插件流水线：从整体到局部，理解 Vite 的核心编译能力

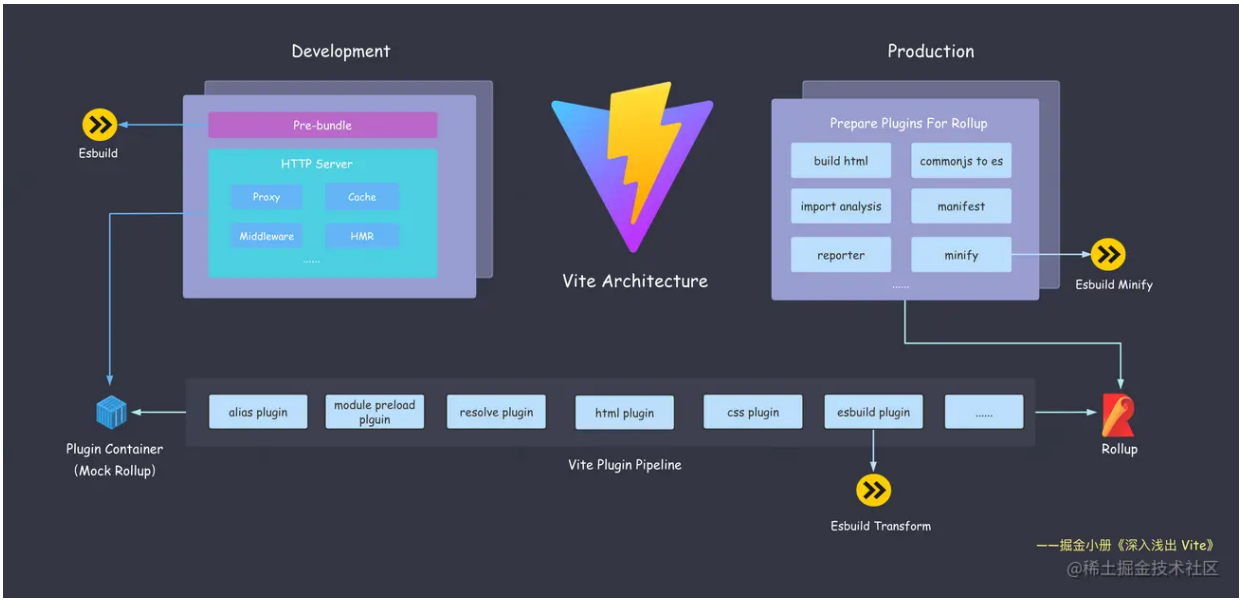
发布于 2022-05-09

我们知道，Vite 在开发阶段实现了一个按需加载的服务器，每一个文件请求进来都会经历一系列的编译流程，然后 Vite 会将编译结果响应给浏览器。在生产环境下，Vite 同样会执行一系列编译过程，将编译结果交给 Rollup 进行模块打包。这一系列的编译过程指的就是 Vite 的插件工作流水线(Pipeline)，而插件功能又是 Vite 构建能力的核心，因此谈到阅读 Vite 源码，我们永远绕不开插件的作用与实现原理。

接下来，我就和你一起分析 Vite 插件流水线的顶层架构，也就是各个插件如何被调度和组织起来的，详细说说 Vite 插件容器(`PluginContainer`)机制的实现，同时带你一起梳理开发阶段和生产环境各自会用到的插件，并分析各自的功能与实现原理，让你能够全面、准确地认识 Vite 的插件流水线！

插件容器

从《双引擎架构》小节中我们知道 Vite 的插件机制是与 Rollup 兼容的，但它在开发和生产环境下的实现稍有差别，你可以回顾一下这张架构图:



我们可以看到:

- 在生产环境中 Vite 直接调用 Rollup 进行打包，所以 Rollup 可以调度各种插件；
- 在开发环境中，Vite 模拟了 Rollup 的插件机制，设计了一个 `PluginContainer` 对象来调度各个插件。

`PluginContainer` (插件容器)对象非常重要，前两节我们也多次提到了它，接下来我们就把目光集中到这个对象身上，看看 Vite 的插件容器机制究竟是如何实现的。

`PluginContainer` 的 实现 基于借鉴于 WMR 中的 `rollup-plugin-container.js` ，主要分为 2 个部分:

