开篇： 让 Vite 助力你的前端工程化之路

发布于 2022-05-09

当下，在项目开发的过程中，前端工程师们越来越离不开构建工具了，可以说**构建工具已经成为了前端工程项目的标配。**

不过，如今的前端构建工具可谓 乱花渐欲迷人眼 ，有远古时代的 browserify 、 grunt ，有传统的 Webpack 、 Rollup 、 Parcel ，也有现代的 Esbuild 、 Vite 等等，不仅种类繁多，更新也很快。

于是，很多朋友会问我，到底哪个构建工具更好用、值得学。事实上，**无论工具层面如何更新，它们解决的核心问题，即前端工程的痛点是不变的。**因此，想要知道哪个工具更好用，就要看它解决前端工程痛点的效果。

那么，前端工程都有哪些痛点呢?

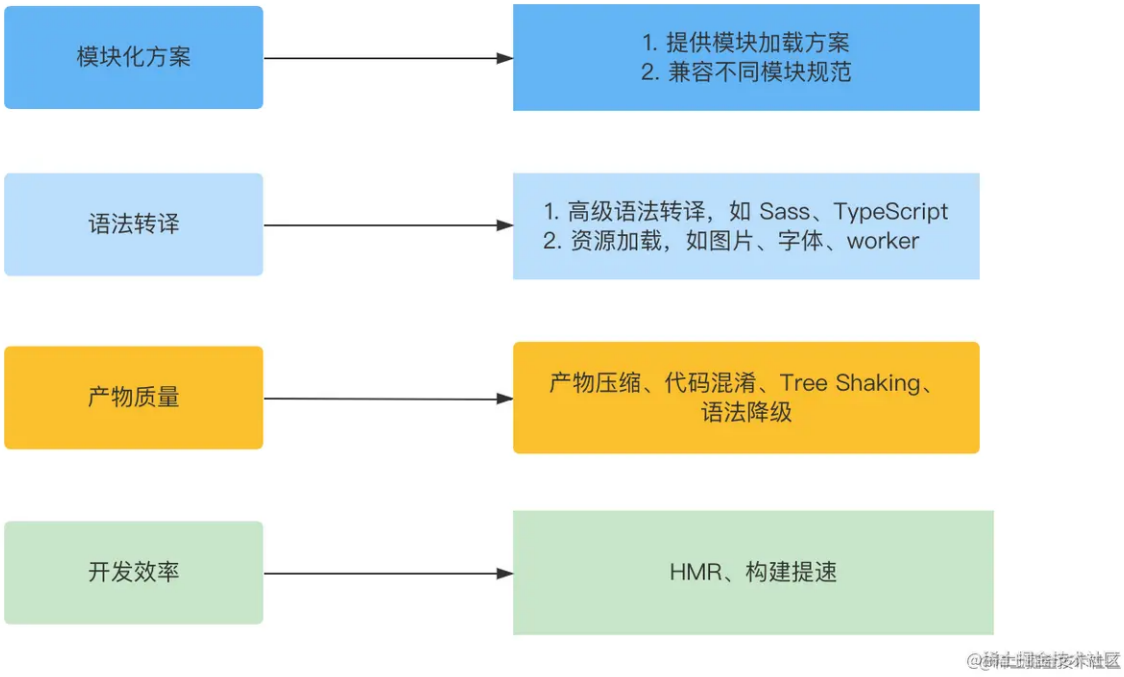
首先是前端的**模块化需求。**我们知道，业界的模块标准非常多，包括 ESM、CommonJS、AMD 和 CMD 等等。前端工程一方面需要落实这些模块规范，保证模块正常加载。另一方面需要兼容不同的模块规范，以适应不同的执行环境。

其次是**兼容浏览器，编译高级语法。**由于浏览器的实现规范所限，只要高级语言/语法（TypeScript、 JSX 等）想要在浏览器中正常运行，就必须被转化为浏览器可以理解的形式。这都需要工具链层面的支持，而且这个需求会一直存在。

