webpack深入浅出

webpack是什么？

> webpack 是一个现代 JavaScript 应用程序的静态模块打包器，当 webpack 处理应用程序时，会递归构建一个依赖关系图，其中包含应用程序需要的每个模块，然后将这些模块打包成一个或多个 bundle。

webpack 的核心概念

- entry: 入口
- output: 输出
- loader: 模块转换器，用于把模块原内容按照需求转换成新内容
- 插件(plugins): 扩展插件，在webpack构建流程中的特定时机注入扩展逻辑来改变构建结果或做你想要做的事情

1.通常入门的步骤：

1. npm init初始化项目
2. npm i -D webpack webpack-cli
3. src中新增main.js文件可以打印一句话
4. 配置package.json命令

```
"scripts":{
    "build": "webpack src/main.js"
}
npm run build //此时会生成一个dist文件夹，并且包含main.js的文件
```

```
// webpack.config.js

const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const {CleanWebpackPlugin} = require('clean-webpack-plugin');
module.exports = {
    mode:'development', // 开发模式
    entry: path.resolve(__dirname,'../src/main.js'),    // 入口文件
    output: {
         filename: '[name].[hash:8].js',      // 打包后的文件名称已经经过hash编码了，每次都会不一样
        path: path.resolve(__dirname,'../dist')  // 打包后的目录
    },
    plugins:[
      new HtmlWebpackPlugin({
        template:path.resolve(__dirname,'../public/index.html')
      }),
      new CleanWebpackPlugin()
    ]
}
```

- HtmlWebpackPlugin:将webpack打包出的js文件动态插入到对应的html中

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>老yuan</title>
</head>
<body>
<script src="main.06eb5041.js"></script></body>
</html>
```

- 如果是多入口文件，则通过生成多个html-webpack-plugin实例来解决这个问题
- clean-webpack-plugin:每次执行npm run build 会发现dist文件夹里会残留上次打包的文件，这里我们推荐一个plugin来帮我们在打包输出前清空文件夹

2.引用CSS

> 我们的入口文件是js,所以我们在入口js中引入我们的css文件，同时我们也需要一些loade来解析我们的css文件或者less文件，如要安装如下：

```
npm i -D style-loader css-loader
npm i -D less less-loader
npm i -D postcss-loader autoprefixer  //为css添加浏览器前缀
// webpack.config.js
module.exports = {
    //...省略其他配置autoprefixer
    module:{
        rules:[{
            test:/\.less$/,
            use:['style-loader','css-loader',{
                loader:'postcss-loader',
                options:{
                    plugins:[require('autoprefixer')]
                }
            },'less-loader'] // 从右向左解析原则
        }]
    }
}
```

这时候我们发现css通过style标签的方式添加到了html文件中，但是如果样式文件很多，全部添加到html中，难免显得混乱。这时候我们想

3.拆分css

```
npm i -D mini-css-extract-plugin
* 这里需要说的细一点,上面我们所用到的mini-css-extract-plugin会将所有的css样式合并为一个css文件。如果你想拆分为一一对应的多

//配置文件如下：
// webpack.config.js

const path = require('path');
const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin')
let indexLess = new ExtractTextWebpackPlugin('index.less');
let indexCss = new ExtractTextWebpackPlugin('index.css');
module.exports = {
    module:{
      rules:[
        {
          test:/\.css$/,
          use: indexCss.extract({
            use: ['css-loader']
          })
        },
        {
          test:/\.less$/,
          use: indexLess.extract({
            use: ['css-loader','less-loader']
          })
        }
      ]
    },
    plugins:[
      indexLess,
      indexCss
    ]
}
```

4.打包 图片、字体、媒体、等文件

- file-loader就是将文件在进行一些处理后（主要是处理文件名和路径、解析文件url），并将文件移动到输出的目录中
- url-loader 一般与file-loader搭配使用，功能与 file-loader 类似，如果文件小于限制的大小。则会返回 base64 编码，否则使用 file-loader 将文件移动到输出的目录中

```
// webpack.config.js
module.exports = {
  // 省略其它配置 ...
  module: {
    rules: [
      // ...
      {
        test: /\.(jpe?g|png|gif)$/i, //图片文件
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 10240,
              fallback: {
                loader: 'file-loader',
                options: {
                    name: 'img/[name].[hash:8].[ext]'
                }
              }
            }
          }
        ]
      },
      {
        test: /\.(mp4|webm|ogg|mp3|wav|flac|aac)(\?.*)?$/, //媒体文件
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 10240,
              fallback: {
                loader: 'file-loader',
                options: {
                  name: 'media/[name].[hash:8].[ext]'
                }
              }
            }
          }
        ]
      },
      {
        test: /\.(woff2?|eot|ttf|otf)(\?.*)?$/i, // 字体
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 10240,
              fallback: {
                loader: 'file-loader',
                options: {
                  name: 'fonts/[name].[hash:8].[ext]'
```

