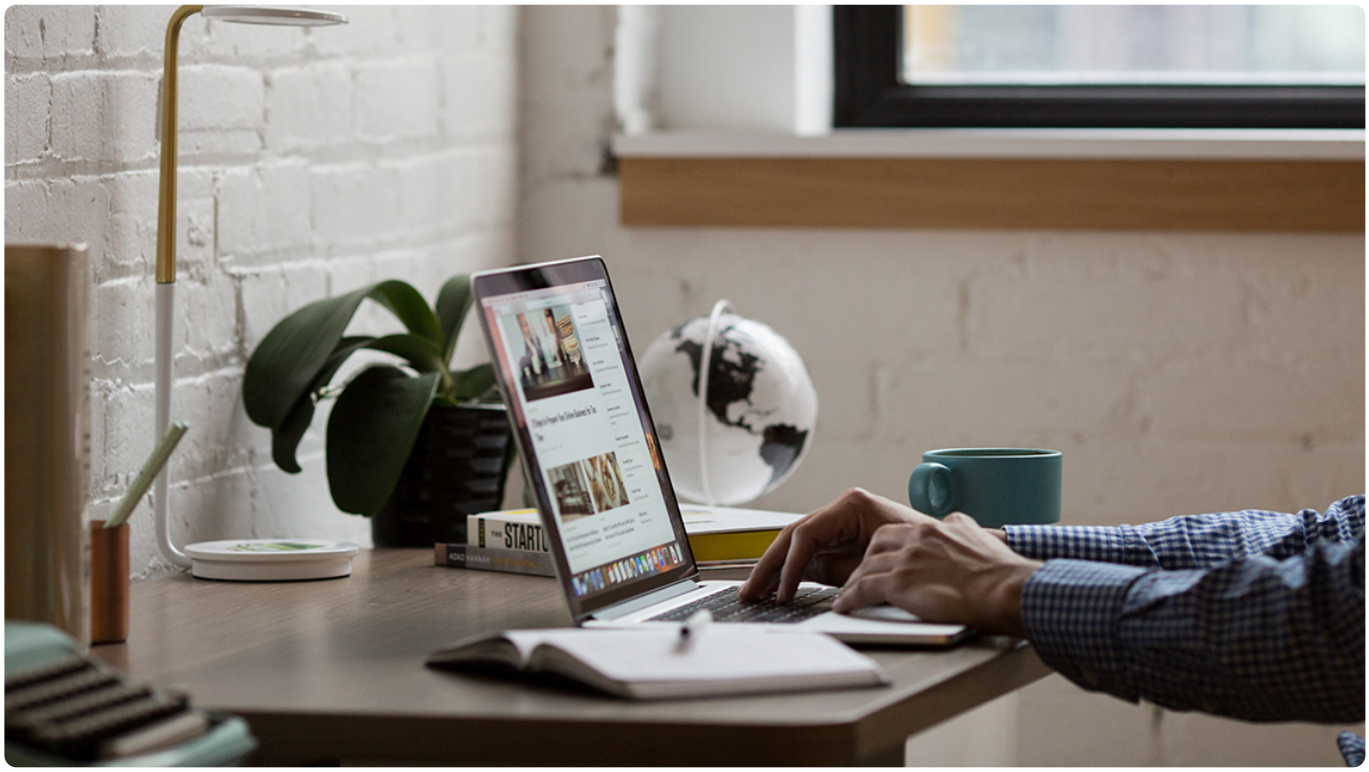
15 | 重剑无锋，大巧不工：JavaScript面向对象

2019-10-14 四火

全栈工程师修炼指南 进入课程 >



讲述：四火
时长 21:10　大小 14.55M

你好，我是四火。

JavaScript 的设计和编程能力可以说是前端工程师的修养之一，而 JavaScript 面向对象就是其中的一个重要组成部分。

我相信对于后端开发来说，面向对象的编程能力是一个程序员必须要熟练掌握的基本技能；而对于前端开发，很多项目，甚至在很多知名互联网公司的项目中，很遗憾，这部分都是缺失的，于是我们看到大量的一个一个散落的方法，以及一堆一堆难以理解的全局变量，这对系统的扩展和维护简直是噩梦。

