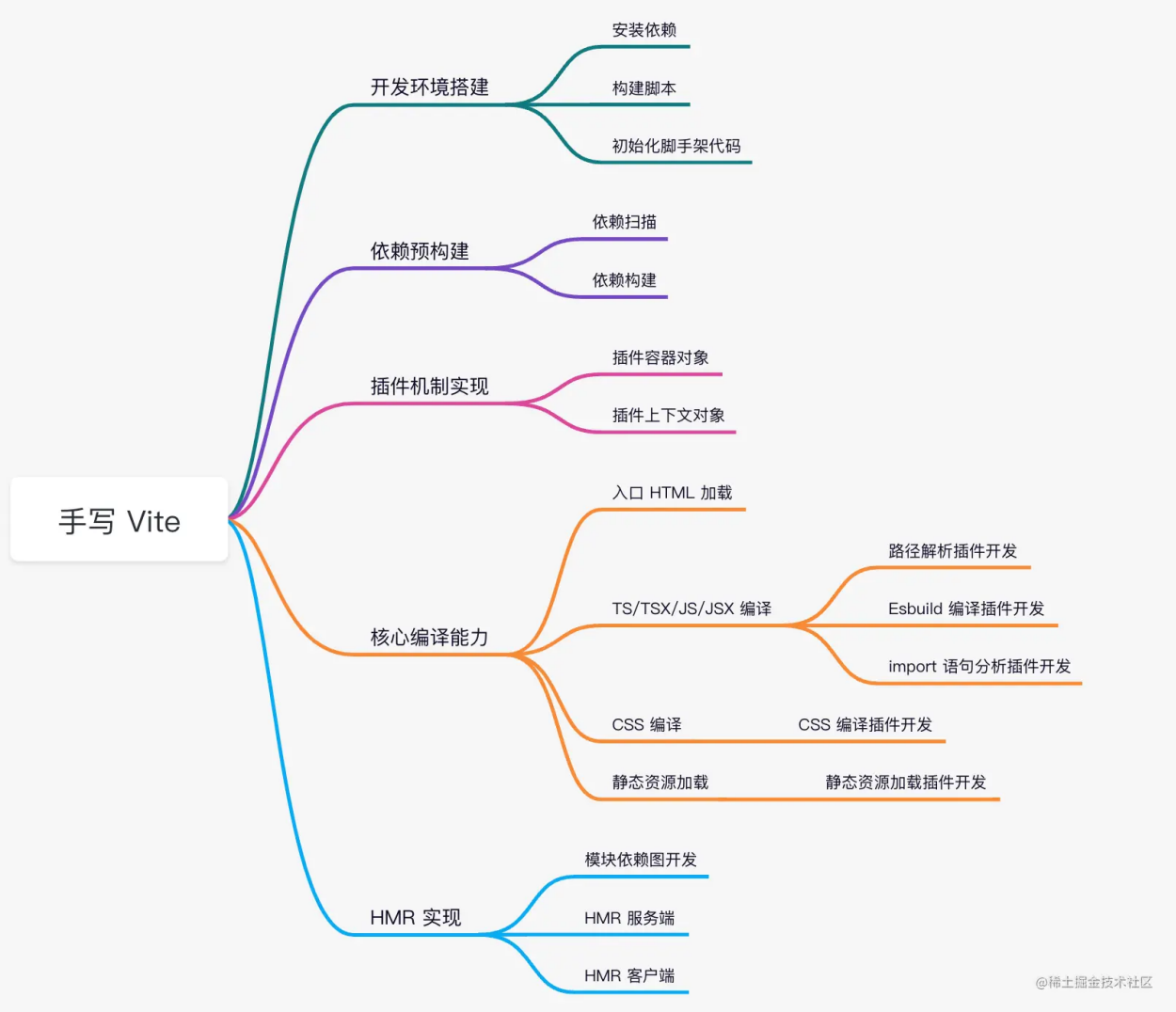
手写 Vite: 实现 no-bundle 开发服务(下)

发布于 2022-05-09

本小节为上一小节的续篇，我们基于下面的导图继续实现 no-bundle 构建服务:



接下来我们需要完成如下的模块:

- CSS 编译插件
- 静态资源加载插件
- 模块依赖图开发，并在 transform 中间件中接入
- HMR 服务端代码开发
- HMR 客户端代码开发

话不多说，下面我们正式进入实战的环节。

> 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

CSS 编译插件

首先，我们可以看看项目中 CSS 代码是如何被引入的:

```
// playground/src/main.tsx
import "./index.css";
```

为了让 CSS 能够在 no-bundle 服务中正常加载，我们需要将其包装成浏览器可以识别的模块格式，也就是 JS 模块 ，其中模块加载和转换的逻辑我们可以通过插件来实现。当然，首先我们需要在 transform 中间件中允许对 CSS 的请求进行处理，代码如下:

```
// src/node/server/middlewares/transform.ts
// 需要增加的导入语句
+ import { isCSSRequest } from '../../utils';

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)) {
+    if (isJSRequest(url) || isCSSRequest(url)) {
      // 后续代码省略
     }

    next();
  };
}
```

然后我们来补充对应的工具函数:

```
// src/node/utils.ts
export const isCSSRequest = (id: string): boolean =>
  cleanUrl(id).endsWith(".css");
```

现在我们来开发 CSS 的编译插件，你可以新建 src/node/plugins/css.ts 文件，内容如下:

```
import { readFile } from "fs-extra";
import { Plugin } from "../plugin";

export function cssPlugin(): Plugin {
  return {
    name: "m-vite:css",
    load(id) {
      // 加载
      if (id.endsWith(".css")) {
        return readFile(id, "utf-8");
      }
    },
    // 转换逻辑
    async transform(code, id) {
      if (id.endsWith(".css")) {
        // 包装成 JS 模块
        const jsContent = `
const css = "${code.replace(/\n/g, "")}";
const style = document.createElement("style");
style.setAttribute("type", "text/css");
style.innerHTML = css;
document.head.appendChild(style);
export default css;
`.trim();
        return {
          code: jsContent,
        };
      }
      return null;
    },
  };
}
```

这个插件的逻辑比较简单，主要是将封装一层 JS 样板代码，将 CSS 包装成一个 ES 模块，当浏览器执行这个模块的时候，会通过一个 style 标签将 CSS 代码作用到页面中，从而使样式代码生效。

