Vue.js - Webpack

在网页中会引用哪些常见的静态资源?

- 图片 .jpg .png .gif .svg
- 字体图标 .ttf .woff .woff2 .svg .eot
- JS .js .jsx .ts
- CSS .css .less .scss
- 模板文件 .html .art .vue

网页中引入的静态资源多了以后有什么问题？？？

1. 网页加载速度慢， 因为 我们要发起很多的二次请求；
2. 要处理错综复杂的依赖关系

如何解决上述两个问题

1. 合并、压缩、精灵图、图片的Base64编码
2. 可以使用之前学过的requireJS、也可以使用webpack可以解决各个包之间的复杂依赖关系；

1.1 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，打包生成对应的静态资源(bundle)。

1.2 webpack 五个核心概念

1.2.1 Entry

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

1.2.2 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。

1.2.3 Loader

Loader：让 webpack 能够去处理那些非 JS 的文件，比如样式文件、图片文件(webpack 自身只理解 JS)

1.2.4 Plugins

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

1.2.5 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 process.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 process.env.NODE_ENV 的值设置为 production。启用 FlagDependencyUsagePlugin, FlagIncludedChunksPlugin, ModuleConcatenationPlugin, NoEmitOnErrorsPlugin, OccurrenceOrderPlugin, SideEffectsFlagPlugin 和 TerserPlugin。	能让代码优化上线运行的环境

二、Webpack 初体验

`nrm`的安装使用

作用：提供了一些最常用的NPM包镜像地址，能够让我们快速的切换安装包时候的服务器地址；
什么是镜像：原来包刚一开始是只存在于国外的NPM服务器，但是由于网络原因，经常访问不到，这时候，我们可以在国内，创建一个和官网完全一样的NPM服务器，只不过，数据都是从人家那里拿过来的，除此之外，使用方式完全一样；

1. 运行`npm i nrm -g`全局安装`nrm`包；
2. 使用`nrm ls`查看当前所有可用的镜像源地址以及当前所使用的镜像源地址；
3. 使用`nrm use npm`或`nrm use taobao`切换不同的镜像源地址；
4. 查看镜像地址：npm config get registry

2.1 初始化配置

1. 初始化 package.json：npm init
2. 下载安装webpack：(webpack4以上的版本需要全局/本地都安装webpack-cli)

   全局安装：npm i webpack webpack-cli -g

   本地安装：npm i webpack webpack-cli -D

2.2 编译打包应用

创建 src 下的 js 等文件后，不需要配置 webpack.config.js 文件，在命令行就可以编译打包。

指令：

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

  webpack会以 ./src/index.js 为入口文件开始打包，打包后输出到 ./build 整体打包环境，是开发环境

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

  webpack会以 ./src/index.js 为入口文件开始打包，打包后输出到 ./build 整体打包环境，是生产环境

结论：

1. webpack 本身能处理 js 资源
2. 生产环境和开发环境将 ES6 模块化编译成浏览器能识别的模块化，但是不能处理 ES6 的基本语法转化为 ES5（需要借助 loader）
3. 生产环境比开发环境多一个压缩 js 代码

三、Webpack 开发环境的基本配置

webpack.config.js 是 webpack 的配置文件。

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

所有构建工具都是基于 nodejs 平台运行的，模块化默认采用 commonjs。

开发环境配置主要是为了能让代码运行。主要考虑以下几个方面：

- 打包样式资源
- 打包 html 资源
- 打包图片资源
- 打包其他资源
- devServer

下面是一个简单的开发环境webpack.confg.js配置文件

```
// resolve用来拼接绝对路径的方法
const { resolve } = require('path')
const HtmlWebpackPlugin = require('html-webpack-plugin') // 引用plugin

module.exports = {
  // webpack配置
  entry: './src/js/index.js', // 入口起点
  output: {
    // 输出
    // 输出文件名
    filename: 'js/build.js',
    // __dirname是nodejs的变量，代表当前文件的目录绝对路径
    path: resolve(__dirname, 'build'), // 输出路径，所有资源打包都会输出到这个文件夹下
  },
  // loader配置
  module: {
    rules: [
      // 详细的loader配置
      // 不同文件必须配置不同loader处理
      {
        // 匹配哪些文件
        test: /\.less$/,
        // 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(先执行css-loader)
          // style-loader：创建style标签，将js中的样式资源插入进去，添加到head中生效
          'style-loader',
          // css-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',
          // less-loader：将less文件编译成css文件，需要下载less-loader和less
          'less-loader'
        ],
      },
      {
        test: /\.css$/,
        use: ['style-loader', 'css-loader'],
      },
      {
        // url-loader：处理图片资源，问题：默认处理不了html中的img图片
        test: /\.(jpg|png|gif)$/,
        // 需要下载 url-loader file-loader
        loader: 'url-loader',
        options: {
          // 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）
          // base64在客户端本地解码所以会减少服务器压力，如果图片过大还采用base64编码会导致cpu调用率上升，网页加载时变卡
          limit: 8 * 1024,
          // 给图片重命名，[hash:10]：取图片的hash的前10位，[ext]：取文件原来扩展名
          name: '[hash:10].[ext]',
          // 问题：因为url-loader默认使用es6模块化解析，而html-loader引入图片是conmonjs，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,
          outputPath: 'imgs',
        },
      },
      {
        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',
      },
      // 打包其他资源(除了html/js/css资源以外的资源)
      {
        // 排除html|js|css|less|jpg|png|gif文件
        exclude: /\.(html|js|css|less|jpg|png|gif)/,
        // file-loader：处理其他文件
        loader: 'file-loader',
        options: {
          name: '[hash:10].[ext]',
          outputPath: 'media',
        },
      },
    ],
  },
  // plugin的配置
  plugins: [
    // html-webpack-plugin：默认会创建一个空的html文件，自动引入打包输出的所有资源（JS/CSS）
    // 需要有结构的HTML文件可以加一个template
    new HtmlWebpackPlugin({
      // 复制这个./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',
    }),
  ],
  // 模式
```

```
  mode: 'development', // 开发模式
  // 开发服务器 devServer：用来自动化，不用每次修改后都重新输入webpack打包一遍（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（不会像之前那样在外面看到打包输出的build包，而是在内存中，关闭后会自动删除）
  // 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,
  },
}
```

