配置解析服务：配置文件在 Vite 内部被转换成什么样子了？

发布于 2022-05-09

我们前面学习了 Vite 的各种高级应用场景，接下来的几个小节，我们再把目光放到 Vite 的实现本身，来深度剖析 Vite 的内部源码实现。

可能你会有一个疑问，我们为什么要去读源码？原因主要有两个：一是加深对框架本身的理解，在面对一些项目的疑难杂症时，排查问题效率会更高；二是在遇到类似的开发场景时，可以举一反三，借鉴某个框架源码的实现思路，将技巧应用到其它的项目中。

本小节我们要介绍 Vite 配置解析服务的源码部分。我们知道，Vite 构建环境分为 开发环境 和 生产环境 ，不同环境会有不同的构建策略，但不管是哪种环境，Vite 都会首先解析用户配置。那接下来，我就与你分析配置解析过程中 Vite 到底做了什么。

首先，我会带你梳理整体的实现流程，然后拆解其中的重点细节，即 如何加载配置文件 ，让你不仅对 Vite 的配置解析服务有系统且完整的认识，还能写一个自己的 配置文件加载器 。

流程梳理

我们先来梳理整体的流程，Vite 中的配置解析由 resolveConfig 函数来实现，你可以对照源码一起学习。

1. 加载配置文件

进行一些必要的变量声明后，我们进入到**解析配置**逻辑中:

```
// 这里的 config 是命令行指定的配置，如 vite --configFile=xxx
let { configFile } = config
if (configFile !== false) {
  // 默认都会走到下面加载配置文件的逻辑，除非你手动指定 configFile 为 false
```

```
  const loadResult = await loadConfigFromFile(
    configEnv,
    configFile,
    config.root,
    config.logLevel
  )
  if (loadResult) {
    // 解析配置文件的内容后，和命令行配置合并
    config = mergeConfig(loadResult.config, config)
    configFile = loadResult.path
    configFileDependencies = loadResult.dependencies
  }
}
```

第一步是解析配置文件的内容(这部分比较复杂，本文后续单独分析)，然后与命令行配置合并。值得注意的是，后面有一个记录 `configFileDependencies` 的操作。因为配置文件代码可能会有第三方库的依赖，所以当第三方库依赖的代码更改时，Vite 可以通过 HMR 处理逻辑中记录的 `configFileDependencies` 检测到更改，再重启 DevServer ，来保证当前生效的配置永远是最新的。

2. 解析用户插件

第二个重点环节是 **解析用户插件**。首先，我们通过 `apply 参数` 过滤出需要生效的用户插件。为什么这么做呢？因为有些插件只在开发阶段生效，或者说只在生产环境生效，我们可以通过 `apply: 'serve' 或 'build'` 来指定它们，同时也可以将 `apply` 配置为一个函数，来自定义插件生效的条件。解析代码如下:

```
// resolve plugins
const rawUserPlugins = (config.plugins || []).flat().filter((p) => {
  if (!p) {
    return false
  } else if (!p.apply) {
    return true
  } else if (typeof p.apply === 'function') {
     // apply 为一个函数的情况
    return p.apply({ ...config, mode }, configEnv)
  } else {
    return p.apply === command
  }
}) as Plugin[]
// 对用户插件进行排序
const [prePlugins, normalPlugins, postPlugins] =
  sortUserPlugins(rawUserPlugins)
```

接着，Vite 会拿到这些过滤且排序完成的插件，依次调用插件 config 钩子，进行配置合并

合并:

```
// run config hooks
const userPlugins = [...prePlugins, ...normalPlugins, ...postPlugins]
for (const p of userPlugins) {
  if (p.config) {
    const res = await p.config(config, configEnv)
    if (res) {
      // mergeConfig 为具体的配置合并函数，大家有兴趣可以阅读一下实现
      config = mergeConfig(config, res)
    }
  }
}
```

然后解析项目的根目录即 `root` 参数，默认取 `process.cwd()` 的结果:

```
// resolve root
const resolvedRoot = normalizePath(
  config.root ? path.resolve(config.root) : process.cwd()
)
```

