前端之巅

GMTC 全球大前端技术大会

主办方 Geekbang. 极客邦科技 InfoQ

卷首语

2019，大前端路在何方?

作者 覃云

一年又一年，2019 年 GMTC 北京站又要开始了，依照惯例，每年 GMTC 我们都会出一本迷你书，今年也不例外。今年，我们精心挑选了 5 个不同方向的文章，希望能为开发者指点迷津。

犹记得去年这个时候，在轻应用的战场上，微信小程序仍独领风骚，然而，没过多久，支付宝、百度、头条小程序开始崭露头角，在不到一年的时间内，迅速发展成为微信小程序的劲敌，轻应用的战场一时间硝烟弥漫。于此同时，各种小程序跨端框架也应运而生，如京东凹凸实验室的 Taro、滴滴的 Chameleon 等，这些框架不仅能支持多种小程序，还支持手机厂商联合推出的快应用，名副其实的“跨多端”，但对开发者来说，究竟应该选择哪个框架才是正确的？为此，有开发者专门对国内 5 个主流的跨多端框架进行了全方位对比，希望能给正在做技术选型的你一些帮助。

不仅轻应用的战场打得火热，三大框架之争也一直都没有停歇过，在下载量上，React 一骑绝尘，而 Vue 远远比不上 React 和 Augular，但值得一提的是，在过去一年，Vue 在 GitHub 上大爆发，在 Star 数上，Vue 成为三大框架之首，再加上具有革命性的 Vue 3.0 即将发布，Vue 的未来无可限量。

不仅如此，编程语言之间也一直在 battle，虽然 JavaScript 在前端领域仍是霸主，但 TypeScript 的潜力也不容小觑，作为 JavaScript 的超集，TypeScript 为大型应用开发而生，它在过去一年已经成功俘获 Vue 的“芳心”——Vue 3.0 将用 TypeScript 重写，伴随着 Vue 的崛起，相信在未来几年 TypeScript 也会迎来新的爆发。

说完了技术，我们来谈谈前端工程师的成长，前端圈最流行的一句话是“求别再更新，学不动了”，不可否认，前端技术的迭代速度的确很快，但正如我去年所说的逆水行舟，不进则退，无论你在哪个行业，都需要不断学习。前端领域的内容多而杂，怎样在每天的工作和学习中不断前进，是每个前端 er 不断思考的问题。最后，引用张鑫旭在文章里讲的一句话送给大家：“心有猛虎，细嗅蔷薇，不为外物蒙双眼，不因碎语扰心智，一步一步，有条不紊，耐心前进，人生是条长河，希望能有双健康有力的双腿，带我看到尽头最美的风景。”

目录 | CONTENT

GMTC特刊

本期主编 覃　云

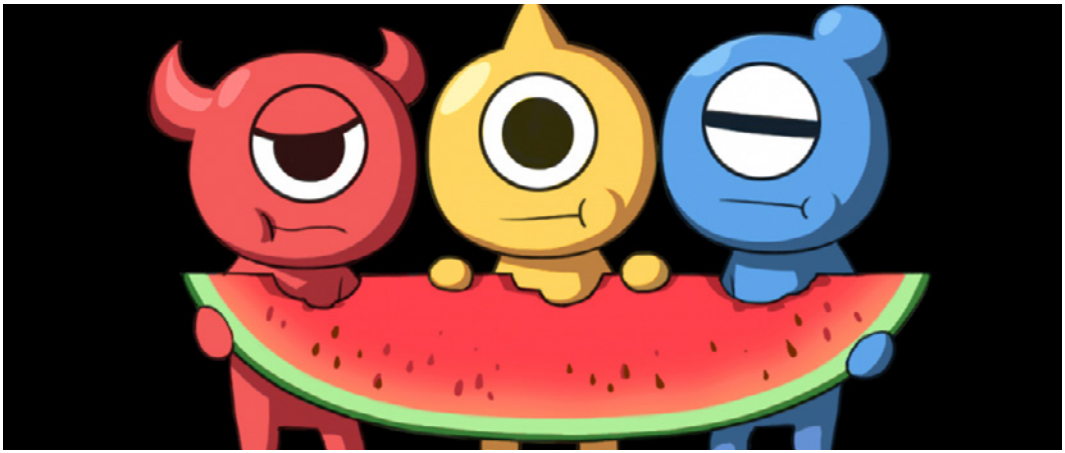
流程编辑 丁晓昀

发行人　霍泰稳

GMTC 全球大前端技术大会是由极客邦科技旗下 **InfoQ** 中国主办的技术盛会，关注前端、移动、**AI** 应用等多个技术领域，促进全球技术交流，推动国内技术升级。**GMTC** 主要面向各行业对前端、移动开发、**AI** 技术感兴趣的中高端技术人员，大会聚焦前沿技术及实践经验，旨在帮助参会者了解大前端 **&** 移动开发领域最新的技术趋势与最佳实践。

前端 2018 解读
小程序大战、框架之争，吃瓜不停

作者 徐川 覃云



以“娱乐圈”著称的前端圈，在 2018 年一如既往地热闹，而我也接手前端之巅一年了，与大家一同见证前端这一年的变化。在下文中，我将为大家做个盘点，细数 2018 年前端圈发生的大小事，请搬起小板凳，准备吃瓜吧。

一、小程序

2017 年 1 月 9 日，张小龙通过微信公开课，把小程序带到大家的生活中，从那以后，小程序战场陆续引来支付宝、百度、今日头条入局，同时，手机厂商大概是看到了小程序对其应用商店的威胁，小米、华为、OPPO、vivo 等九大国内手机厂商联手成立了“快应用联盟”，以快应用死磕小程序，从这可见小程序发展势头之迅猛与前途之无量。

微信小程序

在 2018 年年初的微信公开课 PRO 上，张小龙曾说，小程序是微信期望最大的项目，并表示对微信小程序充满信心。

但是，从媒体报道里，我们却看到有投入到小程序的创业者，在尝试过后退出，小程序并没有带

给他们如 iOS 和 Android 平台这样的红利窗口，小程序的留存和变现，始终是萦绕在开发者心头的问题。

当然，除了“坏消息”，微信小程序还是给开发者带来好消息的。今年 7 月，在微信公开课微信小程序技术专场上，微信公布了面向开发者的技术规划，重点是微信小程序将支持 NPM、小程序云、可视化编程、支持分包等，其中小程序云开发弱化了服务端，可以说是开发者的福音。

百度智能小程序

在 2018 百度 AI 开发者大会上，百度智能小程序宣布正式上线，上线两个月后，官方宣称智能小程序月活破亿，12 月底，月活破 1.5 亿。与其他小程序不同的是，智能小程序核心框架已在 GitHub 开源，并联合其它公司打造开源联盟，提供 SDK，可以集成在其它 App 里，这样，其它 App 就可以打开任意百度小程序，百度试图通过这种方式来扩大它的小程序的使用范围。

支付宝小程序

支付宝小程序被视为“蚂蚁金服未来三年最重要的战略之一”。今年 9 月，在经历 1 年的公测期之后，支付宝小程序宣布上线，与微信小程序相似，支付宝小程序只能在阿里的生态中使用，截止到今年 9 月，官方数据显示，支付宝小程序已经有 3 亿用户，支付宝小程序平台的小程序数量超过 2 万。

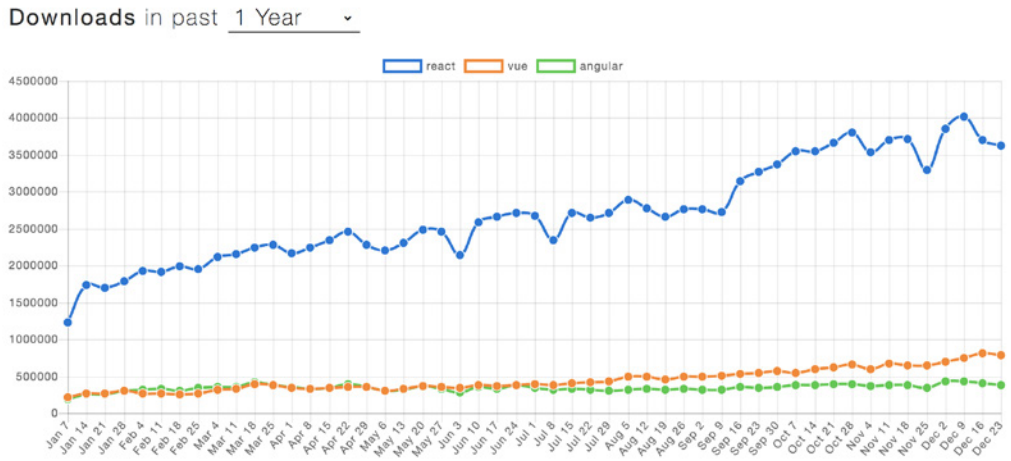
但开发者最关心的支付宝小程序技术问题，除了官方文档，支付宝并没有像前两个小程序一样，向外界透露太多，公开的资料也寥寥无几。

头条小程序

与其他几个小程序的大肆宣传不同，头条小程序十分低调。我们只知道头条在今年 9 月发布了小程序，11 月正式上线，但只对部分企业开放，而且只针对 Android 系统，面向开发者的信息基本为零。

这四家小程序各有特点，微信手持封闭大流量，百度强调开放与生态，支付宝背靠电商大平台、头条依靠内容变现，随着各家小程序逐渐成熟，相信到 2019 年，小程序的战场将更加激烈。

二、三大框架



GitHub Stats

	stars	forks	issues	updated	created
vue	123218	17613	209	Dec 27, 2018	Jul 29, 2013
react	118438	21480	521	Dec 26, 2018	May 25, 2013
angular.js	59321	28999	490	Dec 24, 2018	Jan 6, 2010
angular	43830	11290	2628	Dec 27, 2018	Sep 19, 2014

上图分别是 2018 年 React、Vue、Angular 在 npm 上的下载量及其在 GitHub 上的表现。

不难看出，2018 年，Vue 在 GitHub 上的 star 数虽然超越了 React，但是在 npm 上，React 一尘绝骑，独领风骚，相反，Vue 与 Angular 就差了很多，在这一点上，Vue 的表现出于我的意料。不过，这里只统计了 NPM，由于 NPM 和 Yarn 的托管仓库在国内较难访问的原因，国内通常使用国内镜像源，所以这个数据并不完整。尤雨溪在知乎上提过，更准确的统计方法是统计他们的 Chrome dev tools 插件的用户数，截止到发稿时间，React 的这个数据是 126W，Vue 的数据是 67W，所以 React 和 Vue 的差距并不如图中那么大。

下面让我们再来看看这三大框架在 2018 年都有哪些动态：

React

2018 年，从 3 月份发布的 v 16.3 到现在的 v 16.7，React 一共发布 5 个大版本，其中最受关注的主要有两个特性。

- React 16.6 引入了用于代码拆分的 Suspense，Suspense 是指 React 在组件等待其他事件发生时“暂停”渲染并显示加载指示器的新功能。
- React 16.7.0-alpha 引入了 Hooks，Hooks 让你可以使用功能组件的状态和生命周期等特性，还可以在不引入额外嵌套的情况下在组件之间重用有状态逻辑。

根据 FB 发布的 React 2019 年路线图，React Hooks 将在明年 Q1 被引入 React 16.7 正式版中，同时 React 16.8 将引入并发模式，v 16.9 将上线数据获取的 Suspense 的特性，令人期待。

Angular

Angular 今年更新了两个大版本，分别是 5 月份发布的 Angular 6 和 10 月份发布的 Angular 7，其中 Augular 6 新增了 ng update、ng add 两个非常实用的功能，Angular 7 则带来了 CLI Prompts、虚拟滚动和拖放等特性，最重要的是，angular 7 还带来了 Ivy 的消息，它是 Angular 新的渲染引擎，使 Angular 的编译速度更快、更简单。

尽管 Angular 迭代速度上比其他两个框架快，而且性能也有了很大的提升，但是从之前 Stateofjs 对 2018 年 js 生态调查结果可以看出，Angular 对开发者的吸引力仍不够，Google 要加油了！

Vue

2018 年是 Vue 大火的一年，今年 6 月底，Vue 在 GitHub 上的 star 数反超 React，同一周，Vue 和

React 的 star 数双双破了 10W，这在前端圈算是一件里程碑的事件了。

2018 年 8 月，Vue CLI 3.0 发布，尤雨溪表示 Vue CLI 经历了重构，目的是尽可能减少现代前端工具在配置上的烦恼。

2018 年 Vue 虽然没有重大版本更新，仍停留在 v2.x 版本，不过尤雨溪公布了 Vue 3.0 的开发路线图，Vue 3.0 将会经历重构，代码库也会用 TypeScript 编写，Vue 核心将会变得更小、更快、更强大。

虽然 Vue 的市场份额还比不上 React，但是 Vue 凭借其出色的特性广受开发者喜爱。Stateofjs 的数据显示，在三大框架中，Vue 的潜在用户是最多的，而且，Vue 由于门槛较低，更受初学者的欢迎，这点优势也会随着初学者的成长而慢慢扩大。

三、编程语言

JavaScript

根据 ECMAScript 2018，JavaScript 在过去的一年中，主要增加了正则表达式的“.”标志、对象的剩余属性、对象的延展属性、异步迭代等十个特性。

值得一提的是，2018 年 10 月，TC39 在 GitHub 上还通过了一条 EMCAScript Class field 语法特性的草案，即类私有属性修饰符“#”，这个特性已经在浏览器中实现。这个特性在社区里激起了一些争议，更是暴露出了 TC39 委员会缺乏与社区间沟通的弊病。

Dart

作为 Google 的亲儿子，以及 Flutter 唯一支持的语言，Dart 一出生就自带光环，然而早期定下与 JavaScript 正面对抗的目标被证明是失败的。2018 年 2 月，Google 宣布 Dart 重启，Dart 2 的目的有三个：优化语言开发，增强 Google 对 Web 和移动框架的支持能力，将 Google 内部的一些支持 Dart 的工具和组件带给外部的开发者。

TypeScript

2018 年，TypeScript 发布了 3.0 版本，3.0 虽然是个大版本，但并没有包含太多重大的突破性变更，这说明语言已经逐渐成熟。随着 TypeScript 在开发者间的流行，以及如 Deno 这样的重磅项目默认支持 TypeScript，TypeScript 在未来甚至可能会威胁到 JavaScript 官方版本 EMCAScript 的地位。

四、跨平台开发

React Native

2018 年 6 月，React 博客宣布 FB 将要重构 React Native，使其更轻量，更适应 JavaScript 生态圈的发展。然而，不到一个月，国外公司 Airbnb 宣布放弃使用 React Native，8 月，在线教育机构 Udacity 移动团队也宣布从 App 中移除了使用 React Native 开发的最后一批功能，接连被大公司弃用，这给 React Native 的社区带来沉重打击。

React Native 的问题是综合性的，如果 Facebook 不能倾注更大的力量和资源到 React Native 上，

重构也将不能改变它逐渐沦为小众的命运。

Flutter

自 Flutter 在 2017 年 I/O 大会推出以来，就吸引了很多移动开发者的目光。在 2018 年的最后一个月，Flutter 1.0 终于正式发布了，除了 iOS 和 Android，Flutter 还将支持 Web 和桌面开发，1.0 的发布意味着 Flutter 开始迈向成熟，以后会逐渐走进开发者的业务中，而更多的应用场景说明了 Flutter 的野心：成为终端界面开发的终极解决方案。据了解，目前，国内阿里、京东、美团、腾讯都已经在尝试使用 Flutter，随着 1.0 正式版的推出，明年 Flutter 将会获得新一轮的爆发。

