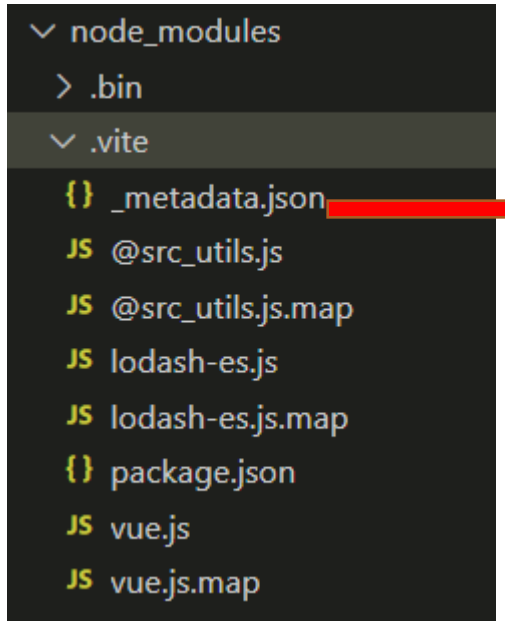
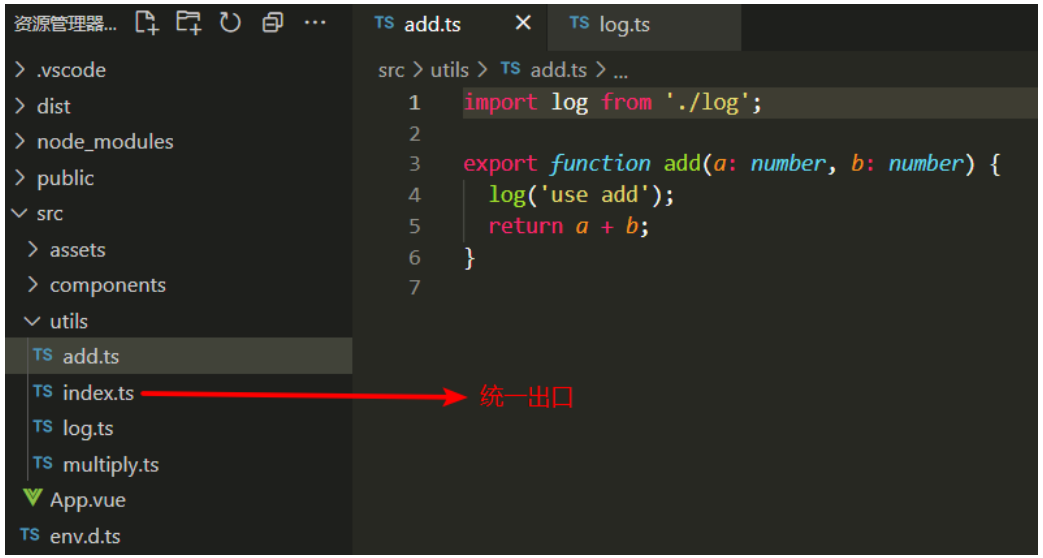
核心吸引力：
使用浏览器原生ES模块功能，不做耗时的打包，达到极速的启动速度。同时达到真正的按需加载。

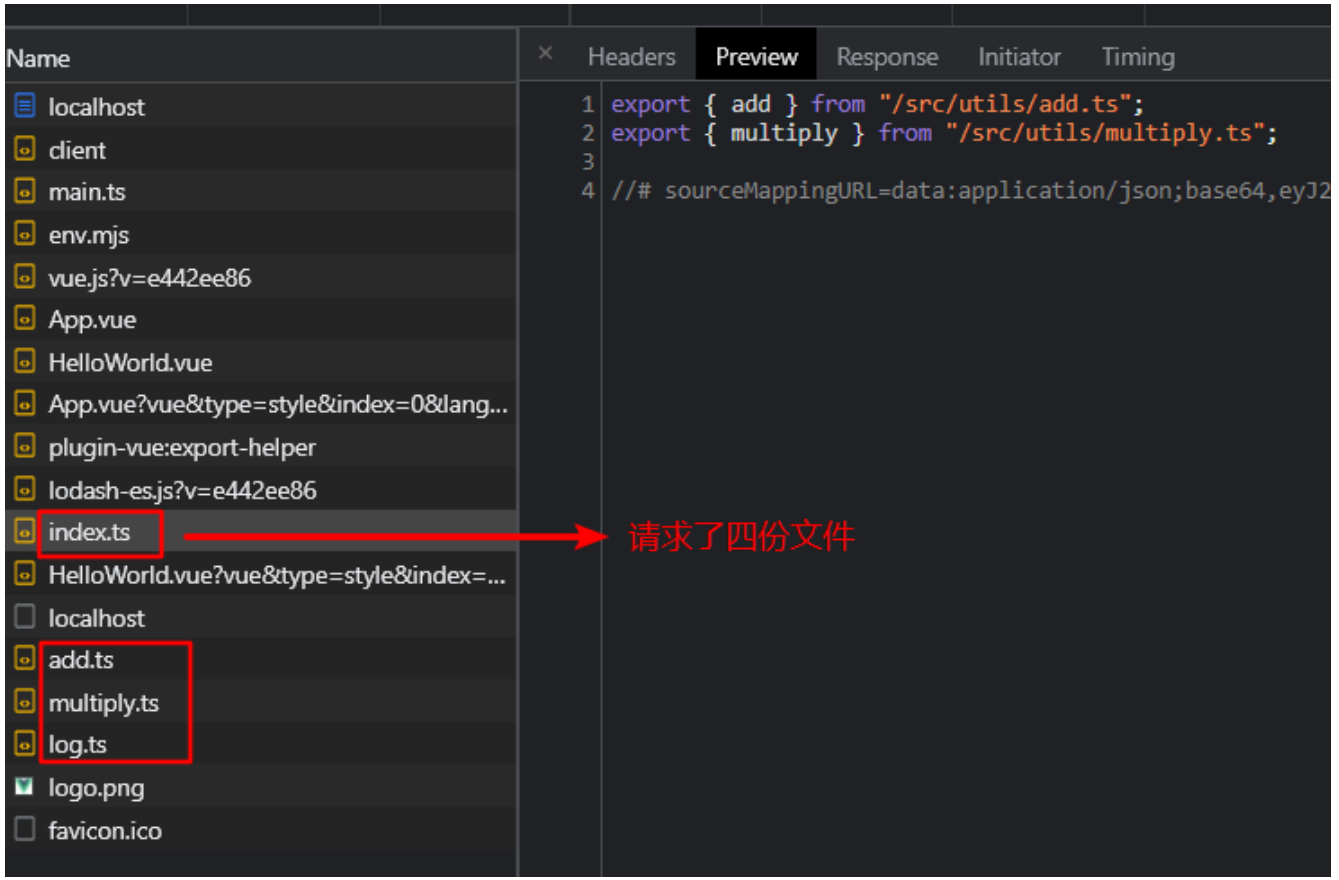
问题一：
第三方不是Esmodule的依赖库要预先处理成ES输出，暂存起来。使用预构建，将处理结果打包输出到node_modules/.vite。注意下方的_metadata.json，记录了“优化依赖元数据”。




```
{
  "hash": "a088e72c",
  "browserHash": "5ee9c455",
  "optimized": {
    "vue": {
      "file": "${root}/vite/node_modules/.vite/vue.js",
      "src": "${root}/vite/node_modules/vue/dist/vue.runtime.esm-bundler.js",
      "needsInterop": false
    },
    "lodash-es": {
      "file": "${root}/vite/node_modules/.vite/lodash-es.js",
      "src": "${root}/vite/node_modules/lodash-es/lodash.js",
      "needsInterop": false
    },
    "@src/utils": {
      "file": "${root}/vite/node_modules/.vite/@src_utils.js",
      "src": "${root}/vite/src/utils/index.ts",
      "needsInterop": false
    }
  }
}
```

