你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true
```

```
# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用`var`声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量`y`，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的`a`将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
```

```
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今

天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于var y所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了console.log(y)访问全局环境中的y。类似的，let x所声明的那个x其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的let/const变量在“声明（和创建）一个标识符”这件事上，与var并没有什么不同，只是JavaScript拒绝访问还没有绑定值的let/const标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的var所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的var x = ...在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的let语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用`var`声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量`y`，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的`a`将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的"结果(result)"只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明

（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* `=` | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表

中的变量是“直接声明的变量”，不能使用delete删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用var声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当var声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码var x = y = 100，在这行代码中，等号的右边是一个表达式y = 100，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。

- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

LeftHandSideExpression < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
```

```
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用var声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当var声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如x。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于var y所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了console.log(y)访问全局环境中的y。类似的，let x所声明的那个x其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的let/const变量在“声明（和创建）一个标识符”这件事上，与var并没有什么不同，只是JavaScript拒绝访问还没有绑定值的let/const标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的var所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的var x = ...在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的let语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用`var`声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量`y`，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的`a`将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“≒”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在"声明（和创建）一个标识符"这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为"变量声明（varDelcs）"，而"let/const"则称为"词法声明（lexicalDecls）"。JavaScript环境在创建一个"变量名（varName in varDecls）"后，会为它初始化绑定一个undefined值，而"词法名字（lexicalNames）"在创建之后就没有这项待遇，所以它们在缺省情况下就是"还没有绑定值"的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，"为变量x绑定一个初值"就可能实现为"在创建环境时将变量x指向一个特定的初始值"。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的"绑定初值"的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* `=` | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实"都是"表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的"变量泄漏"问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么"**一个全局变量是在哪里声明和创建的**"就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为"**间接执行**"，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个"向不存在的变量赋值"的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为"**全局对象**"东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为"**全局对象闭包**"的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与"**with语句**"的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了"可以动态地向全局环境中添加一个

变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用`var`声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量`y`，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的`a`将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

LeftHandSideExpression < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
```

```
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用var声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当var声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量**x**。不可在赋值之前读。

- **const** *x* …

声明常量**x**。不可写。

- **var** *x* …

声明变量**x**。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量**x**。该变量指向一个函数。

- **class** *x* …

声明变量**x**。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

  - 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
  - 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中

的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
```

```
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码var x = y = 100，在这行代码中，等号的右边是一个表达式y = 100，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的delete 0和obj.x之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

  - 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
  - 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“≒”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等

东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用`var`声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量`y`，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的`a`将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
```

```
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

LeftHandSideExpression `=` | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用var声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当var声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量**x**。不可在赋值之前读。

- **const** *x* …

声明常量**x**。不可写。

- **var** *x* …

声明变量**x**。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量**x**。该变量指向一个函数。

- **class** *x* …

声明变量**x**。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“≒”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

LeftHandSideExpression < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用`var`声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量`y`，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的`a`将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如x。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用`var`声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向

不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不

是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* `=` | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用var声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用delete删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用var声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当var声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码var x = y = 100，在这行代码中，等号的右边是一个表达式y = 100，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“≒”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过

动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

LeftHandSideExpression < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
```

```
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用var声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当var声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码`var x = y = 100`，在这行代码中，等号的右边是一个表达式`y = 100`，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的`delete 0`和`obj.x`之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如x。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
  ...
}
```

正是由于var y所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了console.log(y)访问全局环境中的y。类似的，let x所声明的那个x其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的let/const变量在“声明（和创建）一个标识符”这件事上，与var并没有什么不同，只是JavaScript拒绝访问还没有绑定值的let/const标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的var所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的var x = ...在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* < = | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的let语句一样，还要考虑在声明语句

之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用`var`声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用`delete`删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用`var`声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当`var`声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码var x = y = 100，在这行代码中，等号的右边是一个表达式y = 100，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。

你好，我是周爱民。

如果你听过上一讲的内容，心里应该会有一个问题，那就是——在规范中存在的“引用”到底有什么用？它对我们的编程有什么实际的影响呢？

当然，除了已经提及过的delete 0和obj.x之外，在今后的课程中，我还会与你讨论这个“引用”的其它应用场景。而今天的内容，就从标题来看，若是我要说与这个“引用”有关，你说不定得跳起来说我无知；但若说是跟“引用”无关的话呢，我觉得又不能把标题中的这一行代码解释清楚。

为什么这行代码看起来与规范类型中的“引用”无关呢？因为这行代码出现的时候，连ECMAScript这个规范都不存在。

我大概是在JavaScript 1.2左右的时代就接触到这门语言，在我写的最早的一些代码中就使用过它，并且——没错，你一定知道的：它能执行！

有很多的原因会促使你在JavaScript写出表达式连等这样的代码。从C/C++走过来的程序员对这样的一行代码是不会陌生的。它能用，而且结果也与你的预期会一致，例如：

