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
    }),
  ]
};
```

webpack文件指纹策略：hash chunkhash contenthash

hash策略 是以项目为单位的，项目内容改变，则会生成新的hash,内容不变则hash不变

chunkhash 以chunk为单位，当一个文件内容改变，则整个chunk组的模块hash都会改变

contenthash 以自身内容为单位

css main-a42879

main-aadd69

main-aadd69

js main-4d1172c8

main-2f62d571

main-540f7cba

css内容变化

只css文件变化的 敲1

css js文件都会变化的 敲2

多页面打包通用方案

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);
});
```

8.Hot Module Replacement (HMR:热模块替换)

- css模块HMR JS模块HMR
- 不支持抽离出的css 我们要使用style-loader

启动hmr

```
devServer: {
    contentBase: "./dist",
    open: true,
    hot:true,
    //即便HMR不生效，浏览器也不自动刷新，就开启hotOnly
    hotOnly:true
  },
```

配置文件头部引入webpack

```
//const path = require("path");
//const HtmlWebpackPlugin = require("html-webpack-plugin");
//const CleanWebpackPlugin = require("clean-webpack-plugin");

const webpack = require("webpack");
```

在插件配置处添加：

```
plugins: [
    new CleanWebpackPlugin(),
    new HtmlWebpackPlugin({
      template: "src/index.html"
    }),
    new webpack.HotModuleReplacementPlugin()
  ],
```

案例：处理css模块HMR

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

var btn = document.createElement("button");
btn.innerHTML = "新增";
document.body.appendChild(btn);

btn.onclick = function() {
  var div = document.createElement("div");
  div.innerHTML = "item";
  document.body.appendChild(div);
};


//index.css
div:nth-of-type(odd) {
  background: yellow;
}
```

`注意启动HMR后，css抽离会不生效，还有不支持contenthash，chunkhash`

处理js模块HMR

需要使用module.hot.accept来观察模块更新 从而更新

案例：

```
//counter.js
function counter() {
  var div = document.createElement("div");
  div.setAttribute("id", "counter");
  div.innerHTML = 1;
  div.onclick = function() {
    div.innerHTML = parseInt(div.innerHTML, 10) + 1;
  };
  document.body.appendChild(div);
}
export default counter;


//number.js
function number() {
  var div = document.createElement("div");
  div.setAttribute("id", "number");
  div.innerHTML = 13000;
  document.body.appendChild(div);
}
export default number;




//index.js

import counter from "./counter";
import number from "./number";

counter();
number();

if (module.hot) {
  module.hot.accept("./b", function() {
    document.body.removeChild(document.getElementById("number"));
    number();
  });
}
```

Babel处理ES6

官方网站：https://babeljs.io/

中文网站：https://www.babeljs.cn/

Babel是JavaScript编译器，能将ES6代码转换成ES5代码，让我们开发过程中放心使用JS新特性而不用担心兼容性问题。并且还可以通过插件机制根据需求灵活的扩展。

Babel在执行编译的过程中，会从项目根目录下的`.babelrc`JSON文件中读取配置。没有该文件会从loader的options地方读取配置。

测试代码

```
//index.js
const arr = [new Promise(() => {}), new Promise(() => {})];

arr.map(item => {
  console.log(item);
});
```

es6+ ----->babel(presets-env)-----> es5

flow语法 ---->babel(presets-flow)->es5

jsx语法 ---->babel(preset-react) ->es5

ts语法 ---->babel(preset-ts) -->es5

垫片原理 polyfill

ie8

```
src="http://www.polyfill.js"(包含所有es6+新特性的js库)

var arr = [new Promise(function () {}), new Promise(function () {})];
arr.map(function (item) {
  console.log(item);
});
```

安装

```
npm i babel-loader @babel/core @babel/preset-env -D
```

`babel-loader是webpack 与 babel的通信桥梁，不会做把es6转成es5的工作，这部分工作需要用到@babel/preset-env来做`

Webpack.config.js

