



手写 Vite: 实现 no-bundle 开发服务(上)

发布于 2022-05-09

在上一章中，我们一起系统学习了 Vite 的实现源码，从配置解析、依赖预构建、插件流水线 and HMR 这几个方面带你完整的梳理了 Vite 的底层原理，那么，在本小节中，我们将进一步，用实际的代码来写一个迷你版的 Vite，主要实现 Vite 最核心的 no-bundle 构建服务。在学完本节之后，你不仅能够复习之前所介绍的各种原理，也能深入地理解代码层面的实现细节，拥有独立开发一个 no-bundle 构建工具的能力。

实战概览

相较于前面的小节，本小节(以及下一小节)的内容会比较难，手写的代码量也比较多(总共近一千行)。因此，在开始代码实战之前，我先给大家梳理一下需要完成的模块和功能，让大家有一个整体的认知：

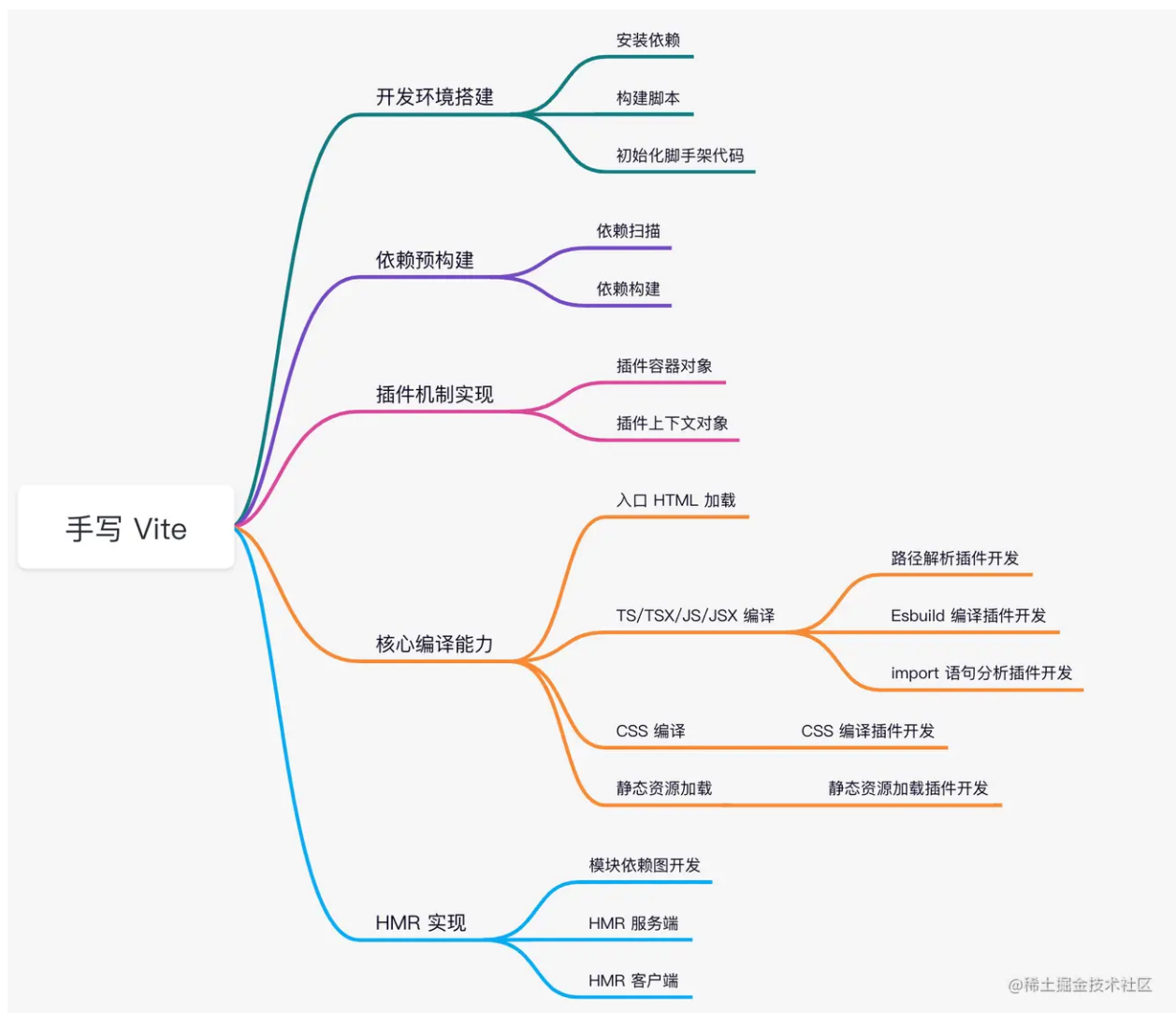
首先，我们会进行开发环境的搭建，安装必要的依赖，并搭建项目的构建脚本，同时完成 cli 工具的初始化代码。

然后我们正式开始实现 **依赖预构建** 的功能，通过 Esbuild 实现依赖扫描和依赖构建的功能。

接着开始搭建 Vite 的插件机制，也就是开发 **PluginContainer** 和 **PluginContext** 两个主要的对象。

搭建完插件机制之后，我们将会开发一系列的插件来实现 no-bundle 服务的编译构建能力，包括入口 HTML 处理、TS/TSX/JS/TSX 编译、CSS 编译和静态资源处理。

