# Webpack是什么

webpack是一种前端资源构建工具，一个静态模块打包器（module bundle）。在webpack看来，前端所有的资源文件（js/json/css/img/less...）都会作为模块处理，他将会根据模块的依赖关系进行静态分析，打包生成对应的静态资源（bundle）

# webpack五个核心概念

## Entry 入口

入口（entry）指webpack以哪个文件为入口起点开始打包，分析构建内部依赖图

## Output 输出

输出（output）指webpack打包后的资源bundles输出到哪里去，以及如何命名

## Loader (各种文件翻译成webpack可以处理文件)

*loader* 让 webpack 能够去处理那些非 JavaScript 文件（webpack 自身只理解 JavaScript）

## Plugins 插件

插件（plugins）可以用于执行范围更广的任务，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等

## Mode 模式

模式（mode)指webpack使用相应模式的配置

通过选择 development（开发模式） 或 production（生产模式） 之中的一个，来设置 mode 参数，你可以启用相应模式下的 webpack 内置的优化

|  |  |  |
| --- | --- | --- |
| 选项 | 描述 | 特点 |
| development | 会将process.env.NODE\_ENV的值设为development  启用NamedChunksPlugin和NamedModulesPlugin | 能让代码本地调试运行的环境 |
| production | 会将process.env.NODE\_ENV的值设为production  启用FlagDependencyUsagePlugin,  FlagincludeChunksPlugin,ModuleConcatenationPlugin,  NoEmitOnErrorsPlugin,OccurenceOrderPlugin,  SideEffectsFlagPlugin 和 UglifysPlugin | 能让代码优化上线运行的环境 |

# 3.配置webpack

cnpm install webpack webpack-cli -g 全局安装

cnpm install webpack webpack-cli -D 本地安装

# 4.基本使用

Webpack --mode development 输入文件.js -o 输出文件.js(名称随便)

（将输入文件可以实现的打包（复制）带输入文件上）

1.运行指令

开发环境 ：(webpac会以./src/index.js为入口文件开始打包，打包后输出到。/build/built.js整体打包环境，是开发环境)

webpack ./src/index.js -o ./build/built.js --mode=development

或者 webpack --mode development ./src/index.js -o /./build/built.js

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

2.结论

\* 2.1 webpack能处理js json资源文件 ，不能处理css img 等其他资源文件

\* 2.2 生产环境和开发环境将ES6模块化编译成浏览器能识别的模块化

\* 2.3 生产环境比开发环境多一个压缩js代码

# 5.打包资源

## 1）生成配置文件webpack.config.js

(作用：指示wenpack干哪些活，当运行webpack指令时，会加载里面的配置)

所有构建工具都是基于nodeis平台运行的模板化默认采用commonjs

const path = require('path')

module.exports = {

    //webpack配置

    //入口起点

    entry : './src/index.js',

    //输出

    output : {

        //输出文件名

        filename : 'built.js',

        path : path.resolve(\_\_dirname,'build') //resolve拼接路径，从后到前直到是绝对路径

    },

    //loader配置

    module : {

        rules :[

            //详细的loader配置

            //不同的文件必须配置不同的loader

            {

                //匹配以.css结尾的文件

                test: /\.css$/,

                //使用哪些loader进行处理 use中是从下向上执行

                use : [

                    //创建style标签， 将js中的样式资源插入进行添加到haed中

                    'style-loader',

                    //将css文件变更成Commonjs模块加载到js中，里面的内容为样式字符串

                    'css-loader'

                ]

            },

            {

                //匹配以.less结尾的文件

                test: /\.less$/,

                //使用哪些loader进行处理 use中是从下向上执行

                use : [

                    //创建style标签， 将js中的样式资源插入进行添加到haed中

                    'style-loader',

                    //将css文件变更成Commonjs模块加载到js中，里面的内容为样式字符串

                    'css-loader',

                    //将less文件变成css文件

                    'less-loader'

                ]

            }

        ]

    },

    //plugins插件的配置

    plugins : [

        //详细的plugins配置

    ],

    //模式

    mode : 'development' //开发环境

    // mode : 'production' //生产环境

}

## 2）src/index.js

//引入文件，才会打包到build/built.js中

import './index.css'

import './index.less'

# 6.开发环境基本配置

## 配置 css/less资源文件（loader）

1.下载loader：style-loader css-loader 2.引用

 module : {

        rules :[

            //详细的loader配置

            //不同的文件必须配置不同的loader

            {

                //匹配以.css结尾的文件

                test: /\.css$/,

                //使用哪些loader进行处理 use中是从下向上执行

                use : [

                    //创建style标签， 将js中的样式资源插入进行添加到haed中

                    'style-loader',

                    //将css文件变更成Commonjs模块加载到js中，里面的内容为样式字符串

                    'css-loader'

                ]

            },

            {

                //匹配以.less结尾的文件

                test: /\.less$/,

                //使用哪些loader进行处理 use中是从下向上执行

                use : [

                    //创建style标签， 将js中的样式资源插入进行添加到haed中

                    'style-loader',

                    //将css文件变更成Commonjs模块加载到js中，里面的内容为样式字符串

                    'css-loader',

                    //将less文件变成css文件

                    'less-loader'

                ]

            }

## 配置html资源文件(plugins插件)

1. 下载plugin：html-webpack-plugin 2.引入 3.引用

下载： cnpm install html-webpak-plugin

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

 plugins : [

        //插件

        //1.下载 2. 引入 3.使用

        //html-webpack-plugin

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

        // new html()

        //需要有结构的html文件

        new html({

            //复制./src/index.html文件，并自动引入打包输出所有资源（js/css）

            template : './src/index.html'

        })

