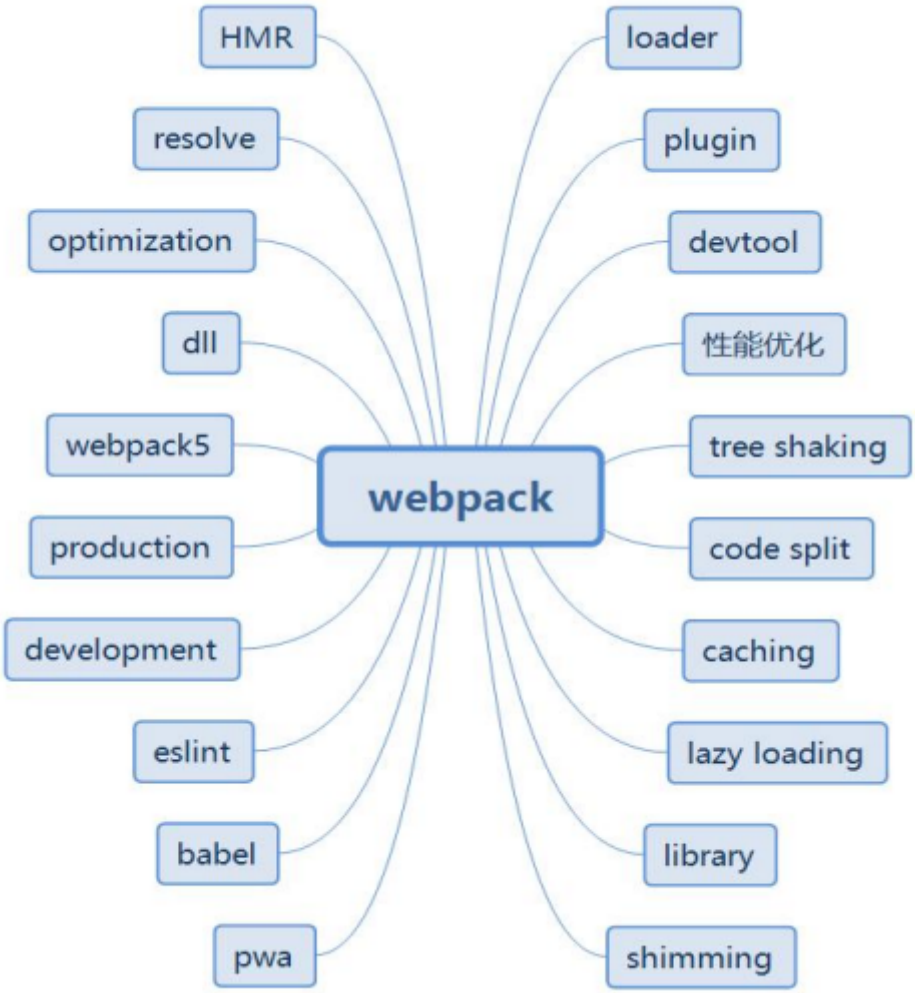
webpack 学习笔记

1. 第一章：webpack 简介

1.1. webpack 知识点

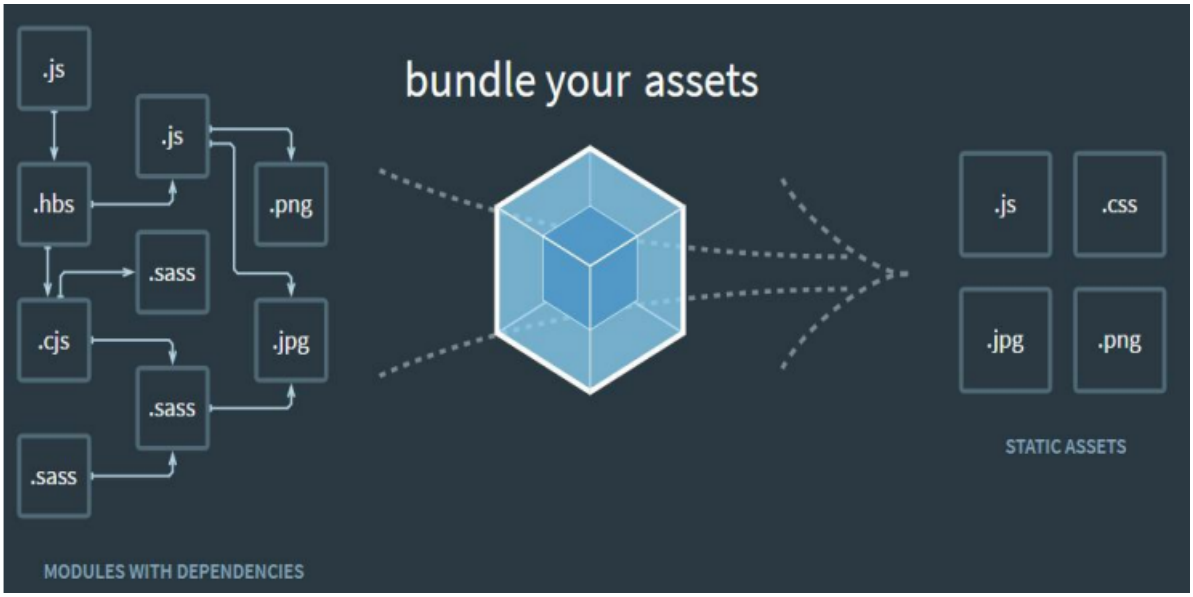


1.2. webpack 是什么

webpack 是一种前端资源构建工具，一个静态模块打包器(module bundler)。

在 webpack 看来，前端的所有资源文件(js/json/css/img/less/...) 都会作为模块处理。

它将根据模块的依赖关系进行静态分析，形成一个 chunk(块)，再打包(对各种静态资源进行处理) 生成对应的静态资源(bundle)。



1.3. webpack 五个核心概念

1.3.1. Entry

- 入口(Entry)指示 webpack 以哪个文件为入口七点开始打包，分析构建内部依赖图。

1.3.2. Output

- 输出(Output)指示 webpack 打包后的资源 bundles 输出到哪里去，以及如何命名。

1.3.3. Loader

- Loader 让 webpack 能够去处理那些非 JavaScript 文件(webpack自身只理解 JavaScript)。

1.3.4. Plugins

- 插件(Plugins)可以用于执行范围更广的任务。插件的范围包括，从打包优化到压缩，一直到重新定义环境的变量等。

1.3.5. Mode

- 模式(Mode)指示 webpack 使用相应模式的配置

选项	描述	特点
development	会将 DefinePlugin 中 precess.env.NODE_ENV 的值设置为 development。启用NamedChunksPlugin 和 NamedModulesPlugin.	能让代码本地调试运行的环境
production	会将 DefinePlugin 中 precess.env.NODE_ENV 的值设置为 production。启用 Flag DependencyUsagePlugin，FlagIncludedChunksPlugin，ModuleConcatenationPlugin，NoEmitOnErrorsPlugiin，OccurrenceOrderPlugin，SideEffectsFlagPlugin 和 TerserPlugin。	能让代码优化上线运行的环境

2. 第二章：webpack 的初体验

2.1. 初始化配置

1. 初始化package.json

```
npm init
```

2. 下载并安装 webpack

```
## 先全局安装
npm install webpack webpack-cli -g
## 再本地安装，-D表示添加到开发依赖
npm install webpack webpack-cli -D
```

2.2. 编译打包应用

1. 创建文件
2. 运行指令
   - 开发环境指令：`webpack --entry ./src/js/index.js -o build/js/built.js --mode=development`
     - 说明：webpack 会以 ./src/js/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(--mode=development)。
     - 功能：webpack 能够编译打包 js 和 json 文件，并且能够将 es6 的模块化语法转换成浏览器能识别的语法。
   - 生产环境指令：`webpack --entry src/js/index.js -o build/js/built.js --mode=production`

        - 说明：webpack 会以 ./src/js/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：生产环境(--mode=production)。
        - 功能：在开发配置功能上多一个功能，压缩代码。
3. 结论
    - webpack 能够编译 js 和 json 文件，不能处理 css/img 等其他资源。
    - 生产环境和开发环境都能够将 es6 模块语法转换成浏览器能识别的语法。
    - 生产环境比开发环境多一个压缩代码功能。
4. 问题：
    - 不能编译打包 css、img 等文件。
    - 不能将 js 的 es6 基本语法转化为 es5 以下语法。

---

3. 第三章：webpack 开发环境的基本配置

3.1. 创建配置文件

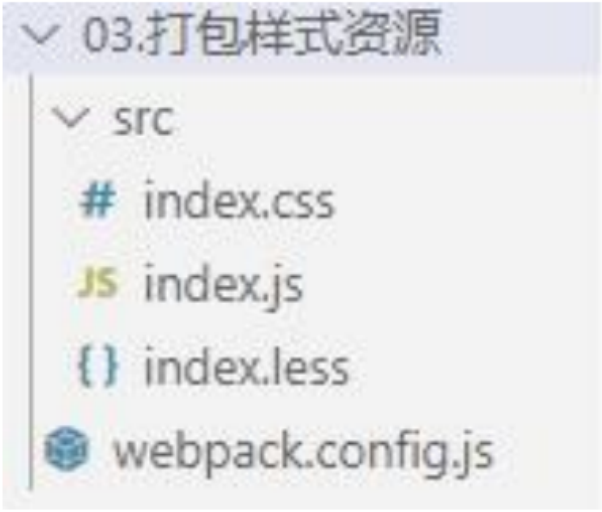
1. 创建文件 webapck.config.js（webpack的配置文件）
    - 作用：只是 webpack 干哪些活（当你运行 webpack 指令时，会加载里面的配置
    - 所有的构建工具都是基于 nodejs 平台运行的~模块化默认采用 commonjs。
2. 配置内容如下：

```
const { resolve } = require('path') // node 内置核心模块，用来处理路径问题。

mudule.exports = {
  entry: './src/js/index.js', // 入口文件
  output: {                   // 输出配置
    filename: './built.js',  // 输出文件名
    path: resolve(__dirname, 'build/js')  // 输出文件路径配置
  },
  mode: 'development'  // 开发环境
}
```