再者是**线上代码的质量**问题。和开发阶段的考虑侧重点不同，生产环境中，我们不仅要考虑代码的 安全性 、 兼容性 问题，保证线上代码的正常运行，也需要考虑代码运行时的性能问题。由于浏览器的版本众多，代码兼容性和安全策略各不相同，线上代码的质量问题也将是前端工程中长期存在的一个痛点。

同时， 开发效率 **也不容忽视。** 我们知道，**项目的冷启动/二次启动时间、热更新时间**都可能严重影响开发效率，尤其是当项目越来越庞大的时候。因此，提高项目的启动速度和热更新速度也是前端工程的重要需求。

那么，前端构建工具是如何解决以上问题的呢？



- 模块化方面，提供模块加载方案，并兼容不同的模块规范。
- 语法转译方面，配合 `Sass` 、 `TSC` 、 `Babel` 等前端工具链，完成高级语法的转译功能，同时对于静态资源也能进行处理，使之能作为一个模块正常加载。
- 产物质量方面，在生产环境中，配合 `Terser` 等压缩工具进行代码压缩和混淆，通过 `Tree Shaking` 删除未使用的代码，提供对于低版本浏览器的语法降级处理等等。
- 开发效率方面，构建工具本身通过各种方式来进行性能优化，包括 使用原生语言 `Go/Rust` 、 `no-bundle` 等等思路，提高项目的启动性能和热更新的速度。

为什么 Vite 是当前最高效的构建工具？

现在，让我们回到一开始提出的问题，到底哪个工具更好用？或者说，哪个工具解决前端工程痛点的效果更好？

The State of JavaScript Survey 最近的调查结果中显示， Vite 在全球开发者中的满意度超过 98%，已经被用到了 `SvelteKit` 、 `Astro` 这些大型框架中，成为当下最受瞩目的前

端构建工具。我也最推荐你使用它。为什么是 Vite 呢？我们可以根据上面说的四个维度来审视它。

首先是开发效率。传统构建工具普遍的缺点就是太慢了，与之相比，Vite 能将项目的启动性能提升一个量级，并且达到毫秒级的瞬间热更新效果。

就拿 Webpack 来说，我在工作中发现，一般的项目使用 Webpack 之后，启动花个几分钟都是很常见的事情，热更新也经常需要等待十秒以上。这主要是因为：

- 项目冷启动时必须递归打包整个项目的依赖树
- JavaScript 语言本身的性能限制，导致构建性能遇到瓶颈，直接影响开发效率

这样一来，代码改动后不能立马看到效果，自然开发体验也越来越差。而其中，最占用时间的就是代码打包和文件编译。

而 Vite 很好地解决了这些问题。一方面，Vite 在开发阶段基于浏览器原生 ESM 的支持实现了 `no-bundle` 服务，另一方面借助 Esbuild 超快的编译速度来做第三方库构建和 TS/JSX 语法编译，从而能够有效提高开发效率。

除了开发效率，在其他三个维度上， Vite 也表现不俗。

- 模块化方面，Vite 基于浏览器原生 ESM 的支持实现模块加载，并且无论是开发环境还是生产环境，都可以将其他格式的产物(如 CommonJS)转换为 ESM。

- 语法转译方面，Vite 内置了对 TypeScript、JSX、Sass 等高级语法的支持，也能够加载各种各样的静态资源，如图片、Worker 等等。

- 产物质量方面，Vite 基于成熟的打包工具 Rollup 实现生产环境打包，同时可以配合 `Terser` 、 `Babel` 等工具链，可以极大程度保证构建产物的质量。

因此，如果你想要学习一个前端构建工具，Vite 将会是你当下一个最好的选择。它不仅解决了传统构建工具的开发效率问题，而且具备一个优秀构建工具的各项要素，还经历了社区大规模的验证与落地。

