xufei commented on 25 Mar

Owner

2015前端组件化框架之路

1. 为什么组件化这么难做

Web应用的组件化是一个很复杂的话题。

在大型软件中，组件化是一种共识，它一方面提高了开发效率，另一方面降低了维护成本。但是在Web前端这个领域，并没有很通用的组件模式，因为缺少一个大家都能认同的实现方式，所以很多框架/库都实现了自己的组件化方式。

前端圈最热衷于造轮子了，没有哪个别的领域能出现这么混乱而欣欣向荣的景象。这一方面说明前端领域的创造力很旺盛，另一方面却说明了基础设施是不完善的。

我曾经有过这么一个类比，说明某种编程技术及其生态发展的几个阶段：

- 最初的时候人们忙着补全各种API，代表着他们拥有的东西还很匮乏，需要在语言跟基础设施上继续完善
- 然后就开始各种模式，标志他们做的东西逐渐变大变复杂，需要更好的组织了
- 然后就是各类分层MVC，MVP，MVVM之类，可视化开发，自动化测试，团队协同系统等等，说明重视生产效率了，也就是所谓工程化

那么，对比这三个阶段，看看关注这三种东西的人数，觉得Web发展到哪一步了？

细节来说，大概是模块化和组件化标准即将大规模落地（好坏先不论），各类API也大致齐备了，终于看到起飞的希望了，各种框架几年内会有非常强力的洗牌，如果不考虑老旧浏览器的拖累，这个洗牌过程将大大加速，然后才能释放Web前端的产能。

但是我们必须注意到，现在这些即将普及的标准，很多都会给之前的工作带来改变。用工业体系的发展史来对比，前端领域目前正处于蒸汽机发明之前，早期机械（比如《木兰辞》里面的机杼，主要是动力与材料比较原始）已经普及的这么一个阶段。

所以，从这个角度看，很多框架/库是会消亡的（专门做模块化的AMD和CMD相关库，专注于标准化DOM选择器铺垫的某些库），一些则必须进行革新，还有一些受的影响会比较小（数据可视化等相关方向），可以有机会沿着自己的方向继续演进。

2. 标准的变革

对于这类东西来说，能获得广泛群众基础的关键在于：对将来的标准有怎样的迎合程度。对前端编程方式可能造成重大影响的标准有这些：

- module
- Web Components

- class
- observe
- promise

module的问题很好理解，JavaScript第一次有了语言上的模块机制，而Web Components则是约定了基于泛HTML体系构建组件库的方式，class增强了编程体验，observe提供了数据和展现分离的一种优秀方式，promise则是目前前端最流行的异步编程方式。

这里面只有两个东西是绕不过去的，一是module，一是Web Components。前者是模块化基础，后者是组件化的基础。

module的标准化，主要影响的是一些AMD/CMD的加载和相关管理系统，从这个角度来看，正如seajs团队的**@afc163** 所说，不管是AMD还是CMD，都过时了。

模块化相对来说，迁移还比较容易，基本只是纯逻辑的包装，跟AMD或者CMD相比，包装形式有所变化，但组件化就是个比较棘手的问题了。

Web Components提供了一种组件化的推荐方式，具体来说，就是：

- 通过shadow DOM封装组件的内部结构
- 通过Custom Element对外提供组件的标签
- 通过Template Element定义组件的HTML模板
- 通过HTML imports控制组件的依赖加载

这几种东西，会对现有的各种前端框架/库产生很巨大的影响：

- 由于shadow DOM的出现，组件的内部实现隐藏性更好了，每个组件更加独立，但是这使得CSS变得很破碎，LESS和SASS这样的样式框架面临重大挑战。
- 因为组件的隔离，每个组件内部的DOM复杂度降低了，所以选择器大多数情况下可以限制在组件内部了，常规选择器的复杂度降低，这会导致人们对jQuery的依赖下降。
- 又因为组件的隔离性加强，致力于建立前端组件化开发方式的各种框架/库（除Polymer外），在自己的组件实现方式与标准Web Components的结合，组件之间数据模型的同步等问题上，都遇到了不同寻常的挑战。
- HTML imports和新的组件封装方式的使用，会导致之前常用的以JavaScript为主体的各类组件定义方式处境尴尬，它们的依赖、加载，都面临了新的挑战，而由于全局作用域的弱化，请求的合并变得困难得多。

3. 当下最时髦的前端组件化框架/库

在2015年初这个时间点看，前端领域有三个框架/库引领时尚，那就是Angular，Polymer，React（排名按照首字母），在知乎的这篇2014 年末有哪些比较火的 Web 开发技术？里，我大致回答过一些点，其他几位朋友的答案也很值得看。关于这三者的细节分析，侯振宇的这篇讲得很好：2015前端框架何去何从