3. 运行指令：webpack
4. 结论：此时功能与上节一致

3.2. 打包样式资源

1. 创建文件



2. 下载安装 loader 包

```
npm i css-loader style-loader less-loader less -D
```

3. 修改配置文件

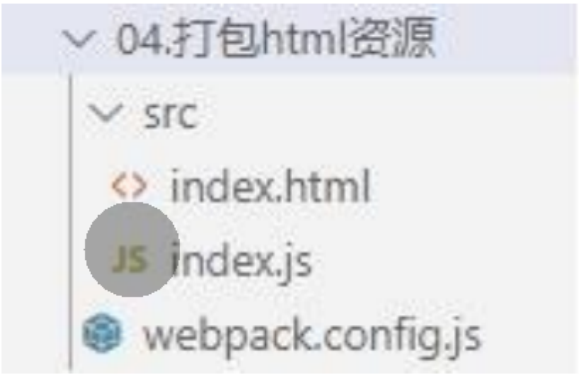
```
// resolve 是用来拼接绝对路径的方法
const { resolve } = require('path')

module.exports = {
  // webpack 配置
  // 入口起点
  entry: './src/index.js',
  // 输出
  output: {
    // 输出文件名
    filename: 'built.js',
    // 输出路径
    // __dirname nodejs的变量，代表当前文件的目录绝对路径
    path: resolve(__dirname, 'build')
  },
  // loader 的配置
  module: {
    rules: [
      // 详细 loader 配置
      // 不同文件必须配置不同的 loader 处理
      {
        // 匹配哪些文件
        test: /\.css$/,
        // 使用哪些 loader 进行处理
        use: [
          // use 数组中 loader 执行顺序：从右到左，从下到上 依次执行
          // 创建 style 标签，将 js 中的样式资源进行插入，添加到 head 中生效
          'style-loader',
          // 将css文件变成commonjs模块加载到js中，里面内容是样式字符串
          'css-loader'
        ]
      },
      {
        test: /\.less$/,
```

```
        use: [
          'style-loader',
          'css-loader',
          // 将 less 文件编译成 css 文件
          // 需要下载 less-loader 和 less
          'less-loader'
        ]
      }
    ]
  },
  // plugins 的配置
  plugins: [
    // 详细的 plugins 的配置
  ],
  // 模式
  mode: 'development', // 开发模式
  // mode: 'production' // 生产模式
}
```

4. 运行指令：`webpack`

3.3. 打包 HTML 资源

1. 创建文件



2. 下载安装 plugin 包

```
npm install --save-dev html-webpack-plugin
## npm i html-webpack-plugin -D // 简写
```

3. 修改配置文件

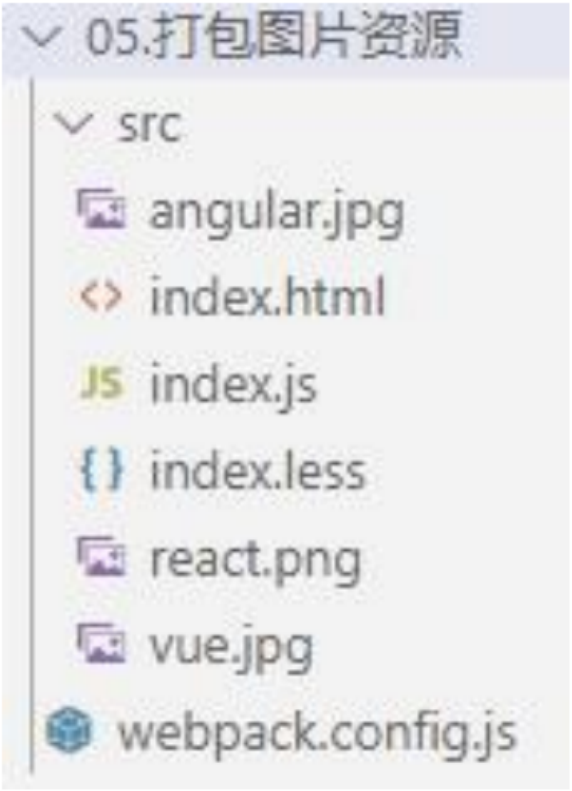
```
/*
  loader: 1. 下载  2. 使用（配置loader）
  plugin: 1.下载  2. 引入  3. 使用
*/

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
    ]
  },
  plugins: [
    // plugins 的配置
    // 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）
    // 需求：需要有结构的 HTML 文件
    new HtmlWebpackPlugin({
      // 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/insec.html'
    })
  ],
  mode: 'development'
}
```

4. 运行指令：`webpack`

3.4. 打包图片资源

1. 创建文件



2. 下载安装 loader 包

```
npm install --save-dev html-loader url-loader file-loader
## --save-dev 简写是 -D
```

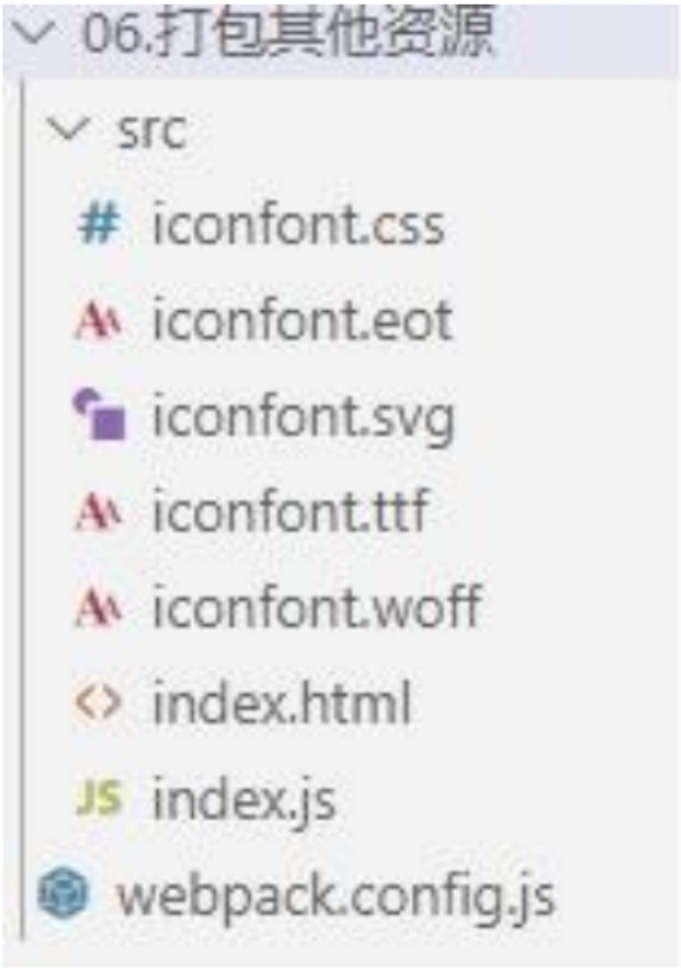
3. 修改配置文件

```
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:/\.less$/,
        // 要使用多个 loader 处理用 use，单个直接用 loader
        use: ['style-loader', 'css-loader', 'less-loader']
      },
      {
        // 问题：默认处理不了 html 中的 img 图片
        // 处理图片资源
        test: /\.(jpg|png|gif)$/,
        type: 'asset',
        generator: {
          // 给图片进行重命名
          // [hash:10]取图片的 hash 的前 10 位
          // [ext]取文件原来的扩展名
          filename: 'img/[name]_[contenthash:6][ext]'
        },
        parser: {
          dataUrlCondition: {
            // 图片大小小于 8kb，就会被 base64 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（文件请求速度更慢）
            maxSize: 200 * 1024
          }
        }
      },
      {
        test: /\.html$/,
        // 处理 html 文件的 img 图片（负责引入 img，从而能被 url-loader 进行处理）
        loader: 'html-loader'
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ],
  mode: 'development'
}
```

4. 运行指令：`webpack`

3.5. 打包其他资源

1. 创建文件



2. 修改配置文件

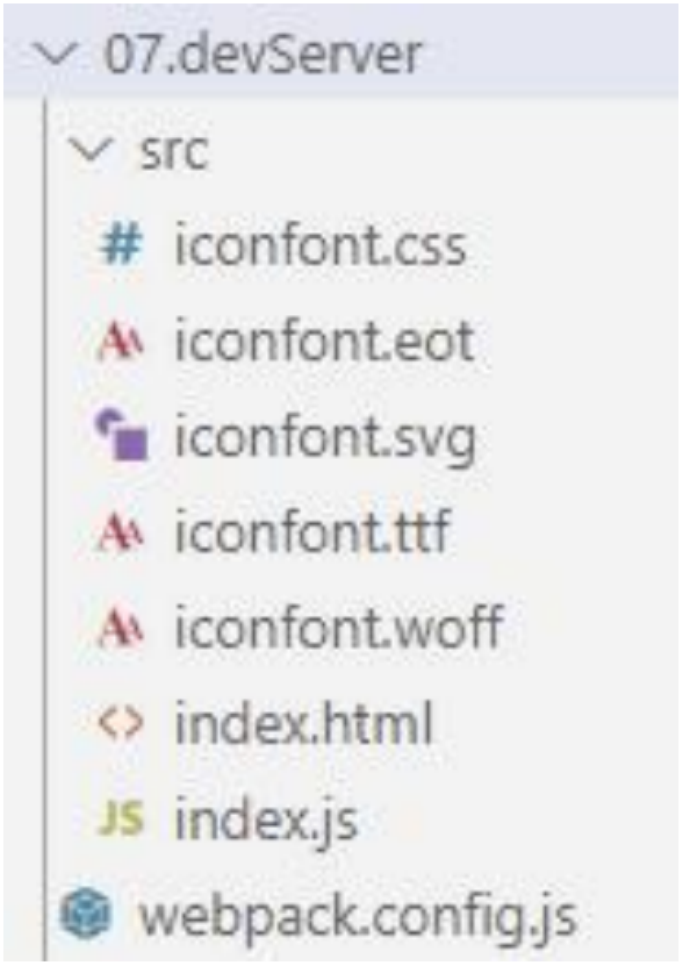
```
const { resolve } = require('path')
const HtmlWebpackPlugin = require(html-webpack-plugin)

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      /* {
        // webpack 4，已弃用
        exclude: /\.(css|js|html|less)$/,
        loader: 'file-loader',
      }, */
      {
        // webpack 5 用法
        test: /\.(eot|ttf|woff|svg)$/,
        type: 'asset/resource',
        generator: {
          filename: 'font/[name]_[contenthash:6][ext]'
        }
```

```
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ],
  mode: 'development'
}
```

3. 运行指令：`webpack`

3.6. devServer

1. 创建文件



2. 修改配置文件

```
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$/,
        use: ['style-loader', 'css-loader']
      },
      // 打包其他资源（除了 html/js/css 资源意外的资源）
      {
        // 排除 css/js/html 资源
        exclude: /\.(css|js|html|less)$/,
        loader: 'file-loader',
        options: {
          name: '[hash:10].[ext]'
        }
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ],
  mode: 'development',

  // 开发服务器 devServer（热重载）：用来自动化（自动编译，自动打开浏览器，自动刷新浏览器）
  // 特点：只会在内存中编译打包，不会有任何输出到本地代码
  // 启动 devServer 指令为：npx webpack serve
  devServer: {
    // 运行代码的目录
    static: {
      directory: resolve(__dirname, 'build'),
    },
    // 启动 gzip 压缩
    compress: true,
    // 端口号
    port: 3000,
    // 自动打开浏览器
    open: true,
  },
}
```

