依赖预构建：Esbuild 打包功能如何被 Vite 玩出花来？

发布于 2022-05-09

在第七节的内容中，我们已经分析过 依赖预构建 的意义以及使用，对于其底层的实现并没有作过多的介绍。而在 Vite 依赖预构建的底层实现中，大量地使用到了 Esbuild 这款构建工具，实现了比较复杂的 Esbuild 插件，同时也应用了诸多 Esbuild 使用技巧。相信在理解这部分的源码之后，你将会对 Vite 预构建以及 Esbuild 本身有更加深入的认识。

接下来，我就来带你揭开 Vite 预构建 神秘的面纱，从核心流程到依赖扫描、依赖打包的具体实现，带你彻底理解预构建背后的技术，学习 Vite 是如何灵活运用 Esbuild，将 Esbuild 这个打包工具 玩出花来 的。

预构建核心流程

关于预构建所有的实现代码都在 `optimizeDeps` 函数当中，也就是在仓库源码的 packages/vite/src/node/optimizer/index.ts 文件中，你可以对照着来学习。

缓存判断

首先是预构建缓存的判断。Vite 在每次预构建之后都将一些关键信息写入到了 `_metadata.json` 文件中，第二次启动项目时会通过这个文件中的 hash 值来进行缓存的判断，如果命中缓存则不会进行后续的预构建流程，代码如下所示:

```
// _metadata.json 文件所在的路径
const dataPath = path.join(cacheDir, "_metadata.json");
// 根据当前的配置计算出哈希值
const mainHash = getDepHash(root, config);
const data: DepOptimizationMetadata = {
  hash: mainHash,
  browserHash: mainHash,
  optimized: {},
};
```

```
// 默认走到里面的逻辑

if (!force) {
  let prevData: DepOptimizationMetadata | undefined;
  try {
    // 读取元数据
    prevData = JSON.parse(fs.readFileSync(dataPath, "utf-8"));
  } catch (e) {}
  // 当前计算出的哈希值与 _metadata.json 中记录的哈希值一致，表示命中缓存，不用预构建
  if (prevData && prevData.hash === data.hash) {
    log("Hash is consistent. Skipping. Use --force to override.");
    return prevData;
  }
}
```

值得注意的是哈希计算的策略，即决定哪些配置和文件有可能影响预构建的结果，然后根据这些信息来生成哈希值。这部分逻辑集中在 `getHash` 函数中，我把关键信息放到了注释中:

```
const lockfileFormats = ["package-lock.json", "yarn.lock", "pnpm-lock.yaml"];
function getDepHash(root: string, config: ResolvedConfig): string {
  // 获取 lock 文件内容
  let content = lookupFile(root, lockfileFormats) || "";
  // 除了 lock 文件外，还需要考虑下面的一些配置信息
  content += JSON.stringify(
    {
      // 开发/生产环境
      mode: config.mode,
      // 项目根路径
      root: config.root,
      // 路径解析配置
      resolve: config.resolve,
      // 自定义资源类型
      assetsInclude: config.assetsInclude,
      // 插件
      plugins: config.plugins.map((p) => p.name),
      // 预构建配置
      optimizeDeps: {
        include: config.optimizeDeps?.include,
        exclude: config.optimizeDeps?.exclude,
      },
    },
    // 特殊处理函数和正则类型
    (_, value) => {
      if (typeof value === "function" || value instanceof RegExp) {
        return value.toString();
      }
      return value;
    }
  );
  // 最后调用 crypto 库中的 createHash 方法生成哈希
  return createHash("sha256").update(content).digest("hex").substring(0, 8);
}
```

依赖扫描

如果没有命中缓存，则会正式地进入依赖预构建阶段。不过 Vite 不会直接进行依赖的预构建，而是在之前探测一下项目中存在哪些依赖，收集依赖列表，也就是进行 依赖扫描 的过程。这个过程是必须的，因为 Esbuild 需要知道我们到底要打包哪些第三方依赖。关键代码如下:

```
({ deps, missing } = await scanImports(config));
```

在 `scanImports` 方法内部主要会调用 Esbuild 提供的 `build` 方法:

```
const deps: Record<string, string> = {};
// 扫描用到的 Esbuild 插件
const plugin = esbuildScanPlugin(config, container, deps, missing, entries);
await Promise.all(
  // 应用项目入口
  entries.map((entry) =>
    build({
      absWorkingDir: process.cwd(),
      // 注意这个参数
      write: false,
      entryPoints: [entry],
      bundle: true,
      format: "esm",
      logLevel: "error",
      plugins: [...plugins, plugin],
      ...esbuildOptions,
    })
  )
);
```

值得注意的是，其中传入的 `write` 参数被设为 false，表示产物不用写入磁盘，这就大大节省了磁盘 I/O 的时间了，也是 依赖扫描 为什么往往比 依赖打包 快很多的原因之一。

