深入理解Vite核心原理

1 Vite介绍

1.1 Vite是什么?

Vite是新一代的前端构建工具，在尤雨溪开发Vue3.0的时候诞生。类似于Webpack+ Webpack-dev-server。其主要利用浏览器ESM特性导入组织代码，在服务器端按需编译返回，完全跳过了打包这个概念，服务器随起随用。生产中利用Rollup作为打包工具，号称下一代的前端构建工具。

Vite有如下特点:

- 快速的冷启动: No Bundle + esbuild 预构建
- 即时的模块热更新: 基于ESM的HMR，同时利用浏览器缓存策略提升速度
- 真正的按需加载: 利用浏览器ESM支持，实现真正的按需加载

1.2 Vite和传统打包方式的对比

	开发环境	生产构建	Dev 启动速度	Git star
Vite	No Bundle (CJS-ESM)	Rollup	快	36.2k
Snowpack	No Bundle (CJS-ESM)	可选 (Webpack\Rollup\Esbuild)	快	19.6k
Webpack	Bundle (CJS\UMD\ESM)	Webpack	慢 (项目规模递增)	60.3k

1.2.1 VS Webapck

Webpack是近年来使用量最大，同时社区最完善的前端打包构建工具，新出的5.x版本对构建细节进行了优化，在部分场景下打包速度提升明显。Webpack在启动时，会先构建项目模块的依赖图，如果在项目中的某个地方改动了代码，Webpack则会对相关的依赖重新打包，随着项目的增大，其打包速度也会下降。

Vite相比于Webpack而言，没有打包的过程，而是直接启动了一个开发服务器devServer。Vite劫持浏览器的HTTP请求，在后端进行相应的处理将项目中使用的文件通过简单的分解与整合，然后再返回给浏览器(整个过程没有对文件进行打包编译)。所以编译速度很快。

1.2.2 VS SnowPack

Snowpack 首次提出利用浏览器原生ESM能力的打包工具，其理念就是减少或避免整个bundle的打包。默认在 dev 和 production 环境都使用 unbundle 的方式来部署应用。但是它的构建时却是交给用户自己选择，整体的打包体验显得有点支离破碎。

而 Vite 直接整合了 Rollup，为用户提供了完善、开箱即用的解决方案，并且由于这些集成，也方便扩展更多的高级功能。

两者较大的区别是在需要bundle打包的时候Vite 使用 Rollup 内置配置，而 Snowpack 通过其他插件将其委托给 Parcel/``webpack。

2 前置知识

2.1 ESM

在了解Vite之前，需要先了解下ESM

ESM是JavaScript提出的官方标准化模块系统，不同于之前的CJS，AMD，CMD等等，ESM提供了更原生以及更动态的模块加载方案，最重要的就是它是浏览器原生支持的，也就是说我们可以直接在浏览器中去执行import，动态引入我们需要的模块，而不是把所有模块打包在一起。

目前ESM模块化已经支持92%以上的浏览器，而且且作为 ECMA 标准，未来会有更多浏览器支持ECMA规范

当我们在使用模块开发时，其实就是在构建一张模块依赖关系图，当模块加载时，就会从入口文件开始，最终生成完整的模块实例图。

ESM的执行可以分为三个步骤：

- 构建: 确定从哪里下载该模块文件、下载并将所有的文件解析为模块记录
- 实例化: 将模块记录转换为一个模块实例，为所有的模块分配内存空间，依照导出、导入语句把模块指向对应的内存地址。
- 运行：运行代码，将内存空间填充

从上面实例化的过程可以看出，ESM使用实时绑定的模式，导出和导入的模块都指向相同的内存地址，也就是值引用。而CJS采用的是值拷贝，即所有导出值都是拷贝值。

2.2 Esbuild

Vite底层使用Esbuild实现对.``ts、jsx、.``js代码文件的转化，所以先看下什么是es-build。

Esbuild是一个JavaScript`` Bundler 打包和压缩工具，它提供了与Webpack、Rollup等工具相似的资源打包能力。可以将JavaScript 和TypeScript代码打包分发在网页上运行。但其打包速度却是其他工具的10～100倍。

目前他支持以下的功能：

- 加载器
- 压缩
- 打包
- Tree shaking
- Source map生成

esbuild总共提供了四个函数：transform、build、buildSync、Service。有兴趣的可以移步官方文档了解。

2.3 Rollup

在生产环境下，Vite使用Rollup来进行打包

Rollup是基于ESM的JavaScript打包工具。相比于其他打包工具如Webpack，他总是能打出更小、更快的包。因为 Rollup 基于 ESM 模块，比 Webpack 和 Browserify 使用的 CommonJS模块机制更高效。Rollup的亮点在于同一个地方，一次性加载。能针对源码进行 Tree Shaking(去除那些已被定义但没被使用的代码)，以及 Scope Hoisting 以减小输出文件大小提升运行性能。

Rollup分为build（构建）阶段和output generate（输出生成）阶段。主要过程如下：

- 获取入口文件的内容，包装成module，生成抽象语法树
- 对入口文件抽象语法树进行依赖解析
- 生成最终代码
- 写入目标文件

如果你的项目（特别是类库）只有JavaScript，而没有其他的静态资源文件，使用Webpack就有点大才小用了。因为Webpack 打包的文件的体积略大，运行略慢，可读性略低。这时候Rollup也不失为一个好选择。

这里想对Rollp进行更深入的学习可以看看官网的介绍。

3 核心原理

详细阐述下：

1. 当声明一个 script标签类型为 module 时,如

```
<script type="module" src="/src/main.js"></script>
```