> 实现 Rollup 插件钩子的调度
>
> 实现插件钩子内部的 Context 上下文对象

首先，你可以通过 container 的定义 来看看各个 Rollup 钩子的实现方式，代码精简后如下:

```
const container = {
  // 异步串行钩子
  options: await (async () => {
    let options = rollupOptions
    for (const plugin of plugins) {
      if (!plugin.options) continue
      options =
        (await plugin.options.call(minimalContext, options)) || options
    }
    return options;
  })(),
  // 异步并行钩子
  async buildStart() {
    await Promise.all(
      plugins.map((plugin) => {
        if (plugin.buildStart) {
          return plugin.buildStart.call(
            new Context(plugin) as any,
            container.options as NormalizedInputOptions
          )
        }
      })
    )
  },
  // 异步优先钩子
  async resolveId(rawId, importer) {
    // 上下文对象，后文介绍
    const ctx = new Context()
```

```
    let id: string | null = null
    const partial: Partial<PartialResolvedId> = {}
    for (const plugin of plugins) {
      const result = await plugin.resolveId.call(
        ctx as any,
        rawId,
        importer,
        { ssr }
      )
      if (!result) continue;
      return result;
    }
  }
  // 异步优先钩子
  async load(id, options) {
    const ctx = new Context()
    for (const plugin of plugins) {
      const result = await plugin.load.call(ctx as any, id, { ssr })
      if (result != null) {
        return result
      }
    }
    return null
  },
  // 异步串行钩子
  async transform(code, id, options) {
    const ssr = options?.ssr
    // 每次 transform 调度过程会有专门的上下文对象，用于合并 SourceMap，后文会介绍
    const ctx = new TransformContext(id, code, inMap as SourceMap)
    ctx.ssr = !!ssr
    for (const plugin of plugins) {
      let result: TransformResult | string | undefined
      try {
        result = await plugin.transform.call(ctx as any, code, id, { ssr })
      } catch (e) {
        ctx.error(e)
      }
      if (!result) continue;
      // 省略 SourceMap 合并的逻辑
      code = result;
    }
    return {
      code,
      map: ctx._getCombinedSourcemap()
    }
  },
  // close 钩子实现省略
}
```

在《Vite 构建基石(下)——深入理解 Rollup 的插件机制》中，我们已经系统学习过 Rollup 中异步、串行、并行等钩子类型的执行原理了，现在再来阅读这部分 `PluginContainer` 的实现代码应该并不困难。

不过值得注意的是，在各种钩子被调用的时候，Vite 会强制将钩子函数的 `this` 绑定为一个上下文对象，如:

```
const ctx = new Context()
const result = await plugin.load.call(ctx as any, id, { ssr })
```

这个对象究竟是用来干什么的呢?

我们知道，在 Rollup 钩子函数中，我们可以调用 `this.emitFile` 、 `this.resolve` 等诸多的上下文方法(详情地址)，因此，Vite 除了要模拟各个插件的执行流程，还需要模拟插件执行的上下文对象，代码中的 `Context` 对象就是用来完成这件事情的。我们来看看 Context 对象的具体实现:

```
import { RollupPluginContext } from 'rollup';
type PluginContext = Omit<
  RollupPluginContext,
  // not documented
  | 'cache'
  // deprecated
  | 'emitAsset'
  | 'emitChunk'
  | 'getAssetFileName'
  | 'getChunkFileName'
  | 'isExternal'
  | 'moduleIds'
  | 'resolveId'
  | 'load'
>

const watchFiles = new Set<string>()

class Context implements PluginContext {
  // 实现各种上下文方法
  // 解析模块 AST(调用 acorn)
  parse(code: string, opts: any = {}) {
    return parser.parse(code, {
      sourceType: 'module',
      ecmaVersion: 'latest',
      locations: true,
      ...opts
    })
  }
  // 解析模块路径
  async resolve(
    id: string,
    importer?: string,
    options?: { skipSelf?: boolean }
  ) {
    let skip: Set<Plugin> | undefined
    if (options?.skipSelf && this._activePlugin) {
```

```
      skip = new Set(this._resolveSkips)
      skip.add(this._activePlugin)
    }
    let out = await container.resolveId(id, importer, { skip, ssr: this.ssr })
    if (typeof out === 'string') out = { id: out }
    return out as ResolvedId | null
  }

  // 以下两个方法均从 Vite 的模块依赖图中获取相关的信息
  // 我们将在下一节详细介绍模块依赖图，本节不做展开
  getModuleInfo(id: string) {
    return getModuleInfo(id)
  }

  getModuleIds() {
    return moduleGraph
      ? moduleGraph.idToModuleMap.keys()
      : Array.prototype[Symbol.iterator]()
  }

  // 记录开发阶段 watch 的文件
  addWatchFile(id: string) {
    watchFiles.add(id)
    ;(this._addedImports || (this._addedImports = new Set())).add(id)
    if (watcher) ensureWatchedFile(watcher, id, root)
  }

  getWatchFiles() {
    return [...watchFiles]
  }

  warn() {
    // 打印 warning 信息
  }

  error() {
    // 打印 error 信息
  }

  // 其它方法只是声明，并没有具体实现，这里就省略了
}
```

