前端开发核心知识进阶：50 讲从夯实基础到突破瓶颈
来自 Lucas ... · 盐选专栏

查看详情 >

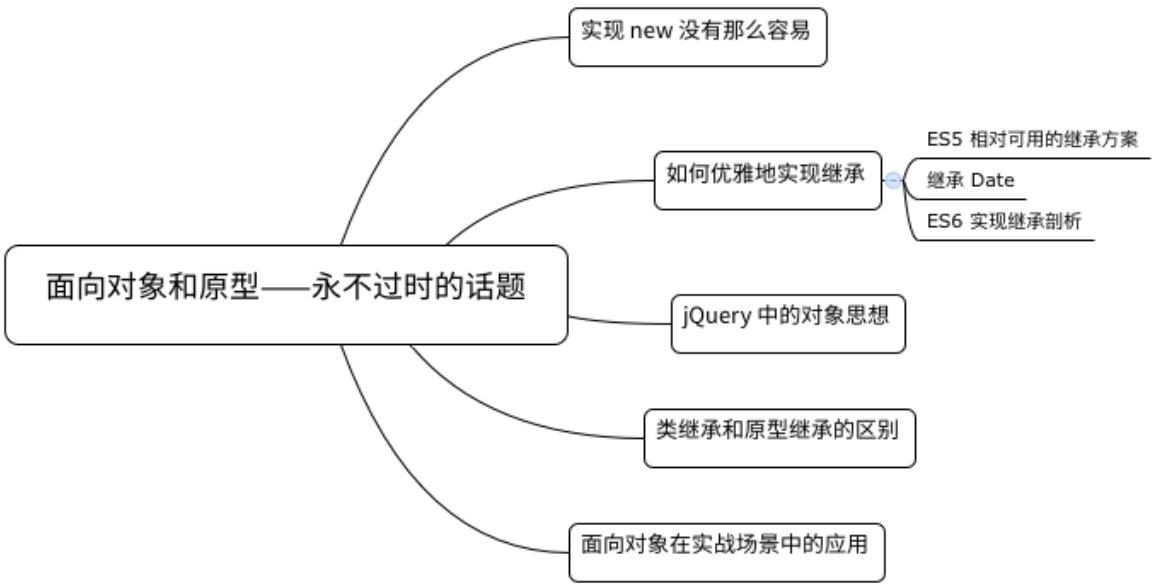
# 面向对象和原型——永不过时的话题

「对象」——这个概念在编程中非常重要，任何语言和领域的开发者都应该具有面向对象思维，能够有效运用对象。良好的面向对象系统设计将是应用强健性、可维护性和可扩展性的关键；反之，如果面向对象环节有失误，那么将是项目的灾难。

说到 JavaScript 面向对象，它实质是基于原型的对象系统，而不是基于类的。这是由设计之初所决定的，是基因层面的。随着 ES Next 标准的进化和新特性的添加，使得 JavaScript 面向对象更加贴近其他传统面向对象型语言。有幸目睹语言的发展和变迁，伴随着某个语言的成长，我认为是开发者之幸。

这一讲就让我们深入对象和原型，理解 JavaScript 在这个方向上的能力。请注意，**我们不再过多赘述基础，而是面向进阶，需要读者具有一定的知识准备。**

相关知识点如下：



## 实现 new 没有那么容易

说起 JavaScript 当中的 new 关键字，有一段很有趣的历史。其实 JavaScript 创造者 Brendan Eich 实现 new 是为了获得更高的流行度，它是强行学习 Java 的一个残留产出，他想让 JavaScript 成为 Java 的小弟。很多人认为这个设计掩盖了 JavaScript 中真正的原型继承，只是表面上看，更像是基于类的继承。