    ],

## 配置img资源文件

1.下载loader：style-loader css.loader url-loader html-loader 2.使用

1. css中的背景图片

 rules : [

            {

                test :/\.css$/,

                use : [

                    'style-loader',

                    'css-loader'

                ]

            },

            {

                //默认处理不了html中的img图片

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

                //使用一个loader ,如果使用多个就要用use处理

                //下载url-loader file-loader

                loader : 'url-loader',

                options : {

                    //图片大小小于10kb，会被base64（编码字符串）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 （文件请求会更慢）

                    //8kb-12kb左右的图片才会进行base64处理

                    limit : 10 \* 1024, //8kb =8 \* 1024

                    /\*\*

                     \* (我的没有出现这个问题)

                     \* 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 [object Module]

                     \* 解决：关闭url-loader的ES6模块化，使用Commonjs解析

                     \* esModule : false

                     \*/

                    //给图片进行重命名

                    //[hash:10]取图片hash的前10位

                    //[ext]取文件原来的后缀名

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

                }

            },

1. Html中的img标签图片

{

                test : /\.html$/,

                //处理html文件的img图片（负责引入img，从而能被url-loader镜像处理）

                loader : 'html-loader'

            }

        ]

## 配置其他资源（ exclude排除）

1.下载loader：file-loader 2.使用

rules : [

            //打包其他资源（除了js/css/html文件以外）

            {

                //排除js/css/html资源文件

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

                loader : 'file-loader'

            }

        ]

## devServer开发服务器

1. 下载包：webpack-dev-server 2.使用

 //开发服务器devServer ：用来自动化（自动编译，自动打开浏览器，自动刷新浏览器，不用每次修改了代码就要重新打包）

    //特点：只会在内存中编译打包，不会有任何输出(不会生成build文件)

    //下载包 cnpm i webpack-dev-server -D

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

    devServer : {

        //运行项目的目录路径

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

        //启动gzip压缩

        compress : true,

        //端口号

        port : 8080,

        //自动打开

        open :true

    }

## 总结：

### loader翻译 :

### css ： style-loader css-loader

### less ：style-loader css-loader less-loader

### Img ：url-loader （是file-loader的升级版）

### Html中的img ：html-loader

### 其他资源 ：file-loader

### css提取单独文件 ： require(‘mini-css-extract-plugin’).loader

### Css兼容性 ：postcss-loader

### plugins插件：

### Html ：html-webpak-plugin

### Css提取 ： mini-css-extract-plugin

### css兼容性 : postcss-preset-env

### Css压缩：optimize-css-assets-webpack-plugin

## 开发环境基本配置

/\*\*

 \* 开发环境

 \* webpack命令会创建build文件

 \* webpack-dev-server 不会自动创建 自动编译，自动更新浏览器

 \*/

const path = require('path')

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

module.exports = {

    entry :'./src/js/index.js',

    output : {

        filename : 'js/built.js',

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

    },

    module : {

        rules : [

            {

                test : /\.css$/,

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

            },

            //打包img

            {

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

                loader : 'url-loader',

                options :{

                    limit : 10 \* 1024,

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

                    outputPath :'imgs'

                },

            },

            // 打包其他资源（除了js/css/html文件以外）

            {

                //排除js/css/html资源文件

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

                loader : 'file-loader',

                options :{

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

                    outputPath : 'edtma'

                },

            },

            // 打包html中的img

            {

                test : /\.html$/,

                loader : 'html-loader'

            }

        ]

    },

    plugins : [

        new html({

            template : './src/index.html'

        })

    ],

mode : 'development',

    //开发服务器devServer ：用来自动化（自动编译，自动打开浏览器，自动刷新浏览器，不用每次修改了代码就要重新打包）

    //特点：只会在内存中编译打包，不会有任何输出(不会生成build文件)

    //下载包 cnpm i webpack-dev-server -D

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

    devServer : {

        //运行项目的目录路径

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

        //启动gzip压缩

        compress : true,

        //端口号

        port : 8080,

        //自动打开

        open :true

    }

}

# 7.生产环境配置

## 配置将css从js中单独提取出来

1. 下载plugin：mini-css-extract-plugin 2.引入 3.在loader中引入 4.在plugins中使用

下载 ：cnpm install mini-css-extract-plugin -D

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

 rules : [

            {

                test : /\.css$/,

                use : [

                    //创建style标签，将样式插入

                    // 'style-loader',

                    //MiniCssExtractPlugin.loader取代了style-loader，将js中的css提取为单独文件

                    minicss.loader,

                    //将css编译为CommonJs整合到js文件中

                    'css-loader'

                ]

            }

]

plugins : [

        new html({

            //复制html的内容到build打包文件的html文件中

            template : './src/index.html'

        }),

        new minicss({

            //对输出的文件进行重命名

            filename : 'css/built.css'

        })

]

## 配置css兼容性

1. 下载loader：postcss-loader plugin：postcss-preset-env 2.使用

3.package.json配置

cnpm install postcss-loader postcss-preset-env -D

//改变nodeJs环境变量，来变为开发环境（默认为生产环境）

process.env.NODE\_ENV = 'development'

 rules : [

            {

                test : /\.css$/,

                use : [

                    minicss.loader,

                    'css-loader',

                    /\*\*

                     \* css兼容性 postcss  =>下载 postcss-loader

  postcss-preset-env

                     \*/

                    //使用loaderd的默认配置

                    //'postcss-loader'

                    //修改loader配置

                    {

                        loader :'postcss-loader',

                        options : {

                            ident : 'postcss',

                            plugins:()=>[

                                //postcss插件

                                //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

                                // "browserslist":{

                                //     开发环境  ==》设置node环境变量：postcss.env.NODE\_ENV='devlopment'