接下来会输出预打包信息:

```
if (!asCommand) {
  if (!newDeps) {
    logger.info(
      chalk.greenBright(`Pre-bundling dependencies:\n  ${depsString}`)
    );
    logger.info(
      `(this will be run only when your dependencies or config have changed)`
    );
  }
} else {
```

```
  logger.info(chalk.greenBright(`Optimizing dependencies:\n  ${depsString}`));
}
```

这时候你可以明白，为什么第一次启动时会输出预构建相关的 log 信息了，其实这些信息都是通过 依赖扫描 阶段来搜集的，而此时还并未开始真正的依赖打包过程。

可能你会有疑问，为什么对项目入口打包一次就收集到所有依赖信息了呢？大家可以注意到 esbuildScanPlugin 这个函数创建 scan 插件 的时候就接收到了 deps 对象作为入参，这个对象的作用不可小觑，在 scan 插件 里面就是解析各种 import 语句，最终通过它来记录依赖信息。由于解析的过程比较复杂，我们放到下一个部分具体讲解，这里你只需要知道核心的流程即可。

依赖打包

收集完依赖之后，就正式地进入到 依赖打包 的阶段了。这里也调用 Esbuild 进行打包并写入产物到磁盘中，关键代码如下:

```
const result = await build({
  absWorkingDir: process.cwd(),
  // 所有依赖的 id 数组，在插件中会转换为真实的路径
  entryPoints: Object.keys(flatIdDeps),
  bundle: true,
  format: "esm",
  target: config.build.target || undefined,
  external: config.optimizeDeps?.exclude,
  logLevel: "error",
  splitting: true,
  sourcemap: true,
  outdir: cacheDir,
  ignoreAnnotations: true,
  metafile: true,
  define,
  plugins: [
    ...plugins,
    // 预构建专用的插件
    esbuildDepPlugin(flatIdDeps, flatIdToExports, config, ssr),
  ],
  ...esbuildOptions,
});
// 打包元信息，后续会根据这份信息生成 _metadata.json
const meta = result.metafile!;
```

元信息写入磁盘

在打包过程完成之后，Vite 会拿到 Esbuild 构建的元信息，也就是上面代码中的 `meta` 对象，然后将元信息保存到 `_metadata.json` 文件中:

```
const data: DepOptimizationMetadata = {
  hash: mainHash,
  browserHash: mainHash,
  optimized: {},
};
// 省略中间的代码
for (const id in deps) {
  const entry = deps[id];
  data.optimized[id] = {
    file: normalizePath(path.resolve(cacheDir, flattenId(id) + ".js")),
    src: entry,
    // 判断是否需要转换成 ESM 格式，后面会介绍
    needsInterop: needsInterop(
      id,
      idToExports[id],
      meta.outputs,
      cacheDirOutputPath
    ),
  };
}
// 元信息写磁盘
writeFile(dataPath, JSON.stringify(data, null, 2));
```

到这里，预构建的核心流程就梳理完了，可以看到总体的流程上面并不复杂，但实际上为了方便你理解，在 依赖扫描 和 依赖打包 这两个部分中，我省略了很多的细节，每个细节代表了各种复杂的处理场景，因此，在下面的篇幅中，我们就来好好地剖析一下这两部分的应用场景和实现细节。

依赖扫描详细分析

1. 如何获取入口

现在让我们把目光聚焦在 `scanImports` 的实现上。大家可以先想一想，在进行依赖扫描之前，需要做的第一件事是什么？很显然，是找到入口文件。但入口文件可能存在于多个配置当中，比如 `optimizeDeps.entries` 和 `build.rollupOptions.input` ，同时需要考虑数组和对象的情况；也可能用户没有配置，需要自动探测入口文件。那么，在 `scanImports` 是如何做到的呢？

```
const explicitEntryPatterns = config.optimizeDeps.entries;
const buildInput = config.build.rollupOptions?.input;
if (explicitEntryPatterns) {
  // 先从 optimizeDeps.entries 寻找入口，支持 glob 语法
  entries = await globEntries(explicitEntryPatterns, config);
} else if (buildInput) {
  // 其次从 build.rollupOptions.input 配置中寻找，注意需要考虑数组和对象的情况
  const resolvePath = (p: string) => path.resolve(config.root, p);
  if (typeof buildInput === "string") {
    entries = [resolvePath(buildInput)];
  } else if (Array.isArray(buildInput)) {
    entries = buildInput.map(resolvePath);
  } else if (isObject(buildInput)) {
    entries = Object.values(buildInput).map(resolvePath);
  } else {
    throw new Error("invalid rollupOptions.input value.");
  }
} else {
  // 兜底逻辑，如果用户没有进行上述配置，则自动从根目录开始寻找
  entries = await globEntries("**/*.html", config);
}
```

其中 `globEntries` 方法即通过 `fast-glob` 库来从项目根目录扫描文件。

接下来我们还需要考虑入口文件的类型，一般情况下入口需要是 `js/ts` 文件，但实际上像 html、vue 单文件组件这种类型我们也是需要支持的，因为在这些文件中仍然可以包含 script 标签的内容，从而让我们搜集到依赖信息。