我们可以看到，Polymer这个东西在这方面是有先天优势的，因为

它的核心理念就是基于Web Components的，也就是说，它基本没有考虑如何解决当前的问题，直接以未来为发展方向了。

React的编程模式其实不必特别考虑Web标准，它的迁移成本并不算高，甚至由于其实现机制，屏蔽了UI层实现方式，所以大家能看到在native上的使用，canvas上的使用，这都是与基于DOM的编程方式大为不同的，所以对它来说，处理Web Components的兼容问题要在封装标签的时候解决，反正之前也是要封装。

Angular 1.x的版本，可以说是跟同时代的多数框架/库一样，对未来标准的兼容基本没有考虑，但是重新规划之后的2.0版本对此有了很多权衡，变成了激进变更，突然就变成一个未来的东西了。

这三个东西各有千秋，在可以预见的几年内将会鼎足三分，也许还会有新的框架出现，能不能比这几个流行就难说了。

此外，原Angular 2.0的成员Rob Eisenberg创建了自己的新一代框架aurelia，该框架将成为Angular 2.0强有力的竞争者。

4. 前端组件的复用性

看过了已有的一些东西之后，我们可以大致来讨论一下前端组件化的一些理念。假设我们有了某种底层的组件机制，先不管它是浏览器原生的，或者是某种框架/库实现的约定，现在打算用它来做一个大型的Web应用，应该怎么做呢？

所谓组件化，核心意义莫过于提取真正有复用价值的东西。那怎样的东西有复用价值呢？

- 控件
- 基础逻辑功能
- 公共样式
- 稳定的业务逻辑

对于控件的可复用性，基本上是没有争议的，因为这是实实在在的通用功能，并且比较独立。

基础逻辑功能主要指的是一些与界面无关的东西，比如underscore这样的辅助库，或者一些校验等等纯逻辑功能。

公共样式的复用性也是比较容易认可的，因此也会有bootstrap，foundation，semantic这些东西的流行，不过它们也不是纯粹的样式库了，也带有一些小的逻辑封装。

最后一块，也就是业务逻辑。这一块的复用是存在很多争议的，一方面是，很多人不认同业务逻辑也需要组件化，另一方面，这块东西究竟怎样去组件化，也很需要思考。

除了上面列出的这些之外，还有大量的业务界面，这块东西很显然复用价值很低，基本不存在复用性，但仍然有很多方案中把它们“组件化”了，使得它们成为了“不具有复用性的组件”。为什么会出现这种情况呢？

组件化的本质目的并不一定是要为了可复用，而是提升可维护性。这一点正如面向对象语言，Java要比C++纯粹，因为它不允许例外情况的出现，连main函数都必须写到某个类里，所以Java是纯面向对象语言，而C++不是。

在我们这种情况下，也可以把组件化分为：全组件化，局部组件化。怎么理解这两个东西的区别呢，有人问过js框架和库的区别是什么，一般来说，有某种较强约定的东西，称为框架，而约定比较松散的，称为库。框架很多都是有全组件化理念的，比如说，很多年前就出现的ExtJS，它是全组件化框架，而jQuery和它的插件体系，则是局部组件化。所以用ExtJS写东西，不管写什么都是差不多一样的写法，而用jQuery的时候，大部分地方是原始HTML，哪里需要有些不一样的东西，就只在那个地方调用插件做一下特殊化。

对于一个有一定规模的Web应用来说，把所有东西都“组件化”，在管理上会有较大的便利性。我举个例子，同样是编写代码，短代码明显比长代码的可读性更高，所以很多语言里会建议“一个方法一般不要超过多少行，一个类最好不要超过多少行”之类。在Web前端这个体系里，JavaScript这块是做得相对较好的，现在入门水平的人，也已经很少会有把一堆js都写在一起的了。CSS这块，最近在SASS，LESS等框架的引领下，也逐步往模块化方面发展，否则直接编写bootstrap那种css，会非常痛苦。

这个时候我们再看HTML的部分，如果不考虑模板等技术的使用，某些界面光布局代码写起来就非常多了，像一些表单，都需要一层套一层，很多简单的表单元素都需要套个三层左右，更不必说一些有复杂布局的东西了。尤其是整个系统单页化之后，界面的header，footer，各种nav或者aside，很可能都有一定复杂性。如果这些东西的代码不作切分，那么主界面的HTML一定比较难看。

我们先不管用什么方式切分了，比如用某种模板，用类似Angular中的include，或者Polymer，React中的标签，或者直接使用原生Web Components，总之是把一块一块都拆开了，然后包含进来。从这个角度看，这些拆出去的东西都像组件，但如果从复用性的角度看，很可能多数东西，每一块都只有一个地方用，压根没有复用度。这个拆出去，纯粹是为了使得整个工程易于管理，易于维护。

这时候我们再来关注不同框架/库对UI层组件化的处理方式，发现有两个类型，模板和函数。

模板是一种很常见的东西，它用HTML字符串的方式表达界面的原始结构，然后通过代入数据的方式生成真正的界面，有的是生成目标HTML，有的还生成各种事件的自动绑定。前者是静态模板，后者是动态模板。