                                //     "development":[

                                //      兼容最近的浏览器版本

                                //       "last 1 chrome version",

                                //       "last 1 firefox version",

                                //       "last 1 safari version"

                                //     ],

                                //     生产环境 默认为生产环境

                                //     "prodution":[

                                //       ">0.2%", 大于99.8% 几乎所有浏览器

                                //       "not dead",不用兼容死的浏览器

                                //       "not op\_mini all" op\_mini所有浏览器

                                //     ]

                                //   }

                                require('postcss-preset-env')

                            ]

                        }

                    }

                ]

## Css压缩

1. 下载plugin： optimize-css-assets-webpack-plugin 2.引入 3.使用

Cnpm install optimize-css-assets-webpack-plugin -D

//压缩css

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

 plugins : [

        new html({

            //复制html的内容到build打包文件的html文件中

            template : './src/index.html'

        }),

        //提取css

        new minicss({

            filename : 'css/built.css'

        }),

        //压缩css

        new OptimizeCssAssetsWebpackPlugin()

    ],

## Js语法检查eslint

1. 下载loader :eslint eslint-loader eslint-config-airbnb-base

eslint-plugin-import 2.package.json配置airbnb-base语法规则 3使用

            /\*\*

             \* 语法检查 ：eslint-loader 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

             \*   "eslintConfig":{

                     "extends":"airbnb-base"

                 }

             \*  airbnb规则 --下载库-->eslint-config-airbnb-base  eslint  eslint-plugin-import

             \*/

            {

                test : /\.js$/, //只针对js进行检查

                exclude : /node\_modules/,//排除第三方

                loader: 'eslint-loader',

                options:{

                    //自动修复eslint的错误

                    fix:true

                }

            }

        ]

    }

## Js兼容性处理

1. 下载loader : babel-loader @babel/core @bable/preset-env

@babel/polyfill core-js 2.使用

 rules : [

                /\*\*

             \* js兼容性处理

             \*  "babel-loader": "^8.0.6",  //不能单独使用，要和@babel/core一起使用

             \*  "@babel/core": "^7.7.7",

                "@babel/preset-env": "^7.7.7",//基本js兼容性处理

                "@babel/preset-react": "^7.7.4",

                "@babel/polyfill": "^7.8.7",//全部js兼容性处理

                "core-js": "^3.6.4"  //按需要加载兼容处理

             1.基本js兼容性处理 ： @babel/preset-env

                    问题 ：只能转换基本语法 ，不能转换更高级的语法例如：promise回调地狱

             2.全部js兼容性处理 ：@babel/polyfill   在入口文件中引入import '@babel/polyfill'

                    问题：引入所有的兼容性问题 我只要解决部分兼容性问题 所有体积太大

             3.需要做兼容性处理就做，按需要加载 ：core-js

               1.按需下载useBuiltIns 2.指定core-js下载版本corejs 3.指定兼容性做的哪个版本浏览器targets

            \*/

           {

                test: /\.js$/,

                exclude : /node\_modules/,

                loader : 'babel-loader',

                options : {

                    //预设：指示babel做怎么样的兼容性处理

                    presets : [

                        [

                            '@babel/preset-env',//基本兼容性处理

                            {

                                //按需下载

                                useBuiltIns : 'usage',

                                //指定core-js下载版本

                                corejs:{

                                    version:3

                                },

                                //指定兼容性做的哪个版本浏览器

                                targets:{

                                    chrome :'60',

                                    firefox : '60',

                                    ie : '9',

                                    safari :'10',

                                    edge :'17'

                                }

                            }

                        ]

                    ]

                }

           }

        ]

## Js压缩

只要node: ’production’ 就可以压缩js文件

## Html文件压缩

 plugins : [

        new html({

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

            //压缩html

            minify :{

                //移除空格

                collapseWhitespace :true,

                //移除注释

                removeComments : true

            }

        })

    ],

## 生产环境完整配置

const path =require('path')

//html

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

//提取css

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

// 压缩css

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

//定义node环境变量，决定使用browserslist的哪个环境 默认是生产环境

process.env.NODE\_ENV = 'prodction'

//复用loader，下面直接调用

const CommCssloader  = [

    MiniCssExtractPlugin.loader,//在js中单独提取css

    'css-loader',//加载到js中

    {

         //兼容性处理

         //在package.json定义browserslist配置环境

        loader : 'postcss-loader',

        options : {

            ident :'postcss',

            plugins : ()=>[

                require('postcss-preset-env')

            ]

        }

    }

]

module.exports = {

    entry :'./src/js/index.js',

    output : {

        filename : 'js/built.js',

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

    },

    module : {

        rules : [

            //css

            {

                test : /\.css$/,

                use : [...CommCssloader]

            },

            //less

            {

                test : /\.less$/,

                use : [...CommCssloader,'less-loader']//将less编译为css

            },

            //正常一个文件只能被一个loader处理

            //当一个文件被多个loader处理，一定要指定loader执行的先后顺序

            //先执行 ：eslint 再执行 ：babel

            {

                //js语法检查eslint

               //在packsge.json定义eslintconfig :{extends:airbnb-bare}规则

                test: /\.js$/,

                exclude : /node\_modules/,

                enforce:'pre', //优先执行

                loader:'eslint-loader',

                options:{

                    fix:true//自动修改

                }

            },

            {

                //js兼容性

                //babel-loader @babel/core @babel/preset-env core-js

                test: /\.js$/,

                exclude : /node\_modules/,

                loader : 'babel-loader',

                options : {

                    presets : [

                        [

                            '@babel/preset-env', //基本语法兼容性处理

                            {

                                useBuiltIns : 'usage',//按需加载

                                corejs : {//版本

                                    version:3

                                },

                                targets :{//指示浏览器

                                    chrome:'60',

                                    ie:'9',

                                    firefox :'60'

