

08 | 深入理解继承、Delegation和组合

2022-10-06 石川 来自北京

天下无鱼
<https://shikey.com/>

《JavaScript进阶实战课》

课程介绍 >



讲述：石川

时长 09:15 大小 8.44M

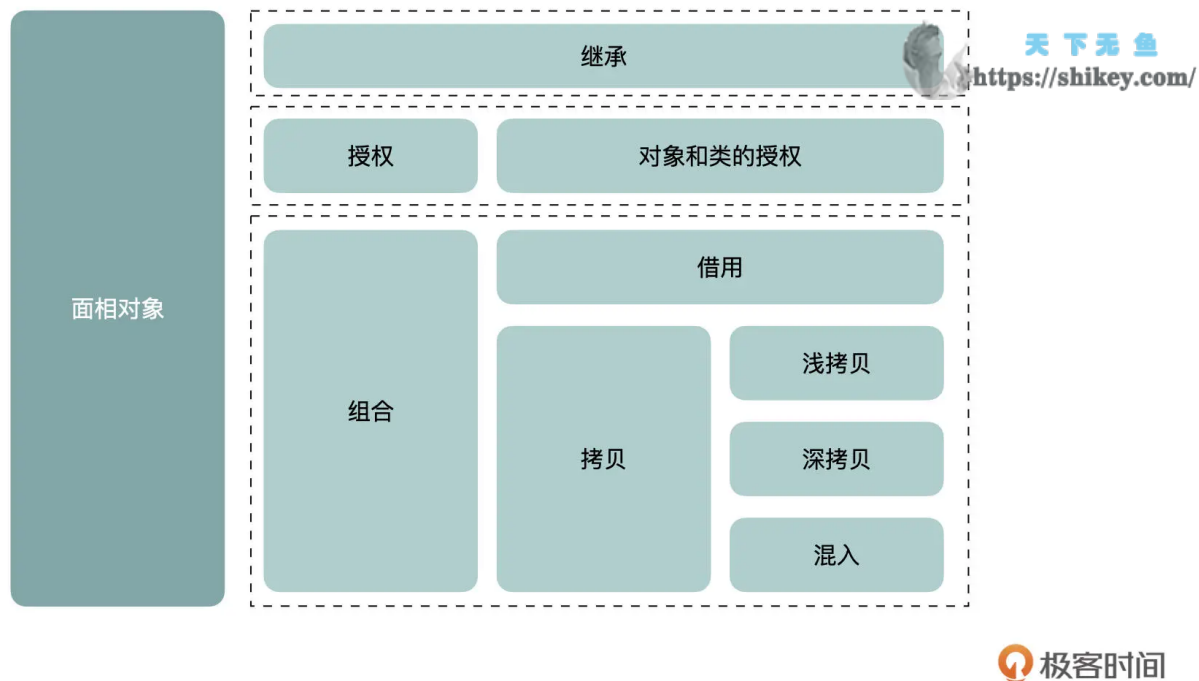


你好，我是石川。

关于面向对象编程，最著名的一本书就数 [GoF](#)（Gang of Four）写的 [《设计模式：可复用面向对象软件的基础》](#) 了。这本书里一共提供了 23 种不同的设计模式，不过今天我们不会去展开了解这些细节，而是会把重点放在其中一个面向对象的核心思想上，也就是**组合优于继承**。

在 JS 圈，有不少继承和组合的争论。其实无论是继承还是组合，我们都不能忘了要批判性地思考。批判性思考的核心不是批判，而是通过深度思考核心问题，让我们对事物能有自己的判断。

所以，无论是继承还是组合，都只是方式、方法，它们要解决的核心问题就是**如何让代码更容易复用**。



那么接下来，我们就根据这个思路，看看 JavaScript 中是通过哪些方法来解决代码复用这个问题的，以及在使用不同的方法时它们各自解决了什么问题、又引起了什么问题。这样我们在实际的业务场景中，就知道如何判断和选择最适合的解决方式了。

继承

在传统的 OOP 里面，我们通常会提到继承（Inheritance）和多态（Polymorphism）。继承是用来在父类的基础上创建一个子类，来继承父类的属性和方法。多态则允许我们在子类里面调用父类的构建者，并且覆盖父类里的方法。

