导读：当我们谈论前端工程化时，我们谈论什么



在日常前端开发工作中，你是否困惑于 npm、Yarn、Browserify、Gulp、Grunt、Webpack 等工具而不清楚他们有什么区别？是否觉得手头的项目环境过于复杂，各种配置已经超出了可控的范围，以至于都不敢更改它？你是否觉得工程模块太多构建太慢，点击发布按钮要等几分钟才能看到效果？

如果答案是肯定的，那么你可能遇到了有关前端工程化的问题。在这篇文章以及接下来的课程中，我会去解释前端工程化，并且尝试去解决上面的那些问题。

### 前端工程化是什么？

这是个很大的概念，但是在我们的日常开发中又很常见。当我们对一个工程进行设计并把它拆分成各个组件和模块时，我们是在做工程化；当我们用 Webpack 构建项目，配置好各个环境的打包配置时，我们是在做工程化；当我们为项目添加了 ESLint，并在每次提交之前自动检查代码质量时，我们是在做工程化。

如果要用一句话来概括，在我的理解中前端工程化是把前端开发工作带入到更加系统和规范体系的一系列过程。这个过程会包括源代码的预编译、模块处理、代码压缩等构建方面的工作。工程化会尽可能保证开发者的开发体验更加友好，保证源代码的质量以及依赖的完整性。工程化也会尽可能高效地将构建完成后的代码送达给客户端，来追求更加良好的用户体验。所有这些都属于工程化。

### 为什么要了解前端工程化？

如果一个开发者想做现代 JavaScirpt 应用，那么他就有理由了解前端工程化，因为通过工程化可以提高代码质量，降低开发成本。

会有人质疑说现在的前端开发是不是过于复杂了？一个没有用过 npm 和 Webpack 的开发者想写一个 Hello World 可能都需要学习和配置半天。在以前没有所谓“工程化”的时候，还不是照样写代码和发布代码来实现用户需求。

其实我们可以打个比方，把发送给客户端的页面理解成呈献给用户的一道菜。前端开发者是厨师，而工程化可以使开发者遵循正确的做菜流程——洗菜、切菜、炒菜，而不是将生的东西直接放到用户面前。

在 Web 技术刚开始的时候，还没有前端工程化这样一个东西。人们只是简单地把 HTML、CSS 和 JavaScript 直接混在一起丢到用户。而就如人类对于食物的追求在不断进步一样，虽然在最初级的阶段需求只是能填饱肚子，但慢慢地人们开始追求食物的质量。对于前端来说也是一样，用户的需求从最开始简单的页面在向复杂的应用发展。前端需要做的事情更多，同时也要追求更友好的用户体验。

对于厨师来说，要想做出高质量的食物需要顺手的工具以及正确的烹饪方法。对于开发者来说，要想输出高质量的代码也同样需要工程化来辅助。作为一个前端工程师，光会写代码是不够的，最多只是提供了好的原材料。要想将代码转化为高质量的产出提供给用户，必须要了解工程化。

另外，工程化也是为开发者服务的。通过预编译语言、模块热加载等技术可以提升开发效率，而利用自动化测试、lint 工具等可以保证代码的功能和质量。工程化可以有效降低开发成本，谁不想省下埋头 debug 的时间去做更有趣的事呢。

### 从工程化出现之前说起

前端工程化不是凭空出现的，一定是为了解决当时的一些问题而出现的，让我们先简单回溯一下前端开发的历史。

曾经的 Web 开发不同于现在，页面功能比较简单。开发者想添加一段逻辑，最直接的做法就是在 HTML 中插入一个 script 标签，然后直接在里面书写代码。听上去似乎十分的原始，但在当时确实很多都是这样做的，而且在业务需求简单的时候这也确实是最直接了当的实现方式。但是这种原始的做法也必然有它的缺点，主要有以下两个方面：

1. 全局作用域的污染。由于在每个 script 标签下顶层作用域即全局作用域，直接进行变量和函数声明会造成全局命名空间污染。假如一个页面有多个 script 标签，它们之间很有可能发生命名冲突。
2. 代码重用性差。在一个多页面应用的场景下，经常会有一些逻辑是这些页面之间共有的，此时我们不得不将这些代码复制粘贴到各个页面中。而当此处逻辑改动的时候我们也需要去更新所有页面的代码，造成很多额外的成本。

后来逐渐有一些针对这些问题的解决办法。首先，可以将 HTML 中内联的 JavaScript 提取出来成为单独的 JavaScript 文件。比如说一些页面公有的逻辑可以放在类似 common.js 中来被各个页面引用，这可以解决各个页面之间重用的问题。至于全局作用域污染的问题，则可以使用立即执行函数表达式将它包起来（ IIFE ），只把接口暴露到全局上。