最后，我们会实现一套系统化的模块热更新的能力，从搭建模块依赖图开始，逐步实现 HMR 服务端和客户端的开发。



搭建开发环境

注: 手写 Vite 项目的所有代码, 我已经放到了小册的 Github 仓库中, [点击查看](#)。

首先, 你可以执行 `pnpm init -y` 来初始化项目, 然后安装一些必要的依赖, 执行命令如下:

对于各个依赖的具体作用, 大家先不用纠结, 我将会在后面使用到依赖的时候介绍。

```
// 运行时依赖
pnpm i cac chokidar connect debug es-module-lexer esbuild fs-extra magic-string picocolors re

// 开发环境依赖
pnpm i @types/connect @types/debug @types/fs-extra @types/resolve @types/ws tsup
```

Vite 本身使用的是 Rollup 进行自身的打包，但之前给大家介绍的 tsup 也能够实现库打包的功能，并且内置 esbuild 进行提速，性能上更加强悍，因此在这里我们使用 tsup 进行项目的构建。

为了接入 tsup 打包功能，你需要在 package.json 中加入这些命令：

```
"scripts": {  
  "start": "tsup --watch",  
  "build": "tsup --minify"  
},
```

同时，你需要在项目根目录新建 `tsconfig.json` 和 `tsup.config.ts` 这两份配置文件，内容分别如下：

```
// tsconfig.json  
{  
  "compilerOptions": {  
    // 支持 commonjs 模块的 default import, 如 import path from 'path'  
    // 否则只能通过 import * as path from 'path' 进行导入  
    "esModuleInterop": true,  
    "target": "ES2020",  
    "moduleResolution": "node",  
    "module": "ES2020",  
    "strict": true  
  }  
}
```

```
// tsup.config.ts  
import { defineConfig } from "tsup";  
  
export default defineConfig({  
  // 后续会增加 entry  
  entry: {  
    index: "src/node/cli.ts",  
  },  
  // 产物格式，包含 esm 和 cjs 格式  
  format: ["esm", "cjs"],  
  // 目标语法  
  target: "es2020",  
  // 生成 sourcemap  
  sourcemap: true,  
  // 没有拆包的需求，关闭拆包能力  
  splitting: false,  
});
```

接着新建 `src/node/cli.ts` 文件，我们进行 cli 的初始化：

```
// src/node/cli.ts
import cac from "cac";

const cli = cac();

// [] 中的内容为可选参数，也就是说仅输入 `vite` 命令下会执行下面的逻辑
cli
  .command("[root]", "Run the development server")
  .alias("serve")
  .alias("dev")
  .action(async () => {
    console.log('测试 cli~');
  });

cli.help();

cli.parse();
```

现在你可以执行 `pnpm start` 来编译这个 `mini-vite` 项目，`tsup` 会生成产物目录 `dist`，然后你可以新建 `bin/mini-vite` 文件来引用产物：

```
#!/usr/bin/env node

require("../dist/index.js");
```

同时，你需要在 `package.json` 中注册 `mini-vite` 命令，配置如下：

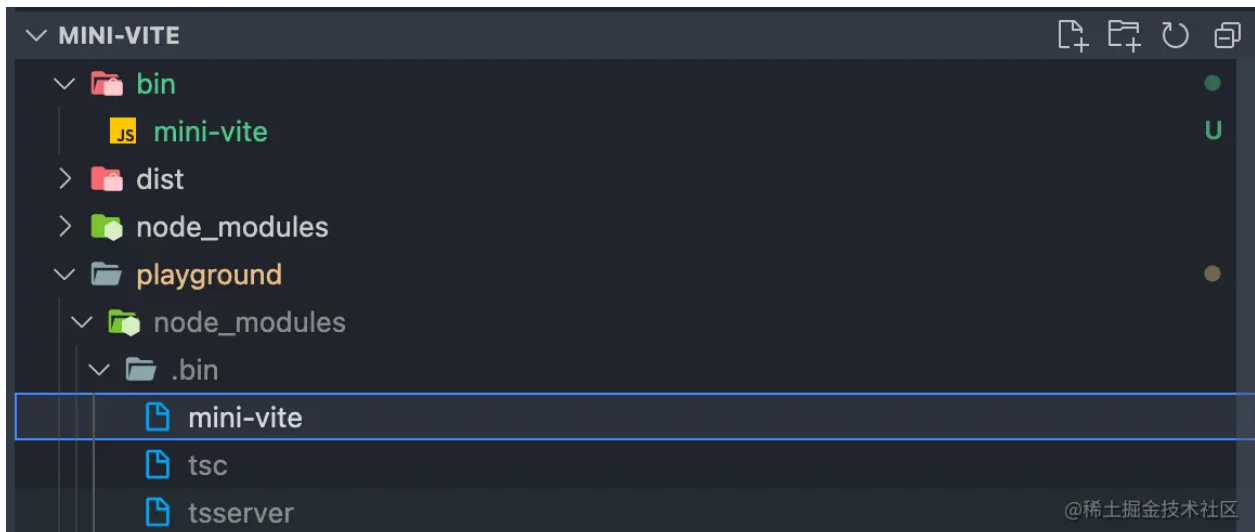
```
{
  "bin": {
    "mini-vite": "bin/mini-vite"
  }
}
```

如此一来，我们就可以在业务项目中使用 `mini-vite` 这个命令了。在小册的 Github 仓库中我为你准备了一个示例的 `playground` 项目，你可以拿来进行测试，[点击查看项目](#)。

将 `playground` 项目放在 `mini-vite` 目录中，然后执行 `pnpm i`，由于项目的 `dependencies` 中已经声明了 `mini-vite`：

```
{
  "devDependencies": {
    "mini-vite": '../'
  }
}
```

