你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`()`指示了参数，而`{}`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果

有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数x
- 执行体x

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是“非惰性求值”的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种

表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个`x`来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量`x`的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将x创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值undefined，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
console.log(x); // ReferenceError: x is not defined
let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值undefined，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是(x = x) => x这样的语法并不违例，而是第二个x导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的x => x。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据x映射成x'。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如f(a=100)；绑定参数的过程执行于函数（的闭包）之内，例如function foo(x=100) ..。
2. x=>x在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数$_x$
- 执行体$_x$

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
```

```
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个"结果"总是存在"值和引用"两种表达形式，所以JavaScript在这里约定"传值"。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）"绑定实际传入的值"。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致"重新声明"一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行"绑定实际传入的参数"的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是"值"，所以没有任何需要引用其它数据的显式执行过程。"值"是数据，而非逻辑。

所以，对于简单参数来说，是没有"求值过程"发生于函数的调用界面上的。然而，对于下面例子中这样的"非简单参数"函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在"绑定实际传入的参数"时，就需要执行一个"x = 100"的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的"参数x"是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的x也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，x显然是声明过的。事实上，这也是两种不同的登记过程（"直接arguments绑定"与"初始器赋值"）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为"**可变绑定**（*Mutable Binding*）"。

"可变"是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
```

```
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x’, output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function`

foo(x=100) ..。
2. x=>x在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法( )指示了参数，而{ }指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数f()有且仅有一个；而在执行后，arr[]中将存在该函数f()的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数$_x$
- 执行体$_x$

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释

什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个`x`来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量`x`的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称

为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。

但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数`x`
- 执行体`x`

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的x也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，x显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数f()完全相同的异常。也就是说，在(x = x) => x中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个x，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将x创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值undefined，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值undefined，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是(x = x) => x这样的语法并不违例，而是第二个x导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的x => x。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据x映射成x'。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。

NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ...`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数`x`
- 执行体`x`

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除

绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是“非惰性求值”的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于“非惰性求值”，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的x也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，x显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个x，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将x创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个x导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据x映射成x'。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
```

```
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数$_x$
- 执行体$_x$

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换

到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是“非惰性求值”的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个`x`来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于“非惰性求值”，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量`x`的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的x也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，x显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个x，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将x创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个x导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据x映射成x'。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
```

```
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数`x`
- 执行体`x`

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
```

```
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在

此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数$_x$
- 执行体$_x$

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是“非惰性求值”的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个`x`来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于“非惰性求值”，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量`x`的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数$_x$
- 执行体$_x$

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也

（首先）代表了这样两个语义组件：

- 参数x
- 执行体x

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是“非惰性求值”的
f(a = 100);
```

在这个示例中，传入函数f()的将是赋值表达式a = 100完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是f(100)。

回顾这个过程，请你注意一个问题：a = 100这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数f()的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如function f(x)中的形式参数名x）“绑定实际传入的值”。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于“非惰性求值”，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的console.log(x)将显示变量x的传入参数值100，而var x = 200;并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数x
- 执行体x

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是“非惰性求值”的
f(a = 100);
```

在这个示例中，传入函数f()的将是赋值表达式a = 100完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是f(100)。

回顾这个过程，请你注意一个问题：a = 100这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数f()的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如function f(x)中的形式参数名x）“绑定实际传入的值”。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
```

```
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致"重新声明"一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行"绑定实际传入的参数"的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是"值"，所以没有任何需要引用其它数据的显式执行过程。"值"是数据，而非逻辑。

所以，对于简单参数来说，是没有"求值过程"发生于函数的调用界面上的。然而，对于下面例子中这样的"非简单参数"函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在"绑定实际传入的参数"时，就需要执行一个"x = 100"的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的"参数x"是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（"直接arguments绑定"与"初始器赋值"）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为"**可变绑定**（*Mutable Binding*）"。

"可变"是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行"初始器赋值"时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是"可变绑定（*Mutable Binding*）"。并且，对于var/let来说，一开始的时候它们其实都是"无初值的绑定"。只不过JavaScript在处理var语句声明的变量时，将这个"绑定（Binding）"赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些"var变量"。而对应的，let语句声明的变量没有"缺省地"赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成"可变绑定"，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的"初始器"来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的"可变绑定"还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数`x`
- 执行体`x`

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个`x`来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内

变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致"重新声明"一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行"绑定实际传入的参数"的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是"值"，所以没有任何需要引用其它数据的显式执行过程。"值"是数据，而非逻辑。