“好的软件质量是设计出来的”，这个设计既包括宏观的架构和组件设计，也包括微观的代码层面的设计。在这一讲中，我们将学习 JavaScript 面向对象的基本知识和技巧，提升代

码层面的面向对象设计和编码能力。

首先，我们将通过面向对象的三大特征，结合实例，介绍 JavaScript 面向对象的知识：封装、继承以及多态。

1. 封装

在面向对象编程中，封装（Encapsulation）说的是一种通过接口抽象将具体实现包装并隐藏起来的方法。具体来说，封装的机制包括两大部分：

限制对对象内部组件直接访问的机制；

将数据和方法绑定起来，对外提供方法，从而改变对象状态的机制。

在 Java 中，在类中通过 private 或 public 这样的修饰符，能够实现对对象属性或方法不同级别的访问权限控制。但是，在 JavaScript 中并没有这样的关键字，但是，通过一点小的技巧，就能让 JavaScript 代码支持封装。

直到 ES6（ECMAScript 6）以前，类（class）这个概念在 JavaScript 中其实不存在，但是 JavaScript 对函数（function）有着比一般静态语言强大得多的支持，我们经常利用它来模拟类的概念。现在，请你打开 Chrome 的开发者工具，在控制台上贴上如下代码：

复制代码

```
function Book(name) {
    this.name = name;
}
console.log(new Book("Life").name);
```

你将看到控制台输出了 “Life” 。从代码中可以看到，name 作为了 Book 这个类的构造函数传入，并赋值给了自己的 name 属性（它和入参 name 重名，但却不是同一个东西）。这样，在使用“Life”作为入参来实例化 Book 对象的时候，就能访问对象的 name 属性并输出了。

但是，这样的 name 属性，其实相当于公有属性，因为外部可以访问到，那么，我们能够实现私有属性吗？当然，请看这段代码 ：

复制代码

```
function Book(name) {
    this.getName = () => {
        return name;
    };
    this.setName = (newName) => {
        name = newName;
    };
}
let book = new Book("Life");
book.setName("Time");
console.log(book.getName()); // Time
console.log(book.name); // 无法访问私有属性 name 的值
```

上面的代码中，有两处变化，一个是使用了 () => {} 这样的语法代替了 function 关键字，使得其定义看起来更加简洁，但是表达的含义依然是函数定义，没有区别；第二个是增加了 getName() 和 setName() 这样的存取方法，并且利用闭包的特性，将 name 封装在 Book 类的对象中，你无法通过任何其它方法访问到私有属性 name 的值。

这里介绍闭包（Closure），我想你应该听说过这个概念。**闭包简单说，就是引用了自由变量的函数。这里的关键是“自由变量”，其实这个自由变量，扮演的作用是为这个函数调用提供了一个“上下文”**，而上下文的不同，将对入参相同的函数调用造成不同的影响，它包括：

函数的行为不同，即函数调用改变其上下文中的其它变量，如例子中的 setName()；

函数的返回值不同，如例子中的 getName()。

和闭包相对的，是一种称为“纯函数”（Pure Function）的东西，即函数不允许引用任何自由变量。因此，和上面两条“影响”对应，纯函数的调用必须满足如下特性：

函数的调用不允许改变其所属的上下文；

相同入参的函数调用一定能得到相同的返回值。

读到这里，你是否想到了 [第 04 讲] 中我们将 HTTP 的请求从两个维度进行划分，即是否幂等，是否安全；在 [第 08 讲] 中我们对 CQRS 依然从这样两个维度进行划分，并作了分析。今天，我们还做相同的划分。

闭包的调用是不安全的，因为它可能改变对象的内部属性（闭包的上下文）；同时它也不是幂等的，因为一次调用和多次调用可能产生不同的结果。

纯函数的调用是安全的，也是幂等的。

于是，我们又一次发现，技术是相通，是可以联想和类比的。**本质上，它们围绕的都是一个方法（函数）是否引用和改变外部状态的问题。**闭包本身是一个很简单的机制，但是，它可以带来丰富的语言高级功能特性，比如高阶函数。