                                }

                            }

                        ]

                    ]

                }

            },

            //图片

            {

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

                loader : 'url-loader',

                options : {

                    limit : 10 \* 1024, //10kb以下的被base64处理

                    name : '[hash:10].[ext]', //图片名只取前10位

                    outputPath : 'imgs'//输出再imgs文件下

                }

            },

            //html中的图片

            {

                test : /\.html$/,

                loader : 'html-loader'

            },

            //其他文件

            {

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

                loader :'file-loader',

                options : {

                    outputPath :'media' //输出再media文件中

                }

            }

        ]

},

    plugins : [

        //html

        new HtmlWebpackPlugin({

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

            minify : {//html压缩

                collapseWhitespace :true, //去空格

                removeComments :true //去注释

            }

        }),

        //提取css

        new MiniCssExtractPlugin({

            filename : 'css/index.css'

        }),

        //压缩css

        new OptimizeCssAssetsWebpackPlugin()

    ],

    mode : 'production',

    //devserver服务器

    devServer : {

        //运行项目的目录路径

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

        //启动gzip压缩

        compress : true,

        //端口号

        port : 8080,

        //自动打开

        open :true

    }

}

# 8.Webpack优化配置

## webpack性能优化

1.开发环境性能优化

1-1.优化打包构建速度

* HMR模块热替换（只打包修改了的文件）

1-2.优化代码调试

* source-map（可以提示源代码错误信息）

2.生产环境性能优化

2-1.优化打包构建速度

* oneOf： (只打包一个loader,找到了自己需要的就不打包其他的了，只会处理 一个 配置，相同的要单独提一个出去)是 个数组
* babel缓存： 在babel-loader中配置：cacheDirectory :true
* 多进程打包：thread-loader （一定要放在某个loader后面）
* externals拒绝被打包 ：externals : {jquery : 'jQuery'}

忽略库名 -->npm包名

拒绝jquery包被打包 ,要手动引入，提高打包速度

* dll将第三方库再进行打包：（先将库打包好，再告诉webpack这 个库不打包，然后自动引入该文件）

1.创建webpack.dll.js文件，在文件中配置 打包第三方库，并生成manifest.json文件形成映射关系

2.引入webpack插件 new webpack.DllReferencePlugin 告诉webpack哪些库不参与打包

3.下载add-asset-html-webpack-plugin插件（将某个文件打包输出去，并在html自动引入该文件（也可以手动引入））

2-2.优化代码运行的性能

* 文件资源缓存：

1. hash(webpack每次打包生成的hash值)
2. chunkhash（来自于同一个chunk，也就是同一个入口文件，共 享同一个hash值）,
3. contenthash（根据文件内容生成hash值）

* tree shacking：去除没有使用的代码 （必须时ES6模块 ，必须是生产环境production(生产环境中有个插件会自动生成tree shacking功能)，配置package.json中sideEffects设置排除去除的代码文件）
* code split代码分割：

1. 单入口 （1.optimization是将node\_modules中的代码单独打包成一个chunk最终输出2.import动态导入语法，能让这个导入的文件单独打包），
2. 多入口（optimization不会重复打包相同的代码，也可以提取node\_modules文件） 可以和dll技术形成更细化的打包

* 懒加载 ：当文件需要使用才加载

预加载 ：提前加载好了js文件，使用时加载缓存文件(等其他需要使用的资源加载完毕了，浏览器空闲了，再偷偷加载不用的资源), 配置webpackprefetch:true

* PWA离线访问 ：

1. workbox插件 ，
2. 注册serviceWorker ，
3. 配置json文件因为eslint不认识window和navigator全局变量，要修改，在eslintConfig后加"env" : {"browser":true}

引入的叫chunk，输出的叫bundle

## HMR模块热替换（只重新打包修改了的文件）

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

 \*  作用：一个模块发生变化，只会重新打包这一个模块（而不打包所有）

 \*  极大提升构建速度

 \*

 \* css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  --需要修改js代码，添加支持HMR功能的代码

 \*  注意：HMR功能对js处理，只能处理非入口js文件的其他文件

 \* html文件（不需要HMR功能） ：默认不能使用HMR功能，同时会导致html不能热更新了（热更新：重新更新，重新让浏览器刷新）

 \*   解决：修改entry入口：entry :['./src/js/index.js','./src/index.html'],

 \*/

在devServer配置中添加 hot :true

 devServer : {

        //运行项目的目录路径

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

        //启动gzip压缩

        compress : true,

        //端口号

        port : 8080,

        //自动打开

        open :true,

        //开启HMR功能（只重新打包修改的文件）

        //当修改了webpack配置文件，要重启服务

        hot:true

    }

在入口文件js中

if(module.hot){//判断module上有没有hot，一旦module.hot为true.说明开启了HMR功能，让HMR功能生效

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

        //方法会监听print.js文件的改变，一旦发生改变，其他模块不会重新打包构建，会执行后面的回调函数

        print()

    })

}

## Source-map

module.exports = {

devtool : 'eval-source-map' //适合开发环境的

}

/\*\*

 \* 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（build中的）出错了，通过映射可以追踪到源代码（src中的））