Weex

2018 年，Weex 几乎从开发者的视线中消失了，不多说了。

Electron

Electron 在 2018 年的下半年接连更新了两个大版本，分别是 v3.0 和 v4.0，主要是更新了 Node 和 Chromium 的版本。

Electron 在与 nw.js 的竞争中可以说已经取得了胜利，但在 2018 年它也受到了一些挑战，一个是 Google 推出的 Carlo，基于 Puppeteer 使 Node 程序和 Chrome 互相通信，从核心原理上与 Electron 是一致的；另一个是 Flutter Desktop Embedding，如果 Flutter 真的成为主流，那么 Electron 地位的确危险了。

PWA

2018 年初，苹果在 iOS 11.3 中引入了 PWA，这使得 PWA 跨平台开发成为可能，10 月，Chrome 70 的 Windows 端支持 PWA。PWA 中的 Service Worker 使用的人逐渐增多，但由于各个平台仍然存在一些兼容问题，完整的 PWA 并未得到大规模的应用。

五、前端大事件

deno issue 事件

2018 年 5 月，Node 之父 Ryan Dahl 发布新的开源项目 deno，根据官方文档的介绍，deno 的性能优于 Node，它也因此被认为是下一代 Node，这本是一件令人开心的事，但没想到中国开发者竟然跑到 deno issue 中刷屏，如“求别更新了，老子学不动了”，并指责 ry 对 Node 不负责，让其中断 deno 项目的开发。最后，ry 不得不关闭了 deno 的 issue，相关 issue 也被删除了，可以说丢人丢到国外去了。

看到这里，我忍不住想说，每次一发技术更新的文章，请不要再在微信后台留言“求别更新了，老子学不动了”之类的话，没用，因为你们都逃不过“真香定律”。

GitHub 重构页面移除了 jQuery

2018 年 7 月，GitHub 页面改版，与此同时，其前端团队还乘机移除了页面中的 jQuery，值得一提的是，GitHub 前端团队并未使用其它框架来代替 jQuery，而是使用原生 JS，具体原因 GitHub 也给出了说法。这也说明曾风靡一个时代，也是前端开发者得心应手的武器库 jQuery 渐渐没落了。

周下载量过 200 万的 npm 包被注入恶意代码

11 月底，npm 下载量超过 200 万的 package 被注入了恶意代码，黑客利用该恶意代码访问热门 JavaScript 库，目标是 copay（开源比特币钱包）及其衍生产品的用户，以此窃取用户的数字货币。

这个被注入恶意代码的 package 名为 event-stream，它是一个用于处理 Node.js 流数据的 JavaScript 软件包，由于 Angular、Vue、Bootstrap、Gatsby 等项目在使用这个包，因此都受到了影响。

微软宣布桌面版 Edge 将基于 Chromium 进行开发

2018 年 12 月 7 日，Windows 副总裁 Joe Belfiore 在 Windows 官方博客上正式宣布桌面版 Edge 将基于 Chromium 进行开发，以减少 Web 开发生态的碎片化，为用户提供更高的 Web 兼容性。

Joe Belfiore 说，在过去几年，微软积极参与开源软件（OSS）社区的建设，也因此成为世界上最大的 OSS 项目支持者之一，他强调，今后，微软将成为 Chromium 项目的重要贡献者之一，这不管是对 Microsoft Edge 还是其他浏览器，都具有重要意义。

阿里 Antd 代码彩蛋事件

12 月 25 日，部分开发者突然发现他们开发的 Web 网页的界面发生了变化，按钮上方出现“积雪”，经过探索发现这是前端 UI 组件库 Ant Design（简称 antd）提前埋入一个未经声明的“彩蛋”，并且没有下线机制，引起了巨大争议。

由于前端开源代码缺乏商业化元素，让一部分人认为随意修改代码并没有责任，对于一些个人的小型项目来说这么说并没有错。但是 antd 作为企业级开源项目，项目的维护者应该抱有更加严谨的态度，毕竟能力越大，责任也越大。

六、2019 年预测

对于前端的未来，有些很好预测，有些却没那么明朗，笔者这里不负责任的做一些大胆的预测，读者可以自行判断。

首先，Hooks 将改变前端框架代码组织的方式，而 2019 年我们很快就能用上了。

然后，GraphQL，国外很多人都看好它，还有一些大公司如 Airbnb 在尝试使用，但在国内推进极难，毕竟涉及到前后端的改变，而国内前端在系统设计中的话语权太低，难以影响技术选型，这个除非在国外已经非常流行，国内后端主动要用，否则感觉难以普及。

再然后，小程序，小程序是国内独有的产品形态，由微信主导，但由于微信的克制，我们很难期待小程序的突然爆发，过不久就是微信公开课 Pro 了，看张小龙这次会放出什么新玩法吧。

Flutter 2019 年会迎来爆发，有人说，React Native 好几年都没有 1.0，Flutter 现在就已经 1.0，所以更相信 Flutter，无言以对。不过，联想到离我们越来越近的 Fuchsia 以及它和 Flutter 的关系，学习 Flutter 的确可以算未雨绸缪。

React Native 上面已经分析过了，重构只能解决一部分问题，更重要的是挽回开发者的信心。

PWA 目前还是很难和原生 App 竞争，如果一直这样下去，只是在技术上做文章的话，仍然很难发展起来。但如果 PWA 能进入 App Store 或者 Google Play，那就是一个 Game Changer 了，这个

2019 年会发生吗？我觉得悬。

WebAssembly 看着是前端领域发生的事情，但实际上是给那些 C++、Rust、Go 语言等的开发者用的，目前来看是雷声大雨点小，从它最有可能的应用方向，游戏来说，它的桥接性能太差，除非整个游戏使用 WebAssembly，否则还不如 JS，所以预测 2019 年也不会有太大变化。

至于框架之争这个敏感的问题，我只能说三者之间很难在短时间分出胜负了，你想问我看好哪个？唔……不如读者来说说你选择哪个，以及为什么？

我们评测了 5 个主流跨端框架，这是它们的区别

作者 凹凸实验室

最近前端届多端框架频出，相信很多有代码多端运行需求的开发者都会产生一些疑惑：这些框架都有什么优缺点？到底应该用哪个？

作为 Taro 开发团队一员，笔者想在本文尽量站在一个客观公正的角度去评价各个框架的选型和优劣。但宥于利益相关，本文的观点很可能是带有偏向性的，大家可以带着批判的眼光去看待，权当抛砖引玉。

那么，当我们在讨论多端框架时，我们在谈论什么？

多端

笔者以为，现在流行的多端框架可以大致分为三类：

1. 全包型

这类框架最大的特点就是从底层的渲染引擎、布局引擎，到中层的 DSL，再到上层的框架全部由自己开发，代表框架是 Qt 和 Flutter。这类框架优点非常明显：性能（的上限）高；各平台渲染结果一致。缺点也非常明显：需要完全重新学习 DSL（QML/Dart），以及难以适配中国特色的端：小程序。

这类框架是最原始也是最纯正的的多端开发框架，由于底层到上层每个环节都掌握在自己手里，也能最大可能地去保证开发和跨端体验一致。但它们的框架研发成本巨大，渲染引擎、布局引擎、DSL、上层框架每个部分都需要大量人力开发维护。

2. Web 技术型

这类框架把 Web 技术（JavaScript，CSS）带到移动开发中，自研布局引擎处理 CSS，使用 JavaScript 写业务逻辑，使用流行的前端框架作为 DSL，各端分别使用各自的原生组件渲染。代表框架是 React Native 和 Weex，这样做的优点有：

- 开发迅速；
- 复用前端生态；
- 易于学习上手，不管前端后端移动端，多多少少都会一点 JS、CSS。

缺点有：

1. 交互复杂时难以写出高性能的代码，这类框架的设计就必然导致 JS 和 Native 之间需要通信，类似于手势操作这样频繁地触发通信就很可能使得 UI 无法在 16ms 内及时绘制。React Native 有一些声明式的组件可以避免这个问题，但声明式的写法很难满足复杂交互的需求。
2. 由于没有渲染引擎，使用各端的原生组件渲染，相同代码渲染的一致性没有第一种高。

3. JavaScript 编译型

这类框架就是我们这篇文章的主角们：Taro、WePY 、uni-app 、 mpvue 、 chameleon，它们的原理也都大同小异：先以 JavaScript 作为基础选定一个 DSL 框架，以这个 DSL 框架为标准在各端分别编译为不同的代码，各端分别有一个运行时框架或兼容组件库保证代码正确运行。

这类框架最大优点和创造的最大原因就是小程序，因为第一第二种框架其实除了可以跨系统平台之外，也都能编译运行在浏览器中。(Qt 有 Qt for WebAssembly, Flutter 有 Hummingbird，React Native 有 react-native-web, Weex 原生支持)

另外一个优点是在移动端一般会编译到 React Native/Weex，所以它们也都拥有 Web 技术型框架的优点。这看起来很美好，但实际上 React Native/Weex 的缺点编译型框架也无法避免。除此之外，编译型框架的抽象也不是免费的：当 bug 出现时，问题的根源可能出在运行时、编译时、组件库以及三者依赖的库等等各个方面。在 Taro 开源的过程中，我们就遇到过 Babel 的 bug，React Native 的 bug，JavaScript 引擎的 bug，当然也少不了 Taro 本身的 bug。相信其它原理相同的框架也无法避免这一问题。

但这并不意味着这类为了小程序而设计的多端框架就都不堪大用。首先现在各巨头超级 App 的小程序百花齐放，框架会为了抹平小程序做了许多工作，这些工作在大部分情况下是不需要开发者关心的。其次是许多业务类型并不需要复杂的逻辑和交互，没那么容易触发到框架底层依赖的 bug。

那么当你的业务适合选择编译型框架时，在笔者看来首先要考虑的就是选择 DSL 的起点。因为有多端需求业务通常都希望能快速开发，一个能够快速适应团队开发节奏的 DSL 就至关重要。不管是 React 还是 Vue（或者类 Vue）都有它们的优缺点，大家可以根据团队技术栈和偏好自行选择。

如果不管什么 DSL 都能接受，那就可以进入下一个环节。

生态

以下内容均以各框架现在（2019 年 3 月 11 日）已发布稳定版为标准进行讨论。

开发工具

就开发工具而言 uni-app 应该是一骑绝尘，它的文档内容最为翔实丰富，还自带了 IDE 图形化开发工具，鼠标点点点就能编译测试发布。

其它的框架都是使用 CLI 命令行工具，但值得注意的是 chameleon 有独立的语法检查工具，Taro 则单独写了 ESLint 规则和规则集。

在语法支持方面，mpvue、uni-app、Taro 、WePY 均支持 TypeScript，四者也都能通过 typing 实现编辑器自动补全。除了 API 补全之外，得益于 TypeScript 对于 JSX 的良好支持，Taro 也能对组件进行自动补全。

CSS 方面，所有框架均支持 SASS、LESS、Stylus，Taro 则多一个 CSS Modules 的支持。

所以这一轮比拼的结果应该是：

uni-app > Taro > chameleon > WePY、mpvue

	chameleon	mpvue	Taro	uni-app	WePY
DSL	类 Vue	Vue	React	Vue	类 Vue
IDE/图形化开发工具	无	无	无	有	无
语法校验工具	自研	无	ESLint 规则	IDE 支持	无
TypeScript	无	有	有	有	有
Typing/自动补全	无	API	API + JSX	IDE 支持	无
样式	sass, less, stylus	sass, less, stylus	sass, less, stylus, CSS Modules	sass, less, stylus	sass, less, stylus

多端支持度

只从支持端的数量来看，Taro 和 uni-app 以六端略微领先（移动端、H5、微信小程序、百度小程序、支付宝小程序、头条小程序），chameleon 少了头条小程序紧随其后。

但值得一提的是 chameleon 有一套自研多态协议，编写多端代码的体验会好许多，可以说是一个能戳到多端开发痛点的功能。uni-app 则有一套独立的条件编译语法，这套语法能同时作用于 js、样式和模板文件。Taro 可以在业务逻辑中根据环境变量使用条件编译，也可以直接使用条件编译文件（类似 React Native 的方式）。

在移动端方面，uni-app 基于 weex 定制了一套 nvue 方案弥补 weex API 的不足；Taro 则是暂时基于 expo 达到同样的效果；chameleon 在移动端则有一套 SDK 配合多端协议与原生语言通信。

H5 方面，chameleon 同样是由多态协议实现支持，uni-app 和 Taro 则是都在 H5 实现了一套兼容的组件库和 API。

mpvue 和 WePY 都提供了转换各端小程序的功能，但都没有 h5 和移动端的支持。

所以最后一轮对比的结果是：

chameleon > Taro、uni-app > mpvue > WePY

	chameleon	mpvue	Taro	uni-app	WePY
移动端容器	Weex	无	React Native	Weex	无
移动端增强	chameleon SDK	无	Expo	nvue	无
多端编译方式	自研多态协议	环境变量条件编译	环境变量 + 文件条件编译	自研条件编译语法	环境变量条件编译
H5 兼容 API	自研多态协议	无	有	有	无
跨端组件库	有	无	有	有	无
小程序支持	微信、百度、支付宝	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝

组件库 / 工具库 /demo

作为开源时间最长的框架，WePY 不管从 Demo，组件库数量，工具库来看都占有一定优势。

uni-app 则有自己的插件市场和 UI 库，如果算上收费的框架和插件比起 WePy 也是完全不遑多让的。

Taro 也有官方维护的跨端 UI 库 taro-ui ，另外在状态管理工具上也有非常丰富的选择（Redux、MobX、dva），但 demo 的数量不如前两个。但 Taro 有一个转换微信小程序代码为 Taro 代码的工具，可以弥补这一问题。

而 mpvue 没有官方维护的 UI 库，chameleon 第三方的 demo 和工具库也还基本没有。

所以这轮的排序是：

WePY > uni-app 、taro > mpvue > chameleon

	chameleon	mpvue	Taro	uni-app	WePY
自研组件库	有	无	有	有	无
第三方组件	较少	丰富	较丰富	丰富	丰富
第三方工具库	较少	较丰富	较丰富	丰富	丰富
Demo	较少	较丰富	较丰富	较丰富	丰富
状态管理工具	Vuex	Vuex	Redux/MobX/Dva	Vuex	Redux
转换微信小程序工具	无	无	有	无	无

接入成本

接入成本有两个方面：

第一是框架接入原有微信小程序生态。由于目前微信小程序已呈一家独大之势，开源的组件和库（例如 wxparse、echart、zan-ui 等）多是基于原生微信小程序框架语法写成的。目前看来 uni-app 、Taro、mpvue 均有文档或 demo 在框架中直接使用原生小程序组件 / 库，WePY 由于运行机制的问题，很多情况需要小改一下目标库的源码，chameleon 则是提供了一个按步骤大改目标库源码的迁移方式。

第二是原有微信小程序项目部分接入框架重构。在这个方面 Taro 在京东购物小程序上进行了大胆的实践，具体可以查看文章《Taro 在京东购物小程序上的实践》。其它框架则没有提到相关内容。

而对于两种接入方式 Taro 都提供了 taro convert 功能，既可以将原有微信小程序项目转换为 Taro 多端代码，也可以将微信小程序生态的组件转换为 Taro 组件。

所以这轮的排序是：

Taro > mpvue 、 uni-app > WePY > chameleon

流行度

从 GitHub 的 star 来看，mpvue 、Taro、WePY 的差距非常小。从 NPM 和 CNPM 的 CLI 工具下载量来看，是 Taro（3k/week）> mpvue (2k/w) > WePY (1k/w)。但发布时间也刚好反过来。笔者估计三家的流行程度和案例都差不太多。

uni-app 则号称有上万案例，但不像其它框架一样有一些大厂应用案例。另外从开发者的数量来看也是 uni-app 领先，它拥有 20+ 个 QQ 交流群（最大人数 2000）。

所以从流行程度来看应该是：

uni-app > Taro、WePY、mpvue > chameleon

	chameleon	mpvue	Taro	uni-app	WePY
GitHub Star	3843	16281	16084	2921	16779
GitHub Issue/PR	68/8	1369/104	2026/368	215/18	1662/441
NPM + CNPM 下载量 [1]	241/周	1971/周	4332/周	N/A	976/周
案例	较少	丰富	丰富	非常丰富	丰富
开发者人数 [2]	~1000 人	~ 5000 人	~ 5000 人	10000+	~ 5000 人
自建开发者社区	无	无	无	有	无