这样的误会使得很多传统 Java 开发者并不能很好理解 JavaScript。实际上，我们前端工程师应该明白，new 关键字到底做了什么事情。

> step1：首先创建一个空对象，这个对象将会作为执行 new 构造函数() 之后，返回的对象实例
>
> step2：将上面创建的空对象的原型（__proto__），指向构造函数的 prototype 属性
>
> step3：将这个空对象赋值给构造函数内部的 this，并执行构造函数逻辑
>
> step4：根据构造函数执行逻辑，返回第一步创建的对象或者构造函数的显式返回值

因为 new 是 JavaScript 的关键字，我们不能直接覆盖，实现一个 newFunc 来进行模拟，预计使用方式：

```
function Person(name) {
 this.name = name
}

const person = new newFunc(Person, 'lucas')

console.log(person)

// {name: "lucas"}
```

实现为：

```
function newFunc(...args) {
 // 取出 args 数组第一个参数，即目标构造函数
 const constructor = args.shift()

 // 创建一个空对象，且这个空对象继承构造函数的 prototype 属性
 // 即实现 obj.__proto__ === constructor.prototype
 const obj = Object.create(constructor.prototype)

 // 执行构造函数，得到构造函数返回结果
 // 注意这里我们使用 apply，将构造函数内的 this 指向为 obj
 const result = constructor.apply(obj, args)

 // 如果造函数执行后，返回结果是对象类型，就直接返回，否则返回 obj 对象
 return (typeof result === 'object' && result != null) ? result : obj
}
```

上述代码并不复杂，几个关键点：

使用 Object.create 将 obj 的 __proto__ 指向为构造函数的原型

使用 apply 方法，将构造函数内的 this 指向为 obj

在 newFunc 返回时，使用三目运算符决定返回结果

我们知道，构造函数如果有显式返回值，且返回值为对象类型，那么构造函数返回结果不再是目标实例。如下代码：

```
function Person(name) {
 this.name = name
 return {1: 1}
}

const person = new Person(Person, 'lucas')
```

```
console.log(person)

// {1: 1}
```

了解这些注意点，对于理解 newFunc 的实现就不再困难。

### 如何优雅地实现继承

实现继承式是面向对象的一个重点概念。我们前面提到过 JavaScript 的面向对象系统是基于原型的，它的继承不同于其他大多数语言。

社区上对于 JavaScript 继承讲解的资料不在少数，这里我不再赘述每一种继承方式的实现过程，还需要开发者事先进行了解。

### ES5 相对可用的继承方案

我们仅总结以下 JavaScript 中实现继承的关键点。

如果想使 Child 继承 Parent，那么

原型链实现继承最关键的要点是：

```
Child.prototype = new Parent()
```

这样的实现，不同的 Child 实例的 `__proto__` 会引用同一 Parent 的实例。

构造函数实现继承的要点是：

```
function Child (args) {
   // ...
   Parent.call(this, args)
}
```

这样的实现问题也比较大，其实只是实现了实例属性继承，Parent 原型的方法在 Child 实例中并不可用。

组合继承的实现才基本可用，其要点是：

```
function Child (args1, args2) {
   // ...
   this.args2 = args2
   Parent.call(this, args1)
}
Child.prototype = new Parent()
Child.prototype.constrcutor = Child
```

它的问题在于 Child 实例会存在 Parent 的实例属性。因为我们在 Child 构造函数中执行了 Parent 构造函数。同时，`Child.__proto__` 也会存在同样的 Parent 的实例属性，且所有 Child 实例的 `__proto__` 指向同一内存地址。

同时上述实现也都没有对静态属性的继承

还有一些其他不完美的继承方式，我们这里不再过多介绍。

一个比较完整的实现为：

```
function inherit(Child, Parent) {
    // 继承原型上的属性
   Child.prototype = Object.create(Parent.prototype)

    // 修复 constructor
   Child.prototype.constructor = Child

   // 存储超类
   Child.super = Parent

   // 静态属性继承
   if (Object.setPrototypeOf) {
       // setPrototypeOf es6
       Object.setPrototypeOf(Child, Parent)
   } else if (Child.__proto__) {
       // __proto__ es6 引入，但是部分浏览器早已支持
```

```
        Child.__proto__ = Parent
    } else {
        // 兼容 IE10 等陈旧浏览器
        // 将 Parent 上的静态属性和方法拷贝一份到 Child 上，不会覆盖 Child 上的方法
        for (var k in Parent) {
            if (Parent.hasOwnProperty(k) && !(k in Child)) {
                Child[k] = Parent[k]
            }
        }
    }

}
```

