注意，这里所说的气泡是严格包含的。我们并不是在讨论文氏图[1]这种可以跨越边界的气泡。换句话说，没有任何函数的气泡可以（部分地）同时出现在两个外部作用域的气泡中，就如同没有任何函数可以部分地同时出现在两个父级函数中一样。

查找

作用域气泡的结构和互相之间的位置关系给引擎提供了足够的位置信息，引擎用这些信息来查找标识符的位置。

在上一个代码片段中，引擎执行 `console.log(..)` 声明，并查找 `a`、`b` 和 `c` 三个变量的引用。它首先从最内部的作用域，也就是 `bar(..)` 函数的作用域气泡开始查找。引擎无法在这里找到 `a`，因此会去上一级到所嵌套的 `foo(..)` 的作用域中继续查找。在这里找到了 `a`，因此引擎使用了这个引用。对 `b` 来讲也是一样的。而对 `c` 来说，引擎在 `bar(..)` 中就找到了它。

如果 `a`、`c` 都存在于 `bar(..)` 和 `foo(..)` 的内部，`console.log(..)` 就可以直接使用 `bar(..)` 中的变量，而无需到外面的 `foo(..)` 中查找。

作用域查找会在找到第一个匹配的标识符时停止。在多层的嵌套作用域中可以定义同名的标识符，这叫作“遮蔽效应”（内部的标识符“遮蔽”了外部的标识符）。抛开遮蔽效应，作用域查找始终从运行时所处的最内部作用域开始，逐级向外或者说向上进行，直到遇见第一个匹配的标识符为止。

全局变量会自动成为全局对象（比如浏览器中的 `window` 对象）的属性，因此可以不直接通过全局对象的词法名称，而是间接地通过对全局对象属性的引用来对其进行访问。

```
window.a
```

通过这种技术可以访问那些被同名变量所遮蔽的全局变量。但非全局的变量如果被遮蔽了，无论如何都无法被访问到。

无论函数在*哪里*被调用，也无论它*如何*被调用，它的词法作用域都*只由*函数被声明时所处的位置决定。

词法作用域查找*只会*查找一级标识符，比如 `a`、`b` 和 `c`。如果代码中引用了 `foo.bar.baz`，词法作用域查找只会试图查找 `foo` 标识符，找到这个变量后，对象属性访问规则会分别接管对 `bar` 和 `baz` 属性的访问。

注 1：集合论中用以表示集合（或类）的一种草图。http://zh.wikipedia.org/wiki/%E6%96%87%E6%B0%8F%E5%9B%BE。——译者注

2.2 欺骗词法

如果词法作用域完全由写代码期间函数所声明的位置来定义，怎样才能在运行时来“修改”（也可以说欺骗）词法作用域呢？

JavaScript 中有两种机制来实现这个目的。社区普遍认为在代码中使用这两种机制并不是什么好注意。但是关于它们的争论通常会忽略掉最重要的点：*欺骗词法作用域会导致性能下降*。

在详细解释性能问题之前，先来看看这两种机制分别是什么原理。

2.2.1 eval

JavaScript 中的 `eval(..)` 函数可以接受一个字符串为参数，并将其中的内容视为好像在书写时就存在于程序中这个位置的代码。换句话说，可以在你写的代码中用程序生成代码并运行，就好像代码是写在那个位置的一样。

根据这个原理来理解 `eval(..)`，它是如何通过代码欺骗和假装成书写时（也就是词法期）代码就在那，来实现修改词法作用域环境的，这个原理就变得清晰易懂了。

在执行 `eval(..)` 之后的代码时，引擎并不“知道”或“在意”前面的代码是以动态形式插入进来，并对词法作用域的环境进行修改的。引擎只会如往常地进行词法作用域查找。

考虑以下代码：

```
function foo(str, a) {
    eval( str ); // 欺骗!
    console.log( a, b );
}

var b = 2;

foo( "var b = 3;", 1 ); // 1, 3
```

`eval(..)` 调用中的 `"var b = 3;"` 这段代码会被当作本来就在那里一样来处理。由于那段代码声明了一个新的变量 `b`，因此它对已经存在的 `foo(..)` 的词法作用域进行了修改。事实上，和前面提到的原理一样，这段代码实际上在 `foo(..)` 内部创建了一个变量 `b`，并遮蔽了外部（全局）作用域中的同名变量。

