vite核心原理（七）—— HMR热重载

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。

在模块依赖图章节曾介绍 vite 使用模块依赖图来管理各个模块之间的依赖关系，这种依赖关系的建立为vite 轻量快速的热重载奠定了基础。本节将详细讲解 vite 的整个热重载实现过程。

一、本地文件监听

chokidar是一个极简高效的跨平台文件监视库，在dev 服务启动函数createServer中会调用chokidar.watch开启文件监听。watch方法接收两个参数，第一个是文件路径、要递归监视的目录或 glob 模式；第二个参数是配置项options，这里举几个本项目中用到的下参数：

- ignored：定义要忽略的文件/路径
- ignoreInitial：初始化时当 chokidar 发现文件路径时是否也要为匹配路径发出 add/addDir 事件，true不发起，false 发起，默认为 false
- disableGlobbing： 如果设置为 true，则传递给 .watch() 和 .add() 的字符串将被视为文字路径名，即使它们看起来像 glob
- ignorePermissionErrors：是否尽可能监视没有读取权限的文件，如果由于设置为 true 的 EPERM 或 EACCES 而导致监视失败，则错误将被静默抑制

```
// node/server/index.ts
export const createServer = async (inlineConfig: InlineConfig = {}) => {
```

```
    // 省略其它代码
    const watcher = chokidar.watch(path.resolve(root),
resolvedWatchOptions);
    // 绑定文件监听事件
    bindingHMREvents(serverContext);
    // 省略其它代码
}

// node/watch.ts
export function resolveChokidarOptions(
    config: ResolvedConfig,
    options: Record<string, any>
): Record<string, any> {
    const { ignored = [], ...otherOptions } = options ?? {};

    const resolvedWatchOptions: Record<string, any> = {
        ignored: [
            '**/.git/**',
            '**/node_modules/**',
            '**/test-results/**', // Playwright
            config.cacheDir + '/**',
            ...(Array.isArray(ignored) ? ignored : [ignored]),
        ],
        ignoreInitial: true,
        ignorePermissionErrors: true,
        ...otherOptions,
    };

    return resolvedWatchOptions;
}
```

bindingHMREvents 绑定文件监听事件

```
// node/server/hmr.ts
export const bindingHMREvents = (serverContext: ServerContext) => {
    const { watcher } = serverContext;
```

```
    // 文件内容改变事件
    watcher.on("change", async (file) => {
        console.log('change')
    });

    // 文件增加事件
    watcher.on("add", async (file) => {
        console.log('add');
    });

    // 文件删除事件
    watcher.on("unlink", async (file) => {
        console.log('unlink');
    });
}
```

二、建立 websocket 连接

当本地文件改动时，服务端需要通知客户端文件发生了变化，客户端需要根据变化类型做出响应的更新操作，因此服务端和客户端都需要建立各自的 webSokcet服务

1. 服务端 websocket
   服务端 websocket 还是在启动服务方法createServer中创建

```
    // node/server/index.ts
    export const createServer = async (inlineConfig: InlineConfig = {})
    => {
       // 省略其它代码
       const ws = createWebSocketServer(app);
       // 省略其它代码
    }

    // node/ws.ts
    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();
        },
    };
}
```

2. 客户端 websocket

   客户端 websocket 则是在client.ts中创建

```
// client/client.ts
```

```
// 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(async (update: any) => {
                if (update.type === "js-update") {
                    // 具体的更新逻辑，后续来开发
                    const cb = await fetchUpdate(update);
                    cb!();
                }
            });
            break;
    }
}
```

这个 client.ts需要在 index.html中引入，但是在我们的index.html中没有写 script 标签去引入这个文件，实际上这个 client.ts是在请求 index.html时候通过插件注入进去的。在indexHtmlMiddware中会调用transformIndexHtml方法，这个transformIndexHtml是createDevHtmlTransformFn的返回值。当执行transformIndexHtml时内部会调用

applyHtmlTransforms，通过 injectToHead方法注入一个 script 标签，script 的资源地址就是 client.ts的地址

```
// server/middleware/indexHtml.ts
export function indexHtmlMiddware(
    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();
    };
}

export const createDevHtmlTransformFn = (server: ServerContext) => {
    return (url: string, html: string, originalUrl: string):
Promise<string> => {
        return applyHtmlTransforms(html);
    }
}

// node/plugins/html.ts
export const applyHtmlTransforms = async (
```

```
    html: string,
): Promise<string> => {
    return injectToHead(html);
};

// 插入客户端脚本
// 即在 head 标签后面加上 <script type="module" src="/@m-vite/client">
</script>
const injectToHead = (
    raw: string,
): string => raw.replace(
    /(<head[^>]*>)/i,
    `$1\n  <script type="module" src="${CLIENT_PUBLIC_PATH}">
</script>`
);
```

