你真的了解前端工程化么

一、为什么需要前端工程化？

目前来说，Web 业务日益复杂化和多元化，前端开发已经由以 WebPage 模式为主转变为以 WebApp 模式为主了。现在随便找个前端项目，都已经不是过去的拼个页面+搞几个 jQuery 插件就能完成的了。如今的前端领域早已扩展到了服务端领域（Node.js），移动端领域（Hybrid 模式 & JS To Native 模式），桌面应用，各种浏览器的 Extension ，而 JS 引擎，除了 Google V8 之外还有 JavaScriptCore，微软的 ChakraCore 等等。代码量可能从以前的千行到如今的万行，甚至十万行。人数从一个人变成了 N 个一起协作开发，于是历史上的“我们”随着这些需求的增加，对 JS 的改造有了很多不同的方案，如：早期的 CoffeeScript 对 JS 的语法糖进行增强，AMD 的模块化（ require.js），打包工具（Grunt.js）等。业务上，我们越来越从中游的承上启下，变成了去承接从点到点业务的全面，我们会去思考这些复杂和多元的场景，而产生的问题，如：

- 如何扩展 javascript 、html、css 本身的语言能力
- 如何进行高效的多人协作
- 如何解决功能复用和变更问题
- 如何保证项目的规范性

二、如何解决以上问题

1. 如何扩展 javascript 、html、css 本身的语言能力？

对于 JavaScript 经历了近 10 年的成长，从 ECMAScript 3.到现在的 5.6.7.8。语言有了很大的成长。但是介于不同的平台对于新的语法，新的格式的支持不统一。我们需要一些工具来辅助这些语法使得他们可用。在网上有很多为了兼容性的整合工具进行操作。包括现代化构建工具，都有相应的包进行处理。对于 JavaScript 不得不说的就是 JavaScript 的超集：TypeScript。

typeScript 是 javascript 的超集，扩展了语法 （类 Classes，接口 interfaces，模块 Modules,类型注解 Type annotaions,编译时类型检查 Compile time type checking,Arrow 函数(类似 c#的 Lambda))使得 JavaScript 变得更强大，对于面向对象编程更好的支持。

CSS 预处理器：SASS LESS Stylus 。

它们基于 CSS 扩展了一套属于自己的 DSL，来解决我们书写 CSS 时难以解决的问题：

- 语法不够强大，比如无法嵌套书写导致模块化开发中需要书写很多重复的选择器；
- 没有变量和合理的样式复用机制，使得逻辑上相关的属性值必须以字面量的形式重复输出，导致难以维护。

所以这就决定了 CSS 预处理器的主要目标：提供 CSS 缺失的样式层复用机制、减少冗余代码，提高样式代码的可维护性。

2. 如何进行高效的多人协作

模块化

简单来说，模块化就是**将一个大文件拆分成相互依赖的小文件，再进行统一的拼装和加载**。只有这样，才有多人协作的可能。

JS 的模块化

在 ES6 之前，JavaScript 一直没有模块系统，这对开发大型复杂的前端工程造成了巨大的障碍。对此社区制定了一些模块加载方案，如 CommonJS、AMD 和 CMD 等，某些框架也会有自己模块系统，比如 Angular1.x。

现在 ES6 已经在语言层面上规定了模块系统，完全可以取代现有的 CommonJS 和 AMD 规范，而且使用起来相当简洁，并且有静态加载的特性。

规范确定了，然后就是模块的打包和加载问题：

1. 用 Webpack+Babel 将所有模块打包成一个文件同步加载，也可以打成多个 chunk 异步加载；

2. 用 SystemJS+Babel 主要是分模块异步加载；

3. 用浏览器的<script type="module">加载

目前 Webpack 远比 SystemJS 流行。Safari 已经支持用 type="module"加载了。

CSS 的模块化

虽然 SASS、LESS、Stylus 等预处理器实现了 CSS 的文件拆分，但没有解决 CSS 模块化的一个重要问题：选择器的全局污染问题。

按道理，一个模块化的文件应该要隐藏内部作用域，只暴露少量接口给使用者。而按照目前预处理器的方式，导入一个 CSS 模块后，已存在的样式有被覆盖的风险。虽然重写样式是 CSS 的一个优势，但这并不利于多人协作。

为了避免全局选择器的冲突，各厂都制定了自己的 CSS 命名风格：

- BEM 风格；
- Bootstrap 风格；
- Semantic UI 风格；
- 我们公司的 NEC 风格；
- ...

