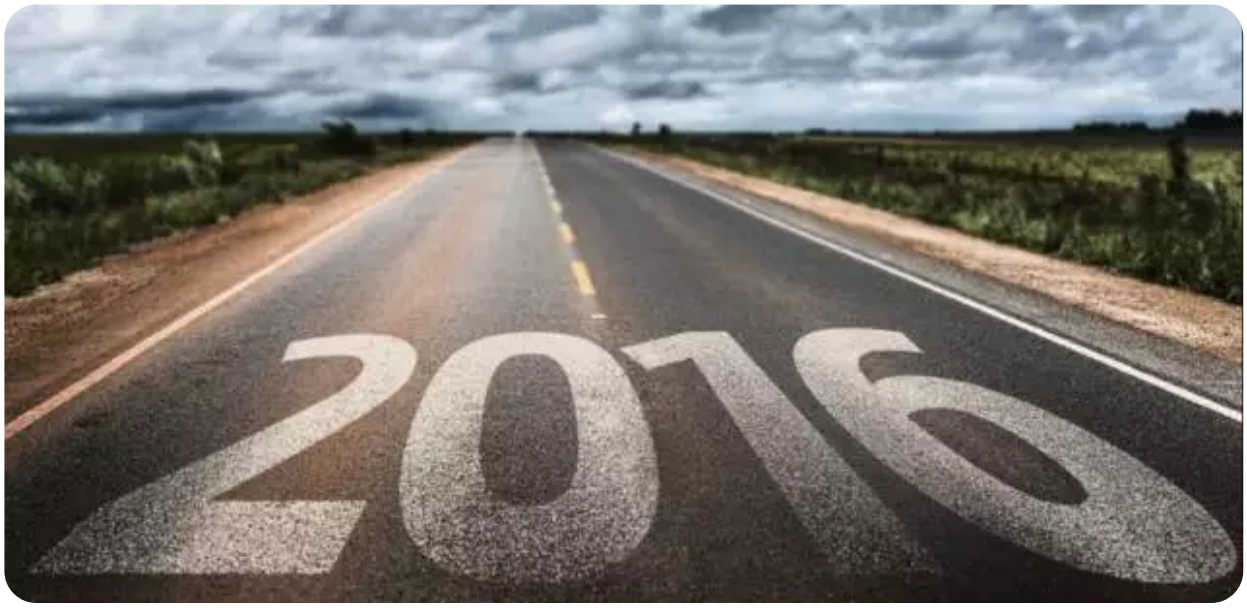
2016前端开发技术巡礼

Original 2016-12-30 殷勇 前端之巅



作者：殷勇

编辑：尾尾

目录

提示：点击文末阅读原文，可查看本文带链接版。

前言

2016 年马上过去了，像过去六年中的每一年一样，Web前端领域又产生了“面目全非”而又“耳目一新”的变化，不但旧事物持续不断地被淘汰，新事物也难保坐久江山，大有岌岌可危之势。开源界如群雄逐鹿，不断生产新的概念、新的框架、新的工具，去年中一些流行的技术今年大多得到了进一步的演进和升级，活跃度非常高，却仍然不能保证前端的未来属于它们。在今年整体资本市场冷却的大环境下，*to B*的创业公司显现出了较强的生命力，这种类型的业务也给Web前端的工作带来了明显的差异性，工程师整体技能方向也展露出一丝不一样的分支。

一、更新的网络与软件环境

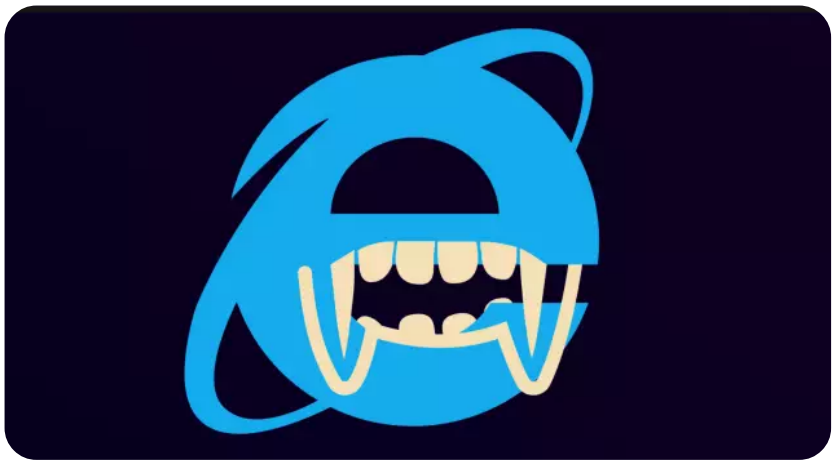
1.1 HTTP/2 的持续普及

今年中，几乎所有的现代桌面浏览器都已经支持了HTTP/2协议，移动端依靠降级为SPDY依旧可以覆盖几乎所有平台，这样使得从协议上优化页面的性能成为了可能。

同时，前端静态资源打包的必要性成为了一定程度上的争论焦点，打包合并作为传统的前端性能优化方案，它的存留对前端工程化影响极大，Facebook公司著名的静态资源动态打包方案的优越性也会被弱化。社区上多篇文章纷纷发表对HTTP/2的性能实验数据，却不尽相同。