2. 继承

在面向对象编程中，继承（Inheritance）指的是一个对象或者类能够自动保持另一个对象或者类的实现的一种机制。我们经常讲的子类具备父类的所有特性，只是继承中的一种，叫做类继承；其实还有另一种，对象继承，这种继承只需要对象，不需要类。

在 ES6 以前，没有继承（extends）关键字，JavaScript 最常见的继承方式叫做**原型链继承**。原型（prototype）是 JavaScript 函数的一个内置属性，指向另外的一个对象，而那个对象的所有属性和方法，都会被这个函数的所有实例自动继承。

因此，当我们对那个原型指向的对象做出任何改变，这个函数的所有实例也将发生相同的改变。这样原型的设计在常见的静态语言中并不常见。当然，它在实现的效果上和静态语言中的“类属性 / 类方法”有一点儿相似。

复制代码

```
function Base(name) {
    this.name = name;
}
function Child(name) {
    this.name = name;
}
Child.prototype = new Base();

var c = new Child("Life");
console.log(c.name); // "Life"
console.log(c instanceof Base); // true
console.log(c instanceof Child); // true
```

请看上面的例子，通过将子类 Child 的原型 prototype 设置为父类的对象，就完成了 Child 继承 Base 的关联，之后我们再判断 Child 的对象 c，就发现它也是 Base 的对象。请注意这样两个要点：

设置 prototype 的语句一定要放到 Base 和 Child 两个构造器之外；

并且要放在实例化任何子类之前。

上面这两条原则非常重要，缺一不可。如果违背第一个要点，即把 prototype 的设置放到子类的里面，变成这样：

复制代码

```
function Child(name) {
    Child.prototype = new Base();
    this.name = name;
}
```

这是完全错误的，每次 Child 在构建的过程中，原型被破坏并重建一次，这可不只是一个资源浪费、状态丢失的问题。由于原型是实例辨识运算 instanceof 的依据，因此它还会影响 JavaScript 引擎对 instanceof 的判断：

复制代码

```
var c = new Child("Life");
console.log(c instanceof Base); // false
console.log(c instanceof Child); // false
```

你看，c 现在不但不是 Base 的实例，甚至也不是 Child 的了。

还有些程序员违反了上面说的第二个要点，即搞错了顺序：

复制代码

```
var c = new Child("Life");
Child.prototype = new Base();
```

后面的判断也出现了错误：

复制代码

```
console.log(c instanceof Base); // false
console.log(c instanceof Child); // false
```

因为 Child 的原型在 c 生成之后发生了破坏并重建，因此无论 Base 还是 Child，都已经和 c 没有关联了。

你再仔细想想的话，你还会发现原型链继承有一个解决不了的问题，即父类的构造方法如果包含参数，就无法被完美地继承下来。比如上例中的 name 构造参数，传入后赋值给对象的操作不得不在子类中重做了一遍。于是，我们引出另一种常见的 JavaScript 实现继承的方式——**构造继承**。

复制代码

```
function Base1(name) {
    this.name = name;
}
function Base2(type) {
    this.type = type;
}
function Child(name, type) {
    Base1.call(this, name); // 让 this 去调用 Base1，并传入参数 name
    Base2.call(this, type);
}

var c = new Child("Life", "book");
console.log(c.name); // "Life"
console.log(c instanceof Base1); // false
console.log(c instanceof Child); // true
```

你看，这种方法就能够保留父类对于构造器参数的处理逻辑，并且，我们居然还不知不觉地实现了**多重继承**！但是，缺点也很明显，使用 instanceof 方法判断的时候，发现子类对象 c 并非父类实例，并且，当父类的 prototype 还有额外属性和方法的时候，它们也无法通过构造继承被自动搬到子类里来。

3. 多态

在面向对象编程中，多态（Polymorphism）指的是同样的接口，有着不同的实现。在 JavaScript 中没有用来表示接口的关键字，但是通过在不同实现类中定义同名的方法，我们可以轻易做到多态的效果，即同名方法在不同的类中有不同的实现。而由于没有类型和参数的强约束，它的灵活性远大于 Java 等静态语言。

