构建单页Web应用

摘自前端农民工的博客

让我们先来看几个网站：

coding

teambition

cloud9

注意这几个网站的相同点，那就是在浏览器中，做了原先“应当”在客户端做的事情。它们的界面切换非常流畅，响应很迅速，跟传统的网页明显不一样，它们是什么呢？这就是单页Web应用。

所谓单页应用，指的是在一个页面上集成多种功能，甚至整个系统就只有一个页面，所有的业务功能都是它的子模块，通过特定的方式挂接到主界面上。它是AJAX技术的进一步升华，把AJAX的无刷新机制发挥到极致，因此能造就与桌面程序媲美的流畅用户体验。

其实单页应用我们并不陌生，很多人写过ExtJS的项目，用它实现的系统，很天然的就已经是单页的了，也有人用jQuery或者其他框架实现过类似的东西。用各种JS框架，甚至不用框架，都是可以实现单页应用的，它只是一种理念。有些框架适用于开发这种系统，如果使用它们，可以得到很多便利。

开发框架

ExtJS可以称为第一代单页应用框架的典型，它封装了各种UI组件，用户主要使用JavaScript来完成整个前端部分，甚至包括布局。随着功能逐渐增加，ExtJS的体积也逐渐增大，即使用于内部系统的开发，有时候也显得笨重了，更不用说开发以上这类运行在互联网上的系统。

jQuery由于偏重DOM操作，它的插件体系又比较松散，所以比ExtJS这个体系更适合开发在公网运行的单页系统，整个解决方案会相对比较轻量、灵活。

但由于jQuery主要面向上层操作，它对代码的组织是缺乏约束的。如何在代码急剧膨胀的情况下控制每个模块的内聚性，并且适当在模块之间产生数据传递与共享，就成为了一种有挑战的事情。

为了解决单页应用规模增大时候的代码逻辑问题，出现了不少MV*框架，他们的基本思路都是在JS层创建模块分层和通信机制。有的是MVC，有的是MVP，有的是MVVM，而且，它们几乎都在这些模式上产生了变异，以适应前端开发的特点。

这类框架包括Backbone，Knockout，AngularJS，Avalon等。

组件化

这些在前端做分层的框架推动了代码的组件化，所谓组件化，在传统的Web产品中，更多的指UI组件，但其实组件是一个广泛概念，传统Web产品中UI组件占比高的原因是它的厚度不足，随着客户端代码比例的增加，相当一部分的业务逻辑也前端化，由此催生了很多非界面型组件的出现。

分层带来的一个优势是，每层的职责更专一了，由此，可以对其作单元测试的覆盖，以保证其质量。传统UI层测试最头疼的问题是UI层和逻辑混杂在一起，比如往往会在远程请求的回调中更改DOM，当引入分层之后，这些东西都可以分别被测试，然后再通过场景测试来保证整体流程。

代码隔离

与开发传统页面型网站相比，实现单页应用的过程中，有一些比较值得特别关注的点。

从单页应用的特点来看，它比页面型网站更加依赖于JavaScript，而由于页面的单页化，各种子功能的JavaScript代码聚集到了同一个作用域，所以代码的隔离、模块化变得很重要。

在单页应用中，页面模板的使用是很普遍的。很多框架内置了特定的模板，也有的框架需要引入第三方的模板。这种模板是界面片段，我们可以把它们类比成JavaScript模块，它们是另一种类型的组件。

模板也一样有隔离的需要。不隔离模板，会造成什么问题呢？模板间的冲突主要存在于id属性上，如果一个模板中包含固定的id，当它被批量渲染的时候，会造成同一个页面的作用域中出现多个相同id的元素，产生不可预测的后果。因此，我们需要在模板中避免使用id，如果有对DOM的访问需求，应当通过其他选择器来完成。如果一个单页应用的组件化程度非常高，很可能整个应用中都没有元素id的使用。

