前端之路总结

笔者从jQuery时代一路走来，经历了以BootStrap为代表的基于jQuery的插件式框架与CSS框架的兴起，到后面以Angular 1为代表的MVVM框架，以及到现在以React为代表的组件式框架的兴起。从最初的认为前端就是切页面，加上一些交互特效，到后面形成一个完整的webapp，总体的变革上，笔者以为有以下几点：

- 移动优先与响应式开发
- 前端组件化与工程化的变革
- 从直接操作Dom节点转向以状态/数据流为中心

笔者在本文中的叙事方式是按照自己的认知过程，夹杂了大量个人主观的感受，看看就好，不一定要当真，毕竟我菜。梳理来说，有以下几条线：

- 交互角度的从PC端为中心到Mobile First
- 架构角度的从以DOM为中心到MVVM/MVP到以数据/状态为驱动。
- 工程角度的从随意化到模块化到组件化。
- 工具角度的从人工到Grunt/Gulp到Webpack/Browserify。

React 并没有提供很多复杂的概念与繁琐的API，而是以最少化为目标，专注于提供清晰简洁而抽象的视图层解决方案，同时对于复杂的应用场景提供了灵活的扩展方案，典型的譬如根据不同的应用需求引入MobX/Redux这样的状态管理工具。React在保证较好的扩展性、对于进阶研究学习所需要的基础知识完备度以及整个应用分层可测试性方面更胜一筹。不过很多人对React的意见在于其陡峭的学习曲线与较高的上手门槛，特别是JSX以及大量的ES6语法的引入使得很多的传统的习惯了jQuery语法的前端开发者感觉学习成本可能会大于开发成本。

Vue则是典型的所谓渐进式库，即可以按需渐进地引入各种依赖，学习相关地语法知识。比较直观的感受是我们可以在项目初期直接从CDN中下载Vue库，使用熟悉的脚本方式插入到HTML中，然后直接在script标签中使用Vue来渲染数据。随着时间的推移与项目复杂度的增加，我们可以逐步引入路由、状态管理、HTTP请求抽象以及可以在最后引入整体打包工具。这种渐进式的特点允许我们可以根据项目的复杂度而自由搭配不同的解决方案，譬如在典型的活动页中，使用Vue能够兼具开发速度与高性能的优势。不过这种自由也是有利有弊，所谓磨刀不误砍材工，React相对较严格的规范对团队内部的代码样式风格的统一、代码质量保障等会有很好的加成。

一言蔽之，笔者个人觉得Vue会更容易被纯粹的前端开发者的接受，毕竟从直接以HTML布局与jQuery进行数据操作切换到指令式的支持双向数据绑定的Vue代价会更小一点，特别是对现有代码库的改造需求更少，重构代价更低。而React及其相对严格的规范可能会更容易被后端转来的开发者接受，可能在初学的时候会被一大堆概念弄混，但是熟练之后这种严谨的组件类与成员变量/方法的操作会更顺手一点。便如Dan Abramov所述，Facebook推出React的初衷是为了能够在他们数以百计的跨平台子产品持续的迭代中保证组件的一致性与可复用性。

前后端分离与全栈:技术与人

我们常说的前后端分离会包含以下两个层面：

- 将原本由服务端负责的数据渲染工作交由前端进行，并且规定前端与服务端之间只能通过标准化协议进行通信。
- 组织架构上的分离，由早期的服务端开发人员顺手去写个界面转变为完整的前端团队构建工程化的前端架构。

前后端分离本质上是前端与后端适用不同的技术选型与项目架构，不过二者很多思想上也是可以融会贯通，譬如无论是响应式编程还是函数式编程等等思想在前后端皆有体现。而全栈则无论从技术还是组织架构的划分上似乎又回到了按照需求分割的状态。不过呢，我们必须要面对现实，很大程度的工程师并没有能力做到全栈，这一点不在于具体的代码技术，而是对于前后端各自的理解，对于系统业务逻辑的理解。如果我们分配给一个完整的业务块，同时，那么最终得到的是无数个碎片化相互独立的系统。

何谓工程化

所谓工程化，即是面向某个产品需求的技术架构与项目组织，工程化的根本目标即是以尽可能快的速度实现可信赖的产品。尽可能短的时间包括开发速度、部署速度与重构速度，而可信赖又在于产品的可测试性、可变性以及Bug的重现与定位。