另外有一些框架/库偏爱用函数逻辑来生成界面，早期的ExtJS，现在的React（它内部还是可能使用模板，而且对外提供的是组件创建接口的进一步封装——jsx）等，这种实现技术的优势是不同平台上编程体验一致，甚至可以给每种平台封装相同的组件，调用方轻松写一份代码，在Web和不同Native平台上可用。但这种方式也有比较麻烦的地方，那就是界面调整比较繁琐。

本文前面部分引用侯振宇的那篇文章里，他提出这些问题：

如何能把组件变得更易重用? 具体一点:

- 我在用某个组件时需要重新调整一下组件里面元素的顺序怎么办?
- 我想要去掉组件里面某一个元素怎么办? 如何把组件变得更易扩展? 具体一点:

- 业务方不断要求给组件加功能怎么办?

  为此，还提出了“模板复写”方案，在这一点上我有不同意见。

  我们来看看如何把一个业务界面切割成组件。

  有这么一个简单场景：一个雇员列表界面包括两个部分，雇员表格和用于填写雇员信息的表单。在这个场景下，存在哪些组件？

  对于这个问题，主要存在两种倾向，一种是仅仅把“控件”和比较有通用性的东西封装成组件，另外一种是整个应用都组件化。

  对前一种方式来说，这里面只存在数据表格这么一个组件。
  对后一种方式来说，这里面有可能存在：数据表格，雇员表单，甚至还包括雇员列表界面这么一个更大的组件。

  这两种方式，就是我们之前所说的“局部组件化”，“全组件化”。

  我们前面提到，全组件化在管理上是存在优势的，它可以把不同层面的东西都搞成类似结构，比如刚才的这个业务场景，很可能最后写起来是这个样子：

```
<Employee-Panel>
    <Employee-List></Employee-List>
    <Employee-Form></Employee-Form>
</Employee-Panel>
```

  对于UI层，最好的组件化方式是标签化，比如上面代码中就是三个标签表达了整个界面。但我个人坚决反对滥用标签，并不是把各种东西都尽量封装就一定好。

  全标签化的问题主要有这些：

  第一，语义化代价太大。只要用了标签，就一定需要给它合适的语义，也就是命名。但实际用的时候，很可能只是为了把一堆html简化一下而已，到底简化出来的那东西应当叫什么名字，光是起名也费不知多少脑细胞。比如你说雇员管理的表单，这个表单有heading吗，有footer吗，能折叠吗，等等，很难起一个让别人一看就知道的名字，要么就是特别长。这还算简单的，因为我们是全组件化，所以很可能会有组合了多种东西的一个较复杂的界面，你想来想去也没法给它起个名字，于是写了个：

```
<Panel-With-Department-Panel-On-The-Left-And-Employee-Panel-On-The-Right>
</Panel-With-Department-Panel-On-The-Left-And-Employee-Panel-On-The-Right>
```

  这尼玛……可能我夸张了点，但很多时候项目规模够大，你不起这么复杂的名字，最后很可能没法跟功能类似的一个组件区分开，因为这些该死的组件都存在于同一个命名空间中。如果仅仅是当作一个界面片段来include，就不存在这种心理负担了。

  比如Angular里面的这种：

```
<div ng-include="'aaa/bbb/ccc.html'"></div>
```

  就不给它什么名字，直接include进来，用文件路径来区分。这个片段的作用可以用其目录结构描述，也就是通过物理名而非逻辑名来标识，目录层次充当了一个很好的命名空间。

  现在的一些主流MVVM框架，比如knockout，angular，avalon，

vue等等，都有一种“界面模板”，但这种模板并不仅仅是模板，而是可以视为一种配置文件。某一块界面模板描述了自身与数据模型的关系，当它被解析之后，按照其中的各种设置，与数据建立关联，并且反过来再更新自身所对应的视图。

不含业务逻辑的UI（或者是业务逻辑已分离的UI）基本不适合作为组件来看待，因为即使在逻辑不变的情况下，界面改版的可能性也太多了。比如即使是换了新的CSS实现方式，从float布局改成flex布局，都有可能把DOM结构少套几层div，因此，在使用模板的方案中，只能把界面层视为配置文件，不能看成组件，如果这么做，就会轻松很多。

部队行军的时候讲究“逢山开路，遇水搭桥”，这句话的重点在于只有到某些地形才开路搭桥，使用MVVM这类模式解决的业务场景，多数时候是一马平川，横着走都可以，不必硬要造路。所以从整个方案看的话，UI层实现应该是模板与控件并存，大部分地方是模板，少数地方是需要单独花时间搞的路和桥。

