

vite核心原理（六）—— 资源请求处理(下)

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。

一、App.tsx请求

我们紧接着上一个章节的内容继续分析App.tsx的处理流程，App.tsx中多了css 和svg 的引入。

```
// App.tsx
import { useState } from "react";
import logo from "./logo.svg";
import "./App.css";

function App() {
  const [count, setCount] = useState(0);

  return (
    <div className="App">
      <header className="App-header">
        <img className="App-logo" src={logo} alt="" />
        <p>Hello Vite + React</p>
        <p>
          <button type="button" onClick={() => setCount((count) => count
+ 1)}>
```

```

        count is: {count}
      </button>
    </p>
    <p>
      Edit <code>App</code> and save e test.
    </p>
    <p>
      <a
        className="App-link"
        href="https://reactjs.org"
        target="_blank"
        rel="noopener noreferrer"
      >
        Learn React
      </a>
      {" | "}
      <a
        className="App-link"
        href="https://vitejs.dev/guide/features.html"
        target="_blank"
        rel="noopener noreferrer"
      >
        Vite Docs
      </a>
    </p>
  </header>
</div>
);
}

export default App;

```

App.tsx的请求流程和之前的 main.tsx大致相同，只是在transform转换这一步略有不同。

首先是 esbuildPlugin的 transform，这一步纯粹是 esbuild 在处理，我们直接看一下返回结果：

```

import { jsx, jsxs } from "react/jsx-runtime";
import { useState } from "react";
import logo from "../logo.svg";
import "../App.css";
function App() {
  const [count, setCount] = useState(0);
  return /* #__PURE__ */ jsx("div", { className: "App", children: /*
  #__PURE__ */ jsxs("header", { className: "App-header", children: [
    /* #__PURE__ */ jsx("img", { className: "App-logo", src: logo, alt:
    "" }),
    /* #__PURE__ */ jsx("p", { children: "Hello Vite + React" }),
    /* #__PURE__ */ jsx("p", { children: /* #__PURE__ */ jsxs("button",
    { type: "button", onClick: () => setCount((count2) => count2 + 1),
    children: [
      "count is: ",
      count
    ] }) }),
    /* #__PURE__ */ jsxs("p", { children: [
      "Edit ",
      /* #__PURE__ */ jsx("code", { children: "App" }),
      " and save e test."
    ] }),
    /* #__PURE__ */ jsxs("p", { children: [
      /* #__PURE__ */ jsx(
        "a",
        {
          className: "App-link",
          href: "https://reactjs.org",
          target: "_blank",
          rel: "noopener noreferrer",
          children: "Learn React"
        }
      ),
      " | ",
      /* #__PURE__ */ jsx(
        "a",

```

```

    {
      className: "App-link",
      href: "https://vitejs.dev/guide/features.html",
      target: "_blank",
      rel: "noopener noreferrer",
      children: "Vite Docs"
    }
  )
] })
] }) });
}
var App_default = App;
export {
  App_default as default
};

```

紧接着会进入importAnalysisPlugin的 transform 逻辑

```

// node/plugins/importAnalysisPlugin.ts
async transform(code: string, importer: string) {
  // 只处理 JS 相关的请求
  if (!isJSRequest(importer) || isInternalRequest(importer)) return
  null;
  // 必须在parse前调用
  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();
if (importer.indexOf('App') !== -1) {
    console.log(imports);
}
// 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);
    /**
     * 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);
    // 静态资源
    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);
}

// 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
};
}

```

1. 首先依旧是先通过 parse 获取到 import 信息

```

// App.tsx
[
  { n: 'react/jsx-runtime', s: 27, e: 44, ss: 0, se: 45, d: -1, a:
-1 },
  { n: 'react', s: 73, e: 78, ss: 47, se: 79, d: -1, a: -1 },
  { n: './logo.svg', s: 99, e: 109, ss: 81, se: 110, d: -1, a: -1 },
  { n: './App.css', s: 120, e: 129, ss: 112, se: 130, d: -1, a: -1 }
]