- 开发速度：开发速度是最为直观、明显的工程化衡量指标，也是其他部门与程序员、程序员之间的核心矛盾。绝大部分优秀的工程化方案首要解决的就是开发速度，不过笔者一直也会强调一句话，磨刀不误砍材工，我们在追寻局部速度最快的同时不能忽略整体最优，初期单纯的追求速度而带来的技术负债会为以后阶段造成不可弥补的损害。
- 部署速度：笔者在日常工作中，最长对测试或者产品经理说的一句话就是，我本地改好了，还没有推送到线上测试环境呢。在DevOps概念深入人心，各种CI工具流行的今天，自动化编译与部署帮我们省去了很多的麻烦。但是部署速度仍然是不可忽视的重要衡量指标，特别是以NPM为代表的难以捉摸的包管理工具与不知道什么时候会抽个风的服务器都会对我们的编译部署过程造成很大的威胁，往往项目依赖数目的增多、结构划分的混乱也会加大部署速度的不可控性。
- 重构速度：听产品经理说我们的需求又要变了，听技术Leader说最近又出了新的技术栈，甩现在的十万八千里。
- 可测试性：现在很多团队都会提倡测试驱动开发，这对于提升代码质量有非常重要的意义。而工程方案的选项也会对代码的可测试性造成很大的影响，可能没有无法测试的代码，但是我们要尽量减少代码的测试代价，鼓励程序员能够更加积极地主动地写测试代码。
- 可变性：程序员说：这个需求没法改啊！
- Bug的重现与定位：没有不出Bug的程序，特别是在初期需求不明确的情况下，Bug的出现是必然而无法避免的，优秀的工程化方案应该考虑如何能更快速地辅助程序员定位Bug。

当我们探究工程化的具体实现方案时，在技术架构上，我们会关注于：

- 功能的模块化与界面的组件化
- 统一的开发规范与代码样式风格，能够在遵循SRP单一职责原则的前提下以最少的代码实现所需要的功能，即保证合理的关注点分离。
- 代码的可测试性
- 方便共享的代码库与依赖管理工具
- 持续集成与部署
- 项目的线上质量保障

前端的工程化需求

当我们基于前端时，笔者在历年的实践中感受到以下几个突出的问题：

当我们落地到前端时，笔者在历年的实践中感受到以下几个突出的问题：

- 前后端业务逻辑衔接：在前后端分离的情况下，前后端是各成体系与团队，那么前后端的沟通也就成了项目开发中的主要矛盾之一。前端在开发的时候往往是根据界面来划分模块，命名变量，而后端是习惯根据抽象的业务逻辑来划分模块，根据数据库定义来命名变量。最简单而是最常见的问题譬如二者可能对于同意义的变量命名不同，并且考虑到业务需求的经常变更，后台接口也会发生频繁变动。此时就需要前端能够建立专门的接口层对上屏蔽这种变化，保证界面层的稳定性。
- 多业务系统的组件复用：当我们面临新的开发需求，或者具有多个业务系统时，我们希望能够尽量复用已有代码，不仅是为了提高开发效率，还是为了能够保证公司内部应用风格的一致性。
- 多平台适配与代码复用：在移动化浪潮面前，我们的应用不仅需要考虑到PC端的支持，还需要考虑微信小程序、微信内H5、WAP、ReactNative、Weex、Cordova等等平台内的支持。这里我们希望能够尽量的复用代码来保证开发速度与重构速度，这里需要强调的是，笔者觉得移动端和PC端本身是不同的设计风格，笔者不赞同过多的考虑所谓的响应式开发来复用界面组件，更多的应该是着眼于逻辑代码的复用，虽然这样不可避免的会影响效率。鱼与熊掌，不可兼得，这一点需要因地制宜，也是不能一概而论。

关于跨界、全栈、公司定岗

常规公司里，会配备好前端开发、iOS、Android、服务端开发这四种技术团队，实际做项目时，几支团队是分工合作，只在必要的地方通过接口配合。在PC时代，不考虑C/S结构软件，只存在前端和服务端。前端和服务端配合时，主要是通过前端提供模板，后端负责数据持久化和逻辑处理来合作的，双方唯一可能起争执的地方就是模板引擎由谁来套，这个模板引擎指的是服务端模板引擎，大部分公司里，这个模板是由服务端来套的。虽然也有AJAX请求，服务端吐JSON数据的地方，但总体来说并不是那么多这种接口，渲染也只是局部渲染，一般到不了使用JavaScript模板引擎的地步。应该说，这个时期前后分离的需求并不高，而SEO的需求又很重，所以前后端两支团队也算泾渭分明井水不犯河水。

关于前端的核心竞争力

如果说服务端同学进击全栈是试试水，Native进击全栈是试试水，那前端里很多同学进击全栈就是在拿生命在玩全栈了。

服务端玩玩Node，不喜欢就算了，玩玩Angular和Bootstrap也就在后台开开荤，前台各位视觉设计，UAT还原检查，各种动效，用Angular和Bootstrap能把自己玩死，而后台基本上一直是服务端的自留地，很多做前端开发的同学甚至没开发过后台界面吧？Django甚至都自动给你生成了。后端的核心竞争力在哪儿？在添删改查，在数据库设计，在性能优化，在shell脚本，在分布式，在网络安全。玩玩票不影响自己的大本营。

