

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》

课程介绍 >



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，当时也还是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 JavaScript1.0 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，但语言特性却大是不同，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，在一个函数里面不停地向 this 对象写属性，最后再用 new 运算符来创建一下它的实例就好了。

```
1 function Car() {  
2   this.name = "Car";  
3   this.color = "Red";  
4 }  
5  
6 var x = new Car();  
7 ...
```

 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。

例如，你可以用类抄写的方式写一个 `Device()` 类，然后再写一个 `Car()` 类，最后你可以将 `Car()` 类的原型指向 `Device`。这一切都是合理的、正常的写法。

 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); //
```

于是现在，你可以用 `new` 运算来创建子类 `Car()` 的实例了，例如按照以前的习惯，我们称这个实例为 `x`，这也仍然没有问题。

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，你用`x instanceof Device`这样的代码来检查一下，看看“`x`是不是`Device()`的子类实例”。



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

于是，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，所以这意味着它在实际实现时，将 Car.prototype 这个值，直接给填到 x 对象的内部属性去了。

```
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来做比对的，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]] [[Prototype]]”这个内部原型，也就是顺着原型链向上查找，并且你将找到一个等值于“x 的内部原型”的东西。

```
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 {}
```

 复制代码




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

如果你尝试将“ES6 的类”作为函数调用，那么 JavaScript 就会抛出一个异常。

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。如果你尝试这样做，JavaScript 也会抛一个异常出来，提示你“这个函数不是一个构造器 (is not a constructor) ”。例如：

```
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
```

 复制代码

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。但是它带来了一个问题，一个从 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，在JavaScript 中这是很正常的用法，例如：

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

 复制代码



接下来，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：

[复制代码](#)

```
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`创建就通过层层的`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，你可以在 class MyFunction 的声明中直接用 extends 指示父类为 Function。

 复制代码

```
1 > class MyFunction extends Function { }
2
3 > f = new MyFunction;
4
5 > f()
6 undefine
```

这样一来，即使MyFunction()的类声明中缺省了“constructor()”构造方法，这种情况下 JavaScript 会在这种情况下为它自动创建一个，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**。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。

这个处理的约定，从 ECMAScript ed3 开始有了些变化。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 JavaScript1.0 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. 除了使用 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。建议你好好地寻求一下答案。

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。

分享给需要的人，Ta购买本课程，你将得 20 元

 生成海报并分享

 赞 4  提建议

© 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，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



JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (18)

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）

ECMAScript 6 的类是由父类或祖先类创建 this 实例的（也就是 this 是继承而来的，也能够契合前面说的“在调用结束之前，是不能使用 this 引用。不知道这个理解能否正确”）

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。^^.





10



青史成灰

2020-01-12

首先谢谢老师每次都认真的回答每个问题，但是下面的这个问题，我一下子没想明白。。。

...

```
function Device() {  
  this.id = 0; // or increment  
}
```

```
function Car() {  
  this.name = "Car";  
  this.color = "Red";  
}
```

```
Car.prototype = new Device();
```

```
var x = new Car();  
console.log(x.id); //  
...
```

这个例子中，为什么`x.constructor === Device`，按我的理解应该是`x.constructor === Car`才对。但是如果我把`Car.prototype = new Device()`这行代码注释掉，那么就符合我的理解了。。。

作者回复: `x.constructor === Car.prototype.constructor === (new Device()).constructor === Device.prototype.constructor`

这个正是因为(new Device()) 导致的原型继承效果。而且，这种继承方法是ES6之前的标准写法，因为那个时代有且仅有`new`运算会在Car/Device之间维护内部原型继承关系，以确保instanceof运算有效。——instanceof运算是检查Car.[[prototype]]这个内部原型的，而不是检查Car.prototype这个属性。

所以，标准的、完整的在ES6之前实现原型继承的代码“总是（且必然是）”两行，而不是示例中的一行：

...