第二，配置过于复杂。有很多东西其实不太适合封装，不但封装的代价大，使用的代价也会很大。有时候会发现，调用代码的绝大部分都是在写各种配置。

就像刚才的雇员表单，既然你不从标签的命名上去区分，那一定会在组件上加配置。比如你原来想这样：

```
<EmployeeForm heading="雇员表单"></EmployeeForm>
```

然后在组件内部，判断有没有设置heading，如果没有就不显示，如果有，就显示。过了两天，产品问能不能把heading里面的某几个字加粗或者换色，然后码农开始允许这个heading属性传入html。没多久之后，你会惊奇地发现有人用你的组件，没跟你说，就在heading里面传入了折叠按钮的html，并且用选择器给折叠按钮加了事件，点一下之后还能折叠这个表单了……

然后你一想，这个不行，我得给他再加个配置，让他能很简单地控制折叠按钮的显示，但是现在这么写太不直观，于是采用对象结构的配置：

```
<EmployeeForm>
    <Option collapsible="true">
        <Heading>
            <h4><strong>雇员</strong>表单</h4>
        </Heading>
    </Option>
</EmployeeForm>
```

然后又有一天，发现有很多面板都可以折叠，然后特意创建了一个可折叠面板组件，又创建了一种继承机制，其他普通业务面板从它继承，从此一发不可收拾。

我举这例子的意思是为了说明什么呢，我想说，在规模较大的项目中，企图用全标签化加配置的方式来描述所有的普通业务界面，是一定事倍功半的，并且这个规模越大就越坑，这也正是ExtJS这类对UI层封装过度的体系存在的最大问题。

这个问题讨论完了，我们来看看另外一个问题：如果UI组件有业务逻辑，应该如何处理。

比如说，性别选择的下拉框，它是一个非常通用化的功能，照理

说是很适合被当做组件来提供的。但是究竟如何封装它，我们就有些犯难了。这个组件里除了界面，还有数据，这些数据应当内置在组件里吗？理论上从组件的封装性来说，是都应当在里面的，于是就这么造了一个组件：

```
<GenderSelect></GenderSelect>
```

这个组件非常美好，只需直接放在任意的界面中，就能显示带有性别数据的下拉框了。性别的数据很自然地是放在组件的实现内部，一个写死的数组中。这个太简单了，我们改一下，改成商品销售的国家下拉框。

表面上看，这个没什么区别，但我们有个要求，本公司商品销售的国家的信息是统一配置的，也就是说，这个数据来源于服务端。这时候，你是不是想把一个http请求封装到这组件里？

这样做也不是不可以，但存在至少两个问题：

- 如果这类组件在同一个界面中出现多次，就可能存在请求的浪费，因为有一个组件实例就会产生一个请求。
- 如果国家信息的配置界面与这个组件同时存在，当我们在配置界面中新增一个国家了，下拉框组件中的数据并不会实时刷新。

第一个问题只是资源的浪费，第二个就是数据的不一致了。曾经在很多系统中，大家都是手动刷新当前页面来解决这问题的，但到了这个时代，人们都是追求体验的，在一个全组件化的解决方案中，不应再出现此类问题。

如何解决这样的问题呢？那就是引入一层Store的概念，每个组件不直接去到服务端请求数据，而是到对应的前端数据缓存中去获取数据，让这个缓存自己去跟服务端保持同步。

所以，在实际做方案的过程中，不管是基于Angular，React，Polymer，最后肯定都做出一层Store了，不然会有很多问题。

5. 为什么MVVM是一种很好的选择

我们回顾一下刚才那个下拉框的组件，发现存在几个问题：

- 界面不好调整。刚才的那个例子相对简单，如果我们是一个省市县三级联动的组件，就比较麻烦了。比如说，我们想要把水平布局改成垂直的，又或者，想要把中间的label的字改改，都会非常麻烦。按照传统的做组件的方式，就要加若干配置项，然后组件里面去分别判断，修改DOM结构。
- 如果数据的来源不是静态json，而是某个动态的服务接口，那用起来就很麻烦。
- 我们更多地需要业务逻辑的复用和纯“控件”的复用，至于那些绑定业务的界面组件，复用性其实很弱。

所以，从这些角度，会尽量期望在HTML界面层与JavaScript业务逻辑之间，存在一种分离。

这时候，再看看绝大多数界面组件存在什么问题：

有时候我们考虑一下DOM操作的类型，会发现其实是很容易枚举

的：

- 创建并插入节点
- 移除节点
- 节点的交换
- 属性的设置

多数界面组件封装的绝大部分内容不过是这些东西的重复。这些东西，其实是可以通过某些配置描述出来的，比如说，某个数组以什么形式渲染成一个select或者无序列表之类，当数组变动，这些东西也跟着变动，这些都应当被自动处理，如果某个方案在现在这个时代还手动操作这些，那真的是一种落伍。

所以我们可以看到，以Angular，Knockout，Vue，Avalon为代表的框架们在这方面做了很多事，尽管理念有所差异，但大方向都非常一致，也就是把大多数命令式的DOM操作过程简化为一些配置。

