

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》

课程介绍 >



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")
```

这个方案基本上来说，就是两个解决思路的集合：使用构造器函数来处理一些“不同的东西”；使用原型继承，来从父类继承“相同的东西”。最后，new 运算符在创建对象的过程中分别处理“原型继承”和构造器函数中的“类抄写”，补齐了最后的一块木板。

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一行代码中“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。既然子类和父类都可能是，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，似乎是没有什​​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，理论上来说应该是在一个 foo() 方法内使用的、类似 super.xxx() 这样的代码。

 复制代码

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

这样，在调用这个 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（其实，也就是 obj）。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“‘规范类型中的’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。

所谓函数上下文，之前略讲过一点，就是函数在调用的时候创建的那个用于调度执行的东西，而这个 thisValue 值就放在它的环境记录里面，也就可以理解成函数执行环境的一部分。

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：



```
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 `consturctor()` 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？这个过程中的 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。

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 2  提建议 

学习推荐

JVM + NIO + Spring

各大厂面试题及知识点详解

限时免费



精选留言 (22)

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(...) 绑定的 this 是从父类 or 祖先类“继承”而来, 这与 constructor() 中直接调用 super() 是否一致?

另, 大师可以适当添加一些代码 or 图片 or 思维导图, 在阅读理解上可以帮助我们更好理清, 感谢!

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get_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



👍 3



蛋黄酱

2020-03-28

二刷了, 来回又看了三遍左右才理解老师表达的是什么。特来评论一下。

老师的内容真的很好, 但是我强烈觉得老师的语言表达需要提高, 很多地方其实本身并不复杂, 但是因为老师的表述具有强烈的迷惑性, 绕了一个大弯, 把人给绕晕了, 最后并不知道你在讲什么。

实际上你看楼上@穿秋裤的男孩 的总结就及其简洁好懂。

作者回复: 多谢。确实是很绕的。:(

注意到你已经读到14讲了, 但其实正确的顺序应该是读到第6讲就看加餐1~2, 而读到第11讲就读加餐3。这个我问问编辑能不能调一下顺序, 至少他们现在这么排列, 不利于读者学习~~

你现在正好应该把三个加餐一并读了。所以.....不妨试试看?

共 2 条评论 >

👍 2



小毛

2020-03-20

老师, 请教个问题, 为什么ECMAScript 6中的class定义里方式是prototype原型方法, 而属性又是实例属性呢, 这样有点诡异

```
class A {  
  x=1  
}  
  
class B extends A {  
  constructor() {  
    super();  
    this.y = super.x + 1;  
  }  
}
```



```
}  
}  
let b = new B;  
console.log(b.x);//1  
console.log(b.y);//NAN
```

作者回复: ES6的class声明并不支持一般属性（使用数据描述符的属性），而只支持存取器属性。所以你的例子，其实是在较高版本的ECMAScript规范中才支持的。

而ES6不支持在类声明中使用一般属性，是因为那个时候ECMAScript规范在有关“类继承设计”方面并没有做完，是个半吊子设计。所以有关super.xxx这样的属性存取，在操作父类方法时是没问题的，但在操作父类属性时，就会出BUG。这个是ECMAScript规范中已知的，并且在test262中已经认可的。但——就目前而言——这个Bug还不会被修复。因为“在类声明中使用属性”的相关提案一直悬而未决。

回到你的例子，问题是 super.x 访问正好撞上了上面的bug。并且关于这个bug和相关的“类声明中的属性”目前没有定论，所以我也不能告诉你有效的处理方法。简单地说，尽量不要在类声明中使用属性声明，也不要试图通过super.xxx来访问父类的属性（但可以访问方法和使用读写器的属性）。



👍 2



油菜

2020-11-17

“在 JavaScript 中，super 只能在方法中使用”