开源建设

一个开源作品能走多远是由框架维护团队和第三方开发者共同决定的。虽然开源建设不能具体地量化，但依然是衡量一个框架 / 库生命力的非常重要的标准。

从第三方贡献者数量来看，Taro 在这一方面领先，并且 Taro 的一些核心包 / 功能（MobX、CSS

Modules、alias）也是由第三方开发者贡献的。除此之外，腾讯开源的 omi 框架小程序部分也是基于 Taro 完成的。

WePY 在腾讯开源计划的加持下在这一方面也有不错的表现；mpvue 由于停滞开发了很久就比较落后了；可能是产品策略的原因，uni-app 在开源建设上并不热心，甚至有些部分代码都没有开源；chameleon 刚刚开源不久，但它的代码和测试用例都非常规范，以后或许会有不错的表现。

那么这一轮的对比结果是：

Taro > WePY > mpvue > chameleon > uni-app

最后补一个总的生态对比图表：

		chameleon	mpvue	Taro	uni-app	WePY
开发工具	DSL	类 Vue	Vue	React	Vue	类 Vue
	IDE/图形化开发工具	无	无	无	有	无
	语法校验工具	自研	无	ESLint 规则	IDE 支持	无
	TypeScript	无	有	有	有	有
	Typing/自动补全	无	API	API + JSX	IDE 支持	无
	样式	sass, less, stylus	sass, less, stylus	sass, less, stylus, CSS Modules	sass, less, stylus	sass, less, stylus
多端支持	小程序支持	微信、百度、支付宝	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝、字节跳动	微信、百度、支付宝
	移动端容器	Weex	无	React Native	Weex	无
	移动端增强	chameleon SDK	无	Expo	nvue	无
	多端编译方式	自研多态协议	环境变量条件编译	环境变量 + 文件条件编译	自研条件编译语法	环境变量条件编译
	H5 兼容 API	自研多态协议	无	有	有	无
	跨端组件库	有	无	有	有	无
组件库/工具库/demo	第三方组件	较少	丰富	较丰富	丰富	丰富
	第三方工具库	较少	较丰富	较丰富	丰富	丰富
	Demo	较少	较丰富	较丰富	较丰富	丰富
	状态管理工具	Vuex	Vuex	Redux/MobX/Dva	Vuex	Redux
	转换微信小程序工具	无	无	有	无	有
	自研组件库	有	无	有	有	无
流行度	GitHub Star	3843	16281	16084	2921	16779
	GitHub Issue/PR	68/8	1369/104	2026/368	215/18	1662/441
	NPM + CNPM 下载量[1]	241/周	1971/周	4332/周	N/A	976/周
	案例	较少	丰富	丰富	非常丰富	丰富
	开发者人数[2]	~1000 人	~ 5000 人	~ 5000 人	10000+	~ 5000 人
	自建开发者社区	无	无	无	有	无

[*] 数据采集于 2019 年 3 月 11 日，以各框架目前稳定版特性为标准

[1] 统计上周各框架 CLI 工具下载量，uni-app 可用 IDE 工具直接开发

[2] 统计框架及相关生态微信/QQ 群的总人数

未来

从各框架已经公布的规划来看：

WePY 已经发布了 v2.0.alpha 版本，虽然没有公开的文档可以查阅到 2.0 版本有什么新功能 / 特性，但据其作者介绍，WePY 2.0 会放大招，是一个「对得起开发者」的版本。笔者也非常期待 2.0 正式发布后 WePY 的表现。

mpvue 已经发布了 2.0 的版本，主要是更新了其它端小程序的支持。但从代码提交， issue 的回复 / 解决率来看，mpvue 要想在未来有作为首先要打消社区对于 mpvue 不管不顾不更新的质疑。

uni-app 已经在生态上建设得很好了，应该会在此基础之上继续稳步发展。如果 uni-app 能加强开源开放，再加强与大厂的合作，相信未来还能更上一层楼。

chameleon 的规划比较宏大，虽然是最后发的框架，但已经在规划或正在实现的功能有：

- 快应用和端拓展协议；

- 通用组件库和垂直类组件库；
- 面向研发的图形化开发工具；
- 面向非研发的图形化页面搭建工具。

如果 chameleon 把这些功能都做出来的话，再继续完善生态，争取更多第三方开发者，那么在未来 chameleon 将大有可为。

Taro 的未来也一样值得憧憬。Taro 即将要发布的 1.3 版本就会支持以下功能：

- 快应用支持；
- Taro Doctor，自动化检查项目配置和代码合法性；
- 更多的 JSX 语法支持，1.3 之后限制生产力的语法只有只能用 map 创造循环组件一条；
- H5 打包体积大幅精简。

同时 Taro 也正在对移动端进行大规模重构；开发图形化开发工具；开发组件 / 物料平台以及图形化页面搭建工具。

结语

那说了那么多，到底用哪个呢?

如果不介意尝鲜和学习 DSL 的话，完全可以尝试 WePY 2.0 和 chameleon，一个是酝酿了很久的 2.0 全新升级，一个有专门针对多端开发的多态协议。

uni-app 和 Taro 相比起来就更像是「水桶型」框架，从工具、UI 库，开发体验、多端支持等各方面来看都没有明显的短板。而 mpvue 由于开发一度停滞，现在看来各个方面都不如在小程序端基于它的 uni-app 。

当然，Talk is cheap。如果对这个话题有更多兴趣的同学可以去 GitHub 另行研究，有空看代码，没空看提交。

张鑫旭：前端专业方向的尽头

作者 张鑫旭

一、纯专业方向的探索之路

一转眼，毕业已经快 10 年了，10 年前我在写页面，10 年过去了，我还在写页面。

这种情形目前并不多见，无论是我的前辈或者是同一年代入行的同辈，几乎都已经脱离一线了，至少我认识的那些都是如此。

每个人都是独立的个体，没有什么按部就班，没有什么理所当然，关键要清楚自己要的什么，自己拥有的是什么，自己能够到达的彼岸有多远。

如果三五年前，我觉得自己的专业对团队对企业的贡献到头了，我也会考虑向人向事这一块转型，只是，当技术积累突破到一定阶段后，你心里面就会明白，自己能够做的事情还有很多，自己可以走出一条别人没有走过的纯专业方向的探索之路。

这是一段特殊的旅程，没有借鉴，没有参考，可能旅程并不顺利，也可能会有别样的风采，但人生在世，本就当如此，遵循自己内心的指引，创造属于自己的价值，看淡那些外在的依附。

价值思考

所以，长久一来，我一直在思考这么一个问题：如何围绕前端专业技术，给团队带来最大的价值？

我总结了下面几点，也是这些年自己一直努力的方向：

1. 项目技术攻坚

在项目中体现自己的专业价值，就偏体验的前端而言包括：

- 能够实现任何设计师提出的动效；
- 各类图形与图像处理技术；
- 小众领域的技术研究与实践，如无障碍访问；
- 新技术新特性的实践与落地；
- 复杂产品复杂系统的架构与设计；
- 以及对细节的把握和产品的品质，这往往与技术积累有较大关联。

2. 基础技术建设

个体的技术再强，也只是强的你一个人，如果能让周围同事也很强，那对于团队的价值就很大了。其中有个非常有效的方法就是把你学到的那些专业知识融入到基础建设中，包括底层框架，或者 UI 组件库，或者标准结局方案等。

举个例子，对于前端而言，无障碍访问本应是必备知识，尤其对于百万、千万级 DAU 的产品而言，实际上，了解并在产品中使用的人寥寥。这很好理解，法律又没规定产品一定要做无障碍，做了无障碍也只是服务小部分人，又不会升职加薪，想让员工靠爱发电，真的很难。但是，如果你把这些学习与研究直接融入到底层基础建设中，其它同事无需学习就能使用，就等同于你也让其他同事变得很强。对于团队而言，是很有价值的一件事情。

然而，如果只是为了功利目的重复造轮子，那真不如 fork 一个优秀的开源项目进行本地化，对大家都好。

3. 工具与生产力释放

一旦工作中出现了重复劳动的场景，就可以考虑能够用技术手段解放生产力，做一个可视化的桌面或者 Web 工具都可以。

作为前端，做工具有个天然优势，就是界面可视化的能力一等一，尤其制作给设计师、产品经理、运营编辑使用的工具尤其受欢迎，这是后端同学无法驾驭的。

现代前端技术发展迅猛，各种新特性强悍无比，最终实现的 Web 产品几乎可以媲美桌面端软件。跨平台，自己人用不要考虑兼容性，各种新特性都可以拿来尝试，又能产生巨大价值，这么爽的事情一定要来者不拒，一定要主动发现需求。

4. 知识分享与人才培养

还是那句话，你一个人再强，没什么卵用的，尤其前端这种偏展现的职位，所谓独木难支。你要想办法让周围同事也变得优秀，所以，多多做专业知识分享，别人成长了，团队也就成长了，这就是价值贡献。

有些人千万不要犯傻，以为知识分享出去让被人技术提高了，自己会被踩，就藏着掖着。脑子一定要清爽，职场中职位高低是与你对团队对公司贡献正向强挂钩的，而与技术高低是弱挂钩。

积极争取带新人培养新人的机会，要真心想要带好新人，帮助他们成长，而不是应付绩效，或者领导让我带我就勉强应付，做人要有良心，新人起步还是挺重要的，不要草草应付，你自己这辈子凉了没事，可不要连累别人。

对于我个人，还有另外一个特殊的价值，就是“吉祥物”。

二、局限与尽头

这些年围绕着前端专业，本着不断创造价值的理念，指引自己的行动，确实也做出了一点微不足道的成绩。有项目产出，基础建设也在很多产品中应用，做了不少工具，有些工具释放了千人次的人力，团队内知识分享次数远远领先，对所有应届生人、实习生每周持续技术培训，同时还有主动参与大量人才招聘的工作。

然而，人总是要不断成长的，在一线工作快 10 年的这个节点，我发现纯专业技术这条路能够给团队进一步提升价值的空间越来越小。

首先是项目这块，人总是高估自己这个职业对产品的价值，前端开发人员也不例外，实际上，一个产品要想成功需要通力协作，没有明显短板要有强项，但我几乎没看到什么产品强项是靠前端突显的。站在企业的角度，前端 80 分和 90 分带来的价值区别并不大，或者说并不紧急，举个极端的例子，你网站 CSS 质量全世界 Number 1，然后呢？没有然后，你的产品不会因为这个风生水起，收益大增。要知道，产品不是艺术品。虽然我自己专业的成长很明确，还要继续耕耘与积累，但是，如何给项目产品带来明显的收益提升却难倒我了，我暂时想到的是 webGL，填补团队这块的空白，至于其它，还没想清楚。

基础建设这块，首先业务线很多，个体的精力有限，不同业务线适合的技术形态也不一样，无法完全兼顾。而且各类技术框架风起云涌，像我这样的老古董确实应接不暇，开始过时了。

工具的问题在于生产力的需求总是有限的，需求解决之后，一旦稳定下来，这方面可以做的事情就越来越少了。

知识分享的问题在于知识的吸收、汇总和落地，如何让没有听过的新人也能 GET 到这些知识，这方面的价值远比分享数量的堆砌和形式主义要高得多。

所以，就有难题摆在我的面前：如何通过前端专业技术给团队进一步带来明显提升的价值呢？

我发现了我的前端专业之路遇到了局限，前端专业方向看到了尽头。

略带悲伤！但，放心，我并没有为此担忧，这是一个必然要遇到的问题。想要通过纯技术，尤其是前端技术想到达到一个很高的级别，那是不可能的，不可能说你前端很厉害，然后给公司增加几千万的收益。哪怕你公司是培训机构，哪怕赶上前端培训巅峰的时候，也不会如此。

相比我的前辈们或者同辈们，我已经比他们多探索了四五年时间，走得更极端也更扎实，准备地更充分，也没有任何遗憾。至少今天在这里留下了印记，证明过，一个技术人员，就算只靠专业技术创造价值，也能有一番属于自己的精彩。

所以，一切都是自然而言，纯专业趋于顶部，那我就开始在另外一条线上探索，从另外一个完全空白的维度进行成长与提升，相当于启动二级火箭，强劲辅助专业那条线进一步成长。这里的"另外一条线"指的就是人和事。

三、绕不开的人和事

简言之就是通过推动别人来推动团队的专业成长，通过事务落地来推动团队的专业成长。

举例来说，大家一起做了那么多大大小小的项目，那些好的专业实践有没有汇总与落地，有没有转化为很多的经验和学习资源。这个事情的价值就很大，但是，这些事情的推动那就不是说你专业技术厉害就搞得定的，要与人到交道，如何制定策略，何种方式汇总，什么形式转化，如何后续推广等等都是需要思考的问题；如果我们需要一个专业的站点进行归档，在什么地方合适，自己开发还是使用已有平台，需要动用哪些资源，实践过程中会遇到哪些困难也是另外一堆需要思考的问题。

虽然这些问题需要的并不是专业能力，但是，对于整个团队带来的专业这块的价值确是非常显著的，这就是通过另外一条线，也就是人和事让自己职业更进一步。

又比如说知识分享，讲的时候听得很 high，结果一段时间过后，当时学了啥的，记不得了。还有个比较严重的问题，那就是新人加入后，以前分享的那些精彩内容如何追溯，或者至少知道以前哪方面课题有同事有过很棒的分享，可以直接学习。所以，即使平均每周一次分享，如果分享之后没有对知识进一步规定，梳理，以更好的形式呈现，那你做那么多分享带来的价值，或者说投入产出比不见得有多高，这是一个非常能够明显提升整个团队专业价值的方向。但同样的，这个方向的提升不是靠你的专业能力，或者你分享数量，而是更好的流程约束，更好的工具辅助，更好的文化熏陶。

而所有这些事情的成功推进落地，离不开与人打交道，如本部门同事、领导，以及其他部门同事，甚至还要老板那边提供资源。

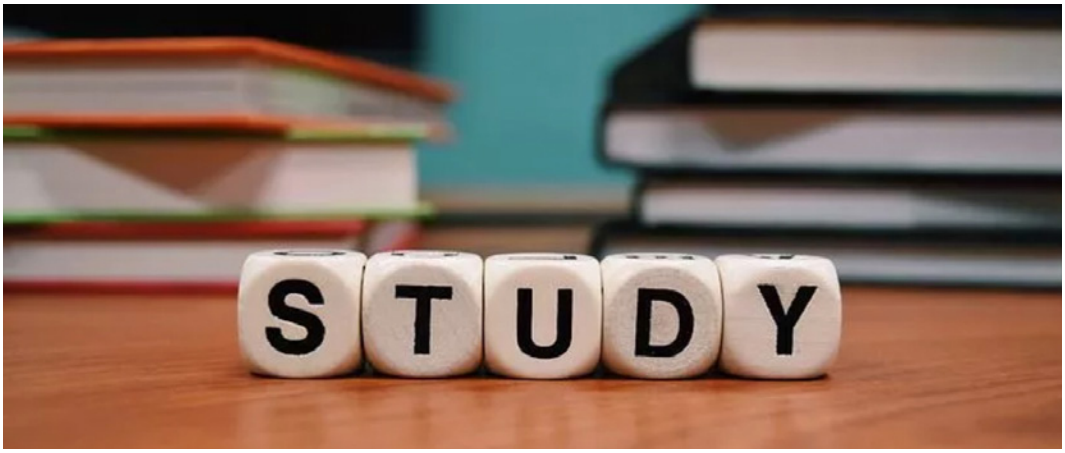
所以，到了一定阶段后，要想进一步提高团队的专业价值，人和事是绕不开的，而这一块，将会是自己接下来几年努力提高、学习以及成长的地方。估计会没有学技术那么顺利，与人打交道不像和代码打交道，不能直来直往，否则容易让别人不开心，我也意识到这个问题了，不过我相信自己的可塑性，罗马不是一天建成的，慢慢改善吧，只是担心在职场磨啊磨，磨啊磨，把棱角都磨掉了，到底是好事还是坏事呢?