有了这种方式之后，我们可以追求不同层级的复用：

- 业务模型因为是纯逻辑，所以非常容易复用
- 视图模型基本上也是纯逻辑，界面层多数是纯字符串模板，同一个视图模型搭配不同的界面模板，可以实现视图模型的复用
- 同一个界面模板与不同的视图模型组合，也能直接组合出完全不同的东西

所以这么一来，我们的复用粒度就非常灵活了。正因为这样，我一直认为Angular这样的框架战略方向是很正确的，虽然有很多战术失误。我们在很多场景下，都是需要这样的高效生产手段的。

6. 组件的长期积累

我们做组件化这件事，一定是一种长期打算，为了使得当前的很多东西可以作为一种积累，在将来还能继续使用，或者仅仅作较小的修改就能使用，所以必须考虑对未来标准的兼容。主要需要考虑的方面有这几点：

- 尽可能中立于语言和框架，使用浏览器的原生特性
- 逻辑层的模块化（ECMAScript module）
- 界面层的元素化（Web Components）

之前有很多人对Angular 2.0的激进变更很不认同，但它的变更很大程度上是对标准的全面迎合。这不仅仅是它的问题，其实是所有前端框架的问题。不面对这些问题，不管现在多么好，将来都是死路一条。这个问题的根源是，这几个已有的规范约束了模块化和元素化的推荐方式，并且，如果要对当前和未来两边做适配的话，基本就没法干了，导致以前的都不得不做一定的迁移。

模块化的迁移成本还比较小，无论是之前AMD还是CMD的，都可以根据一些规则转换过来，但组件化的迁移成本太大了，几乎每种框架都会提出自己的理念，然后有不同的组件化理念。

还是从三个典型的东西来说：Polymer，React，Angular。

Polymer中的组件化，其实就是标签化。这里的标签，并不只是界面元素，甚至逻辑组件也可以这样，比如这个代码：

```
<my-panel>
    <core-ajax id="ajax" url="http://url" params="{{form
data}}" method="post"></core-ajax>
</my-panel>
```

注意到这里的core-ajax标签，很明显这已经是纯逻辑的了，在大多数前端框架或者库中，调用ajax肯定不是这样的，但在浏览器端这么干也不是它独创，比如flash里面的WebService，比如早期IE中基于htc实现的webservice.htc等等，都是这么干的。在Polymer中，这类东西称为非可见元素（non-visual-element）。

React的组件化，跟Polymer略有不同，它的界面部分是标签化，但如果有单纯的逻辑，还是纯JavaScript模块。

既然大家的实现方式都那么不一致，那我们怎么搞出尽量可复用的组件呢？问题到最后还是要绕到Web Components上。

在Web Components与前端组件化框架的关系上，我觉得是这么个样子：

各种前端组件化框架应当尽可能以Web Components为基石，它致力于组织这些Components与数据模型之间的关系，而不去关注某个具体Component的内部实现，比如说，一个列表组件，它究竟内部使用什么实现，组件化框架其实是不必关心的，它只应当关注这个组件的数据存取接口。

然后，这些组件化框架再去根据自己的理念，进一步对这些标准Web Components进行封装。换句话说，业务开发人员使用某个组件的时候，他是应当感知不到这个组件内部究竟使用了Web Components，还是直接使用传统方式。（这一点有些理想化，可能并不是那么容易做到，因为我们还要管理像import之类的事情）。

7. 我们需要关注什么

目前来看，前端框架/库仍然处于混战期，可比中国历史上的春秋战国，百家齐放，作为跟随者来说，这是很痛苦的，因为无所适从，很可能你作为一个企业的前端架构师或者技术经理，需要做一些选型工作，但选哪个能保证几年后不被淘汰呢？基本没有。

虽然我们不知道将来什么框架会流行，但我们可以从一些细节方面去关注，某个具体的方面，将来会有什么，也可以了解一下在某个具体领域存在什么样的方案。一个完整的框架方案，无非是以下多个方面的综合。

7.1 模块化

这块还是不讲了，支付宝seajs还有百度ecomfe这两个团队的人应该都能比我讲得好得多。

7.2 Web Components

本文前面讨论过一些，也不深入了。

7.3 变更检测

我们知道，现代框架的一个特点是自动化，也就是把原有的一些手动操作提取。在前端编程中，最常见的代码是在干什么呢？读写数据和操作DOM。不少现代的框架/库都对这方面作了处理，比如说通过某种配置的方式，由框架自动添加一些关联，当数据变更的时候，把DOM进行相应修改，又比如，当DOM发生变动的时候，也更新对应的数据。

这个关联过程可能会用到几种技术。首先我们看怎么知道数据在变化，这里面有三种途径：

一、存取器的封装。这个的意思也就是对数据进行一层包装，比如：

```
var data = {
    name: "aaa",
    getName: function() {
        return this.name;
    },
    setName: function(value) {
        this.name = value;
    }
}
```

