

Web (/archive/?tag=Web)

PWA (/archive/?tag=PWA)

饿了么的 PWA 升级实践

Upgrading Ele.me to Progressive Web App

Posted by Hux on July 12, 2017

中文 | Chinese

很荣幸在今年 2 月到 5 月的时间里，以顾问的身份加入饿了么，参与 PWA 的相关工作。这篇文章其实最初是在以英文写作发表在 medium 上的：*Upgrading Ele.me to Progressive Web Apps* (<https://medium.com/elemefe/upgrading-ele-me-to-progressive-web-app-2a446832e509>)，获得了一定的关注。所以也决定改写为中文版本再次分享出来，希望能对你有所帮助；）

本文首发于 CSDN (<http://geek.csdn.net/news/detail/210535>) 与《程序员》2017 年 7 月刊，同步发布于 饿了么前端 - 知乎专栏 (<https://zhuanlan.zhihu.com/ElemeFE>)、Hux Blog (<https://huangxuan.me>)，转载请保留链接。

自 Vue.js 官方推特第一次公开 (<https://twitter.com/vuejs/status/834087199008239619>) 到现在，我们就一直在进行着将饿了么移动端网站 (<https://h5.ele.me/msite/#pwa=true>) 升级为 Progressive Web App (<https://developers.google.com/web/progressive-web-apps/>) 的工作。直到近日在 Google I/O 2017 上登台亮相 (<https://m.weibo.cn/status/4109332495285652>)，才终于算告一段落。我们非常荣幸能够发布全世界第一个专门面向国内用户的 PWA，但更荣幸的是能与 Google、UC 以及腾讯合作，一起推动国内 web 与浏览器生态的发展。

多页应用、Vue、PWA？

对于构建一个希望达到原生应用级别体验的 PWA，目前社区里的主流做法都是采用 SPA，即单页面应用模型（Single-page App）来组织整个 web 应用，业内最有名的几个 PWA 案例 Twitter Lite (<https://blog.twitter.com/2017/how-we-built-twitter-lite>)、Flipkart Lite (<https://medium.com/progressive-web-apps/building-flipkart-lite-a-progressive-web-app-2c211e641883>)、Housing Go (<https://medium.com/engineering-housing/progressing-mobile-web-fac3efb8b454>) 与 Polymer Shop (<https://shop.polymer-project.org/>) 无一例外。

然而饿了么，与很多国内的电商网站一样，青睐多页面应用模型（MPA，Multi-page App）所能带来的一些好处，也因此在一年多将移动站从基于 Angular.js 的单页应用重构为目前的多页应用模型。团队最看重的优点莫过于页面与页面之间的隔离与解耦，这使得我们可以将每个页面当做一个独立的“微服务”来看待，这些服务可以被独立迭代，独立提供给各种第三方的入口嵌入，甚至被不同的团队独立维护。而整个网站则只是各种服务的集合而非一个巨大的整体。

与此同时，我们仍然依赖 Vue.js (<https://vuejs.org/>) 作为 JavaScript 框架。Vue 除了是 React/Angular 这种“重型武器”的竞争对手外，其轻量与高性能的优点使得它同样可以作为传统多页应用开发中流行的“jQuery/Zepto/Kissy + 模板引擎”技术栈的完美替代。Vue 提供的组件系统、声明式与响应式编程更是提升了代码组织、共享、数据流控制、渲染等各个环节的开发效率。Vue 还是一个渐进式框架 (https://www.youtube.com/watch?v=pBBSp_ilivM)，如果网站的复杂度继续提升，我们可以按需、增量地引入 Vuex 或 Vue-Router 这些模块。万一哪天又要改回单页呢？（谁知道呢……）

2017 年，PWA 已经成为 web 应用新的风潮。我们决定试试，以我们现有的“Vue + 多页”的架构，能在升级 PWA 的道路上走多远，达到怎样的效果。

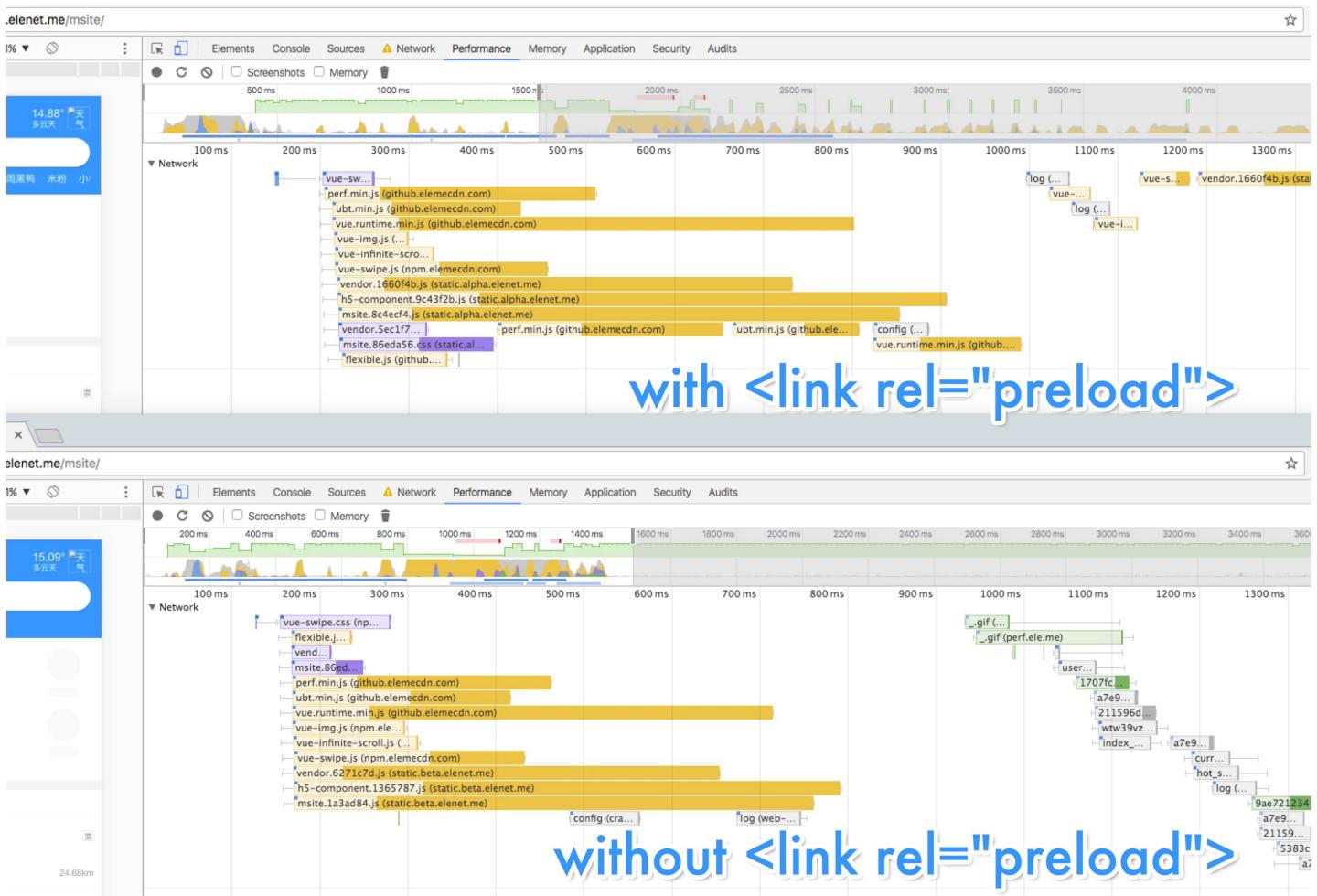
实现 “PRPL” 模式

“PRPL” (<https://developers.google.com/web/fundamentals/performance/prpl-pattern/>)（读作“purple”）是 Google 的工程师提出的一种 web 应用架构模式，它旨在利用现代 web 平台的新技术以大幅优化移动 web 的性能与体验，对如何组织与设计高性能的 PWA 系统提供了一种高层次的抽象。我们并不准备从头重构我们的 web 应用，不过我们可以把实现 “PRPL” 模式作为我们的迁移目标。“PRPL”实际上是 Push/Preload、Render、Precache、Lazy-Load 的缩写，我们会在下文中展开它们的具体含义。