```
var x = y = 100;
console.log(x); // 100
console.log(y); // 100
```

它既没错、又好用，还很酷，你说我们为什么不用它呢？然而很不幸，这行代码可能是JavaScript中最复杂和最容易错用的表达式了。

所以今天我要和你一起好好地盘盘它。

声明

至今为止，除标签声明之外，JavaScript中一共只有六条声明用的语句。注意，所有真正被定义“声明”的语法结构都一定是“语句”，并且都用于声明一个或多个标识符。这里的标识符包括变量、常量等。

严格意义上讲，JavaScript只有变量和常量两种标识符，六条声明语句中：

- **let** *x* …

声明变量x。不可在赋值之前读。

- **const** *x* …

声明常量x。不可写。

- **var** *x* …

声明变量x。在赋值之前可读取到undefined值。

- **function** *x* …

声明变量x。该变量指向一个函数。

- **class** *x* …

声明变量x。该变量指向一个类（该类的作用域内部是处理严格模式的）。

- **import** …

导入标识符并作为常量（可以有多种声明标识符的模式和方法）。

除了这六个语句之外，还有两个语句有潜在的声明标识符的能力，不过它们并不是严格意义上的声明语句（声明只是它们的语法效果）。这两个语句是指：

- **for** (***var***|***let***|***const*** *x* …) …

for语句有多种语法来声明一个或多个标识符，用作循环变量。

- **try** … **catch** (*x*) …

catch子句可以声明一个或多个标识符，用作异常对象变量。

总的来说，除上述的语法，用户是没有其它方式来在当前的代码上下文中“声明”出一个标识符来的。而我之所以在这里严格强调这一“汇总性的”结果，是因为下面的一个简单论断，所有的“声明”：

- 都意味着JavaScript将可以通过“静态”语法分析发现那些声明的标识符；
- 标识符对应的变量/常量“一定”会在用户代码执行前就已经被创建在作用域中。

这个标题中的`var x`就是一个声明。在这个声明的后半部分，使用“=”这个符号引导了一个初始化语法——通常情况下可以将它理解为一个赋值运算。

从读取值到赋值

声明是在语法分析阶段就处理的，并且因此它会使得当前代码上下文在正式执行之前就拥有了被声明的标识符，例如`x`。

这其实非常有趣，因为这表明JavaScript虽然被称为是“动态语言”，但确实是拥有静态语义的。而在JavaScript的早期，这个静态语义其实并没有处理得太好，一个典型的问题就是所谓的“变量提升”。也就是可以在变量声明之前访问该变量。例如：

```
console.log(x); // undefined
var x = 100;
console.log(x); // 100
```

这个“变量提升”还包括“变量被创建于声明它的语法块”之外的意思，但这并不是这里要讨论的内容，我会在今后再讲它。在今天的课程里，你只需要留意这个变量的读写过程就好了。那么，关于读取值，之前声明的变量与常量又有什么不同呢？

如上面已经说过的，由于标识符是在用户代码执行之前就已经由静态分析得到，并且创建在环境中，因此let声明的变量和var声明的变量在这一点上没有不同：它们都是在读取一个“已经存在的”标识符名。例如：

```
var y = "outer";
function f() {
  console.log(y); // undefined
  console.log(x); // throw a Exception
  let x = 100;
  var y = 100;
```

