几乎所有编程语言最基本的功能之一，就是能够存储变量当中的值，并且能在之后对这个值进行访问或修改。事实上，正是这种存储和访问变量的值的能力将状态带给了程序。

这些变量存储在哪里？程序需要时如何找到他们？这些问题说明需要一套设计良好的规则来存储变量，并且之后可以方便的找到这些变量。这套规则被称为作用域。

究竟在哪里而且怎样设计这些作用域呢的规则呢？

## 1.1编译原理

尽管通常将JavaScript归类为“动态”或“解释执行”语言，但事实上它是一门编译语言。但与传统的编译语言不同，它不是提前编译的，编译结果也不能在分布式系统中进行移植。

JavaScript引擎进行编译的步骤和传统的编译语言非常相似，在某些环节可能比预想的要复杂。

在传统编译语言的流程中，程序中的一段源代码在执行之前会经历三个步骤，统称为“编译”。

* 分词/词法分析（Tokenizing/Lexing）

这个过程会将由字符组成的字符串分解成（对编程语言来说）有意义的代码块，这些代码块被称为词法单元（token）。例如，考虑程序var a = 2;。这段程序通常会被分解成下面这些词法单元：var、a、=、2、;。空格是否会被当做词法单元，取决于空格在这门语言中是否具有意义。

分词（tokenizing）和词法分析（Lexing）之间的区别是非常微妙、晦涩的，主要

差异在于词法单元的识别是通过有状态还是无状态的方式进行的。简单来说，如果词法单元生成器在判断a是一个独立的词法单元还是其他词法单元的一部分时，调用的是有状态的解析规则，那么这个过程就被称为词法分析。

* 解析/语法分析（Parsing）

这个过程是将词法单元流（数组）转换成一个由元素逐级套嵌所组成的代表了程序语法结构的树。这个树被称为“抽象语法树”（Abstract Syntax Treee，AST）。

var a = 2;的抽象语法树中可能会有一个叫作VariableDeclaration的顶级节点，接下来是一个叫做Identifier（它的值是a）的子节点，以及一个叫作AssignmentExpression的子节点。AssignmentExpression节点有一个叫作NumericLiteral（它的值是2）的子节点。

* 代码生成

将AST转换为可执行代码的过程被称为代码生成。这个过程与语言、目标平台等息息相关。

抛开具体细节，简单来说就是有某种方法可以将var a = 2;的AST转换为一组机器指令，用来创建一个叫做a的变量（包括分配内存等），并将一个值存储在a中。

关于引擎如何管理系统资源超出了我们的讨论范围，因此只需要简单地了解引擎

可以根据需要创建并储存变量即可。

比起那些编译过程只有三个步骤的语言的编译器，JavaScript引擎要复杂得多。例如，在语法分析和代码生成阶段有特定步骤来对运行性能进行优化，包括对冗余元素进行优化等。

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妙（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新编译）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

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。

* 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，你需要开始引擎（和它的朋友）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。事实上，引擎却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做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）

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

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来判断它是否已经声明过。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。

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。

“L”和“R”的含义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。

当变量出现在赋值操作的左侧时进行LHS，出现在右侧时进行RHS查询。讲得个人能够准确一点，RHS查询与简单的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”。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”。

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。相应的，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 ); //对a进行的RHS引用，并且将得到的值传给console.log

}

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。