

模块化开发

- ES modules 基本特性
 - 自动采用严格模式，忽略 'use strict'
 - 每个ESM模块都是独立的私有作用域
 - ESM是通过CORS去请求外部JS模块的
 - ESM的script标签会延迟执行脚本
- ES modules 和 commonJS之间的关系
 - ES module 中可以导入CommonJS模块
 - CommonJS中不能导入ES modules模块
 - CommonJS始终只能导入一个默认成员
 - 注意 import 不是解构导出对象

加载器分类

编译转换类

文件操作类

代码检查类

```
1  webpack
2
3  默认约定
4  'src/index.js' ==> 'dist/main.js'
5
6  配置文件
7  创建 webpack.config.js 文件
8
9  const path = require("path");
10 {
11   mode: "none",
12   // entry 表示设置webpack打包的入口文件
13   // 相对路径下，这里的./是不能省略的
14   // entry: "./src/main.js",
15   // 多入口打包
```

```
16  entry: {
17    index: "./src/index.js",
18    album: "./src/album.js",
19  },
20  // output: {
21    // 根据不同的名称生成不同的文件
22    //  filename: '[name].bundle.js'
23    // }
24
25  // output表示导出文件设置选项
26  output: {
27    // 导出文件的文件名
28    filename: "bundle.js",
29    // 导出文件的路径，这里必须是绝对路径
30    path: path.join(__dirname, "dist"),
31    // 设置网站的根目录
32    // 最后的斜线不能省略，因为最后生成的图片，是直接跟这个值进行拼接的，少了斜
    // 线就不是路径了
33    publicPath: "dist/",
34  },
35  // 用于 服务器启动时的配置信息
36  devServer: {
37    // 热更新是否开启
38    hot: true,
39    // 设置一些在平时不参与编译静态文件的路径
40    contentBase: "./public",
41    proxy: {
42      // 请求地址前缀，检测到有/api的路径就会走下面的代理
43      "/api": {
44        // target就是最后路径拼接的开头
45        // http://localhost:8080/api/users --> http://api.github.com/ap
        // i/users
46        // 相当于以/api为分界线，把前面的地址替换成target里面的值
47        target: "https://api.github.com",
48        // 这里是讲/api这字段替换成空
49        // 因为有些请求接口是没有api这个路径的，这里是以正则的一个匹配
50        pathRewrite: {
51          "^/api": "",
52        },
53        // 不能使用 localhost:8080 作为请求 GitHub 的主机名
```

```

54      // 主要用于有些网站需要更新主机名判断请求网站的场景，因为localhost这种
    路径是识别不了的
55      changeOrigin: true,
56    },
57  },
58 },
59 // 为压缩后的文件添加source源代码地图，在source处可以更好的调试代码
60 devtool: "source-map",
61 // module 是加载对应的loader
62 module: {
63   rules: [
64     {
65       test: /\.html$/,
66       use: {
67         loader: "html-loader",
68         options: {
69           // 设置html中会检查那些属性中的文件调用
70           // 根据这些属性加载的文件后缀去调用对应的loader
71           attrs: ["img:src", "a:href"],
72         },
73       },
74     },
75     {
76       // 默认因为webpack需要对文件进行打包处理，
77       // 所以会自己处理import 和export这些问题但是不会处理其他js代码转换
78       // 其他代码还是需要babel-loader来转换
79       test: /\.js$/,
80       use: {
81         loader: "babel-loader",
82         options: {
83           presets: ["@babel/preset-env"],
84         },
85       },
86     },
87     {
88       // 检索到css结尾的文件，会使用后面的use里面的，插件
89       test: /\.css$/,
90       // use中如果是一个数组，插件的执行顺序是从后往前执行
91       // css-loader 是将css文件转化为一个js模块，这时候页面上不会显示对应的
    样式