代码合并与加载策略

人们对于单页系统的加载时间容忍度与Web页面不同，如果说他们愿意为购物页面的加载等待3秒，有可能会愿意为单页应用的首次加载等待5-10秒，但在此之后，各种功能的使用应当都比较流畅，所有子功能页面尽量要在1-2秒时间内切换成功，否则他们就会感觉这个系统很慢。

从这些特点来看，我们可以把更多的公共功能放到首次加载，以减小每次加载的载入量，有一些站点甚至把所有的界面和逻辑全部放到首页加载，每次业务界面切换的时候，只产生数据请求，因此它的响应是非常迅速的，比如青云的控制台就是这么做的。

通常在单页应用中，无需像网站型产品一样，为了防止文件加载阻塞渲染，把js放到html后面加载，因为它的界面基本都是动态生成的。

当切换功能的时候，除了产生数据请求，还需要渲染界面，这个新渲染的界面部件一般是界面模板，它从哪里来呢？来源无非是两种，一种是即时请求，像请求数据那样通过AJAX获取过来，另一种是内置于主界面的某些位置，比如script标签或者不可见的textarea中，后者在切换功能的时候速度有优势，但是加重了主页面的负担。

在传统的页面型网站中，页面之间是互相隔离的，因此，如果在页面间存在可复用的代码，一般是提取成单独的文件，并且可能会需要按照每个页面的需求去进行合并。单页应用中，如果总的代码量不大，可以整体打包一次在首页载入，如果大到一定规模，再作运行时加载，加载的粒度可以搞得比较大，不同的块之间没有重复部分。

路由与状态的管理

我们最开始看到的几个在线应用，有的是对路由作了管理的，有的没有。

管理路由的目的是什么呢？是为了能减少用户的导航成本。比如说我们有一个功能，经历过多次导航菜单的点击，才呈现出来。如果用户想要把这个功能地址分享给别人，他怎么才能做到呢？

传统的页面型产品是不存在这个问题的，因为它就是以页面为单位的，也有的时候，服务端路由处理了这一切。但是在单页应用中，这成为了问题，因为我们只有一个页面，界面上的各种功能区块是动态生成的。所以我们要通过对路由的管理，来实现这样的功能。

具体的做法就是把产品功能划分为若干状态，每个状态映射到相应的路由，然后通过pushState这样的机制，动态解析路由，使之与功能界面匹配。

有了路由之后，我们的单页面产品就可以前进后退，就像是在不同页面之间一样。

其实在Web产品之外，早就有了管理路由的技术方案，Adobe Flex中，就会把比如TabNavigator，甚至下拉框的选中状态对应到url上，因为它也是单“页面”的产品模式，需要面对同样的问题。

当产品状态复杂到一定程度的时候，路由又变得很难应用了，因为状态的管理极其麻烦，比如开始的时候我们演示的c9.io在线IDE，它就没法把状态对应到url上。

缓存与本地存储

在单页应用的运作机制中，缓存是一个很重要的环节。

由于这类系统的前端部分几乎全是静态文件，所以它能够有机会利用浏览器的缓存机制，而比如动态加载的界面模板，也完全可以做一些自定义的缓存机制，在非首次的请求中直接取缓存的版本，以加快加载速度。

甚至，也出现了一些方案，在动态加载JavaScript代码的同时，把它们也缓存起来。比如Addy Osmani的这个basket.js，就利用了HTML5 localStorage作了js和css文件的缓存。

在单页产品中，业务代码也常常会需要跟本地存储打交道，存储一些临时数据，可以使用localStorage或者localStorageDB来简化自己的业务代码。

服务端通信

传统的Web产品通常使用JSONP或者AJAX这样的方式与服务端通信，但在单页Web应用中，有很大一部分采用WebSocket这样的实时通讯方式。

WebSocket与传统基于HTTP的通信机制相比，有很大的优势。它可以让服务端很便利地使用反向推送，前端只响应确实产生业务数据的事件，减少一遍又一遍无意义的AJAX轮询。

