########一切不懂就翻vite文档（翻文档）

@@@@！！！！！！！！！！！！

https://cn.vitejs.dev/guide/

vite相对于webpack的优势：

构建工具都是在node环境下去运行的

webpack：如果构建工具webpack需要

处理的代码量大的话，启动项目很慢，他是需要将所有代码解析完才能启动项目，修改文件的时候，项目更新也需要几秒钟才能更新过来，使用HMR也是如此，极大的降低开发效率，

webpack支持多种模块化：因为webpack支持多种模块化, 他一开始必须要统一模块化代码, 所以意味着他需要将所有的依赖全部读一遍，但是vite替代不了webpack 因为webpack的兼容性更好。

vite:由于vite只支持es6 module，不像webpack一样还有commonjs规范，所以他不需要用commonjs规范解析代码，就不需要将所有的代码解析完，所以项目启动很快，但是比不了webpack的兼容性，因为webpack可以用commonjs规范进行多端兼容， 更新代码是秒更新 因为vite是按需更新代码 开发效率大幅度提升 直对浏览器的兼容性更强。

```
// index.js
// 这一段代码最终会到浏览器里去运行
const lodash = require("lodash"); // commonjs 规范
import Vue from "vue"; // es6 module
// webpack是允许我们这么写的
```

```
// webpack的一个转换结果
const lodash = webpack_require("lodash");
const Vue = webpack_require("vue");
```

```
//webpack最终将你的文件编译为：
(function(modules) {
   function webpack_require() {}
   // 入口是index.js
   // 通过webpack的配置文件得来的: webpack.config.js ./src/index.js
   modules[entry](webpack_require);
}, ({
  "./src/index.js": (webpack_require) => {
      const lodash = webpack_require("lodash");
      const Vue = webpack_require("vue");
   }
}))
```

构建工具是运行在服务端的

```
1
```

HMR（热更新）

如果我们运行 例如： yarn create vite 并不是vite在做的事情，而是create vite在做的事情，因为create-vite里面内置了vite，同理: 使用vue-cli里面内置了webpack;
预设：就是在项目搭建的时候 脚手架将项目开发需要的一些依赖都已经搭建好了 例如创建一个vue3的项目需要安装依赖：vite,vue,less,bable这些配置 直接运行create-vite 直接将这些搭建好，这些就是预设。

vite预构建：

vite就是安装就可以用的（开箱即用（out of box ））不需要做任何额外的依赖就可以用vite构建工具来处理构建工作 如果没有安装vite之前，浏览器是不会识别相对路径和绝对路径下的依赖。安装vite以后 vite会对这些相对路径和绝对路径下的配置 进行路径补全 例如 node_modules/.vite/***/***? 浏览器就会识别所需依赖，
vite 搜寻依赖的过程是从当前目录向上查找的过程，直至搜寻到根目录或者查找到对应的依赖为止。
yarn dev 是生产环境运行（每次依赖预构建所重新构建的依赖相对路径都是正确的）
生产环境 vite会全权交给一个rollup的库去完成生产环境的打包（会兼容很多场景和环境和规范）
缓存___>( )
有的项目会用不同的规范 如果不是esmodules规范的话 vite就不识别 浏览器就不会编译 例如（common js(module.exports)规范导出的就不会识别）这种就要用到依赖预构建： （首先 vite会找到对应的依赖，然后调用esbuild（是对js语法进行处理的一个库）：会将其他规范代码转换成esmodules规范的代码，然后放在当前目录下的：node_modules/.vite/deps,同时对esmodule规范各个模块进行统一集成）
optimizeDeps：{
exclude：【 ‘xxxxx’ 】是将xxxxx不用于依赖预构建（exclude vite方法）
}
依赖预构建的3个优点：
1.解决了第三方包会有不同的导出格式（会将其他到出的格式全部转换为esmodules规范格式）
2.在路径的处理上可以直接使用 .vite/deps, 都会集成到这个目录下，方便路径重写（这样子就可以不用在意路径是绝对路径还是相对路径的问题了）
3.解决网络多包传输的性能问题（是原生esmodules规范不敢支持node_module的原因之一），有了依赖预构建以后无论他有多少export，import vite都会尽可能将他集成到一个模块或者几个模块。不然一个模块中会存在很多个 然后所有全部都加载一边，性能会很差。

vite配置文件处理细节：

1.语法提示：

```
第一种方法：
import {defineconfig} from 'vite'//使用这种引入的方法 例如引用defincofig
export default defineconfig({在这一个区域写方法就会有语法提示    因为defineconfig 是一个函数，鼠标放上去会有类型提示和导出的实体对象(TS写法))
    optimizeDeps：{
        exclude:[],   //将指定代码中的依赖不进行依赖预处理
    }
})
第二种方法：不使用import  使用注释 也会进行语法提示
/**@type import("vite").userConfig*/
const viteConfig = {
    optimizeDeps:{
        exculde:[]
```

```
    }
}
export default viteConfig //导出


```

2.环境配置：

会根据当前代码环境产生的值的变化的变量就是环境变量

一般公司代码环境有以下几种： APP_KEY

生产环境，开发环境，灰度环境，测试环境，预发布环境。 env.production_生产环境 env.development_开发环境 env_所有环境变量。

vite内置的第三方库 dotenv 在yarn dev 之类启动项目的时候，vite（包括大量的构建工具）会用这个第三方库（dotenv）用于读到所有的env环境变量配置文件，然后解析文件中对应的环境变量，解析只能将对应的数据解析成字符串，然后dotenv是在node端运行，（例如找到环境变量APP_KEY = 110 因为返回的是字符串，所以他会用原生js的splic用 = 分开 然后形成一个对象），并将其注入到process（是node端关于当前进程的一个对象）中，

但是vite考虑到和其他配置的一些冲突问题，它不会直接注入到process对象下：涉及到一些vite.config.js中的一些配置：

主要是以下两个会影响process vite也给了补救的方法（vite自带的loadEnv来手动确认env文件）

--root，

--envDir（用来配置环境变量文件地址的）

process.cwd:返回当前node进程的目录

.env: 所有环境都需要用到的环境变量

.env.development: 开发环境需要用到的环境变量(默认情况下vite将我们的开发环境取名为development)

.env.production: 生产环境需要用到的环境变量(默认情况下vite将我们的生产环境取名为production)

loadEnv:

1.这个方法是直接找到所有的env文件，并解析环境变量 并放到一个对象里。

2.会将传进的mode的变量的值进行拼接：.env + (mode传进的值) 并根据我们提供的目录去取对应的配置文件方法进行解析，放进一个新的对象里。

```
// 策略模式
const envResolver = {
    "build": () => {
        console.log("生产环境");
        return ({ ...viteBaseConfig, ...viteProdConfig })
    },
    "serve": () => {
        console.log("开发环境");
        return  ({ ...viteBaseConfig, ...viteProdConfig }) // 新配置里是可能会被配置envDir .envA
    }
```

```
}

export default defineConfig(({ command, mode }) => {
    // 是build 还是serve主要取决于我们敲的命令是开启开发环境还是生产环境
    // console.log("process", process.cwd());
    // process.cwd()是当前env文件所在的目录
    // 第二个参数不是必须要使用process.cwd()，可以手写当前的目录路径,parcess.cwd()的意思是获取当前node的执行目录;
    // 第三个参数是env的文件名，env是env名的默认值，可以不传
    const env = loadEnv(mode, process.cwd(), "");
    return envResolver[command]();
})
```

如果比如要在request中区分环境变量（客户端）：

vite提供了一种方法：vite会将对应的环境变量注入到import.meta.env中

如果import.meta.env获取不到环境变量，这是因为vite为了防止我们将隐私性的变量直接送进import.meta.env中，所以做了一次拦截，如果环境变量不是以VITE_开头的，vite就不会将这个环境变量注入到客户端中。

```
import { defineConfig} from 'vite'
export default defineConfig({
    optimizeDeps:{
        exclude:[] //将指定数组中的依赖不进行依赖预构建
    },
    envPrefix:'ENV'// 配置vite注入客户端环境变量校验的env前缀
})
```

为什么vite.config.js可以书写成esmodule规范，因为vite在读取这个vite.config.js文件的时候，它会率先node去解析文件的语法，如果发现你这个是esmodule规范，会直接将这个esmodule规范替换成commonjs规范

```
// 策略模式  如果用着if else判断麻烦  就可以用这个
const envResolver = {
    "build": () => {//如果是build 生产环境下  走这一步
        console.log("生产环境");
        return ({ ...viteBaseConfig, ...viteProdConfig })//然后值
    },
    "serve": () => {
        console.log("开发环境");//如果是开发环境下  走这一步
        return  Object assign({},viteBaseConfig,viteDevConfig) // 新配置里是可能会被配置envDir .envA
        //这个拷贝和上面的...展开运算符是一个意思 最好给一个{}，不然会覆盖掉原有的值
    }
}
```

```
13
14 export default defineConfig(({ command, mode }) => {
15     // 是build 还是serve主要取决于我们敲的命令是开启开发环境还是生产环境
16     // console.log("process", process.cwd());
17     // 当前env文件所在的目录
18     // 第二个参数不是必须要使用process.cwd(),
19     const env = loadEnv(mode, process.cwd(), "");
20     return envResolver[command]();//将需要判断的值传入上面所写的 envResolver判断方法中
21 })
```

vite是怎么让浏览器可以识别.vue文件：

require('koa'),这是commonjs规范的引用， const koa = new koa() ， koa是node的一个框架 写node不能使用esmodule 只能使用commonjs规范

```
1  const Koa = require("koa"); // 不能用esmodule 必须使用commonjs
2  const fs = require("fs");  // ./ / npm install yarn add
3  const path = require("path");// node一个提供路径的方法
4
5  // 读取vite.config.js ??? fs去读取?
6  // 我们不用返回给客户端的吧 而且我们这里约定的名字就叫做vite.config.js
7  const viteConfig = require("./vite.config");
8  const aliasResolver = require("./aliasResolver");
9
10 console.log("vite.Config", viteConfig)
11
12 const app = new Koa();  // const vue = new Vue();
13 app.use(async (ctx) => { //context 上下文 request --> 请求信息 响应信息 get请求 /
14     console.log("ctx", ctx.request, ctx.response);
15     // 用中间件去帮我们读文件就行了
16     if (ctx.request.url === "/") {
17         const indexContent = await fs.promises.readFile(path.resolve(__dirname,
   "./index.html")); // 在服务端一般不会这么用 用node的path找到路径
18         ctx.response.body = indexContent;//作为响应体发给对应请求的人
19         ctx.response.set("Content-Type", "text/html");  //转换的文件内容格式  如果不写这一
   行 浏览器就会识别不出来什么格式
20     }
21
22     if (ctx.request.url === '/App.vue'){ //如果后台返回的路径和/App.vue 一样的话
```

```
        // 然后就从数据库找到这个路径下的信息进行返回给前端
        const mainVueContent = await fs.promises.readFile(path.resolve(__dirname,"./App.vue"))
        //如果是vue文件，会做一个字符串的替换：mainVueContent.toString().find("<template>")，判断vue文件里面有没有这个标签，如果有就进行全部替换
        //然后用AST语法分析 => 再用Vue.createElement()方法 => 构建Dom
        ctx.response.body = mainVueConcent; //作为响应体发给对应请求的人
        console.log(mainVueContent.toString())//会转换为字符串
        //响应体填充好了，要以什么形式返回呢，希望浏览器拿到后用什么方法去解析
        //json => 1.application/json 2.text/html 3.text/javescript
        ctx.response.set("Content-Type",'text/javascript')//替换为原生js的格式 即使是.Vue的文件，也要用javascript的方式去解析
    }

    // 如果当前文件的url是以js后缀结尾的
    // 我们的root js文件一定是和html文件在一个目录的对不对
    if (ctx.request.url.endsWith(".js")) {
        const JSContent = await fs.promises.readFile(path.resolve(__dirname, "." + ctx.request.url)); // 在服务端一般不会这么用
        console.log("JSContent", JSContent);
        // 直接进行alias的替换
        const lastResult = aliasResolver(viteConfig.resolve.alias, JSContent.toString());
        ctx.response.body = lastResult; // 作为响应体发给对应的请求的人
        ctx.response.set("Content-Type", "text/javascript");
    }
})

app.listen(5173, () => {//app.listen类似于 yarn dev 启动项目
    console.log("vite dev serve listen on 5173");
})
```

vite是怎么处理css：

vite天生就支持对css文件直接的处理。

1.vite在读取到main.js中引用到了Index.css

2. 直接去使用fs模块去读取index.css中文件内容

3. 直接创建一个style标签, 将index.css中文件内容直接copy进style标签里

4. 将style标签插入到index.html的head中

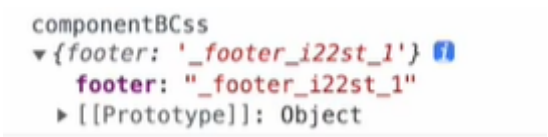
5. 将该css文件中的内容直接替换为js脚本(方便热更新或者css模块化), 同时设置Content-Type为 js 从而让浏览器以JS脚本的形式来执行该css后缀的文件。