```
Car.prototype = new Device();  
Car.prototype.constructor = Car;  
...
```

另外，也有一种写法，用于在Car()这个函数内部来维护这个constructor属性。例如：



```
...  
  
function Car() {  
  this.constructor = Car; // Or, arguments.callee  
  
  ...  
}  
  
Car.prototype = new Device;  
...
```



👍 5



行问

2019-12-13

Object.create()
Object.defineProperty()
ES6 的 proxy 和 Reflect



👍 4



油菜

2020-11-16

“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。”

老师，我的测试结果和这个结论不大一样。

```
function F(){ this.name1 = 'father'}  
function S1(){ this.name1 = 'son1'}  
F.prototype = null;  
S1.prototype =F;  
var s1 = new S1();  
s1.constructor.prototype; //原型对象是[Function] 而不是null  
s1 instanceof S1; //true;  
s1 instanceof F; //TypeError: Function has non-object prototype 'null' in instanceof check
```

class S2 extends F {}
var s2 = new S2();
s2.constructor.prototype; //原型对象是S2 {}, 不是null




```
s2 instanceof S2; // true;
```

```
s2 instanceof F; //Function has non-object prototype 'null' in instanceof check
```

作者回复: 这个地方是讲错了, 这里混淆了一些内容, 一部分是在讨论“它的实例”, 另一部分则是在讨论“它的子类对象”。有关的内容应该改成这样 (用 【】 标出修订之处):

```
...
```

```
# 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) 。
```

```
> function f() {}
```

```
> Object.getOwnPropertyDescriptor(f, 'prototype')
```

```
{ value: f {},  
  writable: true,  
  enumerable: false,  
  configurable: false }
```

```
# 当这个值被修改成 null 值时, 它的子类对象是以 null 值为 【祖先类】 原型的;
```

```
> f.prototype = null
```

```
> class MyObject extends f {}
```

```
> Object.getPrototypeOf(Object.getPrototypeOf(new MyObject)) === null  
true
```

```
# 当它被修改成非对象值时, 【它将不能用于派生子类; 但作为构造器, 它的实例对象】 是以 Object.prototype 为原型的;
```

```
> f.prototype = 1
```

```
> class MyObject2 extends f {} // 异常, 不能派生子类
```

```
> Object.getPrototypeOf(new f) === Object.prototype  
true
```

```
# 否则, 当它是一个对象类型的值时, 【该函数】 才会使用该对象作为原型来创建实例。
```

```
> f.prototype = new Object
```

```
> Object.getPrototypeOf(new f) === f.prototype
```

```
...
```



潇潇雨歇

2019-12-17

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





westfall

2021-02-04

请问老师，constructor 这个属性是不是可有可无的？

作者回复: 从原理上来说，在ES6之后，使用类继承的时候这个属性其实意义就不大了。像生成器（以及异步生成器）等函数的prototype上也就根本没有这个属性，表明这它们是不可构造的。

但是有些操作是需要引用这个属性的，例如Symbol.species这个属性会检测对象的构造器，从而创建一个“相同类型的”实例的，这主要会用到数组和Promise对象上。



1



二二

2020-10-14

老师您好，关于：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。

但是我看babel将es6转成es5是可以实现对于Function的继承，并调用的，请问babel是怎么达到这个效果的呢？

作者回复: babel转码到es5的时候是加了一堆生成代码的，这些代码模拟了es6类创建的过程（例如为了模拟super关键字，它就维护了一个类继承树）。

所以babel中，如果你用es6的class来写，那么转换到es5的时候，这堆代码也仍然是按es6的过程来创建的。

你可以尝试一下在这里写一段es6的class代码，然后转换成es5的，就知道babel干了些什么了：
<https://es6console.com/>



1



人间动物园

2020-05-26

思考题 1，

直接执行一个函数也可以创建新的对象：

```
function Person(name){  
  this.name = name;  
  return this;  
}  
person1 = Person('xiaoming');
```



作者回复: 不能。

这个this是global，并不是创建的新对象。

...

...

```
person1 = Person('xiaoming');  
console.log(person1 === global); // true
```

...

共 3 条评论 >

👍 1



CoolSummer

2020-02-23

1.创建新的对象

字面量方法创建、Object.create()、工厂模式、构造函数模式

2.操作原型 / 原型链

Object.defineProperty()/Object.getProperty()

ES6 的 proxy 和 Reflect

作者回复: 答案1不够完整，例如事实上类声明也会导致对象的创建（例如class MyClass ...会导致MyClass.prototype创建）。

答案2就离题了。例如defineProperty并不操作原型也不会操作原型链。举两个正确的例子，比如说Object.setPrototypeOf()就是操作原型链的，又例如说class MyClass extends ...会导致MyClass和MyClass.prototype的原型都被操作。



👍 1



Smallfly

2020-02-20

老师文中多次提到类继承，您这里指的是从类抄写属性到对象么？我的理解是这个过程属于对象的实例化。JS 只有原型继承一种方式。

还是说因为实例化的过程，包含向 this 对象写入原型，所以称它为类继承，并且包含原型继承？

作者回复: 是指从ES6开始的JavaScript语法：

...



