

13 | new X：从构造器到类，为你揭密对象构造的全程

2019-12-13 周爱民

JavaScript核心原理解析

[进入课程 >](#)



讲述：周爱民

时长 19:18 大小 17.69M



你好，我是周爱民。

今天我只跟你聊一件事，就是 JavaScript 构造器。标题中的这行代码中规中矩，是我这个专栏题目列表中难得的正经代码。

NOTE：需要稍加说明的是：这行代码在 JavaScript 1.x 的某些版本或具体实现中是不能使用的。即使 ECMAScript ed1 开始就将它作为标准语法之一，当时也还有许多语言并不支持它。

构造器这个东西，是 JavaScript 中面向对象系统的核心概念之一。跟“属性”相比，如果属性是静态的结构，那么“构造器”就是动态的逻辑。

没有构造器的 JavaScript，就是一个充填了无数数据的、静态的对象空间。这些对象之间既没有关联，也不能衍生，更不可能发生交互。然而，这却真的就是 JavaScript 1.0 那个时代的所谓“面向对象系统”的基本面貌。

基于对象的 JavaScript

为什么呢？因为 JavaScript 1.0 的时代，也就是最早最早的 JavaScript 其实是没有继承的。

你可能会说，既然是没有继承的，那么 JavaScript 为什么一开始就能声称自己是“面向对象”的、“类似 Java”的一门语言呢？其实这个讲法是前半句对，后半句不对。JavaScript 和 Java 名字相似，但语言特性却大不相同，这就跟北京的“海淀五路居”和“五路居”一样，差了得有 20 公里。

那前半句为什么是对的呢？JavaScript 1.0 连继承都没有，为什么又能称为面向对象的语言呢？

其实从我在前两讲中讲过的内容来看，JavaScript 1.0 确实已经可以将函数作为构造器，并且在函数中向它的实例（也就是 `this` 对象）抄写类声明的那些属性。在早期的面向对象理论里面，就已经可以称这个函数为**类**，而这个被创建出来的实例为**对象**了。

所以，有了类、对象，以及一个约定的构造过程，有了这三个东西，JavaScript 就声称了自己是一门“面向对象”的语言，并且还是一门“有类语言”。

所以 JavaScript 从 1.0 开始就有类，在这个类（也就是构造器）中采用的是所谓“类抄写”的方案，将类所拥有的属性声明一项一项地抄写到对象上面，而这个对象，就是我们现在大家都知道的 `this` 引用。

这样一来，一段声明类和构造对象的代码，大概写出来就是下面这个样子：

```
1 function Car() {  
2   this.name = "Car";  
3   this.color = "Red";  
4 }  
5  
6 var x = new Car();
```

 复制代码

类与构造器

由于在这样的构造过程中，`this`是作为`new`运算所构造出来的那个实例来使用的，因此JavaScript 1.0 约定全局环境中不能使用`this`的。因为全局环境与`new`运算无关，全局环境中也并不存在一个被`new`创建出来的实例。

然而随着JavaScript 1.1的到来，JavaScript 支持“原型继承”了，于是“类抄写”成为了一个过时的方案。对于继承性来说，它显得无用；对于一个具体的实例来说，它又具有“类‘说明了’实例的结构”这样的语义。

因此，从“**原型继承**”在JavaScript 中出现的第一天开始，“类继承 VS 原型继承”之间就存在不可调和的矛盾。在JavaScript 1.1中，类抄写是可以与原型继承混合使用的：

[📄 复制代码](#)

```
1 function Device() {
2   this.id = 0; // or increment
3 }
4
5 function Car() {
6   this.name = "Car";
7   this.color = "Red";
8 }
9
10 Car.prototype = new Device();
11
12 var x = new Car();
13 console.log(x.id); //
```

