

webpack配置

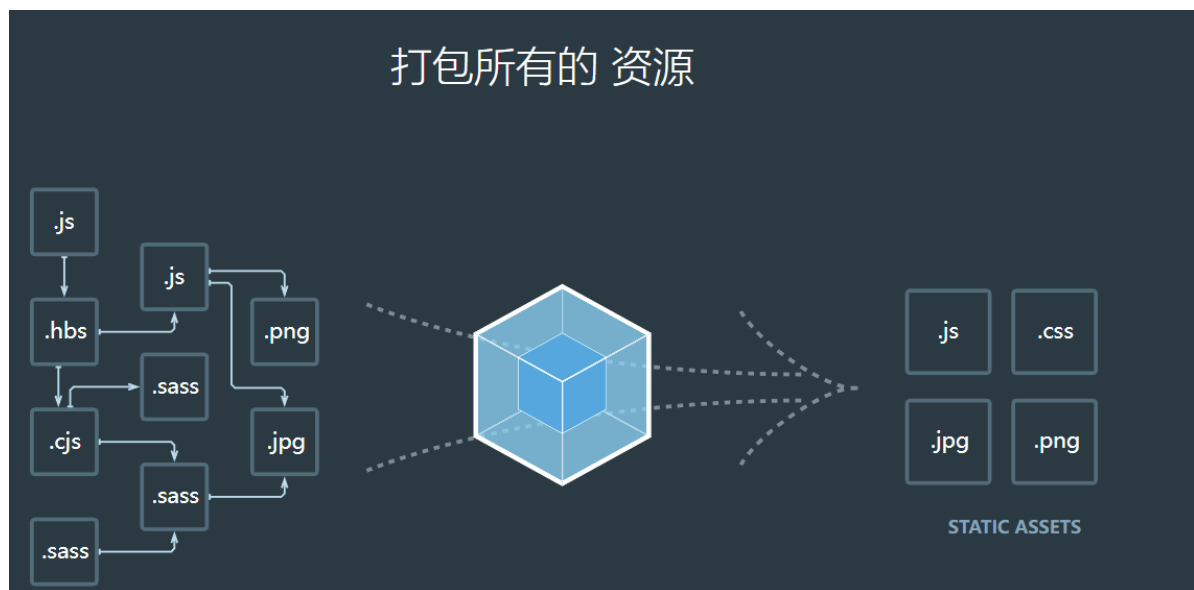
webpack 是代码编译工具，有入口、出口、loader 和插件。webpack 是一个用于现代 JavaScript 应用程序的静态模块打包工具。当 webpack 处理应用程序时，它会在内部构建一个依赖图(dependency graph)，此依赖图对应映射到项目所需的每个模块，并生成一个或多个 bundle。

```
<link rel="stylesheet" href="./bootstrap/css/bootstrap.min.css">
<link rel="stylesheet" href="./bootstrap/font/bootstrap-icons.css">
<link rel="stylesheet" href="./js/layer/theme/default/layer.css">
<script src="./js/base.js" async></script>
```

同时还发现导入了很多个JS的文件

```
<script src="./bootstrap/js/bootstrap.bundle.min.js"></script>
<script src="./js/jquery-3.6.1.js"></script>
<script src="./js/template-web.js"></script>
<script src="./js/layer/layer.js"></script>
<script src="./js/jquery.validate.js"></script>
<script src="./js/messages_zh.js"></script>
```