老师，我的测试问题在哪呢？

```
function F1(){  
  this.a1='aaa';  
  this.b1='bbb';  
  this.f1=function(){  
    console.log('f1f1')  
  }  
}  
  
class S1 extends F1{  
  k1(){  
    super.f1(); //调用父类了f1()方法  
  }  
}
```



```
new S1().f1()); // f1f1
```

```
new S1().k1()); // Uncaught TypeError: (intermediate value).f1 is not a function
```

作者回复: 被super.xxx访问的方法必须是原型方法, 而不能是实例方法。例如:

...

```
F1.prototype.f1 = function() {  
  console.log('HI')  
}
```

// 例1: 通过this.f1()访问到了实例方法

```
(new S1).f1(); // f1f1
```

// 例2: 通过super访问到了原型方法

```
(new S1).k1(); // HI
```

...

共 2 条评论 >

👍 1



海绵薇薇

2020-09-19

老师好, 我理解的基于类构造对象的过程如下:

类的构造过程是在继承的顶层通过new.target拿到最底层的构造函数,

然后拿到继承连最底层构造函数的原型属性, 然后顶层的构造函数使用这个原型属性构造一个对象,

经过中间一层一层的构造函数的加工返回出来。

所以对象是从父类返回的, 在constructor中如果有继承需要先调用super才能拿到this

这个过程几乎描述了整个基于类构造对象的经过, 但关于类实例化对象的过程, 我还有以下疑问:

类有一个相对较新的语法如下:

...

```
class A {  
  c = function () {}  
  b = this.c.bind(this)  
}
```

...



这个 `=` 是一个表达式用于创建实例的属性，是需要执行来计算属性值的，那么这个 `b = this.c.bind(this)` 的执行环境是什么？？？

合理的猜想是constructor 中super调用之后，因为这个环境中存在this。

但是：

```
...  
class A {  
  constructor () {  
    var e = 1  
  }  
  
  c = e  
}  
...
```

这样实例化的时候会报错，如果 `c = e` 的环境在constructor中应该不会报错

除了在constructor中执行 `c = e` 找不到肉眼可见的作用域可以执行这个表达式了

探索过程如下：

```
class E {  
  constructor () {  
    console.log(222) // 2  
  }  
}  
  
class E1 extends E {  
  constructor () {  
    console.log(111) // 1  
    super()  
    console.log(this.c) // 4  
    var e = 1  
    console.log(555) // 5  
  }  
}
```




```
    c = (function() {  
        console.log(333) // 3  
        return 'this.c'  
    })()  
}
```

c = 这个表达式是在super()之后调用的，但却不是在constructor中调用的，

感觉这其中隐藏了什么，望老师指点。

感谢老师的时间

还有一个非本专栏的问题想问下，为什么 = 是表达式 / 运算？？？

javascript语编3中描述 = 是运算，并不能理解 = 是运算（表达式）的说法（虽然确实如此，因为

它可以出现在表达式才能出现的地方，例如(a = 1)不会报错）。

一直以为 = 是语句，赋值语句，作用是将右边的值赋值给左边的引用，并不做运算，虽然有返回值，

但我理解这只是 = 的副作用。

当然符号是运算的说法也不能说服自己，例如 + - 是运算所以 = 符号也是运算这有点接受不了，

毕竟javascript语编3中还介绍了和return相对的yield还是表达式呢，所以没理由=不能是语句，

= 不是语句的原因应该是它本身就不是语句和它是符号的形式没关系。

另：java中也可以打印出 a = 1的结果。 Number a = 0; System.out.println(a = 1);

再次感谢老师的时间



作者回复: 关于类成员的一些新的规范, 应该都是在讨论之中, 目前尚没有定论, 关于这个部分我在《JavaScript语言精髓与编程实践 (第三版)》中有专门讨论过。但是因为目前没有写进规范的, 所以我也不知道按哪种设计原则/原理来讲。(这涉及到对“类/对象声明”作用域的理解和设计, 这尚未成规范)

