双引擎架构: Vite 是如何站在巨人的肩膀上实现的?

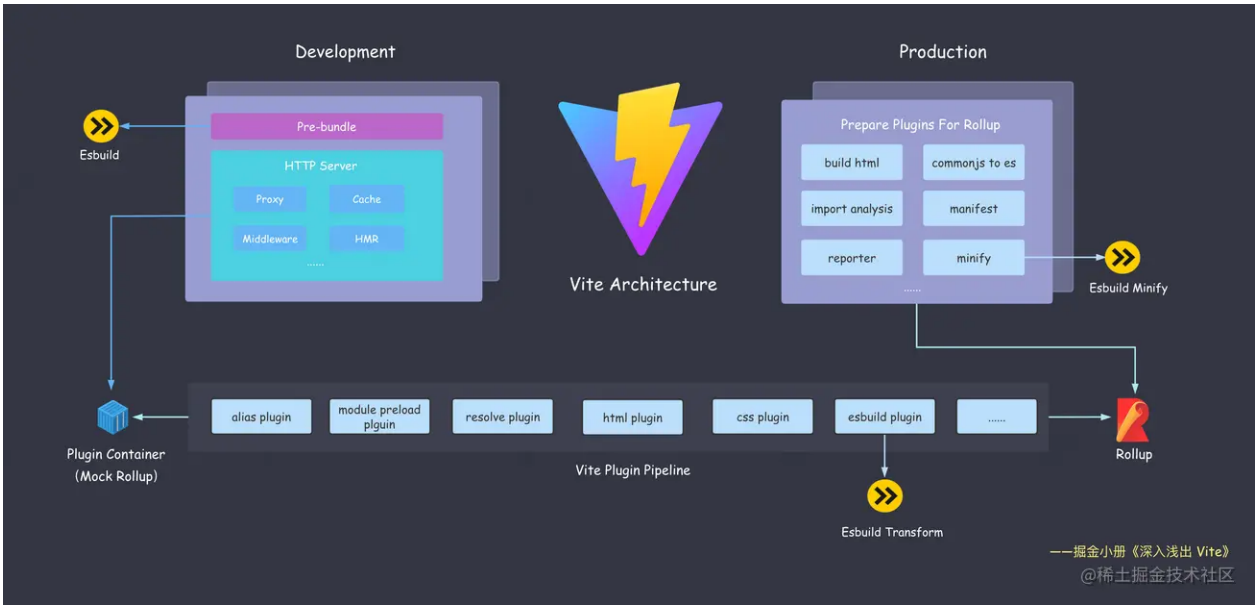
发布于 2022-05-09

在前面的章节中，我们学习了很多 Vite 使用和项目搭建的内容。接下来让我们将目光集中到 Vite 本身的架构上，一起聊聊它是如何站在巨人的肩膀上实现出来的。所谓的 巨人 ，指的就是 Vite 底层所深度使用的两个构建引擎—— `Esbuild` 和 `Rollup` 。

那么，这两个构建引擎对于 Vite 来说究竟有多重要？在 Vite 的架构中，两者各自扮演了什么样的角色？本小节，我将和你一起拆解 Vite 的双引擎架构，深入分析 `Esbuild` 和 `Rollup` 究竟在 Vite 中做了些什么。

Vite 架构图

很多人对 Vite 的双引擎架构仅仅停留在 开发阶段使用 `Esbuild`，生产环境用 `Rollup` 的阶段，殊不知，Vite 真正的架构远没有这么简单。一图胜千言，这里放一张 Vite 架构图：



