webpack是一个前端资源构建工具, 一个静态模块打包器

在浏览器中并不能识别less,sass,scss等css预编译器语言

也无法识别import模块语法

补充 npm i xxx -D 开发依赖

所有的构建工具都是基于node运行的 所以模块化默认采用commonjs

## webpack五个核心概念

### Entry

入口 (Entry).webpack以哪个文件为入口开始打包,分析生成构件图

### Output

输出(Output)只是webpack打包后所有资源输出到哪里以及如何命名

### Loader

Loader让webpack能去处理那些非JavaScript文件(webpack自身只能理解JavaScript)

通常情况下一种文件只能由一个loader处理

当一个文件需要被多个loader处理的时候,一定要指定loader执行的先后顺序,比如js文件需要先执行eslint再执行babel

这个时候需要给eslint loader中添加一条属性: **enforce:”pre”** 表示优先执行(加上这个属性会让eslint这个loader放在所有的loader之前执行)

### 4.Plugins

插件(Plugins)可以用于执行更广的任务. 插件的范围包括从打包优化和压缩,一直到重新定义环境中的变量

### 5.Mode

模式(Mode)只是webpack使用相应的模式配置

development 开发模式

production 生产模式



### webpack 运行指令

webpack首先全局安装,然后在项目中安装

具体指令

npm i webpack webpack-cli -g

npm i webpack webpack-cli -D // 表示开发依赖

开发环境: webpack ./src/index.js -o ./build/built.js –mode=development

webpack 会以index为入口文件打包到build/built.js这个文件中 development表示开发打包

生产环境: webpack ./src/index.js -o ./build/built.js –mode= production

webpack 会以index为入口文件打包到build/built.js这个文件中 development表示生产打包(生产环境会压缩js代码)

生产环境和开发环境打包的作用是将ES6的模块化转化为浏览器能识别的代码

注意 webpack无法在不配置的情况下处理css和图片 只能处理js和json

开发环境中代码没有压缩,可以明确的找到这块代码打包自那个文件

生产环境中代码会压缩,正常人无法识别

### webpack打包样式资源(css)

webpack.config.js 目录与src为同一层级

作用: 指示webpack怎么干活,干那些活(当运行webpack的时候会加载里面的配置)

基本配置

var path = require("path")

module.exports = {

    // webpack配置

    // 入口起点

    entry: "./main/index.js",

    // 输出位置

    output: {

        // 输出文件名

        filename: "LidpVideo.js",

        // 输出路径

        // \_\_dirname 当前文件的绝对路径

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

    },

    // loader配置

    module: {

        rules: [

            // 详细loader配置

        ]

    },

    // 插件配置

    plugins: [

        // 详细插件配置

    ],

    // 开发模式

    mode: "development"

    // 生产模式

    // mode: "production"

}

#### 处理css文件

css的引入顺序与index.js中的引入顺序一致

处理css的loader名字是 css-loader

module: {

        // 注意 这里的loader都需要下载

        rules: [

            {

                test: /\.css$/,

                use: [

                    "style-loader",

                    "css-loader"

                ]

            },

            {

                test: /\.scss$/,

                use: [

                    "style-loader",

                    "css-loader",

                    "sass-loader"

                ]

            }

        ]

    }

#### 处理HTML文件

loader: 1. 下载 2. 使用(配置loader)

plugins: 1. 下载 2. 引入 3. 使用

处理HTML文件的plugins名字是 html-webpack-plugin

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

var path = require("path")

var HtmlWebpackPlugin =require("html-webpack-plugin")

module.exports = {

    // 入口

    entry: "./src/index.js",

    // 出口

    output: {

        // 输出文件名

        filename: "index.js",

        // 输出路径

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

    },

    // 配置loader

    module: {

        // 注意 这里的loader都需要下载

        rules: [

            {

                test: /\.css$/,

                use: [

                    "style-loader",

                    "css-loader"

                ]

            },

            {

                test: /\.scss$/,

                use: [

                    "style-loader",

                    "css-loader",

                    "sass-loader"

                ]

            }

        ]

    },

    plugins: [

        // 插件详细配置

        new HtmlWebpackPlugin({

            // 复制"./src/index.html"文件, 并且自动引入打包输出的资源

            template: "./src/index.html"

        })

    ],

    mode: "development"

}

#### 打包图片资源

图片处理loader名字是: url-loader(依赖file-loader)

需要下载两个包分别是: url-loader file-loader

var path = require("path")

var HtmlWebpackPlugin = require("html-webpack-plugin")

module.exports = {

  // 入口

  entry: "./src/index.js",

  // 出口

  output: {

    // 输出文件名

    filename: "index.js",

    // 输出路径

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

  },

  // 配置loader

  module: {

    // 注意 这里的loader都需要下载

    rules: [

      {

        test: /\.css$/,

        use: [

          "style-loader",

          "css-loader"