在2017年，我相信所有大型站点都会切换**HTTP/2**，但依旧不会放弃对静态资源打包合并的依赖。而且，对于**Server Push**等高级特性，也不会有太多的应用。

1.2 Internet Explorer 8



三年前还在考虑兼容IE6的前端技术社区，在前不久天猫宣布不再支持IE8后又引起了一股躁动。IE8是Windows XP操作系统支持的最高IE版本，放弃IE8意味着放弃了使用IE的所有XP用户。

其实在2016年的今天，前端社区中框架、工具的发展早已不允许IE8的存在，Angular 早在1.3版本就果断放弃了IE8，React 也在年初的v15版本上宣布放弃。在PC领域，你依旧可以使用像Backbone.js一样的其他框架继续对IE进行支持，但无论是从研发效率上还是从运行时效率上，放弃它都是更好的选择。

由于对HTML5兼容性不佳，在2017年，相信**IE9**也会逐渐被社区放弃，以取得更好的性能、更少的代码体积。

二、如何编写(Java)Script

2.1 ES2016？ES2017？Babel!

去年定稿的ES2015（亦称ES6）带来了大量令人激动的新语言特性，并快速被V8和SpiderMonkey所实现。但由于浏览器版本碎片化问题，目前编写生产环境代码仍然以ES5为主。今年年中发布的ES2017带来的新特性数量少的可怜，但这正好给了浏览器厂商消化ES2015的时间，在ES2017到来之前喘口气——是的，明年的**ES2017**势必又会带来一大波新特性。

JS解释引擎对新特性的支持程度并不能阻碍狂热的开发者使用他们，在接下来的很长时间，业界对Babel的依赖必然有增无减。Babel生态对下一代ECMAScript的影响会进一步加大，人们通过先增加新的Babel-plugin，后向ECMA提案的方式成为了ECMAScript进化的常态。开发者编写的代码能直接运行在浏览器上的会越来越少。

但使用Babel导致的编译后代码体积增大的问题并没有被特别关注，由于polyfill可能被重复引入，部署到生产环境的代码带有相当一部分冗余。

2.2 TypeScript

作为ECMAScript语言的超集，TypeScript在今年取得了优异的成绩，Angular 2放弃了传说中的AtScript，成为了TypeScript的最大客户。人们可以像编写Java一样编写JavaScript，有效提升了代码的表述性和类型安全性。

但凡事有两面，TypeScript的特性也在不断升级，在生产环境中，你可能需要一套规范来约束开发者，防止滥用导致的不兼容，这反而增加了学习成本、应用复杂性和升级安全性。个中优劣，仍需有大量的工程实践去积累经验。

此外，TypeScript也可以看做一种转译器，与Babel有着类似的新特性支持。在2017年，我们期待TypeScript与Babel会发展成怎样的一种微妙关系。

2.3 promise、generator 与 async/await

在回调地狱问题上，近两年我们不断被新的方案乱花了眼。过去我们会利用async来简化异步流的设计，直到“正房”Promise的到来。但它们只是callback模式的语法糖，并没有完全消除callback的使用。

ES2015带来的generator/yield似乎成为了解决异步编程的一大法宝，虽然它并非为解决异步编程所设计的。但generaor的运行是十分繁琐的，因此另一个工具co又成为了使用generator的必备之

选。Node.js社区的Koa框架初始就设计为使用generator编写洋葱皮一样的控制流。

但昙花一现，转眼间async/await的语法，配合Promise编写异步代码的方式立即席卷整个前端社区，虽然async/await仍然在ES2017的草案中，但在今天，不写async/await立刻显得你的设计落后社区平均水平一大截。

在Node.js上，v7已经支持在harmony参数下的async/await直接解释，在明年4月份的v8中，将会正式支持，届时，Koa 2的正式版也会发布，几乎完全摒弃了generator。

2.4 fetch

受到回调问题的影响，传统的 `XMLHttpRequest` 有被 `fetch` API 取代之势。如今，成熟的polyfill如whatwg-fetch、node-fetch、isomorphic-fetch在npm上的每日下载量都非常大，即便对于兼容性不好的移动端，开发者也不愿使用繁琐的AJAX。借助async/await的语法，使用fetch API能让代码更简洁。

三、Node.js服务与工具

Koa 2

Koa与流行的Express属于“同根生”的关系，它们由同一团队打造。相比Express，新的Koa框架更轻量、更灵活。但Koa的设计在短时间内曾经出现了较大的变动，这主要受到了async/await语法对异步编程的影响。在v2版本中，Koa的middleware抛弃generator转而支持async，所有第三方middleware实现，要么自行升级，要么使用Koa-convert进行包装转换。