相信对于 Vite 的双引擎架构，你可以从图中略窥一二。在接下来的内容中，我会围绕这张架构图展开双引擎的介绍，到时候你会对这份架构图理解得更透彻。

性能利器 Esbuild

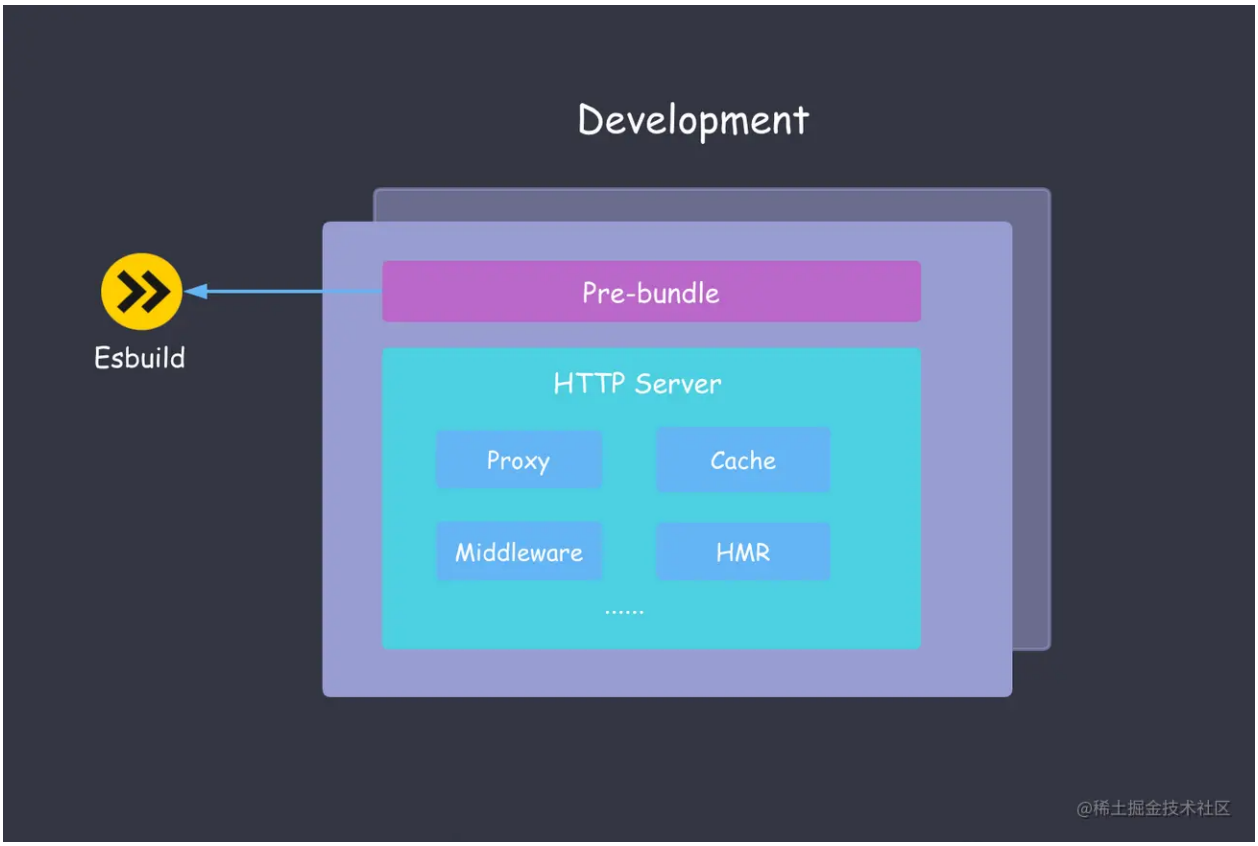
必须要承认的是， `Esbuild` 的确是 Vite 高性能的得力助手，在很多 关键的构建阶段 让 Vite 获得了相当优异的性能，如果这些阶段用传统的打包器/编译器来完成的话，开发体验要下降一大截。

> 关于 Esbuild 为什么快，我会在下一节展开介绍。

那么，Esbuild 到底在 Vite 的构建体系中发挥了哪些作用?

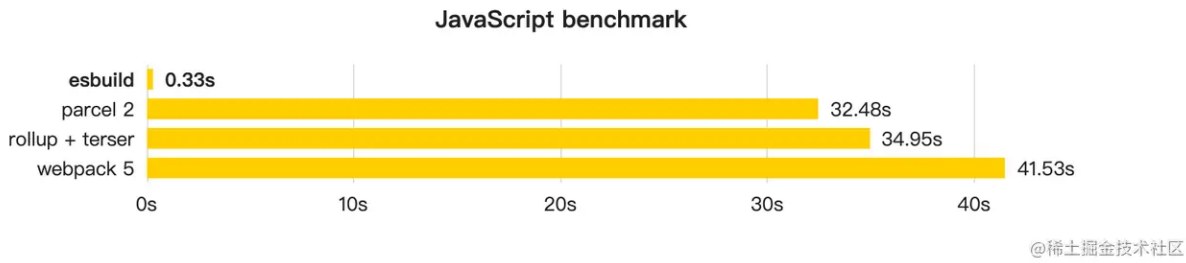
一、依赖预构建——作为 Bundle 工具

首先是**开发阶段的依赖预构建**阶段。



一般来说， `node_modules` 依赖的大小动辄几百 MB 甚至上 GB ，会远超项目源代码，相信大家都深有体会。如果这些依赖直接在 Vite 中使用，会出现一系列的问题，这些问题我们在**依赖预构建**的小节已经详细分析过，主要是 ESM 格式的兼容性问题和海量请求的问题，不再赘述。总而言之，对于第三方依赖，需要在应用启动前进行**打包**并且**转换为 ESM 格式。**