紧接着处理 `alias` ，这里需要加上一些内置的 alias 规则，如 `@vite/env` 、`@vite/client` 这种直接重定向到 Vite 内部的模块:

```
// resolve alias with internal client alias
const resolvedAlias = mergeAlias(
  clientAlias,
  config.resolve?.alias || config.alias || []
)

const resolveOptions: ResolvedConfig['resolve'] = {
  dedupe: config.dedupe,
  ...config.resolve,
  alias: resolvedAlias
}
```

3. 加载环境变量

现在，我们进入第三个核心环节: **加载环境变量**，它的实现代码如下:

```
// load .env files
const envDir = config.envDir
  ? normalizePath(path.resolve(resolvedRoot, config.envDir))
  : resolvedRoot
const userEnv =
  inlineConfig.envFile !== false &&
  loadEnv(mode, envDir, resolveEnvPrefix(config))
```

loadEnv 其实就是扫描 `process.env` 与 `.env` 文件，解析出 env 对象，值得注意的是，这个对象的属性最终会被挂载到 `import.meta.env` 这个全局对象上。

解析 env 对象的实现思路如下:

- 遍历 process.env 的属性，拿到**指定前缀**开头的属性（默认指定为 `VITE_` ），并挂载 env 对象上

- 遍历 .env 文件，解析文件，然后往 env 对象挂载那些以**指定前缀**开头的属性。遍历的文件先后顺序如下(下面的 `mode` 开发阶段为 `development` ，生产环境为 `production`):

  - `.env.${mode}.local`
  - `.env.${mode}`
  - `.env.local`
  - `.env`

> 特殊情况: 如果中途遇到 NODE_ENV 属性，则挂到 `process.env.VITE_USER_NODE_ENV` ，Vite 会优先通过这个属性来决定是否走`生产环境` 的构建。

接下来是对资源公共路径即 `base URL` 的处理，逻辑集中在 resolveBaseUrl 函数当中:

```
// 解析 base url
const BASE_URL = resolveBaseUrl(config.base, command === 'build', logger)
// 解析生产环境构建配置
const resolvedBuildOptions = resolveBuildOptions(config.build)
```

`resolveBaseUrl` 里面有这些处理规则需要注意:

- 空字符或者 `./` 在开发阶段特殊处理，全部重写为 `/`
- `.` 开头的路径，自动重写为 `/`

- 以 `http(s)://` 开头的路径，在开发环境下重写为对应的 pathname
- 确保路径开头和结尾都是 `/`

当然，还有对 `cacheDir` 的解析，这个路径相对于在 Vite 预编译时写入依赖产物的路径:

```
// resolve cache directory
const pkgPath = lookupFile(resolvedRoot, [`package.json`], true /* pathOnly */)
// 默认为 node_module/.vite
const cacheDir = config.cacheDir
  ? path.resolve(resolvedRoot, config.cacheDir)
  : pkgPath && path.join(path.dirname(pkgPath), `node_modules/.vite`)
```

紧接着处理用户配置的 `assetsInclude` ，将其转换为一个过滤器函数:

```
const assetsFilter = config.assetsInclude
  ? createFilter(config.assetsInclude)
  : () => false
```

Vite 后面会将用户传入的 assetsInclude 和内置的规则合并:

```
assetsInclude(file: string) {
  return DEFAULT_ASSETS_RE.test(file) || assetsFilter(file)
}
```

这个配置决定是否让 Vite 将对应的后缀名视为 静态资源文件 （asset）来处理。

4. 路径解析器工厂

接下来，进入到第四个核心环节: **定义路径解析器工厂**。这里所说的 路径解析器 ，是指调用插件容器进行 路径解析 的函数。代码结构是这个样子的:

```
const createResolver: ResolvedConfig['createResolver'] = (options) => {
  let aliasContainer: PluginContainer | undefined
  let resolverContainer: PluginContainer | undefined
  // 返回的函数可以理解为一个解析器
  return async (id, importer, aliasOnly, ssr) => {
    let container: PluginContainer
    if (aliasOnly) {
      container =
        aliasContainer ||
        // 新建 aliasContainer
    } else {
      container
```

```
        container =
          resolverContainer ||
          // 新建 resolveContainer
      }
      return (await container.resolveId(id, importer, undefined, ssr))?.id
    }
  }
```

