你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

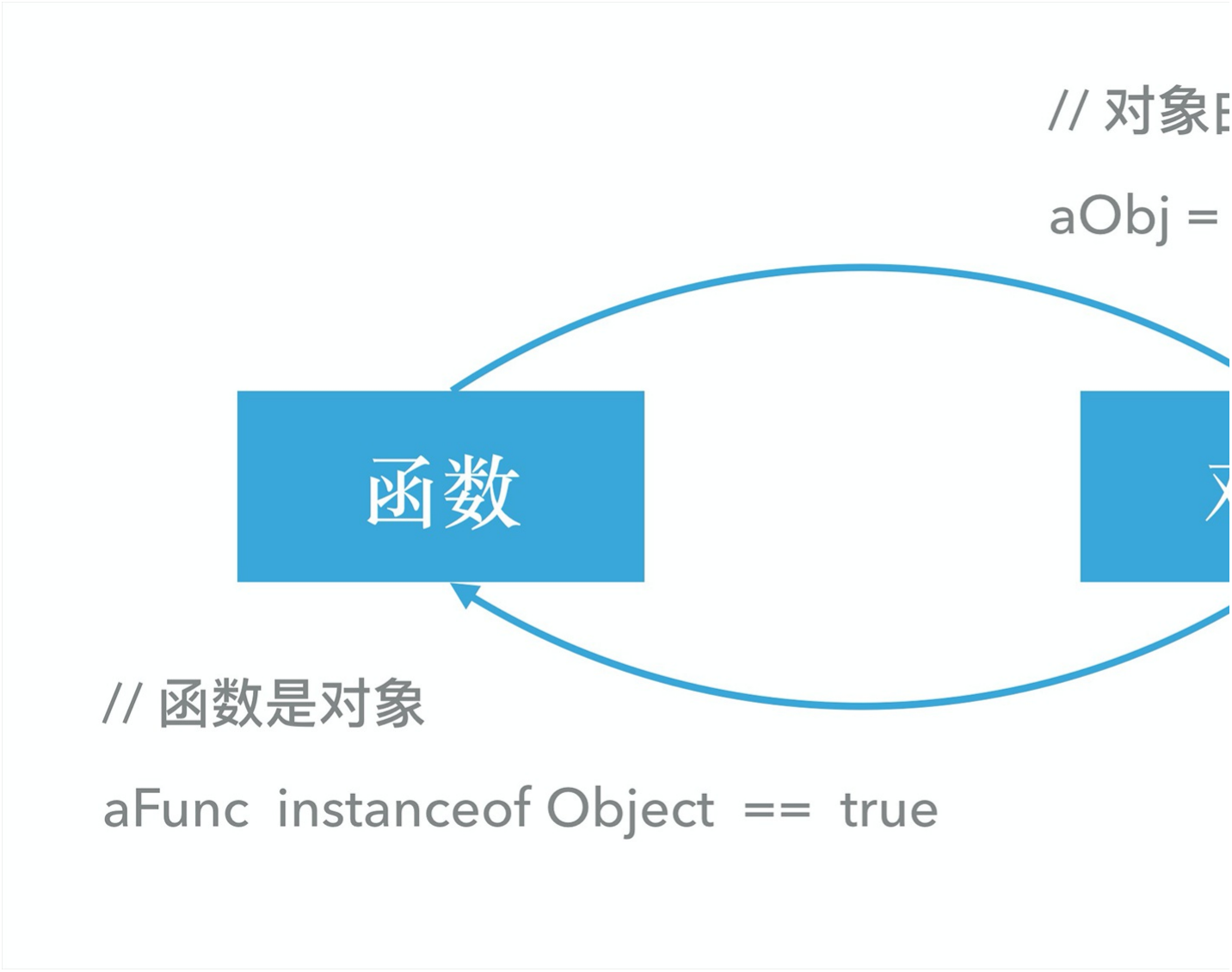
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

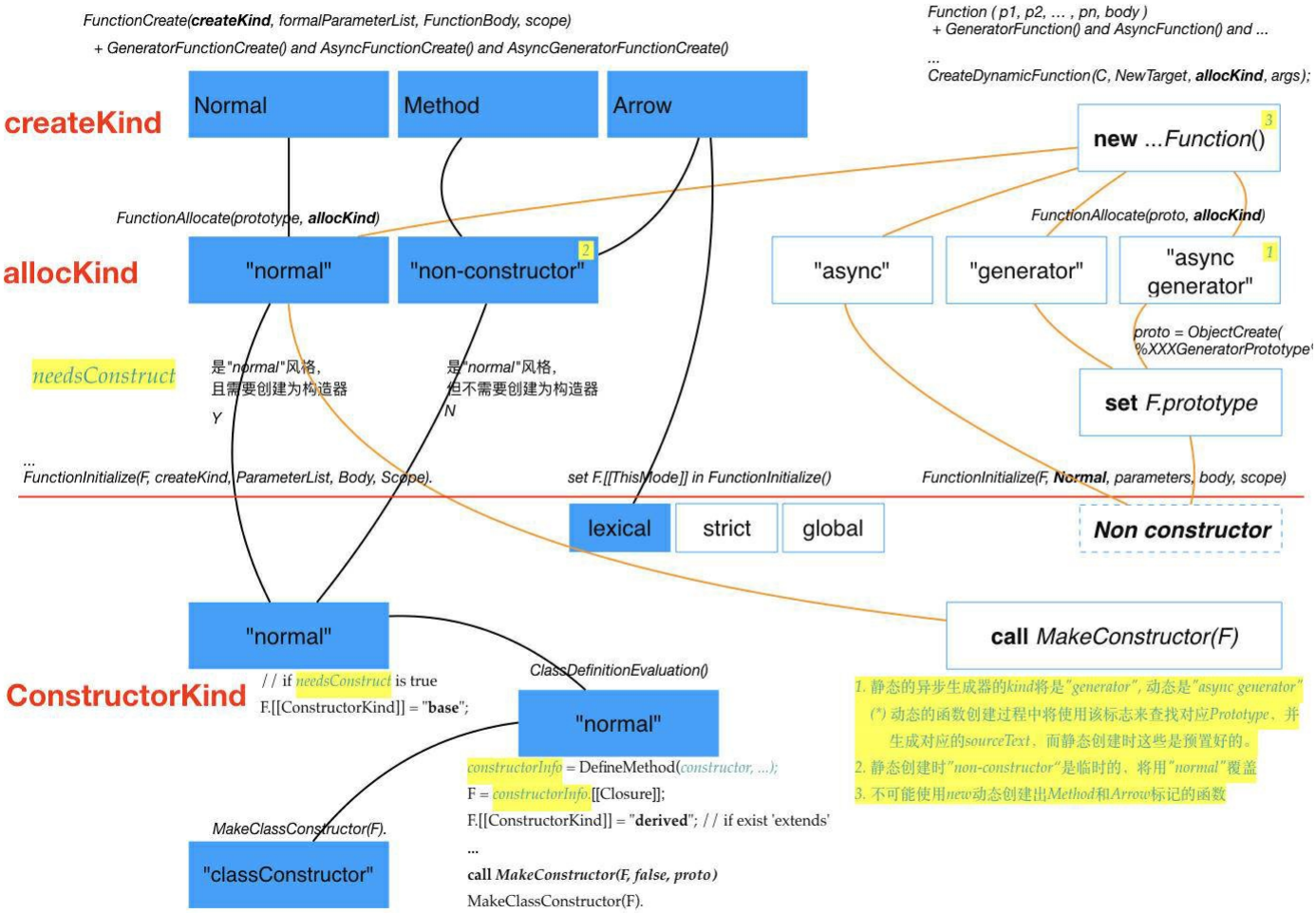
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货吐）。
by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

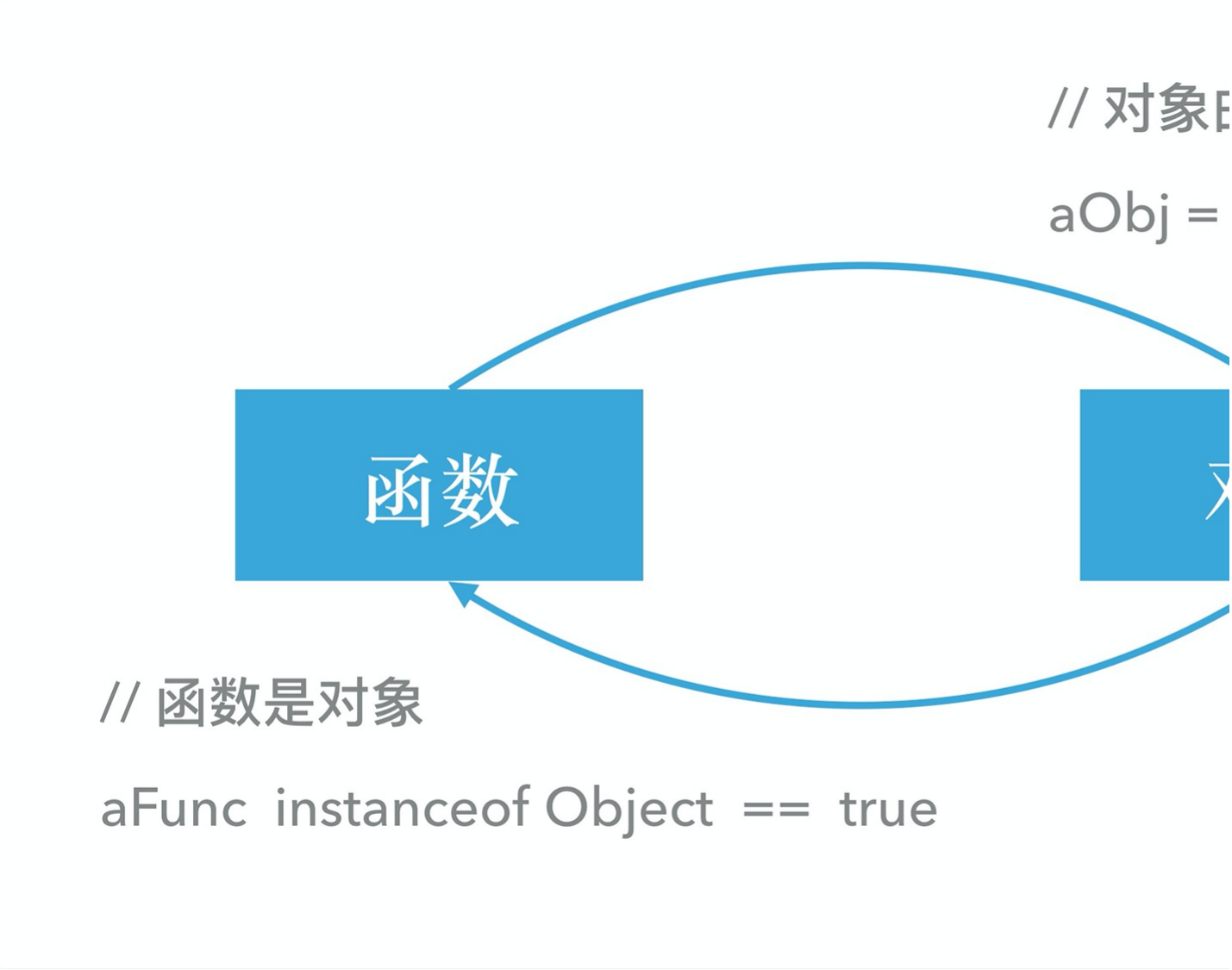
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



> NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

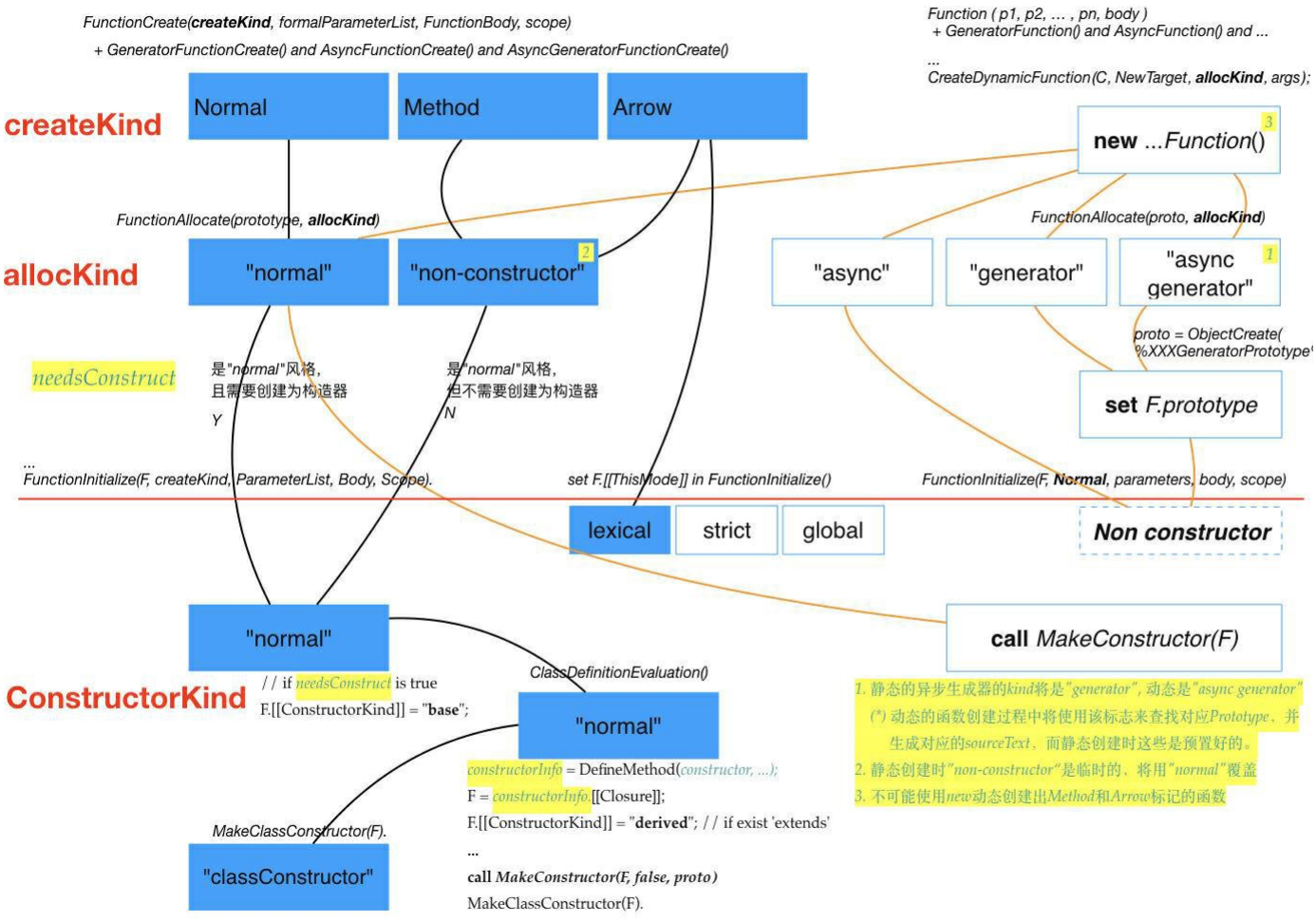
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及

内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

> NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

> NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

> NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货叱）。
> by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default ...）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用`eval()`，例如：