```
              }
            }
          }
        }
      ]
    },
  ]
 }
}
```

5.用babel转义js文件

> 为了使我们的js代码兼容更多的环境我们需要安装依赖
> npm i -D babel-loader @babel/preset-env @babel/core

```
// webpack.config.js
module.exports = {
    // 省略其它配置 ...
    module:{
        rules:[
          {
            test:/\.js$/,
            use:{
              loader:'babel-loader',
              options:{
                presets:['@babel/preset-env']
              }
            },
            exclude:/node_modules/
          },
       ]
    }
}
```

> 上面的babel-loader只会将 ES6/7/8语法转换为ES5语法，但是对新api并不会转换 例如(promise、Generator、Set、Maps、Proxy等)
> 此时我们需要借助babel-polyfill来帮助我们转换

```
npm i @babel/polyfill
```

搭建vue开发环境

- 解析.vue文件

- 配置webpack-dev-server进行热更新

```
npm i -D vue-loader vue-template-compiler vue-style-loader
npm i -S vue
//vue-loader 用于解析.vue文件
//vue-template-compiler 用于编译模板
npm i -D webpack-dev-server //配置webpack-dev-server进行热更新
new vueLoaderPlugin(),
new Webpack.HotModuleReplacementPlugin()
```

```
// webpack.config.js
const path = require('path');
const {CleanWebpackPlugin} = require('clean-webpack-plugin') //打包输出前清空文件夹
const HtmlWebpackPlugin = require('html-webpack-plugin') //将打包出的js自动引入到对应的html中
const MiniCssExtractPlugin = require("mini-css-extract-plugin");//抽离css到单独的文件
const ExtractTextWebpackPlugin = require('extract-text-webpack-plugin')//拆分为一一对应的多个css文件
const vueLoaderPlugin = require('vue-loader/lib/plugin')//用于解决.vue文件
const Webpack = require('webpack')
module.exports = {
    mode:'development', // 开发模式
    entry: {
      main:path.resolve(__dirname,'../src/main.js'),
    },
    output: {
      filename: '[name].[hash:8].js',      // 打包后的文件名称
      path: path.resolve(__dirname,'../dist')  // 打包后的目录
    },
    module:{
      rules:[
        {
          test:/\.vue$/,
          use:['vue-loader']
        },
        {
          test:/\.js$/,
          use:{
            loader:'babel-loader',//使js兼容更多的环境，ES6/7/8转ES5等但是对新api并不会转换 例如(promise、Generator、
            options:{
              presets:[
                ['@babel/preset-env']
              ]
            }
          }
        },
        {
          test:/\.css$/,
          use: ['vue-style-loader','css-loader',{
            loader:'postcss-loader',
            options:{
              plugins:[require('autoprefixer')]
            }
          }]
        },
        {
          test:/\.less$/,
          use: ['vue-style-loader','css-loader',{
            loader:'postcss-loader',
            options:{
              plugins:[require('autoprefixer')]
            }
          },'less-loader']
```

```
        }
      ]
    },
    resolve:{
      alias:{
        'vue$':'vue/dist/vue.runtime.esm.js',
        ' @':path.resolve(__dirname,'../src')
      },
      extensions:['*','.js','.json','.vue']
    },
    devServer:{
      port:3000,
      hot:true,
      contentBase:'../dist'
    },
    plugins:[
      new CleanWebpackPlugin(),
      new HtmlWebpackPlugin({
        template:path.resolve(__dirname,'../public/index.html'),
        filename:'index.html'
      }),
      new vueLoaderPlugin(),
      new Webpack.HotModuleReplacementPlugin()//配置webpack-dev-server进行热更新
    ]
}
```

1.区分开发环境与生产环境

- 开发环境主要是热更新，不要压缩代码，完整的sourceMap
- 生产环境主要实现的是压缩代码、提取css文件、合理的sourceMap、分割代码
- webpack-merge 合并配置
- copy-webpack-plugin 拷贝静态资源
- optimize-css-assets-webpack-plugin 压缩css
- uglifyjs-webpack-plugin 压缩js
  webpack mode设置production的时候会自动压缩js代码。原则上不需要引入uglifyjs-webpack-plugin进行重复工作。但是optimize-css-assets-webpack-plugin压缩css的同时会破坏原有的js压缩，所以这里我们引入uglifyjs进行压缩

```
//webpack.dev.js
const Webpack = require('webpack')
const webpackConfig = require('./webpack.config.js')
const WebpackMerge = require('webpack-merge')
module.exports = WebpackMerge(webpackConfig,{
  mode:'development',
  devtool:'cheap-module-eval-source-map',
  devServer:{
    port:3000,
    hot:true,
    contentBase:'../dist'
  },
  plugins:[
    new Webpack.HotModuleReplacementPlugin()
  ]
})
```