```

```

92      // style-loader 是将css-loader转化后的结果，通过style标签的形式追加
    到html页面上
93      use: ["style-loader", "css-loader"],
94  },
95  {
96      // 文件的加载的loader，主要工作过程就是把文件拷贝到工作目录，然后将文
    件路径返回
97      test: /\.png$/,
98      // use: 'file-loader'
99      // 使用url-loader的形式就会把文件打包成路径的方式返回，
    // 类似图片这种文件就是会变成base64的格式，打包后的文件就不会看到图片
    了
101     use: {
102         loader: "url-loader",
103         // 为url-loader添加配置选项
104         options: {
105             // 因为单位是字节所以要乘以1024
106             // 超出这个大小的文件，会自动调用file-loader来打包
107             limit: 10 * 1024, // 10 KB
108         },
109     },
110 },
111 ],
112 },
113 // webpack优化参数，集中放的地方
114 optimization: {
115     // 将所有公共代码部分，都集中进行打包
116     splitChunks: {
117         chunks: "all",
118     },
119     // 确定当前代码是否有副作用
120     // 在方法的原型上进行拓展的方法属于副作用
121     sideEffects: true,
122     // 只导出外部使用了的成员代码
123     // 负责标记枯树叶
124     usedExports: true,
125     // 开启压缩版，移除掉不使用的代码
126     // 负责移除枯树叶
127     minimize: true,
128     // 尽可能将所有的模块合并输出到一个函数中，提升运行效率，减少代码体积
129     concatenateModules: true,

```

```
130   },
131   // plugins 是加载对应的插件
132   plugins: [
133     // 这里需要对插件创建一个实例
134     new CleanWebpackPlugin(),
135     // 用于生成 index.html页面的
136     // 如果要生成多个，就实例出多个HtmlWebpackPlugin对象，就会生成多个文件
137     new HtmlWebpackPlugin({
138       // 里面是一些配置属性，可以直接cjs模板获取到这里面的值
139       title: "webpack Plugin Sample",
140       meta: {
141         viewport: "width=device-width",
142       },
143       // 设置模板文件名称
144       filename: "index.html",
145       // 使用模板的路径
146       template: "./src/index.html",
147       // chunks 每个打包入口会形成一个独立的chunks 能为不同的包名提供不同的
148       // 包加载，不使用全部加载的方法
149       chunks: ["index"],
150     }),
151     new CopyWebpackPlugin([
152       // 数组中指定需要拷贝的目录
153       "public",
154     ]),
155     // 这是特更新的webpack的内置插件
156     new webpack.HotModuleReplacementPlugin(),
157   ],
158 };
159
160 module.exports = (env, argv) => {
161   const config = {}
162   if (env === 'production') {
163     config.mode = 'production'
164     config.devtool = false
165     config.plugins = [
166       ...config.plugins,
167       new CleanWebpackPlugin(),
168       new CopyWebpackPlugin(['public'])
169     ]
170   }
171 }
```

```
169     ]
170   }
171 }
172
```

vue webpack正确的写法

```
1  let path = require("path");
2  let { CleanWebpackPlugin } = require("clean-webpack-plugin");
3  let VueLoaderPlugin = require("vue-loader/lib/plugin");
4  const HtmlWebpackPlugin = require("html-webpack-plugin");
5  let webpack = require("webpack");
6  const CopyPlugin = require("copy-webpack-plugin");
7
8  module.exports = {
9    mode: "none",
10    entry: "./src/main.js",
11    output: {
12      filename: "build.js",
13      path: path.resolve(__dirname, "./dist"),
14    },
15    module: {
16      rules: [
17        {
18          test: /\.js$/,
19          exclude: /node_modules/,
20          loader: "babel-loader",
21        },
22        {
23          test: /\.css$/,
24          loader: ["vue-style-loader", "css-loader"],
25        },
26        {
27          test: /\.less/,
28          exclude: /node_modules/,
29          loader: ["vue-style-loader", "css-loader", "less-loader"],
30        },
31        {
32          test: /\.png$/,
33          use: {
34            loader: "file-loader",
```

```

35     options: {
36         esModule: false,
37         name: "[path][name].[ext]",
38     },
39 },
40 },
41 {
42     test: /\.vue$/,
43     loader: "vue-loader",
44 },
45 ],
46 },
47 plugins: [
48     new CleanWebpackPlugin(),
49     new VueLoaderPlugin(),
50     new HtmlWebpackPlugin({
51         title: "vue-base",
52         filename: "index.html",
53         template: "./src/index.html",
54     }),
55     new webpack.DefinePlugin({
56         BASE_URL: JSON.stringify("./"),
57     }),
58     new CopyPlugin({
59         patterns: [
60             {
61                 from: path.resolve(__dirname, "public"),
62                 to: path.resolve(__dirname, "dist"),
63             },
64         ],
65     }),
66 ],
67 };