四、远处的风景

19 年算是个转折年吧，想想自己，一条腿走了 10 年还真不容易，是时候把另外一条腿慢慢长起来了。心有猛虎，细嗅蔷薇，不为外物蒙双眼，不因碎语扰心智，一步一步，有条不紊，耐心前进，人生是条长河，希望能有双健康有力的双腿，带我看到尽头最美的风景。

首屏时间从 12.67s 到 1.06s，我是如何做到的？

作者 jerryOnlyZRJ



本文是对之前同名文章的修正，将所有 webpack3 的内容更新为 webpack4，以及加入了笔者近期在公司工作中学习到的自动化思想，对文章内容作了进一步提升。

引言

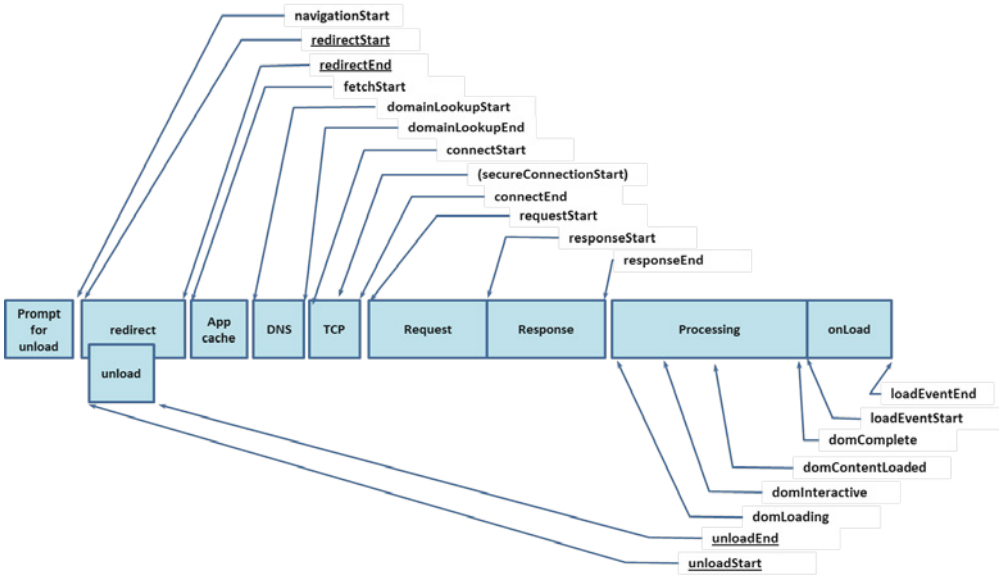
对于网站的性能，在行业内有很多既定的指标，但就以前端 er 而言，我们应该更加关注以下指标: 白屏时间、首屏时间、整页时间、DNS 时间、CPU 占用率。而我之前自己搭建的一个网站完全没做性能优化时，首屏时间是 12.67s，最后经过多方面优化，终于将其降低至 1.06s，并且还未配置 CDN 加速。其中过程我踩了很多坑，也翻了许多专业书籍，最后决定将这几日的努力整理成文，帮助前端爱好者们少走弯路。

文章更新可能之后不会实时同步在论坛上，欢迎大家关注我的 Github，我会把最新的文章更新在对应的项目里，让我们一起在代码的海洋里策马奔腾。

今天，我们将从性能优化的三大方面工作逐步展开介绍，其中包括网络传输性能、页面渲染性能以及 JS 阻塞性能，系统性地带着读者们体验性能优化的实践流程。

1. 网络传输性能优化

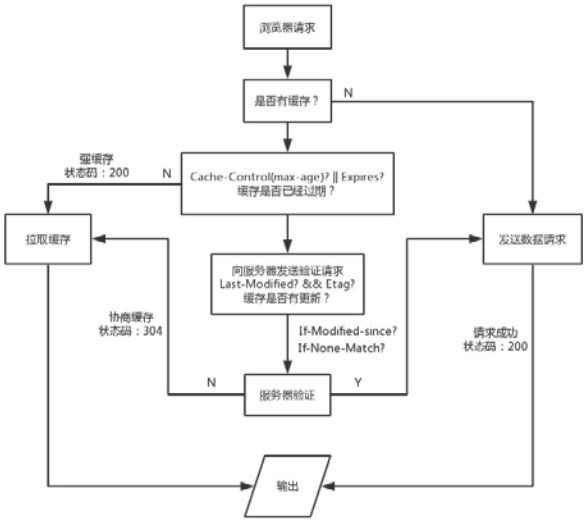
在开始介绍网络传输性能优化这项工作之前，我们需要了解浏览器处理用户请求的过程，那么就必须奉上这幅神图了：



这是 navigation timing 监测指标图，从图中我们可以看出，浏览器在得到用户请求之后，经历了下面这些阶段：重定向→拉取缓存→ DNS 查询→建立 TCP 链接→发起请求→接收响应→处理 HTML 元素→元素加载完成。不着急，我们对其中的细节一步步展开讨论。

1.1 浏览器缓存

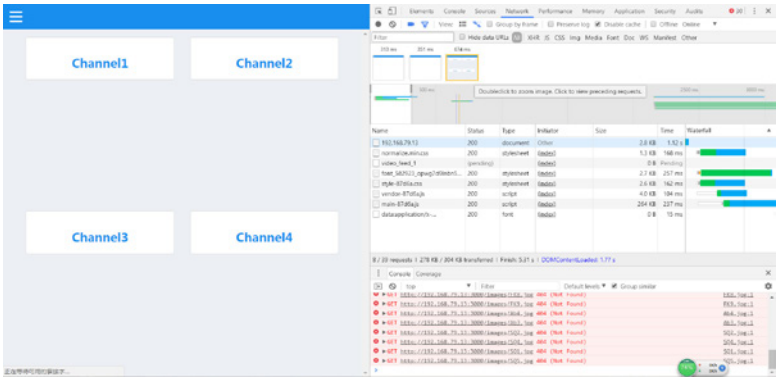
我们都知道，浏览器在向服务器发起请求前，会先查询本地是否有相同的文件，如果有，就会直接拉取本地缓存，这和我们在后台部属的 Redis 和 Memcache 类似，都是起到了中间缓冲的作用，我们先看看浏览器处理缓存的策略。



因为网上的图片太笼统了，而且我翻过很多讲缓存的文章，很少有将状态码还有什么时候将缓存存放在内存（memory）中什么时候缓存在硬盘中（disk）系统地整理出来，所以我自己绘制了一张浏览器缓存机制流程图，结合这张图再更深入地说明浏览器的缓存机制。

这里我们可以使用 chrome devtools 里

的 network 面板查看网络传输的相关信息。（这里需要特别注意，在我们进行缓存调试时，需要去除 network 面板顶部的 Disable cache 勾选项，否则浏览器将始终不会从缓存中拉取数据。）



浏览器默认的缓存是放在内存内的，但我们知道，内存里的缓存会因为进程的结束或者说浏览器的关闭而被清除，而存在硬盘里的缓存才能够被长期保留下去。很多时候，我们在 network 面板中各请求的 size 项里，会看到两种不同的状态：from memory cache 和 from disk cache，前者指缓存来自内存，后者指缓存来自硬盘。而控制缓存存放位置的，不是别人，就是我们在服务器上设置的 Etag 字段。在浏览器接收到服务器响应后，会检测响应头部（Header），如果有 Etag 字段，那么浏览器就会将本次缓存写入硬盘中。

之所以拉取缓存会出现 200、304 两种不同的状态码，取决于浏览器是否有向服务器发起验证请求。只有向服务器发起验证请求并确认缓存未被更新，才会返回 304 状态码。

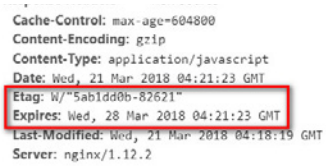
这里我以 nginx 为例，谈谈如何配置缓存：

首先，我们先进入 nginx 的配置文档：

```
$ vim nginxPath/conf/nginx.conf
```

在配置文档内插入如下两项：

```
etag on;    // 开启 etag 验证
expires 7d;     // 设置缓存过期时间为 7 天
```



打开我们的网站，在 chrome devtools 的 network 面板中观察我们的请求资源，如果在响应头部看见 Etag 和 Expires 字段，就说明我们的缓存配置成功了。

特别注意，在我们配置缓存时一定要切记，浏览器在处理用户请求时，如果命中强缓存，浏览器会直接拉取本地缓存，不会与服务器发生任何通信，也就是说，如果我们在服务器端更新了文件，并不会被浏览器得知，就无法替换失效的缓存。所以我们在构建阶段，需要为我们的静态资源添加 md5 hash 后缀，避免资源更新而引起的前后端文件无法同步的问题。

1.2 资源打包压缩

我们之前所作的浏览器缓存工作，只有在用户第二次访问我们的页面才能起到效果，如果要在用户首次打开页面就实现优良的性能，必须对资源进行优化。我们常将网络性能优化措施归结为三大方面：减少请求数、减小请求资源体积、提升网络传输速率。现在，让我们逐个击破：

结合前端工程化思想，我们在对上线文件进行自动化打包编译时，通常都需要打包工具的协助，这里我推荐 webpack，我通常都使用 Gulp 和 Grunt 来编译 node，Parcel 太新，而且 webpack 也一直在自身的特性上向 Parcel 靠拢。

在对 webpack 进行上线配置时，我们要特别注意以下几点：

1.JS 压缩（这点应该算是耳熟能详了，就不多介绍了）

```
optimization: {
    minimizer: [
        new UglifyJsPlugin({
            cache: true,
            parallel: true,
            sourceMap: true // set to true if you want JS source maps
        }),
        ...Plugins
    ]
}
```

2.HTML 压缩

```
new HtmlWebpackPlugin({
        template: __dirname + '/views/index.html',
        // new 一个这个插件的实例，并传入相关的参数
        filename: '../index.html',
        minify: {
            removeComments: true,
            collapseWhitespace: true,
            removeRedundantAttributes: true,
            useShortDoctype: true,
            removeEmptyAttributes: true,
            removeStyleLinkTypeAttributes: true,
            keepClosingSlash: true,
            minifyJS: true,
            minifyCSS: true,
            minifyURLs: true,
        },
        chunksSortMode: 'dependency'
    })
```

我们在使用 html-webpack-plugin 自动化注入 JS、CSS 打包 HTML 文件时，很少会为其添加配置项，这里我给出样例，大家直接复制就行。据悉，在 Webpack5 中，html-webpack-plugin 的功能会像 common-chunk-plugin 那样，被集成到 webpack 内部，这样我们就不需要再 install 额外的插件了。

PS：这里有一个技巧，在我们书写 HTML 元素的 src 或 href 属性时，可以省略协议部分，这样也能简单起到节省资源的目的。

3. 提取公共资源

```
splitChunks: {
```

```
  cacheGroups: {
    vendor: { // 抽离第三方插件
      test: /node_modules/, // 指定是 node_modules 下的第三方包
      chunks: 'initial',
      name: 'common/vendor', // 打包后的文件名，任意命名
      priority: 10 // 设置优先级，防止和自定义的公共代码提取时被覆盖，不进行打包
    },
    utils: { // 抽离自定义公共代码
      test: /\.js$/,
      chunks: 'initial',
      name: 'common/utils',
      minSize: 0 // 只要超出 0 字节就生成一个新包
    }
  }
}
```

4. 提取 css 并压缩

在使用 webpack 的过程中，我们通常会以模块的形式引入 css 文件（webpack 的思想不就是万物皆模块嘛），但是在上线的时候，我们还需要将这些 css 提取出来，并且压缩，这些看似复杂的过程只需要简单的几行配置就行。

PS: 我们需要用到 mini-css-extract-plugin ，所以还得大家自行 npm install。

```
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
module: {
    rules: [..., {
        test: /\.css$/,
        exclude: /node_modules/,
        use: [
            _mode === 'development' ? 'style-loader' : MiniCssExtractPlugin.loader, {
                loader: 'css-loader',
                options: {
                    importLoaders: 1
                }
            }, {
                loader: 'postcss-loader',
                options: {
                    ident: 'postcss'
                }
            }
        ]
    }]
}
```

我这里配置预处理器 postcss，但是我把相关配置提取到了单独的文件 postcss.config.js 里了，其

中 cssnano 是一款很不错的 CSS 优化插件。

5. 将 webpack 开发环境修改为生产环境

在使用 webpack 打包项目时，它常常会引入一些调试代码，以作相关调试，我们在上线时不需要这部分内容，通过配置剔除：

```
devtool: 'false'
```

如果你能按照上述六点将 webpack 上线配置完整配置出来，基本能将文件资源体积压缩到极致了，如有疏漏，还希望大家能加以补充。

最后，我们还应该在服务器上开启 Gzip 传输压缩，它能将我们的文本类文件体积压缩至原先的四分之一，效果立竿见影，还是切换到我们的 nginx 配置文档，添加如下两项配置项目：

```
gzip on;
gzip_types text/plain application/javascriptapplication/x-javascripttext/css application/xml text/javascriptapplication/x-httpd-php application/vnd.ms-fontobject font/ttf font/opentype font/x-woff image/svg+xml;
```

特别注意，不要对图片文件进行 Gzip 压缩，我只会告诉你效果适得其反，至于具体原因，还得考虑服务器压缩过程中的 CPU 占用还有压缩率等指标，对图片进行压缩不但会占用后台大量资源，压缩效果其实并不可观，可以说是“弊大于利”，所以请在 gzip_types 把图片的相关项去掉。针对图片的相关处理，我们接下来会更加具体地介绍。

1.3 图片资源优化

刚刚我们介绍了资源打包压缩，只是停留在了代码层面，而在我们实际开发中，真正占用了大量网络传输资源的，并不是这些文件，而是图片，如果你对图片进行了优化工作，你能立刻看见明显的效果。

1.3.1 不要在 HTML 里缩放图像

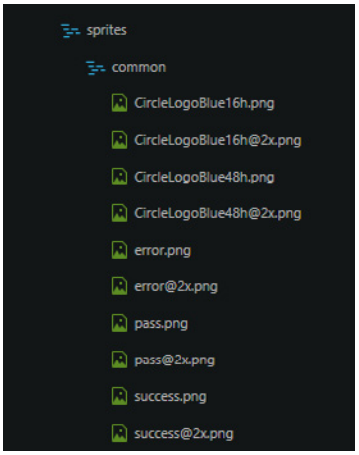
很多开发者可能会有这样的错觉（其实我曾经也是这样），我们会为了方便在一个 200*200 的图片容器内直接使用一张 400*400 的图片，我们甚至认为这样能让用户觉得图片更加清晰，其实不然，在普通的显示器上，用户并不会感到缩放后的大图更加清晰，但这一切却导致网页加速速度下降，同时照成带宽浪费，你可能不知道，一张 200KB 的图片和 2M 的图片的传输时间会是 200m 和 12s 的差距（亲身经历，深受其害）。所以，当你需要用多大的图片时，就在服务器上准备好多大的图片，尽量固定图片尺寸。

1.3.2 使用雪碧图（CSS Sprite）

雪碧图的概念大家一定在生活中经常听见，其实雪碧图是减小请求数的显著运用。而且很奇妙的是，多张图片聘在一块后，总体积会比之前所有图片的体积之和小（你可以亲自试试）。这里给大家推荐一个自动化生成雪碧图的工具：https://www.toptal.com/developers/css/sprite-generator。只要你添加相关资源文件，他就会自动帮你生成雪碧图以及对应的 CSS 样式。