所以，对于简单参数来说，是没有"求值过程"发生于函数的调用界面上的。然而，对于下面例子中这样的"非简单参数"函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在"绑定实际传入的参数"时，就需要执行一个"x = 100"的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的"参数x"是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（"直接arguments绑定"与"初始器赋值"）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为"**可变绑定**（*Mutable Binding*）"。

"可变"是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行"初始器赋值"时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是"可变绑定（*Mutable Binding*）"。并且，对于var/let来说，一开始的时候它们其实都是"无初值的绑定"。只不过JavaScript在处理var语句声明的变量时，将这个"绑定（Binding）"赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些"var变量"。而对应的，let语句声明的变量没有"缺省地"赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
```

```
let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`()`指示了参数，而`{}`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果

有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数x
- 执行体x

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是“非惰性求值”的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种

表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个`x`来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量`x`的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将x创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值undefined，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
console.log(x); // ReferenceError: x is not defined
let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值undefined，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是(x = x) => x这样的语法并不违例，而是第二个x导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的x => x。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据x映射成x'。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如f(a=100)；绑定参数的过程执行于函数（的闭包）之内，例如function foo(x=100) ..。
2. x=>x在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数$_x$
- 执行体$_x$

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
```

```
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个"结果"总是存在"值和引用"两种表达形式，所以JavaScript在这里约定"传值"。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）"绑定实际传入的值"。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致"重新声明"一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行"绑定实际传入的参数"的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是"值"，所以没有任何需要引用其它数据的显式执行过程。"值"是数据，而非逻辑。

所以，对于简单参数来说，是没有"求值过程"发生于函数的调用界面上的。然而，对于下面例子中这样的"非简单参数"函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在"绑定实际传入的参数"时，就需要执行一个"x = 100"的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的"参数x"是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的x也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，x显然是声明过的。事实上，这也是两种不同的登记过程（"直接arguments绑定"与"初始器赋值"）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为"**可变绑定**（*Mutable Binding*）"。

"可变"是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
```

```
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x’, output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function`

foo(x=100) ..。
2. x=>x在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法( )指示了参数，而{ }指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数f()有且仅有一个；而在执行后，arr[]中将存在该函数f()的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数$_x$
- 执行体$_x$

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释

什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个`x`来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量`x`的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的`x`也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，`x`显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称

为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数`f()`完全相同的异常。也就是说，在`(x = x) => x`中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个`x`，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将`x`创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值`undefined`，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值`undefined`，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是`(x = x) => x`这样的语法并不违例，而是第二个`x`导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的`x => x`。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据`x`映射成`x'`。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

> NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。
>
> NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。

但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。

你好，我是周爱民。

在运行期，语句执行和特殊的可执行结构都不是JavaScript的主角，多数情况下，它们都只充当过渡角色而不为开发人员所知。我相信，你在JavaScript中最熟悉的执行体一定是**全局代码**，以及**函数**。

而今天，我要为你解析的就是函数的执行过程。

如同在之前分析语句执行的时候与你谈到过的，语句执行是命令式范型的体现，而函数执行代表了JavaScript中对函数式范型的理解。厘清这样的基础概念，对于今天要讨论的内容来说，是非常重要和值得的。

很多人会从**对象**的角度来理解JavaScript中的函数，认为“函数就是具有[[Call]]私有槽的对象”。这并没有错，但是这却是从静态视角来观察函数的结果。

要知道函数是执行结构，那么执行过程发生了什么呢？这个问题从对象的视角是既观察不到，也得不到答案的。并且，事实上如果上面这个问题问得稍稍深入一点，例如“对象的方法是怎么执行的呢”，那么就必须要回到“函数的视角”，或者运行期的、动态的角度来解释这一切了。

函数的一体两面

用静态的视角来看函数，它就是一个函数对象（函数的实例）。如果不考虑它作为对象的那些特性，那么函数也无非就是“用三个语义组件构成的实体”。这三个语义组件是指：

1. 参数：函数总是有参数的，即使它的形式参数表为空；
2. 执行体：函数总是有它的执行过程，即使是空的函数体或空语句；
3. 结果：函数总是有它的执行的结果，即使是undefined。

并且，重要的是“这三个语义组件缺一不可”。晚一点我会再来帮你分析这个观点。现在你应该关注的问题是——我为什么要在此之前强调“用静态的视角来看”。

在静态的语义分析阶段，函数的三个组件中的两个是显式的，例如在下面的声明中：

```
function f() {
  ...
}
```

语法`( )`指示了参数，而`{ }`指示了执行体，并且，我们隐式地知道该函数有一个结果。这也是JavaScript设计经常被批判的一处：由于没有静态类型声明，所以我们也无法知道函数返回何种结果。当我们把这三个部分构成的一个整体看作执行体的时候：

```
(function f() {
  ...
})
```