同样可以一门语言前后台通吃的Android开发，你看看他们对全栈是不是像前端圈那样热情。从DNA来看，对Java语言可比JavaScript和Node更亲吧？再往远了说，看看C++客户端的同学对服务端有没有那么大热情？

还是那句话，好学是好的，前提是自己的大本营要守住，一专多长。你得先专一门，再想着横向扩展其他领域知识。前端开发的核心竞争力是什么？2016年年中，我在微博上说，前端的核心竞争力在于一些HTML标签、CSS，JavaScript的熟练度上。这些东西是前端自己领域的知识，比如Form2.0、Websocket、离线缓存、Webworker、Border-image、Canvas。一些同学回复说“核心竞争力居然只是些API，这有什么难度？”此言差矣。或许这样认为的，以跨界而来的“全栈”工程师居多。有些知识确实只是API，比如JSON.stringify和window.getComputedStyle之类，看了就会用，用起来也没有什么实践方面的坑。但并不全是，比如Form2.0可是有一系列新东西，新标签如output，新类型如number，新属性如pattern，新的CSS伪类如:valid，需要融合在一起考虑，形成一个Form2.0的解决方案。再比如Canvas2d，Canvas提供了像素级API，可以直接存取颜色，可以把像素导出成base64的字符串，提供了DOM没有能力，但同时也完全没有了DOM的便利，比如Canvas上画的某个按钮该如何进行事件监听呢？比如不能使用CSS了，该如何实现:hover伪类，又如何让布局实现自适应性呢？什么样的情况下该使用Canvas，什么情况下该使用DOM，如果有某个功能必须依赖Canvas实现，比如在网页上做个美图秀秀，将产品的哪些元素放到Canvas上，哪些元素放到DOM上，两者又如何合作呢？换成纯Canvas解决方案会不会更适合呢？前端的知识不同于服务端，大部分的工作量都在图形界面上，而图形界面是件很细的活，工作量和技术含量全在细节。我经常对非前端的同学举一个例子——你知道垂直居中有几种方法，不同方法的优缺点吗？有些跨界而来的同学，以及部分前端圈的新同学都不以为然，嘲笑说这叫“回字的四种写法”。其实，前端在实战时，垂直居中有多种方法，基本上没有方法是无副作用的，要看情况，不同的情况要选用不同的方式才能实现最好的自适应性。感兴趣的同学可以去搜搜看前端的垂直居中方法整理。看完之后，就能明白前端CSS的精彩和玄妙。

我批评很多同学基础不扎实就开始乱折腾，不是说多学习不好，而是说大本营都未扎牢，如何实打实地高效完成日常工作？ES6、CoffeeScript、React、Webpack等，都解决不了你在实战时遇到的具体挑战。这全是些外围功夫，并非核心。先把核心学好，外围功夫什么时候学都可以，又不难，你说对吧？

那么什么是核心呢？HTML、CSS和JavaScript。我指的是原生的这些东西，不用上来就跟我说React的JavaScriptx语法重定义了HTML，Sass改良了CSS，TypeScript给JavaScript带来了静态语言的语法，这些都是外围，今天是React，明天可以换成Angular，今天是Sass明天可以换成Less，今天是TypeScript明天可以是CoffeeScript，这些不重要。就像jQuery鼎盛时期，很多同学不学原生JavaScript，上来直接就上手jQuery一样，走不远。要理解jQuery为什么这么封装，其实在底层发生了什么，用原生会遇到什么问题，直接用原生能解决吗？把原生的技巧学熟了，这些外围的东西上手很快，而且什么情况下用什么，心里会非常有底。

过去，前端领域并不像如今这样浮躁，很多人都知道基础的重要性，也知道基础是什么。但当跨界的“全栈”进入前端圈以后，很多浅显的道理都被有意无意地搅昏了。速成、革命、淘汰、全栈成了主旋律和政治正确。可是，就像投资界里爱说的一句话一样：“风起来了，在风口上猪都会飞。可是等风停了，还在飞的是老鹰，而猪会摔死。”风会停吗？当然。该潜心修炼还得修炼，基本功不扎实以前，别糊里糊涂跟风浪费自己时间，缺什么要恶补。