```
//webpack.prod.js
const path = require('path')
const webpackConfig = require('./webpack.config.js')
const WebpackMerge = require('webpack-merge')
const CopyWebpackPlugin = require('copy-webpack-plugin')//将单个文件或整个目录复制到构建目录。
const OptimizeCssAssetsPlugin = require('optimize-css-assets-webpack-plugin')
const UglifyJsPlugin = require('uglifyjs-webpack-plugin')
module.exports = WebpackMerge(webpackConfig,{
  mode:'production',
  devtool:'cheap-module-source-map',
  plugins:[
    new CopyWebpackPlugin([{
      from:path.resolve(__dirname,'../public'),
      to:path.resolve(__dirname,'../dist')
    }]),
  ],
  optimization:{
    minimizer:[
      new UglifyJsPlugin({//压缩js
        cache:true,
        parallel:true,
        sourceMap:true
    }),
    new OptimizeCssAssetsPlugin({})
    ],
    splitChunks:{
      chunks:'all',
      cacheGroups:{
        libs: {
          name: "chunk-libs",
          test: /[\\/]node_modules[\\/]/,
          priority: 10,
          chunks: "initial" // 只打包初始时依赖的第三方
        }
      }
    }
  }
})
```

优化webpack配置

1.缩小文件的搜索范围(配置include exclude alias noParse extensions)

- alias: 当我们代码中出现 import 'vue'时， webpack会采用向上递归搜索的方式去node_modules 目录下找。为了减少搜索范围我们可以直接告诉webpack去哪个路径下查找。也就是别名(alias)的配

置。

- include exclude 同样配置include exclude也可以减少webpack loader的搜索转换时间。
- noParse 当我们代码中使用到import jq from 'jquery'时，webpack会去解析jq这个库是否有依赖其他的包。但是我们对类似jquery这类依赖库，一般会认为不会引用其他的包(特殊除外,自行判断)。增加noParse属性,告诉webpack不必解析，以此增加打包速度。
- extensions webpack会根据extensions定义的后缀查找文件(频率较高的文件类型优先写在前面)

```
module.exports = {
    module:{
        noParse:/jquery/,//不去解析jquery中的依赖库
        rules:[{
            test:/\.vue$/,
            loader: 'vue-loader',
            include:[path.resolve(__dirname,'src')],
            exclude: /node_modules/
        },
        {
            test:/\.(jep?g|png|gif)$/,
            use:{
                loader:'url-loader',
                include:[path.resolve(__dirname, 'src/assets/icons')],
                exclude: /node_modules/
            }
        }
    ]
    },
    resolve:{
       //例如：import Vue form 'vue',会查询'vue/dist/vue.runtime.esm.js'
        alias:{
            'vue$':'vue/dist/vue.runtime.esm.js',
            '@':path.resolve(__dirname, '../src'),
            'assets':resolve('src/assets'),
            'components': resolve('src/components')
        },
        extensions:['*','.js','.json','.vue']//扩展名
    }
}
```

2.使用HappyPack开启多进程Loader转换

> 在webpack构建过程中，实际上耗费时间大多数用在loader解析转换以及代码的压缩中。日常开发中我们需要使用Loader对js，css，图片，字体等文件做转换操作，并且转换的文件数据量也是非常大。由于js单线程的特性使得这些转换操作不能并发处理文件，而是需要一个个文件进行处理。HappyPack的基本原理是将这部分任务分解到多个子进程中去并行处理，子进程处理完成后把结果发送到主进程中，从而减少总的构建时间

3.使用webpack-parallel-uglify-plugin增强代码压缩

> npm i -D webpack-parallel-uglify-plugin

4.抽离第三方模块

> 对于开发项目中不经常会变更的静态依赖文件。类似于我们的elementUi、vue全家桶等等。因为很少会变更，所以我们不希望这些依赖要被集成到每一次的构建逻辑中去。 这样做的好处是每次更改我本地代码的文件的时候，webpack只需要打包我项目本身的文件代码，而不会再去编译第三方库。以后>只要我们不升级第三方包的时候，那么webpack就不会对这些库去打包，这样可以快速的提高打包的速度。

- 这里我们使用webpack内置的DllPlugin DllReferencePlugin进行抽离
- 在与webpack配置文件同级目录下新建webpack.dll.config.js 代码如下

