# [JepsonGithub](https://github.com/JepsonGithub)/[webpack-public-stu](https://github.com/JepsonGithub/webpack-public-stu)

**------------- day01 --------------**

**webpack 概述**

webpack 是一个现代 javascript 应用程序的 **静态模块打包器 (module bundler)**

[webpack官网](https://webpack.js.org/)

**webpack 能做什么**

webpack是一个静态模块打包器

1. 语法转换
   * less/sass/stylus转换成css
   * ES6转换成ES5
   * ...
2. html/css/js 代码压缩合并 (打包)
3. webpack可以在开发期间提供一个开发环境
   * 自动打开浏览器
   * 保存时自动刷新
4. 项目一般先打包再上线

**webpack 的基本使用**

**webpack基本打包配置**

1. 建目录 dist src/main.js
2. 初始化
3. yarn init -y
4. 安装依赖包 (-D 将依赖记录成开发依赖, 只是开发中需要用的依赖, 实际上线不需要的)
5. yarn add webpack webpack-cli -D
6. 到package.json文件中, 配置scripts
7. scripts: {
8. "build": "webpack --config webpack.config.js"

}

1. 提供 webpack.config.js

参考文档: [https://webpack.docschina.org/concepts/#入口-entry-](https://webpack.docschina.org/concepts/#%E5%85%A5%E5%8F%A3-entry-)

const path = require('path')

module.exports = {

// entry: 配置入口文件 (从哪个文件开始打包)

entry: './src/main.js',

// output: 配置输出 (打包到哪去)

output: {

// 打包输出的目录 (必须是绝对路径)

path: path.join(\_\_dirname, 'dist'),

// 打包生成的文件名

filename: 'bundle.js'

},

// 打包模式 production 压缩/development 不压缩

mode: 'development'

}

1. 执行配置的scripts脚本
2. yarn build

小测试:

​ 假定在main.js中导入一个 aa.js, 多个文件需要打包, wepack会打包成一个文件, 可以节约请求的次数

require('./aa.js')

console.log('这是main模块')

**基于 webpack 实现隔行变色**

* 新建 public/index.html 编写代码
* 在 index.html 中新建一些 li 玩玩
* <!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>Document</title>
* </head>
* <body>
* <div id="app">
* <!-- ul>li{我是第$个li}\*10 -->
* <ul>
* <li>我是第1个li</li>
* <li>我是第2个li</li>
* <li>我是第3个li</li>
* <li>我是第4个li</li>
* <li>我是第5个li</li>
* <li>我是第6个li</li>
* <li>我是第7个li</li>
* <li>我是第8个li</li>
* <li>我是第9个li</li>
* </ul>
* </div>
* <script src="../dist/bundle.js"></script>
* </body>

</html>

需求:

**使用 jquery 隔行变色**

安装jquery

yarn add jquery

main.js

// 需求: 通过jquery实现隔行变色

const $ = require('jquery')

$(function() {

$('#app li:nth-child(odd)').css('color', 'red')

$('#app li:nth-child(even)').css('color', 'green')

})

**自动生成html - html-webpack-plugin插件**

在 index.html 中 手动引入 打包后的资源，是有缺点的

比如: 如果webpack 配置中的输出文件名修改了，需要及时在 index.html 中同步修改

1. 下载 (-D 将依赖记录成开发依赖, 只在开发阶段用, 实际上线是不需要的)
2. yarn add html-webpack-plugin -D
3. **在webpack.config.js文件中，引入这个模块** :

```js

// 引入自动生成 html 的插件

const HtmlWebpackPlugin = require('html-webpack-plugin')

```

1. 配置
2. plugins: [
3. new HtmlWebpackPlugin({ template: './public/index.html' })

]

配置好了之后, public 目录的 index.html 就不需要引入打包后的文件了, 会自动被插件生成 html 引入

**webpack - loaders 的配置**

webpack默认只认识 js 文件, 但是webpack 可以使用 [loader](https://www.webpackjs.com/concepts/loaders) 来加载预处理文件, 允许webpack也可以打包 js之外的静态资源。

所以webpack如果要处理其他文件类型, **记得要先配置对应的 loader**

**webpack中处理 css 文件**

需求: 去掉小圆点, 新建 css 目录

1. 安装依赖
2. yarn add style-loader css-loader -D
3. 配置
4. module: {
5. // loader的规则
6. rules: [
7. {
8. // 正则表达式，用于匹配所有的css文件
9. test: /\.css$/,
10. // 先用 css-loader 让webpack能够识别 css 文件的内容
11. // 再用 style-loader 将样式, 以动态创建style标签的方式添加到页面中去
12. use: [ "style-loader", "css-loader"]
13. }
14. ]

