链滴

2015 前端生态发展回顾

作者：Vanessa

原文链接：https://ld246.com/article/1451532709100

来源网站：链滴

引用苏宁前端架构师(@xufei)的一个总结作为开篇

编程技术及生态发展的三个阶段

- 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，明重视生产效率了，也就是所谓工程化

处在2015年这个时间段来看，前端生态已经进入了第三阶段。看上去好像已经走的挺远了，实则不然如果再用人类历史上的三次工业革命来类比，前端发展其实不过刚刚迈入了蒸汽机时代，开始逐步用具来替代过往相当一部分的人肉作业，但是离电气时代的自动化流水线作业还有很长一段路要走。回一下2015年前端的生态发展，我大致整理了几个我觉得比较有历史意义的事件。

按时间顺序:

1. 年初React Native的发布，引领React正式走上历史舞台。
2. 3月angular2.0第一个预览版发布
3. 5月 http/2.0标准正式发布，同月 iojs 与 nodejs合并。
4. 6月 ES6 和 WebAssembly 落地
5. 7月 迄今为止React生态圈影响最大的Flux实现redux发布1.0版本
6. 8月 Facebook公开了在React上应用GraphQL的relay框架的技术预览版
7. 9月 React Native for Andriod 发布
8. 11月伊始，es标准委员会宣布将历时3年研究的Object.observe从草案中移除，尽管它原本已经是sage2，几乎已经是ES7的事实标准。双十一刚一结束，阿里手淘团队发布了名为 无线电商动态化解决案 的 Weex，也有人给了它一个更具象的名字，vue native。
9. 12月，赶在2015的尾巴，aurelia和angular2先后发布beta版。

css方面，postcss & cssnext先后高调走到台前。

观念的变化

由于近几年前端的野蛮生长以及前端应用的多元化和复杂化，整个技术形态已经跟几年前纯做页面的代完全迥异了。主要观念的变化总结来看在于一点，**现在的前端开发面向的是web app而不是web page**。今天的前端开发模式跟传统的GUI软件(如C++、.NET开发的windows客户端)已经很接近了，而由于现在前端领域为了解决日益复杂的web业务需求及体量，越来越多的借鉴了传统客户端的开发经，导致两者变得越来越趋同。再加上前端一些独特的特性(免安装、增量安装等)，工程上的复杂度有之而无不及。前端如今已经脱离了茹毛饮血、刀耕火种的原始社会，开始步入了工业时代。

框架 & 类库的变化

今年最火的框架/类库毫无疑问当属React了。React从2014年年中开始广泛受到开发者关注，但是真开始在社区独领风骚还得归功于2015年初React Native的发布。React Native的发布使得js统一三端前端、后端、移动端)开发成为可能(现在这个时间点看可能还是过于理想，但是整体方向还是对的)，一针强心剂吸引了大量开发者的眼球。笔者对此最大的感受就是，我在社区发表一篇react的入门教级别的软文便可获得广泛关注及转发，相应的写angular源码剖析的准干货大部分情况则是门可罗雀

原文链接：2015 前端生态发展回顾

oy。

我们挑几个主流的框架来讲讲这一层的变化。

React & Redux

React基本简介可以参考这篇文章React简介，这里不再赘述。我们挑几个核心特征简单来讲：

1. virtual dom

这个可以说是F家工程师超强工程能力的最佳体现了(Relay也算一个)，从本质来看它是通过用js objec来模拟了一个dom tree，然后将这层virtual dom插在react组件跟真实dom之间，配合强劲的dom di f算法实现它一直标榜的高性能。

2. jsx

同样为了配合react中的组件化开发模式，F家发明了一套新的语法jsx。乍看之下它像是html in js，也是初接触的开发者最难以接受的，典型的违背前端推崇多年的表现与业务分离的原则啊。其实这里换个角度来看jsx，它并不是html in js，准确来说它是一个用来构建react组件树的AST。这样来想你能理解react中这一看似怪异的设计了。

3. immutable object

不可变对象是函数式编程中的重要概念，react的介入使得这一理念在前端社区中流行起来。从目前种类库的实现来看，不可变对象在大型应用中拥有传统可变对象不具备的优势。尤其在这个内存不值的年代。从目前immutable object的良好走势来看，将来有可能被es纳入规范之中。目前可以通过fa ebook的immutable.js来实现。