当 `console.log(..)` 被执行时，会在 `foo(..)` 的内部同时找到 `a` 和 `b`，但是永远也无法找到外部的 `b`。因此会输出“1, 3”而不是正常情况下会输出的“1, 2”。

在这个例子中，为了展示的方便和简洁，我们传递进去的“代码”字符串是固定不变的。而在实际情况中，可以非常容易地根据程序逻辑动态地将字符拼接在一起之后再传递进去。eval(..) 通常被用来执行动态创建的代码，因为像例子中这样动态地执行一段固定字符所组成的代码，并没有比直接将代码写在那里更有好处。

默认情况下，如果 eval(..) 中所执行的代码包含有一个或多个声明（无论是变量还是函数），就会对 eval(..) 所处的词法作用域进行修改。技术上，通过一些技巧（已经超出我们的讨论范围）可以间接调用 eval(..) 来使其运行在全局作用域中，并对全局作用域进行修改。但无论何种情况，eval(..) 都可以在运行期修改书写期的词法作用域。

在严格模式的程序中，eval(..) 在运行时有其自己的词法作用域，意味着其中的声明无法修改所在的作用域。

```
function foo(str) {
    "use strict";
    eval( str );
    console.log( a ); // ReferenceError: a is not defined
}

foo( "var a = 2" );
```

JavaScript 中还有其他一些功能效果和 eval(..) 很相似。setTimeout(..) 和 setInterval(..) 的第一个参数可以是字符串，字符串的内容可以被解释为一段动态生成的函数代码。这些功能已经过时且并不被提倡。不要使用它们!

new Function(..) 函数的行为也很类似，最后一个参数可以接受代码字符串，并将其转化为动态生成的函数（前面的参数是这个新生成的函数的形参）。这种构建函数的语法比 eval(..) 略微安全一些，但也要尽量避免使用。

在程序中动态生成代码的使用场景非常罕见，因为它所带来的好处无法抵消性能上的损失。

2.2.2 with

JavaScript 中另一个难以掌握（并且现在也不推荐使用）的用来欺骗词法作用域的功能是 with 关键字。可以有很多方法来解释 with，在这里我选择从这个角度来解释它：它如何同被它所影响的词法作用域进行交互。

with 通常被当作重复引用同一个对象中的多个属性的快捷方式，可以不需要重复引用对象本身。

比如：

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};

// 单调乏味的重复 "obj"
obj.a = 2;
obj.b = 3;
obj.c = 4;

// 简单的快捷方式
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

但实际上这不仅仅是为了方便地访问对象属性。考虑如下代码：

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};