我们打印一下注入后的 html 字符串内容，可以看到index.html文件加上了热重载文件的引入，当页面执行的时候就会去加载这个 client.ts并创建客户端的 websocket 连接

```ts
/**
 * @author: Zhouqi
 * @description: 处理html的中间件
 */
export function indexHtmlMiddware(
    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);
                console.log(html);
                res.statusCode = 200;
                res.setHeader("Content-Type", "text/html");
                return res.end(html);
            }
        }
        return next();
    };
}      You, 上周 • feat: hmr …
```

问题 终端 输出 GITLENS 调试控制台

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

<head>
  <script type="module" src="/@m-vite/client"></script>
  <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>
```

三、HMR Api

vite在 import.meta对象上内置了一个 hot 热重载对象，这个对象包含了所有热重载api

1. hot.accept(deps, callback) 模块更新时的逻辑
   这个 api 非常重要，它决定了热重载的边界。accept 意为接受，表示接受模块更新。它有三种接受状态： 接受自身更新、接收某个子模块更新、接受多个子模块更新
   - 接受自身更新：当前模块会被认为是 hmr 边界，其它模块不会受到影响。使用方式：

```
// parent.tsx
// 边界守卫
if (import.meta.hot) {
  import.meta.hot.accept(()=>{
    console.log('update');
  });
}
```

当自身模块内容发生变化时会触发accept回调函数，打印update

- 接受某个子模块更新：当子模块内容变化时，当前父模块会被认为是hmr边界。使用方式：

```
// parent.tsx
import child from './child';
if (import.meta.hot) {
  import.meta.hot.accept('./child', ()=>{
    console.log('update');
  });
}
```

当子模块child发生变化时会触发父模块下的accept回调函数，打印update

- 接受多个子模块更新：当依赖的多个子模块中的某一个变化时，当前父模块会被认为是hmr边界。使用方式：

```
// parent.tsx
import child from './child';
import child1 from './child1'
if (import.meta.hot) {
  import.meta.hot.accept(['./child', './child1'], ()=>{
    console.log('update');
  });
}
```

当子模块child或者child1发生变化时会触发父模块下的accept回调函数，打印update

2. hot.dispose 模块销毁时逻辑

表示当前模块即将更新前，旧的模块需要销毁时的处理。比如在当前模块设置了一个定时器，当模块内容更新时会再次触发 accept 创建一个定时器，此时有两个定时器，实际上之前的定时器应该被销毁，否则这个 span 里面的内容会混乱

```
// html  <span id="span"></span>

if (import.meta.hot) {
  let timer
  import.meta.hot.accept( ()=>{
  let num = 0;
  timer = setInterval(()=>{
        document.getElementById('span').innerText = num++;
    })
  });

  import.meta.hot.dispose(()=>{
      clearInterval(timer);
  })
}
```

3. hot.data 模块数据共享
   在前面的例子中，每次模块更新，num都会从 0 开始技术，丢失了之前 num 值。针对这个问题可以用 hot.data 来解决。我们可以在hot.data上定义 count 变量，这样每次修改 data.count 的值即可。当模块更新时，hot.data不会因为更新而重置，因此可以从 hot.data上获取之前的 count 值，类似全局状态管理

四、HMR热重载对象注入

前面提到，在代码文件中我们可以通过import.meta.hot.xxx方式去使用hmr api。import.meta是浏览器在ESM模式下内置的一个对象，但是这个 hot 并不是内置的属性，因此 vite 需要在import.meta上注入 hot 对象

在importAnalysisPlugin中有一个变量hasHMR，用来标识当前文件模块是否使用了 hmr api。当解析到代码中使用了 import.meta.hot 时，这个hasHMR就会变成 true。当hasHMR为 true时就会通过prepend方法在顶部追加一段 hmr api的引入代码。这里的 clientPublicPath 就是client/client.ts，也就是客户端的 websocket 文件，在这个文件中定义了 createHotContext用以生成 hot 对象，再将 hot 对象绑定到 import.meta.hot 上，达到注入的效果

```
// node/plugins/importAnalysis.ts
export function importAnalysisPlugin(config: ResolvedConfig): Plugin {
    // 省略其它代码
    return {
        name: "m-vite:import-analysis",
        async transform(code: string, importer: string) {
            // 省略其它代码
            let hasHMR = false;
            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);
                // 判断模块内部是否用了hmr api
                if (rawUrl === "import.meta") {
                    const prop = code.slice(modEnd, modEnd + 4);
                    if (prop === '.hot') {
                        hasHMR = true;
                        if (code.slice(modEnd + 4, modEnd + 11) ===
'.accept') {
                            // 解析aceept接受的依赖
                            if (
                                lexAcceptedHmrDeps(
                                    code,
```