1. 当浏览器解析资源时，会往当前域名发起一个GET请求main.js文件

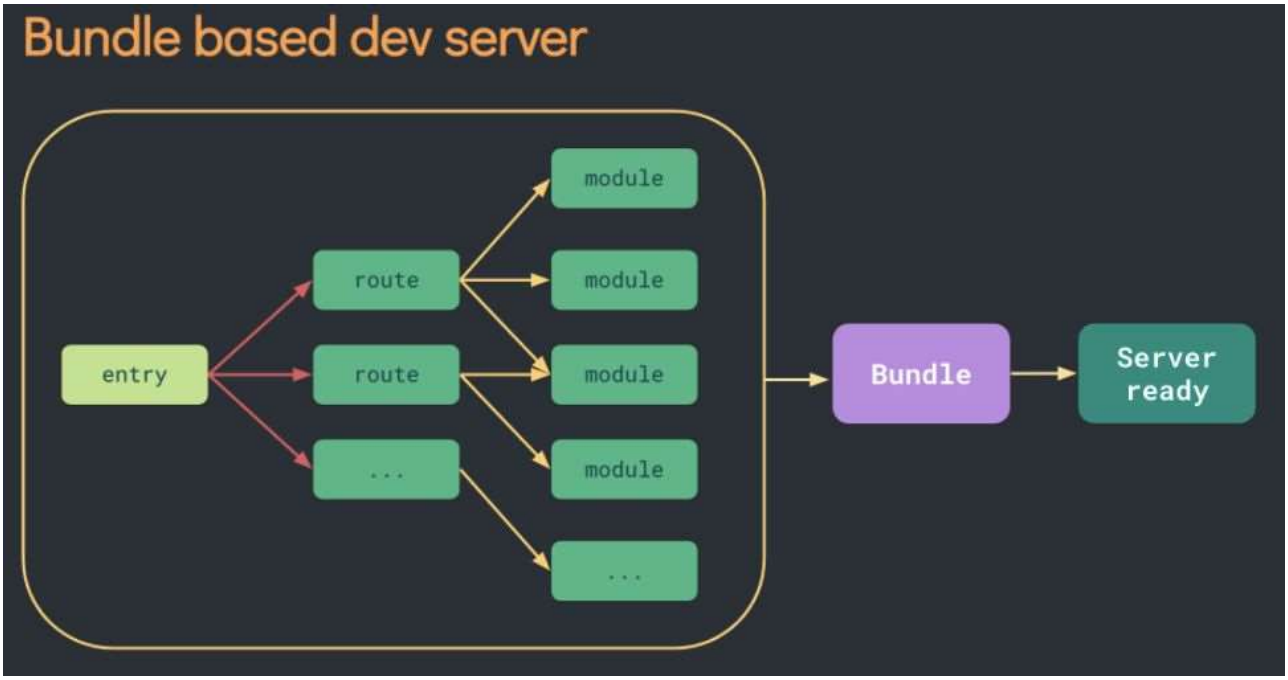
```
// main.js
import { createApp } from 'vue'
import App from './App.vue'
createApp(App).mount('#app')
```

1. 请求到了main.js文件，会检测到内部含有import引入的包，又会import 引用发起HTTP请求获取模块的内容文件，如App.vue、vue文件

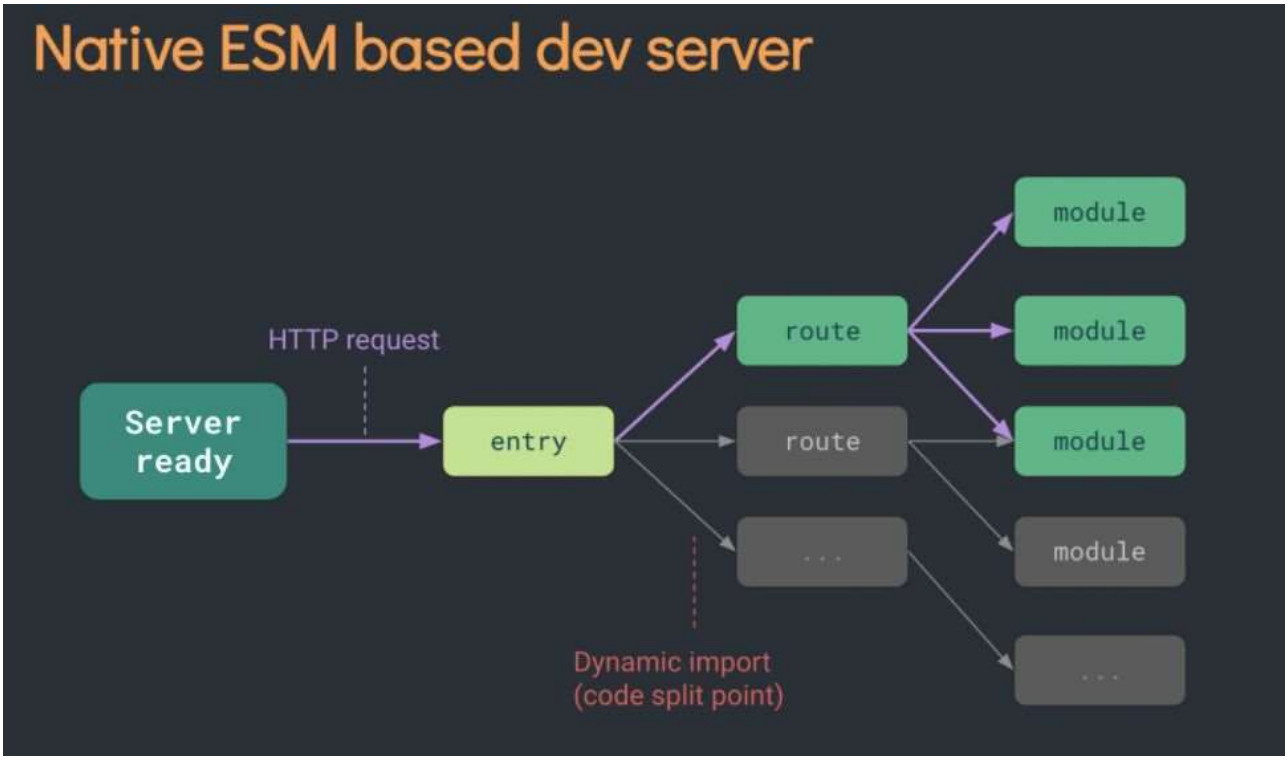
Vite其核心原理是利用浏览器现在已经支持ES6的import,碰见import就会发送一个HTTP请求去加载文件，Vite启动一个 koa 服务器拦截这些请求，并在后端进行相应的处理将项目中使用的文件通过简单的分解与整合，然后再以ESM格式返回返回给浏览器。Vite整个过程中没有对文件进行打包编译，做到了真正的按需加载，所以其运行速度比原始的webpack开发编译速度快出许多！

3.1 基于ESM的Dev server

在Vite出来之前，传统的打包工具如Webpack是先解析依赖、打包构建再启动开发服务器，Dev Server 必须等待所有模块构建完成，当我们修改了 bundle模块中的一个子模块， 整个 bundle 文件都会重新打包然后输出。项目应用越大，启动时间越长。



而Vite利用浏览器对ESM的支持，当 import 模块时，浏览器就会下载被导入的模块。先启动开发服务器，当代码执行到模块加载时再请求对应模块的文件,本质上实现了动态加载。灰色部分是暂时没有用到的路由，所有这部分不会参与构建过程。随着项目里的应用越来越多，增加route，也不会影响其构建速度。



3.2 基于ESM 的 HMR 热更新

目前所有的打包工具实现热更新的思路都大同小异：主要是通过WebSocket创建浏览器和服务器的通信监听文件的改变，当文件被修改时，服务端发送消息通知客户端修改相应的代