```

2. 循环 imports 数组，对里面每一条import 信息进行处理。其中react/jsx-runtime之前已经分析过，这里对剩下三个进行分析：

- react:

路径解析

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

接下去会进行命名导入优化，将import {useState} from 'react'转换成了import __vite__ cjsImport1_react from "/node_modules/.m-vite/deps/react.js"; const useState = __vite__ cjsImport1_react["useState"]

```
// App.tsx
import __vite__cjsImport0_react_jsxRuntime from
"/node_modules/.m-vite/deps/react_jsx-runtime.js"; const jsx =
__vite__cjsImport0_react_jsxRuntime["jsx"]; const jsxs =
__vite__cjsImport0_react_jsxRuntime["jsxs"];
import __vite__cjsImport1_react from "/node_modules/.m-
vite/deps/react.js"; const useState =
__vite__cjsImport1_react["useState"];
// 省略其它代码
```

- ./logo.svg:

路径解析，这里的路径解析会给资源带上?import，这一步由markExplicitImport实现

```

// ./logo/svg
[
  '/src/logo.svg?import',
  '/Users/scout/Documents/frontEnd/vite-source-code/simplify-
vite/playground/src/logo.svg'
]

// markExplicitImport
// 对于一些静态资源, 比如获取svg, img, 会在请求后面加上 ?import 后缀
const markExplicitImport = (url: string) =>
isExplicitImportRequired(url) ? url + '?import' : url;

const isExplicitImportRequired = (url: string): boolean =>
!isJSRequest(cleanUrl(url)) && !isCSSRequest(url);

```

路径替换, 将./logo/svg替换成/src/logo.svg?import

```

// App.tsx
import __vite__cjsImport0_react_jsxRuntime from
"/node_modules/.m-vite/deps/react_jsx-runtime.js"; const jsx =
__vite__cjsImport0_react_jsxRuntime["jsx"]; const jsxs =
__vite__cjsImport0_react_jsxRuntime["jsxs"];
import __vite__cjsImport1_react from "/node_modules/.m-
vite/deps/react.js"; const useState =
__vite__cjsImport1_react["useState"];
import logo from "/src/logo.svg?import";
// 省略其它代码

```

■ ./App.css

路径解析

```

[
  '/src/App.css',
  '/Users/scout/Documents/frontEnd/vite-source-code/simplify-
vite/playground/src/App.css'
]

```


路径替换

```
// App.tsx
import __vite__cjsImport0_react_jsxRuntime from
"/node_modules/.m-vite/deps/react-jsx-runtime.js"; const jsx =
__vite__cjsImport0_react_jsxRuntime["jsx"]; const jsxs =
__vite__cjsImport0_react_jsxRuntime["jsxs"];
import __vite__cjsImport1_react from "/node_modules/.m-
vite/deps/react.js"; const useState =
__vite__cjsImport1_react["useState"];
import logo from "/src/logo.svg?import";
import "/src/App.css";
// 省略其它代码
```

至此，App.tsx的整个请求处理过程就结束。接下去就是发起其内部依赖文件的请求。

二、App.tsx内部资源请求

对于 js 类型的资源请求前面已经分析的差不多了，接下去需要分析css 和图片类的资源是如何被处理的。

1. logo.svg

vite 在处理静态资源时会使用assetPlugin，assetPlugin会将结果处理成默认导出的形式，导出的内容就是该资源的地址。

当通过import logo from '/src/logo.svg?import'时，这个 logo 其实就是 logo.svg的资源地址/src/logo.svg。当我们将 logo 变量用在 img 组件上的 src 属性时，才会去请求真正的svg图标

```
// node/plugins/assets.ts
export function assetPlugin(config: ResolvedConfig): Plugin {
  let serverContext: ServerContext;
  return {
    name: "m-vite:asset",
    configureServer(s) {
      serverContext = s;
    },
    async load(id) {
```

```

// 非静态资源类型直接返回
if (!config.assetsInclude(cleanUrl(id))) return;
const cleanedId = removeImportQuery(cleanUrl(id));
const resolvedId =
`${getShortName(normalizePath(cleanedId), serverContext.root)}`;
// 这里仅处理 svg
return {
  code: `export default "${resolvedId}"`,
};
},
};
}