1. PUSH/PRELOAD，推送/预加载初始 URL 路由所需的关键资源。

无论是 HTTP2 Server Push 还是 `<link rel="preload">`，其关键都在于，我们希望提前请求一些隐藏在应用依赖关系（Dependency Graph）较深处的资源，以节省 HTTP 往返、浏览器解析文档、或脚本执行的时间。比如说，对于一个基于路由进行 code splitting 的 SPA，如果我们在 webpack 清单、路由等入口代码（entry chunks）被下载与运行之前就把初始 URL，即用户访问的入口 URL 路由所依赖的代码用 Server Push 推送或 `<link rel="preload">` 进行提前加载。那么当这些资源被真正请求时，它们可能已经下载好并存在在缓存中了，这样就加快了初始路由所有依赖的就绪。

在多页应用中，每一个路由本来就只会请求这个路由所需要的资源，并且通常依赖也都比较扁平。饿了么移动站的大部分脚本依赖都是普通的 `<script>` 元素，因此他们可以在文档解析早期就被浏览器的 preloader 扫描出来并且开始请求，其效果其实与显式的 `<link rel="preload">` 是一致的。



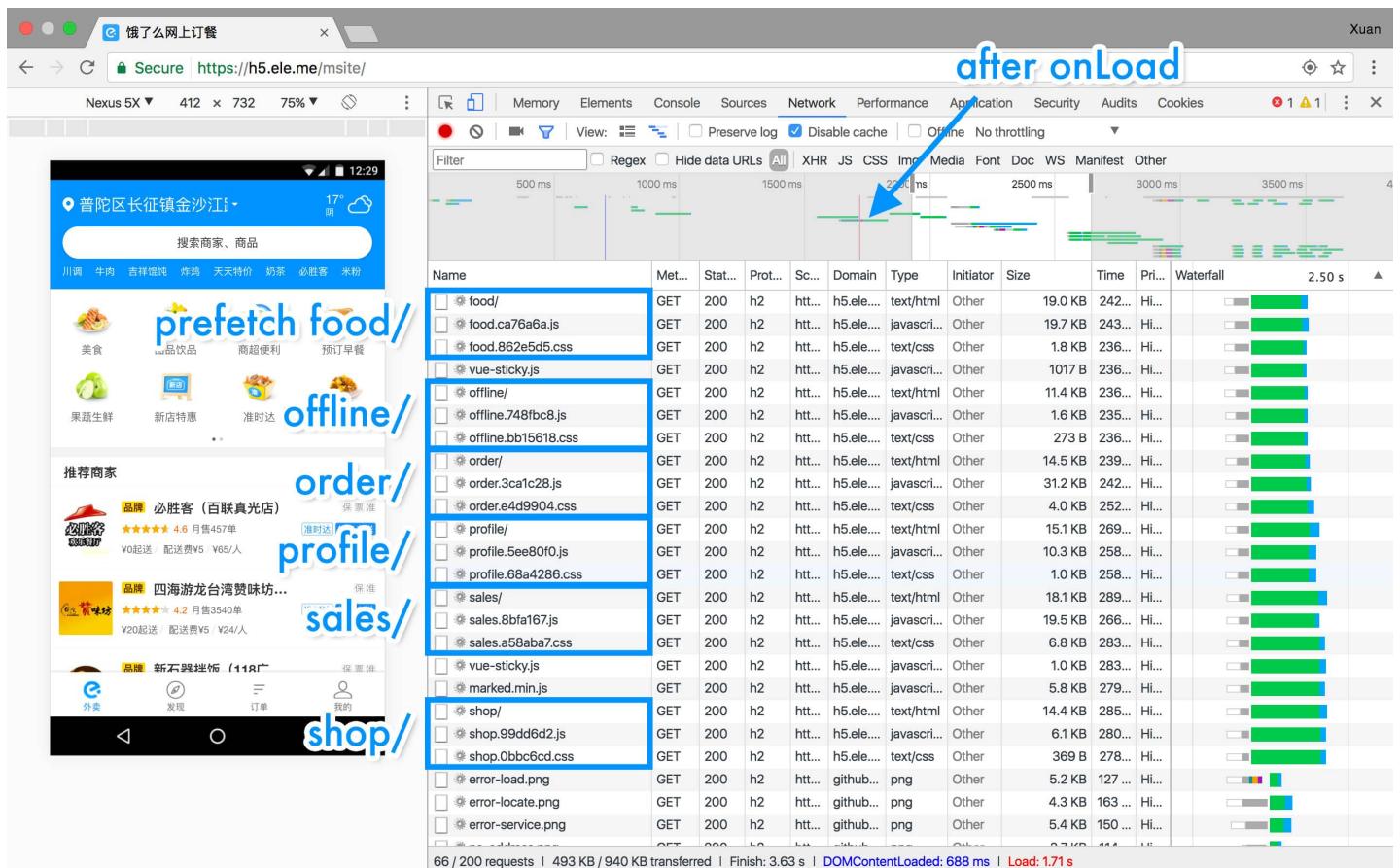
我们还将所有关键的静态资源都伺服在同一域名下（不再做域名散列），以更好的利用 HTTP2 带来的多路复用（Multiplexing）。同时，我们也在进行着对 API 进行 Server Push 的实验（<https://zhuanlan.zhihu.com/p/26757514>）。

2. RENDER, 渲染初始路由，尽快让应用可被交互

既然所有初始路由的依赖都已经就绪，我们就可以尽快开始初始路由的渲染，这有助于提升应用诸如首次渲染时间、可交互时间等指标。多页应用并不使用基于 JavaScript 的路由，而是传统的 HTML 跳转机制，所以对于这一部分，多页应用其实不用额外做什么。

3. PRE-CACHE, 用 Service Worker 预缓存剩下的路由

这一部分就需要 Service Worker (<https://w3c.github.io/ServiceWorker/v1/>) 的参与了，Service Worker 是一个位于浏览器与网络之间的客户端代理，它以可拦截、处理、响应流经的 HTTP 请求，使得开发者得以从缓存中向 web 应用提供资源而闻名。不过，Service Worker 其实也可以主动发起 HTTP 请求，在“后台”预请求与预缓存我们未来所需要的资源。



我们已经使用 Webpack (<https://webpack.github.io/>) 在构建过程中进行 .vue 编译、文件名哈希等工作，于是我们编写了一个 webpack 插件来帮助我们收集需要缓存的依赖到一个“预缓存清单”中，并使用这个清单在每次构建时生成新的 Service Worker 文件。在新的 Service Worker 被激活时，清单里的资源就会被请求与缓存，这其实与 SW-Precache 这个库的运行机制 (<https://medium.com/@Huxpro/how-does-sw-precache-works-2d99c3d3c725>) 非常接近。

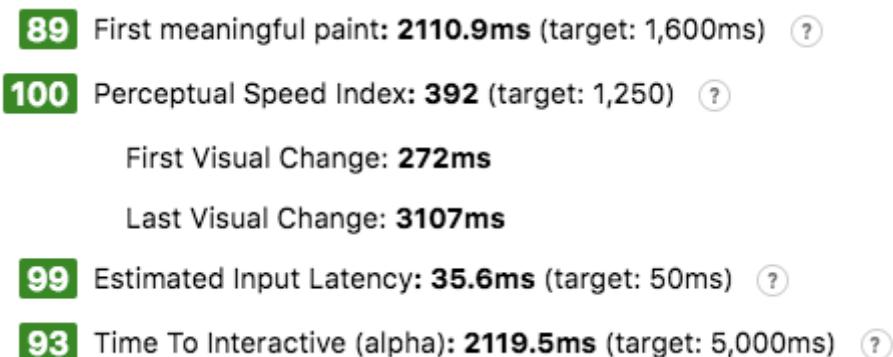
实际上，我们只对我们标记为“关键路由”的路由进行依赖收集。你可以将这些“关键路由”的依赖理解为我们整个应用的 “App Shell” (<https://developers.google.com/web/updates/2015/11/app-shell>) 或者说“安装包”。一旦它们都被缓存，或者说成功安装，无论用户是在线离线，我们的 web 应用都可以从缓存中直接启动。对于那些并不那么重要的路由，我们则采取在运行时增量缓存的方式。我们使用的 SW-Toolbox (<https://googlechrome.github.io/sw-toolbox/>) 提供了 LRU 替换策略与 TTL 失效机制，可以保证我们的应用不会超过浏览器的缓存配额。

