

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

2019-12-16 周爱民

JavaScript核心原理解析

[进入课程 >](#)



讲述: 周爱民

时长 20:26 大小 18.72M



你好, 我是周爱民, 接下来我们继续讲述 JavaScript 中的那些奇幻代码。

今天要说的内容, 打根儿里起还是得从 JavaScript 的 1.0 谈起。在此前我已经讲过了, JavaScript 1.0 连继承都没有, 但是它实现了以“类抄写”为基础的、基本的面向对象模型。而在此之后, 才在 JavaScript 1.1 开始提出, 并在后来逐渐完善了原型继承。

这样一来, 在 JavaScript 中, 从概念上来讲, 所谓对象就是一个从原型对象衍生过来的实例, 因此这个子级的对象也就具有原型对象的全部特征。

然而，既然是子级的对象，必然与它原型的对象有所不同。这一点很好理解，如果没有不同，那就没有必要派生出一级关系，直接使用原型的那一个抽象层级就可以了。

所以，有了原型继承带来的子级对象（这样的抽象层级），在这个子级对象上，就还需要有让它们跟原型表现得有所不同的方法。这时，JavaScript 1.0 里面的那个“类抄写”的特性就跳出来了，它正好可以通过“抄写”往对象（也就是构造出来的那个 this）上面添加些东西，来制造这种不同。

也就是说，JavaScript 1.1 的面向对象系统的设计原则就是：**用原型来实现继承，并在类（也就是构造器）中处理子一级的抽象差异**。所以，从 JavaScript 1.1 开始，JavaScript 有了自己的面向对象系统的完整方案，这个示例代码大概如下：

 复制代码

```
1 // 这里用于处理“不同的东西”
2 function CarEx(color) {
3     this.color = color;
4     ...
5 }
6
7 // 这里用于从父类继承“相同的东西”
8 CarEx.prototype = new Car("Eagle", "Talon TSi", 1993);
9
10 // 创建对象
11 myCar = new CarEx("red")
```

你看，一个对象系统既能处理继承关系中那些“相同的东西”，又能处理“不同的东西”，所以显而易见：**这个系统能处理基于对象的“全部的东西”**。正是因为这种概念上的完整性，所以从 JavaScript 1.1 开始，一直到 ECMAScript 5 都在对象系统的设计上没能再有什么突破。

为什么要有 super?

但是有一个东西很奇怪，这也是对象继承的典型需求，就是说：子级的对象除了要继承父级的“全部的东西”之外，它还要继承“全部的能力”。

为什么只继承“全部的东西”还不够呢？如果只有全部的东西，那子级相对于父级，不过是一个系统的静态变化而已。就好像一棵枯死了的树，往上面添加些人造的塑料的叶子、假的

果子，看起来还是树，可能还很好看，但根底里就是没有生命力的。而这样的一棵树，只有继承了原有的树的生命力，才可能是一棵活着的树。

如果继承来的树是活着的，那么装不装那些人造的叶子、果子，其实就不要紧了。

然而，传统的 JavaScript 却做不到“继承全部的能力”。那个时候的 JavaScript 其实是能够在一定程度上继承来自原型的“部分能力”的，譬如说原型有一个方法，那么子级的实例就可以使用这个方法，这时候子级也就继承了原型的能力。

然而这还不够。譬如说，如果子级的对象重写了这个方法，那么会怎么样呢？

在 ECMAScript 6 之前，如果发生这样的事，那么对不起：**原型中的这个方法相对于子级对象来说，就失效了。**

原则上来讲，在子级对象中就再也找不到这个原型的方法了。这个问题非常地致命：这意味着子级对象必须重新实现原型中的能力，才能安全地覆盖原型中的方法。如果是这样，子级对象就等于要重新实现一遍原型，那继承性就毫无意义了。

这个问题追根溯源，还是要怪到 JavaScript 1.0~1.1 的时候，设计面向对象模型时偷了的那一次懒。也就是直接将“类抄写”用于实现子级差异的这个原始设计，太过于简陋。“类抄写”只能处理那些显而易见的属性、属性名、属性性质，等等，却无法处理那些“方法 / 行为”背后的逻辑的继承。

由于这个缘故，JavaScript 1.1 之后的各种大规模系统中，都有人不断地在跳坑和补坑，致力于解决这么一个问题：**在“类抄写”导致的子类覆盖中，父类的能力丢失了。**