上面静态属性继承存在一个问题：在陈旧浏览器中，属性和方法的继承我们是静态拷贝的，继承完后续父类的改动不会自动同步到子类。这是不同于正常面向对象思想的。但是这种组合式继承，已经相对完美、优雅。

### 继承 Date

值得一提的一个小细节是：这种继承方式无法实现对 Date 对象的继承。我们来进行测试：

```
function DateConstructor() {
    Date.apply(this, arguments)
    this.foo = 'bar'
}

inherit(DateConstructor, Date)

DateConstructor.prototype.getMyTime = function() {
    return this.getTime()
};


let date = new DateConstructor()
```

```
console.log(date.getMyTime())
```

将会得到报错：Uncaught TypeError: this is not a Date object.

究其原因，是因为： **JavaScript 的日期对象只能通过 JavaScript Date 作为构造函数来实例化得到。**

因此 v8 引擎实现代码中就一定有所限制，如果发现调用 getTime() 方法的对象不是 Date 构造函数构造出来的实例，则抛出错误。

那么如何实现对 Date 的继承呢?

```
function DateConstructor() {
   var dateObj = new(Function.prototype.bind.apply(Date,
[Date].concat(Array.prototype.slice.call(arguments))))()

   Object.setPrototypeOf(dateObj,
DateConstructor.prototype)

   dateObj.foo = 'bar'

   return dateObj
}

Object.setPrototypeOf(DateConstructor.prototype,
Date.prototype)

DateConstructor.prototype.getMyTime = function getTime() {
   return this.getTime()
}

let date = new DateConstructor()

console.log(date.getMyTime())
```

我们来分析一下代码：调用构造函数 DateConstructor 返回的对象 dateObj 有：

```
dateObj.__proto__ === DateConstructor.prototype
```

而我们通过：

```
Object.setPrototypeOf(DateConstructor.prototype,
Date.prototype)
```

实现了：

```
DateConstructor.prototype.__proto__ === Date.prototype
```

因此连起来就是：

```
date.__proto__.__proto__ === Date.prototype
```

继续分析，DateConstructor 构造函数里，返回的 dateObj 是一个真正的 Date 对象，因为：

```
var dateObj = new(Function.prototype.bind.apply(Date,
[Date].concat(Array.prototype.slice.call(arguments))))
()var dateObj = new(Function.prototype.bind.apply(Date,
[Date].concat(Array.prototype.slice.call(arguments))))()
```

它终归还是由 Date 构造函数实例化出来，因此它有权调用 Date 原型上的方法，而不会被引擎所限制。

整个实现过程通过更改原型关系，在构造函数里调用原生构造函数 Date，并返回其实例的方法，「欺骗了」浏览器。当然这样的做法比较取巧，其副作用是更改了原型关系，这样也会干扰浏览器某些优化操作。

那么有没有更加「体面」的方式呢?

其实随着 ES6 class 的推出，我们完全可以直接使用 extends 关键字了：

```
class DateConstructor extends Date {
   constructor() {
       super()
       this.foo ='bar'
   }
   getMyTime() {
       return this.getTime()
   }
}

let date = new DateConstructor()
```

上面的方法可以完美执行：

```
date.getMyTime()
// 1558921640586
```

直接在支持 ES6 class 的浏览器中完全没有问题。可是我们项目大部分都是使用 Babel 进行编译。按照上一讲 Babel 编译 class 的方法，运行其产出后，仍然会得到报错：Uncaught TypeError: this is not a Date object.，因此我们得知：Babel 并没有对继承 Date 进行特殊处理，无法做到兼容。