其中，大部分配置都在注释中给出解释。

- 运行项目的两个指令：
  webpack 会将打包结果输出出去（build文件夹）
  npx webpack-dev-server 只会在内存中编译打包，没有输出
- loader 和 plugin 的不同：（plugin 一定要先引入才能使用）

  loader：1. 下载 2. 使用（配置 loader）

  plugins：1.下载 2. 引入 3. 使用

四、Webpack 生产环境的基本配置

而生产环境的配置需要考虑以下几个方面：

- 提取 css 成单独文件
- css 兼容性处理
- 压缩 css
- js 语法检查
- js 兼容性处理
- js 压缩
- html 压缩

下面是一个基本的生产环境下的webpack.config.js配置

```
const { resolve } = require('path')
const MiniCssExtractorPlugin = require('mini-css-extract-plugin')
const OptimiziCssAssetsWebpackPlugin = require('optimizi-css-assets-webpack-plugin')
const HtmlWebpackPlugin = require('html-webpack-plugin')

// 定义node.js的环境变量，决定使用browserslist的哪个环境
process.env.NODE_ENV = 'production'

// 复用loader的写法
const commonCssLoader = [
  // 这个loader取代style-loader。作用：提取js中的css成单独文件然后通过link加载
  MiniCssExtractPlugin.loader,
  // css-loader：将css文件整合到js文件中
```

```
  // 经过css-loader处理后，样式文件是在js文件中的
  // 问题：1.js文件体积会很大2.需要先加载js再动态创建style标签，样式渲染速度就慢，会出现闪屏现象
  // 解决：用MiniCssExtractPlugin.loader替代style-loader
  'css-loader',
  /*
    postcss-loader：css兼容性处理：postcss --> 需要安装：postcss-loader postcss-preset-env
    postcss需要通过package.json中browserslist里面的配置加载指定的css兼容性样式
    在package.json中定义browserslist：
    "browserslist": {
      // 开发环境 --> 设置node环境变量：process.env.NODE_ENV = development
      "development": [ // 只需要可以运行即可
        "last 1 chrome version",
        "last 1 firefox version",
        "last 1 safari version"
      ],
      // 生产环境。默认是生产环境
      "production": [ // 需要满足绝大多数浏览器的兼容
        ">0.2%",
        "not dead",
        "not op_mini all"
      ]
    },
  */
  {
    loader: 'postcss-loader',
    options: {
      ident: 'postcss', // 基本写法
      plugins: () => [
        // postcss的插件
        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（用enforce）
      */
```

```
      {
        /*
          js的语法检查： 需要下载 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
          airbnb(一个流行的js风格) --> 需要下载 eslint-config-airbnb-base eslint-
plugin-import
          设置检查规则：
            package.json中eslintConfig中设置
              "eslintConfig": {
                "extends": "airbnb-base",  // 继承airbnb的风格规范
                "env": {
                  "browser": true // 可以使用浏览器中的全局变量(使用window不会报错)
                }
              }
        */
        test: /\.js$/,
        exclude: /node_modules/, // 忽略node_modules
        enforce: 'pre', // 优先执行
        loader: 'eslint-loader',
        options: {
          // 自动修复
          fix: true,
        },
      },
      /*
        js兼容性处理：需要下载 babel-loader @babel/core
          1. 基本js兼容性处理 --> @babel/preset-env
            问题：只能转换基本语法，如promise高级语法不能转换
          2. 全部js兼容性处理 --> @babel/polyfill
            问题：只要解决部分兼容性问题，但是将所有兼容性代码全部引入，体积太大了
          3. 需要做兼容性处理的就做：按需加载  --> core-js
      */
      {
        // 第三种方式：按需加载
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        options: {
          // 预设：指示babel做怎样的兼容性处理
          presets: [
            '@babel/preset-env', // 基本预设
            {
              useBuiltIns: 'usage', //按需加载
              corejs: { version: 3 }, // 指定core-js版本
              targets: { // 指定兼容到什么版本的浏览器
                chrome: '60',
                firefox: '50',
                ie: '9',
                safari: '10',
                edge: '17'
              },
            },
          ],
        },
      },
      {
        // 图片处理
        test: /\.(jpg|png|gif)/,
```

```
        loader: 'url-loader',
        options: {
          limit: 8 * 1024,
          name: '[hash:10].[ext]',
          outputPath: 'imgs',
          esModule: false, // 关闭url-loader默认使用的es6模块化解析
        },
      },
      // html中的图片处理
      {
        test: /\.html$/,
        loader: 'html-loader',
      },
      // 处理其他文件
      {
        exclude: /\.(js|css|less|html|jpg|png|gif)/,
        loader: 'file-loader',
        options: {
          outputPath: 'media',
        },
      },
    ],
  },
  plugins: [
    new MiniCssExtractPlugin({
      // 对输出的css文件进行重命名
      filename: 'css/built.css',
    }),
    // 压缩css
    new OptimiziCssAssetsWebpackPlugin(),
    // HtmlWebpackPlugin：html文件的打包和压缩处理
    // 通过这个插件会自动将单独打包的样式文件通过link标签引入
    new HtmlWebpackPlugin({
      template: './src/index.html',
      // 压缩html代码
      minify: {
        // 移除空格
        collapseWhitespace: true,
        // 移除注释
        removeComments: true,
      },
    }),
  ],
  // 生产环境下会自动压缩js代码
  mode: 'production',
}
```