目前Koa在Node.js社区的HTTP服务端框架中受到关注度比较高，不过其在npm上latest目前仍处于1.x阶段，预计在2017年4月份发布Node.js v8后，就会升级到2.x。

Koa的轻量级设计意味着你需要大量第三方中间件去实现一个完整的Web应用，目前鲜有看到对Koa的大规模重度使用，因此也就对其无从评价。相信在明年，越来越多的产品应该会尝试部署Koa 2，届时，对第三方资源的依赖冲突也会尖锐起来，这需要一个过程才能让Koa的生态完备起来。预计在2018年，我们会得到一个足够健壮的Koa技术栈。这会促进Node.js在服务端领域的扩展，轻量级的Web服务将会逐渐成为市场上的主流。

四、框架纷争

4.1 jQuery已死?

今年六月份jQuery发布了3.0版本，距离2.0发布已经有三年多的时间，但重大的更新几乎没有。由于老旧浏览器的逐渐放弃和升级，jQuery需要处理的浏览器兼容性问题越来越少，专注于API易用性和效率越来越多。

随着如Angular、React、Ember、Vue.js等大量具备视图数据单双向绑定能力的框架被普及，使用jQuery编写指令式的代码操作DOM的人越来越少。早在2015年便有人声称jQuery已死，社区中也进行了大量雷同的讨论，今天我们看到确实jQuery的地位已大不如前，著名的sizzle选择器在今天已完全可由*querySelector*原生方法替代，操作DOM也可以由框架根据数据的变动自动完成。

明年**jQuery**在构建大型前端产品的过程中的依赖会被持续弱化，但其对浏览器特性的理解和积淀将对现有的和未来的类**Angular**的**MVVM**框架的开发依旧具有很大的借鉴意义。

4.2 Angular 2

好事多磨，Angular 2的正式版终于在今年下半年发布，相比于1.x，新的版本几乎是完全重新开发的框架，已经很难从设计中找到1.x的影子。陡峭的学习曲线也随之而来，npm、ES2015 Modules、Decorator、TypeScript、Zone.js、RxJS、JIT/AOT、E2E Test，几乎都是业界这两年中的最新概念，这着实给初学者带来了不小的困难。

Angular 2也更面向于开发单页应用（**SPA**），这是对ES2015 Modules语法描述的模块进行打包（bundle）的必然结果，因此Angular 2也更依赖于Webpack等“bundler”工具。

虽然Angular 声称支持TypeScript、ECMAScript和Dart三种语言，不过显然业界对Dart没什么太大兴趣，而对于ECMAScript和TypeScript，两种语言模式下Angular 2在API和构建流程上都有着隐式的（文档标注不明的）差异化，这必然会给开发者以困扰。加上业界第三方工具和组件的支持有限，TypeScript几乎是现在开发者唯一的选择。

此外，Angular团队已声明并没有完全放弃对1.x组件的支持，通过特有的兼容API，你可以在2.x中使用针对1.x开发的组件。鉴于不明确的风险，相信很少有团队愿意这样折腾。

现在在产品中使用Angular 2，在架构上，你需要考虑生产环境和开发环境下两种完全不同的构建模式，也就是JIT和AOT，这需要你有两套不一样的编译流程和配置文件。在不同环境下模块是否符合期望，可以用E2E、spec等方式来进行自动化测试，好的，那么Angular 2的测试API又可能成了技术壁垒，它的复杂度可能更甚Angular本身。可以确信，在业务压力的迫使下，绝大部分团队都会放弃编写测试。

总之，**Angular 2**是一个非常具有竞争力的框架，其设计非常具有前瞻性，但也由于太过复杂，很多特性都会成为鸡肋，被开发者所无视。由于React和Vue.js的竞争，Angular 2对社区的影响肯定不如其前辈1.x版本，且其更高级的特性如Server Render还没有被工程化实践，因此相信业界还会持续观望，甚至要等到下一个4.x版本的发布。

4.3 Vue.js 2.0

Vue.js 绝对是类MVVM框架中的一匹黑马，由作者一人打造，更可贵的是作者还是华人。Vue.js在社区内的影响非常之大，特别是2.0的发布，社区快速生产出了无数基于Vue.js的解决方案，这主要还是受益于其简单的接口**API**和友好的文档。可见作为提供商，产品的简单易用性显得尤为重要。在性能上，Vue.js基于ES5 Setter，得到了比Angular 1.x脏检查机制成倍的性能提升。而2.0在模块化上又更进一步，开发难度更低，维护性更好。可以说Vue.js准确地戳中了普通Web开发者的痛点。在国内，Vue.js与Weex达成了合作，期待能给社区带来怎样的惊喜。

