**Webpack**

**安装webpack**

# 一．构建项目结构

1.进入命令行，建立文件夹，进入文件夹

**mkdir webpack\_demo**

**cd webpack\_demo**

2.在node.js安装完成后，安装webpack

**npm install -g webpack**

3.具体项目安装，主要目的生成package.json

**npm init –y(默认项目信息，生成package-json)**

**或者**

**npm init(生成package.json)**

4.安装webpack –dev

**npm init --save-dev webpack**

**npm i webpack webpack-cli –save-dev(生成package-lock.json，node\_modules)**

**package.json修改配置打包指令build**

"scripts": {

    "build":"webpack"

  }

5.查看webpack版本

**webpack -v**

6.进入vscode，在根目录建立文件夹src和dist

**src文件夹：源代码，用于开发环境；**

**dist文件夹：打包好的文件夹，用于生产环境；**

7.编写程序文件

**/dist/index.html**

8.入口文件

**/src/entery.js**

9.webpack命令打包，自动生成打包文件bundle.js

**webpack src/entery.js -o dist/bundle.js(直接指令copy文件)**

10.新建webpack.config.js(最基本的entry&output，可单文件也可多文件)

const  path  =  require('path');

module.exports={

    mode: 'development',

// 生产环境(未压缩)

    // mode: 'production' ,

// 开发环境(压缩)

    entry:{

        entery:'./src/entery.js',

        entery2:'./src/entery2.js'

    },

    output:{

        path,

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

        filename:'[name].js'

    },

    // 下面的开启空文件会报错

    // module:{},

    // plugins:{},

    // devServer:{}

}

11.npm i path –save路径文件

12.npm i webpack-dev-server –save-dev(下载后才能执行webpack-dev-server，host可以通过ipconfig查看，host可不配置默认为localhost)

// 配置webpack开发服务功能：服务和热更新

    devServer:{

        //设置基本目录结构

        contentBase:path.resolve(\_\_dirname,'dist'),

        //服务器的IP地址，可以使用IP也可以使用localhost(通过ipconfig可查看ip)

        host:'192.168.43.236',

        //服务端压缩是否开启

        compress:true,

        //配置服务端口号

        port:1717

    }

13.在package.json配置script后可用npm run server压缩并打开

"scripts": {

    "server":"webpack-dev-server"

  }

**修改src原始js数据，打包数据实时更新，页面效果实现热更新，同步刷新效果。**

# 二．配置相关类

14.实现css引入(建立css文件，引入js中，下载style-loader和css-loader，配置module)

**npm install --save-dev style-loader css-loader（下载loader命令）**

import   './css/index.css';

**webpack.config.js配置module，实现css压缩插入，更改实时热更新~**

**方法一：**

module: {

        rules: [

            {

                test: /\.css$/,

                use: [

                    'style-loader',

                    'css-loader'

                ]

            }

        ]

    }

**方法二：**

module: {

        rules: [

            {

                test: /\.css$/,

                loader: [

                    'style-loader',

                    'css-loader'

                ]

            }

        ]

    }

**方法三：**

module: {

        rules: [

            {

                test: /\.css$/,

                use: [{

                    loader: 'style-loader'

                }, {

                    loader: 'css-loader'

                }]

            }

        ]

    }

15.压缩js(压缩代理，压缩获取js与开发环境压缩效果相同 mode: 'production')

**npm i -D uglifyjs-webpack-plugin(安装代码压缩插件)**

**在webpack.config.js声明添加压缩插件代码，执行webpack命令可压缩js代码**

const uglifyJsPlugin = require('uglifyjs-webpack-plugin');

// 插件，用于生产模版和各项功能

    plugins: [

        new uglifyJsPlugin()

    ]

16.正式进入html压缩打包，删除dist文件夹，src文件夹保留css/index.css，entery.js，index.html，在index.html不用引入js文件entery.js，取消之前设置的css压缩配置。

**npm install --save-dev html-webpack-plugin(安装html代码压缩插件)**

**在webpack.config.js声明添加html压缩插件代码**

const htmlPlugin= require('html-webpack-plugin');

**plugins插件配置：**

// 插件，用于生产模版和各项功能

    plugins: [

        // new uglifyJsPlugin()

        new htmlPlugin({

            minify:{

                removeAttributeQuotes:true //去掉属性的双引号

            },

            hash:true,  //开发中js有缓存效果，所以加入hash，这样可以有效避免缓存JS

            template:'./src/index.html' //要打包的html模版路径和文件名称

        })

    ]

**使用webpack指令打包压缩，生成dist文件夹(包含压缩的js与html文件，entry.js包含css相关信息，index.html压缩去除了双引号&以hash方式引入js文件)**

17.css中图片处理(基于背景图片)

