2016年前端技术观察

分类 大话编程 (/home/大话编程) **关键字** 分享 (/bbs/分享) **发布** ourjs (/userinfo/ourjs) 2016-12-16
注意 转载须保留原文链接，译文链接，作者译者等信息。

作者：曹刘阳，网名阿当，资深Web技术专家，必杀技"前端开发 (http://lib.csdn.net/base/javascript)、软件架构 (http://lib.csdn.net/base/architecture)和敏捷 (http://lib.csdn.net/base/agile)开发"。先后任职于雅虎、淘宝、新浪企业。在"如何编写高质量代码"领域研究颇深，《编写高质量代码——Web前端开发修炼之道》作者

这篇文章可能是火热的前端圈里，泼向很多技术的一盆冷水。我无意冒犯这些技术的传道者，文章只关乎技术，不关乎人。我对任何新技术于个人的成长都持肯定态度，对任何热爱学习的同学都表示欣赏。而技术选型要考虑个人开发还是团队合作，Geek思维在团队合作实战时会有很多隐患。个人开发，我倾向于激进，团队合作，我倾向于保守。

写在前面的话

2015年底，我在微博上谈起对Sass、React (http://lib.csdn.net/base/react)和Angular的不满，出乎意料地引起了一场论战。从知乎至微博，与我相识或不相识的许多人卷了进来。这场争论中很多人的言论让我意识到，前端这两年居然乱到了这个地步——我不怕技术大爆炸，但怕劣币驱逐良币，怕人们丢了思考，陷入不必要的泥潭。

大战中我也被很多人贴了标签——守旧的老人，懒惰不思进取又倚老卖老。有同学笑称我是气宗，看不惯剑宗。

然而对于技术，无论是CSS重构、RIA、JavaScript (http://lib.csdn.net/base/javascript)框架、打通前后端的全栈、H5、移动前端开发、SPA、敏捷、Hybrid、Canvas2d和WebGL，我都非常积极甚至超前。对这两三年流行起来的技术，并非因为守旧而排斥，而是确实做过思考，结合自己过往的经验，持很重的怀疑态度。

希望读者警惕两件事：

跨界而来的技术专家和他们的解决方案，注重个人能力忽略团队合作的Geek。他们的方案出发点可能是好的，但其实有可能会有更具性价比也更容易落地更少隐患的技术方案。一个是视野问题，另一个是实践经验问题。跨界而来的专家往往能带来不同领域的视野，这是好事，但他们同时也缺乏本领域的实践经验。Geek是个有点酷的词，意味着高手，但同时也意味着门槛，而一旦上升到团队合作，门槛和合作时的内耗，以及团队成员流失后交接的隐患就会成倍上升。

只提优点不提缺点的技术布道。布道某种意义上和销售没区别，会选择性地给听众传达信息。因此在面对布道时，不要只听官方说的好处，要自己思考它的缺点——技术方案不会只有优点而没缺点，做任何的技术选型都是在对优点和缺点做权衡。而缺点，别指望布道者会主动告诉你。如果你有足够工作经验，请结合实际经验勇敢地提问，如果没有太多工作经验，就真的很麻烦了，你需要加倍谨慎和怀疑。

经验、反思、置疑精神，希望同学们可以在当下这前端界的乱世下，通过这三个武器寻找到自己的路。

关于CSS预处理器

我对CSS预处理器一直无感。不是说它们毫无价值，而是说，性价比不高。关于预处理器的优点，官方都说过了，对吧？我就不赘述了，说说缺点吧。

预处理器的引入，为本来很简单的CSS书写带来了编译的步骤，也带来了以下几个问题：
* 所有开发团队成员需要安装预处理工具，需要学习预处理器语法，增加了上手门槛。
* 不可以直接对编译后的CSS文件进行修改，要统一修改源文件。这在调试时，增加了工作量。
* 源文件增加了很多原生CSS不具备的功能，如定义变量和嵌套，这在提供方便的同时，也引入了一些不必要的麻烦，比如嵌套带来的副作用——CSS选择符权重增加，HTML结构与样式挂钩耦合，以及预处理器层面需要增加规范（变量和mix模块放在什么地方定义、文件如何拆解以配合@import导入、什么时候定义新的class名、什么时候使用嵌套），以方便团队合作。
* 如果需要跨团队合作，如将代码和另一支不使用预处理器的团队的代码做整合，会怎么样？
* 如果团队有人离职，他维护的代码在交接上是否会需要多进行一些沟通？有新人入职，是否在正式工作前，需要学习预处理器语法，以及公司关于预处理器的使用规范，导致进入工作状态的成本变高了？如果团队在新人入职后的新兵训练营工作做得并不好，又会怎么样？事实上，很多公司压根没有什么新兵训练营计划吧？

CSS预处理器带来的这些麻烦，一旦上升到团队合作，成本和风险就大了，因而并不是笔低成本买卖。再看看它号称的收益，是否真的没有别的解决方案了，如果有话，即使是单兵作战，是不是也不是什么高收益的选择了呢？

变量、mixin、扩展/继承的替代方案 —— 让HTML标签挂多个class

Sass变量方案:

```
$hightLight: red;
.title{color:$hightLight;font-size:20px;}
.title2{ color:$hightLight;fontsize:16px;}
<h1 class="title">标题1</h1>
<h2 class="title2">标题2</h1>
```

多class方案：

```
.hightLight{color: red;}
.title{ font-size:20px;}
.title2{font-size:16px;}
<h1 class=”title highLight”>标题1</h1>
<h2 class=”title2 highLight”>标题2</h1>
```

Sass继承方案：

```
.message{border: 1px solid #ccc; padding:10px; color: #333;}
.success{@extend .message; border-color:green;}
.error{@extend .message; border-color:red;}
.warning{@extend .message; border-color:yellow;}
<div class="success">成功</div>
<div class="error">错误</div>
<div class="warning">警告</div>
```

多class方案：

```
.message{border: 1px solid #ccc; padding:10px; color: #333;}
.success{border-color: green;}
.error{border-color: red;}
.warning{border-color: yellow;}
<div class="message success">成功</div>
<div class="message error">错误</div>
<div class="message warning">警告</div>
```

嵌套的副作用以及低成本的替换方案 —— 长命名
嵌套会带来HTML结构与CSS选择符耦合、选择符权重增加的副作用，而短命名会带来易冲突的问题，如果代码一旦进行修改，耦合就会带来不必要的麻烦，而CSS作为层叠样式表，选择符的权重直接决定了当不同的选择符样式出现冲突时，使用哪个选择符定义的样式。预处理器看似提供了作用域，其实只是个障眼法，编译出来的结果，只是普通的子孙选择器而已。如果依赖预处理器的这种错觉，采用短命名的话，当多人合作或采用低质量的第三方代码时，很可能会产生冲突。你能保证自己不直接使用短命名，一定会嵌套着使用，但无法保证别人不直接在全局使用短命名。如果代码持续维护，如果多人合作，如果DHTML很多，这三个副作用都会是不小的麻烦。

作为一门DSL，CSS没有编程特性，没有作用域，是很正常的，同样是没有编程特性的另一个工具Redis (http://lib.csdn.net/base/redis)，就使用长命名很好地解决了作用域问题，比如Redis在实践过程中，被鼓励这样的命名：usersInfo:1000:name、usersInfo:1001:age。其实CSS也是种键值对的形式，选择符可以嵌套，但也可以完全借长命名来解决作用域问题。嵌套带来的作用域是在预处理器中的障眼法而已，如果和第三方合作，需要一些必要的约定，很脆弱。而长命名带来的作用域，是非常强的真作用域，只要命名空间前辍不冲突，怎么命名都是安全的。