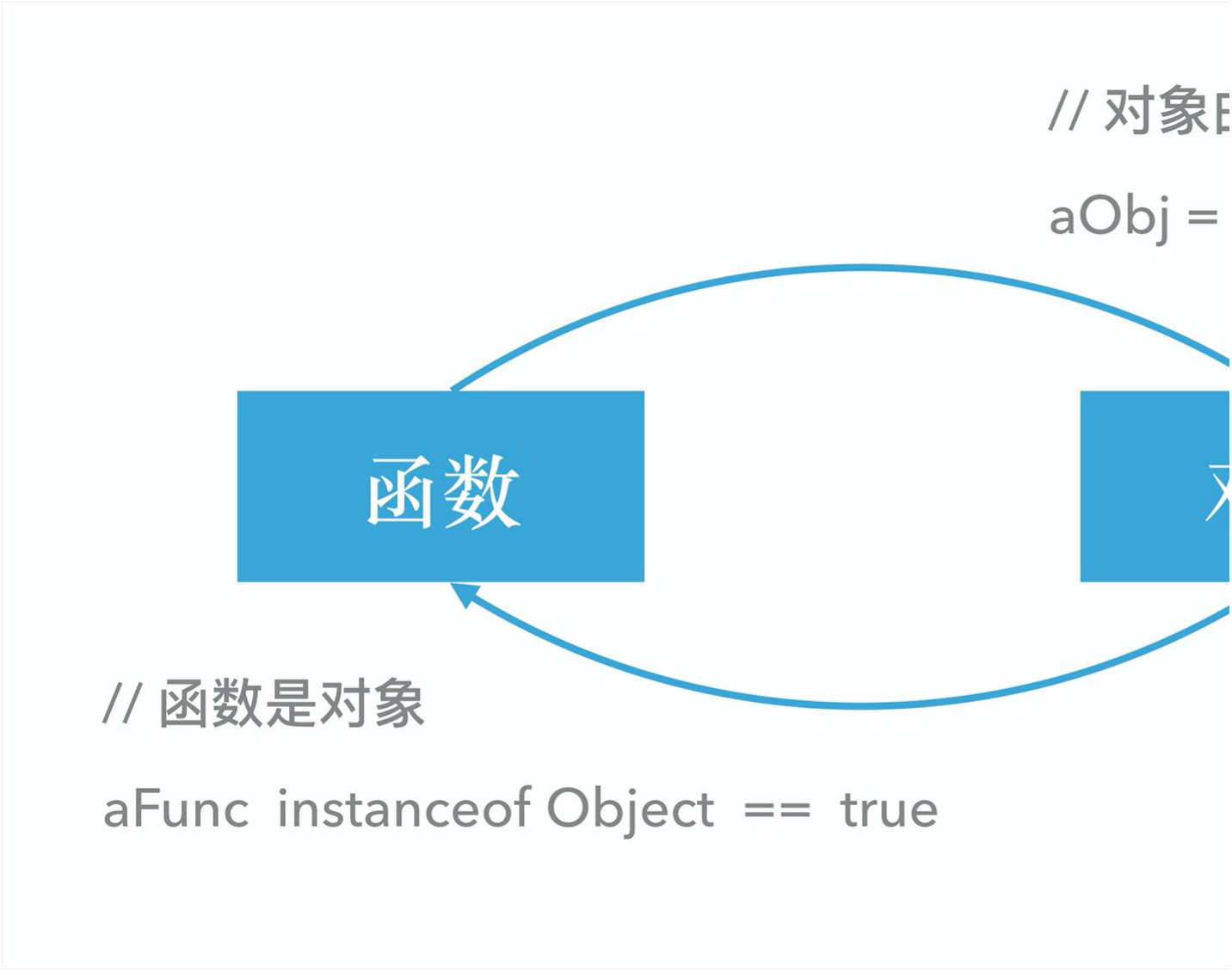
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中`Function()`是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向`Function()`，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

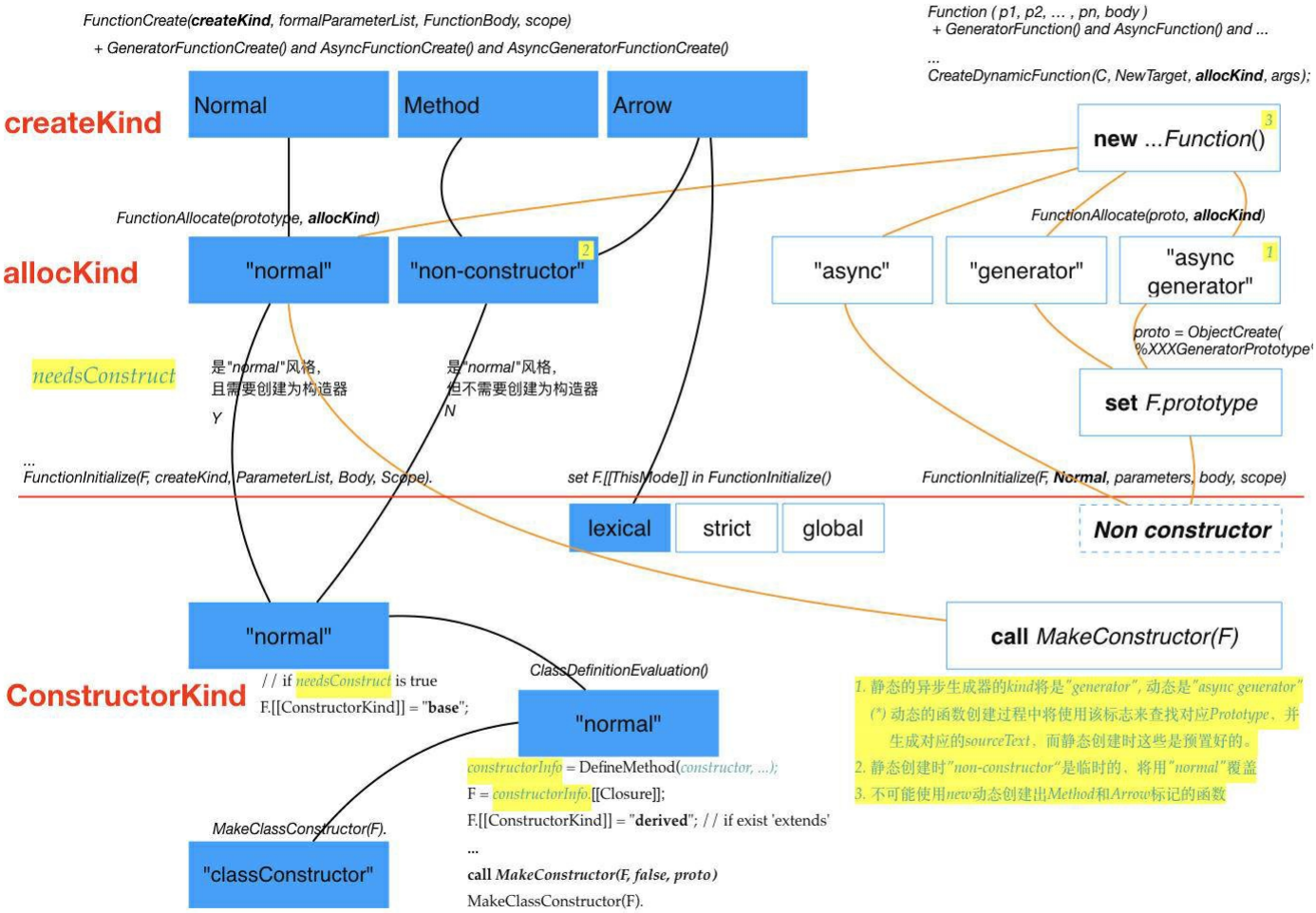
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货吐）。
by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

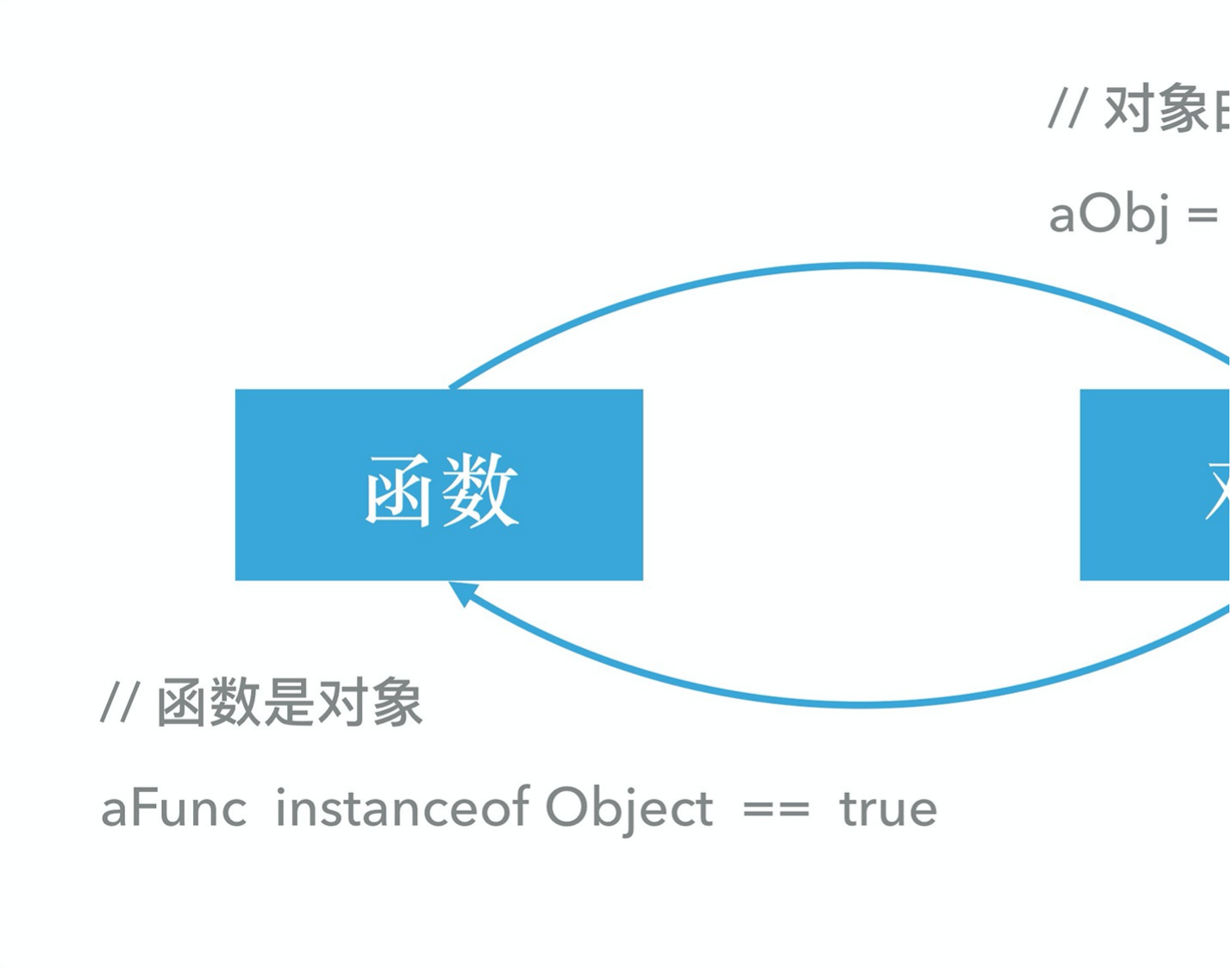
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



> NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

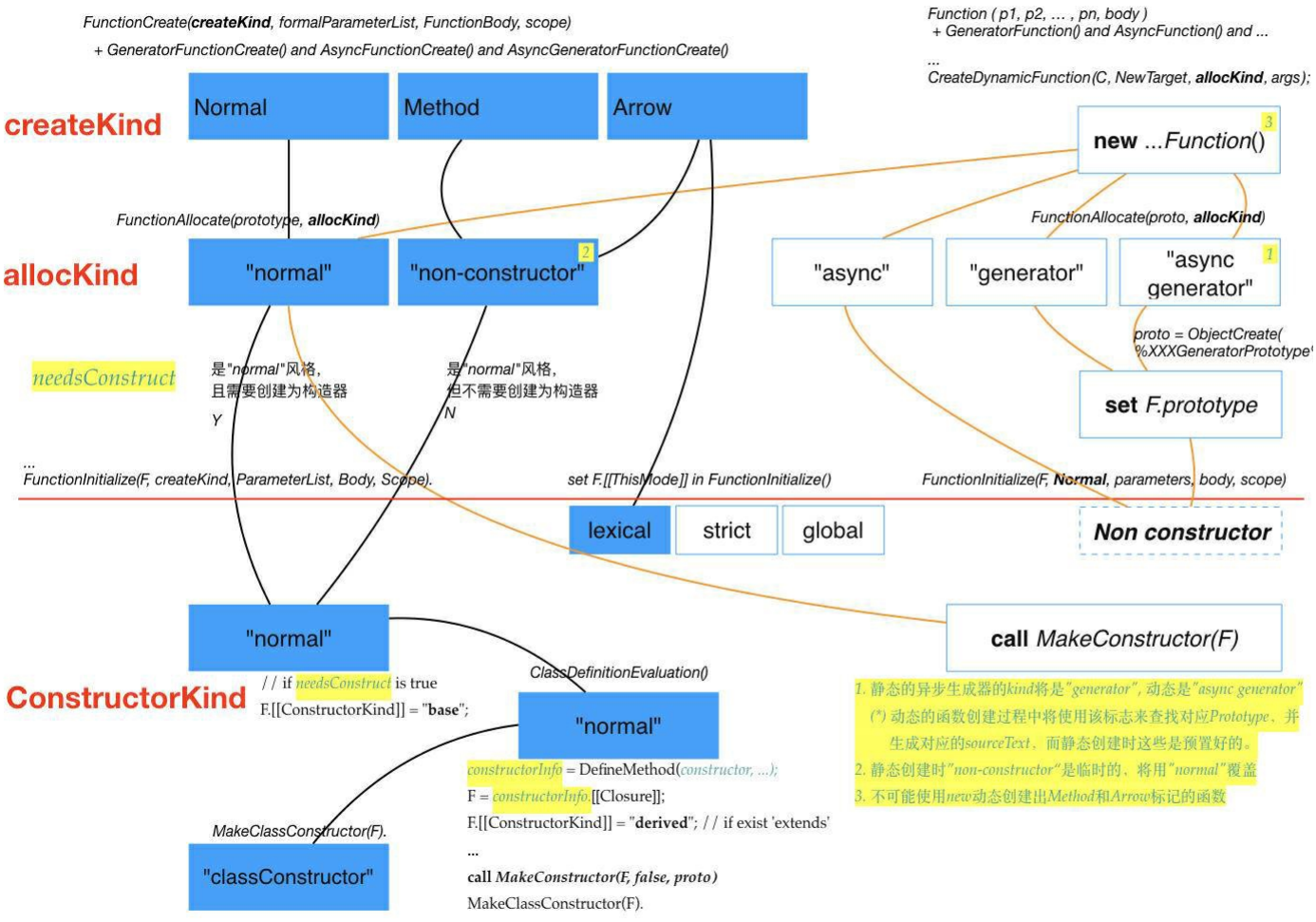
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及

内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

> NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

> NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

> NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货叱）。
> by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default ...）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用`eval()`，例如：

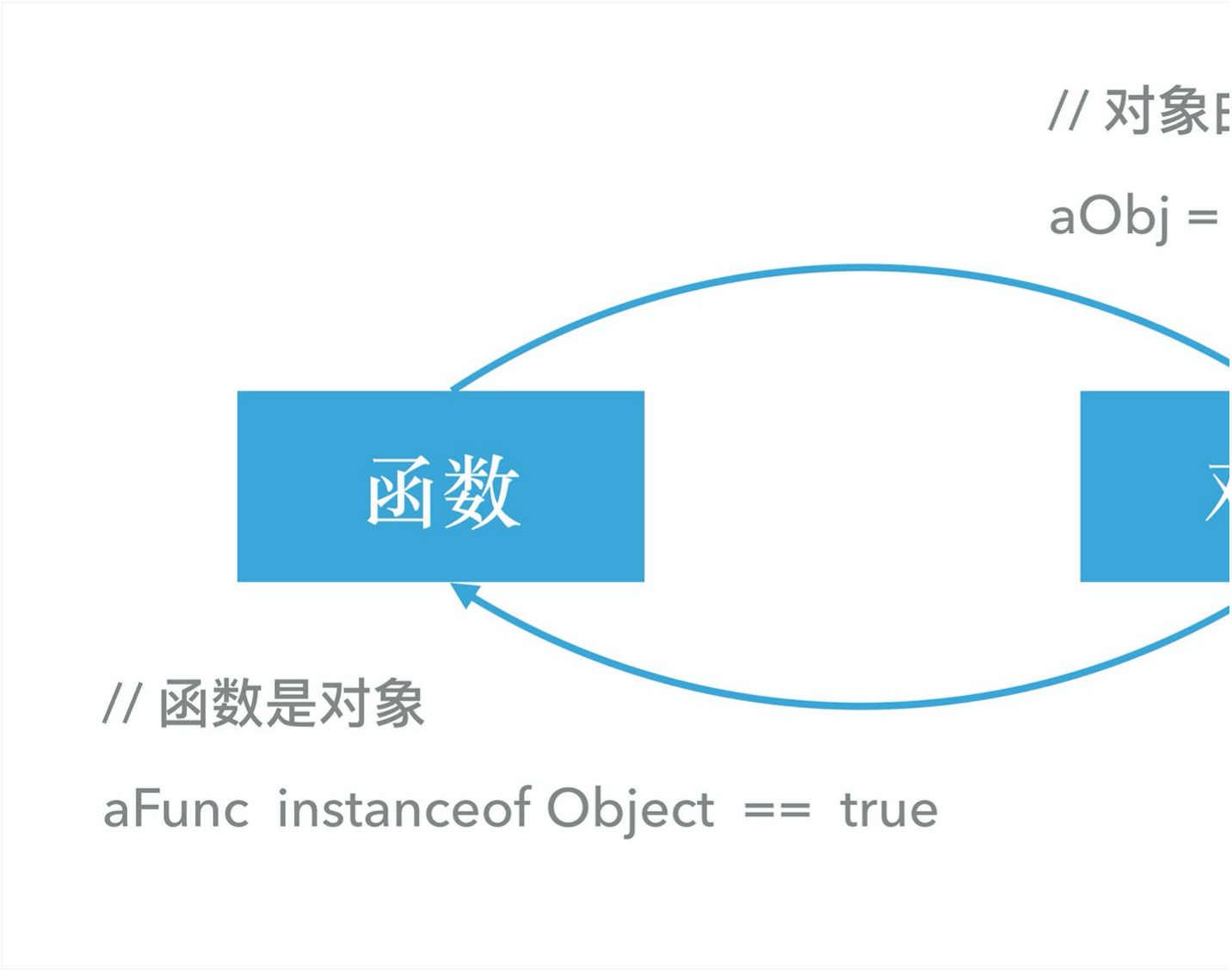
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中`Function()`是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向`Function()`，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