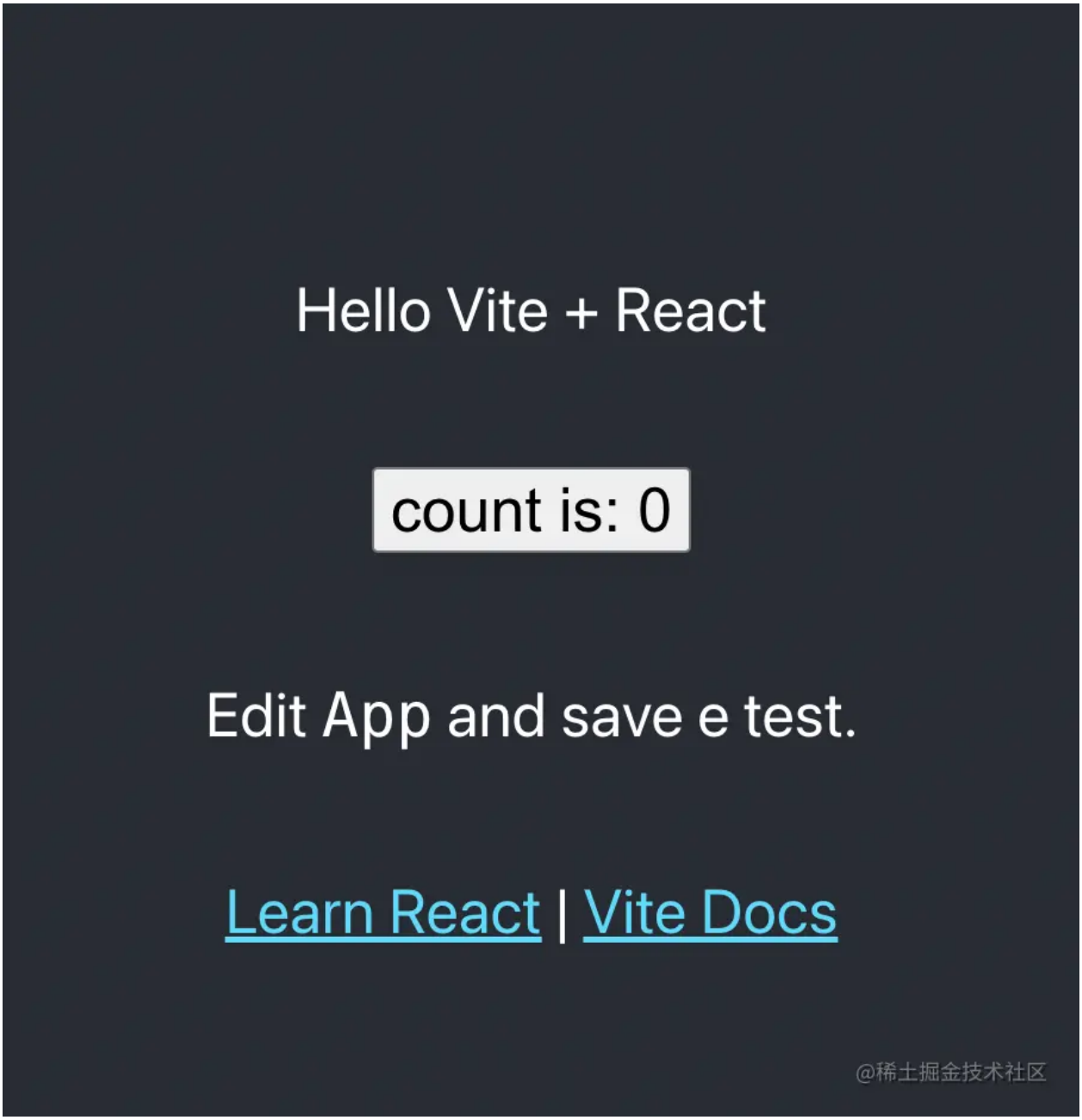
接着我们来注册这个 CSS 插件:

```
// src/node/plugins/index.ts
+ import { cssPlugin } from "./css";

export function resolvePlugins(): Plugin[] {
  return [
    // 省略前面的插件
+     cssPlugin(),
  ];
}
```

现在，你可以通过 `pnpm dev` 来启动 playground 项目，不过在启动之前，需要保证 TSX 文件已经引入了对应的 CSS 文件:

```
// playground/src/main.tsx
import "./index.css";

// playground/src/App.tsx
import "./App.css";
```

在启动项目后，打开浏览器进行访问，可以看到样式已经正常生效:



静态资源加载

在完成 CSS 加载之后，我们现在继续完成静态资源的加载。以 playground 项目为例，我们来支持 svg 文件的加载。首先，我们看看 svg 文件是如何被引入并使用的:

```
// playground/src/App.tsx
import logo from "./logo.svg";

function App() {
  return (
    <img className="App-logo" src={logo} alt="" />
  )
}
```

站在 no-bundle 服务的角度，从如上的代码我们可以分析出静态资源的两种请求:

- import 请求。如 `import logo from "./logo.svg"` 。
- 资源内容请求。如 img 标签将资源 url 填入 src，那么浏览器会请求具体的资源内容。

因此，接下来为了实现静态资源的加载，我们需要做两手准备: 对静态资源的 import 请求返回资源的 url；对于具体内容的请求，读取静态资源的文件内容，并响应给浏览器。

首先处理 import 请求，我们可以在 TSX 的 import 分析插件中，给静态资源相关的 import 语句做一个标记:

```
// src/node/plugins/importAnalysis.ts

async transform(code, id) {
  // 省略前面的代码
  for (const importInfo of imports) {
    const { s: modStart, e: modEnd, n: modSource } = importInfo;
    if (!modSource) continue;
+    // 静态资源
+    if (modSource.endsWith(".svg")) {
+      // 加上 ?import 后缀
+      const resolvedUrl = path.join(path.dirname(id), modSource);
+      ms.overwrite(modStart, modEnd, `${resolvedUrl}?import`);
+      continue;
    }
  }
}
```