五、Webpack 配置详情

5.1 entry

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 文件。

（一般只用在 HMR 功能中让 html 热更新生效）

3. object，多入口

有几个入口文件就形成几个 chunk，输出几个 bundle 文件，此时 chunk 的名称是 key 值

--> 特殊用法：

```
entry: {
  // 最终只会形成一个chunk，输出出去只有一个bundle文件。
  index: ['./src/index.js', './src/count.js'],
  // 形成一个chunk，输出一个bundle文件。
  add: './src/add.js'
}
```

5.2 output

```
output: {
  // 文件名称（指定名称+目录）
  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'),
  // 所有资源引入公共路径前缀 --> 'imgs/a.jpg' --> '/imgs/a.jpg'
  publicPath: '/',
  chunkFilename: 'js/[name]_chunk.js', // 指定非入口chunk的名称
  library: '[name]', // 打包整个库后向外暴露的变量名
  libraryTarget: 'window' // 变量名添加到哪个上 browser：window
  // libraryTarget: 'global' // node：global
  // libraryTarget: 'commonjs' // conmmonjs模块 exports
},
```

5.3 module

```
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'),
      enforce: 'pre', // 优先执行
      // enforce: 'post', // 延后执行
      // 单个loader用loader
      loader: 'eslint-loader',
      options: {} // 指定配置选项
    },
    {
      // 以下配置只会生效一个
```

```
      oneOf: []
    }
  ]
},
```

5.4 resolve

```
// 解析模块的规则
resolve: {
  // 配置解析模块路径别名: 优点：当目录层级很复杂时，简写路径；缺点：路径不会提示
  alias: {
    $css: resolve(__dirname, 'src/css')
  },
  // 配置省略文件路径的后缀名（引入时就可以不写文件后缀名了）
  extensions: ['.js', '.json', '.jsx', '.css'],
  // 告诉 webpack 解析模块应该去找哪个目录
  modules: [resolve(__dirname, '../../node_modules'), 'node_modules']
}
```

这样配置后，引入文件就可以这样简写：`import '$css/index';`

5.5 dev server

```
devServer: {
  // 运行代码所在的目录
  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,
  // 开启HMR功能
  hot: true,
  // 不要显示启动服务器日志信息
  clientLogLevel: 'none',
  // 除了一些基本信息外，其他内容都不要显示
  quiet: true,
  // 如果出错了，不要全屏提示
  overlay: false,
  // 服务器代理，--> 解决开发环境跨域问题
  proxy: {
    // 一旦devServer(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': ''
```

```
      }
    }
  }
}
```

其中，跨域问题：同源策略中不同的协议、端口号、域名就会产生跨域。

正常的浏览器和服务器之间有跨域，但是服务器之间没有跨域。代码通过代理服务器运行，所以浏览器和代理服务器之间没有跨域，浏览器把请求发送到代理服务器上，代理服务器替你转发到另外一个服务器上，服务器之间没有跨域，所以请求成功。代理服务器再把接收到的响应响应给浏览器。这样就解决开发环境下的跨域问题。