

## 16 | 百花齐放，百家争鸣：前端MVC框架

2019-10-16 四火

全栈工程师修炼指南

[进入课程 >](#)



讲述：四火

时长 17:04 大小 13.68M



你好，我是四火。

我在上一章讲到了 MVC 的原理，今天我来讲讲前端的 MVC 框架。这部分发展很快，它们比后端 MVC 框架出现得更晚，但是社区普遍更活跃。

我们在学习的过程中，需要继续保持深度和广度的均衡，既要对自己熟悉的那一款框架做深入了解，知道它的核心特性，明白其基本实现原理，对于其优劣有自己的想法；也要多了解了解这个技术的百花园，看看别的框架是什么，想想有什么优势和缺点，拓宽视野，为自己能够做出合理的技术选型而打下坚实的基础。

### 前端 MVC 的变革

让我们回想一下，在 [\[第 07 讲\]](#) 中，介绍过的 MVC 架构。实际上，我们可以把前端的部分大致归纳到视图层内，可它本身，却还可以按照 MVC 的基本思想继续划分。这个划分，有些遵循着 MVC 两个常见形式之一，有些则遵循着 MVC 的某种变体，比如 MVVM。

我们都知道前端技术的基础是 HTML、CSS 和 JavaScript，可随着技术的发展，它们在前端技术分层中的位置是不断变化的。

在前端技术发展的早期，Ajax 技术尚未被发明或引进，页面是一次性从服务端生成的，即便有视图层的解耦，页面聚合也是在服务端生成的（如果忘记了服务端的页面聚合，请回看 [\[第 09 讲\]](#)）。也就是说，整个页面一旦生成，就可以认为是静态的了。

在这种情况下，如果单独把前端代码的组织也按照 MVC 架构来划分，你觉得 HTML 到底算模型层还是视图层？

有人说，是模型层，因为它承载了具备业务含义的文本和图像等资源，是数据模型的载体，它们是前端的血和肉；

有人说，是视图层，因为它决定了用户最后看到的样子，至于 CSS，它可以决定展示的“部分”效果，但却不是必须的（即便没有 CSS，页面一样可以展示）。

其实，这两种说法都部分正确。毕竟，如果采用服务端聚合，等浏览器收到了响应报文，从前端的角度来看，模型和呈现实际已经融合在一起了，很难分得清楚。

等到 Ajax 技术成熟，客户端聚合发展起来了，情况忽然就不一样了。代表视图的 HTML 模板和代表数据的 JSON 报文，分别依次抵达浏览器，JavaScript 再把数据和模板聚合起来展示，这时候这个过程的 MVC 分层就很清晰了。

曾经 jQuery 是最流行的 JavaScript 库，但是如今随着前端业务的复杂性剧增，一个单纯的库已经不能很好地解决问题，而框架开始扮演更重要的地位，比如大家常常耳闻的前端新三驾马车 Vue.js, Anuglar 和 React。

## Angular

对于现代 MVC 框架的介绍，我将用两个框架来举例。前端框架那么多，希望你的学习不仅仅是知识的堆砌，而是可以领会一些有代表性的玩法，能有自己的解读。第一个是 Angular，我们来看看它的几个特性。


## 1. 双向绑定

曾经，写前端代码的时候，数据绑定都是用类似于 [jQuery](#) 绑定的方式来的，但是，有时候视图页面的数值变更和前端模型的数据变更，这两个变更所需的数据绑定是双向的，这就会引发非常啰嗦的状态同步：

数据对象发生变更以后，要及时更新 DOM 树；


用户操作改变 DOM 树以后，要回头更新数据对象。

比方说，在 JavaScript 中有这样一个数据对象，一本书：

 复制代码

```
1 book = {name: "Steve Jobs Biography"}
```

在 HTML 中有这样的 DOM 元素：

 复制代码


```
1 <input id="book-input" type="text" ... />
2 <label id="book-label" ...></label>
```

我们需要把数据绑定到这样的 DOM 对象上去，这样，在数据对象变更的时候，下面这两个 DOM 对象也会得到变更，从而保证一致性：

 复制代码

```
1 $("#book-input").val(book.name);
2 $("#book-label").text(book.name);
```

相应地，我们还需要些绑定语句来响应用户对 book-input 这个输入框的变更，同步到 book-label 和 JavaScript 的 book 对象上去：

 复制代码


```
1 $("#book-input").keydown(function(){
```

```
2     var data = $(this).val();
3     $("book-label").text(data);
4     book.name = data;
5 });
```

