1. Webpack简介

Webpack构建工具，含（scss,es6转换为浏览器识别）

模块打包器（像vue等模块化开发，一个文件中引入所有资源文件，形成文件依赖，其作为入口文件会将依赖形成chunk块（块中将scss转为css等），最后形成bundle

Webpack的配置文件：当运行webpack指令时，会加载里面的配置，去干活

所有构建工具基于nodejs,模块化默认采用commonjs

项目模块化是es6模块化

1. webpack核心概念

entry:打包入口文件

output

loader(处理css img等，处理成webpack能打包的)webpack只能处理js。loader只起到翻译的作用，起不到打包优化和压缩，pugins可以（如图片压缩等），最后形成bundle输出

mode:dev/production 前者插件少后者插件多

webpack只能处理js/json,不能处理css img等

1. 样式资源处理

// resolve 用来拼接绝对路径的方法

const { resolve } = require('path')

module.exports = {

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

output: {

// 输出文件名filename:'js/[name].[contenthash:10].js',

// \_\_dirname nodejs的变量，代表当前文件的目录绝对路径

path:resolve(\_\_dirname,'build')

},

// loader

module: {

// loader配置

rules:[

// 详细loader配置

{

// 匹配哪些文件

test:/\.css$/,

// 使用哪些loader进行处理

// use数组执行顺序，倒序执行

use: [

// 创建style标签，将js的样式资源插入进行，添加到head中生效（浏览器中显示是这样）

'style-loader',

// 将css文件变成commonjs模块加载在js中，里面内容是样式字符串

'css-loader'

]

},

{

test:/\.less$/,

use: [

'style-loader',

'css-loader',

// 将less文件编译成css文件

'less-loader'

]

}

]

},

plugins:[],

mode:'development'

}

1. 图片资源

1. module: {

rules:[

{

// 处理图片资源

// 问题：默认处理不了html中img图片，其处理background:url图片

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

// url-loader依赖file-loader

loader:'url-loader',

options: {

// 图片大小小于8kb(一般8-12kb会做base64处理）

// 优点：减少请求数量（减轻服务器压力）

// 缺点：图片体积会更大(文件请求速度更慢）

limit: 8 \* 1024,

// 问题：因为url-loader默认使用es6模块化解析，而html-loader引入图片是commonjs

// 解析时会出现问题：[objectModule]

// 解决：关闭url-loader的es6模块化，使用commonjs解析

esModule: false,

// 打包后图片文件名很长，图片重命名

// [hash:10]取图片的hash的前10位

// [ext] 取原文件的扩展名

name:'[hash:10][.ext]',

outputPath:'imgs'// 输出目录,打包后不会有css文件，css/less会输出到js文件中

}

},

{

test:/\.html$/,

// 处理html文件的img图片（负责引入img,从而被url-loader进行处理)

loader:'html-loader'

},

]

1. 处理html文件

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

plugins:[

//详细plugins配置

// 功能：默认会创建一个空的html,自用引入打包输出的所有资源（css/js)

new HtmlwebpackPlugin({

// 增加将html内容复制到打包后的html文件中，并自动引入打包输出的所有资源

template:'./src/index.html',

}),

]

1. 打包其他资源

module: {

rules:[

{

exclude:/\.(html|js|css|less|jpg|png|gif)$/,

loader:'file-loader',

options: {

name:'[hash:10][.ext]'

}

},

]

}

1. devServer

// 当修改了webpack配置，新配置要想生效，必须重新webpack服务

devServer: {

contentBase: resolve(\_\_dirname,'build'),

// 使用gzip压缩

compress:true,

// 端口号

port: 3000,

// 自动打开浏览器

open:true,

// 开启HMR功能

hot:true

}

1. 生产环境配置

const { resolve } =require('path')

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

module.exports = {

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

output: {

filename:'dist/build.js',

path:resolve(\_\_dirname,'build')

},

modules: {

rules:[

{

test:/\.less$/,

use:['style-loader','css-loader','less-loader']

},

{

test:/.\css$/,

use:['style-loader','css-loader']