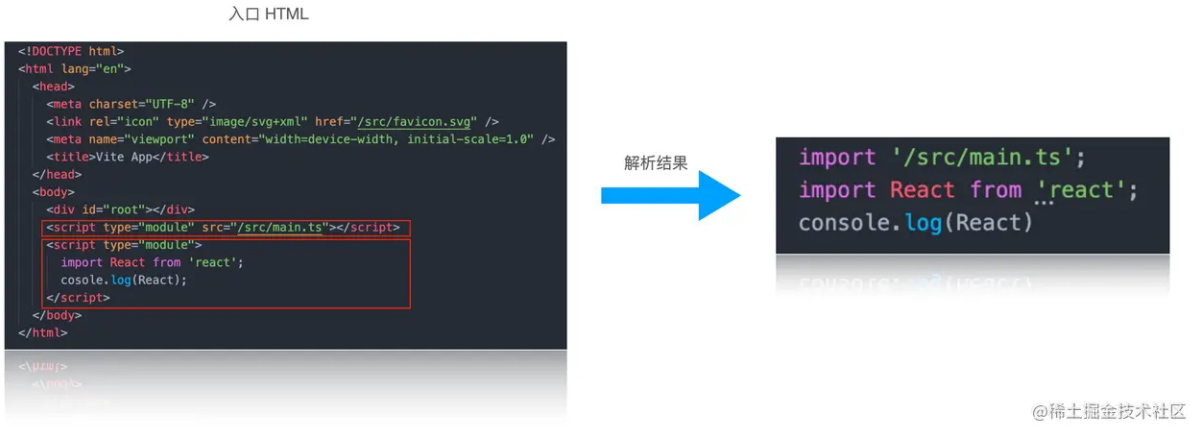
在源码当中，同时对 `html` 、 `vue` 、 `svelte` 、 `astro` (一种新兴的类 html 语法)四种后缀的入口文件进行了解析，当然，具体的解析过程在 依赖扫描 阶段的 Esbuild 插件中得以实现，接着就让我们在插件的实现中一探究竟。

```
const htmlTypesRE = /.(html|vue|svelte|astro)$/;
function esbuildScanPlugin(/* 一些入参 */): Plugin {
  // 初始化一些变量
  // 返回一个 Esbuild 插件
  return {
    name: "vite:dep-scan",
    setup(build) {
      // 标记「类 HTML」文件的 namespace
      build.onResolve({ filter: htmlTypesRE }, async ({ path, importer }) => {
        return {
          path: await resolve(path, importer),
          namespace: "html",
        };
      });

      build.onLoad(
        { filter: htmlTypesRE, namespace: "html" },
        async ({ path }) => {
          // 解析「类 HTML」文件
        }
```

```
      );
    },
  };
}
```

这里来我们以 `html` 文件的解析为例来讲解，原理如下图所示:



在插件中会扫描出所有带有 `type=module` 的 script 标签，对于含有 src 的 `script` 改写为一个 import 语句，对于含有具体内容的 script，则抽离出其中的脚本内容，最后将所有的 script 内容拼接成一段 js 代码。接下来我们来看具体的代码，其中会以上图中的 `html` 为示例来拆解中间过程:

```
const scriptModuleRE =
  /(<script\b[^>]*type\s*=\s*(?: module |'module')[^>]*>)(.*?)</script>/gims
export const scriptRE = /(<script\b(?:\s[^>]*>|>))(.*?)</script>/gims
export const commentRE = /<!--(.|[\r\n])*?-->/
const srcRE = /\bsrc\s*=\s*(?: ([^ ]+) |'([^']+)'|([^\s' >]+))/im
const typeRE = /\btype\s*=\s*(?: ([^ ]+) |'([^']+)'|([^\s' >]+))/im
const langRE = /\blang\s*=\s*(?: ([^ ]+) |'([^']+)'|([^\s' >]+))/im
// scan 插件 setup 方法内部实现
build.onLoad(
  { filter: htmlTypesRE, namespace: 'html' },
  async ({ path }) => {
    let raw = fs.readFileSync(path, 'utf-8')
    // 去掉注释内容，防止干扰解析过程
    raw = raw.replace(commentRE, '<!---->')
    const isHtml = path.endsWith('.html')
    // HTML 情况下会寻找 type 为 module 的 script
    // 正则：/(<script\b[^>]*type\s*=\s*(?: module |'module')[^>]*>)(.*?)</script>/gims
    const regex = isHtml ? scriptModuleRE : scriptRE
    regex.lastIndex = 0
    let js = ''
    let loader: Loader = 'js'
    let match: RegExpExecArray | null
    // 正式开始解析
    while ((match = regex.exec(raw))) {
      // 第一次: openTag 为 <script type= module  src= /src/main.ts >, 无 content
      // 第二次: openTag 为 <script type= module >, 有 content
      const [ , openTag , content] = match
```

```
      const [, openTag, content] = match
      const typeMatch = openTag.match(typeRE)
      const type =
        typeMatch && (typeMatch[1] || typeMatch[2] || typeMatch[3])
      const langMatch = openTag.match(langRE)
      const lang =
        langMatch && (langMatch[1] || langMatch[2] || langMatch[3])
      if (lang === 'ts' || lang === 'tsx' || lang === 'jsx') {
        // 指定 esbuild 的 loader
        loader = lang
      }
      const srcMatch = openTag.match(srcRE)
      // 根据有无 src 属性来进行不同的处理
      if (srcMatch) {
        const src = srcMatch[1] || srcMatch[2] || srcMatch[3]
        js += `import ${JSON.stringify(src)}\n`
      } else if (content.trim()) {
        js += content + '\n'
      }
    }
    return {
      loader,
      contents: js
    }
  )
```