那么 `mini-vite` 命令会自动安装到测试项目的 `node_modules/.bin` 目录中:



接着我们在 `playground` 项目中执行 `pnpm dev` 命令(内部执行 `mini-vite`), 可以看到如下的 log 信息:

```
测试 cli~
```

接着, 我们把 `console.log` 语句换成服务启动的逻辑:

```
import cac from "cac";
+ import { startDevServer } from "./server";

const cli = cac();

cli
  .command("[root]", "Run the development server")
  .alias("serve")
  .alias("dev")
  .action(async () => {
-   console.log('测试 cli~');
+   await startDevServer();
  });
```

现在你需要新建 `src/node/server/index.ts` , 内容如下:

```
// connect 是一个具有中间件机制的轻量级 Node.js 框架。
// 既可以单独作为服务器, 也可以接入到任何具有中间件机制的框架中, 如 Koa、Express
import connect from "connect";
// picocolors 是一个用来在命令行显示不同颜色文本的工具
import { blue, green } from "picocolors";

export async function startDevServer() {
  const app = connect();
  const root = process.cwd();
```

```

const startTime = Date.now();
app.listen(3000, async () => {
  console.log(
    green("🚀 No-Bundle 服务已经成功启动!"),
    `耗时: ${Date.now() - startTime}ms`
  );
  console.log(`> 本地访问路径: ${blue("http://localhost:3000")}`);
});
}

```

再次执行 `pnpm dev`，你可以发现终端出现如下的启动日志:



OK, `mini-vite` 的 cli 功能和服务启动的逻辑目前就已经成功搭建起来了。

依赖预构建

现在我们来进入依赖预构建阶段的开发。

首先我们新建 `src/node/optimizer/index.ts` 来存放依赖预构建的逻辑:

```

export async function optimize(root: string) {
  // 1. 确定入口
  // 2. 从入口处扫描依赖
  // 3. 预构建依赖
}

```

然后在服务入口中引入预构建的逻辑:

```

// src/node/server/index.ts
import connect from "connect";
import { blue, green } from "picocolors";
+ import { optimize } from "../optimizer/index";

export async function startDevServer() {
  const app = connect();
  const root = process.cwd();
  const startTime = Date.now();
  app.listen(3000, async () => {
+   await optimize(root);

    console.log(
      green("🚀 No-Bundle 服务已经成功启动!"),
      `耗时: ${Date.now() - startTime}ms`
    );
  });
}

```

```

    );
    console.log(`> 本地访问路径: ${blue("http://localhost:3000")}`);
  });
}

```

接着我们来开发依赖预构建的功能，从上面的代码注释你也可以看出，我们需要完成三部分的逻辑：

- 确定预构建入口
- 从入口开始扫描出用到的依赖
- 对依赖进行预构建

首先是确定入口，为了方便理解，这里我直接约定为 `src` 目录下的 `main.tsx` 文件：

```

// 需要引入的依赖
import path from "path";

// 1. 确定入口
const entry = path.resolve(root, "src/main.tsx");

```

第二步是扫描依赖：

```

// 需要引入的依赖
import { build } from "esbuild";
import { green } from "picocolors";
import { scanPlugin } from "./scanPlugin";

// 2. 从入口处扫描依赖
const deps = new Set<string>();
await build({
  entryPoints: [entry],
  bundle: true,
  write: false,
  plugins: [scanPlugin(deps)],
});
console.log(
  `${green("需要预构建的依赖")}: \n${[...deps]
    .map(green)
    .map((item) => `  ${item}`)
    .join("\n")}`
);

```

依赖扫描需要我们借助 Esbuild 插件来完成，最后会记录到 `deps` 这个集合中。接下来我们着眼于 Esbuild 依赖扫描插件的开发，你需要在 `optimzier` 目录中新建 `scanPlguin.ts` 文件，内容如下：

```
// src/node/optimizer/scanPlugin.ts
import { Plugin } from "esbuild";
import { BARE_IMPORT_RE, EXTERNAL_TYPES } from "../constants";

export function scanPlugin(deps: Set<string>): Plugin {
  return {
    name: "esbuild:scan-deps",
    setup(build) {
      // 忽略的文件类型
      build.onResolve(
        { filter: new RegExp(`\\.${EXTERNAL_TYPES.join("|")}$`) },
        (resolveInfo) => {
          return {
            path: resolveInfo.path,
            // 打上 external 标记
            external: true,
          };
        }
      );
      // 记录依赖
      build.onResolve(
        {
          filter: BARE_IMPORT_RE,
        },
        (resolveInfo) => {
          const { path: id } = resolveInfo;
          // 推入 deps 集合中
          deps.add(id);
          return {
            path: id,
            external: true,
          };
        }
      );
    },
  };
}
```

需要说明的是，文件中用到了一些常量，在 `src/node/constants.ts` 中定义，内容如下：

```
export const EXTERNAL_TYPES = [
  "css",
  "less",
  "sass",
  "scss",
  "styl",
  "stylus",
  "pcss",
  "postcss",
  "vue",
  "svelte",
  "marko",
  "astro",
  "png",
  "jpe?g",

```



```

    "gif",
    "svg",
    "ico",
    "webp",
    "avif",
  ];

  export const BARE_IMPORT_RE = /^[\w@][^:]/;

```

插件的逻辑非常简单，即把一些无关的资源进行 external，不让 esbuild 处理，防止 Esbuild 报错，同时将 bare import 的路径视作第三方包，推入 deps 集合中。