(**function**() {

// 通过立即执行函数表达式将作用域隔离

**var** foo = 'bar';

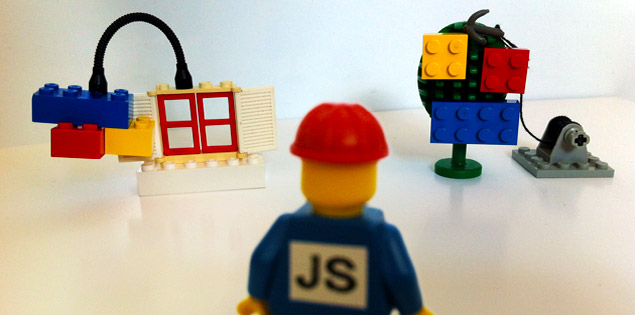
})();

看上去问题已经得到了解决，然而随着页面逻辑的复杂度增加开发者又面临了新的问题：

1. 页面 JavaScript 文件的引用顺序。由于 HTML 页面引用和处理 JavaScript 文件只能是顺序的（不考虑 async 等），因此页面的 JavaScript 之间依赖关系也必须是顺序的。而我们知道一个大型工程内部的模块依赖关系通常是树状的（比如 index.js 依赖 a、b、c 三个模块，而 a、b、c 又有各自的依赖），简单的顺序依赖关系无法满足需求。例如在 jQuery 最流行的时期，jQuery 本身以及其相关的插件之间有着各种各样的依赖关系，有些库可能自身包含 jQuery，不同的插件可能需要不同的 jQuery 版本，这些问题都不是简单的顺序依赖关系可以解决的。
2. 页面引用的 JavaScript 文件的长度与数量如何权衡。随着页面逻辑的增加，工程中的 JavaScript 文件越来越长，也越来越难以维护。一个页面的单个 JavaScript 文件可能有数千行甚至上万行。而如果按照功能来把页面逻辑切割成一个个小的 JavaScript 文件，则最终会走到另一个极端——页面请求过多。我们知道每个 HTTP 请求都是需要连接时间的，对于小模块而言每一个都要单独建立连接总归得不偿失，必然会导致页面渲染速度的下降。

上面所说的这些问题都属于前端开发的“原始时期”，那时候还没有工程化这种说法。然而逐渐暴露出来的问题已经让人们觉得不能再简单粗暴地采用如此原始的开发方式，模块化是第一步。

### 走向正轨的第一步——模块化



一个设计良好的系统应该是模块化的。一个最简单的原因，在一个模块化的系统中，当外界的需求亦或环境变化的时候，开发者可以更快地将问题定位到相应的模块，而不必面对纠缠在一起的逻辑不知如何下手。模块化可以使系统具备更强的可维护性。

被封装良好的模块应该具备特定且单一的功能，对外界只提供接口，而将具体实现封装在内部。Webpack 中有一个核心的理念——”一切皆模块”，即 HTML、JavaScript、CSS、图片等等都是模块，在后面会展开讲。

虽然模块化很重要，但是 JavaScript 诞生的时候并不具备模块这一特性，这主要是因为早期 Web 中的脚本大都比较简单，在设计之初只是为了实现 Web 上一些简单的功能。一直到 CommonJS 以及 AMD 的出现，为前端定义了模块的标准。也有了实现这些模块化的库，比如 RequireJS 以及 Browserify。可以让开发者将自己工程中的代码按模块进行划分，模块之间也不再仅仅是简单的顺序依赖关系。

另外在将代码提供给客户端之前，开发者可以通过 Browserify、Webpack 这些工具将工程代码进行打包，把所有依赖模块打包为单一的 JavaScript 文件。这样一来，对于开发者而言开发体验更加友好，因为开发中每次需要关注的仅仅是单个模块，而不是堆放在一起的上千行 JavaScript 文件；而对于客户端来说则只用接受单一的打包产物，解决了文件数量过多导致 HTTP 请求耗时长的问题。解决模块之间的依赖，并根据依赖树进行打包，是工程化解决的最基本的问题之一。

### 提升前端开发效率——预编译语言

上面说的只是 JavaScript 的模块化，那么我们很自然地就想到 CSS 的模块化。然而因 CSS 本身 @import 的性能问题，一般都是要通过 SASS、LESS 等预编译语言去实现其模块化。