```
{
  test: /\.js$/,
  exclude: /node_modules/,
  use: {
    loader: "babel-loader",
    options: {
      presets: ["@babel/preset-env"]
    }
  }
}
```

通过上面的几步 还不够，默认的Babel只支持let等一些基础的特性转换，Promise等一些还有转换过来，这时候需要借助@babel/polyfill，把es的新特性都装进来，来弥补低版本浏览器中缺失的特性

@babel/polyfill

以全局变量的方式注入进来的。windows.Promise，它会造成全局对象的污染

```
npm install --save @babel/polyfill
```

```
//index.js 顶部
import "@babel/polyfill";
```

按需加载，减少冗余

会发现打包的体积大了很多，这是因为polyfill默认会把所有特性注入进来，假如我想我用到的es6+，才会注入，没用到的不注入，从而减少打包的体积，可不可以呢

当然可以

修改Webpack.config.js

```
options: {
          presets: [
            [
              "@babel/preset-env",
              {
                targets: {
                  edge: "17",
                  firefox: "60",
                  chrome: "67",
                  safari: "11.1"
                },
                corejs: 2,//新版本需要指定核心库版本
                useBuiltIns: "usage"//按需注入
              }
            ]
          ]
        }
```

`useBuiltIns` 选项是 `babel 7` 的新功能，这个选项告诉 `babel` 如何配置 `@babel/polyfill` 。 它有三个参数可以使用： ①entry: 需要在 `webpack` 的入口文件里 `import "@babel/polyfill"` 一次。 `babel` 会根据你的使用情况导入垫片，没有使用的功能不会被导入相应的垫片。 ②usage: 不需要 `import` ，全自动检测，但是要安装 `@babel/polyfill` 。 ③false: 如果你 `import "@babel/polyfill"` ，它不会排除掉没有使用的垫片，程序体积会庞大。(不推荐)

用户想使用3.x的情况下！！！！！！

Core-js 2.x 和 3.x的区别： 包含的新特性数量不同

polyfill 2.x

扩展：

babelrc文件：

新建.babelrc文件，把options部分移入到该文件中，就可以了

```
//.babelrc

{
  presets: [
    [
      "@babel/preset-env",
      {
        targets: {
          edge: "17",
          firefox: "60",
          chrome: "67",
          safari: "11.1"
        },
        corejs: 2, //新版本需要指定核心库版本
        useBuiltIns: "usage" //按需注入
      }
    ]
  ]
}


//webpack.config.js

{
  test: /\.js$/,
  exclude: /node_modules/,
  loader: "babel-loader"
}
```

10.配置React打包环境

安装

```
npm install react react-dom --save
```

编写react代码：

```
//index.js
import React, { Component } from "react";
import ReactDom from "react-dom";

class App extends Component {
  render() {
    return <div>hello world</div>;
  }
}

ReactDom.render(<App />, document.getElementById("app"));
```

安装babel与react转换的插件：

```
npm install --save-dev @babel/preset-react
```

在babelrc文件里添加：

```
{
  "presets": [
    [
      "@babel/preset-env",
      {
        "targets": {
          "edge": "17",
          "firefox": "60",
          "chrome": "67",
          "safari": "11.1",
          "Android":"6.0"
        },
        "useBuiltIns": "usage", //按需注入
      }
    ],
    "@babel/preset-react"
  ]
}
```

如何实现一个plugin

webpack在编译代码过程中 生命周期概念 对应不同的打包阶段

不同打包阶段

- module
- Assets

plugin本质上是一个类

- 它是如何注册到webpack的对应阶段的

webpack的打包流程

1.拿到配置，初始化工作 最终配置

2.实例化一个compiler类，注册插件，对应的生命周期绑定相应的时间

3.执行编译，compiler.run

// 10各阶段

4.compiler(构建阶段)->compilation(第七个阶段，bundle资源被加工成什么样子了)

5.递归处理所有的依赖模块 生成chunk

6.把chunk输出到output指定的位置

暗号：

做人嘛，最重要的是开心

作业：

实现一个打包清单插件

要求，打包结束后，输出目录多出一个fileList.txt

内容：bundle文件的数量，以及全称

over!!

hash chunkhash contenthash

css/index-d5d585.css

css/index-d5d585.css

index-906333.js

index-91aa4d.js

list-928a15.js

A项目

banner

B项目

banner

Webpack5.x