```

Name	Method	Status	Protocol	Type	Initiator	Size	Time	Priority	C.	Waterfall	▲
localhost	GET	200	http/1.1	document	Other	622 B	5 ms	Highest	7..		
client	GET	200	http/1.1	script	(index)	2.0 kB	2 ms	High	7..		
main.tsx	GET	200	http/1.1	script	(index)	702 B	8 ms	High	7..		
react_jsx-runtime.js	GET	200	http/1.1	script	main.tsx:1	36.4 kB	3 ms	High	7..		
react-dom_client.js	GET	200	http/1.1	script	main.tsx:2	926 kB	19 ms	High	7..		
App.tsx	GET	200	http/1.1	script	main.tsx:3	1.9 kB	26 ms	High	7..		
localhost	GET	101	webso...	websocket	client:2	0 B	Pending				
chunk-ZLG4WENO.js	GET	200	http/1.1	script	react_jsx-...	78.0 kB	3 ms	High	7..		
react.js	GET	200	http/1.1	script	App.tsx:2	318 B	2 ms	High	7..		
logo.svg?import	GET	200	http/1.1	script	App.tsx:3	191 B	3 ms	High	7..		
App.css	GET	200	http/1.1	script	App.tsx:4	1.3 kB	4 ms	High	7..		
logo.svg	GET	200	http/1.1	svg+xml	react-do...	2.9 kB	3 ms	High	7..		

Name	x	Headers	Payload	Preview	Response	Initiator	Timing	Cookies
localhost	1				export default "/src/logo.svg"			
client								
main.tsx								
react_jsx-runtime.js								
react-dom_client.js								
App.tsx								
localhost								
chunk-ZLG4WENO.js								
react.js								
logo.svg?import								
App.css								
logo.svg								

2. App.css

vite 会借助cssPlugin对 css 资源进行处理。处理逻辑大致为将 css代码包装成 js代码返回，当浏览器加载完后执行 js 中的updateStyle创建 style 标签，并通过innerHTML的方式渲染 css 内容。

```

// node/plugins/css.ts
export function cssPlugin(): Plugin {
  let serverContext: ServerContext;
  return {
    name: "m-vite:css",
    configureServer(s) {
      serverContext = s;
    },
    load(id) {
      // 加载
      if (id.endsWith(".css")) {
        return readFile(id, "utf-8");
      }
    },
    // 转换逻辑
    transform(code, id) {
      if (id.endsWith(".css")) {
        // 包装成 JS 模块
        const jsContent = `
          import { createHotContext as
__vite__createHotContext } from "${CLIENT_PUBLIC_PATH}";
          import.meta.hot =
__vite__createHotContext("/${getShortName(id,
serverContext.root)}");

          import { updateStyle, removeStyle } from
"${CLIENT_PUBLIC_PATH}"

          const id = '${id}';
          const css = '${code.replace(/\n/g, "")}';

          updateStyle(id, css);
          import.meta.hot.accept();
          export default css;
          import.meta.hot.prune(() =>
removeStyle(id));`.trim();

```

```

        return {
            code: jsContent,
        };
    }
    return null;
},
};
};
}

// client/client.ts
export function updateStyle(id: string, content: string) {
    let style = sheetsMap.get(id);
    if (!style) {
        // 添加 style 标签
        style = document.createElement("style");
        style.setAttribute("type", "text/css");
        style.innerHTML = content;
        document.head.appendChild(style);
    } else {
        // 更新 style 标签内容
        style.innerHTML = content;
    }
    sheetsMap.set(id, style);
}
}