这个解析器未来会在**依赖预构建**的时候用上，具体用法如下:

```
const resolve = config.createResolver()
// 调用以拿到 react 路径
rseolve('react', undefined, undefined, false)
```

这里有 `aliasContainer` 和 `resolverContainer` 两个工具对象，它们都含有 `resolveId` 这个专门解析路径的方法，可以被 Vite 调用来获取解析结果。

> 两个工具对象的本质是 `PluginContainer` ，我们将在「编译流水线」小节详细介绍 `PluginContainer` 的特点和实现。

接着会顺便处理一个 public 目录，也就是 Vite 作为静态资源服务的目录:

```
const { publicDir } = config
const resolvedPublicDir =
  publicDir !== false && publicDir !== ''
    ? path.resolve(
        resolvedRoot,
        typeof publicDir === 'string' ? publicDir : 'public'
      )
    : ''
```

至此，配置已经基本上解析完成，最后通过 resolved 对象来整理一下:

```
const resolved: ResolvedConfig = {
  ...config,
  configFile: configFile ? normalizePath(configFile) : undefined,
  configFileDependencies,
  inlineConfig,
  root: resolvedRoot,
  base: BASE_URL
  // 其余配置不再一一列举
}
```

5. 生成插件流水线

最后，我们进入第五个环节: **生成插件流水线。**代码如下:

```
;(resolved.plugins as Plugin[]) = await resolvePlugins(
  resolved,
  prePlugins,
  normalPlugins,
  postPlugins
)

// call configResolved hooks
await Promise.all(userPlugins.map((p) => p.configResolved?.(resolved)))
```

先生成完整插件列表传给 `resolve.plugins` ，而后调用每个插件的 `configResolved` 钩子函数。其中 `resolvePlugins` 内部细节比较多，插件数量比较庞大，我们暂时不去深究具体实现，编译流水线这一小节再来详细介绍。

至此，所有核心配置都生成完毕。不过，后面 Vite 还会处理一些边界情况，在用户配置不合理的时候，给用户对应的提示。比如：用户直接使用 `alias` 时，Vite 会提示使用 `resolve.alias` 。

最后， `resolveConfig` 函数会返回 resolved 对象，也就是最后的配置集合，那么配置解析服务到底也就结束了。

加载配置文件详解

配置解析服务的流程梳理完，但刚开始 加载配置文件(`loadConfigFromFile`) 的实现我们还没有具体分析，先来回顾下代码。

```
const loadResult = await loadConfigFromFile(/*省略传参*/)
```

这里的逻辑稍微有点复杂，很难梳理清楚，所以我们不妨借助刚才梳理的配置解析流程，深入 `loadConfigFromFile` 的细节中，研究下 Vite 对于配置文件加载的实现思路。

首先，我们来分析下需要处理的配置文件类型，根据 文件后缀 和 模块格式 可以分为下面这几类:

- TS + ESM 格式
- TS + CommonJS 格式

- JS + ESM 格式
- JS + CommonJS 格式

那么，Vite 是如何加载配置文件的？一共分两个步骤:

> 识别出配置文件的类别
>
> 根据不同的类别分别解析出配置内容

1. 识别配置文件的类别

首先 Vite 会检查项目的 package.json ，如果有 `type: "module"` 则打上 `isESM` 的标识:

```
try {
  const pkg = lookupFile(configRoot, ['package.json'])
  if (pkg && JSON.parse(pkg).type === 'module') {
    isMjs = true
  }
} catch (e) {}
```