如果类名重复 最终会导致后面那个类名的样式会覆盖前面那个类名的样式，在协同开tahui发下很容易出问题。有一种处理方法 （cssmodule方法）

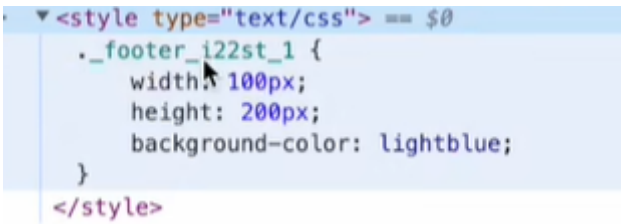
全部都是基于node

1. module.css (module是一种约定, 表示需要开启css模块化) 就会将.module.css后缀的样式文件进行额外处理

2. 他会将你的所有类名进行一定规则的替换（将footer 替换成 _footer_i22st_1）



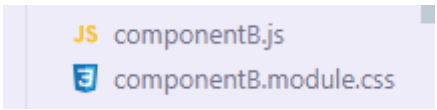
3. 同时创建一个映射对象{ footer: "_footer_i22st_1" }



4. 将替换过后的内容塞进style标签里然后放入到head标签中 (因为能够读到index.html的文件内容)

5. 将componentB.module.css内容进行全部抹除, 替换成JS脚本

5. 将创建的映射对象在脚本中进行默认导出



commponentB.module.css里的样式

```css
.footer {
    width: 100px;
    height: 200px;
    background-color: lightblue;
}
```

componentB.js 文件里的内容

```js
import componentBCss from "./componentB.module.css";//引用有着cssmodule规范的样式文件,
console.log("componentBCss", componentBCss);
const div = document.createElement("div");//选择div标签
document.body.appendChild(div);//在div标签放进body
div.className=componentBCss.footer;//然后获取到cssmodule规范的样式名称，这个是所有类名进行一定规则的替换以后，得到的一个一个映射对象的名称（ _footer_i22st_1）
```

就像这个css文件 需要用到cssmodule规范时（避免类名重复），就加一个.module.css 后缀，然后cssmodule规范会将当前css样式文件里的类名进行一定规则的替换，然后创建一个映射对象，将替换过后的内容放进style标签中

去， 然后再放到head标签中去，然后将componentB.module.css 内容全部抹除，转换成字符串，在替换为JS脚本，然后将创建的映射对象在脚本中默认导出。

vite.config.js中css配置（modules篇）：

在vite.config.js中我们通过css属性去控制整个vite对于css的处理行为
- localConvention: 修改生成的配置对象的key的展示形式(驼峰还是中划线形式)
- scopeBehaviour: 配置当前的模块化行为是模块化还是全局化 (有hash就是开启了模块化的一个标志, 因为他可以保证产生不同的hash值来控制我们的样式类名不被覆盖)
- generateScopedName: 生成的类名的规则(可以配置为函数, 也可以配置成字符串规则: https://github.com/webpack/loader-utils#interpolatename)
- hashPrefix: 生成hash会根据你的类名 + 一些其他的字符串(文件名 + 他内部随机生成一个字符串)去进行生成, 如果你想要你生成hash更加的独特一点, 你可以配置hashPrefix, 你配置的这个字符串会参与到最终的hash生成, （hash: 只要你的字符串有一个字不一样, 那么生成的hash就完全不一样, 但是只要你的字符串完全一样, 生成的hash就会一样）
- globalModulePaths: 代表你不想参与到css模块化的路径

```
import { defineConfig } from "vite";
import path from "path";

export default defineConfig({
    optimizeDeps: {
        exclude: [], // 将指定数组中的依赖不进行依赖预构建
    },
    envPrefix: "ENV_", // 配置vite注入客户端环境变量校验的env前缀
    css: { // 对css的行为进行配置
        // modules配置最终会丢给postcss modules
        modules: { // 是对css模块化的默认行为进行覆盖
            localsConvention: "camelCaseOnly", // 修改生成的配置对象的key的展示形式(驼峰还是中划线形式)
            scopeBehaviour: "local", // 配置当前的模块化行为是模块化还是全局化 (有hash就是开启了模块化的一个标志, 因为他可以保证产生不同的hash值来控制我们的样式类名不被覆盖)
            // generateScopedName: "[name]_[local]_[hash:5]" // https://github.com/webpack/loader-utils#interpolatename
            // generateScopedName: (name, filename, css) => {
            //     // name -> 代表的是你此刻css文件中的类名
            //     // filename -> 是你当前css文件的绝对路径
            //     // css -> 给的就是你当前样式
            //     console.log("name", name, "filename", filename, "css", css); // 这一行会输出在哪？？？ 输出在node
            //     // 配置成函数以后, 返回值就决定了他最终显示的类型
```

```
            //     return `${name}_${Math.random().toString(36).substr(3, 8) }`;
            // }
            hashPrefix: "hello", // 生成hash会根据你的类名 + 一些其他的字符串(文件名 + 他内部随机生成一个字符串)去进行生成，如果你想要你生成hash更加的独特一点，你可以配置hashPrefix，你配置的这个字符串会参与到最终的hash生成，（hash：只要你的字符串有一个字不一样，那么生成的hash就完全不一样，但是只要你的字符串完全一样，生成的hash就会一样）
            globalModulePaths: ["./componentB.module.css"], // 代表你不想参与到css模块化的路径
        },
    },
    resolve: {
        alias: {
            "@": path.resolve(__dirname, "./src"), // 设置别名，以后我们在其他组件中可以使用@来代替src这个目录
        }
    },
    build: { // 构建生产包时的一些配置策略
        rollupOptions: { // 配置rollup的一些构建策略
            output: { // 控制输出
                // 在rollup里面，hash代表将你的文件名和文件内容进行组合计算得来的结果
                assetFileNames: "[hash].[name].[ext]"
            }
        },
        assetsInlineLimit: 4096000, // 4000kb
        outDir: "dist", // 配置输出目录
        assetsDir: "static", // 配置输出目录中的静态资源目录
        emptyOutDir: true, // 清除输出目录中的所有文件
    }
});
```

vite配置文件中css配置流程（preprocessorOptions）：

preprocessorOptions 主要是用来配置css预处理的一些全局参数

```
import { defineConfig } from "vite";
import { ViteAliases } from "vite-aliases";
import viteCompression from 'vite-plugin-compression';
export default defineConfig({
    optimizeDeps: {
        exclude: [], // 将指定数组中的依赖不进行依赖预构建
    },
```

```
    envPrefix: "ENV_",
    css: { // 对css的行为进行配置
        preprocessorOptions: { // key + config key代表预处理器的名 就是如果需要处理的预处理器是less就跟如下是less   如果是sass就是sass
            less: { // 整个的配置对象都会最终给到less的执行参数（全局参数）中去
                // 在webpack里就给less-loader去配置就好了
                math: "always",//https://less.bootcss.com/usage/#less-options-math。更多可以使用的方法可以在这个less官网找
                globalVars: { // 全局变量，需要定义的变量
                    mainColor: "red",// 在别的css文件里 如果用到这个颜色，直接使用：@mainColor 就是red。
                }
            },
        },
        devSourcemap: true,//文件之间的索引：如果我们将文件压缩或者编译过了，然后这个时候程序出错，他就不会产生错误的位置信息，如果设置了Sourcemap，他就会有一个索引的map，提供路径。
    },
});
```

##postcss：

vite 天生就对postcss有非常良好的支持；保证css在执行起来万无一失。（postcss的级别要高于一系列less，sass预处理器）

但是postcss中的less，sass等一系列预处理器已经停止维护了：（之前是完全走的一套，想完全替换掉less，sass这些预处理器，但是less这些预处理器大部分受众了，postcss就打算用postcss-less 插件解决这些问题，就是你用我的postcss在加一个postcss-less相关的一个less插件，然后我再帮你编译这个less语法，然后发现一件事情，就是发现这个postcss中的less或者sass一但更新，他对应的关于postcss相关的一些less插件或者对于postcss相关的一些sass插件都要跟随的去更新，不然就处理不了，慢慢的就感觉没有什么必要，更新起来也很累，慢慢的跟postcss一些相关的预处理器插件就停止维护了。）--------现在就比较佛系，你自己去用less和sass去编译完了然后你把编译结果给我（原生的css文件），但是less和sass还有些东西处理不了（最新css语法降级，自动前缀补全），这些东西还是postcss去做，但是less相关的事情就less自己去做。所以 postcss是后处理器（在less和sass处理完后，在交给postcss去处理，在之后处理的所以叫后处理器）（postcss处理器挺强的，就是经费不充足，如果postcss成本足的话，他可以直接将所有的流程都走完）。如果想postcss支持less的话，就写less的postcss的插件。

postcss处理流程：我们写的css代码(怎么爽怎么来) --> postcss ---> [去将语法进行编译（嵌套语法，函数，变量）成原生css] less，sass 等预处理器都可以做--> 再次对未来的高级css语法进行降级 --> 前缀补全 --> 浏览器客户端 。

就相当于：我们写的css代码(怎么爽怎么来) --> postcss ---> less --> 再次对未来的高级css语法进行降级 --> 前缀补全 --> 浏览器客户端。

postcss是处理css就跟babel处理js一样（babel帮助我们让js执行起来万无一失）：我们写的js代码(怎么爽怎么来) --> babel --> 将最新的ts语法进行转换js语法 --> 做一次语法降级 --> 浏览器客户端去执行。

```
```js     babel语法降级
class App {} // es6的写法

function App() {} // es3的语法
```

对postcss有一个误区: 他们认为postcss和less sass是差不多级别 其实是postcss包含了less这些预处理器（一样可以做将语法进行编译（嵌套语法，函数，变量）成原生css），比less可以做更多的事情。

浏览器的兼容性考虑不到, 预处理器并不能够解决这些问题:

1. 对未来css属性的一些使用降级问题
2. 前缀补全: Google非常卷 --webkit

如果需要使用postcss：https://www.npmjs.com/package/postcss-cli //postcss npm文档

```
1. 安装依赖
```
yarn add postcss-cli //提供这个脚手架一些命令
yarn add postcss -D   //做编译工作
```

2. 书写描述文件
postcss配置文件的格式
- postcss.config.js（可以百度配置）
```

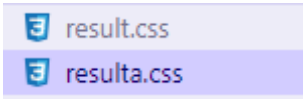
postcss.config.js 文件内容：

plugins里的配置方法：（关键字：postcss-plugins list）https://github.com/postcss/postcss/blob/main/docs/plugins.md

postcss的预设环境：（关键字: postcss-preset-env)https://github.com/csstools/postcss-plugins/tree/main/plugin-packs/postcss-preset-env yarn add post-preset-env -D (预设环境里面是会包含很多的插件)

```
// 就类似于全屋净水系统的加插槽(中间件)
// 预设环境里面是会包含很多的插件
// 语法降级 --> postcss-low-level
// 编译插件 --> postcss-compiler
// ...
const postcssPresetEnv = require("postcss-preset-env");
// 预设就是帮你一次性的把这些必要的插件都给你装上了
// 做语法的编译 less语法 sass语法 （语法嵌套 函数 变量) postcss的插件 -->
```
```

```
module.exports = {
    plugins: [postcssPresetEnv(/* pluginOptions */), ]  //插件  如果再加插件就用，隔开。插件越多，postcss能做的事情就越多。
}
```



result.css:

```
:root { //:root 自定义变量全局使用  例：--globalColor：lightblue
    --globalColor: lightblue;
}

div {
    background-color: lightblue;
    background-color: var(--globalColor);//使用就是 var(--globalColor)
    width: max(200px, min(50%, 200px));
    filter: blur(30px);
}
```

resulta.css:

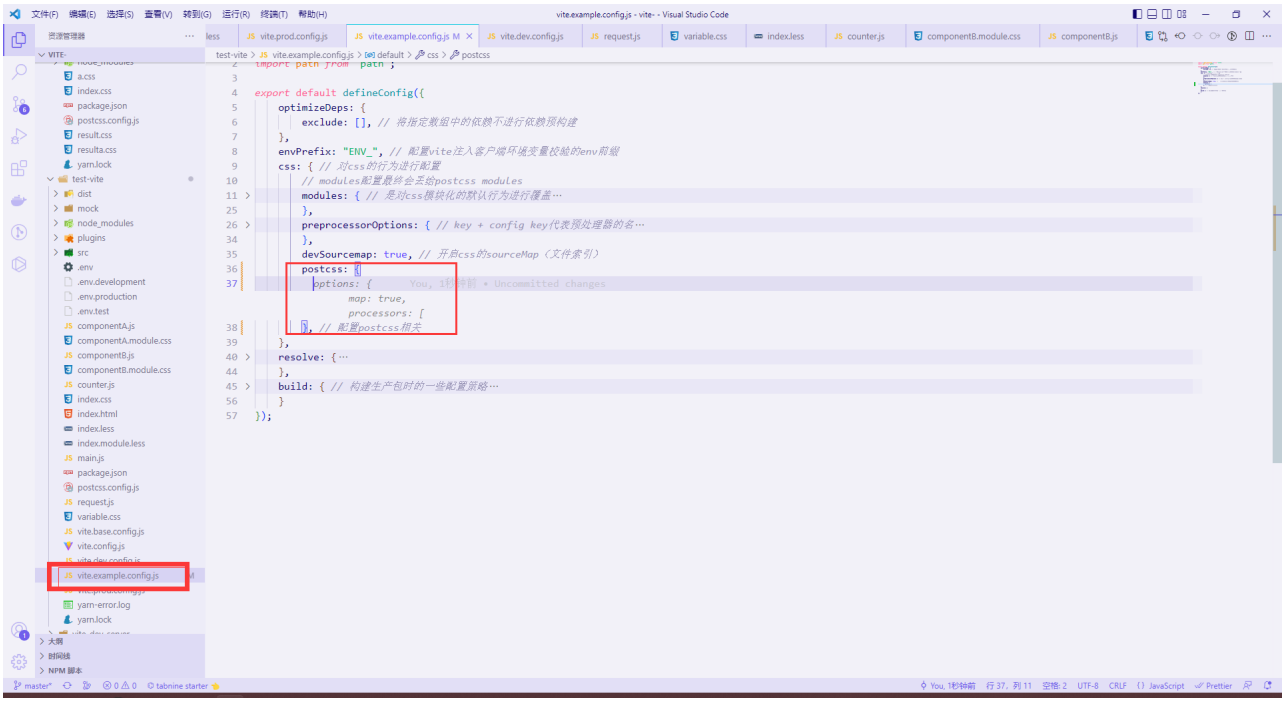
```
div {
    width: 100px;
    background-color: var(--globalColor);
}
```

##vite配置文件config.js中css配置流程（postcss篇）：

直接在css.postcss中进行配置，该属性直接配置的就是postcss的配置；

postcss-preset-env: 预设，支持css变量和一些未来css语法 自动补全(autoprefixer)

第一种，可以在这里面进行配置：



先跑命令 yarn add postcss-plugins-env -D (安装postcss预设)

```
import { defineConfig } from "vite";
import { ViteAliases } from "vite-aliases"

const postcssPresetEnv = require("postcss-preset-env");//导入postcssPresetEnv预设
export default defineConfig({
    css: { // 对css的行为进行配置
        postcss:{//vite可以让postcss在火一次
            plugins:[postcssPresetEnv())]//如果你配置不想用（就是不想单独在使用一些配置）就可以这样子写。
        }
    },
});
```

也可以另一种写法 就是创建一个postcss.config.js 文件 但是优先级没有上面 postcss：{} 高

postcss.config.js

vite一样可以识别

```
const postcssPresetEnv = require("postcss-preset-env");
// 预设就是帮你一次性的把这些必要的插件都给你装上了
// 做语法的编译 less语法 sass语法 （语法嵌套 函数 变量) postcss的插件 -->
module.exports = {
    plugins: [postcssPresetEnv(/* pluginOptions */), ]  //插件  如果再加插件就用，隔开。插件越多，postcss能做的事情就越多。
}
```

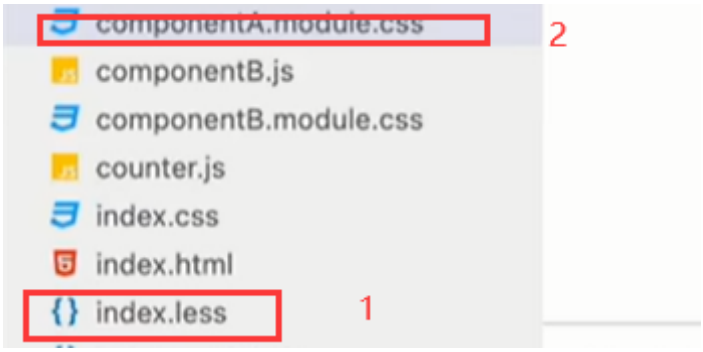
index.module.less

```less

.content {
    width: 800px;
    .main {
        background: green;
        padding: (100px / 2);
        margin: 100px / 2;//如果用less等预处理器；功能就和上面加了（）一样
        background-color: red;
        filter: blur(100px);//(滤镜，一般用于视觉效果)//这一些用法:filter: none | blur() | brightness() | contrast() | drop-shadow() | grayscale() | hue-rotate() | invert() | opacity() | saturate() | sepia() | url();
        // 响应式布局，左侧一个菜单栏 宽度自适应根据屏幕 30%
        // preset-env会帮助我们做语法降级根据vite内部会有一个主流浏览器的支持表（语法降级会查当前浏览器的标准，如果当前属性所有浏览器都能支持了，就不会编译了，例如：clamp所有浏览器都能支持的话，就不会编译了）
        （编译到所有浏览器都支持的属性，例如：clamp 编译后就是max，min，因为只能编译到这个地步所有浏览器都支持）
        // clamp（新属性，有些浏览器还不能识别）
        width: clamp(100px, 30%, 200px);//可以输入三个参数（最小宽，宽度，最大宽）//响应式布局，左侧一个菜单栏 宽度自适应根据屏幕 30% //通过preset-env语法降级就会变成:width:max(100px,min(30%,200px)))
        user-select: none; // 他在其他浏览器上不支持//所以最低兼容的浏览器前就会加前缀//没有加前缀的浏览器就是支持的，相当自信；如下图 这个属性前加了-webkit（谷歌）-moz(火狐)
        //如果以后这些浏览器支持了这些之前最低兼容的属性，这个前缀就会移除掉，因为vite内部有个主流浏览器支持表。从中查看。如果兼容了就不会有前缀（-webkit）
    }
```

```less
._content_17yto_1 ._main_17yto_4 {
  background: green;
  padding: 50px;
  margin: 50px;
  background-color: red;
  filter: blur(100px);
  width: max(100px, min(30%, 200px));
  -webkit-user-select: none;
     -moz-user-select: none;
          user-select: none;
}
```

##关于postcss为什么没有编译全局css变量：

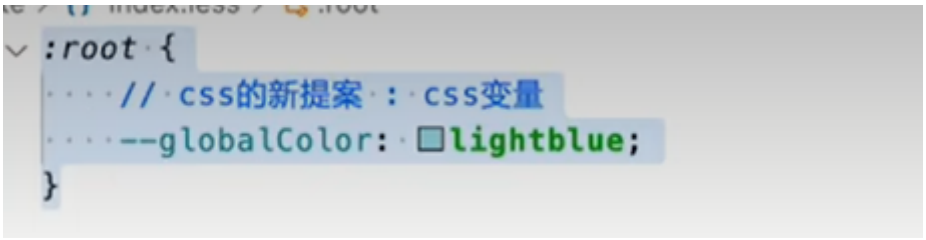
postcss是挨个挨个文件去编译的.



它先将less文件解析完就不解析了，然后到module.css文件的时候，他重新解析了一遍，但是之前less解析的文件就丢了。

我们使用的一些未来的新特性是不需要通过less，sass预处理器来编译的，只能交给postcss来处理。

所以就不需要将后缀写成.less，不然就是浪费性能，因为less也编译不了。



```
:root {
    // css的新提案 : css变量
    --globalColor: lightblue;
}
```

为了避免这种，这种全局变量最好用css后缀命名文件。创建一个.css的文件专门用于装全局css变量（'variable.css'自定义文件名称，如下图）

然后在postcss.config.js里修改

```
const postcssPresetEnv = require("postcss-preset-env")
const path = require("path"); // 做路径处理的

module.exports = {
    plugins: [
        postcssPresetEnv({
            importFrom: path.resolve(__dirname, "./variable.css"), // 就好比你现在让postcss去知道 有一些全局变量他需要记下来
        })//在未来 这个importFrom可能被移除。暂时官方没有替换方案//https://github.com/postcss/postcss-custom-properties
    ]
}
```

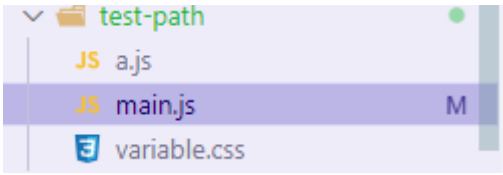
##为什么我们在服务端处理路径的时候一定要用到path：

在这里配置了后：

1.他会去node端读这个文件,然后会去找这个文件在哪，

2.如果写的是相对路径，那么它会尝试去拼接成绝对路径

node是common.js规范，会注入几个变量（__dirname）




```
const fs = require("fs"); // 处理文件的模块，里面有（读文件，修改文件等一系列操作）
const path = require("path"); // path本质上就是一个字符串处理模块，它里面有非常多的路径字符串处理方法
require("./a");
//path.resolve在拼接字符串
//fs.readFileSync（是以同步的方法读文件）                    👇//在这里写相对路径都是没有问题的，因为__dirname始终返回的是当前文件所在的目录
const result = fs.readFileSync(path.resolve(__dirname, "./variable.css")); // 我们希望基于main.js去进行一个绝对路径的生成，所以会用到path；
// __dirname: 含义是始终返回的是当前文件所在的目录
// console.log("result", result.toString(), process.cwd(), __dirname + "/variable.css", path.resolve(__dirname, "./variable.css"));
// node端去读取文件或者操作文件的时候，如果发现你用的是相对路径，则会去使用process.cwd()来进行对应的拼接，我们希望基于main.js去进行一个绝对路径的生成，所以会用到path；
// process.cwd: 获取当前的node执行目录
```

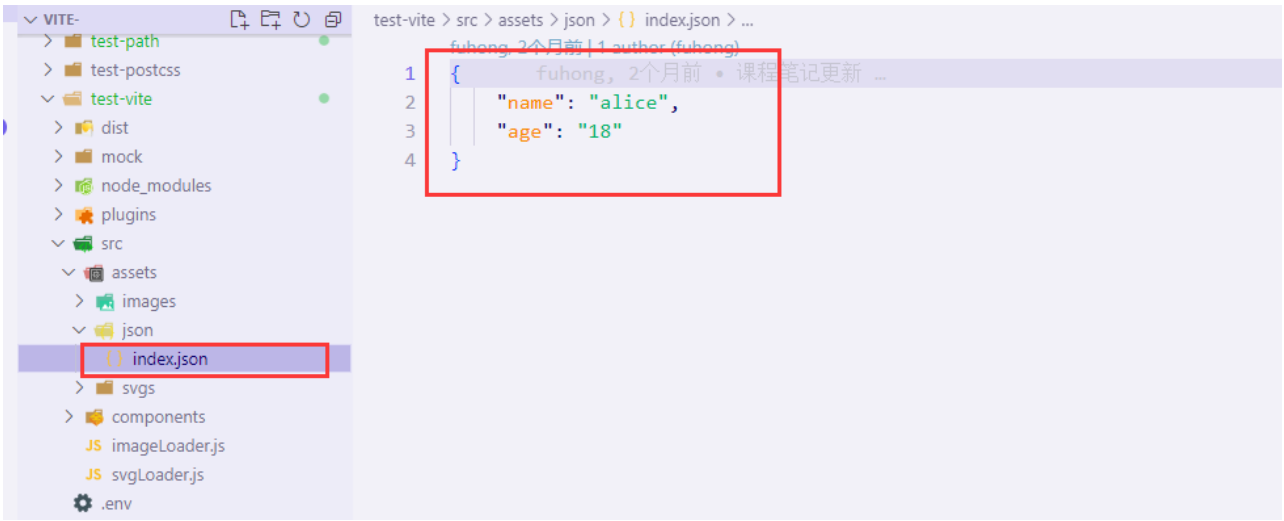
```
console.log('__dirname',__dirname) //为什么没有声明任何东西，还能将__dirname读出来 //这个涉及到commonjs规范的一个原理
//common.js 实现模块化；它可以将所有的js文件，通过fs读出来，读出来了以后它将文件内容复制一下，然后放到一个立即执行函数里面（如下函数）
//arguments 获取全部参数
//console.log('arguments',arguments)后获取参数
//          0        1        2          3          4
(function(exports, require, module, __filename, __dirname) {//立即执行函数，立即函数是可以隔绝作用域的，实现了模块化
  require("")
  console.log("__dirname", __dirname);
}())

//立即执行函数里面有五个参数 {},require,module,__filename，__dirname
//exports = module.exports = {}（第0位）
//__filename 是当前文件的绝对路径
//__dirname: 始终返回的是当前文件所在的目录
```

```
15
```