在这个例子中所创建出来的对象`x`是“`Car()`”的一个实例，但是在面向对象编程（OOP）中，`x`既是`Car()`的子类实例，也是“`Device()`”的子类实例，这是OOP的继承性所约定的基本概念。这正是这门语言很有趣的地方：**一方面使用了类继承的基础结构和概念，另一方面又要实现原型继承和基于原型链检查的逻辑。**例如：

[📄 复制代码](#)

```
1 # `x`是`Device()`的子类实例吗？
2 > x instanceof Device
```

```
3 true
```

这里的`instanceof`运算被实现为一个**动态地访问原型链**的过程：它将从`Car.prototype`属性逆向地在原型链中查到你指定的——“原型”。

首先，JavaScript 从对象`x`的内部结构中取得它的原型。这个原型的存在，与`new`运算是直接相关的——在早期的 JavaScript 中，有且仅有`new`运算会向对象内部写“原型”这个属性（称为“`[[Prototype]]`”内部槽）。由于 `new` 运算是依据它运算时所使用的构造器来填写这个属性的，所以这意味着如下的效果：

[复制代码](#)

```
1 // x = new Car()
2 x.[[Prototype]] === Car.prototype
```

在`instanceof`运算中，`x instanceof AClass`表达式的右侧是一个类名（对于之前的例子来说，它指向构造器 `Car`），但实际上 JavaScript 是使用`AClass.prototype`来做比对，对于“`Car()` 构造器”来说，就是“`Car.prototype`”。但是，如果上一个例子需要检查的是`x instanceof Device`，也就是“`Device.prototype`”，那么这二者显然是不等值的。

所以，`instanceof`运算会再次取“`x.[[Prototype]]`”这个内部原型，也就是顺着原型链向上查找：

[复制代码](#)

```
1 // 因为
2 x.[[Prototype]] === Car.prototype
3 // 且
4 Car.prototype = new Device()
5
6 // 所以
7 x.[[Prototype]].[[Prototype]] === Device.prototype
```

现在，由于在`x`的原型链上发现了“`x instanceof Device`”运算右侧的“`Device.prototype`”，所以这个表达式将返回 `True` 值，表明：

对象`x`是`Device()`或其子类的一个实例。

现在，对于大多数 JavaScript 程序员来说，上述过程应该都不是秘密，也并不是特别难解的核心技术。但是在它的实现过程中所带有的语言设计方面的这些历史痕迹，却不是那么容易一望即知的了。

ECMAScript 6 之后的类


在 ECMAScript 6 之前，JavaScript 中的**函数**、**类**和**构造器**这三个概念是混用的。一般来说，它们都被统一为“**函数 `Car()`**”这个基础概念，而当它用作“`x = new Car()`”这样的运算，或从`x.constructor`这样的属性中读取时，它被理解为**构造器**；当它用作“`x instanceof Car`”这样的运算，或者讨论 OOP 的继承关系时，它被理解为**类**。

习惯上，如果程序要显式地、字面风格地说明一个函数是构造器、或者用作构造过程，那么它的函数名应该首字母大写。同时，如果一个函数要被明确声明为“静态类（也就不需要创建实例的类，例如 `Math`）”，那么它的函数名也应该首字母大写。

NOTE: 仅从函数名的大小写来判断，只是惯例。没有任何方法来确认一个函数是不是“被设计为”构造器，或者静态类，又或者“事实上”是不是二者之一。

从 ECMAScript 6 开始，JavaScript 有了使用`class`来声明“类”的语法。例如：

```
1 class AClass {  
2     ...  
3 }
```

 复制代码

自此之后，JavaScript 的“类”与“函数”有了明确的区别：**类只能用 `new` 运算来创建，而不能使用“`()`”来做函数调用**。例如：

```
1 > new AClass()  
2 AClass {}  
3  
4 > AClass()  
5 TypeError: Class constructor AClass cannot be invoked without 'new'
```