现在，我们在 playground 项目根路径中执行 pnpm dev，可以发现依赖扫描已经成功执行：

当我们收集到所有的依赖信息之后，就可以对每个依赖进行打包，完成依赖预构建了：

```

// src/node/optimizer/index.ts
// 需要引入的依赖
import { preBundlePlugin } from "../preBundlePlugin";
import { PRE_BUNDLE_DIR } from "../constants";

// 3. 预构建依赖
await build({
  entryPoints: [...deps],
  write: true,
  bundle: true,
  format: "esm",
  splitting: true,
  outdir: path.resolve(root, PRE_BUNDLE_DIR),
  plugins: [preBundlePlugin(deps)],
});

```

在此，我们引入了一个新的常量 PRE_BUNDLE_DIR，定义如下：

```

// src/node/constants.ts
// 增加如下代码
import path from "path";

// 预构建产物默认存放在 node_modules 中的 .m-vite 目录中
export const PRE_BUNDLE_DIR = path.join("node_modules", ".m-vite");

```

接着，我们继续开发预构建的 Esbuild 插件：

```

import { Loader, Plugin } from "esbuild";
import { BARE_IMPORT_RE } from "../constants";
// 用来分析 es 模块 import/export 语句的库
import { init, parse } from "es-module-lexer";
import path from "path";
// 一个实现了 node 路径解析算法的库
import resolve from "resolve";
// 一个更加好用的文件操作库
import fs from "fs-extra";
// 用来开发打印 debug 日志的库
import createDebug from "debug";

const debug = createDebug("dev");

export function preBundlePlugin(deps: Set<string>): Plugin {
  return {
    name: "esbuild:pre-bundle",
    setup(build) {
      build.onResolve(
        {
          filter: BARE_IMPORT_RE,
        },
        (resolveInfo) => {
          const { path: id, importer } = resolveInfo;
          const isEntry = !importer;
          // 命中需要预编译的依赖
          if (deps.has(id)) {
            // 若为入口，则标记 dep 的 namespace
            return isEntry
              ? {
                  path: id,
                  namespace: "dep",
                }
              : {
                  // 因为走到 onResolve 了，所以这里的 path 就是绝对路径了
                  path: resolve.sync(id, { basedir: process.cwd() }),
                };
          }
        }
      );
    }
  };
}

// 拿到标记后的依赖，构造代理模块，交给 esbuild 打包
build.onLoad(
  {
    filter: /\.*/,
    namespace: "dep",
  },
  async (loadInfo) => {
    await init;
    const id = loadInfo.path;
    const root = process.cwd();
    const entryPath = resolve.sync(id, { basedir: root });
    const code = await fs.readFile(entryPath, "utf-8");
    const [imports, exports] = await parse(code);
    let proxyModule = [];
    // cjs
    if (!imports.length && !exports.length) {

```

```

// 构造代理模块
// 下面的代码后面会解释
const res = require(entryPath);
const specifiers = Object.keys(res);
proxyModule.push(
  `export { ${specifiers.join(",")} } from "${entryPath}"`,
  `export default require("${entryPath}")`
);
} else {
  // esm 格式比较好处理, export * 或者 export default 即可
  if (exports.includes("default")) {
    proxyModule.push(`import d from "${entryPath}";export default d`);
  }
  proxyModule.push(`export * from "${entryPath}"`);
}
debug("代理模块内容: %o", proxyModule.join("\n"));
const loader = path.extname(entryPath).slice(1);
return {
  loader: loader as Loader,
  contents: proxyModule.join("\n"),
  resolveDir: root,
};
}
);
},
};
}

```

值得一提的是, 对于 CommonJS 格式的依赖, 单纯用 `export default require('入口路径')` 是有局限性的, 比如对于 React 而言, 用这样的方式生成的产物最后只有 default 导出:


```

// esbuild 的打包产物
// 省略大部分代码
export default react_default;


```

那么用户在使用这个依赖的时候, 必须这么使用:

```

//  正确
import React from 'react';

const { useState } = React;

//  报错
import { useState } from 'react';

```

那为什么上述会报错的语法在 Vite 是可以正常使用的呢? 原因是 Vite 在做 import 语句分析的时候, 自动将你的代码进行改写了:

```
// 原来的写法
import { useState } from 'react';

// Vite 的 importAnalysis 插件转换后的写法类似下面这样
import react_default from '/node_modules/.vite/react.js';

const { useState } = react_default;
```

