

前端打包构建

从底层原理到工程实践

一本由真实互动教学沉淀而成的技术电子书

ebook.leoyim.cn

版本 1.0.0 · 导出于 2026-10-12

目录

前言

本书简介

完整目录

正文

第 1 章 · 前端构建基础

第 2 章 · 核心构建原理

第 3 章 · Webpack 与 Vite

第 4 章 · 新一代构建工具

第 5 章 · 构建性能优化

第 6 章 · AI 辅助构建工程

第 7 章 · 工程实践与选型

实践

示例代码

决策

工具选型决策记录

附录

术语表

配置参考

参考资料

前端打包构建：从底层原理到工程实践

一本由真实互动教学沉淀而成的技术电子书。

这本书是什么

2026 年 10 月 8 日至 10 日，一位有 HTML/CSS/JS 与 npm 基础的开发者，以「资深前端工程化专家」为对话对象，用三天的分节互动教学走完了四个模块、二十二节内容，主题是**前端打包构建工具**：

1. **核心原理** —— 依赖图、入口与输出、Loader/Plugin、Tree Shaking、代码分割、HMR、Source Map、开发与生产模式差异
2. **Webpack 与 Vite** —— 打包式构建与原生 ESM 开发服务器的架构分野
3. **新一代构建工具** —— esbuild、SWC、Rspack、Farm、Turbopack、Rolldown，以及 AI 辅助构建
4. **学习路径** —— 分阶段顺序、动手项目、文档与源码的阅读切入点

本书把这三天的讲解、追问、批改与纠错，重组成一本**可以独立阅读、系统复习、实践验证**的知识读物。

它不是聊天记录。对话被按知识依赖重新组织，零散解释被合并成完整讲解，重复内容被去重。**唯一被刻意保留的“过程痕迹”，是那些有学习价值的反例、认知误区和纠错过程**——它们进了各章的「常见误区」。

怎么读

你的情况	建议路径
想建立完整心智模型	按顺序读 01 → 07
只想知道 Webpack 和 Vite 怎么选	03 章 → 07 章 §7.5 决策矩阵
手上有老项目想提速	04 章 §4.3 (Rspack) → 07 章 §7.6 (迁移评估)
想复习关键结论	<code>appendix/glossary.md</code>
想看踩过哪些坑	各章「常见误区」

给谁看

- 前端开发者中**已经会用构建工具**、但说不清「为什么」的人
- 需要为团队做**构建工具选型与迁移决策**的人
- 想理解 Rust/Go 构建工具这一轮换代，在发生什么、该不该跟进的人
- 需要一份**可复用的技术学习方法论**（而不只是知识点）的人

前提：会写 HTML/CSS/JS，装过 npm 包，跑过 `npm run dev` / `npm run build`。

维护说明

- 全书为纯 Markdown，无构建步骤，可直接在任何 Markdown 阅读器 / 静态站点生成器中渲染。
- 目录索引见 `SUMMARY.md`。
- **增量更新方式**：新增内容按章追加；涉及工具版本、成熟度、性能数字时，先核验官方文档再改，改动处标注核验日期。

目录

正文

1. 第 1 章 · 前端构建基础

- 1.1 从无模块到 ESM：浏览器加载代码的三次进化
- 1.2 构建工具究竟做了什么
- 1.3 依赖管理与 npm 的角色
- 1.4 开发构建与生产构建：两个目标，两套策略
- 1.5 全书知识地图

2. 第 2 章 · 核心构建原理

- 2.1 依赖图：构建器的世界模型
- 2.2 入口与输出：从图到 chunk 到文件
- 2.3 Loader 与 Plugin：单文件转换与全流程干预
- 2.4 Tree Shaking：两个执行者的接力
- 2.5 代码分割：三种触发方式与一条分界线
- 2.6 HMR 热更新：一条从文件改动到模块替换的链路
- 2.7 Source Map：调试器与产物之间的反向映射
- 2.8 开发模式与生产模式：差异全景与 `mode` 的边界
- 2.9 本章常见误区（12 条）

3. 第 3 章 · Webpack 与 Vite

- 3.1 架构分野：打包式构建 vs 原生 ESM
- 3.2 依赖预构建：Vite dev 专属的两件事
- 3.3 配置方式对比：默认值覆盖了哪一层
- 3.4 环境变量机制：一个被构建期替换"伪造"出的对象

- 3.5 插件生态对比：一套概念还是两套
- 3.6 速度对比：把"快"拆成四个指标
- 3.7 本章常见误区（12 条）

4. 第 4 章 · 新一代构建工具

- 4.1 为什么原生化：并行度决定难度
- 4.2 esbuild 与 SWC：转译与压缩层
- 4.3 Rspack 与 Farm：借别人的生态
- 4.4 Turbopack 与 Rolldown：另外两种打法
- 4.5 成熟度矩阵与投入建议
- 4.6 本章常见误区（14 条）

5. 第 5 章 · 构建性能优化

- 5.1 优化的前提：先有指标，再有结论
- 5.2 三类耗时来源与各自的优化手段
- 5.3 缓存：三层结构与各自的失效条件
- 5.4 产物层面的优化：体积与结构
- 5.5 并行与增量：原生工具与增量引擎
- 5.6 优化验证：六步测量法
- 5.7 本章常见误区

6. 第 6 章 · AI 辅助构建工程

- 6.1 视角转换：AI 成了构建工具的用户
- 6.2 三层价值：从配置助手到工具使用者
- 6.3 实践形态：结构化诊断与运行时感知
- 6.4 风险与边界
- 6.5 本章常见误区（与术语澄清）

7. 第 7 章 · 工程实践与选型

- 7.1 顺序按能力排，不按工具排

- 7.2 五个台阶与判定标准
- 7.3 每阶段的动手项目
- 7.4 文档与源码的阅读切入点
- 7.5 选型决策矩阵
- 7.6 迁移评估：一页纸模板
- 7.7 本章常见误区（18 条）

实践

- [示例代码](#) —— 配置示例 (`webpack.config.js` / `vite.config.ts`)

决策

- [工具选型决策记录](#)

附录

- [术语表](#)
- [配置参考](#)
- [参考资料](#)

第 1 章 · 前端构建基础

1.1 从无模块到 ESM：浏览器加载代码的三次进化

要理解构建工具为什么存在，得先知道浏览器手上的牌有多难打。

第一阶段：没有模块。 早期页面把 `<script>` 一个接一个贴上，所有变量挤在全局作用域里。需要"模块"就只能靠 IIFE 和命名空间对象自己模拟。加载顺序靠人肉维护，依赖关系不在代码里。

第二阶段：模块出现了，但浏览器不认。 CommonJS (Node 生态) 用 `require`，AMD 用 `define`，本质上都是运行时解析依赖——浏览器要先拿到文件、执行到那一行，才知道还要再下载什么。这导致两个固有问题：请求是串行的、依赖图是运行时才建出来的。

第三阶段：ESM 成为标准。 `import` / `export` 是静态语法——浏览器在解析阶段就能扫出依赖，不必执行代码。这是决定性的变化：依赖图第一次可以在"不运行代码"的前提下被算出来。

正是这条性质，支撑了后面几乎所有优化：Tree Shaking 需要静态分析才能标记未使用的导出；Vite 的按需转换需要知道模块之间的引用关系；代码分割需要知道哪里是图的断点。

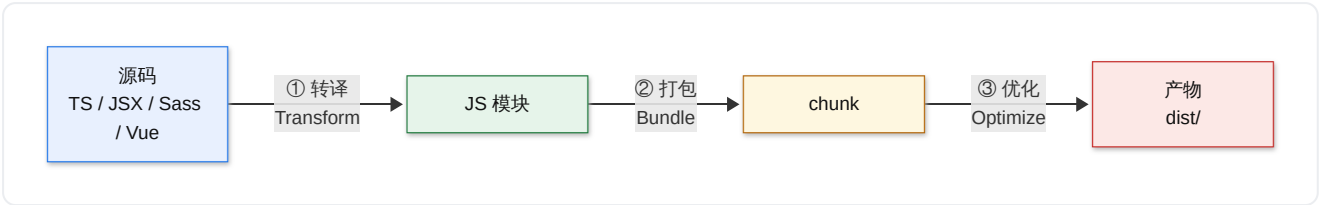
但 ESM 也不解决全部问题，它留下了三件事给人：

问题	为什么 ESM 解决不了
老语法 / 非 JS 语言	浏览器只认 JS。TS、JSX、Sass、Vue SFC 都需要先转成 JS
兼容旧浏览器	原生 ESM 是较新的能力，老浏览器不支持
请求数量	一个依赖几百个内部模块的库（如 <code>lodash-es</code> ），按原生 ESM 加载就是几百个请求

这三件事合起来，就是构建工具的生存空间。

1.2 构建工具究竟做了什么

剥掉所有配置项和插件名，构建工具只做三件事：



环节	做什么	常见执行者
① 转译	把浏览器不认的东西变成浏览器认的	Babel / tsc → SWC / esbuild
② 打包	沿着依赖图，把散落的模块合成若干 chunk	Webpack / Rollup / Rspack / Rolldown
③ 优化	压缩、混淆、摇树、分割、打 hash	Terser / SWC minify + 打包器自身

关键认知：这三件事不是绑在一起的。它们可以分别替换 —— 这个认知是第 4 章理解"原来可以只换一层"的前提。

为什么值得强调：大多数人对构建工具的印象是"Webpack 或者 Vite"，把它们当成两个不可分割的整体。实际上它们是一条流水线的不同段位，而这条流水线的每一段，现在都有原生实现可以替换。

1.3 依赖管理与 npm 的角色

构建工具处理的"依赖"，绝大多数来自 npm。

概念	作用	与构建的关系
<code>package.json</code>	声明依赖版本范围、脚本、以及 <code>sideEffects</code> / <code>main</code> / <code>module</code> / <code>exports</code> 等字段	构建工具会读它 ： <code>sideEffects</code> 决定 Tree Shaking 的激进度； <code>exports</code> 决定 <code>resolve</code> 能解析到什么
<code>package-lock.json</code> / <code>pnpm-lock.yaml</code>	锁定实际安装的精确版本	影响缓存失效判断（Vite 依赖预构建的缓存键包含 lockfile）
<code>node_modules</code>	实际安装位置	<code>resolve</code> 的默认查找路径

三个字段值得单独记住（后文都会用到）：

- **`sideEffects`** —— 告诉打包器“这个模块有没有副作用”。写 `false` 表示可以安全跳过未被引用的模块。它是库作者写的，不是消费方能改的。
- **`exports`** —— 决定哪些路径可以被 `import`、以及在不同环境（`import` / `require` / `browser`）下分别指向哪个文件。**配错会导致“解析得到文件 A，运行时期待文件 B”。**
- **`main` / `module` / `types`** —— 三个入口字段，分别给 CJS 消费者、ESM 消费者、TypeScript 消费者。**发布组件库时必须手动对齐**（见第 7 章 §7.5）。

版本范围与它的代价：`^1.2.3` 允许自动升到 `1.x` 的最新版。好处是能吃到修 bug，代价是同一份 **lockfile** 之外的两台机器可能装到不同版本 → 构建结果不一致。这是“本地能跑、CI 挂了”的一类常见来源。

1.4 开发构建与生产构建：两个目标，两套策略

这是课程中讲过的一节（模块 1 第 8 节），也是理解后面所有“为什么 dev 一套、build 另一套”的总开关。

核心事实：开发和生产的目标是冲突的。

维度	开发模式	生产模式
目标	构建快、反馈快	产物小、加载快
压缩混淆	不做	Terser / SWC minify
Tree Shaking	不启用	启用
Source Map	<code>eval-cheap-*</code> 快而糙	<code>hidden</code> 慢而准
代码分割	不抽公共包	<code>splitChunks</code> / <code>manualChunks</code>
HMR	启用	无
<code>NODE_ENV</code>	<code>development</code>	<code>production</code>

开发模式愿意牺牲产物质量换速度 —— 所以不压缩、不抽公共包、source map 只求"能定位到行"。 生产模式愿意牺牲构建时长换产物质量 —— 所以压缩、摇树、分割、算 hash 全上。

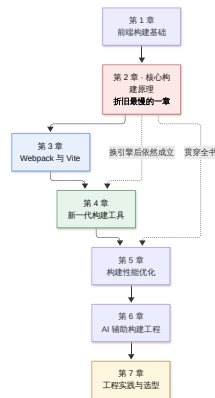
这条冲突解释了很多"奇怪"现象：

- 为什么本地跑得好好的，build 之后行为变了？→ 两套模式下代码走了不同的优化路径（第 4 章 §4.4 的 dev/prod parity 问题就是它的直接后果）。
- 为什么 `console.log` 生产环境还在？→ `mode` 不负责删 **console**，要自己配 `terserOptions.compress.drop_console`。
- 为什么开发时不报类型错误？→ 转译器只剥离类型标注，不做类型检查（第 4 章 §4.2）。

两个模式之间共享的机制，只有一个，但它极其重要 —— 构建期字面量替换。

`process.env.NODE_ENV` 不是运行时的环境变量，而是被构建工具在编译期整体替换成字符串，然后在生产模式下由压缩器把恒假分支物理删除。详细机制在 §2.8。

1.5 全书知识地图



读这张图的方式：

- 第 2 章是**唯一不会被淘汰的部分**。依赖图、Loader/Plugin 分工、Tree Shaking 两阶段、代码分割三策略、dev/prod 差异 —— 这些在 Rspack、Rolldown、Turbopack 里**全都还在**，只是换了执行者。
- 第 3 章是**当下最实用的对比**（Webpack 与 Vite 仍占据绝大多数存量项目）。
- 第 4 章是**正在发生的变化**，折旧最快 —— 背配置的收益极低，理解分层替换的收益极高。
- 第 5 章是**把知识变成结论的能力**（会算账）。
- 第 7 章是**把结论变成决策的能力**（会选型）。

一条贯穿全书的判据：任何技术结论，先问它绑定了什么指标、什么前提。这句话在第 3 章讲"Vite 更快"时会用一次，在第 5 章讲基准测量时会用一次，在学习进度记账时还会用一次 —— 它是同一件事。

第 2 章 · 核心构建原理

主线索：

源码 → 依赖图 → chunk → 产物

其余所有能力，都是挂在这条主线上的分支装置。

2.1 依赖图：构建器的世界模型

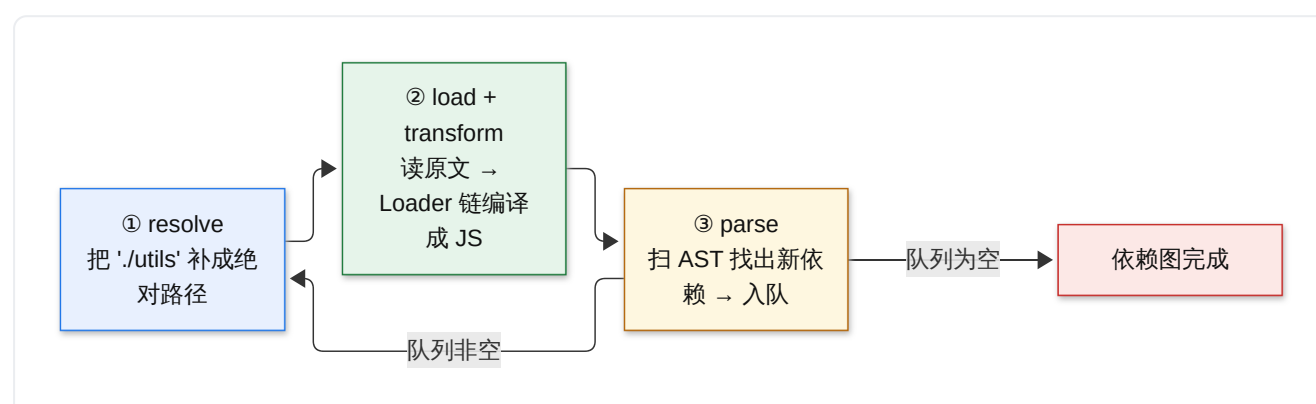
定义

依赖图是一张有向图：

- 节点 = 模块
- 边 = `import` / `require` / 动态 `import()`
- 根 = 入口

它怎么被建出来

构建器从入口出发，循环执行三个动作，直到待处理队列为空：



动作	做什么	受哪些配置影响
resolve	把 <code>./utils</code> 补成绝对路径	<code>resolve.extensions</code> 、 <code>resolve.alias</code> 、 <code>package.json</code> 的 <code>exports</code>
load + transform	读原文，交给 Loader 链编译成 JS	<code>module.rules</code> (Loader 配置)
parse	扫 AST，找出新依赖并入队	代码本身的语法是否可静态分析（见 §2.4 的 ESM 前提）

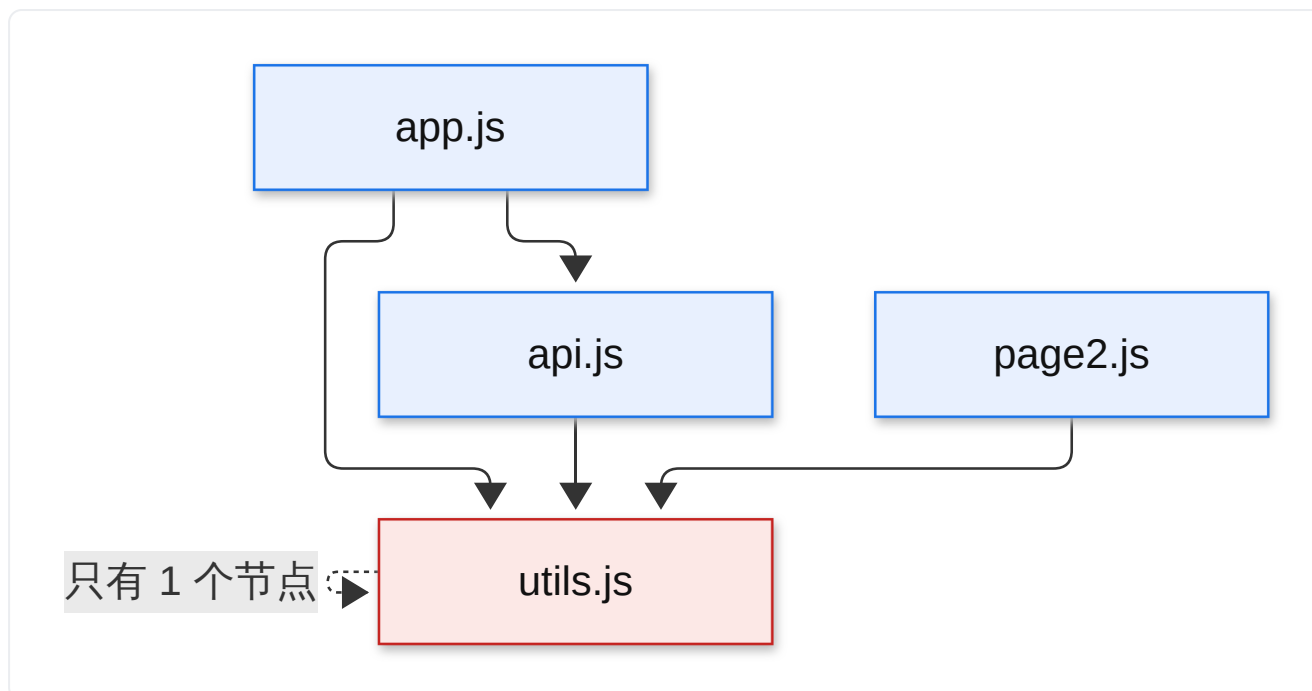
节点里存什么

节点存三样东西：**绝对路径 + 转换后代码 + 依赖列表**。

- **绝对路径用于去重** —— 同一个文件被两条路径引用到，路径归一化后才能认出是同一个模块。
- **代码内容用于输出** —— 去重后只保留一份。

图 ≠ 树

这是最容易误解的一点。



`utils.js` 被三条路径引用到，但在图里只存在一个节点。这就是「模块单例」语义的来源——模块代码只执行一次。

但这里有一个关键区分，很多人会混：

层级	谁决定	结果
内存依赖图	图遍历时去重	共享模块只有 1 个节点
产物文件	<code>splitChunks</code> 配置	配了才抽公共包，否则各入口各存一份

易错点：以为"模块是单例，所以产物里不会重复"。**错。** 图去重和产物去重是**两层的事**。图去重只管内存，产物是否重复由 `splitChunks` 决定。

常见的坑

- 以为图是树：把每个 `import` 都当父子关系，于是理解不了"为什么共享模块只执行一次"。
- 以为路径字符串相同就是同一模块：`./utils` 和 `./utils.js` 在 `resolve` 之前是两个不同的字符串。
- 忽略 `exports` 字段的影响：`package.json` 的 `exports` 会限制哪些子路径能被解析到（这也是一类"装了包但 `import` 报错"的来源）。

2.2 入口与输出：从图到 chunk 到文件

入口

- `entry` 是依赖图的起点。
- 对象写法时，**key** 就是 `[name]`。
- 多入口必须配 `[name]`，否则产物会互相覆盖（都叫 `main.js`）。

```
module.exports = {
  entry: {
    app: './src/app.js',
    admin: './src/admin.js',
  },
  output: {
    filename: '[name].[contenthash:8].js', // 不加 [name] → 互相覆盖
    path: path.resolve(__dirname, 'dist'), // 必须绝对路径
    clean: true, // 防旧产物堆积
  },
};
```

两个容易踩的点：

- `output.path` 必须绝对路径。写成相对路径的行为不可靠（相对于什么，`process.cwd()` 在多包仓库里是变的）。
- `clean: true` 很值得开。不开的话，改了文件名之后旧文件会一直堆在 `dist/` 里——本地产物分析会被污染，部署时也可能把过期文件传上去。

chunk：产物里的文件单位

chunk = module 的打包容器。1 个 chunk ≈ 产物里 1 个 `.js` 文件。

chunk 的三个来源：

来源	触发条件
<code>entry</code>	每个入口生成一个入口 chunk
动态 <code>import()</code>	每个动态导入点生成一个懒加载 chunk
<code>splitChunks</code>	满足抽包条件的公共模块，合并成一个公共 chunk

为什么这个三分法重要：读产物时看到多出来一个 js 文件，第一个问题就是“它属于哪一类来源”——这决定了它是谁生成的、谁在什么时候加载它。这是第 7 章「台阶 3·会读产物」的第一步动作。

内容指纹：三种 hash

hash 类型	计算范围	评价
<code>[hash]</code>	整个构建	禁用 —— 任何改动导致所有文件失效，缓存全废
<code>[chunkhash]</code>	单个 chunk	易被运行时胶水代码污染 —— chunk 内容没改，但运行时片段变了，hash 就变了
<code>[contenthash]</code>	该文件的最终内容	生产首选

chunkhash 被污染是怎么回事：一个 chunk 里除了你自己的模块代码，还包含 Webpack 注入的运行时胶水（模块注册表、`__webpack_require__` 之类）。你的代码一行没动，但另一个 chunk 变化导致模块 ID 分配变化 → 胶水变了 → `chunkhash` 变了 → 缓存白白失效。

contenthash 按文件最终内容计算，天然免疫这个问题。

缓存三件套

只改 `filename` 用 `contenthash` 还不够，三个配置要一起上：

```
module.exports = {
  output: {
    filename: '[name].[contenthash:8].js',
    chunkFilename: '[name].[contenthash:8].chunk.js',
  },
  optimization: {
    moduleIds: 'deterministic', // ① 模块 ID 用确定性算法，不随引用顺序变
    runtimeChunk: 'single', // ② 把运行时胶水单独抽出来
    splitChunks: { chunks: 'all' },
  },
};
```