},

{

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

// url-loader是在file-loader基础上将图片转为64位

loader:'url-loader',

options: {

limit: 8\*1024,

name:'[hash:10].[ext]',

esModule: false,

outputPath: 'imgs'

}

},

{

test:/.\html$/,

loader:'html-loader'

},

{

exclude:/\.(html|js|css|jpg|png|gif)/,

loader:'file-loader',

options: {

name:'[hash:10][.ext]'

}

}

]

},

plugins: [

new HtmlWebpackPlugin({

template:'./src/index.html'

})

],

devServer: {

contentBase:resolve(\_\_dirname,'build'),

compress: true,

port:3030,

open:true

}

}

1. 提取css成单独文件和兼容处理

// 將css文件從js中抽離出來

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

module: {

// loader配置

rules:[

{

test:/\.css$/,

use: [

'MiniCssExtractPlugin.loader',

// 提取js中的css成单独文件，style-loader不能要，使用MiniCssExtractPlugin.loader

'css-loader',

//css兼容处理：postcss -->需要安装postcss-loader 其依赖插件post-preset-env帮助识别某些环境和加载某些配置

// 帮postcss找到package.json中browserslist里面的配置，通过配置加载指定的css兼容样式

// 'post-loader',

// 修改loader的配置

{

loader:'postcss-loader',

options: {

ident:'postcss',

plugins:() => [

// postcss的插件

require('postcss-preset-env')()

]

}

}

]

},

plugins:[

// 页面引入了很多的 css, js 文件，js文件放到最后加载，所以会导致我的js引用的css img 在最后才加载。

// 这会导致页面开始的时候会出现没有样式，而图片刚开始会显示失败的问题。因此抽离出来

new MiniCssExtractPlugin({

// 修改打包后的文件名

filename: '/css/built.css'

}),

]

1. css文件压缩

// 压缩css插件optimize-css-assets-webpack-plugin

const OptimizeCssAssetsWebpackPlugin =require('optimize-css-assets-webpack-plugin')件压缩

pugins:[

new OptimizeCssAssetsWebpackPlugin()

]

1. html文件压缩

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

plugins:[

new HtmlwebpackPlugin({

template:'./src/index.html',

// 压缩html代码

minify: {

// 移除空格

collapseWhitespace: true,

// 移除注释

removeComments: true

}

}),

],

1. js语法检查

module:{

rules:[

// 语法检查：eslint-loader (依赖eslint）

// 注意：只检查自己写的源代码，第三方的库是不用检查的

// 设置检查规则

// package.json中eslintConfig中设置

// airbnb -->eslint-config-airbnb-base eslint eslint-pulgin-import

// eslint-disable-line

// eslint-disable-line，js文件中使用，下一行不进行eslint检查

{

test:/\.js$/,

exclude:/node\_modules/,

// 优先执行，同一文件执行不同loader，指明执行顺序

enforce:'pre',

loader:'eslint-loader',

options: {

// 自动修复eslint的错误

fix:true

}

},

]

}

1. js兼容处理

// js兼容处理:babel-loader @babel/preset-env @babel/core

// 1.基本js兼容处理 @babel/preset-env

// 问题： 只能转换基本语法，如promise高级语法不能转换（ie es5)

// 2. 全部js兼容性处理 @babel/polyfill js文件中import '@babel/polyfill'

// 问题：我只要解决部分兼容性问题，但是将所有兼容性代码全部引入，体积太大了

// 需要做兼容性处理的就做：按需加载 core-js

{

test:/\.js$/,

exclude:/node\_modules/,

loader:'babel-loader',

// 正常来讲，一个文件只能被一个loader处理，当一个文件要被多个loader处理，那么一定要指明loader执行

// 的先后顺序

options: {

// 预设： 指示babel做怎样的兼容性处理

preset: [

[

'@babel/preset-env',

{

// 按需加载

useBuiltIns:'usage',

// 指定core-js版本

corejs: {

version: 3

},

// 指定兼容性做到哪个版本浏览器

targets: {

chrome: '60',

firefox:'60',

ie:'9',

safari:'10',

edge:'10'

}

}

]

]