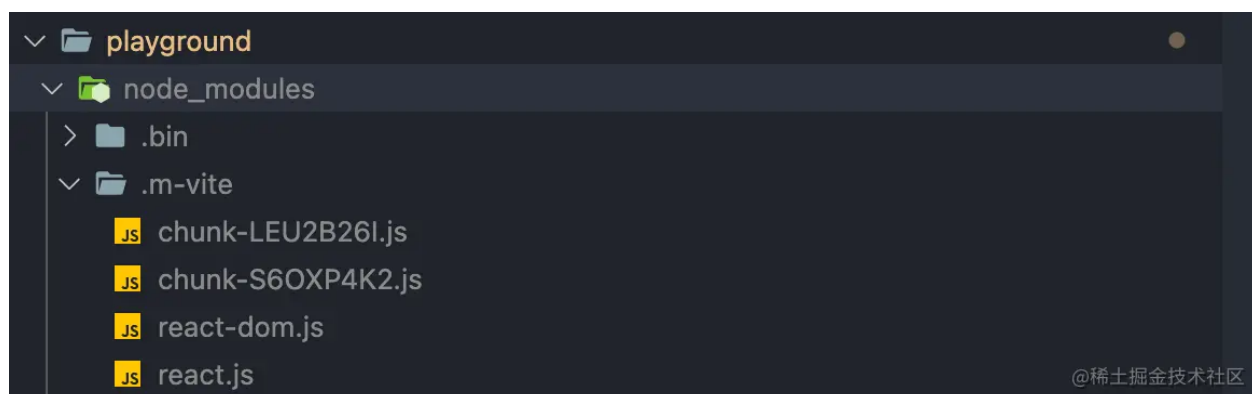
那么，还有没有别的方案来解决这个问题？没错，上述的插件代码中已经用另一个方案解决了这个问题，我们不妨把目光集中在下面这段代码中：

```
if (!imports.length && !exports.length) {
  // 构造代理模块
  // 通过 require 拿到模块的导出对象
  const res = require(entryPath);
  // 用 Object.keys 拿到所有的具名导出
  const specifiers = Object.keys(res);
  // 构造 export 语句交给 Esbuild 打包
  proxyModule.push(
    `export { ${specifiers.join(",")} } from "${entryPath}"`,
    `export default require("${entryPath}")`
  );
}
```

如此一来，Esbuild 预构建的产物中便会包含 CommonJS 模块中所有的导出信息：

```
// 预构建产物导出代码
export {
  react_default as default,
  useState,
  useEffect,
  // 省略其它导出
}
```

OK，接下来让我们来测试一下预构建整体的功能。在 `playground` 项目中执行 `pnpm dev`，接着去项目的 `node_modules` 目录中，可以发现新增了 `.m-vite` 目录及 `react`、`react-dom` 的预构建产物：



插件机制开发

在完成了依赖预构建的功能之后，我们开始搭建 Vite 的插件机制，实现插件容器和插件上下文对象。

首先，你可以新建 `src/node/pluginContainer.ts` 文件，增加如下的类型定义：

```
import type {
  LoadResult,
  PartialResolvedId,
  SourceDescription,
  PluginContext as RollupPluginContext,
  ResolvedId,
} from "rollup";

export interface PluginContainer {
  resolveId(id: string, importer?: string): Promise<PartialResolvedId | null>;
  load(id: string): Promise<LoadResult | null>;
  transform(code: string, id: string): Promise<SourceDescription | null>;
}
```

另外，由于插件容器需要接收 Vite 插件作为初始化参数，因此我们需要提前声明插件的类型，你可以继续新建 `src/node/plugin.ts` 来声明如下的插件类型：

```
import { LoadResult, PartialResolvedId, SourceDescription } from "rollup";
import { ServerContext } from "../server";

export type ServerHook = (
  server: ServerContext
) => (() => void) | void | Promise<(() => void) | void>;

// 只实现以下这几个钩子
export interface Plugin {
  name: string;
  configureServer?: ServerHook;
  resolveId?: (
    id: string,
    importer?: string
  ) => Promise<PartialResolvedId | null> | PartialResolvedId | null;
  load?: (id: string) => Promise<LoadResult | null> | LoadResult | null;
  transform?: (
    code: string,
    id: string
  ) => Promise<SourceDescription | null> | SourceDescription | null;
  transformIndexHtml?: (raw: string) => Promise<string> | string;
}
```

对于其中的 `ServerContext`，你暂时不用过于关心，只需要在 `server/index.ts` 中简单声明一下类型即可：

```
// src/node/server/index.ts
// 增加如下类型声明
export interface ServerContext {}
```

接着，我们来实现插件机制的具体逻辑，主要集中在 `createPluginContainer` 函数中：

```
// src/node/pluginContainer.ts
// 模拟 Rollup 的插件机制
export const createPluginContainer = (plugins: Plugin[]): PluginContainer => {
  // 插件上下文对象
  // @ts-ignore 这里仅实现上下文对象的 resolve 方法
  class Context implements RollupPluginContext {
    async resolve(id: string, importer?: string) {
      let out = await pluginContainer.resolveId(id, importer);
      if (typeof out === "string") out = { id: out };
      return out as ResolvedId | null;
    }
  }
  // 插件容器
  const pluginContainer: PluginContainer = {
    async resolveId(id: string, importer?: string) {
      const ctx = new Context() as any;
      for (const plugin of plugins) {
        if (plugin.resolveId) {
          const newId = await plugin.resolveId.call(ctx as any, id, importer);
          if (newId) {
            id = typeof newId === "string" ? newId : newId.id;
            return { id };
          }
        }
      }
      return null;
    },
    async load(id) {
      const ctx = new Context() as any;
      for (const plugin of plugins) {
        if (plugin.load) {
          const result = await plugin.load.call(ctx, id);
          if (result) {
            return result;
          }
        }
      }
      return null;
    },
    async transform(code, id) {
      const ctx = new Context() as any;
      for (const plugin of plugins) {
        if (plugin.transform) {
          const result = await plugin.transform.call(ctx, code, id);

```

```

        if (!result) continue;
        if (typeof result === "string") {
            code = result;
        } else if (result.code) {
            code = result.code;
        }
    }
}
return { code };
},
};

return pluginContainer;
};