码，客户端对应不同的文件进行不同的操作的更新。

3.2.1 VS Webpack

Webpack: 重新编译，请求变更后模块的代码，客户端重新加载

Vite: 请求变更的模块，再重新加载

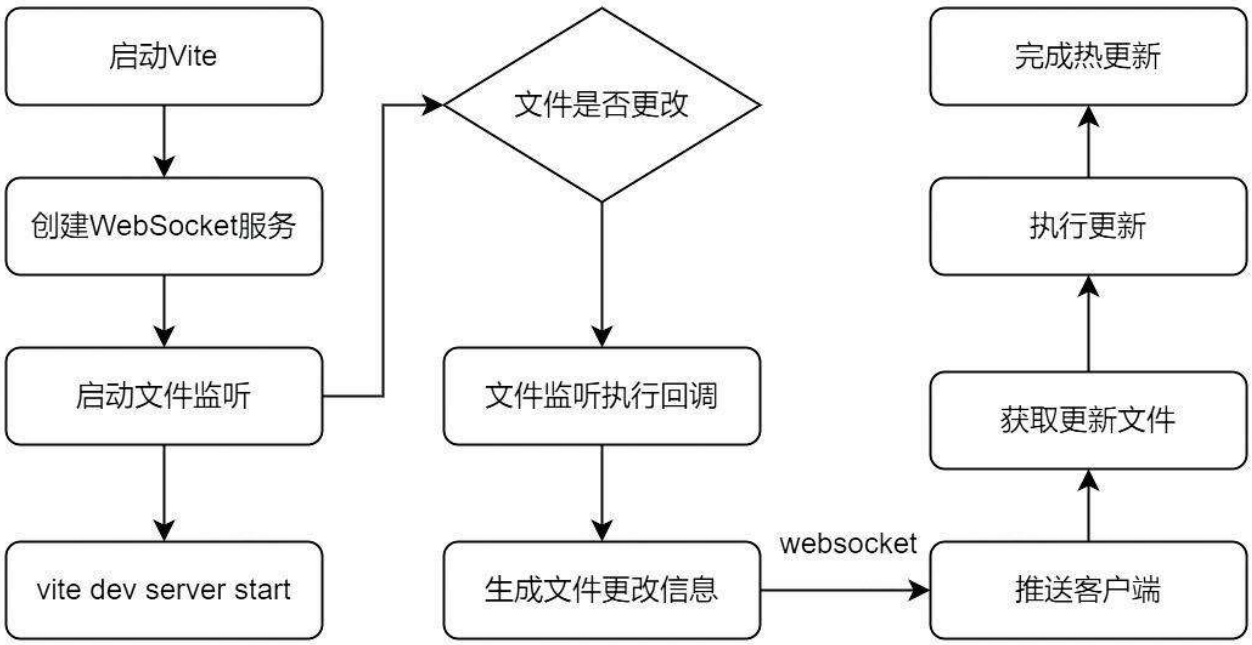
Vite 通过 chokidar 来监听文件系统的变更，只用对发生变更的模块重新加载， 只需要精确的使相关模块与其临近的 HMR边界连接失效即可，这样HMR 更新速度就不会因为应用体积的增加而变慢而 Webpack 还要经历一次打包构建。所以 HMR 场景下，Vite 表现也要好于 Webpack。

3.2.2 核心流程

Vite整个热更新过程可以分成四步

1. 创建一个websocket服务端和client文件，启动服务
2. 通过chokidar监听文件变更
3. 当代码变更后，服务端进行判断并推送到客户端
4. 客户端根据推送的信息执行不同操作的更新

整体流程图：



3.2.2.1 启动热更新：createWebSocketServer

在 Vite`` dev server 启动之前，Vite 会为 HMR 做一些准备工作：比如创建websocket服务，利用chokidar创建一个监听对象 watcher 用于对文件修改进行监听等等，具体核心代码：

源码位置：packages/vite/src/node/server/index.ts

```
export async function createServer(
  inlineConfig: InlineConfig = {}
): Promise<ViteDevServer> {
  ....
  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
  ....
  watcher.on('change', async (file) => {

  })
  watcher.on('add', (file) => {
  })
  watcher.on('unlink', (file) => {
  })
  ...
  return server
}
```

createWebSocketServer这个方法主是创建WebSocket服务并对错误进行一些处理，最后返回封装好的on、off、 send 和 close 方法，用于后续服务端推送消息和关闭服务。

源码位置：packages/vite/src/node/server/ws.ts

```
export function createWebSocketServer(
  server: Server | null,
  config: ResolvedConfig,
  httpsOptions?: HttpsServerOptions
): WebSocketServer {
  let wss: WebSocket
  let httpsServer: Server | undefined = undefined
  // 热更新配置
  const hmr = isObject(config.server.hmr) && config.server.hmr
  const wsServer = (hmr && hmr.server) || server
  // 普通模式
  if (wsServer) {
    wss = new WebSocket({ noServer: true })
    wsServer.on('upgrade', (req, socket, head) => {
      // 监听通过vite客户端发送的websocket消息，通过HMR_HEADER区分
      if (req.headers['sec-websocket-protocol'] === HMR_HEADER) {
        wss.handleUpgrade(req, socket as Socket, head, (ws) => {
          wss.emit('connection', ws, req)
        })
      }
    })
  } else { // 中间件模式
    // vite dev server in middleware mode
    wss = new WebSocket(websocketServerOptions)
  }
  wss.on('connection', (socket) => {
    ...
  })
  // 错误处理
  wss.on('error', (e: Error & { code: string }) => {
    ...
  })
  // 返回
```

```
  return {
    on: wss.on.bind(wss),
    off: wss.off.bind(wss),
    send(payload: HMRPayload) {
      ...
    },
    close() {
      ...
    }
  }
}
```