#vite加载静态资源

静态资源：图片视频等等；但是在服务端除了动态Api以外，所有的资源都是静态资源,除了动态API以外, 百分之九十九资源都被视作静态资源 API --> 来了一个请求 /getUserName

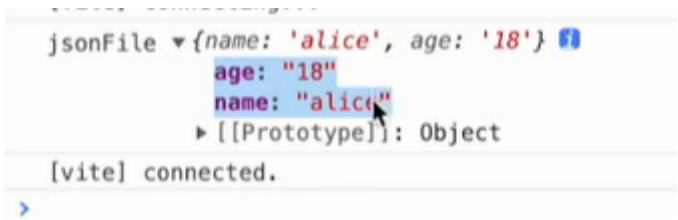
vite对静态资源基本上是开箱即用的,svg也是



可以直接创建json文件， 然后在main.js里导入

```
1 import jsonFile from "./src/assets/json/index.json";//这样子是json全部导入,如果json里的数据多的话，几百条的话,vite就会认为你所有的json都需要用，就不能进行摇树优化。因为你都用到整个json对象了。
2 console.log("jsonFile", jsonFile);// 如果你用的不是vite，在其他的一些构建工具里 json文件的导入会作为一个JSON字符串形式存在
3 import { name } from "./src/assets/json/index.json"; // 也可以这样子写，只要文件里面的name，这样子写的好处：tree shaking摇树优化。就是你引用了name，vite就会认为你不需要别的age，打包的时候他就会将这个vite删掉，提高性能
4
```

就会在console控制台里得到一个对象形式的json，



假如企业级的项目：如果生产环境的代码非常臃肿和性能差，就可以控制导入。使用摇树优化（打包工具会自动帮你移除掉你没有用到的那些变量和方法）

流媒体：vidio标签里的src不是mp4，而是一个rtmp

```
1 import svgIcon from "./assets/svgs/fullScreen.svg?url";//后面可以加url 默认就是url;
2 import svgRaw from "./assets/svgs/fullScreen.svg?raw";//后面可以加raw 返回的是Buffer;服务端，他需要加载这个图片的时候，他会去读取图片的内容，读到的是Buffer,二进制的字符串
```

路径别名配置：

```
import {defineConfig} from 'vite'
const path = require('path')

export default defineConfig({
    resolve:{
        alias:{ //vite在浏览器上会自动将路径别名转换为绝对路径;   字符串replace操作
           '@': path.resolve(__dirname, './src')
           '@assets': path.resolve(__dirname,'./assets/images') //图片路径
        }
    }
})
```

#resolve.alias原理

```
const Koa = require("koa"); // 不能用esmodule 必须使用commonjs
const fs = require("fs");  // ./ / npm install yarn add
const path = require("path");

// 读取vite.config.js ??? fs去读取？我们不用返回给客户端的（就不用fs） 而且我们这里约定的名字就叫做vite.config.js（已经固定了）
const viteConfig = require("./vite.config");//引用vite.config,直接读，不用fs
console.log("vite.Config", viteConfig)

const aliasResolver = require("./aliasResolver");//（看下一张图片）

const app = new Koa();  // const vue = new Vue();
app.use(async (ctx) => { //context 上下文 request --> 请求信息 响应信息 get请求 /
    console.log("ctx", ctx.request, ctx.response);
    // 用中间件去帮我们读文件就行了
    if (ctx.request.url === "/") {
        const indexContent = await fs.promises.readFile(path.resolve(__dirname, "./index.html")); // 在服务端一般不会这么用
        ctx.response.body = indexContent;
        ctx.response.set("Content-Type", "text/html");
    }
    // 我们的root js文件一定是和html文件在一个目录的对不对
    if (ctx.request.url.endsWith(".js")) {// 如果当前文件的url是以js后缀结尾的,所有js文件都走这个流程
```

```
        const JSContent = await fs.promises.readFile(path.resolve(__dirname, "." + ctx.request.url)); // 在服务端一般不会这么用
        console.log("JSContent", JSContent);                                    👆//路径处理报错，加个'.' 就可以
        // 直接进行alias的替换
        const lastResult = aliasResolver(viteConfig.resolve.alias, JSContent.toString());
        ctx.response.body = lastResult; // 作为响应体发给对应的请求的人
        ctx.response.set("Content-Type", "text/javascript");
    }
})

app.listen(5173, () => {
    console.log("vite dev serve listen on 5173");
})
```

JS aliasResolver.js

```
module.exports = function(aliasConf, JSContent) {
    const entires = Object.entries(aliasConf);//es6的写法  Object.entries()可以把一个对象的键值以数组的形式遍历出来，结果和for...in 一致，但不会遍历原型属性。
    console.log("entires", entires);//返回的是[['@','路径']]
    let lastContent = JSContent;// JScontnet返回的是个 Buffer
    entires.forEach(entire => {
        const [alia, path] = entire;
        // 这里是直接找的src，但是vite会做path的相对路径的处理
        const srcIndex = path.indexOf("/src");
        // alias别名最终做的事情就是一个字符串替换
        const realPath = path.slice(srcIndex, path.length);
        lastContent = JSContent.replace(alia, realPath);//replace 替换 第一个参数是要替换的地方，第二个参数是需要替换的值

    })
    console.log("lastContent.........", lastContent);
    return lastContent;
}
```

indexOf

```
String s = "01234560123456";
int a = s.indexOf('1'); // 返回第一个字符1的下标                1
int b = s.indexOf("23"); // 返回第一个字符串“23”的下标          2
```

```
int c = s.indexOf('1',5); // 以下标5开始，返回第一个字符1的下标      8
int d = s.indexOf("23",5); // 以下标5开始，返回第一个字符串“23”的下标     9
```

#vite处理svg资源

svg: scalable vector graphics 可伸缩矢量图形 （1.svg图是不会失真 2.尺寸小） 缺点：没法去表示层次丰富的图片信息 前端里更多是用svg做图标

先创建一个svg图


```
<?xml version="1.0" standalone="no"?><!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"><svg t="1660369691838" class="icon" viewBox="0 0 1024 1024" version="1.1" xmlns="http://www.w3.org/2000/svg" p-id="7114" xmlns:xlink="http://www.w3.org/1999/xlink" width="200" height="200"><defs><style type="text/css">@font-face { font-family: feedback-iconfont; src: url("//at.alicdn.com/t/font_1031158_u69w8yhxdu.woff2?t=1630033759944") format("woff2"), url("//at.alicdn.com/t/font_1031158_u69w8yhxdu.woff?t=1630033759944") format("woff"), url("//at.alicdn.com/t/font_1031158_u69w8yhxdu.ttf?t=1630033759944") format("truetype"); }
</style></defs><path d="M768 298.656h170.656V384h-256V128H768v170.656zM341.344 384h-256V298.656H256V128h85.344v256zM768 725.344V896h-85.344v-256h256v85.344H768zM341.344 640v256H256v-170.656H85.344V640h256z" p-id="7115"></path></svg>
```

然后引用： main.js 也要引用。

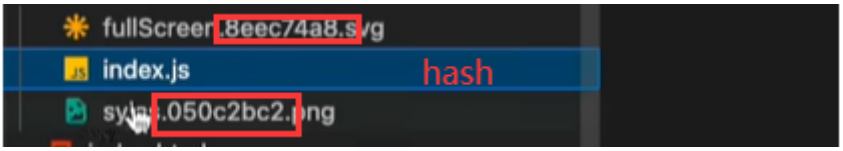
```
import svgIcon from "./assets/svgs/fullScreen.svg?url";// 可以更改？后  默认是url  单纯是图片，更改不了样式，比如颜色
import svgRaw from "./assets/svgs/fullScreen.svg?raw";// 更改成raw后 直接是二进制字符串 可以直接用标签读,可以更改样式啥的（推荐）
console.log("svgIcon", svgIcon, svgRaw);
// 第一种使用svg的方式 （不推荐,url这个引用只是图片）
const img = document.createElement("img");
 img.src = svgIcon;
 document.body.appendChild(img);

// 第二种加载svg的方式（推荐，可以修改样式）
document.body.innerHTML = svgRaw;//因为改成raw后，直接就是二进制字符串，用innerHTML直接读就行了
const svgElement = document.getElementsByTagName("svg")[0];//选择第一张svg
svgElement.onmouseenter = function() { //onmouseenter  鼠标移到文件上时触发
     this.style.fill = "red";// 修改svg颜色不是去改他的background 也不是color 只能是fill属性
 }
```

#vite在生产环境对静态资源的处理。

当我们将工程打包完以后，会发现找不到原来的资源 （因为他现在是基于整个工程来看的，路径问题，看看是否是根盘符），解决就是(单独打开，他就是基于自己的正确路径)

生产环境不会遇到这种事情，都是交给后端或者运维去处理，他们会将dist目录挂到服务器上，进了这个服务器就直接会进入dist目录中（根盘符）

baseUrl： “/” webpack里要配置，是将资源路径变成绝对路径，这样子才能访问。



打包后的静态资源为什么要有hash？： 浏览器是有一个缓存机制，静态资源名字不改，他就会用缓存的，如果刷新，发现资源名字没变，他就会尝试去缓存里面找，文件就不会更新。有hash静态资源的名字就会一直变，就不会找缓存，一直更新，所以要避免名字一致。

hash算法:将一串字符串经过运算得到一份新的乱码字符串，文件里的内容不变，hash生成的乱码字符串不会变，文件里的内容变了一个字，hash就会重新生成乱码字符串

利用好hash算法，可以更好的控制浏览器的缓存机制。

https://www.rollupjs.com/guide/command-line-reference （rollup）

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

export default defineConfig({
    optimizeDeps: {
        exclude: [], // 将指定数组中的依赖不进行依赖预构建
    },
    build: {//配置生产包时的一些配置策略
        rollupOptions: { // 配置rollup的一些构建策略
            output: { // 控制输出
                assetFileNames: "[hash].[name].[ext]"//配置静态资源规则 [ext]:文件后缀名（占位符）;[name]:就是文件名不带后缀，例如1.css 就是1，没有css
                                                    //[hash]: 在rollup里面，hash代表将你的文件名和文件内容进行组合计算得来的结果
            }
        },
        assetsInlineLimit: 4096000, // 默认值是4096（4kb）如果静态图片资源小于4kb就会转换为base64字符；大于了就会转换为静态资源文件;
        outDir: "distDir",//打包后的名称 默认: dist;
        assetsDir: "static" //打包后配置静态资源目录的名称， 默认: assets;
    },
});
```

#vite插件

vite会在生命周期的不同阶段中去调用不同的插件以达到 不同的目的：（每个插件实现的功能不一样）

生命周期: 其实就和我们人一样, vite从开始执行到执行结束, 那么着整个过程就是vite的生命周期

如果问redux中间件是干嘛的: redux会在整个生命周期的不同阶段去调用不同的中间件以达到不同的目的；插件和中间件都是同理，在生命周期的不同阶段中去调用不同的插件 | 中间件 以达到不同的目的。

https://github.com/vitejs/awesome-vite#plugins vite插件地址

#vite-aliases

由简入繁

vite-aliases 可以帮助我们自动生成别名；检测你当前目录下包括src在内所有的文件夹，并帮助我们去生成别名；

https://github.com/subwaytime/vite-aliases vite-aliases:插件地址