那么下面，我们就先来看下在 JavaScript 里，要如何通过构造函数来做继承。

如何通过继承多态重用？

实际上，从 ES6 开始，我们就可以通过 `extends` 的方式来做继承。具体如下所示：

```
1 class Widget {
2   appName = "核心微件";
3   getName () {
4     return this.appName;
5   }
6 }
7
```

复制代码

```

8 class Calendar extends Widget {}
9
10 var calendar = new Calendar();
11 console.log(calendar.hasOwnProperty("appName")); // 返回 true
12 console.log(calendar.getName()); // 返回 "核心微件"
13
14 calendar.appName = "日历应用"
15 console.log(typeof calendar.getName()); // 返回 function
16 console.log(calendar.getName()); // 返回 "日历应用"

```



接着来看多态。从 ES6 开始，我们可以通过 `super` 在子类构建者里面调用父类的构建者，并且覆盖父类里的属性。可以看到在下面的例子里，我们是通过 `super` 将 `Calendar` 的 `appName` 属性从“核心微件”改成了“日历应用”。

复制代码

```

1 class Widget {
2   constructor() {
3     this.appName = "核心微件";
4   }
5
6   getName () {
7     return this.appName;
8   }
9 }
10
11 class Calendar extends Widget {
12   constructor(){
13     super();
14     this.appName = "日历应用";
15   }
16 }
17
18 var calendar = new Calendar();
19 console.log(calendar.hasOwnProperty("appName")); // 返回 true
20 console.log(calendar.getName()); // 返回 "日历应用"
21 console.log(typeof calendar.getName()); // 返回 function
22 console.log(calendar.getName()); // 返回 "日历应用"

```

在一些实际的例子，如 `React` 这样的三方库里，我们也经常可以看到一些继承的例子，比如我们可以通过继承 `React.Component` 来创建一个 `WelcomeMessage` 的子类。

复制代码

```

1 class WelcomeMessage extends React.Component {
2   render() {

```

```
3     return <h1>Hello, {this.props.name}</h1>;
4   }
5 }
```



天下无鱼

<https://shikey.com/>

授权

说完了继承，我们再来看授权这个方法。

什么是授权（Delegation）呢？我打个比方，这里的授权不是我们理解的作为领导（父类）给下属（子类）授权，而是作为个体对象可以授权给一个平台或者他人来一起做一件事。

就好像我和极客时间合作，我的个人精力和专业能力只允许我尽量做好内容，但是我没有精力和经验去做编辑、后期和推广等等，这时就授权给极客时间相关的老师来一起做，我在这件事、这个过程中只是专心把内容的部分做好。

如何通过授权做到重用？

在前面的例子中，结合我们在 [第 1 讲](#) 里提到的基于原型链的继承，我们会发现使用 JavaScript 无论是通过函数构建也好，还是加了语法糖的类也好，来模拟一般的面向对象语言，比如 Java 的类和继承，对于有些开发者来说是比较**反直觉**的。在使用的时候需要大量的思想转换，才能把 JavaScript 的底层逻辑转换成实际呈现出来的实现。

那么有没有一种方式可以让代码更直观呢？这种方式其实就是**通过原型本身来做授权会更符合直觉**。从 ES5 开始，JavaScript 就支持了 `Object.create()` 的方法。下面我们来看一个例子：

复制代码


```
1 var Widget = {
2   setCity : function(City) {this.city = City; },
3   outputCity : function() {return this.city;}
4 };
5
6 var Weather = Object.create(Widget);
7
8 Weather.setWeather = function (City, Tempreature) {
9   this.setCity(City);
10  this.tempreature = Tempreature;
11 };
12
13 Weather.outputWeather = function() {
14   console.log(this.outputCity()+ ", " + this.tempreature);
15 }
16
```

```
17 var weatherApp1 = Object.create(Weather);
18 var weatherApp2 = Object.create(Weather);
19
20 weatherApp1.setWeather("北京","26度");
21 weatherApp2.setWeather("南京","28度");
22
23 weatherApp1.outputWeather(); // 北京, 26度
24 weatherApp2.outputWeather(); // 南京, 28度
```