```
  ...
}
```

正是由于`var y`所声明的那个标识符在函数f()创建（它自己的闭包）时就已经存在，所以才阻止了`console.log(y)`访问全局环境中的`y`。类似的，`let x`所声明的那个`x`其实也已经存在f()函数的上下文环境中。访问它之所以会抛出异常（Exception），不是因为它不存在，而是因为这个标识符被拒绝访问了。

在ECMAScript 6之后出现的`let/const`变量在“声明（和创建）一个标识符”这件事上，与`var`并没有什么不同，只是JavaScript拒绝访问还没有绑定值的`let/const`标识符而已。

回到ECMAScript 6之前：JavaScript是允许访问还没有绑定值的`var`所声明的标识符的。这种标识符后来统一约定称为“变量声明（varDelcs）”，而“let/const”则称为“词法声明（lexicalDecls）”。JavaScript环境在创建一个“变量名（varName in varDecls）”后，会为它初始化绑定一个undefined值，而”词法名字（lexicalNames）”在创建之后就没有这项待遇，所以它们在缺省情况下就是“还没有绑定值”的标识符。

> NOTE：6种声明语句中的函数是按varDecls的规则声明的；类的内部是处于严格模式中，它的名字是按let来处理的，而import导入的名字则是按const的规则来处理的。所以，所有的声明本质上只有三种处理模式：var变量声明、let变量声明和const常量声明。

所以，标题中的`var x = ...`在语义上就是为变量x绑定一个初值。在具体的语言环境中，它将被实现为一个赋值操作。

赋值

如果是在一门其它的（例如编译型的）语言中，“为变量x绑定一个初值”就可能实现为“在创建环境时将变量x指向一个特定的初始值”。这通常是静态语言的处理方法，然而，如前面说过的，JavaScript是门动态的语言，所以它的“绑定初值”的行为是通过动态的执行过程来实现的，也就是赋值操作。

那么请你仔细想想，一个赋值操作在语法上怎么表达呢？例如：

```
变量名 = 值
```

这样对吗？不对！在JavaScript中，这样讲是非常不正确的。正确的说法是：

```
lRef = rValue
```

也就是将右操作数（的值）赋给左操作数（的引用）。它的严格语法表达是：

> *LeftHandSideExpression* `=` | ***AssignmentOperator*** > *AssignmentExpression*

也就是说，在JavaScript中，一个赋值表达式的左边和右边其实“都是”表达式！

向一个不存在的变量赋值

接下来我要给你介绍的是从JavaScript 1.0开始就遗留下来的一个巨坑，也就是所谓的“变量泄漏”问题。这在早期的JavaScript中的确是一个好用的特性：如果你向一个不存在的变量名赋值，那么JavaScript会在全局范围内创建它。

也就是说，代码中不需要显式地声明一个变量了，变量可以随用随声明，也不用像后来的`let`语句一样，还要考虑在声明语句之前能不能访问的问题了。这非常简单，在少量的代码中也相当易用。

但是，如果代码规模扩大，变成百千万行代码，那么“**一个全局变量是在哪里声明和创建的**”就变成一个非常要紧的问题。

如果随时都可能泄露一个代码给全局，或者随时都可能因为忘记本地的声明而读写了全局变量，那对调试除错将是一场灾难。另外，晚一些出现的运行期优化技术也不能很好地处理这种情况。所以从ECMAScript5开始的严格模式就禁止了这种特性，试图避免用户将变量泄露到全局环境。

然而现实中，即使在严格模式下这种漏露也未能避免。这称为“**间接执行**”，这将是另一个巨大的议题，并且是ECMAScript6之后开始的一种新的机制。但是现在这里发生的事情，也就是这个“向不存在的变量赋值”的问题，是从JavaScript 1.0时代就遗留下来的问题，也是ECMAScript为JavaScript填补的最大设计漏洞之一。

那么，在具体技术细节上，这个变量声明是如何发生的呢？

事实上，这是因为在早期设计中，JavaScript的全局环境是引擎使用一个称为“**全局对象**”东西管理起来的。

这个全局对象几乎类似或完全等同于一个普通对象。只不过，JavaScript引擎将全局的一些缺省对象、运行期环境的原生对象等东西都初始化在这个全局对象的属性中，并使用这个对象创建了一个称为“**全局对象闭包**”的东西，从而得到了JavaScript的全局环境。

早期的JavaScript的引擎实现非常简洁，许多基础的技术组件都是直接复用的，例如这里的所谓全局环境、全局闭包，或者全局对象的实现方法，就与“**with语句**”的效果完全相同——他们是相互复用的。

当向一个不存在的变量赋值的时候，由于全局对象的属性表是可以动态添加的，因此JavaScript将变量名作为属性名添加给全局对象。而访问所谓全局变量时，就是访问这个全局对象的属性。因此，实际效果就变成了“可以动态地向全局环境中添加一个变量”。并且，显然地，我们在第一讲已经讲过这个结果——你可以删除掉这个动态添加的“变量”，因为本质上就是在删除全局对象的属性。

那么现在（我是指在ECMAScript6之后）的JavaScript的全局环境有什么不同吗？

为了兼容旧的JavaScript语言设计，现在的JavaScript环境仍然是通过将全局对象初始化为这样的一个全局闭包来实现的。但是为了得到一个“尽可能”与其它变量环境相似的声明效果（varDecls），ECMAScript规定在这个全局对象之外再维护一个变量名列表（varNames），所有在静态语法分析期或在eval()中使用var声明的变量名就被放在这个列表中。然后约定，这个变量名列表中的变量是“直接声明的变量”，不能使用delete删除。

于是，我们得到了这样的一种结果：