4.4 React

目前看来，**React**似乎仍是今年最流行的数据视图层解决方案，并且几乎已经成为了每名前端工程师的标配技能。今年React除了版本从0.14直接跃升至15，放弃了IE8以外，并没有更多爆发式的发展。人们对于使用JSX语法编写Web应用已经习以为常，就像过去十年间写jQuery一样。

React的代码在维护性能上显而易见，如果JSX编写得当，在重渲染性能上也具备优势，但如果只部署在浏览器环境中，那么首屏性能将会受到负面影响，毕竟在现阶段，纯前端渲染仍然快不过后端渲染，况且后端具备天生的chunked分段输出优势。我们在业界中可以看到一些负面的案例，比如某新闻应用利用React全部改写的case，就是对React的一种误用，完全不顾其场景劣势。

围绕着React发展的替代品和配套工具依旧很活跃，preact以完全兼容的API和小巧的体积为卖点，inferno以更快的速度为卖点，等等。每个框架都想在Virtual DOM上有所创新，但它们的提升都不是革命性的，由此而带来的第三方插件不兼容性，这种风险是开发者不愿承担的，笔者认为它们最大的意义在于能为React的内部实现提供另外的思路。就像在自然界，生物多样性是十分必要的，杂交能带来珍贵的进化优势。

4.5 React-Native

今年是React-Native（以下简称RN）支持双端开发的第一年，不断有团队分享了自己在RN上的实践成果，似乎前途一片大好，RN确实有效解决了传统客户端受限于发版周期、H5受限于性能的难题，做到了鱼和熊掌兼得的理想目标。

但我们仍然需要质疑：

首先，**RN**目前以两周为周期发布新版本，没有**LTS**，每个版本向前不兼容。也就是说，你使用0.39.0的版本编写bundle代码，想运行在0.35.0的runtime上，这几乎会100%出问题。在这种情况下，如何制定客户端上RN的升级策略？如果升级，那么业务上如何针对一个以上的runtime版本编写代码？如果不升级，那么这意味着你需要自己维护一个LTS。要知道目前每个RN的版本都会有针对前版本的bug fix，相信没有团队有精力可以在一个老版本上同步这些，如果不能，那业务端面对的将是一个始终存在bug的runtime，其开发心理压力可想而知。

其次，虽然**RN**声称支持**Android**与**iOS**双端，但在实践中却存在了极多系统差异性。有些体现在了RN文档中，有一些则体现在了issue中，包括其他一些问题，GitHub上RN的近700个issue足以让人望而却步。如果不能高效处理开发中遇到的各种匪夷所思的问题，那么工期就会出现严重风险。此外，RN在Android和iOS上的性能也不尽相同，Android上更差一些，即便你完成了整个业务功能，却还要在性能优化上消耗精力。并且无论如何优化，单线程模型既要实现流畅的转场动画，又要操作一系列数据，需要很高的技巧才能保证可观的性能表现。在具体的实践中，对于

H5，往往由于时间关系，业务上先会上一个还算过得去的版本，过后再启动性能优化。然而对于RN，很有可能达到“过得去”的标准都需要大量的重构工作。

再次，**RN**虽然以**Native**渲染元素，但毕竟是运行在**JavaScript Core**内核之上，依旧是单线程，相对于**H5**这并没有对性能有革命性质的提升。Animated动画、大ListView滚动都是老生常谈的性能瓶颈，为了解决一些复杂组件所引起的性能和兼容性问题，诸多团队纷纷发挥主动能动性，自己建设基于Native的高性能组件，这有两方面问题，一是不利于分发共享，因为它严重依赖特定的客户端环境，二是它仍依赖客户端发版，仍需要客户端的开发，违背了RN最最重要的初衷。可以想象，在大量频繁引用Native组件后，RN又退化成了H5+Hybrid模式，其UI的高性能优势将会在设备性能不断升级下被削弱，同时其无stable版本反而给开发带来了更多不可预测的风险变量。

最后，**RN**仍然难以调试和测试。特别是依赖了特定端上组件之后，本地的自动化测试几乎成为了不可能，而绝大多数客户端根本不支持自动化测试。而调试只能利用remote debugger有限的能力，在性能分析上都十分不便。

可以说**RN**的出现带给了移动开发以独特的新视角，使得利用JavaScript开发Native成为了可能，NativeScript、Weex等类似的解决方案也发展开来。显然**RN**目前最大的问题仍然是不够成熟和稳定，利用RN替代Native依然存在着诸多风险，这对于重量级的、长期维护的客户端产品可能并不是特别适合，比如Facebook自己。RN的优势显而易见，但其问题也是严重的，需要决策者对个方面利弊都有所了解，毕竟这种试错的成本不算小。

由于时间关系，市场上并没有一个产品在RN的应用上有着足够久的实践经验，大部分依然属于“我们把RN部署到客户端了”的阶段，我们也无法预测这门技术的长久表现，现在评价RN的最终价值还为时尚早。在2017年，期待RN团队能做出更长足的进步，但不要太乐观，以目前的状态来看，想达到stable状态还是有着相当大的难度。