这就是我们之前做项目的时候所导入的依赖包，这么做有什么问题没有？



首先要知道 webpack 就是打包工具，它可以将我们的所用到的资源进行合理的整合，它相当于一个项目经理，合理的整合资源，分配资源，最后生成资源

常用的打包工具

就目前来说，打包工具不仅仅只有 webpack 还有其它的一些，我们现在认识一下常用的打包工具

1. **webpack** 这一个是2022之前非常火的一个打包工具，目前最新的版本已经到5了，它使用起来非常方便，并且功能强大，后期的 **vue**、**react**、**angular**、**sevlte** 都会使用它
2. **grunt** 打包工具，Grunt 是一个基于任务的JavaScript工程命令行构建工具。它是第一代构建工具，目前基本上已经被淘汰了，不再使用了，它所实现的功能与 **webpack** 相似
3. **gulp** 打包工具，gulp 将开发流程中让人痛苦或耗时的任务自动化，从而减少你所浪费的时间、创造更大价值。这个东西本来应该也过时了，但是现在又活过了，后期面有一个叫 **vite** 的打包工具借鉴它的思路形成了另一个工具

4. vite 打包工具,vite是下一代前端开发与构建工具。[1] Vite意在提供开箱即用的配置,同时它的插件API和 JavaScript API 带来了高度的可扩展性,并有完整的类型支持。【我们后面就主要去使用它了,就不再去使用 webpack 了】
vite 是唯一的一款国产打包工具,好用,文档齐全,并且与 vue 实现高度整合,无需任何配置就可以实现 vue 的脚手架
上面所有的打包工具都有一些共同的特征,所有的打包工具应该都有配置文件,并且配置文件里面必须有四个部分

webpack打包工具的核心点

<https://webpack.docschina.org/>

1. 入口
2. 出口
3. loader
4. 插件 plugins

目前所存在的问题

Person.js

```
1  class Person {
2    constructor(userName) {
3      this.userName = userName;
4    }
5    sayHello() {
6      setTimeout(() => {
7        console.log(`大家好, 我叫${this.userName}`);
8      }, 1000);
9    }
10 }
11 //现在用的是ESModule, 不是CommonJS模块
12 export default Person;
```

Student.js

```
1  import Person from "./Person.js";
2  class Student extends Person{
3    constructor(userName,sex){
4      super(userName)
5      this.sex = sex;
6    }
7    study(){
8      console.log(`${this.sex}学生在学习`);
9    }
10 }
11 export default Student;
```

index.js

```
1  import Student from "./Student.js";
2  let s = new Student("张珊","女");
3  s.sayHello();
4  s.study();
```

当我们把所有的 `js` 文件写好了以后，我们在 `index.html` 的网页里去调用

```
1 <script src="js/index.js" type="module"></script>
```

我们现清楚的看到了这个代码的运行

表面上看是没有任何问题

1. 因为我们现在使用的是 `ESModule` 的方式在使用模块化，所以在 `script` 的标签上面，我们使用了 `type="module"` 这个属性，但是这个属性它是在新版本的浏览器里才支持，旧版本是不支持，所以会有兼容性
2. 我们在 `js` 里面大量使用了 `class`, `extends` 以及箭头函数等ES6的语法，但是这些语法也是有兼容性的，在低版本的浏览器里面，ES6是不被支持的
3. 现在我们打开的网页是以 `http` 协议打开的，当我们双击文件直接打开的时候，这个时候就会报错了

上面的三个问题就是目前所存在的问题，如果想解决上面的问题，我们可以直接使用 `webapck` 打包个，生成一个新的文件就可以了

webapck工具的安装

webpack是nodejs平台下面的一个包，我们可以直接通过 `npm` 来进行安装

```
1 $ npm install webpack webpack-cli --save-dev
```

webpack的配置文件

webpack只是一个工具，它需要参照一定的规则来处理事情，这个规则就是配置文件，它的配置文件是固定格式，具体可以参考它的官方文档

概念 | [webpack 中文文档 \(docschina.org\)](#)

我们在当前的项目下面新建一个 `webpack.config.js` 的文件，然后配置如下

```
1 const path = require("path");
2 /**
3  * @type {import("webpack").Configuration}
4  */
5 const config = {
6   // 入口
7   entry: "./js/index.js",
8   // 出口
9   output: {
10     path: path.join(__dirname, "./dist"),
11     filename: "bundle.js"
12   },
13   //loader
14   //插件
15 }
16 module.exports = config;
```