很显然，Vite 将 Rollup 的 `PluginContext` 对象重新实现了一遍，因为只是开发阶段用到，所以去除了一些打包相关的方法实现。同时，上下文对象与 Vite 开发阶段的 ModuleGraph 即模块依赖图相结合，是为了实现开发时的 HMR。 HMR 实现的细节，我们将在下一节展开介绍。

另外，transform 钩子也会绑定一个插件上下文对象，不过这个对象和其它钩子不同，实现代码精简如下:

```
class TransformContext extends Context {
  constructor(filename: string, code: string, inMap?: SourceMap | string) {
```

```
    super()
    this.filename = filename
    this.originalCode = code
    if (inMap) {
      this.sourcemapChain.push(inMap)
    }
  }

  _getCombinedSourcemap(createIfNull = false) {
    return this.combinedMap
  }

  getCombinedSourcemap() {
    return this._getCombinedSourcemap(true) as SourceMap
  }
}
```

可以看到， `TransformContext` 继承自之前所说的 `Context` 对象，也就是说 transform 钩子的上下文对象相比其它钩子只是做了一些扩展，增加了 sourcemap 合并的功能，将不同插件的 transform 钩子执行后返回的 sourcemap 进行合并，以保证 sourcemap 的准确性和完整性。

插件工作流概览

在分析配置解析服务的小节中，我们提到过生成插件流水线即 `resolvePlugins` 的逻辑，但没有具体展开，这里我们就来详细拆解一下 Vite 在这一步究竟做了啥。

让我们把目光集中在 `resolvePlugins` 的实现上，Vite 所有的插件就是在这里被收集起来的。具体实现如下:

```
export async function resolvePlugins(
  config: ResolvedConfig,
  prePlugins: Plugin[],
  normalPlugins: Plugin[],
  postPlugins: Plugin[]
): Promise<Plugin[]> {
  const isBuild = config.command === 'build'
  // 收集生产环境构建的插件，后文会介绍
  const buildPlugins = isBuild
    ? (await import('../build')).resolveBuildPlugins(config)
    : { pre: [], post: [] }

  return [
    // 1. 别名插件
    isBuild ? null : preAliasPlugin(),
    aliasPlugin({ entries: config.resolve.alias }),
    // 2. 用户自定义 pre 插件(带有`enforce: "pre"`属性)
    ...prePlugins,
```

```
    // 3. Vite 核心构建插件
    // 数量比较多，暂时省略代码
    // 4. 用户插件（不带有 `enforce` 属性）
    ...normalPlugins,
    // 5. Vite 生产环境插件 & 用户插件(带有 `enforce: "post"`属性)
    definePlugin(config),
    cssPostPlugin(config),
    ...buildPlugins.pre,
    ...postPlugins,
    ...buildPlugins.post,
    // 6. 一些开发阶段特有的插件
    ...(isBuild
      ? []
      : [clientInjectionsPlugin(config), importAnalysisPlugin(config)])
  ].filter(Boolean) as Plugin[]
}
```

从上述代码中我们可以总结出 Vite 插件的具体执行顺序。

> **别名插件**包括 `vite:pre-alias` 和 `@rollup/plugin-alias` ，用于路径别名替换。
>
> 用户自定义 pre 插件，也就是带有 `enforce: "pre"` 属性的自定义插件。
>
> Vite 核心构建插件，这部分插件为 Vite 的核心编译插件，数量比较多，我们在下部分一一拆解。
>
> 用户自定义的普通插件，即不带有 `enforce` 属性的自定义插件。
>
> `Vite 生产环境插件` 和用户插件中带有 `enforce: "post"` 属性的插件。
>
> 一些开发阶段特有的插件，包括环境变量注入插件 `clientInjectionsPlugin` 和 import 语句分析及重写插件 `importAnalysisPlugin` 。

那么，在执行过程中 Vite 到底应用了哪些插件，以及这些插件内部究竟做了什么？我们来一一梳理一下。

插件功能梳理

> 这一节，我们主要围绕实现原理展开，并不会详细介绍所有插件的代码实现细节，不过相应的源码链接我都会放到文章当中，感兴趣的同学可以在课后进一步阅读。

除用户自定义插件之外，我们需要梳理的 Vite 内置插件有下面这几类:

- 别名插件
- 核心构建插件
- 生产环境特有插件
- 开发环境特有插件

1. 别名插件

别名插件有两个，分别是 vite:pre-alias 和 @rollup/plugin-alias。 前者主要是为了将 bare import 路径重定向到预构建依赖的路径，如:

```
// 假设 React 已经过 Vite 预构建
import React from 'react';
// 会被重定向到预构建产物的路径
import React from '/node_modules/.vite/react.js'
```

后者则是实现了比较通用的路径别名(即 `resolve.alias` 配置)的功能，使用的是 Rollup 官方 Alias 插件。

2. 核心构建插件

2.1 module preload 特性的 Polyfill

当你在 Vite 配置文件中开启下面这个配置时:

```
{
  build: {
    polyfillModulePreload: true
  }
}
```

Vite 会自动应用 `modulePreloadPolyfillPlugin` 插件，在产物中注入 module preload 的 Polyfill 代码，具体实现 摘自之前我们提到过的 `es-module-shims` 这个库，实现原理如下:

> 扫描出当前所有的 modulepreload 标签，拿到 link 标签对应的地址，通过执行 fetch 实现预加载；
>
> 同时通过 MutationObserver 监听 DOM 的变化，一旦发现包含 modulepreload 属性的 link 标签，则同样通过 fetch 请求实现预加载。

> 由于部分支持原生 ESM 的浏览器并不支持 module preload，因此某些情况下需要注入相应的 polyfill 进行降级。

2.2 路径解析插件

路径解析插件(即 `vite:resolve`)是 Vite 中比较核心的插件，几乎所有重要的 Vite 特性都离不开这个插件的实现，诸如依赖预构建、HMR、SSR 等等。同时它也是实现相当复杂的插件，一方面实现了 Node.js 官方的 resolve 算法，另一方面需要支持前面所说的各项特性，可以说是专门给 Vite 实现了一套路径解析算法。

这个插件的实现细节足以再开一个小节专门分析了，所以本节我们就不展开了，你初步了解就可以了。

2.3 内联脚本加载插件

对于 HTML 中的内联脚本，Vite 会通过 `vite:html-inline-script-proxy` 插件来进行加载。比如下面这个 script 标签:

```
<script type="module">
import React from 'react';
console.log(React)
</script>
```

这些内容会在后续的 build-html 插件从 HTML 代码中剔除，并且变成下面的这一行代码插入到项目入口模块的代码中:

```
import '/User/xxx/vite-app/index.html?http-proxy&index=0.js'
```

而 vite:html-inline-script-proxy 就是用来加载这样的模块，实现如下:

```
const htmlProxyRE = /\?html-proxy&index=(\d+)\.js$/

export function htmlInlineScriptProxyPlugin(config: ResolvedConfig): Plugin {
  return {
    name: 'vite:html-inline-script-proxy',
    load(id) {
      const proxyMatch = id.match(htmlProxyRE)
      if (proxyMatch) {
        const index = Number(proxyMatch[1])
        const file = cleanUrl(id)
        const url = file.replace(normalizePath(config.root), '')
        // 内联脚本的内容会被记录在 htmlProxyMap 这个表中
        const result = htmlProxyMap.get(config)!.get(url)![index]
        if (typeof result === 'string') {
          // 加载脚本的具体内容
          return result
        } else {
          throw new Error(`No matching HTML proxy module found from ${id}`)
        }
      }
    }
  }
}
```

2.4 CSS 编译插件

即名为 vite:css 的插件，主要实现下面这些功能:

- CSS 预处理器的编译
- CSS Modules
- Postcss 编译
- 通过 @import 记录依赖，便于 HMR

这个插件的核心在于 compileCSS 函数的实现，感兴趣的同学可以阅读一下这部分的源码。

2.5 Esbuild 转译插件

即名为 `vite:esbuild` 的插件，用来进行 `.js` 、 `.ts` 、 `.jsx` 和 `tsx` ，代替了传统的 Babel 或者 TSC 的功能，这也是 Vite 开发阶段性能强悍的一个原因。插件中主要的逻辑是 `transformWithEsbuild` 函数，顾名思义，你可以通过这个函数进行代码转译。当然，Vite 本身也导出了这个函数，作为一种通用的 transform 能力，你可以这样来使用:

```
import { transformWithEsbuild } from 'vite';

// 传入两个参数: code, filename
transformWithEsbuild('<h1>hello</h1>', './index.tsx').then(res => {
  // {
  //   warnings: [],
  //   code: '/* @__PURE__ */ React.createElement("h1", null, "hello");\n',
  //   map: {/* sourcemap 信息 */}
  // }
  console.log(res);
})
```