 \*

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

 \*

 \* 内联和外部区别 ：1.外部生成了文件，内联没有 2.内联构建速度快

 \* source-map                  外部 生成一个built.js.map文件

 \* 提示错误代码的准确信息和源代码的错误位置（可以精确到具体行和列）

 \* inline-source-map           内联source-map 在文件最后面

 \* 提示错误代码的准确信息和源代码的错误位置

 \* hidden-source-map           外部

 \* 提示错误代码的错误原因，但是没有错误位置，也不能追踪到源代码错误位置，只能提示构建后代码的错误位置（半隐藏源代码）

 \* eval-source-map             内联  每一个文件都生成对应的sourec-map，都在eval中

 \* 提示错误代码的准确信息和源代码的错误位置

 \* nosources-source-map        外部

 \* 提示错误代码的准确信息，但是没有任何源代码的错误信息（全隐藏源代码）

 \* cheap-source-map            外部

 \* 提示错误代码的信息和源代码的错误位置（只能精确到行，不能精确到列，打包速度会快一点）

 \* cheap-module-source-map    外部

 \* 提示错误代码的准确信息和源代码的错误位置

 \* module会将loader的source-map加入

 \*

 \* 开发环境：速度快，调试更友好

 \* 速度快   （eval > inline >cheap ...）

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

 \* 调试更友好 source-map > cheap-module-source-map >cheap-source-map

 \* eval-source-map或eval-cheap-module-source-map是比较好的

 \*

 \* 生成环境：源代码要不要隐藏，调试要不要更友好

 \* 内联会让代码体积变大，可以生成环境不用内联

 \* 隐藏源代码：nosource-source-map（全部隐藏）  hidden-source-map （隐藏源代码，提示构建后代码错误信息）

 \* 调试友好：source-map > cheap-module-source-map >cheap-source-map > nosource-source-map 或 hidden-source-map

 \* source-map /cheap-module-source-map 是比较好的

 \*

 \*

 \*/

## oneOf 只配置一个文件

rules下loader只会匹配一个 ，不会每一个loader都会重新执行一次，

 \*  提高性能，提升构建速度

    注意：不能有两个配置同属处理一种类型 （所以把eslint-loader提取出了）

 rules : [

            //正常一个文件只能被一个loader处理

            //当一个文件被多个loader处理，一定要指定loader执行的先后顺序

            //先执行 ：eslint 再执行 ：babel

            {

                //js语法检查eslint

               //在packsge.json定义eslintconfig :{extends:airbnb-bare}规则

                test: /\.js$/,

                exclude : /node\_modules/,

                enforce:'pre', //优先执行

                loader:'eslint-loader',

                options:{

                    fix:true//自动修改

                }

            },

            {

                //以下loader只会匹配一个 ，不会每一个loader都会重新执行一次，提高性能，提升构建速度

                //注意：不能有两个配置同属处理一种类型 （所有把eslint-loader提取出了）

                oneOf : [

                   //css

                    {

                        test : /\.css$/,

                        use : [

MiniCssExtractPlugin.loader,//在js中单独提取css

     'css-loader',//加载到js中

     {

           //兼容性处理

          //在package.json定义browserslist配置环境

         loader : 'postcss-loader',

         options : {

             ident :'postcss',

             plugins : ()=>[

                 require('postcss-preset-env')

             ]

          }

     }

]

                    },

                    //js

                    {

                        //js兼容性

                        //babel-loader @babel/core @babel/preset-env core-js

                        test: /\.js$/,

                        exclude : /node\_modules/,

                        loader : 'babel-loader',

                        options : {

                            presets : [

                                [

                                    '@babel/preset-env', //基本语法兼容性处理

                                    {

                                        useBuiltIns : 'usage',//按需加载

                                        corejs : {//版本

                                            version:3

                                        },

                                        targets :{//指示浏览器

                                            chrome:'60',

                                            ie:'9',

                                            firefox :'60'

                                        }

                                    }

                                ]

                            ]

                        }

                    }

]

## babel缓存和文件缓存

缓存：（配置node服务是要写开发资源的时间maxAge）

### babel缓存

（当100个js文件，只修改其中一个，就只会重新构建这一个，没有修改的就读取缓存）

 第二次构建时，会读取之前缓存  速度更快

cacheDirectory :true

{

                        //js兼容性

                        //babel-loader @babel/core @babel/preset-env core-js

                        test: /\.js$/,

                        exclude : /node\_modules/,

                        loader : 'babel-loader',

                        options : {

                            presets : [

                               ..........

                            ],

                            //开启babel缓存（当100个js文件，只修改其中一个，就只会重新构建这一个，没有修改的就读取缓存）

                            //第二次构建时，会读取之前缓存

                            //速度更快

                            cacheDirectory :true

                        }

                    },

### 文件资源缓存

css/index.[contenthash:10].css   js/built.[contenthash:10].js

     2-1.hash: 修改文件名（加上hash值，每次webpack构建会自动生成一个唯一的hash值）

    问题：因为css和js使用的是同一个hash值，

            如果重新打包，会导致所有是缓存失效（可是我只修改了一个文件）

    2-2.chunkhash: 根据chunk生成的hash值，如果打包来源于同一个chunk，那么他们的hash值就是一样是

    问题：因为css和js都是同一个入口文件引入的，hash值还是同一个chunk（入口文件和其引入的css,js依赖文件会形成一个文件，这个文件就是chunk代码块），

    2-3.contenthash: 根据文件的内容生成hash值，不同的文件hash值不一样

        -->让代码上线运行缓存更好使用

\*/

## Tree shacking 去除没有使用的代码

/\*\*

 \* tree shaking :去除没有用的代码

 \*  前提：1.必须使用ES6模块化 2.开启production环境

 \*  作用：减少代码体积

 \*  在package.json配置 "sideEffects" : false  所有代码都没有副作用（都可以进行tree shaking去除无用代码）

 \*  问题：可能会把css / @babel/polyfill （副作用）文件去除

 \*        "sideEffects" : ["\*.css","\*.less"]就不会去除css.less

 \*/

Package.json配置

"sideEffects": [

    "\*.css"

  ]