Vite 1.x 版本中使用 Rollup 来做这件事情，但 Esbuild 的性能实在是太恐怖了，Vite 2.x 果断采用 Esbuild 来完成第三方依赖的预构建，至于性能到底有多强，大家可以参照它与传统打包工具的性能对比图:



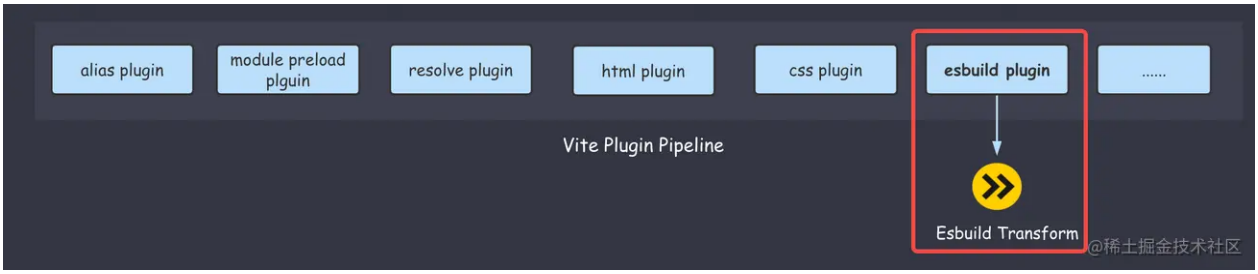
当然，Esbuild 作为打包工具也有一些缺点。

- 不支持降级到 `ES5` 的代码。这意味着在低端浏览器代码会跑不起来。
- 不支持 `const enum` 等语法。这意味着单独使用这些语法在 esbuild 中会直接抛错。
- 不提供操作打包产物的接口，像 Rollup 中灵活处理打包产物的能力(如 `renderChunk` 钩子)在 Esbuild 当中完全没有。
- 不支持自定义 Code Splitting 策略。传统的 Webpack 和 Rollup 都提供了自定义拆包策略的 API，而 Esbuild 并未提供，从而降级了拆包优化的灵活性。

尽管 Esbuild 作为一个社区新兴的明星项目，有如此多的局限性，但依然不妨碍 Vite 在**开发阶段**使用它成功启动项目并获得极致的**性能提升**，生产环境处于稳定性考虑当然是采用功能更加丰富、生态更加成熟的 Rollup 作为依赖打包工具了。

二、单文件编译——作为 TS 和 JSX 编译工具

在依赖预构建阶段， Esbuild 作为 Bundler 的角色存在。而在 TS(X)/JS(X) 单文件编译上面，Vite 也使用 Esbuild 进行语法转译，也就是将 Esbuild 作为 Transformer 来用。大家可以在架构图中 `Vite Plugin Pipeline` 部分注意到:

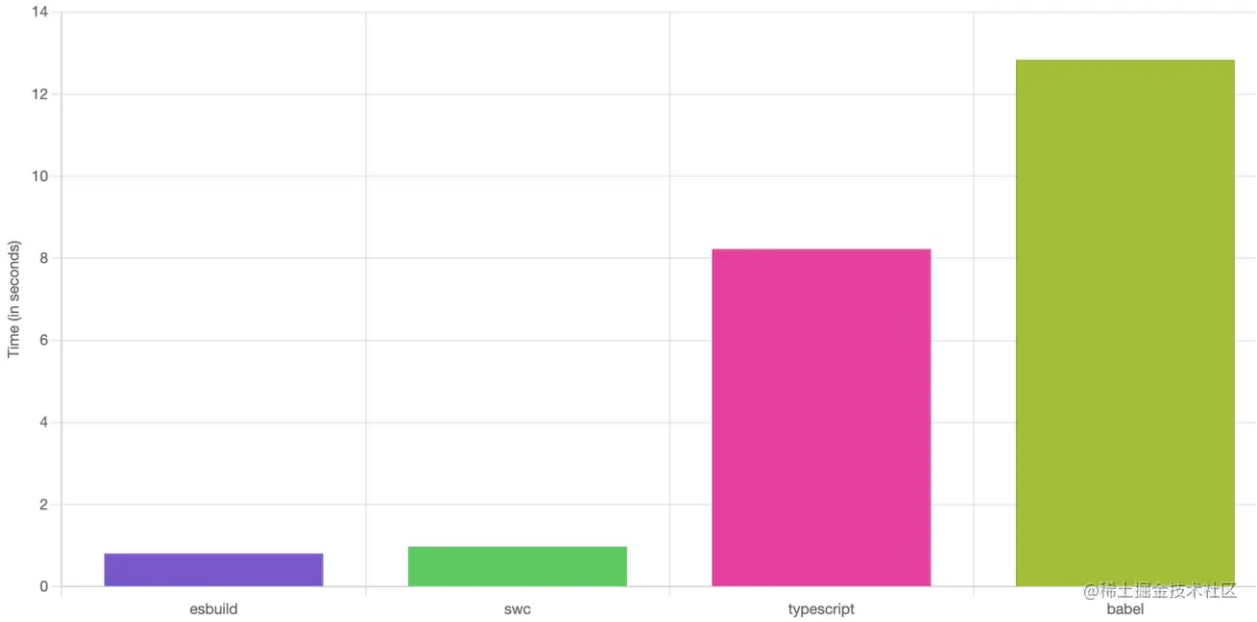


也就是说，Esbuild 转译 TS 或者 JSX 的能力通过 Vite 插件提供，这个 Vite 插件在开发环境和生产环境都会执行，因此，我们可以得出下面这个结论:

> Vite 已经将 Esbuild 的 Transformer 能力用到了生产环境。尽管如此，对于低端浏览器场景，Vite 仍然可以做到语法和 Polyfill 安全，详情见 小册第 15 节——语法降级与 Polyfill。

这部分能力用来替换原先 Babel 或者 TSC 的功能，因为无论是 Babel 还是 TSC都有性能问题，大家对这两个工具普遍的认知都是: **慢，太慢了。**

当 Vite 使用 Esbuild 做单文件编译之后，提升可以说**相当大**了，我们以一个巨大的、50 多 MB 的纯代码文件为例，来对比 `Esbuild` 、 `Babel` 、 `TSC` 包括 `SWC` 的编译性能:



可以看到，虽然 Esbuild Transfomer 能带来巨大的性能提升，但其自身也有局限性，最大的局限性就在于 TS 中的类型检查问题。这是因为 Esbuild 并没有实现 TS 的类型系统，在编译 `TS` (或者 `TSX`) 文件时仅仅抹掉了类型相关的代码，暂时没有能力实现类型检查。

也因此，**快速上手**这一节，我让大家注意初始化工程的构建脚本， `vite build` 之前会先执行 `tsc` 命令，也就是借助 TS 官方的编译器进行类型检查。