3. 运行指令：`npx webpack-dev-server`

3.7. 开发环境配置

- 注意：用到的 loader 包 跟 plugin 插件都要提前下载好

1. 创建文件

修改配置文件

```
/*
  开发环境配置：能让代码运行
    运行项目指令：
      webpack 会打包结果输出出去
      npx webpack serve 只会在内存中编译打包，没有输出
*/

const HtmlWebpackPlugin = require('html-webpack-plugin')
const { resolve } = require('path')

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      },
      {
        test: /\.less$/,
        use: ['style-loader', 'css-loader', 'less-loader']
      },
      {
        // 处理图片资源
        test: /\.(jpg|png|jpeg|gif)$/i,
        type: 'asset',
        generator: {
          filename: 'imgs/[name]_[contenthash:6][ext]'
        },
        parser: {
```

```
          dataUrlCondition: {
            maxSize: 8 * 1024
          }
        }
      },
      {
        // 处理html中的 图片资源
        test: /\.html$/,
        loader: 'html-loader'
      },
      {
        // 处理字体资源
        test: /\.(eot|ttf|woff|svg)$/,
        type: 'asset/resource',
        generator: {
          outputPath: 'media'
        }
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ],
  mode: 'development',
  devServer: {
    static: {
      directory: resolve(__dirname, 'build')
    },
    compress: true,
    port: 5000,
    open: true
  }
}
```

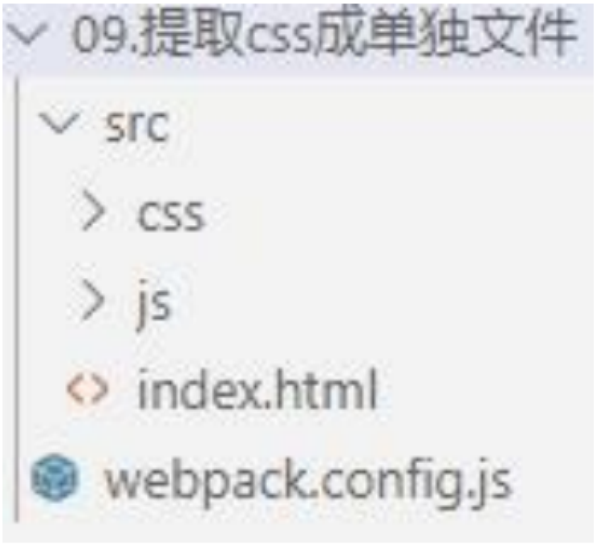
3. 运行指令：`npx webpack serve`

---

4. 第四章：webpack 生产环境的基本配置

4.1. 提取 css 成单独文件

1. 创建文件



2. 下载安装包

3. 下载插件

```
npm install --save-dev mini-css-extract-plugin
```

4. 修改配置文件

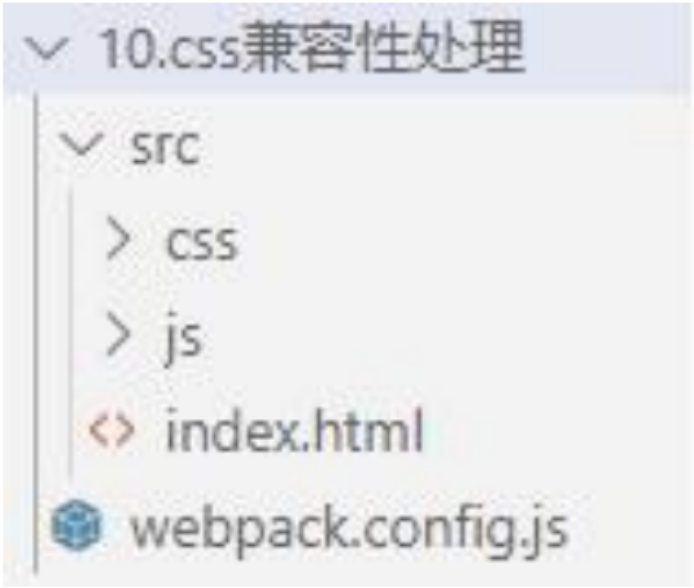
```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [
          // 创建 style 标签，将样式放入
          // 'style-loader',
          // 这个 loader 取代 style-loader。作用：提取 js 中的 css 成单独文件
          MiniCssExtractPlugin.loader,
          // 将 css 文件整合到 js 文件中
          'css-loader'
        ]
      },
      {
        test: /\.(jpg|png|gif)$/,
        type: 'asset'
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
```

```
    }),
    // 将css提取成单独文件
    new MiniCssExtractPlugin({
      filename: 'css/built.css'
    })
  ],
  mode: 'development'
}
```

5. 运行指令：`webpack`

4.2. css 兼容性处理

1. 创建文件



2. 下载 loader

```
npm install --save-dev postcss-loader postcss-preset-env
## --save-dev 相当于 -D
```

3. 修改配置文件

webpack.config.js:

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')

// 设置 nodejs 环境变量
// process.env.NODE_ENV = 'development'

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
```

```
    rules: [
      {
        test: /\.css$/,
        use: [
          MiniCssExtractPlugin.loader,
          'css-loader',
          /*
            css 兼容性处理：postcss --> postcss-loader postcss-preset-env

            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
            // 更多的browserverslist配置可以去github搜索 关键字`browserverslsit`
            "browserslist": {
              // 开发环境 --> 设置nodejs环境变量：process.env.NODE_ENV =
development
              "development": [
                "last 1 chrome version", // 兼容最近的一个 chrome 版本
                "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"
              ]
            }
          */
          // 使用 loader 的默认配置
          // 'postcss-loader'
          // 修改loader的配置（写成对象的形式）
          {
            loader: 'postcss-loader',
            options: {
              postcssOptions: {
                plugins: [require('postcss-preset-env')()],
              },
            },
          },
        ]
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    }),
    new MiniCssExtractPlugin({
      filename: 'css/built.css'
    })
  ],
  mode: 'development'
}
```

4. 修改 package.json

```
"browserslist": {
```

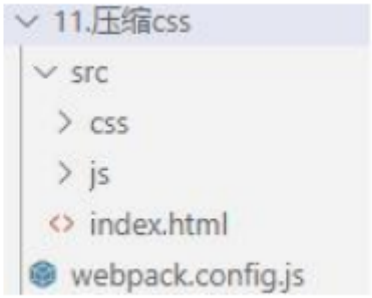
```
  // 开发环境 --> 设置nodejs环境变量：process.env.NODE_ENV = development，不设置默认是production
  "development": [
    "last 1 chrome version", // 兼容最近的一个 chrome 版本
    "last 1 firefox version",
    "last 1 safari version"
  ],
  // 生产环境：默认是看生产环境
  "production": [
    ">0.2%",
    "not dead",
    "not op_mini all"
  ]
}
```

- 更多的browserverslist配置可以去github搜索 关键字

4. 运行指令：`webpack`

4.3. 压缩 css

1. 创建文件



2. 下载安装包

```
npm installoptimize-css-assets-webpack-plugin -D
```

3. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-webpack-plugin')

// 设置 nodejs 环境变量
process.env.NODE_EVN = 'development'

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build'),
  },
  module: {
    rules: [
```

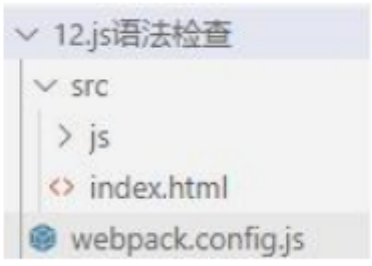
```
      {
        test: /\.css$/,
        use: [
          // 插件 style 标签，将样式放入
          // 'style-loader',
          // 这个 loader 取代 style-loader 。作用：提取js中的css成单独文件
          MiniCssExtractPlugin.loader,
          // 将css 文件整合到js文件中
          'css-loader',

          // 使用loader默认配置
          // 直接：loader: 'postcss-loader',
          // 修改loader配置，需要写到对象里面
          {
            loader: 'postcss-loader',
            options: {
              postcssOptions: {
                plugins: ['postcss-preset-env'],
              },
            },
          },
        ],
      },
      {
        test: /\.(jpg|png|gif)$/,
        type: 'asset',
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
    new MiniCssExtractPlugin({
      filename: 'css/built.css',
    }),
    // 压缩css
    new CssMinimizerWebpackPlugin(),
  ],
  mode: 'development',
}
```

4. 运行指令：`webpack`

4.4. js 语法检查（eslint）

1. 创建文件

!



2. 下载安装包

```
npm install eslint-loader eslint eslint-config-airbnb-base eslint-plugin-import -D
```

3. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const OptimizeCssAssetsWebpackPlugin = require('optimize-css-assets-webpack-plugin')

// 设置 nodejs 环境变量
// process.env.NODE_ENV

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"
              }
            airbnb --> eslint-config-airbnb-base eslint-plugin-import eslint
      */
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'eslint-loader',
        options: {
          // 自动修复 eslint 的错误
          fix: true
        }
      }
    ]
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    }),
    new MiniCssExtractPlugin({
      filename: 'css/built.css'
    }),
    new OptimizeCssAssetsWebpackPlugin()
  ],
  mode: 'development'
}
```

打包时，在 js 文件中，不推荐出现console.log，可以写成

```
// index.js
// 下一行 eslint 所有规则都失效（下一行不进行eslint检查）
// eslint-disable-next-line
console.log('innn')
```

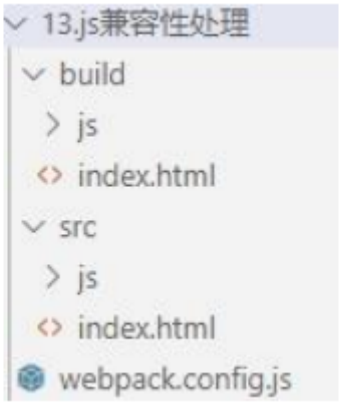
4. 配置 package.json

```
"eslintConfig": {
  "extends": "airbnb-base",
  "env": {
    "browser": true
  }
}
```

5. 运行指令：`webpack`

4.5. js 兼容性处理

1. 创建文件



2. 下载安装包

```
npm install babel-loader @babel/core @babel/preset-env @babel/polyfill
core-js
```

3. 修改配置文件

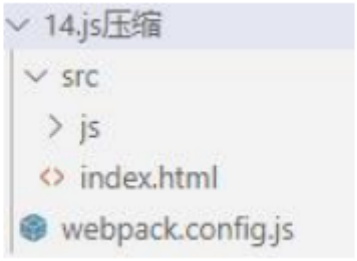
```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      /*
        js兼容性处理：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等高级语法不能转换
          2. 全部兼容性处理 --> @babel/polyfill
            用法：直接在js文件中引入：import '@babel/polyfill'
            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 --> corejs
            使用第三种方案的时候，不能在用第二种方案，必须注释掉
        总结：我们是结合第一种（基本兼容性处理）和第三种方法（高级语法兼容性处理）完成兼容型处理，第二种体积太大了一般不考虑
      */
      {
        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',
            {
              // 按需加载
              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'
}
```

4. 运行命令：`webpack`

4.6. js 压缩

1. 创建文件

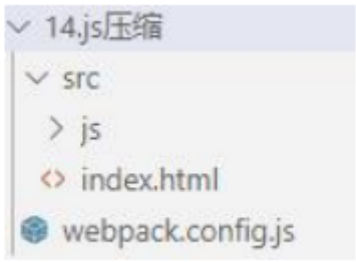


2. 修改配置文件