如何才能学好 Vite ？

不过，很多人在学习和应用 Vite 的过程中总会遇到各种各样的问题。

比如说，很多 Vite 学习资料既不系统，也不深入。绝大多数的文章只能教会我们如何搭建一个简单的脚手架项目，甚至代码都不一定正确。

即使通过资料学完了 Vite 的相关知识，但因为对 Vite 的生态了解不够，遇到实际问题的时候依然不知道要使用哪些插件或者解决方案。

- 第三方库里面含有 CommonJS 代码导致报错了怎么办?
- 想在开发过程中进行 Eslint 代码规范检查怎么办?
- 生产环境打包项目后，如何产出构建产物分析报告?
- 如果要兼容不支持原生 ESM 的浏览器，怎么办?

而且，如果你对 Vite 底层使用的构建引擎 Esbuild 和 Rollup 不够熟悉，遇到一些需要定制的场景，往往也会捉襟见肘。

- 写一个 Esbuild 插件来处理一下问题依赖
- 对于 Rollup 打包产物进行自定义拆包，解决实际场景中经常出现的循环依赖问题
- 使用 Esbuild 的代码转译和压缩功能会出现哪些兼容性问题？如何解决?

当然，作为一个构建工具，Vite 的难点不仅在于它本身的灵活性，也包含了诸如 `Babel`、`core-js` 等诸多前端工具链的集成和应用。

- `@babel/preset-env` 的 `useBuiltIns` 属性各个取值有哪些区别?
- `@babel/polyfill` 与 `@babel/runtime-corejs` 有什么区别?
- `@babel/plugin-transform-runtime` 与 `@babel/preset-env` 的 `useBuiltIn` 相比有什么优化?
- core-js 的作用是什么？其产物有哪些版本？ `core-js` 和 `core-js-pure` 有什么区别?

此外，由于构建工具(不仅包括 Vite，也包括底层引擎 Rollup)的源码晦涩难懂，涉及大量的基础工具库，导致很多人对构建工具原理的理解只浮于表面，很难更进一步。

作为一名深耕在一线的前端工程师，我的日常工作就是跟各种构建工具打交道，在公司中诸多的业务项目中落地了 Vite，有丰富的 Vite 实战经验和源码阅读经验，也给 Vite 仓库贡献过一些代码。因此，我也非常乐意将自己在 Vite 方面的实战经验与学习方法通过小册系统性地分享给大家。

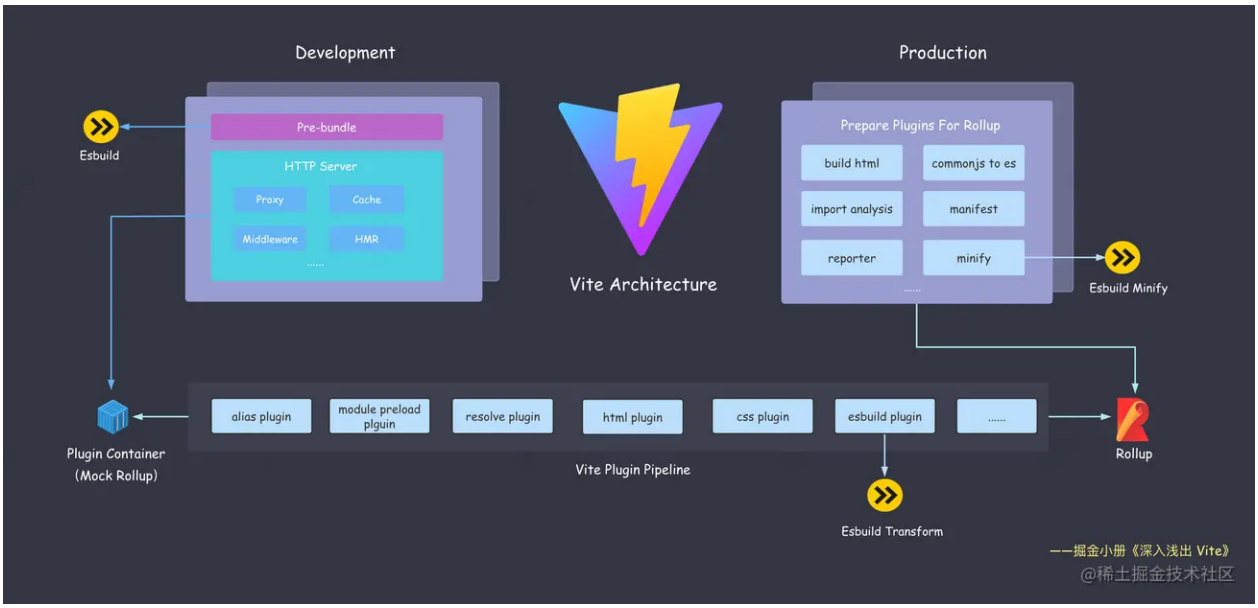
那么， Vite 该如何学习呢？我按照**循序渐进、可实操、可延伸**的三个原则，由浅入深设计课程内容，提供大量的实战场景和案例，同时尽可能给大家提供解决问题的方法和视角，让大家学完课程后能做到举一反三。具体来说，我将课程设计为 5 个模块。