Sass嵌套方案：

```
.box{
.header{}
.body{}
.footer{}
}

<div class="box">
<div class="header">header</div>
<div class="body">body</div>
<div class="footer">footer</div>
</div>
```

长命名方案：

```
.box{}
.box_header{}
.box_body{}
.box_footer{}

<div class="box">
<div class="box_header">header</div>
<div class="box_body">body</div>
<div class="box_footer">footer</div>
</div>
```

所以对于CSS预处理器，我个人并不支持，更推荐使用原生CSS技巧来解决问题。性价比高，副作用小。

跨界、CoffeeScript、TypeScript和ES6

从2011年Ruby社区为前端圈带来了CoffeeScript开始，前端圈开始对语法糖特别关心了。CoffeeScript、TypeScript、ES6接踵而来。印象里，对语言最感兴趣的，是Server端工程师，老爱谈这个语言的语法特性，那个语言的语法特性，这个语言的性能，那个语言的性能。很少见C++客户端、AS、游戏客户端的同学谈论语言吧？为什么呢？

因为客户端和图形界面打交道，而图形界面在细节上的工作量非常大，这里有没有对齐，那里能否自适应兼容硫片化终端，这里的动效是否流畅，图形界面提供的API其实很繁复，仅是熟悉这些API，了解不同API在实战时可维护性上的区别就会花上大量时间。我常举一个例子，说明H5在布局方案上有多头痛——垂直居中。听到这四个字，有经验的前端工程师，脑子里能马上想到诸多种情况以及多种实现方案，不同的方案有不同的优缺点。而做服务端开发的同学其实完全不会遇到这些问题。再举一个例子吧，同样是使用Java (http://lib.csdn.net/base/javase)语言的，你看看搞Android (http://lib.csdn.net/base/android)开发的同学是不是像搞Java服务端的一样，那么喜欢研究不同语言语法糖和性能？

我入行不久就开始前后端都写，前端写HTML、CSS、JavaScript、Flash，后端写PHP (http://lib.csdn.net/base/php)。之后一度专攻前端，然后又从2010年开始继续前后端都写，后端我没有继续选PHP，也完全看不上Node，探了下Ruby on Rails的坑后，果断选了Python (http://lib.csdn.net/base/python)。虽然后端我不是那么精通，但编程经验还是不少。可能有不少后端的同学可能会不服气，但我的确觉得后端的工作量比前端少很多，也简单很多。我做过的几个包揽前后端的项目里，花在前端上的时间至少3倍于后端时间。在MVC框架的帮助下，后端的编码体验其实很轻松，设计好数据库 (http://lib.csdn.net/base/mysql)和缓存，工作量基本上就完成了一半。然后Controller里的代码不会很跳跃，专心在逻辑上就可以了。但在写前端代码时，除了要思考逻辑，还要思考表现层的工作。前端表现层的工作可不像服务端所谓的view层那样，搞个模板引擎就完了，前端的CSS能否用好，一是关系到设计图的还原度，二是关系到终端碎片化的自适应，三是关系到需求变更时维护的难易度。

很多搞服务端的同学转到前端来时，都会栽在CSS里。所以Bootstrap火了，你问问谁对Bootstrap最依赖，绝对是搞服务端的程序员。CSS2时代，服务端的同学对CSS尚且难以攻克，到了Mobile和CSS3 (http://lib.csdn.net/base/css3)时代，这鸿沟怕是更难以轻易逾越了。同样是DSL，CSS可比SQL复杂多得多。

有追求一点的服务端同学为什么那么喜欢研究不同语言的语法糖，以及性能问题？一是因为性能问题对于服务端是真需求，特别对于高访问量高并发的产品，二是因为工作量没那么大，知识点也没那么多，三是因为换语言这事其实学习成本没那么高。

怎么理解“换语言这事学习成本没那么高”这话？做任何一个领域的开发，要掌握的知识都是多方位的，语言只是其中之一而已，比如服务端开发至少要学一门语言，一个框架，一种数据库，一种Web服务器，另外，大多数情况下还得学习一种缓存，最好再学一下Linux (http://lib.csdn.net/base/linux)运维。大多数语言只是语法糖有或大或小的区别，核心都差不多，变量、常量、类、对象、数组、字典、字符串、布尔、if else、switch、函数、try catch、for循环。即使不使用该语言的高级语法糖，也一样可以用常规的通用的编程技巧完成工作。

我用过JavaScript、AS、PHP、Java、Ruby和Python语言，也用过HTML、Flash、Flex、wxPython、Canvas2d和WebGL界面开发，还用过SSH、ROR、Django的服务端开发，所以当服务端同学在聊不同语言时，我一点也不羡慕，别自娱自乐了，真那么好学，有本事把CSS3搞定。

说回到CoffeeScript、TypeScript和ES6上。CoffeeScript和TypeScript分别是Ruby和C#社区的产物，它们都觉得JavaScript语法不够好，想改善它。只是它们俩分别代表了豪放派和严谨派，是两个极端，让它们俩先打一架好了。

我既不支持CoffeeScript也不支持TypeScript，原因很简单，我不想多加一层编译环节，JavaScript语法固然不完美，可是不完美的语言多了，Lua更简陋，一样活得好好的。GWT曾经想让Java语言通过编译直接解决JavaScript语言的问题，然后呢？然后GWT死了。连谈恋爱的人都知道，不要试图改变对方，你能改变的只有你自己。与其想着保留自己熟悉的语法习惯试图改变JavaScript语法，不如改变自己认真开始学JavaScript。有同学可能会说，TypeScript是JavaScript的超集，你可以完全按照自己的习惯写JavaScript。同学，可是在团队合作的规范上，这不是给自己埋坑吗？

CoffeeScript只在刚出来时，我身边有几位同学短时间地表示了一下兴趣，然后也没见着他们真的跟进去了。而TypeScript出来时，我身边没有一个前端工程师有丝毫兴趣，最感兴趣的是写C++和AS3的同学。我完全不看好这两门JavaScript预处理语言，在我看来，它们都是非前端社区的产物，迎合的也是非前端的工程师群体的编程习惯，它们很难打入真前端工程师的圈子。

再说ES6，它固然提供了更友好的语法糖，可是，这些语法糖真的像很多同学说的那样，是救世主，能“极大地提高前端开发的工作效率”吗？我持怀疑态度。首先，我觉得JavaScript的语法规则在进化是好事，只是在实际工作时不必拔苗助长，我反对Babel。Babel的做法太激进了，ES6的语法规则在团队合作上，肯定会带来门槛和内
耗，Babel编译又进一步加大了调试的成本，得不偿失。工作效率的极大改善，从来不在语法糖这里，就像Ruby宣称的，语法糖是为了让程序员工作起来开心的。这开心的成本若太高，就不值当了。还是那话，个人开发和团队开发是有区别的。个人开发可以激进，而团队开发最好保守。

语言层面保守的解决方案是什么呢？是通过JavaScript框架 (http://lib.csdn.net/base/angularjs)来封装语法糖。最著名的例子就是jQuery (http://lib.csdn.net/base/jquery)，jQuery几乎把原生的DOM方法都给重写了吧？为一些实用的语法特性封装个Lang库，把想要的语法封装起来嘛，class、extend、map，没有什么功能是封装不出来的吧？其实在Babel出来之前，Lang库一直是主流的解决方案，并非新鲜事物。