```
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
  // 生产环境下会自动加载很多插件，其中的 UglifyPlugin(现在用terser插件) 就是用来压缩 js 代码的
  mode: 'production'
}
```

4.7. HTML 压缩

1. 创建文件



2. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/buit.js',
    path: resolve(__dirname, 'build')
  },
  plugins: [
```

```
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩 html 代码
      minify: {
        // 移除空格
        collapseWhitespace: true,
        // 移除注释
        removeComments: true
      }
    })
  ],
  mode: 'production'
}
```

4.8. 生产环境配置

1. 创建文件

2. 修改配置文件（用到的插件和loader要提前下载好）

```
const { resolve } = require('path')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const OptimizeCssAssetsWebpackPlugin = require('optimeze-css-webpack-plugin')  // css压缩
const HtmlWebpackPlugin = require('html-webpack-plugin')  // 生成并引入index.html

// 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',
  {
    // css兼容性处理
    // 还需要在 package.json中定义browserslist
    loader: 'postcss-loader',
    options: {
      postcssOptions: {
        plugins: [require('postcss-preset-env')()]
      }
    }
  }
]

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
```

```
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [...commonCssLoader]
      },
      {
        test: /\.less$/,
        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
      */
      {
        // js 语法检查
        // 在 package.json 中 eslintConfig --> airbnb
        test: /\.js$/,
        exclude: /node_modules/,
        // 优先执行
        enforce: 'pre',
        loader: 'eslint-loader',
        options: {
          // 自动修正
          fix: true
        }
      },
      {
        // js 兼容性处理
        test:/\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        options: {
          presets: [
            [
              '@babel/preset-env',
              {
                // 按需加载
                useBuiltIns: 'usage',
                corejs: {version: 3},
                // 要兼容的浏览器版本
                targets: {
                  chrome: '60',
                  firefox: '50'
                }
              }
            ]
          ]
        }
      },
      {
        // 处理图片资源
        test: /\.(jpg|png|gif|jpeg)$/i,
        type: 'asset',
        generator: {
```

```
          filename: '[name]_[contenthash:6][ext]'
        },
        parser: {
          dataUrlCondition: {
            maxSize: 10 * 1024
          }
        }
      },
      {
        // 处理 html 中的图片
        test:/\.html$/,
        loader: 'html-loader'
      },
     {
        // 处理其他资源
        exclude: /\.(js|css|less|html|jpg|png|gif|jpeg)$/,
        type: 'asset/resource',
        generator: {
          filename: 'media/[name]_[contenthash:6][ext]'
        }
      }
    ]
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      filename: 'css/built.css'
    }),
    // 压缩 css
    new CssMinimizerWebpackPlugin()
  ],
  mode: 'production'
}
```

3. 运行指令：`webpack`

5. 第五章：webpack 优化配置

5.1. webpack性能优化

- 开发环境性能优化
- 生产环境性能优化

5.1.1. 开发环境性能优化

- 优化打包构建速度（HMR）
    - 问题：只要修改了一个模块，重新构建时其他模块也会重新构建
    - HMR（热模块替换）：构建时如果只有一个模块发生变化，只会重新构建这一个模块，而其他模块会用它之前的缓存
- 优化代码调试（source-map)
    - source-map

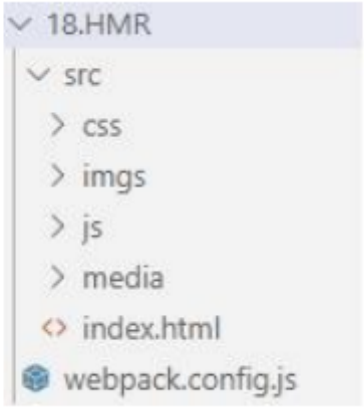
5.1.2. 生产环境性能优化

- 优化打包构建速度
    - oneOf
    - babel 缓存
    - 多进程打包
    - externals(完全不打包，通过链接引入)
    - dll(打包一次，之后就不打包了)
- 优化代码运行的性能
    - 缓存（文件资源）
        - hash
        - chunkhash
        - contenthash
    - tree shaking（树摇）
    - code split（代码分割）
        - 可以配合 dll 把node_modules库中的各种第三方库都分
    - 懒加载 / 预加载
    - pwa（离线可访问技术）

5.2. HMR（模块热更新）

说明：使我们代码在更新的时候不是全部更新，而是只更新变化的部分

1. 创建文件



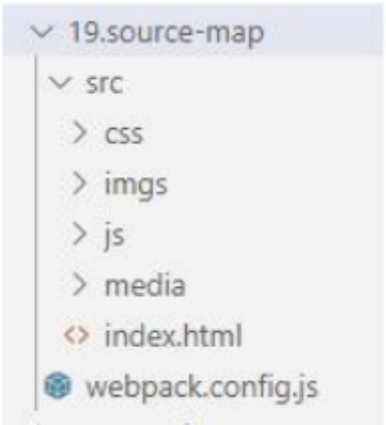
2. 修改配置文件

```
/*
  HMR：hot module replacement 热模块替换 / 模块热更新
    作用：一个模块发生变化，只会重新打包这一个模块（而不是打包所有模块）
      极大提升构建速度

      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 --> 需要修改 js 代码，在js文件中添加支持HMR功能的代码
        注意：HMR功能对js的处理，只能处理非入口js文件的其他文件。
        if(module.hot) {
          // 一旦 module.hot 为 true，说明开启了 HMR 功能。 --> 让HMR功能代码生效
          module.hot.accept('./print.js', function () { // 方法会监听print.js文件的变化，一旦发生变化，其他模块不会重新打包构建。
          // 会执行后面的回调函数
            print()
          })
        }
      html文件：默认不能使用 HMR 功能，同时会导致问题：html 文件不能热更新了~（不用做HMR功能）
        解决：修改entry入口，将 html 文件引入
*/

const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  entry: ['./src/js/index.js', './src/index.html'],
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      // loader 配置
      {
        // 处理 less 资源
        test: /\.less$/,
        use: ['style-loader', 'css-loader', 'less-loader']
      },
      {
        // 处理 css 资源
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      },
      {
        // 处理图片资源
        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'
        }
```

```
      },
      {
        // 处理 html 中 img 资源
        test: /\.html$/,
        loader: 'html-loader'
      },
      {
        // 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: [
    // plugins 的配置
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ],
  mode: 'development',
  devServer: {
    static: {
      directory: resolve(__dirname, 'build'),
    },
    compress: true,
    port: 5000,
    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服务
    // hot: true, // webpack5 默认开启
  },
}
```

5.3. source-map（代码映射）

1. 创建文件



2. 修改配置文件

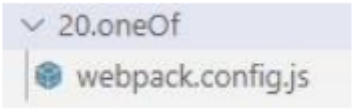
```
const HtmlWebpackPlugin = require('html-webpack-plugin')
const { resolve } = require('path')

module.exports = {
  entry: ['./src/js/index.js', './src/index.html'],
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build'),
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.less$/,
        use: ['style-loader', 'css-loader', 'less-loader'],
      },
      {
        // 处理图片资源
        test: /\.(jpg|png|jpeg|gif)$/i,
        type: 'asset',
        generator: {
          filename: 'imgs/[name]_[contenthash:6][ext]',
        },
        parser: {
          dataUrlCondition: {
            maxSize: 8 * 1024,
          },
        },
      },
      {
        // 处理html中的 图片资源
        test: /\.html$/,
        loader: 'html-loader',
      },
      {
        // 处理字体资源
        test: /\.(eot|ttf|woff|svg)$/,
        type: 'asset/resource',
        generator: {
          outputPath: 'media',
        },
      },
    ],
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
    }),
  ],
  mode: 'development',
  devServer: {
    static: {
      directory: resolve(__dirname, 'build'),
    },
    compress: true,
```

```
    port: 5000,
    open: true,
    hot: true, // webpack5 默认开启
  },
  devtool: 'eval-source-map',
}

/*
  source-map：一种提供源代码到构建后代码映射的技术（如果构建后代码出错了，通过映射可以追踪到源代码错误）

  [inline-|hidden-|eval-][nosources-][chep-[module-]]source-map

  source-map：外部
    错误代码的准确信息 和 源代码的
  inline-source-map：内联
    只生成一个内联source-map
    错误代码的准确信息 和 源代码的错误位置
  hidden-source-map：外部
    错误代码的错误原因，但是没有错误位置
    不能追踪到源代码错误，只能提示到构建后代码的错误位置
  eval-source-map：内联
    每一个文件都生成一个对应的source-map，都在eval
    错误代码的准确信息 和 源代码的错误位置
  nosources-source-map：外部
    错误代码的准确信息，但是没有任何源代码信息
  cheap-source-map：外部
    错误代码的准确信息 和 源代码的错误位置
    只能精确到行
  cheap-module-source-map：外部
    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

  内联 和 外部的区别：1. 外部生成了文件，内联没有 2. 内联构建速度更快

  开发环境：速度快，调式更友好
    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(开发环境一般用这个)  /  eval-cheap-module-source-map

  生产环境：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，所以在生产环境不用内联
    nosources-source-map 全部隐藏
    hidden-source-map 只隐藏源代码，会提示构建后代码错误

    --> source-map(生产环境一般用这个)  /  cheap-module-source-map

*/
```

3. 运行指令：`webpack`

5.4. oneOf（loader 限单次匹配）

1. 创建文件



2. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-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',
  {
    // css兼容性处理
    // 还需要再 package.json中定义browserslist
    loader: 'postcss-loader',
    options: {
      postcssOptions: {
        plugins: [require('postcss-preset-env')()],
      },
    },
  },
]

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        // 在package.json中设置eslintConfig  --> airbnb
        test: /\.js$/,
        exclude: /node_modules/,
        // 优先执行
        enforce: 'pre',
        loader: 'eslint-loader', // eslint 语法检查
        options: {
          fix: true
        }
      },
      {
```

```
        // 一下loader只会匹配一个，解决每个文件都会被全部loader过一遍（相当于找到直接break掉）
        // 不能有两个配置处理同一类型的文件
        oneOf: [
          {
            test: /\.css$/,
            use: [...commonCssloader]
          },
          {
            test: /\.less$/,
            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
      */