那么它的结果是一个函数类型的“数据”。这在函数式语言中称为“函数是第一类型的”，也就是说函数既是可以执行的**逻辑**，也同时是可以被逻辑处理的**数据**。

函数作为数据时，它是“原始的函数声明”的一个实例（注意这里并不强调它是对象实例）。这个实例必须包括上述三个语义组件中的两个，即**参数**与**执行体**。否则，它作为实例将是不完整的、不能准确复现原有的函数声明的。为了达到这个目的，JavaScript为每个实例创建了一个闭包，并且作为上述“函数类型的‘数据’”的实际结果。例如：

```
var arr = new Array;
for (var i=0; i<5; i++) arr.push(function f() {
  // ...
});
```

在这个例子中，静态的函数`f()`有且仅有一个；而在执行后，`arr[]`中将存在该函数`f()`的5个实例，每一个称为该函数的一个运行期的闭包。它们各各不同，例如：

```
> arr[0] === arr[1]
false
```

所以简而言之，任何时候只要用户代码引用一次这样的函数（的声明或字面量），那么它就会拿到该函数的一个闭包。注意，得到这个闭包的过程与是否调用它是无关的。

两个语义组件

上面说过，这样的闭包有两个语义组件：参数和执行体。在创建这个闭包时，它们也将同时被实例化。

这样做的目的，是为了保证每个实例/闭包都有一个自己独立的运行环境，也就是**运行期上下文**。JavaScript中的闭包与运行环境并没有明显的语义差别，唯一不同之处，仅在于这个“运行环境”中每次都会有一套新的“参数”，且执行体的运行位置（如果有的话）被指向函数代码体的第一个指令。

然而，你或许会问：我为什么要如此细致地强调这一点，巨细无遗地还原创建这样的环境的每一步呢？

这样的小心和质疑是必要的！如果你真的这样问了，那么非常感谢，你提出了“函数”的一个关键假设：它可以是多次调用的。

这显然是废话。

但是，如果你将它与之前讨论过的for循环对照起来观察的话，就会发现一个事实：函数体和for的循环体（这些用来实现逻辑复用的执行结构）的创建技术，是完全一样的！

也就是说，命令式语句和函数式语言，是采用相同的方式来执行逻辑的。只不过前者把它叫做_iteratorEnv_，是_loopEnv_的实例；后者把它叫做闭包，是函数的实例。

再往源头探究一点：导致for循环需要多个_iteratorEnv_实例的原因，在于循环语句试图在多个迭代中复用参数（迭代变量），而函数这样做的目的，也同时是为了处理这些参数（形式参数表）的复用而已。

所以，闭包的作用与实现方法都与“for循环”中的迭代环境没有什么不同。同样地，对于这一讲的标题中的这行代码来说，它也（首先）代表了这样两个语义组件：

- 参数$_x$
- 执行体$_x$

在闭包创建时，参数x将作为闭包（作用域/环境）中的名字被初始化——这个过程中“参数x”只作为名字或标识符，并且“将会在”闭包中登记一个名为“x”的变量；按照约定，它的值是undefined。并且，还需要强调的是，这个过程是引擎为闭包初始化的，发生于用户代码得到这个闭包之前。

然而所谓“参数的登记过程”很重要吗？当然重要。

简单参数类型

完整而确切地说，这一讲标题中的函数是一个“简单参数类型的箭头函数”。而下面这个就不“简单”了：

```
(x = x) => x;
```

在ECMAScript 6之前的函数声明中，它们的参数都是“简单参数类型”的。在ECMAScript 6之后，凡是在参数声明中使用了缺省参数、剩余参数和模板参数之一的，都不再是“简单的”（*non-simple parameters*）。在具体实现中，这些新的参数声明意味着它们会让函数进入一种特殊模式，由此带来三种限制：

1. 函数无法通过显式的"use strict"语句来切换到严格模式，但能接受它被包含在一个严格模式的语法块中（从而隐式地切换到严格模式）；
2. 无论是否在严格模式中，函数参数声明都将不接受“重名参数”；
3. 无论是否在严格模式中，形式参数与arguments之间都将解除绑定关系。

这样处理的原因在于：在使用传统的简单参数时，只需要将调用该参数时传入的实际参数与参数对象（arguments）绑定就可以了；而使用“非简单参数”时，需要通过“初始器赋值”来完成名字与值的绑定。同样，这也是导致“形式参数与arguments之间解除绑定关系”的原因。