为了解决这种继承问题，ECMAScript 6 就提出了一个标准解决方案，这就是今天我们讲述的这行代码中“super”这个关键字的由来。ECMAScript 6 约定，如果父类中的名字被覆盖了，那么你可以在子类中用 super 来找到它们。

super 指向什么？

既然我们知道 super 出现的目的，就是解决父类的能力丢失这一问题，那么我们也就很容易理解一个特殊的语言设计了：**在 JavaScript 中，super 只能在方法中使用。**所谓方法，

其实就是“类的，或者对象的能力”，`super`正是用来弥补覆盖父类同名方法所导致的缺陷，因此只能出现在方法之中，这也就是很显而易见的事情了。

当然，从语言内核的角度上来说，这里还存在着一个严重的设计限制，这个问题是：怎么找到父类？

在传统的 JavaScript 中，所谓方法，就是函数类型的属性，也就是说它与一般属性并没有什么不同（可以被不同的对象抄写来抄写去）。其实，方法与普通属性没有区别，也是“类抄写”机制得以实现的核心依赖条件之一。然而，这也就意味着所谓“传统的方法”没有特殊性，也就没有“归属于哪个类或哪个对象”这样的性质。因此，这样的方法根本上也就找不到它自己所谓的类，进而也就找不到它的父类。

所以，实现 `super` 这个关键字的核心，在于为每一个方法添加一个“它所属的类”这样的性质，这个性质被称为“主对象（HomeObject）”。

所有在 ECMAScript 6 之后，通过方法声明语法得到的“方法”，虽然仍然是函数类型，但是与传统的“函数类型的属性（即传统的对象方法）”存在着一个根本上的不同：这些新的方法增加了一个内部槽，用来存放这个主对象，也就是 ECMAScript 规范中名为 `[[HomeObject]]` 的那个内部槽。这个主对象就用来对在类声明，或者字面量风格的对象声明中，（使用方法声明语法）所声明的那些方法的主对象做个登记。这有三种情况：

1. 在类声明中，如果是类静态声明，也就是使用 `static` 声明的方法，那么主对象就是这个类，例如 `AClass`。
2. 就是一般声明，那么该方法的主对象就是该类所使用的原型，也就是 `AClass.prototype`。
3. 第三种情况，如果是对象声明，那么方法的主对象就是对象本身。

但这里就存在一个问题了：`super` 指向的是父类，但是对象字面量并不是基于类继承的，那么为什么字面量中声明的方法又能使用 `super.xxx` 呢？既然对象本身不是类，那么 `super` “指向父类”，或者“**用于解决覆盖父类能力**”的含义岂不是就没了？

这其实又回到了 JavaScript 1.1 的那项基础设计中，也就是“用原型来实现继承”。

原型就是一个对象，也就是说本质上子类或父类都是对象；而所谓的类声明只是这种继承关系的一个载体，真正继承的还是那个原型对象本身。既然子类 and 父类都可能是，或者说必须是对象，那么对象上的方法访问“父一级的原型上的方法”就是必然存在的逻辑了。

出于这个缘故，在 JavaScript 中，只要是方法——并且这个方法可以在声明时明确它的“主对象 (HomeObject)”，那么它就可以使用 `super`。这样一来，对象方法也就可以引用到它父级原型中的方法了。这一点，其实也是“利用原型继承和类抄写”来实现面向对象系统时，在概念设计上的一个额外的负担。

但接下来所谓“怎么找到父类”的问题就变得简单了：当每一个方法都在其内部登记了它的主对象之后，ECMAScript 约定，**只需要在方法中取出这个主对象 HomeObject，那么它的原型就一定是所谓的父类**。这很明显，因为方法登记的是它声明时所在的代码块的 HomeObject，也就是声明时它所在的类或对象，所以这个 HomeObject 的原型就一定是父类。也就是把“通过原型继承得到子类”的概念反过来用一下，就得到了父类的概念。

super.xxx()

我们今天讲的内容到现在为止，只说明了两件事。第一件，是为什么要有 `super`；第二件，就是 `super` 指向什么。

接下来我们要讲 `super.xxx`。简单地说，这就是个属性存取。这从语法上一看就明白了，似乎没有什么特殊的，对吧？未必如此！

回顾一下我们在第 7 讲中讲述到的内容：`super.xxx` 在语法上只是属性存取，但 `super.xxx()` 却是方法调用；而且，`super.xxx()` 是表达式计算中罕见的、在双表达式连用中传递引用的一个语法。