```
                                    code.indexOf('(', modEnd + 11) + 1,
                                    acceptedUrls,
                                )
                            ) {
                                // 接受自身更新
                                isSelfAccepting = true;
                            }
                        }
                    }
                    // 省略其它代码
                }

                // 只对使用了hmr api的模块进行处理
                if (hasHMR) {
                    // 注入 HMR 相关的工具函数
                    str().prepend(
                        `import { createHotContext as
__vite__createHotContext } from "${clientPublicPath}";` +
                        `import.meta.hot =
__vite__createHotContext(${JSON.stringify(importerModule.url)});`,
                    )
                }
                // 省略其它代码
            },
        };
}

// clientPublicPath = @vite/client/client.ts

// client/client.ts
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) {
            // 处理不同的deps类型
            if (typeof deps === "function" || !deps) {
                // import.meta.hot.accept()
                // 接受自身热更新
                acceptDeps([ownerPath], ([mod]: any) => deps?.(mod));
            } else if (Array.isArray(deps)) {
                acceptDeps(deps, callback);
            } else if (typeof deps === 'string') {
                acceptDeps([deps], callback);
            }
        },
        // 模块不再生效的回调
        // import.meta.hot.prune(() => {})
        prune(cb: (data: any) => void) {
            pruneMap.set(ownerPath, cb);
        },
    };
};
```

五、整体更新流程

当本地文件改动时会触发 change 事件：

1. 清除模块依赖图中当前文件模块的缓存，因为文件内容发生了变动，所以要清除之前的缓存信息
2. 调用handleHMRUpdate方法向客户端发送更新信息。在handleHMRUpdate方法中会判断配置文件是否变动，如果变动了直接重启服务，但是不需要像 webapck 一样手动重启。最后调用updateModules向客户端发送 websocket 消息

```
// node/server/hmr.ts
watcher.on("change", async (file) => {
        file = normalizePath(file);
        const { moduleGraph } = serverContext;
        // 清除模块依赖图中的缓存
        await moduleGraph.onFileChange(file);
        // 向客户端发送更新信息
        await handleHMRUpdate(file, serverContext);
});

const handleHMRUpdate = async (file: string, serverContext:
ServerContext) => {
    const { config, moduleGraph } = serverContext;
    const shortFile = getShortName(file, config.root);
    // 是否是配置文件有改动
    const isConfig = file === config.configFile;
    if (isConfig) {
        // 重启服务
        console.log(`${blue("[config change]")} ${green(shortFile)}`);
        // todo 服务重启
        return;
    }
    const mod = moduleGraph.getModuleById(file);
    console.log(`✨${blue("[hmr]")} ${green(shortFile)} changed`);
    updateModules(file, mod, serverContext);
}
```

```
// node/server/moduleGraph.ts
onFileChange(file: string) {
        // 根据文件名找到模块节点
        const mod = this.getModuleById(file);
        mod && this.invalidateModule(mod);
}

invalidateModule(mod: ModuleNode) {
        // 更新时间戳
        mod.lastHMRTimestamp = Date.now();
        // 清除代码转换结果
        mod.transformResult = null;
        // 引用当前模块的模块也需要清除缓存
        mod.importers.forEach((importer) => {
            this.invalidateModule(importer);
        });
 }
```

3. 在 updateModules 中会调用propagateUpdate判断是否需要全量刷新。propagateUpdate逻辑做了简化处理，只需要判断模块是否接受自身或者子模块更新，如果接受则不需要全量更新，否则会递归向上查找父模块的更新状态直到顶层模块。如果需要全量刷新，则向客户端发送full-reload标识。否则发起update标识，并将需要更新的模块信息发送给客户端。

```
// node/server/hmr.ts
export const updateModules = (
    file: string,
    mod: ModuleNode | undefined,
    { ws, moduleGraph, root }: ServerContext
): void => {
    if (!mod) {
        console.log(yellow(`no update happened `) + blue(file));
        return;
    }
    // 是否需要全量刷新
```

```
    let needFullReload = false;
    const boundaries = new Set<{ boundary: ModuleNode }>();
    const hasDeadEnd = propagateUpdate(mod, boundaries);
    moduleGraph.invalidateModule(mod);
    if (hasDeadEnd) needFullReload = true;
    // 是否需要全量刷新
    if (needFullReload) {
        console.log(green(`page reload `));
        ws.send({
            type: 'full-reload',
        });
        return;
    }
    ws.send({
        type: "update",
        updates: [
            ...[...boundaries].map(({ boundary }) => ({
                type: `${boundary.type}-update`,
                timestamp: Date.now(),
                path: boundary.url
            }))
        ],
    });
}

const propagateUpdate = (
    node: ModuleNode,
    boundaries: Set<{ boundary: ModuleNode }>
) => {
  // 接受自身更新是不需要全量刷新的
    if (node.isSelfAccepting) {
        // 添加边界信息
        boundaries.add({
            boundary: node,
            acceptedVia: node,
        });
```