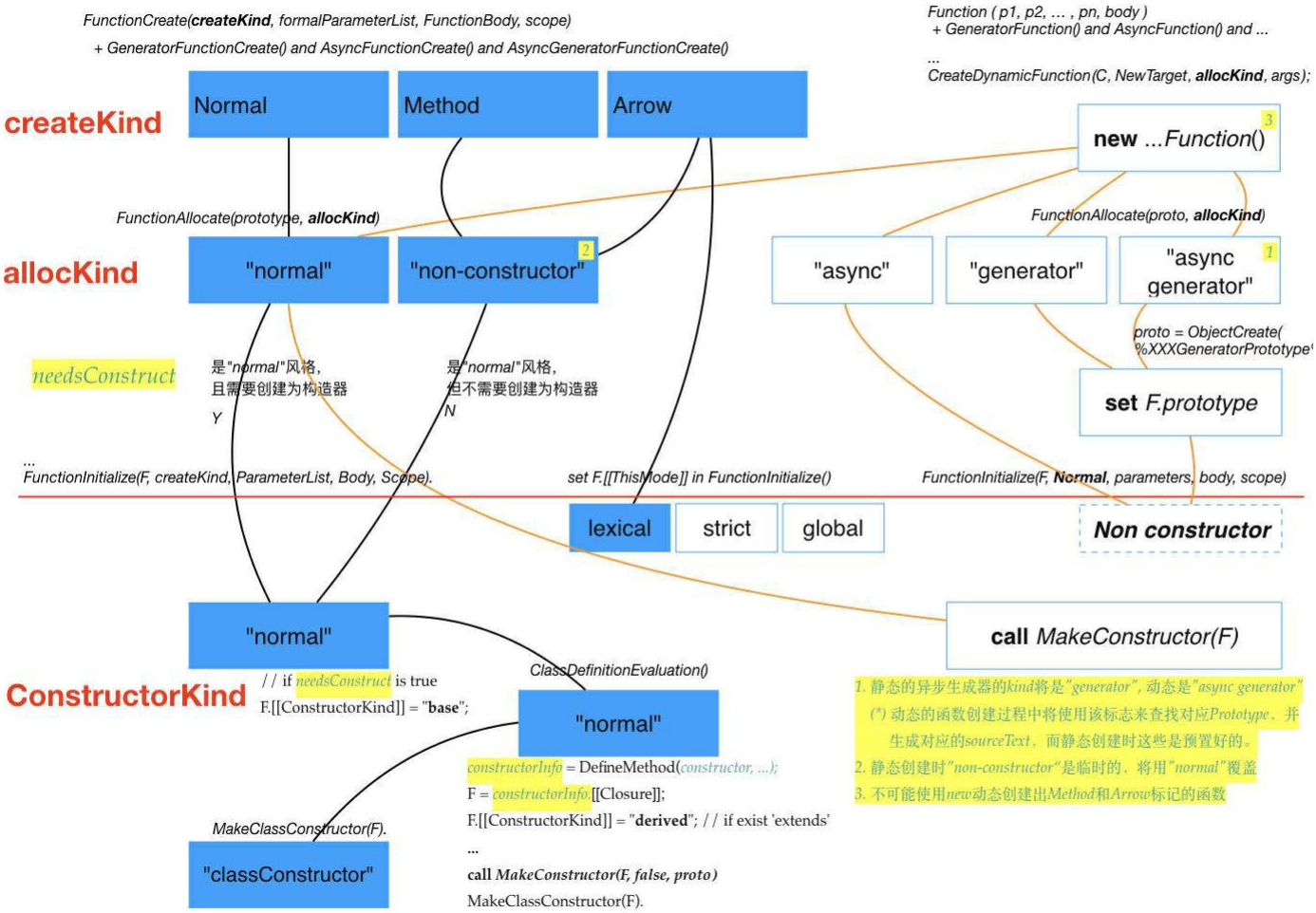
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货吐）。
by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

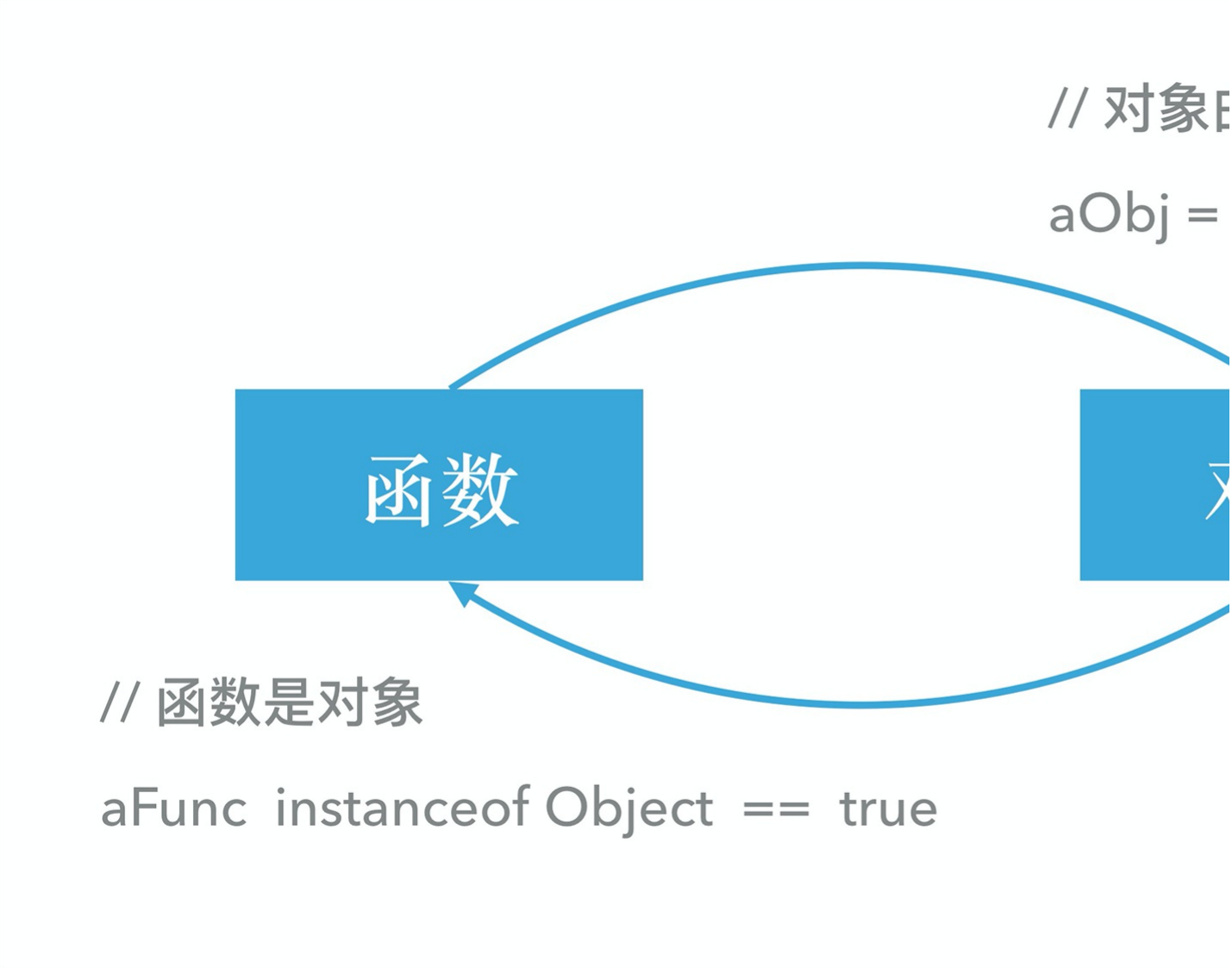
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



> NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

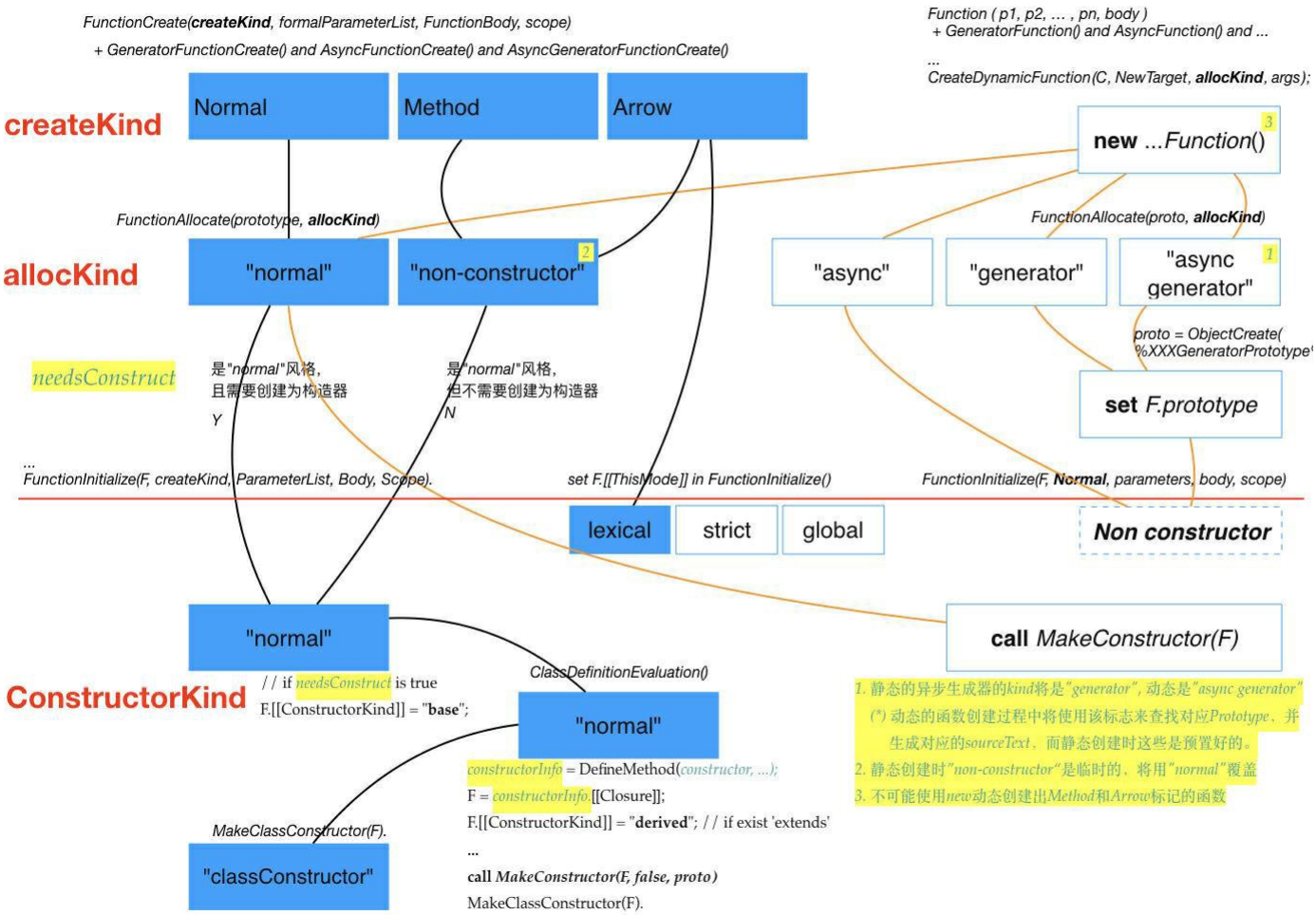
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及

内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

> NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

> NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

> NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货叱）。
> by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default ...）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用`eval()`，例如：

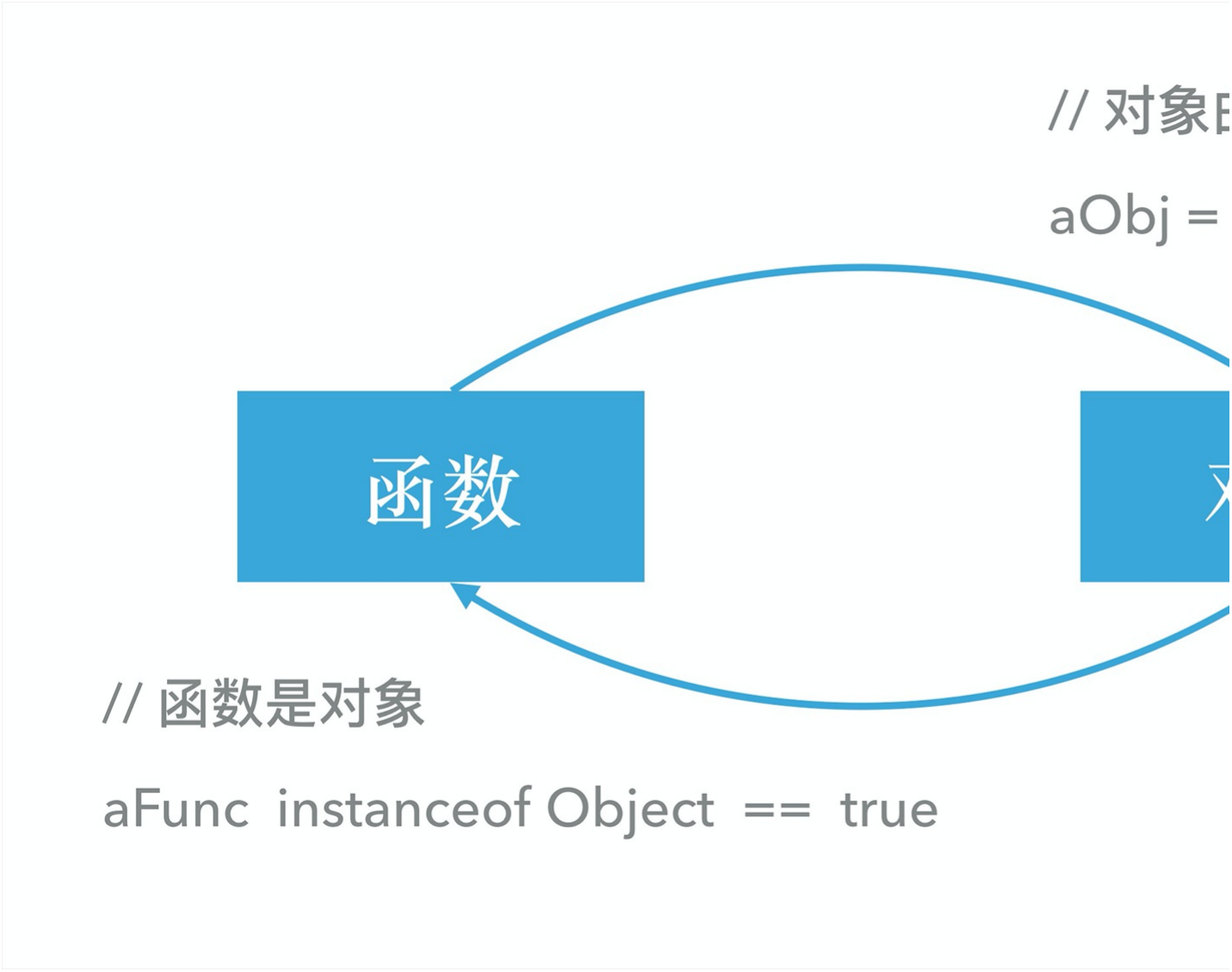
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中`Function()`是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向`Function()`，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