你可以想象，当这样的关联变更很多的时候，类似的样板代码该有多少，复杂度和恶心程度该有多高。

于是 Angular 跳出来说，让我们来使用双向绑定解决这个问题吧。无论我们“主动”改变模型层的业务对象（book 对象），还是视图层的这个业务对象的展示（input 标签），都可以自动完成模型层和视图层的同步。

实现方法呢，其实只有两步而已。首先模型层需要告知 DOM 受到哪个控制器控制，比如这里的 BookController，然后使用模板的方式来完成从模型到视图的绑定：

 复制代码

```
1 <div ng-app ng-controller="BookController">
2     <input type="text" value="{{book.name}}" />
3     <label>{{book.name}}</label>
4 </div>
```

接着在 JavaScript 代码中定义控制器 BookController，将业务对象 book 绑定到 \$scope 以暴露出去：

 复制代码

```
1 function BookController($scope) {
2     $scope.book = {name : "Steve Jobs Biography"};
3 }
```

你看，这样 label、input 和 \$scope.book 这三者就同步了，这三者任一改变，另两者会自动同步，保持一致。**这大大简化了复杂绑定行为的代码，尽可能地将绑定的命令式代码移除出去，而使用声明式代码来完成绑定的关联关系的定义。**

## 2. 依赖注入

你可能还记得我们在 [\[第 11 讲\]](#) 中介绍过依赖注入，在前端，借助 Angular 我们也可以做到，比如下面的例子：

 复制代码


```
1 function BookController($scope, $http) {  
2   $http.get('/books').success(function(data) {  
3     $scope.books = data;  
4   });  
5 }
```

你看，无论是 `$scope` 还是 `$http` 模块，写业务代码的程序员都不需要关心，只需要直接使用即可，它们被 Angular 管理起来并在此注入。这个方法，是不是很像我们介绍过的 Spring 对对象的管理和注入？

### 3. 过滤器

注意，这是 Angular 的过滤器，并不是我们之前讲到的 Servlet Filter。

过滤器是个很有趣的特性，让人想起了管道编程。你大概也发现 Angular 真是一个到处“抄袭”，哦不，是“借鉴”各种概念和范型的东西，比如依赖注入抄 Spring，标签定义抄 Flex，过滤器抄 Linux 的管道。从一定角度来说，还是那句话，技术都是相通的。比如：

 复制代码

```
1 {{ book.name | uppercase | replace:' ':'_' }}
```

你看，这就是把书名全转成大写，再把空格用下划线替换。我觉得这“管道”用得就很酷了。它的一大意义是，业务对象到视图对象的转换，被这样简单而清晰的方式精巧地解决了。

## React + Redux


这两个放到一起说，是因为 [React](#) 其实只是一个用户界面的库，它的组件化做得特别出色，但本身的贡献主要还是在视图层；而 [Redux](#) 是一个 JavaScript 状态容器，提供可预测



的状态管理能力。有了 Redux，才能谈整个 MVC 框架。

## 1. JSX

没有 JSX 的话 React 也能工作，但是如果没有 JSX，React 会变得索然无味许多，JSX 是 React 带来的最有变革意义的部分。比如这样一个简单的 JSX 文件：

 复制代码

```
1 class BookInfo extends React.Component {
2   render() {
3     return (
4       <div>{this.props.name}</div>
5     );
6   }
7 }
8
9 ReactDOM.render(
10   <BookInfo name="Steve Jobs Biography" />,
11   document.getElementById('book-info')
12 );
```

前半部分定义了一个输出图书信息的组件 BookInfo，内容很容易理解；后半部分则是将这个组件渲染到指定的 DOM 节点上。<div>{this.props.name}</div> 这个东西，如果你初次见到，可能会感到新奇：

看起来像是 HTML，可是居然放在 JavaScript 代码里返回了；

也没有使用双引号，因此看起来也不像是单纯的字符串。

没错，它二者都不是，而是 JavaScript 的一种语法扩展。

我们总在说解耦，于是我们把用于呈现的模板放到 HTML 里，把和模板有关的交互逻辑和数据准备放到 JavaScript 里（这被称为“标记和逻辑的分离”）。

可是越来越多的程序员发现，**这样的解耦未必总能带来“简化”，原因就在呈现模板本身，还有为了最终呈现而写的渲染逻辑，二者有着紧密的联系，脱离开模板本身的渲染逻辑，没有存在的价值，也难以被阅读和理解。**

既然如此，那为什么还要把它们分开呢？

原来，它们分开的原因并不仅仅是为了分层解耦本身，还因为当时承载技术发展的限制。还记得我们谈到过的声明式和命令式代码的区别吗？两种不同的编程范式，由于技术等种种限制，就仿佛井水不犯河水，二者采用的技术是分别发展的。