当我们把配置文件写好了以后，我们就可以启动 `webpack` 的打包工具，然后开始打包，在打包之前我们还要知道怎么样启动 `webpack`

webpack的启动

在 `package.json` 下面的 `scripts` 目录配置启动命令

```
1  "scripts": {
2    "test": "echo \"Error: no test specified\" && exit 1",
3    "build": "webpack --config ./webpack.config.js"
4  },
```

现在我们就来启动一下刚刚配置的命令

```
1  $ npm run build
```

使用 `babel` 插件来处理JS的兼容性



它是 `webpack` 的一个插件，作用就是将 `ES6` 的代码转换成兼容性比较高的 `ES5` 代码

在这之前我们使用 `webpack` 进行打包的时候，我们都发现我们的JS代码虽然完成了打包合并，但是最大的问题它没有提高兼容性，仍然是ES6的代码，如果这些代码要是在浏览器里面运行很可能会有兼容性的问题



在设置这一页里面，我们可以找到 `webpack` 的配置

第一步：安装 `babel` 的需要的依赖包

```
1  npm install --save-dev babel-loader @babel/core
```

第二步：配置webpack中的babel插件

```
1  //省略部分代码.....
2  module: {
3    //你在import的时候会导入不同的文件，我们要根据不同的文件来制定不同的规则
4    rules:[
5      {
```

```

6      //如果我在import的时候发现你的后缀名是.js的文件，那么我就要使用babel去转换
    一下，
7      //将你转换成es5的代码
8      test: /\.js$/,
9      exclude: /node_modules/, exclude: /node_modules/,
10     // use: ["babel-loader"]
11     use: [
12       {
13         loader: "babel-loader"
14       }
15     ]
16   }
17 }
18 ]
19 },

```

第三步：指定 babel 的转换规则

4 Create babel.config.json configuration file

Great! You've configured Babel but you haven't made it actually do anything. Create a `babel.config.json` config in your project root and enable some `presets`.

To start, you can use the `env preset`, which enables transforms for ES2015+

```

Shell Copy
npm install @babel/preset-env --save-dev

```

在上面的截图里面，我们可以看到它使用了 `@babel/preset-env` 的规则，这个规则是根据你所配置的环境来决定转换规则

环境也就是开发环境，就是根据我们自己的开发环境来决定怎么转换

它提示我们需要安装一个包，那么我们现在就安装

```
1 $ yarn add @babel/preset-env --dev
```

当包安装完成以后，我们可以在刚刚创建的配置文件里面写入转换规则了

再我们再次去启动的时候，我们发现仍然没有执行转换规则，这是为什么呢？因为我们还指定JS的运行环境

在设置preset的时候，我们需要第三方的插件，分别是下面三个包

1. `@babel/preset-env` 这个包的作用是环境配置环境来决定怎么样转换JS代码
2. `@babel/polyfill` 这个包主要的作用就是实现ES6中的一些扩展功能，如 `Map` , `Set` , `Symbol`
3. `core-js` 这个包主要的作用就是用于实现Generator function生成器函数的功能,以及 `async/await/Promise`

```
1 $ yarn add @babel/polyfill core-js --dev
```

```

1  {
2    "presets": [
3      [
4        "@babel/preset-env",
5        {
6          "useBuiltIns": "usage",
7          "corejs": "3",
8          "targets": {
9            "browsers": [
10             "last 2 version",

```

```

11         "> 1%",
12         "not dead",
13         "ie 8"
14     ]
15 }
16 }
17 ]
18 ]
19 }

```

代码说明：

1. `useBuiltIns` 我们的预设里面有很多转换规则，`usage` 需要哪些就使用哪些
2. `corejs` 后面的3代表的是当前安装的版本是多少
3. `targets` 这个指js转换以后的目标平台，我们看到这里是 `browsers` 代表 `js` 在浏览器里面运行，后面有一个数组
 - `last 2 version` 代表要支持到浏览的最后2个版本
 - `> 1%` 浏览器的使用市份额大于 `1%` 就可以了
 - `not dead` 没有死的浏览器都支持