## Code split 代码分割

### 1.多入口

//单入口文件

    // entry :'./src/js/index.js',

    //多入口文件：有几个入口，就会输出几个bundle

    entry :{

        main : './src/js/index.js',

        test : './src/js/test.js'

    },

### optimization将node\_modules单独打包

//chunk ： 打包后的代码块

    //可以将node\_modules中的代码单独打包成一个chunk最终输出

    //自动分析多入口chunk中，有没有公共的文件（文件要比较大），如果有就会打包成单独一个chunk

    optimization : {

        splitChunks : {

            chunks : 'all'

        }

    },

### import动态导入语法

在入口js文件中

/\*\*

 \* 通过js代码，让某个文件能被单独打包成一个chunk

 \* import动态导入语法：能让这个导入的文件单独打包

 \* webpackChunkName ： 设置chunk的名字

 \*/

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

.then((result)=> {

    //文件加载成功

    //eslint-disble-next-line

    console.log(result)

    console.log(result.cheng(3,4))

})

.catch(()=>{

    //eslint-disable-next-line

    console.log('文件加载失败。。。')

})

## 懒加载，预加载

  懒加载：当文件需要使用才加载

 加上webpackPrefetch : true就是预加载

 预加载 prefetch : 提前加载好了js文件，使用时加载缓存文件

 （等其他需要使用的资源加载完毕了，浏览器空闲了，再偷偷加载不用的资源）

 正常加载 ： 是并行加载 （同一时间加载多个文件）

 document.getElementById('btn').onclick =function(){

    import(/\* webpackChunkName:'test', webpackPrefetch : true\*/'./test').then((result)=> {

        //文件加载成功

        //eslint-disble-next-line

        console.log(result)

        console.log(result.cheng(3,4))

    })

}

## PWA 离线加载

/\*\*

 \* PWA:渐进式网络开发应用程序（离线也可以访问）

 \*  1.workbox -->下载workbox-webpack-plugin插件

 \*  2.在入口文件中注册serviceWorker

 \*    因为eslint不认识window ,navagator全局变量，所有在package.json中配置

 \*    "eslintConfig" : {

 \*      extends : "airbnb-base",

 \*      "env" : {

 \*          "brower" : true //支持全局变量

 \*      }

 \*    }

 \*    要在服务器上运行才能注册 -->nodejs

 \*/

### 使用workbox-webpack-plugin插件

//离线访问

        new workbox.GenerateSW({

            //帮助serviceworker快速启动

            //删除旧的serviceworker

            //生成一个service-worker.js配置文件

            clientsClaim:true,

            skipWaiting :true

        })

### package.json配置

/\*\*

 \* 1.package.json配置

    eslint不认识window,navigator全局变量

    解决：需要修改package.json配置

    "eslintConfig": {

    "extends": "airbnb-base",

        "env":{

        "browser":true //支持浏览器端全局变量

        }

    }

  2.serviceWorker代码必须运行在服务器上 -->nodejs

 \*/

### 注册serviceworker配置

// 处理兼容性问题

/\*\*

 \* addEventListener 为元素添加事件

 \* 判断serviceworker是否再navigator上

 \* 如果在就支持，就绑定load事件（页面中的图片或其它东西加载完成之后，执行该函数） ,注册serviceworker

 \* \*/

if ('serviceWorker' in navigator) {

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

    // 注册文件/service-worker.js，由workbox插件来生成文件

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

      .then(() => {

        // eslint-disable-next-line

        console.log('注册成功');

      })

      .catch(() => {

        // eslint-disable-next-line

        console.log('注册失败');

      });

  });

}

## 多进程打包

/\*\*

 \* 多进程打包 提高打包速度

 \* 1.下载 ：thread-loader

 \* 2.使用 ：一定要放在某个loader后面（loader执行从下到上），

 \*         就会对这个loader进行多进程打包（一般用于代码多的js文件）

 \* //开启多进程打包

   //进程启动大概为600ms,进程通信也需要时间

   //只有工作量大的时候消耗时间长，才需要多进程打包

 \*

 \*/

 {

                        //babel-loader @babel/core @babel/preset-env core-js

                        test: /\.js$/,

                        exclude : /node\_modules/,

                        use : [

                            //开启多进程打包

                            //进程启动大概为600ms,进程通信也需要时间

                            //只有工作量大的时候消耗时间长，才需要多进程打包

                            {

                                loader : 'thread-loader',

                                options : {

                                    workers :2 //2个进程

                                }

                            },

                            //js兼容性

                            {

                                loader : 'babel-loader',

                                options : {

                                    presets : [

                                        ......

                                    ],

                                    //开启babel缓存（当100个js文件，只修改其中一个，就只会重新构建这一个，没有修改的就读取缓存）

                                    //第二次构建时，会读取之前缓存

                                    //速度更快

                                    cacheDirectory :true

                                }

                            }

## externals拒绝库或包被打包

module.exports= {

......

externals : {

        //拒绝jquery包被打包 ,要手动引入，提高打包速度

        //忽略库名 -->npm包名

        jquery : 'jQuery'

}

}

## dll将第三方库再进行单独打包

/\*\*

 \* dll 将第三方库再进行单独打包 不必重复打包

 \* 1.创建webpack.dll.js文件，在文件中配置 打包第三方库，并生成manifest.json文件形成映射关系

 \* 2.引入webpack插件 new webpack.DllReferencePlugin 告诉webpack哪些库不参与打包

 \* 3.下载add-asset-html-webpack-plugin插件（将某个文件打包输出去，并在html自动引入该文件（也可以手动引入））

 \*/

### 创建webpack.dll.js文件

module.exports = {

    entry : {

        //最终打包生成的[name] -->jquery

        //['jquery'] -->要打包的库是jquery (可写多个和jquery有关的)

        jquerys : ['jquery']