```
// vite.config.js
import { ViteAliases } from 'vite-aliases'

export default {
        plugins: [//插件
                ViteAliases({
      //这里面就可以写 vite-Aliaese 的一些配置
  })
        ]
};

```

#手写vite-aliases插件

插件生命周期：config

手写Vite-aliases其实就是抢在vite执行配置文件之前去改写配置文件

vite的插件必须返回给vite一个配置对象

```
// vite.config.js
import { ViteAliases } from 'vite-aliases'

export default {
        plugins: [//使用插件的地方
```

```
    {};
    {}; //这种一个一个对象  vite会在特定的时间去遍历plugins的数组，去查看遍历的每一项需不需要执行
    {};
ViteAliases({
    //这里面就可以写 vite-Aliaese 的一些配置
})
        ]
};

```

```
// vite的插件必须返回给vite一个配置对象
// 插件导出的必须是个函数，可以让用这个插件的人有更多的定制化选择
// 构建工具都是在node环境下去运行的
const fs = require("fs");
const path = require("path");

function diffDirAndFile(dirFilesArr = [], basePath = "") {//辅助函数    区分文件还是目录
    const result = {
        dirs: [],//目录
        files: [], //文件
    }
    dirFilesArr.forEach(name => {//name 是循环的每一项文件名
        // 我直接用异步的方式去写的                                    👇下面传过来的'../src'
        const currentFileStat = fs.statSync(path.resolve(__dirname, basePath + "/" + name));
        console.log("current file stat", name, currentFileStat.isDirectory());//isDirectory() 是目录返回true，是文件返回false
        const isDirectory = currentFileStat.isDirectory();//isDirectory() 是目录返回true，是文件返回false
        if (isDirectory) {//如果是目录 就将循环的每一项文件名（name）push到上面定义的dirs（目录）
            result.dirs.push(name);
        } else {//否则 就将循环的每一项文件名（name）push到上面定义的files （文件）
            result.files.push(name);
        }
    })

    return result;
}
```

```
26
27 function getTotalSrcDir(keyName) {//用户就可以自定义路径别名的开头
28     const result = fs.readdirSync(path.resolve(__dirname, "../src"));//获取src下所有的目录
29     const diffResult = diffDirAndFile(result, "../src");//result是src下每一个文件名 //'../src' 是src目录
30     console.log("diffResult", diffResult);
31     const resolveAliasesObj = {}; // 放的就是一个一个的别名配置 @assets: xxx
32     diffResult.dirs.forEach(dirName => {//目录文件的每一项名称
33         const key = `${keyName}${dirName}`;//keyName 用户就可以自定义路径别名的开头
34         const absPath = path.resolve(__dirname, "../src" + "/" + dirName);
35         resolveAliasesObj[key] = absPath;
36     })
37     return resolveAliasesObj;
38 }
39
40 module.exports = ({//建议用commonjs规范，因为再写node端
41     keyName = "@"//用户就可以自定义路径别名的开头
42 } = {}) => {
43     return {
44         config(config, env) {//这个必须有 是一个函数  相当于将写在config里的文件拷贝到vite.config.js里覆盖掉
45             // 只是传给你 有没有执行配置文件: 没有
46             // config: 目前的一个配置对象
47             // production  development
48             // env: mode: string, command（serve build命令）: string
49             // config函数可以返回一个对象，这个对象是部分的viteconfig配置【部分：其实就是你想改的那一部分】
50             console.log("config", config, env);//这个打印是用vite跑的，只能在node端口打印出来，浏览器都是处理过后的数据
51             const resolveAliasesObj = getTotalSrcDir( );
52             console.log("resolve", resolveAliasesObj);
53             return {
54                 // 在这我们要返回一个resolve出去，将src目录下的所有文件夹进行别名控制
55                 // 返回的对象最终会合并到vite.config.js里面去
56                 // 读目录
57                 resolve: {
58                     alias: resolveAliasesObj
59                 }
60             };
61         }
```

```
    }
}

```

通过vite.config.js 返回出去的配置对象以及我们在插件的config生命周期中返回的对象都不是最终的一个配置对象，vite会把这个几个对象进行一个merge合并，例如：Object.assign（）或者 ... 展开运算符

#vite常用插件之vite-plugin-html

插件生命周期： transformIndexHtml

https://github.com/vbenjs/vite-plugin-html reslovePlugins

vite内置了很多核心插件

不像webpack一样还要配置；

vue更偏向于用户的体验，将很多插件都内置了，vite也和vue一样，减少用户的代码量；

但是react比较难，因为他更多的是让用户自己去选择用什么插件和配置；

vite-plugins-html 帮我们动态的去控制整个html文件的内容； 使用的是ejs规范https://ejs.bootcss.com/ 语法： <%= ‘内容’ %>

ejs会在服务端（node）使用的比较频繁，因为服务端经常需要动态的去修改html文件的内容

手写vite-plugin-html：

```
module.exports = (options) => {
  return {
    transformIndexHtml: { // 转换html的；直接显示在浏览器上
      enforce: "pre", // 将我们插件的一个执行时机提前；设置执行时间
      transform: (html, ctx) => {//配合着transform
        // ctx 表示当前整个请求的一个执行期上下文: api /index.html  /user/userlist json get post headers
        console.log("html", html);
        return html.replace(/<%= title %>/g, options.inject.data.title); //  /g 是全局匹配
      }
    }
    }
}
```

#vite常用插件之vite-plugin-mock

http://mockjs.com/examples.html//mock.js文档

https://www.npmjs.com/package/vite-plugin-mock// vite-plugins-mock;npm文档

mock数据：就是模拟数据；

前后端一般都是并行开发；

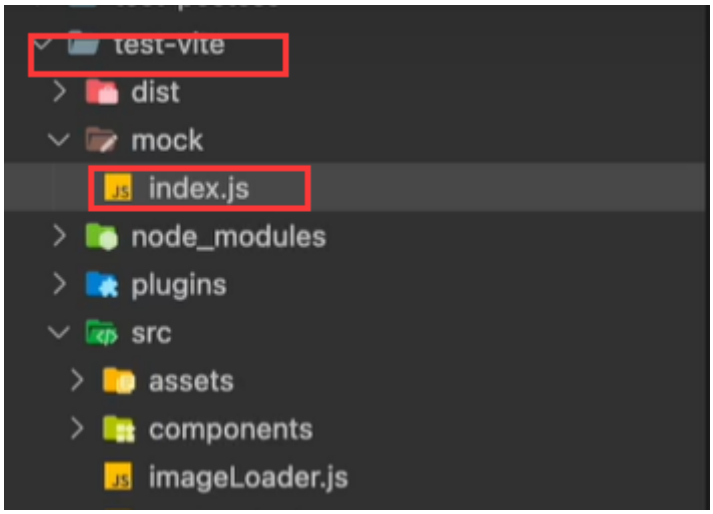
mock数据 去做你前端的工作

1. 简单方式: 直接去写死一两个数据 方便调试

-缺陷： - 没法做海量数据测试 - 没法获得一些标准数据 - 没法去感知http的异常

2. mockjs: 模拟海量数据的，

vite-plugin-mock的依赖项就是mockjs 依赖项跑完后 在config里的plugin里将 引入的vite-plugin-mock加进去，然后在根目录下创建 mock文件（他会自动找到这个mock文件）



mock文件：

```
const mockJS = require("mockjs");//commonjs引用mock包
const userList = mockJS.mock({ //mock里面必须是一个模板
  "data|100": [{//将 生成 100 条data数据 每个都是一个对象
    name: "@name", // @name mock语法：表示生成不同的中文名
    ename: mockJS.Random.last(), //@last 生成不同的英文名
    'age|11-99':1,//年龄
    address:'@county(true)',//随机地址
    email:'@email',//随机邮箱
    isMale:'@boolean',//随机性别
    createTime:'@datetime',//创建时间
    phone:/^1[34578]\d{9}$/,//随机电话号码
    "id|+1": 1, // 每个id 自增+1
    time: "@time",//生成不同的时间
    date: "@date"//生成不同的日期
    avatar() {//用户头像
    return
Mock.Random.image('100×100',Mock.Random.color(),'#757575','png',this.nickName)
    }
  }]
```

```
})
module.exports = [//commonjs到处这个包
  {
    method: "post",
    url: "/api/users",
    response: ({ body }) => {
      // body -> 请求体
      // page pageSize body
      return {
        code: 200,
        msg: "success",
        data: userList
      };
    }
  },

]


```

用html5新增的方法去读接口 fetch

```
fetch("/api/users", {
  method: "post"
}).then(data => {
  console.log("data", data);
}).catch(error => {
  console.log("error", error);
})
```

#手写vite-plugin-mock

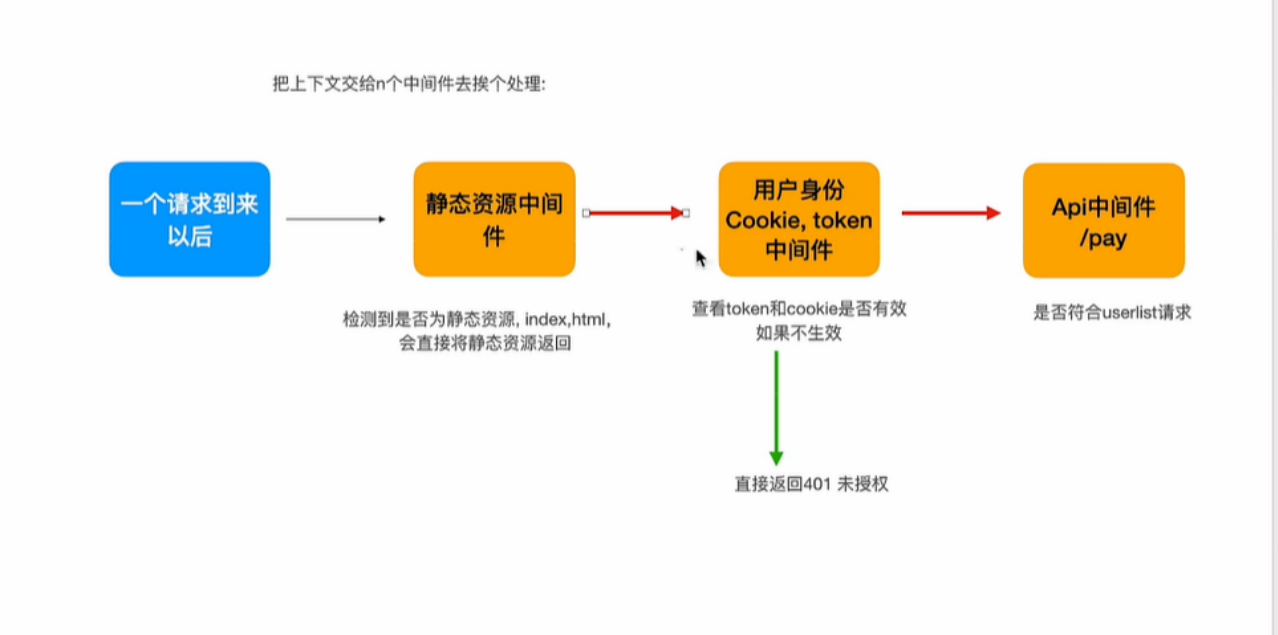
```
const fs = require("fs");
const path = require("path");

export default (options) => { //每个插件导出的都是一个函数 然后在return一个对象
  // 做的最主要的事情就是拦截http请求
  // 当我们使用fetch或者axios去请求的
```