当然 我们把所有的东西都弄完了以后，最后在 `webpack.config.js` 的配置文件上面添加如下代码就可以实现最终的ES6转ES5了

```

1  const config = {
2      target: ["web", "es5"]
3  }

```

配置 `html-webpack-plugin`

这个插件的作用就是帮助我们在这个地方将生成好的JS与CSS插入到HTML文件当中去
安装这个插件

```
1  $ yarn add html-webpack-plugin --dev
```

配置html-webpack-plugin插件

```

1  // 插件
2  plugins: [
3      new HTMLWebpackPlugin({
4          // 源文件
5          template: path.join(__dirname, "index.html"),
6          // 生成的目标文件的名称
7          filename: "index.html",
8          // 把打包的js文件或后期的css文件也通过标签链接进去
9          inject: true
10     })
11 ]

```

配置 Clean-Webpack-Plugin

在刚刚的代码里面，我们可以看到，我们每次生成的东西都会在 `dist` 的目录下面，但是有些文件我们是不需要的，所以我們希望在每次生成的时候都将目标目录 `dist` 清空，这个时候就需要使用 `Clean-Webpack-Plugin`

安装

```
1 $ yarn add clean-webpack-plugin --dev
```

配置插件

```
1 const { CleanWebpackPlugin } = require("clean-webpack-plugin");
2 //配置文件
3 plugins:[
4     //省略部分代码
5     new CleanWebpackPlugin()
6 ]
```

上面的插件在 `webpack5` 里面已经被自帶了

```
1 const config= {
2     //省略部分代码
3     output:{
4         clean:true
5     }
6 }
```

webpack处理CSS文件

loader

webpack 只能理解 JavaScript 和 JSON 文件，这是 webpack 开箱可用的自带能力。loader 让 webpack 能够去处理其他类型的文件，并将它们转换为有效 模块，以供应用程序使用，以及被添加到依赖图中。

Warning

webpack 的其中一个强大的特性就是能通过 `import` 导入任何类型的模块（例如 `.css` 文件），其他打包程序或任务执行器的可能并不支持。我们认为这种语言扩展是很有必要的，因为这可以使开发人员创建出更准确的依赖关系图。

在更高层面，在 webpack 的配置中，loader 有两个属性：

1. `test` 属性，识别出哪些文件会被转换。
2. `use` 属性，定义出在进行转换时，应该使用哪个 loader。

`webpack.config.js`

首先要弄清楚一个问题，webpack 只能够处理 `js`，但是我們可以通过不同的 `loader` 来让 webpack 加载不同的文件

当我们向 `js` 文件里面导入一个 `css` 的时候文件，如下所示

```
1 import "../css/index.css";
```



```

1  //这里就是postcss的配置文件
2  module.exports = {
3    plugins: [
4      require("postcss-import"),
5      require("postcss-cssnext"),
6      require("cssnano")
7    ]
8  }

```

当配置完成以后，我们还要在 `package.json` 里面去配置它们的转换规则

```

1  "browserslist": [
2    "last 2 version",
3    "> 1%",
4    "ie 6"
5  ]

```

如果在编译的时候报 `true is not Postcss Plugin` 是因为 `postcss` 的版本过低，我们要手动提升版本

```
1  $ yarn add postcss@8 --dev
```