但这毕竟是弱约束。选择器随着项目的增长变得越多越复杂，然后项目组里再来个新人带入自己的风格，就更加混乱了。

从工具层面，社区又创造出 Shadow DOM、CSS in JS 和 CSS Modules 三种解决方案。

- Shadow DOM 是 WebComponents 的标准。它能解决全局污染问题，但目前很多浏览器不兼容，对我们来说还很久远；
- CSS in JS 是彻底抛弃 CSS，使用 JS 或 JSON 来写样式。这种方法很激进，不能利用现有的 CSS 技术，而且处理伪类等问题比较困难；
- CSS Modules 仍然使用 CSS，只是让 JS 来管理依赖。它能够最大化地结合 CSS 生态和 JS 模块化能力，目前来看是最好的解决方案。Vue 的 scoped style 也算是一种。

资源的模块化

Webpack 的强大之处不仅仅在于它统一了 JS 的各种模块系统，取代了 Browserify、RequireJS、SeaJS 的工作。更重要的是它的万能模块加载理念，即所有的资源都可以且也应该模块化。

资源模块化后，有三个好处：

8. 依赖关系单一化。所有 CSS 和图片等资源的依赖关系统一走 JS 路线，无需额外处理 CSS 预处理器的依赖关系，也不需处理代码迁移时的图片合并、字体图片等路径问题；
9. 资源处理集成化。现在可以用 loader 对各种资源做各种事情，比如复杂的 vue-loader 等等。
10. 项目结构清晰化。使用 Webpack 后，你的项目结构总可以表示成这样的函数： dest = webpack(src, config)

3. 如何解决功能复用和性能问题

为了解决复用问题，引入组件化的概念。何以提高代码的复用性。

组件化

首先，组件化≠模块化。好多人对这两个概念有些混淆。

模块化只是在文件层面上，对代码或资源的拆分；而组件化是在设计层面上，对UI（用户界面）的拆分。

从UI拆分下来的**每个包含模板(HTML)+样式(CSS)+逻辑(JS)功能完备的结构单元，我们称之为组件。**

其实，组件化更重要的是一种**分治思想**。

> Keep Simple. Everything can be a component.

这句话就是说页面上所有的东西都是组件。页面是个大型组件，可以拆成若干个中型组件，然后中型组件还可以再拆，拆成若干个小型组件，小型组件也可以再拆，直到拆成DOM元素为止。DOM元素可以看成是浏览器自身的组件，作为组件的基本单元。

传统前端框架/类库的思想是先组织DOM，然后把某些可复用的逻辑封装成组件来操作DOM，是DOM优先；而组件化框架/类库的思想是先来构思组件，然后用DOM这种基本单元结合相应逻辑来实现组件，是组件优先。这是两者本质的区别。

其次，组件化实际上是一种按照模板(HTML)+样式(CSS)+逻辑(JS)三位一体的形式**对面向对象的进一步抽象**。

所以我们除了封装组件本身，还要合理处理组件之间的关系，比如**（逻辑）继承**、**（样式）扩展**、**（模板）嵌套**和**包含**等，这些关系都可以归为**依赖**。

其实组件化不是什么新鲜的东西，以前的客户端框架，像WinForm、WPF、Android等，它们从诞生的那天起就是组件化的。而前端领域发展曲折，是从展示页面为主的WebPage模式走过来的，近两年才从客户端框架经验中引入了组件化思想。其实我们很多前端工程化的问题都可以从客户端那里寻求解决方案。