4. LAZY-LOAD 按需懒加载、懒实例化剩下的路由

懒加载与懒实例化剩下的路由对于 SPA 是一件相对麻烦点儿的事情，你需要实现基于路由的 code splitting 与异步加载。幸运的是，这又是一件不需要多页应用担心的事情，多页应用中的各个路由天生就是分离的。

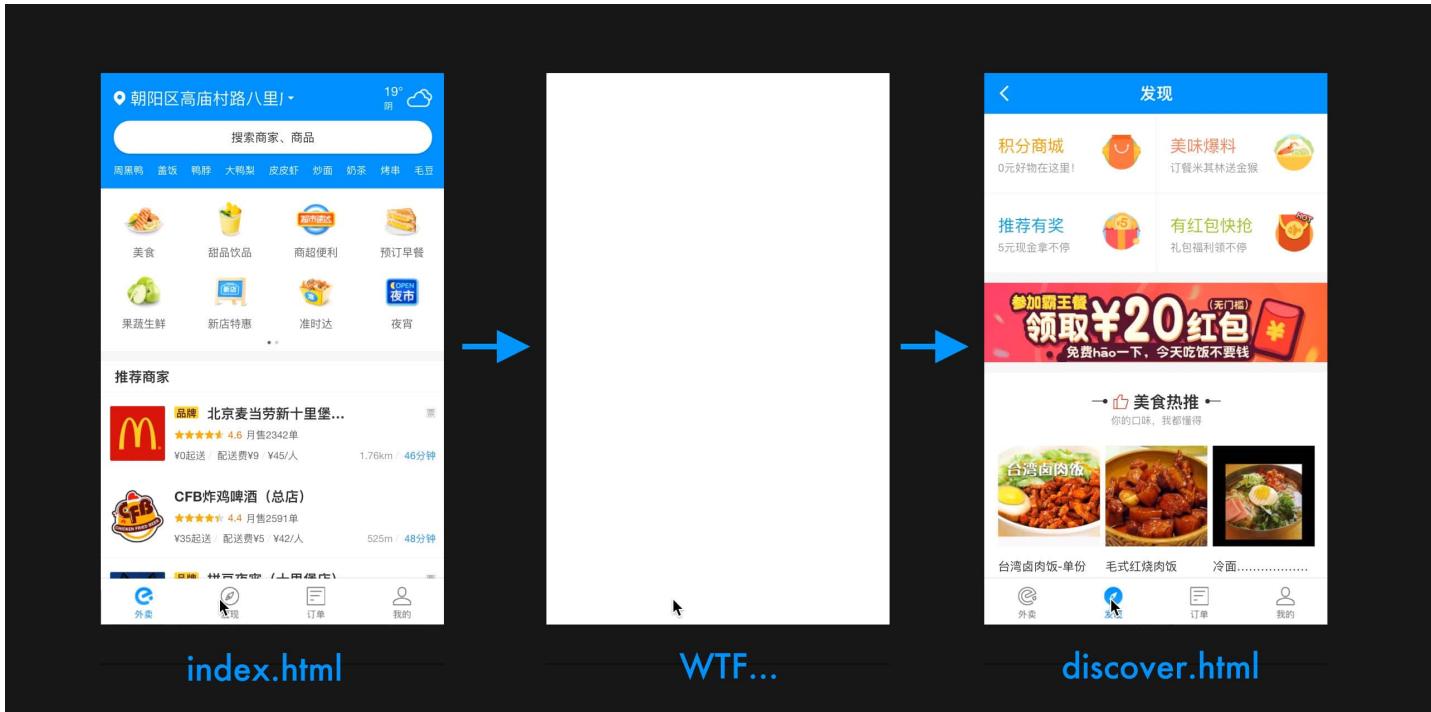
值得说明的是，无论单页还是多页应用，如果在上一步中，我们已经将这些路由的资源都预先下载与缓存好了，那么懒加载就几乎是瞬时完成的了，这时候我们就只需要付出实例化的代价。

这四句话即是 PRPL 的全部了。有趣的是，我们发现多页应用在实现 PRPL 这件事甚至比单页还要容易一些。那么结果如何呢？



根据 Google 推出的 Web 性能分析工具 Lighthouse (v1.6) , 在模拟的 3G 网络下, 用户的初次访问(无任何缓存)大约在 2 秒左右达到“可交互”,可以说非常不错。而对于再次访问,由于所有资源都直接来自于 Service Worker 缓存,页面可以在 1 秒左右就达到可交互的状态了。

但是,故事并不是这么简单得就结束了。在实际的体验中我们发现,应用在页与页的切换时,仍然存在着非常明显的白屏空隙,由于 PWA 是全屏运行的,白屏对用户体验所带来的负面影响甚至比以往在浏览器内更大。我们不是已经用 Service Worker 缓存了所有资源了吗,怎么还会这样呢?

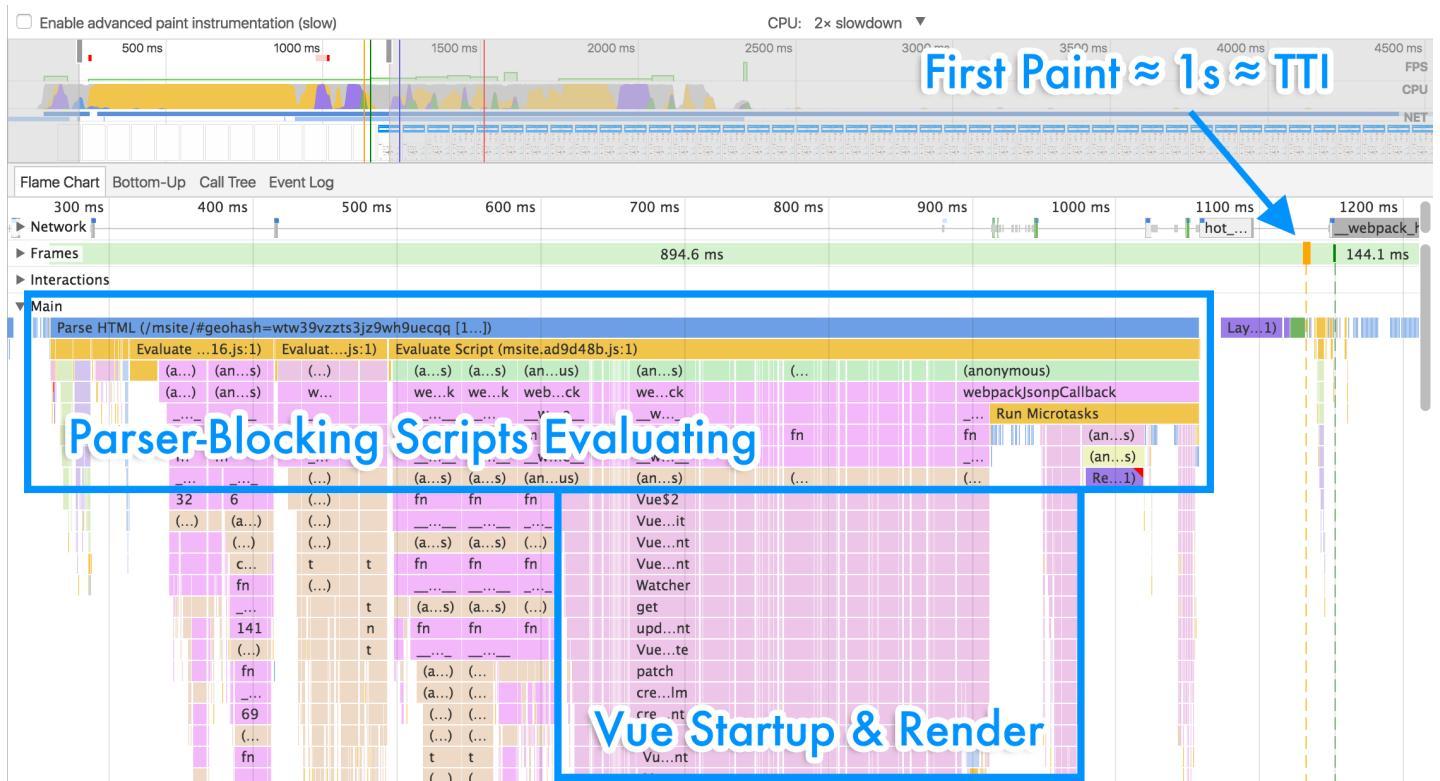


从首页点击到发现页，跳转过程中的白屏

多页应用的陷阱：重启开销

与 SPA 不同,在多页应用中,路由的切换是原生的浏览器文档跳转 (Navigating across documents),这意味着之前的页面会被完全丢弃而浏览器需要为下一个路由的页面重新执行所有的启动步骤:重新下载资源、重新解析 HTML、重新运行 JavaScript、重新解码图片、重新布局页面、重新绘制.....即使其中的很多步骤本是在多个路由之间复用的。这些工作无疑将产生巨大的计算开销,也因此需要付出相当的时间成本。