配置	缺了会怎样
contenthash	内容没变文件名也变
runtimeChunk: 'single'	运行时胶水留在各 chunk 里，一改就污染别人
moduleIds: 'deterministic'	模块 ID 按引用顺序分配，加个 import 别人就全变

判据（第 7 章台阶 3 的核心）：改一行代码，你能预测哪些产物的 **hash** 会变吗？改不了的三件套，一定会出现“只改了一个按钮文案，用户却要重新下载整个 vendor”。**hash 稳定性不是玄学，是这三项配齐的必然结果。**

splitChunks 的要点

```
optimization: {
  splitChunks: {
    chunks: 'all', // 默认只处理 async 的，写 all 才管同步和异步
    minChunks: 2, // 被 2 个以上 chunk 引用才抽
    cacheGroups: { /* 自定义分组 */ },
  },
}
```

- **minChunks** 数的是“引用它的 **chunk 数量**”，不是“模块数量”。这是常被读错的一个参数。
- **Webpack 5 默认 chunks: 'async'** —— 意思是默认只对动态 **import** 产生的 **chunk** 做抽包，同步入口之间共享的模块不会自动抽。所以“抽公共包”不是打包器的默认行为，是你要显式开启的。

常见的坑

坑	后果	正确做法
多入口没配 <code>[name]</code>	产物互相覆盖	文件名模板必带 <code>[name]</code>
<code>output.path</code> 写相对路径	行为不确定	用 <code>path.resolve(__dirname, ...)</code>
用 <code>[hash]</code>	缓存全废	用 <code>[contenthash]</code>
以为默认会抽公共包	各入口各存一份	显式 <code>chunks: 'all'</code>
<code>minChunks</code> 理解成模块数量	抽包阈值设错	它是引用 chunk 数
没开 <code>clean</code>	旧产物堆积污染分析	<code>output.clean: true</code>

2.3 Loader 与 Plugin：单文件转换与全流程干预

这是 Webpack 体系里最基础、也最容易判反的一对概念。

对照表

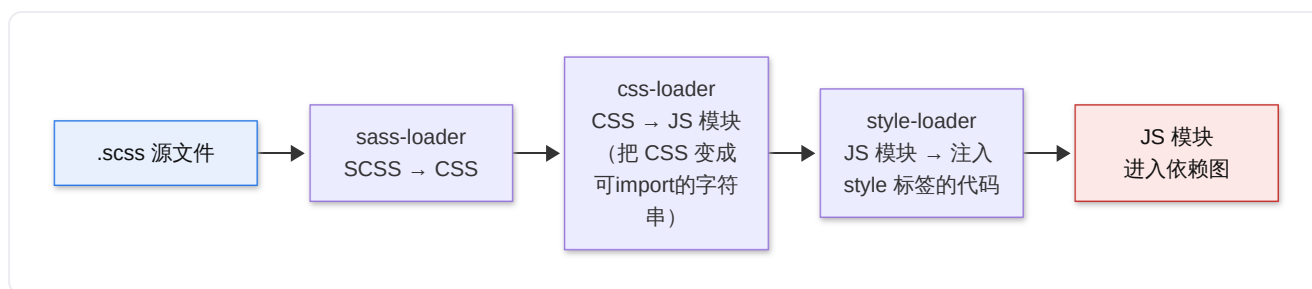
维度	Loader	Plugin
作用范围	单个模块（编译阶段内）	整个构建流程（初始化 / 编译 / 生成 / 输出四阶段）
解决什么	格式转换（ts / sass / vue / 图片）	流程干预（抽 CSS、压缩、生成 HTML）
配置形式	<code>module.rules</code> 数组	<code>plugins</code> 数组， <code>new XxxPlugin()</code>
能否操作产物	不能	能 —— 可生成 / 修改 / 删除最终文件
本质	一个「输入代码 → 输出代码」的函数	一个带 <code>apply(compiler)</code> 的类实例

Loader：一条类型转换管道

执行顺序：从右到左、从下到上。

```
module: {  
  rules: [  
    {  
      test: /\.scss$/,  
      use: ['style-loader', 'css-loader', 'sass-loader'],  
      //           ↑ 第 3 个执行    ↑ 第 2 个执行    ↑ 第 1 个执行  
    },  
  ],  
}
```

实际执行顺序是 `sass-loader` → `css-loader` → `style-loader`：



为什么是从右到左：把它理解成函数组合 —— `style(css(sass(file)))`，最内层的先执行。数组合写法是"书写顺序"而不是"执行顺序"。

本质是类型转换管道 —— 每一步的输出必须匹配下一步的输入。顺序写反了不是"配置语法错"，而是喂错了类型：`sass-loader` 拿到 CSS 会解析失败，或者 `css-loader` 拿到 SCSS 会产出错误结果。

Plugin：往构建流程里插钩子

```
class MyPlugin {
  apply(compiler) {
    compiler.hooks.emit.tap('MyPlugin', (compilation) => {
      // compilation.assets 里是即将写出的产物，可以生成/修改/删除
    });
  }
}
```

- 插件是一个类，通过 `apply(compiler)` 拿到 `compiler` 对象，再往 `compiler.hooks.xxx` 上 `tap()` 注册回调。
- 可以介入四个阶段：初始化 → 编译 → 生成 → 输出。
- 能碰产物 —— 这是它和 Loader 最本质的能力差异。

判别口诀

处理「单个源文件的内容」→ **Loader** 涉及「构建阶段、跨文件、产物编排」→ **Plugin**

用"流程动作"判断会出错。比如"注入 style 标签"，看起来是个流程动作，但它作用在单个 **CSS 模块**上，所以 `style-loader` 是 Loader。而"生成 manifest"要遍历所有 chunk、输出一个新文件，所以 `WebpackManifestPlugin` 是 Plugin。

判据是「它处理的是谁」，不是「它看起来在干什么」。

常见的坑

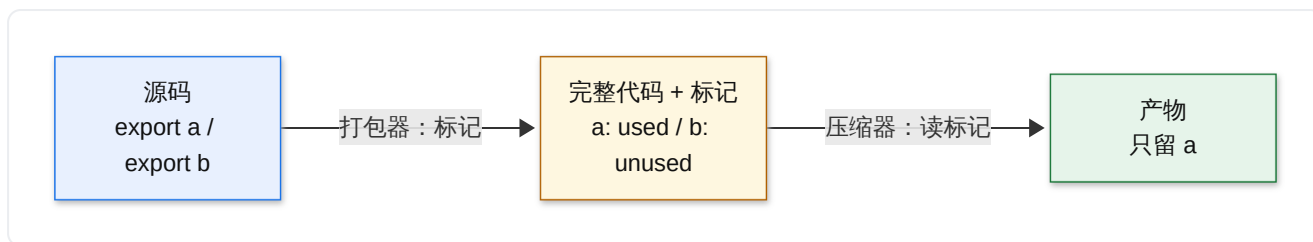
坑	原因
Loader 顺序写反	理解为"书写顺序"而不是"管道方向"
<code>test</code> 正则过窄导致漏转	例如只写 <code>/\.ts\$/</code> 漏掉了 <code>.tsx</code>
Plugin 忘了 <code>new</code>	直接写 <code>MyPlugin()</code> 或 <code>MyPlugin</code> 都不对，要 <code>new MyPlugin()</code>
想用 Plugin 改单个文件内容	那是 Loader 的活；Plugin 该做的是产物编排
以为 Webpack 插件能迁到 Vite	钩子体系不同（Tapable vs Rollup），搬不过去，见 §3.5

2.4 Tree Shaking：两个执行者的接力

两阶段，缺谁都白搭

很多人以为 Tree Shaking 是"打包器的一个开关"。实际上它是两个执行者的接力：

阶段	谁执行	干什么
① 标记	打包器（Rollup 内置 / Webpack <code>optimization.usedExports</code> ）	给未使用的导出打上 <code>unused harmony</code> 标记，但代码仍完整保留在产物里
② 删除	压缩器（Terser / SWC minify）	读取标记，物理删除未使用代码并混淆



这个"两阶段"的认知价值在于排错方向：

- 代码摇不掉 → 先看是标记阶段没生效，还是删除阶段没运行。
 - 标记没生效：不是 ESM？不是生产模式？`usedExports` 没开？
 - 删除没运行：压缩器被关了 (`minimize: false`)、用了不读标记的压缩器。

三个概念的粒度对照

概念	粒度	谁提供
<code>usedExports</code>	导出的成员	打包器（自动）
<code>sideEffects</code>	整个模块能否跳过	你，在 <code>package.json</code> 里手写
Terser / SWC	单文件 AST	压缩器

`sideEffects` 的语义

```
{  
  "sideEffects": false  
}
```

意思是：未被引用的模块可以整体跳过（不进产物）；有引用则照常执行。

例外必须写数组（否则会误删）：

```
{  
  "sideEffects": [ "*.css", "./src/polyfill.js" ]  
}
```

CSS 文件和 polyfill 是典型的"被引入是为了产生副作用"的模块 —— 它们没有导出被使用，但删掉就坏。只写 `false` 不写例外，是这类 **bug** 的经典来源。

两个前提

1. 必须是 **ESM**。CJS 的 `require` 是运行时求值，静态分析不了 —— 打包器无法确定"这个导出有没有被用到"。
2. 生产模式才启用。开发模式要保证构建快，不做这层分析。

最有效的一招

改「导入方式」比改配置管用。

```
import { debounce } from 'lodash';           // ❌ CJS + 全量导入，摇不掉
import debounce from 'lodash-es/debounce'; // ✅ ESM + 按具体路径导入
```

为什么这比配 `sideEffects` 更有效：`sideEffects` 是库作者在自己 `package.json` 里写的字段，消费方改不了。消费方能控制的只有两件事 —— 从哪个路径导入、用什么语法导入。

一句话总结：库写得好不好占一半，你的导入姿势占另一半。

常见的坑

坑	正确认知
以为 Tree Shaking 一步完成	打包器标记 + 压缩器删除，两个执行者
在自己项目里配 <code>sideEffects</code> 想减小第三方库	第三方看库作者；你能做的是改导入路径
用了 CJS 库还期待摇树	<code>require</code> 静态分析不了
开发模式验证摇树效果	生产模式才启用
<code>sideEffects: false</code> 不写例外	CSS / polyfill 会被误删
以为 <code>import { x } from 'lodash'</code> 能摇掉	包本身是 CJS + 整包导入，摇不动

2.5 代码分割：三种触发方式与一条分界线

三种方式

触发方式	拆分时机	加载时机
① 多入口	构建期	页面级（不同 HTML 各引各的）
② 动态 <code>import()</code>	构建期分包	运行时按需（用户触发才下载）
③ <code>splitChunks</code>	构建期	随入口一起

核心分界线

只有 ② 能改变「运行时什么时候下载」。① 和 ③ 都是构建期静态拆分——产物变多了，但下载时机没变。

这条分界线的实用价值：当有人问“怎么优化首屏”时，只有 ② 是答案。① 和 ③ 优化的是“缓存复用”和“并行下载”，不是“首屏要不要下载”。

动态 import 的真实效果

```
const Chart = () => import('./Chart.vue');
button.onclick = async () => {
  const { default: C } = await import('./Chart.js');
  new C().render();
};
```

效果：减小首屏加载体积。不效果：不减小总产物体积。代码照常打包，只是变成了一个独立的 chunk，被推迟下载。

这句话是易错点的高发区——很多人说“用了动态 import，包变小了”。准确的表述是“首屏要下载的字节少了，总量没变”。

两层叠加技巧

把大依赖彻底隔离出首屏的做法：

1. 路由懒加载：`component: () => import('./views/Heavy.vue')`
2. 组件内再动态 **import**：重型依赖（图表库、编辑器）在真正需要时才 `await import()`

先按路由切一刀，再在首屏路由内把大依赖再切一刀。关键是大依赖不能出现在入口的静态 **import** 链上。

Vite 的差异

- Rollup 自动抽公共包，不需要配 `splitChunks`。

- 要自定义分组用 `build.rollupOptions.output.manualChunks`。

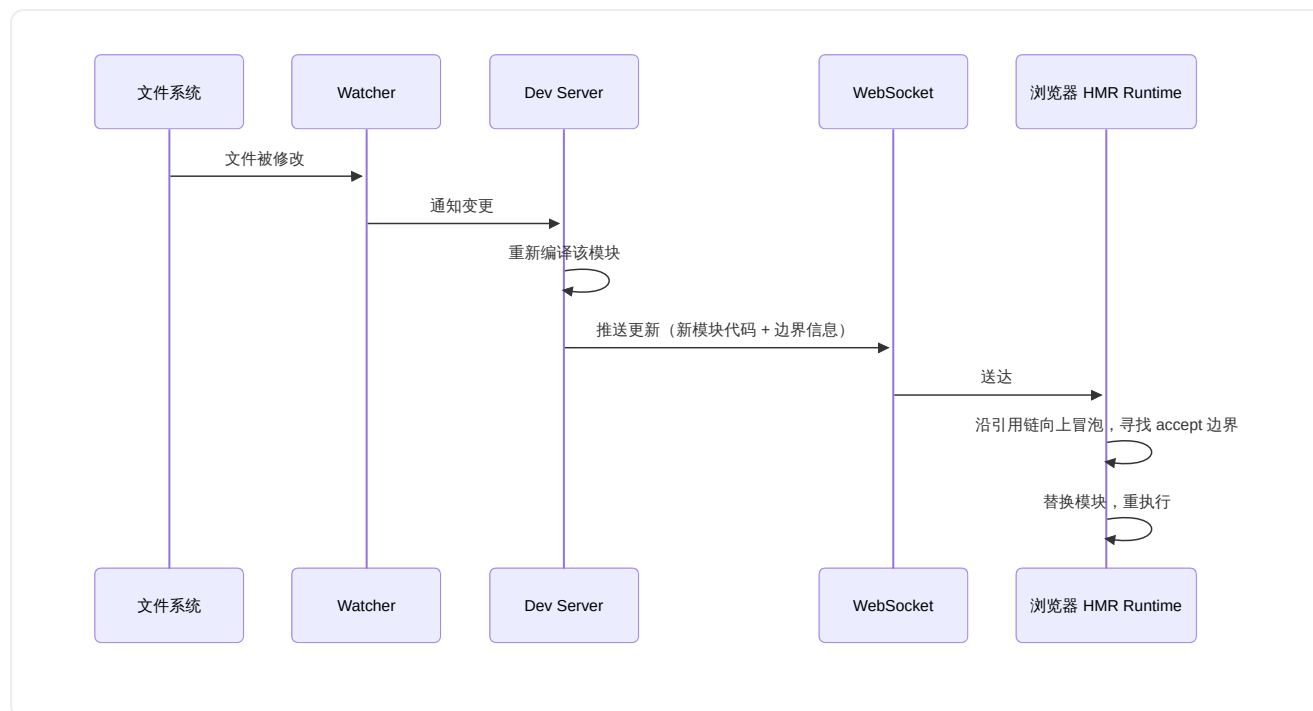
```
// vite.config.ts
export default {
  build: {
    rollupOptions: {
      output: {
        manualChunks: {
          'vendor-chart': ['echarts'],
          'vendor-ui': ['element-plus'],
        },
      },
    },
  },
};
```

常见的坑

坑	正确认知
以为动态 import 能减小总体积	只减小首屏体积
把 ①③ 也当成"按需加载"	只有 ② 改变运行时下载时机
以为 Webpack 默认会抽公共包	默认 <code>chunks: 'async'</code> ，只处理动态 import
大依赖留在入口静态链上还想优化首屏	静态 import 的都会进首屏
<code>manualChunks</code> 分组过细	chunk 数量爆炸，请求数反而变多
动态 import 后忘了处理加载态/失败	网络会失败，要有 fallback

2.6 HMR 热更新：一条从文件改动到模块替换的链路

完整链路



accept 边界

- HMR 从改动的模块出发，沿引用链向上冒泡。
- 找到最近一个声明了 `accept` 的模块 → 它就是替换落点。
- 找不到就一直上溯到入口 → 退化为整页刷新。

这就是“为什么有时候 HMR 生效、有时候整个页面刷新了”的答案。不是 HMR 坏了，是改动位置到最近边界之间的链路没有可接受的模块。

边界是谁提供的

框架插件：

框架	提供边界的插件
Vue (Webpack)	<code>vue-loader</code>
Vue (Vite)	<code>@vitejs/plugin-vue</code>
React	<code>react-refresh</code>

没有这些插件的裸项目，**HMR** 会频繁退化成整页刷新 —— 因为没人声明 `accept` 边界。

Vite HMR 快的两条腿

- 1. 不打包 —— 改动后只重新转换那一个文件，不需要重新打包受影响的模块图。
- 2. 沿依赖图做局部化失效传播 —— 只让受影响的模块失效，而不是整图重建。

与整页刷新的对比

	整页刷新	HMR
浏览器行为	重新加载整个页面	只替换变更模块
应用状态	全部丢失	保留（未变更模块不重建）
速度	慢	快

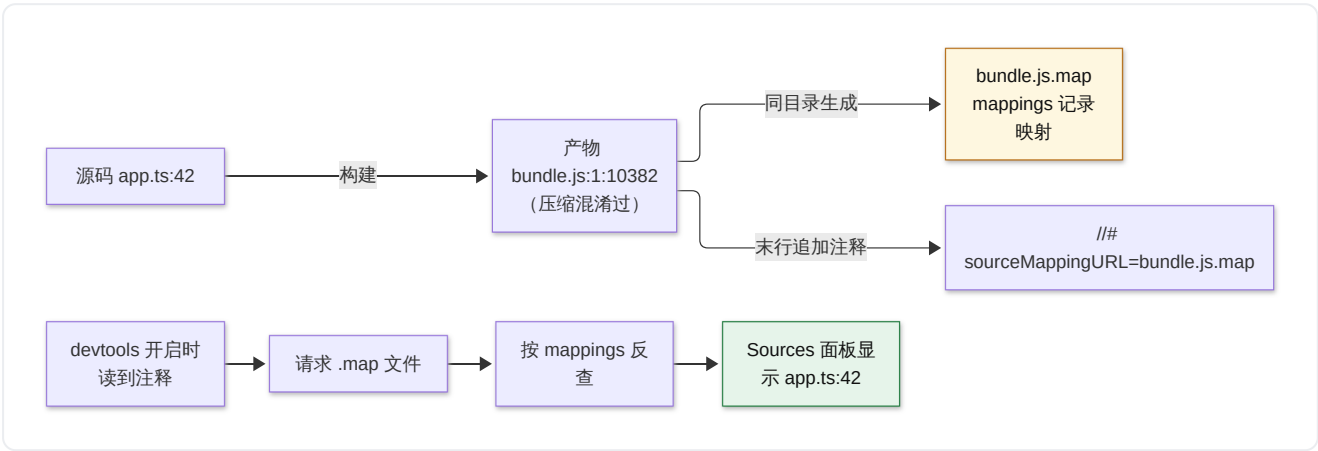
"保留应用状态"是 **HMR** 真正的价值 —— 改一个弹窗样式，不需要重新点开弹窗、重新填一遍表单。

常见的坑

坑	正确认知
以为 HMR 总是不刷新页面	找不到 accept 边界就退化为整页刷新
改了状态相关代码却发现状态丢了	该模块在边界之上，必然整体重建
裸项目期待 HMR 好用	边界要框架插件提供
以为 HMR 影响生产	生产模式没有 HMR
混淆"重新编译"和"重新打包"	Vite dev 只转换单文件，不是打包

2.7 Source Map：调试器与产物之间的反向映射

机制



- 构建时产出 `.map` 文件，记录产物位置 ↔ 源码位置的映射。
- 在产物最后一行追加注释 `//# sourceMappingURL=bundle.js.map`。

- **devtools** 开启状态下加载脚本时读到该注释 → 请求 `.map` → 按 `mappings` 反查 → Sources 面板显示源码原文。

第二种机制：通过 HTTP 响应头 `SourceMap` 指定，可以不依赖注释。

关键性质：**Source Map** 不改变运行结果。它只是给调试器看的一条旁路信息 —— 产物该跑多快还是多快，只是浏览器多下载一个文件。

devtool 梯度 (Webpack)

devtool	速度	质量	适用
<code>eval</code>	极快	差	dev 迭代
<code>eval-cheap-module-source-map</code>	快	中	Webpack dev 推荐
<code>source-map</code>	慢	最高	精确定位
<code>hidden-source-map</code>	慢	最高	生产：生成但不写注释
<code>nosources-source-map</code>	慢	高	生产需对外暴露时

dev 的取舍逻辑：`eval` 系列最快，因为它把 source map 内联进 `eval()` 里，不额外请求文件；代价是映射精度差（`cheap` 表示只映射到行，不映射到列）。

Vite

```
// vite.config.ts
export default {
  build: {
    sourcemap: false, // 默认值
    // false | true | 'inline' | 'hidden'
  },
};
```

Vite 默认 `false` —— 不生成 source map，产物最小。

为什么生产不能用 source-map

注释留在产物里，任何人都能下载 .map 还原完整源码 → 源码泄露。

```
// ❌ 生产危险配置
devtool: 'source-map'
// 产物末行有注释 → 任何人 curl 一下 .map 就拿到你的全部源码（含注释、含业务逻辑）
```

正确做法：

场景	配置
想上传 source map 到监控平台（Sentry 等），但不公开	hidden-source-map —— 生成 .map 但不写注释，手动传给平台
需要对外的产物但仍想模糊源码	nosources-source-map —— 保留行列映射但不含源码内容
完全不需要	不生成（Vite 默认即此）

常见的坑

坑	正确认知
生产用 devtool: 'source-map'	源码泄露，用 hidden-source-map
以为 source map 会影响性能	不改变运行结果，只是多下载一个文件
dev 用最精确的 source-map	dev 要的是快，eval-cheap-module-source-map 更合适
以为 devtools 没开也能映射	devtools 开启状态下才会去读注释、请求 .map
上了 Sentry 但没配 source map	堆栈是压缩后的，定位不到源码

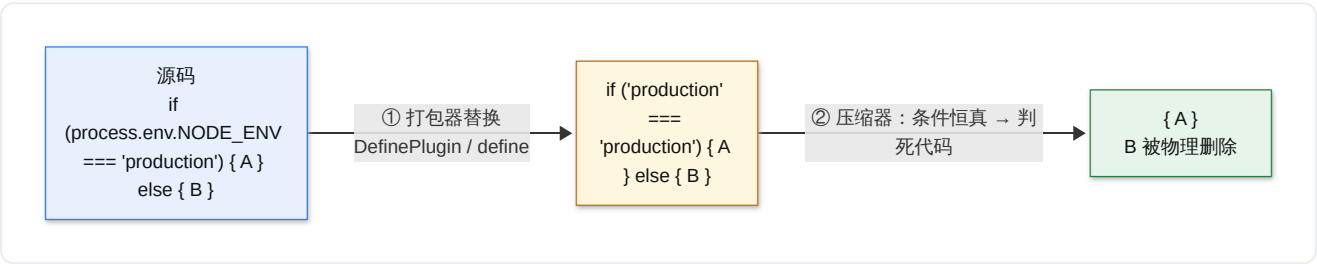
2.8 开发模式与生产模式：差异全景与 mode 的边界

差异全景

维度	开发模式	生产模式
目标	构建快、反馈快	产物小、加载快
压缩混淆	不做	Terser / SWC minify
Tree Shaking	不启用	启用
Source Map	eval-cheap-* 快而糙	hidden 慢而准
代码分割	不抽公共包	splitChunks / manualChunks
HMR	启用	无
NODE_ENV	development	production

