**1.开发环境**

**整体代码:**

/\*  开发环境配置：

1.最终效果:只要代码能运行即可。

2.运行项目指令：方式1: webpack 会将打包结果输出出去

                方式2:npx webpack-dev-server 只会在内存中编译打包，没有输出

3.开发环境中共配置了如下资源:

(1)样式资源(css，css预处理)

(2)图片

(3)html

(4)其他资源

(5)开发服务器。

\*/

// 引入path核心模块

const { resolve } = require("path");

// 作用:html打包

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  // 入口

  entry: "./src/js/index.js",

  // 出口

  output: {

    filename: "js/built.js",

    // 所有的出口路径文件都是以build为根目录的。【★★★★★★】

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

  },

  // 模块

  module: {

    // 不同文件类型的Loader配置

    rules: [

      {

        // 1.打包less资源

        test: /\.less$/,

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

      },

      {

        // 2.打包css资源

        test: /\.css$/,

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

      },

      {

        // 3.打包图片资源

        test: /\.(jpg|png|gif)$/,

        loader: "url-loader",

        options: {

          limit: 8 \* 1024,

          name: "[hash:10].[ext]",

          // 关闭es6模块化

          esModule: false,

          outputPath: "imgs" //图片打包后自动存储在build/imgs目录下

        }

      },

      {

        // 3-2.处理html中img资源

        test: /\.html$/,

        loader: "html-loader"

      },

      {

        // 4.处理其他资源

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

        loader: "file-loader",

        options: {

          name: "[hash:10].[ext]",

          outputPath: "media"

        }

      }

    ]

  },

  plugins: [

    // 5.打包html资源

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    })

  ],

  // 开发模式

  mode: "development",

  // 配置开发服务器

  devServer: {

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

    compress: true,

    port: 3000,

    open: true

  }

};

**1-1.打包样式资源**

/\*

♣该配置文件目的:webpack打包样式资源，例如css/less/sass/stylus等。

♣重点:【★★★★★★★★】

  1.webpack.config.js是webpack的配置文件，当在03.打包样式资源这个跟目录下运行:webpack命令，则会按照如下打包进行。

  2.所有构建工具都是基于nodejs平台运行的~模块化默认采用commonjs。【★★★★★★】

  3.所有下载的依赖包要都下载到最外层目录，这样当某一个子目录中需要该包时，可以直接在最外层根目录

  下的node\_module中查找。

  4.打包css样式资源的步骤:

  (1)搭建webpack.config.js配置文件基本项结构

  (2)在最外层目录下下载css-loader style-loader;

  (3)在module>rules数组中创建其对应的loader对象，并指定test匹配规则；在Use中配置需要的loader包；

  (4)包的执行顺序为先下后上。先调用"css-loader", 将css文件变成commonjs模块加载js中，里面内容是样式字符串；再调用style-loader，当打开浏览器时，会自动创建style标签，将js中的样式字符串资源插入进行，添加到head中使得样式生效

\*/

// resolve用来拼接绝对路径的方法

const { resolve } = require("path");

module.exports = {

  // webpack配置

  // 入口起点【注解:打包的入口文件的相对路径。】

  entry: "./src/index.js",

  // 输出【是一个对象】

  output: {

    // 输出文件名

    filename: "built.js",

    // 输出路径【★★★★★★】

    /\*notes:path中的resolve中共有两个变量:

    (1)\_\_dirname是nodejs中的变量，代表当前文件(webpack.config.js)的目录绝对路径,这里的dirname的值就是‘03.打包样式资源’;

    (2)第二个变量'build'表示当前绝对目录下的build文件，然后filname表示文件中的文件名为built.js。

    (3)在path之前应该引入node中的path模块。

    \*/

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

  },

  // loader的配置

  module: {

    rules: [

      /\*  每一种类型的配置都是一个独立的对象。【★★★★★★】

      use数组中loader执行顺序：从右到左，从下到上 依次执行【★★★★★★】\*/

      // 配置css的Loader【★★★★★★★★】

      {

        // 匹配哪些文件，以.css结尾

        test: /\.css$/,

        // 使用哪些loader进行处理

        use: [

          // use数组中loader执行顺序：从右到左，从下到上 依次执行【★★★★★★】

          // 当打开浏览器时，会自动创建style标签，将js中的样式字符串资源插入进行，添加到head中使得样式生效

          "style-loader",

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

          "css-loader"

        ]