图中为我们的入口页（同时也是最重的页面）在 2 倍 CPU 节流模拟下的 profile 数据。即使我们可以将“可交互时间”控制在 1 秒左右，我们的用户仍然会觉得这对于“仅仅切换个标签”来说实在是太慢了。



巨大的 JavaScript 重启开销

根据 Profile，我们发现在首次渲染（First Paint）发生之前，大量的时间（900 毫秒）都消耗在了 JavaScript 的运行上（Evaluate Script）。几乎所有脚本都是阻塞的（Parser-blocking），不过因为所有的 UI 都是由 JavaScript/Vue 驱动的，倒也不会有性能影响。这 900ms 中，约一半是消耗在包括 Vue 运行时、组件、库等依赖的运行上，而另一半则花在了业务组件实例化时 Vue 的启动与渲染上。从软件工程角度来说，我们需要这些抽象，所以这里并不是想责怪 JavaScript 或是 Vue 所带来的开销。

但是，在 SPA 中，JavaScript 的启动成本是均摊到整个生命周期的：每个脚本都只需要被解析与编译一次，诸如生成 Virtual DOM 等较重的任务可以只执行一次，像 Vue 的 ViewModel 或是 Virtual DOM 这样的大对象也可以被留在内存里复用。可惜在多页应用里就不是这样了，我们每次切换页面都为 JavaScript 付出了巨大的重启代价。

浏览器的缓存啊，能不能帮帮忙？

能，也不能。

V8 提供了代码缓存 (code caching) (<https://v8project.blogspot.com/2015/07/code-caching.html>)，可以将编译后的机器码在本地拷贝一份，这样我们就可以在下次请求同一个脚本时一次省略掉请求、解析、编译的所有工作。而且，对于缓存在 Service Worker 配套的 Cache Storage 中的脚本，会在第一次执行后就触发 V8 的代码缓存，这对于我们的多页切换能提供不少帮助。

另外一个你或许听过的浏览器缓存叫做“进退缓存”，Back-Forward Cache，简称 bfcache。浏览器厂商对其的命名各异，Opera 称之为 Fast History Navigation，Webkit 称其为 Page Cache。但是思路都一样，就是我们可以让浏览器在跳转时把前一页留在内存中，保留 JavaScript 与 DOM 的状态，而不是全都销毁掉。你可以随便找个传统的多页网站在 iOS Safari 上试试，无论是通过浏览器的前进后退按钮、手势，还是通过超链接（会有一些不同），基本都可以看到瞬间加载的效果。

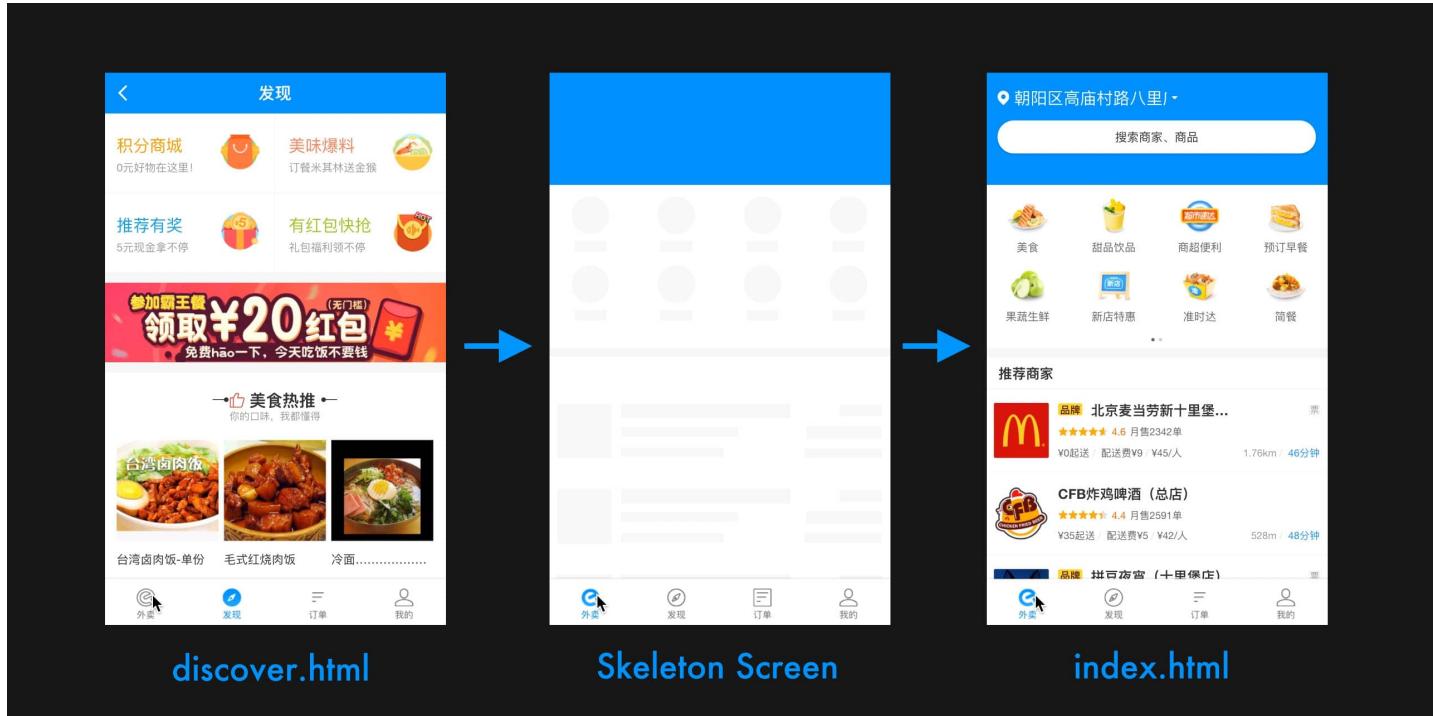
Bfcache 其实非常适合多页应用。但不幸的是，Chrome 由于内存开销与其多进程架构等原因目前并不支持。Chrome 现阶段仅仅只是用了传统的 HTTP 磁盘缓存，来稍稍简化了一下加载过程而已。对于 Chromium 内核霸占的 Android 生态来说，我们没法指望了。

为“感知体验”奋斗

尽管多页应用面临着现实中的不少性能问题，我们并不想这么快就妥协。一方面，我们尝试尽可能减少在页面达到可交互时间前的代码执行量，比如减少/推迟一些依赖脚本的执行，还有减少初次渲染的 DOM 节点数以节省 Virtual DOM 的初始化开销。另一方面，我们也意识到应用在感知体验上还有更多的优化空间。

Chrome 产品经理 Owen 写过一篇 Reactive Web Design: The secret to building web apps that feel amazing (<https://medium.com/@owencm/reactive-web-design-the-secret-to-building-web-apps-that-feel-amazing-b5cbfe9b7c50>)，谈到两种改进感知体验的手段：一是使用骨架屏 (Skeleton Screen) 来实现瞬间加载；二是预先定义好元素的尺寸来保证加载的稳定。跟我们的做法可以说不谋而合。