关于第二个问题, “=”是有二义性的。在表达式的概念集合中, 它确实是运算符; 而在语句的概念集合中 (例如“var x = 1”), 它只是语法符号。对于后者, 也就是语句来说, “var x”是声明, 其中“x”称为“(被绑定的)标识符”; 而“= 1”是作为单独一个语法组件来解释的, 称为“初始绑定 (初始器)”——而并不是“赋值”。

关于后面这一点, 可以参考:

<https://tc39.es/ecma262/#prod-VariableDeclaration>



恐怖如斯

2022-01-24

之前看关于super的题感觉好乱好难背容易忘, 看了老师的课后, 理解了如何实现的原理反而清晰易懂, 赞



云

2021-12-13

好绕, 难懂。得静下心来多刷, 然后看评论。

老师很厉害。



zy

2021-03-24

最近看到一段代码:

```
{  
  function foo(){}  
  foo = 1  
  foo = 2  
  foo = 3  
  foo = 4
```



```
foo = 5
function foo(){}
foo = 6
foo = 7
foo = 8
foo = 9
foo = 10
}
```

console.log(foo)

打印出来是5，分析了半天没搞明白为什么，能不能请老师帮忙分析一下

作者回复: 这段时间正好是在跟另一位同学聊相关的例子，于是写了一篇文章专门来讨论这个。在这里：

<https://blog.csdn.net/aimingoo/article/details/115270358>

简单地说，就是块语句中的函数声明会存在两个隐式的提升操作，因此会导致全局中一个同名的变量得到块作用域中的、该函数名的当前值。

共 2 条评论 >



刘大夫

2021-03-11

不得不说，super 这节讲的真好，其实也不一定非得逐字逐句把周老说的抠懂，毕竟每个人的表达方式都不一样。再加上周老多少年的浸淫，有些含义不是几句话就能说清楚的，我的收获是，

1、super 当做函数使用，只能在子类的 constructor 中调用，ES6规范规定继承时必须先跑一次super();

2、super 当做对象使用，如果是在类的一般方法(原型方法)里，其“主对象”就是这个类的原型，若 Child extends Parent，其主对象就是 Parent，那么 super 指向 Parent.prototype；如果在类的静态方法里使用，主对象就是这个类 Child 本身，super 指向 主对象的原型即 Parent；

如果在对象字面量声明的方法里使用，道理和上面一样，可以自己写个方法试试

```
class Parent {
  constructor() {
    // console.log(new.target);
  }
  a() {
    console.log('Parent a');
  }
  b () {
    console.log('Parent b')
```



```
}
static b() {
  console.log('Parent static b');
}
}

class Child extends Parent {
  constructor() {
    super();
  }
  a() {
    console.log(super.prototype);
    // super.b();
  }
  static a() {
    console.log(super.prototype.constructor);
    // super.a();
  }
  static b() {
    super.b();
  }
}

const child = new Child();
child.a();
// Child.b();
Child.a();
```



HoSalt

2020-05-25

「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」
「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」
老师，前面一段话说可能从两个地方取「Super 引用中的 thisValue 值」是指通过bind方法绑定的值？

作者回复: 第一句话是“从两个地方取”这个意思。——按顺序来确实如此。

但是由于Super引用中的thisValue总是存在，所以事实上只可能取到一个。



「Super 引用中的 thisValue 值」是特殊处理的，不是一般意义上（例如Function.prototype.bind）的绑定。



HoSalt

2020-05-25

「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」

老师，理论上可以通过HomeObject.constructor 拿到 MyClass，是因为HomeObject.constructor拿到的值不靠谱，所以不这么去拿？

作者回复: 的确是。constructor和constructor.prototype是外部原型链，不可依赖。



穿秋裤的男孩

2020-01-07

读后感

1. super的出现是为了解决子类同名属性覆盖父类的属性，通过super可以直接访问到父类。
2. 伪代码：key[[HomeObject]].__proto__ === super，不知道对不对

作者回复: 是的。👍



穿秋裤的男孩

2020-01-07

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

eg:

```
class Parent { getName() { return this } };
class Son extends Parent { getName() { return super.getName() } };
const s = new Son();
s.getName(); // 会正常打印Son类。
```

疑问：现在的js引擎是不是会自动加一个默认的constructor函数，并且执行super(),不然按照老师的说法，这边在用super之前没有super(),是不能访问super.getName()的吧？



作者回复: 是的。确实会“自动加一个”，这在下一讲里面有讲的。^^.