然后，Vite 会寻找配置文件路径，代码简化后如下:

```
let isTS = false
let isESM = false
let dependencies: string[] = []
// 如果命令行有指定配置文件路径
if (configFile) {
  resolvedPath = path.resolve(configFile)
  // 根据后缀判断是否为 ts 或者 esm，打上 flag
  isTS = configFile.endsWith('.ts')
  if (configFile.endsWith('.mjs')) {
      isESM = true
    }
} else {
  // 从项目根目录寻找配置文件路径，寻找顺序:
  // - vite.config.js
  // - vite.config.mjs
  // - vite.config.ts
  // - vite.config.cjs
  const jsconfigFile = path.resolve(configRoot, 'vite.config.js')
  if (fs.existsSync(jsconfigFile)) {
    resolvedPath = jsconfigFile
  }

  if (!resolvedPath) {
    const mjsconfigFile = path.resolve(configRoot, 'vite.config.mjs')
    if (fs.existsSync(mjsconfigFile)) {
```

```
      resolvedPath = mjsconfigFile

      isESM = true
    }
  }

  if (!resolvedPath) {
    const tsconfigFile = path.resolve(configRoot, 'vite.config.ts')
    if (fs.existsSync(tsconfigFile)) {
      resolvedPath = tsconfigFile
      isTS = true
    }
  }

  if (!resolvedPath) {
    const cjsConfigFile = path.resolve(configRoot, 'vite.config.cjs')
    if (fs.existsSync(cjsConfigFile)) {
      resolvedPath = cjsConfigFile
      isESM = false
    }
  }
}
```

在寻找路径的同时， Vite 也会给当前配置文件打上 `isESM` 和 `isTS` 的标识，方便后续的解析。

2. 根据类别解析配置

ESM 格式

对于 ESM 格式配置的处理代码如下：

```
let userConfig: UserConfigExport | undefined

if (isESM) {
  const fileUrl = require('url').pathToFileURL(resolvedPath)
  // 首先对代码进行打包
  const bundled = await bundleConfigFile(resolvedPath, true)
  dependencies = bundled.dependencies
  // TS + ESM
  if (isTS) {
    fs.writeFileSync(resolvedPath + '.js', bundled.code)
    userConfig = (await dynamicImport(`${fileUrl}.js?t=${Date.now()}`))
      .default
    fs.unlinkSync(resolvedPath + '.js')
    debug(`TS + native esm config loaded in ${getTime()}`, fileUrl)
  }
  //  JS + ESM
  else {
    userConfig = (await dynamicImport(`${fileUrl}?t=${Date.now()}`)).default
    debug(`native esm config loaded in ${getTime()}`, fileUrl)
```

```
  }
}
```

首先通过 Esbuild 将配置文件编译打包成 js 代码:

```
const bundled = await bundleConfigFile(resolvedPath, true)
// 记录依赖
dependencies = bundled.dependencies
```

对于 TS 配置文件来说，Vite 会将编译后的 js 代码写入 临时文件 ，通过 Node 原生 ESM Import 来读取这个临时的内容，以获取到配置内容，再直接删掉临时文件:

```
fs.writeFileSync(resolvedPath + '.js', bundled.code)
userConfig = (await dynamicImport(`${fileUrl}.js?t=${Date.now()}`)).default
fs.unlinkSync(resolvedPath + '.js')
```

> 以上这种先编译配置文件，再将产物写入临时目录，最后加载临时目录产物的做法，也是 AOT (Ahead Of Time)编译技术的一种具体实现。

而对于 JS 配置文件来说，Vite 会直接通过 Node 原生 ESM Import 来读取，也是使用 dynamicImport 函数的逻辑。 `dynamicImport` 的实现如下:

```
export const dynamicImport = new Function('file', 'return import(file)')
```

你可能会问，为什么要用 new Function 包裹？这是为了避免打包工具处理这段代码，比如 `Rollup` 和 `TSC` ，类似的手段还有 `eval` 。

你可能还会问，为什么 import 路径结果要加上时间戳 query？这其实是为了让 dev server 重启后仍然读取最新的配置，避免缓存。

CommonJS 格式

对于 CommonJS 格式的配置文件，Vite 集中进行了解析:

```
// 对于 js/ts 均生效
// 使用 esbuild 将配置文件编译成 commonjs 格式的 bundle 文件
const bundled = await bundleConfigFile(resolvedPath)
```

```
dependencies = bundled.dependencies

// 加载编译后的 bundle 代码
userConfig = await loadConfigFromBundledFile(resolvedPath, bundled.code)
```

`bundleConfigFile` 的逻辑上文中已经说了，主要是通过 Esbuild 将配置文件打包，拿到打包后的 bundle 代码以及配置文件的依赖(dependencies)。