由于WebSocket只在比较先进的浏览器上被支持，有一些库提供了在不同浏览器中的兼容方案，比如socket.io，它在不支持WebSocket的浏览器上会降级成使用AJAX或JSONP等方式，对业务代码完全透明、兼容。

内存管理

传统的Web页面一般是不需要考虑内存的管理的，因为用户的停留时间相对少，即使出现内存泄漏，可能很快就被刷新页面之类的操作冲掉了，但单页应用是不同的，它的用户很可能会把它开一整天，因此，我们需要对其中的DOM操作、网络连接等部分格外小心。

样式的规划

在单页应用中，因为页面的集成度高，所有页面聚集到同一作用域，样式的规划也变得重要了。

样式规划主要是几个方面：

基准样式的分离

这里面主要包括浏览器样式的重设、全局字体的设置、布局的基本约定和响应式支持。

组件样式的划分

这里面是两个层面的规划，首先是各种界面组件及其子元素的样式，其次是一些修饰样式。组件样式应当尽量减少互相依赖，各组件的样式允许冗余。

堆叠次序的管理

传统Web页面的特点是元素多，但是层次少，单页应用会有些不同。

在单页应用中，需要提前为各种UI组件规划堆叠次序，也就是z-index，比如说，我们可能会有各种弹出对话框，浮动层，它们可能组合成各种堆叠状态。新的对话框的z-index需要比旧的高，才能确保盖在它上面。诸如此类，都需要我们对这些可能的遮盖作规划，那么，怎样去规划呢？

了解通信知识的人，应当会知道，不同的频率段被划分给不同的通信方式使用，在一些国家，领空的使用也是有划分的，我们也可以用同样的方式来预先分段，不同类型的组件的z-index落到各自的区间，以避免它们的冲突。

单页应用的产品形态

我们在开始的时候提到，存在着很多新型Web产品，使用单页应用的方式构建，但实际上，这类产品不仅仅存在于Web上。点开Chrome商店，我们会发现很多离线应用，这些产品都可以算是单页应用的体现。

除了各种浏览器插件，借助node-webkit这样的外壳平台，我们可以使用Web技术来构建本地应用，产品的主要部分仍然是我们熟悉的单页应用。

单页应用的流行程度正在逐渐增加，大家如果关注了一些初创型互联网企业，会发现其中很大一部分的产品模式是单页化的。这种模式能带给用户流畅的体验，在开发阶段，对JavaScript技能水平要求较高。

单页应用开发过程中，前后端是天然分离的，双方以API为分界。前端作为服务的消费者，后端作为服务的提供者。在此模式下，前端将会推动后端的服务化。当后端不再承担模板渲染、输出页面这样工作的情况下，它可以更专注于所提供的API的实现，而在这样的情况下，Web前端与各种移动终端的地位对等，也逐渐使得后端API不必再为每个端作差异化设计了。

部署模式的改变

在现在这个时代，我们已经可以看到一种产品的出现了，那就是“无后端”的Web应用。这是一种什么东西呢？基于这种理念，你的产品很可能只需要自己编写静态Web页面，在某种BaaS（Backend as a Service）云平台上定制服务端API和云存储，集成这个平台提供的SDK，通过AJAX等方式与之打交道，实现注册认证、社交、消息推送、实时通信、云存储等功能。

我们观察一下这种模式，会发现前后端的部署已经完全分离了，前端代码完全静态化，这意味着可以把它们放置到CDN上，访问将大大地加速，而服务端托管在BaaS云上，开发者也不必去关注一些部署方面的繁琐细节。