```
            return false;
        }
        // 已经达到顶层模块
        if (!node.importers.size) return true;
        // 向上查找父模块的接受状态
        for (const importer of node.importers) {
            // 父模块中有接受自身更新的情况，需要把父模块添加到边界中
            if (importer.acceptedHmrDeps.has(node)) {
                boundaries.add({
                    boundary: importer,
                    acceptedVia: node,
                });
                continue;
            }
            if (propagateUpdate(importer, boundaries)) return true;
        }
        return false;
    }
```

这里涉及到两个点：

- 接受自身更新的标记isSelfAccepting，这个标记是在importAnalysisPlugin插件中赋值的。在进行 import 分析时会判断是否使　　用了import.meta.hot.accept方法，并调用lexAcceptedHmrDeps方法分析 accept 函数的第一个参数，前面讲到 accept 的第一个参数代　表当前模块接受哪些模块的更新，通过分析 accept 第一个参数字符串可以知道是否是一个接受自身更新的模块。如果第一个参数不传或者传入的字符串中包含当前模块的字符串则表示接受自身更新
- 判断引入当前模块的模块是否接受当前模块acceptedHmrDeps。通过lexAcceptedHmrDeps进行 accept 第一个参数的分析时能够得到模块接受哪些依赖的更新，并且为了保证路径的准确，需要对原始路径进行 resolve 后替换。比如accpet('./App')，需要把./App转换成/src/App.tsx，当进行propagateUpdate的时候就会判断引用当前模块的模块是否接受当前模块的更新

```
    export function importAnalysisPlugin(config: ResolvedConfig): Plugin {
      // 省略其它代码
```

```
    return {
        // 省略其它代码
        async transform(code: string, importer: string) {
            // 省略其它代码
            const [imports] = parse(code);
            let isSelfAccepting = false;
            const acceptedUrls = new Set<{
                url: string
                start: number
                end: number
            }>();
            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);
                // 判断模块内部是否用了hmr api
                 if (rawUrl === "import.meta") {
                    const prop = code.slice(modEnd, modEnd + 4);
                    if (prop === '.hot') {
                        hasHMR = true;
                        if (code.slice(modEnd + 4, modEnd + 11) ===
'.accept') {
                            // 解析aceept接受的依赖
                            if (
                                lexAcceptedHmrDeps(
                                    code,
                                    code.indexOf('(', modEnd + 11) + 1,
                                    acceptedUrls,
                                )
                            ) {
                                // 接受自身更新
                                isSelfAccepting = true;
                            }
                        }
                    }
```

```
                }
            }
            // 省略其它代码
            // 对热更新 accept 中的 url 做处理
            const normalizedAcceptedUrls = new Set<string>()
            for (const { url, start, end } of acceptedUrls) {
                const [normalized] = await moduleGraph.resolveUrl(
                    toAbsoluteUrl(markExplicitImport(url)),
                )
                normalizedAcceptedUrls.add(normalized)
                str().overwrite(start, end, JSON.stringify(normalized),
{
                    contentOnly: true,
                })
            }
            if (!isCSSRequest(importer)) await
moduleGraph.updateModuleInfo(importerModule, importedUrls,
normalizedAcceptedUrls, isSelfAccepting);
        }
```

4. 客户端接受更新消息。当消息类型为full-reload时，页面会调用 location.reload 刷新整个页面。当类型为 update 时会循环updates边界信息，根据对应类型（css or js）走不同的更新逻辑。如果是js更新，则会调用fetchUpdate方法发起import请求获取新的模块内容，然后执行模块绑定的 accept 回调函数

```
// client/client.ts
socket.addEventListener("message", async ({ data }) => {
    handleMessage(JSON.parse(data)).catch(console.error);
});

// 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(async (update: any) => {
                if (update.type === "js-update") {
                    const cb = await fetchUpdate(update);
                    cb!();
                }
            });
            break;
        }
        case "full-reload": {
            // 全量刷新
            location.reload();
            break;
        }
    }
}

async function fetchUpdate({ path, timestamp }: any) {
  const mod = hotModulesMap.get(path);
    if (!mod) return;
    const moduleMap = new Map();
    const [acceptedPathWithoutQuery, query] = acceptedPath.split(`?
`);

    // 从 callbacks 中过滤出需要执行 accept 回调
    const qualifiedCallbacks = mod.callbacks.filter(({ deps }) =>
        deps.includes(acceptedPath),
    );
    try {
        // 通过动态 import 拉取最新模块
        const newMod = await import(
            acceptedPathWithoutQuery + `?t=${timestamp}${query ?
`&${query}` : ""}`
```

```
            );
            moduleMap.set(path, newMod);
        } catch (e) { }

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