穿秋裤的男孩

2020-01-07

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。

eg:

```
class Parent {
  getName() { return 'Parent prototype name' }
};
class Son extends Parent {
  getName() {
    console.log(super.getName === Son.prototype.__proto__.getName);
    console.log(super.getName === Son.prototype.getName);
  }
}
const s = new Son();
s.getName(); // true; false;
```

问：从打印的结果看，如果是普通形式的声明，那么方法内部的super应该是指向Son.prototype.__proto__对象，而不是Son.prototype。

我感觉这个super就是当前对象的__proto__，即伪代码：obj.__proto__ === obj.super



穿秋裤的男孩

2020-01-07

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

前提：我理解您这边的说的主对象就是指super

例子：

```
class Parent { static obj = { name: 'parent static obj name' } }
class Son extends Parent {
  static obj = { name: 'son static obj name' }
  static getObj() { return super.obj === Parent.obj } // static声明的方法
}
Son.getObj(); // true
```



问：按照您说的话，static声明的方法，super应该指像Son，那么Son.obj应该是指向自己的static obj,也就不应该出现super.obj === Parent.obj为true的结论。这边是不是应该是：super指向Son的__proto__,也就是Parent本身？

作者回复：“主对象就是指super”。——不是的。

主对象（HomeObject）是每一个静态方法和原型方法（以及原型方法中的构造方法）作为函数对象时的一个内部槽。如果设“方法为f”，那么主对象就是“f.[HomeObject]”。于是，

```
...  
  
# 对于静态方法  
> f.[HomeObject] === MyClass // true  
  
# 对于原型方法  
> f.[HomeObject] === MyClass.prototype // true  
  
# 构造方法也是原型方法，即MyClass.prototype.constructor。因此  
> constructor.[HomeObject] === MyClass.prototype // true  
  
# 在类声明中，“构造方法”与“类”是同一个函数  
> MyClass === constructor // true  
> MyClass.[HomeObject] === MyClass.prototype // true  
...
```

接下来，super是什么呢？“super是主对象的原型”。因此：

```
...  
  
# 在原型方法（MyClass.prototype.f）中的super  
> Object.getPrototypeOf(f.[HomeObject]) === Object.getPrototypeOf(MyClass.prototype)  
  
# （亦即是说，在f()中使用super.x）  
> super.x === Object.getPrototypeOf(MyClass.prototype).x // true  
...
```

类似如上的，在静态方法f()中使用super.x

```
...  
  
# 静态方法MyClass.f中使用super.x  
> Object.getPrototypeOf(f.[HomeObject]) === Object.getPrototypeOf(MyClass); // true  
  
> super.x === Object.getPrototypeOf(MyClass).x // true  
...
```



**Elmer**

2020-01-06

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。

这句话没懂。。

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，是 MyClass.prototype.constructor? 指向的是 MyClass ?

该构造器的原型？请问这里是怎么指向 Object.prototype.constructor 的。

作者回复: 如下示例:

```
```
```

```
> class MyObject {}
```

```
> class MyObjectEx extends MyObject {}
```

```
> MyObjectEx.prototype.constructor === MyObjectEx; // true
```

```
> Object.getPrototypeOf(MyObjectEx) === MyObject; // true
```

```
```
```

**海绵薇薇**

2019-12-29

Hello 老师好:

感悟:

```
constructor() {  
  super()  
}
```

为什么不能写成

```
constructor() {  
  super.constructor()  
}
```

这种形式呢？之后过了好一会转念一想 super.constructor 是需要 this 的但是上面 super 调用之前是拿不到 this 的。

问题1:



a.b()方法中this是动态获取的，是作为this传入的a。

super.b 中的this是super.b这个引用的thisValue（执行环境的this），引用的thisValue优先级高于作为this传入的super。

通过测试发现bind绑定的this优先级高于super.a这个引用的thisValue。

但是bind，call，apply这些方法绑定的this是存在哪里的呢？bind的mixin底层也是调用的apply。

作者回复: ^^.