```

Name	×	Headers	Preview	Response	Initiator	Timing	Cookies
localhost	1			import { createHotContext as __vite__createHotContext } from "@vite/client";			
client	2			import.meta.hot = __vite__createHotContext("/src/App.css");			
main.tsx	3						
react_jsx-runtime.js	4			import { updateStyle, removeStyle } from "@vite/client"			
react-dom_client.js	5						
App.tsx	6			const id = '/Users/scout/Documents/frontEnd/vite-source-code/simplify-vite			
localhost	7			const css = '.App { text-align: center;}.App-logo { height: 40vmin; po			
chunk-ZLG4WENO.js	8						
react.js	9			updateStyle(id, css);			
logo.svg?import	10			import.meta.hot.accept();			
App.css	11			export default css;			
logo.svg	12			import.meta.hot.prune(() => removeStyle(id));			
12 requests		1.1 MB tra		{}			

以上就是App.tsx 中其它资源的加载处理。

至此，我们已经将整个应用的资源请求加载过程大致捋了一遍，包括 html、js、css、svg 的处理，这些资源都是由对应的插件负责处理。毫无疑问，插件体系是整个 vite 架构中非常重要的部分，这些插件能够帮助我们处理各种资源或者扩展各种功能。

三、资源请求前的一些事儿

回顾预构建章节，我们曾提到，vite 在第一次启动服务时会进行初步的预构建，然后将需要预构建的资源进行处理后输出到缓存目录中。不知大家是否还记得，在资源输出的时候，有一个临时缓存目录，叫做processingCacheDir (/node_modules/vite/deps_temp)。在第一次进行预构建时，所有的资源只会存在deps_temp中，此时还不存在真正的缓存目录cacheDir (/node_modules/vite/deps)。但是，在前面资源请求部分我们讲到，所有预构建资源的请求地址最终都指向了 cacheDir。但是之前又不存在cacheDir，那么这些资源是如何获取到的？

经过前面的讲解，我们知道资源内容是通过插件的 load 钩子获取的，因此，在读取预构建资源之前我们需要等待资源生成到cacheDir目录中，这样才能保证读取到内容。这块等待的逻辑在optimizedDepsPlugin的 load 钩子中实现：

1. 判断资源是否是预构建资源，如果是则根据依赖优化器对象中的metadata获取到资源的构建信息。optimizedDepInfoFromFile方法实际上就是遍历 metadata 上的depInfoList，匹配到对应 id 的资源信息
2. 等待info.processing执行完成，这个processing 是一个 promise。在预构建章节中曾提到，vite 会将发现的预构建资源信息存储到 metadata上的discovered属性中。这个过程会调用addMissingDep方法往预构建资源对象中添加processing属性，值为 depOptimizationProcessing.promise。这个depOptimizationProcessing 就是之前提到的预构建任务处理对象，这个对象包含一个 promise 还有一个 resolve 方法。也就是说，等待info.processing就是在等待depOptimizationProcessing的resolve方法被调用

```
// node/optimizer/optimizer.ts
depsOptimizer.scanProcessing = new Promise(resolve => {
  setTimeout(async () => {
    try {
      deps = await
discoverProjectDependencies(config);
      // 添加缺失的依赖到 metadata.discovered 中
```

```

        for (const id of Object.keys(deps)) {
            if (!metadata.discovered[id]) {
                addMissingDep(id, deps[id]);
            }
        }
        const knownDeps = prepareKnownDeps();
        postScanOptimizationResult =
runOptimizeDeps(config, knownDeps);
        } catch (error) {

    }
    finally {
        resolve();
        depsOptimizer.scanProcessing = undefined;
    }
});
});

function addMissingDep(id: string, resolved: string) {
    return addOptimizedDepInfo(metadata, 'discovered', {
        id,
        file: getOptimizedDepPath(id, config),
        src: resolved,
        processing: depOptimizationProcessing.promise,
        exportsData: extractExportsData(resolved, config),
    });
}