          {
            // js兼容性处理
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel-loader',
            options: {
              presets: [
                [
                  '@babel/preset-env',
                  {
                    useBuiltIns: 'usage',
                    corejs: {
                      version: 3
                    },
                    targets: {
                      chrome: '60',
                      firefox: '50',
                      ie: '9',
                      safari: '10'
                    }
                  }
                ]
              ]
            }
          },
          {
            // 处理图片资源
            test: /\.(jpg|png|gif|jpeg)$/i,
            type: 'asset',
            generator: {
              filename: '[name]_[contenthash:6][ext]'
            },
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024
              }
            }
          },
          {
            // 处理html 中的图片资源
            test: /\.html$/,
```

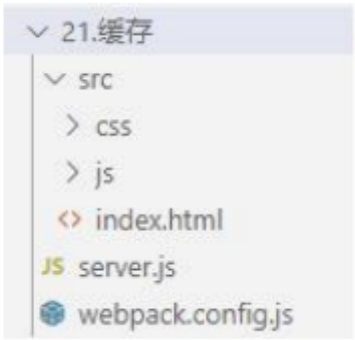
```
            loader: 'html-loader'
          },
          {
            // 处理其他资源
            exclude: /\.(js|css|less|html|jpg|png|gif|jpeg)$/,
            type: 'asset/resource',
            generator: {
              filename: 'media/[name]_[contenthash:6][ext]'
            }
          }
        ]
      }
    ]
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      filename: 'css/built.css'
    }),
    // 压缩 css
    new CssMinimizerWebpackPlugin()
  ],
  mode: 'production'
}
```

3. 运行指令：`webpack`

5.5. 缓存

- babel 缓存
- 文件资源缓存

1. 创建文件



2. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-webpack-plugin')

/*
  缓存:
    babel缓存
      cacheDirectory: true
      --> 让第二次打包更快
    文件资源缓存
      hash：每次webpack构建时会生成一个唯一的hash值。
        问题：因为js和css同时使用一个hash值。
          如果重新打包，会导致所有的缓存失效。（可能我却只改动一个文件）
      chunkhash：根据chunk生成的hash值。如果打包来源于同一个chunk，那么hash值一样
        问题：js和css的hash值还是一样的
          因为css是在js中被引入的，所以同属于一个chunk
      contenthash：根据文件的内容生成hash值。不同文件hash值一定不一样
      --> 让代码上线运行缓存更好使用
*/

// 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',
  {
    // css兼容性处理
    // 还需要再 package.json中定义browserslist
    loader: 'postcss-loader',
    options: {
      postcssOptions: {
        plugins: [require('postcss-preset-env')()]
      }
    }
  }
]

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.[contenthash:10].js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        // 在package.json中设置eslintConfig  --> airbnb
        test: /\.js$/,
        exclude: /node_modules/,
        // 优先执行
        enforce: 'pre',
        loader: 'eslint-loader', // eslint 语法检查
        options: {
          fix: true
        }
```

```
      },
      {
        // 以下loader只会匹配一个，解决每个文件都会被全部loader过一遍（相当于找到直接break掉）
        // 不能有两个配置处理同一类型的文件
        oneOf: [
          {
            test: /\.css$/,
            use: [...commonCssloader]
          },
          {
            test: /\.less$/,
            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
      */

          {
            // js兼容性处理
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel-loader',
            options: {
              presets: [
                [
                  '@babel/preset-env',
                  {
                    useBuiltIns: 'usage',
                    corejs: {
                      version: 3
                    },
                    targets: {
                      chrome: '60',
                      firefox: '50',
                      ie: '9',
                      safari: '10'
                    }
                  }
                ]
              ],
              // 开启babel缓存，第二次构建时，会读取之前的缓存
              cacheDirectory: true
            }
          },
          {
            // 处理图片资源
            test: /\.(jpg|png|gif|jpeg)$/i,
            type: 'asset',
            generator: {
              filename: '[name]_[contenthash:6][ext]'
            },
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024
              }
            }
```

```
          },
          {
            // 处理html 中的图片资源
            test: /\.html$/,
            loader: 'html-loader'
          },
          {
            // 处理其他资源
            exclude: /\.(js|css|less|html|jpg|png|gif|jpeg)$/,
            type: 'asset/resource',
            generator: {
              filename: 'media/[name]_[contenthash:6][ext]'
            }
          }
        ]
      }
    ]
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      filename: 'css/built.[contenthash:10].css'
    }),
    // 压缩 css
    new CssMinimizerWebpackPlugin()
  ],
  mode: 'production',
  devtool: 'source-map'
}

```

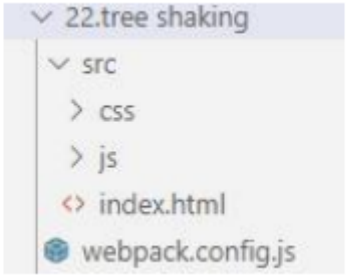
package.json

```
"eslintConfig": {
  "extends": "airbnb-base",
  "env": {
    "browser": true
  }
},
```

3. 运行指令： `webpack`

5.6. tree shaking（树摇）

1. 创建文件



2. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-webpack-plugin')

/*
  tree shaking：去除无用代码
    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中配置
      "sideEffects": false  所有代码都没有副作用（都可以进行tree shaking)
        问题：可能会把css / @babel/polyfill（副作用）文件干掉
      "sideEffects": [*.css]
*/

// 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',
  {
    // css兼容性处理
    // 还需要再 package.json中定义browserslist
    loader: 'postcss-loader',
    options: {
      postcssOptions: {
        plugins: [require('postcss-preset-env')()]
      }
    }
  }
]

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.[contenthash:10].js',
    path: resolve(__dirname, 'build')
  },
  module: {
```

```
    rules: [
      {
        // 在package.json中设置eslintConfig  --> airbnb
        test: /\.js$/,
        exclude: /node_modules/,
        // 优先执行
        enforce: 'pre',
        loader: 'eslint-loader', // eslint 语法检查
        options: {
          fix: true
        }
      },
      {
        // 以下loader只会匹配一个，解决每个文件都会被全部loader过一遍（相当于找到直接break掉）
        // 不能有两个配置处理同一类型的文件
        oneOf: [
          {
            test: /\.css$/,
            use: [...commonCssloader]
          },
          {
            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
      */

          {
            // js兼容性处理
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel-loader',
            options: {
              presets: [
                [
                  '@babel/preset-env',
                  {
                    useBuiltIns: 'usage',
                    corejs: {
                      version: 3
                    },
                    targets: {
                      chrome: '60',
                      firefox: '50',
                      ie: '9',
                      safari: '10'
                    }
                  }
                ]
              ],
              // 开启babel缓存，第二次构建时，会读取之前的缓存
              cacheDirectory: true
            }
          },
          {
```

```
            // 处理图片资源
            test: /\.(jpg|png|gif|jpeg)$/i,
            type: 'asset',
            generator: {
              filename: '[name]_[contenthash:6][ext]'
            },
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024
              }
            }
          },
          {
            // 处理html 中的图片资源
            test: /\.html$/,
            loader: 'html-loader'
          },
          {
            // 处理其他资源
            exclude: /\.(js|css|less|html|jpg|png|gif|jpeg)$/,
            type: 'asset/resource',
            generator: {
              filename: 'media/[name]_[contenthash:6][ext]'
            }
          }
        ]
      }
    ]
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      filename: 'css/built.[contenthash:10].css'
    }),
    // 压缩 css
    new CssMinimizerWebpackPlugin()
  ],
  mode: 'production'
  // devtool: 'inline-source-map'
}
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-webpack-plugin')

/*
  tree shaking：去除无用代码
    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
```

```
158
159     在package.json中配置
160       "sideEffects": false  所有代码都没有副作用（都可以进行tree shaking)
161         问题：可能会把css / @babel/polyfill（副作用）文件干掉
162       "sideEffects": [*.css]
163 */
164
165 // 定义nodejs环境变量：决定使用browserslist的哪个环境
166 process.env.NODE_ENV = 'production'
167
168 // 复用loader
169 const commonCssloader = [
170   MiniCssExtractPlugin.loader,
171   'css-loader',
172   {
173     // css兼容性处理
174     // 还需要再 package.json中定义browserslist
175     loader: 'postcss-loader',
176     options: {
177       postcssOptions: {
178         plugins: [require('postcss-preset-env')()]
179       }
180     }
181   }
182 ]
183
184 module.exports = {
185   entry: './src/js/index.js',
186   output: {
187     filename: 'js/built.[contenthash:10].js',
188     path: resolve(__dirname, 'build')
189   },
190   module: {
191     rules: [
192       {
193         // 在package.json中设置eslintConfig  --> airbnb
194         test: /\.js$/,
195         exclude: /node_modules/,
196         // 优先执行
197         enforce: 'pre',
198         loader: 'eslint-loader', // eslint 语法检查
199         options: {
200           fix: true
201         }
202       },
203       {
204         // 以下loader只会匹配一个，解决每个文件都会被全部loader过一遍（相当于找到直接
    break掉）
205         // 不能有两个配置处理同一类型的文件
206         oneOf: [
207           {
208             test: /\.css$/,
209             use: [...commonCssloader]
210           },
211           {
212             test: /\.less$/,
213             use: [...commonCssloader, 'less-loader']
214           },
```

```
          /*
          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
        */

          {
            // js兼容性处理
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel-loader',
            options: {
              presets: [
                [
                  '@babel/preset-env',
                  {
                    useBuiltIns: 'usage',
                    corejs: {
                      version: 3
                    },
                    targets: {
                      chrome: '60',
                      firefox: '50',
                      ie: '9',
                      safari: '10'
                    }
                  }
                ]
              ],
              // 开启babel缓存，第二次构建时，会读取之前的缓存
              cacheDirectory: true
            }
          },
          {
            // 处理图片资源
            test: /\.(jpg|pnconst { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-webpack-plugin')

/*
  tree shaking：去除无用代码
    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中配置
      "sideEffects": false  所有代码都没有副作用（都可以进行tree shaking)
        问题：可能会把css / @babel/polyfill（副作用）文件干掉
      "sideEffects": [*.css]
*/

// 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',
  {
```

```
    // css兼容性处理
    // 还需要再 package.json中定义browserslist
    loader: 'postcss-loader',
    options: {
      postcssOptions: {
        plugins: [require('postcss-preset-env')()]
      }
    }
  }
]

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.[contenthash:10].js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        // 在package.json中设置eslintConfig  --> airbnb
        test: /\.js$/,
        exclude: /node_modules/,
        // 优先执行
        enforce: 'pre',
        loader: 'eslint-loader', // eslint 语法检查
        options: {
          fix: true
        }
      },
      {
        // 以下loader只会匹配一个，解决每个文件都会被全部loader过一遍（相当于找到直接break掉）
        // 不能有两个配置处理同一类型的文件
        oneOf: [
          {
            test: /\.css$/,
            use: [...commonCssloader]
          },
          {
            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
      */

          {
            // js兼容性处理
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel-loader',
            options: {
              presets: [
                [
                  '@babel/preset-env',
                  {
```

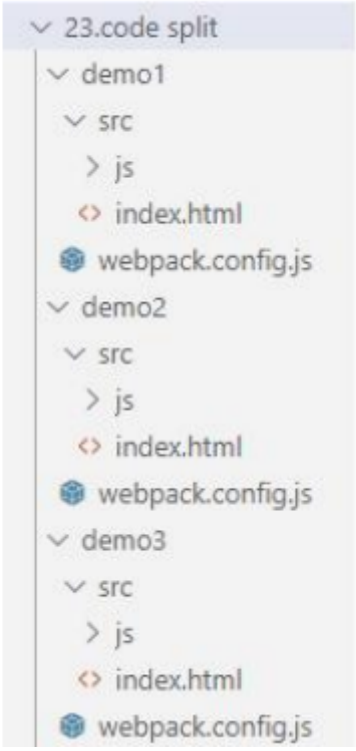
```
                    useBuiltIns: 'usage',
                    corejs: {
                      version: 3
                    },
                    targets: {
                      chrome: '60',
                      firefox: '50',
                      ie: '9',
                      safari: '10'
                    }
                  }
                ]
              ],
              // 开启babel缓存，第二次构建时，会读取之前的缓存
              cacheDirectory: true
            }
          },
          {
            // 处理图片资源
            test: /\.(jpg|png|gif|jpeg)$/i,
            type: 'asset',
            generator: {
              filename: '[name]_[contenthash:6][ext]'
            },
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024
              }
            }
          },
          {
            // 处理html 中的图片资源
            test: /\.html$/,
            loader: 'html-loader'
          },
          {
            // 处理其他资源
            exclude: /\.(js|css|less|html|jpg|png|gif|jpeg)$/,
            type: 'asset/resource',
            generator: {
              filename: 'media/[name]_[contenthash:6][ext]'
            }
          }
        ]
      }
    ]
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    }),
    // 提取css成单独文件
```