为了消除白屏时间，我们同样引入了尺寸稳定的骨架屏来帮助我们实现瞬间的加载与占位。即使是在硬件很弱的设备上，我们也可以在点击切换标签后立刻渲染出目标路由的骨架屏，以保证 UI 是稳定、连续、有响应的。我录了两个 (<https://youtu.be/K5JBGnMYO1s>) 视频 (<https://youtu.be/w1ZbNsHmRjs>) 放在 Youtube 上，不过如果你是国内读者，你可以直接访问饿了么移动网站来体验实地的效果 ;) 最终效果如下图所示。



在添加骨架屏后，从发现页点回首页的效果

这效果本该很轻松的就能实现，不过实际上我们还费了点功夫。

在构建时使用 Vue 预渲染骨架屏

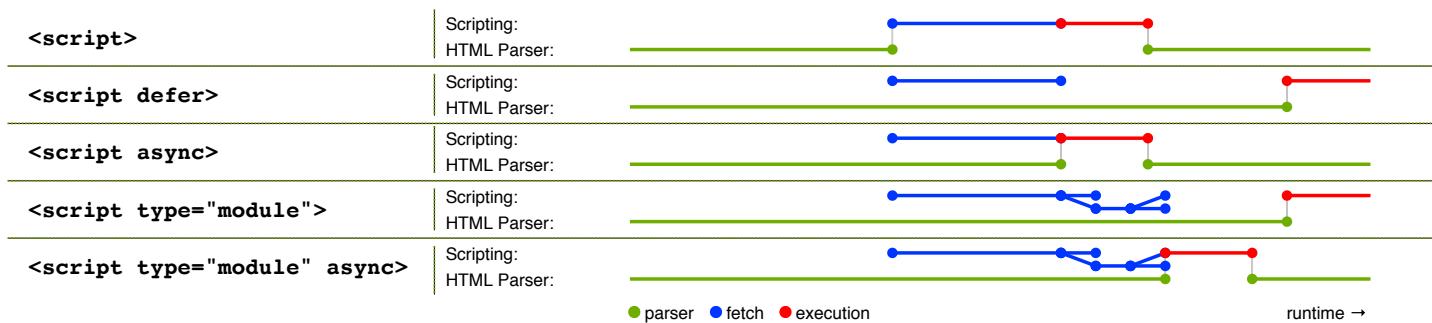
你可能已经想到了，为了让骨架屏可以被 Service Worker 缓存，瞬间加载并独立于 JavaScript 渲染，我们需要把组成骨架屏的 HTML 标签、CSS 样式与图片资源一并内联至各个路由的静态 *.html 文件中。

不过，我们并不准备手动编写这些骨架屏。你想啊，如果每次真实组件有迭代（每一个路由对我们来说都是一个 Vue 组件）我们都需要手动去同步每一个变化到骨架屏的话，那实在是太繁琐且难以维护了。好在，骨架屏不过是当数据还未加载进来前，页面的一个空白版本而已 (<https://www.lukew.com/ff/entry.asp?1797>)。如果我们能将骨架屏实现为真实组件的一个特殊状态——“空状态”的话，我们理论上就可以从真实组件中直接渲染出骨架屏来。

而 Vue 的多才多艺就在这时体现出来了，我们真的可以用 Vue.js 的服务端渲染模块 (<https://ssr.vuejs.org/en/>) 来实现这个想法，不过不是用在真正的服务器上，而是在构建时用它把组件的空状态预先渲染成字符串并注入到 HTML 模板中。你需要调整你的 Vue 组件代码使得它可以在 Node 上执行，有些页面对 DOM/BOM 的依赖一时无法轻易去除得，我们目前只好额外编写一个 `*.shell.vue` 来暂时绕过这个问题。

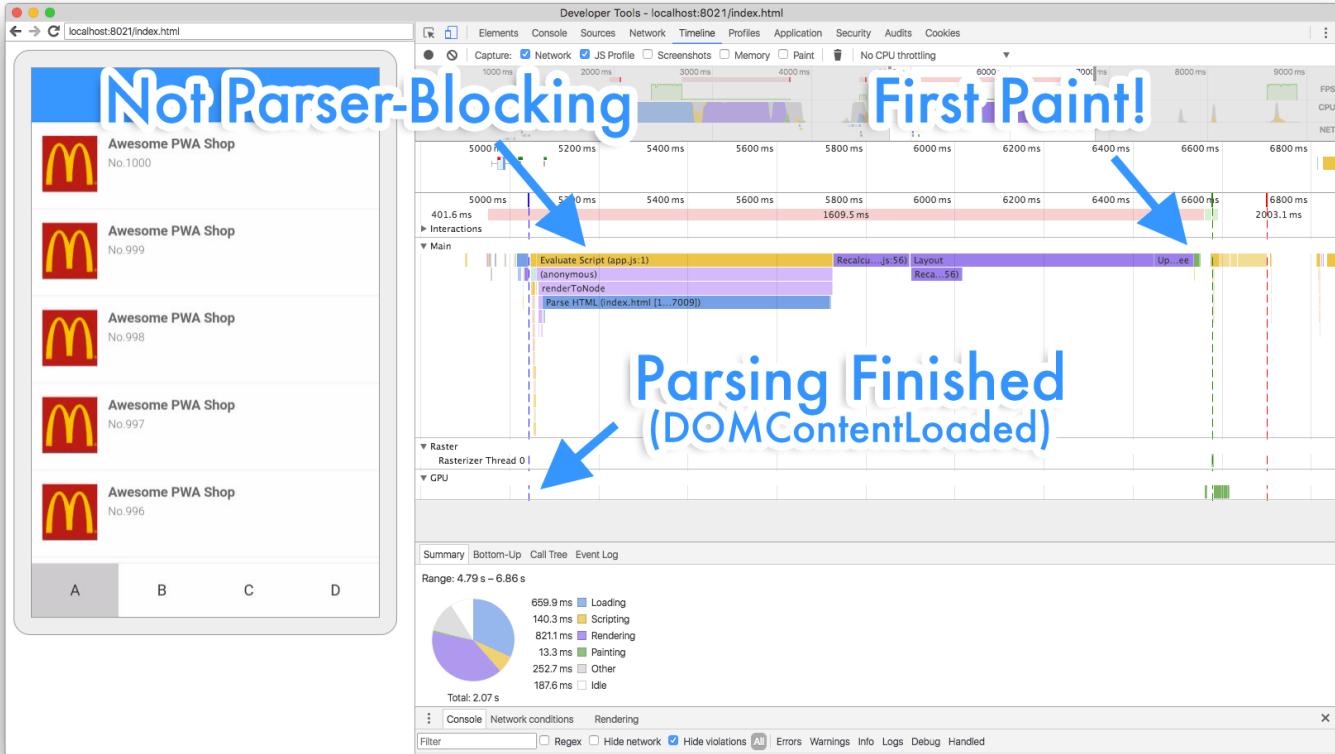
关于浏览器的绘制 (Painting)

HTML 文件中有标签并不意味着这些标签就能立刻被绘制到屏幕上，你必须保证页面的关键渲染路径 (<https://developers.google.com/web/fundamentals/performance/critical-rendering-path/>) 是为此优化的。很多开发者相信将 `script` 标签放在 `body` 的底部就足以保证内容能在脚本执行之前被绘制，这对于能渲染不完整 DOM 树的浏览器（比如桌面浏览器常见的流式渲染）来说可能是成立的。但移动端的浏览器很可能因为考虑到较慢的硬件、电量消耗等因素并不这么做。不仅如此，即使你曾被告知设为 `async` 或 `defer` 的脚本就不会阻塞 HTML 解析了，但这可不意味着浏览器就一定会在执行它们之前进行渲染。



首先我想澄清的是，根据 HTML 规范 Scripting 章节 (<https://html.spec.whatwg.org/multipage/scripting.html>)，`async` 脚本是在其请求完成后立刻运行的，因此它本来就可能阻塞到解析。只有 `defer`（且非内联）与最新的 `type=module` 被指定为“一定不会阻塞解析”。（不过 `defer` 目前也有点小问题……我们稍后会再提到）