```

```

// node/server/transformRequest.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/index.ts
export function optimizedDepInfoFromFile(
    metadata: DepOptimizationMetadata,
    file: string,
): OptimizedDepInfo | undefined {
    return metadata.depInfoList.find((depInfo) => depInfo.file === file)
}

```

因此，为了让预构建资源能正常被获取到，这个 resolve 就必须在预构建资源生成到缓存目录 cacheDir 后再调用。之前提过，在第一次启动服务时会执行一次预构建，此时只是将资源构建到临时目录中。程序运行时也可能会发起多次预构建任务，因为部分依赖是在运行时发现的。知道这点后，我们就要知道运行时的预构建是何时发生的。

我们回到doTransform方法中，这个方法在之前介绍资源转换时有提及，我们注意最后一段代码 getDepsOptimizer(config)?.delayDepsOptimizerUntil(id, () => transformResult)。getDepsOptimizer是根据配置获取依赖优化器对象，在依赖优化器对象上有一个delayDepsOptimizerUntil方法。

delayDepsOptimizerUntil方法的作用：延迟依赖优化直到满足条件。这个满足条件在第一次运行时指所有非预构建依赖加载完毕。

seenIds：用来判断资源是否已经被处理过

registeredIds：用来记录非预构建的依赖对象（main.tsx、App.tsx等），这个对象包含其资源id 已经 done 函数，这个 done 函数就是() => transformResult，transformResult是执行loadAndTransform方法后返回的 promise 对象

```
// node/server/transformRequest.ts
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的依赖
  getDepsOptimizer(config)?.delayDepsOptimizerUntil(id, () =>
transformResult);
  return transformResult;
};
```



```
// node/optimizer/optimizer.ts
function delayDepsOptimizerUntil(id: string, done: () => Promise<any>):
void {
    // 运行过程中发现的依赖，非预构建阶段发现的依赖
    if (!depsOptimizer.isOptimizedDepFile(id) && !seenIds.has(id)) {
        seenIds.add(id)
        registeredIds.push({ id, done })
        // 空闲时进行优化处理
        runOptimizerWhenIdle();
    }
}
```

在delayDepsOptimizerUntil中会调用runOptimizerWhenIdle发起运行时的预构建任务。runOptimizerWhenIdle中涉及到的waitingOn，定时器等是为了控制预构建执行的频率。频繁得进行预构建会阻塞资源的加载，浏览器将长时间处于 loading 状态。当registeredIds中的资源都加载完毕后，就会执行onCrawlEnd方法处理预构建缓存。

```
// node/optimizer/optimizer.ts
// 空闲时进行优化处理
function runOptimizerWhenIdle() {
    // 是否已经有在进行处理的依赖
    if (!waitingOn) {
        const next = registeredIds.pop();
        if (next) {
            waitingOn = next.id;
            const afterLoad = () => {
                waitingOn = undefined;
                registeredIds.length > 0 ? runOptimizerWhenIdle() :
onCrawlEnd();
            };
            next
                .done()
                .then(
                    () => setTimeout(
                        afterLoad,
                        registeredIds.length > 0 ? 0 : 100,

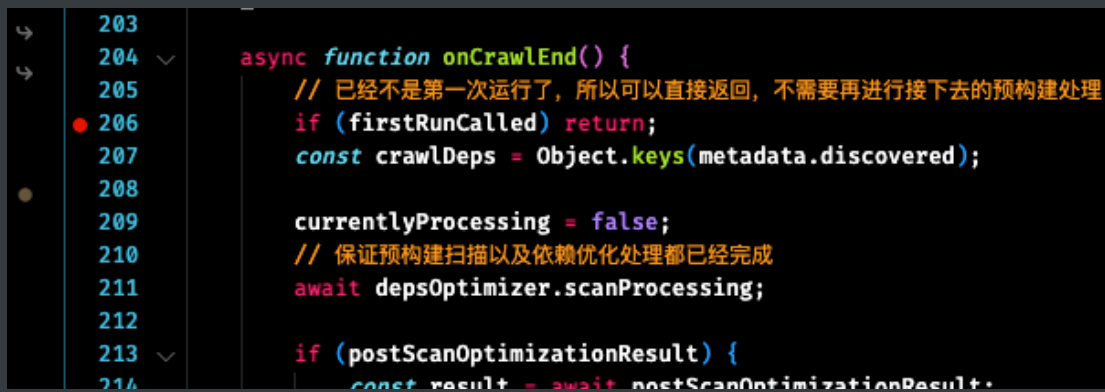
```

```

    )
  )
  .catch(afterLoad);
}
}
}