```

html-webpack-plugin 生成模板写法

```

1 <!DOCTYPE html>

```

```

2  <html>
3    <head>
4      <meta charset="utf-8" />
5      <title>Webpack App</title>
6      <meta name="viewport" content="width=device-width, initial-
scale=1" />
7    </head>
8
9    <body>
10      <div class="container">
11        <h1><%= htmlWebpackPlugin.options.title %></h1>
12      </div>
13      <script src="bundle.js"></script>
14    </body>
15  </html>

```

webpack 自定义插件写法

```

1 class MyPlugin {
2   // compiler 对象里面包含了此次构建的所有信息
3   // compiler 里面包含了所有钩子事件
4   apply(compiler) {
5     console.log(compiler);
6     compiler.hooks.emit.tap("MyPlugin", (compilation) => {
7       // compilation ==> 可以理解为此次打包的上下文，就是打包所有的内容都在里面
8       // 这个对象的键就是我们每个文件的名称
9       for (let name in compilation.assets) {
10         // console.log(name);
11         // 通过这个对象的source对象可以拿到，改文件的值
12         // console.log(compilation.assets[name].source())
13         if (name.endsWith(".js")) {
14           const contents = compilation.assets[name].source();
15           const withoutComments = contents.replace(/\\/\\*\\*+\\*\\/g, "");
16           compilation.assets[name] = {
17             // source 是为了覆盖原来的source上面的值
18             source: () => withoutComments,
19             // 这里的size是规定的，一定要求返回文件的大小
20             size: () => withoutComments.length,
21           };
22         }
23       }
24     });
25   }
26 }

```



```
23     }  
24   });  
25 }  
26 }
```

HMR 热更新书写格式

```
1 import createEditor from './editor'  
2  
3 let lastEditor = editor  
4 module.hot.accept('./editor', () => {  
5   // 这里检测到editor中的代码有变化的时候就会执行这里面的代码  
6   // 如果没有写更新的代码，就会走自动刷新页面的形式  
7   console.log(createEditor)  
8 })
```

图片 HMR热更新

```
1 module.hot.accept('./better.png', () => {  
2   // 这里的background就会获取到，更新文件的路径，就能实现热更新  
3   img.src = background  
4 })
```

webpack 不同环境下的配置

```
1 module.exports = (env, argv) => {  
2   // config 中是配置的内容，因为太多，所以这里省略  
3   const config = {}  
4   if (env === 'production') {  
5     // 根据env的值来判定配置文件  
6     config.mode = 'production'  
7     config.devtool = false  
8     config.plugins = [  
9       ...config.plugins,  
10      new CleanWebpackPlugin(),  
11      new CopyWebpackPlugin(['public'])  
12    ]  
13   }  
14 }  
15
```

webpack 中多个配置文件的方法

```
1 分别创建三个文件  
2 webpack.common.js // 公共部分的配置  
3 webpack.dev.js // 开发环境配置
```

```

4 webpack.prod.js // 生产环境配置
5
6 const common = require("./webpack.common.js");
7 // webpack-merge的模块是专门用与配置文件合并的
8 const merge = require("webpack-merge");
9 const { CleanWebpackPlugin } = require("clean-webpack-plugin");
10 const CopyWebpackPlugin = require("copy-webpack-plugin");
11
12 module.exports = merge(common, {
13   mode: "production",
14   // 此处的内容merge里面会自动按照它的逻辑进行合并
15   plugins: [new CleanWebpackPlugin(), new CopyWebpackPlugin()],
16 });
17
18