理解对象创建

在对面向对象的三大特征有了一定的理解之后，我们再来看看实际的对象创建。你可能会说，对象创建不是一件很简单的事儿吗，有什么可讲的？

别急，JavaScript 和一般的静态语言在对象创建上有着明显的不同，JavaScript 奇怪的行为特别多，还是让我们来看看吧。

在 Java 等多数静态语言中，是使用 new 关键字加基于类名的方法调用来创建对象，但是如果不使用 new 关键字，只使用基于类名的方法调用，则什么都不是，编译器直接报错。但是 JavaScript 不同，我们对于类的概念完全是通过强大的函数特性来实现的，先看下面这个容易混淆函数调用和对象创建的例子：

复制代码

```
function Book(name) {
    this.name = name;
    return this;
}
console.log(new Book("Life").name); // 输出 Life
console.log(Book("Life").name); // 也输出 Life
```

你看，在 Book() 中，我们最终返回了 this，这就让它变得模糊，这个 Book() 到底是类的定义，还是普通函数（方法）定义？

> 代码中使用 this 关键字来给对象自己赋值，看起来 Book 应该是类，那么 Book() 其实就是类的构造器，而这个赋值是完成对象创建的一部分；
>
> 可是它居然又有返回（return 语句），那么从这个角度看，Book 应该是普通函数定义，函数调用显式返回了一个对象。

于是，我们从上述最下面的两行代码中看到，无论使用 new 来创建对象，还是不使用 new，把它当成普通方法调用，都能够获得对象 name 属性的值“Life”，因此看起来用不用 new 似乎没有区别嘛?

其实不然，没有区别只是一个假象。JavaScript 是一个特别善于创造错觉的编程语言，有许多古怪无比“坑”等着你去踩，而这只是其中一个。我们要来进一步理解它，就必须去理解代码中的 this，众所周知 this 可以看做是对象对于它自己的引用，那么我们在执行上述两步操作时，this 分别是什么呢?

复制代码

```
function Book(name) {
    console.log(this);
    this.name = name;
    return this;
}
new Book("Life"); // 打印 Book {}
Book("Life"); // 打印 Window { ... }
window.Book("Life") // 打印 Window { ... }
```

在这段代码中，我在 Book() 内部把 this 打印出来了。原来，在使用 new 的时候，this 是创建的对象自己；而在不使用 new 的时候，this 是浏览器的内置对象 window，并且，这个效果和使用 window 调用 Book() 是一样的。也就是说，**当我们定义了一个“没有归属”的全局函数的时候，这个函数的默认宿主就是 window。**

实际上，上述例子在使用 new 这个关键字的时候，JavaScript 引擎就帮我们做了这样几件事情。

第一件，创建一个 Book 的对象，我们把它叫做 x 吧。
第二件，绑定原型：x.**proto** = Book.prototype。
第三件，指定对象自己：this = x，并调用构造方法，相当于执行了 x.Book()。
第四件，对于构造器中的 return 语句，根据 typeof x === ‘object’ 的结果来决定它实际的返回：

如果 return 语句返回基本数据类型（如 string、boolean 等），这种情况 typeof x 就不是“object”，那么 new 的时候构造器的返回会被强制指定为 x；

如果 return 语句返回其它类型，即对象类型，这种情况 typeof x 就是“object”，那么 new 的时候会遵循构造器的实际 return 语句来返回。

前面三件其实很好理解，我们的试验代码也验证了；但是第四件，简直令人崩溃对不对？这是什么鬼设计，**难道创建对象的时候，还要根据这个 return 值的类型来决定 new 的行为?**

很遗憾，说对了……我们来执行下面的代码：

复制代码

```
function Book1(name) {
    this.name = name;
    return 1;
}
console.log(new Book1("Life")); // 打印 Book1 {name: "Life"}

function Book2(name) {
    this.name = name;
    return [];
}
console.log(new Book2("Life")); // 打印 []
```

你看，Book1 的构造器返回一个基本数据类型的数值 1，new 返回的就是 Book1 的实例对象本身；而 Book2 的构造器返回一个非基本数值类型 []（数组），new 返回的就是这个数组了。