4.6 Redux 与 Mobx

Redux 成功成为了 React 技术栈中的最重要成员之一。与Vue.js一样，Redux也是凭借着比其他Flux框架更简单易懂的API才能脱颖而出。不过已经很快有人开始厌烦它每写一个应用都要定义action、reducer、store以及一大堆函数式调用的繁琐做法了。

Mobx也是基于ES5 setter，让开发者可以不用主动调用action函数就可以触发视图刷新，它只需要一个store对象以及几个decorator就能完成配置，确实比Redux简单得多。

在数据到视图同步上，无论使用什么样的框架，都有一个至关重要的问题是需要开发者自己操心，那就是在众多数据变动的情形下，如何保证视图以最少的但合理的频率去刷新，以节省极其敏感的性能消耗。在Redux或Mobx上都会出现这个问题，而Mobx尤甚。为了配合提升视图的性能，你依然需要引入action、transaction等高级概念。在控制流与视图分离的架构中，这是开发者无可避免的关注点，而对于Angular、Vue.js，框架会帮你做很多事情，开发者需要考虑的自然少了许多。

4.7 Bootstrap

BootstrapBootstrap 4处于alpha阶段已经非常久了，即使现在3.x已经停止了维护，它似乎受到了Twitter公司业务不景气的影响，GitHub上的issue还非常多。Bootstrap是建设内部平台最佳的CSS框架，特别是对于那些对前端不甚了解的后端工程师。我们不清楚Bootstrap还能坚持多久，如果Twitter不得不放弃它，最好的归宿可能是把它交给第三方开源社区去维护

五、工程化与架构

5.1 Rollup 与 Webpack 2

Rollup是近一年兴起的又一打包工具，其最大卖点是可以对**ES2015 Modules**的模块直接打包，以及引入了Tree-Shaking算法。通过引入Babel-loader，Webpack一样可以对ES2015 Modules进行打包，于是Rollup的亮点仅在于Tree-Shaking，这是一种能够去除冗余，减少代码体积的技术。通过分析AST（抽象语法树），Rollup可以发现那些不会被使用的代码，并去除它。

不过Tree-Shaking即将不是Rollup的专利了，Webpack 2也将支持，并也原生支持ES6 Modules。这可以说是“旁门左道”对主流派系进行贡献的一个典型案例。

Webpack是去年大热的打包工具，俨然已经成为了替代**grunt/gulp**的最新构建工具，但显然并不是这样。笔者一直认为Webpack作为一个*module bundler*，做了太多与其无关的事情，从而表象上看来这就是一个工程构建工具。经典的构建需要有任务的概念，然后控制任务的执行顺序，这正是Ant、Grunt、Gulp做的事情。Webpack不是这样，它最重要的概念是*entry*，一个或者多个，它必须是类JavaScript语言编写的磁盘文件，所有其他如CSS、HTML都是围绕着entry被处理的。估计你很难一眼从配置文件中看出Webpack对当前项目进行了怎样的“构建”，不过似乎社区中并没有人提出过异议，一切都运行得很好。

> 题外话：如何使用Webpack构建一个没有任何JavaScript代码的工程？

新的Angular 2使用Webpack 2编译效果更加，不过，已经提了一年的Webpack 2，至今仍处于beta阶段，好在现在已经rc，相信离release不远了。

5.2 npm、jspm、Bower与Yarn

在模块管理器这里，npm依旧是王者，但要说明的是，npm的全称是`node package mamager`，主要用来管理运行在Node上的模块，但现在却托管了大量只能运行在浏览器上的模块。造成这种现象的几个原因：

1. Webpack的大量使用，使得前端也可以并习惯于使用CommonJS类型的模块；
2. 没有更合适的替代者，Bower以前不是，以后更不会是。

前端的模块化规范过去一直处于战国纷争的年代。在Node上CommonJS没什么意见。在浏览器上，虽然现在有了ES2015 Modules，却缺少了模块加载器，未来可能是SystemJS，但现在仍处于草案阶段。无论哪种，都仍处于JavaScript语言层面，而完整的前端模块化还要包括CSS与HTML，以及一些二进制资源。目前最贴近的方案也就只能是JSX+CSS in JS的模式了，这在Webpack环境下大行其道。这种现象甚至影响了Angular 2、Ember 2等框架的设计。从这点看来，jspm只是一个加了层包装的壳子，完全没有任何优势。

npm本身也存在着各种问题，这在实践中总会影响效率、安全以及一致性，Facebook果断地出品了Yarn——npm的替代升级版，支持离线模式、严格的依赖版本管理等在工程中非常实用的特性。

至于前端模块化，JavaScript有CommonJS和ES2015 Modules就够了，但工程中的组件，可能还需要在不同的框架环境中重复被开发，它们依旧不兼容。未来的话，WebComponents可能是一个比较优越的方案。