可见，我们创建的 **Weather** 天气预报这个对象，授权给了 **Widget**，让 **Widget** 在得到授权的情况下，帮助 **Weather** 来设定城市和返回城市。**Widget** 对象在这里更像是一个平台，它在得到 **Weather** 的授权后为 **Weather** 赋能。而 **Weather** 对象可以在这个基础上，专注于实现自己的属性和方法，并且产出 **weatherApp1** 和 **weatherApp2** 的实例。

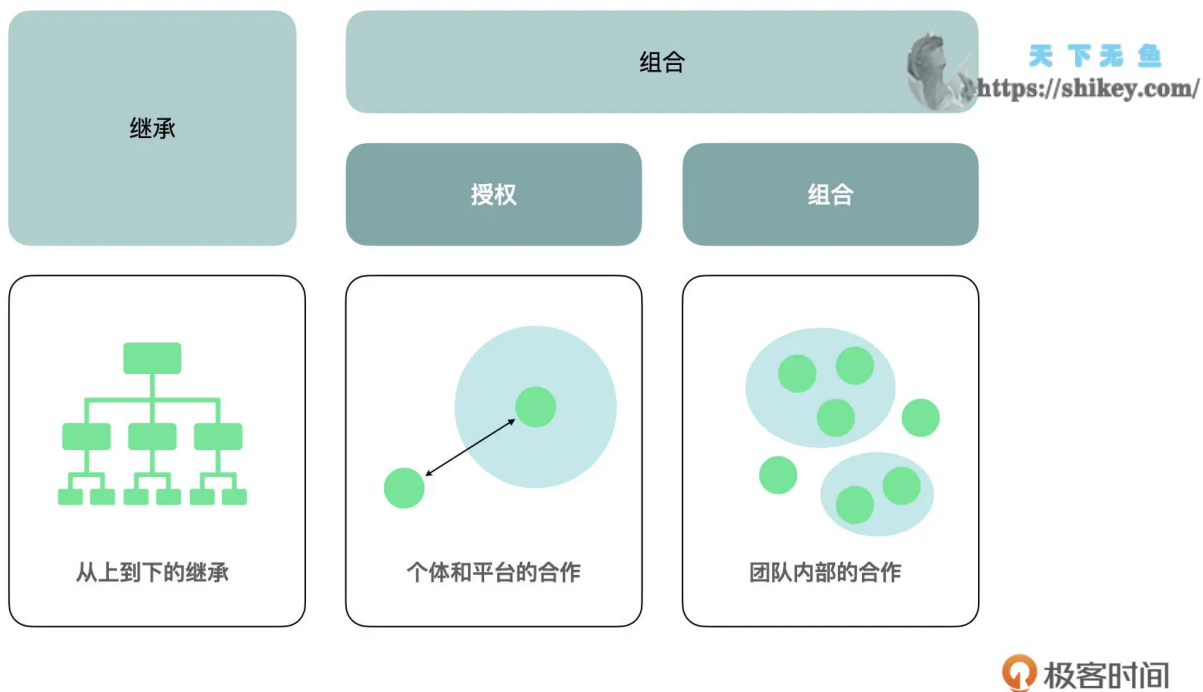
当然也有开发者认为 **class** 的方式没有什么反直觉的，那授权同样可以通过 **class** 来实现。比如我们如果想在上一讲提到过的集合（**Set**）和字典（**Map**）的基础上，加上计数的功能，可以通过继承 **Set** 来实现。但是我们也可以反之，在把部分功能授权给 **Map** 的基础上，自己专注实现一些类似 **Set** 的 **API** 接口。

 复制代码

```
1 class SetLikeMap {
2     // 初始化字典
3     constructor() { this.map = new Map(); }
4     // 自定义集合接口
5     count(key) { /*...*/ }
6     add(key) { /*...*/ }
7     delete(key) { /*...*/ }
8     // 迭代返回字典中的键
9     [Symbol.iterator]() { return this.map.keys(); }
10    // 部分功能授权给字典
11    keys() { return this.map.keys(); }
12    values() { return this.map.values(); }
13    entries() { return this.map.entries(); }
14 }
```

组合

说完了授权，我们再来看看组合。当然上面我们说的授权，广义上其实就是一种组合。但是这种组合更像是“个体和平台的合作”；而另一种组合更像是“团队内部的合作”，它也有很多的应用和实现方式，我们可以来了解一下。