Redux则是目前react配套的Flux模式的各种实现(其实现在两者的关系越来越模糊了)中最火的一个，此基础上它引入了函数式编程、单一数据源、不可变数据、中间件等概念。一定程度来讲，redux是年react生态甚至整个前端生态中影响最大的一个框架，它给整个前端技术栈引入了很多新成员，尽这些概念可能在其他领域已经有了广泛的应用。虽然它们是否会在大规模的应用实践中被广大开发者可还需要再检验，但至少给我们带来了一些新的思路。其中的单一数据源、不可变数据、中间件等思目前来看还是非常有价值的，尤其是单一数据源跟不可变数据，很有可能在将来成为大型应用架构中标配(目前来看至少在应用中构建Store层在当前的前端架构中是势在必行的)。单一数据源就好比在前构建了一个集中式数据库，所有的数据存取操作对象都是它，不单如此它里面还实现了触发器，当有in ert/update操作时它会对相应组件作rerender动作，这个在各组件之间有数据同步需求的场景下就非有用了。

至于我对函数式编程的看法，后面单独阐述。

在我看来，react的优势并不在组件化，组件化的实现方案多种多样。react的优势在于virtual dom及个几乎构成闭环的强大生态，这归功于Facebook工程师强大的工程能力跟架构能力。virtual dom将用表现层从浏览器这个基于dom的上下文中抽离出来，通过原生js对象模型的方式使得react具备在任环境支撑上层表现的能力。上层的渲染引擎可以是canvas、native、服务端甚至是桌面端，只要相应端提供基于react组件的渲染能力，即可达到一套代码、或者只要很少的改动就能移植到任一终端环的效果，这个就非常夸张了。react从0.14版本之后便将react-dom抽出来变成一个独立的库，可见re ct的野心并不局限于浏览器，相反从这点来看，react反而是受到了dom的掣肘。

Angular2 & Vue.js

ng2跟ng1相比是一个完全革命性版本而不是升级版，它是一个为了迎合未来的标准及理念来设计的新框架，而这些新的理念又无法通过改进ng1.x的方式来实施，所以angular团队做了这么一个看似激的决策，可以理解成重构已经无法满足需求只能重写了。ng2也采用纯组件化的开发思路，任何单元于它来说都是组件。同时，ng2里面也引入了一些全新的概念(对于前端而言)来提升框架的性能及设，例如基于worker的数据检测机制能大幅度提升渲染性能(对应实现是zone.js)，基于响应式编程的新

编程模型能更大的改善编码体验(对应实现RxJS)。赶在2015年的尾巴，ng2正式发布beta版，对于angular的这次自我革命是否能成功，还有待后续检验。另外原angular团队中出来的一个成员开发了一类ng2的框架aurelia，有相当的开发者认为它更配称为ng2，值得关注。

由于阿里在背后的技术实践及支持，Vue.js今年也开始得到越来越多的关注。vue相对于angular1.x优势在于轻量、易用、更优异的性能及面向组件化的设计，目前发展态势也非常好，是移动端开发的个重要技术选型之一。

标准 & 语言的变化

现在回顾起来，2015年是很有意义的一年：这一年是Web诞生25岁周年，也是js诞生的20周年。同又是ES6标准落地的一年。ES6是迄今为止ECMAScript标准最大的变革(如果不算上胎死腹中的ES4的)，带来了一系列令开发者兴奋的新特性。从目前es的进化速度来看，es后面应该会变成一个个的feature发布而不是像以前那样大版本号的方式，所以现在官方也在推荐 ES+年份 这种叫法而不是 ES +版。

ES2015(ES6) & ES2016(ES7) & TypeScript

6月中ES2015规范正式发布，从ES2015带来的这些革命性的新语法来看，JS从此具备了用于开发大应用的语言的基本要素：原生的mudule支持、原生的class关键字、更简洁的api及语法糖，更稳定的据类型。而这些new features中，有几个我认为是会影响整个前端发展进程的：

