因此在这里只进行宏观、简单的介绍，接下来你就会发现我们介绍的这些看起来有点高深的内容与所要讨论的事情有什么关联。

首先，JavaScript 引擎不会有大量的（像其他语言编译器那么多的）时间用来进行优化，因为与其他语言不同，JavaScript 的编译过程不是发生在构建之前的。

对于 JavaScript 来说，大部分情况下编译发生在代码执行前的几微秒（甚至更短！）的时间内。在我们所要讨论的作用域背后，JavaScript 引擎用尽了各种办法（比如 JIT，可以延迟编译甚至实施重编译）来保证性能最佳。

简单地说，任何 JavaScript 代码片段在执行前都要进行编译（通常就在执行前）。因此，JavaScript 编译器首先会对 `var a = 2;` 这段程序进行编译，然后做好执行它的准备，并且通常马上就会执行它。

1.2 理解作用域

我们学习作用域的方式是将这个过程模拟成几个人物之间的对话。那么，由谁进行这场对话呢？

1.2.1 演员表

首先介绍将要参与到对程序 `var a = 2;` 进行处理的过程中的演员们，这样才能理解接下来将要听到的对话。

- 引擎
  从头到尾负责整个 JavaScript 程序的编译及执行过程。
- 编译器
  引擎的好朋友之一，负责语法分析及代码生成等脏活累活（详见前一节的内容）。
- 作用域
  引擎的另一位好朋友，负责收集并维护由所有声明的标识符（变量）组成的一系列查询，并实施一套非常严格的规则，确定当前执行的代码对这些标识符的访问权限。

为了能够完全理解 JavaScript 的工作原理，你需要开始像引擎（和它的朋友们）一样思考，从它们的角度提出问题，并从它们的角度回答这些问题。

1.2.2 对话

当你看见 `var a = 2;` 这段程序时，很可能认为这是一句声明。但我们的新朋友引擎却不这么看。事实上，引擎认为这里有两个完全不同的声明，一个由编译器在编译时处理，另一

个则由引擎在运行时处理。

下面我们将 `var a = 2;` 分解，看看引擎和它的朋友们是如何协同工作的。

编译器首先会将这段程序分解成词法单元，然后将词法单元解析成一个树结构。但是当编译器开始进行代码生成时，它对这段程序的处理方式会和预期的有所不同。

可以合理地假设编译器所产生的代码能够用下面的伪代码进行概括："为一个变量分配内存，将其命名为 a，然后将值 2 保存进这个变量。"然而，这并不完全正确。

事实上编译器会进行如下处理。

1. 遇到 `var a`，编译器会询问作用域是否已经有一个该名称的变量存在于同一个作用域的集合中。如果是，编译器会忽略该声明，继续进行编译；否则它会要求作用域在当前作用域的集合中声明一个新的变量，并命名为 a。
2. 接下来编译器会为引擎生成运行时所需的代码，这些代码被用来处理 `a = 2` 这个赋值操作。引擎运行时会首先询问作用域，在当前的作用域集合中是否存在一个叫作 a 的变量。如果是，引擎就会使用这个变量；如果否，引擎会继续查找该变量（查看 1.3 节）。

如果引擎最终找到了 a 变量，就会将 2 赋值给它。否则引擎就会举手示意并抛出一个异常！

总结：变量的赋值操作会执行两个动作，首先编译器会在当前作用域中声明一个变量（如果之前没有声明过），然后在运行时引擎会在作用域中查找该变量，如果能够找到就会对它赋值。

1.2.3 编译器有话说

为了进一步理解，我们需要多介绍一点编译器的术语。

编译器在编译过程的第二步中生成了代码，引擎执行它时，会通过查找变量 a 来判断它是否已声明过。查找的过程由作用域进行协助，但是引擎执行怎样的查找，会影响最终的查找结果。