正是因为这样那样的问题，ES5 开始提供了严格模式（Strict Mode），可以让代码对一些可能造成不良后果的不严谨、有歧义的用法报错。

在实际项目中，我们应当开启严格模式，或是使用 TypeScript 这样的 JavaScript 超集等等替代方案。写 JavaScript 代码的时候，心中要非常明确自己使用 function 的目的，是创建一个类，是创建某个对象的方法，还是创建一个普通的函数，并且在命名的时候，根据项目的约定给予清晰明确的名字，看到名字就立即可以知道它是什么，而不需要联系上下文去推导，甚至猜测。

正确的代码是写给机器看的，但是优秀的代码是写给别的程序员看的。

总结思考

今天我们学习了 JavaScript 面向对象的实现方式和相关的重要特性，希望你能够掌握介绍到的知识点，通过思考和吸收，最终可以在项目中写出易于维护的高质量代码。现在，我想提两个问题，请你挑战一下：

> 在你经历的项目中，是否使用过面向对象来进行 JavaScript 编码，项目的代码质量是怎样的？
>
> 和静态语言不同的是，JavaScript 有好多种不同的方式来实现继承效果，除了文中介绍的原型链继承和构造继承以外，你是否还知道其它的 JavaScript 继承实现方式？

好，今天的内容就到这里。欢迎你在留言区和我讨论，也欢迎你把文章分享出去，和朋友一起阅读。

选修课堂：当函数成为一等公民

众所周知，有一种经典的学习一门新语言的方法是类比法，比如从 C 迁入 JavaScript 的程序员，就会不由自主地比较这两门语言的语法映射，从而快速掌握新语言的写法。

但是，**仅仅通过语法映射的学习而训练出来的程序员，只是能写出符合 JavaScript 语法的 C 语言而已，本质上写的代码依然是 C。**因此，在类比以外，我们还要思考和使用 JavaScript 不一样的核心特性，比如接下去要介绍的函数“一等公民”地位。

首先，我们需要理解，何为“函数成为一等公民”。这指的是，**函数可以不依附于任何类或对象等实体而独立存在，它可以单独作为参数、变量或返回值在程序中传递。**

回想 Java 语言，如果 Book 这个类，有一个方法 getName()，这个方法必须依附于 Book 而存在，一般情况下必须使用 Book 或它的对象才能调用。这就是说，Java 中的函数或方法，无法成为一等公民。可 JavaScript 完全不同了，你可能还记得上文中出现了这样的调用：

复制代码

```
Base1.call(this, name);
```

Base1 实际是一个函数，而函数的宿主对象 this 被当作参数传进去了，后面的 name 则是调用参数，这种以函数为核心的方法调用，在许多传统的静态语言中是很难见到的。我们来看一个更完整的例子：

复制代码

```
function getName() {
    return this.name;
}
function Book(name) {
    this.name = name;
}

let book = new Book("Life");
console.log(getName.call(book, getName)); // "Life"
```

你看，同样使用 function 关键字，getName 是函数（方法），Book 是书这个类，实例化得到 book 以后，通过 call 关键字调用，把 book 作为 getName() 的宿主，即其中的 this 传入，得到了我们期望的值“Life”。

上面就是对于函数成为一等公民的一个简单诠释：以往我们只能先指定宿主对象，再来调用函数；现在可以反过来，先指定函数，再来选择宿主对象，完成调用。请注意，函数的调用必须要有宿主对象，如果你使用 null 或者 undefined 这样不存在的对象，window 会取而代之，被指定为默认的宿主对象。

扩展阅读

对于系统地学习 ES 6，推荐阅读阮一峰的翻译作品 ECMAScript 6 入门。

文中介绍了严格模式（Strict Mode），感兴趣的话可以看看 MDN 的介绍。

文章多次提到了静态语言和动态语言，我曾经写过一篇文章编程范型：工具的选择，对它们做了介绍，供你参考。

对于文中提到的 instanceof 运算符，如果你想了解它是怎样实现的，它和对象原型有何关系，请参阅 JavaScript instanceof 运算符深入剖析。

上一篇　14 | 别有洞天：从后端到前端

精选留言 (2)

写留言

许童童
2019-10-14

老师你好，问个问题，对象继承是什么，有相关的资料可以看一下吗？

joker
2019-10-14

面试有问原型链的，老师您能给讲一下不

展开