```

eslint

eslint步骤

需要eslint配置文件

yarn eslint --init 初始化配置文件

```

1 module.exports = {
2   env: {
3     // 这里开启支持的环境
4     // 下面的环境都是可以同时开启的
5     browser: true,
6     es2020: true
7   },
8   extends: [
9     'standard'
10  ],
11  // 这里只是检查语法，对于成员是否可用，还要看env中的环境
12  parserOptions: {
13    ecmaVersion: 11
14  },
15  // rules对具体规则进行状态提示
16  rules: {},
17  // 全局设置变量引用，设置只读就不会报undifande了
18  globals: {
19    "jQuery": "readonly"

```

```
20  }  
21  }
```

所有的环境信息

browser - 浏览器环境中的全局变量。
node - Node.js 全局变量和 Node.js 作用域。
commonjs - CommonJS 全局变量和 CommonJS 作用域 (用于 Browserify/WebPack 打包的只在浏览器中运行的代码)。
shared-node-browser - Node.js 和 Browser 通用全局变量。
es6 - 启用除了 modules 以外的所有 ECMAScript 6 特性 (该选项会自动设置 `ecmaVersion` 解析器选项为 6)。
worker - Web Workers 全局变量。
amd - 将 `require()` 和 `define()` 定义为像 `amd` 一样的全局变量。
mocha - 添加所有的 Mocha 测试全局变量。
jasmine - 添加所有的 Jasmine 版本 1.3 和 2.0 的测试全局变量。
jest - Jest 全局变量。
phantomjs - PhantomJS 全局变量。
protractor - Protractor 全局变量。
qunit - QUnit 全局变量。
jquery - jQuery 全局变量。
prototypejs - Prototype.js 全局变量。
shelljs - ShellJS 全局变量。
meteor - Meteor 全局变量。
mongo - MongoDB 全局变量。
applescript - AppleScript 全局变量。
nashorn - Java 8 Nashorn 全局变量。
serviceworker - Service Worker 全局变量。
atomtest - Atom 测试全局变量。
embertest - Ember 测试全局变量。
webextensions - WebExtensions 全局变量。
greasemonkey - GreaseMonkey 全局变量

通过注释进行代码的忽略不校验

```
// eslint-disable-line
```

eslink 在gulp中的使用 就是安装 `gulp-eslint`进行使用就行了

在使用gulp-eslint的时候也是需要创建eslint配置文件的

在gulp-eslint默认是只检查不反馈的,

反馈需要这样定义

```
return src('src/assets/scripts/*.js', { base: 'src' })  
  .pipe(plUGINS.eslint())  
  .pipe(plUGINS.eslint.format())  
  .pipe(plUGINS.eslint.failAfterError())  
  .pipe(plUGINS.babel({ presets: ['@babel/preset-env'] })))  
  .pipe(dest('temp'))  
  .pipe(bs.reload({ stream: true })))  
}
```

webpack中的eslint是在loader中进行校验的

也需要先生成eslint配置文件

对于react需要 插件的支持 `eslint-plugin-react` 这是专门针对react的eslint的插件

在eslint中进行配置

```

1  module.exports = {
2    ...
3    env: {
4      browser: true,
5      es2020: true
6    },
7    extends: [
8      'standard'
9    ],
10   parserOptions: {
11     ecmaVersion: 11
12   },
13   rules: {
14     'react/jsx-uses-react': 2,
15     'react/jsx-uses-vars': 2
16   },
17   plugins: [
18     'react'
19   ]
20 }

```

用继承的方式就不用单个去定义

```

5) 查看(V) 转到(G) 运行(R) 终端(T) 帮助(H)
.eslintrc.js x main.js index.js
1  module.exports = {
2    ...
3    env: {
4      browser: true,
5      es2020: true
6    },
7    extends: [
8      'standard',
9      'plugin:react/recommended'
10   ],
11   parserOptions: {
12     ecmaVersion: 11
13   },

