

vite核心原理（五）—— 请求处理（上）

该部分解析基于我们实现的简单vite中的代码，是vite源码的阉割版，希望用最简洁的代码来了解vite的核心原理。其中大部分逻辑和结构都和源码保持一致，方便阅读源代码。

我们知道vite在开发阶段是基于浏览器对ESM的支持实现的本地服务，每一个import都会发起一次资源的请求，而vite会通过服务监听的方式监听到资源的请求，对于不同类型的请求，vite会调用不同的插件进行转换处理，最终处理成浏览器能够识别的内容响应到客户端。

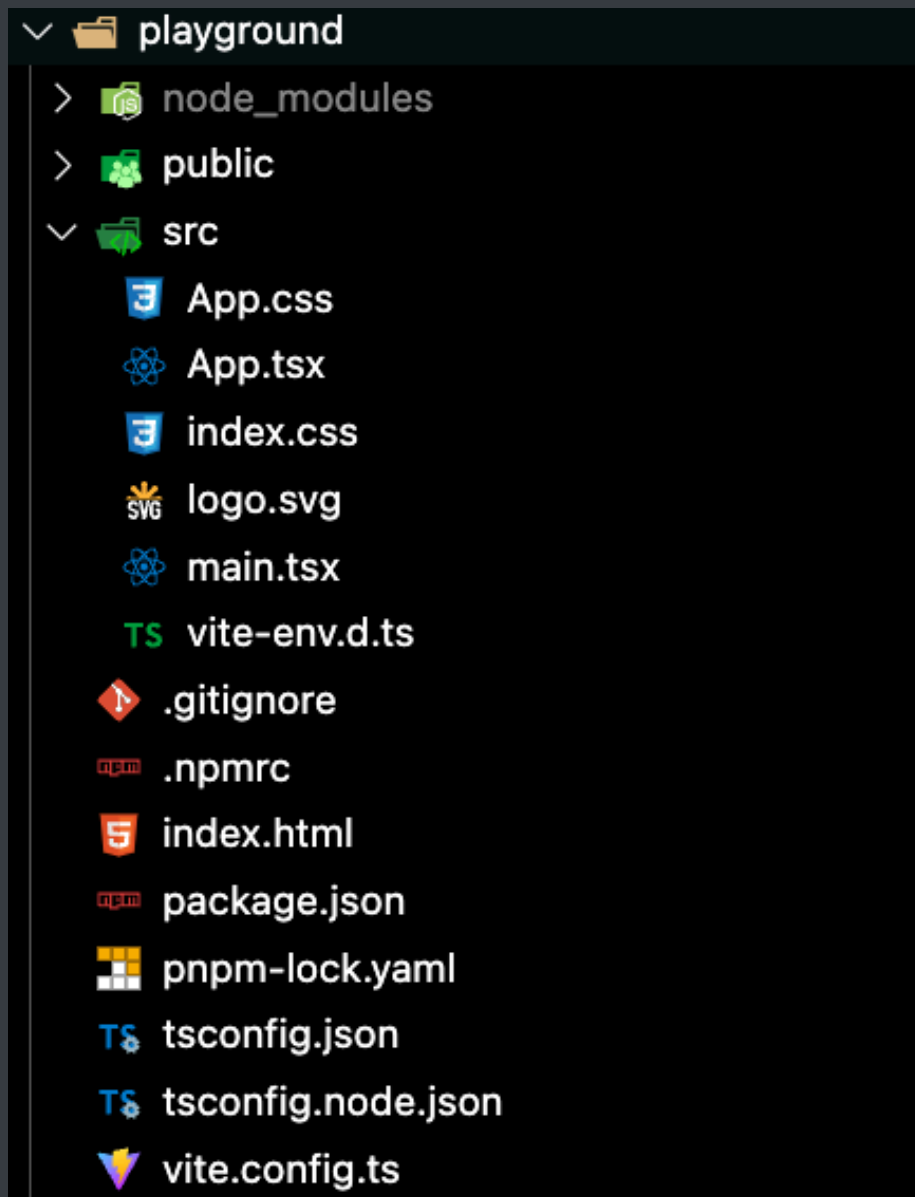
首先，在第一部分的源码解析中有提到vite服务的启动入口createServer函数，这个函数中会借助connect这个中间件框架来扩展vite服务的功能。在本项目中，主要注册了三个中间件：

