常是 { .. } 内部）。换句话说，let 为其声明的变量隐式地了所在的块作用域。

```
var foo = true;

if (foo) {
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}

console.log( bar ); // ReferenceError
```

用 let 将变量附加在一个已经存在的块作用域上的行为是隐式的。在开发和修改代码的过程中，如果没有密切关注哪些块作用域中有绑定的变量，并且习惯性地移动这些块或者将其包含在其他的块中，就会导致代码变得混乱。

为块作用域显式地创建块可以部分解决这个问题，使变量的附属关系变得更加清晰。通常来讲，显式的代码优于隐式或一些精巧但不清晰的代码。显式的块作用域风格非常容易书写，并且和其他语言中块作用域的工作原理一致：

```
var foo = true;
```

```
if (foo) {
    { // <-- 显式的快
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
    }
}

console.log( bar ); // ReferenceError
```

只要声明是有效的，在声明中的任意位置都可以使用 { .. } 括号来为 let 创建一个用于绑定的块。在这个例子中，我们在 if 声明内部显式地创建了一个块，如果需要对其进行重构，整个块都可以被方便地移动而不会对外部 if 声明的位置和语义产生任何影响。

关于另外一种显式的块作用域表达式的内容，请查看附录 B。

在第 4 章，我们会讨论提升，提升是指声明会被视为存在于其所出现的作用域的整个范围内。

但是使用 let 进行的声明不会在块作用域中进行提升。声明的代码被运行之前，声明并不“存在”。

```
{
    console.log( bar ); // ReferenceError!
    let bar = 2;
}
```

1. 垃圾收集

另一个块作用域非常有用的原因和闭包及回收内存垃圾的回收机制相关。这里简要说明一下，而内部的实现原理，也就是闭包的机制会在第 5 章详细解释。

考虑以下代码：

```
function process(data) {
    // 在这里做点有趣的事情
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt) {
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

click 函数的点击回调并不需要 someReallyBigData 变量。理论上这意味着当 process(..) 执行后，在内存中占用大量空间的数据结构就可以被垃圾回收了。但是，由于 click 函数形成了一个覆盖整个作用域的闭包，JavaScript 引擎极有可能依然保存着这个结构（取决于具体实现）。

块作用域可以打消这种顾虑，可以让引擎清楚地知道没有必要继续保存 someReallyBigData 了：

```
function process(data) {
    // 在这里做点有趣的事情
}

// 在这个块中定义的内容可以销毁了！
{
    let someReallyBigData = { .. };

    process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

为变量显式声明块作用域，并对变量进行本地绑定是非常有用的工具，可以把它添加到你的代码工具箱中了。

2. let循环

一个 let 可以发挥优势的典型例子就是之前讨论的 for 循环。

```
for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError
```

for 循环头部的 let 不仅将 i 绑定到了 for 循环的块中，事实上它将其重新绑定到了循环的每一个迭代中，确保使用上一个循环迭代结束时的值重新进行赋值。

下面通过另一种方式来说明每次迭代时进行重新绑定的行为：

```
{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // 每个迭代重新绑定！
        console.log( i );
    }
}
```

每个迭代进行重新绑定的原因非常有趣，我们会在第 5 章讨论闭包时进行说明。

由于 let 声明附属于一个新的作用域而不是当前的函数作用域（也不属于全局作用域），当代码中存在对于函数作用域中 var 声明的隐式依赖时，就会有很多隐藏的陷阱，如果用 let 来替代 var 则需要在代码重构的过程中付出额外的精力。

考虑以下代码：

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    if (baz > bar) {
        console.log( baz );
    }

    // ...
}
```

这段代码可以简单地被重构成下面的同等形式：

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;
    // ...
}

if (baz > bar) {
    console.log( baz );
}
```

但是在使用块级作用域的变量时需要注意以下变化：

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- 移动代码时不要忘了 bar!
        console.log( baz );
    }
}
```

参考附录 B，其中介绍了另外一种块作用域形式，可以用更健壮的方式实现目的，并且写出的代码更易维护和重构。

3.4.4 const

除了 let 以外，ES6 还引入了 const，同样可以用来创建块作用域变量，但其值是固定的（常量）。之后任何试图修改值的操作都会引起错误。