}

}

1. js压缩

mode:'production' 生产环境下会自动压缩js代码，内部UglifyJsPugin插件自动压缩js代码

1. html压缩

plugins:[

new HtmlwebpackPlugin({

template:'./src/index.html',

// 压缩html代码

minify: {

// 移除空格

collapseWhitespace: true,

// 移除注释

removeComments: true

}

}),

],

1. 生产环境配置

const { resolve } =require('path')

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

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

// 设置nodejs环境变量：决定使用browserslist的哪个环境

const OptimizeCssAssetsWebpackPlugin = require('optimize-css-assets-webpack-plugin')

process.env.NODE\_ENV = 'development'

const CommonCssLoader =[MiniCssExtractPlugin.loader,'css-loader',{

loader:'postcss-loader',

options: {

ident:'postcss',

plugins:() => [

require('postcss-preset-env')()

]

}

}]

module.exports = {

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

output: {

filename:'dist/build.js',

path:resolve(\_\_dirname,'build')

},

modules: {

rules:[

{

test:/\.less$/,

use:[...CommonCssLoader]

},

{

test:/.\css$/,

use:[...CommonCssLoader,'less-loader']

},

{

test:/\.js$/,

exclude:/node\_modules/,

enforce:'pre',

loader:'eslint-loader',

options: {

fix:true

}

},

{

test:/\.js$/,

exclude:/node\_modules/,

loader:'babel-loader',

options: {

presets: [

'@babel/preset-env',

{

useBuiltIns: 'usage',

corejs:{version:3},

targets: {

chrome:'60',

firefox:'50'

}

}

]

}

},

{

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

// url-loader是在file-loader基础上将图片转为64位

loader:'url-loader',

options: {

limit: 8\*1024,

name:'[hash:10].[ext]',

esModule: false,

outputPath: 'imgs'

}

},

{

test:/.\html$/,

loader:'html-loader'

},

{

exclude:/\.(html|js|css|jpg|png|gif)/,

loader:'file-loader',

options: {

name:'[hash:10][.ext]'

}

}

]

},

plugins: [

new HtmlWebpackPlugin({

template:'./src/index.html',

minify: {

collapseWhitespace:true,

removeComments:true

}

}),

new MiniCssExtractPlugin({

filename:'css/built.css'

}),

new OptimizeCssAssetsWebpackPlugin()

],

mode:'production',

devServer: {

contentBase:resolve(\_\_dirname,'build'),

compress: true,

port:3030,

open:true

}

}

1. HMR

// HMR: hot module replacement 热模块替换/模块热替换

// 作用：一个模块发生变化，只会重新打包这个模块而不是打包所有模块，极大提升构建速度

// 样式文件：可以使用HMR功能，因为style-loader内部实现了

// js ：默认无HMR功能-->需要修改代码，添加支持HMR功能

// 注意：HMR功能对js的处理，只能处理非入口js文件的其他文件

// js文件中

// if(module.hot) {

// 一旦module.hot为true,说明开启了HMR功能 让HMR功能代码生效

// module.hot.accept('./print.js',function() {

// 方法会监听print.js文件的变化，一旦发生变化，其他模块不会重新打包构建

// 会执行后面的回调函数

// })

// }

// html : 默认无HMR功能，同时导致不能热更新了

// 解决：修改entry入口，将html文件引入，html文件通常只有一个，一个变化，肯定会重新打包该模块，

// html不考虑HMR功能

devServer: {

// 开启HMR功能

hot:true

},

// 解决hot:true,HTML导致不能热更新，更改entry:['./src/js/index.js','./src/index.html']