**npm install --save-dev file-loader url-loader(file-loader：解决引用路径的问题；** **url-loader：会将引入的图片编码，生成dataURl)**

**配置module，跟css的类似，limit限制大小，打包后小于限制大小，把小图片打包成Base64格式写入js中，大于则直接单独在dist中生成随机符号的图片**

module: {

        rules: [

            {//css

                test: /\.css$/,

                use: [{

                    loader: 'style-loader'

                }, {

                    loader: 'css-loader'

                }]

            },

            {//图片

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

                use: [{

                    loader: 'url-loader',

                    options: {

                        limit: 5000，

outputPath: 'images/'                    }

                }]

            }

        ]

    }

18.css分离与图片路径处理

**npm install extract-text-webpack-plugin@next(安装分离css插件)**

**webpack.config.js配置：**

// css分离

const extractTextPlugin = require("extract-text-webpack-plugin");

plugins: [

        // 分离css目录及名称设置

        new extractTextPlugin("css/index.css")

    ]

 module: {

        rules: [

            {//css

                test: /\.css$/,

                // css直接打包进js

                // use: [{

                //     loader: 'style-loader'

                // }, {

                //     loader: 'css-loader'

                // }]

                // css分离打包

                use: extractTextPlugin.extract({

                    fallback: "style-loader",

                    use: "css-loader"

                  })

            }

        ]

    }

**完成css分离压缩，当img过大超过limit，由于路径引入问题(相对路径)img不能在网页获取显示，需要再加配置：**

**webpack.config.js配置(路径与host一致)：**

// 声明公共路径，与host一致

var website ={

    publicPath:"http://127.0.0.1:1717/"

}

output: {

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

        filename: '[name].js',

        // 公共路径配置

        publicPath:website.publicPath

    }

**再次webpack打包，引入路径变为从publicPath访问（绝对路径），网页端可获取显示外部img图片。**

19.html图片打包(src引用图片)

**script添加build可用npm run build打包(也可直接webpack打包)，npm run server打包开启路由**

"scripts": {

    "server": "webpack-dev-server",

    "build":"webpack"

  }

**npm install html-withimg-loader --save(安装html-withimg-loader插件)**

**配置webpack.config.js:**

 module: {

        rules: [

            {//图片

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

                use: [{

                    loader: 'url-loader',

                    options: {

                        limit: 5000,

                        outputPath:'images/',

                        //添加esModule为false,才能用html-withimg-loader配置压缩src路径

                        esModule: false,

                    }

                }]

            },

            {//html-withimg-loader插件，配置压缩src引入的图片

                test: /\.(htm|html)$/i,

                loader: 'html-withimg-loader'

            }

        ]

    }

20.less文件的打包与分离

**npm install --save-dev less（安装less）**

**npm install --save-dev less-loader（安装less-loader打包服务）**

**配置webpack.config.js:**

module: {

        rules: [

            {//配置less

                test: /\.less$/,

                use: [{

                    loader: "style-loader" // creates style nodes from JS strings

                }, {

                    loader: "css-loader" // translates CSS into CommonJS

                }, {

                    loader: "less-loader" // compiles Less to CSS

                }]

            }

        ]

    }

**编写less文件引入html，js文件，npm run build打包后，less文件包含在压缩的js文件中**

**然后进行less文件分离配置：**

module: {

        rules: [

            {//配置less

                // 未分离less（压缩后less在压缩的js文件中）

                // test: /\.less$/,

                // use: [{

                //     loader: "style-loader" // creates style nodes from JS strings

                // }, {

                //     loader: "css-loader" // translates CSS into CommonJS

                // }, {

                //     loader: "less-loader" // compiles Less to CSS

                // }]

                //less文件分离

                test: /\.less$/,

                use: extractTextPlugin.extract({

                    use: [{

                        loader: "css-loader"

                    }, {

                        loader: "less-loader"

                    }],

                    // use style-loader in development

                    fallback: "style-loader"

                })

            }

        ]

    }

**打包压缩后less样式进入css文件中，实现less文件分离**

21.sass文件的打包分离

**npm install --save-dev node-sass（安装node-sass）**

**npm install --save-dev sass-loader（安装sass-loader）**

**sass配置与分离类似less：**

module: {

        rules: [

            {//配置scss

                // 未分离scss（压缩后scss在压缩的js文件中）

                // test: /\.scss$/,

                // use: [{

                //     loader: "style-loader" // creates style nodes from JS strings

                // }, {

                //     loader: "css-loader" // translates CSS into CommonJS

                // }, {

                //     loader: "sass-loader" // compiles Sass to CSS

                // }]

                // scss分离文件

                test: /\.scss$/,

                use: extractTextPlugin.extract({

                    use: [{

                        loader: "css-loader"

                    }, {

                        loader: "sass-loader"