所以，关键不是在于 `super.xxx` 如何存取属性，而在于 `super.xxx` 存取到的属性在 JavaScript 内核中是一个“引用”。按照语法设计，这个引用包括了左侧的对象，并且在它连用“函数调用 ()”语法的时候，将这个左侧的对象作为 `this` 引用传入给后者。


更确切地说，假如我们要问“在 `super.xxx()` 调用时，函数 `xxx()` 中得到的 `this` 是什么”，那么按照传统的属性存取语法可以推论出来的答案是：这个 `this` 值应该是 `super`！

但是很不幸，这不是真的。

super.xxx() 中的 this 值

在 super.xxx() 这个语法中，xxx() 函数中得到的 this 值与 super——没有“一点”关系！不过，还是有“半点”关系的。不过在具体讲这“半点”关系之前呢，我需要先讲讲它会得到一个怎样的 this，以及如何能得到这个 this。

super 总是在一个方法（如下例中的 obj.foo 函数）中才能引用。这个是我们今天这一讲前半段中所讨论的。这个方法自己被调用的时候，理论上来说应该是类似下面这样的方式：

 复制代码

```
1 obj = {  
2   foo() {  
3     super.xxx();  
4   }  
5 }  
6  
7 // 调用 foo 方法  
8 obj.foo();
```

调用这个 foo() 方法时，它总是会将 obj 传入作为 this，所以 foo() 函数内的 this 就应该是 obj。而我们看看其中的 super.xxx()，我们期望它调用父类的 xxx() 方法时，传入的当前实例（也就是 obj）正好是在是在 foo() 函数内的那个 this。继承来的行为，应该是施加给现实中的当前对象的，施加给原型（也就是这里的 super）是没什么用的。所以，在这几个操作符的连续运算中，只需要把当前函数中的那个 this 传给父类 xxx() 方法就行了。

然而怎么传呢？

我们说过，super.xxx 在语言内核上是一个“引用”，ECMAScript 约定将这个语法标记成“Super 引用 (SuperReference)”，并且为这个引用专门添加了一个 thisValue 域。这个域，其实在函数的上下文中也有一个（相同名字的，也是相同的含义）。然后，ECMAScript 约定了优先取 Super 引用中的 thisValue 值，然后再取函数上下文中的。

如此一来，在函数（也就是我们这里的方法）中取 super 的 this 值时，就得到了为 super 专门设置的这个 this 对象。而且，事实上这个 thisValue 是在执行引擎发现 super 这个标识符 (GetIdentifierReference) 的时候，就从当前环境中取出来并绑定给 super 引用的。

回顾上述过程，`super.xxx()` 这个调用中有两个细节需要你多加注意：

1. `super` 关键字所代表的父类对象，是通过当前方法的 `[[HomeObject]]` 的原型链来查找的；
2. `this` 引用是从当前环境所绑定的 `this` 中抄写过来，并绑定给 `super` 的。

为什么要关注上面这两个特别特别小的细节呢？

我们知道，在构造方法中，`this` 引用（也就是将要构造出来的对象实例）事实上是由祖先类创建的。关于这一点如果你印象不深了，请回顾一下上一讲（也就是第 13 讲 “new X”）的内容。那么，既然 `this` 是祖先类创建的，也就意味着在刚刚进入构造方法时，`this` 引用其实是没有值的，必须采用我们这里讲到的“继承父类的行为”的技术，让父类以及祖先类先把 `this` 构造出来才行。

所以这里就存在了一个矛盾，这是一个“先有鸡，还是先有蛋”的问题：一方面构造方法中要调用父类构造方法，来得到 `this`；另一方面调用父类方法的 `super.xxx()` 需要先从环境中找到并绑定一个 `this`。

概念上这是无解的。


ECMAScript 为此约定：只能在调用了父类构造方法之后，才能使用 `super.xxx` 的方式来引用父类的属性，或者调用父类的方法，也就是访问 `SuperReference` 之前必须先调用父类构造方法（这称为 `SuperCall`，在代码上就是直接的 `super()` 调用这一语法）。这其中也隐含了一个限制：在调用父类构造方法时，也就是 `super()` 这样的代码中，`super` 是不绑定 `this` 值的，也不在调用中传入 `this` 值的。因为这个阶段根本还没有 `this`。