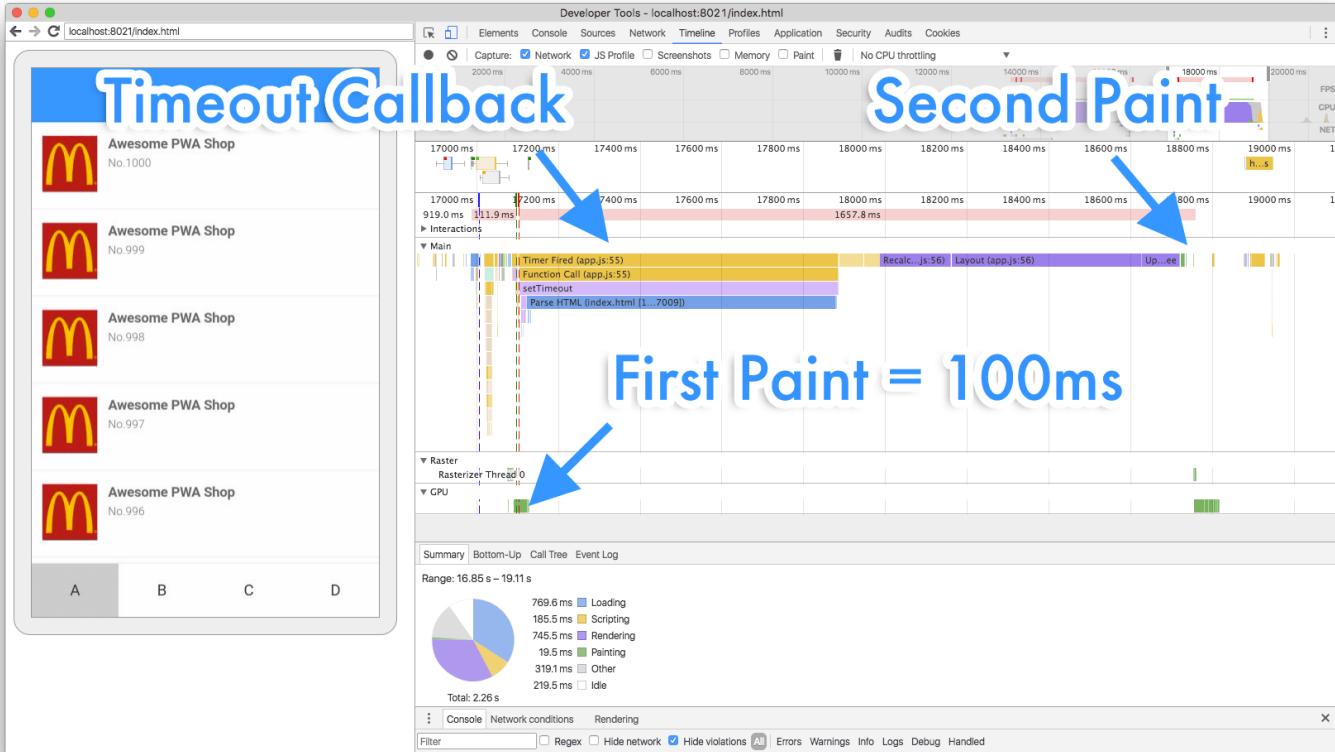
而更重要的是，一个不阻塞 HTML 解析的脚本仍然可能阻塞到绘制。我做了一个简化的“最小多页 PWA”（Minimal Multi-page PWA，或 MMPWA）来测试这个问题，：我们在一个 `async`（且确实不阻塞 HTML 解析）脚本中，生成并渲染 1000 个列表项，然后测试骨架屏能否在脚本执行之前渲染出来。下面是通过 USB Debugging 在我的 Nexus 5 真机上录制的 profile：



是的，出乎意料吗？首次渲染确实被阻塞到脚本执行结束后才发生。究其原因，**如果我们在浏览器还未完成上一次绘制工作之前就过快得进行了 DOM 操作，我们亲爱的浏览器就只好抛弃所有它已经完成的像素，且一直要等待到 DOM 操作引起的所有工作结束之后才能重新进行下一次渲染。**而这种情况更容易在拥有较慢 CPU/GPU 的移动设备上出现。

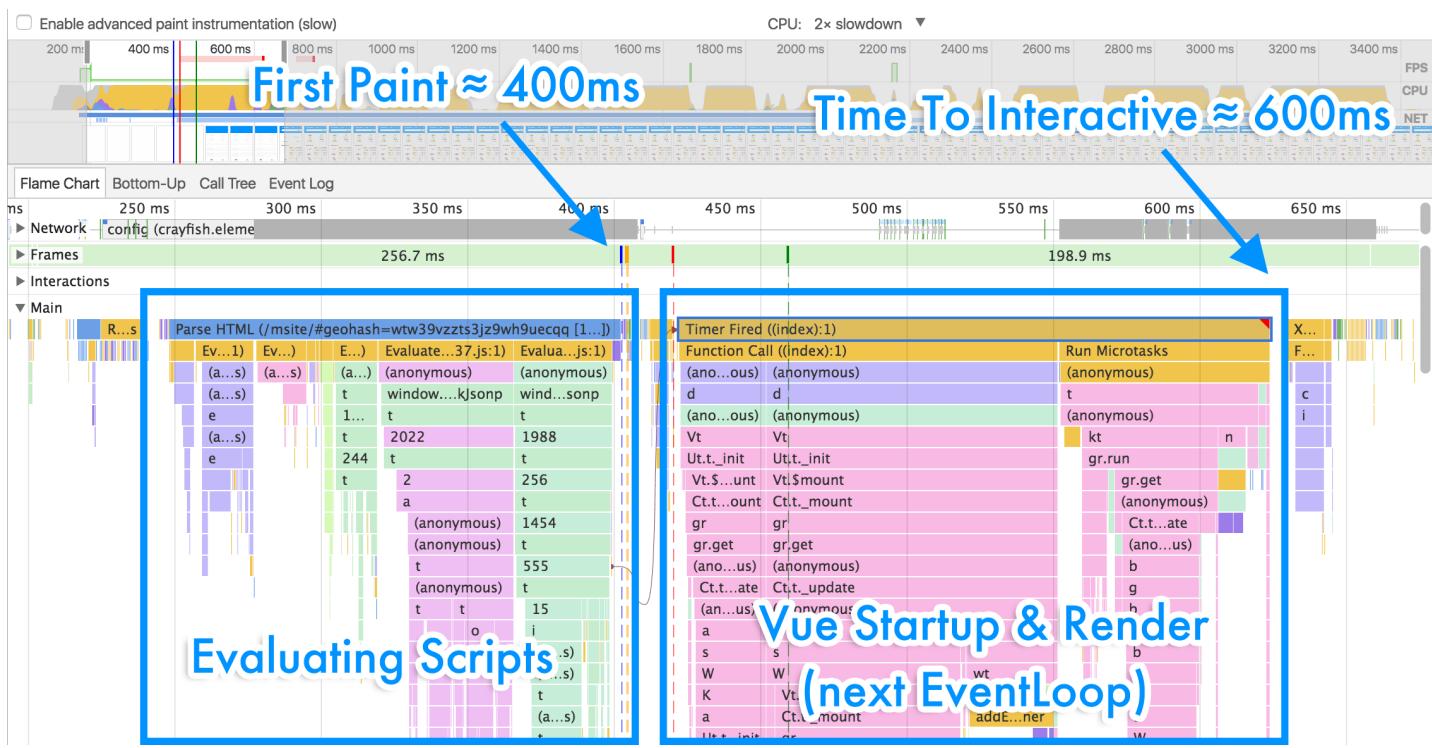
黑魔法：利用 setTimeout() 让绘制提前

不难发现，骨架屏的绘制与脚本执行实际是一个竞态。大概是 Vue 太快了，我们的骨架屏还是有非常大的概率绘制不出来。于是我们想着如何能让脚本执行慢点，或者说，“懒”点。于是我们想到了一个经典的 Hack：`setTimeout(callback, 0)`。我们试着把 MMPWA 中的 DOM 操作（渲染 1000 个列表）放进 `setTimeout(callback, 0)` 里.....



