前端工程化与模块化

前端工程化是什么？一提到工程化我们的第一反应往往就是 **webpack**。webpack 确实是前端工程化中重要的工具，但二者并不能划等号。

其实顾名思义，前端，工程化，就是把前端做成一项工程。这其中的区别类似于，**写一个简单的展示页面 = 小孩子搭积木，一个大型的项目 = 开发商盖房子**。搭积木只需要简单的手工操作，而盖房子则需要的工程规划、设计、资源管理等一系列流程。

前端工程化贯穿从编码、发布到运维的整个前端研发生命周期，**一切以提高效率、降低成本、质量保证为目的的手段都属于工程化**。它借鉴了软件工程相关的方法和思想，通过使用工具、流程、最佳实践和规范来提高前端开发效率、质量和可维护性；旨在解决前端开发中出现的各种挑战，包括项目复杂性、跨浏览器兼容性、性能优化、代码可维护性以及团队协作等问题。

前端开发模式进化史

首先让我们回顾一下前端开发模式的演化历史，前端工程化正是为了应对这些演化中出现的挑战和需求而发展起来的：

1. **前后端混合**：服务端渲染，javascript仅实现交互
2. **前后端分离**：借助 ajax 实现前后端分离、单页应用(SPA)等新模式
3. **模块化开发**：npm 管理模块、Webpack 编译打包资源
4. **模块化 + MVVM**：基于 React 或 Vue 等框架进行组件化开发，不再手动操作 html 元素

前端工程化解决了什么问题

那么前端工程化究竟解决了什么问题呢:

1. **全局作用域问题**：前端工程化可以帮助解决全局作用域污染问题。模块化开发工具如Webpack和ES6模块化帮助开发者将代码分解为模块，避免全局变量冲突，并提高代码的可维护性。
2. **编码规范**：通过代码规范工具（如ESLint、TSLint）和自动化代码格式化工具（如Prettier），前端工程化可以确保代码风格一致，减少错误和提高可读性。
3. **资源合并和压缩**：前端工程化工具可以自动合并和压缩CSS、JavaScript和图片等前端资源，以减小文件大小，提高页面加载速度，并减少带宽占用。
4. **高版本JS预发降级**：前端工程化工具可以使用特性检测和polyfill库，以确保新版本JavaScript特性在旧版本浏览器中仍然可用。这有助于实现跨浏览器兼容性。
5. **模块管理**：前端工程化工具和模块化开发使前端项目更易于管理，避免依赖混乱，促进代码的重用和维护。

6. **自动化测试**：前端工程化通过测试工具和自动化测试流程，帮助检测和预防潜在的问题，确保代码的可靠性。
7. **持续集成和持续交付（CI/CD）**：前端工程化支持CI/CD流程，以确保代码在每次更改后都经过构建、测试和部署。这有助于快速交付功能，减少错误。
8. **性能优化**：前端工程化工具支持性能优化策略，如延迟加载、资源缓存和减少HTTP请求次数，以提供更好的用户体验。
9. **团队协作**：前端工程化规范化项目结构、版本控制、文档和工作流程，促进多人协作，减少沟通和协调成本。

举几个企业中的例子，比如前端团队从几个人增加到了几百人，如果不统一代码规范，阅读和接手他人代码的心智负担较大；项目代码从几百几千行增加到几万几十万行之后，如果不做模块化处理，单个文件过大，阅读和维护困难，可复用性差；项目数量从几十个发展到成千上万个，如果没有前端研发的脚手架，每个项目都要重复搭建，不同开发人员构建的项目难以统一管理。

前端模块化

前端模块化是前端工程化的一个重要组成部分，前者关注代码的组织和结构，而后者关注整个前端开发过程的自动化和最佳实践。前端工程化借助前端模块化来提高代码的组织和可维护性，从而解决前端开发中的一系列问题。

前端模块化是什么?