假设你是一名创业者，正在做的是一种实时协同的单页产品，可以在云平台上，快速定制后端服务，把绝大部分宝贵的时间花在开发产品本身上。

单页应用的缺陷

单页应用最根本的缺陷就是不利于SEO，因为界面的绝大部分都是动态生成的，所以搜索引擎很不容易索引它。

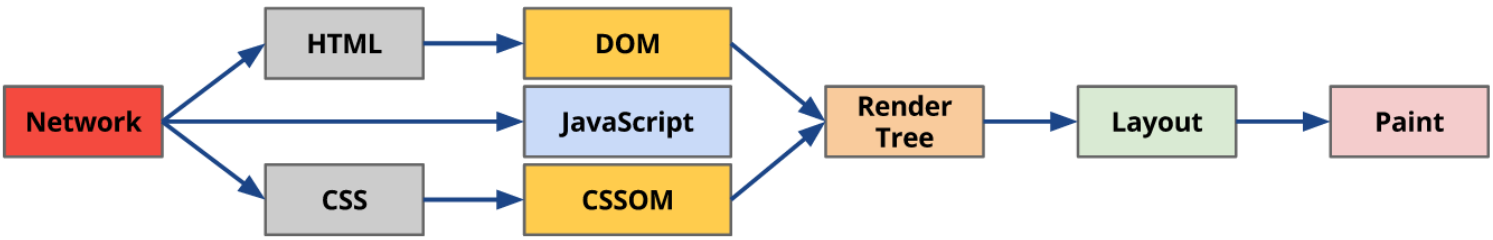
产品单页化带来的挑战

一个产品想要单页化，首先是它必须适合单页的形态。其次，在这个过程中，对开发模式会产生一些变更，对开发技能也会有一些要求。

开发者的JavaScript技能必须过关，同时需要对组件化、设计模式有所认识，他所面对的不再是一个简单的页面，而是一个运行在浏览器环境中的桌面软件。

用JS渲染的单页面应用其实性能还是比较差的

证明这个结论之前，要先阐述一下浏览器的渲染机制，这里先祭出这篇文章：《关键呈现路径》，文章主要介绍了浏览器渲染过程，其实大家也大概都了解过：

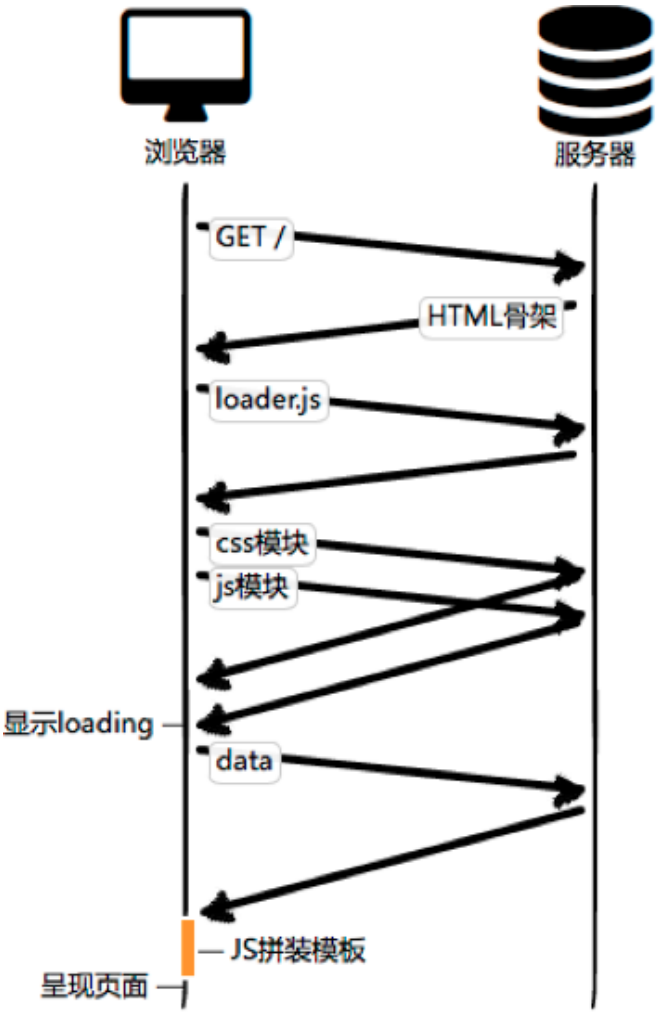


如上图，浏览器通过网络请求加载页面资源，在页面呈现之前无论如何都要经历以下过程：

1. HTML→DOM
2. CSS→CSSOM
3. DOM + CSSOM → Render Tree

4. 对Render Tree进行布局计算(Layout)
5. 对布局结果进行屏幕绘制(Paint)

如果在JS渲染页面模式下，需要在前端用JS加载样式并组装数据生成HTML插入页面，以上浏览器渲染过程必须等到页面加载完CSS，并且JS加载完数据拼装好HTML之后才能开始进行，一般的网络时序如下：