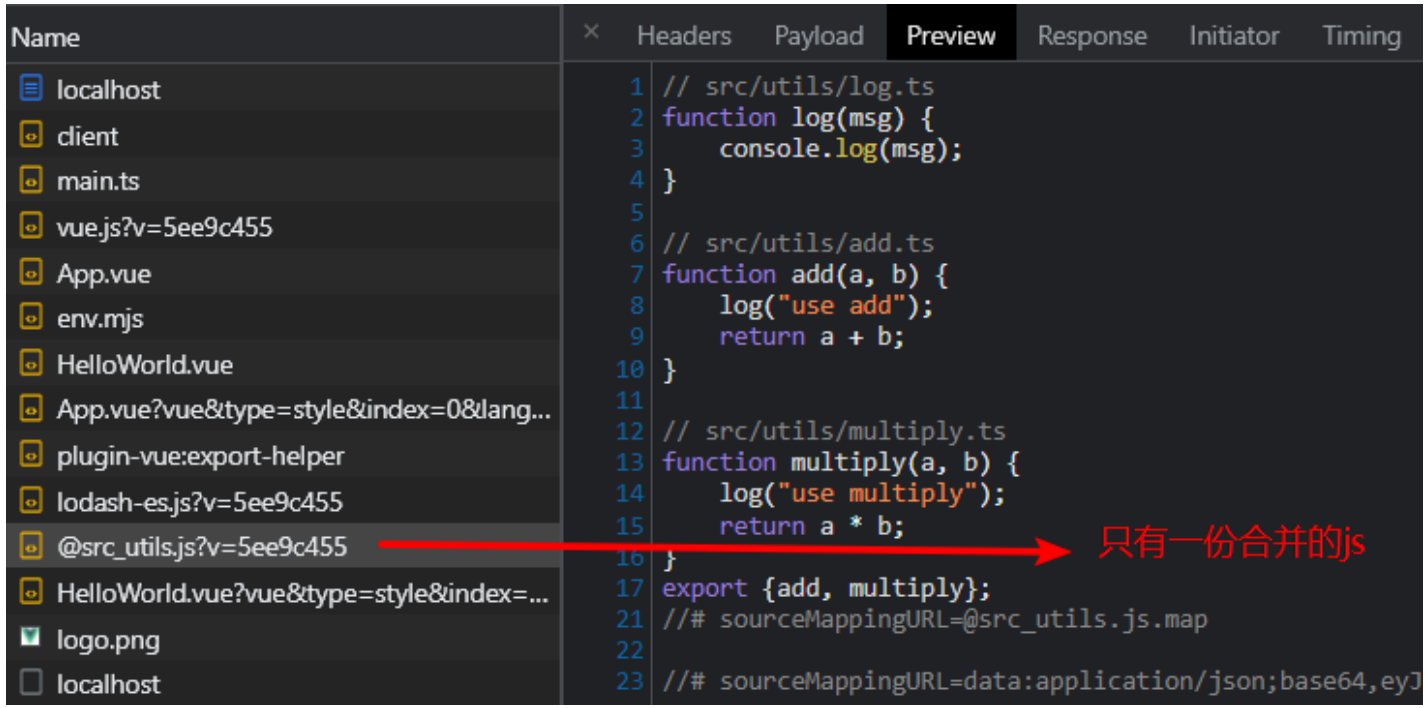
预构建还有一个好处，就是把一些零散资源打成一份，减少请求次数。
示例：写几份工具方法，在不预处理的情况下，查看请求情况





接着把这几份方法的入口文件配置进优化项，在vite.config进行配置，再查看网络请求情况

```
// https://vitejs.dev/config/
export default defineConfig({
  plugins: [vue()],
  resolve: {
    alias: {
      '@src': path.resolve(process.cwd(), 'src'),
    }
  },
  optimizeDeps: {
    include: ['@src/utils'],
  },
  clearScreen: false,
```



问题二：
依赖资源的请求。以前面说的@src_utils.js为例，我们代码里写的都是分散地对ES module的引用，但实际请求的确实只有一份打包合并的文件。所以必然在运行时挟持了请求路径。这里借助一下modern.js官网的文字给予解释。vite server也是这么处理的。