3.2.2.2 执行热更新：moduleGraph+handleHMRUpdate模块

接收到文件改动执行的回调，这里主要两个操作：`moduleGraph.onFileChange`修改文件的缓存和`handleHMRUpdate`执行热更新

源码位置：packages/vite/src/node/server/index.ts

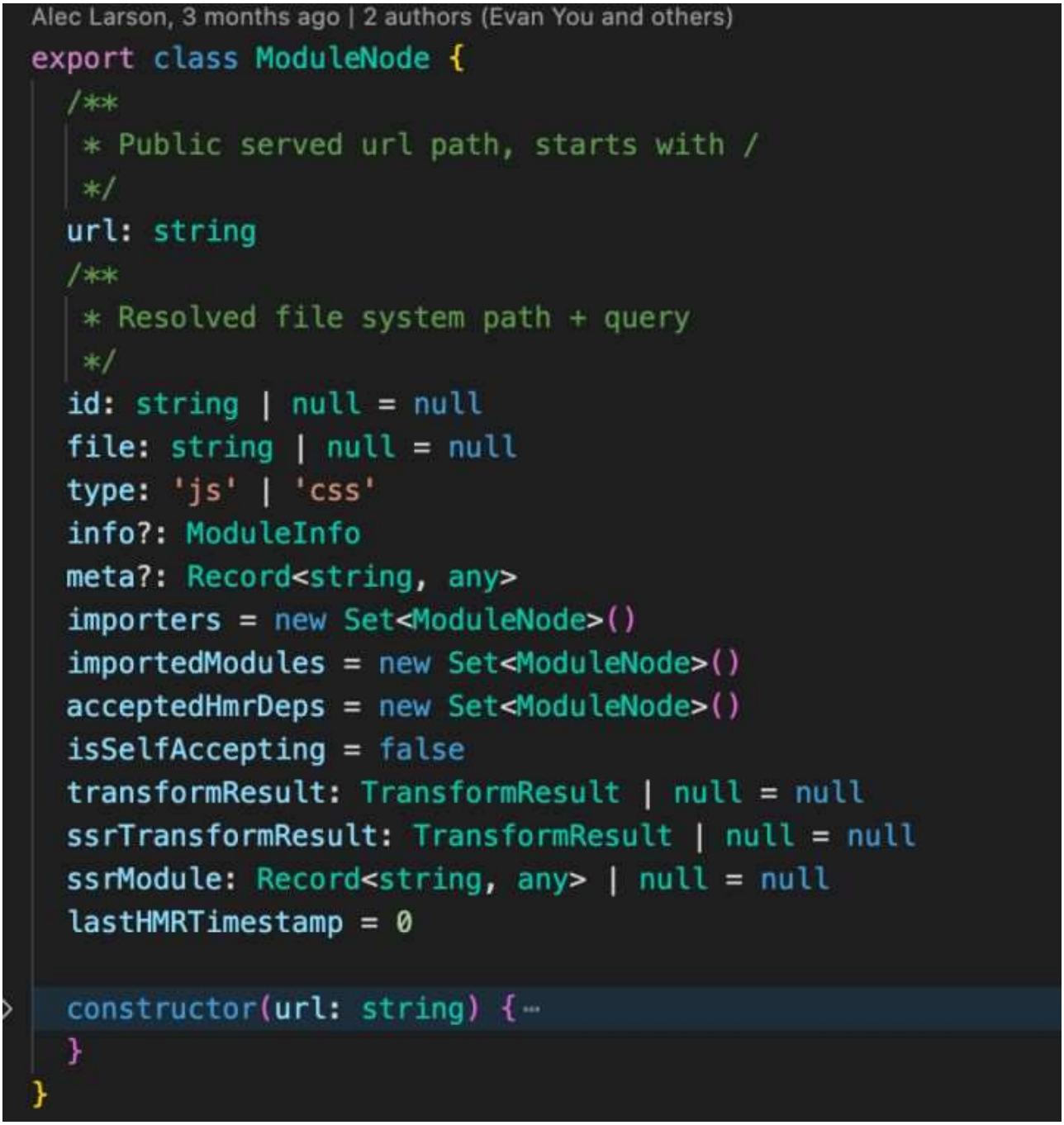
```
watcher.on('change', async (file) => {
    file = normalizePath(file)
    if (file.endsWith('/package.json')) {
      return invalidatePackageData(packageCache, file)
    }
    // invalidate module graph cache on file change
    moduleGraph.onFileChange(file)
    if (serverConfig.hmr !== false) {
      try {
        await handleHMRUpdate(file, server)
      } catch (err) {
        ws.send({
          type: 'error',
          err: prepareError(err)
        })
      }
    }
  })
```

3.2.2.2.1 moduleGraph

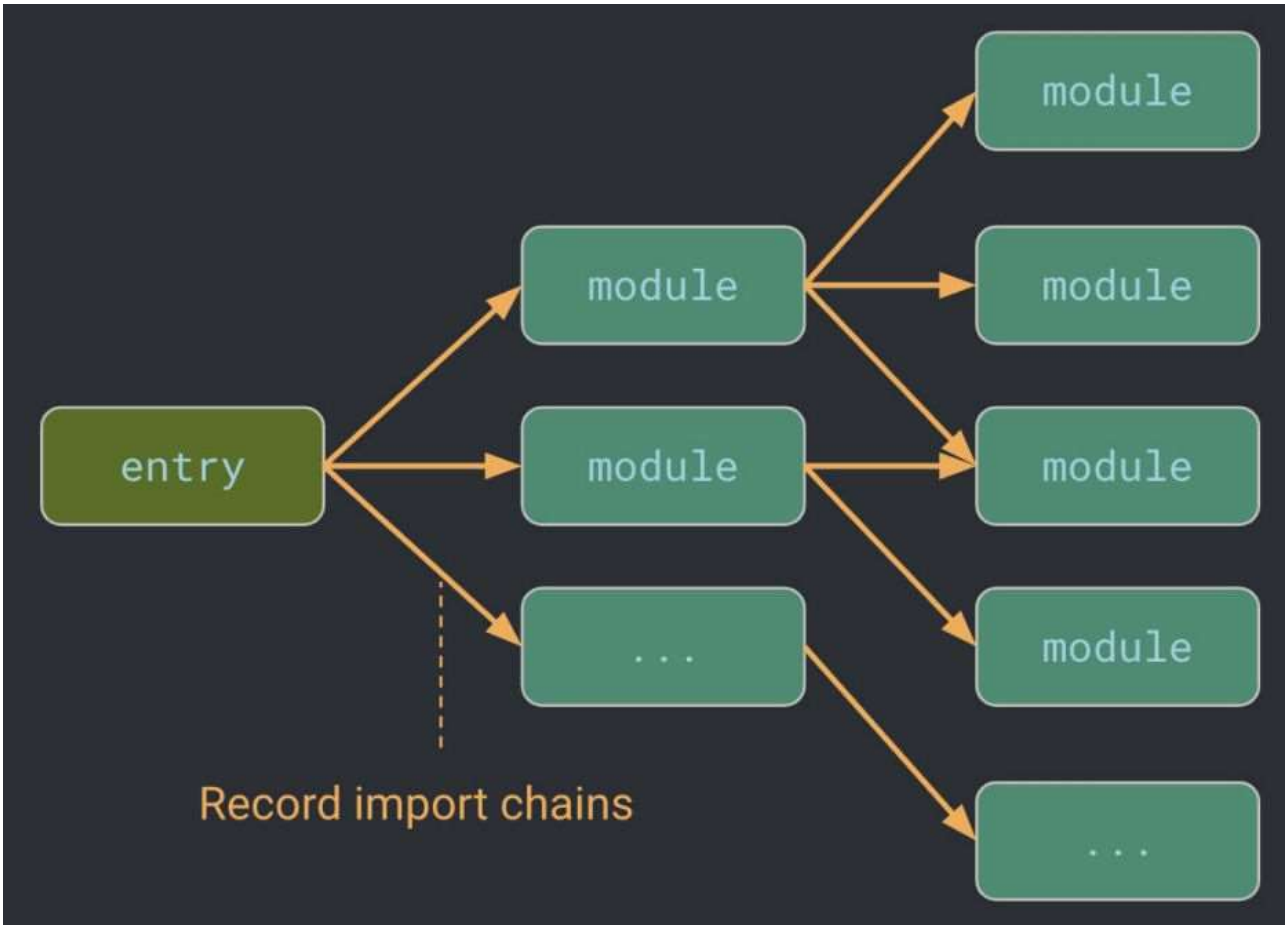
`moduleGraph` 是`Vite`定义的用来记录整个应用的模块依赖图的类，除此之外还有`moduleNode`。