急什么呢？CSS2不是过渡到CSS3了吗？HTML4不是过渡到HTML5 (http://lib.csdn.net/base/html5)了吗？该来的会来的，不该来的，急也不急不来，看看ES4怎么死的？再看看Python 2和Python 3。

关于Node

上面我提到的任何一个领域，要用到的知识都是多方位的。比如服务端要掌握的知识除了语言核心语法，还有语言的功能模块、Web框架、数据库、缓存、服务器配置、Linux运维、性能优化，甚至可能还得学消息中间件和网络安全。这是一整套知识体系，而且编程体验和前端截然不同。

所以当Node横空出世时，我看了下它楚楚可怜的模块库、文档和版本号，还有作者开发它的初心，心想这个玩具也就写写脚本吧，肯定忽悠不了人的，谁会拿生产环境陪它躺坑？它唯一能提供的只有可怜的“核心语法”这一项吧？但这一项的学习成本对于整个服务端开发知识体系来说，算个啥？Node距离成熟，还有很长的一段路要走，所以当npm笑话一般的unpublish事件爆发时，我一点也不意外，这是填坑必须付出的代价。

事实上，打从一开始，我就没觉得Node跟前端有什么关系。是啊，服务端有了个基于es语法的解决方案了，可是，关前端什么事？那些说全栈的同学，麻烦等一等，下面我会专门提到全栈的事。先还是说说Node的处境吧。

Node后来的发展，我猜中了一半，猜错了另一半。Node当前有两个用途，一是服务端开发，二是脚本工具。我猜中了传统服务端圈不待见它，但也很惊讶它作为脚本工具居然如火如荼。

因为传统服务端开发的同学，对es核心语法本就不算熟，所以Node在安利他们时，只能拿异步性能说事，但事实上服务端也有异步的框架，比如Gevent和Twisted，所以并没有太大的吸引力。也不是说完全忽悠不到任何一些服务端工程师，只是人家很可能在保留Java、PHP、Python为主武器的基础上，简单试试Node，毕竟es语言的学习成本也不高，什么MD5模块啊、Express之类的框架啊并没有什么新东西，新瓶装旧酒而已，比起CSS可熟悉太多了。只是这波同学忠诚度不会高的，人家愿意多学一门Node的，也不会介意再多学一门Go (http://lib.csdn.net/base/go)、Erlang、Ruby什么的。而且真需要团队合作了，需要赶时间了，需要接受性能挑战了，人家还是会换回主武器的。就算真起来个新贵，从口碑来看，我也更看好Go。

Node在安利Web前端圈的同学时，就主要拿es说事，什么一门语言就可以前后通吃了呀，全栈呀。拜托，后端编程的知识体系和前端编程的知识体系是一样的吗？十万八千里好吗？如果是些工作经验足够的前端工程师，你让他扩展下后端知识，这也是有益无害的，问题是，Node忽悠了一大片又懒又有野心的新手孩子。我之所以讨厌Node圈的布道者，就是这个原因。试想想，用Node做服务端开发，对于一个传统前端工程师来说，唯一可以偷懒的地方是什么？不只有“es核心语法”这一点吗？一个连核心语法都懒得学的人，你指望他花大量精力去攻克后端的那些庞杂知识体系？现在搞Node的前端工程师最喜欢的数据库是什么？是MongoDB (http://lib.csdn.net/base/mongodb)。我一点不意外。为什么？因为数据库好设计，不就是个JSON吗？嵌套多好设计，数组字典都有，还搞什么外键，什么三大范式啊多累。可是你知道传统服务端工程师都不敢在正式项目中使用MongoDB做持久化存储，只倾向于拿它存存日志吗？好吧，就算这帮孩子真的花了大量精力成了合格的后端，而不仅仅是个票友，可是这时他还是前端工程师吗？也许他成了合格的后端，但他的前端知识学扎实了吗，日常工作都已经熟练了吗？怕就怕哪个都没学会，哪个都是个半吊子，既不对自己负责，也不对公司负责，然后还想当然地自以为全栈，一心想做架构师，想马上加薪，以对得起自己的“过人能力”。

Node进军Server端，当前来看，是失败的，它丝毫动摇不了Java和PHP的主力地位。也许一些公司里有Node团队，但看客请注意三件事：

- Node的市场份额如何；
- 公司Node团队的Leader是什么出身，前端还是服务端；
- Node团队负责的业务是不是稳定性、安全性要求高的核心业务。

某位JavaScript框架作者跟我争论时，理直气壮地拿出几篇国外的博客给我看，说Node在国外已被广泛应用了。我看了下，一篇是这样的：2013年，一个设计师出身转型工程师具体经验不明的人，在内部Java服务端工程师们持怀疑态度的情况下，将一小块功能尝试性地由Java迁到了Node，并证明性能不错。很高兴地向外界发了博文宣布公司里生产环境用上了Node，并表示未来会用Node重构更多地方。另一篇是这样的：1个前Google工程师为招人，写了篇介绍公司技术选型介绍的文章，提到了服务端选型用到了Node和Go，Node因为事件循环阻塞出现了性能问题，作者通过多起实例的办法填了这坑，然后用Go的部分一切顺利。数据库选型没听过。作者自认技术选型非主流，而且招聘的工程师要求也非主流，最好全端全栈。这就是Node做服务端开发拿出来说事的案例？ 这里有一篇关于Facebook员工的访谈，谈到了Facebook是否使用Node进行服务端开发的问题，答案是“Whenever they want to run JavaScript on server (e.g. rendering React on server) they just use V8 directly rather than using Node”（文章见：10 things you probably didn’t know about JavaScript (React and Node.js (http://lib.csdn.net/base/nodejs)) and GraphQL development at Facebook）请读者朋友们想想为什么。

Node写写脚本工具还是可以的，毕竟不用团队合作，也不用上生产环境去接受考验。只是写脚本，很多语言都可以胜任，Perl、Python、Ruby、Java都是不错的选择，根据自己喜好来吧。比如Sass是Ruby写的，Ant是Java写的，曾经很流行的YUI compressor也是Java写的。我在雅虎工作时，脚本工具用的是Ant，在新浪工作时，是团队用PHP自己写的脚本，在盛大工作时，我自己用Python写脚本。曾经为了方便打包合并压缩方便，我自己用wxPython写了个带GUI的工具，现在还挂在CSDN上，叫GUIcompressor。原理很简单，用wxPython写GUI，通过Python的os模块调
用YUI的compressor，简单来说就是包个壳。Node的脚本也有很多都是从别的社区包个壳就拿过来了。

脚本工具这东西确实能帮上些忙，只是在我看来，并不是太重要，也并不像某些同学说的那样能“极大地提高工作效率”。所以在过去相当长一段时间里，其实前端的所谓工程化脚本，并没有流行开来，好一点的公司才自己折腾下，一般的公司了不起就做做CSS和JavaScript文件压缩就完了。多年以前，前端的部署问题很简单，往服务器上一扔就完了。但后来随着前端的网络性能优化技巧逐渐完善，前端要做的事情变得越来越麻烦：图片合并、图片压缩、代码合并、代码压缩、动态加载、CDN。如果再加上jsLint之类的代码检查，就更加麻烦了。之前流行的加载是线上动态加载，requirejs那种方式，而现在流行在打包阶段按需加载，这就使得打包变得更加复杂。我声明下对Node脚本的态度，我觉得Node能为前端带来专业的开源的工具，这是打包脚本最好的结果，比自己折腾些质量一般的脚本好太多了！对于一个成熟的团队来说，有开源的成熟的打包工具是应该的，这是Node为前端圈做的一件好事。