5.3 同构

同构的设计在软件行业早就被提出，不过在Web前端，还是在Node.js、特别是React的出现后，才真正成为了可能，因为React内核的运行并不依赖于浏览器DOM环境。

React的同构是一个比较低成本的方案，只要注意代码的执行环境，前后端确实可以共享很大一部分代码，随之带来的一大收益是有效克服了SPA这种前端渲染的页面在首屏性能上的瓶颈，这是所有具备视图能力的框架Angular、Vue.js、React等的共性问题，而现在，它们都在一种程度上支持server render。

可以想到的做前后端同构面临的几个问题：

1. 静态资源如何引入，CSS in JS模式需要考虑在Node.js上的兼容性；
2. 数据接口如何**fetch**，在浏览器上是AJAX，在Node.js上是什么；
3. 如何做路由同构，浏览器无刷新切换页面，新路由在服务端可用；
4. 大量**DOM**渲染如何避免阻塞**Node.js**的执行进程。

目前GitHub上star较多的同构框架包括Vue.js的nuxt和React的next.js，以及数据存储全包的meteor。可以肯定的是，不论它们是否能部署在生产环境中，都不可能满足你的所有需求，适当的重新架构是必要的，在这个新的领域，没有太多的经验可以借鉴。

六、未来技术与职业培养

6.1 大数据方向

越来越多做***toB***业务的公司对前端的需求都是在数据可视化上，或者更通俗一些——报表。这个部分在从前通常都是前端工程师嗤之以鼻的方向，认为无聊、没技术。不过在移动端时代，特别是大数据时代，对此类技能的需求增多，技术的含金量也持续提升。根据“面向工资编程”的原则，一定会有大量工程师加入进来。

对这个方向的技术技能要求是**Canvas**、**WebGL**，但其实绝大多数需求都不需要你直接与底层**API**打交道，已经有大量第三方工具帮你做到了，不乏非常优秀的框架。如百度的ECharts，国外的Chart.js、Highcharts、D3.js等等，特别是D3.js，几乎是大数据前端方向的神器，非常值得学习。

话说回来，作为工程师，心存忧患意识，一定不能以学会这几款工具就满足，在实际的业务场景中，更多的需要你扩展框架，生产自己的组件，这需要你具备一定的数学、图形和**OpenGL**底层知识，可以说是非常大的技术壁垒和入门门槛。

6.2 WebVR

今年可以说是VR技术爆发式的一年，市场上推出了多款VR游戏设备，而淘宝更是开发出了平民的*buy*+购物体验，等普及开来，几乎可以颠覆传统的网上购物方式。

VR的开发离不开对3D环境的构建，WebVR标准还在草案阶段，A-Frame可以用来体验，另一个three.js框架是一个比较成熟的构建3D场景的工具，除了能在未来的VR应用中大显身手，同样也在构建极大丰富的3D交互移动端页面中显得必不可少，淘宝就是国内这方面的先驱。

6.3 WebAssembly

asm.js已发展成WebAssembly，由谷歌、微软、苹果和Mozilla四家共同推动，似乎是非常喜人乐见的事情，毕竟主要浏览器内核厂商都在这里了。不过合作的一大问题就是低效，今年终于有了可以演示的demo，允许编写C++代码来运行在浏览器上了，你需要下载一大堆依赖库，以及一次非常耗时的编译，不过好歹是个进步。

短时间内，我们都不太可能改变使用JavaScript编写前端代码的现状，Dart失败了，只能期望于未来的WebAssembly。有了它，前端在运行时效率、安全性都会上一个台阶，其他随之而来的问题，就只能等到那一天再说了。

6.4 WebComponents

WebComponents能带给我们什么呢？HTML Template、Shadow DOM、Custom Element和HTML Import？是的，非常完美的组件化系统。Angular、React的组件化系统中，都是以Custom Element的方式组合HTML，但这都是假象，它们最终都会被编译成JavaScript才会执行。但WebComponents不一样，Custom Element原生就可以被浏览器解析，DOM元素本身的方法都可以自定义，而且元素内部的子元素、样式，由于Shadow DOM的存在，不会污染全局空间，真正成为了一个沙箱，组件化就应该是这个样子，外部只关心接口，不关心也不能操纵内部的实现。

当前的组件化，无不依赖于某一特定的框架环境，或者是Angular，或者是React，想移植就需要翻盘推倒重来，也就是说他们是不兼容的。有了WebComponents，作为浏览器厂商共同遵循和支持的标准，这一现状将极有可能被改写。

未来的前端组件化分发将不会是npm那么简单，可能只是引用一个HTML文件，更有可能的是包含CSS、HTML、JavaScript和其他二进制资源的一个目录。

目前只有最新的Chrome完全支持WebComponents的所有特性，所以距离真正应用它还尚需时日。由于技术上的限制，WebComponents polyfill的能力都非常受限，Shadow DOM不可能实现，而其他三者则更多需要离线编译实现，可以参考Vue.js 2的实现，非常类似于WebComponents。