源码位置：packages/vite/src/node/server/moduleGraph.ts

```
Francois Valdy, a month ago | 6 authors (Evan You and others)
export class ModuleGraph {      Evan You, a year ago • wip: refactor into module gr
  urlToModuleMap = new Map<string, ModuleNode>()
  idToModuleMap = new Map<string, ModuleNode>()
  // a single file may corresponds to multiple modules with different queries
  fileToModulesMap = new Map<string, Set<ModuleNode>>()
  safeModulesPath = new Set<string>()

  constructor(…
  ) {}

  async getModuleByUrl(…
  }

  getModuleById(id: string): ModuleNode | undefined {…
  }

  getModulesByFile(file: string): Set<ModuleNode> | undefined {…
  }

  onFileChange(file: string): void {…
  }

  invalidateModule(mod: ModuleNode, seen: Set<ModuleNode> = new Set()): void {…
  }

  invalidateAll(): void {…
  }

  /**…
  async updateModuleInfo(…
  }

  async ensureEntryFromUrl(rawUrl: string, ssr?: boolean): Promise<ModuleNode> {…
  }

  // some deps, like a css file referenced via @import, don't have its own…
  createFileOnlyEntry(file: string): ModuleNode {…
  }

  // for incoming urls, it is important to:…
  async resolveUrl(url: string, ssr?: boolean): Promise<ResolvedUrl> {…
  }
}
```



moduleGraph是由一系列 map 组成，而这些map分别是url、id、file等与ModuleNode的映射，而 ModuleNode 是 Vite中定义的最小模块单位。通过这两个类可以构建下面的模块依赖图：



可以看看moduleGraph.``onFileChange这个函数：主要是用来清空被修改文件对应的ModuleNode对象的 transformResult 属性，使之前的模块已有的转换缓存失效。这块也就是Vite在热更新里的缓存机制。可以看看官网的介绍。

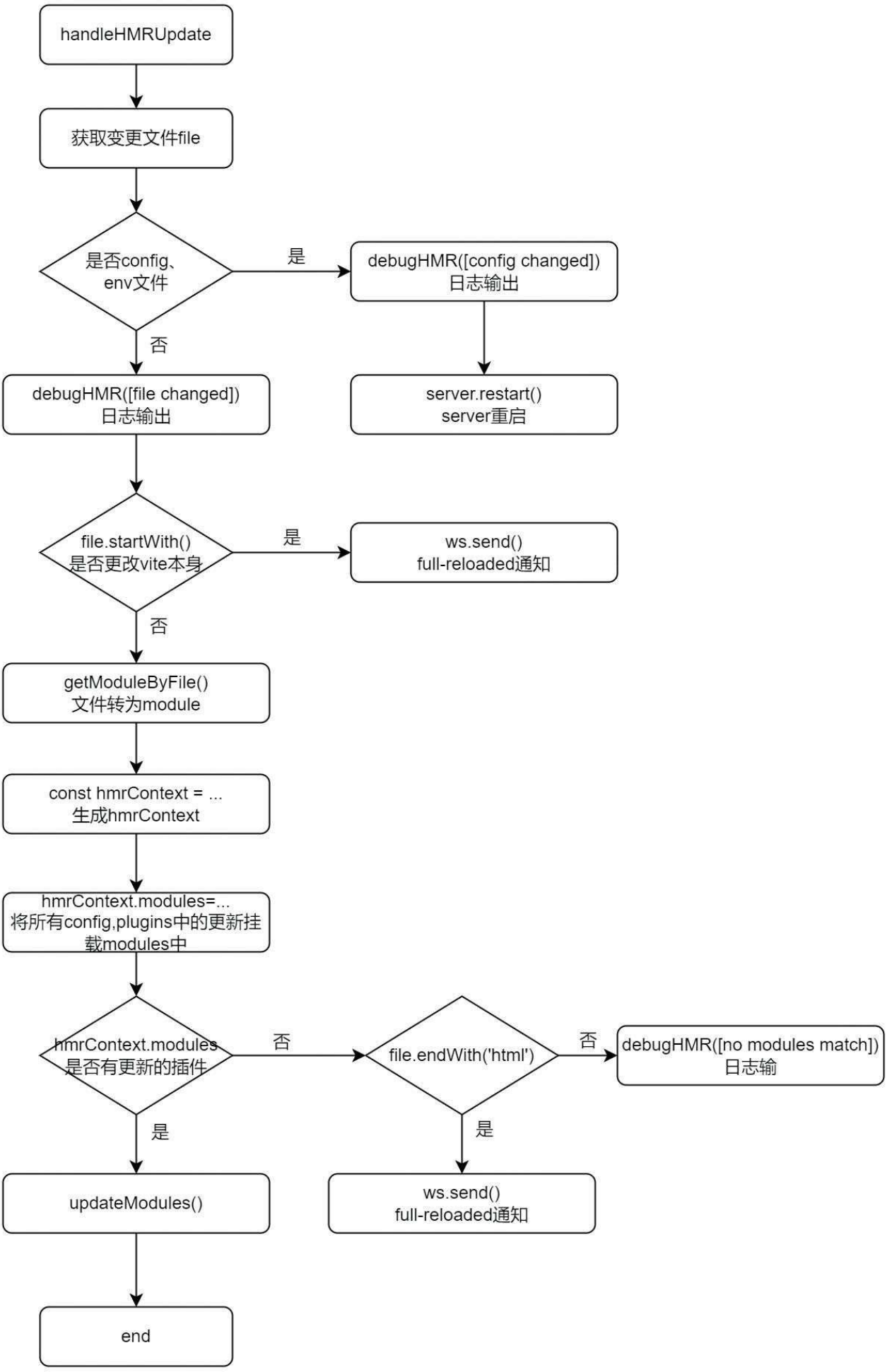
源码位置：packages/vite/src/node/server/moduleGraph.ts

```
onFileChange(file: string): void {
    const mods = this.getModulesByFile(file)
    if (mods) {
      const seen = new Set<ModuleNode>()
      mods.forEach((mod) => {
        this.invalidateModule(mod, seen)
      })
    }
  }

  invalidateModule(mod: ModuleNode, seen: Set<ModuleNode> = new Set()): void {
    mod.info = undefined
    mod.transformResult = null
    mod.ssrTransformResult = null
    invalidateSSRModule(mod, seen)
  }
```