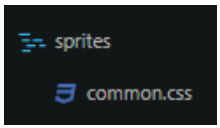
其实我们在工程中还有更为自动的方法，便是一款雪碧图生成插件 webpack-spritesmith。首先，先简单介绍一下使用插件生成雪碧图的思路。

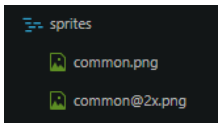
首先，我们会把我们所需要的小图标放置在一个文件夹内以便于管理。



这里的 @2x 图片是为了适配视网膜二倍屏的图片资源，webpack-spritesmith 内有专门为适配多倍屏提供的配置项，稍候将会讲到。

然后，我们需要插件去读取这个文件夹内的所有图片资源文件，以文件夹名称为图片名称生成一张雪碧图到指定位置，并且输出能够正确使用这些雪碧图的 CSS 文件。





如今，webpack-spritesmith 这款插件能实现我们想要的一切，先奉上配置内容：

```
const SpritesmithPlugin = require('webpack-spritesmith')
//获取你放雪碧图的所有文件夹
const spritesDirs = glob.sync('./src/client/assets/sprites/*')
const spritesPlugins = spritesDirs.map(spritesDir => {
  //获取每个文件夹名称
  let dirName = basename(spritesDir)
  return new SpritesmithPlugin({
    src: {
      // 图片所在文件夹（光标于文件夹）
      cwd: resolve(spritesDir),
      // 匹配 png 文件，可以用glob语法，比如 '*.{png|jpg}' 这样
      // PS：png和jpg拼一起，有时候图片无法正常显示
      glob: '*.png'
    },
    // 输出雪碧图文件及样式文件
    target: {
      // 将其输出到 src/assets 目录下
      // 这个是打包前的目录，所以不要学某个教程将其输出到 dist 目录下
      image: resolve(`src/client/assets/images/sprites/${dirName}.png`),
      // 可以是字符串，这省数据
      css: resolve(`src/client/assets/styles/sprites/${dirName}.css`)
    },
    apiOptions: {
      generateSpriteName: function() {
        const fileName = arguments[0].match(/[^\\]+$/)[0].replace(/\.[a-zA-Z]+/, '')
        // 雪碧图每个元素生成的类名，.icon-dirname-filename
        // console.log(`.icon-${dirName}-${fileName}`)
        return `${dirName}-${fileName}`
      },
      // 简单来说，这个就是雪碧图的 css 文件怎么找到 雪碧图的 png 文件
      cssImageRef: `../../images/sprites/${dirName}.png`
    },
    spritesmithOptions: {
      // 这个是雪碧图的排列顺序（从上到下）
      algorithm: 'left-right', // or top-down
      // 雪碧间距，图片和图片的距离，单位是px
      padding: 20
    },
    //自动适配视网膜二倍屏
    retina: '@2x'
  })
})
```

具体可参照 webpack-spritesmith 官方文档。

执行 webpack 之后，就会在开发目录里生成上面两张图的结果，我们可以看看 common.css 里面的内容。

```
.icon-common-CircleLogoBlue16h {
  background-image: url(../../images/sprites/common.png);
  background-position: -70px 0px;
  width: 16px;
  height: 16px;
}
.icon-common-CircleLogoBlue48h {
  background-image: url(../../images/sprites/common.png);
  background-position: -106px 0px;
  width: 48px;
  height: 48px;
}
.icon-common-error {
  background-image: url(../../images/sprites/common.png);
  background-position: 0px 0px;
  width: 15px;
  height: 15px;
}
.icon-common-pass {
  background-image: url(../../images/sprites/common.png);
  background-position: -35px 0px;
  width: 15px;
  height: 15px;
}
.icon-common-success {
  background-image: url(../../images/sprites/common.png);
  background-position: -174px 0px;
  width: 235px;
  height: 103px;
}
```

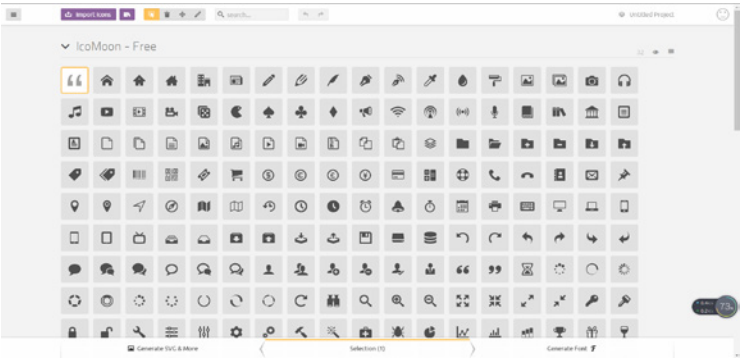
我们可以看到，所有我们之前放在 common 文件夹里的图片资源都自动地生成了相应的样式，这些都不需要我们手动处理，webpack-spritesmith 这款插件就已经帮我们完成了！

1.3.3 使用字体图标（iconfont）

无论是压缩后的图片，还是雪碧图，终归还是图片，只要是图片，就还是会占用大量网络传输资源。但是字体图标的出现，却让前端开发者看到了另外一个神奇的世界。

我最喜欢用的是阿里矢量图标库，里面有大量的矢量图资源，而且你只需要像在淘宝采购一样把他们添加至购物车就能把它们带回家，整理完资源后还能自动生成 CDN 链接，可以说是完美的一条龙服务了。

图片能做的很多事情，矢量图都能作，而且它只是往 HTML 里插入字符和 CSS 样式而已，和图片请求比起来资源占用完全不在一个数量级，如果你的项目里有小图标，就是用矢量图吧。



但如果我们做的是公司或者团队的项目，需要使用到许多自定义的字体图标，可爱的设计小姐姐们只是丢给你了几份 .svg 图片，你又该如何去做呢?

其实也很简单，阿里矢量图标库就提供了上传本地 SVG 资源的功能，这里另外推荐一个网站——icomoon。icomoon 这个网站也为我们提供了将 SVG 图片自动转化成 CSS 样式的功能。

我们可以点击 Import Icons 按钮导入我们本地的 SVG 资源，然后选中他们，接下来生成 CSS 的事情，就交给 icomoon 吧，具体的操作，就和阿里矢量图标库类同了。

1.3.4 使用 WebP

WebP 格式，是谷歌公司开发的一种旨在加快图片加载速度的图片格式。图片压缩体积大约只有 JPEG 的 2/3，并能节省大量的服务器带宽资源和数据空间。Facebook、Ebay 等知名网站已经开始测试并使用 WebP 格式。

我们可以使用官网提供的 Linux 命令行工具对项目中的图片进行 WebP 编码，也可以使用我们的线上服务，这里我推荐叉拍云。但是在实际的上线工作中，我们还是得编写 Shell 脚本用命令行工具进行自动化编译，测试阶段用线上服务方便快捷。（图片来自叉拍云官网）

1.4 网络传输性能检测工具——Page Speed

除了 network 版块，其实 chrome 还为我们准备好了一款监测网络传输性能的插件——Page

Speed，咱们的文章封面，就是用的 Page Speed 的官方宣传图（因为我觉得这张图再合适不过了）。我们只需要通过下面步骤安装，就可以在 chrome devtools 里找到它了：chrome 菜单→更多工具→拓展程序→ chrome 网上应用商店→搜索 pagespeed 后安转即可。

我们只需要打开待测试的网页，然后点击 Page Speed 里的 Start analyzing 按钮，它就会自动帮我们测试网络传输性能了，这是我的网站测试结果：

Page Speed 最人性化的地方，便是它会对测试网站的性能瓶颈提出完整的建议，我们可以根据它的提示进行优化工作。这里我的网站已经优化到最好指标了，Page Speed Score 表示你的性能测试得分，100/100 表示已经没有需要优化的地方。

优化完毕后再使用 chorme devtools 的 network 版块测量一下我们网页的白屏时间还有首屏时间，是不是得到了很大的提升？

1.5 使用 CDN

Last but not least，再好的性能优化实例，也必须在 CDN 的支撑下才能到达极致。

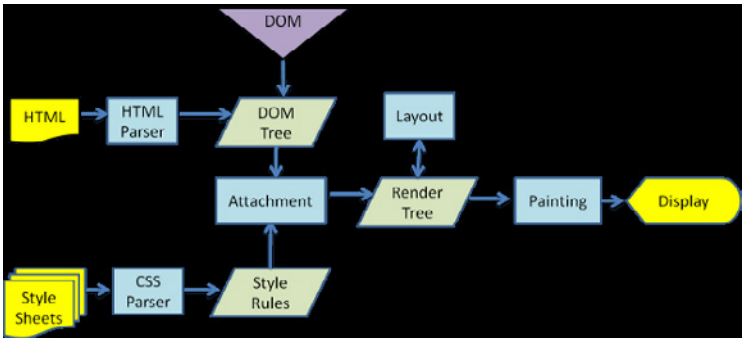
如果我们在 Linux 下使用命令 $ traceroute targetIp 或者在 Windows 下使用批处理 > tracert targetIp，都可以定位用户与目标计算机之间经过的所有路由器，不言而喻，用户和服务器之间距离越远，经过的路由器越多，延迟也就越高。使用 CDN 的目的之一便是解决这一问题，当然不仅仅如此，CDN 还可以分担 IDC 压力。

当然，凭着我们单个人的资金实力（除非你是王思聪）是必定搭建不起来 CDN 的，不过我们可以使用各大企业提供的服务，诸如腾讯云等，配置也十分简单，这里就请大家自行去推敲啦。

2. 页面渲染性能优化

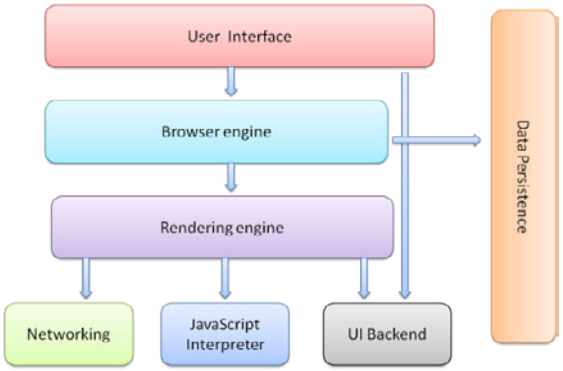
2.1 浏览器渲染过程（Webkit）

其实大家应该对浏览器将的 HTML 渲染机制比较熟悉了，基本流程同上图所述，大家在入门的时候，你的导师或者前辈可能会告诉你，在渲染方面我们要减少重排和重绘，因为他们会影响浏览器



性能。不过你一定不知道其中原理是什么，对吧。今天我们就结合《Webkit 技术内幕》（这本书我还是很推荐大家买来看看，好歹作为一名前端工程师，你得知道我们天天接触的浏览器内核是怎样工作的）的相关知识，给大家普及普及那些深层次的概念。

PS：这里提到了 Webkit 内核，我顺带提一下浏览器内部的渲染引擎、解释器等组件的关系，因为经常有师弟或者一些前端爱好者向我问这方面的知识，分不清他们的关系，我就拿一张图来说明。（如果你对着不感兴趣，可以直接跳过）



浏览器的解释器，是包括在渲染引擎内的，我们常说的 Chrome（现在使用的是 Blink 引擎）和 Safari 使用的 Webkit 引擎，Firefox 使用的 Gecko 引擎，指的就是渲染引擎。而在渲染引擎内，还包括着我们的 HTML 解释器（渲染时用于构造 DOM 树）、CSS 解释器（渲染时用于合成 CSS 规则）还有我们的 JS 解释器。不过后来，由于 JS 的使用越来越重要，工作越来越繁杂，所以 JS 解释器也渐渐独立出来，成为了单独的 JS 引擎，就像众所周知的 V8 引擎，我们经常接触的 Node.js 也是用的它。

2.2 DOM 渲染层与 GPU 硬件加速

如果我告诉你，一个页面是有许多许多层级组成的，他们就像千层面那样，你能想象出这个页面实际的样子吗？这里为了便于大家想象，我附上一张之前 Firefox 的 3D View 插件的页面 Layers 层级图。

对，你没看错，页面的真实样子就是这样，是由多个 DOM 元素渲染层（Layers）组成的，实际上一个页面在构建完 render tree 之后，是经历了这样的流程才最终呈现在我们面前的。

- 浏览器会先获取 DOM 树并依据样式将其分割成多个独立的渲染层。

- CPU 将每个层绘制进绘图中。
- 将位图作为纹理上传至 GPU（显卡）绘制。
- GPU 将所有的渲染层缓存（如果下次上传的渲染层没有发生变化，GPU 就不需要对其进行重绘）并复合多个渲染层最终形成我们的图像。

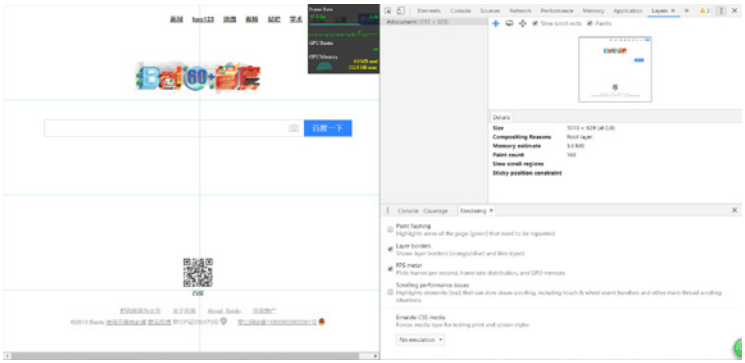
从上面的步骤我们可以知道，布局是由 CPU 处理的，而绘制则是由 GPU 完成的。

其实在 chrome 中，也为我们提供了相关插件供我们查看页面渲染层的分布情况，以及 GPU 的占用率。（所以说，平时我们得多去尝试尝试 chrome 的那些莫名其妙的插件，真的会发现好多东西都是神器）

chrome 开发者工具菜单→ more tools → Layers（开启渲染层功能模块）

chrome 开发者工具菜单→ more tools → rendering（开启渲染性能监测工具）

执行上面的操作后，你会在浏览器里看到这样的效果：



太多东西了，分模块讲吧：

1. 最先是页面右上方的小黑窗：其实提示已经说的很清楚了，它显示的就是我们的 GPU 占用率，能够让我们清楚地知道页面是否发生了大量的重绘。

2. Layers 版块：这就是用于显示我们刚提到的 DOM 渲染层的工具了，左侧的列表里将会列出页面里存在哪些渲染层，还有这些渲染层的详细信息。

3. Rendering 版块：这个版块和我们的控制台在同一个地方，大家可别找不到它。前三个勾选项是我们最常使用的，让我来给大家解释一下他们的功能（充当一次免费翻译）

① Paint flashing：勾选之后会对页面中发生重绘的元素高亮显示

② Layer borders：和我们的 Layer 版块功能类似，它会用高亮边界突出我们页面中的各个渲染层

③ FPS meter：就是开启我们在（一）中提到的小黑窗，用于观察我们的 GPU 占用率

可能大家会问我，和我提到 DOM 渲染层这么深的概念有什么用啊，好像跟性能优化没一点关系啊？大家应该还记得我刚说到 GPU 会对我们的渲染层作缓存对吧，那么大家试想一下，如果我们把那些一直发生大量重排重绘的元素提取出来，单独触发一个渲染层，那样这个元素不就不会“连累”其他元素一块重绘了对吧。

那么问题来了，什么情况下会触发渲染层呢？大家只要记住：

video 元素、WebGL、Canvas、CSS3 3D、CSS 滤镜、z-index 大于某个相邻节点的元素都会触发新的 Layer，其实我们最常用的方法，就是给某个元素加上下面的样式：

```
transform: translateZ(0);
backface-visibility: hidden;
```

这样就可以触发渲染层啦。

我们把容易触发重排重绘的元素单独触发渲染层，让它与那些“静态”元素隔离，让 GPU 分担更多的渲染工作，我们通常把这样的措施成为硬件加速，或者是 GPU 加速。大家之前肯定听过这个说法，现在完全清楚它的原理了吧。