只是前端圈太不理性了，一会儿Grunt，一会儿Gulp，一会儿Webpack，然后听说又有人想折腾npm打包。管它哪个，选一个就完了呀，猴子掰玉米般捡一个扔一个真心没必要。类似的闹剧，在前端社区之外，还真没听说过，这也是前端社区一个极不成熟的表现。

关于跨界、全栈、公司定岗

对于跨界的人和技术方案，我一直持特别谨慎的态度，原因无他，因为我自己就跨界跨得厉害，我知道坑很多。我最早是做视觉设计的，做点线面、欧美风、韩风、icon设计、矢量图位图，然后玩DW里拖表格布局，做Flash动画、Flash脚本、Flash全站，再到CSS重构和DHTML，做后端PHP，用Flex、用Ruby on Rails、Python、Socket编程、wxPython桌面编程，用Canvas2d、WebGL、游戏开发，用敏捷，Scrum、XP、精准看板还有TDD。而现在我还参与是产品设计和产品营销。看的多了，踩的坑多了，也就开始有了自己的判断和坚持了。

在我看来，前端圈子当前的混乱，主要原因来自跨界。从2011年CoffeeScript、Less、Node同时进入前端圈推广开始，到后来的Angular、React Native (http://lib.csdn.net/base/reactnative)（React我觉得还是有价值的，但React Native就相当不接地气了），全都走偏了路。CoffeeScript是Ruby社区从自己的语法习惯出发的，Less是Ruby社区觉得CSS写起来很麻烦，好心地帮不能编程的CSS加入了一些更高级的特性，Node是位玩语言的票友意外看到了V8引擎，包了个壳，本人一点前端的基因也没有，而Angular是Google的服务端团队好心帮前端带来了更接近服务端MVC的框架，React Native是Facebook一支没有iOS (http://lib.csdn.net/base/ios)开发的团队为了让前端开发出原生App程序而折腾的。这帮人全是跨界的，带着自己的专长和对前端或深或浅的理解，提供了自认为不错的解决方案，解决了自己的问题。至于有没有更好的解决方案，至于解决方案是否会有坑，至于这些技术落地需要什么样的人员配备和职位定岗，他们都没有去讲，也没有义务去讲，为了技术推广甚至会恶意地回避去讲。而这些问题，是需要使用者们自己去思考自己去承担的。

没有技术是只有好处而全无坏处的，从来都是两面同时存在，而不同的项目，不同的团队，做技术选择时需要权衡地去看，只有优点大于缺点，风险可控的情况时，一些有风险的选择才是合适的。让我非常担心的一点是，跨界而来的这些方案，基本上只听得到赞美声，很难听到置疑声。而赞美声全是官方宣扬的，其实是不用动脑子的，也不需要自己的经验去验证的，人云亦云即可。置疑声需要有独立思考能力、项目经验支撑，另外，以我的经验来看，最近前端圈的攻击性特别强。因此，置疑者还需要有非常强大的内心。

所以，我最近常问的问题是——你知道自己正在用的技术，缺点是什么吗？看客不妨也反思下这个问题。 当我在问这个问题时，经常会得到这么几个答案：

- 你懒，你不好学。不同的技术都有它适用的场景，学学总是好的。
- 你固步自封，老守着前端这一个小领域，我要做全栈。、
- 你反对新技术是因为你思想还停留在IE 6时代吧，你切一辈子页面，写一辈子jQuery吧。

前文已经提到笔者曾经研究和实践过的技术，关于是否好学，翻翻Blog便能得到一路过来的痕迹。所以请不要怀疑我的置疑动机，而是认真想想我提出的问题。

我觉得全栈这个口号其实很毁人——别误解我的意思，我不是反对个人往全栈方向发展，而是想说，全栈并不容易，不是说前端开发的同学学了Node就全栈了，也不是说后端同学有了Angular和Bootstrap就全栈了，更不是说前端或者后端玩玩React Native，就能让iOS和Android下岗了。我发现一个可怕的事情是，全栈的口号让一些同学忽略了其他领域的知识深度，然后靠自我安慰甚至是自欺欺人地贬低其他领域的知识，以安慰自己已然全栈。

世界并不以某个人为中心旋转，前端在通过Node抢夺服务端的地盘的同时，服务端也在通过Angular和Bootstrap抢夺前端的地盘，各方在通过React Native想抢夺App的地盘时，你听说Swift (http://lib.csdn.net/base/swift)也在进击服务端了吗？做iOS开发的同学也在号称全栈了，而Android更是Java语言早就占领了服务端开发了。在这场互相抢夺地盘的战争中，每一方都号称自己全栈，可是啃下对方的地盘真的那么容易吗？玩票是可以的，但真要实战起来，会遇到一个巨槛——公司的职位定岗。

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。

到了移动时代，原来PC时代B/S结构一家独大的局面被打破了，iOS和Android异军突起，C/S结构将B/S结构压得死死的，Native App强势压制WebApp，一时间Web已死的论调甚嚣尘上。作为服务端开发，与Native合作天然就只能采用JSON接口——你让它套个iOS和Android模板试试？然后与前端合作怎么办呢？服务端当然可以选择继续套模板，但问题是，这样服务端的工作量就上去了，在服务端看来，iOS、Android和前端都是客户端GUI技术，这三个统一都叫前端都可以，无所谓B/S还是C/S，这三端的界面最好长得一模一样，功能迭代统一走，服务端统一只吐一套JSON数据出来。于是在移动时代，前后端分离终于有了契机。

有些同学稀里糊涂地以为前后分离是前端技术革命带来的，真不是。愿意的话，HTML4时代就完全可以做到前后分离了。技术造型除了技术本身，这里还有一个台面下的问题，就是不同团队的定职定岗问题。前端的同学学了Node就可以在公司做服务端开发了？你先问问服务端团队答不答应，出了问题你负责吗？就算你愿意负责，工作交给你前端来做了，服务端团队是不是可以裁员回家了？学了React Native就可以在公司做App开发了？你先问问iOS和Android团队答不答应。这些问题是忽略不掉的。之所以现在可以玩前后分离了，是因为服务端团队因为工作量原因，人家答应了呀。可是你玩Node服务端和ReactNative，能在公司里落地吗？你可以对外宣称，更少的人做更多的工作，可是这些方案在质量上，真的比得过人家专职团队吗？先把自己的坑填完再来挑战人家专职团队吧，人家可不会把你当做朋友的，你是来抢饭碗的。

那这些所谓全栈，能在什么环境里落地呢？很简单：
* 个人项目里
* 成熟公司里的非核心项目里
* 创业公司里，恨不得1个人当10个人用，重视成本重视速度，不重视也不懂质量问题的那种

全栈之风从哪儿吹来的？硅谷。硅谷有一堆的小创业公司，也有一些工程师能力极强的工程师文化公司。而你呢？你在恨不得一个人当多个人用的创业公司里，还是在Google、Facebook？全栈在公司定岗上天然会水土不服的。即便在美国，我相信除了硅谷，也一样很难有全栈的容身之地。当然，如果你仅想提高个人能力，多学多看绝对是好事，我是欣赏的，而且我自己也是这样做的。