前端渲染模式

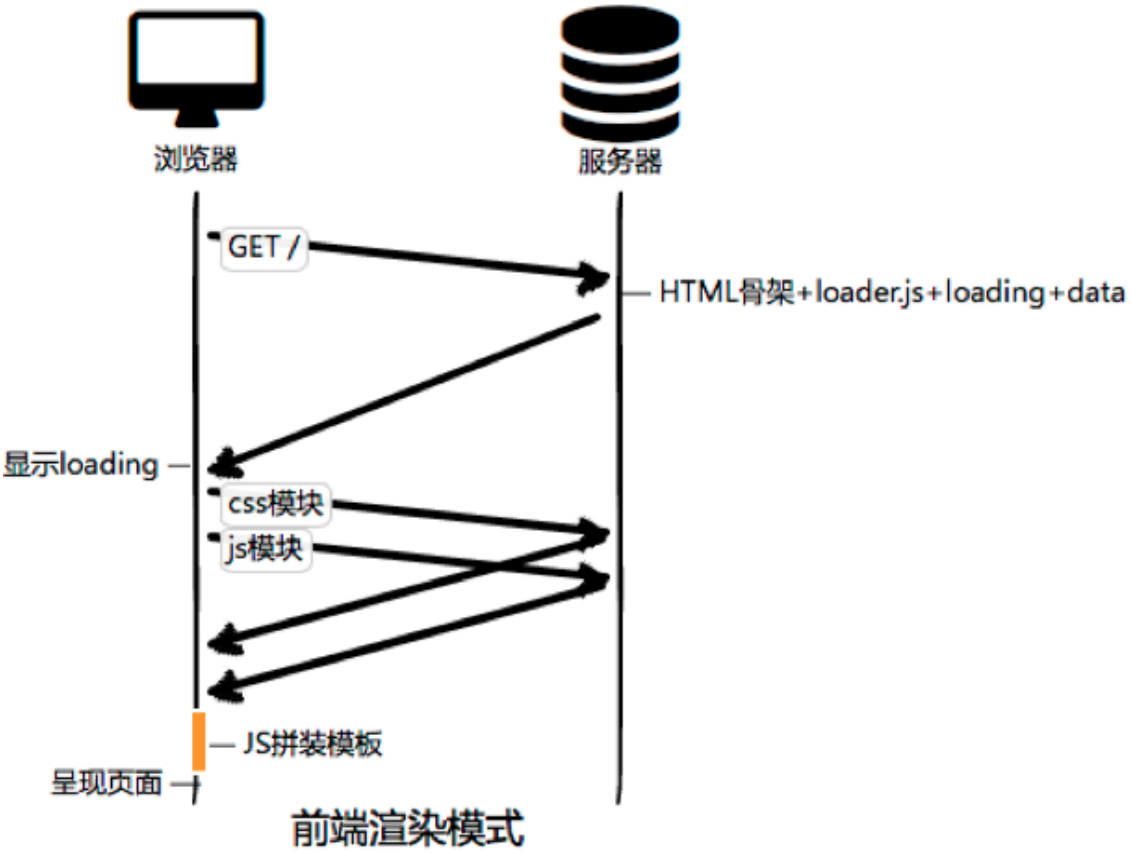
大概阐述一下这个流程：

1. 浏览器发起请求加载主文档
2. 服务端响应一个基本骨架的主文档
3. 浏览器加载主文档中外链的loader.js（根据路由控制资源加载的）
4. 服务端响应loader.js
5. loader.js执行，根据页面url判断用户访问到哪个虚拟页面，然后再发起请求加载对应页面的js和css
6. 页面所需JS和CSS都加载完毕，JS执行，发起请求加载数据