例如在 SASS 中，通过 @import 语句我们可以导入其它模块。SASS 的编译器会处理模块之间的依赖，并最终将代码打包在一起生成 CSS。

在实际开发中，我们很多时候都会使用预编译语言来进行编码工作，然后经过 Webpack 等工具的构建将其编译为实际页面中的代码。使用预编译语言的主要目的是为了实现 HTML、CSS、JavaScript 不具备的特性。比如说上面提到的最常见的 SASS，它是 CSS 的预编译语言，通过它开发者可以使用模块、定义变量、编写嵌套规则等等来提高开发效率。

除了 CSS 的预编译语言，HTML 对应的有 HAML，JavaScript 对应的有 Coffee 等等。总体而言这些预编译语言的目的就是使开发体验更友好，开发者可以更高效地编写和维护代码。

当然现在预编译已经不仅仅是这些，我们还可以使用 Babel 预编译 JavaScript 来实现新的 ES 特性，以及使用 TypeScript 去做类型检查等，在预编译这里还可以有更多的想象力。

### 现代 JavaScript 应用必不可少的部分——包管理器

和 Java、C++ 这些语言不同，JavaScript 没有强大的标准库。许多常用的功能，比如日期处理、URL 处理、异步流程控制等往往都需要手工去编写，而采用外部已有的开源框架库或许是节约成本的最好办法。

Bower 作为包管理器最早进入人们的视野，大部门前端框架库也都提供了通过 Bower 安装的方式。通过它你可以获取项目需要的依赖，并且通过打包工具和业务代码打包到一起。

虽然现在当我们说包管理器可能首先想到的是 npm，但它最开始其实主要是为 Node.js 服务的，而人们逐渐意识到它也可以用于前端并且真正担当起前端包管理器的大任也就是近两年的时间。在现阶段来说 npm 已经超越 Bower 成为开发者首选的包管理器，而 Yarn 作为 Facebook 出品的新生代也不过是管理 node\_modules 的另一个工具罢了，与 npm 并没有什么本质上的区别。在后面会更详细地介绍包管理器常见的问题和处理办法。

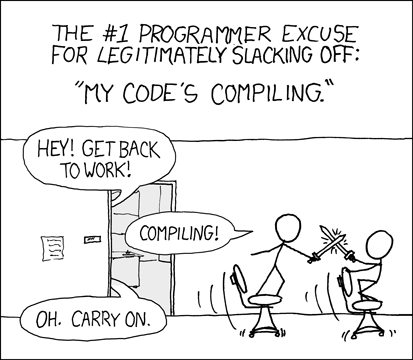
### 让机器做更多的事——构建流程管理

随着工程化的发展，交给构建过程的任务也越来越多，并且在不同环境下需要对任务进行区分。比如对于一个前端工程来说，除了需要预编译各种类型的文件、资源打包之外，本地环境下还要生成 source-map、配置模块热加载等等便于调试代码；而到了生产环境下则要对资源进行压缩，生成版本号等等。

因此对于开发者来说需要将这些任务统一起来进行管理，也就有了 Gulp 和 Grunt 等构建流程管理工具。这类工具的出现使得构建变得更加傻瓜化，通过项目中的一些配置，开发者可以使用简单的一行命令启动本地开发环境或者构建和发布整个工程。

相比于 Gulp 和 Grunt，Webpack 出现的更晚。它和前两者的核心定位其实不太一样，Webpack 本身只是作为一个模块打包工具的姿态出现的，但是利用一些相关工具和插件我们也可以完成整个工程的构建。Webpack 的“一切皆模块”以及“按需加载”两大特性使得它更好地服务于工程化。在后面的文章中会有很多关于 Webpack 的部分，包括 Webpack 的打包原理及优化、从零搭建起一个开发环境等，会详细讲解 Webpack 的使用以及其最新的特性。

### 摆脱冗长的等待——构建流程优化



现在当我们讨论工程化，效率和优化是出现得越来越频繁的词。当把工程化的各种功能都实现在我们的工程中之后，却发现完整地构建一遍需要好几分钟甚至更长。工程本身的越发庞大以及越来越多的构建任务使得耗时越来越久，此时是不是又怀念起了在 HTML 里写内联脚本的日子。

除了构建速度的问题，推给用户端的资源体积过大也是问题。需要针对项目的特点采用按需加载、异步加载、长效缓存等等策略。在后面会单独有一篇来讲构建方面的优化。

我们会将工程化相关的原理和实践结合起来，穿插进行。首先是模块化的相关内容，后面则会介绍 Webpack、包管理器、构建优化等等。在实践部分会有实际的工程演示

最后会介绍在去哪儿网前端开发流程中，大型项目是如何做工程化相关工作的，曾经踩过的坑以及解决方案。