最后再泼盆冷水——就算你在小创业公司里做着全栈，该来的问题还是会来。Twitter还是家小创业公司时，用的是时髦的Ruby on Rails，号称5分钟搭个博客系统出来的服务端框架。可等它做大，系统整个就重写了。

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 (http://lib.csdn.net/base/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。

关于Angular，后台，SPA

关于Angular，我是不认同的。Angular是Google服务端团队折腾的作品，整套代码组织的思路和服务端的MVC框架如出一辙：URL路由 + Controller + 数据抽象 + 模板引擎。虽然服务端出身的同学会倍感亲切，但这真的不是前端代码最好的组织方式。

我对此的判断是，Angular团队因为自身服务端出身的基因和思维模式，创造了一个对服务端团队最友好的框架，目前用户群体也是以服务端团队为主，适用场景以后台这种定制动效和定制UI要求不高的场景为主。事实上，Angular2决定使用TypeScript开发我也毫不意外，因为TypeScript本来就是Java服务端团队更喜欢的静态严格语法，Angular团队决定讨好他们自己讨好和他们相同背景的程序员群体，放弃普通前端工程师习惯的脚本弱类型习惯，也完全是情理之中的事。

唯一让我奇怪的是，我们的传统前端工程师为什么要去追捧这么一个框架呢？人家根本就没在意你好吗？就像从传统服务端转到Node服务端的同学会奇怪，为什么一群前端工程师对Node服务端这么感兴趣一样？这关前端什么事？我们去凑什么热闹？最近流行一个段子，说一个放羊的和一个砍柴的聊了一天，然后天黑了，人家放羊的羊吃饱了回家去了，可你砍柴的呢，你的柴呢？你的前端基本功呢？你的前台应用场景呢？

我觉得“全栈”这个词很讨厌，原因在于它搅混了很多事。从定岗上来说，后台的前端部分，到底是由谁来开发？是由前端工程师来开发呢，还是由服务端工程师自己来开发？在过去相当长的一段时间里，后台是怎么开发的？是由Frameset来组织代码的，并非SPA，后台的界面往往非常难看，基本上只是后端工程师自己顺手就给做了，没什么UI定制需求，也没什么动效，了不起有些正则表达式验证而已，连AJAX的要求都不高。前端工程师基本上是不做后台相关的工作的。那么如果后台并不是给自己使用的，而是给第三方使用的呢？是外包公司呢？后台界面如果需要有更好的卖相怎么办？Extjs就派上用场了，还算不错的界面设计和丰富的组件，而且完全支持SPA开发，只不过Extjs组织SPA是以组件为单位来组织的，并非服务端熟悉的MVC那一套。所以这种情况下，后台是需要依赖专业的前端工程师介入的。Angular最适用的场景应该就是这样的情况了，取代Extjs，让后端工程师自己按照自己的熟悉的方式进行“全栈”开发。

问题是，后台因为种种原因，从来也不是前端工程师的主战场啊。后端能自己搞定自己玩去嘛，前端工程师在实际工作中，更多的工作在什么地方，在前台啊，和视觉设计师的PSD直接打交道吧？和交互设计师或产品经理设计的交互动效直接打交道吧？要切页面，要玩透CSS吧？跟人后端工程师屁股后面追Angualr，结果人家Angular升级到2用TypeScript了，不兼容了，傻眼了吧？信不信人服务端过来的“全栈”比你一前端玩Angular玩的更溜？而且就算你辛辛苦苦把Angular那么复杂约束性那么强的一套框架学得特别顺了，可是别忘了你的工作主战场在什么地方？你是天天在写后台，当我没说，可是如果你天天在写前台，那是真心找虐……醒醒，你这砍柴的别陪人放羊的一起谈全栈了。

前台最适合以什么方式组织代码，下面我会再讲，现在先跟后端工程师谈谈心 ——以Angular来为后台组织SPA真的是最好的实践吗？和传统的Frameset相比，真的性价比更高吗？首先，我赞成从纯技术角度来看，Angular是适合后台场景的方案之一，只是即使是后台开发，也有三种技术方案可以使用：

- Frameset——一般的后台，能用就行了，样式，无刷新都不重要，反正给自己人用的，怎么快怎么来。开发速度快，需要前端配合，分工也容易，维护也容易，招人门槛、人流失进行交接都风险小得多。在技术选型之前，先想想你是否真的需要为后台提供SPA。
- Angular——给第三方使用的后台，对样式，无刷新体验要求较高的，而团队中又没有专职前端工程师，需要服务端工程师自己上的，结合Angular和Bootstrap是个不错的选择。只是Angular这一套的学习成本不低，而这一套其实又很难和普通前端前台需求普适，性价比不高。如果想用一套技术完美通吃前后台的前后端代码组织，你应该考虑方案3。
- 组件化组织，自己抽象——其实思路和Extjs一样，以组件化方式来组织后台的SPA。只是Extjs的学习成本也不低，而且样式也不好定制，所以你需要的并不是去学Extjs，而是学习如何自己封装组件。封装些Scene、SceneManager、Widget抽象类、组织个SPA出来很简单的，模块化组件化可维护性都不是问题。有同学会问了，那模板呢？通信呢？首先模板不是必须的，我个人不推荐在前端使用模板引擎，这在本质上和组件化是相悖的，就算真的要用模板引擎，有很多垂直的JavaScript模板引擎可以选择，通信也不用Angular那么麻烦，通信最主要是想解决一个什么问题？组件和组件之间解耦，但又需要保留组件内部的Context，很简单，通过全局的自定义事件，为事件进行传参，就可以轻松实现组件的通信了。多简单一件事，折腾出约束性这么强的一个框架出来，何苦呢？

关于React

2015年底我批了Angular，同时也批了React。但我反对它们的理由其实并不一样。

首先，React是组件化的思路，这个我是认同的。首先跳出Web前端这个圈子，咱们看看别的GUI技术，无论是桌面软件开发、Flex、移动还是游戏，都是以组件的方式来组织代码的，可以说GUI编程按组件来组织代码是普适的最佳实践。Web前端有两方面的原因，导致其代码组织方式出现了Angular这种非主流：

- Web服务端是唯一可以和HTML无缝结合的，这是为什么JavaScript、PHP、ASP又被叫做动态网页的原因。所以前端和服务端有扯不断理还乱的历史原因。看看服务端染指iOS或者桌面开发容易不？
- 前端的原生控件实在太少也太弱（看看.NET和Flex的控件），数都数得清楚，text、radio、checkbox、password、button、img、audio、video等，长期以来，前端不得不自己封装组件，Extjs、YUI、Dojo、Prototype无不如此，还有丰富的各种jQuery组件。成也封装败也封装，封装给前端带来了强劲表现力的同时，也降低了前端同学自己动手写组件的能力。大多数人都是拿来主义，但拿来主义并不是最可怕的，可怕的是拿来主义的同学没有意识到组件之于GUI的意义是那么不可动摇。

举个简明的例子说明Angular的MVC和组件化之间的不同吧：

Angular是什么思路：

```
{
m : [a_m, b_m, c_m],
v : [a_v, b_v, c_v],
c : [a_c, b_c, c_c]
}
```

而组件化是什么思路：

```
{
a: [a_m, a_v, a_c],
b: [b_m, b_v, b_c],
c: [c_m, c_v, c_c]
}
```