`super()` 中的父类构造方法

事实上不仅仅如此。因为如果你打算调用父类构造方法（注意之前讲的是父类方法，这里是父类构造方法，也就是构造器），那么很不幸，事实上你也找不到 `super`。

以 `new MyClass()` 为例，类 `MyClass` 的 `constructor()` 方法声明时，它的主对象其实是 `MyClass.prototype`，而不是 `MyClass`。因为，后者是静态类方法的主对象，而显然 `constructor()` 方法只是一般方法，而不是静态类方法（例如没有 `static` 关键字）。所以，

在 MyClass 的构造方法中访问 super 时，通过 HomeObject 找到的将是原型的**父级对象**。而这并不是父类构造器，例如：

 复制代码

```
1 class MyClass extends Object {  
2   constructor() { ... } // <- [[HomeObject]] 指向 MyClass.prototype  
3 }
```

我们知道，**super()** 的语义是“调用父类构造方法”，也就应当是**extends**所指定的**Object()**。而上面讲述的意思是说，在当前构造方法中，无法通过 [[HomeObject]] 来找到父类构造方法。

那么 JavaScript 又是怎么做的呢？其实很简单，在这种情况下 JavaScript 会从当前调用栈上找到当前函数——也就是 new MyClass() 中的当前构造器，并且返回该构造器的原型作为 super。

也就是说，类的原型就是它的父类。这又是我们在上面讨论过的：把“通过原型继承得到子类”的概念反过来用一下，就得到了父类的概念。

为什么构造方法不是静态的？

也许你会提一个问题：为什么不直接将 constructor() 声明为类静态方法呢？事实上我在分析清楚这个 super() 逻辑的时候，第一反应也是如此。类静态方法中的 [[HomeObject]] 就是 MyClass 自己啊，如果这样的话，就不必换个法子来找到 super 了。

是的，这个逻辑没错。但是我们记得，在构造方法 constructor() 中，也是可以使用 super.xxx() 的，与调用父类一般方法（即 MyClass.prototype 上的原型方法）的方式是类似的。

因此，根本问题在于：一方面 super() 需要将父类构造器作为 super，另一方面 super.xxx 需要引用父类的原型上的属性。

这两个需求是无法通过同一个 [[HomeObject]] 来实现的。这个问题只会出现在构造方法中，并且也只与 super() 冲突。所以 super() 中的 super 采用了别的方法（这里是指在调

用栈上查找当前函数的方式) 来查找当前类以及父类, 而且它也是作为特殊的语法来处理的。

现在, JavaScript 通过当前方法的 `[[HomeObject]]` 找到了 `super`, 也找到了它的属性 `super.xxx`, 这个称为 Super 引用 (SuperReference); 并且在背地里, 为这个 SuperReference 绑定了一个 `thisValue`。于是, 接下来它只需要做一件事就可以了, 调用 `super.xxx()`。

知识回顾

下面我来为第 13 讲做个总结, 这一讲有 4 个要点:

1. 只能在方法中使用 `super`, 因为只有方法有 `[[HomeObject]]`。
2. `super.xxx()` 是对 `super.xxx` 这个引用 (SuperReference) 作函数调用操作, 调用中传入的 `this` 引用是在**当前环境的上下文**中查找的。
3. `super` 实际上是在通过原型链查找父一级的对象, 而与它是不是**类继承**无关。
4. 如果在类的声明头部没有声明 `extends`, 那么在构造方法中也就不能调用父类构造方法。

注: 第 4 个要点涉及到两个问题: 其一是它显然 (显式的) 没有所谓 `super`, 其二是没有声明 `extends` 的类其实是采用传统方式创建的构造器。但后者不是在本讲中讨论的内容。

思考题

1. 请问 `x = super.xxx.bind(...)` 会发生什么? 这个过程 `thisValue` 会如何处理?
2. `super` 引用是动态查找的, 但类声明是静态声明, 请问二者会有什么矛盾? (简单地说, `super` 引用的并不一定是你所预期的 (静态声明的) 值, 请尝试写一个这种示例)
3. `super.xxx` 如果是属性 (而不是函数 / 方法), 那么绑定 `this` 有什么用呢?