2.3 重排与重绘

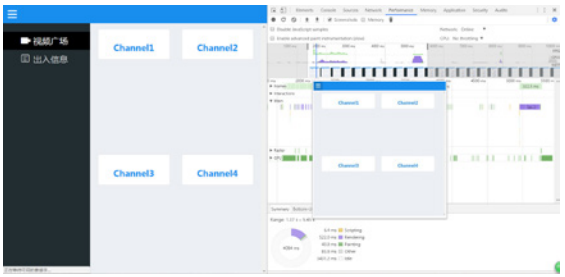
现在到我们的重头戏了，重排和重绘。先抛出概念：

1. 重排（reflow）：渲染层内的元素布局发生修改，都会导致页面重新排列，比如窗口的尺寸发生变化、删除或添加 DOM 元素，修改了影响元素盒子大小的 CSS 属性（诸如：width、height、padding）。

2. 重绘（repaint）：绘制，即渲染上色，所有对元素的视觉表现属性的修改，都会引发重绘。

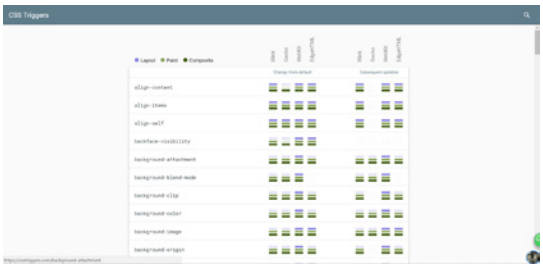
我们习惯使用 chrome devtools 中的 performance 版块来测量页面重排重绘所占据的时间。

- 蓝色部分：HTML 解析和网络通信占用的时间
- 黄色部分：JavaScript 语句执行所占用时间
- 紫色部分：重排占用时间
- 绿色部分：重绘占用时间



不论是重排还是重绘，都会阻塞浏览器。要提高网页性能，就要降低重排和重绘的频率和成本，近可能少地触发重新渲染。正如我们在 2.3 中提到的，重排是由 CPU 处理的，而重绘是由 GPU 处理的，CPU 的处理效率远不及 GPU，并且重排一定会引发重绘，而重绘不一定会引发重排。所以在性能优化工作中，我们更应当着重减少重排的发生。

这里给大家推荐一个网站 https://csstriggers.com/ ，里面详细列出了哪些 CSS 属性在不同的渲染引擎中是否会触发重排或重绘。



2.4 优化策略

谈了那么多理论，最实际不过的，就是解决方案，大家一定都等着急了吧，做好准备，一大波干货来袭：

1. CSS 属性读写分离：浏览器没次对元素样式进行读操作时，都必须进行一次重新渲染（重排 + 重绘），所以我们在使用 JS 对元素样式进行读写操作时，最好将两者分离开，先读后写，避免出现两者交叉使用的情况。最最最客观的解决方案，就是不用 JS 去操作元素样式，这也是我最推荐的。

2. 通过切换 class 或者 style.csstext 属性去批量操作元素样式

3. DOM 元素离线更新：当对 DOM 进行相关操作时，例、appendChild 等都可以使用 Document Fragment 对象进行离线操作，带元素“组装”完成后再一次插入页面，或者使用 display:none 对元素隐藏，在元素“消失”后进行相关操作。

4. 将没用的元素设为不可见：visibility: hidden，这样可以减小重绘的压力，必要的时候再将元素显示。

5. 压缩 DOM 的深度，一个渲染层内不要有过深的子元素，少用 DOM 完成页面样式，多使用伪元素或者 box-shadow 取代。

6. 图片在渲染前指定大小：因为 img 元素是内联元素，所以在加载图片后会改变宽高，严重的情况会导致整个页面重排，所以最好在渲染前就指定其大小，或者让其脱离文档流。

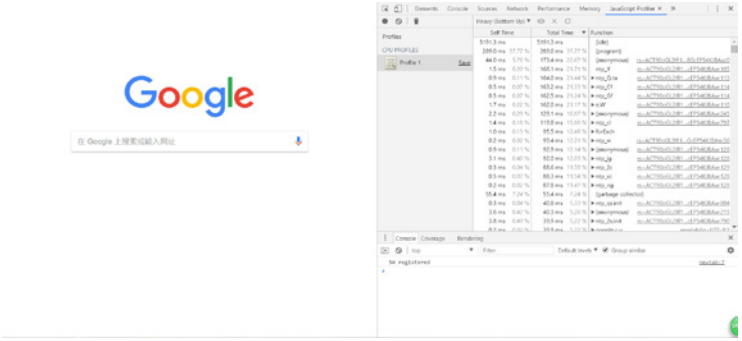
7. 对页面中可能发生大量重排重绘的元素单独触发渲染层，使用 GPU 分担 CPU 压力。（这项策略需要慎用，得着重考量以牺牲 GPU 占用率能否换来可期的性能优化，毕竟页面中存在太多的渲染层对与 GPU 而言也是一种不必要的压力，通常情况下，我们会对动画元素采取硬件加速。）

3. JS 阻塞性能

JavaScript 在网站开发中几乎已经确定了垄断地位，哪怕是一个再简单不过的静态页面，你都可能看到 JS 的存在，可以说，没有 JS，就基本没有用户交互。然而，脚本带来的问题就是他会阻塞页面的平行下载，还会提高进程的 CPU 占用率。更有甚者，现在 node.js 已经在前端开发中普及，稍有不慎，我们引发了内存泄漏，或者在代码中误写了死循环，会直接造成我们的服务器奔溃。在如今这个 JS 已经遍布前后端的时代，性能的瓶颈不单单只是停留在影响用户体验上，还会有更多更为严重的问题，对 JS 的性能优化工作不可小觑。

在编程的过程中，如果我们使用了闭包后未将相关资源加以释放，或者引用了外链后未将其置空（比如给某 DOM 元素绑定了事件回调，后来却 remove 了该元素），都会造成内存泄漏的情况发生，进而大量占用用户的 CPU，造成卡顿或死机。我们可以使用 chrome 提供的 JavaScript Profile 版块，开启方式同 Layers 等版块，这里我就不再多说了，直接上效果图。

我们可以清除看见 JS 执行时各函数的执行时间以及 CPU 占用情况，如果我在代码里增加一行 while(true){}, 那么它的占用率一定会飙升到一个异常的指标（亲测 93.26%）。



其实浏览器强大的内存回收机制在大多数时候避免了这一情况的发生，即便用户发生了死机，他只要结束相关进程（或关闭浏览器）就可以解决这一问题，但我们要知道，同样的情况还会发生在我们的服务器端，也就是我们的 node 中，严重的情况，会直接造成我们的服务器宕机，网站奔溃。所以更多时候，我们都使用 JavaScript Profile 版块来进行我们的 node 服务的压力测试，搭配 node-inspector 插件，我们能更有效地检测 JS 执行时各函数的 CPU 占用率，针对性地进行优化。

（PS: 没修炼到一定水平，千万别在服务端使用闭包，一个是真没啥用，我们会有更多优良的解决办法，二是真的很容易内存泄漏，造成的后果是你无法预期的）

4.【拓展】负载均衡

之所以将负载均衡作为拓展内容，是因为如果是你自己搭建的个人网站，或者中小型网站，其实并不需要考虑多大的并发量，但是如果你搭建的是大型网站，负载均衡便是开发过程不可或缺的步骤。

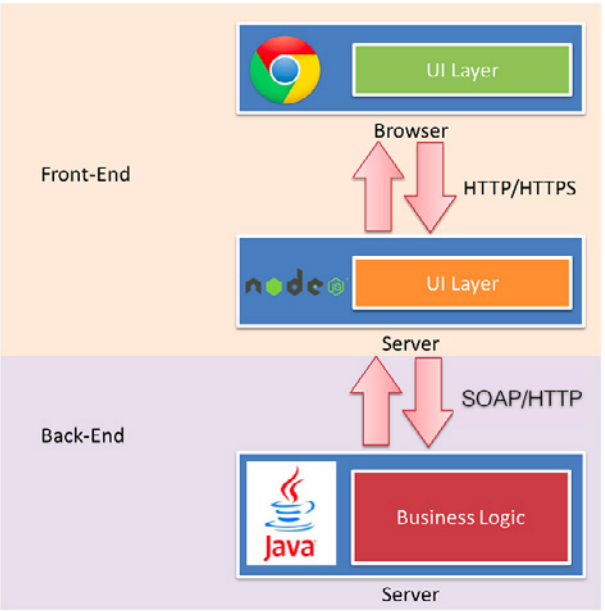
4.1 Node.js 处理 IO 密集型请求

现在的开发流程都注重前后端分离，也就是软件工程中常提到的“高内聚低耦合”的思想，你也可以用模块化的思想去理解，前后解耦就相当与把一个项目分成了前端和后端两个大模块，中间通过接口联系起来，分别进行开发。这样做有什么好处？我就举最有实际效果的一点：“异步编程”。这是我自己想的名字，因为我觉得前后解耦的形式很像我们 JS 中的异步队列，传统的开发模式是“同步”的，前端需要等后端封装好接口，知道了能拿什么数据，再去开发，时间短，工程大。而解耦之后，我们只需要提前约定好接口，前后两端就可以同时开发，不仅高效而且省时。

我们都知道 node 的核心是事件驱动，通过 loop 去异步处理用户请求，相比于传统的后端服务，它们都是将用户的每个请求分配异步队列进行处理，推荐大家去看这样一篇博文。特别生动地讲解了事件驱动的运行机制，通俗易懂。事件驱动的最大优势是什么？就是在高并发 IO 时，不会造成堵塞，

对于直播类网站，这点是至关重要的，我们有成功的先例——快手，快手强大的 IO 高并发究其本质一定能追溯到 node。

其实现在的企业级网站，都会搭建一层 node 作为中间层。大概的网站框架如图所示。

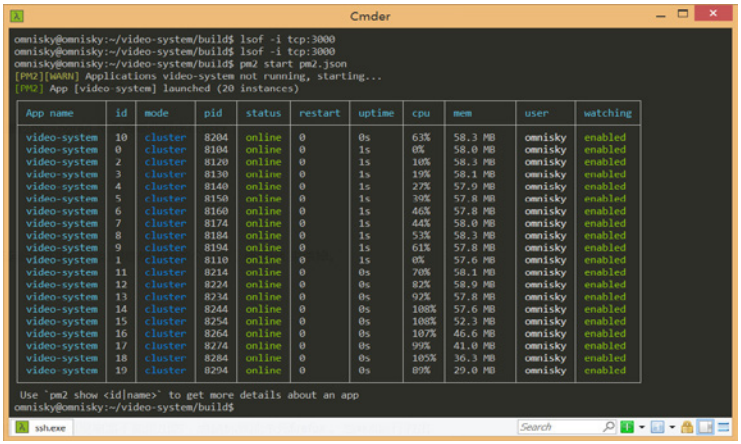


4.2 pm2 实现 Node.js“多线程”

我们都知道 node 的优劣，这里分享一份链接，找了挺久写的还算详细：https://www.zhihu.com/question/19653241/answer/15993549 。其实都是老套路，那些说 node 不行的都是指着 node 是单线程这一个软肋开撕。

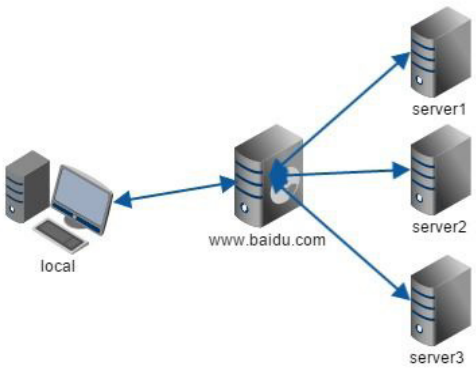
告诉你，我们有解决方案了——pm2。这是它的官网：http://pm2.keymetrics.io/ 。它是一款 node.js 进程管理器，具体的功能，就是能在你的计算机里的每一个内核都启动一个 node.js 服务，也就是说如果你的电脑或者服务器是多核处理器（现在也少见单核了吧），它就能启动多个 node.js 服务，并且它能够自动控制负载均衡，会自动将用户的请求分发至压力小的服务进程上处理。听起来这东西简直就是神器啊！而且它的功能远远不止这些，这里我就不作过多介绍了，大家知道我们在上线的时候需要用到它就行了，安装的方法也很简单，直接用 npm 下到全局就可以了 $ npm i pm2 -g 具体的使用方法还有相关特性可以参照官网。这里我在 build 文件夹内添加了 pm2.json 文件，这是 pm2 的启动配置文件，我们可以自行配置相关参数，具体可参考 github 源码，运行时我们只要在上线目录下输入命令 $ pm2 start pm2.json 即可。

下面是 pm2 启动后的效果图。



4.3 nginx 搭建反向代理

在开始搭建工作之前，首先得知道什么是反向代理。可能大家对这个名词比较陌生，先上一张图:



所谓代理就是我们通常所说的中介，网站的反向代理就是指那台介于用户和我们真实服务器之间的服务器（说的我都拗口了），它的作用便是能够将用户的请求分配到压力较小的服务器上，其机制是轮询。听完这句话是不是感觉很耳熟，没错，在我介绍 pm2 的时候也说过同样的话，反向代理起到的作用同 pm2 一样也是实现负载均衡，你现在应该也明白了两者之间的差异，反向代理是对服务器实现负载均衡，而 pm2 是对进程实现负载均衡。

大家如果想深入了解反向代理的相关知识，我推荐知乎的一个贴子。但是大家会想到，配服务器是运维的事情啊，和我们前端有什么关系呢？的确，在这部分，我们的工作只有一些，只需要向运维提供一份配置文档即可。

```
http {
 upstream video {
     ip_hash;
     server localhost:3000;
 }
```

```
server {
    listen: 8080;
    location / {
        proxy_pass: http://video
    }
  }
}
```

也就是说，在和运维对接的时候，我们只需要将上面这几行代码改为我们配置好的文档发送给他就行了，其他的事情，运维小哥会明白的，不用多说，都在酒里。

但是，这几行代码该怎么去改呢？首先我们得知道，在 nginx 中，模块被分为三大类：handler、filter 和 upstream。而其中的 upstream 模块，负责完成完成网络数据的接收、处理和转发，也是我们需要在反向代理中用到的模块。接下来我们将介绍配置代码里的内容所表示的含义。

4.3.1 upstream 配置信息

upstream 关键字后紧跟的标识符是我们自定义的项目名称，通过一对花括号在其中增添我们的配置信息。

ip_hash 关键字：控制用户再次访问时是否连接到前一次连接的服务器。

server 关键字：我们真实服务器的地址，这里的内容肯定是需要我们去填写的，不然运维怎么知道你把项目放在那个服务器上了，也不知道你封装了一层 node 而得去监听 3000 端口。

4.3.2 server 配置信息

server 是 nginx 的基本配置，我们需要通过 server 将我们定义的 upstream 应用到服务器上。

listen 关键字：服务器监听的端口。

location 关键字：和我们之前在 node 层说到的路由是起同样的功能，这里是把用户的请求分配到对应的 upstream 上。

5. 拓展阅读

网站的性能与监测是一项复杂的工作，还有很多很多后续的工作，我之前所提到的这些，也只能算是冰山一角，在熟悉开发规范的同时，也需要实践经验的积累。

在翻阅了许多与网站性能相关的书籍后，我还是更钟情于唐文前辈编著的《大型网站性能监测、分析与优化》，里面的知识较新，切合实际，至少我读完一遍后很有收获、醍醐灌顶，我也希望对性能感兴趣的读者在看完我的文章后能去翻翻这本著作。