```
    new MiniCssExtractPlugin({
      filename: 'css/built.[contenthash:10].css'
    }),
    // 压缩 css
    new CssMinimizerWebpackPlugin()
  ],
  mode: 'production'
  // devtool: 'inline-source-map'
}
g|gif|jpeg)$/i,
            type: 'asset',
            generator: {
              filename: '[name]_[contenthash:6][ext]'
            },
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024
              }
            }
          },
          {
            // 处理html 中的图片资源
            test: /\.html$/,
            loader: 'html-loader'
          },
          {
            // 处理其他资源
            exclude: /\.(js|css|less|html|jpg|png|gif|jpeg)$/,
            type: 'asset/resource',
            generator: {
              filename: 'media/[name]_[contenthash:6][ext]'
            }
          }
        ]
      }
    ]
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      filename: 'css/built.[contenthash:10].css'
    }),
    // 压缩 css
    new CssMinimizerWebpackPlugin()
  ],
  mode: 'production'
  // devtool: 'inline-source-map'
}

```

3. 运行指令：`webpack`

5.7. code split（代码分割）

1. 创建文件



2. 修改配置文件
   - 修改 demo1 配置文件

```
const { resolve } = require('path')
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'

module.exports = {
  // 单入口
  // entry: './src/js/index.js',
  // 多入口：有一个入口，最终输出就有一个bundle
  entry: {
    index: './src/js/index.js',
    test: './src/js/test.js'
  },
  output: {
    // [name]：取文件名
    filename: 'js/[name].[contenthash:10].js',
    path: resolve(__dirname, 'build')
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
```

```
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    })
  ],
  mode: 'production'
}
```

- 修改 demo2 配置文件（多入口模式最终方案）

```
const { resolve } = require('path')
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'

module.exports = {
  // 单入口
  entry: './src/js/index.js',
  // entry: {
  //  index: './src/js/index.js',
  //  test: './src/js/test.js'
  // },
  output: {
    // [name]：取文件名
    filename: 'js/[name].[contenthash:10].js',
    path: resolve(__dirname, 'build')
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    })
  ],
  /*
    可以将node_module中代码单独打包成一个chunk最终输出
    自动分析多入口文件chunk中，有没有公共的文件（）。如果有会打包成单独一个chunk，不重复打包
  */
  optimization: {
    splitChunks: {
      chunks: 'all'
    }
  },
  mode: 'production'
}
```

- 修改 demo3 配置文件（单入口模式）
  - 通过 js 代码，让某个文件被单独打包成一个 chunk
  - 再结合 optimization 配置实现代码分割

```
/*
  通过js代码，让某个文件单独打包成一个chunk
  import动态导入语法：配合optimization配置，能将某个文件单独打包，实现代码分割
*/
import(/* webpackChunkName: 'test' */ './test')
  .then(({ mul, count }) => {
    // 文件加载成功
    // eslint-disable-next-line
    console.log(mul(2, 5))
  })
  .catch(() => {
    // eslint-disable-next-line
    console.log('文件加载失败')
  })

```

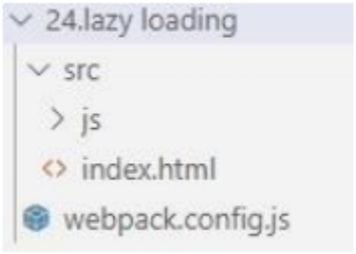
```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

/*
  通过js代码，让某个文件单独打包成一个chunk
  import动态导入语法：配合optimization配置，能将某个文件单独打包，实现代码分割
*/

// 定义nodejs环境变量：决定使用browserslist的哪个环境
process.env.NODE_ENV = 'production'

module.exports = {
  // 单入口
  entry: './src/js/index.js',
  output: {
    // [name]：取文件名
    filename: 'js/[name].[contenthash:10].js',
    path: resolve(__dirname, 'build'),
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true,
      },
    }),
  ],
  /*
    可以将node_module中代码单独打包成一个chunk最终输出
    自动分析多入口文件chunk中，有没有公共的文件。如果有会打包成单独一个chunk，不重复打包
  */
  optimization: {
    splitChunks: {
      chunks: 'all',
    },
```

```
  },
  mode: 'production',
}
```

3. 运行指令：`webpack`

5.8. lazy loading（懒加载）

1. 创建文件



2. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
   // 单入口
  entry: './src/js/index.js',
  output: {
    filename: 'js/[name].[contenthash:10].js',
    path: resolve(__dirname, 'build')
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html',
      minify: {
        collapseWhitespace: true,
        removeComents: true
      }
    })
  ],
  optimization: {
    spliChunks: {
      chunks: 'all'
    }
  },
  mode: 'production'
}
```

3. index.js（懒加载和预加载代码）

```
console.log('index.js文件被加载了~')

import { mul } from './test'

document.getElementById('btn').onclick = function () {
  // console.log(mul(4, 5)) // mul 是 test 模块中的一个乘法计算的函数方法
```

```
  // 懒加载：当文件需要使用时才加载
  // import(/* webpackChunkName: 'test' */'./test').then(({mul}) => {
  //   console.log(mul(4, 5))
  // })

  // 预加载 prefetch  --> webpackPrefetch: true
  //     会在使用之前，提前加载 js 文件
  //     正常加载可以认为是并行加载（同一时间加载多个文件  预加载 prefetch：等其他资源加载完毕，浏览器空闲了，再偷偷加载
  //
  import(/* webpackChunkName: 'test', webpackPrefetch: true */'./test').then(({mul}) => {
    console.log(mul(4, 5))
  })
}
```

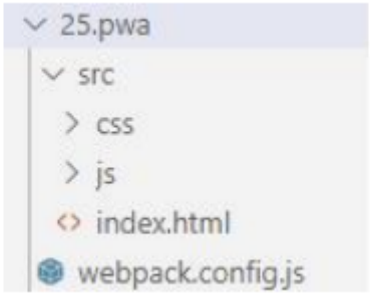
4. 运行指令：`webpack`

5. 懒加载和预加载的区别：（正常加载可以认为是并行加载）
   - 懒加载：当文件需要使用时才加载
   - 预加载（兼容性较差）：会在使用之前，提前加载 js 文件，等其他资源加载完毕，浏览器空闲了，再偷偷加载

5.9. PWA（离线加载）

- PWA：渐进式网络开发应用程序（离线加载）

1. 创建文件



2. 下载安装包

```
npm install wordbox-webpack-plugin -D
```

3. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-webpack-plugin')
const WorkboxWebpackPlugin = require('workbox-webpack-plugin')

/*
  PWA：渐进式网络开发应用程序（离线可访问）
    workbox --> workbox-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',
  {
    // css兼容性处理
    // 还需要再 package.json中定义browserslist
    loader: 'postcss-loader',
    options: {
      postcssOptions: {
        plugins: [require('postcss-preset-env')()]
      }
    }
  }
]

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.[contenthash:10].js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        // 在package.json中设置eslintConfig  --> airbnb
        test: /\.js$/,
        exclude: /node_modules/,
        // 优先执行
        enforce: 'pre',
        loader: 'eslint-loader', // eslint 语法检查
        options: {
          fix: true
        }
      },
      {
        // 以下loader只会匹配一个，解决每个文件都会被全部loader过一遍（相当于找到直接break掉）
        // 不能有两个配置处理同一类型的文件
        oneOf: [
          {
            test: /\.css$/,
            use: [...commonCssloader]
          },
          {
            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
      */

          {
            // js兼容性处理
```

```
            test: /\.js$/,
            exclude: /node_modules/,
            loader: 'babel-loader',
            options: {
              presets: [
                [
                  '@babel/preset-env',
                  {
                    useBuiltIns: 'usage',
                    corejs: {
                      version: 3
                    },
                    targets: {
                      chrome: '60',
                      firefox: '50',
                      ie: '9',
                      safari: '10'
                    }
                  }
                ]
              ],
              // 开启babel缓存，第二次构建时，会读取之前的缓存
              cacheDirectory: true
            }
          },
          {
            // 处理图片资源
            test: /\.(jpg|png|gif|jpeg)$/i,
            type: 'asset',
            generator: {
              filename: '[name]_[contenthash:6][ext]'
            },
            parser: {
              dataUrlCondition: {
                maxSize: 10 * 1024
              }
            }
          },
          {
            // 处理html 中的图片资源
            test: /\.html$/,
            loader: 'html-loader'
          },
          {
            // 处理其他资源
            exclude: /\.(js|css|less|html|jpg|png|gif|jpeg)$/,
            type: 'asset/resource',
            generator: {
              filename: 'media/[name]_[contenthash:6][ext]'
            }
          }
        ]
      }
    ]
  },
  plugins: [
    // 打包html资源
    new HtmlWebpackPlugin({
```

```
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true
      }
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      filename: 'css/built.[contenthash:10].css'
    }),
    // 压缩 css
    new CssMinimizerWebpackPlugin(),
    new WorkboxWebpackPlugin.GenerateSW({
      /*
        1. 帮助serviceworker快速启动
        2. 删除旧的 serviceworker

        生成一个 serviceworker 配置文件，并在入口文件中注册serviceworker
      */
      clientsClaim: true,
      skipWaiting: true
    })
  ],
  mode: 'production'
  // devtool: 'inline-source-map'
}

```

4. 入口 index.js 中

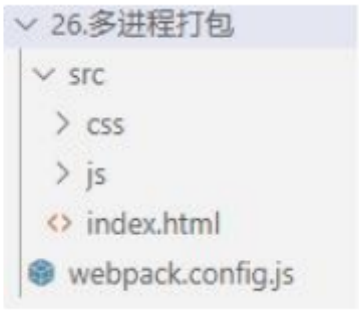
```
impor { mul } from './test'
import '../css/index.css'

function sum(...args) {
  return args.reduce((p, c) => p + c, 0)
}



/*
  1. eslint不认识 window、navigator 全局变量
    解决：需要修改 package.json 中 eslintConfig 配置
      "env": {
        "browser": true // 支持浏览器端全局变量，如果要支持node则改成"node": true
      }
  2. serviceworker(SW) 代码必须运行在服务器上
    --> nodejs
    -->
      npm i server -g
      server -s build 启动服务器，将 build 目录下所有资源作为静态资源暴露出去
*/
// 注册 serviceworker
// 处理兼容性问题
if ('serviceWorker' in navigator) {
  window.addEventListener('load', () => {
    navigator.serviceWorker.register('/service-worker.js')
      .then(() => {
```