> NOTE 1: 两种绑定模式的区别在于：通常将实际参数与参数对象绑定时，只需要映射两个数组的下标即可，而“初始器赋值”需要通过名字来索引值（以实现绑定），因此一旦出现“重名参数”就无法处理了。

所以，所谓参数的登记过程，事实上还影响了它们今后如何绑定实际传入的参数。

传入参数的处理

要解释“参数的传入”的完整过程，得先解释**为什么“形式参数需要两种不同的登记过程”**。而在这所有一切之前，还得再解释什么是“**传入的参数**”。

首先，JavaScript的函数是“非惰性求值”的，也就是说在函数界面上不会传入一个延迟计算的求值过程，而是“积极地”传入已经求值的结果。例如：

```
// 一般函数声明
function f(x) {
  console.log(x);
}

// 表达式`a=100`是"非惰性求值"的
f(a = 100);
```

在这个示例中，传入函数`f()`的将是赋值表达式`a = 100`完成计算求值之后的结果。考虑到这个“结果”总是存在“值和引用”两种表达形式，所以JavaScript在这里约定“传值”。于是，上述示例代码最终执行到的将是`f(100)`。

回顾这个过程，请你注意一个问题：`a = 100`这行表达式执行在哪个上下文环境中呢？

答案是：在函数外（上例中是全局环境）。

接下来才来到具体调用这个函数`f()`的步骤中。而直到这个时候，JavaScript才需要向环境中的那些名字（例如`function f(x)`中的形式参数名x）“绑定实际传入的值”。对于这个x来说，由于参数与函数体使用同一个块作用域，因此如果函数参数与函数内变量同名，那么它们事实上将是同一个变量。例如：

```
function f(x) {
  console.log(x);
  var x = 200;
}
// 由于"非惰性求值"，所以下面的代码在函数调用上完全等义于上例中`f(a = 100)`
f(100);
```

在这个例子中，函数内的三个x实际将是同一个变量，因此这里的`console.log(x)`将显示变量x的传入参数值100，而`var x = 200;`并不会导致“重新声明”一个变量，仅仅是覆盖了既有的x。

现在我们回顾之前讨论的两个关键点：

1. 参数的登记过程发生在闭包创建的过程中；
2. 在该闭包中执行“绑定实际传入的参数”的过程。

意外

对于后面这个过程来说，如果参数是简单的，那么JavaScript引擎只需要简单地绑定它们的一个对照表就可以了。并且，由于所有被绑定的、传入的东西都是“值”，所以没有任何需要引用其它数据的显式执行过程。“值”是数据，而非逻辑。

所以，对于简单参数来说，是没有“求值过程”发生于函数的调用界面上的。然而，对于下面例子中这样的“非简单参数”函数声明来说：

```
function foo(x = 100) {
  console.log(x);
}
foo();
```

在“绑定实际传入的参数”时，就需要执行一个“x = 100”的计算过程。不同于之前的`f(a = 100)`，在这里的表达式`x = 100`将执行于这个新创建的闭包中。这很好理解，左侧的“参数x”是闭包中的一个语法组件，是初始化创建在闭包中的一个变量声明，因此只有将表达式放在这个闭包中，它才可以正确地完成计算过程。

然而这样一来，在下面这个示例中，表达式右侧的x也将是该闭包中的x。

```
f = (x = x) => x;
```

这貌似并没有什么了不起的，但真正使用它的时候，会触发一个异常：

```
> f()
ReferenceError: x is not defined
    at f (repl:1:10)
```

这是一个意外。

无初值的绑定

这个异常提示其实并不准确，因为在这个上下文环境（闭包）中，x显然是声明过的。事实上，这也是两种不同的登记过程（“直接arguments绑定”与“初始器赋值”）的主要区别之一。尽管在本质上，这两种登记过程所初始化的变量都是相同的，称为“**可变绑定**（*Mutable Binding*）”。

“可变”是指它们可以多次赋值，简单地说就是let/var变量。但显然地，上述的示例正好展示了var/let的两种不同性质：

```
function foo() {
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
}
foo();
```

由于let变量不能在它的声明语句之前（亦即是未初始化之前）访问，因此上例触发了与之前的箭头函数f()完全相同的异常。也就是说，在(x = x) => x中的三个x都是指向相同的变量，并且当函数在尝试执行“初始器赋值”时会访问第2个x，然而此时由于变量x是未赋值的，因此它就如同let变量一样不可访问，从而触发异常。