目前市面上的组件化框架很多，主要的有Vue、React、Angular 2。

4. 如何保证项目的规范性

模块化和组件化确定了开发模型，而这些东西的实现就需要规范去落实。

规范化其实是工程化中很重要的一个部分，项目初期规范制定的好坏会直接影响到后期的开发质量。

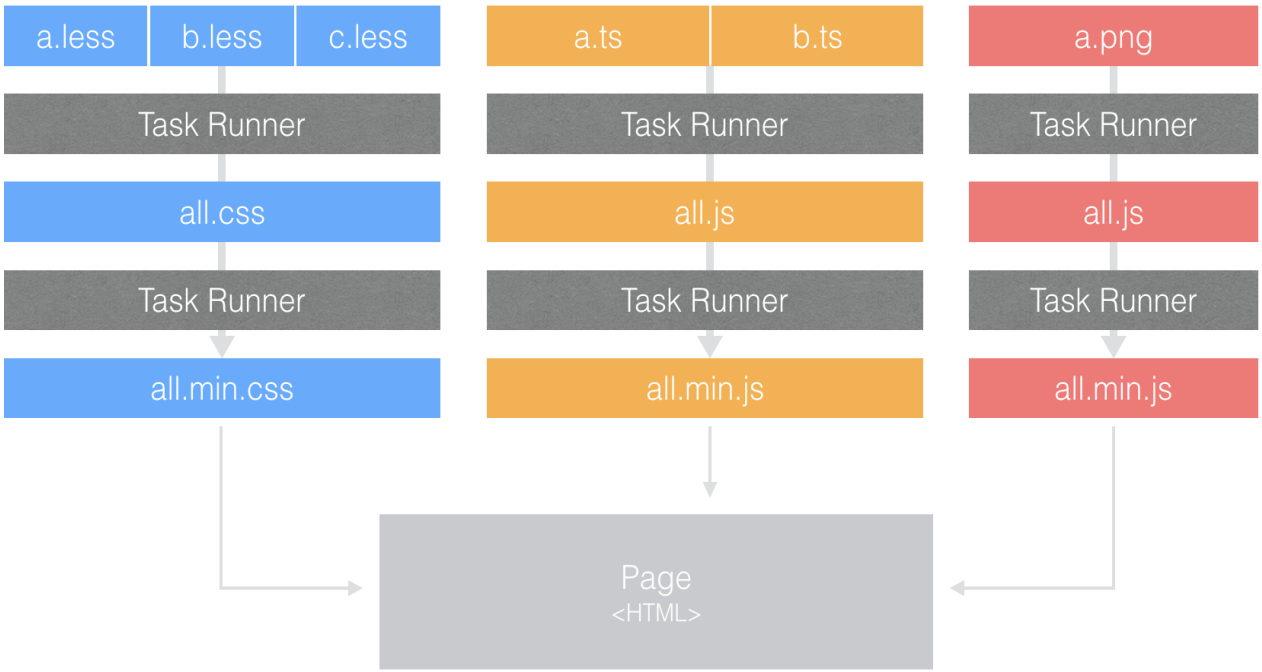
我能想到的有以下一些内容：

- 目录结构的制定
- 编码规范
- 前后端接口规范
- 文档规范
- 组件管理
- Git 分支管理
- Commit 描述规范
- 定期 CodeReview
- 视觉图标规范

三、现有工具

1、Grunt & Gulp

工作流：
这两款工具都是基于任务类型，所以它们的工作流是一致的：



可以看到它们打包的策略通常是 All in one，最后页面还是引用 css、img、js，开发流程与徒手开发相比并无差异。

特点与不足

Grunt

Grunt 是老牌的构建工具，特点是配置驱动，你需要做的就是了解各种插件的功能，然后把配置整合到 Gruntfile.js 中。

Grunt 缺点也是配置驱动，当任务非常多的情况下，试图用配置完成所有事简直就是个灾难；再就是它的 I/O 操作也是个弊病，它的每一次任务都需要从磁盘中读取文件，处理完后再写入到磁盘，例如：我想对多个 less 进行预编译、压缩操作，那么 Grunt 的操作就是：