        ]

      },

      {

        test: /\.scss$/,

        use: [

          "style-loader",

          "css-loader",

          "sass-loader"

        ]

      },

        // 配置图片处理

        // 会有一个问题,可以处理css中的图片资源,但是无法处理html中的图片资源,所以在下方需要再引入一个loader

      {

        test: /.(jpg|png|gif)$/,

        // 此处需要下载 file-loader  url-loader

        use: [

            {

                loader:  "url-loader",

                options: {

                    limit: 8 \* 1024,

                    // 因为html-loader引入图片采用的是commonjs(require)  而 url-loader采用的是es6(import)

                    // 所以解析是会出现个问题  html中的<img src="[object Module]" alt="">

                    // 解决方法 : 关闭url-loader中的es6解析

                    esModule:false

                }

            }

        ]

      },

        // html-loader 处理html中的image文件,负责image文件的引入 从而能够被url-loader处理

        {

            test:/.html$/,

            use:"html-loader"

        }

    ]

  },

  plugins: [

    // 插件详细配置

    new HtmlWebpackPlugin({

      // 复制"./src/index.html"文件, 并且自动引入打包输出的资源

      template: "./src/index.html"

    })

  ],

  mode: "development"

}

#### 打包其他资源

rules:[

...,

{

// 也可以用test

// 排除

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

loader:”file-loader”,

options:{

// 修改输出文件名

name:”[hash:10].[ext]”

}

}

]

#### devServer开发服务器(热编译)

自动编译,自动打开浏览器,自动刷新浏览器

启动devServer指令为 npx webpack-dev-server

需要下载 webpack-dev-server

npm i webpack-dev-server-D

var path = require("path")

var HtmlWebpackPlugin =require("html-webpack-plugin")

module.exports = {

    // 入口

    entry: "./src/index.js",

    // 出口

    output: {

        // 输出文件名

        filename: "index.js",

        // 输出路径

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

    },

    // 配置loader

    module: {

        // 注意 这里的loader都需要下载

        rules: [

            {

                test: /\.css$/,

                use: [

                    "style-loader",

                    "css-loader"

                ]

            },

            {

                test: /\.scss$/,

                use: [

                    "style-loader",

                    "css-loader",

                    "sass-loader"

                ]

            }

        ]

    },

    plugins: [

        // 插件详细配置

        new HtmlWebpackPlugin({

            // 复制"./src/index.html"文件, 并且自动引入打包输出的资源

            template: "./src/index.html"

        })

    ],

    mode: "development",

    // 开发服务器 热编译 自动刷新

    // 只会在内存中编译打包,不会有本地输出

    // 启动devServer指令为 npx webpack-dev-server

    // 需要下载 webpack-dev-server

    devServer: {

        // 端口号

        port:8000,

        // 项目构建后的路径

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

        // 启用g-zip压缩

        compress:true,

        // 自动打开浏览器

        open:true

    }

}

### css兼容处理

package.json

{

  "scripts": {

    "dev": "npx webpack-dev-server",

    "build": "webpack --mode development",

    "build2": "webpack --mode production"

  },

  "browserslist":{

    "development":[

      "last 1 chrome version",

      "last 1 firefox version",

      "last 1 safari version"

    ],

    "production":[

      ">0.01%",

      "not dead",

      "not op\_mini all"

    ]

  }

}

webpack.config.js

// 下载插件  mini-css-extract-plugin 本插件为导出css为单独文件

/\*

\* module:{

\*   rules:[

\*       {

\*           test:/\.css$/,

\*           use:[

\*               // 因为style-loader会将css-loader处理过后的代码插入到html中

\*               // 所以此处不适用style-loader

\*               // 使用 MiniCssExtractPlugin.loader 代替style-loader  作用为提取js中的css到单独文件中 并且会引入到html中

\*               MiniCssExtractPlugin.loader

\*               ,"css-loader"

\*           ]

\*       }

\*   ]

\* }

\*

\*

\* \*/

var {resolve} = require("path")

var HtmlWebpackPlugin = require("html-webpack-plugin")

var MiniCssExtractPlugin = require("mini-css-extract-plugin")

// process.env.NODE\_ENV = "development"

module.exports = {

    entry: "./src/index.js",

    output: {

        filename: "js/build.js",

        path: resolve(\_\_dirname,"dist")

    },

    module: {

        rules: [

            {

                test: /\.css$/,

                use:[

                    MiniCssExtractPlugin.loader,

                    "css-loader",

                    /\*\*

                     \* css兼容性处理需要用到 postcss

                     \* 在websocket中使用的话需要用到 postcss-loader

                     \* 还需要一个插件 postcss-preset-env  这个插件可以帮助postcss识别使用环境 从而精确到兼容ie几

                     \*

                     \*

                     \*帮postcss找到package.json中browserslist里面的配置,通过配置加载指定的css兼容性样式

                        // 可以去GitHub上面找  browserslist 搜详细配置

                      "browserslist":{

                            // 开发环境

                            // 需要设置nodejs的环境变量

                            // 在本文件的上方

                            // process.env.NODE\_ENV = "development"