在我们的例子中，引擎会为变量 a 进行 LHS 查询。另外一个查找的类型叫作 RHS。

我打赌你一定能猜到"L"和"R"的含义，它们分别代表左侧和右侧。

什么东西的左侧和右侧？是一个赋值操作的左侧和右侧。

换句话说，当变量出现在赋值操作的左侧时进行 LHS 查询，出现在右侧时进行 RHS 查询。

讲得更准确一点，RHS 查询与简单地查找某个变量的值别无二致，而 LHS 查询则是试图找到变量的容器本身，从而可以对其赋值。从这个角度说，RHS 并不是真正意义上的“赋值操作的右侧”，更准确地说是“非左侧”。

你可以将 RHS 理解成 retrieve his source value（取到它的源值），这意味着“得到某某的值”。

让我们继续深入研究。

考虑以下代码：

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

其中对 a 的引用是一个 RHS 引用，因为这里 a 并没有赋予任何值。相应地，需要查找并取得 a 的值，这样才能将值传递给 console.log(..)。

相比之下，例如：

```
a = 2;
```

这里对 a 的引用则是 LHS 引用，因为实际上我们并不关心当前的值是什么，只是想要为 = 2 这个赋值操作找到一个目标。

LHS 和 RHS 的含义是“赋值操作的左侧或右侧”并不一定意味着就是“=赋值操作符的左侧或右侧”。赋值操作还有其他几种形式，因此在概念上最好将其理解为“赋值操作的目标是谁（LHS）”以及“谁是赋值操作的源头（RHS）”。

考虑下面的程序，其中既有 LHS 也有 RHS 引用：

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

foo( 2 );
```

最后一行 foo(..) 函数的调用需要对 foo 进行 RHS 引用，意味着“去找到 foo 的值，并把它给我”。并且 (..) 意味着 foo 的值需要被执行，因此它最好真的是一个函数类型的值！

这里还有一个容易被忽略却非常重要的细节。

代码中隐式的 a = 2 操作可能很容易被你忽略掉。这个操作发生在 2 被当作参数传递给 foo(..) 函数时，2 会被分配给参数 a。为了给参数 a（隐式地）分配值，需要进行一次 LHS 查询。

这里还有对 a 进行的 RHS 引用，并且将得到的值传给了 console.log(..)。console.log(..) 本身也需要一个引用才能执行，因此会对 console 对象进行 RHS 查询，并且检查得到的值中是否有一个叫作 log 的方法。

最后，在概念上可以理解为在 LHS 和 RHS 之间通过对值 2 进行交互来将其传递进 log(..)（通过变量 a 的 RHS 查询）。假设在 log(..) 函数的原生实现中它可以接受参数，在将 2 赋值给其中第一个（也许叫作 arg1）参数之前，这个参数需要进行 LHS 引用查询。

你可能会倾向于将函数声明 function foo(a) {... 概念化为普通的变量声明和赋值，比如 var foo、foo = function(a) {...。如果这样理解的话，这个函数声明将需要进行 LHS 查询。

然而还有一个重要的细微差别，编译器可以在代码生成的同时处理声明和值的定义，比如在引擎执行代码时，并不会有线程专门用来将一个函数值“分配给”foo。因此，将函数声明理解成前面讨论的 LHS 查询和赋值的形式并不合适。

1.2.4 引擎和作用域的对话

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

foo( 2 );
```