```
> var a = 100;
> x = 200;

# `a`和`x`都是global的属性
> Object.getOwnPropertyDescriptor(global, 'a');
{ value: 100, writable: true, enumerable: true, configurable: false }
> Object.getOwnPropertyDescriptor(global, 'x');
{ value: 200, writable: true, enumerable: true, configurable: true }

# `a`不能删除, `x`可以被删除
> delete a
false
> delete x
true

# 检查
> a
100
> x
ReferenceError: x is not defin
```

所以，表面看起来“泄漏到全局的变量”与使用var声明的都是全局变量，并且都实现为global的属性，但事实上它们是不同的。并且当var声明发生在eval()中的时候，这一特性又还有所不同，例如：

```
# 使用eval声明
> eval('var b = 300');

# 它的性质是可删除的
> Object.getOwnPropertyDescriptor(global, 'b').configurable;
true

# 检测与删除
> b
300
> delete b
true
> b
ReferenceError: b is not define
```

这种情况下使用var声明的变量名尽管也会添加到varNames列表，但它也可以从varNames中移除（这是唯一一种能从varNames中移除项的特例，而lexicalNames中的项是不可移除的）。

发生了什么？

所以，现在回到今天讨论的这行代码var x = y = 100，在这行代码中，等号的右边是一个表达式y = 100，它发生了一次“向不存在的变量赋值”，所以它隐式地声明了一个全局变量y，并赋值为100。

而一个赋值表达式操作本身也是有“结果（Result）”的，它是右操作数的值。注意，这里是“值”而非“引用”，例如下面的测试中的a将是一个函数，而不是带着“this对象”信息的方法：

```
// 调用obj.f()时将检测this是不是原始的obj
> obj = { f: function() { return this === obj } };

// false，表明赋值表达式的“结果(result)”只是右侧操作数的值，即函数f
> (a = obj.f)();
false
```

到现在为止，我们讲述了整个语句的过程，也就是说，由于“y = 100”的结果是100，所以该值将作为初始值赋值“变量x”。并且，从语义上来说，这是变量“x”的初始绑定。

之所以强调这一点，是因为相同的分析过程也可以用在const声明上，而const声明是只有一次绑定的，常量的初始绑定也是通

过“执行赋值过程”来实现的。

知识回顾

- var等声明语句总是在变量作用域（变量表）或词法作用域中静态地声明一个或多个标识符。
- 全局变量的管理方式决定了“向一个不存在的变量赋值”所导致的变量泄漏是不可避免的。
- 动态添加的“var声明”是可以删除的，这是唯一能操作varNames列表的方式（不过它并不存在多少实用意义）。
- 变量声明在引擎的处理上被分成两个部分：一部分是静态的、基于标识符的词法分析和管理，它总是在相应上下文的环境构建时作为名字创建的；另一部分是表达式执行过程，是对上述名字的赋值，这个过程也称为绑定。
- 这一讲标题里的这行代码中，x和y是两个不同的东西，前者是声明的名字，后者是一个赋值过程可能创建的变量名。

思考题

根据今天讲解的内容，我希望你可以尝试回答以下问题：

- 严格来说，声明不是语句。但是，是哪些特性决定了声明不是“严格意义上的”语句呢？

在下一讲中我会来讲一讲JavaScript社区中的一个历史悬案，这桩悬案与今天讨论的这行代码的唯一区别在于：它不是声明语句，而是赋值表达式。