3.2.2.2.2 handleHMRUpdate

handleHMRUpdate 模块主要是监听文件的更改，进行处理和判断通过WebSocket给客户端发送消息通知客户端去请求新的模块代码。

源码位置：packages/vite/packages/vite/src/node/server/hmr.ts



3.2.2.3 客户端：websocket通信和更新处理

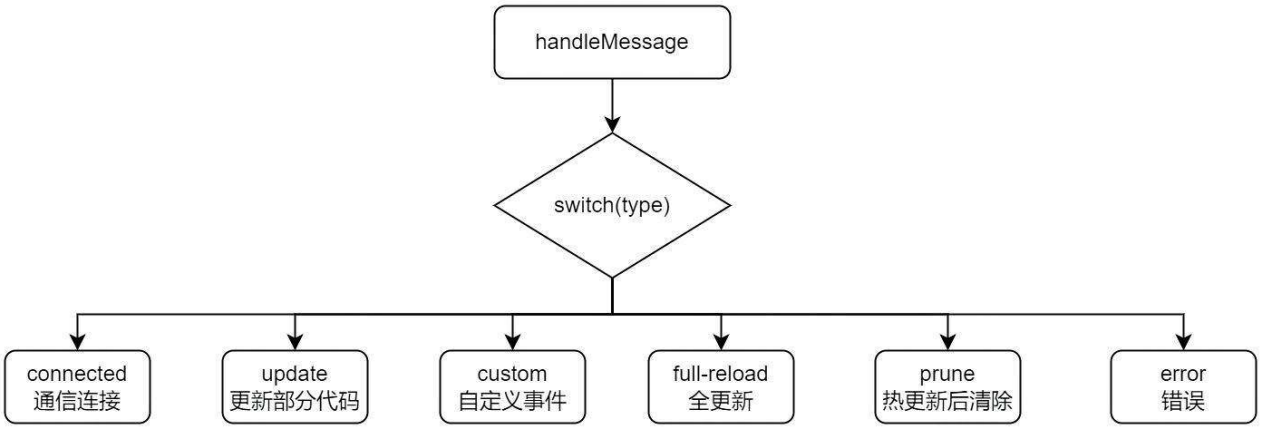
客户端：当我们配置了热更新且不是ssr的时候，Vite底层在处理html的时候会把HMR相关的客户端代码写入在我们的代码中，如下：

```
if (hasHMR && !ssr) {
  debugHmr(      Evan You, a year ago • wip: hmr propagation
    `${
      isSelfAccepting
        ? `[self-accepts]`
        : acceptedUrls.size
        ? `[accepts-deps]`
        : `[detected api usage]`
    } ${prettyImporter}`
  )
  // inject hot context
  str().prepend(
    `import { createHotContext as __vite__createHotContext } from "${clientPublicPath}";` +
      `import.meta.hot = __vite__createHotContext(${JSON.stringify(
        importerModule.url
      )});`
  )
}
```

当接收到服务端推送的消息，通过不同的消息类型做相应的处理，如(connected、update、custom...)，在实际开发热更新中使用最频繁的是update(动态加载热更新模块)和full-reload(刷新整个页面)事件。

源码位置：packages/vite/packages/vite/src/client/client.ts

核心代码实现



3.2.2.4 优化：浏览器的缓存策略提高响应速度

同时，Vite 还利用HTTP加速整个页面的重新加载。设置响应头使得依赖模块(dependency module)进行强缓存，而源码文件通过设置 304 Not Modified 而变成可依据条件而进行更新。

若需要对依赖代码模块做改动可手动操作使缓存失效:

```
vite --force
```

或者手动删除 node_modules/.``vite 中的缓存文件。

3.3 基于esbuild的依赖预编译优化

3.3.1 为什么需要预构建?

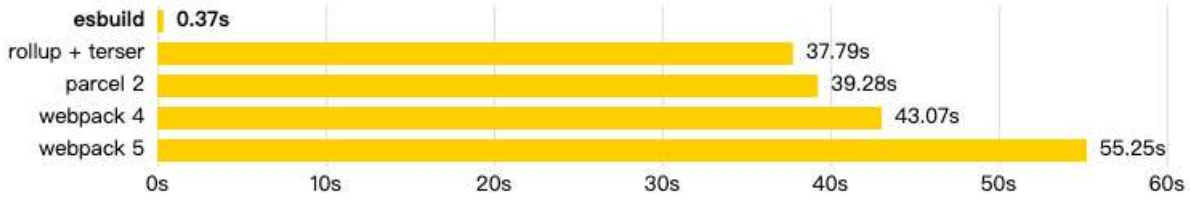
1. 支持commonJS依赖
2. 上面提到Vite是基于浏览器原生支持ESM的能力实现的，但要求用户的代码模块必须是ESM模块，因此必须将commonJs的文件提前处理，转化成 ESM 模块并缓存入 node_modules/.vite
3. 减少模块和请求数量

除此之外，我们常用的lodash工具库，里面有很多包通过单独的文件相互导入，而 lodash-es 这种包会有几百个子模块，当代码中出现 import { debounce } from 'lodash-es' 会发出几百个 HTTP 请求，这些请求会造成网络堵塞，影响页面的加载。

Vite 将有许多内部模块的 ESM 依赖关系转换为单个模块，以提高后续页面加载性能。

通过预构建 lodash-es 成为一个模块，也就只需要一个 HTTP 请求了!

3.3.2 为什么使用Esbuild?