核心机制：构建期字面量替换

这是全章最值得理解透彻的一个机制，因为它同时解释了"环境变量"和"死代码删除"两件事。



步骤	谁干	干什么
① 替换	打包器 (Webpack <code>DefinePlugin</code> / Vite <code>define</code>)	把 <code>process.env.NODE_ENV</code> 整体替换成字符串字面量 <code>"production"</code>
② 删除	压缩器 (Terser / SWC)	条件恒假 → 判为死代码 → 物理删除整段

两个执行者，一个都不能少。

关键推论：

- `process` 是 **Node** 全局对象，浏览器里不存在。Webpack 不是“提供了 `process`”，而是在编译期把这段文本换掉了。
- 静态替换只能替换完整字面量表达式。 `process.env[k]`、`import.meta.env[key]` 这类动态取值拿不到值。
- 所以 `NODE_ENV` 不是运行时变量。想靠改服务器环境变量来切换前端行为是无效的。

`mode: 'production'` 自动带来了什么

它只是预设默认值。开启后自动设置：

- `optimization.minimize: true` (Terser 压缩混淆)
- `optimization.nodeEnv: 'production'` (经 `DefinePlugin` 实现上面第 ① 步)
- `optimization.usedExports: true` (Tree Shaking 标记阶段)
- `optimization.sideEffects: true`
- `optimization.concatenateModules: true` (Scope Hoisting)
- `moduleIds` / `chunkIds: 'deterministic'`
- `devtool: false` (不生成 **source map**)

`mode` 管不到的

这是最容易出错的地方 —— 把 `mode` 当“全能开关”。

事项	谁来管
抽公共包	显式配 <code>optimization.splitChunks.chunks: 'all'</code>
懒加载	代码里的 <code>import()</code>
剔除 <code>console</code>	<code>terserOptions.compress.drop_console: true</code>
Source map 策略	显式配 <code>devtool</code>

判断口诀：`mode` 只管"默认值有没有被设定"，不管"你的项目需不需要"。要显式配置的东西，`mode` 一概不管。

配置方式差异（Webpack vs Vite）

	Webpack	Vite
切换方式	一份 config + <code>mode</code> 字段	两条命令 <code>vite</code> / <code>vite build</code>
环境变量	<code>process.env.NODE_ENV</code>	<code>import.meta.env.PROD</code> / <code>DEV</code> / <code>MODE</code> （默认不注入 <code>process.env</code> ）

常见的坑

坑	正确认知
以为 <code>mode</code> 是全能开关	它只是预设默认值
把 <code>process.env.NODE_ENV</code> 当运行时变量	它是构建期字面量替换
动态取 <code>import.meta.env[k]</code>	静态替换只能替换完整字面量
生产忘了配 <code>devtool</code>	<code>mode: 'production'</code> 下 <code>devtool: false</code> , 默认不生成 source map
以为 <code>mode: 'production'</code> 会删 console	要显式配 <code>drop_console</code>
dev 也用 <code>mode: 'production'</code>	会失去 HMR、构建变慢，得不偿失

2.9 本章常见误区（12 条）

#	误区	正确认知
1	以为共享模块在产物里也去重	图去重只管内存；产物是否重复由 <code>splitChunks</code> 决定
2	把「抽公共包」当打包器默认行为	Webpack 5 默认 <code>chunks: 'async'</code> ，只处理动态 import
3	以为 loader 顺序错了「所以报错」	本质是类型管道，喂错类型才断
4	用「流程动作」判断 Loader / Plugin	判别看作用域：单文件 → Loader；跨文件/阶段 → Plugin
5	以为 Tree Shaking 一步完成	打包器标记 + 压缩器删除，两个执行者
6	改自己项目 <code>sideEffects</code> 想减小第三方库	第三方看库作者；你能做的是改导入路径
7	混淆 <code>chunkhash</code> 和 <code>contenthash</code>	<code>chunkhash</code> 看「哪个包」， <code>contenthash</code> 看「这份文件」
8	以为动态 import 能减小总体积	只减小首屏体积
9	以为 HMR 总是不刷新页面	找不到 accept 边界就退化为整页刷新
10	生产误配 <code>devtool: 'source-map'</code>	会泄露源码；生产用 <code>hidden-source-map</code>
11	以为 <code>mode</code> 是全能开关	它只是预设默认值
12	<code>process.env.NODE_ENV</code> 当成运行时变量	它是构建期字面量替换

下一章：第 3 章 · Webpack 与 Vite —— 把本章的机制放到两个具体工具上对照。

第 3 章 · Webpack 与 Vite

主线索：

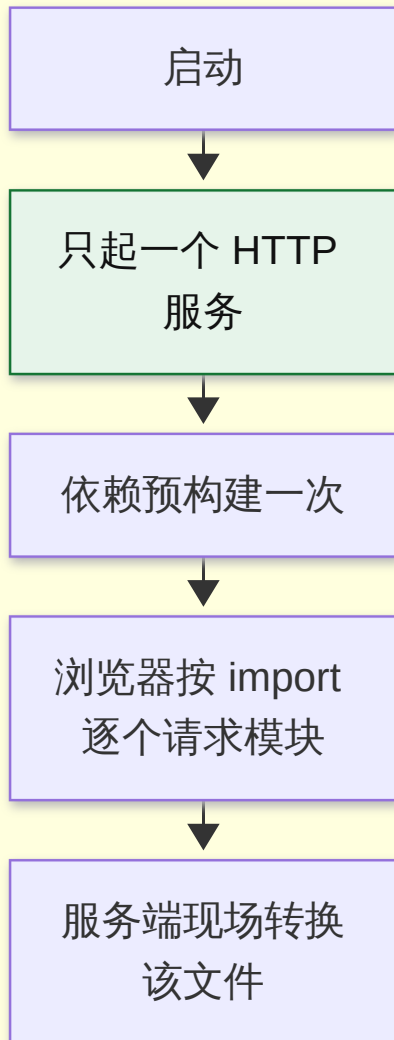
「快」必须绑定指标；选型必须绑定场景。Webpack 用配置换控制力，Vite 用约定换简洁 —— 两者都没有绝对优劣。

3.1 架构分野：打包式构建 vs 原生 ESM

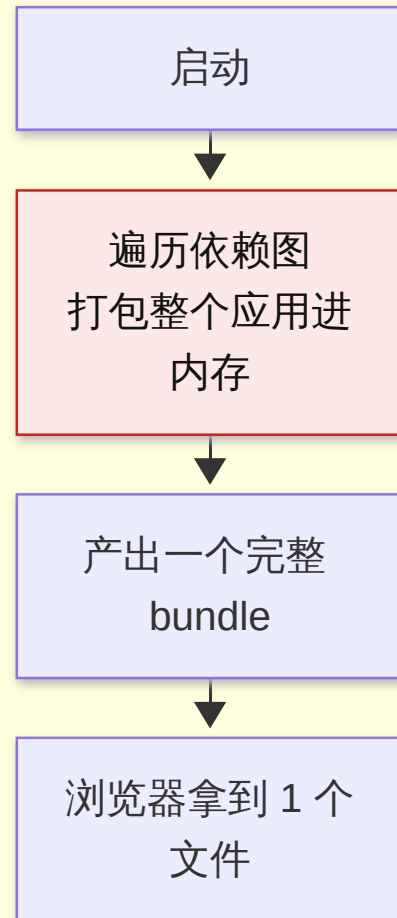
这是两个工具所有差异的源头。

开发服务器的两种架构

Vite dev：先服务，按需转换



Webpack dev：先打包，再服务



对照表

维度	Webpack dev	Vite dev
启动时做什么	打包整个应用进内存	只起 HTTP 服务 (+ 依赖预构建)
浏览器拿到	1 个完整 bundle	很多个原始模块文件
冷启动	$\propto$ 项目规模	$\approx$ 常数
首次访问页面	快 (已打包好)	略慢 (现场转换若干文件)
改动之后	重新打包受影响模块	只转换那一个文件
浏览器要求	无 (产物已降级)	需支持原生 ESM

关键区分：「Vite 冷启动快」 $\neq$ 「页面首屏打开快」

这是一个必须分清的区别，因为两者经常被混为一谈。

- **Vite 把成本从「启动时」挪到了「首次访问时」** —— 启动时不用打包，代价是浏览器要现场请求并等待服务端转换。
- **而且这个代价不影响生产** —— `vite build` 仍然走完整打包 (Rollup/Rolldown)，产出的仍是打包好的产物。

推论：

说法	判定
"Vite 冷启动快"	✅ 成立
"Vite 首屏快"	❌ 不一定 —— dev 下反而可能略慢
"Vite 生产构建快"	❌ 不保证，见 §3.6

为什么这个区分重要：它决定你在什么场景下能拿"Vite 快"当理由。开发时省的是你自己的等待，不是用户的加载时间。

3.2 依赖预构建：Vite dev 专属的两件事

Vite dev 启动时，会先把 `node_modules` 里的依赖"预构建"一遍，产物放在 `node_modules/.vite`。

官方的两个目的，顺序不能反：

目的	说明
① CommonJS / UMD 兼容	dev 以原生 ESM 提供代码，而 CJS/UMD 依赖浏览器不认。不做预构建会直接报错跑不起来（浏览器不认 <code>require</code> ）
② 性能	内部模块极多的 ESM 依赖会被合并成单个模块。例： <code>lodash-es</code> 有 600+ 内部模块，不合并会产生 600+ 个请求，造成网络拥塞

为什么①排在②前面：很多人以为预构建"只是为了快"。实际上不做就根本跑不起来——这是功能性需求，不是优化。

关于"600+ 请求"的准确表述：那是并发风暴导致的网络拥塞，不是"请求瀑布"。请求瀑布（waterfall）指串行的依赖链——A 加载完才知道要加载 B，B 加载完才知道要 C。这个区分在第 4 章讲 Farm 的 Partial bundling 时会用到。

缓存行为

- 缓存位置：`node_modules/.vite`
- 失效条件：只随 **lockfile / vite.config / NODE_ENV** 变化重跑
- 浏览器端：依赖请求带 `max-age=31536000, immutable` 强缓存

实用含义：遇到"明明改了依赖但没生效"，先删 `node_modules/.vite` 重跑。这是排查 Vite dev 异常的常规动作。

执行者

当前官方文档显示依赖预构建由 **Rolldown** 执行（早期为 esbuild）。

这条信息的价值不在于“记工具名”，而在于：它是第 4 章整章主题的一个具体实例 —— 同一条流水线上，某一层的执行者被换掉了，但机制没变。

3.3 配置方式对比：默认值覆盖了哪一层

核心认知

Vite 配置少，是因为它把大量约定内置成了默认值。

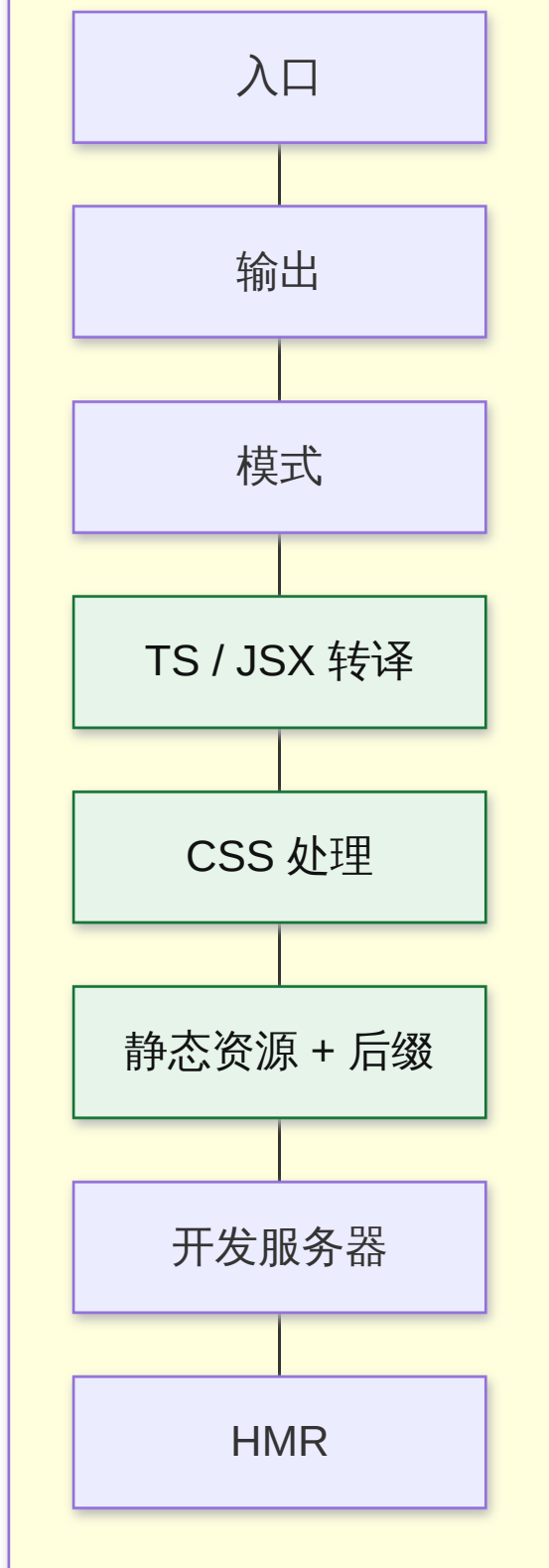
这与第 2 章 §2.8 的「mode 只是预设默认值」是同一个逻辑，只是 Vite 的覆盖面宽得多。

关键差异：默认值管到哪一层

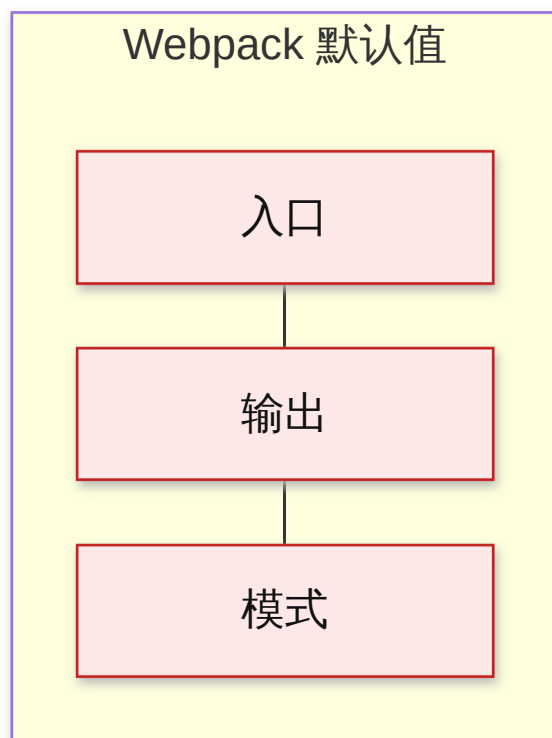
	默认值覆盖范围
Webpack	只管骨架（入口 / 输出 / 模式），不管“各类文件怎么被理解”
Vite	连语言层一起管（TS 转译、CSS、静态资源、资源后缀）

一句话：**Webpack** 管「东西放哪」，**Vite** 还管「东西怎么被理解」。

Vite 默认值



Webpack 默认值



逐项对照表

事项	Webpack	Vite
处理 TS / JSX	<code>module.rules</code> + <code>babel-loader</code> / <code>ts-loader</code>	内置
处理 CSS	<code>css-loader</code> + <code>style-loader</code>	内置，直接 <code>import './a.css'</code>
静态资源	<code>asset modules</code> (v5 已内置，仍需一行 <code>type: 'asset/resource'</code>)	内置， <code>?url</code> / <code>?raw</code> / <code>?inline</code>
路径别名	<code>resolve.alias</code>	<code>resolve.alias</code> (同名，可迁移)
环境变量	<code>DefinePlugin</code>	<code>define</code> + <code>.env</code> + <code>import.meta.env</code>
开发服务器	<code>devServer</code> (需装 <code>webpack-dev-server</code>)	<code>server</code> (内置)
生产构建	仍是 Webpack	Rollup / Rolldown
插件写法	<code>new XxxPlugin()</code>	<code>xxx()</code> 工厂函数
HMR	需配置	内置

一个精度修正：说"Webpack 的静态资源处理要配 loader"是不准确的。Webpack 5 的 `asset modules` 官方表述是 *"without configuring additional loaders"* —— 但仍需要一行 `module.rules` 的 `type: 'asset/resource'` 声明。准确说法是「已内置，但需一行规则声明」。

三个常见误判

误判	为什么错
"生产构建"是差异项	❌ 两边默认都是 production
"Webpack 什么都要手配"	❌ 它有默认值，只是只覆盖骨架层
"静态资源"是纯差异项	⚠️ 半成立 —— 见上面的精度修正

该举而容易漏的两个例子：**TS 转译**、**CSS 处理** —— 这两个才是"语言层"的典型代表。

3.4 环境变量机制：一个被构建期替换"伪造"出的对象

这一节与第 2 章 §2.8 的「构建期字面量替换」同源，只是换到两个具体工具上看。

事实

`process` 是 **Node** 全局对象，浏览器里不存在。

Webpack 不是"提供了 `process`"，而是靠 **DefinePlugin** 在构建期做文本替换，把 `process.env.X` 整体换成一个字符串字面量 —— "伪造"出来的。

两边的做法

Webpack :

```
// webpack.config.js
const webpack = require('webpack');

module.exports = {
  plugins: [
    new webpack.DefinePlugin({
      'process.env.API_URL': JSON.stringify(process.env.API_URL),
    }),
  ],
};
```

Vite :

```
# .env
VITE_API_URL=https://api.example.com

// 代码里
const base = import.meta.env.VITE_API_URL;

// 要在 config 里读非 VITE_ 前缀的变量
import { defineConfig, loadEnv } from 'vite';

export default defineConfig(({ mode }) => {
  const env = loadEnv(mode, process.cwd(), ''); // 第三个参数 '' = 不过滤前缀
  return {
    define: {
      __APP_VERSION__: JSON.stringify(env.APP_VERSION),
    },
  };
});
```

三条规则

1. 只有 `VITE_` 前缀的变量默认暴露给前端代码。—— 这是安全设计：防止服务端密钥（如 `DB_PASSWORD`）意外打进前端产物。
2. Vite 默认不注入 `process.env`。但可以用 `define` 显式达成 —— 所以准确说法不是“Vite 不支持”，而是“默认不替换”。
3. `import.meta.env.X` 同样是静态替换。必须写成完整字面量表达式，动态取值拿不到。

```
// ❌ 拿不到值
const key = 'VITE_API_URL';
const url = import.meta.env[key];

// ✅ 必须字面量
const url = import.meta.env.VITE_API_URL;
```

为什么这是高频坑

因为它在开发时可能完全不报错 —— 只是取到 `undefined`，然后你的请求打到了 `undefined/api` 或者相对路径上。表现为“接口地址莫名其妙不对”，而不是一个明确的语法错误。

常见的坑

坑	正确认知
说“Vite 不支持 <code>process.env</code> ”	是浏览器没有 <code>process</code> ；Webpack 靠构建期替换伪造
动态取 <code>import.meta.env[k]</code>	静态替换只能替换完整字面量表达式
把密钥写成 <code>VITE_</code> 前缀	会被打进前端产物，泄露
以为 <code>NODE_ENV</code> 可以运行时切换	构建期就定死了
config 里读不到非 <code>VITE_</code> 变量	用 <code>loadEnv(mode, cwd, '')</code>

3.5 插件生态对比：一套概念还是两套

扩展点的数量

	扩展点	说明
Webpack	两套	Loader（转单个模块）+ Plugin（管整个流程）—— 写扩展前要先判断类型
Vite	一套	统一的 Plugin（Rollup 插件接口的超集），文件转换在 <code>transform</code> 钩子里

核心取舍

Vite 用「少一层概念」换来了插件作者的「多一套适配」。

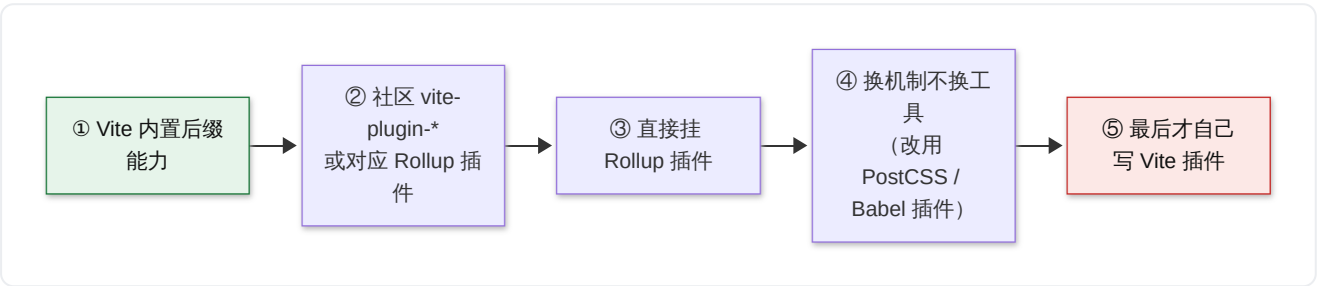
复杂度从使用者（多数）转移到了插件作者（少数）。

这不是“谁更好”，这是一个明确的取舍选择，而且与 Vite“配置少、约定多”是同一设计逻辑 —— 都是“把复杂度集中到少数人手里，让多数人少做决定”。

另一面：Vite 借力 Rollup 生态，省掉了从零建生态的成本。这正是第 4 章「双边网络效应」的一个正面实例 —— 它没有另造生态，而是复用了现成的插件接口。

遇到「某个 loader 在 Vite 里没有对应物」怎么办

排查优先级，成本从低到高：



优先级	做法	成本
①	Vite 内置后缀能力替代 —— <code>?raw</code> 替代 raw-loader、 <code>?url</code> 替代 file-loader、 <code>?inline</code> 替代 url-loader	零配置
②	找社区 <code>vite-plugin-*</code> 或对应 Rollup 插件	装一个包
③	直接挂 Rollup 插件 (<code>plugins</code> / <code>build.rollupOptions.plugins</code>)	需要处理 dev/build 差异
④	换机制不换工具 (改用 PostCSS / Babel 插件实现)	换实现思路
⑤	最后才自己写 Vite 插件	自己维护