                    }],

                    // use style-loader in development

                    fallback: "style-loader"

                })

            }

        ]

    }

**打包压缩后scss样式进入css文件中，实现scss文件分离**

22.自动处理css3属性前缀

**npm install --save-dev css-loader style-loader**

**npm install --save-dev postcss-loader autoprefixer postcss**

**新建postcss.config.js，写入：**

//配置css3前缀

module.exports = {

    plugins: [

        require('autoprefixer')({overrideBrowserslist: ['> 0.15% in CN']})// 自动添加css前缀

    ]

}

**webpack.config.js中配置postcss-loader：**

{//css

                test: /\.css$/,

                // css直接打包进js

                // use: [{

                //     loader: 'style-loader'

                // }, {

                //     loader: 'css-loader',

                //     options: {

                //         modules: true

                //     }

                // }, {

                //     loader: "postcss-loader"

                // }

                // ],

                // css分离打包

                use: extractTextPlugin.extract({

                    fallback: "style-loader",

                    use: [

                        { loader: "css-loader", options: { importLoaders: 1 } },

                        //自动处理css3属性前缀载入postcss-loader

                        "postcss-loader"

                    ]

                })

            }

**在css中写入新样式，压缩打包，自动配置css3属性前缀，效果如下：**

-webkit-transform: rotate(45deg);

        -ms-transform: rotate(45deg);

            transform: rotate(45deg);

23.消除未使用的css

**npm i -D purifycss-webpack purify-css（安装PurifyCSS-webpack）**

**webpack.config.js中配置**

//消除未使用的css

const glob = require('glob');

const PurifyCSSPlugin = require("purifycss-webpack");

**plugins配置：**

//清除未使用的css

        new PurifyCSSPlugin({

            // Give paths to parse for rules. These should be absolute!

            paths: glob.sync(path.join(\_\_dirname, 'src/\*.html')),

        })

**插入无关css样式，打包压缩后无效css样式未显示在压缩后的css文件中**

24.给webpack增加bable支持

**Babel其实是一个编译JavaScript的平台，可通过配置将一些高级版本语言转化为es5进行编译。**

**基于babel7配置：**

**npm install -D babel-loader @babel/core @babel/preset-env webpack（babel安装）**

**npm i -D @babel-preset-react**

**npm i -D @babel/plugin-proposal-class-properties**

**babel配置经常会报错，多版本需先卸载老版本**

**npm un再重新下载！！！！！！！**

**Webpack.config.js>module>rules配置：**

{//babel配置

                test:/\.js$/,   //匹配JS文件

                loader:'babel-loader',

                exclude:/node\_modules/   //排除node\_modules目录

            }

**新建.babelrc文件配置：**

{

    "presets":["@babel/preset-env","@babel/preset-react"]

}

**在js文件中写入es5+语言，打包压缩后自动转为es5通用语言**

25.打包调试

**在使用webpack时只要通过简单的devtool配置，webapck就会自动给我们生产source maps 文件，map文件是一种对应编译文件和源文件的方法，让我们调试起来更简单。**

source-map:在一个单独文件中产生一个完整且功能完全的文件。这个文件具有最好的source map,但是它会减慢打包速度；

cheap-module-source-map:在一个单独的文件中产生一个不带列映射的map，不带列映射提高了打包速度，但是也使得浏览器开发者工具只能对应到具体的行，不能对应到具体的列（符号）,会对调试造成不便。

eval-source-map:使用eval打包源文件模块，在同一个文件中生产干净的完整版的sourcemap，但是对打包后输出的JS文件的执行具有性能和安全的隐患。在开发阶段这是一个非常好的选项，在生产阶段则一定要不开启这个选项。

cheap-module-eval-source-map:这是在打包文件时最快的生产source map的方法，生产的 Source map 会和打包后的JavaScript文件同行显示，没有影射列，和eval-source-map选项具有相似的缺点。

如果大型项目可以使用source-map，如果是中小型项目使用eval-source-map就完全可以应对，需要强调说明的是，source map只适用于开发阶段，上线前记得修改这些调试设置。

**配置：在开发模式下配置webpack.config.js devtool**

module.exports = {

    devtool:'eval-source-map'

# 三．实践技巧配置

26.实践技巧：开发和生产并行设置

**一个项目中是有开发环境和生产环境的，这两个环境的依赖也是不同的**

* 开发依赖：只在开发中用来帮助你进行开发，简化代码或者生成兼容设置的以来包。你可以打开package.json来查看，devDependencies的下面的这些包为开发使用的包。这些包在生产环境中并没有用处。
* 生产依赖：就是比如我们的js使用了jquery，jquery的程序要在浏览器端起作用，也就是说我们最终的程序也需要这个包，这就是生产依赖。这些包在dependencies中。