编译后的 App.tsx 内容如下:

```
import { useState } from "/Users/[illegible]/code/juejin-book-vite/mini-vite/playground/node_modules/.m-vite/react.js";
import React from "/Users/[illegible]/code/juejin-book-vite/mini-vite/playground/node_modules/.m-vite/react.js";
import logo from "/Users/[illegible]/code/juejin-book-vite/mini-vite/playground/src/logo.svg?import";
import "/src/App.css";
function App() {
  const [count, setCount] = useState(0);
  return /* @__PURE__ */ React.createElement("div", {
    className: "App"
  }, /* @__PURE__ */ React.createElement("header", {
    className: "App-header"
  }, /* @__PURE__ */ React.createElement("img", {
    className: "App-logo",
    src: logo,
    alt: ""
  }), /* @__PURE__ */ React.createElement("p", null, "Hello Vite + React"), /* @__PURE__ */ React.createElement("p", null
    type: "button",
    onClick: () => setCount((count2) => count2 + 1)
  }, "count is: ", count)), /* @__PURE__ */ React.createElement("p", null, "Edit ", /* @__PURE__ */ React.createElement("
    className: "App-link",
    href: "https://reactjs.org",
    target: "_blank",
    rel: "noopener noreferrer"
  }, "Learn React"), " | ", /* @__PURE__ */ React.createElement("a", {
    className: "App-link",
    href: "https://vitejs.dev/guide/features.html",
    target: "_blank",
```


接着浏览器会发出带有 `?import` 后缀的请求，我们在 transform 中间件进行处理:

```
// src/node/server/middlewares/transform.ts
// 需要增加的导入语句
+ import { isImportRequest } from '../../utils';

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) || isCSSRequest(url)) {
+    if (isJSRequest(url) || isCSSRequest(url) || isImportRequest(url)) {
      // 后续代码省略
     }

    next();
  };
}
```

然后补充对应的工具函数:

```
// src/node/utils.ts
export function isImportRequest(url: string): boolean {
  return url.endsWith("?import");
}
```

此时，我们就可以开发静态资源插件了。新建 `src/node/plugins/assets.ts` ，内容如下:

```
import { Plugin } from "../plugin";
import { cleanUrl, removeImportQuery } from "../utils";
```

```
export function assetPlugin(): Plugin {
  return {
    name: "m-vite:asset",
    async load(id) {
      const cleanedId = removeImportQuery(cleanUrl(id));
      // 这里仅处理 svg
      if (cleanedId.endsWith(".svg")) {
        return {
          // 包装成一个 JS 模块
          code: `export default "${cleanedId}"`,
        };
      }
    },
  };
}
```

接着来注册这个插件:

```
// src/node/plugins/index.ts
+ import { assetPlugin } from "./assets";

export function resolvePlugins(): Plugin[] {
  return [
    // 省略前面的插件
+     assetPlugin(),
  ];
}
```

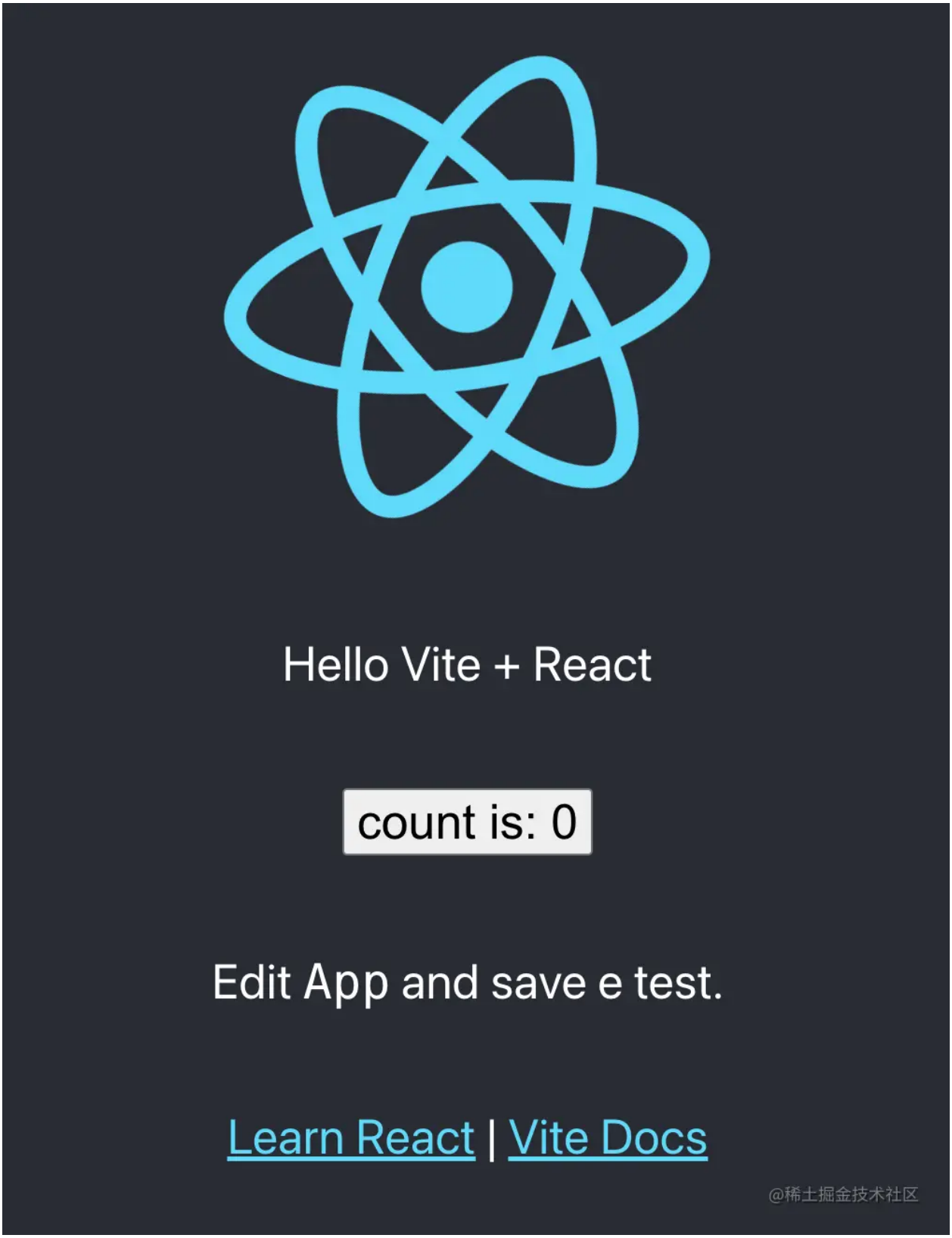
OK，目前我们处理完了静态资源的 import 请求，接着我们还需要处理非 import 请求，返回资源的具体内容。我们可以通过一个中间件来进行处理:

```
// src/node/server/middlewares/static.ts
import { NextHandleFunction } from "connect";
import { isImportRequest } from "../../utils";
// 一个用于加载静态资源的中间件
import sirv from "sirv";

export function staticMiddleware(): NextHandleFunction {
  const serveFromRoot = sirv("/", { dev: true });
  return async (req, res, next) => {
    if (!req.url) {
      return;
    }
    // 不处理 import 请求
    if (isImportRequest(req.url)) {
      return;
    }
    serveFromRoot(req, res, next);
  };
}
```