Angular为代表的类服务端MVC框架，上来直接分成M、V、C三个层，然后将互相关联的三块分到了三个层里去，这么干最大的坏处是破坏了抽象。而组件化的核心就在于抽象，将相关的属性方法内聚到一个组件里去，这是符合面向对象的思维模式的。我自己既做前端开发也做服务端开发，服务端开发在MVC框架的帮助下，基本上在应用层不需要做任何的抽象，Controller不过是函数而已，M是和数据库、缓存打交道的命令而已，即使有ORM抽象，那也是框架替我抽象好的，不用我自己做任何抽象，而模板引擎就更加机械化操作了。但我做前端开发时状态完全不一样，我需要通过组件化来将界面变成一个个自定义的组件，我需要不停地抽象：这是不是可以抽象成一个组件？它的属性有什么？它的方法有什么？它的DOM节点最外层容器是什么？我如果要封装方法该封装什么方法为公用，什么方法为私用呢？我需要不停地进行思考，因为前台的界面是各种各样的，不可能像服务端一样由框架高度抽象出一些通用逻辑进行封装的。事实上，前端除了少数的通用组件可以直接使用第三方的，比如模拟弹窗、富文本编辑器什么的，大多数的需求只能自己封装，这是件非常考验人面向对象思维能力的事。

所以，我认为前端的前台需求，不应该由重框架来提供解决方案，“轻框架 + 组件化 + OO”设计才是最优解。也就是说，前端不应该多提供约束性的框架，而应该提供基础工具帮助和通用组件就可以了，在应用层，要考验前端工程师自己的能力，没有任何框架可以帮你去抽象你的业务需求。所以我觉得前端比服务端苦逼很多。

那么我说的“轻框架 + 组件化 + OO”设计是不是之前没有业内的实践呢？当然不是。事实上，前端长久以来，一直是轻框架 + 第三方组件的模式。只不在过应用层，各家公司的前端工程师大多数能力水平很有限，而业务需求也并不复杂，所以在前端圈里OO设计和自定义组件并没有流行开来，究其原因主要是门槛有些高，而大多数
程序员们又自我要求很低。

听不明白轻框架 + 第三方组件 + 应用层面条代码？想想jQuery + jQuery组件 + 大多数前端工程师用jQuery写的代码你就明白了。

既然jQuery已然可以提供必要的帮助了，那么还要React干嘛？jQuery有一个致命的问题，就是jQuery本身是基于plug模式来写扩展的，对第三方组件并没有统一的更组件化的方式来标准化它们，这就直接造成了不同的人写jQuery组件会有不同的风格。这对团队合作并不是什么好事情，如果说是第三方组件，如果封装的质量很好，倒也罢了，毕竟只使用它的API就够了，内部实现再怎么乱也不用去计较。问题出在自家公司的自定义组件，如果公司里有几个人团队合作，一个人一个风格怎么行？我们需要一套标准来约束大家按相同的风格来组织代码。这就是React最大的意义所在，填补了jQuery的这个不足。但这是否意味着离开React就无法组件化了？当然不是，比如我的视频教程里就有另一套风格的组件化范式。

这套风格并非是我原创，而是从YUI3学来的，思路是可以穿越语言和框架的，我不过是将这套思路从YUI3带到了jQuery。事实上我是YUI的忠实粉丝，真的是座巨大的金矿，随着雅虎日渐衰落而停止了维护，真是让人遗憾不已。requireJs、React的组件生命周期，YUI早已在2008年就实现了，它的思想超出这个时代很多年。但这又如何？技术理念的先进有什么用呢？还不如会推广和上手门槛低来得实惠，我觉得YUI和jQuery相比，一个是航空母舰，一个是小渔船，但又如何？jQuery获了开源大奖，API上了犀牛书，成了事实上的工业标准，而YUI呢，谁还记得这个低调的大拿呢？

我不再追流行，也不看理念是否先进，而是重视工业标准，原因正因为我并没觉得有多少新东西，而且理念先进不等于接地气，不接地气不讨好普通的懒惰的程序员就很可能活不下来。可能因为受过太多伤，所以变得不再容易被挑唆，变得格外挑剔吧。有兴趣的同学可以看看我曾经写过的一篇文章，文章写于Flex在前端圈异军突起，无比时髦的2008年。现在看来，这些激情还是太幼稚了。《[RIA之争，我的看法]》一些喜欢说我不求上进倚老卖老的同学，想送给他们一句王朔的话：谁没有年轻过，可是你老过吗？

说回到React的问题上。我之所以不认同，并不是认为React在技术上一定是走错了路，而是因为React把简单的事情越搞越复杂，现在一提React就是React全家桶，它的野心很大，还想解决Native开发的问题，但经验告诉我，这很危险，这意味着学习门槛、团队合作成本、招人成本、人员流失风险。如果React并不能带来技术普及，只是昙花一现呢？谁来为公司里的那些遗留代码负责？谁又来为初出茅庐基本功都未扎实的同学浪费的时间负责呢？我脑子里时常想的是YUI、Dojo、Extjs这些过去前辈们的心血，就这么基本毫无痕迹地被遗忘了，而jQuery这小草根居然活下来了。谁能保证React这个套路下去，不是个短暂的狂欢呢？除了一些在社区里闹腾的同学，这些技术真的被广泛采纳和推广下去了吗？以我的经验，深表怀疑。

再强调一下，对于个人要学习，没有任何人会阻拦你，而且学习是对的。问题是，别轻易在团队里强推，想清楚在公司里推广起来坑多还是利多。我总是习惯性站在一个技术负责人的立场上来看问题，难免保守，我只是希望激进的同学理解，公司并非个人的实验田，你不开心你会考虑换工作，但你留下的摊子，还得有人收拾。所以我想，冲突的根本应该就在于此吧。

React总的来说，不像Angular走错了路，但因为全家桶的原因门槛越来越高，这不是个好现象，未来有待观察。个人不是太看好，但确实也没有更有力的挑战者。要么像我一样坚守jQuery，自己做组件抽象类，配合jQuery的第三方组件工作，等待更有冠军相的角色出现，代替jQuery成为新工业标准，要么就小心翼翼用上React好了，我也想不到有什么别的路可走，当下这阶段确实挺邋里嗲尬的。另外，别小看jQuery，我认为jQuery仍然非常坚挺，而且仍然有非常大的可能性成为最后赢家——补上组件抽象类和单向数据绑定就可以了。而这些通过jQueryui都不难做到。

组件化是对的，但组件化框架其实是很轻量级的，用不用框架，以及用什么框架问题都不大，不是说有了什么组件化框架，就像奢龙刀一样开了外挂了，不是的。组件化不过是编写高质量代码的基础而已，真正的挑战在抽象上。能不能养成面向对象的思维模式才是关键，而面向对象要具体去抽象组件时，是没有定式的，要看应用层具
体的需求去做抽象，这也是为什么我说前端不该去做约束性特别强的框架的原因所在。轻框架，重应用层抽象才是GUI编程的正途。

关于SPA和Web Site

说到Angular、React之类，很多同学都会提到SPA和Web App。然后说SPA的春天来了，说代码质量要提高了，jQuery可以去死了。但事实上，真的如此吗？