- indexHtmlMiddleware: html文件处理中间件
- transformMiddleware: 资源转换处理中间件，内部调用核心编译函数进行处理
- staticMiddleware: 静态资源处理中间件，用于处理类似图片等静态资源获取的中间件

下面会用一个简单的react项目来讲解vite的整个请求处理流程以及上述三个中间件的功能。

```
// node/server/index.ts
export const createServer = async (inlineConfig: InlineConfig = {}) => {
  // 省略其他代码
  // 中间件框架
  const app = connect() as any;
  // 注册中间件
  app.use(indexHtmlMiddleware(serverContext));
  app.use(transformMiddleware(serverContext));
  app.use(staticMiddleware(serverContext.root));
}
```

我们通过vite搭建一个简单的react18项目，目录结构大致如下。其中静态页面为index.html，src下有一个入口文件main.tsx，一个函数式组件App.tsx以及样式文件App.css和静态资源logo.svg。我们可以将node_modules中的vite源码替换成自己实现的阉割版vite，方便调试代码。



项目创建完成后可以通过pnpm dev的方式启动，这时候就会走我们前面两节介绍的配置解析、本地服务启动以及初步的预构建流程。至于说为什么是初步的预构建，有以下两点：

1. 启动时的预构建只是将产物放到了临时目录deps_temp中，并没有将产物输出到真实目录deps中，只有当资源真正被访问时才会输出到真实目录下
2. 启动时的预构建只是基于静态分析的构建，运行时有可能会产生新的依赖需要构建（动态导入依赖）

当我们通过服务器启动的地址打开浏览器时会先请求html资源，这时候就会进入indexHtmlMiddleware中间件的处理逻辑：

1. 拦截请求路径。当我们打开浏览器请求时，默认第一个请求是/
2. 通过getHtmlFilename方法获取html文件路径并读取文件内容

3. 调用transformIndexHtml方法处理读取到的html内容（此处处理的内容会在hmr部分介绍），返回处理后的结果并通过res.end响应

```
// node/server/middleware/indexHtml.ts
export function indexHtmlMiddleware(
  serverContext: ServerContext
): NextHandleFunction {
  return async (req, res, next) => {
    const url = req.url;
    if (url === "/") {
      const filename = getHtmlFilename(url, serverContext) +
"/index.html";
      // 判断文件是否存在
      if (fs.existsSync(filename)) {
        let html = fs.readFileSync(filename, 'utf-8');
        // 调用html解析函数
        html = await serverContext.transformIndexHtml(url, html,
req.originalUrl);
        res.statusCode = 200;
        res.setHeader("Content-Type", "text/html");
        return res.end(html);
      }
    }
    return next();
  };
}

const getHtmlFilename = (url: string, server: ServerContext) => {
  return
decodeURIComponent(normalizePath(path.join(server.config.root,
url.slice(1))));
}
```

当浏览器接收到html内容时就会开始解析内容，解析过程中如果遇到需要请求其它资源时，就会再次发情网络请求。以项目中的index.html为例会再发起对main.tsx文件的请求。

```

<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Vite App</title>
</head>

<body>
  <div id="root"></div>
  <!-- 测试 -->
  <script type="module" src="/src/main.tsx"></script>
</body>

</html>

```

当发起对/src/main.tsx文件的请求时，会进入到transformMiddleware中间件的处理流程：

1. 拦截js、css或者带有import（类似图片等资源可能会携带?import）的请求
2. 调用核心编译函数transformRequest对请求内容进行处理并响应结果

```

// node/server/middleware/transform.ts
export function transformMiddleware(
  serverContext: ServerContext
): NextHandleFunction {
  return async (req, res, next) => {
    if (req.method !== "GET" || !req.url) {
      return next();
    }
    const url = req.url;
    if (isJSRequest(url) || isCSSRequest(url) ||
    isImportRequest(url)) {
      // 核心编译函数
      let result = await transformRequest(url, serverContext);
      if (!result) {
        return next();
      }
      if (result && typeof result !== "string") {
        result = result.code;
      }
    }
    return next();
  };
}

```

```

    }
    // 编译完成，返回响应给浏览器
    res.statusCode = 200;
    res.setHeader("Content-Type", "application/javascript");
    return res.end(result);
  }

  next();
};
}