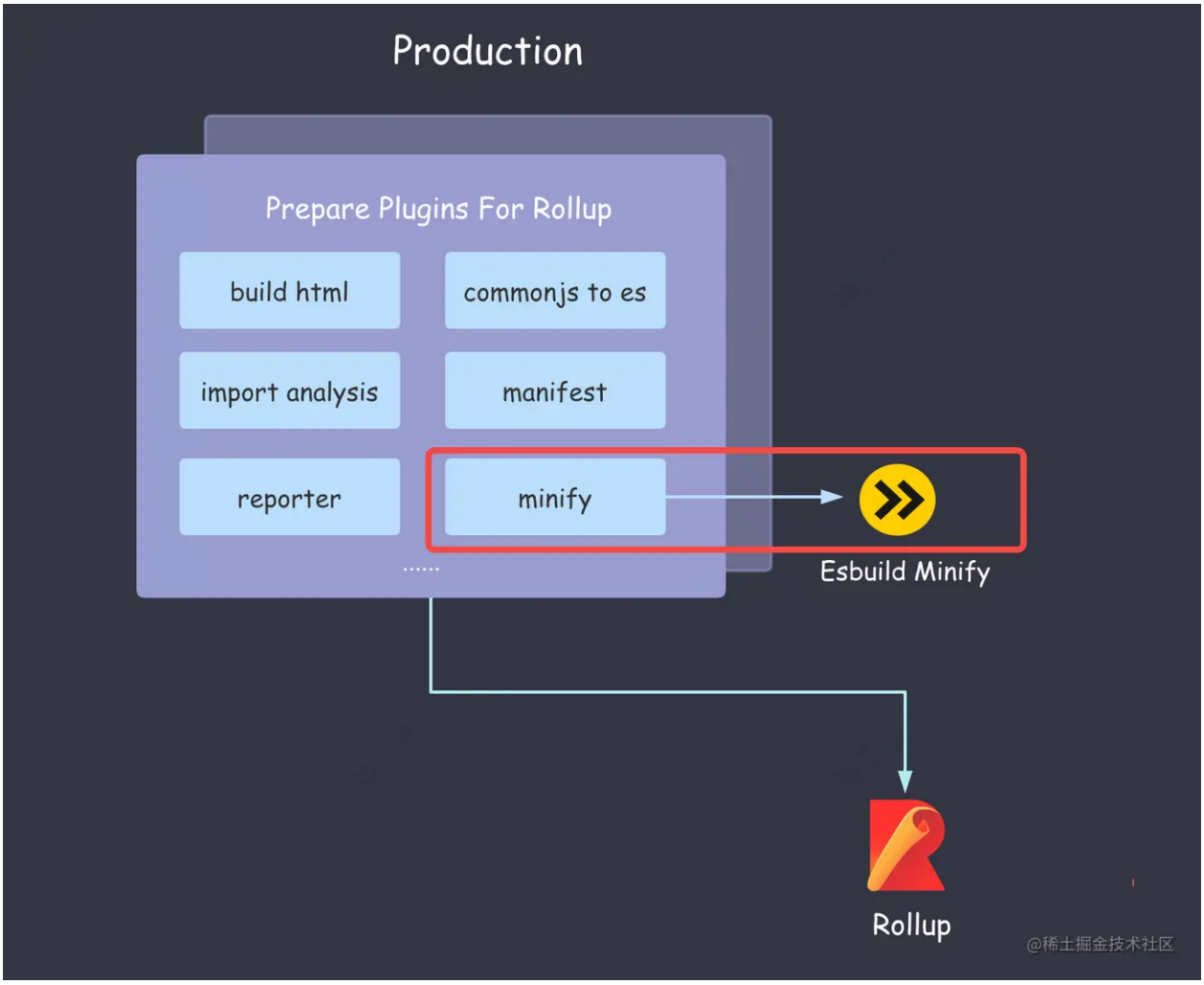
当然，要解决类型问题，我更推荐大家使用 TS 的编辑器插件。在开发阶段就能早早把问题暴露出来并解决，不至于等到项目要打包上线的时候。

三、代码压缩——作为压缩工具

> Vite 从 2.6 版本开始，就官宣默认使用 Esbuild 来进行生产环境的代码压缩，包

> 括 JS 代码和 CSS 代码。

从架构图中可以看到，在生产环境中 Esbuild 压缩器通过插件的形式融入到了 Rollup 的打包流程中:



那为什么 Vite 要将 Esbuild 作为生产环境下默认的压缩工具呢？因为压缩效率实在太高了！

传统的方式都是使用 Terser 这种 JS 开发的压缩器来实现，在 Webpack 或者 Rollup 中作为一个 Plugin 来完成代码打包后的压缩混淆的工作。但 Terser 其实很慢，主要有 2 个原因。

> 压缩这项工作涉及大量 AST 操作，并且在传统的构建流程中，AST 在各个工具之间无法共享，比如 Terser 就无法与 Babel 共享同一个 AST，造成了很多重复解析的过程。
> JS 本身属于解释性 + JIT（即时编译） 的语言，对于压缩这种 CPU 密集型的工作，其性能远远比不上 Golang 这种原生语言。

因此，Esbuild 这种从头到尾**共享 AST** 以及**原生语言编写**的 Minifier 在性能上能够甩开传统工具的好几十倍。

传统工具的好几十倍。

举个例子，我们可以看下面这个实际大型库(`echarts`)的压缩性能测试项目:

Artifact	Original size	Gzip size
echarts v5.1.1 (Source)	3.20 MB	689.67 kB

Minifier	Minified size	Minzipped size	Time
terser	🏆-69% **1.00 MB**	🏆-53% **322.12 kB**	*24x* 8,798 ms
terser.no-compress	-66% 1.07 MB	-52% 330.73 kB	*11x* 4,027 ms
esbuild	-68% 1.01 MB	-52% 331.66 kB	🏆 **361 ms**
uglify-js.no-compress	-67% 1.07 MB	-52% 331.66 kB	*7x* 2,709 ms
babel-minify *Timed out*	—	—	—
google-closure-compiler.simple *Timed out*	—	—	—
swc *Invalid output: SyntaxError*	—	—	—
uglify-js *Timed out*	—	—	—

压缩一个大小为 `3.2 MB` 的库，Terser 需要耗费 `8798 ms` ，而 Esbuild 仅仅需要 `361 ms` ，压缩效率较 Terser 提升了二三十倍，并且产物的体积几乎没有劣化，因此 Vite 果断将其内置为默认的压缩方案。

总的来说，Vite 将 Esbuild 作为自己的性能利器，将 Esbuild 各个垂直方向的能力(`Bundler` 、 `Transformer` 、 `Minifier`)利用的淋漓尽致，给 Vite 的高性能提供了有利的保证。

构建基石——Rollup

Rollup 在 Vite 中的重要性一点也不亚于 Esbuild，它既是 Vite 用作生产环境打包的核心工具，也直接决定了 Vite 插件机制的设计。那么，Vite 到底基于 Rollup 做了哪些事情?

生产环境 Bundle

虽然 ESM 已经得到众多浏览器的原生支持，但生产环境做到完全 `no-bundle` 也不行，会有网络性能问题。为了在生产环境中也能取得优秀的产物性能，Vite 默认选择在生产环境中利用 `Rollup` 打包，并基于 Rollup 本身成熟的打包能力进行扩展和优化，主要包含 3 个方面:

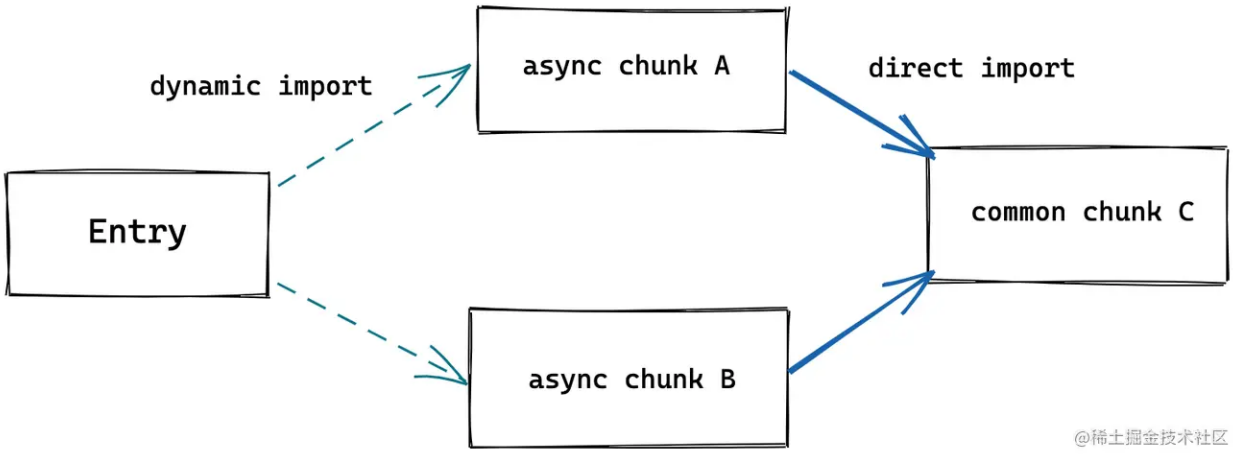
CSS 代码分割。如果某个异步模块中引入了一些 CSS 代码，Vite 就会自动将这些 CSS 抽取出来生成单独的文件，提高线上产物的 缓存复用率 。

自动预加载。Vite 会自动为入口 chunk 的依赖自动生成预加载标签 `<link rel="moduelpreload">` ，如:

```
<head>
  <!-- 省略其它内容 -->
  <!-- 入口 chunk -->
  <script type="module" crossorigin src="/assets/index.250e0340.js"></script>
  <!--  自动预加载入口 chunk 所依赖的 chunk-->
  <link rel="modulepreload" href="/assets/vendor.293dca09.js">
</head>
```