感悟中这种转念一想特别好。有恍然大悟的感觉，然后记得特别清楚，而且很多以前不理解的东西一下子就顺了，问题就像自然而然地全都解决了一般。

关于bind。它是单独创建了一个新的对象，用一个内部槽来放着绑定的this。类似于：

...

```
// x = func.bind(thisValue, ...args)
```

```
x.[[BoundTargetFunction]] = func
```

```
x.[[BoundArguments]] = args
```

```
x.[[BoundThis]] = thisValue
```

...

...

参见：<https://www.ecma-international.org/ecma-262/9.0/#sec-boundfunctioncreate>

至于call/apply，倒不麻烦，因为内部处理一个函数调用的时候，是将thisValue和args分别传入的（也就是隐式地传入一个参数），所以在调用时直接替换一下就好了，不需要暂存，也不通过引用来传递。



小童

2019-12-27

<https://github.com/aimingoo/prepack-core> 我克隆下来了 我不理解在如何引擎级别打断点？ 不知道怎么搞 看到一个scripts文件夹里面有 test262-runner.js文件 然后运行了脚本没运行成功。

作者回复: 看README.md，先把repl运行起来（这是运行在node环境中的），是一个类似node的控制台，可以在上面写代码试着玩。

如果你要调试，那么就让node运行在调试模式下。这个你查node的使用手册，会有命令行和端口，然后可以在chrome里打开指定的地址上的调试器环境。——这种情况下，由于nodejs把prepack-core运行在一个调试环境中，所以你就可以用断点跟进去看prepack-core里的代码如何运行了。这个时



候的操作环境在chrome里面。

你可以直接在chrome集成的调试器里面给nodejs里面的prepack-core项目打断点，也可以写debugger这个语句来在代码中触发。

总之，你先弄明白如何用chrome/firefox来调用nodejs中的项目吧.....这个网上有各种教程的。然后，你就可以调试prepack-core了。



小童

2019-12-27

我在浏览器中输出的方法中没有看到 [[HomeObject]]插槽，老师这个能在哪里找到吗？

作者回复：绝大多数内部槽在浏览器控制台上都看不到。

好象也没有别的办法看得到。

如果你有兴趣，可以自己run一个引擎，然后在引擎级别打调试断点来看。我很推荐你试试这个：

> <https://github.com/aimingoo/prepack-core>

由于它的源代码也是.js的，很易读。并且是按ECMAScript逐行写的，所以对于理解ECMAScript很有帮忙。



晓小东

2019-12-25

问题1： 如果super.xxx.bind(obj)() xxx执行上下文的thisValue域 将会被改变为obj， 而调用super.yyy()则按正常处理逻辑进行。（我这里只是测试了下执行效果， 老师可否用引擎执行层面的术语帮我们解答一下）

问题2： super 是静态绑定的，也就是说super引用跟我们书写代码位置有关。super引用的thisValue是动态运行，是从执行环境抄写过来的，所以当我调用一个从别处声明的方法是，其super代表其他地方声明对象的父类。（不知道这样表述的正不正确）代码如下：

```
let test = {
  a() {
    this.name = super.name;
  }
}
Object.setPrototypeOf(test, {name: 'test proto'})
```

```
class B extends A {
```



```

    constructor() {
        super();
        this.a = test.a;
        this.a();
        console.log(this.name); // test proto
    }
}