6.5 关于微信小程序

微信小程序对于今年不得不说，但笔者却也无话可说。依托于庞大的用户量，微信官方出品了自有的一套开发技术栈，只能说给繁杂的前端开发又填了一个角色——微信前端工程师。

七、总结

最后还有几点需要说明。

7.1 工程化

首先，现在业界都在大谈前端工程化，人人学构建，个个会打包。鄙人认为，工程化的要点在于“平衡诸方案利弊，取各指标的加权收益最大化”。仅仅加入了项目构建是远远不够的，在实践中，我们经常需要考虑的方向大可以分为两种：一是研发效率，这直接应该响应业务需求的能力；二是运行时性能，这直接影响用户的使用体验，同时也是产品经理所关心的。这两点都直接影响了公司的收入和业绩。

具体到细节的问题上来，比如说：

1. 静态资源如果组织和打包，对于具备众多页面的产品，考虑到不断的迭代更新，如何打包能让用户的代码下载量最少（性能）？不要说使用Webpack打成一个包，也不要说编译common chunk就万事大吉了，难道还需要不断地调整编译脚本（效率）？改错了怎么办？有测试方案么？
2. 利用Angular特别是React构建纯前端渲染页面，首屏性能如何保证（性能）？引入服务端同构渲染，好的，那么服务端由谁来编写？想来必是由前端工程师来编写，那么服务端的数据层架构是怎么样的？运维角度看，前端如何保证服务的稳定（效率）？
3. 组件化方案如何制定（效率）？如果保证组件的分发和引用的便捷性？如何保证组件在用户端的即插即用（性能）？

对于工程师来说，首先需要量化每个指标的权重，然后对于备选方案，逐个计算加权值，取最大值者，这才是科学的技术选型方法论。

然而在业界，很少能看到针对工程化的更深入分享和讨论，大多停留在“哪个框架好”，“使用XXX实现XXX”的阶段，往往是某一特定方向的优与劣，很少有科学的全局观。甚至只看到了某一方案的优势，对其弊端和可持续性避而不谈。造成这种现状的原因是多方面的，一是技术上，工程师能力的原因并没有考虑得到，二是政治上，工程师需要快速实现某一目标，以取得可见的KPI收益，完成团队的绩效目标，但更多的可能是，国内绝大多数产品的复杂性都还不够高，根本无需考虑长久的可持续发展和大规模的团队合作对于技术方案的影响。

因此，你必须接受的现状是，无论你是否使用**CSS**预处理器、使用**Webpack**还是**Grunt**、使用**React**还是**Angular**，使用**RN**还是**Hybrid**，对于产品极有可能都不是那么地敏感和重要，往往更取决于决策者的个人喜好。

7.2 角色定位

确实，近两年，Web前端工程师开始不够老实，要么用Node.js插手服务端开发，要么用RN插手客户端开发。如何看待这些行为呢？

鄙人以为，涉足服务端开发是没问题的，因为只涉及到渲染层面，还是属于“前端”的范畴的。况且，在实际的工程实践中，已经可以证明，优秀的前端研发体系确实离不开服务端的参与，想想Facebook的BigPipe。不过，这需要服务端良好的分层架构，数据与渲染完全解耦分离，后端工程师只负责业务数据的CRUD，并提供接口，前端工程师从接口中获取数据，并推送到浏览器上。数据解耦是比接口解耦更加优越的方案。因此现在只要你的服务端架构允许，Node.js作为Web服务已经比较成熟，前端负责服务端渲染是完全没有问题的。

前端涉足客户端开发也是合理的，毕竟都运行在用户端，也属于前端的范畴。抛开阿里系的Weex鄙人不甚了解，NativeScript、RN都还缺乏大规模持续使用的先例，这是与涉足服务端领域的不同，客户端上的方案都还不够成熟，工具的限制阻碍了前端向客户端的转型，仍然需要时间的考验。不过时间可能不会很多，未来的Web技术依托高性能硬件以及普及的WebGL、WebRTC、Payment API等能力，在性能和功能上都会挑战Native的地位。最差的情况，还可以基于Hybrid，利用Native适当扩展能力，这就是合作而非竞争关系了。

总之前端工程师的本仍然在浏览器上，就这一点，范围就足够广使得没人有敢言自己真正“精通”前端。如果条件允许的话，特别是技术成熟之后，涉猎其他领域也是鼓励的。

7.3 写在最后

在各种研发角色中，前端注定是一个比较心累的一个。每一年的年末，我们都能看到几乎完全不一样的世界，这背后是无数前端人烧脑思考、激情迸发的结果。无论最终产品的流行与否，都推动着前端技术领域的高速更新换代。正是印证了那一句“唯有变化为不变”。作为业务线的研发工程师，我们的职责是甄选最佳组合方案，取得公司利益最大化。这个“最佳”的涉猎面非常广，取决于设计者的技术视野广度，也有关于决策者的管理经验，从来都不是一件简单的事。