### ES6 实现继承剖析

在 ES6 时代，我们可以使用 class extends 进行继承。但是我们都知道 ES6 的 class 其实也就是 ES5 原型的语法糖。我们通过研究 Babel 编译结果，来深入了解一下。

首先，我们定义一个父类：

```
class Person {
   constructor(){
       this.type = 'person'
```

```
    }
}
```

这个类包含了一个实例属性。

然后，实现一个 Student 类，这个「学生」类继承「人」类：

```
class Student extends Person {
    constructor(){
        super()
    }
}
```

从简出发，我们定义的 Person 类只包含了 type 为 person 的这一个属性，不含有方法。我们 Student 类也继承了同样的属性。

如下：

```
var student1 = new Student()
student1.type // "person"
```

我们进一步可以验证原型链上的关系：

```
student1 instanceof Student // true
student1 instanceof Person // true
student1.hasOwnProperty('type') // true
```

那么，经过 Babel 编译，我们的代码是什么样呢?

一步一步来看：

```
class Person {
    constructor(){
        this.type = 'person'
```

```
    }
}
```

被编译为：

```
var Person = function Person() {
    _classCallCheck(this, Person);
    this.type = 'person';
};
```

我们看到其实还是构造函数那一套。

```
class Student extends Person {
    constructor(){
        super()
    }
}
```

编译结果：

```
// 实现定义 Student 构造函数，它是一个自执行函数，接受父类构造函数为参数
var Student = (function(_Person) {
    // 实现对父类原型链属性的继承
    _inherits(Student, _Person);

    // 将会返回这个函数作为完整的 Student 构造函数
    function Student() {
        // 使用检测
        _classCallCheck(this, Student);
        // _get 的返回值可以先理解为父类构造函数
        _get(Object.getPrototypeOf(Student.prototype),
'constructor', this).call(this);
    }

    return Student;
```

```
})(Person);

// _x 为 Student.prototype.__proto__
// _x2 为'constructor'
// _x3 为 this
var _get = function get(_x, _x2, _x3) {
   var _again = true;
   _function: while (_again) {
       var object = _x,
           property = _x2,
           receiver = _x3;
       _again = false;
       // Student.prototype.__proto__ 为 null 的处理
       if (object === null) object = Function.prototype;
       // 以下是为了完整复制父类原型链上的属性，包括属性特性的描述符
       var desc = Object.getOwnPropertyDescriptor(object,
property);
       if (desc === undefined) {
           var parent = Object.getPrototypeOf(object);
           if (parent === null) {
               return undefined;
           } else {
               _x = parent;
               _x2 = property;
               _x3 = receiver;
               _again = true;
               desc = parent = undefined;
               continue _function;
           }
       } else if ('value' in desc) {
           return desc.value;
       } else {
           var getter = desc.get;
           if (getter === undefined) {
               return undefined;
           }
           return getter.call(receiver);
```

```
        }
    }
};

function _inherits(subClass, superClass) {
    // superClass 需要为函数类型，否则会报错
    if (typeof superClass !== 'function' && superClass !==
null) {
        throw new TypeError('Super expression must either
be null or a function, not ' + typeof superClass);
    }
    // Object.create 第二个参数是为了修复子类的 constructor
    subClass.prototype = Object.create(superClass &&
superClass.prototype, {
        constructor: {
            value: subClass,
            enumerable: false,
            writable: true,
            configurable: true
        }
    });
    // Object.setPrototypeOf 是否存在做了一个判断，否则使用
__proto__
    if (superClass) Object.setPrototypeOf ?
Object.setPrototypeOf(subClass, superClass) :
subClass.__proto__ = superClass;
}
```

我们进行拆解：

```
var Student = (function(_Person) {
    _inherits(Student, _Person);

    function Student() {
        _classCallCheck(this, Student);
        _get(Object.getPrototypeOf(Student.prototype),
'constructor', this).call(this);
```

```
    }

    return Student;
})(Person);
```