```

我们在onCrawlEnd函数里面打上断点，看看浏览器请求的状态：

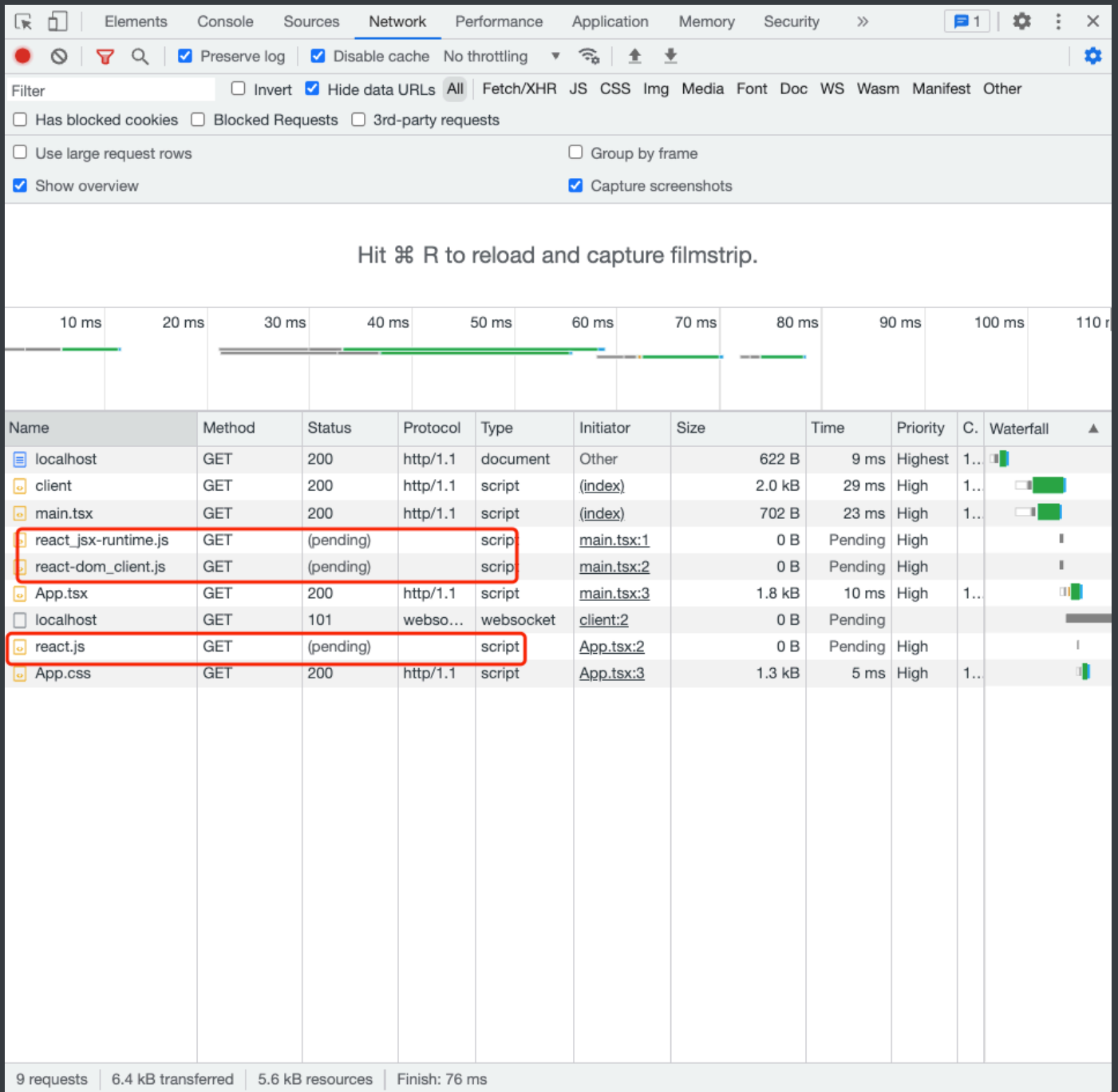


```

203
204
205
206 ● async function onCrawlEnd() {
207   // 已经不是第一次运行了，所以可以直接返回，不需要再进行接下去的预构建处理
208   if (firstRunCalled) return;
209   const crawlDeps = Object.keys(metadata.discovered);
210
211   currentlyProcessing = false;
212   // 保证预构建扫描以及依赖优化处理都已经完成
213   await depsOptimizer.scanProcessing;
214
215   if (postScanOptimizationResult) {
216     const result = await postScanOptimizationResult;