让我们把上面这段代码的处理过程想象成一段对话，这段对话可能是下面这样的。

引擎：我说作用域，我需要为 foo 进行 RHS 引用。你见过它吗？

作用域：别说，我还真见过，编译器那小子刚刚声明了它。它是一个函数，给你。

引擎：哥们太够意思了！好吧，我来执行一下 foo。

引擎：作用域，还有个事儿。我需要为 a 进行 LHS 引用，这个你见过吗？

作用域：这个也见过，编译器最近把它声名为 foo 的一个形式参数了，拿去吧。

引擎：大恩不言谢，你总是这么棒。现在我要把 2 赋值给 a。

引擎：哥们，不好意思又来打扰你。我要为 console 进行 RHS 引用，你见过它吗？

作用域：咱俩谁跟谁啊，再说我就是干这个。这个我也有，console 是个内置对象。给你。

引擎：么么哒。我得看看这里面是不是有 log(..)。太好了，找到了，是一个函数。

引擎：哥们，能帮我再找一下对 a 的 RHS 引用吗？虽然我记得它，但想再确认一次。

作用域：放心吧，这个变量没有变动过，拿走，不谢。

引擎：真棒。我来把 a 的值，也就是 2，传递进 log(..)。

……

1.2.5 小测验

检验一下到目前的理解程度。把自己当作引擎，并同作用域进行一次“对话”：

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. 找到其中所有的 LHS 查询。（这里有 3 处！）
2. 找到其中所有的 RHS 查询。（这里有 4 处！）

查看本章小结中的参考答案。

1.3 作用域嵌套

我们说过，作用域是根据名称查找变量的一套规则。实际情况中，通常需要同时顾及几个作用域。

当一个块或函数嵌套在另一个块或函数中时，就发生了作用域的嵌套。因此，在当前作用域中无法找到某个变量时，引擎就会在外层嵌套的作用域中继续查找，直到找到该变量，或抵达最外层的作用域（也就是全局作用域）为止。

考虑以下代码：

```
function foo(a) {
    console.log( a + b );
}

var b = 2;

foo( 2 ); // 4
```

对 b 进行的 RHS 引用无法在函数 foo 内部完成，但可以在上一级作用域（在这个例子中就是全局作用域）中完成。

因此，回顾一下引擎和作用域之间的对话，会进一步听到：

> 引擎：foo 的作用域兄弟，你见过 b 吗？我需要对它进行 RHS 引用。
> 作用域：听都没听过，走开。

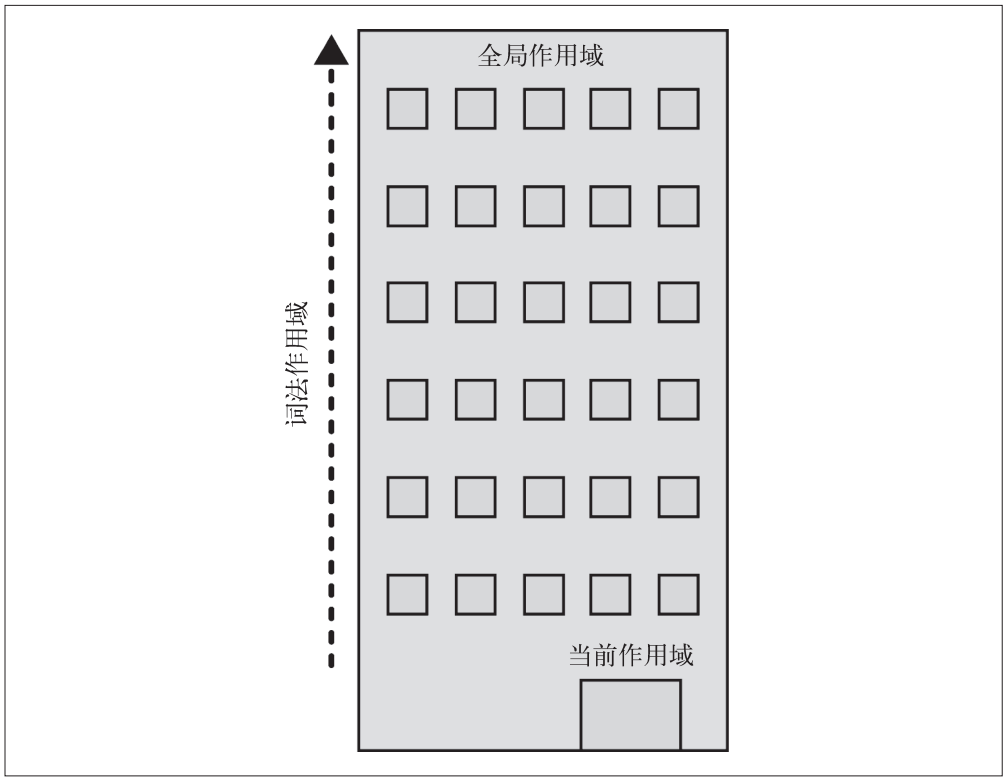
引擎：foo 的上级作用域兄弟，咦？有眼不识泰山，原来你是全局作用域大哥，太好了。你见过 b 吗？我需要对它进行 RHS 引用。

