webpack工程化实战

项目准备

- 初始化

```
npm init -y  # 初始化npm配置文件
npm install --save-dev webpack # 安装核心库
npm install --save-dev webpack-cli # 安装命令行工具
```

- .npmrc

大家一开始使用 npm 安装依赖包时，肯定感受过那挤牙膏般的下载速度，上网一查只需要将 npm 源设置为淘宝镜像源就行，在控制台执行一下以下命令：

`npm config set registry https://registry.npm.taobao.org`

从此过上了速度七十迈，心情是自由自在的生活。

但是大家想想，万一某个同学克隆了你的项目之后，准备在他本地开发的时候，并没有设置淘宝镜像源，又要人家去手动设置一遍，我们作为项目的发起者，就先给别人省下这份时间吧，只需要在根目录添加一个 .npmrc 并做简单的配置即可：

```
# 创建 .npmrc 文件

touch .npmrc

# 在该文件内输入配置

registry=https://registry.npm.taobao.org/
```

- 创建src目录及入口文件
- 创建webpack配置文件，默认配置

```
# webpack.config.js
const path = require("path");
module.exports = {
  entry: "./src/index.js",
  output: {
    path: path.resolve(__dirname, "./dist"),
    filename: "[name].js",
  },
  mode: "development",
};
```

样式处理

- 集成css样式处理：css-loader style-loader
- 创建index.css

```
# 安装
npm install style-loader css-loader -D

# 配置
module: {
  rules: [
    {
      test: /\.css$/,
      use: ["style-loader", "css-loader"],
    },
  ],
},
```

- 集成less sass

```
# sass
npm install node-sass sass-loader -D

# less
npm install less less-loader -D

#配置
rules:[
  {
    test: /\.scss$/,
    use: ["style-loader","css-loader","sass-loader"]
  },
  {
    test: /\.less$/,
```

```
    use: ["style-loader","css-loader","less-loader""]
  }
]
```

- 集成postcss:

  Github:https://github.com/postcss/postcss

  相当于babel于JS

  postcss主要功能只有两个：第一就是把css解析成JS可以操作的抽象语法树AST，第二就是调用插件来处理AST并得到结果；所以postcss一般都是通过插件来处理css，并不会直接处理 比如：

  - 自动补齐浏览器前缀: autoprefixer
  - css压缩等 cssnano

```
npm install postcss-loader autoprefixer cssnano -D
```

```
# 创建postcss.config.js

# 配置postcss.config.js
module.exports = {
  plugins: [require("autoprefixer")],
};

# 配置package.json
"browserslist":["last 2 versions", "> 1%"],


# 或者直接在postcss.config.js里配置
module.exports = {
  plugins: [
    require("autoprefixer")({
      overrideBrowserslist: ["last 2 versions", "> 1%"],
    }),
  ],
};

# 或者创建.browserslistrc文件
> 1%
last 2 versions
not ie <= 8
```

- 样式文件分离

经过如上几个loader处理，css最终是打包在js中的，运行时会动态插入head中，但是我们一般在生产环境会把css文件分离出来（有利于用户端缓存、并行加载及减小js包的大小），这时候就用到 mini-css-extract-plugin 插件。

一般用于生产环境

```
# 安装
npm install mini-css-extract-plugin -D

# 使用
const MiniCssExtractPlugin = require('mini-css-extract-plugin');

module.exports = {
  module: {
    rules: [
      {
        test: /\.less$/,
        use: [
          // 插件需要参与模块解析，须在此设置此项，不再需要style-loader
          {
             loader: MiniCssExtractPlugin.loader,
             options: {
                hmr: true, // 模块热替换，仅需在开发环境开启
                // reloadAll: true,
                // ... 其他配置
             }
          },
           'css-loader',
           'postcss-loader',
           'less-loader'
        ],
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
      filename: '[name].css', // 输出文件的名字
      // ... 其他配置
    }),
  ]
};
```

图片/字体文件处理

- url-loader
- file-loader

`url-loader` 和 `file-loader` 都可以用来处理本地的资源文件，如图片、字体、音视频等。功能也是类似的， 不过`url-loader` 可以指定在文件大小小于指定的限制时，返回 `DataURL`，不会输出真实的文件，可以减少昂贵的网络请求。

```
# 安装
npm install url-loader file-loader -D


# 入口文件
import pic from "./logo.png";

var img = new Image();
img.src = pic;
img.classList.add("logo");

var root = document.getElementById("root");
root.append(img);


# 使用
module.exports = {
    modules: {
        rules: [
            {
                test: /\.(png|jpg|gif|jpeg|webp|svg|eot|ttf|woff|woff2)$/,
                use: [
                    {
                        loader: 'url-loader', // 仅配置url-loader即可，内部会自动调用file-loader
                        options: {
                            limit: 10240, //小于此值的文件会被转换成DataURL
                            name: '[name]_[hash:6].[ext]', // 设置输出文件的名字
                            outputPath: 'assets', // 设置资源输出的目录
                        }
                    }
                ],
                exclude: /node_modules/
            }
        ]
    }
}
```

> 注意：

> limit的设置要设置合理，太大会导致JS文件加载变慢，需要兼顾加载速度和网络请求次数。
>
> 如果需要使用图片压缩功能，可以使用 image-webpack-loader 。