在基础使用篇中，我将与你从 0 开始实现 Vite 项目初始化，接入各种现代化的 CSS 方案，集成 Eslint、Styelint、Commonlint 等一系列 Lint 工具链，处理各种形式的静态资源，掌握 Vite 预编译的各种使用技巧，最终让你能独立搭建一个相对完整的脚手架工程。

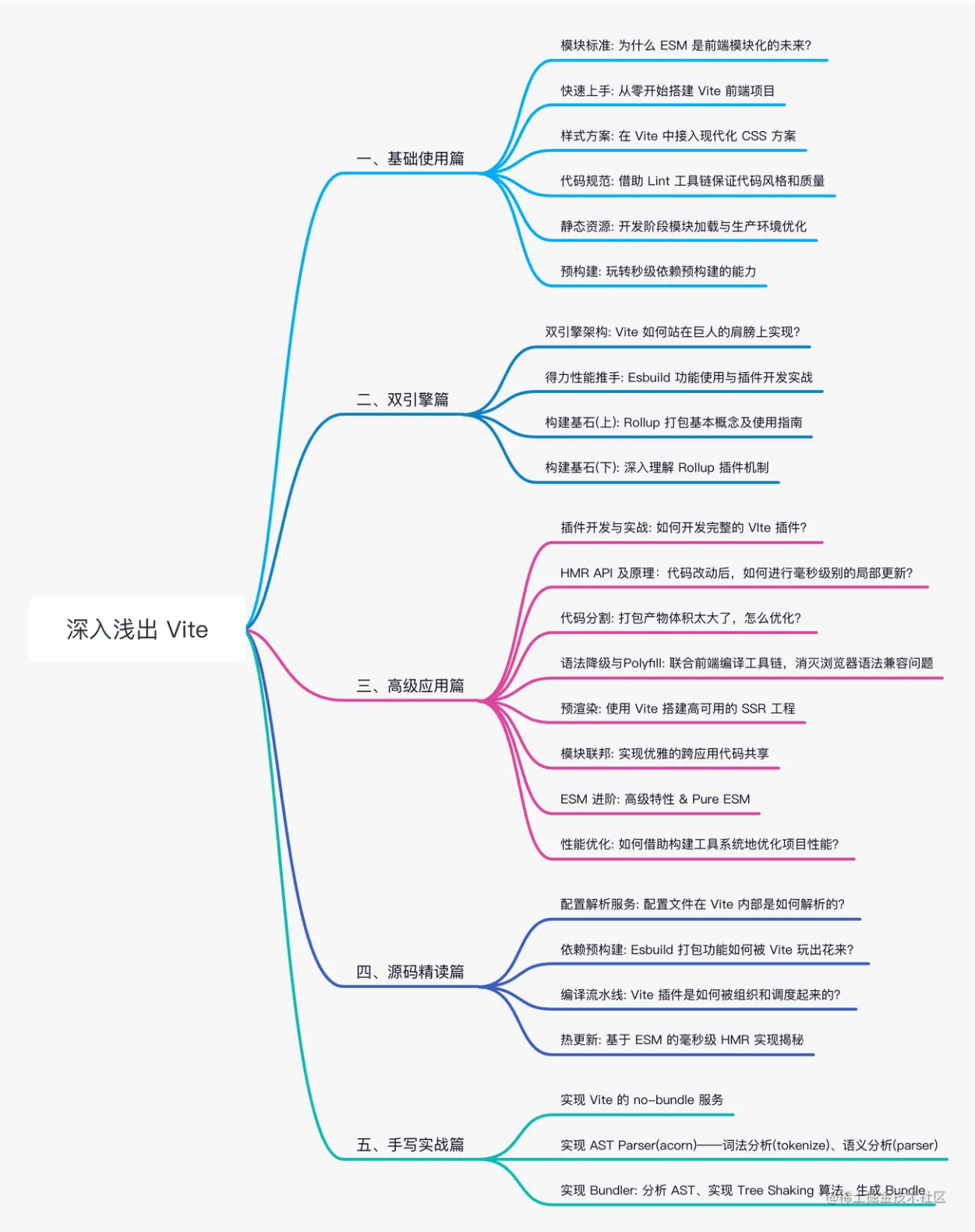
在双引擎篇中，我们会学习 Vite 的双引擎架构， Esbuild 和 Rollup 相关的内容，包括它们的基本使用和插件开发，掌握 最小必要知识 ，为后续的高级应用作铺垫。