```
        console.log('SW注册成功了')
      })
      .catch(() => {
        console.log('SW注册失败了')
      })
  })
}
```

5. 运行指令：`webpack`

5.10. 多进程打包

1. 创建文件



2. 下载安装包

```
npm install thread-loader -D
```

3. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const MiniCssExtractPlugin = require('mini-css-extract-plugin')
const CssMinimizerWebpackPlugin = require('css-minimizer-webpack-plugin')
const WorkboxWebpackPlugin = require('workbox-webpack-plugin')

/*
  PWA：渐进式网络开发应用程序（离线可访问）
    workbox --> workbox-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',
  {
    // css兼容性处理
    // 还需要再 package.json中定义browserslist
    loader: 'postcss-loader',
    options: {
      postcssOptions: {
        plugins: [require('postcss-preset-env')()],
```

```
      },
    },
  },
]

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/built.[contenthash:10].js',
    path: resolve(__dirname, 'build'),
  },
  module: {
    rules: [
      {
        // 在package.json中设置eslintConfig  --> airbnb
        test: /\.js$/,
        exclude: /node_modules/,
        // 优先执行
        enforce: 'pre',
        loader: 'eslint-loader', // eslint 语法检查
        options: {
          fix: true,
        },
      },
      {
        // 以下loader只会匹配一个，解决每个文件都会被全部loader过一遍（相当于找到直接break掉）
        // 不能有两个配置处理同一类型的文件
        oneOf: [
          {
            test: /\.css$/,
            use: [...commonCssloader],
          },
          {
            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
      */

          {
            // js兼容性处理
            test: /\.js$/,
            exclude: /node_modules/,
            use: [
              /*
                开启多进程打包
                进程启动大概为600ms，进程通信也有开销。
                只有工作消耗时间比较长，才需要多进程打包
              */
              {
                loader: 'thread-loader',
                options: {
                  workers: 2 // 进程2个
                }
              },
```

```
            {
              loader: 'babel-loader',
              options: {
                presets: [
                  [
                    '@babel/preset-env',
                    {
                      useBuiltIns: 'usage',
                      corejs: {
                        version: 3,
                      },
                      targets: {
                        chrome: '60',
                        firefox: '50',
                        ie: '9',
                        safari: '10',
                      },
                    },
                  ],
                ],
                // 开启babel缓存，第二次构建时，会读取之前的缓存
                cacheDirectory: true,
              },
            },
          ],
        },
        {
          // 处理图片资源
          test: /\.(jpg|png|gif|jpeg)$/i,
          type: 'asset',
          generator: {
            filename: '[name]_[contenthash:6][ext]',
          },
          parser: {
            dataUrlCondition: {
              maxSize: 10 * 1024,
            },
          },
        },
        {
          // 处理html 中的图片资源
          test: /\.html$/,
          loader: 'html-loader',
        },
        {
          // 处理其他资源
          exclude: /\.(js|css|less|html|jpg|png|gif|jpeg)$/,
          type: 'asset/resource',
          generator: {
            filename: 'media/[name]_[contenthash:6][ext]',
          },
        },
      ],
    },
  ],
},
plugins: [
  // 打包html资源
```

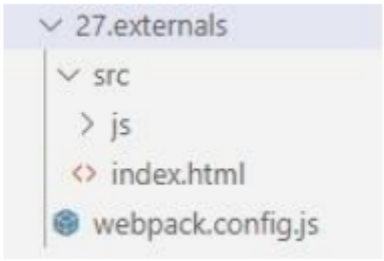
```
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html
      minify: {
        collapseWhitespace: true,
        removeComments: true,
      },
    }),
    // 提取css成单独文件
    new MiniCssExtractPlugin({
      filename: 'css/built.[contenthash:10].css',
    }),
    // 压缩 css
    new CssMinimizerWebpackPlugin(),
    new WorkboxWebpackPlugin.GenerateSW({
      /*
        1. 帮助serviceworker快速启动
        2. 删除旧的 serviceworker

        生成一个 serviceworker 配置文件，并在入口文件中注册serviceworker
      */
      clientsClaim: true,
      skipWaiting: true,
    }),
  ],
  mode: 'production',
  // devtool: 'inline-source-map'
}

```

4. 运行指令：`webpack`

5.11. externals（防止某些包打包至bundle中）

- 规定某些包，不打包进来，而是通过链接引入
- 在需要用到的地方需要引入（index.html 中通过链接引入）

1. 创建文件



2. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'js/built.js',
```

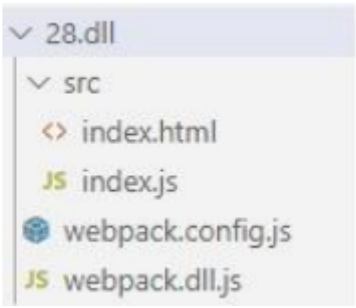
```
    path: resolve(__dirname, 'build')
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: './src/index.html'
    })
  ],
  mode: 'production',
  externals: {
    // 拒绝 jQuery 被打包进来
    // 忽略库名:  'npm包名'
    jquery: 'jQuery'
  }
}
```

3. 运行指令：`webpack`

5.12. dll（动态链接库）

说明：dll 类似 externals 会指示 webpack 哪些库不参与打包，不同的是 dll 会单独的对某些库进行单独打包，将多个库打包成一个 chunk，dll 需要打包一次，之后就不需要打包了，externals 是始终不打包

1. 创建文件



2. 写 webpack.dll.js 文件

```
/*
  使用dll技术，对某些库（第三方库：jQuery、react、vue）进行单独打包
    当你运行 webpack 时，默认查找 webpack.config.js 配置文件
    需求：需要运行 webpack.dll.js 文件
      --> webpack --config webpack.dll.js
*/
const { resolve } = require('path')
const webpack = require('webpack')

module.exports = {
  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]', // 打包的库向外暴露出去的内容叫什么名字
  },
  plugins: [
```

```
    // 打包生成一个 manifest.json --> 提供jquery映射
    new webpack.DllPlugin({
      name: '[name]_[hash]', // 映射库的暴露的内容名称
      path: resolve(__dirname, 'dll/manifest.json'), // 输出文件路径
    }),
  ],
  mode: 'production',
}
```

3. 运行指令：`webpack --config webpack.dll.js` --> 打包指定库并生成 dll/manifest.json
4. 修改配置文件

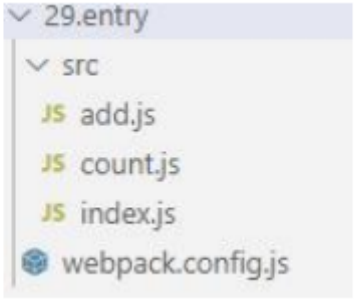
```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const webpack = require('webpack')
const AddAssetHtmlWebpackPlugin = require('add-asset-Html-webpack-plugin')

module.exports = {
  entry: './src/index.js',
  output: {
    filename: './src/index.js',
    path: resolve(__dirname, 'build')
  },
  plugins: [
    new HtmlWebpackPlugin({
      tempalte: './src/index.html'
    }),
    // 告诉 webpack 哪些库不参与打包，同时使用时的名称也得变~
    new webpack.DllReferencePlugin({
      manifest: resolve(__dirname, 'dll/manifest.json')
    })
    // 将某个文件打包输出出去，并在 html 中自动引入该资源
    new AddAssetHtmlWebpackPlugin({
      filepath: resolve(__dirname, 'dll/jquery.js')
    })
    mode: 'production'
  ]
}
```

5. 运行指令：`webpack`

6. 第六章：webpack 配置详情

6.1. entry

1. 创建文件



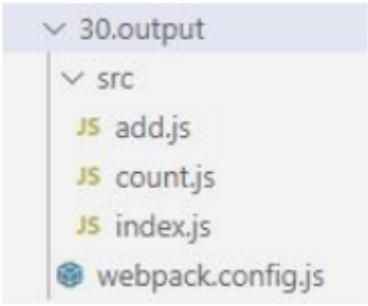
2. 修改配置文件

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
const { resolve } = require('path')

/*
  entry：入口起点
    1. string --> './src/index.js'
      单入口
      打包形成一个chunk。输出一个bundle文件
      此时chunk的名称默认是 main
    2. array --> ['./src/index.js', './src/add.js']
      多入口
      所有入口文件最终只会形成一个chunk，输出出去只有一个bundle
        --> 只有在HMR功能中让html热更新生效才使用
    3. object --> {key: value, key: value}
      多入口
      有几个文件就形成几个chunk，输出几个bundle
      此时chunk名称是key

      --> 特殊用法(dll中用到)：
        entry: {
          所有入口文件最终只会形成一个chunk，输出出去只有一个bundle
          // 把'./src/index.js', './src/count.js'打包到index.js
中，'./src/add.js'单独打包
          index: ['./src/index.js', './src/count.js'],
          // 形成一个chunk，输出一个bundle
          add: './src/add.js'
        },
*/

module.exports = {
  entry: {
    index: './src/index.js',
    add: './src/add.js',
  },
  output: {
    filename: '[name].js',
    path: resolve(__dirname, 'build'),
  },
  module: {
    rules: [],
  },
  plugins: [new HtmlWebpackPlugin()],
  mode: 'development',
}

```

3. 运行指令：`webpack`

6.2. output

1. 创建文件



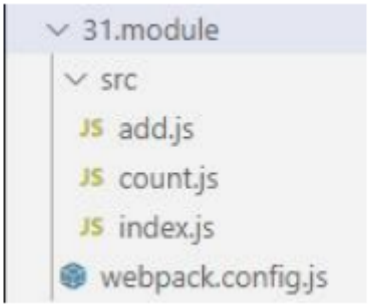
2. 修改配置文件

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
const { resolve } = require('path')

module.exports = {
  entry: './src/index.js',
  output: {
    // 文件名称（指定目录+名称）
    filename: 'js/[name].js',
    // 输出文件目录（将来所有资源输出的公共目录）
    path: resolve(__dirname, 'build'),
    environment: {
      // 告诉webpack不使用箭头函数
      arrowFuntion: false,
      // 告诉 webpack 不使用 const
      const: false
    },
    // 所有资源引入公共路径前缀 --> 'imgs/a.jpg' --> '/imgs/a.jpg'
    publicPath: '/',
    // 非入口chunk的名称
    chunkFilename: '[name]_chunk.js',
    // 整个库向外暴露的变量名，结合dll使用
    library: '[name]',
    // libraryTarget: 'window', // 变量名添加到哪个上 browser
    // libraryTarget: 'global', // 变量名添加到哪个上 node
    libraryTarget: 'commonjs',
  },
  module: {
    rules: [],
  },
  plugins: [new HtmlWebpackPlugin()],
  mode: 'development',
}
```

3. 运行指令：`webpack`

6.3. module

1. 创建文件



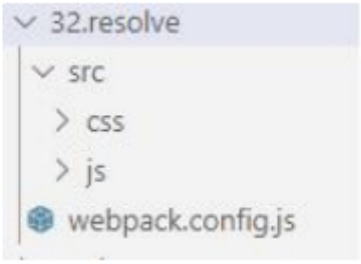
2. 修改配置文件