```

图上很明显能看到，当非预构建资源全部加载完毕时，预构建资源（react_jsx-runtime、react-dom_client、react.js）全部处于 pending 状态。



接下去就是onCrawlEnd的处理逻辑：

1. 判断是否是第一次运行，已经运行过了就不需要再进行接下去的预构建处理
2. await depsOptimizer.scanProcessing 等待第一次启动服务时预构建程序执行完成
3. 判断postScanOptimizationResult是否存在，postScanOptimizationResult是第一次启动服务时执行预构建优化返回的对象（预构建章节提到的processingResult对象），如果存在则说明是第一次启动服务并运行程序，后续执行步骤4-6，否则执行步骤 7
4. await postScanOptimizationResul 获取processingResult对象

5. 根据 metadata 信息获取新发现的依赖信息 (crawlDeps) 以及已经扫描优化过的依赖信息 (scanDeps)，如果两者都没有，则说明没有需要预构建的依赖，此时需要调用 processingResult 的 cancel 方法将预构建缓存目录删除并结束流程
6. 比较 crawlDeps 和 scanDeps，查找是否有缺失依赖，也就是没有进行过预构建的依赖。如果有缺失的依赖需要构建，则会将之前预构建扫描的结果删除并执行 debouncedProcessing，否则直接调用 runOptimizer。debouncedProcessing 实际上就是加了防抖的 runOptimizer，也是为了避免多次执行。相对来说，如果运行时发现了缺失的依赖则有可能会有其它缺失的依赖后续被发现，比如当前缺失的依赖内部又动态依赖其它内容，因此需要加上防抖处理。如果没有发现依赖，则直接执行 runOptimizer 即可
7. 不是第一次执行程序，也就是之前启动运行过了，那么只需要判断是否发现了新的依赖，如果有则同步骤 6 执行 debouncedProcessing，没有则结束

```
// node/optimizer/optimizer.ts
async function onCrawlEnd() {
    // 已经不是第一次运行了，所以可以直接返回，不需要再进行接下去的预构建处理
    if (firstRunCalled) return;
    const crawlDeps = Object.keys(metadata.discovered);

    currentlyProcessing = false;
    // 保证第一次预构建程序执行完成
    await depsOptimizer.scanProcessing;

    if (postScanOptimizationResult) {
        const result = await postScanOptimizationResult;
        postScanOptimizationResult = undefined;
        const scanDeps = Object.keys(result.metadata.optimized);

        /**
         * 这种情况针对第一次进行预构建运行并且预构建中没有需要预构建的依赖的情况
         * 这种情况下需要把预构建创建的临时目录给删除并标记firstRunCalled为true
         */
        if (scanDeps.length === 0 && crawlDeps.length === 0) {
            result.cancel();
        }
    }
}
```

```

        firstRunCalled = true;
        return;
    }

    // 判断是否有缺失的依赖，如果有缺失的新依赖，则需要重新进行预构建处理
    const scannerMissedDeps = crawlDeps.some((dep) =>
!scanDeps.includes(dep));
    const outdatedResult = scannerMissedDeps;
    if (outdatedResult) {
        // 删除此扫描结果，并执行新的优化以避免完全重新加载
        result.cancel();
        // 重新进行预构建优化
        for (const dep of scanDeps) {
            if (!crawlDeps.includes(dep)) {
                addMissingDep(dep,
result.metadata.optimized[dep].src);
            }
        }
        debouncedProcessing(0);
    } else {
        runOptimizer(result);
    }
} else {
    // 没有发现需要优化的新依赖
    if (!crawlDeps.length) {
        console.log(
            green(
                `✨ no dependencies found while crawling the
static imports`,
            ),
        );
        firstRunCalled = true;
    } else {
        debouncedProcessing(0);
    }
}
}