CSS文件分离

当我们处理好CSS的兼容性以后，我们发现我们的CSS代码是在 `JS` 里面的，我们要将CSS的代码从JS里面的分离出来

如果我们要分离CSS，又要使用一个包，这外叫叫 `mini-css-extract-plugin`

安装

```
1  $ yarn add mini-css-extract-plugin --dev
```

配置这个包

```

1  const path = require("path");
2  const HTMLWebpackPlugin = require("html-webpack-plugin");
3  const { CleanWebpackPlugin } = require("clean-webpack-plugin");
4  //压缩CSS，并分离CSS为一个单独的文件
5  const MiniCssExtractPlugin = require("mini-css-extract-plugin");
6
7  /**
8   * @type {import("webpack").Configuration}
9   */
10 const config = {
11   target: ["web", "es5"],
12   mode: "development",
13   // 入口
14   entry: "./js/index.js",
15   // 出口
16   output: {
17     path: path.join(__dirname, "./dist"),
18     filename: "js/bundle.[hash:8].js",
19     publicPath: "."
20     // 自带的清除
21     // clean:true

```

```

22     },
23     //模块，也就是加载器loader,当你去import一个文件时候 就会触发这个module里面的loader
24     module: {
25         //你在import的时候会导入不同的文件 ，我们要根据不同的文件来制定不同的规则
26         rules: [
27             {
28                 //如果我在import的时候发现你的后缀名是.js的文件，那么我就要使用babel去转换
                一下，
29                 //将你转换成es5的代码
30                 test: /\.js$/,
31                 exclude: /node_modules/,
32                 use: [
33                     {
34                         loader: "babel-loader"
35                     }
36                 ]
37             }, {
38                 test: /\.css$/,
39                 use: [
40                     MiniCssExtractPlugin.loader,
41                     {
42                         loader: "css-loader",
43                         options: {
44                             importLoaders: 1
45                         }
46                     },
47                     "postcss-loader"
48                 ]
49             }
50         ]
51     },
52     //插件
53     plugins: [
54         new HTMLWebpackPlugin({
55             // 源文件
56             template: path.join(__dirname, "index.html"),
57             // 生成的目标文件的名称
58             filename: "index.html",
59             // 把打包的js文件或后期的css文件也通过标签链接进去
60             inject: true
61         }),
62         new CleanWebpackPlugin(),
63         //配置CSS的分离插件
64         new MiniCssExtractPlugin({
65             filename: "css/index.[hash:8].css",
66             ignoreOrder: false
67         })
68     ]
69 }
70
71 module.exports = config;
72
73 const path = require("path");
74 const HTMLWebpackPlugin = require("html-webpack-plugin");
75 const { CleanWebpackPlugin } = require("clean-webpack-plugin");
76 //压缩CSS，并分离CSS为一个单独的文件

```

```

77  const MiniCssExtractPlugin = require("mini-css-extract-plugin");
78
79  /**
80   * @type {import("webpack").Configuration}
81   */
82  const config = {
83    target: ["web", "es5"],
84    mode: "development",
85    // 入口
86    entry: "./js/index.js",
87    // 出口
88    output: {
89      path: path.join(__dirname, "./dist"),
90      filename: "js/bundle.[hash:8].js",
91      publicPath: "./"
92      // 自带的清除
93      // clean:true
94    },
95    //模块，也就是加载器loader，当你去import一个文件时候 就会触发这个module里面的loader
96    module: {
97      //你在import的时候会导入不同的文件，我们要根据不同的文件来制定不同的规则
98      rules: [
99        {
100          //如果我在import的时候发现你的后缀名是.js的文件，那么我就要使用babel去转换
101          //一下，
102          //将你转换成es5的代码
103          test: /\.js$/,
104          exclude: /node_modules/,
105          use: [
106            {
107              loader: "babel-loader"
108            }
109          ],
110          {
111            test: /\.css$/,
112            use: [
113              MiniCssExtractPlugin.loader,
114              {
115                loader: "css-loader",
116                options: {
117                  importLoaders: 1
118                }
119              },
120              "postcss-loader"
121            ]
122          }
123        },
124        //插件
125      ],
126      plugins: [
127        new HTMLWebpackPlugin({
128          // 源文件
129          template: path.join(__dirname, "index.html"),
130          // 生成的目标文件的名称
131          filename: "index.html",
132          // 把打包的js文件或后期的css文件也通过标签链接进去