```
// webpack.dll.config.js
const path = require("path");
const webpack = require("webpack");
module.exports = {
  // 你想要打包的模块的数组
  entry: {
    vendor: ['vue','element-ui']
  },
  output: {
    path: path.resolve(__dirname, 'static/js'), // 打包后文件输出的位置
    filename: '[name].dll.js',
    library: '[name]_library'
     // 这里需要和webpack.DllPlugin中的`name: '[name]_library',`保持一致。
  },
  plugins: [
    new webpack.DllPlugin({
      path: path.resolve(__dirname, '[name]-manifest.json'),
      name: '[name]_library',
      context: __dirname
    })
  ]
};
```

> 在package.json中配置如下命令
> "dll": "webpack --config build/webpack.dll.config.js"
> 执行 npm run dll
> 会发现生成了我们需要的集合第三地方 代码的vendor.dll.js 我们需要在html文件中手动引入这个js文件;这样如果我们没有更新第三方依赖包，就不必npm run dll。直接执行npm run dev npm run build的时候会发现我们的打包速度明显有所提升。因为我们已经通过dllPlugin将第三方依赖包抽离出来了。

5.配置缓存

> 我们每次执行构建都会把所有的文件都重复编译一遍，这样的重复工作是否可以被缓存下来呢，答案是可以的，目前大部分 loader 都提供了cache 配置项。比如在 babel-loader 中，可以通过设置 cacheDirectory 来开启缓存，babel-loader?cacheDirectory=true 就会将每次的编译结果写进硬盘文件（默认是在项目根目录下的node_modules/.cache/babel-loader目录内，当然你也可以自定义）

> 但如果 loader 不支持缓存呢？我们也有方法,我们可以通过cache-loader ，它所做的事情很简单，就是 babel-loader 开启 cache 后做的事情，将 loader 的编译结果写入硬盘缓存。再次构建会先比较一下，如果文件较之前的没有发生变化则会直接使用缓存。使用方法如官方 demo 所示，在一些性能开销较大的 loader 之前添加此 loader即可
> npm i -D cache-loader

6.优化打包文件的体积

引入webpack-bundle-analyzer分析打包后的文件

- webpack-bundle-analyzer将打包后的内容束展示为方便交互的直观树状图，让我们知道我们所构建包中真正引入的内容
- npm i -D webpack-bundle-analyzer
- windows请安装npm i -D cross-env
- "analyz": "cross-env NODE_ENV=production npm_config_report=true npm run build"
- 接下来npm run analyz浏览器会自动打开文件依赖图的网页

react16x项目实战分析

详细内容见代码webpack.config.js

vue项目分析

> 海豚脚手架 2.x 是基于 Vue Cli 3 的一套海豚项目代码生成工具, 其在剥离业务功能之后，保留了路由、菜单、面包屑、权限、多语言、换肤等解决方案, 用于快速搭建海豚体系项目的基本结构.

Vue CLI 是一个基于 Vue.js 进行快速开发的完整系统

Vue CLI 是一个基于 Vue.js 进行快速开发的完整系统，提供：

- 通过 @vue/cli 搭建交互式的项目脚手架。
- 通过 @vue/cli + @vue/cli-service-global 快速开始零配置原型开发。
- 一个运行时依赖 (@vue/cli-service)，该依赖：
- 可升级；

- 基于 webpack 构建，并带有合理的默认配置；
- 可以通过项目内的配置文件进行配置；
- 可以通过插件进行扩展。
- 一个丰富的官方插件集合，集成了前端生态中最好的工具。
- 一套完全图形化的创建和管理 Vue.js 项目的用户界面。

CLI 服务是构建于 webpack 和 webpack-dev-server 之上的。它包含了

- 加载其它 CLI 插件的核心服务；
- 一个针对绝大部分应用优化过的内部的 webpack 配置；
- 项目内部的 vue-cli-service 命令，提供 serve、build 和 inspect 命令。

```
D:\HIK-08\180\xcascade\xcascade-web\web-src>ue inspect --rule vue
'ue' 不是内部或外部命令，也不是可运行的程序
或批处理文件。

D:\HIK-08\180\xcascade\xcascade-web\web-src>vue inspect --rule vue
You are using Node v8.11.3.
Node.js 8.x has already reached end-of-life and will not be supported in future
major releases.
It's strongly recommended to use an active LTS version instead.
/* config.module.rule('vue') */
{
  test: /\.vue$/,
  use: [
    /* config.module.rule('vue').use('cache-loader') */
    {
      loader: 'cache-loader',
      options: {
        cacheDirectory: 'D:\\HIK-08\\180\\xcascade\\xcascade-web\\web-src\\node_
modules\\.cache\\vue-loader',
        cacheIdentifier: 'fda677b6'
      }
    },
    /* config.module.rule('vue').use('vue-loader') */
    {
      loader: 'vue-loader',
      options: {
        compilerOptions: {
          preserveWhitespace: false
        },
        cacheDirectory: 'D:\\HIK-08\\180\\xcascade\\xcascade-web\\web-src\\node_
modules\\.cache\\vue-loader',
        cacheIdentifier: 'fda677b6'
      }
    }
  ]
}

D:\HIK-08\180\xcascade\xcascade-web\web-src>_


        半:
```