依赖预处理

当前，很多第三方依赖只提供了 CommonJS 产物，无法直接在浏览器中运行，另外，即使第三方依赖提供 ESM 产物，如果按照习惯使用，例如:

```
import { something } from 'some-package';
```

在浏览器中直接运行也会报错。 Modern.js 为了解决上述问题，会对第三方依赖进行如下处理。

对第三方依赖进行如下处理方式

1. 首次启动 Dev Server 时，分析项目源代码，找出使用到的第三方依赖，例如 react、react-dom 等。
2. 根据依赖在 `node_modules/` 目录下的实际安装位置，获取精确的版本号信息。
3. 根据包名和版本号，依次检查是否命中本地缓存和 Modern.js 的云端缓存，均未命中的情况下，本地编译该模块，转换为 ESM 格式。后续针对获取到的 ESM 文件，使用 esbuild 打包成一个文件，以减少项目运行时浏览器中的请求数量。
4. Dev Server 启动时，动态改写源码文件中对第三方依赖的引用路径，例如：

   ```
   import { useState } from 'react'
   ```

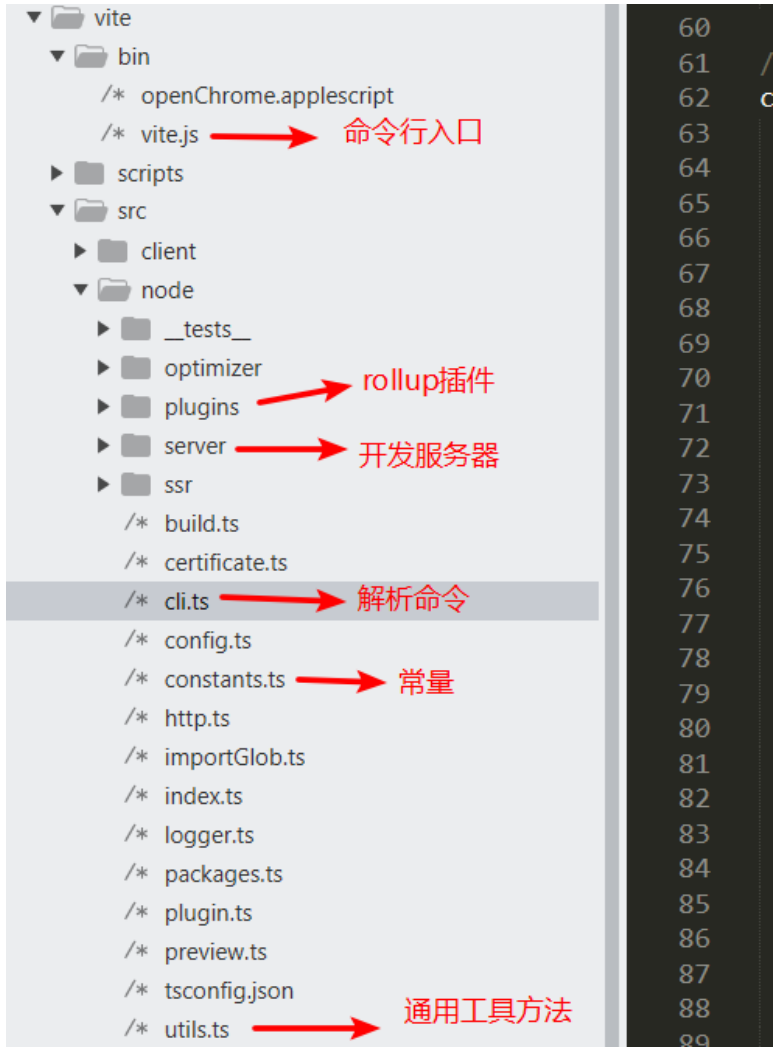
   会被改写为：

   ```
   import { useState } from 'node_modules/.modern_js_web_modules/react.js'
   ```