今天说得很热闹的HTML5其实是从HTML4加强而来，两者不是替换关系，而是“强化”，就像ES6之于ES3一样。很多新入行的同学希望可以速成，然后从哪儿热门往哪儿入手，这其实不对，最好的学习方法是从HTML4学起，尽管在实践时有很多HTML4时代的技巧，在HTML5时代有了更好的替换方案，但也有很多HTML4时代可以一直用过来的技巧。让我担心的是，HTML4时代的好书，到了HTML5时代已经不再出版了，而HTML5相关的书籍基本上只讲了HTML5相较于HTML4的增量部分。而HTML4时代的书和相关技巧就这么失传了。除了书，博客和所谓社区也是一样，现在已经不再讨论以前的一些精华技巧了，有些技巧确实是淘汰掉了没有什么价值，比如IE6的hack技术，但也有些技术是很棒的CSS技巧，比如CSS滑动门依然适用。我推荐一下几本书和学习步骤，给有心弥补基本功的同学：

- 《CSS网站布局实录》——国产CSS2入门书，有些技巧已经淘汰，但仍不失为最好的CSS入门教程。
- 《无懈可击的Web设计》——讲CSS应用技巧的书，国内外粉丝别多，说是开创了CSS技巧流派也不为过。
- 《DOM JavaScript编程艺术——JavaScript最好的入门书，没有之一，这本书是帮助你了解如何将DOM、CSS和JavaScript连接起来的一本书。严格来说，后端Node根本不算JavaScript，JavaScript是基于ES语法的一门脱水语言，如何实现的胶水？这本书将带你入门。
- 《JavaScript高级程序设计》 ——JavaScript必读的一本精典，读完之后对JavaScript的理解和实践会上升非常大的一个台阶。
- 《编写高质量代码——Web前端开发修炼之道》 —— 举贤不避亲，这本书是我写的。推荐的原因是，这本书重点讲团队合作的注意事项。虽然一些具体的技巧，在今天已然过时，比如IE6的hack，但在团队合作方面的思考，直到今天我也没看到其他书在讲，这些思想没有其他书可替代。
- 《HTML5和CSS3权威指南》——目前为止，我读过的HTML5方面最好的一本原创书。配合实例进行API讲解，非常详细具体。连HTML5都提供了哪些底层的东西都不知道，又该如何去用好它呢？在我看来，是学习HTML5的必读书。
- 《响应式Web设计：HTML5和CSS3实战》——作者是《无懈可击的Web设计》忠实粉丝，所以很自然地，这也是本CSS技巧流派的书，侧重点在CSS3的实践技巧上，让人大开眼界。
- 《JavaScript设计模式》——JavaScript在实战时的高级技巧。

前端很棒的书有很多，这只是几本我觉得最不该错过的书而已。从HTML4一路到HTML5和移动时代，一路上有了很多新技巧，也淘汰了一些旧技巧。当下的学习氛围虽然前所未有的强

烈，但急功近利和盲目无头绪现象也很严重。在我看来，很多人不愿意做苦活累活扎扎实实打基本功，一句“那些都淘汰了”就拒绝了所有的优秀遗产，希望花少量时间看看流行时髦的新工具新框架，然后就迅速挤身行业顶端，这想法既偷懒又幼稚。什么是外围功夫，什么是真核心技巧，什么是珍珠什么是盒子要分得清，自欺欺人并不是什么明智的想法。你可以几天几个星期就掌握的东西，别人也可以，就算人家比你笨，多花一倍的时间也能跟上你吧？要真的拉开和其他人的距离，只有下苦功这一途。

这些话，恐怕没有几位前端老人愿意说。当我问他们，拼命强调新风向，而不再提基本功，造成知识断层，造成这些同学心高气傲但完成不了工作怎么办时，一些老人的回答是“他们自己不重视基本功，怪我喽”。如果你基本功很扎实了，想学什么外围功夫都可以，虽然多学总不是坏事，只是在决定投入使用时，还需要看团队情况再慎重决定，团队合作要考虑的事情有很多，要有责任感，别只顾着自己当Geek。

1：我们可能在任何情况下都需要**声明式的渲染功能**，并希望尽可能避免手动操作，或者说是可变的**命令式操作**，希望尽可能地让DOM的更新操作是自动的，状态变化的时候它就应该自动更新到正确的状态；

2：我们需要**组件系统**，将一个大型的界面切分成一个一个更小的可控单元；

3：客户端路由——这是针对单页应用而言，不做就不需要，如果需要做单页应用，那么就需要有一个URL对应到一个应用的状态，就需要有路由解决方案；

4：大规模的状态管理——当应用简单的时候，可能一个很基础的状态和界面映射可以解决问题，但是当应用变得很大，涉及多人协作的时候，就会涉及多个组件之间的共享、多个组件需要去改动同一份状态，以及如何使得这样大规模应用依然能够高效运行，这就涉及大规模状态管理的问题，当然也涉及到可维护性，

5：还有**构建工具**。现在，如果放眼前端的未来，当HTTP2普及后，可能会带来构建工具的一次革命。但就目前而言，尤其是在中国的网络环境下，打包和工程构建依然是非常重要且不可避免的一个环节。