如何通过借用做到重用？

在 JavaScript 中，函数有自带的 `apply` 和 `call` 功能。我们可以通过 `apply` 或 `call` 来“借用”一个功能。这种方式，也叫**隐性混入**（Implicit mixin）。比如在数组中，有一个原生的 `slice` 的方法，我们就可以通过 `call` 来借用这个原生方法。

如下代码示例，我们就是通过借用这个功能，把函数的实参当做数组来 `slice`。

```
1 function argumentSlice() {
2     var args = [].slice.call(arguments, 1, 3);
3     return args;
4 }
5 // example
6 argumentSlice(1, 2, 3, 4, 5, 6); // returns [2,3]
```

复制代码

如何通过拷贝赋予重用？

除了“借力”以外，我们还能通过什么组合方式来替代继承呢？这就要说到“拷贝”了。这个方法顾名思义，就是把别人的属性和方法拷贝到自己的身上。这种方式也叫**显性混入**（Explicit mixin）。

在 ES6 之前，人们通常要偷偷摸摸地“抄袭”。在 ES6 之后，JavaScript 里才增加了“赋予”，也就是 `Object.assign()` 的功能，从而可以名正言顺地当做是某个对象“赋予”给另外一个对象它的“特质和能力”。



那么下面，我们就先看看在 ES6 之后，JavaScript 是如何名正言顺地来做拷贝的。

首先，通过对象自带的 `assign()`，我们可以把 `Widget` 的属性赋予 `calendar`，当然在 `calendar` 里，我们也可以保存自己本身的属性。和借用一样，借用和赋予都不会产生原型链。如以下代码所示：

复制代码

```
1 var widget = {
2   appName : "核心微件"
3 }
4
5 var calendar = Object.assign({
6   appVersion: "1.0.9"
7 }, widget);
8
9 console.log(calendar.hasOwnProperty("appName")); // 返回 true
10 console.log(calendar.appName); // 返回 “核心微件”
11 console.log(calendar.hasOwnProperty("appVersion")); // 返回 true
12 console.log(calendar.appVersion); // 返回 “1.0.9”
```

好，接着我们再来看看在 ES6 之前，人们是怎么通过“抄袭”来拷贝的。

这里实际上分为“浅度拷贝”和“深度拷贝”两个概念。“浅度拷贝”类似于上面提到的赋予 `assign` 这个方法，它所做的就是遍历父类里面的属性，然后拷贝到子类。我们可以通过 JavaScript 中专有的 `for in` 循环，来遍历对象中的属性。

细心的同学可能会发现，我们在 [第 2 讲](#)中说到用拷贝来做到不可变时，就了解过通过延展操作符来实现浅拷贝的方法了。

复制代码

```
1 // 数组浅拷贝
2 var a = [ 1, 2 ];
3 var b = [ ...a ];
4 b.push( 3 );
5 a; // [1,2]
6 b; // [1,2,3]
```

```
7 // 对象浅拷贝
8
9 var o = {
10     x: 1,
11     y: 2
12 };
13 var p = { ...o };
14 p.y = 3;
15 o.y; // 2
16 p.y; // 3
```



而在延展操作符出现之前，人们大概可以通过这样一个 `for in` 循环做到类似的浅拷贝。

复制代码

```
1 function shallowCopy(parent, child) {
2     var i;
3     child = child || {};
4     for (i in parent) {
5         if (parent.hasOwnProperty(i)) {
6             child[i] = parent[i];
7         }
8     }
9     return child;
10 }
```

至于深度拷贝，是指当一个对象里面存在嵌入的对象就会深入遍历。但这样会引起一个问题：**如果这个对象有多层嵌套的话，是每一层都要遍历吗？究竟多深算深？还有就是如果一个对象也引用了其它对象的属性，我们要不要也拷贝过来？**

所以相对于深度拷贝，浅度拷贝的问题会少一些。但是在 [第 2 讲](#) 的留言互动区，我们也说过，如果我们想要保证一个对象的深度不可变，还是需要深度拷贝的。深度拷贝的一个相对简单的实现方案是用 `JSON.stringify`。当然这个方案的前提是这个对象必须是 `JSON-safe` 的。