为什么这个顺序值得记：它是一条通用的排查顺序 —— 先替代 → 再复用 → 最后自写。同样的顺序适用于“构建慢怎么办”“依赖冲突怎么办”等一大类问题。

措辞上的一个修正：在很多情况下，你根本不需要“替换”某个 loader，而是“找等价物”或“重新实现”。第 ① 项就是典型 —— 它不需要任何插件。

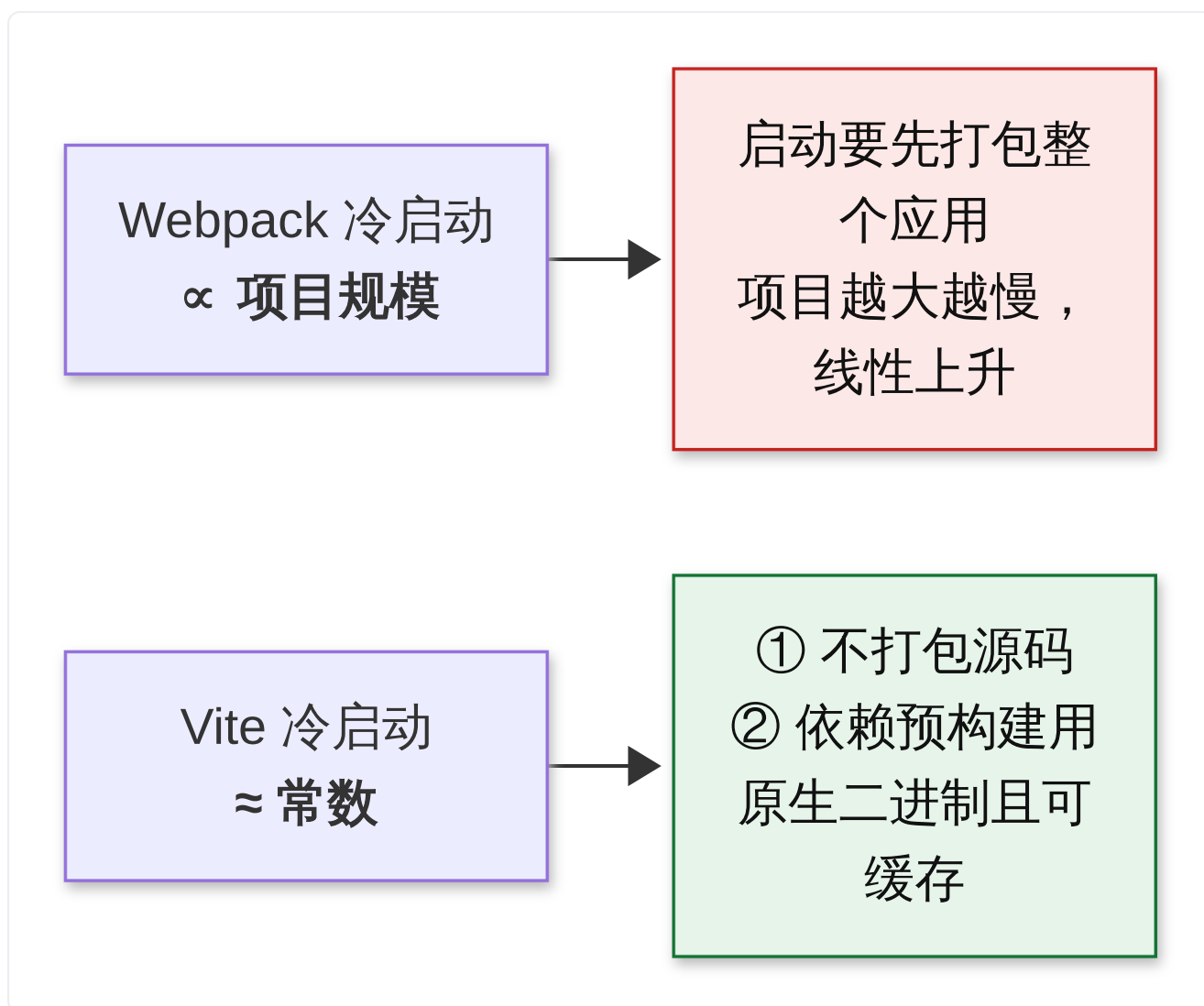
常见的坑

坑	原因
以为 Rollup 插件拿来就能用	dev 下不打包 ， <code>generateBundle</code> / <code>writeBundle</code> 等钩子不触发
忘配 <code>apply</code>	<code>apply: 'serve' 'build'</code> 决定插件在哪个模式生效
想把 Webpack 插件迁到 Vite	钩子体系不同 (Tapable vs Rollup)， 无法迁移 ，只能重写
一上来就自己写 Vite 插件	顺序错了，先看 ①②
在 dev 下调试只在 build 触发的钩子	那个钩子在 dev 里根本不会被调用

3.6 速度对比：把「快」拆成四个指标

本节刻意不给具体倍数。理由是倍数强依赖项目规模、依赖数量、插件数量、机器配置 —— 给了反而误导。

两条曲线（机制决定，不是跑分得出）



Vite 冷启动是「近似」常数而非「严格」常数 —— 因为它启动时仍要做**依赖预构建**（Rolldown / esbuild，原生二进制 + 有缓存），依赖越多这一步越慢。**完整原因是两条叠加：**① 不打包源码 + ② 依赖预构建用原生二进制且可缓存。

四个指标的判定

说法	判定	理由
"Vite 冷启动更快"	✅ 通常成立	规模越大差距越大（两条曲线斜率不同）
"Vite 开发体验更快"	✅ 通常成立	冷启动 + HMR 都占优
"Vite 构建更快"	❌ 不保证	build 走 Rollup/Rolldown 单线程，超大项目可能更慢
"Vite 页面首屏更快"	❌ 不一定	dev 下首屏反而略慢（见 §3.1）

"Vite 快"只属于 **dev**，不属于 **build**。把它当成通用结论，是选型时最常见的一类依据错误。

别忘了 Webpack 的缓存

Webpack 5 的持久化缓存 `cache: { type: 'filesystem' }` 让二次构建接近增量。讨论"谁更快"时，拿一个没开缓存的 Webpack 去比，是不公平的对比。

```
module.exports = {
  cache: {
    type: 'filesystem',          // 持久化到磁盘
    buildDependencies: {
      config: [__filename],     // config 变了才失效
    },
  },
};
```

测量方法

六步法（第 5 章 §5.6 展开）：

- 1. 拆指标 —— 冷构建 / 增量构建 / HMR / 首屏，一次只测一个
- 2. 固定变量 —— 同机器、同代码规模
- 3. 清三层缓存 —— `node_modules/.cache`、`node_modules/.vite`、浏览器无痕

- 4. 各跑 5 次取中位数 —— 单次测的是系统负载，不是构建性能
- 5. 记录元数据 —— 代码行数 / 依赖数 / 插件数
- 6. 出带前提的结论

常见的坑

坑	正确认知
说"Vite 构建更快"	build 走单线程，超大项目可能更慢
拿厂商基准数字当自己的结论	厂商数字是起点，不是判据
比较时 Webpack 没开持久化缓存	不公平对比
只跑一次就下结论	单次测的是系统负载
不区分冷启动和首屏	两者机制不同，结论相反
不记录代码规模就报倍数	结论无法复现

3.7 本章常见误区（12 条）

#	误区	正确认知
1	以为 Vite dev 依赖预构建只是为了性能	首要理由是 CJS/UMD → ESM 兼容 ；不做则直接跑不起来
2	把"600+ 请求"叫 请求瀑布	那是 并发风暴 导致的网络拥塞；瀑布指串行依赖链
3	"Vite 冷启动快" = "首屏快"	成本被 挪到首次访问 ；且不影响生产
4	说"Webpack 什么都要手配"	Webpack 有默认值，只是只覆盖 骨架层
5	说"Vite 不支持 <code>process.env</code> "	是 浏览器没有 <code>process</code> ；Webpack 靠构建期替换伪造
6	动态取 <code>import.meta.env[k]</code>	静态替换只能替换 完整字面量表达式
7	以为 Rollup 插件拿来就能用	dev 不打包， <code>generateBundle</code> 等钩子 不触发
8	一上来就自己写 Vite 插件	顺序： 内置后缀替代 → 社区插件 → Rollup 插件 → 自写
9	说"Vite 构建更快"	build 走单线程，超大项目可能更慢
10	选型先算 ROI	应先问 有没有真实痛点 ；无痛点则收益侧为空
11	组件库直接用 Vite 默认配置	需 dts 插件、显式 <code>formats</code> 、 <code>external</code> <code>peer</code> 、对齐 <code>exports</code>
12	选型理由是"更现代 / 更先进"	理由必须 对着场景需求 说

下一章：第 4 章 · 新一代构建工具 —— 当同一条流水线的每一层，都换成了 Rust / Go。

第 4 章 · 新一代构建工具

主线索：

「换引擎」是这一轮的全部主题。四条路线用四种方式把 JS 引擎换成 Rust / Go，但只有三种打法。判断该学什么，看折旧率，不看新旧。

4.1 为什么原生化：并行度决定难度

问题在哪

构建这件事的很大一部分是「逐文件文本转换」——把 TS 转成 JS、把 SCSS 转成 CSS、把 JS 压缩。这类工作有两个特点：

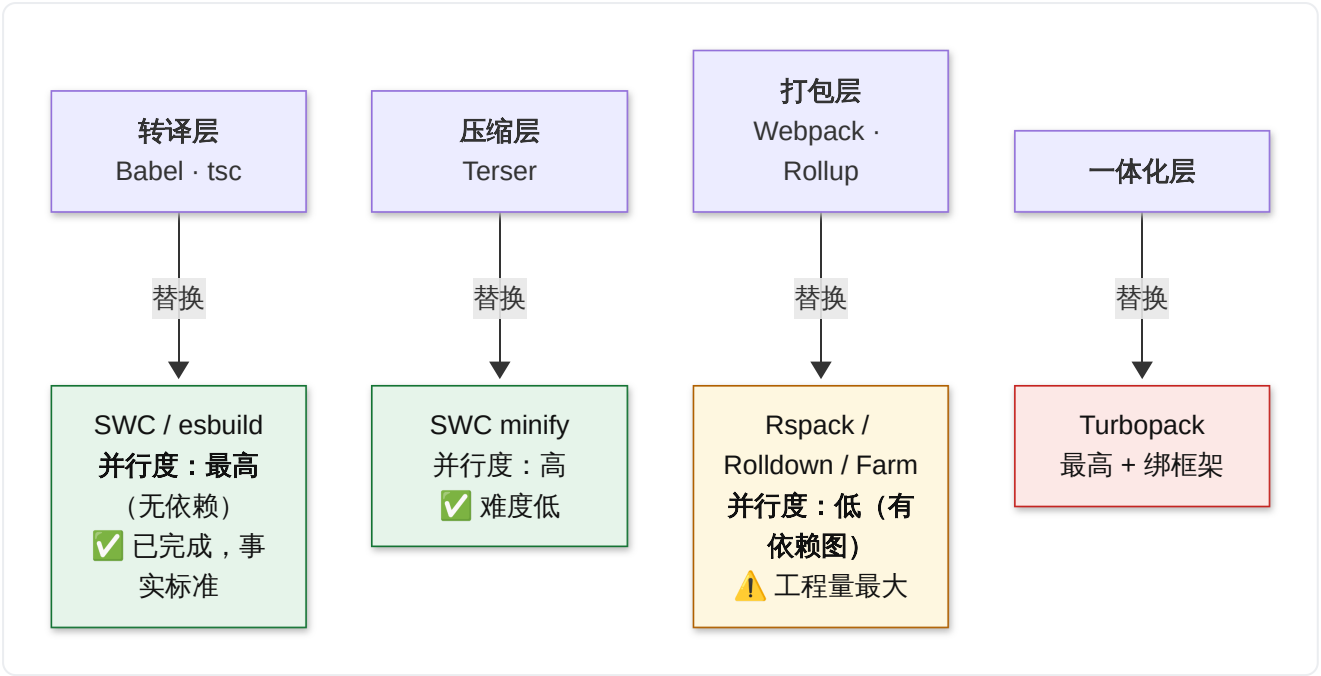
- CPU 密集
- 天然可并行 —— 文件 A 的转换不依赖文件 B 的转换结果

但 **JavaScript** 是单线程的，用不上多核。

原生的三个优势

优势	说明
机器码无预热	无需 JIT 预热，启动即满速
真多核	可以真正使用多核并行
紧凑内存	数据布局更紧凑，序列化 / 反序列化开销更低

分层替换路线图



层	并行度	难度	现状
转译层	最高 (无依赖)	低	已完成, 事实标准
压缩层	高	低	已完成
打包层	低 (有依赖图)	最高	四方混战
一体化层	—	最高 + 绑框架	单框架内已默认

核心结论

并行度决定原生化难度。打包层最难，因为依赖图是串行约束 —— 你必须先知道谁依赖谁，才能决定怎么打包。这份"先知道"的关系，无法像"转换单个文件"那样简单并行。

这条结论的实用价值：它解释了为什么"转译层早就换完了，打包层还在打"。这不是"打包层的人不努力"，是问题本身的并行度不同。

一个精度修正：说"前端构建是纯单线程"并不准确。**Webpack 5 里 Terser 默认 `parallel: true`**（用 worker 线程），**并非纯单线程**。换 SWC minify 的收益来自 ① 原生执行速度 ② **更低的序列化 / 内存开销**（worker 之间传递数据本身有成本）。

4.2 esbuild 与 SWC：转译与压缩层

这两个工具最可能被真正用上，因为可以**单独替换**，不牵动整个构建流程。

对照表

维度	esbuild	SWC
实现语言	Go	Rust
定位	"够用就好"的极速转译器	"Babel 的替代品"
类型检查	❌ 不做	❌ 不做
Babel 插件兼容	❌	部分
打包能力	有基础能力，但生态不足	无
典型场景	Vite dev 转译、依赖预构建	Next.js、 <code>swc-loader</code> 、 <code>@swc/jest</code>

共同边界：只剥离类型标注，不验证类型

这是本节最重要的认知。

✗ 不负责

类型错误报告

类型检查

类型推导

✓ 负责

压缩与混淆

TS 类型标注剥

离 TS



esbuild 与 SWC 都不做类型检查 —— 它们只剥离类型标注，不验证类型是否正确。

直接后果：`vite build` 默认不会因类型错误失败

这一点极其容易踩。你的项目里可能有 30 个类型错误，`vite build` 照样构建成功、照样产出产物、照样能部署。

正确做法：显式把类型检查挂到构建脚本或 CI：

```
{
  "scripts": {
    "dev": "vite",
    "build": "vue-tsc --noEmit && vite build",
    "build:react": "tsc --noEmit && vite build"
  }
}
```

命令	适用
<code>tsc --noEmit</code>	React / 纯 TS 项目
<code>vue-tsc --noEmit</code>	Vue 项目

推论：看到"类型检查"这件事，要问**谁在做** —— 转译器不做，那是**另一条独立的流水线**在管（TS 编译器）。**转译和类型检查是两件事**，只是过去 **Babel + tsc** 的组合让人以为它们是一体的。

SWC 不能完全替代 Babel

	Babel 插件	SWC 插件
实现语言	JavaScript	Rust / WASM
能否互搬	—	搬不过去

所以"把 **Babel** 换成 **SWC**"的真实成本，取决于你用了多少 **Babel** 插件、以及这些插件有没有 **SWC** 等价物。一个用了 5 个 Babel 插件的老项目，动手前必须先把插件清单核一遍 —— 这是第 7 章 §7.3 台阶 5「迁移评估」的一个具体场景。

常见的坑

坑	正确认知
以为 esbuild / SWC 能做类型检查	只剥离类型标注，不验证类型
以为 vite build 会因类型错误失败	默认不会，需显式挂 tsc --noEmit
以为 SWC 能完全替代 Babel	插件体系不同（Rust/WASM vs JS），搬不过去
以为 esbuild 能直接替代 Webpack 打包	打包能力与生态都不具备
忽略装饰器语义差异	TS 装饰器在不同实现下语义有差别，需实跑验证
以为换原生转译器一律"快很多"	小项目上收益不明显，瓶颈可能在别处

4.3 Rspack 与 Farm：借别人的生态

这两条路线的共同点：都选择兼容一个既有生态，而不是另造一套。

对照表

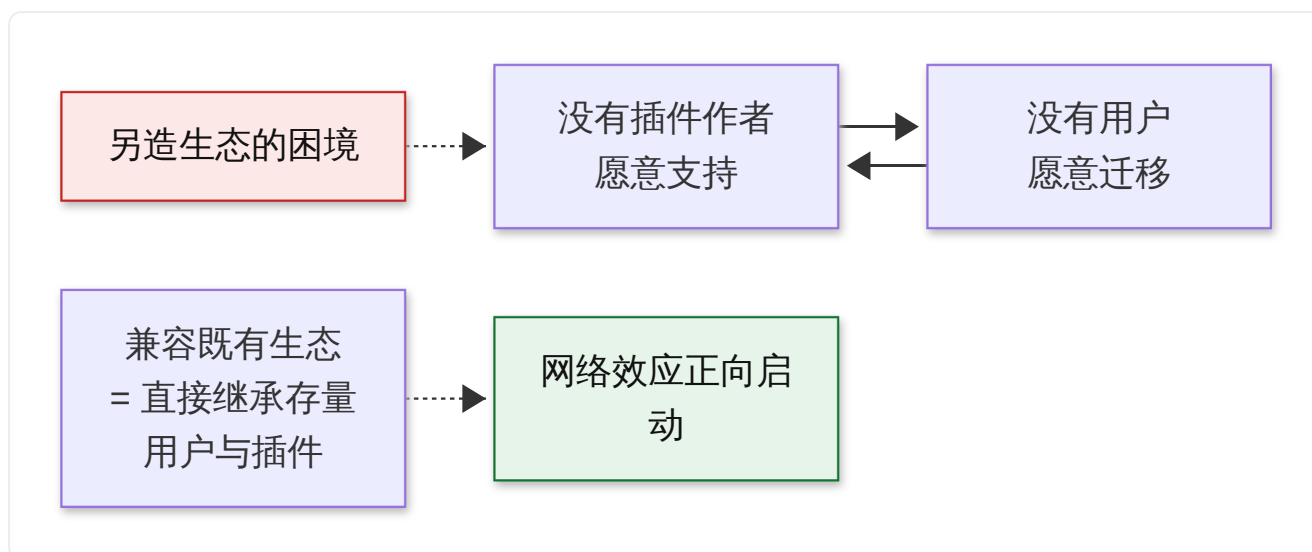
维度	Rspack	Farm
兼容目标	Webpack	Vite
配置	<code>rspack.config.js</code> (≈ webpack 配置改名)	<code>farm.config.ts</code> (需重写)
插件	loader runner 与 Webpack 完全兼容；top 50 webpack 插件 >80% 可用或有替代	<code>vitePlugins</code> 直接挂 Vite / Rollup / unplugin 插件
成熟度	高 (1.x 生产可用；字节内部 >1000 应用)	中 (独立项目，生态小)
差异化	Module Federation 2.0	Partial bundling

采用方：Rspack 在字节内部有 >1000 个 Web 应用（抖音、飞书、Coze），外部有微软、亚马逊、阿里、Discord 等。Farm 2023 年开源，2024-2025 达生产可用，生态与社区规模明显小于 Rspack。

核心机制：为什么必须兼容

生态迁移成本 > 重写成本，所以必须兼容。

这不是"性价比考虑"，是生存条件。 关键机制是双边网络效应：



机制拆解：

- 插件作者没有动力为"没有用户的新工具"重写插件 —— 而他们的用户都还在 Webpack / Vite 上。
- 所以另造生态不是"成本高"，是"冷启动根本起不来"。

边际成本对照：

做法	边际成本	可控性
兼容	在 Rust 侧实现一套 JS 插件运行器（一次性投入）	自己可控
造生态	说服成千上万第三方作者（持续投入）	不可控

一个决定性的先例：**Vite** 自己就是这么起来的 —— 直接复用 Rollup 的插件接口（见 §3.5）。这不是巧合，是唯一被验证过的路径。

策略层的一层：兼容谁 = 抢谁的存量用户。Rspack 选 Webpack、Farm 选 Vite，两条路线选了不同的兼容对象，同时避免了彼此同质化。

Rspack 迁移清单

原来	换成
HtmlWebpackPlugin	HtmlRspackPlugin
MiniCssExtractPlugin	CssExtractRspackPlugin
css-loader	内置 type: 'css'
babel-loader / swc-loader	builtin:swc-loader

关于 `builtin:swc-loader`：推荐它是因为避免 JS ↔ Rust 之间的往返通信开销 —— 在 JS 侧跑一个 loader，数据要序列化传给 Rust 再传回来，这个开销在大量文件上很可观。确需 Babel 插件时，应把它的作用范围限制在少数文件上。

少数不支持项：`resolve.plugins` 等。所以"完全兼容"这个说法要精确 —— 是**"几乎全部社区 loader 可用 + top 50 插件 >80% 可用或有替代"，不是 100%。

Farm 的两个特有概念

① vitePlugins

```
// farm.config.ts
export default defineConfig({
  vitePlugins: [
    // 直接挂 Vite / Rollup / unplugin 插件
  ],
});
```

② 三类插件，其中 JS 插件必须配 filters

类型	例
Farm Rust 插件	@farmfe/plugin-react、@farmfe/plugin-sass
Farm JS 插件	必须配 <code>filters</code> ，否则明显掉速
SWC 插件	SWC 生态插件

`filters` 的作用：告诉 Rust 侧"哪些文件才需要调用这个 JS 插件"。不配的话，每个文件都要跨语言往返一次，性能优势直接被吃掉。

③ Partial bundling

Farm 的独有策略，介于"全打包"与"不打包 ESM"之间：既避免"不打包"导致的请求瀑布，也避免"全打包"产生巨型产物。

为什么这个中间态有意义：§3.2 讲过两个问题 —— 请求瀑布（串行依赖链）和网络拥塞（并发太多）。Partial bundling 是在两者之间找一个平衡点。

常见的坑

坑	正确认知
以为"兼容"等于 100% 无痛迁移	是"几乎全部 loader + top 50 插件 >80%"
以为另造生态只是"成本高"	是冷启动起不来（双边网络效应）
Farm 的 JS 插件不配 <code>filters</code>	明显掉速
在 Rspack 里继续用 <code>babel-loader</code>	应换 <code>builtin:swc-loader</code> ，减少跨语言往返
以为配置能直接改名就用	Farm 的 <code>farm.config.ts</code> 需要重写

4.4 Turbopack 与 Rolldown：另外两种打法

前两条路线都是"借别人的生态"。这一节是另外两种：一条谁都不兼容，一条由生态主人自己动刀。

对照表

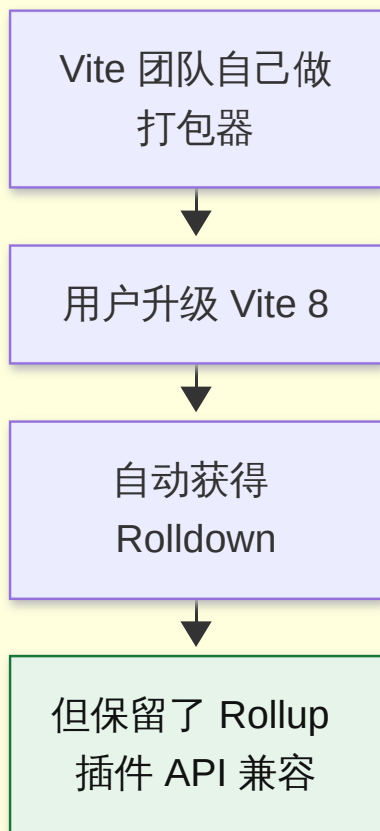
维度	Turbopack	Rolldown
归属	Vercel (Next.js)	VoidZero (Vite 团队)
兼容对象	无	Rollup 插件 API
默认落地	Next.js 16 (2025-10) dev 与 build 双双默认	Vite 8 (2026-03-12) 默认；Rolldown 1.0 (2026-05-07)
核心卖点	函数级增量 + 落盘、单图覆盖多环境、dev 懒编译	dev / prod 同一引擎 (parity)；官方称比 Rollup 快 10-30x
硬约束	不支持 webpack 插件；有 webpack() 配置则 next build 直接失败	Node 20.19+ / 22.12+；ESM-only；Yarn PnP 兼容问题 (Windows 尤甚)