引用尤大的一句话: “快”就一个字

这是Esbuild首页的图。新一代的打包工具，提供了与Webpack、Rollup、Parcel 等工具相似的资源打包能力，但在时速上达到10～100倍的差距，耗时是Webpack2%~3%

1. 编译运行 VS 解释运行

- 大多数前端打包工具都是基于 JavaScript 实现的，大家都知道JavaScript是解释型语言，边运行边解释。而 Esbuild 则选择使用 Go 语言编写，该语言可以编译为原生代码,在编译的时候都将语言转为机器语言，在启动的时候直接执行即可，在 CPU 密集场景下，Go 更具性能优势。

1. 多线程 VS 单线程
2. JavaScript 本质上是一门单线程语言，直到引入 WebWorker 之后才有可能在浏览器、Node 中实现多线程操作。就我对Webpack的源码理解，其源码也并未使用 WebWorker 提供的多线程能力。而GO天生的多线程优势。
3. 对构建流程进行了优化，充分利用 CPU 资源

3.3.3 实现原理?

Vite预编译之后，将文件缓存在node_modules/.vite/文件夹下。根据以下地方来决定是否需要重新执行预构建。

- package.json中：dependencies发生变化
- 包管理器的lockfile

如果想强制让Vite重新预构建依赖，可以使用--force启动开发服务器，或者直接删掉node_modules/.vite/文件夹。

3.3.3.1 核心代码实现

1. 通过createServer创建server对象后，当服务器启动会执行httpServer.listen方法
2. 在执行createServer时，Vite底层会重写server.listen方法:首先调用插件的buildStart再执行runOptimize()方法
3. runOptimize()调用optimizeDeps()和createMissingImporterRegisterFn()方法

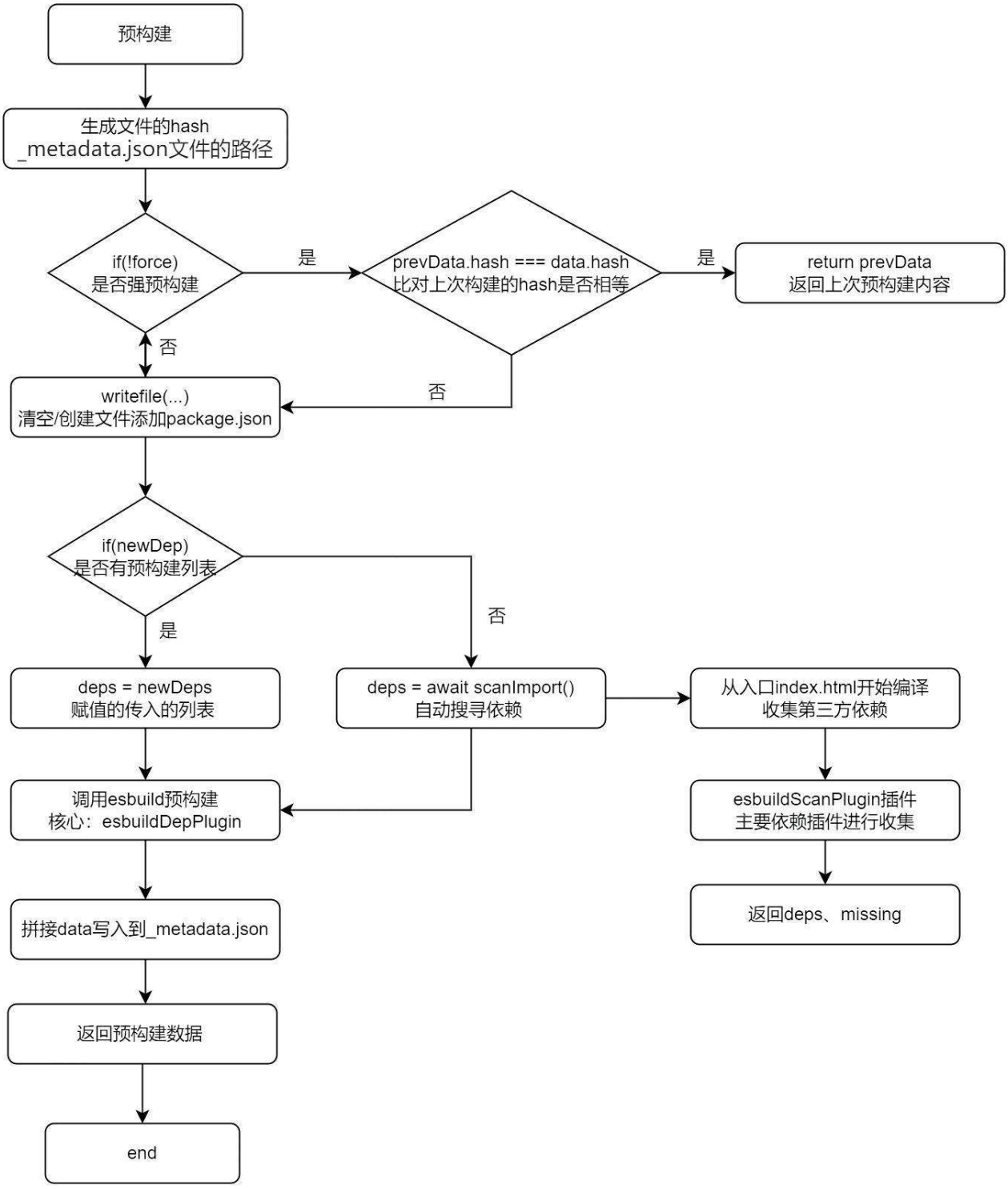
```
const runOptimize = async () => {
    if (config.cacheDir) {
      server._isRunningOptimizer = true
      try {
        server._optimizeDepsMetadata = await optimizeDeps(
          config,
          config.server.force || server._forceOptimizeOnRestart
        )
      } finally {
        server._isRunningOptimizer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

1. optimizeDeps()主要是根据配置文件生成hash，获取上次预购建的内容(存放在_metadata.json文件)。如果不是强预构建就对比_metadata.json文件的hash和新生成的hash：一致就返回_metadata.json文件的内容，否则清空缓存文件调用Esbuild构建模块再次存入_metadata.json文件

3.3.3.2 整体的流程图

核心代码都在packages/``vite``/``src``/``node``/optimizer/index.ts里面

- 自动搜寻依赖主要模块：esbuildScanPlugin
- 预构建编译主要模块：esbuildDepPlugin



3.4 基于Rollup的 Plugins

Vite 从 preact 的 WMR 中得到了启发，将Vite Plugins继承Rollup`` Plugins `` ``API，在其基础上进行了一些扩展(如Vite特有的钩子等)，同时Vite也基于Rollup plugins机制提供了强大的插件API。目前和 Vite 兼容或者内置的插件，可以查看vite-rollup-plugins

3.4.1 Vite插件是什么

使用Vite插件可以扩展Vite能力，通过暴露一些构建打包过程的一些时机配合工具函数，让用户可以自定义地写一些配置代码，执行在打包过程中。比如解析用户自定义的文件输入，在打包代码前转译代码，或者查找。

在实际的实现中，Vite 仅仅需要基于Rollup设计的接口进行扩展，在保证兼容 Rollup插件的同时再加入一些Vite特有的钩子和属性来进行扩展。

3.4.2 Vite独有钩子

对于各个钩子的具体使用可以移步这里

- config：可以在Vite被解析之前修改Vite的相关配置。钩子接收原始用户配置config和一个描述配置环境的变量env
- configResolved：解析Vite配置后调用，配置确认
- configureserver：主要用来配置开发服务器，为dev-server添加自定义的中间件
- transformindexhtml：主要用来转换index.html，钩子接收当前的 HTML 字符串和转换上下文
- handlehotupdate：执行自定义HMR更新，可以通过ws往客户端发送自定义的事件

3.4.3 通用钩子

这里举一些常用的通用钩子，其余的通用钩子可以移步这里

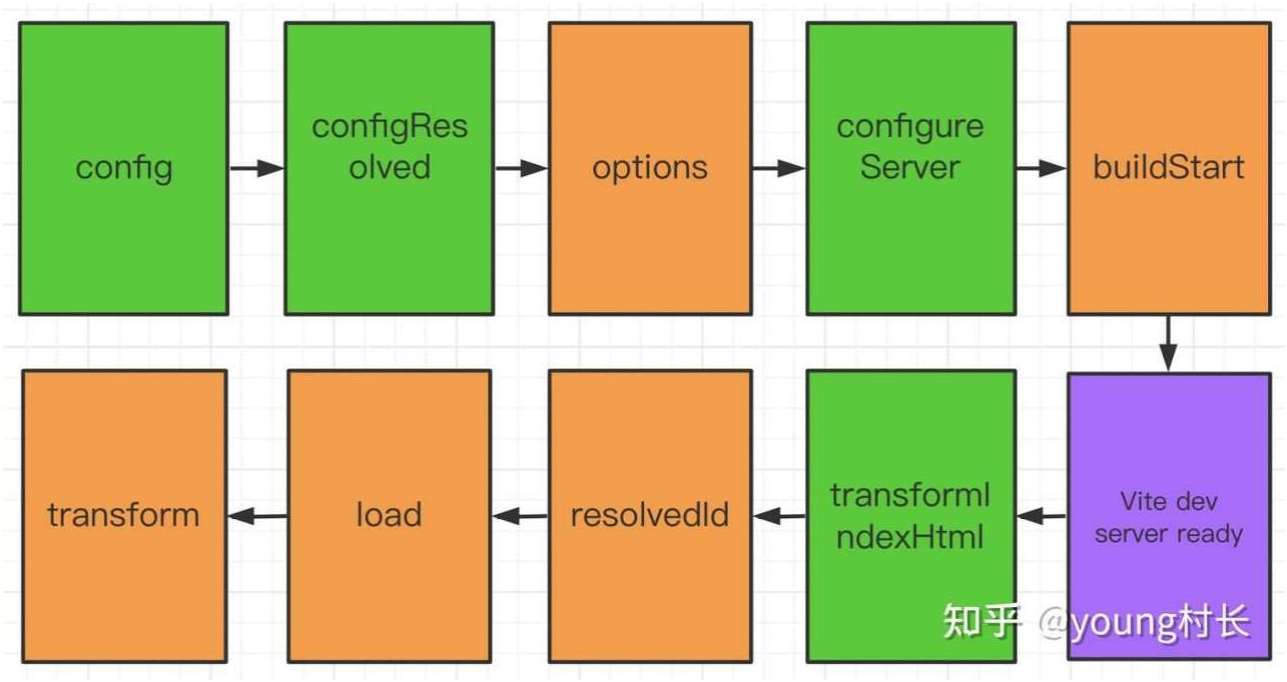
- 服务启动时调用一次
    - options：获取、操纵Rollup选项
    - buildstart：开始创建

- 在每个传入模块请求时被调用
    - resolveId：创建自定义确认函数，可以用来定位第三方依赖
    - load：可以自定义加载器，可用来返回自定义的内容
    - transform：在每个传入模块请求时被调用，主要是用来转换单个模块

- 服务关闭时调用一次
    - buildend：在服务器关闭时被调用
    - closeBundle

3.4.4 钩子的调用顺序

引用@young村长的一个图:



Vite的官网可以看出：Vite插件可以用一个 enforce 属性（类似于 Webpack加载器）来调整它的应用顺序。enforce 的值可以是pre 或 post。解析后的插件将按照以下顺序排列：

- Alias
- enforce:'pre'的自定义插件
- Vite核心插件
- 没有enforce的自定义插件
- Vite构建用的插件
- enforce:'post'的自定义插件
- Vite后置构建插件

3.4.5 自定义插件

- 编写插件代码