未来的**Web**前端开发体验一定是更丰富的，依托WebComponents的标准化组件体系，基于WebAssembly的高性能运行时代码，以及背靠HTTP/2协议的高速资源加载，前端工程师不必在性

能上、兼容性上分散太多精力，从而可以专注于开发具备丰富式交互体验的下一代Web APP，可能是VR，也可能是游戏。

在迎接**2017**的同时，我们仍然要做好心理准备，新的概念、新的框架和工具以及新的语法依旧会源源不断的生产出来，不完美的现状也依旧会持续。

由于水平有限，笔者在上述内容中难免有失偏颇，请多包涵。

前端之巅

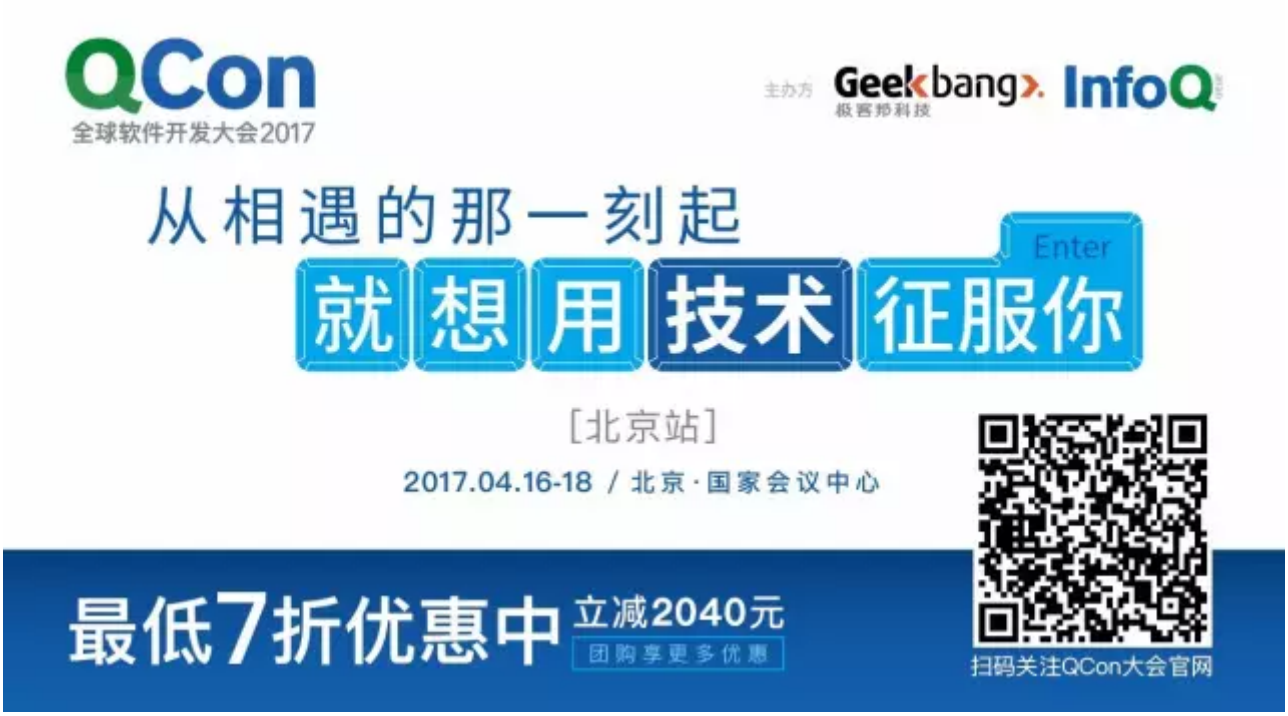
「前端之巅」是InfoQ旗下关注前端技术的垂直社群，欢迎各位前端工程师的加入！加群请关注“前端之巅”公众号并发送“加群”，投稿请请发邮件到editors@cn.infoq.com，注明“前端之巅投稿”。

视野拓展

每一次相遇都是久别重逢。时隔一年，QCon北京站华丽归来。20+热点专题出炉，涵盖区块链、VR、TensorFlow、深度学习等潮流技术，及前端工程、研发安全、移动专项、智能运维、业务架构等一手实践。国内外技术专家共襄盛举，即刻报名，尽享7折特惠。

在前端技术飞速发展的现在，我们手上有大把技术方案，但同时业务复杂度和团队规模越来越大，工程难度不断增高。应该如何利用现有技术和平台，系统性地构建和维护前端工程？

扫描下图二维码，相约**QCon北京站前端工程实践专场**，这里有你想要的答案！



长按二维码关注

前 端 之 巅

紧跟前端发展
共享一线技术
万名淀粉互助
共登前端之巅

Read more