```

对于transformMiddleware中间件来说，核心逻辑都在transformRequest中的doTransform方法中：

1. 清除url后面的时间戳，热更新重新发起请求时会带上时间戳（后续热更新会提及）
2. 根据模块依赖图获取之前编译缓存的结果。如果有，则返回缓存内容
3. 调用插件的resolveId方法获取资源的路径 —— resolveId
4. 根据转换后的资源路径去查找资源并进行转化处理 —— loadAndTransform
5. 处理运行过程中发现的依赖，返回处理后的结果

```

// node/server/transformRequest.ts
export async function transformRequest(
  url: string,
  serverContext: ServerContext
) {
  const transformResult = doTransform(url, serverContext);
  return transformResult;
}

const doTransform = async (
  url: string,
  server: ServerContext,
) => {
  // 清除url后面的时间戳，热更新重新发起请求时会带上时间戳

```

```

url = removeTimestampQuery(url);
const { pluginContainer, config } = server;
// 获取缓存的模块
const module = await server.moduleGraph.getModuleByUrl(url);
// 如果有缓存则直接返回缓存的结果
const cached = module && module.transformResult;
if (cached) {
  console.log(green(`[memory] ${url}`));
  return cached;
}
const id = (await pluginContainer.resolveId(url))?.id || url;
const transformResult = loadAndTransform(id, url, server);
// 处理运行过程中发现的依赖
getDepsOptimizer(config)?.delayDepsOptimizerUntil(id, () =>
transformResult);
return transformResult;
};

```

这块调用了插件的resolveId钩子函数，resolveId是一个优先的钩子，只要其中一个插件的resolveId返回了数据就会终止执行。我们深入到resolveId钩子，看看插件如何解析/src/main.tsx：

1. /src/main.tsx属于绝对路径，会进入if (id.startsWith('/'))这个条件分支，在这个分支中会调用tryFsResolve解析路径
2. 在tryFsResolve中，首先会调用tryResolveFile方法获取文件路径，如果没有找到文件路径则会尝试拼接文件后缀进行查找
3. tryResolveFile方法中会先调用fs.statSync方法查找文件信息，如果文件信息存在则调用getRealPath获取真实路径
4. getRealPath核心逻辑是调用fs.realpathSync，用于同步计算给定路径的规范路径名。它是通过解决。..以及路径中的符号链接，并返回解析后的路径
5. 假设我们的项目本地路径是/users/test-vite，那么/src/main.tsx会解析成/users/test-vite/src/main.tsx

```
// node/plugins/resolve.ts
```

```
export function resolvePlugin(resolveOptions: Record<string, any>):
Plugin {
  const { root } = resolveOptions;
  return {
    name: "m-vite:resolve",
    async resolveId(id: string, importer?: string, resolveOpts?:
Record<string, any>) {
      const options = {
        ...resolveOptions,
        scan: resolveOpts?.scan ?? resolveOptions.scan,
      };
      const depsOptimizer = resolveOptions.getDepsOptimizer?();

      // 预构建依赖的特殊处理
      if (depsOptimizer?.isOptimizedDepUrl(id)) {
        return normalizePath(path.resolve(root, id.slice(1)));;
      }
      // 1. 绝对路径
      if (id.startsWith('/')) {
        let res;
        const fsPath = path.resolve(root, id.slice(1));
        if ((res = tryFsResolve(fsPath, options))) {
          return res;
        }
      }
      // 2. 相对路径
      else if (id.startsWith(".")) {
        // 省略其它代码
      }
      // 外部包的导入
      if (bareImportRE.test(id)) {
        // 省略其它代码
      }
      return null;
    },
  };
};
```



```
}
```

```
const tryFsResolve = (fsPath: string, options: any) => {  
  let res;  
  if ((res = tryResolveFile(fsPath, options))) {  
    return res;  
  }  
  // 尝试添加后缀名获取文件  
  for (const ext of options.extensions) {  
    if (res = tryResolveFile(fsPath + ext, options)) {  
      return res;  
    }  
  }  
};
```

```
const tryResolveFile = (  
  file: string,  
  options: any  
) => {  
  let stat;  
  try {  
    // 获取文件信息, 判断文件是否存在  
    stat = fs.statSync(file, { throwIfNoEntry: false });  
  }  
  catch {  
    return;  
  }  
  // 如果文件存在则获取文件的真实路径  
  if (stat) {  
    return getRealPath(file, options.preserveSymlinks);  
  }  
}
```

```
const getRealPath = (resolved: string, preserveSymlinks?: boolean) => {  
  // 用于同步计算给定路径的规范路径名。它是通过解决。,...以及路径中的符号链接, 并返回解析后的路径
```

```
    resolved = fs.realpathSync(resolved);  
    return normalizePath(resolved);  
};
```