复制代码

```
1 function deepCopy(o) { return JSON.parse(JSON.stringify(o)); }
```

同时，在 [第 2 讲](#) 的留言区中，也有同学提到过另外一种递归的实现方式，所以我们也大致可以通过这样一个递归来实现：


```

1 function deepCopy(parent, child) {
2   var i,
3   toString = Object.prototype.toString,
4   astr = "[object Array]";
5   child = child || {};
6   for (i in parent) {
7     if (parent.hasOwnProperty(i)) {
8       if (typeof parent[i] === "object") {
9         child[i] = (toString.call(parent[i]) === astr) ? [] : {};
10        deepCopy(parent[i], child[i]);
11      } else {
12        child[i] = parent[i];
13      }
14    }
15  }
16  return child;
17 }

```



如何通过组合做到重用？

上面我们说的无论是借用、赋予，深度还是浅度拷贝，都是一对一的关系。最后我们再来看看，如何通过 ES6 当中的 `assign` 来做到组合混入，也就是说把几个对象的属性都混入在一起。其实方法很简单，以下是参考：

```

1 var touchscreen = {
2   hasTouchScreen: () => true
3 };
4
5 var button = {
6   hasButton: () => true
7 };
8 var speaker = {
9   hasSpeaker: () => true
10 };
11
12 const Phone = Object.assign({}, touchscreen, button, speaker);
13
14 console.log(
15   hasTouchScreen: ${ Phone.hasChocolate() }
16   hasButton: ${ Phone.hasCaramelSwirl() }
17   hasSpeaker: ${ Phone.hasPecans() }
18 );

```

在 **React** 当中，我们也可以看到 [组合优于继承的无处不在](https://shikekey.com/)，并且它同样体现在我们前面讲过的两个方面，一个是“团队内部的合作”，另一个是“个体与平台合作”。下面，我们先看看“团队内部的合作”的例子，在下面的例子里，**WelcomeDialog** 就是嵌入在 **FancyBorder** 中的一个团队成员。

复制代码

```
1 function FancyBorder(props) {
2   return (
3     <div className={'FancyBorder FancyBorder-' + props.color}>
4       {props.children}
5     </div>
6   );
7 }
8
9 function WelcomeDialog() {
10  return (
11    <FancyBorder color="blue">
12      <h1 className="Dialog-title">
13        Welcome
14      </h1>
15      <p className="Dialog-message">
16        Thank you for visiting our spacecraft!
17      </p>
18    </FancyBorder>
19  );
20 }
```

另外，我们也可以看到“个体与平台合作”的影子。在这里，**WelcomeDialog** 是一个“专业”的 **Dialog**，它授权给 **Dialog** 这个平台，借助平台的功能，实现自己的 **title** 和 **message**。这里就是用到了组合。

复制代码

```
1 function Dialog(props) {
2   return (
3     <FancyBorder color="blue">
4       <h1 className="Dialog-title">
5         {props.title}
6       </h1>
7       <p className="Dialog-message">
8         {props.message}
9       </p>
10    </FancyBorder>
11  );
12 }
13 function WelcomeDialog() {
```

```

14     return (
15         <Dialog
16             title="Welcome"
17             message="Thank you for visiting our spacecraft!" />
18     );
19 }