这样，不允许用户直接调用data.name，而是调用对应的两个函数。Backbone就是通过这样的机制实现数据变动观测的，这种方式适用于几乎所有浏览器，缺点就是比较麻烦，要对每个数据进行包装。

这个机制在稍微新一点的浏览器中，也有另外一种实现方式，那就是defineProperty相关的一些方法，使用更优雅的存取器，这样外界可以不用调用函数，而是直接用data.name这样进行属性的读写。

国产框架avalon使用了这个机制，低版本IE中没有defineProperty，但在低版本IE中不止有JavaScript，还存在VBScript，那里面有存取器，所以他巧妙地使用了VBS做了这么一个兼容封装。

基于存取器的机制还有个麻烦，就是每次动态添加属性，都必须再添加对应的存取器，否则这个属性的变更就无法获取。

二、脏检测。

以Angular 1.x为代表的框架使用了脏检测来获知数据变更，这个机制的大致原理是：

保存数据的新旧值，每当有一些DOM或者网络、定时器之类的事件产生，用这个事件之后的数据去跟之前保存的数据进行比对，如果相同，就不触发界面刷新，否则就刷新。

这个方式的理念是，控制所有可能导致数据变更的来源（也就是各种事件），在他们可能对数据进行操作之后，判断新旧数据是否有变化，忽略所有中间变更，也就是说，如果你在同一个事件中，把某个数据任意修改了很多次，但最后改回来了，框架会认为你什么都没干，也就不会通知界面去刷新了。

不可否认的是，脏检测的效率是比较低的，主要是不能精确获知数据变更的影响，所以当数据量更大的情况下，浪费更严重，需要手动作一些优化。比如说一个很大的数组，生成了一个界面上

的列表，当某个项选中的时候，改变颜色。在这种机制下，每次改变这个项的数据状态，就需要把所有的项都跟原来比较一遍，然后，还要再全部比较一次发现没有关联引起的变化了，才能对应刷新界面。

三、观察机制。

在ES7里面，引入了Object的observe方法，可以用于监控对象或数组的变动。

这是目前为止最合理的观测方案。这个机制很精确高效，比如说，连长跟士兵说，你去观察对面那个碉堡里面的动静。这个含义很复杂，包括什么呢？

- 是不是加人了
- 是不是有人离开了
- 谁跟谁换岗了
- 上面的旗子从太阳旗换成青天白日了

所谓观察机制，也就是观测对象属性的变更，数组元素的新增，移除，位置变更等等。我们先思考一下界面和数据的绑定，这本来就应当是一个外部的观察，你是数据，我是界面，你点头我微笑，你伸手我打人。这种绑定本来就应当是个松散关系，不应当因为要绑定，需要破坏原有的一些东西，所以很明显更合理。

除了数据的变动可以被观察，DOM也是可以的。但是目前绝大多数双向同步框架都是通过事件的方式把DOM变更同步到数据上。比如说，某个文本框绑定了一个对象的属性，那很可能，框架内部是监控了这个文本框的键盘输入、粘贴等相关事件，然后取值去往对象里写。

这么做可以解决大部分问题，但是如果你直接myInput.value="111"，这个变更就没法获取了。这个不算大问题，因为在一个双向绑定框架中，一个既被监控，又手工赋值的东西，本身也比较怪，不过也有一些框架会尝试从HTMLInputELement的原型上去覆盖value赋值，尝试把这种东西也纳入框架管辖范围。

另外一个问题，那就是我们只考虑了特定元素的特定属性，可以通过事件获取变更，如何获得更广泛意义上的DOM变更？比如说，一般属性的变更，或者甚至子节点的增删？

DOM4引入了MutationObserver，用于实现这种变更的观测。在DOM和数据之间，是否需要这么复杂的观测与同步机制，目前尚无定论，但在整个前端开发逐步自动化的大趋势下，这也是一种值得尝试的东西。

复杂的关联监控容易导致预期之外的结果：

- 慕容复要复国，每天读书练武，各种谋划
- 王语嫣观察到了这种现象，认为表哥不爱自己了
- 段誉看到神仙姐姐闷闷不乐，每天也茶饭不思
- 镇南王妃心疼爱子，到处调查这件事的原委，意外发现段正淳还跟旧爱有联系
- ……

总之这么下来，最后影响到哪里了都不知道，谁让丘处机路过牛

家村呢？

所以，变更的关联监控是很复杂的一个体系，尤其是其中产生了闭环的时候。搭建整个这么一套东西，需要极其精密的设计，否则熟悉整套机制的人只要用特定场景轻轻一推就倒了。灵智上人虽然武功过人，接连碰到欧阳锋，周伯通，黄药师，全部都是上来就直接被抓了后颈要害，大致就是这意思。

polymer实现了一个observe-js，用于观测数组、对象和路径的变更，有兴趣的可以关注。

在有些框架，比如aurelia中，是混合使用了存取器和观察模式，把存取器作为观察模式的降级方案，在浏览器不支持observe的情况下使用。值得一提的是，在脏检测方式中，变更是合并后批量提交的，这一点常常被另外两种方案的使用者忽视。其实，即使用另外两种方式，也还是需要一个合并与批量提交过程。