我们接着看loadAndTransform方法的实现：

1. 清除url后面的参数
2. 调用插件的load方法获取文件内容，load方法中的id就是前面通过resolveId解析得到的结果
3. 创建资源url对应的模块节点
4. 调用插件的transform方法对代码进行转换
5. 将转换的结果缓存到依赖节点上并返回转换结果

```
// node/server/transformRequest.ts  
const loadAndTransform = async (  
  id: string,  
  url: string,  
  server: ServerContext,  
) => {  
  const { pluginContainer, moduleGraph } = server;  
  url = cleanUrl(url);  
  let transformResult;  
  let code = await pluginContainer.load(id);  
  if (typeof code === "object" && code !== null) code = code.code;  
  // 模块加载成功，则将模块更新到模块依赖图中  
  const mod = await moduleGraph.ensureEntryFromUrl(url);  
  if (code) {  
    transformResult = await pluginContainer.transform(  
      code as string,  
      id  
    );  
  }  
  // 缓存模块转换结果  
  mod && (mod.transformResult = transformResult);
```

```
    return transformResult;
}
```

load和transform也是vite插件体系内的两个钩子函数。load是异步优先钩子，transform则是异步串行钩子。这里继续深入了解这两个插件钩子的执行过程。

首先看一下main.tsx的内容：

```
// src/main.tsx
import ReactDOM from "react-dom/client";
import App from "./App";

const root = ReactDOM.createRoot(document.getElementById('root') as
HTMLElement);
root.render(<App />);
```

在执行插件的load方法时会执行optimizedDepsPlugin的load方法：

1. 调用getDepsOptimizer获取依赖优化器对象，依赖优化器对象depsOptimizer在预构建章节有提及
2. 调用depsOptimizer的isOptimizedDepFile方法判断资源id是否需要预构建的依赖，判断依据就是资源id是否是以预构建缓存地址开头 (/users/test-vite/node_modules/vite/deps/)
3. /src/main.tsx经过resolveId处理后为/users/vite-test/src/main.tsx，不属于需要预构建的资源，因此直接返回

```
// node/plugins/optimizedDepsPlugin.ts
export const optimizedDepsPlugin = (config: ResolvedConfig): any => {
  return {
    name: 'm-vite:optimized-deps',
    async resolveId(id: string) {
      // 判断是否是预构建的依赖
      if (getDepsOptimizer(config)?.isOptimizedDepFile(id)) {
        return id;
      }
    }
  }
}
```

```

    },
    async load(id: string, options: Record<string, any>) {
        const depsOptimizer = getDepsOptimizer(config);
        // 判断是否是预构建的依赖
        if (depsOptimizer?.isOptimizedDepFile(id)) {
            const metadata = depsOptimizer.metadata;
            const info = optimizedDepInfoFromFile(metadata, id);
            if (info) {
                // 如果info存在需要等待其预构建完成, 此时磁盘中
                (/node_modules/m-vite/deps)已经生成了预构建结果
                await info.processing;
            }
            try {
                return await fs.readFile(id, 'utf-8');
            } catch (error) {
                console.log(id, error);
            }
        }
    }
}

// node/optimizer/optimizer.ts
export const getDepsOptimizer = (
    config: ResolvedConfig,
): DepsOptimizer | undefined => depsOptimizerMap.get(config);


// node/optimizer/index.ts
export const isOptimizedDepFile = (
    id: string,
    config: ResolvedConfig,
): boolean => id.startsWith(getDepsCacheDirPrefix(config));

```

由于上面的optimizedDepsPlugin的load没有返回内容, 因此会走下一个插件的load逻辑, 下一个执行的插件为esbuildPlugin。esbuildPlugin的load方法很简单, 直接调用fs.readFile读取文件内容, 返回代码字符串

```
// node/plugins/esbuild.ts
export function esbuildPlugin(): Plugin {
  return {
    name: "m-vite:esbuild",
    // 加载模块
    async load(id) {
      if (isJSRequest(id)) {
        try {
          const code = await readFile(id, "utf-8");
          return code;
        } catch (e) {
          return null;
        }
      }
    }
  };
  // 省略其它代码
};
}
```