作用域：当然了，给你吧。

遍历嵌套作用域链的规则很简单：引擎从当前的执行作用域开始查找变量，如果找不到，就向上一级继续查找。当抵达最外层的全局作用域时，无论找到还是没找到，查找过程都会停止。

把作用域链比喻成一个建筑

为了将作用域处理的过程可视化，我希望你在脑中想象下面这个高大的建筑：



这个建筑代表程序中的嵌套作用域链。第一层楼代表当前的执行作用域，也就是你所处的位置。建筑的顶层代表全局作用域。

LHS 和 RHS 引用都会在当前楼层进行查找，如果没有找到，就会坐电梯前往上一层楼，如果还是没有找到就继续向上，以此类推。一旦抵达顶层（全局作用域），可能找到了你所需的变量，也可能没找到，但无论如何查找过程都将停止。

1.4 异常

为什么区分 LHS 和 RHS 是一件重要的事情？

因为在变量还没有声明（在任何作用域中都无法找到该变量）的情况下，这两种查询的行为是不一样的。

考虑如下代码：

```
function foo(a) {
    console.log( a + b );
    b = a;
}

foo( 2 );
```

第一次对 b 进行 RHS 查询时是无法找到该变量的。也就是说，这是一个“未声明”的变量，因为在任何相关的作用域中都无法找到它。

如果 RHS 查询在所有嵌套的作用域中遍寻不到所需的变量，引擎就会抛出 ReferenceError 异常。值得注意的是，ReferenceError 是非常重要的异常类型。

相较之下，当引擎执行 LHS 查询时，如果在顶层（全局作用域）中也无法找到目标变量，全局作用域中就会创建一个具有该名称的变量，并将其返还给引擎，前提是程序运行在非“严格模式”下。

“不，这个变量之前并不存在，但是我很热心地帮你创建了一个。”

ES5 中引入了“严格模式”。同正常模式，或者说宽松 / 懒惰模式相比，严格模式在行为上有很多不同。其中一个不同的行为是严格模式禁止自动或隐式地创建全局变量。因此，在严格模式中 LHS 查询失败时，并不会创建并返回一个全局变量，引擎会抛出同 RHS 查询失败时类似的 ReferenceError 异常。

接下来，如果 RHS 查询找到了一个变量，但是你尝试对这个变量的值进行不合理的操作，比如试图对一个非函数类型的值进行函数调用，或着引用 null 或 undefined 类型的值中的属性，那么引擎会抛出另外一种类型的异常，叫作 TypeError。

ReferenceError 同作用域判别失败相关，而 TypeError 则代表作用域判别成功了，但是对结果的操作是非法或不合理的。

1.5 小结

作用域是一套规则，用于确定在何处以及如何查找变量（标识符）。如果查找的目的是对变量进行赋值，那么就会使用 LHS 查询；如果目的是获取变量的值，就会使用 RHS 查询。

赋值操作符会导致 LHS 查询。=操作符或调用函数时传入参数的操作都会导致关联作用域的赋值操作。

JavaScript 引擎首先会在代码执行前对其进行编译，在这个过程中，像 `var a = 2` 这样的声明会被分解成两个独立的步骤：

1. 首先，`var a` 在其作用域中声明新变量。这会在最开始的阶段，也就是代码执行前进行。
2. 接下来，`a = 2` 会查询（LHS 查询）变量 `a` 并对其进行赋值。