2019 年大前端技术趋势深度解读

作者 狼叔（桑世龙）

大家好，我是阿里巴巴前端技术专家狼叔，今天想跟你们分享 2019 年我对前端现状及未来发展趋势的理解。

我其实特别反感很多人说“前端娱乐圈”这种话，诚然，爆发式增长必然会带来焦点，但也不必过度解读，2018 年的几件大事儿我都了解，真的不是大家看到的那样的。学会辩证的看问题，用心去体味背后的趋势，我想这比所谓的“正直”更有价值，我更希望大家能够坚持学习，保持思辨和平和。

大前端

2018 年的事儿特别多，从 React v16 普及，到 jQuery 被 GitHub 下掉完成阶段性历史使命，在唏嘘之外，版本帝 AngularJS 又发布了 v6 和 v7 两个版本。这些其实都不算啥大新闻，反观三大框架，写法越来越像，越来越贴近 WebComponents 标准，而周边应用层面的封装已经开始指数级增长。小程序是今年最火的技术，接连出现，快应用也想分一杯羹。PWA 进入稳定期，尤其是 PWA 桌面版，可以让我们更好的看清楚 PC 桌面版开发的全貌。移动端还是以强运营为主，各大公司都开始不再 all in 移动，开始重视多端并进，到了开始拼细节的阶段了。TypeScript 全面开花，GraphQL 蠢蠢欲动，WebAssembly 更是打开了浏览器上多语言的大门。所有的这一切都在暗示，浏览器即操作系统，你能

想象到未来前端的样子么？下面跟着我一一进行解读吧。

三大框架标准化

有朋友吐槽：“Vue 的特点就是上手快，初期相当好用，但如果接手一个别人写的 Vue 项目，再和 React 对比一下，你会感谢 React 的”。但当 Vue 3.0 发布之后，估计他就不会这样说了。因为 Vue 3 的 Class API 和 React 的写法几乎是一模一样的，这个改动不是 Proxy 和 TypeScript，而是支持原生 Class 的写法。你用 Class 来写，那代码和 React 写法几乎是一模一样的！

```
import Vue from 'vue'

class App extends Vue.Component {
      count = 0

      up() {
    this.count++
  }

  down() {
    this.count--
  }

  render() {
    return (
      <div>
        <h1>{this.count}</h1>
        <button onClick={() => this.up()}>+</button>
        <button onClick={() => this.down()}>-</button>
      </div>
    )
  }
}

Vue.render(<App />, document.body as HTMLElement)
```

从上面的讨论可以看出，前端三大框架已经趋于平稳化、标准化，在我看来未来是 WebComponents 的。

WebComponents 是规范，最早最知名的一个是 Google 主推的 JavaScript 库 Polymer，它可帮助我们创建自定义的可重用 HTML 元素，并使用它们来构建高性能、可维护的 App。在 I/O 大会上，Google 推出了 Polymer 3.0，Polymer 3.0 致力于将 Web 组件的生态系统从 HUML Imports 转移到 ES Modules，包管理系统将支持 npm，这使你更容易将基于 Polymer 的 Web 组件和你喜欢的工具、框架协同使用。

```
<script src="node_modules/@webcomponents/webcomponents-loader.js"></script>
 <script type="module">
```

```
  import {PolymerElement, html} from '@polymer/polymer';

  class MyElement extends PolymerElement {
    static get properties() { return { mood: String }}
    static get template() {
      return html`
        <style> .mood { color: green; } </style>
        Web Components are <span class="mood">[[mood]]</span>!
      `;
    }
  }

  customElements.define('my-element', MyElement);
</script>

<my-element mood="happy"></my-element>
```

另外还有 2 个项目具有一定的参考价值：

1. Rax 也提供了一个名为 atag 的 UI WebComponents 库。
2. LitElement 是一个简单的轻量级的快速创建 WebComponents 的基类，可以理解成是 Polymer 最小的实现版本。

LitElement 主要的特性包括 WebComponent 生命周期模型支持和单向的数据绑定模型。它遵守 WebComponents 标准，使用 lit-html 模块这个快速的 HTML 渲染引擎定义和渲染 HTML 模板。最重要的是它对浏览器兼容性非常好，对主流浏览器都能有非常好的支持。由于 LitHtml 基于 tagged template，结合 ES6 中的 template，使得它无需预编译、预处理，就能获得浏览器原生支持，并且扩展能力更强，性能更好。

```
import { LitElement, html } from '@polymer/lit-element';

// Create your custom component
class CustomGreeting extends LitElement {
  // Declare properties
  static get properties() {
    return {
      name: { type: String }
    };
  }
  // Initialize properties
  constructor() {
    super();
    this.name = 'World';
  }
  // Define a template
  render() {
```

```
    return html`<p>Hello, ${this.name}!</p>`;
  }
}

// Register the element with the browser
customElements.define('custom-greeting', CustomGreeting);
```

是不是看着更眼熟了？《三国演义》里有这样一句：“话说天下大势，分久必合，合久必分。周末七国分争，并入于秦。及秦灭之后，楚、汉分争，又并入于汉。汉朝自高祖斩白蛇而起义，一统天下，后来光武中兴，传至献帝，遂分为三国。”

前端从 2014 年到 2017 年是混战期，得益于 Node.js 的辅助加成，外加各种前端优秀的创意和实践，使得 React/Vue/Angular 三足鼎立。无论 React 发布 v16，增加 Fiber 和 Hooks，还是 Vue 3.0 发布，其实最终都是朝着 W3C WebComponents 标准走。一言以蔽之，Follow 标准是趋势，如果能够引领标准，那将是框架的无上荣耀。

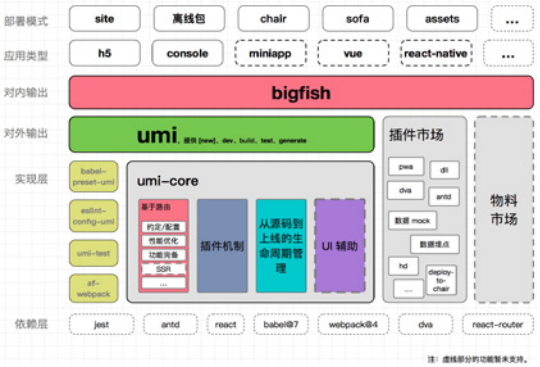
我们可以参考一下技术成熟度曲线（Hype Cycle -Wikipedia），这个曲线把技术发展分成五个步骤:【科技诞生的促动期】—»【过高期望的峰值】—» 【泡沫化的底谷期】—» 【稳步爬升的光明期】—»【实质生产的高原期】。从前端现在的热度来看，应该是处于从第三阶段【泡沫化的底谷期】到第四阶段【稳步爬升的光明期】的爬坡过程，创新不会那么多，更多的是偏于应用层的内容。

对于当下的前端发展情况，我其实是有隐忧的。当年 Java 世界曾经搞各种 GUI，创造了 MVC 模式，结果没火，没想到到了 Web 开发领域，MVC 成了基本约定。之后 Model 1 和 Model 2 等企业开发模型渐渐成熟，出现了 Struts、Spring、Hibernate 三大框架。在之后很长的一段时间里，Java 程序员都是言必称“SSH”。再之后 Spring 一家独大，一统江湖，恐怕今天还记得 EJB 的人已经不多了。框架一旦稳定，就会有大量培训跟进，导致规模化开发。这是把双刃剑，能满足企业开发和招人的问题，但也给创新探索领域上了枷锁。

应用层封装进入爆发期

框架和工程化基本探索稳定后，大家就开始思考如何更好的用，更简单的用。目前，各家大厂都在前端技术栈思考如何选型和降低成本，统一技术栈。

举个例子 Umi，Umi 就是一套零配置（约定高于配制），按最佳实践进行开发的，开箱即用的前端框架 : React 全家桶 + dva + jest + antd (mobile) + less + eslint。如下图所示：



从上图中可以看出，Umi 已经思考的相对全面，从技术选型、构建到多端输出、性能优化、发布等方面进行了拆分，使得 Umi 的边界更为清晰，是前端最佳实践，目前大多数前端组都是类似的实现方式。说白了，Umi 和 create-react-app（cra）类似，就是现有技术栈的组合，封装细节，让开发者用起来更简单，只写业务代码就可以了。

- 零配置就是默认选型都给你做好了。
- 开箱即用就是技术栈都固化了。
- 约定大于配置，开发模式也固化好了。

Umi 的核心是 af-webpack 模块，它封装了 Webpack 和各种插件，把 webpack-dev-server 等 Node.js 模块直接打包进去，同时对配置做了更好的处理以及插件化。af-webpack 核心是 webpack-chain 模块，通过链式写法来修改 Webpack 配置，使得 af-webpack 极为灵活。其实以 React 全家桶为例，开发者最大的负担就是 Webpack 工程化构建。关于 Webpack 的封装实践有很多，比如知名的还有 YKit、EasyWebpack 等。

- YKit 是去哪儿开源的 Webpack，内置 Connect 作为 Web server，结合 dev 和 hot 中间件，对于多项目构建提效明显，对版本文件发布有不错的实践。
- EasyWebpack 也是插件化，但对解决方案、boilerplate 等做了非常多的集成，比如 Egg 的 SSR 是有深入思考的，尽管不赞同这种做法。

另外，在 create-react-app（cra）项目里使用的是 react-scripts 作为启动脚本，它和 egg-scripts 类似，也是通过约定，隐藏具体实现细节，让开发者不需要关注构建。在未来，类似的封装还会有更多的封装，并且更偏于应用层面。

PWA 进入稳定期

PWA 和 native app（移动应用）的核心区别在于以下几点：

1. 安装：PWA 是一个不需要下载安装即可使用的应用。

2. 缓存使用：native app 主要是对 sqlite 缓存，以及文件读写操作，而 PWA 对缓存数据库操作支持的非常好，足以应对各种场景。

3. 基本能力补齐，比如推送。

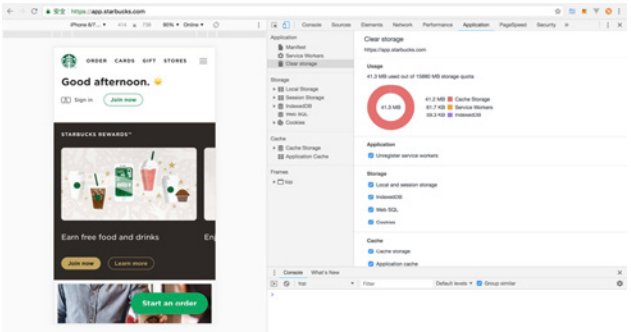
现在 PWA 已经支持的很好了，唯一麻烦的是缓存策略和发版比较麻烦，应用轻量化的趋势已经很明朗了。下面讲一下 PWA 的一些关键点。

1. 通用本地存储的解决方案 Workbox

Workbox 是 GoogleChrome 团队推出的一套 Web App 静态资源和请求结果本地存储的解决方案，该解决方案包含一些 JS 库和构建工具，Workbox 背后则是 Service Worker 和 Cache API 等技术和标准在驱动。在 Workbox 之前，GoogleChrome 团队较早时间推出过 sw-precache 和 sw-toolbox 库，但骂声很多，直到 Workbox 才真正诞生了能方便统一的处理离线能力的更完美的方案。

Workbox 现在已经发布到了 3.0 版本，不管你的站点是用何种方式构建的，它都可以为你的站点

提供离线访问能力，几乎不用考虑太多的具体实现，只用做一些配置就可以。就算你不考虑离线能力，它也能让你的站点访问速度更快。



比如星巴克的 PWA 应用，对缓存的应用高达 41.3mb。这是浏览器端非常大的突破，尽管没啥新技术。

2. PWA 桌面版

纵观 PC 桌面端的发展过程，早期 Delphi/VB/VF/VC 等构建起的 c/s 时代，即使到今天依然有很大的量。最近两年，随着 Atom/ VSCode 的火爆，带动了 node webkit 相关模块的爆发，比如 NW.js 和 Electron 等。通过 Web 技术来构建 pc client，确实是省时省力，用户体验也非常好，比如钉钉客户端、石墨文档客户端等，最主要的是可以统一技术栈，比如某些算法，用 JS 写一次，之后可以到前端、node、pc client 等多处复用。当然更好的是使用 Web 技术进行开发，不需要加壳打包，PWA 桌面版就是这样的技术。

接下来就具体聊一下桌面端的 3 个发展阶段。

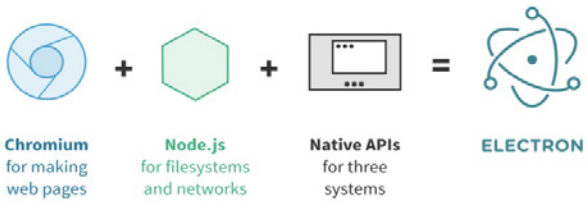
第一阶段：原生开发 Native

早年的 VB/VF/VC/Delphi 等原生开发方式，到后来出现 QT 类的跨平台软件，但依然可以理解成是原生开发。

第二阶段：混搭开发 Hybrid

谷歌于 2008 年 9 月 2 日首次发布了 Chrome 浏览器，Node.js 是 Ryan Dahl 于 2009 年发布的，他把 V8 引擎（Chrome 核心 JavaScript 引擎）搬到了后端，使用 js 编写服务器程序变为现实。随后 npm 发展极为迅猛，跨平台技术也突飞猛进，出现了 NW.js 这样的轻量级跨平台框架，基于 Chromium（Chrome 开源版本） + Node.js，使得 PC 桌面端能够通过 Web 开发技术开发，最终打包编译成各个平台支持的应用格式，给 PC 桌面开发带来了太多的可能性。

而 Atom 是 GitHub 在 2014 年发布的一款基于 Web 技术构建的文本编辑器，其中 atom-shell，也就是后来的 Electron，是和 NW.js 类似的技术。它允许使用 Node.js（作为后端）和 Chromium（作为前端）来完成桌面 GUI 应用程序的开发。Chromium 提供了渲染页面和响应用户交互的能力，而 Node.js 提供了访问本地文件系统和网络的能力，也可以使用 NPM 上的几十万个第三方包。在此基础之上，Electron 还提供了 Mac、Windows、Linux 三个平台上的一些原生 API，例如全局快捷键、文件选择框、托盘图标和通知、剪贴板、菜单栏等。



Erich Gamma 老爷子设计的 Monaco/VS Code，同样基于 Electron，但性能比 Atom 强多了。VS Code 会先启动一个后台进程，也就是 Electron 的主进程，它负责编辑器的生命周期管理、进程间通讯、UI 插件管理、升级和配置管理等。后台进程会启动一个（或多个）渲染进程，用于展示编辑器窗口，它负责编辑器的整个 UI 部分，包括组件、主题、布局管理等等。每个编辑器窗口都会启动一个 Node.JS 子进程作为插件的宿主进程，在独立进程里跑插件逻辑，然后通过事件或者回调的方式通知 Renderer 结果，避免了 Renderer 的渲染被插件中 JS 逻辑阻塞。

演进过程：chrome > Node.js > nw.js > atom(electron) > vs code

在第二阶段里，我们可以看到 PC 桌面端以 Web 开发技术作为核心，以浏览器内核作为跨平台核心，最终将 Web 开发的代码和浏览器内核打包。这样做的好处是前端开发相对简单，相对于 C++ 等语言更为方便，另外从成本上考虑，也是极为划算的。

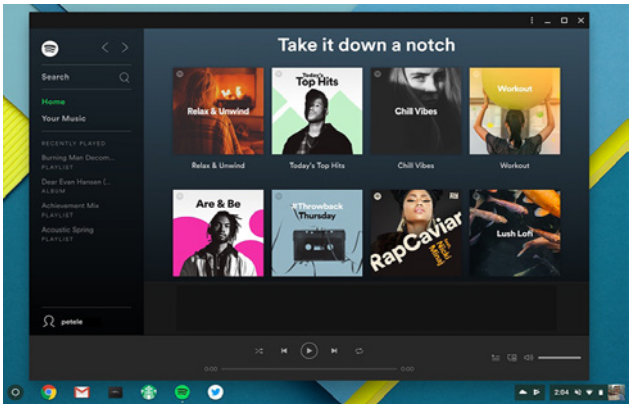
如今，很多应用都开始基于 Electron 构建，比如微信小程序 ide、微信 pc 版本等，另外非常令人激动的是，2018 年 10 月 18 日，迅雷论坛发文称新版（从迅雷 X 10.1 版本开始）采用 Electron 软件框架完全重写了迅雷主界面。使用新框架的迅雷 X 可以完美支持 2K、4K 等高清显示屏，界面中的文字渲染也更加清晰锐利。从技术层面来说，新框架的界面绘制、事件处理等方面比老框架更加灵活高效，因此界面的流畅度也显著优于老框架的迅雷。