核心洞见：谁能不兼容

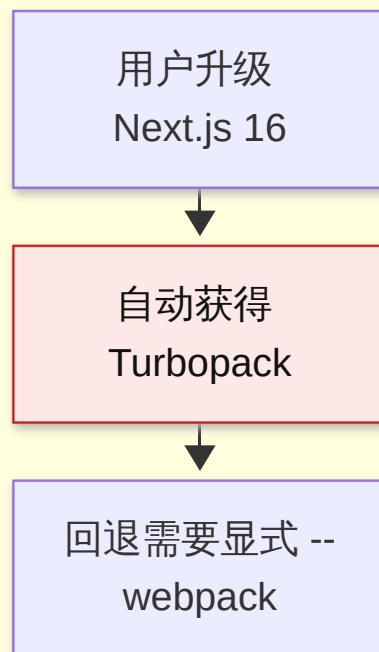
只有"能连用户一起换"的玩家才敢不兼容。

Turbopack 敢，因为打包器不是独立选项 —— 它被包在框架里，用户升级 Next.js 就自动换了。

Rolldown : 生态主人自己动手



Turbopack : 打包器被包在框架里



敢不兼容的三个条件：

条件	说明
① 有分发渠道	打包器被包在框架里，随框架一起分发
② 回退高摩擦	不是禁止，而是把回退做成**"逆默认值"**操作
③ 有资格	足够大的用户基数 + 不可替代性

强制 ≠ 禁止。强制 = 默认 + 高摩擦回退。

为什么"高摩擦"就够：--webpack 这个 flag 一直都在，技术上完全可行。但大多数团队不会去按——因为默认值已经能满足需求，而按下去意味着"我知道得比框架作者多"。门槛不在于能不能做，在于要不要为它付出额外的心智成本。

推论：没用户基数的工具学这招会直接死。三个条件缺一个，强制就变成自杀。

Turbopack 的三条设计

支撑它速度的是三件事：

设计	说明
① 单张模块图	浏览器 + Node + Server Components 共用一张图（不用为不同环境各建一张）
② 函数级增量 + 落盘	增量计算的缓存粒度到函数级，并持久化到磁盘（16.1 dev 落盘，16.3 build 默认开）
③ dev 懒编译	只编译访问到的路由，不编译整个应用

其他事实：

- 转换用 **SWC**，CSS 用 **Lightning CSS**。
- 不做类型检查 —— `next build` 另行跑 TypeScript。
- 自有 loader 语法 `experimental.turbo.rules`，非全部 **webpack loader** 可用。
- Sass 的 `~` 前缀不再支持。
- Vercel 口径：生产构建快 2-5x、Fast Refresh 快至 10x。厂商自测数据，需按项目自测。

关于“**webpack 配置导致 build 失败**”：这是故意设计 —— `next.config` 里存在 `webpack()` 函数时，`next build` 直接失败，拒绝静默忽略。三条出路：把配置迁移到 `turbopack` 键 / 忽略 `webpack` 配置构建 / 显式 `--webpack` 退回。

Rolldown 的时间线与收益

时间线：

2024-04 首发 0.10.1
2024-12 1.0.0-beta.1
2025-05 rolldown-vite 预览
2025-12 Vite 8 Beta
2026-01 1.0 RC
2026-03-12 Vite 8.0 稳定 — Rolldown 成为默认
2026-05-07 Rolldown 1.0 稳定

技术底座：Rust，构建在 **Oxc** 之上，兼容 **Rollup** 插件 **API**。

Vite 8 的变化：单引擎取代 esbuild(dev) + Rollup(build)；转换改由 **Oxc**，CSS 压缩改由 **Lightning CSS**。

官方口径：比 Rollup 快 **10-30x**，与 esbuild 持平。

官方公布的真实案例：Linear 46s→6s (-87%)、GitLab 2.5min→22s、Ramp -57%、Beehiiv -64%、Mercedes-Benz.io -38%。

最值钱的收益不是速度，是 parity

Rolldown 最被低估的收益是 **dev / prod** 一致性。

问题是什么：以前 esbuild(dev) 与 Rollup(build) 是两套模块解析、两套转换、两套 **tree-shaking** 语义。同一份代码在 dev 下能跑，build 之后行为变了 —— 而且只在部署后暴露。

统一引擎之后：本地测的就是要发布的，“本地能跑、生产挂”这一整类 **bug** 消失。

为什么一致性比速度值钱：

维度	速度提升	一致性提升
上限	有上限，且一次性	问题的排查成本与业务风险无上限
替代方案	有 —— 增量缓存、换机器、加人	没有 —— 除了统一引擎
暴露时机	立刻可见	不修就一直有

一个有力的反证：**Vite 8 的 dev HMR 速度相比 Vite 7 基本没变**——收益在生产构建与依赖预构建。如果只图速度，这次升级的收益会很尴尬。

Rolldown 的其他要点

- 内置能力（省掉插件）：CJS/ESM 互操作（免 `@rollup/plugin-commonjs`）、Node 解析（免 `@rollup/plugin-node-resolve`）、`output.codeSplitting`。
- 配置变化：`build.rollupOptions` → `build.rolldownOptions`。
- 硬约束：Node **20.19+ / 22.12+**；包变为 **ESM-only**；Yarn PnP 兼容性有问题（Windows 尤甚）。

常见的坑

坑	正确认知
以为 Turbopack 可以通用	框架内置，离开 Next.js 无意义
以为 Turbopack 会做类型检查	<code>next build</code> 另行跑 TS
以为 Turbopack "没有替代品"	仍可显式 <code>--webpack</code> 。强制 = 默认 + 高摩擦回退，不是禁止
以为 Vite 8 的收益在 dev	HMR 相比 Vite 7 基本没变 ，收益在生产构建与依赖预构建
以为换引擎只是"变快"	产物结构与体积是 另一回事 ，必须回归对比
忽略 Rolldown 的 ESM-only 与 Node 版本要求	老项目升级前先核对
忽略 Yarn PnP 兼容问题	Windows 上尤其注意

4.5 成熟度矩阵与投入建议

判断原则

看折旧率，不看新旧。

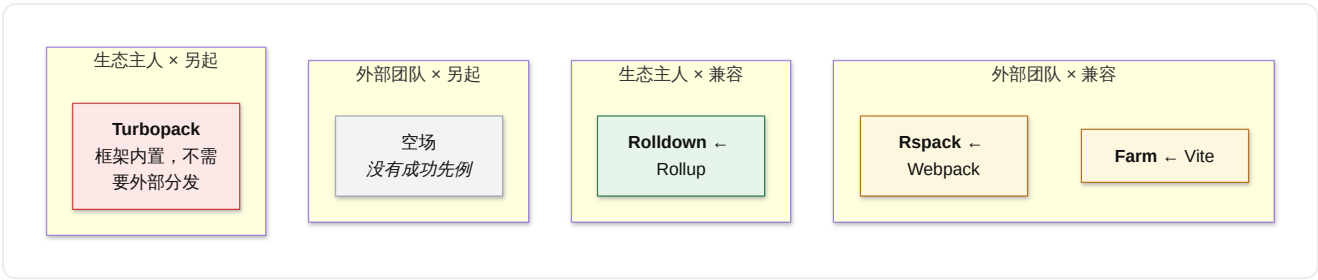
三条判据：

判据	含义
① 层越低，折旧越慢	转译层（SWC/esbuild/Oxc）已定型；打包层还在洗牌 —— 背配置的折旧极快
② 是否已在默认路径上	在默认路径上的，重点是从"用过"变成"知道它为什么在那儿"
③ 是否兼容存量生态	兼容的，收益可立刻兑现；另起的，学了只对一个框架有效

成熟度矩阵

工具	层	成熟度	投入建议
SWC / esbuild	转译 · 压缩	事实标准	已在用 —— 不必特意学，需知其为何快
Oxc / Lightning CSS	解析 · 转换 · CSS	上升中	了解即可
Rspack	打包	生产可用 (1.x)	值得学 —— 存量提速首选
Rolldown	打包	1.0 稳定	值得学 —— Vite 8 已默认
Turbopack	打包 (框架内置)	Next.js 16 默认	仅 Next.js 用户需要
Farm	打包	可用但生态小	观察

四条路线的 2×2 全景



注意第 Q3 象限是空的 —— "外部团队 + 另起生态"没有成功先例。这不是巧合，是双边网络效应在起作用 (§4.3)。

本章的最终结论

模块 1 (第 2 章) 的知识折旧最慢。

依赖图、Loader/Plugin 分工、Tree Shaking 两阶段、代码分割三策略、dev/prod 差异 —— 在 Rspack、Rolldown、Turbopack 里**全都还在**，只是换了执行者。

这是整章最重要的产出：它告诉你该把学习时间投在哪。

常见的坑

坑	正确认知
按"谁最新"决定学什么	判据是折旧率 + 默认路径 + 兼容对象
花时间背 Rspack 的配置项默认值	学"为什么兼容、迁移要核对什么"值；背每个字段默认值不值
以为 Turbopack 通用	框架内置
以为换引擎只是"变快"	产物结构与体积要回归对比
现在就在新项目押注 Farm	生态未成型，可能打水漂

4.6 本章常见误区 (14 条)

#	误区	正确认知
1	以为 esbuild / SWC 能做类型检查	只剥离类型标注，不验证类型
2	以为 vite build 会因类型错误失败	默认不会，需显式挂 tsc --noEmit
3	以为 SWC 能完全替代 Babel	插件体系不同 (Rust/WASM vs JS)
4	以为 esbuild 能直接替代 Webpack 打包	打包能力与生态都不具备
5	以为"兼容"等于 100% 无痛迁移	Rspack 是"几乎全部 loader + top 50 插件 >80%"
6	以为另造生态只是"成本高"	是冷启动起不来 (双边网络效应)
7	以为 Turbopack 可以通用	框架内置，离开 Next.js 无意义
8	以为 Turbopack 会做类型检查	next build 另行跑 TS
9	以为 Turbopack"没有替代品"	仍可 --webpack。强制 = 默认 + 高摩擦回退
10	以为 Vite 8 的收益在 dev	HMR 相比 Vite 7 基本没变，收益在生产构建与预构建
11	以为换引擎只是"变快"	产物结构与体积是另一回事，必须回归对比
12	按"谁最新"决定学什么	判据是折旧率 + 默认路径 + 兼容对象
13	忽略 Rolldown 的 ESM-only 与 Node 版本要求	升级前先核对
14	说"前端构建是纯单线程所以慢"	Webpack 5 的 Terser 默认 parallel: true，并非纯单线程

下一章：第 5 章 · 构建性能优化 —— 把"更快"变成一个可以验证的结论。

第 5 章 · 构建性能优化

主线索：

优化的前提是「能证明」。不能证明"问题在这里"就动手，本质是在赌。而能证明的最小单位，是一个绑定好指标与前提的数字。

5.1 优化的前提：先有指标，再有结论

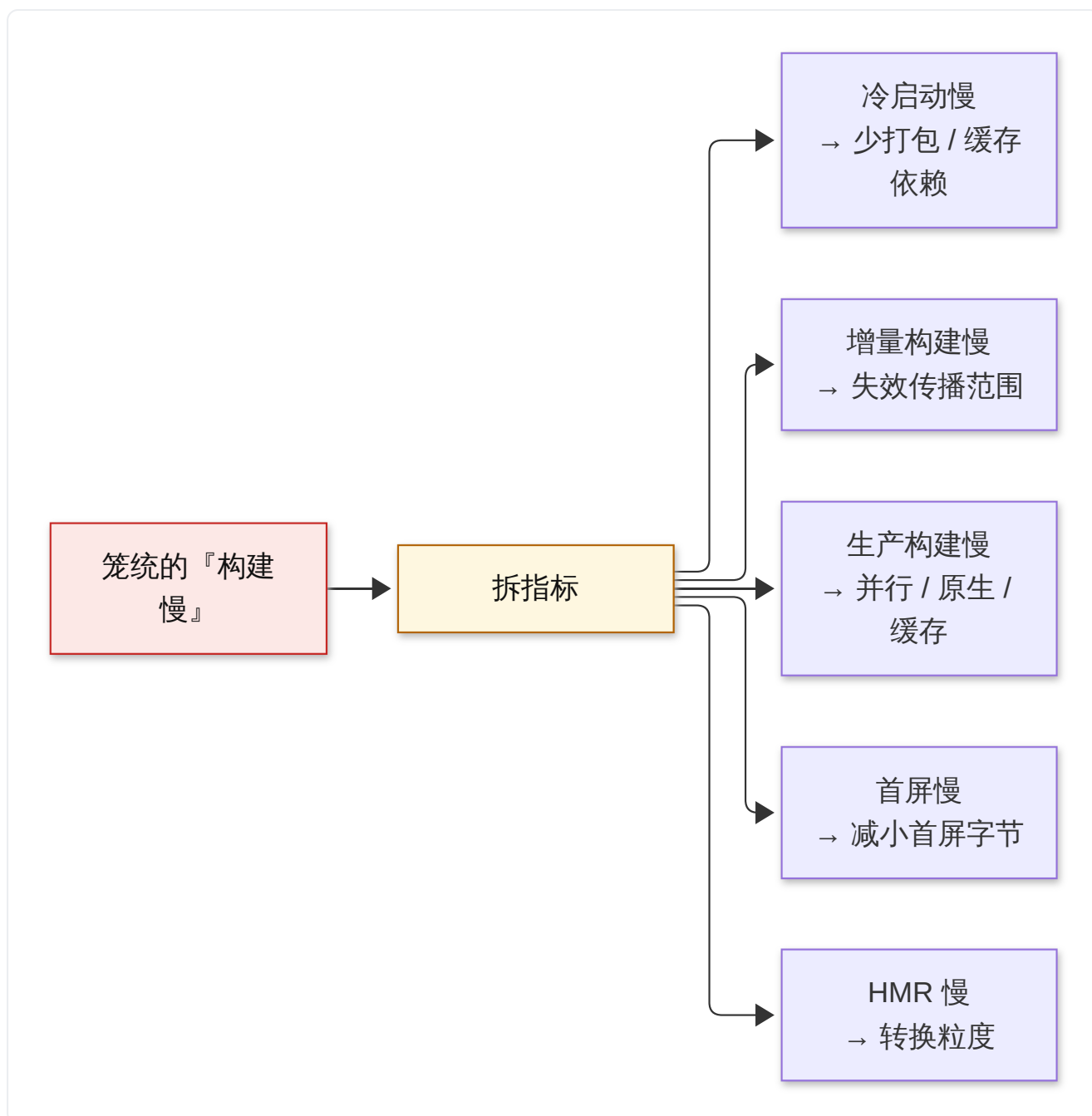
一句话

"更快"不是结论，"哪个指标快了、在什么前提下、怎么测的"才是。

四个必须分开的指标

指标	定义	谁在等它
冷启动	dev server 从启动到可用	开发者（一天几次）
增量构建	改一个文件到产物更新	开发者（高频）
HMR 响应	改一个文件到浏览器完成替换	开发者（最高频）
生产构建	从零到 <code>dist/</code> 完成	CI（一天几次到几十次）
首屏加载	用户从请求到可用	用户（最该优化，但最容易被混淆）

为什么要分这么细：因为这五个指标的优化手段几乎不重叠。



不拆指标的后果：你可能会为了优化 CI 上的生产构建，去改 dev server 的配置 —— 完全无效，但看起来"做了优化"。

三类最容易混淆的"快"

说法	实际说的是	常见误用
"Vite 快"	dev 冷启动 + HMR 快	被用来论证"生产构建也快" (✗)
"构建快"	通常指 生产构建	被用来论证"开发体验好" (✗)
"首屏快"	用户侧加载	被用来论证"构建工具选得好" (✗，两者机制不同)

判据：听到"更快"，先问三个问题 —— 哪个阶段？什么前提？怎么测的？

5.2 三类耗时来源与各自的优化手段

构建耗时按**来源**分三类，每类的解法不同。

耗时来源	典型占比	优化手段
① 模块转换	大	换原生转译器 (SWC / esbuild / Oxc)
② 依赖解析与打包	大 (大项目尤甚)	换原生打包器 (Rspack / Rolldown) ；减少模块数量
③ 产物优化 (压缩 / 混淆 / 摇树)	中	换原生压缩器；并行化
④ 插件执行	不定，可能是隐藏大头	精简插件；避免 JS ↔ 原生往返
⑤ I/O 与文件系统	小到中	缓存；SSD

一个常被忽略的点：插件是隐藏的大头

Rspack 推荐用 `builtin:swc-loader` 而不是 `babel-loader`，理由正是这个 —— 在 **JS** 侧跑 **loader**，数据要序列化传给 **Rust** 再传回来，这个往返开销在大量文件上很可观。

同理：Farm 的 JS 插件**必须配** `filters`，不配的话每个文件都要跨语言往返一次，原生带来的优势直接被吃掉。

通用判据：跨语言 / 跨进程边界的调用，要么批量化，要么加上过滤条件。

减少模块数量为什么有效

因为构建的核心工作量与模块数量成正比（每个模块都要 `resolve` → `load` → `transform` → `parse`）。所以：

- `lodash` 全量导入 → `lodash-es/debounce` 按路径导入，不只是摇树问题，也直接减少被处理的模块数。
- 依赖预构建把 `lodash-es` 的 600+ 内部模块合并成 1 个，既省请求，也省 **dev** 下的处理量。

5.3 缓存：三层结构与各自的失效条件

缓存是投入产出比最高的一类优化 —— 它不改变构建的机制，只是让重复的部分不重做。

三层缓存

层	位置 / 机制	失效条件
① 构建工具缓存	Webpack <code>cache: { type: 'filesystem' }</code>	配置变化 / 版本变化
② 依赖预构建缓存	<code>node_modules/.vite</code>	lockfile / vite.config / NODE_ENV 变化
③ 浏览器 HTTP 缓存	产物文件名 hash + 强缓存头	文件名变化即失效
④ 系统层	OS 文件缓存	内存压力

Webpack 持久化缓存

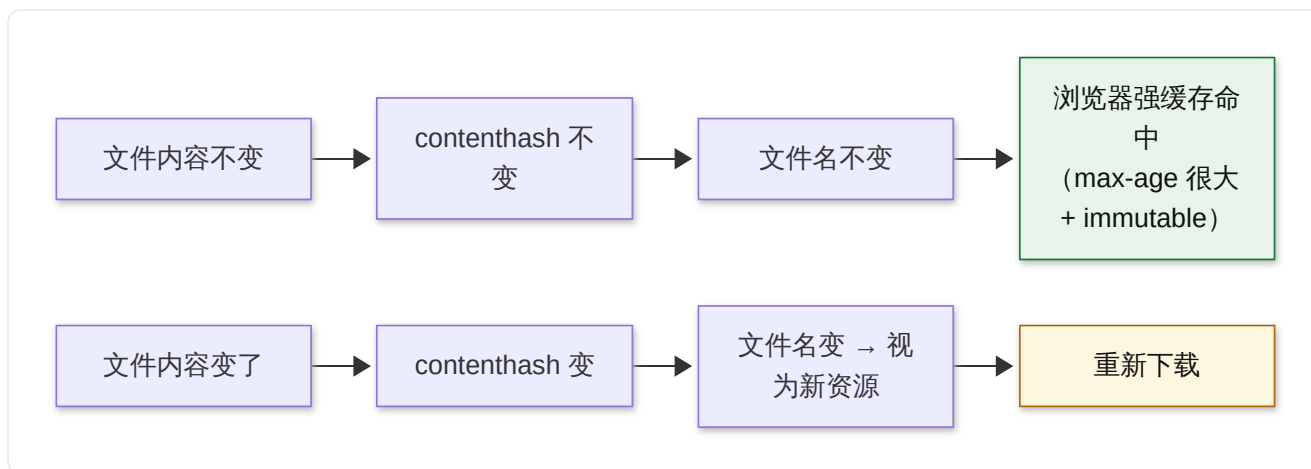
```
module.exports = {
  cache: {
    type: 'filesystem',
    buildDependencies: {
      config: [__filename],    // config 变了才让缓存失效
    },
  },
};
```

关键点：`buildDependencies` 不配的话，改了配置文件但缓存不失效，会出现"改了配置没生效"的诡异现象。

效果：二次构建接近增量。这是讨论"Webpack 慢"时最容易被忽略的一项 —— 拿没开缓存的 Webpack 去比是不公平的对比。

产物 hash 与 HTTP 缓存的配合

机制链条：



为什么这套机制能成立：因为内容变了文件名就变，所以"同一个文件名"等价于"同一份内容"——浏览器可以放心长期强缓存，不需要每次去问服务端。

所以 **hash 稳定性不是可选项**：一旦 hash 不稳（只改文案却导致 vendor 文件名变化），**整套长效缓存就失效了**。这就是 §2.2 「缓存三件套」必须配齐的原因。

`index.html` 这类入口文件**不应该**被强缓存（否则用户拿不到新的资源引用），但它引用的带 hash 的 JS/CSS **应该**被强缓存。这条策略的具体配置涉及服务端 / CDN，课程未展开。

5.4 产物层面的优化：体积与结构

构建快 ≠ 用户加载快。产物优化针对的是后者。

四条路径

路径	手段	效果
减小总量	Tree Shaking (§2.4)、换更小的替代库	减少下载字节
减小首屏	动态 import、路由懒加载 (§2.5)	首屏字节减少，总量不变
提高缓存命中	<code>contenthash</code> + 缓存三件套 (§2.2)	二次访问几乎零下载
优化结构	合理的 <code>splitChunks</code> / <code>manualChunks</code> 分组	并行下载 + 复用

三个必须记住的"不"

常见误解	事实
"用了动态 import，包变小了"	✗ 总量没变，只是首屏要下载的少了
"配了 Tree Shaking，产物一定小"	✗ 需要 ESM + 生产模式 + 压缩器三件事齐备，且库本身要写得好
"chunk 拆得越细越好"	✗ 拆太细导致请求数爆炸，反而变慢

一个实用判据

先查成分，再改配置。

看到 `vendor.js` 1.2MB，正确的第一个动作不是去写 `manualChunks`，而是：

- 1. 生成构建元数据 (`stats` / 产物分析报告)
- 2. 看这个 chunk 包含哪些模块、按体积排序
- 3. 定位是谁把它们打进了同一个 **chunk** (是我的 `manualChunks` 规则？默认分组？还是入口静态 import 了一个大依赖？)
- 4. 判断它是否在首屏路径上

先查成分和来源，不是先读代码，更不是先改配置。

(一个附加易错点：**1.2MB** 的生产产物是压缩混淆过的，不该去"读"它 —— 该读的是构建元数据。)

5.5 并行与增量：原生工具与增量引擎

并行

层	手段
转译	原生实现 (SWC / esbuild / Oxc) 真多核
压缩	Terser <code>parallel: true</code> (worker 线程) / 换原生压缩器
打包	依赖图是 串行约束 ，并行空间最小 (见 §4.1)

注意：并行不是万能的。**依赖图的串行约束**决定了打包层的并行上限 —— 这也是"打包层原生化最难"的根本原因。

增量

工具	增量机制
Webpack	<code>cache: { type: 'filesystem' }</code> 持久化缓存
Vite dev	依赖预构建缓存 (<code>node_modules/.vite</code>) + 单文件转换
Vite build (Rolldown)	<code>output.codeSplitting</code>
Turbopack	函数级增量 + 落盘 (16.1 dev 落盘, 16.3 build 默认开)
Rolldown	统一的 dev/prod 引擎, dev 侧沿依赖图做 局部化失效传播

增量粒度的意义：粒度越细，改动后需要重算的范围越小。

- 模块级 → 改 1 个文件重编译 1 个模块
- **函数级** (Turbopack) → 只重算受影响的函数
- **局部化失效传播** (Vite/Rolldown) → 只让受影响的模块失效，不整图重建