    },

    output : {

        filename : '[name].js',

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

        //打包的库开放出去的内容叫什么名字(在打包后jquery文件中接收的变量名称)，

        //等同于app.use('/public/',express.static('./public/'))里面的/public/

        library : '[name]\_[hash]'

    },

    plugins :[

        //打包生成一个manifest.json -->提供和jquery映射

        new webpack.DllPlugin({

            name : '[name]\_[hash]', //打包映射库开放出去内容的名称

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

        })

    ],

    mode :'production'

}

### 引入webpack插件 告诉webpack哪些库不参与打包

        // 告诉webpack哪些库不参与打包（忽略打包），同时使用时的名称也要改变

        //就去找..dll/manifest.json文件，因为这个文件和jquery有映射关系，所以jquery不参与打包

        new webpack.DllReferencePlugin({

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

        }),

### 3.下载add-asset-html-webpack-plugin插件（将某个文件打包输出去，并在html自动引入该文件（也可以手动引入））

//将某个文件打包输出去，并在html自动引入该文件 （也可以手动引入）

        //<script src="jquery.js"></script>

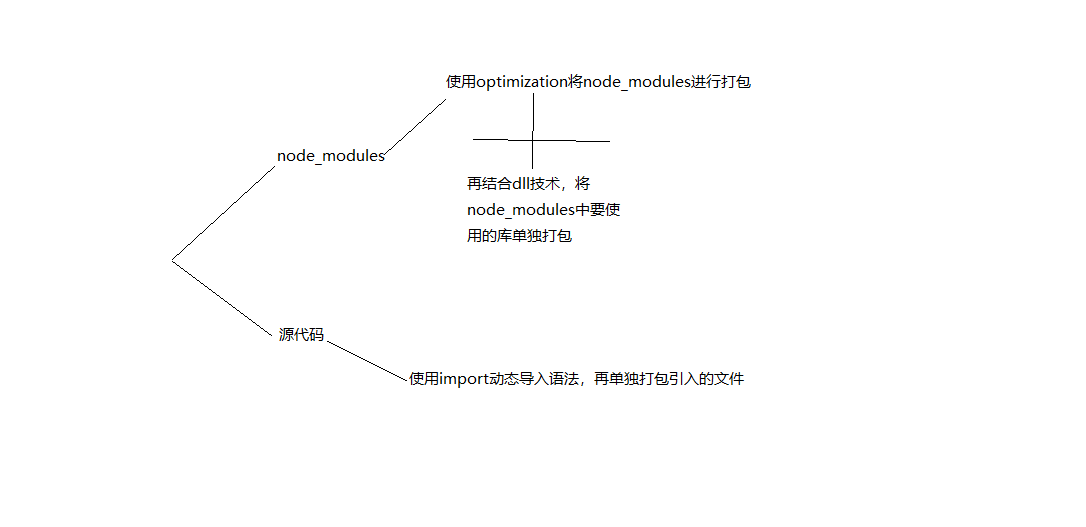
        //不参与打包就相当于不存在（将库引入），AddAssetHtmlWebpackPlugin就是将文件打包输出的

        new AddAssetHtmlWebpackPlugin({

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

        })

## code split和dll结合打包



# 9.Webpack配置详解

## entry入口

/\*\*

 \* entry 入口

 \*  1.string -->'./src/js/index.js'

 \*     单入口

 \*     打包形成一个chunk，输出一个bundle文件

 \*      chunk默认名称：main

 \*  2.array -->['./src/js/index.js','./src/js/add.js']

 \*      多入口  多个文件打包再一起

 \*      所有文件最终会形成一个chunk，输出只有一个bundle文件

 \*      -->只有再HMR功能中让html热更新生效

 \*  3.object -->{

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

                     add :'./src/js/add.js'

                }

        多入口

        有几个入口文件就有几个chunk，输出几个bundle文件

        chunk名称为key值（index,add）

        特殊用法： entry:{

                    //所有文件最终会形成一个chunk，输出只有一个bundle文件

                     index : ['./src/js/index.js','.src/js/const.js'],

                     //形成一个chunk，输出一个bundle文件

                     add :'./src/js/add.js'

                  }

 \*/

### String

 entry : './src/js/index.js',

### Array

 entry :['./src/js/index.js','./src/js/add.js'],

### 3.object

entry:{

        index : ['./src/js/index.js','./src/js/content.js'],

        add :'./src/js/add.js'

}

## output输出

output : {

        //输出打包文件的名字

        filename : 'js/[name].js',

        //打包文件的路径

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

        //所有资源引入的公共路径前缀 --> 'imgs/a.png' ->'/imgs/a.png'

        publicPath : '/',

        //非入口文件的chunk名称

        chunkFilename : 'js/[name]\_chunk.js',

        //在打包js文件接收变量名字 var main =function...

        library : '[name]',

        //变量名添加到哪个上面

        libraryTarget : 'window'  //window["main"] =function...

        // libraryTarget : 'global',

        // libraryTarget : 'commonjs'   //exports["main"] =...