7. 数据加载完毕，JS执行前端模板拼装，插入DOM节点，然后浏览器开始前述渲染过程
8. 最终页面呈现

概括一下，加载时序大概是这样的：

以上加载过程均为串行，需要至少多付出3次RTT。如果把这种架构应用在高延迟的网络环境下（比如移动2G），那就是找死啊（其实国内现在的网络环境很好了，这样搞问题或许不太明显）。

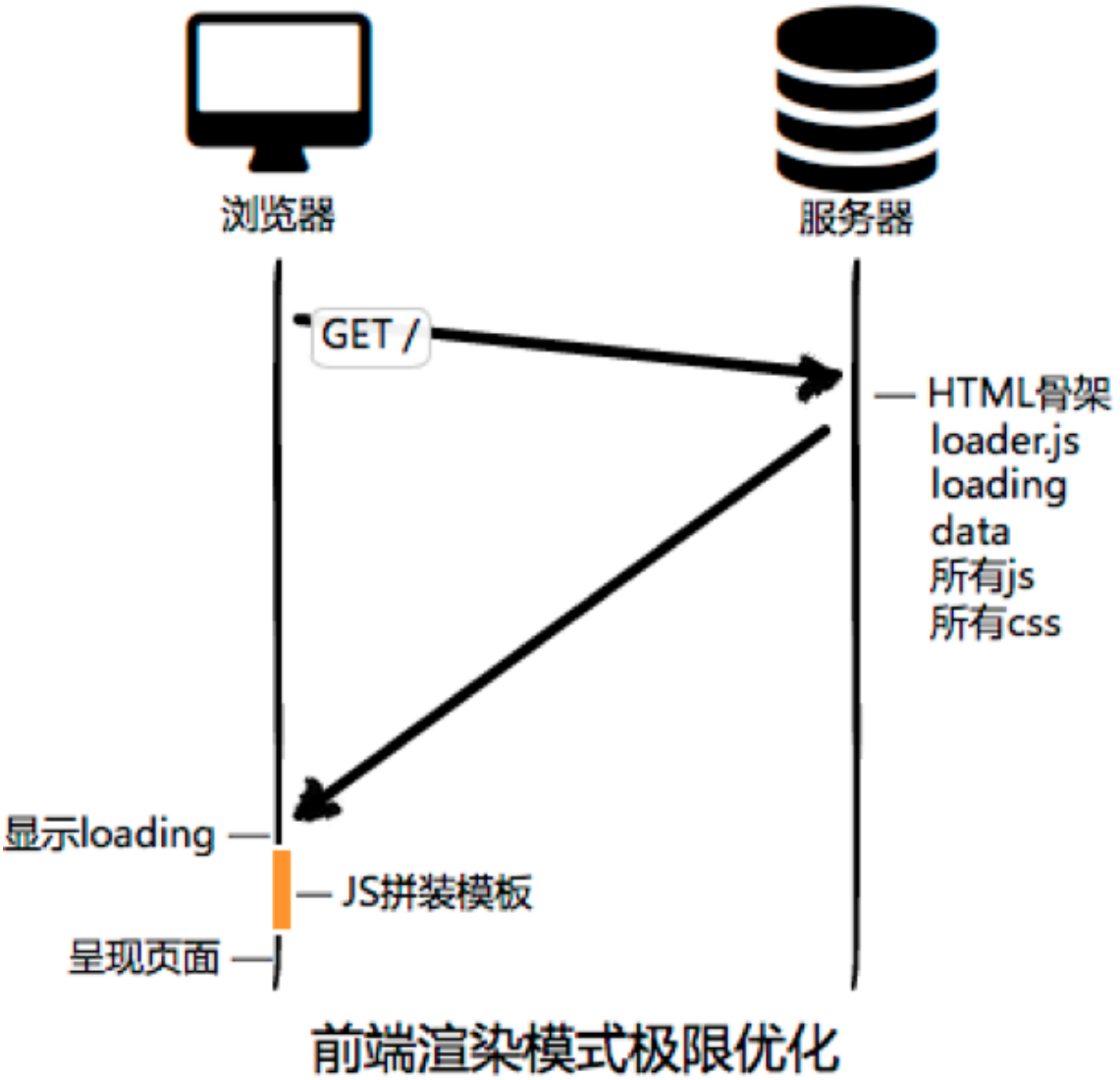
当然，上面的例子还是常规了一些，有些请求可以适当合并，进一步优化之后，大概可以搞成这个样子：