**npm安装（以jquery为例，三种安装方法）：**

**删除node\_modules模块，模拟多人开发**

* 1. **npm install jquery（**在package.json中并不存在这个包的依赖**）**
  2. **npm install jquery –save（**存在于package.json的dependencies中，也就是说它是生产环境需要依赖的包（上线时需要的包）**）**
  3. **npm install jquery --save-dev（**存在于package.json的devDependencies中，也就是说它是开发环境中需要的，上线并不需要这个包的依赖**）**

**安装全部项目依赖包：**

**npm install**

**安装生产环境依赖包：**

**npm install –production**

**配置生产和开发并行：**

**在webpack.config.js配置：**

console.log(encodeURIComponent(process.env.type));

// 声明公共路径，与host一致

if (process.env.type == "build") {//生产环境配置

    var website = {

        publicPath: "http://jspang.com:1717/"

    }

} else {//开发环境

    var website = {

        publicPath: "http://127.0.0.1:1717/"

    }

}

**在package.json修改配置：**

"scripts": {

    "server": "webpack-dev-server --open",

    "build": "set type=build&webpack",

    "dev":"set type=dev&webpack"

  }

**当运行npm run build 实行生产模式打包（上线），打包后html公共域名为jspang…**

**当运行npm run dev 实行开发模式打包（开发阶段），打包后html公共域名为127.0…**

**使用npm run server 启动项目**

27.实战技巧：webpack模块化配置

**example:js实现模块化开发**

**新建src/jspang.js:**

// js实现模块化开发

function jspang(){

    alert('jspang.com:'+'webpack');

}

module.exports=jspang;

**src/entery.js配置:**

//js模块化开发引入

import jspang from './jspang.js';

jspang();

**导出-引入 独立js文件构建模块化开发**

**webpack模块实现模块化开发（**把webpack.config.js中的entry入口文件进行模块化设置，单独拿出来制作成一个模块**）：**

**src下新建文件夹webpack\_config,新建文件entry\_webpack.js~~~~~~**

**src/webpack\_config/entry\_webpack.js配置：**

//webpack  entry模块化开发

//声明entry变量

const entry ={};

//声明路径属性

entry.path={

    entry:'./src/entery.js'

}

//进行模块化

module.exports =entry;

**webpac.config.js配置：**

//webpack entry 模块化开发引入

const entry = require("./webpack\_config/entry\_webpack.js")

**module：**

module.exports = {

    //打包调试配置devtool，只用于开发阶段，上线前删除

    devtool: 'eval-source-map',

    mode: 'development',

    // 生产环境(未压缩)

    // mode: 'production' ,

    // 开发环境(压缩)

    //entry配置

    // entry: {

    //     entery: './src/entery.js',

    // },

    //使用模块化开发entry后配置

    entry: entry.path

**最后正常npm run dev打包~**

**注意：配置时注意路径和名称，容易出错！！！**

**require与import区别：**

****

28.实战技巧：优雅打包第三方类库

**引用第三方库的方法有很多种，但是有些并不是很优雅，还有些方法会出现打包问题，以jquery引入为例，两种方法：**

**方法一：**

**npm install --save jquery**

**entery.js配置：**

window.$=$

import $ from 'jquery';

**修改语言即可**

//jquery语言

$('#title').html('Hello JSpang');

**方法二（用plugin引入）：**

ProvidePlugin是一个webpack自带的插件，Provide的意思就是装备、提供。因为ProvidePlugin是webpack自带的插件，所以要先再webpack.config.js中引入webpack。

**webpack.config.js中配置：**

//webpack引入

const webpack = require('webpack');

**plugins设置**

plugins: [

        //优雅打包ProvidePlugin插件引入

        new webpack.ProvidePlugin({

            $:"jquery"

        })

    ]

**即可使用jquery库**

29.实战技巧：watch的正确使用方法

**webpack --watch (基础指令，保存后压缩文件自动更新)**

**如果上述指令报错用下方配置：**

**更高级配置webpack.config.js：（若报错取消此配置）**

watchOptions: {

        poll: 1000,//监测修改的时间(ms)

        aggregeateTimeout: 500, //防止重复按键，500毫米内算按键一次

        ignored: /node\_modules/,//不监测

    }

**然后webpack –watch监听**

30.为文档添加版权名字标记（js注释）

**Webpack.config.js配置plugins：**

//为每个 chunk 文件头部添加 banner

        new webpack.BannerPlugin('king版权所有~~~')

**打包更新后压缩js文件（entry.js）头部出现注释：**

/\*! king版权所有~~~ \*/

/\*\*\*\*\*\*/ (function(modules) { // webpackBootstrap

/\*\*\*\*\*\*/  // The module cache

/\*\*\*\*\*\*/  var installedModules = {};