      },

      // // 配置less的Loader【★★★★★★★★】

      {

        test: /\.less$/,

        use: [

          "style-loader",

          "css-loader",

          // less-loader作用:将less文件编译成css文件

          // notes:如果想使用后less-loader则需要下载 less-loader和less【★★★★★★】

          "less-loader"

        ]

      }

    ]

  },

  // plugins的配置

  plugins: [

    // 详细plugins的配置

  ],

  // 模式

  mode: "development" // 开发模式

  // mode: 'production'

};

**1-2.打包html资源**

/\*

该配置文件目的:打包html资源

重点:

  1.loader和plugins使用步骤:【★★★★★】

  (1)loader: 1. 先下载   2. 直接在配置文件中使用（配置loader）

  (2)plugins: 1. 先下载  2. 再引入  3. 最后再在plugin插件配置想中使用New调用

  2.打包html资源的步骤:

  (1)搭建基本的webpack.config.js文件，定义基本配置项；

  (2)在最外层目录下载'html-webpack-plugin'插件；

  (3)在配置文件中引入插件:const HtmlWebpackPlugin = require('html-webpack-plugin');

  (4)在插件配置项中调用插件new HtmlWebpackPlugin({})

  (5)在实例对象中使用template属性，定义Html结构，并把打包后的js.css等代码自动存入其中。

  3.html-webpack-plugin插件的功能:

    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）,但是没有基本的html结构，所以需要借助实例对象中的tempalte指定Html结构。

\*/

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  entry: "./src/index.js",

  output: {

    filename: "built.js",

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

  },

  module: {

    rules: [

      // loader的配置

    ]

  },

  plugins: [

    // plugins的配置

    // html-webpack-plugin

    // 该插件的功能：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）【★★★★★★】

    // 需求：需要有结构的HTML文件

    new HtmlWebpackPlugin({

      // template作用:复制 './src/index.html' 文件，并自动引入打包输出的所有资源（JS/CSS）

      template: "./src/index.html"

    })

  ],

  mode: "development"

};

**1-3.打包图片资源资源**

/\*

目的:打包图片资源配置

1.打包图片资源的步骤:

(1)配置webpack基础项结构，把样式打包，Html打包先配置完成；按照其各自的配置规则；

(2)在最外层目录下载url-loader,file-loader包;

(3)配置打包图片的loader，其中包括:test,loader,options3大块;

(4)在最外层目录下载html-loader，用来处理html文件的img图片（负责引入img，从而能被url-loader进行处理）;

2.配置图片常见问题:

(1).如果html文件中有img引入的图片，如何实现打包?

答:url-loader默认的解析方式为es6;但是html-loader中的解析的方式为commonjs,所以可以关闭url-loader中的es6解析，从而实现正确解析html中的图片。

3.在打包某种资源时，有的需要多个loader，有的需要一个loader；写法如下:

(1)一个loader  loader: "html-loader"

(2)多个Loader  use: ["style-loader", "css-loader", "less-loader"]

\*/

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  entry: "./src/index.js",

  output: {

    filename: "built.js",

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

  },

  module: {

    rules: [

      // 打包less资源【★★★★★】

      {

        test: /\.less$/,

        // 要使用多个loader处理用use

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

      },

      // 打包图片资源

      {

        // 问题：默认处理不了html中img图片

        // 打包图片资源【★★★★★★】

        test: /\.(jpg|png|gif)$/,

        // 使用一个loader

        // 下载 url-loader file-loader

        loader: "url-loader",

        options: {

          // 图片大小小于8kb，就会被base64处理,这个limit值是可以根据实际情况自定义的。

          // 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

          // 解析时会出问题：[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]取图片的hash的前10位

          // [ext]取文件原来扩展名

          name: "[hash:10].[ext]"

        }

      },

      // 处理html文件的img图片（负责引入img，从而能被url-loader进行处理）【★★★★★★★】

      {

        test: /\.html$/,

        loader: "html-loader"

      }

    ]

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    })

  ],

  mode: "development"

};

**1-4.打包其他资源**

/\*

1.目的:打包其他资源

2.什么是其他资源:就是在打包过程种不需要做任何优化的资源，例如字体，图标等。

\*/

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  entry: "./src/index.js",

  output: {

    filename: "built.js",

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

  },

  module: {

    rules: [

      // 打包css资源

      {

        test: /\.css$/,

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

      },

      // 打包其他资源(除了html/js/css资源以外的资源)【★★★★★】

      {

        // 排除css/js/html资源

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

        loader: "file-loader",

        options: {

          name: "[hash:10].[ext]"

        }

      }

    ]

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    })

  ],

  mode: "development"