这里对源码做了一定的精简，省略了 `vue` / `svelte` 以及 `import.meta.glob` 语法的处理，但不影响整体的实现思路，这里主要是让你了解即使是 `html` 或者类似这种类型的文件，也是能作为 Esbuild 的预构建入口来进行解析的。

2. 如何记录依赖?

入口的问题解决了，接下来还有一个问题: 如何在 Esbuild 编译的时候记录依赖呢?

Vite 中会把 `bare import` 的路径当做依赖路径，关于 `bare import` ，你可以理解为直接引入一个包名，比如下面这样:

```
import React from "react";
```

而以 `.` 开头的相对路径或者以 `/` 开头的绝对路径都不能算 `bare import`：

```
// 以下都不是 bare import
import React from "../node_modules/react/index.js";
import React from "/User/sanyuan/vite-project/node_modules/react/index.js";
```

对于解析 `bare import` 、记录依赖的逻辑依然实现在 scan 插件当中:

```
build.onResolve(
  {
    // avoid matching windows volume
    filter: /^[\w@][^:]/,
  },
  async ({ path: id, importer }) => {
    // 如果在 optimizeDeps.exclude 列表或者已经记录过了，则将其 externalize (排除)，直接 return

    // 接下来解析路径，内部调用各个插件的 resolveId 方法进行解析
    const resolved = await resolve(id, importer);
    if (resolved) {
      // 判断是否应该 externalize，下个部分详细拆解
      if (shouldExternalizeDep(resolved, id)) {
        return externalUnlessEntry({ path: id });
      }

      if (resolved.includes("node_modules") || include?.includes(id)) {
        // 如果 resolved 为 js 或 ts 文件
        if (OPTIMIZABLE_ENTRY_RE.test(resolved)) {
          // 注意了！现在将其正式地记录在依赖表中
          depImports[id] = resolved;
        }
        // 进行 externalize，因为这里只用扫描出依赖即可，不需要进行打包，具体实现后面的部分会讲到
        return externalUnlessEntry({ path: id });
      } else {
        // resolved 为 「类 html」 文件，则标记上 'html' 的 namespace
        const namespace = htmlTypesRE.test(resolved) ? "html" : undefined;
        // linked package, keep crawling
        return {
          path: path.resolve(resolved),
          namespace,
        };
      }
    } else {
      // 没有解析到路径，记录到 missing 表中，后续会检测这张表，显示相关路径未找到的报错
      missing[id] = normalizePath(importer);
    }
  }
);
```

顺便说一句，其中调用到了 resolve ，也就是路径解析的逻辑，这里面实际上会调用各个插件的 resolveId 方法来进行路径的解析，代码如下所示:

```
const resolve = async (id: string, importer?: string) => {
  // 通过 seen 对象进行路径缓存
  const key = id + (importer && path.dirname(importer));
  if (seen.has(key)) {
    return seen.get(key);
  }
  // 调用插件容器的 resolveId
  // 关于插件容器下一节会详细介绍，这里你直接理解为调用各个插件的 resolveId 方法解析路径即可
  const resolved = await container.resolveId(
    id
```

```
    importer && normalizePath(importer)
  );
  const res = resolved?.id;
  seen.set(key, res);
  return res;
};
```

3. external 的规则如何制定?