1. Module & Module Loader

ES2015中加入的原生模块机制支持可谓是意义最重大的feature了，且不说目前市面上五花八门的moule/loader库，各种不同实现机制互不兼容也就罢了(其实这也是非常大的问题)，关键是那些模块定义装载 语法都丑到爆炸，但是这也是无奈之举，在没有语言级别的支持下，js只能做到这一步，正所谓妇难为无米之炊。ES2016中的Module机制借鉴自CommonJS，同时又提供了更优雅的关键字及语法虽然也存在一些问题)。遗憾的是同样有重大价值的Module Loader在2014年底从ES2015草案中移了，我猜测可能是对于浏览器而言Module Loader的支持遭遇了一些技术上的难点，从而暂时性的舍了这一feature。但是一个原生支持的模块加载器是非常有意义的，相信它不久后还是会回归到ES规中(目前由WHATWG组织在单独维护)。

2. Class

准确来说class关键字只是一个js里构造函数的语法糖而已，跟直接function写法无本质区别。只不过了Class的原生支持后，js的面向对象机制有了更多的可能性，比如衍生的extends关键字(虽然也只是法糖)。

3. Promise & Reflect API

Promise的诞生其实已经有几十年了，它被纳入ES规范最大意义在于，它将市面上各种异步实现库的佳实践都标准化了。至于Reflect API，它让js历史上第一次具备了元编程能力，这一特性足以让开发们脑洞大开。

关于ES2016的最重磅的消息莫过于11月初es标准委员会宣布将Object.observe从ES2016草案中移除，尽管它已经是stage2几乎已经是事实标准。官方给出的解释是，这3年的时间前端世界变化实在太，社区已经有了一些更优秀简洁的实现了(polymer的observe-js)，而且React带来的immutable object在社区的流行使得基于可变数据的Object.observe的处境变的尴尬，O.o再继续下去的意义不大了。

除此之外，ES2016的相关草案也已经确定了一大部分其他new features。这里提两个我比较感兴趣的ew feature：

1. async/await

写过C#的同学应该对这两个关键字很熟悉了，async/await是为了更优雅的异步编程做的一个关键字别的封装，术语叫协程。ES2016中 async/await 实际是对Generator&Promise的上层封装，几乎同

的写法写异步比Promise更优雅更简单，非常值得期待。

2. decorator

字面意思是装饰器，其实等同于Java里面的注解。注解机制对于大型应用的开发的作用想必不用我过赘述了。用过的同学都说好。

目前ES2015/ES2016都有了比较优秀的转译器支持(没错我说的是babel)，但是也不是all features supported，尝新的过程中需要注意。

至于Typescript，你可以将它理解成加入了静态类型的js的超集。不过我对于这种转译型语言一直不冒(包括CoffeeScript)，有兴趣同学自己去了解下吧。。

WebAssembly

WebAssembly选择了跟ES2015在同一天发布，其项目领头人是大名鼎鼎的js之父Brendan Eich。WbAssembly旨在解决js作为解释性语言的先天性能缺陷，试图通过在浏览器底层加入编译机制从而提高s性能。这个事情跟当时V8做的类似(有兴趣的同学可以去了解下JIT)，V8也因此一跃成为世界上跑的快的js引擎。但是由于js是弱类型的动态语言，V8很快就触碰到了性能优化的天花板，因为很多场景还是免不了recompile的过程。因此WebAssembly索性将编译过程前移(AOT)。WebAssembly提供具将各种语言转换成特定的字节码，浏览器直接面向字节码编译程序。其实在此之前,firefox已经搞过am.js做类似的事情，只不过WebAssembly的方案更激进。有人认为WebAssembly可能是2016年最的黑马，如果wa能发展起来，若干年后我们看js编写的应用会像现在看汇编语言写出的大型程序的感。WebAssembly项目目前由苹果、谷歌、微软、Mozila四大浏览器厂商共同推进，还是非常值得期的(写不下去了我决定回去翻开我那本落灰的编译原理。。)。

Web Components

webcomponents规范起草于2013年，w3c标准委员会意图提供一种浏览器级别的组件化解决方案，过浏览器的原生支持定义一种标准化的组件开发方式。webcomponents提出之际引发了整个前端圈躁动，大家似乎在跨框架的组件化方案上看到了曙光。但是前端这圈子发展实在太特么快了，在当前个时间点，webcomponents也遭遇到了跟Object.observe相似的尴尬处境。我们先来看看webcomponents的几个核心特性：

1. Shadow DOM
2. Custom Element
3. Template Element
4. HTML Imports

其中1、4现在都能很容易的通过自动化的工程手段解决了(shadow dom对应的是scoped css)，而自义标签这种事情不论是React还是Angular这类组件框架都能轻松解决，那么我用你webcomponent的理由呢?