   从而保证浏览器能够正确加载第三方依赖。

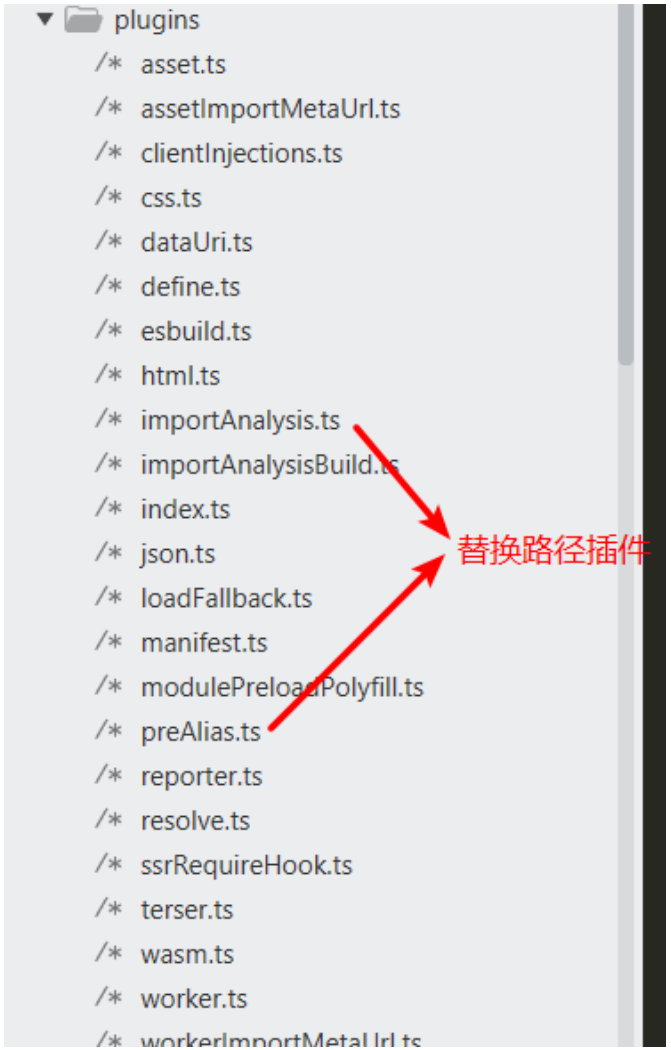
源码解析
现在就来看看vite具体是怎么做。我们从目录结构看起：

vite总目录：



server目录：

plugins目录：



从cli解析命令入口：
vite不带参数即默认启动开发，即 vite === vite serve === vite dev

```
cli
  .command('[root]', 'start dev server') // default command
  .alias('serve') // the command is called 'serve' in Vite's API
  .alias('dev') // alias to align with the script name
  .option('--host [host]', `[string] specify hostname`)
  .option('--port <port>', `[number] specify port`)
  .option('--https', `[boolean] use TLS + HTTP/2`)
  .option('--open [path]', `[boolean | string] open browser on startup`)
  .option('--cors', `[boolean] enable CORS`)
  .option('--strictPort', `[boolean] exit if specified port is already in use`)
  .option(
    '--force',
    `[boolean] force the optimizer to ignore the cache and re-bundle`
  )
  .action(async (root: string, options: ServerOptions & GlobalCLIOptions) => {
    // output structure is preserved even after bundling so require()
    // is ok here
    const { createServer } = await import('./server')
    try {
      const server = await createServer({
        root,
        base: options.base,
        mode: options.mode,
        configFile: options.config,
        logLevel: options.logLevel,
        clearScreen: options.clearScreen,
        server: cleanOptions(options)
      })

      if (!server.httpServer) {
        throw new Error('HTTP server not available')
      }

      await server.listen()
```

这里可以看到创建了一个服务器对象，然后进行监听端口操作，其实就是启动服务器。所以所有文章都是在这个服务器上。

createServer返回了什么功能的服务器实例?

```
export async function createServer(
  inlineConfig: InlineConfig = {}
): Promise<ViteDevServer> {
  const config = await resolveConfig(inlineConfig, 'serve', 'development')
  const root = config.root
  const serverConfig = config.server
  const httpsOptions = await resolveHttpsConfig(
    config.server.https,
    config.cacheDir
  )
  let { middlewareMode } = serverConfig
  if (middlewareMode === true) {
    middlewareMode = 'ssr'
  }

  const middlewares = connect() as Connect.Server
  const httpServer = middlewareMode
    ? null
    : await resolveHttpServer(serverConfig, middlewares, httpsOptions)
  const ws = createWebSocketServer(httpServer, config, httpsOptions)

  const { ignored = [], ...watchOptions } = serverConfig.watch || {}
  const watcher = chokidar.watch(path.resolve(root), {
    ignored: [
      '**/node_modules/**',
      '**/.git/**',
      ...(Array.isArray(ignored) ? ignored : [ignored])
    ],
    ignoreInitial: true,
    ignorePermissionErrors: true,
    disableGlobbing: true,
    ...watchOptions
  }) as FSWatcher

  const moduleGraph: ModuleGraph = new ModuleGraph((url, ssr) =>
    container.resolveId(url, undefined, { ssr })
  )

  const container = await createPluginContainer(config, moduleGraph, watcher)
  const closeHttpServer = createServerCloseFn(httpServer)

  // eslint-disable-next-line prefer-const
  let exitProcess: () => void
```

config：所有配置，注意这里config收集了所有plugin。

middlewares：一个基础服务应用，可以看做类似express或者koa的app实例。

httpServer：根据config及上述的基础服务应用创建出来的服务器实例。

container：根据config等参数创建的插件容器。上面封装了调用所有插件的各个方法，用于对内容进行处理。
注意：这里各个插件是以rollup插件格式编写的，但是serve时只是共用上面的处理方法，并没有体现其rollup插件的一面。

resolveConfig方法：

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

```
export async function resolvePlugins(
  config: ResolvedConfig,
  prePlugins: Plugin[],
  normalPlugins: Plugin[],
  postPlugins: Plugin[]
): Promise<Plugin[]> {
  const isBuild = config.command === 'build'

  const buildPlugins = isBuild
    ? (await import('../build')).resolveBuildPlugins(config)
    : { pre: [], post: [] }

  return [
    isBuild ? null : preAliasPlugin(),
    aliasPlugin({ entries: config.resolve.alias }),
    ...prePlugins,
    config.build.polyfillModulePreload ...
    resolvePlugin({ ...
    }),
    htmlInlineProxyPlugin(config),
    cssPlugin(config),
    config.esbuild !== false ? esbuildPlugin(config.esbuild) : null,
    jsonPlugin( ...
    ),
    wasmPlugin(config),
    webWorkerPlugin(config),
    workerImportMetaUrlPlugin(config),
    assetPlugin(config),
    ...normalPlugins,
    definePlugin(config),
    cssPostPlugin(config),
    config.build.ssr ? ssrRequireHookPlugin(config) : null,
    ...buildPlugins.pre,
    ...postPlugins,
    ...buildPlugins.post,
    // internal server-only plugins are always applied after everything
    ...(isBuild
      ? []
      : [clientInjectionsPlugin(config), importAnalysisPlugin(config)])
  ].filter(Boolean) as Plugin[]
}
```

```
export function importAnalysisPlugin(config: ResolvedConfig): Plugin {
  const { root, base } = config
  const clientPublicPath = path.posix.join(base, CLIENT_PUBLIC_PATH)

  let server: ViteDevServer

  return {
    name: 'vite:import-analysis',

    configureServer(_server) {
      server = _server
    },

    async transform(source, importer, options) {
      const ssr = options?.ssr === true
      const prettyImporter = prettifyUrl(importer, root)

      if (canSkip(importer)) {
        isDebug && debug(colors.dim(`[skipped] ${prettyImporter}`))
        return null
      }

      const start = performance.now()
      await init
      let imports: readonly ImportSpecifier[] = []
      // strip UTF-8 BOM
      if (source.charCodeAt(0) === 0xfeff) {
        source = source.slice(1)
      }
      try {
```

importAnalysisPlugin插件返回的对象上有transform方法。这里重点关注该插件，因为它就是负责处理替换文件的依赖路径。

container对象（由createPluginContainer方法返回）：

```
export async function createPluginContainer(
  { plugins, logger, root, build: { rollupOptions } }: ResolvedConfig,
  moduleGraph?: ModuleGraph,
  watcher?: FSWatcher
```

config作为参数传进，所以这里可以获取到所有插件。

```
class Context implements PluginContext {
  meta = minimalContext.meta
  ssr = false
  _activePlugin: Plugin | null
  _activeId: string | null = null
  _activeCode: string | null = null
  _resolveSkips?: Set<Plugin>
  _addedImports: Set<string> | null = null
  constructor(initialPlugin?: Plugin) {
    this._activePlugin = initialPlugin || null
  }
  parse(code: string, opts: any = {}) {
    return parser.parse(code, {
    })
  }
  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
  }
```

```
class TransformContext extends Context {
}
```