然后在服务中注册这个中间件:

```
// src/node/server/index.ts
// 需要添加的引入语句
+ import { staticMiddleware } from "./middlewares/static";

export async function startDevServer() {
  // 前面的代码省略
+  app.use(staticMiddleware());

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

现在，你可以通过 `pnpm dev` 启动 playground 项目，在浏览器中访问，可以发现 svg 图片已经能够成功显示了:



其实不光是 svg 文件，几乎所有格式的静态资源都可以按照如上的思路进行处理:

> 通过加入 `?import` 后缀标识 import 请求，返回将静态资源封装成一个 JS 模块，即 `export default xxx` 的形式，导出资源的真实地址。

> 对非 import 请求，响应静态资源的具体内容，通过 `Content-Type` 响应头告诉浏览器资源的类型(这部分工作 sirv 中间件已经帮我们做了)。

模块依赖图开发

模块依赖图在 no-bundle 构建服务中是一个不可或缺的数据结构，一方面可以存储各个模块的信息，用于记录编译缓存，另一方面也可以记录各个模块间的依赖关系，用于实现 HMR。

接下来我们来实现模块依赖图，即 `ModuleGraph` 类，新建 `src/node/ModuleGraph.ts` ，内容如下:

```
import { PartialResolvedId, TransformResult } from "rollup";
import { cleanUrl } from "./utils";

export class ModuleNode {
  // 资源访问 url
  url: string;
  // 资源绝对路径
  id: string | null = null;
  importers = new Set<ModuleNode>();
  importedModules = new Set<ModuleNode>();
  transformResult: TransformResult | null = null;
  lastHMRTimestamp = 0;
  constructor(url: string) {
    this.url = url;
  }
}

export class ModuleGraph {
  // 资源 url 到 ModuleNode 的映射表
  urlToModuleMap = new Map<string, ModuleNode>();
  // 资源绝对路径到 ModuleNode 的映射表
  idToModuleMap = new Map<string, ModuleNode>();

  constructor(
    private resolveId: (url: string) => Promise<PartialResolvedId | null>
  ) {}

  getModuleById(id: string): ModuleNode | undefined {
    return this.idToModuleMap.get(id);
  }

  async getModuleByUrl(rawUrl: string): Promise<ModuleNode | undefined> {
    const { url } = await this._resolve(rawUrl);
    return this.urlToModuleMap.get(url);
  }

  async ensureEntryFromUrl(rawUrl: string): Promise<ModuleNode> {
    const { url, resolvedId } = await this._resolve(rawUrl);
    // 首先检查缓存
    if (this.urlToModuleMap.has(url)) {
      return this.urlToModuleMap.get(url) as ModuleNode;
    }
    // 若无缓存，更新 urlToModuleMap 和 idToModuleMap
```

```
    const mod = new ModuleNode(url);
    mod.id = resolvedId;
    this.urlToModuleMap.set(url, mod);
    this.idToModuleMap.set(resolvedId, mod);
    return mod;
  }

  async updateModuleInfo(
    mod: ModuleNode,
    importedModules: Set<string | ModuleNode>
  ) {
    const prevImports = mod.importedModules;
    for (const curImports of importedModules) {
      const dep =
        typeof curImports === "string"
          ? await this.ensureEntryFromUrl(cleanUrl(curImports))
          : curImports;
      if (dep) {
        mod.importedModules.add(dep);
        dep.importers.add(mod);
      }
    }
    // 清除已经不再被引用的依赖
    for (const prevImport of prevImports) {
      if (!importedModules.has(prevImport.url)) {
        prevImport.importers.delete(mod);
      }
    }
  }

  // HMR 触发时会执行这个方法
  invalidateModule(file: string) {
    const mod = this.idToModuleMap.get(file);
    if (mod) {
      // 更新时间戳
      mod.lastHMRTimestamp = Date.now();
      mod.transformResult = null;
      mod.importers.forEach((importer) => {
        this.invalidateModule(importer.id!);
      });
    }
  }

  private async _resolve(
    url: string
  ): Promise<{ url: string; resolvedId: string }> {
    const resolved = await this.resolveId(url);
    const resolvedId = resolved?.id || url;
    return { url, resolvedId };
  }
}
```

相信经过第 23 小节的学习，你已经对模块依赖图的实现结构比较熟悉了，对于代码细节这里也不再赘述。接着我们看看如何将这个 ModuleGraph 接入到目前的架构中。

首先在服务启动前，我们需要初始化 ModuleGraph 实例:

```
// src/node/server/index.ts
+ import { ModuleGraph } from "../ModuleGraph";

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

export async function startDevServer() {
+  const moduleGraph = new ModuleGraph((url) => pluginContainer.resolveId(url));
  const pluginContainer = createPluginContainer(plugins);
  const serverContext: ServerContext = {
    root: process.cwd(),
    app,
    pluginContainer,
    plugins,
+    moduleGraph
  };

}
```

然后在加载完模块后，也就是调用插件容器的 load 方法后，我们需要通过 `ensureEntryFromUrl` 方法注册模块:

```
// src/node/server/middlewares/transform.ts
let code = await pluginContainer.load(resolvedResult.id);
if (typeof code === "object" && code !== null) {
  code = code.code;
}
+ const { moduleGraph }  = serverContext;
+ mod = await moduleGraph.ensureEntryFromUrl(url);
```

当我们对 JS 模块分析完 import 语句之后，需要更新模块之间的依赖关系:

```
// src/node/plugins/importAnalysis.ts
export function importAnalysis() {
  return {
    transform(code: string, id: string) {
      // 省略前面的代码
+      const { moduleGraph } = serverContext;
+      const curMod = moduleGraph.getModuleById(id)!;
+      const importedModules = new Set<string>();
      for(const importInfo of imports) {
        // 省略部分代码
        if (BARE_IMPORT_RE.test(modSource)) {
          // 省略部分代码
```