而接下来的事情就是考虑如何加载 bundle 代码了，这也是 `loadConfigFromBundledFile` 要做的事情。我们来看一下这个函数具体的实现:

```
async function loadConfigFromBundledFile(
  fileName: string,
  bundledCode: string
): Promise<UserConfig> {
  const extension = path.extname(fileName)
  const defaultLoader = require.extensions[extension]!
  require.extensions[extension] = (module: NodeModule, filename: string) => {
    if (filename === fileName) {
      ;(module as NodeModuleWithCompile)._compile(bundledCode, filename)
    } else {
      defaultLoader(module, filename)
    }
  }
  // 清除 require 缓存
  delete require.cache[require.resolve(fileName)]
  const raw = require(fileName)
  const config = raw.__esModule ? raw.default : raw
  require.extensions[extension] = defaultLoader
  return config
}
```

大体的思路是通过拦截原生 require.extensions 的加载函数来实现对 bundle 后配置代码的加载。代码如下:

```
// 默认加载器
const defaultLoader = require.extensions[extension]!
// 拦截原生 require 对于`.js`或者`.ts`的加载
require.extensions[extension] = (module: NodeModule, filename: string) => {
  // 针对 vite 配置文件的加载特殊处理
  if (filename === fileName) {
    ;(module as NodeModuleWithCompile)._compile(bundledCode, filename)
  } else {
    defaultLoader(module, filename)
  }
}
```

而原生 require 对于 js 文件的加载代码是这样的:

```
Module._extensions['.js'] = function (module, filename) {
  var content = fs.readFileSync(filename, 'utf8')
  module._compile(stripBOM(content), filename)
}
```

Node.js 内部也是先读取文件内容，然后编译该模块。当代码中调用 `module._compile` 相当于手动编译一个模块，该方法在 Node 内部的实现如下:

```
Module.prototype._compile = function (content, filename) {
  var self = this
  var args = [self.exports, require, self, filename, dirname]
  return compiledWrapper.apply(self.exports, args)
}
```

等同于下面的形式:

```
;(function (exports, require, module, __filename, __dirname) {
  // 执行 module._compile 方法中传入的代码
  // 返回 exports 对象
})
```

在调用完 `module._compile` 编译完配置代码后，进行一次手动的 require，即可拿到配置对象:

```
const raw = require(fileName)
const config = raw.__esModule ? raw.default : raw
// 恢复原生的加载方法
require.extensions[extension] = defaultLoader
// 返回配置
return config
```

> 这种运行时加载 TS 配置的方式，也叫做 `JIT` (即时编译)，这种方式和 `AOT` 最大的区别在于不会将内存中计算出来的 js 代码写入磁盘再加载，而是通过拦截 Node.js 原生 require.extension 方法实现即时加载。

至此，配置文件的内容已经读取完成，等后处理完成再返回即可:

```
// 处理是函数的情况
const config = await (typeof userConfig === 'function'
  ? userConfig(configEnv)
```

```
    : userConfig)

  if (!isObject(config)) {
    throw new Error(`config must export or return an object.`)
  }
  // 接下来返回最终的配置信息
  return {
    path: normalizePath(resolvedPath),
    config,
    // esbuild 打包过程中搜集的依赖
    dependencies
  }
```

总结

配置解析的源码精读部分到这里就结束了，再次恭喜你，学习完了本小节的内容。本小节中，你需要重点掌握 Vite 配置解析的整体流程 和 加载配置文件的方法 。

首先，Vite 配置文件解析的逻辑由 resolveConfig 函数统一实现，其中经历了加载配置文件、解析用户插件、加载环境变量、创建路径解析器工厂和生成插件流水线这几个主要的流程。

其次，在 加载配置文件 的过程中，Vite 需要处理四种类型的配置文件，其中对于 ESM 和 CommonJS 两种格式的 TS 文件，分别采用了 AOT 和 JIT 两种编译技术实现了配置加载。

最后，我想留一个问题: 如果现在让你设计一个 cli 工具，用来支持 TS 的配置文件，你会如何进行配置解析呢?

上一篇：性能优化: 如何体系化地对 Vite 项目进行性能优化?

下一篇：依赖预构建：Esbuild 打包功能如何被 Vite 玩出花来?