```

```

132         inject: true
133     })),
134     new CleanWebpackPlugin(),
135     //配置CSS的分离插件
136     new MiniCssExtractPlugin({
137         filename: "css/index.[hash:8].css",
138         ignoreOrder: false
139     })
140 ]
141 }
142
143 module.exports = config;

```

配置 sass 的 loader

sass的编译是依赖于第三方的环境，所以我们需要安装第三方的包

```
1 $ yarn add sass sass-loader --dev
```

配置sass-loader

```

1  , {
2      test: /\.s[ca]ss$/,
3      use: [
4          MiniCssExtractPlugin.loader,
5          {
6              loader: "css-loader",
7              options: {
8                  importLoaders: 2
9              }
10         },
11         "postcss-loader",
12         "sass-loader"
13     ]
14 }

```

解决 webpack 中图片的问题

第一种解决方法

```

}, {
    test: /\.s[ca]ss$/,
    use: [
        {
            loader: MiniCssExtractPlugin.loader,
            options: {
                publicPath: "../"
            }
        },
        "css-loader"
    ]
}

```

第二种解决办法

在webpack里面如果要解决图片的问题，我们就要使用第三方的 loader，因为 loader 就是用于帮我们加载资源的

安装相应的loader

```
1 $ yarn add url-loader file-loader --dev
```

配置相应的依赖包

```
}, {
  test: /\. (jpe?g|bmp|svg|gif|png)$/,
  use: [
    {
      loader: "url-loader",
      options: {
        // 小于8KB以后的图片就转base64, 否则就不转
        limit: 8 * 1024,
        name: "[name].[hash:8].[ext]",
        outputPath: "img/",
        esModule: false,
        publicPath: "../img"
      }
    }
  ],
  type: "javascript/auto"
}
```

上面的2种写法尽量使用第二种，因为它可以将图片放在一个指定的地方，这样方便，同时也可以设置图片是否以 `base64` 的形式去展示

webpack去处理字体

```
1 @font-face {
2   font-family: bgg;
3   src: url("../fonts/hksn.ttf");
4 }
5
```

在 `webpack` 里面，如果我们去加载 字体也会出现像图片一样的问题，如下所示

```
1 , {
2   test: /\. (ttf|eot|woff|woff2)$/,
3   use: [
4     {
5       loader: "url-loader",
6       options: {
7         // 小于8KB以后的图片就转base64, 否则就不转
8         limit: 8 * 1024,
9         name: "[name].[hash:8].[ext]",
10        outputPath: "fonts/",
11        esModule: false,
12        publicPath: "../fonts"
13      }
14    }
15  ],
16  type: "javascript/auto"
17 }
```

扩展yarn包管理工具

以前我们在安装第三方模块的时候我们使用的是 `npm` 来进行安装，但是这个东西并不是很完美，因为 `npm` 每次都要从服务器去下载包，这样做非常浪费流量，也非常耗时，所以我们要推荐第三方的包管理工具 `yarn`

安装

```
1 $ npm install yarn -g
```

配置镜像地址

yarn每次去下载包的时候也是从国外的服务器下载的，我们仍然要配置国内的镜像地址

```
1 $ yarn config set registry https://registry.npmmirror.com
```

配置缓存的目录

因为yarn会缓存每一次下载过的包，它的缓存目录默认是在C盘，我们要将这个缓存的目录重新设置一下

```
1 $ yarn config set cache-folder D:\yarn_cache
```

yarn也是一个包管理工具，所以它也可以安装包，也可以卸载包，也可以生成依赖文件

yarn命令与npm命令的对比

npm命令	yarn 命令
npm install 包名	yarn add 包名
npm uninstall 包名	yarn remove 包名
--save	不需要，它自动添加
--save-dev	--dev
npm init	yarn init
npm run 脚本	yarn run 脚本
npm install	yarn

当我们如果需要清理缓存目录的时候，我们要使用下面的命令

```
1 $ yarn cache clean --force
```