怎么理解这个事情呢？数据的绑定，最终都是要体现到界面上的，对于界面来说，其实只关注你每一次操作所带来的数据变更的始终，并不需要关心中间过程。比如说，你写了这么一个循环，放在某个按钮的点击中：

```
for (var i=0; i<10000; i++) {
    obj.a += 1;
}
```

界面有一个东西绑定到这个a，对框架来说，绝对不应当把中间过程直接应用到界面上，以刚才这个例子来说，合理的情况只应当存在一次对界面DOM的赋值，这个值就是对obj.a进行了10000次赋值之后的值。尽管用存取器或者观察模式，发现了对obj上a属性的这10000次赋值过程，这些赋值还是都必须被舍弃，否则就是很可怕的浪费。

React使用虚拟DOM来减少中间的DOM操作浪费，本质跟这个是一样的，界面只应当响应逻辑变更的结束状态，不应当响应中间状态。这样，如果有一个ul，其中的li绑定到一个1000元素的数组，当首次把这个数组绑定到这个ul上的时候，框架内部也是可以优化成一次DOM写入的，类似之前常用的那种DocumentFragment，或者是innerHTML一次写入整个字符串。在这个方面，所有优化良好的框架，内部实现机制都应当类似，在这种方案下，是否使用虚拟DOM，对性能的影响都是很小的。

7.4 Immutable Data

Immutable Data是函数式编程中的一个概念，在前端组件化框架中能起到一些很独特的作用。

它的大致理念是，任何一种赋值，都应当被转化成复制，不存在指向同一个地方的引用。比如说：

```
var a = 1;
var b = a;
b = 2;

console.log(a==b);
```

这个我们都知道，b跟a的内存地址是不一致的，简单类型的赋值会进行复制，所以a跟b不相等。但是：

```
var a = {
    counter : 1
};
var b = a;

b.counter++;
console.log(a.counter==b.counter);
```

这时候因为a和b指向相同的内存地址，所以只要修改了b的counter，a里面的counter也会跟着变。

Immutable Data的理念是，我能不能在这种赋值情况下，直接把原来的a完全复制一份给b，然后以后大家各自变各自的，互相不影响。光凭这么一句话，看不出它的用处，看例子：

对于全组件化的体系，不可避免会出现很多嵌套的组件。嵌套组件是一个很棘手的问题，在很多时候，是不太好处理的。嵌套组件所存在的问题主要在于生命周期的管理和数据的共享，很多已有方案的上下级组件之间都是存在数据共享的，但如果内外层存在共享数据，那么就会破坏组件的独立性，比如下面的一个列表控件：

```
<my-list list-data="{arr}">
    <my-listitem></my-listitem>
    <my-listitem></my-listitem>
    <my-listitem></my-listitem>
</my-list>
```

我们在赋值的时候，一般是在外层整体赋值一个类似数组的数据，而不是自己挨个在每个列表项上赋值，不然就很麻烦。但是如果内外层持有相同的引用，对组件的封装性很不利。

比如在刚才这个例子里，假设数据源如下：

```
var arr = [
    {name: "Item1"},
    {name: "Item2"},
    {name: "Item3"}
];
```

通过类似这样的方式赋值给界面组件，并且由它在内部给每个子组件分别进行数据项的赋值：

```
list.data = arr;
```

赋值之后会有怎样的结果呢？

```
console.log(list.data == arr);
console.log(listitem0.data == arr[0]);
console.log(listitem1.data == arr[1]);
console.log(listitem2.data == arr[2]);
```

这种方案里面，后面那几个log输出的结果都会是true，意思就是内层组件与外层共享数据，一旦内层组件对数据进行改变，外层中的也就改变了，这明显是违背组件的封装性的。

所以，有一些方案会引入Immutable Data的概念。在这些方案里，内外层组件的数据是不共享的，它们的引用不同，每个组件实际上是持有了自己的数据，然后引入了自动的赋值机制。

这时候再看看刚才那个例子，就会发现两层的职责很清晰：

- 外层持有一个类似数组的东西arr，用于形成整个列表，但并不关

注每条记录的细节

- 内层持有某条记录，用于渲染列表项的界面
- 在整个列表的形成过程中，list组件根据arr的数据长度，实例化若干个listitem，并且把arr中的各条数据赋值给对应的listitem，而这个赋值，就是immutable data起作用的地方，其实是把这条数据复制了一份给里面，而不是把外层这条记录的引用赋值进去。内层组件发现自己的数据改变之后，就去进行对应的渲染
- 如果arr的条数变更了，外层监控这个数据，并且根据变更类型，添加或者删除某个列表项
- 如果从外界改变了arr中某一条记录的内容，外层组件并不直接处理，而是给对应的内层进行了一次赋值
- 如果列表项中的某个操作，改变了自身的值，它首先是把自己持有的数据进行改变，然后，再通过immutable data把数据往外同步一份，这样，外层组件中的数据也就更新了。