},

**分离 css 文件**

将css放到了style标签中, 请求次数是少了,

但是如果css文件太大的话，也不是太好，那有没有什么办法把css分离出来呢？

* 有一个插件，mini-css-extract-plugin，这个插件支持webpack4.x
* 之前的插件extract-text-webpack-plugin对webpack3.x的版本支持 (目前已废弃)

1. 安装依赖包
2. yarn add mini-css-extract-plugin -D
3. **在webpack.config.js文件中，引入这个模块**

// 引入分离 css 文件的 模块

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

1. 配置loaders
2. // 模块加载
3. module: {
4. // loader的规则
5. rules: [
6. // 配置 css 文件的解析
7. {
8. test: /\.css$/,
9. use: [ // 根据官方文档写的，注意'css-loader'的书写位置
10. {
11. loader: MiniCssExtractPlugin.loader,
12. options: {
13. publicPath:'../',
14. },
15. },
16. 'css-loader'
17. ]
18. },
19. ],

}

1. 插件的配置
2. // 配置插件
3. plugins: [
4. ...
5. // 定义打包好的文件的存放路径和文件名
6. new MiniCssExtractPlugin({
7. filename:'css/index.css'
8. })

],

**webpack 中处理 less 文件**

1. 下载依赖包

注意: 解析less文件需要识别 less 语法, 所以除了 less-loader 需要额外下载 less 包

less-loader: 将less转换成 css

yarn add less less-loader -D

1. 配置
2. // 配置 less 文件的解析
3. {
4. test: /\.less$/,
5. use: [
6. // 分离出 css 内容
7. {
8. loader: MiniCssExtractPlugin.loader,
9. options: {
10. publicPath:'../',
11. },
12. },
13. 'css-loader',
14. 'less-loader'
15. ]

}

**webpack 中处理图片 - url-loader**

我们试了一下，在css中用到一下背景图片，结果就报错了，难道webpack连图片也认不出来吗？

没有错，的确认不出来, 此时需要转换图片的 loader, 来处理图片的问题, 主要用到 url-loader 和 file-loader

注意: url-loader 中的部分功能要用到 file-loader, 要下载两个模块

1. 下载依赖包
2. yarn add url-loader file-loader -D
3. 配置loader
4. {
5. test: /\.(png|jpg|gif)$/i,
6. use: [
7. { loader: 'url-loader' }
8. ]

}

图片默认转成 base64 字符串了,

* + 好处就是浏览器不用发请求了，直接可以读取
  + 坏处就是如果图片太大，再转base64就会让图片的体积增大 30% 左右, 得不偿失

所以需要通过 options 配置选项进行配置 limit, 可以设置一个临界值, 大于这个值会整个文件直接打包到目录中, 得到是路径,

如果小于这个值, 就会转成 base64, 节约请求的次数

{

test: /\.(png|jpg|gif)$/i,

use: [

{

loader: 'url-loader',

options: {

// 超过 8k 就不转 base64, 小于 8k 才转

limit: 8 \* 1024

}

}

]

}

**清除dist目录的插件**

使用 [clean-webpack-plugin插件](https://www.webpackjs.com/guides/output-management/#%E6%B8%85%E7%90%86-dist-%E6%96%87%E4%BB%B6%E5%A4%B9) 在每次打包前清除下dist文件夹。

安装依赖包

yarn add clean-webpack-plugin -D

webpack.config.js

// 其他代码

// 导入清除插件

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

module.exports = {

// 其他配置

plugins: [

// ....

// 调用清除打包目录插件

new CleanWebpackPlugin()

]

};

**配置图片的打包输出目录**

默认是直接输出到了 dist 根目录, 可以通过 options 进行配置

{

test: /\.(png|jpg|gif)$/,

use: [

{

loader: 'url-loader',

options: {

// 超过 8k 就不转 base64, 小于 8k 才转字符串

limit: 8 \* 1024,

// 配置输出的文件名

name: '[name].[ext]',

// 配置静态资源的引用路径

publicPath: "../images/",

// 配置输出的文件目录

outputPath: "images/"

}

}

]

}

**-------------- day02 --------------**

**webpack开发服务器**

**webpack 使用 babel 处理高版本的 js 语法**

babel 的介绍 => 用于处理高版本 js语法 的兼容性

1. 安装包
2. yarn add -D babel-loader @babel/core @babel/preset-env
3. 配置规则
4. module: {
5. rules: [
6. {
7. test: /\.js$/,
8. exclude: /(node\_modules|bower\_components)/,
9. use: {
10. loader: 'babel-loader',
11. options: {
12. presets: ['@babel/preset-env']
13. }
14. }
15. }
16. ]