```

```
}
```

最后到了关键的runOptimizer方法的实现，这个方法的核心部分就是调用commitProcessing方法：

1. 调用processingResult.commit将预构建缓存临时目录替换成真实目录
2. 更新已发现的依赖信息
3. 调用resolveEnqueuedProcessingPromises方法，执行depOptimizationProcessing的resolve方法，将promise置为成功状态

```
// node/optimizer/optimizer.ts
async function runOptimizer(preRunResult?: any) {
  // 确保不会为当前发现的 deps 发出重新运行
  if (handle) clearTimeout(handle);
  const processingResult = preRunResult ?? (await optimizeNewDeps());
  const newData = processingResult.metadata;
  const needsInteropMismatch =
    findInteropMatches(metadata.discovered, newData.optimized);
  const needsReload = needsInteropMismatch.length > 0
  // 预构建依赖优化处理完成，更新依赖缓存
  const commitProcessing = async () => {
    await processingResult.commit();
    // 更新已发现的依赖信息
    for (const o in newData.optimized) {
      const discovered = metadata.discovered[o];
      if (discovered) {
        const optimized = newData.optimized[o];
        discovered.needsInterop = optimized.needsInterop;
        discovered.processing = undefined
      }
    }
  }
  // 执行所有的预构建处理进行，将内部的promise都resolve掉
  resolveEnqueuedProcessingPromises();
}
if (!needsReload) {
```

```

        await commitProcessing();
    } else {
        throw new Error('needsReload');
    }
}

// node/optimizer/index.ts
const processingResult = {
    metadata,
    async commit() {
        // 写入元数据文件，删除 `deps` 文件夹并将 `processing` 文件夹重命名
        // 为 `deps` 处理完成，
        // 我们现在可以用 depsCacheDir 替换 processingCacheDir 将文件路径
        // 从临时处理目录重新连接到最终的 deps 缓存目录
        await removeDir(depsCacheDir);
        await renameDir(processingCacheDir, depsCacheDir);
    },
    cancel() {
        // 取消预构建，删除预构建临时目录
        fs.rmSync(processingCacheDir, { recursive: true, force: true });
    }
};

```

这下终于确定了只有当预构建完成并更新临时缓存目录为真实目录后才会调用 `depOptimizationProcessing.resolve` 方法，后续执行 `await info.processing` 后面的逻辑，去读取预构建的资源内容并返回

四、结束

vite 的整个资源请求处理过程就结束了，这里只是简单地对整个过程进行了分析，不过这个过程也足以让我们对 vite 有一个更加深入的认知