```

上面的代码比较容易理解，并且关于插件钩子的执行原理和插件上下文对象的作用，在小册第 22 节中也有详细的分析，这里就不再赘述了。

接着，我们来完善一下之前的服务器逻辑：

```

// src/node/server/index.ts
import connect from "connect";
import { blue, green } from "picocolors";
import { optimize } from "../optimizer/index";
+ import { resolvePlugins } from "../plugins";
+ import { createPluginContainer, PluginContainer } from "../pluginContainer";

export interface ServerContext {
+ root: string;
+ pluginContainer: PluginContainer;
+ app: connect.Server;
+ plugins: Plugin[];
}

export async function startDevServer() {
    const app = connect();
    const root = process.cwd();
    const startTime = Date.now();
+ const plugins = resolvePlugins();
+ const pluginContainer = createPluginContainer(plugins);

+ const serverContext: ServerContext = {
+     root: process.cwd(),
+     app,
+     pluginContainer,
+     plugins,
+ };

+ for (const plugin of plugins) {
+     if (plugin.configureServer) {
+         await plugin.configureServer(serverContext);
+     }
+ }
}

```

```

app.listen(3000, async () => {
  await optimize(root);
  console.log(
    green("🚀 No-Bundle 服务已经成功启动!"),
    `耗时: ${Date.now() - startTime}ms`
  );
  console.log(`> 本地访问路径: ${blue("http://localhost:3000")}`);
});
}

```

其中 `resolvePlugins` 方法我们还未定义，你可以新建 `src/node/plugins/index.ts` 文件，内容如下：

```

import { Plugin } from "../plugin";

export function resolvePlugins(): Plugin[] {
  // 下一部分会逐个补充插件逻辑
  return [];
}

```

入口 HTML 加载

现在我们基于如上的插件机制，来实现 Vite 的核心编译能力。

首先要考虑的就是入口 HTML 如何编译和加载的问题，这里我们可以通过一个服务中间件，配合插件机制来实现。具体而言，你可以新建

`src/node/server/middlewares/indexHtml.ts`，内容如下：

```

import { NextHandleFunction } from "connect";
import { ServerContext } from "../index";
import path from "path";
import { pathExists, readFile } from "fs-extra";

export function indexHtmlMiddleware(
  serverContext: ServerContext
): NextHandleFunction {
  return async (req, res, next) => {
    if (req.url === "/") {
      const { root } = serverContext;
      // 默认使用项目根目录下的 index.html
      const indexHtmlPath = path.join(root, "index.html");
      if (await pathExists(indexHtmlPath)) {
        const rawHtml = await readFile(indexHtmlPath, "utf8");
        let html = rawHtml;
        // 通过执行插件的 transformIndexHtml 方法来对 HTML 进行自定义的修改
        for (const plugin of serverContext.plugins) {
          if (plugin.transformIndexHtml) {
            html = await plugin.transformIndexHtml(html);
          }
        }
      }
    }
    next();
  };
}

```



```

    }
  }

  res.statusCode = 200;
  res.setHeader("Content-Type", "text/html");
  return res.end(html);
}
}
return next();
};
}

```

然后在服务中应用这个中间件:

```

// src/node/server/index.ts
// 需要增加的引入语句
import { indexHtmlMiddleware } from "../middlewares/indexHtml";

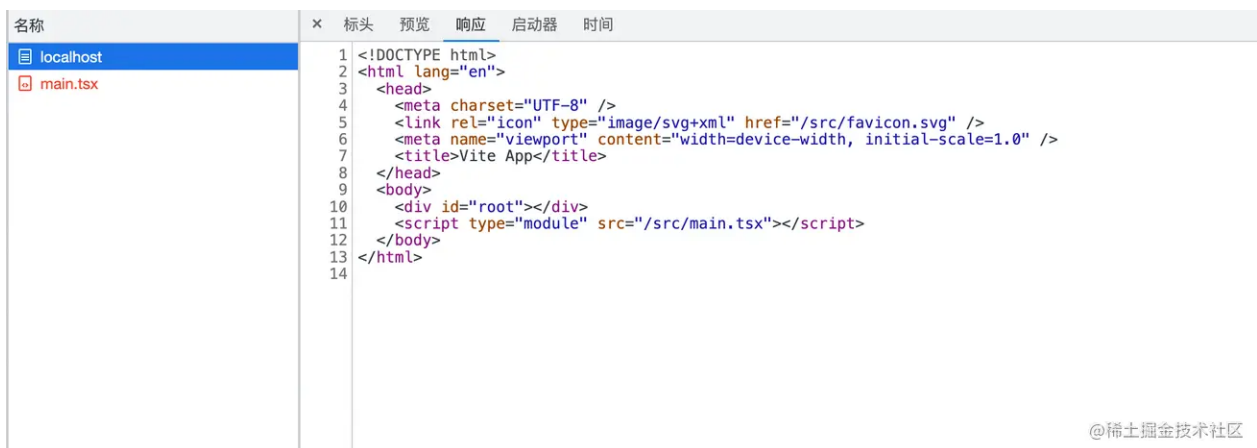
// 省略中间代码

// 处理入口 HTML 资源
app.use(indexHtmlMiddleware(serverContext));

app.listen(3000, async () => {
  // 省略
});

```

接下来通过 `pnpm dev` 启动项目，然后访问 `http://localhost:3000`，从网络面板中你可以查看到 HTML 的内容已经成功返回:



不过当前的页面并没有任何内容，因为 HTML 中引入的 TSX 文件并没有被正确编译。接下来，我们就来处理 TSX 文件的编译工作。

JS/TS/JSX/TSX 编译能力

首先新增一个中间件 `src/node/server/middlewares/transform.ts` , 内容如下:

```
import { NextHandleFunction } from "connect";
import {
  isJSRequest,
  cleanUrl,
} from "../../utils";
import { ServerContext } from "../index";
import createDebug from "debug";

const debug = createDebug("dev");

export async function transformRequest(
  url: string,
  serverContext: ServerContext
) {
  const { pluginContainer } = serverContext;
  url = cleanUrl(url);
  // 简单来说, 就是依次调用插件容器的 resolveId、load、transform 方法
  const resolvedResult = await pluginContainer.resolveId(url);
  let transformResult;
  if (resolvedResult?.id) {
    let code = await pluginContainer.load(resolvedResult.id);
    if (typeof code === "object" && code !== null) {
      code = code.code;
    }
    if (code) {
      transformResult = await pluginContainer.transform(
        code as string,
        resolvedResult?.id
      );
    }
  }
  return transformResult;
}

export function transformMiddleware(
  serverContext: ServerContext
): NextHandleFunction {
  return async (req, res, next) => {
    if (req.method !== "GET" || !req.url) {
      return next();
    }
    const url = req.url;
    debug("transformMiddleware: %s", url);
    // transform JS request
    if (isJSRequest(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;
      }
      // 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();
};
}

```

同时，我们也需要补充如下的工具函数和常量定义：

```

// src/node/utils.ts
import { JS_TYPES_RE } from './constants.ts'

export const isJSRequest = (id: string): boolean => {
    id = cleanUrl(id);
    if (JS_TYPES_RE.test(id)) {
        return true;
    }
    if (!path.extname(id) && !id.endsWith("/")) {
        return true;
    }
    return false;
};

export const cleanUrl = (url: string): string =>
    url.replace(HASH_RE, "").replace(QEURY_RE, "");

// src/node/constants.ts
export const JS_TYPES_RE = /\.(?:j|t)sx?$|\.(mjs)$/;
export const QEURY_RE = /\?.*$/s;
export const HASH_RE = /\#.*$/s;

```

从如上的核心编译函数 `transformRequest` 可以看出，Vite 对于 JS/TS/JSX/TSX 文件的编译流程主要是依次调用插件容器的如下方法：

- `resolveId`
- `load`
- `transform`

其中会经历众多插件的处理逻辑，那么，对于 TSX 文件的编译逻辑，也分散到了各个插件当中，具体来说主要包含以下的插件：

- 路径解析插件
- Esbuild 语法编译插件
- import 分析插件

接下来，我们就开始依次实现这些插件。

1. 路径解析插件

当浏览器解析到如下的标签时：

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

会自动发送一个路径为 `/src/main.tsx` 的请求，但如果服务端不做任何处理，是无法定位到源文件的，随之会返回 404 状态码：

A screenshot of a web browser's developer console. It shows a red error message: "GET http://localhost:3000/src/main.tsx net::ERR_ABORTED 404 (Not Found)". The address bar at the top shows "localhost:3000".

因此，我们需要开发一个路径解析插件，对请求的路径进行处理，使之能转换真实文件系统中的路径。你可以新建文件 `src/node/plugins/resolve.ts`，内容如下：

```
import resolve from "resolve";
import { Plugin } from "../plugin";
import { ServerContext } from "../server/index";
import path from "path";
import { pathExists } from "fs-extra";
import { DEFAULT_EXTENSIONS } from "../constants";
import { cleanUrl } from "../utils";

export function resolvePlugin(): Plugin {
  let serverContext: ServerContext;
  return {
    name: "m-vite:resolve",
    configureServer(s) {
      // 保存服务端上下文
      serverContext = s;
    },
    async resolveId(id: string, importer?: string) {
      // 1. 绝对路径
      if (path.isAbsolute(id)) {
        if (await pathExists(id)) {
          return { id };
        }
      }
      // 加上 root 路径前缀，处理 /src/main.tsx 的情况
      id = path.join(serverContext.root, id);
      if (await pathExists(id)) {
        return { id };
      }
      // 2. 相对路径
      else if (id.startsWith(".")) {
        if (!importer) {
          throw new Error("`importer` should not be undefined");
        }
      }
    }
  };
}
```

```

const hasExtension = path.extname(id).length > 1;
let resolvedId: string;
// 2.1 包含文件名后缀
// 如 ./App.tsx
if (hasExtension) {
  resolvedId = resolve.sync(id, { basedir: path.dirname(importer) });
  if (await pathExists(resolvedId)) {
    return { id: resolvedId };
  }
}
// 2.2 不包含文件名后缀
// 如 ./App
else {
  // ./App -> ./App.tsx
  for (const extname of DEFAULT_EXTENSIONS) {
    try {
      const withExtension = `${id}${extname}`;
      resolvedId = resolve.sync(withExtension, {
        basedir: path.dirname(importer),
      });
      if (await pathExists(resolvedId)) {
        return { id: resolvedId };
      }
    } catch (e) {
      continue;
    }
  }
}
return null;
},
};
}

```

这样对于 `/src/main.tsx`，在插件中会转换为文件系统中的真实路径，从而让模块在 load 钩子中能够正常加载(加载逻辑在 Esbuild 语法编译插件实现)。

接着我们来补充一下目前缺少的常量:

```

// src/node/constants.ts
export const DEFAULT_EXTENSIONS = [".tsx", ".ts", ".jsx", ".js"];