当当！首次渲染瞬间就被提前了。如果你熟悉浏览器的**事件循环模型（event loop）**的话，这招 Hack 其实是通过 `setTimeout` 的回调把 DOM 操作放到了事件循环的任务队列中以避免它在当前循环执行，这样浏览器就得以在主线程空闲时喘息一下（更新一下渲染）了。如果你想亲手试试 MMPWA 的话，你可以访问 github.com/Huxpro/mmpwa (<https://github.com/Huxpro/mmpwa>) 或 huangxuan.me/mmpwa/ (<https://huangxuan.me/mmpwa/>) 访问代码与 Demo。我把 UI 设计为了 A/B Test 的形式并改为渲染 5000 个列表项来让效果更夸张一些。

回到饿了么 PWA 上，我们同样试着把 `new Vue()` 放到了 `setTimeout` 中。果然，黑魔法再次显灵，骨架屏在每次跳转后都能立刻被渲染。这时的 Profile 看起来是这样的：



现在，我们在 400ms 时触发首次渲染（骨架屏），在 600ms 时完成真实 UI 的渲染并达到页面的可交互。你可以拉上去详细对比下优化前后 profile 的区别。

被我“defer”的有关 defer 的 Bug

不知道你发现没有，在上图的 Profile 中，我们仍然有不少脚本是阻塞了 HTML 解析的。好吧让我解释一下，由于历史原因，我们确实保留了一部分的阻塞脚本，比如侵入性很强的 lib-flexible (<https://github.com/amfe/lib-flexible>)，我们没法轻易去除它。不过，profile 里的大部分阻塞脚本实际上都设置了 defer，我们本以为他们应该在 HTML 解析完成之后才被执行，结果被 profile 打了一脸。

我和 Jake Archibald (<https://twitter.com/jaffathecake>) 聊了一下 (<https://twitter.com/Huxpro/status/859842124849827841>)，果然这是 Chrome 的 Bug： defer 的脚本被完全缓存时，并没有遵守规范等待解析结束，反而阻塞了解析与渲染。Jake 已经提交在 crbug (<https://bugs.chromium.org/p/chromium/issues/detail?id=717979>) 上了，一起给它投票吧~

最后，是优化后的 Lighthouse 跑分结果，同样可以看到明显的性能提升。需要说明的是，能影响 Lighthouse 跑分的因素有很多，所以我建议你以控制变量（跑分用的设备、跑分时的网络环境等）的方式来进行对照实验。

100 First meaningful paint: **999.5ms** (target: 1,600ms) [?](#)

100 Perceptual Speed Index: **435** (target: 1,250) [?](#)

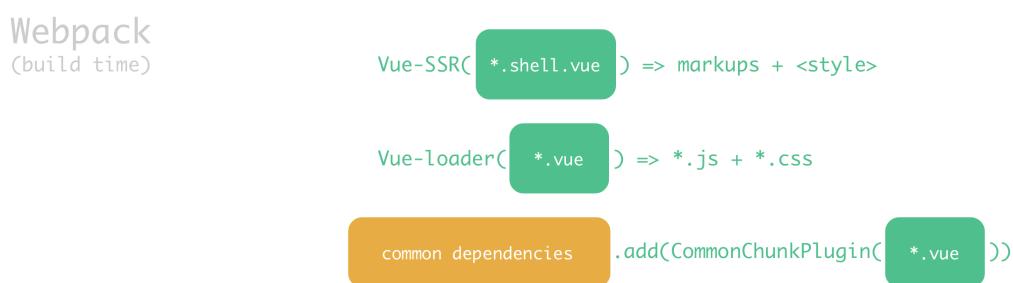
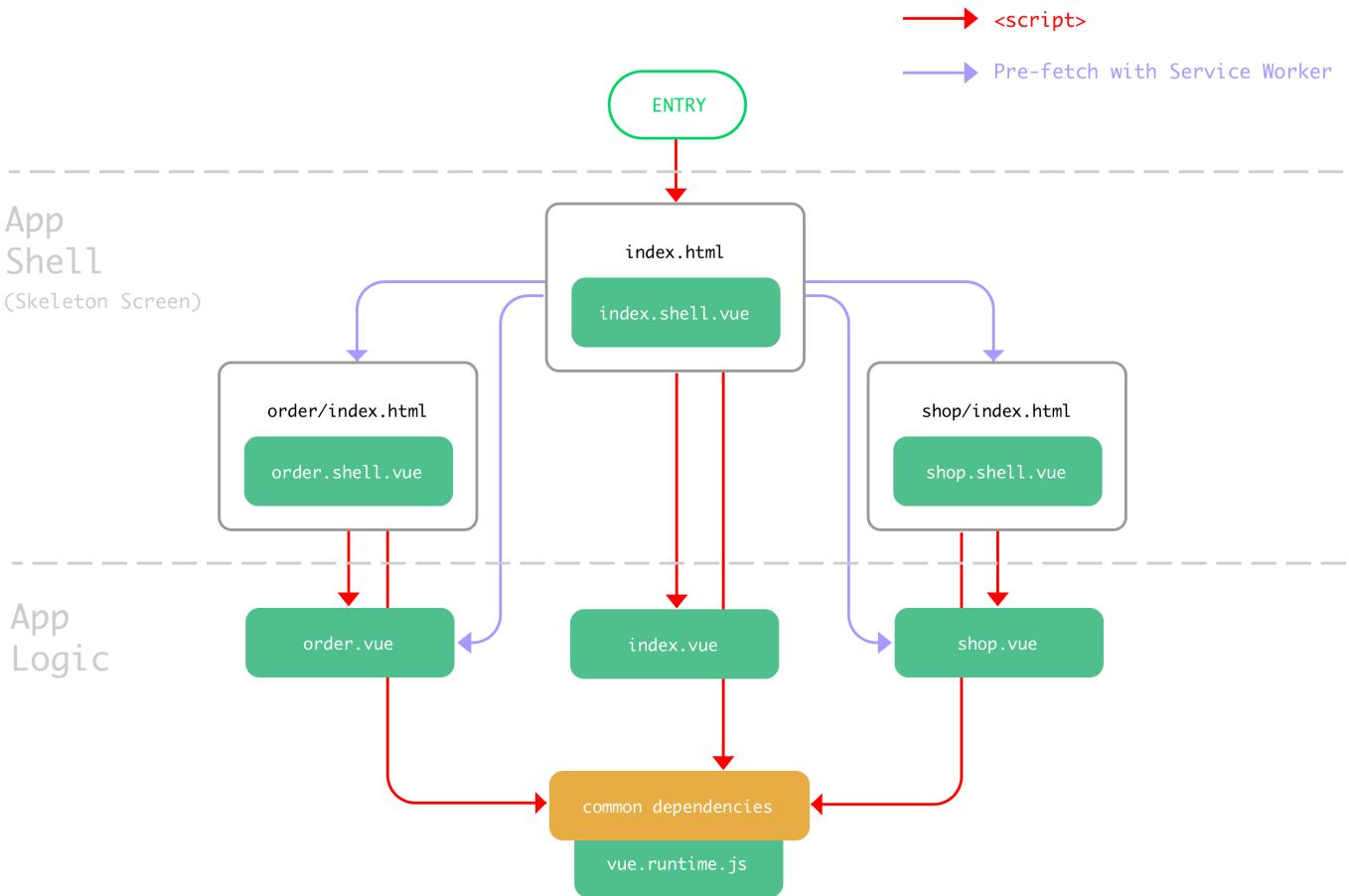
First Visual Change: **320ms**

Last Visual Change: **2686ms**

98 Estimated Input Latency: **41.4ms** (target: 50ms) [?](#)

99 Time To Interactive (alpha): **1272.1ms** (target: 5,000ms) [?](#)

最后附上一张图，这张图当时是做给 Addy Osmani 的 I/O 演讲用的，描述了饿了么 PWA 是如何结合 Vue 来实现多页应用的 PRPL 模式，可以作为一个架构的参考与示意图。



一些感想

多页应用仍然有很长的路要走

Web 是一个极其多样化的平台。从静态的博客，到电商网站，再到桌面级的生产力软件，它们全都是 Web 这个大家庭的第一公民。而我们组织 web 应用的方式，也同样只会更多而不会更少：多页、单页、Universal JavaScript 应用、WebGL、以及可以预见的 Web Assembly。不同的技术之间没有贵贱，但是适用场景的差距确是客观存在的。