一致性：一个容易被忽略的"性能"维度

dev 与 **prod** 用两套引擎时，会制造一类特殊的"性能问题"：不是慢，是行为不一致导致的排查成本。

esbuild(dev) 与 Rollup(build) 是两套模块解析、两套转换、两套 tree-shaking 语义 → 同一份代码"本地能跑、build 后行为变了"。

Rolldown 最值钱的收益不是速度，是 **parity** —— 因为速度有替代方案（增量缓存、换机器），一致性没有（详见 §4.4）。

5.6 优化验证：六步测量法

不做测量的优化，等于没做。

六步法

① 拆指标
冷构建 / 增量 /
HMR / 首屏



② 固定变量
同机器、同代码
规模



③ 清三层缓存
.cache / .vite / 浏
览器无痕



④ 各跑 5 次，取
中位数

⑤ 记录元数据
行数 / 依赖数 /
插件数

⑥ 出带前提的结
论

步骤	为什么必须做
① 拆指标	一次只测一个指标，否则测到的是混合结果
② 固定变量	机器性能和代码规模是最强的两个干扰项
③ 清三层缓存	三层是：构建缓存（ <code>node_modules/.cache</code> 、 <code>node_modules/.vite</code> ）、浏览器强缓存（无痕）、系统层
④ 跑 5 次取中位数	单次测的是系统负载，不是构建性能；平均值受异常值影响，中位数更稳
⑤ 记录元数据	没有元数据的倍数无法复现，也无法对比
⑥ 带前提的结论	这是唯一的产出形式

判据：什么叫"带前提"

不合格：

"升级到 Vite 8 后构建提速约 6 倍。"

合格：

"在 X 机器（8 核 / 16G）上，对一个约 N 行代码、M 个依赖、K 个插件的项目，清空 `node_modules/.cache`、`node_modules/.vite` 与浏览器缓存后，生产构建耗时从 A 秒降到 B 秒（各跑 5 次取中位数）。"

差在哪：机器、代码规模、依赖数、清理了哪些缓存、跑了几次、取的中位数 —— 缺任何一项，这个倍数就不可复现。

一条诊断经验

90s → 12s 这类惊人的差异，有可能只是"首次冷构建 vs 有持久化缓存的第二次构建"的差异。

所以第三步（清缓存）不能省。不清缓存做的对比，测的是缓存命中率，不是构建器的性能。

5.7 本章常见误区

#	误区	正确认知
1	不拆指标就谈"构建慢"	五个指标，优化手段几乎不重叠
2	拿 dev 的结论说 build	"Vite 快"只属于 dev
3	单次测量下结论	单次测的是 系统负载 ，不是构建性能
4	不清缓存就做对比	测到的是缓存命中率差，不是构建器性能差
5	报倍数不带前提	无法复现的倍数等于没测
6	拿厂商数字当自己的结论	厂商数字是 起点 ，不是 判据
7	以为动态 import 能减小总体积	只减小首屏
8	以为配了 Tree Shaking 产物一定小	需 ESM + 生产 + 压缩器，且库写得好
9	chunk 拆得越细越好	请求数爆炸，反而变慢
10	看到大 chunk 直接改配置	先查成分和来源
11	去"读"压缩后的生产产物	该读 构建元数据
12	比较时一方没开缓存	不公平对比
13	只优化 CI 构建，忽略 HMR	HMR 是开发者最高频的等待
14	忽略插件带来的跨语言往返开销	用 <code>builtin:swc-loader</code> / 给 JS 插件配 <code>filters</code>

下一章：第 6 章 · AI 辅助构建工程 —— 当构建工具多了一类用户。

第 6 章 · AI 辅助构建工程

主线索：

AI 辅助构建的主战场，不是"让 AI 帮你写配置"，而是"AI 成了构建工具的用户"。

6.1 视角转换：AI 成了构建工具的用户

一句官方原话

Cloudflare 在 2026 年 6 月宣布 VoidZero 加入 Cloudflare 的公告里写道：

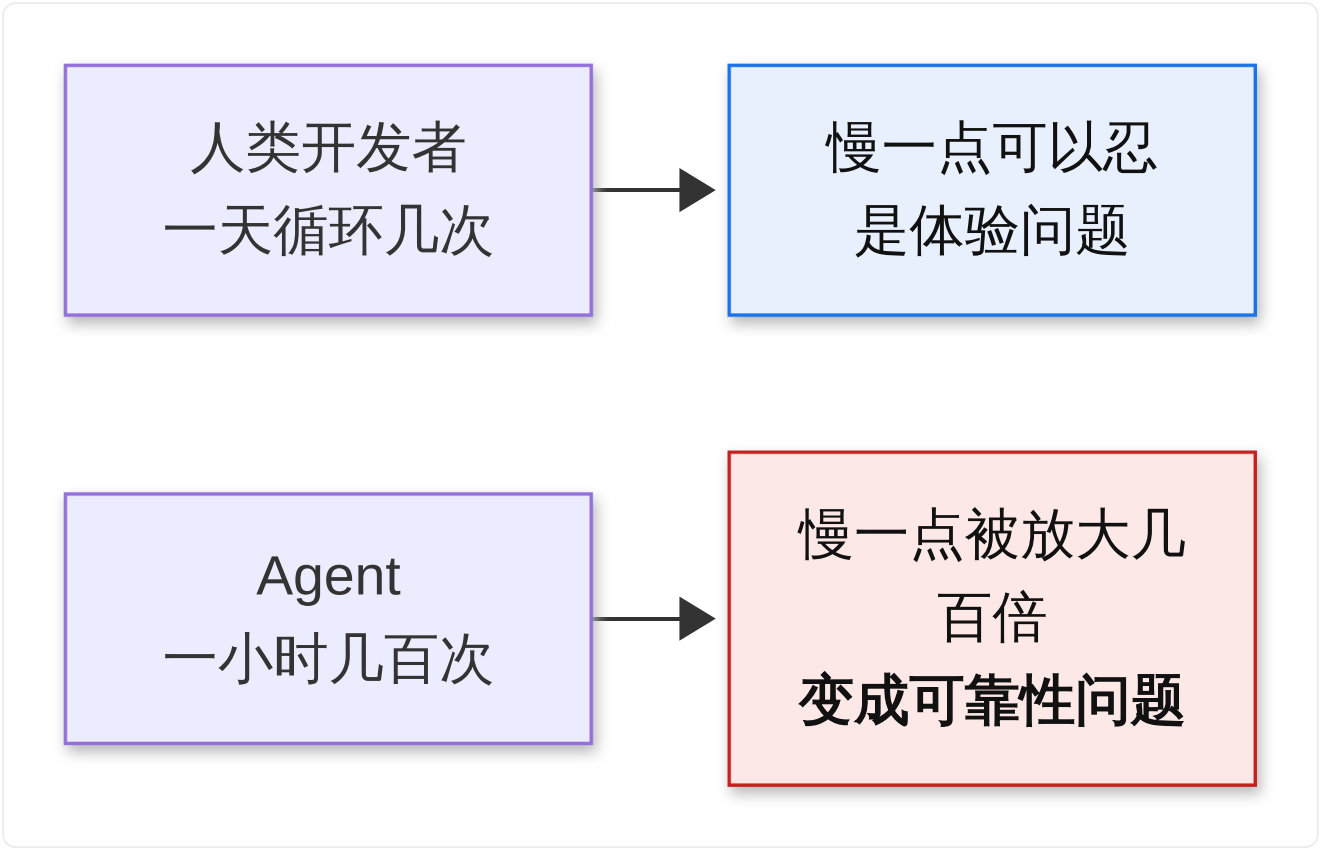
"Developers used to be the only users of dev servers, bundlers, linters, formatters, and CLIs. That is no longer true: agents are using them too, constantly."

(开发者曾经是 dev server、打包器、linter、formatter 和 CLI 的唯一用户。这不再成立：agent 也在用它们，而且是持续在用。)

这句话为什么重要：它把"AI 辅助构建"这个话题的主语换掉了。

- 旧框架：AI 是一个助手，帮人类写配置、查文档。
- 新框架：AI 是一个用户，它会直接调用 dev server、打包器、linter —— 而且频率远高于人类。

频率差带来的性质变化



Cloudflare 的表述是：

"Speed was always a developer quality-of-life improvement. **For agents, speed is a reliability requirement.**"

推论：同一条"构建很慢"的事实，对人和对 agent 的性质不同。

对象	"构建慢"的性质
人类开发者	体验问题 —— 等待时可以摸鱼、可以并行做别的事
Agent	可靠性问题 —— 迭代次数暴增，慢会被放大几百倍，直接影响能不能完成任务

这条推论的实际含义：它改变的是优先级排序，不是"该学什么新技能"。速度这件事，从"nice to have"变成了"必选项"。

Cloudflare 列出的 agent 对工具的要求

要求	说明
快的构建、测试、lint / format	三者从"偶发检查"变成持续护栏
清晰结构化的错误	agent 必须程序化读取并据此行动
一致的 CLI	输出或 flag 的小不一致，会导致 agent 大幅绕路

"三者成为护栏而非偶发检查"这句值得注意 —— 人类跑 lint 是提交前跑一次；agent 是每改一次都跑。频率变了，对延迟的容忍度也变了。

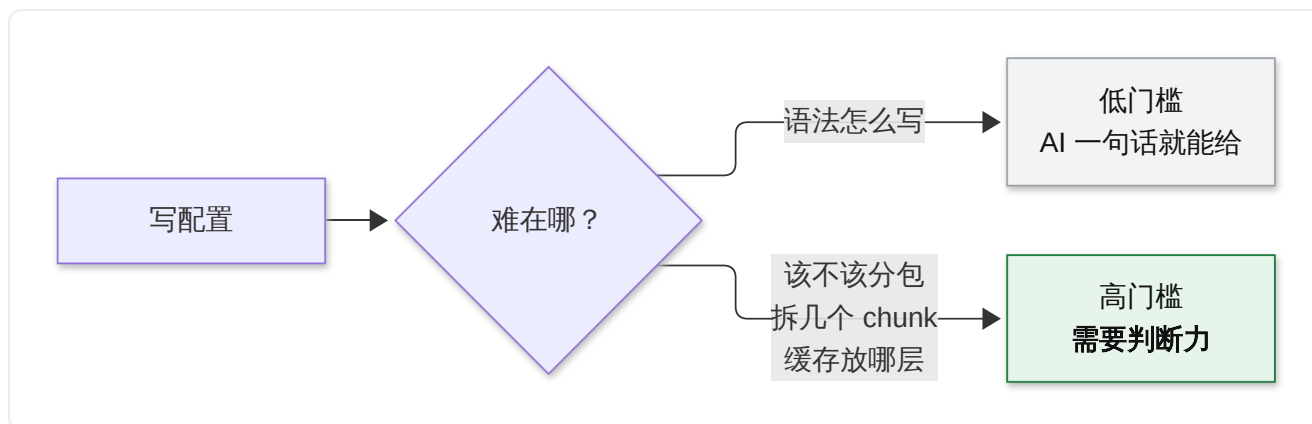
6.2 三层价值矩阵

这是本节的核心框架。

层次	痛点	成熟度	价值
① AI 当配置助手	写配置的体力活	成熟	低 —— 一锤子买卖；门槛是判断力（该不该分包）不是语法
② 构建输出对 AI 友好	agent 读不懂 build log / 看不见浏览器 / 不知道运行时状态 / 被 CLI 不一致绊倒	早期，无标准	中高 —— 唯一需要主动决策的部分
③ AI 成为构建工具的用户	agent 迭代次数暴增	已成立	极高，但改变的是优先级排序，不是该学的技能

为什么层次①的价值低

因为配置这件事的门槛从来不在语法。



AI 能帮你写 `splitChunks` 的语法，但它不知道你的项目该不该拆、按什么维度拆。而后者才是真正费脑子的部分。

结论：让 AI 写配置是一锤子买卖——写完就不再需要。它的价值上限，就是省下查文档的那几分钟。

为什么层次②值得主动决策

这是三层里唯一需要学习者主动做决定的部分，因为：

- 它没有标准（早期阶段）——没人告诉你“正确的做法是什么”
- 它需要你自己动手改（不是装个插件就完事）
- 它直接影响你未来和 AI 协作的效率

为什么层次③“已成立”却不需要学新技能

因为它的结论是排序变化，不是知识增加：

- 你不需要学“专门给 AI 用的构建工具”
- 你需要知道“速度现在是硬需求”，从而在选择工具 / 配置缓存时把这一项权重调高

6.3 实践形态：结构化诊断与运行时感知

层次②的具体做法，目前有两种形态。

形态一：把构建输出变成结构化诊断

问题：一份完整的 build log 一次几万 token。直接塞给模型，既贵又全是噪声。

做法：把日志预处理成结构化诊断：

```
{
  "file": "src/components/Chart.vue",
  "line": 42,
  "column": 8,
  "severity": "error",
  "message": "Property 'value' does not exist on type 'Props'",
  "code": "TS2339"
}
```

三个关键设计：

设计	效果
原始日志不进上下文窗口	只把结构化结果给模型
两步式	<code>build()</code> 先返回紧凑摘要；需要时才 <code>get_errors()</code> 取详情
字段固定	agent 可以程序化读取并据此行动

两步式的价值：**一次 build 可能产生几百条信息，但真正需要修的可能只有 3 条。**先给摘要，让 agent 判断要不要深入 —— 这是在不牺牲信息量的前提下省 token。

已有实践：`@paretools/build`、`cpp-build-mcp` 一类 MCP build server 就是这个模式。

形态二：让 AI 感知运行时

问题：**AI** 看不见浏览器。DOM、console 输出、computed style、组件树，这些都在运行时，源码里没有。

做法：以 dev server 中间件 / 浏览器代理的形式，把运行时的东西通过 **MCP** 暴露给 agent：

暴露的内容
DOM 结构
组件树
computed style
编译后的模块图
路由状态
服务端日志

已有实践：Frontman、Stagewise 一类工具。

一个官方动作：`server.forwardConsole`

Vite 8 提供 `server.forwardConsole`：把客户端 **console** 转发到终端。

官方文档明确说明："在使用 **AI** 编码代理时尤为实用"。

为什么这个功能能进官方：因为"AI 看不见浏览器"是一个被官方承认的痛点 —— 它不再只是社区自发的补丁。

一个技术之外的原因：训练数据占比

"Vite 被 **agent** 选中"这件事，有一条非技术的解释。

数据点：`@cloudflare/vite-plugin` 周下载量约 **1400 万**，约为 Vite 自身安装量的 **10%**。
Cloudflare 的解释是：

"Agent-coded applications are choosing Vite."

Vite 胜出的原因：快、文档好、兼容广，而且在大模型训练数据中出现得多。

这条呼应的是第 4 章的「默认即权力」—— 只是从"框架的默认值"换成了"模型训练数据的默认偏好"。当 AI 成为用户时，训练数据里的出现频率，成了一种新的默认。

6.4 风险与边界

四个明确的边界

边界	说明
AI 看不见运行时	DOM / console 在运行时，源码里没有 —— 需要专门暴露
AI 不理解你的业务约束	该不该拆包、要不要兼容旧浏览器，取决于业务，不取决于技术
结构化诊断是"有损"的	预处理会丢信息，需要保留取详情的通道（两步式）
训练数据滞后	模型对新工具的认知可能落后于官方文档（这正是本章标注核验日期的原因）

一条实践建议

别把原始日志塞给 AI。这不是"省 token"的问题，是信噪比的问题 —— 几万 token 的日志里，真正有用的可能只有几行。

正确顺序：

- 构建 → 结构化诊断 (file/line/severity/message/code)
- 给 AI 摘要
- AI 判断需要详情时再取
- 定位到具体文件的具体位置

6.5 本章常见误区

#	误区	正确认知
1	以为 AI 辅助构建 = 让 AI 写配置	主战场是"AI 成为构建工具的用户"
2	把原始 build log 整个塞给 AI	一次几万 token，应转结构化诊断
3	以为 AI 看得见浏览器	DOM / console 在运行时，源码里没有
4	以为 AI 辅助只是"体验优化"	对 agent，速度是可靠性要求
5	以为要给 AI 专门学一套新工具	变的是优先级排序，不是知识清单
6	忽略 CLI 一致性	输出或 flag 的小不一致，会让 agent 大幅绕路
7	用"生态优势"解释 Vite 被 agent 选中	更具体、可验证的机制是训练数据中的出现频率

一个术语澄清

课程中出现过一个典型偏差：用短语或案例代替概念命名。这里给出正确说法：

曾出现的说法	问题	准确的表述
"生态优势"	是标签，不是机制	Vite 在大模型训练数据里出现得多 → 这是一种生态位优势，但机制是"训练数据占比"
"TurboPack"（作为解释）	指向了案例，没有命名概念	正确概念是第 4 章的「默认即权力」（强制 = 默认 + 高摩擦回退）

通用规则：回答抽象问题时，先命名概念，再举例 —— 顺序不能反。命名之后，这句话才可以证伪。

下一章：第 7 章 · 工程实践与选型 —— 把知识变成路径和决策。

第 7 章 · 工程实践与选型

主线索：

按能力排，不按工具排。跟着工具换代走，每次换工具都要重学配置；按能力排，只需学一遍。

7.1 顺序按能力排，不按工具排

两种学习顺序

路线	顺序	代价
按工具排（多数人的默认）	Webpack → Vite → Rspack →	每次换工具都要重学一遍配置，且旧知识迅速报废
按能力排	建立图 → 会用默认值 → 会读产物 → 会算账 → 会迁移	学一遍，跨工具通用

为什么"按工具排"是陷阱：因为工具名会变，机制不会变。

依赖图、Loader/Plugin 分工、Tree Shaking 两阶段、代码分割三策略 —— 这些在 Rspack、Rolldown、Turbopack 里全都还在，只是换了执行者（\$4.5 的结论）。

你花时间背的 `optimization.splitChunks` 字段默认值会过期；你理解的"为什么需要抽公共包"不会。

7.2 五个台阶与判定标准

台阶表

台阶	目标	学什么	判据	常见跑偏
1 建立图	通用心智模型	依赖图 / Loader·Plugin / Tree Shaking / 代码分割	说清「源码 → 依赖图 → chunk → 产物」	背配置项
2 会用默认值	跑通真实项目	<code>vite.config.ts</code> 最小配置、 <code>.env</code> + <code>import.meta.env</code> 、 <code>build.sourcemap</code> 、 <code>server.proxy</code>	每条配置项能说出「为什么存在」	一上手就装一堆插件
3 会读产物★	从使用者变成工程化	读 <code>stats</code> / 产物分析、chunk 划分、hash 稳定性	改一行代码能预测哪些产物 <code>hash</code> 会变；错了能解释为什么	只看体积，不看来源
4 会算账	结论站得住	六步测量法、三层缓存、指标拆分	听到「变快了」能拆出绑定的指标与前提	拿厂商数字当结论
5 会迁移选型	对存量下判断	插件清单核对、产物回归对比、回退路径设计	给得出「能不能迁、要多久、翻车了怎么办」	先比技术优劣，不问痛点

1 · 建立图
能说清主线



2 · 会用默认值
知道每行配置为
什么在



3 · 会读产物 ★
从使用者变成工
程化
判据：我预测得
准



4 · 会算账
结论带前提



5 · 会迁移选型
能对存量下判断

第 3 级为什么是分水岭

前几级的判据是「我说得清」；第 3 级的判据是「我预测得准」。

级别	判据的性质
1、2 级	陈述性 —— 能复述、能解释
3 级	预测性 —— 能对未发生的事给出可验证的判断
4、5 级	预测性 + 决策性

"改一行代码能预测哪些 hash 会变"这个判据的威力：它一次性检验了「依赖图 + chunk 来源 + hash 计算范围 + 缓存三件套」四块知识。你预测错了，就一定有一块没真正理解。

不读产物，所有优化都是猜的。

依赖关系

- 1 → 2 → 3 必须串行（后面的判据依赖前面的能力）。
- 4、5 可以在真实项目需求出现时再补。
- 不要为了"学完"而学。

7.3 每阶段的动手项目

配项目的两条标准

① 项目要小到一次做完 ② 必须能自证过关（有判据，不靠感觉）

"能自证"是更重要那条 —— 没有判据的项目，做完也不知道自己会不会。

五个项目

台阶	动手项目	规模	过关判据
1	手写迷你打包器	≤100 行	跑通「两个文件互相 import」；能解释共享模块为何只处理一次
2	零插件 Vite 项目	1 小时	每行配置能说出「为什么存在」；删掉配置项目仍能跑
3	hash 预测游戏（三页面项目）	半天	改之前先写预测，连续 5 次正确
4	构建基准测量（六步法）	半天	结论每句都带前提
5	迁移评估一页纸	半天	结论含「翻车了怎么办」

每条的细节

台阶 1 · 手写迷你打包器

- 目的不是做一个能用的打包器，是验证依赖图不是背的。
- 最小实现：读入口 → 找 import → 递归解析 → 拼成一个函数表 → 输出一个文件。
- 判据里的"共享模块为何只处理一次"，直指 §2.1 的"图 ≠ 树" —— 你的迷你实现必须自己处理这个去重，才能真正体会。

台阶 2 · 零插件 Vite 项目

- 反直觉动作：故意删掉所有配置，看还能不能跑。
- 你会发现大部分配置本来就不需要 —— 这正是 §3.3 「Vite 默认值覆盖了语言层」的体感验证。

台阶 3 · hash 预测游戏（本书最推荐的一项）

- 准备一个三页面的小项目。
- 分别做四种改动，每次改前先写下预测（哪些产物的文件名会变），改完对照：

改动类型	预测时应考虑
改业务代码	该模块所在 chunk 变化
改依赖版本	相关 vendor chunk 变化
改 CSS	样式产物 + 引用它的 chunk
只改注释	应该什么都不变吗？ —— 如果 hash 变了，说明缓存三件套没配齐

- 加分项：装 rollup-plugin-visualizer，看 vendor chunk 里的模块排行 —— 这一步顺手把"会读产物"练了。

台阶 4 · 构建基准测量

- 按 §5.6 的六步法跑一遍，产出一段带前提的结论文字。
- 判据不是"数据好看"，是"每句都带前提"。

台阶 5 · 迁移评估一页纸

- 五段结构：

段落	内容
① 依赖清单	现有的 loader / plugin 清单
② 硬阻塞	有没有用到不支持的能力
③ 产物回归方案	怎么证明迁移后产物没变坏
④ 回退路径	翻车了怎么退回去
⑤ 结论	能不能迁、要多久

- 判据是第 ④ 段 —— 没有回退路径的是假报告。

一个自检问题

五个项目里，哪个最容易被自己"糊过去"（明面做了、实际没拿到能力）？

答案是台阶 2。因为它最容易"跑起来就算完成" —— 建个项目、`npm run dev` 能开，就以为过关了。但判据是"每行配置能说出为什么存在"+"删掉配置还能跑" —— 加这两个判据，就是堵住糊弄的办法。

通用规则：先写判据，再动手。

7.4 文档与源码的阅读切入点

文档：用来查证，不是用来读完

两种失败的读法：

- 从第一页读到最后一页（读不完，且大部分用不上）
- 只搜 API（知道字段，不知道为什么）

三种正确读法：

读法	用法
① 查 API	只搜，不读目录
② 读机制页	只有少数几页值得逐字读：Webpack Concepts 、Vite Why Vite 与 Dependency Pre-Bundling 、Rollup Plugin Development
③ 读迁移指南 ★	性价比最高 —— 写的是官方承认的差异，等于一份免费的对照表

为什么迁移指南性价比最高：因为它是官方替你总结好的差异清单。你不用自己发现"Rspack 和 Webpack 在哪些地方不一样" —— 官方文档里已经列出来了。Rspack 的 migration 页 + `rspack-compat` 仓库就是典型。