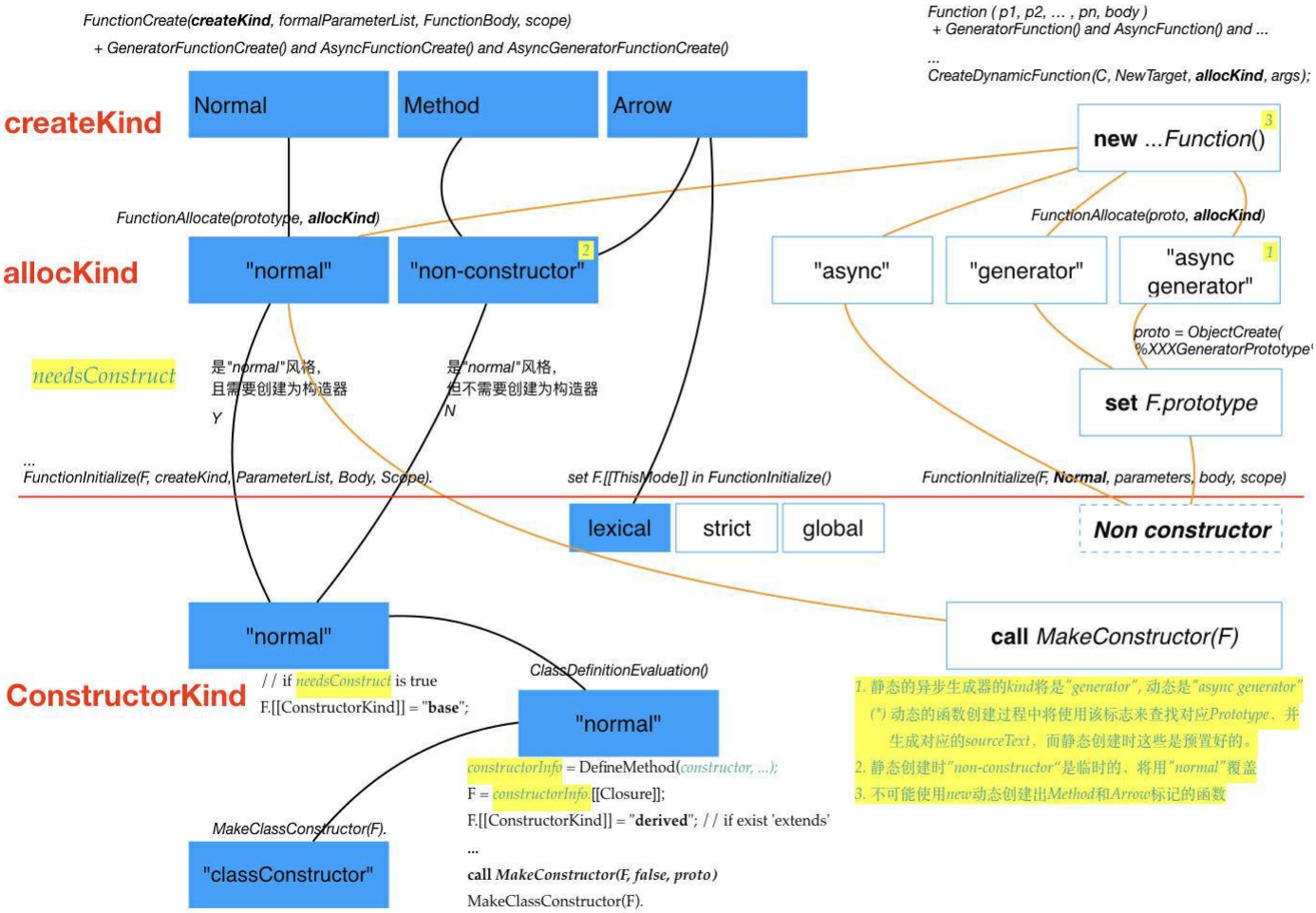
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为"allocKind"的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为"函数初始化（FunctionInitialize）"的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数"作为对象"的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！"好像"什么也没有发生？！事实上，在引擎层面，所谓的"动态函数创建"就是什么也没有发生，因为执行引擎并不理解"声明一个函数"与"动态创建一个函数"之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成"值类型"才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论"动态函数创建"内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种"没有差异"反而体现了"函数式语言"的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么"动态创建的函数"与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到"唯一一点不同"。而这，也将是我在"动态语言"这个系列中为你揭示的最后一个秘密。

唯一一点不同

在"函数初始化（FunctionInitialize）"这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的"域（即引擎的一个实例）"中取出全局环境，然后将它作为"父级的作用域（scope）"，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的"动态函数"的父级作用域将指向全局！

你绝不可能在"当前上下文（环境/作用域）"中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货吐）。
by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

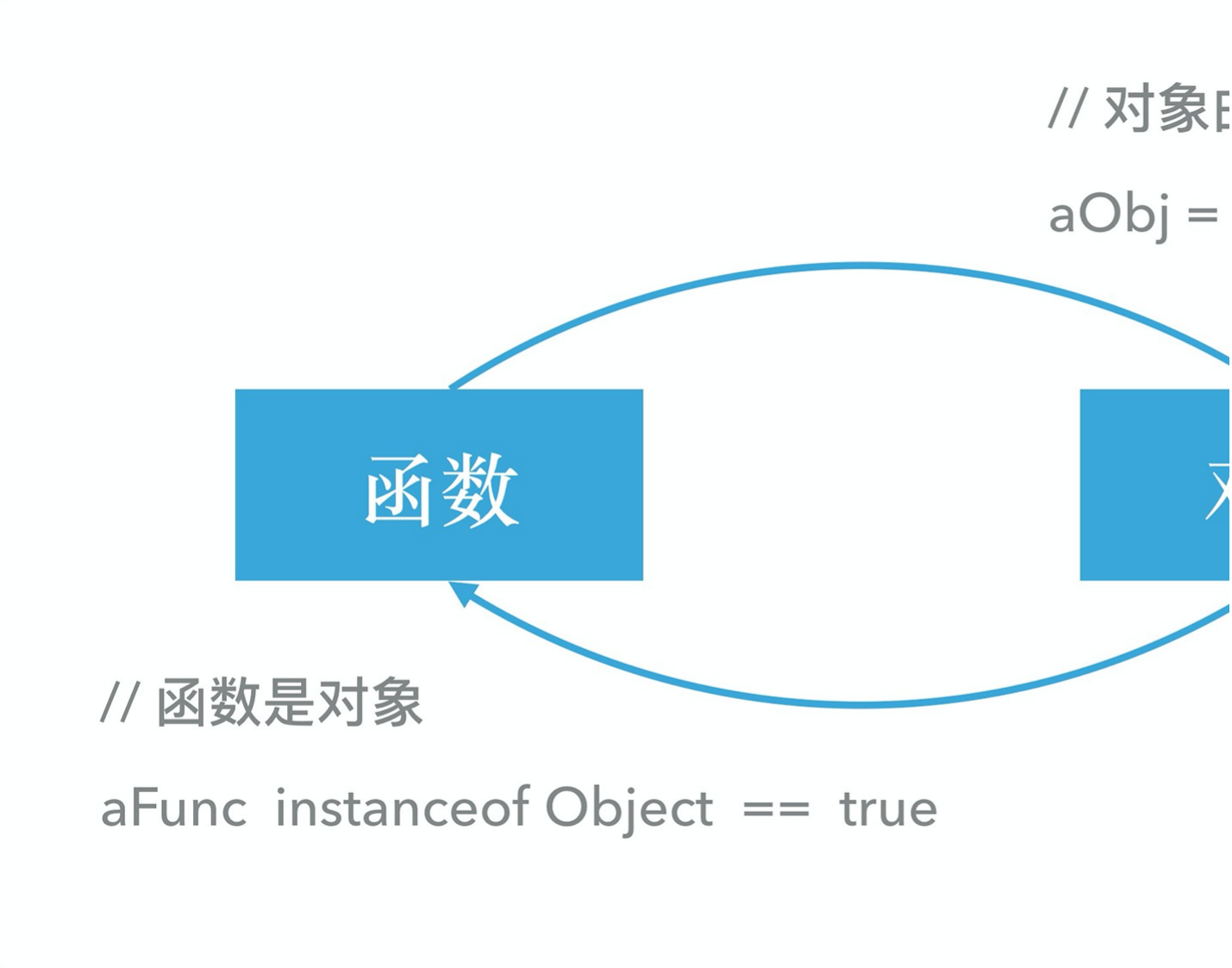
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



> NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

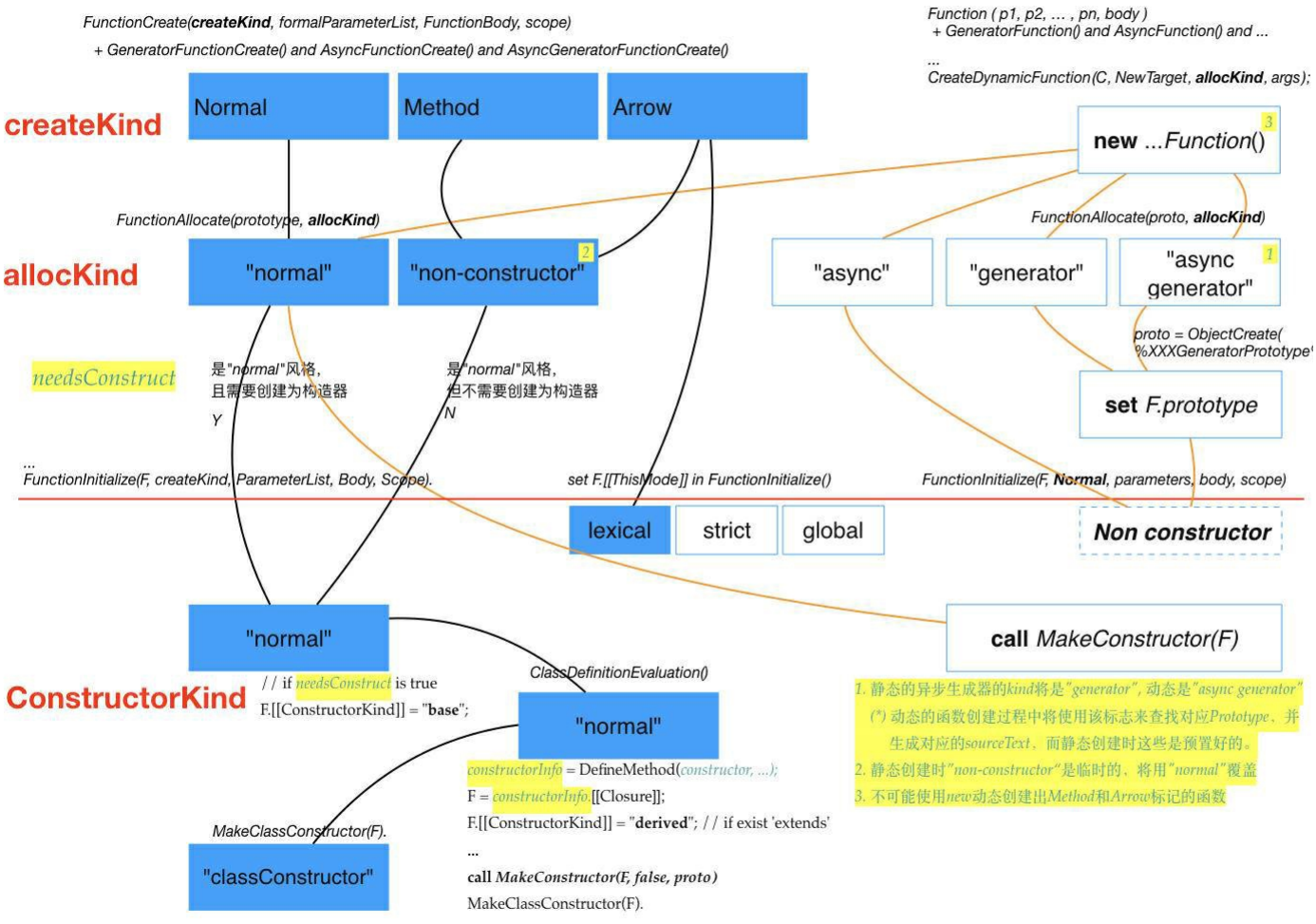
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及

内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

> NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

> NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

> NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货叱）。
> by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default ...）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用`eval()`，例如：

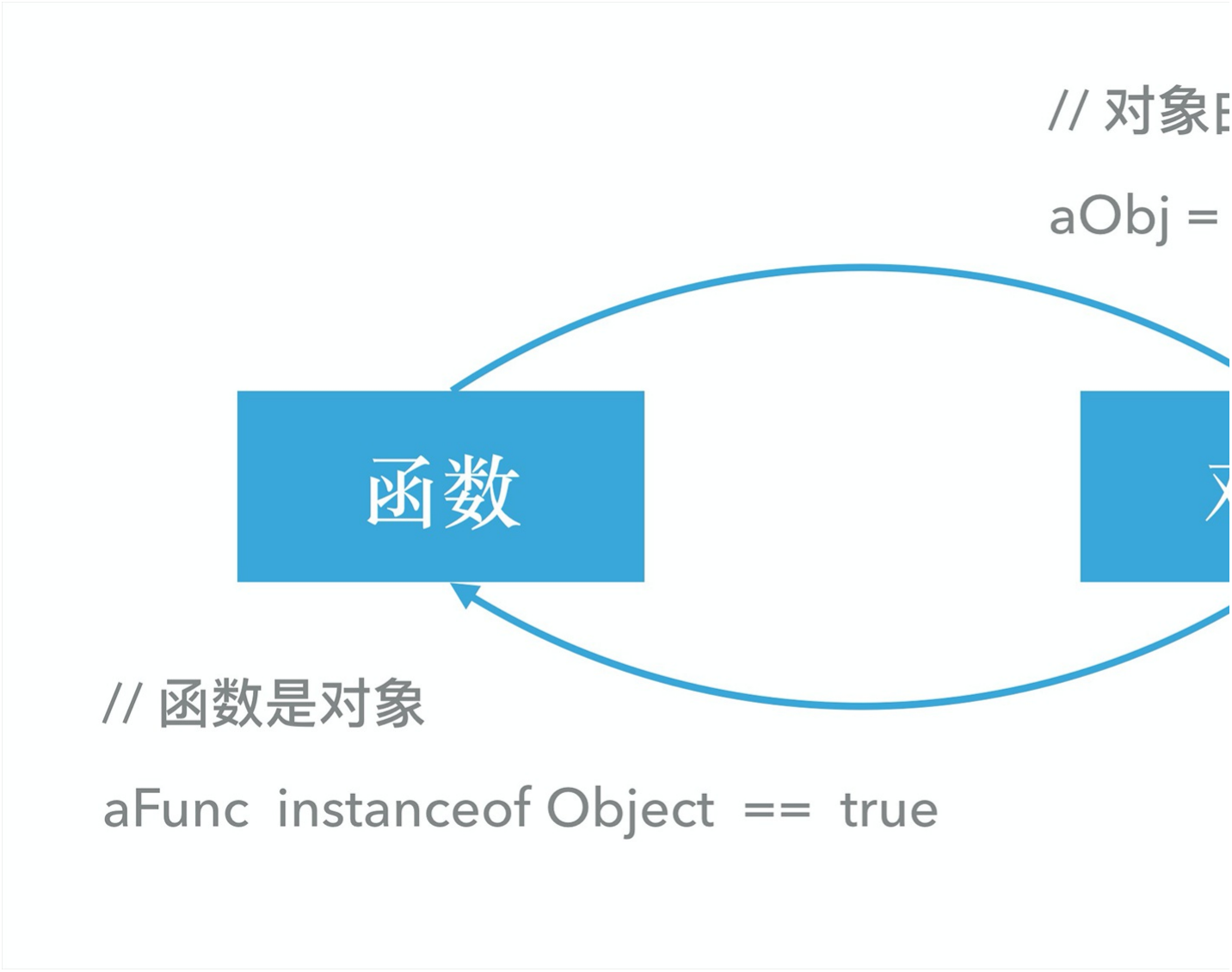
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中`Function()`是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向`Function()`，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

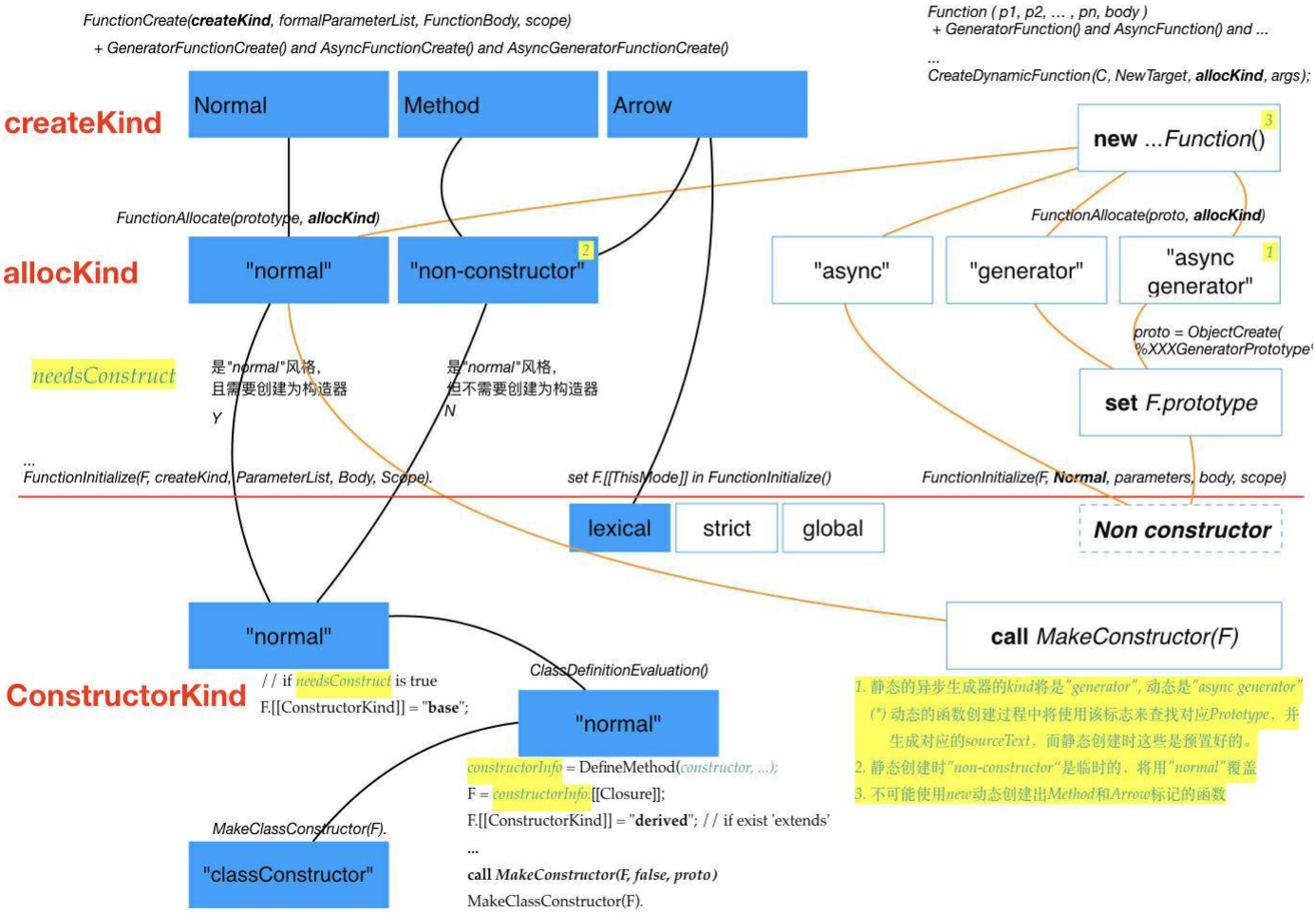
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货吐）。
by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