```

问题3: `super.xxx` 如果是属性, 也就代表对属性的进行存取两个操作 当`super.xxx` 进行取值时 `super` 所代表的的是父类的对象, `super.xxx`(规范引用: 父类的属性引用) 所以可以正常取到父类的属性值 (值类型数据或引用类型据)

但如果向`super.xxx`置值时, 此时会发生在当前`this`身上 (而不是父类对象身上)。分析: 应当是当`super.xxxx`当做左操作数时 (规范引用), 会从其引用`thisValue`取值, 属性存取发生在`thisValue`所在引用类型的数据数据身上, 而该引用正是当前被抄写过来的`this`

总结: `super.xxx` 取值时拿到的是父类的属性值

`super.xxx` 置值时会作用在当前`this`, 也就是实例身上

这样可以防止父类对象在子类实例化中被篡改

有个问题老师, 对于规范引用是否都有一个`thisValue`域, 是只有`SuperReference`有吗, 其作用是什么?

作者回复: 问题1>

`super.xxx()`中, 左侧的`super.xxx`是一个“引用 (规范类型)”, 而`thisValue`是在调用中被绑定到这个引用上的, 它相当于`(super.xxx)()`。接下来, 如果是`super.xxx.bind(obj)()`, 这时它相当于`(x=super.xxx.bind(obj))()`, 而这个`obj`是绑定在`x`这个`BindingFunction`上的, 在JavaScript中, `BindingFunction`有一个私有槽来存放这个`obj`。所以最后当作一个普通函数来执行`x()`的时候, 才会有一个`obj`可以取出来并作为`this`。

问题2>

不是。

`super`所依赖的`HomeObject`是静态绑定的。所以:

```

```

obj = {
 foo() {
 super.xxx
 }
}
```

```

这段代码中, `obj.foo.[[HomeObject]]`是静态绑定的, 恒为即`HomeObject === obj`。但是, ``super``是依赖这个`[[HomeObject]]`动态计算的, `super === Object.getPrototypeOf(HomeObject)`。



所以，它也就等同于Object.getPrototypeOf(obj) 。但你知道，obj的原型可以重置，所以：

```
...
```

```
Object.setPrototypeOf(obj, x = new Object);
```

```
super === Object.getPrototypeOf(HomeObject); // 这个结果也就变掉了
```

```
...
```

然而thisValue不是这样处理的，thisValue总是取当前执行时的上下文中的this。所以：

```
...
```

```
f = obj.foo;
```

```
obj2 = { f };
```

```
obj2.f() // <-- 现在foo()里面拿到的thisValue将是obj2
```

```
obj.foo() // <-- 现在foo()里面拿到的thisValue将是obj
```

```
...
```

问题3>

你的理解是对的。并且也确实应该这样。但是在ECMAScript中，super.xxx的取值实现得是“正确的”，逻辑和语义上都没问题；而super.xxx的置值是实现得不正确的，是个BUG设计，所以我才说“super是一个半吊子设计”。它根本上就是没实现完整，因此关于super.xxx，我绕过了它作为“super属性”的部分，一直尽量避开没有讲，而只是讲了“super.xxx()方法”的部分。

ECMAScript在这上面是个坑，并且一直没修。我接受他们没有修这个地方的原因，但结果很不爽。^
^.

原因是在class中声明属性成员，以及声明属性的可见性等问题还没有被设计出来。而现在的TC39又因为这个东西设计不出来，搞了一个class fields来替代它，真的很脑残！

（吐槽完了）

问题4>

只有SuperReference有。作用就是暂存一下当前上下文中的this。——因为从语法形式上来说，super.xxx()调用中xxx()方法应该得到的this是super，而实际上不是，所以需要新加一个域来暂存。

再换而言之，在obj.foo()中，JavaScript引擎其实是帮你隐式地传了一个参数this进去对吧？那么在super.xxx()中，它需要隐式的传两个参数，就是super和thisValue，那么就需要多添加一个域。仅此而已。