上面我们分析了在 Esbuild 插件中如何针对 `bare import` 记录依赖，那么在记录的过程中还有一件非常重要的事情，就是决定哪些路径应该被排除，不应该被记录或者不应该被 Esbuild 来解析。这就是 `external 规则` 的概念。

在这里，我把需要 external 的路径分为两类: **资源型**和**模块型**。

首先，对于资源型的路径，一般是直接排除，在插件中的处理方式如下:

```
// data url，直接标记 external: true，不让 esbuild 继续处理
build.onResolve({ filter: dataUrlRE }, ({ path }) => ({
  path,
  external: true,
}));
// 加了 ?worker 或者 ?raw 这种 query 的资源路径，直接 external
build.onResolve({ filter: SPECIAL_QUERY_RE }, ({ path }) => ({
  path,
  external: true,
}));
// css & json
build.onResolve(
  {
    filter: /.(css|less|sass|scss|styl|stylus|pcss|postcss|json)$/,
  },
  // 非 entry 则直接标记 external
  externalUnlessEntry
);
// Vite 内置的一些资源类型，比如 .png、.wasm 等等
build.onResolve(
  {
    filter: new RegExp(`\.(${KNOWN_ASSET_TYPES.join("|")})$`),
  },
  // 非 entry 则直接标记 external
  externalUnlessEntry
);
```

其中 `externalUnlessEntry` 的实现也很简单:

```
const externalUnlessEntry = ({ path }: { path: string }) => ({
  path,
```

```
    // 非 entry 则标记 external

    external: !entries.includes(path),
  });
```

其次，对于模块型的路径，也就是当我们通过 resolve 函数解析出了一个 JS 模块的路径，如何判断是否应该被 externalize 呢？这部分实现主要在 `shouldExternalizeDep` 函数中，之前在分析 `bare import` 埋了个伏笔，现在让我们看看具体的实现规则:

```
export function shouldExternalizeDep(
  resolvedId: string,
  rawId: string
): boolean {
  // 解析之后不是一个绝对路径，不在 esbuild 中进行加载
  if (!path.isAbsolute(resolvedId)) {
    return true;
  }
  // 1. import 路径本身就是一个绝对路径
  // 2. 虚拟模块(Rollup 插件中约定虚拟模块以`\0`开头)
  // 都不在 esbuild 中进行加载
  if (resolvedId === rawId || resolvedId.includes("\0")) {
    return true;
  }
  // 不是 JS 或者 类 HTML 文件，不在 esbuild 中进行加载
  if (!JS_TYPES_RE.test(resolvedId) && !htmlTypesRE.test(resolvedId)) {
    return true;
  }
  return false;
}
```

依赖打包详细分析

1. 如何达到扁平化的产物文件结构

一般情况下，esbuild 会输出嵌套的产物目录结构，比如对 vue 来说，其产物在 `dist/vue.runtime.esm-bundler.js` 中，那么经过 esbuild 正常打包之后，预构建的产物目录如下:

```
node_modules/.vite
├── _metadata.json
├── vue
│   └── dist
│       └── vue.runtime.esm-bundler.js
```

由于各个第三方包的产物目录结构不一致，这种深层次的嵌套目录对于 Vite 路径解析来说，其实是增加了不少的麻烦的，带来了一些不可控的因素。为了解决嵌套目录带来的问题，Vite 做了两件事情来达到扁平化的预构建产物输出:

> 嵌套路径扁平化， `/` 被换成下划线，如 `react/jsx-dev-runtime` ，被重写为 `react_jsx-dev-runtime` ；
> 用虚拟模块来代替真实模块，作为预打包的入口，具体的实现后面会详细介绍。

回到 `optimizeDeps` 函数中，其中在进行完依赖扫描的步骤后，就会执行路径的扁平化操作:

```
const flatIdDeps: Record<string, string> = {};
const idToExports: Record<string, ExportsData> = {};
const flatIdToExports: Record<string, ExportsData> = {};
// deps 即为扫描后的依赖表
// 形如: {
//    react :  /Users/sanyuan/vite-project/react/index.js  }
//    react/jsx-dev-runtime :  /Users/sanyuan/vite-project/react/jsx-dev-runtime.js
// }
for (const id in deps) {
  // 扁平化路径，`react/jsx-dev-runtime`，被重写为`react_jsx-dev-runtime`；
  const flatId = flattenId(id);
  // 填入 flatIdDeps 表，记录 flatId -> 真实路径的映射关系
  const filePath = (flatIdDeps[flatId] = deps[id]);
  const entryContent = fs.readFileSync(filePath, "utf-8");
  // 后续代码省略
}
```

对于虚拟模块的处理，大家可以把目光放到 `esbuildDepPlugin` 函数上面，它的逻辑大致如下:

```
export function esbuildDepPlugin(/* 一些传参 */) {
  // 定义路径解析的方法

  // 返回 Esbuild 插件
  return {
    name: 'vite:dep-pre-bundle',
    set(build) {
      // bare import 的路径
      build.onResolve(
        { filter: /^[\w@][^:]/ },
        async ({ path: id, importer, kind }) => {
          // 判断是否为入口模块，如果是，则标记上`dep`的 namespace，成为一个虚拟模块
        }
    }

    build.onLoad({ filter: /.*/, namespace: 'dep' }, ({ path: id }) => {
      // 加载虚拟模块
```

```
    }
  }
}
```

如此一来，Esbuild 会将虚拟模块作为入口来进行打包，最后的产物目录会变成下面的扁平结构:

```
node_modules/.vite
├── _metadata.json
├── vue.js
├── react.js
├── react_jsx-dev-runtime.js
```