另外webcomponents将目标对准的是HTML体系下的组件化，这一点跟React比就相对狭隘了(但是并不表明React把战线拉的那么长就不会有问题)。

不过原生支持的跨框架的组件还是有存在的意义的，比如基础组件库，只是在当前来看web components发展还是有点营养不良。期待2016年能有实质上的突破吧。

架构的变化

2015年出现的新的技术及思路，影响最大的就是技术选型及架构了。我们可以从下面几点来看看它对端架构上都有哪些影响。

组件化

React的风靡使得组件化的开发模式越来越被广大开发者关注。首先要肯定的是，组件化是一个非常得去做的事情，它在工程上会大大提升项目的可维护性及拓展性，同时会带来一些代码可复用的附加果。但这里要强调的一点是，组件化的指导策略一定是分治(分而治之)而不是复用，分治的目的是为使得组件之间解耦跟正交，从而提高可维护性及多人协同开发效率。如果以复用为指导原则那么组件后一定会发展到一个配置繁杂代码臃肿的状态。如果以组件的形态划分，可以分为两个类型：基础控和业务组件。基础控件不应包含业务逻辑不然达不到拿来即用的效果，因此它也会表现出可复用的价，但是根本还是为了提高业务组件的可维护性。至于业务组件，可复用的价值就很低了。

1. **组件化指的是什么**

组件化这个词，在UI这一层通常指“标签化”，也就是把大块的业务界面，拆分成若干小块，然后进组装。

_狭义的组件化一般是指标签化，也就是以自定义标签（自定义属性）为核心的机制。_这也是我们通认识的组件。

_广义的组件化包括对数据逻辑层业务梳理，形成不同层级的能力封装。_它不一定是一个自定义语义签：它可以是一个包含逻辑(js)、样式(css)、模版(html)的功能完备的结构单元，也就是我们常“口口传”的模块(从术语准确性的角度来看模块这个描述并不合适，应该称之为组件)；它也可以是一个单的js，比如http组件这种纯逻辑单元。严格从概念上来讲，css跟html是不具备单独/组合成一个组件，它们不具备描述逻辑的能力(非图灵完备)。从这个层面来看，全组件化是没有任何问题及疑义的。

2. **是否需要全组件化**

我们通常说的组件指的是狭义上的组件，而且往往我们理解的全组件化也是建立在狭义的组件基础上，

代表框架是React。React + Flux体系下，它提倡尽可能将页面作细粒度的组件拆分，组件的数据全部父级组件通过props传递而来。这本身是一件非常有价值的事情，能有效的确保应用状态的稳定及可测性，但是应用一旦复杂庞大起来，组件树变得“枝繁叶茂”导致叶子节点层级过深，当出现数据问时，我们必须一层层的回溯来定位bug。而且组件树过于庞大也会增加组件之间的通讯负担。从狭义组件来看，我对全组件化是存怀疑态度的，工程上的成本太高是最大的问题，而且大部分开发者很难捏合适的组件粒度，容易出现过细 / 过粗的拆分。很多场景其实并不适合实现成狭义上的组件，它以散的模板的方式存在更合适。

但是如果从广义的组件来看，全组件的意义是很大的，我们需要通过拆分页面逻辑区块的方式实现程的解耦，从而提升应用的可维护性。

综合来看，我觉得工程上更具可行的全组件化方案应该是：细粒度的基础组件库 + 粗粒度的模板/组。

工程化

工程化是近年前端提到最多的问题之一，而且个人认为是当前前端发展阶段最有价值的问题，也是前开发通往工业化时代的必经之路。这里不赘述，有兴趣的同学看我前阵子整理的一篇文章前端工程化识点回顾