 复制代码

在 ECMAScript 6 之后，JavaScript 内部是明确区分方法与函数的：不能对方法做 new 运算。例如：

 复制代码

```
1 # 声明一个带有方法的对象字面量
2 > obj = { foo() {} }
3 { foo: [Function: foo] }
4
5 # 对方法使用 new 运算会导致异常
6 > new obj.foo()
7 TypeError: obj.foo is not a constructor
```

注意这个异常中又出现了关键字 “constructor”。这让我们的讨论又一次回到了开始的话题：**什么是构造器？**

在 ECMAScript 6 之后，函数可以简单地分为三个大类：

1. 类：只可以做 new 运算；
2. 方法：只可以做调用 “()” 运算；
3. 一般函数：（除部分函数有特殊限制外，）同时可以做 new 和调用运算。

其中，典型的“方法”在内部声明时，有三个主要特征：

1. 具有一个名为“主对象[[HomeObject]]”的内部槽；
2. 没有名为“构造器[[Construct]]”的内部槽；
3. 没有名为“prototype”的属性。

后两种特征（没有[[Construct]]内部槽和prototype属性）完全排除了一个普通方法用作构造器的可能。对照来看，所谓“类”其实也是作为方法来创建的，但它有独立的构造过程和原型属性。

函数的“.prototype”的属性描述符中的设置比较特殊，它不能删除，但可以修改（‘writable’ is true）。当这个值被修改成 null 值时，它的子类对象是以 null 值为原型

的；当它被修改成非对象值时，它的子类对象是以 `Object.prototype` 为原型的；否则，当它是一个对象类型的值时，它的子类才会使用该对象作为原型来创建实例。

运算符 “new” 总是依照这一规则来创建对象实例 `this`。

不过，对于 “类” 和一般的 “构造器（函数）”，这个创建过程会略有不同。

创建 `this` 的顺序问题


如前所述，如果对 ECMAScript 6 之前的构造器函数（例如 `f`）使用 `new` 运算，那么这个 `new` 运算会使用 `f.prototype` 作为原型来创建一个 `this` 对象，然后才是调用 `f()` 函数，并将这个函数的执行过程理解为 “类抄写（向用户实例抄写类所声明的属性）”。从用户代码的视角上来看，这个新对象就是由当前 `new` 运算所操作的那个函数 `f()` 创建的。

这在语义上非常简洁明了：由于 `f()` 是 `this` 的类，因此 `f.prototype` 决定了 `this` 的原型，而 `f()` 执行过程决定了初始化 `this` 实例的方式。但是它带了一个问题：从 JavaScript 1.1 开始至今都困扰 JavaScript 程序员的问题：

无法创建一个有特殊性质的对象，也无法声明一个具有这类特殊性质的类。

这是什么意思呢？比如说，所有的函数有一个公共的父类 / 祖先类，称为 `Function()`。所以你可以用 `new Function()` 来创建一个普通函数，这个普通函数也是可以调用的，例如：

```
1 > f = new Function;  
2  
3 > f instanceof Function  
4 true  
5  
6 > f()  
7 undefined
```

 复制代码

你也确实可以用传统方法写一个 `Function()` 的子类，但这样的子类创建的实例就不能调用。例如：

```
1 > MyFunction = function() {};  
2  
3 > MyFunction.prototype = new Function;  
4  
5 > f = new MyFunction;  
6  
7 > [f instanceof MyFunction, f instanceof Function]  
8 [ true, true ]  
9  
10 > f()  
11 TypeError: f is not a funct
```

至于原因，你可能也已经知道了：JavaScript 所谓的函数，其实是“一个有[[Call]]内部槽的对象”。而`Function()`作为 JavaScript 原生的函数构造器，它能够在创建的对象（例如`this`）中添加这个内部槽，而当使用上面的继承逻辑时，用户代码（例如`MyFunction()`）就只是创建了一个普通的对象，因为用户代码没有能力操作 JavaScript 引擎层面才支持的那些“内部槽”。