前端渲染模式

就是首次请求的主文档尽量多内嵌一些东西，除了HTML骨架之外，把loader.js内嵌，再加一个loading界面，让用户觉得没那么长时间白屏，另外如果前端路由切换是pusState控制的话，可以在服务端知道前端路由url，然后在主文档中直接内嵌数据，主文档体积大了不少，但是可以减少2次RTT，优化对比：

当然，如果你的单页面应用体量很小，完全不用按需加载，主文档内嵌一切可以再减少一次RTT，得到：



前端渲染模式极限优化

不过这么极端的做法其限制就是：你的应用千万不能太大！

前端渲染模式我厂的代表产品：UC奇趣百科 ，其优化点：
* 主文档loader.js内嵌、数据内嵌、loading界面内嵌
* 页面资源按需加载，请求动态合并
* localstorage存储JS/CSS

在国内的网络环境下感觉还OK吧。。。

兼顾性能、兼顾SEO，还是单页面应用，是可以做到的！

很明显，前端JS渲染由于违背了浏览器的优化策略，总是存在一个不可突破的瓶颈：

> JS和数据没加载完，JS拼装数据的逻辑没执行完，浏览器不能开始正常的渲染流程。

这个性能差异我感觉短时间内这种JS渲染的webapp是无法跟传统页面输出模式相比较的，因为浏览器的各种渲染优化策略基本上都是围绕着传统页面时序展开的。有没有办法突破这个性能瓶颈，并且兼顾SEO，但还保留单页面应用的体验呢？

答案是：有办法。

有人或许会想到 Isomorphic Javascript，所谓的同构JavaScript，或者什么前后端模板复用，相信我，这个概念根本就是扯淡！

其实办法很简单，根本用不着同构JS，页面还是服务端拼装好的，CSS在head中，主文档是完整的HTML，JS在body尾部；但需要在后端模板中实现一种功能：允许通过特殊的ajax请求以json格式响应页面中的局部区域。这项技术被称为 Quickling。

此外，单页面应用还有一项优化手段，叫PageCache，前端控制页面切换时，把之前的页面缓存到内存中，下次再回到这个页面就直接展现，不用再次请求数据拼装模板渲染，进一步优化用户在站内浏览的体验。

基于Quickling和PageCache我们在印度市场（网络环境超差）实现了两个单页面应用产品：YoloSong 和 Huntnews ，其优化点：

- 首次访问服务端渲染，页面间Quickling切换，单页面体验
- 所有链接可爬取，解决SEO问题
- PageCache缓存已访问页面，加速切换，历史记录前进后退
- 可 **全站禁用JS**，不影响浏览体验
- 按需加载，请求合并

分类： JavaScript