```
+          importedModules.add(bundlePath);
         } else if (modSource.startsWith(".") || modSource.startsWith("/")) {
           const resolved = await resolve(modSource, id);
           if (resolved) {
             ms.overwrite(modStart, modEnd, resolved);
+            importedModules.add(resolved);
           }
         }
       }
+      moduleGraph.updateModuleInfo(curMod, importedModules);
       // 省略后续 return 代码
     }
   }
}
```

现在，一个完整的模块依赖图就能随着 JS 请求的到来而不断建立起来了。另外，基于现在的模块依赖图，我们也可以记录模块编译后的产物，并进行缓存。让我们回到 transform 中间件中:

```
export async function transformRequest(
  url: string,
  serverContext: ServerContext
) {
  const { moduleGraph, pluginContainer } = serverContext;
  url = cleanUrl(url);
+  let mod = await moduleGraph.getModuleByUrl(url);
+  if (mod && mod.transformResult) {
+    return mod.transformResult;
+  }
  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;
    }
    mod = await moduleGraph.ensureEntryFromUrl(url);
    if (code) {
      transformResult = await pluginContainer.transform(
        code as string,
        resolvedResult?.id
      );
    }
  }
+  if (mod) {
+    mod.transformResult = transformResult;
+  }
  return transformResult;
}
```

在搭建好模块依赖图之后，我们把目光集中到最重要的部分——HMR 上面。

HMR 服务端

HMR 在服务端需要完成如下的工作:

- 创建文件监听器，以监听文件的变动
- 创建 WebSocket 服务端，负责和客户端进行通信
- 文件变动时，从 ModuleGraph 中定位到需要更新的模块，将更新信息发送给客户端

首先，我们来创建文件监听器:

```
// src/node/server/index.ts
import chokidar, { FSWatcher } from "chokidar";

export async function startDevServer() {
  const watcher = chokidar.watch(root, {
    ignored: ["**/node_modules/**", "**/.git/**"],
    ignoreInitial: true,
  });
}
```

接着初始化 WebSocket 服务端，新建 `src/node/ws.ts` ，内容如下:

```
import connect from "connect";
import { red } from "picocolors";
import { WebSocketServer, WebSocket } from "ws";
import { HMR_PORT } from "./constants";

export function createWebSocketServer(server: connect.Server): {
  send: (msg: string) => void;
  close: () => void;
} {
  let wss: WebSocketServer;
  wss = new WebSocketServer({ port: HMR_PORT });
  wss.on("connection", (socket) => {
    socket.send(JSON.stringify({ type: "connected" }));
  });

  wss.on("error", (e: Error & { code: string }) => {
    if (e.code !== "EADDRINUSE") {
      console.error(red(`WebSocket server error:\n${e.stack || e.message}`));
    }
  });

  return {
    send(payload: Object) {
      const stringified = JSON.stringify(payload);
      wss.clients.forEach((client) => {
        if (client.readyState === WebSocket.OPEN) {
          client.send(stringified);
        }
```

```
      });
    },

    close() {
      wss.close();
    },
  };
}
```

同时定义 `HMR_PORT` 常量:

```
// src/node/constants.ts
export const HMR_PORT = 24678;
```

接着我们将 WebSocket 服务端实例加入 no-bundle 服务中:

```
// src/node/server/index.ts
export interface ServerContext {
  root: string;
  pluginContainer: PluginContainer;
  app: connect.Server;
  plugins: Plugin[];
  moduleGraph: ModuleGraph;
+  ws: { send: (data: any) => void; close: () => void };
+  watcher: FSWatcher;
}

export async function startDevServer() {
+  // WebSocket 对象
+  const ws = createWebSocketServer(app);
  // // 开发服务器上下文
  const serverContext: ServerContext = {
    root: process.cwd(),
    app,
    pluginContainer,
    plugins,
    moduleGraph,
+    ws,
+    watcher
  };
}
```

下面我们来实现当文件变动时，服务端具体的处理逻辑，新建 `src/node/hmr.ts`：

```
import { ServerContext } from "./server/index";
import { blue, green } from "picocolors";
import { getShortName } from "./utils";

export function bindingHMREvents(serverContext: ServerContext) {
  const { watcher, ws, root } = serverContext;
```

```
  watcher.on("change", async (file) => {
    console.log(`✨${blue("[hmr]")} ${green(file)} changed`);
    const { moduleGraph } = serverContext;
    // 清除模块依赖图中的缓存
    await moduleGraph.invalidateModule(file);
    // 向客户端发送更新信息
    ws.send({
      type: "update",
      updates: [
        {
          type: "js-update",
          timestamp: Date.now(),
          path: "/" + getShortName(file, root),
          acceptedPath: "/" + getShortName(file, root),
        },
      ],
    });
  });
}
```

注意补充一下缺失的工具函数:

```
// src/node/utils.ts
export function getShortName(file: string, root: string) {
  return file.startsWith(root + "/") ? path.posix.relative(root, file) : file;
}
```

接着我们在服务中添加如下代码:

```
// src/node/server/index.ts
+  import { bindingHMREvents } from "../hmr";

// 开发服务器上下文
const serverContext: ServerContext = {
  root: process.cwd(),
  app,
  pluginContainer,
  plugins,
  moduleGraph,
  ws,
  watcher,
};
+ bindingHMREvents(serverContext);
```

HMR 客户端

HMR 客户端指的是我们向浏览器中注入的一段 JS 脚本，这段脚本中会做如下的事情:

- 创建 WebSocket 客户端，用于和服务端通信

- 在收到服务端的更新信息后，通过动态 import 拉取最新的模块内容，执行 accept 更新回调
- 暴露 HMR 的一些工具函数，比如 import.meta.hot 对象的实现

首先我们来开发客户端的脚本内容，你可以新建 `src/client/client.ts` 文件，然后在 tsup.config.ts 中增加如下的配置:

```
import { defineConfig } from "tsup";

export default defineConfig({
  entry: {
    index: "src/node/cli.ts",
+    client: "src/client/client.ts",
  },
});
```

> 注: 改动 tsup 配置之后，为了使最新配置生效，你需要在 `mini-vite` 项目中执行 `pnpm start` 重新进行构建。

客户端脚本的具体实现如下:

```
// src/client/client.ts
console.log("[vite] connecting...");

// 1. 创建客户端 WebSocket 实例
// 其中的 __HMR_PORT__ 之后会被 no-bundle 服务编译成具体的端口号
const socket = new WebSocket(`ws://localhost:__HMR_PORT__`, "vite-hmr");