Jake (<https://twitter.com/jaffathecake>) 曾在 Chrome Dev Summit 2016 (<https://youtu.be/J2dOTKBoTL4?list=PLNYkxOF6rcIBTs2KPy1E6tlYaWoFcG3uj>) 上说过 “PWA != SPA”。可是尽管我们已经用上了一系列最新的技术（PRPL、Service Worker、App Shell……），我们仍然因为多页应用模型本身的缺陷有着难以逾越的一些障碍。多页应用在未来可能会有“bfcache API”、Navigation Transition 等新的规范以缩小跟 SPA 的距离，不过我们也必须承认，时至今日，多页应用的局限性也是非常明显的。

而 PWA 终将带领 web 应用进入新的时代

即使我们的多页应用在升级 PWA 的路上不如单页的那些来得那么闪亮，但是 PWA 背后的想法与技术却实实在在的帮助我们在 web 平台上提供了更好的用户体验。

PWA 作为下一代 Web 应用模型 (<https://zhuanlan.zhihu.com/p/25167289>)，其尝试解决的是 web 平台本身的根本性问题：对网络与浏览器 UI 的硬依赖。因此，任何 web 应用都可以从中获益，这与你是多页还是单页、面向桌面还是移动端、是用 React 还是 Vue 无关。或许，它还终将改变用户对移动 web 的期待。现如今，谁还觉得桌面端的 web 只是个看文档的地方呢？

还是那句老话：让我们的用户，也像我们这般热爱 web 吧。

最后，感谢饿了么的王亦斯、任光辉、题叶，Google 的 Michael Yeung、DevRel 团队，UC 浏览器团队，腾讯 X5 浏览器团队在这次项目中的合作。感谢尤雨溪、陈蒙迪和 Jake Archibald 在写作过程中给予我的帮助。

PREVIOUS

HOW DOES SW-PRECACHE WORKS?
(</2017/05/28/SW-PRECACHE/>)

NEXT

FAREWELL, FLASH. 感谢你，但这一次是真正的永别。
(</2017/07/26/FAREWELL-FLASH/>)

 推荐 5 推文 分享

评分最高



加入讨论...

通过以下方式登录

或注册一个 DISQUS 帐号 

姓名

**Qiang Xie** · 2年前

为什么标题前面有一个#号呢

 2 ^ |  · 回复 · 分享,**Jesse** → **Qiang Xie** · 2年前

Markdown Atx

 |  · 回复 · 分享,**Qiang Xie** → **Jesse** · 2年前

我知道在写Markdown的时候使用#插入标题，但是为什么网页显示的时候前面会显示个#，如何把它去掉呢 求告知 谢谢

 |  · 回复 · 分享,**Richard Zg** → **Qiang Xie** · 2年前

post.html 往下拉的 anchors.options 里的 icon 定义成你要的形状或者去掉这块就好了

 |  · 回复 · 分享,**mengzhenwei1993** → **Richard Zg** · 2年前

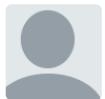
我config文件sidebar-avatar配置改了为什么还显示不出来头像啊(img没有高度) 朋友 - -!

 |  · 回复 · 分享,**Qiang Xie** → **Richard Zg** · 2年前

解决了，非常感谢！

 |  · 回复 · 分享,**Fabrizio Coltellaro** · 4个月前

支持一下

 |  · 回复 · 分享,**Le Wang** · 5个月前

大佬，看了这篇明白了自己还有很长的路要走

 |  · 回复 · 分享,



rpl · 6个月前

666

^ | v · 回复 · 分享 ·



绊运猫舍 · 8个月前

大神厉害，喜欢猫咪的可以访问<http://www.banyunmao.com/>

^ | v · 回复 · 分享 ·



smrtyan · 10个月前

11

^ | v · 回复 · 分享 ·



杜万江 · 1年前

感谢大神，你做的网站非常好，我也照着你的模板搭建了一个blog

^ | v · 回复 · 分享 ·



arvin · 1年前

可以，很强

^ | v · 回复 · 分享 ·



ABU LIU · 1年前

,,,

^ | v · 回复 · 分享 ·



shane · 1年前

thanks

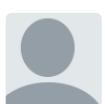
^ | v · 回复 · 分享 ·



Huabing Zhao · 2年前

博主还在维护这个博客吗？怎么这么久都没更新了？

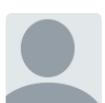
^ | v · 回复 · 分享 ·



test · 2年前

test

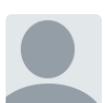
^ | v · 回复 · 分享 ·



Zhi Wang · 2年前

试一试

^ | v 1 · 回复 · 分享 ·



ZZtracy · 2年前

111

^ | v 2 · 回复 · 分享 ·

在 HUX BLOG 上还有

ES5, ES6, ES2016, ES.Next: JavaScript 的版本是怎么回事? 「译」

8条评论 • 4年前



Weijing Lin — Strawman 翻译成草众更贴切

hUX 随想录 (一) : Digital native 数字原住民

1条评论 • 2年前



Zespri — 好有趣! 我的孩子快出生. 父母在說絕對不讓小孩子使用手機, 眼睛會不好. 這篇讓我確信要讓小孩子很早就開始接觸電子設備. 像是

「知乎」如何评价 MIUI 6? - 黄玄的博客 | Hux Blog

2条评论 • 2年前



CPUDream — 谢谢博主的主题

Unix/Linux 扫盲笔记 - 黄玄的博客 | Hux Blog

1条评论 • 2年前



Huiyi.FYJ — 最狭义的 Unix, 稍广义的 Unix, 最广义的 Unix.
大多数时候 Unix 都是指稍广义的。



订阅



在您的网站上使用 Disqus 添加 Disqus



Disqus 隐私政策

隐私政策

FEATURED TAGS (/archive/)

知乎 (/archive/?tag=%E7%9F%A5%E4%B9%8E)

笔记 (/archive/?tag=%E7%AC%94%E8%AE%BD)

Coq (/archive/?tag=Coq)

SF (软件基础) (/archive/?tag=SF+%28%E8%BD%AF%E4%BB%B6%E5%9F%BA%E7%A1%80%29)

PLF (编程语言基础) (/archive/?tag=PLF+%28%E7%BC%96%E7%A8%8B%E8%AF%AD%E8%A8%80%E5%9F%BA%E7%A1%80%29)

Web (/archive/?tag=Web)

LF (逻辑基础) (/archive/?tag=LF+%28%E9%80%BB%E8%BE%91%E5%9F%BA%E7%A1%80%29)

UX/UI (/archive/?tag=UX%2FUI)

产品 (/archive/?tag=%E4%BA%A7%E5%93%81)

PWA (/archive/?tag=PWA)

JavaScript (/archive/?tag=JavaScript)

Slides (/archive/?tag=Slides)

生活 (/archive/?tag=%E7%94%9F%E6%B4%BB)

译 (/archive/?tag=%E8%AF%91)

阿里 (/archive/?tag=%E9%98%BF%E9%87%8C)

英国 (/archive/?tag=%F0%9F%87%AC%F0%9F%87%A7)

Vim (/archive/?tag=Vim)

计算机科学 (/archive/?tag=%E8%AE%A1%E7%AE%97%E6%9C%BA%E7%A7%91%E5%AD%A6)

计算理论 (/archive/?tag=%E8%AE%A1%E7%AE%97%E7%90%86%E8%AE%BA)

CSS (/archive/?tag=CSS)

Wechat (/archive/?tag=Wechat)

hUX 随想录 (/archive/?tag=hUX+%E9%9A%8F%E6%83%B3%E5%BD%95)

(/archive/?tag=hUX+%E9%9A%8F%E6%83%B3%E5%BD%95)

FRIENDS

(/archive/?tag=hUX+%E9%9A%8F%E6%83%B3%E5%BD%95)

(/archive/?tag=hUX+%E9%9A%8F%E6%83%B3%E5%BD%95)

(/archive/?tag=hUX+%E9%9A%8F%E6%83%B3%E5%BD%95) 乱序 (<http://mida.re/>)

Sherry Wu (<https://xuechundesign.github.io>) Luke 的自留地 (<https://hmqk1995.github.io>)
Ebn's Blog (<http://ebnbin.com/>) SmdCn's Blog (<https://blog.smdcn.net>) JiyinYiyong (<http://tiye.me/>)
David's Game (<https://www.ruoyaowu.com/>) DHong Say (<http://dhong.co>) 尹峰以为 (<https://ingf.github.io/>)



Copyright © Hux Blog 2019
Powered by Hux Blog (<https://huangxuan.me>) | [Star](#)