应用架构层 MVVM & Flux

MVVM想必大部分前端都耳熟能详了，代表框架是angular、vue、avalon。angular在1.2版本之后入了controllerAs语法，使得controller可以变成一个真正意义上的VM，angular整个架构也才真正称之为严格的MVVM(之前只能说是带有双向绑定的MVC/MVP)。

Flux是facebook随React一并推出的新的(准确来说其实是改进的，并非原创)架构模型，核心概念是向数据流。Flux实质上就是一个演进版的中介者模式，不同的是它同时包装了action、store、dispatc

er、view等概念。关于Flux对应用分层、数据在不同层之间只能单向流转的方式我是很赞成的。应用分层在业务稍复杂的应用中都是很有必要的，它更利于应用的伸缩及拓展，副作用是会带来一定的复度(在我看来这点复杂度根本就可以忽略不计)。

今年被黑的最多的前端主流框架莫过于angular了。老实讲前端圈真的挺善变的，去年各种大会都在享angular黑jquery，今年就变成了都在分享react黑angular了。黑的点大致有三：

1. angular的部分实现太low
2. 太多Java身上带来的臭毛病(并没有在黑Java)
3. mvvm自身的缺陷

第一点第二点我并无异议。angular的脏值检测机制相对于其他mvvm框架的双向绑定实现方式确实太优雅，同样有硬伤的还有失败的模块语法及过多过于复杂的概念。但是对于第三点，我有不同的看。

大多数人黑mvvm会以Facebook那张经典的flux vs mvc的图为论据，对于双向绑定造成的数据流紊及应用状态的不确定导致问题定位困难的观点我是认同的，这一点我也有切身体会，但是单纯的这一就足以否定mvvm么？就说flux比mvvm高明？

MVVM在富表格型(自造的词□smile)应用
发效率上是高于Flux的，典型的就是一些后台管控平台。而且最重要的是，MVVM跟Flux并不互斥，们在MVVM中照样可以引入Flux中的一些机制从而确保应用状态的稳定。很多时候我们对于框架 / 架的孰优孰劣的争论是没意义的，抛开业务场景谈解决方案都是耍流氓。

业务数据层 Relay & falcor

这一层对大部分前端来说可能是比较新的概念，其实我们可以这样理解：在一个完整的应用中，业务据层指的就是数据来源，在angular体系中可以等同于ngResource模块(准确来说应该是$http)。

Relay是f家推出的在react上应用GraphQL的框架，它的大致思路是：前端通过在应用中定义一系列的chema来声明需要的接口数据结构，后端配合GraphQL引擎返回相应的数据。整个事情对于前端来意义简直是跨时代的，工业化典范！不仅能极大提升前后端协同的开发效率，还能增加前端对于应用整的掌控力。但是目前来看问题就是实施过于复杂，而且还得后端服务支持，工程成本太高，这一点Meteor显然做的更好。

falcor则是Netflix出品的一个数据拉取库，核心理念是单一数据源，跟Redux的单store概念一致。用跟Realy类似，也需要前端定义数据schema。

另外还有一个新的W3C标准api：fetch，它的级别等同于XMLHttpRequest，旨在提供比ajax更优雅资源获取方式，目前几个主流浏览器支持的都还不错，也有官方维护的polyfill，几乎可以确定是未来主流数据请求api。

业务数据层是前端应用中比较新的概念，它的多元化主要会影响到应用的架构设计，这里不细讲后面来说。

新的编程范式

函数式编程(FP)

函数式编程(functional programming)是近年比较火爆的一个编程范式，FP基于lambda演算，与以灵机为基础的指令式编程(Java、C++)有着明显的差异。lambda演算更关注输入输出，更符合自然为场景，所以看上去更适合事件驱动的web体系，这点我也认同。但问题是，太多开发者看到redux么火爆就急着学redux用js去玩函数式，我觉得这个有待商榷。js作为一个以基于函数(scheme，父亲跟基于对象(Self，母亲)的编程语言为蓝本设计然后语法又靠近Java(隔壁老王)的“混血”语言，你非用它去写函数式，是不是过于一厢情愿？尤其是在现在浏览器还不支持尾调用优化的情况下，你让那