这是一个自执行函数，它接受一个参数 Person（就是它要继承的父类），返回一个构造函数 Student。

上面 _inherits 方法的本质其实就是让 Student 子类继承 Person 父类原型链上的方法。它的实现原理可以归结为一句话：

```
Student.prototype = Object.create(Person.prototype);
Object.setPrototypeOf(Student, Person)
```

是不是这就非常熟悉了。注意，Object.create 接收了第二个参数，这顺带实现了对 Student 的 constructor 修复。

以上通过 _inherits 实现了对父类原型链上属性的继承，那么对于父类的实例属性（就是 constructor 定义的属性）的继承，也可以归结为一句话：

```
Person.call(this);
```

我们看到 Babel 将 class extends 编译成了 ES5 组合模式的继承，这才是 JavaScript 面向对象的实质。

### jQuery 中的对象思想

可能会有读者有这样的问题：「所有的面试官都那么注重面向对象，可是我在工作中很少涉及到啊？面向对象到底有什么用？」

回答这个问题我想说，「如果你没有开发大型复杂项目的经验，不具备封装抽象的思想，也许确实用不到面向对象，也很难解释为什么要有面向对象的设计和考察。」这一讲，我从 jQuery 源码架构设计入手，来分析一下基本的原型和原型链知识如何在 jQuery 源码中发挥作用的。

「什么，这都哪一年了你还在说 jQuery？」

其实优秀的思想是永远不过时的，研究清楚 $ 到底是个什么，你会受益匪浅。顺带我自己的一个知乎回答：jQuery 为什么还在发布新版本？ 。

从一个问题开始：

```
const pNodes = $('p')
// 我们得到一个数组
const divNodes= $('div')
// 我们得到一个数组
```

但是我们又可以：

```
const pNodes = $('p')
pNodes.addClass('className')
```

数组上可是没有 addClass 方法的吧?

这个问题先放一边。我们想一想 $ 是什么？你的第一反应可能是一个函数，因此我们可以这么调用执行：

```
$('p')
```

但是你一定又见过这样的使用：

```
$.ajax()
```

那么 $ 又是一个对象，它有 ajax 的静态方法。

类似：

```
// 构造函数
function $() {
```

```
}

$.ajax = function () {
   // ...
}
```

实际上，我们翻看 jQuery 源码架构会发现（具体内容有删减和改动）：

```
var jQuery = (function(){
   var $

   // ...

   $ = function(selector, context) {
       return function (selector, context) {
           var dom = []
           dom.__proto__ = $.fn

           // ...

           return dom
       }
   }

   $.fn = {
       addClass: function() {
           // ...
       },
       // ...
   }

   $.ajax = function() {
       // ...
   }

   return $
})()
```

```
window.jQuery = jQuery
window.$ === undefined && (window.$ = jQuery)
```

我们顺着源码分析，当调用 `$('p')` 时，最终返回的是 dom，而 `dom.__proto__` 指向了 `$.fn`，`$.fn` 是包含了多种方法的对象集合。因此返回的结果（dom）可以在其原型链上找到 addClass 这样的方法。同理，`$('span')` 也不例外，任何实例都不例外。

```
$('span').__proto__ === $.fn
```

同时 ajax 方法直接挂载在构造函数 `$` 上，它是一个静态属性方法。

请读者仔细体会整个 jQuery 的架构，其实翻译成 ES class 就很好理解了（不完全对等）：

```
class $ {
 static ajax() {
   // ...
 }

 constructor(selector, context) {
   this.selector = selector
   this.context = context

   // ...
 }

 addClass() {
   //  ...
 }
}
```

这个应用虽然并不复杂，但是还是很微妙地表现出来了面向对象的精妙设计。

### 类继承和原型继承的区别

我们了解了 JavaScript 中的原型继承，那么它和传统面向对象语言的类继承有什么不同呢？这就涉及到编程语言范畴了，传统的面向对象语言的类继承，会引发一些问题：