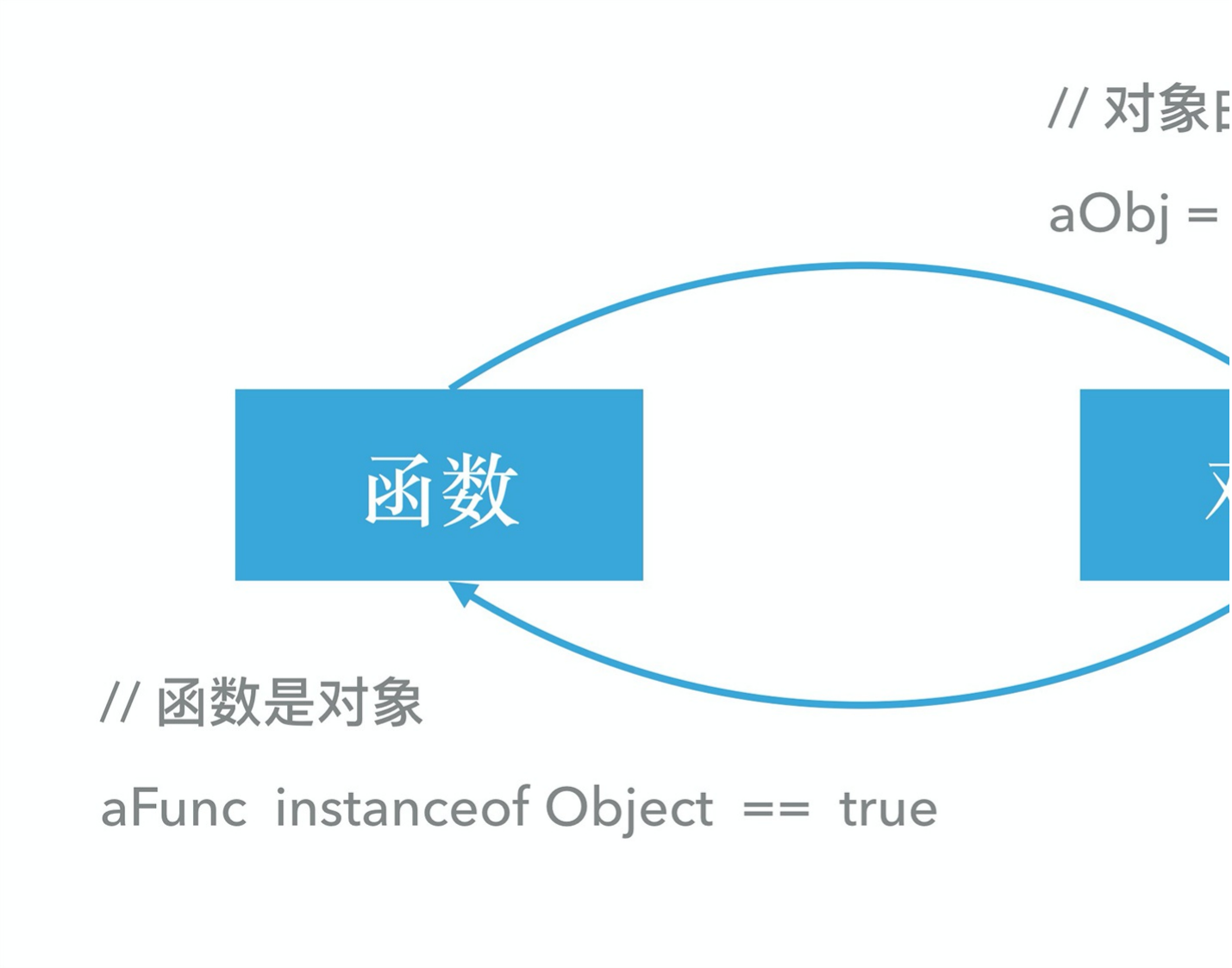
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



> NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

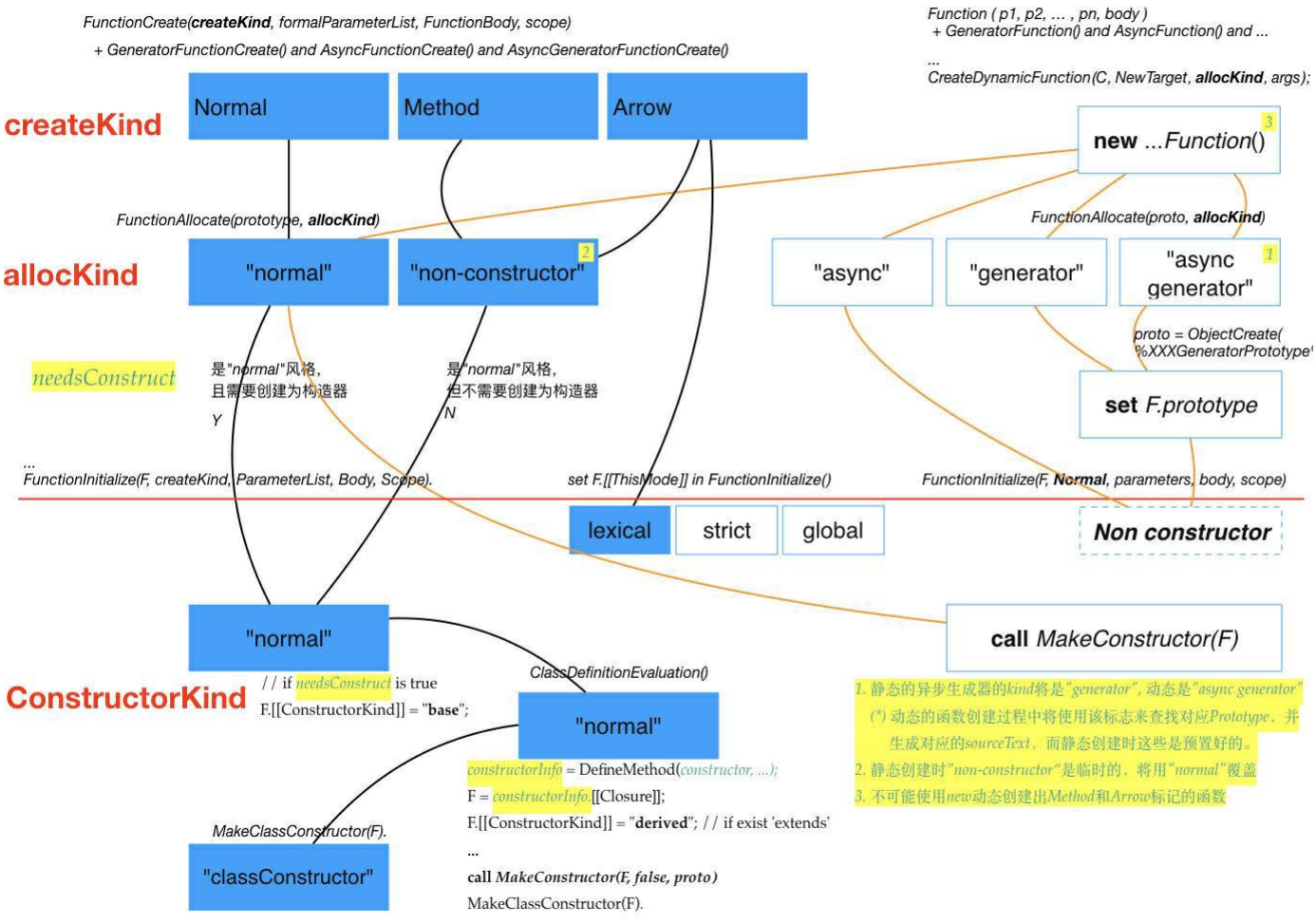
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及

内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

> NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

> NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

> NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货叱）。
> by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default ...）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用`eval()`，例如：

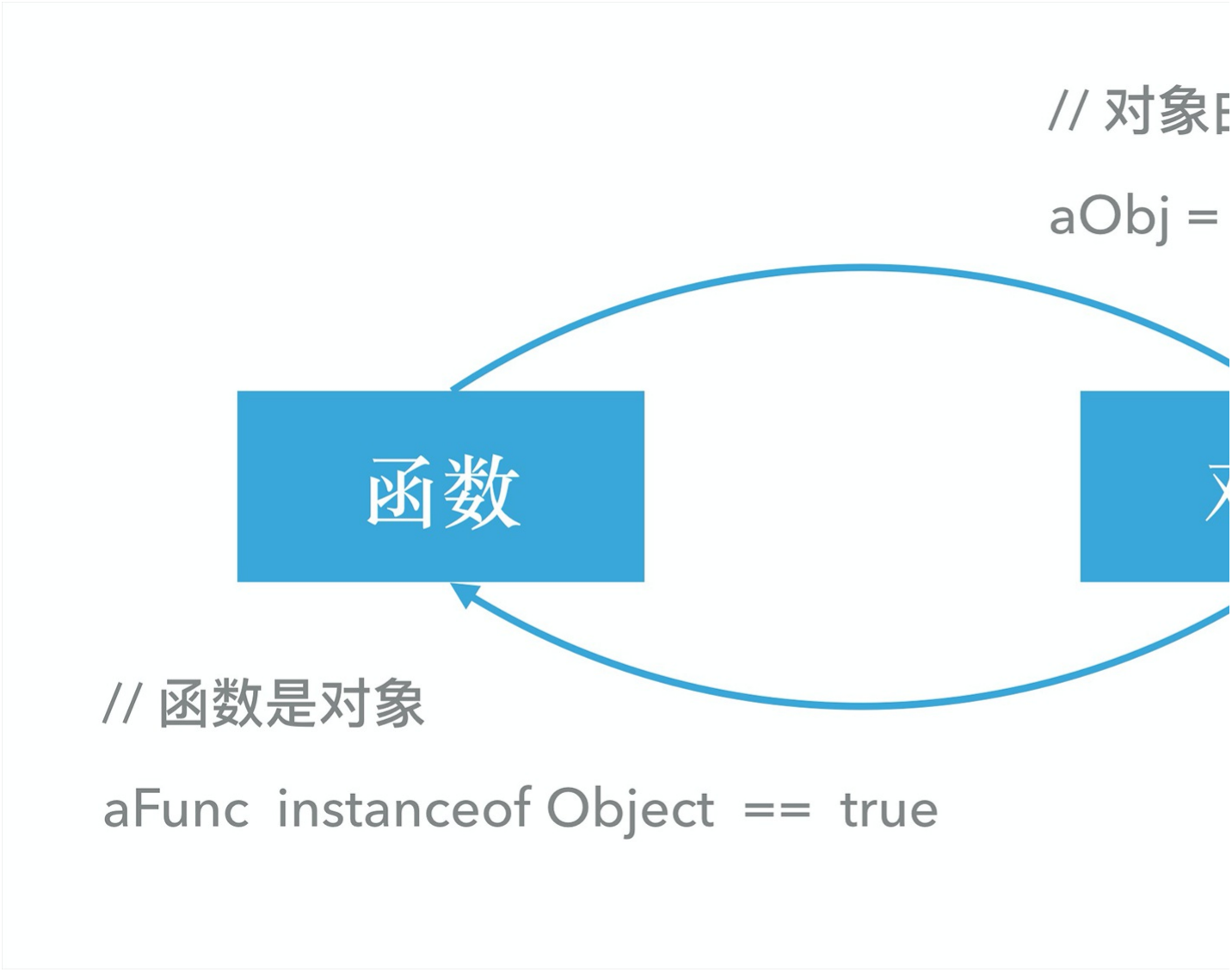
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中`Function()`是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向`Function()`，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

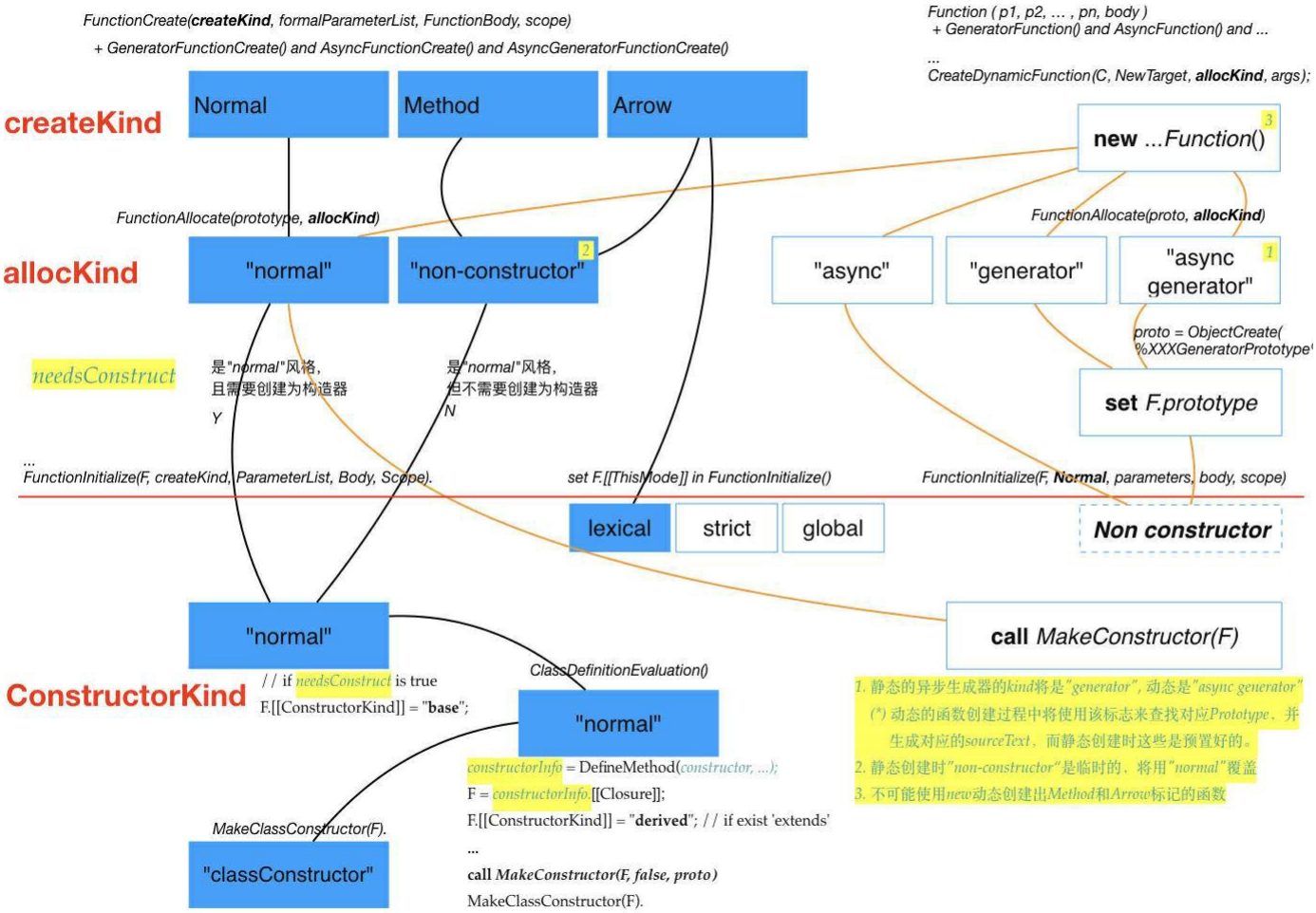
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货吐）。
by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