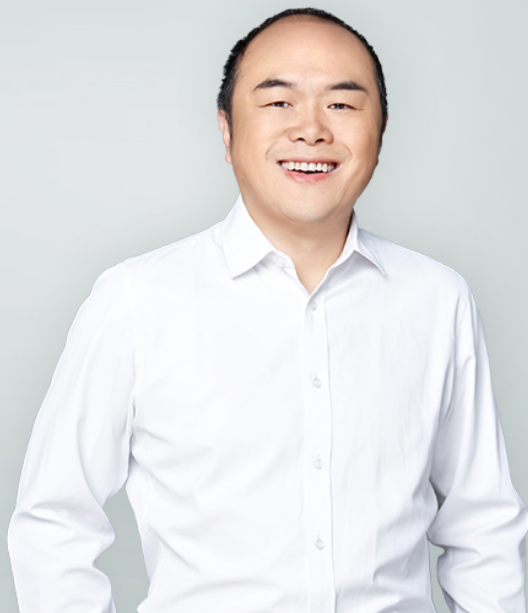
希望你能将自己的答案分享出来, 让我也有机会听听你的收获。

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 13 | new X：从构造器到类，为你揭密对象构造的全程

下一篇 15 | return Object.create(new.target.prototype)：做框架设计的基本功：写一个根类

精选留言 (3)


 写留言



行问

2019-12-16

实话实说，对 Class 和 super 的知识概念还不熟悉，有几个问题请教下老师

- 1、在“继承”上，xxx.apply() 和 xxx.call() 算是继承吗？与 super.xxx() 又有什么区别？
- 2、super.xxx() 的 this 引用是在当前环境的上下文中查找的。那么，x = super.xxx.bind...
展开 

作者回复：第一个问题，它们与super.xxx没什么关系，它们自己也不算继承。xxx.apply/xxx.call 就是普通的函数调用，而super.xxx是先查找super，然后使用当前环境中的this来调用super.xxx()。

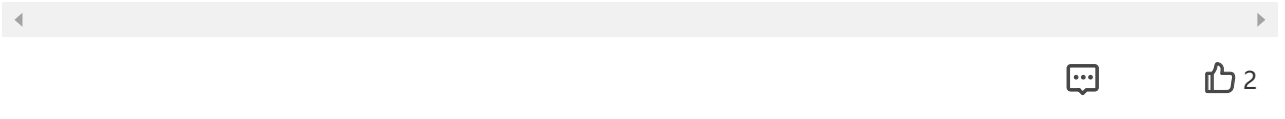
第二个问题，super()与super.xxx()其实很不相同，它们是分别独立实现的。super()相当于get_super(current_constructor).call()；而如果是在一般的、非静态声明的方法中，super.xxx()倒是与g

et_super(current_constructor).prototype.xxx.bind(current_this, ...)有些类似。——注意这两种情况下的current_constructor，是等同于当前正在声明的类的。

关于图片和思维导图这类，这次极客时间的课程里面，真的没做什么。不过仅是说今天这一讲的话，可以看看之前我讲过的《无类继承》，今天的许多内容都可以看到更详细的介绍。在这里：<https://v.qq.com/x/page/d0719xls8eb.html>

或者看搜狐的，还有PPT：

http://www.sohu.com/a/258358348_609503

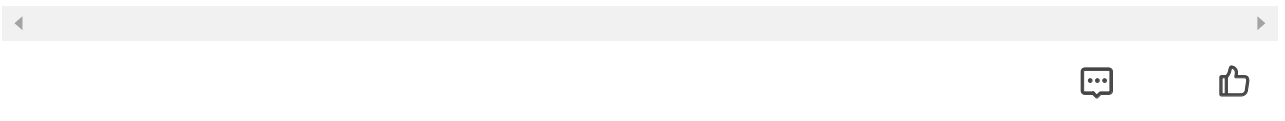


ttttt

2019-12-18

老师能不能详细贴一下每讲对应的 ecma 规范地址呢？对照着看会更好

作者回复: 这个倒是挺难的，因为每一节讲的内容都会涉及很多个ECMAScript的点，要是指着看，反倒不见得清楚，没有上下文还可能会更乱。而且主要是多，整理起来着实不易。



许童童

2019-12-16

老师讲得很好，我要再消化消化

展开 ▾