读取 less 文件 -> 编译成 css -> 存储到磁盘 -> 读取 css -> 压缩处理 -> 存储到磁盘

这样一来当资源文件较多，任务较复杂的时候性能就是个问题了。

Gulp

Gulp 特点是代码驱动，写任务就和写普通的 Node.js 代码一样

再一个对文件读取是流式操作（Stream），也就是说一次 I/O 可以处理多个任务，还是 less 的例子，Gulp 的流程就是：

读取 less 文件 -> 编译成 css -> 压缩处理 -> 存储到磁盘

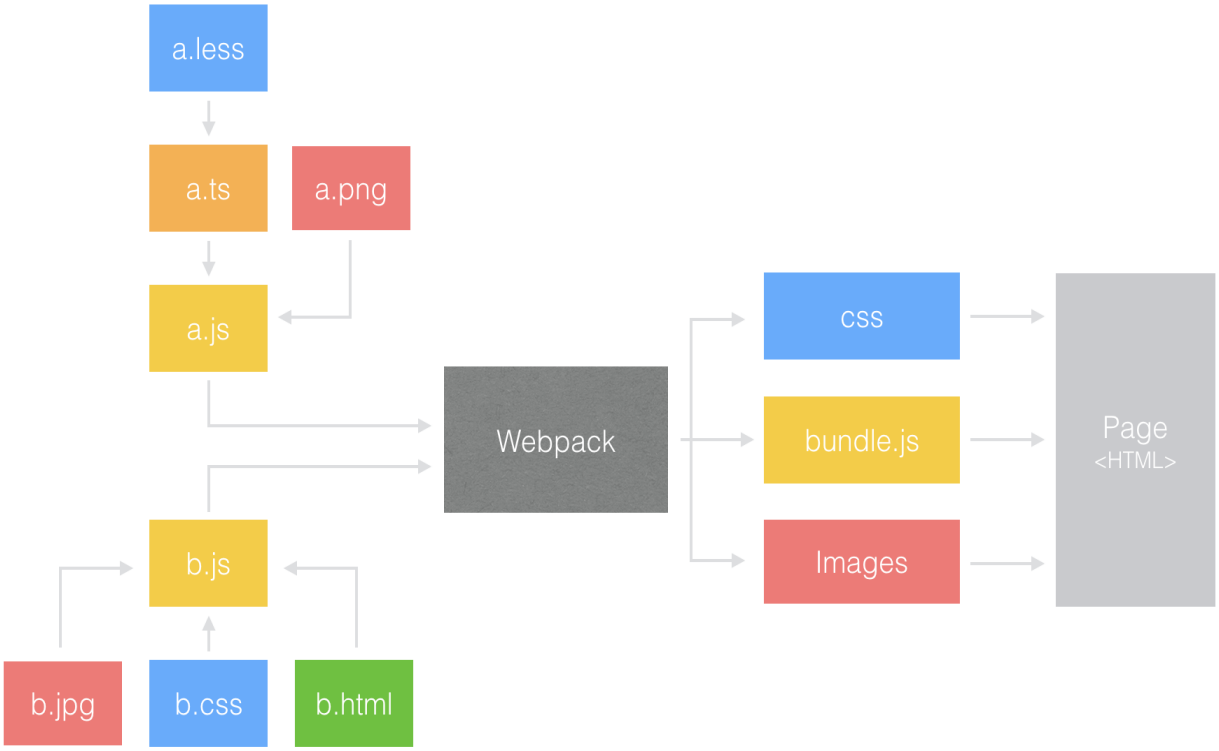
Gulp 作为任务类型的工具没有明显的缺点，唯一的问题可能就是完成相同的任务它需要写的代码更多一些，所以除非是项目有历史包袱（原有项目就是基于 Grunt 构建）在 Grunt 与 Gulp 对比看来还是比较推荐 Gulp!

适用场景:

通过上面的介绍可以看出它们侧重对整个过程的控制管理，实现简单、对架构无要求、不改变开发模式，所以非常适合前端、小型、需要快速启动的项目。

2、Webpack

Webpack 是目前最热门的前端资源模块化管理和打包工具，还是先通过一张图大致了解它的运行方式：



特点与不足

Webpack 的特点：

1. 把一切都视为模块：不管是 CSS、JS、Image 还是 HTML 都可以互相引用，通过定义 entry.js，对所有依赖的文件进行跟踪，将各个模块通过 loader 和 plugins 处理，然后打包在一起。
2. 按需加载：打包过程中 Webpack 通过 Code Splitting 功能将文件分为多个 chunks，还可以将重复的部分单独提取出来作为 commonChunk，从而实现按需加载。

Webpack 也是通过配置来实现管理，与 Grunt 不同的时，它包含的许多自动化的黑盒操作所以配置起来会简单很多

Webpack 的不足：

1. 上手比较难：官方文档混乱、配置复杂、难以调试（Webpack2 已经好了很多）对于新手而言需要经历踩坑的过程；
2. 对于 Server 端渲染的多页应用有点力不从心：Webpack 的最初设计就是针对 SPA，所以在处理 Server 端渲染的多页应用时，不管你如何 chunk，总不能真正达到按需加载的地步，往往要去考虑如何提取公共文件才能达到最优状态。

其实每个工具的官网上都有对工具的设计思想、要解决的问题、与其他工具的对比。自己摘抄下来，做个表格对比一下。高亮出每个工具独特的特性。这样你就知道什么时候需要用哪个工具了。
比如，你的工程模块依赖很简单，不需要把 js 或各种资源打包，只需要简单的合并、压缩，在页面中引用就好了。那就不需要 Browserify、Webpack。Gulp 就够用了。

反过来，如果你的工程庞大，页面中使用了很多库（SPA 很容易出现这种情况），那就可以选择某种模块化方案。至于是用 Browserify 还是 Webpack 就需要根据其他因素来判断了。比如团队已经在使用了某种方案，大家都比较熟悉了。再比如，你喜欢 Unix 小工具协作的方式，那就 Browserify。

充分了解各种工具、方案，选择合适的和自己需要的。没有绝对的好。优点换了场景也会变成缺点。