    },

## module

 module : {

        rules : [

  {

                test : /\.js$/,

                //排除node\_modules下的js文件

                exclude : /node\_modules/,

                //只检查src下的js文件

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

                //优先执行

                enforce :'pre',

                //延后执行

                // enforce :'post',

                loader :'eslint-loader',

                options : {

                    fix :true

                }

            }

]

}

## resolve解析模板的规则

//resolve 解析模块的规则

    resolve :{

        //配置解析模块路径别名,简写路径

        alias:{

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

        },

        //配置省略文件路径后缀名

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

        //告诉webpack解析模块是去找哪个目录

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

    }

## devServer配置

//开发服务器devServer ：用来自动化（自动编译，自动打开浏览器，自动刷新浏览器，不用每次修改了代码就要重新打包）

    //特点：只会在内存中编译打包，不会有任何输出(不会生成build文件)

    //下载包 cnpm i webpack-dev-server -D

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

    devServer : {

        //运行项目的目录路径

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

        //监视contentBase目录下所有文件，一旦文件变化就会reload重载，重新打包，重新刷新浏览器

        watchContentBase:true,

        watchOptions : {

            //忽略文件

            ignored : /node\_modules/

        },

        //启动gzip压缩

        compress : true,

        //端口号

        port : 8080,

        //域名

        host : 'localhost',

        //自动打开

        open :true,

        //开启HMR模块热替换功能

        hot : true,

        //不要显示启动服务器日志信息

        ClientLogLevel : 'none',

        //除了一些基本启动信息以外，其他内容都不要显示

        quiet : true,

        //如果出错了，不要全屏提示

        overlay : false,

        //服务器代理 --> 解决开发环境跨域问题

        proxy : {

            //一旦devserver（8080）服务器接收到api/xxxx的请求，就会把请求转发到另外一个服务器（3000）

            'api':{

                target : 'http://localhost:3000',

                //发送请求时，请求路径重写：将/api/xxx --> /xxx (会去掉/api)

                pathRewrite : {

                    '^/api':''

                }

            }

        }

    }

## optimization单独打包nodu\_modules文件详细配置

//下载插件

//terser压缩js和css文件

const terserWebpckPlugin = require('terser-webpack-plugin')

代码分割 ； splitChunks

单独打包引入js文件的hash值：runtimeChunk

压缩方案 js和css：minimizer、

//单独打包node\_modules文件

    optimization:{

        //代码分割

### **splitChunks:**

{

            chunks : 'all'

            //下面都是默认值，可以不用写

            // minSize : 30 \* 1024 , //分割的chunk最小为30kb,小于30kb的就不分割

            // maxSize : 0 ,//最大没有限制

            // minChunks : 1 , //要提取的chunk最少要被引用1次

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

            // maxInitialRequests : 3  ,//入口js文件最大并行请求数量

            // automaticNameDelimiter : '~' ,//名称连接符

            // name :true , //可以使用命名规则

            // cacheGroups : { //分割chunk的组

            //     //node\_modules文件会被打包到vendors组的chunk中 ，打包后名称为vendors~xxxx.js

            //     //也要满足上面是需求 ：大小超过30kb，至少被引用一次

            //     vendors : {

            //         test : /[\\/]node\_modules[\\/]]/,//检测node\_modules下的文件

            //         //打包的优先级为-10

            //         priority : -10

            //     },

            //     default : {

            //         minChunks : 2 , //要提取的chunk最少要被引用2次

            //         priority : -20 , //优先级为-20

            //         reuseExistingChunk : true  //如果当前要打包的模块和之前已经被提取的打包模块是一样的，就会复用，而不是重新打包模块

            //     }

            // }

        },

        //单独打包当前文件中引入js文件的hash值 ,文件名：runtime

        //原因：因为当前js文件中保存着引入文件的hash值，

        //      当引入的文件发生改变，入口文件也会随之改变

### **runtimeChunk ：**

{

            // entrys.name ：当前模块文件的name  入口文件为main

            //function entrys(){return runtime-${entrys.name}}

            name : entrys =>`runtime-${entrys.name}`

        },

        //配置生产环境的压缩方案 js和css文件   terser库压缩 下载terserWebpackPlugin

### **minimizer :**

        [

            new terserWebpckPlugin({

                cache : true ,//开启缓存

                parallel : true , //开启多进程打包

                sourceMap :true //开启source-map ,不开启会被压缩

            })

        ]

    }

# Webpack5

1. 使用持久化缓存提高构建性能；
2. 使用更好的算法和默认值改进长期缓存（long-term caching）；
3. 通过更好的树摇tree shacking和代码生成来改善捆绑包大小；
4. 清理内部结构而不引入任何破坏性的变化；
5. 引入一些breaking changes，以便尽可能长的使用v5版本。

## 下载

npm install webpack@next -D

## 自动删除nodejs polyfill

使用nodejs要自动引入一个polyfill

## Chunk 和模块ID

在webpack4之前要修改chunk的id

import(/\* webpackChunkName:’test’\*/ ,’./test’)要手动修改，但是在webpack5中内部有chunk命名规则，不在是以id命名了

## Tree shacking

import {someting} form ‘./someting’

export function test(){}

* 更加精准的删除多个模块中没有用的代码
* 在package.json配置文件中设置sideEffects:false时，一旦发现方法test中没有使用，不但删除test，还会删除’/something’
* 现在可以处理CommonJs中的tree shacking

## output输出

Webpack4默认只输出ES5代码

Webpack5新增了一个属性outpute.cmaVersion,可以生成ES5和ES6/ES2015代码 output.ecmaVersion:2015

## splitChunk

跟价精确设置大小

MiniSize : {

Javascript :3000, //30kb

Style:5000 //50kb

}

## Caching 缓存

配置缓存

Cache ： {

//磁盘存储

Type : ‘filesystem’,

buildDependencies:{

//当配置修改时，缓存失败

Config : [\_\_filename]

}

}

## 监视输出的文件

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只会更新修改的文件

而webpack5在第一次构建时会找到输出文件是否有改变，从而决定要不要输出全部文件

## 默认值

Entry : ‘./src/index.js’,

Output:{

Filename : ‘[name].js’,

Path : path.resolve(\_\_dirname,’dist’)

}

如果是默认值就可以省略不用写

## 官方文档地址

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