};

**1-5.devserver开发服务器配置**

/\*

1.目的: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~~）

2.如何配置开发服务器？【★★★★★★★】

(1)创建devServer开发服务器，添加项目构建后的路径；启动gzip压缩，配置端口号，配置是否自动发开浏览器

(2)在最外层目录下下载webpack-dev-server :npm i webpack-dev-server -D

(3)在当前07.devServer目录下运行以上指令 npx webpack-dev-server

\*/

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  entry: "./src/index.js",

  output: {

    filename: "built.js",

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

  },

  module: {

    rules: [

      {

        test: /\.css$/,

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

      },

      // 打包其他资源(除了html/js/css资源以外的资源)

      {

        // 排除css/js/html资源

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

        loader: "file-loader",

        options: {

          name: "[hash:10].[ext]"

        }

      }

    ]

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    })

  ],

  mode: "development",

  // 配置开发服务器 devServer：【★★★★★★★】

  // 特点：只会在内存中编译打包，不会有任何输出

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

  devServer: {

    // 项目构建后路径

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

    // 启动gzip压缩

    compress: true,

    // 端口号

    port: 3000,

    // 自动打开浏览器

    open: true

  }

};

**2.生产环境**

**整体代码**

const { resolve } = require("path");

// 作用:从js中抽取css资源

const MiniCssExtractPlugin = require("mini-css-extract-plugin");

// 作用:压缩css的插件

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

// 作用:打包html的插件

const HtmlWebpackPlugin = require("html-webpack-plugin");

// 定义nodejs环境变量：决定使用browserslist的哪个环境

process.env.NODE\_ENV = "production";

// 复用loader

const commonCssLoader = [

  MiniCssExtractPlugin.loader,

  "css-loader",

  {

    // 还需要在package.json中定义browserslist

    loader: "postcss-loader",

    options: {

      ident: "postcss",

      plugins: () => [require("postcss-preset-env")()]

    }

  }

];

module.exports = {

  entry: "./src/js/index.js", //入口文件

  output: {

    filename: "js/built.js", //输出到该文件

    path: resolve(\_\_dirname, "build") //输出文件的根目录文件，需要引入resolve。

  },

  module: {

    rules: [

      // 1.打包css资源【★★★★★★★★】

      {

        test: /\.css$/,

        use: [...commonCssLoader]

      },

      // 2.打包less资源【★★★★★★★★】

      {

        test: /\.less$/,

        use: [...commonCssLoader, "less-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，那么一定要指定loader执行的先后顺序：

          先执行eslint 再执行babel

      \*/

      //  3.打包js资源【★★★★★★★★】

      // (1).js语法检查【★★★★】

      {

        // 在package.json中eslintConfig --> airbnb

        test: /\.js$/,

        exclude: /node\_modules/, //排除掉node\_nodules，让其不检查。

        enforce: "pre", // 优先执行

        loader: "eslint-loader",

        options: {

          fix: true // 自动修复eslint的错误

        }

      },

      // (2).js兼容性处理【★★★★】

      {

        test: /\.js$/, //要检查的文件为js文件

        exclude: /node\_modules/, //排除node\_modules的转换；

        loader: "babel-loader", //依赖的loader

        options: {

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

          presets: [

            [

              "@babel/preset-env", //该包只能转换基本ES6及以上语法，如promise高级语法不能转换

              {

                useBuiltIns: "usage", // 按需加载,需要做兼容性处理的就做。

                corejs: { version: 3 }, // 指定core-js版本

                targets: {

                  // 指定兼容性做到哪个版本浏览器

                  chrome: "60",

                  firefox: "50"

                }

              }

            ]

          ]

        }

      },

      // 4-1.打包图片【★★★★★★★】

      {

        test: /\.(jpg|png|gif)/,

        loader: "url-loader",

        options: {

          limit: 8 \* 1024,

          name: "[hash:10].[ext]",

          outputPath: "imgs",

          esModule: false

        }

      },

      // 4-2.打包html中引入的图片【★★★★★★】

      {

        test: /\.html$/,

        loader: "html-loader"

      },

      // 5.打包其他图片【★★★★★★】

      {

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

        loader: "file-loader",

        options: {

          outputPath: "media" //指定输出目录，以build为根目录

        }

      }

    ]

  },

  plugins: [

    // 为提取出来的css文件选定路径。

    new MiniCssExtractPlugin({

      filename: "css/built.css"