```
const HtmlWebpackPlugin = require('html-webpack-plugin')
const { resolve } = require('path')

module.exports = {
  entry: './src/index.js',
  output: {
    filename: 'js/[name].js',
    path: resolve(__dirname, 'build'),
  },
  module: {
    rules: [
      // loader 的配置
      {
        test: /\.css$/,
        // 多个loader用use
        use: ['style-loader', 'css-loader'],
      },
      {
        test: /\.js$/,
        // 排除node_modules下的js文件
        exclude: /node_modules/,
        // 只检查src下的js文件
        include: resolve(__dirname, 'src'),
        // 优先执行
        enforce: 'pre',
        // 延后执行
        enforce: 'post',
        // 单个loader用loader
        loader: 'eslint-loader',
        // loader的配置选项
        options: {}
      },
      {
        // 以下配置只会生效一个
        oneOf: [],
      },
    ],
  },
  plugins: [new HtmlWebpackPlugin()],
  mode: 'development',
}
```

3. 运行指令：`webpack`

6.4. resolve

1. 创建文件



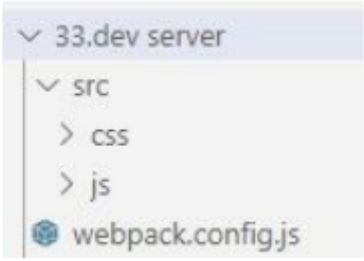
2. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/[name].js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      }
    ]
  },
  plugins: [new HtmlWebpackPlugin()],
  mode: 'development',
  // 解析模块的规则
  resolve: {
    // 配置解析模块路径别名：优点是简写路径，缺点是路径没有提示
    alias: {
      $css: resolve(__dirname, 'src/css')
    },
    // 配置省略文件路径的后缀名
    extensions: ['.js', 'json', '.jsx'],
    // 告诉 webpack 解析模块去找哪个目录
    modules: [resolve(__dirname, '../../node_modules'), 'node_modules']
  }
}
```

3. 运行指令：`webpack`

6.5. devServer

1. 创建文件



2. 修改配置文件

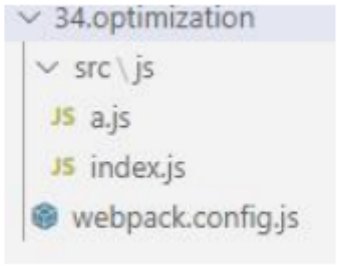
```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/[name].js',
    path: resolve(__dirname, 'build')
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      }
    ]
  },
  plugins: [new HtmlWebpackPlugin()],
  mode: 'development',
  // 解析模块的规则
  resolve: {
    // 配置解析模块路径别名：优点是简写路径，缺点是路径没有提示
    alias: {
      $css: resolve(__dirname, 'src/css')
    },
    // 配置省略文件路径的后缀名
    extensions: ['.js', 'json', '.jsx', '.css'],
    // 告诉 webpack 解析模块去找哪个目录
    modules: [resolve(__dirname, '../../node_modules'), 'node_modules']
  },
  devServer: {
    // 运行代码的目录
    contentBase: resolve(__dirname, 'build'),
    // 监视 contentBase 目录下的所有文件，一旦文件变化就会 reload
    watchContentBase: true,
    watchOptions: {
      // 忽略文件
      ignored: /node_modules/
    },
    // 启动 gzip 压缩
    compress: true,
    // 端口号
    port: 5000,
```

```
    // 域名
    host: 'localhost',
    // 自动打开浏览器
    open： true,
    // 自动开启 HMR 功能
    hot: true,
    // 不要显示启动服务器日志信息
    clientLogLever: 'none',
    // 除了一些基本启动信息以外，其他内容都不要提示
    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(5000) 服务器接收到 /api/xxx 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（去掉/api）
        pathRewrite: {
          '^/api': ''
        }
      }
    }
  }
}
```

3. 运行指令：`webapck`

6.6. optimization

1. 创建文件



2. 修改配置文件

```
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin')
// 版本到4.26以上之后，js的压缩不再用UglifyPlugin，而是用 TerserWebpackPlugin压缩
const TerserWebpackPlugin = require('terser-webpack-plugin')

module.exports = {
  entry: './src/js/index.js',
  output: {
    filename: 'js/[name].[contenthash:10].js',
    path: resolve(__dirname, 'build'),
    chunkFilename: 'js/[name].[contenthash:10]_chunk.js'
  },
```

```
  module: {
    rules: [
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader']
      }
    ]
  },
  plugins: [new HtmlWebpackPlugin()],
  mode: 'production',
  resolve: {
    alias: {
      $css: resolve(__dirname, 'src/css')
    },
    extensions: ['.js', '.json', '.jsx', '.css'],
    modules: [resolve(__dirname, '../../node_modules'), 'node_modules']
  },
  optimization: {
    splitChunks: {
      chunks: 'all'
      // 默认值，可以不写~
      /* minSize: 30 * 1024, // 分割的chunk最小为30kb
      maxSize: 0, // 最大没有限制
      minChunks: 1, // 要提取的chunk最少被引用1次
      maxAsyncRequest: 5, // 按需加载时并行加载的文件的最大数量
      maxInitialRequest: 3, // 入口js文件最大并行请求数量
      automaticNameDlimiter: '~', // 名称连接符
      name: true, // 可以使用命名规则
      cacheGroup: { // 分割chunk 的组
          // node_modules 中的文件会被打包到 ventors 组的 chunk 中。 --
>ventors~xxx.js
          // 满足上面规则，如：大小超过 30kb，至少被引用一次。
          vendors:  {
            test: /[\\/]node_modules[\\/]/,
            // 优先级
            priority: -10
          },
          default: {
            minChunks: 2, // 要提取的chunk最少被引用2次
            // 优先级
            priority: -20,
            // 如果当前要打包的模块，和之前已经被提取的模块时同一个，就会服用，而不是重新
打包模块
            reuseExistingChunk: true
          }
       } */
    },
    // 将当前模块记录其他模块的 hash 单独打包为一个文件 runtime
    // 解决：修改 a 文件导致 b 文件的 contenthash 变化
    runtimeChunk:  {
      name: entrypoint => `runtime-${entrypoint.name}`
    },
    minimizer: [
      // 配置生产环境的压缩方案：js 和 css
      new TerserWebpackPlugin({
        // 开启缓存
        cache: true,
        // 开启多进程打包
```

```
        parallel: true,
        // 启动 source-map
        sourceMap: true
      })
    ]
  }
}
```

7. 第七章：webpack5 介绍和使用

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（tree shaking）和代码生成来改善捆绑包大小；
- 清除怪异状态的内部解构，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使用 v5.

7.1. 下载

```
npm i webpack@next webpack-cli -D
```

7.2. 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，一旦模块使用了任何核心模块（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 使不必要的。

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。错误消息提示如何实现该目标。

7.3. 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: "deteministic"
```

7.4. Chunk ID

你可以不使用 `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

7.5. 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

2. webpack 现在能够处理好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"`

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

7.6. Output

webpack 4 默认只能输出 ES5 代码

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

如：`output.ecmaVersion: 2015`

7.7. SplitChunk

```
// webpack 4
minSize: 30000;
```

```
// webpack 5
minSize: {
  javascript: 30000,
    style: 50000,
}
```

7.8. Caching

```
// 配置缓存
cache: {
  // 磁盘缓存
  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`

7.9. 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。

7.10. 默认值

- `entry: './src/index.js'`
- `output.path: path.resolve(__dirname, 'dist')`
- `output.filename: '[name].js'`

7.11. 更多内容

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

7.12. webpack 5 处理图片文件

```
// 含转换base64的功能  "asset",  （等价于 file + url 两个loader）
{
  test: test: /\.(jpe?g|png|gif|svg)$/,
    type: 'asset',
    generator: {
      filename: 'img/[name]_[hash].[ext]',
    },
    parser: {
      dataUrlCondition: {
        maxSize: 100 * 1024  //(表示100kb以下的文件转换成base64编码)
      }
    }
 }
```

7.13. 打包项目中的静态资源

资源模块类型(asset module type)，通过添加 4 种新的模块类型，来替换之前使用的一些 loader：

- `asset/resource` 发送一个单独的文件并导出 URL。之前通过使用 `file-loader` 实现。
- `asset/inline` 导出一个资源的 data URI(base64格式)。之前通过使用 `url-loader` 实现。
- `asset/source` 导出资源的源代码。之前通过使用 `raw-loader` 实现。
- `asset` 在导出一个 data URI 和发送一个单独的文件之间自动选择。之前通过使用 `url-loader`，并且配置资源体积限制实现。

7.13.1. asset/resource

- 配置loader

```
{
  test: /\.(png|jpe?g|gif)$/,
  type: "asset/resource",
}
```

- 使用

```
// index.js
import reactUrl from '../imgs/react.jpg'
document.querySelector('.react').src = reactUrl
```

- 让图片输出到指定目录

```
output: {
    ...,
  assetModuleFilename: "images/[hash:8][ext]",
}
或:
{
  test: /\.(png|jpe?g|gif|svg)$/,
  type: "asset/resource",
  // generator只适用于asset和asset/resource
  generator: {
     filename: 'imgs/[name][hash][ext]',
  },
}
```

- 运行指令：`webpack`

7.13.2. asset/inline

- 配置 loader

```
{
  test: /\.svg$/,
  type: 'asset/inline'
}
```

- 使用
- 使用

```
// index.js
import wexin from '../imgs/微信.svg'
document.querySelector('.wexin').src = wexin
```

- 运行指令：`webpack`

7.13.3. asset/source

- 配置 loader

```
{
  test: /\.txt/,
  type: 'asset/source',
}
```

- 使用

```
//hello.txt
  hello中的文本

// index.js
import text from '../assets/hello.txt' //text就是hello.txt的内容
document.querySelector('#txt').innerText = text
```

- 运行指令：`webpack`

7.13.4. asset/source

在asset/resource和asset/inline之间自动选择

- 配置 loader

```
{
  test: /\.(png|jpe?g|gif|svg)$/,
  type: "asset",
  parser: {
     dataUrlCondition: {
         maxSize: 3 * 1024, // 小于3kb以下的图片会被打包成base64格式
      },
  },
}
```

- 使用

```
// index.js
import reactUrl from '../imgs/react.jpg'
document.querySelector('.react').src = reactUrl
```

- 让图片输出到指定目录

```
output: {
    ...,
  assetModuleFilename: "images/[hash:8][ext]",
}
或:
{
  test: /\.(png|jpe?g|gif|svg)$/,
  type: "asset",
  // generator只适用于asset和asset/resource
  generator: {
     filename: 'imgs/[name][hash][ext]',
  },
}
```

- 运行指令：`webpack`