所以我们再看这个过程，真是非常清晰明了，而且内外层各司其职，互不干涉。这是非常有利于我们打造一个全组件化的大型Web应用的。各级组件之间存在比较松散的联系，而每个组件的内部则是封闭的，这正是我们所需要的结果。

说到这里，需要再提一个容易混淆的东西，比如下面这个例子：

```
<outer-component>
    <inner-component></inner-component>
</outer-component>
```

如果我们为了给inner-component做一些样式定位之类的事情，很可能在内外层组件之间再加一些额外的布局元素，比如变成这样：

```
<outer-component>
    <div>
        <inner-component></inner-component>
    </div>
</outer-component>
```

这里中间多了一级div，也可能是若干级元素。如果有用过Angular 1.x的，可能会知道，假如这里面硬造一级作用域，搞个ng-if之类，就可能存在多级作用域的赋值问题。在上面这个例子里，如果在最外层赋值，数据就会是outer -> div -> inner这样，那么，从框架设计的角度，这两次赋值都应当是immutable的吗？

不是，第一次赋值是非immutable，第二次才需要是，immutable赋值应当仅存在于组件边界上，在组件内部不是特别有必要使用。刚才的例子里，依附于div的那层变量应当还是跟outer组件在同一层面，都属于outer组件的人民内部矛盾。

这里是facebook实现的immutable-js库

7.6 Promise与异步

前端一般都习惯于用事件的方式处理异步，但很多时候纯逻辑的“串行化”场景下，这种方式会让逻辑很难阅读。在新的ES规范里，也有yield为代表的各种原生异步处理方案，但是这些方案仍然有很大的理解障碍，流行度有限，很大程度上会一直停留在基础较好的开发人员手中。尤其是在浏览器端，它的受众应该会比node里面还要狭窄。

前端里面，处理连续异步消息的最能被广泛接受的方案是promise，我这里并不讨论它的原理，也不讨论它在业务中的使用，而是要提一下它在组件化框架内部所能起到的作用。

现在已经没有哪个前端组件化框架可以不考虑异步加载问题了，因为，在前端这个领域，加载就是一个绕不过去的坎，必须有了加载，才能有执行过程。每个组件化框架都不能阻止自己的使用者规模膨胀，因此也应当在框架层面提出解决方案。

我们可能会动态配置路由，也可能在动态加载的路由中又引入新的组件，如何控制这些东西的生命周期，值得仔细斟酌，如果在框架层面全异步化，对于编程体验的一致性是有好处的。将各类接口都promise化，能够在可维护性和可扩展性上提供较多便利。

我们之前可能熟知XMLHTTP这样的通信接口，这个东西虽然被广为使用，但是在优雅性等方面，存在一些问题，所以最近出来了替代方案，那就是fetch。

细节可以参见月影翻译的这篇【翻译】这个API很“迷人”——(新的Fetch API)

在不支持的浏览器上，也有github实现的一个polyfill，虽然不全，但可以凑合用window.fetch polyfill

大家可以看到，fetch的接口就是基于promise的，这应当是前端开发人员最容易接受的方案了。

7.7 Isomorphic JavaScript

这个东西的意思是前后端同构的JavaScript，也就是说，比如一块界面，可以选择在前端渲染，也可以选择在后端渲染，值得关注，可以解决像seo之类的问题，但现在还不能处理很复杂的状况，持续关注吧。

8. 小结

很感谢能看到这里，以上这些是我近一年的一些思考总结。从技术选型的角度看，做大型Web应用的人会很痛苦，因为这是一个青黄不接的年代，目前已有的所有框架/库都存在不同程度的缺陷。当你向未来看去，发现它们都是需要被抛弃，或者被改造的，人最痛苦的是在知道很多东西不好，却又要从中选取一个来用。@严清 跟@寸志 @题叶讨论过这个问题，认为现在这个阶段的技术选型难做，不如等一阵，我完全赞同他们的观点。

选型是难，但是从学习的角度，可真的是挺好的时代，能学的东西太多了，我每天路上都在努力看有可能值得看的东西，可还是看不完，只能努力去跟上时代的步伐。

以下一段，与诸位共勉：

It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness, it was the spring of hope, it was the winter of despair, we had everything before us, we

had nothing before us, we were all going direct to Heaven, we were all going direct the other way--in short, the period was so far like the present period, that some of its noisiest authorities insisted on its being received, for good or for evil, in the superlative degree of comparison only.