我们可以在vscode的调试终端中打印出code的内容。当我们拿到load之后的内容后就会结束load钩子的调用，接下去就是transform钩子的执行

```
 No-Bundle 服务已经成功启动! 耗时: 3ms
> 本地访问路径: http://localhost:3000
import ReactDOM from "react-dom/client";
import App from "./App";

const root = ReactDOM.createRoot(document.getElementById('root') as HTMLElement);
root.render(<App />);
```

首先会执行esbuildPlugin的transform方法：

1. 获取文件后缀名，后续通过esbuild打包时会根据后缀名使用不同的loader
2. 针对react的相关配置项扩展，例如jsx的处理
3. 调用esbuild的transform方法进行代码转换

```
export function esbuildPlugin(): Plugin {
```

```

return {
  name: "m-vite:esbuild",
  // 省略其它代码
  async transform(code, id) {
    if (isJSRequest(id)) {
      const extname = path.extname(id).slice(1);
      const compilerOptions: Record<string, any> = {};
      if (extname === 'tsx' || extname === 'ts') {
        compilerOptions.jsx = 'react-jsx';
      }
      const { code: transformedCode, map } = await
transform(code, {
        loader: extname as "js" | "ts" | "jsx" | "tsx",
        sourcefile: id,
        target: "esnext",
        format: "esm",
        sourcemap: true,
        treeShaking: false,
        tsconfigRaw: {
          compilerOptions
        }
      });
      return {
        code: transformedCode,
        map,
      };
    }
    return null;
  },
};
}

```

我们继续打印一下main.tsx的转换结果，这个结果会作为参数传到下一个插件的transform方法中：

```
// main.tsx经过esbuild处理后的结果
import { jsx } from 'react/jsx-runtime';
import ReactDOM from 'react-dom/client';
import App from './App';
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(/* #__PURE__ */ jsx(App, {}));
```

接着会执行importAnalysisPlugin的transform方法：

1. 排除非js以及需要忽略的请求资源处理
2. 调用es-module-lexer插件的init方法，再调用parse方法得到import的分析结果
3. 创建MagicString实例，这是一个可以操作字符串库（magic-string）
4. 初始化模块依赖图相关的数据（importerModule、importedUrls），在这部分更新模块依赖图的依赖信息
5. 循环处理import信息，这部分主要是对import的语句信息作分析转换。比如导入资源路径的转换，导入方式的转换等等
6. 将转换结果通过MagicString进行字符串操作替换，得到最终结果并返回

```
export function importAnalysisPlugin(config: ResolvedConfig): Plugin {
  let serverContext: ServerContext;
  const { root } = config;
  return {
    name: "m-vite:import-analysis",
    // 省略其它代码
    async transform(code: string, importer: string) {
      // 只处理 JS 相关的请求
      if (!isJSRequest(importer) || isInternalRequest(importer))
        return null;
      // parse前需要调用
      await init;
      // 解析 import 语句
      const [imports] = parse(code);
      let s: MagicString | undefined;
      const str = () => s || (s = new MagicString(code));
```

```

const normalizeUrl = async (url: string, pos: number) => {
  const resolved = await this.resolve!(url, importer);
  if (!resolved) console.error('error');
  const id = resolved.id;
  if (id.startsWith(root + '/')) {
    url = id.slice(root.length);
  }
  if (isExternalUrl(url)) {
    return [url, url];
  }
  // 对于非js和非css的资源，例如静态资源，会在url后面加上 ?

```

import 后缀

```

    url = markExplicitImport(url);
    return [url, id];
  };
const depsOptimizer = getDepsOptimizer(config);
const { moduleGraph } = serverContext;
const importerModule = moduleGraph.getModuleById(importer)!;
const importedUrls: Set<string | ModuleNode> = new Set();

// 对每一个 import 语句依次进行分析
for (let index = 0; index < imports.length; index++) {
  const importInfo = imports[index];
  let { s: modStart, e: modEnd, n: specifier } =
importInfo;

  const rawUrl = code.slice(modStart, modEnd);

  // 省略其它代码

  /**
   * 静态导入或动态导入中的有效字符串，如果可以解析，让我们解析它
   */
  if (!specifier) continue;
  const [url, resolvedId] = await normalizeUrl(specifier,
modStart);