所以，有一些“类 / 构造器”在 ECMAScript 6 之前是不能派生子类的，例如 `Function`，又例如 `Date`。

而到了 ECMAScript 6，它的“类声明”采用了不同的构造逻辑。ECMAScript 6 要求所有子类的构造过程都不得创建这个`this`实例，并主动的把这个创建的权力“交还”给父类、乃至祖先类。这也就是 ECMAScript 6 中类的两个著名特性的由来，即，如果类声明中通过 `extends` 指定了父类，那么：

1. 必须在构造器方法（constructor）中显式地使用`super()`来调用父类的构造过程；
2. 在上述调用结束之前，是不能使用`this`引用的。

显然，真实的`this`创建就通过层层`super()`交给了父类或祖先类中支持创建这个实例的构造过程。这样一来，子类中也能得到一个“拥有父类所创建的带有内部槽的”实例，因此上述的`Function()`和`Date()`等等的子类也就可以实现了。例如：

```
1 > class MyFunction extends Function { }  
2
```



```
3 > f = new MyFunction;  
4  
5 > f()  
6 undefine
```

在上面个例子中，`MyFunction()` 的类声明中缺省了 “`constructor()`” 构造方法。这种情况下 JavaScript 会为它自动创建一个，并且其内部也仅有一个 “`super()`” 代码。关于这些过程的细节，我将留待下一讲再具体地与你解析。在这里，你最应该关注的是这个过程带来的必然结果：

ECMAScript 6 的类是由父类或祖先类创建`this`实例的。

不过仍然有一点是需要补充的：如果类声明`class`中不带有`extends`子句，那么它所创建出来的类与传统 JavaScript 的函数 / 构造器是一样的，也就是由自己来创建`this`对象。很显然，这是因为它无法找到一个显式指示的父类。不过关于这种情况，仍然隐藏了许多实现细节，我将会在下一讲中与你一起来学习它。

用户返回 new 的结果

在 JavaScript 中关于 `new` 运算与构造函数的最后一个有趣的设计，就是**用户代码可以干涉 new 运算的结果值**。默认情况下，这个结果就是上述过程所创建出来的`this`对象实例，但是用户可以通过在构造器函数 / 方法中使用`return`语句来显式地重置它。

这也是从 JavaScript 1.0 就开始具有的特性。因为 JavaScript 1.x 中的函数、类与构造器是混用的，所以用户代码在函数中“返回些什么东西”是正常的语法，也是正常的逻辑需求。但是 JavaScript 要求在构造器中返回的数据必须是一个对象，否则就将抛出一个运行期的异常。


从 ECMAScript ed3 开始，检测构造器返回值的逻辑从`new`运算符中移到了`[[Construct]]`的处理过程中，并且重新约定：当构造器返回无效值（非对象值或 `null`）时，使用原有已经创建的`this`对象作为构造过程`[[Construct]]`的返回值。

因此到了 ECMAScript 6 之后，那些一般函数，以及非派生类，就延续了这一约定：**使用已经创建的`this`对象来替代返回的无效值**。这意味着它们总是能返回一个对象，要么是

`new` 运算按规则创建的 `this`，要么是用户代码返回的对象。

NOTE: 关于为什么非派生类也支持这一约定的问题，我后续的课程中会再次讲到。基本上来说，你可以认为这是为了让它与一般构造器保持足够的“相似性”。

然而严格来说，引擎是不能理解“为什么用户代码会在构造器中返回一个一般的值类型数据”的。因为对于类的预期是返回一个对象，返回这种“无效值”是与预期矛盾的。因此，对于那些派生的子类（即声明中使用了`extends`子句的类），ECMAScript 要求严格遵循“不得在构造器中返回非对象值（以及 `null` 值）”的设计约定，并在这种情况下直接抛出异常。例如：