// 2. 接收服务端的更新信息
socket.addEventListener("message", async ({ data }) => {
  handleMessage(JSON.parse(data)).catch(console.error);
});

// 3. 根据不同的更新类型进行更新
async function handleMessage(payload: any) {
  switch (payload.type) {
    case "connected":
      console.log(`[vite] connected.`);
      // 心跳检测
      setInterval(() => socket.send("ping"), 1000);
      break;

    case "update":
      // 进行具体的模块更新
      payload.updates.forEach((update: Update) => {
        if (update.type === "js-update") {
          // 具体的更新逻辑，后续来开发
        }
```

```
      });
      break;
    }
  }
```

关于客户端具体的 JS 模块更新逻辑和工具函数的实现，你暂且不用过于关心。我们先把这段比较简单的 HMR 客户端代码注入到浏览器中，首先在新建 `src/node/plugins/clientInject.ts`，内容如下:

```ts
import { CLIENT_PUBLIC_PATH, HMR_PORT } from "../constants";
import { Plugin } from "../plugin";
import fs from "fs-extra";
import path from "path";
import { ServerContext } from "../server/index";

export function clientInjectPlugin(): Plugin {
  let serverContext: ServerContext;
  return {
    name: "m-vite:client-inject",
    configureServer(s) {
      serverContext = s;
    },
    resolveId(id) {
      if (id === CLIENT_PUBLIC_PATH) {
        return { id };
      }
      return null;
    },
    async load(id) {
      // 加载 HMR 客户端脚本
      if (id === CLIENT_PUBLIC_PATH) {
        const realPath = path.join(
          serverContext.root,
          "node_modules",
          "mini-vite",
          "dist",
          "client.mjs"
        );
        const code = await fs.readFile(realPath, "utf-8");
        return {
          // 替换占位符
          code: code.replace("__HMR_PORT__", JSON.stringify(HMR_PORT)),
        };
      }
    },
    transformIndexHtml(raw) {
      // 插入客户端脚本
      // 即在 head 标签后面加上 <script type="module" src="/@vite/client"></script>
      // 注: 在 indexHtml 中间件里面会自动执行 transformIndexHtml 钩子
      return raw.replace(
        /(<head[^>]*>)/i,
        `$1<script type="module" src="${CLIENT_PUBLIC_PATH}"></script>`
      );
    },
```

```
  };
}
```

同时添加相应的常量声明:

```
// src/node/constants.ts
export const CLIENT_PUBLIC_PATH = "/@vite/client";
```

接着我们来注册这个插件:

```
// src/node/plugins/index.ts
+ import { clientInjectPlugin } from './clientInject';

export function resolvePlugins(): Plugin[] {
  return [
+     clientInjectPlugin()
    // 省略其它插件
  ]
}
```

需要注意的是， `clientInject` 插件最好放到最前面的位置，以免后续插件的 load 钩子干扰客户端脚本的加载。

接下来你可以在 playground 项目下执行 `pnpm dev` ，然后查看页面，可以发现控制台出现了如下的 log 信息:

查看网络面板，也能发现客户端脚本的请求被正常响应:

OK，接下来我们就来继续完善客户端脚本的具体实现。

值得一提的是，之所以我们可以在代码中编写类似 `import.meta.hot.xxx` 之类的方法，是因为 Vite 帮我们在模块最顶层注入了 `import.meta.hot` 对象，而这个对象由

createHotContext 来实现，具体的注入代码如下所示:

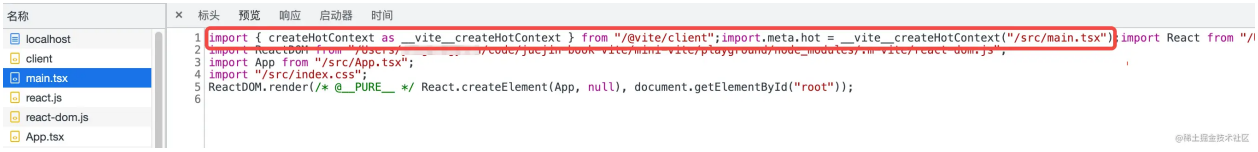
```
import { createHotContext as __vite__createHotContext } from "/@vite/client";
import.meta.hot = __vite__createHotContext("/src/App.tsx");
```

下面我们在 import 分析插件中做一些改动，实现插入这段代码的功能:

```
import { init, parse } from "es-module-lexer";
import {
  BARE_IMPORT_RE,
  CLIENT_PUBLIC_PATH,
  PRE_BUNDLE_DIR,
} from "../constants";
import {
  cleanUrl,
+  getShortName,
  isJSRequest,
} from "../utils";
import MagicString from "magic-string";
import path from "path";
import { Plugin } from "../plugin";
import { ServerContext } from "../server/index";

export function importAnalysisPlugin(): Plugin {
  let serverContext: ServerContext;
  return {
    name: "m-vite:import-analysis",
    configureServer(s) {
      serverContext = s;
    },
    async transform(code: string, id: string) {
+      if (!isJSRequest(id) || isInternalRequest(id)) {
        return null;
      }
      await init;
      const importedModules = new Set<string>();
      const [imports] = parse(code);
      const ms = new MagicString(code);
+      const resolve = async (id: string, importer?: string) => {
+        const resolved = await this.resolve(
+          id,
+          importer
+        );
+        if (!resolved) {
+          return;
+        }
+        const cleanedId = cleanUrl(resolved.id);
+        const mod = moduleGraph.getModuleById(cleanedId);
+        let resolvedId = `/${getShortName(resolved.id, serverContext.root)}`;
+        if (mod && mod.lastHMRTimestamp > 0) {
+          resolvedId += "?t=" + mod.lastHMRTimestamp;
+        }
+        return resolvedId;
+      };
```