                          "development":[

                              "last 1 chrome version",  // 最后一个版本的谷歌浏览器

                              "last 1 firefox version", // 最后一个版本的火狐浏览器

                              "last 1 safari version"   // 最后一个版本的苹果浏览器

                          ],

                          // 生产环境  默认看生产环境

                          "production":[

                              ">0.2%", // 用户>0.2%的浏览器

                              "not dead",  // 不要已经去世的浏览器

                              "not op\_mini all"  // 不要op\_mini浏览器全部版本

                          ]

                      }

                     \*/

                    {

                        // 修改loader的配置

                        loader: "postcss-loader",

                        options: {

                            ident:"postcss",  // 固定值

                            plugins:()=>[

                                // postcss的插件

                                require("postcss-preset-env")()

                            ]

                        }

                    },

                ]

            }

        ]

    },

    plugins: [

        new HtmlWebpackPlugin({

            template: "./src/index.html"

        }),

        new MiniCssExtractPlugin({

            filename: "css/dist.css"

        })

    ],

    mode: "production",

    devServer: {

        port:3000,

        contentBase:resolve(\_\_dirname,"dist"),

        compress:true,

        // 自动打开浏览器

        open:true

    }

}

### 压缩css

插件名 optimize-css-assets-webpack-plugin

plugins: [

        new HtmlWebpackPlugin({

            template: "./src/index.html"

        }),

        new MiniCssExtractPlugin({

            filename: "css/dist.css"

        }),

        //压缩css

        new OptimizeCssassetsWebpackPlugin()

    ],

### js语法检查 ESLint

语法检查:可以检查常见的语法错误,统一代码风格

eslint-loader eslint

设置检查规则:

在package.json中eslintConfig中设置

推荐使用airbnb规则

eslint eslint-config-airbnb-base eslint-plugin-import

在loader中写

{

test: /\.js$/,

// 这里不检查node\_modules下的文件

exclude:/node\_modules/

loader:”eslint-loader”,

options:{

// 自动修复eslint的错误

fix:true

}

}

package.json中添加

“eslintConfig”:{

“extends”:”airbnb-base”

}

如果希望eslint的规则对下一行失效

添加如下注释则会忽略

// eslint-disable-next-line

### js兼容性处理

兼容处理需要使用到 babel-loader @babel/preset-env @babel/core @babel/polyfill core-js

babel-loader js兼容处理的loader

@babel/preset-env 基本兼容

@babel/core 上面两个包的依赖项

@babel/polyfill 做全部语法兼容用

core-js 做按需加载用

webpack.config.js

{

test:/\.js$/,

exclude:/node\_modules/,

loader:”babel-loader”,

options:{

// 预设: 只是babel做怎样的兼容性处理 下方是基本兼容性处理

// 只能做普通的兼容性处理 无法处理如promise等等

// 全部兼容性处理 使用 @babel/polyfill

// 这个包会将所有兼容处理引入,体积太大

// 所以我们需要做按需加载 corejs包

presets: [

// 注意此处的中括号

["@babel/preset-env",

{

// 按需加载

useBuiltIns: "usage",

// 指定corejs版本

corejs: {

version: 3

},

// 指定浏览器版本

targets: {

chrome: "60",

firefox: "60",

ie: "9",

safari: "10",

edge: "17"

}

}]

]

}

### js和html压缩

webpack.config.js

js压缩: 将mode改为production // 生产环境会自动压缩js代码

plugins: [

        new HtmlWebpackPlugin({

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

            minify: {

                // 移除空格

                collapseWhitespace: true,

                // 移除注释

                removeComments: true,

            },

        }),

    ]

### 性能优化

开发环境性能优化

优化webpack的打包构建速度

优化代码调试功能

生产环境性能优化

优化webpack打包构建速度

优化代码运行的性能

#### 优化开发环境打包构建速度

1. HMR

目前是一个模块发生变化所有模块都重新打包

HMR : hot module replacement 模块热替换

一个模块发生变化只会重新打包这一个模块,而不是打包所有,这样会帮我们极大提高构建速度

样式文件: 可以使用HMR功能: 因为style-loader内部实现了

js文件: 默认不能使用HMR功能

解决: 添加支持HMR功能的代码

if(module.hot){

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

    // 当print.js更新后执行代码

    print()

  })

}

注意: 只能处理非入口文件js

html文件: 默认不能使用HMR功能,同时会导致HTML文件不能热更新了(不需要HMR

功能)