紧耦合问题

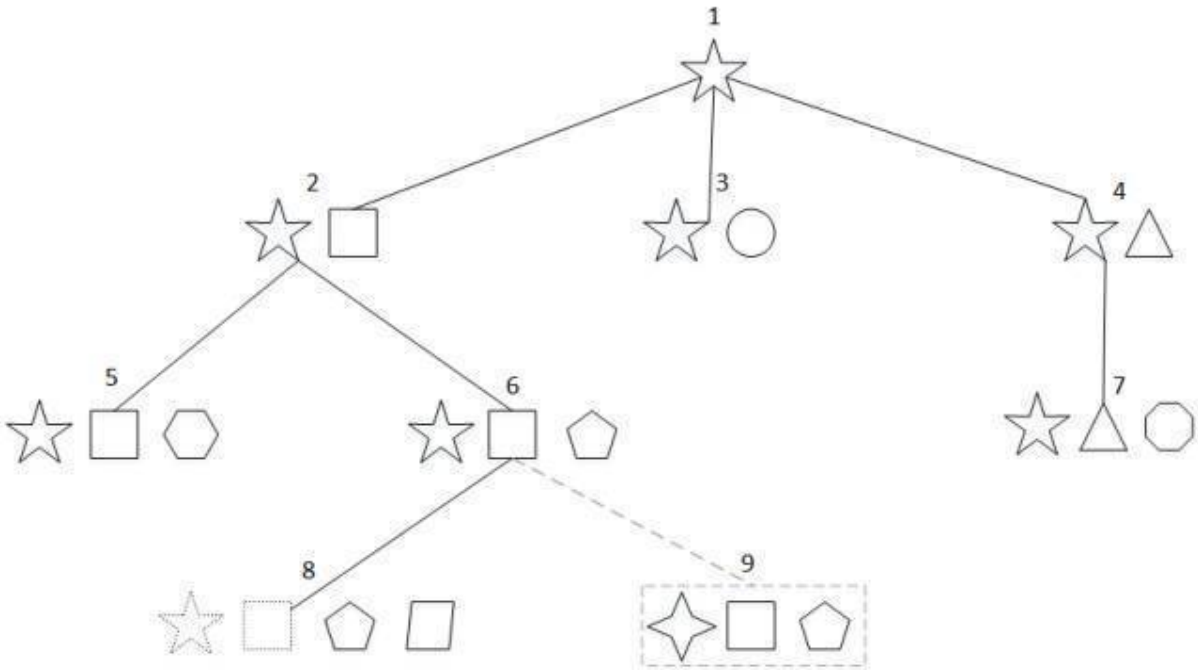
脆弱基类问题

层级僵化问题

必然重复性问题

大猩猩—香蕉问题

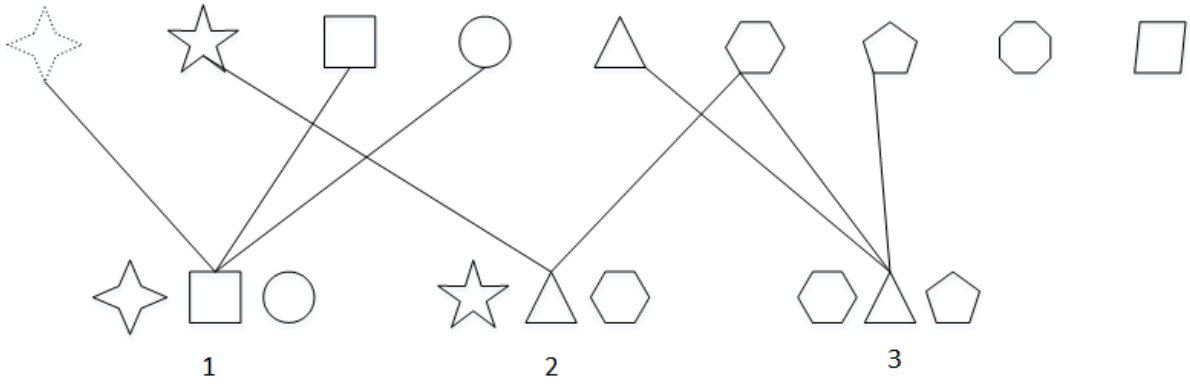
这些内容属于纯理论，多说无益。但我借用 Eric Elliott 的著名文章：Difference between class prototypal inheritance，展开一点：