Context继承自PluginContext并且定义了resolve方法。
TransformContext继承自Context，所以它也有resolve方法。

container对象（由createPluginContainer方法返回）：

```
const container: PluginContainer = {
  options: await (async () => { ···
  })(),

  getModuleInfo,

  async buildStart() { ···
  },

  async resolveId(rawId, importer = join(root, 'index.html'), options) { ···
  },

  async load(id, options) { ···
  },

  async transform(code, id, options) {
    const inMap = options?.inMap
    const ssr = options?.ssr
    const ctx = new TransformContext(id, code, inMap as SourceMap)
    ctx.ssr = !!ssr
    for (const plugin of plugins) {
      if (!plugin.transform) continue
      ctx._activePlugin = plugin
      ctx._activeId = id
      ctx._activeCode = code
      const start = isDebug ? performance.now() : 0
      let result: TransformResult | string | undefined
      try {
        result = await plugin.transform.call(ctx as any, code, id, { ssr })
      } catch (e) {
        ctx.error(e)
      }
```

container上有几个与插件类似的方法。其实就是遍历各个插件，如果有相应的方法，就调用来处理返回内容。
transform方法里初始化了一个ctx=TransformContext实例，此时ctx实例是有resolve方法的。
在调用插件的transform方法时，改变了this指向到ctx，所以插件方法里面用this.resolve方法则是使用到以下方法：

```
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
}
```

注意这里又继续调用了container的resolveId方法

```
async resolveId(rawId, importer = join(root, 'in
  const skip = options?.skip
  const ssr = options?.ssr
  const ctx = new Context()
  ctx.ssr = !!ssr
  ctx._resolveSkips = skip
  const resolveStart = isDebug ? performance.now

  let id: string | null = null
  const partial: Partial<PartialResolvedId> = {}
  for (const plugin of plugins) {
    if (!plugin.resolveId) continue
    if (skip?.has(plugin)) continue

    ctx._activePlugin = plugin

    const pluginResolveStart = isDebug ? perform
    const result = await plugin.resolveId.call(
      ctx as any,
      rawId,
      importer,
      { ssr }
    )
    if (!result) continue
```

再继续看createServer里的middlewares：

```
// serve static files under /public
// this applies before the transform middleware so that these files are served
// as-is without transforms.
if (config.publicDir) {
  middlewares.use(servePublicMiddleware(config.publicDir))
}

// main transform middleware
middlewares.use(transformMiddleware(server))

// serve static files
middlewares.use(serveRawFsMiddleware(server))
middlewares.use(serveStaticMiddleware(root, server))

// spa fallback
if (!middlewareMode || middlewareMode === 'html') {
  middlewares.use(spaFallbackMiddleware(root))
}
```

servePublicMiddleware：
处理public文件夹下的资源请求

transformMiddleware：
处理文件内容转换（重点中间件）

serveRawFsMiddleware：
处理根目录外的文件请求

serveStaticMiddleware：
处理其他格式文件（如图片）的请求

最后在看createServer里的一步重要操作：

```
const runOptimize = async () => {
  server._isRunningOptimizer = true
  try {
    server._optimizeDepsMetadata = await optimizeDeps(
      config,
      config.server.force || server._forceOptimizeOnRestart
    )
  } finally {
    server._isRunningOptimizer = false
  }
  server._registerMissingImport = createMissingImporterRegisterFn(server)
}

if (!middlewareMode && httpServer) {
  let isOptimized = false
  // overwrite listen to run optimizer before server start
  const listen = httpServer.listen.bind(httpServer)
  httpServer.listen = (async (port: number, ...args: any[]) => {
    if (!isOptimized) {
      try {
        await container.buildStart({})
        await runOptimize()
        isOptimized = true
      } catch (e) {
        httpServer.emit('error', e)
        return
      }
    }
    return listen(port, ...args)
  }) as any
} else {
  await container.buildStart({})
  await runOptimize()
}
```

runOptimize：
里面执行了optimizeDeps方法，把返回值赋予了server._optimizeDepsMetadata。这个值就是前面提到“优化依赖元数据”。也就是在运行过程中，根据这份数据上的映射来转换成真正的请求路径。

接着这里改写了服务器实例的listen方法，给其加上执行container.buildStart（这里实际上就是执行各个rollup上有的buildStart方法）。已经执行runOptimize，获得“优化依赖元数据”，挂载到server对象上。

现在我们知道替换内容的操作是在transformMiddleware中间件里，请求在这里判断处理，将获取到的原始内容进行转换，然后返回。现在看一下transformMiddleware中间件的内容：

```
// resolve, load and transform using the plugin container
const result = await transformRequest(url, server, {
  html: req.headers.accept?.includes('text/html')
})
if (result) {
  const type = isDirectCSSRequest(url) ? 'css' : 'js'
  const isDep =
    DEP_VERSION_RE.test(url) ||
    (cacheDirPrefix && url.startsWith(cacheDirPrefix))
  return send(req, res, result.code, type, {
    etag: result.etag,
    // allow browser to cache npm deps!
    cacheControl: isDep ? 'max-age=31536000,immutable' : 'no-cache',
    headers: server.config.server.headers,
    map: result.map
  })
}
```