解决: 修改entry入口 ,改为数组,将html文件引入

entry: ["./src/index.js", "./src/index.html"],

devServer: {

        // 端口号

        port: 8000,

        // 项目构建后的路径

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

        // 启用g-zip压缩

        compress: true,

        // 自动打开浏览器

        open: true,

        // 开启HMR功能

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

        hot: true

    }

#### source-map

一种提供源代码到构建后代码的映射关系的技术

直接在webpack最后添加

,devtool: "source-map"

/\*\*

 \* devtool的值有以下几种

 \* 注意: 以下中括号可以随意组合

 \* [inline- | hidden- | eval-][nosources-][cheap-[module-]]source-map

 \*

 \* source-map: 外部

 \*      错误代码的准确信息和原代码精确到错误行和列

 \*

 \* inline-source-map:内联  会在打包后的出口文件中生成一个内联source-map base64编码

 \*      1.只生成一个内联的source-map

 \*      错误代码的准确信息和原代码精确到错误行和列

 \*

 \* hidden-source-map:外部

 \*      错误代码的准确信息

 \*      但是没有错误位置(错误位置提示的是build后的位置),无法追踪到原代码的位置

 \*

 \* eval-source-map:内联

 \*      每一个文件都生成一个source-map并且都在eval函数

 \*      错误代码的准确信息和原代码精确到错误行和列(会多一个哈希值)

 \*

 \* nosources-source-map: 外部

 \*      错误代码的准确信息和原代码精确到错误行和列

 \*      但是无法看到原代码信息(隐藏原代码,安全性)

 \*

 \* cheap-source-map:外部

 \*      错误代码的准确信息和原代码精确到错误行(只精确到行而不是列)

 \*      cheap会提示一整行都报错了,但是其他的会提示更加准确的报错位置

 \*

 \* cheap-module-source-map:外部

 \*      错误代码的准确信息和原代码精确到错误行(只精确到行而不是列)

 \*      会将loader的source-map加入

 \*

 \* 内联和外部的区别:1.外部的生成了文件,内联没有 2.内联的构建速度更快

 \*

 \*

 \* 开发环境下  速度要快,调试要友好

 \*            速度排行 eval>inline>cheap

 \*            最快的是 eval-cheap-source-map

 \*            调试友好 source-map

 \*                    cheap-module-source-map

 \*                    cheap-source-map

 \*            --> eval-source-map / eval-cheap-module-source-map

 \* 生产环境下  原代码要不要隐藏,调试要达到那个程度

 \*            内联会让代码体积变得很大,所以内联的方式直接排除掉

 \*              hidden-source-map  只隐藏原代码会提示构建前代码错误信息

 \*              nosources-source-map  全部隐藏

 \*            --> source-map / cheap-module-source-map

 \*/

#### oneOf

loader只会匹配一个文件,不能同时存在两个处理一样的文件的loader

 rules: [

            {

                test: /\.js$/,

                exclude: /node\_modules/, // 不检查node\_modules目录下的文件

                loader: "eslint-loader",

                enforce: "pre",

                options: {

                    fix: true //开启自动修复

                }

            },

            {

                // 以下loader只会匹配一个

                // 注意 不能同时有两个配置处理同一类文件

                oneOf: [{

                    test: /\.js$/,

                    exclude: /node\_modules/,

                    loader: "babel-loader",

                    options: {

                        // 预设: 指示babel做怎样的兼容性处理  下方是基本兼容性处理

                        // 只能做普通的兼容性处理 无法处理如promise等等

                        // 全部兼容性处理 使用 @babel/polyfill

                        // 这个包会将所有兼容处理引入,体积太大

                        // 所以我们需要做按需加载 core-js包

                        presets: [

                            // 注意此处有个中括号

                            ["@babel/preset-env",

                                {

                                    // 按需加载

                                    useBuiltIns: "usage",

                                    // 指定corejs版本

                                    corejs: {

                                        version: 3

                                    },

                                    // 指定浏览器版本

                                    targets: {

                                        chrome: "60",

                                        firefox: "60",

                                        ie: "9",

                                        safari: "10",

                                        edge: "17"

                                    }

                                }]

                        ]

                    }

                },

                {

                    test: /\.css$/,

                    use: [

                        ...cssComm

                    ]

                },

                {

                    test: /\.scss$/,

                    use: [

                        ...cssComm,

                        "sass-loader"

                    ]

                },

                {

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

                    // 此处需要下载 file-loader  url-loader

                    use: [

                        {

                            loader: "url-loader",

                            options: {

                                limit: 8 \* 1024,

                                // 因为html-loader引入图片采用的是commonjs(require)  而 url-loader采用的是es6(import)

                                // 所以解析是会出现个问题  html中的<img src="[object Module]" alt="">

                                // 解决方法 : 关闭url-loader中的es6解析

                                esModule: false,

                                // hash:10  取图片hash值的前十位

                                // . 还是那个点

                                // [ext]  文件原后缀名

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

                            }

                        }

                    ]