    }),

    // 压缩css

    new OptimizeCssAssetsWebpackPlugin(),

    // 6.打包html资源

    new HtmlWebpackPlugin({

      template: "./src/index.html",

      // 压缩html资源

      minify: {

        collapseWhitespace: true,

        removeComments: true

      }

    })

  ],

  mode: "production"

};

**2-1.提取css为单独文件**

/\*

重点:

1.目的:提取css成单独文件。

2.步骤:【★★★★★★★】

(1)按照开发模式创建webpack基本配置框架;

(2)在最外层目录下载mini-css-extract-plugin插件，

(3)在该配置文件中引入该插件;

(4)在plugins中配置该插件，并给提取出来的css文件配置路径及命名

(5)禁用打包css资源中的style-loader，用 MiniCssExtractPlugin.loader代替，这样可以提取出js中的css。

3.将css单独提取出来的优点?

答:

(1)css不会再以style标签的方式引入到head中，而出现闪屏现象，而是单独提取出来，自动生成link引入样式;

(2)js的体积也较少了很多，解析速度会更好一些。

\*/

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

const MiniCssExtractPlugin = require("mini-css-extract-plugin"); //【★★★★★★★】

module.exports = {

  entry: "./src/js/index.js",

  output: {

    filename: "js/built.js",

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

  },

  module: {

    rules: [

      {

        test: /\.css$/,

        use: [

          // 创建style标签，将样式放入其中

          // 'style-loader',

          // 这个loader取代style-loader。作用：提取js中的css成单独文件

          MiniCssExtractPlugin.loader, //【★★★★★★★】

          // 将css文件整合到js文件中

          "css-loader"

        ]

      }

    ]

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    }),

    new MiniCssExtractPlugin({

      //【★★★★★★★】

      // 对输出的css文件进行重命名

      filename: "css/built.css"

    })

  ],

  mode: "development"

};

**2-2.css兼容性处理**

/\*

1.css兼容性配置的步骤:

(1)创建基本的webpack架构。并且已经实现了css文件单独提取

(2)在最外层目录下载 postcss-loader postcss-preset-env

(3)修改'postcss-loader'的配置，具体如下;

(4)找到package.json,配置browserslist里面的的信息，通过配置加载指定的css兼容性样式.

(5)如果想要启动开发模式下browserslist中的配置，则需要在该文件中配置Node的临时环境变量:process.env.NODE\_ENV = 'development';然后webpack就会在built中产生开发模式下css自动兼容的效果。

(6)如果想要启用生产模式下的css兼容性代码，则不需要设置Node环境变量，直接默认情况下就是生产模式下的css兼容性代码

2.css兼容性处理如何实现?

答：css兼容性处理需要依赖三个东西:

(1).需要使用一个库postcss;

(2)但是postcss要想在webpack中使用则需要下载post-Loader;

(3)还需要下载一个插件postcss-preset-env,该插件的作用是帮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，从而使得css兼容性精确到某一个浏览器具体版本。

\*/

// 作用:出口使用

const { resolve } = require("path");

// 作用:打包html

const HtmlWebpackPlugin = require("html-webpack-plugin");

// 作用:提取css资源

const MiniCssExtractPlugin = require("mini-css-extract-plugin");

// 作用:设置nodejs环境变量，加载browserslist中的开发模式下的css兼容性代码。

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

module.exports = {

  // 入口

  entry: "./src/js/index.js",

  // 出口

  output: {

    filename: "js/built.js",

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

  },

  // 模块

  module: {

    // loader配置

    rules: [

      // 1.打包css

      {

        test: /\.css$/,

        use: [

          MiniCssExtractPlugin.loader, //(1)将css从js中抽取出来

          "css-loader", //(2)将css代码解析为Js字符串。

          // 使用loader的默认配置

          // 'postcss-loader',

          // (3)修改loader的配置

          {

            loader: "postcss-loader",

            // 在options中修改psotcss-loader中的配置。

            options: {

              ident: "postcss",

              plugins: () => [

                // postcss的插件

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

              ]

            }

          }

          /\*(4)配置package.json中的browserslist。

            "browserslist": {

              // 开发环境 --> 设置node环境变量：process.env.NODE\_ENV = development

              "development": [

                "last 1 chrome version",

                "last 1 firefox version",

                "last 1 safari version"

              ],

              // 生产环境：默认是看生产环境

              "production": [

                ">0.2%",

                "not dead",

                "not op\_mini all"

              ]

            }

          \*/

        ]

      }

    ]

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    }),

    new MiniCssExtractPlugin({

      filename: "css/built.css"

    })

  ],

  mode: "development"

};