}

**webpack-dev-server自动刷新**

1. 下载

yarn add webpack-dev-server -D

1. 配置scripts

scripts: {

"build": "webpack --config webpack.config.js"

"dev": "webpack-dev-server --config webpack.config.js"

}

1. webpack-dev-server 的配置

module.exports = {

...

devServer: {

port: 3000, // 端口号

open: true // 自动打开浏览器

}

}

**生产环境 和 开发环境**

生产环境和开发环境刚好相反，开发环境在本地运行，而生产环境是要产出运行在线上服务器面向用户使用的代码，因此两者的构建目标差异很大，比如打包后的文件在生产环境中要尽可能的小，逻辑代码分离，优化静态资源（压缩图片）等。

**因此开发环境和生产环境不能共用一份webpack配置文件，需要分别指定**

但是两个环境还是有很多配置可以共用的，比如entry、output、module等，因此可以把公共部分的配置抽离出来放到一个独立的文件然后进行合并，我们可以使用webpack-merge工具来进行合并。

\*\*注意：\*\*entry、output、module这些配置在我们当前示例通用，但未必适合所有项目。

安装依赖

yarn add webpack-merge -D

开始拆分webpack.config.js文件，拆分后这个文件就不要了。

新建config文件夹:

- webpack-demo

- config // 存放配置文件的文件夹

- webpack.base.js // 公共的配置

- webpack.dev.js // 开发环境的配置

- webpack.pro.js // 生成环境的配置

- // 其他文件

**配置文件**

config/webpack.base.js

// 存放公共的部分

const path = require('path')

// 引入自动生成 html 的插件

const HtmlWebpackPlugin = require('html-webpack-plugin')

// 引入分离 css 文件的 模块

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

// 导入清除插件, 可以在每次打包之前, 清除 dist目录的内容

const { CleanWebpackPlugin } = require('clean-webpack-plugin')

// 配置webpack的配置文件, 需要将配置的对象导出, 给webpack使用

module.exports = {

// 入口 entry, 从哪个文件开始打包

entry: './src/main.js',

// 出口 output, 打包到哪里去

output: {

// 打包输出的目录 (输出的目录必须是一个绝对路径)

path: path.join(\_\_dirname, '../dist'),

// 打包后生成的文件名

filename: 'js/bundle.js'

},

// 配置module模块加载规则

// 默认, webpack只认识json, javascript, 不认识其他文件, 如果希望打包处理其他文件, 需要配置对应loader

module: {

rules: [

// (1) 配置css文件的解析

{

// 正则: 匹配所有以css结尾的文件

test: /\.css$/,

// 实际处理顺序: 从右往左

// css-loader 让webpack能够识别解析 css 文件

// style-loader 通过动态的创建style标签的方式(js), 让解析后的css内容, 能够作用到页面中

use: [

{

loader: MiniCssExtractPlugin.loader,

options: {

publicPath: '../',

},

},

'css-loader'

]

},

// (2) 配置less文件的解析

{

test: /\.less$/,

use: [

{

loader: MiniCssExtractPlugin.loader,

options: {

publicPath: '../',

},

},

'css-loader',

'less-loader'

]

},

// (3) 配置图片文件的解析 i 表示忽视大小写 .PNG

{

test: /\.(png|jpg|gif)$/i,

use: [

// url-loader 如果不配置, 默认都会将文件转成base64字符串的格式

{

loader: 'url-loader',

// 8k以内, 转成base64, 节约请求次数, 8k以外, 单独一个文件请求

options: {

limit: 8 \* 1024,

// 配置输出的文件名

name: '[name].[ext]',

// 配置静态资源的引用路径

publicPath: "../images/",

// 配置输出的文件目录

outputPath: "images/"

}

}

]

},

// (4) 配置新版本js文件的解析

{

test: /\.js$/,

exclude: /(node\_modules|bower\_components)/,

use: {

loader: 'babel-loader',

options: {

presets: ['@babel/preset-env']

}

}

}

]

},

// 配置插件

plugins: [

// 自动生成 html 的插件

new HtmlWebpackPlugin({ template: './public/index.html' }),

// 分离css的插件, 定义打包好的文件的e存放路径和文件名

new MiniCssExtractPlugin({

filename:'css/index.css'

}),

// 清除dist目录的插件

new CleanWebpackPlugin()