```

typescript 的eslint使用

配置文件生成

```

module.exports = {
  env: {
    browser: true,
    es2020: true
  },
  extends: [
    'standard'
  ],
  parser: '@typescript-eslint/parser',
  parserOptions: {
    ecmaVersion: 11
  },
  plugins: [
    '@typescript-eslint'
  ],
  rules: {
  }
}

```

stylelint的使用 需要用插件支持 sass less postcss的检查

配置文件.stylelintrc.js

安装stylelint的继承配置项 stylelint-config-standard

然后输入继承的配置名称



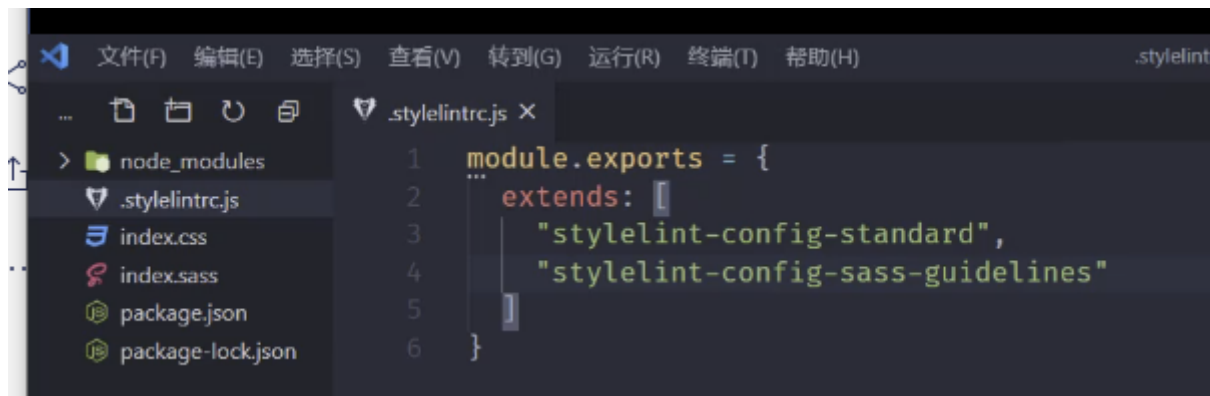
```

1 module.exports = {
2   extends: "stylelint-config-standard"
3 }

```

sass的支持需要安装 stylelint-config-sass-guidelines

引入方式一样



```

1 module.exports = {
2   extends: [
3     "stylelint-config-standard",
4     "stylelint-config-sass-guidelines"
5   ]
6 }