```



```

        let rewriteDone = false;
        /**
         * 对于优化的 cjs deps, 通过将命名导入重写为 const 赋值来支持命名导入
         * 内部优化的块不需要 es interop 并且被排除在外 (chunk-xxxx)
         */
        if (
            depsOptimizer?.isOptimizedDepFile(resolvedId) &&
            !resolvedId.match(optimizedDepChunkRE)
        ) {
            const file = cleanUrl(resolvedId); // 删除 ?v={hash}
            const needsInterop = await
optimizedDepNeedsInterop(depsOptimizer.metadata, file, config);
            if (needsInterop) {
                interopNamedImports(str(), imports[index], url,
index);

                rewriteDone = true;
            }
        }
        if (!rewriteDone) {
            str().overwrite(modStart, modEnd, url, {
                contentOnly: true,
            });
        }
        importedUrls.add(url);
    }

    // 省略其它代码

    // 处理非css资源的模块依赖图, css的依赖关系由css插件内部处理
    if (!isCSSRequest(importer)) await
moduleGraph.updateModuleInfo(importerModule, importedUrls);

    if (s) return transformStableResult(s);
    return {
        code

```

```

    };
  },
};
}

```

上面是整个transform的大致流程，我们接着以main.tsx为例进行分析：

1. 在上一个插件（esbuildPlugin）的transform中，我得到了初步的转换代码：

```

// main.tsx经过esbuild.transform转换后的结果
import { jsx } from "react/jsx-runtime";
import ReactDOM from "react-dom/client";
import App from "./App";
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(/* #__PURE__ */ jsx(App, {}));

```

接着调用es-module-lexer的parse方法对这段代码字符串作import的解析：

```

// parse后的结果
[
  { n: 'react/jsx-runtime', s: 21, e: 38, ss: 0, se: 39, d: -1, a: -1 },
  { n: 'react-dom/client', s: 63, e: 79, ss: 41, se: 80, d: -1, a: -1 },
  { n: './App', s: 99, e: 104, ss: 82, se: 105, d: -1, a: -1 }
]

```

2. 循环分析每一个import信息：

第一个是react/jsx-runtime，它会先经过normalizeUrl处理，得到转换后的路径信息：

```

// react/jsx-runtime ==> [url, id]
[
  '/node_modules/.m-vite/deps/react_jsx-runtime.js',
  '/users/test-vite/node_modules/.m-vite/deps/react_jsx-runtime.js'
]

```

紧接着会判断这个资源是否是需要同优化的cjs依赖：

- 是否是预构建缓存资源
- 内部优化的资源不需要 es interop 并且被排除在外（react在构建时会生成chunk-xxxx.js）
- 是否需要转换（needsInterop）。上述import { jsx } from "react/jsx-runtime"在浏览器端运行会报错，因为react/jsx-runtime内部是个默认导出，它需要通过将命名导入重写为 const 赋值来支持命名导入。其中needsInterop的值也是借助了es-module-lexer的parse方法加后续分析得到，具体可以看optimizedDepNeedsInterop的逻辑

```
✖ Uncaught SyntaxError: The requested module '/node_modules/.m-vite/deps/react_jsx-runtime.js' does not provide an export named 'jsx' (at App.tsx:1:148)
```

```
// node/plguins/importAnalysis.ts
if (
  depsOptimizer?.isOptimizedDepFile(resolvedId) &&
  !resolvedId.match(optimizedDepChunkRE)
) {
  const file = cleanUrl(resolvedId); // 删除 ?v={hash}
  const needsInterop = await
optimizedDepNeedsInterop(depsOptimizer.metadata, file, config);
  if (needsInterop) {
    interopNamedImports(str(), imports[index], url, index);
    rewriteDone = true;
  }
}

// 重写导入方式
export const interopNamedImports = (
  str: MagicString,
  importSpecifier: ImportSpecifier,
  rewrittenUrl: string,
  importIndex: number,
) => {
  const source = str.original;
  const { s: start, e: end, ss: expStart, se: expEnd, d:
dynamicIndex, } = importSpecifier;
  if (dynamicIndex > -1) {
```

```

        // 重写 `import('package')` 为default默认导入
        str.overwrite(expStart, expEnd,
`import('${rewrittenUrl}').then(m => m.default &&
m.default.__esModule ? m.default : ({ ...m.default, default:
m.default })))`, { contentOnly: true });
    } else {
        const exp = source.slice(expStart, expEnd);
        const rawUrl = source.slice(start, end);
        // 重写内容
        const rewritten = transformCjsImport(exp, rewrittenUrl,
rawUrl, importIndex);
        rewritten ?
            str.overwrite(expStart, expEnd, rewritten, {
contentOnly: true }) :
            // export * from '...'
            str.overwrite(start, end, rewrittenUrl, { contentOnly:
true });
    }
};