                },

                // html-loader 处理html中的image文件,负责image文件的引入 从而能够被url-loader处理

                {

                    test: /.html$/,

                    use: "html-loader"

                },

                {

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

                    loader: "file-loader"

                }]

            }

        ]

#### 缓存

1.开启bable缓存

让第二次打包速度更快

{

                        test: /\.js$/,

                        exclude: /node\_modules/,

                        loader: "babel-loader",

                        options: {

                            // 预设: 指示babel做怎样的兼容性处理  下方是基本兼容性处理

                            // 只能做普通的兼容性处理 无法处理如promise等等

                            // 全部兼容性处理 使用 @babel/polyfill

                            // 这个包会将所有兼容处理引入,体积太大

                            // 所以我们需要做按需加载 core-js包

                            presets: [

                                // 注意此处有个中括号

                                ["@babel/preset-env",

                                    {

                                        // 按需加载

                                        useBuiltIns: "usage",

                                        // 指定corejs版本

                                        corejs: {

                                            version: 3

                                        },

                                        // 指定浏览器版本

                                        targets: {

                                            chrome: "60",

                                            firefox: "60",

                                            ie: "9",

                                            safari: "10",

                                            edge: "17"

                                        }

                                    }]

                            ],

                            // 开启bable缓存

                            // 第二次构建的时候会读取之前的缓存

                            cacheDirectory: true

                        }

                    }

1. 文件资源缓存

后台设置文件强制缓存 上线运行缓存更好使用

给打包文件设置hash值

问题: 只改动一个文件,所有的文件都会被重新请求,因为js和css同时使用一个hash值

 output: {

        filename: "js/build.[hash:10].js",

        path: resolve(\_\_dirname, "dist")

    },

plugins: [

        new HtmlWebpackPlugin({

            template: "./src/index.html"

        }),

        new MiniCssExtractPlugin({

            filename: "css/dist.[hash:10].css"

        }),

    ],

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

chunk : 根据入口文件创建一个hash 同一个入口文件下引入的文件属于一个chunk

contenthash : 根据文件的内容生成hash值,不同的文件的hash一定不一样

最终结论 使用contenthash

#### tree shaking

去除应用程序中没有使用的代码,让应用程序变得更小

在生产环境中自动启用

#### code split

代码分割

为了不让js文件成为一个巨大的js文件,为此拆分为多个单独的文件,从而实现并行加载,加载速度更快

下面是多页面应用的解决方案

1. 将入口 (entry)改为多入口{main:”./src/js/index.js”,test:”./src/js/test.js”}

出口(output)改为{filename:”js/[name].[contenthash:10].js”}

entry: {

        index: "./src/index.js",

        test: "./src/test.js"

    },

    output: {

        filename: "js/[name].[contenthash:10].js",

        path: resolve(\_\_dirname, "dist")

    },

1. 下方可以进行单入口也可以进行多入口,会将node\_modules里面的引用打包成单独文件

/\*

\* 可以将node\_modules中的代码单独打包成一个chunk输出

\* 自动分析多入口chunk中,有没有公共文件,如果有会打包成单独的chunk

    \* \*/

    optimization: {

        splitChunks: {

            chunks: "all"

        }

    },

1. 使用js语法打包为单独文件

/\*

\* 通过js代码将某个单独文件打包成一个chunk

\* import动态导入语法: 能将某个文件单独打包

\* \*/

import (/\* webpackChunkName:"test" \*/"./test")

    .then(res=>{

        console.log(res.add(2,2))

    })

    .catch(err=>console.log("错误",err))

#### PWA

渐进式网络应用程序(progressive web application - PWA)，是一种可以提供类似于native app(原生应用程序) 体验的 web app(网络应用程序)。(离线也能访问)

注意:

1.eslint不认识wendow navigator等全局变量

解决需要修改eslint配置(package.json中的eslintConfig配置):

“env”:{“browser”:true // 支持浏览器全局变量}

2.serviceWorker 必须运行在服务器上

#### 多进程打包

多进程打包需要使用 thread-loader

一般给babel-loader使用

多线程打包的缺陷: 进程启动大概为600mx 进程通信也有开销

只有工作小号时间比较长才需要多进程打包

注意 thread-loader 需要放在use数组中的某一个loader的后面(注意这里的放在后面是指数组下标小于前面的loader的数组下标)(因为webpack的loader的执行顺序为从数组下标最大的执行到数组下标最小的) 就会对前面的loader开启多线程打包