]

}

webpack.dev.js

// 存放开发模式下的配置 development

const base = require('./webpack.base.js')

// 用于合并webpack配置的插件

const merge = require('webpack-merge')

// merge 可以接受多个参数, 把参数对象合并成一个对象

// 后面的对象属性, 会覆盖前面的对象属性

module.exports = merge(base, {

// 配置开发服务器

devServer: {

port: 3000, // 端口号

open: true // 自动打开浏览器

},

mode: 'development'

})

webpack.pro.js

// 存放生产模式的配置 production

const base = require('./webpack.base.js')

// 用于合并webpack配置的插件

const merge = require('webpack-merge')

// merge 可以接受多个参数, 把参数对象合并成一个对象

// 后面的对象属性, 会覆盖前面的对象属性

module.exports = merge(base, {

mode: 'production' // 声明当前是生产环境

})

\*\*注意：\*\*拆分完webpack.config.js后可以把该文件删除了。

修改scripts启动命令，**注意指定配置文件的路径变化**

package.json

{

"scripts": {

"build": "webpack --config config/webpack.pro.js",

"dev": "webpack-dev-server --config config/webpack.dev.js"

},

}

**多入口多出口**

多入口需要修改entry配置，在这之前我们都是把src/main.js打包成dist/bundle.js 引入到项目中，那如果有多个main.js类型的文件需要引入呢？ 就需要配置多入口

- webpack-demo

- src

- index.js

- about.js

\*\*注意：\*\*index.js和about.js没有任何关系，都是独立的不相互引用。

src/index.js

var element = document.createElement("span");

element.innerHTML = `hello`;

document.body.appendChild(element);

src/about.js

var element = document.createElement("div");

element.innerHTML = `about`;

document.body.appendChild(element);

config/webpack.base.js

// 其他代码

module.exports = {

// 用对象的方式配置多个入口

entry: {

index: "./src/index.js",

about: "./src/about.js"

},

output: {

// 修改输出路径和文件名，[name]是动态的，读取entry的属性

path: path.join(\_\_dirname, "../dist"),

filename: "js/[name].bundle.js"

},

// 其他代码

}

我们执行npm run build命令，可以看到 dist 的结构如下

- webpack-demo

- dist

- js

- index.bundle.js

- about.bundle.js

- index.html

**提取公共模块**

当在 index 和 about 这两个独立入口文件中, 如果引入了相同的模块, 这些模块会被重复打包, 我们需要提取公共模块!

将 jquery 库分别引入到 index.js 和 about.js 中。

const $ = require('jquery')

执行构建命令

yarn build

查看打包后的 about.bundle.js 和 index.bundle.js 文件源码，会发现它们都把 jquery.js 打包进去了，这样做的后果不敢想象。所以我们需要把类似公共的依赖模块提取到一个单独的文件中。

config/webpack.base.js

// 其他代码

module.exports = {

// 其他代码

// + 提取公共模块配置

optimization: {

splitChunks: {

chunks: 'all' // 提取所有文件的共同模块

}

}

}

再次执行打包

yarn build

可以看到当前项目的公共模块 jquery 的内容已经被打包到一个 独立的 about~index.bundle.js文件中了，当然这个文件名可以[通过配置](https://webpack.docschina.org/plugins/split-chunks-plugin/#splitchunks-name)修改的。

\*\*注意：\*\*公共模块的大小必须大于 30kb才会被独立打包，jquery 的大小是 87kB。

**webpack处理vue**

**安装 vue**

yarn add vue

**vue单文件组件**

**single-file components(单文件组件)** ，文件扩展名为 .vue 的文件，需要安装vetur插件

[单文件组件文档](https://cn.vuejs.org/v2/guide/single-file-components.html)

单文件组件的结构说明

<template>

<div>

<h1>这是单文件组件的模板内容</h1>

</div>

</template>

<script>

// 这是组件的js代码

export default {

data () {

return {

msg: 'hello vue'

}

}

}

</script>

<style>

/\* 这是单文件组件的样式 \*/

h1 {

color: red;

}

</style>

**vue-loader的配置**

Vue Loader 是一个 [webpack](https://webpack.js.org/) 的 loader，它允许你以一种名为[单文件组件](https://vue-loader.vuejs.org/zh/spec.html)的格式撰写 Vue 组件：

* 安装依赖包

yarn add vue-loader vue-template-compiler -D

* webpack配置

// webpack.config.js

const VueLoaderPlugin = require('vue-loader/lib/plugin')

module.exports = {

module: {

rules: [

// ... 其它规则

{

test: /\.vue$/,

loader: 'vue-loader'