```
export default function myVitePlugin () {
  // 定义vite插件唯一id
  const virtualFileId = '@my-vite-plugin'
  // 返回的整个插件对象
  return {
    // 必须的，将会显示在 warning 和 error 中
    name: 'vite-plugin',
    // 钩子
    // config
    config: (config, env) => ({
       console.log('config',config)
       return {}
    }),
    // 确认config
    configResolved: config => ({}),
    options: options => ({}),
    buildStart: options => ({}),
    transformIndexHtml: (html, ctx) => ({
        return html
```

```
    }),
    //确认
    resolveId: (source, importer) => ({}),
    // 转换
    transform: (code, id) => ({})
  }
}
```

- 引入插件：vite.config.js/ts 中引用

```
// vite.config.js/ts
import myVitePlugin from '...'
export default defineConfig{
    plugins:[vue(),myVitePlugin()]
}
```

4 总结

最后总结下Vite相关的优缺点：

- 优点：
    - 快速的冷启动：采用No Bundle和esbuild预构建，速度远快于Webpack
    - 高效的热更新：基于ESM实现，同时利用HTTP头来加速整个页面的重新加载，增加缓存策略
    - 真正的按需加载：基于浏览器ESM的支持，实现真正的按需加载

- 缺点
    - 生态：目前Vite的生态不如Webapck，不过我觉得生态也只是时间上的问题。
    - 生产环境由于esbuild对css和代码分割不友好使用Rollup进行打包

Vite.js虽然才在构建打包场景兴起，但在很多场景下基本都会优于现有的解决方案。如果有生态、想要丰富的loader、plugins的要求可以考虑成熟的Webpack。在其余情况下，Vite.js不失为一个打包构建工具的好选择。