 {

                test: /\.js$/,

                exclude: /node\_modules/,

                use:[

                    // 如果项目比较小加入多线程打包甚至可能会变慢

                    {

                        loader: "thread-loader",

                        options: {

                            workers: 2 // 进程为2

                        }

                    },

                    {

                        loader: "babel-loader",

                        options: {

                            // 预设: 指示babel做怎样的兼容性处理  下方是基本兼容性处理

                            // 只能做普通的兼容性处理 无法处理如promise等等

                            // 全部兼容性处理 使用 @babel/polyfill

                            // 这个包会将所有兼容处理引入,体积太大

                            // 所以我们需要做按需加载 core-js包

                            presets: [

                                // 注意此处有个中括号

                                ["@babel/preset-env",

                                    {

                                        // 按需加载

                                        useBuiltIns:"usage",

                                        // 指定corejs版本

                                        corejs:{

                                            version:3

                                        },

                                        // 指定浏览器版本

                                        targets:{

                                            chrome:"60",

                                            firefox:"60",

                                            ie:"9",

                                            safari:"10",

                                            edge:"17"

                                        }

                                    }]

                            ]

                        }

                    }

                ],

            },

#### extrnals

当忽略某些包的打包就需要手动引入这个包

不打包某些库

    mode: "production",

    externals: {

        // 忽略的库名 - - npm包名

        jquery:"jquery"

    }

#### dll

动态链接库

类似于extrnals 指示某些库不进行打包,不同的是dll会单独进行打包 将多个库打包成一个chunk

做了哪些事情:

1. 通过webpack.dll.js打包输出jQuery和manifest.json(指示哪些库不需要打包)
2. webpack.config.js中 引入webpack.DllReferencePlugin和add-asset-html-webpack-plugin插件
3. 其中 webpack.DllReferencePlugin插件配置manifest属性并且指向manifest.json文件用于标识那些文件不需要打包
4. add-asset-html-webpack-plugin插件用于将dll打包好的文件原样输出到最后构建的dist目录下并且在HTML中引入输出后的文件

新建文件: webpack.dll.js (文件名可以不是这个)

/\*

\* 使用dll技术,对某些第三方库进行单独打包

\* 当去运行webpack时,默认查找的是webpack.config.js

\*

\* 所以运行的时候需要运行 webpack --config webpack.dll.js

\* \*/

var {resolve} = require("path")

var webpack = require("webpack")

module.exports = {

    // 下方两个配置是专门用来打包jQuery的 jQuery的名称是jQuery\_hash.js

    entry:{

        // 最终打包生成的name -> jquery

        // ["jQuery"]  要打包的库是 jQuery

        jquery:["jquery"]

    },

    output: {

        filename: "[name].js",

        path: resolve(\_\_dirname,"dll"),

        library: "[name]\_[hash:8]", // 打包的库向外暴露的名字是什么名字

    },

    plugins: [

        // 打包生成一个manifest.json文件 --> 提供和jQuery的映射关系

        new webpack.DllPlugin({

            name:"[name]\_[hash:8]", // 映射库的暴露的名称

            path: resolve(\_\_dirname,"dll/manifest.json") // 输出文件路径

        })

    ],

    mode: "production"

}

在webpack.config.js

add-asset-html-webpack-plugin 这个插件需要单独安装

var path = require("path")

var HtmlWebpackPlugin =require("html-webpack-plugin")

var webpack = require("webpack")

var AddAssetHtmlWebpackPlugin = require("add-asset-html-webpack-plugin")

module.exports = {

    entry: "./src/index.js",

    output: {

        filename: "index.js",

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

    },

    plugins: [

        new HtmlWebpackPlugin({

            template: "./src/index.html"

        }),

        // 告诉webpack 那些库不参与打包,同时使用时的名称也得改

        // 所以,不需要加extrnals

        new webpack.DllReferencePlugin({

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

        }),

        // 将某个文件打包输出出去,并且在html中自动引入该资源

        new AddAssetHtmlWebpackPlugin({

            filepath:path.resolve(\_\_dirname,"dll/jquery.js")

        })

    ],

    mode: "production"

}

### 总结:

开发环境性能优化:

打包构建速度:

HMR功能(模块热替换)

html文件不需要做热替换

style-loader本身支持热替换

js需要在代码中写需要哪个模块进行热替换

优化代码调试:

source-map:(代码映射,报错行等等)

--> eval-source-map / eval-cheap-module-source-map

生产环境性能优化:

打包构建速度:

1. oneOf
2. 缓存 (babel缓存[优化打包速度],)
3. 多进程打包(当代码体积变很大的时候可以考虑使用多进程打包)(一般作用于babel-loader)
4. extrnals(让某些库不被打包)(直接不打包,通过cdn引入)
5. dll(让某些库不被打包)(先打包一次库,打包成通过映射关系,引入到html中)

优化代码运行性能:

1. 持久化缓存[后台缓存,使用contenthash] hash (每次打包会生成一个唯一hash) chunkhash(同一个入口下会生成同一个hash), contenthash(根据文件内容生成hash)
2. tree shaking 树摇 会将无意义代码删除(必须开启es6模块化,mode:”production”)
3. code split(可以使用dll技术做细分)