前端模块化指的是将前端代码分解成**独立的**、**可复用的**模块，以便更好地组织、维护和扩展代码。模块可以包括JavaScript、CSS、HTML等各种前端资源。前端模块化的目标是将复杂的前端应用程序分解为小块，每个块都有特定的功能，可以独立开发和测试。

前端模块是一种**规范**，而不是具体的实现。比方说 `Node.js` 实现了 `CommonJS` 规范， `ES6` 模块提供了 `ESM` 规范。这些规范有两个共性：

- 将复杂程序根据规范拆分成若干模块，**一个模块包括输入和输出**
- **模块的内部实现是私有的，对外暴露接口与其他模块通信**

前端模块化发展史

① 全局函数模式

将不同的功能封装到单独的全局函数中，通过函数引用功能。比方说一个加法模块：

```
1 js
2 复制代码
3 function sum(a, b) {    return a + b;}
```

这就是一个全局函数，可以在任何地方调用。

缺点： 如果出现相同的函数名，容易引起冲突。

② 命名空间模式

在 `window` 下新建一个对象属性来存放模块(命名只要不冲突即可，这样只需要确保这个属性名唯一，就能解决全局函数模式的缺点，如 `__Module`)，再将模块中的变量和功能作为 `__Module` 的属性。

```
1 js
2 复制代码
3 var __Module = {    sum: function(a, b) {        return a + b;    }}
```

缺点： 外部能够修改模块内部的数据，丧失了封装性。(`window.__Module.属性名` 可以直接修改模块)

③ 立即执行函数模式

通过**IIFE（立即调用函数表达式）**，利用**闭包**来创建私有变量。

为了解决 `namespace` 模式中的缺点，我们可以将创建 `__Module` 的过程放在一个IIFE中：

```
1 js
2 复制代码
3 (function () {    var x = 1;    function getX() {        return x;    }
  function setX(val) {        x = val;    }    function sum(a, b) {        return
   a + b;    }    window.__Module = {        x,        setX,        getX,
  sum,    };})();
```

这样 `window.__Module` 下的 `x` 只是一个拷贝，真正的 `x` 存放在IIFE的私有作用域中，而不是全局作用域。这样，外部代码就无法直接访问或修改真正的 `x` 变量。

我们还可以稍加改动，实现一个增强的IIFE模式，使它能够支持自定义传入依赖：

```
1 js
2 复制代码
3 // 模块A(function (dependencyA, dependencyB) {    // 在这里可以使用 dependencyA
  和 dependencyB    function doSomething() {        dependencyA.doThis();
  dependencyB.doThat();    }    // 向全局暴露公共接口    window.ModuleA = {
  doSomething: doSomething    };})(window.DependencyA, window.DependencyB);// 模块
  B(function () {    function doThis() {        // 实现某些功能    }    // 向全局暴
  露公共接口    window.DependencyA = {        doThis: doThis    };})();// 模块
  C(function () {    function doThat() {        // 实现某些功能    }    // 向全局暴
  露公共接口    window.DependencyB = {        doThat: doThat    };})();
```

至此，已经是我们自己通过纯 JavaScript 来实现模块化方案的最终方案了。但是，它没有特定的语法支持，代码阅读困难，也没有完善的依赖管理、模块加载机制。

所以，为了应对大型项目中复杂的模块化需求，我们需要更加现代的模块系统。

⭐ CommonJS

`CommonJS` 是 `Node.js` 中默认的模块化规范：

- 文件级别的模块作用域(**每个文件就是一个作用域**)：每个 **CommonJS** 模块都有自己的作用域。
- 使用 `require` 函数来导入，通过 `module.exports` 导出。
- `CommonJS` 模块是**同步加载**的，这意味着模块在导入时会阻塞执行，直到模块完全加载并可用，并且模块加载的顺序会按照其在代码中出现的顺序。
- 模块可以多次加载，**首次加载的时候会运行模块并对输出结果进行缓存**，再次加载时会直接使用缓存中的结果。

Node 中 CommonJS 的原理可以分成三部分来看：

主模块加载

应用程序的入口点，包含 `require` 调用加载其他模块。

模块加载

1. **解析模块标识符：** 它会解析模块的标识符，通常是一个文件路径，以确定要加载的模块。
2. **检查模块缓存：** CommonJS实现会检查模块缓存来查看是否已经加载了该模块。如果已经加载，它会直接返回缓存中的模块对象。
3. **创建模块对象：** 如果模块尚未加载，CommonJS实现会创建一个新的模块对象，通常是一个包含了 `exports` 和 `module` 属性的对象。
4. **执行模块代码：** 接下来，CommonJS实现会将模块的代码包装在一个IIFE中，并向该IIFE传递 `exports` 、 `module` 和 `require` ，以确保模块的作用域是隔离的。模块的代码会在这个作用域内执行，可以在模块内定义变量和函数，并通过 `exports` 和 `module.exports` 暴露模块的接口。

```
1 js
2 复制代码
3 (function (exports, module, require) {  // 模块内部的代码  // 可以在这里定义模块内的
  变量和函数  // 通过exports和module.exports来暴露模块的接口})(exports, module,
  require);
```