```

2. Esbuild 语法编译插件

这个插件的作用比较好理解，就是将 JS/TS/JSX/TSX 编译成浏览器可以识别的 JS 语法，可以利用 Esbuild 的 Transform API 来实现。你可以新建 `src/node/plugins/esbuild.ts` 文件，内容如下:

```

import { readFile } from "fs-extra";
import { Plugin } from "../plugin";
import { isJSRequest } from "../utils";
import esbuild from "esbuild";
import path from "path";

export function esbuildTransformPlugin(): Plugin {
  return {
    name: "m-vite:esbuild-transform",
    // 加载模块
    async load(id) {
      if (isJSRequest(id)) {
        try {
          const code = await readFile(id, "utf-8");
          return code;
        } catch (e) {
          return null;
        }
      }
    },
    async transform(code, id) {
      if (isJSRequest(id)) {
        const extname = path.extname(id).slice(1);
        const { code: transformedCode, map } = await esbuild.transform(code, {
          target: "esnext",
          format: "esm",
          sourcemap: true,
          loader: extname as "js" | "ts" | "jsx" | "tsx",
        });
        return {
          code: transformedCode,
          map,
        };
      }
      return null;
    },
  };
}

```

3. import 分析插件

在将 TSX 转换为浏览器可以识别的语法之后，是不是就可以直接返回给浏览器执行了呢？

显然不是，我们还考虑如下的一些问题：

- 对于第三方依赖路径(bare import)，需要重写为预构建产物路径；
- 对于绝对路径和相对路径，需要借助之前的路径解析插件进行解析。

好，接下来，我们就在 import 分析插件中一一解决这些问题：

```

// 新建 src/node/plugins/importAnalysis.ts
import { init, parse } from "es-module-lexer";
import {
  BARE_IMPORT_RE,
  DEFAULT_EXTENSIONS,
  PRE_BUNDLE_DIR,
} from "../constants";
import {
  cleanUrl,
  isJSRequest,
} from "../utils";
// magic-string 用来作字符串编辑
import MagicString from "magic-string";
import path from "path";
import { Plugin } from "../plugin";
import { ServerContext } from "../server/index";
import { pathExists } from "fs-extra";
import resolve from "resolve";

export function importAnalysisPlugin(): Plugin {
  let serverContext: ServerContext;
  return {
    name: "m-vite:import-analysis",
    configureServer(s) {
      // 保存服务端上下文
      serverContext = s;
    },
    async transform(code: string, id: string) {
      // 只处理 JS 相关的请求
      if (!isJSRequest(id)) {
        return null;
      }
      await init;
      // 解析 import 语句
      const [imports] = parse(code);
      const ms = new MagicString(code);
      // 对每一个 import 语句依次进行分析
      for (const importInfo of imports) {
        // 举例说明: const str = `import React from 'react'`
        // str.slice(s, e) => 'react'
        const { s: modStart, e: modEnd, n: modSource } = importInfo;
        if (!modSource) continue;
        // 第三方库: 路径重写到预构建产物的路径
        if (BARE_IMPORT_RE.test(modSource)) {
          const bundlePath = path.join(
            serverContext.root,
            PRE_BUNDLE_DIR,
            `${modSource}.js`
          );
          ms.overwrite(modStart, modEnd, bundlePath);
        } else if (modSource.startsWith(".") || modSource.startsWith("/")) {
          // 直接调用插件上下文的 resolve 方法, 会自动经过路径解析插件的处理
          const resolved = await this.resolve(modSource, id);
          if (resolved) {
            ms.overwrite(modStart, modEnd, resolved.id);
          }
        }
      }
    }
  }
}

```

```

    }

    return {
      code: ms.toString(),
      // 生成 SourceMap
      map: ms.generateMap(),
    };
  },
};
};
}

```

现在，我们便完成了 JS 代码的 import 分析工作。接下来，我们把上面实现的三个插件进行注册：

```

// src/node/plugin/index.ts
import { esbuildTransformPlugin } from "./esbuild";
import { importAnalysisPlugin } from "./importAnalysis";
import { resolvePlugin } from "./resolve";
import { Plugin } from "../plugin";

export function resolvePlugins(): Plugin[] {
  return [resolvePlugin(), esbuildTransformPlugin(), importAnalysisPlugin()];
}

```

然后在 playground 项目下执行 `pnpm dev`，在浏览器里面访问 `http://localhost:3000`，你可以在网络面板中发现 `main.tsx` 的内容以及被编译为下面这样：

```

1 import React from "/Users/.../code/juejin-book-vite/mini-vite/playground/node_modules/.m-vite/react.js";
2 import ReactDOM from "/Users/.../code/juejin-book-vite/mini-vite/playground/node_modules/.m-vite/react-dom.js";
3 import App from "/Users/.../code/juejin-book-vite/mini-vite/playground/src/App.tsx";
4 ReactDOM.render(/* @__PURE__ */ React.createElement(App, null), document.getElementById("root"));
5

```

@稀土掘金技术社区

同时，页面内容也能被渲染出来了：

Hello Vite + React

count is: 0

Edit App and save e test.

[Learn React](#) | [Vite Docs](#)

@稀土掘金技术社区

OK，目前为止我们就基本上完成 JS/TS/JSX/TSX 文件的编译。

小结

本小节的内容就到这里，相信你如果能一直跟着做到这里，也已经收获满满了。我们最后来回顾和小结一下，这一节我们主要来手写 Vite 的 no-bundle 服务，完成了**开发环境搭建、预构建功能的开发、插件机制的搭建、入口 HTML 加载和 JS/TS/JSX/TSX 的编译功能**。

在下一小节，我们将继续完善当前的 no-bundle 服务器，完成 CSS 编译、静态资源加载和 HMR 系统的实现，让我们下一节再见👋

上一篇：热更新：基于 ESM 的毫秒级 HMR 的实现揭秘

下一篇：手写 Vite: 实现 no-bundle 开发服务(下)