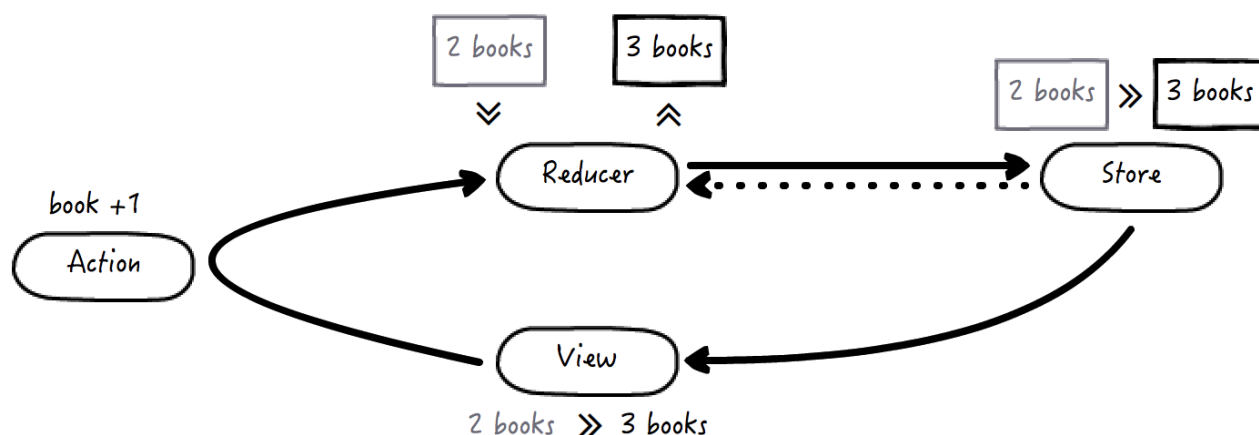
如今，大胆的 JSX 反其道而行，把呈现和渲染逻辑放在了一起，并且，它还没有丢掉二者自身的优点。比如说，整体看 JSX 内跑的是 JavaScript 代码，但是嵌入在 JSX 中的 HTML 标签依然可以以它原生的结构存放，支持 JSX 的开发工具也可以实时编译并告知 HTML 标签内的错误。换言之，JSX 中的“HTML 标签”它依然是具有结构属性的 HTML，而不是普通字符串！

并且，这两者放到一起以后，带来了除了内聚性增强以外的其它好处。比如说，测试更加方便，所有的呈现代码都可以作为 JavaScript 的一部分进行测试了，这大大简化了原本需要针对 HTML 而进行单独的打桩、替换、变化捕获而变得复杂的测试过程。

## 2. Redux 对状态的管理

复杂前端程序的一大难题是对于状态的管理，本质上这种状态的不可预知性是由前端天然的事件驱动模型造成的（如有遗忘，请回看 [\[第 14 讲\]](#)），它试图用一种统一的模式和机制来简化状态管理的复杂性，达到复杂系统状态变化“可预测”的目的。


下面我通过一个最简单的例子，结合图示，来把这个大致过程讲清楚。



首先，最核心的部分，是图中右侧是 Store，它是唯一一个存储状态的数据来源，要获知整个系统的状态，只要把握住 Store 的状态就可以了。假设一开始存放了两本书。


在图中的最下方，由 View 来展现数据，这部分我们已经很熟悉了，根据 Store 的状态，视图会展示相应的内容。一旦 Store 的状态有了更新，View 上会体现出来，这个数据绑定后的同步由框架完成。一开始，展示的是书的数量 2。

这时，用户在 View 上点击了一个添加书本的按钮，一个如下添加书本的 Action 对象生成，发送 (dispatch) 给 Reducer：

 复制代码

```
1 { type: 'ADD_BOOK', amount: 1 }
```

Reducer 根据 Action 和 Store 中老的状态，来生成新的状态。它接收两个参数，一个是当前 Store 中的状态 state，再一个就是上面的这个 action，返回新的 state：

 复制代码

```
1 (state = 0, action) => {  
2   switch (action.type) {  
3     case 'ADD_BOOK':  
4       return state + action.amount;  
5     default:  
6       return state;  
7   }  
8 };
```

于是，Store 中的 state 由 2 变成了 3，相应地，View 展示的图书数量也得到了更新。

那为什么 Redux 能将复杂的状态简化？我觉得有这么几个原因：

整个流程中**数据是单向流动的，状态被隔离，严格地管理起来了**，只有 Store 有状态，这就避免了散落的状态混乱而互相影响。

无论多么复杂的 View 上的操作或者事件，都会统一转换成若干个 Redux 系统能够识别的 Action。换句话说，**不同的操作，只不过引起 Action 的 type 不同，或者上面承载的业务数据不同。**