```

经过interopNamedImports转换后：

```

// main.tsx中的import { jsx } from "react/jsx-runtime"被转换了
import __vite__cjsImport0_react_jsxRuntime from "/node_modules/.m-
vite/deps/react_jsx-runtime.js"; const jsx =
__vite__cjsImport0_react_jsxRuntime["jsx"];
import ReactDOM from "react-dom/client";
import App from "./App";
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(/* #__PURE__ */ jsx(App, {}));

```

第二个是react-dom/client，它经过normalizeUrl处理后会得到如下结果：

其中react-dom/client会被结构扁平化处理成react-dom_client，这块在预构建章节也有提到

```
// react-dom/client ==> [url, id]
[
  '/node_modules/.m-vite/deps/react-dom_client.js',
  '/users/test-vite/node_modules/.m-vite/deps/react-dom_client.js'
]
```

同jsx-runtime一样，react-dom的引入也会被转换：

```
// main.tsx中的import ReactDOM from "react-dom/client"被转换了
import __vite__cjsImport0_react_jsxRuntime from "/node_modules/.m-
vite/deps/react_jsx-runtime.js"; const jsx =
__vite__cjsImport0_react_jsxRuntime["jsx"];
import __vite__cjsImport1_reactDom_client from "/node_modules/.m-
vite/deps/react-dom_client.js"; const ReactDOM =
__vite__cjsImport1_reactDom_client.__esModule ?
__vite__cjsImport1_reactDom_client.default :
__vite__cjsImport1_reactDom_client;
import App from "./App";
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(/* @__PURE__ */ jsx(App, {}));
```

最后一个就是./App，先看一下路径解析结果：

```
// ./App ==> [url, id]
[
  '/src/App.tsx',
  '/users/test-vite/src/App.tsx'
]
```

显然./App不需要经过interopNamedImports转换处理，只需要后续的路径替换即可：

```
// 没有经过字符串重写，将原先的导入替换成绝对路径导入
if (!rewriteDone) {
  str().overwrite(modStart, modEnd, url, {
    contentOnly: true,
  });
}
```

最终替换的结果：

```
// 将import App from "../App"替换成了import App from "/src/App.tsx";
import __vite__cjsImport0_react_jsxRuntime from "/node_modules/.m-
vite/deps/react_jsx-runtime.js"; const jsx =
__vite__cjsImport0_react_jsxRuntime["jsx"];
import __vite__cjsImport1_reactDom_client from "/node_modules/.m-
vite/deps/react-dom_client.js"; const ReactDOM =
__vite__cjsImport1_reactDom_client.__esModule ?
__vite__cjsImport1_reactDom_client.default :
__vite__cjsImport1_reactDom_client;
import App from "/src/App.tsx";
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(/* #__PURE__ */ jsx(App, {}));
```

这里我们可能好奇./App是如何被替换成/src/App.tsx的，这块在路径解析resolveId部分也被省略了，下面来解释一下这部分的逻辑：

1. ./App会先经过path.resolve(basedir, id)处理得到/users/test-vite/src/App，这是一个没有后缀的路径
2. 执行tryFsResolve方法，tryFsResolve方法会先根据当前请求id去判断资源是否存在，如果不存在则会通过追加后缀的方式去尝试获取资源。这里经过追加后缀的方式能够获取到资源/users/test-vite/src/App.tsx。因此，我们最终得到的路径就是/users/test-vite/src/App.tsx。

```
// node/plugins/resolve.ts
// 2. 相对路径的处理
else if (id.startsWith(".")) {
  if (!importer) {
```

```

        throw new Error("`importer` should not be undefined");
    }
    const basedir = importer ? path.dirname(importer) :
process.cwd();
    const fsPath = path.resolve(basedir, id);
    let res;
    if ((res = tryFsResolve(fsPath, options))) {
        return {
            id: res,
        };
    }
}

const tryFsResolve = (fsPath: string, options: any) => {
    let res;
    if ((res = tryResolveFile(fsPath, options))) {
        return res;
    }
    // 尝试添加后缀名获取文件
    for (const ext of options.extensions) {
        if (res = tryResolveFile(fsPath + ext, options)) {
            return res;
        }
    }
};

```

至此，整个main.tsx运行时处理就完成了，后续分析App.tsx的内部处理