foo( o1 );
console.log( o1.a ); // 2

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2——不好，a 被泄漏到全局作用域上了！
```

这个例子中创建了 o1 和 o2 两个对象。其中一个具有 a 属性，另外一个没有。foo(..) 函数接受一个 obj 参数，该参数是一个对象引用，并对这个对象引用执行了 with(obj) {..}。在 with 块内部，我们写的代码看起来只是对变量 a 进行简单的词法引用，实际上就是一个 LHS 引用（查看第 1 章），并将 2 赋值给它。

当我们将 o1 传递进去，a = 2 赋值操作找到了 o1.a 并将 2 赋值给它，这在后面的 console.log(o1.a) 中可以体现。而当 o2 传递进去，o2 并没有 a 属性，因此不会创建这个属性，o2.a 保持 undefined。

但是可以注意到一个奇怪的副作用，实际上 a = 2 赋值操作创建了一个全局的变量 a。这是怎么回事？

with 可以将一个没有或有多个属性的对象处理为一个完全隔离的词法作用域，因此这个对象的属性也会被处理为定义在这个作用域中的词法标识符。

尽管 with 块可以将一个对象处理为词法作用域，但是这个块内部正常的 var 声明并不会被限制在这个块的作用域中，而是被添加到 with 所处的函数作用域中。

eval(..) 函数如果接受了含有一个或多个声明的代码，就会修改其所处的词法作用域，而 with 声明实际上是根据你传递给它的对象凭空创建了一个全新的词法作用域。

可以这样理解，当我们传递 o1 给 with 时，with 所声明的作用域是 o1，而这个作用域中含有一个同 o1.a 属性相符的标识符。但当我们将 o2 作为作用域时，其中并没有 a 标识符，因此进行了正常的 LHS 标识符查找（查看第 1 章）。

o2 的作用域、foo(..) 的作用域和全局作用域中都没有找到标识符 a，因此当 a = 2 执行时，自动创建了一个全局变量（因为是非严格模式）。

with 这种将对象及其属性放进一个作用域并同时分配标识符的行为很让人费解。但为了说明我们所看到的现象，这是我能给出的最直白的解释了。

另外一个不推荐使用 eval(..) 和 with 的原因是会被严格模式所影响（限制）。with 被完全禁止，而在保留核心功能的前提下，间接或非安全地使用 eval(..) 也被禁止了。

2.2.3 性能

eval(..) 和 with 会在运行时修改或创建新的作用域，以此来欺骗其他在书写时定义的词法作用域。

你可能会问，那又怎样呢？如果它们能实现更复杂的功能，并且代码更具有扩展性，难道不是非常好的功能吗？答案是否定的。

JavaScript 引擎会在编译阶段进行数项的性能优化。其中有些优化依赖于能够根据代码的词法进行静态分析，并预先确定所有变量和函数的定义位置，才能在执行过程中快速找到标识符。

但如果引擎在代码中发现了 eval(..) 或 with，它只能简单地假设关于标识符位置的判断都是无效的，因为无法在词法分析阶段明确知道 eval(..) 会接收到什么代码，这些代码会如何对作用域进行修改，也无法知道传递给 with 用来创建新词法作用域的对象的内容到底是什么。

最悲观的情况是如果出现了 eval(..) 或 with，所有的优化可能都是无意义的，因此最简单的做法就是完全不做任何优化。

如果代码中大量使用 eval(..) 或 with，那么运行起来一定会变得非常慢。无论引擎多聪明，试图将这些悲观情况的副作用限制在最小范围内，也无法避免如果没有这些优化，代码会运行得更慢这个事实。

2.3 小结

词法作用域意味着作用域是由书写代码时函数声明的位置来决定的。编译的词法分析阶段基本能够知道全部标识符在哪里以及是如何声明的，从而能够预测在执行过程中如何对它们进行查找。

JavaScript 中有两个机制可以“欺骗”词法作用域：eval(..) 和 with。前者可以对一段包含一个或多个声明的“代码”字符串进行演算，并借此来修改已经存在的词法作用域（在运行时）。后者本质上是通过将一个对象的引用当作作用域来处理，将对象的属性当作作用域中的标识符来处理，从而创建了一个新的词法作用域（同样是在运行时）。

这两个机制的副作用是引擎无法在编译时对作用域查找进行优化，因为引擎只能谨慎地认为这样的优化是无效的。使用这其中任何一个机制都将导致代码运行变慢。不要使用它们。

第 3 章

函数作用域和块作用域

正如我们在第 2 章中讨论的那样，作用域包含了一系列的“气泡”，每一个都可以作为容器，其中包含了标识符（变量、函数）的定义。这些气泡互相嵌套并且整齐地排列成蜂窝型，排列的结构是在写代码时定义的。

但是，究竟是什么生成了一个新的气泡？只有函数会生成新的气泡吗？ JavaScript 中的其他结构能生成作用域气泡吗？

3.1 函数中的作用域

对于前面提出的问题，最常见的答案是 JavaScript 具有基于函数的作用域，意味着每声明一个函数都会为其自身创建一个气泡，而其他结构都不会创建作用域气泡。但事实上这并不完全正确，下面我们来看一下。

首先需要研究一下函数作用域及其背后的一些内容。

考虑下面的代码：

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

    // 一些代码

    function bar() {
        // ...
    }
```

```
    // 更多的代码

    var c = 3;
}
```

在这个代码片段中，foo(..) 的作用域气泡中包含了标识符 a、b、c 和 bar。无论标识符声明出现在作用域中的何处，这个标识符所代表的变量或函数都将附属于所处作用域的气泡。我们将在下一章讨论具体的原理。

bar(..) 拥有自己的作用域气泡。全局作用域也有自己的作用域气泡，它只包含了一个标识符：foo。

由于标识符 a、b、c 和 bar 都附属于 foo(..) 的作用域气泡，因此无法从 foo(..) 的外部对它们进行访问。也就是说，这些标识符全都无法从全局作用域中进行访问，因此下面的代码会导致 ReferenceError 错误：

```
bar(); // 失败

console.log( a, b, c ); // 三个全都失败
```