```
  // 一般在公司里面axios的request封装会有一个baseUrl // 请求地址
  // 当打给本地的开发服务器的时候 viteserver服务器接管

  return {
    configureServer(server) {//configureServer vite官网提供的方法
      // 参数server 服务器的相关配置
      const mockStat = fs.statSync("mock");
      const isDirectory = mockStat.isDirectory();//判断上面 fs.statSync读取的mock是不是目录，是目录返回true不是就是false
      let mockResult = [];
      if (isDirectory) {
        // process.cwd() ---> 获取你当前的执行根目录,
        mockResult = require(path.resolve(process.cwd(), "mock/index.js"));//用process.cwd获取用户目录，因为mock是固定写在src下的，后面就直接mock.index.js
        console.log("result", mockResult);
      }
      server.middlewares.use((req, res, next) => {//serve参数里有个方法就是middlewares方法，就是你写的插件||中间件
      // req, 请求对象 --> 用户发过来的请求, 请求头请求体 url cookie
      // res: 响应对象, - res.header(请求头)
      // next: 是否交给下一个中间件, 调用next方法会将处理结果交
        console.log("req", req.url);
        // 看我们请求的地址在mockResult里有没有
        const matchItem = mockResult.find(mockDescriptor => mockDescriptor.url === req.url);
        console.log("matchItem", matchItem);
        if (matchItem) {
          console.log("进来了", );
          const responseData = matchItem.response(req);
          console.log("responseData", responseData);
          // 强制设置一下他的请求头的格式为json
          res.setHeader("Content-Type", "application/json");
          res.end(JSON.stringify(responseData)); // 设置请求头 异步的
        } else {//因为res.end是异步的，所以要等else才行，不然同步的话就读取不到res.end
          next(); // 你不调用next 你又不响应 也会响应东西
        }

      })  // 插件 === middlewares
    }
  }
}
```

```
44
45
```

中间件

一级一级往下走，如果满足当前中间件，就会停止往下走，走当前匹配的中间件，如果不满足，就一直往下走，直到找到满足的



#vite调用插件原理以及查漏补缺

https://vitejs.dev/guide/api-plugin.html#configureserver 文档

```
1  // import { defineConfig } from 'vite'
2  // import vue from '@vitejs/plugin-vue'
3
4  // // https://vitejs.dev/config/
5  // export default defineConfig({
6  //   plugins: [vue()]
7  // })
8  import { defineConfig } from "vite";
9   export default defineConfig({
10   plugins:[{
11     config(options){
12       // console.log(options,'options')
13       return{
14       }
15     },
16     configResolved(options){//整个配置文件完全解析完以后走的配置钩子
```

```
      //vite在内部有默认的配置文件
      //https://vitejs.dev/guide/api-plugin.html
      //到最后存和取才用的到
      console.log('options', options);

    },
    configureServer(server){//用于开发服务器的钩子，  中间件

    },
    configurePreviewServer(server){//跟configureServer一样  用于预览的服务器钩子 中间件

    },
    handleHotUpdate(ctx){// 热更新钩子 自定义热更新行为 ，覆盖官方的热更新配置；

    }
  }],
})
```

#vite与TS结合

Ts就是语法检验 检查隐性语法问题， 并且提供一些代码提示

vite对ts有着天生的支持 属于强类型锁定，

/https://vitejs.dev/guide/features.html#typescript vite--typescript 官方文档

```
//正常的ts语法：
let s:string = '123'
let s = '123' //这个不写：string也是正确的，因为后面的等于是个字符串，ts会有个类型推导；
```

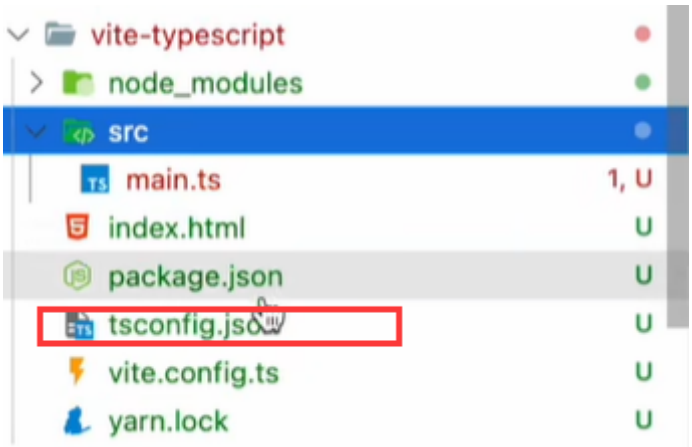
怎么让ts的错误直接输出到控制台

1.安装 vite-plugin-checker 这个插件

```
import {defineConfig} from 'vite'
import checker from 'vite-plugin-checker'//将ts的错误直接输出到控制台：弹出一个黑框

export default defineConfig {
    plugin:[
        checker:[{//将ts的错误直接输出到控制台：弹出一个黑框
            typescript: true
        }]
    ]
```

```
10 }
```

一般用ts的话，要新建一个tsconfig.json目录



如下：

```
1 //这个文件用来配置一些ts的检查手段和规则
2 {
3     "compilerOptions":{
4         "skipLibcheck": true; //是否跳过node_modules目录的检查
5     }
6 }
```

如何在ts有语法报错的时候，走yarn build 打包，它不进行打包

如果前面的tsc没有跑通，就别走后面的build了

#vite性能优化

1： 开发时态的构建速度优化：yarn dev之类的项目跑起来的命令到项目完成后的中间的过程时间：

1-1：webpack在性能优化这个方面很下功夫：webpack3用的是cache-loader webpack4用的是cache（缓存模块的打包结果（loader（如果两次构建源代码没有产生变化，就使用缓存，不调用loader））），还有这个选项：thread-loader（开启多线程去构建源代码）；

1-2：vite 启动模块是按需加载，所以不需要cacer方面；

2： 页面性能指标：和我们怎么去写代码有关：

2-1：首屏渲染的时间：fcp（first content paint）；也有可能是 (first content paint => 页面第一个元素的渲染时长)：

2-1-1：懒加载（用自己写代码实现的）

2-1-2：http优化：协商缓存（是否使用缓存需要和后端商量一下，当服务端给我们打上协商缓存的标记后，客户端会在下次需要刷新页面需要重新请求资源时会发送一个协商请求给服务端，如果服务器说需要变化，则会响应具体的内容，如果服务端觉得没有变化则会响应304；）和强缓存（服务端给响应头追加一些字段（比如：expires），客户端会记住这写字段，在expires（失效时间）没有到达之前，无论你怎么刷新页面，浏览器dou'bu'hui重新请求页面，而是从缓存里面获取；

页面中最大元素的一个加载时常：lcp（largest content paint）

3: js逻辑：

3-1：我们要注意副作用的清除：因为组件是会频繁的挂载和卸载，比如：我们在某一个组件中有一个计时器（setTimeout）,如果我们在卸载的时候不去清除这个定时器，下次再次挂载的时候，计时器就等于开了两个线程，如果线程多了会造成浏览器卡顿掉帧。

一般都是这个写（react）

```
const [timer,setTimer] = useState(null);
useEffect(()=>{
    setTime(setTimeout(()=>{}));
    return()=> clearTimeout(timer);//每次都要清除这个定时器
})
```

3-2：写法上也要有一个注意的事项：requestAnimationFrame，requestIdleCallback 就是用来卡浏览器的帧率的，要对浏览器有着一定的认识 然后在做这个方面的优化

3-2-1：浏览器的帧率：16.6ms去更新一次（执行js的逻辑以及重排和重绘。。。），假设js执行的逻辑超过了浏览器设定的16.6ms，浏览器就会掉帧从而卡

3-2-2：requestIdlecallback：传进去一个函数，浏览器是16.6ms去更新一次（如果你在这个16.6ms内更新完后还有剩余的时间，它就会执行这个requestIdlecallback这个传进去的函数，这个时候就不会造成卡帧的现象）。

3-2-3：react就出了一个concurrent mode --》concurency 分批调用，可终端渲染

4: 防抖和节流尽量不要自己写，要用lodash（js工具库）这里面的方法的性能是最好的：

```
//比如说，我们要循环一个数据 有几万条的那种
let arr = []//比如几万条
arr.forEach(()=>{//不用用这个forEach循环，用lodash库里的forEach来进行循环
```

```
4
5 })),
6
7 let arr = []
8 lodash.forEach(()=>{//用lodash库里写好的forEach来循环，他的forEach里性能是最好的，因为这里面写了很多算法，把你的性能提到最高
9
10 })
```

5: 对作用域的一个控制:

```
1 //比如
2 const arr = [1,2,3]
3 for(let i = 0; i < arr.length; i++){//不建议这么写，因为第一次i=0走完后，下次i=1时，他又要找arr要一次，arr是在window下的，然后1=2也是同理
4
5 }
6
7 for(let i = 0; arr1 = arr.length; i <arr1; i++){//这样子它每次找的时候都在自己的块级作用域里找。推荐
8
9 }
```

6: css:

1：继承属性：能继承的属性尽量不要写，例如：font-size,width,font-weight

2：避免css太过于深的css嵌套

7: 构建优化：vite（rollup），webpack

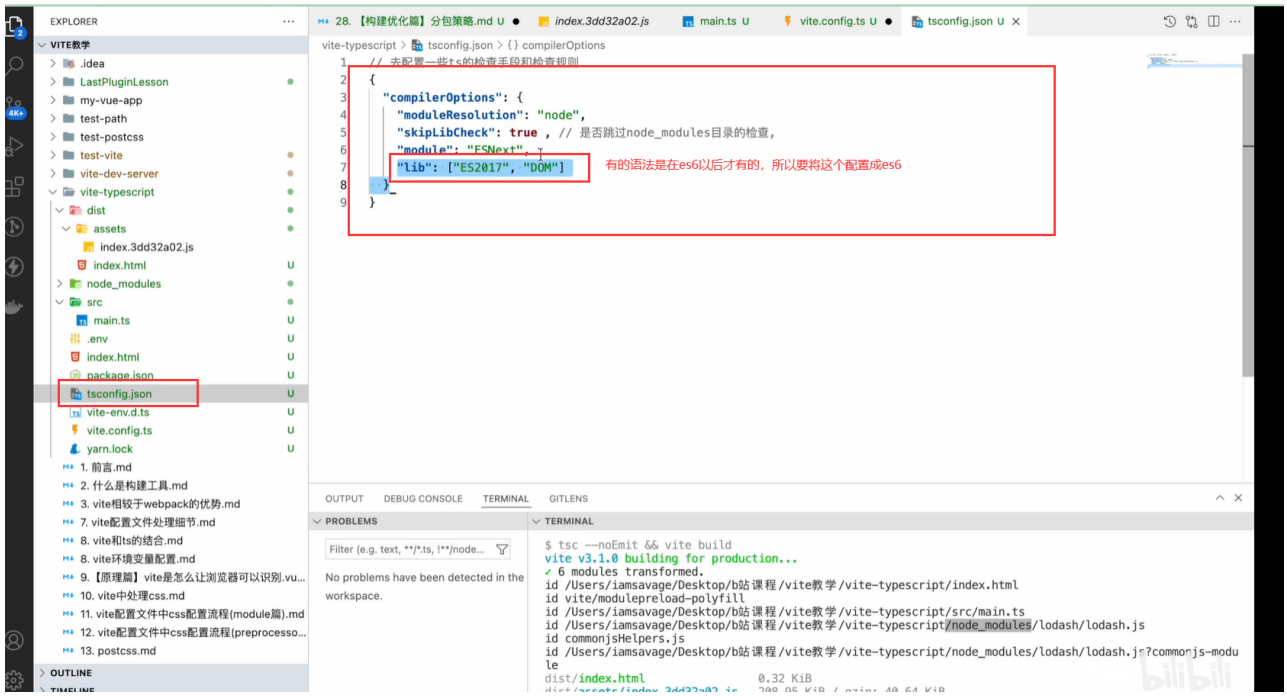
1：比如：优化体积：cdn加载，压缩，treeshaking，图片资源压缩，分包

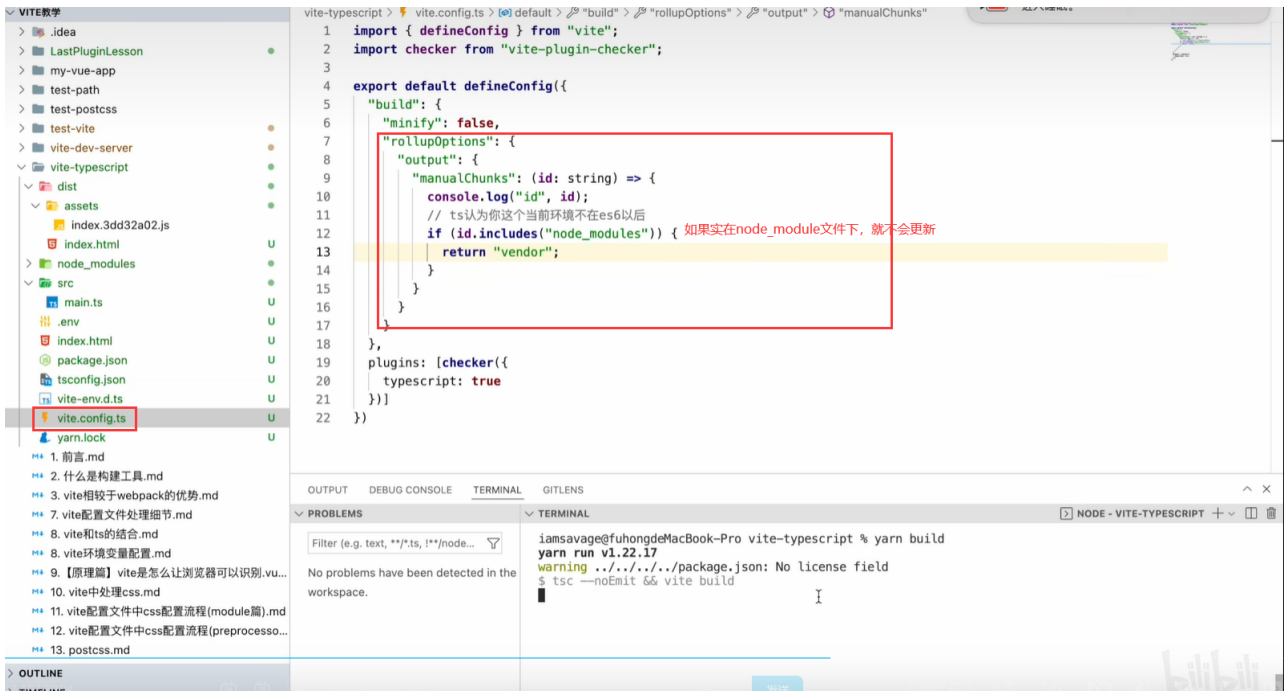
#分包策略（构建优化中的一部分）

分包就是把一些不会常规更新的文件，进行单独打包处理

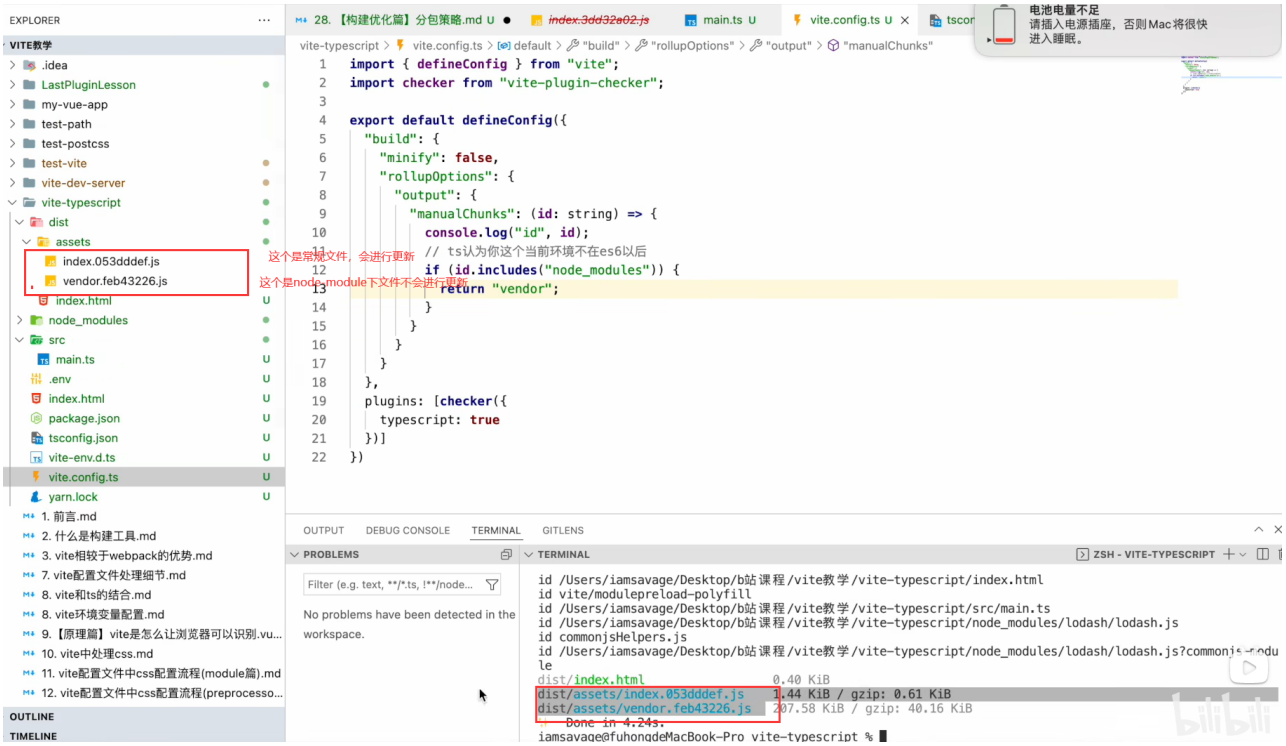
1.浏览器的缓存策略：静态资源--》文件名字没有变化，那么浏览器不会更新不会向服务器要内容，直接去缓存拿， 这就是为什么有时候非要清理缓存浏览器的数据才会更新（hash乱码）文件名不变，服务器就不会去要文件内容，就会取缓存，如果文件名变了服务器就会更新文件内容，所有要用到hash乱码，文件内容一变，文件名字也会跟着变，但是有时候下载的包，例如lodash，常规文件变的时候，然后hash乱码构成的文件名字也会变，这样子它整个文件里又要重新加载，然后lodash又不会改变，也不用重新加载，但是文件名字变了整个都会重新加载，这样子很消耗性能，所有要用分包，将一些不会常规更新的文件，进行单独打包处理（将需要更新的部分一个文件，不需要更新的是一个部分被浏览器缓存）

2.多包，就是有多个xxx.html 和xxxx.js 然后打包 vite优化了


tsconfig.json
{
"compilerOptions": {
"moduleResolution": "node",
"skipLibCheck": true , // 是否跳过node_modules目录的检查,
"module": "ESNext",
"lib": ["ES2017", "DOM"]
}
}
有的语法是在es6以后才有的，所以要将这个配置成es6
$ tsc --noEmit && vite build
vite v3.1.0 building for production...
✓ 6 modules transformed.
id /Users/iamsavage/Desktop/b站课程/vite教学/vite-typescript/index.html
id vite/modulepreload-polyfill
id /Users/iamsavage/Desktop/b站课程/vite教学/vite-typescript/src/main.ts
id /Users/iamsavage/Desktop/b站课程/vite教学/vite-typescript/node_modules/lodash/lodash.js
id commonjsHelpers.js
id /Users/iamsavage/Desktop/b站课程/vite教学/vite-typescript/node_modules/lodash/lodash.js?commonjs-module
dist/index.html 0.32 KiB


vite.config.ts
import { defineConfig } from "vite";
import checker from "vite-plugin-checker";
export default defineConfig({
"build": {
"minify": false,
"rollupOptions": {
"output": {
"manualChunks": (id: string) => {
console.log("id", id);
// ts认为你这个当前环境不在es6以后
if (id.includes("node_modules")) {
return "vendor";
}
}
}
}
},
plugins: [checker({
typescript: true
})]
})
如果实在node_module文件下，就不会更新
iamsavage@fuhongdeMacBook-Pro vite-typescript % yarn build
yarn run v1.22.17
warning ../../../../package.json: No license field
$ tsc --noEmit && vite build

打包以后：



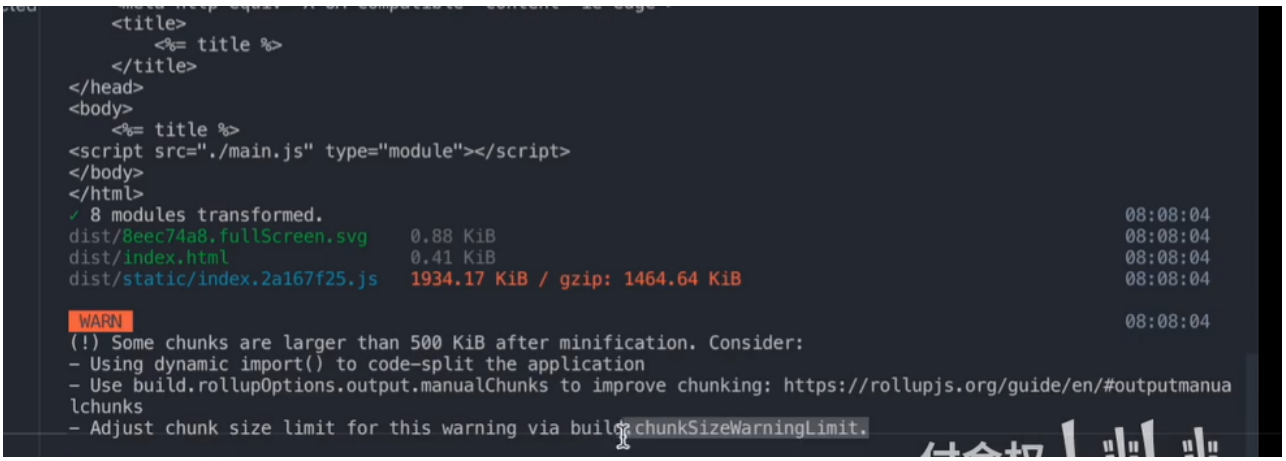
#gzip压缩（构建优化中的一部分）

有时候我们的文件资源太大了，在http中传输非常损耗传输性能，体积越大压力越大；

所以就将所有的静态文件进行压缩，达到减少体积的目的；

服务端读文件后 ==》 压缩文件，然后将压缩文件通过http传输给客户端，客户端收到压缩包 ==》浏览器再进行解压缩的动作；

从入口文件到他的一系列依赖最终打包成的js文件就叫块（chunk）；但是：块最终会映射成js文件，但不是js文件



安装gzip

yarn add vite-plugin-compression-D https://github.com/vbenjs/vite-plugin-compression/tree/main#readme

```
import viteCompression from 'vite-plugin-compression';

export default () => {
```

```
  return {
    plugins: [viteCompression()],
  };
};
```

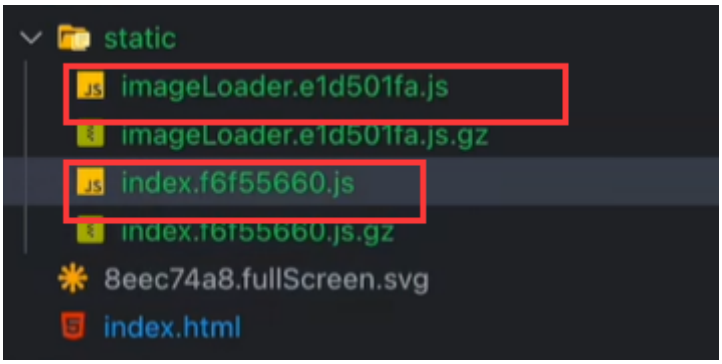
然后在打包，就会提示已经用gzip压缩了
然后将打包后的文件发给后端或者运维
就告诉后端，文件里面已经用gzip压缩过了 就别压缩了
返回服务端读取gzip文件（.gz后缀），后端会设置一个响应头（content-encoding）-->gzip(代表告诉浏览器这个文件是用gzip压缩过的文件)；然后浏览器收到响应结果，发现响应头里有gzip的字段，就会赶紧解压，得到原原本本的js文件（但是浏览器还是会承担一定的解压时间的）；！！！ 如果体积不是很大，就不要用gzip压缩（因为浏览器还要解压时间，没有必要）！！

#动态导入（构建优化中的一部分）

vite启动项目是按需加载，webpack是全部加载再启动
动态导入和按需加载是异曲同工
动态导入是es6一个新特性 （import） ，import函数始终返回一个promise
动态道路一般会用到router

```
import('../src/路径').then((data)=>{
    console.log(data)
})
```

会打包出两个文件（代码分割）