```

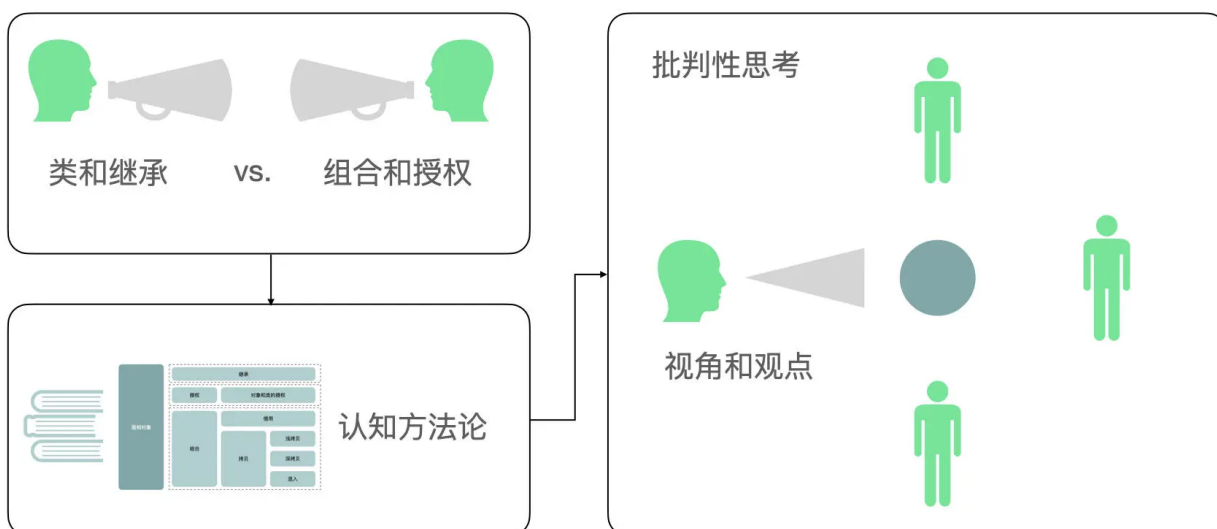


总结

这节课，我们了解了通过 JavaScript 做到代码复用的几种核心思想和方法，从传统的继承，到 JavaScript 特色的授权以及组合等方式都有分析。虽然我说授权和组合优于继承，但实际上它们之间的关系不是非黑即白的。

我们看到在前端圈，有很多大佬比如道格拉斯·克rock福德（Douglas Crockford）和凯尔·辛普森（Kyle Simpson），都是基于授权的对象创建的积极拥护者；而像阿克塞尔·劳施迈尔博士（Dr. Axel Rauschmayer）则是基于类的对象构建的捍卫者。

我们作为程序员，如果对对象和面向对象的理解不深入，可能很容易在 [不同的论战和观点](#) 面前左摇右摆。而实际的情况是，真理本来就不止一个。我们要的“真理”，只不过是通过对一个观察角度，形成的一个观点。这样，才能分析哪种方式适合我们当下要解决的问题。这个方式，只有在当下，才是“真理”。而我们通过这个单元整理的方法，目的就是帮助我们做到这样的观测。



思考题

在前面一讲中，我们试着通过去掉对象私有属性的语法糖，来看如何用更底层的语言能力来实现类似的功能。那么，今天你能尝试着实现下 JS 中的类和继承中的 `super`，以及原型和授权中的 `Object.create()` 吗？

欢迎在留言区分享你的答案、交流学习心得或者提出问题，如果觉得有收获，也欢迎你把今天的内容分享给更多的朋友。我们下节课见！

分享给需要的人，Ta 购买本课程，你将得 18 元

生成海报并分享

赞 0 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 07 | 深入理解对象的私有和静态属性

下一篇 09 | 面向对象：通过词法作用域和调用点理解 `this` 绑定

精选留言 (2)

写留言



Change

2022-10-13 来自河北

```
/* ***** JS 中的类和继承中的 super ***** */
```

```
function Widget() {  
  this.appName = "核心微件";  
}
```

```
Widget.prototype.getAppName = function () {  
  return this.appName;  
};
```

```
function Calendar() {  
  // 调用super
```

```

let widget = new Widget();
this.__proto__.__proto__ = widget.__proto__; // Calendar.prototype.__proto__ = Widget.p
rototype;
for (let key in widget) {
  if (widget.hasOwnProperty(key)) {
    this[key] = widget[key];
  }
}

this.name = "Calendar";
}

Calendar.prototype.getName = function () {
  return this.name;
};

/*****Object.create*****/
function Person() {
  this.name = "Person";
}

// Object.Create
function ObjectCreate(o) {
  let obj = {};
  obj.__proto__ = o;
  return obj;
}

let o = ObjectCreate(new Person());

```



褚琛

2022-10-10 来自海南

//js中的类和继承

```

function Widget (appName) {
  this.appName = appName
}

Widget.prototype.getName = function() {
  return this.appName;
}

```

```
function Calendar (appName) {  
  Widget.call(this, appName);  
}
```



```
Calendar.prototype = { ...Widget.prototype };
```

```
var calendar = new Calendar('日历应用');  
console.log(calendar.hasOwnProperty("appName")); // 返回 true  
console.log(calendar.getName()); // 返回 "日历应用"  
console.log(typeof calendar.getName); // 返回 function  
console.log(calendar.getName()); // 返回 "日历应用"
```

```
//Object.create()  
function create(o) {  
  let Cls = function() {};  
  let obj = new Cls();  
  obj.prototype = o;  
  return obj;  
}
```