但是，这些标识符（a、b、c、foo 和 bar）在 foo(..) 的内部都是可以被访问的，同样在 bar(..) 内部也可以被访问（假设 bar(..) 内部没有同名的标识符声明）。

函数作用域的含义是指，属于这个函数的全部变量都可以在整个函数的范围内使用及复用（事实上在嵌套的作用域中也可以使用）。这种设计方案是非常有用的，能充分利用 JavaScript 变量可以根据需要改变值类型的“动态”特性。

但与此同时，如果不细心处理那些可以在整个作用域范围内被访问的变量，可能会带来意想不到的问题。

3.2 隐藏内部实现

对函数的传统认知就是先声明一个函数，然后再向里面添加代码。但反过来想也可以带来一些启示：从所写的代码中挑选出一个任意的片段，然后用函数声明对它进行包装，实际上就是把这些代码“隐藏”起来了。

实际的结果就是在这个代码片段的周围创建了一个作用域气泡，也就是说这段代码中的任何声明（变量或函数）都将绑定在这个新创建的包装函数的作用域中，而不是先前所在的作用域中。换句话说，可以把变量和函数包裹在一个函数的作用域中，然后用这个作用域来“隐藏”它们。

为什么“隐藏”变量和函数是一个有用的技术？

有很多原因促成了这种基于作用域的隐藏方法。它们大都是从最小特权原则中引申出来的，也叫最小授权或最小暴露原则。这个原则是指在软件设计中，应该最小限度地暴露必要内容，而将其他内容都“隐藏”起来，比如某个模块或对象的 API 设计。

这个原则可以延伸到如何选择作用域来包含变量和函数。如果所有变量和函数都在全局作用域中，当然可以在所有的内部嵌套作用域中访问到它们。但这样会破坏前面提到的最小特权原则，因为可能会暴漏过多的变量或函数，而这些变量或函数本应该是私有的，正确的代码应该是可以阻止对这些变量或函数进行访问的。

例如：

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

在这个代码片段中，变量 b 和函数 doSomethingElse(..) 应该是 doSomething(..) 内部具体实现的“私有”内容。给予外部作用域对 b 和 doSomethingElse(..) 的“访问权限”不仅没有必要，而且可能是“危险”的，因为它们可能被有意或无意地以非预期的方式使用，从而导致超出了 doSomething(..) 的适用条件。更“合理”的设计会将这些私有的具体内容隐藏在 doSomething(..) 内部，例如：

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

doSomething( 2 ); // 15
```

现在，b 和 doSomethingElse(..) 都无法从外部被访问，而只能被 doSomething(..) 所控制。功能性和最终效果都没有受影响，但是设计上将具体内容私有化了，设计良好的软件都会依此进行实现。

规避冲突

“隐藏”作用域中的变量和函数所带来的另一个好处，是可以避免同名标识符之间的冲突，两个标识符可能具有相同的名字但用途却不一样，无意间可能造成命名冲突。冲突会导致变量的值被意外覆盖。

例如：

```
function foo() {
    function bar(a) {
        i = 3; // 修改 for 循环所属作用域中的 i
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // 糟糕，无限循环了！
    }
}

foo();
```

`bar(..)` 内部的赋值表达式 `i = 3` 意外地覆盖了声明在 `foo(..)` 内部 `for` 循环中的 `i`。在这个例子中将会导致无限循环，因为 `i` 被固定设置为 3，永远满足小于 10 这个条件。

`bar(..)` 内部的赋值操作需要声明一个本地变量来使用，采用任何名字都可以，`var i = 3;` 就可以满足这个需求（同时会为 `i` 声明一个前面提到过的“遮蔽变量”）。另外一种方法是采用一个完全不同的标识符名称，比如 `var j = 3;`。但是软件设计在某种情况下可能自然而然地要求使用同样的标识符名称，因此在这种情况下使用作用域来“隐藏”内部声明是唯一的最佳选择。

1. 全局命名空间

变量冲突的一个典型例子存在于全局作用域中。当程序中加载了多个第三方库时，如果它们没有妥善地将内部私有的函数或变量隐藏起来，就会很容易引发冲突。

这些库通常会在全局作用域中声明一个名字足够独特的变量，通常是一个对象。这个对象被用作库的*命名空间*，所有需要暴露给外界的功能都会成为这个对象（命名空间）的属性，而不是将自己的标识符暴漏在顶级的词法作用域中。

例如：

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
        // ...
```