从上图，我们看出一些问题（单一继承、紧耦合以及层级分类问题），对于类 8，只想继承五边形的属性，却得到了继承链上其他并不需要的属性，比如五角星，正方形属性。这就是大猩猩/香蕉问题，「我只想要一个香蕉，但是你给我了整个森林」。

对于类 9，对比其父类，我只需要把五角星属性修改成四角形，但是五角星继承自基类 1，如果要去修改，那就影响整个继承树（脆弱基类/层级僵化问题）；好吧，我不去修改，那就需要给类 9 新建一个基类（必然重复性问题）。

那么基于原型的继承可以怎么解决上述问题呢?




采用原型继承，其实本质是对象组合，可以避免复杂纵深的层级关系。当类 1 需要四角星特性的时候，只需要组合新特性即可，不会影响到其他实例。

**上述图示出自：类继承和原型继承的区别**

了解了这些，你还会吐槽 JavaScript 吗？请爱上我们的 JavaScript 吧！

### 面向对象在实战场景中的应用

最后，让我们分析一个真实场景案例。

在产品当中，一个页面可能存在多处「收藏」组件：


原创 **社交恐惧症患者如何转移对自己的过于关注？** 听语音

2017-05-14 | 0 | 0

其实这个问题问的非常好，其实对于社交恐惧者来说，不敢在公众场合说话，怕说错话是因为自己过于的关注自己，过于的在意别人是怎么看自己

收藏

步骤阅读 >


点击按钮，对页面进行收藏，成功收藏之后，按钮的状态会变为「已收藏」，再点击不会有响应。

这样就出现页面中多处「收藏」组件之间通信问题，点击页面顶部收藏按钮成功收藏之后，页面底部的收藏按钮状态也需要变化，进行同步。

其实实现这个功能很简单，但是历史代码实现方式如果落后，耦合严重就很麻烦了。良好的设计和肆意而为的实现差别是巨大的。

以 ES6 class 实现为例，不借助任何框架，我们实现这样的对象关系：所有 UI 组件（包括收藏组件）都会继承 UIBase class：

```
class Widget extends UIBase {
   constructor() {
       super();
       ...
   }
}
```

而 UIBase 本身会产生一个全局唯一的 id，这样通过继承，使得所有组件都有一个唯一的 id 标识。同时，UIBase 又继承 EventEmitter 这个 pub/sub 模式组件：

```
class UIBase extends EventEmitter{
   constructor() {
       super();
       this.guid = guid();
   }
}
```