```

prettier 格式化工具

安装插件prettier

prettier style.css --write 就能格式化，然后写入文件

prettier不需要进行单个类型文件的配置，直接能格式化所有类型
而且eslint是提示错误，prettier是直接格式化

git Hooks

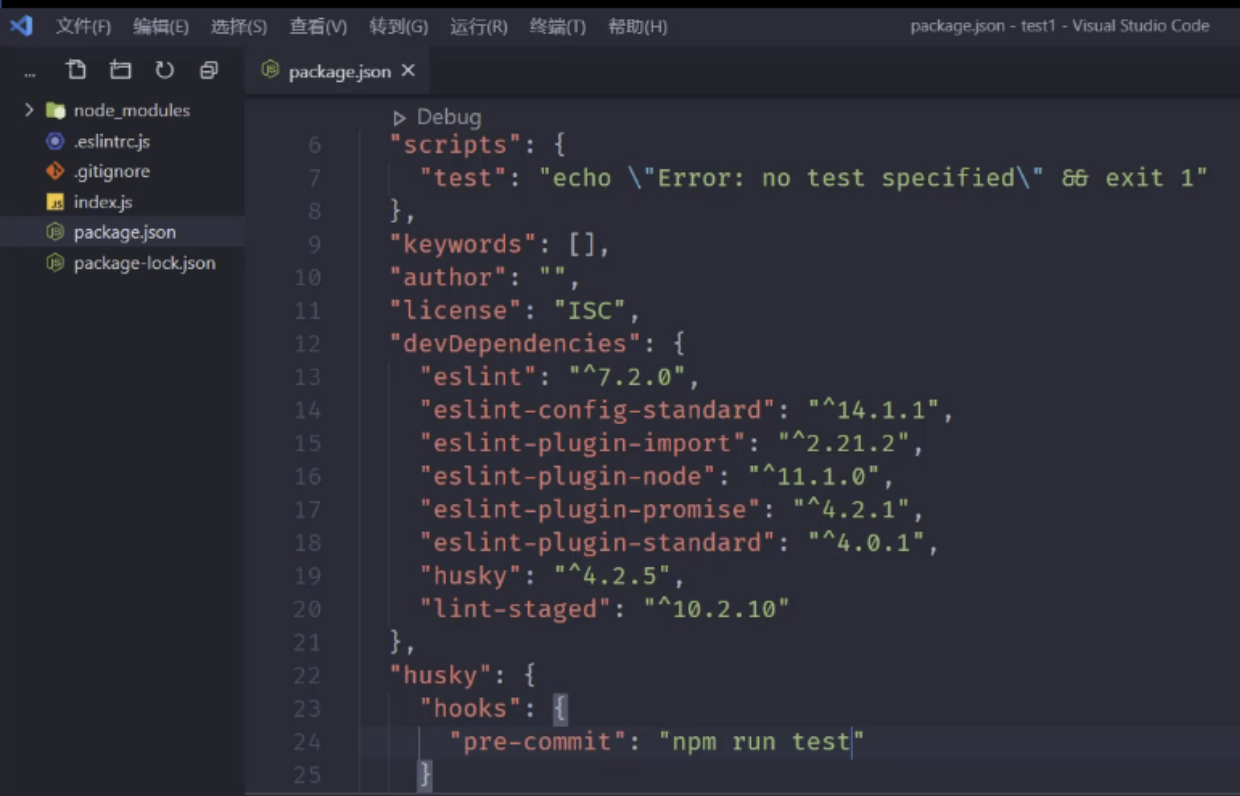
.git 隐藏文件中的hooks文件夹内的每个文件都是一个钩子

pro-commit.sample 当我们在进行commit之前进行执行，就能触发里面写的任务

Husky 在不使用shell脚本的情况下也能进行编写

安装 husky

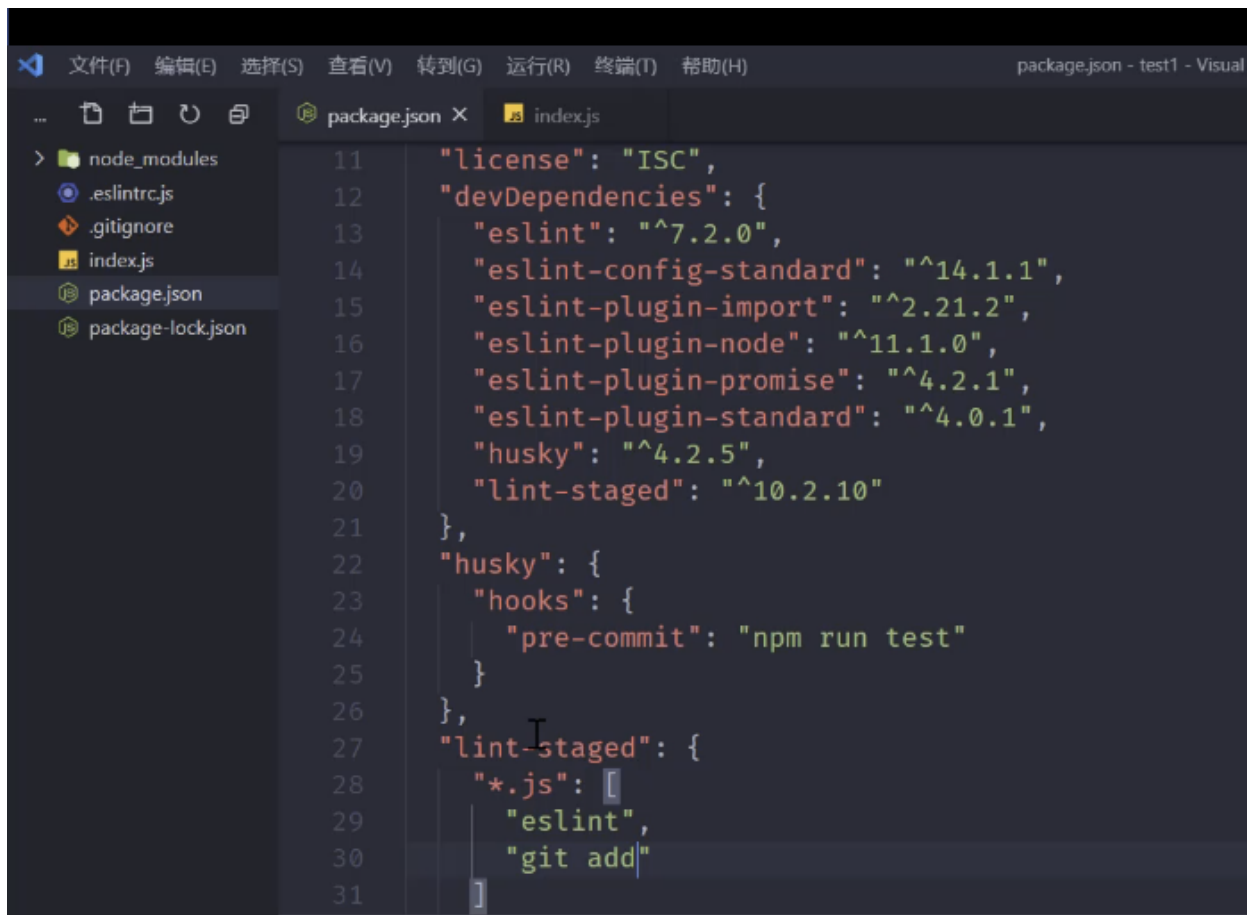
在package.json文件中添加husky

A screenshot of the Visual Studio Code editor showing the package.json file. The file is open in the editor, and the left sidebar shows the file explorer with the following files: node_modules, .eslintrc.js, .gitignore, index.js, package.json (selected), and package-lock.json. The package.json file content is as follows:

```
6  "scripts": {
7    "test": "echo \"Error: no test specified\" && exit 1"
8  },
9  "keywords": [],
10 "author": "",
11 "license": "ISC",
12 "devDependencies": {
13   "eslint": "^7.2.0",
14   "eslint-config-standard": "^14.1.1",
15   "eslint-plugin-import": "^2.21.2",