```
      const { moduleGraph } = serverContext;
      const curMod = moduleGraph.getModuleById(id)!;

      for (const importInfo of imports) {
        const { s: modStart, e: modEnd, n: modSource } = importInfo;
        if (!modSource || isInternalRequest(modSource)) continue;
        // 静态资源
        if (modSource.endsWith(".svg")) {
          // 加上 ?import 后缀
          const resolvedUrl = path.join(path.dirname(id), modSource);
          ms.overwrite(modStart, modEnd, `${resolvedUrl}?import`);
          continue;
        }
        // 第三方库: 路径重写到预构建产物的路径
        if (BARE_IMPORT_RE.test(modSource)) {
          const bundlePath = path.join(
            serverContext.root,
            PRE_BUNDLE_DIR,
            `${modSource}.js`
          );
          ms.overwrite(modStart, modEnd, bundlePath);
          importedModules.add(bundlePath);
        } else if (modSource.startsWith(".") || modSource.startsWith("/")) {
+          const resolved = await resolve(modSource, id);
          if (resolved) {
            ms.overwrite(modStart, modEnd, resolved);
            importedModules.add(resolved);
          }
        }
      }
      // 只对业务源码注入
+     if (!id.includes("node_modules")) {
+       // 注入 HMR 相关的工具函数
+       ms.prepend(
+         `import { createHotContext as __vite__createHotContext } from "${CLIENT_PUBLIC_PAT
+           `import.meta.hot = __vite__createHotContext(${JSON.stringify(
+             cleanUrl(curMod.url)
+           )});`
+       );
+     }

      moduleGraph.updateModuleInfo(curMod, importedModules);

      return {
        code: ms.toString(),
        map: ms.generateMap(),
      };
    },
  };
}
```

接着启动 playground，打开页面后你可以发现 import.meta.hot 的实现代码已经被成功插入:



现在，我们回到客户端脚本的实现中，来开发 `createHotContext` 这个工具方法:

```ts
interface HotModule {
  id: string;
  callbacks: HotCallback[];
}

interface HotCallback {
  deps: string[];
  fn: (modules: object[]) => void;
}

// HMR 模块表
const hotModulesMap = new Map<string, HotModule>();
// 不在生效的模块表
const pruneMap = new Map<string, (data: any) => void | Promise<void>>();

export const createHotContext = (ownerPath: string) => {
  const mod = hotModulesMap.get(ownerPath);
  if (mod) {
    mod.callbacks = [];
  }

  function acceptDeps(deps: string[], callback: any) {
    const mod: HotModule = hotModulesMap.get(ownerPath) || {
      id: ownerPath,
      callbacks: [],
    };
    // callbacks 属性存放 accept 的依赖、依赖改动后对应的回调逻辑
    mod.callbacks.push({
      deps,
      fn: callback,
    });
    hotModulesMap.set(ownerPath, mod);
  }

  return {
    accept(deps: any, callback?: any) {
      // 这里仅考虑接受自身模块更新的情况
      // import.meta.hot.accept()
      if (typeof deps === "function" || !deps) {
        acceptDeps([ownerPath], ([mod]) => deps && deps(mod));
      }
    },
    // 模块不再生效的回调
    // import.meta.hot.prune(() => {})
    prune(cb: (data: any) => void) {
      pruneMap.set(ownerPath, cb);
    },
  };
};
```

在 accept 方法中，我们会用 hotModulesMap 这张表记录该模块所 accept 的模块，以及 accept 的模块更新之后回调逻辑。

接着，我们来开发客户端热更新的具体逻辑，也就是服务端传递更新内容之后客户端如何来派发更新。实现代码如下:

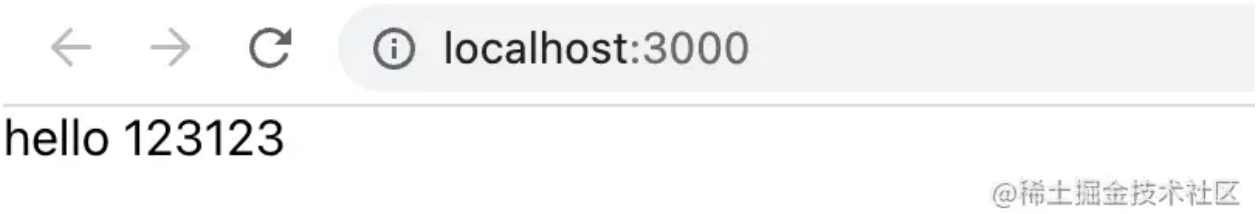
```
async function fetchUpdate({ path, timestamp }: Update) {
  const mod = hotModulesMap.get(path);
  if (!mod) return;

  const moduleMap = new Map();
  const modulesToUpdate = new Set<string>();
  modulesToUpdate.add(path);

  await Promise.all(
    Array.from(modulesToUpdate).map(async (dep) => {
      const [path, query] = dep.split(`?`);
      try {
        // 通过动态 import 拉取最新模块
        const newMod = await import(
          path + `?t=${timestamp}${query ? `&${query}` : ""}`
        );
        moduleMap.set(dep, newMod);
      } catch (e) {}
    })
  );

  return () => {
    // 拉取最新模块后执行更新回调
    for (const { deps, fn } of mod.callbacks) {
      fn(deps.map((dep: any) => moduleMap.get(dep)));
    }
    console.log(`[vite] hot updated: ${path}`);
  };
}
```

现在，我们可以来初步测试一下 HMR 的功能，你可以暂时将 main.tsx 的内容换成下面这样:

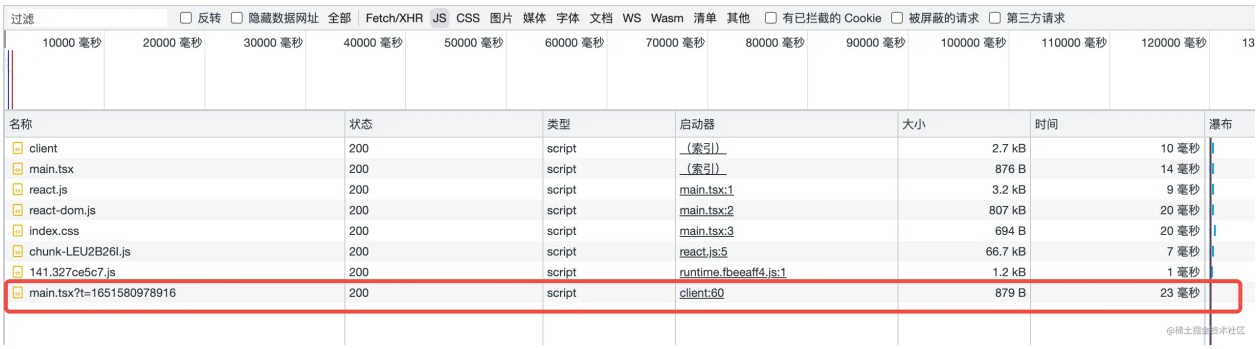
```
import React from "react";
import ReactDOM from "react-dom";
import "./index.css";

const App = () => <div>hello 123123</div>;