1. **缓存模块：** 模块加载完成后，CommonJS实现会将这个模块对象缓存起来，使用模块标识符作为键，以便后续 `require` 调用可以直接返回缓存的模块对象。

模块缓存

已加载的模块以模块标识符为键，模块对象为值存储在模块缓存中。 这个缓存允许模块在后续的 `require` 调用中被快速访问，而不需要重新加载。

AMD

Node 模块通常都位于本地，加载速度快，不用担心同步加载带来的阻塞问题。但是在浏览器运行过程中，同步加载会阻塞页面的渲染。

`require.js` 中实现了**AMD（Asynchronous Module Definition）**，旨在解决浏览器环境中的异步模块加载和依赖管理问题。它的主要特点是允许在浏览器中异步加载模块，以提高性能和模块化的管理。

CMD

`sea.js` 中实现了**CMD（Common Module Definition）**，它整合了 CommonJS 和 AMD 的优点。

实际上，AMD和CMD等模块规范有一个最大的问题：它们没有得到官方的JavaScript语言规范的支持，这也是他们最终过时了的原因。在现代前端开发中，ES6模块已经成为了主要的模块化解决方案。

⭐ ESM

前面所说的几种模块化规范都必须在运行时才能确定依赖和输入输出，而 `ESModule` 的理念是在编译时就确定模块依赖的输入输出。

⭐ CommonJS 和 ESModule 规范对比：

- CommonJS 模块输出的是值的**拷贝**，ESM 模块输出的是值的**引用**。
- CommonJS 模块是**运行时**加载，ESM 模块是**编译时**输出接口。
- CommonJS 是单个对象导出，多次导出会覆盖之前的结果；ESM 可以导出多个。
- CommonJS 模块是**同步加载**，ESM 支持**异步加载**。
- CommonJS 的 this 是当前模块，ESM 的 this 是 undefined。

现在大多数浏览器中默认的模块化规范都是 ESM 了，作为一种规范它已经比较成熟了，但是我们在浏览器模块化问题上仍有一些问题未能解决：

- 浏览器没有模块管理能力，模块分散在各个项目中无法复用。
- 性能加载慢，大型项目中无法直接使用。

为了解决这两个问题，前端工程化又引入了两个新的工具：

引入 **npm** 负责管理模块，引入打包工具比如 **webpack** 进行打包聚合提高性能。

npm 简介

npm 的全称是 **Node Package Manager**，它是一个用于 Node.js 包的默认包管理器。npm 的主要目标是提供一个集中的、共享的解决方案，用于开发者之间共享和复用代码。在 npm 出现之前，开发者想要在另一个项目中复用某个模块，只能通过复制和粘贴文件的方式。有了 npm 之后，开发者可以把所有模块都上传到仓库（registry）：在模块内创建 `package.json` 文件来标注模块的基本信息，然后通过 `npm publish` 命令发布模块；使用时通过 `npm install` 命令安装指定模块到 `node_modules` 目录。

webpack 简介

虽然 npm 能解决模块的管理问题，但它无法解决加载性能问题。为了解决这个问题，webpack 诞生了。webpack 的整个工作流程可以简单地理解为先合并、再分割：

- **合并**：为了解决项目中依赖文件过多，而导致 HTTP 请求过多的问题，webpack 会把所有资源文件视为模块，分析模块之间的依赖关系，并把所有的依赖打包成一个或多个 `bundle.js` 文件。
- **分割**：合并打包会带来一个问题——单文件过大，导致加载时间过长。为了解决这个问题，Webpack 引入了代码分割的概念。代码分割允许你将一个大的 bundle 文件分割成多个小文件，这些小文件在需要时才会被加载。这提高了应用程序的加载性能，因为浏览器只需要下载当前页面所需的代码块，而不是整个应用的所有代码。Webpack 提供了不同的代码分割策略，如按路由、按组件或按异步加载。

在 webpack 打包过程中，首先会通过 `entry` （入口）找到需要打包的文件，然后通过 `module` （模块）来处理各种类型的文件。在处理文件时，会用到各种 `loader` （加载器），比如 `babel-loader` 用来处理 JS 文件，`css-loader` 和 `style-loader` 用来处理 CSS 文件。最后通过 `output` （输出）把处理过的文件输出到指定的目录。