读文档的判据：

读到「已知限制 / deprecated」段落时，停下来精读。

那是坑的官方来源 —— 官方主动告诉你"这里有问题""这个要废弃了"，比任何第三方文章都权威。

源码：别从 `index.js` 读起

三个切入点：

切入点	门槛	能验证什么
① 产物反推	低	读 <code>dist/</code> 的 runtime 胶水代码 → 直接看到动态 import 与 HMR 的实现证据
② 插件钩子	中	挑最熟的官方插件，只读钩子函数体，看它接收什么、返回什么
③ 具体 bug	高	先搜报错字符串 → 顺调用栈往回读那一条路径

为什么推荐从产物反推开始：产物是你自己的代码生成的，你熟悉它应该长什么样。看到 runtime 里那段 `__webpack_require__` 的胶水代码，你能直接对应上 §2.2 讲的"运行时胶水污染 chunkhash"。

铁律：

只读你有一个具体问题的那条路径。漫读源码是最低效的学法。

两条纪律：

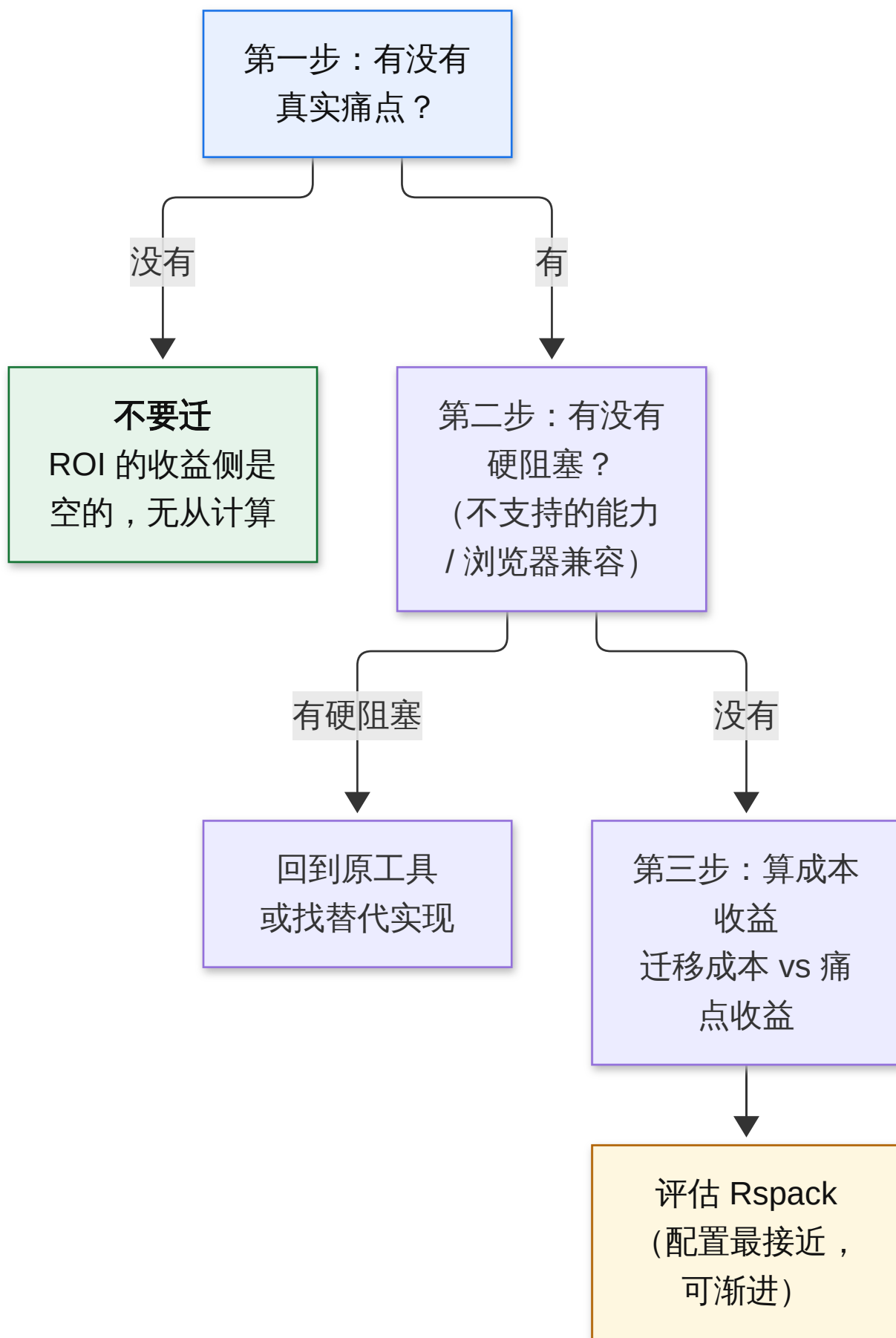
- 1. 先读类型定义 / 接口描述（地图），再读实现（地形）。
- 2. 读完必须留下一个可验证的结论。写不下「我验证了 X 是 Y」= 没读。

7.5 选型决策矩阵

本节是全书选型内容的唯一决策出口。§3.6 负责说明"为什么"，本节负责"怎么办"。

第一步：顺序比结论重要

决策顺序：先看有无迁移成本 → 再看有无硬阻塞 → 最后才比成本收益。不是先比技术优劣。



为什么第一步是"有没有痛点"：

没有痛点，**ROI** 的收益侧是空的 —— 你没法计算"获得了多少"，因为你没有要解决的问题。而迁移成本是实打实的（功能冻结、回归风险、CI 改造、下游产物影响）。

关于 **ROI** 的实践警告：ROI 标准在实践中很难用准 —— 收益是感受性的、隐性成本容易被低估（功能冻结期、回归风险、CI 改造、下游依赖产物结构的人）。所以把它当**思考框架**用，别当**计算公式**用。

场景对照表

情境	建议	理由
全新项目	Vite	约定完备、冷启动快、生态成熟
中小存量项目、无硬阻塞	保持 Webpack	迁移成本 > 收益
存量项目、 build 成瓶颈	评估 Rspack	配置接近 Webpack，可渐进迁移
必须深度定制产物结构	Rspack / Webpack	产物编排与干预点更强
多页 / 微前端	看生态，常落 Webpack / Rspack	对插件生态与产物控制要求高
需兼容旧浏览器	Webpack ，或 Vite + <code>@vitejs/plugin-legacy</code>	Vite dev 依赖原生 ESM
库 / 组件包	Vite library mode ，或 tsup / unbuild	需求单一，够用

三条原则

- 1. 新项目默认 **Vite**
- 2. 存量项目不轻易迁移
- 3. 只有在明确证据「**Vite** 不够用」时才换

组件库场景的硬坑

这一节值得单独记，因为它的需求与"应用"完全不同。

坑	正确做法
Vite 默认不生成 <code>.d.ts</code>	用 <code>vite-plugin-dts</code> ，或单独 <code>tsc --emitDeclarationOnly</code>
<code>build.lib.formats</code> 需显式声明	写 <code>['es', 'cjs']</code> (Vite 5 后默认只出 ESM)
peer 依赖被打进产物	必须 <code>rollupOptions.external</code> 掉 (否则产物里带一份 React)
入口字段没对齐	<code>package.json</code> 的 <code>main</code> / <code>module</code> / <code>types</code> / <code>exports</code> 要手动对齐产物

为什么"组件库选 Vite"的理由不能照抄"应用选 Vite"的理由：

应用关注	组件库关注
冷启动快不快 ❌ 不相关	产物格式 (ESM + CJS)
生态成不成熟 ⚠️ 弱相关	类型声明 (<code>.d.ts</code>)
—	外部依赖不打包 (<code>external</code> peer)
—	入口字段对齐 (<code>main/module/types/exports</code>)

组件库在 CI 里跑 build，没有 dev server —— 所以"冷启动快"对它几乎没有意义。这是"复用上一个场景的答案"最典型的翻车现场。

通用动作：回答前先问 —— 这个场景和上一个场景，哪里不一样？

7.6 迁移评估：一页纸模板

判断迁移该不该做，写一页纸就够了。写不满一页说明想得不清楚；超过一页说明在凑字数。

<项目名> 构建工具迁移评估

1. 现状与痛点（必须有具体数字）

- 当前工具与版本：
- 生产构建耗时：____ 秒（N 次中位数，机器规格：____，代码规模：____ 行 / ____ 依赖）
- 痛点场景：CI 等待 / 本地反馈慢 / 其他

2. 依赖清单核对

现有 loader / plugin 目标工具是否支持 替代方案
--- --- ---

3. 硬阻塞

- 浏览器兼容要求：
- 是否使用目标工具不支持的能力：
- 若有阻塞 → 结论：不迁移 / 换方案

4. 产物回归方案（怎么证明没变坏）

- 产物结构与体积对比方式：
- 验收判据：
- 灰度 / 双跑方式：

5. 回退路径（翻车了怎么办）

- 回退触发条件：
- 回退动作：
- 回退耗时：

6. 结论

- 能不能迁：
- 预计耗时：
- 前置条件：

第 5 段是关键。 没有回退路径的评估报告是**假报告**——它假设了迁移一定成功，而这在工程上不成立。

第 1 段也必须带数字。 "构建很慢"不是痛点，"生产构建 4 分 20 秒、每次 CI 阻塞发布"才是。

7.7 本章常见误区（18 条）

#	误区	正确认知
1	按工具换代排学习顺序	按能力排，配置只学一遍
2	把「能跑起来」当学会了	判据是「你能预测什么」
3	跳过第 3 级（读产物）	不读产物，所有优化都是猜的
4	第 2 级就开始装插件	默认值没吃透 = 不知道基线
5	拿厂商基准数字当自己的结论	厂商数字是起点，不是判据
6	以为五台阶必须全部走完	1→2→3 串行；4、5 按需补
7	项目做得太大	目的是验证概念，不是做产品
8	没有判据就动手	先写判据，再动手
9	台阶 1 跳过不写	它的价值是验证依赖图不是背的
10	台阶 3 只看体积数字	hash 变化才是因果
11	台阶 4 只跑一次	单次测的是系统负载，不是构建性能
12	台阶 5 写了「能迁」就收工	没有回退路径的是假报告
13	想「把文档读完」	文档是字典，不是教材
14	只读 API 不读机制页	知道字段，不知道为什么
15	忽略迁移指南	官方承认的差异清单，白送的对照表
16	从 <code>index.js</code> 漫读源码	先有问题，再读源码
17	读源码不读类型定义	地图先于地形

#	误区	正确认知
18	读完不留可验证结论	写不下「我验证了 X 是 Y」= 没读

本书的三条带走结论

如果只记三句话，记这三句：

1. 默认即权力 —— 强制 = 默认 + 高摩擦回退。
2. 兼容还是另起，取决于用户有没有选择权。
3. 知识折旧率按层排序 —— 原理层最慢、打包层最快。因此第 2 章折旧最慢、回报最高。

回到开头：[README](#) · [目录](#)

配置示例

本目录的示例配置用于演示第 2、3 章讲解的机制，**不是可直接复制的生产模板**。

文件	演示的机制	对应章节
<code>webpack.config.js</code>	缓存三件套、Loader 顺序、splitChunks、dev/prod 分叉、Source Map 策略	§2.2、§2.3、§2.5、§2.7、§2.8
<code>vite.config.ts</code>	默认值覆盖、别名、环境变量、manualChunks、library mode 硬坑	§3.3、§3.4、§3.6、§7.5

使用建议

读法：每个配置块都写了注释说明「为什么这么配」和「不这么配会怎样」——**注释比配置本身更重要**。

动手验证：把某个配置项删掉，观察产物变化。这比读配置有效得多（对应 §7.3 台阶 2 的判据）。

不要照抄：这些示例里包含了一些**为演示而刻意保留的非最优写法**（例如把 `lodash` 整包导入然后再用按路径导入做对比），并已用注释标出。

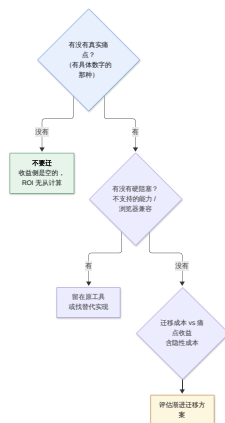
工具选型决策记录

性质：决策工具 + 依据留档。对应第 7 章 §7.5、§7.6。

说明：本文件记录的是**决策方法与已核验的事实依据**，不是“某公司选了什么”。课程中未在真实项目上做过实测，因此本文件不包含实测数据 —— 所有倍数均为厂商公开口径，见 `appendix/references.md`。

一、决策顺序（最重要的一条）

先看有无迁移成本 → 再看有无硬阻塞 → 最后才比成本收益。不是先比技术优劣。



为什么第一步是"有没有痛点"

没有痛点，**ROI** 的收益侧是空的 —— 你没法计算"获得了多少"，因为没有要解决的问题。而成本是实打实的。

ROI 作为框架的警告

ROI 在实践中**很难用准**：

难点	说明
收益是感受性的	"开发体验变好"很难折算成数字
隐性成本易低估	功能冻结期、回归风险、CI 改造、下游依赖产物结构的人

用法：当思考框架，不当计算公式。

一条追问（用于识别"假 ROI 问题"）

这是"划不来"的问题，还是"根本做不到"的问题？

示例：问"Rspack 为什么不另造生态，兼容 Webpack 多费力？"

- 用 ROI 回答 → "再造生态成本高、ROI 低"（方向上对，量级错）

- 正确回答 → 这不是 **ROI** 选择，是生存条件 —— 另造生态的困境是**双边网络效应**导致的冷启动起不来 (§4.3)

二、场景决策表

情境	建议	依据
全新项目	Vite	约定完备（默认值覆盖语言层）、冷启动快、生态成熟
中小存量项目、无硬阻塞	保持 Webpack	迁移成本 > 收益
存量项目、 build 成瓶颈	评估 Rspack	配置接近 Webpack (<code>rspack.config.js</code>)，可渐进迁移
必须深度定制产物结构	Rspack / Webpack	产物编排与干预点更强
多页 / 微前端	看生态，常落 Webpack / Rspack	对插件生态与产物控制要求高
需兼容旧浏览器	Webpack ，或 Vite + <code>@vitejs/plugin-legacy</code>	Vite dev 依赖原生 ESM
库 / 组件包	Vite library mode ，或 tsup / unbuild	需求单一，够用
Next.js 项目	用默认 (Turbopack)	框架内置，非独立选项

三条原则

1. 新项目默认 **Vite**
2. 存量项目不轻易迁移

3. 只有在明确证据「Vite 不够用」时才换

三、按需求维度拆解

当上面那张表不适用时（例如你的项目横跨多个场景），按维度拆：

维度	倾向 Webpack / Rspack	倾向 Vite
产物结构控制	干预点更多	够用但不极致
浏览器兼容	产物已降级，无要求	dev 需原生 ESM
插件生态	存量最多	借 Rollup 生态，够用
冷启动速度	$\propto$ 项目规模	$\approx$ 常数
生产构建速度	有持久化缓存	Rolldown 已大幅改善
配置心智负担	高（两套扩展点）	低（一套 + 大量默认值）
微前端 / Module Federation	Rspack 支持 2.0	生态较弱

注意"生产构建速度"这一行：Vite 走 Rollup 单线程时超大项目可能不如 **Webpack**；Vite 8 换 Rolldown 后大幅改善，但具体到你的项目仍需自测（\$5.6）。

四、已核验的事实依据

以下事实在课程中经官方来源核验（核验日期 **2026-10-10**）。工具链演进快，使用前请复核。

事实	值
Rspack 兼容性	loader runner 与 Webpack 完全兼容；下载量 top 50 webpack 插件 >80% 可用或有替代
Rspack 成熟度	1.0 已发布，声明 production-ready；字节内部 >1000 Web 应用
Rspack 不支持项	<code>resolve.plugins</code> 等少数项
Rspack 推荐 loader	<code>builtin:swc-loader</code> (减少 JS ↔ Rust 往返)
Farm 兼容目标	Vite (独立项目，非字节)
Farm 插件机制	<code>vitePlugins</code> ；Farm JS 插件 必须配 <code>filters</code>
Farm 独有策略	Partial bundling
Farm 成熟度	2023 开源，2024-2025 生产可用；生态明显小于 Rspack
Turbopack 默认状态	Next.js 16 (2025-10) dev 与 build 双双默认
Turbopack 回退	显式 <code>--webpack</code> ；有 <code>webpack()</code> 配置则 <code>next build</code> 直接失败
Turbopack 兼容性	完全不支持 webpack 插件
Turbopack 硬要求	Next.js —— 离开框架无意义
Rolldown 兼容性	Rollup 插件 API 兼容
Rolldown 默认状态	Vite 8 (2026-03-12) 默认；Rolldown 1.0 (2026-05-07)
Rolldown 硬约束	Node 20.19+ / 22.12+ ； ESM-only ； Yarn PnP 兼容问题 (Windows 尤甚)
Rolldown 配置变化	<code>build.rollupOptions</code> → <code>build.rolldownOptions</code>

事实	值
esbuild / SWC	都不做类型检查；只剥离类型标注
SWC 插件体系	Rust / WASM，与 Babel 的 JS 插件不通用
Vite 8 dev HMR	相比 Vite 7 基本没变（收益在生产构建与依赖预构建）
Vite 依赖预构建执行者	当前为 Rolldown （早期为 esbuild）

性能数字（均为厂商公开口径，非本书实测）：

工具	厂商口径
Rspack	"10x faster than webpack"
Turbopack	生产构建快 2-5x、Fast Refresh 快至 10x
Rolldown	比 Rollup 快 10-30x，与 esbuild 持平
Rolldown 案例	Linear 46s→6s (-87%)、GitLab 2.5min→22s、Ramp -57%、Beehiiv -64%、Mercedes-Benz.io -38%

使用提醒：厂商数字是起点，不是判据。要在自己项目上用，先按 §5.6 六步法自测。

五、决策记录模板

给每个实际做过的选型决策留一份记录。

决策：<项目名> 的构建工具选型

- 日期：
- 决策人：
- 状态：提议 / 已采纳 / 已废弃

背景

(现状用什么、痛点是什么，必须有数字)

候选方案

方案	优点	缺点	硬阻塞
---	---	---	---

决策依据

(对照第 7 章 §7.5 的决策顺序：痛点 → 硬阻塞 → 成本收益)

未采纳方案的原因

(这一节很重要 —— 它防止半年后有人重新提同一个方案)

生效条件与复查点

(什么情况下应该重新评估这个决策)

"未采纳方案的原因"这一节最容易被省略，也最值钱 —— 它避免团队反复讨论同一个问题。

六、需要重新评估的信号

出现以下情况时，之前的选型决策应该被重新审视：

信号	说明
生产构建时间 显著增长 并开始阻塞发布	痛点的定义变了
出现当前工具 无法解决 的构建问题	硬阻塞出现了
团队 频繁 在构建配置上踩坑	心智负担成为真实成本
目标工具 进入默认路径 （被框架默认采用）	学习成本大幅下降
当前工具 进入维护模式	长期风险
生态出现 事实标准级 的替代（如 Rolldown 进入 Vite 默认）	迁移成本大幅下降

注意最后两条：它们说明"该不该迁"这件事**会随时间改变** —— 因为**迁移成本在变**（生态成熟度、默认路径）。所以选型决策应该带**复查点**，而不是"一次定终身"。

术语表

全书关键术语的统一定义。凡正文与该表冲突，以本表为准。

构建基础

模块 (Module) 构建器处理的最小单位，通常对应一个文件。在依赖图中是一个节点，节点存「绝对路径 + 转换后代码 + 依赖列表」。

依赖图 (Dependency Graph) 以模块为节点、`import` / `require` / 动态 `import()` 为边的有向图，根是入口。构建器从入口出发，循环执行 `resolve` → `load+transform` → `parse` 直到队列为空。**性质**：共享模块在图里只有 **1 个节点**（去重），因此模块代码只执行一次。**图 ≠ 树**。

resolve 把 `./utils` 这类相对路径补成**绝对路径**的过程。受 `resolve.extensions`、`resolve.alias`、`package.json` 的 `exports` 影响。**为什么需要**：路径归一化是去重的前提。

转换 (Transform) 把浏览器不认的源码变成浏览器认的 JS。执行者是 Loader 链 / 转译器 (Babel、SWC、esbuild、Oxc)。

转译 (Transpile) 狭义的转换：保留语义的前提下改变语法 (TS → JS、JSX → JS、新语法降级到旧语法)。与「编译」的区别：转译不改语言层级（都在 JS 家族内）；编译通常指更彻底的翻译。

产物

chunk module 的打包容器。1 个 chunk ≈ 产物里 1 个 `.js` 文件。**三个来源**：① `entry`（入口 chunk）② 动态 `import()`（懒加载 chunk）③ `splitChunks`（公共 chunk）。**读产物的**

第一步：判断一个 chunk 属于哪一类来源。

入口 (Entry) 依赖图的起点。对象写法时，key 即产物名模板里的 `[name]`。

产物 (Bundle / Asset) 构建输出的、将被部署的文件集合（通常在 `dist/`）。

hash (内容指纹) 产物文件名里的一段十六进制字符串，用于缓存控制。三种：

- `[hash]` —— 整个构建 → 禁用
- `[chunkhash]` —— 单个 chunk → 易被运行时胶水污染
- `[contenthash]` —— 该文件的最终内容 → 生产首选

运行时胶水 (Runtime Glue) 打包器注入到产物里的模块注册表与 `require` 实现 (Webpack 的 `__webpack_require__` 等)。**为什么要单独抽出来：**否则它存在于每个 chunk 里，任何一处变化会污染其他 chunk 的 `chunkhash` → 缓存白白失效。

缓存三件套 `contenthash` + `runtimeChunk: 'single'` + `moduleIds: 'deterministic'`。缺一个 hash 就不稳。

hash 稳定性 内容不变时文件名不变的属性。它是**长效 HTTP 缓存成立的前提**。**判据：**改一行代码能预测哪些产物 hash 会变。

扩展点

Loader 单模块级的转换管道。配置在 `module.rules`，数组化。**执行顺序：**从右到左、从下到上（等价于函数组合 `f(g(h(x)))`）。**本质：**类型转换管道 —— 每一步的输出必须匹配下一步的输入。

Plugin 全流程级的干预装置。通过 `apply(compiler)` + `compiler.hooks.xxx.tap()` 注册。**能介入四阶段：**初始化 / 编译 / 生成 / 输出。**能生成、修改、删除最终产物。**

判别口诀 处理单个源文件内容 → Loader；涉及构建阶段、跨文件、产物编排 → Plugin。**判据是**「它处理的是谁」，不是「它看起来在干什么」。

Tapable Webpack 的钩子体系。与 **Rollup** 插件接口不通用 —— 这是 Webpack 插件无法迁移到 Vite 的根本原因。

产物优化

Tree Shaking 移除未被使用的导出。两个执行者两阶段：① 标记（打包器 `usedExports` / Rollup）—— 打标记，代码仍保留 ② 删除（压缩器 Terser / SWC minify）—— 读标记，物理删除 两个前提：必须是 ESM；生产模式才启用。

`usedExports` 打包器自动生成的标记，粒度是导出的成员。

`sideEffects` `package.json` 字段，粒度是整个模块能否跳过。它由库作者提供，消费方改不了。语义：`false` 表示未被引用的模块可整体跳过；有引用则照常执行。CSS / polyfill 需写数组例外。

代码分割 (**Code Splitting**) 把产物拆成多个 chunk。三种触发方式：多入口 / 动态 `import()` / `splitChunks`。核心分界线：只有动态 `import()` 能改变「运行时什么时候下载」。