后台是否搞SPA我其实并不关心，搞不搞都可以，搞的话我也不打算用Angular来做，服务端的全栈们如果要用Angular搞后台我也没意见，只是你自己开发的自己维护，别你做的技术选型硬拉上我。但对于前台部分，我期待SPA其实很多年了。从我用Flash做RIA应用时，我就知道前端做SPA比起Web Site要复杂多少了。我2008年开始写HTML的SPA，早期做的一个很复杂的例子，现在还有保存，请见：http://www.adanghome.com/tbs/manage.html (http://www.adanghome.com/tbs/manage.html)。真的很复杂，当时通宵了好几天。我从2009年开始，做的前端程序基本全是SPA，我很早就在一些技术大会上讲，前端应用时代会来临（当时还没有SPA这个叫法），模糊C/S和B/S的产品形态边界，同学们要抓紧时间提高自己，不要到时被淘汰了。但我等了很多年，也没迎来SPA的大爆发。Web Site一直是主力产品形态，根本就等不来SPA时代。今天很多同学说SPA说得很热闹，事实上很可能只是这些同学的一厢情愿。

事实上，SPA时代等是等不来的。一个可悲的事实是，工程师在公司里的定位都是执行层，而非策划层，策划是上游，对应的岗位是公司老板，最次也是产品经理，不是工程师。而产品经理们会什么呀？他们身上没有技术基因，不会主动想到Web前端可以做新的产品形态的。如果产品层面没需求，你怎么等来SPA时代？只有一个可能
性能等来，就是国外有一个非常成功的项目，是用SPA做的。那么国内很可能从老板到产品经理会跟风去抄袭一个。也就是说，国外的老外们不做出点什么产品创新，我们国内的工程师就很难有机会有生产环境上做些SPA的富应用了。有些同学拿了几个简单的页面给我看， 到了AJAX无刷新，然后display：none和block实现了场景切换，跟我说，看，我们SPA了。其实真不是。在我看来，SPA真正的威力不是把多个网页变成单个网页，而是在产品形态上就彻底跳出网页般的排版布局。比如说做游戏、做网络IDE、做网页版PhotoShop之类。而这，需要的不仅是技术能力，而进一步需要产品设计能力，能不能技术驱动产品创新，不要再折腾什么技术工程化，发点力在技术产品化上。如果Google的工程师不用AJAX做Gmail，网络邮箱是不是会一直是Frameset方式？如果Google不做Google Map，是不是我们就一直没有网页地图可用了？我们自己能不能主动想想Websocket、Canvas2d、WebGL、CSS3之类的东西能不能做些和HTML4时代不同的产品出来？我举个例子，带图画板的网络聊天室怎么样？基于WebGL的Mmorpg怎么样？在同一个页面上打通DOM和WebGL会怎么样？我再留一个问题给各位思考：Canvas可以实现截图，toDataURL方法嘛，那么DOM能不能通过技术手段实现截图功能？给个提示：遍历DOM节点，然后做z轴上的排序，然后一个一个将他们画到一个看不见的Canvas上去……

几年前我就一直置力于技术驱动产品创新，号召圈里的同学们多投入点精力在技术产品化上，不然HTML5那么多牛逼的功能都白瞎了，说是进入了HTML5时代，可是产品形态依然没有啥变化，身为前端工程师我觉得很耻辱。可是，我一个人的能力实在有限，我努力了几年，我也号召了几年，然后终于放弃了。现在我已经死心转型做产品了。相信我写了这样的文章，看官看完之后，依然会该干嘛干嘛。

关于SPA，再多说两句，就是SPA的代码组织方式不用围绕URL路由来组织，如果是个纯单机应用没有服务端，是不是代码就没法组织了？当然不是。路由是好东西，但我推荐用自定义事件来进行路由，不要用URL。这里有个我写的斗地主单机小游戏，有兴趣的同学可以看下：http://www.adanghome.com/js_demo/29/ (http://www.adanghome.com/js_demo/29/) 。

最后，SPA什么时候可以迎来大爆发？我反正挺悲观的，我等了6、7年没等来。现在说得热闹，可是没见着几个SPA的强产品需求。没有产品层面的强需求，其实意味着折腾来折腾去，不过是前端圈自己在玩，孤芳自赏，以及面试时为难应聘者，跟前端面试时面算法 (http://lib.csdn.net/base/datastructure)似的。

React Native和PhoneGap

再说说React Native。我不看好这种方式，阉割版的CSS导致前端技能的受限，对Native底层的黑盒导致调试和扩展的困难，另外“learn once write anywhere”的性价比也并不高。而且，公司定岗的原因，iOS和Android团队也绝对不会对你友好，就像前端学了点Node服务端去挑战Java、PHP后端位置一下。于技术，于团队，真的是坑多于利的技术。

相比之下，PhoneGap那种思路，GUI交给Webview里的H5，底层交给iOS和Android，Native和H5通过jsBridge通信，是团队合作成本最低的方式，而且只要jsBridge写得好，一套HTML5代码也是最容易实现“write once run anywhere”的。三端的界面可以共用一套代码。我更倾向于后者。

当然，一个不容忽视的问题是，webview当前的性能还有严重问题，如果交互效果复杂一些，在低端机上的表现就会卡顿，所以采用这种技术方案时，受限于性能，产品的设计要尽量简单。

当下并没有完美的解决方案，但各方面原因考虑下来，hybird仍然是更好的选择。React Native唯一能拿出来说事的只有webview性能不好这一点，只是如果webview的性能问题在未来得到解决了呢？记得摩尔定律吗？所以说React Native不过是个不怎么样的临时解决方案而已，一旦webview的性能问题得到质的变化，React Native就没什么存在价值了。当然，话又说回来了，webview性能问题什么时候能够被解决呢？也没那么乐观，反正我从2011年等到现在也没等到。iOS其实已经很不错了，Android的碎片化是个深坑。

对了，我徒弟就用HTML5做了个SPA，用hybrid方式包成了iOS和Android的App，叫做“健康日记”（一个致力于帮助你养成健康规律生活习惯的App）。在Android下确实被碎片化折腾死了，但开发团队确实成本很低。感兴趣的同学可以下载试试。

关于微信小程序

今年一个很热闹的词就是微信 (http://lib.csdn.net/base/wechat)小程序，从上半年张小龙宣布应用号在筹备，到下半年小程序SDK掀开神秘面纱，小程序着实牵动了很多人的心。

很多人稀里糊涂说微信小程序是H5技术，其实看完SDK我发现这跟HTML5没有半毛钱关系。微信小程序和之前百度、UC、QQ浏览器、Chrome Web Store、Firefox OS的Webapp完全不同。作为一个开放平台，不知道微信为什么放弃百度轻应用的思路，设计这么个SDK。坦白讲，这么设计SDK讨好不到任何人，Native开发的同学并不熟悉，而前端开发的同学也会觉得怪怪的很受限制。最重要的是，HTML5代码没法简单适配一下移植过来，还得重新开发一套。也就是说，这是一个伪HTML5的技术，开发的同学又多了一个平台要侍候。

我猜微信平台的同学可能会说，这是出于性能上的考虑。但我只能说，店大欺客，仗着用户量高，对开发者真不友好。做“健康日记”时，本来打算借应用号的东风的，希望在应用号发布上简单适配一下就上微信的，结果等了半年等来这么个SDK，一看程序完全没法移植，要完全重新开发一套出来，只好放弃了。