跳过前面复杂多情况的判断，直接到获取内容的地方。通过transformRequest方法获取了一个result对象，然后在下方执行了send方法。

send方法前两个参数是中间件的参数：请求和响应方法。第三个是result.code，基本可以知道send方法里面通过响应方法把code内容返回请求。

所以transformRequest就是处理请求，返回实际文件内容的操作了。

所以重点看看transformRequest 怎么处理请求进而返回内容的:

```ts
export function transformRequest(
  url: string,
  server: ViteDevServer,
  options: TransformOptions = {}
): Promise<TransformResult | null> {
  const cacheKey = (options.ssr ? 'ssr:' : options.html ? 'html:' : '') + url
  let request = server._pendingRequests.get(cacheKey)
  if (!request) {
    request = doTransform(url, server, options)
    server._pendingRequests.set(cacheKey, request)
    const done = () => server._pendingRequests.delete(cacheKey)
    request.then(done, done)
  }
  return request
}
```

请求的地址

挂载了很多东西的那个server对象

有缓存拿缓存

结果由这个函数返回

可以看出核心处理方法是doTransform

doTransform重点步骤:

由请求路径得出本地资源绝对路径:

```ts
// resolve
const id =
  (await pluginContainer.resolveId(url, undefined, { ssr }))?.id || url
const file = cleanUrl(id)

let code: string | null = null
let map: SourceDescription['map'] = null
```

根据绝对路径读出文件内容:

```ts
if (options.ssr || isFileServingAllowed(file, server)) {
  try {
    code = await fs.readFile(file, 'utf-8')
    isDebug && debugLoad(`${timeFrom(loadStart)} [fs] ${prettyUrl}`)
  } catch (e) {
    if (e.code !== 'ENOENT') {
      throw e
    }
  }
}
```

转换处理文件内容:

```ts
// transform
const transformStart = isDebug ? performance.now() : 0
const transformResult = await pluginContainer.transform(code, id, {
  inMap: map,
  ssr
})
```

以/src/main.ts为例，看看这个处理流程：

/src/main.ts内容如下：

```
import { createApp } from 'vue'
import App from './App.vue'

createApp(App).mount('#app')
```

1、由pluginContainer.resolveId得出文件的绝对地址：${root}/ src/main.ts

2、由fs.readFile读取出文件内容，即左图上代码

3、由pluginContainer.transform（各插件的transform ）得出处理后要返回的内容

得出的transformResult如下：
可以看到原来的“vue”依赖路径已经被替换成“/node_modules/.vite/vue.js?v=3bd8dabc”

```
/src/main.ts ▼{code: 'import { createApp } from "/node_modules/.vite/vue…om "/src/App.vue";\ncreateApp(App).mount("#app");\n', map: {…}}
                code: "import { createApp } from \"/node_modules/.vite/vue.js?v=3bd8dabc\";\nimport App from \"/src/App.vue\";\ncreateApp(App).mount(\"#app\");\n"
              ▶ map: {version: 3, sources: Array(1), sourcesContent: Array(1), mappings: 'AAAA;AACA;AAEA,UAAU,KAAK,MAAM;', names: Array(0)}
              ▶ [[Prototype]]: Object
```

```
if (
  transformResult == null ||
  (isObject(transformResult) && transformResult.code == null)
) {
  // no transform applied, keep code as-is
  isDebug &&
    debugTransform(
      timeFrom(transformStart) + colors.dim(` [skipped] ${prettyUrl}`)
    )
} else {
  isDebug && debugTransform(`${timeFrom(transformStart)} ${prettyUrl}`)
  code = transformResult.code!
  map = transformResult.map
}
```

赋值code和map

```
if (ssr) {
  return (mod.ssrTransformResult = await ssrTransform(
    code,
    map as SourceMap,
    url
  ))
} else {
  return (mod.transformResult = {
    code,
    map,
    etag: getEtag(code, { weak: true })
  } as TransformResult)
}
```

把code和map返回出去

importAnalysisPlugin插件内容：

结合前文我们知道开发代码是经过各个插件的transform 方法一部分一部分转换的，其中importAnalysisPlugin负责替换依赖资源的路径，所以继续看看importAnalysisPlugin插件：

这里先介绍一下es-module-lexer模块，是专门用来解析import语法，提取如“import vue from ‘vue’”的组成ast。并且可以从ast反向输出import语句字符串。词法解析三件套：parse->transform->generate。

```
export function importAnalysisPlugin(config: ResolvedConfig): Plugin {
  const { root, base } = config
  const clientPublicPath = path.posix.join(base, CLIENT_PUBLIC_PATH)
  let server: ViteDevServer
  return {
    name: 'vite:import-analysis',
    configureServer(_server) {
      server = _server
    },
    async transform(source, importer, options) {
      const ssr = options?.ssr === true
      const prettyImporter = prettifyUrl(importer, root)
      if (canSkip(importer)) {
        isDebug && debug(colors.dim(`[skipped] ${prettyImporter}`))
        return null
      }
      const start = performance.now()
      await init
      let imports: readonly ImportSpecifier[] = []
      // strip UTF-8 BOM
      if (source.charCodeAt(0) === 0xfeff) {
        source = source.slice(1)
      }
      try {
        imports = parseImports(source)[0]
```

文件绝对路径

解析import语法

```
▼ (2) [{…}, {…}]
  ▼ 0:
      a: -1
      d: -1
      e: 30
      n: "vue"
      s: 27
      se: 31
      ss: 0
    ▶ [[Prototype]]: Object
  ▼ 1:
      a: -1
      d: -1
      e: 59
      n: "./App.vue"
      s: 50
      se: 60
      ss: 33
    ▶ [[Prototype]]: Object
    length: 2
  ▶ [[Prototype]]: Array(0)
```

n就是模块路径

```
let s: MagicString | undefined
const str = () => s || (s = new MagicString(source))
// vite-only server context
const { moduleGraph } = server
// since we are already in the transform phase of the importer, i
// have been loaded so its entry is guaranteed in the module grap
const importerModule = moduleGraph.getModuleById(importer)!
const importedUrls = new Set<string>()
const staticImportedUrls = new Set<string>()
const acceptedUrls = new Set<{
  url: string
  start: number
  end: number
}>()
const toAbsoluteUrl = (url: string) =>
  path.posix.resolve(path.posix.dirname(importerModule.url), url)

const normalizeUrl = async (
  url: string,
  pos: number
): Promise<[string, string]> => {
```