```
function import(path) {
  // resolve不被调用的话 Promise永远是pending状态
  return new Promise((resolve) => {
    // vite用的es原生的动态导入 let const c++
    // 不让他进入fullfilled状态
    // 进入到对应路由时将webpack_require.e这个promise的状态设置为fullfilled 调用resolve
    // 如果我从来没进入过home页面，我就让这个webpack_require.e永远在悬停(pending)状态
    // 创造了一个promise.all 创建一个script标签  src指向home这个编译过后的js文件 webpack一早就把jsx代码编译过了只不过
    没有给浏览器
    // 推到body里去就好了
    webpack__require.e().then(() => {
      const result = await webpack__require(path)
    })

    // 当没有进入过某个页面或者组件的时候，我们让这个组件的代码放入一个script标签里 但是这个script标签不塞入到body里去
    // 当进入这个页面时，我们将script标签塞入到整个body里去

    // 会不会被webpack编译 ？？？？ 肯定是会被编译
    // 会不会被加载？？ 不会
```

#CDN加速（构建优化中的一部分）

cdn：content delivery network （内容分发网络） 一个链接 **https://www.jsdelivr.com/**

将我们依赖的第三方模块全部写成cdn的形式，然后保证我们自己代码的一个小体积(体积小服务器和客户端的传输压力就没那么大)

依赖的第三方模块可以通过cdn来加载 就是链接 由于第三方模块是通过cdn加载的，所以自身的体积就变小了

https://www.npmjs.com/package/vite-plugin-cdn-import //cdn 使用cdn加速

第一步：

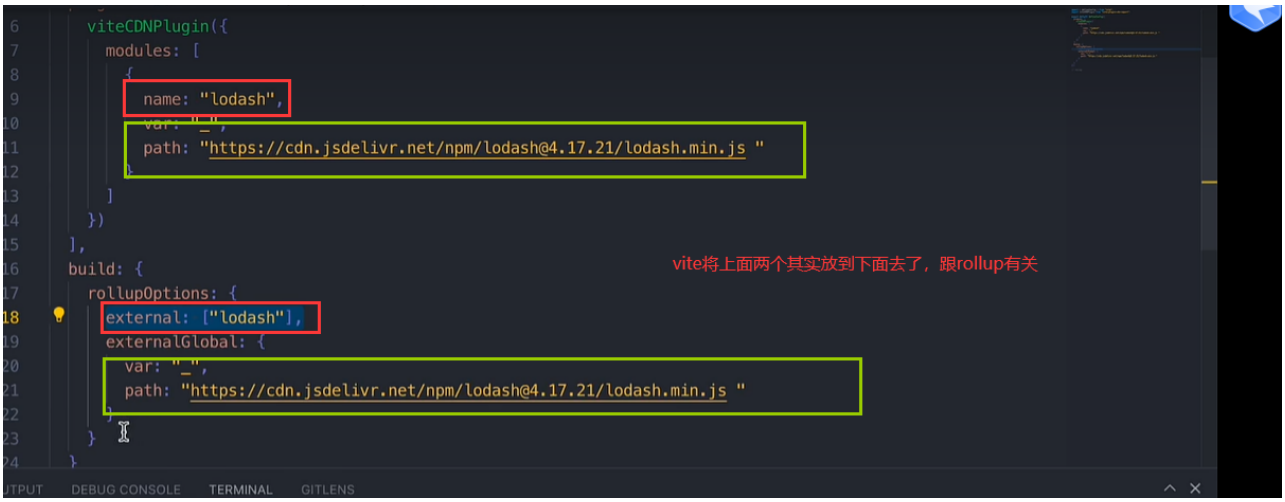
```
yarn add vite-plugin-cdn-import -D
```

第二步：

```
// vite.config.js
import _ from 'lodash'
import importToCDN from 'vite-plugin-cdn-import'

export default {
    plugins: [
        importToCDN({
            modules: [
                {
                    name: 'lodash',//引用的包的名字
                    var: '_',//变量，导出的符号，Vue是vue，jqery是$,lodash是_
                    path: `https://cdn.jsdelivr.net/npm/lodash@4.17.21/lodash.min.js`,//cdn地址
                },
```

```
14
15                ],
16            }),
17        ],
18    }
```

1.打包后，index.html给到客户端后 第一步会注入个script标签，生产环境会，开发环境不会
2.rollup一些配置



#跨域（构建优化中的一部分）

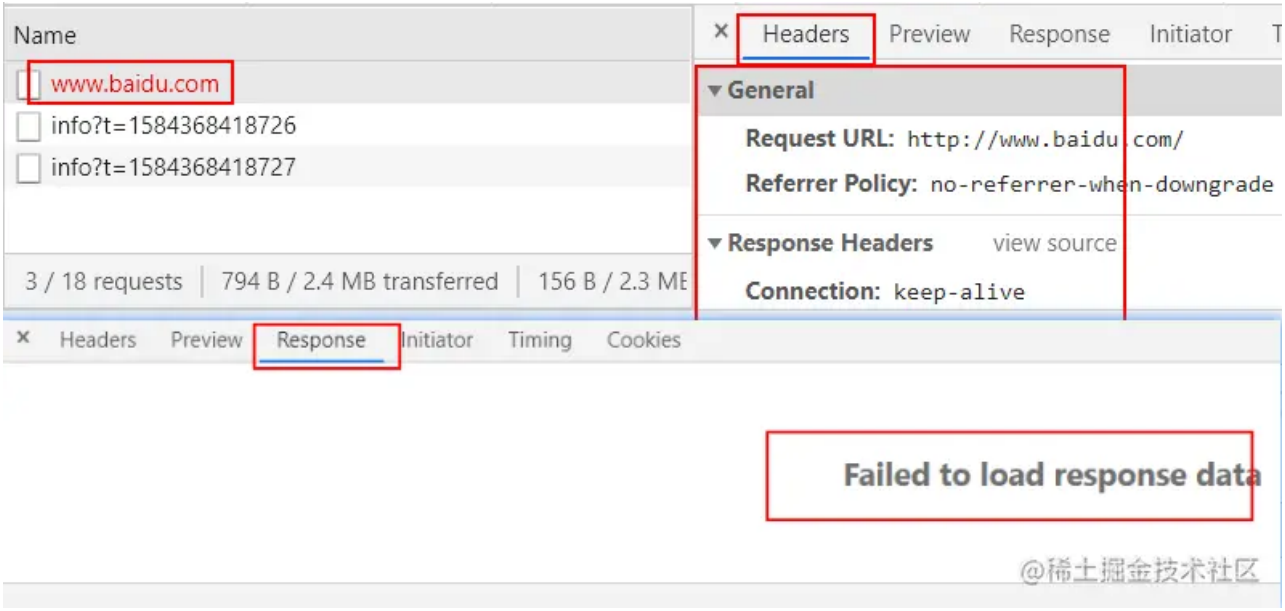
先来说说那么到底什么是跨域？跨域是指一个域下的文档或脚本去请求另一个域下的资源。跨域问题则是指浏览器出于安全考虑而需要遵循同源策略，限制不同源的网站的文件的执行，同源指的是“协议+域名+端口号”都相同。

如果非同源，如下三种行为会受到限制：

- 无法获取非同源网页的cookie、localstorage和IndexedDB
- 无法访问非同源网页的DOM（iframe）
- 无法向非同源地址发送AJAX请求（可以发送，但浏览器拒绝接受响应）

必须明确以下几点：

- 跨域问题只存在于浏览器中，而在C/S架构中，如App中是不存在跨域问题的。
- 浏览器的同源策略并不限制请求的发送，跨域时不同域的服务器是能收到请求的，但浏览器拒绝接受响应，如下图。



那么为什么浏览器才有跨域问题而App不会存在跨域问题呢？很简单，app是自家的，所有的请求都是到自家的服务器，而浏览器是可以访问很多网站的，每个网站都可以带有cookie等信息，如果被其他恶意网站利用，后果不堪设想。

跨域解决方案

跨域的解决方案很多，有如下几种：

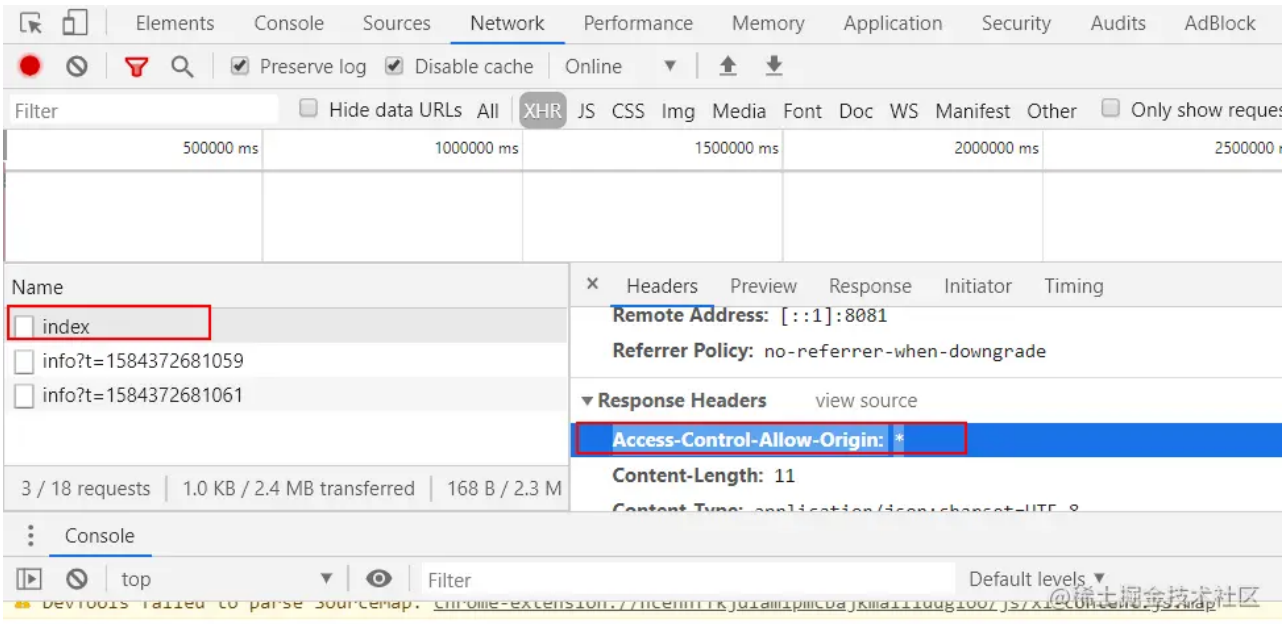
常用的方案有：

- 跨域资源共享(CORS)
- Nginx代理跨域
- JSONP

跨域资源共享CORS

CROS全称是跨域资源共享(Cross-origin resource sharing)，它的提出就是为了解决跨域请求的。跨域资源共享(CORS)标准新增了一组 HTTP 首部字段，允许服务器声明哪些源站有权限访问哪些资源。对于简单请求服务器配置后直接接受请求，对于对那些可能对服务器数据产生副作用非简单的HTTP请求方法（特别是GET以外的HTTP请求，或者搭配某些MIME类型的POST请求），浏览器必须首先使用OPTIONS方法发起一个预检请求（preflight request），从而获知服务端是否允许该跨域请求。
我们上面说过，浏览器跨域时其实请求已经发送到服务器了并返回了结果，那如果服务器配置了是允许跨域的，浏览器是否就会放行呢？答案是肯定的，当浏览器拿到响应头Access-Control-Allow-Origin: *时，会匹配其中的允许跨域的网址是否有自己，有的话则接收响应执行。如下，我在服务端Spring Boot配置了注解，成功跨域。
服务端配置：当然你可以配置成全局的，不用每个类都写注解

响应如下：



Nginx代理跨域

CORS除了上面简单使用Spring Boot注解配置外，还可以使用Nginx配置。如下，对于非简单请求，浏览器总共会发两次请求，第一次是预检请求OPTIONS。

```
location / {
    add_header Access-Control-Allow-Origin *;
    add_header Access-Control-Allow-Methods 'GET, POST, OPTIONS';
    add_header Access-Control-Allow-Headers 'DNT,X-Mx-ReqToken,Keep-Alive,User-Agent,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Authorization';

    if ($request_method = 'OPTIONS') {
        return 204;
    }
}
复制代码
```

JSONP

JSONP的原理是利用了<script>标签的src属性没有跨域的限制，每次需要跨域请求A地址时，动态生成一个<script>指向A地址并且带上回调函数名(例如名为callback)，服务端收到请求后使用该回调名拼接返回参数，如callback('helloworld')，成功返回后，浏览器将执行本地写好的callback方法。JSONP需要浏览器和服务器配合