为什么在处理函数的参数表时要将x创建为一个let变量，而不是var变量呢？

事实上，这二者并没有区别，如之前我所讲过的，它们都是“可变绑定（*Mutable Binding*）”。并且，对于var/let来说，一开始的时候它们其实都是“无初值的绑定”。只不过JavaScript在处理var语句声明的变量时，将这个“绑定（Binding）”赋了一个初值undefined，因此你才可以在代码中自由、提前地访问那些“var变量”。而对应的，let语句声明的变量没有“缺省地”赋这个初值，所以才不能在第一行赋值语句之前访问它，例如：

```
  console.log(x); // ReferenceError: x is not defined
  let x = 100;
```

处理函数参数的过程与此完全相同：参数被创建成“可变绑定”，如果它们是简单参数则被置以初值undefined，否则它们就需要一个所谓的“初始器”来赋初值。也就是说，并非JavaScript要刻意在这里将它作为var/let变量之一来创建，而只是用户逻辑执行到这个位置的时候，所谓的“可变绑定”还没有来得及赋初值罢了。

然而，唯一在这个地方还存疑的是：**为什么不干脆就在“初始器”创建的时候，就赋一个初值undefined呢？**

说到这里，可能你也猜到了，因为在“缺省参数”的语法设计里面，undefined正好是一个有意义的值，它用于表明参数表指定位置上的形式参数是否有传入，所以参数undefined也就不能作为初值来绑定，这就导致了使用“初始器”的参数表中，所对应那些变量是一个“无初值的绑定”。

因此如果这个“初始器”（我是指在它初始化的阶段里面）正好也要访问变量自身，那么就会导致出错了。而这个出错过程也就与如下示例的代码是一样的，并且也导致一样的错误：

```
> let x = x;
ReferenceError: x is not defined
```

所以，最终的事实是(x = x) => x这样的语法并不违例，而是第二个x导致了非法访问“无初值的绑定”。

最小化的函数式语言示例

那么现在我再来为你解析一下标题中的x => x。

这行代码意味着一个最小化的函数。它包括了一个函数完整的三个语法组件：**参数**、**执行体**和**结果**，并且也包括了JavaScript实现这三个语法组件的全部处理过程——这些是我在这一讲中所讨论的全部内容。重要的是，它还直观地反映了“函数”的本质，就是“数据的转换”。也就是说，所有的函数与表达式求值的本质，都是将数据x映射成x'。

我们编写程序的这一行为，在本质上就是针对一个“输入（x, input/arguments）”，通过无数次的数据转换来得到一个最终的”输出（x', output/return）”。所有计算的本质皆是如此，所有的可计算对象也可以通过这一过程来求解。

因此，函数在能力上也就等同于全部的计算逻辑（等同于结构化程序思想中的“单入口->单出口”的顺序逻辑）。

箭头函数是匿名的，并且事实上所谓名字并不是函数在语言学中的重要特性。名字/标识符，是语法中的词法组件，它指代某个东西的抽象，但它本身既不是计算的过程（逻辑），也不是计算的对象（数据）。

那么，我接下来要说的是，**没有名字的函数在语言中的意义是什么呢？**

它既是**逻辑**，也是**数据**。例如：

```
let f = x => x;
let zero = f.bind(null, 0);
```

现在，zero既是一个逻辑，是可以执行的过程，它返回数值0；也是一个数据，它包含数值0。

NOTE 1：箭头函数与别的函数的不同之处在于它并不绑定“this”和“arguments”。此外，由于箭头函数总是匿名的，因此它也不会在环境中绑定函数名。

NOTE 2： ECMAScript 6之后的规范中，当匿名函数或箭头函数赋给一个变量时，它将会以该变量名作为函数名。但这种情况下，该函数名并不会绑定给环境，而只是出现在它的属性中，例如“*f.name*”。

知识回顾

现在我来为这一讲的内容做个回顾。

1. 传入参数的过程执行于函数之外，例如`f(a=100)`；绑定参数的过程执行于函数（的闭包）之内，例如`function foo(x=100) ..`。
2. `x=>x`在函数界面的两端都是值操作，也就是说input/output的都是数据的值，而不是引用。
3. 参数有两种初始化方法，它们根本的区别在于绑定初值的方式不同。
4. 闭包是函数在运行期的一个实例。

思考题

1. 表达式如何等价于上述计算过程？
2. 表达式与函数在抽象概念上的异同？
3. 试以表达式来实现标题中的箭头函数的能力。

欢迎你在进行深入思考后，与其他同学分享自己的想法，也让我有机会能听听你的收获。