2. 代理模块加载

虚拟模块代替了真实模块作为打包入口，因此也可以理解为 代理模块 ，后面也统一称之为 代理模块 。我们首先来分析一下代理模块究竟是如何被加载出来的，换句话说，它到底了包含了哪些内容。

拿 `import React from "react"` 来举例，Vite 会把 `react` 标记为 `namespace` 为 `dep` 的虚拟模块，然后控制 Esbuild 的加载流程，对于真实模块的内容进行重新导出。

那么第一步就是确定真实模块的路径:

```
// 真实模块所在的路径，拿 react 来说，即`node_modules/react/index.js`
const entryFile = qualified[id];
// 确定相对路径
let relativePath = normalizePath(path.relative(root, entryFile));
if (
  !relativePath.startsWith("./") &&
  !relativePath.startsWith("../") &&
  relativePath !== "."
) {
  relativePath = `./${relativePath}`;
}
```

确定了路径之后，接下来就是对模块的内容进行重新导出。这里会分为几种情况:

- CommonJS 模块
- ES 模块

那么，如何来识别这两种模块规范呢?

我们可以暂时把目光转移到 optimizeDeps 中，实际上在进行真正的依赖打包之前，Vite 会读取各个依赖的入口文件，通过 es-module-lexer 这种工具来解析入口文件的内容。这里稍微解释一下 es-module-lexer ，这是一个在 Vite 被经常使用到的工具库，主要是为了解析 ES 导入导出的语法，大致用法如下:

```
import { init, parse } from "es-module-lexer";
// 等待`es-module-lexer`初始化完成
await init;
const sourceStr = `
  import moduleA from './a';
  export * from 'b';
  export const count = 1;
  export default count;
`;
// 开始解析
const exportsData = parse(sourceStr);
// 结果为一个数组，分别保存 import 和 export 的信息
const [imports, exports] = exportsData;
// 返回 `import module from './a'`
sourceStr.substring(imports[0].ss, imports[0].se);
// 返回 ['count', 'default']
console.log(exports);
```

值得注意的是，export * from 导出语法会被记录在 import 信息中。

接下来我们来看看 optimizeDeps 中如何利用 es-module-lexer 来解析入口文件的，实现代码如下:

```
import { init, parse } from "es-module-lexer";
// 省略中间的代码
await init;
for (const id in deps) {
  // 省略前面的路径扁平化逻辑
  // 读取入口内容
  const entryContent = fs.readFileSync(filePath, "utf-8");
  try {
    exportsData = parse(entryContent) as ExportsData;
  } catch {
    // 省略对 jsx 的处理
  }
  for (const { ss, se } of exportsData[0]) {
    const exp = entryContent.slice(ss, se);
    // 标记存在 `export * from` 语法
    if (/export\s+*\s+from/.test(exp)) {
      exportsData.hasReExports = true;
    }
```

```
  }
  // 将 import 和 export 信息记录下来
  idToExports[id] = exportsData;
  flatIdToExports[flatId] = exportsData;
}
```

OK，由于最后会有两张表记录下 ES 模块导入和导出的相关信息，而 `flatIdToExports` 表会作为入参传给 Esbuild 插件:

```
// 第二个入参
esbuildDepPlugin(flatIdDeps, flatIdToExports, config, ssr);
```

如此，我们就能根据真实模块的路径获取到导入和导出的信息，通过这份信息来甄别 CommonJS 和 ES 两种模块规范。现在可以回到 Esbuild 打包插件中**加载代理模块**的代码:

```
let contents = "";
// 下面的 exportsData 即外部传入的模块导入导出相关的信息表
// 根据模块 id 拿到对应的导入导出信息
const data = exportsData[id];
const [imports, exports] = data;
if (!imports.length && !exports.length) {
  // 处理 CommonJS 模块
} else {
  // 处理 ES  模块
}
```

如果是 CommonJS 模块，则导出语句写成这种形式:

```
let contents = "";
contents += `export default require( ${relativePath} );`;
```

如果是 ES 模块，则分**默认导出**和**非默认导出**这两种情况来处理:

```
// 默认导出，即存在 export default 语法
if (exports.includes("default")) {
  contents += `import d from  ${relativePath} ;export default d;`;
}
// 非默认导出
if (
  // 1. 存在 `export * from` 语法，前文分析过
  data.hasReExports ||
  // 2. 多个导出内容
  exports.length > 1 ||
  // 3. 只有一个导出内容，但这个导出不是 export default
  exports[0] !== "default"
```

```
) {
  // 凡是命中上述三种情况中的一种，则添加下面的重导出语句
  contents += `\nexport * from  ${relativePath} `;
}
```

现在，我们组装好了 代理模块 的内容，接下来就可以放心地交给 Esbuild 加载了:

```
let ext = path.extname(entryFile).slice(1);
if (ext === "mjs") ext = "js";
return {
  loader: ext as Loader,
  // 虚拟模块内容
  contents,
  resolveDir: root,
};
```

3. 代理模块为什么要和真实模块分离?