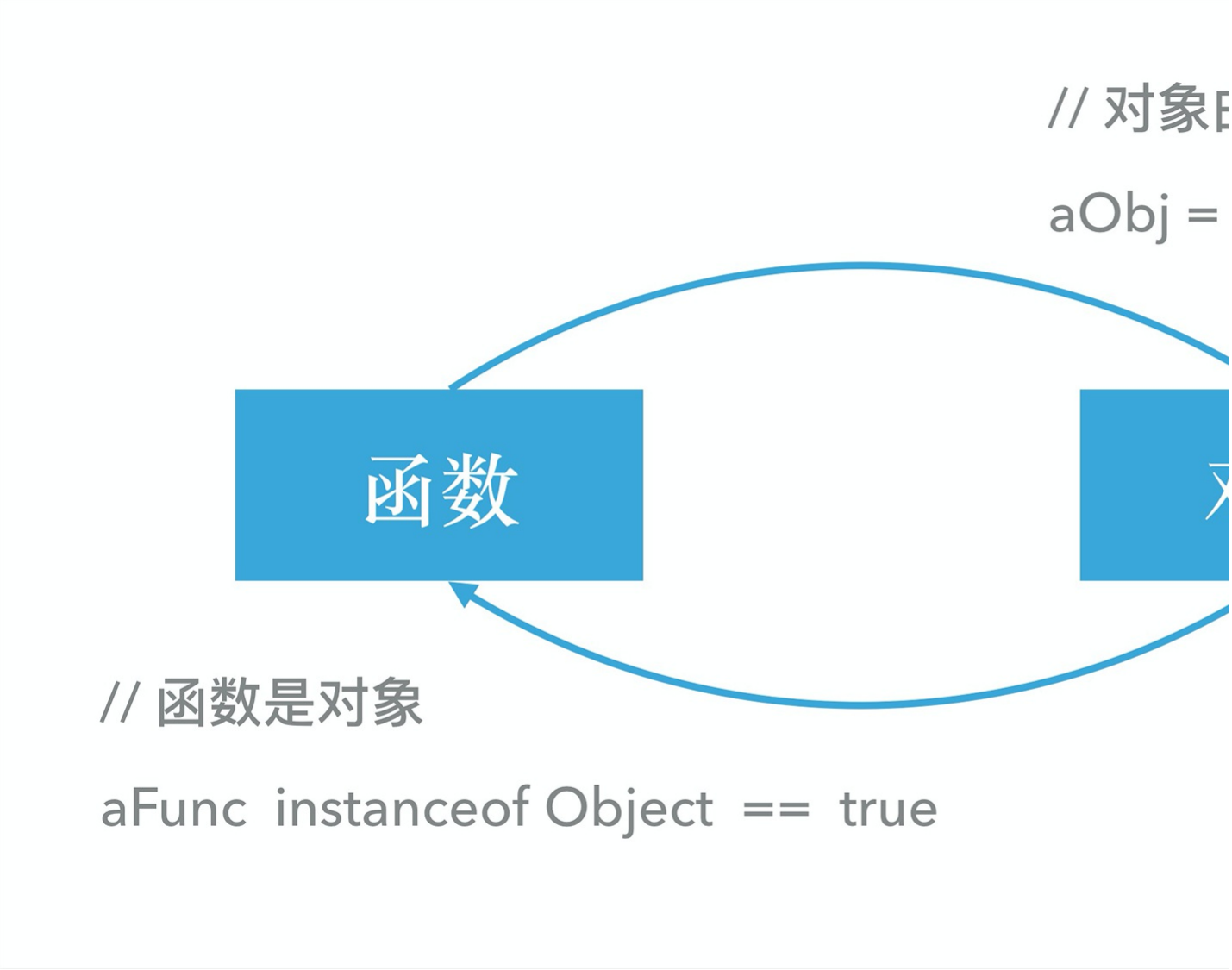
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



> NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

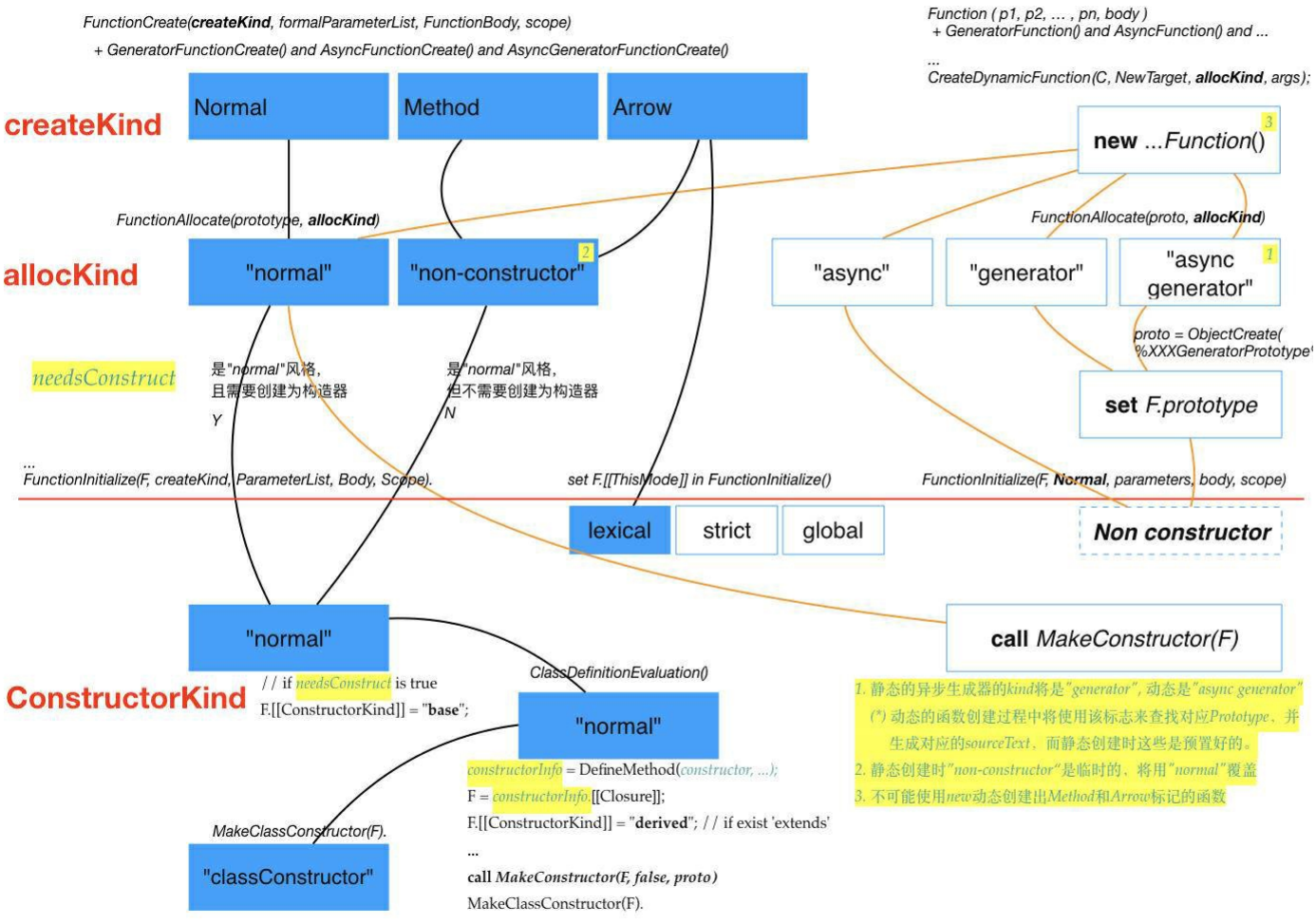
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及

内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

> NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

> NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

> NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货叱）。
> by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default ...）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用`eval()`，例如：

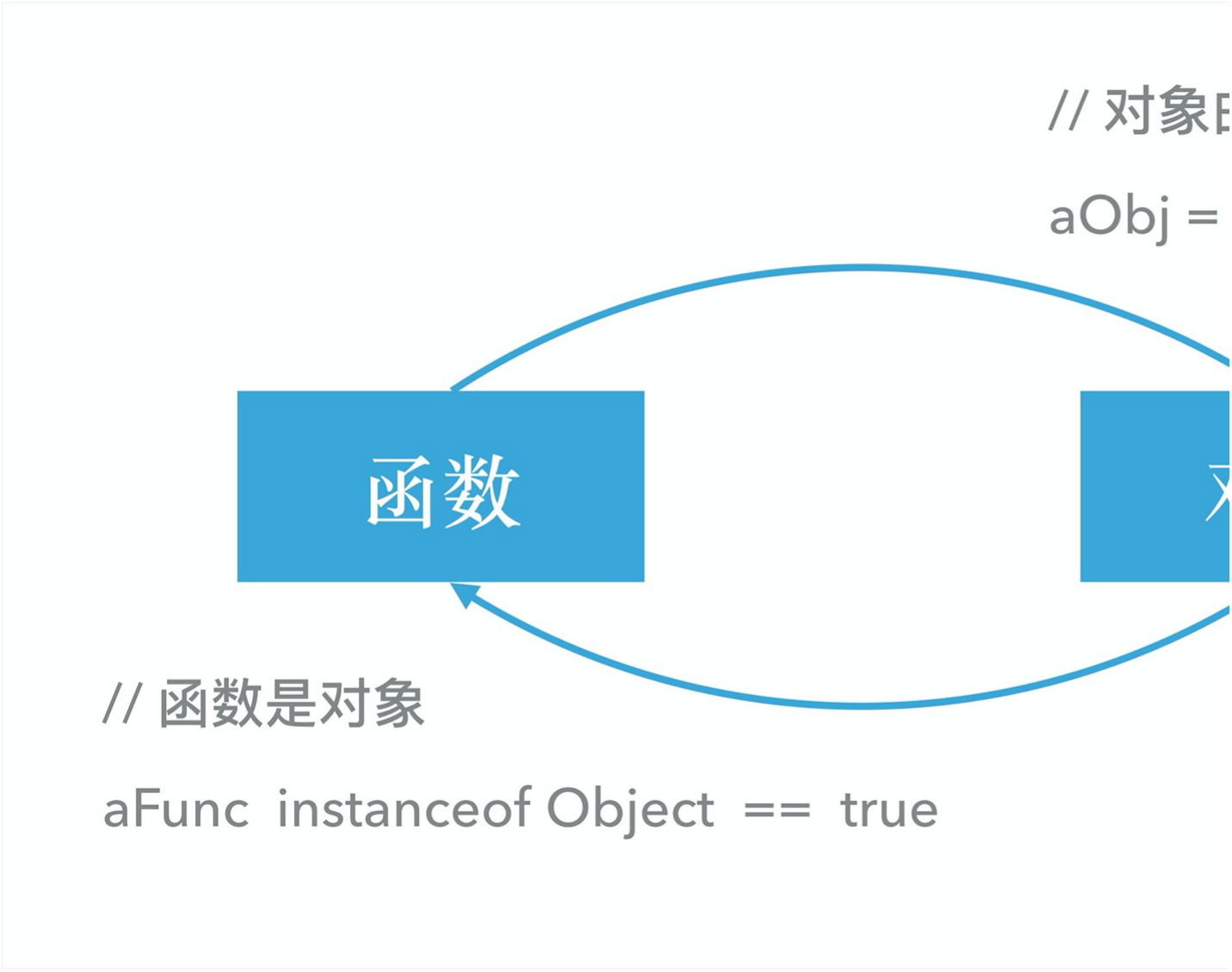
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中`Function()`是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向`Function()`，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

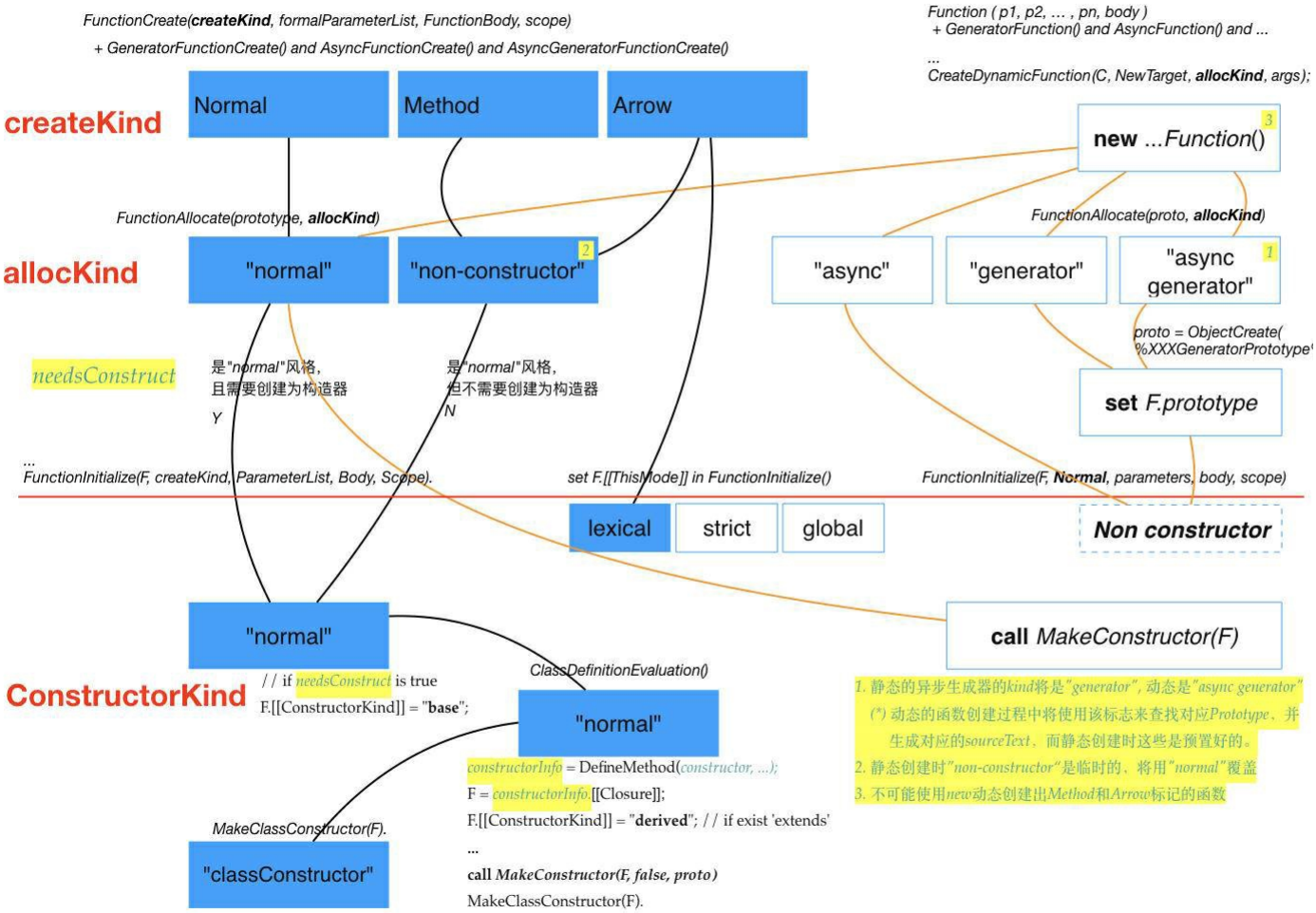
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货吐）。
by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

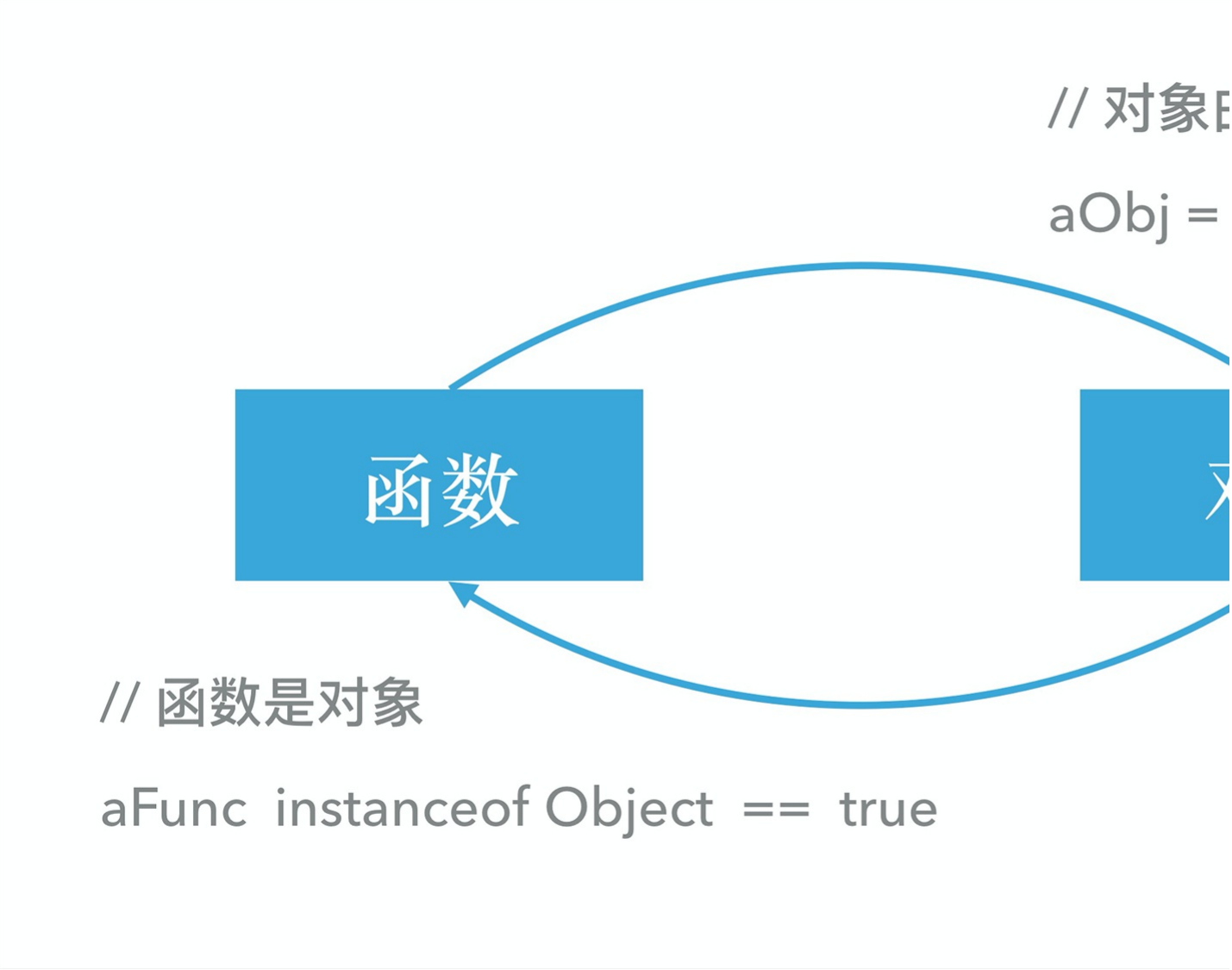
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



> NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

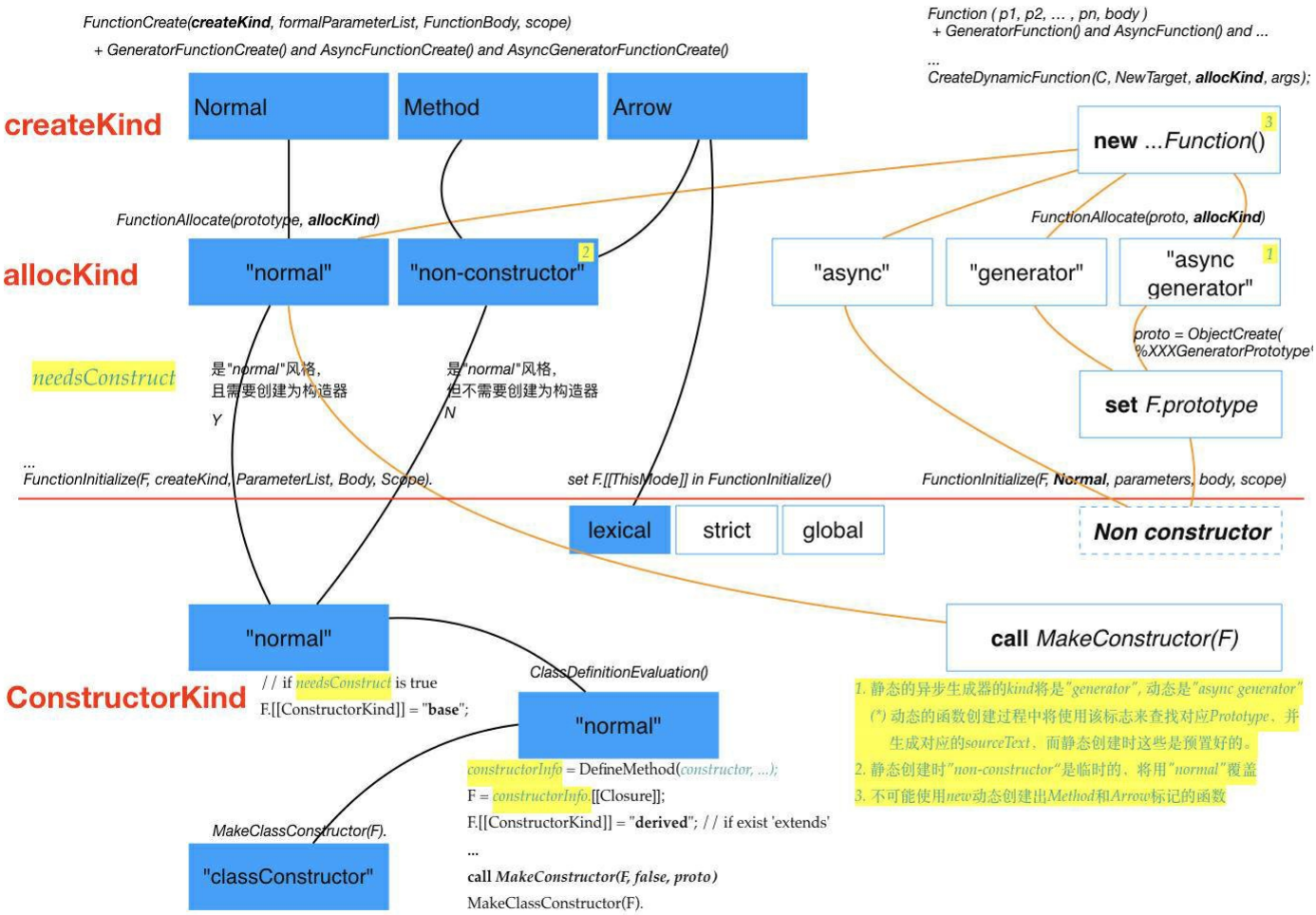
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及

内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

> NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

> NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

> NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货叱）。
> by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default ...）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用`eval()`，例如：

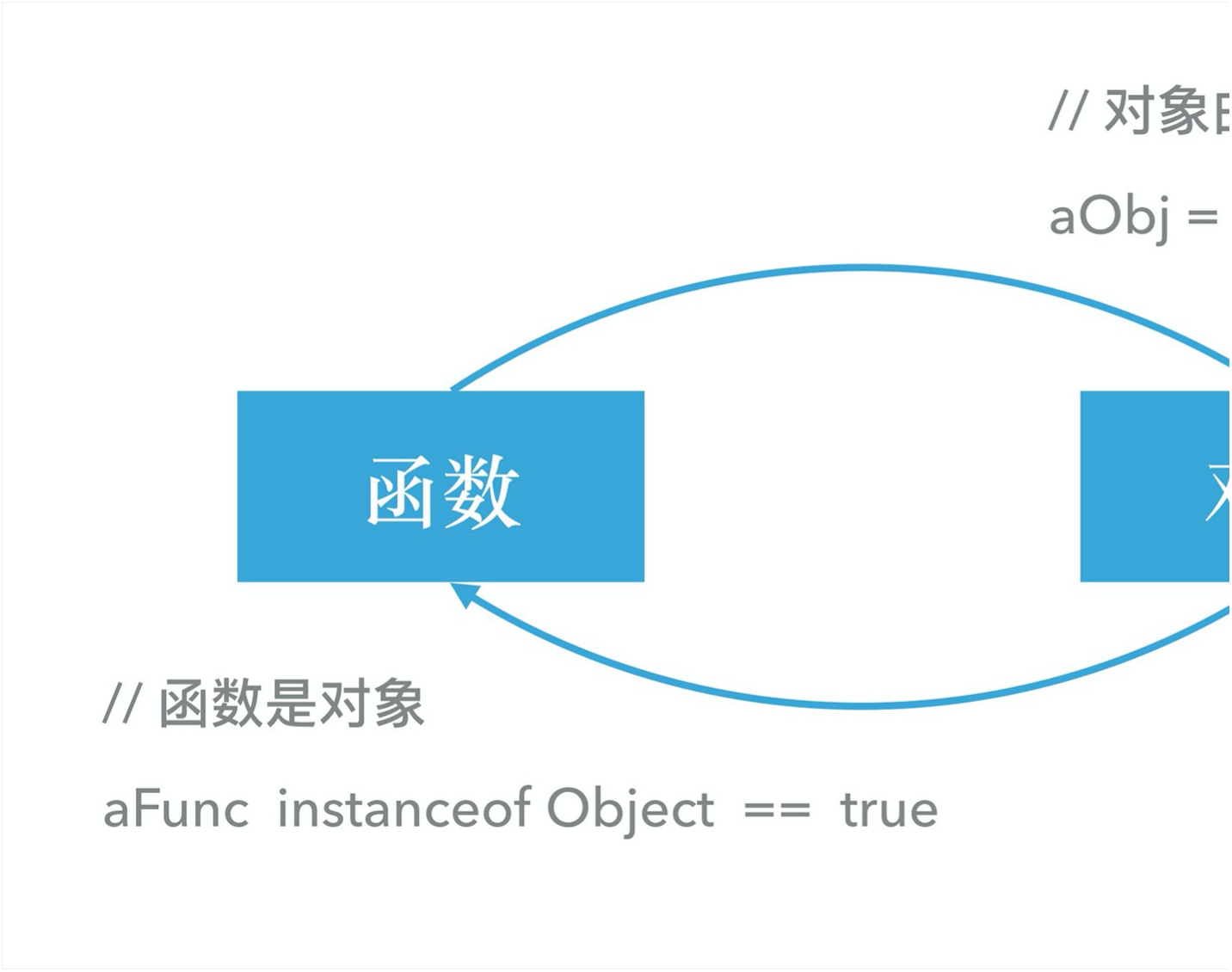
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中`Function()`是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向`Function()`，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

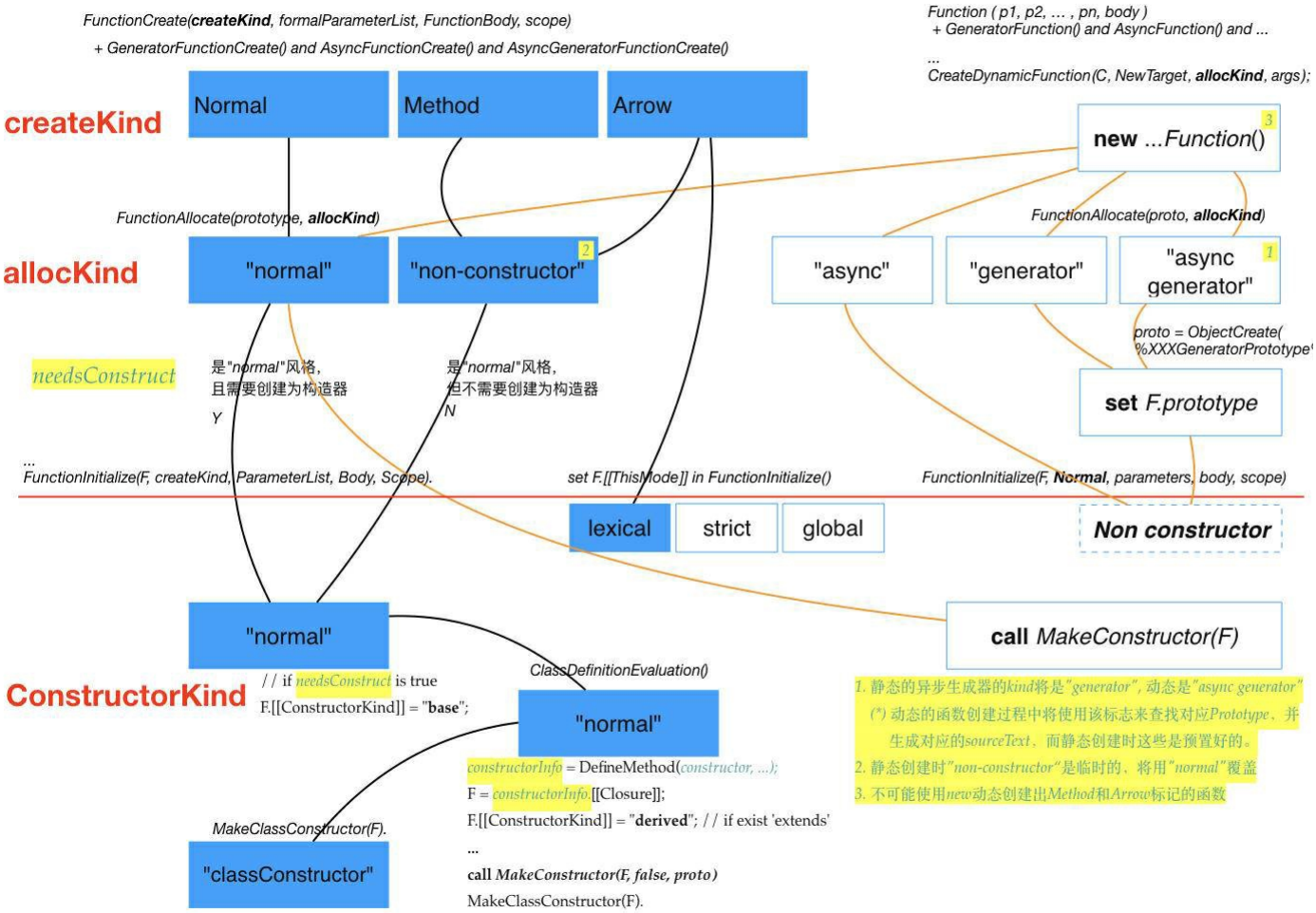
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货吐）。
by aimingoo.

你好，我是周爱民，欢迎回到我的专栏。

今天是专栏最后一讲，我接下来要跟你聊的，仍然是JavaScript的动态语言特性，主要是动态函数的实现原理。

标题中的代码比较简单，是常用、常见的。这里稍微需要强调一下的是“最后一对括号的使用”，由于运算符优先级的设计，它是在new运算之后才被调用的。也就是说，标题中的代码等义于：

```
// （等义于）
(new Function('x = 100'))()

// （或）
f = new Function('x = 100')
f()
```

此外，这里的new运算符也可以去掉。也就是说：

```
new Function(x)

// vs.
Function(x)
```

这两种写法没有区别，都是动态地创建一个函数。

函数的动态创建

如果在代码中声明一个函数，那么这个函数必然是具名的。具名的、静态的函数声明有两个特性：

1. 是它在所有代码运行之前被创建；
2. 它作为语句的执行结果将是“空（Empty）”。

这是早期JavaScript中的一个硬性的约定，但是到了ECMAScript 6开始支持模块的时候，这个设计就成了问题。因为模块是静态装配的，这意味着它导出的内容“应该是”一个声明的结果或者一个声明的名字，因为只有**声明**才是静态装配阶段的特性。但是，所有声明语句的完成结果都是Empty，是无效的，不能用于导出。

> NOTE：关于6种声明，请参见《第02讲》。

而声明的名字呢？不错，这对具名函数来说没问题。但是匿名函数呢？就成了问题了。

因此，在支持匿名函数的“缺省导出（export default …）”时，ECMAScript就引入了一个称为“函数定义（Function Definitions）”的概念。这种情况下，函数表达式是匿名的，但它的结果会绑定给一个名字，并且最终会导出那个名字。这样一来，函数表达式也就有了“类似声明的性质”，但它又不是静态声明（Declarations），所以概念上叫做定义（Definitions）。

> NOTE：关于匿名函数对缺省导出的影响，参见《第04讲》。

在静态声明的函数、类，以及这里说到的函数定义之外，用户代码还可以创建自己的函数。这同样有好几种方式，其中之一，是使用eval()，例如：

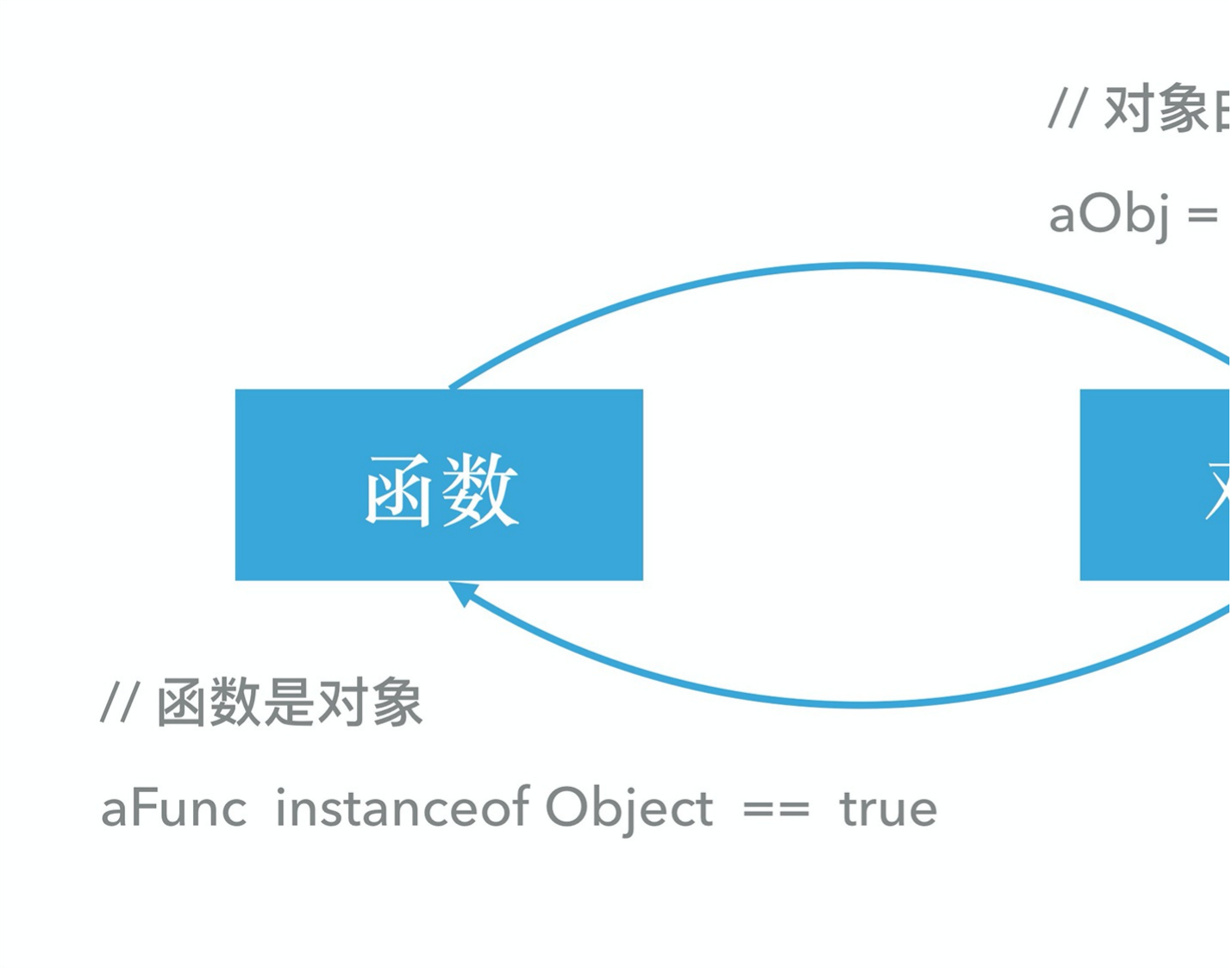
```
# 在非严格模式下，这将在当前上下文中“声明”一个名为foo的函数
> eval('function foo() {}')
```