处理字体： https://www.iconfont.cn/?spm=a313x.7781069.1998910419.d4d0a486a

```
//css
@font-face {
  font-family: "webfont";
  font-display: swap;
  src: url("webfont.woff2") format("woff2");
}

body {
  background: blue;
  font-family: "webfont" !important;
}

//webpack.config.js
{
  test: /\.(eot|ttf|woff|woff2|svg)$/,
  use: "file-loader"
}
```

html页面处理

HtmlWebpackPlugin

htmlwebpackplugin会在打包结束后，自动生成一个html文件，并把打包生成的js模块引入到该html中。

```
npm install --save-dev html-webpack-plugin
```

配置：

```
title: 用来生成页面的 title 元素
filename: 输出的 HTML 文件名，默认是 index.html, 也可以直接配置带有子目录。
template: 模板文件路径，支持加载器，比如 html!./index.html
inject: true | 'head' | 'body' | false  ,注入所有的资源到特定的 template 或者 templateContent 中，如果设置为 true 或者 body，所有的 javascript 资源将被放置到 body 元素的底部，'head' 将放置到 head 元素中。
favicon: 添加特定的 favicon 路径到输出的 HTML 文件中。
minify: {} | false , 传递 html-minifier 选项给 minify 输出
hash: true | false, 如果为 true, 将添加一个唯一的 webpack 编译 hash 到所有包含的脚本和 CSS 文件，对于解除 cache 很有用。
cache: true | false，如果为 true, 这是默认值，仅仅在文件修改之后才会发布文件。
showErrors: true | false, 如果为 true, 这是默认值，错误信息会写入到 HTML 页面中
chunks: 允许只添加某些块 (比如，仅仅 unit test 块)
chunksSortMode: 允许控制块在添加到页面之前的排序方式，支持的值：'none' | 'default' | {function}-default:'auto'
excludeChunks: 允许跳过某些块，(比如，跳过单元测试的块)
```

案例：

```
const path = require("path");
const htmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {
 ...
  plugins: [
    new htmlWebpackPlugin({
      title: "My App",
      filename: "app.html",
      template: "./src/index.html"
    })
  ]
};


//index.html
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <meta http-equiv="X-UA-Compatible" content="ie=edge" />
    <title><%= htmlWebpackPlugin.options.title %></title>
  </head>
  <body>
    <div id="root"></div>
```

```
  </body>
</html>
```

clean-webpack-plugin

```
npm install --save-dev clean-webpack-plugin
```

```
const { CleanWebpackPlugin } = require("clean-webpack-plugin");

...

plugins: [
    new CleanWebpackPlugin()
]
```

clean-webpack-plugin:如何做到dist目录下某个文件或目录不被清空： 使用配置项:cleanOnceBeforeBuildPatterns 案例：cleanOnceBeforeBuildPatterns: ["**/*", "!dll", "!dll/**"], ！ 感叹号相当于exclude 排除，意思是清空操作排除dll目录，和dll目录下所有文件。 注意：数组列表里的“*/”是默认值，不可忽略，否则不做清空操作。

sourceMap

源代码与打包后的代码的映射关系，通过sourceMap定位到源代码。

在dev模式中，默认开启，关闭的话 可以在配置文件里

```
devtool:"none"
```

devtool的介绍：https://webpack.js.org/configuration/devtool#devtool

eval:速度最快,使用eval包裹模块代码,

source-map： 产生`.map`文件

cheap:较快，不包含列信息

Module：第三方模块，包含loader的sourcemap（比如jsx to js ， babel的sourcemap）

inline： 将`.map`作为DataURI嵌入，不单独生成`.map`文件

配置推荐：

```
devtool:"cheap-module-eval-source-map",// 开发环境配置

//线上不推荐开启
devtool:"cheap-module-source-map",   // 线上生成配置
```

7.WebpackDevServer

- 提升开发效率的利器

每次改完代码都需要重新打包一次，打开浏览器，刷新一次，很麻烦,我们可以安装使用webpackdevserver来改善这块的体验

- 安装

```
 npm install webpack-dev-server@3.11.0 -D
```

- 配置

修改下package.json

```
"scripts": {
    "server": "webpack-dev-server"
  },
```

在webpack.config.js配置：

```
devServer: {
    contentBase: "./dist",
    open: true,
    port: 8081
  },
```

- 启动

```
npm run server
```

启动服务后，会发现dist目录没有了，这是因为devServer把打包后的模块不会放在dist目录下，而是放到内存中，从而提升速度

- 本地mock,解决跨域：

联调期间，前后端分离，直接获取数据会跨域，上线后我们使用nginx转发，开发期间，webpack就可以搞定这件事

启动一个服务器，mock一个接口：

```
// npm i express -D
// 创建一个server.js 修改scripts "server":"node server.js"

//server.js
const express = require('express')

const app = express()

app.get('/api/info', (req,res)=>{
    res.json({
        name:'开课吧',
        age:5,
        msg:'欢迎来到开课吧学习前端高级课程'
    })
})

app.listen('9092')

//node server.js

http://localhost:9092/api/info
```

项目中安装axios工具

```
//npm i axios -D

//index.js
import axios from 'axios'
axios.get('http://localhost:9092/api/info').then(res=>{
    console.log(res)
})

会有跨域问题
```

修改webpack.config.js 设置服务器代理

```
proxy: {
      "/api": {
        target: "http://localhost:9092"
      }
    }
```

修改index.js

```
axios.get("/api/info").then(res => {
  console.log(res);
});
```