1. source-map

// source-map: 一种提供源代码到构建后代码映射技术（如果构建后代码出错了，通过映射可以追踪源代码错误）

// source-map：外部，错误源代码准确信息和源代码错误位置

// inline-source-map:内联，只生成一个内联source-map，错误源代码准确信息和源代码错误位置

// hidden-source-map:外部，错误代码错误原因，但是没有错误位置，不能追踪源代码错误

// ，只能提示到构建后代码的错误位置

// eval-source-map:内联，每一个文件都生成对应的source-map，都在eval，错误源代码准确信息和源代码错误位置

// nosources-source-map:外部，错误代码准确信息，但是没有任何源代码信息

// cheap-source-map:外部，错误源代码准确信息和源代码错误位置，只能精确到行，不能精确行出错在哪里

// cheap-module-source-map:外部，module会将loader的source map加入

// 内联和外部的区别：1.外部生成了文件，内联没有 2.内联构建速度更快

1. oneOf

module: {

rules:[

{

test:/\.js$/,

exclude:/node\_modules/,

enforce:'pre',

loader:'eslint-loader',

options: {

fix:true

}

},

{

// 同一文件，以下loader只会匹配一个

// 注意：不能有两个配置处理同一种类型文件 如：eslint-loader和babel-loader将

// eslint-loader置于oneOf外

oneOf: [

{

test:/\.css$/,

use: [

'style-loader',

'css-loader',

{

loader:'postcss-loader',

options: {

ident:'postcss',

plugins:() => [

require('postcss-preset-env')()

]

}

}

]

},

{

test:/\.less$/,

use: [

'style-loader',

'css-loader',

// 将less文件编译成css文件

'less-loader'

]

}

]

}

]

},

1. 缓存

// 文件资源缓存：修改文件名

// 注意：资源在强制缓存期间，不会访问服务器，会直接读取本地缓存

// hash每次构建时会生成唯一哈希值，文件名设置hash值，文件名变了，就会从服务器重新请求，否取缓存

// 问题：因为js和css使用同一个哈希值，如果重新打包，会导致所有缓存失效（可只能改一个文件），导致多余请求

// chunkhash:根据chunk生成的hash值，如果打包来源于同一个chunk,那么hash值就一样

// chunkhash问题：js和css的hash值还是一样的

// 因为css是在js中被引入的，所以同属于一个chunk

// contenthash:根据文件的内容生成的hash值，不同文件hash值一定不一样

output: {

// 文件名设置hash值，文件名变了，就会从服务器重新请求，否取缓存

filename:'build.[hash:10]js', //[chunkhash:10] [contenthash:10]

path: resolve(\_\_dirname,'build')

},

1. tree shaking

// tree shaking:去除无用代码，减少代码体积

// js 前提：1.必须使用es6模块 2.开启production环境 会自动去除

// package.json中"sideEffects":false,所有代码都没有副作用（可以进行tree shaking)

// 问题：可能会把css / @babel/polyfill (副作用）文件干掉

// 解决："sideEffects":["\*.css"]

22.code split

// 文件分割，可并行加载，还可实现按需引入

// 单入口文件情境下：

// 通过js代码，让某个文件单独打包成一个chunk

// import 动态导入进入，能将某个文件单独打包

// import(/\* webpackChunkName:'test' \* /'./test').then({mul,cunt} => {}).catch(err => {})

// /\* webpackChunkName:'test' \* /为打包后文件取名

// entry:'./src/index.js',// 单入口 index中引入test

entry: {

index:'./src/index.js',//情景：js文件不相互依赖,不存在引入

test:'./src/test.js'

},

// 1.单入口：可以将node\_modules中代码(依赖）单独打包一个chunk最终输出

// 2.自动分析多入口chunk中，没有公共的文件。如果有会打包成一个单独chunk

optimization: {

splitChunks: {

chunks:'all'

}

},

1. lazy loading

// 懒加载 ：条件引入，需要的时候引入

// 不会被重复加载，会读取缓存

// import(/\* webpackChunkName: 'test' \*/'./test').then({mul} => {console.log(mul(4,5))})

//预加载：prefetch,会在使用之前，提前加载js文件

// import(/\* webpackChunkName: 'test'，webpackPrefetch:true \*/'./test').then({mul} => {console.log(mul(4,5))})

//正常加载可以认为是并行加载（同一时间加载多个文件） 预加载prefetch ：

// 等其它资源加载完毕，浏览器空闲了，再偷偷加载资源 兼容性差

1. pwa

const WorkboxWbpackPlugin = require('workbox-webpack-plugin')

plugins:[

new WorkboxWbpackPlugin.GenerateSW({

// 1. 帮助serviceworker快速启动

// 2. 删除旧的serviceWorker

// 入口文件：注册serviceworker

// 处理兼容文件

// if('serviceworker' in navigator) {

// window.addEventListener('load', () => {

// navigator.service.register('/serviec-worker.js')