这里注意一下初始化了一个MagicString对象，这个对象提供了插入前后置、替换文本等功能。以及normalizeUrl方法，用以获取替换后的依赖路径。

importAnalysisPlugin插件内容：

```
for (let index = 0; index < imports.length; index++) {
  const {                    遍历处理每个import语句的解析对象
    s: start,
    e: end,
    ss: expStart,
    se: expEnd,
    d: dynamicIndex,
    // #2083 User may use escape path,
    // so use imports[index].n to get the unescaped string
    // @ts-ignore
    n: specifier
  } = imports[index]          得出原始依赖模块

  const rawUrl = source.slice(start, end)

  // check import.meta usage
  if (rawUrl === 'import.meta') {
```

```
// normalize
const [normalizedUrl, resolvedId] = await normalizeUrl(
  specifier,
  start
)
let url = normalizedUrl   得出替换后的依赖路径
```

依然是以/src/main.ts请求为例，normalizeUrl的运行会是：

输入：

specifier — vue

start — 27

输出:

normalizedUrl — /node_modules/.vite/vue.js?v=3bd8dabc

resolvedId — ${root}/vite/node_modules/.vite/vue.js?v=3bd8dabc

```
// rewrite
if (url !== specifier) {
  // for optimized cjs deps, support named imports by rewriting named
  // imports to const assignments.
  if (resolvedId.endsWith(`&es-interop`)) {
    url = url.slice(0, -11)
    if (isDynamicImport) {…
    } else {…
    }                          对source内容重写，就是替换依赖路径
  } else {
    str().overwrite(start, end, isDynamicImport ? `'${url}'` : url)
  }
}
```

```
if (s) {
  return s.toString()
} else {
  return source
}
```

文本化后返回输出

normalizeUrl方法内容：

```
const normalizeUrl = async (
  url: string,
  pos: number
): Promise<[string, string]> => {
  if (base !== '/' && url.startsWith(base)) {
    url = url.replace(base, '/')
  }
  let importerFile = importer
  if (
    moduleListContains(config.optimizeDeps?.exclude, url) &&
    server._optimizeDepsMetadata
  ) {
    // if the dependency encountered in the optimized file wa
    // the dependency needs to be resolved starting from the
    // because starting from node_modules/.vite will not find
    // (that is, if it is under node_modules directory in the
    for (const optimizedModule of Object.values(
      server._optimizeDepsMetadata.optimized
    )) {
      if (optimizedModule.file === importerModule.file) {
        importerFile = optimizedModule.src
      }
    }
  }
  const resolved = await this.resolve(url, importerFile)
```

饶了一大圈，终于看到和前文提到的“优化依赖元数据”攀上关系了。下方的this.resolve在前文也有提到，最终是调用到各个rollup插件的resolve方法。而输入的两个参数，url是被依赖的模块地址，importerFile是被依赖模块的引入文件，比如importerFile为 ${root}/vite/src/main.ts文件里面用到了url为vue的模块。

```
export function preAliasPlugin(): Plugin {
  let server: ViteDevServer
  return {
    name: 'vite:pre-alias',
    configureServer(_server) {
      server = _server
    },
    resolveId(id, importer, options) {
      if (!options?.ssr && bareImportRE.test(id)) {
        return tryOptimizedResolve(id, server, importer)
      }
    }
  }
}
```

最终是preAliasPlugin插件处理了这个事，里面又调用了tryOptimizedResolve方法。

tryOptimizedResolve方法内容：

```ts
export function tryOptimizedResolve(
  id: string,
  server: ViteDevServer,
  importer?: string
): string | undefined {
  const depData = server._optimizeDepsMetadata
  if (!depData) return
  const getOptimizedUrl = (optimizedData: typeof depData.optimized[string]) => {
    return (
      optimizedData.file +
      `?v=${depData.browserHash}${
        optimizedData.needsInterop ? `&es-interop` : ``
      }`
    )
  }
  // check if id has been optimized
  const isOptimized = depData.optimized[id]
  if (isOptimized) {
    return getOptimizedUrl(isOptimized)
  }
  if (!importer) return
  // further check if id is imported by nested dependency
  let resolvedSrc: string | undefined
  for (const [pkgPath, optimizedData] of Object.entries(depData.optimized)) {
    // check for scenarios, e.g.
    //   pkgPath  => "my-lib > foo"
    //   id       => "foo"
    // this narrows the need to do a full resolve
    if (!pkgPath.endsWith(id)) continue
    // lazily initialize resolvedSrc
    if (resolvedSrc == null) {
      try {
        // this may throw errors if unable to resolve, e.g. aliased id
        resolvedSrc = normalizePath(resolveFrom(id, path.dirname(importer)))
      } catch {
        // this is best-effort only so swallow errors
        break
      }
    }
    // match by src to correctly identify if id belongs to nested dependency
    if (optimizedData.src === resolvedSrc) {
      return getOptimizedUrl(optimizedData)
    }
  }
}
```

可以看出， tryOptimizedResolve方法里也用到了“优化依赖元数据”，用判断是否处于优化依赖里。

注意最后返回的optimizedData.file是个绝对路径。再回到importAnalysisPlugin插件的normalizeUrl方法里，会把获得的绝对路径掐头去尾，才变成形如/node_modules/.vite/vue.js?v=3bd8dabc这样的路径。