还有一种常见的方式，就是使用动态创建。

几种动态函数的构造器

在JavaScript中，“动态创建”一个东西，意味着这个东西是一个对象，它创建自类/构造器。其中Function()是一切函数缺省的构造器（或类）。尽管内建函数并不创建自它，但所有的内建函数也通过简单的映射将它们的原型指向Function。除非经过特殊的处理，所有JavaScript中的函数原型均最终指向Function()，它是所有函数的祖先类。

这种处理/设计使得JavaScript中的函数有了“完整的”面向对象特性，函数的“类化”实现了JavaScript在函数式语言和面向对象语言在概念上的大一统。于是，一个内核级别的概念完整性出现了，也就是所谓：对象创建自函数；函数是对象。如下图所示：



> NOTE：关于概念完整性以及它在“体系性”中的价值，参见《加餐3：让JavaScript运行起来》。

在ECMAScript 6之后，有赖于类继承体系的提出，JavaScript中的函数也获得了“子类化”的能力，于是用户代码也可以派生函数的子类了。例如：

```
class MyFunction extends Function {
  // ...
}
```

但是用户代码无法重载“函数的执行”能力。很明显，这是执行引擎自身的能力，除非你可以重写引擎，否则重载执行能力也就无从谈起。

NOTE：关于类、派生，以及它们在对原生构造器进行派生时的贡献，请参见《第15讲》。

除了这种用户自定义的子类化的函数之外，JavaScript中一共只有四种可以动态创建的函数，包括：**一般函数**（Function）、**生成器函数**（GeneratorFunction）、**异步生成器函数**（AsyncGeneratorFunction）和**异步函数**（AsyncFunction）。又或者说，用户代码可以从这四种函数之任一开始来派生它们的子类，在保留它们的执行能力的同时，扩展接口或功能。

但是，这四种函数在JavaScript中有且只有`Function()`是显式声明的，其他三种都没有直接声明它们的构造器，这需要你用如下代码来得到：

```
const GeneratorFunction = (function* (){}).constructor;
const AsyncGeneratorFunction = (async function* (){}).constructor
const AsyncFunction = (async x=>x).constructor;

// 示例
(new AsyncFunction)().then(console.log); // promise print 'undefined'
```

函数的三个组件

我们提及过函数的三个组件，包括：**参数**、**执行体**和**结果**。其中“结果（Result）”是由代码中的return子句负责的，而其他两个组件，则是“动态创建一个函数”所必须的。这也是上述四个函数（以及它们的子类）拥有如下相同界面的原因：

Function (p1, p2, … , pn, body)

NOTE：关于函数的三个组件，以及基于它们的变化，请参见《第8讲、第9讲、第10讲，它们分别讨论“三个组件”、改造“执行体”，以及改造“参数和结果”》。

其中，用户代码可以使用字符串来指定p1…pn的形式参数（Formals），并且使用字符串来指定函数的执行体（Body）。类似如下：

```
f = new Function('x', 'y', 'z', 'console.log(x, y, z)');

// 测试
f(1,2,3); // 1 2 3
```

JavaScript也允许用户代码将多个参数合写为一个，也就是变成类似如下形式：

```
f = new Function('x, y, z', ...);
```

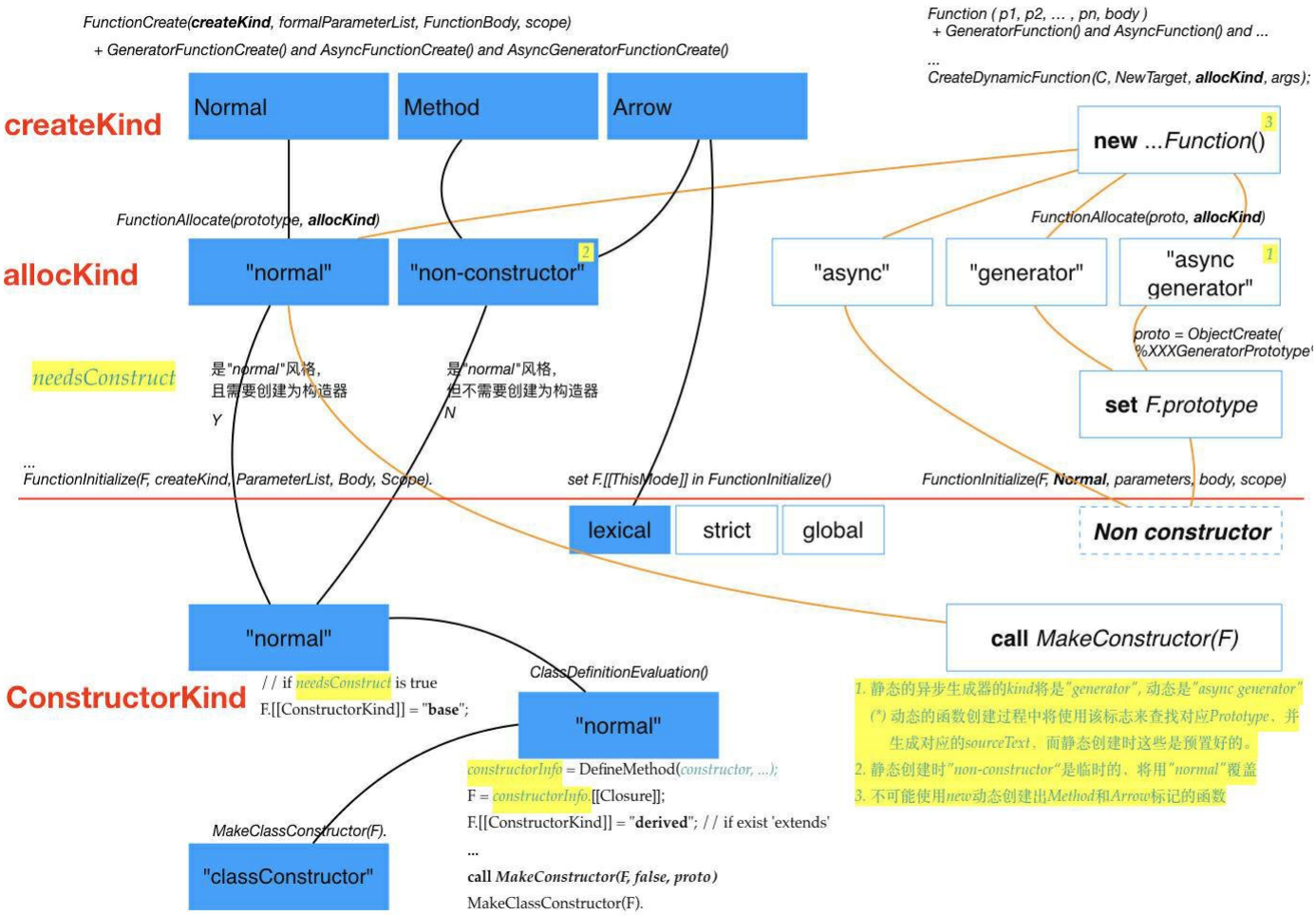
或者在字符串声明中使用缺省参数等扩展风格，例如：

```
f = new Function('x = 0, ...args', 'console.log(x, ...args)');
f(undefined, 200, 300, 400); // 0 200 300 400
```

动态函数的创建过程

所有的四种动态函数的创建过程都是一致的，它们都将调用内部过程CreateDynamicFunction()来创建函数对象。但相对于静态声明的函数，动态创建（CreateDynamicFunction）却有自己不同的特点与实现过程。

NOTE：关于对象的构造过程，请参见《第13讲（13 | new X）》。



JavaScript在创建函数对象时，会为它分配一个称为“allocKind”的标识。相对于静态创建，这个标识在动态创建过程中反而更加简单，正好与上述四种构造器一一对应，也就不再需要进行语法级别的分析与识别。其中除了`normal`类型（它所对应的构造器是`Function()`）之外，其他的三种都不能作为构造器来创建和初始化。所以，只需要简单地填写它们的内部槽，并置相应的原型（原型属性F.prototype以及

内部槽F.[[Prototype]]）就可以了。

最后，当函数作为对象实例完成创建之后，引擎会调用一个称为“函数初始化（FunctionInitialize）”的内置过程，来初始那些与具体实例相关的内部槽和外部属性。

> NOTE：在ECMAScript 6中，动态函数的创建过程主要由FunctionAllocate和FunctionInitialize两个阶段完成。而到了ECMAScript 9中，ECMAScript规范将FunctionAllocate的主要功能归入到OrdinaryFunctionCreate中（并由此规范了函数“作为对象”的创建过程），而原本由FunctionInitialize负责的初始化，则直接在动态创建过程中处理了。

然后呢？然后，函数就创建完了。

是的！“好像”什么也没有发生？！事实上，在引擎层面，所谓的“动态函数创建”就是什么也没有发生，因为执行引擎并不理解“声明一个函数”与“动态创建一个函数”之间的差异。

我们试想一下，如果一个执行引擎要分别理解这两种函数并尝试不同的执行模式或逻辑，那么这个引擎的效率得有多差。

作为一个函数

通常情况下，接下来还需要一个变量来引用这个函数对象，或者将它作为表达式操作数，它才会有意义。如果它作为引用，那么它跟普通变量或其他类型的数据类似；如果它作为一般操作数，那么它应该按照上一讲所说的规则，转换成“值类型”才能进行运算。

> NOTE：关于引用、操作数，以及值类型等等，请参见《第01讲（01 | delete 0）》。

所以，如果不讨论“动态函数创建”内在的特殊性，那么它的创建与其他数据并没有本质的不同：创建结果一样，对执行引擎或运行环境的影响也一样。而这种“没有差异”反而体现了“函数式语言”的一项基本特性：函数是数据。也就是说，函数可以作为一般数据来处理，例如对象，又例如值。

函数与其他数据不同之处，仅在于它是可以调用的。那么“动态创建的函数”与一般函数相比较，在调用/执行方面有什么特殊性吗？

答案是，仍然没有！在ECMAScript的内部方法`Call()`或者函数对象的内部槽`[[Call]]` `[[Construct]]`中，根本没有任何代码来区别这两种方式创建出来的函数。它们之间毫无差异。

> NOTE：事实上，不惟如此，我尝试过很多的方式来识别不同类型的函数（例如构造器、类、方法等）。除了极少的特例之外，在用户代码层面是没有办法识别函数的类型的。就现在的进展而言，`isBindable()`、`isCallable()`、`isConstructor()`和`isProxy()`这四个函数是可以实现的，其他的类似`isClassConstructor()`、`isMethod()`和`isArrowFunction()`都没有有效的识别方式。

> NOTE：如上的这些识别函数，需要在不利用toString()方法，以及不调用函数的情况下来完成。因为执行函数会带来未知的结果，而toString方法的实现在许多引擎中并不标准，不可依赖。

不过，如果我们将时钟往回拨一点，考察一下这个函数被创建出来之前所发生的事情，那么，我们还是能找到“唯一一点不同”。而这，也将是我在“动态语言”这个系列中为你揭示的最后一个秘密。

唯一一点不同

在“函数初始化（FunctionInitialize）”这个阶段中，ECMAScript破天荒地约定了几行代码，这段规范文字如下：

> Let realmF be the value of F's [[Realm]] internal slot.
> Let **scope** be realmF.[[GlobalEnv]].
> Perform *FunctionInitialize*(F, Normal, parameters, body, **scope**).

它们是什么意思呢？

规范约定需要从函数对象所在的“域（即引擎的一个实例）”中取出全局环境，然后将它作为“父级的作用域（scope）”，传入`FunctionInitialize()`来初始化函数`F`。也就是说，所有的“动态函数”的父级作用域将指向全局！

你绝不可能在“当前上下文（环境/作用域）”中动态创建动态函数。和间接调用模式下的`eval()`一样，所有动态函数都将创建在全局！

一说到跟“间接调用eval()”存有的相似之处，可能你立即会反应过来：这种情况下，`eval()`不仅仅是在全局执行，而且将突破“全局的严格模式”，代码将执行在非严格模式中！那么，是不是说，“动态函数”既然与它有相似之处，是不是也有类似性质呢？

> NOTE：关于间接调用`eval()`，请参见《第21讲》。

答案是：的确！

出于与“间接调用eval()”相同的原因——即，在动态执行过程中无法有效地（通过上下文和对应的环境）检测全局的严格模式状态，所以动态函数在创建时只检测代码文本中的第一行代码是否为`use strict`指示字，而忽略它“外部scope”是否处于严格模式中。

因此，即使你在严格模式的全局环境中创建动态函数，它也是执行在非严格模式中的。它与“间接调用eval()”的唯一差异，仅在于“多封装了一层函数”。

例如：

```
# 让NodeJS在启动严格模式的全局
> node --use-strict

# （在上例启动的NodeJS环境中测试）
> x = "Hi"
ReferenceError: x is not defined

# 执行在全局，没有异常
> new Function('x = "Hi"')()
undefined

# `x`被创建
> x
'Hi'

# 使用间接调用的`eval`来创建`y`
> (0, eval)('y = "Hello"')
> y
'Hello'
```

结尾

所以，回到今天这一讲的标题上来。标题中的代码，事实与上一讲中提到的“间接调用eval()”的效果一致，同样也会因为在全局中“向未声明变量赋值”而导致创建一个新的变量名`x`。并且，这一效果同样不受所谓的“严格模式”的影响。

在JavaScript的执行系统中出现这两个语法效果的根本原因，在于执行系统试图从语法环境中独立出来。如果考虑具体环境的差异性，那么执行引擎的性能将会较差，且不易优化；如果不考虑这种差异性，那么“严格模式”这样的性质就不能作为（执行引擎理解的）环境属性。

在这个两难中，ECMAScript帮助我们做出了选择：牺牲一致性，换取性能。

> NOTE：关于间接调用`eval()`对环境的使用，以及环境相关的执行引擎组件的设计与限制，请参见《第20讲》。

当然这也带来了另外一些好处。例如终于有了`window.execScript()`的替代实现，以及通过`new Function`这样来得到的、动态创建的函数，就可以“安全地”应用于并发环境。

至于现在，《JavaScript核心原理解析》一共22讲内容就全部结束了。

在这个专栏中，我为你讲述了JavaScript的静态语言设计、面向对象语言的基本特性，以及动态语言中的类型与执行系统。这看起来是一些零碎的、基本的，以及应用性不强的JavaScript特性，但是事实上，它们是你理解“更加深入的核心原理”的基础。

如果不先掌握这些内容，那么更深入的，例如多线程、并行语言特性等等都是空中楼阁，就算你勉强学来，也不过是花架子，是理解不到真正的“核心”的。

而这也是我像现在这样设计《JavaScript核心原理解析》22讲框架的原因。我希望你能在这些方面打个基础，先理解一下ECMAScript作为“语言设计者”这个角色的职责和关注点，先尝试一下深入探索JavaScript核心原理的乐趣（与艰难）。然后，希望我们还有机会在新的课程中再见！

多谢你的收听，最后邀请你填写这个专栏的调查问卷，我也想听听你的意见和建议，我将继续答疑解惑、查漏补缺，与你回顾这一路行来的苦乐。

再见。

> NOTE：编辑同学说还有一个“结束语”，我真不知道怎么写。不过，如果你觉得意犹未尽的话，到时候请打开听听吧（或许还有好货叱）。
> by aimingoo.