单入口: 默认输出只有一个bundle,将一个大的js文件拆分成多个小的js文件,可以实现并行加载,提高加载速度

第一种方式: optimization: {  
 splitChunks: {  
 chunks: "all"  
 }  
}

第二种方式:

import()语法进行分割

多入口: 将入口改为对象,有几个入口就打包几个文件

需要加上optimization 否则会打包重复代码

1. 懒加载/预加载

懒加载使用时加载

预加载在加载完成后偷偷加载(兼容性较差)

1. pwa (离线访问)(兼容性较差)

### entry

可以用clean-webpack-plugin每次打包时清除上次打包的内容 引入插件无需配置,直接读取output的path

入口起点

1. string: 指向js文件作为入口,打包形成一个chunk输出一个bundle文件此时chunk的name默认为main
2. array: 指向
3. object

### output

output: {

        // 输出文件名称 可以使用 js/built.js 指定目录

        filename: "built.js"

        // 输出文件目录(将来所有资源输出的公共目录)

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

        // 所有引入的资源的公共路径的前缀 -> 输出路径的前面 -> 默认为 "img/a.png" ==> "/images/a.png"

        // 如果按照下方这么写的话 输出为  /asd/built.js

        , publicPath: "/asd"

        // 非入口chunk的名称

        /\*

        \* 通过import()分割出来的chunk

        \* 通过optimization分割出来的chunk

        \* \*/

        // 此处的name 为chunk的name  为 0,1,2.....

        , chunkFilename: "[name]\_chunk.js"

        // 将整个库内容暴露出去给外部使用

        , library: "[name]"

        // 适用于浏览器  意思是将这个文件模块挂载到window下

        // 如果是服务端(nodejs)  libraryTarget: "global"

        // 如果是libraryTarget: "commonjs" 是通过commonjs的规则暴露出去

        // 一般情况下是结合dll将某个库单独打包出去,正常打包不需要library

        , libraryTarget: "window"

    }

### module

module: {

        rules: [

            // loader的配置

            {

                // 正则匹配文件后缀

                test:/\.css$/,

                // 使用多个loader

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

            }

            ,{

                // 正则匹配文件后缀

                test:/\.js$/,

                // 排除node\_modules目录下的文件

                exclude:/node\_modules/,

                // 只检查src下的js文件

                include: path.resolve(\_\_dirname,"src"),

                // 如果不写 enforce规则的话则按照顺序执行

                // 延后执行

                // enforce: "post",

                // 优先执行

                enforce: "pre",

                // 使用单个loader

                loader: "eslint-loader"

            },

            {

                // 以下loader只会生效一个

                oneOf: [

                    // XXXX

                ]

            }

        ]

    }

### resolve

resolve: {

        // 配置解析模块路径别名: 帮助简写别名

        // 优点: 简写路径  缺点: 写路径时没有提示了

        alias: {

            $css:path.resolve(\_\_dirname,"src/css")

        },

        // 配置省略文件路径的后缀名

        // 默认为 [".js",".json"]

        // 就是为什么写import的时候可以省略 .js后缀名

        // 找文件的顺序按照数组下标走 如果同一文件夹下 既有index.js 又有index.css的话 import "index" 找到的必然是两次index.js

        extensions: [".js",".json",".css"],

        // 告诉webpack 解析模块时去找那个目录

        modules: [path.resolve(\_\_dirname,"../../node\_modules"),"node\_modules"]

    }

#### devServer

devServer: {

        // 端口号

        port:8000,

        // 运行代码的目录

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

        // 监视contentBase目录下的所有文件,一旦文件变化就会reload

        watchContentBase:true,

        watchOption:{

            // 忽略文件

            ignored:/node\_modules/

        }

        // 启用g-zip压缩

        ,compress:true,

        // 自动打开浏览器

        open:true,

        // 指定域名

        host:"localhost",

        // 开启HMR功能

        hot: true,

        // 不要显示启动服务器的日志信息

        clientLogLevel: "none",

        // 出了基本的启动信息以外,其他内容都不要打印

        quiet: true,

        // 如果出现错误不要全屏提示

        overlay:false,

        // 服务器代理 -> 解决开发环境下跨域问题

        proxy:{

            "/api":{

                target:"http://localhost:3000",

                pathRewrite:{

                    // 发送请求时路径重写  将/api/xxx -> /xxx

                    "^/api":""