 复制代码

```
1  ## (注: ES3 之前将抛出异常)
2  > new (function() {return 1});
3  {}
4
5  ## 非派生类的构造方法返回无效值
6  > new (class { constructor() { return 1 } })
7  {}
8
9  ## 派生类的构造方法返回无效值
10 > new (class extends Object { constructor() { return 1 } })
11 TypeError: Derived constructors may only return object or undefined
```

知识回顾

今天这一讲的一些知识点，是与你学习后续的专栏内容有关的。包括：

1. 在使用类声明来创建对象时，对象是由父类或祖先类创建的实例，并使用`this`引用传递到当前（子级的）类的。
2. 在类的构造方法和一般构造器（函数）中返回值，是可以影响 `new` 运算的结果的，但 JavaScript 确保 `new` 运算不会得到一个非对象值。
3. 类或构造器（函数）的首字母大写是一种惯例，而不是语言规范层面的约束。
4. 类继承过程也依赖内部构造过程（`[[Construct]]`）和原型属性（`prototype`），并且类继承实际上是原型继承的应用与扩展，不同于早期 JavaScript 1.0 使用的类抄写。

无论如何，从 JavaScript 1.0 开始的“类抄写”这一特性依然是可用的。无论是在普通函数、类还是构造器中，都可以向`this`引用上抄写属性，但这个过程变得与“如何实现继承性”完全无关。这里的`this`可以是函数调用时传入的，而不再仅仅来自于 `new` 运算的内置的构造过程创建。

思考题

1. 除使用 `new X` 运算，还有什么方法可以创建新的对象？
2. 在 ECMAScript 6 之后，除了 `new X` 之外，还有哪些方法可以操作原型 / 原型链？

这些问题既是对本小节内容的回顾，也是下一阶段的课程中会用到的一些基础知识。建议你好好的寻求一下答案。

最后，如果你喜欢我的分享，也欢迎你把文章分享给你的朋友。

点击参与 

打卡 46 天，彻底搞定
JavaScript


扫一扫参与小程序打卡



新版升级：点击「 请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 1 in 1..constructor：这行代码的结果值，既可能是`true`，也可能是`false`

下一篇 14 | `super.xxx()`：虽然直到ES10还是个半吊子实现，却也值得一讲

精选留言 (5)

写留言



行问

2019-12-13

谈谈今天的理解：

在 instanceof 运算中，`x instanceof AClass` 表达式的右侧是一个类名（对于 instanceof 的理解之前是有误解，今天才领悟到）

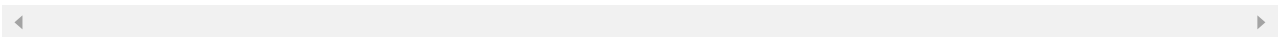
...

展开

作者回复：这个this实例的确是由父类或祖先类创建的。但它不是“继承”来的，因为“继承”这个说法严格来说在JavaScript中就是原型继承，而这个this不是靠原型继承来“传递到”子类的。

在`super()`调用之前，当前函数——例如子类的构造器——无法访问this，是它的作用域里面没有this这个名字（因为还没有被创建出来嘛）。而`super()`调用之后，JavaScript引擎会把this“动态地添加到”作用域中，于是this就能访问了。

这个“动态的添加”其实很简单，因为`super`是子类向父类调用的，所以显然父类调用结束并退出时的当前作用域（或环境）就是子类的，因此ECMAScript约定在退出`super()`的时候就把已经创建好的this直接“抄写”给当前环境就可以了。这里大概只有一两行代码，很简单的。^^



2



行问

2019-12-13

`Object.create()`

`Object.defineProperty()`

ES6 的 proxy 和 Reflect

展开



2



潇潇雨歇

2019-12-17

ES6操作原型/原型链方法：`Object.create()`、`Object.setPrototypeOf()`、`Object.getPrototypeOf()`



1



潇潇雨歇



2019-12-16

1、Object.create()

展开 ▾



sprinty

2019-12-13

```
function X() {  
  this.x = 4  
}
```

...

展开 ▾