增的调用栈可如何是好joy如果你确实钟情于函数式，
以去玩玩那些更functional的语言(Haskell、Clojure等)，而不是从js入手。最近看到一个老外关于js函数式编程的看法，最后一句总结很精辟：Never forget that javascript hate you.
oy

函数式响应型编程(FRP)

函数式响应型编程(functional reactive programming)不是一个新概念，但也不过是近两年才引入前端领域的，代表类库就是ng2在用的rxjs。FRP关注的是事件及对应的数据流，你可以把它看作是一基于事件总线(event bus)的观察者模式，它主要适用于以GUI为核心的交互软件中。但FRP最大的困之处在于，如果你想使用这样的编程范式，那么你的整个系统必须以reactive为中心来规划。目前微维护的ReactiveX项目已经有各种语言的实现版本，有兴趣的同学可以去了解下。

工具链的变化

去年最主流的前端构建工具还是grunt&gulp，2015年随着react的崛起和web标准的快速推进，一又有了新的变化。

webpack & browserify & jspm

webpack跟browserify本质上都是module bundler，差异点在于webpack提供更强大的loader机制其更变得更加灵活。当然，webpack的流行自然还是离不开背后的react跟facebook(可见有个强大的爹多么重要)。但是从现在HTTP/2标准的应用及实施进展来看，webpack/browserify这种基于bundl的打包工具也面临着被历史车轮碾过的危机，相对的基于module loader的jspm反而更具前景(虽然在使用前两者的开发者都多于jspm)。

PostCSS & cssnext

PostCSS作为新一代的css处理器大有取Sass/Less而代之的趋势，Bootstrap v5也有着基于PostCSS开发的计划。但从本质来讲它又不算一个处理器，它更像是一个插件平台，能通过配置各种插件从而现预处理器跟后处理器的效果。

cssnext官方口号是“使用来自未来的语法开发css，就在今天！”，但是cssnext又不是css4，它是个能让开发者现在就享受最新的css语法(包括自定义属性、css变量等)的转换工具。这一块笔者还没过具体实践，暂不多言。

写在最后

从前端的发展现状来看，未来理想的前端技术架构应该是每一层都是可组装的，框架这种重型组合的用场景会越来越局限。原因在于各部件不可拆卸会增加架构的升级代价同时会限制应用的灵活性。举例子，我有一套面向pc端的后台管控平台的架构，view层采用angular开发，哪天我要迁移到移动端，angular性能不行啊，我换成vue就好了。哪天觉得ajax的写法太挫，把http层替换成fetch就好了又有一天后端的GranphQL平台搭好了，我把ngResource换成relay就OK了。

这种理想的方式当然是完全正确的方向，但是目前来看它对 开发者/架构师 的要求还是太高，工业级上一套带有约束性的框架还是有相当的需求的(特别是当团队开发者的水平良莠不齐时。当然我觉得更确的方式是流程上有一套完整的自动化方案用于确保团队提交的代码质量，只是目前基于动态分析的码质量检测工具还没有出现，而且估计很长一段时间内都不会有)。虽然美好但是组合的方式也不是没问题，各种五花八门的搭配容易造成社区的分化跟内耗，一定程度上不利于整个生态圈的发展。

近年前端生态的野蛮发展影响最大的应该就是新产品的技术选型了，乱花迷人眼，我们很难设计出一适应大部分场景、而且短时间内不会被淘汰的架构。前端的变化太快通常会导致一些技术决策的反复

今天的最佳实践很可能明天就被视为反模式。难道最合适的态度是各种保留各种观望，以不变应万变在这一点上即使如我这般在技术上一向激进的人都有点畏手畏脚了。那句话怎么说的来着？从来没有个圈子像今天的前端一样混乱又欣欣向荣了。有人说2015年或许是大前端时代的元年，目前看来，如不是2015，那么它也一定会是2016年。

最后引用计子winter的一句话作为结语吧：

> 前端一直是一个变化很快的职能，它太年轻，年轻意味着可能性和机会，也意味着不成熟和痛苦。我常担心的事情就是，很可能走到最后，我们会发现，我们做了很多，却还是一无所获。所幸至今回顾每年还是总有点不同，也算给行业贡献了些经验值吧。

原文链接：2015 前端生态发展回顾