因此，所有的组件也同样拥有了 pub/sub 模式，即事件发布订阅功能。这就相对完美的解决了组件之间的通信问题，达到了「高内聚、低耦合」的效果。

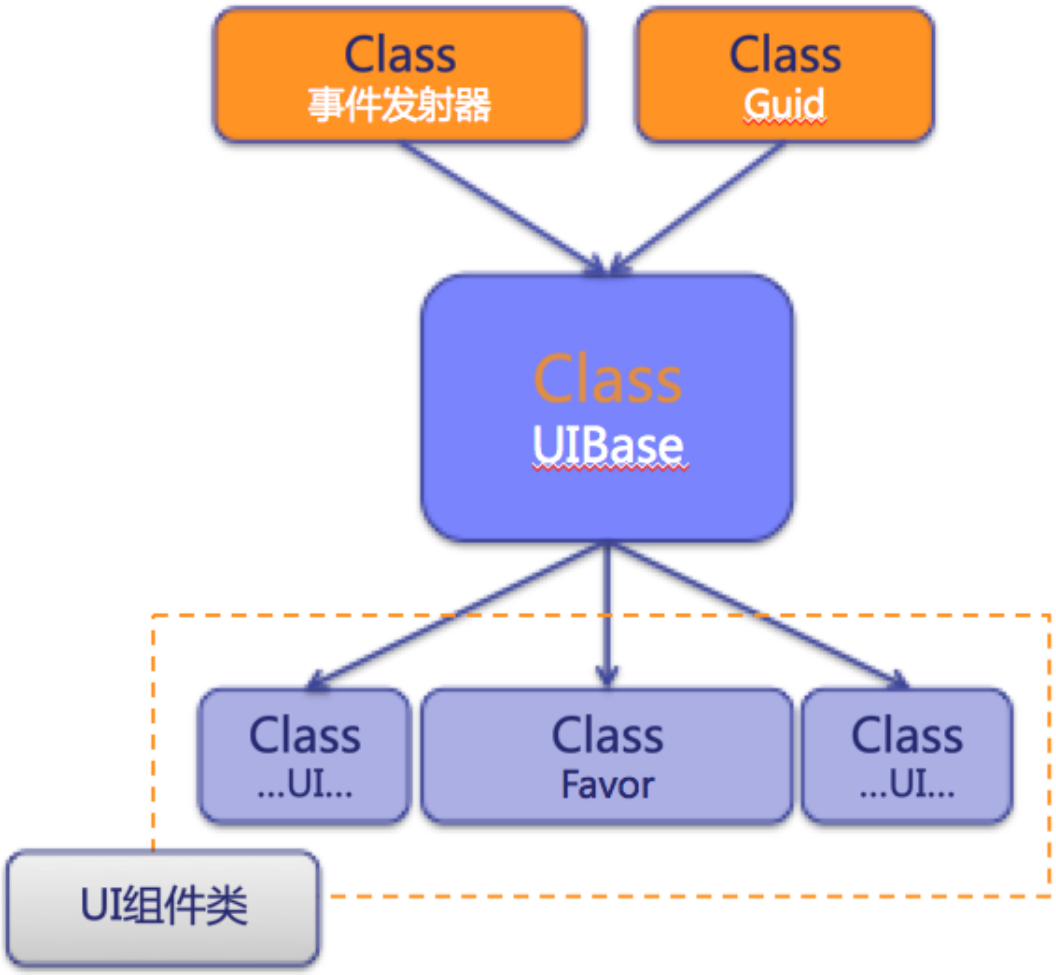
具体来说，我们的任何组件，当然包括收藏按钮在发起收藏行为时：

```
widget.emit('favorAction')
```

同时，其他的收藏组件：

```
widget.on('favorAction', function() {
   // toggle status
})
```

具体的实现结构如图：




这样的组件行为在一些先进的 MVVM、MVC 等框架中可以良好的实现，比如 React 框架中，可以借助 Redux 实现组件间的通信。Redux 实质就是一个事件发布订阅系统，而 connect 就是将组件的行为具备「发布和订阅」的能力。在上述简单的架构中，我们通过面向对象继承，自动具备了这样的能力。

同样的设计思想也可以在 NodeJS 源码中找到线索，想想 NodeJS 中的 EventEmitter 类即可。

### 总结

面向对象是一个永远说不完的话题，更是一个永远不会过时的话题，具备良好的面向对象架构能力，对于开发者来说至关重要。同时由于 JavaScript 面向对象的特殊性，使它区别于其他语言，而「与众不同」。我们在了解 JavaScript 原型、原型链知识的前提下，对比其他语言的思想，就变得非常重要和有意义了。

点击查看下一节

## 究竟该如何学习与时俱进的 ES Next