就仗着用户量，iPhone不也活活逼死了Flash吗，你牛。

关于前端的缺人和高薪水

我工作十年以来，前端圈前所未有如此盛世，技术圈也热闹，薪水也涨得飞快。但不得不说，技术圈的热闹和薪水涨得飞快并没有关系，仅仅是个巧合。

技术圈的热闹，原因在于跨界和伪全栈。薪水涨得飞快，原因在于HTML5的应用场景变多了，和技术没有关系。

说说HTML5现在的应用场景吧：

- PC端浏览器
- 移动端浏览器
- 超级App的Hybrid
- 微信公众号这种App开放平台
- 微信朋友圈微博的营销页面
- 百度轻应用（很小众）
- 后台（部分公司）
- 微信小程序（姑且也算）

特别需要注意的是“超级App的hybrid”、“微信公众号”和“微信朋友圈微博营销”这三点，在移动互联网之初的2011、2012年，C/S结构的App是主流，Web已死是主基调，这两年前端的日子不算好过。然后随着淘宝、京东这种超级App越来越多，C/S结构更新麻烦，特别是iOS还需要在App Store审核的问题越来越放大，超级App不得不选择Hybrid的方式来进行发布和团队合作。而随着App也提供开放平台，Hybrid是唯一的选择（直到微信小程序打破它），前端的需求量突然大增，供不应求，于是前端长久以来一直被严重低估的局面终于得以扭转，薪水一路水涨船高。这种局面其实在2011、2012年的Android和iOS圈里也发生过，然后培训机构大量提供Android和iOS培训，再然后供求关系得以平衡，Native工程师的薪水恢复到了正常情况。有需求就会有供给，值得注意的是现在培训机制也在开展HTML5培训了，等到HTML5工程师被培训机制批量提供出来，前端的好日子也就到头了。

到那时，风就停了，猪就会摔下来。所以不要稀里糊涂地被技术圈的大跃进冲昏了头脑，你学会了两个时髦工具，面试官也在面试时问到了你时髦工具，然后你也顺利到了很高的薪水，就真以为是工具帮了你，你的能力真的就高了。不是的，是时代的原因。那个坐你旁边工作了好几年的一声不吭的老Java工程师薪水没你这个小毛头高，并不是人家水平没你好，并不是人家工作技术含量低，并不是人家比你笨，仅仅是因为做Java开发的太多了，市场饱和度高，仅此而已。所以偷着乐就可以了，然后赶紧补基本功。

写在最后

我曾经写过一篇文章，叫“置疑精神”。所以也请读者带着置疑精神来看我的这篇文章，认为有道理的，就听，认为不对的，保留你的疑问，不要迷信任何权威，这是技术人该有的美德。虽然我写了这么多，但也有可能，全是错的。

原文地址： blog.csdn.net (http://blog.csdn.net/hwhsong/article/details/53639833?utm_source=ourjs.com)

[非英文:20039, 总字符:26139]

社区评论 (Beta版)

#0 秦与汉 1天前

有怀疑的态度是非常正确的。

回复

#1 盛归危 23小时前

让自己走得更远，走得更有价值，走得更有可能性。

回复

游客 侯毕丁

发布评论

访问404页面，寻找丢失儿童 (/404)

热门文章 - 分享最多

1. 在Debian上安装Nginx并搭建一个最简单的静态网站服务器（以OnceAI为例） (/detail/5834f6716345657c15b8db95)
2. NodeJS教程：基于OnceIO框架实现文件上传和验证 (/detail/583e1a404edfe07ccdb23426)
3. NodeJS中的客户端缓存、浏览器缓存、304缓存和OnceIO的缓存控制 (/detail/581d26b271e01c68e9619152)
4. NodeJS中的Middleware是什么?在OnceIO中创建和使用中间件 (/detail/580ac5de71e01c68e9619119)
5. OnceIO(NodeJS)中的服务器端缓存、模板预加载和静态资源文件的缓存和Gzip压缩机制 (/detail/5823beaf71e01c68e961915a)
6. OnceIO(Node.JS)中的模板引擎是什么及MVC设计模式的使用与实现 (/detail/580ef56371e01c68e9619124)
7. OnceIO(Node.JS)的静态文件路由(app.static) (/detail/581afd1e71e01c68e961914c)
8. OnceIO(Node.JS)的网页(模板)的引用与嵌套 (/detail/582a5d9e71e01c68e9619161)
9. OnceIO(Node.JS)服务器端Cookie设置、添加、删除、显示及其实现原理 (/detail/5830f4a071e01c68e961916b)
10. OnceIO(Node.JS)的路由(Routing)、路由方法和路由变量 (/detail/5819cac971e01c68e9619149)
11. AirJD-简单好用的免费建站工具 (http://web.airjd.com)

相关阅读 - 大话编程

1. 在Debian上安装Nginx并搭建一个最简单的静态网站服务器（以OnceAI为例） (/detail/5834f6716345657c15b8db95)
2. OnceIO(Node.JS)的静态文件路由(app.static) (/detail/581afd1e71e01c68e961914c)
3. 乌云和漏洞盒子停业整顿—可能与国内“白帽子”黑客圈高度关注的“袁炜事件”有关 (/detail/579067d488feaf2d031d2588)
4. 如日中天的Uber究竟使用什么开发语言？ (/detail/5716d62688feaf2d031d24d6)
5. 微软降低OneDrive的免费存储空间容量，个人云存储热度减退 (/detail/570763c888feaf2d031d24c3)
6. 一个程序员是如何搞挂NPM和Node社区的 (/detail/56f47f1b88feaf2d031d24a9)
7. 程序bug导致了天大的损失，要枪毙程序猿吗？ (/detail/56f2212788feaf2d031d24a6)
8. NodeJS初学者教程：Node.js之HTTP (/detail/56dfb29588feaf2d031d2488)
9. 谷歌人工智能AlphaGo挑战人类围棋冠军 (/detail/56cd331988feaf2d031d247b)
10. 美国程序员低价雇中国人替其编程被解雇 (/detail/56c4244588feaf2d031d2473)

关键字 - 分享

1. wemall团购版商城系统商城源码 (/detail/5854eb614edfe07ccdb2344c)
2. 2016年前端技术观察 (/detail/5853d91e4edfe07ccdb2344b)
3. 前端开发人员所必备的十大技能 (/detail/58511b1e4edfe07ccdb23449)
4. Aspose.Cells for .NET入门教程合集【连载中】 (/detail/5851112a4edfe07ccdb23448)
5. DevExpress官方正式公布16.2版本更新发布会时间 (/detail/584fa99e4edfe07ccdb23447)
6. 数据库管理，2016热门产品推荐 (/detail/584f8d884edfe07ccdb23446)
7. Zend Studio使用教程之在Docker容器中调试PHP Web应用（三） (/detail/584f54564edfe07ccdb23444)
8. Python成为美国大学第一流行的教学编程语言 (/detail/53bcea879a908d3107000001)
9. 流行算法类软件盘点（一）：混合整数线性规划(MILP)解算器lpsolve (/detail/584e787f4edfe07ccdb23443)
10. dotConnect for Oracle 更新至v9.2，新增程序集添加复选框，EF支持升级|附下载 (/detail/584a7dc24edfe07ccdb23442)

欢迎订阅 - 技术周刊

我们热爱编程, 我们热爱技术; 我们是高端, 大气, 上档次, 有品味, 时刻需要和国际接轨的码农; **欢迎您订阅我们的技术周刊**; 您只需要在右上角输入您的邮箱即可; 我们**注重您的隐私，您可以随时退订**.
加入我们吧! 让我们一起找寻码农的快乐，探索技术, 发现IT人生的乐趣;

关注我们

我们的微信公众号: ourjs-com
打开微信扫一扫即可关注我们:
IT文摘-程序员(码农)技术周刊