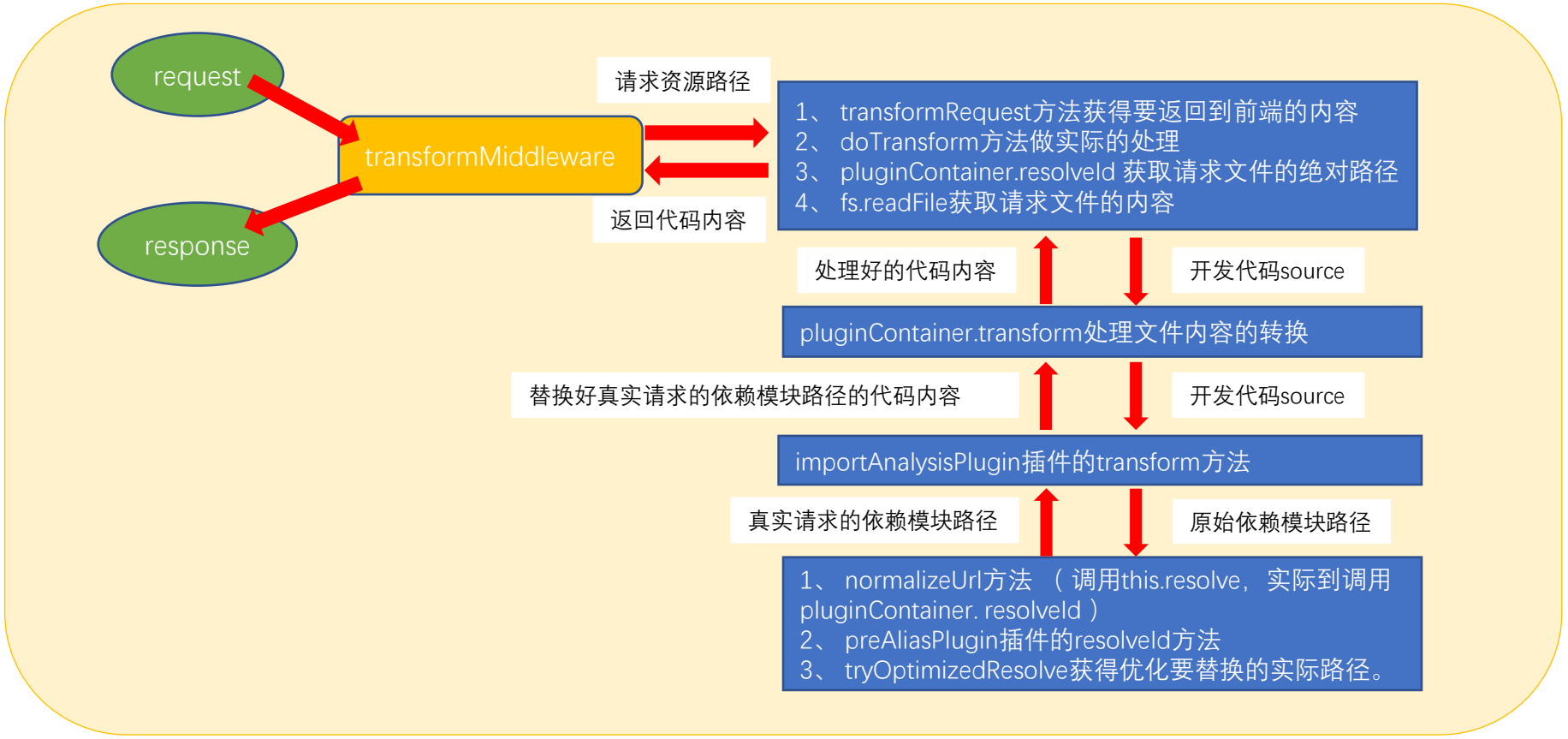
```ts
const resolved = await this.resolve(url, importerFile)
if (!resolved) { ...
}

const isRelative = url.startsWith('.')
const isSelfImport = !isRelative && cleanUrl(url) === cleanUrl(importer)

// normalize all imports into resolved URLs
// e.g. `import 'foo'` -> `import '/@fs/.../node_modules/foo/index.js`
if (resolved.id.startsWith(root + '/')) {
  // in root: infer short absolute path from root
  url = resolved.id.slice(root.length)
} else if (fs.existsSync(cleanUrl(resolved.id))) {
  // exists but out of root: rewrite to absolute /@fs/ paths
  url = path.posix.join(FS_PREFIX + resolved.id)
} else {
  url = resolved.id
}
```

梳理一下全过程：

前置：启动一个服务器，初始化配置，收集所有rollup插件，将其挂载于一个贯通全周期的server对象上。使用transformMiddleware中间件来处理返回内容。 optimizeDeps方法进行预构建，生成“优化依赖元数据”文件放置于./node_modules/.vite/_metadata.json，同时预构建打包后的文件也放置于./node_modules/.vite目录下。
运行时处理过程：



花絮

vite除了预构建把一些零散的js打包成一个js外，其他都是直接加载原文件的，有多少个就请求多少次。项目一大，请求一多，无可避免会造成网络拥塞。本来有个https配置项，可以开启TLS + HTTP/2，利用起HTTP2的多路复用，应该就可以解决请求多的问题。但是里面写明了，当开启了proxy功能的时候，就只剩TLS了，也就是个加密功能。

在createServer创建httpServer的源码里，调用了resolveHttpServer方法创建了服务器：

```
export async function resolveHttpServer(
  { proxy }: CommonServerOptions,
  app: Connect.Server,
  httpsOptions?: HttpsServerOptions
): Promise<HttpServer> {
  /*
   * Some Node.js packages are known to be using this undocumented function,
   * notably "compression" middleware.
   */
  app.prototype._implicitHeader = function _implicitHeader() {
    this.writeHead(this.statusCode)
  }

  if (!httpsOptions) {
    return require('http').createServer(app)
  }

  if (proxy) {
    // #484 fallback to http1 when proxy is needed.
    return require('https').createServer(httpsOptions, app)
  } else {
    return require('http2').createSecureServer(
      {
        ...httpsOptions,
        allowHTTP1: true
      },
      app
    )
  }
}
```

可以看出当启用proxy功能的时候，就用回https模块了。看了注释说的#484issue，代理用的http-proxy模块还不支持HTTP2，所以只能退回使用HTTP1。相信后面会支持或者有别的方法可行。