}

]

},

plugins: [

// 请确保引入这个插件！

new VueLoaderPlugin()

]

}

* 提供App.vue组件

<template>

<div>我是app</div>

</template>

<script>

export default {

}

</script>

<style>

</style>

* 编写入口文件main.js

import Vue from 'vue'

import App from './App.vue'

new Vue({

el: '#app',

// render函数用于渲染一个组件作为根组件（固定写法）

render (createElement) {

// 把App组件作为根组件

return createElement(App)

}

})

**webpack项目中路由的配置**

**基本步骤**

* 新建views文件夹，存放Home.vue组件和Login.vue组件
* 安装vue-router

yarn add vue-router

* 创建路由实例

import Vue from 'vue'

import App from './App.vue'

import VueRouter from 'vue-router'

import Home from './components/Home.vue'

import Login from './components/Login.vue'

Vue.use(VueRouter)

const router = new VueRouter({

routes: [

{ path: '/home', component: Home },

{ path: '/login', component: Login }

]

})

new Vue({

el: '#app',

// render函数用于渲染一个组件作为根组件（固定写法）

render (createElement) {

// 把App组件作为根组件

return createElement(App)

},

router

})

**抽取路由代码**

把路由功能从main.js中抽取出来

新建router/index.js文件

// 配置所有的路由的功能

// 模块化环境开发

import Vue from 'vue'

import VueRouter from 'vue-router'

import Login from '../components/Login.vue'

import Home from '../components/Home.vue'

Vue.use(VueRouter)

const router = new VueRouter({

routes: [

{ path: '/login', component: Login },

{ path: '/home', component: Home}

]

})

export default router

修改main.js

import Vue from 'vue'

import App from './App.vue'

import router from './router'

new Vue({

el: '#app',

// render函数用于渲染一个组件作为根组件（固定写法）

render (createElement) {

// 把App组件作为根组件

return createElement(App)

},

router

})

**vue-cli 脚手架环境**

通过学习webpack的配置, 我们更深入的理解了脚手架里面的一些配置原理,

下面会介绍一下, 脚手架中移动端的rem配置 和 反向代理配置, 这些都是实际工作中常用的

先通过脚手架创建项目

vue create vue-mobile

在项目根目录新建 vue.config.js 进行配置, 这个vue.config.js 会覆盖默认cli的webpack配置, 非常方便

module.exports = {

devServer: {

port: 3000,

open: true

}

}

运行项目

yarn serve

**rem 布局 - 插件 postcss-pxtorem的配置**

<https://www.cnblogs.com/lml2017/p/9953429.html>

1. 安装插件
2. yarn add lib-flexible postcss-px2rem
3. **在 public 中的 index.html 中删除 meta 标签**

flexible会为页面根据屏幕自动添加<meta name='viewport' >标签，动态控制initial-scale，maximum-scale，minimum-scale等属性的值。

1. 在 src / main.js 中导入插件包
2. // 导入 rem 的 js, 动态的设置了, 不同屏幕的html根元素的 font-size

import 'lib-flexible'

1. 配置 vue.config.js
2. module.exports = {
3. devServer: {
4. port: 3000,
5. open: true
6. },
7. // rem 的配置
8. css: {
9. loaderOptions: {
10. css: {},
11. postcss: {
12. plugins: [
13. require('postcss-px2rem')({
14. // 适配 375 屏幕, 设计图750中量出来的尺寸要 / 2
15. // 配置成 37.5 是为了兼容 没有适配 rem 布局的第三方 ui 库
16. remUnit: 37.5
17. })
18. ]
19. }
20. }
21. }

}

**反向代理的配置说明**

webpack的反向代理, 可以起一个临时的代理服务器, 帮助解决在开发过程中的跨域问题, 就算跨域了也能拿到后台的数据

安装 axios, 发送ajax请求

yarn add axios

发送请求

import axios from 'axios'

export default {

async created () {

const url = `/music/getmv\_by\_tag?g\_tk=5381&loginUin=0&hostUin=0&format=json&inCharset=utf8&outCharset=GB2312&notice=0&platform=yqq.json&needNewCode=0&cmd=shoubo&lan=all`

const res = await axios.get(url)

console.log(res)

}

}

配置代理 (配置vue.config.js文件)

module.exports = {

devServer: {

port: 3000,

open: true,

proxy: {

'/music': {

target: 'https://c.y.qq.com/mv/fcgi-bin/',

pathRewrite: { '^/music': '' }

}

}

},

// rem 的配置

// ....

}