Reducer 是无状态的，它是一个纯函数，但它的职责是根据 Action 和 Store 中老的状态来生成新的状态。这样，**Store 中状态的改变也只有一个来源，就是 Reducer 的操**



作。

## 总结思考


今天我们学习了从前端的角度怎样理解 MVC 架构，特别学习了 Angular 和 React + Redux 两个实际框架的具有代表性的特性。

下面，留两个思考题给你：

**问题一：**你在项目中是否使用过前端 MVC 框架，你觉得它带来了什么好处和坏处？

**问题二：**案例判断。


我们曾经学过要解耦，把行为从 HTML 中分离出去，比如这样的代码：

 复制代码

```
1 <img onclick="setImage()">
```

我们说它“不好”，因为点击行为和视图展现耦合在一起了，因此我们使用 jQuery 等工具在 JavaScript 中完成绑定，才最终把它移除出去，完成了“解耦”。

可是，作为现代的 JavaScript 框架，Angular 却又让类似的代码回来了：

 复制代码

```
1 <img ng-click="setImage()">
```

对此，你怎么看，你觉得它会让代码结构和层次变得更好，还是更糟？

好，今天就到这里，欢迎你打卡，把你的总结或者思考题的答案，分享到留言区，我们一起讨论。

## 扩展阅读

和其它技术相比，[Angular 的中文站](#)做得非常出色，关于 Angular 的中文教程到上面去找就好了。

对于 React 的学习，[官方的中文翻译文档](#)是非常适合的起点；对于 Redux 的学习，请参考 [Redux 中文文档](#)。

【基础】文中提到了 jQuery，我相信很多前端程序员对它很熟悉了，它在前端开发中的地位无可替代，它是如此之好用和通用，以至于让一些程序员患上了“jQuery 依赖症”，离开了它就不会写 JavaScript 来操纵 DOM 了。我们当然不鼓励任何形式的“依赖症”，但我们确实需要学好 jQuery，廖雪峰的网站上有一个[简短的入门](#)。

[Chrome 开发者工具的命令行 API](#)，熟知其中的一些常用命令，可以非常方便地在 Chrome 中定位前端问题，其中选择器的语法和 jQuery 非常相似。



# 全栈工程师修炼指南

从全栈入门到技能实战

熊燚

Oracle 首席软件工程师



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 15 | 重剑无锋，大巧不工：JavaScript面向对象

下一篇 17 | 不一样的体验：交互设计和页面布局



pyhhou  
2019-10-17

1. 使用过 React，前端的 MVC 框架的出现带动了整个前端的发展，现在前端可以和后端分离开来设计与实现，对比之前，前端页面的聚合需要后端，这些框架的出现，在大的层面上看也算是进行了一次前后端的解耦吧；不好的地方就是前端的技术变得非常的多，而且杂，并且很多前端技术的生命周期都很短，这增加了普通工程师的学习难度的同时，前端在技术层面上也没有一个大的“统一” ...

展开 ∨

作者回复: 第2点讲得非常好，👍。

关于你的问题，Reducer返回的是新状态，而不是改变原有状态（新状态会放到这个store里面），这点请注意，你可以参见 <https://redux.js.org/basics/reducers#handling-actions>。  
typo 我已经知会编辑修改，感谢！



👍 2



sky  
2019-10-21

我是直接从使用vue开始前端之路的。总体上来说感觉还是利大于弊的。



sky  
2019-10-21

三驾马车没有讲vue，老师对这个框架没有什么想说的吗？



没带就是没写  
2019-10-16

jquery现在已经不流行了，除非是维护老项目。开发比较大的新项目用不到，但是它的理念是很有启发性的，write less,do more.

展开 ∨



靠人品去赢  
2019-10-16

我觉得是更好，看到这个我就知道这个被绑定了，找后面的代码处理逻辑也好找。  
JQuery确实是解耦了，你定义好属性，后面各种选择器对应起来，但是感觉不好管理。根据ID我不小心又加了两个逻辑和之前的逻辑有冲突，可能达不到你之前预想。类选择器更是坑爹，可能我只是要个样式而已，结果你还顺便帮忙做了点别的。

展开 ∨



**leslie**

2019-10-16

打卡吧：程序的东西学起来太苦了，现在的框架完全不是早年的那些了、、、慢慢实践慢慢补、、、



**我叫徐小晋**

2019-10-16

一直以来没有用框架。都是用jquery。。。老师如果要选择一个框架。那个入门会好一点？