```
class X extends ParentCls {
```

```
...
```

```
}
```

```
...
```

在JavaScript中，尽管这个类继承语法仍然是通过原型继承来实现的，但是它的确在语义与概念上是类继承的，并且由于有了“使用super来指向父类”这样的特性，所以在类继承的概念上也是完整的。



1



小胖

2019-12-25

{Foo () {}}创建的Foo方法不能使用new关键字调用；

但{Foo: function Foo() {}}是可以的。

所以说，ES6提供的方法简写形式添加的方法和不使用简写形式添加的方法是有区别的。

作者回复：不。

ES6提供的并不“仅仅是”方法的简写。这种语法风格声明是的“真正的”对象方法。——在ES6之后，真正的对象方法（Method）是与传统的函数在性质上不完全相同的。

而传统的声明方式，也就是“{foo: function ...}”被理解为“函数类型的对象属性”，它是一个Normal Functions，也就是一般的函数，而并非“ES6规范中的对象方法”。

方法是特殊的函数，就如同ES6中“类”是特殊的函数一样。



1



Kids See Ghost

2022-01-20

请问老师，在规范里有没有具体哪里讲了如何比较两个object的？比如为什么两个empty object不想等`{} !== {}`在规范里找了很久没找到。在规范里 <https://tc39.es/ecma262/#sec-samevaluenonnumeric> 只说了 "7. If x and y are the same Object value, return true. Otherwise, return false. "。但是没有具体地说怎么样的object value算一样，怎么样算不一样。比如`{}`和`{}`就不一样。

作者回复：这就跟既有的知识是一样的了，比较两个对象，就是比较它们的地址（指针）；比较两个值，就是比较地址上的数据。我想可能是因为这个原因，反正ECMAScript里面也没有写~~又也许，还与具体引擎的实现有关？就是说引擎想怎么实现就怎么实现.....





Chor

2021-05-31

老师，下面这段话：

“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”是否可以改成“它的子类对象是以 Object.prototype 为原型的”呢？

因为：

```
const Fn = function( ){ }
```

```
Fn.prototype = null
```

```
const obj = new Fn( )
```

```
Object.getPrototypeOf(obj) === Object.prototype // true
```

作者回复：多谢多谢。

这个地方是写错了。在之前的评论回复中有说明这里的修订，但是好象极客时间的编辑没有把这个修订的内容更新掉，所以.....

你先用关键字“修改成非对象值时”在评论回复里找一下更新的内容。我请编辑老师尽快改掉。



孜孜

2020-08-04

我还是理解不了以下代码？`testF.call` 明明是function啊。

```
let testF = Object.create(Function.prototype);
```

```
testF.call({}) //Uncaught TypeError: testF.call is not a function
```

作者回复：《JavaScript语言精髓与编程实践》的“3.4.5 特殊效果的继承”专门说过这种问题，通过这种方式来派生（或构建实例）是不能继承“函数（的特殊效果）”的——也就是说得到的函数是不能执行的。

可以用class来解决这个问题。例如：`class FF extends Function {}`，然后`testF = new FF`。



HoSalt

2020-05-23

「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」

老师没有prototype能直接看到，「没有[[Construct]]内部槽」这个能看见吗，这是在方法对象上还是方法对象的原型链上？



作者回复: 好像在chrome的控制台里面可以看到这些内部槽的状况。它的调试器支持这个



HoSalt

2020-05-23

在ES6之前 new X 先生成this,再将设置this.__proto__=X.prototype
那在class 继承中这一步是在何时执行的, 最后一个super执行完后, 还是最祖先类生成this的时候执行的?

```
class X extends Y{}
```

```
new X
```

作者回复: 是祖先类生成this的时候写的原型。这也就是new.target存在的意义, 它会一直传递到祖先类。



潇潇雨歇

2019-12-16

1、Object.create()



weineel 

2019-12-13

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

```
function Y() {  
  this.y = 5  
}
```

// Y.prototype = new X(), new 运算可以用以下三行模拟。

```
var xxx = {}
```

```
X.apply(xxx)
```

```
xxx.__proto__ = X.prototype
```

```
Y.prototype = xxx
```




```
// var y = new Y()  
var y = {}  
Y.apply(y)  
y.__proto__ = Y.prototype
```

共 1 条评论 >