现在，相信你已经清楚了 Vite 是如何组装代理模块，以此作为 Esbuild 打包入口的，整体的思路就是先分析一遍模块真实入口文件的 `import` 和 `export` 语法，然后在代理模块中进行重导出。这里不妨回过头来思考一下: 为什么要对真实文件先做语法分析，然后重导出内容呢?

对此，大家不妨注意一下代码中的这段注释:

```
// It is necessary to do the re-exporting to separate the virtual proxy
// module from the actual module since the actual module may get
// referenced via relative imports - if we don't separate the proxy and
// the actual module, esbuild will create duplicated copies of the same
// module!
```

翻译过来即:

> 这种重导出的做法是必要的，它可以分离虚拟模块和真实模块，因为真实模块可以通过相对地址来引入。如果不这么做，Esbuild 将会对打包输出两个一样的模块。

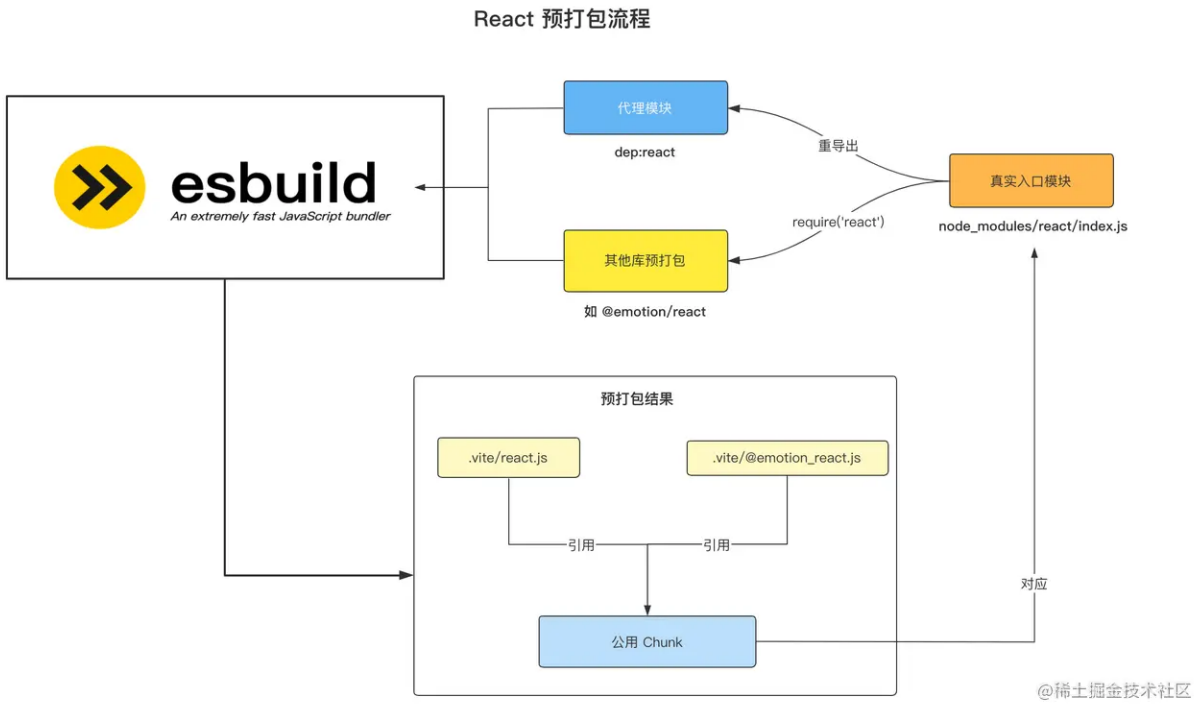
刚开始看的确不太容易理解，接下来我会通过对比的方式来告诉你这种设计到底解决了什么问题。