而高级应用篇， 我们将学习 Vite 的各种高级用法和构建性能优化手段，学会如何编写一个完整的 Vite 插件，熟练进行生产环境拆包，使用 Vite 搭建复杂的 SSR 工程，实现基于模块联邦的跨应用模块共享架构。不管是项目性能优化技巧，还是对前端底层标准和规范的理解，你都会从这一模块得到不少提升。

接下来，我们将一起剖析 **Vite 的核心源码**，理解诸如 `JIT` 、 `Proxy Module` 、 `Module Graph` 、 `HMR Boundary` 和 `Plugin Container` 等源码中重要概念的作用及底层实现，一步步教你学会阅读 Vite 的源码，将如下架构图中的关键环节各个击破，学透 Vite 实现原理。



最后是手写实战篇。 首先，我们会手写 Vite 的开发时 no-bundle 服务，也就是开发环境下基于浏览器原生 ESM 的 Dev Server。然后，我也会带你一步步完成一个生产环境打包工具（Bundler），从 AST 解析的功能开始，完成代码的词法分析（tokenize）和语义分析（parse），实现模块依赖图和作用域链的搭建，并完成 Tree Shaking、循环依赖检测及 Bundle 代码生成，最终实现一个类似 Rollup 的 Bundler。



可以看到，我们在课程中非常重视上手实战。课程的代码全部会上传至 Github 仓库(仓库地址)，基本上每一节内容都有能 run 起来的代码案例。尤其在最后一章，为了让你理解构建工具的底层原理，我会带你一步步搭建一个简单的构建工具，进行上千行代码的手写实战，做到真正的代码可实操。

最后，我希望在这本小册中，我们能一起深入 Vite 的实战要点和实现原理 ，领略前端工程化构建领域的底层风光，真正实现 Vite 从入门到进阶!

上一篇：课程介绍

下一篇：模块标准：为什么 ESM 是前端模块化的未来?