第三阶段：**PWA 桌面版**

王国维在《人间词话》中提出“隔与不隔”这一文学命题，这个问题在开发领域也是存在的。明明是 Web 开发的，为什么还要打包加壳呢？除了体积非常大以外，使用安装也极为麻烦。

Spotify 的 PWA 桌面版应用体验是非常好的，在 mac 上丝般顺滑。

2018 年 Google IO 大会上，微软宣布 win10 全力拥抱 PWA，通过爬虫爬取 PWA 页面，并将其转



换为 Appx，继而在其应用商店里提供应用，体验和原生 Native 应用非常相近，对此我非常看好。

浏览器有着超强的缓存能力，外加 PWA 其他功能，使得浏览器上的 PWA 应用能够取得媲美 Native 应用的性能。在浏览器里可以直接打开，无需加壳，很明显这是极为方便的。

PWA 必然会改变前端与移动端之间的格局，再加之 AOT(ahead-of-time) 与 WebAssembly 为 JS 带来的性能上的突破，JavaScript 将撼动所有领域，从移动端（PWA）到桌面应用、物联网、VR、AR、游戏乃至人工智能等等。

Google 接下来会大力推进 PWA 的桌面版，再加上 win10 和 Chrome 加持，Web 应用无需加壳就能达到近乎原生的体验，前端的领域再一次被拓宽，未来真的可以大胆的想想。

很多人问 PWA 在国内为什么感觉不火，原因很简单，PWA 在弱网环境下表现极好，但中国的网络是全球最好的，所以 PWA 其实没有给我们带来那么大的收益。不过当做一个补位方案也挺好的，毕竟 2G/3G 还有点量，另外在服务器渲染 SSR 上，PWA 也能够起到很好的效果。

小程序火爆

如果说和 PWA 比较像的，大概就是小程序了，小程序也可以说是今年最火的技术。

微信小程序的下一步计划，支持 NPM、小程序云、可视化编程、支持分包等，听起来很美好，但坑依然不少。小程序原生提供的 DSL 不够好用，所以就有了上层开发框架或者脚手架来优化开发效率，目前比较主流的有 3 个。

开发框架	维护者	框架简介
WePY	腾讯	借鉴了 Vue 的语法，封装了微信小程序的接口，优化小程序的开发体验。对于组件化、npm、ES6+ 等特性支持的比较好。
mpVue	美团	直接 fork 并改造了 Vue，底层对接到微信小程序的 API。可以使用 Vue 的语法开发小程序，能复用 Vue 社区的一部分库和框架。
Taro	京东	基于 React 的语法开发小程序，目标是实现一份代码在微信小程序、H5、React Native 环境中运行。

今年还冒出了微信小程序之外的头条小程序、支付宝小程序、百度智能小程序等，未来还会有很多。同时，手机厂商大概是看到了小程序对其应用商店的威胁，小米、华为、OPPO、vivo 等九大国内手机厂商联手成立了“快应用联盟”，基于 react-native 技术栈，整体也很不错，尤其是天猫调用菜鸟裹裹的快应用，安卓下有非常好的体验。相较而言，微信是基于 Webview 的，而快应用使用的是原生渲染方案，其他家也大抵如此。

其实 5G 时代很快就到了，大家可以畅想一下，在网速、内存和 CPU 更高的情况下，5G 每秒最高下载速度高达 1.4G，秒下 PWA 或小程序应用，到底是离线，还是在线，犹未可知吧。

前端能讲的东西实在太多了，但受限于篇幅，本文只能先简单跟你分享 React/Vue/Angular 三大框架标准化、应用层封装进入爆发期、PWA 进入稳定期、小程序火爆等方面的内容。下一篇文章中，我将继续跟你聊聊移动端局面、多端拉齐的必然性等内容，以及 2019 年不可忽视的 TypeScript 和 WebAssembly 这两大技术，欢迎继续关注，也欢迎留言与我多多交流。

多端拉齐，并重用户体验

在 AI 时代，没有“端”的支持可以么？明显是不可以的。首先感谢苹果，将用户体验提升到了前无古人的位置。移动互联网兴起后，PC Web 日渐没落。我个人非常欣赏玉伯，在当年无线 ALL IN 战略中，他还是选择留下来继续做 PC Web 的前端。不过，虽然很多公司的重点转向无线，但 PC 业务也一直没停，这是很多公司的现状，也是客观事实。那么，PC 端这样的“老古董”的出路到底在哪里呢？

1. 我们可以利用 PC/H5 快速发版本的优势，快速验证 AI 算法，继而为移动端提供更好的模型和数据上的支撑。

2. 多端对齐，打好组合拳。既然不能在移动端有更大的突破，大家只能在细节上血拼。

大家的战场已经不是点了，已经升级到打组合策略的阶段了。未来一定是多端拉齐，并重用户体验的。

今天的大前端，除了 Web 外，还包括各种端，比如移动端、OTT，甚至是一些新的物联网设备。我们有理由相信 Chrome OS 当年的远见：“给我一个浏览器，我就能给你一个世界。”如果说的苟且一点：“给我一个 Webview，我就能给你一个世界。”

TypeScript

我之前就非常关注 TypeScript，但迟迟未下定决心在团队内落地。今年 1 月份北京 Node Party 上组了个局，和几位嘉宾一起聊了一下，确认提效非常明显，落地难度也不大，大家一致认为 2019 年 TypeScript 将有非常大的增长。本身前端团队变大，规模化编程也必然依赖类型系统和面向对象的，从这点上看，TypeScript 也是完胜的。

这里再简单介绍一下 TypeScript，它是有类型定义的 JavaScript 的超集，包括 ES5、ES5+ 和其他一些诸如反射、泛型、类型定义、命名空间等特征的集合，为了大规模 JavaScript 应用开发而生。复杂软件需要用复杂的设计，面向对象就是一种很好的设计方式，使用 TypeScript 的一大好处就是 TypeScript 提供了业界认可的类（ ES5+ 也支持）、泛型、封装、接口面向对象设计能力，以提升 JavaScript 的面向对象设计能力。市面上的框架也对 TypeScript 提供了非常好的支持。

React 对 .tsx 支持非常好，比如我在 Midway controller 里支持 tsx 写法，这是非常大胆的，对于后面 react ssr 来说是一个极大便利；

Vue 从 v2.5.0 之后对 ts 支持就非常好；

Node.js Web 框架，尤其是 Egg.js 对 ts 支持非常好，当然还有更高级更专注的的 Midway 框架，Midway 基于 Egg 生态，同时提供 IoC 等高级玩法；

在使用 Webpack 编译前端应用式，通过 TypeScript-loader 可以很轻松地将 TypeScript 引入到 Webpack 中。有了 TypeScript-loader，就可以一边使用 TypeScript 编写新代码，一边零碎地更新老代码。毕竟 ts 是 js 超集，你有空就改，非强制，特别包容。

WebAssembly

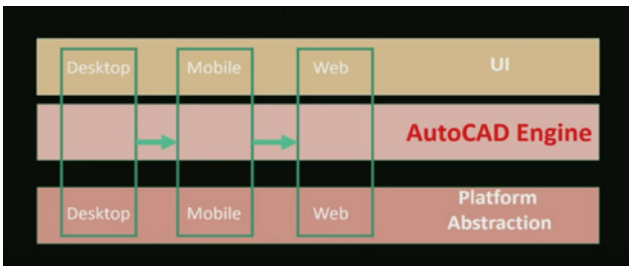
WebAssembly 是一种新的字节码格式，目前主流浏览器都已经支持 WebAssembly。和 JS 需要解释执行不同的是，WebAssembly 字节码和底层机器码很相似，可以快速装载运行，因此性能相对于 JS 解释执行而言有了极大的提升。也就是说 WebAssembly 并不是一门编程语言，而是一份字节码标准，需要用高级编程语言编译出字节码放到 WebAssembly 虚拟机中才能运行，浏览器厂商需要做的就是根据 WebAssembly 规范实现虚拟机。这很像 Java 早年的 Applet，能够让其他语言运行在浏览器里。Applet 是一种 Java 程序，它可以运行在支持 Java 的 Web 浏览器内。因为它有完整的 Java API 支持，所以 Applet 是一个全功能的 Java 应用程序。

有了 WebAssembly，在浏览器上可以跑任何语言。从 Coffee 到 TypeScript，到 Babel，这些都是需要转译为 js 才能被执行的，而 WebAssembly 是在浏览器里嵌入 vm，直接执行，不需要转译，执行效率自然高得多。

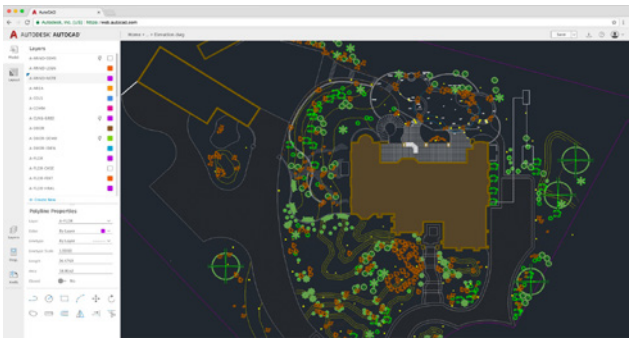
举个例子，AutoCAD 软件是由美国欧特克有限公司（Autodesk）出品的一款自动计算机辅助设计软件，可以用于绘制二维制图和基本三维设计。使用它时，无需懂得编程，即可自动制图，因此它

在全球被广泛应用于土木建筑、装饰装潢、工业制图、工程制图、电子工业、服装加工等诸多领域。

AutoCAD 是由大量 C++ 代码编写的软件，经历了非常多的技术变革，从桌面到移动端再到 web。之前，InfoQ 上有一个演讲，题目是《AutoCAD & WebAssembly: Moving a 30 Year Code Base to the Web》，即通过 WebAssembly，让很多年代久远的 C++ 代码在 Web 上可以运行，并且保证了执行效率。



本来，我以为 WebAssembly 离我们很远，但在 2018 年 Google I/O 大会亲眼见到 AutoCad Web 应用后，非常震撼，效果如下图所示。



能够让如此庞大的项目跑在 Web 端，真的是非常了不起。通过 WebAssembly 技术，既能复用之前的 C++ 代码，又能完成 Web 化，这也许就是所谓的两全其美吧。

之前，全民直播的前端研发经理赵洋曾分享了 WebAssembly 在全民直播里对直播编解码方面的应用，效果也非常不错。

另外，许式伟在 ECUG Con 2018 上也分享了一个 Topic，主题是《再谈 Go 语言在前端的应用前景》，Go 的发展也遇到了瓶颈，专注后端开发是没办法让 Go 排到第一的，目前的一个方向是借助 GopherJS，将 Go 代码编译为 JS。这种实践是没问题的，和 Kotlin 类似，对于绝大部分 Go 用户也是非常好的。但问题在于，真正的前端不太可能换语言，目前连 Babel、ts 这种都折腾的心累，更何况切换到 Go。“求别更新了，老子学不动了”，这是大部分前端工程师的心声。

从 WebAssembly 的现状来看，对于复杂计算耗时的部分采用其他语言实现，确实是比较好的一种方式。从趋势上看，WebAssembly 让所有语言都能跑在浏览器上，浏览器上有了 vm，浏览器不就是操作系统了吗?

Chrome 的核心 JavaScript 引擎 V8 目前已包含了 Liftoff 这一新款 WebAssembly baseline 编译器。

Liftoff 简单快速的代码生成器极大地提升了 WebAssembly 应用的启动速度。不过在桌面系统上，V8 依然会通过让 TurboFan 在后台重新编译代码的方式最终让代码运行性能达到峰值。

目前，V8 v6.9 (Chrome 69) 中的 Liftoff 已经设置为默认工作状态，也可以显式地通过 --liftoff/–no-liftoff 或者 chrome://flags/#enable-webassembly-baseline 开关来控制。另外，Node.js v11 采用的 v8 引擎的 v7 版本，对 WebAssembly 支持更好，虽然这没啥意义，但练手还是蛮好的。

移动端

Flutter 是 Google 推出的帮助开发者在 Android 和 iOS 两个平台，同时开发高质量原生应用的全新移动 UI 框架，和 React-native/Weex 一样支持热更新。Flutter 使用 Google 自己家的 Dart 语言编写，刚好今年 Dart 2 也正式发布，不知道二者之间是否有关联。目前 Dart 主攻 Flutter 和 Web 两块，同时提供了 pub 包管理器，俨然是一门全新的语言，学习成本有些高。反观 TypeScript 就非常容易被接受，基于 npm 生态，兼容 ES 语法，因此，2019 年对 Dart 我还是会持观望态度。

除了不喜欢 Dart 外，Flutter 的其他方面都很好，在移动端现在强运营的背景下，支持热更新是必备能力。

关于 Weex，一边骂一边用，很无奈的一种状态。Weex 本身是好东西，捐给了 Apache，目前在孵化中，会有一个不错的未来。但社区维护的非常差，问题 issue 不及时，文档不更新。如果公司没有架构组，还是比较难搞定的。

不过也有很多不错的案例，比如 2018 年优酷双十一活动就是使用 Weex 开发的，效果非常不错。通过自建的可视化活动搭建平台，能够非常高效的完成开发，结合 App 内的缓存，整体效果比 H5 好的多。

我对 Weex 的看法是，以前 Weex 只是解决 H5 渲染效率的问题，但如今强运营的背景，使得 Weex 承载了非常多的内容，比如动画、游戏甚至是图形图像处理等。可以看到，未来 Weex 还会战略性的增加。

总结

总结一下，2018 年大前端的现象：

前端三大框架已趋于平稳，标准化，向 Web Components 看齐。

应用层面开始进入过渡封装周边的阶段，很多细节都会埋在框架里。

PWA 平稳发展，兼容 4/5 浏览器，workbox 3 进一步简化开发，另外 PWA 桌面版已经开始兴起，未来会更多。

多端受到重视，不再只是 all in mobile。

WebAssembly 让更多语言可以运行在浏览器上，AutoCAD 的 web 版是非常好的例子。

强运营背景下，移动端以前端开发为主，已成定局。Flutter 局势暂不好说，还在观望中（主要是不喜欢 Dart）。

TypeScript 落地很好，包容性更好：React 对 .tsx 支持非常好，Vue 从 v2.5.0 之后对 ts 支持就非常好，Node.js（尤其是 Egg.js、midway）对 ts 支持也非常好。

5G 时代快来了，互联网的长期在线情况有可能会被打破。本地设备即客户端，可以大胆的想想。对前端来说，本地 web 服务，辅助日常开发，类似于 je 这样的模块会越来越多。

终上所述，未来浏览器会越来越重要，Web Os 的概念正在慢慢落地。另外三大框架趋于稳定，写法上也越来越像，学习成本是降低的。但周边应用层面的封装还会是爆发式增长，更多复杂的细节会被包装到应用框架里，可能还有很多不一样的开发方式需要大家熟悉。

对于开发者而言，唯一不变的就是学习能力。掌握了学习能力就能够应对这些趋势变化，无论是在三大框架混战时代，还是后面周边封装时代都能很开心的“折腾”。哪怕有一天 AI 真的能够替人写代码，能应变的人自然也是不怕的。

关于大前端的现状和未来我就分享到这里，希望能对你有所帮助。