这种适当预加载的做法会让浏览器提前下载好资源，优化页面性能。

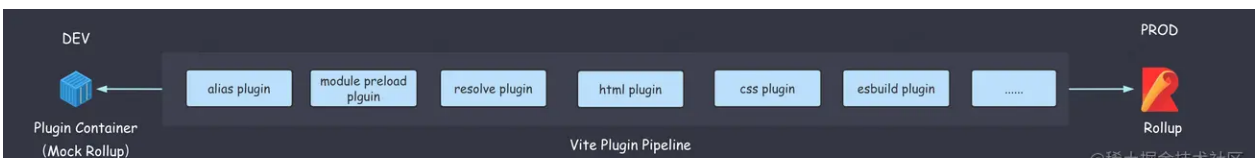
异步 Chunk 加载优化。在异步引入的 Chunk 中，通常会有一些公用的模块，如现有两个异步引入的 Chunk: `A` 和 `B` ，而且两者有一个公共依赖 C，如下图:



一般情况下，Rollup 打包之后，会先请求 A，然后浏览器在加载 A 的过程中才决定请求和加载 C，但 Vite 进行优化之后，请求 A 的同时会自动预加载 C，通过优化 Rollup 产物依赖加载方式节省了不必要的网络开销。

兼容插件机制

无论是开发阶段还是生产环境，Vite 都根植于 Rollup 的插件机制和生态，如下面的架构图所示:



在开发阶段，Vite 借鉴了 WMR 的思路，自己实现了一个 `Plugin Container`，用来模拟 Rollup 调度各个 Vite 插件的执行逻辑，而 Vite 的插件写法完全兼容 Rollup，因此在生产环境中将所有的 Vite 插件传入 Rollup 也没有问题。

反过来说，Rollup 插件却不一定能完全兼容 Vite(这部分我们会在**插件开发**小节展开来说)。不过，目前仍然有不少 Rollup 插件可以直接复用到 Vite 中，你可以通过这个站点查看所有兼容 Vite 的 Rollup 插件: vite-rollup-plugins.patak.dev/ 。

狼叔在《以框架定位论前端的先进性》提到现代前端框架的几大分类，Vite 属于 人有我优 的类型，因为类似的工具之前有 Snowpack，Vite 诞生之后补齐了作为一个 no-bundle 构建工具的 Dev Server 能力(如 HMR)，确实比现有的工具能力更优。但更重要的是，Vite 在**社区生态**方面比 Snowpack 更占先天优势。

Snowpack 自研了一套插件机制，类似 Rollup 的 Hook 机制，可以看出借鉴了 Rollup 的插件机制，但并不能兼容任何现有的打包工具。如果需要打包，只能调用其它打包工具的 API，自身不提供打包能力。

而 Vite 的做法是从头到尾根植于的 Rollup 的生态，设计了和 Rollup 非常吻合的插件机制，而 Rollup 作为一个非常成熟的打包方案，从诞生至今已经迭代了 六年多 的时间，npm 年下载量达到 上亿次 ，产物质量和稳定性都经历过大规模的验证。某种程度上说，这种根植于已有成熟工具的思路也能打消或者降低用户内心的疑虑，更有利于工具的推广和发展。

小结

本小节的内容中，我给你拆解了 Vite 底层双引擎的架构，分别介绍了 Esbuild 和 Rollup 究竟在 Vite 中做些了什么，你需要重点掌握 **Vite 的整体架构**以及 **Esbuild 和 Rollup 在 Vite 中的作用**。

首先，Esbuild 作为构建的性能利器，Vite 利用其 Bundler 的功能进行依赖预构建，用其 Transformer 的能力进行 TS 和 JSX 文件的转译，也用到它的压缩能力进行 JS 和 CSS 代码的压缩。

接着，我给你介绍了 Vite 和 Rollup 的关系。在 Vite 当中，无论是插件机制、还是底层的打包手段，都基于 Rollup 来实现，可以说 Vite 是对于 Rollup 一种场景化的深度扩展，将 Rollup 从传统的 JS 库打包场景扩展至完整 Web 应用打包，然后结合开发阶段 `no-bundle` 的核心竞争力，打造出了自己独具一格的技术品牌。

因此，你可以看出双引擎对于 Vite 的重要性，如果要深入学习和应用 Vite，那么掌握 Esbuild 和 Rollup 的基础使用和插件开发是非常有必要的。在下面的几个小节中，我们将一起进入双引擎本身的学习。

上一篇：预构建: 如何玩转秒级依赖预构建的能力?

下一篇：得力的性能推手: Esbuild 功能使用与插件开发实战