2.6 静态资源加载插件

静态资源加载插件包括如下几个:

- **vite:json** 用来加载 JSON 文件，通过 `@rollup/pluginutils` 的 `dataToEsm` 方法可实现 JSON 的按名导入，具体实现见链接;

- **vite:wasm** 用来加载 `.wasm` 格式的文件，具体实现见链接;

- **vite:worker** 用来 Web Worker 脚本，插件内部会使用 Rollup 对 Worker 脚本进行打包，具体实现见链接;

- **vite:asset**，开发阶段实现了其他格式静态资源的加载，而生产环境会通过 `renderChunk` 钩子将静态资源地址重写为产物的文件地址，如 `./img.png` 重写为 `https://cdn.xxx.com/assets/img.91ee297e.png` 。

值得注意的是，Rollup 本身存在 asset cascade 问题，即静态资源哈希更新，引用它的 JS 的哈希并没有更新(issue 链接)。因此 Vite 在静态资源处理的时候，并没有交给 Rollup 生成资源哈希，而是自己根据资源内容生成哈希(源码实现)，并手动进行路径重写，以此避免 `asset-cascade` 问题。

3. 生产环境特有插件

3.1 全局变量替换插件

提供全局变量替换功能，如下面的这个配置:

```
// vite.config.ts
const version = '2.0.0';

export default {
  define: {
    __APP_VERSION__: `JSON.stringify(${version})`
  }
}
```

全局变量替换的功能和我们之前在 Rollup 插件小节中提到的@rollup/plugin-replace 差不多，当然在实现上 Vite 会有所区别:

- 开发环境下，Vite 会通过将所有的全局变量挂载到 `window` 对象，而不用经过 define 插件的处理，节省编译开销；
- 生产环境下，Vite 会使用 define 插件，进行字符串替换以及 sourcemap 生成。

> 特殊情况: SSR 构建会在开发环境经过这个插件，仅替换字符串。

3.2 CSS 后处理插件

CSS 后处理插件即 `name` 为 `vite:css-post` 的插件，它的功能包括 `开发阶段 CSS 响应结果处理` 和 `生产环境 CSS 文件生成` 。

首先，在开发阶段，这个插件会将之前的 CSS 编译插件处理后的结果，包装成一个 ESM 模块，返回给浏览器，点击查看实现代码。

其次，生产环境中，Vite 默认会通过这个插件进行 CSS 的 code splitting，即对于每个异步 chunk，Vite 会将其依赖的 CSS 代码单独打包成一个文件，关键代码如下(源码链接):

```
const fileHandle = this.emitFile({
  name: chunk.name + '.css',
  type: 'asset',
  source: chunkCSS
});
```

如果 CSS 的 code splitting 功能被关闭(通过 `build.cssCodeSplit` 配置)，那么 Vite 会将所有的 CSS 代码打包到同一个 CSS 文件中，点击查看实现。

最后，插件会调用 Esbuild 对 CSS 进行压缩，实现在 `minifyCSS` 函数中，点击查看实现。

3.3 HTML 构建插件

`HTML` 构建插件 即 `build-html` 插件。之前我们在 `内联脚本加载插件` 中提到过，项目根目录下的 `html` 会转换为一段 JavaScript 代码，如下面的这个例子:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  // 普通方式引入
  <script src="./index.ts"></script>
  // 内联脚本
  <script type="module">
    import React from 'react';
    console.log(React)
  </script>
</body>
</html>
```

首先，当 Vite 在生产环境 `transform` 这段入口 HTML 时，会做 3 件事情:

对 HTML 执行各个插件中带有 `enforce: "pre"` 属性的 transformIndexHtml 钩子；

> 我们知道插件本身可以带有 `enforce: "pre"|"post"` 属性，而 transformIndexHtml 本身也可以带有这个属性，用于在不同的阶段进行 HTML 转换。后文会介绍 transformIndexHtml 钩子带有 `enforce: "post"` 时的执行时机。

将其中的 script 标签内容删除，并将其转换为 `import 语句` 如 `import './index.ts'` ，并记录下来；

在 transform 钩子中返回记录下来的 import 内容，将 import 语句作为模块内容进行加载。也就是说，虽然 Vite 处理的是一个 HTML 文件，但最后进行打包的内容却是一段 JS 的内容，点击查看具体实现。代码简化后如下所示:

```
export function buildHtmlPlugin() {
  name: 'vite:build',
  transform(html, id) {
    if (id.endsWith('.html')) {
      let js = '';
      // 省略 HTML AST 遍历过程(通过 @vue/compiler-dom 实现)
      // 收集 script 标签，转换成 import 语句，拼接到 js 字符串中
      return js;
    }
  }
}
```

其次，在生成产物的最后一步即 `generateBundle` 钩子中，拿到入口 Chunk，分析入口 Chunk 的内容, 分情况进行处理。

如果只有 import 语句，先通过 Rollup 提供的 `chunk` 和 `bundle` 对象获取入口 chunk 所有的依赖 chunk，并将这些 chunk 进行后序排列，如 `a 依赖 b，b 依赖 c` ，最后的依赖数组就是 `[c, b, a]` 。然后依次将 c，b, a 生成三个 script 标签，插入 HTML 中。最后，Vite 会将入口 chunk 的内容从 bundle 产物中移除，因此它的内容只要 import 语句，而它 import 的 chunk 已经作为 script 标签插入到了 HTML 中，那入口 Chunk 的存在也就没有意义了。

如果除了 import 语句，还有其它内容， Vite 就会将入口 Chunk 单独生成一个 `script 标签` ，分析出依赖的后序排列(和上一种情况分析手段一样)，然后通过注入 `<link`

`rel="modulepreload">` 标签对入口文件的依赖 chunk 进行预加载。

最后，插件会调用用户插件中带有 `enforce: "post"` 属性的 transformIndexHtml 钩子，对 HTML 进行进一步的处理。点击查看具体实现。

3.3 Commonjs 转换插件

我们知道，在开发环境中，Vite 使用 Esbuild 将 Commonjs 转换为 ESM，而生产环境中，Vite 会直接使用 Rollup 的官方插件 @rollup/plugin-commonjs。

3.4 date-uri 插件

date-uri 插件用来支持 import 模块中含有 Base64 编码的情况，如:

```
import batman from 'data:application/json;base64, eyAiYmF0bWFuIjogInRydWUiIH0=';
```

点击查看实现。

3.5 dynamic-import-vars 插件

用于支持在动态 import 中使用变量的功能，如下示例代码:

```
function importLocale(locale) {
  return import(`./locales/${locale}.js`);
}
```

内部使用的是 Rollup 的官方插件 @rollup/plugin-dynamic-import-vars。

3.6 import-meta-url 支持插件

用来转换如下格式的资源 URL:

```
new URL('./foo.png', import.meta.url)
```

将其转换为生产环境的 URL 格式，如:

```
// 使用 self.location 来保证低版本浏览器和 Web Worker 环境的兼容性
new URL('./assets.a4b3d56d.png, self.location)
```

同时，对于动态 import 的情况也能进行支持，如下面的这种写法:

```
function getImageUrl(name) {
  return new URL(`./dir/${name}.png`, import.meta.url).href
}
```

Vite 识别到 `./dir/${name}.png` 这样的模板字符串，会将整行代码转换成下面这样:

```
function getImageUrl(name) {
    return import.meta.globEager('./dir/**.png')[`./dir/${name}.png`].default;
}
```

点击查看具体实现

3.7 生产环境 import 分析插件

`vite:build-import-analysis` 插件会在生产环境打包时用作 import 语句分析和重写，主要目的是对动态 import 的模块进行预加载处理。

对含有动态 import 的 chunk 而言，会在插件的 `tranform` 钩子中被添加这样一段工具代码用来进行模块预加载，逻辑并不复杂，你可以参考源码实现。关键代码简化后如下:

```
function preload(importModule, deps) {
  return Promise.all(
    deps.map(dep => {
      // 如果异步模块的依赖还没有加载
      if (!alreadyLoaded(dep)) {
        // 创建 link 标签加载，包括 JS 或者 CSS
        document.head.appendChild(createLink(dep))
        // 如果是 CSS，进行特殊处理，后文会介绍
        if (isCss(dep)) {
          return new Promise((resolve, reject) => {
            link.addEventListener('load', resolve)
            link.addEventListener('error', reject)
          })
        }
      }
    })
  ).then(() => importModule())
}
```

我们知道，Vite 内置了 CSS 代码分割的能力，当一个模块通过动态 import 引入的时候，这个模块会被单独打包成一个 chunk，与此同时这个模块中的样式代码也会打包成单独的 CSS 文件。如果异步模块的 CSS 和 JS 同时进行预加载，那么在某些浏览器下(如 IE)就会出现 FOUC 问题，页面样式会闪烁，影响用户体验。但 Vite 通过监听 link 标签 `load` 事件的方式来保证 CSS 在 JS 之前加载完成，从而解决了 FOUC 问题。你可以注意下面这段关键代码:

```
if (isCss) {
  return new Promise((res, rej) => {
    link.addEventListener('load', res)
    link.addEventListener('error', rej)
  })
}
```

现在，我们已经知道了预加载的实现方法，那么 Vite 是如何将动态 import 编译成预加载的代码的呢?

从源码的 `transform` 钩子实现中，不难发现 Vite 会将动态 import 的代码进行转换，如下代码所示:

```
// 转换前
import('a')
// 转换后
__vitePreload(() => 'a', __VITE_IS_MODERN__ ?"__VITE_PRELOAD__":void)
```

其中，`__vitePreload` 会被加载为前文中的 `preload` 工具函数，`__VITE_IS_MODERN__` 会在 renderChunk 中被替换成 true 或者 false，表示是否为 Modern 模式打包，而对于 `"__VITE_PRELOAD__"` ，Vite 会在 generateBundle 阶段，分析出 a 模块所有依赖文件(包括 CSS)，将依赖文件名的数组作为 preload 工具函数的第二个参数。

同时，对于 Vite 独有的 import.meta.glob 语法，也会在这个插件中进行编译，如:

```
const modules = import.meta.glob('./dir/*.js')
```

会通过插件转换成下面这段代码:

```
const modules = {
  './dir/foo.js': () => import('./dir/foo.js'),
```

```
  './dir/bar.js': () => import('./dir/bar.js')
}
```

具体的实现在 transformImportGlob 函数中，除了被该插件使用外，这个函数被还依赖预构建、开发环境 import 分析等核心流程使用，属于一类比较底层的逻辑，感兴趣的同学可以精读一下这部分的实现源码。

3.8 JS 压缩插件

Vite 中提供了两种 JS 代码压缩的工具，即 Esbuild 和 Terser，分别由两个插件插件实现:

- **vite:esbuild-transpile** (点击查看实现)。在 renderChunk 阶段，调用 Esbuild 的 transform API，并指定 minify 参数，从而实现 JS 的压缩。

- **vite:terser**(点击查看实现)。同样也在 renderChunk 阶段，Vite 会单独的 Worker 进程中调用 Terser 进行 JS 代码压缩。

3.9 构建报告插件

主要由三个插件输出构建报告:

- **vite:manifest**(点击查看实现)。提供打包后的各种资源文件及其关联信息，如下内容所示:

```
// manifest.json
{
  "index.html": {
    "file": "assets/index.8edffa56.js",
    "src": "index.html",
    "isEntry": true,
    "imports": [
      // JS 引用
      "_vendor.71e8fac3.js"
    ],
    "css": [
      // 样式文件应用
      "assets/index.458f9883.css"
    ],
    "assets": [
```

```
      // 静态资源引用
      "assets/img.9f0de7da.png"
    ]
  },
  "_vendor.71e8fac3.js": {
    "file": "assets/vendor.71e8fac3.js"
  }
}
```

- **vite:ssr-manifest**(点击查看实现)。提供每个模块与 chunk 之间的映射关系，方便 SSR 时期通过渲染的组件来确定哪些 chunk 会被使用，从而按需进行预加载。最后插件输出的内容如下:

```
// ssr-manifest.json
{
  "node_modules/object-assign/index.js": [
    "/assets/vendor.71e8fac3.js"
  ],
  "node_modules/object-assign/index.js?commonjs-proxy": [
    "/assets/vendor.71e8fac3.js"
  ],
  // 省略其它模块信息
}
```

- **vite:reporter**(点击查看实现)。主要提供打包时的命令行构建日志:

```
vite v2.9.1 building for production...
✓ 26 modules transformed.
dist/assets/favicon.17e50649.svg    1.49 KiB
dist/assets/img.9f0de7da.png        29.57 KiB
dist/index.html                     0.54 KiB
dist/manifest.json                  0.33 KiB
dist/ssr-manifest.json              4.09 KiB
dist/assets/index.458f9883.css      0.29 KiB / gzip: 0.22 KiB
dist/assets/index.8edffa56.js       1.14 KiB / gzip: 0.64 KiB
dist/assets/vendor.71e8fac3.js      128.58 KiB / gzip: 41.57 KiB
```


4. 开发环境特有插件

4.1 客户端环境变量注入插件

在开发环境中，Vite 会自动往 HTML 中注入一段 client 的脚本(点击查看实现):