                }

            }

        }

    }

### optimization

 optimization: {

        splitChunks: {

            chunks: "all",

            // 下方的设置都是默认值,一般不需要修改

            minSize: 30\*1024, // 分割的chunk最小为30KB

            maxSize:0, // 最大没有限制

            minChunks: 1, // 要提取的chunk最少被引用一次

            maxAsyncRequests: 5 , // 按需加载时并行加载的文件的最大数量为5

            maxInitialRequests: 3, // 入口js文件最大并行数量

            automaticNameDelimiter: "~", // 文件名称连接符

            name: true, // 可以使用命名规则

            cacheGroups: {  // 分割chunk组

                // node\_modules 文件会被打包到 vendors组的chunk中  vendors~xxx.js

                // 满足上面的公共规则 , 上面的规则对下方规则都生效

                vendors: {

                    test: /[\\/.]node\_modules[\\/]/,

                    // 优先级

                    priority: -10

                },

                default:{

                    // 要提取的chunk最少被引用两次

                    minChunks:2,

                    // 优先级比上面的低

                    priority: -20,

                    // 如果当前要打包的模块,和之前已经被提取的模块时同一个,就会复用,而不是重新打包模块

                    reuseExistingChunk: true

                }

            }

        },

        // 将当前模块的记录其他模块的hash单独打包为一个文件 runtime

        // 解决: 修改A文件导致B文件的contenthash变化 导致缓存失效

        runtimeChunk: {name: entrypoint =>{return `runtime-${entrypoint.name}`}}

        , minimizer: [

            //配置生产环境的压缩方案 : js , css

// 此处需要下载 terser-webpack-plugin

            new TerserWebpackPlugin({

                // 开启缓存

                cache:true,

                // 开启多进程打包

                parallel: true,

                sourceMap: true

            })

        ]

    }

## webpack5

webpack4该怎么用 webpack5还是怎么用

webpack5

此版本重点关注以下内容:

* 通过持久缓存提高构建性能.
* 使用更好的算法和默认值来改善长期缓存.
* 通过更好的树摇和代码生成来改善捆绑包大小.
* 清除处于怪异状态的内部结构，同时在 v4 中实现功能而不引入任何重大更改.
* 通过引入重大更改来为将来的功能做准备，以使我们能够尽可能长时间地使用 v5.

下载

* npm i webpack@next webpack-cli -D

自动删除 Node.js Polyfills

早期，webpack 的目标是允许在浏览器中运行大多数 node.js 模块，但是模块格局发生了变化，许多模块用途现在主要是为前端目的而编写的。webpack <= 4 附带了许多 node.js 核心模块的 polyfill，一旦模块使用任何核心模块（即 crypto 模块），这些模块就会自动应用。

尽管这使使用为 node.js 编写的模块变得容易，但它会将这些巨大的 polyfill 添加到包中。在许多情况下，这些 polyfill 是不必要的。

webpack 5 会自动停止填充这些核心模块，并专注于与前端兼容的模块。

迁移：

* 尽可能尝试使用与前端兼容的模块。
* 可以为 node.js 核心模块手动添加一个 polyfill。错误消息将提示如何实现该目标。

Chunk 和模块 ID

添加了用于长期缓存的新算法。在生产模式下默认情况下启用这些功能。

chunkIds: "deterministic", moduleIds: "deterministic"

Chunk ID

你可以不用使用 import(/\* webpackChunkName: "name" \*/ "module") 在开发环境来为 chunk 命名，生产环境还是有必要的

webpack 内部有 chunk 命名规则，不再是以 id(0, 1, 2)命名了

Tree Shaking

1. webpack 现在能够处理对嵌套模块的 tree shaking

*// inner.js*

export const a = 1;

export const b = 2;

*// module.js*

import \* as inner from './inner';

export { inner };

*// user.js*

import \* as module from './module';

console.log(module.inner.a);

在生产环境中, inner 模块暴露的 b 会被删除

1. webpack 现在能够多个模块之前的关系

import { something } from './something';

function usingSomething() {

return something;

}

export function test() {

return usingSomething();

}

当设置了"sideEffects": false时，一旦发现test方法没有使用，不但删除test，还会删除"./something"

1. webpack 现在能处理对 Commonjs 的 tree shaking

Output

webpack 4 默认只能输出 ES5 代码

webpack 5 开始新增一个属性 output.ecmaVersion, 可以生成 ES5 和 ES6 / ES2015 代码.

如：output.ecmaVersion: 2015

SplitChunk

*// webpack4*

minSize: 30000;

*// webpack5*

minSize: {

javascript: 30000,

style: 50000,

}

Caching

*// 配置缓存*

cache: {

*// 磁盘存储*

type: "filesystem",

buildDependencies: {

*// 当配置修改时，缓存失效*

config: [\_\_filename]

}

}

缓存将存储到 node\_modules/.cache/webpack

监视输出文件

之前 webpack 总是在第一次构建时输出全部文件，但是监视重新构建时会只更新修改的文件。

此次更新在第一次构建时会找到输出文件看是否有变化，从而决定要不要输出全部文件。

默认值

* entry: "./src/index.js
* output.path: path.resolve(\_\_dirname, "dist")
* output.filename: "[name].js"

更多内容

<https://github.com/webpack/changelog-v5>