**2-3.js语法检查**

/\*

1.js语法检查配置步骤:【★★★★★★】

(1)在最外层目录下载如下四个包:eslint-loader  eslint eslint-config-airbnb-base  eslint-plugin-import

注解:① eslint-loader  eslint 作用是提供语法检查;

     ②如果想使用airbnb规格进行检查，需要安装eslint-config-airbnb-base  eslint-plugin-import eslint

(2)在rules中创建对象,配置js语法检查的相关参数。其中包括test、exclude、loader、options具体如下

(3)在package.json中配置检查规则。具体如下或者参考package.json中的"eslintConfig"

2.js语法检查原理:【★★★★★★】

(1)首先使用eslint-loader eslint库对js文件进行语法检查，但是一定要排除Node\_modules,否则会报错；

(2)但是eslint不知道要检查什么东西，所以需要写检查js的语法规则，js检查规则依赖于airbnb这个库，但是airbnb这个库又依赖于eslint eslint-config-airbnb-base  eslint-plugin-import这三个包，所以需要下载；

(3)随后在package.json中的"eslintConfig"中配置检查规则即可。

3.如果js中的某一行代码不想被eslint检查，则只需要在该行代码前加上如下代码:【★★★★★★】 eslint-disable-next-line

\*/

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  entry: "./src/js/index.js",

  output: {

    filename: "js/built.js",

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

  },

  module: {

    rules: [

      /\*

        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"  //让eslint继承airbnb规格检查

              }

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

      \*/

      {

        test: /\.js$/, //检查的内容为js代码

        exclude: /node\_modules/, //排除掉node\_nodules，让其不检查。

        loader: "eslint-loader", //依赖的包

        options: {

          // 自动修复eslint的错误

          fix: true

        }

      }

    ]

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    })

  ],

  mode: "development"

};

**2-4.js兼容性处理**

/\*

js兼容性处理:

1.背景:当源代码中有用到es6及以上语法时，如果打包后还是ed6及以上语法，那么部分浏览器是不支持es6语法的,此时es6语法就无法在浏览器中正常执行。所以此时就需要做js兼容性处理，将es6及以上格式的语法转换为es5及一下的语法，使得浏览器兼容可以正常读取。

2.js兼容型处理的步骤:

(1)在最外层目录下载babel-loader @babel/core  @babel/preset-env core-js 4个包;

(2)在rules中创建{},配置所有的js兼容性代码,重点记住预设中的内容配置,具体参考如下。

3.js兼容性处理的三种方式:

  共用包：babel-loader @babel/core

  (1). 基本js兼容性处理 --> @babel/preset-env

    问题：只能转换基本语法，如promise高级语法不能转换

  (2). 全部js兼容性处理 --> @babel/polyfill  ,下载后直接在入口js文件中引入即可。

    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~

  (3). 需要做兼容性处理的就做：按需加载  --> core-js

  推荐方式:(1)+(3)结合起来处理。

\*/

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  entry: "./src/js/index.js",

  output: {

    filename: "js/built.js",

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

  },

  module: {

    rules: [

      //  【★★★★★★★★★】

      {

        test: /\.js$/, //要检查的文件为js文件

        exclude: /node\_modules/, //排除node\_modules的转换；

        loader: "babel-loader", //依赖的loader

        options: {

          // 定义预设：指示babel做怎么样的兼容性处理

          presets: [

            [

              "@babel/preset-env", //该包只能转换基本ES6及以上语法，如promise高级语法不能转换

              {

                // 按需加载,需要做兼容性处理的就做。

                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: "17"

                }

              }

            ]

          ]

        }

      }

    ]

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    })

  ],

  mode: "development"

};

**2-5.js压缩**

/\*js压缩:只需要将Model设置为production即可。\*/

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  entry: "./src/js/index.js",

  output: {

    filename: "js/built.js",

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

  },

  plugins: [

    new HtmlWebpackPlugin({

      template: "./src/index.html"

    })

  ],

  // 生产环境下会自动压缩js代码

  mode: "production"

};

**2-5.html压缩**

const { resolve } = require("path");

const HtmlWebpackPlugin = require("html-webpack-plugin");

module.exports = {

  entry: "./src/js/index.js",

  output: {

    filename: "js/built.js",

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

  },

  plugins: [

    new HtmlWebpackPlugin({

      // 打包Html代码

      template: "./src/index.html",

      // 压缩html代码

      minify: {

        // 移除空格

        collapseWhitespace: true,

        // 移除注释

        removeComments: true

      }

    })

  ],

  mode: "production"

};