LHS 和 RHS 查询都会在当前执行作用域中开始，如果有需要（也就是说它们没有找到所需的标识符），就会向上级作用域继续查找目标标识符，这样每次上升一级作用域（一层楼），最后抵达全局作用域（顶层），无论找到或没找到都将停止。

不成功的 RHS 引用会导致抛出 `ReferenceError` 异常。不成功的 LHS 引用会导致自动隐式地创建一个全局变量（非严格模式下），该变量使用 LHS 引用的目标作为标识符，或者抛出 `ReferenceError` 异常（严格模式下）。

小测验答案

```
function foo(a) {
    var b = a;
    return a + b;
}

var c = foo( 2 );
```

1. 找出所有的 LHS 查询（这里有 3 处！）
   c = ..;、*a* = 2（隐式变量分配）、*b* = ..
2. 找出所有的 RHS 查询（这里有 4 处！）
   foo(2..、= *a*;、*a* ..、.. *b*

第 2 章

词法作用域

在第 1 章中，我们将“作用域”定义为一套规则，这套规则用来管理引擎如何在当前作用域以及嵌套的子作用域中根据标识符名称进行变量查找。

作用域共有两种主要的工作模型。第一种是最为普遍的，被大多数编程语言所采用的*词法作用域*，我们会对这种作用域进行深入讨论。另外一种叫作*动态作用域*，仍有一些编程语言在使用（比如 Bash 脚本、Perl 中的一些模式等）。

附录 A 中介绍了动态作用域，在这里提到它只是为了同 JavaScript 所采用的作用域模型，即词法作用域模型进行对比。

2.1 词法阶段

第 1 章介绍过，大部分标准语言编译器的第一个工作阶段叫作词法化（也叫单词化）。回忆一下，词法化的过程会对源代码中的字符进行检查，如果是有状态的解析过程，还会赋予单词语义。

这个概念是理解词法作用域及其名称来历的基础。

简单地说，词法作用域就是定义在词法阶段的作用域。换句话说，词法作用域是由你在写代码时将变量和块作用域写在哪里来决定的，因此当词法分析器处理代码时会保持作用域不变（大部分情况下是这样的）。

后面会介绍一些欺骗词法作用域的方法，这些方法在词法分析器处理过后依然可以修改作用域，但是这种机制可能有点难以理解。事实上，让词法作用域根据词法关系保持书写时的自然关系不变，是一个非常好的最佳实践。

考虑以下代码：

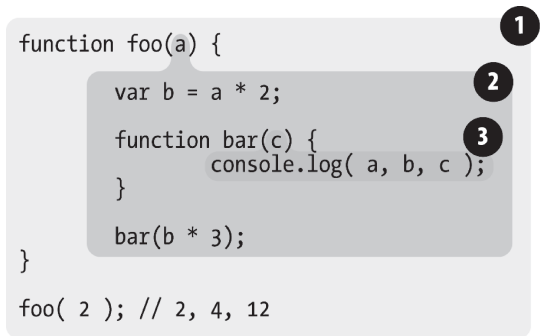
```
function foo(a) {
    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }

    bar( b * 3 );
}

foo( 2 ); // 2, 4, 12
```

在这个例子中有三个逐级嵌套的作用域。为了帮助理解，可以将它们想象成几个逐级包含的气泡。



❶ 包含着整个全局作用域，其中只有一个标识符：foo。

❷ 包含着 foo 所创建的作用域，其中有三个标识符：a、bar 和 b。

❸ 包含着 bar 所创建的作用域，其中只有一个标识符：c。

作用域气泡由其对应的作用域块代码写在哪里决定，它们是逐级包含的。下一章会讨论不同类型的作用域，但现在只要假设每一个函数都会创建一个新的作用域气泡就好了。

bar 的气泡被完全包含在 foo 所创建的气泡中，唯一的原因是那里就是我们希望定义函数 bar 的位置。