假设我不像源码中这么做，在虚拟模块中直接将**真实入口的内容**作为传给 Esbuild 可不可

以呢？也就是像这样：

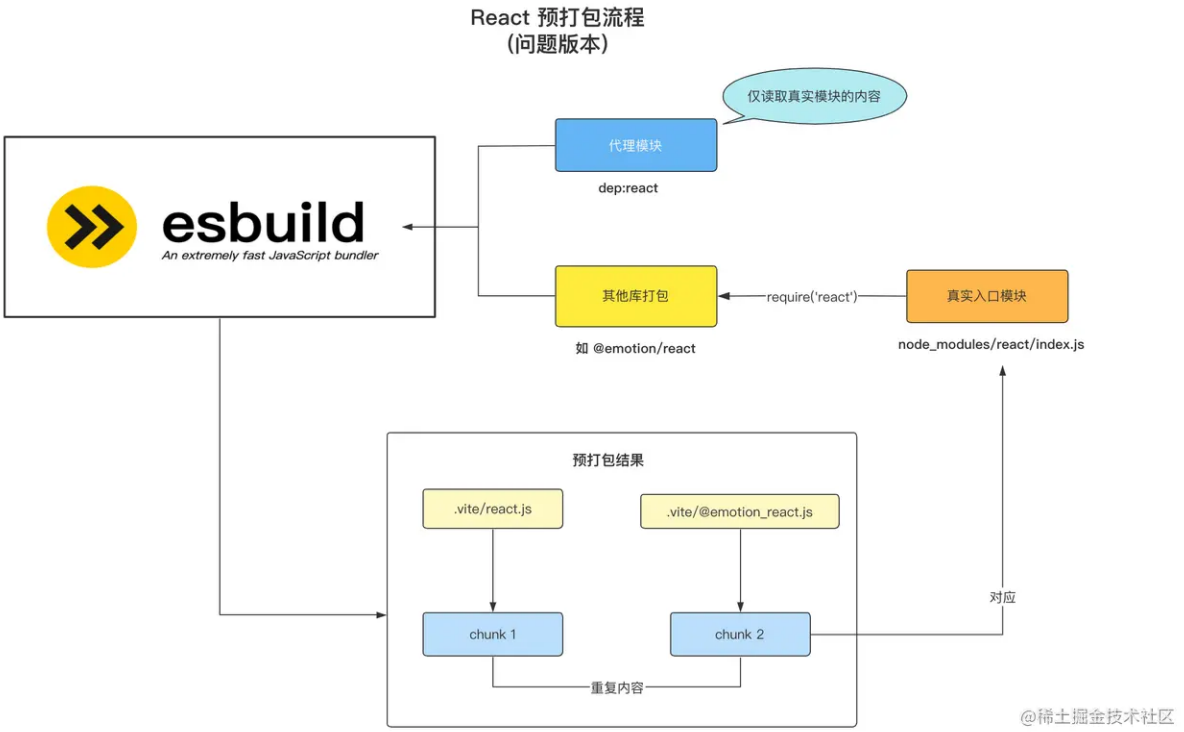
```
build.onLoad({ filter: /.*/, namespace: 'dep' }, ({ path: id }) => {
  // 拿到查表拿到真实入口模块路径
  const entryFile = qualified[id];
  return {
    loader: 'js',
    contents: fs.readFileSync(entryFile, 'utf8');
  }
}
```

那么，这么实现会产生什么问题呢？我们可以先看看正常的预打包流程（以 React 为例）：



Vite 会使用 `dep:react` 这个代理模块来作为入口内容在 Esbuild 中进行加载，与此同时，其他库的预打包也有可能会引入 React，比如 `@emotion/react` 这个库里面会有 `require('react')` 的行为。那么在 Esbuild 打包之后， `react.js` 与 `@emotion_react.js` 的代码中会引用同一份 Chunk 的内容，这份 Chunk 也就对应 React 入口文件（`node_modules/react/index.js`）。

这是理想情况下的打包结果，接下来我们来看看上述有问题的版本是如何工作的:



现在如果代理模块通过文件系统直接读取真实模块的内容，而不是进行重导出，因此由于此时代理模块跟真实模块并没有任何的引用关系，这就导致最后的 `react.js` 和 `@emotion/react.js` 两份产物并不会引用同一份 Chunk，Esbuild 最后打包出了内容完全相同的两个 Chunk！

这也就能解释为什么 Vite 中要在代理模块中对真实模块的内容进行重导出了，主要是为了避免 Esbuild 产生重复的打包内容。此时，你是不是也恍然大悟了呢？

小结

本文的正文内容到此就接近尾声了，我们终于学习完了 Esbuild 预构建的底层实现，在这一节中，我首先带你熟悉一遍预构建的核心流程，包括**缓存判断**、**依赖扫描**、**依赖打包**和**元信息写入磁盘**这四个主要的步骤，让你从宏观上对 Vite 预构建流程有了初步的认识。

从微观的实现层面，我带你深入分析了 依赖扫描 的具体实现，从三个角度梳理了依赖扫描要解决的三个问题，分别是 如何获取入口 、 如何记录依赖 以及 如何制定 `external` 的规则 ，并且与你重点分析 `scanImports` 函数的实现。接着我们继续深入到 依赖打包 的源码实现，带你了解到 Vite 是如何通过 嵌套路径扁平化 和 代理模块 最终达到了扁平化的预构建产物结构，然后重点带你剖析了 代理模块 背后的设计原因，如果不这么做会产生什么问题，让你不仅知其然，同时也知其所以然。

相信经历过这一节的内容，你已经对 Vite 的预构建有了更加深刻的理解，也恭喜你，拿下了这一块困难而又深度的内容，我们下节再见。

上一篇：配置解析服务：配置文件在 Vite 内部被转换成什么样子了？

下一篇：插件流水线：从整体到局部，理解 Vite 的核心编译能力