```
<script type="module" src="/@vite/client"></script>
```

这段脚本主要提供 注入环境变量 、 处理 HMR 更新逻辑 、 构建出现错误时提供报错界面 等功能，而我们这里要介绍的 vite:client-inject 就是来完成时环境变量的注入，将 client 脚本中的 __MODE__ 、 __BASE__ 、 __DEFINE__ 等等字符串替换为运行时的变量，实现环境变量以及 HMR 相关上下文信息的注入，点击查看插件实现。

4.2 开发阶段 import 分析插件

最后，Vite 会在开发阶段加入 import 分析插件，即 vite:import-analysis 。与之前所介绍的 vite:build-import-analysis 相对应，主要处理 import 语句相关的解析和重写，但 vite:import-analysis 插件的关注点会不太一样，主要围绕 Vite 开发阶段的各项特性来实现，我们可以来梳理一下这个插件需要做哪些事情:

- 对 bare import，将路径名转换为真实的文件路径，如:

```
// 转换前
import 'foo'
// 转换后
// tip: 如果是预构建的依赖，则会转换为预构建产物的路径
import '/@fs/project/node_modules/foo/dist/foo.js'
```

主要调用 PluginContainer 的上下文对象方法即 this.resolve 实现，这个方法会调用所有插件的 resolveId 方法，包括之前介绍的 vite:pre-alias 和 vite:resolve ，完成路径解析的核心逻辑，点击查看实现。

- 对于 HMR 的客户端 API，即 import.meta.hot ，Vite 在识别到这样的 import 语句后，一方面会注入 import.meta.hot 的实现，因为浏览器原生并不具备这样的 API，点击查看注入代码；另一方面会识别 accept 方法，并判断 accept 是否为 接受自身更新 的类型(如果对 HMR 更新类型还不了解，可以回顾一下第十三节的内容)，如果是，则标记为上 isSelfAccepting 的 flag，便于 HMR 在服务端进行更新时进行 HMR Boundary 的查找。对于具体的查找过程，下一节会详细介绍。

- 对于全局环境变量读取语句，即 import.meta.env ，Vite 会注入 import.meta.env 的实现，也就是如下的 env 字符串:

```
// config 即解析完的配置
let env = `import.meta.env = ${JSON.stringify({
  ...config.env,
  SSR: !!ssr
})};`
// 对用户配置的 define 对象中，将带有 import.meta.env 前缀的全局变量挂到 import.meta.env 对象上
for (const key in config.define) {
  if (key.startsWith(`import.meta.env.`)) {
    const val = config.define[key]
    env += `${key} = ${
      typeof val === 'string' ? val : JSON.stringify(val)
    };`
  }
}
```

- 对于 `import.meta.glob` 语法，Vite 同样会调用之前提到的 `transformImportGlob` 函数来进行语法转换，但与生产环境的处理不同，在转换之后，Vite 会将该模块通过 glob 导入的依赖模块记录在 server 实例上，以便于 HMR 更新的时候能得到更准确的模块依赖信息，点击查看实现。

小结

好，本小节的内容讲完了。

这一节我们介绍了 Vite 的插件机制实现以及各个编译插件的作用和实现，信息密度比较大，需要你对照着官方的代码好好梳理一遍。其中，你需要重点掌握**PluginContainer 的实现机制**和 **Vite 内置插件各自的作用。**

首先，PluginContainer 主要由两部分实现，包括 Rollup 插件钩子的调度和插件钩子内部的 Context 上下文对象实现，总体上模拟了 Rollup 的插件机制。

其次，Vite 内置的插件包括四大类: **别名插件**、**核心构建插件**、**生产环境特有插件**和**开发环境特有插件。**这些插件包含了 Vite 核心的编译逻辑，可以说是 Vite 作为构建工具的命脉所在，希望你能对照本小节的内容及其对应的源码链接，了解各个插件的作用。

此外，在学习这些插件的过程中，我们切忌扎到众多繁琐的实现细节中，要尽可能抓关键的实现思路，来高效理解插件背后的原理，这样学习效率会更高。进一步来讲，在你理解了各个插件的实现原理之后，如果遇到某些场景下需要调试某些插件的代码，你也可以做到有的放矢。

最后，欢迎大家在评论区记录自己的学习收获和心得，也欢迎大家来一起讨论，把这部分的难点啃下来，让你对 Vite 底层的理解更上一层楼！

上一篇：依赖预构建：Esbuild 打包功能如何被 Vite 玩出花来？

下一篇： 热更新：基于 ESM 的毫秒级 HMR 的实现揭秘