// }.then(() => {}).catch(() => {})

// }

// 1.eslint不认识window,navigator全局变量

// 解决： 需要修改package.json中eslintConfig配置

// "env": {"browser": true} // 支持浏览器端全局变量

// 2.sw 代码必须运行在服务器上 dist打包文件，http-server

})

],

1. 多进程

module: {

rules:[

{

test:/\.js$/,

exclude:/node\_modules/,

//把这个 loader 放置在其他 loader 之前， 放置在这个 loader 之后的 loader 就会在一

// 个单独的 worker【worker pool】 池里运行，一个worker 就是一个nodeJS 进程

// 【node.js proces】，每个单独进程处理时间上限为600ms，各个进程的数据交换也会限制在这个时间内。

//开启多进程打包，进程启动大概为600ms,进程通信也有开销。只有工作耗时较长，才需要多进程打包

loader:'thread-loader',

options: {

workers: 2 // 进程2个

}

},

{

test:/\.js$/,

exclude:/node\_modules/,

loader:'babel-loader',

}

]

},

1. Externals

externals: {

// 拒绝jquery被打包进来

jquery: 'jquery' // 库不打包，使用cdn链接

}

1. Dll

plugins:[

new webpack.DllReferencePlugin({

// 告诉webpack哪些库不参与打包，同时使用时的名称也得变

manifest:resolve(\_\_dirname,'dll/manifest.json')

}),

// 将某个文件打包输出去，并在html中自动引入该资源

new AddAssetHtmlWebpackPlugin({

filePath:resolve(\_\_dirname,'dill/jquery.js')

})

],

1. 性能优化

##webpack性能优化

\*开发环境性能优化

\*生产环境性能优化

##开发环境性能优化

\*优化打包构建速度

\*HMR

\*优化代码调试

\*source-map

##生产环境性能优化

\*优化打包构建速度

\*oneOf

\*babel缓存

\*多进程打包

\*externals

\*dll

\*优化代码运行时的性能

\*缓存（hash-chunkhash-contenthash)

\*tree shaking(条件：es6 mode:'production') 搭配sideEffects使用

\*代码分割（单入口 一个chunk-一个bundle，多入口）

\*懒加载/预加载

\*pwa

ts-webpack配置

{

module: {

rules: [

{

test: /\.vue$/,

loader: 'vue-loader',

options: { /\* ... \*/ },

},

{

test: /\.ts$/,

loader: 'ts-loader',

options: {

appendTsSuffixTo: [/\.vue$/],

}

},

],

}

}

ts-loader 会检索当前目录下的 tsconfig.json 文件

tsconfig.json配置

{

// 编译选项

"compilerOptions": {

// 输出目录

"outDir": "./output",

// 是否包含可以用于 debug 的 sourceMap

"sourceMap": true,

// 以严格模式解析

"strict": true,

// 采用的模块系统

"module": "esnext",

// 如何处理模块

"moduleResolution": "node",

// 编译输出目标 ES 版本

"target": "es5",

// 允许从没有设置默认导出的模块中默认导入

"allowSyntheticDefaultImports": true,

// 将每个文件作为单独的模块

"isolatedModules": false,

// 启用装饰器

"experimentalDecorators": true,

// 启用设计类型元数据（用于反射）

"emitDecoratorMetadata": true,

// 在表达式和声明上有隐含的any类型时报错

"noImplicitAny": false,

// 不是函数的所有返回路径都有返回值时报错。

"noImplicitReturns": true,

// 从 tslib 导入外部帮助库: 比如\_\_extends，\_\_rest等

"importHelpers": true,

// 编译过程中打印文件名

"listFiles": true,

// 移除注释

"removeComments": true,

"suppressImplicitAnyIndexErrors": true,

// 允许编译javascript文件

"allowJs": true,

// 解析非相对模块名的基准目录

"baseUrl": "./",

// 指定特殊模块的路径

"paths": {

"jquery": [

"node\_modules/jquery/dist/jquery"

]

},

// 编译过程中需要引入的库文件的列表

"lib": [

"dom",

"es2015",

"es2015.promise"

]

}

}