ReactDOM.render(<App />, document.getElementById("root"));

// @ts-ignore
import.meta.hot.accept(() => {
```

```
  ReactDOM.render(<App />, document.getElementById("root"));
});
```

启动 playground，然后打开浏览器，可以看到如下的文本:



现在回到编辑器中，修改文本内容，然后保存，你可以发现页面内容也跟着发生了变化，并且网络面板发出了拉取最新模块的请求，说明 HMR 已经成功生效:



同时，当你再次刷新页面，看到的仍然是最新的页面内容。这一点非常重要，之所以能达到这样的效果，是因为我们在文件改动后会调用 ModuleGraph 的 invalidateModule 方法，这个方法会清除热更模块以及所有上层引用方模块的编译缓存:

```
// 方法实现
invalidateModule(file: string) {
  const mod = this.idToModuleMap.get(file);
  if (mod) {
    mod.lastHMRTimestamp = Date.now();
    mod.transformResult = null;
    mod.importers.forEach((importer) => {
      this.invalidateModule(importer.id!);
    });
  }
}
```

这样每次经过 HMR 后，再次刷新页面，渲染出来的一定是最新的模块内容。

当然，我们也可以对 CSS 实现热更新功能，在客户端脚本中添加如下的工具函数:

```
const sheetsMap = new Map();

export function updateStyle(id: string, content: string) {
```

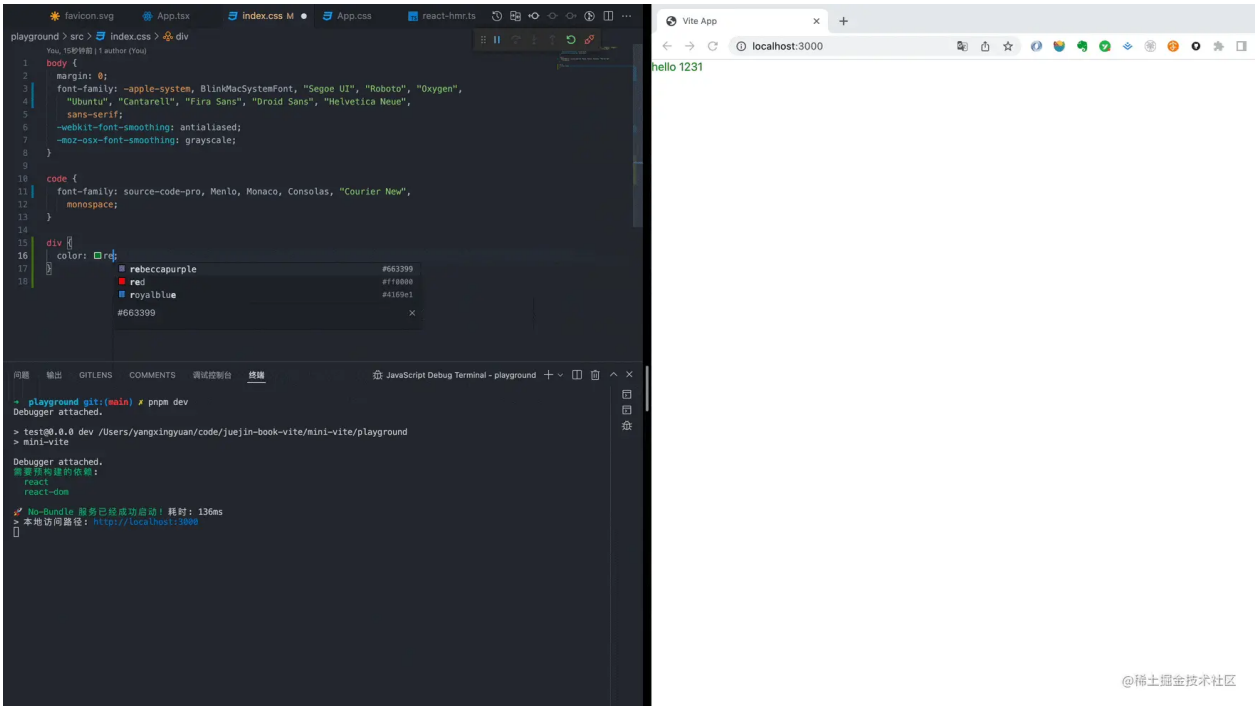
```
  let style = sheetsMap.get(id);
  if (!style) {
    // 添加 style 标签
    style = document.createElement("style");
    style.setAttribute("type", "text/css");
    style.innerHTML = content;
    document.head.appendChild(style);
  } else {
    // 更新 style 标签内容
    style.innerHTML = content;
  }
  sheetsMap.set(id,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

export function removeStyle(id: string): void {
  const style = sheetsMap.get(id);
  if (style) {
    document.head.removeChild(style);
  }
  sheetsMap.delete(id);
}
```

紧接着我们调整一下 CSS 编译插件的代码:

```
import { readFile } from "fs-extra";
import { CLIENT_PUBLIC_PATH } from "../constants";
import { Plugin } from "../plugin";
import { ServerContext } from "../server";
import { getShortName } from "../utils";

export function cssPlugin(): Plugin {
  let serverContext: ServerContext;
  return {
    name: "m-vite:css",
    configureServer(s) {
      serverContext = s;
    },
    load(id) {
      if (id.endsWith(".css")) {
        return readFile(id, "utf-8");
      }
    },
    // 主要变动在 transform 钩子中
    async 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;
    },
  };
}
```

最后，你可以重启 playground 项目，本地尝试修改 CSS 代码，可以看到类似如下的热更新效果:



小结

OK，本节的内容到这里就结束了，恭喜你完成了本次手写 Vite 的实战项目，最后我们来总结和回顾一下。

在这一小节，我们完成了 CSS 编译插件、静态资源加载插件、模块依赖图、编译缓存、HMR 服务端和客户端的实现。其中，你需要重点掌握以下的开发要点:

> CSS 的模块热更新如何实现?
> 静态资源的加载分为哪两种请求？no-bundle 服务中分别是如何处理的?
> HMR 服务端和客户端做了哪些事情?
> 如何保证在 HMR 更新之后，刷新页面后依然能保证是最新的模块内容?

最后，欢迎你把自己的学习心得打在评论区，大家一起来交流，我们下一节再见。

上一篇：手写 Vite: 实现 no-bundle 开发服务(上)

下一篇：手写 Bundler: 实现 JavaScript AST 解析器——词法分析、语义分析