16   "eslint-plugin-node": "^11.1.0",
17   "eslint-plugin-promise": "^4.2.1",
18   "eslint-plugin-standard": "^4.0.1",
19   "husky": "^4.2.5",
20   "lint-staged": "^10.2.10"
21 },
22 "husky": {
23   "hooks": {
24     "pre-commit": "npm run test"
25   }
26 }
```

Husky可以在commit之前对代码进行一个检查，不会纠正

安装lint-staged



```
11  "license": "ISC",
12  "devDependencies": {
13    "eslint": "^7.2.0",
14    "eslint-config-standard": "^14.1.1",
15    "eslint-plugin-import": "^2.21.2",
16    "eslint-plugin-node": "^11.1.0",
17    "eslint-plugin-promise": "^4.2.1",
18    "eslint-plugin-standard": "^4.0.1",
19    "husky": "^4.2.5",
20    "lint-staged": "^10.2.10"
21  },
22  "husky": {
23    "hooks": {
24      "pre-commit": "npm run test"
25    }
26  },
27  "lint-staged": {
28    "*.js": [
29      "eslint",
30      "git add"
31    ]
  }
```

Husky 和lint-staged的搭配方式是这样的

设置commit之前的执行的命令为lint-staged,

然后再再lint-staged中定义多个任务，就能按顺序执行多个任务了

<http://eslint.cn/docs/user-guide/configuring#configuring-rules>