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"
};
```