动态 **import** `import()` 表达式。构建期分包，运行时按需下载。效果：减小首屏体积；不减小总产物体积。

`splitChunks` Webpack 的抽公共包配置。注意：Webpack 5 默认 `chunks: 'async'` —— 只对动态 import 产生的 chunk 生效，同步入口之间的共享模块不会自动抽。`minChunks` 数的是「引用它的 **chunk 数量**」，不是模块数量。

`manualChunks` Rollup / Vite 侧的自定义分组配置
(`build.rollupOptions.output.manualChunks`)。

Scope Hoisting / `concatenateModules` 把多个模块的作用域合并，减少运行时胶水，便于压缩器做跨模块死代码消除。`mode: 'production'` 下默认开启。

开发体验

HMR (Hot Module Replacement) 运行时替换变更的模块，保留应用状态。链路：改文件 → Watcher → Dev Server 重新编译 → WebSocket 推送 → 浏览器 HMR Runtime 替换。

accept 边界：从改动模块沿引用链向上冒泡，找到最近声明 `accept` 的模块作为替换落点；找不到就上溯到 **entry** → 退化为整页刷新。边界由框架插件提供 (`vue-loader` / `@vitejs/plugin-vue` / `react-refresh`)。

Watcher 监听文件系统变化的组件。

Source Map 产物位置 ↔ 源码位置的反向映射 (`.map` 文件 + 产物末行的 `//# sourceMappingURL=` 注释)。性质：不改变运行结果，只是给调试器看的旁路信息。

devtool (Webpack) / build.sourcemap (Vite) Source Map 的生成策略。核心取舍：速度 ↔ 精度 ↔ 是否泄露源码。生产禁用 `source-map` (注释留在产物里 → 可下载 `.map` 还原源码)；用 `hidden-source-map` (生成但不写注释)。

模式与环境

mode Webpack 的预设默认值开关 (`development` / `production`)。不是全能开关 —— 只管“默认值有没有被设定”，不管“你的项目需不需要”。

NODE_ENV 构建期字面量替换的产物，不是运行时环境变量。两个执行者：① 打包器 (`DefinePlugin` / Vite `define`) 整体替换成字符串字面量；② 压缩器发现条件恒假 → 判死代码 → 物理删除。

构建期字面量替换 (Compile-time Text Substitution) 在编译期把一段源代码文本替换成另一个字面量。重要推论：只能替换完整字面量表达式。`import.meta.env[key]` 这类动态取值拿不到。

VITE_ 前缀 Vite 中默认暴露给前端代码的环境变量前缀。安全设计 —— 防止服务端密钥进前端产物。

Webpack 与 Vite

依赖预构建 (Dependency Pre-Bundling) Vite dev 启动时对 `node_modules` 依赖做的预处理。两个目的（顺序不能反）：① **CJS/UMD → ESM 兼容** —— 不做则浏览器不认 `require`，直接跑不起来 ② **性能** —— 把内部模块极多的 ESM 依赖合并（`lodash-es` 600+ 模块 → 1）**缓存位置**：`node_modules/.vite`。**执行者**：当前为 Rolldown（早期 esbuild）。

原生 ESM 开发服务器 不打包，直接以原生 ESM 提供源码，按 import 请求逐个转换的开发服务器模型。Vite dev 采用此模型。

请求瀑布 (Request Waterfall) 串行的依赖链 —— A 加载完才知道要加载 B。与"并发太多导致的网络拥塞"是两个不同问题。

网络拥塞 大量请求并发导致的问题（如 600+ 个模块请求同时发出）。**不要把它叫请求瀑布**。

新一代工具

打包层 / 转译层 / 压缩层 / 一体化层 构建管线的分层。**分层替换**：转译层 Babel·tsc → SWC·esbuild；压缩层 Terser → SWC minify；打包层 Webpack·Rollup → Rspack·Rolldown·Farm；一体化层 → Turbopack。

并行度决定难度 打包层最难，因为依赖图是串行约束 —— 必须先知道谁依赖谁，才能决定怎么打包。

双边网络效应 插件作者没有动力为"没有用户的新工具"写插件，而他们的用户都在既有生态上 → 另造生态不是"成本高"，是"冷启动根本起不来"。

默认即权力 强制 ≠ 禁止。强制 = 默认 + 高摩擦回退。敢不兼容的三个条件：① 有分发渠道（打包器被包在框架里）② 回退高摩擦（逆默认值）③ 有资格（用户基数 + 不可替代性）。

dev / prod parity (一致性) 开发与生产使用同一个引擎，消除"本地能跑、生产挂"这一整类 bug。**Rolldown** 最值钱的收益不是速度，是 **parity** —— 速度有替代方案，一致性没有。

Partial bundling Farm 的独有策略。介于"全打包"与"不打包 ESM"之间，同时避开请求瀑布与巨型产物。

Oxc / Lightning CSS Rolldown 的技术底座：Oxc 负责转换，Lightning CSS 负责 CSS 压缩。

折旧率 知识的过时速度。判断该学什么的依据之一。**判据**：① 层越低折旧越慢 ② 是否已在默认路径上 ③ 是否兼容存量生态。**结论**：原理层最慢、打包层最快 —— 因此第 2 章折旧最慢、回报最高。

AI 与构建

agents as users AI agent 成为构建工具（dev server / bundler / linter / CLI）的**直接用**户，而不只是人类的助手。**推论**：对人类是体验问题，对 agent 是**可靠性要求**（迭代频率高几百倍）。

结构化诊断 (Structured Diagnostics) 把构建日志预处理成固定字段（`file` / `line` / `column` / `severity` / `message` / `code`），**原始日志不进上下文窗口**。

两步式 (Two-step Retrieval) 先返回摘要，需要时才取详情 —— 用于在信息量与 token 成本之间取平衡。

运行时感知工具 以 dev server 中间件 / 浏览器代理的形式，把 DOM、组件树、computed style、模块图、路由、服务日志通过 MCP 暴露给 agent。

学习方法

判据 (Criterion) "怎么算过关"的可判定标准。**先写判据，再动手**。**反例**："能跑起来"不是判据；"改一行代码能预测哪些 hash 会变"才是。

可自证 项目做完后，**不靠感觉**就能判断自己是否掌握。配动手项目的两条标准之一。

五台阶 按能力而非工具组织的学习顺序：① 建立图 ② 会用默认值 ③ **会读产物**（分水岭）④ 会算账 ⑤ 会迁移选型。

六步测量法 ①拆指标 ②固定变量 ③清三层缓存 ④各跑 5 次取中位数 ⑤记录元数据 ⑥出带前提的结论。

三层缓存 ① 构建工具缓存（`node_modules/.cache`、`node_modules/.vite`）② 浏览器强缓存（无痕验证）③ 系统层。

带前提 结论中同时给出机器、代码规模、依赖数、缓存状态、测量次数等条件。**缺前提的倍数不可复现。**

以例代概念 用实例或短语指代原则，导致答案无法复用、无法证伪。 **修正动作**：回答抽象问题**先命名概念再举例**；回答分类问题**先给类别名再列实例**。

配置参考

关键配置项的作用、默认行为、适用条件与常见坑。

版本提示：本文件基于 2026-10-10 的官方文档整理。配置项的默认值随版本变化，使用前请复核官方文档。核验线索见 `references.md`。

可运行的完整示例见 `practice/examples/`。

一、Webpack 关键配置

入口与输出

配置项	作用	默认 / 注意	常见坑
<code>entry</code>	依赖图起点	默认 <code>./src/index.js</code>	多入口必须配 <code>[name]</code> ，否则产物互相覆盖
<code>output.path</code>	输出目录	默认 <code>./dist</code>	必须绝对路径，用 <code>path.resolve(__dirname, ...)</code>
<code>output.filename</code>	入口产物名	默认 <code>main.js</code>	生产用 <code>[name].[contenthash:8].js</code>
<code>output.chunkFilename</code>	非入口 chunk 名	—	非入口 chunk 也要配，否则也是固定名
<code>output.clean</code>	构建前清空输出目录	默认 <code>false</code>	不开会导致旧产物堆积，污染产物分析
<code>output.publicPath</code>	产物引用前缀	默认 <code>'auto'</code>	CDN 部署时必配

optimization

配置项	作用	production 下的默认	常见坑
<code>minimize</code>	是否压缩	<code>true</code>	关掉后 Tree Shaking 的删除阶段不执行
<code>nodeEnv</code>	设置 <code>NODE_ENV</code>	<code>'production'</code>	经 DefinePlugin 实现构建期替换
<code>usedExports</code>	Tree Shaking 标记	<code>true</code>	只做标记，删除靠压缩器
<code>sideEffects</code>	启用 sideEffects 分析	<code>true</code>	需要库本身在 <code>package.json</code> 里写了这个字段
<code>concatenateModules</code>	Scope Hoisting	<code>true</code>	—
<code>moduleIds</code>	模块 ID 生成方式	<code>'deterministic'</code>	非 <code>deterministic</code> 时 hash 极不稳
<code>chunkIds</code>	chunk ID 生成方式	<code>'deterministic'</code>	同上
<code>splitChunks.chunks</code>	抽包范围	<code>'async'</code> (注意！)	想抽同步共享模块必须写 <code>'all'</code>
<code>splitChunks.minChunks</code>	抽包阈值	1	数的是「引用它的 chunk 数量」，不是模块数量
<code>runtimeChunk</code>	抽运行时	<code>false</code>	缓存三件套之一，建议 <code>'single'</code>
<code>cache.type</code>	缓存类型	—	设 <code>'filesystem'</code> 让二次构建接近增量

module.rules (Loader)

配置项	作用	常见坑
test	匹配文件正则	写过窄会漏转： <code>/\.(ts\$)/</code> 漏掉 <code>.tsx</code>
use	Loader 数组	执行顺序从右到左
exclude	排除目录	一般要 <code>exclude: /node_modules/</code> ，否则构建极慢
type	asset modules 类型	静态资源需一行 <code>type: 'asset/resource'</code>

devtool

值	速度	质量	适用
eval	极快	差	dev 迭代
eval-cheap-module-source-map	快	中	dev 推荐
source-map	慢	最高	生产禁用 （源码泄露）
hidden-source-map	慢	最高	生产推荐 ：生成但不写注释
nosources-source-map	慢	高	生产需对外暴露时

默认行为：`mode: 'production'` 下 `devtool: false` —— 默认不生成 **source map**。

resolve

配置项	作用	常见坑
extensions	省略扩展名时的查找顺序	顺序影响解析结果
alias	路径别名	与 Vite 同名，可迁移
plugins	resolve 插件	Rspack 不支持，迁移时需处理
exportsFields	是否读 package.json 的 exports	影响子路径能否被解析

plugins

配置项	作用	常见坑
plugins: [new XxxPlugin()]	注册插件	忘了 new 会报错

二、Vite 关键配置

一句话总览

Vite 的默认值覆盖了语言层（TS 转译、JSX、CSS、静态资源、后缀、dev server、HMR），所以配置文件通常很短。

顶层

配置项	作用	默认	常见坑
<code>plugins</code>	插件数组 (工厂函数)	<code>[]</code>	Webpack 是 <code>new XxxPlugin()</code> ，这里是 <code>xxx()</code>
<code>resolve.alias</code>	路径别名	—	与 Webpack 同名，可迁移
<code>define</code>	构建期字面量替换	—	值要 <code>JSON.stringify()</code>
<code>server</code>	开发服务器	内置	不需要装 <code>webpack-dev-server</code>
<code>build</code>	生产构建	走 Rollup / Rolldown	—

server

配置项	作用	常见坑
<code>port</code>	端口	默认 <code>5173</code>
<code>proxy</code>	接口代理	常用 <code>changeOrigin: true</code>
<code>hot</code>	HMR	默认开启
<code>forwardConsole</code>	把客户端 console 转发到终端 (Vite 8)	官方说明「在使用 AI 编码代理时尤为实用」

build

配置项	作用	默认	常见坑
<code>sourcemap</code>	Source Map	<code>false</code>	<code>true</code> 会泄露源码；用 <code>'hidden'</code>
<code>lib</code>	库模式	—	见下方组件库硬坑
<code>lib.formats</code>	产物格式	Vite 5 后只出 ESM	需显式 <code>['es', 'cjs']</code>
<code>rollupOptions</code>	透传给 Rollup	—	—
<code>rollupOptions.output.manualChunks</code>	自定义分组	—	分组太细 → 请求数爆炸
<code>rolldownOptions</code>	Rolldown 的对应配置	—	Vite 8 起由 <code>rollupOptions</code> 改为 <code>rolldownOptions</code>
<code>target</code>	目标浏览器	—	决定语法降级程度

环境变量

文件 / 机制	作用
<code>.env</code>	通用环境变量
<code>.env.local</code>	本地覆盖，不应提交
<code>.env.[mode]</code>	模式专属
<code>VITE_</code> 前缀	只有带此前缀的变量默认暴露给前端代码（安全设计）
<code>import.meta.env</code>	代码中读取的入口
<code>loadEnv(mode, cwd, '')</code>	config 中读非 <code>VITE_</code> 前缀变量

核心坑：`import.meta.env.X` 是静态替换，必须写完整字面量表达式。
`import.meta.env[key]` 拿不到值。

三、产物体积与结构

事项	Webpack	Vite
抽公共包	<code>optimization.splitChunks</code> （默认 <code>chunks: 'async'</code> ）	Rollup 自动；自定义用 <code>manualChunks</code>
动态导入	代码里的 <code>import()</code>	同
排除外部依赖	<code>externals</code>	<code>rollupOptions.external</code>
产物分析	<code>webpack-bundle-analyzer</code>	<code>rollup-plugin-visualizer</code>

四、组件库场景的四个硬坑（Vite library mode）

#	坑	正确做法
1	Vite 默认不生成 <code>.d.ts</code>	<code>vite-plugin-dts</code> ，或单独 <code>tsc --emitDeclarationOnly</code>
2	<code>build.lib.formats</code> 需显式声明	<code>['es', 'cjs']</code> （Vite 5 后默认只出 ESM）
3	peer 依赖被打进产物	必须 <code>rollupOptions.external</code> （否则产物里带一份 React）
4	入口字段没对齐	<code>package.json</code> 的 <code>main</code> / <code>module</code> / <code>types</code> / <code>exports</code> 手动对齐产物

第 4 条的表现：「解析得到文件 A，运行时期待文件 B」—— 一类很难排查的运行时错误。

五、Rspack 迁移对照

原 Webpack 配置	Rspack 对应
<code>HtmlWebpackPlugin</code>	<code>HtmlRspackPlugin</code>
<code>MiniCssExtractPlugin</code>	<code>CssExtractRspackPlugin</code>
<code>css-loader</code>	内置 <code>type: 'css'</code>
<code>babel-loader</code> / <code>swc-loader</code>	<code>builtin:swc-loader</code> （减少 JS ↔ Rust 往返）
<code>resolve.plugins</code>	不支持
配置文件	<code>rspack.config.js</code> （≈ webpack 配置改名）

兼容性边界：loader runner 与 Webpack 完全兼容；下载量 top 50 的 webpack 插件 >80% 可用或有替代 —— 不是 100%。

六、Farm 关键配置

配置项	作用	常见坑
<code>vitePlugins</code>	直接挂 Vite / Rollup / unplugin 插件	—
Farm JS 插件	自定义 JS 逻辑	必须配 <code>filters</code> ，否则明显掉速
配置文件	<code>farm.config.ts</code>	与 Vite 配置不通用，需重写

七、常用别名与等价物速查

目的	Webpack	Vite
原始文本导入	<code>raw-loader</code>	<code>?raw</code>
资源 URL 导入	<code>file-loader</code>	<code>?url</code>
内联为 DataURL	<code>url-loader</code>	<code>?inline</code>
注入样式到 DOM	<code>style-loader</code> + <code>css-loader</code>	内置 (<code>import './a.css'</code>)
抽 CSS 文件	<code>MiniCssExtractPlugin</code>	内置 (<code>build.cssCodeSplit</code>)
生成 HTML	<code>HtmlWebpackPlugin</code>	内置 (根目录 <code>index.html</code>)
定义常量	<code>DefinePlugin</code>	<code>define</code>
开发服务器	<code>webpack-dev-server</code>	内置
HMR	需配置	内置

参考资料

诚实声明（请先读）

本文件只列实际核验过的来源，并标注核验方式与粒度。

课程进行中的核验方式是联网检索官方文档 / 官方站点 / 发布记录，核验日期集中在 **2026-10-10**。

粒度说明：过程记录中只逐字保留了一个完整 URL（Vite 依赖预构建文档）。其余来源是在官方站点 / 官方仓库 / 官方博客这一层级记录的，未逐字保留具体 **URL 与版本快照**。本文件如实标注这一差别 —— 需要精确引用时，请到对应官方站点核对当前内容。

凡本书中未经核验的结论，其来源均不在本文件内 —— 它们是没有来源的推断。

一、逐字记录过 URL 的来源

#	来源	支撑的结论	章节
1	<code>https://vite.dev/guide/dep-pre-bundling</code>	依赖预构建的两个目的（CJS/UMD → ESM 兼容 + 性能）； <code>lodash-es</code> 600+ 内部模块 → 600+ 请求；执行者为 Rolldown （早期 esbuild）	§3.2

二、按官方站点层级记录（未逐字保留

URL)

官方文档

来源	类型	支撑的结论	章节
Webpack 官方文档 — Concepts 系列	机制文档	依赖图三动作、Loader / Plugin 概念、 <code>module.rules</code> 与 <code>plugins</code> 的关系	§2.1–§2.3
Webpack 官方文档 — Asset Modules	机制文档	静态资源 "without configuring additional loaders" 的原文表述，以及仍需 <code>type</code> 声明的精度修正	§3.3
Webpack 官方文档 — Optimization / SplitChunksPlugin	API 文档	<code>chunks</code> 默认 <code>'async'</code> ; <code>minChunks</code> 的语义	§2.2、§2.5
Webpack 官方文档 — Devtool	API 文档	devtool 各值的速度 / 质量梯度	§2.7
Webpack 官方文档 — Cache	API 文档	<code>cache: { type: 'filesystem' }</code>	§5.3
Vite 官方文档 — Why Vite	机制文档	打包式 vs 原生 ESM 的架构差异	§3.1
Vite 官方文档 — Env Variables and Modes	机制文档	<code>VITE_</code> 前缀、 <code>loadEnv</code> 、 <code>import.meta.env</code> 为静态替换	§3.4
Vite 官方文档 — Build Options	API 文档	<code>build.sourcemap</code> 默认 <code>false</code> ; <code>build.lib.formats</code> ; <code>forwardConsole</code>	§2.7、§7.5、§6.3
Vite 官方文档 — Features (静态资源后缀)	API 文档	<code>?raw</code> / <code>?url</code> / <code>?inline</code>	§3.5

来源	类型	支撑的结论	章节
Rollup 官方文档 — Plugin Development	机制文档	Rollup 插件钩子 (含 <code>generateBundle</code> 等在 dev 下不触发的钩子)	§3.5

官方站点 / 官方仓库 (工具事实)

来源	支撑的结论	章节
Rspack 官方站点	1.0 发布与 production-ready 声明；"10x faster than webpack"；loader runner 完全兼容；top 50 插件 >80%； <code>builtin:swc-loader</code> 推荐；采用了 Module Federation 2.0；采用方名单（字节内部 >1000 应用、微软 / 亚马逊 / 阿里 / Discord）	\$4.3
Rspack 迁移文档 + <code>rspack-compat</code> 仓库	<code>HtmlWebpackPlugin</code> → <code>HtmlRspackPlugin</code> 、 <code>MiniCssExtractPlugin</code> → <code>CssExtractRspackPlugin</code> 、 <code>css-loader</code> → 内置、 <code>resolve.plugins</code> 不支持；逐插件兼容起始版本	\$4.3、\$7.6
Farm 官方站点	<code>vitePlugins</code> ；Farm JS 插件必须配 <code>filters</code> ；Partial bundling；三类插件的划分	\$4.3
Next.js / Turbopack 官方发布记录	Next.js 15.0 中 <code>next dev</code> 稳定； Next.js 16.0 (2025-10-21) dev 与 build 双双稳定并成为默认 ； <code>--webpack</code> 退回； <code>webpack()</code> 配置导致 <code>next build</code> 失败；三条设计（单图 / 函数级增量落盘 / dev 懒编译）；转换用 SWC、CSS 用 Lightning CSS、不做类型检查； <code>experimental.turbo.rules</code> ；Sass <code>~</code> 前缀不支持	\$4.4
Vercel 官方口径	Turbopack 生产构建快 2-5x、Fast Refresh 快至 10x（厂商自测）	\$4.4
Rolldown / VoidZero 官方博客与发布记录	时间线（2024-04 首发 → 2026-05-07 1.0 稳定）；基于 Oxc；Rollup 插件 API 兼容；Vite 8（2026-03-12）成为默认；比 Rollup 快 10-30x 的官方口径；真实案例（Linear 46s→6s、GitLab 2.5min→22s、Ramp -57%、Beehiiv -64%、Mercedes-Benz.io -38%）；Node 20.19+ / 22.12+ 与 ESM-only；Yarn PnP 兼容问题； <code>build.rolldownOptions</code>	\$4.4、\$4.5
Cloudflare 官方公告 (VoidZero 加入 Cloudflare，2026-06-04)	agents as users 原话；agent 对工具的三项要求；"For agents, speed is a reliability requirement"； <code>@cloudflare/vite-plugin</code> 周下载量与解释；100 万美元 Vite 生态基金；Vite 保持 MIT / 厂商中立	\$6.1、\$6.3

社区实践（AI 辅助构建部分）

来源	支撑的结论	章节
@paretools/build、cpp-build-mcp 一类 MCP build server	结构化诊断（file / line / column / severity / message / code）；两步式（摘要 → 详情）；原始日志不进上下文窗口	§6.3
Frontman、Stagewise 一类运行时感知工具	以 dev server 中间件 / 浏览器代理形式，通过 MCP 暴露 DOM / 组件树 / computed style / 模块图 / 路由 / 服务日志	§6.3

三、官方文档的推荐阅读入口

课程给出的"读哪些页"的清单（对应 §7.4）：

工具	值得逐字读的机制页
Webpack	Concepts 系列
Vite	Why Vite、Dependency Pre-Bundling
Rollup	Plugin Development
Rspack	migration 指南 + rspack-compat 仓库

读文档的判据：

读到「已知限制 / deprecated」段落时，停下精读。那是坑的官方来源。

四、本书未使用的来源

以下类型的来源未被本书采用，列出来是为了说明取材边界：

类型	为什么不用
二手技术博客 / 公众号文章	结论可能滞后或转述失真，且无法确认一手来源
厂商营销页的性能对比图	无法确认测试条件，且是厂商自测
社区问答（Stack Overflow 等）	答案质量与时效差异大，不适合作为书籍依据
未标注版本的教程	配置项默认值随版本变化

本书中所有性能倍数均标注为「厂商公开口径」，并明确提示需自测 —— 见 `decisions/tool-selection.md` 第四节。

五、复核建议

工具链演进速度快。阅读本书时若发现某条版本相关结论与当前官方文档不符，以官方文档为准。

建议的复核顺序：

- 1. 先查该工具的官方 **migration / breaking changes** 页 —— 官方承认的变更，性价比最高
- 2. 再查对应配置项的官方 **API 文档** —— 确认默认值有没有变
- 3. 最后查发布记录 / 官方微博 —— 确认版本时间线

复核后请更新：对应章节的结论。